Final Review

CSCE 740 - Lecture 26 - 12/02/2015

We Will Cover

- You have a final next Monday.
  - December 7th, 9:00 - 11:30 AM
- There is a practice exam on Moodle.
- Let’s go over it!

Question 1

Which of the following make sense as classes (rather than objects) in a class diagram?

1. Homework Assignment
2. Manton Matthews
3. Group 5's Assignment 5
4. Person

Which of the following coverage criteria **always** requires more test cases than the others?

1. Statement Coverage
2. Branch Coverage
3. Path Coverage
4. None of the above

Question 1

Natural language is typically used for requirements specifications for the following reason(s):

1. Ease of understanding
2. It is unambiguous
3. It is precise
4. It eliminates misunderstanding between the customer and supplier of the software

Which of the following are benefits of using checklists when writing requirements:

1. They can be used to refine the existing requirements.
2. They can convince the users to purchase your system.
3. They can help engineers derive missing requirements.
4. They can allow engineers to generate source code from the requirements.

Question 1 - True/False

- Requirements-based test cases help the writer clarify the requirements.
- In UML, a Class describes an Object.
- The goal of software testing is to remove defects from the goal.
- The use of global variables generally increases coupling.
- An oracle is needed to determine whether a test succeeded.
- Testing can be used to demonstrate that a program is free of faults.
- Most defects are introduced in the coding stage.
- Path coverage is generally impossible to achieve, but if we could, we would expose all faults in the program.

Question 2

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

Question 2 - Solution

Black-box testing treats the program as a machine that accepts input and issues output, with no visibility into its internal workings. Thus, tests are based on the requirements and specifications. You do not know what classes or methods are in the code, and you do now know what objects exist at runtime.

White-box involves testing the independent logic paths with full knowledge of the source code. However, you do not have full knowledge of the intended functionality (white box tests cannot look for unimplemented code).

Question 3

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.)

Question 3 - Solution

Many possible reasons for this, but two big ones include:

- 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

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.

Question 4 - 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

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

Question 5 - 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 faults such as divide-by-zero and null-pointer-dereferencing might not be caught.

Question 6

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.

Question 6 (2)

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:
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:
Input: station with empty container, valid trainID
Output: trainID in container

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

Question 6 (2) - Solution

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)*

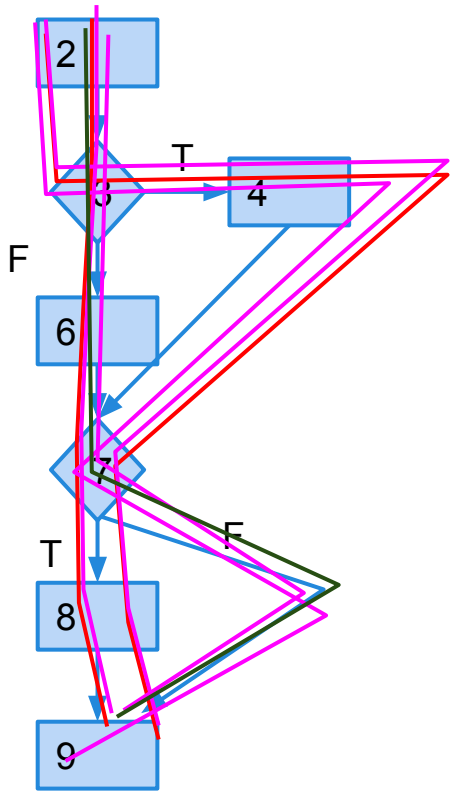
Delete	*ID in station / valid ID*	*ID no longer in list*
	ID not in station / valid ID	*no change (or error)*
	ID in station / invalid or malformed ID	*no change (or error)*
	ID not in station / invalid or malformed ID	*no change (or error)*
	empty list for station/ valid ID	*no change (or error)*
	empty list for station/ invalid or malformed ID	*no change (or error)*

Question 7

- Draw the control-flow graph for this method.
- Develop test input that will provide statement coverage.
- Develop test input that will provide branch coverage.
- Develop test input that will provide path coverage.

```
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;
}
```

Question 7 - 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. }
```

Statement:
(3,2,4), (2,3,4)

Branch:
(3,2,4), (3,4,1)

Path:
(4,2,5), (4,2,1), (2,3,4), (2,3,1)

Question 7 - Solution

- Modify the program to introduce a fault such that even path coverage could miss the fault.

Use (a >b+1) instead of (a>b) and the test input from the last slide: (4,2,5), (4,2,1), (2,3,4), (2,3,1) will not reveal the fault.

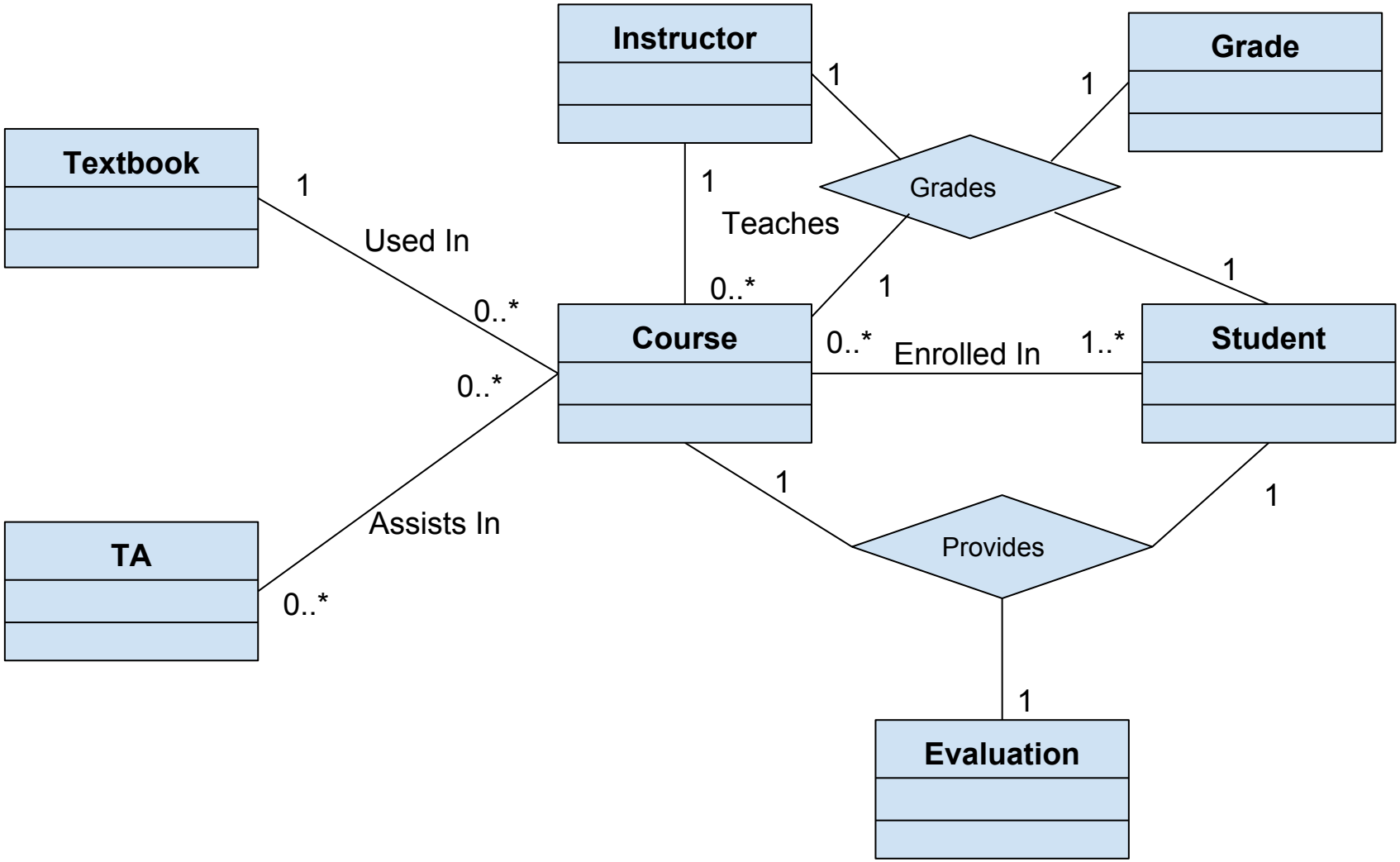
```
int findMax(int a, int b, int c) {
     int temp;
     if (a>b)
          temp=a;
     else
          temp=b;
     if (c>temp)
          temp = c;
     return temp;
}
```

Question 8

Students at USC 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). We do not offer classes with no students at all.
To allocate teaching effort, there is one instructor assigned to each class. Some instructors might not teach any class. Each class uses a textbook (a book that 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.

Develop the **class diagram** for the situation described above.

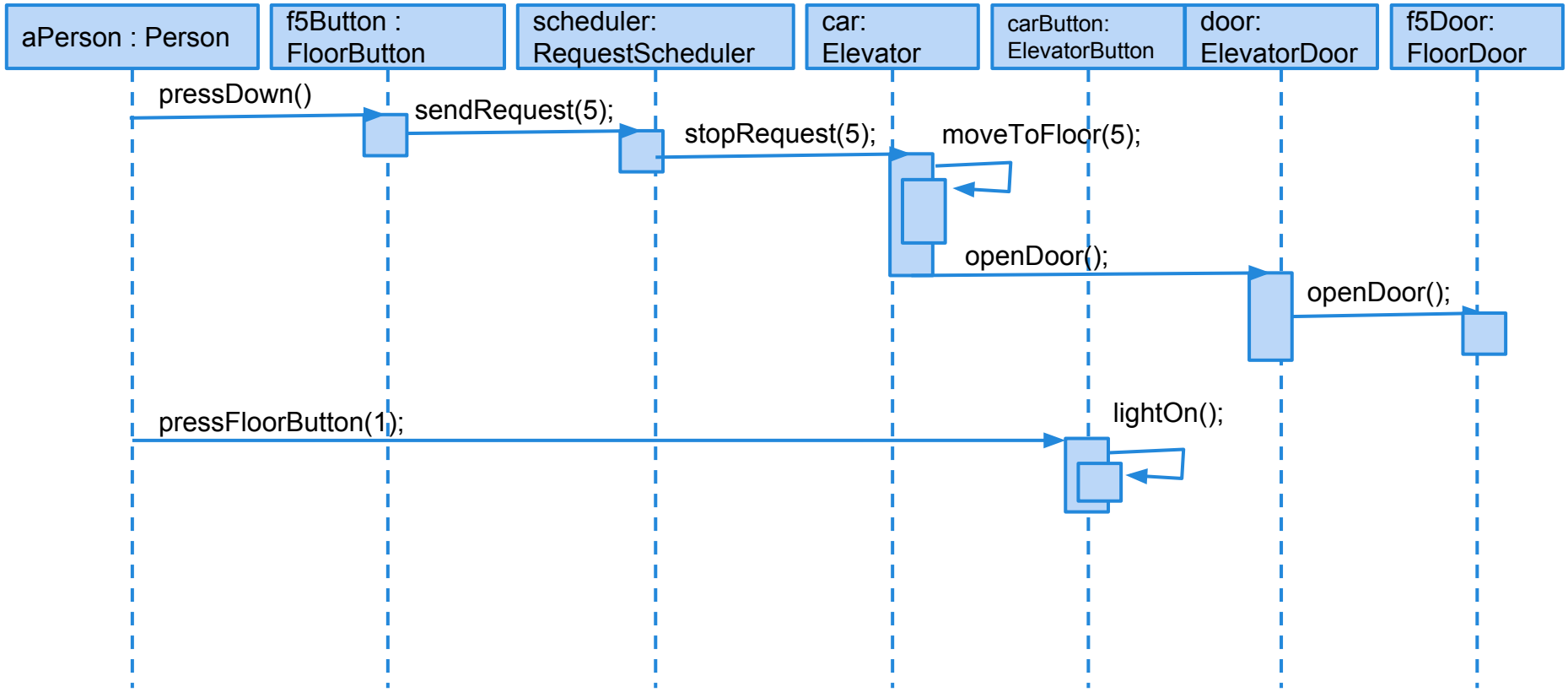
Question 8 - Solution



Question 9 - Scenario 1

Scenario 1 (Requesting a Ride Down):

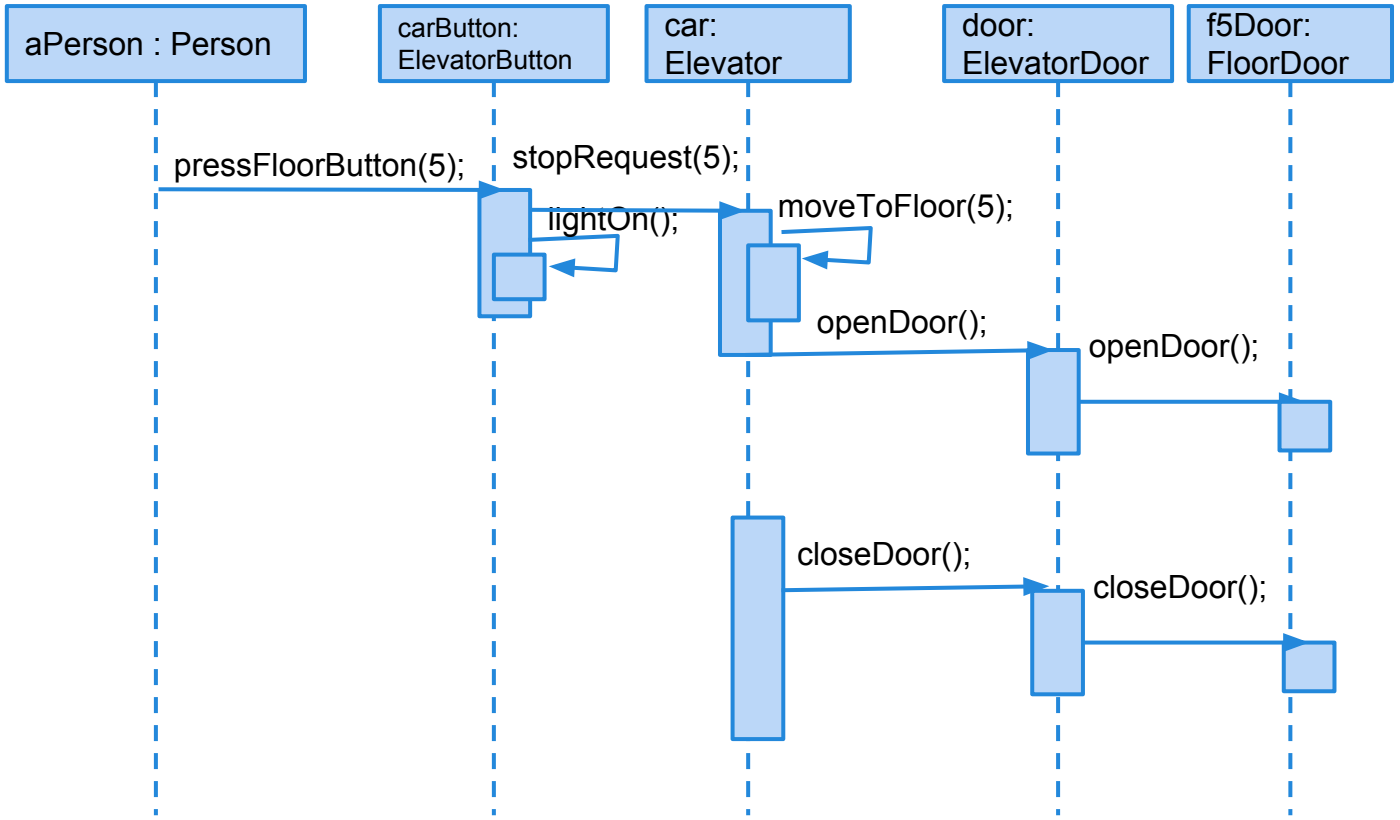
A person approaches the elevator on the fifth floor. She wants to go down so she presses the “down” button next to the elevators. She waits until an elevator arrives and the doors open. She enters the elevator and presses the elevator button for the ground floor (floor 1). The light next to the button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



Question 9 - Scenario 2

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.



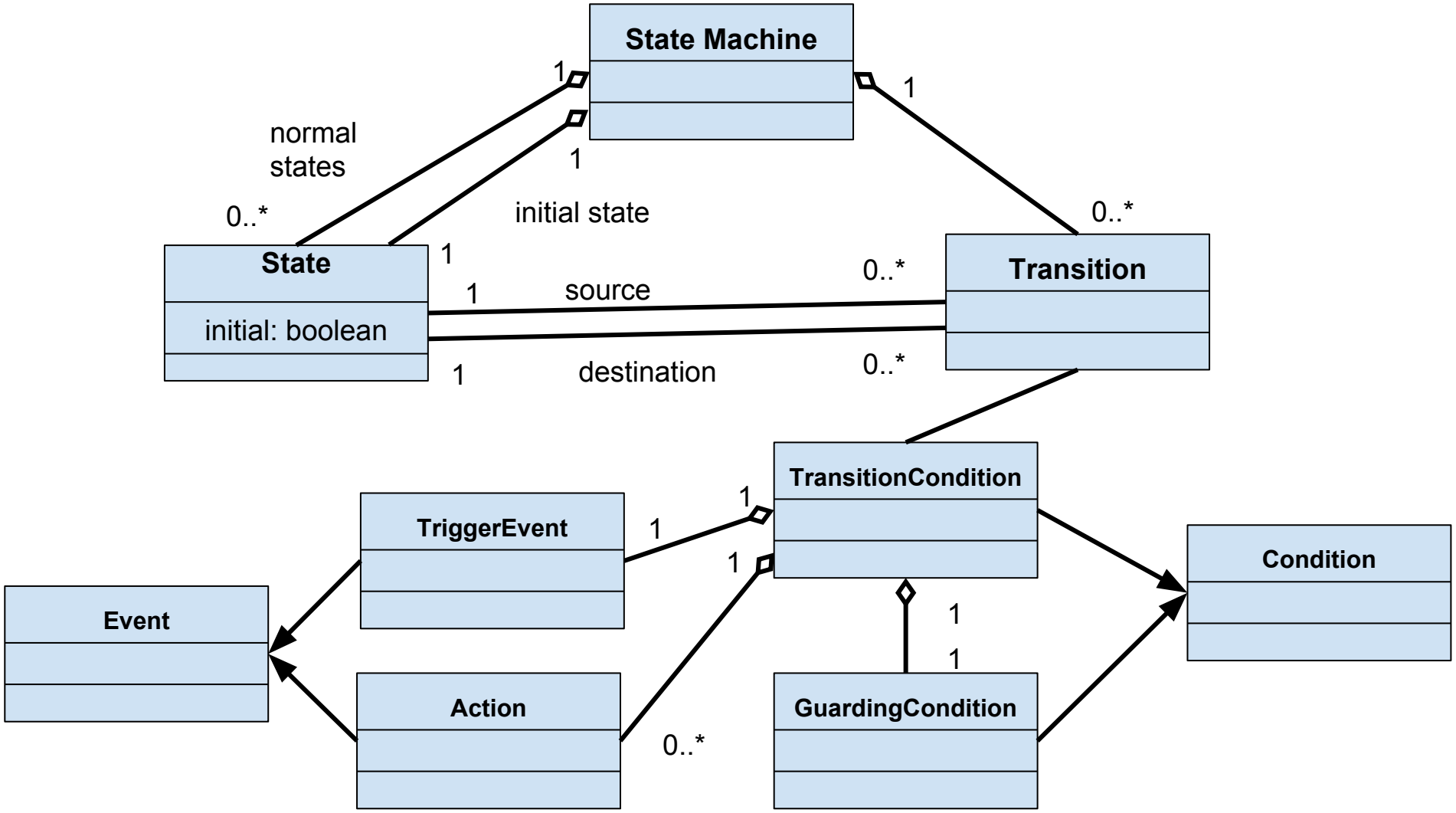
Question 10

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).

Develop the Class Diagram for this software.

Question 10 - Solution



Question 11

During development and maintenance, some organizations track “bad fixes” - a bug fix that introduces new faults in the software when the original fault is corrected. The ratio of bad fixes to “good fixes” can be measured.

- For example, the ratio of bad fixes to good fixes could be 1% (there is one bad fix for every 100 good fixes).
- In some troubled projects the bad fix ratio might be over 100%!

What effect will a bad fix ratio of >100% have on software quality? What do you think would be the main contributor to a very high bad fix ratio? Justify your answer.

Question 11 - Solution

A bad fix ratio over 100% means that more faults are being added than are being fixed. Software quality will deteriorate.

Poorly-structured software is likely to be the culprit. With low cohesion, high coupling, or hard-to-understand algorithms, it is hard to track down the real source of a fault (may only make a partial fix) and easy to introduce new faults (hard to determine the effect of a fix on other parts of the program).

Question 12

A class diagram in UML is generally used during design, but can also be a useful tool in the requirements elicitation stage of a software development project. Discuss briefly how class diagrams might be used in this stage of development.

Question 12 - Solution

UML class diagrams are useful for visualizing entities and their relationships at any level of abstraction.

- Relationships between data items in the problem domain.
- Clarify the relationships between concepts (credit cards and customers, students and professors, students and grades, etc).
- Model can serve as the foundation for natural language requirements by structuring the problem domain.
  - Cleaner and easier to write specifications because we can relate to the diagram.

Question 13

Why is it so important to include boundary values in your black-box test-data?

- Make sure your answer includes a brief description of what a boundary value is.

Question 13 - Solution

Boundary values are the inputs that are on or close to the boundaries between the input equivalence partitions as well as special values we know are tricky to handle correctly.

- We know from experience that programmers make mistakes with boundary values.
- Thus we should include test cases to see if these cases are handled correctly.
  - Include values such as zero, very large, very small, empty list, max long list, etc.

Question 14

When performing reliability (statistical) testing, an operational profile is absolutely essential for the test-data selection. Why? What is the effect of an inaccurate operational profile?

Question 14 - Solution

Since the reliability metrics are designed to measure the reliability of a system under normal operating conditions it is essential to know what “normal” operating conditions are.

- This is what the operational profile is supposed to capture.
- If the reliability is assessed using a profile that does not accurately capture the real operating conditions, the measure is meaningless.

Question 15

Consider the use case “Submit Assignment Solution” for a homework assignment and collection system, where a student is asked to complete and submit an “assignment submission form” where they give various pieces of information including name, course number, session number, and assignment number. The student then attaches her solution and presses the submit button. Before the assignment solution is accepted, the student-provided information is validated against the class roster.

List two (2) exceptions you think can occur for this use case. For each of the exceptions, also specify what you think the proper response from HACS should be.

Question 15 - Solution

Erroneous information: A student enters the wrong class or session number or the wrong assignment number. In both cases a warning message and possibly a helpful message along the lines of “You are not enrolled in CSCE 747 for the Fall of 2015, you are enrolled in: CSCE 740.”

No assignment attached: All information is correct, but no file to upload has been selected. An error message pointing this out would be an appropriate response.

Question 16

Early in the class we discussed several Fundamental Principles of software engineering. One was **Separation of Concerns**. This principle is visible in the fact that in a comprehensive modeling framework, such as UML, there are several different diagrams supported.

Please briefly explain how the principle of Separation of Concerns is manifested in fact that UML contains many different diagram types.

Question 16 - Solution

Each separate concern can be realized in a diagram suitable for modeling that concern.

- A class diagram for structure.
- Sequence diagrams to illustrate the order of messages or events.
- State machines for the reactive behavior.
- Use cases to examine the goals of actors.

Any other questions?

Thank you for a great semester!