CSCE 247 - Practice Midterm
Name:

This is a 75-minute exam. On all essay type questions, you will receive points based on the quality of the answer - not the quantity.

Make an effort to write legibly. Illegible answers will not be graded and awarded 0 points.

There are a total of 10 questions and 100 points available on the test.

Question 1

Briefly explain why a software system must change or become progressively less useful.

Suggested Solution:

The world is constantly changing, if the software does not change with the world, it will eventually be completely out of date and useless. The changes in the world might be organizational (the users' needs have changed and the software must adapt), infrastructure (the hardware and operating systems are changing and the software must be adapted), the computational model changes (formerly stand alone systems must now be networked), etc. etc.

Question 2

The following requirements are unclear and ambiguous. Explain why, and then rewrite the statements so that they can be objectively evaluated.

a. The response time should be minimized.

b. The alarm should be raised quickly after a high fuel level has been detected.

Suggested Solution:

a: "should" != shall. What does minimized mean? Response time to what?

"The system shall respond to a user request within ten seconds."

b. Quickly? Is "high fuel level" a boolean condition or a specific quantity?

"The alarm shall be raised within 5 seconds of the fuel level reaching 10 cm."

Question 3

What are the properties of a “good” individual requirement (as opposed to properties of a requirements document)? Mention four (4) properties of a “good” requirement and explain what they mean. For each, give an example of a requirement that violates the property.

Suggested solutions:

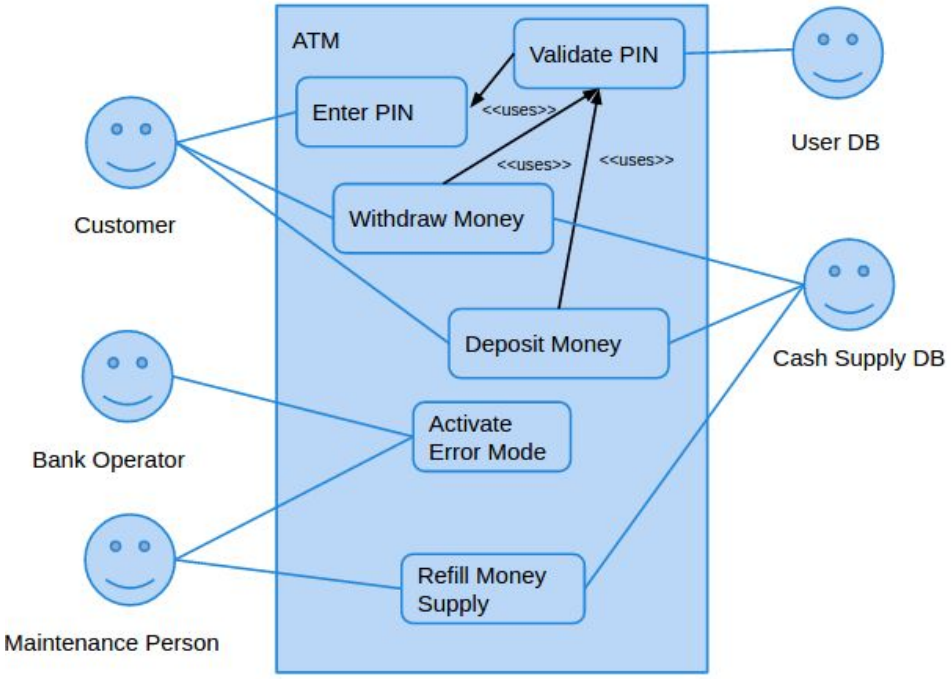
- *Unambiguous—There is only one way of interpreting the requirement.*
- *Correct—It captures the stakeholders true needs.*
- *Testable—It must be written so that it can be objectively used as an oracle during testing process.*
- *Traceable—We must be able to trace the requirement to its definition, environmental assumptions, business case, use-case, etc.*
- *Feasible—We must be able to actually implement this requirement.*
- *Prioritized—We must prioritize the requirements so that we can decide wisely which ones to do first when we (inevitably) run out of time.*
- *Complete—Make sure we have covered all aspects of this particular requirement.*
- *Consistent—A requirement cannot be self-contradictory.*
- *Precise, unambiguous, and clear*
- *Relevant*
- *Free of design detail*

Examples of all of these are in the slides, but also be able to come up with your own.

Question 4

You are all familiar with an Automatic Teller Machine (cash machine). Please draw a use-case diagram with at least three (3) use-cases and the actors involved in these use-cases.

Suggested Solution:
Something along the lines of the following will be accepted:



Question 5

Pick one of the use-cases from the previous question and write down the typical scenario for this use-case.

Suggested Solution:
See examples from the webpage and lecture slides on the ATM.

Question 6

Briefly explain the key differences between centralized and distributed version control. What are the strengths and weaknesses of each approach?

Suggested Solution:
The main difference between centralized and distributed version control is the number of repositories. In centralized version control, there is just one repository, and in distributed version control, there are multiple repositories. In centralized VCS, the repository is stored on a central server. In decentralized VCS, all working copies are paired with a copy of the repository. A central repository is used to coordinate and synchronize the local repositories.

Decentralized VCS has become standard. A centralized VCS has a single point of failure, which is the remote central VCS instance. If this instance is lost, it can cause productivity and data loss, and it will need to be replaced with another copy of the source code. If it temporarily becomes unavailable, it will prevent developers from pushing, merging or rolling-back code. A distributed model architecture avoids these pitfalls by keeping a full copy of the source code at each VCS instance. If any of the previously mentioned centralized failure scenarios happen within the distributed model, a new VCS instance can be swapped in to lead development mitigating any serious drop in productivity. If the central server is lost, any copy could become the "central" source of truth.

The main trade-off is a slightly more complicated workflow. Centralized VCS only requires staging, commits, and updates. Decentralized VCS adds an additional pull step before an update, and an additional push step after a commit. Decentralized VCS can also require more work to resolve conflicts if any of the repositories go out-of-sync.

Question 7

Part 1:
Explain the difference between *validation* and *verification*.

Part 2:
Validation is generally considered harder. Why?

Suggested Solution:

Part 1:
Validation is concerned with building the product the customer wants. It attempts to answer the question "Are we building the right product?"

Verification is concerned with if we are building a product with the properties we said it was going to have (does the product match the requirements specification?). It is trying to answer, "Are we building the product right?"

Part 2:

Validation is harder because it is subjective - up to the whims of the customers. You can "measure" verification, build evidence that the product works under the conditions that you have set. In verification, you have the specification to compare the product to. In validation, you have to make the customers happy. Customers change their minds, they leave out details, and - if there are multiple stakeholders - they may have conflicting desires. This makes validation harder.

Question 8

The airport connection check is part of a travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary.

validConnection(Flight arrivingFlight, Flight departingFlight) returns ValidityCode.

A Flight is a data structure consisting of:

- A unique identifying flight code (string, three characters followed by four numbers).
- The originating airport code (three character string).
- The scheduled departure time (in universal time).
- The destination airport code (three character string).
- The scheduled arrival time (in universal time).

There is also a flight database, where each record contains:

- Three-letter airport code (three character string).
- Airport country (two character string).
- Minimum connection time (integer, minimum number of minutes that must be allowed for flight connections).

ValidityCode is an integer with value 0 for OK, 1 for invalid airport code, 2 for a connection that is too short, 3 for flights that do not connect (arrivingFlight does not land in the same location as departingFlight), or 4 for any other errors (malformed input or any other unexpected errors).

In order to design requirements-based test cases, perform input partitioning and derive representative values of the parameters from this specification for the validConnection function.

Suggested Solution:
The following are the essential input partitions. Others are possible.

Recall the lectures on requirements-based testing. The approximate process of taking a requirement and writing tests is to:

1. *Refine the requirement so that it is testable.*
2. *As a requirement is just a property of the software, you usually can't directly "test the requirement." Instead, you need to use a function of the software and show that the requirement holds during that execution. So, the next step is to identify the independently testable features of the software.*
3. *Now that you have those, you need to look at the parameters and figure out which inputs to pass in. You cannot exhaustively test a function, there are too many possible parameters.* ***So, instead, you partition the input domain into representative regions.*** *If you try inputs from each of these areas, you are more likely to trigger a fault than through random testing alone. We discussed some methods of doing so during Lecture 8.*

4. *Once you have the inputs partitioned, you can form abstract test cases for which you can transform into actual test cases by coming up with concrete input values from the identified partitions.*

In this question, you have been given a testable feature, so you have been asked to perform the activity from Step #3 above - given the parameters, partition the inputs into representative regions. This function has two explicit inputs - an arriving flight and a departing flight - and an implicit input - an airport connection database.

A flight is a complex data structure containing several fields, so for each field, you need to partition the inputs. Remember that the function's parameters may influence each other (testing this function requires considering both the arriving and departing flight's field values as well as what is in the database), so that may influence the partitions.

<u>Parameter: Arriving flight</u>

Flight code:
malformed
not in database
valid

Originating airport code:
malformed
not in database
valid city

Scheduled departure time:
syntactically malformed
out of legal range
legal

Destination airport (transfer airport - where connection takes place):
malformed
not in database
valid city

Scheduled arrival time (tA):
syntactically malformed
out of legal range
legal

<u>Parameter: Departing flight</u>

Flight code:
malformed
not in database
valid

Originating airport code:
malformed
not in database
differs from transfer airport
same as transfer airport

Scheduled departure time:
syntactically malformed
out of legal range
before arriving flight time (tA)
between tA and tA + minimum connection time (CT)
equal to tA + CT
greater than tA + CT

Destination airport code:
malformed
not in database
valid city

Scheduled arrival time:
syntactically malformed
out of legal range
legal

Parameter: Database record

This parameter refers to the database time record corresponding to the transfer airport.

Airport code:
malformed
not found in database
valid

Airport country:
malformed
invalid
valid

Minimum connection time:
not found in database
invalid
valid

Question 9

You are writing test cases for the following method:

```
public double max(double a, double b);
```

This method returns the largest of two integers.

Using the JUnit notation, write three test cases for this method.

Suggested Solution:

```
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("a should be larger", a > b);
    assertEquals(expected, actual);
  }

@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals(a,b);
    assertEquals(expected, actual);
  }

@Test
  public void bothNegative() {
    double a = -2.0;
    double b = -1.0;
    double expected = -1.0;
    double actual = max(a,b);
    assertTrue("answer should be negative", actual < 0);
    assertEquals(expected, actual);
  }
```

Advice: Review assertions, and try to think of a variety that could be applied.

Question 10

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.