Dependability

1. What is the time ordering (in terms of when a mistake is made or occurs in the running system) of the following: (1) an error, (2) a fault, and (3) a failure?

- 1,2,3
- 1,3,2
- 2,1,3
- 2,3,1
- 3,1,2
- 3,2,1

2. Testing helps with which kind of dependability criteria?

- fault avoidance
- fault tolerance
- error removal
- fault forecasting

3. Availability is the same as reliability.

- True
- False

4. A correct system (with respect to its requirements) will be safe.

- True
- False

5. A correct system will be reliable.

- True
- False

6. Robust systems are reliable.

- True
- False

7. Safe systems are robust.

- True
- False

Test Principle- Where

1. Why do floating point numbers sometimes lead to erroneous code?

- Arithmetic on floating point numbers is often approximate, so it can introduce errors
- Floating point numbers have values that are not actually numbers, such as infinity and NaN (not a number), that can cause computations to behave strangely.
- In Java, floating point numbers cannot represent numbers as large as ints and longs
- As floating point computations are approximate, equality comparisons fail after computations that would succeed when using real numbers

2. Why do relational boundaries sometimes lead to erroneous code?

- Programmers often make 'off-by-one' errors
- Determining strict limits on ranges is difficult in requirements engineering
- Relational boundaries define points of discontinuity for programs
- The Java compiler sometimes optimizes these expressions incorrectly

3. Why do casts sometimes lead to erroneous code?

- When converting an integer to a smaller bit length (e.g. long to int), the value may be truncated.
- When converting to a larger bit length (e.g., int to long), the value may change from positive to negative.
- When converting from signed to unsigned types (e.g., int to byte), negative numbers can't be represented.
- When converting from a double to int, the value is truncated to a whole number.
- When converting from an int to double, the integer value may not be exactly representable.

Test Principle - How

1. Why are we often able to test more rigorously at the unit level rather than the system level?

- There are always fewer dependencies for unit tests than there are for system tests.
- The tests tend to run faster, so we can run more of them.
- We can usually see more of the internal state at the unit level so we can build stronger oracles.

2. When we say that we want redundant verification, what do we mean?

- We want to re-run proofs of multiple subsystems to make sure we get the same answer.
- We want several different verification techniques checking the same program or subsystem.
- We want lots of similar or same tests.

3. Which of the following are good ways to work with developers to reduce systematic errors?

- Create tools to test/verify specific kinds of common errors.
- Create libraries or utility functions to encapsulate operations that developers tend to get wrong.
- Create checklists for developers based on the most common errors seen in test.
- E-mail developers lists of bugs that have been found over the last few months.
- Use languages / IDEs that eliminate certain classes of errors by compile-time checks.

4. We state that programs, as well as tests, can be flakey. What does it mean for a program to be flakey?

- Sometimes the program creates different outputs for the same input.
- Sometimes the program ends up in different internal states for the same input.
- Given the same inputs, sometimes the program fails a test and other times it does not.

5. Why is observability an important issue in testing?

- If a program error is transient (it happens and is masked out by other code), you might not be able to "see" it and the test may pass.
- Often programs are stateful - that is, a test may trigger an error, but it only becomes visible as an output if a long sequence of steps are executed, whereas it might be immediately visible by examining internal state.
- Developers often want to hide important information related to their code because they think it leads to job security.
- Because developers like to use the 'Observer' pattern and this can often lead to errors because it decouples the domain object and the GUI presentation.

The V-model

1. The V-Model is:

- A testing framework for running tests at each step during software development.
- A software development model that pairs different stages of software development with the appropriate testing procedure. These tests are later used when checking the verification of each phase of the software.
- All of the above

2. The system design is tested using:

- Unit Testing.
- Module Testing.
- Validation Testing.
- Verification Testing.

3. The specification of the system is tested using:

- Unit Testing
- Validation Testing.
- Verification Testing.
- System Integration Testing.

Validation and Verification in the “V-Model”

1. The difference between validation and verification is:

- Validation confirms that we are building the right product while verification confirms that we are building the product right.
- Validation is planned during the user requirement stage in the V-model while verification is planned during the user specification stage.
- None of the above
- A and B

2. With respect to the V-model, a system that has passed verification testing means:

- The system is done and no further testing is needed.
- The system has passed all tests from unit testing through verification testing and is now ready for validation testing.
- The system needs to be tested all the way from unit testing up through verification testing, before being claimed correct.

3. With respect to the V-model, a valid system implies a verified system:

- True.
- False.

Structural Testing

1. Structural testing is:

- Black-box testing
- Blue-box testing
- White-box testing

2. What is not true about structural testing?

- In most cases, it is infeasible to reach 100% structural coverage.
- Structural coverage criteria use the structure of the code to measure the adequacy of tests.
- Structural testing is white-box testing.
- The goal of testing is to achieve 100% structural coverage to ensure the absence of bugs.

Mutation Testing

1. Which of the following is **not true** about mutation testing?

- With mutation testing, you can know how much of the code structure you covered.
- The mutation adequacy score tells you the quality of tests; the higher the score, the better the quality of test cases.
- A mutant is killed when there exists one or more tests that can differentiate between the output of the mutant and the output of the original program.

2. Which of the following are **not true** about mutation testing?

- The mutation operator introduces a syntactic change to the program so that the mutant cannot be compiled. *{ The mutation operator does introduce a syntactic change to the original program, but the resulting mutant must be syntactically correct and compilable in order to be used in mutation testing.}*
- You only create one mutant for mutation testing. { *You need to create more than one mutant. You need anywhere from a few tens to a few hundred or more to assess the quality of your test suite. The quality of your test suite is assessed by the number of mutants your tests kill.*}
- A mutant and the original program are syntactically different.

3. Jenny wrote a program and created 10 test cases. She also created 50 mutants from the original program to measure the adequacy of her test inputs using mutation testing. After running all of the test cases against each of the mutants, Jenny found that all of the test cases had passed on all of the mutants. In this situation, what is the most appropriate step she should take next?

- She should write more test cases.
- She can stop testing because all the tests have passed.
- She should create more mutants.

That all the tests have passed means that none of the tests succeed in detecting the changes (deviations from the original program, thus faults) introduced by mutation. It is likely that the ten test cases were not good enough in detecting those mutants.