1. Design and prototype walk-throughs with users are examples of:

Validation

1. Unit tests mapping requirements (or User Stories in Agile development) to class behaviour are examples of:

Verification

1. User interviews to see if requirements/User Stories match the user's expectations of how the system will perform are examples of:

Validation

1. Refactoring is a technique that modifies the structure of software to improve the design (or some non-functional attribute such as performance) without changing the functionality of the software.

Regression testing after refactoring the system to determine whether it behaves the same as the non-refactored version is an example of:

Verification

1. Which of the following are strengths of testing?

It checks the whole system, including software that you didn't write

It documents system behavior.

1. Is it always possible to prove whether any program meets its requirements in all cases, given enough effort?

NO(It has been shown to be theoretically undecidable to even determine whether an arbitrary program completes its execution (halts), let alone if it produces the correct value. This is called the ["Halting Problem"](https://en.wikipedia.org/wiki/Halting_problem) and was explored by Alan Turing.)

1. Is testing a (primarily) optimistic or pessimistic verification technique?

Optimistic(Testing only checks certain program paths, so it is (usually) optimistic. Testing in very rare cases can be pessimistic, but only if the test outcome is incorrectly defined.)

1. We said that software testing is different than testing in many disciplines because software is discontinuous. In mathematics, continuity means that if you know the value of a function at one point, you can make claims about its value at points nearby. Why does this matter for testing?

Continuous systems are simpler to test because you can often extrapolate test results from one point to points 'nearby'.

1. We said that the Zune needed tests for boundary conditions. These are values that check that the boundaries of arithmetic or relational expressions (like ==, !=, <, >, <=, >=) are tested. What are examples of boundary condition tests for the following function?

int flipSome(int A , int N, int X)

{

  int i=0;

  while (i<N)

  {

    if (A<X)

      A = -A;

    i++;

  }

  return(1);

}

A = 1, X = 10, N = 1(This is a boundary value for the expression i < N. It can test whether the correct expression should be i < N or i <= N once i = 1)

A = 5, X = 5, N = 5(This test checks the boundary A < X)

Test coverage is defined as a metric in Software Testing that measures the amount of testing performed by a set of test. It will include gathering information about which parts of a program are executed when running the test suite to determine which branches of conditional statements have been taken.

1. An automated oracle is:

A program that automatically matches the actual program output to the expected output of the test case.

(An oracle can be a human or a program. The function of an oracle is to check that the output of the program under test matches the expected output of the test case.)

1. The **setup** step for a test case is important to establish a specific state for the program to run the test.

T/F

1. Tasks that can be part of the **tear down** phase are

Remove data you added after testing is done

Close connection after testing is done

1. **Assessment** is the phase where:

Software output is checked by an oracle.

(Part of the assessment is to compare the output of the software under test with the expected output. This can be done manually, with a human checking that the result meets his/her expectations, or automatically, with an oracle.)

1. Testing execution frameworks (e.g., JUnit) are important because:
2. They allow automated checks against an oracle (e.g., Do the outputs of the program match expectations?) to determine whether or not a test passes.

ii. They run all test cases and provide feedback on which test cases passed and which failed.

1. They allow programmers to unit test each method.
2. To test a main method you need to:

Redirect input to be entered by the test case.

1. Given a program that calculates the surface area of sphere A using the equation below where r is the radius of the sphere and is provided by the user, which of the following are valid test cases?

$$A = 4 \pi r^2$$

input r = 1/2, expected output = 3.14(Valid Test case)

input r = -1/2, expected output = invalid input(Invalid Test case)

input r = 0, expected output = invalid input( This is a good test case: checks boundary condition)

input r = 30, expected output = 11309.73(Valid)

input r = -1, expected output = invalid input(Valid)

1. Redirection of system input and output for the Coffee Maker example is done in order to test:

The main method.

1. Testing is all about corner cases.

False (While it is important to test corner cases, we must also check that the system functions correctly under normal use. Test cases that check the normal use of the system (called nominal test cases) are very important because they check the code that most users will regularly execute.)

1. The strategy of writing test cases before implementation is:

A good idea. Test cases should be built based on the expected output and should not be influenced by the implementation.

1. When you have a new test case:

All test cases need to be run including the new one.

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?

2,1,3

1. Testing helps with which kind of dependability criteria?

Error Removal

1. Availability is the same as reliability.

False(availability says the system is running, reliability says that it is doing the right thing.)

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

False (A correct system will be safe if the requirements are adequate to ensure safety, but this is not a given.)

1. A correct system will be reliable.

True (Reliability is defined in terms of meeting requirements; by definition, a correct system is reliable.)

1. Robust systems are reliable.

False (the system may not crash (robust), but it still may not do the right thing (reliable))

1. Safe systems are robust.

False (For example, if a car in a driveway does not start, it is safe, but not very robust.)

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.

As floating point computations are approximate, equality comparisons fail after computations that would succeed when using real numbers

1. 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

1. 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.

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

We can usually see more of the internal state at the unit level so we can build stronger oracles.

The tests tend to run faster, so we can run more of them.

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

We want several different verification techniques checking the same program or subsystem.

1. 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.

Use languages / IDEs that eliminate certain classes of errors by compile-time checks.

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

Given the same inputs, sometimes the program fails a test and other times it does not.

1. Why is observability an important issue in testing?

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.

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.

1. The V-Model is:

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.

1. The system design is tested using:

Module Testing

1. The specification of the system is tested using:

Verification testing

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.

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

The system has passed all tests from unit testing through verification testing and is now ready for validation testing.

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

True

1. Structural testing is:

White-box testing

1. What is **not true** about structural testing?

The goal of testing is to achieve 100% structural coverage to ensure the absence of bugs.

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.

1. 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.

You only create one mutant for mutation testing.

1. 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.( 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.)