**Software fault:** a static defect in the software. **Software failure:** external, incorrect behavior with respect to the requirements or other description of the expected behavior. **Software error**: an incorrect internal state that is the manifestation of some fault. **Validation**: the process of evaluating software at the end of software development to ensure compliance with intended usage. **Verification**: the process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase – independent V&V. **Process Maturity**: (Level 0): there’s no difference between testing and debugging (Level 1): the purpose of testing is to show correctness (Level 2): the purpose of testing is to show that the software doesn’t work (Level 3): the purpose of testing is not to prove anything specific, but to reduce the risk of using the software (Level 4): testing is a mental discipline that helps all IT professionals develop higher quality software. **Testing:** exercising software and observing/evaluating its execution. **Test failure:** execution of a test that results in an observable incorrect result (failure). **Debugging:** process of finding a fault given a failure. **Fault/Failure Model:** (Reachability): test must execute the faulty code (Infection): execution must create an incorrect state of the program – error (Propagation): the error has to propagate to the output or final state of program (Revealability): the tester has to observe the incorrect output/final state. **Top Down Testing Acceptance testing:** satisfies requirements/ user needs, must involve users with domain knowledge. **Top Down Testing System testing:** evaluates overall system behavior; assumes individual parts already tested. **Integration testing:** evaluates subsystem design; puts modules together. **Module testing**: evaluates design of individual modules. **Unit testing**: tests individual functions/units with respect to implementation**. Inter-class testing:** test multiple classes together. **Regression testing:** Performed throughout program maintenance; each time a program is modified; ensures that the previously working code has not broken (regressed); expensive. **Intra-class testing**: test an entire class as sequences of calls. **Inter-method testing:** test pairs of methods in same class**. Intra-method testing:** test each method individually. **Specification**: derive tests from the specification or external description; often used at system level. **Structural:** derive tests from code; often used at unit level. **JaCoCo**: uses Junit for testing; instruments classes to provide code coverage. **Model Driven Test Design:** test design test automation, test execution, test evaluation. **Automation:** frees up engineer time; allows tests to be run and re-run; helps eliminate errors of omission. **Revenue:** contribute directly to the solution of the problem. **Excise:** don’t contribute directly to the solution. **Testability**: degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine if these criteria have been met. **Software observability**: how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other components. **Software controllability**: how easy its to provide a program with the needed inputs. **Prefix values:** inputs necessary to put software into an appropriate state to receive test case values; initialization. **Postfix values**: inputs that need to be sent after the test runs; finalization. **Verification values:** necessary to see the results of the test case. **Exit values:** values/commands needed to terminate the program or return it to a stable state. **RIPR Model:** tests are designed to look for a fault in particular location; prefix values provide R; test case values achieve I; postfix values achieve P; expected results reveal R. **Software Life Cycle:** requirements gathering and analysis; architecture design and specification; coding and testing; delivery and deployment; maintenance and evolution; retirement. **Test requirement:** a specific element of a software artifact that a test case must satisfy or cover. **Coverage criterion:** collection of rules that impose test requirements on a test set. **Coverage**: given a set of test requirements, TR for a coverage criterion C, a test set satisfies C iff, for every test requirement, tr, in TR, at least one test, t, in TR exists that satisfies tr. **Minimal Test Set**: given TR and a test set T that satisfies all test requirements, T is minimal if removing any single test from T will cause T to no longer satisfy any all test requirements. **Minimum Test Set:** Given TR and a test set T that satisfies all test requirements, T is minimum if there is no smaller set of test that also satisfy all test requirements. **Evosuite**: uses a search algorithm to find test cases that cover as much code as possible; creates Junit test cases; builds oracles from the code itself. **Fuzzing:** used to detect program crashes/memory leaks for security. **Event flow model goal (GUI):** represent all possible event interactions on the GUI; will allow a model that can be traversed for test case generation. **Graph model (GUI):** vertex represents an event; an edge from vertex shows than an event y can be performed immediately after event x. **Event Flow Graph**: represents all possible event sequences that can be executed in a dialogue. **Oracle information**: expected output. **Oracle procedure**: compares expected with actual. **Input Space Partitioning**: logically divide the input space into logical partitions of the input; independent of the RIPR model – focuses only on the input domain. **Modeling the input domain**: (1) identify testable functions (2) find all the parameters (3) model the input domain; scoped by the parameters; partitioning characteristics into blocks and values (4) apply a test criterion to choose combinations of values (5) refine the combinations of blocks into test inputs. **Interface based approach:** develops characteristics directly from individual input parameters. **Functionality based approach:** develops characteristics from a behavioral view of the program under test.

**RIPR model in relation to software testing:** In order to successfully find faults we have to have (1) reachability, (2) infection, (3) propagation (4) and revealability. Our test case has to reach the faulty line of code. It then has to change some state of the program and propagate out of there to be detected. Last, we need an oracle that can determine that the final output of the program is incorrect. We saw this in assignment 2 where it was possible to reach the faulty branch of code (R) and infect it (I) by changing the state inside of the if statement, but it did not propagate (p) since the next if statement was a duplicate check, hence the fault was not revealed by any of the unit tests. **Differentiate fault, error, and failure:** Fault – incorrect syntax or logic of a program Error – incorrect state of the program Failure – when an error propagates and is detected. **In Junit, use assertions to designate the test oracle. Flaky test case, 2 ways negative impact on testing**: Flaky tests are test cases that do not fail deterministically. Sometimes they fail and sometimes they pass. These are very problematic in industry. (1) If they fail (incorrectly) this leads to a false positive and it means unnecessary work for the tester and perhaps the development team will no longer trust the testing results. (2) If they incorrectly pass it can lead to faulty software being released to the customer. **Guitar and Selenium, model-based:** GUITAR is the model-based tool. It builds a graph model of the user interface (the EFG) and this graph can be used to generate test cases to cover the graph (using different criterion). Selenium is a capture-replay tool. It does not necessarily have a model of the interface, but rather just records test cases and replays them automatically. **Length-2 test cases GUITAR for EFG:** 7 (all the edges). **Length 3 test cases EFG:** Edit-Edit-Copy. **Run test cases during regression testing, automated oracle:** By default, GUITAR just generates tests and we run them. In our exercise we also looked at the output of the test case – the program state—which includes the properties of all of the widgets on the screen at the time that the test ends. Since this is a GUI we can’t just check a functional output so we need to look at the state. This can be done in several ways (which are all correct). When we run the test cases on the original program, we can capture the guitar.STA file for each test case and then when we re-run them after program modifications, we can compare the new state against the old state using a diff tool. The state file has the properties of all of the widgets on the screen at that time. Another option is to use one of several visual screen capture tools which generate screen shots at each step and can do a pixel comparison. **Triangle program, tests reach and reveal fault equilateral:** Yes. Test case: 3,2,2 Isosceles – this will reach the fault and incorrectly propagate/return equilateral which is a wrong result since s2==s3. The correct line should have an ‘and’ not an ‘or’ between the conditions on that line. **Test case reaches fault, should reveal, but doesn’t:** Test case 2,2,2 reaches the code and correctly returns equilateral. However, this should not pass since we are testing for equilateral and that code is not correct. Therefore, we include a test for equilateral in our test set, but it is not the test case that detects this fault. **Test case to increase branch coverage:** 5, 0, 4, - will check the s2<=0 branch of line 18**. Changing order of triangle checks can change results**: If you change the isosceles and equilateral code (line 24-25) , with the isosceles check (line 27-28) then it will pass. But the program can never return equilateral and in fact it becomes unreachable (just as the isosceles cases checked by the current equilateral are unreachable for isosceles right now). Instead you may want to refactor the checks in some other way to make this program more robust. **Possible for complete line coverage:** There is unreachable code in this example – we can never execute line 7 because y is always 1 or 2. This means 100 percent code coverage is impossible (6/7) is the best we can do. **Regression testing:** occurs anytime we change/modify a program. We need to retest the program to ensure that we have introduced new faults into the existing (formerly correct) code. For instance, if we fix the fault in the triangle program above, we would re-test to both ensure that we fixed the fault but to also make sure that no new faults are found. **2 approaches to generate test suites**: Generator: We can either use a model and generate all tests to cover it for a particular criterion (such as we did with GUITAR) Recognizer: or we can write tests and then observe the coverage using a tool such as Cobertura and then add tests to increase missing coverage. **White/Black box testing**: White box testing tests using knowledge of the program internals (code). It is used to structurally cover the code (i.e. all branches, statements) as we have done in the given triangle program on this exam. Black box testing works without internal knowledge. It uses only the program specifications. For instance, if we don’t have the code for the triangle program, we can still use the specs of a triangle (i.e. equilateral/isosceles, etc.) to come up with a set of test cases. We would, however miss faults like we see in the given example here since equilateral would incorrectly pass. Hence, we usually need a combination of the two**. Integration testing:** Integration testing tests more than one unit at a time (subsystem) but does not necessarily run the entire program. Suppose we have 3 classes that work together. We would first unit test each alone. Then we would integrate 2 and 3, 1 and 3 and test those. Last, we would perform a full system test which may also include the environment. **Continuous integration testing:** occurs anytime a change is made to a program. The idea is that we have a set of unit tests and each time we re-build the system, we re-test it. We usually do this in combination with a build system that integrates our new changes and tests to make sure we haven’t broken anything by adding in our new code. It works well with TDD since test driven development builds unit tests as part of the development process. These tests can then be stored/submitted and re-used each time a program changes. **C1 covers all marble colors, C2 covers all finishes. Criteria satisfies T: {1 yellow-shiny, 2 yellow-striped, 1 green-dots, 2 red-dots, 1 black-striped}:** C2; missing color blue so doesn’t satisfy C1. **First criterion selected, is it minimal test set:** No. We can remove a test value such as the yellow-striped test and we still cover C1 Minimal test set: (any that satisfies finish is good) {1 yellow-shiny, 1 green-dot,1 black-striped}. **Subsumption relationship between C1 and C2:** C1 subsumes C1. If we have a test set that covers all of the colors we have to also cover all of the finishes. If we have the three colors (blue, red and black) in our test suite we already have covered all finishes. To satisfy C1 we also need to add the other colors, but this will not increase our finish covering.