Testing

Subject

* Reasons of defects:
  1. Faulty requirements: garbage in -> garbage out
     1. eg incorrect specification
     2. incomplete specification such as not testing all edge cases
  2. Ignored or modified requirements:
  3. Design errors: when we take our high level specification and we translate it down to our low level design
  4. Implementation errors: *when we convert our low level design into source actual code eg missing a guard/other small mistake in the code*
  5. Untested implementation
  6. Procedural errors: when user doesn’t follow specifications aka user errors - impossible to anticipate all the things that users might do with the system
* Automated testing is about testing our system with a variety of inputs and outputs and validating that the behaviour of the system are actually correct
* Automated testing also makes us go back to the specifications and look at it closely to capture what the expected and unexpected behaviours are
* Shortcoming of automated testing: Can show only the presence of defects not absence
* We try to find the most impactful bugs
* Steps for effective testing
  + Chose: chose the most important parts to be tested
  + Create: create the test cases
  + Execute: execute the tests and answering these kinds of high level questions
    - should they be run by the developer before committing the code
    - should they be run on a continuous integration server
    - will we run all tests on each change individually
  + Examine results: introspection
  + Evaluate
* 4 aspects of testing:
  + White box testing
  + Black box testing
  + Assertions
  + Testability
* Terminology:
  + Test case: Evaluates a single explicit behaviour within a program
  + Test suite: sets of test cases
  + **CUT:** code under test aka System Under Test (SUT) represents the code that an individual test case or a set of test cases in a suite is/are evaluating
  + Two mechanisms to write tests:
    - White box testing: we look at the code under test
    - Black box testing: we look at the specification and design our test cases from that
* Favourable Properties:
  1. Fast: should execute quickly as Fast tests encourages developers to run them more often
  2. Reliable: results should be reliable i.e a test case should fail only if there is a real problem
  3. Isolate: a test case failure should tell us exactly where in the code under test the problem arises
  4. Simulate user
* Large number of test failures - reduced ability to isolate failure: When a lot of tests are failing, it's really difficult to tell from where each of the failures arise, or even if there's only one defect that propagates its errors. The tests could still be reliable, as long as they fail consistently.
* Focus on user observable tests:
  + Let go of certain errors that only the system can observe or will handle correct
  + Spend less time fixing bugs that arise from system states you shouldn't be able to enter
* Kinds of tests:
  1. Unit test: measure a single small self contained method
     + Fast:
     + good at Defect isolation:
     + these are the primary form of tests
  2. Acceptance test:
     + slow
     + not good for defect isolation
     + manual
     + done by the user/software owner
  3. Integration test:
     1. less isolated than unit test - more isolated than acceptance test
     2. slower than unit test - faster than acceptance test
     3. smoke test: subset of integration test: more isolated
        + If we only want to validate that a piece of code is running or has basic functionality without robustly ensuring its quality, we can run some simple smoke tests
  4. System test: Encompasses integration (including smoke) and acceptance test
     + Execute broad parts of the system at once, usually with synthetic data
* Red green refactor: Iterative process of writing a test and making it pass is an iterative one.
* Steps for this are:
  1. Write test and make sure it fails
  2. Make the written tests pass
  3. Refactor and extend the test
* **Coverage guided testing:**
  + Coverage measures the proportion of the system being tested
  + coverage reported as percentage
  + coverage% = (covered / (covered + uncovered)
  + Two high level categories of coverage:
    1. Flow independent: cheap to compute
       - Block: measures proportion of blocks in the system that can be executed in the test suite (generally smaller than methods)
       - Line: Measures the proportion of lines executed by the test suite
       - Statement: measure the proportion of statements executed by the test suites
       - Executes each piece of the system and makes sure everything is fine
    2. Flow dependent: more difficult to compute + hard to reason about. gives more insight into how stringently we’ve executed the system
       1. Branch: makes sure we’ve executed both (all) halves of every branching statement
       2. Path: we execute all paths in the system, eg if we have two if statements we execute all combinations of those if statements
       3. MCC coverage: All statements are independently evaluated w.r.t the outcome of a function. (needed only in safety specific systems)
    - Coverage does let us know if any exceptions are raised during execution, but cannot tell us that our program actually did the right thing. A method may do the wrong thing (e.g. return the wrong value) but would achieve the same level of code coverage that a correctly-implemented method would.