Phase
Testing

Overview of Implementation phase

Define Implementation Plan
(+ determine subphases)  
Create Class Skeletons  
Define Coding Standards

For each group of units

Create unit Test plans  
Release unit for integration  
Implement Methods in classes

For each unit

Create system Test plan  
Integration Testing  
Unit test  
Code review

Create integration Test plan

Testing

Goal: Deliberately trying to cause failures in a software system in order to detect any defects that might be present

- Test effectively: uncovers as many defects as possible
- Test efficiently: find largest possible number of defects using fewest possible tests
Basic Definitions

- **Fault:**
  - is a condition that causes the software to malfunction or fail

- **Failure:**
  - is the inability of a piece of software to perform according to its specifications (violation of functional and non-functional requirements)
  - A piece of software has failed if its actual behaviour differs in any way from its expected behaviour

- **Error:**
  - refers to any discrepancy between an actual, measured value (result of running the code) and a theoretical or predicted value (expected result of code from specifications)

Failures are caused by faults, but not all faults cause failures. Unexercised faults will go undetected.

If software system has not failed during testing (no errors were produced), we still cannot guarantee that the software system is without faults.

Testing can only show presence of faults.

A successful test does not prove our system is faultless!!!!
Faults and Failures: example

- Consider an array of integers `myarray[]`, the array is declared to have 10 elements.
- Code contains a loop to print the contents of the array. The array may be partially filled. Loop to print `k` elements: `for (i=0; i<k; i++) {}`
- There is no check in the code to prevent printing more values than the number of available array elements.
- Code is tested for an array with 6 elements. No failures occur.
- There is an undetected fault in the code that will cause failure if `k>9`.

Test Plan

- Specifies how we will demonstrate that the software produces expected results and behaves according to the requirements specification.
- Various tests (each representing a phase in our project schedule):
  - Unit test: usually included within the implementation plan.
  - Integration test: may be included within the implementation plan.
  - System test.
  - User Acceptance test.

Unit Test Plan: (not unit testing)

- Unit test planning activity: (performed by programmer)
  1. Select and create test cases: Plan how to test the unit in isolation from the rest of the system.
- Related Unit testing activity: (performed by tester, probably not the same person as the programmer)
  1. Implement planned test cases into a unit test driver.
  2. Exercise (run) test cases and collect results.
  3. Record and report test results.
Unit Test

- Goal: ensure that a unit (class) works (e.g., methods' post conditions are satisfied)
- Testing "in isolation"
- Activities:
  1. Implement test cases created, (write a unit test driver)
  2. Run unit test driver(s)
  3. Report results of tests

Integration Test Plan

(!= integration and test plan)

- Activity:
  1. Select and create test cases
  2. Plan how we shall test our builds
- Related phase is Integration Test phase
  - > where we exercise integration test cases

Integration Test

- Goal: performed on partially constructed software system (builds) to verify that, after integrating additional bits to our software, it still operates as planned (i.e., as specified in the requirements document)
- Testing "in context"
- Activities:
  1. Write drivers/stubs
  2. Run integration test(s)
  3. Report results of tests
System Testing

- Goal: ensure that the system actually does what the customer expects it to do
- For us, in 275, our system test plan is to successfully execute all of our use cases
- Activities:
  1. Run system test using User Manual
  2. Report results of test

User Acceptance Testing

- Goal: client/users test our software system by mimicking real world activities
- Users should also intentionally enter erroneous values to determine the robustness of the software system

Developing Test Cases

- We could test code using all possible inputs
- Exhaustive testing is impractical
- More extensive testing for mission critical software
- Instead, we shall select a set of inputs (i.e., a set of test cases)
Test Case

- Selection of test cases based on two strategies:
  - Black box testing strategy
    - Select test cases by examining expected behaviour and results of software system
  - Glass (white) box testing strategy
    - Select test cases by examining internal structure of code

Test Case Format

- Test id -> uniquely identify our test case
- Test purpose -> what we are testing, coverage type, resulting sequence of statements ...
- Requirement # -> from SRS, to validate our test cases
- Inputs -> input values and states of objects
- Testing procedure -> steps to be performed by testers to execute this test case (algorithm of test driver for automated testing)
- Evaluation -> step to be performed by testers in order to evaluate whether test is successful or not
- Expected behaviours and results -> test oracle
- Actual behaviours and results -> one way to report

Expected Results

- Expected results are created from the requirements specification, class invariants, post conditions of methods, etc.
- When expected results are not observed a failure has occurred
- Be careful (when developing tests)
  - keep in mind that the expected results, themselves, could be erroneous
Selecting Test Cases

Selection of test cases based on two strategies:

- Black box testing strategy
  - Select test cases by examining expected behaviour and results of software system
- Glass (white) box testing strategy
  - Select test cases by examining internal structure of code

Glass (white) box testing strategy

We treat the software system as a glass box
- Can see inside, i.e., we know about
  - Source code (steps taken by algorithms)
  - Internal data
  - Design documentation
- We can select test cases using our knowledge of the course code, hence test cases can exercise all aspects of each algorithm and data
- More thorough but more time-consuming than black box testing

Select Set of Test Cases using Glass (white) box testing strategy

Goal: Select test cases, i.e., determine values of input, that will cause flow of execution to exercise as much (greatest coverage) of your method as possible

- Test as much of your method as possible with the minimum number of test cases.
- Compromise between extent of test coverage and time needed to complete the tests
  - More robust testing → longer test phase → more expensive testing
  - Less robust testing → shorter test phase → less expensive testing
Black Box Testing

- We treat software system as a black box
- We cannot see inside, i.e., we know nothing about
  - source code
  - internal data
  - design documentation
- So what do we know then?
  - Public interface of the function or class (methods, parameters)
  - Class skeleton (pre and post conditions, description, invariants)
- Test cases are selected based on
  - input values or situation
  - expected behaviour and output of methods

Select Set of Test Cases using Black Box Testing Strategy

1. Determine what to test
   - Which method of which class, type of test
2. For each parameter and object
   a) Establish the parameters equivalence classes OR consider that various states of object
      - Determine all valid (invalid) values or all ranges of valid (invalid) values, and boundaries between those ranges.
      - Determine valid and invalid states of objects (preconditions)
   b) Select representative values for each equivalence class (one from each range, and boundary values) to use as basis of your test cases