Software Construction and Design II Notes

Lecture 1: Introduction To Software Testing

Testing in Software Engineering

- need to build high quality software systems under resource constraints - social, economical, time to market
- software quality assurance - ensuring software under development has high quality (satisfies users needs, correct behaviour, easy to use, does not crash etc., easy to debug and enhance), and creating processes and standards in organisation that lead to high quality software
- software quality is often determined through testing

Software Testing

- validation testing - demonstrate the software meets its requirements
- defect testing - find incorrect or undesired behaviour caused by defects/bugs
- part of software verification and validation (V&V) process
- **unit/functional testing -** process of verifying the functionality of software components (functional units, subprograms) independently from whole system
- **integration testing -** process of verifying interactions/communications among software components. Incremental integration testing vs big bang testing.
- **system testing -** process of verifying functionality and behaviour of entire software system including security, performance, reliability, and external interfaces to other applications
- **acceptance testing -** verifying desired acceptive criteria are met in the system (functional and non-functional) from user point of view

Unit Testing

- process of verifying functionality of software components independently
- method, functions, object classes
- a piece of code written by a developer that executes a specific functionality in the code under test and asserts a certain behaviour or state
- external dependencies are removed
- not suitable for complex user interface or component interaction
- validate each functional unit behaves as expected (defect testing)
- carried out by developers/SW testers and implemented in test framework
- first level of testing
- test fixture is the context for testing - shared set of testing data, methods for setup of that data
- **advantages**
  - change and maintain code at a smaller scale
  - provides notion of documentation - functionality of unit can be derived from unit test
  - discover defects early and fix it at cheaper costs
  - automation of testing processes using test infrastructures
  - easy debugging - reduces difficulty of discovering errors
  - enhances test coverage - ideally, unit test covers all statements in unit
  - code reusability - makes programming agile
  - simplifies integration testing
  - reduces costs of SW development and increases productivity
- **how?**
  - design test cases
    - test normal operation of the unit (positive test) or test abnormality: common problems/ defects (negative test)
    - techniques
      - partition testing (aka equivalence partitioning)
        - identify groups with common characteristics
        - from these groups, choose representative
        - use program specifications, documentation and/or experience
        - *choose on the boundary of partitions*
        - *close to the midpoint of the partition*
        - choose inputs that force the system to generate all error messages

        - design inputs that cause input buffers to overflow
        - repeat the same input of series of inputs numerous times
        - force invalid outputs to be generated
        - force computation results to be too large or too small
      - guideline-based testing
        - use testing guidelines based on previous experience of the kinds of errors often made
        - cookbook-recipe style testing
  - prepare test data
  - run test cases using test data
    - **Junit**
      - an open source framework for writing and running unit tests for java
      - uses annotations to identify methods that specify a test
      - test suite contains several junit tests (test classes) which will be all executed in the specified order
      - assert to check an expected result versus actual results
      - @Before - executed before each test - to prepare test environment (e.g. read input data, initialise the class)
      - @BeforeClass - executed once, before the start of all tests - to perform time intensive activities e.g. to connect to a database
      - junit assumes that all test methods can be executed in arbitrary order
  - compare results to test cases
    - **assert class**
      - provides static methods to test for certain conditions
      - compares actual value with expected
  - prepare test reports
  - **executing tests**
    - tests can be executed from the command line
      - runClass() allows developers
      - information about tests can be retrieved using org.junit.runner.Result object
  - static import is a feature that allows fields and methods in a class as public static to be used without specifying the class in which the field is defined
    - import static org.junit.Assert.assertEquals

Lecture 2a: Review Slides - Design Principles

Design smells
structures in the design that indicate violation of fundamental design principles and negatively impact code quality
- **rigidity (difficult to change)** - every change forces many other changes to other parts of system
- **fragility (easy to break)** - changes causes system to break in places with no conceptual relationship to part changed
- **immobility (difficult to reuse)** - hard to detangle systems into components that can be reused in other systems
- **viscosity -** doing things right is harder than doing things wrong - when design-preserving methods are more difficult to use than the others
- **needless complexity -** when design contains elements that are not useful - when developers anticipate changes to requirements and put in facilities to deal with these potential changes
- **needless repetition**
- **opacity -** tendency of module to be difficult to understand - code is written in unclear and non-expressive way

GRASP
- **responsibility driven design**
  - knowing responsibility - what must a class know, private data, related objects - related to attributes, associations in the domain model
  - doing responsibility - implemented by means of methods
- **creator principle**
  - assign class B the responsibility to create instance of class A if one of these is true:

    - B contains/records/closely uses/has initialising data for A
- **dependency**
  - exists between two elements if changes to the definition of one element (supplier) may cause changes to the other (client)
  - e.g. when class send message to another, one class has another as its data, one class mentions another as parameter to operation, one class is superclass or interface of another
  - <<call>>
  - <<use>>
  - <<parameter>>
- **coupling**
  - how strongly one element is connected to, has knowledge of, or depends on other element
  - assign responsibility so that coupling remains low
- **cohesion**
  - how strongly related and focused responsibilities of an element are
  - if an object has different methods performing different operations on the same set of instance variables, the class is cohesive
  - assign responsibilities so cohesion remains high

SOLID
- **Single responsibility**
  - every class should have single responsibility and that responsibility should be entirely met by that class
- **Open/closed**
  - open for extension but not for mutilation
  - should be able to extend code without breaking it - not altering superclasses when you can do as well by adding a subclass
  - new subtypes of a class should not require changes to the super class
- **Liskov substitution principle**
  - if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program
- **Interface segregation**
  - you should not be forced to implement interfaces you don't use
- **Dependency inversion**
  - no complex class should depend on simpler classes it uses - they should be separated by interfaces (abstract classes)

Key API Design Principles
- **KISS Principle**
  - most systems work best if they are kept simple
  - break down tasks into subtasks
  - keep your methods small
- **YAGNI Principle**
  - a principle of XP - a programmer should not add functionality until really necessary
- **DRY Principle**
  - every piece of knowledge must have a single, unambiguous, authoritative representation within a system
  - minimise code duplication
- **OCAM's Razor**
  - if you have two equally likely solutions to a problem, choose the easiest
- **Documenting an API**
  - should be self-documenting - good names drive good design
  - document religiously anyway
- **Information hiding**
  - minimise the accessibility of classes, fields and methods
  - in public classes, use accessor methods, not public fields
  - prefer interfaces over abstract classes
  - consider implementing a factory method instead of constructor
  - composition - private field to reference instance of existing class
  - prefer composition over inheritance

    - minimise conceptual weight - minimise how many concepts a programmer must learn to use your API
- other programmers are your users
- follow language and platform dependent conventions
- avoid cryptic abbreviations
- minimise mutability - classes should be immutable unless there's a good reason to do otherwise
    - document mutability
    - don't provide mutators, make fields final and private
- use appropriate parameter and return types - favour interface types over classes for input, use most specific type for input type
- use consistent parameter ordering
- avoid long lists of parameters

Lecture 2b: Software Testing Theory

Testing Objectives
- defect discovery
- dealing with unknowns
- incorrect or undesired behaviour, missing requirement, system property
- verifying different system properties - functional, nonfunctional (security, performance, reliability, interoperability, and usability)
- objectives should be stated precisely and quantitatively to measure and control the test process
- testing completeness has never been feasible
    - select test cases sufficient for specific purpose (test adequacy criteria)
    - coverage criteria and graph theories used to analyse test effectiveness
- **defect testing -** find inputs that cause anomalous behaviour
- **validation testing -** find inputs that lead to expected correct outcomes
- **who does testing -** developer, developer peers, specialist tester, users
- in waterfall, testing is done after implementation, during the verification phase
- in agile, unit testing is done daily

Software Testing Process
- select and prepare suitable test cases (selection criteria)
- selection of suitable test techniques
- test plans execution and analysis (study and observe test output)
- root cause analysis and problem solving
- trade-off analysis (schedule, resources, test coverage or adequacy)

Software verification and validation
- **software validation -** are we building the correct product? Process of checking whether the specification captures the customer's needs. *Did I build what I said I would?*
- **software verification -** are we correctly building the product? Process of checking software meets the specification. *Did I build what I need?*
- test cases can disambiguate the requirements

More JUnit
- **parameterised test**
    - a class the contains a test method and that test method is executed with different parameters provided
    - marked with @RunWith(Parameterised.class) annotation
    - the test class must contain static method annotated with @Parameters
    - this method generates and returns a collection of arrays. Each item in this collection is used as a parameter for the test method
- **@Test(timeout = 1000)** to automatically fail tests that 'runaway' or take too long
- **Rules**
    - a way to add or redefine the behaviour of each test method in a test class
    - e.g. ErrorCollector - lets execution of test continue after first problem is found

- will apply to all tests in test suite
- ExpectedException - allows specifying expected exception along with expected exception message

Lecture 3: Advanced Testing Techniques

Integration Testing

- process of verifying interactions/communications among software components behave according to its specifications
- **incremental:** integrate components one by one using stubs or drivers
  - interaction between units are tested incrementally
  - missing components replaced by
    - **stubs:** modules that act as lower level modules that are not integrated yet
    - **driver:** module that act as upper level modules that are not integrated yet
  - **advantage:**
    - defects found early in smaller assembly
    - more thorough testing of whole system
  - **disadvantage:**
    - cost of testing is high/more effort is required
  - **top-down integration:**
    - from top to bottom with regard to dependencies
    - unavailable components substituted by stubs
  - **bottom-up integration:**
    - bottom to top
    - unavailable components substituted by drivers
  - **functional incremental**
    - merging units for testing individual functional requirements as per SW specification
- **big bang:** all components are integrated in one shot
  - **disadvantage:**
    - difficult to isolate errors
    - high probability of missing some critical defects
    - difficult to cover all the cases for integration testing
  - **advantage:**
    - no planning required
    - suitable for small systems
- test cases should be designed to test the behaviour of this module through its interface
- testing may detect incorrect behaviour that results from interactions

Regression testing

- verifies that a software behaviour has not changed by incremental changes to software
- verifies:
  - pre-tested functionality is working as expected
  - no new bugs introduced
- **final regression tests:** validation of the build for deployment/shipment
- **regression tests:** build hasn't broken any other parts by code changes
- **techniques**
  - **rest-test all:** run all tests in test suite
  - **test selection:** run certain ones based on changes in code
  - **test case prioritisation:** run tests in order of priority - determined by how critical and impact of test cases on product
  - **hybrid:** re-run selected test cases based on it's priority

Software Testing Techniques

- **black-box testing**
  - no programming or software knowledge
  - done by software testers
  - specifications based testing

  - used for acceptance and system testing
  - test planned without knowledge of code
- **white-box testing**
  - examines program structure
  - test developer must reason about implementation
  - reveals 'hidden' errors in code

Advanced testing techniques/methods
- **test double**
  - an object that can stand in for a real object in a test
  - includes stubs, mocks and fakes
  - when you have dependency on components that can't be used
  - reduce complexity
  - dummy object
    - **dummy (object):** pass object(s) that never actually used (to fill parameter list)
      - objects passed around by never actually used
      - usually used to fill parameter lists
      - pass object with no implementation
      - SUT's methods to be call often take objects stored in instance variables
    - **stub (test stub):** test specific object that provides indirect inputs into system under test (SUT)
      - act as lower level module that is not yet integrated
      - provide canned answers to calls made during the test
      - responding to test workloads only
      - a test-specific object that provides indirect inputs during tests
      - control indirect inputs of the SUT using test stub
      - e.g. stub of grades system - *our test database? no - fake.*
      - *when(gradebook.gradesFor(student).thenReturn(grades(6,6,0)); //stubbing grades book*
    - **spy (test spy):** capture indirect output calls made by SUT to another component for later verification
      - specific stubs recording information based on how they were called
      - e.g. email service that records how many messages it was sent
      - capture output calls made by SUT to another component for later verification
      - get enough visibility of the outputs generate by the SUT (observation point)
    - **fake:** objects to provide simpler implementation of heavy component
      - objects with working implementations but take shortcuts
      - example: inMemoryTestDatabase
      - objects to provide simplified implementation of a heavy (real) component
      - SUT depends on other components that are unavailable or make testing complex or slow
      - should not be used when want to control inputs to SUT or outputs to SUT
    - **mock:** objects to verify indirect output of tested code
      - pre-programmed with expectations form a specification of the calls they are expected to receive
      - throw and exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting
      - objects that verify indirect output of the tested code
      - e.g. function that calls email sending service, not to really send emails but verify email sending service was called
      - calling real implementation during testing is tedious, or the side effect is not the testing pool

Contact Test
- process of running periodic tests against real components to check the validity of test doubles results
- How
  - run your own test against the double
  - periodically run separate contract tests (real tests to call the real service)