SENG 438 Notes

Brian Pho

April 22, 2018

Contents

Chapter 1

Introduction

1.1 Terminology

What Affects Software Quality

- Time
- Cost
- Quality

Threats Error → Fault → Failure
Failures have two attributes: mode (how the failure is found/caused) and effect (what the failure causes and how it propagates).

- **Error**: Human action that causes a fault
- **Fault**: Discrepancy in code that causes a failure
- **Failure**: External behavior that is incorrect

Attributes $Availability = \frac{Uptime}{Uptime\ +\ Downtime}$
Reliability (probability that the system works)
$Reliability = e^{-\lambda t}$
$MTTF = \frac{1}{\lambda}$
MTBF = MTTF + MTTR

- Lambda is the failure rate
- Failure density is good if you do a lot of testing
- Testing vs Inspection
- Inspections are code reviews. Testing is writing cases. Inspection is difficult with large systems.
- Reliability: continuity of correct service. Availability: readiness for correct service.
- Fault prevention, tolerance, removal, forecasting
- Software quality: conformance to requirements and fitness for use

Chapter 2

Foundations of Testing

Testing *Unit Testing* (testing single component)→ *Integration Testing* (testing combination of components)→ *System Testing* (testing all combined components)→ *Acceptance Testing* (testing with respect to user needs, requirements)

- Validation: building the **right thing**
- Verification: building the **thing right**
- Direct input variable vs indirect input variable. Use direct for unit/initial testing, then use indirect for integration testing
- Test case: specifying direct input and expected outcome, specifying indirect input and expected context
- Test suite: a set of test cases
- Test plan: a set of test suites
- Test Plan → Test Suite → Test Case
- Write fewer test cases but maximize the chance of covering faults

2.1 Testing Approaches

- Scripted: planned test cases
- Exploratory: random ad hoc
- Regression: testing after a software update to check that no faults have been reintroduced
  - In the small: choose small set of inputs, consider features in isolation
  - In the large: sequence of several small test cases
- Unit testing: testing of individual components
- Integration testing: testing of combination of components

- System testing: testing the complete system
- Acceptance testing: testing by users
- Test Driven Development: write tests first, then code to pass the tests
- Functional testing: testing functional requirements
- Non-functional testing: testing non-functional requirements
- Exhaustive testing: test using all possible inputs
- Random testing: test randomly
- Partitioning testing: test in cases

2.2 Bug Report

Report the following

- Coding error
- Design issue
- Requirements issue
- Documentation/code mismatch
- Specification/code mismatch

2.3 Requirement Analysis

- Requirement Traceability Matrix (RTM): a table that maps test cases (rows) to the requirement (columns) and x the cell where that testing case tests that requirement

Chapter 3

Unit Testing

3.1 Testing Approaches

- Exhaustive: testing all possible inputs
- Random: ad hoc willy nilly
- Partitioning: testing a set of possible behaviors

3.2 Criteria Based Technique

- White box: access to source code, goal of increase code coverage
- Black box: access to specification, goal of increase input domain coverage

3.3 Unit Testing

- Must only test one specific unit of functionality
- No access to database, file, network

3.4 JUnit

- Test runner: executable program that runs tests implemented using JUnit
- Test fixtures: the set of preconditions
- Test case: most elemental class
- Test suites: set of tests that all share the same fixture
- Test drivers: modules that act as temporary replacement for a calling module
- Test execution: the execution of an individual unit test
- Test result formatter: a test runner produces results in one or more output formats
- Assertions: an assertion is a function/macro that verifies the behavior of the unit test

3.4.1 Setup and Teardown

- @BeforeClass
- @Before
- @Test
- @After
- @AfterClass

3.4.2 Assertions

- void assertEquals("message", expected, actual)
- void assertTrue("message", condition)

- JUnit doesn't follow the given order of test methods in the test class for execution (don't write tests that depend on other tests, aka be self-contained)

3.4.3 JUnit Coding Tips

- Well structured assertions: expected value should be at left and the test cases should have a message
- Expected answer objects: use to minimize testing
- Naming test cases: use long descriptive names
- Good assertion messages: don't put expected and actual output as JUnit already does that
- Pervasive Timeouts: use timeouts for every test case to avoid infinite loop
- Should only have 1 assert statement
- Should avoid logic (conditionals, try/catch, loops)

- Testing for Exceptions: test passes if it does throw the given exception

3.5 Dependencies: Stubbing & Mocking

- Unit tests should not have dependencies
- SUT: system under test
- A test double is a replacement for a DOC (dependent on component)
- Stubs/mocks are fake/duplicate components
- Stubs: supplies responds to requests
- Mock: a fake object that decides whether a unit test has passed or failed by watching interactions between objects

- Mock vs stub: same for state testing (what is the result?), only mock for behavioral testing (how the result has been achieved?)
- Mock interacts directly with the unit test, the stub doesn't
- Stub → SUT → Test
- Mock → SUT and Test
- Using a mock → unit testing
- Using actual implementations → integration testing
- Benefits of Mocking

  If an object has any of the following characteristics, it may be useful to use a mock object in its place:

  - supplies non-deterministic results (E.g. the current time/temperature)
  - states that are difficult to create/reproduce (E.g. network error)
  - is slow (E.g. connecting to database)
  - does not yet exist or may change behavior (E.g. polymorphism)
  - would have to include information and methods exclusively for testing purposes

Chapter 4

Black Box & Combinatorial Testing

4.1 BB and WB Testing

- Black box testing: testing (functional or non-functional) without reference to the internal structure of the components or system. Applies at all granularity levels:
  - Unit
  - Integration
  - System
  - Regression
- White box testing: testing based on an analysis of the internal structure of a component or system. Only applies to:
  - Unit
  - Integration
- White box testing, unlike black box testing that is using the program *specifications* to examine outputs, white box testing is based on specific *knowledge of the source code* to define the test cases and to examine outputs

4.2 Equivalence Classes

- Inputs in an equivalence class are assumed to be "treated the same" by the system
- You divide the input set into partitions that can be considered the same
- Equivalence classes: partitions of the input set in which input data have the same effect on the SUT (system under test)
- The entire input set is covered: completeness
- Disjoint classes: to avoid redundancy
- A group of tests cases are "equivalent" if:

  - They all test the same unit
  - If one test case can catch a bug, the others will probably do the same
  - If one test case does not catch a bug, the others probably will not do the same
  - They involve the same input variables
  - They all affect the same output variables
- The entire set of inputs to any application can be divided into at least two subsets:
  - One containing all of the expected/legal inputs
  - One containing all of the unexpected/illegal inputs
- Each of these two subsets can be further subdivided
- One-dimensional partitioning: apply partitioning to each variable individually
- Multi-dimensional partitioning: the Cartesian product of the input variables

4.2.1 Weak/Strong Equivalence Class Testing

- Weak ECT: choosing one variable value from each EC such that all classes are covered $max(|A|, |B|, |C|)$
- Strong ECT: based on the Cartesian product of the partition subsets $|A|\ x\ |B|\ x\ |C|$

4.2.2 ECT Hints

1. Don't forget EC for invalid inputs
2. Look for extreme range of variables
3. Look for maximum size of memory variables
4. If an input must belong to a group, one EC must include all members of that group
5. Analyze levels for binary value variables
6. Look for dependent variables and replace them by their equivalents

4.3 Boundary Value Testing (BVT)

- Programmers make mistakes in processing values at and near the boundaries of equivalence classes
- Tests derived using either of the two techniques may overlap
- While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.
- For each EC, setting values for input variables just below their min, at their min, just above the min, a nominal value, just below their max, at their max, and just above their max

- Limitations: Need $4n + 1$ test cases for n variables
- Works well with variables that represent bounded physical quantities
- But has no consideration of the nature of the function and the meaning of the variables
- BVA tests all but BLB and AUB due to outside of EC range

4.3.1 Boundary Values

- BLB: below lower bound
- LB: lower bound
- ALB: above lower bound
- NOM: nominal value
- BUB: below upper bound
- UB: upper bound
- AUB: above upper bound

4.3.2 BVA Hints

- Check if the boundary conditions for variables are set correctly
- Check if the inequality boundary conditions can be changed to equality or not
- Check if the counter variables allow departure from the loop correctly or not
- If a program needs a specific number of inputs, give it that many, one more, and one fewer
- When reading from or writing to a file, check the first and last characters in the file

4.4 BB Testing Variation

4.4.1 Robustness Testing

- Testing the BLB (below lower boundary) and AUB (above upper boundary)
- Weak-Robust ECT: enough to cover the domain (including BLB and AUB)
- Strong-Robust ECT: covers all possible combinations (within and outside of EC) of the domain

4.4.2 Worst Case Testing

- BVA makes the assumption that failures, most of the time, originate from one fault related to an extreme value
- What happens when more than one variable has an extreme value?
- Cartesian product of $\{LB, ALB, NOM, BUB, UB\}$
- 5^n number of test cases for n variables

4.4.3 Category-Partition Testing

- Makes it easier to define special case categories
- Combines BVA and ECT

4.4.4 Decision Tables

- Can help us deal with combination of inputs which produce different results
- Helps express testing requirements in a directly usable form
- Conditions $\rightarrow$ Decisions $\rightarrow$ Actions
- Conditions express relationships among decision variables
- Actions are responses to be produced when corresponding combinations of conditions are true
- Actions are independent of input order and the order in which conditions are evaluated
- Their will be one test case for each test rule from the decision table

4.5 Combinatorial Testing

- Combine values systematically but not exhaustively
- Rationale: The most common bugs in a program are generally triggered by either a single input parameter or an interaction between pairs of parameters. Bugs involving interactions between three or more parameters are both progressively less common and also progressively more expensive to find—such testing has as its limit the testing of all possible inputs.
- Look for interaction of the inputs
- $\frac{n!}{t!(n-t)!} = \binom{n}{t}$ where: If a set has n elements, the number of t-way independent combinations is equal to the binomial coefficient
- Pair-wise Testing: for each pair of input parameters to a system, tests all possible combinations of each pair

4.6 Conclusions

- Black Box: testing without knowing the internals, goal of testing behavior of system
- EC: tested based on partitioning the input space
- BVA: testing the boundaries of the ECs
- Robustness Testing: BVA but including below the lower bound and above the upper bound
- Combinatorial Testing: testing based on combinations of inputs

Chapter 5

White Box Testing

5.1 Control-Flow Coverage

5.1.1 Control Flow Graph (CFG)

Software structure can have three attributes:

- Control-flow structure: the sequence of execution of instructions of the program
- Data flow: keeping track of the data as it is created or handled by the program
- Data structure: the organization of data itself independent of the program

Graph

- N nodes
- N_0 initial nodes
- N_f final nodes
- E edges
- Path: a sequence of nodes
- Length: the number of edges
- Subpath: a subsequence of nodes in p is a subpath of p
- Test path: a path that starts at an initial node and ends at a final node

We use graphs in testing to:

- Develop a model of the software as a graph
- Require tests to visit or tour specific sets of nodes and edges
- **Structural Coverage Criteria**: Defined on a graph just in terms of nodes and edges
- **Data Flow Coverage Criteria**: Requires a graph to be annotated with references to variables

- Graph: usually the control flow graph (CFG)
- Node coverage: execute every program statement
- Edge coverage: execute every branch
- Loops: looping structures
- Data flow coverage: augment the CFG with extra information
- CFG: directed graph/ UML activity diagram
- Prime Flow Graphs: are flow graphs that cannot be decomposed non-trivially by sequencying and nesting
- Sequencing and nesting

Constructing CFG

Can use UML Activity diagram to model/draw CFG

- Each node corresponds to a line of code
- Each edge is a path to that line of code
- if statement without else
- if statement with else
- if statement with return inside if
- while loop
- for loop
- do-while loop
- switch case
- try-catch

5.1.2 Control Flow Coverage

Steps for control-flow based testing

1. From the source code create a control flow graph
2. Design test cases to cover certain elements of CFG
3. Decide upon appropriate coverage metrics to report test results
4. Execute tests and collect coverage data

5.1.3 Coverage

- Test coverage measures the amount of testing performed by a set of tests
- $Coverage = \frac{Number\ of\ coverage\ items\ exercised}{Total\ number\ of\ coverage\ items} * 100\%$
- $Statement\ Coverage = \frac{Number\ of\ statments\ exercised}{Total\ number\ of\ statements} * 100\%$
- $Decision\ Coverage = \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decision\ outcomes} * 100\%$

Statement Coverage

- Hypothesis: faults cannot be discovered if code lines containing them are not executed
- Equivalent to covering *all* nodes in CFG
- Executing a statement is a weak guarantee of correctness but easy to achieve
- How can we minimize the number of test cases so we can achieve a given statement coverage percentage?

Decision Coverage

- Select tests such that each edge of the program's CFG is traversed at least once
- We need to traverse all decisions/branches possible to check if we enter that branch

Condition Coverage

- Design a test set such that each individual condition in the program to be both true and false (this may cause branch coverage to suffer though)
- Differs from decision coverage in that it checks the condition, not whether the condition branches or not

Condition vs Decision Coverage

- In Decision Coverage (also know as Branch Coverage) you have to test all possible branches.
- In Condition Coverage (also know as Predicate Coverage) each of the Boolean expressions must be evaluated to true and false at least once.

Modified Condition-Decision Coverage (MC/DC)

- Only test the important combinations of conditions
- Only combinations of values such that every atomic condition toggles the overall condition's truth value
- Minimum set size = N + 1

- Create a truth table of the input parameters as columns and see which parameter affects which test cases
- Determine which parameter affects which test by supposing the parameter changed values, does the outcome change? If it does then both tests are a pair.
- Select test cases by choosing the pairs that cover each condition

Path Coverage

- Select a test set such that by executing the program for each test case, all paths leading from the initial to the final node of the program's CFG are traversed
- Full path coverage will lead to full branch coverage

Subsumption

Subsumes: x subsumes y if and only if x covers everything that y can do

- MC/DC subsumes Branch and Condition coverage
- Branch and Condition coverage subsumes Branch coverage
- Branch coverage subsumes Statement coverage
- Branch and Condition coverage subsumes Condition coverage

5.2 Cyclomatic Complexity

- A program's complexity can be measured by the cyclomatic number of the program flowgraph
- There is always a trade-off between control-flow and data structure. Programs with higher cyclomatic complexity usually have less complex data structure. This means programs with higher cyclomatic complexity require less effort.
- The cyclomatic number can be calculated in three different ways:
  - Flowgraph based
  - Code based
  - The number of regions of the flow graph

5.2.1 Flowgraph

- For a program with the program flowgraph, G, the cyclomatic complexity, v(G), is measured as $v(G) = e - n + 2p$
- e: number of edges
- n: number of nodes

- p: number of connect components
- Another way of calculating v(G), $v(G) = 1 + d$
- d: number of predicate nodes (nodes with an out-degree other than 1, aka conditions and loops)

5.2.2 Code

- Count the number of conditionals (ifs) and loops (while, for, do-while) and add 1 to that

5.2.3 Regions of the Flow Graph

- The areas of a FG that are enclosed by the edges, in addition to the outside region

5.3 Data-Flow Coverage

- Motivation: statements interact through data flow
- Node and edge coverage doesn't test interactions
- Basic Idea: test the connects between variable definitions (write) and variable uses (read)
- 100% MC/DC cannot guarantee that a program does not fail

5.3.1 Definitions

- Data Flow Analysis: focus on paths that are significant for data flow
- Definition occurrence: a value is written to a variable
- Use occurrence: value of a variable is read
- Predicate use: a variable that is used to decide whether a condition is true/false
- Computational use: a variable that is used to compute a value
- All-Defs Coverage (ADC): Some path from each definition to some use
- All-Uses Coverage (AUC): Some path from each definition to each use
- All-Du-Paths Coverage (ADUPC): All paths from each definition to each use
- A definition-use pair (DU pair) is a pairing of definition and use of a variable, with at least one def-clear path between them (there could be many).

5.3.2 Conclusions

- Coverage is a measure of WB testing effort to detect potential faults
- 100% statement coverage means that you tested for every bug that can be revealed by simple execution of a line of code
- 100% branch coverage means you will find every fault that can be revealed by testing each branch
- 100% coverage means that you tested for every possible fault
- To get 100% decision coverage every decision in the program must take all possible outcomes at least once.
- To get 100% condition coverage, every condition in a decision in the program must take all possible outcomes at least once.
- Modified condition decision coverage (MCDC): every condition in a decision has been shown to independently affect that decisions outcome.

Chapter 6

Mutation Testing

- Coverage-based (BB, WB testing): the better test suite covers more from specification/code
- Fault-based (Mutation testing): the better test suite detects more faults
- Coverage metrics cannot ensure test quality
- Mutation = bug injection
- Mutation testing = finding injected bugs

6.1 Mutation Testing: Concept

- We intentionally inject faults and run our test suite to see if it can detect the injected faults
- A good test suite is supposed to kill mutants
- Surviving means changing the source code did not change the test result (bad)
- Killed means changing the source code changed the test results (good)

6.2 Mutation Testing: Mutants

- Stillborn mutants: syntactically incorrect, killed by compiler
- Trivial mutants: killed by almost any test case
- Equivalent mutant: always acts in the same behavior as the original program

6.2.1 Object-Oriented Mutations

- Access Modifier Change (AMC): changes the access level for instance variables and methods
- Hiding Variable Deletion (IHD): deletes a hiding variable
- Hiding Variable Insertion (IHI): inserts a hiding variable into a subclass
- Parameter Variable Declaration (PPD): same as PMD except that it operates on parameters rather than on instance and local variables

6.2.2 Mutation Assessment

- Mutation Coverage/Score = $\frac{\#\ of\ mutants\ killed\ by\ the\ test\ suite}{\#\ of\ all\ non-equivalent\ mutants}$
- Killing mutant conditions
  - A test must reach the mutated statement
  - Test input data should infect the program state by causing different program states for the mutant and the original program
  - The incorrect program state must propagate to the program's output and be checked by the test
- Strong Mutation (All three)
- Weak Mutation (The first two)
- If a mutant is not killed by a test suite, the test suite is inadequate and needs to be enhanced

Chapter 7

GUI Test Automation

Problems with Manual Testing

- Not very repeatable
- Very effort intensive

Major things to test with GUIs

- Events
- States
- Functions (logical)

Coverage Criteria

- Event-coverage: all events of the GUI need to be executed at least once
- State-coverage: all states of the GUI need to be exercised at least once
- Functionality-coverage: using a functional point of view

Myths, Misses, Hints

- Test Automation is simple (myth)
- Don't underestimate the cost of automation
- Don't underestimate the need for staff training
- Commercial test tools are expensive
- Don't shoot for 100% automation
- Don't create test scripts that won't be easy to maintain
- Treat test automation as a genuine programming project
- Don't forget to document your work

General Notes

- A capture and replay testing tool captures user sessions and stores them in scripts suitable to be used to replay a user session

Chapter 8

Software Quality & Reliability Engineering

- Quality: fitness for use, conformance to requirement
- Software Reliability Engineering (SRE) is the middle solution between slow, expensive development and fast, cheap development
- Hardware faults are mostly physical faults
- Software faults are mostly design faults
- SRE is a multi-faceted discipline covering both technical and management activities in three basic areas:
  - Software Development and Maintenance
  - Measurement and analysis of reliability data
  - Feedback of reliability information into the software life cycle activities
- SRE is a practice for quantitatively planning and guiding software development and testing
- SRE does three things simultaneously
  - Ensure product reliability and availability meet user needs
  - Delivers the product to market faster
  - Increases productivity
- Reliability $= e^{-\lambda t} = e^{\frac{-t}{MTTF}}$

8.1 Software Reliability Engineering Process

- Define necessary reliability
- Develop operational profiles
- Prepare for test

- Execute test
- Apply failure data to guide decisions

Chapter 9

System Reliability

Can we calculate system reliability as a function of the reliabilities of its components and of the relationship between the components?

9.1 Reliability Block Diagram

- Reliability Block Diagram (RBD): graphical representation of how the components of a system are connected from a reliability POV
- Reliability of the system is derived in terms of reliabilities of its individual components
- Common RBD configurations are: serial and parallel
- **Serial system configuration**: elements must all work for the system to work and the system fails if one of the components fails
- **Parallel system configuration**: elements are considered to be redundant and the system will cease to work if all the parallel elements fail
- The overall reliability of a serial system is lower than the reliability of its individual components but it's higher for parallel systems

9.1.1 RBD Process/Steps

1. Define boundary of the system for analysis
2. Break system down into functional units
3. Determine serial-parallel configurations
4. Represent each components as a separate block in the diagram
5. Draw lines connecting the blocks in a logical order for mission success

9.2 Serial System Reliability

- No redundancy
- Can be calculated from component reliabilities
- Each component has an R value and a Lambda value where Lambda is the failure rate (E.g. R_1/λ_1)
- $R = \Pi_{k=1}^{Q_p} R_k$ and $\lambda = \Sigma_{k=1}^{Q_p} \lambda_k$ where Q_p is the number of components, R_k is the component reliability, and λ_k is the component failure rate.
- **To get the system reliability, multiply all of the component reliabilities**
- Component reliabilities (R_k) must be expressed with respect to a common interval (E.g. 10 hours of operation)
- $R_k \leq 1$ since a serial system always has smaller reliability than its components

9.3 Parallel System Reliability

- Has redundancy
- Can be calculated from component reliabilities
- $R = 1 - \Pi_{k=1}^{Q_p}(1 - R_k)$ and $\frac{1}{\lambda} = \Sigma_{k=1}^{Q_p} \frac{1}{\lambda_k} = MTTF = \Sigma_{k=1}^{Q_p} MTTF_k$ where $1 - R_k$ is the unreliability of a component
- **To get the system reliability, do 1 - the component reliability, multiply all of those up, then do 1 - that answer**

9.4 Series-Parallel Reliability

- Can think of each component like a resistor and apply the corresponding formula to that set of components

How to avoid single point of failure:

- Adopt redundancy
- Use extremely reliable equipment
- Perform maintenance
- Reduce/eliminate extreme service stress

9.5 m - out of - n System

- System has n components
- At least m components need to work correctly
- m = n for serial system
- m = 1 for parallel system

9.6 Fault Tree Analysis (FTA)

- Fault Tree Analysis: graphical representation of the major faults or critical failures associate with a product, the causes for the faults, and potential countermeasures. Uses logical operators.
- The graph represents the pathways within a system that can lead to a foreseeable undesirable event.
- Failure Modes and Effects Analysis (FMEA): determines the failure modes that are likely to cause failure events.
- In order to perform FTA, first need to perform FMEA
- Converting FTA to RBD: AND turns into parallel, OR turns into serial
- Pros: makes trace back easy to view, produces meaningful data, evaluation is effective
- Cons: must foresee all possibilities, time consuming, probabilities may not be accurate

Chapter 10

Reliability Testing Tools and Techniques

10.1 SRE Tools

- Failure Data → SRE Tool → Output Data

10.2 System Reliability Assessment

Types of Assessment

- Decisions related to certification test
- Decisions related to reliability growth test
- Decisions related to adequacy of tests

Three Techniques

- Reliability Demonstration Chart (RDC): based on inter failure times and target failure intensity or MTTF
- Reliability Growth Analysis: based on inter failure times/ failure count and failure intensity or MTTF
- Zero Failure Testing: based on failure density

10.3 Assessment Criteria

10.3.1 Certification Testing Using RDC

- A reliability demonstration chart (RDC) shows when cumulative failure observations indicate that a failure intensity objective has or has not been met.
- If a system passes its target MTTF without failure, it is acceptable.

- γ is the risk related to the measured entity
- β is the risk in assessing the entity which is actually wrong to be right (purchaser risk/ false positive)
- α is the risk in assess the entity which is actually right to be wrong (supplier risk/ false negative)
- λ_F is the failure-intensity objective (FIO)
- Failure times are normalized by multiplying by the appropriate failure-intensity objective.
- Vertical axis: failure number
- Horizontal axis: normalized failure data (failure time * λ_F or failure time / MTTF)
- When risk levels (α and β) decrease, the system will require more tests
- When discrimination ratio (γ) decreases, the system will require more tests
- Pros: time and cost efficient
- Cons: cannot come up with a quantitative number for reliability

10.3.2 Reliability Growth

10.3.3 Zero Failure Testing

10.3.4 Special Cases

Chapter 11

Integration, System, and Acceptance Testing

11.1 Introduction

- Unit testing: take a small piece of code and test it in isolation
- Since defects can happen at different levels and stages, need testing at different levels and stages
  - Unit Testing: testing of individual components
  - Integration Testing: testing of combination of components
  - System Testing: testing of the complete system
  - Acceptance Testing: testing system from users' perspective

11.2 Integration Testing

- Goal: test all interfaces between subsystems and the interaction of subsystems
- Integration testing strategy determines *the order* in which subsystems are selected for testing and integration
- Decomposition-base integration
  - Big Bang: combine all unit tests into one big test
  - Bottom Up
  - Top Down
  - Sandwich
- Incremental Integration: test one component, then two, then three, ...

  Easier to find fault location and fix

- Continuous Integration: test from the beginning and continually test while building
  Requires integrated tool support

11.2.1 Big Bang Strategy

- Reason: if already have tested units, why not combine them all at once to save time?
- But based on false assumption of no interaction faults
- Takes longer to locate and fix faults

11.2.2 Bottom Up Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously test subsystems
- This is repeated until all subsystems are included
- Drivers are needed
- Drivers: a component that calls the TestedUnit, it controls the test cases
- Pros: visibility of detail and fault localization
- Cons: tests the most important subsystem last and drivers needed

11.2.3 Top Down Strategy

- Test the top layer/controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Test Doubles are needed to do the testing
- Test Doubles: mocks or objects that can stand in for a real object
- Pros: test cases can be defined in terms of the functionality of the system and critical control structure is tested first and most often
- Cons: large number of doubles needed and details are left for the last tests

11.2.4 Sandwich Strategy

- Combines the bottom up and top down approach
- Pretty much big bang integration on a subtree
- Pros: less stub and driver development
- Cons: more difficult to find fault location

11.3 System Testing

11.3.1 Functional Testing

- The last integration step
- Tests both the functional and non-functional
- Goal: test the functionality of the system
- Test cases are designed from the requirements analysis document
- the system is usually treated as a black box

11.3.2 Testing

- Suggested test sequence
    1. Unit Test
    2. Integration Test
    3. Regression Test
- Identifying System Failures
    1. Analyze the test output for deviation
    2. Determine which deviations are failures
    3. Establish when the failures occurred
    4. Assign failure severity classes, which will be used in prioritizing failure resolution
    5. Document failures

11.4 Acceptance Testing

- The final stage of validation
- Goal: demonstrate the system is ready for operational use
- "If you don't have the patience to test the system, the system will surely test your patience."
- Reasons: users know realistic use cases, variants to standard tasks

11.4.1 Alpha and Beta Tests INCOMPLETE

Similarities

- Testing by customers/representatives of your market
- When software is stable
- Uses the product in a realistic way
- Give comments back on the product

Differences

- Alpha Testing is simulated
- Best Testing is conducted at the client's environment

Chapter 12

Post Release Activities

NOT COVERED IN FINAL EXAM.