CA267 Software Testing

Accompanying Books:
Software Testing Foundations by Andreas Spillner
Lecturer:
Dr Silvana MacMahon

Syllabus:

1. Fundamentals of testing
2. Testing throughout the software lifecycle
3. Static techniques
4. Test design techniques
5. Test Management
6. Technology support for testing

- Fundamentals of testing
  - What is testing?
  - General testing principles
  - Fundamental test process
  - The psychology of testing

- Testing throughout the software life cycle
  - Software development models
  - Test levels
  - Test types
  - Maintenance testing

- Static techniques
  - Static techniques and the test process
  - Review process
  - Static analysis by tools

- Test design techniques
  - The test development process
  - Categories of test design techniques
  - Black-box techniques
  - White-box techniques
  - Experience-based techniques
  - Choosing test techniques

- Test management
  - Test organization
  - Test planning and estimation
  - Test progress monitoring and control
  - Configuration management
  - Risk and testing
  - Incident management

- Technology support for testing
  - Types of test technologies
  - Effective use of tools: potential benefits and risks
  - Introducing a tool into an organization

ISTQB is the global standard for certification of software testers
International **S**oftware **T**esting **Q**ualifications **B**oard

Understand concepts and be able to apply them and express your opinions on them

Fundamentals of Testing

What is the scope of software testing?

↓ Requirements
↓ Design ↑
↓ Coding ↑
↓ Testing ↑
↓ Fixing ↑
↓ Shipping
↓ Maintenance

Testing is now done at all stages of the life cycle
Testing is expensive
Not testing enough can be more expensive.
If testing is put at the final stage of development, you could be subject to significant time pressure.

Why is testing necessary?

A failure is present if a user's expectation is not fulfilled adequately or there is a deviation of the software from its expected delivery or service.

E.g., products are too hard to use or too slow, i.e., the user experiences a problem

Role of testing in development maintenance and operations
Rigorous testing of systems and documentation can help to:

- Reduce the risk of problems occurring in an operational environment
- Contribute to quality

Software testing may be required :

- To meet contractual or legal requirements
- Or industry-specific standards

Test gives us the confidence that software will work

Testing can measure quality of a product and indirectly improve its quality

How much testing is enough?

There is no upper limit, you can test a piece of software indefinitely.
If we do not test enough, we might miss things and get faults in production
Think of managing a deficit or a surplus.
If you test too much it takes time, money, and resources. If you test too little you get blamed for defects later found

Objective coverage measures can be used:

Standards may impose a level of testing

Test design techniques give an objective target

Time is often a limiting factor so we might have to rely on a consensus view and at least do the most important tests.

Complete testing is nearly impossible :

- The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid and invalid inputs.
- The program may have many states.
- The design issues may be too complex (especially for larger systems)
- It may not be possible to simulate every single possible test case

Where are the bugs?

Experience tells us:

Bugs tend to cluster (You find a lot of them in certain areas)

Some parts of the system will be relatively bug free

Bug fixing and maintenance are error prone (We could make changes that make more problems)

Test as a fishing net

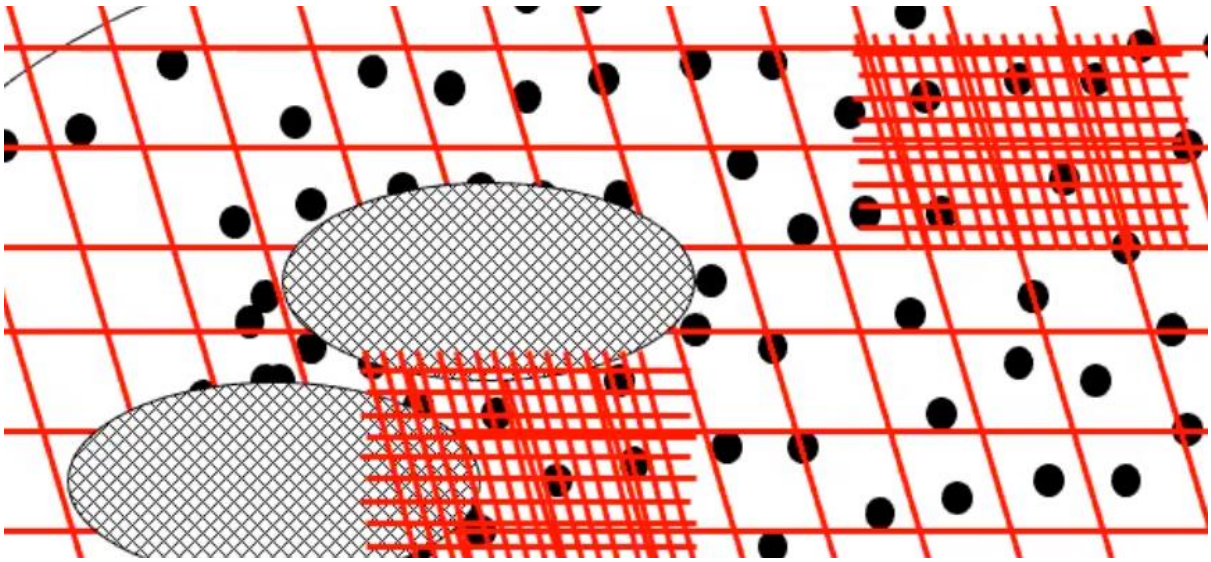
Size of the net reflects the size of the test

The smaller the mesh the more depth and thoroughness the test has

Large nets, large mesh to give overall confidence

Small nets, small mesh to find bugs:

- Use them in the cluster areas we expect bugs to be
- Use them in the critical areas, important areas



What about bugs we don't find?

What are testing and debugging?

Testing is the systematic exploration of a system with the main aim of finding and reporting defects
Testing rigorously examines the system behaviour and reports defects found
Tests are repeated to ensure that defect corrections have been effective

Debugging is a process undertaken by developers to identify the cause of defects in code and undertake corrections.
Debugging is done first to ensure that the component or system is at a level to enable rigorous testing.
Debugging can be used to understand the root cause of observed failures.
Programmer activity that identifies the cause of a defects, repairs the code and checks that the defect has been fixed correctly

Debugging happens before testing
Testers test, Developers Debug

What is a test?

A test is a controlled exercise involving an object under test, a definition of the environment, a definition of the inputs, a definition of expected outputs or result.

After the test we can make a decision on whether the test was a success or failure.
Before testing we need:

- Planning and control (how much testing is enough, approach)
- To choose test conditions (design of test cases)
- To design test cases (defining inputs)
- To prepare expected results
- To check actual results against expected results
- To evaluate completion criteria (Have we done enough testing)
- To report on the testing process and system to stakeholders
- To close the test phase (learnings to use on a new project)

Test objectives:

- Finding defects
- Gaining confidence about the level of quality
- Preventing defects (taxonomy-based testing)

In acceptance testing, the main objective may be to confirm that the system works as expected or to gain confidence that it has met the requirements

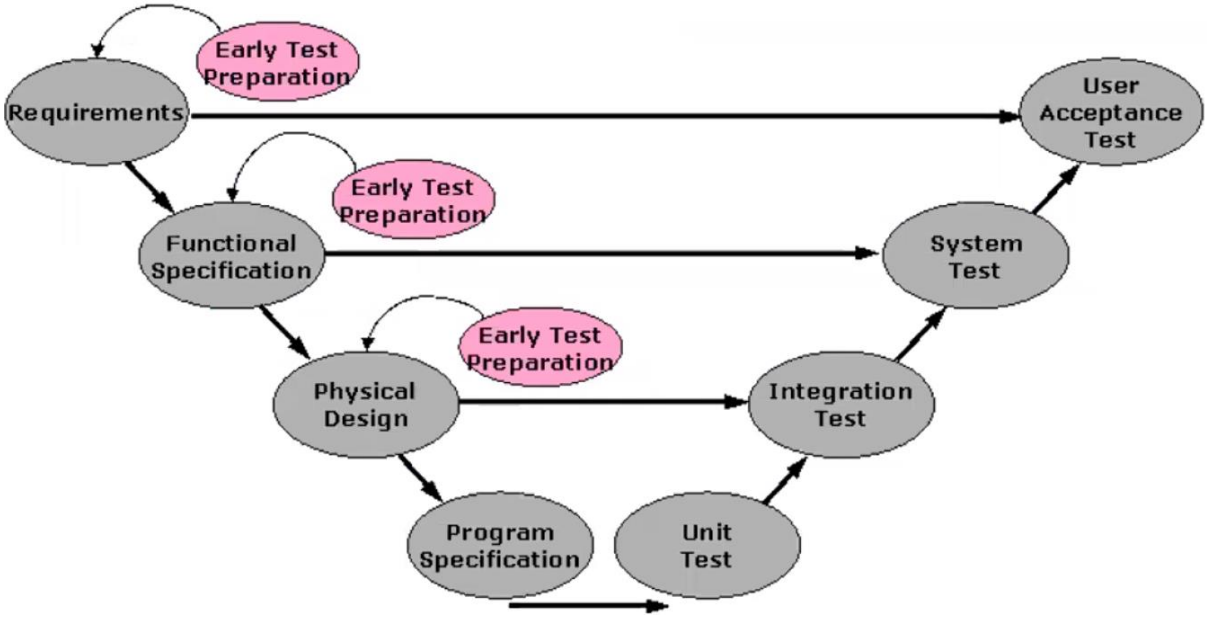
Static and Dynamic Testing

Static Testing is testing without executing the program (we don't actually run the code)
Looking for potential faults. Reviewing of documents (source code, requirements docs, design docs, test plans)

Dynamic Testing is testing by executing the program with real inputs.
Coding stage must be completed beforehand.

Static testing in the life cycle:
- Testing happens in:
  - Reviews, walkthroughs, inspections of documentation.

Dynamic testing in the life cycle.
- Testing happens in:
  - Program, integration or component testing,
  - System testing:
    - Non-functional (e.g performance, security, backup and recovery, stress)
    - Functional testing
  - User acceptance testing



Types of testing

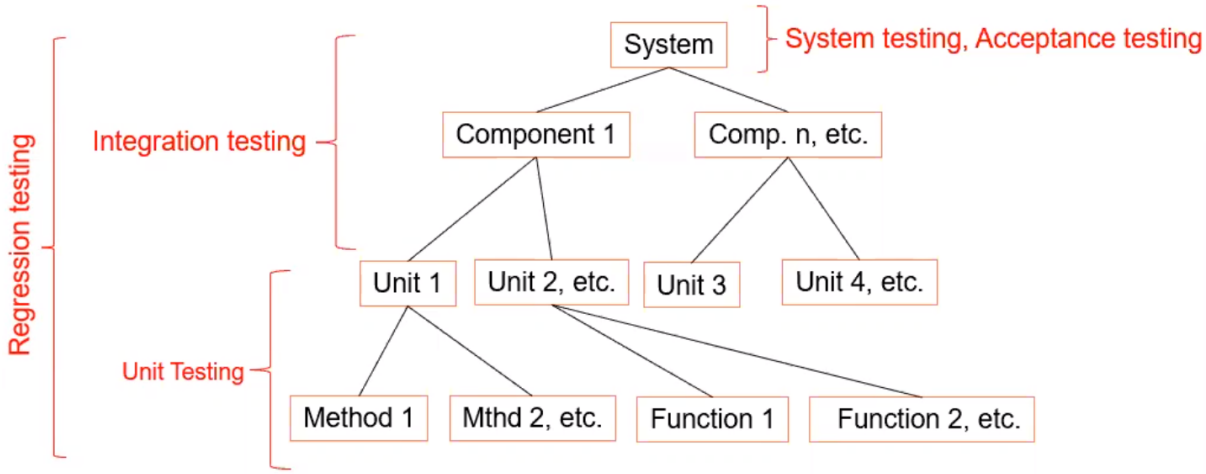
In some cases, the main objective of testing may be to assess the quality of the software to give information to stakeholders.
This happens in regression/ maintenance testing. This often includes testing that no new errors have been introduced during the development of the changes.

During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Types of Dynamic Testing

Software System Decomposition

Unit testing, Integration testing, System testing Acceptance testing, Regression testing



Unit testing addresses the quality of individual components (e.g methods, functions)
Integration testing checks if individual components operate together
System Testing checks if if the system is as a whole delivering the desired functionality
Acceptance testing allows users to establish if the system is acceptable to the users' requirements

General Testing Principles

Principle 1 - Testing shows the presence of defects
Testing can show that defects are present but cannot prove that there are not defects

Principle 2 - Exhaustive testing is impossible
Exhaustive testing all inputs is impossible, it would take years

Principle 3 - Early testing
Testing activities should start as early as possible in the software development life cycle. If you find defects earlier in the development life cycle the cost of correction is significantly lower than if you find it later.

Principle 4 - Defect clustering
A small number of modules contains most of the defects

Principle 5 - Pesticide paradox
If the same tests are repeated again and again, eventually the test cases will no longer find new defects.

Principle 6 - Testing is context dependable
The testing we do must take account of the context

Principle 7 - Absence of errors fallacy
A system may have a good specification, and be tested thoroughly but it is still possible for the system to not meet the needs of its users

Other Principles to keep in mind:

1. The goal of testing is to find defects before customers find them.
2. Understand the reason behind the test.
3. Test the tests first.
4. Corrections have side effects. (Regression testing)
5. Defects occur in convoys or clusters, and testing should focus on these convoys
6. Testing encompasses defect prevention. (If we do walkthroughs, we can spot defects and fix them before we actually execute the program)
7. Testing is a fine balance of defect prevention and defect detection.
8. Intelligent and well-planned automation is key to realise the benefits of testing.
9. Testing requires talented, committed people who believe in themselves and work in teams

The Psychology of Testing

A tester employs professional cynicism. Must try to prove that the solution doesn't work. A tester must be sceptic. They do not make assumptions. The developer has a sense of ownership because they wrote the code. A tester is unbiased.

Communication between testers and developers:

- Collaboration
- Neutral, objective, avoid personalities
- Understand how the other person may feel
- Confirm that the person has understood what you have said and vice versa

The goal is to locate defects
If we are effective at finding defects and then we can't find any, we can be confident the system works

Tester mindset
"Tread lightly as you tread on their dreams"
Must be pedantic, sceptical to software
Trust the developer but doubt the product
Do not have confidence in the product until you've tested it

Independence

Fundamental Test Process

A test is a controlled exercise involving an object under test, a definition of the environment, a definition of the inputs and a definition of expected outputs or result

The Test Process
The fundamental test process consists of:

- Test planning
- Test control
- Test analysis
- Test design
- Test implementation
- Test execution and recording
- Evaluating exit criteria and reporting
- Test closure activities

Although the steps are logically sequential, the activities in the process may overlap or take place concurrently.

1. Test Planning

Test planning involves:

- Determining the scope and risks and identifying the objectives of testing
- Determining the test approach(techniques, coverage, test ware)
- Determining the required resources
- Implementing the test policy and/or the test strategy
- Scheduling test analysis and design tasks
- Scheduling test implementation, execution, and evaluation
- Determining the exit criteria

2. Test Control

Test control involves:

- Measuring and analysing results
- Monitoring and documenting progress, test coverage and exit criteria
- Initiation of corrective actions
- Deciding what to do next

3. Test analysis

Test analysis is the activity where tangible test conditions and test designs are derived from the baseline documents

Baseline documents: requirements documents, architecture documents, design documents, interfaces

4. Test design

- Designing the test environment set-up and identifying any requirements infrastructure and tools.
- Test inventory is prepared:
    - The features to be tested
    - Logical test cases to be exercised
    - Test case prioritization, where necessary

5. Test Implementation

- Test implementation is where the test conditions are transformed into test cases and test ware, and the environment is set up
- We identify and create test data, write procedures, prepare expected results, and automate the tests

6. Test execution and recording

- Perform you “Pre-flight checks”
- Test cases are run either manually or by using test execution tools, according to the planned sequence

7. Raising incidents, re-testing, and regression testing

We report test failures or discrepancies as incidents and analyse them to establish their cause

We then do retests or regression tests

8. Evaluating exit criteria

- Evaluating exit criteria is the activity where the test execution is assessed against the defined objectives
- Exit criteria is asking if we have tested enough, you might then change the criteria to do more tests
- This should be done for every test, regardless of phase

If you are under time pressure you might decide that some faults are acceptable to maintain in the system or you did not have time to run all the tests you planned: you then change your criteria calculating the risk involved.

9. Test closure activities

A test summary must be written for the stakeholders at the end of the test phase
The summary provides stakeholders the evidence they need to make a decision (to release, postpone, or cancel)

Use the lessons learned in future testing and projects

Baseline and exit criteria

Expected results

Our baselines (Specifications, requirements etc.) define what the software is required to do
The expected results are defined before we execute the test

A baseline document describes how we require the system to behave

The expected results are defined before we execute the test, derived from the baseline

Exit criteria is to trigger to say " We've done enough"
The exit criteria should be a measurable, achievable target.
Criterion extensively used:

- 80% coverage
- All tests executed without failure
- All critical business scenarios covered
- All outstanding incidents are waved

Coverage items are usually defined in terms of requirements, conditions business transactions, code statements

Software Development Models

Software Process defines the way to produce software. It includes:

- Software life-cycle model
- Tools to use
- Individual building software

Software life-cycle model defines how different phases of the cycle are managed

Software Development Life Cycle (SDLC)

Phases of Software Development

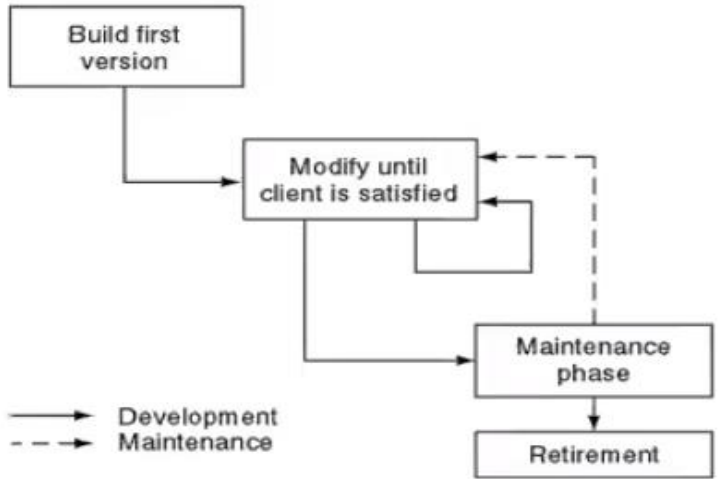
1. Requirements gathering
2. Specification
3. Design
4. Code
5. Testing
6. Implementation

These phases can overlap

Life-Cycle Models

Build-and-fix model
Waterfall model
Rapid prototyping model
Spiral model
Phased Development model
Agile models (Scrum)

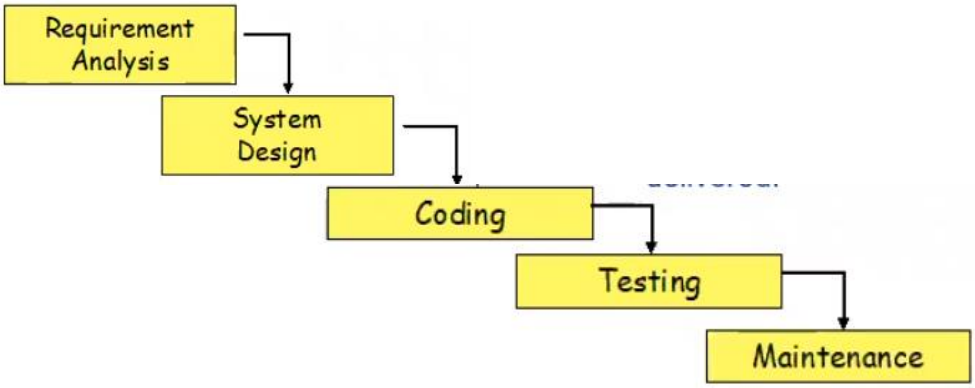
Build-and Fix Model



Used in student projects. You circle through until you're happy
No specification or design. Unsatisfactory

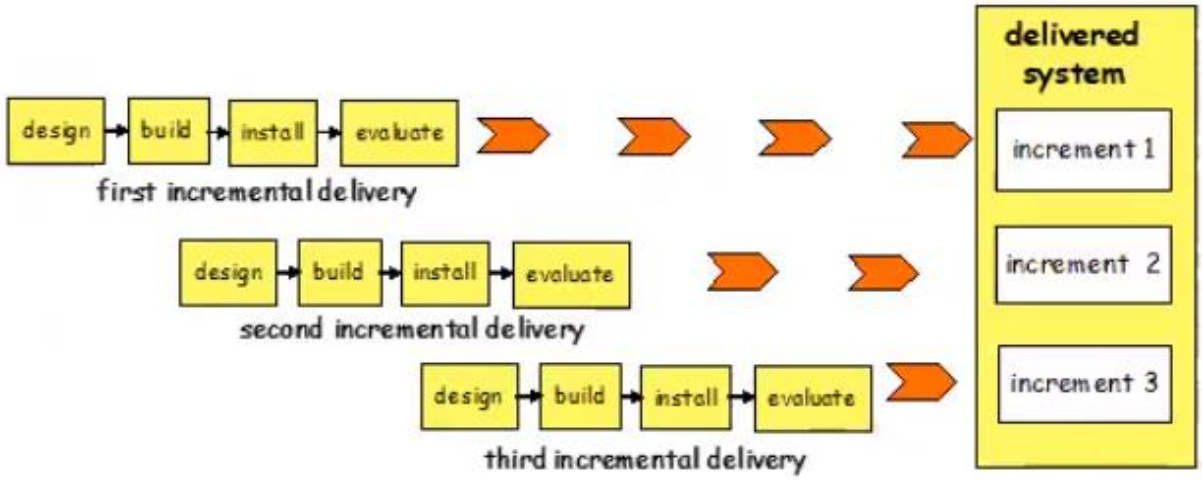
Waterfall model

Output from one phase is the input for the next phase
Once one phase is finished and signed off you move into the next phase
Each phase is very well documented but if there is a mismatch between what you're building and what the client wants, it is very extensive to fix



Iterative development method is the process of establishing requirements , designing, building, and testing a system, done as a series of smaller developments. Regression testing is important on every iteration

Incremental Model



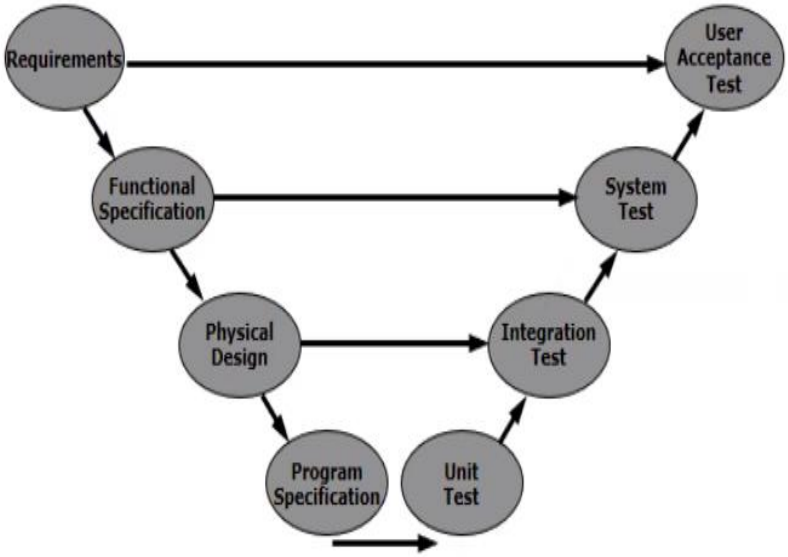
Break the system into smaller components, gradually increased functionality

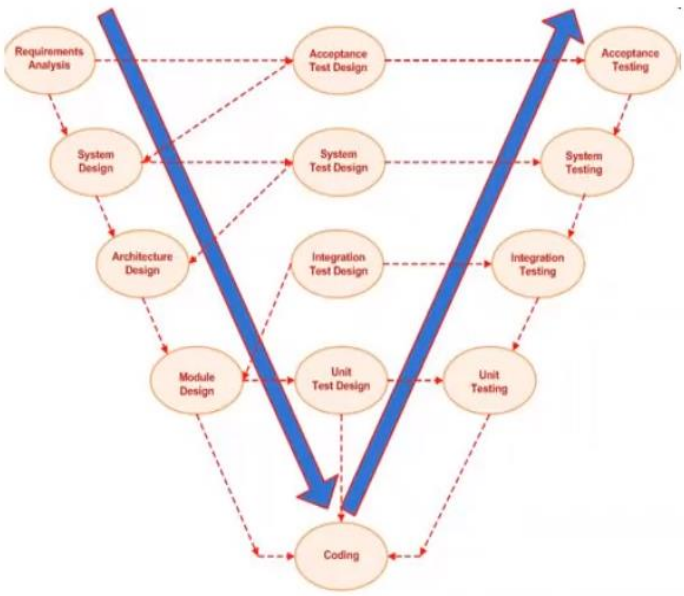
Testing within a life cycle model

We divide and conquer the task of testing
Testing builds up as we progress through the various stages

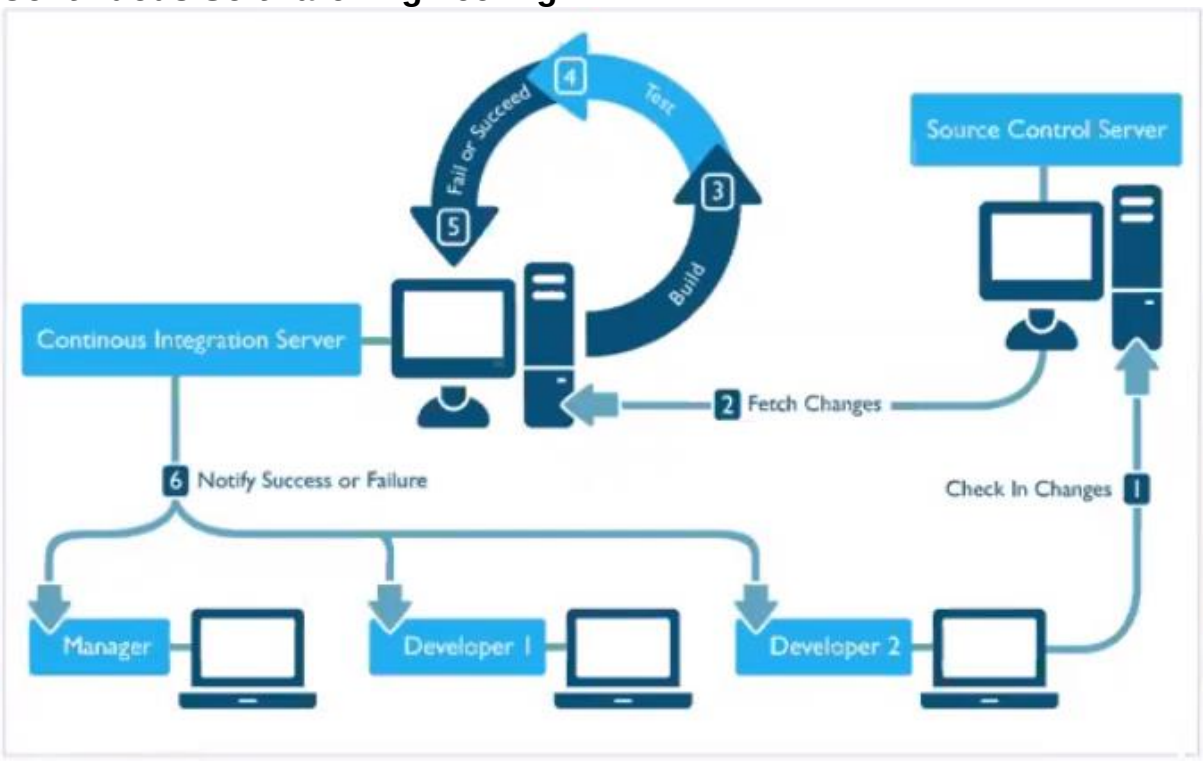
The V-Model (Verification and Validation)
Four common levels are:

- Component (unit) testing
- Integration testing
- System testing
- Acceptance testing





Continuous Software Engineering



Examples of Tooling Used in CSE

Phase	Tool
Development	Fuge - http://fuge.io/
	Seneca - http://senecajs.org/
	HAPI - http://hapijs.com/
Repository	Git / github - https://github.com/
	BitBucket - https://bitbucket.org/
Build	Jenkins - https://jenkins.io/
	Drone - https://drone.io/
Test	Tap - https://testanything.org/
	Phantom - http://phantomjs.org/
	Pdiffy - https://github.com/kennychua/pdiffy
Infrastructure	EC2 - https://aws.amazon.com/ec2/
	ELB - https://aws.amazon.com/elasticloadbalancing/
	Consul - https://www.consul.io/
Deployment	Docker registry- https://docs.docker.com/registry/
	Code Deploy – https://aws.amazon.com/codedeploy/
	Swarm - https://docs.docker.com/swarm/
	Kubernetes - http://kubernetes.io/

Automation and tooling doesn't replace existing life cycles

Influences in the test process
We need to think about:

- The type of faults to test for
- The object under test
- Capabilities of developers , testers, users
- Availability of environment, tools, data, resources
- The purpose(s) of testing

Test Design Techniques

The purpose of test design techniques is to identify test conditions and test cases.

Why use techniques?

- Exhaustive testing of all program paths and inputs is impractical
- If we did test every input, we would begin to get duplicates. After a while we would find any more defects.
- Therefore, we need to select tests which are
  - Effective at finding results
  - Efficient

Category of Test Design Techniques

Specification based Black Box

- Deriving test cases directly from a specification or model of a system/proposed system (Baseline documents)

Structure based White Box

- Deriving test cases directly from the code written (or design) to implement a system (We can look at the code and create tests)

Experience based

- Deriving test cases from the tester's experience of similar systems and general experiences of testing (They would have knowledge about likely defects and distribution)

Black Box Techniques

Equivalence Partitioning

Equivalence partitioning is the process of methodically reducing the large set of possible test cases into a small, but equally effective, set of test cases.

- Can be applied at all test levels
- Recommended as one of the first techniques to use
- It is based on dividing a set of test conditions into groups (partitions) that can be considered the same, that is, the system will treat them equivalently
- We choose one condition from the partition and assume that all the other conditions in that partition will work the same
- The result of testing a single value from an equivalence partition is considered representative of the complete partition.

Valid and Invalid

Equivalence partitions may also be treated in terms of valid and non-valid partitions.

Invalid partition values may still be entered by the user, but they are not expected by the system. In this case we must ensure that the software will be able to handle these values by displaying the appropriate error message

Validity rules can be a range, count, set or a must

Test Case Strategy

Once the set of equivalence classes has been identified, we derive test cases:

Assign a unique identifier to each equivalence class

Consider creating an equivalence class partition that handles the default, empty, blank, null, zero and none

Boundary Value Analysis
Experience shows more faults at boundaries of equivalence partitions where partitions are continuous.
Boundary value analysis focuses on the development of test cases that are above below and on the boundaries of the equivalence partitions.
Choosing a boundary can be subjective

Test Organisation

Who does the testing?
Programmers do the ad hoc testing (build and fix)
Users (with support) perform the user acceptance testing
Independent organisations may be called upon to do any test phase

Test team roles

Test manager (test lead)
- Plan, organise, manage and control the testing
- Project manager for testing activities

Tester
- Prepare test procedures, data, environments, expected results
- Execute tests and log incidents
- Execute re-tests and regression tests

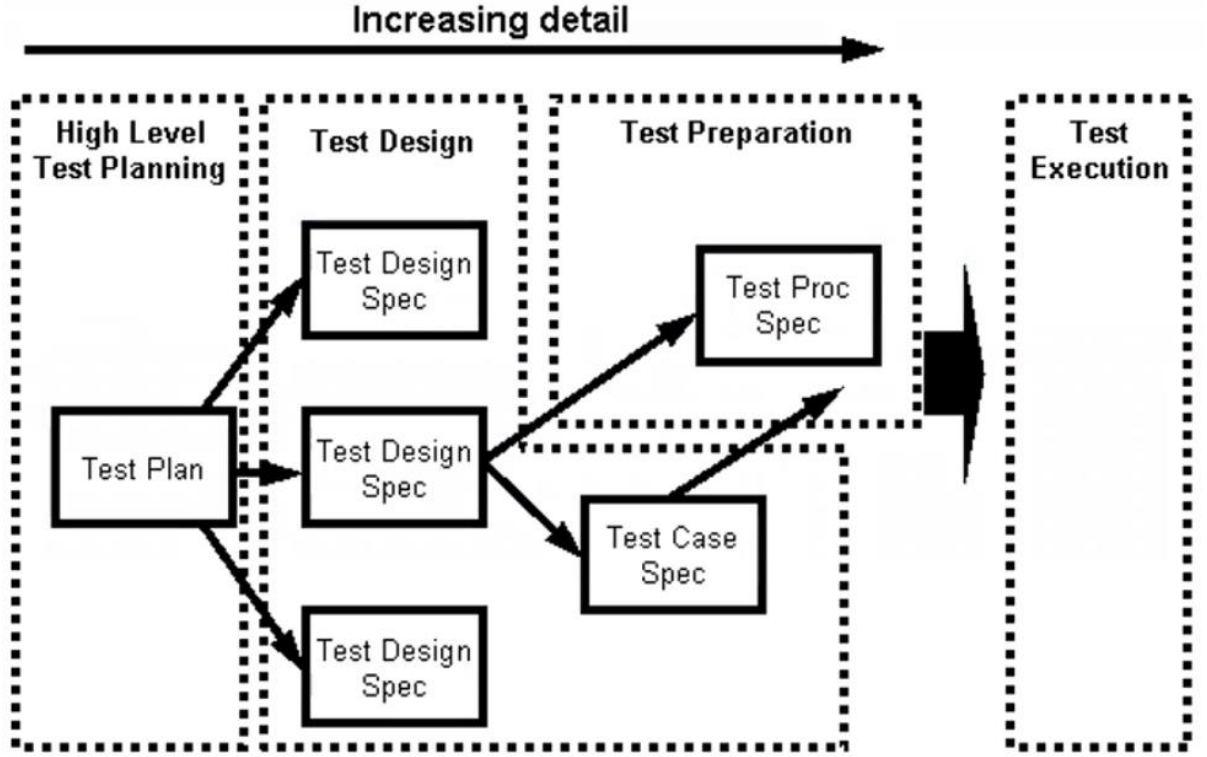
Test analyst
- To scope the testing, conduct expert interviews
- Document test specifications
- Liaison with the test environment manager, technicians, users, testers
- Prepare test reports

Test automation technician
- Record, code and test automated scripts
- Prepare test data, test cases and expected results
- Execute automated scripts, log results, and prepare reports

Support Staff
- Database administrators
    - help find, extract, and manipulate test data
- Tool smiths
    - build utilities to extract data, execute tests, compare results
- Experts

- Provide direction in terms of technicalities and business

Test Planning and Estimation



The further right we move on the graph the more detail is involved

Test Planning
Planning is influenced by the test policy of the organisation, the scope of testing, objectives, risks, constraints, criticality, testability, and the availability of resources.

Test Planning activities

- Defining the overall approach to testing
- Integrating testing activities into the plan
- Scoping, assigning responsibilities, setting entry/exit criteria
- Assigning resources for the different tasks defined
- Defining the amount, level of detail, structure, and templates for the test documentation
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues
- Setting the level of detail for test procedures to provide enough information to support reproducible test preparation and execution.

Exit Criteria

The purpose of exit criteria is to define when to stop testing, such as at the end of a test level or when a set of tests has a specific goal.

Typically exit criteria may consist of:

- Thoroughness measures, such as coverage of code, functionality, or risk
- Estimates of defect density or reliability measures
- Cost
- Residual risks, such as defects not fixed or lack of test coverage in certain areas
- Schedules such as those based on time to market

Exit criteria should be objective and measurable

- All tests run successfully
- All faults found are fixed and re-tested
- Code Coverage target (set and met)
- Time or cost limit exceeded

Coverage items are defined in terms of:

- Requirements, conditions , business transactions
- Code statements, branches

Component Integration Test completion criteria

Structural coverage

- 100% coverage of transfers of control or data between components in scope

Functional coverage

- 100% equivalence partitions for all message and function call parameters (valid and invalid values)
- 100% valid transfers of control and data
- 100% invalid/erroneous transfers of control or data

All tests, passed without failure
All faults raised, corrected, retested, signed off.

Typical (Functional) System Test completion criteria

Functional coverage

- 100% equivalence partitions
- 100% boundary values
- 100% transactions flows identified in Functional Specification

All tests, passed without failure
All faults raised, corrected, retested, signed off or waived by Product Assurance Team.

Acceptance test completion criteria

Acceptance criteria are business oriented

- 100% of all critical system, features
- 100% branch testing through high priority business processes
- 100% process coverage of medium priority business processes
- 100% coverage of low priority business processes (at least once)

All test (deemed high priority) passed without failure
All faults raised, corrected, retested, signed off or waived by Testing Board

Defect Severity & Priority

Defect severity is the extent to which the defect can affect the software.

Ranking of severity types (Highest to lowest):

1. Critical:
    - The defect that results in the termination of the complete system or one or more components of the system and causes extensive corruption of the data. There is no acceptable alternative method to achieve the required results.
2. Major:
    - The defect that results in the termination of the complete system or one or more components of the system and causes extensive corruption of the data. The failed function is unusable but there exists an acceptable alternative method to achieve the required results.
3. Moderate
    - The defect that does not result in the termination, but causes the system to produce incorrect, incomplete, or inconsistent.
4. Minor
    - The defect that does not result in the termination and does not damage the usability of the system and the desired results can be easily obtained by working around the defects.
5. Cosmetic
    - The defect that is related to the enhancement of the system where the changes are related to the look and feel of the application.

Defect Priority defines the order in which we should resolve a defect

Ranking of defect priority types:

1. Low:
    - The defect is an irritant which should be repaired, but repair can be deferred until after more serious defects have been fixed
2. Medium

    - The defect should be resolved in the normal course of development activities. It can wait until a new build or version is created.
3. High
    - The defect must be resolved as soon as possible because the defect is affecting the application or the product severely. The system cannot be used until the repair has been done.

We rank each defect in terms of both severity and priority
High Priority & High Severity

Decision Time
When deadline arrives, may have to stop tests now

- Some faults may be acceptable
- Some tests may not be run at all

Might have to relax the exit criteria - assess the risk and agree with customer and document the changes
(Release software not but continue testing)
If exit criteria met the software can be release

What if you finish the test plan early?
Ask yourself, have you done enough?
Strengthen the exit criteria, perhaps create more tests

Test Estimation

Why do we need to estimate?

The test manager wants to predict:

- How long the testing will take
- How much effort will be needed?
- When you will need resources and experts

Resources can be testers, external consultants or services, tools, environments, hardware, or software

What is an estimate?
What resources are required:

- People, tools, test environments
- How long will it take?
- What is the resource utilization? Full time, 50%, occasionally?
- What is the lead time for obtaining that resource?

Test Estimation is an approximate calculation based on:

- Direct experience
- Related experience
- Informed guesswork

Break down large complex tasks into smaller, simpler tasks to start with
You need to consider:

- Double accounting
- Contingencies

What gets forgotten in Estimation?

- Training
- True costs
- Reading time “getting up to speed”
- Meetings and interviews
- Q&A’s and reviews
- Contingency
- Reallocation of people, resources, desks, pcs etc

Contingency is identifying the risk responses and allowing more time and resources to allow that
Agree with a percentage contingency for risky tasks
It’s important that we allow time for all stages of the test process and overtime
Don’t underestimate the time taken
Testing rarely goes smoothly
Assume 2 - 3 passes through a system test

Problems in estimating:

- Can't predict how many faults will occur
- Can't predict severity of defects we find
- Can't predict when testing will stop
- Can't predict how long faults will take to fix

Test Approaches (Strategies)

Preventative VS Reactive Test approaches

Preventative approaches are where tests are designed as early as possible.

Reactive approaches are where test design comes after the software or system has been produced.

When Selecting an approach, we need to consider:

- The risk of failure of the project, hazards to product and risks of product failure to humans, the environment, and the company (Is there risk to human life?)
- The skills and experience of the people in the proposed techniques, tools, and methods (Are the people we are working with familiar with the approach we are taking?)
- The objective of the testing and the mission of the testing team
- Regulatory aspects, such as external and internal regulations for the development process

- The nature of the product and the business

Test Progress Monitoring and Control

The purpose of test monitoring is to give feedback and visibility about test activities
Information to be monitored may be collected manually or automatically and may be used to measure the exit criteria, such as coverage
Metrics may also be used to assess progress against the planned schedule and budget

Common test metrics

- Percentage of work done in test case preparation
- Percentage of work done in test environment preparation
- Test case execution
- Defect information
- Test coverage requirements, risks, or code
- Testing costs, including the cost compared to the benefit of finding the next defect or run to the next test

Test reporting
Test reporting is concerned with summarizing information about the testing endeavour, including:
What happened during a period of testing, such as dates when exit criteria were met?
Analysed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit.

Test control
Test control describes any guiding or corrective actions taken because of information and metrics gathered and reported.

Examples of test control actions are:

- Reprioritise test when an identified risk occurs
- Change the test schedule due to availability of the test environment
- Set an entry criterion requiring fixes to have been retested by a developer before accepting them into a build

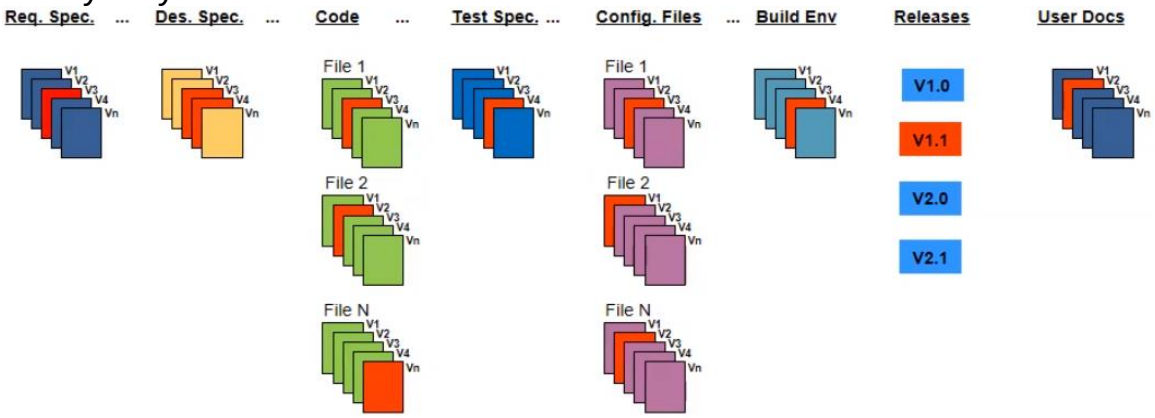
Configuration Management

For testing, configuration management may involve ensuring that:

- All items of test ware are identified, version controlled, tracked for changes, related to each other, and related to development items

- All identified documents and software items are referenced unambiguously in test documentation

Everybody should have the same versions of test ware and documentation.



Poor configuration examples:

- Can't find latest version of source code
- Bugs that were fixed suddenly reappear
- Wrong code is tested
- Tested features suddenly disappear

Configuration Management answers:

- What is the current software configuration?
- What is its status?
- How do we control changes to our configuration?
- What changes have been made to our software?
- Do anyone else's changes affect our software?

Risk and Testing

Stakeholder Objectives and Risks

A stakeholder objective is one of the fundamental objectives of the system to be built

Software risks are potential events or things that threaten the cardinal objectives of a project

A risk is a threat to on or more of the cardinal objectives of a project that has an uncertain probability

Risks only exist where there is uncertainty. If something has a 0% or a 100% probability of occurring, it is not a risk

Risk Examples:

- Safety Critical risk (Aircraft system fails)
- Political (Immigration system fails)
- Economic or financial (Banking system fails)

- Technical (Memory allocation)
- Security (Security vulnerabilities, fraud, hackers)

Three types of software risk

- **Project risk**
  - Primarily a management responsibility:
  - Resource constraints, external interfaces, supplier relationships, contract restrictions

- **Process risk**
  Planning and the development process are the main issues here:
  - Variances in planning and estimation, shortfalls in staffing, failure to track progress, lack of quality assurance and configuration management
  - Poor planning, poor monitoring, poor control

- **Product Risk**
  Requirement's risk are the most significant risks reported in the risk assessments:
  - Lack of requirements stability, complexity, design quality, coding quality, non-functional issues, test specifications.

Risk-Based Testing
Risks are used to decide where to start testing and where to test more; testing is used to reduce the risk of an adverse effect occurring, or to reduce its impact.

To ensure that the chance of a product failure is minimized

Incident management

The objective of incident reporting is:
- To provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
- To provide test leaders a means of tracking the quality of the system under test and the progress of the testing
- To provide an input to test process improvement initiatives.

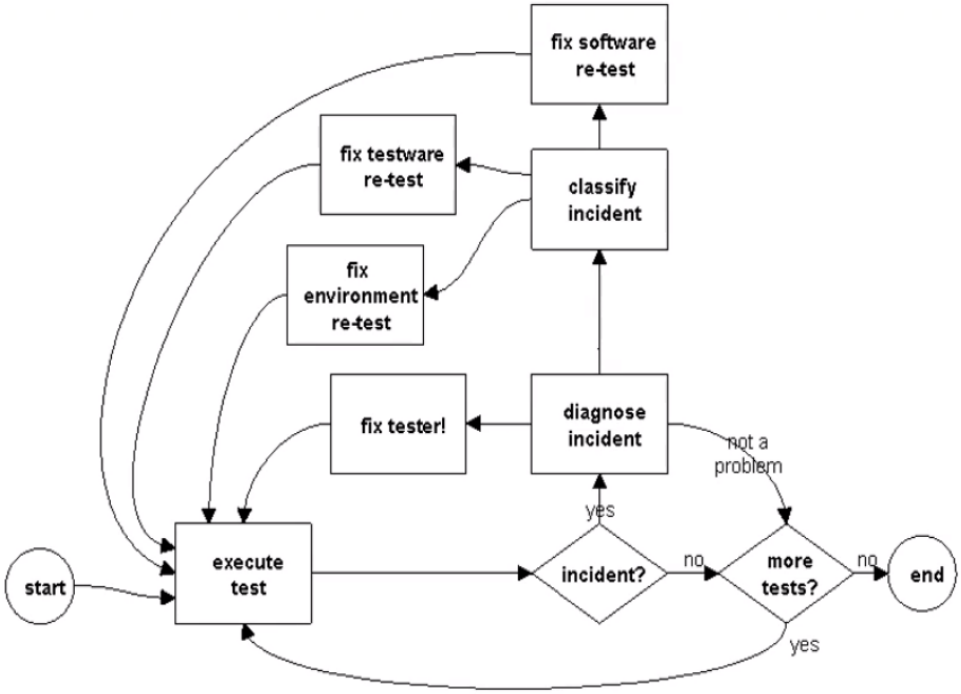
We log incidents when a test result appears to be different from the expected result.
This could be because:
- The test is written incorrectly
- Our expected results are wrong

- The result could be misinterpreted
- Something is wrong with the baseline
- Something is wrong with the test environment
- Software fault

The tester should stop and complete an incident report:

- Describe exactly what is wrong
- Document the test script and script step
- Attach any outputs that may be useful (Screen shots, printouts)
- Impact on your current tests

Typical test execution and incident management process



Bug Tracking software:

- Reporting facilities
- Assigning
- Workflow
- History/ work log/ comments
- Reports
- Storage and Retrieval

Pair Programming

Pair programming refers to the practise where two programmers work side by side on one computer , collaborating on the same design, algorithm, code or test.

It's an **agile development** method.

Started in the 1940s by the ENIAC women (Betty Snyder and Jean Bartik)

Two reasons why it isn't really used:
It's a long game, results are seen after a long time
It's hard

Driver:
writes code
Navigator:
makes suggestions, asks questions

Benefits of pair programming:
Easier to work problems out with two people
Better than working alone
Better understanding if you have someone to brainstorm with
Fun
Prep for real world collaboration and group work

Do's:
Always be talking and communicating with your partner
Always listen, be engaged in the project
Switch roles periodically, helps to prevent errors
Be patient, explain to partner, to teach is to learn
Respect your partner
Take breaks
Be prepared

Don'ts:
Don't hog the keyboard, even if you are better than them
Don't be bossy
Don't be intimidated of partner
Don't be quiet, speak up if you have problems

Effectiveness:
- Knowledge sharing
- Combine 2 modes of thinking (Tactical [Driver] and strategic[Navigator])
- Reflection

Focus:
- Forced to explain and justify why you do what you do
- Focus for the team

True continuous integration

Challenges:
- Exhausting and energy draining

- Don’t pair 8 hours a day, take breaks, switch roles and modes

Collaboration:
- Feedback
- Exchange READMEs
- Awareness of each of your differences

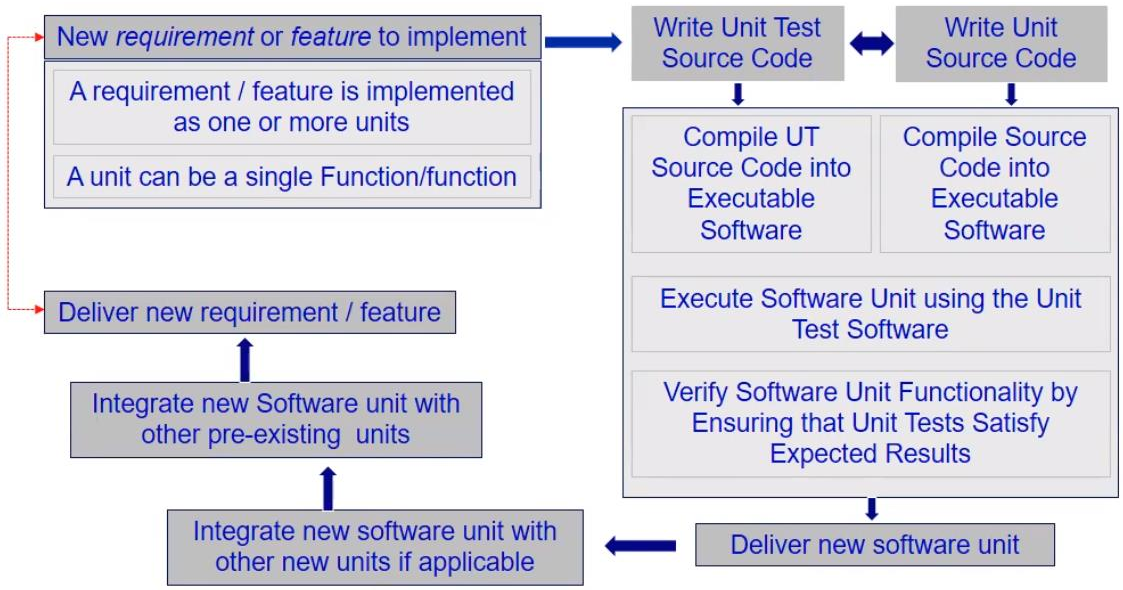
Pair programming helps you with Concentration, Task organization, Time management, Communication, Giving & receiving feedback

Diversity on a team is good.
Pomodoro technique

Unit Testing

PyUnit is a unit testing framework for Python
Plays a crucial role in Test-driven development (TDD)
Advocates the idea of “testing first then coding”
Part of python standard library (import unittest)
E.g assertEqual() to check if an actual result corresponds to the expected results
A pyunit test case is a code that checks that another
- Set up the test data for a piece of code
- Test the test setup
- Write the code
- Unit test the code using test setup

Why use unit tests?
- Unit testing can increase programme productivity
- Unit tests help developers find defects immediately
- Unit tests help developers to verify that the logic / behaviour of a piece of code is correct
- Re-performing unit tests can identify software regressions introduced by changes in the code
- Unit testing frameworks can be used for integration and system level testing
- May not be sensible to write tests for all code



Continuous integration (CI) systems

- Continuous integration is a development practise that requires developers to integrate code into a shared repository several times a day
- Each check in is verified by a automated build allowing teams to detect problems early
- By integrating regularly, you can detect errors quickly and locate them more easily.

A unit test case is characterized by:

- A known input
- An expect output

Both inputs and expected outputs are defined before the code is written

What are positive and negative test cases?

- Positive test cases - test the behaviour of the code using data that is expected to be received in a correct operational mode testing for something that should happen, does happen
- Negative test cases are - test the behaviour of the code when incorrect data is entered (how does the code handle instances of incorrect data)

Important to adapt both perspectives testing for something that shouldn't happen and does not happen

Unit test tips:
Provide meaningful and accurate messages in assert statements:

- Fixing issues will be easier
- It may not be the same person running the tests that made the code, so make sure your assertions, messages and comments are well described

Testing Software Requirements

Importance of requirements
Poor requirements occur because clarification and elicitation is challenging and developers don't invest sufficient time and effort into requirements

Problems
We use natural language to express most things
We assume that:

- We have a sufficient complete understanding of some topics
- We have communicated our understanding completely to other concerned parties
- Others have completely understood our communication

There are many gaps in natural language, communication and understanding
Pay attention to language used
Ask questions like "what do you mean by .."

Requirements
Abstract concepts
Converting abstract concepts into reliable working software is very challenging - there are many pitfalls and hurdles along the way.

Agile and non-agile worlds?
When compared with traditional approaches, agile software development breaks development down into much smaller development cycles and increases release frequency. We increase functionality as we progress

It is designed to avoid lengthy phases of requirements documentation providing users with feedback regularly.

Implemented correctly, agile software development can help produce solutions that users actually want by building up small increments of functionality regularly and not going too far off course during implementation. Each time requirements are defined and code is written, it should be considered as an economic proposition

Simple requirement (Paris to Cannes)
Requirements should be critically evaluated.

Testing Requirements

Evaluating quality of requirements
Is each quality requirement:

- Mandatory (Is it relevant?)
- Feasible (Does it make sense to do this, "The system shall be secure and reliable")
- Scalable (clear how much quality is required)

- Unambiguous (Everyone can interpret it the same way)
- Verifiable (Can it be tested or demonstrated?)
- Correct (Meets needs of stakeholder)
- Prioritized (Do we understand how critical)
- Traced to its source
- Rationale (Understand why a requirement is being implemented)

Requirements Quality - Collective:

- Overall
- Complete (No information or requirement missing)
- Consistent (no conflict with other software requirements)
- Modifiable (Can revise requirements when necessary)
- Traceable (Link each software requirement to its source)

Critical analysis
Think about the overall intention/purpose of the system
Keep in mind the constraints that the system could be subject to
Don't overlook practicalities and inconsistencies
Ask: is sufficient information provided?
Consider user stories individually and collectively

Test Levels, Decomposition, Stubs and Drivers

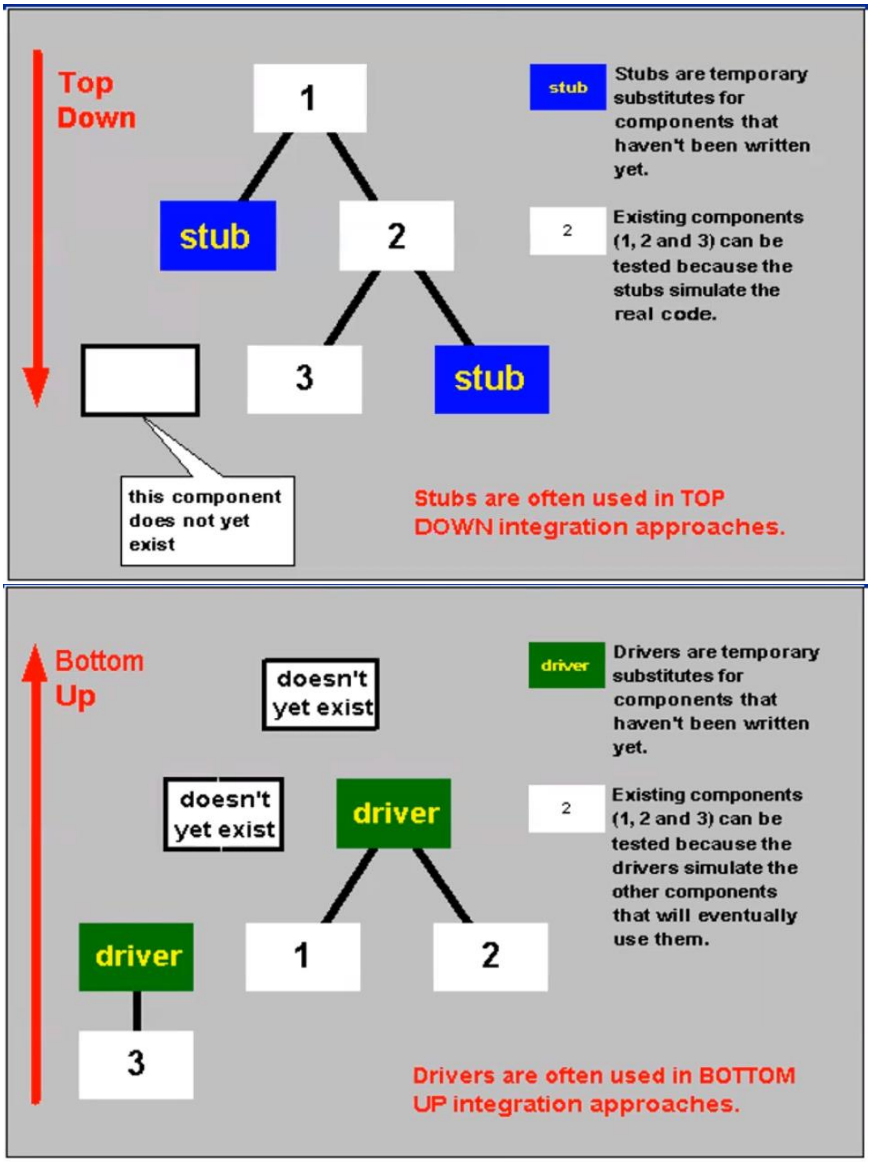
Component testing

A component can be a angle unit or a collection of units
Performed on components in isolation - generally we don't have full functionality of the component so stubs, drivers and simulators may be used
May include testng of some non-functional characteristics.

Integration testing

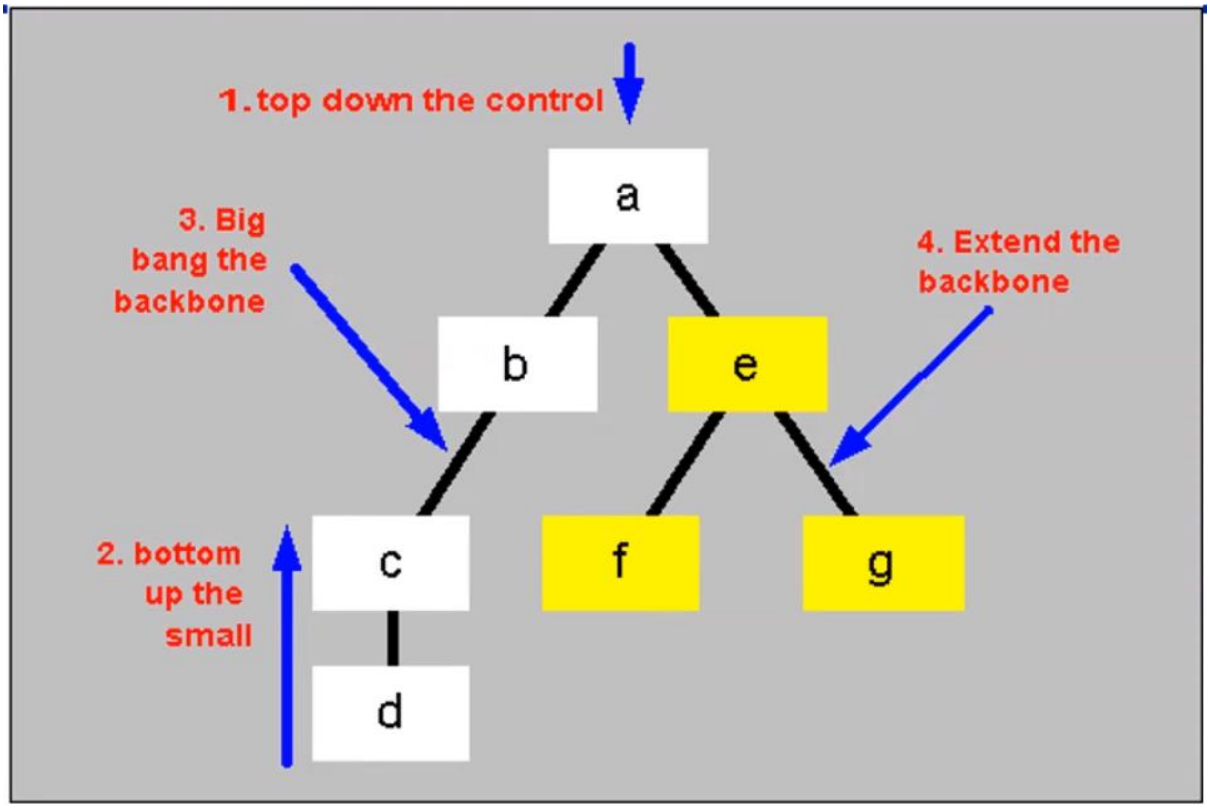
May components may be integrated into a large system
May need to perform some dedicated integration testing
Incremental approach may be used

Stubs are temporary substitutes for components that haven't been written yet. (Top down testing).
Drivers are temporary substitutes for components that haven't been written yet (Bottom up testing).



Stubs and drivers take many forms:
They can be a script that interacts with the software, A separate program, or a 3rd party product.

Mixed integration strategy (big bang)



Error-Guessing

An approach to testing based on intuition and guesswork with experience of the tester.
Good test cases can be derived in this way
Error-guessed test cases should be documented as part of the test plan

Exploratory testing (ET)
Where the documents that form the basis for the test design are of low quality or obsolete or do not exist.
No application of a structured test process

We become exploratory when:

- The next test is influenced by the result of the last test
- We can't tell what tests should be run in advance
- We haven't had the opportunity to create those tests

When to use ET?
When normal testing is suspended, or you believe more focused tests in an area will be productive

Choosing test techniques
Poor test design basis documents - ET

Alpha and beta testing
- Assess the reaction of marketplace to the product
- Any major features missing
- Is the product ready for release?

Path Testing and coverage

Path testing is concerned with test cases that cause chosen paths
It's primarily applicable to component testing
It requires intimate knowledge of the program structure e.g source code

When a program is executed, it can follow various paths
In path testing we are concerned with identifying possible paths through code and identifying test cases with knowledge of the possible paths in advance
Path testing helps to raise our confidence in a piece of code because the more possible paths we have tested the more certain we can be about the behaviour of the system

Coverage

Statement coverage
- A statement is executable code that does something
- Most basic
- Every statement executed at least once

Branch coverage (BC)

-
- More refined
- Every outcome of every decision executed at least once

Coverage measurement
SC = (statements executes / total statements) * 100
BC = (branch outcomes executed / total branch outcomes) * 100
PC = (paths executed / total number of paths) * 100

Procedure for statement testing:
- Identify all executable statements
- Trace the execution from the first statement
- For each decision, choose the true outcome first; write down values for the variables in the predicates that make it true
- Are all statements covered? If not, select test values to reach the next uncovered statement in te code and continue covering by at least one test case

- Repeat until all statements covered at least one test case
- For all test cases, record the variable required to force the execution path you take

Path coverage is most difficult as it's concerned with testing all possible outcomes from all possible statements. Different combinations.

From paths to test cases:

The process of choosing input values is called “sensitising the path”

Data Flow Analysis

Data flow analysis is concerned with how data is used on different paths through the code
There are three different usage states for each data variable:

- Undefined (u) The data has no defined value
- Defined (d) The data is assigned a value
- Referenced (r) The data is used

Data flow analysis cannot detect errors
But it can identify anomalies:

- ur-anomaly - An undefined data item is read in a program oath
- du-anomaly - A data item that has been assigned a value becomes undefined without being used
- dd-anomaly - A data item that has been assigned a value is assigned another value without being used in the meantime

Control Flow Graphs

Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back o the loop condition node

The objective of path testing is to ensure that the set of test cases is such that the desired paths through the program are executed at least once

Flowgraphs consist of three primitives:
A decision is a program point at which the control can diverge
(e.g if)
A junction is a program point where the control flow can merge
(e.g end if, end loop)
A process block is a sequence of program statements uninterrupted by either decision of junctions (i.e straight lines of code)

A process has one entry and one exit
A program does not jump into or out of a progress

A path through a program is a sequence of statements that starts at an entry, junction or decision and ends at another junction, decision or exit.
Paths consist of segments

The smallest segment is a link. A link is a single process that lies between 2 nodes
The length of a path is the number of links in a path