Software Testing

- **Unit Testing**: Test individual components or functions isolation
- **Integration Testing**: Test interactions between multiple unit-tested components
- **Regression Testing**: Ensures that new code changes do not break existing functionality. Test suite contains:
  - **General functional** tests: Small but meaningful set of existing test cases that cover all major features of the software
  - **Impact-based** tests: Additional tests targeting components most likely to be affected by recent changes
  - **Change-specific** test: Existing tests focusing on the exact components that were modified
- **Smoke Testing**: Quick, high-level test to check if the basic/major functionalities of a software build work before running deeper tests
- **Validation Testing**: Ensures that software meets user requirements and performs as intended
- **System Testing**: Performed on completely integrated system to ensure that the system works correctly as a whole
- **Black-box Testing**: Internal code or implementation is not known to the tester
- **White-box Testing**: Tester has knowledge of the code, logic and structure of the system

Unit Testing

- A unit test is an automated test that verifies a small piece of code (unit) quickly in an isolated manner
- Faster debugging, faster devlopment, reduce future cost
- Structure (AAA pattern):
  - **Arrange**: Set up all required test objects and prepare prerequisites, i.e. set up the test environment (e.g. create mocks, init objects)
  - **Act**: Execute the code that is being tested (e.g. call methods of an SUT)
  - **Assert**: Verify that outcome of unit test matches expectations (using assertion methods)

```
describe("MathUtils Tests", () => {
   it("should return correct Fibonacci Number", () => {
      // Arrange
      const mathUtils = new MathUtils();
      // Act
      const result = mathUtils.getFibonacciNumber(6)
      // Assert
      expect(result).to.equal(8)
   }
}
```

- Properties of good unit tests:
  - **Protection against regressions**: If there is a regression, the tests should reveal it. It avoids cases where test cases pass but functionality is broken (false negatives, type II errors)
  - **Resistance to refactoring**: Refactoring application code should not break tests. It avoids cases where test cases fail but functionality is correct (false positives, type I errors)
  - **Fast feedback**: Tests should run quickly to catch bugs faster
  - **Maintainability**: Tests should be small and readable. External dependencies (e.g. databases or APIs) should be mocked
- Test doubles:
  - **Stub**: Returns pre-defined (hardcoded) data. Don't need to care about its behaviour
  - **Mock**: Stub but with behavioural expectations (e.g. testing how many times a method is called). Typically used to test interactions
  - **Fake**: Simplified, real implementation but not production level (e.g. in-memory DB instead of a real one)
- Styles of unit testing:
  - **Output-based**: Feed input to SUT and check output
  - **State-based**: Verifying state of SUT or its collaborators or any out-of-process dependency (e.g. DB) after completing an operation
  - **Communication-based**: Use mocks to verify communications between SUT and its collaborators

Code Coverage

- **Statement coverage**: % of lines of code executed at least once
- **Function coverage**: % of functions executed
- **Branch coverage**: % of branches/paths in the code is executed at least once
  - Conditional statements: if-else, switch-case
  - Loops: Skip the loop, execute once, execute more than once
  - Try-catch
- **Conditional coverage**: % of individual conditions evaluated to both true and false at least once

Control Flow Graphs

- Basic block: longest possible sequence of consecutive statements such that control can enter the block only at the first statement and exit from the last. No conditional statements inside other than at its end
- **Feasible Path**: A path *p* is feasible for program *P* if there exists at least one test case which when input to *P* causes *p* to be traversed
- **Dominator**: X dominates Y if all possible paths from START to Y has to pass X
  - X strictly dominates Y if X dominates Y and X ≠ Y
  - X is the immediate dominator of Y if X is the last strict dominator of Y along a path from START to Y
- **Post-dominator**: X post-dominates Y if every possible path from Y to END has to pass X
  - X immediately post-dominates Y if X is the first strict post-dominator of Y
- **Data Dependence**: Y is data-dependent on X if:
  1. There is a variable v that is defined at X and used at Y, and
  2. There exists a path of non-zero length from X to Y along which v is not re-defined
- **Control Dependence**: Y is control-dependent on X if:
  1. X is not strictly post-dominated by Y, and
  2. There exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

Test Generation

Equivalence Partitioning

- Partition input domain into partitions/equivalence classes where all values in a partition are expected to behave similarly.
- An equivalence class is "covered" when at least one test has been selected form it
- Equivalence classes may overlap. In that case, choosing a test case that covers each of the overlapping equivalence classes is sufficient
- **Test selection**: Pick the least number of test cases s.t. all equivalence classes are covered
- **Guidelines**:
  - If input condition is a range (i.e a < X < b):
    - Create one test case for the valid case: a ≤ X ≤ b
    - Create two test cases for the invalid cases: X < a and X > b
  - If input condition is a single value (i.e. X = a):
    - Create one test case for the valid value: X = a
    - Create two test cases for invalid values (below and above the valid value): X < a and X > a. If the value is a boolean, 1 invalid value is sufficient (which is the negation of the valid value)
  - If input condition specifies a member of a set
    - Create one test case for when X is in the set
    - Create on test case for when X is not in the set

Boundary Value Analysis (BVA)

- **Test selection**:
  - Valid values lying on the boundaries of each equivalence class
  - Invalid values lying just outside the boundaries of each equivalence class
  - [NOT REQUIRED IN THIS COURSE] Valid values from each equivalence class (non-boundary)

Decision Tables

- Each column is a rule and corresponds to at least one test requirement
- For each test requirement, find a set of input values of variables such that the selected rule is satisfied

	1	2	3	4	5
C1	N	N	Y	Y	Y
C2		N	N	Y	Y
C3			N	Y	N
A1				X	
A2			X		X
A3		X			

Combinatorial Testing

- **Factors**: Different variables that can affect the SUT (e.g. input variable, OS)
- **Levels**: Possible values each factor can take
- With exhaustive testing, if there are *k* factors each with *n* possible levels, the number of test cases/factor combinations would be n^k, which is huge
- **Interaction Faults** are system failures due to a specific combination of factors rather than a single factor alone.
- **T-way testing** helps to detect t-way interaction faults by ensuring that all possible t-factor interactions are tested at least once while minimising the number of test cases required

Pairwise Testing

- 2-way testing which can reveal up to 2-way interaction faults
- Generate a subset of factor combinations such that every pair of factors is covered

- Technique:
  1. Order the factors such that the one with highest level is in the first column and the least at the last column
  2. First column: For each value, repeat it as many times as the number of levels of the factor with next highest level (i.e. second column)
  3. Subsequent columns: Alternate values
  4. Verify that all pairs are covered (first two columns are guaranteed)
  5. If some pair(s) is/are not covered, tweaks or add more tests

Orthogonal Arrays

- Definition: An orthogonal array is a $N \times k$ matrix where entries are from a finite set S of s symbols s.t. any $N \times t$ sub array contains each t-tuple exactly the same number of times. Denoted by $OA(N, k, s, t)$ and t = **strength** of the OA = # of factor s.t. every combination of factor levels appears at least in a run

Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

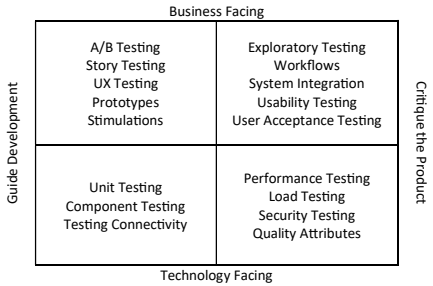
- For example, $OA(4, 3, 2, 2)$. Any 4×2 sub-array consists of (i, j) where $i, j \in \{1, 2\}$ and appears exactly once (see "index")
- **Index** of an OA, denoted by $\lambda = N/s^t$. For example, $\lambda = 4/2^2 = 1$ implies that each t-tuple (e.g. pair) appears exactly once ($\lambda = 1$) in any 4×2 sub-array
- **Alternative notation**: $L_N(s^k)$ = OA of N runs with k factors that can take any value from a set of s symbols (i.e. each factor has the same level, aka fixed level OA). E.g, $L_4(2^3)$

Agile Testing

Traditional vs Agile Testing

- Traditional (waterfall): Testing is a phase on its own at the end of the waterfall
- Agile: Iterative and incremental (e.g. development in one sprint, black-box testing in the next sprint)
  - Non-functional testing typically in later sprints when product has matured. Functional testing typically in earlier sprints

Agile Testing Quadrants



Test-Driven Development (TDD)

- Defn: Unit tests are written in small incremental steps before writing code to make these (initially failing) tests pass, then refactoring
- Helps guide **code design and implementation** for **developers**

Acceptance Test-Driven Development (ATDD)

- Defn: Developers, testers and business stakeholders collaborate to define acceptance criteria as tests before coding begins
- Helps ensure software meets **business requirements**
- Similar to TDD, except TDD focuses on low-level unit tests and is developer-centric, whereas ATDD focuses on high-level acceptance tests and is user-centric and collaborative
- User stories can be turned into acceptance tests

Behaviour-Driven Development (BDD)

- Defn: Developers, testers, business stakeholders and customers collaborate to define behaviours of the software before coding begins
- Similar to ATDD, except ATDD uses structured and technical terms to describe tests (user stories), and BDD uses natural language
  - E.g. Gherkin format: **Given-When-Then**
  - BDD translates user stories from ATDD into behaviours
- Helps ensure software meets **expected behaviour**

Non-Functional Tests

Performance Testing

- Can be done by either QA team or developers
- Evluate responsiveness, speed, scalability, stability, throughput and overlal behaviour under a variety of conditions
- Test metrics:
  - Response Time: total time to respond to a user request
    - Minimum, maximum, average, 90th percentile
  - Throughput = total # of requests / total time taken
  - Error rate = (# of failed requests / total # of requests) ✕ 100
  - CPU utilization = (1 – (idle time / total time)) ✕ 100
  - Memory utilization = (used memory / total memory) ✕ 100
  - Latency = processing time + network transit time
  - Network latency = time taken for response – time spent

Mobile Testing

- Need to test different
  - Devices (e.g. OS, hardware)
  - App (e.g. Native, Mobile web, Progressive web app)
  - Network (e.g. offline, 3G, 4G)
- Weighted Device Platform Matrix (WDPM)
  1. List omportant OS variants as column labels
  2. List targeted devices as matrix row lavels
  3. Assign ranking to indicate relative importance of each OS and each device
  4. Compute the product of each OS-device pair and enter each product as the cell entry in the matrix

		OS1	OS2	OS3
	Ranking	3	4	7
Device 1	7	NA	28	49
Device 2	3	9	NA	NA
Device 3	4	14	NA	NA
Device 4	9	NA	36	63

SE Testing Methods used in AI

Differential Testing

- Run same inputs on multiple implementations/programs and compare outputs – if the results differ, at least one is wrong
- Often used when correct output is unknown but multiple versions or implemetations exist

Metamorphic Testing

- Verify correctness of a program by verifying whether expected relationships hold betweeen inputs and their corresponding outputs
- Often used when its hard to know what the correct output should be but you can still define how the output should behave when input changes

Mutation Testing

- Used to evaluate quality of test cases by injecting faults (mutations) into code (to create a mutants) and checking if the test catch them
- If test results for original and mutants are the **same**, the test cases are **not thorough enough** to detect the bug(s)
- Mutants are killed if tests are failing (because it means that their bugs are caught)
- Mutation score = # mutants killed / total # of mutants
- Higher mutation score → better test suite
- Limitations
  - Some mutants are hard/impossible to kill (e.g. unreachable code, false positives)
  - Mutants are artifical (does not consider meaning of code, only syntax)
  - Only able to detect a subset of all possible defects

Fuzz Testing

- Automatically generate large amount of pre-defined, random or malformed inputs to feed into program to find crashes, bugs or security vulnerabilities
- Good at revealing edge cases
- However, most bugs found tend to be of the same type

Test Metrics

Manual Test Metrics

- Test Execution Coverage (%) = $\frac{\text{\# tests to run}}{\text{total \# of tests to be run}} \times 100$
- Requirements Coverage (%) = $\frac{\text{\# requirements covered}}{\text{total \# of requirements}} \times 100$
- Test Effectiveness Metrics = $\frac{\text{\# bugs found in test}}{\text{total \# of bugs found}} \times 100$
- Bug find rate (number of defects per test hour) = $\frac{\text{total \# of defects}}{\text{total \# of test hours}}$
- Number of tests run per time period = $\frac{\text{\# of tests run}}{\text{total time}}$
- Number of bugs per test = $\frac{\text{total \# of defects}}{\text{total \# of tests}}$
- Average time to test a bug fix = $\frac{\text{total time between defect fix to retest for all defects}}{\text{total \# of defects}}$
- Passed Test Case (%) = $\frac{\text{\# of passed tests}}{\text{total \# of tests}} \times 100$
- Fixed Defects (%) = $\frac{\text{\# of defects fixed}}{\text{total \# of defects reported}} \times 100$
- Average time to repair defects = $\frac{\text{total time taken for bug fixes}}{\text{total \# of bugs}}$

Product Quality Metrics

- Error: Human mistake resulting in incorrect software
- Defect: System's actual behaviour deviates from the expected behaviour
- Failure: Software doesn't perform its required function
- **Mean Time To Failure (MTTF)** = $\frac{\text{total operation time}}{\text{total \# of failures}}$
  - Average time a system runs before encountering complete failure which is **non-repairable**
- **Mean Time Between Failures (MTBF)** = $\frac{\text{total operation time+time to repair}}{\text{total \# of failures}}$
  - Average time between failures for a repairable system
- **Defect Density** (per KLOC) = $\frac{\text{\# of bugs}}{\text{total KLOC}}$
  - Can be influenced by code complexity, developers' skill levels and defect type
  - Helps validate testing quality, save testing resources and compare developer efficiency
- **Defect Gap** (%) = $\frac{\text{total \# of defects fixed}}{\text{total \# of } \textbf{valid} \text{ defects reported}} \times 100$
- **Problems per user month (PUM)** = total # of problems (invalid + valid defects, first time and repeated) reported by customers for a time period + total # of license-months of the software during the period
  - # of license-months = # of install licenses of the software × # of months in the calculated period
- **Customer Satisfaction Metrics**
  - Measured via customer survey data with five-point scale (very dissatisfied to very satisfied)
  - Non-satisfied = neutral + dissatisfied + very dissatisfied
  - Net satisfaction index = $\frac{(\text{\# satisfied +\# very satisfied})-(\text{\# dissatisfied + \# very dissatisfied})}{\text{\# responses}} \times 100$
- **Defect Removal Effectiveness** = $\frac{\text{\# of defects removed during dev. phase}}{\text{\# defects latent in the product}} \times 100$
  - Denominator can only be approximated (= defects removed during the phase + defects found later)

Productivity Metrics

- Error: Human mistake resulting in incorrect software
- Defect: System's actual behaviour deviates from the expected behaviour
- Failure: Software no performing the required functions
- **Defects per 100 hours of testing** = $\frac{\text{\# of defects found for a period}}{\text{total hrs spent to get those defects}} \times 100$
- **Test cases executed per 100 hours of testing** = $\frac{\text{total \# of test cases executed for a period}}{\text{total hrs spent in test execution}} \times 100$
- **Test cases developed per 100 hours of testing** = $\frac{\text{total \# of test cases developed for a period}}{\text{total hrs spent in test case development}} \times 100$