Unit Tests

- Unit testing is widely used, regardless of software architectural style
- Unit testing means testing the smallest separate module in the system, for most languages this will be a class
- The principle of unit testing is to test each unit in isolation
- Thorough unit testing increases confidence in the software when units are integrated

Why objects and not methods? - Methods are usually not independent of each other in a class

1 Software Contracts

Contracts identify what a unit should do

Consider the impact of any changes to a contract (interface), such as

- Addition of new methods
- Removal of existing methods
- Renaming methods or changes to their prototype/signature

Tests will then need to be updated to deal with any changes that occur to the contract so contract changes should be minimised to avoid excessive regression testing
Testing public methods implies black-box testing

- No need (ability) to see the source code for the unit under test
- Can test third party libraries without access to source code
- No need to now about the precise implementation (encapsulation)

The contract should define what the correct output should be (hence the contract provides the oracle)
Unit testing can use the methods defined in the contract for the unit
For contracts defined by an interface, we have to test a concrete class that implements the contract
Contracts need not be interfaces - the contract for a class could be its public methods, therefore we test public methods

2 Functional Testing

- Focuses on what the module does, not how it does it
- We usually aim to select test cases that cover normal operation, boundary conditions, and errors

3 Equivalence testing

For a given test case, we should be able to separate input values into equivalence classes. These are organised (categorised) so that:

- Every possible input belongs to one equivalence class, so the set of classes together contain the complete set of possible inputs
- No input value belongs to more than one equivalence class
- An element can then represent all other elements in the class, so if a fault can be detected using one member of the equivalence class, there is a high probability that every other member of that class will reveal the same fault

This means that the overall test set only needs to contain representative members of each class

4 Code Reviews

These are a form of structural testing. Two forms:

- A **walkthrough** is led by the coder who represents the code to the team, the members comment on its correctness in an informal atmosphere and the aim is to find faults, not to fix them
- A **code inspection** is more formal than a walkthrough, with the team checking code and documentation against a prepared list of concerns

These approaches have been studied experimentally for many years, and are recognised as being consistently successful as a means of finding faults

Because of this, many organisations also use reviews to test the requirements specification, design etc

5 Structural Testing Approaches

These are based upon the way that the data is manipulated by the code. Major forms are:

- Statement testing - Every statement in the component is executed at least once in some test
- Branch testing - For every decision point, each branch is chosen at least once in some test
- Path testing - Every distinct path through code is executed at least once in some test
- Definition use path testing - Every path between the definition of every variable to every use of that definition is exercised in some test
- All-uses testing - The test set includes at least one path from every definition to every use that can be reached by that definition
- All-predicate-uses/some-computational-uses testing: For every variable and definition of that variable, a test includes at least one path from the definition to every predicate use; if there are definitions not covered by that, then include computational uses so that every definition is covered
- All-computational-uses/some-predicate uses testing - Redefinition of above definition

6 Automation

Testing by hand is:

- Slow
- Tedious
- Error prone

To automate unit testing, code is written to:

- Set up the unit
- Formulate inputs to methods
- Execute methods with inputs to get outputs
- Compare outputs with expected (correct) value

Tests can be run at any time. But again, needs a dependable oracle

7 When to stop testing

- Ideally, when we have found all of the faults. But software testing is an ISP
- However, if we find a lot of faults, there is a good probability that a lot more remain at whatever point we stop testing
- Beyond a certain point, the effort needed to detect each fault will rise - no real correlation with the importance of the remaining ones either
- Techniques for determining when to stop include:
  - Fault seeding - put in intentional faults to test how good error detection is
  - Fault estimates
  - No of tests still to perform complete coverage