Unit Testing

CO328 - Software Engineering

Malintha Adikari

Unit testing

- A level of the software testing process where individual units/components of a software/system are tested.
- The purpose is to validate that each unit of the software performs as designed.
- A method by which individual units of source code are tested to determine if they are fit for use
- Concerned with functional correctness and completeness of individual program units
- Typically written and run by software developers to ensure that code meets its design and behaves as intended.
- Its goal is to isolate each part of the program and show that the individual parts are correct.

What is Unit Testing

Concerned with

- Functional correctness and completeness
- Error handling
- Checking input values (parameter)
- Correctness of output data (return values)
- Optimizing algorithm and performance

Types of testing

- **Black box testing** – (application interface, internal module interface and input/output description)
- **White box testing**- function executed and checked
- **Gray box testing** - test cases, risks assessments and test methods

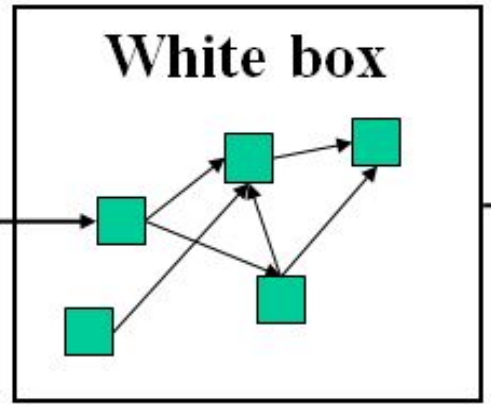
Black-, Gray-, & White-box Testing

Input determined by...

Result

… requirements → **Black box** → ***Actual*** **output compared with *required* output**

… requirements & key design elements → **Gray box** → ***As for black- and white box testing***



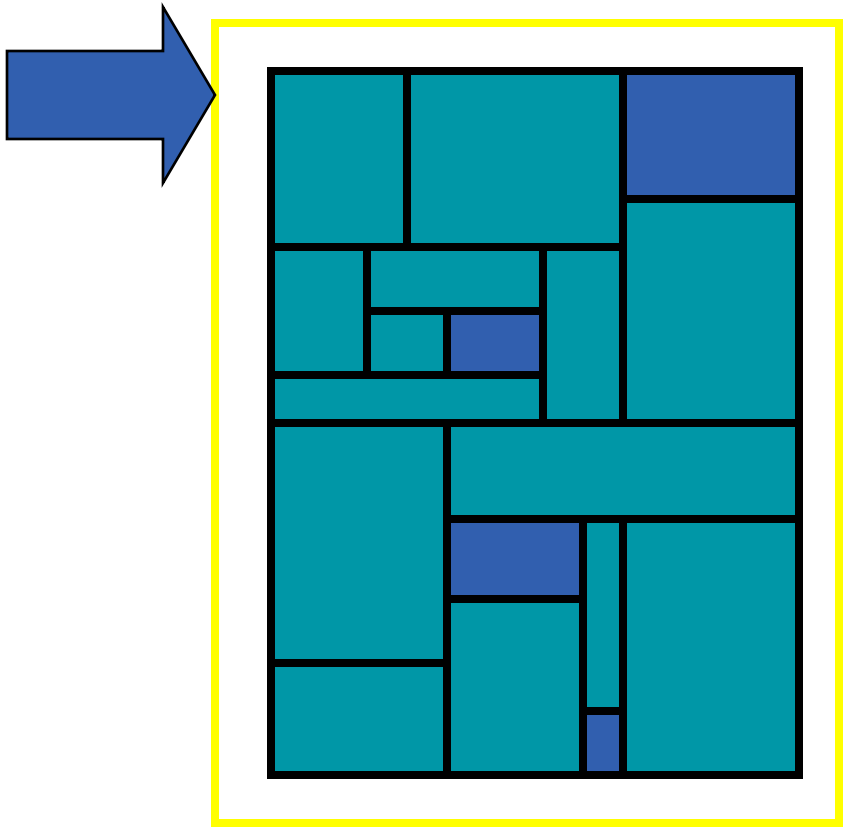
… design elements → **Confirmation of expected behavior**

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001), with permission.

Traditional testing vs. Unit Testing

Traditional Testing

- Test the system as a whole
- Individual components rarely tested
- Errors go undetected
- Isolation of errors difficult to track down

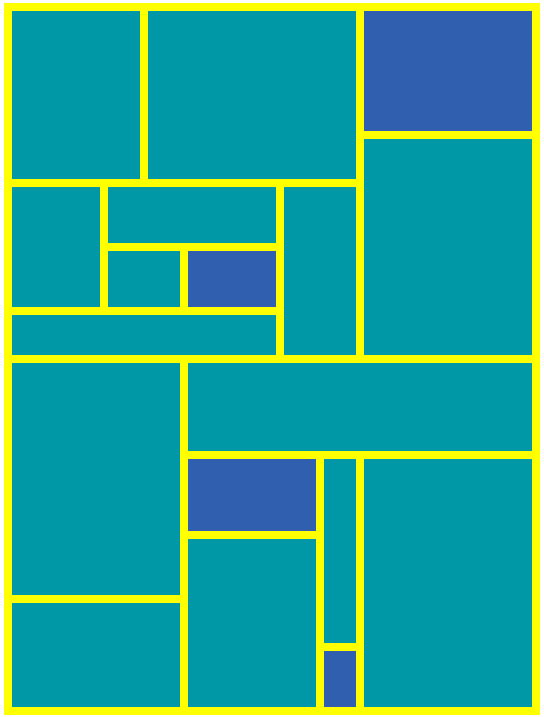


Traditional Testing Strategies

- Print Statements
- Use of Debugger
- Debugger Expressions
- Test Scripts

Unit Testing

- Each part tested individually
- All components tested at least once
- Errors picked up earlier
- Scope is smaller, easier to fix errors



Unit Testing Ideals

- Isolatable
- Repeatable
- Automatable
- Easy to Write

Why Unit Test?

- Faster Debugging
- Faster Development
- Better Design
- Excellent Regression Tool
- Reduce Future Cost

Benefits

- Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly.
- By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.
- Unit testing provides a sort of living documentation of the system.

Best Practises

3 - Steps

- Prepare an input (arrange)
- Call a method (act)
- Check an output (assert)

Also called **Arrange -> Act -> Assert**

Best Practises

Be Fast in Execution

We need fast unit tests because the nature of frequent execution

- Several times per day [Test-after development]
- Several times per hour [Test driven development]
- Every few minutes [IDE - Execute after save]

Best Practises

Consistent

Multiple invocations of the test should consistently return true or consistently return false, provided no changes was made on code.

Best Practises

Atomic

- Only two possible results: Pass or Fail
- No partially successful tests
- A test fails -> The whole suite fails Broken window effect

Best Practises

Single responsibility

- One test should be responsible for one scenario only.
- Test behaviour, not methods: One method, multiple behaviours
- Multiple tests One behaviour, multiple methods
- Multiple asserts in the same test - acceptable as long as they check the same behaviour

Best Practises

Test Isolation

- Tests should be independent from one another
- Different execution order - the same results
- No state sharing
- Instance variables
  - JUnit - separated
  - TestNG - shared

Best Practises

Environment Isolation

Unit tests should be isolated from any environmental influences

- Database access
- Webservices calls
- JNDI look up
- Environment variables
- Property files
- System date and time

Best Practises

Fully Automated

- Automated tests execution Automated results gathering Automated decision making (success or failure)

- Automated results distribution
  - Email
  - IM
  - System tray icon
  - Dashboard web page
  - IDE integration

Best Practises

Self Descriptive

- Unit test = development level documentation
- Unit test = method specification which is always up to date
- Unit test must be easy to read and understand Variable names Method names Class names

Best Practises

No Conditional Logic

- Correctly written test contains no "if" or "switch" statements.
- No uncertainty All input values should be known
- Method behaviour should be predictable
- Expected output should be strict
- Split the test into two (or more) tests instead of adding "if" or "switch" statement.

Best Practises

No Loops

High quality test contains no "while", "do-while" or "for" statements. Typical scenarios involving loops:

- Hundreds of repetitions
- A few repetitions
- Unknown number of repetitions

Case 1: Hundreds of repetitions

If some logic in a test has to be repeated hundreds of times, it probably means that the test is too complicated and should be simplified.

Best Practises

No Loops...cont

Case 2: A few repetitions

Repeating things several times is OK, but then it's better to type the code explicitly without loops.You can extract the code which needs to be repeated into method and invoke it a few times in a row.

Case 3: Unknown number of repetitions

If you don't know how many times you want to repeat the code and it makes you difficult to avoid loops, it's very likely that your test is incorrect and you should rather focus on specifying more strict input data.

Best Practises

No Exception Catching

- Catch an exception only if it's expected
- Catch only expected type of an exception
- Catch expected exception and call "fail" method
- Let other exceptions go uncatched

Best Practises

Assertions

- Use various types of assertions provided by a testing framework
- Create your own assertions to check more complicated, repetitive conditions
- Reuse your assertion methods

Best Practises

Separation per business module

- Create suites of tests per business module
- Use hierarchical approach
- Decrease the execution time of suites by splitting them into smaller ones (per sub-module)
- Small suites can be executed more frequently

Best Practises

No test logic in production code

- Separate unit tests and production code
- Don't create methods/fields used only by unit tests
- Use "Dependency Injection"

Best Practises

Informative assertion messages

- By reading an assertion message only, one should be able to recognize the problem.
- It's a good practice to include business logic information into assertion message.
- Assertion messages: Improve documentation of the code
  Inform about the problem in case of test failure

Guidelines

- Know the cost of testing
  - Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%.

- Prioritize testing
  - Unit testing is a typical bottom-up process, and if there is not enough resources to test all parts of a system priority should be put on the lower levels first.

Guidelines

- Prepare test code for failures
  - If the first assertion is false, the code crashes in the subsequent statement and none of the remaining tests will be executed. Always prepare for test failure so that the failure of a single test doesn't bring down the entire test suite execution.

Guidelines

- Write tests to reproduce bugs
  - When a bug is reported, write a test to reproduce the bug (i.e. a failing test) and use this test as a success criteria when fixing the code.

- Know the limitations
  - Unit tests can never prove the correctness of code.

Unit Testing Techniques

- Structural, Functional & Error based Techniques

Structural Techniques:

- It is a White box testing technique that uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs.

Unit Testing Techniques

Functional testing techniques:

These are Black box testing techniques which tests the functionality of the application.

Some of Functional testing techniques

- **Input domain testing:** This testing technique concentrates on size and type of every input object in terms of boundary value analysis and Equivalence class.

- **Boundary Value:** Boundary value analysis is a software testing design technique in which tests are designed to include representatives of boundary values.

- **Syntax checking:** This is a technique which is used to check the Syntax of the application.

- **Equivalence Partitioning:** This is a software testing technique that divides the input data of a software unit into partition of data from which test cases can be derived

Unit Testing Techniques

Error based Techniques

The best person to know the defects in his code is the person who has designed it.

Few of the Error based techniques

- **Fault seeding:** techniques can be used so that known defects can be put into the code and tested until they are all found.

- **Mutation Testing:** This is done by mutating certain statements in your source code and checking if your test code is able to find the errors. Mutation testing is very expensive to run, especially on very large applications.

- **Historical Test data:** This technique calculates the priority of each test case using historical information from the previous executions of the test case.