UNIT TESTING

- is 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.

UNIT TESTING

- 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**

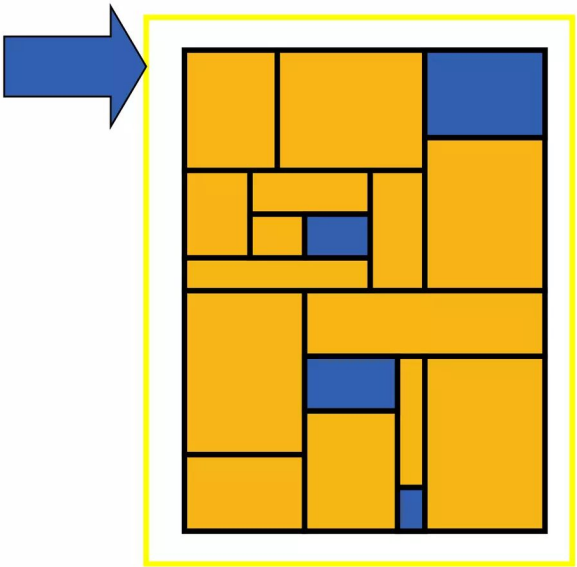
... requirements & key design elements → **Gray box** → ***As for black- and white box testing***

... design elements → **White box** → **Confirmation of expected behavior**

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

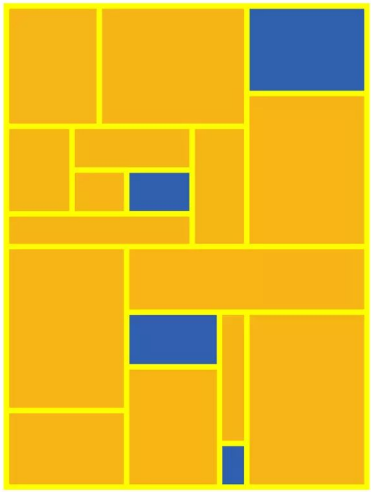


Traditional Testing Strategies

- 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.

GUIDELINES

- **Keep unit tests small and fast**
  - Ideally the entire test suite should be executed before every code check in. Keeping the tests fast reduce the development turnaround time.

- **Unit tests should be fully automated and non-interactive**
  - The test suite is normally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are not proper unit tests.

GUIDELINES

- **Make unit tests simple to run**
  - Configure the development environment so that single tests and test suites can be run by a single command or a one button click.

- **Measure the tests**
  - Apply coverage analysis to the test runs so that it is possible to read the exact execution coverage and investigate which parts of the code is executed and not.

GUIDELINES

- **Fix failing tests immediately**
  - Each developer should be responsible for making sure a new test runs successfully upon check in, and that all existing tests runs successfully upon code check in. If a test fails as part of a regular test execution the entire team should drop what they are currently doing and make sure the problem gets fixed.

GUIDELINES

- **Keep testing at unit level**
  - Unit testing is about testing *classes*. There should be one test class per ordinary class and the class behaviour should be tested in isolation. Avoid the temptation to test an entire work-flow using a unit testing framework, as such tests are slow and hard to maintain. Work-flow testing may have its place, but it is not unit testing and it must be set up and executed independently.

GUIDELINES

- **Start off simple**
  - One simple test is infinitely better than no tests at all. A simple test class will establish the target class test framework, it will verify the presence and correctness of both the build environment, the unit testing environment, the execution environment and the coverage analysis tool, and it will prove that the target class is part of the assembly and that it can be accessed.

GUIDELINES

- **Keep tests independent**
  - To ensure testing robustness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed.

- **Name tests properly**
  - Make sure each test method test one distinct feature of the class being tested and name the test methods accordingly. The typical naming convention is test[what] such As testSaveAs(), testAddListener(), testDeleteProperty() etc.

GUIDELINES

- **Keep tests close to the class being tested**
  - If the class to test is Foo the test class should be called FooTest (*not* TestFoo) and kept in the same package (directory) as Foo. Keeping test classes in separate directory trees makes them harder to access and maintain. Make sure the build environment is configured so that the test classes doesn't make its way into production libraries or executables.

GUIDELINES

- **Test public API**
  - Unit testing can be defined as *testing classes through their public API*. Some testing tools makes it possible to test private content of a class, but this should be avoided as it makes the test more verbose and much harder to maintain. If there is private content that seems to need explicit testing, consider refactoring it into public methods in utility classes instead. But do this to improve the general design, not to aid testing.

GUIDELINES

- **Think black-box**
  - Act as a 3rd party class consumer, and test if the class fulfills its requirements. And try to tear it apart.

- **Think white-box**
  - After all, the test programmer also wrote the class being tested, and extra effort should be put into testing the most complex logic.

GUIDELINES

- **Test the trivial cases too**
  - It is sometimes recommended that all non-trivial cases should be tested and that trivial methods like simple setters and getters can be omitted. However, there are several reasons why trivial cases should be tested too:
    - *Trivial* is hard to define. It may mean different things to different people.
    - From a black-box perspective there is no way to know which part of the code is *trivial*.
    - The *trivial* cases can contain errors too, often as a result of copy-paste operations:

GUIDELINES

- **Cover boundary cases**
  - Make sure the parameter boundary cases are covered. For numbers, test negatives, 0, positive, smallest, largest, NaN, infinity, etc. For strings test empty string, single character string, non-ASCII string, multi-MB strings etc. For collections test empty, one, first, last, etc. For dates, test January 1, February 29, December 31 etc. The class being tested will suggest the boundary cases in each specific case. The point is to make sure as many as possible of these are tested properly as these cases are the prime candidates for errors.

GUIDELINES

- **Provide a random generator**
  - When the boundary cases are covered, a simple way to improve test coverage further is to generate random parameters so that the tests can be executed with different input every time. To achieve this, provide a simple utility class that generates random values of the base types like doubles, integers, strings, dates etc. The generator should produce values from the entire domain of each type.

GUIDELINES

- **Test each feature once**
  - When being in testing mode it is sometimes tempting to assert on "everything" in every test. This should be avoided as it makes maintenance harder. Test exactly the feature indicated by the name of the test method. As for ordinary code, it is a goal to keep the amount of test code as low as possible.

GUIDELINES

- **Use explicit asserts**
  - Always prefer assertEquals(a, b) to assertTrue(a == b) (and likewise) as the former will give more useful information of what exactly is wrong if the test fails. This is in particular important in combination with random value parameters as described above when the input values are not known in advance.

GUIDELINES

- **Provide negative tests**
  - Negative tests intentionally misuse the code and verify robustness and appropriate error handling.

- **Design code with testing in mind**
  - Writing and maintaining unit tests are costly, and minimizing public API and reducing cyclomatic complexity in the code are ways to reduce this cost and make high-coverage test code faster to write and easier to maintain.

GUIDELINES

- **Don't connect to predefined external resources**
  - Unit tests should be written without explicit knowledge of the environment context in which they are executed so that they can be run anywhere at anytime. In order to provide required resources for a test these resources should instead be made available by the test itself.

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.

Major Structural techniques are:

- **Statement Testing:** A test strategy in which each statement of a program is executed at least once.
- **Branch Testing:** Testing in which all branches in the program source code are tested at least once.
- **Path Testing:** Testing in which all paths in the program source code are tested at least once.
- **Condition Testing:** Condition testing allows the programmer to determine the path through a program by selectively executing code based on the comparison of a value
- **Expression Testing:** Testing in which the application is tested for different values of Regular Expression.

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.