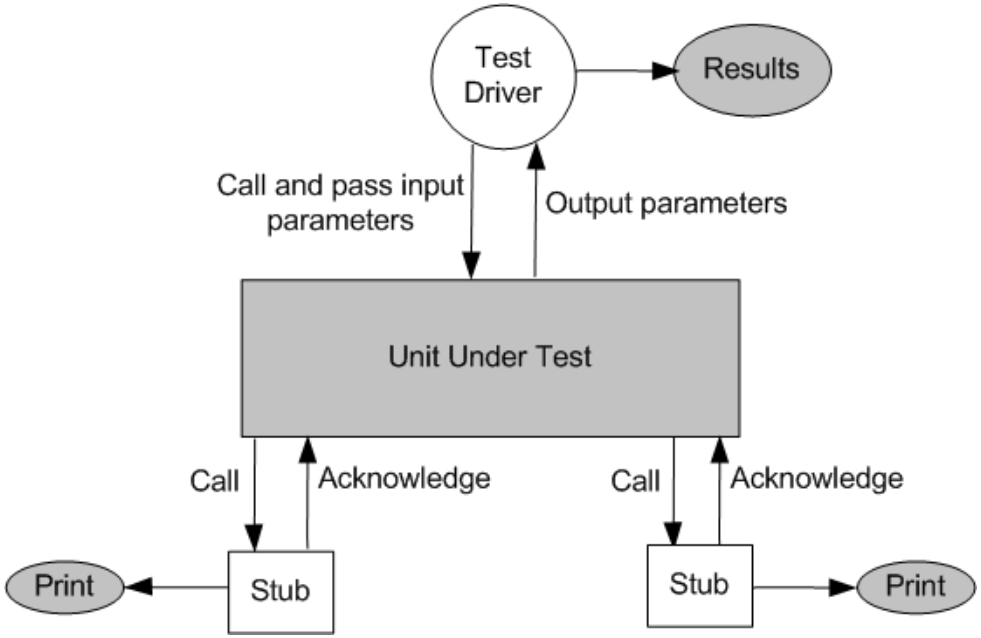
Dynamic Unit Testing

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
  - A *test driver* is a program that invokes the unit under test (UUT)
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs*
  - It is a dummy program
- The *test driver* and the *stubs* are together called *scaffolding*
- The low-level design document provides guidance for selection of input test data



Unit Testing Ideals

- Isolatable
- Repeatable
- Automatable
- Easy to Write

Structure of a Unit Test?

```
[TestFixture]
0 references
public class CalculateTest
{
    [Test]
    0 references
    public void checkAddTwoNumber() {
        //Arrange
        Calculate calculate = new Calculate();
        //Act
        int result = calculate.addTwoNumber(2, 3);
        //Assert
        Assert.AreEqual(6, result);
    }
}
```

LAB

- Lab ut-01

GUIDELINES

- **Keep unit tests small and fast**
  - Entire test suite is executed before every code check in. Keeping the tests fast.

- **Unit tests has to be fully automated and non-interactive**
  - Since the test suite is executed on a regular basis, it must be fully automated.

GUIDELINES

- **Make unit tests simple to run**
  - Test Suite has to run out of the box within the IDE.

- **Measure the tests**
  - Apply coverage analysis (or other KPIs) to the test runs so that it is possible measure the quality.

LAB

- Lab ut-02

GUIDELINES

- **Fix failing tests immediately**
  - Each test has to run successfully before check in.
  - If a test fails 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
  - Work-flow testing is part of integration testing.

GUIDELINES

- **Start off simple**
  - One simple test is infinitely better than no tests at all.
  - Start with a few tests and increase continously.

GUIDELINES

- **Keep tests independent**
  - Tests should never rely on other tests nor should they depend on the ordering in which tests are executed.

- **Name tests properly**
  - Make sure the test class name is acurate.
  - The typical naming convention is test[what] such As testSaveAs(), testAddListener(), testDeleteProperty() etc.

LAB

- Lab ut-03

GUIDELINES

- **Keep unit tests small and fast**
  - Entire test suite is executed before every code check in. Keeping the tests fast.

- **Unit tests has to be fully automated and non-interactive**
  - Since the test suite is executed on a regular basis, it must be fully automated.

GUIDELINES

- **Keep tests close together**
  - If the class to test is Foo the test class should be called FooTest (*not* TestFoo).
  - Keeping test classes in a separate directory tree ensures a clear structure.
  - Test classes are not allowed to put into the production package.

GUIDELINES

- **Test public API**
  - Unit testing can be defined as *testing classes through their public API*.

GUIDELINES

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

- **Think white-box**
  - Typically, the programmer write the tests as well.
  - This can cause problems.
  - Be reminded that extra time is necessary to write the test logic.

GUIDELINES

- **Test the trivial cases too**
  - Often, ist recommended that all non-trivial cases should be tested

    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

- **Focus on execution coverage first**
  - Differentiate between *execution coverage* and *actual test coverage*.
  - The initial goal of a test is to ensure high execution coverage.

  When this achieved, test coverage has to be increased.

  Note that actual test coverage cannot be easily measured (and is always close to 0% anyway).

GUIDELINES

- **Test each feature once**
  - One Test has to cover one case.
  - Test exactly the feature indicated by the name of the test method.

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

LAB

- Lab ut-04

GUIDELINES

- **Don't connect to predefined external resources**
  - Unit tests has to be written without explicit knowledge of the environment context.
  - Use Mocks in order to provide required resources for a test.

GUIDELINES

- **Know the cost of testing**
  - Not writing unit tests is costly, but writing unit tests is costly too.
  - Trade-off in terms of execution coverage is at about 80%.

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.
- The tester chooses the input data of the test wisely to exercise paths through the code and determines the desired outputs.

Unit Testing Techniques:

Functional testing techniques:

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

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.