Why we need TDD:

Unit test: set of the test that can be run automatically to verify if the everything works fine or anything wrong int code

If the unit test can cover 100% of the code base then wee can avoid lot of errors in the deployment phase.

Eg: Canonical form where in search functionality is performed with canonical for of text

Here first algorithm is designed

The deployed

Bug found in the production (null reference)

Write a test and deploy again

again, new bug found (empty text issue)

again, fixed and deployed.

TDD is a solution this problem we do a development along with unit test cases written using unit test framework

TDD: test driven development, Test first development

Write test ahead of production code.

Unit testing is a framework where in you write testable code. Code which passes the unit test will go to the production else we need to write the complex code to make it go to the production.

Common misconception

1. Double the developer efforts (Avoid lot of production bug)
2. Cant write test until know the design, cant know the design until you implement the code (helps in arriving the better design by identifying all possible race conditions.)
3. Have to write all test before you start code

Approach would be:

Write one test

Watch it fail

Implement the code

Watch it pass

Repeate

Main technique of TDD is Red/ Green/ Refactor:

Red mean test fail

Green is test pass

Refactor is we basically dos changes to the code so that next time when the test run it meets all acceptance criteria.

3 laws of TDD

1. Don’t write production code before writing a failing test
2. Write minimal test which is enough to see the failure.
3. Don’t write production code more than enough to pass the failing test.

Changing requirement:

Rigidity:

Any changes to the system takes lot of time to implement.

High coupling between the modules

Fragile:

Small change to one module causes failure to the other module reason tight coupling

Isolate module

Immobile: again because of tight coupling things cannot be reused or shared among multiple modules.

Viscosity:

Adding feature lead to checking lot of things.

F I R S T:

% Requirement decides the good unit test.

F: Fast should run very fast

I:Independednt (each test should be independent of each other) Ordered test feature of the TDD

Helps is running the test in some order

R:Repeatabe produce same result and consistence wherever you run

S: self-valid table pass or fail result should be clear

T: in time written before production code so that no logic before the production code.

Code coverage:

Matrix that indicate the amount of the code covered by the test

Always shown in percentage

.Ie total lines of code divided by the amount of the code test has gone through

Best coverage is 705 of the code.

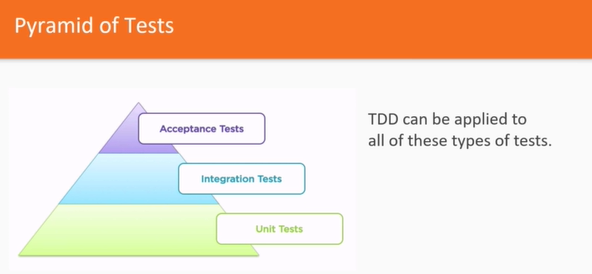
Types of test and TDD:

Unit : verify the behaviour of the unit under test isolation

Integration: verify the part of the system or whole system.

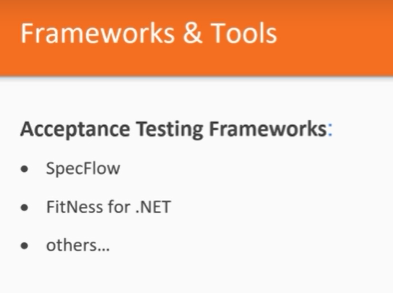
Acceptance test: verify the software from the user point of view (special test cases written will be converted into u it test by the tools)

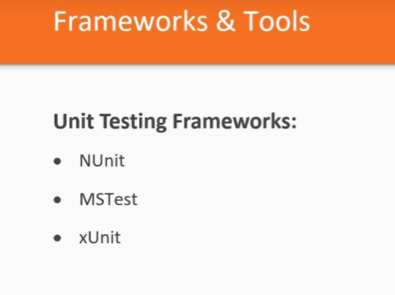
Pyramid of test:



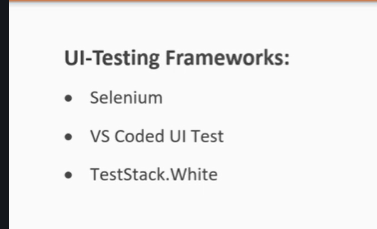
Framework and tools:

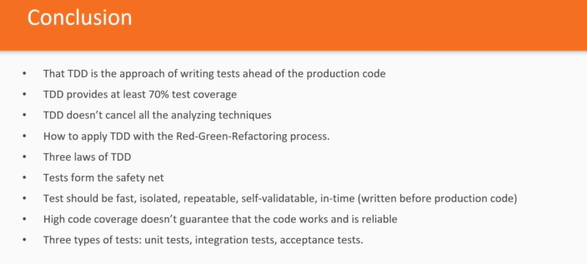
Writing test ahead of production code











In action:

Triangulation:

Not generalizing implementation. Drive the implementation by writing the test is called triangulation

Initially install the Nuinit from the NuGet package manager

And use the name space

Using NUnit.Framework

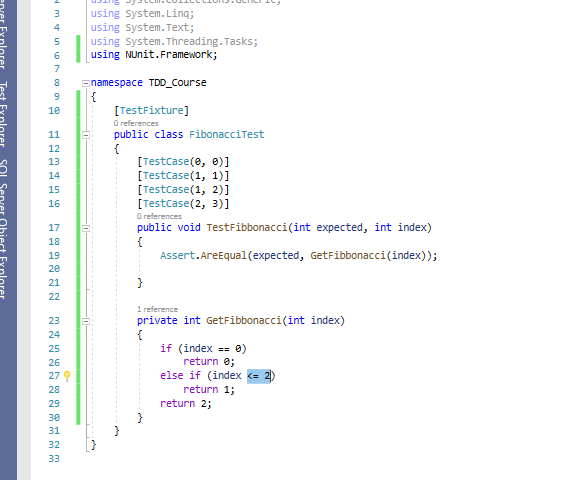
Attributes used:

TestFicture for the class

Test

TestCase(parameter list)

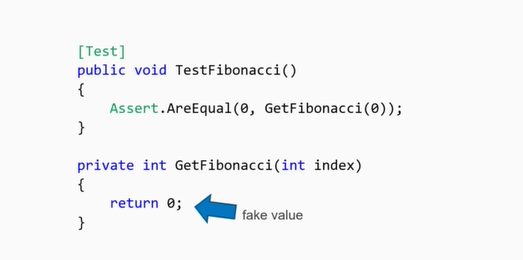




3 techniques used in TDD

Faking

Send fake result of constant value from the methods



Triangulation

1. Keep adding the test cases until the right way of implementation emerged out
2. Two or more test cases are enough to stop triangulation.
3. Faking triangulation lead to the implementation where the generalization becomes obvious.

Obvious implementation:

Right implementation is obvious skip triangulation and faking

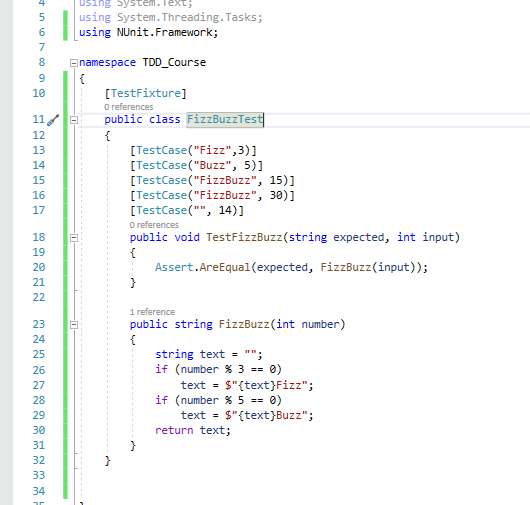
Careful implementing the whole algorithms out of mind

Very often you think implementation is obvious

Often it fails in edge cases.

Obvious implementation

Fizz buzz



CT: Continuous testing:

Tools

Ncrunch

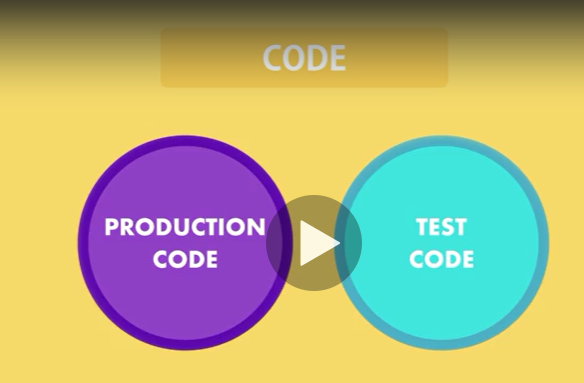
Live unit testing

TIcTacToe or crosses and Noughts

NCRUNCH test runner tool

Automated testing:

Practice to write a code to test our production code. Later this test code is run automatically to test our production ready code



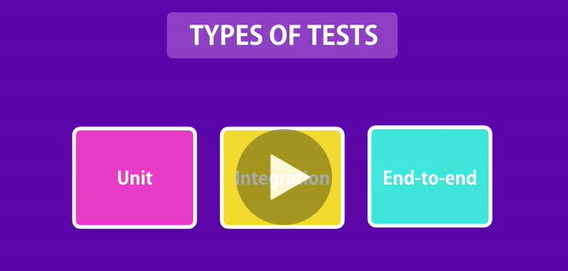
Why:

Code can be tested automatically any number of time with less time and without the involvement of the user

Bugs can be minimized before the code goes to production

Code refracturing: changing the structure without changing the behaviour. After changing the code test can be run automatically.

Automated test has



Unit test: test code without external dependency.

Cheap to write, execute fast

Unit of functionality is tested

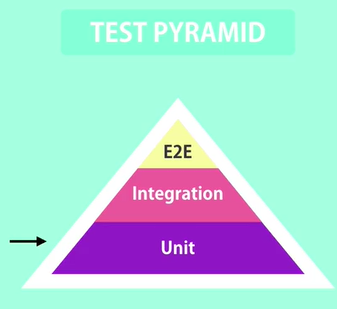
Test logic, looping ,if cases, edge conditions.

Integration text: text for the external dependency. (DB, file, services)

End to end test:

Drives the application with its UI

Test Pyramid:





Tools:

Unit test:

NUinit

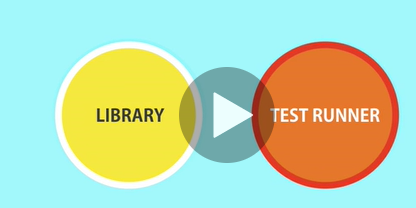
MStest

XUnit

All the unit test framework has two part

Library which helps in writing the unit test

Test runner to run the test cases. (report of test pass and fail)



MSTest:

You have to add test project for this purpose VS provides with the built in template for this.

Each test in Mstest follows AAA method

A—Arrange (deals with the creation of the data or object)

A – Act (executing the methods n that object)

A – Assert (Basically execute the test logic here)

Uses name space:

using Microsoft.VisualStudio.TestTools.UnitTesting;

Class ned to be decorated with [TestClass] and method with [TestMethod]

While writing the test user the format

Methos\_Secnario\_ExpecetedResult

NUnit testing:

We need 2 package to be installed for this purpose

Install-Package NUnit -version 3.1.1.1

Install-Package NUnitAdaptor -version 3.1.1.1

TextFixture decorator for the class

Test decorator for the method

Asset.That(value, is.Equall);

TDD:

Test driven development or test ahead development

Foundation for TDD

First you write the failing test

Then write simplest possible code of implementation that will pass all the test

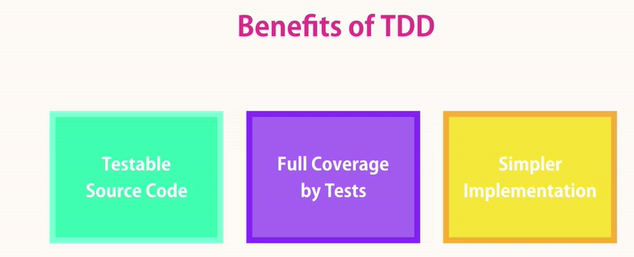
Finally refractor your code.



Code is tested in advance

Covers every line of code

Simpler implementation



Fundamentals of unit testing:

Characteristics of good unit test:

1. Firsts class citizen
2. Clean readable maintainable code
3. No logic in test like loop conditions etc (simply call methods and assert)
4. Isolated
5. Not to specific not too general

What to test and what not to test:

1.All query methods basically it returns some data so all such methods can be tested

2. Command methods: does some changes to the data present on the memory or performs some action and it also returns value from the executed action.

Not to test:

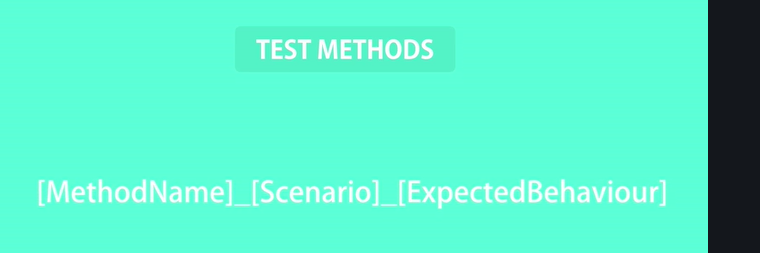
3rd part library

Test project

Maintain separate project for the test

For every class in main project add test class in unit test project and add one or more unit test method

Number of test methods is greater than equal to the execution path.



Assert.That(result, Is.EqualTo(3));

Blackbox testing:

We make a test without knowing actual logic inside the method

We start write possible test case that covers all execution paths

Setup and teardown

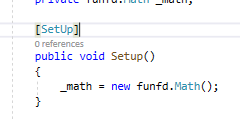
Associated with the arrange part of the code that may be a few lines of code in real world scenarios.

Setup:

Decorator called by the test runner before call the test method for initialization part

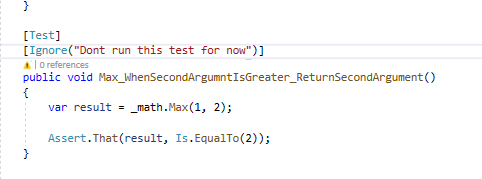
Tear down: attribute

Called when the test method is completed basically to perform the clean up in integration test



Ignoring test:

Basically used to exclude the test from the test instead of deleting or the commenting



Writing the trust worthy test:

Which we can trust that TDD

Core unit testing teqniques

Testing string:

Assert.That(result, Is.EqualTo("<strong>abc</strong>").IgnoreCase);

Assert.That(result, Does.StartWith("<strong").IgnoreCase);

Assert.That(result, Does.EndWith("</strong>").IgnoreCase);

Assert.That(result, Does.Contain("abc").IgnoreCase);

Testing array:

//Assert.That(result, Is.Not.Empty);

//Assert.That(result.Count, Is.EqualTo(3));

//Assert.That(result, Does.Contain(1));

//Assert.That(result, Does.Contain(3));

//Assert.That(result, Does.Contain(5));

Assert.That(result, Is.EquivalentTo(new[] { 1, 3, 5 }));

// Assert.That(result, Is.Ordered);

//Assert.That(result, Is.Unique);

Testing return type:

Assert.That(res, Is.TypeOf<Ok>()); (exact match of the type)

Assert.That(res, Is.InstanceOf<Ok>()); (exact match or derivative of the type)

Testing void methods (without return values):

These are command functions that does some changes to the file od db or perform some actions.

Or change the state of an object

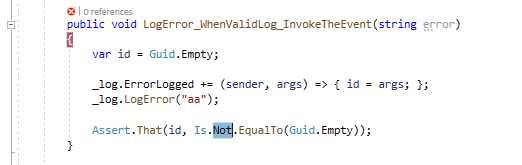
Test method that throws an exception

Here we cannot use normal asset result check we have to use the delegate to compare the results



Testing the event in the method:

For this first subscribe to the event in the test method then read the value from the event



Testing private methods:

Shouldn’t implement the private method

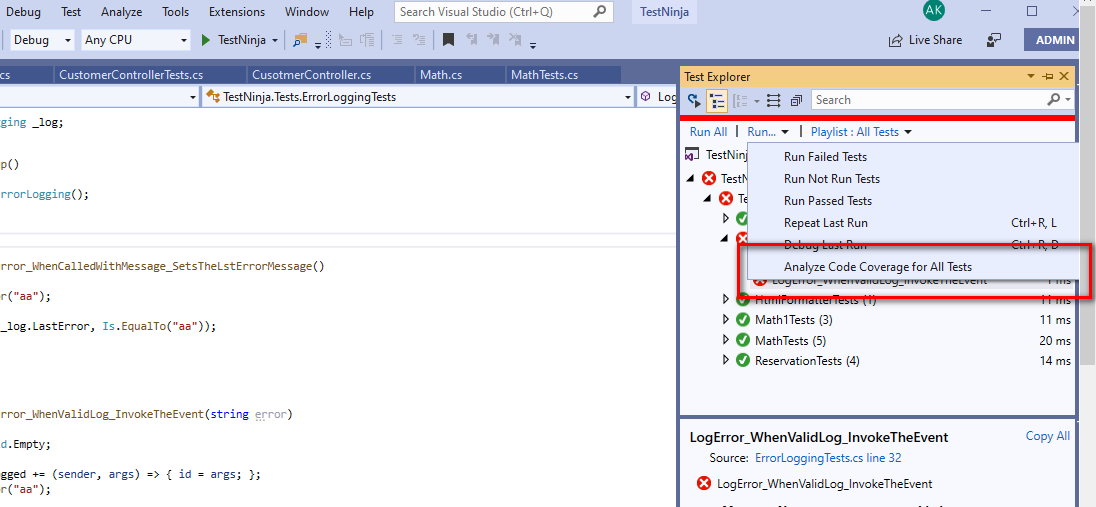
Always test public methods and Api

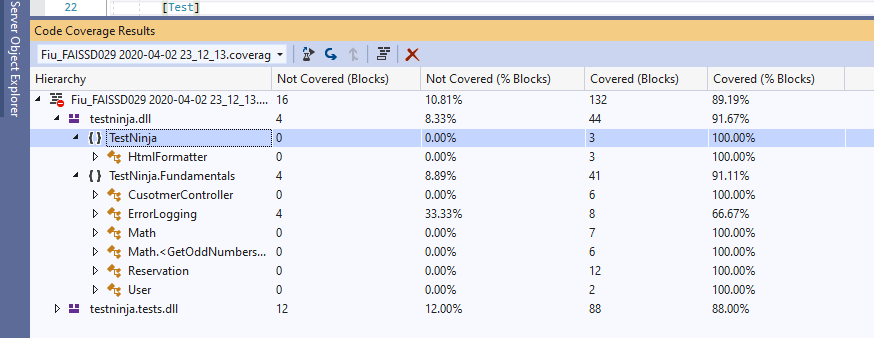
Code coverage tools:

Toll scans the code and test and tells you which is covered in test and what not.

Reshaper ultimate

VS Enterprise editon





Exercises

FIzzBuzz

Print fizz if the number divisible by 3

Print Buzz if it is divisible by 5

Print FizzBuzz if it is divisible by both 3 and 5.

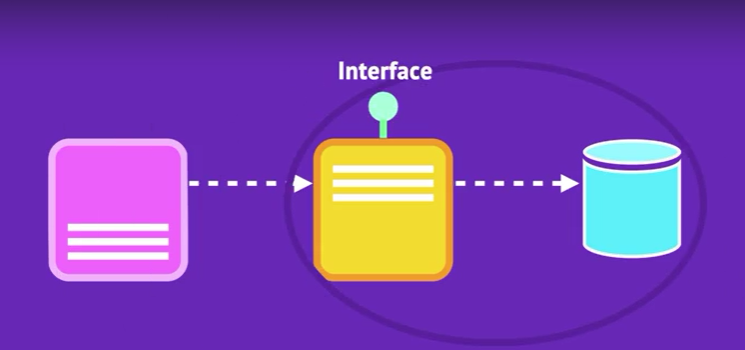
Breaking external dependency:

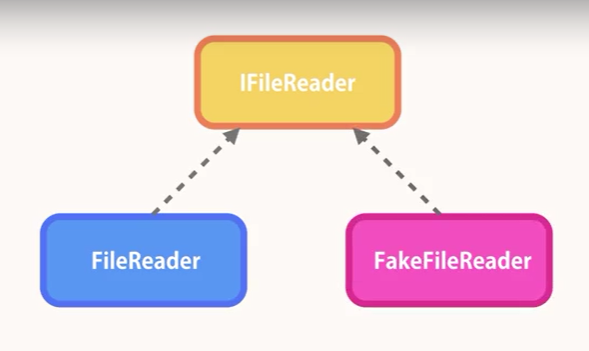
Test that touches externa; dependency called integration testing

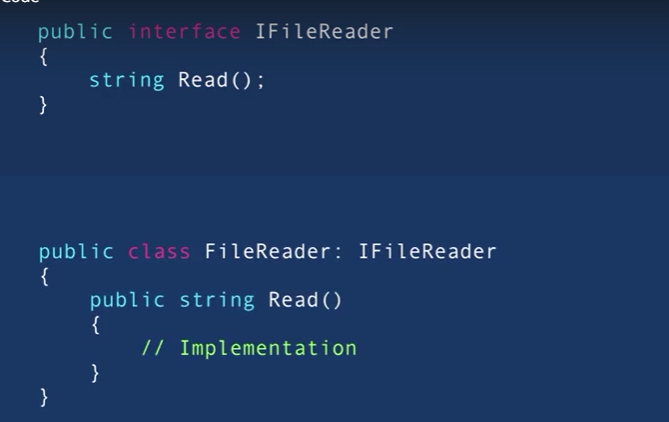


Class accessing the data from DB

Fake or test double class to fake the external object







Mock objects: (basically we are making the fake objects to test our real world object and the class)

Stubs

Mocks

Use the concept of dependency injection to pass the object to the low level calls to use the implementation

Concepts of Di make sure that eliminate the tight coupling of the class by passing objects created as a parameter to the low low level class tht provide the implementation.

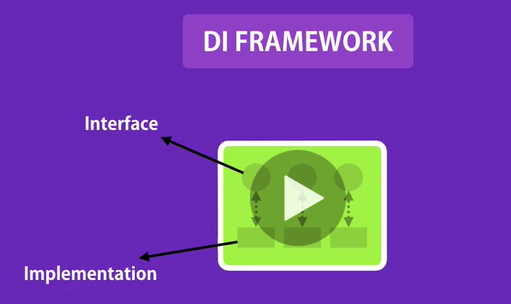
Di using property:

Di using method parameter:

DI using constructor:

In real world enterprise level application we have DI framework

Which basically has the container with all interfaces and the implementation of this interfaces

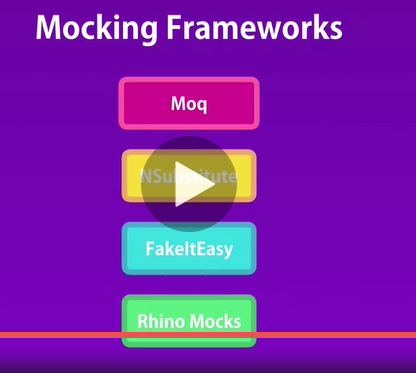


Famous DI framework:



Mocking:

Using this frame work we can create entire object to be tested and



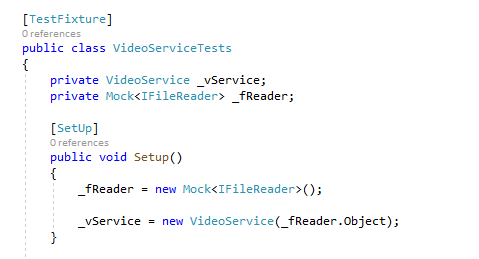
Here we are using moq frame work

Install from Nuget Packaage manager

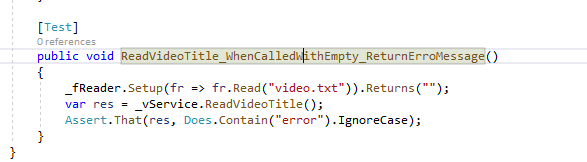
Refer

using Moq;

Create an instance of the mock object

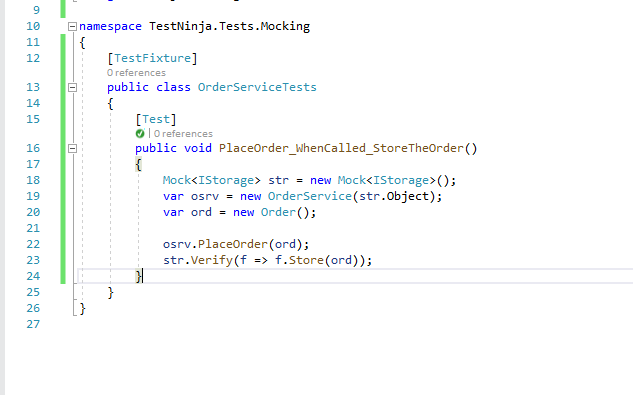


Implement mock to implement the interface



State based Vs Interaction based testing:

In order to compare if the two objects interaction we need use verify method of the Mock objects

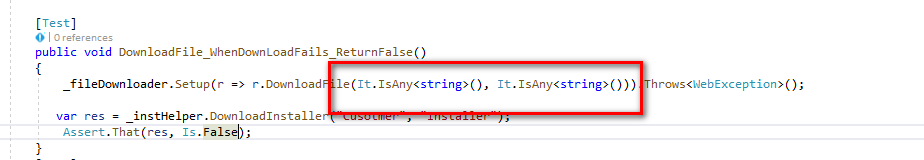


Abusing mock

Exercises:

1. VideoService
2. Installer Helper

To generate any random generic string



1. Employee helper