Unit Testing

Lesson Objectives

- Unit Test Fundamentals
- Unit Test Technique
- Nunit

Section 1

UNIT TESTING BASICS

Lesson Objectives

The course helps attendees understand:

- ✓ Unit Test Fundamentals: Answer the question of what, why, when and how to do unit test
- ✓ Unit Test Technique: Introduce approaches and techniques to do unit test

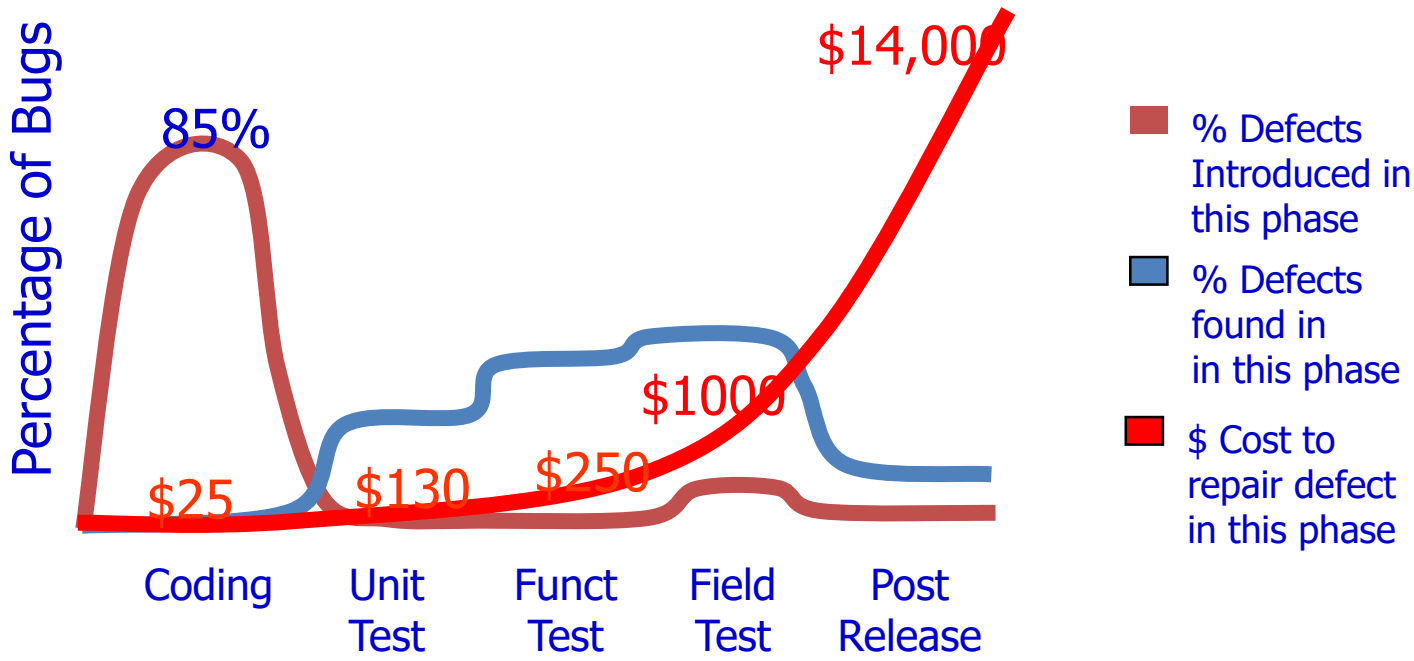
Unit Test – What and Who ?

- Unit Testing Actions:
  - ✓ Validate that individual units of software program are working properly
  - ✓ A unit is the smallest testable part of an application (In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is always a method)
  - ✓ Units are distinguished from modules in that modules are typically made up of units
- Unit Testing Deliverables:
  - ✓ Tested software units
  - ✓ Related documents (Unit Test case, Unit Test Report)
- Unit Testing Conductor: Development team

Unit Test – Why ?

- Ensure quality of software unit
- Detect defects and issues early
- Reduce the Quality Effort & Correction Cost

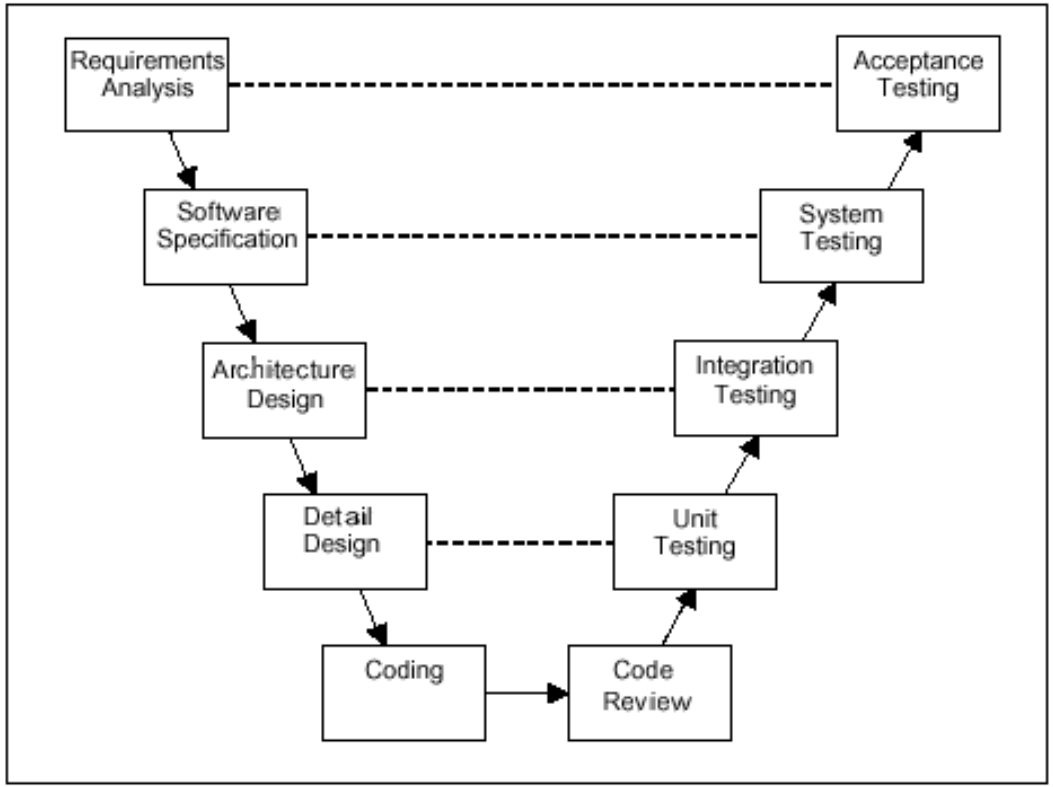
Cost of bugs



Source: Applied Software Measurement, Capers Jones, 1996

Unit Test – When ?

- After Coding
- Before Integration Test

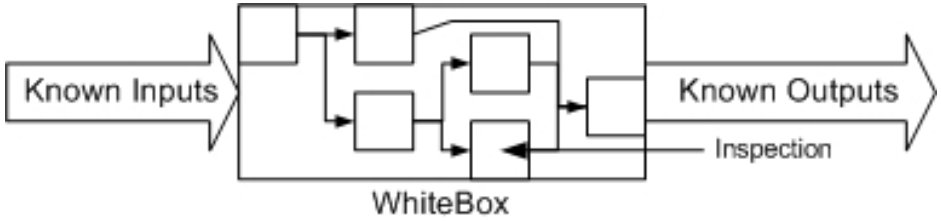


Unit Test – How ?(1) Methodologies

A Simple Black box Specification

- Black-box testing
  - ✓ Functional testing: ensure each unit acts right as its design
  - ✓ Business testing: ensure the software program acts right as user requirement

Unit Test – How ?(2) Methodologies



- White-box testing
  - ✓ Developer does himself
    - Check syntax of code by compiler to avoid syntax errors
    - Run code in debug mode, line by line, through all independent paths of program to ensure that all statement of codes has been executed at least one time
    - Examine local data structure to ensure that data stored temporarily maintains its integrity during all all steps of code execution
    - Check boundary conditions to ensure that code will run properly at the boundaries established as requirements
    - Review all error handling paths
  - ✓ Apply WB Test techniques

Unit Test – How ? Techniques

- Black box test (Functional)
  - ✓ Specification derived tests
  - ✓ Equivalence partitioning
  - ✓ Boundary value analysis
- White box (Structural)
  - ✓ Statement coverage
  - ✓ Decision (branch) coverage
  - ✓ Path coverage

Black box test – Specification derived test

- You can choose all or some statements in the specification of software
- Create test cases for each statements of specification
- Execute test cases to check test result will output as the specification

Example Specification

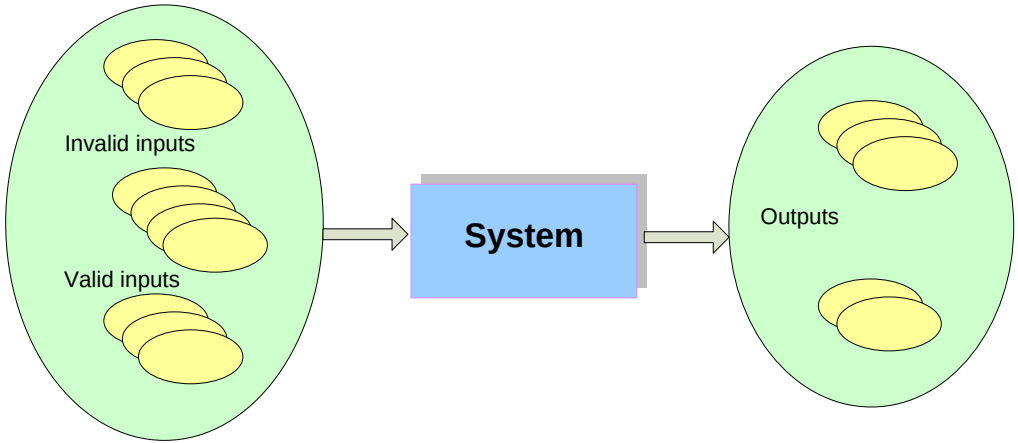
- Method: Positive square root of the positive number
- Input - real number
- Output - real number
- When given an input of 0 or greater, the positive square root of the input shall be returned.
- When given an input of less than 0, the error message "Square root error - illegal negative input" shall be displayed and a value of 0 returned

Example Test Cases

- **Test Case 1**: Input 4, Return 2
  - ✓ Use the first statement in the specification
  - ✓ ("When given an input of 0 or greater, the positive square root of the input shall be returned.").
- **Test Case 2**: Input -10, Return 0, Output "Square root error - illegal negative input"
  - ✓ Use the second and third statements in the specification
  - ✓ ("When given an input of less than 0, the error message "Square root error - illegal negative input" shall be displayed and a value of 0 returned.").

Black box test: Equivalence partitioning

- Divide the input of a program into classes of data from which test cases can be derived. This might help you to reduce number of test cases that must be developed.
- Behavior of software is equivalent for any value within particular partition
- A limited number of representative test cases should be chosen from each partition



Example Test Cases

Input partitions		Output partitions	
1	>= 0	**a**	>= 0
2	< 0	**b**	Error

- **Test Case 1**: Input 4, Return 2
  - Use the >=0 input partition (1)
  - Use the >=0 output partition (a)

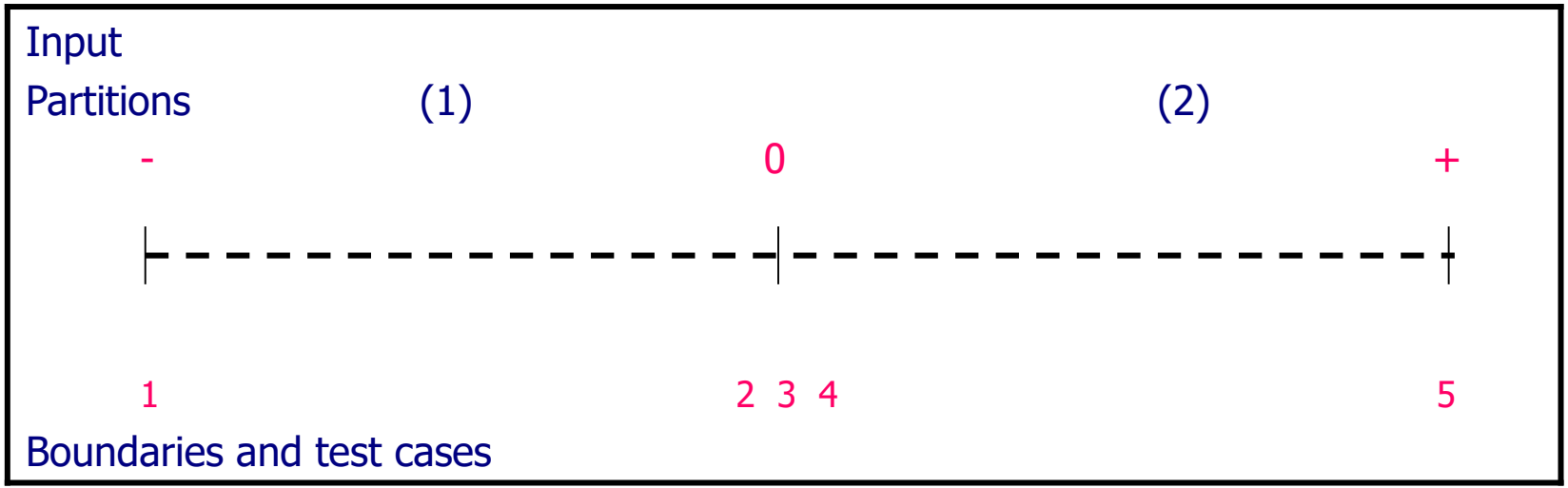
- **Test Case 2**: Input -10, Return 0, Output "Square root error - illegal negative input"
  - Use the < 0 input partition (2)
  - Use the "Error" output partition (b)

Black box test: Boundary value analysis

- It is similar to equivalence partitioning, is a selection of test cases, test input that check bounding values
- Anticipate that errors are most likely to exist at the boundaries between partitions
- Test the software at either side of boundary values

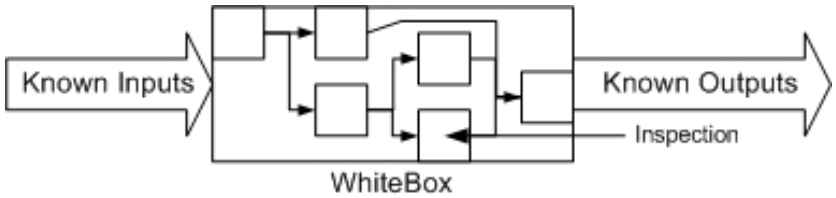
Example Test Cases

Input partitions		Output partitions	
1	>= 0	**a**	>= 0
2	< 0	**b**	Error



White Box Testing – What?

- **WHITE BOX TESTING** (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested



White Box Testing – Example

- A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.
- White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

Levels Applicable To

- White Box Testing method is applicable to the following levels of software testing:
  - ✓ Unit Testing: For testing paths within a unit.
  - ✓ Integration Testing: For testing paths between units.
  - ✓ System Testing: For testing paths between subsystems.

- However, it is mainly applied to Unit Testing.

White Box Testing – Advantages

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

White Box Testing – Disadvantages

- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

White Box Testing - Techniques

- 3 Main White Box Testing Techniques:
  - ✓ Statement Coverage
  - ✓ Branch Coverage
  - ✓ Path Coverage

- Before understanding the three techniques above, we need to understand the concept of “node listing".

White-box testing
Techniques

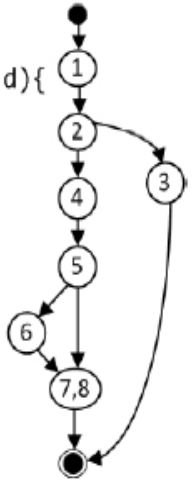
- **White box (Structural)**
  - ✓ Statement coverage
  - ✓ Decision (branch) coverage
  - ✓ Path coverage
- **Calculating Structural Testing Effectiveness:**
  - ✓ Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %
  - ✓ Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %
  - ✓ Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100 %

White box test

Statement coverage

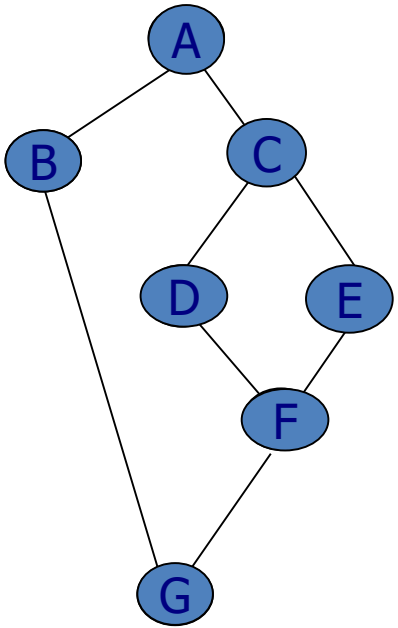
- The **statement coverage** is also known as **line coverage** or **segment coverage**.
  - ✓ In this process **each and every line of code needs to be checked**.

```
float foo(int a, int b, int c, int d){
1.  float e;
2.  if (a==0)
3.     return 0;
4.  int x = 0;
5.  if ((a==b) || (c==d))
6.    x = 1;
7.  e = 1/x;
8.  return e;
}
```



ID	Inputs	EO	RO	Note
tc1	0, 1, 2, 3	0		
tc2	1, 1, 2, 3	1		

White box test
Statement coverage



Total Nodes = 7;

Test case ABG covers 3 = 43%

+

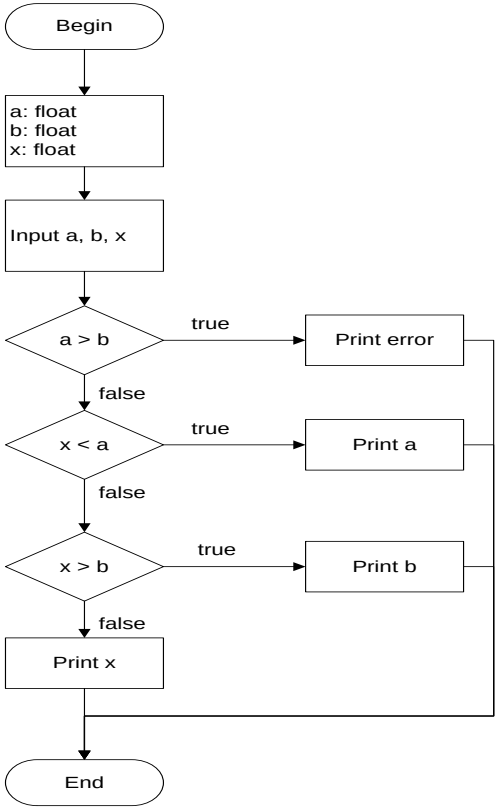
Test case ACDFG

Now covers 6/7 = 86%

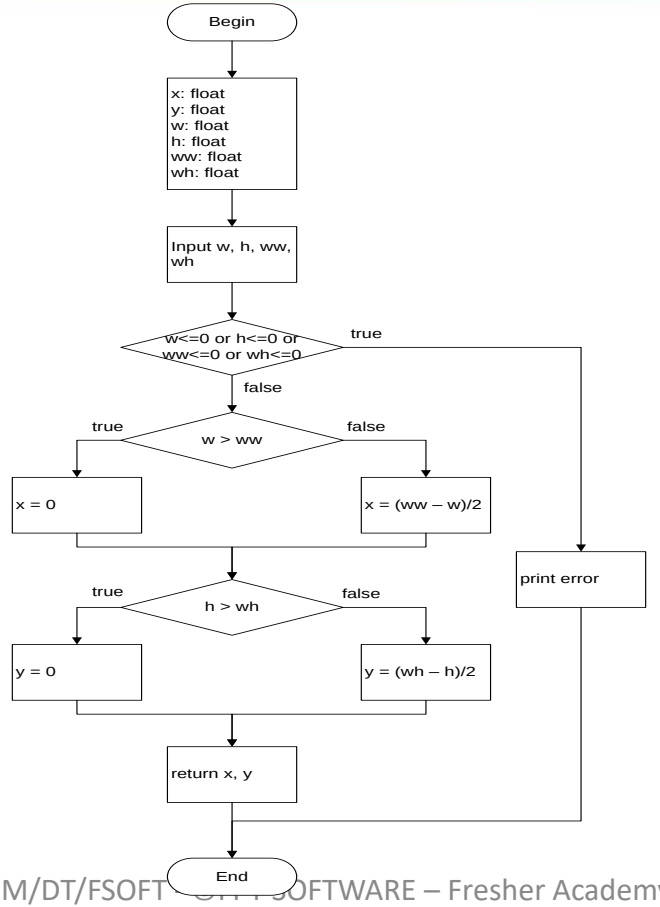
Need 1 more for 100% statement coverage – ACEFG

White box test
Decision (branch) coverage

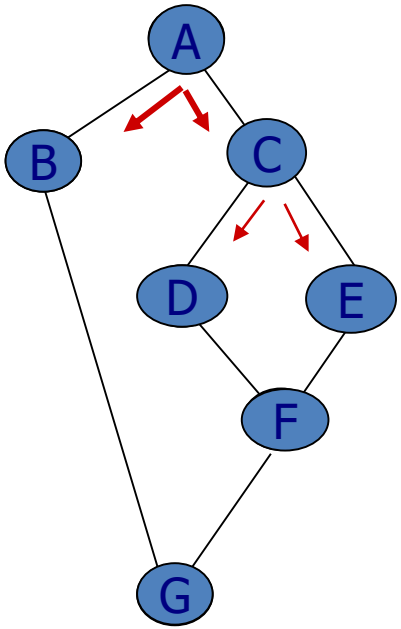
- **Decision coverage/Branch coverage** is a testing method, which aims[mục tiêu] to ensure that **each one of the possible branch from each decision point is executed at least once**.



White box test
Decision (branch) coverage



White box test
Decision (branch) coverage



What branch coverage is achieved by ABG, ACDFG, ACEFG?

4 in total.

4 covered

So 4/4 = 100% branch coverage

White box test
Path coverage

- **Path Coverage (PC)**: Path Coverage ensures covering of all the paths from start to end.
- **Example:** All possible paths are-

  1A-2B-E-4F

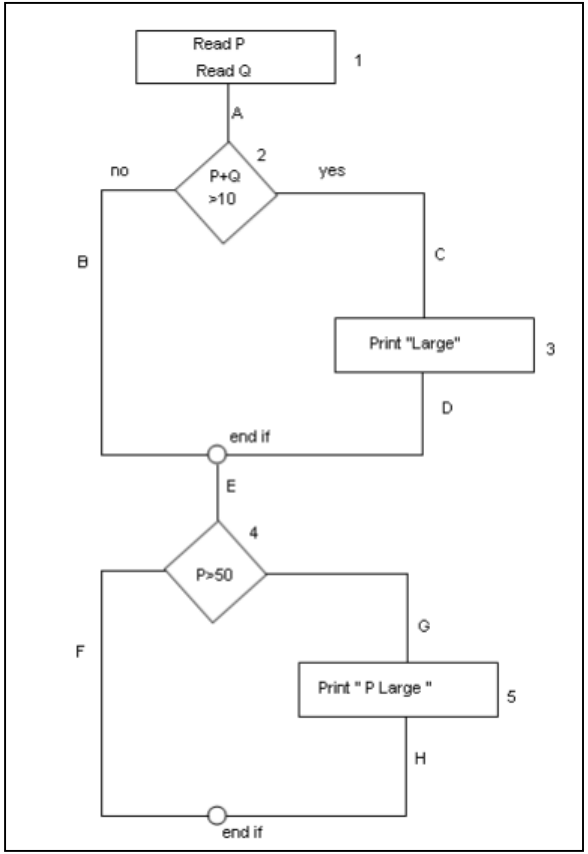
  1A-2B-E-4G-5H

  1A-2C-3D-E-4G-5H

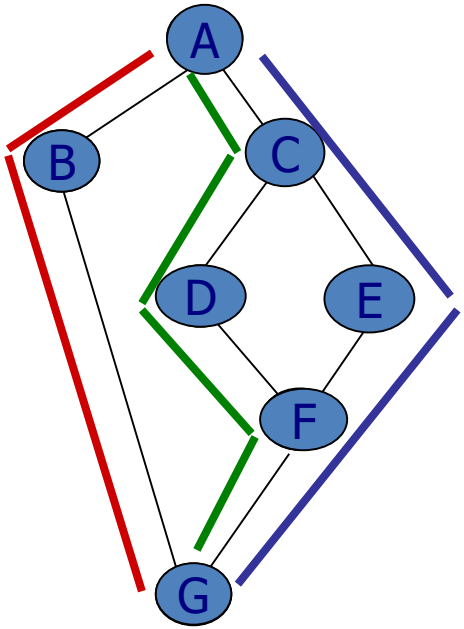
  1A-2C-3D-E-4F

➔ So path coverage is 4.

- **Thus for the above example** SC=1, BC=2 and PC=4.



White box test
Path coverage



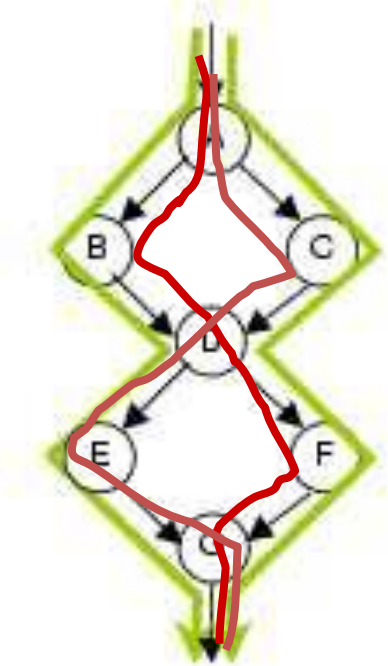
What path coverage is achieved by ABG, ACDFG, ACEFG?

3 in total.

3 covered

So 3/3 = 100% path coverage

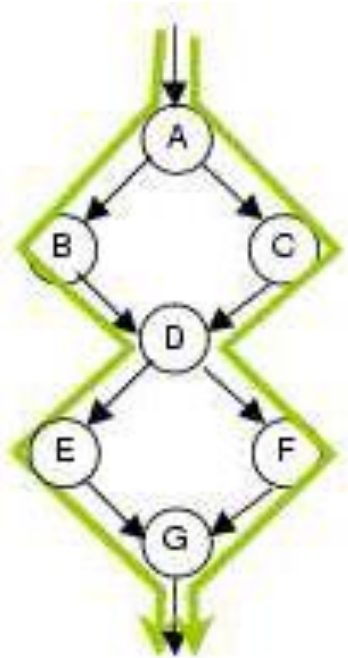
White box test case



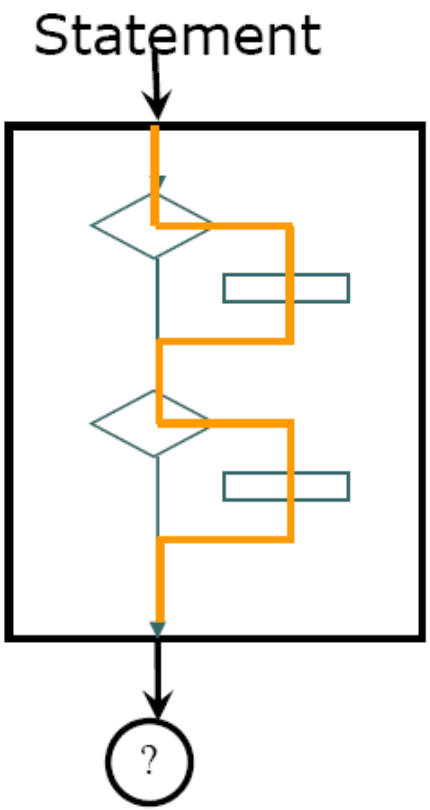
- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%).
- They, however, only cover 2/4 paths (50%).
- 2 more tests are required to achieve 100% path coverage.
  - ABDFG
  - ACDEG

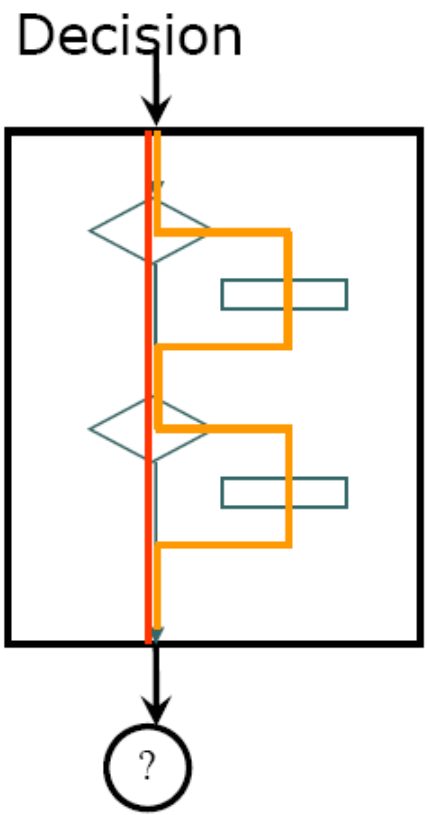
White box test: Example

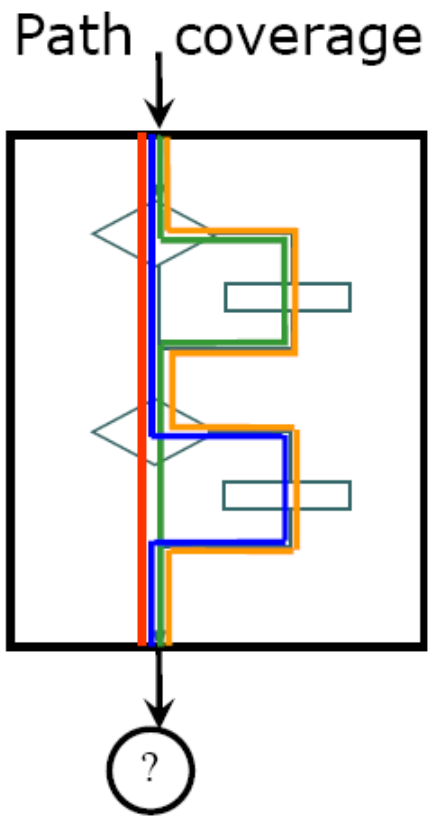
- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).
- 2 more tests are required to achieve 100% path coverage
  - ABDFG
  - ACDEG



White box test: Comparison







Differences Between Black & White Box

Criteria	Black Box Testing	White Box Testing
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
Levels Applicable To	Mainly applicable to higher levels of testing:Acceptance Testing System Testing	Mainly applicable to lower levels of testing:Unit Testing Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design

Section 2

NUNIT

Lesson Objectives

- Introduction to Nunit
- Install and Configure
- Write test
- Run test
- Report

NUnit

- NUnit is a unit testing framework for .NET.
- It is the most used framework for writing unit test cases.
- We can write testing code in C#
- NUnit is very easy to use.
- Assert classes is used to test the conditions whether system under test (SUT) satisfy a condition or not.

Custom Attributes

- TestFixture
- Setup
- TearDown
- Test
- Category
- Ignore
- TestCase
- Repeat
- MaxTime

Install and Configure

- Step 0: Create Projects
  - ✓ Always creates separate project when creating project for NUnit.
  - ✓ Naming conventions test project name should be [Project Under Test].[Tests].
  - ✓ Use Class Library project template

Install and Configure

- Practice: Create Projects
  - ✓ Create new solution
  - ✓ Create new project
    - Name: FA.Training
    - Template: Class Library
    - Edit Class1 to Student
    - Add new method inside Student class to get Average of 3 numbers:

```
public decimal Average(decimal a, decimal b, decimal c)
    {
        return (a + b + c) / 3;
    }
```

Install and Configure

- Practice: Create Projects (cont…)
  - ✓ Create new test project
    - Name: FA.Training.Test
    - Template: Class Library

Install and Configure

- Step 1: Install NUnit
  - ✓ Install NUnit package from nugget test project
    - Install-Package NUnit -Version 3.12.0
  - ✓ Install NUnit3TestAdapter package from nugget to test project
    - Install-Package NUnit3TestAdapter -Version 3.14.0
  - ✓ Add reference from project test to project need to test
    - From FA.Training.Test, add reference to FA.Training to use all method from FA.Training

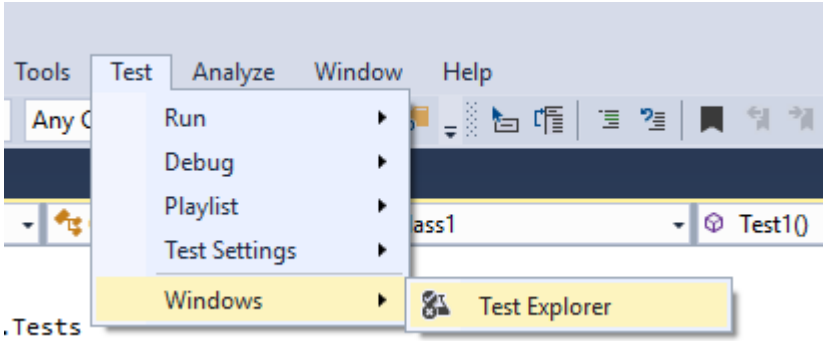
Write test case

- Step 3: Write test case
  - ✓ In project FA.Training.Test, change Class1 to StudentTests
  - ✓ Add test method:

```
namespace FA.Training.Test
{
    [TestFixture]
    0 references
    public class StudentTests
    {
        [TestCase]
        0 references
        public void GetStudentMark()
        {
            Student student = new Student();
            var avgMark = student.Average(7, 9, 9);
            Assert.That(avgMark == 8.33m);
        }
    }
}
```

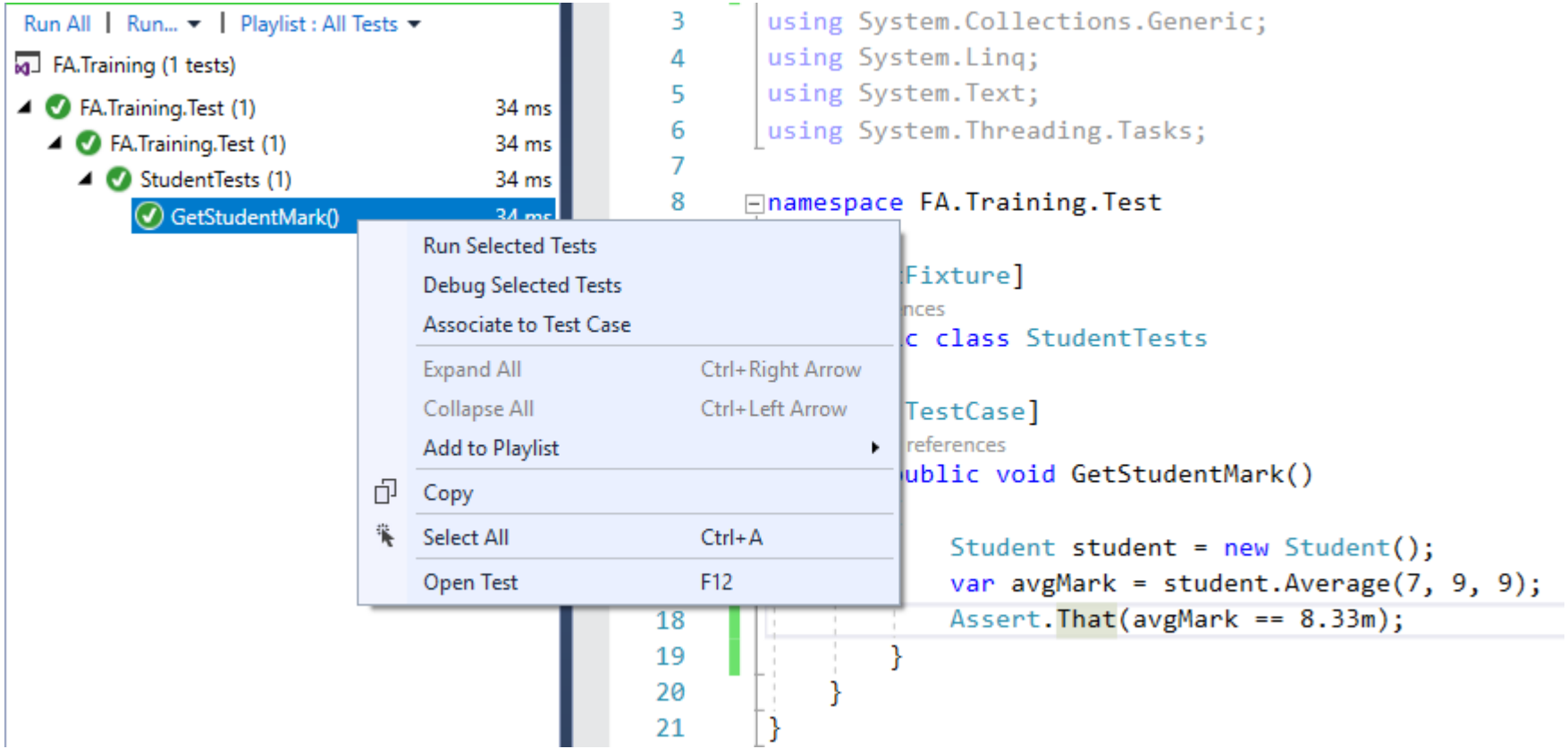
Run test case

- Step 4: Run test case
  - ✓ Choose visual studio Test Menu -> Windows -> Test Explorer

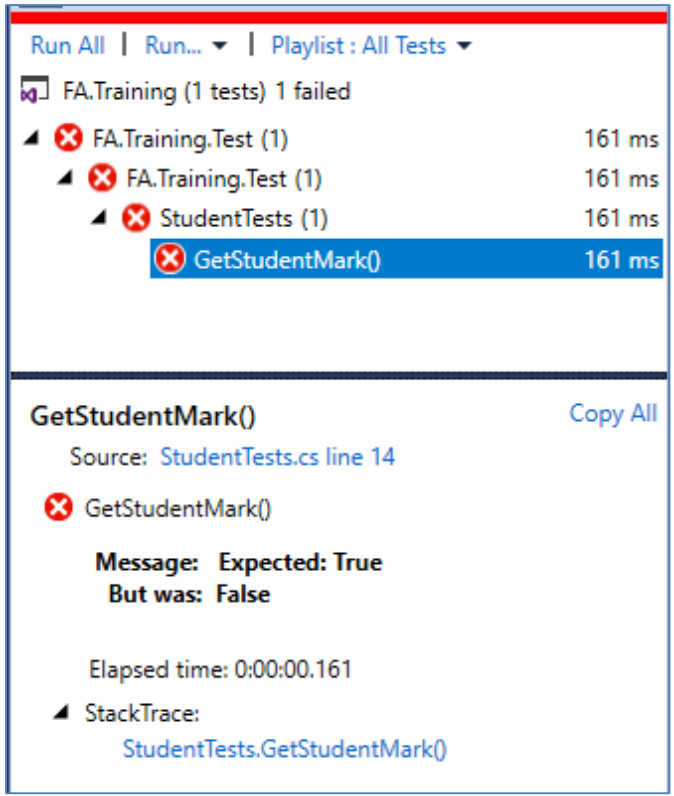


Run test case

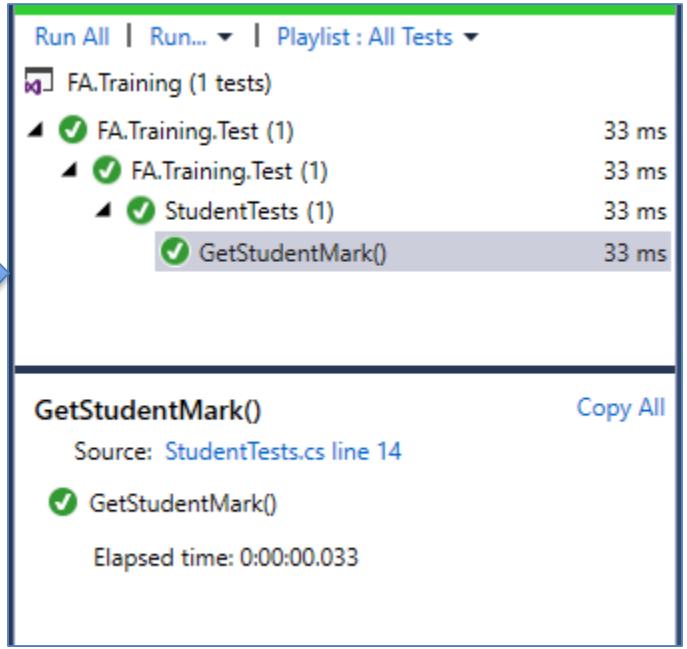
- Step 4: Run test case
  - ✓ Explorer test case then Run Selected Test


Run All | Run... | Playlist : All Tests
FA.Training (1 tests)
FA.Training.Test (1) 34 ms
FA.Training.Test (1) 34 ms
StudentTests (1) 34 ms
GetStudentMark()
Run Selected Tests
Debug Selected Tests
Associate to Test Case
Expand All Ctrl+Right Arrow
Collapse All Ctrl+Left Arrow
Add to Playlist
Copy
Select All Ctrl+A
Open Test F12
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace FA.Training.Test
Fixture]
nces
c class StudentTests
TestCase]
references
ublic void GetStudentMark()
Student student = new Student();
var avgMark = student.Average(7, 9, 9);
Assert.That(avgMark == 8.33m);
}
}
}

Check result



Resolve



Assert Class

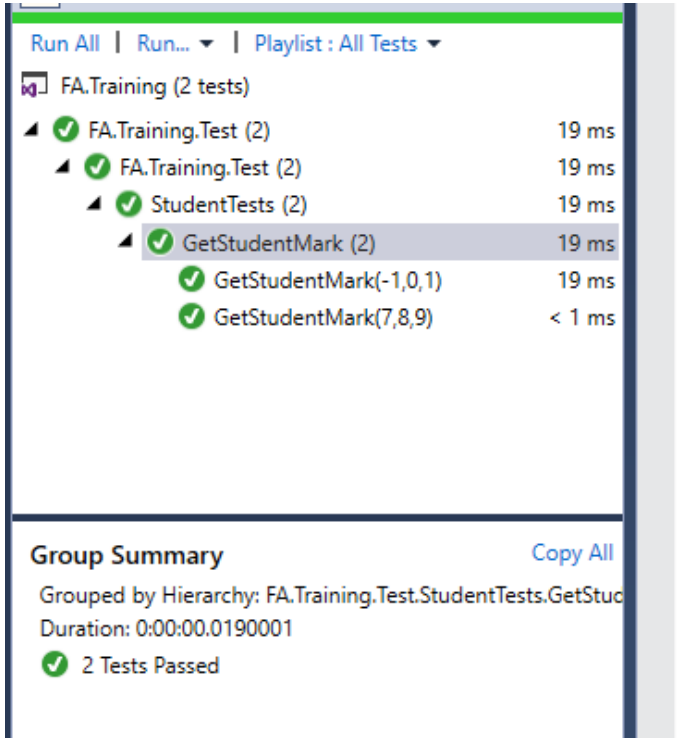
- Assert is NUnit framework class which has static methods to verify the expected behaviour.
- If the Assert condition is passed then the NUnit test case is passed else failed.

Assert Methods

- Assert.That
- Assert.AreEqual/AreNotEqual
- Assert.AreSame/AreNotSame
- Assert.Equals
- Assert.Contains
- Assert.Greater/ GreaterOrEqual
- ….

TestCase Attribute

- TestCase arguments: use same test case with different data.
- ExpectedResult: specify different results for different parameters.
- Author: specify author name in the test method who has written the test case
- TestName: use different name than the specified test method name.




```
using NUnit.Framework;

namespace FA.Training.Test
{
    [TestFixture]
    0 references
    public class StudentTests
    {
        [TestCase(-1, 0, 1, ExpectedResult = 0)]
        [TestCase(7, 8, 9, ExpectedResult = 8)]
        0 references
        public decimal GetStudentMark(decimal a, decimal b, decimal c)
        {
            Student student = new Student();
            return student.Average(a, b, c);
        }
    }
}
```

References

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
- https://www.dofactory.com/reference/csharp-coding-standards
- https://docs.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2019

Thank you