**NUnit and MOQ Handson Solutions**

**Date:**09-06-2021

**Name:**Asritha Cherukuri

**EmpId:** 916179

**Handson1**

*Given:*

The method called GetEmployees returns a collection of Employee object.

Scenario 1

· Ensure that there is no null value in the collection

Scenario 2

· Verify whether the employee having his/her id 100 exists in the collection.

Scenario 3

· Check whether the GetEmployees function returns only unique employees. If employee id is different in every employee object, then the list is considered as unique list. You may modify the source project so that you can achieve the goal here.

Hint: Override Equals() & GetHashCode() methods in Empoyee class.

Scenario 3

· Both GetEmployees() and GetEmployeesWhoJoinedInPreviousYears() return a set of employee object. Verify whether all items in both the collections are same or not.

Try both Classic Model as well as Constraint Model of Assertions while writing test cases.

*Given Problem Code:*

*EmployeeManager.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CollectionsLib

{

public class Employee

{

public int EmpId { get; set; }

public string EmpName { get; set; }

public double Salary { get; set; }

public DateTime DOJ { get; set; }

}

public class EmployeeManager

{

private static readonly List<Employee> employees;

static EmployeeManager()

{

employees = new List<Employee>

{

new Employee { EmpId=100, EmpName="John",DOJ=DateTime.Now.AddYears(-5),Salary=30000},

new Employee { EmpId=101, EmpName="Mary",DOJ=DateTime.Now.AddYears(-2),Salary=10000},

new Employee { EmpId=102, EmpName="Steve",DOJ=DateTime.Now.AddYears(-2),Salary=10000},

new Employee { EmpId=103, EmpName="Allen",DOJ=DateTime.Now.AddYears(-7),Salary=50000},

};

}

public List<Employee> GetEmployees()

{

return employees;

}

public List<Employee> GetEmployeesWhoJoinedInPreviousYears()

{

return employees.FindAll(x=>x.DOJ<DateTime.Now);

}

}

}

---------------------------------------------------------------------------------------

*Implementation Code:*

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using CollectionsLib;

using NUnit.Framework;

using System.Linq;

using Assert = NUnit.Framework.Assert;

namespace UnitTestProject1

{

[TestFixture]

public class UnitTest1

{

[Test]

public void Test1()

{

EmployeeManager emp = new EmployeeManager();

var cnull = emp.GetEmployees().Contains(null);

Assert.AreEqual(cnull, false);

var l1 = emp.GetEmployees();

//check empid 100

int empno = l1.Where(e => e.EmpId == 100).Count();

Assert.That(empno, Is.EqualTo(1));

//unique

var u = l1.Distinct().Count();

Assert.That(u, Is.EqualTo(l1.Count));

var ans = emp.GetEmployees();

var result = emp.GetEmployeesWhoJoinedInPreviousYears();

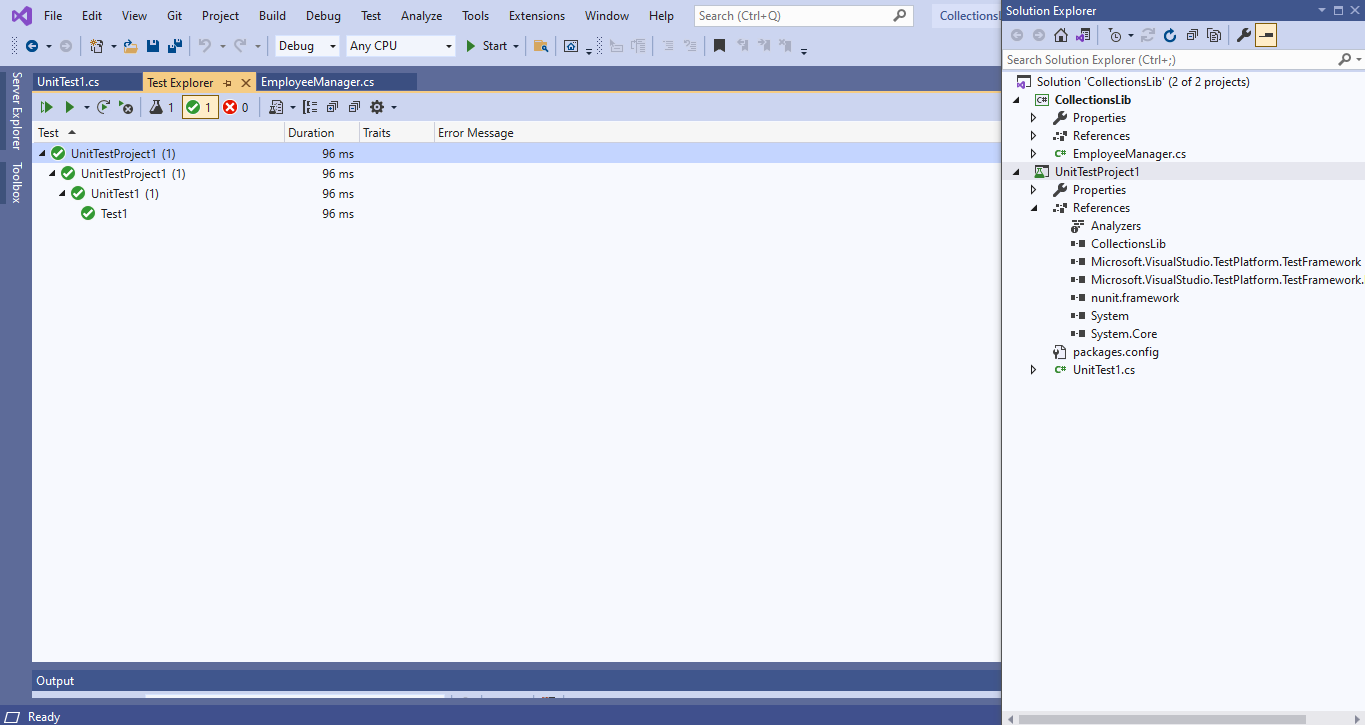
Assert.That(ans, Is.EquivalentTo(result));

}

}

}

*Results:*

**

**Handson**

*Given:*

Following is the business scenario implemented in the source project.

· The function accepts a month name and returns what season in that month (both inclusive) based on certain logic. The criteria is elaborated in the matrix below.

Seasons Month Climate

Spring February to March Sunny and pleasant

Summer April to June Hot

Monsoon July to September Wet, hot and humid

Autumn September to November Pleasant

Winter December to January Very Cool

Make sure that you are not writing multiple test methods that are equal to the number of execution paths. Your focus should be of writing minimum code while unit testing.

Try both straight forward and alternate ways of working with the TestCaseSource attribute for the above scenario.

*Given Problem Code:*

*Season.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SeasonsLib

{

public class SeasonTeller

{

public string DisplaySeasonBy(string monthName)

{

string seasonName;

if (monthName.Equals("February", StringComparison.OrdinalIgnoreCase) || monthName.Equals("March", StringComparison.OrdinalIgnoreCase))

{

seasonName = "Spring";

}

else if (monthName.Equals("April", StringComparison.OrdinalIgnoreCase) || monthName.Equals("May", StringComparison.OrdinalIgnoreCase) || monthName.Equals("June", StringComparison.OrdinalIgnoreCase))

{

seasonName = "Summer";

}

else if (monthName.Equals("July", StringComparison.OrdinalIgnoreCase) || monthName.Equals("August", StringComparison.OrdinalIgnoreCase) || monthName.Equals("September", StringComparison.OrdinalIgnoreCase))

{

seasonName = "Monsoon";

}

else if (monthName.Equals("October", StringComparison.OrdinalIgnoreCase) || monthName.Equals("November", StringComparison.OrdinalIgnoreCase))

{

seasonName = "Autumn";

}

else if (monthName.Equals("December", StringComparison.OrdinalIgnoreCase) || monthName.Equals("January", StringComparison.OrdinalIgnoreCase))

{

seasonName = "Winter";

}

else

{

return "Invalid Season";

}

return seasonName;

}

}

}

*-------------------------------------------------------------------------------------------------------------*

*Implementation Code:*

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using NUnit.Framework;

using SeasonsLib;

using Assert = NUnit.Framework.Assert;

namespace UnitTestProject2

{

[TestFixture]

public class UnitTest1

{

[Test]

[TestCaseSource(nameof(str))]

public void Test1(string s1, string s2)

{

SeasonTeller s = new SeasonTeller();

Assert.That(s2, Is.EqualTo(s.DisplaySeasonBy(s1)));

}

static object[] str =

{

new object[] { "April", "Summer" },

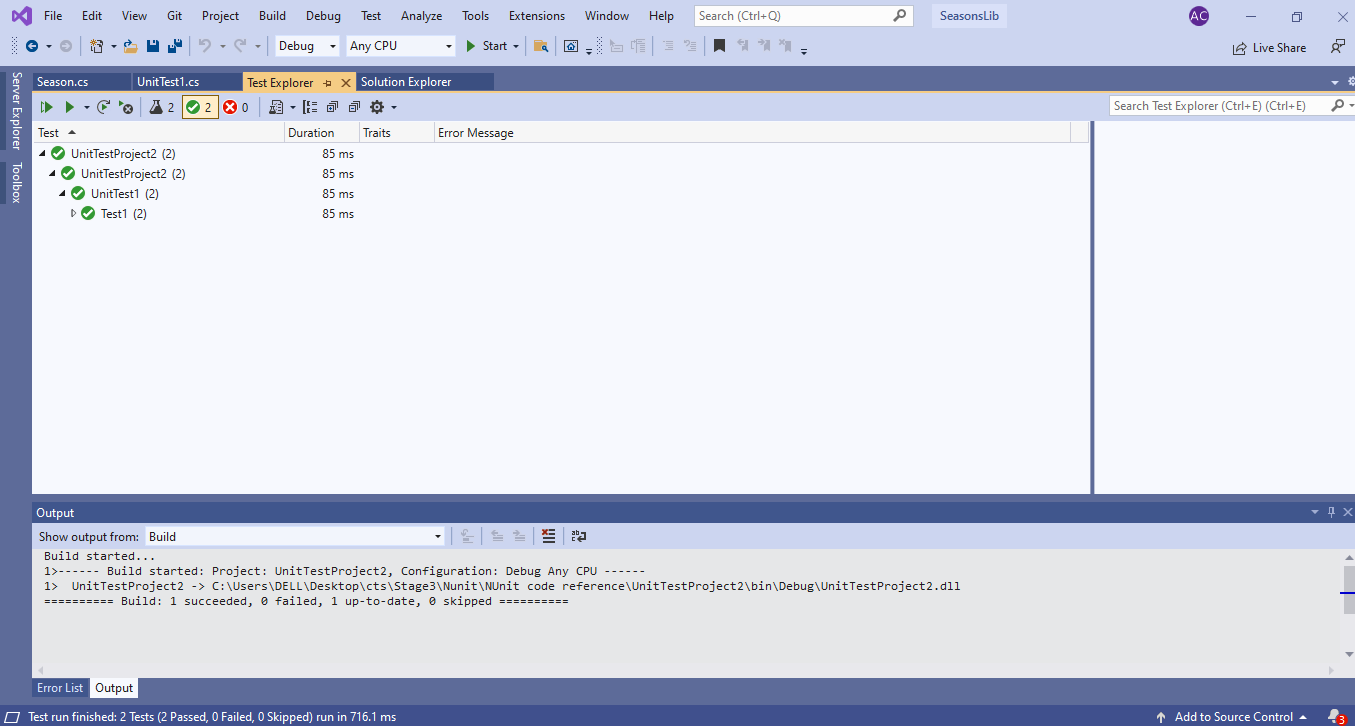
new object[] { "October", "Autumn" }

};

}

}

*Results:*

**

**Handson**

*Given:*

The LeapYearCalculatorLib application tells the user whether the entered year is a leap year or not. Also it checks the input value to make sure that the data given is a valid year.

· If the given year is a Leap Year, the program will output 1. If it’s not, then the program will result 0.

· Any value between 1753 and 9999 (both inclusive) will be a valid year. Violation of this rule will result -1.

*Given Problem Code:*

*LeapYearCalculator.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LeapYearCalculatorLib

{

public class LeapYearCalculator

{

public int IsLeapYear(int year)

{

int output;

if (year<1753||year>9999)

{

return -1;

}

if (((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))

output = 1;

else

output = 0;

return output;

}

}

}

*-------------------------------------------------------------------------------------------------------------*

*Implementation Code:*

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using NUnit.Framework;

using LeapYearCalculatorLib;

using Assert = NUnit.Framework.Assert;

namespace UnitTestProject3

{

[TestFixture]

public class UnitTest1

{

[Test]

[TestCase(1999, 0)]

[TestCase(2000, 1)]

[TestCase(1750, -1)]

public void Test(int yr, int result)

{

LeapYearCalculator lp = new LeapYearCalculator();

int ans = lp.IsLeapYear(yr);

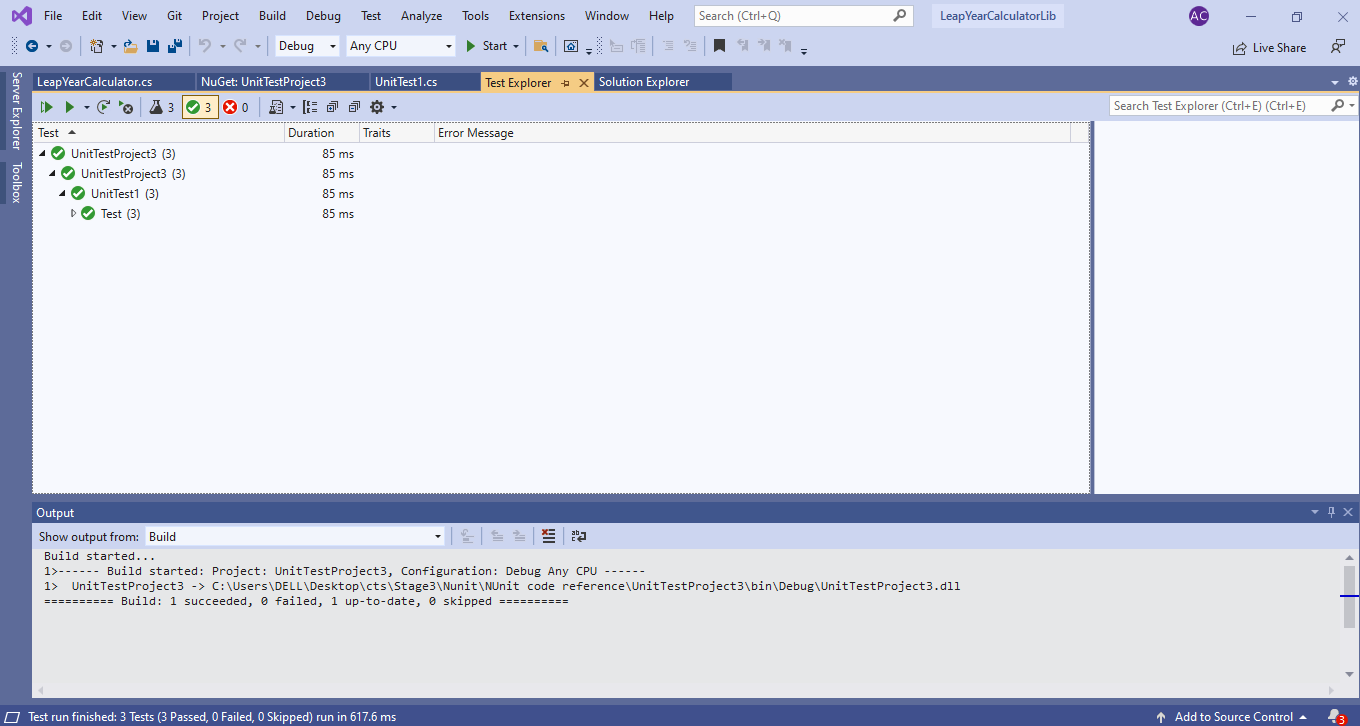
Assert.That(ans, Is.EqualTo(result));

}

}

}

*Results:*

**

**Handson**

*Given:*

Following is the application logic of the given UserManagerLib source project.

User creation will be successful under the below given condition

· PANCardNo property reads only 10 characters length value from the user. It is a mandatory property while creating the user.

Following exceptions may occur while creating the user.

o NullReferenceException- If the input value is empty or null

o FormatException-If the input string does not meet the length criteria.

· While writing test cases for the above program, you need to ensure that you are handling all types of exception that may raise during the CreateUser method call. Also, write test method for happy path in the function.

*Given Problem Code:*

*User.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace UserManagerLib

{

public class User

{

public Guid Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string EmailId { get; set; }

public string PANCardNo

{

get;

set;

}

public string ValidatePANCardNumber(string panCard)

{

if (string.IsNullOrEmpty(panCard))

{

throw new NullReferenceException("Invalid Pan Card Number");

}

else if (panCard.Length != 10)

{

throw new FormatException("Pan Card Number Should contain only 10 characters");

}

else

{

return "Valid";

}

}

public void CreateUser(User user)

{

if (ValidatePANCardNumber(user.PANCardNo).Equals("Valid"))

{

//Do something

}

}

}

}

*-------------------------------------------------------------------------------------------------------------*

*Implementation Code:*

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using NUnit.Framework;

using UserManagerLib;

using Assert = NUnit.Framework.Assert;

namespace UnitTestProject4

{

[TestFixture]

public class UnitTest1

{

[Test]

[TestCase("9553130666")]

[TestCase("CBENU4CSE1")]

public void Test(string check)

{

User u = new User();

try

{

u.CreateUser(new User { PANCardNo = check });

}

catch (NullReferenceException e)

{

Assert.Fail(e.Message);

}

catch (FormatException e)

{

Assert.Fail(e.Message);

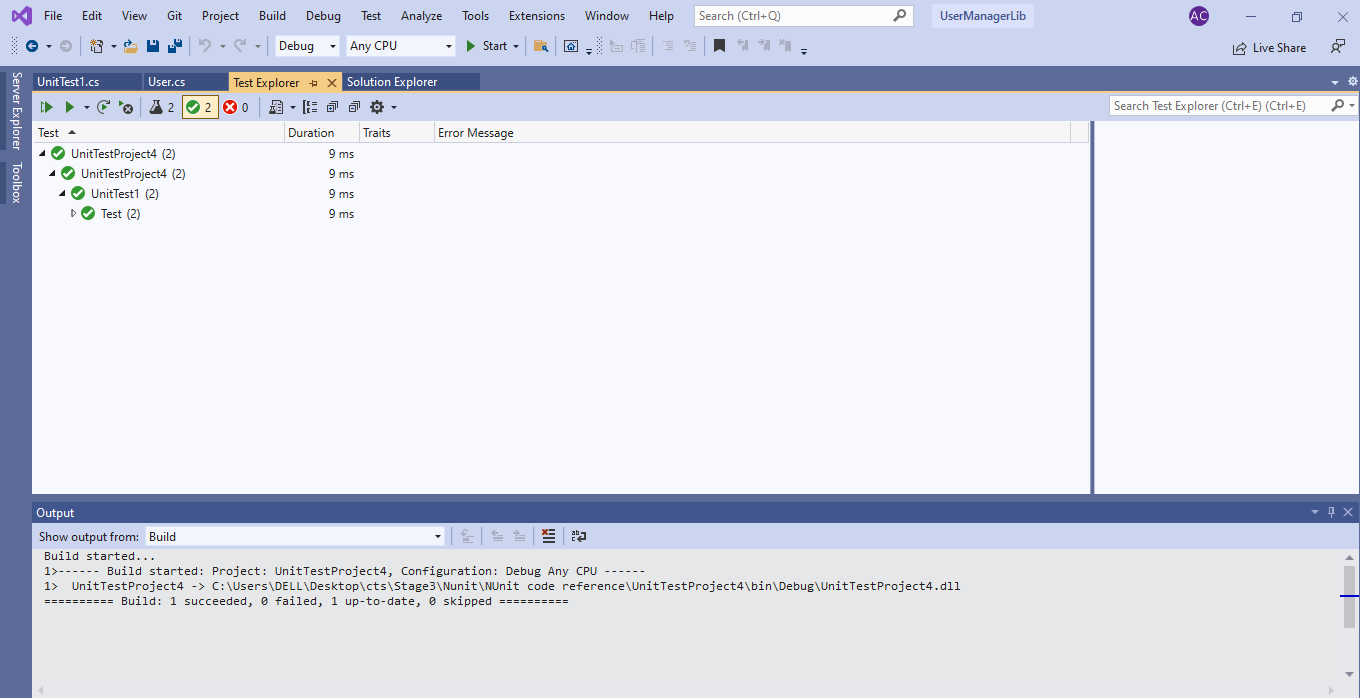
}

}

}

}

*Results:*

**

**Handson**

*Given:*

Create a Class Library project in the same solution which is provided and name it as suggested.

2) Rename the class file name (<SUT>Tests.cs).

3) Add the assembly reference of the ConverterLib project to the test project.

4) Additionally, add the reference of NUnit, NUnit3TestAdapter and Moq in the test project using NuGet Package Manager (NPM).

5) Write the suggested test methods.

6) Run your tests.

7) Break the test by modifying the source project functionality.

8) Rerun the test.

9) Observe the test result.

*Given Problem Code:*

*Converter.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using CurrencyConverterApp;

namespace ConverterLib

{

public class Converter : IConverter

{

IDollarToEuroExchangeRateFeed \_exchangeRateFeed;

public Converter(IDollarToEuroExchangeRateFeed exchangeRateFeed)

{

\_exchangeRateFeed = exchangeRateFeed;

}

public double CelsiusToKelvin(double celsius)

{

return celsius + 273.15;

}

public double KilogramToPound(double kilogram)

{

return kilogram \* 2.205;

}

public double KilometerToMile(double kilometer)

{

return kilometer / 1.609;

}

public double LiterToGallon(double liter)

{

return liter / 3.785;

}

public double USDToEuro(double dollar)

{

return dollar \* \_exchangeRateFeed.GetActualUSDollarValue();

}

}

}

*IConverter.cs*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConverterLib

{

public interface IConverter

{

double CelsiusToKelvin(double celsius);

double KilogramToPound(double kilogram);

double KilometerToMile(double kilometer);

double LiterToGallon(double liter);

double USDToEuro(double dollar);

}

}

*--------------------------------------------------------------------------------------------------------*

*Implementation Code:*

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using ConverterLib;

using NUnit.Framework;

using CurrencyConverterApp;

using Moq;

using Assert = NUnit.Framework.Assert;

namespace UnitTestProject5

{

[TestFixture]

public class UnitTest1

{

[Test]

public void Test()

{

double dll= 1.5;

double ab = 10000;

//double a;

Mock<IDollarToEuroExchangeRateFeed> ue = new Mock<IDollarToEuroExchangeRateFeed>();

ue.Setup(t => t.GetActualUSDollarValue()).Returns(dll);

Converter cv = new Converter(ue.Object);

var e = cv.USDToEuro(ab);

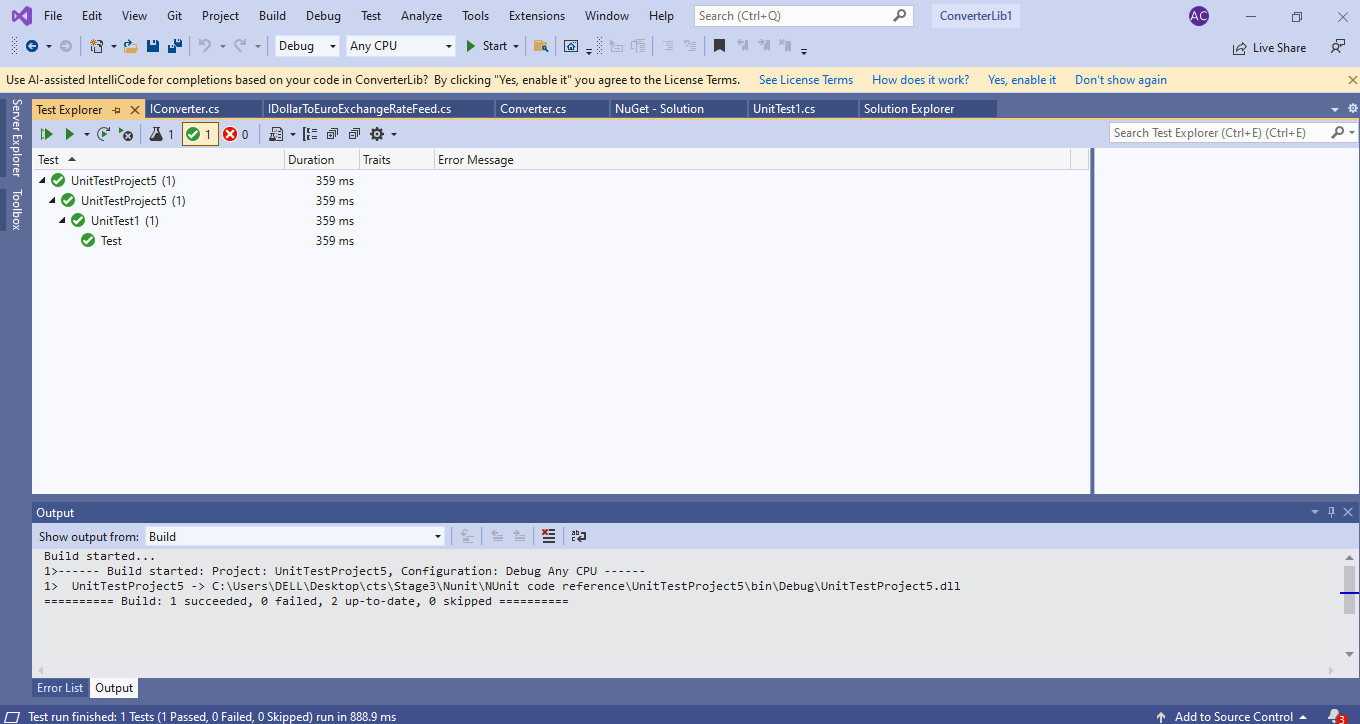
Assert.AreEqual(e, 15000);

}

}

}

*Results:*

**