Exercise:NUnit HandsOn

namespace CalculatorLibrary

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

public int Subtract(int a, int b)

{

return a - b;

}

// Add other operations as needed

}

}

using NUnit.Framework;

using CalculatorLibrary; // Assuming your Calculator class is in this namespace

namespace CalculatorLibrary.Tests

{

[TestFixture] // Marks a class that contains test methods

public class CalculatorTests

{

private Calculator \_calculator; // Declaring a private field for the calculator instance

[SetUp] // Method decorated with [SetUp] is executed before each test method

public void Setup()

{

\_calculator = new Calculator(); // Initialize a new Calculator instance before each test

}

[Test] // Marks a method as a test method

public void Add\_TwoPositiveNumbers\_ReturnsCorrectSum()

{

// Arrange (Set up the test data and objects)

int num1 = 5;

int num2 = 10;

int expected = 15;

// Act (Execute the code under test)

int actual = \_calculator.Add(num1, num2);

// Assert (Verify the outcome)

Assert.AreEqual(expected, actual, "Addition of two positive numbers should be correct.");

}

[Test]

public void Add\_PositiveAndNegativeNumber\_ReturnsCorrectSum()

{

// Arrange

int num1 = 5;

int num2 = -10;

int expected = -5;

// Act

int actual = \_calculator.Add(num1, num2);

// Assert

Assert.AreEqual(expected, actual, "Addition of positive and negative numbers should be correct.");

}

[Test]

public void Add\_TwoNegativeNumbers\_ReturnsCorrectSum()

{

// Arrange

int num1 = -5;

int num2 = -10;

int expected = -15;

// Act

int actual = \_calculator.Add(num1, num2);

// Assert

Assert.AreEqual(expected, actual, "Addition of two negative numbers should be correct.");

}

[Test]

public void Add\_ZeroToNumber\_ReturnsNumber()

{

// Arrange

int num1 = 7;

int num2 = 0;

int expected = 7;

// Act

int actual = \_calculator.Add(num1, num2);

// Assert

Assert.AreEqual(expected, actual, "Adding zero to a number should return the number itself.");

}

[TearDown] // Method decorated with [TearDown] is executed after each test method

public void Teardown()

{

// Clean up resources if necessary. In this simple case, it's not strictly needed.

\_calculator = null; // Dereference the object

}

[Test]

[Ignore("This test is temporarily ignored for demonstration purposes.")] // Ignores this test

public void ThisTestIsIgnored()

{

Assert.Fail("This test should not run.");

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.

Exercise 1:Write Testable code with Moq

// CustomerComm.Tests.csproj (Simplified project file content - no changes needed here)

// Make sure your project references are correct.

// <Project Sdk="Microsoft.NET.Sdk">

//   <PropertyGroup>

//     <TargetFramework>net8.0</TargetFramework>

//     <ImplicitUsings>enable</ImplicitUsings>

//     <Nullable>enable</Nullable>

//     <IsPackable>false</IsPackable>

//     <IsTestProject>true</IsTestProject>

//   </PropertyGroup>

//

//   <ItemGroup>

//     <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.9.0" />

//     <PackageReference Include="Moq" Version="4.20.70" />

//     <PackageReference Include="NUnit" Version="4.1.0" />

//     <PackageReference Include="NUnit.Analyzers" Version="4.1.0">

//       <PrivateAssets>all</PrivateAssets>

//       <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>

//     </PackageReference>

//     <PackageReference Include="NUnit3TestAdapter" Version="4.5.0" />

//   </ItemGroup>

//

//   <ItemGroup>

//     <ProjectReference Include="..\CustomerCommLib\CustomerCommLib.csproj" />

//   </ItemGroup>

// </Project>

// CustomerCommTests.cs

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

    [TestFixture]

    public class CustomerCommTests

    {

        private Mock<IMailSender>? \_mockMailSender;

        private CustomerCommLib.CustomerComm? \_customerComm;

        /// <summary>

        /// This method runs \*before each test\* in the fixture.

        /// FIX: Changed from [OneTimeSetUp] to [SetUp] to ensure each test gets a fresh mock and SUT instance.

        /// This prevents shared state issues, especially with mock invocation counts.

        /// </summary>

        [SetUp] // Changed from [OneTimeSetUp]

        public void Setup()

        {

            // 1. Initialize a NEW mock object for IMailSender for EACH test.

            \_mockMailSender = new Mock<IMailSender>();

            // 2. Configure the mock's default behavior for THIS test.

            \_mockMailSender.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))

                           .Returns(true);

            // 3. Instantiate a NEW CustomerComm class for EACH test, injecting the fresh mock.

            \_customerComm = new CustomerCommLib.CustomerComm(\_mockMailSender.Object);

        }

        /// <summary>

        /// Test method to verify that SendMailToCustomer returns true when mail sending is successful.

        /// </summary>

        [Test]

        public void SendMailToCustomer\_WhenMailSentSuccessfully\_ReturnsTrue()

        {

            // Act: Call the method under test.

            // Since Setup runs before this test, \_customerComm will be fresh.

            bool result = \_customerComm!.SendMailToCustomer();

            // Assert: Verify the outcome.

            Assert.That(result, Is.True, "SendMailToCustomer should return true when mail is sent successfully.");

            // Verify that SendMail was called exactly once on this test's fresh mock.

            \_mockMailSender!.Verify(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()), Times.Once());

        }

        /// <summary>

        /// Test case to demonstrate what happens if the mock returns false.

        /// This simulates a failure in the mail sending process.

        /// </summary>

        [Test]

        public void SendMailToCustomer\_WhenMailSendingFails\_ReturnsFalse()

        {

            // Arrange: Reconfigure the mock specifically for this test case.

            // This setup only applies to the \_mockMailSender instance created for THIS test.

            \_mockMailSender!.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))

                           .Returns(false);

            // Act: Call the method under test.

            bool result = \_customerComm!.SendMailToCustomer();

            // Assert: Verify the outcome.

            Assert.That(result, Is.False, "SendMailToCustomer should return false when mail sending fails.");

            // Verify that SendMail was called exactly once on this test's fresh mock.

            // (Note: `Times.AtLeastOnce()` is also valid here, but `Times.Once()` is more precise for this method's logic)

            \_mockMailSender!.Verify(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()), Times.Once());

            // No need to reset the mock's behavior here, as a new mock will be created for the next test.

        }

    }

}

// CustomerComm.cs

using System; // Required for ArgumentNullException

namespace CustomerCommLib

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender ?? throw new ArgumentNullException(nameof(mailSender), "IMailSender cannot be null.");

}

public bool SendMailToCustomer()

{

// Actual logic to determine the recipient and message would go here.

// For this example, we use hardcoded values.

string customerEmail = "cust123@abc.com";

string messageContent = "Some important message regarding your recent transaction.";

// Delegate the mail sending to the injected dependency.

// This call will be intercepted by Moq during unit testing.

bool isMailSent = \_mailSender.SendMail(customerEmail, messageContent);

Console.WriteLine($"Attempted to send mail to customer. Result: {isMailSent}");

return isMailSent;

}

}

}

// CustomerComm.Tests.csproj (Simplified project file content - no changes needed here)

// Make sure your project references are correct.

// <Project Sdk="Microsoft.NET.Sdk">

// <PropertyGroup>

// <TargetFramework>net8.0</TargetFramework>

// <ImplicitUsings>enable</ImplicitUsings>

// <Nullable>enable</Nullable>

// <IsPackable>false</IsPackable>

// <IsTestProject>true</IsTestProject>

// </PropertyGroup>

//

// <ItemGroup>

// <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.9.0" />

// <PackageReference Include="Moq" Version="4.20.70" />

// <PackageReference Include="NUnit" Version="4.1.0" />

// <PackageReference Include="NUnit.Analyzers" Version="4.1.0">

// <PrivateAssets>all</PrivateAssets>

// <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>

// </PackageReference>

// <PackageReference Include="NUnit3TestAdapter" Version="4.5.0" />

// </ItemGroup>

//

// <ItemGroup>

// <ProjectReference Include="..\CustomerCommLib\CustomerCommLib.csproj" />

// </ItemGroup>

// </Project>

// CustomerCommTests.cs

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender>? \_mockMailSender;

private CustomerCommLib.CustomerComm? \_customerComm;

/// <summary>

/// This method runs \*before each test\* in the fixture.

/// FIX: Changed from [OneTimeSetUp] to [SetUp] to ensure each test gets a fresh mock and SUT instance.

/// This prevents shared state issues, especially with mock invocation counts.

/// </summary>

[SetUp] // Changed from [OneTimeSetUp]

public void Setup()

{

// 1. Initialize a NEW mock object for IMailSender for EACH test.

\_mockMailSender = new Mock<IMailSender>();

// 2. Configure the mock's default behavior for THIS test.

\_mockMailSender.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))

.Returns(true);

// 3. Instantiate a NEW CustomerComm class for EACH test, injecting the fresh mock.

\_customerComm = new CustomerCommLib.CustomerComm(\_mockMailSender.Object);

}

/// <summary>

/// Test method to verify that SendMailToCustomer returns true when mail sending is successful.

/// </summary>

[Test]

public void SendMailToCustomer\_WhenMailSentSuccessfully\_ReturnsTrue()

{

// Act: Call the method under test.

// Since Setup runs before this test, \_customerComm will be fresh.

bool result = \_customerComm!.SendMailToCustomer();

// Assert: Verify the outcome.

Assert.That(result, Is.True, "SendMailToCustomer should return true when mail is sent successfully.");

// Verify that SendMail was called exactly once on this test's fresh mock.

\_mockMailSender!.Verify(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()), Times.Once());

}

/// <summary>

/// Test case to demonstrate what happens if the mock returns false.

/// This simulates a failure in the mail sending process.

/// </summary>

[Test]

public void SendMailToCustomer\_WhenMailSendingFails\_ReturnsFalse()

{

// Arrange: Reconfigure the mock specifically for this test case.

// This setup only applies to the \_mockMailSender instance created for THIS test.

\_mockMailSender!.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))

.Returns(false);

// Act: Call the method under test.

bool result = \_customerComm!.SendMailToCustomer();

// Assert: Verify the outcome.

Assert.That(result, Is.False, "SendMailToCustomer should return false when mail sending fails.");

// Verify that SendMail was called exactly once on this test's fresh mock.

// (Note: `Times.AtLeastOnce()` is also valid here, but `Times.Once()` is more precise for this method's logic)

\_mockMailSender!.Verify(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()), Times.Once());

// No need to reset the mock's behavior here, as a new mock will be created for the next test.

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.