**PROG2070 – Assignment 2 – Unit Testing**

**Created By:**

Aleksandr Ainidinov (8905450)

Judith Oparaocha (?)

Harsh Maheshbhai Patel (?)

**Date:**

2025-02-06

**Repository Link:** <https://github.com/AleksandrAinidinov/ECommerceApp/tree/master>

**A screenshot of a computer

Description automatically generatedUnit Tests Screenshot:**

**Git Log Screenshot:**

A screenshot of a computer program

Description automatically generated

**Product class code:**

namespace ECommerceApp

{

public class Product

{

public int ProdID { get; set; }

// Error is thrown if the product name is null or empty

private string \_prodName;

public string ProdName {

get { return \_prodName; }

set

{

if (string.IsNullOrEmpty(value))

{

throw new ArgumentNullException("Product name can't be NULL or EMPTY!");

}

\_prodName = value;

}

}

public decimal ItemPrice { get; set; }

public int StockAmount { get; set; }

// Constructor

public Product(int prodID, string prodName, decimal itemPrice, int stockAmount)

{

ProdID = prodID;

ProdName = prodName;

ItemPrice = itemPrice;

StockAmount = stockAmount;

}

// Method to Increase stock

public void IncreaseStock(int amount)

{

if (amount > 0)

StockAmount += amount;

}

// Method to Decrease stock

public void DecreaseStock(int amount)

{

if (amount > 0 && StockAmount - amount >= 0)

StockAmount -= amount;

}

}

}

**Unit Test class code:**

namespace ECommerceApp.nUnitTests

{

[TestFixture]

public class ProductTests

{

/// <summary>

/// Test that uses a valid within range value for ProdID

/// </summary>

[Test]

public void ProdId\_ShouldAssignCorrectly\_WhenWithinRange()

{

// Arrange values

Random rnd = new Random();

int prodId = rnd.Next(6, 5000);

string prodName = "ValidTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ProdID, Is.EqualTo(prodId));;

Assert.That(newProduct.ProdID, Is.LessThanOrEqualTo(4999));

Assert.That(newProduct.ProdID, Is.GreaterThanOrEqualTo(6));

}

/// <summary>

/// Test that uses the maximum allowed ProdID value - "50000"

/// </summary>

[Test]

public void ProdId\_ShouldAssignEdgeMaxValue\_WhenMaxValue()

{

// Arrange values

int prodId = 50000;

string prodName = "MaxTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ProdID, Is.EqualTo(prodId));

}

/// <summary>

/// Test that uses less than minimum allowed ProdID value - "less than 5"

/// </summary>

[Test]

public void ProdId\_ShouldNotAssignLessThanMinValue\_WhenLessThanMinValue()

{

// Arrange values

int prodId = 4;

string prodName = "LessThanMinTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ProdID, Is.LessThanOrEqualTo(5));

}

/// <summary>

/// Test that uses a valid within range value for ItemPrice

/// </summary>

[Test]

public void ItemPrice\_ShouldAssignCorrectly\_WhenWithinRange()

{

// Arrange values

Random rnd = new Random();

decimal itemPrice = rnd.Next(6, 5000);

int prodId = 52;

string prodName = "ValidTestProduct";

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ItemPrice, Is.EqualTo(itemPrice));

Assert.That(newProduct.ItemPrice, Is.LessThanOrEqualTo(4999.00m));

Assert.That(newProduct.ItemPrice, Is.GreaterThanOrEqualTo(6.00m));

}

/// <summary>

/// Test that uses the minimum allowed ItemPrice value - "5.00m"

/// </summary>

[Test]

public void ItemPrice\_ShouldAssignEdgeMinValue\_WhenMinValue()

{

// Arrange values

int prodId = 52;

string prodName = "MinTestProduct";

decimal itemPrice = 5.00m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ItemPrice, Is.EqualTo(itemPrice));

}

/// <summary>

/// Test that uses greater than maximum allowed ItemPrice value - "greater than 5000.00m"

/// </summary>

[Test]

public void ItemPrice\_ShouldNotAssignGreaterThanMaxValue\_WhenGreaterThanMaxValue()

{

// Arrange values

int prodId = 52;

string prodName = "GreaterThanMaxTestProduct";

decimal itemPrice = 5001.00m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ItemPrice, Is.GreaterThanOrEqualTo(5000.00m));

}

/// <summary>

/// Test that uses a valid within range value for StockAmount

/// </summary>

[Test]

public void StockAmount\_ShouldAssignCorrectly\_WhenWithinRange()

{

// Arrange values

Random rnd = new Random();

int stockAmount = rnd.Next(6, 500000);

int prodId = 52;

string prodName = "ValidTestProduct";

decimal itemPrice = 10.10m;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.StockAmount, Is.EqualTo(stockAmount));

Assert.That(newProduct.StockAmount, Is.LessThanOrEqualTo(499999));

Assert.That(newProduct.StockAmount, Is.GreaterThanOrEqualTo(6));

}

/// <summary>

/// Test that uses the maximum allowed StockAmount value - "500000"

/// </summary>

[Test]

public void StockAmount\_ShouldAssignEdgeMaxValue\_WhenMaxValue()

{

// Arrange values

int prodId = 52;

string prodName = "MaxTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 500000;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.StockAmount, Is.EqualTo(stockAmount));

}

/// <summary>

/// Test that uses less than minimum allowed StockAmount value - "less than 5"

/// </summary>

[Test]

public void StockAmount\_ShouldNotAssignLessThanMinValue\_WhenLessThanMinValue()

{

// Arrange values

int prodId = 52;

string prodName = "LessThanMinTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 4;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.StockAmount, Is.LessThanOrEqualTo(5));

}

/// <summary>

/// Test that uses a valid within range value for ProdName

/// </summary>

[Test]

public void ProdName\_ShouldAssignCorrectly\_WhenWithinRange()

{

// Arrange values

int prodId = 52;

string prodName = "ValidTestProduct";

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ProdName, Is.EqualTo(prodName));

}

/// <summary>

/// Test that uses the minimum allowed ProdName value length - "1 symbol"

/// </summary>

[Test]

public void ProdName\_ShouldAssignEdgeMinValue\_WhenMinLength()

{

// Arrange values

int prodId = 52;

string prodName = "f";

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - create a Product instance

Product newProduct = new Product(prodId, prodName, itemPrice, stockAmount);

// Assert - check the result

Assert.That(newProduct.ProdName, Is.EqualTo(prodName));

}

/// <summary>

/// Test that uses empty ProdName value and expects an error to be thrown

/// </summary>

[Test]

public void ProdName\_ShouldThrowErrorIfEmptyValue\_WhenEmptyValue()

{

// Arrange values

int prodId = 52;

string prodName = string.Empty;

decimal itemPrice = 10.10m;

int stockAmount = 10;

// Act - try to create a Product instance and check if the exception is thrown

ArgumentException ex = null;

try

{

new Product(prodId, prodName, itemPrice, stockAmount);

}

catch (ArgumentException e)

{

ex = e;

}

// Assert - check the result exception message

Assert.That(ex.Message, Does.Contain("Product name can't be NULL or EMPTY!"));

}

/// <summary>

/// Test that increases the StockAmount by a positive value

/// </summary>

[Test]

public void IncreaseStock\_ShouldIncreaseStock\_WhenPositiveValue()

{

// Arrange - create a Product instance

Product product = new Product(101, "Phone", 500.00m, 50);

// Act - increase stock by a positive value

product.IncreaseStock(20);

// Assert - check the stock amount changes (it should)

Assert.That(product.StockAmount, Is.EqualTo(70));

}

/// <summary>

/// Test that does not change the StockAmount when a negative value is used

/// </summary>

[Test]

public void IncreaseStock\_ShouldNotChangeStock\_WhenNegativeValue()

{

// Arrange - create a Product instance

Product product = new Product(102, "Tablet", 300.00m, 30);

// Act - increase stock by a negative value

product.IncreaseStock(-10);

// Assert - check the stock amount changes (it shouldn't)

Assert.That(product.StockAmount, Is.EqualTo(30));

}

/// <summary>

/// Test that does not change the StockAmount when "0" is used

/// </summary>

[Test]

public void IncreaseStock\_ShouldNotChangeStock\_WhenZeroValue()

{

// Arrange - create a Product instance

Product product = new Product(103, "Monitor", 200.00m, 20);

// Act - increase stock by a "0"

product.IncreaseStock(0);

// Assert - check the stock amount changes (it shouldn't)

Assert.That(product.StockAmount, Is.EqualTo(20));

}

/// <summary>

/// Test that decreases the StockAmount by a positive value

/// </summary>

[Test]

public void DecreaseStock\_ShouldDecreaseStock\_WhenPositiveValue()

{

// Arrange - create a Product instance

Product product = new Product(104, "Mouse", 50.00m, 40);

// Act - decrease stock

product.DecreaseStock(10);

// Assert - check the stock amount changes (it should)

Assert.That(product.StockAmount, Is.EqualTo(30));

}

/// <summary>

/// Test that does not change the StockAmount when a negative value is used

/// </summary>

[Test]

public void DecreaseStock\_ShouldNotChangeStock\_WhenNegativeValue()

{

// Arrange - create a Product instance

Product product = new Product(105, "Keyboard", 100.00m, 25);

// Act - decrease stock by a negative value

product.DecreaseStock(-5);

// Assert - check the stock amount changes (it shouldn't)

Assert.That(product.StockAmount, Is.EqualTo(25));

}

/// <summary>

/// Test that does not change the StockAmount when "0" is used

/// </summary>

[Test]

public void DecreaseStock\_ShouldNotChangeStock\_WhenZero()

{

// Arrange - create a Product instance

Product product = new Product(106, "Speaker", 150.00m, 60);

// Act - decrease stock by a "0"

product.DecreaseStock(0);

// Assert - check the stock amount changes (it shouldn't)

Assert.That(product.StockAmount, Is.EqualTo(60));

}

}

}