Занятие 4. Unit-тестирование

Цель занятия

1. Получить практический опыт написания юнит-тестов.

Немного теории

Для тестирования кода в .NET применяются специальные тестовые проекты.

Данные проекты используют специальные фреймворки, которые позволяют написать тесты, которые будут запускаться автоматически.

После написания тестов в тестовых проектах, их можно запускать вручную или автоматизировать запуск.

Выбор объектов для тестирования

На данном занятии предлагается рассмотреть тестирование следующих классов:

- `PedalEngine`
- `Car`
- `CarService`

Подготовка проекта

В качестве фреймворка для тестирования используется `xUnit`.

Для создания проекта с поддержкой `xUnit` необходимо выполнить следующие шаги:

1. Открыть терминал и перейти в директорию с решением.
2. Выполнить команду `dotnet new xunit -n UniversalCarShop.Tests`.
3. Выполнить команду `dotnet sln add UniversalCarShop.Tests/UniversalCarShop.Tests.csproj`.

После выполнения этих шагов в директории решения появится новый проект с поддержкой `xUnit`, в котором уже будет добавлен файл с тестом `UnitTest1.cs`. Данный файл можно удалить, так как он не будет использоваться.

Чтобы мы могли тестировать классы из основного проекта, необходимо добавить ссылку на него в тестовый проект.

Для этого в терминале необходимо:

1. Перейти в директорию с тестовым проектом.
2. Выполнить команду `dotnet add reference ../UniversalCarShop/UniversalCarShop.csproj`.

Здесь вместо `../UniversalCarShop/UniversalCarShop.csproj` необходимо указать путь к основному проекту.

Тестирование класса `PedalEngine`

Данный класс реализует интерфейс `IEngine` и содержит:

- поле `Size` для хранения размера педалей
- метод `IsCompatible` для проверки совместимости покупателя с двигателем
- метод `ToString` для получения строкового представления двигателя

Начнем с тестирования свойства `Size`. Данное свойство является доступным только для чтения и должно быть инициализировано при создании объекта.

Проверим, что оно возвращает значение, которое было передано при создании объекта.

Для этого в тестовом проекте добавим новый файл `PedalEngineTests.cs` и напишем следующий код:

```
using Xunit;
using UniversalCarShop;

namespace UniversalCarShop.Tests;

public class PedalEngineTests
{
    [Fact]
    public void Size_Read_ShouldReturnExpectedValue()
    {
        // Arrange
        var engine = new PedalEngine(10);

        // Act & Assert
        Assert.Equal(10, engine.Size);
    }
}
```

В данном коде мы видим несколько элементов, на которые стоит обратить внимание:

- `[Fact]` - это атрибут, который обозначает, что данный метод является тестом.
- `Size_Read_ShouldReturnExpectedValue` - это название теста. Оно должно быть уникальным и описывать, что именно проверяет данный тест.
- `Arrange` - это секция, в которой мы создаем объекты, которые будут использоваться в тесте.
- `Act` - это секция, в которой мы вызываем методы, которые будут тестироваться.
- `Assert` - это секция, в которой мы проверяем результат.
- `Assert.Equal(10, engine.Size);` - это строка, которая проверяет, что свойство `Size` возвращает значение 10.

Правилом хорошего тона является отражение в названии теста того, что именно проверяет данный тест, а также разделение на секции `Arrange`, `Act` и `Assert`.

Попробуем запустить тесты. Для этого в терминале необходимо выполнить команду `dotnet test`.

Если тесты будут пройдены успешно, то в консоли будет выведено соответствующее сообщение.

Далее проверим метод `IsCompatible`. Данный метод должен возвращать `true`, если сила ног покупателя больше 5, и `false` в противном случае.

Для этого в файле `PedalEngineTests.cs` после кода метода `Size_Read_ShouldReturnExpectedValue` добавим следующий код:

```
[Theory]
[InlineData(6, true)]
[InlineData(5, false)]
[InlineData(4, false)]
public void IsCompatible_Call_ShouldReturnExpectedResult(int legPower, bool
expectedResult)
{
    // Arrange
    var engine = new PedalEngine(10);
    var customer = new Customer("Test", legPower);

    // Act & Assert
    Assert.Equal(expectedResult, engine.IsCompatible(customer));
}
```

Здесь мы видим новый атрибут `[Theory]`, который обозначает, что данный метод является параметризованным тестом. Параметризованные тесты позволяют проверять один и тот же тест с разными входными данными.

Для указания входных данных используется атрибут `[InlineData]`, который принимает набор значений, которые будут использоваться для проверки теста.

В данном случае мы проверяем, что метод `IsCompatible` возвращает `true`, если сила ног покупателя больше 5, и `false`, если сила ног покупателя меньше или равна 5.

Попробуем запустить тесты. Для этого в терминале необходимо выполнить команду `dotnet test`.

Если тесты будут пройдены успешно, то в консоли будет выведено соответствующее сообщение.

Тестирование класса `Car`

Данный класс содержит:

- Свойство `Number` для хранения номера автомобиля
- Поле `_engine` для хранения двигателя автомобиля
- Метод `IsCompatible` для проверки совместимости покупателя с автомобилем

Проверку свойства `Number` можно реализовать аналогично тому, как это было сделано для свойства `Size` в классе `PedalEngine`.

Проверим метод `IsCompatible`. Данный метод должен возвращать `true`, если двигатель автомобиля совместим с покупателем, и `false` в противном случае.

Для этого необходимо затронуть понятие моков. Мок — это объект, который имитирует поведение другого объекта.

В данном случае мы можем создать мок двигателя, который будет возвращать `true` или `false` в зависимости от того, какой результат нам нужен.

Для создания моков в .NET может использоваться библиотека `NSubstitute`.

Для установки данной библиотеки в терминале необходимо:

1. Перейти в директорию с тестовым проектом.
2. Выполнить команду `dotnet add package NSubstitute`.

После добавления библиотеки в проект, создадим файл для тестирования класса `Car` и назовем его `CarTests.cs`.

В данном файле напишем следующий код:

```
using Xunit;
using UniversalCarShop;
using NSubstitute;

namespace UniversalCarShop.Tests;

public class CarTests
{
    [Theory]
    [InlineData(true)]
    [InlineData(false)]
    public void IsCompatible_Call_ShouldReturnExpectedResult(bool isCompatible)
    {
        // Arrange
        var engine = Substitute.For<IEngine>();
        engine.IsCompatible(Arg.Any<Customer>()).Returns(isCompatible);

        var car = new Car(engine, 1);
        var customer = new Customer("Test", legPower: 0, handPower: 0);

        // Act & Assert
        Assert.Equal(isCompatible, car.IsCompatible(customer));
    }
}
```

Здесь новой является строка `var engine = Substitute.For<IEngine>();`.

Эта строка создает мок объекта, который реализует интерфейс `IEngine`.

Следующая строка `engine.IsCompatible(Arg.Any<Customer>()).Returns(isCompatible);` определяет поведение метода `IsCompatible` для мока.

Рассмотрим подробнее, что именно происходит в этой строке.

- `engine.IsCompatible(...)` — указывает, что мы хотим определить поведение метода `IsCompatible` на моке.
- `Arg.Any<Customer>()` — указывает, что метод `IsCompatible` может принимать любой объект типа `Customer`.
- `.Returns(isCompatible)` — указывает, что метод `IsCompatible` должен возвращать значение, которое передано в параметре `isCompatible` теста.

Следующая строка `var car = new Car(engine, 1);` создает объект класса `Car`, который использует мок двигателя.

После этого мы проверяем, что метод `IsCompatible` возвращает значение, которое было передано в параметре `isCompatible` теста.

Так как между вызовами теста параметры покупателя не меняются, то таким образом мы проверяем, что метод `IsCompatible` класса `Car` возвращает то же значение, которое возвращается методом `IsCompatible` двигателя.

Тестирование класса `CarService`

Класс `CarService` содержит:

- Поле `_cars` для хранения списка автомобилей
- Метод `AddCar` для добавления автомобиля в список
- Метод `TakeCar` для получения автомобиля из списка

Из этих элементов нам интересны методы `AddCar` и `TakeCar`.

Метод `AddCar` должен добавлять автомобиль в список. При этом номер автомобиля должен увеличиваться на 1 при каждом вызове метода.

Для начала протестируем генерацию номера автомобиля. Для этого создадим файл `CarServiceTests.cs` и напишем следующий код:

```
using Xunit;
using UniversalCarShop;
using NSubstitute;

namespace UniversalCarShop.Tests;

public class CarServiceTests
{
    [Fact]
    public void AddCar_Call_NumberShouldBeUnique()
    {
        // Arrange
        var carService = new CarService();
        var carFactory = Substitute.For<ICarFactory<int>>();
        var carParams = 1;

        // Act
        carService.AddCar(carFactory, carParams);
```

```
        carService.AddCar(carFactory, carParams);

        // Assert
        carFactory.Received(1).CreateCar(carParams, 1);
        carFactory.Received(1).CreateCar(carParams, 2);
    }
}
```

Здесь новый для нас код расположен в секции `Assert`.

При помощи метода `Received` можно проверить, что метод был вызван определенное количество раз с определенными параметрами.

В данном случае мы проверяем, что метод `CreateCar` был вызван дважды с одинаковыми параметрами, но с разными номерами.

Запустим тесты и проверим, что они пройдены успешно.

Теперь проверим, что метод `TakeCar` возвращает добавленные автомобили. Для этого добавим в список два автомобиля и проверим, что метод `TakeCar` вернет их.

```
[Fact]
public void TakeCar_Call_ShouldReturnAddedCars()
{
    // Arrange
    var carService = new CarService();

    var engine = Substitute.For<IEngine>();
    engine.IsCompatible(Arg.Any<Customer>()).Returns(true);

    var car1 = new Car(engine, 1);
    var car2 = new Car(engine, 2);

    var carFactory = Substitute.For<ICarFactory<int>>();
    carFactory.CreateCar(Arg.Any<int>(), Arg.Any<int>()).Returns(car1, car2);

    var carParams = 1;

    // Act
    carService.AddCar(carFactory, carParams);
    carService.AddCar(carFactory, carParams);

    var actualCar1 = carService.TakeCar(new Customer("Test", 0, 0));
    var actualCar2 = carService.TakeCar(new Customer("Test", 0, 0));

    // Assert
    Assert.Equal(car1, actualCar1);
    Assert.Equal(car2, actualCar2);
}
```

Здесь мы получили довольно комплексный тест, в котором происходит следующее:

1. Для начала создается мок двигателя, который будет совместим с любым покупателем.
2. Создаются два автомобиля с разными номерами.
3. Создается мок фабрики автомобилей, которая возвращает ранее созданные автомобили.
4. При помощи фабрики создаются два автомобиля и добавляются в список.
5. Вызывается метод `TakeCar` для получения автомобилей из списка.
6. Проверяется, что автомобили, которые вернул метод `TakeCar`, совпадают с автомобилями, которые были добавлены в список.

Запустим тесты и проверим, что они пройдены успешно.

Самостоятельная работа

1. Проверьте, что метод `IsCompatible` класса `PedalEngine` не учитывает силу рук покупателя.
2. Проверьте, что свойство `Number` класса `Car` содержит значение, которое было передано при создании объекта.
3. Проверьте, что метод `GetCustomers` класса `CustomersStorage` возвращает добавленных покупателей.
4. Проверьте, что метод `SellCars` класса `HseCarService` продает автомобили добавленным покупателям.
5. Проверьте, что метод `SellCars` класса `HseCarService` не продает один и тот же автомобиль двум покупателям.