Занятие 4. Unit-тестирование

Теоретическая часть

Unit-тестирование (unit testing) — это подход к тестированию программного обеспечения, при котором отдельные части программы (обычно методы или классы) проверяются изолированно от остальной системы. Цель unit-тестов — убедиться, что каждый компонент работает правильно сам по себе.

Зачем нужны unit-тесты:

- Позволяют быстро находить ошибки на ранних этапах разработки.
- Обеспечивают уверенность при рефакторинге и доработках — если тесты проходят, значит изменения не сломали существующую логику.
- Упрощают сопровождение и развитие кода.

Обычно для написания unit-тестов используют специальные фреймворки, такие как xUnit, NUnit или MSTest для C#. Эти фреймворки позволяют удобно описывать тестовые методы, автоматически запускать их и проверять результаты.

Для изоляции тестируемого кода от внешних зависимостей (например, базы данных, сетевых сервисов) часто применяют технику мокирования (mocking). С помощью мок-объектов можно подменять реальные зависимости на "заглушки", которые ведут себя предсказуемо и позволяют тестировать только нужную логику.

Для рассмотрения Unit-тестирования в .NET воспользуемся примером из занятия 2 с применением принципов SOLID.

Практическая часть

Создание проекта

Начнем с создания проекта. Сделаем это следующими командами:

```
dotnet new solution -n 04-unit-testing
dotnet sln migrate
rm 04-unit-testing.sln
dotnet new console -n AppForTesting
dotnet sln add ./AppForTesting/AppForTesting.csproj
```

Далее скопируем код из примера с использованием принципов SOLID из занятия 2, а именно:

- `Models/Report.cs`
- `Services/IReportSaver.cs`
- `Services/Implementations/TextReportSaver.cs`
- `Services/IReportSender.cs`
- `Services/Implementations/EmailReportSender.cs`
- `Services/ReportService.cs`

- `Program.cs`

И изменим корневое пространство имен на `AppForTesting`.

Усложнение модели отчета

Чтобы нам было интереснее тестировать отчеты, давайте усложним модель отчета, а именно добавим проверки:

- Количество проданных автомобилей должно быть больше или равно 0;
- Количество проданных мотоциклов должно быть больше или равно 0;
- Название отчета должно быть не пустым.

После внесения изменений наш код должен выглядеть следующим образом:

```
using System.Text;

namespace AppForTesting.Models;

public sealed record Report
{
    public Report(string title, DateOnly date, TimeOnly time, int carsSold, int
motorcyclesSold)
    {
        Title = string.IsNullOrEmpty(title)
            ? throw new ArgumentException("Title cannot be null or empty")
            : title;
        Date = date;
        Time = time;
        CarsSold = carsSold < 0
            ? throw new ArgumentException("CarsSold cannot be less than 0")
            : carsSold;
        MotorcyclesSold = motorcyclesSold < 0
            ? throw new ArgumentException("MotorcyclesSold cannot be less than 0")
            : motorcyclesSold;
    }

    public string Title { get; }
    public DateOnly Date { get; }
    public TimeOnly Time { get; }
    public int CarsSold { get; }
    public int MotorcyclesSold { get; }

    public override string ToString()
    {
        var builder = new StringBuilder();

        builder
            .AppendLine(Title)
            .AppendLine($"Дата: {Date:dd.MM.yyyy}")
            .AppendLine($"Время: {Time:HH:mm:ss}")
            .AppendLine("--------------------------------")
```

```
            .AppendLine($"Продано автомобилей: {CarsSold} шт.")
            .AppendLine($"Продано мотоциклов: {MotorcyclesSold} шт.")
            .AppendLine("--------------------------------");

        return builder.ToString();
    }
}
```

Как можно видеть из кода, мы заменили primary-конструктор на обычный конструктор, в котором добавили проверки на пустоту названия отчета и на отрицательное значение количества проданных автомобилей и мотоциклов.

Для компактности записи проверки мы использовали синтаксис тернарного оператора - он работает по логике "если условие истинно, то вернуть первое значение, иначе вернуть второе".

Чтобы код скомпилировался, нам также нужно обновить вызов конструктора в `Program.cs`:

```
using AppForTesting.Models;
using AppForTesting.Services;
using AppForTesting.Services.Implementations;

var textReportSaver = new TextReportSaver("report.txt");
var emailReportSender = new EmailReportSender("example@example.com");

var reportService = new ReportService(textReportSaver, emailReportSender);

var report = new Report(
    title: "Отчет",
    date: DateOnly.FromDateTime(DateTime.Now),
    time: TimeOnly.FromDateTime(DateTime.Now),
    carsSold: 100,
    motorcyclesSold: 50
);

reportService.ProcessReport(report);
```

Проверим, что приложение запускается и работает так же, как и прежде:

```
dotnet run --project ./AppForTesting/AppForTesting.csproj
```

Создание проекта тестов

Теперь, когда у нас есть что тестировать, давайте добавим Unit-тесты. Начать стоит с создания проекта тестов - для этого выполним следующие команды:

```
dotnet new xunit -n AppForTesting.Tests
dotnet add ./AppForTesting.Tests/AppForTesting.Tests.csproj reference
```

```
./AppForTesting/AppForTesting.csproj
dotnet sln add ./AppForTesting.Tests/AppForTesting.Tests.csproj
```

Первая команда создает проект тестов с использованием фреймворка xUnit.

Вторая команда добавляет зависимость от основного проекта, так как без нее мы не сможем использовать классы основного проекта в коде тестов.

Третья команда добавляет пакет Shouldly, который позволяет использовать более читаемый синтаксис для проверки результатов тестов.

Четвертая команда добавляет проект тестов в решение.

Тесты без параметров

Начнем с простейшего теста - теста без параметров. Такие тесты могут быть использованы либо когда результат не зависит от параметров, либо когда хочется проверить всего один success-case без corner-cases.

В xUnit для создания теста без параметров используется атрибут `[Fact]`. Этот атрибут указывает, что метод является тестовым и должен быть запущен тестовым фреймворком.

В нашем случае в таком тесте можно проверить, правильно ли работает конструктор класса `Report`, а именно правильно ли он копирует параметры в свойства класса.

Для создания теста создадим файл `ReportTests.cs` в корне тестового проекта и добавим следующий код:

```
using AppForTesting.Models;
using Xunit;

namespace AppForTesting.Tests;

public class ReportTests
{
    [Fact]
    public void Ctor_Call_PropertiesHaveExpectedValues()
    {
        // Arrange
        var expectedTitle = "Отчет";
        var expectedDate = DateOnly.FromDateTime(DateTime.Now);
        var expectedTime = TimeOnly.FromDateTime(DateTime.Now);
        var expectedCarsSold = 100;
        var expectedMotorcyclesSold = 50;

        // Act
        var report = new Report(
            title: expectedTitle,
            date: expectedDate,
            time: expectedTime,
            carsSold: expectedCarsSold,
```

```
            motorcyclesSold: expectedMotorcyclesSold
        );

        // Assert
        Assert.Equal(expectedTitle, report.Title);
        Assert.Equal(expectedDate, report.Date);
        Assert.Equal(expectedTime, report.Time);
        Assert.Equal(expectedCarsSold, report.CarsSold);
        Assert.Equal(expectedMotorcyclesSold, report.MotorcyclesSold);
    }
}
```

Разберем различные части кода.

```
[Fact]
```

Атрибут `[Fact]` указывает, что метод является тестовым и должен быть запущен тестовым фреймворком.

```
public void Ctor_Call_PropertiesHaveExpectedValues()
```

Название теста. Каждый тест - это, по сути, метод, который проверяет другие методы. Хорошей практикой считается формирование названия теста в формате: `<Тестируемый элемент>_<Действие>_<Ожидаемый результат>`.

```
// Arrange
var expectedTitle = "Отчет";
var expectedDate = DateOnly.FromDateTime(DateTime.Now);
var expectedTime = TimeOnly.FromDateTime(DateTime.Now);
var expectedCarsSold = 100;
var expectedMotorcyclesSold = 50;
```

Раздел кода, в котором мы определяем входные данные для теста. Обычно называется `Arrange` и помечается соответствующим комментарием.

```
// Act
var report = new Report(
    title: expectedTitle,
    date: expectedDate,
    time: expectedTime,
    carsSold: expectedCarsSold,
    motorcyclesSold: expectedMotorcyclesSold
);
```

Раздел кода, в котором мы вызываем тестируемый метод. Обычно называется `Act` и помечается соответствующим комментарием.

```
// Assert
Assert.Equal(expectedTitle, report.Title);
Assert.Equal(expectedDate, report.Date);
Assert.Equal(expectedTime, report.Time);
Assert.Equal(expectedCarsSold, report.CarsSold);
Assert.Equal(expectedMotorcyclesSold, report.MotorcyclesSold);
```

Раздел кода, в котором мы проверяем результат. Обычно называется `Assert` и помечается соответствующим комментарием. В данном случае мы используем метод `Assert.Equal`, который проверяет, что два значения равны.

Теперь запустим тест командой:

```
dotnet test
```

В выводе консоли мы должны увидеть, что тест прошел успешно.

Тесты с параметрами

Тесты с параметрами могут быть использованы когда результат зависит от параметров. Либо когда мы хотим за раз проверить несколько вариантов входных данных и убедиться, что поведение класса не зависит от них. В xUnit для создания теста с параметрами используется атрибут `[Theory]`. Он носит такой же смысл, что и `[Fact]`, но позволяет передавать параметры в тест.

Так как до этого мы добавили проверки в конструктор класса `Report`, то мы можем проверить, что они работают правильно.

Добавим в класс `ReportTests` новый тест с параметрами:

```
[Theory]
[InlineData(0)]
[InlineData(10)]
public void Ctor_CallWithValidCarsSold_ShouldNotThrow(int carsSold)
{
    // Act & Assert
    var _ = new Report(
        title: "Отчет",
        date: DateOnly.FromDateTime(DateTime.Now),
        time: TimeOnly.FromDateTime(DateTime.Now),
        carsSold: carsSold,
        motorcyclesSold: 50
    );
}
```

Разберем различные части кода.

```
[Theory]
```

Атрибут `[Theory]` помечает метод как параметризованный тест.

```
[InlineData(0)]
[InlineData(10)]
```

Атрибут `[InlineData]` позволяет определить набор параметров для теста. В данном случае наш метод принимает один параметр - поэтому в атрибут мы передаем единственный аргумент, но также данный атрибут поддерживает и передачу нескольких аргументов для тестов с несколькими параметрами.

Как можно увидеть, данный тест не использует вспомогательный класс `Assert`, так как в нем мы ходим проверить, что исключение не будет выброшено. Соответственно, если вызов конструктора выбросит исключение, то тест будет провален. Из-за этого нам не нужно дополнительно использовать класс `Assert`.

Запустим тест командой:

```
dotnet test
```

Далее проверим кейсы с отрицательными значениями количества проданных автомобилей, но для передачи параметров воспользуемся другим приемом.

Для начала добавим ниже кода класса `ReportTests` новый класс `TestData`:

```
file static class TestData
{
    public static TheoryData<int> InvalidCarsSoldData => new() {
        {-1},
        {-10}
    };
}
```

Здесь мы видим уровень видимости `file`, добавленный в последних версиях C# - он позволяет объевить элемент кода, видимый только в текущем файле.

Внутри класса `TestData` мы объявили статическое свойство `InvalidCarsSoldData`, которое возвращает набор параметров для теста.

Теперь воспользуемся этим классом для написания нового теста. Добавим в класс `ReportTests` новый метод:

```
[Theory]
[MemberData(nameof(TestData.InvalidCarsSoldData), MemberType = typeof(TestData))]
public void Ctor_CallWithInvalidCarsSold_ShouldThrow(int carsSold)
{
    // Act & Assert
    var _ = Assert.Throws<ArgumentException>(() => new Report(
        title: "Отчет",
        date: DateOnly.FromDateTime(DateTime.Now),
        time: TimeOnly.FromDateTime(DateTime.Now),
        carsSold: carsSold,
        motorcyclesSold: 50
    ));
}
```

В данном коде новым является атрибут `[MemberData]`. Он позволяет указать свойство или метод, которые возвращают аргументы для теста. Это позволяет использовать в тесте не только статически описанные данные, но и генерируемые динамически.

Также в данном коде мы используем класс `Assert.Throws`, который позволяет проверить, что метод выбросит исключение определенного типа.

Рассмотрим пример с динамически генерируемыми данными. Для этого добавим в класс `TestData` новый метод:

```
public static TheoryData<int> GetValuesInRange(int min, int max)
{
    var data = new TheoryData<int>();

    for (var i = min; i <= max; i++)
    {
        data.Add(i);
    }

    return data;
}
```

Как видим, это просто код, который в цикле добавляет значения в набор параметров. Логика генерации может быть абсолютно любой - это лишь пример.

Теперь воспользуемся этим методом для написания нового теста. Добавим в класс `ReportTests` новый метод:

```
[Theory]
[MemberData(nameof(TestData.GetValuesInRange), 0, 2, MemberType =
typeof(TestData))]
public void Ctor_CallWithValidMotorcyclesSold_ShouldNotThrow(int motorcyclesSold)
{
    // Act & Assert
```

```
    var _ = new Report(
        title: "Отчет",
        date: DateOnly.FromDateTime(DateTime.Now),
        time: TimeOnly.FromDateTime(DateTime.Now),
        carsSold: 100,
        motorcyclesSold: motorcyclesSold
    );
}
```

Как видим, для передачи дополнительных аргументов в метод-генератор мы просто перечисляем эти аргументы после названия метода.

Теперь проверим, что тесты проходят успешно:

```
dotnet test
```

Тесты с мокированием

Выше мы посмотрели на тесты, которые проверяют относительно простые в проверке вещи, вроде равенства значений или выброса исключений. Но бывают случаи, когда мы хотим проверить, как наш код интегрируется с другими компонентами. Например, хотим проверить, что сервис сохраняет значение в базу данных при помощи репозитория.

Такую проверку можно осуществить двумя способами:

- Интеграционное тестирование - вид тестирования, при котором мы запускаем приложение целиком и проверяем, как оно работает с реальной инфраструктурой;
- Unit-тесты с использованием мокирования - это те же тесты, которые мы рассмотрели выше, но в них мы можем проверить, был ли в процессе выполнения кода вызван тот или иной метод.

В качестве примера рассмотрим проверку того, что при вызове метода `ReportService.ProcessReport` происходит сохранение отчета.

Для начала добавим тестовый проект новую зависимость:

```
dotnet add ./AppForTesting.Tests/AppForTesting.Tests.csproj package NSubstitute
```

В данном случае мы добавляем зависимость от библиотеки NSubstitute, которая позволяет создавать мок-объекты. Мок-объекты позволяют имитировать поведение интерфейсов и классов, а также проверять, был ли имитирующий код вызван в рамках теста.

После добавления зависимости добавим в корень тестового проекта файл `ReportServiceTests.cs` и добавим в него следующий код:

```
using AppForTesting.Services;
using AppForTesting.Models;
```

```
using NSubstitute;

namespace AppForTesting.Tests;

public class ReportServiceTests
{
    [Fact]
    public void ProcessReport_Call_ShouldSaveReport()
    {
        // Arrange
        var reportSaver = Substitute.For<IReportSaver>();
        var reportSender = Substitute.For<IReportSender>();
        var reportService = new ReportService(reportSaver, reportSender);

        var report = new Report(
            title: "Отчет",
            date: DateOnly.FromDateTime(DateTime.Now),
            time: TimeOnly.FromDateTime(DateTime.Now),
            carsSold: 100,
            motorcyclesSold: 50
        );

        // Act
        reportService.ProcessReport(report);

        // Assert
        reportSaver.Received(1).SaveReport(report);
    }
}
```

Рассмотрим новые части кода.

```
var reportSaver = Substitute.For<IReportSaver>();
var reportSender = Substitute.For<IReportSender>();
var reportService = new ReportService(reportSaver, reportSender);
```

В данном случае мы создаем мок-объекты для интерфейсов `IReportSaver` и `IReportSender`, после чего создаем экземпляр класса `ReportService` и передаем в него эти мок-объекты.

```
reportService.ProcessReport(report);
```

В данном случае мы вызываем метод `ProcessReport` класса `ReportService` и передаем в него созданный ранее отчет. Мы хотим проверить, что при вызове этого метода был вызван метод `SaveReport` интерфейса `IReportSaver`.

```
reportSaver.Received(1).SaveReport(report);
```

Здесь мы проверяем, что метод `SaveReport` был вызван 1 раз и с передачей в него созданного ранее отчета.

Запустим тест командой:

```
dotnet test
```

NSubstitute обладает богытм API, который позволяет по-разному настроить мок-объекты и проверить различные сценарии. Более подробно можно почитать в официальной документации.

Тестирование внутренних компонентов

Периодически возникают ситуации, когда мы хотим протестировать не публично-доступный код, а внутренние компоненты сборки. Например, давайте протестируем, что при вызове метода `SaveReport` класса `TextReportSaver` происходит сохранение отчета в файл.

Для этого добавим в корень основного проекта новый файл `AssemblyAttributes.cs` и добавим в него следующий код:

```
[assembly: InternalsVisibleTo("AppForTesting.Tests")]
```

После добавления этого атрибута мы можем использовать внутренние компоненты сборки `AppForTesting` вне данной сборки, но только в указанном тестовом проекте.

Теперь проверим, что механизм работает. Добавим в корень тестового проекта файл `TextReportSaverTests.cs` и добавим в него следующий код:

```
using AppForTesting.Services.Implementations;
using AppForTesting.Models;

namespace AppForTesting.Tests;

public class TextReportSaverTests
{
    [Fact]
    public void SaveReport_Call_ShouldSaveReport()
    {
        // Arrange
        var textReportSaver = new TextReportSaver("report.txt");

        var report = new Report(
            title: "Отчет",
            date: DateOnly.FromDateTime(DateTime.Now),
            time: TimeOnly.FromDateTime(DateTime.Now),
            carsSold: 100,
            motorcyclesSold: 50
        );
```

```
        // Act
        textReportSaver.SaveReport(report);

        // Assert
        Assert.True(File.Exists("report.txt"));
    }
}
```

Из совершенно нового в данном коде мы видим использование метода `Assert.True`, который проверяет, что условие истинно. В данном случае мы проверяем, что файл `report.txt` существует.

Запустим тест командой и убедимся, что тестирование внутренних компонентов работает.

```
dotnet test
```

Итог

В данном занятии мы познакомились с основами Unit-тестирования в C#. Мы рассмотрели различные способы написания тестов, а также различные подходы к тестированию. Для закрепления материала рекомендуется самостоятельно написать тесты, которые:

- Проверяют, что мы не можем создать отчет с пустым названием;
- Проверяют, что при обработке отчета происходит его отправка.