# Day 05. Тестирование

## Общие требования

- Убедитесь, что на вашем компьютере установлен SDK для разработки на .NET 5 и вы используете именно его.
- Помните, ваш код будут читать! Обратите особое внимание на оформление вашего кода и именование переменных. Обязательно придерживайтесь общепринятых стандартов C# Coding Conventions.
- Самостоятельно выберите удобную для себя IDE.
- Программа должна иметь возможность запуска через командную строку dotnet.
- В каждом из заданий указаны примеры ввода и вывода. Решение должно использовать их как верный формат.
- В начале каждого задания приведен список рекомендованных языковых конструкций.
- Если затрудняетесь в решении задачи, обратитесь с вопросами к другим участникам бассейна, интернету, Google, посмотрите на StackOverflow.
- С основными возможностями языка C# можно ознакомиться в официальной спецификации.
- Избегайте **хардкода** и “**магических чисел**”.
- Вы демонстрируете все решение, верный результат работы программы – лишь один из способов проверки ее корректной работы. Поэтому когда необходимо получить определенный вывод в результате работы ваших программ, запрещено показывать пред рассчитанный результат.
- Обращайте особое внимание на термины, выделенные **bold** шрифтом: их лучше сразу погуглить, их изучение пригодится вам как в выполнении текущего задания, так и в вашей дальнейшей карьере .NET разработчика.
- Have fun :)

## Требования к заданиям дня

- Каждому из заданий должно соответствовать отдельное решение, точкой входа в которое является консольное приложение, созданное на основе стандартного шаблона .NET SDK.
- Используйте **var**.
- Название решения (и его отдельного каталога) должно выглядеть как d{*xx*}, где *xx* - цифры текущего дня. Названия проектов указаны в задании.
- Для форматирования выходных данных используйте **культуру** en-GB: N2 для вывода денежных сумм, d для дат.

# Интро

Программист - профессия, где всегда нужно учиться. Новые инструменты, технологии, методологии, во всем этом приходится быстро разбираться. Вы уже заметили, что с ростом опыта, решение некоторых задач дается проще и быстрее, качественнее.

Единственное, что занимает много времени, это отладка. Каждый раз проверять сценарию работы ПО по одному не удобно и очень расточительно. И вот вы, вдохновленные Кентом Беком и его книгой Test Driven Development: By Example, решаете помочь OpenSource сообществу и написать тесты к библиотеке markdown-generator, которую планируете использовать на следующем проекте.

Библиотека эта позволяет генерировать документацию к коду в формате **Markdown**, которую в дальнейшем можно разместить на GitHub Wiki. Пример вы можете увидеть здесь.

Репозиторий необходимо склонировать к себе на компьютер, и изменить origin на тот репозиторий, который вы создали для данного задания.

```
$ git remote rm origin

$ git remote add origin your/repository/project.git

$ git config master.remote origin

$ git config master.merge refs/heads/master
```

## Виды тестов

На разных уровнях программного обеспечения используются различные виды тестов. Например **unit-тесты** используются для тестирования методов и функций, которые являются малой частью целого. В тоже время **интеграционные тесты** проверяют взаимодействие модулей системы. Немного больше о видах тестов вы можете узнать здесь.

## Методологии

Кроме видов тестирования, существуют различные к нему подходы.

В данном уроке мы будем писать **BDD** тесты - тесты, основанные на текущем поведении системы. Помимо этого существую подход, который называется **TDD** - разработка через тестирование. О плюсах и минусах подходов вы можете прочитать здесь.

## Что тестировать?

Стоит понимать, что тесты - это тоже часть кодовой базы, которую нужно поддерживать. На поддержку, актуализацию этой кодовой базы также тратятся ваши силы, часы и деньги работодателя. Поэтому следует задаться вопросом, а какая цель тестирования?

Есть интересная статья на данную тему. Как и в статье, в нашем случае мы будем защищать себя от **регрессии** проекта.

## Best practices

На сайте Microsoft собрана краткая информация обо всех вышеперечисленных пунктах, также там есть раздел о правильном именовании тестов. Обязательно к изучению.

## Фреймворки

Конечно, тесты не пишутся с нуля. Для их разработки существует несколько основных фреймворков. Для unit-тестов это **XUnit**, **NUnit** и **MsTest**. Подробнее о каждом из них вы можете прочитать здесь.

Для данного проекта мы выберем XUnit, так как он активно поддерживается community и вбирает в себя лучшее, что есть в остальных фреймворках. Посмотрите, как запускать unit-тесты в IDE, в которой вы разрабатываете проекты.

## Структура проекта

Создайте в решении с библиотекой папку *Tests*, в папке *Tests* создайте проект *Markdown.Generator.Core.Tests* по **шаблону Unit Test Project** (название зависит от используемой IDE). Укажите в качестве используемого фреймворка XUnit . Все тесты данных заданий необходимо размещать в данном проекте. Название каждого задания является названием файла с тестами.

```
Markdown.Generator/
        Tests/
                Markdown.Generator.Core.Tests/
                        ElementsTests.cs
                        MarkdownBuilderTests.cs
                        MarkdownableTypeTests.cs
                        GithubWikiDocumentBuilderTests
        Markdown.Generator.Core/
        Markdown.Generator.Application/
```

## Напутствие

Библиотека markdown-generator действительно содержит некоторое количество багов. В рамках написания тестов вы встретитесь с ним, и их необходимо будет исправить.

Для названия тестов существуют различные конвенции, но мы будем использовать характерную для BDD:
Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior.

# Задание 00. ElementsTests

В библиотеке markdown-generator конечный документ в формате Markdown формируется с помощью небольших строительных блоков, таких как заголовки

(*Header*), ссылки (*Link*), таблицы (*Table*) и другие. Данные элементы инкапсулирует в себе логику форматирования переданных параметров в элемент разметки Markdown.

Элементов много - при реализации легко ошибиться в синтаксисе, и готовый документ будет содержать ошибку. Только сразу вы этого не заметите, потому что документация может быть довольно объемной.

Поэтому для начала мы напишем тесты для всех элементов из пространства имен *Markdown.Generator.Core.Elements*, которые будут проверять, что разметка соответствует разметке Markdown.

Разберем на примере элемента *Code*. Этот элемент предназначен для отображения исходного кода, включая форматирование и подсветку синтаксиса. В Markdown этот элемент выглядит следующим образом:

```
```csharp
some code
```
```

Как мы видим, этот элемент состоит из последовательности символов ```, названия языка программирования, непосредственно кода для отображения и завершается той же последовательностью символов ```.

На языке C# данная строка с учетом переноса строк будет выглядеть как:
```csharp\nsome code\n```\n

Создайте тест в файле *ElementsTests*.

Следуя конвенции именования тестов название теста будет следующим:
Given_Code_When_LanguageAndCodeAsParameter_Then_ReturnMarkdownCodeMarkup.

Далее, возьмем строку ```csharp\nsome code\n```\n как эталон и поместим ее значение в переменную *expected*. Затем создайте объект типа *Code* с параметрами `csharp и some code` и вызовите метод *Create*. Результат сохраните в переменную *actual*. Теперь для проверки значения воспользуйтесь статическим классом **Assert** и сравните два значения. Если значения не равны, тест выполнится не успешно.

Действия, которые описаны выше, это паттерн построения unit-тестов, который называется **AAA - Arrange, Act, Assert**. В блоке **Arrange** настраивается окружения unit-теста (создание объекта *Code*, объявление *expected*). В блоке **Act** происходит выполнение тестируемого сценария. И в блоке **Assert** происходит проверка результатов. Следуйте данному паттерну, и ваши тесты будут чистыми, их будет легко читать.

Напишите тесты элементов *CodeQuote, Header, List, Link, Image*. Примеры возвращаемых значений вы можете взять из документации к библиотеке и использовать их в качестве expected значений при написании тестов.
```

Больше информации о разметке вы можете найти в Markdown-Cheatsheet.

# Задание 01. MarkdownBuilderTests

Рассмотрим класс *MarkdownBuilder*. Этот класс реализует паттерн строитель для построения markdown-разметки документа из элементов разметки.

Для добавления элементов в документ реализованы соответствующие методы.
Так как элементов много, легко ошибиться в реализации, например в методе *CodeQuote* создавать элемент *Code*.

*MarkdownBuilder* позволяет просмотреть список добавленных элементов через свойство *Elements*.

Необходимо реализовать unit-тесты, которые проверяют, что после вызова методов *CodeQuote, Code, Link, Header* класса *MarkdownBuilder* в коллекцию *Elements* попадают элементы одноименных типов. Для успешного прохождения тестов, необходимо проверить, что в коллекцию *Elements* добавляется только один элемент, который имеет тип *CodeQuote, Code, Link, Header* (в зависимости от того, для какого элемента написан тест). В этом поможет статический класс **Assert** и его методы *Equals* и *Contains*

# Задание 02. MarkdownableTypeTests

Класс *MarkdownableType* является оберткой над **System.Type**, которая реализует методы для удобного доступа к информации о членах класса, таких как **FieldInfo, MethodInfo, PropertyInfo** и других. Подробнее можно ознакомиться здесь.

Помимо этого, класс переопределяет метод **ToString** - при его вызове возвращается информация о типе в виде строки markdown.

В классе присутствует метод *GetMethods*, который возвращает массив MethodInfo - информацию о методах для документации. Но есть проблема, в документации не должны указываться приватные методы, так как они являются частью внутреннего api программного обеспечения и часто меняются. Эти изменения не должны затрагивать пользователей ПО. Но данный метод их возвращает.

Прежде чем искать ошибку, необходимо написать тест. Для этого необходим объект, у которого нам заранее известен список методов публичных и приватных. Этот объект будет называться **stub**, он будет имитировать заданное состояние.

Для данного теста создайте класс *Sut* в одном файле с *MarkdownableTypeTests*

```
public class Sut
{
   public void PublicMethod(){ }
```

```
    private void PrivateMethod(){ }
}
```

Напишите тест, который создает объект *MarkdownableType* для типа *Sut,* и вызовите метод *GetMethods*. Проверьте, что в результате содержится только публичный метод *PublicMethod*, например по названию, или используя **MethodInfo.IsPrivate**.

Тест завершится с ошибкой. Теперь отредактируйте код метода *GetMethods* так, чтобы тест выполнился успешно.

Дополнительно напишите аналогичные тесты на методы *GetFields* и *GetProperties*.

# Задание 03. GithubWikiDocumentBuilderTests

В проекте есть класс *GithubWikiDocumentBuilder*, который отвечает за генерацию файлов документации для последующего размещения на Github Wiki. Класс **обобщенный**, и принимает в конструктор объект, реализующий интерфейс *IMarkdownGenerator*.

При написании unit-тестов для класса *GithubWikiDocumentBuilder* мы сталкиваемся с проблемой - чтобы изолировать тесты для *GithubWikiDocumentBuilder* от реализации *IMarkdownGenerator* и не зависеть от его реализации и побочных эффектов, нам необходимо имитировать поведение *IMarkdownGenerator*. Для этого, как упоминалось ранее, мы можем использовать **stub**-объект, например создать реализацию данного интерфейса, которая возвращает заранее заготовленные объекты.

Но если нам нужно проверить различные сценарии работы, такие как исключения, позитивные сценарии, негативные сценарии, это может быть неудобно - создавать **stub**-объекты для каждого из тестов.

**Stub**-имитирует состояние объекта. Но для имитации различного поведения, нам понадобятся объекты другой категории - **Mock**-объекты.

**Mock**-и позволяют имитировать поведение объекта, возвращать результат функций в виде констант или набора случайных значений. Помимо предоставления результатов mock позволяет проверить вызывалась ли функция, с какими параметрами и сколько раз.

В .NET есть отличная библиотека для создания mock-объектов - Moq. Как гласит документация к библиотеке, нам достаточно создать объект класса **Mock** и указать в качестве generic-типа интерфейс который мы хотим замокать. Затем используя методы **Setup** и **Returns** настроить, какие именно методы мы хотим подменить, и какие результаты они должны возвращать.

Используя библиотеку *Moq* реализуйте следующие тесты.

Рефлексия может существенно повлиять на скорость работы приложения. Почему мы об этом задумались? Потому что методы *Load* в классе, реализующем интерфейс *IMarkdownGenerator,* активно используют рефлексию для нахождения сбора информации о типах.

Необходимо, с помощью пакета Moq проверить, что перегрузки методов *Load* вызываются только один раз для генерации одного набора документации (т.е за один вызов метода *Generate*). Да поможет вам **Verify** и **Times.**

Так как нам доступно несколько перегрузок метода *Load*, при этом все перегрузки используются в классе *GithubWikiDocumentBuilder*, необходимо проверить, что вызываются корректные перегрузки с правильными параметрами.