Тестування ПЗ

«Pay attention to zeros. If there is a zero, someone will divide by it.»

Dr. Cem Kaner

Тестування ПЗ

Тестування – це процес перевірки поведінки програмного забезпечення на відповідність заданим вимогам

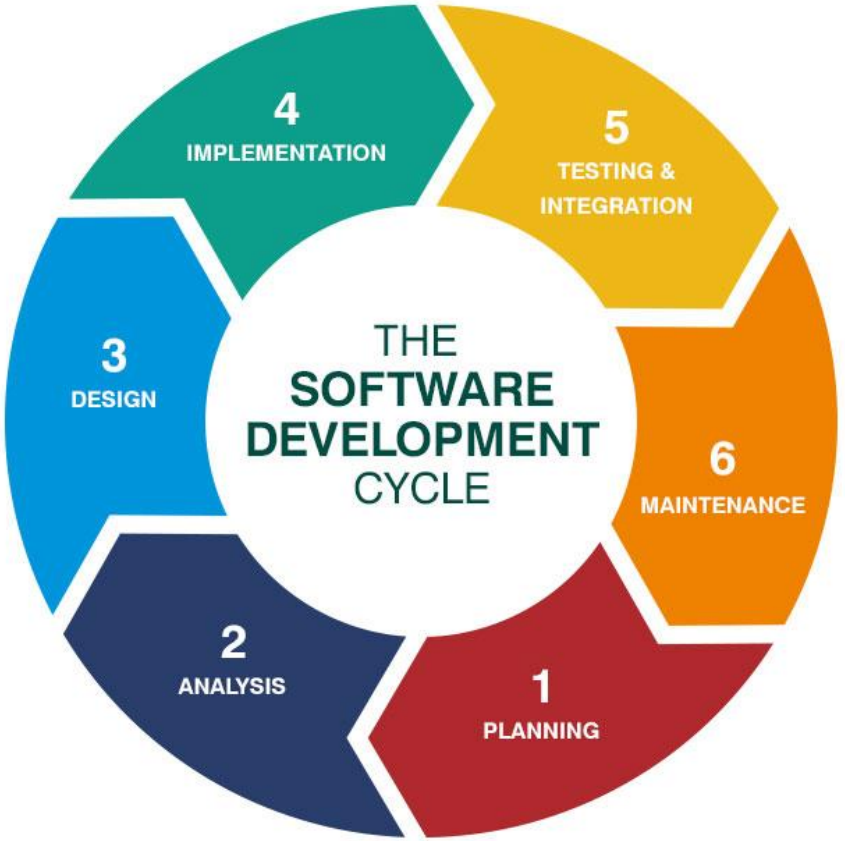
Процес тестування включає:

- Планування робіт (Test Management)
- Проектування тестів (Test Design)
- Виконання тестування (Test Execution)
- Аналіз результатів (Test Analysis)

Документи:

- Test Plan
- Test Case
- Test Case Specification
- Bug Report

Тестування в SDLC



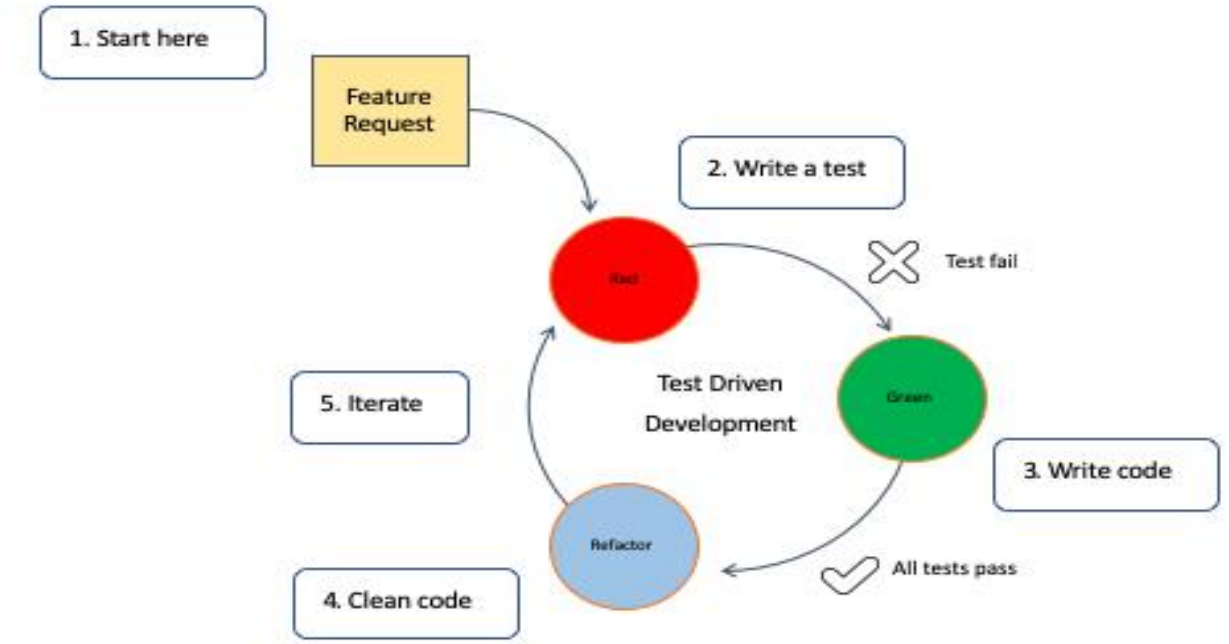
Test-Driven Development (TDD)

1. Не варто писати жодного коду, доки не написано відмовного тесту.

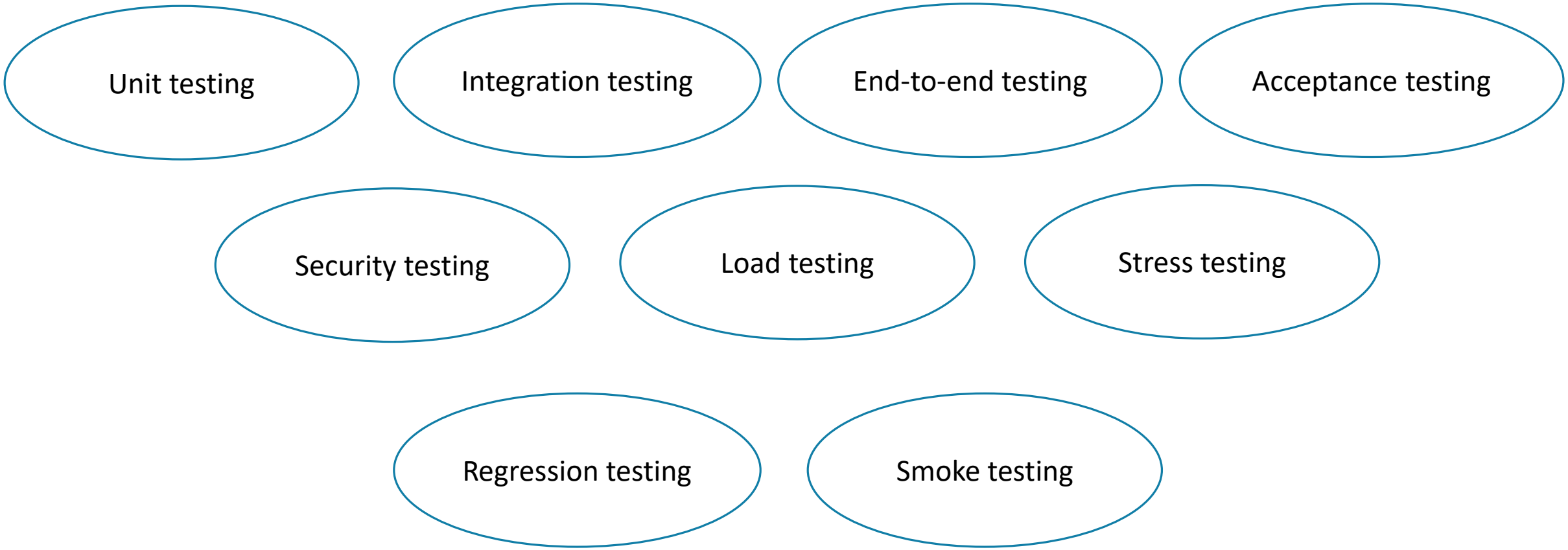
2. Не варто писати більше тесту ніж необхідно для його відмови. Помилка компіляції - це також відмова.

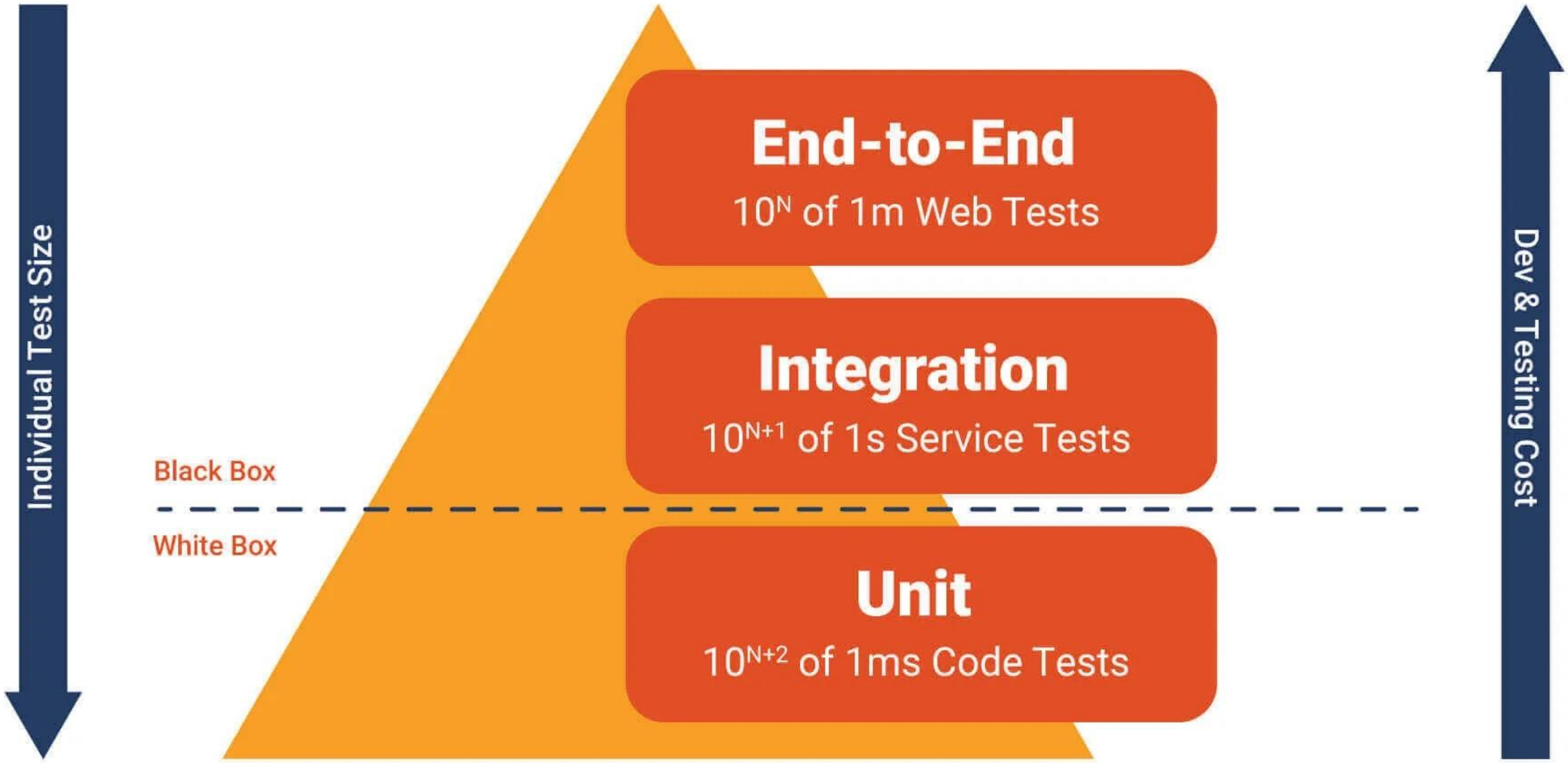
3. Не варто писати більше коду, ніж необхідно для проходження поточного відмовного тесту.

Red-Green factor



Види тестів


Unit testing
Integration testing
End-to-end testing
Acceptance testing
Security testing
Load testing
Stress testing
Regression testing
Smoke testing


Individual Test Size
End-to-End
10^N of 1m Web Tests
Integration
10^N+1 of 1s Service Tests
Black Box
White Box
Unit
10^N+2 of 1ms Code Tests
Dev & Testing Cost

Unit тестування

Переваги unit тестів:

- Швидко виконуються
- Запобігають повторному перетестовуванню
- Документують код
- Забезпечують малу зв’язність компонентів

Характеристики якісного unit тесту

- **Fast**
- **Isolated**
- **Repeatable**
- **Self-validating**
- **Thorough**

Термінологія unit тестування

- **Тестовий метод** – метод, що реалізовує тест.
- **Тестовий клас** – клас, який містить тестові методи.
- **Контекст тексту –** набір даних, що необхідні для виконання тесту.
- **Stub об’єкт –** керована заміна залежності класу, який тестують.
- **Mock об’єкт –** stub об’єкт, за допомогою якого вирішують чи правильно виконався тест.
- **Fake об’єкт** – загальний термін, який використовується для опису як stub об’єкта, так і mock об’єкта.

Фреймворки для unit тестування у .NET

xUnit.net

NUnit

MSTest

MSTest

- Розроблений Microsoft
- Інтегрований у Visual Studio
- Версія 2 має відкритий вихідний код
- https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest
- https://github.com/microsoft/testfx/blob/main/docs/README.md
- https://learn.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2022

NUnit 

- Безкоштовний фреймворк з відкритим вихідним кодом для написання unit тестів у .NET.
- Є частиною xUnit сімества, виконує ту ж саму роль, що й SUnit для Smalltalk та JUnit для java
- https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit
- https://docs.nunit.org/
- https://docs.nunit.org/articles/nunit/intro.html

xUnit.net

- Безкоштовний фреймворк з відкритим вихідним кодом для написання unit тестів у .NET.
- Створений автором NUnit v2 Джеймсом Нюкірком
- Є найбільш сучасним інтрументом для unit тестування у .NET
- https://xunit.net/#documentation
- https://xunit.net/docs/why-did-we-build-xunit-1.0

Порівняння фреймворків

	MSTest	NUnit	xUnit
Кількість завантажень на день (nuget.org)	73 тис.	54 тис.	68 тис.
Кількість тегів (stack overflow)	3223	7300	4000

Criteria	NUnit	xUnit.net	MSTest
Setting Up Tests	[SetUp]	Not available(Use constructor)	[TestInitialize]
TearDown Tests	[TearDown]	Not Available(Use Dispose)	[TestCleanup]
Marking method as Test	[Test]	[Fact]/[Theory]	[TestMethod]/ [DataTestMethod]
One-Time Set up for All Tests in One Class	[OneTimeSetUp]	Not Available(Use IClassFixture<T>)	[ClassInitialize]
One-Time Teardown for All Tests in One Class	[OneTimeTearDown]	Not Available(Use IClassFixture<T>)	[ClassCleanup]
Collection Fixture Setup and Teardown	Not available	ICollectionFixture<T>	Not Available
Ignore or Skip tests	[Ignore(“reason”)]	[Fact(Skip=”reason”)]	[Ignore]
Specify the Test category	[Category()]	[Trait(“Category”, “”)	[TestCategory()]
Assembly level one-time setup	Not available	Not available	[AssemblyInitialize]
Assembly level one time clean up	Not available	Not available	[AssemblyCleanUp]

Архітектура xUnit.net

Виконання unit тестів у xUnit.net відбувається за допомогою двох компонетів:

- **runner** – програма, що здійснює пошук тестових збірок і активує знадені тестові фреймворки. Активація фреймворків відбувається за допомогою runner utility library – xunit.runner.utility

- **test framework (тестовий фреймворк)** – відповідає за виявлення та запуск unit тестів (xunit.core.dll, xunit.execution.dll).

Комунікація між runner і test framework здійснюється за допомогою абстракцій у збірці xunit.abstractions.dll

Створення тесту в xUnit.net

- Для позначення тестового методу використовуються атрибути **[Fact]** і **[Theory]**
- Атрибут **[Fact]** позначає тест, який є інваріантим стосовно вхідних параметрів
- Атрибут **[Theory]** позначає тест, який виконується для певного набору параметрів
- Атрибут **[Theory]** використовується разом з атрибутами **[InlineData]**, **[ClassData]** та **[MemberData]**

Заміна залежностей класу

- Unit тести мають перевіряти роботу компонента в ізоляції від інших.
- Для забезпечення ізольованого тестування компонента передбачені додаткові бібліотеки, які дають змогу замінити залежності на fake об’єкти, які повністю контролюються розробником тестів.
- Moq https://github.com/moq/moq4
- Fake It Easy https://fakeiteasy.github.io/

Спільний контекст для тестів в xUnit.net

Виділення спільного коду для ініціалізації і завершення контексту тестів в xUnit.net відбувається за допомогою:

- конструктора і методу Dispose тестового класу;
- class fixtures
- collection fixtures
- https://xunit.net/docs/shared-context.html

Конструктор і метод Dispose тестового класу для керування контекстом

- Конструктор і метод Dispose тестового класу використовують тоді, коли код ініціалізації і видалення контексту є спільним для всіх тестів у межах тестового класу.

- Для кожного тесту ініціалізація і видалення контексту відбувається заново.

Class fixtures для керування контекстом

- **Class fixture** – це клас, в якому визначений контекст та конструктор і метод Dispose для ініціалізації і видалення контексту відповідно.

- Тестовому класу необхідно реалізувати інтерфейс **IClassFixture<TFixture>** та визначити поле типу **TFixture**, де **TFixture** – class fixture

- При використанні class fixture контекст ініціалізується лише один раз перед виконанням всіх тестів у класі та, відповідно, видаляється після їх завершення.

Collection fixtures для керування контекстом

- **Collection fixture** – це клас, який реалізовує один або декілька інтерфейсів **ICollectionFixture<TFixture>**, де **TFixture** – class fixture, та анотований атрибутом **[CollectionDefinition(<string>)]**, де <string> назва collection fixture

- Тестовий клас має бути анотований **[Collection(<string>)]**, де <string> назва collection fixture

- При використанні collection fixture контекст ініціалізується лише один раз перед виконанням усіх тестів у всіх класах, що анотовані однаковим [Collection] атрибутом та, відповідно, видаляється після їх завершення.

Виконання тестів

- За замовчуванням для кожного тесту створюється новий екземпляр тестового класу.
- Тести в межах одного класу за замовчуванням групуються в одну колекцію.
- Тести із різних класів виконуються в окремих потоках.
- Для того, щоб тести із різних класів не виконувались паралельно, необхідно додати до класів атрибут [Collection(<string>)] із однаковим параметром <string>.
- Для зміни параметрів паралельного виконання див.: https://xunit.net/docs/running-tests-in-parallel.html

Рекомендації до написання unit тестів

1. Уникайте залежностей від інфраструктурний об’єктів, використовуйте mock, stub та принцип інверсії залежностей.
2. Виберіть правильну назву для тесту.
3. Впорядкуйте внутрішню структуру тесту.
4. Реалізовуйте мінімалістичні тести.
5. Уникайте «магічних» даних.
6. Уникайте бізнес-логіки в тестах.
7. Уникайте виклику кількох методів, що тестуються в межах одного тесту.
8. Створюйте класи-обгортки для статичних залежностей.

https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Уникайте залежностей від інфраструктурний компонентів

Клас має тестуватись в ізоляції, усі залежності повинні бути визначені як інтерфейси, що легко можна замінити fake object, згідно з принципом інверсії залежностей.

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

Виберіть правильну назву тесту

Назва тесту має складатись з:

- **назви методу**, який тестують;
- **сценарію**, в контексті якого тестують метод;
- **очікуваної поведінки** в результаті виконання сценарію.

Погано

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Добре

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Впорядкуйте внутрішню структуру

Внутрішня структура тесту має відповідати патерну **Arrange-Act-Assert** :

- **Arrange** – ініціалізує об’єкт класу, який тестують;
- **Act –** виконує метод класу, який тестують;
- **Assert** – перевіряє результат виконання тесту.

Погано

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

Добре

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

Реалізовуйте мінімалістичні тести

Використовуйте якомога простіші вхідні параметри для перевірки тесту.

Погано

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

Добре

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Уникайте «магічних» даних

Винесенння «магічних» даних у константи допомагає краще зрозуміти, що перевіряє тест.

Погано

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

Добре

```
[Fact]
void Add_MaximumSumResult_ThrowsOverflowException()
{
    var stringCalculator = new StringCalculator();
    const string MAXIMUM_RESULT = "1001";

    Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

    Assert.Throws<OverflowException>(actual);
}
```

Уникайте бізнес логіки в тестах

Тест не має копіювати реалізацію методу, а лише перевіряти вихідні дані.

Погано

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }
}
```

Добре

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

Уникайте виклику кількох методів, що тестуються, в межах одного тесту

В рамках патерну **Arrange-Act-Assert**, секція **Act** має виконуватись лише один раз

Погано

```
[Fact]
public void Add_EmptyEntries_ShouldBeTreatedAsZero()
{
    // Act
    var actual1 = stringCalculator.Add("");
    var actual2 = stringCalculator.Add(",");

    // Assert
    Assert.Equal(0, actual1);
    Assert.Equal(0, actual2);
}
```

Добре

```
[Theory]
[InlineData("", 0)]
[InlineData(",", 0)]
public void Add_EmptyEntries_ShouldBeTreatedAsZero(string input, int expected)
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add(input);

    // Assert
    Assert.Equal(expected, actual);
}
```

Створюйте класи обгортки для статичних залежностей

Статичні залежності класу, наприклад, DateTime.Now, не можуть контролюватись виконавцем тесту, тому їх краще винести в окремий клас і додати як залежність класу, що тестується.

Погано

```
public int GetDiscountedPrice(int price)
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Добре

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if (dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Покриття коду тестами

Для визначення покриття коду тестами використовують інструменти двох типів:

- **Data collectors (колектори даних)** – збирають інформацію про виконання тестів.
- **Report generators (генератори звіту)** – використовують інформацію про виконання тестів для генерування звітів.

Coverlet

- Coverlet – популярний колектор даних з відкритим вихідним кодом, є частиною .NET Foundation
- За замовчуванням додається в проекти типу xUnit
- Інтегрований з Visual Studio Test Platform (coverlet.collector) та MSBuild (coverlet.msbuild)
- https://github.com/coverlet-coverage/coverlet

ReportGenerator

- ReportGenerator – генератор звітів з відкритим вихідним кодом
- Перетворює зібрані дані про виконання тестів у звіти в зручних для читання форматах, наприклад html
- https://github.com/danielpalme/ReportGenerator

SonarQube

- SonarQube – аналізатор проектів для різних мов програмування
- Окрім аналізу якості програмного коду, SonarQube дає змогу переглядати покриття коду тестами
- https://www.sonarsource.com/products/sonarqube/

Інтеграційне тестування

Переваги інтеграційних тестів:

- Тестують логіку декількох компонентів, що взаємодіють між собою
- Дають змогу покрити більшу частину коду, яка пов’язана з інфраструктурою програми

Інтеграційне тестування в .NET

- Для виконання інтеграційних тестів у .NET можна також використовувати фремворки xUnit.net, NUnit, MSTest
- Відмінність від unit тестування полягає в тому, що не обов’язково замінювати залежності класу на fake об’єкти
- Також необхідно розгортати тестову базу даних для тестування та запускати тестовий сервер

Інтеграційне тестування в .NET

- Для написання інтеграційних тестів використовують клас **WebApplicationFactory<TProgram>,** де TProgram – клас, що налаштовує та запускає ASP .NET Core програму.

- WebApplicationFactory запускає тестовий сервер, на якому виконується ASP .NET Core програма

- Для додаткового налаштування ASP .NET Core програми в тестовому режимі можна створити новий клас, похідний від WebApplicationFactory та перевизначити метод ConfigureWebHost

Load and stress тестування в .NET

- Azure Load Testing
- Apache Jmeter
- ApacheBench (ab)
- k6
- Locust
- https://learn.microsoft.com/en-us/aspnet/core/test/load-tests?view=aspnetcore-7.0

Висновки

- Тестування є необхідною умовою випуску якісного програмного продукту
- Тести забезпечують гнучкість програмного коду та зручність його супроводу
- Для реалізації тестів повинні застосовуватися усі правила написання «чистого» коду
- Автоматизоване тестування надає можливість швидко розгортати нові версії продукту і є важливою частиною CI/CD процесу