Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів тестування та мокування
програмного забезпечення на платформі .Net
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-5

Богун В.М.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного
забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Голян В.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(шифр і назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Богуну Владиславу Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *«Дослідження методів тестування та мокування програмного забезпечення на платформі .Net»*

Затверджена наказом по університету від 29.03. 2024р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 21.06.2024

3. Вихідні дані до роботи: *методи створення механізмів тестування та мокування (Mocking). В результаті буде знайдено оптимальний засіб тестування відносно певних умов проекту та оточення, також буде проаналізовано фреймворк для максимально ефективного мокування коду.*

4. Перелік питань, що потрібно опрацювати в роботі *вступ, аналіз предметної галузі, аналіз існуючих підходів та методів, знаходження комплексного та ефективного методу, дослідження можливостей бібліотек, експерементальне мокування статики, висновки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд літератури, аналіз проблеми та постановка задачі дослідження	22.01.2024 – 12.02.2024	виконано
2	Дослідження методів тестування та мокування програмного забезпечення на платформі .Net	12.02.2024 – 25.03.2024	виконано
3	Програмна реалізація застосунку	25.02.2024 – 18.05.2024	виконано
4	Проведення експериментальних досліджень та аналіз результатів	01.04.2024 – 29.04.2024	виконано
5	Підготовка пояснювальної записки	22.04.2024 – 20.05.2024	виконано
6	Перевірка на плагіат та нормоконтроль	08.06.2024	виконано
7	Підготовка презентації та доповіді	12.06.2024	виконано
8	Рецензування	15.06.2024	виконано
9	Попередній захист	19.06.2024	виконано
10	Занесення диплома в електронний архів	19.06.2024	виконано
11	Допуск до захисту у зав. кафедри	20.06.2024	виконано

Дата видачі завдання 22.01.2024 р.

Студент (ка) _____ Богун В.М.
(підпис)

Керівник роботи _____ доц. Голян В.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка: 58 стор., 8 рис., 12 джерел.

АНАЛІТИКА, ПАТЕРН, ПРОГРАМНА СИСТЕМА, МОКУВАННЯ.

Об’єктом дослідження – методи створення механізмів тестування та мокування (Mocking).

Метою роботи є дослідження методів тестування та мокування програмного забезпечення на платформі .Net

В результаті буде знайдено оптимальний засіб тестування відносно певних умов проекту та оточення, також буде проаналізовано фреймворк для максимально ефективного мокування коду.

ANALYTICS, PATTERN, SOFTWARE SYSTEM, MOCKING.

The object of research is methods of creating testing and mocking mechanisms.

The purpose of the work is to study the methods of testing and mocking software on the .Net platform.

As a result, the optimal testing tool will be found in relation to certain project conditions and environment, and the framework for the most efficient code mocking will be analyzed.

Я, Богун Владислав Миколайович, студент гр. ІПЗм-22-5, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота на тему «Дослідження методів тестування та мокування програмного забезпечення на платформі .Net», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві

відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

У сучасному світі швидкого технологічного прогресу та стрімкого нарощування обсягів даних, ефективність та точність у розробці програмного забезпечення набувають особливої ваги. Розвиток методів тестування та мокування (Mocking) є критичним аспектом, що забезпечує якість та надійність програмних продуктів. Від початкових етапів розвитку інформаційних технологій, роль тестування та валідації програмного коду постійно зростала, адже вони відіграють ключову роль у забезпеченні стабільності та ефективності програмних систем.

Тестування та мокування стали ще важливішими з огляду на складність сучасних програмних систем і мультиплатформеність. Розробники зіткнулися з викликами, пов'язаними з потребою у комплексних та гнучких інструментах, які б дозволяли ефективно моделювати та перевіряти поведінку програм в різноманітних середовищах. Виникає необхідність у розробці методологій та патернів, що сприяють створенню ефективних механізмів тестування та мокування, здатних забезпечити надійність та високу продуктивність програмних рішень.

Зосередження на методах створення механізмів тестування та мокування дозволяє розробникам підвищити якість програмного забезпечення, мінімізувати ризики, пов'язані з помилками в коді, та сприяє підвищенню ефективності розробки. Особливу увагу в цьому контексті заслуговують підходи та інструменти, що дозволяють автоматизувати тестування, надавати розробникам гнучкість у виборі стратегій тестування та водночас забезпечувати глибоке покриття тестами різних аспектів програмного коду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної області тестування ПО

Тестування програмного забезпечення – це процес, що полягає у систематичній перевірці та аналізі програми чи системи з метою забезпечення їхньої коректності, надійності та відповідності вимогам. Воно стало невід'ємною складовою розробки програм і систем завдяки декільком причинам.

Тестування виникло від необхідності виявлення та виправлення помилок у програмах. Помилки завжди були присутні в програмах через людський фактор або незрозуміння вимог. Тестування дозволило виявити ці помилки на ранніх стадіях розробки, що заощадило час і ресурси.

Зростання складності програмних продуктів призвело до потреби в усуненні можливих проблем ще до того, як вони стануть критичними.

Збільшення вимог користувачів до якості програмного забезпечення стало фактором, що підсилює потребу в тестуванні [1]. Користувачі сьогодні очікують, що програми будуть працювати безперебійно та без помилок, і тестування допомагає відповісти на ці вимоги.

Отже, тестування програмного забезпечення вирішує проблеми виявлення помилок, забезпечення відповідності вимогам, підвищення якості та надійності продукту, заощадження коштів на подальшій розробці та підвищення впевненості користувачів у якості програми.\

Механізми тестування програмного забезпечення на платформі .NET відноситься до набору методів, інструментів, підходів та практик, які використовуються для перевірки якості та функціональності програмного забезпечення. Механізми тестування включають у себе наступні елементи:

- інструменти та фреймворки для тестування – це спеціалізоване програмне забезпечення, яке дозволяє автоматизувати процес тестування. Наприклад, NUnit та xUnit.net є популярними фреймворками для написання та виконання юніт-тестів у .NET;

- Mocking та штучні об'єкти – це методика створення імітаційних версій залежностей об'єктів (наприклад, баз даних або веб-сервісів), щоб можна було тестувати частини програми в ізоляції. Moq та NSubstitute - це приклади інструментів для мокування в .NET [2];
- принципи та патерни тестування – це методології та практики для тестування, такі як TDD (Test-Driven Development) або BDD (Behavior-Driven Development), які впливають на спосіб розробки та тестування програм;
- використання скриптів та інструментів для автоматичного виконання тестів, що значно допомагає виявляти помилки на ранніх стадіях розробки;
- інтеграція з системами контролю версій та CI/CD – це інтеграція процесу тестування з системами контролю версій та системами неперервної інтеграції/неперервного розгортання (CI/CD) для забезпечення якості коду і швидкої доставки змін.

1.2 Актуальність проблеми

У процесі дослідження предметної області ми можемо встановити деякі проблеми із підходом до тестування. Вибір фреймворку та підходу до побудови механізмів тестування може бути обумовлений технічними вимогами проекту. Наприклад, платформа, на якій розробляється проект (наприклад, .NET Core або .NET 5+), може вплинути на вибір фреймворку.

Наявність досвіду роботи команди з певним фреймворком також може вплинути на вибір. Команда, яка вже знайома з одним інструментом, може бути більш продуктивною.

Тип та характер проекту можуть вимагати різних підходів до тестування. Наприклад, для одиничного тестування компонентів може підходити один фреймворк, тоді як для інтеграційних тестів інший. Ця проблема актуальна через різницю у специфікаціях проектів.

Фреймворк з активною спільнотою і підтримкою може бути надійнішим вибором. Спільнота забезпечує швидше виявлення та виправлення помилок, а також забезпечує доступ до ресурсів та допомоги.

Можливість інтегрувати фреймворк з іншими інструментами, такими як системи автоматизованої збірки (CI/CD), також важлива для розробників.

Розробники можуть не розуміти всі різновиди тестування, інтеграційне тестування та інше. Це може призвести до невірного вибору фреймворку для конкретного виду тестування.

Саме тому наше дослідження методів створення механізмів тестування та мокування(Mocking) програмного забезпечення, на платформі .Net – є актуальним, адже вирішить вище зазначені труднощі розробників, та закриє багато питань з вибору підходу та стратегії до тестування ПО.

1.3 Постановка задачі

Уході даної роботи нами буде досліджено предметну область побудови тестових механізмів за допомогою фреймворків та підходів до тестування на платформі .Net.

Ми проведемо порівняльний аналіз та аналіз призначення кожного фреймворку та підходу до тестування.

Ця робота спрямована створення дорожньої карти для розробників що до знаходження най-ефективнішого та комплексного підходу до тестування застосунків. Реалізація наукового дослідження складається з наступних етапів:

– аналіз предметної області;

– аналіз методів тестування;

– розглядання особливостей NUnit, xUnit, MSTest;

– порівняння атрибутів методів побудови механізмів тестування;

– аналіз мокування;

– формування висновків.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ

2.1 Метод тестування через NUnit

NUnit – це відкритий фреймворк для юніт-тестування, який спеціалізується на мові програмування C# та платформі .NET від Microsoft [3]. Він є частиною сімейства xUnit фреймворків, які використовуються для модульного тестування в різних мовах програмування. NUnit став одним із найпопулярніших інструментів для юніт-тестування в екосистемі .NET, завдяки своїй гнучкості, легкості використання та потужним можливостям.

Розберемо основні характеристики даного фреймворку:

- використовуються атрибути C#, такі як [Test] та [TestFixture], для позначення тестових класів та методів. Це дозволяє легко ідентифікувати тестові сценарії та організовувати їх;
- NUnit пропонує широкий спектр асертів для перевірки різних умов у тестовому коді. Це включає асерти для порівняння значень, перевірки винятків, перевірки булевих умов та багато іншого;
- підтримуются параметризовані тести, де тестові дані можна подавати безпосередньо в тестовий метод, що дозволяє використовувати однаковий тестовий код для різних сценаріїв;
- фреймворк дозволяє групувати тести в набори для зручності організації та керування;
- легка інтеграція з багатьма середовищами розробки, такими як Microsoft Visual Studio, та підтримує роботу з інструментами неперервної інтеграції;
- сумісність із багатьма версіями .NET, включаючи .NET Framework та .NET Core, що робить його використання можливим у різних проектах.

NUnit широко використовується в практиках TDD, де тестування є інтегральною частиною розробки програмного забезпечення. Розробники спочатку пишуть тести, які визначають вимоги або функціональність, а потім пишуть код, який має пройти ці тести.

NUnit широко використовується в проектах з багатошаровою архітектурою, де кожен шар (наприклад, логіка даних, бізнес-логіка, інтерфейс користувача) потребує окремого тестування. Це дозволяє ізолювати та тестувати кожен компонент окремо, забезпечуючи високу якість загальної системи.

Фреймворк є важливим інструментом при рефакторингу коду. Наявність тестів, написаних з використанням NUnit, дозволяє розробникам вносити зміни в код, не побоюючись ненавмисно порушити існуючу функціональність, оскільки тести відразу ж виявлять будь-які проблеми.

Оскільки NUnit є open source проектом, він часто використовується в спільнотах розробників для спільної розробки та тестування проектів. Його відкритий характер дозволяє легко інтегруватися з іншими інструментами та бібліотеками.

Великим плюсом є те що великі корпорації також використовують NUnit для забезпечення надійності та якості їхнього програмного забезпечення. Відповідно це підтверджує ефективність NUnit у керуванні складними тестовими сценаріями та інтеграція з різними платформами та інструментами роблять його ідеальним рішенням для великомасштабних проектів.

Все це робить NUnit незамінним інструментом у сучасній розробці програмного забезпечення, особливо в контексті .NET-розробки. Його використання дозволяє розробникам забезпечувати більш високий рівень якості коду, знижувати кількість помилок та забезпечувати стабільність функціональності протягом усього циклу розробки.

NUnit складається з декількох основних компонентів, які взаємодіють між собою для надання потужного та гнучкого інструменту для юніт-тестування:

- Test Runner – це "серце" NUnit, яке виконує тести. Test Runner може бути як консольним застосунком, так і графічним інтерфейсом користувача (GUI);
- Test Cases – окремі блоки тестового коду, які представляють конкретні сценарії тестування;

– Test Fixtures – класи, які містять тестові випадки. Вони використовуються для групування тестів, які мають спільні ресурси або логіку налаштування;
– Assertions – методи, які використовуються для перевірки певних умов у тестовому коді;
– Setup and Teardown – методи, що викликаються перед і після кожного тесту або групи тестів, для ініціалізації та очищення необхідних ресурсів;
– Attributes – атрибути, які використовуються для визначення поведінки тестових випадків та фікстур, наприклад [Test], [TestFixture], [SetUp], [TearDown];
– Test Data – дані, які використовуються для параметризації тестів. Це може включати дані для тестів, що вводяться вручну або генеруються динамічно.

Взаємодію цих компонентів можна побачити на рисунку 2.1.

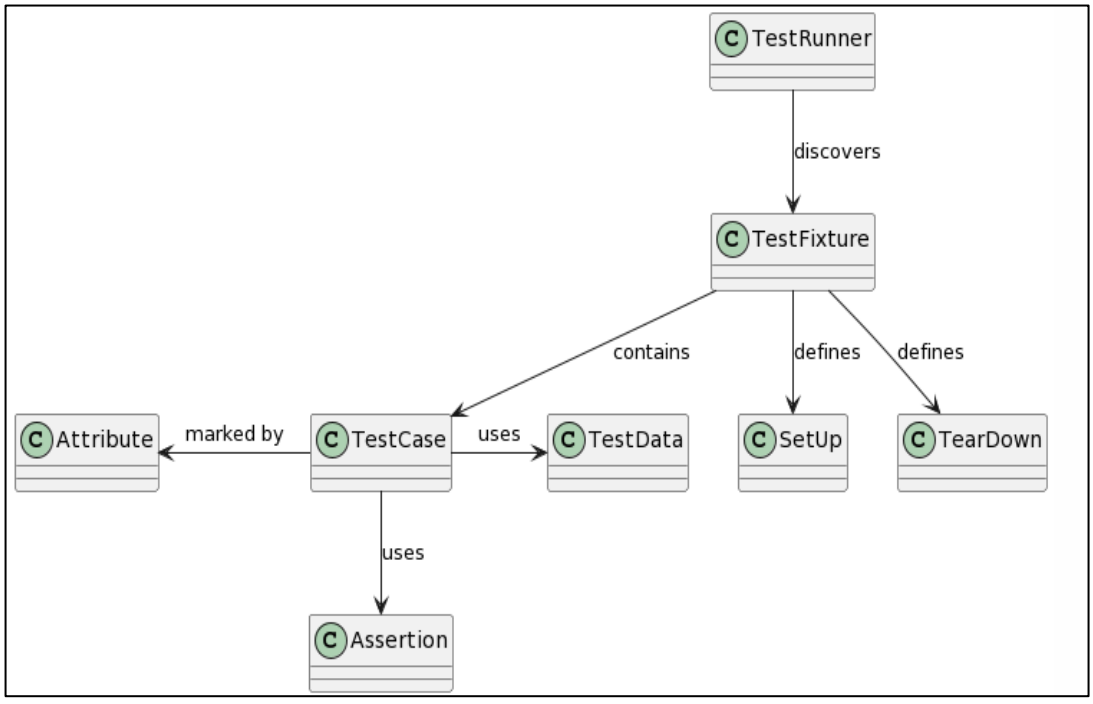

Рисунок 2.1 – Взаємодія структурних елементів NUnit

Розглянемо роботу компонентів:

– Test Runner – в ініціалізує процес тестування, виявляючи та збираючи всі Test Fixtures;

- в межах кожного Test Fixture, SetUp методи запускаються перед кожним тестом для підготовки середовища;
- Test Cases виконуються, використовуючи Assertions для перевірки відповідності фактичних результатів очікуваним;
- після кожного тесту, TearDown методи запускаються для очищення ресурсів;
- Test Runner збирає результати та відображає їх користувачу.

У підсумку, NUnit відіграє важливу роль у розвитку якісного та надійного програмного забезпечення. Він допомагає розробникам швидко виявляти та виправляти помилки, підтримує регулярне оновлення функціональності та забезпечує, що новий або змінений код відповідає очікуванням та вимогам. Саме ці якості роблять NUnit одним із найпопулярніших та найбільш цінних механізмів тестування в області .NET розробки.

2.2 Метод тестування через xUnit

xUnit – це загальна назва для родини фреймворків юніт-тестування, які походять від SUnit, першого фреймворку, розробленого Кентом Беком для мови Smalltalkn [4]. Відтоді концепція xUnit була адаптована для різних мов програмування, у тому числі і для .NET, де вона відома як xUnit.net. Ці фреймворки є стандартом де-факто для юніт-тестування у багатьох мовах програмування і служать основою для розробки тестів, які перевіряють окремі блоки коду незалежно один від одного.

Далі розглянемо деякі характеристики та особливості фреймворку. xUnit використовує атрибути, щоб позначити методи як тестові. Наприклад, атрибут [Fact] використовується для позначення методу, який містить один юніт-тест. Атрибути дозволяють фреймворку автоматично ідентифікувати та виконувати тести. Це спрощує організацію тестового коду та робить його більш читабельним.

Також підтримується параметризація тестів за допомогою атрибутів, таких як [Theory] і [InlineData]. Це дозволяє виконувати один і той же тестовий метод з різними вхідними даними, значно підвищуючи гнучкість та охоплення

тестування. Такий підхід дозволяє розробникам легко додавати нові сценарії тестування без необхідності писати додатковий тестовий код.

У xUnit кожен тестовий метод запускається в своєму екземплярі тестового класу, що допомагає ізолювати тести один від одного. Це унікальна особливість, яка гарантує, що спільний стан не зберігається між тестами, знижуючи ризик помилок, пов'язаних зі спільними даними або станом.

Для покращення сприйняття та аналізу результатів, фреймворк містить велику бібліотеку асертів, які використовуються для перевірки різних умов у тестах. Це включає перевірку рівності, нерівності, винятків та багато іншого. Підтримка перевірки винятків дозволяє тестувати код на правильну обробку помилкових сценаріїв.

Також варто відзначити що система легко інтегрується з багатьма сучасними середовищами розробки та інструментами для неперервної інтеграції. Це робить його зручним для використання в широкому спектрі проектів та забезпечує плавну інтеграцію з різними робочими процесами.

Ці особливості роблять xUnit дуже потужним інструментом для юніт-тестування в екосистемі .NET, дозволяючи розробникам створювати ефективні, гнучкі та легко підтримувані тестові сценарії. Його можливості роблять його ідеальним вибором для різноманітних проектів, від невеликих до великих корпоративних систем, і підтримка з боку спільноти забезпечує актуальність та надійність цього фреймворку.

xUnit, як і більшість фреймворків для юніт-тестування, складається з кількох основних компонентів, які разом забезпечують ефективне створення та виконання тестів. Ось основні складові та їх взаємодія:

- Test Runner – це програма або інструмент, який використовується для виконання тестів. Він знаходить та запускає тестові методи, використовуючи інші компоненти фреймворку;
- Test Classes – класи, які містять тестові методи. Кожен тестовий клас відповідає окремій групі тестів;

– Test Methods – методи, позначені атрибутами (наприклад, [Fact] або [Theory]), які виконують фактичне тестування;
– Asserts – компоненти, що використовуються для перевірки очікуваного результату тесту. Вони дозволяють визначити, чи тест пройшов успішно;
– Data Attributes – атрибути, такі як [InlineData], [ClassData], або [MemberData], використовуються для надання параметрів для параметризованих тестів;
– Setup/TearDown Logic: хоча xUnit використовує новий екземпляр тестового класу для кожного тестового методу, існують спеціальні методи (наприклад, конструктори для налаштування та IDisposable для очищення), які можуть використовуватися для підготовки та очищення перед і після кожного тесту.

Більш детально вигляд взаємодії компонентів зображено на рисунку 2.2.

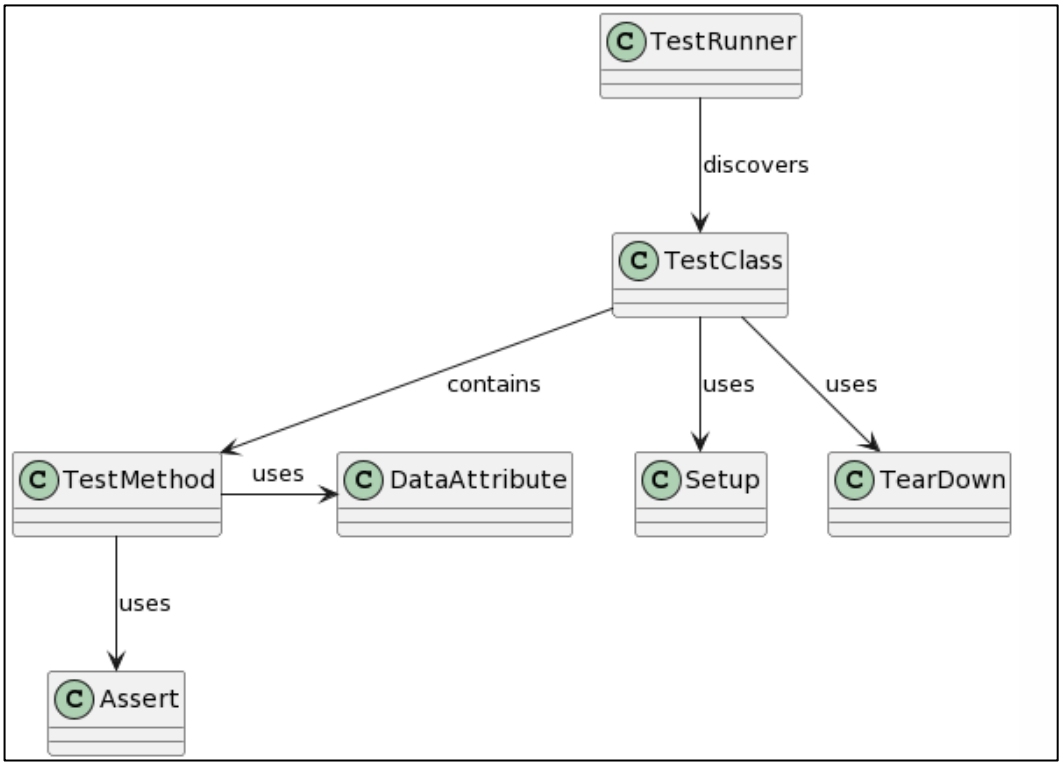

Рисунок 2.2 – Взаємодія структурних елементів xUnit

Розглянемо взаємодію цих компонентів:

– Test Runner ініціалізує процес тестування, виявляючи всі Test Classes;

- в кожному Test Class, Test Methods виконуються ізолюючи один від одного;
- під час виконання кожного Test Method, Asserts використовуються для перевірки умов тесту;
- Data Attributes застосовуються до Test Methods для надання параметрів у випадку параметризованих тестів;
- Setup/TearDown Logic використовується для налаштування та очищення ресурсів перед та після кожного тесту.

2.3 Метод тестування через MSTest

MSTest – це фреймворк для юніт-тестування, розроблений компанією Microsoft для .NET-платформи [5]. Він є частиною Visual Studio і широко використовується для створення, виконання та управління тестами у проектах .NET. MSTest інтегрується безпосередньо з середовищем Visual Studio, що робить його зручним інструментом для розробників, які використовують це середовище для розробки програмного забезпечення.

MSTest тісно інтегрований з Visual Studio, що забезпечує легкість у створенні, запуску та аналізі тестів без потреби використовувати сторонні інструменти або розширення. Ця інтеграція робить процес тестування більш ефективним і плавним для розробників, які вже працюють у середовищі Visual Studio.

Подібно до інших тестових фреймворків, MSTest використовує атрибути для визначення тестових класів і методів. Атрибут [TestMethod] використовується для позначення окремих методів як тестових, спрощуючи ідентифікацію та організацію тестів.

Фреймворк включає обширну бібліотеку асертів, яка дозволяє перевіряти різні умови в тестах, включаючи рівність, порівняння, винятки та інше [6]. Ці асерти є ключовими для перевірки коректності коду та логіки програми.

За допомогою атрибутів [TestInitialize] і [TestCleanup], MSTest дозволяє визначати методи для підготовки (setup) та очищення (teardown) ресурсів перед і

після кожного тесту. Це корисно для налаштування тестового середовища та гарантування його ізоляції.

MSTest підтримує параметризовані тести за допомогою атрибутів, таких як DataRow і DataTestMethod [7]. Це дозволяє проводити один і той же тестовий метод з різними вхідними даними, значно підвищуючи гнучкість та охоплення тестування.

Взаємодія компонентів наведена на рисунку 2.3.

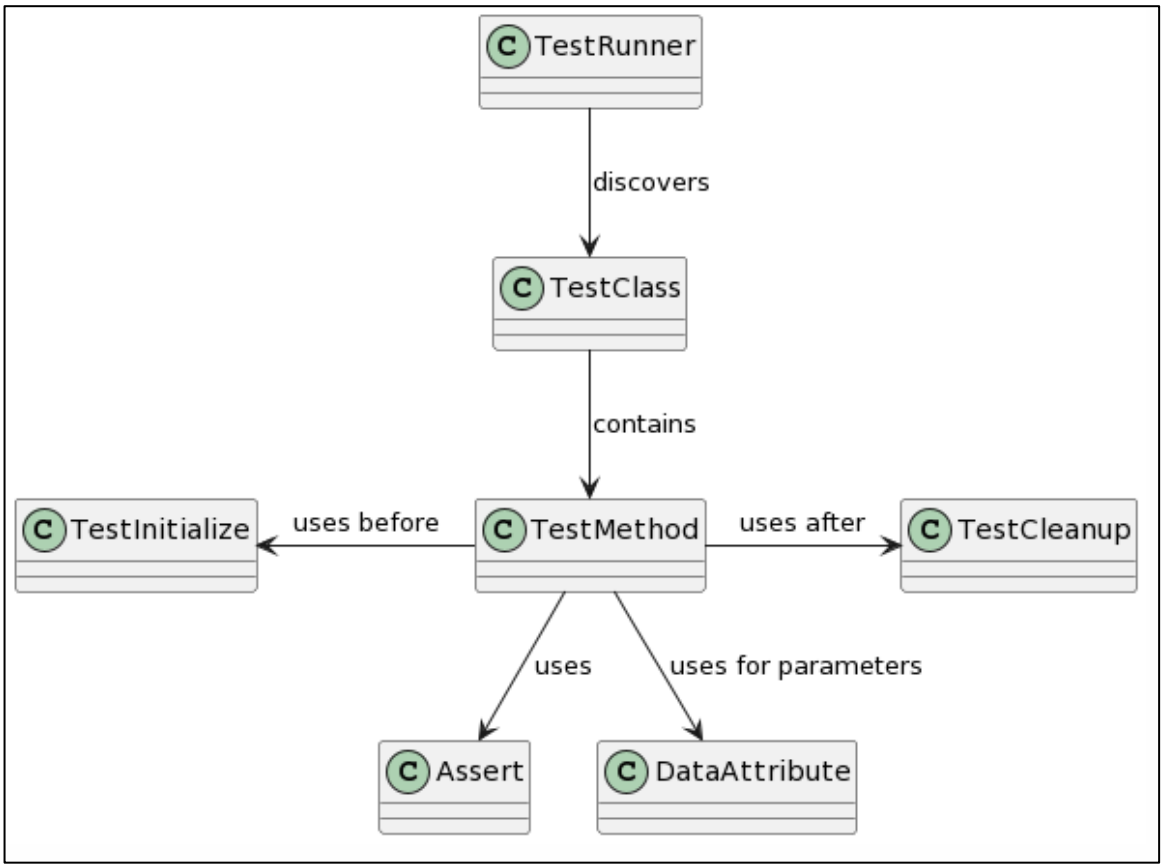

Рисунок 2.3 – Взаємодія структурних елементів MSTest

Із схеми можна побачити що Test Runner ініціалізує процес тестування, виявляючи та виконуючи Test Methods в межах Test Classes, Test Methods використовують Asserts для перевірки результатів.

Test Initialize/Cleanup методи запускаються перед і після кожного Test Method для налаштування та очищення ресурсів та Data Attributes використовуються для передачі параметрів у параметризовані Test Methods [8].

MSTest складається з декількох ключових компонентів, які разом забезпечують ефективне створення та виконання тестів у середовищі .NET:

- Test Runner – програма або утиліта, яка відповідає за виконання тестів. У Visual Studio це інтегровано безпосередньо в середовище розробки;
- Test Classes – класи, які містять тестові методи. Ці класи організовують тести та забезпечують структуру;
- Test Methods – окремі методи, позначені атрибутами (наприклад, [TestMethod]), які визначають фактичні тести;
- Asserts – методи, які використовуються для перевірки очікуваних результатів у тестах;
- Test Initialize/Cleanup – методи, позначені атрибутами [TestInitialize] та [TestCleanup], використовуються для налаштування та очищення перед і після кожного тесту;
- Data Attributes – атрибути, такі як [DataRow], які використовуються для надання параметрів для параметризованих тестів.

2.4 Особливості критеріїв мокування

Мокування (від англ. "mocking") у тестуванні програмного забезпечення – це процес створення імітаційних об'єктів, методів, змінних або систем, які імітують реальну поведінку компонентів програми, з метою ізолювати частини програми для більш ефективного та контрольованого тестування [9]. Різні бібліотеки та фреймворки надають різні можливості для мокування, і вибір інструменту залежить від специфічних вимог проєкту. Порівнюючи інструменти для мокування, можна звернути увагу на наступні параметри:

- легкість інтеграції: наскільки легко інструмент може бути інтегрований у ваш поточний стек технологій та процес розробки;
- підтримка мов програмування: чи підтримує інструмент мову програмування, на якій написаний ваш проєкт;

- гнучкість у створенні моків: здатність точно налаштовувати поведінку моків, включаючи виключення, повернення значень за певних умов тощо;
- підтримка автоматичного введення моків: автоматичне створення та інжектування моків у тестові випадки, що спрощує написання тестів;
- інтеграція з іншими інструментами тестування: сумісність із фреймворками для модульного тестування, системами неперервної інтеграції та іншими інструментами;
- відтворення та перевірка викликів: можливість перевірити, чи був викликаний певний метод з очікуваними аргументами;
- спільнота та документація: наявність детальної документації, прикладів використання та активна спільнота користувачів, яка може надати підтримку;
- ліцензія та вартість: умови використання інструменту, наявність безкоштовної версії або вартість ліцензії;
- підтримка асинхронного коду: можливість мокування асинхронних методів та функцій, що є критично важливим для сучасних асинхронних додатків;
- масштабованість та продуктивність: вплив інструменту на час виконання тестів, особливо в великих та складних проєктах, де оптимізація часу тестування є критичною;
- версіонування та сумісність: сумісність інструмента з різними версіями мов програмування та його здатність адаптуватися до оновлень мови або технологічного стеку;
- підтримка різних типів тестування: можливість використання інструмента для юніт-тестування, інтеграційного тестування, тестування арі та ін;
- розширені можливості: наявність додаткових функцій, таких як мокування зовнішніх http запитів, баз даних, вебсокетів тощо;

- екосистема: наскільки добре інструмент інтегрований з іншими інструментами та бібліотеками у вашому технологічному стеку;
- безпека: забезпечення безпеки мокових даних, особливо коли мокуються зовнішні сервіси або коли в тестах використовуються чутливі дані.

Обираючи інструмент для мокування, важливо розглянути всі ці параметри у контексті конкретних потреб проєкту та команди. Часто вибір зводиться не лише до технічних характеристик інструменту, але й до зручності його використання командою, наявності досвіду роботи з ним у розробників, а також до стратегічних рішень компанії щодо технологічного стеку.

Наприклад, для JavaScript існують такі популярні інструменти для мокування, як Jest, Sinon.js та Mockery, кожен з яких має свої переваги та недоліки в залежності від згаданих вище параметрів. В інших мовах програмування, таких як Java або Python, також існують свої рішення для мокування, як-от Mockito для Java або unittest.mock для Python.

3 ЗНАХОДЖЕННЯ КОМПЛЕКСНОГО ТА ЄФЕКТИВНОГО МЕТОДУ

3.1 Порівняння атибутів

Порівняння атрибутів у NUnit, xUnit і MSTest важливе для розуміння того, як кожен із цих фреймворків юніт-тестування використовує атрибути для організації та виконання тестів. Атрибути дозволяють фреймворкам ідентифікувати тестові методи та класи, керувати процесом тестування, налаштовувати тестове середовище та інші.

Незалежно від середовища модульного тестування C#, атрибути використовуються для опису класу, методів, властивостей тощо. Атрибути — це мета-теги, які надають додаткову інформацію про реалізацію під цим конкретним тегом.

Щоб зробити порівняння NUnit проти XUnit проти MSTest більш зрозумілим, ми розглядаємо важливі атрибути в кожній із цих структур. Порівняння атрибутів NUnit проти XUnit проти MSTest також допоможе перенести реалізацію тесту з однієї тестової системи на іншу. Список атрибутів наведено у додатку Г.

Щоб продемонструвати використання атрибутів і послідовність, у якій виконується тестовий код NUnit, ми розглянемо приклад простої реалізації тесту який представлений у додатку А.

Журнал виконання можна побачити на рисунку 3.1.

```
1 Inside SetUp
2 Inside TestMethod Test_1
3 Inside TearDown
4
5 Inside SetUp
6 Inside TestMethod Test_2
7 Inside TearDown
```

Рисунок 3.1 – Журнал виконання тестового блоку NUnit

Як видно з тестового блоку у додатку Б, атрибути [SetUp] і [TearDown] із структури NUnit відсутні. Замість цього конструктор використовується для ініціалізації, а метод Dispose використовується для деініціалізації.

xUnit створює новий екземпляр тестового класу для кожного тесту. У деяких критичних за часом сценаріях тестування виконання створення & код очищення може зайняти час і зробити тести повільнішими. Фікстури класу можна використовувати в таких ситуаціях, коли екземпляр фікстури створюється перед виконанням будь-якого з тестів, а після завершення виконання тестів відбувається очищення об’єкта фікстури за допомогою виклику Dispose (якщо він присутній).

У цьому прикладі використовується Class Fixtures, що означає, що навіть якщо є два тестових класи, код ініціалізації та очищення буде викликано лише один раз.

Розглядаючи MSTest можна побачити що тестовий код(див. додаток В) містить лише Console.WriteLine, який розміщується для відстеження потоку виконання. Більшість широко використовуваних атрибутів MSTest використовуються в продемонстрованому тестовому коді.

Реалізація під [ClassInitialize] & Атрибути [ClassCleanup] викликаються відповідно один раз перед виконанням методів і один раз після виконання тестових методів. Реалізація в атрибутах [TestInitialize] і [TestCleanup] викликається один раз перед & після виконання контрольних робіт у кожному класі [10].

У висновку зазначимо що – NUnit є відмінним вибором для розробників, які шукають гнучкість та розширені можливості для управління тестами. Його підтримка параметризації та можливість визначення порядку тестів робить його ідеальним для складних тестових сценаріїв.

xUnit – відзначається своєю високою ізоляцією тестів та чистотою коду. Це робить його підходящим для проектів, де важлива ізоляція та незалежність тестів. Його підхід до параметризації також ефективний та гнучкий.

MSTest – є зручним варіантом для тих, хто працює в середовищі Visual Studio і шукає стабільний, інтегрований засіб для тестування. Його підтримка Data-Driven тестів та простота у використанні роблять його популярним серед користувачів Visual Studio, особливо у великих проектах.

Кожен фреймворк має свої унікальні переваги та найкраще підходить для певних сценаріїв. Вибір між ними залежить від конкретних потреб проекту, переваг розробника, та контексту, у якому проводиться тестування.

3.2 Відмінності між NUnit, xUnit, MSTest

3.2.1 Ізоляція тестів

Фреймворк xUnit відзначається високою ступенем ізоляції тестів у порівнянні з фреймворками NUnit і MSTest. Для кожного окремого тестового випадку створюється відокремлений тестовий клас, який виконується і автоматично знищується після завершення тестування. Ця практика гарантує, що тести можуть бути запущені в будь-якому порядку, оскільки вони не мають взаємних залежностей. Виконання кожного тесту у власному контексті мінімізує ризик впливу одного тесту на успішність інших.

Крім того, в xUnit існує можливість надавати спільний доступ до коду налаштування і очищення, який можна використовувати між різними тестами без необхідності об'єднання екземплярів об'єктів. Один з шляхів реалізації цього - використання Class Fixtures, які були показані раніше. xUnit надає докладну документацію щодо того, як обмінюватися контекстами між тестами, що сприяє кращій організації коду і забезпечує його читабельність.

3.2.2 Розширюваність, ініціалізація та деініціалізація

При порівнянні різних фреймворків для тестування, таких як NUnit, xUnit і MSTest, важливим аспектом є розширюваність, яка впливає на вибір конкретного фреймворку. Вибір тестового фреймворку може залежати від потреб конкретного проекту, але можливості для розширення можуть визначити вибір структури для тестування.

У порівнянні з MSTest і NUnit, фреймворк xUnit відзначається більшою розширюваністю завдяки використанню атрибутів [Fact] і [Theory].

Важливо підкреслити, що багато атрибутів, які доступні в структурі NUnit, такі як [TestFixture], [TestFixtureSetup], [TestFixtureTearDown], [ClassCleanup], [ClassInitialize], [TestCleanup] і так далі, не включені в xUnit.

Особливий внесок у розширюваність xUnit зроблений за допомогою атрибута [Theory] для параметризованих тестів. Це спрощує можливість створювати власні функції тестування у структурі xUnit. Крім того, використання конструктора класу для ініціалізації тесту та інтерфейсу IDisposable для деініціалізації є прикладами розширеної функціональності xUnit у порівнянні з іншими фреймворками.

Важливо відзначити, що xUnit для кожного тесту створює новий екземпляр класу, в якому він розташований, тоді як у NUnit та MSTest всі тести виконуються у межах одного класу (одного приладу).

Фреймворк xUnit використовує підхід з Assert.Throws, відмінно від [ExpectedException], що застосовується в NUnit і MSTest. Однією з недоліків [ExpectedException] є те, що він може не повідомляти про помилки, якщо вони виникають у неправильній частині коду. Наприклад, якщо передбачається, що assert має виникнути через виняток безпеки, але виникає помилка аутентифікації, то [ExpectedException] не спрацює.

На відміну від цього, Assert.Throws викликає assert навіть у випадку, коли виняток є загальним. Це гарантує, що assert буде викликаний, навіть якщо виникне виняток.

Важливо відзначити, що всі три фреймворки для модульного тестування в C# підтримують паралельне виконання тестів і є добрими варіантами для автоматизованого тестування з використанням Selenium, оскільки швидкодія грає важливу роль в автоматизованому тестуванні.

3.3 Вибір кращого фреймворку для мокування

NUnit надає гнучкість через різноманітні атрибути, які можуть бути використані для деталізованого управління тестовими сценаріями. Наприклад,

атрибути [TestCase] та [Theory] дозволяють легко впроваджувати параметризовані тести.

Також NUnit підтримує паралельне виконання тестів, що дозволяє оптимізувати час виконання тестів і підвищує гнучкість в управлінні ресурсами.

xUnit використовує конструктори класів для ініціалізації та інтерфейс IDisposable для очищення, що дозволяє більшу гнучкість в управлінні станом тестів. Це сприяє написанню чистих та ізольованих тестових сценаріїв, також він використовує [Fact] для непараметризованих тестів і [Theory] для параметризованих, що дозволяє гнучко управляти даними тестів.

MSTest, будучи інтегрованим рішенням в Visual Studio, часто вважається менш гнучким порівняно з NUnit та xUnit, оскільки він менш орієнтований на спільноту та має меншу підтримку з боку сторонніх розробників інструментів.

У висновку NUnit та xUnit пропонують більшу гнучкість у контексті мокування завдяки своїм особливим підходам до ініціалізації, управління станом тесту, та більшому вибору атрибутів та підтримці з боку спільноти. Це робить їх більш підходящими для складних сценаріїв тестування, де важливо глибоке управління тестовими випадками та ізоляція станів. MSTest, навпаки, краще підходить для інтегрованих сценаріїв у середовищі Visual Studio, де може бути достатньо його базового функціоналу.

4 ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ БІБЛІОТЕК

4.1 Бібліотека MOQ

Бібліотека Moq є однією з найпопулярніших і широко використовуваних бібліотек для створення моків у мові програмування C#. Вона призначена для спрощення процесу тестування, дозволяючи розробникам легко імітувати об'єкти, що мають складні залежності. Це особливо корисно при тестуванні великих систем, де необхідно ізолювати роботу окремих компонентів для перевірки їх функціональності незалежно від інших частин системи. Moq використовує LINQ-подібний синтаксис для налаштування моків, що робить процес більш інтуїтивним та зрозумілим. З його допомогою можна налаштувати поведінку моків, задавши певні умови, при яких моки мають повертати певні значення, викликати виключення або виконувати певні дії.

Користувачі Moq можуть легко визначати, як об'єкти повинні реагувати на різні вхідні дані, а також перевіряти, чи правильно виконуються певні методи або властивості у тестованому коді. Це значно спрощує роботу при тестуванні коду, який інтегрується з базами даних, файловими системами або зовнішніми веб-сервісами. Більше того, Moq дозволяє розробникам концентруватися на написанні ємного коду, мінімізуючи потребу у великій кількості тестового коду для кожного сценарію, що робить процес розробки швидшим і ефективнішим.

Таким чином, Moq є важливим інструментом у арсеналі сучасного розробника, який дозволяє ефективно та точно тестувати програмне забезпечення, гарантуючи високу якість і надійність розроблюваних рішень.

Бібліотека Moq надає широкий спектр можливостей для мокування в тестуванні програмного забезпечення на .NET, забезпечуючи гнучкість і контроль над поведінкою імітованих об'єктів. Ось деякі з ключових функціональних можливостей, які роблять Moq таким корисним інструментом:

- легке створення моків: moq дозволяє легко створювати моки для інтерфейсів або класів з віртуальними членами. це включає методи, властивості, події та індексатори. моки можуть бути налаштовані так,

щоб вони поводилися певним чином, коли їх викликають під час тестування;

- налаштування повернення значень: за допомогою moq можна налаштувати моки повертати певні значення або виконувати певні дії при виклику методів. це дозволяє симулювати різні сценарії, наприклад, що б відбулося, якби зовнішня система повернула певні дані;
- перевірка викликів: moq дозволяє перевіряти, чи були викликані певні методи з очікуваними параметрами. це корисно для переконання, що тестований код взаємодіє з залежностями, як передбачено;
- виключення помилок: моки можуть бути налаштовані на виклик виключень при певних умовах, що дозволяє тестувати, як система реагує на помилки або виняткові умови;
- послідовні відповіді: moq може налаштовувати моки так, щоб вони повертали різні значення або викликали різні дії в залежності від кількості викликів методу, дозволяючи симулювати більш складні взаємодії;
- асинхронні методи: moq підтримує мокування асинхронних методів, що є важливим для сучасних додатків, які використовують асинхронне програмування для покращення відгуку та продуктивності;
- легка інтеграція: moq добре інтегрується з більшістю популярних фреймворків юніт-тестування, таких як nunit та xunit, що робить його зручним вибором для багатьох розробників .net.

Ці можливості роблять Moq потужним інструментом для розробників, які хочуть ефективно тестувати свої додатки, забезпечуючи, що їхні компоненти можуть правильно функціонувати в різних умовах і сценаріях.

4.2 Бібліотека Substitude

Бібліотека NSubstitute є ще одним популярним інструментом для мокування в екосистемі .NET, призначена для полегшення процесу створення моків і стабів у тестах. Вона була розроблена з метою зробити синтаксис створення моків якомога

більш чистим і зрозумілим, використовуючи властивості мови C#, такі як лямбда-вирази і розширення.

NSubstitute відрізняється від інших бібліотек своєю простотою і інтуїтивністю в налаштуванні моків. Розробники часто вибирають NSubstitute за її здатність швидко налаштовувати об'єкти з мінімальними зусиллями і використанням коду, що робить процес тестування менш трудомістким і більш ефективним. Крім того, бібліотека надає розробникам можливість легко перевіряти, як методи були викликані з відповідними аргументами, що є важливою частиною підтвердження правильності взаємодії між компонентами системи.

Завдяки своїй гнучкості і високому рівню абстракції NSubstitute чудово підходить для складних тестових сценаріїв, де потрібно імітувати поведінку зовнішніх систем або складних залежностей [11]. Вона також добре інтегрується з основними фреймворками юніт-тестування, такими як NUnit та xUnit, і є вибором багатьох розробників, які цінують чистоту і лаконічність коду.

Використання NSubstitute може значно знизити кількість коду, необхідного для написання повного комплекту тестів, забезпечуючи при цьому гнучкість і потужність, які потрібні для ефективного тестування.

NSubstitute є потужним інструментом для мокування в тестуванні програмного забезпечення, розробленим для платформи .NET. Ця бібліотека надає розробникам зручні інструменти для створення моків і стабів, дозволяючи імітувати поведінку об'єктів без необхідності створювати складні структури або великі обсяги коду.

Основною функцією NSubstitute є можливість швидко створювати фальшиві об'єкти, що імітують інтерфейси або класи з віртуальними членами. Це дозволяє розробникам тестувати частини програми в ізоляції від їхніх залежностей, що є ключовим у великих і складних програмних системах. Використання NSubstitute сприяє чіткішому розмежуванню компонентів та полегшує виявлення і виправлення помилок.

Розробники можуть встановлювати правила для моків, налаштовуючи їх на повернення певних значень або виконання певних дій при виклику методів. Це особливо корисно в сценаріях, де потрібно моделювати різноманітні відповіді залежностей, таких як помилки в базах даних або зовнішніх сервісах.

NSubstitute також дозволяє перевіряти, чи були викликані певні методи з очікуваними аргументами. Це дає можливість переконатися, що код взаємодіє з моками правильно, що є важливим для забезпечення високої якості та надійності програмного забезпечення.

Крім того, бібліотека підтримує тестування асинхронних методів, що робить її придатною для сучасних додатків, які використовують асинхронне програмування для підвищення продуктивності та відгуку.

Інтеграція NSubstitute з основними тестовими фреймворками, такими як NUnit і xUnit, робить її зручною для використання в багатьох проектах і забезпечує легке впровадження у вже існуючі процеси розробки. Ця бібліотека є ідеальним вибором для тих, хто шукає ефективний спосіб впровадження мокування в свої тестові стратегії, зменшуючи зусилля на написання і підтримку тестового коду.

4.3 Бібліотека FakeItEasy

Бібліотека FakeItEasy є ще одним зручним інструментом для створення моків і стабів у тестуванні програмного забезпечення в .NET. Цей фреймворк розроблений з акцентом на простоту використання і легкість налаштування поведінки моків, що робить його ідеальним вибором для розробників, які прагнуть максимально спростити процес написання тестів.

FakeItEasy дозволяє швидко створювати фальшиві версії об'єктів, інтерфейсів, класів із віртуальними методами та абстрактних класів, що значно полегшує модульне тестування шляхом ізоляції компонентів. Його API орієнтоване на читабельність і простоту, дозволяючи розробникам інтуїтивно налаштовувати моки за допомогою декларативного стилю. Розробники можуть легко задати, як моки повинні реагувати на певні виклики, повертаючи значення,

викидаючи винятки або виконуючи дії, що сприяє створенню детальних і контрольованих тестових сценаріїв.

Однією з ключових переваг FakeItEasy є його можливість допомогти розробникам відслідковувати інтеракції з моками, перевіряючи, що певні методи викликалися з відповідними аргументами. Це робить його незамінним інструментом для верифікації залежностей у складних програмах, де правильна взаємодія між компонентами може бути критично важливою.

Використовуючи FakeItEasy, розробники можуть ефективно автоматизувати і покращити процес тестування, забезпечуючи, що їх програми є стійкими, надійними та легко підтримуваними. Цей інструмент становить чудову альтернативу іншим бібліотекам мокування, забезпечуючи баланс між потужністю та простотою використання, що робить його популярним вибором у спільноті розробників .NET.

Бібліотека FakeItEasy надає широкий спектр можливостей для мокування і стабінгу, які полегшують процес тестування компонентів програмного забезпечення. Ось основні функціональні можливості FakeItEasy:

- створення фальшивих об'єктів: можливість легко створювати фальшиві версії інтерфейсів, класів з віртуальними методами, абстрактних класів та навіть звичайних класів (з деякими обмеженнями);
- налаштування поведінки фальшивих об'єктів: здатність налаштувати фальшиві об'єкти для повернення певних значень, викидання винятків, або виконання дій при виклику методів;
- перевірка викликів: функціональність для перевірки, чи були певні методи викликані на фальшивому об'єкті, з можливістю перевірки аргументів, що були передані цим методам;
- асинхронні методи: підтримка мокування асинхронних методів, дозволяючи тестувати асинхронні взаємодії без зайвих складнощів;
- конфігурація виключень: можливість налаштувати фальшиві об'єкти на автоматичне викидання винятків при виклику певних методів для імітації помилкових умов;

- підтримка лямбда виразів: використання лямбда виразів для конфігурації моків, що робить код чистішим і більш зрозумілим;
- ланцюжкові виклики: можливість налаштувати поведінку для ланцюжкових викликів у одному виразі, спрощуючи створення комплексних моків;
- інтеграція з тестовими фреймворками: легка інтеграція з популярними юніт-тестовими фреймворками, такими як nunit і xunit, для зручності використання в різних тестових середовищах.

Ці можливості роблять FakeItEasy важливим інструментом для розробників, які прагнуть ефективно ізолювати компоненти програми під час тестування, забезпечуючи гнучкість і контроль над процесом тестування.

4.4 Проблема мокінгу статики

Мокування статичних методів є складною задачею через особливості, які мають статичні елементи в багатьох мовах програмування, зокрема в C#. Наведемо кілька ключових причин, чому бібліотеки для мокування як Moq, NSubstitute, і FakeItEasy не підтримують обробку статичних методів:

- глобальний стан: статичні методи та властивості є частиною класу, а не екземпляру класу. вони мають глобальний стан, який зберігається між викликами і доступний в будь-якій точці програми. це означає, що зміна поведінки статичного методу може вплинути на всі частини програми, які його використовують, що ускладнює контроль та ізоляцію в тестах;
- ізоляція та відтворення: одна з основних цілей мокування – ізолювати компоненти для тестування, забезпечуючи, що залежності можуть бути точно симульовані і контрольовані. статичні методи, які впливають на глобальний стан, ускладнюють цю задачу, оскільки вони не можуть бути ізольовані від інших частин програми так само ефективно, як екземпляри класів;
- технічні обмеження: статичні методи і поля прив'язані до типів, а не до об'єктів. мокування зазвичай працює шляхом створення проксі-об'єктів

або підкласів для імітації поведінки оригінальних екземплярів класів. оскільки статичні методи не асоційовані з екземплярами, створення проксі для них стає технічно невиконуваним без глибоких втручань в роботу мови програмування;

- принципи дизайну програмного забезпечення: використання статичних методів часто вказує на потенційні проблеми в дизайні програми, такі як надмірна залежність від глобального стану або погана відокремленість залежностей. мокування статичних методів може відводити увагу від потреби в рефакторингу і покращенні архітектури програми.

Для обходу цих обмежень розробники часто використовують паттерни дизайну, як "Адаптер" або "Фасад", для створення обгорток навколо статичних методів, які потім можуть бути моковані. Це дозволяє краще контролювати залежності і полегшує тестування коду.

Бібліотеки для мокування як Moq, NSubstitute і FakeItEasy в .NET не можуть мокувати статичні методи або властивості через ряд обмежень, які випливають із природи статичних членів у програмуванні:

- статичні члени мають глобальний стан, що ускладнює їх ізоляцію для незалежного тестування, оскільки будь-які зміни можуть впливати на всю програму;
- технічні обмеження в мові програмування не дозволяють створювати проксі для статичних методів або властивостей, оскільки вони прив'язані до класу, а не до екземпляра об'єкта;
- відсутність спеціалізованих інструментів в бібліотеках мокування для роботи зі статичними членами також обумовлена впливом статичних методів на архітектурну чистоту програми, зокрема погану відокремленість залежностей і надмірну залежність від глобального стану.

5 ЕКСПЕРЕМЕНТАЛЬНЕ МОКУВАННЯ СТАТИКИ

5.1 Вирішення проблеми статики

Для вирішення проблеми було обрано бібліотеку Harmony.

Бібліотека Harmony є потужним інструментом для рантайм-патчингу методів у .NET додатках. Вона дозволяє модифікувати, перехоплювати або замінювати виклики методів без зміни вихідного коду бібліотек або програм. Ось декілька ключових аспектів, які роблять Harmony особливо корисною:

- на відміну від багатьох інших популярних бібліотек мокування, які обмежені мокуванням лише об'єктів або віртуальних/інтерфейсних методів, harmony дозволяє мокування статичних методів. це відбувається через маніпуляції на рівні il (intermediate language), що є мовою низького рівня, на якій базується .net;
- harmony дозволяє розробникам вставляти префікси та постфікси до існуючих методів. префікс виконується до оригінального методу і може змінити вхідні дані або навіть запобігти виклику оригінального методу. постфікс виконується після методу і може модифікувати результати або виконати додаткову логіку після основної операції;
- harmony вносить зміни безпосередньо під час виконання програми, що означає, що зміни не потребують перекомпіляції або зміни вихідного коду. це особливо корисно для модифікації поведінки закритих або запечатаних бібліотек, до яких немає джерельного коду;
- завдяки своїй гнучкості та потужності, harmony використовується не тільки для мокування у тестуванні, але й для створення модіфікацій до ігор, змін у поведінці складних системних додатків, або навіть у покращенні функціоналу програм без доступу до їх вихідного коду;
- використання бібліотеки може вирішити проблеми сумісності між різними версіями компонентів або відсутністю необхідних арі для досягнення бажаної функціональності.

Harmony є винятково потужним інструментом для розробників .NET, який надає значні можливості для рантайм-маніпуляції з кодом. Це дає розробникам значну свободу в модифікації, розширенні та тестуванні програмного забезпечення незалежно від доступу до вихідного коду компонентів. Структура взаємодії із рантаймом зображено на рисунку 5.1

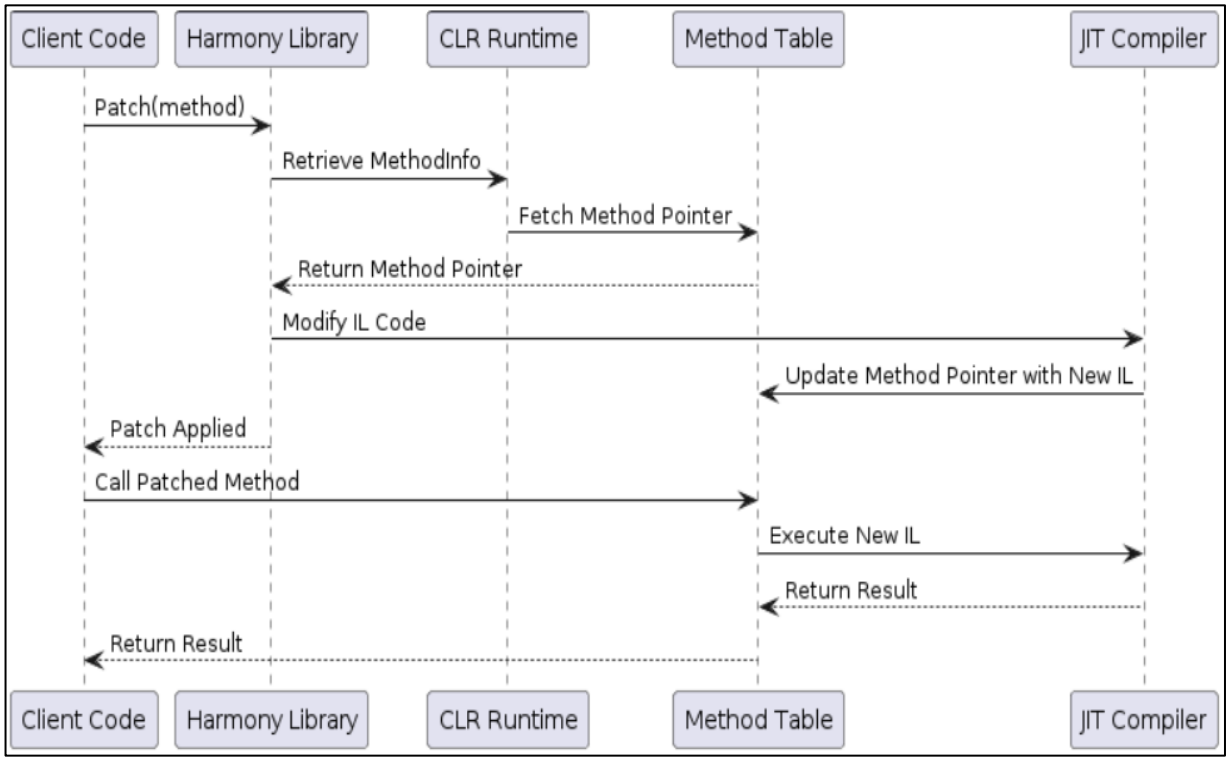

Рисунок 5.1 – Взаємодія Harmony із рантаймом

Опис компонентів і взаємодій:

- Client Code: це код, який ініціює процес мокування або патчінгу за допомогою Harmony;
- Harmony Library: бібліотека, яка забезпечує інтерфейси та методи для патчінгу методів;
- CLR Runtime (Common Language Runtime): середовище виконання .NET, яке відповідає за управління пам'яттю, виклик методів, збірку сміття тощо;
- Method Table: структура даних, яка містить покажчики на методи об'єктів або класів;

– JIT Compiler (Just-In-Time Compiler): компілятор, що виконує компіляцію IL коду у машинний код під час виконання програми.

Процес патчінгу:

– запит harmony на патчінг: код клієнта викликає метод у harmony для патчінгу певного методу;
– пошук інформації про метод: harmony запитує clr про інформацію методу, отримуючи покажчик з method table;
– модифікація il коду: harmony вносить зміни в il код, які відповідають патчу;
– оновлення покажчика на метод: jit компілює модифікований il код і оновлює покажчик методу в method table;
– виклик зміненого методу: коли клієнт викликає метод, викликається новий, змінений код.

5.2 Розробка методу мокінгу статики

Надалі для вирішення проблеми – нами було написано функціонал який через бібліотеку та її методи дозмолить мокати статику, код наведено нижче:

```
namespace MapTest
{ [TestFixture]
    public class ComplexCalculatorTests
    {private Harmony _harmony;
        [SetUp]
        public void Setup()
        {// Ініціалізуємо Harmony
            _harmony = new Harmony("com.example.complexcalculator");
            // Отримуємо метод, який потрібно патчити
            var originalMethod =
typeof(ComplexCalculator).GetMethod("ComplexOperation", BindingFlags.Static
| BindingFlags.Public);
            // Отримуємо метод патча
            var prefixMethod =
typeof(ComplexCalculatorPatches).GetMethod("Prefix", BindingFlags.Static |
BindingFlags.Public);
            // Застосовуємо патч
            _harmony.Patch(originalMethod, new
HarmonyMethod(prefixMethod));
        } [TearDown]
        public void TearDown()
        {// Вимкнути Harmony патч після завершення тесту
            _harmony.UnpatchAll(_harmony.Id);
```

```
        } [Test]
        public void ComplexOperation_Mocked_ReturnsFixedResult()
        {// Act
            var result = ComplexCalculator.ComplexOperation(5, 3);
            // Assert using TestContext for output
            TestContext.WriteLine($"Result of ComplexOperation: {result}");
            ClassicAssert.AreEqual(100, result, "The result should be fixed
to 100 due to mocking.");
        }public static class ComplexCalculator
        {public static long ComplexOperation(int x, int y)
            {return Factorial(x) + Factorial(y);}
            private static long Factorial(int num)
            {if (num < 0)
            throw new ArgumentException("Number must be non-negative");
                long result = 1;
                for (int i = 2; i <= num; i++)
                {result *= i;}return result;}}
        public static class ComplexCalculatorPatches
        {public static bool Prefix(ref long __result, int x, int y){
                __result = 100;  // Фіксоване значення для демонстрації
мокування
                TestContext.WriteLine($"Mocked result: {__result} for
inputs {x} and {y}");
                return false;    // Повертаємо false, щоб уникнути виклику
оригінального методу
            }}}}
```

Розглянемо детально, як працює наш тестовий клас ComplexCalculatorTests, який використовує бібліотеку Harmony для патчингу статичного методу ComplexOperation у класі ComplexCalculator.

Ініціалізація та налаштування Harmony (SetUp метод):

- створення інстанції harmony: при виклику методу setup, спершу створюється новий екземпляр harmony з унікальним ідентифікатором "com.example.complexcalculator". цей ідентифікатор важливий для уникнення конфліктів патчів, якщо в одній програмі використовується кілька інстанцій harmony;
- отримання інформації про метод для патчингу: використовується reflection api для знаходження методу complexoperation, який знаходиться у статичному класі complexcalculator. метод вибирається за його ім'ям та параметрами (у цьому випадку — публічний статичний метод);
- застосування патчу: harmony застосовує патч, використовуючи метод prefix, визначений у класі complexcalculatorpatches. цей метод буде

виконуватися перед оригінальним методом complexoperation і може модифікувати його поведінку або повністю замінити;

- виклик патченого методу: коли викликається метод complexoperation з параметрами 5 та 3, замість оригінальної логіки виконується патч (метод prefix), який встановлює результат виконання методу в фіксоване значення 100;
- перевірка результату: результат, який повертається з методу complexoperation, перевіряється на рівність 100. це підтверджує, що патч був застосований правильно і метод було змінено відповідно до вимог тесту;
- відміна патчів: після завершення тесту всі застосовані патчі відміняються, щоб очистити стан і уникнути впливу змін на інші тести або на подальше виконання програми;
- complexcalculator: статичний клас, що містить метод complexoperation, який обчислює суму факторіалів двох чисел. логіка включає розрахунок факторіала в допоміжному приватному методі factorial;
- complexcalculatorpatches: статичний клас, який визначає метод prefix, який перехоплює виконання complexoperation і встановлює результат в 100, ефективно замінюючи логіку обчислення.

Використання TestContext.WriteLine для виведення результату в консоль Test Runner-а дозволяє візуалізувати зміни в поведінці методу, підтверджуючи успішне мокування і вплив Harmony на роботу методу.

Тестування проводилось у веб-застосунку у проекті для тестів та Unit тестів (див. рис. 5.2).

У тестовому класі ComplexCalculatorTests, що використовує бібліотеку Harmony, процес ініціалізації починається зі створення екземпляру Harmony з унікальним ідентифікатором "com.example.complexcalculator". Це дуже важливо, адже унікальний ідентифікатор допомагає уникнути потенційних конфліктів, якщо в межах однієї програми використовуються кілька інстанцій Harmony.

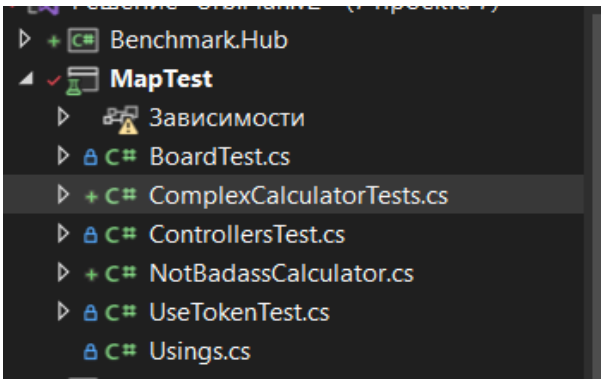

Рисунок 5.2 – Файлова структура тестів

За допомогою Reflection API визначається метод ComplexOperation у статичному класі ComplexCalculator, який вибирається за назвою та сигнатурою як публічний статичний метод. Далі застосовується патч за допомогою методу Prefix, розташованого у класі ComplexCalculatorPatches. Цей метод Prefix виконується перед викликом оригінального методу ComplexOperation і може змінити його поведінку або навіть повністю його замінити.

Коли в тестовому методі запускається ComplexOperation з параметрами 5 та 3, виконується патчений код, що встановлює результат методу на фіксоване значення 100 замість обчислення фактичної суми факторіалів цих чисел. Після виконання методу результат перевіряється на відповідність значенню 100, що підтверджує коректність застосування патчу і зміну поведінки методу згідно з вимогами тесту.

Після завершення тесту всі патчі, застосовані у рамках тесту, анулюються для очищення стану і запобігання впливу змінених методів на інші тести або на подальше виконання програми. Це важливий крок для забезпечення ізоляції тестів і уникнення побічних ефектів.

У класі ComplexCalculator метод ComplexOperation відповідає за обчислення суми факторіалів двох чисел, а розрахунок самого факторіалу відбувається у приватному методі Factorial. Ця структура демонструє чітку роздільність відповідальностей між методами. У класі ComplexCalculatorPatches метод Prefix перехоплює виконання ComplexOperation і встановлює результат обчислень у 100, ефективно ігноруючи оригінальну логіку методу.

5.3 Аналіз результатів

Застосування TestContext.WriteLine у тесті дозволяє вивести результат у консоль тестового середовища, що робить можливим візуалізацію змін у поведінці методу. Це підтверджує успішне мокування.

Результат наведено на рисунку 5.3

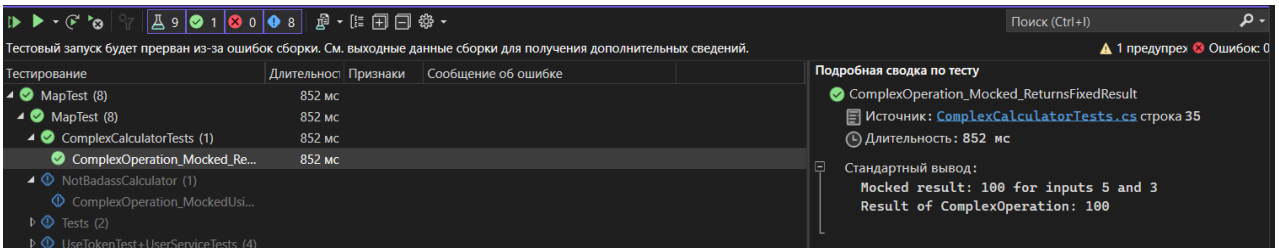

Рисунок 5.3 – Результат мокування статичного методу

Результати тесту виводяться у стандартний вивід, де вказано: "Mocked result: 100 for inputs 5 and 3", що підтверджує правильність мокування та роботи патча, а також "Result of ComplexOperation: 100", що відображає кінцевий результат виконання тестованого методу. Ці повідомлення забезпечують зрозумілість і перевірку того, що метод поводиться відповідно до очікувань після застосування модифікації.

Завдяки цим результатам, ми маємо доказ того, що бібліотека Harmony ефективно дозволяє мокувати статичні методи, що не підтримується більшістю традиційних бібліотек мокування. Це надає значних переваг у тестуванні коду, де необхідно ізолювати методи від їхніх залежностей або змінювати їхню поведінку на етапі виконання для перевірки різних сценаріїв [12].

Для прикладу наведемо реалізацію мокування через бібліотеку MOQ, результат зображено на рисунку 5.4.

На рисунку код спрямований на демонстрацію, що спроба встановлення моку для статичного методу призводить до помилки компіляції з повідомленням про неможливість використання статичних класів як аргументів типу для Moq. Це підтверджує, що традиційні бібліотеки мокування, такі як Moq, не здатні

обробляти статичні методи, що обмежує їхнє використання у певних сценаріях тестування.

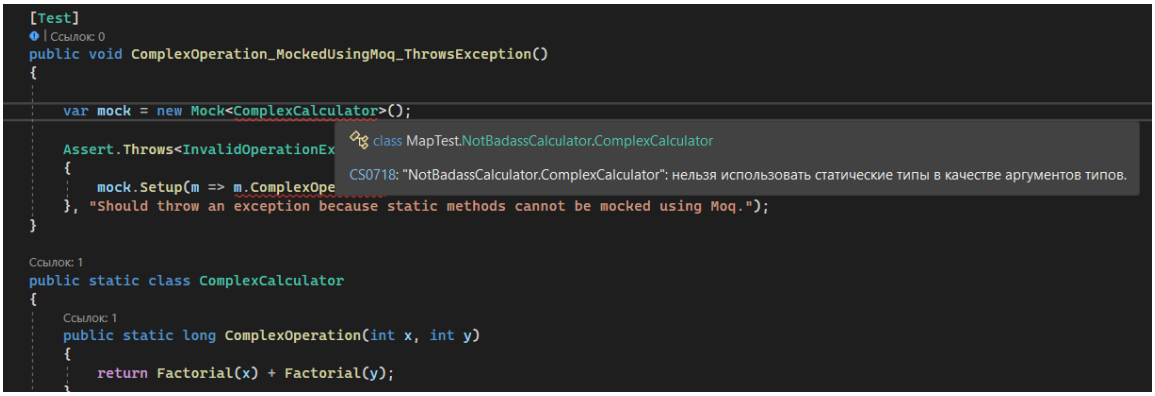

Рисунок 5.4 – Помилка у MOQ

Тому для мокування статичних методів потрібні спеціалізовані інструменти, як-от Harmony, що може модифікувати IL код на льоту, дозволяючи змінювати поведінку статичних методів.

ВИСНОВКИ

У цій роботі ми провели всебічний аналіз трьох провідних фреймворків для юніт-тестування в екосистемі .NET: NUnit, xUnit та MSTest. Кожен з цих фреймворків має свої унікальні особливості, переваги та можливі сценарії використання, що робить їх підходящими для різних типів проектів та потреб розробників.

NUnit відзначається своєю гнучкістю та розширеними можливостями, що робить його ідеальним для складних тестових сценаріїв. Підтримка параметризації та можливість визначення порядку виконання тестів дозволяє розробникам ефективно адаптувати NUnit до своїх потреб.

xUnit пропонує високу ступінь ізоляції тестів та чистоту коду, що робить його відмінним вибором для проектів, де необхідна ізоляція та незалежність тестів. Ефективність у параметризації та унікальний підхід до управління тестовим середовищем роблять xUnit привабливим для багатьох розробників.

MSTest, як інтегрований компонент Visual Studio, забезпечує легкість використання та зручність для користувачів Visual Studio. Цей фреймворк є гарним вибором для проектів, що вже інтегровані з екосистемою Microsoft та потребують тісної інтеграції з інструментами Visual Studio. Також вім завдяки більш пророблениим сценаріям тестування – більше підходить до процесу мокування.

Загалом щоб остаточно визначити найефективніший метод – потрібно додатково провести автоматизовані тести через локалні сітки та більш детальне крос браузерне тиестування.

В кінці нами було досліджено підходи різних бібліотек до мокування статичних методів та зясували що безкоштовні бібліотеки подібне робити не в змозі, тому за допомогою бібліотеки Harmony – ми через рантайм код змогли замокати статичний метод та продемонструвати працездатність підходу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз предметної області: https://www.telerik.com/products/mocking/unit-testing.aspx (дата звернення: 02.04.2024).

2. Аналіз існуючих рішень:

https://www.telerik.com/products/mocking/unit-testing.aspx (дата звернення: 05.04.2023).

3. Аналіз NUnit: https://nunit.org (дата звернення: 10.04.2024).

4. Аналіз імплементації xUnit: https://medium.com/nuances-of-programming/тестирование-сервиса-asp-net-core-с-помощью-xunit-3219c9530451 (дата звернення: 13.04.2024).

5. Аналіз MSTest: https://learn.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-with-dotnet-test (дата звернення: 17.04.2024).

6. Shubin I. , Kozyriev A. Method for Solving Quantifier Linear Equations for Formation of Optimal Queries to Databases. 2023 7th International Conference on Computational Linguistics and Intelligent Systems, Computational Linguistics Workshop, Kharkiv,Ukraine,8-20 April 2023. 21 April 2023.,URL: https://doi.org/10.1109/picst47496.2019.9061407 (дата звернення: 20.04.2024).

7. MSTest Documentation, URL: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest (дата звернення: 24.04.2024).

8. Hybrid system of computational intelligence based on bagging and group method of data handling / Y. Bodyanskiy et al. System research and information technologies. 2024. No. 1. P. 75–85. URL: https://doi.org/10.20535/srit.2308-8893.2024.1.06 (date of access: 24.04.2024).

9. Stack Overflow - Mocking Libraries, URL: https://stackoverflow.com/ (дата звернення: 29.04.2024).

10. Jenkins Documentation, URL: https://www.jenkins.io/ (дата звернення: 03.05.2024).

11. Azure DevOps Documentation, URL: https://azure.microsoft.com/en-us/services/devops/ (дата звернення: 03.05.2024).

12. Models of adaptive integration of weighted interval data in tasks of predictive expert assessment / I. Ruban et al. Eastern-European Journal of Enterprise Technologies. 2022. Vol. 5, no. 4(119). P. 6–15. URL: https://doi.org/10.15587/1729-4061.2022.265782 (date of access: 06.05.2024).