Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Магістра
(назва освітнього ступеня)

на тему: Проектування та розробка фреймворку автоматизації десктопних додатків та API на базі C# з використанням TestStack.White та HttpClient

Виконав(ла): студент(ка) 6 курсу, групи СПм-61
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

		Гавура Р. В.
	(підпис)	(прізвище та ініціали)
Керівник		Бойко І. В.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		Бойко І. В.
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Петрик М. Р.
	(підпис)	(прізвище та ініціали)
Рецензент		Дмитроца Л. П
	(підпис)	(прізвище та ініціали)

Тернопіль
2020

АНОТАЦІЯ

Кваліфікаційна робота магістра на тему «Проектування та розробка фреймворку автоматизації десктопних додатків та API на базі C# з використанням TestStack.White та HttpClient» Гавури Романа Валентиновича. – Тернопільський національний технічний університет імені Івана Пулюя, Факультет комп'ютерно-інформаційних систем і програмної інженерії, Кафедра програмної інженерії, група СПм–61 // Тернопіль, 2020.

С. – 71, рис. – 14, табл. – 4, додат. – 3, бібліогр. – 57.

Метою роботи є дослідження та огляд явища автоматизації тестування програмного забезпечення, огляд найкращих методів, які використовуються фахівцями в даній сфері, а також проектування та реалізації автоматизованого тестування в життєвий цикл розробки програмного забезпечення.

Методи та програмні засоби, використані при виконанні розробки та проектування системи: мова програмування C# та .NET Framework, середовище розробки Visual Studio, інструмент TestStack.White, бібліотека HttpClient, фреймворки MS Test та NUnit, гнучка методологія розробки (Agile), патерн проектування ScreenObject, інструмент CI/CD Bamboo.

Результатом роботи є розроблений фреймворк автоматизації тестування, який дозволяє запускати тестові сценарії, надавати звітність з результатів виконання тестів для подальшого аналізу, а також спроектовані та реалізовані тестові випадки.

Ключові слова: ПРОЦЕС РОЗРОБКИ ПРОГРАНОГО ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ІНТЕРФЕЙС КОРИСТУВАЧА, C#, SCREEN OBJECT, API, MS TEST, NUNIT.

ABSTRACT

Masters qualification work on the topic «Design and development of the desktop and API automation framework with C# using TestStack.White and HttpClient» by student Havura Roman Valentynovych. – Ternopil Ivan Pul'uj National Technical University, Faculty of Computer Information Systems and Software Engineering, Software engineering department, group SPm-61 // Ternopil, 2020.

Pages. – 71, pictures. – 14, tables. – 4, add – 2, bibl.ref. – 57.

The purpose of the work is to analyze and study automated testing of software products as a phenomenon, to look through best methods and practices that are being used by the field specialists, as well as the design and implementation of automated testing into a software development lifecycle.

The tools and methods used during the implementation and design of the software system: C# programming language and .NET Framework, integrated development environment Visual Studio, TestStack.White, HttpClient library, unit testing frameworks MS Test and NUnit, agile development methodology, ScreenObject design pattern, Bamboo CI/CD tool.

The result of the work is implemented test automation framework that allows to run test scripts, provide test run results for further analysis, along with implemented test scenarios.

KEYWORDS: SOFTWARE DEVELOPMENT PROCESS, AUTOMATED TESTING, SOFTWARE QUALITY, USER INTERFACE, C#, SCREEN OBJECT, API, MS TEST, NUNIT.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПЗ	програмне забезпечення
UI (User Interface)	інтерфейс користувача
IDE (Integrated Development Environment)	комп'ютерна програма, що допомагає програмістові розробляти нове програмне забезпечення чи модифікувати (удосконалювати) вже існуюче
API (Application Programming Interface)	набір визначень взаємодії різнотипного програмного забезпечення
SDK (Software Development Kit)	набір із засобів розробки, утиліт і документації, який дозволяє програмістам створювати прикладні програми за визначеною технологією або для певної платформи (програмної або програмно- апаратної)
Система контролю версій	програмний інструмент для керування версіями одиниці інформації: вихідного коду програми, скрипту, веб-сторінки, веб-сайту, 3D моделі, текстового документу тощо

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 5

ВСТУП 8

1. ОГЛЯД ІНТСРУМЕНТІВ ТА АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ 11

1.1 Огляд середовищ розробки 11
1.1.1 Середовище розробки Rider 15
1.1.2 Використання Inspect 16
1.1.3 Використання Swagger 17
1.1.4 Використання Insomnia 18

1.2 Огляд мов програмування для розробки 19
1.2.1 C# 19
1.2.2 Особливості і переваги C# 21

1.3 Аналіз вимог до тестування програмної системи 23

1.4 Огляд та аналіз предметної області 24

1.5 Опис системи конструювання документів Litera Forte 26

1.6 Роль тестування у циклі розробки програмного забезпечення 28

1.7 Рівні тестування 29

1.8 Види тестування 31
1.8.1 Функціональне тестування 31
1.8.2 Нефункціональне тестування 33
1.8.3 Тестування змін 34

1.9 Автоматизоване тестування 35

2. СПЕЦІАЛЬНА ЧАСТИНА 42

2.1 Розгортання тестового середовища 42

2.2 Опис та реалізація основних тестових випадків 44
2.2.1 Використання TestStack.White 47
2.2.2 Створення тестового фреймворку. 49
2.2.3 Реалізація тестових випадків 57
2.2.4 Реалізація тестових випадків API 66

2.3 Використання Bamboo CI/CD 70

3 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 72

3.1 Охорона праці 72

3.2 Безпека в надзвичайних ситуаціях 73

ВИСНОВКИ.....79

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....81

ДОДАТОК А Технічне завдання 87

ДОДАТОК Б Лістинг коду запуску та очистки тестів.....93

ДОДАТОК Б Тези.....98

ВСТУП

Програмна інженерія це процес аналізу потреб користувача, і потім, відповідно до потреб, дизайн, конструювання і тестування кінцевих користувацьких застосунків які задовільнять ці потреби через використання мов програмування.

Тестування це невід'ємна частина процесу розробки програмного забезпечення, і для застосунків які прагнуть досягнути найвищих стандартів якості, тестування може бути нелегким процесом. З збільшенням кількості функціональних можливостей, якими наділені сьогодні програми, не кажучи вже про велику кількість платформ та браузерів, на які потрібно зважати, постійно зростає можливість помилок та проблем залишитись непоміченими. Однак висококонкурентний ринковий сценарій не дозволяє розробникам програмного забезпечення розкіш дозволяти продуктам навіть з незначними помилками виходити на ринок, оскільки такий продукт буде відхилений.

Ручне тестування програмного забезпечення дозволяє порівняти застосунок з вимогами та очікуваннями клієнта. Тестер знає домен програми дуже добре і може перевірити чи є в новій версії застосунку невідповідності. Якщо застосунок часто оновлюється, тестер буде виконувати свої стандартні дії під час процесу тестування. Це означає що він буде виконувати одні і ті самі дії знову і знову. Основною ціллю створення автоматизованих тестів не є пошук нових помилок, а швидше перевірка, чи з’являються відхилення в новій версії продукту від попередньої.

Найбільш багатообіцяючим аспектом процесу автоматизації тестування є цінність, яку вона додає для всіх зацікавлених сторін. Це покращує імідж бренду, приносить більший дохід та забезпечує вищий рівень утримання клієнтів. Як результат, відбувається збільшення інвестицій у дослідження продуктів та інновації процесів, тим самим допомагаючи організації підкоряти нові вершини та здобувати перевагу на ринку. Автоматизована тестування

допомагає створювати більш якісне програмне забезпечення з меншими зусиллями.

Багато компаній, в певній мірі, вже застосовують автоматизоване тестування, але все ще значною мірою залежать від ручного тестування, оскільки вони не знають, як правильно використовувати переваги автоматизованого тестування в їхньому процесі розробки програмного забезпечення.

Тестування вручну проводиться шляхом ретельного виконання заздалегідь визначених тестових випадків, порівняння результатів із очікуваною поведінкою та запис результатів. Тести вручну повторюються щоразу, коли вихідний код змінюється і схильний до помилок. Ручне тестування також важко виконувати на декількох платформах.

Необхідно вкласти багато часу та зусиль при впровадженні автоматизованого тестування в організації. Однак фінансових зобов’язань не надто багато, принаймні починаючи з малого масштабу. Існує безліч засобів автоматизації з відкритим кодом, якими можна скористатися, особливо на ранніх стадіях впровадження процесу.

Враховуючи що більшість компаній, при розробці продукту, навіть якщо це на початкових етапах десктопний додаток, в майбутньому приходять до розглядання розробки веб версії додатку, це не дивно, тому, що більшість розроблюваних продуктів це веб застосунки.

При розробці веб додатку не можна не згадати про REST. REST API широко розглядається як стандартний протокол для web-api. Тоді як інші протоколи, що використовуються для web-api, менш популярні (SOAP).

REST API швидко набирає популярність з 2005 року. Розробники покладаються на цей API, оскільки його легше зрозуміти, в порівнянні з іншими web-api.

В популярності REST API є кілька причин. Наприклад, оскільки REST використовує стандартні запити HTTP, API для перевірки даних та адрес є простими для розуміння та використання дизайнерами. Більше того, архітектури

RESTful спрощують надання результатів у більш пристосованих форматах даних, таких як JSON.

Враховуючи важливість автоматизації тестування та популярність REST API, можна прийти до висновку що рішення компанії інвестувати в автоматизацію тестування REST API є виправданою та логічною.

Метою магістерської роботи є розробка фреймворку автоматизації тестування десктопних додатків та автоматизації REST API використовуючи C# та TestStack.White та HTTPClient. Користувачами цього додатка будуть як і ручні тестери, так і розробники.

Для досягнення поставленої мети потрібно вирішити такі завдання:

1. Провести огляд літератури, з обраної тематики;
2. Вивчити архітектуру, інструменти та особливості розробки рішень з автоматизованого тестування на C#;
3. Описати моделі даних;
4. Розробити основний фреймворк;
5. Розробити початковий набір тестів;
6. Протестувати розроблений фреймворк.

1. ОГЛЯД ІНТСРУМЕНТІВ ТА АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

1.1 Огляд середовищ розробки

Microsoft Visual Studio – це інтегроване середовище розробки від Microsoft. Воно використовується як для розробки комп'ютерних програм, так і для розробки веб сайтів, веб додатків, веб сервісів і мобільних додатків. Visual Studio використовує платформи розробки програмного забезпечення Microsoft, такі як Windows API, Windows Forms, Windows Presentation Foundation, Windows Store та Microsoft Silverlight. Visual Studio може виробляти як і машинний так і керований код.

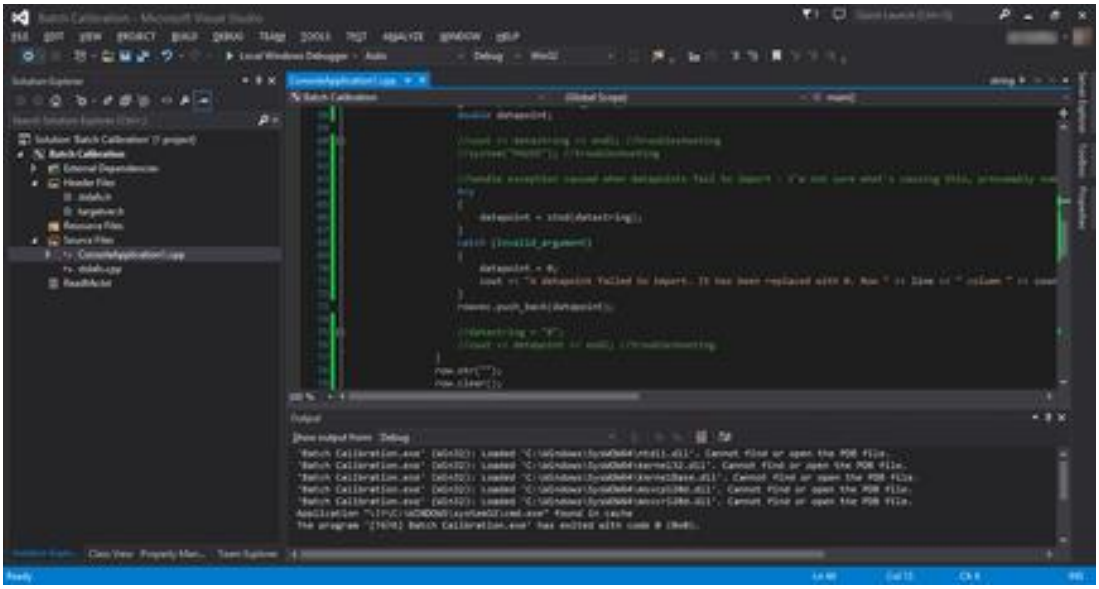

Рисунок 1.1 – Microsoft Visual Studio

Visual Studio включає в себе редактор коду який підтримує IntelliSense (компонент завершення коду) і також інструменти рефакторингу. Інтегрований дебагер працює як і з вихідним кодом, так і з кодом машинного рівня. Інші вбудовані інструменти включають код профайлер, дизайнер для будування GUI додатків, веб дизайнер, дизайнер класів, і дизайнер схем баз даних. Також

підтримуються плагіни які розширюють функціонал на майже кожному рівні, додаючи підтримку систем контролю версій (таких як Subversion та Git) і додаючи нові набори інструментів, такі як редактори і візуальні дизайнери для специфічних доменних мов, або набори інструментів для інших аспектів циклу розробки програмного забезпечення [1].

Visual Studio підтримує 36 різних мов програмування і дозволяє редактору коду та дебагеру підтримувати (в певній мірі) майже будь яку мову програмування, якщо існує специфічний сервіс цієї мови. Вбудовані мови включають C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML і CSS. Підтримка таких мов як Python, Ruby, Node.js та M доступна через плагіни. Java та J# підтримувались в минулому.

Visual Studio має велику кількість можливостей для того щоб полегшити процес розробки, включаючи наступні:

- середовище для редагування коду, що включає можливості підсвічування синтаксису, автозаповнення коду та індексацію символів;
- можливість простого дебагінгу, профілювання та діагностики коду;
- комплексні інструменти тестування які допомагають писати високоякісний код;
- просто інтеграція системи контролю версіями для гнучкої і ефективної сумісної роботи;
- тисячі розширень які дозволяють налаштувати середовище розробки під свої потреби;
- розробка додатків та ігор для всіх пристроїв під управлінням Windows;
- створення власних або гібридних додатків для Android, iOS та Windows;
- проста збірка, розгортання масштабованих хмарних додатків і керування ними в Azure;

- розробка сучасних веб додатків з гнучкими і багатофункціональними методами розробки з відкритим вихідним кодом ;
- багатофункціональні методи для всіх типів розробки з Office;
- розробка, написання коду і дебагінг ігор з використанням сучасних інструментів графіки та скриптів;
- написання власних розширень для Visual Studio;
- проста розробка і розгортання баз даних SQL Server та SQL Azure;

IntelliTest (для створення та обслуговування модульних тестів) (рис. 1.2)

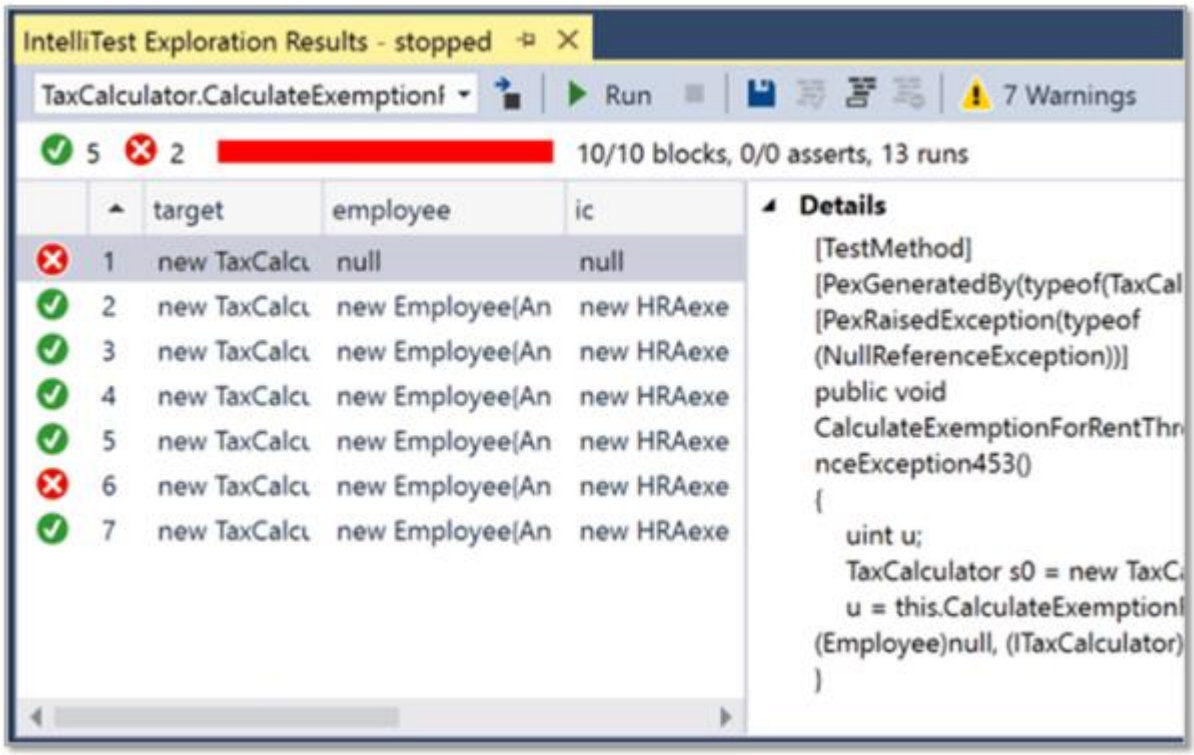

Рисунок 1.2 – IntelliTest

IntelliTest оглядає .NET код та генерує тестові дані і набір юніт тестів. Для кожного виразу в коді генерується тестовий набір даних який виконає даний вираз. Також проводиться аналіз для кожного умовного розгалуження в коді. Наприклад, if вирази, перевірки, і всі операції які можуть викликати помилки будуть проаналізовані. Цей аналіз використовується для генерації тестових

даних для параметризованих юніт тестів для кожного методу, створюючи юніт тести з високим покриттям коду.

Запустивши IntelliTest, можна легко побачити які тести не працюють і додати необхідний код для того щоб їх виправити. Можна вибрати генеровані тести для збереження в тестовий проект, щоб надати регресивний набір тестів. Змінюючи код потрібно перезапускати IntelliTest для того щоб згенеровані тести синхронізувались з змінами в коді.

Використовуючи Test Explorer можна запускати юніт тести з Visual Studio або з юніт тести проектів третіх сторін. Test Explorer також можна використовувати для групування тестів за категоріями, фільтрування списку тестів та створення, зберігання і запуску списків відтворення з тестами. Також можна аналізувати покриття коду та відлагоджувати юніт тести.

Test Explorer може запускати тести з декількох тестових проектів в рішенні. Тестові проекти можуть використовувати різні тестові фреймворки. Коли код який тестується написаний для .NET, тестовий проект може бути написаний на будь-якій мові яка також націлена на .NET, незважаючи на мову, якою написаний код який тестується.

Для створення юніт тест проекту в Visual Studio потрібно виконати наступні кроки:

1. Відкрити проект який буде тестуватись в Visual Studio;
2. Додати новий юніт тест проект до існуючого рішення;
3. Додати посилання на проект з кодом, який буде тестуватись;
4. Додати код в юніт тест метод.

Рисунок 1.3 – Test Explorer

1.1.1 Середовище розробки Rider

Rider – це інтегроване середовище розробки (IDE) .NET, ASP.NET, .NET CORE, Xamarin та Unity на Windows, Mac та Linux. Rider можна розглядати як аналог Visual Studio, особливістю Rider є його схожість до найпопулярнішого середовища розробки для Java – IntelliJ IDEA, що робить дане середовище хорошим вибором для розробників які мали досвід розробки використовуючи Java, та перейшли на .NET.

Rider це крос-платформна IDE для .NET розробників, заснована на платформі IntelliJ і ReSharper.

Rider підтримує .NET Framework, нову платформу .NET Core і проекти на основі Mono. IDE дозволяє розробляти десктопні додатки, .NET сервіси і бібліотеки, ігри на рушії Unity, мобільні додатки Xamarin, веб додатки ASP.NET і ASP.NET Core.

Rider надає більше 2200 інспекцій коду, сотні контекстних дій і рефакторингів, запозичених з ReSharper, разом з прогресивним функціоналом середовищ розробки на основі платформи IntelliJ. Не дивлячись на великий набір функцій, Rider – швидка і чуйна IDE. Механізм аналізу помилок буде шукати помилки по всьому рішенню і повідомляти про них, навіть якщо проблемний файл не відкритий в редакторі.

Розумний редактор Rider надає різні види автодоповнення і шаблонів, автоматично вставляє потрібні знаки і імпортує простори імен яких недостатньо. Підказки і іконки на полях допомагають легко переміщуватись по ієрархії наслідування, контекстні дії роблять розробку зручною та ефективною.

Rider запозичує з ReSharper більше 60 рефакторингів і передбачає більше 450 контекстних дій для різних цілей. Рефакторингі дозволяють з легкістю перейменовувати і витягувати методи, інтерфейси і класи, переміщати і копіювати типи, використовувати альтернативний синтаксис і виконувати інші перетворювання.

Rider допомагає запускати та відлагоджувати юніт тести NUnit, xUnit.net та MSTest. ReSharper допомагає вивчати тести, групувати їх по сесіях, переглядати результати тестів і переходити до вихідного коду з трасуючого стеку.

Вбудований відладчик для застосунків на .NET Framework, Mono і .NET Core підтримує покрокове виконання, дозволяє обраховувати вираження на льоту, запускати програму від поточної виконуваної стрічки до стрічки з курсором, відслідковувати і міняти значення змінних. Крім того, Rider включає в себе браузер NuGet, дозволяє переглядати трасування стеку, підтримує різноманітні системи контролю версій і бази даних.

Можна працювати з SQL і базами даних прямо в IDE. Rider допомагає підключатися до баз даних, редагувати схеми і таблиці, виконувати запити і аналізувати схеми за допомогою UML-діаграм.

Моментально можна переходити до будь-якого файлу, типу або члену в базі коду, а також швидко знаходити потрібні налаштування і дії IDE. Від будь-якого символу в коді можна моментально перейти до базових і породжуючих символів, реалізацій, а також місць виконання.

Rider підтримує JavaScript, TypeScript, HTML, CSS і Sass. Спеціально для цих технологій IDE включає в себе можливості рефакторингу, відлагодження і модульного тестування з WebStorm.

Rider використовують багато провідних компаній, та, в загальному, порівнюючи з Visual Studio, Rider працює швидше та виглядає більш сучасно.

1.1.2 Використання Inspect

Inspect – це інструмент який базується на платформі Windows, що дозволяє вибирати будь-які UI елементи і переглядати дані цих елементів. Можна переглядати властивості Microsoft UI Automation a Microsoft Active Accessibility. Inspect також дозволяє тестувати навігаційну структуру елемента

автоматизації в UI Automation дереві, і доступ до об’єктів в ієрархії Microsoft Active Accessibility.

Inspect встановлюється разом з Windows Software Development Kit (SDK). Він знаходиться в \bin\<version>\<platform> папці де встановлений SDK. Слід зауважити що Inspect є застарілим інструментом, і Microsoft рекомендує використовувати Accessibility Insights.

Вікно Inspect складається з декількох основних частин:

- Title bar;
- Menu bar;
- Toolbar;
- Tree view;
- Data view.

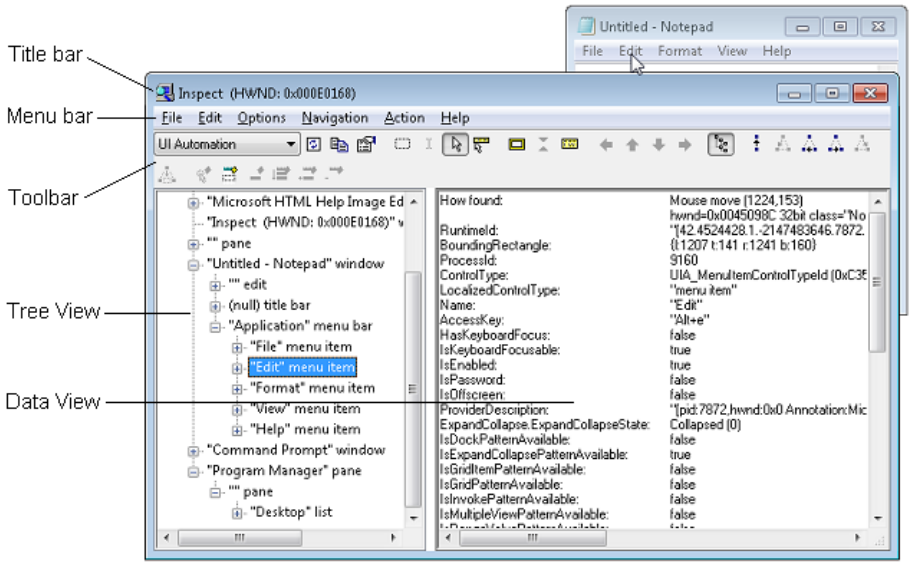

Рисунок 1.4 – Inspect

1.1.3 Використання Swagger

Swagger це мова опису інтерфейсів для опису RESTful APIs виражених через JSON. Swagger використовують разом з наборами програмного забезпечення відкритого доступу, для дизайну, розробки, документації та використання RESTful веб сервісів. Swagger включає автоматизовану

документацію, генерацію коду (підтримуючи багато мов програмування), і генерацію тестових випадків.

Використання Swagger можна розділити на кілька основних категорій: розробка, взаємодія з API, і документація [2].

При створенні API, Swagger надає інструменти, за допомогою яких можна автоматично генерувати Open API документ який базується на коді. Це вбудовує опис API в вихідний код проекту. Альтернативно, використовуючи Swagger Codegen, розробники можуть розділити вихідний код від Open API документу, і генерувати клієнтський і серверний код напряму з дизайну. Це дозволяє відкласти аспект кодування.

Використовуючи Swagger Codegen проект, кінцевий користувач генерує клієнтське SDK прямо з OpenAPI документу, зменшуючи потреби людського генерування коду.

Якщо описаний OpenAPI документом, Swagger можна використовувати для взаємодії з API через Swagger UI. Даний проект дозволяє підключатись до API напряму через інтерактивний HTML інтерфейс. Запити можна робити прямо з інтерфейсу користувача.

1.1.4 Використання Insomnia

Insomnia це крос-платформний REST клієнт, побудований на Electron. Insomnia доступна для Mac, Windows і Linux.

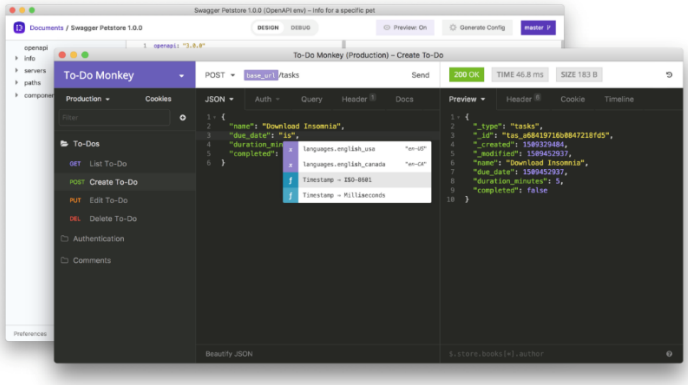

Рисунок 1.5 – REST клієнт Insomnia

Використовуючи Insomnia можна швидко створювати та групувати запити, задавати змінні середовища, автентифікацію та генерувати фрагменти коду.

Також можна отримати усі деталі відповідей. Переглянути весь життєвий цикл запиту, статус коди, тіло, заголовки, кукі, та інше. Також підтримується створення робочих груп, папок, середовищ, drag-and-drop запити, і просте імпортування та експортування даних.

Insomnia підтримує створення, редагування, і керування всіма специфікаціями OpenAPI в єдиному середовищі. Підтримується генерування конфігурацій для Kong API Gateway та Kong for Kubernetes.

В Insomnia можна синхронізувати свої API дизайни з систем контролю версій, таких як Github або Gitlab, і розгортати напряму в API Gateways одним кліком.

Insomnia дає можливість писати юніт тести використовуючи JavaScript, та запускати їх в терміналі, використовуючи Inso.

1.2 Огляд мов програмування для розробки

1.2.1 C#

Програмування – основа основ комп'ютерної техніки. C# це сучасна об’єктно-орієнтована і типобезпечна мова програмування.

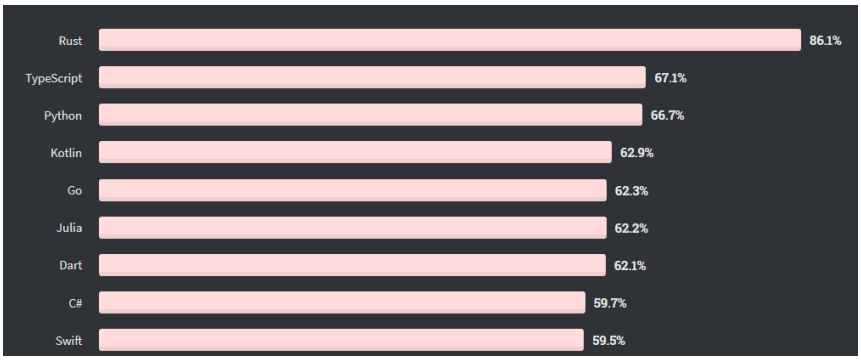

Рисунок 1.6 – Stack Overflow 2020 Most Loved мови програмування

C# відноситься до широко відомого сімейства мов C, і буде здаватись добра знайомою будь-кому, хто працював з C, C++, Java або JavaScript.

C# - це об’єктно і компонентно-орієнтована мова програмування. C# представляє мовні структури для безпосередньої підтримки такої концепції роботи. Завдяки цьому дана мова підходить для створення і застосування програмних компонентів. З моменту створення, мова C# набула функцій для підтримки нових робочих навантажень і сучасних рекомендацій по розробці програмного забезпечення.

Ось декілька функцій мови C#, які забезпечують надійність і стійкість додатків. Збірник сміття автоматично звільняє пам’ять, зайняту недосяжними використовуваними об’єктами. Обробник вийняткових ситуацій надає структурований та обширний підхід до виявлення помилок і відновлення після них. Лямбда-вирази підтримують прийоми функціонального програмування. Синтаксис запитів створює загальний шаблон для роботи з даними з будь-якого джерела. Підтримка мов для асинхронних операцій надає синтаксис для створення розподілених систем. Співставлення шаблонів надає синтаксис для простого розділення даних з алгоритмів в сучасних розподілених системах [3].

В C# діє єдина система типів. Всі типи C#, включаючи типи примітиви, такі як int та double, унаслідуються від єдиного кореневого типу object. Всі типи використовують загальний набір операцій, а значення будь-якого типу можна зберігати, передавати і обробляти подібним образом. Більш того, C# підтримує як і оголошені користувачами типи посилань, так і типи значень. C# дозволяє динамічно виділяти об’єкти і зберігати спрощені структури в стеку.

В C# особлива увага приділяється керуванню версіями для забезпечення сумісності програм і бібліотек при їх зміні. Питання управління версіями істотно вплинули на такі аспекти розробки C#, як роздільні модифікатори virtual та override, правила дозволу перевантаження методів і підтримка явного оголошення членів інтерфейсу.

В C# основним поняттями організаційної структури являються програми, простори імен, типи, елементи та збірки. В програмі оголошуються типи, ці типи

можна організувати в просторі імен. Прикладами типів є класи, структури та інтерфейси. До членів відносяться поля, методи, властивості і події.

При компіляції, програми на C# упаковуються в збірки. Збірка це файл, зазвичай з розширенням .exe або .dll, якщо вона реалізує додаток або бібліотеку відповідно.

Збірки зберігають виконуваний код програми в виді інструкцій проміжної мови (IL) і символьну інформацію в виді метаданих. Перед виконанням JIT-компілятор середовища CLR .NET перетворює код IL в збірці, в код, який залежить від процесора.

Збірка повністю описує саму себе і містить увесь код і метадані, тому в C# не використовуються директиви #include і файли заголовків [4].

Програми C# можна зберігати в декількох вихідних файлах. При компіляції програми C#, всі вихідні файли оброблюються разом, при цьому вони можуть вільно посилатись один на одного. По суті, це аналогічно тому, якби всі вихідні файли були об’єднані в один великий файл перед обробкою. В C# ніколи не використовуються випереджаючі оголошення, так як порядок оголошення, за рідкісними виключеннями, не грає ніякої ролі. В C# немає вимог оголошувати тільки один відкритий тип в одному вихідному файлі, а також ім’я вихідного файлу не зобов’язане співпадати з типом, яким оголошений в цьому файлі.

1.2.2 Особливості і переваги C#

C# одна з найбільш популярних мов програмування, була запущена Microsoft 20 років тому, як частина .NET Framework.

Синтаксис мови забирає важкості C++ і надає такі можливості як онульовані значення типів, перечислення, делегати, лямбда-вирази і прямий доступ до пам’яті. C# підтримує універсальні методи і типи, які підвищують безпечність типів і швидкодію. Ітератори дозволяють класам створювачів колекцій оголошувати спеціальну поведінку ітерацій, яку легко

використовувати в клієнтському коді. Вираження інтегрованої мови запитів (LINQ) роблять строго типізований запит першокласною конструкцією мови.

C# - об’єктно орієнтована мова, це означає що вона підтримує інкапсуляцію, наслідування і поліморфізм. Клас може унаслідуватись напряму від одного, єдиного батьківського класу, але може реалізовувати будь-яку кількість інтерфейсів. Методи, які заново оголошують віртуальні методи в батьківському класі, потребують вживання ключового слова override для попередження випадкового нового оголошення.

C# полегшує розробку програмних компонентів за допомогою деяких інноваційних конструкцій мови:

- Інкапсульовані сигнатури методів, які називають делегатами, включають сповіщення про безпечність типів;
- Властивості служать доступом до змінних закритих частин;
- Атрибути надають декларативні метадані про типи під час виконання;
- Лінійні документаційні коментарі XML;
- Інтегрована мова запитів (LINQ) надає вбудовані можливості запитів між різними джерелами даних.

Версія Core – це модульна крос платформна open source реалізація стандарту .NET загального призначення. Вона підтримує велику API з .NET Framework і включає середовище виконання, фреймворк, а також інструменти, підтримуючі різноманітні ОС і процесори. Реалізація знаходилась під управлінням напрацювань з ASP.NET Core, але потребувала більш сучасного підходу. Вона може використовуватись для різноманітних засобів, в хмарі і для вбудованих сценаріїв [5].

.NET Core надає ключову функціональність для реалізації потрібних особливостей додатку, а також можливість повторного використання коду незалежно від платформи. Підтримується Windows, Linux та macOS. Можна писати додатки та бібліотеки, які запускаються на всіх підтримуваних ОС без змін.

.NET Core один з небагатьох проектів, який знаходиться під управлінням .NET Foundation і доступний на GitHub. Використання .NET Core в якості open source проекту сприяє прозорому процесу розробки і активного розвитку з залученням спільноти.

Є два головних способи розгортання додатку: залежний від фреймворку і автономний. В першому випадку встановлюються тільки додаток і залежності. Додаток залежить від глобальної версії .NET Core. В другому випадку, версія .NET Core, яка використовувалась для збору додатку, розгортається разом з додатком і залежностями, і може виконуватись паралельно до інших версій [6].

.NET Core володіє модульністю тому, що випускається через NuGet в менших пакетах збірок. На відмінну від однієї великої збірки з усім вбудованим функціоналом, .NET Core доступний в вигляді менших функціонально-орієнтованих пакетів. Це призводить до більш глибокої моделі розробки, дозволяє оптимізувати додаток, включаючи тільки необхідні пакети NuGet. Переваги меншого поля покриття додатку відображаються в підвищеній безпеці, менш затратному обслуговуванню, високій швидкодії і в зниженні затрат за принципом плати тільки за те, чим користуєшся.

1.3 Аналіз вимог до тестування програмної системи

Розробка програмного забезпечення – це процес комплексний процес створення комп’ютерних програм. Процес, також відомий як цикл розробки програмного забезпечення, включає декілька фаз які надають методи для створення продуктів які відповідають технічній специфікації і вимогам користувачів.

Цикл розробки програмного забезпечення надає загальний стандарт який компанії можуть використовувати для побудови та покращення своїх продуктів. Він надає сталу структуру, дотримуючись якої, команди розробки отримають продукт високої якості. Ціллю процесу розробки є створення ефективних продуктів, враховуючи виділений бюджет і часові рамки.

Беручи до уваги тему магістерської роботи, доречно, з усіх етапів циклу розробки програмного забезпечення, виділити етап тестування. Фаза тестування перевіряє продукт на помилки і верифікує швидкодію, перед тим як продукт надходить кінцевим користувачам. На даному етапі, тестери перевіряють функціонал продукту, для того, щоб впевнитись що він відповідає технічній специфікації та вимогам користувачів.

Тестери використовують різні методи тестування для перевірки індивідуальних компонентів програмного забезпечення. Один з найефективніших методів перевірки базового функціоналу на розбіжність з вимогами є автоматизоване тестування.

В якості предметних областей, яких стосується дана магістерська робота можна виділити наступні:

- Інструменти роботи з текстом – адже, відповідно до теми та технічного завдання, моєю задачею є дослідити та побутувати автоматизоване тестування системи роботи з юридичними документами;
- Процес тестування програмного забезпечення, як частина життєвого циклу розробки програмного забезпечення.

1.4 Огляд та аналіз предметної області

Останні розробки у великомасштабних можливостях зберігання даних та пов’язаний з ними прогрес в аналізі юридичних даних створили безпрецедентні інструменти для виявлення закономірностей – таких як статистичні, семантичні, засновані на цитуваннях та тимчасова структура у великих сховищах юридичних даних. Ці розробки приводять до нового розуміння як взаємозв’язку між авторитетними юридичними текстами та поведінкою судів, законодавчих органів, адвокатів та інших юридичних професіоналів.

Юридична індустрія великою мірою була змінена можливостями AI – допомагати людині швидко аналізувати велику кількість даних та знаходити

релевантну інформацію. Без можливостей штучного інтелекту в огляді юридичних документів, юристи просто не змогли б ефективно розпоряджатись великими об’ємами даних, які на сьогодні зустрічаються під час eDiscovery процесу [7].

Окрім аналізу документів, як в юридичній так і в інших сферах, важлива можливість автоматизації документів. Конструктор документів (або автоматизація документів) – це програне забезпечення, яке дозволяє автоматизувати створення документів генеруючим їх з розумних шаблонів. Дана технологія робить можливим створювати навіть комплексні документи набагато швидше ніж вручну, або іншими методами. Також даний тип програмного забезпечення допомагає уникати помилок при формуванні документу [8].

Програмне забезпечення з автоматизації документів надає широкі можливості, набагато ширші ніж самі програми для роботи з документами [9]. Під час генерації документів користувач повинен надавати інформацію і приймати рішення які потрібні для формування кінцевого документу [10].

До переваг програмного забезпечення такого типу можна віднести:

- Продуктивність. Класична перевага програмного забезпечення конструкції документів [11];
- Якість. Забезпечуючи перевірені часом тексти та методи, системи забезпечують якість та мінімізують небажані метадані з необережно повторно використаного робочого продукту;
- Мала залежність від індивідуальних експертів. Працівники інформаційних технологій не можуть знаходитися одночасно у двох місцях, і не можуть працювати цілодобово. Системи складання документів можуть зробити доступними деякі аспекти експертизи, незалежно від експерта, а також консолідувати численні джерела знань [12];
- Задоволення роботою та збагачення. Життя юридичного працівника приносить більше задоволення, коли він продуктивний, виконує високоякісну роботу, позбавлений механічної праці та

може зосередитись на завданнях, що вимагають судження та творчості;

- Навчання та безперервна освіта. Системи можуть використовуватись для підготовки співробітників та неспеціалістів, які потім виходять із процесу володіючи більшою кількістю інформації;
- Маркетингові та рекрутингові переваги. Технологічна майстерність у ключових сферах роботи може стати потужним джерелом диференціації ринку – залучення як клієнтів, так і персоналу [13];
- Практика покращень. Сама активність автоматизації робочих практик може покращити їх і змусити інших людей задуматись над тим що вони роблять. Систематизація може викрити незбіжності або помилки які були непомічені в ранніх, ручних підходах [14].

Технології конструювання документів вже не тільки користуються популярністю серед професіоналів. Зараз, ці продукти використовуються простими користувачами [15]. Великі інтернаціональні компанії продають підписки на онлайн-експертні системи, що забезпечують складний аналіз без безпосередньої участі людини. Корпоративні відділи оснащують персонал такими системами. Суди та програми правової допомоги забезпечують розумні форми для непредставлених учасників судових процесів [16]. Автоматизація документів постійно набирає популярності в адвокатських конторах [17]. Присутні ознаки дуже швидкого зростання зацікавленістю даним типом програмного забезпечення, включаючи енергійну конкуренцію серед продуктів на ринку, багато кваліфікованих консультантів, підвищені очікування від клієнтів, претенденти на професію та величезні можливості для вдосконалення процесу в юридичній роботі [18] [19] .

1.5 Опис системи конструювання документів Litera Forte

Forte – це інструмент конструювання документів, який безпечно зменшує час, необхідний для створення першої чернетки до 85 відсотків. Він надає доступ до затверджених фірмою шаблонів, прецедентів та положень безпосередньо в Microsoft Word, проникає інформацію про клієнта та питання, а потім автоматично виконує завдання, пов'язані зі створенням високоякісних юридичних документів.

На фірми зростає тиск з боку клієнтів робити більше за меншу ціну, будь-яка неефективність у робочому процесі, що додає витрат або часу, є безпосередньо загрозою для збереження клієнтів. З огляду на невідкладність, юридичні фахівці часто переробляють документи та шаблони, а потім вручну знаходять і замінюють старі дані новими. Розглянемо як Forte усуває цей трудомісткий процес.

Використання Forte дозволяє впорядкувати повторювані завдання, пов'язані зі створенням юридичних документів, та заощадити до 85% часу на створенні початкових чернеток. Також, даний додаток робить можливим запобігання помилок та забезпечує незмінний зовнішній вигляд, коли документи виготовляються в стандартному вигляді та з оновленим вмістом. Також, будучи інструментом автоматизації конструювання документів, дозволяє прискорити процес складання за допомогою, власне, автоматизації, яка зберігає гнучкість для фірм з різноманітними та мінливими потребами в документах.

Можна виділити такі ключові можливості продукту:

- Шаблони. Можна створювати шаблони, прецеденти та пакети шаблонів і робити їх доступними для всіх або певних користувачів;
- Нумерація та зміст. Дозволяється визначати та маніпулювати існуючими нумераціями, стилями, і генерувати зміст в документі;
- Юриспруденція. Можливість застосування стилів фірми для генерації важливих юридичних документів;
- Doc ID. Управління Doc ID з гнучкими та динамічними вставками та можливість, при потребы, конвертувати Doc ID штампи [20].

Forte підтримує інтеграцію з багатьма сторонніми системами, серед них можна виділити наступні:

- iManage;
- NetDocument;
- eDocs;
- SharePoint;
- Worldox
- Microsoft Outlook;
- InterAction;
- Contact Ease;
- Salesforce;
- Microsoft Dynamics CRM .

1.6 Роль тестування у циклі розробки програмного забезпечення

Ціллю тестування програмного забезпечення є надання інформації про якість продукту. Тестування також може надати об’єктивну, незалежну оцінку продукту, для розуміння ризиків в реалізації програмного забезпечення [21]. Техніки тестування включають процеси виконання програми або компоненту з ціллю знаходження помилок або дефектів, і перевірки того, що продукт придатний для використання.

Процес тестування передбачає виконання програмного або системного компонента для оцінки однієї або кількох зацікавлених властивостей. В загальному, ці властивості вказують на те, наскільки тестований компонент чи система в цілому:

- Відповідає вимогам, якими керуються при дизайні та розробці;
- Правильно обробляє всі види вхідної інформації;
- Виконує свої функції протягом прийнятного проміжку часу;

- Може встановлюватись і запускатись в передбачуваних вимогами середовищах;
- Досягає результату зацікавлених сторін (замовників та користувачів).

Враховуючи те, що кількість можливих тестів, навіть для найпростіших компонентів є практично нескінченною, в тестуванні використовують деякі стратегії вибору тестів, доступних для виконання в різних ситуаціях, враховуючи виділений час та ресурси. Таким чином, процес тестування, зазвичай, передбачає виконання програми з метою знаходження проблем або дефектів. Тестування це ітеративний процес, коли один дефект виправлено, він може показати інші, глибші дефекти, або навіть створити нові [22].

Тестування можна розпочинати при існуванні навіть найменшої, виконуваної частини продукту. Загальний підхід до тестування зазвичай визначає коли і як проводиться тестування. Для прикладу, в фазовому процесі, більшість тестування проводиться після визначення системних вимог і реалізації системного продукту. В гнучкому підході, вимоги, програмування і тестування, зазвичай, виконуються паралельно [23].

1.7 Рівні тестування

Можна виділити щонайменше три рівні тестування: юніт тестування, інтеграційне, та системне тестування. Але, розробники також можуть включати четвертий рівень – приймальне тестування [24].

Найбільш базовим типом тестування є юніт, або компонентне тестування. Ціллю юніт тестування є перевірка кожної частини продукту в ізоляції, для впевненості в тому, що ізольований компонент відповідає вимогам і описаному функціоналу. Цей тип тестування виконується на ранішніх стадіях процесу розробки, і в багатьох випадках виконується розробниками перед тим як передати продукт для тестування.

Перевагою виявлення помилок в програмному забезпеченні на початкових етапах є мінімізація ризиків розробки, а також трата грошей та ресурсів при потребі повертатись назад для вирішення базових проблем.

Інтеграційне тестування націлюється на тестування різних частин програмного забезпечення в комбінації один з одним, для того щоб перевірити коректність роботи компонентів разом. Тестуючи компоненти разом, можна виявити проблеми взаємодії компонентів.

Існує багато шляхів тестування того, як різні компоненти системи функціонують як інтерфейси, тестери можу адаптувати методологію знизу-вверх, або зверху-вниз.

В тестуванні знизу-вверх, тестування базується на результатах юніт тестування, тестуючи комбінації компонентів вищих рівні в більш комплексних сценаріях.

Наступним рівнем є рівень системного тестування. Виходячи з імені, усі компоненти системи тестуються разом, для впевненості в тому, що загальний продукт відповідає вимогам.

Системне тестування є дуже важливим кроком, адже програмне забезпечення на цьому етапі я найбільш близьким до того, яким його отримають кінцеві користувачі. Системне тестування дозволяє тестерам впевнитись що продукт відповідає бізнес вимогам, а також що він коректно працює в відповідному середовищі [25]. Цей тип тестування зазвичай виконується спеціалізованою командою тестерів.

Останнім рівнем є тестування прийняття – це рівень тестування коли продукту дають, або не дають, зелене світло. Ціллю даного типу тестування є перевірка того, що система відповідає вимогам кінцевих користувачів, і чи готова вона виходити на ринок.

Команда тестерів, на даному етапі, використовує різноманітні методи тестування, такі як готові сценарії для того щоб визначити чи можна покращити продукт.

Обсяг тестування прийняття варіюється з пошуку простих граматичних помилок, до виявлення дефектів які можуть значною мірою впливати на програмне забезпечення [26].

1.8 Види тестування

Умовно, усі існуючі види тестування програмного забезпечення, залежно від мети та цілей, можна розділити на наступні групи:

- Функціональне тестування (Functional testing);
- Нефункціональне тестування (Non-functional testing);
- Тестування пов’язане зі змінами (Regression testing).

1.8.1 Функціональне тестування

Функціональне тестування базується на функціональних особливостях, а також на взаємодії з іншими системами, і може бути присутнє на усіх рівнях тестування: юніт тестування, інтеграційного тестування, системного тестування та тестування приймання.

Нижче наведені найпопулярніші види функціонального тестування:

- Тестування безпеки (Security testing);
- Функціональне тестування (Functional testing);
- Тестування взаємодії (Interoperability testing) [27];

Проводити функціональне тестування можна відносно двох аспектів:

- Бізнес-процесів;
- Вимог.

Тестування відносно бізнес-процесів використовує самі бізнес-процеси, які описують сценарії використання системи. Тестові сценарії, в такому

випадку, зазвичай ґрунтуються на прикладах використання системи кінцевими користувачами.

Тестування вимог бере за основу створення тестових випадків функціональні вимоги до системи [28]. У цьому випадку зазвичай створюється список тестових сценаріїв, проводиться пріорітизація тестового функціоналу, та відповідно цього, пріорітизуються самі тестові випадки. Це дозволяє при тестуванні не пропускати найбільш важливі функціональні точки.

Основною перевагою функціонального тестування є те, що воно, фактично, імітує використання системи кінцевим користувачем.

До недоліків можна віднести можливість упущення логічних помилок в програмному забезпеченні, та надмірне тестування.

Автоматизація функціонального тестування є досить поширеною.

Тестування безпеки (Security testing) – це стратегія тестування, яку використовують для перевірки безпеки системи та аналізу ризиків, пов’язаних із забезпеченням підходу до захисту програмного забезпечення від атак хакерів, вірусів та несанкціонованого доступу до конфіденційних даних [29].

Стратегія безпеки програмного забезпечення ґрунтується на трьох основних принципах: цілісність, доступність та конфіденційність.

Визначаючи поняття цілісності можна виділити наступне:

- Довіра – Очікується, що зміна ресурсу відбувається загальноприйнятим способом, відповідною групою користувачів;
- Пошкодження і відновлення – в разі неправильної зміни або пошкодженні даних, слід визначити важливість відновлення уражених даних.

Доступність являє собою вимоги про доступність ресурсів авторизованим користувачам, внутрішнім об’єктам та пристроям. Зазвичай, чим більш критичний ресурс, тим менший рівень доступності.

Конфіденційність – це приховування певної інформації та ресурсів. Під конфіденційністю можна розуміти обмеження доступу до ресурсів деякій

категорії користувачів, або умови, за яких авторизований користувач може отримати доступ до відповідних ресурсів [30].

Тестування взаємодії (Interoperability testing) – це функціональне тестування, яке дозволяє перевірити здатність продукту взаємодіяти з одним, або декількома компонентами або системами і включає в себе тестування сумісності (compatibility testing) і інтеграційне тестування (integration testing).

1.8.2 Нефункціональне тестування

Нефункціональне тестування це тип тестування, необхідний для визначення характеристик продукту, які можна вимірювати різними величинами. В загальному, це тестування того, як працює система.

До основних видів нефункціонального тестування можна віднести:

- Тестування продуктивності (Performance testing);
- Тестування встановлення (Installation testing);
- Тестування зручності користування (Usability testing);
- Тестування на відмову і відновлення (Failover and Recovery testing);
- Конфігураційне тестування (Configuration Testing or Portability testing);
- Тестування локалізації (Localization testing).

Тестування навантаження дозволяє визначити масштабованість додатку під навантаженням, при цьому відбувається наступне: вимір часу виконання операцій, визначається кількість користувачів, що одночасно працюють з додатком, визначаються межі прийнятної продуктивності при збільшенні навантаження, досліджується продуктивність при граничних, стресових та високих навантаженнях [31].

Тестування встановлення спрямоване на перевірку успішності інсталяції та її налаштувань, також оновлення та видалення програмного забезпечення.

Тестування зручності користування дає оцінку рівня зручності використання програми за наступними критеріями:

- Ефективність та продуктивність;
- Правильність;
- Активізація в пам’яті;
- Емоційна реакція.

Конфігураційне тестування це підвид тестування продуктивності. Під час такого типу тестування тестується ефект впливу на продуктивність змін у конфігурації.

Тестування локалізації перевіряє правильність перекладу елементів інтерфейсу користувача, системних повідомлень і помилок.

1.8.3 Тестування змін

Тестування змін, або регресивне тестування – це вид тестування, метою якого є перевірка того, що попередньо розроблений функціонал працює відповідно до вимог, після впровадження змін. Зміни які можуть потребувати регресивного тестування включають: виправлення помилок, покращення програмного забезпечення, конфігураційні зміни. Так як набір регресивних тестів постійно зростає, зазвичай впроваджується автоматизоване тестування.

Існують різні підходи до регресивного тестування. Тестування всього функціоналу перевіряє всі тестові випадки. Воно може бути трудомістким, але завершивши його можна переконатись що нових помилок немає [32].

Регресивна тестова вибірка – це техніка, під час якої вибирають частину з тестів тестового набору.

Пріоритизація тестів спрямована на підвищення ефективності використовуваних тестів. Тобто тести з вищим пріоритетом виконуються першими, після них виконуються тести з нижчим пріоритетом [33].

Регресивне тестування виконується тоді, коли продукт зазнає змін, або виправляється попередньо знайдена помилка. Регресивне тестування можна виконувати декількома шляхами, коли використовується шлях тестування

всього функціоналу, він дає впевненість в тому що зміни в програмному продукті не вплинули на попередньо робочий функціонал.

Регресивні тести можна розділити на функціональні та юніт тести [34]. Функціональні тести тестують продукт повністю, з різними вхідними даними. Юніт тести тестують індивідуальні функції та методи. Як і функціональні так і юніт методи зазвичай автоматизуються і не є частиною основного програмного продукту.

Сам по собі термін “регресивне тестування”, в залежності від контексту використання може мати різний зміст. Сем Канер описав 3 основних типи регресивного тестування:

- Регресія багів (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена;
- Регресія старих багів (Old bugs regression) – спроба довести, що недавня зміна коду чи даних зламала виправлення старих помилок, тобто старі баги стали знову відтворюватися;
- Регресія побічного ефекту (Side effect regression) – спроба довести, що недавня зміна коду чи даних зламала інші частини продукту .

Також до підтипу регресивного тестування можна віднести димове тестування, воно застосовується для поверхневої перевірки всіх модулів програми і швидкого виявлення критичних помилок або дефектів [35].

1.9 Автоматизоване тестування

Автоматизація тестування – це використання програмних продуктів, окремих від програмного продукту який тестується, для контролю виконання тестів і порівняння актуальних результатів з очікуваними. Автоматизоване тестування може автоматизувати деякі часто повторювані, але важливі, завдання в формалізованому тестовому процесі, або виконувати додаткові перевірки, які

виконати вручну було б досить складно [36]. Автоматизоване тестування це критичний процес для постійного виходу на ринок та постійного тестування.

Існує багато підходів до автоматизації тестування, нижче наведені основні, які широко використовуються:

- Тестування інтерфейсі користувача. Тестовий фреймворк генерує події інтерфейсу користувача, такі як натиснення клавіш, кліки миші, і спостерігає за змінами спричиненими цими діями, для впевненості в тому, що продукт працює коректно;
- API кероване тестування. Тестовий фреймворк який використовує інтерфейс програмування для перевірки коректності роботи продукту який тестується. Зазвичай, API тестування повністю оминає інтерфейс користувача. Цей тип тестування також може тестувати публічні інтерфейси до класів, модулів та бібліотек з різними аргументами для упевненості в коректності роботи.

Одним з шляхів генерації тестів автоматичним шляхом є тестування базоване на моделі (model-based), через використання моделі системи для створення тестових випадків. В деяких випадках, model-based підхід дозволяє не технічним користувачам створювати автоматизовані бізнес тестові випадки на простій англійській мові, без використання програмування [37].

Що автоматизувати, коли автоматизувати, або навіть чи взагалі потрібно автоматизувати є ключовими рішеннями які команда тестерів (або розробників) повинна прийняти [38]. Існує п’ять основних факторів які потрібно оцінити, приймаючи рішення в автоматизації:

1. Система яка тестується (SUT);
2. Типи і кількість тестових випадків;
3. Тестові інструменти;
4. Людські і організаційні питання;
5. Наскрізні фактори.

Найбільш частішими факторами зазвичай є питання про потребу в регресивному тестуванні, економічні фактори, та зрілість системи яку тестують [39].

Стрімко зростаючим трендом в розробці програмного забезпечення є використання юніт-тест фреймворків, таких як xUnit (також Junit та NUnit) для виконання юніт тестів з метою визначення того, що різноманітні секції коду працюють правильно в різних умовах [40]. Тестові випадки описують те, що потрібно перевірити при виконанні програми, щоб впевнитись в правильності її роботи.

Автоматизоване тестування, яке здебільшого використовує юніт тестування, є ключовою властивістю XP методології і гнучкої методології розробки програмного забезпечення, де це також називають тест-керована розробка (TDD) [41]. Юніт тести можуть бути написані для того, щоб описати функціонал ще до того як буде написаний код цього функціоналу. Однак, ці тести еволюціонують з часом розробки продукту, знаходяться проблеми та міняється код. Тільки тоді, коли всі тести, для всіх потрібних модулів проходять успішно – код вважається завершеним [42]. Прихильники такого підходу вважають що з його використанням отримується більш надійний продукт з меншою затратою коштів, ніж той який би тестувався вручну. Він вважається більш надійним тому, що покриття коду краще, і тести запускаються постійно, на відмінну від одного разу, в кінці циклу розробки (за моделлю водоспаду). Розробник знаходить дефект зразу при введенні змін, коли його найдешевше виправити. Також, рефакторинг коду безпечніший коли використовуються юніт тести [43].

Деякі завдання тестування (такі як постійне регресивне тестування) можуть стати важкими і займати багато часу, виконуючись вручну. Також, ручний підхід не завжди може бути ефективним в знаходженні дефектів специфічного типу [44]. Автоматизація тестування дає можливість виконувати ці типи тестування ефективніше.

Після того як автоматизовані тести розроблені, вони можуть запускатись часто і швидко. Досить часто це може бути фінансово ефективним методом регресивного тестування програмних продуктів які мають довгий цикл підтримки. Навіть найменші виправлення з часом життя програмного продукту можуть негативно вплинути на функціонал який працював раніше.

В той час як повторне використання автоматизованих тестів цінується компаніями з розробки програмного забезпечення, ця властивість може бути і недоліком. Вона призводить до так званого парадоксу пестицидів (Pesticide Paradox), коли часто виконувані скрипти не знаходять помилок які виходять за межі їхніх фреймворків. В таких випадках, ручне тестування може стати кращою інвестицією. Ця неоднозначність ще раз приводить до висновку, що рішення про автоматизоване тестування слід приймати індивідуально, враховуючи вимоги та особливості проекту.

Засоби автоматизації можуть бути дорогими і зазвичай використовуються разом з ручним тестуванням. Автоматизацію тестування можна зробити економічно вигідною в довготривалій перспективі, особливо при повторному використанні в регресивному тестуванні. Хорошим кандидатом для автоматизованого тестування є тест на загальний потік програми, оскільки він повинен виконуватись кожен раз, коли в додатку робляться вдосконалення. Автоматизація тестування зменшує зусилля пов’язані з ручним тестуванням. Потрібні ручні зусилля для розробки та підтримки автоматичних перевірок, а також для перегляду результатів тестових запусків.

При автоматизованому тестуванні інженер або особа, яка займається забезпеченням якості, повинна мати знання в програмуванні, оскільки тестові методи написанні у формі вихідного коду, який під час запуску видає результати відповідно до тверджень, що входять до його складу. Деякі засоби автоматизації дозволяють створювати тести за ключовими словами замість кодування, що не вимагає знань мов програмування [45].

API тестування також широко використовується тестерами тому, що дає можливість перевірити вимоги незалежно від імплементації інтерфейсу

користувача. Це включає пряме тестування API як частину інтеграційного тестування, для підтвердження вимог функціоналу, надійності, швидкодії та захисту. API тестування вважається критичним коли API сервери є прямими інтерфейсами логіки програмного продукту.

Багато інструментів автоматизації дають можливість запису і програвання, що дозволяє користувачеві інтерактивно записувати дії і програвати їх будь-яку кількість разів, порівнюючи очікуваний результат з актуальним. Перевагою даного підходу є те, що він майже не вимагає знань з розробки програмного забезпечення. Даний підхід можна застосувати до будь-якого програмного продукту з графічним інтерфейсом користувача. Але, зміни функціоналу призводить до значних проблем в надійності і підтримуванні тестів даного типу. Зміна назви кнопки або зміна її розташування може призвести до потреби перезапису тестового випадку. Запис і програвання також, зазвичай, додає непотрібні дії або неправильні записи деяких дій.

Варіант цього типу інструментів призначений для тестування веб-сайтів. Тут інтерфейс – це веб-сторінка. Однак такий фреймворк використовує зовсім інші методи, оскільки він відображає HTML і прослуховує DOM події замість подій операційної системи. Зазвичай для цього використовуються рішення на основі веб-драйвера Selenium.

Інший варіант цього типу засобів автоматизації тестування – це тестування мобільних додатків. Це дуже корисно, враховуючи кількість різних розмірів, роздільну здатність та операційні системи, що використовуються на мобільних пристроях Для цього варіанту використовується фреймворк для створення екземплярів дій на мобільному пристрої та збору результатів дій [46].

Стратегія вибору кількості автоматизованих тестів називається пірамідою автоматизації тестування.

Рисунок 1.7 – Піраміда автоматизації тестування (Mike Cohn)

Дана стратегія пропонує писати три типи тестів, з різною деталізацією. Чим вищий рівень, тим меншу кількість тестів даного типу потрібно писати.

Розглянемо кожен з рівнів більш детальніше:

- Як міцна основа, юніт тестування забезпечує надійність програмного продукту. Тестування окремих частин коду дозволяє легко писати та запускати тести;
- Сервісний рівень стосується тестування сервісів додатку окремо від користувацького інтерфейсу;
- На верхньому рівні знаходиться тестування інтерфейсу користувача, яке має менше тестів через різні атрибути, які роблять його складнішим для запуску, наприклад, крихкість тестів, коли невелика зміна в інтерфейсі користувача може порушити багато тестів і додає зусиль в обслуговуванні.

Фреймворк автоматизованого тестування це інтегрована система яка задає набір правил автоматизації специфічного продукту. Ця система інтегрує функціональні бібліотеки, структури даних, деталі об’єктів і різні багаторазові моделі. Ці компоненти є малими будівельними блоками які потрібно скласти щоб відобразити бізнес процес. Фреймворк надає базу для автоматизації і зменшує зусилля, потрібні для автоматизації.

Основною перевагою фреймворків, та інших інструментів які підтримують автоматизацію є низька ціна підтримки. Якщо тестовий випадок

зазнає змін – тільки файл цього тесту потребує змін, драйвер скрипти та скрипти запуску залишаються незмінними. В ідеалі, потреби оновлення скриптів в разі змін додатку бути не повинно.

Вибір правильної техніки скриптингу, або вибір правильного фреймворку допомагає зберігати дешеву підтримку. Ціна підтримки обумовлюється розробкою та зусиллями підтримки. Підхід до написання автоматизації має вплив на ціну.

Зазвичай використовуються різні підходи до автоматизації:

1. Лінійний;
2. Структурований;
3. Керований даними;
4. Керований ключовими словами;
5. Гібридний;
6. Гнучкий.

Фреймворк автоматизації тестування відповідає за наступні речі:

1. Оголошення формату в якому виражаються очікування;
2. Створення механізму керування додатком;
3. Виконання тестів;
4. Звітування про результати.

2. СПЕЦІАЛЬНА ЧАСТИНА

2.1 Розгортання тестового середовища

Тестовий десктопний додаток не потребує додаткових конфігурацій та встановлюється стандартний інсталятором .msi, розглянемо процес розгортання локального сервера для запуску Swagger UI.

Для перегляду локального OpenAPI файлу потрібно мати запущений HTTP сервер. Це потрібно для задоволення вимог безепеки Swagger UI, тому, що CORS (cross-origin resource sharing) буде блокувати запуск Swagger UI.

Можна створити локальний веб сервер через Python SimpleHTTPServer модуль. Для цього потрібно виконати ряд кроків.

Спершу потрібно завантажити та встановити Python.

Рисунок 2.1 – Встановлення Python

Після встановлення потрібно перезавантажити командний рядок та відкрити в ньому директорію dist Swagger UI. Після цього слід виконати запуск сервера командою python -m http.server після чого сервер буде запущено.

За замовчуванням, Swagger UI використовує Petstore OpenAPI документ конфігурації в url параметрі файлу index.html. Цей параметр потрібно замінити своїм локальним файлом.

Свій OpenAPI файл потрібно помістити в директорію dist. Після чого структура буде виглядати наступним чином (рисунок 2.2).

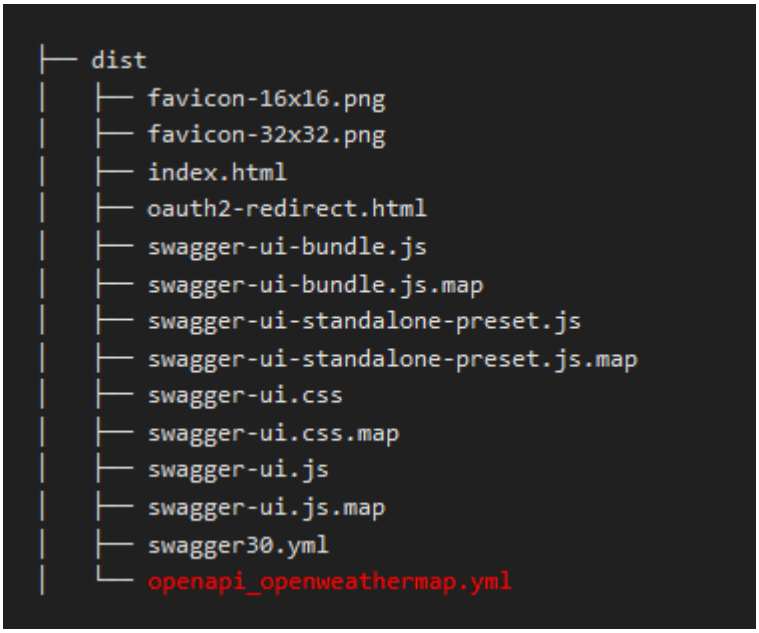

Рисунок 2.2 – Файлова структура директорії dist

Після цього потрібно відкрити index.html файл і замінити значення url параметра своїм відносним шляхом до YAML файлу, після чого необхідно зберегти зміни в файлі.

Переглядаючи файл index.html локально в браузері, використовуючи Python сервер, після завершення конфігурації, Swagger UI повинен відображати OpenAPI.

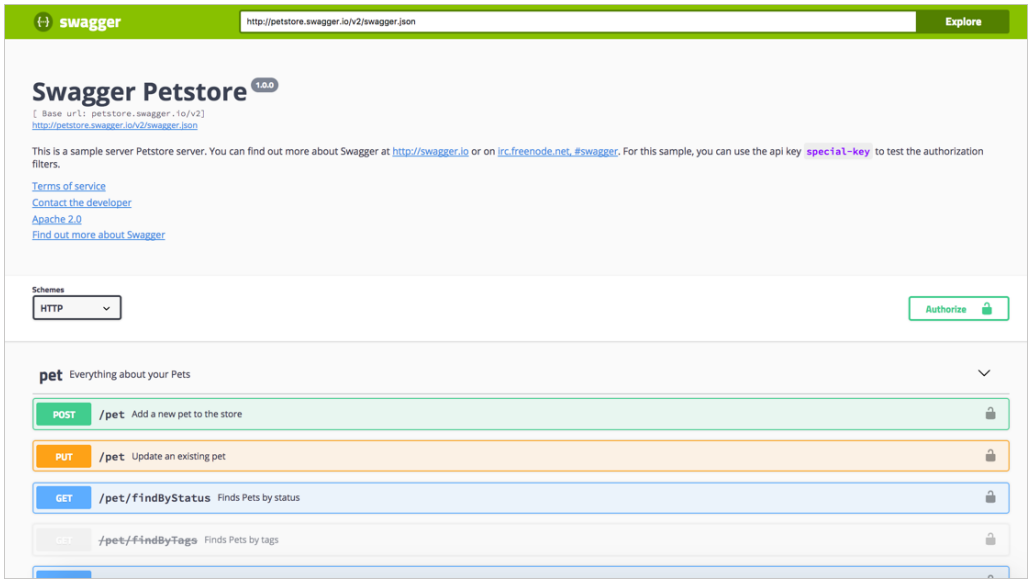

Рисунок 2.3 – Зовнішній вигляд Swagger UI

Після успішного встановлення можна виконувати API запити до кінцевих точок, використовуючи зручний веб інтерфейс у браузері.

2.2 Опис та реалізація основних тестових випадків

Після встановлення десктопного додатку, доцільним буде проведення так званого димового тестування. Даний тип тестування спрямований на перевірку базового функціоналу програмного продукту.

У таблиці 2.1 наведено опис тестового випадку виконання базового пошуку.

Таблиця 2.1 – Тестовий випадок базового пошуку

Назва тесту: Виконання базового пошуку	ID: TC-001
Опис тесту: Перевірити, що при пошуку за відповідним ключовим словом повертається правильна кількість результатів	
Кроки	Очікуваний результат
1. Відкрийте Microsoft Word	–
2. Натисніть на вкладку Forte	–

3. Натисніть кнопку Show Task Pane	
4. В рядку пошуку введіть ключове слово master	
5. Натисніть Enter	Система повертає 11 результатів пошуку. Кожен результат містить ключове слово master.

У таблиці 2.2 наведено опис тестового випадку прогресивного пошуку.

Таблиця 2.2 – Тестовий випадок прогресивного пошуку

Назва тесту: Виконання прогресивного пошуку	ID: TC-002
Опис тесту: Перевірити, що при пошуку за відповідним ключовим словом та відповідними параметрами повертається правильна кількість результатів	
Кроки	Очікуваний результат
1. Відкрийте Microsoft Word	–
2. Натисніть на вкладку Forte	–
3. Натисніть кнопку Show Task Pane	–
4. Натисніть кнопку Advanced Search	–
5. В полі пошуку введіть ключове слово master	–
6. В випадаючому меню Content Type виберіть Segments	–
7. Натисніть OK	Система повертає 1 результат пошуку. Результат пошуку містить ключове слово master.

У таблиці 2.3 наведено опис тестового випадку створення нової папки в системі.

Таблиця 2.3 – Створення нової папки в системі

Назва тесту: Створення нової папки	ID: TC-003
Опис тесту: Перевірити, що система коректно створює нову папку	
Кроки	Очікуваний результат
1. Відкрийте Microsoft Word	–
2. Натисніть на вкладку Forte	–
3. Натисніть кнопку Show Task Pane	–
4. Виберіть будь-яку папку та натисніть на неї правою кнопкою миші	–
5. В контекстному меню виберіть Create New Folder	Відкривається вікно створення нової папки.
6. В полі Display Name введіть назву папки.	–
7. В полі Help Text введіть допоміжний текст папки	–
8. Натисніть OK	Нова папка відображається в дереві папок, та має коректну назву, введену в кроці 6.

У таблиці 2.4 наведено опис тестового випадку видалення папки з системи.

Таблиця 2.4 – Видалення папки з системи

Назва тесту: Видалення папки	ID: TC-004
Опис тесту: Перевірити, що система коректно створює нову папку	
Кроки	Очікуваний результат
1. Відкрийте Microsoft Word	–

2. Натисніть на вкладку Forte	–
3. Натисніть кнопку Show Task Pane	–
4. Виберіть будь-яку папку та натисніть на неї правою кнопкою миші	–
5. В контекстному меню виберіть Delete	Відкривається вікно підтвердження видалення.
7. В новому вікні натисніть Yes	–
8. Відкрийте список папок	Видалена папка не відображається в списку папок системи.

Для впровадження базової Rest API автоматизації не обов’язково потрібно мати готовий набір тестів. Початкові тести можна написати спираючись на опис кінцевих точок API OpenAPI. Swagger UI надає зручний інтерфейс, використовуючи який можна зрозуміти які дані на вході очікує та чи інша кінцева точка API, та які дані вона повинна повертати після виконання запиту.

2.2.1 Використання TestStack.White

TestStack.White – це фреймворк для автоматизації додатків, які базуються на Wind32, WinForms, WPF, Silverlight та SWT (Java) платформах. Даний фреймворк базується на .NET і не вимагає використання конкретних мов програмування. Рішення автоматизації які використовують White можуть бути написані на будь якій .NET мові, та використовуючи будь-який набір інструментів. White надає постійне об’єктно орієнтоване API, приховуючи складність бібліотеки Microsoft UIAutomation.

Керовані UI додатки мають механізм ідентифікації контролерів за їхнім іменем. Ці імена доступні для пошуку використовуючи UIAutomation API. Властивість імені відображається як AutomationId, при використанні UIA API.

В вікні, будь який UIItem може бути ідентифікований за декількома критеріями.

AutomationId – програмований ідентифікатор який вказується розробником компоненту. В WinForm і WPF цей тип є ім'ям яке має елемент. Даної властивості немає в SWT та Win2 додатках:

```
     SearchCriteria searchCriteria =
SearchCriteria.ByAutomationId("btnOK");
     Button button = window.Get<Button>("btnOK");
//default search mechanism is by automation id
     button = window.Get<Button>(searchCriteria); // is
same as above
```

Для un-managed додатків:

```
     Button button = window.Get<Button>("OK"); //default
search mechanism is by UIAutomation name
     button =
window.Get<Button>(SearchCriteria.ByText("OK")); // same
as above
     UIItem тип (наприклад Button, ComboBox):
     Button button = window.Get<Button>("btnOK");
//<Button> acts as criteria as well as the return type
     button = (Button)
window.Get(SearchCriteria.ByAutomationId("btnOK").AndCon
trolType(typeof(Button))); // same as above
```

Для отримання вікон, відкритих в додатку, використовується наступний код:

```
     List<Window> windows = application.GetWindows();
//Returns a list of all main windows in belonging to an
application. It doesn't return modal windows.
     Window mainWindow = application.GetWindow("main");
//Returns a window with provided title.
```

Використовуючи TestStack.White, можна здійснювати спеціальні операції над вікнами. Прикріплена клавіатура і миш надає функціонал доступний для виконання на стандартній клавіатурі та миші.

Кожне вікно може мати один видимий Tooltip об’єкт. Коли миша фокусується на даному об’єкті – відображається його повідомлення. Фреймворк може перехопити це повідомлення, витягуючи відповідну властивість об’єкту, без потреби фокусу на ньому.

Також об’єкт вікна імплементує IDisposable інтерфейс, для можливості використання в using блоці. Також можна перевіряти чи закрите вікно. Коли викликається window.Close(), фреймворк намагається закривати вікно використовуючи WindowPattern який надається UIAutomation. Якщо даний патерн не імплементований, закриття вікна буде виконане через TitleBar об’єкт.

Вікно може знаходитись в трьох станах відображення: maximized, minimized та restored. Властивість DisplayState вікна дає доступ до цих станів. Core.UIItems.WindowItems.DisplayState enum може бути використаний для встановлення стану вікна. Для того щоб отримати інформацію про положення вікна, можна звернутись до властивості IsCurrentlyActive.

2.2.2 Створення тестового фреймворку.

Створення тестового фреймворку відбувалося із використанням принципу ScreenObjects що базується на PageObjects, а також із застосуванням інструменту TestStack.White.

Screen Object дизайн паттерн є шаром абстракції який відображає інтерфейс що дозволяє користувачу керувати елементами додатку або перевіряти стан сторінки або вікна.

Переваги ScreenObject та PageObject підходу:

- Розділення між логікою роботи та представлення;
- Зменшення дублювання коду для пошуку елементів керування застосунком;

- При змінах в інтерфейсі, потрібно лише змінити код в ScreenObject або PageObject класах, код тестів залишиться незмінним.

PageObject та ScreenObject підходи є класичними прикладами інкапсуляції – вони приховують деталі UI структури від інших компонентів, в даному випадку від тестів. Як з будь-якою інкапсуляцією, це веде до двох основних переваг.

Збираючи логіку яка маніпулює UI елементами в одному місці, її можна модифікувати без впливу на інші компоненти в системі.

Наступною перевагою є те що тестовий код стає більш зрозумілим, тому, що логіка полягає в меті тесту і не містить лишніх деталей з маніпуляції UI елементів.

ScreenObjects та PageObjects на даний час є стандартними підходами, які використовують при десктопній та веб автоматизації. Навіть якщо фреймворк не використовує PageObjects або ScreenObjects підходи повністю, ідеєю більшості фреймворків з автоматизації є розділення елементів інтерфейсу і тестів які ці елементи перевіряють.

Проаналізувавши тестований додаток, я проектував наступну структуру фреймворку.

Збірка складається з трьох наступних проектів: Forte.Infrastructure, Forte.Tests та Forte.UIControls.

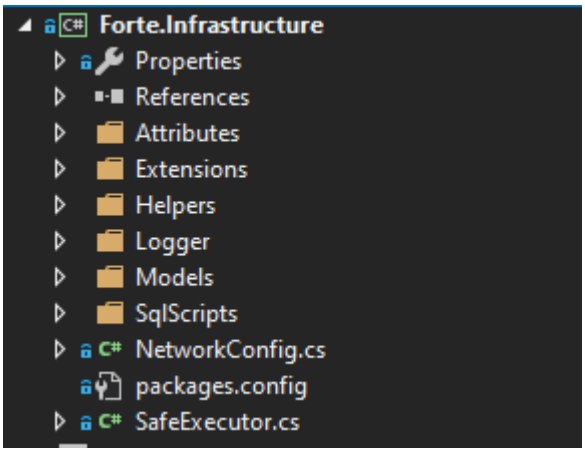

Рисунок 2.4 – Структура проекту Infrastructure

Проект Forte.Infrastrucutre містить спеціальні атрибути, методи розширення, допоміжні методи, класи конфігурації та моделі даних, а також SQL скрипти.

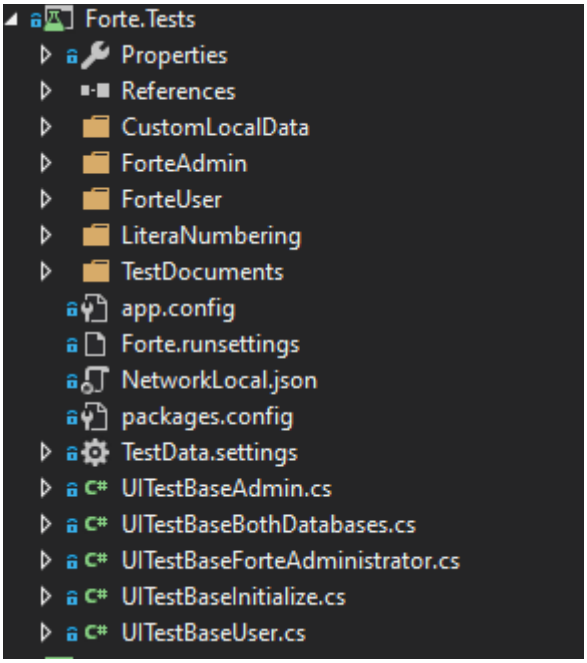

Рисунок 2.5 – Структура проекту Tests

Проект Forte.Test містить класи ініціалізації тестів і самі тестові класи. Також, в цьому проекті знаходяться тестові бази даних та тестові документи. Дані, які часто використовуються, зберігаються в файлі TestData.settings. Працювати з .settings файлом досить просто. Відриваючи файл такого типу в Visual Studio, можна побачити таблицю з наступними колонками: Name, Type, Scope та Value. Для створення .settings змінної потрібно дати їй ім’я в колонці Name, вибрати відповідний тип даних в колонці Type та задати бажане значення змінної в колонці Value. Після цього, для отримання значення певної змінної в коді, потрібно задати назву .settings файлу, та через синтаксис звернення до властивостей, звернутись до відповідної змінної за її ім’ям.

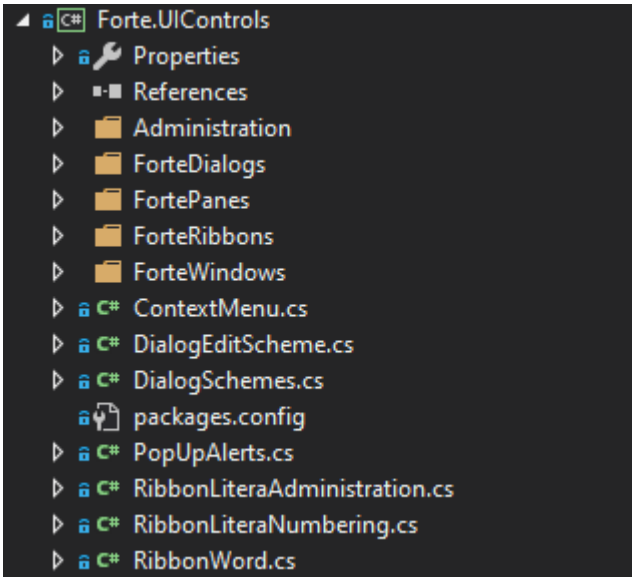

Рисунок 2.6 – Структура проекту UIControls

Проект Forte.UIControls містить класи які описують елементи інтерфейсу користувача та можливі дії над ними.

З огляду на сплановану структуру фреймворку, потрібно розділити Assemlby та Class ініціалізацію та очищення.

Код класу Assembly ініціалізації та очищення виглядає наступним чином:

```
[TestClass]
    public class UITestBaseInitialize
    {
        public static Exception LastException {
get; private set; }

        [AssemblyInitialize]
        public static void
TestRunStartUp(TestContext testContext)
        {
            ProcessHelper.KillWordProcesses();

AppDomain.CurrentDomain.FirstChanceException += (s, e)
=> ForteLogger.Exception = e.Exception;
        }
        [AssemblyCleanup]
        public static void AssemblyCleanup()
        {
            if
(SetupNetworkDatabaseAttribute.IsNetworkDatabaseCreated)
```

```
DataBaseHelper.DropNetworkDatabase();
            }
        }
```

В тестовому фреймворку MS Test використовується набір атрибутів для керуванням життєвим циклом тесту.

До атрибутів MS Test відносяться:

- [TestClass];
- [AssemblyInitialize];
- [ClassInitialize];
- [TestInitialize];
- [AssemblyCleanup];
- [ClassCleanup];
- [TestCleanup];
- [TestMethod];

Атрибут [TestClass] є обов’язковим і помічає клас який містить в собі MS Test юніт тести. [AssemblyInitialize] виконується один раз, перед тестовим запуском, не є обов’язковим. [ClassInitialize] виконується один раз перед виконанням класу, не є обов’язковим. [TestInitialize] виконується перед кожним тестом не є обов’язковим. [AssemblyCleanup] виконується один раз після виконання тестового запуску, не є обов’язковим. [ClassCleanup] виконується один раз, після виконання всіх тестів в класі, не є обов’язковим. Також, даний атрибут не гарантує виконання методу, поміченого ним, моментально після завершення усіх тестів в класі. [TestCleanup] виконується після закінчення кожного тесту, не є обов’язковим. [TestMethod] позначає тестовий юніт тест метод, є обов’язковим.

Використовуючи ці атрибути, можна сформувати клас ініціалізації та очищення тестових станів.

Для автоматизації API я використав юніт тест фреймворк NUnit. Так як і MS Test, NUnit використовує набір атрибутів для керування життєвим циклом тесту.

До основних атрибутів NUnit відносять:

- [TestFixture];
- [OneTimeSetUp];
- [SetUp];
- [Test];
- [TearDown];
- [OneTimeTearDown];
- [SetUpFixture];

Атрибут [TestFixrure] позначає клас який буде містити NUnit тести, він є обов'язковим. [OneTimeSetUp] виконується один раз для всього тестового класу, не є обов'язковим. [SetUp] виконується перед кожним тестом, не є обов'язковим. [Test] позначає тестовий метод, для виконання тесту є обов'язковим. [TearDown] виконується після кожного тесту, не є обов'язковим. [OneTimeTearDown] виконується один раз, після виконання усіх тестів в класі, не є обов'язковим. Даний атрибут не гарантує моментального виконання методу. [SetUpFixture] позначає клас, який може містити [SetUp] та [TearDown] атрибути, які будуть виконуватись для всієї збірки.

До основних відмінностей MSTest та NUnit можна віднести наступне:

- NUnint має атрибут [TestCase], який дозволяє реалізовувати параметризовані тести;
- NUnit набагато швидший за MSTest;
- NUnit може виконувати тести в 32 та 64 бітностях;
- NUnit дозволяє абстрактним класам бути тестовими наборами, даючи можливість наслідувати тестові набори;
- NUnit має Assert.Throw API, яке дозволяє тестувати виключні ситуації на конкретних рядках коду.

Структура фреймворку для автоматизації тестування API виглядає наступним чином:

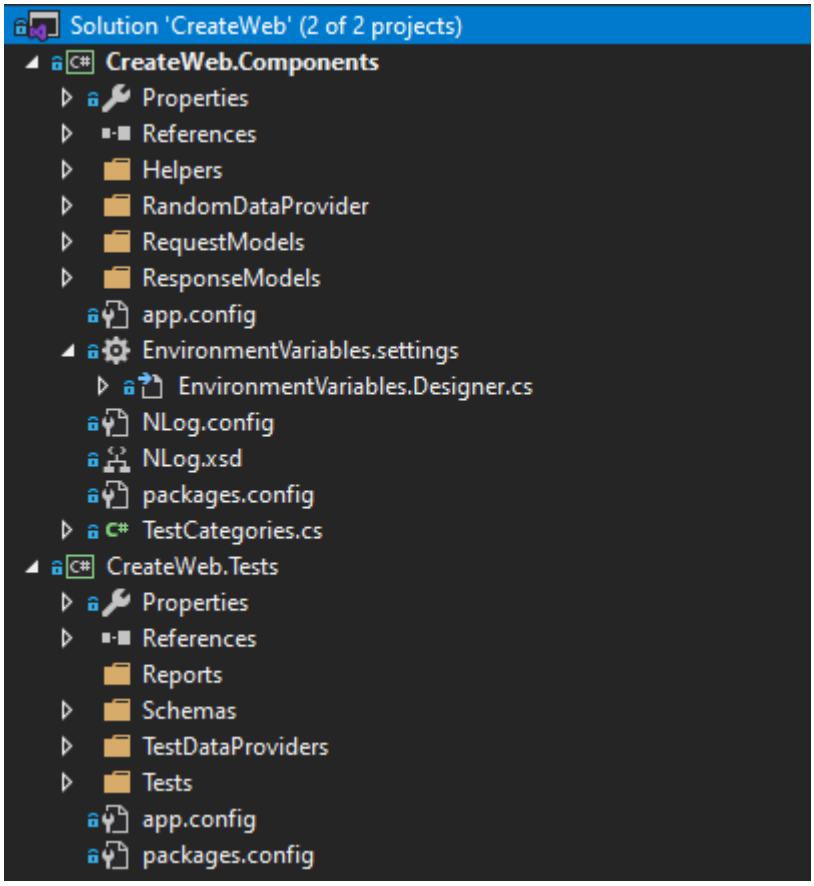

Рисунок 2.7 – Структура API фреймворку

Збірка складається з двох проектів: Components та Tests. Components проект містить допоміжні методи, та моделі даних для виконання API запитів. Tests проект містить тести та допоміжний код для виконання тестів.

Зазвичай, більшість комплексних фреймвокрів автоматизації містять як мінімум один клас який виконується перед запуском тестових класів. Призначенням цих класів є створення потрібного стану додатку, чи налаштування додаткових інструментів.

Як правило, один клас створює загальний, спільний для усіх тестів, стан додатку. Інший клас налаштовує логування, та інструменти звітування. Одним з найпопулярніших, на даний час, інструментів звітування є Extent Reports.

Extent Reports – це бібліотека звітування з відкритим кодом, яку зазвичай використовують для автоматизації тестів. Вона легко інтегрується з більшістю тестових фрейморків, таких як: JUnit, NUnit, TestNG. Звіти це HTML документи які показують результати в графіках. Також надається можливість створення логів, знімків екрану та інших деталей.

Extent Reports містить два важливих класи, які використовуються найчастіше: ExtentReports та ExtentTest.

```
ExtentReports reports = new ExtentReports("Path of directory to store the resultant HTML file", true/false);
ExtentTest test = reports.StartTest("TestName");
```

Клас ExtentReports генерує HTML звіт, базуючись на шляху, вказаному як параметр. Також, існуючі звіти можуть бути перезаписані новими звітами, що вказується Boolean прапорцем. Стандартним значенням є True, тобто старі дані будуть перезаписуватись новими.

Клас ExtentTest логує кроки тестів в попередньо згенерований HTML звіт. Обидва класи можуть використовуватись з наступними вбудованими методами:

- StartTest: виконує передумови тесту;
- EndTest: виконує післяумови тесту;
- Log: логує статус кожного тестового кроку в HTML зві;
- Flush: видаляє всі дані з попередньо згенерованого звіту та створює новий.

Статус тесту Extent Reports ідентифікується наступними значеннями:

- PASS;
- FAIL;
- SKIP;
- INFO.

Нижче наведено відповідний синтаксис коду для кожного з статусів.

```
reports.EndTest();
test.log(LogStatus.PASS,"Test Passed");
test.log(LogStatus.FAIL,"Test Failed");
test.log(LogStatus.SKIP,"Test Skipped");
test.log(LogStatus.INFO,"Test Info");
```

Використання Extent Reports дає користувачам наступні переваги:

- Інтеграцію з багатьма юніт тест фреймворками (NUnit, JUnit, TestNG);
- Можливість відображення знімків екрану для кожного кроку в тесті;
- Простежування декількох тестових запусків в одному звіті;
- Відображення часу виконання тестів;
- Можливість графічного налаштування відображення кожного кроку в тесті.

2.2.3 Реалізація тестових випадків

В ході виконання завдання було створено та спроектовано описані вище тестові скрипти.

Автоматизація описаних сценаріїв відбувалась з використанням наступних інструментів:

- Описаних тестових випадків;
- Фреймворку автоматизованого тестування TestStack.White;
- Юніт тест фреймворку MS Test;
- Класів Screen Objects, які описують елементи інтерфейсу користувача.

Система підтримує два типи користувачів: Administrator та User, для легкого використання в коді, дані типи описані наступним чином:

```
public enum LoginType
{
    User,
    Admin
}
```

Використання enum дозволяє, схиляючись на тип переданого користувача, методу виконувати відповідний код для різного типу користувачів. Нижче наведено приклад використання описаного вище enum.

```
     public static void CopyCustomDataBaseFile(string
filePath, LoginType type)
             {
                 DeleteLogIfExists();

                 if
(FileHelper.GetFileExtension(filePath,
type).Equals(".mdf"))
                 {
                     switch (type)
                     {
                         case LoginType.Admin:

File.Copy(GetPathToCustomDataBase(filePath, type),
FilePaths.adminDestinationPath, true);
                             break;
                         case LoginType.User:

File.Copy(GetPathToCustomDataBase(filePath, type),
FilePaths.userDestinationPath, true);
                             break;
                     }
                 }
                 else if
(FileHelper.GetFileExtension(filePath,
type).Equals(".zip"))
                 {
                     switch (type)
                     {
                         case LoginType.Admin:

FileHelper.ExtractFile(FileHelper.GetFileInFolder(filePa
th, type), FilePaths.adminDestinationPath);
                             break;
                         case LoginType.User:

FileHelper.ExtractFile(FileHelper.GetFileInFolder(filePa
th, type), FilePaths.userDestinationPath);
                             break;
                     }
                 }
             }
```

Даний метод копіює файл бази даних, і в залежності від переданого типу користувача, вставляє файл в відповідну папку.

Ще одним прикладом використання типу користувача є метод який оновлює базу даних даними нового користувача.

```
     public static void UpdateForteDataBase(DatabaseType
type)
             {
                 var connectionString =
type.Equals(DatabaseType.Admin) ? adminConnectionString
: userConnectionString;

                 using (var connection = new
SqlConnection(connectionString))
                 {
                     try
                     {
                         connection.Open();
                     }
                     catch
                     {
                         StopSqlLocalDbInstance();
                         connection.Open();
                     }


ImpersonateUserInForteAdmin(connection);
                 }
             }
```

Даний метод, в залежності від типу користувача, формує стрічку з’єднання з базою даних, і далі виконує скрипт оновлення даних користувача.

В десктопній автоматизації, однією з найважливіших частин проектування автоматизованих тестів є пошук елементів інтерфейсу користувача. Якщо в елемента присутній унікальний ідентифікатор – знайти його буде досить легко, але, при відсутності значення унікального ідентифікатора, можуть виникнути проблеми пошуку елемента.

UI Automation клієнт розглядає всі UI Automation елементи інтерфейсу як набір об’єктів типу AutomationElement, який організований в деревоподібній структурі.

Використовуючи клас TreeWalker, клієнт може виконувати навігацію по дереві елементів, переходячи від одного елемента до іншого в заданому напрямку, використовуючи GetFirstChild, GetLastChild, GetPreviousSibling, GetNextSibling та GetParent методи.

Для спрощення роботи з класом TreeWalker, я створив допоміжний клас TreeHelper, який спрощує пошук елементів в дереві, надаючи набір методів для цього.

```
public static List<UIItem> GetTreeItems(Tree tree)
        {
            UIItem firstChild = new
UIItem(TreeWalker.RawViewWalker.GetFirstChild(tree.Autom
ationElement), new NullActionListener());

            List<UIItem> items = new List<UIItem>
            {
                firstChild
            };

            var startingItem = firstChild;
            while
(TreeWalker.RawViewWalker.GetNextSibling(startingItem.Au
tomationElement) != null)
            {
                var nextItem = new
UIItem(TreeWalker.RawViewWalker.GetNextSibling(startingI
tem.AutomationElement), new NullActionListener());
                items.Add(nextItem);
                startingItem = nextItem;
            }
            return items;
        }
```

Описаний вище метод, отримуючи як параметр дерево, знаходить перший дочірній елемент в ньому, після цього, використовуючи цей елемент,

через цикл знаходяться усі сусідні елементи в дереві та повертається список з всіма елемантами.

```
    public static UIItem GetFirstChild(IUIItem item)
                => new
UIItem(TreeWalker.RawViewWalker.GetFirstChild(item.Autom
ationElement), new NullActionListener());

            public static UIItem GetLastChild(IUIItem
item)
                => new
UIItem(TreeWalker.RawViewWalker.GetLastChild(item.Automa
tionElement), new NullActionListener());

            public static UIItem
GetPreviousSibling(IUIItem item)
                => new
UIItem(TreeWalker.RawViewWalker.GetPreviousSibling(item.
AutomationElement), new NullActionListener());

            public static UIItem GetNextSibling(IUIItem
item)
                => new
UIItem(TreeWalker.RawViewWalker.GetNextSibling(item.Auto
mationElement), new NullActionListener());
```

Також, для простоти користування класом TreeWalker, я створив додаткові методи-обгортки: GetFirstChild, GetLastChild, GetPreviousSibling та GetNextSibling. Дані методи приймають тип IUIItem та повертають відповідний, наступний елемент.

```
    public static IUIItem TryFindElement(Func<IUIItem>
locationStrategy)
            {
                RetryHelper.Instance.Try(() =>
locationStrategy?.Invoke()).
                                    WithTryInterval(3500).
                                    WithMaxTryCount(10).
                                    UntilNoException();

                return locationStrategy?.Invoke();
            }
```

Наведений вище метод використовує бібліотеку Retry для пошуку елементів з інтервалом в 3,5 секунди. Це зроблено для підвищення надійності деяких тестів, тому що пошук деяких елементів потребує більше часу.

Тестовий випадок базового пошуку, який відповідає випадку, описаному в таблиці 2.1:

```
     [TestMethodEx]

[TestCategory(TestCategories.TaskPaneCreateInsertFind)]
             public void TaskPaneBasicFind() //FORTE-
1790
             {

AppInstanceHelper.TryActionUntilNoException(() =>
ForteRibbon.ClickForteTab(LoginType.Admin));

AppInstanceHelper.TryActionUntilNoException(() =>
ForteRibbon.RibbonButton_ShowTaskPane.ClickEx());

ForteTaskPaneCreateInsert.FindField.SetValue(TestData.De
fault.SearchQuery_Master);

WordApp.appHandle().Keyboard.PressSpecialKey(KeyboardInp
ut.SpecialKeys.RETURN);
                 //Verify that there are 11 search
results

ForteTaskPaneCreateInsert.GetAllSearchResults().Count.Sh
ould().Be(11);
                 //Verify that each search result
contains 'Master'

ForteTaskPaneCreateInsert.GetAllSearchResults()
                     .ForEach(searchResult =>
searchResult.Name.Should().ContainEquivalentOf(TestData.
Default.SearchQuery_Master));
             }
```

Для реалізації перевірки результату, в усіх тестах, використовується бібліотека Fluent Assetions. Fluent Assetions це набір .NET методів розширення які дозволяють більш природньо задавати очікуваний результат в стилі TDD або BDD. Це надає простий, інтуїтивний синтаксис, підключивши бібліотеку через

ключове слово using. Це дозволяє використовувати велику кількість методів розширення, наприклад, можна перевірити що стрічка починається, закінчується, та містить певну фразу.

```
     string actual = "ABCDEFGHI";
     actual.Should().StartWith("AB").And.EndWith("HI").A
nd.Contain("EF").And.HaveLength(9);
```

Для перевірки того, що всі елементи в колекції відповідають предикату та містять певну кількість елементів.

```
     IEnumerable numbers = new[] { 1, 2, 3 };
     numbers.Should().OnlyContain(n => n > 0);
     numbers.Should().HaveCount(4, "because we thought
we put four items in the collection");
```

В прикладі вище, при не проходженні тесту, буде виведено помилку з повідомленням, яке передано як стрічка в методі HaveCount.

Тестовий випадок складного пошуку, який відповідає тестовому випадку, описаному в таблиці 2.2:

```
     [TestMethodEx]

[TestCategory(TestCategories.TaskPaneCreateInsertFind)]
             public void TaskPaneAdvancedFind() //FORTE-
1791
             {

ForteRibbon.ClickForteTab(LoginType.Admin);

ForteRibbon.RibbonButton_ShowTaskPane.ClickEx();

ForteTaskPaneCreateInsert.AdvancedSearchButton.ClickEx()
;

AdvancedFindOptionsWindow.SearchForField.ClickEx();

AdvancedFindOptionsWindow.SearchForField.SetValue(TestDa
ta.Default.SearchQuery_Master);

AdvancedFindOptionsWindow.ContentType.Select(TestData.De
fault.ContentType_Segments);
```

```
AdvancedFindOptionsWindow.OKButton.ClickEx();
                //Verify that there is only 1 search
result

ForteTaskPaneCreateInsert.GetAllSearchResults().Count.Sh
ould().Be(1);
                //Verify that search result contains
'Master'

ForteTaskPaneCreateInsert.GetAllSearchResults().Single()
.Name.Should().ContainEquivalentOf(TestData.Default.Sear
chQuery_Master);
            }
```

Тестовий випадок створення папки в системі, який відповідає тестовому випадку, описаному в таблиці 2.3:

```
     [TestMethodEx]

[TestCategory(TestCategories.TaskPaneCreateInsertFolder)
]
            public void TaskPaneCreateNewFolder()
//FORTE-1767
            {

ForteRibbon.ClickForteTab(LoginType.Admin);

ForteRibbon.RibbonButton_ShowTaskPane.ClickEx();

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameCorrespondence).RightClick();

ContextMenu.GetItem(TestData.Default.ContextItem_CreateN
ewFolder).ClickEx();

CreateNewFolderWindow.DisplayName.SetValue(TestData.Defa
ult.FolderNameAlpha);
                CreateNewFolderWindow.HelpText.Focus();

CreateNewFolderWindow.HelpText.SetValue(TestData.Default
.HelpTextBeta);

CreateNewFolderWindow.OKButton.ClickEx();
                //Verify that "Alpha" folder is created
```

```
ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameAlpha).Name.Should().Be(TestData.Default.FolderNa
meAlpha);
                //Verify that "Alpha" folder is visible

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameAlpha).Visible.Should().BeTrue();
            }
```

Тестовий випадок видалення папки з системи, який відповідає тестовому випадку, описаному в таблиці 2.4:

```
     [TestMethodEx]

[TestCategory(TestCategories.TaskPaneCreateInsertFolder)
]
            public void TaskPaneDeleteFolder() //FORTE-
1769
            {

ForteRibbon.ClickForteTab(LoginType.Admin);

ForteRibbon.RibbonButton_ShowTaskPane.ClickEx();

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameCorrespondence).RightClick();

ContextMenu.GetItem(TestData.Default.ContextItem_CreateN
ewFolder).ClickEx();

CreateNewFolderWindow.DisplayName.SetValue(TestData.Defa
ult.FolderNameAlpha);
                CreateNewFolderWindow.HelpText.Focus();

KeyboardFunctions.EnterText(TestData.Default.HelpTextBet
a);

CreateNewFolderWindow.OKButton.ClickEx();

AppInstanceHelper.WaitForElementToBeVisible(ForteTaskPan
eCreateInsert.GetFolder(TestData.Default.FolderNameAlpha
));
                //Verify that Alpha folder is a child
of the Correspondence folder using X and Y bounds
```

```
ForteTaskPaneCreateInsert.IsFolderAChild(ForteTaskPaneCr
eateInsert.GetFolder(TestData.Default.FolderNameCorrespo
ndence),

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameAlpha),

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derName_Envelopes)).Should().BeTrue();

ForteTaskPaneCreateInsert.GetFolder(TestData.Default.Fol
derNameAlpha).RightClick();

ContextMenu.GetItem(TestData.Default.ContextItem_Delete)
.ClickEx();
                LiteraForteDialog.YesButton.ClickEx();
                //Verify that "Alpha" folder does not
exist

ForteTaskPaneCreateInsert.DoesFolderExist(TestData.Defau
lt.FolderNameAlpha).Should().BeFalse();
            }
```

2.2.4 Реалізація тестових випадків API

Для реалізації тестових запитів API, я створив допоміжні методи, на основі HttpClient класу, які виконують основні типи запитів: POST, GET, PUT та DELETE.

Реалізуючи методи запитів, я звернув увагу на формування uri запиту. Враховуючи що запит може мати query та path параметри, було реалізовано механізм формування повного uri, базуючись на типові параметрів. Path параметри передаються в формі List<string>, query параметри, в свою чергу, передаються як Dictionary<string, string>.

Нижче наведено приклад коду, який перевіряє, чи тип параметра рівний List<string>:

```
if (parameters.GetType() == typeof(List<string>))
                    {
```

```
                        completeUri                    =
GetPathUri(endpointPath:    endpoint,    parametersPath:
parameters, segmentPath: segment);
                    }
```

Аналогічно перевіряється query параметр тип на typeof(Dictionary<string,string). Тестування API я провів в режимі Ad Hoc. Ad Hoc тестування, це часто використовуваний тип тестування, який виконується без планування та документації, але може застосовуватись на ранніх етапах розробки продукту.

Враховуючи деяку схожість функціоналу API та функцій додатку Forte в Microsoft Word, я вирішив реалізувати суміжні методи.

Реалізація створення папки за допомогою API виклику:

```
    [Test]
            [Category(TestCategories.Folders)]
            [Order(1)]
            public void CreateFolder() // POST
/api/v1/Folders
            {
                var requestBody = new
SpecificFolderPostModel();
                var queryParameters = new
Dictionary<string, string>() {{"type", "1"}};
                //Post request should create the new
folder and the fields in the response body should equal
the expected ones.
                var response =
RequestHelper.Instance.PostRequest(endpoint:
EnvironmentVariables.Default.FoldersEndpoint,
parameters: queryParameters, requestBody: requestBody);

response.StatusCode.Should().Be(HttpStatusCode.Created);

                //Response body is deserealized into
the SpecificFolderPostResponseModel type object.Its
properties are validated.
                var postResponseBody =
response.GetJsonStringResponseBody<SpecificFolderPostRes
ponseModel>();
```

```
RuntimeDataBridge.Instance.AddGlobalTestData(DataLabels.
Default.FolderLabel, postResponseBody);


postResponseBody.Name.Should().Be(requestBody.Name,
because: "The folder with the defined 'name' should be
created.");

postResponseBody.ParentId.Should().Be(requestBody.Parent
Id, because: "The folder with the defined 'parentId'
should be created.");

postResponseBody.SharedFolderId.Should().Be(requestBody.
SharedFolderId, because: "The folder with the defined
'SharedFolderId' should be created.");

postResponseBody.Description.Should().Be(requestBody.Des
cription, because: "The folder with the defined
'Description' should be created.");

postResponseBody.Index.Should().Be(requestBody.Index,
because: "The folder with the defined 'Description'
should be created.");
            }
```

Кожен тестовий метод, який створює дані, отримує модель цих даних як параметр. В прикладі вище передається модель SpecificFolderModel.

```
    public class SpecificFolderPostModel
        {
            public string Name { get; }
            public Guid? ParentId { get; set; }
            public Guid? SharedFolderId { get; }
            public string Description { get; }
            public int Index { get; }

            public SpecificFolderPostModel()
            {
                Name =
$"{DataGenerator.Instance.RandomFirstName}{DataGenerator
.Instance.GetRandomNumber(999)}";
                ParentId = null;
                SharedFolderId = null;
```

```
                Description =
DataGenerator.Instance.RandomFirstName;
                Index = 0;
            }
        }
```

Моделі створюються використовуючи схему в Swagger UI. Клас моделі, зазвичай, має набір властивостей та конструктор який формує конкретну модель з заповненими властивостями. Також, як і у фреймворку десктопної автоматизації, для перевірки правильності виконання тесту використовується бібліотека Fluent Assertions. При кожній перевірці в параметр because передається повідомлення, яке відображається у разі провалу проходження тесту.

Аналогічно реалізовується тестовий випадок видалення папки:

```
    [Test]
            [Category(TestCategories.Folders)]
            [Order(10)]
            public void DeleteFolder() // DELETE
/api/v1/Folders/{id}
            {
                var savedFolderMemberData =
(AddFolderMemberResponseModel)
RuntimeDataBridge.Instance.ExtractGlobalTestData(DataLab
els.Default.FoldersFolderMemberLabel);

                //Delete folder member first, before
deleting a folder.

RequestHelper.Instance.DeleteRequest(endpoint:
EnvironmentVariables.Default.FolderMembersEndpoint,
parameters: savedFolderMemberData.Id)

.StatusCode.Should().Be(HttpStatusCode.OK);

                var savedFolderData =
(SpecificFolderPostResponseModel)
RuntimeDataBridge.Instance.ExtractGlobalTestData(DataLab
els.Default.FolderLabel);

                //Response should be received and its
response code validated.
```

```
RequestHelper.Instance.DeleteRequest(endpoint:
EnvironmentVariables.Default.FoldersEndpoint,
parameters: savedFolderData.Id)

.StatusCode.Should().Be(HttpStatusCode.OK);

                //Response should be received and its
response code validated.

RequestHelper.Instance.GetRequest(endpoint:
EnvironmentVariables.Default.FoldersEndpoint,
parameters: savedFolderData.Id)

.StatusCode.Should().Be(HttpStatusCode.NotFound);
        }
```

2.3 Використання Bamboo CI/CD

CI/CD технології дозволяють пришвидшити процес тестування та розгортання продукту, та забезпечують швидкий та ефективний результат [47].

CI має на увазі постійну інтеграцію. Вона показує постійні генерації збірок та послідовності тестів для будь-якого програмного продукту який збирається. CI постійно перевіряє код на випадки змін та модифікацій.

CD має на увазі постійну доставку. Вона забезпечує адміністрування автоматичних збірок продукту.

CI/CD лінія – це легкий в користуванні фреймворк. Багато інструментів використовують це для забезпечення швидкого виходу на ринок своїх продуктів. Bamboo є одним з продуктів який реалізовує CI/CD фреймворк.

Bamboo CI сервер допомагає:

- автоматизувати тестування програмних продуктів для пришвидшення релізів, створюючи CD лінію;
- автоматизувати збірки, документацію логів та виконання тестів [48].

В цілях автоматизації було створено план. План, в bamboo, описує все що потрібно для процесу збірки [49].

Для створення плану потрібно виконати ряд наступних кроків:

1. В меню Bamboo вибрати Create, потім Create plan;
2. Заповнити деталі плану на сторінці конфігурації;
3. Зв’язати репозиторій коду з планом;
4. Натиснути Configure plan.

Після успішного налаштування плану, результати виконання планів автоматично відображаються в системі контролю версій [50].

Нижче наведений приклад успішної збірки, результат якої відображається в Bitbucket.

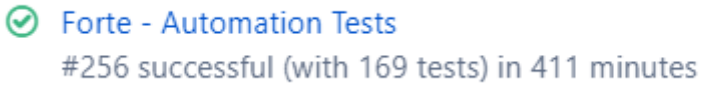

Рисунок 2.8 – Результат виконання успішного Bamboo плану

В результаті відображається назва плану, номер збірки, ідентифікація успіху, кількість виконаних тестів та час їх виконання [51].

Аналогічна інформація відображається в разі неуспішного виконання плану.

Рисунок 2.9 – Результат виконання неуспішного Bamboo плану

Отже у розділі було розглянуто особливості архітектури автоматизації десктопних та API додатків та допоміжних інструментів для реалізації автоматизації [52]. Реалізовано набір тестів, та розглянуто процес інтеграції звітування. Крім цього, описано переваги використання CI/CD процесів [53].

3 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

3.1 Охорона праці

Враховуючи, що розробка і експлуатація програмного продукту “фреймворк автоматизації десктопних та REST API додатків” здійснюється на комп’ютері, потрібно розглянути основні нормативно-правові документи та особливості охорони праці під час використання комп'ютерної техніки.

Перелік нормативно-правових актів, що певним чином регулюють це питання, є доволі широким. Обов’язки роботодавця в питанні забезпечення працівникам безпечних та комфортних умов для здійснення роботи, і також права працівників на ці умови передбачено частиною 2 ст. 2 та ч. 1 ст. 21 КЗпП, а також ст. 13 Закону України «Про охорону праці». Згаданий закон оприділяє основні положення які покривають питання реалізації конституційного права працівників на охорону їх життя і здоров'я під час процесу трудової діяльності, на належні, безпечні і здорові умови праці, також, цей закон регулює відносини між працівником та роботодавцем з питань безпеки, гігієни праці та виробничого середовища і встановлює єдиний порядок організації охорони праці в Україні. Переважну частину актів у сфері розробки програмного забезпечення становлять акти підзаконного рівня, тобто численні інструкції, правила, державні санітарні правила і норми (ДСанПІН) тощо, якими врегульовуються окремі моменти щодо власне конструкції електронно-обчислювальної техніки, особливостей облаштування приміщень для роботи з нею та низки інших подібних вимог.

На даний час, до основних підзаконних актів у сфері розробки програмного забезпечення можна віднести:

- Примірну інструкцію з охорони праці під час експлуатації електронно-обчислювальних машин, затверджену наказом Міністерства доходів і зборів України від 05.09.2013 № 443 [54];

- Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПіН 3.3.2.007-98, затверджені постановою Головного державного санітарного лікаря України від 10.12.1998 № 7 [55];
- Постанову Міністерства Охорони Здоров'я України «Про санітарні норми мікроклімату виробничих приміщень» від 01.12.1999 № 42.

Під час розробки програмного продукту, вимоги щодо розміщення і планування приміщень для роботи з комп'ютером були дотримані. В обладнанні приміщень не використовувались полімерні матеріали, які могли виділяти шкідливі хімічні речовини. Також, під час розробки фреймворку автоматизації, приділялась увага забезпеченню достатнім для здійснення роботи рівнем освітлення (природного та штучного – у темну пору доби) та звукоізоляції. Також, у приміщеннях, де здійснювалась робота з комп'ютерами, періодично проводиться вологе прибирання, з метою недопущення запиленості підлоги та меблів [56].

Підбиваючи підсумки, можна зробити висновок, що розробка програмної системи автоматизації відповідає вимогам охорони праці та пожежної безпеки.

3.2 Безпека в надзвичайних ситуаціях

Електробезпека —система організаційних і технічних заходів та засобів, що забезпечують захист людей від шкідливої і небезпечної дії електричного струму, електричної дуги, електричного поля і статичної електрики.

Чинники, що впливають на тяжкість ураження людини електричним струмом, поділяються на три групи: електричного характеру, неелектричного характеру і чинники виробничого середовища. Основні чинники електричного характеру – це величина струму, що проходить крізь людину, напруга, під яку вона потрапляє, та опір її тіла, рід і частота струму. Величина струму, що проходить крізь тіло людини, безпосередньо і найбільше впливає на тяжкість ураження електричним струмом. За характером дії на організм виділяють:

- відчутний струм – викликає при проходженні через організм відчутні подразнення;
- невідпускаючий струм – викликає при проходженні через організм непереборні судомні скорочення м'язів руки, в якій затиснуто провідник;
- фібриляційний струм – при проходженні через організм викликає фібриляцію серця.

Відповідно до наведеного вище:

- пороговий відчутний струм (найменше значення відчутного струму) для змінного струму частотою 50 Гц коливається в межах 0,6 – 1,5 мА і 5 – 7 мА – для постійного струму;
- пороговий невідпускаючий струм (найменше значення невідпускаючого струму) коливається в межах 10 – 15 мА для змінного струму і 50 – 80 мА – для постійного;
- пороговий фібриляційний струм (найменше значення фібриляційного струму) знаходиться в межах 100 мА для змінного струму і 300 мА для постійного.

Гранично допустимий струм, що проходить крізь тіло людини при нормальному (неаварійному) режимі роботи електроустановки, не повинен перевищувати 0,3 мА для змінного і 1 мА для постійного.

Величина напруги, під яку потрапляє людина, впливає на тяжкість ураження електричним струмом в тій мірі, що зі збільшенням прикладеної до тіла напруги зменшується опір тіла людини. Останнє призводить до збільшення струму в мережі замикання через тіло людини і, як наслідок, до збільшення тяжкості ураження.

Дія статичної електрики для людини безпечна, бо сила струму дуже мала, але: розряд енергії відбувається у вигляді помірного і сильного уколу або поштовху; вплив зарядів може призвести до тяжких нещасних випадків внаслідок рефлекторного руху поблизу незахищених та рухомих частин, перебування на висоті, іскрові розряди можуть спричинити спалах або вибух

горючих речовин; вибухи при перевезенні рідин у незаземлених цистернах тощо.

Приміщення із робочими місцями користувачів комп'ютерів для забезпечення електробезпеки обладнання, а також для захисту від ураження електричним струмом самих користувачів ПК повинні мати достатні технічні засоби захисту.

Під час монтажу та експлуатації ліній електромережі необхідно повністю унеможливити виникнення електричного джерела загоряння внаслідок короткого замикання та перевантаження проводів, обмежувати застосування проводів з легкозаймистою ізоляцією і, за можливості, перейти на негорючу ізоляцію.

Лінія електромережі для живлення ЕОМ, периферійних пристроїв ЕОМ та устаткування для обслуговування, ремонту та налагодження ЕОМ виконується як окрема групова трипровідна мережа, шляхом прокладання фазового, нульового робочого та нульового захисного провідників. Нульовий захисний провідник використовується для заземлення (занулення) електроприймачів.

Використання нульового робочого провідника як нульового захисного провідника забороняється. Нульовий захисний провід прокладається від стійки групового розподільчого щита, розподільчого пункту до розеток живлення. Не допускається підключення на щиті до одного контактного затискача нульового робочого та нульового захисного провідників. Площа перерізу нульового робочого та нульового захисного провідника в груповій трипровідній мережі повинна бути не менше площі перерізу фазового провідника.

Усі провідники повинні відповідати номінальним параметрам мережі та навантаження, умовам навколишнього середовища, умовам розподілу провідників, температурному режиму та типам апаратури захисту, вимогам Правил налаштування електроустанов.

У приміщенні, де одночасно експлуатується або обслуговується більше

п'яти персональних ЕОМ, на помітному та доступному місці встановлюється аварійний резервний вимикач, який може повністю вимкнути електричне живлення приміщення, крім освітлення.

ЕОМ, периферійні пристрої ЕОМ та устаткування для обслуговування, ремонту та налагодження ЕОМ повинні підключатися до електромережі тільки з допомогою справних штепсельних з'єднань і електророзеток заводського виготовлення. Штепсельні з'єднання та електророзетки крім контактів фазового та нульового робочого провідників повинні мати спеціальні контакти для підключення нульового захисного провідника. Конструкція їх має бути такою, щоб приєднання нульового захисного провідника відбувалося раніше ніж приєднання фазового та нульового робочого провідників. Порядок роз'єднання при відключенні має бути зворотним. Необхідно унеможливити з'єднання контактів фазових провідників з контактами нульового захисного провідника.

Неприпустимим є підключення ЕОМ та периферійних пристроїв ЕОМ до звичайної двопровідної електромережі, в тому числі – з використанням перехідних пристроїв.

Електромережі штепсельних з'єднань та електророзеток для живлення ЕОМ, периферійних пристроїв слід виконувати за магістральною схемою, по 3…6 з’єднань або електророзеток в одному колі. Штепсельні з'єднання та електророзетки для напруги 12 В та 36 В за своєю конструкцією повинні відрізнятися від штепсельних з'єднань для напруги 127 В та 220 В і мають бути пофарбовані в колір, який візуально значно відрізняється від кольору штепсельних з'єднань, розрахованих на напругу 127 В та 220 В.

Індивідуальні та групові штепсельні з'єднання та електророзетки необхідно монтувати на негорючих або важкогорючих пластинах з урахуванням вимог Правил налаштування електроустанов та Правил пожежної безпеки в Україні.

Електромережу штепсельних розеток для живлення ЕОМ, периферійних пристроїв ЕОМ при розташуванні їх уздовж стін приміщення прокладають по

підлозі поряд зі стінами приміщення, як правило, в металевих трубах і гнучких металевих рукавах з відводами відповідно до затвердженого плану розміщення обладнання та технічних характеристик обладнання.

При розташуванні в приміщенні за його периметром до 5 ЕОМ, використанні трипровідникового захищеного проводу або кабелю в оболонці з негорючого або важкогорючого матеріалу дозволяється прокладання їх без металевих труб та гнучких металевих рукавів.

Електромережу штепсельних розеток для живлення ЕОМ при розташуванні їх у центрі приміщення, прокладають у каналах або під знімною підлогою в металевих трубах або гнучких металевих рукавах. При цьому не дозволяється застосовувати провід і кабель в ізоляції з вулканізованої гуми та інші матеріали, що містять сірку. Відкрита прокладка кабелів під підлогою забороняється. Металеві труби та гнучкі металеві рукави повинні бути заземлені. Заземлення повинно відповідати вимогам НПАОП 40.1-1.21-98.

Для підключення переносної електроапаратури застосовують гнучкі проводи в надійній ізоляції.

Тимчасова електропроводка від переносних приладів до джерел живлення виконується найкоротшим шляхом без заплутування проводів у конструкціях машин, приладів та меблях. Доточувати проводи можна тільки шляхом паяння з наступним старанним ізолюванням місць з'єднання.

Є неприпустимими:

– експлуатація кабелів та проводів з пошкодженою або такою, що втратила захисні властивості за час експлуатації, ізоляцією; залишення під напругою кабелів та проводів з неізольованими провідниками;

– застосування саморобних подовжувачів, які не відповідають вимогам Правил влаштування електроустанов до переносних електропроводок;

– застосування для опалення приміщення нестандартного (саморобного) електронагрівального обладнання або ламп розжарювання;

– користування пошкодженими розетками, розгалужувальними та з'єднувальними коробками, вимикачами та іншими електровиробами, а також лампами, скло яких має сліди затемнення або випинання;

– підвішування світильників безпосередньо на струмопровідних проводах, обгортання електроламп і світильників папером, тканиною та іншими горючими матеріалами, експлуатація їх зі знятими ковпаками (розсіювачами);

– використання електроапаратури та приладів в умовах, що не відповідають вказівкам (рекомендаціям) підприємств-виготовлювачів.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було оглянуто процес тестування програмних продуктів, описані переваги інтеграції автоматизованого тестування в процес розробки програмного забезпечення. Також було описано переваги використаної мови програмування, та переваги кожного з допоміжних інструментів реалізації проекту.

В якості системи автоматизації тестування було спроектовано та реалізовано фреймворк, який дозволяє легко запускати написані тестові скрипти та надавати звітність після закінчення тестових запусків. Проектуючи та реалізовуючи фреймворк, були дотримані найкращі практики проектування та розробки програмного забезпечення, що робить розроблений фреймворк схильним до масштабування та підтримуваності [57].

Фреймворк описує об’єкти елементів інтерфейсу, які дозволяють з легкістю взаємодіяти з інтерфейсом користувача, слідуючи принципам ScreenObject підходу.

Під час реалізації проекту були спроектовані тестові випадки, на основі яких були реалізовані власне тестові скрипти.

Враховуючи масштабованість розробленого фреймворку, можна з легкістю створювати нові тестові випадки, розширюючи бібліотеку ScreenObjects, та при зміні дизайну програмного продукту, робити зміни лише в об’єктах елементів інтерфейсу, а не у самих тестах.

Основною метою проекту, та його реалізації, є аналіз та впровадження процесу оцінки якості програмного забезпечення в глобальний життєвий цикл розробки. Розроблений тестовий фреймворк може використовуватись будь-ким, так як він інтегрований в процеси CI/CD, що дозволяє будь-кому запустити певний набір тестів, і отримати результати в інтерфейсі веб браузера. Також фреймворк можна розширювати, реалізовуючи інші тестові сценарії та інші компоненти інформаційної системи.

Підсумовуючи виконану роботу, для проектування та реалізації фреймворку автоматизації, було використано набір наступних інструментів, підходів та технологій:

- Мову програмування C# та .NET Framework;
- Середовище розробки Visual Studio 2019;
- Фреймворки для юніт-тестування: MS Test та NUnit;
- Інструмент автоматизації десктопних додатків TestStack.White;
- Інструмент опису API Swagger UI;
- Інструмент пошуку WinForms/WPF локаторів Inspect;
- Патерн проектування ScreenObjects;
- Інструмент реалізації CI/CD Bamboo.

Проект реалізовано в предметній області яка завжди залишається актуальною та корисною суспільству. Програмне рішення повинно скоротити витрати та час на перевірку якості програмних продуктів, та надати зацікавленим сторонам загальну картину про стан розроблюваного програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Best Practices in Developing Programs [Digital resource] – Mode of access: URL: https://www.cs.utexas.edu/~mitra/csSummer2014/cs312/lectures/bestPractices.html. – Last access: 2020. – Title from the screen.

2. Glossary of Computer System Software Development Terminology [Digital resource] – Mode of access: URL: https://www.fda.gov/inspections-compliance-enforcement-and-criminal-investigations/inspection-guides/glossary-computer-system-software-development-terminology-895 – Last access: 2020. – Title from the screen.

3. Jon Skeet. C# in Depth [Text] – 2008 – 39 p.

4. Joseph Albahari. C# 7.0 In a Nutshell: The Definitive Reference [Text] – 2017 – 144 p.

5. Roy Osherove. The Art of Unit Testing [Text] – 2009 – 123 p.

6. Robert Cecil Martin. Clean Code [Text] – 2008 – 94 p.

7. What is document assembly? [Digital resource] – Mode of access: URL : http://www.capstonepractice.com/what-is-document-assembly – Last access: 2020. – Title from the screen.

8. Fall in line with document assembly [Digital resource] – Mode of access: URL: https://static1.squarespace.com/static/571acb59e707ebff3074f461/t/5946f7724f14bc4eadc55c3f/1497823100241/loc2006.pdf – Last access: 2020. – Title from the screen.

9. Power Tools for Document Preparation [Digital resource] – Mode of access: URL: https://static1.squarespace.com/static/571acb59e707ebff3074f461/t/5946f640ccf210058d375d32/1497822784461/amlaw6.pdf – Last access: 2020. – Title from the screen.

10. Reinventing Reinvention [Digital resource] – Mode of access: URL : https://static1.squarespace.com/static/571acb59e707ebff3074f461/t/5946f7fedb29d6

52c27d1867/1497823230912/Reinventing.pdf – Last access: 2020. – Title from the screen.

11. The Case for Court-Based Document Assembly Programs [Digital resource] – Mode of access: URL : https://www.srln.org/node/391/article-case-court-based-document-assembly-programs-review-new-york-state-court-system%E2%80%99s – Last access: 2020. – Title from the screen.

12. Substantive Legal Software Quality – A Gathering Storm? [Digital resource] – Mode of access: URL: https://static1.squarespace.com/static/571acb59e707ebff3074f461/t/5cfe69e4a851d80001bdf162/1560177127345/Substantive+Legal+Software+Quality+-+A+Gathering+Storm.pdf – Last access: 2020. – Title from the screen.

13. Document Assembly – Changing Talk into Action [Digital resource] – Mode of access: URL: https://www.adamsdrafting.com/document-assembly/ – Last access: 2020. – Title from the screen.

14. Document Assembly Systems [Digital resource] – Mode of access: URL: http://www.elawexchange.com/index.php?option=com_content&view=article&id=302&Itemid=334 – Last access: 2020. – Title from the screen.

15. Document Management and Automation [Digital resource] – Mode of access: URL: https://lawyerist.com/reviews/document-management-automation/ – Last access: 2020. – Title from the screen.

16. Sam Harden. How to DIY Document Assembly [Digital resource] – Mode of access: URL: https://lawyerist.com/blog/do-it-yourself-document-assembly/ – Last access: 2020. – Title from the screen.

17. How Document Assembly Solves Three Biggest Challenges [Digital resource] – Mode of access: URL: https://www.attorneyatwork.com/document-assembly-solves-three-biggest-challenges/ – Last access: 2020. – Title from the screen.

18. Web Content Accessibility Guidelines 1.0 [Electronic resource] – Mode of access: URL: https://www.w3.org/TR/WAI-WEBCONTENT/ – Last access: 2020. – Title from the screen.

19. St.Amant, Kirk. Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives [Text] – 2007 – 767 p.

20. Litera Forte Features [Digital resource] – Mode of access: URL: https://www.litera.com/products/legal/document-assembly/ – Last access: 2020. – Title from the screen.

21. Kaner, Cem. Exploratory Testing [Digital resource] – Mode of access: URL: http://www.kaner.com/pdfs/ETatQAI.pdf – Last access: 2020. – Title from the screen.

22. Why is software testing necessary? [Digital resource] – Mode of access: URL: http://tryqa.com/why-is-testing-necessary/ – Last access: 2020. – Title from the screen.

23. Software Testing Objectives [Digital resource] – Mode of access: URL: https://www.professionalqa.com/testing-objectives – Last access: 2020. – Title from the screen.

24. Lewis, W.E. Software Testing and Continuous Quality Improvement (3rd ed) [Text] – 2016 – 642 p.

25. IEEE standard for software test documentation [Electronic resource] – Mode of access: URL: https://web.cs.dal.ca/~arc/teaching/CS3130/Templates/TestingTemplates/Test%20Plan%20Templates/IEEEStandardTestPlans.doc – Last access: 2020. – Title from the screen.

26. Google Tech Dev Guide [Digital resource] – Mode of access: URL: https://techdevguide.withgoogle.com/ – Last access: 2020. – Title from the screen.

27. Robert Cecil Martin. Agile Software Development: Principles, Patterns, and Practices [Text] – 2002 – 199 p.

28. Wiegers, K. Creating a Software Engineering Culture [Text] – 2013 – 328 p

29. Jeff Carollo. How Google Tests Software [Text] – 2012 – 120p.

30. Glenford Myers. The Art of Software Testing [Text] – 1979 – 193 p.

31. Binder, Robert V. Testing Object-Oriented Systems: Objects, Patterns, and Tools [Text] – 1999 – 844 p.

32. Woods, Anthony J. Operational Acceptance – an application of the ISO 29119 Software Testing standard [Digital resource] – Mode of access: URL: https://www.scribd.com/document/257086897/Operational-Acceptance-Test-White-Paper-2015-Capgemini – Last access: 2020. – Title from the screen.

33. Alan C. Page. How We Test Software at Microsoft [Text] – 2008 – 284 p.

34. Standard Glossary of Terms used in Software Testing [Digital resource] – Mode of access: URL: https://www.astqb.org/documents/Glossary-of-Software-Testing-Terms-v3.pdf – Last access: 2020. – Title from the screen.

35. Ron Patton. Software Testing [Text] – 2000 – 248p.

36. Rex Black. Pragmatic Software Testing [Text] – 2007 – 178p.

37. Robert Cecil Marin. Clean Architecture: A Craftsman's Guide to Software Structure and Design [Text] – 2017 – 367 p.

38. Kaner, Cem; Falk, Jack; Nguyen, Hung Q. Testing Computer Software [Text] – 1999 – 477 p.

39. IEE. A Systematic Review of Software Testability Measurement Techniques [Digital resource] – Mode of access: URL: https://ieeexplore.ieee.org/abstract/document/8675006 – Last access: 2020. – Title from the screen.

40. Casey Rosenthal; Nora Jones. Chaos Engineering: System in Practice [Text] – 2020 – 248p.

41. Software Testing Life Cycle – Different Stages of Testing [Digital resource] – Mode of access: URL: https://www.edureka.co/blog/software-testing-life-cycle/ – Last access: 2020. – Title from the screen.

42. Mark Fewster. Software Test Automation: Effective Use of Test Execution Tools [Text] – 1999 – 86p.

43. Kristie Magowan. Shift Left Testing: What, Why and How To Shift Left [Digital resource] – Mode of access: URL: https://www.bmc.com/blogs/what-is-shift-left-shift-left-testing-explained/ – Last access: 2020. – Title from the screen.

44. Udi Weinberg. When Should a Test Be Automated? [Electronic resource] – Mode of access: URL: https://techbeacon.com/app-dev-testing/introducing-automation-your-organization-strategic-move – Last access: 2020. – Title from the screen.

45. Charles Rodriguez; Alejandro Berardinelli. When to Automate a Test? [Digital resource] – Mode of access: URL: https://abstracta.us/blog/test-automation/when-to-automate-a-test/ – Last access: 2020. – Title from the screen.

46. Dave Astles. Test Driven Development: A Practical Guide [Digital resource] – Mode of access: URL: https://dl.acm.org/doi/book/10.5555/864016 – Last access: 2020. – Title from the screen.

47. Cathy Zhang. Enable Continuous Delivery with Layered Testing Approach [Digital resource] – Mode of access: URL: https://blogs.perficient.com/2017/06/08/enable-continuous-delivery-with-layered-testing-approach/ – Last access: 2020. – Title from the screen.

48. Getting Started with Bamboo [Digital resource] – Mode of access: URL: https://confluence.atlassian.com/bamboo/getting-started-with-bamboo-289277283.html – Last access: 2020. – Title from the screen.

49. Understanding the Bamboo CI Server [Digital resource] – Mode of access: URL: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html – Last access: 2020. – Title from the screen.

50. Marko Anastasov. The benefits of Continuous Integration and Delivery [Digital resource] – Mode of access: URL: https://www.techradar.com/news/the-benefits-of-continuous-integration-and-delivery – Last access: 2020. – Title from the screen.

51. NUnit [Електронний ресурс] – Режим доступу: URL: https://uk.wikipedia.org/wiki/NUnit – Назва з екрана.

52. Test Design Considerations [Digital resource] – Mode of access: URL: https://docs.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern – Last access: 2020. – Title from the screen.

53. Martin Fowler. PageObject [Digital resource] – Mode of access: URL: https://martinfowler.com/bliki/PageObject.html – Last access: 2020. – Title from the screen.

54. Міністерство Доходів і Зборів України [Електронний ресурс] – Режим доступу: URL: https://zakon.rada.gov.ua/rada/show/v0443810-13#Text – Назва з екрану.

55. Міністерство Охорони Здоров'я України [Електронний ресурс] – Режим доступу: URL: https://zakon.rada.gov.ua/rada/show/v0007282-98#Text - Назва з екрану.

56. В.І. Голінько, М.Ю. Іконніков, Я.Я. Лебедєв Охорона праці в галузі інформаційних технологій [Текст] – Дніпропетровськ : НГУ – 2015. – 247 с.

57. М.Р. Петрик, Д.М. Михалик, О.Ю. Петрик, Г.Б. Цуприк. Методичні вказівки до виконання атестаційної роботи магістра за спеціальністю 121 – "Інженерія програмного забезпечення" для усіх форм навчання [Текст] – Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя – 2020 – 27 с.

ДОДАТКИ

Додаток А

Технічне завдання

Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії

Кафедра програмної інженерії

ЗАТВЕРДЖУЮ

Завідувач кафедрою

програмної інженерії

“___ “_____ 2020 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання кваліфікаційної роботи магістра

на тему: «Проектування та розробка системи для автоматизації веб та десктоп додатків на базі C#, HttpClient та TestStack.White»

Гавурі Роману Валентиновичу <СПм-> ТЗ

Керівник роботи:

к.ф.-м.н., доцент Бойко І.В.

“___” _____ 2020р.

_____

Виконавець:

студент групи СПм-61

Гавура Роман Валентинович

“___” _____ 2020р.

_____

м. Тернопіль – 2020

ЗМІСТ

Вступ

1. Підстави до розробки

2. Призначення до розробки

3. Вимоги до програмного продукту

3.1 Функціональні характеристики

3.2 Склад та параметри технічних засобів

3.3 Інформаційна та програмна сполучність

4. Стадії розробки

5. Програмна документація

6. Порядок контролю та приймання

1 ПІДСТАВИ ДО РОЗРОБКИ

Розробка проводиться у відповідності до графіку навчального плану на 2020 рік, та згідно наказу на виконання кваліфікаційної роботи студента-магістра.

Тема проекту: «Проектування та розробка системи для автоматизації веб та десктоп додатків на базі C#, HttpClient та TestStack.White».

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

Тестування це невід'ємна частина процесу розробки програмного забезпечення, і для застосунків які прагнуть досягнути найвищих стандартів якості, тестування може бути нелегким процесом. З збільшенням кількості функціональних можливостей, якими наділені сьогодні програми, не кажучи вже про велику кількість платформ та браузерів, на які потрібно зважати, постійно зростає можливість помилок та проблем залишитись непоміченими. Однак висококонкурентний ринковий сценарій не дозволяє розробникам програмного забезпечення розкіш дозволяти продуктам навіть з незначними помилками виходити на ринок, оскільки такий продукт буде відхилений.

Інтеграція автоматизованого тестування в життєвий цикл розробки програмного забезпечення дозволяє зацікавленим сторонам бути впевненими в якості розроблюваного програмного продукту.

Об’єктом дослідження є процес розробки сучасних програмних систем. Предмет дослідження: якість програмного забезпечення та можливість її покращення впроваджуючи процес автоматизованого тестування у процес розробки програмного забезпечення. У даній дослідницькій роботі застосовуються сучасні методи розробки та написання автоматизованого тестування веб та десктопних додатків за допомогою таких інструментів, як мова

програмування C#, а також TestStack.White та NUnit. Засобом для цього є розширення для текстового редактора Microsoft Word.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Функціональні характеристики

Програмне забезпечення має виконувати наступні дії:

- надавати доступ до виконання автоматизованих тестових сценаріїв у легкій формі;
- автоматизовано тестувати комплексні декстоп та веб додатки за допомогою інструментів TestStack.White та NUnit;
- спрощувати створення нових тестових сценаріїв в існуючій системі;
- мінімізувати зусилля, які витрачаються на підтримку фреймворку та тестових скриптів;

3.2 Склад та параметри технічних засобів

1) ПК із 4096 Мб оперативної пам‘яті, встановленою операційною системою Windows Seven, 8, 8.1, 10. Не менше 1024 Мб вільного місця на жорсткому диску. Двоядерний процесор з тактовою частотою від 1.2 GHz і більше.

2) Наявність встановленого .NET фреймворку версії 4.5 й більше.

3) Наявність встановленого NUnit Console Runner та Visual Studio Developer Console для запуску тестів із dll-файлів.

3.3 Інформаційна та програмна сполучність

Програмний продукт повинен коректно функціонувати в операційних системах Windows Seven, 8, 8.1, 10, на яких доступний для встановлення .NET фреймворк версії 4.5 й більше. Розроблювана система повинна бути

пристосована для автоматизованого тестування інформаційних систем. Розробку виконувати у середовищі розробки Visual Studio 2019 з використанням інструментів TestStack.White та NUnit 3 або MS Test 2.

4. СТАДІЇ РОЗРОБКИ

В ходів реалізації роботи проект повинен пройти крізь наступні стадії розробки:

- аналіз предметної області;
- аналіз важливості тестування у процесах розробки програмного забезпечення;
- аналіз автоматизованого тестування програмних систем;
- написання тестових випадків;
- проектування та розробка фреймворку автоматизованого тестування;
- проектування та розробка скриптів для автоматизованого тестування;
- оформлення супровідної документації;
- здача роботи.

5. ПРОГРАМНА ДОКУМЕНТАЦІЯ

Для програмного продукту повинні бути розроблені наступні документи:

- Пояснювальна записка;
- Технічне завдання;
- Презентаційний матеріал;
- Додатки.

6. ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Розроблений програмний продукт має виконувати всі вимоги, що складаються з перерахованих у п. 3.1 характеристик.

Приймання проводиться спеціально створеною екзаменаційною комісією в термін до:

“__” грудня 2020р.

Додаток Б
Лістинг алгоритму запуску та очистки тестів

```
[TestClass]
   public class UITestBaseUser
   {
      public TestContext TestContext { get; set; }

      public static ProcessHelper.ProcessMonitor monitor;

      [TestInitialize]
      public void TestStartUp()
      {
         CoreAppXmlConfiguration.Instance.Interceptors.Add(new
ActionLoggerInterceptor(
            new StreamWriter(Path.Combine(Directory.CreateDirectory(
               Path.Combine(TestContext.TestResultsDirectory,
TestContext.TestName, TestMethodExAttribute.Attempt.ToString())).FullName,
               ForteLogger.TestLogFileName))));

         DataBaseHelper.DropUserDatabase();
         DataBaseHelper.CopyDataBaseFile(FilePaths.pathToUserDataBaseFile,
FilePaths.userDestinationPath);

         var type = Type.GetType(TestContext.FullyQualifiedTestClassName);
         GetAndTrackEntityAttribute(type.GetMethod(TestContext.TestName));

         switch
(IsDatabaseUpdateSuppressed(type.GetMethod(TestContext.TestName)))
         {
            case true:
               break;

            case false:
               DataBaseHelper.UpdateForteDataBase(DatabaseType.User);
               break;

            default:
               DataBaseHelper.UpdateForteDataBase(DatabaseType.User);
               break;
         }

         switch (IsWordLaunchSuppressed(type.GetMethod(TestContext.TestName)))
         {
            case true:
```

```
                break;

            case false:
                WordActions.RelaunchWord();
                WordActions.CheckForDisabledAddin();

                HandleWordNotResponding(GetWordProcess());
                break;

            default:
                WordActions.RelaunchWord();
                WordActions.CheckForDisabledAddin();

                HandleWordNotResponding(GetWordProcess());
                break;
        }
    }

    [TestCleanup]
    public void TestCleanUp()
    {
        var type = Type.GetType(TestContext.FullyQualifiedTestClassName);

        switch (IsWordLaunchSuppressed(type.GetMethod(TestContext.TestName)))
        {
            case true:
                break;

            case false:
                monitor.UnsubscribeEvent();
                break;

            default:
                monitor.UnsubscribeEvent();
                break;
        }

        if (TestContext.CurrentTestOutcome.Equals(UnitTestOutcome.Failed))
            Reporter.TakeScreenshot(TestContext.TestResultsDirectory,
Path.Combine(TestContext.TestName,
TestMethodExAttribute.Attempt.ToString()));

        CoreAppXmlConfiguration.Instance.Interceptors.Clear();
        ForteLogger.LogCaughtExceptions();
```

```
            WordActions.CloseWord(WordActions.GetWordInstance().Application);
            if
(!Process.GetProcessesByName("WINWORD").FirstOrDefault().WaitForExit(180000))
                ProcessHelper.KillWordProcesses();

            DataBaseHelper.DropUserDatabase();
        }

        private void GetAndTrackEntityAttribute(MemberInfo memberInfo)
        {
            var copyCustomDataBaseAttribute =
memberInfo.GetCustomAttribute(typeof(CopyCustomDataBaseAttribute));

            if (copyCustomDataBaseAttribute != null)
            {
                var path =
copyCustomDataBaseAttribute.GetType().GetProperty("FilePath").GetValue(copyCustomDataBaseAttribute, null).ToString();

Enum.TryParse(copyCustomDataBaseAttribute.GetType().GetProperty("LoginType").GetValue(copyCustomDataBaseAttribute, null).ToString(),
                    out LoginType type);

copyCustomDataBaseAttribute.GetType().GetMethod("CopyDataBase").Invoke(new CopyCustomDataBaseAttribute(path, type), null);
            }
        }

        private bool IsDatabaseUpdateSuppressed(MemberInfo memberInfo)
        {
            var updateDBUserAttribute =
memberInfo.GetCustomAttribute(typeof(SuppressDatabaseUpdateAttribute));

            if(updateDBUserAttribute != null)
            {
                return true;
            }

            return false;
        }

        private bool IsWordLaunchSuppressed(MemberInfo memberInfo)
        {
```

```
            var supressWordLaunchAttribute =
memberInfo.GetCustomAttribute(typeof(SuppressWordLaunchAttribute));

            if (supressWordLaunchAttribute != null)
            {
                return true;
            }

            return false;
        }

        private static void HandleWordNotResponding(Process process)
        {
            try
            {
                monitor = new ProcessHelper.ProcessMonitor(process);
                monitor.NotResponding += (s, e) =>
                {
                    if (!process.HasExited)
                    {
                        RetryHelper.Instance.Try(() =>
                        {
                            if (!process.HasExited)
                            {
                                process.Refresh();
                            }
                        })
                        .WithTryInterval(3000)
                        .WithMaxTryCount(5)
                        .Until(() => process.Responding.Equals(true));
                    }
                };
                monitor.Start();
            }
            catch
            {
                Console.WriteLine("Process exited, moving on.");
            }

        }

        private static Process GetWordProcess()
        {
            return RetryHelper.Instance.Try(() =>
Process.GetProcessesByName("WINWORD").FirstOrDefault())
```

```
            .WithTryInterval(2000)
            .WithMaxTryCount(30)
            .UntilNoException();
    }
}
```