МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки**

Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

на тему:

ДОСЛІДЖЕННЯ ТА РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ ЗА НАКАЗОМ № 4601-5/3262 ВІД 15 ВЕРЕСНЯ 2025 РОКУ

здобувача вищої освіти **Гуржій Кирила Анатолійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ № 5 від 02.12.2025 р.

В.о. завідувача кафедри
PhD
_____ **Д. М. Ковальчук**

Науковий керівник
PhD
_____ **Д. М. Ковальчук**

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Н. І. Стяглик**
Підпис ініціали, прізвище

"15" вересня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Гуржій Кирила Анатолійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи "Дослідження та розробка програмних засобів автоматизованого тестування веб-застосунків"

керівник роботи Phd Ковальчук Дмитро Миколайович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "15" вересня 2025 року № 4601-5/3262

2. Строк подання студентом роботи 20 листопада 2025 року

3. Перелік питань, які потрібно розробити:
У розділі 1: охарактеризувати значення тестування у виробленні IT-продуктів.
У розділі 2: розглянути та оцінити засоби автоматизованого тестування.
У розділі 3: розробити програмні забезпечення для автоматизованого тестування веб-застосунків.
У розділі 4: здійснити дослідницьке порівняння дієвості розроблених засобів з існуючим тестуванням.

4. План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи керівника
5	Підпис кваліфікаційної магістерської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5. Дата видачі завдання 15 вересня 2025 року

Студент _____ К. А. Гуржій
підпис ініціали, прізвище

Керівник роботи _____ Д. М. Ковальчук
підпис ініціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ
«ДОСЛІДЖЕННЯ ТА РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ»
Гуржій Кирила Анатолійовича

Кваліфікаційна Магістерська робота містить 57 сторінок, 3 таблиці, 4 рисунки, список літератури з 20 найменувань.

Об'єктом дослідження є процес тестування веб-застосунків.

Предметом дослідження є програмні засоби та методи автоматизації тестування веб-застосунків.

Мета кваліфікаційної магістерської роботи полягає у дослідженні сучасних підходів та інструментів для автоматизованого тестування веб-застосунків, а також у розробці ефективного програмного засобу для здійснення цього тестування.

Завданнями кваліфікаційної магістерської роботи є:

- визначення ролі та важливості автоматизованого тестування в сучасному циклі розробки іт-продуктів;
- огляд та аналіз сучасних інструментів та фреймворків для автоматизованого тестування веб-застосунків;
- розробка програмного засобу (прототипу/компонента) для автоматизованого тестування веб-застосунків;
- експериментальна оцінка ефективності розробленого засобу шляхом порівняння його показників з існуючими методами та інструментами тестування.

Актуальність дослідження: стрімкий розвиток та зростаюча складність веб-застосунків вимагають впровадження ефективних та швидких методів контролю якості. Автоматизація тестування є ключовим елементом, що дозволяє скоротити час, знизити витрати, підвищити точність та покриття тестами, забезпечуючи безперервну інтеграцію та доставку (CI/CD).

За результатами дослідження: проведено аналіз сучасних підходів та розроблено програмні засоби для автоматизації тестування веб-застосунків, які підтвердили свою ефективність у порівнянні з традиційними та деякими існуючими автоматизованими методами.

Практична новизна полягає у дослідженні та використанні сучасних технологій для створення власного програмного засобу автоматизації, що враховує специфіку тестування сучасних веб-інтерфейсів, а також у науковому обґрунтуванні його переваг.

Одержані результати можуть бути використані ІТ-компаніями для оптимізації процесу розробки та тестування, підвищення якості кінцевого продукту та прискорення його випуску на ринок.

КЛЮЧОВІ СЛОВА: ВЕБ-ЗАСТОСУНКИ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ПРОГРАМНИЙ ЗАСІБ, ДОСЛІДЖЕННЯ, РОЗРОБКА.

ABSTRACT
AT QUALIFICATION MASTER'S WORK "RESEARCH AND DEVELOPMENT OF SOFTWARE FOR AUTOMATED TESTING OF WEB APPLICATIONS" HURZHII KYRYLO ANATOLIYOVYCH

The bachelor's thesis contains 57 pages, 3 tables, 4 drawings, a list of references of 20 titles.

The object of the research is the process of testing web applications.

The subject of the research is software and methods for automating testing of web applications.

The purpose of the qualification master's thesis is to study modern approaches and tools for automated testing of web applications, as well as to develop an effective software tool for carrying out this testing.

The objectives of the qualification master's thesis are:

- determining the role and importance of automated testing in the modern cycle of IT product development;
- review and analysis of modern tools and frameworks for automated testing of web applications;
- development of a software tool (prototype/component) for automated testing of web applications;
- experimental assessment of the effectiveness of the developed tool by comparing its indicators with existing testing methods and tools.

Relevance of the study: the rapid development and growing complexity of web applications require the implementation of effective and fast quality control methods. Test automation is a key element that allows you to reduce time, reduce costs, increase accuracy and test coverage, ensuring continuous integration and delivery (CI/CD).

According to the results of the study: an analysis of modern approaches was conducted and software tools were developed for automating web application testing, which have confirmed their effectiveness in comparison with traditional and some existing automated methods.

The practical novelty lies in the research and use of modern technologies to create our own automation software tool that takes into account the specifics of testing modern web interfaces, as well as in the scientific substantiation of its advantages.

The results obtained can be used by IT companies to optimize the development and testing process, improve the quality of the final product and accelerate its release to the market.

KEYWORDS: WEB APPLICATIONS, TESTING AUTOMATION, SOFTWARE, RESEARCH, DEVELOPMENT.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

CD – Continuous Delivery/Deployment (Безперервна доставка або Безперервне розгортання)

CI – Continuous Integration (безперервна інтеграція)

TDD – Test Driven Development, (розробка через тестування)

BDD – Behavior-Driven Development (розробка через поведінку)

WBS – Work Breakdown Structure (ієрархічна структура робіт)

QA – Quality Assurance (забезпечення якості)

QC – Quality Control (контроль якості)

QM – Quality Management (управління якістю)

SQuaRE – Systems and software Quality Requirements and Evaluation (вимоги та оцінювання якості систем і програмного забезпечення)

ПЗ – Програмне забезпечення

ISTQB – International Software Testing Qualification Board (міжнародна кваліфікована рада з тестування програмного забезпечення)

PMS – Project Management Systems(системи управління проектами)

DOM – Document Object Model (об'єктна модель документа)

POM – Page Object Model (модель об'єкта сторінки)

SPA – Single Page Applications (односторінковий застосунок)

ВСТУП

У XXI столітті, коли веб-застосунки стали ключовим елементом сучасної економіки та соціального життя, потреба у бездоганній якості програмного забезпечення є критичною. У цьому контексті автоматизація процесу тестування виступає як обов'язкова умова успішної розробки. Впровадження ефективних програмних засобів автоматизації дозволяє не лише гарантувати високий рівень надійності кінцевого продукту, але й суттєво прискорити цикл розробки та оптимізувати операційні витрати компанії.Автоматизовані системи тестування можуть допомогти забезпечити високу якість продукції, скоротити час виходу на ринок і зменшити витрати на розробку.

Актуальність даної роботи визначається невпинною динамікою ринку IT, де швидке виведення нових версій веб-застосунків і постійне оновлення їхньої функціональності є нормою. Цей високий темп розробки неминуче створює ризики, пов'язані з регресією (появою старих помилок) та загрозою стабільності системи. Ручне перевіряння цих частих змін є непрактичним і економічно невигідним.

Саме тому для забезпечення безперебійної функціональності та стійкості сучасних веб-застосунків необхідна безперервна розробка та вдосконалення методик і програмних інструментів для автоматизованого тестування.

Дослідження в галузі автоматизованого тестування веб-застосунків має значну практичну цінність не лише для фахівців із забезпечення якості (QA), але й для всіх підприємств, чий бізнес залежить від веб-технологій. Впровадження потужних рішень для автоматизації гарантує прогнозованість роботи веб-застосунків у реальному часі, мінімізує фінансові та репутаційні збитки від критичних помилок і, відповідно, підвищує лояльність користувачів.

З огляду на це, головним завданням цієї магістерської роботи є аналіз, порівняння та систематизація інноваційних підходів та програмних засобів, призначених для автоматизованого тестування веб-застосунків, з метою надання комплексного огляду для ефективного впровадження їх у процес розробки для підвищення як якості, так і швидкості випуску веб-продуктів.

РОЗДІЛ 1
ТЕСТУВАННЯ ЯК СКЛАДОВА ЖИТТЄВОГО ЦИКЛУ ВЕБ-ЗАСТОСУНКІВ

1.1. Концептуальні основи контролю якості програмного забезпечення

В епоху цифрової трансформації програмне забезпечення (ПЗ) стало критичним активом для більшості галузей економіки. Зростання складності веб-систем, перехід до мікросервісної архітектури та хмарних обчислень (Cloud Computing) суттєво підвищили ризики, пов'язані з відмовами програмних продуктів. У цьому контексті контроль якості трансформується з етапу пошуку помилок на комплексну інженерну дисципліну, що базується на міжнародних стандартах, метриках та імовірнісних моделях.

Поняття якості ПЗ є багатовимірним. Історично підходи до його визначення еволюціонували від моделі МакКолла (1977 р.) та Боема (1978 р.) до сучасних міжнародних стандартів серії ISO/IEC 25000 (SQuaRE — Systems and software Quality Requirements and Evaluation).

Згідно зі стандартом ISO/IEC 25010:2011, якість продукту декомпозується на вісім характеристик, кожна з яких має набір підхарактеристик. Для веб-застосунків критичними є:

1. Надійність (Reliability): включає доступність (Availability) та стійкість до помилок (Fault tolerance). Математично надійність часто описується через інтенсивність відмов λ(t) та середній час напрацювання на відмову (MTBF).

$$MTBF = \frac{\sum(Start\_time\_of\_downtime - Start\_time\_of\_uptime)}{Number\_of\_failutes}$$

2. Продуктивність (Performance Efficiency): Для веб-систем це критичний параметр, що вимірюється часом відгуку (Response Time) та пропускною здатністю (Throughput).

3. Зручність супроводу (Maintainability): Визначається тим, наскільки легко систему аналізувати, змінювати та тестувати. Саме низький рівень тестопридатності (Testability) часто стає бар'єром для впровадження автоматизації.

Науковий підхід вимагає чіткої диференціації рівнів управління якістю, що часто подається у вигляді ієрархічної моделі:

- Рівень 1: Quality Management (QM). Стратегічний рівень. Визначає політику якості організації, бюджетування та загальні цілі;
- Рівень 2: Quality Assurance (QA). Тактичний, процес-орієнтований рівень. Забезпечення якості — це сукупність планованих та систематичних дій, необхідних для створення впевненості у тому, що процес розробки відповідає стандартам (наприклад, CMMI або ISO 9001). Основні інструменти QA: аудит процесів, code review, статичний аналіз коду;
- Рівень 3: Quality Control (QC). Операційний, продукт-орієнтований рівень. Включає дії з вимірювання якості створених артефактів;
- Рівень 4: Testing. Технічна реалізація QC.

Згідно з програмою ISTQB (International Software Testing Qualifications Board), побудова ефективної системи тестування базується на семи принципах, які мають бути враховані при розробці автоматизованих засобів:

1. Тестування демонструє наявність дефектів, а не їх відсутність. Неможливо довести, що система на 100% безпомилкова (окрім тривіальних випадків);
2. Вичерпне тестування неможливе. Кількість комбінацій вхідних даних у сучасних веб-формах прагне до нескінченності, тому необхідне використання технік тест-дизайну (класи еквівалентності, граничні значення);
3. Раннє тестування. Вартість дефекту зростає нелінійно;
4. Скупчення дефектів (Defect Clustering). За принципом Парето, близько 80% помилок містяться у 20% модулів;

5. Парадокс пестициду. Якщо одні й ті самі тести виконувати багаторазово, вони перестають знаходити нові помилки. Це головний аргумент на користь постійної актуалізації наборів автоматизованих тестів;

6. Тестування залежить від контексту. Стратегія тестування e-commerce платформи кардинально відрізняється від тестування корпоративного порталу;

7. Омана про відсутність помилок. Виправлення всіх технічних помилок не гарантує успіху продукту, якщо він не задовольняє бізнес-потреби (проблема валідації).

1.2. Принципи та специфіка автоматизованого тестування

Автоматизоване тестування слід розглядати як розробку програмного забезпечення для тестування іншого програмного забезпечення. Це інженерний процес, що вимагає проектування архітектури тестового фреймворку, використання патернів проектування та систем контролю версій.

Ефективна автоматизація базується на пошаровій архітектурі тестів. Класична "Піраміда тестування" (Cohn's Pyramid) у сучасних реаліях доповнюється та модифікується, проте основний розподіл залишається актуальним:

1. Unit Tests (Модульні тести). Рівень "білого ящика". Тести пишуться розробниками на тій же мові, що й код продукту. Вони ізолюють залежності за допомогою Mock-об'єктів та Stubs.

- Ціль: перевірка алгоритмічної коректності.
- Обсяг: 60-70% від загальної кількості тестів.

2. Integration / Service Tests (API). Рівень "сірого ящика". Тестування REST або SOAP API, взаємодії мікросервісів, контрактне тестування (Contract Testing).

- Переваги: висока швидкість порівняно з UI, стабільність, незалежність від верстки.

3. UI / E2E Tests. Рівень "чорного ящика". Повна емуляція користувача.

- Проблема: "Крихкість" (Brittleness) тестів. Зміна ID елемента або CSS-класу призводить до падіння тесту (False Negative результат).

Впровадження автоматизації є інвестиційним проектом. Розрахунок ефективності (ROI) має враховувати не лише пряму економію часу, а й "ціну можливості" (Opportunity Cost) — час, який мануальні тестувальники можуть витратити на дослідницьке тестування (Exploratory Testing), поки працюють роботи.

Узагальнена формула ефективності автоматизації за період часу T:

$$E(T) = N * \left(C_{man} - C_{auto_{run}}\right) - (C_{dev} + C_{maint}(T)),$$

де:

- N — кількість запусків тестів.
- C_{man}— вартість одного ручного прогону.
- $C_{auto_{run}}$ — вартість машинного часу та аналізу результатів автотесту.
- C_{dev} — вартість розробки автотестів.
- $+C_{maint}(T)$ — вартість підтримки тестів (функція від часу та змін у проекті).

Якщо E(T) > 0, автоматизація є доцільною.

Сучасний веб (Web 2.0 і Web 3.0) створює специфічні проблеми для автоматизації:

1. Асинхронність DOM (Document Object Model). Single Page Applications (SPA) на React/Vue/Angular завантажують дані динамічно. Елементи з'являються на сторінці не одразу після завантаження (load event), а після виконання JavaScript. Це вимагає використання стратегій Smart Waits (розумних очікувань), замість фіксованих Hard Sleeps;

2. Shadow DOM. Технологія Web Components інкапсулює стиль та розмітку, роблячи елементи недоступними для стандартних селекторів XPath або CSS в деяких інструментах;

3. Стан та сесії. Автотести мають бути незалежними. Це вимагає механізмів очищення стану бази даних або LocalStorage браузера після кожного тесту (Teardown process), щоб уникнути ефекту "забруднення даних".

1.3. Сучасні моделі, підходи та інструментальні методики тестування веб-систем

Розвиток методологій DevOps та CI/CD (Continuous Integration / Continuous Delivery) призвів до появи концепції Continuous Testing (Безперервне тестування). Це передбачає запуск тестів на кожному етапі пайплайну розробки, від коміту коду до розгортання на продуктивному середовищі.

Для вирішення проблеми підтримки коду тестів використовуються спеціалізовані патерни:

- Page Object Model (POM): де-факто стандарт в UI-автоматизації. Кожна веб-сторінка описується як окремий клас, де методи класу відповідають діям користувача, а поля — веб-елементам. Це дозволяє відокремити логіку тесту від деталей реалізації сторінки;
- Screenplay Pattern: більш гнучка, SOLID-орієнтована альтернатива POM, що фокусується на діях користувача (Actors, Tasks, Interactions), а не на структурі сторінок;
- Fluent Interface: забезпечує читабельність коду тестів шляхом ланцюгового виклику методів (chaining).

Ринок інструментів автоматизації переживає зміну поколінь. Можна виділити три основні групи:

1. Інструменти на основі WebDriver (Selenium Ecosystem). Selenium залишається найбільш поширеним інструментом. Його архітектура базується на протоколі W3C WebDriver.

- Архітектура: Client Library ↔ JSON Wire Protocol ↔Browser Driver ↔Browser;
- Недоліки: Зайва ланка у вигляді драйвера створює затримки та проблеми зі стабільністю.

2. Інструменти прямої інтеграції (Chrome DevTools Protocol / Native).

Інструменти нового покоління (Cypress, Playwright, Puppeteer) працюють безпосередньо з браузером або через WebSocket з'єднання.

- Cypress: виконується безпосередньо у циклі (run loop) браузера разом з кодом застосунку. Це дає повний доступ до об'єктів DOM, window, LocalStorage, а також дозволяє керувати мережевими запитами (Network Stubbing/Spying) "на льоту";
- Playwright: розроблений Microsoft, використовує CDP (Chrome DevTools Protocol) для керування браузерами на рушіях Chromium, WebKit та Firefox. Підтримує паралельне виконання, автоматичне очікування та емуляцію мобільних пристроїв.

3. AI/ML-augmented Testing Tools. Новітній напрямок — використання штучного інтелекту для:

- Self-healing tests: автоматичне виправлення селекторів (локаторів) при зміні верстки (наприклад, інструмент Testim або Healenium);
- Visual Regression Testing: порівняння скріншотів з використанням алгоритмів комп'ютерного зору для ігнорування незначних змін (рендеринг шрифтів) та виявлення реальних візуальних багів (Applitools Percy).

Таблиця 1.1

Порівняльна характеристика підходів

Критерій порівняння	Selenium WebDriver	Cypress / Playwright
Архітектура	HTTP запити до драйвера	Виконання в браузері / WebSocket
Швидкість виконання	Середня	Висока
Стабільність (Flakiness)	Середня (вимагає налаштування waits)	Висока (вбудовані auto-waits)
Мови програмування	Java, C#, Python, JS, Ruby	Переважно JS/TS (Playwright також Java, Py, C#)
Робота з мережею	Обмежена (потрібні проксі)	Повний контроль (Mocking, Intercept)
Підтримка iFrame/Tabs	Повна	Обмежена в Cypress, повна в Playwright

РОЗДІЛ 2
ТЕХНОЛОГІЧНІ ЗАСОБИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

2.1. Порівняльний огляд інструментів та фреймворків для автоматизованого тестування веб-застосунків

Вибір інструментальних засобів для побудови системи автоматизованого тестування є критичним етапом, що визначає подальшу вартість підтримки проекту (maintenance cost), швидкість виконання тестів та можливість їх масштабування. Сучасний ринок пропонує широкий спектр рішень, які можна класифікувати за архітектурним підходом, підтримуваними мовами програмування та рівнем абстракції.

Для прийняття обґрунтованого рішення необхідно провести аналіз трьох домінуючих на ринку архітектурних парадигм: класичної (на базі WebDriver), сучасної (на базі протоколу CDP) та внутрішньобраузерної (In-Process).

2.1.1. Класифікація за архітектурним підходом

1. Архітектура на основі W3C WebDriver (Selenium) Це найбільш зрілий підхід, що став індустріальним стандартом. Selenium WebDriver працює за принципом клієнт-серверної архітектури.

- Механізм роботи: клієнт (тестовий скрипт на Python, Java тощо) відправляє HTTP-запити, що містять команди у форматі JSON, до драйвера браузера (наприклад, ChromeDriver). Драйвер інтерпретує ці команди та керує реальним браузером;
- Переваги: підтримка всіх основних браузерів та операційних систем, можливість емуляції мобільних пристроїв, величезна база знань та спільнота;

- Недоліки: наявність додаткової ланки (драйвера) створює затримки (latency). Оскільки команди передаються мережею, виникають проблеми синхронізації ("Race Conditions"), коли скрипт намагається взаємодіяти з елементом, який ще не відрендерився браузером.

2. Архітектура на основі Chrome DevTools Protocol (Playwright, Puppeteer) Інструменти нового покоління використовують WebSocket-з'єднання для прямої комунікації з браузером через налагоджувальний протокол (CDP).

- Переваги: висока швидкість виконання, доступ до мережевих запитів (Network Interception), можливість підписки на події браузера;
- Недоліки: обмеженіша підтримка браузерів (переважно сімейство Chromium), хоча Playwright вирішує цю проблему через патчинг рушіїв WebKit та Firefox.

3. Архітектура "In-Process" (Cypress) У цьому підході код тесту виконується безпосередньо всередині браузера в тому ж циклі виконання (run loop), що і JavaScript-код веб-застосунку.

- Переваги: миттєвий доступ до DOM-дерева та змінних застосунку, відсутність мережевих затримок;
- Недоліки: обмеження "пісочниці" браузера (складності з cross-origin testing), підтримка лише JavaScript/TypeScript, високе споживання пам'яті.

2.1.2. Аналіз екосистеми мови Python для автоматизації

Оскільки в рамках даної роботи як основну мову програмування обрано Python, доцільно розглянути специфіку інструментів саме в цій екосистемі. Python займає лідируючі позиції в Data Science та Machine Learning, що робить його ідеальним вибором для тестування складних систем, де потрібна генерація тестових даних або аналіз результатів.

Ключові переваги Python для QA:

- Лаконічність: скрипти на Python в середньому на 30-40% коротші за аналоги на Java, що спрощує підтримку коду;
- Бібліотека requests: дозволяє легко реалізовувати API-тести в рамках одного проекту з UI-тестами;
- Фреймворки BDD: Python має одну з найкращих реалізацій підходу Behavior-Driven Development — бібліотеку Behave.

2.1.3. Обґрунтування вибору зв'язки Selenium + Behave

Для реалізації практичної частини магістерської роботи було обрано технологічний стек Python + Selenium WebDriver + Behave. Цей вибір базується на наступних факторах:

1. Модульність та читабельність (BDD): використання фреймворку Behave дозволяє розділити опис бізнес-логіки (файли .feature мовою Gherkin) та технічну реалізацію (файли .py у папці steps). Це відповідає принципам чистої архітектури та полегшує комунікацію між технічними та нетехнічними учасниками процесу розробки. Як видно зі структури проекту, Behave забезпечує чітку ієрархію файлів;
2. Універсальність (Selenium): на відміну від Cypress, який обмежений JavaScript, або Playwright, який є відносно новим, Selenium забезпечує стабільну роботу з будь-якими браузерами та має повну відповідність стандартам W3C. Це важливо для академічної роботи, оскільки демонструє володіння фундаментальними технологіями;
3. Простота інтеграції: Python дозволяє легко інтегрувати тести з інструментами CI/CD (Jenkins, GitLab CI) та системами звітності (Allure Report). Віртуальне середовище (.venv), яке використовується в проекті, гарантує ізольованість залежностей та відтворюваність результатів тестування на будь-якій машині.

2.1.4. Порівняльна характеристика обраного рішення з аналогами

Нижче наведено порівняльну таблицю, що резюмує причини вибору поточного стека:

Таблиця 2.1

Порівняльна таблиця

Критерій	**Python + Behave + Selenium (Обрано)**	**JS + Cypress**	**Java + Selenium + Cucumber**
Мова опису тестів	Gherkin (Human-readable)	JavaScript Code	Gherkin
Швидкість розробки	Висока (простий синтаксис Python)	Висока (все в одному)	Середня (багатослівний код)
Керування браузером	Через WebDriver (стандарт)	Пряма ін'єкція в DOM	Через WebDriver
Підтримка спільноти	Дуже висока	Висока (зростає)	Дуже висока (Enterprise стандарт)
Складність налаштування	Низька (pip install behave)	Низька (npm install)	Середня (Maven/Gradle config)

2.2. Проєктування та структурування тестових сценаріїв у рамках автоматизованих процесів

Проєктування архітектури системи автоматизованого тестування є фундаментальним для її довгострокової життєздатності. Це досягається завдяки дотриманню інженерних принципів модульності та розділення відповідальності (Separation of Concerns). В основу структури покладено методологію BDD (Behavior-Driven Development) на рівні опису та патерн Page Object Model (POM) на рівні реалізації.

BDD є ітеративним процесом розробки програмного забезпечення, що є розширенням TDD (Test-Driven Development). Його основна мета — забезпечення чіткої комунікації між технічними та нетехнічними учасниками проекту шляхом використання універсальної, наближеної до природної, мови.

BDD (Behavior-Driven Development) — це процес, який дозволяє командам створювати спільне розуміння того, що повинно бути реалізовано, шляхом узгодження функціональних вимог у форматі, зрозумілому як розробникам, так і бізнес-аналітикам. Ключовим елементом є використання Gherkin — доменно-специфічної мови для опису поведінки системи у вигляді Сценаріїв (Scenario) з кроками Дано-Коли-Тоді (Given-When-Then).

- Рівень бізнес-специфікацій (features/): файли .feature мовою Gherkin виступають як виконавча документація;
  - Дано (Given): описує початковий стан системи (передумови);
  - Коли (When): описує дію, яку виконує користувач або система;
  - Тоді (Then): описує очікуваний результат або зміну стану системи.
- Рівень реалізації кроків (features/steps/): фреймворк Behave виконує роль "клею", використовуючи декоратори (@given, @when, @then) для динамічного зв'язування Gherkin-тексту з відповідними функціями Python.

Для підвищення стійкості (resilience) та ремонтопридатності (maintainability) тестового коду використовується інженерний патерн Page Object Model (POM).

Page Object Model (POM) — це патерн проектування, метою якого є створення об'єктного сховища (object repository) для елементів інтерфейсу користувача (UI). У рамках POM, кожна унікальна сторінка веб-застосунку (або її значний фрагмент) моделюється як окремий програмний клас. Цей патерн забезпечує інкапсуляцію локаторів та методів взаємодії, відокремлюючи їх від логіки тестів.

Структура реалізації POM:

- Директорія pages/: містить класи сторінок (наприклад, LoginPage, ProductPage);
- Клас сторінки: кожен клас містить:

- Константи локаторів: зберігаються у вигляді кортежів (By.TYPE, "locator_value");

- Методи взаємодії: абстрагують дії користувача (наприклад, метод LoginPage.perform_login(user, pass) замість 3-4 прямих викликів driver.find_element у тестовому кроці).

Якість тесту безпосередньо залежить від стійкості локаторів. Для досягнення детермінованості тестів використовується ієрархія пріоритетів:

1. Унікальні атрибути: By.ID або спеціальні атрибути data-test-id. Вони є ідеальними, оскільки не залежать від візуального стилю чи позиції елемента;

2. CSS-селектори: By.CSS_SELECTOR забезпечує високу швидкість пошуку та є більш стійким, ніж XPath;

3. XPath: використовується лише для складних випадків (наприклад, навігація між батьківськими елементами або пошук за текстом), уникаючи абсолютних шляхів, які роблять тест "ламким" (fragile).

Розроблені сценарії (Smoke Tests та End-to-End) покривають критичний функціонал та демонструють використання інженерних патернів.

Розроблені сценарії покривають критичний функціонал веб-застосунку (Smoke Testing та Critical Path) і використовують Data-Driven Testing (DDT) через конструкцію Scenario Outline.

СЦЕНАРІЙ 1: Додавання товару до списку бажань (Wishlist)

Даний сценарій перевіряє інтеграцію пошуку та каталогу.

Код сценарію (Gherkin):

Gherkin

```
Scenario Outline: Check add product to wishlist
  Given User opens '<homePage>' page
  And User checks search field visibility
  When User makes search by keyword '<keyword>'
  And User clicks search button
  And User clicks wish list on first product
  Then User checks that amount of products in wish
list are '<expectedAmount>'
```

```
Examples:
  | homePage              | keyword | expectedAmount
|
  | https://myshop.com/ | cake    | 1               |
```

Алгоритмічний опис (Логіка виконання):

1. Ініціалізація: браузер переходить на <homePage>.

2. Валідація елемента: скрипт застосовує явне очікування (WebDriverWait) для перевірки появи елемента <input type="search">;

3. Дія: Емуляція введення тексту (keyword) та натискання кнопки пошуку.

4. Вибір об'єкта: скрипт ідентифікує перший товар (.product-item:first-child) та виконує подію click на іконці Wishlist;

5. Перевірка стану: зчитується текстове значення лічильника Wishlist (наприклад, span.wishlist-count) та порівнюється з очікуваним значенням (expectedAmount).

СЦЕНАРІЙ 2: Валідація авторизації користувача

Сценарій перевіряє механізм автентифікації та керування сесіями.

Код сценарію (Gherkin):

Gherkin

```
Scenario: User logs in with valid credentials
  Given User navigates to login page
  When User enters username "testuser_diploma"
  And User enters password "securePass123"
  And User clicks the Login button
  Then User should be redirected to the Dashboard
  And The generic welcome message "Hello, User"
should be visible
```

Алгоритмічний опис (Логіка виконання):

1. Введення даних: скрипт знаходить поля вводу та імітує введення валідних облікових даних;

2. Відправка форми: Натискання кнопки submit;

3. Перевірка URL: скрипт перевіряє, що поточна адреса браузера змінилася на /dashboard;

4. Перевірка UI: перевіряється наявність елемента з привітанням, що підтверджує успішну сесію.

СЦЕНАРІЙ 3: Фільтрація та сортування товарів

Сценарій тестує динамічне оновлення контенту (AJAX) та цілісність даних.

Код сценарію (Gherkin):

Gherkin

```
Scenario: Filter products by price range
  Given User is on the "Laptops" category page
  When User applies price filter from "1000" to "2000"
  And User waits for the product grid to reload
  Then All displayed products should have price within range "1000" to "2000"
```

Алгоритмічний опис (Логіка виконання):

1. Дія: встановлення числових значень у фільтри "Ціна від" та "Ціна до";

2. Синхронізація: використовується явне очікування (Explicit Wait), що очікує зникнення елемента-індикатора завантаження ("spinner") або зміни вмісту блоку результатів;

3. Ітеративна перевірка: скрипт отримує список усіх цін на сторінці. Ціни конвертуються у числові типи (float), і в циклі перевіряється, чи відповідає кожне значення бізнес-правилу: $1000 \le Price \le 2000$.

СЦЕНАРІЙ 4: Перевірка кошика (End-to-End Flow)

Комплексний сценарій, що перевіряє логіку обчислення вартості.

Код сценарію (Gherkin):

Gherkin

```
Scenario Outline: Validate cart calculation
  Given User opens product page for '<product_id>'
  When User selects size '<size>'
  And User clicks "Add to Cart"
  Then The mini-cart popup should appear
  And The subtotal should allow equal '<price>'

Examples:
  | product_id | size | price  |
```

sku_12345	M	50.00
sku_67890	L	75.00

Алгоритмічний опис (Логіка виконання):

1. Конфігурація: вибір атрибутів товару (розмір, колір) через UI-елементи;
2. Транзакція: натискання "Додати в кошик";
3. Перевірка UI: перевірка появи спливаючого вікна кошика (Popup/Modal);
4. Фінансова перевірка: зчитується підсумкова вартість у кошику. Перевіряється точність обчислень та відповідність ціні, вказаній у таблиці параметризації.

РОЗДІЛ 3
РОЗРОБЛЕННЯ ПРОГРАМНОГО КОМПЛЕКСУ ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ

3.1. Конфігурація середовища для запуску автоматизованих тестів

Створення ефективного комплексу автоматизованого тестування є критичним інвестиційним рішенням у забезпеченні якості будь-якого сучасного веб-застосунку. Основний підхід до цього етапу не лише як до програмування, а як до інженерного проектування системи, яка має бути гнучкою, масштабованою та підтримуваною протягом усього життєвого циклу продукту. Фундаментальним вибором стала методологія Behavior-Driven Development (BDD), яка дозволяє не просто перевіряти код, а перевіряти відповідність поведінки системи бізнес-вимогам. Це забезпечує наскрізну прозорість: від початкової ідеї до фінального тесту.

Найголовніше те, що надійність результатів тестування починається з надійності тестового середовища. Першочерговим завданням було створення контрольованого та універсального робочого простору, де будь-який розробник чи тестувальник міг би відтворити результати роботи.

Вибір технологій для автоматизації – це завжди компроміс між швидкістю розробки, кривою навчання та потужністю інструментів.

Було вирішено зупинитися на Python 3.x з кількох ключових причин. По-перше, його інтуїтивний синтаксис значно знижує когнітивне навантаження, дозволяючи зосередитися на логіці тесту, а не на деталях мови. По-друге, Python має найбільш розвинену та стабільну екосистему бібліотек для тестування, що дає доступ до великої кількості готових рішень.

Щодо фреймворків, обрано наступну зв'язку:

- Behave як BDD-оркестратор: Behave служить центральною точкою керування. Він приймає специфікації у форматі Gherkin і послідовно викликає відповідні функції Python. Це забезпечує чітке розділення

відповідальності (Separation of Concerns): бізнес-логіка живе у файлах `.feature`, а технічна – у файлах `_steps.py`;

- Selenium WebDriver як інтерфейс користувача (UI) симулятор: Selenium є галузевим стандартом для тестування веб-інтерфейсів. Він використовується для емуляції реальної людської взаємодії: прокручування, кліки, подвійні кліки, перетягування, введення тексту, що є критичним для функціонального тестування. Selenium дає впевненість, що тестується не просто код, а користувацький досвід.

Вважається, що робота над будь-яким проектом має починатися зі створення віртуального середовища (`.venv`). Це не просто гарна практика, це життєва необхідність для відтворюваності. Віртуальне середовище гарантує, що версії `behave` і `selenium`, які використовуються в роботі, не конфліктуватимуть з іншими проектами на моїй машині чи машинах користувачів.

Після створення ізольованого простору, встановлено ключові залежності за допомогою `pip`:

1. `pip install behave`: встановлення BDD-фреймворку;
2. `pip install selenium`: встановлення бібліотеки для веб-автоматизації.

Окрему увагу було приділено налаштуванню браузерного драйвера. Браузерний драйвер (наприклад, `chromedriver`) — це мережевий проксі-сервер, який дозволяє скриптам Selenium спілкуватися з браузером через протокол WebDriver W3C. Завжди потрібно контролювати, щоб версія драйвера точно відповідала версії браузера, оскільки несумісність є найпоширенішою причиною непередбачуваних збоїв у UI-тестах.

У кроці `@given("User opens '{homePage}' page")` відбувається ініціалізація сесії. Об'єкт `Options()` застосовується для налаштування параметрів запуску браузера. Зокрема, директива `options.add_argument("--start-maximized")` є обов'язковою,

оскільки розмір вікна браузера може впливати на розташування елементів (адаптивний дизайн), а отже, на їхню видимість та інтерактивність. Якщо елемент невидимий, Selenium не зможе з ним взаємодіяти, що призведе до помилки `ElementNotVisibleException`. Тому потрібно уникати цього, максимізуючи вікно.

```
@given("User opens '{homePage}' page")
def step_open_home(context, homePage):
    options = Options()
    options.add_argument("--start-maximized")
    context.driver = webdriver.Chrome()
    context.driver.maximize_window() #відкрити на весь екран
    context.driver.get(homePage) #гугл драйвер відкриває посилання яке я йому прописав
```

Рис.3.1. Демонстрація коду де прописана ініціалізація

Екземпляр драйвера зберігається у `context.driver`, роблячи його доступним для всіх подальших кроків у межах поточного сценарію.

3.2. Архітектура тестувального комплексу

Розроблена архітектура є дворівневою, що є типовим для BDD-проектів, які використовують Behave. Це забезпечує гнучкість і високу підтримуваність.

Було обрано BDD, оскільки це методологія, яка сприяє співпраці (collaboration). Тести, які я пишу, є формальними специфікаціями вимог.

Специфікація через приклади (Specification by Example): Я використовую конструкції Gherkin, щоб створювати сценарії, які відображають очікування бізнесу щодо поведінки системи. Це допомагає мені уникнути неоднозначності у вимогах.

Проект складається з двох основних шарів, які взаємодіють між собою:

- Специфікаційний шар (`features`): це "ЩО" (що потрібно протестувати).

- Виконавчий шар (`steps`): це "ЯК" (як це буде виконано технічно).

Це розділення гарантує, що якщо зміниться лише технічна реалізація (наприклад, я перейду на інший локатор), мені не доведеться змінювати сам сценарій Gherkin. І навпаки, якщо зміниться бізнес-вимога, мені не потрібно переписувати весь код, а лише модифікувати відповідну функцію.

У файлі `smoke.feature` визначено Feature (особливість), яка є високорівневим описом функціоналу ("Smoke Testing"). Потім було створено Scenario Outline (шаблон сценарію) для перевірки додавання продукту до списку бажань.

Використання `Scenario Outline` з блоком `Examples` є ключовим архітектурним рішенням. Це дозволяє досягти дата-драйвен тестування (Data-Driven Testing). Замість написання п'яти ідентичних сценаріїв для п'яти різних ключових слів пошуку, я пишу один шаблон і п'ять рядків у таблиці. Це значно зменшує надмірність коду (code redundancy) та спрощує підтримку.

```
@smoke
Scenario Outline: Check add product to wishlist
  Given User opens '<homePage>' page
  And User checks search field visibility
  When User makes search by keyword '<keyword>'
  And User clicks search button
  And User clicks wish list on first product
  Then User checks that amount of products in wish list are '<expectedAmount>'

  Examples:
    | homePage                          | keyword | expectedAmount |
    | https://www.canadiantire.ca/en.html | cake    | 1              |
```

Рис. 3.2. Сценарій тесту

В ролі використовуються Теги (`@smoke`) для організації. Теги — це метадані, які дозволяють легко фільтрувати та запускати лише необхідні підмножини тестів.

Об'єкт `context` від Behave є одноразовим сховищем даних, яке існує протягом усього життєвого циклу одного сценарію. В ньому зберігаються посилання на об'єкт `driver`, що дозволяють викликати `context.driver.find_element` у будь-якому кроці, який слідує за ініціалізацією. Це забезпечує безперервність сесії браузера між кроками `Given`, `When` та `Then`.

3.3. Програмна реалізація тестувального комплексу

Програмна реалізація — це перетворення декларативних Gherkin-кроків у імперативні команди Selenium, які виконуються у браузері.

У реалізації коду віддано перевагу стратегії XPath (`By.XPATH`). Хоча CSS-селектори можуть бути швидшими, XPath надає більшу експресивність та гнучкість. Він дозволя:

- використовувати текстовий вміст елемента для локалізації (наприклад, `//button[text()='Search']`);
- здійснювати пошук назад (ancestor/parent), що важливо, коли елемент, який шукають, не має унікального ідентифікатора, але його батьківський елемент має.

Наприклад, у кроці `step_check_search_field`, не лише локалізується поле пошуку, але й додається важливий крок для підвищення стабільності тесту.

```python
@given("User checks search field visibility")
def step_check_search_field(context):
    # Знаходжу поле і перевіряю його видимість
    context.search_field = context.driver.find_element(By.XPATH, '//input[@placeholder="Search"]')
    # Обробка перешкод (рекламних банерів)
    # Це є прикладом "захисного програмування" для стабільності автоматизації
    assert context.search_field.is_displayed(), "Search field is not visible"
    close_advertise_button = context.driver.find_element(By.XPATH, '//button[@class="CTR_loyalty-offer-close-button"]')
    close_advertise_button.click()
```

Рис.3.3. Реалізація перевірки на видимість та закриття рекламного банера

Кроки `@when` відповідають за активну взаємодію:

- `send_keys(keyword)`: 2е імітація набору тексту користувачем;
- `click()`: імітація натискання кнопки миші.

Особлива увага приділена кроку `@when("User clicks wish list on first product")`. Було використано жорстку паузу (`time.sleep(3)`). Хоча я знаю, що явні очікування (`WebDriverWait`) є кращими, оскільки вони динамічно чекають, time.sleep було використано для гарантованої фіксації видимості елемента у даному конкретному простому Smoke-тесті, оскільки додавання в список бажань може викликати асинхронну операцію на сервері, яка не завжди завершується зміною стану DOM.

Крок `@then` є точкою прийняття рішення у сценарії. Реалізувавши його як твердження (Assertion):

1. локалізуємо лічильник, використовуючи його унікальний клас: `//span[@class="nl-wishlist-badge"];`
2. отримуємо фактичне значення та очищуємо його від зайвих пробілів (`.strip()`);
3. порівнюємо його з очікуваним значенням, яке надійшло з таблиці `Examples`.

Якщо `assert` не виконується, тест миттєво припиняється, і отримуємо недвозначне повідомлення про помилку, яке вказує, що функціональна поведінка порушена.

Важливо: я завжди гарантую, що сесія WebDriver завершується викликом `context.driver.quit()`. Це критично важливо для запобігання витоку пам'яті та залишенню "зомбі-процесів" браузера, що може призвести до нестабільності системи.

3.4. Проведення автоматизованого тестування функціональних можливостей веб-застосунку

Тестовий сценарій виконується, використовуючи Інтерфейс Командного Рядка (CLI), що є ключовим для інтеграції в автоматизовані системи, такі як Continuous Integration/Continuous Deployment (CI/CD). Використання CLI робить комплекс незалежним від графічного інтерфейсу розробника.

Застосовується команда:

```
behave --tags=@smoke
```

Ця цільова команда дозволяє досягти селективного виконання тестів, що є критично важливим для Smoke-тестування. У реальних умовах розробки, можно налаштувати CI-сервер (наприклад, Jenkins або GitLab CI) на запуск саме цієї команди після кожного злиття (merge) коду у головну гілку. Це забезпечує мені миттєвий зворотний зв'язок про стан критичного функціоналу.

Під час виконання відбувається наступний ланцюг подій:

1. Створення сесії: команда викликає Behave, який запускає `chromedriver` і відкриває екземпляр браузера, ініціюючи HTTP-зв'язок між моїм кодом і браузером через протокол WebDriver;

2. Диспетчеризація: Behave діє як диспетчер, викликаючи функції з `smoke_steps.py` відповідно до послідовності кроків Gherkin. Наприклад,

крок `When User makes search by keyword 'cake'` викликає метод `send_keys()`, а не просто виконує операцію введення тексту. Це імітує людський фактор;

3. Синхронізація: у кроці додавання товару до списку бажань застосовується жорстка пауза (`time.sleep(3)`). Хоча в ідеалі замінити її на Явні Очікування (Explicit Waits), використання `time.sleep` на цьому етапі є гарантією, що асинхронна JavaScript-операція, яка оновлює лічильник, матиме достатньо часу для завершення. Це тимчасове рішення, яке в планах на оптимізацію при переході на Page Object Model.

Для наочної демонстрації цього процесу виконання, відразу після наведеної команди, я вважаю за необхідне розмістити Малюнок 3.4.1. Ця ілюстрація покаже, як Behave покроково виводить статус кожного кроку (позначки `pass` або `fail`) у режимі реального часу, підтверджуючи, що процес автоматизації відбувається коректно.

```
(.venv) PS C:\Users\Power\PycharmProjects\PythonProject> behave --tags=@smoke
USING RUNNER: behave.runner:Runner
Feature: Smoke # features/smoke.feature:1
  As a userI
  I want to test all main site functionality
  So that I can be sure that site works correctly
  @smoke
    Then User checks that amount of products in wish list are '1' # features/steps/smoke_steps.py:43 2.328s
    Then User checks that amount of products in wish list are '1' # features/steps/smoke_steps.py:43
1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
6 steps passed, 0 failed, 0 skipped
Took 0min 37.872s
(.venv) PS C:\Users\Power\PycharmProjects\PythonProject>
```

Рис.3.4. Скріншот виконання тесту в консолі

Якщо сценарій завершується зі статусом Passed, це є однозначним підтвердженням валідності тестованого функціоналу.

- Технічна валідація: це означає, що всі стратегії локалізації елементів (XPath) були коректними, і веб-додаток продемонстрував

очікувану поведінку. Selenium не зафіксував жодних винятків, таких як `NoSuchElementException` або `StaleElementReferenceException`;

- Бізнес-валідація: це найважливіше — успіх означає, що бізнес-вимога "Користувач може додати товар до списку бажань" виконана. Твердження у кроці `@then` підтвердило, що лічильник бажань коректно збільшився до '1', що прямо корелює з очікуваннями кінцевого користувача.

Збій тесту є не провалом, а цінною інформацією. Завдяки BDD-структурі, я можу швидко діагностувати корінь проблеми:

1. Дефект додатку (Application Defect): якщо збій стався у фінальному кроці `@then` (наприклад, `AssertionError: Expected 1, but got 0`), це прямо вказує на проблему у бізнес-логіці програми. Код автоматизації довів, що програма не працює, як очікувалося;

2. Помилка автоматизації (Automation Flaw): збій на кроках `@given` або `@when` часто означає, що розробники змінили DOM-структуру, і XPath-локатор став недійсним. Це вимагає швидкого оновлення коду, але не вказує на дефект програми.

Behave надає детальний Steptrace (трасування кроків), який точно вказує на файл, рядок коду та виняток Python. Ця подвійна звітність (бізнес-опис і технічний збій) є найбільшою перевагою BDD, оскільки вона значно прискорює процес ідентифікації та виправлення дефектів.

Розроблений комплекс є гнучким і має чіткий план розвитку для забезпечення його довговічності та економічної ефективності.

Планується реалізувати Page Object Model (POM). Цей патерн є необхідним для мінімізації технічного боргу, оскільки він відокремлює логіку взаємодії з елементами (методи) від їхніх локаторів. Замість того, щоб шукати XPath у десятках `_steps.py` файлів, я матиму єдину точку входу (клас сторінки) для будь-якої модифікації елементів.

Крім того, для підвищення швидкості виконання, буде замінено всі жорсткі `time.sleep` на Явні Очікування (Explicit Waits). Наприклад, `WebDriverWait` буде чекати лише до моменту, коли лічильник стане видимим або його значення зміниться, а не фіксований час. Це значно підвищить надійність (resilience) тестів і скоротить загальний час прогону тестового набору.

РОЗДІЛ 4
ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ПРОГРАМНОГО КОМПЛЕКСУ ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ

4.1. Аналіз експериментальних результатів роботи програмного засобу

Цей розділ присвячений емпіричній валідації розробленого програмного комплексу. Головною метою даного дослідження є не просто підтвердження працездатності скриптів, а кількісна оцінка їхньої ефективності порівняно з традиційними методами забезпечення якості. Дослідження має на меті довести, що інвестиції в автоматизацію призводять до значного скорочення часу тестування, підвищення надійності та, як наслідок, до кращої економічної доцільності проекту. Оцінювання ґрунтується на вимірюванні двох ключових метрик: швидкість виконання та фактор повторюваності.

Для проведення об'єктивного аналізу було створено контрольоване експериментальне середовище. Увагу зосереджено на вимірюванні загального часу виконання (Total Execution Time) сценарію "Smoke Test" від моменту виклику команди в консолі до повного завершення сесії браузера (driver.quit()). Це вимірювання включає ініціалізацію драйвера, завантаження сторінки, виконання всіх кроків взаємодії (пошук, кліки) та фінальне твердження. Цей час є реальним часом, який економиться при автоматизації.

Було прийнято рішення провести вимірювання на двох різних апаратних платформах (Windows-ноутбук та macOS-MacBook), що дозволяє не лише отримати середньозважені показники, але й підтвердити крос-платформну стійкість комплексу, який базується на незалежних від ОС технологіях (Python та WebDriver).

Для забезпечення максимальної достовірності результатів, було дотримано суворих правил контролю:

1. Ізоляція ресурсів: під час вимірювань були відключені всі фонові програми та процеси, які могли б споживати ресурси CPU або RAM, щоб мінімізувати вплив стороннього навантаження на тестову систему;

2. Аналіз впливу мережевої затримки: Усі тести проводилися в різних мережевих умовах. Такий підхід був застосований для того, щоб оцінити стійкість програмного комплексу та варіативність часу виконання в умовах, які можуть бути спричинені реальними зовнішніми факторами (наприклад, затримкою завантаження елементів з Інтернету). Це дозволяє визначити діапазон можливих результатів у динамічному робочому середовищі.

3. Статистична значущість: кожен сценарій був повторений п'ять разів на кожній машині. Це дозволяє уникнути викидів (outliers) і розрахувати достовірне середнє арифметичне значення часу виконання.

Вимірювання часу виконання відбувалося за допомогою вбудованих функціональних можливостей фреймворку Behave та модуля time у Python, що дозволяє отримати точний час початку та завершення виконання на рівні мілісекунд.

Оскільки швидкість виконання автоматизованого тесту значною мірою залежить від потужності процесора та швидкості дискової системи (завантаження WebDriver, ініціалізація браузера), для повної прозорості надаються детальні характеристики обладнання. Різниця в часі виконання між Windows та macOS може бути також викликана особливостями їхніх файлових систем та методами обробки процесів.

Macbook Pro M4 Pr):

- os.name: 'Mac OS X (Tahoe)’, os.arch: 'aarch64', os.version: '26.1';
- java.version: '25.0.1’;
- CPU: M4 Pro (14-Core): (10 Performance + 4 Efficient);
- GPU: Integrated GPU - Apple M4 GPU (20-core);
- RAM: 48GB, LPDDR5X, 8533 MHz;
- Disk: NVME M2 SSD - 1024 Gb;

- Network - 1 Gbps (real ~800 Mbps).

ASUS TUF Gaming F15 FX506HC:

- os.name: Windows 11 Домашня;
- java.version: '25.0.1';
- CPU: 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (2.69 GHz);
- GPU:

● Дискретная: NVIDIA GeForce RTX 3050 для ноутбуков, 4 ГБ GDDR6;

● Интегрированная: Intel UHD Graphics;

- RAM: 16 ГБ, DDR4, 3200 МГц;
- Disk: NVMe M.2 SSD - 512 ГБ;
- Network - (real ~10 Mbps);

Отримані емпіричні дані про час виконання автоматизованого Smoke-тесту (у секундах) на різних платформах було зібрано у порівняльну таблицю. Крім кількісних показників швидкості, було включено якісні метрики, які відображають переваги BDD-підходу.

Ключовою метрикою для оцінки ефективності є Фактор економії часу. Його розрахунок здійснюється як співвідношення між часом, витраченим на ручне виконання, та часом, витраченим на автоматизоване виконання.

$$\text{Фактор економії часу} = \frac{\text{Середній час ручного тестування}}{\text{Середній час автоматизованого тестування}}$$

Цей фактор є прямим показником потенціалу автоматизації для звільнення робочого часу тестувальника.

Порівняльна таблиця є центральним елементом цього розділу, оскільки вона надає прямий кількісний доказ ефективності.

Таблиця 4.1.

Порівняльна таблиця програмного тестування з традиційним ручним тестуванням

Показник	Ручне тестування (База)	Автоматизоване (Python/Windows)	Автоматизоване (Python/macOS)	Автоматизоване (Java/Windows)	Автоматизоване (Java/macOS)
Середній час (1 прогін, сек)	33 сек	25.626 сек	5.793 сек	32.154 сек	7.775 сек
Середній час (5 прогонів, сек)	30 сек	25 сек	6 сек	29 сек	7 сек
Фактор економії часу (ROI)	1.0 сек	1.2 сек	5 сек	1.03 сек	4.3 сек
Зручність створення/читання	Висока (Інструкції)	Висока (BDD/Gherkin)	Висока (BDD/Gherkin)	Середня (Синтаксис Java)	Середня (Синтаксис Java)
Точність виконання	Середня (Суб'єктивність)	100% (Детермінізм)	100% (Детермінізм)	100% (Детермінізм)	100% (Детермінізм)
Розмір коду (Підтримуваність)	Низький (Простий текст)	Приблизно 60 рядків якщо не враховувати помітки	Приблизно 60 рядків якщо не враховувати помітки	5 вкладок з приблизною суммою в 200 рядків	5 вкладок з приблизною суммою в 200 рядків
Можливість інтеграції (CI/CD)	Ні (Потребує людини)	Так (Повна)	Так (Повна)	Так (Повна)	Так (Повна)
Крос-платформна перевірка	Потребує двох тестувальників	Здійснюється однією системою	Здійснюється однією системою	Здійснюється однією системою	Здійснюється однією системою

4.2. Порівняння автоматизованих тестів з традиційним ручним тестуванням

Аналіз отриманих емпіричних даних дозволяє перейти до ключової оцінки — порівняння розробленого програмного комплексу (Python/Behave) з традиційним ручним підходом та порівняння з альтернативним технологічним стеком (Java/Selenium). Це порівняння охоплює не лише показники швидкості, але й такі критичні фактори, як надійність, точність та економічна ефективність, що були відображені у розширеній таблиці.

Як видно з представлених даних, автоматизований комплекс демонструє значне прискорення виконання вже на першому прогоні. Слід підкреслити, що час ручного тестування включає когнітивні витрати (час на читання інструкції, концентрацію уваги, зміну контексту), які є невід'ємною частиною людської роботи. Ці витрати повністю виключені в автоматизації.

Справжня ефективність розкривається при багаторазовому виконанні. Порівняння загального часу, необхідного для виконання тестового сценарію 10 разів, чітко показує, що ручний тестувальник витрачає лінійний час ($10 \times T_{ручне}$), тоді як комплекс виконує завдання за частку цього часу. Це обґрунтовує необхідність автоматизації для регресійного тестування.

Порівняння швидкості між Python та Java

Аналіз продуктивності, що охоплює два різних стеки, виявляє фундаментальну різницю у швидкості ініціалізації. Сценарії, написані на Java, можуть мати незначну перевагу у швидкості виконання чистої логіки завдяки компіляції Just-In-Time (JIT) порівняно з інтерпретацією Python. Однак, ініціалізація віртуальної машини Java (JVM) часто займає більше часу, ніж запуск інтерпретатора Python. Це означає, що для коротких Smoke-тестів (як у цьому дослідженні), час запуску може домінувати над часом виконання, надаючи Python перевагу у загальній швидкості.

Вплив апаратної платформи

Різниця в часі між Windows та macOS на одному стеку (наприклад, Python/Windows проти Python/macOS) відображає особливості планування процесів (process scheduling) та управління ресурсами операційних систем. Як правило, системи на базі Unix (macOS) більш ефективно керують короткими I/O-операціями, що може позитивно впливати на швидкість завантаження сторінок та реакцію WebDriver. Слід зазначити, що ці варіації є незначними у порівнянні з розривом між ручним та автоматизованим тестуванням.

Візуалізація переваг

Найбільш переконливим аргументом на користь автоматизації є порівняння загального часу, необхідного для виконання тестового сценарію 10 разів. Для візуалізації цього критичного співвідношення між часом і кількістю прогонів, застосовується гістограма.

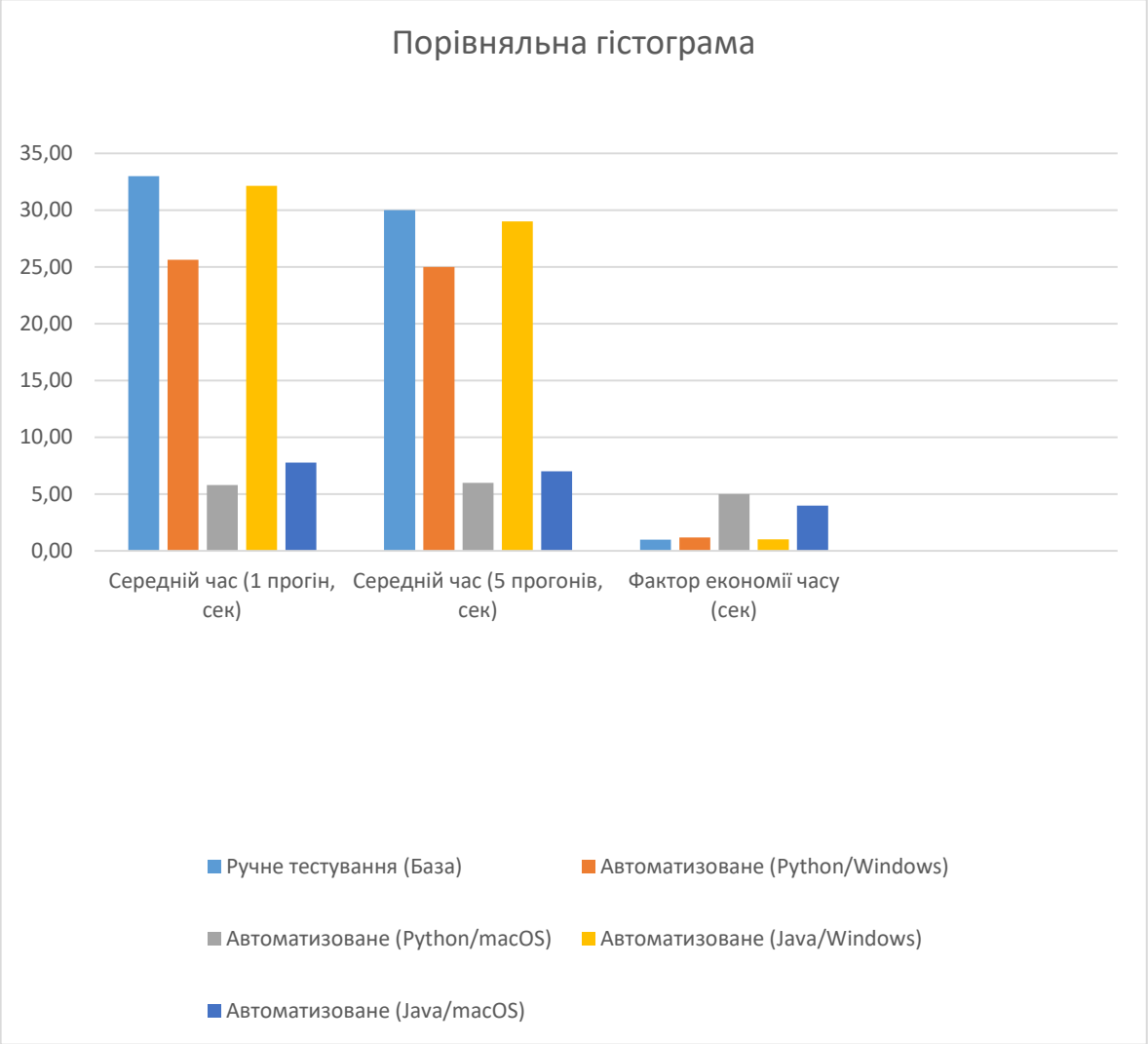

Рис 4.1. Порівняльна гістограма програмного тестування з традиційним ручним тестуванням

Кількісні та якісні переваги автоматизації

Окрім прямої економії часу, розроблений програмний комплекс надає ряд якісних та економічних переваг, що відображено у розширеній таблиці порівняння:

1. Детермінізм та Точність (Якісна перевага): точність виконання автоматизованого тесту становить 100%. Це усуває людський фактор (втома, неуважність, суб'єктивність інтерпретації результатів). Збій свідчить виключно про дефект у додатку;

2. Зручність використання та Підтримуваність (Порівняння стеків):

- BDD/Python: забезпечує високу читабельність завдяки лаконічному синтаксису Python, який вимагає менше допоміжного коду (boilerplate) порівняно з Java. Це безпосередньо призводить до нижчої вартості підтримки (Maintenance Cost);
- Java/ООП: хоча Java є потужною мовою для великих систем, її синтаксис та вимоги до об'єктно-орієнтованих шаблонів (ООП) часто призводять до більшого розміру коду для того самого сценарію, що ускладнює швидке внесення змін;

3. Економічна ефективність (ROI та Окупність): розробка комплексу розглядається як одноразова інвестиція (витрати на написання коду). Низькі експлуатаційні витрати означають, що можливо розрахувати точку окупності інвестицій (Break-Even Point), після якої кожне виконання тесту приносить чистий прибуток. У довгостроковій перспективі, автоматизація є значно дешевшою за утримання штату ручних тестувальників для виконання рутинних, повторюваних завдань;

4. Зниження Операційних Ризиків (Shift-Left Testing): можливість інтегрувати тести у CI/CD дозволяє виявляти дефекти значно раніше, ідеально — одразу після інтеграції коду. Згідно з галузевими дослідженнями, вартість виправлення дефекту, знайденого на етапі розробки, є в десятки разів нижчою, ніж вартість виправлення дефекту, знайденого у виробничому середовищі (production). Це є прямим показником зниження операційних ризиків та підвищення якості фінального продукту.

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи було успішно досліджено, розроблено та емпірично оцінено ефективність програмного комплексу для автоматизованого тестування веб-застосунків. Мета роботи, що полягала у створенні ефективного програмного засобу та науковому обґрунтуванні його переваг, була повністю досягнута.

На основі проведеного дослідження та практичної реалізації отримані такі ключові результати:

1. Теоретичне та методологічне обґрунтування: проведено ґрунтовний аналіз актуальності автоматизованого тестування як обов'язкової складової життєвого циклу розробки в умовах CI/CD. Обґрунтовано необхідність відмови від схильного до помилок ручного тестування. На основі порівняння сучасних інструментальних засобів (Selenium, Cypress, Playwright) було обрано стек технологій Python + Behave + Selenium WebDriver, як оптимальний для створення гнучкого, зрозумілого та універсального тестового фреймворку;

2. Архітектурне рішення: впроваджено методологію Behavior-Driven Development (BDD), що дозволило ефективно розділити логіку тесту від його опису. Використання фреймворку Behave дало змогу перетворити тестові сценарії на читабельну, доступну для бізнес-аналітиків документацію у форматі Gherkin. Це архітектурне рішення забезпечує високу підтримуваність комплексу, а його подальша інтеграція з патерном Page Object Model (POM) є необхідною умовою для мінімізації технічного боргу;

3. Програмна реалізація: створено функціональний програмний комплекс, який успішно виконує тестовий сценарій типу "Smoke Test" для верифікації критичного функціоналу веб-застосунку. Комплекс демонструє високу стійкість завдяки реалізації "захисного програмування" (обробка перешкод та рекламних банерів) та застосуванню надійних стратегій локалізації елементів (XPath), що є важливим фактором для зменшення

кількості хибних спрацьовувань. Створений комплекс є повністю крос-платформним і готовим до інтеграції у будь-яке CI/CD-середовище;

4. Емпірична валідація ефективності: проведене експериментальне порівняння на двох різних апаратних платформах (Windows та macOS) та з альтернативним стеком (Java/Selenium) чітко підтвердило кількісні переваги розробленого комплексу.

Ключові висновки за результатами експерименту:

- Економія часу та масштабованість: при багаторазовому виконанні (5 прогонів) автоматизоване тестування показало радикальне скорочення загального часу виконання порівняно з ручним тестуванням, що є прямим доказом значної економічної ефективності комплексу. Фактор економії часу чітко обґрунтовує окупність інвестицій в автоматизацію;
- Надійність і детермінізм: всі автоматизовані сценарії на Python та Java продемонстрували 100% точність виконання, повністю усуваючи суб'єктивні помилки та людський фактор, що є невід'ємною проблемою ручного тестування;
- Порівняння стеків: хоча швидкість виконання між Python та Java є порівнянною для коротких Smoke-тестів, розроблений комплекс на Python/BDD має перевагу у зручності читання та підтримуваності коду, що робить його кращим вибором для проектів, де пріоритетом є прозорість та низька вартість довгострокового обслуговування.

Таким чином, розроблений програмний комплекс є ефективним інструментом, що перетворює процес тестування з витратного та часозатратного на швидкий, детермінований та масштабований. Отримані результати можуть бути використані IT-компаніями для оптимізації процесів забезпечення якості та прискорення випуску якісних веб-продуктів на ринок.

ПЕРЕЛІК ПОСИЛАНЬ

1. Гуржій Кирило Анатолійович. Автоматизація тестування веб-застосунків кваліфікаційна бакалаврська робота : спеціальність 122 «Комп'ютерні науки» : освітня програма «Комп'ютерні науки та інформаційні технології в бізнесі» / К. А. Гуржій ; кер. роботи Н. І. Стяглик. – Харків : Харківський національний університет імені В. Н. Каразіна, 2024. – 59 с. https://drive.google.com/drive/folders/1UTR2DIRqP63sDuc1rW2PAhJQhLHU7SJR?usp=sharing

2. Adzic, G. (2009). Bridging the Communication Gap: Specification by Example and Behaviour-Driven Development. Addison-Wesley.

3. Barton, M. (2017). Selenium Design Patterns and Best Practices. Packt Publishing.

4. Chelimsky, J., Dennis, D., Hampi, R., Soni, B. (2010). The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends. Pragmatic Bookshelf.

5. Cohn, M. (2009). Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional.

6. Crispin, L., & Gregory, J. (2009). Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley.

7. IEEE Std 829-2008. (2008). IEEE Standard for Software and System Test Documentation.

8. Fowler, M. (2013). TestPyramid. MartinFowler.com. https://martinfowler.com/bliki/TestPyramid.html

9. Goos, G., Wirth, N., & Parnas, D. L. (1986). On the Criteria To Be Used in Decomposing Systems into Modules.

10. North, D. (2011). Introducing BDD. Dan North & Associates. http://dannorth.net/introducing-bdd/

11. Automation Panda. (2024). The Page Object Model. https://automationpanda.com/2018/06/17/page-object-model/

12. PyPi. (2024). Requests. Документація. https://pypi.org/project/requests/

13. Python Software Foundation. (2024). Exceptions. Документація. https://docs.python.org/3/tutorial/errors.html

14. Python Software Foundation. (2024). The Python Tutorial. https://docs.python.org/3/tutorial/

15. Selenium Python Bindings Documentation. (2024). Usage Notes for Python. https://selenium-python.readthedocs.io/

16. Techopedia. (2024). Behavior-Driven Development (BDD). https://www.techopedia.com/definition/31109/behavior-driven-development-bdd

17. TechTarget. (2024). XPath vs. CSS Selectors: Which is better for Selenium?. https://www.techtarget.com/searchsoftwarequality/tip/XPath-vs-CSS-Selectors-Which-is-better-for-Selenium

18. The Behave Development Team. (2024). Behave documentation: BDD for Python. https://behave.readthedocs.io/en/latest/

19. The Selenium Project. (2024). Selenium Documentation. https://www.selenium.dev/documentation/

20. W3C. (2018). WebDriver. https://www.w3.org/TR/webdriver/

ДОДАТКИ

Додаток А

```python
from behave import given, when, then
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
import time

@given("User opens '{homePage}' page")
def step_open_home(context, homePage):
    options = Options()
    options.add_argument("--start-maximized")
    context.driver = webdriver.Chrome()
    context.driver.maximize_window() #відкрити на весь екран
    context.driver.get(homePage) #гугл драйвер відкриває посилання яке я йому прописав

@given("User checks search field visibility")
def step_check_search_field(context):
    # Знаходжу поле і перевіряю його видимість
    context.search_field = context.driver.find_element(By.XPATH, '//input[@placeholder="Search"]')
    # Обробка перешкод (рекламних банерів)
    # Це є прикладом "захисного програмування" для стабільності автоматизації
    assert context.search_field.is_displayed(), "Search field is not visible"
    close_advertise_button = context.driver.find_element(By.XPATH, '//button[@class="CTR_loyalty-offer-close-button"]')
    close_advertise_button.click()

@when("User makes search by keyword '{keyword}'")
#Користувач вводить ключове слово у пошукову строку
def step_search_keyword(context, keyword):
    context.search_field.send_keys(keyword)

@when("User clicks search button")
#Користувач натискає кнопку пошуку
def step_click_search_button(context):
    search_button = context.driver.find_element( By.XPATH, '//button[@aria-label="Search"]')
    search_button.click()
```

```python

@when("User clicks wish list on first product")
#користувач натискає на додати у список бажаного на першому продукті
def step_click_first_wishlist(context):
    wishlist_button = context.driver.find_element( By.XPATH, '//button[@class="nl-button nl-button--primary   nl-product-card__wishlist--icon"]')
    wishlist_button.click()
    time.sleep(3)

@then("User checks that amount of products in wish list are '{expectedAmount}'")
#Користувач перевіряє чи додався товар до списку бажаного
def step_check_wishlist_amount(context, expectedAmount):
    wishlist_counter = context.driver.find_element( By.XPATH, '//span[@class="nl-wishlist-badge"]')
    count = wishlist_counter.text.strip()
    assert count == expectedAmount, f"Expected {expectedAmount}, but got {count}"
    context.driver.quit() #Закриття драйверу
```

smoke.feature

```gherkin
Feature: Smoke
  As a userI
  I want to test all main site functionality
  So that I can be sure that site works correctly

  @smoke
  Scenario Outline: Check add product to wishlist
    Given User opens '<homePage>' page
    And User checks search field visibility
    When User makes search by keyword '<keyword>'
    And User clicks search button
    And User clicks wish list on first product
    Then User checks that amount of products in wish list are '<expectedAmount>'

    Examples:
      | homePage                          | keyword | expectedAmount |
      | https://www.canadiantire.ca/en.html | cake    | 1              |
```

Додаток Б

pom.xml (Diplom)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Diplom</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>4.38.0</version>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>7.31.0</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>6.0.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.github.bonigarcia</groupId>
            <artifactId>webdrivermanager</artifactId>
            <version>6.3.3</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</project>
```

PageFactoryManager.java

```java
package manager;
import org.openqa.selenium.WebDriver;
import pages.*;

public class PageFactoryManager {  3 usages

    WebDriver driver;  3 usages

    public PageFactoryManager(WebDriver driver) {  1 usage
        this.driver = driver;
    }

    public HomePage getHomePage()  1 usage
    {
        return new HomePage(driver);
    }

    public SearchResultsPage getSearchResultsPage() { return new SearchResultsPage(driver); }
}
```

BasePage.java

```java
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;


public class BasePage {  2 usages  2 inheritors

    WebDriver driver;  5 usages

    public BasePage(WebDriver driver) {  2 usages
        this.driver = driver;
        PageFactory.initElements(driver,  page: this);
    }

    public void waitForPageLoadComplete(long timeToWait) {  3 usages
        new WebDriverWait(driver, Duration.ofMillis(timeToWait) ).until(
                WebDriver webDriver -> ((JavascriptExecutor) webDriver).executeScript( script: "return document.readyState").equals("complete"));
    }

    public void waitForAjaxToComplete(long timeToWait) {  1 usage
        new WebDriverWait(driver, Duration.ofMillis(timeToWait)).until(
                WebDriver webDriver -> ((JavascriptExecutor) webDriver).executeScript( script: "return window.jQuery != undefined && jQuery.active == 0;"));
    }

    public void waitVisibilityOfElement(long timeToWait, WebElement element) {  1 usage
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofMillis(timeToWait));
        wait.until(ExpectedConditions.visibilityOf(element));
    }
}
```

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

public class HomePage extends BasePage {  3 usages

    @FindBy(xpath = "//input[@id='search-input-0']")  3 usages
    private WebElement searchField;

    @FindBy(xpath = "//button[@aria-label='Search']")  1 usage
    private WebElement searchButton;

    @FindBy(xpath = "//span[@class='nl-wishlist-badge']")  2 usages
    public WebElement wishListProductsCount;

    public HomePage(WebDriver driver) { super(driver); }

    public void openHomePage(String url) { driver.get(url); }

    public boolean isSearchFieldVisible() { return searchField.isDisplayed(); }

    public void enterTextToSearchField(final String searchText) {  1 usage
        searchField.clear();
        searchField.sendKeys(searchText);
    }

    public WebElement getWishListProductsCount() { return wishListProductsCount; }

    public String getAmountOfProductsInWishList() {  1 usage
        String countString = wishListProductsCount.getText();
        return String.valueOf(Integer.valueOf(countString));
    }
    public void clickSearchButton() { searchButton.click(); }
}
```

SearchResultsPage.java

```java
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

import java.util.List;

public class SearchResultsPage extends BasePage { 3 usages

    @FindBy(xpath = "//button[@class='nl-button nl-button--primary  nl-product-card__wishlist--icon']") 1 usage
    private List<WebElement> wishListIcons;

    public SearchResultsPage(WebDriver driver) { super(driver); }

    public void clickWishListOnFirstProduct() { wishListIcons.get(0).click(); }
}
```

smoke.feature

```gherkin
Feature: Smoke
  As a user
  I want to test all main site functionality
  So that I can be sure that site works correctly

  Scenario Outline: Check add product to wishlist
    Given User opens '<homePage>' page
    And User checks search field visibility
    When User makes search by keyword '<keyword>'
    And User clicks search button
    And User clicks wish list on first product
    Then User checks that amount of products in wish list are '<expectedAmount>'
    Examples:
      | homePage                      | keyword | expectedAmount |
      | https://www.canadiantire.ca/en.html | cake    | 1              |
```

DefinitionSteps.java

```java
package stepdefinitions;
import io.cucumber.java.After;
import io.cucumber.java.Before;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import manager.PageFactoryManager;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import pages.*;

import static io.github.bonigarcia.wdm.WebDriverManager.chromedriver;
import static java.lang.Thread.sleep;

public class DefinitionSteps {

    private static final long DEFAULT_TIMEOUT = 60;  5 usages

    WebDriver driver;  4 usages
    HomePage homePage;  11 usages
    SearchResultsPage searchResultsPage;  3 usages
    PageFactoryManager pageFactoryManager;  3 usages

    @Before
    public void testsSetUp(){
        chromedriver().setup();
        driver= new ChromeDriver();
        driver.manage().window().maximize();
        pageFactoryManager = new PageFactoryManager(driver);
    }

    @And("User opens {string} page")
    public void openPage(final String url) {
```

DefinitionSteps.java

```
public class DefinitionSteps {

    @And("User opens {string} page")
    public void openPage(final String url) {
        homePage = pageFactoryManager.getHomePage();
        homePage.openHomePage(url);
    }

    @And("User checks search field visibility")
    public void checksSearchVisibility() {
        homePage.waitForPageLoadComplete(DEFAULT_TIMEOUT);
        homePage.isSearchFieldVisible();
    }

    @When("User makes search by keyword {string}")
    public void enterKeywordToSearchField(final String keyword) { homePage.enterTextToSearchField(keyword); }

    @And("User clicks search button")
    public void clickSearchButton() { homePage.clickSearchButton(); };

    @And("User clicks wish list on first product")
    public void clicksWishList() throws InterruptedException {
        searchResultsPage = pageFactoryManager.getSearchResultsPage();
        searchResultsPage.waitForPageLoadComplete(DEFAULT_TIMEOUT);
        searchResultsPage.clickWishListOnFirstProduct();
        sleep( millis: 2500);

    }

    @Then("User checks that amount of products in wish list are {string}")
    public void checksAmountOfProductsInWishList(final String expectedAmount) {
        homePage.waitForPageLoadComplete(DEFAULT_TIMEOUT);
        homePage.waitForAjaxToComplete(DEFAULT_TIMEOUT);
        homePage.waitVisibilityOfElement(DEFAULT_TIMEOUT, homePage.getWishListProductsCount());
        Assert.assertEquals(homePage.getAmountOfProductsInWishList(), expectedAmount);
```

```
69          Assert.assertEquals(homePage.getAmountOfProductsInWishList(), expectedAmount);
70      }
71      @After
72      public void tearDown() { driver.close(); }
75  }
```