МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2023 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

ФРЕЙМВОРК ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ WEB-САЙТУ

Спеціальність 122 «Комп'ютерні науки»

122 – БКР – 402.22020202

Виконала студентка 4-го курсу, групи 402
_____ М. Д. Д'яконова
« » червня 2023 р.

Керівник: канд. техн. наук, доцент
_____І. О. Калініна
« » червня 2023 р.

Миколаїв – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____2022 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студентці групи 402 факультету комп'ютерних наук Д'яконовій Марії Дмитрівні.

1. Тема кваліфікаційної роботи «Фреймворк для автоматизації тестування web-сайту».

Керівник роботи Калініна Ірина Олександрівна, канд. техн. наук, доцент.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «___» _____ 20___ р. № ____

2. Строк представлення кваліфікаційної роботи студентом «___» _____ 20__ р.

3. Вхідні (початкові) дані до роботи: наявні бібліотеки для автоматизації тестування web-продуктів; найбільш вживані патерни проектування, що застосовуються в автоматизації тестування; теорія тестування.

Очікуваний результат: власний фреймворк для автоматизації web-сайту, декілька автоматизованих сценаріїв перевірки функціоналу web-сайту.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

– аналіз існуючих підходів до автоматизації тестування web-продуктів;

– проектування архітектури власного фреймворку з використанням сучасних паттернів проектування;

– безпосередня розробка власного фреймворку для автоматизації тестування web-сайту та його практичне застосування.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: «Проведення оцінки умов праці в офісному приміщенні».

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Боженко А. Л.	

Керівник роботи __канд. техн. наук, доцент Калініна І. О._____

(наук. ступінь, вчене звання, прізвище та ініціали)

_____

(підпис)

Завдання прийнято до виконання ___Д'яконова М. Д._____

(прізвище та ініціали)

_____

(підпис)

Дата видачі завдання «__» _____ 20___ р.

КАЛЕНДАРНИЙ ПЛАН
виконання бакалаврської кваліфікаційної роботи

Тема: Фреймворк для автоматизації тестування web-сайту

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників БКР	26.10.2022	30.10.2022	
2	Отримання завдання на виконання БКР	20.11.2022	24.11.2022	
3	Складання календарного плану роботи на весь період виконання БКР	07.12.2022	09.12.2022	
4	Отримання завдання на переддипломну практику	26.04.2023	01.05.2023	
5	Проходження переддипломної практики, збір та аналіз матеріалів до БКР	01.05.2023	14.05.2022	
6	Розробка звіту з переддипломної практики	15.05.2023	17.05.2023	
7	Виконання БКР: аналіз сучасних підходів до автоматизації тестування, дослідження паттернів програмування, що застосовуються в автоматизації, розробка власного фреймворку	15.05.2023	19.06.2023	
8	Попередній захист БКР на засіданні комісії кафедри	29.05.2023	30.05.2023	
9	Доробка та остаточне оформлення БКР	02.06.2023	19.06.2023	
10	Подання БКР рецензенту	15.06.2023	17.06.2023	
11	Подання БКР, її електронної копії та інших документів (відгуку, рецензії) до захисту	19.06.2023	22.06.2023	
12	Захист БКР перед екзаменаційною комісією (ЕК)	26.06.2023	29.06.2023	

Розробив студент __Д’яконова М. Д._____ _____

(прізвище, ім’я, по батькові студента) (*підпис*)

Керівник роботи __канд. техн. наук, доцент Калініна І. О. _____

(*посада, прізвище, ім’я, по батькові)* *(підпис)*

«_09_» __грудня___ 2022 р.

АНОТАЦІЯ

бакалаврської кваліфікаційної роботи

студентки групи 402 ЧНУ ім. Петра Могили

Д'яконової Марії Дмитрівни

Тема: «Фреймворк для автоматизації тестування web-сайту»

Актуальність роботи складає зростаюча необхідність бізнесів у проведенні якісного тестування свого web-продукту в найбільш оптимальний часо- та ресурсозатратний спосіб.

Об'єкт роботи – процес розробки фреймворку для автоматизації тестування інтерфейсу та API сервісу web-сайту.

Предмет роботи – інструментальні засоби та методологічні підходи до розробки фреймворку, що призначено для автоматизації тестування web-сайту.

Метою бакалаврської кваліфікаційної роботи є автоматизація тестування web-сайту шляхом розробки власного фреймворку на основі сучасних патернів програмування, що дозволить пришвидшити процес тестування, забезпечуючи ефективне виявлення помилок та перевірку правильності його роботи.

Пояснювальна записка складається зі вступу, чотирьох розділів, спеціального розділу з охорони праці, висновків та додатків.

У першому розділі розглядаються підходи до автоматизації тестування web-продуктів, їх переваги та недоліки.

У другому розділі досліджено основні принципи проектування власного фреймворку, а також послідовність написання автоматизованих тестів.

У третьому розділі описано створення та налаштування власного унікального фреймворку, використовуючи мову програмування Java та такі існуючі інструментів автоматизації, як Selenium WebDriver та Retrofit.

Апробацію розробленого фреймворку було проведено в четвертому розділі на основі автоматизованих кейсів для перевірки функціональності сервісу Trello.

Таким чином, в результаті практичної частини розроблено фреймворк для автоматизації тестування web-сайту, а також декілька наочних автоматизованих тестів.

Бакалаврська кваліфікаційна робота містить 68 сторінок, 21 рисунок, 1 таблиця, 32 використаних джерела та 4 додатки.

Ключові слова: автоматизація тестування, тестовий фреймворк, автоматизовані тести.

ABSTRACT

of a bachelor's degree work of a student of group 402 at Petro Mohyla Black Sea National University

Diakonova Mariia

Topic: "Framework for testing automation of a website"

The relevance of the work is the growing need for businesses to conduct high-quality testing of their web product in the most optimal time- and resource-consuming way.

The object of the work is process of development of a framework for website testing automation.

The subject of the work is the instruments and methodological approaches to a framework development, which is designed for automating the testing process of a website.

The goal of the bachelor's degree work is to automate testing by developing a framework based on modern programming patterns, which will allow to speed up the process of testing the website interface, providing effective error detection and checking the correctness of its operation.

The explanatory note consists of an introduction, four sections, a special section on labor protection, conclusions and appendices.

In the first section, there are analyzed approaches to the automation of UI testing of web products, alongside their advantages and disadvantages.

The second chapter explores the basic principles of building a unique framework, as well as the process of automated tests writing.

The third chapter describes the creation and configuration of a unique framework using the Java programming language and the existing tools for automation, such as Selenium WebDriver and Retrofit.

Approbation of the developed framework took place in the fourth chapter through automated tests that cover Trello service functionality check.

As a result, a framework for automating website testing was developed, as well as several autotests for visual representation.

Bachelor thesis consists of 68 pages, 21 pictures, 1 table, 32 used sources and 4 appendices.

Keywords: testing automation, testing framework, automated tests.

ЗМІСТ

ВСТУП

У сучасному світі, де web-сайти є важливою складовою будь-якого бізнесу чи організації, якість їхнього функціонування та інтерфейсу стає ледь не вирішальним фактором успіху. Тестування web-сайтів, в свою чергу, є необхідним етапом життєвого циклу програмного забезпечення (SDLC), але якщо виконувати його вручну, воно може бути часо- та ресурсозатратним завданням. З метою полегшення та прискорення цього процесу, а також підвищення якості web-продуктів, в багатьох компаніях впроваджується автоматизоване тестування на основі розробки, підтримки та покращення власного фреймворку. Під фреймворком в даному випадку розуміють певний каркас, набір інструкцій, який підходить під конкретну бізнес-логіку і виконує заздалегідь визначені та чітко прописані завдання.

Впровадження автоматизації тестування вирішує декілька питань відразу: затрати часу (ручне тестування усіх робочих процесів, полів та негативних сценаріїв вимагає багато часу) [1], затрати ресурсів (оскільки прогон автотесту не вимагає втручання людини, зменшується необхідність виконання людиною рутинних процесів і таким чином ресурс тестувальника звільнюється на пошук незвичних та неочевидних тестових комбінацій), фінансові затрати (за рахунок збільшення тестового покриття мінімізується ризик виникнення неочікуваних дефектів на стороні кінцевого користувача, що економить гроші на розробку продукту), а автоматизація тестування на основі створення власного фреймворку дозволяє створити єдиний уніфікований інтерфейс, який буде зрозумілим та звичним для користування всіма зацікавленими учасниками команди розробки. Дані фактори складають *актуальність* даної роботи.

Метою даної роботи є автоматизація тестування web-продукту шляхом розробки власного фреймворку, що дозволить пришвидшити процес тестування інтерфейсу web-сайту, забезпечуючи ефективне виявлення помилок та перевірку правильності його роботи. Фреймворк буде базуватися на сучасних методологіях тестування та надавати широкий набір інструментів для зручного та

повторюваного виконання тестових сценаріїв.

Основні завдання даної роботи включатимуть в себе:

- аналіз існуючих рішень для автоматизації тестування;
- проектування архітектури фреймворку;
- розробку необхідних модулів та компонентів;
- безпосереднє проведення тестування та оцінку ефективності розробленого фреймворку.

Об'єктом дослідження даної роботи є процес розробки фреймворку для автоматизації тестування інтерфейсу та API сервісу web-сайту, а *предметом дослідження* – інструментальні засоби та методологічні підходи до розробки фреймворку, що призначено для автоматизації тестування web-сайту.

Для розв'язання вищезазначених задач будуть використовуватися різноманітні *методи* тестування (позитивні, функціональні, смоук тестові сценарії, написані за принципом чорного ящика на основі попередньо створених тест кейсів), *інструментальні засоби* (Selenium WebDriver та Retrofit в якості основи для фреймворку та IntelliJ IDEA в якості середовища програмування), а також інформаційні технології (власний фреймворк та приклади автотестів будуть написані мовою програмування Java).

Очікується, що розроблений фреймворк спростить процес тестування обраного web-сайту, зменшить затрати часу та ресурсів, а також забезпечить високу якість та надійність розробленого існуючого web-продукту.

1 ДОСЛІДЖЕННЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ З ВИКОРИСТАННЯМ ВЛАСНОГО ФРЕЙМВОРКУ. ПОСТАНОВКА ЗАДАЧІ

Тестування – це процес взаємодії з продуктом або його компонентом у певних умовах, ведення звітності такої взаємодії та складання оцінки продукту або його компонента на цій основі. Простими словами, це перевірка програми на відповідність вимогам (як бізнес, так і клієнтським очікуванням) та визначення якості продукту на основі цієї перевірки; звідси можна визначити, що можна вдосконалити для підвищення цієї якості [2].

Тестування можна розділити на ручне та автоматизоване. При ручному тестуванні сценарії перевірки виконуються безпосередньо вручну, без використання яких-небудь засобів автоматизації, автоматизоване ж тестування припускає використання спеціального ПО для контролю виконання тестів та порівняння очікуваного і фактичного результату роботу програми [3].

В даній роботі будемо аналізувати автоматизоване тестування, яке, в свою чергу, має наступні види:

- автоматизація тестування коду (code-driven testing) – покриття функціоналу unit-тестами, яке зазвичай здійснюється розробником, по суті це просте тестування на рівні програмних модулів, класів та бібліотек;
- автоматизація тестування API (Application Programming Interface) – спеціальна програма (фреймворк автоматизації тестування), що допомагає тестувати програмний інтерфейс програми, покликаний взаємодіяти з іншими програмами або з користувачем;
- автоматизація тестування графічного інтерфейсу користувача (graphical user interface testing) – спеціальна програма (фреймворк автоматизації тестування), що дозволяє отримувати результати тестування, які будуть максимально приближеними до дій звичайного користувача [3].

Дані види автоматизації співпадають з класичною пірамідою тестування, наведеною нижче.

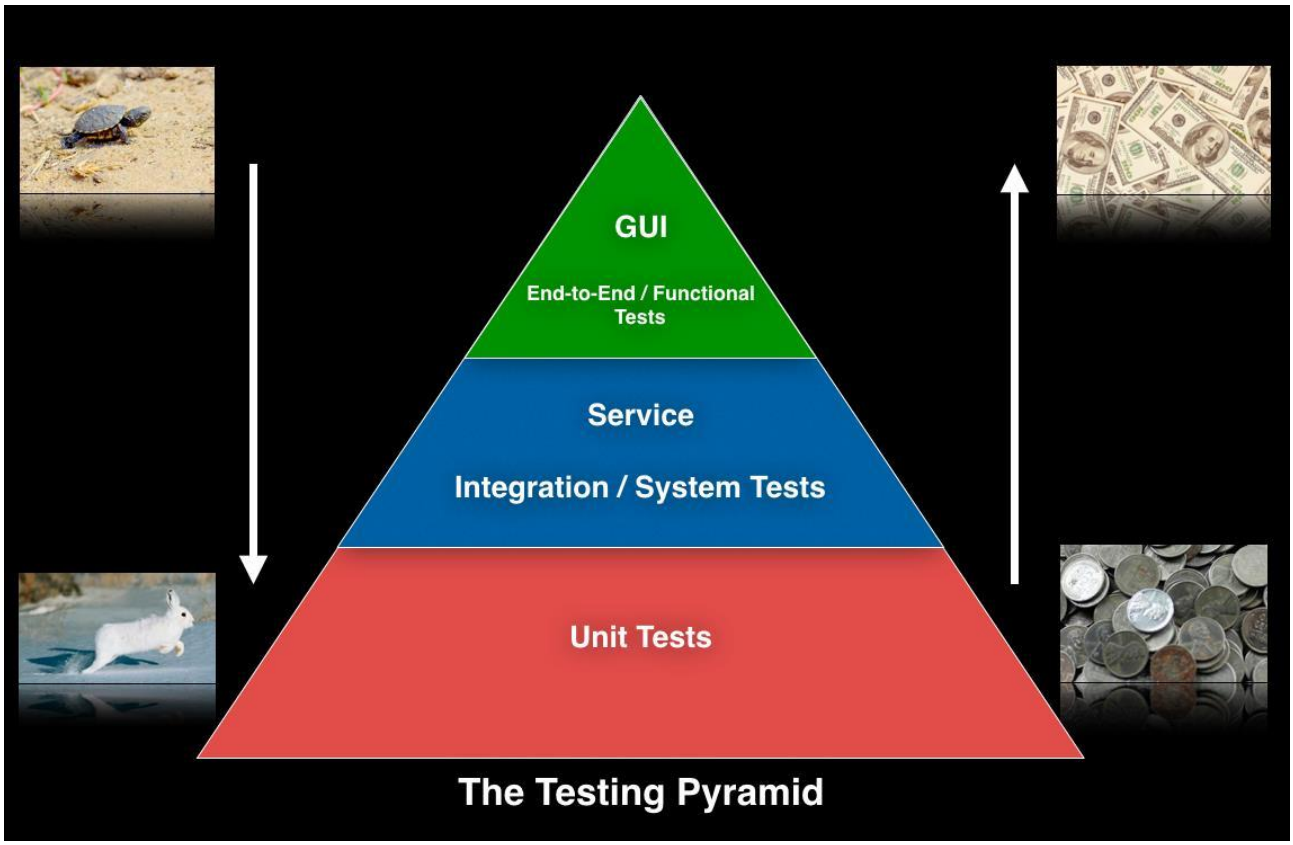

Рисунок 1 – Стандартна піраміда тестування

Представлена модель покликана продемонструвати, що різні рівні автоматизації тестування переслідують різні цілі. Так, ступені піраміди представляють групи тестів. Чим вище рівень, тим нижча деталізація тесту, його ізольованість (вплив одного окремого блоку функціональності на інші) та час виконання. Тести на нижньому щаблі невеликі, ізольовані, швидкі та перевіряють невелику частину функціональності, тому зазвичай їх необхідно написати багато задля досягнення прийнятного рівня покриття. Верхній щабель, навпаки, представляє собою комплексні високорівневі тести, які, як правило, повільніші, ніж тести нижчих рівнів, і зазвичай перевіряють велику частину функціональності, тому їх потрібно кількісно менше для досягнення прийнятного рівня покриття [4].

Таким чином, відповідно до класичного підходу, чим тест нижче в піраміді, тим швидше його можна виконати і отримати результат, також якщо тест упав на більш високому рівні, він уже точно дасть побічний ефект. Так, наприклад, в зворотну сторону ця теорія не працює.

1.1 Автоматизація тестування графічного інтерфейсу, її переваги, обмеження та підходи

Графічний інтерфейс (GUI або UI) – це перше, що бачить перед собою та з чим взаємодіє кінцевий користувач під час використання будь-якої програми. Саме тому команді розробки важливо переконатися, що всі елементи продукту відображаються і працюють як слід. Це включає перегляд таких графічних ресурсів і елементів керування програми, як: значки, кнопки, модальні вікна, панелі інструментів тощо.

Вручну це робити ресурсо- та часозатратно, адже в такому випадку людина має не тільки перевірити абсолютно всі компоненти продукту окремо, а також їх взаємодію один з одним. Більше того, необхідно враховувати також різноманітні тестові сценарії, обставини та технічну конфігурацію, до якої входять операційні системи, браузери, типи пристроїв, розміри їх екранів, тощо [5].

Вичерпний список перевірок, які необхідно здійснити в рамках тестування графічного інтерфейсу і на що треба звертати увагу при кожній з них, виглядає наступним чином:

- загальний вигляд сторінки: перевірка цілісності зовнішнього відображення на відповідність дизайну та макетам, загальної гармонії, реакції продукту на масштабування, тощо;
- текст: перевірка тексту на наявність помилок та похибок, правильне форматування та однакове вирівнювання в усіх місцях продукту;
- вибір елементів: перевірка інтуїтивно зрозумілого виділення елементів, з якими кінцевий користувач може взаємодіяти на різних етапах цієї взаємодії (наприклад, підсвічення тексту з посиланням яскравим кольором на стартовій сторінці, виведення маленького віконця з додатковою інформацією при наведенні на посилання і відкриття повноцінного ресурсу після кліку на посилання);
- робота з полями: перевірка того, що при загальній уніфікації полів, статичні поля, які не можна редагувати, чітко відрізняються від полів, які

потребують від кінцевого користувача введення інформації;

- ворми: перевірка зовнішнього вигляду та розташування форми, а також її наповнення: радіокнопок, чекбоксів, випадаючих списків, кнопок, текстових полей тощо;
- інші вимоги: специфічні вимоги, пов’язані з замовником чи середовищем накшталт маркетплейсу, в якого можуть бути свої специфікації, яких необхідно дотримуватись, інакше продукт може не пройти перевірку та не бути розміщеним на бажаних ресурсах [6].

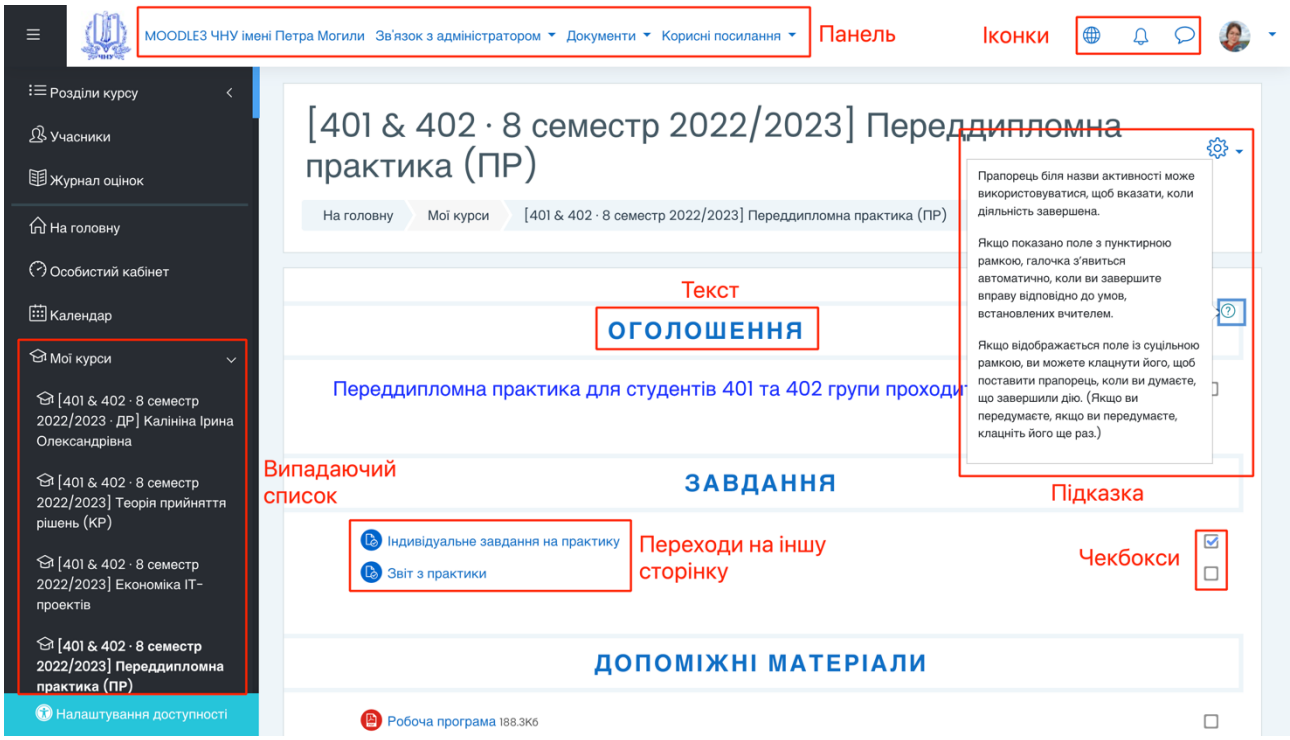

Рисунок 1.1 – Список перевірок графічного інтерфейсу на прикладі системи Moodle

1.1.1 Переваги та недоліки автоматизації тестування користувацького інтерфейсу

Вичерпне тестування графічного користувацького інтерфейсу має вагомі переваги, до яких відносять:

- покращення функціональності продукту: чим раніше буде знайдено дефект, тим швидше його можна виправити з меншою витратою фінансів,

що підвищує конверсію користувачів та їх задоволення продуктом;

- покращення користувацького досвіду (user experience): залучення інструментів автоматизації до процесу тестування програми є зручним та корисним способом її оптимізації, адже навіть при найоптимальнішому коді програми складний або незручний у використанні інтерфейс здатний занизити рівень адаптації кінцевих користувачів і, відповідно, відвернути їх від продукту;
- підвищення рейтинговості та конкурентоспроможності продукту: якщо тестування інтерфейсу проведено належним чином, це робить web-продукт достойним обличчям бренду, що підвищує загальну репутацію бізнесу [7].

Якісна автоматизація процесу тестування дозволяє не лише втілити всі наведені вище переваги перевірки користувацького інтерфейсу, а й в значній мірі пришвидшити його, адже з впровадженням практик безперервної інтеграції та безперервної доставки (Continuous integration / Continuous delivery, CI/CD) участь людини не знадобиться не лише в процесі безпосереднього виконання тесту, а й навіть для його запуску, адже він починатиметься автоматично як тільки певна частина коду буде зачеплена, і по завершенні, бо вивантаження результатів прогону тестів можна також налаштувати автоматично.

Крім того, з залученням автоматизації, можна уникнути помилок, пов’язаних з людським фактором по типу неуважності чи втоми, або нестачі робочого часу, адже середовище виконання тесту не залежить від таких обставин.

Разом з тим, необхідно також розуміти, що автоматизація рутинних процесів тестування користувацького інтерфейсу може мати і певні обмеження, як, наприклад, відсутність людської взаємодії, критичного мислення та креативного підходу у випадках, коли треба перевірити нестандартну поведінку програми, або неефективна витрата часу на налаштування емуляцій для точкової перевірки невеличкої правки в коді. Також не слід забувати, що саме при ручному тестуванні створюється найбільш приближений до реального контекст для пошуку проблем, на які зазвичай потрапляють кінцеві користувачі, адже реальну поведінку людини

не так легко симулювати і тому деякі дефекти можуть бути пропущеними чи непоміченими під час автоматизованого тестування графічного користувацького інтерфейсу.

1.1.2 Шляхи уникнення обмежень, які накладаються на автоматизацію тестування користувацького інтерфейсу

Логічним буде припустити, що для превалювання переваг автоматизованого тестування користувацького інтерфейсу над його обмеженнями достатньо вибудувати правильну стратегію покриття функціоналу автоматизованими тестами та мати реалістичні очікування в їх застосуванні. Так, зазвичай, автоматизації піддаються прості позитивні перевірки, тобто поведінка, яку ми очікуємо отримати в безумовній більшості випадків, тоді як негативні сценарії та нетипова (унікальна) поведінка найкраще перевіряється вручну.

Найбільш розповсюджені підводні камені UI-тестування та шляхи їх вирішення наведені нижче.

1) «тестування займає багато часу»:
   - запуск тестів на паралельне виконання або з використанням більше ніж одного віртуального середовища;
   - групування тестів на основі передумов та почерговий запуск тестів без необхідності відтворювати середовище з нуля;
   - виконання базового сценарію в першу чергу, виключаючи необхідність тестування комплексного функціоналу у випадку, якщо зламалося щось основоположне;
   - нічний запуск тестів;
   - першочергова перевірка кількох E2E-послідовностей (end-to-end або наскрізне тестування, що перевіряє продукт від початку до кінця);
   - підтримання атомарності окремих тестів задля забезпечення можливості їх окремого запуску без необхідності запускати цілий тестовий набір;
   - перенесення деяких передумов або перевірка кінцевих результатів з UI

на бекенд. Наприклад, якщо для передумови тесту потрібне створення нового користувача і це не є основною перевіркою сценарію, його можна створити на бекенді;

2) нестабільні/динамічні елементи:
   - встановлення заборони або обмеження на оновлення середовища під час роботи тесту;
   - замість неявного очікування, яке прописується в одиницях часу, використовувати явне очікування на взаємодію з необхідним елементом;
   - застосування повторних спроб підтвердження наявності необхідного елементу, в помірній кількості;
   - логування результатів запуску у вигляді наочного матеріалу: скріншотів, гіф, коротких відео;
3) «тести не помічають зміни в UI»:
   - застосування наявних або створення власних додаткових інструментів для порівняння скріншотів з головних сторінок, вкладок, пунктів, тощо. Це підвищить вірогідність того, що зміни, які не було виявлено тестами, будуть «відловлені» за допомогою порівнянь [5].

1.1.3 Підходи до тестування UI у web-сервісах: record-and-playback та тестування на основі моделей

При виборі підходів до автоматизації тестування графічного користувацького інтерфейсу, першими на думку спадають два найбільш поширені: «запис-та-відтворення» (record-and-playback) і тестування на основі моделей (model-based testing). Розглянемо кожен окремо.

Основа методу запису-та-відтворення лежить в його назві – це запис усіх взаємодій користувача з web-застосунком та їх відтворення в тій самій послідовності за допомогою спеціальних скриптів. Виконання таких скриптів повторюється знову і знову без втручання тестувальника або іншого зацікавленого

члена команди. Такий спосіб підходить, зокрема, для простих інтерфейсів з невеликою кількістю функцій. З плюсів підходу record-and-playback відразу можна відмітити легкість у використанні та відсутність необхідності у тестувальника знань та вмінь писати код.

Звичайно, є і свої мінуси. Даний підхід вимагає постійного контролю: скрипти можуть не відтворюватися коректно у разі незначних оновлень, переміщення UI-елементів, використання нових ID або класів. Також, іноді простіше та швидше перевірити помилку вручну на вже наявному наборі даних, аніж перезаписувати весь скрипт з самого початку чи якоїсь конкретної відправної точки. Але, безперечно, все залежить від інструментів, які використовує команда розробки, та функціональності, яка перевіряється.

Розширене порівняння переваг та недоліків підходу наведене у таблиці 1.2.

Таблиця 1.2 – Переваги та недоліки record-and-playback підходу автоматизації тестування

Переваги	Недоліки
Відсутність необхідності знань та вмінь писати код у тестувальника.	Зміни в інтерфейсі продукту вимагають перезапису тестових сценаріїв з нуля.
Будь-який зацікавлений член команди розробки зможе створити свої тести після засвоєння роботи з обраним інструментом.	Труднощі підтримки автотестів при великій їх кількості, що забирає час та ресурс у тестувальника.
Підхід забезпечує легкий перехід з ручного тестування в автоматизоване.	Відсутність необхідності вміння кодувати для створення таких автотестів забирає варіативність.

Аналізуючи представлену таблицю можна дійти висновку, що підхід захоплення-та-відтворення ідеально підходить для невеликих проектів або

стартапів з незначною кількістю тестів, тоді як для великих або стрімко зростаючих проектів з високим покриттям функціональності він може виявитись занадто «неповоротким» [8].

Прикладами інструментів, що пропонують зазначений підхід на сучасному ринку, є:

- Katalon;
- Selenium IDE;
- TestComplete;
- Testim;
- Ranorex Studio;
- Telerik UI.

Тестування на основі моделей, в свою чергу, більше підходить для сценаріїв, коли інженери створюють абстрактну модель з описом функціональності та будують процес тестування на її основі.

Проте такий спосіб вимагає певних знань і досвіду в автоматизації та програмуванні. Написання оптимального під потреби проекту та команди фреймворку для тестування займає певний час і для тестувальників, які слабо розуміються на технічних питаннях, «прочитати» та зрозуміти всі взаємозв’язки та залежності може бути важко. Однак цю та інші складнощі більшою мірою досить просто обійти завдяки рішенням, що пропонують сучасні інструменти, тому вважатимемо, що даний метод має більше переваг, ніж вищезазначений record-and-playback.

Тестування на основі моделей заощаджує час та гроші, адже його легко підлаштувати під різні умови: середовища, машини, браузери та інше. Якщо знадобиться покрити нові функції, зазвичай це не спричинить таких додаткових витрат, щ обули описані вище, як у тестуванні методом запису-та-відтворення. І останнє, але не менш важливе: завдяки методу на основі моделей легко виявляти та виправляти помилки.

Разом з тим необхідно пам’ятати про те, що тестування на основі моделей має

бути виконано в чітко визначених рамках:

1) моделювання – етап збору та опису вимог, які будуть використовуватись для створення тестів. На цьому ж етапі передбачається окреслення очікуваного результату. Даний етап відповідає на питання «Що буде тестуватися?»;
2) планування тестування – етап визначення критеріїв тестування, відповідає на питання «Як буде тестуватися функціонал?»;
3) тест-дизайн – етап вибору підходу до тестування на основі попередньо зіставлених критеріїв, відповідає на питання «Яким чином буде тестуватися функціонал?»;
4) створення тестової документації (тест-кейсів) – етап безпосереднього написання вичерпних тестових сценаріїв;
5) виконання тестування – етап безпосереднього виконання тестування і порівняння очікуваного результату з фактичним [9].

1.2 Автоматизація тестування API-сервісу та її особливості у порівнянні з ручним тестуванням

API (Application Programming Interface) розшифровується як інтерфейс прикладного програмування і являє собою набір визначень, протоколів та правил, які рекомендовано до використання розробниками при створенні прикладного програмного забезпечення та його інтеграції в існуючі системи і платформи [10]. За рахунок цього, API спрощує взаємодію (інтеграцію) двох або більше систем разом шляхом реалізації їх роботи у вигляді обміну ланцюжків запитів-відповідей, кожен з яких побудовано за певними визначеними принципами.

Тестування API, в свою чергу, являється рівнем тестування програмного забезпечення, який гарантує, що інтерфейс працює так, як очікується, у відповідності до загальних норм та надійно виконуючи свої функції без негативного впливу на продуктивність програми в цілому. Таке тестування полягає у перевірці HTTP-запитів атомарно (окремо), або у зв’язці один з одним.

Варто розуміти, що в залежності від масштабу проекту, кількість таких

запитів може сягати сотень або тисяч, тому процес тестування обмежується ручним зазвичай лише на початкових етапах, коли в кодову базу вноситься велика кількість незначних частих змін, тим самим забезпечуючи більшу гнучкість. В усіх інших випадках ручне тестування API є складовою частиною процесу забезпечення якості і застосовується наряду з автоматизованим, коли необхідно ретельно перевірити новий функціонал та зіставити основу для подальшої автоматизації, або верифікувати точкові зміни та виправлення у старому функціоналі без звернення до подекуди складної та довгої автоматизованої системи.

До переваг автоматизації тестування API можна віднести ті ж самі аргументи, що і до користувацького інтерфейсу, – це більш високий рівень точності, адже автоматизована система переглядає код методично, тестуючи кожну функцію по черзі і щоразу однаково, а також збереження часового та енергетичного ресурсу. Повертаючись до піраміди тестування, наведеній на рисунку 1, повна або майже повна автоматизація тестування даного рівня функціоналу можлива завдяки тому, що кожна окрема складова частина API-сервісу не має великого впливу на інші та не дає зворотніх ефектів у разі виникнення дефекту, завдяки чому тести на такий функціонал будуть швидкими, простими та ізольованими.

Разом з тим, інколи стоїть задача відслідкувати користувацькі дані чи поведінку в реальному часі для складання певних метрик: це може бути відстеження способу взаємодії клієнтів з web-сайтом або перетворення їх особистої інформації у змінні параметри взаємодії з продуктом, тощо. Такі дослідження в результаті дійсно можуть розширити функціональність web-сайту, але водночас стати причиною унікальних проблем в процесі тестування API. Якщо будь-які реальні дані призводять до серйозних і неочікуваних відхилень продуктивності, це може спричинити проблеми з бекендом або ввести в оману решту учасників процесу розробки [10]. І це складає недоліки, з якими може зіткнутися команда при автоматизації тестування API-частини.

Такі недоліки впровадження автоматизації тестування API можуть обмежити ефективність набору тестів, але це скоріше проблеми, про які варто знати, ніж ті,

що повністю руйнують корисність системи.

До поширених інструментів автоматизації тестування API відносять:

- Postman;
- SoapUI;
- JMeter;
- Katalon;
- Rest Assured.

1.3 Постановка задачі

Постановка задачі включатиме в себе наступні пункти:

1) проведення аналізу існуючих методів автоматизації тестування web-сайтів, їх переваг та недоліків;
2) перегляд існуючих альтернатив серед інструментів автоматизації тестування інтерфейсу та вибір найбільш підходящих;
3) визначення вимог до фреймворку для автоматизації тестування web-сайту;
4) розробка такої архітектури фреймворку, що забезпечить зручний та ефективний спосіб написання, виконання і управління тестами;
5) розробка основних функцій фреймворку, таких як взаємодія з браузером, маніпуляції з елементами сторінки, перевірка стану елементів, робота з тестовими даними, тощо;
6) тестування і валідація фреймворку на реальних web-сайтах для перевірки його функціональності та ефективності.

Вимоги до програмної реалізації, що буде зроблено:

- фреймворк має бути простим, зрозумілим та лаконічним;
- у фреймворку мають використовуватися сучасні підходи до автоматизації;
- вхідні дані є статичними, але можуть змінюватися тестувальником або будь-якою зацікавленою особою в самому тесті.

Висновки до розділу 1

У даному розділі було проведено аналіз наявних методів тестування інтерфейсу та API-сервісу web-сайтів і розглянуто їх переваги, недоліки та обмеження, а також запропоновано шляхи вирішення та оптимальні підходи. Виявлено, що в ряді випадків ручне тестування є часо- та ресурсозатратним процесом, який може бути покращений шляхом впровадження автоматизації.

Під час аналізу було виявлено, що наявні фреймворки для автоматизації тестування інтерфейсу та API-сервісу web-сайтів мають свої обмеження та недоліки. Деякі з них не забезпечують достатньої гнучкості та простоти використання, інші не підтримують широкий спектр браузерів та технологій.

На основі проведеного аналізу було сформульовано вимоги до фреймворку для автоматизації тестування web-сайту. Вимоги включають:

- зручний та інтуїтивно зрозумілий інтерфейс для написання тестових скриптів;
- підтримка різних браузерів та платформ;
- можливість легко маніпулювати елементами сторінки, виконувати перевірку стану елементів;
- забезпечувати підготовку тестових даних, а також здійснювати перевірку виконаних дій через звернення до бекенду;
- представляти дані про результати тестування.

Отже, на основі отриманих результатів можна зробити висновок, що розробка власного фреймворку для автоматизації комплексного тестування web-сайту є доцільною задачею, оскільки існуючі на ринку рішення не повністю задоволняють вимогам будь-якого проекту, натомість власний проект завжди потребуватиме внесення певних коригувань з урахуванням його вимог, особливостей та цілей.

2 ІНСТРУМЕНТИ, ПІДХОДИ ТА ПАТЕРНИ ПРОГРАМУВАННЯ, ЩО ВИКОРИСТОВУЮТЬСЯ В АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ. ОСНОВНІ ПРИНЦИПИ НАПИСАННЯ АВТОМАТИЗОВАНИХ ТЕСТІВ

Для того, щоб приступити до розробки архітектури власного фреймворку для автоматизації, вважаємо доцільним ознайомитись з інструментами, підходами та патернами проектування, що застосовуються в автоматизації тестування, адже їх немало і кожен має свої переваги та недоліки, і для того, щоб зупинитися на конкретних, необхідно розуміти як нагальні потреби, так і майбутні перспективи бізнесу і web-продукту, для якого розроблятиметься фреймворк, технічні можливості команди, тощо.

Також, оскільки фреймворк не існує у вакуумі, а покликаний бути основою для автоматизованих тестів, в даному розділі розглядатимуться основні принципи написання автотестів, а саме: на базі чого вони створюються, які бувають, та інше.

2.1 Selenium WebDriver як інструмент автоматизації тестування користувацького інтерфейсу, його особливості

Selenium WebDriver – бібліотека для керування браузерами, яка безпосередньо надсилає команди браузерам, використовуючи рідне для кожного окремого браузера API, та отримує результати тестування, які будуть максимально близькі до дій звичайного користувача [11].

Для початку роботи з інструментом необхідно:

1. браузер – реальний браузер (Chrome, FireFox, Opera, Safari) певної версії, встановлений на певній ОС, роботу якого користувач бажає автоматизувати;
2. Driver – найважливіша сутність, що запускає браузер, взаємодіє з ним відправляючи команди і закриває, по своїй суті є web-сервером. У зв’язку з тим, що в кожного браузера свої відмінні команди управління, які по-своєму реалізовані, у кожного браузера є свій драйвер;
3. скрипт – сам тест, що містить набір команд певною мовою програмування. Такі скрипти використовують готові Selenium WebDriver бібліотеки;

4. WebElement – сутність, яка представляє собою абстракцію над web-елементами (текст, поля для вводу, кнопки, іконки, тощо);
5. By – абстракція над локатором web-елемента; клас, що інкапсулює інформацію про селектор, а також сам локатор елемента, тобто усю інформацію, необхідну для знаходження потрібного елемента на сторінці і подальшій роботі з ним [12].

Якщо з браузером середньостатистичний користувач знайомий, то driver, скрипт, WebElement та By потребують окремого розгляду.

2.1.1 Driver

Ініціалізувати driver можна двома способами (даний приклад, а також всі подальші приклади даної пояснювальної записки до бакалаврської кваліфікаційної роботи, буде наведено мовою програмування Java):

```
driver = new ChromeDriver();

або

ChromeOptions options = new ChromeOptions();
options.setPageLoadStrategy(PageLoadStrategy.NONE);
driver = new ChromeDriver(options);
```

У першому випадку ми створюємо «голий» драйвер без будь-яких параметрів, тоді як у другому спочатку ми вказуємо необхідні нам налаштування браузера, як от режим перегляду, інформація про користувача, стратегія завантаження сторінки, тощо, і тільки потім створюємо драйвер, передаючи йому бажані параметри [13]. Про додаткові можливості ініціалізації драйвера йтиметься далі.

До методів драйвера за замовчуванням відносяться: пошук елемента/-ів на сторінці, відкриття та закриття сторінки, переключення між вкладками, отримання заголовку сторінки, закриття браузера, вилучення посилання на поточну сторінку, різноманітні очікування елементів, управління параметрами, отримання HTML коду сторінки, та інші. Повний список методів наведений на рис. 2.1.1.

```
driver.
    findElement(By by)                                  WebElement
    get(String url)                                     void
    switchTo()                                          TargetLocator
    getTitle()                                          String
    quit()                                              void
    close()                                             void
    getCurrentUrl()                                     String
    wait(long timeoutMillis)                            void
    wait()                                              void
    wait(long timeoutMillis, int nanos)                 void
    manage()                                            Options
    navigate()                                          Navigation
    findElements(By by)                                 List<WebElement>
    getPageSource()                                     String
    getWindowHandle()                                   String
    getWindowHandles()                                  Set<String>
    equals(Object obj)                                  boolean
    hashCode()                                          int
    toString()                                          String
    getClass()                                          Class<? extends WebDriver>
    notify()                                            void
    notifyAll()                                         void
    Press ⏎ to insert, ⇥ to replace  Next Tip
```

Рисунок 2.1.1 – Методи WebDriver за замовчуванням

Слід також зазначити, що представлені методи можуть мати в собі ще декілька методів-наслідників, а ті, в свою чергу, також. Реалізація найбільш популярних з вищезазначених методів у коді виглядає наступним чином:

Відкриття сторінки та навігація [14]:

- `driver.get("https://google.com");` // Відкриття сторінки за посиланням, причому вказувати мережевий протокол обов'язково;
- `driver.navigate.to("https://google.com");` // Відкриття сторінки за посиланням, але в даному випадку driver буде очікувати повного завантаження сторінки;
- `driver.navigate().refresh();` // Оновити сторінку, відповідник

до ключа F5;

- `driver.navigate().back();` // Аналог кнопки «Назад», повернення на попередню сторінку;
- `driver.navigate().forward();` // Аналог кнопки «Вперед», перехід на наступну сторінку;
- `driver.getWindowHandles();` // Отримати список відкритих вкладок (вікон);
- `driver.switchTo().window("");` // Переключитися на вікно.

Закриття сторінки, вкладки [15]:

- `driver.close();` // Закрити поточну вкладку або вікно;
- `driver.quit();` // Закрити браузер.

Отримання корисної інформації про сторінку [16]:

- `driver.getPageSource();` // Отримати HTML код сторінки;
- `driver.getCurrentUrl();` // Отримати посилання на активне джерело в поточному вікні;
- `driver.getTitle();` // Отримати заголовок сторінки.

2.1.2 By

Окрім шляхів взаємодії з вікном браузера, є також безліч шляхів взаємодії з численними елементами на сторінці: формами, кнопками, текстом, списками, модальними вікнами, повідомленнями, тощо. Але для того, що отримати елемент, його необхідно знайти на сторінці. З цим допомагає абстракція By.

Шукати елемент можна за декількома критеріями:

- `By.id` – пошук елемента по атрибуту ID;
- `By.tagName` – пошук елемента по атрибуту назви тега;
- `By.className` – пошук елемента по назві класу, використовується коли на сторінці наявний набір однотипних елементів, причому складені імена класів не дозволені;
- `By.cssSelector` – пошук елементу за значенням CSS селектору, що визначає до якого елементу або групи елементів було застосовано певний

стиль;

- `By.name` – пошук елемента по атрибуту name;
- `By.linkText` – пошук елемента по конкретному тексту, якщо він заздалегідь відомий;
- `By.partialLinkText` – пошук елемента по частковому збігу текста, якщо збігається декілька елементів, буде повернуто лише перший;
- `By.xpath` – знаходить елементи за значенням xpath селектору [17].

В кодовому представленні це виглядатиме наступним чином [18]:

```
    driver.findElement(By.id("lname"));
    driver.findElement(By.tagName("a"));
    driver.findElement(By.className("information"));
    driver.findElement(By.cssSelector("#fname"));
    driver.findElement(By.name("newsletter"));
    driver.findElement(By.linkText("Selenium Official
Page"));
    driver.findElement(By.partialLinkText("Official
Page"));
    driver.findElement(By.xpath("//input[@value='f']"));
```

При здійсненні пошуку елементу на сторінці необхідно розуміти, що якщо однотипних елементів, які відповідають критерію пошуку, декілька, то повертатиметься перший елемент у списку.

Для пошуку групи елементів існує метод `findElements(By by)`, який повертатиме не лише перший підходящий результат, а список усіх елементів, які підпадають під пошуковий запит.

Щодо застосування, то єдиних правил та вимог щодо того, коли який локатор використовувати немає і до кожного з вищенаведених способів можна звертатися за необхідністю. Єдиною рекомендацією залишається не змішувати xpath та CSS селектори в рамках одного фреймворку задля підтримки консистентності та чистоти коду. Зазвичай вибір того чи іншого виду селекторів обумовлюється наявністю попереднього досвіду команди у роботі з одним із них та/або

побажаннями замовника, якщо такі є [19].

2.1.3 WebElement

Тепер, коли механізм пошуку елемента на сторінці став зрозумілішим, пропонуємо розглянути які дії можна виконувати з елементами на сторінці.

По-перше, для того щоб скористатися дефолтними (за замовчуванням) методами, необхідно ініціалізувати елемент за допомогою класу WebElement:

```
    WebDriver driver = new ChromeDriver();
    WebElement webElement = driver.findElement(new
By.ByCssSelector("[class='gLFyf']"));
```

Тепер ми можемо взаємодіяти зі змінною `webElement`. Далі, за аналогією з driver, бачимо методи для елементів сторінки за замовчуванням: ввести, вилучити або порівняти текст (для текстових полів, яких ще називають інпутами, від англ. input), підтвердити введення інформації, отримати значення атрибуту, нажати, дізнатися розташування або розмір, виконати просту перевірку на наявність, відсутність, готовність елемента до взаємодії, різноманітні види очікувань, тощо. Повний список методів за замовчуванням, доступних до виклику та використання в роботі з елементами сторінки, відображений на рис. 2.1.3.1.

Важливо враховувати, що проініціалізована змінна `webElement` вказує не на селектор, а на певний елемент в структурі DOM (Document Object Model, об’єктна модель документа), зі своїм унікальним ID. Дана структура є динамічною, тому за цим же шляхом через будь-які зміни на сторінці може з’явитися елемент вже з іншим ID, і при зверненні до нього ми отримаємо виключення. З цієї причини, пошук елемента на сторінці необхідно виконувати безпосередньо перед його використанням, цей процес також називають відкладеною або лінивою ініціалізацією селекторів [20].

```
webElement.
    sendKeys(CharSequence... keysToSend)                    void
    getAttribute(String name)                               String
    click()                                                 void
    equals(Object obj)                                      boolean
    findElement(By by)                                      WebElement
    getText()                                               String
    submit()                                                void
    clear()                                                 void
    findElements(By by)                                     List<WebElement>
    getAccessibleName()                                     String
    getAriaRole()                                           String
    getCssValue(String propertyName)                        String
    getDomAttribute(String name)                            String
    getDomProperty(String name)                             String
    getLocation()                                           Point
    getRect()                                               Rectangle
    getShadowRoot()                                         SearchContext
    getSize()                                               Dimension
    getTagName()                                            String
    isDisplayed()                                           boolean
    isEnabled()                                             boolean
    isSelected()                                            boolean
    hashCode()                                              int
    toString()                                              String
    getScreenshotAs(OutputType<X> target)                   X
    getClass()                                              Class<? extends WebElement>
    notify()                                                void
    notifyAll()                                             void
    wait()                                                  void
    wait(long timeoutMillis)                                void
    wait(long timeoutMillis, int nanos)                     void
    Press ⏎ to insert, ⇥ to replace  Next Tip
```

Рисунок 2.1.3.1 – Методи WebElement за замовчуванням

Окрім досить очевидних при достатньому рівні володіння англійською мовою методів, пропонуємо зупинитися детальніше на методі `getAttribute(String attributeName)`, що повертає текстове значення певного заданого атрибута елемента. Це значення береться не з вкладки Element у консолі розробника в браузері, а з вкладки Properties, різницю можна побачити на рис. 2.1.3.2.

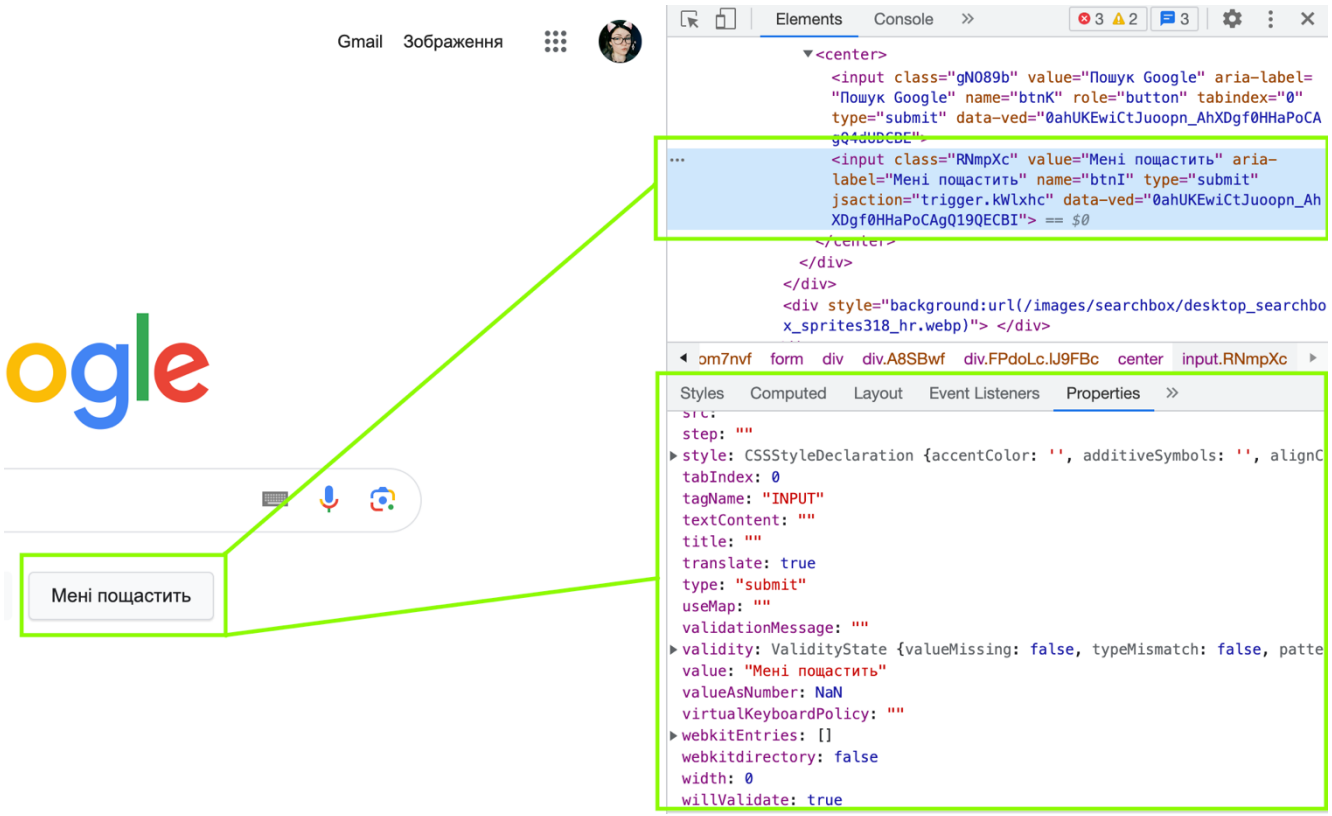

Рисунок 2.1.3.2 – Вкладка Properties та наявні в ній атрибути

Як бачимо з рисунку, для будь-якого елементу передбачена достатня кількість атрибутів, серед яких: `textContent` (кінцевий видимий текст елемента, який відображається користувачу), `outerHTML` (весь HTML з властивостями), `value` (значення вмісту для текстових полів), `tagName` (назва тегу елемента), `className` (назва класу, якому належить елемент), атрибути, пов'язані з визначенням розміру елемента, такі як, наприклад, `size` (розмір елемента), `scrollHeight` (висота елемента в пікселях) та `scrollWidth` (ширина елемента в пікселях), атрибути, що надають інформацію про сусідні елементи накшталт `previousSibling` (попередній аналогічний елемент в межах одного класу), `nextSibling` (наступний аналогічний елемент в межах одного класу), `parentElement` (назва батьківського класу), різноманітні булеві атрибути, які повертають значення true або null по типу `disabled` (чи відключений елемент), `draggable` (чи можна елемент переміщати по сторінці), `hidden` (чи елемент прихований).

2.1.4 Скрипт

Для створення простого скрипту – інструкції перевірки тестового сценарію – необхідно придумати якою буде сама перевірка. Пропонуємо почати з нескладного користувацького кейсу: користувач відкриває браузер Google Chrome, опиняється на домашній сторінці Google, здійснює простий пошуковий запит та натискає Enter.

Все це ми перенесемо в код пам'ятаючи про те, що браузер необхідно спершу ініціювати та явно передати бажану для першого відкриття сторінку, а після здійснення пошукового запиту треба перевірити, що у вкладці з'явились релевантні результати пошуку. Це можна зробити завдяки перевірці заголовку сторінки.

Просимо звернути увагу на те, що в даному розділі бакалаврської кваліфікаційної роботи перевірки виконуються шляхом виводу необхідної інформації в консоль, а не безпосередньо за рахунок написання та виконання тесту у середовищі програмування, адже на даному етапі це не принципово.

Приклад скрипту, написаного з застосуванням вищезазначеної інформації на практиці, наведено на рис. 2.1.4.1.

```
public static void main(String[] args) throws Exception {

    // Створюємо опції користувача
    ChromeOptions options = new ChromeOptions();
    options.addArguments("--remote-allow-origins=*");
    // Ініціюємо браузер, в який передаємо опції
    WebDriver driver = new ChromeDriver(options);
    // Відкриваємо стартову сторінку Google
    driver.get("https://www.google.com");
    // За CSS селектором знаходимо елемент поля для пошуку
    WebElement element = driver.findElement(new By.ByCssSelector("[class='gLFyf']"));
    // Вводимо текст в поле пошуку
    element.sendKeys( ...keysToSend: "Тестовий пошуковий запит");
    // Виконуємо пошук
    element.submit();
    // Перевіряємо заголовок сторінки
    System.out.println("Page title is: " + driver.getTitle());
    // Закриваємо браузер
    driver.quit();
```

Рисунок 2.1.4.1 – Скрипт простого сценарію взаємодії з web-сторінкою

Поетапні результати виконання скрипту наведені нижче на рисунках 2.1.4.2-2.1.4.5. Скріншоти було зроблено завдяки ставленню процесу виконання коду на запуск в необхідних точках.

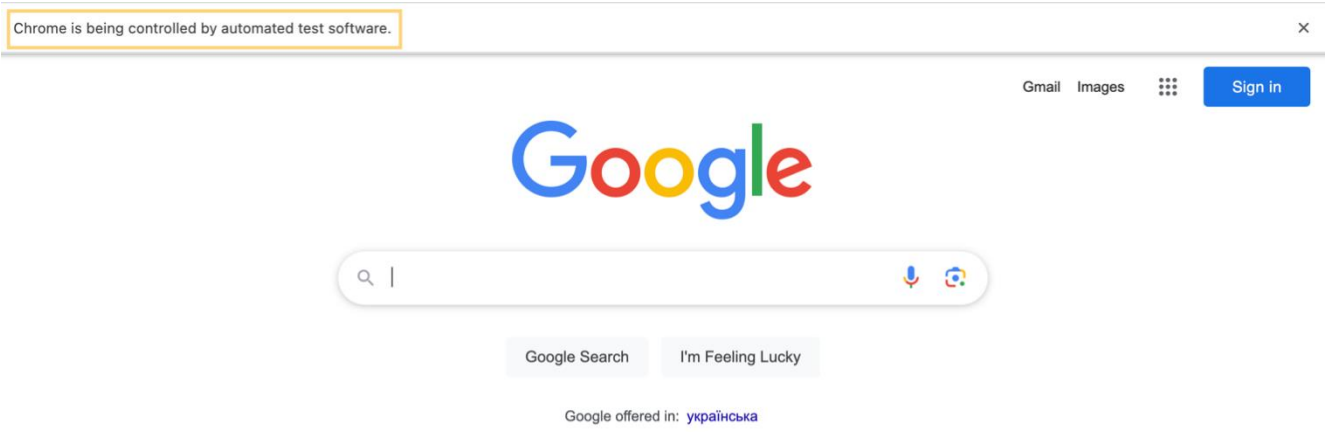

Рисунок 2.1.4.2 – Відкриття домашньої сторінки Google Chrome

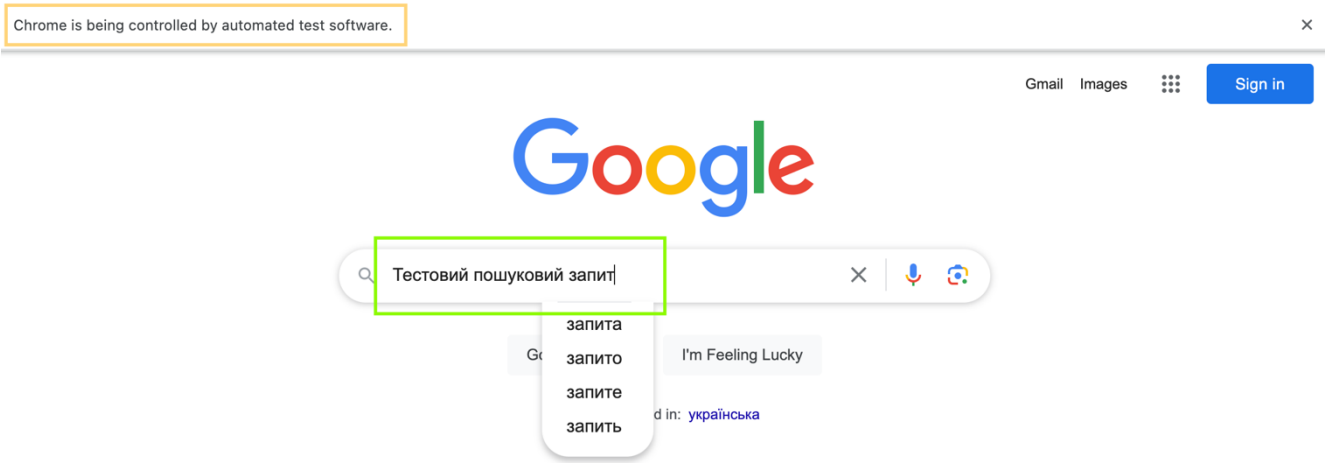

Рисунок 2.1.4.3 – Введення пошукового запиту та емуляція натискання Enter

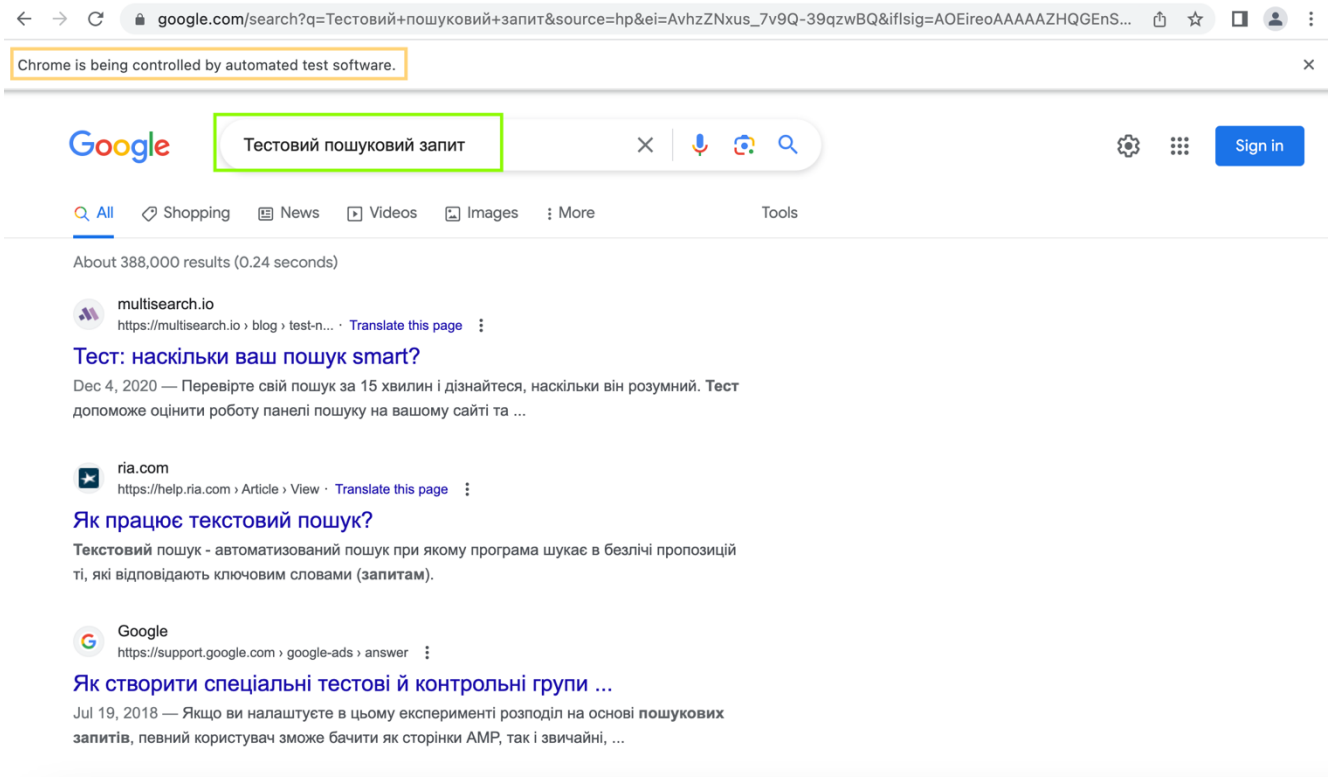

Рисунок 2.1.4.4 – Результат пошукового запиту у вікні браузера

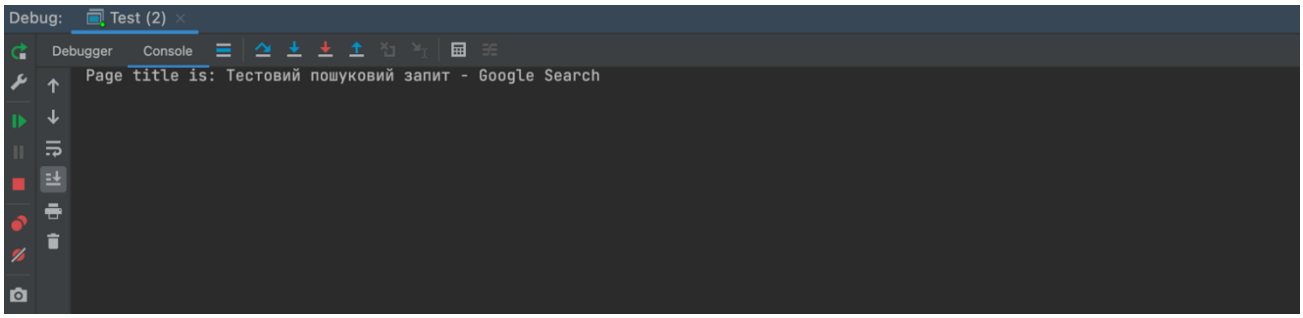

Рисунок 2.1.4.5 – Виведення заголовку сторінки в консоль

2.1.5 Додаткові можливості Selenium WebDriver

Окрім основних та найбільш розповсюджених способів взаємодії зі сторінками та елементами, існують й інші, які зустрічаються рідше, проте являються не менш цікавими для ознайомлення.

Так, наприклад, в опції драйвера можна передати готовність сторінки браузера до взаємодії.

Перед виконанням кожної команди Selenium перевіряє значення властивості `document.readyState` і за замовчуванням призупиняє виконання команди до

тих пір, поки дана властивість не набуде значення “complete”.

У процесі обробки сторінки браузер змінює вказану властивість, яка відображає інформацію про поточний етап завантаження web-контенту:

- `loading` – означає, що сторінка ще завантажується;
- `interactive` – означає, що основний зміст сторінки завантажено та відмальовано, користувач уже може з нею взаємодіяти, але ще продовжується завантаження додаткових ресурсів;
- `complete` – означає, що всі додаткові ресурси також завантажені.

Для того щоб драйвер не очікував значення “complete”, можна при ініціалізації драйвера встановити відповідне значення для capability з назвою `pageLoadStrategy`. В кодовій реалізації цей процес виглядатиме наступним чином:

```
ChromeOptions options = new ChromeOptions();
options.setPageLoadStrategy(PageLoadStrategy.NONE);
driver = new ChromeDriver(options);
```

Допустимими значеннями для браузера Google Chrome є:

- `NORMAL` – очікувати, поки властивість `document.readyState` прийме значення “complete”, значення, встановлене за замовчуванням;
- `EAGER` – очікувати, поки властивість `document.readyState` прийме знамення “interactive”;
- `NONE` – не очікувати [13].

Також можна налаштувати необхідні cookies. Алгоритм дій для цього наступний:

1) відкриваємо необхідну сторінку;
2) встановлюємо cookie пам’ятаючи про те, що вони будуть дійсними для всього web-сайту;
3) передаємо створені cookie в драйвер.

В реалізації за допомогою коду наведений вище алгоритм матиме наступний вигляд:

```
    driver.get("http://www.example.com");
    Cookie cookie = new Cookie("key", "value"); // В даному випадку "key" це назва cookie, а "value" його значення
    driver.manage().addCookie(cookie);
    Set<Cookie> allCookies = driver.manage().getCookies(); // Виводимо всі cookie, доступні для поточного URL
    for (Cookie loadedCookie : allCookies) {
        System.out.println(String.format("%s -> %s", loadedCookie.getName(), loadedCookie.getValue()));
```

Для видалення cookie можна скористатись одним з трьох способів [20]:

```
    driver.manage().deleteCookieNamed("CookieName"); // Використовуючи ім'я
    driver.manage().deleteCookie(loadedCookie); // Використовуючи об'єкт Cookie
    driver.manage().deleteAllCookies(); // Або все відразу
```

Можна також зберігати ілюстративні матеріали у вигляді скріншотів для поліпшення історії результатів прогонів [15]:

```
    File screenshotFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(screenshotFile, new File("/Users/username/folder/screeenshot.png"));
```

Окрім цього, можна відтворити взаємодію через емуляцію мобільного девайсу, яку реальний користувач може викликати за допомогою інструментів розробника в браузері. Приклад такої емуляції наведено на рис. 2.1.5.

В коді зробити це можна наступним чином:

```
    Map<String, Object> mobileEmulation = new
HashMap<String, Object>();
    mobileEmulation.put("deviceName", "iPhone 5");
    ChromeOptions options = new ChromeOptions();
    options.setExperimentalOption("mobileEmulation",
mobileEmulation);
    driver = new ChromeDriver(options);
```

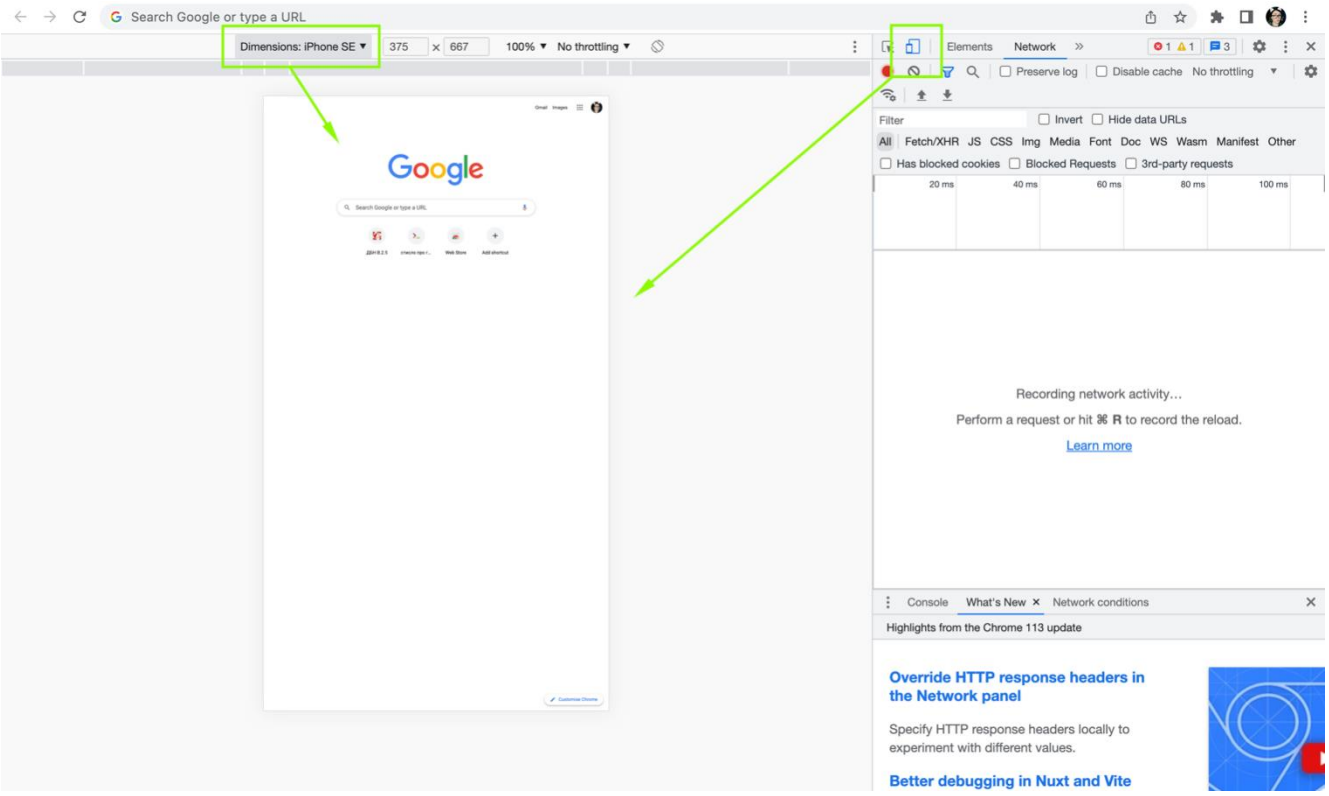

Рисунок 2.1.5 – Емуляція роботи в мобільному пристрої через web-браузер Google Chrome

Для елементів сторінки є цікавий клас `Actions`, який дозволяє працювати з динамічними (рухомими) елементами, відтворюючи нетипову роботу з комп'ютерною мишкою: перетягувати елементи з одного положення в інше, наводити з очікуванням появи додаткових компонентів, клікати з зажимом, тощо [22].

Для цього необхідно створити об'єкт класу `Actions` та передати в нього драйвер, і потім викликати необхідні методи створеного об'єкту. Наприклад:

```
    Actions actions = new Actions(driver);
    actions.moveToElement(elem).perform();

    або

    Actions builder = new Actions(driver);
    Action dragAndDrop = builder.dragAndDrop(From,
To).build();
    dragAndDrop.perform();
```

Проаналізувавши можливості Selenium WebDriver бачимо, що інструмент дійсно допомагає якнайточніше відтворити реальну поведінку звичайного користувача, пропонуючи безліч методів взаємодії з вікном браузера та елементами на сторінці за замовчуванням, а також можливість створювати власні функції у випадку якщо наявних виявиться недостатньо.

2.2 Retrofit як інструмент автоматизації API-сервісу, його особливості

Для кращого розуміння що таке Retrofit і для чого він потрібен, пропонуємо спочатку нагадати значення таких термінів, як REST, HTTP та JSON.

REST (Representational State Transfer, з англ. «передача репрезентативного стану») – це архітектурний стиль взаємодії компонентів розподілених застосунків в мережі, в основі якого закладено принципи функціонування HTTP. Простішими словами, – це набір правил стосовно того, як розробнику краще організувати написання коду серверного застосунку таким чином, щоб всі системи легко обмінювались даними і застосунок було можливо масштабувати [23].

HTTP (HyperText Transfer Protocol, з англ. «протокол передачі гіпертекстових документів») – це протокол передачі даних, який припускає, що клієнтська програма (браузер) здатна відображати web-сторінки та інші файли у зручній для кінцевого користувача формі, і для коректного відображення таких сторінок і файлів працює по принципу запит-відповідь, дізнаючись мову та кодування символів web-сторінки, її версію, тощо. Є певний перелік HTTP методів, основними з яких є GET, POST, PATCH, DELETE, кожен метод має свою чітко визначену структуру запиту та відповіді [24].

JSON (JavaScript Object Notation, з англ. «запис об’єктів JavaScript») – це текстовий формат обміну структурованими даними через мережу за рахунок серіалізації, який базується на тексті, легкодоступному для людського сприйняття. Під серіалізацією прийнято розуміти процес перетворення будь-якої структури даних у послідовність бітів, а зворотній процес відновлення початкового стану структури з набору бітів називається десеріалізацією [25].

Тож Retrofit – це клієнт для Java та Android, який дозволяє легко отримати та завантажити JSON (або будь-яку іншу структуру даних) через web-сервіс на основі REST. Retrofit працює на основі декількох бібліотек: бібліотек з описом конвертерів для серіалізації та десеріалізації даних (в залежності від того, яким інструментом розробнику зручно користуватися, в нашому випадку це буде бібліотека Gson), а також бібліотеки OkHttp для обробки HTTP-запитів.

Для роботи з Retrofit знадобляться наступні три класи:

- Model class (клас-модель) або ж POJO (Plain Old Java Object), суть якого полягає в тому щоб реалізувати JSON-відповідь від сервера як модель, тобто клас мовою програмування Java;
- Interface (інтерфейс), що визначатиме можливі HTTP операції по типу GET, POST, DELETE, тощо;
- Retrofit.Builder клас з визначенням кінцевої точки URL для HTTP операцій, тобто клас з методом `baseUrl()` для опрацювання результатів [26].

Реалізація та наповнення класів моделей та інтерфейсів залежатиме безпосередньо від об’єкта тестування і розглядатиметься безпосередньо у 3 розділі даної роботи, а ось клас Builder всюди виступає головним клієнтом, а тому буде схожим для більшості випадків і виглядатиме наступним чином:

```
    public class ExternalEventClient {

        public ExternalEventService;
        public OtherService;    // Тут вказуються класи моделей та
інтерфейсів

        public String BASE_URL = "http://api.com/1/";

            OkHttpClient client = new
OkHttpClient.Builder()
                    .connectTimeout(20, TimeUnit.SECONDS)
                    .writeTimeout(20, TimeUnit.SECONDS)
                    .readTimeout(20, TimeUnit.SECONDS)
                    .addInterceptor(new
HTTPLogInterceptor())
                    .addInterceptor(new AuthInterceptor())
                    .build();

            Retrofit retrofit = new Retrofit.Builder()
                    .client(client)
                    .baseUrl(BASE_URL)
                    .addConverterFactory(new
JSONAPIConverterFactory(resourceConverter)) // Якщо потрібен
JSON API

.addConverterFactory(GsonConverterFactory.create())
                    .build();

            externalEventService =
retrofit.create(ExternalEventService.class,
OtherService.class);

             // А тут безпосередньо ініціалізуються класи з інтерфейсами
        }
    }
```

Як запит, так і відповідь можна перехопити та внести туди необхідні зміни за допомогою класу `Interceptor` з бібліотеки OkHttp. Найчастіше це робиться з

метою того, щоб, наприклад, передавати дані для авторизації протягом всієї сесії без необхідності створювати токен для кожного послідуючого запиту. Також розповсюдженим застосуванням даного механізму є логування (виведення в консоль поточного статусу роботи програми) та обробка серверних помилок.

При роботі з `Interceptor`, спочатку створюється об'єкт «перехоплювача» і передається в OkHttp, який у свою чергу слід явно підключити до `Retrofit.Builder` через метод `client()`. Реалізація в коді виглядатиме так:

```
    public class ExternalIventInterceptor implements
Interceptor{

        public String slateAddonToken;

        @Override
        public Response intercept (Chain chain) throws
IOException {
            Request newRequest =
chain.request().newBuilder()
                    .addHeader("Authorization",
slateAddonToken)
                    .addQueryParameter("key", "value")
                    .build();
            return chain.proceed(newRequest);
        }

        public ExternalIventInterceptor(String
slateAddonToken){
            this.slateAddonToken = slateAddonToken;
        }
```

Розглянувши та проаналізувавши можливості Retrofit, робимо висновок що це зручний та гнучкий високорівневий інструмент для роботи з API частиною продукту, який дозволяє перенести тестування інтеграцій в код з застосуванням основних принципів ООП.

2.3 Патерни та підходи програмування, що використовуються в автоматизації

Автоматизація тестування – це те ж програмування, а тому в своїй практиці активно використовує розповсюджені патерни проектування та сучасні підходи до програмування задля того, щоб вирішувати типові проблеми при розробці програмного забезпечення ефективно та з найменшими затратами по часу та ресурсам. Завдяки створенню типових шаблонів коду, які легко перевикористовувати, процес програмування стає простішим при підвищенні загальної ефективності розробки.

2.3.1. Найбільш розповсюджені патерни проектування

Патерн проектування – це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм. Він являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, реалізація якого має бути підлаштованою під потреби тієї чи іншої програми.

Патерни вирішують три основних питання: стандартизація коду, перевірені рішення та слугування в якості загального уніфікованого словника програмістів. Всі вони відрізняються за рівнем складності, деталізації, охоплення проектованої системи і за призначенням [27].

Найбільш популярним патерном проектування при роботі з Selenium WebDriver є PageObject (з англ. об’єкт сторінки) – це патерн, в якому кожна окрема сторінка продукту описується окремим класом. Принцип роботи даного патерну полягає в тому, що абстрагуватись від окремих елементів HTML та інкапсулювати їх у функції доступу до елементів інтерфейсу вищого рівня, як їх бачить користувач.

За своєю структурою PageObject зазвичай містить лише код для доступу до елементів керування і не містить ніяких тестових припущень. Єдині перевірки, які здійснюються під час створення об’єкта – це те, що інтерфейс та елементи керування на ньому завантажились та відобразились коректно. З елементами керування можна або взаємодіяти, або отримувати від них інформацію. Так,

наприклад, галочка може відображатись у змінну типу boolean.

Основними перевагами даного патерну проектування є:

- розділення логіки роботи та представлення;
- зменшення дублювання коду для пошуку елементів керування застосунком;
- об’єднання всіх дій по взаємодії з web-сторінкою в одному місці;
- при змінах інтерфейсу, що не зачіпають логіки продукту, потрібно буде змінити лише PageObject, а не логіку тестів [28].

Наступним популярним та доступним в плані розуміння для новачків є Builder (з англ. будівельник) – це патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів. Зазвичай, в коді його можна впізнати таким чином: клас має один створюючий метод і декілька допоміжних методів налаштування створюваного продукту. В такому випадку викликати його прийнято ланцюжком, метод за методом.

До переваг Builder можна віднести:

- створення продуктів покроково;
- забезпечення перевикористання одного і того самого коду для створення різноманітних продуктів;
- ізоляція складного коду конструювання продукту від його головної бізнес-логіки.

Разом з тим, даний патерн проектування має і свої недоліки, серед яких: ускладнення коду програми за рахунок додаткових класів, а також прив’язка клієнта до конкретних класів-будівельників, адже в інтерфейсі будівельника може не опинитись методу отримання необхідного результату [27].

І, нарешті, третім основним патерном проектування виступає Singleton (з англ. одинак) – це патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього, дозволяючи «достукатись» до даного об’єкта з будь-якої точки програми.

Він застосовується у випадках, коли в програмі повинен бути єдиний екземпляр якого-небудь класу, доступний усім клієнтам (наприклад, спільний доступ до бази даних з різних частин програми); або коли було прийняте рішення мати більше контролю над глобальними змінними.

Перевагами сінглтону є:

- гарантування наявності єдиного екземпляра класу;
- надання глобальної точки доступу до бажаного об’єкту;
- реалізація відкладеної ініціалізації об’єкта-одинака.

Недоліками сінглтону, в свою чергу, виступають:

- порушення принципу єдиного обов’язку класу;
- маскування слабкостей тест-дизайну;
- вірогідність виникнення проблем з багатопоточністю [27].

В практичній частині (розділ 3) даної бакалаврської кваліфікаційної роботи будуть застосовані елементи всіх трьох вищенаведених патернів проектування, адже кожен з них має явні сильні сторони.

2.3.2 Рекомендації та best practices в автоматизації тестування

Деякі підходи, які рекомендується використовувати в автоматизації продуктів було зазначено в попередніх частинах даного розділу. Пропонуємо їх підсумувати, а також зазначити декілька незгаданих best practices.

Необхідно добре знати та орієнтуватися в можливостях обраних інструментів автоматизації тестування задля написання максимально ефективних автоматизованих тестів. Так, наприклад, визначення стратегії взаємодії зі сторінкою заздалегідь шляхом передавання параметру очікування певного стану завантаження контенту може значно пришвидшити час проходження тесту.

При роботі з елементами сторінки, пошук конкретного елементу необхідно виповнювати безпосередньо перед роботою з ним, що обумовлено динамічною структурою DOM, інакше є ризик того, що за обраним локатором буде знайдено елемент під іншим ID або жодного елементу, і це в свою чергу призведе до завершення програми через прокидування виключення.

Звичайно, можна використовувати різні способи пошуку елементів на сторінці в залежності від дизайну самого продукту та потреб бізнесу та/або технічної команди, проте задля збереження консистентності та читабельності коду, а також спрощення його підтримки, рекомендується використовувати один і той же тип локаторів, не змішуючи, наприклад, CSS та xPath селектори.

Працюючи з очікуваннями, рекомендується на противагу неявним очікуванням, які призупиняють роботу всієї програми на строго визначений час без прив'язки до конкретного результату очікування, використовувати явні очікування, які будуються на тому, щоб періодично перевіряти елемент на предмет готовності до певної взаємодії (наявність, видимість, клікабельність, тощо). Це робить перевірки більш очевидними та зрозумілими для читання. До того ж, деякі випадки, як, наприклад, зникнення елементу, можна перевірити лише за допомогою явних очікувань.

2.4 Основні принципи написання автоматизованих тестів

Будь-які автоматизовані тести пишуться на основі тестової документації. Тестова документація (або тестові артефакти) – це задокументовані процеси, правила та порядок виконання тестування на різних рівнях. До тестової документації зазвичай відносять тест-план, чек-лист, тест-кейс, баг-репорт та матрицю покриття вимог [29]. Основними причинами для написання, ведення та підтримування тестової документації в актуальному вигляді виступають:

1) якщо ми не говоримо про старі проекти з заздалегідь розробленим продуктом, де нічого нового не створюється, а лише підтримується існуючий функціонал, то тести пишуться на основі вимог. Таким чином, ми розбираємо вимоги і тестуємо їх ще до того, як побачити саму програму;
2) для «наслідування» – якщо нова людина приходить на проект, їй буде легше вникнути і, наприклад, приступити до регресії;
3) для систематизації знань про продукт, якщо тримати все в голові – щось можна забути;

4) для звітності, якщо це бажання чи вимога з боку замовника;
5) як основа для подальшої автоматизації.

Найбільш розповсюдженим видом тестової документації виступає тест-кейс. Тест-кейс (test case) – це тестовий артефакт, суть якого полягає у виконанні деякої кількості дій та/або умов, необхідних для перевірки певної функціональності програмної системи, що розробляється [30].

Характеристиками тест-кейсів є:

- однозначність: тест-кейси не повинні трактуватися по-різному;
- унікальність: тест-кейси перевіряють унікальну поведінку;
- індивідуальність: один тест-кейс перевіряє одну умову;
- незалежність / атомарність: кожен тест можна виконувати окремо;
- повторюваність: сценарій, описаний в тест-кейсі, має бути легким та зрозумілим для проходження;
- стабільність: за результатами виконання тест-кейсу можна отримати лише успіх (PASS) або помилку (FAIL), тест-кейси не можуть обриватися внутрішніми або зовнішніми причинами;
- ефективність;
- організованість: кожен тест-кейс повинен бути в своїй правильній категорії;
- фіксованість: публікація звітів має відбуватися на ресурсах, доступних всім працівникам, звіти повинні містити помилки (якщо є) з описанням та скріншотами або іншою наглядною інформацією, процентом успіху, часом виконання, тощо;
- підтримуваність: тестова документація має бути актуальною, забезпечити їй такий стан можна завдяки використання технік тест дизайну при написанні, також уникати дублювання і писати прості тест-кейси з простими перевірками;
- вартість довіри: в назві тест-кейсу має відображатися його суть;
- цінність: тестовий сценарій, покритий тест-кейсом, повинен бути

- важливий з точки зору бізнесу [31].

Структура даного артефакту полягає в так званій «трійці»:

Виконувана дія (Action) – Очікуваний результат (Expected result) – Фактичний результат (Test result).

Тест-кейси можуть бути різними за цілями перевірок: функціональними і нефункціональними, регресійними, смоук або аксептенс, методом чорного або білого ящиків, тощо.

Автоматизовані тести, в свою чергу, спираються на тест-кейси і несуть те ж смислове навантаження, просто реалізоване у вигляді коду, а не тексту. Попри це, найбільш розповсюдженою рекомендацією є автоматизовувати прості позитивні перевірки, щоб готові автотести виходили атомарними, незалежними один від одного та були швидкими у виконанні.

Висновки до розділу 2

У розділі 2 даної бакалаврської кваліфікаційної роботи було розглянуто Selenium WebDriver та Retrofit як інструменти для роботи з автоматизацією користувацького інтерфейсу та API сервісу web-продуктів, відповідно, розглянуто їх особливості, принципи роботи, основні та додаткові можливості, сучасні патерни проектування, доступні для розуміння та використання новачками, і підходи до автоматизації, включаючи рекомендації та best practices, а також визначено на основі чого пишуться автоматизовані тести та виокремлено базові вимоги до них.

В результаті дослідження та опрацювання даного розділу було сформовано уявлення про те, яким має бути фреймворк, що розроблятиметься в практичній частині даної роботи, а саме:

- отримані знання про широкий спектр можливостей інструментів для автоматизації Selenium WebDriver та Retrofit дозволить зробити свій код гнучким та максимально приближеним до відтворення дій реального користувача;
- огляд сучасних патернів проектування разом з їх перевагами і недоліками дозволить виокремити необхідні властивості з кожного і створити такий

фреймворк, який був би легкочитабельним, -підримуваним та -змінюваним;

- ознайомлення з рекомендаціями та підходами до автоматизації тестування дозволить увібрати найкращі практики та скористатися досвідом більш досвідчених колег, за можливості уникаючи помилок початківців;
- поглиблення у теорію тестування, зокрема види та цілі написання тестової документації, дозволить краще зрозуміти специфіку того, які сценарії мають бути автоматизовані в першу чергу і чому.

3 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ФРЕЙМВОРКУ ДЛЯ АВТОМАТИЗАЦІЇ WEB-САЙТУ

3.1 Встановлення Maven та TestNG в якості допоміжних інструментів

3.1.1 Встановлення, огляд та налаштування Maven

Maven – це зручний інструмент для автоматизації збірки проектів, що дозволяє управляти та складати програми. До його переваг можна віднести декларативний опис проекту (Maven налаштовується за замовчуванням, зміни потрібні тільки тоді, коли розробник сам бажає відійти від стандартних налаштувань), гнучке управління залежностями (інструмент сам здатен їх підвантажувати), а також незалежність від операційної системи.

Стандартна структура для Maven-проекту наведена на рис. 3.1.1.1 і виглядає наступним чином:

- в папці `src/main/java` зберігаються Java-класи (моделі сервісів та сторінок, тощо);
- в папці `src/main/resources` зберігаються ресурси, які використовуються розроблюваним застосунком (HTML-сторінки, картинки, таблиці стилів тощо);
- папка `src/test` слугує безпосередньо для тестів.

Усі операції здійснюються на основі файлу `pom.xml`, який містить такі теги:

- `project` – є базовим і містить всю інформацію про проект;
- `properties` – містить особливі налаштування, такі як кодування файлу та використовувана версія компілятора Java;
- `dependencies` – залежності на інші проекти, кожну залежність необхідно виділити тегом `dependency` окремо і вказати унікальні ідентифікаційні дані;
- `plugins` – плагіни, які допомагають виконати додатковий функціонал, поділяються на плагіни збірки `build plugins` і плагіни звіту `reporting plugins`;

- `parameters` тощо.

Стандартний вигляд `pom.xml` файлу наведений на рис. 3.1.1.2.

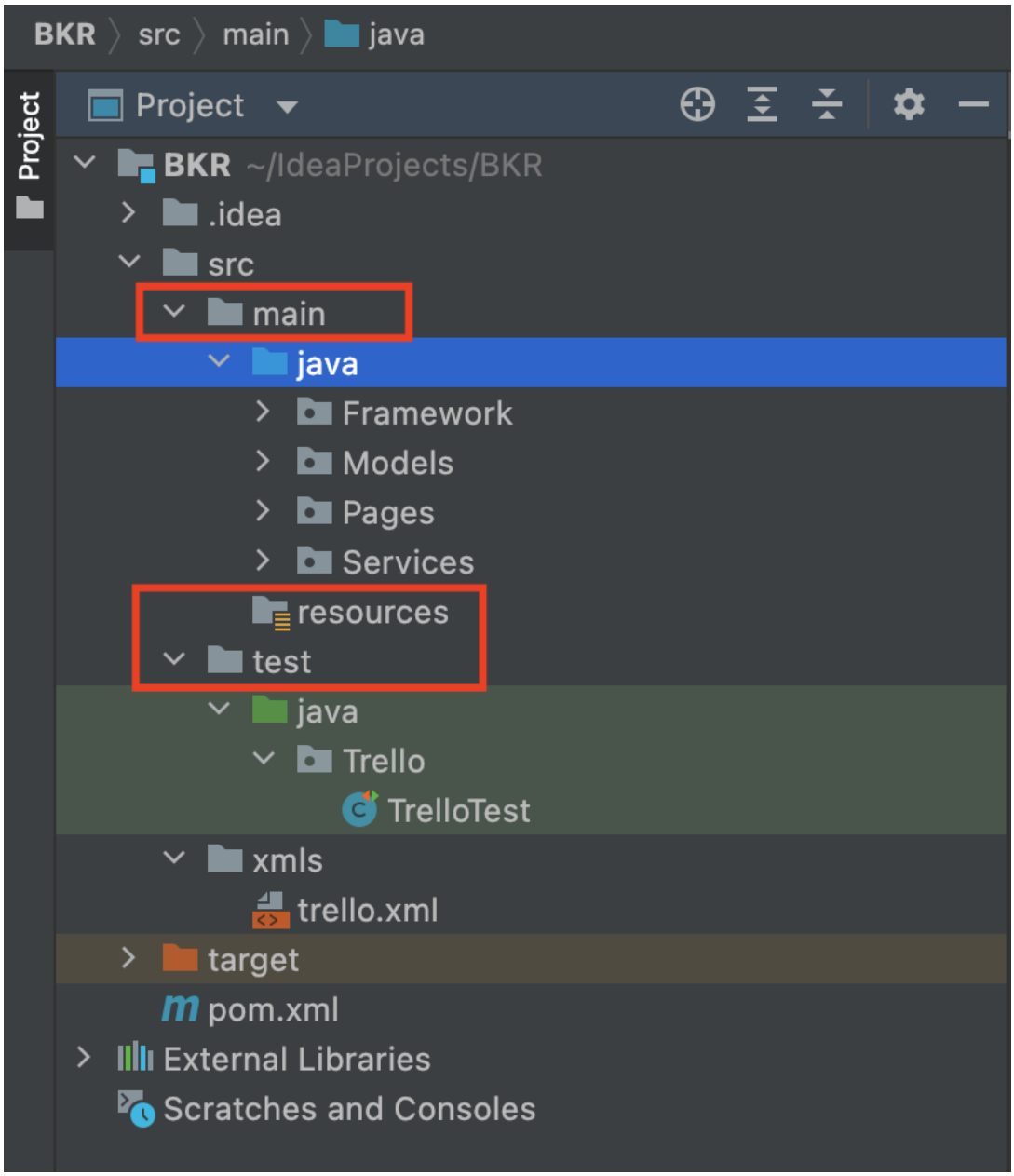

Рисунок 3.1.1.1 – Стандартна структура Maven проекту

pom.xml (BKR)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>org.example</groupId>
8      <artifactId>BKR</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <dependencies>
12         <dependency...>
18         <dependency...>
23         <dependency...>
28         <dependency...>
34         <dependency...>
39         <dependency...>
44         <dependency>
45             <groupId>com.squareup.retrofit2</groupId>
46             <artifactId>converter-gson</artifactId>
47             <version>2.9.0</version>
48         </dependency>
49     </dependencies>
50
51     <properties>
52         <maven.compiler.source>11</maven.compiler.source>
53         <maven.compiler.target>11</maven.compiler.target>
54     </properties>
55
56 </project>
```

Рисунок 3.1.1.2 – Приклад pom.xml файлу

До залежностей, які нам знадобляться у розробці, належать: Selenium, TestNG, OkHttp, Retrofit, Gson, Log4j. Код для їх правильного підключення у проект за допомогою Maven можна знайти на офіційних сайтах.

Також для гнучкого управління проектом існують 9 фаз життєвого циклу проекту Maven:

1) фаза `clean`, під час якої видаляються всі скомпільовані файли з цільового каталогу “target” (місце, в якому зберігаються готові артефакти);
2) фаза `validate`, що перевіряє, чи вся необхідна інформація доступна для збірки проекту;
3) фаза `compile`, запуск якої спричиняє компілювання файлів з вихідним кодом;

4) фаза `test`, що запускає тести на виконання;
5) фаза `package`, що упаковує скомпільовані файли в .jar, .war та інші види архівів;
6) фаза `verify`, під час якої виконуються перевірки для підтвердження готовності упакованого файлу;
7) фаза `install`, після якої пакет з проектом розміщується в локальний репозиторій, в результаті чого він може використовуватися іншими проектами як зовнішня бібліотека;
8) фаза `site`, яка створює документацію проекту;
9) фаза `deploy`, після якої зібраний архів копіюється у віддалений репозиторій.

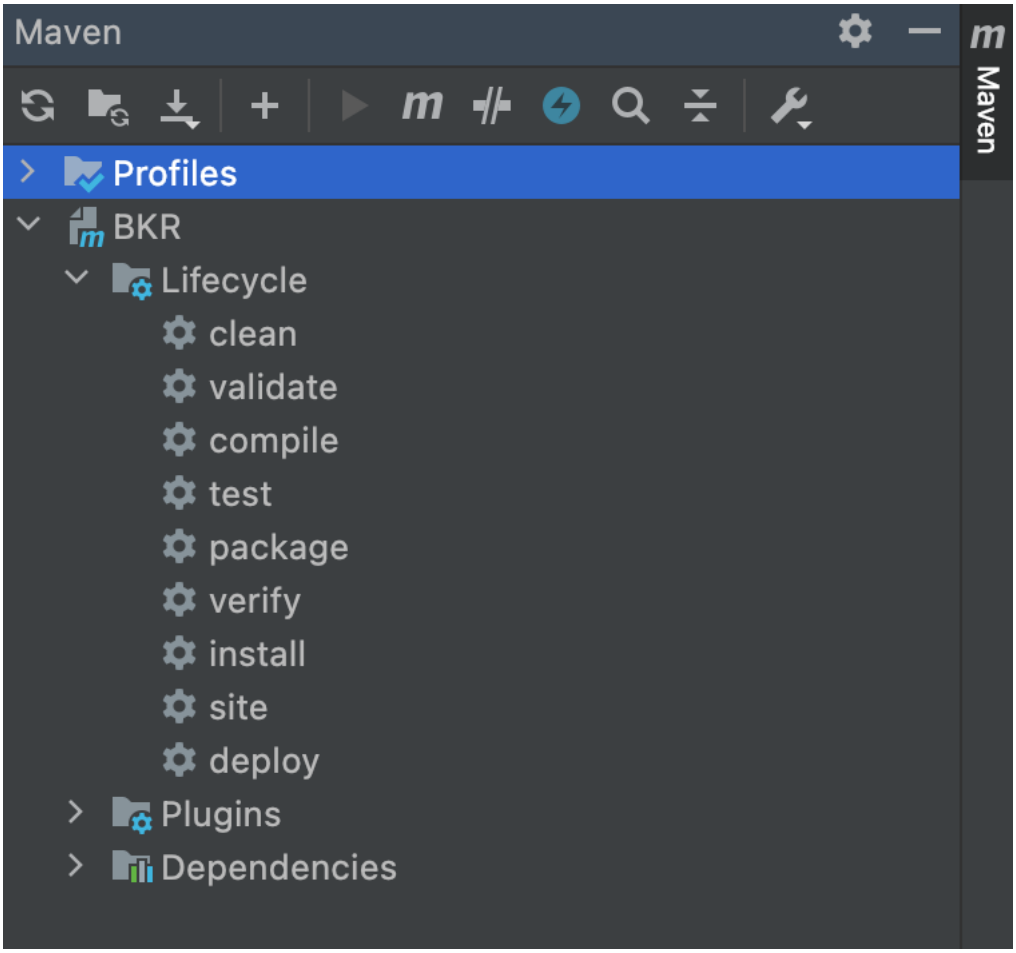

Рисунок 3.1.1.3 – Життєві фази проекту на вкладці Lifecycle

3.1.2 Встановлення, огляд та налаштування TestNG

TestNG – це тестовий фреймворк, покликаний задовольнити значну частину потреб тестування завдяки забезпеченню гнучких точок валідації і підтримці структурованості тестів [32].

До можливостей TestNG відносять:

- наявність зручних анотацій;
- використання XML (Extensible Markup Language, з англ. «розширювана мова розмітки») формату для гнучкого конфігурування тестів;
- підтримка data-driven тестування (тестування на основі даних; виконується за допомогою анотації `@DataProvider`);
- наявність залежних методів для тестування серверних програм;
- підтримка в таких середовищах програмування, як Eclipse, IDEA, Ant, Maven, Netbean, Hudson;
- забезпечення багатопоточного тестування коду, що забезпечує безпеку та швидкодію.

Для того щоб підключити бібліотеку до проекту, у частині <dependencies> файлу pom.xml необхідно прописати наступне:

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.7.0</version>
    <scope>test</scope>
</dependency>
```

Тепер ми можемо додавати різноманітні анотації для гнучкості і зручності управління тестами, найголовнішими з яких є @Test, що безпосередньо помічає клас або метод частиною тестового набору, @Parameters, що зберігає параметри, які можна передати в тестовий метод, і @DataProvider, який дозволяє прогнати один і той же тестовий клас або метод декілька разів з різним набором вхідних даних.

В анотації @Test є декілька допоміжних налаштувань, як от: пріоритет

(priority), позначення включення чи виключення тесту з тестового набору (enabled), очікувані в процесі виконання коду програми виключення (expectedExceptions), кількість повторних запусків обраних тестів (incovationCount), залежність від інших тестових методів (dependsOnMethods) та тестових наборів (dependsOnGroups) тощо. Такі анотації зазвичай позначаються перед тестовим методом, в дужках буде вказано необхідний додатковий параметр. Приклад наведено на рис. 3.1.2.

```
@Test(priority = 0)
public void createNewBoard() throws IOException {
    Random random = new Random();
    int randomInt = random.nextInt( bound: 251 - 1) + 1;
    Board board = client.boardsService.createBoard( name: "New Board #" + randomInt).execute().body();
    this.boardID = board.id;
    System.out.println("A " + board.name + " with " + boardID + " id has been created");
}

@Test(dependsOnMethods = "createNewBoard")
public void getNewBoard() throws IOException {
    Board board = client.boardsService.getBoard(boardID).execute().body();
    Assert.assertEquals(board.id, boardID,  message: "Oops. Something went wrong and Board id we recently created " +
            "IS NOT the same we got");
}
```

Рисунок 3.1.2 – Приклад використання анотації @Test з допоміжними параметрами

Як бачимо з рисунку, для параметру пріоритету (priority) вказується деяке число, причому чим менше це число, тим більшим вважатиметься пріоритет даного тестового методу, числа можуть бути також від’ємними. В даному випадку хочемо, щоб метод `createNewBoard()` запускався на виконання першим, а тому йому єдиному з усіх ставимо пріоритет 0.

Також порядок виконання тестів можна позначити за допомогою параметру залежності одного метода від іншого (dependsOnMethods), в такому випадку в дужках необхідно вказати назву основного методу і пам’ятати, що при запуску тестів основний метод буде завжди йти перед залежним. Тобто, якщо тестовий метод `getNewBoard()` залежить від `createNewBoard()`, то метод

`createNewBoard()` буде запускатися на виконання першим. Також не слід забувати, що оновлення параметрів в анотаціях не відбувається автоматично, тому якщо ми ссилалися на якийсь з методів як основоположний, а потім його перейменували – необхідно зробити це у відповідній анотації також.

3.2 Проектування необхідних моделей для тестування UI з використанням Selenium WebDriver

Для автоматизації тестування користувацького інтерфейсу було створено фреймворк, який включає наступні складові частини: драйвер, клас для роботи з опціями браузера, клас для роботи з елементами сторінки і безпосередньо сторінка сайту, що тестується. Далі поетапно розглядаємо кожен створений клас з деталями реалізації та поясненнями до кожного прийнятого рішення.

3.2.1 Створення класів Driver та Options

Як було зазначено раніше, клас драйверу існує для того, щоб налаштувати роботу з браузером. Для гнучкості налаштування було створено наступні методи: метод налаштування драйверу, метод отримання поточного драйверу без додаткових опцій і метод отримання поточного драйверу з додатковими опціями. Просимо звернути увагу на те, що останні два методи спочатку перевіряють наявність драйверу, якщо на момент перевірки такого немає, – то для початку він автоматично ініціалізується. Готовий клас драйвера виглядає наступним чином:

```
public class Driver {

    private static WebDriver driver;

    private Driver() {};

    public static WebDriver getChromeDriver() {
        if (driver == null) {
            driver = new ChromeDriver();
        }
        return driver;
    }
```

```
        public static WebDriver
getChromeDriver(ChromeOptions options) {
            if (driver == null) {
                driver = new ChromeDriver(options);
            }
            return driver;
        }

        public static WebDriver setupDriver() {
            ChromeOptions options = new Options()
                    .openBrowserFullScreen()

.setPageLoadStrategy(PageLoadStrategy.EAGER)
                    .getOptions();
            return Driver.getChromeDriver(options);
        }
    }
```

Для зручності управління опціями браузера, було створено окремий клас Options, методи якого відповідають за стратегію очікування завантаження сторінки та відкриття вікна в повному розмірі, а також геттер, який повертає встановлені значення. Звичайно, за бажанням можна також додати будь-які опції. Загалом клас опцій виглядає так:

```
    public class Options {

        private ChromeOptions options = new ChromeOptions();

        public Options setPageLoadStrategy(PageLoadStrategy
strategy) {
            options.setPageLoadStrategy(strategy);
            return this;
        }
```

```
        public Options openBrowserFullScreen() {
            options.addArguments("--remote-allow-
origins=*");
            return this;
        }

        public ChromeOptions getOptions() {
            return options;
        }
    }
```

3.2.2 Створення класу Elem

Клас Elem являється надбудовою над класом WebElement і створений для того, щоб мати у власному фреймворку тільки ті методи, які необхідні під цілі та особливості даного проекту/продукту.

Для створення такого класу нам знадобилися поля очікування елемента за замовчуванням, поле драйвера, пошук по локатору By, а також поле CSS селектора. В конструктор ми передаємо саме селектор. В наведеному нижче прикладі продемонстровані поля, конструктор та один з методів створеного класа Elem, а саме клік по елементу:

```
    public class Elem {

        private static final long DEFAULT_TIMEOUT = 30;
        private static WebDriver driver;
        private By by;
        private String cssSelector;

        public Elem(String cssSelector) {
            this.cssSelector = cssSelector;
            this.by = By.cssSelector(cssSelector);
            driver = getChromeDriver();
        }
```

```
    public void click() throws
WebElementNotClickableException {
            try {
                new WebDriverWait(driver,
Duration.ofSeconds(DEFAULT_TIMEOUT))

.until(ExpectedConditions.elementToBeClickable(by));
                driver.findElement(by).click();
            } catch (TimeoutException e) {
                throw new
WebElementNotClickableException(e.getMessage());
            }
        }
    }
```

Повний приклад класу Elem наведено в додатку А.

3.2.3 Створення класу PageObject

В якості PageObject розглянемо головну сторінку Trello – з найбільш помітного, вона має свій хедер з декількома елементами: іконкою з логотипом та випадаючими списками накшталт «Пропозиції», «Рішення», «Плани», «Ціни» та «Ресурси». Клас має включати в собі драйвер та посилання на сторінку, з якою ми будемо працювати. Далі вказуються селектори елементів, які будуть перевірятися. Також є конструктор, що викликає драйвер і метод для відкриття сторінки, додатково можуть бути також інші методи для взаємодії зі сторінкою чи її елементами. Спроектована сторінка в коді виглядає так:

```
    public class TrelloMainPage {

        public static final String PAGE_URL =
"https://trello.com/home";
        private WebDriver driver;
```

```
        public Elem header = new
Elem("[class='BigNavstyles__NavBar-sc-1mttgq7-3 caTbTe']");
        public Elem trelloIcon = new Elem("data-
testid='logo_link'");
        public Elem featuresDropdown = new Elem ("[data-
testid='bignav-tab']:nth-of-type(1)");
        public Elem solutionsDropdown = new Elem ("[data-
testid='bignav-tab']:nth-of-type(2)");
        public Elem plansDropdown = new Elem ("[data-
testid='bignav-tab']:nth-of-type(3)");
        public Elem pricingPage = new Elem ("[data-
testid='bignav-tab']:nth-of-type(4)");
        public Elem resourcesDropdown = new Elem ("[data-
testid='bignav-tab']:nth-of-type(5)");

        public TrelloMainPage() {
            this.driver = getChromeDriver();
        }

        public void openPage() {
            driver.get(PAGE_URL);
        }
    }
```

3.3 Проектування необхідних моделей для тестування API з використанням Retrofit

Для автоматизації тестування API сервісу було створено фреймворк, який включає наступні складові частини: основний клієнт, Interceptor клас, класи моделей та сервісів. Далі поетапно розглядаємо кожен створений клас з деталями реалізації та поясненнями до кожного прийнятого рішення.

3.2.1 Створення основного клієнту та Interceptor класу

Опис налаштування основного клієнту вже був наведений в розділі 2 даної роботи, тому в даному розділі вважаємо доцільним навести вже готову реалізацію, яка виглядає наступним чином:

```
    public class TrelloClient {

        public static final String BASE_URL =
"https://api.trello.com/1/";

        OkHttpClient client = new
OkHttpClient().newBuilder()
                .connectTimeout(20, TimeUnit.SECONDS)
                .readTimeout(20, TimeUnit.SECONDS)
                .writeTimeout(20, TimeUnit.SECONDS)
                .addInterceptor(new AuthInterceptor())
                .build();

        Retrofit retrofit = new Retrofit.Builder()
                .client(client)
                .baseUrl(BASE_URL)

.addConverterFactory(GsonConverterFactory.create())
                .build();

        public BoardsService boardsService =
retrofit.create(BoardsService.class);
        public ListsService listService =
retrofit.create(ListsService.class);
        public CardsService cardsService =
retrofit.create(CardsService.class);
    }
```

Статична фіналізована змінна `BASE_URL` тут використовується задля того, щоб уникнути необхідності прописувати незмінну частину посилання “https://api.trello.com/1/” для кожного запиту, натомість вона буде підставлятися на початок автоматично. Далі було створено okHttp клієнт з базовими налаштуваннями очікувань і передано його в об’єкт класа Retrofit. Останнім, але не менш важливим, пунктом є додавання екземплярів класів сервісів, створених власноруч, приклади яких буде наведено в наступній частині даного розділу.

Іншим основоположним компонентом фреймворку для автоматизації

тестування API є клас Interceptor, який покликаний для того, щоб перехоплювати запити і додавати в них необхідну для тестування інформацію, як-от кукі чи дані про авторизацію поточного користувача системи. Саме останній момент було реалізовано в нашому фреймворку. Пропонуємо розглянути клас перехоплювача більш детально:

```
     public class AuthInterceptor implements Interceptor {

         public static final String API_KEY =
"1ab02543dfa68283436ca16655152b6c";
         public static final String TOKEN =
"e70c45dfef7d144561eb7a5080bc81ddb8776afb96ab36d214f8e5fb94
92f5c2";

         @Override
         public Response intercept(Chain chain) throws
IOException {
             Request original = chain.request();
             HttpUrl originalHttpUrl = original.url();

             HttpUrl url = originalHttpUrl.newBuilder()
                     .addQueryParameter("key", API_KEY)
                     .addQueryParameter("token", TOKEN)
                     .build();

             Request.Builder requestBuilder =
original.newBuilder()
                     .url(url);

             Request request = requestBuilder.build();

             return chain.proceed(request);
         }
     }
```

З наведеного вище лістингу коду видно, що для початку у вигляді статичних фіналізованих змінних задаються параметри авторизації – API-ключ та токен – які згодом передаються до ланцюжку запиту, тим самим змінюючи його, і передаючи програмі далі вже з необхідними нам параметрами.

3.2.2 Створення інтерфейсів сервісів

Сервіси в API фреймворку слугують для того, щоб описати HTTP методи методами мови програмування Java. Для цього на кожний сервіс – умовну логічну одиницю програми – створюється окремий інтерфейс. Зазвичай для проектування сервісів використовують саме інтерфейси, а не класи, з тієї причини, що будь-який клас може реалізовувати скільки завгодно інтерфейсів не будучи з ними логічно пов’язаним, тоді як наслідуватися можна лише від одного класу. Окрім цього, інтерфейс є простішим рішенням для реалізації в класах, які можуть не мати між собою нічого спільного, а от класи (в тому числі абстрактні), на противагу, є сенс використовувати коли йдеться про близький зв’язок типу “is-a” (коли клас-наслідник є продовженням та доповненням батьківського класа).

Для того, щоб розпочати написання сервісу, необхідно ознайомитися з тестовою документацією продукту, тестування якого ми хочемо автоматизувати, і опанувати специфічний, притаманний для бібліотеки, незвичний синтаксис.

Так, наприклад, для роботи з картками – найбільш базовою одиницею інформації в Trello – передбачений наступний список дій: отримати інформацію по картці за ID, створити нову картку, редагувати існуючу картку, додати коментар чи учасника до картки, видалити картку, тощо. Кожен з них має певний HTTP метод, може мати змінні вхідні параметри та тіло запиту.

Розглянемо найпростішу взаємодію з карткою – отримати картку за ID. На офіційному сайті з API документацією бачимо опис HTTP методу з прикладами запитів для різних мов програмування та середовищ, які можна скопіювати, а також прикладами відповідей від серверу. Наочно запит продемонстровано на рис. 3.2.2.1 та 3.2.2.2.

Get a Card

GET /1/cards/{id}

Get a card by its ID

Request

PATH PARAMETERS

id **REQUIRED**
string

The ID of the Card

Pattern: `^[0-9a-fA-F]{24}$`

Рисунок 3.2.2.1 – Приклад запиту для отримання сутності картки за ID

Example

cURL Node.js Java Python PHP

```
curl --request GET \
  --url 'https://api.trello.com/1/cards/{id}?key=APIKey&token=APIToken' \
  --header 'Accept: application/json'
```

Responses

200

Success

Content type	Value
application/json	Card

Рисунок 3.2.2.2 – Приклад демонстраційних запитів різними мовами програмування та успішної відповіді від сервера

В коді даний запит матиме наступний вигляд:

```
    @GET("cards/{idCard}")
    Call<Card> getCard(@Path("idCard") String idCard);
```

Тут, за допомогою анотації @GET позначається тип HTTP методу, в дужках вказано URL за винятком основної частини BASE_URL, яка була винесена в клієнт для зручності. Цікавим моментом є змінна {idCard}, винесена у фігурних дужках, яка буде слугувати вхідним параметром для методу, тому в самому тілі запиту ми маємо явно вказати що це динамічна складова за допомогою анотації @Path і задання типу змінної – текстового рядку.

В нашому виконанні інтерфейс CardsService має наступний вигляд:

```
    public interface CardsService {

        @GET("cards/{idCard}")
        Call<Card> getCard(@Path("idCard") String idCard);

        @POST("cards")
        Call<Card> postCard(@Query("idList") String idList,
@Body Card card);

        @PUT("cards/{idCard}")
        Call<Card> editCard(@Path("idCard") String idCard,
@Body Card card);

        @DELETE("cards/{idCard}")
        Call<Card>    deleteCard(@Path("idCard")    String
idCard);
    }
```

Також, в ході виконання даної роботи було створено наступні сервіси: BoardsService (сервіс для роботи з дошками, на яких безпосередньо відображаються картки, а також їх поточний статус) та ListsService (сервіс для

роботи з колекцією карток). Їх наведено окремо в додатку Б.

3.2.3 Створення класів моделей

Моделі у фреймворку слугують нам для того, щоб використовувати екземпляри змодельованих класів для передачі та отримання необхідної інформації про властивості певних окремих компонентів програми. Дані про такі властивості знову беремо в офіційній API документації.

Розглянемо відому нам вже сутність Cards, кожній картці притаманні: свій унікальний ідентифікаційний номер (ID), опис, назва, статус закриття, інформація про теги та учасників, що працювали над даною карткою, дані про час останнього внесення змін, ідентифікаційні номера борду та списку, до якого відноситься дана картка, тощо. Це можна побачити на рис. 3.2.3.

Card

Details Example

id
string

Pattern: `^[0-9a-fA-F]{24}$`

address
string

Nullable: `true`

badges
object

checkItemStates
Array<string>

closed
boolean

coordinates
string

Nullable: `true`

creationMethod
string

Nullable: `true`

Рисунок 3.2.3 – Документація по сутності Card та її властивостям

В кодовому представленні це буде виглядати як звичайний клас з полями в якості властивостей, конструктором (за замовчуванням або спроектованим відповідно до наших потреб) та, опціонально, перевизначеним методом `toString()`, в якому можна вказати цікавлячу нас інформацію про сутність у більш легкодоступному вигляді задля полегшення виявлення потенційних помилок чи відображення поточного статусу сутності.

Наприклад:

```
    public class Card {

        public String id;
        public String desc;
        public String name;
        public boolean closed;
        public List<String> idLabels;
        public List<String> idMembers;
        public String dateLastActivity;
        public int pos;
        public String cardRole;
        public String idBoard;
        public String idList;

        public Card(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return name + "Card id = " + id + "\nCard
description: " + desc;
        }
    }
```

Інші моделі, створені в процесі розробки власного фреймворку для автоматизації тестування API, наведені в додатку В.

Висновки до розділу 3

У даному розділі було розглянуто процес розробки та імплементації фреймворку для автоматизації тестування вебсайту відповідно до визначених раніше цілей та вимог, серед яких ключовими були здатність до автоматизації взаємодії з веб-елементами, підтримка різних типів браузерів та операційних систем, а також легкість розширення та підтримки. З метою досягнення цих цілей, була обрана мова програмування Java, і використані такі сучасні інструменти, як Selenium WebDriver та Retrofit при дотриманні сучасних патернів проектування та кращих підходів у програмуванні.

У результаті розробки було створено наступні модулі:

- Driver, Elem, Options та PageObject клас для автоматизації тестування користувацького інтерфейсу;
- Client, AuthInterceptor, класи моделей та сервісів для автоматизації тестування API частини.

Головною метою створення даного фреймворку було полегшити та прискорити процес тестування, а також забезпечити стабільність та надійність web-сайту. Безпосередня перевірка працездатності створеного фреймворку буде проводитись в наступному розділі даної роботи.

4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ФРЕЙМВОРКУ

Провести тестування розробленого фреймворку було вирішено на основі сервісу Trello – системи управління проектами у вигляді дошки з завданнями, на якій візуалізовано поточні задачі, інформацію по ним, а також актуальні статуси. Таке рішення було прийнято з двох причин: популярність використання сервісу на реальних проектах, а також те, що API документація програми наведена у вільному доступі і є досить легкою для розуміння і сприйняття новачком.

Працездатність готового фреймворку перевіряється шляхом створення автоматизованих тестів. Як було зазначено в другому розділі даної роботи, автоматизовані тести пишуться на основі документації – це можуть бути чек-лісти, тест-кейси або юзерські (користувацькі) сценарії. Для перевірки нашого фреймворку було вирішено спроектувати декілька демонстраційних тестових кейсів для перевірки як функціоналу фронтенду, так і бекенду, а також порівняти швидкість тестування вручну та за допомогою створених автоматизованих тестів для визначення ефективності розробленого фреймворку.

Для тестування користувацького інтерфейсу можна використати наступний тест-кейс:

№	Крок	Очікуваний результат
1	Відкрити головну сторінку https://trello.com/home.	Головна сторінка успішно відкрита.
2	Впевнитися, що хедер присутній і вміщує такі елементи: логотип, випадаючі списки «Пропозиції», «Рішення», «Плани», титул «Ціни» та випадаючий список «Ресурси».	Хедер присутній на сторінці і містить зазначені елементи.

3	Клікнути на «Ціни».	Відкрилась сторінка з цінами за посиланням https://trello.com/pricing.

Кодове виконання такого тест кейсу виглядає наступним чином:

```
    public class TrelloUITest extends Setup {

        TrelloMainPage mainPage = new TrelloMainPage();

        @Test()
        public void assertAllElementsArePresent() {
            mainPage.openPage();
            mainPage.header.isPresent();
            mainPage.trelloIcon.isPresent();
            mainPage.featuresDropdown.isPresent();
            mainPage.solutionsDropdown.isPresent();
            mainPage.plansDropdown.isPresent();
            mainPage.pricingPage.isPresent();
            mainPage.resourcesDropdown.isPresent();
        }

        @Test()
        public void checkPricingPage() throws
WebElementNotClickableException {
            mainPage.pricingPage.isPresent();
            mainPage.pricingUrl.click();

Assert.assertEquals(Driver.getChromeDriver().getCurrentUrl(
), "https://trello.com/pricing", "URLs are not matching!");

        }
    }
```

Після запуску успішний результат виглядає так:

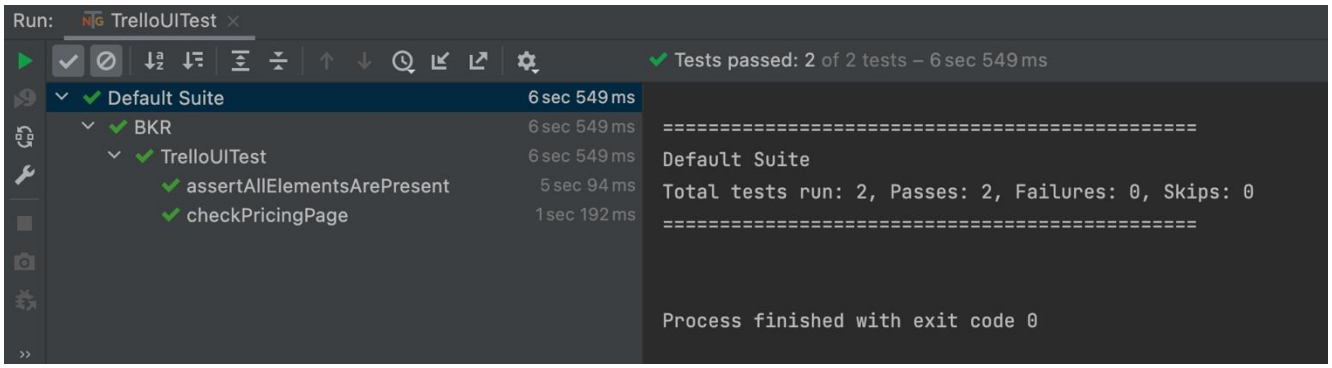

Рисунок 4.1 – Приклад успішних UI тестів

Перевіримо функціональність розробленого фреймворку на предмет того, чи не надають автоматизовані тести помилково позитивного результату. Для цього навмисно зламаємо селектор для елементу `pricingUrl` і поставимо тест на запуск ще раз. Результат можна подивитися на рис. 4.2.

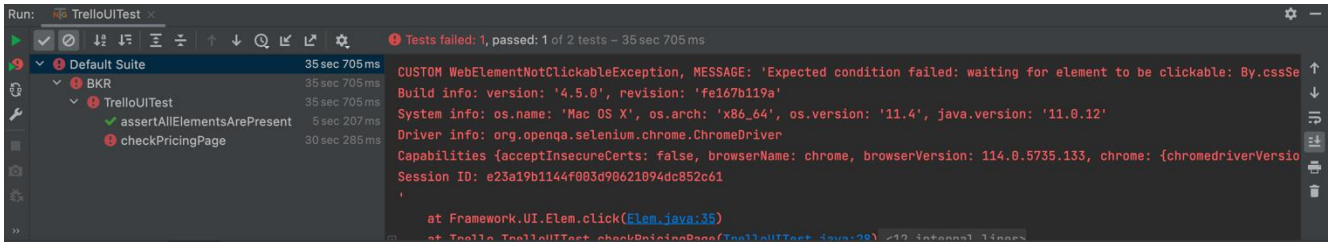

Рисунок 4.2 – Приклад неуспішного failed UI тесту

Далі до API частини. Автоматизуємо наступний юзерський сценарій: створюємо нову дошку та перевіряємо її наявність – отримуємо список карток в дошці – створюємо нову картку – редагуємо картку та перевіряємо чи зміни були збережені – видаляємо картку – видаляємо дошку. Лістинг коду наведено в додатку Г. Результат прогону автоматизованих тестів можна переглянути на рис. 4.3.

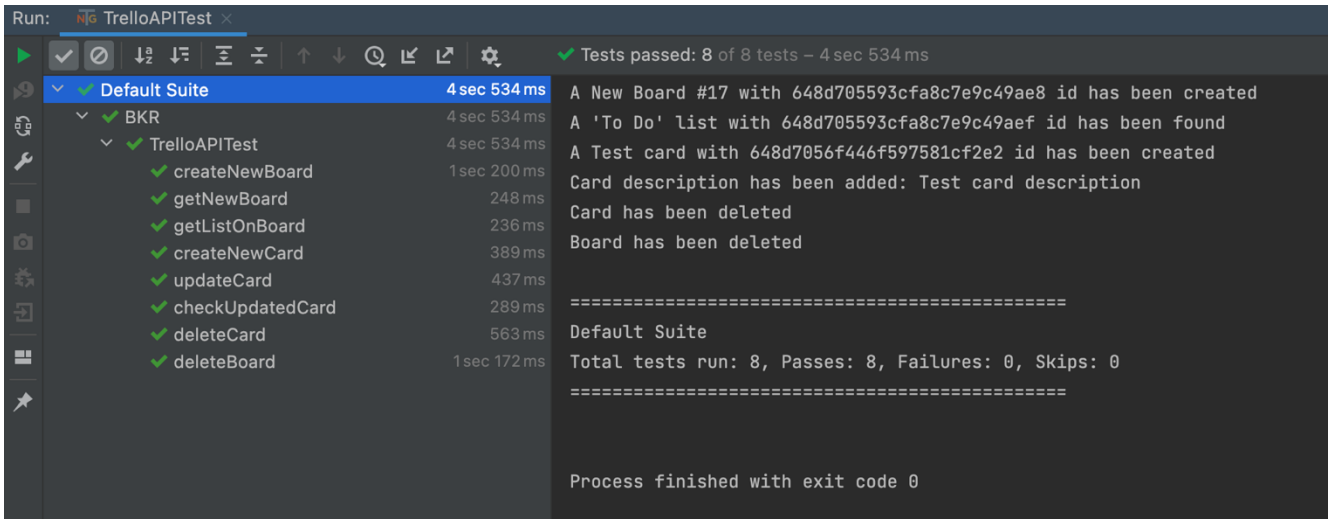

Рисунок 4.3 – Приклад успішного API тесту

Цікаво відмітити, що при ручній перевірці даного користувацького сценарію, в тестувальника йде в середньому 6-8 хвилин на те, щоб здійснити всі дії та перевірити необхідні id сутностей та їх взаємозв’язки, не говорячи вже про налаштування тестових даних у вигляді запитів в сторонній програмі по типу Postman чи роботу з консоллю розробника в браузері як альтернативу. Автоматизований же тест справляється з такою перевіркою за 4,5 секунди.

Висновки до розділу 4

В даному розділі було протестовано створений фреймворк шляхом написання автоматизованих тестів для базових користувацьких сценаріїв: взаємодії з початковою сторінкою (перегляд хедера та перехід за посиланнями в ньому) на стороні користувацького інтерфейсу, а також перевірці функціоналу створення, редагування та видалення карток в межах борду (дошки) на стороні API сервісу.

В результаті проведеної апробації було встановлено, що автоматизація тестування наведених сценаріїв дійсно прискорює процес перевірки якості функціоналу продукту, особливо якщо мова йде про регресійне тестування.

Звичайно, завжди є можливості для покращення та розширення фреймворку за рахунок додавання нових методів взаємодії з елементами чи перевірок для HTTP методів. Але в цілому фреймворк показав себе як очікувалось.

ВИСНОВКИ

У ході виконання даної роботи було проведено аналіз сучасних підходів та методик тестування web-продуктів, в результаті чого було виявлено, що автоматизація тестування є необхідним етапом в процесі розробки програмного забезпечення, особливо в сфері web-розробки. Це дозволяє прискорити процес тестування, забезпечити високу якість продукту та знизити витрати на ручне тестування.

Метою даної роботи було розробити фреймворк, який би спрощував та автоматизував процес тестування web-продуктів. Для досягнення цієї мети було проведено аналіз різних інструментів та технологій, а також визначено вимоги до фреймворку.

Результатом роботи є розроблений фреймворк для тестування як користувацького інтерфейсу, так і API сервісу, який має наступні ключові характеристики:

- взаємодія з різними web-елементами: фреймворк надає можливість автоматизувати взаємодію з різними елементами web-сторінок, такими як кнопки, форми, посилання тощо;
- розширення та модульність: фреймворк побудований з урахуванням принципів розширення та модульності, що дозволяє легко додавати нові функціональні можливості та адаптувати його під конкретні потреби проекту;
- підтримка різних браузерів та операційних систем: фреймворк забезпечує можливість виконання тестів на різних платформах та браузерах, забезпечуючи крос-платформенну сумісність;
- легкість використання: фреймворк має зрозумілий та зручний інтерфейс, що дозволяє швидко навчитися його використовувати за ефективно застосовувати.

Під час розробки та тестування фреймворку було проведено ряд експериментів та перевірок, які підтвердили його працездатність та ефективність.

Отримані результати свідчать про успішне досягнення мети кваліфікаційної роботи та реалізацію фреймворку для автоматизації тестування web-продуктів.

У подальшому роботу над фреймворком можна продовжити, додавши нові функціональні можливості, покращивши інтерфейс та розширивши підтримку платформ та браузерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Автоматизоване тестування. База знань. *QaLight: Центр підготовки IT фахівців*: вебсайт. URL: https://qalight.ua/baza-znaniy/avtomatizovane-testuvannya/.
2. IEEE. Guide to the Software Engineering Body of Knowledge. SWEBOK, 2004. 335 p.
3. Ручне та автоматизоване тестування. База знань. *QaLight: Центр підготовки IT фахівців*: вебсайт. URL: https://qalight.ua/baza-znaniy/ruchne-ta-avtomatizovane-testuvannya.
4. International Software Testing Qualifications Board. ISTQB Certification – Foundation Level syllabus, ISTQB. 2023. 74 p.
5. Ляшенко Дар’я. Автоматизоване тестування UI: стисло про головне. 2021. *Codeguida*: вебсайт. URL: https://codeguida.com/post/2881.
6. Тестування UI (інтерфейсу користувача). Блог. 2021. *Wezom*: вебсайт. URL: https://wezom.com.ua/ua/blog/testing-ui-user-interface.
7. Що таке тестування програмного забезпечення інтерфейсу користувача? Глибоке занурення в типи, процеси, інструменти та реалізацію. *Zaptest. Unlimited Software Automation*: вебсайт. URL: https://www.zaptest.com/uk/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7#%D0%9A%D0%BE%D0%BB%D0%B8_%D1%82%D0%B0_%D0%BD%D0%B0%D0%B2%D1%96%D1%89%D0%BE_%D0%B2%D0%B0%D0%BC_%D0%BF%D0%BE%D1%82%D1%80%D1%96%D0%B1%D0%BD%D1%96_%D1%82%D0%B5%D1%81%D1%82%D0%B8_%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%

B9%D1%81%D1%83_%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0.

8. Record and Playback Testing: The Easiest Way To Start Automating Tests. 2023. *Software Testing Help*: вебсайт. URL: https://www.softwaretestinghelp.com/record-and-playback-testing/#What_is_Record_and_Playback_in_Testing.
9. Model Based Testing Using Generic Algorithm. 2023. *Software Testing Help*: вебсайт. URL: https://www.softwaretestinghelp.com/model-based-testing/.
10. Що таке тестування API? Глибоке занурення в автоматизацію тестування API, процеси, підходи, інструменти, фреймворки та багато іншого! *Zaptest. Unlimited Software Automation*: вебсайт. URL: https://www.zaptest.com/uk/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-api-%D0%B3%D0%BB%D0%B8%D0%B1%D0%BE%D0%BA%D0%B5-%D0%B7%D0%B0%D0%BD%D1%83%D1%80%D0%B5%D0%BD%D0%BD.
11. WebDriver. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/.
12. Що таке Selenium WebDriver? *QualityAssuranceGroup*: вебсайт. URL: https://qagroup.com.ua/publications/shcho-take-selenium-webdriver/.
13. Browser Options. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/drivers/options/.
14. Browser Navigation. Documentation. *Selenium:* вебсайт. URL: https://www.selenium.dev/documentation/webdriver/interactions/navigation/.
15. Working with windows and tabs. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/interactions/windows/.
16. Browser Interactions. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/interactions/.
17. Selenium WebDriver як інструмент для автоматизованого тестування. 2021.

QATestLab: training center: вебсайт. URL: https://training.qatestlab.com/blog/technical-articles/selenium-webdriver/.

18. Locator strategies. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/elements/locators/.
19. XPath vs CSS Selector. *QualityAssuranceGroup*: вебсайт. URL: https://qagroup.com.ua/publications/xpath-vs-css-selector/.
20. Java Program to Demonstrate the Lazy Initialization Thread-Safe. *GeeksForGeeks*: вебсайт. URL: https://www.geeksforgeeks.org/java-program-to-demonstrate-the-lazy-initialization-thread-safe/.
21. Working with cookies. Documentation. *Selenium*: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/interactions/cookies/.
22. Mouse actions. Documentation. Selenium: вебсайт. URL: https://www.selenium.dev/documentation/webdriver/actions_api/mouse/#drag-and-drop-on-element/.
23. Leonard Richardson. RESTful Web APIs: Services for a Changing World. 1st Edition. O'Reilly Media Inc., 2013. 406 p.
24. HTTP: The Definitive Guide. 1st Edition. / David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy. O'Reilly Media Inc., 2002. 656 p.
25. Douglas Crockford. JSON: The Fat-Free Alternative to XML. XML conference. Boston, 2006.
26. Retrofit. A type-safe HTTP Client for Android and Java. *GitHub*: website. URL: https://square.github.io/retrofit/.
27. Олександр Швець. Занурення в патерни проектування. 2020. 393 с.
28. Martin Fowler. PageObject. 2013. *Martin Fowler*: вебсайт. URL: https://martinfowler.com/bliki/PageObject.html.
29. Навіщо і як QA писати тестову документацію. Структуруємо та робимо її зрозумілою. *DOU*: вебсайт. URL: https://dou.ua/lenta/columns/creating-user-friendly-qa-documentation/.
30. Тестування. Основи. Test case. *QALight*: вебсайт. URL: https://qalight.ua/baza-

znaniy/test-case-2/.

31. 12 характеристик високоефективних тест-кейсів. *Software-testing*: вебсайт. URL: https://software-testing.ru/library/testing/test-analysis/3495-12-traits-of-highly-effective-tests.
32. Cedric Beust, Hani Suleiman. Next Generation Java Testing: TestNG and Advanced Concepts. 1st Edition. Addison-Wesley Professional, 2007. 512 p.

ДОДАТОК А Лістинг коду класу Elem

```
public class Elem {

    private static final long DEFAULT_TIMEOUT = 30;
    private static WebDriver driver;
    private By by;
    private String cssSelector;

    public Elem(String cssSelector) {
        this.cssSelector = cssSelector;
        this.by = By.cssSelector(cssSelector);
        driver = getChromeDriver();
    }

    public void click() throws WebElementNotClickableException {
        try {
            new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                    .until(ExpectedConditions.elementToBeClickable(by));
            driver.findElement(by).click();
        } catch (TimeoutException e) {
            throw new WebElementNotClickableException(e.getMessage());
        }
    }

    public String getText() throws WebElementNotFoundException {
        try {
            new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                    .until(ExpectedConditions.presenceOfElementLocated(by));
            return driver.findElement(by).getText();
        } catch (Exception e) {
            throw new WebElementNotFoundException(e.getMessage());
        }
    }

    public String getCssSelector() {
        return cssSelector;
    }

    public String getAttribute(String attribute) throws WebElementNotFoundException {
        try {
```

```
            new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                    .until(ExpectedConditions.presenceOfElementLocated(by));
            return driver.findElement(by).getAttribute(attribute);
        } catch (Exception e) {
            throw new WebElementNotFoundException(e.getMessage());
        }
    }

    public List<Elem> findElements() {
        List<WebElement> list = driver.findElements(by);
        int count = list.size();
        List<Elem> result = new ArrayList<>(count);
        for (int i = 1; i <= count; i++) {
            String selector = getCssSelector() + ":nth-child(" + i + ")";
            Elem elem = new Elem(selector);
            result.add(elem);
        }
        return result;
    }

    public void switchToFrame() throws WebElementNotFoundException {
        try {
            new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                    .until(ExpectedConditions.presenceOfElementLocated(by));
            driver.switchTo().frame(driver.findElement(by));
        } catch (Exception e) {
            throw new WebElementNotFoundException(e.getMessage());
        }
    }

    public void checkAbsence() {
        new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                .until(ExpectedConditions.invisibilityOfElementLocated(by));
    }

    public boolean isPresent() {
        try {
            new WebDriverWait(driver, Duration.ofSeconds(2))
                    .until(ExpectedConditions.presenceOfElementLocated(by));
            return true;
        } catch (TimeoutException e) {
            return false;
```

```
        }
    }

    public void sendKeys(String key) {
        new WebDriverWait(driver, Duration.ofSeconds(DEFAULT_TIMEOUT))
                .until(ExpectedConditions.presenceOfElementLocated(by));
        driver.findElement(by).sendKeys(key);
    }

}
```

ДОДАТОК Б Лістинг коду інтерфейсів BoardsService та ListsService

```
public interface BoardsService {

    @POST("boards")
    Call<Board> createBoard(@Query("name") String name);

    @GET("boards/{id}")
    Call<Board> getBoard(@Path("id") String id);

    @DELETE("boards/{idBoard}")
    Call<Board> deleteBoard(@Path("idBoard") String idBoard);

}



public interface ListsService {
    @GET("boards/{idBoard}/lists")
    Call<List<Lists>> getList(@Path("idBoard") String idBoard);

}
```

ДОДАТОК В Лістинг коду моделей Board та List

```
public class Board {

    public String id;
    public String name;

    @Override
    public String toString() {
        return "Board name: "+ name +"\nBoard id = " + id;
    }

}



public class Lists {

    public String id;
    public String name;
    public Boolean closed;
    public String idBoard;

    @Override
    public String toString() {
        return "List name: " + name + "\nList id = " + id + "\nClosed = " + closed;
    }

}
```

ДОДАТОК Г Лістинг коду автоматизованого тесту API частини

```
public class TrelloAPITest {

    TrelloClient client = new TrelloClient();
    String boardID;
    String listID;
    Card card;
    String cardID;
    String cardDesc;

    @Test(priority = 0)
    public void createNewBoard() throws IOException {
        Random random = new Random();
        int randomInt = random.nextInt(251 - 1) + 1;
        Board board = client.boardsService.createBoard("New Board #" +
randomInt).execute().body();
        this.boardID = board.id;
        System.out.println("A " + board.name + " with " + boardID + " id has been
created");
    }

    @Test(dependsOnMethods = "createNewBoard")
    public void getNewBoard() throws IOException {
        Board board = client.boardsService.getBoard(boardID).execute().body();
        Assert.assertEquals(board.id, boardID, "Oops. Something went wrong and
Board id we recently created " +
                "IS NOT the same we got");
    }

    @Test(dependsOnMethods = "getNewBoard")
    public void getListOnBoard() throws IOException {
        List<Lists> lists = client.listService.getList(boardID).execute().body();
        for (Lists list : lists) {
            if (list.name.equals("To Do")) {
                this.listID = list.id;
            }
        }
        System.out.println("A 'To Do' list with " + listID + " id has been found");
    }

    @Test(dependsOnMethods = "getListOnBoard")
    public void createNewCard() throws IOException {
        this.card = new Card("Test card");
```

```
        card = client.cardsService.postCard(listID, card).execute().body();
        this.cardID = card.id;
        System.out.println("A " + card.name + " with " + cardID + " id has been created");
    }

    @Test(dependsOnMethods = "createNewCard")
    public void updateCard() throws IOException {
        card.desc = "Test card description";
        this.cardDesc = card.desc;
        client.cardsService.editCard(cardID, card).execute().body();
        System.out.println("Card description has been added: " + cardDesc);
    }

    @Test(dependsOnMethods = "updateCard")
    public void checkUpdatedCard() throws IOException {
        card = client.cardsService.getCard(cardID).execute().body();
        Assert.assertEquals(cardDesc, card.desc, "Oops. Something went wrong and Card info IS NOT the same");
    }

    @Test(dependsOnMethods = "checkUpdatedCard")
    public void deleteCard() throws IOException {
        client.cardsService.deleteCard(cardID).execute().body();
        System.out.println("Card has been deleted");
    }

    @Test(dependsOnMethods = "deleteCard")
    public void deleteBoard() throws IOException {
        client.boardsService.deleteBoard(boardID).execute().body();
        System.out.println("Board has been deleted");

    }
}
```