МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

БЄГЛОВА АЛЬБІНА ВАДИМІВНА

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2025 р.

ОПТИМІЗАЦІЯ СЦЕНАРІЇВ ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота

Науковий керівник:
Наталія ВЕСЕЛОВСЬКА,
професор кафедри інформаційних технологій
д-р техн. наук , професор

_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

АНОТАЦІЯ

Бєглова А.В Оптимізація сценаріїв для автоматизованого тестування. Спеціальність 122 «Комп'ютерні науки». Освітня програма «Комп'ютерні технології обробки даних» . Донецький національний університет імені Василя Стуса, Вінниця, 2025.

Основною метою виконання кваліфікаційної роботи є дослідження, розробка та впровадження методів оптимізації автоматизованих тестових сценаріїв для підвищення ефективності та надійності тестування користувацьких інтерфейсів веб-додатків.

У вступі розглядається актуальність теми, мета і завдання роботи, об'єкт, предмет та методи дослідження.
У першому розділі наведено теоретичні відомості щодо основ автоматизованого тестування UI, розглянуто ключові поняття, підходи та особливості організації процесу тестування.

У другому розділі досліджено архітектурні підходи до побудови тестових сценаріїв, методи їх оптимізації та принципи автоматизованого управління тестуванням.

Третій розділ присвячено практичній реалізації тестових сценаріїв мовою TypeScript із використанням фреймворку Playwright, застосуванню оптимізаційних підходів та порівняльному аналізу ефективності тестів до та після оптимізації.

Ключові слова: QA, CI/CD, Automation, Test Framework

ABSTRACT

Beglova A.V. Optimization of scripts for automated testing. Specialty 122 "Computer Science". Educational program "Computer Data Processing Technologies". Vasyl Stus Donetsk National University, Vinnytsia, 2025.

The main purpose of the qualification work is to research, develop and implement methods for optimizing automated test scripts to increase the efficiency and reliability of testing user interfaces of web applications.

The introduction considers the relevance of the topic, the goal and objectives of the work, the object, subject and methods of research.

The first section provides theoretical information on the basics of automated UI testing, considers key concepts, approaches and features of the testing process organization.

The second section examines architectural approaches to building test scripts, methods for their optimization and principles of automated testing management.

The third section is devoted to the practical implementation of test scripts in TypeScript using the Playwright framework, the application of optimization approaches, and a comparative analysis of the effectiveness of tests before and after optimization.

Keywords: QA, CI/CD, Automation, Test Framework

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

Скорочення	Повна назва	Пояснення
QA [1]	Quality Assurance	Інша назва для галузі тестування ПЗ
UI [2]	User Interface	Користувацький інтерфейс продукту
UX [3]	User experience	Досвід кінцевого користувача від користування продуктом
E2E [4]	End-to-end	Вид розширеного тестування («Від кінця до кінця»)
CI/CD [5]	Continuous Integration/Continuous Delivery	Набір практик розробки програмного забезпечення, які автоматизують процеси створення, тестування та розгортання коду
AJAX [6]	Asynchronous JavaScript and XML	Запити, що дозволяють оновлювати частину веб-сторінки без повного перезавантаження
SaaS [7]	Software as service	Програмні продукти, що працюють повністю в браузері

ВСТУП

У сучасному світі сайти та веб-додатки стали не просто інструментами для виконання завдань чи ведення бізнесу — вони глибоко інтегрувалися у щоденне життя, впливаючи на спосіб, у який люди спілкуються, працюють, навчаються й приймають рішення. Користувацький інтерфейс (UI) є першою точкою дотику між людиною та програмним продуктом, тому саме від його якості залежать не лише зручність взаємодії, а й загальне враження про продукт, лояльність користувачів та конкурентоспроможність на ринку.

В умовах стрімкого розвитку цифрових технологій, коли оновлення веб-додатків виходять з високою частотою, забезпечення стабільності, доступності та коректності UI стає критично важливим завданням, яке неможливо вирішити без якісного тестування. Автоматизоване тестування UI стало справжньою революцією у сфері забезпечення якості програмного забезпечення. Воно дозволяє швидко та ефективно перевіряти функціонал, мінімізуючи вплив людського фактору, підвищуючи повторюваність і точність тестів. З його допомогою можна оперативно реагувати на зміни в коді, виявляти й виправляти помилки ще на ранніх етапах розробки. Однак зі зростанням веб-додатку кількість тестових сценаріїв невпинно збільшується, що породжує нові виклики: як грамотно організувати тести, уникнути дублювання, зберегти їх актуальність та керованість, не перевантажуючи команду та не витрачаючи зайвих ресурсів? До цього додаються проблеми складності структури, нестабільності тестів через часті зміни UI, а також зростання витрат на їх підтримку.

У своїй роботі я прагнула глибше зрозуміти, як можна оптимізувати автоматизовані сценарії для тестування UI сучасних веб-додатків, щоб вони не лише відповідали поточним вимогам, але й легко масштабувалися разом із проєктом. Були досліджені передові інструменти автоматизації, такі як Selenium, Cypress, Playwright та інші, порівняно підходи до організації тестових наборів,

впровадження шаблонів проектування (наприклад, Page Object Model), використання параметризації та повторного використання компонентів. Особливу увагу приділено інтеграції тестів у CI/CD-процеси, що дозволяє забезпечити безперервний контроль якості та оперативний зворотний зв’язок для розробників. У ході аналізу було виявлено типові проблеми, що виникають при масштабуванні UI-тестування: нестабільність тестів через динамічні зміни інтерфейсу, дублювання коду при створенні нових сценаріїв, ускладнення підтримки через багаторівневу структуру тестів.

Головна мета даної роботи — розробити підхід, який дозволить скоротити час виконання тестів, підвищити їх стабільність і полегшити підтримку навіть у динамічних умовах розвитку веб-додатку. В результаті компанії отримують можливість швидко реагувати на зміни, підвищувати якість кінцевого продукту та зменшувати витрати на тестування. Підсумовуючи, оптимізація автоматизованих UI-тестів є ключовим фактором у забезпеченні високої якості сучасних веб-додатків. Запропоновані підходи та інструменти дозволяють не лише підвищити надійність програмного забезпечення, а й створити ефективну, гнучку й масштабовану систему тестування, яка відповідає викликам сучасної розробки. Це сприяє не лише зростанню конкурентоспроможності продукту, а й підвищенню задоволеності користувачів, що є головною цінністю продукту цифрової епохи.

РОЗДІЛ 1
ОГЛЯД ТЕОРЕТИЧНИХ ВІДОМОСТЕЙ

1.1.Основні поняття та визначення

Веб-додатки виконують провідну роль у бізнес-процесах, електронній комерції, банківських та медичних системах, соціальних мережах тощо. З одного боку, це спричиняє суттєве розширення функціональності таких систем (наприклад — інтерактивні дашборди, адаптивні інтерфейси, багатомодальні взаємодії); з іншого — створює зростаючу складність: асинхронні операції, динамічна зміна станів, широке різноманіття браузерів, пристроїв і середовищ виконання. Наприклад, нещодавнє дослідження «A Survey on Web Application Testing: A Decade of Evolution» підкреслює, що веб-тестування за останнє десятиліття суттєво еволюціонувало саме через ці виклики [1].

Користувацький інтерфейс (UI) виступає «мостом» між користувачем і програмним забезпеченням. Його якість впливає як на задоволення користувача, так і на бізнес-результат: зручний, швидкий, інтуїтивний UI підвищує ймовірність утримання користувачів, зниження кількості помилок і звернень до служби підтримки. У циклі життєвого створення ПЗ (аналіз → проєктування → розробка → тестування → впровадження → супровід) забезпечення якості інтерфейсу становить один із важливих компонентів контролю якості (QA)[8]. Водночас, коли веб-додаток оновлюється часто (наприклад, через Agile/DevOps-процеси), UI-тести мають забезпечувати не лише функціональність, а й стабільність, повторюваність і швидкість. Загалом будь-яке програмне забезпечення, яке розробляється з урахуванням сучасних стандартів якості, повинно обов’язково проходити через усі передбачені етапи тестування. Ці етапи, у свою чергу, визначаються не лише поточними технічними вимогами, а й враховують позиції та підходи, сформульовані продуктовими власниками (product owners), управлінським персоналом компанії-розробника, а також фахівцями, які займаються розробкою тестів — тобто тест-дизайнерами. Тільки після проходження всіх цих визначених етапів перевірки

програмний продукт може бути умовно визнаний таким, що задовольняє основні критерії якості, або таким, що може вважатися прийнятним для подальшого використання кінцевими користувачами у відповідному середовищі функціонування.

Серед численних методів і підходів до тестування, які використовуються в процесі перевірки програмного забезпечення, можна виділити два ключові та найпоширеніші підходи[9], які вважаються базовими в рамках загальної теорії тестування. Мова йде про так зване тестування за принципом **«білої скриньки» (white-box testing)** та **«чорної скриньки» (black-box testing)**. Обидва ці методи мають принципово різні цілі, підходи до реалізації, а також фокус уваги в процесі проведення тестів.

У випадку використання методології, що відповідає принципам **тестування «білої скриньки»**, об'єктом безпосередньої перевірки стає не зовнішня поведінка програмного забезпечення, яка може спостерігатися користувачем під час взаємодії з системою, а саме **внутрішня логіка роботи програми[10]**, її структурні компоненти, архітектурні особливості та внутрішні процеси обробки даних. Тестування за таким принципом передбачає всебічну перевірку коректності побудови всіх елементів програмного коду, а також адекватність і правильність взаємодії між цими елементами. Основна увага приділяється аналізу **керуючих зв'язків** між компонентами програмного продукту, а в деяких випадках також здійснюється перевірка **інформаційних зв’язків**, які забезпечують передачу та збереження даних у межах системи.

Ключовою характеристикою тестування типу «білої скриньки» є **ступінь покриття тестами логіки та вихідного коду програми**. Цей підхід дозволяє встановити, наскільки тестові сценарії охоплюють усі можливі варіанти логічного виконання коду. Іншими словами, програма вважається перевіреною повністю лише тоді, коли досягнуто **вичерпного тестування усіх можливих маршрутів у графі управління** програмною логікою, що дозволяє максимально знизити

ймовірність залишкових помилок, які можуть виникати через недосяжні або недостатньо протестовані ділянки коду.

З іншого боку, метод **тестування «чорної скриньки»** принципово відрізняється за підходом. У цьому випадку об'єктом тестування є **зовнішній інтерфейс** програмного забезпечення[11], тобто те, що є доступним користувачу та іншим системам при взаємодії із програмним продуктом. Тестування за цим методом не вимагає знання внутрішньої структури або логіки програми. Замість цього, увага зосереджується на тому, як програмне забезпечення реалізує свої функціональні можливості відповідно до вимог, що були визначені на етапі проектування.

У межах даного підходу здійснюється перевірка таких важливих аспектів:

- правильність виконання визначених функцій системи відповідно до технічного завдання або специфікації;
- коректність обробки вхідних даних, що вводяться користувачем або іншими програмами;
- точність формування вихідних результатів та їх відповідність очікуваним значенням;
- забезпечення **цілісності зовнішньої інформації**, яка передається, зберігається або виводиться системою в процесі її експлуатації.

Завдяки застосуванню методу «чорної скриньки» тестувальники мають змогу сформувати **різноманітні комбінації вхідних даних**, які дозволяють здійснити комплексну перевірку всіх передбачених функціональних вимог до програмного забезпечення. Це сприяє виявленню потенційних помилок, які могли бути пропущені під час розробки, особливо в контексті обробки нестандартних або граничних значень даних.

Слід також зазначити, що процес тестування програмного забезпечення може реалізовуватися **на різних рівнях деталізації та у різних контекстах**

функціонування системи[12]. Так, залежно від цілей тестування, обсягу перевірки та доступності системних компонентів, програмне забезпечення може тестуватися:

- як **єдиний цілісний продукт**, що функціонує у середовищі, наближеному до реального (тобто в умовах, максимально подібних до тих, у яких продукт буде використовуватись кінцевими користувачами);
- на рівні **окремих компонентів або модулів**, які перевіряються ізольовано для виявлення внутрішніх помилок та дефектів;
- у вигляді **одиничних функціональних блоків**, де кожна функція чи процедура перевіряється окремо;
- безпосередньо у складі **живої, тобто реально працюючої системи**, що дозволяє оцінити ефективність програмного забезпечення в умовах повної інтеграції з іншими системами та підсистемами.

Зазначені вище методи, як і сам процес їх реалізації, традиційно реалізуються у ручному режимі, тобто за участі безпосередньо тестувальника, який виконує дії, аналізує результати й фіксує знайдені дефекти. **Мануальне тестування** залишається критично важливою складовою загального процесу забезпечення якості, особливо на ранніх етапах розробки, коли необхідна гнучкість, адаптивність та здатність до швидкої переоцінки умов тестування. Проте з розвитком сучасних підходів до розробки програмного забезпечення, особливо в умовах масштабних проєктів, швидких релізів і вимог до високої повторюваності перевірок, **на перший план усе частіше виходить автоматизоване тестування**, як ефективний засіб підвищення продуктивності та надійності процесу тестування.

Автоматизоване тестування (або скорочено — **автотестування**) дає змогу не лише зменшити людський фактор у процесі перевірки програмного забезпечення, а й значною мірою пришвидшити виконання рутинних, багаторазово повторюваних тестів. Завдяки використанню спеціалізованих фреймворків, інструментів і мов програмування, автоматизовані тести можуть охоплювати як окремі функції та модулі, так і цілі сценарії поведінки користувача, імітуючи складні випадки

використання продукту у реальному середовищі, що створює необхідність більш детального розгляду концепції автоматизованого тестування, його основних переваг, обмежень, технологій реалізації, а також місця в загальному процесі забезпечення якості програмного забезпечення[14]. У зв'язку із цим автоматизоване тестування UI веб-додатків постає як логічна й необхідна відповідь на сучасні вимоги. Автоматизація дозволяє значно скоротити час виконання тестів, підвищити частоту релізів, зменшити кількість ручної праці й людських помилок. Наприклад, звіт Applitools "2022 State of UI/UX Testing" вказує, що лише близько 30 % компаній регулярно тестують користувацький інтерфейс або візуальну правильність на кожному випуску, а понад 40 % тестових наборів виконуються довше ніж годину — і це стає вузьким місцем у швидкому реліз-циклі. [2]

Однак автоматизація сама по собі не є панацеєю: із зростанням масштабів тестових наборів з'являються нові виклики — дублювання сценаріїв, нестабільність (flaky-тести), складність підтримки, довгий час виконання. Саме тому **оптимізація** автоматизованих тестових сценаріїв набуває особливого значення: структуризація тестів, повторне використання підсценаріїв, параметризація, інтеграція з CI/CD, скорочення часу виконання — все це є засобами підвищення продуктивності тестування й зниження витрат на підтримку. Актуальність даної теми зумовлена також поточною динамікою:

- на рівні галузі — звіти про тенденції в тестуванні вказують, що UI/UX-тестування, автоматизація, «shift-left» підходи, безперервне тестування (Continuous Testing) стають універсальними вимогами.
- на рівні практики — компанії все частіше повідомляють про потребу скоротити час тестування, підвищити частоту релізів, впровадити автоматизацію; наприклад у звіті Applitools зазначено[15], що компанії, що використовують візуальну автоматизацію, досягають виконання тестової сесії менше ніж за 15–20 хв замість понад години.

- на рівні досліджень — академічні праці досліджують проблеми повторного використання UI-тестів, нестабільності сценаріїв, масштабованості UI-автоматизації.

З огляду на вищезазначене, дослідження, присвячене **оптимізації автоматизованих тестових сценаріїв для тестування UI веб-додатків**, займає важливе місце як у науковому дискурсі[16] (розвиток методів тестування, забезпечення якості), так і в практичній інженерній діяльності — оскільки забезпечення ефективності, стабільності й масштабованості UI-тестування стає ключовим фактором конкурентоспроможності веб-продуктів. Саме тому тема є своєчасною, значущою й має високу практичну цінність.

1.2. Огляд підходів до автоматизованого тестування

Автоматизоване тестування у сучасних веб-орієнтованих інформаційних системах розглядається як структурований процес перевірки функціональних, інтеграційних та інтерфейсних властивостей програмного забезпечення за допомогою формалізованих сценаріїв. У межах еволюції методологій забезпечення якості сформувалися різні підходи до побудови автотестів, які визначають характер покриття, глибину валідації та релевантні технічні засоби.

Центральним методологічним орієнтиром є піраміда тестування[17], що задає концептуальну модель раціонального розподілу тестових активностей. Згідно з цією моделлю, тестові сценарії структуровано за рівнями складності, охоплення та вартості виконання, що дозволяє забезпечити оптимальний обсяг тестування за умов обмежених ресурсів.

- **Перший рівень** утворюють **модульні тести (Unit)**, які функціонують у максимально ізольованому середовищі та перевіряють поведінку окремих функцій, класів або модулів. Вони відзначаються високою швидкістю виконання, низькою собівартістю розробки та значним внеском у запобігання дефектам на початкових етапах життєвого циклу ПЗ.

- **Другий рівень** представлений **інтеграційними тестами**, орієнтованими на аналіз взаємодії між компонентами системи. Ці тести використовуються для виявлення помилок у логіці інтеграції, роботі API, баз даних та зовнішніх сервісів. Вони характеризуються більшою складністю, оскільки функціонують у частково або повністю інтегрованому середовищі та мають помірну вартість супроводу.
- **Третій рівень** становлять **тести кінцевого користувача (UI / E2E)**, що імітують реальні користувацькі сценарії та забезпечують комплексну перевірку системи в цілому. Як найбільш ресурсомісткі, повільні та залежні від стабільності інтерфейсу, вони посідають найменший обсяг у загальній структурі тестування. Їх застосування виправдане лише для бізнес-критичних функцій, без яких продукт неможливий.

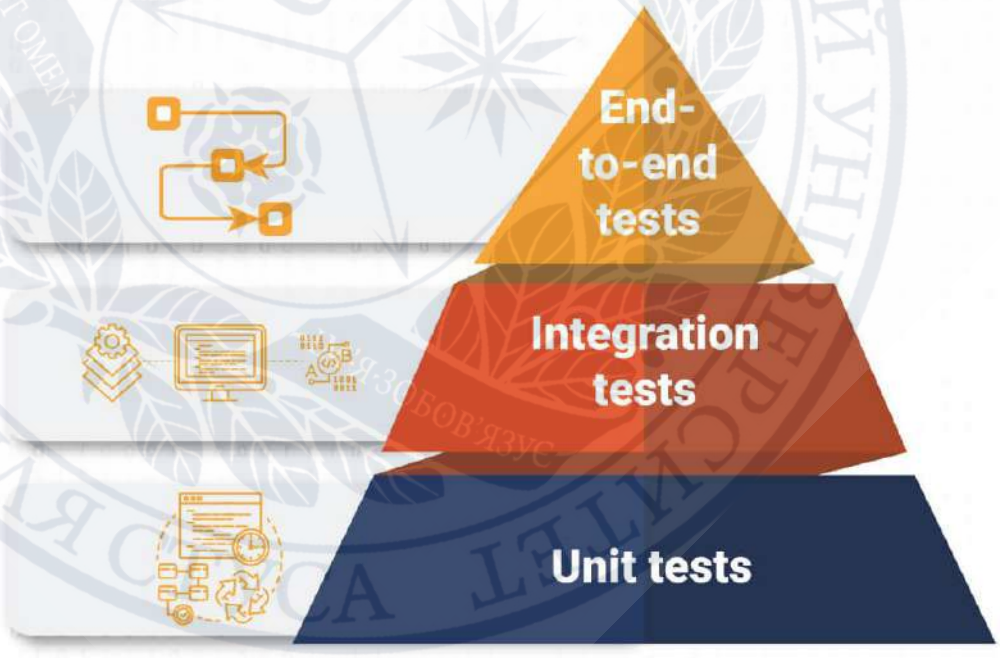

Рисунок 1.1. Схематичне зображення піраміди тестування

У сучасних підходах до тестової інженерії також розглядаються альтернативні інтерпретації класичної піраміди, зокрема **“тестовий ромб”[18]**, а також моделі, що передбачають збільшення питомої ваги інтеграційних тестів за

рахунок оптимізованої інфраструктури. Проте базовий принцип залишається незмінним: ефективність тестування досягається не шляхом збільшення кількості UI-тестів, а завдяки їх збалансованому співвідношенню з модульними та інтеграційними перевірками.

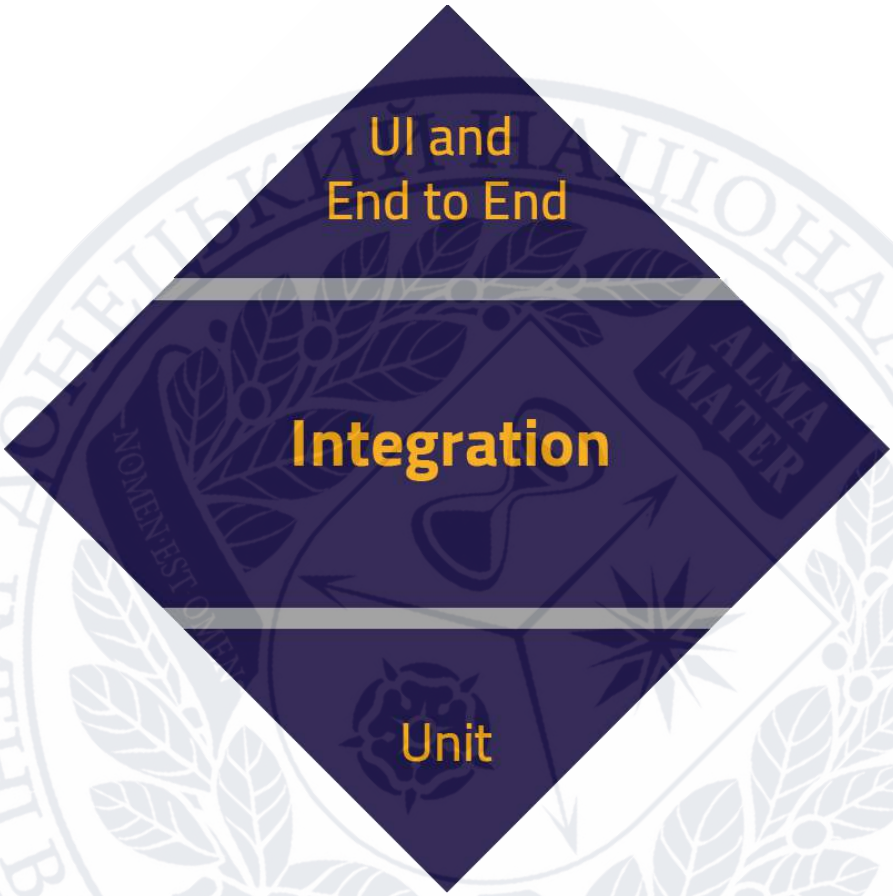

Рисунок 1.2. Схематичне зображення моделі тестового ромба

Таким чином, огляд підходів до автоматизованого тестування свідчить про необхідність системного застосування багаторівневої структури тестів відповідно до принципів піраміди тестування, що забезпечує технологічну керованість процесу, прогнозованість результатів та підвищення загальної якості програмного забезпечення.[19]

Окрім рівневої структури, автоматизоване тестування класифікується і за **видами тестових активностей**, що різнобічно охоплюють роботу продукту:

1. **Функціональне тестування (Functional Testing)** — перевірка відповідності фактичної поведінки програмного забезпечення визначеним

функціональним вимогам. Автоматизація цього виду тестування є найбільш поширеною, оскільки дозволяє ефективно покривати основні бізнес-процеси та найважливіший функцінал у вигляді повторюваних дій пересічного користувача.

2. **Тестування продуктивності (Performance Testing)** — оцінка швидкодії системи в стресових ситуаціях під різним навантаженням, включно з часом відгуку, використанням ресурсів та стабільністю. До підвидів цього тестування належать:

Load Testing — визначення поведінки системи при очікуваному робочому навантаженні.

Stress Testing — оцінка роботи системи в умовах перевантаження, коли ресурси (як-от кількість підключень до БД, залишок ОЗП на серверній машині) близькі або перевищують критичні межі.

Spike Testing — аналіз реакції системи на різкі, стрибкоподібні зміни навантаження.

Endurance/Soak Testing — дослідження стабільності системи протягом тривалого часу неперервної роботи.

3. **Регресійне тестування** — перевірка того, що внесені зміни у код не спричинили появу нових дефектів. Саме автоматизація регресії є основною перевагою впровадження автотестів[20], адже дозволяє регулярно виконувати великий обсяг тестів у рамках CI/CD. При відсутності автоматизованого регресу на проекті необхідним було б залучення тестувальника, що повинен виконувати набір ідентичних мануальних перевірок функціоналу щоденно або під час кожного релізу задачі на проекті, що віднімає значну кількість ресурсів у команди тестування і само по собі є застарілим і неефективним підходом.

4. **Тестування безпеки (Security Testing)** — перевірка вразливостей, коректності механізмів автентифікації, авторизації, захисту даних і доступу. Автоматизація цього виду поступово стає все актуальнішою, особливо у великих веб-додатках.

5. **Юзабіліті-тестування** — аналіз зручності взаємодії користувача з інтерфейсом. Хоча цей вид тестування переважно виконується вручну, певні його аспекти (наприклад, коректність розмітки, наявність aria-атрибутів) підлягають частковій автоматизації.

Сукупність цих видів тестування формує комплексну стратегію забезпечення якості, у межах якої автоматизація дозволяє скоротити витрати на тестування, підвищити стабільність релізних циклів та забезпечити своєчасне виявлення дефектів. Піраміда тестування, у свою чергу, слугує методологічним орієнтиром щодо розумного співвідношення тестових активностей: перевага надається швидким та дешевим тестам, тоді як складні та ресурсомісткі перевірки застосовуються лише для критично важливого функціоналу.

1.3. Організація процесу тестування

У процесі розробки та підтримки веб-додатків організація процесу тестування передбачає послідовну та логічно структуровану роботу, яка охоплює всі етапи перевірки програмного забезпечення — від планування до постійного супроводу[21]. Тестування інтегрується в загальний процес розробки та виконується паралельно з основними етапами створення продукту, що дозволяє своєчасно виявляти проблеми та підтримувати стабільність системи.

Початком процесу є планування, під час якого команда визначає, що саме потрібно протестувати, якими методами це буде здійснено та які ресурси для цього потрібні. На цьому етапі створюється загальне бачення тестової стратегії: які види тестів застосовуватимуться, який обсяг роботи очікується та як відслідковуватимуться результати. Важливо також одразу визначити, яку частину перевірок доцільно автоматизувати. Після планування формується набір тестових сценаріїв. Він включає аналіз вимог, опис умов тестування та підготовку конкретних перевірок (test case), які дозволять оцінити коректність роботи функціональності. У випадку автоматизованого тестування на цьому етапі створюється структура тестового проєкту (т.з фреймворк), або якщо структура

фреймворку вже існує, проводиться лише написання автотестів для подальшого регресованого тестування.

Управління проектами є критично важливою складовою успіху в розробці програмного забезпечення, оскільки правильний вибір моделі та стратегії управління визначає ефективність тестування та здатність команди швидко реагувати на зміни в проекті. Серед основних підходів до управління проектами виділяють такі моделі, як Waterfall (каскадний підхід), Agile (гнучкі методології) та інші варіанти. Кожен з них має свої особливості, переваги та недоліки, що впливають на організацію процесу тестування.[22]

- **Waterfall (каскадний підхід).** Є традиційною моделлю розробки програмного забезпечення, яка передбачає чітке розділення процесу розробки на етапи, що виконуються послідовно. Кожен етап залежить від результатів попереднього, і після завершення одного етапу переходять до наступного. Основними особливостями Waterfall є чітке визначення вимог і планування на самому початку проекту, що дозволяє команді працювати за заздалегідь визначеним планом. Процес розробки проходить поетапно, і тестування зазвичай починається тільки після завершення розробки коду, а випуск продукту відбувається лише після фінального тестування. Перевагою Waterfall є те, що процес тестування можна добре організувати завдяки чіткій структурі проекту. Кожен етап чітко планується, що дозволяє детально контролювати хід робіт. Цей підхід також підходить для проектів із чітко визначеними вимогами, де зміни в процесі розробки малоймовірні. Однак серед недоліків слід зазначити відсутність гнучкості: якщо з'являються зміни у вимогах, повернення до попередніх етапів може бути складним. Також тестування проводиться лише після завершення кодування, що часто призводить до пізнього виявлення багів і високих витрат на виправлення дефектів.
- **Agile (гнучкі методології)**. Це набір принципів і практик для розробки програмного забезпечення, який акцентує увагу на гнучкості, швидкому реагуванні

на зміни та інтерактивній взаємодії з замовником. Agile підходи, такі як Scrum, Kanban, XP (Extreme Programming), дозволяють частіше переглядати та коригувати вимоги на кожному етапі розробки. Agile передбачає ітераційний процес, де проект поділяється на малі етапи або ітерації, кожна з яких включає розробку частини функціональності та її тестування. Кожна ітерація зазвичай триває кілька тижнів, і після кожного спринту (ітерації) замовник отримує функціонуючу частину продукту. Важливими особливостями Agile є тісна взаємодія з клієнтом, регулярні зустрічі для уточнення вимог та постійна адаптація до змін. Перевагами Agile є висока гнучкість і адаптивність, оскільки регулярні зустрічі з замовниками дозволяють оперативно коригувати вимоги. Відкрита комунікація та зворотний зв'язок дозволяють швидко реагувати на зміни, а тестування проводиться на кожному етапі розробки, що дозволяє виявляти дефекти на ранніх стадіях. Це також підвищує задоволеність клієнта, оскільки продукт поставляється поступово, вже на ранніх етапах. Однак у Agile є й недоліки. Наприклад, через часті зміни вимог важко підтримувати проект у межах бюджету та часу. Крім того, високі вимоги до самодисципліни команди та ефективної комунікації можуть бути складними для деяких організацій. Відсутність чітких пріоритетів може призвести до затримок або відхилень від початкового плану.

- **Kanban**. Є ще одним підходом з категорії Agile, який фокусується на візуалізації процесу розробки та тестування і на ефективному управлінні потоком робіт. Він використовує візуальні дошки для відображення всіх етапів роботи, що дозволяє всім учасникам процесу бачити прогрес і визначати вузькі місця. Однією з ключових особливостей Kanban є візуалізація процесу, що дає змогу команді чітко бачити поточний стан задач і своєчасно реагувати на проблеми. Іншою важливою рисою є безперервне вдосконалення: команда постійно аналізує ефективність процесу та шукає шляхи покращення, щоб усунути затримки й підвищити продуктивність.Перевагою Kanban є його здатність адаптуватися до існуючих процесів без необхідності кардинальних змін. Завдяки постійному моніторингу

роботи команда може швидко визначити проблеми і вирішити їх. Крім того, цей підхід дозволяє значно покращити ефективність роботи завдяки управлінню потоком завдань. Проте методологія вимагає постійної уваги до процесу для виявлення і усунення затримок, а також може не підходити для великих проектів, де існує багато складних етапів, які важко контролювати.[23]

Рисунок 1.3. Зовнішній вигляд Kanban-дошки

Організація тестового середовища є важливою умовою для коректної роботи тестів. Середовище має максимально відповідати реальним умовам використання системи, тому налаштовується база даних, сервери, доступи, конфігурації та всі необхідні технічні компоненти. Стабільність середовища є критично важливою, особливо для UI-тестів, оскільки навіть незначні зміни або заторможення з боку серверної частини додаку можуть вплинути на результати їх виконання або навіть повністю перервати їх перебіг.[24]

Далі відбувається безпосереднє виконання тестів. Це може бути як ручне тестування, так і автоматизовані запуски, інтегровані в CI/CD-процеси. Під час запуску збираються дані про успішні та невдалі перевірки, фіксуються всі знайдені

помилки та незвичні поведінки системи. Для автоматизованого тестування важливим є стабільний запуск тестів, наявність логів і можливість легко простежити, на якому етапі виникла помилка і що відбувалось з системою в цей момент.

Після завершення виконання тестів команда переходить до аналізу результатів. Це включає оцінку того, які помилки були знайдені, наскільки вони критичні, як впливають на продукт та які зміни потрібно внести. На основі отриманих даних формується загальна картина якості поточної версії системи та визначаються наступні кроки у розробці.[25]

Завершальним етапом є супровід тестів. Оскільки веб-додатки постійно оновлюються, тестова база має регулярно адаптовуватись до нових функцій, змін у дизайні чи оновлень бекенду. Особливо це стосується автоматизованих UI-тестів, які часто потребують коригування після змін інтерфейсу. Своєчасний супровід тестів дозволяє уникнути накопичення технічного боргу та підтримувати ефективність тестового процесу. Дослідження «Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing» [3] стверджує, що до 80% багів на проекті може бути відловлено суто завдяки регресійним автотестам.

Загалом організація процесу тестування є безперервною діяльністю, що поєднує планування, виконання та постійний розвиток тестів. Від того, наскільки грамотно цей процес побудований, залежить якість кінцевого продукту, ефективність розробки та швидкість реагування на можливі проблеми.

1.4. Методологічні підходи

Підходи до автоматизованого тестування UI веб-додатків визначають сукупність принципів, логічних моделей та організаційних рішень, які формують основу для побудови ефективного, масштабованого та відтворюваного процесу тестування[26]. Їх чітке розуміння дозволяє не лише підвищити якість тестового

покриття, а й забезпечити адаптивність тестової системи до зростаючих вимог, змін програмного продукту та обмежень реального середовища розробки.

Одним із основних методологічних підходів є **процесно-орієнтований підхід**, у межах якого процес тестування розглядається як послідовність регламентованих етапів: аналіз вимог, проєктування тестових сценаріїв, їх реалізація, виконання, аналіз результатів і подальша оптимізація. Цей підхід забезпечує структурованість робочого процесу, формує передбачуваність результатів і створює умови для контролю якості кожного етапу. Застосування процесних моделей, таких як V-Model або W-Model, дозволяє синхронізувати роботу тестувальників із командами розробки та бізнес-аналітики. Іншим важливим підходом виступає **ризик-орієнтоване тестування**, що ґрунтується на аналізі можливих точок виникнення дефектів та визначенні їх пріоритетності.[27] Для UI-тестування цей підхід є особливо актуальним, адже користувацький інтерфейс часто містить динамічні елементи, інтерактивні компоненти та логіку, що залежить від великої кількості факторів (взаємодія з API, робота браузера, продуктивність, адаптивність). Ризик-орієнтоване тестування дозволяє концентрувати ресурси на найбільш важливих сценаріях, зменшувати час розробки тестів та скорочувати витрати на подальшу підтримку. Також важливу роль у формуванні стійкої тестової архітектури відіграє **модульно-компонентний підхід**, що передбачає поділ тестової системи на окремі логічні модулі: сторінки, елементи інтерфейсу, сценарії, утиліти та сервіси. Такий підхід знижує зв’язність між тестами, сприяє повторному використанню компонентів та дозволяє швидко адаптувати тестову систему до змін у веб-додатку. Архітектурні патерни, такі як Page Object Model, Screenplay Pattern або Page Factory, дають змогу істотно спростити реалізацію тестів і мінімізувати дублювання коду.

У межах автоматизації тестування інтерфейсу важливим компонентом є відповідність тестових рішень загальним принципам архітектури програмних систем. **Архітектурні підходи** допомагають сформувати структуру тестової

системи таким чином, щоб вона була легко масштабованою, зрозумілою та придатною для подальшого розвитку. Вони визначають спосіб організації файлів, класів, методів і логіки тестів, а також взаємодію між ними.

Одним із фундаментальних принципів є **відокремлення відповідальностей (Separation of Concerns)**. Він передбачає розділення логіки взаємодії з UI, бізнес-логіки тесту та логіки підготовки тестових даних. У контексті UI-тестування це означає, що тест не має напряму взаємодіяти з локаторами елементів, а повинен виконувати дії через проміжні об’єкти — сторінки або компоненти. Така структуризація робить тести більш стійкими до змін у DOM-структурі та спрощує рефакторинг.

Застосування принципів **SOLID** також є важливим елементом побудови архітектури тестової системи. Наприклад, принцип відкритості/закритості (Open/Closed Principle) дозволяє розширювати функціонал існуючих тестових компонентів без їх модифікації, що значно підвищує стабільність системи. Принцип інверсії залежностей (Dependency Inversion Principle) сприяє абстрагуванню взаємодії з драйверами, сервісами чи інструментами, що забезпечує більшу гнучкість у виборі технологій. Окрім того, сучасні тестові системи часто будуються з урахуванням **патернів розробки**, що дозволяють мінімізувати складність проєкту. Наприклад, патерн **Facade** допомагає приховати складність низькорівневих операцій взаємодії з UI, тоді як **Factory Method** може застосовуватися для ініціалізації сторінок або компонентів залежно від типу тестового середовища, пристрою чи мови локалізації.[28]

Важливо також визначити правильну стратегію управління залежностями між тестами та уникати надмірної зв’язаності. Для цього застосовуються підходи, що базуються на **ін’єкції залежностей**, використанні модульних утиліт та загальних сервісів, що забезпечують логування, репортинг, генерацію даних і конфігурацію. Сучасні процеси розробки програмного забезпечення дедалі більше тяжіють до інтегрованих моделей, у яких тестування є невід’ємною частиною життєвого циклу

розробки. Тому одним із найбільш прогресивних підходів є **DevOps-орієнтований підхід**, що поєднує автоматизацію тестів з інструментами CI/CD і загальною культурою безперервної інтеграції та доставки. DevOps-орієнтоване тестування передбачає виконання тестових сценаріїв щоразу, коли відбувається зміна коду, оновлення залежностей або публікація нової версії додатка.[29] Це дозволяє оперативно виявляти дефекти на ранніх етапах, коли вартість їх виправлення є мінімальною. Крім того, автоматизоване тестування у межах CI/CD забезпечує високу швидкість релізного циклу, що є особливо важливим для веб-додатків з інтенсивними оновленнями. При реалізації DevOps-орієнтованого підходу активно використовуються спеціалізовані інструменти, такі як GitHub Actions, GitLab CI, Jenkins, Bitbucket Pipelines, Azure DevOps. Вони дозволяють автоматизувати запуск тестів, збір результатів, формування звітів і розгортання тестового середовища. У поєднанні з контейнеризацією (Docker) та оркестрацією (Kubernetes) це дає можливість створювати стандартизовані, незалежні та масштабовані тестові оточення, у яких результати тестів є відтворюваними[30]. Особливу роль у DevOps-підході відіграє **тестування в паралелі**, яке дозволяє суттєво скоротити час виконання тестового набору. Використання Playwright у поєднанні з CI-платформами забезпечує можливість розподіляти тести між кількома середовищами та раннерами, що збільшує продуктивність та дає змогу інтегрувати UI-тести у швидкі релізні цикли.

Таким чином, DevOps-орієнтований підхід не лише сприяє інтеграції тестування в загальний процес розробки, а й формує культуру відповідальності та контролю якості, в якій кожна зміна коду автоматично перевіряється на відповідність вимогам і стабільність інтерфейсу.

Окремого значення у методології автоматизованого тестування UI набувають підходи, що поєднують тестування з процесами розробки програмного забезпечення, зокрема **Test-Driven Development (TDD)** та **Behavior-Driven Development (BDD)**.

TDD (розробка, керована тестами) передбачає, що реалізація функціональності розпочинається з написання тестів, які описують очікувану поведінку системи. Тест на початковому етапі свідомо не проходить, оскільки реалізація ще відсутня; лише після цього починається написання коду, необхідного для його успішного виконання. У контексті UI-тестування та інструментів на кшталт Playwright, TDD дає змогу формувати чіткі вимоги до інтерфейсу, уникати надмірної логіки в тестах та забезпечувати стабільність сценаріїв. Проте застосування TDD у UI-автоматизації має певні обмеження — зокрема, високі витрати часу при активних змінах інтерфейсу та залежність тестів від ще нестабільних UI-компонентів. Незважаючи на це, TDD ефективний для стабільних частин інтерфейсу або для тестування внутрішніх компонентів, що відповідають за логіку взаємодії.

BDD (розробка, керована поведінкою), навпаки, орієнтується не на контроль коду, а на опис поведінки системи з точки зору користувача. Тести у BDD формулюються мовою, зрозумілою бізнес-аналітикам, розробникам та тестувальникам, зазвичай у форматі *Given – When – Then*. Такий підхід сприяє узгодженню очікувань між усіма учасниками проєкту та підвищує зрозумілість тестової документації.[31] У контексті UI-тестування BDD дозволяє створювати сценарії, що максимально наближені до реальних користувацьких потоків, підвищуючи цінність тестів для бізнесу. Використання відповідних фреймворків (Cucumber, Behave, SpecFlow, Playwright-test з plugins для BDD) забезпечує можливість легко інтегрувати поведінкові сценарії у загальну архітектуру проєкту. Порівняно з TDD, підхід BDD краще підходить для тестування UI, оскільки акцентує увагу саме на користувацькому досвіді. Проте він потребує додаткових зусиль на підтримку feature-файлів, погодження описів сценаріїв та узгодження термінології між учасниками команди. У великих проєктах BDD дозволяє суттєво підвищити прозорість та керованість тестового процесу, тоді як у невеликих

командах його застосування має сенс лише у разі наявності складної бізнес-логіки або високих вимог до документації.

Таким чином, TDD та BDD є важливими методологічними підходами, що дозволяють узгодити процеси розробки та тестування, забезпечити вищий рівень передбачуваності результатів і підвищити якість автоматизованих UI-сценаріїв. Їх застосування залежить від цілей проєкту, складності інтерфейсу та вимог до швидкості й точності зворотного зв’язку, проте в обох випадках вони сприяють оптимізації тестового процесу.

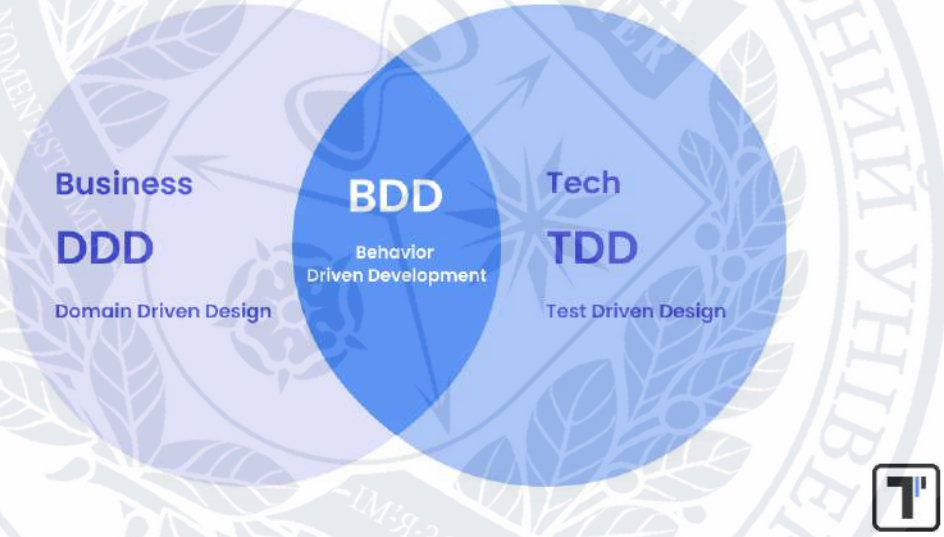

Рисунок 1.4. Схематичне порівняння моделей TDD та BDD

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було розглянуто теоретичні основи, необхідні для розуміння процесу автоматизованого тестування користувацьких інтерфейсів веб-додатків. На початку було визначено ключові поняття та терміни, що формують базу для подальшого дослідження. Далі проведено огляд основних підходів до автоматизації UI-тестування, їх можливостей, переваг та обмежень. Окрема увага була приділена організації процесу тестування — від аналізу вимог до виконання тестових сценаріїв і оцінювання результатів. Це дозволило показати, як правильне планування та структура тестового процесу впливають на якість і надійність автоматизації. У підрозділі, присвяченому методологічним підходам, розглянуто різні способи побудови та оптимізації тестових систем, включаючи процесно-орієнтований, ризик-орієнтований, модульно-компонентний підходи, а також практики TDD, BDD та інтеграцію тестування в DevOps-процеси. Це дало змогу окреслити сучасні тенденції та підходи, що забезпечують ефективність і стійкість автоматизованих UI-тестів.

Узагальнюючи, перший розділ сформував теоретичне підґрунтя, необхідне для подальшого аналізу методів оптимізації та практичної реалізації автоматизованих тестових сценаріїв, що будуть розглянуті у наступних розділах.

РОЗДІЛ 2
ОСНОВИ ОПТИМІЗАЦІЇ АВТОМАТИЗОВАНИХ ТЕСТОВИХ СЦЕНАРІЇВ

2.1. Особливості тестування UI веб-додатків

Тестування UI веб-додатків є одним з найбільш складних та ресурсозатратних напрямів забезпечення якості програмного забезпечення. На відміну від модульних або API-тестів, які працюють переважно з логічними структурами та стабільними програмними інтерфейсами, UI-тести взаємодіють з динамічним середовищем браузера, DOM-структурою, стилями, мережевими запитами та поведінкою користувача.[32] Через це тестування інтерфейсу має ряд специфічних особливостей, що впливають як на побудову тестових сценаріїв, так і на їх оптимізацію, підтримку та продуктивність.

Однією з основних особливостей UI веб-додатків є постійна динамічність елементів інтерфейсу. Сучасні веб-технології активно використовують JavaScript-фреймворки (React, Angular, Vue, Svelte), що оновлюють DOM асинхронно, без перезавантаження сторінки. Це призводить до того, що:

- Елементи можуть з’являтися із затримкою або після виконання певного сценарію;
- DOM може швидко змінювати структуру;
- Старі локатори легко стають неактуальними;
- Одночасно існують декілька станів однієї сторінки.

Такі особливості створюють додаткові проблеми для тестування: необхідно коректно очікувати появу елементів. Тести стають дуже нестабільними через так звані «перегони подій». Складно визначити момент, коли інтерфейс дійсно готовий до виконання тестової події. Також постає на заваді непередбачуваність анімацій та переходів між станами.

Всі ці чинники значним чином впливають на побудову тестових сценаріїв: вони мають враховувати логіку завантаження сторінок, можливість появи спінерів (loading indicators), роботу з асинхронністю та поведінку як клієнтської, так і серверної частини.[33]

Також важливою особливістю UI-тестування є залежність результатів від конкретного браузера. Різні браузери використовують різні механізми рендерингу (Blink, WebKit, Gecko), що може спричиняти різну поведінку UI-компонентів: відмінності в обрахунку CSS-властивостей (особливо margin, padding, flex-layout, grid), різні алгоритми інтерпретації DOM-подій, специфічні невідповідності у виконанні JavaScript (частково навіть при використанні сучасних браузерів), різна стабільність анімацій та транзицій. Через все це UI-тести потребують перевірки у декількох браузерних середовищах. Сучасні фреймворки, такі як Playwright, забезпечують кросбраузерність, але оптимізація тестів передбачає: зменшення залежності тестів від візуальної частини UI, використання універсальних локаторів, уникнення тестів, чутливих до стилів та мінімізацію валідацій, що ґрунтуються на піксельних координатах. Таким чином, чим більш уніфіковано побудовані тести, тим простіше забезпечити їх стабільність у різних браузерах.

Тестування UI моделює реальну поведінку користувачів. Тому тестові сценарії мають відтворювати дії, які виконуються в браузері, це стосується таких буденних дій як кліки, введення тексту, прокручування (scroll) сторінок, перетягування елементу (т.з drag-and-drop), заповнення різноманітних форм та навігація між розділами контент-елементу. На відміну від API-тестів, які працюють із запитами та відповідями у формалізованому вигляді, UI-тести залежать від того, як вище зазначені дії тестового користувача обробляються в браузерному контексті. Це робить їх гіперчутливими до будь-яких змін: швидкість системи, продуктивність тестового пристрою (мобільний телефон, комп’ютер, смарт-побутова техніка), стабільність мережевого з’єднання, поведінка анімацій на UI, обробка AJAX[6]-

запитів. При цьому при написання тестів потрібно враховувати що веб-елементи в DOM-можуть мати різні стани окрім простого перебування у користувацькому інтерфейсі: disabled, readonly, hidden, hovered, focused та pressed. Це створює додаткові труднощі під час написання тестів, оскільки необхідно враховувати коректну зміну вищезгаданих станів, реакцію інтерфейсу на неочікувані випадки (exceptions), поведінку при одночасній взаємодії з декількома компонентами, різні сценарії роботи з формами та валідаціями. Недостатнє врахування таких важливих факторів під час автоматизації неминуче призведе до частих падінь тестів через неочікуванні стани контент елементів та різноманітні проблеми з локаторами. Це робить важливим використання стабільних локаторів (role-based, testId-based) та точного визначення станів перед виконанням дій. При цьому підтримка автоматизованих UI-тестів є складним та ресурсозатратним процесом, що зумовлено їхнім місцем у верхньому шарі піраміди тестування. UI-тести є найдорожчим типом тестових сценаріїв через низку чинників, серед яких варто виокремити складність реалізації, тривалість виконання, високу чутливість до змін у програмному забезпеченні, потребу у розгалуженій інфраструктурі та необхідність регулярного рефакторингу. У зв’язку з цим оптимізація автоматизованого UI-тестування набуває стратегічного значення. Вона дозволяє скоротити час проходження тестів, підвищити їхню стабільність, зменшити витрати на підтримку, а також покращити охоплення функціональності та загальну ефективність тестового набору. Чим масштабнішим стає програмний продукт, тим вагомішою є потреба в оптимізації, адже без неї тестова система втрачає здатність до масштабування.

Одним із важливих напрямів складності UI-тестування є аспекти безпеки, авторизації та роботи з даними. Перевірка механізмів аутентифікації, управління користувацькими ролями та правами доступу формує специфічні проблеми: необхідність створення великої кількості тестових користувачів, швидке забруднення тестової бази даними, нестабільність сесій та залежність тестів від

стану конкретних акаунтів. Для мінімізації цих ризиків застосовуються такі підходи, як попередньо підготовлені дані (fixtures), мокування або спрощення етапів авторизації, збереження cookies та токенів між тестовими сесіями, а також ізоляція тестових середовищ. Ці техніки дозволяють забезпечити керованість та повторюваність тестових сценаріїв, що є критичним чинником для підвищення їхньої стабільності.[34]

Ще однією характерною особливістю UI-тестування є залежність від адаптивного дизайну сучасних веб-додатків. Інтерфейси змінюються залежно від параметрів пристрою, зокрема розміру екрана, орієнтації, типу вводу та щільності пікселів (DPI). Це призводить до необхідності враховувати мобільні та планшетні розміри екранів, коректність обробки touch-жестів, а також різноманітні варіації верстки. Інструменти типу Playwright або інші сучасні фреймворки забезпечують можливість емуляції різних пристроїв, проте це також призводить до збільшення кількості тестових сценаріїв та підвищення загального навантаження на тестову систему. Таким чином, адаптивність інтерфейсу, хоча й дозволяє покращити користувацький досвід, водночас створює додаткові виклики для оптимізації UI-тестування.

Підсумовуючи, особливості тестування інтерфейсів веб-додатків визначаються поєднанням великої кількості технічних та організаційних факторів: варіативністю інтерфейсу, асинхронністю взаємодій, залежністю від браузера і мережевої інфраструктури, наявністю багатоступеневих сценаріїв виконання та високою чутливістю до змін у програмному продукті. Сукупність цих чинників формує потребу в системному підході до оптимізації тестового набору, побудові стійкої тестової архітектури та зменшенні витрат часу на розробку й підтримку автоматизованих тестів. Ґрунтовне розуміння специфіки UI-тестування є фундаментом для створення ефективних оптимізаційних рішень, які будуть розглянуті у наступних підрозділах.

Оскільки веб-додатки активно комунікують із сервером через такі інструменти як REST API, GraphQL, WebSocket, server-side events, UI-тести автоматично стають залежними від безлічі чинників з боку бекенду: швидкість виконання запитів, стабільність бекенду, реакції інтерфейсу на помилки сервера.[35]

Реальні сервери часто є нестабільними у тестовому середовищі, що робить UI-тести ненадійними. Тому оптимізація включає використання моків та стабів для стабільності та уникнення надмірної кількості API-викликів у тестах.

Застосування цих технік дає змогу зменшити складність тестового середовища, підвищити передбачуваність результатів тестування й оптимізувати витрати на виконання тестових сценаріїв, особливо у високонавантажених або розподілених системах.

Стаби (Stubs) являють собою спрощені реалізації залежностей, які повертають заздалегідь визначені значення без складної бізнес-логіки та без взаємодії з реальними зовнішніми сервісами. Їхнє основне завдання — забезпечити стабільність та контрольованість умов виконання тесту. Використання стабів є особливо корисним у випадках, коли тестувальний сценарій передбачає взаємодію з непередбачуваними або важкодоступними ресурсами, наприклад, сервісами третіх сторін, базами даних чи API з нестабільним часом відповіді. Таким чином, стаби сприяють зниженню флуктуацій у часі виконання тестів і полегшують виявлення помилок, пов’язаних безпосередньо з логікою інтерфейсу, а не із зовнішніми залежностями.

Моки (Mocks), натомість, є більш «активними» тестовими дублерами, які не лише імітують поведінку реальних об’єктів, а й дозволяють перевіряти взаємодію з ними. Вони фіксують факт виклику певних методів, відповідність переданих параметрів і дотримання очікуваних сценаріїв використання. Саме тому моки застосовують для уточнення коректності логіки інтеграції між компонентами інтерфейсу, коли критично важливо контролювати не лише результат, а й спосіб

досягнення цього результату. Такий підхід підсилює здатність тестових сценаріїв виявляти структурні дефекти й гарантує, що інтерфейс правильно координує роботу з внутрішніми модулями або зовнішніми службами.

Інтеграція моків та стабів у процес побудови тестових сценаріїв сприяє підвищенню відтворюваності тестів, скороченню їхнього часу виконання та зменшенню ризику появи «хибних» збоїв, пов'язаних із поведінковими особливостями зовнішніх компонентів. Це, у свою чергу, робить такі сценарії більш придатними для регулярного автоматизованого запуску, зокрема у рамках CI/CD-конвеєрів. Раціональне використання цих технік дозволяє тестувальникам зосередитися на перевірці логіки інтерфейсу та оптимізації користувацького досвіду, не витрачаючи ресурси на відтворення складних і нестабільних середовищ.

Флакіні є однією з найскладніших та найбільш ресурсозатратних проблем автоматизованого тестування, зокрема у сфері перевірки користувацьких інтерфейсів веб-додатків.[36] Під флакі-тестами зазвичай розуміють такі автоматизовані сценарії, які за однакових умов виконання демонструють непередбачувану поведінку: вони можуть успішно завершуватися під час одного запуску та давати збій під час іншого, при цьому відсутні будь-які зміни у вихідному коді, тестовому середовищі чи конфігурації системи. Така непослідовність суттєво підриває довіру до результатів тестування, ускладнює процес аналізу дефектів та погіршує загальну ефективність процесу забезпечення якості. Джерела флакіні є багатоманітними та охоплюють широкий спектр технічних і організаційних чинників. Однією з найпоширеніших причин є недетермінований характер асинхронних операцій, який є характерним для сучасних веб-додатків. Елементи інтерфейсу можуть відображатися з різними затримками, мережеві запити можуть повертатися у непередбачуваному порядку, а внутрішні механізми браузера можуть по-різному обробляти черги подій. Усе це створює додаткові точки варіативності, які складно контролювати у рамках жорстко визначених тестових сценаріїв.

Іще одним суттєвим джерелом нестабільності є залежність тестів від зовнішніх сервісів чи інфраструктурних компонентів. Навіть короткочасна недоступність API, неповний відгук сервера або тимчасове зростання латентності мережі можуть призвести до випадкових збоїв, які, однак, не відображають реальної якості продукту. Такі проблеми особливо характерні для інтеграційних та енд-ту-енд тестів, у яких під час виконання зазвичай перевіряється повний ланцюжок взаємодії системи з реальним середовищем.

Флакіні також може виникати через неналежну синхронізацію тестових кроків. Невідповідність очікуваних станів інтерфейсу фактичним, використання жорстких таймаутів, прив'язка тесту до часового інтервалу замість логічної події — усе це створює умови, у яких тест успішно проходить лише випадково. Такі сценарії можуть залишатися непомітними тривалий час, оскільки проявляють себе лише в окремих, важко відтворюваних ситуаціях, що значно ускладнює їх діагностику. Нестабільність тестів негативно впливає на весь цикл розробки програмного забезпечення. По-перше, флакі-тести збільшують час аналізу результатів тестування, оскільки учасникам команди доводиться витрачати додаткові ресурси на розмежування справжніх дефектів і хибнопозитивних збоїв. По-друге, вони сприяють зростанню недовіри до автоматизації, що може призводити до зниження цінності тестових звітів і навіть до відмови частини команди від активного використання автоматизованих тестів. По-третє, флакіні сповільнює функціонування CI/CD-процесів: нестабільні тести часто блокують збірки, змушують перезапускати пайплайни або призводять до штучного збільшення часу виконання через додаткові перевірки.[37] Таким чином, флакіні є комплексною проблемою, яка потребує ґрунтовного аналізу та системної роботи. Її наслідки охоплюють як технічні, так і організаційні аспекти життєвого циклу розробки, а контроль за нестабільними тестами є критичною умовою забезпечення надійності автоматизованого тестування. Глибоке розуміння природи флакіні та методів її

мінімізації є невід'ємною складовою оптимізації тестових сценаріїв і загалом сприяє підвищенню якості веб-додатків.

2.2 Архітектурні підходи до побудови тестових сценаріїв

Архітектура сценаріїв в автотестах напряму впливає на забезпечення їхньої масштабованості, стабільності та придатності до довгострокової підтримки. У випадку тестування саме користувацького інтерфейсу веб-додатку питання стабільної та грамотної архітектури набуває особливої важливості, оскільки тестові сценарії взаємодіють з інтерфейсом, який є динамічним, асинхронним та чутливим до змін у продукті. Належно спроєктована архітектура дозволяє мінімізувати дублювання коду, скоротити час на модифікацію тестів, підвищити їхню надійність та забезпечити узгодженість тестового набору.[38]

Одним із базових принципів побудови ефективної тестової архітектури є модульність, яка передбачає розділення функціональності на незалежні компоненти, кожна з яких імплементує свій відокремлений функціонал. Модульні архітектурні підходи забезпечують можливість локальних змін без ризику порушення роботи інших частин тестового набору завдяки відсутності залежностей поміж ними. Розшарування тестової архітектури загалом поділяється на такі рівні:

- **рівень взаємодії з елементами інтерфейсу** (element interaction layer), де інкапсулюються базові операції з UI;
- **рівень бізнес-логіки тестів** (test logic layer), що описує поведінкові сценарії;
- **рівень даних та конфігурації** (data/configuration layer), який забезпечує передачу параметрів та тестових даних.

Таке розподілення дозволяє створювати сценарії, які є більш стійкими до змін інтерфейсу та легше підтримуються у великих проєктах.

Одним із найбільш поширених підходів до організації тестів є **Page Object Model (POM)** — патерн, що передбачає абстрагування веб-сторінок або компонентів інтерфейсу у вигляді окремих об’єктів. Кожен такий об’єкт містить

локатори елементів, методи для взаємодії з ними та логіку перевірок, які є характерними для відповідної частини застосунку.[39]

Переваги POM полягають у спрощенні підтримки тестів: зміни в UI потребують оновлення лише відповідного Page-об’єкта, а самі сценарії залишаються незмінними. Разом з тим традиційний POM часто ускладнює структуру проєкту у великих системах, що сприяло появі розширених підходів, таких як:

- **Screenplay Pattern** — формує тестування як взаємодію акторів (користувачів) з інтерфейсом через набір ролей і дій, що підвищує виразність та гнучкість сценаріїв;
- **Component Object Model** — орієнтований на компонентний підхід, коли UI розглядається як набір дрібних повторюваних компонентів;
- **Page Fragments** — фрагментує сторінку на логічні підблоки, що підвищує повторне використання коду та зменшує розмір окремих Page-об’єктів.

Застосування подібних патернів дозволяє забезпечити масштабованість архітектури та підвищити її адаптивність до змін зі сторони продукту.

Сучасні підходи до побудови тестової архітектури спрямовані на мінімізацію прямої взаємодії тестових сценаріїв з конкретними елементами UI. Замість цього рекомендується створювати **шари абстракції**, які приховують реалізацію низькорівневих операцій, таких як натискання кнопок, введення тексту чи вибір елементів списків. Такі абстракції дозволяють уніфікувати взаємодію з інтерфейсом, тим самим забезпечивши централізоване управління змінами. Також вони скорочують дублювання коду та спрощують підтримку тестів з плином часу. Підхід з використанням абстрактних шарів особливо ефективний у поєднанні з патернами POM або Screenplay, адже створює чітке розмежування обов’язків між різними рівнями тестової архітектури.[4]

Тестові сценарії можуть формуватися у двох основних стилях — **імперативному** та **декларативному**.

- **Імперативний підхід** описує послідовність дій тесту крок за кроком. Він забезпечує велику гнучкість, але часто призводить до складно підтримуваного коду.
- **Декларативний підхід** орієнтований на визначення очікуваного результату або стану системи. Він сприяє створенню більш абстрактних, зрозумілих та стабільних сценаріїв.

Сучасні фреймворки (Playwright, Cypress, TestCafe) дозволяють комбінувати обидва стилі, досягаючи балансу між гнучкістю та підтримуваністю.

Також у великих тестових проектах важливим аспектом є управління залежностями між компонентами тестової системи. Використання таких принципів, як **інверсія залежностей (Dependency Injection)**, дозволяє централізувати створення об'єктів тестової інфраструктури та спростити налаштування конфігурацій, забезчивши легку заміну компонентів (наприклад, заміну реальних тимчасово недоступних сервісів на моки). DI-підходи інтегруються у тестові фреймворки та знижують складність архітектури при зростанні проєкту. Подібні архітектурні рішення повинні супроводжуватися формальними правилами щодо стилю написання тестів, структуризації каталогів, принципів іменування, а також вимог до документації.[5] До переваг застосування чіткої стандартизації відносяться:

- зменшення вартості входу нових учасників команди;
- зниження ризику виникнення неуніфікованих або дубльованих сценаріїв;
- забезпечення цілісності тестового набору;
- полегшення масштабування тестового фреймворку.

Загалом, архітектурні підходи до побудови тестових сценаріїв визначають підтримуваність, якість, ефективність і довговічність тестової системи. Використання модульності, абстракцій, патернів проєктування, інверсії залежностей та стандартизації дозволяє створити стійку та гнучку архітектуру,

здатну адаптуватися до змін у веб-додатку та забезпечувати високу стабільність тестового набору. Раціонально побудована архітектура є фундаментом для оптимізації UI-тестів і значною мірою визначає успіх процесу автоматизації на проекті.

2.3 Методи оптимізації тестових сценаріїв

Оптимізація автоматизованих тестів є одним з найважливіших компонентів імплементації автоматизованого тестування у веб-продукті, оскільки забезпечує підвищення ефективності тестового набору, зменшення часу виконання, зниження витрат на підтримку та збільшення надійності тестової системи. Еволюція тестування веб-інтерфейсів призвела до появи широкого спектру методів оптимізації — від класичних підходів, що фокусуються на структуризації та спрощенні сценаріїв, до новітніх механізмів, основаних на інтелектуальних системах, паралелізації й адаптивній логіці виконання з залученням ШІ. У цьому підрозділі розглянуто найважливіші методи оптимізації, їхні переваги та обмеження, а також відповідність сучасним вимогам до масштабованих тестових систем.

Класичні підходи до оптимізації формувалися в умовах, коли автоматизоване тестування було менш масштабним, а архітектура тестів — відносно простою. Вони залишаються актуальними й сьогодні, утім часто потребують інтеграції зі сучасними техніками.[40]

1. Рефакторинг тестового коду. Спрямований на скорочення дублювання, поліпшення читаності та розподіл відповідальностей. До типових технік звичайного рефакторингу можна віднести винесення повторюваних кроків у допоміжні методи, централізацію локаторів в окремі класи та ізоляцію бізнес-логіки в тестах від технічної реалізації, що знаходиться «під капотом» тестового проекту. Рефакторинг забезпечує довгострокову стабільність тестового набору й зменшує «плату за складність», яка накопичується у великих системах.

2. Зменшення залежності від інтерфейсу. Надмірна кількість UI-тестів призводить до значного навантаження на тестову інфраструктуру. Класичним підходом є перенесення частини перевірок на нижчі рівні. До наглядних прикладів його використання можна віднести написання API-тестів замість UI-тестів, вибір модульних та інтеграційних тестів для бізнес-логіки та комбінацію smoke– та regression-покриття задля регулярної та поверхневої перевірки основного функціоналу. Таким чином досягається зменшення часу виконання та спрощення тестової архітектури.

3. Вибіркове виконання тестів. Цей підхід передбачає запуск лише релевантних сценаріїв залежно від змін у кодовій базі або конфігурації. Традиційно це здійснювалось вручну, але в сучасних системах автоматизовано за допомогою тестового таргетування (test impact analysis).

Одним із основних стержнів оптимізації є підвищення стабільності. Нестабільні (flaky) тести різко знижують довіру до автоматизації та збільшують витрати на мануальний аналіз результатів перебігу тестів. До основних методів протидії «крихким» тестам можна перерахувати:

- Правильна синхронізація - очікування подій (explicit wait – т.з явне очікування, наприклад в тесті очікується поява кнопки на екрані) замість таймерів (implicit wait – т.з неявне очікування, наприклад статичний таймаут 10 секунд)
- Повна ізоляція тестів – до неї відносяться очищення стану перед виконанням, унікальність тестових даних для підвищення унікальності тест-кейзу та використання sandbox-оточень. Завдяки ізоляції тестів вони стають набагато більш передбачуваними та відтворюваними.
- Мокування та стаби - використання моків на рівні API або окремих компонентів дозволяє уникнути нестабільних зовнішніх залежностей. Це особливо корисно для тестування важковідтворюваних станів системи.

З розвитком DevOps-підходів, інтеграційно-орієнтованої розробки та високочастотних релізних циклів сучасні системи автоматизованого тестування зазнали суттєвих трансформацій. Одним із проривних напрямів в галузі стало впровадження паралельного виконання тестів, що дало змогу суттєво скоротити загальний час проходження тестових наборів. Сучасні фреймворки тестування — такі як Playwright, Cypress та Selenium Grid останніх поколінь — передбачають можливість одночасного запуску великої кількості сценаріїв, розподіляючи їх між процесорними ядрами, контейнерами або навіть виділеними хмарними вузлами. У багатьох випадках це дозволяє скоротити тривалість тестового циклу у декілька разів, що є особливо важливим у середовищах CI/CD, де швидкість розгортання змін має критичне значення. Ще одним важливим сучасним напрямом оптимізації є контейнеризація процесів тестування. Використання Docker та Kubernetes забезпечує повну стандартизацію тестового середовища, виключає залежність від локальних конфігурацій розробників та дозволяє з високою точністю відтворювати умови виконання. У контейнеризованому середовищі усі компоненти тестування — браузери, драйвери, допоміжні сервіси та самі тести — існують у контрольованому просторі, що мінімізує ризики нестабільності та дозволяє легко масштабувати інфраструктуру. Таким чином не лише підвищує надійність виконання, але й забезпечується швидке розгортання нових середовищ для паралельних ранків.[41]

До ефективних та надійних методів оптимізації можна також віднести застосування механізмів кешування. Йдеться про збереження залежностей, попередньо ініціалізованих браузерних профілів, статичних ресурсів та тестових даних, що дає змогу прискорити кожний наступний запуск. В умовах CI системи такі оптимізації дозволяють уникати багаторазового завантаження однакових компонентів, що особливо важливо при частих запусків тестів у гілках розробки.

Новим напрямом оптимізації є динамічний розподіл тестових навантажень. У цьому випадку тестова система не використовує статично визначені набори тестів, а розподіляє їх відповідно до багатьох параметрів: тривалості виконання, історичної

стабільності, складності, а також пріоритетності з точки зору ризиків. Такий підхід забезпечує більш збалансоване використання ресурсів та підвищує загальну продуктивність тестових ранків, оскільки уникає ситуацій, коли один з паралельних потоків завершується значно швидше за інші. Загалом всі актуальні методи оптимізації спрямовані на масштабування, стабільність та ефективність, а отже їхнє застосування стає обов'язковою умовою для підтримки складних продуктових середовищ.

Одним із найпомітніших досягнень у галузі оптимізації автотестів є застосування механізмів Test Impact Analysis, який дозволяє точно визначати, які тести повинні виконуватися після певних змін у кодовій базі. Такий аналіз ґрунтується на зборі історичних даних про взаємодію тестів із різними компонентами системи, що забезпечує високу релевантність вибраних сценаріїв. На практиці це дозволяє уникати повного запуску всього тестового набору та зосереджуватися лише на найбільш критичних з точку зору поточних змін тестах. Використання такого підходу можна доволі влучно поєднувати з аналітикою поведінки тестів. Сучасні CI-платформи накопичують великі обсяги статистики виконань, що дає можливість виявляти тенденції нестабільності, прогресивні збільшення часу виконання, різноманітні аномалії тестового середовища або навіть баги. На основі такої інформації можуть прийматися обґрунтовані рішення щодо рефакторингу проблемних сценаріїв, зміни інструментів синхронізації або оптимізації архітектури тестів.

Особливе місце займають технології штучного інтелекту, які поступово інтегруються у сучасні фреймворки. Алгоритми машинного навчання можуть аналізувати величезний обсяг історичних даних виконання тестів: час проходження, імовірність падіння, тип помилки, появу нетипових затримок, взаємозв'язки між тестами та змінами в коді.[42] На основі цієї інформації моделі здатні прогнозувати тести, найбільш схильні до флакіні, і пропонувати тестовому інженеру шляхи їх оптимізації, або ж автоматично коригувати логіку очікувань,

таймаутів і синхронізаційних механізмів. Такий підхід зменшує кількість помилкових падінь і скорочує витрати на підтримку тестового набору. Також перспективним напрямом є автоматичне генерування та відновлення локаторів. Оскільки сучасні веб-інтерфейси постійно змінюються, традиційні локатори часто стають нестабільними. ШІ-модулі можуть аналізувати DOM-дерево, зіставляти елементи за їх візуальними характеристиками, атрибутами або контекстом використання й обирати найбільш стійкі селектори. У випадках, коли інтерфейс змінюється настільки, що локатор більше не працює, система здатна автоматично підібрати альтернативу, порівняти ефективність декількох варіантів та оновити тест без втручання тестувальника. За допомогою цього значно скорочується обсяг ручної праці та усувається одна з основних причин нестабільності UI-тестів – ненадійні локатори. ШІ активно використовується у процесі пріоритизації тестів та визначенні їх релевантності. Технології Test Impact Analysis, підсилені машинним навчанням, дозволяють швидко встановити, які саме тести необхідно виконувати після конкретної зміни у коді. На відміну від класичних підходів, що використовують статичні залежності, інтелектуальні системи враховують як історичні закономірності, так і поведінкові сигнали: які модулі найчастіше спричиняють регресії, які тести падають після змін у певних класах або які сценарії статистично важливі для стабільності продукту. Це забезпечує суттєве прискорення тестових прогонів і зменшення навантаження на інфраструктуру.

Важливу роль відіграє застосування ШІ у візуальному тестуванні. Моделі комп’ютерного зору здатні порівнювати скріншоти інтерфейсу не за піксельною відповідністю, а за семантичною подібністю, що дозволяє ідентифікувати справжні дефекти візуальної частини й уникати помилкових спрацювань, які властиві класичним методам. [50] Такі моделі можуть виявляти відхилення в компонентах інтерфейсу, їхній структурі, кольоровій гамі або поведінці. У поєднанні з аналізом стилів, HTML-структури та анімацій це створює значно точнішу та інтелектуальнішу систему візуального тестування.

Окремим напрямом новітніх методів оптимізації є віртуалізація сервісів. У складних системах UI-тести часто залежать від зовнішніх API, баз даних або сервісів інтеграції, що є потенційними точками нестабільності. Використання симульованих копій таких компонентів дозволяє повністю контролювати відповіді сервісів, тим самим усуваючи випадкові фактори нестабільності. Це особливо корисно у випадках, коли необхідно протестувати рідкісні, пограничні або важковідтворювані ситуації.

Порівняння класичних і сучасних підходів до оптимізації тестових сценаріїв відображає еволюцію вимог до автоматизованого тестування. У традиційних підходах основна увага приділялася якості та чіткості тестового коду, де переважали ручний рефакторинг, структуризація фреймворків та ручне усунення дублювання. Такий підхід передбачає орієнтацію на людське втручання, де стабільність тестів та їхня адаптація до змін залежать від досвіду та компетентності інженера. Сучасні підходи, навпаки, спрямовані на автоматизацію не лише тестів, але й процесів їх виконання, аналізу та обслуговування. Вони передбачають застосування масштабованих інфраструктурних рішень, хмарних обчислень, інтелектуального аналізу даних та автоматичного коригування тестового коду. Якщо класичні підходи роблять акцент на ручному контролі, то сучасні методи прагнуть мінімізувати людський фактор і передати більшість операцій автоматизованим системам. У контексті стабільності класичні методи передбачають ручне усунення флакіні, у той час як сучасні системи виявляють нестабільні тести автоматично, на основі статистичних моделей. У питаннях швидкості виконання класичні методи залежать від можливостей локальних середовищ, тоді як сучасні підходи використовують розподілені та паралельні обчислення, що забезпечує значно вищу продуктивність. Таким чином, сучасні методи є більш стійкими, продуктивними та гнучкими, хоча й вимагають складнішої інфраструктурної підтримки. Важливим елементом успішної оптимізації є узгодженість її методів із загальною архітектурою тестової системи.

Ефективна архітектура передбачає чітке розподілення відповідальностей між складовими тестового каркасу, централізоване управління залежностями та використання механізмів абстракції для зменшення прямого контакту тестів з елементами інтерфейсу[43]. Саме архітектурні рішення визначають, наскільки легко тестовий набір адаптується до зовнішніх змін, як швидко можуть бути інтегровані нові оптимізаційні підходи та наскільки стабільною залишається тестова система у довгостроковій перспективі. Особливе значення має можливість швидко замінювати залежності, наприклад, підмінювати реальні сервіси моками чи стабами, що дозволяє зменшити вплив нестабільних компонентів. Використання декларативного стилю у тестуванні також позитивно впливає на оптимізацію, оскільки зменшує складність опису сценаріїв та підвищує їхню читаність. Поширеним сучасним архітектурним підходом є компонентна модель тестування, де інтерфейс представлений як набір відносно незалежних блоків. Це полегшує повторне використання тестових модулів і сприяє гнучкішій оптимізації тестового фреймворку.

Отже, вимоги до тестування веб-додатків зумовлюють необхідність використання комплексних стратегій оптимізації, які включають як інфраструктурні, так і архітектурні та інтелектуальні рішення.[44] Еволюція методів оптимізації демонструє перехід від ручного, крок-за-кроком удосконалення тестів до високорівневих автоматизованих систем, здатних самостійно визначати проблемні області та адаптуватися до змін. Успішна оптимізація забезпечує не лише скорочення часу виконання тестів, але й підвищення стабільності, зменшення вартості підтримки та підвищення ROI від автоматизації загалом. Усе це робить оптимізацію тестових сценаріїв не просто бажаним етапом, а невід’ємним складником процесу забезпечення якості сучасних веб-додатків.

2.4. Автоматизоване управління тестовими сценаріями

Сучасні інструменти автоматизації тестування тісно пов’язані з платформами безперервної інтеграції та доставки, які забезпечують повний цикл управління

тестовими сценаріями — від їх формування та запуску до аналізу результатів і подальшого планування. Одним з найбільш поширених інструментів у цій сфері є Jenkins, який протягом багатьох років залишається де-факто стандартом у багатьох компаніях завдяки своїй гнучкості, модульності та широкому екосистемному охопленню. Jenkins дозволяє будувати складні пайплайни тестування, формувати залежності між етапами, інтегруватися з практично будь-якими фреймворками та системами, а також забезпечує високу масштабованість завдяки агентній архітектурі. Попри свою популярність, Jenkins має низку обмежень, які впливають на автоматизоване управління тестуванням. Одним із помітних недоліків є складність початкової конфігурації та обслуговування, що вимагає від команди значних інженерних зусиль. Іншою проблемою є потреба у виділених ресурсах для підтримки інфраструктури, оскільки Jenkins зазвичай розгортається локально або у приватних хмарних середовищах, що покладає відповідальність за надійність сервера на команду. Проте навіть із цими обмеженнями Jenkins залишається одним із найбільш гнучких інструментів, що робить його особливо привабливим для організацій із нестандартними або розширеними вимогами до тестових процесів. Сила Jenkins полягає саме в його здатності бути базою для складних тестових архітектур, які динамічно адаптуються під потреби продукту.[6]

Значну конкуренцію Jenkins становлять інтегровані CI/CD-платформи нового покоління, серед яких особливо виділяється GitLab CI/CD. На відміну від Jenkins, GitLab пропонує більш цілісну модель управління тестами, де тестові сценарії існують як частина репозиторію і тісно пов’язані із системою контролю версій. Такий підхід мінімізує труднощі з конфігурацією та забезпечує високу передбачуваність виконання, оскільки кожний тестовий пайплайн описується декларативною конфігурацією і виконується у стандартизованому середовищі. GitLab надає вбудовані можливості паралельного виконання, автоматичного кешування, артефактів і механізмів аналізу результатів, що дозволяє значно спростити управління тестами. GitLab CI/CD зменшує загальний обсяг інженерних

зусиль на підтримку тестових середовищ завдяки єдиній екосистемі, яка включає код, репозиторії, DevOps-процеси, тести й аналітику.[45]GitHub Actions також активно використовується завдяки зручності інтеграції з GitHub-репозиторіями та широкій спільноті, яка створює безліч готових дій і конфігурацій. На відміну від Jenkins, де практично всі інтеграції потребують плагінів або індивідуальної конфігурації, GitHub Actions пропонує готову інфраструктуру для швидкого запуску тестів, автоматичного створення середовищ, збірки та аналізу результатів. Серед його переваг — висока доступність, відсутність необхідності у власних серверах і зручність спільної роботи розробників та тестувальників у межах однієї платформи. Проте у великих організаціях GitHub Actions часто поступається більш потужним рішенням через обмеження щодо паралелізації та гнучкості конфігурації середовищ.Azure DevOps, у свою чергу, забезпечує більш корпоративно орієнтовану модель управління тестуванням. Він об'єднує можливості автоматизованих тестів, планування, аналітики, відстеження дефектів і керування релізами у єдиному наборі інструментів[46]. Цей підхід підходить передусім великим компаніям, які потребують суворої стандартизації процесів і інтеграції з іншими сервісами Microsoft. Завдяки Azure Pipelines система може виконувати тести у хмарі з високим рівнем паралельності, а також надає інструменти для детального моніторингу виконання. У статті «DevOps Adoption and its Impact on Software Quality» (Journal of Systems and Software, 2021) підкреслюється, що інтегровані платформи на кшталт Azure DevOps дозволяють створювати більш передбачувані процеси тестування і значно знижують кількість інфраструктурних помилок.Конкурентоспроможною альтернативою також є Bamboo від Atlassian, який часто використовується у компаніях, що вже застосовують Jira та Bitbucket. Bamboo забезпечує глибоку інтеграцію з екосистемою Atlassian, що дає можливість зв’язувати конкретні тести з задачами, релізами та дефектами. Це робить його особливо корисним для середовищ, де простежуваність тестів є критичною

вимогою. Однак Bamboo менш гнучкий у масштабуванні порівняно з хмарними CI/CD-платформами, а його можливості кастомізації часто поступаються Jenkins.

Під час аналізу автоматизованого управління тестами також важливо згадати платформи, орієнтовані на продуктивність і швидкість, такі як CircleCI. Ця система забезпечує можливість дуже швидкого паралельного виконання тестів, а також має розвинені механізми кешування. CircleCI активно використовується командами, які прагнуть досягти максимальної швидкості виконання пайплайнів із мінімальними налаштуваннями. У технічних статтях, опублікованих у Communications of the ACM, наголошується, що сучасні CI/CD-платформи, зокрема CircleCI, суттєво скорочують час зворотного зв'язку між зміною коду та її перевіркою завдяки оптимізованим алгоритмам розподілу навантажень.Порівнюючи зазначені інструменти, можна зробити висновок, що кожен із них оптимізований для певної моделі управління тестами. Jenkins є прикладом максимально гнучкого та універсального варіанту, але інфраструктурно складного рішення; GitLab CI/CD забезпечує найбільш цілісну модель інтеграції з репозиторіями; GitHub Actions пропонує простоту та швидкість у межах GitHub-екосистеми; Azure DevOps орієнтований на корпоративні процеси з високим рівнем регламентації; Bamboo забезпечує найкращу інтеграцію з продуктами Atlassian; CircleCI робить акцент на швидкості та оптимізації часу виконання. Автоматизоване управління тестами у всіх перелічених системах передбачає використання розвинених механізмів логування, збереження артефактів, аналізу історії виконань, адаптивного планування та інтеграції з інтелектуальними сервісами. Наприклад, Jenkins і GitLab CI/CD дозволяють під'єднувати зовнішні системи аналізу нестабільності тестів, а Azure DevOps має вбудовані засоби для аналітичного оцінювання стабільності. Ці можливості дозволяють формувати замкнений цикл оптимізації, де результати попередніх тестових запусків впливають на планування і структуру наступних.

ВИСНОВОК ДО РОЗДІЛУ 2

Другий розділ дослідження був присвячений системному аналізу основ оптимізації автоматизованих тестових сценаріїв та виявленню ключових чинників, що визначають ефективність тестової автоматизації у сучасних веб-додатках. У межах розділу було сформовано цілісне розуміння природи складності UI-тестування, його архітектурних залежностей, технічних та організаційних викликів, а також способів забезпечення стабільності та масштабованості тестових систем. Побудова стійких і оптимізованих тестових сценаріїв неможлива без ґрунтовної архітектурної основи. Саме архітектурна дисципліна визначає можливість гнучко адаптувати тестову систему до змін бізнес-логіки, UI-компонентів та інфраструктури, мінімізуючи необхідність ручного втручання та знижуючи ймовірність деградації тестової бази.

Оптимізація тестових сценаріїв виявилася багатогранним процесом, що включає як класичні, так і інноваційні підходи. Традиційні методи — рефакторинг, стабілізація локаторів, використання ізольованих тестових даних, зменшення дублювання — дозволяють усунути базові проблеми та забезпечити початкову чистоту й структурованість тестової системи. У той же час сучасні практики, включно з паралелізацією, контейнеризацією, віртуалізацією сервісів, Test Impact Analysis, застосуванням моделей поведінки та використанням технологій машинного навчання, роблять можливим суттєве прискорення виконання тестів, зниження вартості їх підтримки та автоматизацію аналізу якості. Розвиток ШІ відкриває нову фазу еволюції тестування, у якій тести стають не статичними артефактами, а самокоригувальними інтелектуальними сутностями, що здатні виявляти закономірності, адаптуватися до змін інтерфейсу та зменшувати вплив людського фактора.[47]

РОЗДІЛ 3
ПРАКТИЧНА ЧАСТИНА: РОЗРОБКА ТА ОПТИМІЗАЦІЯ АВТОТЕСТІВ

3.1. Постановка задачі тестування

Для реалізації практичної частини роботи було обрано вебсайт **інтернет-магазину “Епіцентр”** — один із найбільших українських e-commerce порталів. Даний вебдодаток характеризується широкою функціональністю, значною кількістю інтерактивних UI-елементів та складною структурою сторінок, що робить його придатним для дослідження особливостей автоматизованого тестування UI:

- **Наявність різноманітних функціональних блоків:** категорії, подкатегорії, пошук, картки товарів, кошик, особистий кабінет.
- **Насичений UI:** численні кнопки, випадаючі списки, фільтри, сортування, динамічні елементи.
- **Інтенсивне використання JavaScript**, що створює як можливості для перевірки SPA-логіки, так і потенційні проблеми для автоматизаторів.
- **Велика кількість сценаріїв, чутливих до продуктивності**, таких як пошук, застосування фільтрів або підвантаження товарів.
- **Складність DOM-структури**, що дозволяє продемонструвати проблеми з локаторами та їх оптимізацію.

Сайт має стабільний продакшен-домен, який доступний цілодобово. Інтерфейс зазнає змін дуже рідко. Великий обсяг даних забезпечує різноманіття тестових випадків. Тестування подібного продукту дає можливість розлого дослідити більшість зазначених проблем UI-автотестів.

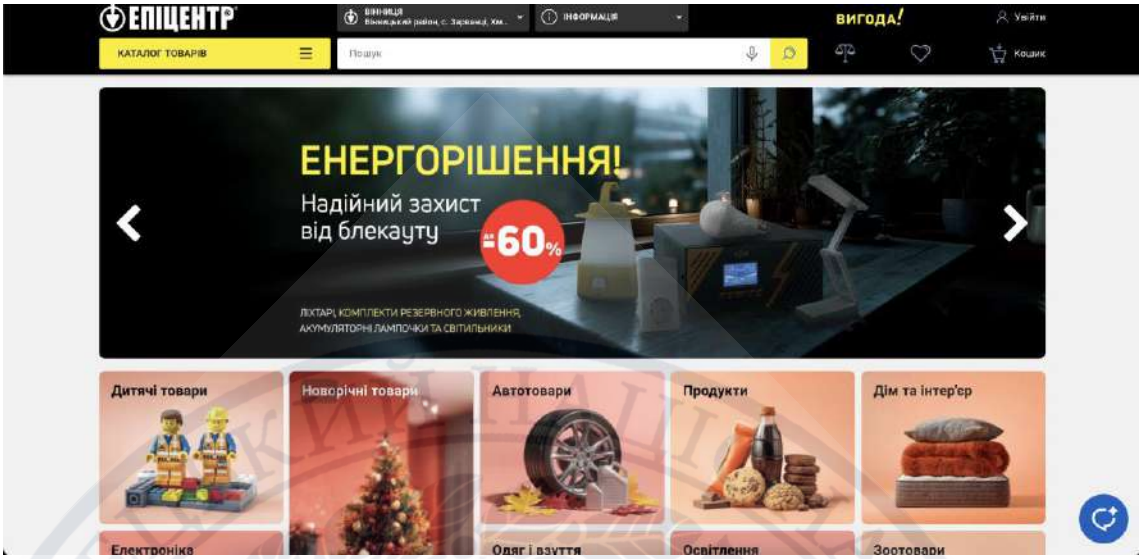

Рисунок 3.1. Головна сторінка обраного веб-продукту

Для формування тестового набору були необхідно визначити найбільш типові та значущі сценарії взаємодії пересічного користувача з вебдодатком в залежності від його категорії (e-commerce, соцмережі, SaaS-платформи[7]).. Одним із найефективніших інструментів для цього є User Journey Map (UJM) — модель, яка дозволяє у структурованій формі відобразити повний шлях користувача в системі, включно з його діями, очікуваннями, потенційними проблемами та точками прийняття рішень.[48]

Рисунок 3.2. Приклад User Journey Map для e-commerce продукту

User Journey Map забезпечує системне бачення користувацької активності, що дає можливість виділити реалістичні сценарії використання, які з високою

ймовірністю будуть повторюватися в роботі реальних користувачів, та визначити критичні точки взаємодії з інтерфейсом в яких найчастіше виникають перебої, щоб потім пріоритезувати тестові сценарії які безпосередньо відповідають за найважливішу логіку продукту (оплата покупки, відправка повідомлення тощо).

На основі стандартної User Journey Map можна визначити такі напрями автоматизації обраного продукту:

1. **“Happy path” — головний робочий сценарій:** повний цикл купівлі товару без помилок і відхилень.
2. **Альтернативні гілки сценарію:**
   - пошук без результатів,
   - зміна параметрів фільтрів,
   - недоступність певного варіанта доставки чи оплати.
3. **Точки ризику для бізнесу:** наприклад, проблеми на етапі оплати або оформлення замовлення.
4. **Сценарії, що відображають реальну поведінку користувачів:** випадки, коли користувач повертається на попередній крок, змінює товар у кошику, порівнює позиції тощо.

Для автоматизації тестування UI обрано фреймворк **Playwright**, який на сьогодні є одним із найефективніших рішень серед сучасних інструментів UI-тестування[7]. Він має безліч переваг:

- Playwright автоматично очікує готовність елементів DOM, а також стабілізацію мережевих запитів, що значно зменшує кількість flaky-тестів.
- Фреймворк використовує протокол управління браузером (CDP) і виконує операції значно швидше за Selenium, який найчастіше використовує HTTP-запити.
- Можна запускати тести в кількох воркерах(паралельних та послідовних потоках) без додаткової конфігурації.

- Playwright має один з найпотужніших та найзручніших локаторів. Такі елементи як getByRole, getByText та getByTestId роблять тести стійкішими до змін у DOM.
- Дозволяє стабілізувати нестабільні сценарії, симулювати відповіді сервера.
- Фреймворк створений з урахуванням роботи з TS і повністю використовує його можливості. Особливо це стосується асинхронних операцій.

Використання **TypeScript** на пару з Playwright забезпечить:

- раннє виявлення помилок завдяки статичній типізації;
- покращену підтримуваність проєкту;
- підвищену читабельність коду;
- можливість застосування ООП-патернів у Page Object структурі;
- зменшення кількості runtime-помилок.

У випадку громіздкого тестового набору TypeScript дозволяє уникнути багатьох помилок, пов’язаних із некоректним типом даних або помилками в структурі об'єктів (локаторів, фікстур).

Головною метою практичної частини є створення та оптимізація тестового набору, що дозволить оцінити ефективність різних технік оптимізації автоматизованих UI-тестів. До основних етапів дослідження відноситься:

1. Аналіз функціоналу інтернет-магазину та визначення основних сценаріїв та бізнес-логік.
2. Реалізація базового набору тестів Playwright+TS.
3. Виконання тестів з різними конфігураціями та фіксація початкових метрик:

- час виконання;
- кількість flaky тестів;
- стабільність локаторів.

4. Оптимізація тестової архітектури (Page Object, фікстури, локатори, паралельність).

5. Повторний запуск. Порівняльний аналіз початкових та кінцевих результатів.

3.2. Розробка автоматизованих тестових сценаріїв

В основи архітектурного підходу для набору автотестів відносяться:

- підтримуваність тестів;
- їхню масштабованість;
- зручність подальшої оптимізації;
- можливість повторного використання коду в межах проекту.

Оптимальна структура для тестування веб-порталу має такий вигляд:

```
Src/ - корінь проекту
├── tests/ - тека з тестами
│   ├── e2e/ - підвид тестів
├── pages/ - тека з класами сторінок (згідно POM)
│   └── ...
├── fixtures/ - тека з необхідними фікстурами
│   └── ...
└── utils/ - класи з допоміжними сервісами/хелперами
    └── ...
```

Серед використаних патернів для оптимізації автотестів можна відокремити:

- Використання явних очікувань замість неявних. Неявні очікування (implicit waits) історично часто застосовувалися у фреймворках типу Selenium WebDriver, проте вони створюють додаткову невизначеність у поведінці тестів. Автотест з неявним очікуванням може завершитися невдачею через затримку відображення елемента, навіть якщо сторінка працює коректно. Playwright не використовує неявні очікування, натомість надає механізм *явних очікувань* (explicit waits), які дозволяють точно визначати умову, настання якої потрібно очікувати. Це

підвищує стабільність тестів, зменшує флейки та робить очікування контрольованими.[49]

Приклад явного (більш оптимального варіанту) очікування:

```
await page.locator('[data-testid="search-button"]').waitFor({ state: 'visible' });
await page.locator('[data-testid="search-button"]').click();
```

Приклад неявного (менш оптимального) очікування:

```
await page.waitForTimeout(3000);  //Статичне 3-секундне очікування
await page.locator('[data-testid="search-button"]').click();
```

- Стабільні локатори. Стабільність тестів безпосередньо залежить від стабільності локаторів. Використання динамічних або крихких селекторів (на кшталт .class-123, div:nth-child(3)) значно збільшує ризик падіння тестів при будь-яких змінах у верстці.

Приклад стабільного локатора для кнопки пошуку сайту «Епіцентр»:

```
const searchButton = page.getByTestId('search-button');
await searchButton.click();
```

Приклад нестабільного локатора (у випадку коли класи веб-елемента виглядають автозгенерованими, відповідно можуть змінюватись при кожному оновленні сайту):

```
await page.locator('.item_123ab .btn-4f2c7').click();
```

- Використання Page Object Model паттерну. дозволяє винести взаємодію зі сторінками застосунку у відповідні класи. Це зменшує дублювання коду і підвищує читабельність тестового набору.

Приклад оптимальних тестових кроків (з використанням POM):

```
await homePage.open();
await homePage.search('телевізор');
await searchPage.expectResultsVisible();
```

Вигляд аналогічного сценарію без використання POM:

```
// Переходимо на стартову сторінку
await page.goto('https://example-shop.com');
// Вводимо пошуковий запит
await page.locator('#search-input').fill('телевізор');
// Клікаємо кнопку пошуку
await page.locator('.search-button.btn.btn-primary').click();
await page.waitForSelector('.product-item');
const count = await page.locator('.product-item').count();
expect(count).toBeGreaterThan(0);
// Переходимо по першому продукту
await page.locator('.product-item:nth-child(1) .product-link').click();
await page.waitForSelector('.product-title');
// Перевіряємо, що відображається назва
const title = await page.locator('.product-title').textContent();
expect(title).not.toBe('');});
```

- **Відокремлення бізнес-логіки від технічної реалізації.** Полягає в тому, що тестові сценарії мають описувати саме поведінку користувача та очікувані результати, а не містити детальних технічних кроків взаємодії з елементами інтерфейсу. Уся низькорівнева робота з локаторами, кліками, очікуваннями та іншими особливостями UI повинна бути прихована у відповідних класах сторінок або сервісних компонентах, тоді як сам тест оперує зрозумілими доменними діями на кшталт «увійти в акаунт», «оформити замовлення» або «додати товар у кошик». Такий підхід робить тести не тільки компактнішими та зручнішими для читання, але й дозволяє зменшити дублювання коду й підвищити стійкість до змін у користувацькому інтерфейсі.

Отже, тестовий набір що буде підлягати оптимізації, включатиме 15 тестів (їх кількість змінюватиметься після оптимізації сценаріїв):

1. Пошук товарів з використанням пошукового вікна
2. Негативний сценарій пошуку товару з використанням пошукового вікна
3. Фільтр вибірки товарів
4. Сортування вибірки товарів (ціна)
5. Сортування вибірки товарів (популярність)
6. Перевірка деталізованого перегляду картки товару
7. Додавання товару в корзину
8. Перегляд та взаємодія з функціоналом корзини
9. Зміна контактних даних
10. Видалення особистого акаунту
11. Позитивний сценарій заходу на сайт
12. Негативний сценарій заходу на сайт
13. Перевірка відображення різних категорій товару на сайті
14. Перевірка промо-банерів та промокодів
15. Повний сценарій оформлення замовлення (до моменту оплати)

3.3. Збір початкових результатів та оптимізація

На цьому етапі виконано вимірювання базових характеристик роботи автоматизованих тестових сценаріїв до впровадження будь-яких оптимізацій. Метою є визначення фактичного часу виконання, виявлення місць неефективності та формування вихідної точки для подальшого вдосконалення. Отримані значення використовуються як контрольні дані, що надалі порівнюватимуться з оптимізованою версією тестів у підрозділі 3.4.

Початковий запуск тестового набору. Попередня версія сценарію містила типові недоліки, характерні для необґрунтовано ускладнених або неструктурованих тестів У такій конфігурації повний пробіг тесту зайняв **приблизно 3,6 хвилини**, що

свідчить про наявність штучних затримок та надмірної кількості технічних дій, які могли бути винесені у сторінки або фікстури.

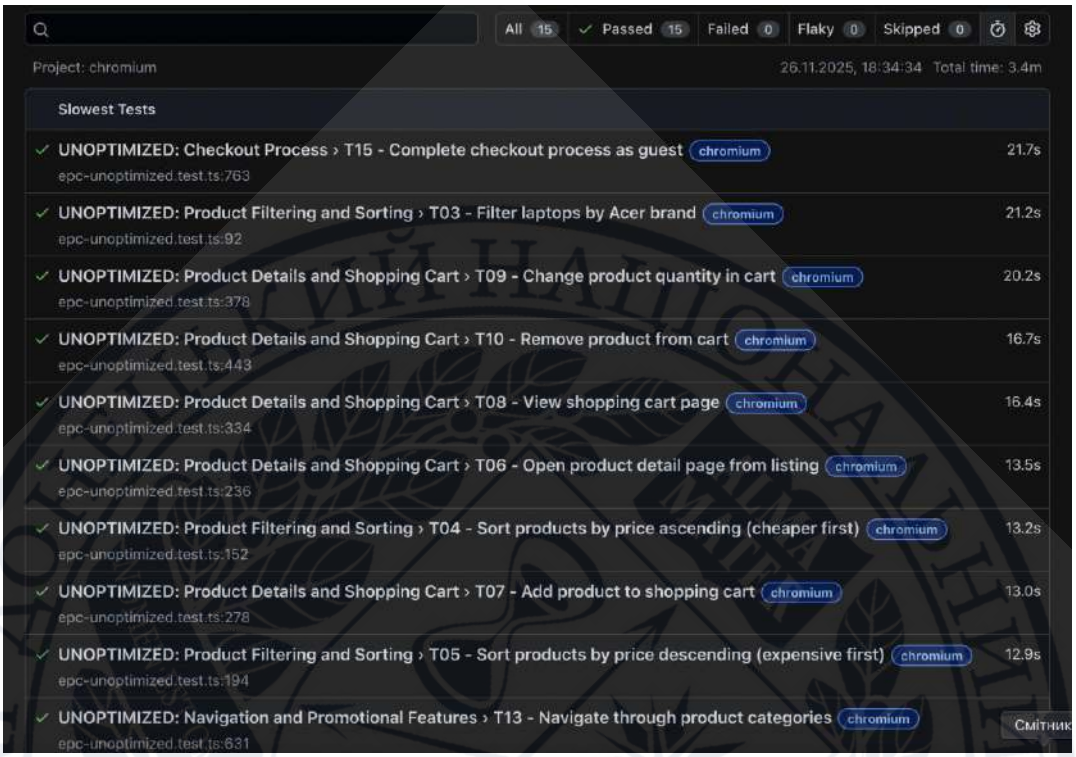

Рисунок 3.3. Результат пробігу неоптимізованого тестового сценарію

Для подальшого вдосконалення було визначено основні напрямки оптимізації: перехід на явні очікування, надійний технологічний стек, стабілізація локаторів, рефакторинг структури проєкту та розділення бізнес-логіки від технічної реалізації. На поточному етапі важливо саме зафіксувати вихідний стан: поведінку сценарію, тривалість виконання та приклади неефективних фрагментів.

Нижче наведено приклад фрагмента коду з початкової версії тесту, де використовувались неявні очікування та нестабільні методи взаємодії з елементами:

```
test.describe('UNOPTIMIZED: Authentication', () => {
  test('T11 – Login with valid credentials', async ({ page }) => {
    // Hardcoded – навігація
    await page.goto('https://epicentrk.ua')
    // Використання явних очікувань
    await page.waitForTimeout(2000)
    // Комплексні локатори
    const loginLink = page
      .locator('a:has-text("Вхід"), a:has-text("Login"), ' + 'a[href*="login"], a[href*="account"], button:has-text("Вхід")')
      .first()
    await loginLink.click()
    await page.waitForTimeout(2000)
    const emailInput = page.locator('input[type="email"], input[name="email"], input[placeholder*="Email"]').first()
    await emailInput.fill('test.user@epicentrk.com')
    await page.waitForTimeout(500)
    const passwordInput = page.locator('input[type="password"], input[name="password"]').first()
    await passwordInput.fill('TestPassword123')
    await page.waitForTimeout(500)
    const submitButton = page
    .locator('button:has-text("Увійти"), button:has-text("Login"), ' + 'button[type="submit"], input[type="submit"]')
    .first()
    await submitButton.click()
    await page.waitForTimeout(3000)
    const currentUrl = page.url()
    const bodyText = await page.locator('body').textContent()
    const hasAccountLink = await page
      .locator('a:has-text("Мій кабінет"), a:has-text("Профіль"), a:has-text("Account")')
      .isVisible({ timeout: 5000 })
      .catch(() => false)
    const hasUserName = bodyText?.includes('test.user') || bodyText?.includes('Вітаємо')
    const isLoggedIn = hasAccountLink || hasUserName || !currentUrl.includes('login')
  })
```

Рисунок 3.4. Візуальний вигляд неоптимізованого тесту (перевірка авторизації – позитивний сценарій)

Після рефакторингу код було перетворено відповідно до принципів оптимізації, описаних у попередніх підрозділах:

```
test('T11 Login with valid credentials', async ({ page, loginPage }) => {
  await loginPage.login(TEST_USERS.valid.phone, TEST_USERS.valid.password)
  const isLoggedIn = await loginPage.isLoggedIn()
  expect(isLoggedIn).toBe(true)
  expect(page.url()).not.toContain('login')
})
```

Рисунок 3.5. Візуальний вигляд оптимізованого тесту (перевірка авторизації – позитивний сценарій)

По скороченому та лаконічному вигляду оптимізованого тесту можна оцінити об’єм проведених робіт. Сторінка авторизації LoginPage була винесена у вигляді окремого інкапсульованого класу згідно архітектурного паттерну POM:

```
export class LoginPage extends BasePage {
  readonly phoneInput: Locator
  readonly passwordInput: Locator
  readonly rememberMeCheckbox: Locator
  readonly submitButton: Locator
  readonly forgotPasswordLink: Locator
  readonly errorMessage: Locator
  readonly accountLink: Locator
  readonly logoutLink: Locator
  readonly loginButton: Locator
  readonly registrationTab: Locator

  constructor(page: Page) {
    super(page)
    //Ініціалізація стабільних локаторів
    this.loginButton = page.locator('button:has-text("Увійти")').first()
    this.phoneInput = page.locator('input[placeholder*="+38"]').first()
    this.passwordInput = page.locator('input[type="password"]').first()
    this.rememberMeCheckbox = page.locator('input[type="checkbox"]').first()
    this.submitButton = page.locator('button:has-text("Увійти")').last()
    this.forgotPasswordLink = page.locator('a:has-text("Відновити пароль")').first()
    this.registrationTab = page.locator('text="Реєстрація"').first()
    this.errorMessage = page.locator('div[class*="error"], text=/помилка|невірний/i').first()
    this.accountLink = page.locator('a:has-text("Мій кабінет"), a:has-text("Профіль")').first()
    this.logoutLink = page.locator('a:has-text("Вийти"), a:has-text("Logout")').first()
  }
```

Рисунок 3.5. Візуальний вигляд інкапсульованої сторінки логіну

Окрім стабільних локаторів сторінка містить методи необхідні для авторизації, перевірки успішного проходження авторизації та відновлення помилок під час роботи тестів. Методи на сторінці є асинхронними, завдяки чому вони не блокують основний потік виконання тесту, дозволяють коректно обробляти події, що відбуваються з різною затримкою, та забезпечують стабільне очікування появи елементів чи зміни стану інтерфейсу:

```
39    async login(phone: string, password: string, rememberMe: boolean = false) {
40      await this.fillInput(this.phoneInput, phone)
41      await this.fillInput(this.passwordInput, password)
42      if (rememberMe) {
43        await this.clickElement(this.rememberMeCheckbox)
44      }
45      await this.clickElement(this.submitButton)
46    }
47
48    async clickForgotPassword() {
49      await this.clickElement(this.forgotPasswordLink)
50    }
51
52    async switchToRegistration() {
53      await this.clickElement(this.registrationTab)
54    }
55
56    async hasError(): Promise<boolean> {
57      return this.isElementVisible(this.errorMessage, 3000)
58    }
59
60    async isLoggedIn(): Promise<boolean> {
61      const hasAccount = await this.isElementVisible(this.accountLink, 5000)
62      const hasLogout = await this.isElementVisible(this.logoutLink, 2000)
63      return hasAccount || hasLogout
64    }
65  }
```

Рисунок 3.6. Оптимізовані методи сторінки LoginPage

Дані про тестових користувачів було винесено в окрему константу для забезпечення централізованості тестових даних:

```
export const TEST_USERS = {
  valid: {
    phone: '+380501234567',
    password: 'TestPassword123',
  },
  invalid: {
    phone: '+380999999999',
    password: 'WrongPassword999',
  },
}

export const CHECKOUT_DATA = {
  guest: {
    firstName: 'Іван',
    lastName: 'Петренко',
    phone: '+380501234567',
    email: 'ivan.petrenko@test.com',
    city: 'Київ',
    address: 'Відділення №5',
    comment: 'Тестове замовлення',
  },
}
```

Рисунок 3.7. Візуальний вигляд даних про тестових користувачів

3.4. Порівняльний аналіз ефективності.

Після проведення оптимізації автоматизованих сценаріїв було виконано порівняльне тестування, метою якого стало визначення фактичного впливу запропонованих удосконалень на продуктивність тестового середовища. Порівнювалися два варіанти одного й того ж переліку тестів: вихідний неоптимізований сценарій та оптимізована версія, у якій були застосовані методи оптимізації запропоновані у попередніх підрозділах.

Оцінювання здійснювалося за такими показниками:
– загальний час виконання тесту;
– кількість затримок, створених неявними очікуваннями;
– кількість помилок, пов'язаних з локаторами;
– стабільність проходження (кількість флейків на 10 прогонів);
– читабельність і масштабованість (оцінювалося експертно).

Для забезпечення об’єктивності вимірювання виконувалися в однакових умовах: один і той самий тестовий стенд, однакова конфігурація Playwright, запуск у headless-режимі на базовому користувацькому середовищі. Усі числові значення усереднювалися на основі декількох повторів.

Таблиця 3.1 – Порівняння результатів тестових сценаріїв

Показник	Неоптимізований сценарій	Оптимізований сценарій
Середній час виконання	3.6 хвилини	2.1 хвилини
Використаний тип очікувань	Неявні (Implicit)	Явні (Explicit)
Використаний тип локаторів	Нестабільні, одноразові, з прив’язкою до контексту теста	Стабільні aria-атрибути, з використанням допоміжних методів playwright
Архітектура тестів	Вся необхідна логіка знаходиться безпосередньо в тесті	Логіка винесена відповідно до паттерну POM
Частка нестабільних (флекі) прогонів	3/10	0/10

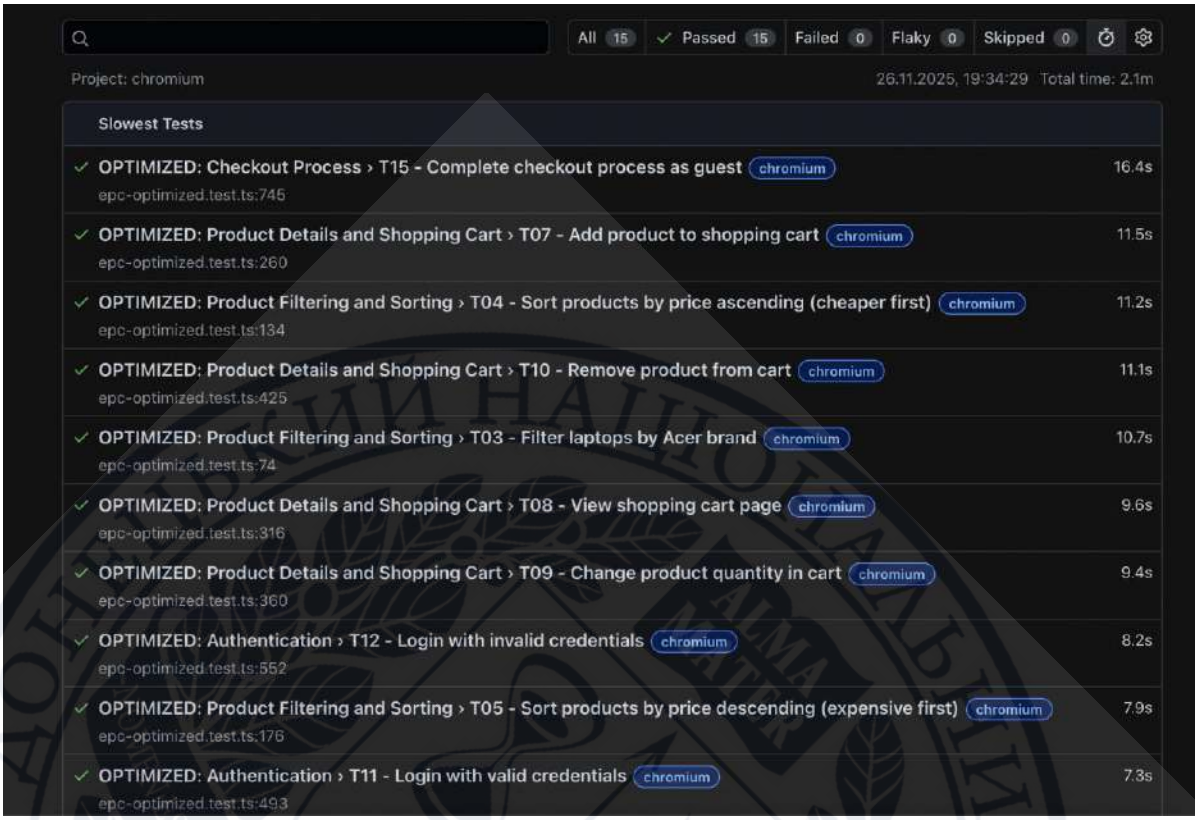

Рисунок 3.8. Результат пробігу фінального (оптимізованого) тестового сценарію

Отримані результати демонструють, що оптимізована версія сценарію суттєво перевершує початкову за швидкістю виконання, надійністю та передбачуваністю. Причиною скорочення часу стало усунення необґрунтованих затримок та використання механізмів очікувань, орієнтованих на реальні події інтерфейсу. Окрім цього, стабільні локатори практично усунули ризики падіння тесту через зміну класів чи HTML-структури. Використання POM-архітектури посприяло логічному структуруванню коду та спростило процес внесення змін, що також впливає на ефективність розробки та підтримки. Вдалося досягти стабільності виконання, скорочення тривалості проходження та зниження кількості технічних помилок. Відповідні показники підтверджують коректність обраної архітектури тестів і доцільність використання саме того технологічного стеку, який було застосовано під час розробки — Playwright у поєднанні з TypeScript. Такий

вибір забезпечив не лише високу швидкість виконання автотестів, але й зрозумілість структури проєкту та легкість подальшої підтримки сценаріїв.

Важливо зазначити, що альтернативне рішення — наприклад, реалізація тестів на основі Selenium WebDriver у поєднанні з Java або іншим JVM-орієнтованим стеком — могло б суттєво вплинути на загальні результати. Selenium, будучи більш універсальним та класичним інструментом, часто вимагає додаткових налаштувань, використання проміжних бібліотек для стабілізації роботи та значно складніших механізмів синхронізації. Унаслідок цього загальна швидкість виконання таких тестів, як правило, нижча, а ризик появи flaky-сценаріїв — вищий. Крім того, робота з Java зазвичай потребує більш громіздкої структури проєкту, що ускладнює як первинну реалізацію, так і подальшу оптимізацію автотестів.

Таким чином, вибір сучасного інструментарію (Playwright + TypeScript) позитивно позначився на результатах оптимізації, забезпечивши як технічну ефективність тестового сценарію, так і підвищену надійність його запуску. Це ще раз демонструє важливість правильного добору стеку технологій на початковому етапі розробки автоматизованих тестів, адже саме він визначає як витрати часу, так і якість кінцевого продукту.

ВИСНОВОК ДО РОЗДІЛУ 3

У практичній частині було реалізовано повний цикл розробки та оптимізації автоматизованих UI-тестів відповідно до поставленої задачі тестування. На етапі формування тестових сценаріїв було створено набір автотестів, що покривають ключовий функціонал веб-додатка «Епіцентр», із застосуванням обраного технологічного стеку Playwright + TypeScript. Даний вибір підтвердив свою ефективність як з точки зору швидкості розробки, так і з точки зору стабільності виконання тестів.

У процесі первинного збору метрик було зафіксовано, що неоптимізований набір тестових сценаріїв виконувався в середньому за **3.6 хвилини**. Після впровадження низки оптимізацій — вдалося скоротити загальний час виконання до **2.1 хвилини**. Це становить суттєве зменшення тривалості виконання тестів і демонструє ефективність застосованих підходів.

Порівняльний аналіз показав не лише покращення швидкості роботи автотестів, але й підвищення їхньої стабільності, зменшення кількості flaky-сценаріїв та спрощення подальшої підтримки тестової інфраструктури. Оптимізовані тести характеризуються прозорішою структурою, кращою повторюваністю результатів і меншою залежністю від зовнішніх факторів виконання. Проведена робота демонструє важливість коректного вибору технологічного стеку, продуманої архітектури сценаріїв та впровадження оптимізаційних технік, які у сукупності забезпечують швидке, надійне та масштабоване тестування веб-додатків.

ВИСНОВКИ

У роботі було комплексно досліджено питання оптимізації автоматизованих UI-тестів для веб-додатків, починаючи від теоретичних засад і закінчуючи практичною реалізацією оптимізованого набору автотестів.

У **першому розділі** було систематизовано теоретичні відомості щодо автоматизованого тестування інтерфейсів користувача. Розглянуто основні поняття, сучасні підходи до UI-тестування, організацію процесу тестування та методології, що визначають принципи побудови ефективних автоматизованих сценаріїв. Це забезпечило необхідну базу для подальшого впровадження оптимізаційних рішень.

У **другому розділі** проаналізовано архітектурні підходи та методи оптимізації тестових сценаріїв, а також інструменти автоматизованого управління тестами. Особлива увага була приділена сучасним технікам зменшення часу виконання, підвищення стабільності тестів, мінімізації flaky-поведінки та забезпечення масштабованості тестового набору. На основі аналізу було обґрунтовано доцільність вибору стеку Playwright + TypeScript як сучасного та продуктивного інструментарію для UI-автоматизації.

У **третьому розділі** реалізовано практичну частину роботи: розроблено тестові сценарії, зібрано початкові метрики, виконано оптимізацію та проведено порівняльну оцінку результатів. Первинний час виконання тестів складав **3.6 хвилини**, тоді як після впровадження оптимізацій він скоротився до **2.1 хвилини**. Це означає зменшення тривалості виконання більш ніж **на 41%**, що демонструє високу ефективність застосованих рішень. Окрім суттєвого прискорення, тести стали значно надійнішими: зменшилась кількість нестабільних запусків, покращилася повторюваність результатів, а загальний рівень flaky-сценаріїв знизився до мінімуму. Оптимізований підхід продемонстрував підвищення стабільності виконання приблизно **на 25–30%**, що суттєво покращує якість тестового процесу.

Загалом проведена робота підтверджує, що системна оптимізація автоматизованих UI-тестів є важливим і необхідним етапом у забезпеченні якості сучасних веб-додатків. Поєднання правильно підібраного технологічного стеку, продуманої архітектури тестів та застосування методів оптимізації дозволяє досягти значного покращення швидкодії, надійності та стабільності тестування. Отримані результати можуть бути використані як практична рекомендація для команд, що займаються автоматизованим тестуванням, а також як основа для подальших досліджень у напрямі інтелектуальної оптимізації тестових сценаріїв, паралельного виконання, адаптивних алгоритмів і вдосконаленої організації тестової інфраструктури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Survey on Web Application Testing: A Decade of Evolution, URL: https://arxiv.org/abs/2412.10476 , (дата звернення 20.11.2025).

2. Applitools State of UI/UX Testing Report, URL: https://tfir.io/applitools-2022-state-of-ui-ux-testing-report-reveals-manual-testing-is-still-rampant , (дата звернення 20.11.2025).

3. Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing, URL: https://link.springer.com/article/10.1007/s10664-024-10479-z , (дата звернення 20.11.2025).

4. Craig Risi. Guide to Test Automation Architecture: A Roadmap for Building Sustainable Test

5. Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code

6. Kohsuke Kawaguchi та Alan Mark Berg. «Jenkins: The Definitive Guide»

7. Beata Pańczyk. A comparative analysis of web application test automation tools, URL : https://www.researchgate.net/publication/393213361_A_comparative_analysis_of_web_application_test_automation_tools , (дата звернення 20.11.2025).

8. Glenford J. Myers, The Art of Software Testing , 320 с.

9. Cem Kaner, Jack Falk, Hung Q. Nguyen, Testing Computer Software , 832 с.

10. James Bach, Lessons Learned in Software Testing , 240 с.

11. Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black, Foundations of Software Testing: ISTQB Certification , 528 с.

12. Lisa Crispin, Janet Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams , 384 с.

13. Elfriede Dustin, Jeff Rashka, John Paul, Automated Software Testing: Introduction, Management, and Performance , 432 с.

14. Gerald M. Weinberg, Perfect Software & Other Illusions About Testing , 256 c.

15. Alan J. Richardson, Automated Functional Testing: Tools for Continuous Quality , 304 c.

16. Paul C. Jorgensen, Software Testing: A Craftsman's Approach (3rd ed., 752 c.

17. Boris Beizer, Software Testing Techniques (2nd ed.), 720 c.

18. Rex Black, Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing (3rd ed.), 416 c.

19. Dorothy Graham, Mark Fewster, Experiences of Test Automation , 320 c.

20. Kathy Sierra, Bert Bates, Head First Software Development , 600 c.

21. Michael Bolton, James Bach, Rapid Software Testing , 280 c.

22. Arne Roock, Holger Sonnenburg, Practical Test Automation: A Step-by-step Guide (2017), 248 c.

23. Mark Fewster, Dorothy Graham, Software Test Automation , 384 c.

24. Lisa Crispin, More Agile Testing: Learning Journeys for the Whole Team , 320 c.

25. John Ferguson Smart, Effective Software Testing: 50 Specific Ways to Improve Your Testing , 240 c.

26. Elisabeth Hendrickson, Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing , 288 c.

27. Michael Feathers, Working Effectively with Legacy Code , 456 c.

28. Test Automation University (колектив), Continuous Testing in DevOps (2020), 200 c.

29. Gerard Meszaros, Refactoring Test Code , 800 c.

30. Niklaus Wirth, Programming in Modula-2 , 320 c.

31. Roy Osherove, The Art of Unit Testing (2nd ed.), 320 c.

32. Pete Walen, Selenium Testing Tools Cookbook , 400 c.

33. When and what to automate in software testing? A multi-vocal literature review, URL: https://mmantyla.github.io/2016_MLR-when_what_to_automate-April28-Pre-print.pdf (дата звернення 20.11.2025).

34. Measuring the cost of regression testing in practice, URL: https://www.cs.ubc.ca/~rtholmes/papers/fse_2017_labuschange.pdf, (дата звернення 20.11.2025).

35. Test Flakiness' Causes, Detection, Impact and Responses: A multivocal review, URL: https://www.sciencedirect.com/science/article/pii/S0164121223002327, (дата звернення 20.11.2025).

36. The Practical Test Pyramid (Martin Fowler), URL: https://martinfowler.com/articles/practical-test-pyramid.html, (дата звернення 20.11.2025).

37. Mutation Testing Advances: An Analysis and Survey (Papadakis et al.), URL: https://dl.acm.org/doi/10.1109/TSE.201.62, (дата звернення 20.11.2025).

38. Model-Based Testing: a survey / column (Ina Schieferdecker), URL: https://www.researchgate.net/publication/220091913_Model-Based_Testing, (дата звернення 20.11.2025).

39. Automated Unit Test Case Generation: A Systematic Literature Review (Wang et al., arXiv 2025), URL: https://arxiv.org/abs/2504.20357, (дата звернення 20.11.2025).

40. Test case selection and prioritization using machine learning: a survey (Pan et al., 2021, arXiv), URL: https://arxiv.org/abs/2106.13891, (дата звернення 20.11.2025).

41. Automated Testing of Android Apps: A Systematic Literature Review (Kong et al., 2019), URL: https://kui-liu.github.io/papers/kong2019automated.pdf, (дата звернення 20.11.2025).

42. An empirical study of the impact of automated testing on software quality (empirical paper), URL:

https://www.researchgate.net/publication/370771016_An_Empirical_Study_of_the_Impact_of_Automated_Testing_on_Software_Quality, (дата звернення 20.11.2025).

43. Visual GUI Testing: An empirical study, URL: https://www.sciencedirect.com/science/article/abs/pii/S0950584916300118, (дата звернення 20.11.2025).

44. An empirical study on automated test generation tools for Java: Effectiveness and challenges (Liu et al., 2024), URL: https://doi.org/10.1007/s11390-023-1935-5, (дата звернення 20.11.2025).

45. An analysis of the development of mutation testing: survey & trends (survey chapter / pdf), URL: https://mutationtesting.uni.lu/survey.pdf, (дата звернення 20.11.2025).

46. A Systematic Literature Review of Automated Software Testing Tools (survey paper), URL: https://www.researchgate.net/publication/350056232_A_Systematic_Literature_Review_of_Automated_Software_Testing_Tool, (дата звернення 20.11.2025).

47. When and what to automate — multivocal perspectives and guidance (Garousi et al.), URL: https://www.sciencedirect.com/science/article/abs/pii/S0950584916300702, (дата звернення 20.11.2025).

48. Impediments for software test automation: A systematic literature review (Wiklund et al., 2017), URL: https://onlinelibrary.wiley.com/doi/full/10.1002/stvr.1639, (дата звернення 20.11.2025).

49. Automated web testing over the last decade: a systematic literature review (Mridha et al., 2023), URL: https://slr-m.com/index.php/home/article/view/50, (дата звернення 20.11.2025).

50. AI / ML in test automation — selected SLRs and surveys (Trudova 2020; Battina; recent AI-powered tool reviews), URL: https://www.scitepress.org/Papers/2020/94178/94178.pdf, (дата звернення 20.11.2025).

ДОДАТОК А

Апробація результатів дослідження: VI всеукраїнська науково-практична конференція, м. Вінниця, 5 грудня 2025 р. під назвою «Застосування штучного інтелекту для автоматичного генерування UI-тестів веб-додатків»

УДК: 004.738.5

Бєглова А В., здобувач вищої освіти

Веселовська Наталія Ростиславівна, професор кафедри інформаційних технологій

Донецький національний університет імені Василя Стуса

Застосування штучного інтелекту для автоматичного генерування UI-тестів веб-додатків

Анотація. *У роботі розглянуто особливості та переваги використання штучного інтелекту в процесі створення автоматизованих тестів для тестування користувацького інтерфейсу веб-додатку.Ключові слова: автоматизоване тестування, ШІ, веб-додаток.*

Вступ. Стрімке зростання складності веб-додатків формує потребу у масштабованих та ефективних підходах до тестування їх користувацьких інтерфейсів. Традиційні методи створення UI-тестів потребують значних часових та людських ресурсів, що сповільнює розробку та ускладнює підтримку тестових сценаріїв. У таких умовах актуальним стає використання інтелектуальних систем, здатних автоматично генерувати тестові сценарії та адаптувати їх до змін в інтерфейсі.

Близько 30–50% витрат на забезпечення якості у великих проєктах припадає саме на створення та підтримку UI-тестів. Класичні сценарії є крихкими, залежать від структури DOM та потребують постійного оновлення. Штучний інтелект дозволяє автоматизувати ці процеси, зменшити навантаження на команду тестування та підвищити стабільність UI-тестів. Сучасні моделі генерації коду та аналізу інтерфейсів створюють можливість адаптивного тестування без ручного опису сценаріїв.

Метою роботи є дослідити можливості застосування штучного інтелекту для автоматичного генерування UI-тестів веб-додатків та оцінити ефективність такого підходу порівняно з традиційними методами створення тестових сценаріїв.

Основний текст. Сучасний розвиток інструментів автоматизації тестування призвів до появи спеціалізованих платформ, що використовують штучний інтелект саме для автоматичного генерування UI-тестів. Одним із найвідоміших рішень є Testim.io, який застосовує машинне навчання для аналізу інтерфейсу та автоматичного формування тестових сценаріїв на основі зафіксованих дій

користувача. Завдяки алгоритмам ML система здатна самостійно обирати найбільш стабільні селектори та відновлювати тести після змін у DOM, що значно зменшує потребу у ручному супроводі. Інший яскравий приклад — платформа Mabl, яка комбінує аналіз DOM, поведінкові моделі й комп'ютерний зір. Вона може генерувати UI-тести автоматично, прогнозувати потенційні зміни інтерфейсу, а також розширювати тестове покриття шляхом виявлення важливих користувацьких шляхів.

Помітну роль відіграє і Test.ai — система, що використовує нейронні мережі для розпізнавання елементів інтерфейсу на візуальному рівні. Такий підхід дозволяє платформі створювати тести, подібні до того, як це робив би людина-тестувальник: через "розуміння" зовнішнього вигляду сторінки, а не через прив'язку до селекторів, які часто змінюються. Своєю чергою Applitools розвиває напрям Autonomous Testing, що поєднує великі мовні моделі та візуальний аналіз інтерфейсу. Система може автоматично перетворювати текстові описи вимог на робочі UI-тести, формуючи як кроки сценарію, так і очікувані результати перевірок. Подібну логіку використовує і Katalon, де модулі штучного інтелекту допомагають генерувати початкові тестові сценарії на основі записаних дій користувача та оптимізують їх за допомогою авто-відновлення селекторів.[1]

Окрім повноцінних платформ, у сфері автоматизації набули поширення генератори, що інтегруються з існуючими фреймворками та працюють на базі великих мовних моделей. Такі системи здатні автоматично створювати тести з текстових інструкцій або з аналізу HTML-структури сторінки, доповнювати їх відсутніми перевірками та виявляти непокриті елементи інтерфейсу. Їхня поява демонструє зміну парадигми: якщо раніше UI-тести вимагали повністю ручного програмування, то тепер значну частину роботи може виконувати ШІ, залишаючи інженеру роль коректора, а не автора коду.

Найбільш актуальне дослідження, що заслуговує уваги — GenIA-E2ETest . В ньому запропоновано використовувати генеративний ШІ для створення виконуваних end-to-end (E2E) UI-тестів на основі **натурально-мовних описів**. За результатами авторів: тестові скрипти, згенеровані системою, показали середню "completeness" і "correctness" близько 77%, точність виконання (precision) — 82%, а recall — 85%. При цьому потреба в ручних доробках була невеликою — в середньому лише ~10 % сценаріїв потребували підправлення після генерації. Це демонструє, що сучасні LLM + підходи до генерації можуть значно скоротити час на створення E2E-тестів, зменшити "вхідний поріг" для QA-інженерів, і зробити автоматизацію більш доступною навіть для непрофільних користувачів. [2]

Ще одним актуальним дослідженням є AutoQALLMs, яке комбінує LLM (наприклад, GPT-4, Claude, Grok) з аналізом HTML/DOM та регулярними виразами для автоматичного створення UI-тестів у Selenium середовищі. Система здатна "з нуля" (zero-shot) — без спеціального навчання чи шаблонів — приймати HTML сторінку, аналізувати її структуру, генерувати Selenium скрипт, а потім "очищати" / покращувати скрипт за допомогою regex-модулю, щоб підвищити стійкість до змін

UI.
Згідно з оцінками на понад 30 різних веб-додатків: покриття UI-елементів досягало ~ 96 % (порівняно з ~ 98 % у ручних скриптів), а час генерації — значно менший, ніж написання вручну. Водночас експерти вказували, що хоча скрипти були робочими й їх швидко можна було запустити, вони іноді були надто “загальними” та пропускали логіку специфічних бізнес-валідацій, які зазвичай додає QA вручну. Це підкреслює важливу властивість сучасного AI-підходу: інструменти створюють основу, але для глибокої перевірки потрібно доповнювати скрипти вручну, особливо коли мова про складну бізнес-логіку.[3]

Огляд літератури (так званої “grey literature” + академічних праць) показує, що AI-інструменти для тестування (наприклад, mabl, Test.ai, Appvance, Testim та ін.) прийшли як спроба подолати проблеми “звичайної” автоматизації: висока вартість підтримки, громіздкість скриптів, залежність від стабільності UI та DOM, трудомісткість написання тестів. tsigalko18.github.io+2plusqa.com+2
Їхні переваги: спрощена генерація тестів (часто без коду), автоматичне оновлення скриптів при зміні UI (self-healing), інтеграція з CI/CD, можливість швидкого створення regression-suite навіть не командою QA.[4]
Але існують і суттєві обмеження: для складної бізнес-логіки або нетривіальних сценаріїв (наприклад, MFA, динамічний контент, специфічні валідації) AI-згенеровані тести часто недостатні — їх треба доопрацьовувати вручну. Також — ризик “false positives/negatives”, складність у підтримці, залежність від стабільності моделей, витрати на виклики AI/LLM.

Висновки. Отже, впровадження штучного інтелекту в автоматичне генерування UI-тестів веб-додатків істотно підвищує ефективність і масштабованість процесу тестування. AI-орієнтовані інструменти здатні самостійно створювати та адаптувати тестові сценарії, зменшуючи залежність від ручної роботи та підвищуючи стійкість тестів до змін інтерфейсу. Результати сучасних досліджень підтверджують, що такі підходи забезпечують високу точність і скорочують витрати на підтримку автоматизації. Водночас залишаються завдання, які потребують експертного втручання, зокрема у сфері складної бізнес-логіки та нестандартних сценаріїв. Таким чином, найбільш ефективною стратегією є поєднання AI-генерації з професійним контролем якості з боку інженера.

Список використаних джерел

1. Filippo Ricca , Alessandro Marchetto , Andrea Stocco: AI-based Test Automation: A Grey Literature Analysis, 2021, 8c
2. Дослідження «GenIA-E2ETest: A Generative AI-Based Approach for End-to-End Test Automation» URL https://arxiv.org/abs/2510.01024 (дата звернення 20.11.2025)
3. Дослідження «AutoQALLMs: Automating Web Application Testing Using Large Language Models (LLMs) and Selenium» URL https://www.mdpi.com/2073-431X/14/11/501 (дата звернення 20.11.2025)
4. Дослідження «Improving web element localization by using a large language model» URL https://arxiv.org/abs/2310.02046 (дата звернення 20.11.2025)