

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методи ручного тестування веб-додатків у процесі
забезпечення якості програмного забезпечення»

на здобуття освітнього ступеня бакалавра
зі спеціальності 126 – Інформаційні системи та технології
освітньо-професійної програми Інформаційні системи та технології

*Кваліфікаційна робота містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання на
відповідне джерело*

_____ Владислав ОНИЩЕНКО

Виконав: здобувач вищої освіти гр. ІСД-41
Владислав ОНИЩЕНКО

Керівник: Ольга ЖИДКА
*науковий ступінь, доктор філософії
вчене звання*

Рецензент:
*науковий ступінь,
вчене звання*

Київ 2026

6. Дата видачі завдання « » 2026 р.

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 127 стор., 14 рис., 3 табл., 47 джерел.

Мета роботи – дослідження методів ручного тестування веб-додатків та формування практичних рекомендацій щодо їх використання у процесі забезпечення якості програмного забезпечення.

Об'єкт дослідження – процес забезпечення якості веб-додатків.

Предмет дослідження – методи ручного тестування веб-додатків та особливості їх застосування під час перевірки якості програмного забезпечення.

Короткий зміст роботи:

У роботі розглянуто теоретичні основи тестування програмного забезпечення та роль ручного тестування у процесі забезпечення якості веб-додатків. Проаналізовано основні види та рівні тестування програмного забезпечення, сучасні підходи до організації процесу тестування та особливості ручної перевірки веб-систем. Досліджено архітектурні особливості веб-додатків, методи та техніки ручного тестування, типові дефекти веб-систем і способи їх виявлення. Розглянуто інструменти та тестову документацію, що використовуються у процесі ручного тестування веб-додатків.

У практичній частині роботи сформовано рекомендації щодо підвищення ефективності ручного тестування веб-додатків, описано алгоритм проведення тестування та наведено приклади практичного застосування методів перевірки веб-систем. Також виконано аналіз процесу тестування демонстраційного або існуючого веб-додатка, розглянуто приклади тестових сценаріїв та оформлення звітів про дефекти. Окрему увагу приділено економічним аспектам виконання роботи та питанням охорони праці й безпеки під час роботи з комп'ютерними системами.

КЛЮЧОВІ СЛОВА: ручне тестування, веб-додаток, якість, програмне забезпечення, вебсайт.

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій**

**ПОДАННЯ
ГОЛОВІ ДЕРЖАВНОЇ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня бакалавр**

Направляється здобувач Онищенко В.О. до захисту кваліфікаційної роботи
(прізвище та ініціали)

за спеціальністю 126 Інформаційні системи та технології
(код, найменування спеціальності)

освітньо-професійної програми Інформаційні системи та технології
(назва)

на тему: Методи ручного тестування веб-додатків у процесі забезпечення якості програмного забезпечення

Кваліфікаційна робота і рецензія додаються.

Директор ННІ ІТ _____ Катерина НЕСТЕРЕНКО
(підпис) (ім'я, ПРІЗВИЩЕ)

Висновок керівника кваліфікаційної роботи

Здобувач Онищенко Владислав Олегович продемонстрував належний рівень теоретичної підготовки, володіння сучасними інформаційними технологіями та навички самостійної роботи з науковими й навчальними джерелами. У процесі виконання кваліфікаційної роботи проявив відповідальність, аналітичне мислення та здатність до вирішення практичних завдань у сфері забезпечення якості програмного забезпечення.

Виконана кваліфікаційна робота заслуговує на оцінку «відмінно», а здобувач Онищенко Владислав Олегович – на присвоєння кваліфікації «бакалавр з інформаційних систем та технологій».

Керівник кваліфікаційної роботи _____ Ольга ЖИДКА
(підпис) (ім'я, ПРІЗВИЩЕ)

« ____ » _____ 2026 року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач Онищенко В.О. допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедрою _____ ІСТ _____ Каміла СТОРЧАК
(назва) (підпис) (ім'я, ПРІЗВИЩЕ)

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну роботу
на здобуття освітнього ступеня бакалавра

здобувача вищої освіти **Онищенко Владислава Олеговича**
на тему: **«Методи ручного тестування веб-додатків у процесі**
забезпечення якості програмного забезпечення»

Актуальність.

Тема роботи є актуальною через необхідність забезпечення високої якості сучасних веб-додатків та вдосконалення процесів ручного тестування програмного забезпечення.

Позитивні сторони.

Позитивною стороною роботи є поєднання теоретичного аналізу з практичною реалізацією тестування веб-додатка. У роботі сформовано алгоритм ручного тестування, наведено приклади виявлення дефектів та розроблено практичні рекомендації щодо підвищення ефективності процесу перевірки програмного забезпечення.

Недоліки.

До недоліків роботи можна віднести обмежений аналіз автоматизованого тестування та недостатньо детальне порівняння різних підходів до перевірки якості веб-додатків.

Висновок: *кваліфікаційна робота на здобуття ступеня бакалавра заслуговує оцінку «відмінно», а здобувач Онищенко Владислав Олегович заслуговує присвоєння кваліфікації: “бакалавр з інформаційних систем та технологій”.*

Рецензент: _____

ЗМІСТ

ВСТУП.....	5
1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА РУЧНОГО ТЕСТУВАННЯ	8
1.1 Поняття якості програмного забезпечення та роль тестування	8
1.2 Основні види та рівні тестування програмного забезпечення	26
1.3 Ручне тестування як складова забезпечення якості програмного забезпечення	38
1.4 Аналіз сучасних досліджень і підходів у сфері ручного тестування веб-додатків	45
2 ОСОБЛИВОСТІ РУЧНОГО ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ	53
2.1 Архітектурні та функціональні особливості веб-додатків	53
2.2 Основні напрями ручного тестування веб-додатків	60
2.3 Методи та техніки ручного тестування веб-додатків	66
2.4 Типові дефекти веб-додатків та способи їх виявлення	70
2.5 Інструменти та документація у процесі ручного тестування	76
3 РОЗРОБКА РЕКОМЕНДАЦІЙ ЩОДО РУЧНОГО ТЕСТУВАННЯ ВЕБ- ДОДАТКІВ	81
3.1 Формування алгоритму ручного тестування веб-додатків	81
3.2 Практична реалізація тестування демонстраційного веб-додатка	89
3.3 Формування практичних рекомендацій щодо підвищення ефективності ручного тестування	102
3.4 Оцінка ефективності запропонованих рекомендацій	106

4 ЕКОНОМІЧНЕ ОБГРУНТУВАННЯ ЕФЕКТИВНОСТІ РУЧНОГО ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ	110
4.1 Аналіз витрат на організацію процесу ручного тестування	110
4.2 Оцінка економічного ефекту від своєчасного виявлення дефектів ...	113
4.3 Оцінка ефективності запропонованого підходу до ручного тестування	117
5 ЗАХОДИ З ОХОРОНИ ПРАЦІ	120
5.1 Оцінка умов праці під час ручного тестування веб-додатків	120
5.2 Заходи щодо забезпечення безпечних умов праці	121
ВИСНОВКИ	125
ПЕРЕЛІК ЛІТЕРАТУРНИХ ДЖЕРЕЛ	128
ДОДАТОК А ВИХІДНИЙ КОД РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	134
ДОДАТОК Б ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (ПРЕЗЕНТАЦІЯ)	174

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням кількості веб-додатків, які використовуються практично в усіх сферах діяльності людини. Веб-системи застосовуються у банківській сфері, електронній комерції, освіті, медицині, державному управлінні, соціальних мережах та багатьох інших напрямках. Постійне збільшення складності програмних продуктів, розширення їх функціональних можливостей та необхідність забезпечення стабільної роботи для великої кількості користувачів створюють підвищені вимоги до якості програмного забезпечення.

Якість веб-додатків безпосередньо впливає на ефективність роботи організацій, рівень довіри користувачів та конкурентоспроможність програмних продуктів. Наявність помилок у роботі веб-систем може призводити до фінансових втрат, погіршення репутації компаній, втрати даних або порушення роботи окремих сервісів. Саме тому процес забезпечення якості програмного забезпечення займає важливе місце у сучасній розробці інформаційних систем.

Одним із основних засобів забезпечення якості програмного забезпечення є тестування. Воно дозволяє своєчасно виявляти дефекти, перевіряти відповідність програмного продукту встановленим вимогам та оцінювати стабільність роботи системи в різних умовах використання. Незважаючи на активний розвиток автоматизованих засобів перевірки програмного забезпечення, ручне тестування залишається важливою складовою процесу контролю якості веб-додатків. У багатьох випадках саме ручне тестування дозволяє оцінити зручність використання системи, коректність роботи інтерфейсу, логіку взаємодії користувача з веб-додатком та виявити помилки, які складно визначити за допомогою автоматизованих засобів.

Ручне тестування широко застосовується під час перевірки нових функціональних можливостей, аналізу поведінки веб-додатків у нестандартних ситуаціях, тестування користувацьких інтерфейсів та оцінювання загальної якості

взаємодії користувача із системою. Особливо актуальним ручне тестування є для веб-додатків, оскільки вони повинні коректно працювати на різних пристроях, у різних браузерях та за різних умов використання. Значна кількість можливих сценаріїв взаємодії з веб-системою потребує системного підходу до організації процесу тестування та використання різноманітних методів перевірки програмного забезпечення.

Водночас ручне тестування має низку особливостей та складнощів. Значний обсяг перевірок, необхідність постійного повторення тестових сценаріїв, залежність результатів від досвіду тестувальника та людського фактору можуть негативно впливати на ефективність процесу забезпечення якості. У зв'язку з цим виникає потреба у дослідженні сучасних методів ручного тестування веб-додатків, аналізі особливостей їх застосування та формуванні рекомендацій щодо підвищення ефективності процесу тестування.

Дослідження методів ручного тестування веб-додатків має важливе практичне значення, оскільки дозволяє систематизувати підходи до організації процесу перевірки програмного забезпечення та визначити найбільш доцільні способи виявлення дефектів у веб-системах. Аналіз особливостей тестування веб-додатків також сприяє підвищенню якості програмних продуктів, покращенню взаємодії між учасниками процесу розробки та зменшенню ризику виникнення помилок під час експлуатації програмного забезпечення.

Актуальність теми полягає у необхідності забезпечення високої якості сучасних веб-додатків та вдосконалення процесів ручного тестування програмного забезпечення в умовах постійного зростання складності веб-систем. Незважаючи на розвиток автоматизованого тестування, ручна перевірка залишається важливою частиною процесу забезпечення якості програмного забезпечення, особливо під час аналізу користувацького інтерфейсу, оцінювання зручності використання та перевірки нестандартних сценаріїв роботи веб-додатків.

Об'єктом дослідження є процес забезпечення якості веб-додатків.

Предметом дослідження є методи ручного тестування веб-додатків та особливості їх застосування у процесі перевірки якості програмного забезпечення.

Метою роботи є дослідження методів ручного тестування веб-додатків та формування практичних рекомендацій щодо їх використання у процесі забезпечення якості програмного забезпечення.

Для досягнення поставленої мети у роботі необхідно вирішити низку завдань, серед яких аналіз теоретичних основ тестування програмного забезпечення, дослідження особливостей ручного тестування веб-додатків, аналіз сучасних методів та технік тестування, визначення типових дефектів веб-систем, а також формування рекомендацій щодо підвищення ефективності процесу ручного тестування.

У процесі виконання роботи використовуються методи аналізу наукової літератури, порівняння сучасних підходів до тестування програмного забезпечення, узагальнення практичних методів ручного тестування веб-додатків та аналізу процесів забезпечення якості програмного забезпечення.

Практичне значення роботи полягає у можливості використання сформованих рекомендацій під час організації процесу ручного тестування веб-додатків у навчальних або практичних проєктах. Отримані результати можуть бути використані для покращення процесів перевірки якості веб-систем, підвищення ефективності виявлення дефектів та вдосконалення підходів до створення тестової документації.

Структура роботи складається зі вступу, трьох основних розділів, економічної частини, розділу з охорони праці та безпеки в надзвичайних ситуаціях, висновків, списку використаних джерел і додатків. У першому розділі розглядаються теоретичні основи тестування програмного забезпечення та ручного тестування. Другий розділ присвячений особливостям ручного тестування веб-додатків, методам перевірки веб-систем та аналізу типових дефектів. У третьому розділі формуються практичні рекомендації щодо організації процесу ручного тестування веб-додатків та розглядаються приклади практичного застосування методів тестування.

1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА РУЧНОГО ТЕСТУВАННЯ

1.1 Поняття якості програмного забезпечення та роль тестування

Розвиток інформаційних технологій суттєво змінив підходи до створення та використання програмного забезпечення. Сучасні програмні продукти виконують значну кількість функцій, забезпечують автоматизацію різноманітних процесів та використовуються як у професійній діяльності, так і в повсякденному житті людей. Від стабільності та коректності роботи програмного забезпечення залежить ефективність роботи підприємств, доступність інформаційних сервісів, безпека даних користувачів та зручність взаємодії з цифровими системами.

Поступове збільшення складності програмних продуктів призвело до появи великої кількості взаємопов'язаних компонентів, кожен із яких повинен працювати узгоджено та без помилок. Навіть незначні недоліки у роботі окремих елементів системи можуть негативно впливати на функціонування всього програмного продукту. Особливо це стосується веб-додатків, де користувачі постійно взаємодіють з інтерфейсом, формами введення даних, системами авторизації, механізмами обробки інформації та іншими складовими веб-систем. У таких умовах вимоги до якості програмного забезпечення постійно зростають, а забезпечення стабільної роботи програмних продуктів стає одним із найважливіших завдань процесу розробки.

Використання програмного забезпечення у фінансовій сфері, електронній комерції, транспорті, медицині, освіті та державних інформаційних системах значно підвищує відповідальність розробників за якість створюваних програмних продуктів. Помилки у роботі програмного забезпечення можуть призводити до втрати даних, порушення роботи сервісів, фінансових збитків або створення незручностей для користувачів. У деяких випадках навіть незначні програмні дефекти здатні суттєво впливати на репутацію компаній та рівень довіри до цифрових сервісів.

Якість програмного забезпечення формується не лише на етапі написання програмного коду. Вона залежить від багатьох чинників, серед яких правильність постановки вимог, організація процесу розробки, взаємодія між учасниками проєкту, контроль виконання поставлених завдань та своєчасне виявлення помилок. У сучасній розробці програмного забезпечення особлива увага приділяється саме процесам перевірки та контролю якості, оскільки вони дозволяють зменшити ризик виникнення проблем під час експлуатації програмних продуктів.

Необхідність забезпечення стабільної роботи програмного забезпечення сприяла розвитку різноманітних методів перевірки програмних систем. Залежно від типу програмного продукту, складності функціоналу та особливостей його використання можуть застосовуватись різні підходи до тестування. Для веб-додатків важливого значення набуває перевірка взаємодії користувача із системою, правильності роботи інтерфейсу, коректності обробки введених даних та сумісності програмного забезпечення з різними браузерами й пристроями.

Сучасні користувачі очікують від програмного забезпечення швидкої роботи, зрозумілого інтерфейсу та відсутності помилок під час виконання повсякденних дій. Умови високої конкуренції між цифровими сервісами змушують компанії приділяти значну увагу рівню якості власних програмних продуктів, оскільки навіть незначні недоліки можуть впливати на популярність веб-систем та рівень задоволеності користувачів. Саме тому процес забезпечення якості програмного забезпечення став невід'ємною частиною сучасної розробки інформаційних систем.

Зростання обсягів програмного коду, поява складних веб-архітектур та необхідність швидкого оновлення програмних продуктів створюють додаткові труднощі під час перевірки програмного забезпечення. Розробка нових функціональних можливостей часто супроводжується появою помилок, які можуть залишатися непоміченими без проведення належного тестування. У таких умовах перевірка програмного забезпечення виконує не лише функцію пошуку

дефектів, але й дозволяє оцінити загальну стабільність роботи системи та відповідність програмного продукту очікуванням користувачів.

Важливість процесів контролю якості особливо помітна під час створення веб-додатків, де програмне забезпечення постійно взаємодіє з великою кількістю користувачів та зовнішніх сервісів. Робота веб-систем у різних середовищах, підтримка різних типів пристроїв та постійне оновлення функціоналу потребують системного підходу до перевірки програмного забезпечення. Це зумовлює необхідність використання ефективних методів тестування та постійного вдосконалення підходів до забезпечення якості програмних продуктів.

Якість програмного забезпечення є одним із найважливіших показників ефективності сучасних цифрових продуктів. У процесі розвитку інформаційних технологій вимоги до програмного забезпечення постійно зростають, оскільки програмні системи використовуються для виконання великої кількості завдань у різних сферах діяльності людини. Від рівня якості програмного продукту залежить стабільність його роботи, швидкість виконання операцій, зручність використання, безпека даних та загальне задоволення користувачів. Саме тому питання забезпечення якості програмного забезпечення займає важливе місце у сучасній програмній інженерії та процесах розробки цифрових продуктів [1-3].

Поняття якості програмного забезпечення не має єдиного універсального трактування, оскільки різні дослідники, організації та стандарти пропонують власні підходи до його визначення. У більшості випадків якість програмного забезпечення розглядається як сукупність характеристик програмного продукту, які визначають його здатність задовольняти встановлені або очікувані потреби користувачів. При цьому оцінювання якості може здійснюватися як з позиції відповідності технічним вимогам, так і з позиції зручності та ефективності практичного використання програмного забезпечення. [1,2].

Одним із поширених підходів до визначення якості є трактування її як відповідності програмного продукту встановленим вимогам. У межах такого підходу програмне забезпечення вважається якісним у випадку, якщо воно повністю реалізує визначений функціонал та відповідає технічній документації.

Водночас існує інший підхід, відповідно до якого якість визначається здатністю програмного продукту задовольняти потреби користувачів та бути придатним для практичного використання. У реальних умовах ці два підходи тісно пов'язані між собою, оскільки навіть повна відповідність документації не гарантує зручності використання або задоволення очікувань користувачів [1,2].

Проблема визначення якості програмного забезпечення ускладнюється тим, що очікування користувачів та вимоги до програмних продуктів можуть змінюватися залежно від умов використання, типу програмної системи та сфери її застосування. Часто користувачі очікують від програмного забезпечення певних можливостей або характеристик, які не були явно сформульовані під час створення вимог. У таких випадках навіть формально правильна реалізація функціоналу може не забезпечувати належного рівня задоволеності користувачів. Це створює необхідність постійного вдосконалення процесів аналізу вимог, розробки та тестування програмного забезпечення [1,2].

Сучасні підходи до забезпечення якості програмного забезпечення базуються на використанні міжнародних стандартів та моделей якості. Значного поширення набули стандарти серії ISO 9000, які визначають загальні принципи управління якістю та підходи до організації процесів забезпечення якості програмних продуктів. Одним із найбільш актуальних стандартів у сфері оцінювання якості програмного забезпечення є ISO/IEC 25010:2011, який визначає систему характеристик якості програмного забезпечення та дозволяє комплексно оцінювати програмні продукти [1,2].

Відповідно до сучасних моделей якості, програмне забезпечення оцінюється за низкою характеристик, серед яких функціональна відповідність, продуктивність, сумісність, зручність використання, надійність, безпека, зручність супроводу та переносимість. Кожна з цих характеристик має власні атрибути та показники оцінювання. Наприклад, функціональна відповідність характеризує повноту та правильність реалізованих функцій, а надійність визначає здатність програмного забезпечення стабільно працювати у різних умовах експлуатації [1,2].

Особливого значення у сучасних програмних продуктах набуває зручність використання програмного забезпечення. Користувачі очікують від цифрових систем не лише правильного виконання функцій, але й простоти взаємодії, зрозумілого інтерфейсу та мінімальної кількості дій для досягнення необхідного результату. У зв'язку з цим важливими характеристиками якості стають легкість навчання роботі з програмним продуктом, доступність інтерфейсу та захист користувачів від помилкових дій [2,3].

Не менш важливими характеристиками сучасного програмного забезпечення є безпека та надійність. Використання веб-додатків для роботи з персональними даними, фінансовими операціями та конфіденційною інформацією вимагає забезпечення належного рівня захисту даних та стійкості систем до можливих збоїв або зовнішніх загроз. Програмне забезпечення повинно забезпечувати стабільну роботу навіть за умов високого навантаження або виникнення помилкових ситуацій [1-3].

Якість програмного забезпечення безпосередньо пов'язана з якістю процесів його розробки. Значний вплив на рівень якості мають організація життєвого циклу програмного забезпечення, ступінь формалізації процесів розробки, правильність постановки вимог, ефективність взаємодії між учасниками проєкту та контроль результатів виконання робіт. Високий рівень організації процесів розробки дозволяє зменшити кількість помилок ще на ранніх етапах створення програмного продукту та підвищити ефективність подальшого тестування [1].

У сучасній програмній інженерії важливе місце займає концепція забезпечення якості протягом усього життєвого циклу програмного забезпечення. Якість не може формуватися лише на завершальному етапі перевірки готового продукту. Для досягнення належного рівня якості необхідно здійснювати контроль процесів аналізу вимог, проєктування, розробки, тестування, впровадження та супроводу програмного забезпечення. Саме тому значна увага приділяється процесам верифікації та валідації, які дозволяють оцінити

правильність реалізації програмного продукту та його відповідність потребам користувачів [1].

Важливу роль у забезпеченні якості програмного забезпечення відіграє тестування. Тестування дозволяє виявляти дефекти, оцінювати стабільність роботи програмного продукту та перевіряти відповідність реалізованого функціоналу встановленим вимогам. Сучасні підходи до тестування охоплюють перевірку окремих модулів програмного забезпечення, аналіз взаємодії компонентів системи та комплексне оцінювання роботи програмного продукту в цілому [3].

Модульне тестування використовується для перевірки окремих функціональних елементів програми та дозволяє виявляти помилки ще на ранніх етапах розробки. Інтеграційне тестування спрямоване на перевірку взаємодії між окремими компонентами системи, а системне тестування дозволяє оцінити роботу програмного забезпечення як єдиного цілого. Використання різних рівнів тестування дозволяє підвищити ефективність виявлення дефектів та зменшити ризик виникнення помилок під час експлуатації програмного продукту [3].

Сучасні процеси забезпечення якості програмного забезпечення активно використовують автоматизацію тестування. Автоматизовані засоби дозволяють скоротити час перевірки програмного забезпечення, забезпечити регулярне виконання тестів та зменшити вплив людського фактору на результати тестування. Особливо ефективною автоматизація є для регресійного тестування, коли необхідно багаторазово виконувати однакові перевірки після внесення змін у програмний код [3].

Для організації процесів тестування та управління якістю використовуються спеціалізовані програмні засоби. Значного поширення набули інструменти для автоматизованого тестування веб-додатків, системи управління дефектами та засоби тестування API. Використання таких інструментів дозволяє підвищити ефективність процесу забезпечення якості та покращити взаємодію між учасниками розробки програмного забезпечення [3].

Важливим аспектом забезпечення якості програмного забезпечення є економічна доцільність процесів тестування та контролю якості. Виявлення дефектів на ранніх етапах розробки дозволяє суттєво зменшити витрати на їх усунення у майбутньому. Помилки, які залишаються невиявленими до моменту впровадження програмного продукту, можуть призводити до значних фінансових втрат, необхідності повторного виконання робіт та погіршення репутації компанії. Саме тому інвестиції у процеси забезпечення якості та тестування програмного забезпечення є важливою складовою успішної реалізації сучасних цифрових проєктів [1,3].

Рівень якості програмного забезпечення безпосередньо впливає на конкурентоспроможність цифрових продуктів. Програмне забезпечення, яке працює стабільно, відповідає очікуванням користувачів та забезпечує зручність використання, має більше шансів отримати позитивну оцінку користувачів та зайняти стійкі позиції на ринку. Висока якість програмних продуктів також сприяє формуванню позитивної репутації компанії та підвищенню довіри з боку клієнтів і партнерів [2,3].

Сучасний розвиток технологій створює нові вимоги до процесів забезпечення якості програмного забезпечення. Постійне оновлення цифрових продуктів, використання хмарних технологій, розвиток веб-систем та інтеграція елементів штучного інтелекту вимагають удосконалення підходів до тестування та контролю якості. У зв'язку з цим забезпечення якості програмного забезпечення залишається одним із найважливіших напрямів розвитку сучасної програмної інженерії [1–3].

Для забезпечення високого рівня якості програмного забезпечення недостатньо лише загального розуміння характеристик якісного продукту. У процесі розвитку програмної інженерії виникла потреба у створенні спеціальних моделей та методів оцінювання якості, які дозволяють формалізувати вимоги до програмного забезпечення, визначати критерії оцінювання та виконувати порівняння різних програмних рішень. Використання моделей якості дозволяє

структурувати уявлення про властивості програмного продукту та забезпечує більш об'єктивний підхід до оцінювання його характеристик [4,5].

Моделі якості програмного забезпечення формувались поступово у відповідь на ускладнення програмних систем та зростання вимог до їх функціонування. На ранніх етапах розвитку програмної інженерії оцінювання якості переважно базувалось на суб'єктивному сприйнятті користувачів або досвіді розробників. Згодом виникла необхідність створення універсальних підходів, які б дозволяли оцінювати якість програмного забезпечення за визначеними характеристиками та метриками. У результаті були розроблені спеціалізовані моделі якості, що використовуються для аналізу програмних продуктів, постановки вимог та контролю результатів розробки [5].

Однією з перших відомих моделей оцінювання якості програмного забезпечення стала модель Маккола, запропонована у 1977 році. У цій моделі якість програмного забезпечення розглядається через систему факторів, критеріїв та метрик. Фактори описують програмне забезпечення з позиції користувача та його очікувань, критерії використовуються для оцінювання властивостей програмного продукту з позиції розробки, а метрики дозволяють кількісно оцінити рівень якості окремих характеристик [5].

Модель Маккола стала однією з перших спроб систематизувати процес оцінювання якості програмного забезпечення. Вона передбачає поділ факторів якості на декілька груп залежно від способу взаємодії користувача із програмним продуктом. Структура цієї моделі традиційно подається у вигляді так званого трикутника Маккола, який демонструє взаємозв'язок між факторами, критеріями та метриками якості програмного забезпечення (рис. 1.1) [5].

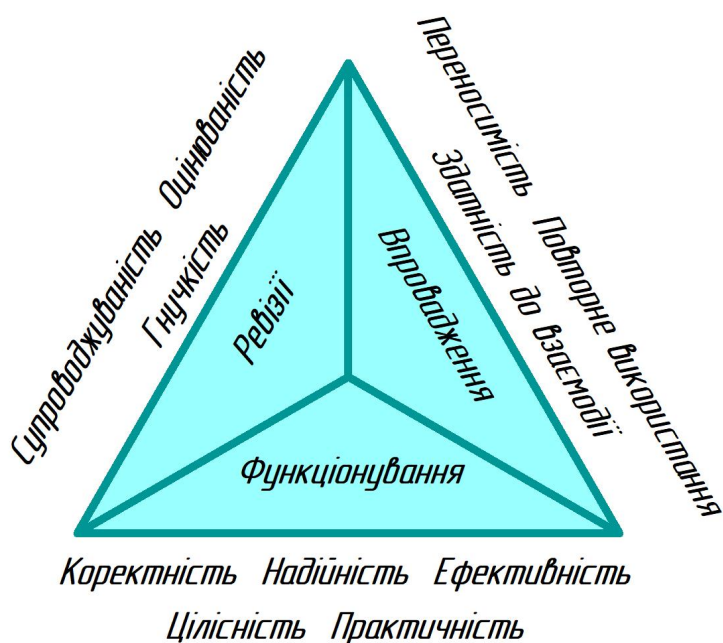


Рис. 1.1 – Трикутник Маккола

У межах моделі Маккола значна увага приділяється використанню метрик якості. Метрики дозволяють оцінювати характеристики програмного забезпечення за допомогою кількісних показників, що спрощує процес аналізу та порівняння програмних продуктів. До таких метрик належать точність роботи програмного забезпечення, стійкість до помилок, ефективність виконання операцій, модульність, простота використання, захищеність, незалежність від апаратної або програмної платформи та інші характеристики. Використання метрик дозволяє зменшити рівень суб'єктивності під час оцінювання якості програмного забезпечення [5].

Подальший розвиток підходів до оцінювання якості привів до створення моделі Боєма, яка стала розширенням ідей Маккола. У цій моделі характеристики якості програмного забезпечення класифікуються відповідно до способів використання програмного продукту. Модель Боєма доповнила попередні підходи новими атрибутами, серед яких зрозумілість програмного забезпечення, зручність внесення змін, документованість, можливість відновлення після збоїв та економічна ефективність. Такий підхід дозволив більш детально враховувати особливості експлуатації програмних систем та їх супроводу [5].

Важливим етапом розвитку моделей якості стало створення моделі FURPS компанією Hewlett Packard. На відміну від попередніх моделей, FURPS

передбачає поділ вимог до програмного забезпечення на функціональні та нефункціональні. Основними характеристиками цієї моделі є функціональність, зручність використання, надійність, продуктивність та супроводжуваність. Надалі модель була розширена до версії FURPS+, у якій додатково враховуються вимоги до проектування, реалізації, інтерфейсу та фізичних компонентів системи [5].

Особливістю моделі FURPS є її практична орієнтація на процес розробки програмного забезпечення. Ця модель широко використовується під час формування вимог до програмних систем та оцінювання готових програмних продуктів. Водночас дослідники зазначають, що окремі сучасні характеристики веб-додатків, зокрема доступність або відповідність правовим вимогам, у класичній моделі FURPS враховані недостатньо повно [5].

Окремі моделі якості програмного забезпечення приділяють увагу не лише характеристикам готового продукту, але й процесам його створення. Наприклад, модель Гецци розділяє поняття якості програмного продукту та якості процесу розробки. У межах цього підходу оцінюються такі характеристики, як цілісність програмного забезпечення, продуктивність, практичність, можливість повторного використання компонентів, зрозумілість, ефективність та своєчасність реагування системи [5].

Для оцінювання ризиків та ефективності процесів розробки використовуються також спеціалізовані моделі, орієнтовані на аналіз процесів забезпечення якості. Одним із прикладів є модель SATC, розроблена в центрі забезпечення якості програмного забезпечення NASA. Ця модель передбачає окреме оцінювання якості вимог, програмного забезпечення, документації, тестування та процесів виконання робіт. Подібний підхід дозволяє оцінювати не лише кінцевий програмний продукт, але й ефективність організації процесу його створення [5].

Однією з найбільш відомих моделей оцінювання якості програмного забезпечення є стандарт ISO 9126, прийнятий у 1991 році. У цій моделі якість програмного забезпечення оцінюється за допомогою системи цілей, атрибутів та метрик. Цілі визначають загальні характеристики, яких очікують від програмного

продукту, атрибути описують властивості програмного забезпечення, а метрики дозволяють кількісно оцінити рівень відповідності цим характеристикам [5].

Модель ISO 9126 передбачає шість основних характеристик якості програмного забезпечення: функціональність, надійність, зручність використання, ефективність, супроводжуваність та переносимість. Пізніше стандарт був розширений шляхом додавання нових атрибутів, пов'язаних із відповідністю стандартам та привабливістю програмного продукту для користувачів [5].

Використання моделей якості дозволяє систематизувати вимоги до програмного забезпечення та забезпечити єдиний підхід до оцінювання його характеристик. Завдяки цьому стає можливим визначення сильних і слабких сторін програмного продукту, оцінювання рівня його відповідності очікуванням користувачів та прийняття рішень щодо подальшого вдосконалення системи [5].

У сучасних умовах значного поширення набувають програмні системи, що використовують алгоритми штучного інтелекту. Це створює додаткові труднощі у процесі оцінювання якості програмного забезпечення, оскільки результати роботи таких систем можуть залежати від процесу навчання алгоритмів та змінюватися навіть за однакових вхідних даних. У деяких випадках оцінювання якості результатів роботи систем штучного інтелекту ускладнюється через відсутність чітких критеріїв для аналізу текстів, зображень або інших даних, створених програмою [4].

Особливість оцінювання якості програмного забезпечення із застосуванням алгоритмів штучного інтелекту полягає також у високому рівні впливу людського фактору. Наприклад, під час аналізу згенерованих текстів або зображень оцінювання результатів часто залежить від суб'єктивного сприйняття користувачів. У зв'язку з цим сучасні дослідження спрямовані на розробку нових моделей та методів оцінювання, які дозволять підвищити точність оцінювання якості та зменшити вплив суб'єктивних факторів [4].

Незважаючи на значну кількість існуючих моделей оцінювання якості програмного забезпечення, жодна з них не може повністю охопити всі можливі характеристики сучасних програмних продуктів. Постійний розвиток веб-

технологій, мобільних застосунків, хмарних сервісів та систем штучного інтелекту створює необхідність подальшого вдосконалення підходів до оцінювання якості. Саме тому сучасні моделі якості постійно доповнюються новими характеристиками, атрибутами та методами оцінювання, які враховують особливості сучасних програмних систем [4,5].

Верифікація програмного забезпечення посідає важливе місце у процесі створення інформаційних систем, оскільки саме вона дозволяє оцінити відповідність отриманого результату встановленим вимогам ще до моменту завершення розробки продукту. У сучасних умовах програмні системи стають дедалі складнішими, а кількість взаємопов'язаних компонентів постійно зростає, тому контроль правильності реалізації окремих функцій і механізмів набуває особливого значення. Відсутність належної перевірки на проміжних етапах створення програмного забезпечення може призвести до накопичення помилок, які в подальшому значно ускладнюють супровід системи, підвищують витрати на її підтримку та негативно впливають на стабільність роботи готового продукту [6].

Під верифікацією розуміють процес визначення того, наскільки програмні засоби та їх окремі компоненти відповідають вимогам, сформованим для них на різних етапах життєвого циклу програмного забезпечення. Основною метою такого процесу є підтвердження правильності реалізації програмного продукту та виявлення дефектів, які могли виникнути під час проєктування, програмування або внесення змін у систему. Верифікація не обмежується лише пошуком помилок, а також передбачає їх фіксацію, аналіз причин виникнення та оцінку можливих наслідків для подальшого функціонування програмного забезпечення [6].

Особливого значення верифікація набуває під час колективної розробки програмних систем, коли результати роботи одних фахівців використовуються іншими учасниками команди як основа для подальших етапів створення продукту. У таких умовах необхідним стає постійний контроль якості проміжних результатів, що дозволяє своєчасно виявляти невідповідності та уникати поширення помилок між окремими частинами системи. Завдяки цьому

забезпечується більш узгоджена взаємодія між учасниками розробки та зменшується ймовірність появи критичних дефектів у фінальній версії програмного забезпечення [6].

Верифікацію доцільно розглядати як процес, тісно пов'язаний із проєктуванням, реалізацією та супроводом програмних систем. Її ефективність значною мірою залежить від того, наскільки органічно вона інтегрована у загальний процес розробки. Регулярна перевірка програмних компонентів дозволяє своєчасно виявляти проблеми, знижувати складність їх усунення та підтримувати необхідний рівень якості програмного продукту протягом усього життєвого циклу системи [6].

Важливо розмежовувати поняття верифікації та налагодження програмного забезпечення, оскільки ці процеси мають різне призначення. Налагодження спрямоване безпосередньо на пошук і усунення помилок у роботі системи, тоді як верифікація орієнтована на виявлення самих фактів наявності дефектів, умов їх виникнення та перевірку відповідності програмного продукту встановленим вимогам. На відміну від налагодження, верифікація має більш системний характер і включає планування перевірок, аналіз отриманих результатів, оцінювання можливих ризиків та контроль впливу виправлень на інші компоненти системи. Саме тому її розглядають як один із важливих засобів підтримання необхідного рівня якості програмного забезпечення [6].

Розробка сучасного програмного забезпечення є складним багатоступеневим процесом, що потребує чіткого планування, координації дій команди та контролю якості на всіх етапах створення продукту. Для впорядкування процесу розробки використовується поняття життєвого циклу програмного забезпечення (Software Development Life Cycle, SDLC).

Життєвий цикл програмного забезпечення – це сукупність етапів і процесів, пов'язаних зі створенням, впровадженням, експлуатацією та супроводом програмного продукту від моменту формування вимог до завершення його використання [6,7]. Основною метою застосування моделей життєвого циклу є

створення якісного програмного продукту, який відповідає вимогам замовника, є економічно доцільним та придатним до подальшого супроводу [8,11].

Використання моделей життєвого циклу дозволяє структурувати процес розробки, спростити управління проєктом, своєчасно виявляти помилки та адаптувати програмний продукт до змін вимог або ринкових умов. Крім того, життєвий цикл допомагає вирішувати типові проблеми розробки програмного забезпечення, пов'язані з неправильним розумінням потреб користувачів, труднощами налагодження, змінами вимог та забезпеченням безпеки продукту [9].

У більшості моделей життєвого циклу виділяють такі основні етапи [8,9]:

- аналіз та збір вимог;
- проєктування системи;
- реалізація (кодування);
- тестування;
- розгортання;
- супровід і підтримка.

На етапі аналізу вимог визначаються цілі проєкту, функціональні можливості системи, вимоги замовника та обмеження майбутнього програмного продукту. Результатом цього етапу зазвичай є специфікація вимог до програмного забезпечення (SRS) [8].

Етап проєктування передбачає розробку архітектури системи, визначення структури програмних компонентів, інтерфейсів та способів взаємодії між модулями. Після цього здійснюється реалізація програмного коду відповідно до сформованих вимог і проєктних рішень [8,9].

Після завершення реалізації виконується тестування програмного забезпечення, під час якого перевіряється відповідність продукту встановленим вимогам, виявляються дефекти та оцінюється коректність роботи системи. Далі програмний продукт розгортається у робочому середовищі та переходить на етап експлуатації й супроводу [8].

Існує декілька основних моделей життєвого циклу програмного забезпечення, які відрізняються організацією етапів розробки, способами взаємодії із замовником та підходами до внесення змін.

Однією з перших моделей стала каскадна модель (Waterfall model). Вона передбачає послідовне виконання етапів розробки, де перехід до наступного етапу можливий лише після повного завершення попереднього. Такий підхід забезпечує чіткість структури проєкту та спрощує планування, однак є малогнучким у разі зміни вимог під час розробки [6,7].

Подальшим розвитком каскадного підходу стала V-подібна модель (V-model), у якій кожному етапу розробки відповідає окремий етап перевірки та контролю. Особливістю цієї моделі є раннє планування тестування та постійна перевірка коректності результатів, що дозволяє зменшити ризик виявлення критичних помилок на пізніх стадіях розробки [6,7].

Інкрементна модель (Incremental model) базується на поетапному створенні програмного продукту у вигляді окремих версій або інкрементів. Кожна нова версія розширює функціональність системи, що дозволяє поступово вдосконалювати продукт та отримувати зворотний зв'язок від користувачів [7].

Для складних проєктів із невизначеними або змінними вимогами часто застосовується спіральна модель (Spiral model). Вона поєднує ітеративну розробку з постійним аналізом ризиків та повторюваними циклами уточнення вимог і вдосконалення системи. Кожен виток спіралі завершується створенням чергової версії продукту або його прототипу [6,7].

У сучасній розробці програмного забезпечення значного поширення набули гнучкі методології (Agile). Agile-підхід базується на ітеративній розробці, тісній взаємодії команди із замовником, швидкому реагуванні на зміни та постійному вдосконаленні продукту. Робота виконується короткими ітераціями, що дозволяє регулярно отримувати працездатні версії програмного забезпечення [7,10].

Однією з найпоширеніших реалізацій Agile є Scrum. У цій методології розробка організовується у вигляді коротких спринтів тривалістю від одного до кількох тижнів, після завершення яких команда демонструє готовий результат.

Scrum забезпечує швидкий зворотний зв'язок, високу адаптивність до змін вимог та ефективну взаємодію між учасниками проєкту [7].

Таким чином, вибір моделі життєвого циклу залежить від особливостей програмного проєкту, складності системи, стабільності вимог, доступних ресурсів та необхідного рівня гнучкості. Незалежно від обраної моделі, тестування та контроль якості залишаються невід'ємними складовими процесу розробки програмного забезпечення.

Забезпечення якості програмного забезпечення є одним із ключових аспектів сучасної розробки ПЗ. У процесі створення програмного продукту важливо не лише реалізувати необхідний функціонал, а й гарантувати його надійність, стабільність, безпечність та відповідність вимогам користувачів. Саме тому в інженерії програмного забезпечення широко використовуються поняття Quality Assurance (QA), Quality Control (QC) та Software Testing. Попри те, що ці терміни часто використовуються як синоніми, вони мають різне призначення та виконують різні функції у межах життєвого циклу розробки програмного забезпечення [11–14].

Тестування не є ізольованим етапом розробки. Воно інтегроване у життєвий цикл програмного забезпечення (SDLC) та залежить від обраної моделі розробки. У сучасних методологіях тестування виконується практично на всіх етапах створення продукту – від аналізу вимог до супроводу готової системи. Основною метою використання процесів QA, QC та тестування є створення якісного програмного продукту, який відповідає потребам замовника та очікуванням кінцевих користувачів [11,15].

Quality Assurance (QA) – це комплекс процесів і заходів, спрямованих на забезпечення якості програмного продукту на всіх етапах його розробки. Основна мета QA полягає у попередженні виникнення дефектів ще до їх появи. На відміну від тестування, яке переважно орієнтоване на пошук помилок у готовому продукті, QA фокусується на побудові правильних процесів розробки. QA охоплює планування якості, визначення стандартів розробки, створення тестових стратегій, аналіз вимог, аудит процесів та вдосконалення методологій роботи команди.

Таким чином, QA забезпечує правильність і передбачуваність процесу створення програмного забезпечення [12–14].

QA має процесно-орієнтований характер. Його головна ідея полягає у створенні таких умов розробки, за яких імовірність появи дефектів буде мінімальною. Якщо провести аналогію з процесом приготування їжі, то QA можна порівняти з контролем рецепта, перевіркою якості інгредієнтів та дотриманням технології приготування. У межах розробки програмного забезпечення QA починається ще до написання програмного коду. Наприклад, спеціалісти QA можуть перевіряти вимоги до системи, аналізувати можливі ризики, проводити аудит документації та брати участь у плануванні архітектури програмного продукту [12–14].

Quality Control (QC) – це процес контролю якості готового програмного продукту або його компонентів. Основна мета QC полягає у виявленні дефектів, помилок і невідповідностей встановленим вимогам. На відміну від QA, який спрямований на попередження проблем, QC орієнтується на їх виявлення та усунення. QC виконується переважно після реалізації функціоналу або створення окремих компонентів системи [12–14].

QC має продуктно-орієнтований характер, оскільки головним об'єктом його уваги є вже створений програмний продукт. Якщо QA відповідає за правильність процесу створення системи, то QC оцінює якість кінцевого результату. Для кращого розуміння різниці між QA та QC часто використовують аналогії з реального життя. Наприклад, у процесі приготування торта QA відповідає за дотримання рецепта, правильний вибір інгредієнтів і налаштування духовки, тоді як QC полягає у перевірці готового торта на смак, якість випікання та відповідність очікуванням [14].

Тестування програмного забезпечення (Software Testing) – це процес аналізу та перевірки програмного забезпечення з метою визначення його відповідності вимогам, очікуванням користувачів і поставленим бізнес-цілям. Традиційно тестування визначають як процес пошуку дефектів і помилок, порівняння фактичного результату з очікуваним або перевірку відповідності програмного

продукту вимогам. Проте сучасний підхід до тестування є значно ширшим. Тестування розглядається не лише як процес пошуку багів, а як комплексна діяльність, спрямована на аналіз поточного стану проєкту, оцінку ризиків і покращення якості як самого продукту, так і процесу його розробки [15].

У сучасній розробці тестування інтегрується в усі етапи SDLC і виконується постійно. Воно включає аналіз вимог, створення тестової документації, виконання тестів, реєстрацію та аналіз дефектів, перевірку виправлень, оцінювання стабільності й продуктивності системи. Крім пошуку помилок, тестування дозволяє оцінювати якість продукту, перевіряти відповідність вимогам, знижувати ризики виходу неякісного програмного забезпечення та покращувати процеси розробки [13,15].

QA, QC і тестування є взаємопов'язаними процесами забезпечення якості програмного забезпечення. Їх можна представити у вигляді ієрархії: тестування є частиною QC, а QC, своєю чергою, є частиною QA. Таким чином, QA відповідає за якість процесів, QC контролює якість продукту, а тестування забезпечує практичну перевірку системи та пошук дефектів. Дані, отримані під час тестування, використовуються для вдосконалення процесів розробки та підвищення загальної якості програмного забезпечення [12–14].

Таблиця 1.1

Параметр	QA (Quality Assurance)	QC (Quality Control)
Основна мета	Попередження дефектів	Виявлення дефектів
Орієнтація	Процеси	Продукт
Основний підхід	Превентивний	Коригуючий
Основні методи	Планування, аудит, стандартизація	Тестування, перевірка
Етап виконання	Усі етапи SDLC	Переважно після розробки
Результат	Покращення процесів	Контроль якості продукту

Отже, QA, QC та тестування є невід'ємними складовими сучасної розробки програмного забезпечення. Їх комплексне використання дозволяє не лише

своєчасно виявляти дефекти, а й будувати ефективні процеси створення якісного, надійного та безпечного програмного продукту [11–15].

Таким чином, було розглянуто основні теоретичні засади забезпечення якості програмного забезпечення. Визначено сутність верифікації та її роль у підтвердженні відповідності встановленим вимогам та окреслено її відмінності від налагодження. Також було проаналізовано поняття життєвого циклу програмного забезпечення, його основні етапи та найпоширеніші моделі розробки. Встановлено, що вибір моделі життєвого циклу впливає на організацію процесу розробки, тестування та внесення змін до програмного продукту. Крім того, розглянуто взаємозв'язок понять QA, QC та тестування. Визначено, що QA спрямований на забезпечення якості процесів розробки, QC – на контроль якості готового продукту, а тестування є основним практичним засобом виявлення дефектів і перевірки відповідності програмного забезпечення вимогам. Комплексне застосування цих підходів дозволяє підвищити якість, надійність та стабільність програмних систем.

1.2 Основні види та рівні тестування програмного забезпечення

Тестування програмного забезпечення є одним із основних етапів забезпечення якості сучасних програмних продуктів. У процесі розробки програмного забезпечення виникає велика кількість потенційних помилок, пов'язаних як із реалізацією функціоналу, так і з особливостями взаємодії окремих компонентів системи. Саме тому перевірка працездатності, стабільності, безпеки та відповідності програмного продукту встановленим вимогам є необхідною умовою створення надійного програмного забезпечення.

Сучасні програмні системи відзначаються високою складністю, значною кількістю взаємопов'язаних модулів та необхідністю роботи в різних програмних і апаратних середовищах. У таких умовах застосування лише одного підходу до тестування є недостатнім, оскільки різні типи помилок можуть проявлятися на різних етапах розробки та експлуатації програмного продукту. Для підвищення

ефективності перевірки програмного забезпечення використовуються різні види та рівні тестування, кожен із яких має власне призначення, методи виконання та область застосування.

Залежно від поставлених цілей тестування може бути спрямоване як на перевірку функціональних можливостей системи, так і на оцінювання її нефункціональних характеристик, зокрема продуктивності, сумісності, зручності використання та стабільності роботи. Крім того, важливу роль відіграє класифікація тестування за рівнями, що дозволяє здійснювати послідовну перевірку окремих компонентів системи, їх взаємодії та працездатності програмного продукту в цілому.

Комплексне використання різних видів і рівнів тестування дозволяє своєчасно виявляти дефекти, знижувати ризики виникнення критичних помилок після випуску програмного забезпечення та забезпечувати належний рівень якості програмних систем. Саме тому класифікація підходів до тестування та аналіз особливостей їх застосування є важливою складовою сучасної інженерії програмного забезпечення.

Тестування програмного забезпечення є процесом перевірки та оцінювання якості програмного продукту з метою виявлення помилок, дефектів і невідповідностей вимогам. Його основне призначення полягає у підтвердженні того, що програмне забезпечення працює коректно, забезпечує необхідний рівень надійності, безпеки та продуктивності, а також відповідає очікуванням користувачів. Процес тестування включає планування перевірок, створення тестових сценаріїв, виконання тестів, аналіз результатів та формування звітності. У сучасній практиці тестування охоплює велику кількість підходів, рівнів і методів. У деяких класифікаціях описується понад сто різних типів тестування, однак більшість із них є спеціалізованими або похідними від базових категорій [16,18].

Однією з основних класифікацій є поділ тестування на функціональне та нефункціональне. Функціональне тестування перевіряє відповідність програмного забезпечення функціональним вимогам і бізнес-логіці. Воно орієнтоване на

перевірку правильності роботи функцій, обробки вхідних даних і формування очікуваних результатів. Таке тестування імітує реальне використання системи та виконується на різних рівнях: модульному, інтеграційному, системному та приймальному. Нефункціональне тестування, навпаки, оцінює характеристики, які не пов'язані безпосередньо з функціональністю: продуктивність, стабільність, безпеку, зручність використання, масштабованість і сумісність. Саме нефункціональні перевірки дозволяють оцінити поведінку системи в умовах реальної експлуатації [16-18].

Важливе місце займає класифікація тестування за рівнями перевірки програмного забезпечення. Найнижчим рівнем є модульне або unit-тестування, яке використовується для перевірки окремих модулів, процедур чи класів. Під час такого тестування аналізуються логічні гілки алгоритму, граничні умови та коректність роботи окремих компонентів. Повнота тестування визначається рівнем покриття операторів, умов або гілок коду. Модульне тестування вважається одним із найпоширеніших видів перевірки, оскільки дозволяє виявляти помилки на ранніх етапах розробки [6,16,18].

Після перевірки окремих модулів виконується інтеграційне тестування. Його мета полягає у перевірці взаємодії між компонентами системи. Навіть якщо окремі модулі працюють правильно, під час їх об'єднання можуть виникати помилки обміну даними, несумісності інтерфейсів або проблеми з логікою взаємодії. Інтеграційне тестування проводиться протягом усього життєвого циклу розробки, оскільки раннє виявлення інтеграційних дефектів значно спрощує їх локалізацію та усунення. Для інтеграції компонентів можуть використовуватись різні підходи: top-down, bottom-up, hybrid або big bang integration [6,16,18].

Завершальним етапом перевірки є системне тестування, під час якого система розглядається як єдине ціле. На цьому рівні перевіряється відповідність програмного забезпечення функціональним і нефункціональним вимогам, а також поведінка системи в реальному середовищі. Системне тестування охоплює сценарії роботи користувачів, перевірку стабільності, продуктивності, взаємодії із зовнішніми сервісами та правильності виконання бізнес-процесів. Близьким до

нього є end-to-end testing, яке моделює повний цикл роботи користувача із системою. Окремо виділяють acceptance testing – приймальне тестування, яке дозволяє замовнику визначити, чи відповідає продукт поставленим вимогам і чи готовий він до впровадження [6,16,18].

Серед нефункціональних перевірок особливе значення має performance testing – тестування продуктивності. Воно оцінює швидкість системи, час відгуку, стабільність роботи та використання ресурсів. До категорії performance testing належить декілька спеціалізованих видів тестування. Load testing використовується для перевірки роботи системи за нормального очікуваного навантаження. Stress testing досліджує поведінку системи при перевищенні допустимого навантаження та дозволяє оцінити стійкість до перевантажень. Volume testing перевіряє здатність програмного забезпечення працювати з великими обсягами даних. Endurance або soak testing використовується для оцінки стабільності роботи системи протягом тривалого часу, а spike testing аналізує поведінку системи під час різких стрибків навантаження. Навантажувальні перевірки є особливо важливими для вебсервісів, банківських систем і багатокористувацьких платформ [6,17,18].

Окрему категорію складає тестування безпеки. Security testing спрямоване на виявлення вразливостей і перевірку захищеності системи від зовнішніх атак. До нього належать penetration testing, vulnerability testing та fuzz testing. Під час fuzz testing до системи подаються випадкові або некоректні дані з метою виявлення збоїв, витоків пам'яті або помилок обробки введення. Penetration testing імітує дії зловмисника та дозволяє оцінити рівень захисту системи. Подібні перевірки є критично важливими для фінансових систем, державних ресурсів і вебзастосунків, що працюють із персональними даними користувачів [17,18].

Для оцінки зручності взаємодії користувача із системою застосовується usability testing. Воно дозволяє визначити, наскільки програмне забезпечення є зрозумілим, простим і комфортним у використанні. Такі перевірки часто виконуються за участю кінцевих користувачів. Близьким видом є UI testing, яке перевіряє коректність роботи графічного інтерфейсу, кнопок, меню, форм

введення та реакцій системи на дії користувача. Важливим напрямом також є accessibility testing, що оцінює доступність програмного забезпечення для людей з обмеженими можливостями [17,18].

У сучасних умовах значну роль відіграє compatibility testing – тестування сумісності. Воно перевіряє коректність роботи програмного забезпечення на різних операційних системах, браузерях, пристроях і мережевих конфігураціях. Related напрямом є configuration testing, яке дозволяє оцінити роботу системи при різних апаратних і програмних конфігураціях. Для продуктів, орієнтованих на міжнародний ринок, використовуються localization, globalization та internationalization testing. Localization testing перевіряє правильність перекладу інтерфейсу, форматів дати, часу та валюти. Globalization testing оцінює здатність системи працювати в різних мовних і культурних середовищах [17,18].

Тестування також класифікується за рівнем доступу до внутрішньої структури програми. Black box testing базується на перевірці функціональності без знання внутрішньої реалізації системи. White box testing, навпаки, передбачає аналіз внутрішньої логіки коду та використовується для перевірки покриття операторів, гілок і шляхів виконання програми. Gray box testing поєднує обидва підходи, коли тестувальник має часткові знання про внутрішню структуру системи [17,18].

За ступенем автоматизації виділяють ручне, автоматизоване та напівавтоматизоване тестування. Ручне тестування передбачає безпосередню взаємодію тестувальника із системою без використання автоматизованих інструментів. Його перевагою є гнучкість і можливість виявлення нестандартних проблем інтерфейсу або поведінки програми. Автоматизоване тестування використовує спеціальні інструменти та скрипти для виконання тестових сценаріїв. Воно дозволяє пришвидшити перевірки, зменшити вплив людського фактору та багаторазово виконувати однакові тести. Проте автоматизація потребує додаткових ресурсів на створення і підтримку тестових фреймворків. На практиці найефективнішим підходом є поєднання ручного та автоматизованого тестування [16–18].

Серед спеціалізованих підходів до тестування варто виділити *exploratory testing* та *ad-hoc testing*. *Exploratory testing* поєднує дослідження системи, проєктування тестів і їх виконання одночасно. Тестувальник аналізує результати попередніх перевірок і на їх основі створює нові сценарії тестування. Такий підхід потребує високої кваліфікації та практичного досвіду. *Ad-hoc testing* є менш формалізованим і виконується без попередньої документації або детального плану. Основна мета таких перевірок – швидке виявлення нестандартних дефектів [17,18].

Важливими є також регресійне, повторне та димове тестування. *Regression testing* використовується після внесення змін до системи для перевірки того, що нові зміни не порушили вже реалізовану функціональність. *Re-testing* спрямоване на повторну перевірку виправленого дефекту. *Smoke testing* дозволяє швидко визначити працездатність основних функцій системи після створення нової збірки. У деяких класифікаціях *sanity testing* розглядається як різновид *smoke testing* [17,18].

Окрім динамічних перевірок, важливу роль відіграють статичні методи контролю якості. До них належать рев'ю, аналіз документації та формальні інспекції. Формальна інспекція передбачає незалежну перевірку програмного коду або документації групою спеціалістів, які не брали участі у створенні відповідного артефакту. Такий підхід дозволяє виявляти дефекти ще до запуску програмного забезпечення та зменшує вартість подальшого виправлення помилок [6,18].

Таким чином, тестування програмного забезпечення є багаторівневим процесом, який охоплює перевірку функціональності, продуктивності, безпеки, сумісності та зручності використання програмного продукту. Сучасна практика тестування включає велику кількість типів і методів перевірки, кожен з яких вирішує конкретні завдання та використовується залежно від особливостей проєкту. Поєднання різних видів тестування дозволяє підвищити якість програмного забезпечення, зменшити кількість дефектів і забезпечити стабільну роботу системи в реальних умовах експлуатації [6,16–18].

Методи тестування програмного забезпечення визначають спосіб перевірки системи, рівень доступу тестувальника до внутрішньої структури програми та підхід до аналізу її роботи. Найпоширенішими методами є тестування «чорного ящика», «білого ящика» та «сірого ящика». Кожен із них використовується для різних завдань і вимагає різного рівня технічної підготовки тестувальника [6,19].

Метод «чорного ящика» ґрунтується на перевірці системи без доступу до її внутрішньої реалізації. Тестувальник працює лише з вимогами до програмного забезпечення, подає на вхід різні дані та аналізує результати роботи системи. У цьому випадку програмний код, архітектура та алгоритми залишаються прихованими. Основна мета такого тестування — перевірка відповідності поведінки системи функціональним вимогам. Також перевіряється реакція програми на помилкові або нестандартні дії користувача. Якщо система поводить себе не так, як зазначено у вимогах, або демонструє неочікувану поведінку, ці проблеми фіксуються та передаються розробникам. Через те, що джерелом інформації для побудови тестів є вимоги, цей метод ще називають тестуванням за вимогами [6].

Тестування «чорного ящика» є найбільш наближеним до поведінки звичайного користувача. QA-інженер оцінює роботу програми без знання коду та внутрішньої логіки системи. Такий підхід дозволяє перевіряти функціональність, інтерфейс користувача, обробку введених даних, навігацію та загальну коректність роботи програмного забезпечення. Саме цей метод найчастіше використовується під час функціонального, системного та приймального тестування [19].

Метод «білого ящика» передбачає доступ тестувальника до внутрішньої структури системи та програмного коду. Завдяки цьому можна перевіряти не лише зовнішню поведінку програми, а й правильність реалізації окремих алгоритмів, логічних умов, циклів і гілок виконання. Тестувальник може аналізувати, чи існують вимоги для всіх частин коду, а також виявляти фрагменти, які не покриті тестами або взагалі не мають функціонального призначення. Такі ділянки можуть бути джерелом дефектів або неочікуваної поведінки системи [6].

Тестування «білого ящика» часто використовується під час модульного тестування та перевірки логіки роботи окремих компонентів. Воно потребує знання мов програмування, структури проєкту та принципів побудови програмного забезпечення. QA-інженер або розробник не лише виявляє помилки, а й визначає конкретне місце їх виникнення у коді. До цього методу належать перевірки покриття операторів, умов, гілок та шляхів виконання програми [6,19].

Проміжним варіантом між двома підходами є тестування «сірого ящика». У цьому випадку тестувальник має часткові знання про внутрішню реалізацію системи. Наприклад, він може знати структуру бази даних, принцип роботи окремих модулів або архітектуру програми, але не мати повного доступу до всього коду. Такий метод дозволяє ефективніше знаходити дефекти інтеграції, проблеми обміну даними та помилки взаємодії компонентів [19].

Окремим методом є тестування моделей. Його особливість полягає в тому, що перевіряється не сама система, а її формальна модель. За допомогою моделювання можна аналізувати поведінку системи у ситуаціях, які важко або неможливо відтворити у реальному середовищі. Такий підхід застосовується для дослідження стійкості системи, оптимальності алгоритмів і коректності складних процесів. Проте через складність побудови формальних моделей цей метод використовується обмежено, переважно у спеціалізованих галузях, наприклад у системах зв'язку або критично важливому програмному забезпеченні [6].

Ще одним методом є аналіз програмного коду або інспекції. На відміну від динамічного тестування, у цьому випадку система може взагалі не запускатися. Спеціалісти вручну перевіряють програмний код, документацію або окремі алгоритми для виявлення потенційних помилок. Інспекції дозволяють знаходити дефекти у фрагментах коду, які складно активувати під час звичайного тестування, наприклад обробники виняткових ситуацій. Такий підхід допомагає виявляти логічні помилки ще до запуску програми [6].

Методи тестування також можна класифікувати за способом виконання перевірок. Ручне тестування передбачає, що тестувальник самостійно виконує тестові сценарії, перевіряє реакцію системи та аналізує результати. Воно дозволяє

оцінити поведінку програми з точки зору користувача, перевірити зручність інтерфейсу та виявити проблеми, які складно автоматизувати. Автоматизоване тестування виконується за допомогою спеціальних інструментів і тестових скриптів. Воно забезпечує швидке повторне виконання тестів, підвищує точність перевірок і зменшує вплив людського фактора. Найчастіше у сучасних проєктах поєднують обидва підходи [16,19].

Залежно від ступеня свободи тестувальника використовуються різні підходи до організації тестів. Тестування за тест-кейсами виконується відповідно до заздалегідь підготовлених сценаріїв. Дослідницьке тестування передбачає одночасне створення та виконання тестів без жорсткого плану. Вільне тестування базується переважно на досвіді та інтуїції QA-інженера. Такі методи дозволяють виявляти помилки, які не були передбачені стандартними сценаріями перевірки [18,19].

У практиці тестування також застосовуються позитивні та негативні перевірки. Позитивне тестування перевіряє роботу системи з коректними вхідними даними та очікуваними діями користувача. Негативне тестування, навпаки, досліджує поведінку програми у випадку помилкових, некоректних або неочікуваних дій. Саме негативні перевірки дозволяють оцінити стійкість системи та якість обробки помилок [18,19].

Для підвищення ефективності перевірок використовуються спеціальні техніки тест-дизайну. До найвідоміших належать еквівалентне розбиття, аналіз граничних значень, pairwise testing, exploratory testing, fuzz testing та error guessing. Вони допомагають скоротити кількість тестів без втрати якості перевірки та підвищують ймовірність виявлення дефектів. Наприклад, аналіз граничних значень орієнтований на перевірку крайніх допустимих значень даних, а fuzz testing полягає у подачі випадкових або некоректних даних для перевірки стійкості системи [18].

Таким чином, методи тестування програмного забезпечення відрізняються рівнем доступу до системи, способом організації перевірок та підходами до пошуку дефектів. У реальних проєктах зазвичай комбінують декілька методів

одночасно, оскільки жоден із них окремо не може гарантувати повне виявлення всіх помилок. Поєднання функціональних перевірок, аналізу коду, автоматизації та дослідницького тестування дозволяє забезпечити вищий рівень якості програмного забезпечення. [6,16,18,19].

Рівні тестування програмного забезпечення утворюють послідовну структуру перевірок, яка охоплює окремі компоненти системи, їхню взаємодію, роботу програмного продукту загалом та його відповідність очікуванням користувачів. Використання кількох рівнів тестування дозволяє виявляти дефекти на різних етапах життєвого циклу розробки, зменшувати витрати на виправлення помилок і підвищувати стабільність програмного забезпечення. Тестування проводиться протягом усього циклу створення та супроводу ПЗ, а кожен рівень має власну мету, набір методів і типові інструменти [20–22].

Найнижчим рівнем є модульне або компонентне тестування. Воно призначене для перевірки окремих елементів програми: функцій, методів, класів, процедур чи невеликих модулів. Такі тести виконуються ізольовано, без залежності від інших частин системи. Основна мета – переконатися, що кожен окремий компонент працює правильно та повертає очікуваний результат. Модульне тестування дозволяє швидко виявляти локальні дефекти, спрощує налагодження та допомагає контролювати якість коду після внесення змін. Для кожного модуля зазвичай створюються окремі тестові сценарії, що знижує ризик появи регресійних помилок. У сучасній розробці часто застосовується підхід *test-driven development*, коли тести створюються ще до написання основного коду [20–22].

Модульне тестування здебільшого автоматизується, оскільки ручна перевірка великої кількості дрібних компонентів є неефективною. Для цього використовують спеціалізовані фреймворки та середовища налагодження. Автоматизовані модульні тести запускаються після кожної зміни коду, що дозволяє оперативно виявляти нові дефекти. Перевагою такого підходу є швидкість виконання, повторюваність перевірок і можливість інтеграції в системи безперервної інтеграції [20,21].

Після перевірки окремих компонентів виконується інтеграційне тестування. Його завдання полягає у перевірці взаємодії між модулями, підсистемами або зовнішніми сервісами. Навіть якщо окремі частини системи працюють правильно, проблеми можуть виникати під час обміну даними чи спільного виконання операцій. Інтеграційне тестування дозволяє виявити помилки передачі параметрів, конфлікти між модулями, порушення логіки взаємодії або несумісність компонентів. Саме тому цей рівень тестування виконується після модульного та є необхідним етапом перед перевіркою всієї системи [20–22].

У практиці розробки виділяють два різновиди інтеграційного тестування: компонентне та системне. Компонентний рівень перевіряє взаємодію між окремими модулями після їхньої локальної перевірки, тоді як системний інтеграційний рівень оцінює взаємодію між завершеними підсистемами або навіть окремими програмними продуктами. Для автоматизації інтеграційних перевірок часто використовуються системи Continuous Integration, які забезпечують регулярне тестування після оновлення коду [20,21].

Наступним етапом є системне тестування. На цьому рівні програмне забезпечення розглядається як єдина завершена система. Перевіряються як функціональні вимоги, так і нефункціональні характеристики: продуктивність, стабільність, сумісність, зручність використання, робота в реальному середовищі, обробка помилок і використання ресурсів. Системне тестування дозволяє оцінити поведінку програми в умовах, максимально наближених до експлуатації. Саме на цьому етапі часто виявляються проблеми, які неможливо знайти під час ізольованої перевірки компонентів [20–22].

Для системного тестування застосовують два основні підходи. Перший базується на вимогах: для кожної функціональної вимоги створюються окремі тестові випадки. Другий орієнтується на сценарії використання системи. У такому випадку тестувальники моделюють реальні дії користувачів і створюють тести на основі типових сценаріїв роботи з програмою. Комбінація цих підходів дозволяє досягти більш повного покриття функціональності [20,21].

Фінальним основним рівнем є приймальне тестування. Воно виконується для визначення готовності програмного продукту до впровадження та експлуатації. На цьому етапі оцінюється відповідність системи вимогам замовника, бізнес-цілям і очікуванням кінцевих користувачів. Приймальне тестування може проводитися самим замовником, сторонньою організацією або спільно із командою розробників. Якщо результати тестування визнаються задовільними, продукт допускається до релізу або впровадження [20–22].

Особливістю приймального тестування є його орієнтація не на технічні деталі реалізації, а на практичну придатність системи. Перевіряється, наскільки продукт відповідає поставленим завданням, чи задовольняє користувацькі потреби та чи готовий до використання в реальних умовах. Підготовка до цього етапу починається заздалегідь: формується план приймальних робіт, визначаються сценарії перевірки та критерії успішного завершення тестування [20].

Окрім базових рівнів, у практиці тестування використовуються додаткові види перевірок. Регресійне тестування виконується після внесення змін у програмний код та дозволяє переконатися, що нові модифікації не пошкодили вже реалізований функціонал. Альфа-тестування проводиться всередині організації спеціально підготовленими фахівцями до офіційного випуску продукту. Бета-тестування передбачає передачу програмного забезпечення реальним користувачам для перевірки в умовах реальної експлуатації. Такі підходи допомагають виявити помилки, які складно знайти в лабораторному середовищі [22].

Кожен рівень тестування має власне призначення та не може повністю замінити інші. Модульні перевірки дозволяють контролювати якість окремого коду, інтеграційні – забезпечують правильну взаємодію компонентів, системні – оцінюють поведінку продукту загалом, а приймальні – підтверджують готовність програмного забезпечення до використання. Поєднання всіх рівнів тестування забезпечує комплексний контроль якості та значно знижує ризик появи критичних помилок після релізу продукту [20–22].

Таким чином, тестування програмного забезпечення є невід’ємною складовою процесу забезпечення якості програмних продуктів. Воно дозволяє своєчасно виявляти дефекти, перевіряти відповідність функціональних можливостей вимогам та забезпечувати стабільну роботу системи в різних умовах експлуатації. Сучасне тестування охоплює значну кількість підходів і використовується на всіх етапах життєвого циклу програмного забезпечення.

Основними видами тестування є функціональне та нефункціональне тестування. Функціональні перевірки спрямовані на оцінку правильності роботи функцій програмного продукту, тоді як нефункціональні дозволяють оцінити продуктивність, безпеку, сумісність, зручність використання та інші характеристики якості. У практиці розробки також широко застосовуються регресійне тестування, smoke testing, sanity testing, usability testing, compatibility testing та ін., кожен з яких виконує окрему роль у процесі контролю якості.

Важливе значення мають і рівні тестування, що забезпечують послідовну перевірку системи від окремих компонентів до готового програмного продукту. Модульне тестування дозволяє виявляти помилки в окремих елементах коду, інтеграційне – перевіряти взаємодію між компонентами, системне – оцінювати роботу програмного забезпечення загалом, а приймальне тестування підтверджує готовність системи до використання кінцевими користувачами.

Комплексне використання різних видів, методів і рівнів тестування забезпечує підвищення якості програмного забезпечення, зниження ризику появи критичних помилок після релізу та підвищення надійності роботи інформаційних систем.

1.3 Ручне тестування як складова забезпечення якості програмного забезпечення

Попри активний розвиток автоматизованих засобів перевірки програмного забезпечення, ручне тестування продовжує залишатися важливою складовою забезпечення якості програмних продуктів. У багатьох випадках саме участь людини дозволяє виявити помилки, пов'язані з логікою роботи системи, зручністю використання або некоректною поведінкою інтерфейсу, які складно або неможливо визначити за допомогою автоматизованих сценаріїв. Особливо це стосується складних користувацьких взаємодій, нестандартних сценаріїв використання та перевірки загального сприйняття програмного продукту.

Ручне тестування відіграє важливу роль як на ранніх етапах розробки, так і під час підтримки та вдосконалення програмного забезпечення. Воно дозволяє гнучко реагувати на зміни функціональності, швидко перевіряти нові можливості системи та оцінювати якість програмного продукту з позиції кінцевого користувача. Значну роль у цьому процесі відіграє Manual QA Engineer, від уважності, аналітичного мислення та практичного досвіду якого безпосередньо залежить ефективність перевірки.

Важливою частиною ручного тестування є ведення тестової документації, яка забезпечує впорядкованість процесу перевірки та спрощує взаємодію між учасниками розробки. Використання тест-кейсів, чек-листів і звітів про дефекти дозволяє систематизувати результати тестування, контролювати покриття функціональності та забезпечувати повторюваність перевірок. Саме тому ручне тестування залишається одним із базових інструментів контролю якості програмного забезпечення навіть за умов широкого використання автоматизації.

Ручне та автоматизоване тестування є двома основними підходами до перевірки програмного забезпечення, які не замінюють одне одного, а використовуються спільно залежно від особливостей проєкту, складності системи та поставлених завдань. Ручне тестування передбачає безпосередню участь QA-інженера у виконанні тестових сценаріїв, аналізі поведінки системи та оцінці її роботи з позиції кінцевого користувача. Автоматизоване тестування базується на використанні спеціальних інструментів та тестових скриптів, що дозволяють виконувати повторювані перевірки без постійного втручання людини [23,24].

Попри активний розвиток автоматизації, ручне тестування залишається важливою складовою процесу забезпечення якості програмного забезпечення. Воно дозволяє перевіряти інтерфейс, логіку взаємодії користувача із системою, оцінювати зручність використання продукту, а також виконувати дослідницьке тестування, під час якого тестувальник самостійно аналізує нестандартні сценарії роботи програми. Саме людина здатна помітити помилки, пов'язані з незручністю інтерфейсу, нелогічною поведінкою системи або некоректною взаємодією окремих елементів, які складно формалізувати для автоматичної перевірки [23,24].

Однією з головних переваг ручного тестування є його гнучкість. QA-інженер може швидко змінювати сценарії перевірки, адаптуватися до нових умов та перевіряти програму без попередньої підготовки складної технічної інфраструктури. Крім того, ручне тестування ефективне на ранніх етапах розробки, коли функціональність продукту ще нестабільна та постійно змінюється. У таких умовах створення автоматизованих тестів може бути недоцільним через необхідність їх регулярного оновлення [23,24].

Водночас ручне тестування має певні обмеження. Повторне виконання однакових тестових сценаріїв потребує значних витрат часу та ресурсів, а людський фактор підвищує ризик пропуску дефектів через втому або неуважність. Особливо це помітно під час регресійного тестування великих систем, де необхідно регулярно перевіряти значну кількість функцій після внесення змін у програмний код. У таких випадках використання лише ручного тестування є малоефективним [23,24].

Автоматизоване тестування дозволяє значно прискорити процес перевірки програмного забезпечення. Автоматичні тести можуть запускатися необмежену кількість разів, забезпечуючи стабільність перевірок та однакову точність результатів. Автоматизація особливо ефективна для регресійного тестування, API-тестування, перевірки стабільних бізнес-процесів та виконання великої кількості однотипних операцій. Крім того, автоматизовані тести часто інтегруються у процеси безперервної інтеграції та доставки програмного забезпечення, що дозволяє оперативно виявляти дефекти після внесення змін у код [23,24].

Разом із перевагами автоматизація має і власні недоліки. Створення тестових сценаріїв потребує додаткових витрат часу, знань мов програмування та навичок роботи з фреймворками автоматизації. Такі тести також потребують постійної підтримки, оскільки будь-які зміни інтерфейсу або логіки роботи програми можуть вимагати оновлення тестового коду. Крім того, автоматизовані перевірки не здатні повноцінно оцінювати користувацький досвід, дизайн інтерфейсу або нестандартну поведінку системи [23,24].

Для більшості сучасних ІТ-проектів характерне поєднання ручного та автоматизованого тестування. На початкових етапах розробки переважає ручне тестування, яке дозволяє перевірити нову функціональність та оцінити поведінку системи. Після стабілізації основних модулів частина перевірок автоматизується, що забезпечує швидке виконання регресійних тестів та підтримку стабільності продукту у довгостроковій перспективі. Такий комбінований підхід дозволяє поєднати гнучкість ручного тестування із швидкістю та повторюваністю автоматизації [23,24].

Таблиця 1.2 [24]

Критерій	Ручне тестування	Автоматизоване тестування
Виконання тестів	Виконується QA-інженером вручну	Виконується тестовими скриптами
Швидкість перевірки	Нижча, потребує значного часу	Висока швидкість виконання
Повторюваність	Існує ризик людських помилок	Забезпечується стабільна точність
Масштабування	Ускладнюється зі збільшенням системи	Легко масштабується
Необхідні навички	Теорія тестування, аналітичне мислення	Додатково потрібні навички програмування
Основні задачі	UX/UI, exploratory testing, перевірка логіки	Регресійне, API та навантажувальне тестування
Вартість	Нижча на старті	Вища на етапі впровадження

Отже, ручне тестування займає важливе місце у сучасному процесі забезпечення якості програмного забезпечення, оскільки дозволяє оцінювати не

лише технічну правильність роботи системи, а й фактичний досвід взаємодії користувача з продуктом. Незважаючи на активний розвиток автоматизації, саме manual QA engineer часто першим виявляє проблеми, які неможливо ефективно знайти за допомогою автоматичних перевірок. Йдеться про помилки у логіці роботи інтерфейсу, незручність навігації, некоректну поведінку елементів системи або нестандартні сценарії використання, які складно передбачити заздалегідь.

Важливою особливістю ручного тестування є його орієнтація на реальну поведінку користувача. Тестувальник не лише перевіряє відповідність функціоналу технічним вимогам, а й аналізує, наскільки зрозумілою та зручною є система для людини. Саме тому ручне тестування широко використовується під час перевірки вебзастосунків, мобільних програм, інформаційних систем та інших продуктів, де велике значення має якість користувацького досвіду. У багатьох випадках навіть повністю працездатна система може виявитися незручною або нелогічною для користувача, що безпосередньо впливає на ефективність її використання.

Окрему роль у процесі ручного тестування відіграє людський фактор. Manual QA Engineer не обмежується виконанням заздалегідь підготовлених інструкцій, а постійно аналізує поведінку системи, формує нові припущення щодо можливих помилок та перевіряє нестандартні сценарії. Такий підхід особливо важливий у дослідницькому тестуванні, де значна частина дефектів виявляється саме завдяки практичному досвіду, уважності та аналітичному мисленню тестувальника. У цьому полягає одна з головних відмінностей ручного тестування від автоматизованого, оскільки автоматичні перевірки здатні виконувати лише ті дії, які були попередньо закладені у тестові сценарії [23,24].

Ефективність ручного тестування значною мірою залежить від правильної організації процесу перевірки та якості тестової документації. Одним із основних документів є тест-кейс – опис конкретного сценарію перевірки із зазначенням початкових умов, послідовності дій та очікуваного результату. Використання тест-кейсів дозволяє стандартизувати процес тестування, забезпечити

повторюваність перевірок та спростити контроль виконаних робіт. Для швидких або менш формалізованих перевірок часто застосовуються чек-листи, які містять перелік функцій або елементів системи, що підлягають перевірці.

Не менш важливим елементом ручного тестування є документування виявлених дефектів. Для цього використовуються bug reports – звіти про помилки, у яких фіксуються умови виникнення проблеми, очікуваний та фактичний результат, а також додаткові матеріали для відтворення дефекту. Якісне оформлення звітів про помилки дозволяє пришвидшити процес виправлення дефектів та покращує взаємодію між тестувальниками і розробниками. У великих проєктах ведення тестової документації є обов'язковою частиною процесу забезпечення якості, оскільки дозволяє контролювати стан перевірок, відстежувати історію змін та оцінювати стабільність програмного продукту.

Попри очевидні переваги автоматизації, повністю відмовитися від ручного тестування у сучасній розробці програмного забезпечення практично неможливо. Автоматизовані тести добре підходять для перевірки стабільних і повторюваних сценаріїв, однак вони не здатні повною мірою оцінити зручність використання системи, якість інтерфейсу або реакцію користувача на поведінку програми. Крім того, створення та підтримка автоматизованих тестів потребує значних ресурсів, тому автоматизація є доцільною не для всіх типів проєктів і не на всіх етапах розробки [23].

У практичній діяльності більшість ІТ-команд використовують комбінований підхід, поєднуючи ручне та автоматизоване тестування. Ручне тестування застосовується під час перевірки нової функціональності, дослідницького аналізу системи, тестування інтерфейсу та нестандартних сценаріїв. Автоматизація, своєю чергою, забезпечує швидке виконання регресійних перевірок та підтримку стабільності продукту після внесення змін у програмний код. Така взаємодія дозволяє досягти більш повного покриття перевірок та підвищити загальну якість програмного забезпечення [24].

Актуальність ручного тестування зберігається навіть в умовах стрімкого розвитку сучасних технологій автоматизації. Зі збільшенням складності

програмних продуктів, кількості користувацьких сценаріїв та вимог до зручності використання роль Manual QA Engineer лише посилюється. Сучасний тестувальник виконує не лише функцію пошуку помилок, а й бере участь у формуванні загальної якості продукту, аналізує поведінку системи з точки зору кінцевого користувача та допомагає виявляти проблеми ще до виходу програмного забезпечення у промислову експлуатацію.

Таким чином, ручне тестування залишається необхідною складовою процесу забезпечення якості програмного забезпечення. Воно забезпечує гнучкість перевірок, дозволяє оцінювати систему з позиції користувача та доповнює можливості автоматизованих методів тестування. Використання ручного тестування у поєднанні з сучасними підходами контролю якості дозволяє підвищити стабільність програмного продукту, зменшити кількість дефектів та забезпечити відповідність системи вимогам користувачів і замовників.

Ручне тестування застосовується практично для всіх категорій програмного забезпечення незалежно від масштабу або сфери використання продукту. Воно використовується під час перевірки вебзастосунків, мобільних програм, десктопного програмного забезпечення, корпоративних інформаційних систем, онлайн-сервісів та спеціалізованих програмних рішень. Особливо важливим ручне тестування є для продуктів, орієнтованих на широку аудиторію користувачів, де значення мають не лише технічна працездатність системи, а й зручність інтерфейсу та загальна комфортність використання.

Сучасні тенденції розвитку програмного забезпечення демонструють постійне зростання ролі контролю якості у процесі розробки. Часті оновлення програмних продуктів та необхідність швидкого впровадження нової функціональності потребують регулярного проведення тестування на всіх етапах життєвого циклу системи. За таких умов ручне тестування залишається важливим інструментом оперативної перевірки змін, оцінки поведінки системи та виявлення дефектів, які можуть залишатися непоміченими під час автоматизованих перевірок.

1.4 Аналіз сучасних досліджень і підходів у сфері ручного тестування веб-додатків

Сучасні веб-додатки характеризуються високою складністю, постійним оновленням функціональності та необхідністю швидкого реагування на потреби користувачів. За таких умов забезпечення якості програмного забезпечення стає безперервним процесом, що супроводжує розробку на всіх етапах життєвого циклу продукту. Саме тому питання ефективності ручного тестування веб-додатків залишається актуальним навіть попри стрімкий розвиток засобів автоматизації.

У професійній та науковій літературі значна увага приділяється пошуку підходів, які дозволяють підвищити якість перевірки програмного забезпечення, скоротити час виявлення дефектів та забезпечити стабільність роботи веб-систем у реальних умовах використання. Особливий інтерес викликають методики, що поєднують гнучкість ручного тестування з перевагами автоматизованих засобів контролю якості.

Доцільним є аналіз сучасних досліджень і практичних підходів у сфері ручного тестування веб-додатків, оскільки саме вони формують актуальні уявлення про організацію процесів перевірки програмного забезпечення, роль manual QA engineer у команді розробки та напрями розвитку сучасного тестування. Окремої уваги потребують підходи, орієнтовані на раннє виявлення помилок, безперервне тестування та адаптацію процесів забезпечення якості до умов швидкої розробки веб-систем.

У сучасній науковій та практичній літературі питання тестування програмного забезпечення розглядаються досить активно, проте більшість досліджень зосереджена саме на автоматизації процесів перевірки веб-додатків. Значна увага приділяється автоматизованому регресійному тестуванню, CI/CD-підходам, автоматизації API-перевірок, використанню фреймворків тестування та

інтеграції тестів у процес безперервної розробки. Натомість ручне тестування веб-додатків часто розглядається лише як допоміжний елемент загальної системи забезпечення якості, без окремого детального аналізу його можливостей, обмежень і практичних особливостей застосування.

Подібна ситуація значною мірою пояснюється сучасними тенденціями розвитку ІТ-галузі, орієнтованими на швидкість випуску програмних продуктів, масштабованість процесів тестування та скорочення витрат часу на повторювані перевірки. Автоматизація дозволяє значно прискорити виконання типових тестових сценаріїв, тому саме вона найчастіше стає предметом технічних досліджень і практичних публікацій. Водночас ручне тестування продовжує відігравати важливу роль у перевірці користувацького інтерфейсу, usability, нестандартних сценаріїв використання та загальної логіки роботи веб-додатків, тобто тих компонентів, де людський фактор залишається незамінним. У зв'язку з цим доцільно детальніше розглянути окремі сучасні дослідження та підходи, присвячені організації процесів тестування веб-додатків.

У дослідженні [25] автори розглядають особливості сучасного тестування веб-додатків у середовищі актуальних програмних фреймворків та аналізують підходи до організації процесів перевірки якості програмного забезпечення. Основна увага приділяється співвідношенню ручного та автоматизованого тестування, проблемам трудомісткості ручних перевірок і можливостям автоматизації тестових процесів. Автори підкреслюють, що тестування веб-додатків ускладнюється багаторівневою архітектурою сучасних веб-систем, динамічною зміною станів інтерфейсу та залежністю роботи додатка від навантаження й зовнішнього середовища [25].

У роботі детально розглядається концепція піраміди тестування Майка Кона, що включає модульні, інтеграційні та UI-тести. Автори аналізують особливості кожного рівня тестування, їх переваги та недоліки, а також звертають увагу на необхідність правильної організації співвідношення між різними типами тестів. Окремо досліджуються сучасні підходи до автоматизованого тестування, зокрема TDD та BDD, які дозволяють інтегрувати процес перевірки якості безпосередньо

у цикл розробки програмного забезпечення. Крім цього, розглядається використання шаблону Dependency Injection як засобу підвищення тестованості програмного коду та спрощення модульного тестування [25].

Попри основний акцент на автоматизації, дослідження має безпосередній зв'язок із тематикою ручного тестування веб-додатків. Автори наголошують, що традиційні методи тестування типу black-box і white-box потребують значного обсягу ручної роботи, а окремі аспекти перевірки інтерфейсу користувача, поведінки системи та usability залишаються складними для повної автоматизації. Таким чином, робота підтверджує, що навіть за активного розвитку автоматизованих засобів ручне тестування продовжує залишатися важливою складовою забезпечення якості сучасних веб-додатків [25].

Інше дослідження [26] присвячене аналізу сучасних методів тестування безпеки веб-застосунків та інструментів для проведення тестування на проникнення. Автори розглядають як ручні, так і автоматизовані підходи до перевірки безпеки, приділяючи особливу увагу використанню інструментів автоматизації та технологій штучного інтелекту для виявлення вразливостей. У роботі підкреслюється, що тестування на проникнення традиційно виконується вручну та вимагає високої кваліфікації фахівців, однак сучасні кіберзагрози стимулюють активний розвиток автоматизованих рішень [26].

У межах дослідження проаналізовано низку популярних комерційних платформ для тестування безпеки веб-застосунків, зокрема Burp Suite Professional, Metasploit Pro, Astra, Pentest-Tools.com та інші. Автори порівнюють їх за функціональними можливостями, типами вразливостей, які вони здатні виявляти, а також особливостями інтеграції у процеси DevSecOps та безперервного тестування. Окремо зазначається, що деякі рішення поєднують автоматизоване сканування з ручною перевіркою результатів, що дозволяє підвищити точність виявлення складних проблем безпеки [26].

Водночас дослідження демонструє, що у сфері тестування веб-застосунків питання кібербезпеки є одним із найбільш активно досліджуваних напрямів. На відміну від робіт, присвячених загальному ручному тестуванню веб-додатків,

досліджень у сфері безпекового тестування існує значно більше, що пояснюється постійним зростанням кількості кіберзагроз та необхідністю захисту веб-систем від атак. Це підтверджує актуальність розвитку як автоматизованих, так і ручних методів перевірки безпеки програмного забезпечення [26].

Питання автоматизації тестування безпеки веб-додатків активно досліджуються як у науковій, так і в практичній площині, що підтверджує високу актуальність проблематики кібербезпеки в умовах постійного зростання кількості та складності атак. Значна частина сучасних робіт присвячена підвищенню ефективності автоматизованого виявлення вразливостей, інтеграції засобів тестування у DevSecOps-процеси, а також удосконаленню підходів до аналізу веб-застосунків, API та хмарних середовищ. Особливу увагу дослідники приділяють питанням масштабованості, зниження кількості хибнопозитивних спрацювань та адаптації інструментів до сучасних архітектур програмного забезпечення [27].

Водночас більшість наявних праць зосереджується або на аналізі окремих інструментів, або на практичному тестуванні конкретних сценаріїв використання, без формування узагальненого підходу до їх систематизації та порівняння. Незважаючи на велику кількість досліджень, присвячених саме питанням кібербезпеки та автоматизованого тестування, проблема обґрунтованого вибору інструментів для різних етапів пентесту залишається актуальною. Це зумовлює необхідність проведення структурованого порівняльного аналізу сучасних open-source рішень з урахуванням не лише функціональних можливостей, але й аспектів інтеграції, підтримки сучасних технологій та ефективності використання в реальних умовах [26,27].

У роботі [28] досліджено методи тестування сучасних SPA-вебдодатків, розроблених із використанням React та Redux, із акцентом на застосування графових моделей для автоматизації процесу тестування. Авторка розглядає підхід, за якого логіка роботи веб-застосунку подається у вигляді орієнтованого графа станів і переходів, що дозволяє формалізувати поведінку системи та автоматично генерувати тестові сценарії. У межах дослідження також реалізовано

прототип SPA-додатку та систему автоматизованої генерації тестів із використанням Cypress для E2E-тестування [28].

У результаті проведеного аналізу встановлено, що традиційні підходи та ручне тестування мають обмежену ефективність для складних динамічних SPA-додатків через значну кількість можливих сценаріїв взаємодії, високу залежність від людського фактора та значні часові витрати. Авторка підкреслює, що ручне тестування залишається важливим для перевірки бізнес-логіки та поведінки інтерфейсу, однак автоматизація дозволяє суттєво підвищити швидкість перевірок, уникнути дублювання сценаріїв і забезпечити більш повне покриття функціональності веб-застосунку [28].

Під час експериментального дослідження підтверджено ефективність використання графового підходу для автоматизованого тестування. Встановлено переваги використання Cypress та Jest для інтеграційного, E2E- і модульного тестування порівняно з іншими інструментами. Отримані результати демонструють, що поєднання автоматизованих засобів із ручним тестуванням дозволяє підвищити якість веб-додатків, оптимізувати процес перевірки та забезпечити більш надійне виявлення помилок у сучасних SPA-системах [28].

Автор дослідження [29] досліджує методики тестування веб-додатків з акцентом на їхню ефективність, практичне застосування та доцільність використання в різних сценаріях розробки програмного забезпечення. У дослідженні проаналізовано сучасні підходи до ручного й автоматизованого тестування, розглянуто їхні переваги, недоліки та сфери застосування. Окрему увагу приділено оцінці якості тестування з урахуванням витрат часу, складності впровадження та впливу на загальний цикл розробки веб-додатків. Автор підкреслює, що подібні дослідження є особливо актуальними в умовах постійного ускладнення веб-систем та зростання вимог до їхньої надійності й безпеки [29].

У ході роботи було проведено порівняння ручного та автоматизованого тестування на основі практичних кейсів, реалізовано автоматизовані сценарії перевірки веб-додатків та досліджено економічну доцільність автоматизації. Результати показали, що автоматизоване тестування є найбільш ефективним для

регресійних, навантажувальних і повторюваних перевірок, особливо у великих проєктах, тоді як ручне тестування залишається незамінним для оцінки UX/UI, перевірки нестандартних сценаріїв і виявлення помилок, які складно формалізувати алгоритмічно. Також автор дослідив можливості використання елементів штучного інтелекту для формування тест-кейсів і чек-листів, що дозволяє спростити роботу фахівців із тестування та бізнес-аналітиків [29].

Отримані результати підтвердили, що найбільш ефективним підходом є комбінування ручного та автоматизованого тестування. Автор встановив, що змішаний підхід дозволяє суттєво скоротити витрати часу та ресурсів на тестування, водночас підвищуючи якість програмного продукту. Практичне значення дослідження полягає у формуванні рекомендацій щодо вибору оптимальної методики тестування залежно від типу проєкту, етапу розробки та специфіки функціоналу веб-додатка [29].

Авторка у роботі [30] досліджує підходи до побудови системи мультикритеріального тестування веб-додатків, спрямованої на підвищення якості програмного забезпечення шляхом поєднання різних методів перевірки. У дослідженні проведено аналіз сучасних методів тестування, інструментів автоматизації та технологій безперервної інтеграції, а також розглянуто вразливості веб-додатків і способи їх виявлення. Авторка підкреслює, що ефективне тестування веб-систем потребує комплексного підходу, який поєднує різні типи перевірок та дозволяє адаптувати процес тестування до особливостей конкретного проєкту. У результаті роботи було реалізовано систему тестування веб-додатків із використанням технологій безперервної інтеграції та проведено експериментальне дослідження ефективності різних інструментів тестування на реальних даних. Отримані результати дозволили визначити сильні та слабкі сторони окремих методів і підтвердили доцільність їх комбінування для забезпечення більшої надійності веб-додатків [30].

Проведений огляд наукових досліджень показав, що сучасні роботи у сфері тестування веб-додатків переважно зосереджені на автоматизації процесів перевірки якості програмного забезпечення, інтеграції тестування у DevSecOps-

підходи, використанні графових моделей, засобів безперервної інтеграції, а також застосуванні штучного інтелекту та спеціалізованих інструментів для виявлення вразливостей. Значна кількість досліджень присвячена саме підвищенню швидкості тестування, покриття сценаріїв і зменшенню витрат ресурсів шляхом автоматизації. Це підтверджує високий рівень актуальності проблематики забезпечення якості та безпеки веб-додатків в умовах постійного зростання складності сучасних інформаційних систем.

Водночас аналіз досліджень показав, що питання ручного тестування розглядаються значно рідше та переважно як допоміжний елемент загального процесу тестування. Більшість авторів акцентують увагу саме на автоматизованих підходах, інструментах і методиках, однак майже в кожній роботі підкреслюється неможливість повної відмови від ручного тестування. Це пов'язано з тим, що автоматизовані засоби ефективно працюють для повторюваних, формалізованих і типових сценаріїв, проте мають обмеження при перевірці нестандартної поведінки системи, бізнес-логіки, UX/UI, а також під час аналізу складних взаємодій користувача з веб-додатком.

Окремі дослідження демонструють, що саме комбінування ручного та автоматизованого тестування дозволяє досягти найвищої ефективності процесу забезпечення якості програмного забезпечення. Автори наголошують, що ручне тестування залишається важливим для виявлення дефектів, які складно формалізувати або передбачити за допомогою автоматизованих сценаріїв. Крім того, ручний аналіз часто є необхідним на ранніх етапах розробки, коли функціонал системи швидко змінюється, а автоматизація ще є економічно недоцільною або технічно складною.

Таким чином, результати проведеного огляду підтверджують актуальність дослідження методів ручного тестування веб-додатків. Незважаючи на активний розвиток автоматизованих засобів, ручне тестування залишається невід'ємною складовою процесу забезпечення якості та безпеки веб-застосунків. Недостатня кількість комплексних досліджень, присвячених саме методам ручного тестування, свідчить про наявність наукової та практичної потреби у подальшому

вивченні цього напрямку, зокрема в контексті сучасних веб-технологій, складних інтерфейсів та інтеграції з автоматизованими процесами розробки.

2 ОСОБЛИВОСТІ РУЧНОГО ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

2.1 Архітектурні та функціональні особливості веб-додатків

Сучасні веб-додатки є складними програмними системами, що забезпечують виконання широкого спектра функцій – від обробки користувацьких запитів і взаємодії з базами даних до інтеграції із зовнішніми сервісами та API. Постійний розвиток веб-технологій, збільшення кількості користувацьких сценаріїв та необхідність підтримки різних платформ і пристроїв суттєво ускладнюють процес забезпечення якості таких систем. У цих умовах ручне тестування залишається важливим елементом перевірки коректності роботи веб-додатків, оскільки дозволяє оцінювати не лише функціональність, а й зручність використання, логіку взаємодії та поведінку інтерфейсу в реальних умовах експлуатації.

Особливості архітектури веб-додатків безпосередньо впливають на підходи до їх тестування. Наявність клієнтської та серверної частин, використання динамічного оновлення даних, взаємодія через HTTP/HTTPS-запити, застосування REST API та сучасних frontend-фреймворків формують специфічні вимоги до процесу перевірки якості. Крім того, веб-додатки повинні коректно працювати у різних браузерах, на різних типах пристроїв та за різних умов використання, що створює додаткові труднощі під час тестування.

Важливою особливістю сучасних веб-систем є також активний перехід від класичних багатосторінкових рішень до SPA-додатків, у яких значна частина логіки виконується безпосередньо на стороні клієнта. Це змінює характер взаємодії користувача із системою та ускладнює процес виявлення дефектів, особливо тих, що пов'язані зі станами інтерфейсу, асинхронною обробкою даних або роботою користувацьких сценаріїв. Тому дослідження архітектурних і функціональних особливостей веб-додатків є необхідним для розуміння специфіки ручного тестування та визначення підходів до забезпечення якості сучасних веб-систем.

Сучасні веб-додатки є складними програмними системами, функціонування яких базується на взаємодії великої кількості компонентів, сервісів та мережевих технологій. Архітектура веб-додатку визначає принципи організації його складових, способи обміну даними між ними, а також особливості масштабування, підтримки та забезпечення стабільної роботи системи [31], [32]. Від правильного вибору архітектурного підходу залежить не лише продуктивність веб-додатку, а й складність його тестування, підтримки та подальшого розвитку. Саме тому аналіз архітектурних і функціональних особливостей сучасних веб-додатків є важливою складовою дослідження процесів забезпечення якості програмного забезпечення.

Більшість сучасних веб-додатків базуються на клієнт-серверній моделі взаємодії, у межах якої клієнтська частина відповідає за відображення інтерфейсу користувача та обробку його дій, а серверна – за виконання бізнес-логіки, роботу з базами даних та обробку запитів [31,32]. Користувач взаємодіє із веб-додатком через браузер, який надсилає HTTP або HTTPS-запити до сервера. Після обробки запиту сервер повертає необхідні дані або сформовані сторінки, що забезпечує функціонування веб-системи. Використання HTTPS-протоколу є особливо важливим для захисту передавання даних, оскільки він забезпечує шифрування інформації між клієнтом і сервером, підвищуючи безпеку веб-додатку [31].

Ключовими складовими архітектури веб-додатків є клієнтська частина, серверна частина та рівень доступу до даних [31,32,34]. Клієнтська частина, або frontend, відповідає за формування користувацького інтерфейсу та реалізацію взаємодії користувача із системою. Для її створення переважно використовуються HTML, CSS та JavaScript, а також сучасні бібліотеки та фреймворки. Саме frontend визначає зручність використання веб-додатку, швидкість реакції інтерфейсу та адаптивність під різні типи пристроїв. Серверна частина, або backend, реалізує бізнес-логіку системи, обробляє запити користувачів, забезпечує автентифікацію, авторизацію, обробку даних та взаємодію з базами даних і зовнішніми сервісами [32]. Рівень доступу до даних забезпечує збереження, зміну та отримання інформації, використовуючи системи керування базами даних, що можуть бути реляційними або нереляційними.

У сучасних веб-додатках широко застосовується багаторівнева архітектура, найпоширенішою з яких є трирівнева модель [31,34]. Вона передбачає поділ системи на рівень представлення, рівень бізнес-логіки та рівень доступу до даних. Такий підхід забезпечує розподіл відповідальності між компонентами системи, що спрощує підтримку, масштабування та тестування веб-додатку. Рівень представлення відповідає за взаємодію з користувачем, рівень бізнес-логіки – за реалізацію функціональних можливостей системи, а рівень доступу до даних – за роботу з базами даних та зовнішніми сервісами [31]. Розділення системи на окремі рівні також дозволяє ізолювати помилки та зменшити вплив змін в одному компоненті на інші частини системи.

Важливою особливістю сучасних веб-додатків є використання різних архітектурних підходів залежно від потреб проєкту, очікуваного навантаження та вимог до масштабованості. Одним із базових підходів є монолітна архітектура, у межах якої всі компоненти системи об'єднані в єдину програмну структуру [33,34]. Такий підхід спрощує початкову розробку, розгортання та налагодження системи, оскільки всі модулі знаходяться в одній кодовій базі. Монолітні системи часто використовуються для невеликих проєктів або на ранніх етапах розробки, коли вимоги до масштабування ще не є критичними. Проте зі збільшенням складності веб-додатку виникають труднощі з підтримкою, оновленням та масштабуванням системи, оскільки будь-які зміни можуть впливати на роботу всього застосунку [33].

Доцільним є використання ілюстрації монолітної архітектури для демонстрації структури єдиної взаємопов'язаної системи, у якій усі компоненти працюють у межах одного програмного середовища (рис. 2.1) [33].

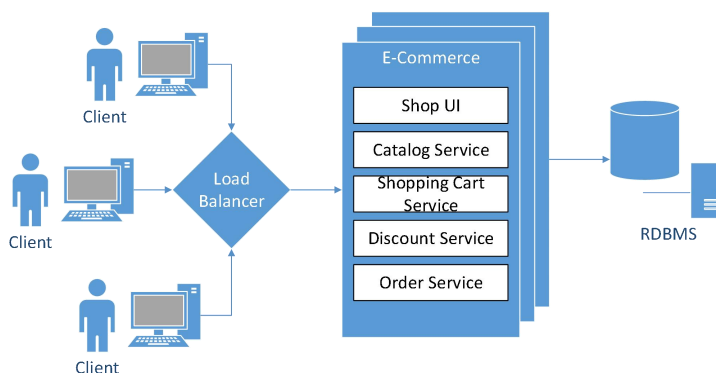


Рис. 2.1 – Монолітна архітектура

Іншим поширеним підходом є мікросервісна архітектура, що базується на поділі системи на окремі незалежні сервіси. Кожен сервіс виконує окрему функцію та може розроблятися, тестуватися і розгортатися незалежно від інших. Мікросервісний підхід забезпечує високу гнучкість, покращує масштабованість системи та дозволяє використовувати різні технології для реалізації окремих компонентів. Водночас така архітектура суттєво ускладнює процес тестування, оскільки необхідно перевіряти не лише окремі сервіси, а й правильність їхньої взаємодії між собою через API та мережеві протоколи. Також збільшується складність пошуку дефектів і контролю стабільності роботи всієї системи [31,33].

Для кращого розуміння принципів побудови мікросервісної архітектури доцільно використати окрему ілюстрацію, що демонструє поділ системи на незалежні сервіси та їх взаємодію між собою (рис. 2.2) [33].

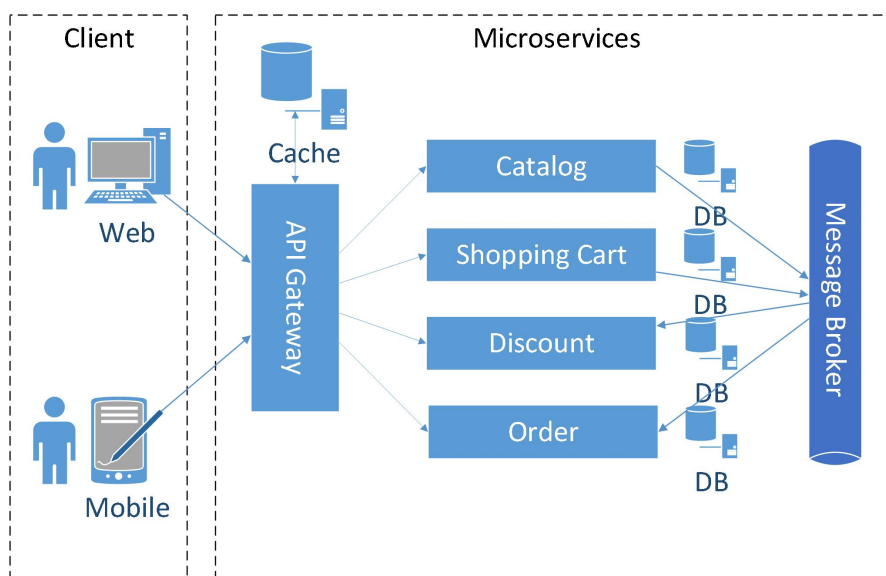


Рис. 2.2 – Мікросервісна архітектура

Окремої уваги заслуговує безсерверна архітектура, у якій значна частина управління інфраструктурою переноситься на хмарного провайдера. У такій моделі розробник зосереджується переважно на реалізації функціональності системи, а масштабування, балансування навантаження та підтримка серверної інфраструктури виконуються автоматично. Основними перевагами безсерверного підходу є швидке масштабування, зниження витрат на підтримку серверів та спрощення розгортання веб-додатків. Водночас така архітектура може ускладнювати процес тестування через обмежений контроль над серверним середовищем та залежність від зовнішніх хмарних сервісів [31,33].

Ілюстрація безсерверної архітектури може бути використана для демонстрації принципу взаємодії веб-додатку з хмарною інфраструктурою та автоматизованого керування ресурсами (рис. 2.3) [33].

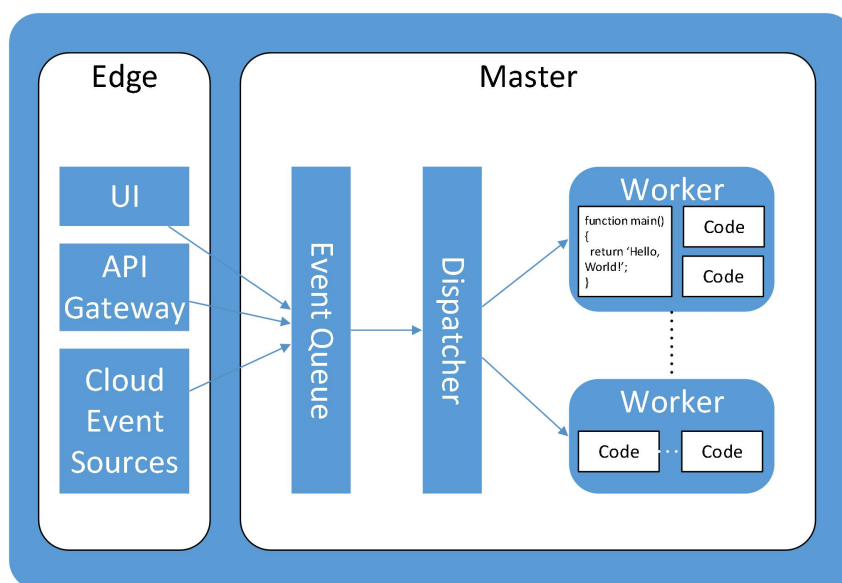


Рис. 2.3 – Безсерверна архітектура

Значний вплив на функціональні особливості веб-додатків має спосіб організації інтерфейсу користувача. Сучасні системи можуть реалізовуватися як багатосторінкові або односторінкові веб-додатки [31,32]. Багатосторінкові веб-додатки передбачають повне оновлення сторінки під час переходу між різними розділами системи. Такий підхід є традиційним і забезпечує кращу пошукову

оптимізацію, проте може негативно впливати на швидкість взаємодії користувача з інтерфейсом. Односторінкові веб-додатки, навпаки, забезпечують динамічне оновлення контенту без повного перезавантаження сторінки, що створює більш швидкий та інтерактивний користувацький досвід. Водночас такі системи мають складнішу структуру клієнтської частини та вимагають ретельнішого тестування роботи інтерфейсу, навігації та взаємодії із сервером [31].

Сучасні веб-додатки також активно використовують API для взаємодії між окремими сервісами та зовнішніми системами [31,32]. API забезпечують передачу даних між frontend та backend компонентами, а також інтеграцію з платіжними системами, сервісами автентифікації, аналітики та іншими зовнішніми платформами. Наявність великої кількості інтеграцій підвищує функціональні можливості веб-додатків, проте одночасно ускладнює процес забезпечення їхньої стабільності та безпеки. У процесі тестування необхідно перевіряти правильність обробки запитів, коректність передавання даних, стабільність роботи інтеграцій та реакцію системи на помилки зовнішніх сервісів [35].

Однією з важливих характеристик сучасних веб-додатків є підтримка адаптивного дизайну та сумісності з різними браузерами й пристроями. Користувачі можуть працювати із системою через персональні комп'ютери, планшети або смартфони, використовуючи різні браузери та операційні системи. У зв'язку з цим веб-додаток повинен забезпечувати коректне відображення інтерфейсу та стабільну роботу функціональних елементів у різних середовищах. Особливої уваги потребує перевірка адаптивності інтерфейсу, роботи елементів навігації, форм введення даних та мультимедійного контенту на різних розмірах екранів [31,35].

Важливу роль у забезпеченні стабільності роботи веб-додатків відіграють додаткові інфраструктурні компоненти, серед яких балансувальники навантаження, кешування, CDN та системи черг повідомлень [31,32]. Балансувальники навантаження розподіляють запити між кількома серверами для запобігання перевантаженню системи. Кешування дозволяє зменшити час відповіді сервера шляхом збереження часто використовуваних даних, а CDN

забезпечують швидке завантаження статичних ресурсів із найближчих до користувача серверів. Наявність таких компонентів позитивно впливає на продуктивність веб-додатків, однак створює додаткові особливості тестування, пов'язані з перевіркою стабільності роботи системи за високих навантажень та коректністю оновлення даних [31].

Архітектурні особливості веб-додатків безпосередньо впливають на процес забезпечення якості програмного забезпечення та організацію тестування. Від структури системи, способу взаємодії компонентів, використаних технологій та рівня складності залежить перелік потенційних дефектів, складність їх виявлення та вибір методів тестування. Сучасні веб-додатки характеризуються високою інтерактивністю, великою кількістю інтеграцій та необхідністю підтримки різних платформ і середовищ виконання, що суттєво підвищує вимоги до якості тестування. Саме тому розуміння архітектурних і функціональних особливостей веб-додатків є необхідною передумовою для ефективної організації процесу ручного тестування та забезпечення належного рівня якості програмних систем [35].

Отже, сучасні веб-додатки характеризуються складною багаторівневою архітектурою, використанням клієнт-серверної взаємодії, інтеграцією з базами даних та зовнішніми сервісами. Особливості побудови frontend і backend частин, застосування API, адаптивного дизайну та хмарних технологій безпосередньо впливають на продуктивність, безпеку та стабільність роботи веб-систем. Крім цього, архітектура веб-додатку визначає способи організації даних, обробки запитів користувачів та взаємодії між окремими компонентами системи.

У ході дослідження встановлено, що різні архітектурні підходи, зокрема монолітна, мікросервісна та безсерверна архітектури, мають власні переваги й обмеження, які визначають особливості розробки, підтримки та тестування веб-додатків. Сучасні веб-системи також повинні забезпечувати коректну роботу на різних пристроях, підтримувати сумісність із браузерами та гарантувати належний рівень зручності використання для користувачів. Усе це суттєво підвищує вимоги до забезпечення якості програмного забезпечення.

Таким чином, архітектурні та функціональні особливості веб-додатків створюють основу для подальшого дослідження процесів ручного тестування. Зростання складності веб-систем, збільшення кількості інтеграцій та високі вимоги до надійності й безпеки підтверджують актуальність детального аналізу підходів до перевірки якості сучасних веб-додатків.

2.2 Основні напрями ручного тестування веб-додатків

Ручне тестування веб-додатків є важливою складовою процесу забезпечення якості програмного забезпечення, оскільки дозволяє оцінити не лише технічну коректність роботи системи, а й зручність її використання, логічність поведінки інтерфейсу та відповідність очікуванням користувачів. Попри активний розвиток автоматизованого тестування, значна кількість задач у сфері перевірки веб-додатків і надалі потребує безпосередньої участі тестувальника. Особливо це стосується перевірки інтерфейсів, навігації, візуального відображення інформації та оцінки загального користувацького досвіду.

Сучасні веб-додатки характеризуються високою інтерактивністю, великою кількістю функціональних елементів та необхідністю підтримки різних платформ, браузерів і типів пристроїв. У зв'язку з цим процес ручного тестування охоплює широкий спектр напрямів, кожен із яких орієнтований на перевірку окремих аспектів роботи веб-системи. Під час тестування необхідно враховувати правильність реалізації функціональних можливостей, стабільність взаємодії користувача з інтерфейсом, коректність обробки даних, а також зручність і доступність використання веб-додатку.

Важливе місце у процесі ручного тестування займає перевірка роботи форм введення даних, механізмів навігації, авторизації користувачів, адаптивності інтерфейсу та сумісності веб-додатка з різними середовищами виконання. Крім цього, значна увага приділяється оцінці коректності відображення інформації, локалізації інтерфейсу, підтримки різних мов та базовим аспектам безпеки веб-систем. Наявність великої кількості можливих сценаріїв взаємодії користувача із

системою суттєво ускладнює процес тестування та підвищує вимоги до повноти перевірки функціональності веб-додатків.

Таким чином, ручне тестування веб-додатків охоплює комплекс різних напрямів перевірки якості програмного забезпечення, що дозволяють виявляти дефекти, недоліки інтерфейсу та проблеми взаємодії користувача із системою. Дослідження основних напрямів ручного тестування є необхідним для розуміння особливостей забезпечення якості сучасних веб-додатків та визначення найбільш ефективних підходів до їх перевірки.

Ручне тестування веб-додатків є одним із ключових етапів забезпечення якості програмного забезпечення, оскільки саме воно дозволяє оцінити роботу системи з позиції реального користувача та виявити помилки, які складно або неможливо знайти автоматизованими засобами [39]. На відміну від автоматизованого тестування, ручна перевірка базується на безпосередній взаємодії тестувальника з веб-додатком, аналізі поведінки інтерфейсу, перевірці логіки виконання функцій, оцінці зручності використання та пошуку нестандартних сценаріїв взаємодії користувача із системою. Такий підхід особливо важливий для веб-додатків, оскільки вони працюють у різних браузерах, операційних системах, мережевих умовах і на різних типах пристроїв, що значно підвищує ризик появи помилок і несумісностей [35]. У сучасних умовах ручне тестування розглядається не лише як етап перевірки готового продукту, а як безперервний процес контролю якості, який супроводжує розробку веб-додатка протягом усього життєвого циклу програмного забезпечення [36].

Одним із найважливіших напрямів ручного тестування веб-додатків є функціональне тестування, метою якого є перевірка правильності роботи всіх функцій системи відповідно до технічних вимог і очікувань користувачів [36]. У межах такого тестування перевіряється робота навігації, форм введення даних, процесів авторизації та реєстрації, механізмів пошуку, обробки замовлень, передачі даних між клієнтською та серверною частинами, а також коректність виконання бізнес-логіки [35]. Для веб-додатків особливо важливою є перевірка правильності взаємодії між фронтендом і бекендом, оскільки навіть незначні

помилки у роботі API або серверної логіки можуть призвести до порушення функціонування всієї системи. Функціональне тестування передбачає не лише перевірку позитивних сценаріїв використання, коли користувач виконує очікувані дії, але й негативне тестування, під час якого до системи вводяться некоректні або неочікувані дані з метою перевірки стійкості веб-додатка до помилок [36]. Саме завдяки ручному тестуванню можна оцінити, наскільки зрозумілими є повідомлення про помилки, чи коректно система реагує на неправильні дії користувача та чи не виникають критичні збої під час нестандартної взаємодії із системою.

Важливим напрямом ручного тестування є тестування користувацького інтерфейсу та usability testing, які орієнтовані на оцінку зручності використання веб-додатка, логічності його структури та якості взаємодії користувача з інтерфейсом [36]. Веб-додаток повинен не лише коректно працювати, а й забезпечувати комфортну та інтуїтивно зрозумілу взаємодію для користувача. Саме тому під час ручного тестування перевіряється зрозумілість навігації, доступність основних функцій, розташування елементів інтерфейсу, читабельність тексту, зручність роботи з формами та швидкість виконання типових дій користувача [38]. Особливу увагу приділяють перевірці сценаріїв, які безпосередньо впливають на користувацький досвід, наприклад процесу оформлення замовлення, реєстрації нового користувача або відновлення пароля. Тестувальник оцінює не лише технічну працездатність функцій, але й те, наскільки зрозуміло система взаємодіє з користувачем, чи не перевантажений інтерфейс зайвими елементами та чи не виникає труднощів під час виконання основних дій [36]. У багатьох випадках саме usability testing дозволяє виявити проблеми, які не є технічними помилками, але негативно впливають на ефективність використання веб-додатка та задоволеність користувачів.

Не менш важливим напрямом є тестування сумісності, яке забезпечує перевірку коректної роботи веб-додатка у різних середовищах виконання [36]. Сучасні веб-системи використовуються на великій кількості пристроїв – від персональних комп'ютерів до смартфонів і планшетів – а також у різних

браузерах та операційних системах. Через відмінності у реалізації веб-стандартів браузерами або особливості роботи мобільних платформ одна й та сама сторінка може відображатися або функціонувати по-різному [38]. Саме тому під час ручного тестування перевіряється кросбраузерна та кросплатформна сумісність веб-додатка, адаптивність інтерфейсу, правильність масштабування елементів, робота JavaScript-скриптів, форм введення та навігації на різних типах екранів [36]. Окремо перевіряється коректність роботи веб-додатка на мобільних пристроях, оскільки значна частина користувачів взаємодіє із веб-ресурсами саме через смартфони [38]. При цьому тестувальник оцінює не лише візуальне відображення інтерфейсу, а й зручність використання touch-елементів, швидкість роботи та стабільність функціонування додатка у мобільному середовищі.

У процесі ручного тестування веб-додатків важливе місце займає localization testing, або тестування локалізації, яке передбачає перевірку коректності адаптації веб-додатка до різних мовних та регіональних особливостей [36]. Локалізація охоплює не лише переклад текстових елементів інтерфейсу, а й правильне відображення дат, часу, валют, форматів чисел, повідомлень системи та інших компонентів, що залежать від мовного середовища користувача. Під час тестування локалізації перевіряється відсутність помилок перекладу, правильність розташування тексту в інтерфейсі, відповідність термінології предметній області та збереження функціональності після зміни мови системи. Особливо важливо контролювати ситуації, коли текст іншою мовою має більшу довжину та може порушувати структуру інтерфейсу або спричиняти некоректне відображення елементів сторінки [36]. Для міжнародних веб-додатків localization testing є необхідним етапом забезпечення якості, оскільки помилки локалізації безпосередньо впливають на довіру користувачів та зручність використання системи.

Окремим напрямом є accessibility testing, або тестування доступності веб-додатків, яке спрямоване на перевірку можливості використання системи людьми з різними обмеженнями [35]. У сучасній веб-розробці доступність розглядається як один із важливих показників якості програмного забезпечення, оскільки веб-

додаток повинен бути придатним для використання максимально широким колом користувачів. Під час ручного тестування перевіряється доступність навігації за допомогою клавіатури, правильність роботи екранних читачів, наявність альтернативних текстів для зображень, контрастність елементів інтерфейсу та зручність сприйняття контенту користувачами з порушеннями зору чи моторики [38]. Таке тестування часто потребує безпосередньої участі тестувальника, оскільки автоматизовані засоби не завжди здатні оцінити реальну зручність взаємодії користувача із системою. Забезпечення доступності веб-додатків є не лише технічною вимогою, але й важливим аспектом створення інклюзивного цифрового середовища.

Значну роль у процесі ручного тестування відіграє базова перевірка безпеки веб-додатків, оскільки сучасні веб-системи часто працюють із конфіденційними даними користувачів і можуть бути об'єктом різноманітних атак [35]. На базовому рівні тестування безпеки передбачає перевірку коректності авторизації та автентифікації користувачів, захисту форм введення даних, обмеження доступу до окремих функцій системи та стійкості до найпоширеніших типів атак, таких як SQL-ін'єкції або міжсайтовий скриптинг [36]. Тестувальник перевіряє, чи неможливо отримати доступ до закритих сторінок без авторизації, чи правильно обробляються сесії користувачів, чи безпечно передаються дані між клієнтом і сервером, а також чи коректно працюють механізми відновлення паролів і зміни персональної інформації. Важливим аспектом є також перевірка роботи SSL-сертифікатів, безпечності адміністративної панелі та коректності повідомлень про помилки, які не повинні розкривати внутрішню структуру системи [38]. Хоча повноцінне тестування безпеки часто потребує спеціалізованих інструментів та окремих методик, базова ручна перевірка дозволяє виявити значну кількість критичних вразливостей ще на ранніх етапах розробки.

Під час ручного тестування веб-додатків важливою складовою є перевірка роботи форм введення даних, навігації та механізмів взаємодії користувача з інтерфейсом [36]. Форми є одним із найважливіших елементів веб-систем, оскільки саме через них користувачі виконують реєстрацію, авторизацію,

оформлення замовлень, надсилання повідомлень та інші ключові дії. Тестувальник перевіряє правильність валідації полів, коректність обробки обов'язкових та необов'язкових даних, відображення повідомлень про помилки, а також поведінку системи при введенні некоректних або нестандартних значень [36]. Особливу увагу приділяють перевірці навігації, оскільки складна або нелогічна структура сторінок негативно впливає на користувацький досвід та може ускладнювати виконання навіть базових дій [38]. Крім того, тестувальники аналізують поведінку системи при різних сценаріях взаємодії, наприклад при одночасному відкритті кількох вкладок браузера, оновленні сторінки під час виконання операцій або втраті з'єднання із сервером. Такі перевірки дозволяють оцінити стабільність роботи веб-додатка та його готовність до реального використання у різних умовах.

Ручне тестування також охоплює *exploratory testing* та *ad-hoc testing*, які дозволяють виявляти помилки за межами формально описаних тестових сценаріїв. На відміну від строго структурованих тест-кейсів, *exploratory testing* базується на досвіді, інтуїції та аналітичному мисленні тестувальника, який самостійно визначає напрям перевірки та експериментує із різними сценаріями використання системи. Такий підхід є особливо ефективним для складних веб-додатків із великою кількістю інтерактивних елементів, де не всі можливі варіанти взаємодії можуть бути передбачені заздалегідь. *Ad-hoc testing*, у свою чергу, дозволяє швидко оцінити загальну стабільність функціоналу без попередньої підготовки детальної документації. Саме завдяки цим видам тестування часто виявляються нестандартні дефекти, пов'язані з неочікуваною поведінкою користувачів або комбінаціями дій, які складно формалізувати у вигляді класичних тестових сценаріїв [36,39].

Таким чином, ручне тестування веб-додатків охоплює широкий спектр напрямів, спрямованих на перевірку функціональності, стабільності, сумісності, безпеки та зручності використання веб-систем. Кожен із напрямів виконує важливу роль у процесі забезпечення якості програмного забезпечення та дозволяє оцінити веб-додаток не лише з технічної точки зору, а й з позиції

кінцевого користувача. Незважаючи на активний розвиток автоматизованого тестування, ручна перевірка залишається необхідною складовою процесу QA, оскільки саме вона дозволяє враховувати людський фактор, оцінювати реальний користувацький досвід та виявляти помилки, які важко формалізувати або автоматизувати [39].

Отже, ручне тестування веб-додатків охоплює різні напрями перевірки якості веб-систем – від функціональності та сумісності до безпеки й зручності використання. Комплексне застосування цих підходів дозволяє своєчасно виявляти дефекти, покращувати користувацький досвід і забезпечувати стабільну роботу веб-додатка.

2.3 Методи та техніки ручного тестування веб-додатків

Ручне тестування веб-додатків базується не лише на перевірці окремих функцій системи, але й на використанні спеціалізованих методів і технік, що дозволяють організувати процес тестування більш системно та ефективно. Застосування відповідних підходів допомагає оптимізувати кількість тестових перевірок, підвищити якість покриття функціональності та збільшити ймовірність виявлення дефектів на різних етапах роботи веб-додатка.

У процесі ручного тестування використовуються як формалізовані техніки, що ґрунтуються на аналізі вхідних даних, логіки роботи системи та можливих сценаріїв поведінки користувача, так і більш гнучкі підходи, орієнтовані на практичний досвід тестувальника та дослідницький підхід до перевірки програмного забезпечення. Кожна техніка має власну сферу застосування, переваги та особливості використання залежно від типу веб-додатка, складності функціоналу та поставлених цілей тестування.

Далі пропонується розглянути основні методи та техніки ручного тестування веб-додатків, їх практичне значення та особливості застосування під час перевірки функціональності веб-систем.

Методи та техніки ручного тестування веб-додатків є важливим інструментом забезпечення якості програмного забезпечення, оскільки дозволяють систематизувати процес перевірки та підвищити ефективність виявлення дефектів. На відміну від загальних напрямів тестування, вони орієнтовані безпосередньо на підходи до створення тестових сценаріїв, аналізу вхідних даних і моделювання поведінки користувача. Застосування різних технік тест-дизайну дозволяє оптимізувати кількість перевірок, забезпечити достатній рівень покриття функціональності та підвищити стабільність веб-додатків [36,39].

Однією з найбільш поширених технік є *equivalence partitioning* (розбиття на класи еквівалентності). Відповідно до цього підходу, вхідні дані поділяються на окремі групи, у межах яких система повинна поводитися однаково [36]. Це дозволяє скоротити кількість тестових випадків без втрати ефективності тестування. Наприклад, якщо веб-додаток приймає вік користувача лише у межах від 18 до 65 років, тестувальник формує класи допустимих і недопустимих значень та перевіряє лише декілька представників кожної групи. Такий підхід допомагає виявляти помилки обробки даних і зменшує обсяг повторюваних перевірок. Особливо ефективною ця техніка є під час тестування форм реєстрації, авторизації або введення персональних даних у веб-системах [39].

Тісно пов'язаною з попереднім методом є *boundary value analysis* (аналіз граничних значень). Дана техніка базується на припущенні, що значна кількість помилок виникає саме на межах допустимих діапазонів [36]. У процесі тестування перевіряються мінімальні, максимальні та сусідні до них значення. Наприклад, якщо пароль повинен містити від 8 до 20 символів, доцільно протестувати введення 7, 8, 20 та 21 символу. Аналіз граничних значень широко застосовується під час перевірки веб-форм, систем бронювання, фільтрів пошуку та механізмів обмеження доступу. Практика показує, що саме на межових значеннях часто

виникають помилки валідації, які можуть призводити до некоректної роботи веб-додатка або появи вразливостей безпеки [35].

Для перевірки складних логічних залежностей у веб-додатках використовується decision table testing (тестування таблиць рішень). Ця техніка дозволяє структурувати різні комбінації умов та очікуваних результатів у вигляді таблиць. Особливо корисною вона є для веб-систем, у яких поведінка залежить від багатьох параметрів: ролей користувачів, типу облікового запису, статусу замовлення, способу оплати чи доступності певних функцій [37]. Наприклад, інтернет-магазин може по-різному обробляти оформлення замовлення залежно від наявності товару, обраного способу доставки та підтвердження оплати. Використання таблиць рішень дозволяє уникнути пропуску важливих комбінацій умов та забезпечити більш системний підхід до тестування бізнес-логіки веб-додатка.

Ще однією важливою технікою є state transition testing (тестування переходів станів). Вона застосовується у випадках, коли поведінка системи залежить від її поточного стану та послідовності дій користувача [39]. У веб-додатках це характерно для процесів авторизації, оформлення замовлень, роботи з кошиком або workflow-систем. Наприклад, замовлення може перебувати у станах «створене», «оплачене», «відправлене» або «скасоване». Завдання тестувальника полягає у перевірці правильності переходів між цими станами та недопущенні нелогічних операцій, таких як повторна оплата вже завершеного замовлення. Тестування переходів станів дозволяє оцінити коректність реалізації бізнес-процесів і забезпечити цілісність роботи системи [35].

Важливу роль у ручному тестуванні веб-додатків відіграє exploratory testing (дослідницьке тестування). На відміну від формалізованих методів, цей підхід передбачає одночасне вивчення системи, створення тестових сценаріїв та виконання перевірок [36]. Тестувальник діє більш гнучко, спираючись на власний досвід, інтуїцію та аналіз поведінки користувача. Exploratory testing особливо ефективно у випадках, коли документація є неповною або функціональність веб-додатка часто змінюється. Завдяки цьому підходу можна виявляти нестандартні

дефекти, які складно передбачити за допомогою класичних тестових сценаріїв. Наприклад, тестувальник може перевіряти поведінку системи під час швидкого перемикання між сторінками, одночасного відкриття декількох вкладок або некоректного порядку виконання дій користувача [39].

Подібною за характером є техніка *ad-hoc testing*, яка ґрунтується на імпровізації та не потребує заздалегідь підготовлених тестових випадків [36]. Її основною метою є швидке виявлення помилок у найбільш критичних або нестабільних компонентах системи. У процесі такого тестування спеціаліст намагається навмисно створити нестандартні умови роботи веб-додатка: вводить некоректні символи, виконує повторні натискання кнопок або перериває операції під час обробки даних сервером. Попри відсутність формальної структури, *ad-hoc testing* часто дозволяє знаходити дефекти, які залишаються непоміченими під час виконання стандартних тестових сценаріїв [39].

Для стандартизації процесу перевірки функціональності широко використовується *checklist-based testing*. Дана техніка базується на використанні заздалегідь підготовлених чек-листів, які містять перелік обов'язкових перевірок [38]. Чек-листи дозволяють забезпечити єдиний підхід до тестування та спростити контроль виконання тестових завдань. У веб-додатках вони можуть включати перевірку навігації, роботи форм, коректності посилань, адаптивності інтерфейсу, повідомлень про помилки та функціонування інтерактивних елементів. Використання *checklist-based testing* є особливо корисним під час регресійного тестування, коли необхідно швидко переконатися, що внесені зміни не вплинули на вже реалізовану функціональність [36].

Ще однією поширеною технікою ручного тестування є *error guessing*. Вона базується на досвіді тестувальника та його здатності прогнозувати потенційно проблемні місця веб-додатка [39]. На відміну від формалізованих методів, *error guessing* не має чітко визначених правил і значною мірою залежить від професійної підготовки спеціаліста. Наприклад, тестувальник може припустити, що система некоректно оброблятиме надто довгі рядки, спеціальні символи, нестандартні формати введення або повторні запити до сервера. Завдяки знанню

типових дефектів та особливостей веб-систем така техніка дозволяє швидко знаходити критичні помилки, що можуть впливати на стабільність або безпеку додатка [35].

Отже, методи та техніки ручного тестування забезпечують системний підхід до перевірки веб-додатків та дозволяють підвищити ефективність процесу виявлення дефектів. Формалізовані техніки, такі як *equivalence partitioning*, *boundary value analysis* чи *decision table testing*, допомагають оптимізувати тестове покриття та структурувати перевірки. Водночас *exploratory testing*, *ad-hoc testing* та *error guessing* забезпечують гнучкість і дозволяють знаходити нестандартні помилки, які складно передбачити на етапі планування. Комплексне використання різних методів тестування сприяє підвищенню якості веб-додатків, стабільності їх роботи та покращенню користувацького досвіду [35–39].

Отже, використання різних методів і технік ручного тестування дозволяє більш ефективно перевіряти функціональність веб-додатків, виявляти дефекти та забезпечувати стабільність роботи системи. Поєднання формалізованих підходів і практичного досвіду тестувальника сприяє підвищенню якості тестування та покращенню користувацького досвіду.

2.4 Типові дефекти веб-додатків та способи їх виявлення

Веб-додатки є складними програмними системами, у яких взаємодіють різноманітні програмні компоненти, користувацькі інтерфейси, серверна логіка та зовнішні сервіси. У процесі їх розробки та експлуатації можуть виникати численні дефекти, що впливають на коректність функціонування, продуктивність, безпеку та зручність використання системи. Наявність навіть незначних помилок здатна негативно позначатися на користувацькому досвіді, стабільності роботи веб-додатка та довірі користувачів до програмного продукту.

Особливістю веб-додатків є залежність їх роботи від великої кількості зовнішніх факторів, зокрема типу браузера, операційної системи, характеристик пристрою та мережевого середовища. У зв'язку з цим під час тестування важливо враховувати не лише функціональну коректність роботи системи, але й особливості взаємодії користувача з інтерфейсом, адаптивність сторінок, правильність відображення інформації та обробки даних у різних умовах використання.

Ручне тестування дозволяє виявляти широкий спектр дефектів, які складно або неможливо повністю визначити автоматизованими засобами. Завдяки аналізу поведінки веб-додатка з позиції кінцевого користувача забезпечується можливість оцінити не лише технічну правильність роботи системи, а й загальну якість взаємодії з веб-ресурсом.

Одним із найбільш поширених напрямів виявлення дефектів веб-додатків під час ручного тестування є перевірка користувацького інтерфейсу та зручності взаємодії із системою. Помилки, пов'язані з UI та UX, безпосередньо впливають на сприйняття веб-додатка користувачем, ефективність виконання ним цільових дій та загальний рівень задоволеності від роботи із системою. Під UI (User Interface) розуміють візуальну частину веб-додатка – структуру сторінок, кнопки, меню, кольорову гаму, елементи навігації та інші компоненти інтерфейсу. UX (User Experience), своєю чергою, характеризує загальний користувацький досвід взаємодії із системою, зокрема простоту використання, логічність навігації, швидкість виконання дій та зручність роботи із функціоналом веб-додатка [40].

Під час тестування веб-додатків особлива увага приділяється виявленню дефектів, які ускладнюють взаємодію користувача із системою або створюють негативний досвід використання. Однією з типових проблем є надмірне ускладнення інтерфейсу. Використання великої кількості анімацій, нестандартних елементів навігації або перевантажених сторінок може негативно впливати на швидкість пошуку необхідної інформації та виконання основних дій користувачем. У процесі ручного тестування подібні дефекти виявляються

шляхом перевірки сценаріїв навігації, оцінювання логічності розташування елементів та аналізу кількості дій, необхідних для досягнення певної функції [40].

Поширеною UX-проблемою також є відсутність чіткої орієнтації користувача на цільову дію. У веб-додатках це може проявлятися у незрозумілому розташуванні кнопок, відсутності візуальних акцентів або надмірно нав'язливих елементів інтерфейсу. Під час ручного тестування перевіряється, наскільки швидко користувач може знайти необхідну функцію, оформити замовлення, зареєструватися чи виконати іншу цільову дію без додаткових пояснень або складних переходів між сторінками [40].

Окрему увагу приділяють перевірі послідовності інтерфейсу веб-додатка. Однією з характерних помилок є різке змінення стилю, структури або логіки роботи окремих сторінок, через що користувач може втрачати орієнтацію під час навігації. У процесі тестування оцінюється єдність дизайну, стабільність розташування елементів меню, кнопок та форм на різних сторінках системи [40].

Важливою складовою ручного тестування є також перевірка адаптивності веб-додатка. Інтерфейс повинен коректно відображатися на різних пристроях, екранах та у різних браузерях. Помилки адаптивності можуть проявлятися у зміщенні елементів, некоректному масштабуванні зображень, зміні кольорів або втраті функціональності окремих компонентів інтерфейсу. Для виявлення таких дефектів тестування проводиться у різних браузерях та на пристроях із різними параметрами екрана [40].

Крім візуальних та навігаційних проблем, під час ручного тестування веб-додатків перевіряється коректність обробки помилок та відображення системних повідомлень. Одним із ключових аспектів є аналіз HTTP-кодів відповіді сервера, які сигналізують про стан виконання запиту. Коди класу 4xx вказують на помилки з боку клієнта, а коди 5xx – на проблеми сервера [41].

Серед найбільш поширених помилок веб-додатків виділяють HTTP 400 (Bad Request), яка виникає через некоректний або пошкоджений запит користувача. Причинами можуть бути неправильна URL-адреса, пошкоджені cookie або некоректно передані дані форми. Під час ручного тестування подібні дефекти

виявляються шляхом введення некоректних значень, перевірки роботи форм та тестування поведінки системи при помилкових запитах [41].

Не менш поширеною є помилка HTTP 404 (Not Found), що виникає у випадках відсутності сторінки за вказаною адресою. Під час тестування перевіряється коректність внутрішніх посилань, навігаційного меню та обробки переходів на неіснуючі сторінки. Окремо оцінюється зручність сторінки помилки 404 та наявність можливостей повернення до основного функціоналу системи [41].

До критичних дефектів належать серверні помилки HTTP 500 (Internal Server Error), HTTP 502 (Bad Gateway) та HTTP 503 (Service Unavailable). Такі помилки можуть бути пов'язані із проблемами серверної логіки, перевантаженням системи або неправильними налаштуваннями веб-сервера. У процесі ручного тестування перевіряється поведінка веб-додатка при високому навантаженні, виконанні некоректних дій користувача або втраті зв'язку між серверними компонентами [41].

Таким чином, ручне тестування UI/UX та перевірка типових помилок веб-додатків дозволяють виявити значну кількість дефектів, які безпосередньо впливають на стабільність роботи системи, зручність використання та ефективність взаємодії користувача із веб-додатком.

Ще одним важливим напрямом ручного тестування веб-додатків є перевірка адаптивності, кросбраузерної сумісності, usability та accessibility. Наявність дефектів у цих аспектах безпосередньо впливає на доступність веб-системи для користувачів, зручність її використання та стабільність роботи на різних пристроях і програмних платформах.

Адаптивність веб-додатка визначає його здатність коректно відображатися на різних типах пристроїв – персональних комп'ютерах, ноутбуках, планшетах і смартфонах незалежно від роздільної здатності екрана. Адаптивний дизайн забезпечує автоматичне підлаштування структури сторінки, блоків, тексту та графічних елементів під параметри вікна браузера, що дозволяє користувачеві комфортно взаємодіяти із системою [42].

Під час ручного тестування адаптивності перевіряється коректність відображення веб-додатка на різних пристроях та в різних орієнтаціях екрана. Типовими дефектами є зміщення елементів інтерфейсу, накладання блоків один на одного, обрізання тексту, неправильне масштабування зображень, поява горизонтальної прокрутки або недоступність окремих функцій на мобільних пристроях. Окремо тестується стабільність роботи меню, форм введення даних та кнопок при зміні розмірів вікна браузера [42].

Важливим аспектом є також кросбраузерне тестування, метою якого є перевірка однакової роботи веб-додатка у різних браузерах та їх версіях. Через відмінності у механізмах рендерингу HTML, CSS і JavaScript окремі елементи можуть працювати некоректно або відображатися по-різному. У процесі ручного тестування здійснюється перевірка функціональності веб-додатка у популярних браузерах, зокрема Google Chrome, Mozilla Firefox, Microsoft Edge та Safari. Особлива увага приділяється відображенню інтерфейсу, роботі інтерактивних елементів, форм, анімацій та навігації між сторінками.

Окрему категорію дефектів становлять usability-проблеми, пов'язані із загальною зручністю використання веб-додатка. До них належать складна навігація, перевантажені інтерфейси, незрозумілі повідомлення системи, неочевидне розташування функціональних елементів або надмірна кількість дій для виконання базових операцій. Подібні недоліки часто не є технічними помилками у класичному розумінні, проте вони суттєво впливають на ефективність взаємодії користувача із системою. Ручне тестування usability базується на моделюванні реальної поведінки користувачів та оцінюванні зручності виконання типових сценаріїв роботи.

Під час тестування сучасних веб-додатків дедалі більшого значення набуває accessibility – доступність веб-ресурсу для людей із різними фізичними або когнітивними обмеженнями. Перевірка accessibility передбачає оцінювання контрастності тексту, можливості навігації за допомогою клавіатури, наявності альтернативного тексту для зображень, коректності роботи екранних озвучувачів та зрозумілості структури сторінок. Недостатня увага до інклюзивності веб-

додатків може призводити до обмеження доступу окремих категорій користувачів до інформації та функціональних можливостей системи [43].

Таким чином, тестування адаптивності, кросбраузерної сумісності, usability та accessibility дозволяє виявляти дефекти, які впливають не лише на технічну працездатність веб-додатка, а й на комфорт, доступність та ефективність його використання різними категоріями користувачів.

Кібербезпека є важливою складовою якості веб-додатків, оскільки сучасні веб-системи постійно піддаються різноманітним кіберзагрозам та атакам. Одним із найважливіших міжнародних орієнтирів у сфері безпеки веб-додатків є проєкт OWASP (Open Web Application Security Project), який визначає найбільш критичні ризики та вразливості веб-систем. Особливу роль відіграє список OWASP Top 10, що містить найпоширеніші та найнебезпечніші дефекти безпеки веб-додатків [44–46].

Одними з найпоширеніших дефектів веб-додатків є вразливості, пов'язані з некоректною обробкою введених даних, помилками автентифікації, порушенням контролю доступу та неправильними налаштуваннями безпеки [44–46].

Однією з найбільш небезпечних вразливостей є SQL-ін'єкція (SQL Injection). Вона виникає у випадках, коли користувацькі дані без належної перевірки передаються до SQL-запитів. У результаті зловмисник може виконувати довільні SQL-команди, отримувати несанкціонований доступ до бази даних, змінювати або видаляти інформацію [44–46].

Іншим поширеним дефектом є міжсайтовий скриптинг (XSS, Cross-Site Scripting). Ця вразливість дозволяє впроваджувати шкідливий JavaScript-код у веб-сторінки, які переглядають інші користувачі. Наслідками XSS-атак можуть бути викрадення файлів cookie, захоплення користувацьких сесій або підміна вмісту сторінки [44–46].

До критичних дефектів також належать порушення механізмів автентифікації та авторизації. Використання слабких паролів, некоректне керування сесіями або відсутність багатофакторної автентифікації можуть призвести до несанкціонованого доступу до облікових записів користувачів.

Пов'язаною проблемою є порушення контролю доступу, коли користувач отримує можливість виконувати дії або переглядати ресурси, доступ до яких повинен бути обмеженим [44–46].

Поширеною вразливістю є також CSRF-атака (Cross-Site Request Forgery), під час якої зловмисник змушує користувача виконати небажані дії у веб-додатку від свого імені. Крім того, небезпеку становлять XXE-вразливості (XML External Entity), пов'язані з неправильною обробкою XML-даних, що може призводити до витоку конфіденційної інформації [44–46].

До дефектів веб-додатків також належать неправильна конфігурація безпеки, використання застарілих або вразливих сторонніх компонентів, небезпечна десеріалізація даних та недостатнє журналювання і моніторинг подій безпеки. Такі проблеми ускладнюють своєчасне виявлення атак і можуть значно підвищувати ризик компрометації системи [44–46].

Для зменшення кількості подібних вразливостей використовують перевірку введених даних, параметризовані SQL-запити, шифрування інформації, багатофакторну автентифікацію, регулярне оновлення компонентів та постійний моніторинг безпеки [44–46].

Таким чином, вразливості веб-додатків є одними з найнебезпечніших типів дефектів, оскільки вони безпосередньо впливають на конфіденційність, цілісність та доступність даних. Саме тому питання кібербезпеки повинні враховуватися на всіх етапах розробки та експлуатації веб-додатків [44–46].

Отже, своєчасне виявлення та усунення дефектів веб-додатків є необхідною умовою забезпечення їхньої надійності, безпеки, зручності використання та стабільної роботи на різних пристроях і в різних середовищах.

2.5 Інструменти та документація у процесі ручного тестування

Ефективність ручного тестування веб-додатків значною мірою залежить не лише від професійних навичок тестувальника, а й від використання відповідних інструментів та правильно організованої тестової документації. У процесі тестування важливо забезпечити систематичність перевірок, можливість відстеження дефектів, документування результатів тестування та координацію роботи між учасниками команди розробки. Саме тому Manual QA Engineers активно використовують спеціалізовані програмні засоби та стандартизовані типи документації.

Документація у процесі тестування виконує важливу роль у забезпеченні якості програмного продукту. Вона дозволяє структурувати процес перевірки функціональності, фіксувати очікувані результати, описувати знайдені дефекти та контролювати стан виконання тестування. До основних видів тестової документації належать тест-плани, тест-кейси, чек-листи та bug reports, які забезпечують послідовність і відтворюваність тестування.

Окрім документації, у ручному тестуванні широко застосовуються спеціалізовані інструменти для управління завданнями, тестовими сценаріями, API-запитами та аналізу роботи веб-додатків. Серед найбільш поширених засобів можна виділити Jira, TestRail, Postman та браузерні DevTools, які значно спрощують процес тестування, підвищують ефективність роботи тестувальників та забезпечують зручну взаємодію між членами команди розробки.

У процесі ручного тестування веб-додатків використовується широкий спектр програмних інструментів, які допомагають організувати тестову діяльність, перевіряти функціональність системи, аналізувати дефекти та контролювати якість програмного продукту. Сучасні QA-команди дедалі частіше застосовують хмарні веборієнтовані рішення, що дозволяють працювати з тестами без встановлення додаткового програмного забезпечення та забезпечують зручну взаємодію між учасниками проєкту [47].

Одним із найбільш поширених інструментів для організації тестового процесу є TestRail. Даний сервіс використовується для централізованого керування тест-кейсами, тест-планами та результатами тестування. Він дозволяє

структурувати сценарії перевірок, відстежувати виконання тестів, формувати звіти про стан якості продукту та інтегруватися з іншими системами, зокрема Jira, GitHub або Postman. Завдяки цьому TestRail часто виступає як основний центр керування тестуванням у QA-команді [47].

Для управління дефектами та організації роботи команди широко застосовується Jira. Цей інструмент дає змогу створювати задачі, фіксувати помилки, контролювати їхній статус та координувати взаємодію між тестувальниками, розробниками й менеджерами проєкту. Jira підтримує різні методології розробки, зокрема Agile та Kanban, а також забезпечує прозоре відстеження життєвого циклу дефекту – від моменту виявлення до виправлення.

Під час тестування серверної частини веб-додатків важливу роль відіграє Postman – інструмент для роботи з API. За його допомогою тестувальники можуть надсилати HTTP-запити, перевіряти відповіді сервера, аналізувати коректність передачі даних та автоматизувати перевірку бекенд-логіки. Крім того, Postman підтримує створення колекцій запитів, автоматизованих моніторингів та генерацію документації API, що значно спрощує процес тестування вебсервісів [47].

Для перевірки роботи веб-додатків на різних пристроях і в різних браузерах використовуються спеціалізовані платформи, зокрема BrowserStack. Даний сервіс надає доступ до великої кількості реальних мобільних і настільних пристроїв через браузер, що дозволяє тестувати адаптивність інтерфейсу, сумісність із різними операційними системами та стабільність роботи вебдодатка без необхідності утримання власної лабораторії пристроїв [47].

Серед інструментів для автоматизації окремих перевірок варто виділити BugBug – браузерний сервіс, який дозволяє записувати та запускати тести без необхідності програмування. Такий підхід особливо корисний для Agile-команд і стартапів, які прагнуть швидко автоматизувати повторювані перевірки та регресійне тестування. Також для перевірки поведінки продукту в реальних умовах застосовується платформа Global App Testing, що забезпечує краудтестинг

із залученням великої кількості тестувальників з різних країн і реальних пристроїв [47].

Окреме місце у процесі ручного тестування займають браузерні DevTools – вбудовані інструменти розробника в сучасних браузерах. Вони дозволяють аналізувати HTML-структуру сторінки, CSS-стилі, JavaScript-код, мережеві запити, продуктивність вебдодатка та помилки консолі. Використання DevTools значно спрощує пошук дефектів інтерфейсу, проблем адаптивності, некоректного завантаження ресурсів або помилок взаємодії клієнтської та серверної частин системи.

Таким чином, сучасні інструменти ручного тестування забезпечують ефективну організацію QA-процесу, автоматизацію рутинних операцій, контроль дефектів і всебічний аналіз роботи веб-додатків, що сприяє підвищенню якості та надійності програмного забезпечення.

Важливою складовою процесу ручного тестування є ведення тестової документації, яка забезпечує систематизацію перевірок, фіксацію результатів тестування та контроль відповідності веб-додатка визначеним вимогам. Основним документом у цьому процесі є тест-план – структурований опис перевірок, що визначає сценарії тестування, вхідні дані, очікувані результати та критерії успішного проходження тестів. Тест-плани створюються на основі тест-вимог і можуть охоплювати як окремі функції системи, так і комплексні сценарії взаємодії користувача з веб-додатком [6].

Кожен тестовий приклад у межах тест-плану містить ідентифікатор, опис мети перевірки, посилання на вимоги, перелік дій тестувальника та очікувану реакцію системи. Для ручного тестування такі сценарії зазвичай оформлюються у вигляді текстових документів або чек-листів, що дозволяє тестувальникам послідовно виконувати перевірки та фіксувати результати. У випадках, коли вхідні та вихідні дані мають чітко формалізований характер, тестові приклади можуть подаватися у вигляді таблиць, що спрощує аналіз покриття тестами та систематизацію сценаріїв [6].

Окрему роль у процесі тестування відіграють звіти про проходження тестів і звіти про дефекти. Після виконання тестів формується звіт, у якому фіксується статус кожного тестового прикладу – успішне або неуспішне проходження. У разі виявлення помилок створюються defect reports (bug reports), що містять опис проблеми, умови відтворення, очікуваний і фактичний результат, а також пріоритет дефекту. Такі звіти є основою для подальшого виправлення помилок та проведення регресійного тестування [6].

Для оцінювання повноти тестування також використовуються звіти про покриття програмного коду, які демонструють, які частини функціональності були перевірені тестами, а які залишилися непокритими. Аналіз покриття дозволяє виявити недостатньо протестовані ділянки системи та вдосконалити набір тестових сценаріїв. Таким чином, тестова документація є невід’ємним елементом процесу ручного тестування, оскільки забезпечує прозорість, повторюваність та контроль якості веб-додатків [6].

Таким чином, ефективність ручного тестування веб-додатків значною мірою залежить не лише від професійних навичок тестувальника, а й від правильного використання інструментів та якісного ведення тестової документації. Використання систем керування тестуванням, платформ для відстеження дефектів та засобів аналізу роботи веб-додатків дозволяє структурувати процес тестування, підвищити швидкість виявлення помилок і забезпечити прозорість взаємодії між учасниками команди розробки. Інструменти на кшталт Jira, TestRail, Postman і браузерних DevTools стали невід’ємною частиною сучасного процесу забезпечення якості вебзастосунків.

Отже, поєднання сучасних інструментів ручного тестування з належно організованою тестовою документацією забезпечує більш системний і ефективний підхід до контролю якості веб-додатків, сприяє своєчасному виявленню дефектів та підвищує надійність програмного продукту в умовах сучасної веброзробки.

3 РОЗРОБКА РЕКОМЕНДАЦІЙ ЩОДО РУЧНОГО ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

3.1 Формування алгоритму ручного тестування веб-додатків

Ручне тестування веб-додатків є багатостадійним процесом, ефективність якого значною мірою залежить від послідовності виконання перевірок, організації тестового процесу та правильності підготовки тестової документації. У сучасних умовах розробки програмного забезпечення забезпечення якості веб-додатків потребує не лише виявлення окремих дефектів, а й застосування системного підходу до планування, виконання та аналізу результатів тестування. Саме тому важливого значення набуває формування узагальненого алгоритму ручного тестування, який дозволяє структурувати дії тестувальника та забезпечити повноту перевірки функціональних і нефункціональних характеристик веб-додатка.

Процес ручного тестування охоплює декілька взаємопов'язаних етапів – від аналізу вимог і підготовки тестових сценаріїв до виконання перевірок, документування результатів та повторного тестування після виправлення дефектів. Кожен із цих етапів виконує окрему функцію в забезпеченні якості програмного продукту та впливає на ефективність виявлення помилок. Водночас систематизація процесу тестування дозволяє зменшити ризик пропуску критичних дефектів, підвищити повторюваність тестових перевірок і покращити контроль за станом якості веб-додатка на різних етапах його розробки.

Ефективність ручного тестування веб-додатків значною мірою залежить від наявності чітко визначеного алгоритму виконання перевірок, який забезпечує послідовність дій тестувальника та дозволяє систематизувати процес контролю якості. В умовах сучасної веб-розробки, де програмні системи постійно змінюються, а функціональність оновлюється короткими ітераціями, хаотичне виконання перевірок суттєво підвищує ризик пропуску критичних дефектів. Саме тому процес ручного тестування повинен базуватися на структурованому підході,

що охоплює етапи підготовки, виконання перевірок, аналізу результатів та повторного тестування після внесення змін до програмного забезпечення.

Формування алгоритму ручного тестування доцільно розпочинати з аналізу вимог до веб-додатка. На цьому етапі тестувальник вивчає функціональні та нефункціональні вимоги, специфікації, макети інтерфейсу, опис бізнес-процесів та іншу проєктну документацію. Основною метою є визначення очікуваної поведінки системи, виявлення критичних функціональних компонентів та встановлення потенційних ризиків. Якість цього етапу безпосередньо впливає на повноту подальших перевірок, оскільки помилки або неоднозначності у вимогах можуть призвести до неправильного формування тестових сценаріїв.

Особливу увагу під час аналізу вимог необхідно приділяти:

- критичним бізнес-функціям веб-додатка;
- сценаріям взаємодії користувача із системою;
- обмеженням і правилам валідації даних;
- інтеграції із зовнішніми сервісами;
- вимогам до безпеки та захисту даних;
- особливостям роботи інтерфейсу на різних пристроях і браузерах.

На основі аналізу вимог формується стратегія ручного тестування, яка визначає підходи до перевірки різних компонентів системи. У сучасних умовах доцільним є використання адаптивного підходу до тестування, за якого методи перевірки обираються залежно від рівня ризику функціональності, етапу розробки та типу тестових сценаріїв. Такий підхід дозволяє поєднувати формалізовані методи перевірки з дослідницьким тестуванням, що особливо важливо для веб-додатків із високою динамікою змін.

Для компонентів із високим рівнем ризику рекомендується використовувати детально формалізовані тест-кейси у поєднанні з *exploratory testing*. Це дозволяє не лише перевірити відповідність функціональності вимогам, а й виявити дефекти, пов'язані з нестандартними сценаріями використання. Для менш критичних компонентів може бути достатнім застосування чек-листів або скорочених

сценаріїв перевірки. Таким чином забезпечується більш раціональне використання часу та ресурсів тестування.

Після визначення підходів до тестування виконується підготовка тестових сценаріїв і тестових даних. Тестові сценарії описують послідовність дій користувача та очікувану реакцію системи у відповідь на виконані дії. Вони повинні охоплювати як позитивні, так і негативні варіанти використання функціональності. Позитивні сценарії перевіряють коректну роботу системи за умови введення правильних даних, тоді як негативні спрямовані на перевірку обробки помилок, некоректних значень та нестандартних ситуацій.

При формуванні тестових сценаріїв доцільно враховувати:

- типові дії користувачів;
- граничні значення вхідних даних;
- некоректні або неповні дані;
- можливі помилки користувача;
- сценарії втрати з'єднання або нестабільної роботи мережі;
- взаємодію між окремими модулями веб-додатка.

Важливим етапом є підготовка тестових даних. Недостатньо лише визначити послідовність дій – необхідно також підготувати набір даних, який дозволить максимально повно перевірити функціональність системи. Тестові дані повинні охоплювати різні типи значень: коректні, некоректні, граничні та нестандартні. Наприклад, при перевірці форми реєстрації доцільно використовувати як валідні адреси електронної пошти, так і помилкові формати, надто довгі значення або порожні поля.

Після підготовки сценаріїв формується набір тест-кейсів. Тест-кейс є формалізованим описом окремої перевірки та, як правило, містить:

- ідентифікатор тесту;
- назву та мету перевірки;
- попередні умови виконання;
- тестові дані;
- послідовність дій;

- очікуваний результат;
- фактичний результат після виконання тесту.

Наявність структурованих тест-кейсів забезпечує повторюваність перевірок та спрощує процес регресійного тестування. Крім того, це дозволяє різним тестувальникам виконувати однакові перевірки з мінімальними відмінностями в результатах. Водночас надмірна формалізація може зменшувати гнучкість тестування, тому в практиці ручного QA часто застосовується комбінування тест-кейсів із чек-листами та дослідницьким тестуванням.

Окрему роль у процесі ручного тестування відіграє пріоритизація перевірок. У реальних проєктах тестувальник часто обмежений у часі, тому перевірка всієї функціональності в повному обсязі може бути неможливою. У такому випадку першочергово тестуються найбільш критичні компоненти системи, помилки в яких можуть призвести до серйозних наслідків для користувачів або бізнес-процесів. До таких компонентів зазвичай належать:

- авторизація та автентифікація;
- операції з персональними даними;
- платіжні модулі;
- механізми обробки замовлень;
- функції взаємодії з базою даних;
- елементи безпеки веб-додатка.

Risk-based testing дозволяє оптимізувати процес перевірки та зосередити ресурси тестування на найбільш вразливих або критичних ділянках системи. Такий підхід є особливо актуальним для великих веб-додатків із широкою функціональністю та частими оновленнями.

Після завершення етапу підготовки тестової документації та формування набору тестових сценаріїв здійснюється безпосереднє виконання ручного тестування веб-додатка. На цьому етапі тестувальник послідовно перевіряє функціональність системи відповідно до визначених сценаріїв, аналізує поведінку інтерфейсу, реакцію системи на введені дані та загальну стабільність роботи веб-додатка. Основною метою виконання тестування є виявлення дефектів, які

можуть впливати на коректність функціонування системи, зручність використання або безпеку обробки даних.

Під час виконання тестування важливу роль відіграє не лише перевірка очікуваних сценаріїв використання, а й аналіз нетипових ситуацій, які можуть виникати в реальних умовах експлуатації. Саме тому поряд із формалізованими тест-кейсами доцільно використовувати елементи *exploratory testing*, що дозволяє тестувальнику гнучко змінювати напрям перевірки залежно від поведінки системи. Такий підхід особливо ефективний для виявлення дефектів інтерфейсу, проблем навігації, некоректної обробки нестандартних дій користувача та помилок взаємодії між окремими компонентами веб-додатка.

У процесі виконання перевірок тестувальник повинен контролювати:

- коректність відображення елементів інтерфейсу;
- правильність роботи функціональних компонентів;
- обробку валідних і невалідних даних;
- стабільність роботи веб-додатка;
- швидкість реакції системи;
- коректність повідомлень про помилки;
- збереження та передачу даних між модулями;
- адаптивність інтерфейсу для різних пристроїв і браузерів.

Для аналізу поведінки веб-додатка в процесі ручного тестування широко використовуються спеціалізовані інструменти. Зокрема, браузерні DevTools дозволяють перевіряти структуру HTML-документа, аналізувати CSS-стили, відстежувати мережеві запити, досліджувати повідомлення консолі та виявляти помилки JavaScript. При тестуванні серверної частини та API доцільно використовувати Postman, який забезпечує можливість надсилання HTTP-запитів, перевірки відповідей сервера та аналізу структури переданих даних. Використання таких інструментів суттєво підвищує ефективність ручного тестування та дозволяє виявляти дефекти, які неможливо встановити лише шляхом візуальної перевірки інтерфейсу.

У разі виявлення дефекту тестувальник формує bug report – структурований опис проблеми, необхідний для її подальшого виправлення. Якість оформлення bug report безпосередньо впливає на швидкість локалізації та усунення дефекту розробниками. Недостатньо деталізований опис помилки може ускладнити її відтворення та збільшити час виправлення.

Типовий bug report містить:

- ідентифікатор дефекту;
- короткий опис проблеми;
- кроки для відтворення помилки;
- очікуваний результат;
- фактичний результат;
- середовище тестування;
- рівень пріоритету та критичності;
- додаткові матеріали (скріншоти, відео, логи).

Після виправлення дефектів виконується повторне тестування (retesting), метою якого є перевірка коректності внесених змін. На цьому етапі тестувальник повторно виконує сценарії, пов'язані з раніше виявленими помилками, та аналізує, чи була проблема повністю усунута. Водночас внесення змін до програмного коду може спричинити появу нових дефектів у вже перевірених компонентах системи, тому після retesting додатково проводиться регресійне тестування.

Регресійне тестування спрямоване на перевірку того, що внесені зміни не порушили роботу існуючої функціональності веб-додатка. У сучасних проєктах, де оновлення виконуються регулярно, regression testing є одним із ключових елементів процесу забезпечення якості. Обсяг регресійних перевірок залежить від масштабу змін, рівня ризику та критичності функціональних компонентів.

Під час регресійного тестування доцільно першочергово перевіряти:

- основні бізнес-процеси;
- механізми авторизації та автентифікації;
- роботу бази даних;
- інтеграцію із зовнішніми сервісами;

- функції обробки та збереження даних;
- критичні елементи інтерфейсу користувача.

Важливою складовою алгоритму ручного тестування є документування результатів перевірки. У процесі тестування формується набір звітів, які дозволяють оцінити поточний стан якості веб-додатка, рівень покриття тестами та кількість виявлених дефектів. Документування результатів забезпечує прозорість процесу тестування та створює основу для подальшого аналізу ефективності перевірок.

До основних видів тестової документації, що формуються на завершальних етапах тестування, належать:

- звіти про проходження тестів;
- звіти про дефекти;
- звіти про покриття тестами;
- підсумкова тестова звітність;
- чек-листи виконаних перевірок.

Аналіз результатів тестування дозволяє оцінити рівень стабільності веб-додатка та визначити проблемні компоненти системи. Якщо в процесі тестування виявляється велика кількість дефектів у певному модулі, це може свідчити про недоліки архітектури, неякісно сформовані вимоги або недостатній рівень покриття тестами. У такому випадку результати тестування стають підставою для перегляду підходів до розробки, уточнення вимог або розширення тестових сценаріїв.

Таким чином, узагальнений алгоритм ручного тестування веб-додатків повинен охоплювати всі етапи процесу забезпечення якості – від аналізу вимог і підготовки тестових сценаріїв до виконання перевірок, повторного тестування та формування звітності. Використання адаптивного підходу до організації тестування, що поєднує формалізовані та дослідницькі методи перевірки, дозволяє більш ефективно використовувати ресурси тестування та підвищує ймовірність своєчасного виявлення дефектів. Систематизація процесу ручного

тестування забезпечує більш повний контроль якості веб-додатків і сприяє підвищенню надійності програмного забезпечення в умовах сучасної веброзробки.

Отже, узагальнений алгоритм ручного тестування веб-додатків включає послідовне виконання таких етапів: аналіз вимог і визначення об'єктів тестування; оцінювання ризиків функціональних компонентів; формування тестових сценаріїв, тест-кейсів і тестових даних; підготовку тестового середовища; виконання функціонального, інтерфейсного та нефункціонального тестування; фіксацію виявлених дефектів і результатів перевірки; проведення повторного тестування після виправлення помилок; регресійну перевірку змінених компонентів; формування підсумкової звітності щодо результатів тестування. Така послідовність дій дозволяє забезпечити системність процесу контролю якості та підвищити повноту тестового покриття веб-додатка.

Важливою особливістю сформованого алгоритму є застосування адаптивного підходу до ручного тестування, який передбачає комбінування різних методів перевірки залежно від рівня ризику функціональності, типу тестових сценаріїв та етапу життєвого циклу програмного забезпечення. Для критично важливих компонентів доцільним є поєднання формалізованих тест-кейсів із *exploratory testing*, що дозволяє не лише перевіряти відповідність функціональним вимогам, а й виявляти нетипові дефекти та помилки взаємодії компонентів системи. Для менш ризикових елементів можуть використовуватися спрощені сценарії перевірки та чек-листи, що забезпечує більш раціональне використання ресурсів тестування.

Таким чином, запропонований алгоритм ручного тестування забезпечує комплексний підхід до перевірки веб-додатків і створює основу для ефективного виявлення дефектів на різних етапах розробки програмного забезпечення. Поєднання структурованої тестової документації, системного виконання перевірок та адаптивного використання методів тестування сприяє підвищенню якості веб-додатків, мінімізації ризику пропуску критичних помилок і покращенню ефективності процесу забезпечення якості програмного забезпечення.

3.2 Практична реалізація тестування демонстраційного веб-додатка

Практична реалізація ручного тестування веб-додатків потребує наявності програмної системи, яка містить типові компоненти сучасних вебзастосунків та дозволяє відтворити реальні сценарії перевірки функціональності, безпеки й стабільності роботи програмного забезпечення. З цією метою у межах роботи було розроблено демонстраційний веб-додаток «QA TaskFlow», призначений для моделювання процесу ручного тестування та демонстрації основних етапів забезпечення якості веб-програм.

Розроблений веб-додаток являє собою спрощену систему керування завданнями, реалізовану з використанням технологій PHP, MySQL, HTML, CSS, JavaScript та Bootstrap. Архітектура системи включає клієнтську та серверну частини, що дозволяє виконувати перевірку як інтерфейсної логіки, так і механізмів обробки даних на стороні сервера. Веб-додаток підтримує базовий набір функціональних можливостей, характерних для більшості сучасних інформаційних систем, зокрема:

- реєстрацію та авторизацію користувачів;
- роботу із сесіями;
- створення, редагування та видалення завдань;
- систему коментарів;
- взаємодію з базою даних;
- обробку форм введення даних;
- механізми валідації та контролю доступу.

Особливістю розробленої системи є наявність спеціального режиму `BUG_MODE`, який дозволяє навмисно активувати дефектні або вразливі ділянки програмного коду. Такий підхід дає змогу використовувати один і той самий веб-додаток як для демонстрації процесу виявлення дефектів, так і для подальшої перевірки коректності їх виправлення. У режимі `BUG_MODE` в системі навмисно

реалізовано типові проблеми веб-додатків, зокрема SQL-ін'єкції, Stored XSS, порушення контролю доступу (IDOR), відсутність належної серверної валідації та небезпечну обробку HTTP-запитів. Після вимкнення режиму система переходить до захищеного режиму роботи із застосуванням prepared statements, додаткової валідації та перевірок прав доступу.

Використання такого підходу дозволяє не лише моделювати процес ручного тестування, а й демонструвати повний цикл роботи Manual QA Engineer: від виявлення дефекту до проведення повторного тестування та регресійної перевірки після внесення змін до програмного коду. Крім того, це створює умови для оцінювання ефективності різних підходів до тестування та аналізу впливу вразливостей на роботу веб-додатка.

Під час тестування демонстраційного веб-додатка основна увага приділялася перевірці функціональних можливостей системи, правильності обробки користувацьких даних, стабільності роботи інтерфейсу та безпеки взаємодії між клієнтською і серверною частинами. Для цього були сформовані тестові сценарії, які охоплювали як позитивні, так і негативні перевірки. Позитивні сценарії використовувалися для підтвердження коректної роботи функціональності відповідно до вимог, тоді як негативні були спрямовані на виявлення помилок обробки некоректних або небезпечних даних.

До основних тестових сценаріїв, реалізованих у межах роботи, належать:

- перевірка реєстрації нового користувача;
- перевірка авторизації з коректними та некоректними обліковими даними;
- тестування завершення сесії користувача;
- створення нового завдання;
- редагування та видалення завдань;
- перевірка роботи системи коментарів;
- перевірка серверної та клієнтської валідації форм;
- тестування контролю доступу до чужих даних;
- перевірка стійкості до SQL-ін'єкцій;

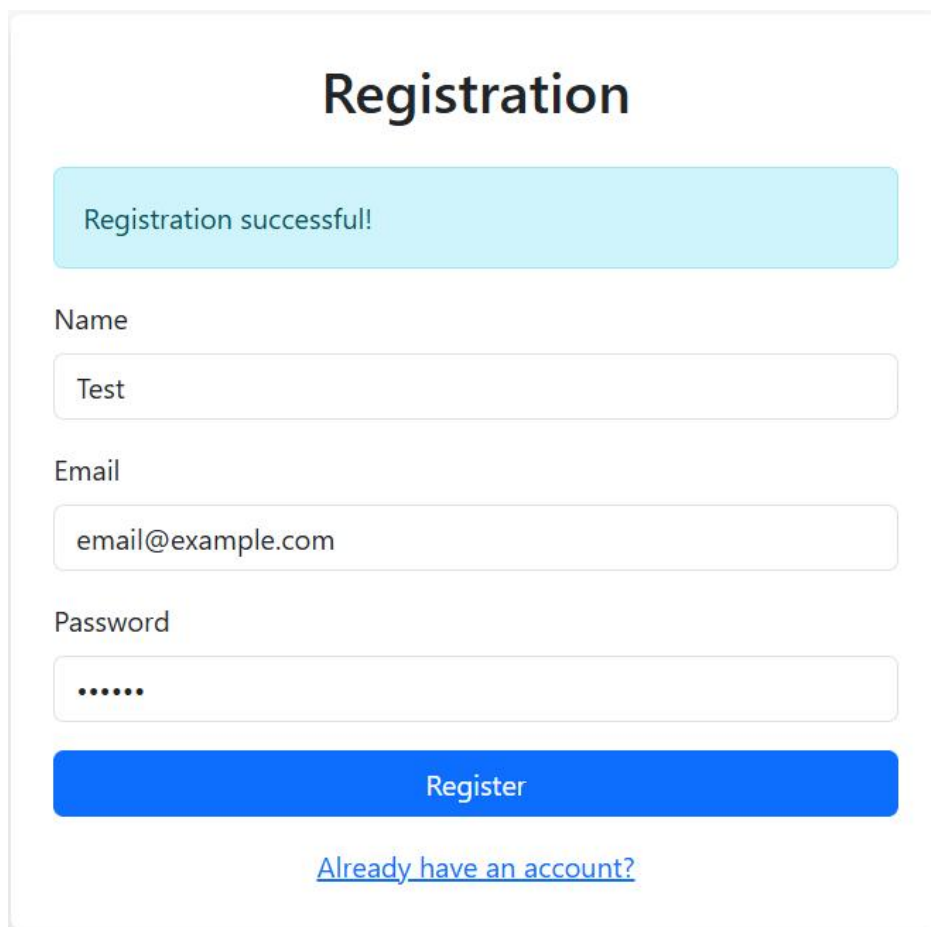
- тестування захисту від XSS-атак;
- перевірка коректності відображення даних у користувацькому інтерфейсі;
- тестування поведінки системи при введенні граничних та неочікуваних значень.

Формування тестових сценаріїв здійснювалося з урахуванням ризик-орієнтованого та адаптивного підходів до ручного тестування. Для функціональності, яка має підвищений рівень ризику, зокрема авторизації та обробки користувацького введення, використовувалося поєднання формалізованих тест-кейсів із елементами дослідницького тестування. Це дозволило не лише перевірити відповідність функціональності заданим вимогам, а й виявити нетипові дефекти, які складно передбачити на етапі формалізації тестової документації.

Таким чином, розроблений демонстраційний веб-додаток створює повноцінне середовище для практичної реалізації ручного тестування веб-систем та дозволяє продемонструвати основні процеси забезпечення якості програмного забезпечення. Наявність навмисно реалізованих дефектів і можливості їх подальшого виправлення забезпечує наочність процесу тестування та дозволяє комплексно оцінити ефективність застосованих методів ручної перевірки веб-додатків.

Після формування тестових сценаріїв було проведено ручне тестування демонстраційного веб-додатка «QA TaskFlow» відповідно до запропонованого алгоритму тестування. Основна увага приділялася перевірці функціональних можливостей системи, коректності обробки користувацьких даних, стабільності роботи інтерфейсу та виявленню типових дефектів і вразливостей веб-додатків. Тестування виконувалося у середовищі локального веб-сервера OpenServer із використанням браузера Google Chrome та інструментів DevTools. Перевірка проводилася спочатку в режимі `BUG_MODE=true`, у якому система містила навмисно залишені дефекти, а після їх усунення – повторно в режимі `BUG_MODE=false`.

Першим етапом було протестовано модуль реєстрації користувачів. Для цього виконувалася перевірка створення нового облікового запису із введенням коректних даних користувача. Очікуваним результатом була успішна реєстрація користувача та створення нового запису в базі даних. У процесі перевірки було встановлено, що реєстрація працює коректно, а користувач успішно додається до таблиці users. Результат роботи форми реєстрації наведено на рис. 3.1.

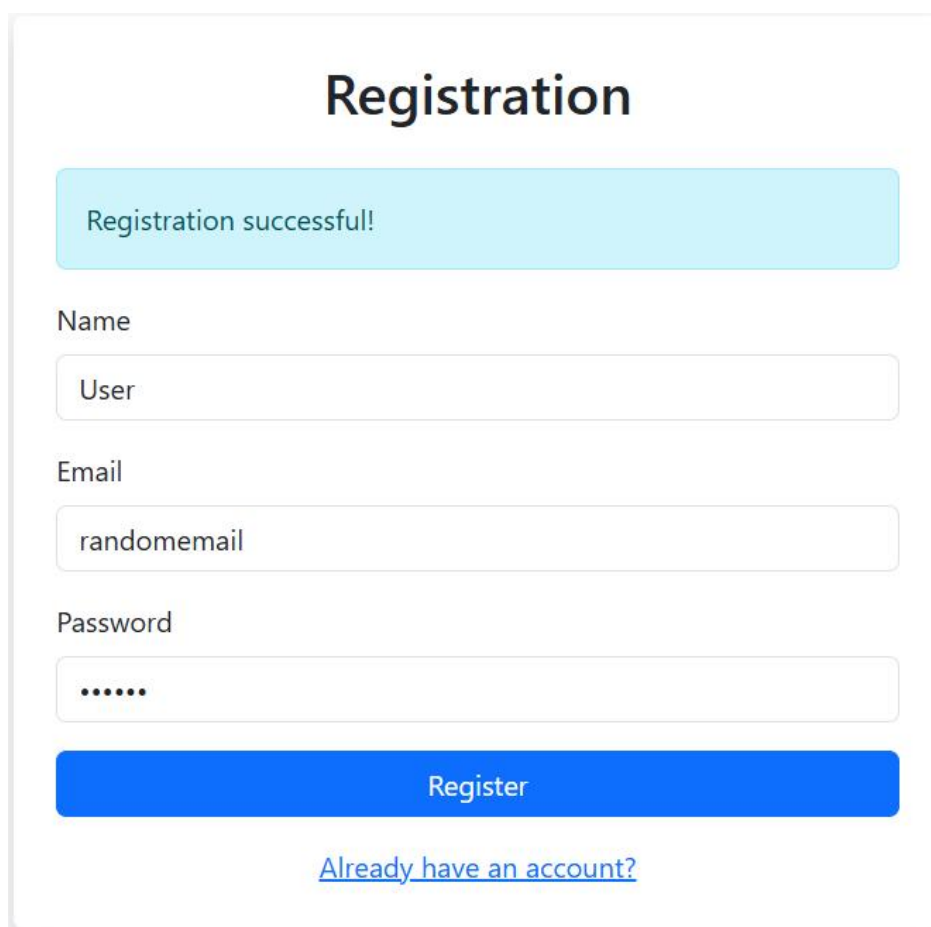


The image shows a web form titled "Registration". At the top, there is a light blue success message box that says "Registration successful!". Below this, there are three input fields: "Name" with the value "Test", "Email" with the value "email@example.com", and "Password" with masked characters ".....". At the bottom of the form is a blue button labeled "Register". Below the button is a link that says "Already have an account?".

Рис. 3.1 – Результат роботи форми реєстрації

Після цього було проведено негативне тестування форми реєстрації. У поле електронної адреси вводилися некоректні значення без символу «@» та доменної частини. Очікувалося, що система відхилить такі дані та відобразить повідомлення про помилку валідації. Фактичний результат показав, що в режимі BUG_MODE серверна перевірка email була відсутня, унаслідок чого користувач створювався навіть із некоректною адресою електронної пошти. Це свідчило про

дефект механізму валідації даних. Приклад некоректної обробки email-адреси наведено на рис. 3.2.



The image shows a web registration form titled "Registration". At the top, a light blue message box says "Registration successful!". Below this are three input fields: "Name" with the value "User", "Email" with the value "randomemail", and "Password" with masked characters ".....". A blue "Register" button is at the bottom, followed by a blue link that says "Already have an account?".

Рис. 3.2 – Некоректна обробка email-адреси

За результатами тестування було сформовано bug report, у якому зазначалося, що система не виконує серверну перевірку формату електронної адреси. Дефект було класифіковано як Major, оскільки він не призводив до порушення працездатності системи, проте дозволяв створювати некоректні облікові записи та негативно впливав на цілісність даних.

Наступним етапом стало тестування модуля авторизації користувачів. Спочатку перевірявся стандартний сценарій входу до системи з використанням коректних облікових даних. Очікуваним результатом була успішна авторизація користувача та створення активної сесії. Фактичний результат відповідав очікуваному: після введення валідних даних система виконувала перенаправлення

на головну сторінку додатка, що підтверджувало коректну роботу механізму авторизації. Результат тестування наведено на рис. 3.3.

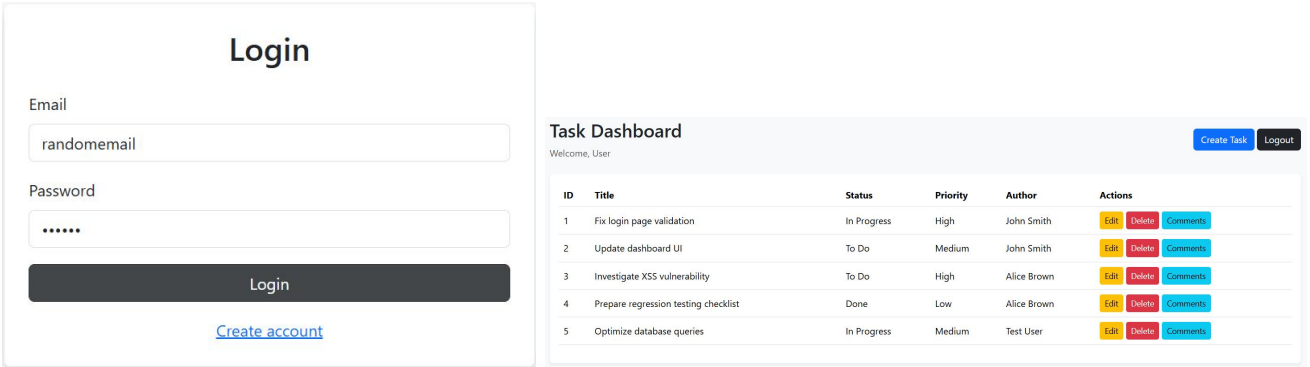


Рис. 3.3 – Коректна робота авторизації

Окрему увагу було приділено перевірці захищеності форми входу від SQL-ін'єкцій. Для цього у поле email вводився спеціально сформований SQL-вираз:

```
admin@taskflow.local' OR TRUE #
```

При цьому у поле пароля вводилося довільне значення. Очікуваним результатом було відхилення некоректного запиту та заборона доступу до системи. Однак фактичний результат показав, що система успішно авторизувала користувача без перевірки правильності пароля. Причиною цього стало формування SQL-запиту шляхом прямої конкатенації користувацьких даних без використання параметризованих запитів. У результаті SQL-вираз «OR TRUE» робив умову авторизації істинною, а символ «#» перетворював залишок запиту на коментар. Демонстрацію успішної SQL-ін'єкції наведено на рис. 3.4.

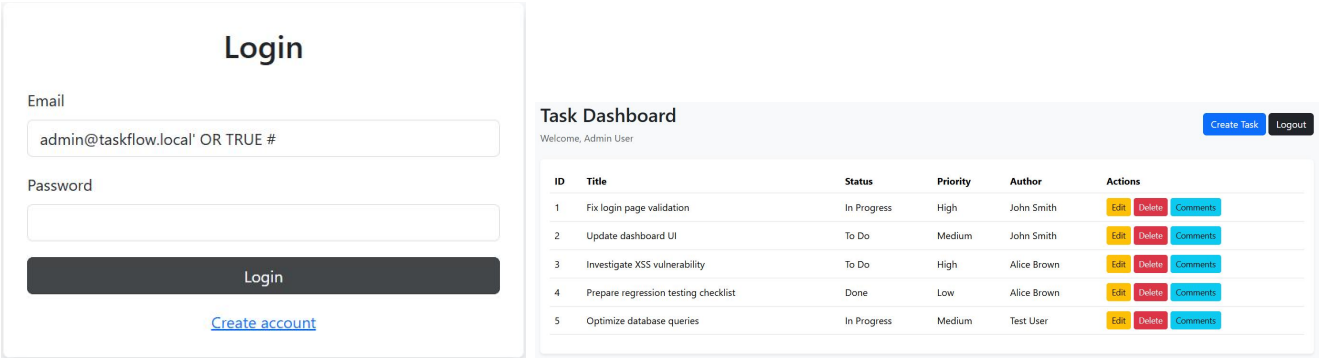


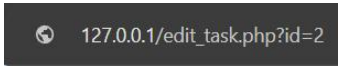
Рис. 3.4 – Успішна SQL-ін'єкція

Під час аналізу дефекту було встановлено, що вразливість виникла через використання небезпечної конструкції SQL-запиту:

```
$sql = "  
SELECT * FROM users  
WHERE email = '$email'  
AND password = '$password'  
";
```

У bug report було зазначено, що вразливість дозволяє отримати несанкціонований доступ до системи без знання облікових даних користувача. Дефект було класифіковано як Critical через високий рівень ризику для безпеки веб-додатка.

Після цього було протестовано модуль роботи із завданнями та каталогом даних. Під час перевірки здійснювалося створення нових записів, редагування інформації та видалення елементів. Очікувалося, що система коректно оброблятиме введені дані та відображатиме актуальний стан інформації в базі даних. У більшості випадків функціональність працювала стабільно, однак у режимі BUG_MODE було виявлено проблему відсутності належної перевірки прав доступу до окремих записів. Зокрема, користувач міг отримати доступ до редагування чужих записів шляхом прямої зміни параметра id у URL-адресі. Приклад такої поведінки системи наведено на рис. 3.5.



127.0.0.1/edit_task.php?id=2

Edit Task

Title
Update dashboard UI

Description
Improve responsive layout for mobile devices.

Status
Open

Priority
Low

Save Changes Back

Рис. 3.5 – редагування чужого запису шляхом підміни id

Виявлений дефект відповідає вразливості типу IDOR (Insecure Direct Object Reference), яка виникає через відсутність перевірки приналежності об'єкта поточному користувачеві. У bug report зазначалося, що проблема може призвести до несанкціонованого перегляду або зміни чужих даних, а тому була віднесена до категорії High.

Наступним етапом було протестовано модуль коментарів та обробки користувацького введення. Для цього у форму коментаря вводився JavaScript-код:

```
<script>alert("XSS")</script>
```

Очікувалося, що система виконає екранування HTML-тегів та відобразить текст як звичайний рядок. Проте фактичний результат показав, що код виконувався безпосередньо у браузері користувача, внаслідок чого відображалося JavaScript-повідомлення. Це підтвердило наявність Stored XSS-вразливості. Приклад виконання шкідливого скрипта наведено на рис. 3.6.

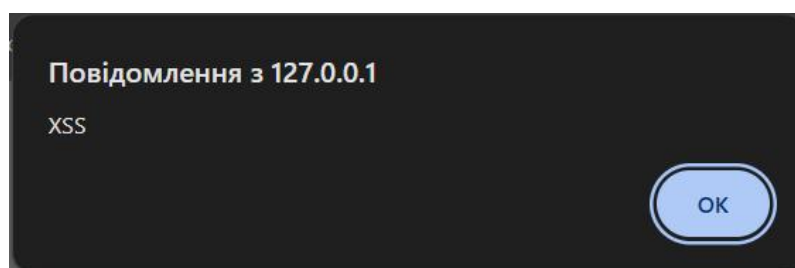


Рис. 3.6 – Виконання XSS

Причиною дефекту була відсутність екранування користувацького введення перед його виведенням у HTML-сторінці. У bug report зазначалося, що така вразливість може використовуватися для викрадення сесій користувачів, виконання довільного JavaScript-коду та реалізації атак на клієнтську частину веб-додатка. Дефект було класифіковано як Critical.

Крім функціонального та безпекового тестування, було виконано перевірку коректності роботи інтерфейсу користувача. Здійснювалася перевірка адаптивності сторінок при зміні роздільної здатності екрана, тестування навігаційних елементів, повідомлень про помилки та поведінки форм введення даних. У процесі тестування було встановлено, що окремі елементи інтерфейсу в режимі BUG_MODE некоректно відображалися на мобільних пристроях через відсутність адаптивних CSS-стилів. Результати перевірки адаптивності інтерфейсу наведено на рис. 3.7.

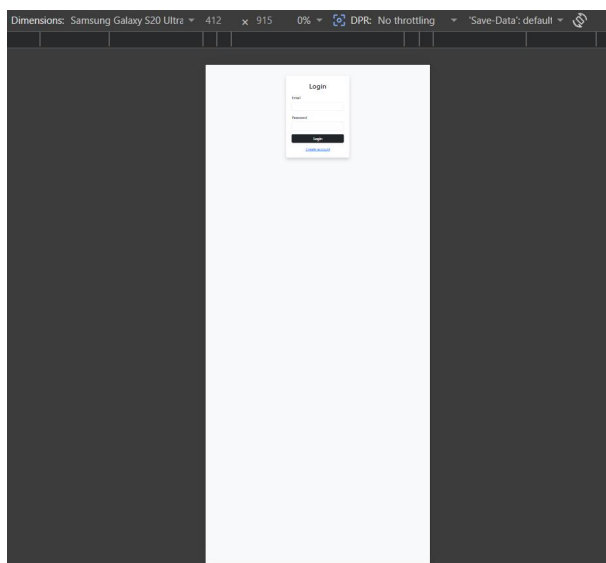


Рис. 3.7 – Результат перевірки адаптивності інтерфейсу

Після виявлення дефектів було виконано внесення змін до програмного коду веб-додатка. Для усунення SQL-ін'єкцій було використано prepared statements, для захисту від XSS – функцію htmlspecialchars(), а для усунення IDOR-вразливості – перевірку приналежності записів поточному користувачеві. Після цього було проведено повторне тестування та регресійна перевірка системи.

Результати повторного тестування показали, що після вимкнення режиму дефектів система коректно обробляє введені дані, блокує SQL-ін'єкції, виконує екранування HTML-коду та забороняє доступ до чужих записів. Повторна перевірка також підтвердила стабільність роботи основних функціональних модулів веб-додатка після внесення змін до програмного коду.

Таким чином, проведене ручне тестування дозволило виявити як функціональні дефекти, так і типові вразливості веб-додатків, пов'язані з некоректною обробкою користувацьких даних та відсутністю належних механізмів захисту. Поєднання формалізованих тест-кейсів із елементами дослідницького тестування дозволило ефективно виявити дефекти різного типу та підтвердило доцільність застосування адаптивного підходу до ручного тестування веб-додатків.

За результатами проведеного ручного тестування було встановлено, що навіть відносно простий веб-додаток може містити значну кількість дефектів і вразливостей, пов'язаних із некоректною обробкою користувацьких даних, відсутністю належної серверної валідації та недостатнім контролем доступу. Найбільш критичні проблеми були виявлені у механізмах авторизації та обробки введених користувачем даних. Зокрема, під час тестування було підтверджено наявність SQL-ін'єкції у формі входу до системи, Stored XSS-вразливості у модулі коментарів, а також IDOR-вразливості, пов'язаної з відсутністю перевірки прав доступу до окремих записів.

Проведене тестування також показало, що значна частина дефектів виникає через недостатню перевірку вхідних даних на стороні сервера. Відсутність валідації email-адрес, некоректна обробка помилок та можливість передачі довільних значень через URL-параметри створювали умови для порушення цілісності даних і несанкціонованого доступу до функціональності веб-додатка. Крім того, окремі проблеми були пов'язані з інтерфейсом користувача, зокрема адаптивністю сторінок та коректністю відображення елементів на різних типах пристроїв.

Практична реалізація тестування підтвердила доцільність поєднання формалізованих тест-кейсів із елементами дослідницького тестування. Саме exploratory testing дозволило виявити частину дефектів безпеки та нетипових сценаріїв взаємодії, які складно було б передбачити виключно за допомогою стандартних тестових сценаріїв. Водночас використання структурованих test cases забезпечило повторюваність перевірок та можливість проведення регресійного тестування після внесення змін до програмного коду.

Для систематизації результатів тестування було сформовано bug reports для найбільш критичних дефектів. Приклад bug report для SQL-ін'єкції наведено нижче.

Ідентифікатор дефекту: BR-01

Назва дефекту: SQL Injection у формі авторизації

Рівень критичності: Critical

Передумови: користувач перебуває на сторінці входу до системи

Кроки відтворення:

1. Відкрити сторінку авторизації.
2. У поле email ввести: admin@taskflow.local' OR TRUE #
3. У поле пароля ввести довільне значення.
4. Натиснути кнопку входу.

Очікуваний результат: система повинна відхилити некоректні облікові дані та заборонити авторизацію.

Фактичний результат: система успішно виконує авторизацію без перевірки правильності пароля.

Статус: виправлено після використання prepared statements.

Демонстрацію дефекту наведено на рис. 3.4, виправлення – на рис. 3.8.

Login

Invalid credentials.

Email

admin@taskflow.local' OR TRUE #

Password

Login

[Create account](#)

Рис. 3.8 – Виправлена SQL-ін'єкція

Другим критичним дефектом стала Stored XSS-вразливість у модулі коментарів.

Ідентифікатор дефекту: BR-02

Назва дефекту: Stored XSS у системі коментарів

Рівень критичності: Critical

Передумови: користувач авторизований у системі

Кроки відтворення:

1. Відкрити сторінку коментарів.
2. У поле введення коментаря вставити: `<script>alert("XSS")</script>`
3. Надіслати коментар.

Очікуваний результат: HTML- та JavaScript-код повинні відображатися як текст без виконання.

Фактичний результат: браузер виконує JavaScript-код, унаслідок чого відображається системне повідомлення.

Статус: виправлено шляхом використання `htmlspecialchars()`.

Приклад виконання XSS-коду наведено на рис. 3.6, виправлення – на рис. 3.9.

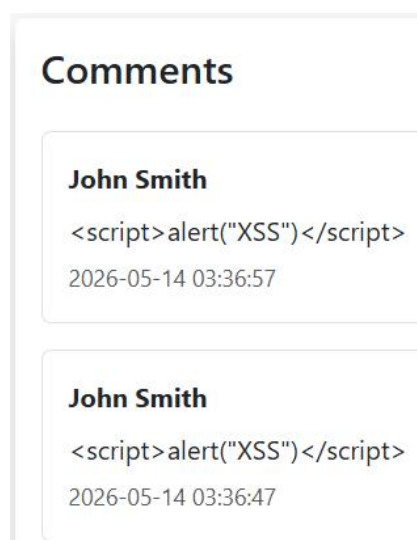


Рис. 3.9 – Виправлений XSS

Також у процесі тестування було виявлено дефект серверної валідації email-адрес.

Ідентифікатор дефекту: BR-03

Назва дефекту: Відсутність серверної перевірки email

Рівень критичності: Major

Передумови: відкрито сторінку реєстрації користувача

Кроки відтворення:

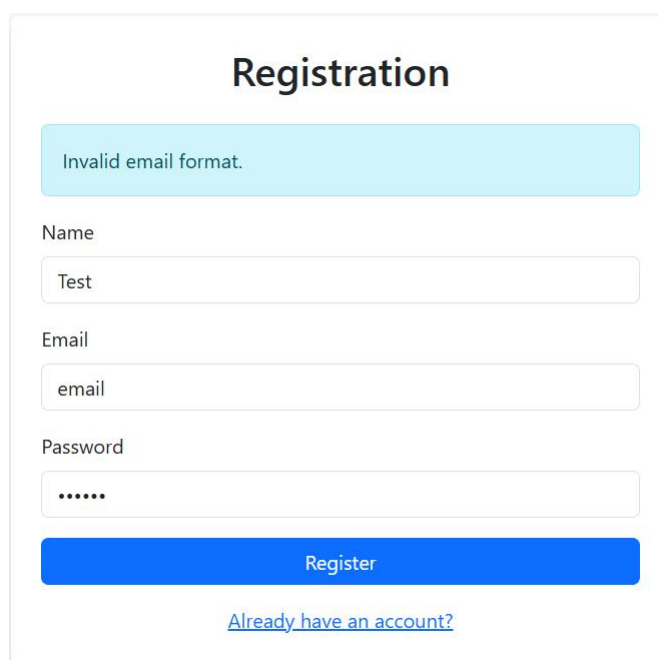
1. Увести довільне ім'я користувача.
2. У поле email ввести некоректне значення без символу @.
3. Завершити реєстрацію.

Очікуваний результат: система повинна відобразити повідомлення про помилку валідації.

Фактичний результат: користувач успішно створюється в системі з некоректною email-адресою.

Статус: виправлено після додавання серверної перевірки формату email.

Результат тестування наведено на рис. 3.10.



The image shows a web registration form titled "Registration". At the top, there is a light blue error message box that says "Invalid email format.". Below this, there are three input fields: "Name" with the value "Test", "Email" with the value "email", and "Password" with masked characters "*****". A blue "Register" button is positioned below the password field. At the bottom of the form, there is a blue link that says "Already have an account?".

Рис. 3.10 – Додана серверна перевірка email

Після усунення виявлених дефектів було проведено повторне тестування та регресійна перевірка основних функціональних модулів веб-додатка. Повторна перевірка підтвердила відсутність SQL-ін'єкцій, XSS-вразливостей і проблем із контролем доступу. Крім того, система почала коректно виконувати серверну валідацію користувацьких даних та обробляти помилки введення.

Таким чином, практична реалізація ручного тестування продемонструвала ефективність системного підходу до перевірки веб-додатків і підтвердила важливість поєднання функціонального, безпекового та дослідницького тестування у процесі забезпечення якості програмного забезпечення.

3.3 Формування практичних рекомендацій щодо підвищення ефективності ручного тестування

Ефективність ручного тестування веб-додатків значною мірою залежить не лише від технічної підготовки тестувальників, але й від організації самого процесу тестування. Зі збільшенням складності сучасних вебзастосунків зростає

кількість можливих сценаріїв взаємодії користувача із системою, що ускладнює забезпечення повного тестового покриття та підвищує ризик пропуску критичних дефектів. У таких умовах особливого значення набуває формування практичних підходів, які дозволяють підвищити результативність ручного тестування без суттєвого збільшення витрат часу та ресурсів.

Одним із ключових напрямів удосконалення процесу ручного тестування є оптимізація роботи з тестовою документацією, пріоритизація тестових сценаріїв і раціональне поєднання різних підходів до перевірки програмного забезпечення. Практика показує, що використання лише формалізованих тест-кейсів не завжди дозволяє виявити нетипові дефекти, тоді як виключно дослідницьке тестування ускладнює повторюваність перевірок і контроль тестового покриття. Тому підвищення ефективності тестування потребує комбінування структурованих та адаптивних підходів залежно від особливостей функціональності веб-додатка.

Важливу роль також відіграє якість взаємодії між учасниками процесу розробки програмного забезпечення. Своєчасна комунікація між тестувальниками, розробниками та аналітиками сприяє швидшому усуненню дефектів, зменшує кількість неоднозначностей у вимогах та дозволяє оперативно реагувати на зміни функціональності системи. У результаті правильно організований процес ручного тестування забезпечує більш стабільну роботу веб-додатків, підвищує якість програмного продукту та знижує ризик виникнення критичних помилок у процесі експлуатації.

Підвищення ефективності ручного тестування веб-додатків потребує комплексного підходу, який охоплює організацію процесу тестування, формування тестової документації, вибір методів перевірки та взаємодію між учасниками команди розробки. На практиці значна частина проблем у процесі тестування пов'язана не лише зі складністю програмного забезпечення, а й із нераціональним використанням ресурсів тестування, дублюванням перевірок, недостатнім покриттям критичних сценаріїв та відсутністю єдиного підходу до організації роботи manual QA engineers. У зв'язку з цим доцільним є формування

набору практичних рекомендацій, спрямованих на оптимізацію процесу ручного тестування та зменшення ризику пропуску дефектів.

Однією з основних рекомендацій є використання ризик-орієнтованого підходу до тестування (risk-based testing). Практика розробки веб-додатків показує, що різні компоненти системи мають різний рівень критичності для користувача та бізнес-процесів. Наприклад, модулі авторизації, обробки даних користувачів або взаємодії з базою даних мають значно вищий пріоритет тестування, ніж другорядні інформаційні сторінки чи допоміжний функціонал. За умов обмеженого часу тестування саме критичні компоненти повинні перевірятися в першу чергу та з максимально можливим рівнем покриття.

Для реалізації risk-based testing доцільно класифікувати функціональність веб-додатка за рівнем ризику. Високоризикові компоненти рекомендується тестувати із застосуванням як формалізованих тест-кейсів, так і додаткових дослідницьких перевірок. Для функціональності середнього рівня ризику можуть використовуватися скорочені набори тестових сценаріїв, а низькоризикові компоненти можуть перевірятися переважно під час регресійного тестування або exploratory testing. Такий підхід дозволяє більш раціонально розподіляти ресурси тестування та зосереджувати увагу на найбільш критичних ділянках системи.

Важливою рекомендацією є комбінування формалізованого та дослідницького тестування. Використання лише детально описаних тест-кейсів забезпечує повторюваність перевірок, однак часто обмежує тестувальника рамками заздалегідь визначених сценаріїв. У результаті можуть залишатися непоміченими нетипові дефекти, пов'язані з нестандартною поведінкою користувача або неочікуваними комбінаціями дій. Водночас виключно дослідницьке тестування ускладнює контроль тестового покриття та документування результатів перевірки.

З огляду на це доцільним є адаптивний підхід до ручного тестування, який передбачає варіативне поєднання різних методів перевірки залежно від типу функціональності, етапу розробки та рівня ризику. На ранніх етапах створення веб-додатка більш ефективним може бути exploratory testing, оскільки

функціональність ще нестабільна та часто змінюється. У процесі стабілізації системи доцільно поступово збільшувати частку формалізованих тест-кейсів і регресійних перевірок. Такий підхід дозволяє поєднати гнучкість дослідницького тестування з контрольованістю структурованих методів перевірки.

Окрему увагу слід приділяти оптимізації тестової документації. Надмірно деталізовані тест-кейси можуть суттєво збільшувати витрати часу на підтримку документації та ускладнювати її актуалізацію після змін функціональності. З іншого боку, недостатньо деталізовані сценарії можуть призводити до неоднозначного трактування дій тестувальниками та втрати повторюваності тестування. У практичній діяльності доцільно використовувати комбінований підхід, за якого для критичних бізнес-сценаріїв створюються повноцінні тест-кейси, а для другорядної функціональності застосовуються чек-листи або короткі сценарії перевірки.

Практика тестування веб-додатків також демонструє важливість повторного тестування та регресійної перевірки після внесення змін до програмного коду. Виправлення одного дефекту може спричинити появу нових помилок у суміжних компонентах системи, особливо в умовах складної взаємодії frontend і backend частин веб-додатка. Тому після усунення критичних дефектів доцільно виконувати не лише повторну перевірку виправленої функціональності, а й регресійне тестування пов'язаних модулів системи.

Для підвищення ефективності ручного тестування важливим є також використання сучасних інструментів підтримки QA-процесів. Системи управління тестуванням, зокрема TestRail або Jira, дозволяють централізовано зберігати тестову документацію, контролювати стан виконання тестів та відстежувати життєвий цикл дефектів. Інструменти на кшталт Postman і браузерних DevTools забезпечують можливість аналізу HTTP-запитів, тестування API, перевірки роботи клієнтської частини веб-додатка та локалізації помилок взаємодії між компонентами системи.

Окрему увагу доцільно приділяти перевірці безпеки веб-додатків навіть у межах ручного тестування. Тому до базового набору перевірок рекомендується

включати тестування типових вебвразливостей, зокрема SQL-ін'єкцій, XSS-атак, некоректної валідації введених даних, порушень контролю доступу та проблем із механізмами автентифікації. Використання рекомендацій OWASP дозволяє систематизувати процес перевірки безпеки та мінімізувати ризик пропуску критичних вразливостей.

Таким чином, підвищення ефективності ручного тестування веб-додатків потребує поєднання організаційних, методологічних та технічних підходів. Використання risk-based testing, адаптивного комбінування формалізованих і дослідницьких методів перевірки, стандартизація тестової документації, застосування сучасних QA-інструментів та налагодження ефективної взаємодії між учасниками команди дозволяють суттєво підвищити якість тестування, зменшити ризик пропуску критичних дефектів і забезпечити стабільну роботу веб-додатків у процесі експлуатації.

Важливим практичним аспектом підвищення ефективності ручного тестування є використання результатів попередніх перевірок під час планування нових тестових сценаріїв. Аналіз повторюваних дефектів і найбільш проблемних компонентів веб-додатка дозволяє визначати функціональність, яка потребує додаткового контролю, та більш раціонально розподіляти ресурси тестування. Це також сприяє зменшенню ризику повторної появи критичних помилок у процесі подальшої розробки системи.

Крім того, ефективність ручного тестування значною мірою залежить від гнучкості процесу тестування та здатності команди адаптуватися до змін у проєкті. Поєднання формалізованих тестових сценаріїв із exploratory testing дозволяє забезпечити як повторюваність перевірок, так і виявлення нетипових дефектів, що підвищує загальну якість тестування веб-додатків.

3.4 Оцінка ефективності запропонованих рекомендацій

Ефективність процесу ручного тестування веб-додатків значною мірою залежить від рівня організації тестової діяльності, структури тестової документації та обґрунтованості вибору методів перевірки. Запропоновані у попередньому підрозділі рекомендації спрямовані на підвищення якості тестування за рахунок систематизації процесу перевірки, раціонального використання ресурсів тестування та поєднання різних підходів до виявлення дефектів. Оцінка ефективності таких рекомендацій дозволяє визначити їх практичну доцільність і вплив на результативність процесу забезпечення якості веб-додатків.

Одним із ключових результатів застосування систематизованого підходу до ручного тестування є підвищення повноти тестового покриття. Використання структурованого алгоритму тестування, який охоплює етапи аналізу вимог, формування тестових сценаріїв, підготовки тестових даних, виконання перевірок та документування результатів, дозволяє зменшити ймовірність пропуску важливих функціональних сценаріїв. На відміну від хаотичного або виключно інтуїтивного тестування, системний підхід забезпечує послідовність виконання перевірок і створює умови для повторного використання тестових сценаріїв під час регресійного тестування.

Практична реалізація тестування демонстраційного веб-додатка показала, що використання структурованих тест-кейсів дозволяє ефективно виявляти як функціональні дефекти, так і проблеми, пов'язані з безпекою веб-застосунків. Під час тестування були виявлені дефекти валідації даних, помилки автентифікації, SQL-ін'єкції, XSS-вразливості та проблеми з обробкою користувацьких запитів. Це підтверджує, що формалізований підхід до ручного тестування дозволяє не лише перевіряти відповідність функціональності вимогам, а й своєчасно виявляти потенційно критичні вразливості системи.

Водночас проведене тестування показало обмеження використання виключно формалізованих тест-кейсів. Частина дефектів була виявлена саме під час exploratory testing, коли перевірка виконувалася без жорстко визначеного сценарію. Це підтверджує доцільність адаптивного підходу до ручного тестування,

який передбачає комбінування формалізованих і дослідницьких методів перевірки залежно від типу функціональності, рівня ризику та етапу розробки програмного забезпечення.

Використання risk-based testing також продемонструвало позитивний вплив на ефективність процесу тестування. Пріоритизація критичних компонентів веб-додатка дозволила зосередити основну увагу на функціональності, пов'язаній із автентифікацією користувачів, обробкою даних та взаємодією з базою даних. Такий підхід забезпечив більш раціональний розподіл часу тестування та підвищив ймовірність виявлення критичних дефектів на ранніх етапах перевірки. В умовах обмежених ресурсів це має особливе значення для практичної організації процесу manual QA testing.

Позитивний вплив на ефективність тестування також має використання структурованої тестової документації. Формалізовані тест-кейси, чек-листи та bug reports забезпечують повторюваність перевірок і спрощують процес аналізу результатів тестування. Крім того, документування дефектів у стандартизованій формі покращує взаємодію між тестувальниками та розробниками, оскільки дозволяє швидше локалізувати причини помилок і скоротити час їх виправлення. Наявність якісної тестової документації також спрощує процес регресійного тестування після внесення змін до програмного коду.

Оцінка ефективності запропонованих рекомендацій показує, що важливим фактором успішного ручного тестування є поєднання технічних і організаційних підходів. Використання сучасних інструментів підтримки QA-процесів, зокрема систем управління тестуванням, браузерних DevTools та засобів перевірки API, дозволяє спростити аналіз роботи веб-додатка та підвищити швидкість локалізації дефектів. Водночас самі інструменти не забезпечують якісного тестування без належної організації процесу перевірки та достатнього рівня підготовки тестувальників.

Проведений аналіз також підтвердив важливість перевірки безпеки веб-додатків у межах ручного тестування. Навіть у невеликих веб-системах можуть виникати критичні вразливості, пов'язані з некоректною обробкою

користувацьких даних або відсутністю належної валідації введеної інформації. Використання рекомендацій OWASP і включення базових security-перевірок до процесу ручного тестування дозволяє суттєво знизити ризик експлуатації типових вебвразливостей у реальних умовах використання програмного забезпечення.

Крім того, запропонований адаптивний підхід до ручного тестування сприяє більш гнучкій організації процесу забезпечення якості. Його використання дозволяє враховувати особливості конкретного проєкту, швидкість змін функціональності та рівень стабільності веб-додатка. Поєднання exploratory testing із формалізованими перевітками створює можливість одночасно забезпечувати контроль тестового покриття та виявляти дефекти, які можуть залишатися непоміченими під час виконання лише стандартних тестових сценаріїв.

Таким чином, оцінка ефективності запропонованих рекомендацій підтверджує доцільність використання систематизованого та адаптивного підходу до ручного тестування веб-додатків. Поєднання risk-based testing, структурованої тестової документації, формалізованих і дослідницьких методів перевірки, а також використання сучасних інструментів підтримки QA-процесів дозволяє підвищити якість тестування, забезпечити більш повне покриття функціональності та зменшити ризик пропуску критичних дефектів у процесі розробки веб-додатків.

4 ЕКОНОМІЧНЕ ОБГРУНТУВАННЯ ЕФЕКТИВНОСТІ РУЧНОГО ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

4.1 Аналіз витрат на організацію процесу ручного тестування

Організація процесу ручного тестування веб-додатків потребує використання трудових, часових і програмно-технічних ресурсів, що формують загальні витрати на забезпечення якості програмного забезпечення. Незважаючи на відносну простоту ручного тестування порівняно з автоматизованими підходами, його ефективність безпосередньо залежить від рівня організації процесу перевірки, кваліфікації тестувальників та використання допоміжних інструментів. У сучасних умовах розробки веб-додатків ручне тестування залишається необхідним етапом забезпечення якості, особливо під час перевірки користувацького інтерфейсу, usability, бізнес-логіки та нетипових сценаріїв взаємодії із системою.

Витрати на організацію ручного тестування формуються з декількох основних складових: оплати праці тестувальників, витрат часу на підготовку тестової документації, використання програмних інструментів підтримки тестування, а також витрат, пов'язаних із повторним тестуванням і регресійними перевітками після виправлення дефектів. Крім того, під час оцінки економічної доцільності тестування необхідно враховувати непрямі витрати, пов'язані з можливими наслідками пропуску критичних дефектів у продуктивне середовище.

Для оцінки витрат на проведення ручного тестування демонстраційного веб-додатка приймемо, що процес тестування виконується одним manual QA engineer протягом повного циклу перевірки системи. До складу робіт входять:

- аналіз функціональних вимог;
- формування тестових сценаріїв;
- створення тест-кейсів;
- виконання тестування;
- оформлення bug reports;

- повторне тестування після виправлення дефектів;
- регресійна перевірка системи.

Загальні трудові витрати на тестування визначаються за формулою:

$$T = T_1 + T_2 + T_3 + T_4 + T_5 + T_6 + T_7 \quad (4.1)$$

де:

T_1 – час на аналіз вимог;

T_2 – час на створення тест-кейсів;

T_3 – час на підготовку тестових даних;

T_4 – час на виконання функціонального тестування;

T_5 – час на оформлення звітів про дефекти;

T_6 – час на повторне тестування;

T_7 – час на регресійне тестування.

Для демонстраційного веб-додатка приймемо такі значення:

$$T = 4 + 8 + 3 + 14 + 5 + 4 + 6 = 44 \text{ год}$$

Таким чином, сумарні трудові витрати на проведення повного циклу ручного тестування становлять 44 години.

Вартість оплати праці тестувальника визначається за формулою:

$$C_{salary} = T \times R \quad (4.2)$$

де:

T – сумарний час тестування;

R – погодинна ставка manual QA engineer.

Для розрахунків приймемо середню погодинну ставку тестувальника на рівні 300 грн/год.

$$C_{salary} = 44 \times 300 = 13200 \text{ грн}$$

Окрім оплати праці, під час організації процесу тестування використовуються допоміжні програмні засоби, зокрема браузерні DevTools, системи управління дефектами, середовища для перевірки API та засоби організації тестової документації. Незважаючи на те, що частина інструментів

може використовуватися безкоштовно, у реальних умовах розробки часто застосовуються комерційні рішення або корпоративні підписки.

Витрати на використання програмних інструментів можна визначити за формулою:

$$C_{tools} = \sum_{i=1}^n C_i \quad (4.3)$$

де:

C_i – вартість використання окремого інструмента тестування.

Для спрощення розрахунків прийmemo:

- Jira – 600 грн/місяць;
- TestRail – 900 грн/місяць;
- Postman – 300 грн/місяць.

Тоді:

$$C_{tools} = 600 + 900 + 300 = 1800 \text{ грн}$$

Крім прямих витрат, необхідно враховувати витрати, пов'язані з повторним тестуванням після виправлення дефектів. У процесі тестування демонстраційного веб-додатка було виявлено декілька критичних помилок, пов'язаних із SQL-ін'єкціями, XSS-вразливостями та некоректною валідацією введених даних. Усунення таких дефектів потребує додаткового часу на перевірку виправлень і проведення регресійного тестування.

Витрати на повторне тестування визначаються за формулою:

$$C_{retest} = T_{retest} \times R \quad (4.4)$$

де:

T_{retest} – час, витрачений на повторні перевірки;

R – погодинна ставка тестувальника.

Для розрахунку прийmemo:

$$C_{retest} = 10 \times 300 = 3000 \text{ грн}$$

Загальні витрати на організацію процесу ручного тестування визначаються як сума всіх складових:

$$C_{total} = C_{salary} + C_{tools} + C_{retest} \quad (4.5)$$

$$C_{total} = 13200 + 1800 + 3000 = 18000 \text{ грн}$$

Отримані результати свідчать про те, що організація повного циклу ручного тестування навіть для відносно невеликого веб-додатка потребує помітних витрат ресурсів. Водночас ці витрати є економічно обґрунтованими, оскільки дозволяють своєчасно виявляти функціональні дефекти та вразливості безпеки ще до передачі системи користувачам. Виявлення критичних помилок на ранніх етапах розробки дозволяє уникнути значно більших витрат, пов'язаних із виправленням дефектів після релізу, простоєм системи або компрометацією користувацьких даних.

Крім того, використання структурованого підходу до ручного тестування та якісної тестової документації дозволяє зменшити витрати часу на повторні перевірки, спростити процес регресійного тестування та підвищити загальну ефективність забезпечення якості веб-додатків. Саме тому витрати на організацію процесу тестування слід розглядати не як додаткові втрати, а як інвестицію у стабільність, безпеку та надійність програмного продукту.

4.2 Оцінка економічного ефекту від своєчасного виявлення дефектів

Однією з головних цілей тестування програмного забезпечення є своєчасне виявлення дефектів до моменту передачі системи кінцевим користувачам. Практика розробки програмного забезпечення показує, що вартість виправлення помилок суттєво зростає залежно від етапу життєвого циклу, на якому ці помилки були виявлені. Якщо дефект усувається під час тестування, витрати обмежуються часом роботи розробника та тестувальника. У випадку ж виявлення помилки після релізу додатково виникають витрати, пов'язані з технічною підтримкою, повторним розгортанням системи, обробкою звернень користувачів, репутаційними ризиками та можливими фінансовими втратами.

Особливо критичними для веб-додатків є дефекти, пов'язані з безпекою, оскільки їх експлуатація може призвести до компрометації даних користувачів або порушення роботи системи. У процесі тестування демонстраційного веб-додатка були виявлені SQL-ін'єкції, XSS-вразливості та помилки валідації введених даних. Умовно припустимо, що подібні дефекти були б виявлені вже після розгортання веб-додатка у продуктивному середовищі. У такому випадку витрати на їх усунення були б значно вищими через необхідність проведення термінового виправлення системи, повторного тестування та усунення наслідків можливої компрометації даних.

Для оцінки економічного ефекту ручного тестування визначимо середню вартість виправлення одного дефекту на різних етапах життєвого циклу програмного забезпечення. Прийmemo такі умовні значення:

- виправлення дефекту під час тестування – 700 грн;
- виправлення дефекту після релізу – 6000 грн.

Різниця між цими величинами фактично визначає економію від своєчасного виявлення помилки:

$$E_1 = C_{prod} - C_{test}$$

де:

E_1 – економічний ефект від виявлення одного дефекту під час тестування;

C_{prod} – вартість виправлення дефекту після релізу;

C_{test} – вартість виправлення дефекту під час тестування.

Підставимо значення:

$$E_1 = 6000 - 700 = 5300 \text{ грн}$$

Таким чином, своєчасне виявлення одного дефекту дозволяє уникнути додаткових витрат у розмірі 5300 грн.

У процесі ручного тестування демонстраційного веб-додатка було виявлено:

- 3 критичних дефекти безпеки;
- 5 функціональних дефектів;
- 4 помилки валідації та обробки введених даних.

Загальна кількість виявлених дефектів становить:

$$N = 3 + 5 + 4 = 12$$

Тоді сумарний економічний ефект від своєчасного виявлення дефектів визначається за формулою:

$$E_{total} = E_1 \times N$$

де:

E_{total} – загальний економічний ефект;

N – кількість дефектів, виявлених до релізу.

Підставимо значення:

$$E_{total} = 5300 \times 12 = 63600 \text{ грн}$$

Отриманий результат свідчить про те, що проведення ручного тестування дозволило потенційно уникнути витрат у розмірі 63600 грн, які могли б виникнути у випадку виявлення дефектів уже після введення системи в експлуатацію.

Для оцінки економічної доцільності ручного тестування визначимо коефіцієнт економічної ефективності:

$$K_{eff} = \frac{E_{total}}{C_{total}}$$

де:

K_{eff} – коефіцієнт економічної ефективності;

E_{total} – економічний ефект від виявлення дефектів;

C_{total} – загальні витрати на організацію тестування.

Тоді:

$$K_{eff} = \frac{63600}{18000} \approx 3.53$$

Отримане значення означає, що кожна гривня, витрачена на організацію ручного тестування, потенційно дозволяє уникнути понад 3,5 гривні додаткових витрат, пов'язаних із виправленням дефектів після релізу системи.

Окремо слід враховувати непрямі економічні втрати, які можуть виникати через наявність критичних дефектів у веб-додатках. До таких втрат належать:

- зниження довіри користувачів;
- погіршення репутації компанії;
- можливі фінансові втрати через недоступність системи;
- ризик витоку конфіденційної інформації;
- додаткові витрати на технічну підтримку.

У випадку вразливостей типу SQL Injection або Stored XSS потенційні збитки можуть суттєво перевищувати прямі витрати на виправлення програмного коду. Саме тому перевірка безпеки веб-додатків є важливою складовою ручного тестування навіть для відносно невеликих програмних систем.

Використання структурованого та адаптивного підходу до ручного тестування також дозволяє зменшити витрати на подальше супроводження програмного забезпечення. Наявність тестової документації, повторно використовуваних тест-кейсів і чітко задокументованих bug reports спрощує проведення регресійного тестування після внесення змін до системи та скорочує час локалізації дефектів. Це особливо важливо в умовах Agile-розробки, де функціональність веб-додатків постійно змінюється та доповнюється.

Крім того, результати проведеного аналізу підтверджують економічну доцільність використання risk-based testing. Пріоритизація найбільш критичних компонентів системи дозволяє виявляти найбільш небезпечні дефекти на ранніх етапах перевірки та раціонально використовувати ресурси тестування. Такий підхід забезпечує більш ефективне використання часу тестувальників і зменшує ризик пропуску помилок, які можуть призвести до найбільших фінансових втрат.

Таким чином, проведена оцінка показує, що ручне тестування веб-додатків є не лише технічно необхідним, а й економічно обґрунтованим процесом. Своєчасне виявлення дефектів дозволяє суттєво зменшити витрати на супроводження програмного забезпечення після релізу, підвищити стабільність роботи системи та мінімізувати ризики, пов'язані з експлуатацією критичних помилок у реальних умовах використання веб-додатків.

4.3 Оцінка ефективності запропонованого підходу до ручного тестування

Ефективність процесу ручного тестування значною мірою залежить від рівня організації перевірок, структури тестової документації та правильного вибору підходів до виконання тестових сценаріїв. У межах даної роботи було запропоновано систематизований підхід до ручного тестування веб-додатків, який поєднує формалізовані методи перевірки, risk-based testing та елементи exploratory testing. Такий підхід дозволяє підвищити повноту тестового покриття та забезпечити більш ефективне виявлення дефектів під час перевірки веб-додатків.

Запропонований підхід базується на адаптивному принципі організації тестування, відповідно до якого методи перевірки обираються залежно від критичності функціональності, типу тестованого компонента та етапу життєвого циклу програмного забезпечення. Для критичних компонентів системи використовуються детально формалізовані тест-кейси та повторювані регресійні перевірки, тоді як для менш критичних функцій допускається використання більш гнучких дослідницьких методів тестування. Це дозволяє раціонально використовувати ресурси тестування та зосереджувати основну увагу на найбільш ризикових компонентах веб-додатка.

Під час практичної реалізації тестування демонстраційного веб-додатка застосування структурованого підходу дозволило виявити низку критичних дефектів, пов'язаних із безпекою, обробкою введених даних та порушенням бізнес-логіки системи. Зокрема, були виявлені SQL-ін'єкції, Stored XSS-вразливості, помилки авторизації та недоліки валідації користувацьких даних. Виявлення таких дефектів на етапі тестування дозволило уникнути потенційних ризиків, пов'язаних із експлуатацією системи в продуктивному середовищі.

Для оцінки ефективності запропонованого підходу порівнюємо результати неструктурованого тестування та тестування із застосуванням сформованого алгоритму ручної перевірки (таблиця 4.1).

Таблиця 4.1

Показник	Без системного підходу	Із запропонованим підходом
Кількість виявлених дефектів	7	12
Критичні дефекти безпеки	1	3
Повторно відкриті дефекти	4	1
Час регресійного тестування	14 год	8 год
Повнота тестового покриття	середня	висока

Результати демонструють, що використання структурованої документації та адаптивного підходу до ручного тестування дозволяє підвищити ефективність виявлення дефектів і скоротити витрати часу на повторні перевірки. Особливо важливим є зменшення кількості повторних дефектів, оскільки це свідчить про підвищення якості первинного тестування та точності оформлення bug reports.

Окрему роль у підвищенні ефективності тестування відіграє використання risk-based testing. Пріоритизація тестових сценаріїв дозволяє в першу чергу перевіряти найбільш критичні компоненти веб-додатка, що зменшує ризик пропуску небезпечних дефектів. Такий підхід є особливо актуальним у проєктах із обмеженими часовими ресурсами, коли повне тестове покриття всіх можливих сценаріїв є практично недосяжним.

Крім того, поєднання формалізованих тест-кейсів із exploratory testing дозволяє підвищити гнучкість процесу тестування. Формалізовані сценаріїв забезпечують повторюваність перевірок та контроль функціонального покриття, тоді як дослідницьке тестування дозволяє виявляти нетипові дефекти, які можуть залишатися непоміченими під час виконання заздалегідь визначених сценаріїв. Практика тестування демонстраційного веб-додатка показала, що саме exploratory testing дозволило виявити частину вразливостей, пов'язаних із некоректною обробкою введених даних і поведінкою системи в нестандартних умовах.

Важливою перевагою адаптивного підходу є також покращення взаємодії між учасниками процесу розробки програмного забезпечення, оскільки він дозволив спростити передачу інформації між тестувальниками та розробниками,

зменшити неоднозначність під час опису і пришвидшити процес виправлення дефектів, що позитивно впливає на забезпечення якості програмного забезпечення.

Таким чином, використання адаптивного поєднання формалізованих і дослідницьких методів тестування, структурованої тестової документації та risk-based testing дозволяє підвищити якість тестового покриття, зменшити ризик пропуску дефектів і раціональніше використовувати ресурси тестування.

5 ЗАХОДИ З ОХОРОНИ ПРАЦІ

5.1 Оцінка умов праці під час ручного тестування веб-додатків

У процесі виконання робіт, пов'язаних із ручним тестуванням веб-додатків, працівник більшу частину робочого часу проводить за персональним комп'ютером, використовуючи спеціалізоване програмне забезпечення для перевірки функціональності системи, оформлення тестової документації та аналізу результатів тестування. Характер роботи передбачає постійну взаємодію з вебінтерфейсами, базами даних, браузерними засобами налагодження, системами керування дефектами та офісними програмами. Робота належить до категорії розумової праці з підвищеним навантаженням на органи зору та нервову систему.

Робоче місце Manual QA Engineer зазвичай організовується в офісному приміщенні, обладнаному персональними комп'ютерами, засобами мережевої комунікації та оргтехнікою. Для забезпечення комфортних умов праці приміщення повинно мати достатню площу, природне та штучне освітлення, систему вентиляції й відповідати санітарно-гігієнічним вимогам щодо організації робочих місць користувачів персональних комп'ютерів. Типове приміщення може мати площу близько 50–70 м², висоту не менше 2,5 м та бути розрахованим на декілька робочих місць. Стіни й стеля повинні мати світлі матові поверхні, що зменшують рівень відблисків, а покриття підлоги – антистатичні властивості.

У приміщенні можуть розташовуватися персональні комп'ютери, монітори, маршрутизатори, мережеве обладнання, принтери та інші технічні засоби, необхідні для організації процесу тестування. Тривала робота з комп'ютерною технікою супроводжується впливом ряду несприятливих виробничих факторів, які можуть негативно впливати на стан здоров'я працівника та його працездатність.

До основних фізичних факторів належать електромагнітні випромінювання від моніторів і електронного обладнання, підвищене навантаження на органи зору, статична електрика, шум від систем охолодження комп'ютерної техніки та

недостатня або нерівномірна освітленість робочої зони. Тривалий вплив таких факторів може спричиняти втому, погіршення концентрації уваги, головний біль, подразнення очей та зниження продуктивності праці. Особливу небезпеку становить електрична мережа напругою 230 В, пошкодження якої може призвести до ураження електричним струмом.

До хімічних факторів належать накопичення пилу в приміщенні, зниження рівня аероіонізації повітря та можливе утворення озону й інших сполук під час роботи електронного обладнання. Недостатня якість повітря негативно впливає на дихальну систему працівників, а також сприяє швидкому виникненню втоми та зниженню працездатності. Особливого значення набуває регулярне провітрювання приміщення та підтримання належного рівня вологості повітря.

Психофізіологічні фактори є одними з найбільш характерних для діяльності тестувальника програмного забезпечення. До них належать тривале перебування у сидячому положенні, монотонність роботи, високе інтелектуальне навантаження, необхідність постійної концентрації уваги та значний обсяг інформації, що аналізується. Постійна робота з інтерфейсами веб-додатків і повторювані перевірки можуть спричиняти емоційне виснаження, зниження уважності та підвищення ризику пропуску дефектів. Крім того, неправильна організація робочого місця здатна викликати порушення постави, болі у спині, шиї та зап'ястях.

Таким чином, діяльність, пов'язана з ручним тестуванням веб-додатків, супроводжується впливом комплексу фізичних, хімічних та психофізіологічних факторів, що потребує дотримання вимог охорони праці, ергономіки та санітарно-гігієнічних норм для забезпечення безпечних і комфортних умов роботи.

5.2 Заходи щодо забезпечення безпечних умов праці

Для забезпечення безпечних та комфортних умов праці під час виконання робіт, пов'язаних із ручним тестуванням веб-додатків, необхідно дотримуватися вимог ергономіки, електробезпеки, санітарно-гігієнічних норм та правил організації робочого місця користувача персонального комп'ютера. Оскільки діяльність manual QA engineer характеризується значним інтелектуальним навантаженням і тривалою роботою за відеодисплейними терміналами, особливого значення набуває правильна організація робочого простору та мінімізація впливу несприятливих виробничих факторів.

Робоче місце тестувальника повинно забезпечувати зручне розташування обладнання, достатній простір для виконання робочих операцій та можливість підтримання правильної робочої пози. Під час організації робочого місця необхідно враховувати положення тіла працівника, особливості розташування монітора, клавіатури, маніпулятора «миша», а також можливість використання документації та допоміжних матеріалів без надмірного фізичного навантаження.

Конструкція робочого столу повинна відповідати ергономічним вимогам та забезпечувати оптимальне розміщення комп'ютерної техніки. Рекомендована висота робочої поверхні столу становить 680–800 мм, ширина – 600–1400 мм, а глибина – 800–1000 мм. Простір для ніг має бути достатнім для вільного положення працівника: висота – не менше 600 мм, ширина – не менше 500 мм. Поверхня столу повинна бути матовою та не створювати відблисків під час роботи з монітором.

Робоче крісло повинно мати регульовану висоту сидіння, спинку та підлокітники. Висота сидіння має регулюватися в межах 400–500 мм, а кут нахилу спинки – у межах 0–30° від вертикального положення. Конструкція крісла повинна забезпечувати підтримання природного положення хребта та зменшення статичного навантаження на м'язи спини й шиї. Для зменшення навантаження на нижні кінцівки доцільно використовувати підставку для ніг із можливістю регулювання висоти та кута нахилу.

Монітор необхідно розташовувати таким чином, щоб верхній край екрана знаходився приблизно на рівні очей користувача, а погляд був спрямований трохи

вниз. Відстань від очей до екрана повинна становити 60–70 см залежно від діагоналі монітора та розміру елементів інтерфейсу. Клавіатуру слід розміщувати на відстані приблизно 100–150 мм від краю столу, а кут її нахилу має перебувати в межах 5–10°. Таке розташування обладнання дозволяє знизити навантаження на кисті рук, шию та органи зору.

Освітлення робочого приміщення повинно бути комбінованим – природним і штучним. Робочі місця необхідно розташовувати таким чином, щоб уникати потрапляння прямих сонячних променів на екран монітора та виникнення відблисків. Для штучного освітлення рекомендується використовувати рівномірне розсіяне світло. Недостатнє або надмірне освітлення негативно впливає на працездатність працівника і може спричиняти швидке перевтомлення органів зору.

Для підтримання сприятливого мікроклімату необхідно забезпечити належну вентиляцію приміщення та регулярне провітрювання. Температура повітря в робочому приміщенні повинна підтримуватися в межах 20–24°C, а відносна вологість – 40–60%. Особливу увагу необхідно приділяти контролю рівня пилу та чистоті обладнання, оскільки накопичення пилу негативно впливає як на стан комп'ютерної техніки, так і на здоров'я працівників.

Важливим аспектом безпеки є дотримання вимог електробезпеки. Усе обладнання повинно бути справним, мати заземлення та захист від короткого замикання й перевантаження. Забороняється експлуатація пошкоджених кабелів, несправних розеток або блоків живлення. Для підключення комп'ютерної техніки доцільно використовувати мережеві фільтри та джерела безперебійного живлення, які забезпечують захист обладнання від перепадів напруги.

З метою зниження психофізіологічного навантаження необхідно організовувати регламентовані перерви під час роботи за комп'ютером. Для працівників, діяльність яких пов'язана з постійною роботою за відеодисплейними терміналами, рекомендується робити перерви тривалістю 10–15 хвилин через кожні 1–2 години роботи. Під час перерв доцільно виконувати вправи для очей, змінювати положення тіла та здійснювати коротку рухову активність для профілактики втоми й захворювань опорно-рухового апарату.

Для забезпечення пожежної безпеки приміщення повинно бути оснащено первинними засобами пожежогасіння, а електричне обладнання – відповідати вимогам пожежної безпеки. Працівники повинні бути ознайомлені з правилами евакуації та діями у випадку виникнення аварійної ситуації.

Таким чином, раціональна організація робочого місця сприяє підвищенню продуктивності праці, зниженню рівня втоми та мінімізації негативного впливу виробничих факторів на здоров'я працівників.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було досліджено теоретичні та практичні аспекти ручного тестування веб-додатків у процесі забезпечення якості програмного забезпечення. Проведений аналіз сучасних підходів до тестування показав, що ручне тестування залишається важливою складовою процесу контролю якості навіть в умовах активного розвитку автоматизованих засобів перевірки програмного забезпечення. Це пояснюється необхідністю оцінювання поведінки веб-додатків з позиції кінцевого користувача, перевірки usability, логіки взаємодії інтерфейсу та виявлення нетипових сценаріїв роботи системи.

У першому розділі було розглянуто поняття якості програмного забезпечення, визначено роль тестування у процесі забезпечення якості та проаналізовано взаємозв'язок між QA, QC і testing. Досліджено основні види та рівні тестування програмного забезпечення, а також особливості ручного тестування як окремого напрямку діяльності manual QA engineer. Окрема увага була приділена аналізу сучасних досліджень і підходів у сфері тестування веб-додатків, зокрема концепціям risk-based testing, exploratory testing, Agile-підходам та практикам безперервного тестування.

У другому розділі досліджено специфіку веб-додатків як об'єктів тестування. Проаналізовано архітектурні особливості сучасних веб-систем, принципи клієнт-серверної взаємодії, особливості frontend і backend компонентів, а також специфіку тестування адаптивних інтерфейсів і веб-застосунків у різних браузерах та на різних пристроях. Було розглянуто основні напрями ручного тестування веб-додатків, методи та техніки проведення перевірок, а також типові дефекти, що виникають у процесі розробки веб-систем. Додатково проаналізовано сучасні інструменти та тестову документацію, які використовуються manual QA engineers у практичній діяльності.

У третьому розділі сформовано узагальнений алгоритм ручного тестування веб-додатків, що охоплює етапи аналізу вимог, формування тестових сценаріїв, підготовки тестових даних, виконання перевірок, документування результатів тестування та проведення регресійної перевірки. На основі проведеного аналізу було запропоновано адаптивний підхід до ручного тестування, який передбачає комбінування формалізованих методів перевірки з *exploratory testing* залежно від рівня ризику функціональності, етапу життєвого циклу програмного забезпечення та типу тестованих сценаріїв. Такий підхід дозволяє підвищити ефективність процесу тестування та більш раціонально використовувати ресурси команди забезпечення якості.

Практична частина роботи полягала у розробці демонстраційного веб-додатка та проведенні його ручного тестування. У межах тестування було сформовано тестові сценарії, створено тест-кейси, проведено перевірку функціональних можливостей системи та оформлено *bug reports* для виявлених дефектів. Під час тестування були виявлені помилки, пов'язані з валідацією даних, обробкою сесій, SQL-ін'єкціями та XSS-вразливостями. Після усунення дефектів було виконано повторне та регресійне тестування, результати якого підтвердили коректність роботи виправленої версії веб-додатка.

На основі проведеного дослідження було сформульовано практичні рекомендації щодо підвищення ефективності ручного тестування веб-додатків. Визначено доцільність використання *risk-based testing* для пріоритизації тестових сценаріїв, необхідність підтримки структурованої тестової документації та важливість комбінування формалізованих і дослідницьких підходів до тестування. Також було обґрунтовано значення ефективної взаємодії між тестувальниками, розробниками та іншими учасниками процесу розробки програмного забезпечення.

В економічній частині роботи було проведено оцінку витрат на організацію процесу ручного тестування веб-додатків та визначено економічну доцільність впровадження структурованого підходу до тестування. Виконані розрахунки показали, що використання систематизованого процесу тестування дозволяє

зменшити витрати, пов'язані з виправленням дефектів на пізніх етапах розробки, а також знизити ризики виникнення критичних помилок під час експлуатації програмного забезпечення.

У розділі з охорони праці та безпеки в надзвичайних ситуаціях було проаналізовано умови праці фахівців, діяльність яких пов'язана з тривалою роботою за комп'ютером. Розглянуто основні шкідливі та небезпечні фактори, що можуть впливати на працівників у процесі виконання професійних обов'язків, а також сформульовано рекомендації щодо організації безпечного та ергономічного робочого місця.

Отже, поставлену мету роботи досягнуто, а всі визначені завдання виконано. Проведене дослідження підтвердило, що ручне тестування залишається важливим інструментом забезпечення якості веб-додатків, а використання адаптивного підходу, структурованої тестової документації та комплексної організації процесу тестування дозволяє підвищити ефективність виявлення дефектів і забезпечити більш високий рівень надійності програмного забезпечення.

ПЕРЕЛІК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи якості програмного забезпечення. Луцький національний технічний університет. URL: https://elib.lntu.edu.ua/sites/default/files/elib_upload/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F/page4.html.
2. Якість програмного забезпечення (за ISO/IEC 25010:2011). Центр підготовки ІТ фахівців QALight. URL: <https://qalight.ua/baza-znaniy/yakist-programnogo-zabezpechennya-za-iso-iec-250102011/>.
3. Лапець О.В., Полонська А.Є., Сірик С.Ф. Роль якості програмного забезпечення та тестування у створенні успішних цифрових рішень. Актуальні проблеми автоматизації та інформаційних технологій. Том 28. 2024. С. 161–165.
4. Мірошник М.А., Галайчук Ю.В. Моделі і методи оцінювання якості програмного забезпечення, що використовує алгоритми штучного інтелекту. Інформаційно-керуючі системи на залізничному транспорті, 2023. № 3. С. 38–39. URL: <http://lib.kart.edu.ua/bitstream/123456789/18565/3/%D0%9C%D1%96%D1%80%D0%BE%D1%88%D0%BD%D0%B8%D0%BA.pdf>.
5. Панченко Т.Д., Тузова І.А., Тузов О.В., Чумак О.А., Стародуб В.І. Аналіз моделей якості програмного забезпечення. Scientific Collection «InterConf+», 2024. № 43(193), С. 560–569. DOI: 10.51582/interconf.19-20.03.2024.054.
6. Смагіна О.О. Переяславська С.О. Якість програмного забезпечення та тестування : навч. посіб. до вивчення дисц. для студ. спец. 121 – „Інженерія програмного забезпечення”; Держ. закл. „Луган. нац. ун-т імені Тараса Шевченка”. Старобільськ : ДЗ „ЛНУ імені Тараса Шевченка”, 2021. 286 с.
7. Популярні життєві цикли розробки ПЗ. QATestLab. URL: <https://training.qatestlab.com/blog/technical-articles/popular-software-development-life-cycles/>.

8. Zosym Maxym. Життєвий цикл розробки програмного забезпечення (Software Development Life Cycle - SDLC). Maxym Zosym. URL: <https://www.maxzosim.com/software-development-life-cycle-sdlc/>.
9. Життєвий цикл програмного забезпечення. JavaRush. URL: <https://javarush.com/ua/quests/lectures/ua.questservlets.level15.lecture00>.
10. Життєвий цикл розробки програмного забезпечення за Agile-методологією. Що це таке і які фази охоплює? EPAM Systems. URL: <https://campus.epam.ua/ua/blog/581>.
11. Життєвий цикл ПЗ. Центр підготовки ІТ фахівців QALight. URL: <https://qalight.ua/baza-znaniy/zhittjevij-tsikl-pz/>.
12. QA, QC і тестування. Центр підготовки ІТ фахівців QALight. URL: <https://qalight.ua/baza-znaniy/qa-qc-i-testuvannya/>.
13. QA і QC: їхня роль і відмінності в процесі розробки ПЗ. FoxmindEd. URL: <https://foxminded.ua/riznytsia-mizh-qa-ta-qc/>.
14. Клімук Ю. Quality Assurance vs Quality Control: просте пояснення. Комп'ютерна школа Hillel. URL: <https://blog.ithillel.ua/articles/quality-assurance-control>.
15. Oleg Zarevych. Що таке Software Testing? Medium. URL: <https://olegzarevych.medium.com/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-software-testing-f1fc14f6b8db>.
16. Тестування програмного забезпечення: типи, види та застосування. FoxmindEd. URL: <https://foxminded.ua/testuvannia-prohramnoho-zabezpechennia/>.
17. Види тестування та відмінності між ними. Шпаргалка з тестування. QualityAssuranceGroup. URL: <https://qagroup.com.ua/publications/vydy-testuvannya-ta-vidminnosti-mizh-nymy/>.
18. Nataliia Popelyshko. Всі відомі типи тестування (100+ штук). DOU.ua. URL: <https://dou.ua/forums/topic/40666/>.
19. Донич Д. Види тестування програмного забезпечення. LemonSchool. URL: <https://lemon.school/blog/vydy-testuvannya-programnogo-zabezpechennya>.

20. Рівні тестування. Q&A : Навчальний ресурс з тестування програмного забезпечення. URL: <https://qlearning.com.ua/theory/lectures/material/testing-levels/>.
21. Мелешкевич С. Рівні і методи тестування ПЗ. Avada Media. URL: <https://avada-media.ua/blog/urovni-i-metody-testirovaniya-po/#rivni-i-metodi-testuvannya-pz>.
22. Які існують рівні тестування? DevZone. URL: <https://devzone.org.ua/qna/iaki-isnuiut-rivni-testuvannia>.
23. Ручне та автоматизоване тестування. QALight. URL: <https://qalight.ua/baza-znaniy/ruchne-ta-avtomatizovane-testuvannya/>.
24. Ручне і автоматизоване тестування: в чому різниця та який підхід обрати. Академія для героїв. URL: <https://www.academy4heroes.com/novyny/ruchne-i-avtomatyzovane-testuvannya/>.
25. Багрій Р.О., Петровський С.С. Особливості сучасного тестування веб-додатків. Вісник Хмельницького національного університету, 2022. № 3(309). С. 70–74. DOI: 10.31891/2307-5732-2022-309-3-70-74.
26. Толкачова А.Ю., Піскозуб А.З. Методи тестування безпеки веб-застосунків. Кібербезпека: освіта, наука, техніка : електронне фахове наукове видання, 2024. № 2(26). С. 115–122. DOI: 0.28925/2663-4023.2024.26.668.
27. Хома В.В., Партика А.І., Сабодашко Д.В. Систематизація та порівняльний аналіз інструментів автоматизованого тестування на проникнення веб-додатків. Кібербезпека: освіта, наука, техніка : електронне фахове наукове видання, 2025. № 3(31). С. 717–726. DOI: 10.28925/2663-4023.2025.31.1068.
28. Рогатовська А.А. Дослідження методів тестування веб-додатків для покращення їх якості. : кваліфікаційна робота магістра : 121 «Інженерія програмного забезпечення» / Херсонський національний технічний університет. Хмельницький, 2025. 96 с URL: <https://eir.kntu.net.ua/jspui/bitstream/123456789/2448/1/%D0%A0%D0%BE%D0%B3%D0%B0%D1%82%D0%BE%D0%B2%D1%81%D1%8C%D0%BA%D0%B0%20%D0%90.%20%D0%90.%20%D0%94%D0%BE%D1%81%D0%BB%D1%96%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F%20%D0%BC%D0%B5%D1%82>

%D0%BE%D0%B4%D1%96%D0%B2%20%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%20%D0%B2%D0%B5%D0%B1-

%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D1%96%D0%B2%20%D0%B4%D0%BB%D1%8F%20%D0%BF%D0%BE%D0%BA%D1%80%D0%B0%D1%89%D0%B5%D0%BD%D0%BD%D1%8F%20%D1%97%D1%85%20%D1%8F%D0%BA%D0%BE%D1%81%D1%82%D1%96.pdf

29. Рудь А.Є. Розробка методики тестування програмного забезпечення у контексті ефективності та практичного застосування для різних сценаріїв у веб-додатках : кваліфікаційна робота магістра : 122 «Комп'ютерні науки» / Університет митної справи та фінансів. Дніпро, 2025. 77 с. URI: <http://biblio.umsf.dp.ua/xmlui/handle/123456789/7428>.

30. Лабо (Черній) В.Р. Система мультикритеріального тестування веб-додатків : кваліфікаційна робота магістра : 8.091501 «Комп'ютерні системи та мережі» / Тернопільський національний економічний університет. Тернопіль, 2017. 102 с. URL: <https://api.dspace.wunu.edu.ua/api/core/bitstreams/b8cc44f7-9fa8-4019-98c0-2c8fca668b74/content>.

31. Княжина Р. Архітектура вебзастосунків 2024: Ультимативний гайд для розробників. Laba LLC. URL: <https://robotdreams.cc/uk/blog/567-arhitektura-vebzastosunkiv>.

32. Purvi Parmar. Web Application Architecture: A Complete Guide for Developers and CTOs. LinkedIn. URL: <https://ua.linkedin.com/pulse/web-application-architecture-complete-guide-ctos-purvi-parmar-5rj6f>.

33. Волошин О., Добринін М. Як обрати архітектуру для веб додатку. Комп'ютерна школа Hillel. URL: <https://blog.ithillel.ua/articles/web-application-architecture>.

34. Felix Rose-Collins. Understanding Modern Web App Architectures. Ranktracker. URL: <https://www.ranktracker.com/blog/understanding-modern-web-app-architectures/>.

35. Архітектура веб-сайту. TestMatick LTD. URL: <https://testmatick.com/uk/glossary/arhitektura-veb-sajtu/>.

36. Тестування веб-проектів: основні етапи та поради. QALight. URL: <https://qalight.ua/baza-znaniy/testuvannya-veb-proektiv-osnovni-etapi-ta-poradi/>.

37. Перхулин І. Ручне тестування веб-сайтів та веб-додатків. Scribd Inc. URL:

<https://www.scribd.com/document/643044573/%D0%A0%D1%83%D1%87%D0%BD%D0%B5-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B2%D0%B5%D0%B1-%D1%81%D0%B0%D0%B9%D1%82%D1%96%D0%B2-%D1%82%D0%B0-%D0%B2%D0%B5%D0%B1-%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D1%96%D0%B2-pdf>.

38. Притула Б. Як правильно тестувати сайт: чек-лист із прикладами. Brander. URL: <https://brander.ua/blog/yak-pravylno-testuvaty-sayt-chek-lyst-iz-prykladamy>.

39. testRigor. Manual Testing: A Beginner's Guide. LinkedIn. URL: <https://ua.linkedin.com/pulse/manual-testing-beginners-guide-testrigor-3ybve>.

40. Проблеми UX/UI дизайну і методи їх вирішення: Як створити зручний та інтуїтивний сайт. GL.ua. URL: <https://gl.ua/blog/problemy-uxui-dyzaynu-i-metody-yikh-vyrishennya-yak-stvoryty>.

41. 404 та інші розповсюджені коди помилок при доступі до веб-ресурсів: як їх зрозуміти та виправити. ТОВ «МАКСНЕТ». URL: <https://maxnet.ua/blog/404-ta-inshi-rozpovsyudzheni-kodi-pomilok-pri-dostupi-do-veb-resursiv-yak-yih-zrozumiti-ta-vipraviti/>.

42. Колесніков Д. Що таке адаптивний сайт? Brainlab. URL: <https://brainlab.com.ua/uk/blog-uk/shho-take-adaptivnij-sajt>.

43. Клим В.Ю., Тарасенко Ю.С. Тестування веб-ресурсів на інклюзивність: аудит для реінженірингу. Вісник Вінницького політехнічного інституту, 2022. № 3. С. 60–66. DOI: 10.31649/1997-9266-2022-162-3-60-66.

44. Топ-10 посібників OWASP з безпеки веб-додатків. Hostragons Global Limited. URL:

<https://www.hostragons.com/uk/%D0%B1%D0%BB%D0%BE%D0%B3/%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D0%BA%D0%B0-%D0%B2%D0%B5%D0%B1-%D0%B4%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D1%96%D0%B2-owasp-%D1%82%D0%BE%D0%BF-10/>.

45. Стражник Б.О., Смирнов С.А. Найпопулярніші атаки на веб-додатки та методи протидії їм. Системи та технології кібернетичної безпеки : Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених. С. 307–309. URL: <https://ela.kpi.ua/server/api/core/bitstreams/68a81487-152c-4eac-9988-2a9dd32553ea/content>.

46. Найвідоміші вразливості веб застосунків. XSS та SQL ін'єкції, вразливості автентифікації. DOU.ua. URL: <https://dou.ua/forums/topic/40613/>.

47. Nho Vo. Топ-5 інструментів для тестування вебзастосунків, які варто використовувати. WebCatalog, Inc. URL: <https://webcatalog.io/uk/blog/5-web-app-testing-tools-to-consider>.

ДОДАТОК А

Вихідний код розробленого програмного забезпечення

comments.php:

```
<?php
require_once "config.php";
require_once "db.php";
if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
$taskId = $_GET["task_id"];
$taskSql = "SELECT * FROM tasks WHERE id = '$taskId'";
$taskResult = $conn->query($taskSql);
if ($taskResult->num_rows !== 1) {
    die("Task not found.");
}
$task = $taskResult->fetch_assoc();
$message = "";
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $comment = $_POST["comment"];
    $userId = $_SESSION["user_id"];
    /*
    |-----
    | BUG MODE
    |-----
    | Stored XSS
    |-----
    */
```

```

if (BUG_MODE) {
    $sql = "
        INSERT INTO comments(task_id, user_id, comment_text)
        VALUES('$taskId', '$userId', '$comment')
    ";
    $conn->query($sql);
} else {
    if (!empty($comment)) {
        $stmt = $conn->prepare(
            "INSERT INTO comments(task_id, user_id, comment_text)
            VALUES(?, ?, ?)"
        );
        $stmt->bind_param(
            "iis",
            $taskId,
            $userId,
            $comment
        );
        $stmt->execute();
    }
}

$commentsSql = "
    SELECT comments.*, users.name
    FROM comments
    LEFT JOIN users ON comments.user_id = users.id
    WHERE comments.task_id = '$taskId'
    ORDER BY comments.created_at DESC
";
$commentsResult = $conn->query($commentsSql);

```

```

?>
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <title>Comments</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
<div class="container py-5">
  <a href="dashboard.php" class="btn btn-secondary mb-4">
    Back
  </a>
  <div class="card shadow border-0 mb-4">
    <div class="card-body">
      <h3>
        <?= $task["title"] ?>
      </h3>
      <p>
        <?= $task["description"] ?>
      </p>
    </div>
  </div>
  <div class="card shadow border-0 mb-4">
    <div class="card-body">
      <h4 class="mb-3">
        Add Comment
      </h4>
      <form method="POST">

```

```

<textarea
    name="comment"
    class="form-control mb-3"
    rows="4"
></textarea>

<button class="btn btn-primary">
    Add Comment
</button>

</form>

</div>

</div>

<div class="card shadow border-0">
    <div class="card-body">
        <h4 class="mb-4">
            Comments
        </h4>
        <?php while($comment = $commentsResult->fetch_assoc()): ?>
            <div class="border rounded p-3 mb-3">
                <strong>
                    <?= $comment["name"] ?>
                </strong>
                <p class="mb-1 mt-2">
                    <?php if (BUG_MODE): ?>
                        <?= $comment["comment_text"] ?>
                    <?php else: ?>
                        <?= htmlspecialchars(
                            $comment["comment_text"],
                            ENT_QUOTES,
                            'UTF-8'
                        ) ?>

```

```

        <?php endif; ?>
    </p>
    <small class="text-muted">
        <?= $comment["created_at"] ?>
    </small>
</div>
<?php endwhile; ?>
</div>
</div>
</div>
</body>
</html>

```

config.php:

```

<?php
session_start();
/*
|-----
| BUG MODE
|-----
| true  = багована версія
| false = виправлена версія
|-----
*/
define('BUG_MODE', false);
?>

```

create_task.php:

```

<?php
require_once "config.php";

```

```

require_once "db.php";
if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
$message = "";
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $title = $_POST["title"];
    $description = $_POST["description"];
    $priority = $_POST["priority"];
    $userId = $_SESSION["user_id"];
    if (BUG_MODE) {
        /*
        |-----
        | Навмисні дефекти:
        | - Stored XSS
        | - відсутність валідації
        | - SQL Injection
        |-----
        */
        $sql = "
            INSERT INTO tasks(user_id, title, description, priority)
            VALUES('$userId', '$title', '$description', '$priority')
        ";
        if ($conn->query($sql)) {
            $message = "Task created.";
        } else {
            $message = "Error: " . $conn->error;
        }
    } else {

```

```

if(
    empty($title) ||
    empty($description)
) {
    $message = "All fields are required.";
} else {
    $stmt = $conn->prepare(
        "INSERT INTO tasks(user_id, title, description, priority)
        VALUES(?, ?, ?, ?)"
    );
    $stmt->bind_param(
        "iss",
        $userId,
        $title,
        $description,
        $priority
    );
    if ($stmt->execute()) {
        header("Location: dashboard.php");
        exit;
    } else {
        $message = "Failed to create task.";
    }
}
}
?>
<!DOCTYPE html>
<html lang="uk">
<head>

```

```

<meta charset="UTF-8">
<title>Create Task</title>
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
<div class="container py-5">
  <div class="row justify-content-center">
    <div class="col-md-7">
      <div class="card shadow border-0">
        <div class="card-body p-4">
          <h2 class="mb-4">
            Create Task
          </h2>
          <?php if(!empty($message)): ?>
            <div class="alert alert-info">
              <?= $message ?>
            </div>
          <?php endif; ?>
          <form method="POST">
            <div class="mb-3">
              <label class="form-label">
                Title
              </label>
              <input
                type="text"
                name="title"
                class="form-control"
              >
            </div>

```

```

<div class="mb-3">
  <label class="form-label">
    Description
  </label>
  <textarea
    name="description"
    class="form-control"
    rows="5"
  ></textarea>
</div>
<div class="mb-3">
  <label class="form-label">
    Priority
  </label>
  <select
    name="priority"
    class="form-select"
  >
    <option value="Low">Low</option>
    <option value="Medium">Medium</option>
    <option value="High">High</option>
  </select>
</div>
<button class="btn btn-primary">
  Create
</button>
<a href="dashboard.php" class="btn btn-secondary">
  Back
</a>
</form>

```

```

        </div>
    </div>
</div>
</div>
</div>
</body>
</html>

```

dashboard.php:

```

<?php
require_once "config.php";
require_once "db.php";
if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
$userId = $_SESSION["user_id"];
/*
|-----
| BUG MODE
|-----
| Навмисний дефект:
| - відсутність фільтрації задач по користувачу
|-----
*/
if (BUG_MODE) {
    $sql = "
        SELECT tasks.*, users.name AS author
        FROM tasks
        LEFT JOIN users ON tasks.user_id = users.id
    ";
}

```

```

ORDER BY tasks.created_at DESC

";
} else {
    $sql = "
        SELECT tasks.*, users.name AS author
        FROM tasks
        LEFT JOIN users ON tasks.user_id = users.id
        WHERE tasks.user_id = $userId
        ORDER BY tasks.created_at DESC
    ";
}
$result = $conn->query($sql);
?>
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Dashboard</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
<div class="container py-5">
    <div class="d-flex justify-content-between align-items-center mb-4">
        <div>
            <h2>
                Task Dashboard
            </h2>
            <p class="text-muted mb-0">
                Welcome,

```

```

        <?= $_SESSION["user_name"] ?>
    </p>
</div>
<div>
    <a href="create_task.php" class="btn btn-primary">
        Create Task
    </a>
    <a href="logout.php" class="btn btn-dark">
        Logout
    </a>
</div>
</div>
<div class="card shadow-sm border-0">
    <div class="card-body">
        <table class="table table-hover align-middle">
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Title</th>
                    <th>Status</th>
                    <th>Priority</th>
                    <th>Author</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                <?php while($task = $result->fetch_assoc()): ?>
                    <tr>
                        <td>
                            <?= $task["id"] ?>

```

```

</td>
<td>
    <?= $task["title"] ?>
</td>
<td>
    <?= $task["status"] ?>
</td>
<td>
    <?= $task["priority"] ?>
</td>
<td>
    <?= $task["author"] ?>
</td>
<td>
    <a
        href="edit_task.php?id=<?= $task["id"] ?>"
        class="btn btn-sm btn-warning"
    >
        Edit
    </a>
    <a
        href="delete_task.php?id=<?= $task["id"] ?>"
        class="btn btn-sm btn-danger"
    >
        Delete
    </a>
    <a
        href="comments.php?task_id=<?= $task["id"] ?>"
        class="btn btn-sm btn-info"
    >

```

Comments

```

        </a>
    </td>
</tr>
<?php endwhile; ?>
</tbody>
</table>
</div>
</div>
</div>
</body>
</html>

```

db.php:

```

<?php
$host = "localhost";
$dbname = "taskflow_db";
$username = "root";
$password = "";
$conn = new mysqli($host, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Database connection failed: " . $conn->connect_error);
}
$conn->set_charset("utf8mb4");
?>

```

delete_task.php:

```

<?php
require_once "config.php";
require_once "db.php";

```

```

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
$id = $_GET["id"];
/*
|-----
| BUG MODE
|-----
| - delete via GET
| - no ownership validation
|-----
*/
if (BUG_MODE) {
    $sql = "DELETE FROM tasks WHERE id = '$id'";
} else {
    $userId = $_SESSION["user_id"];
    $sql = "
        DELETE FROM tasks
        WHERE id = '$id'
        AND user_id = '$userId'
    ";
}
$conn->query($sql);
header("Location: dashboard.php");
exit;

edit_task.php:
<?php
require_once "config.php";

```

```

require_once "db.php";
if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
$id = $_GET["id"];
/*
|-----
| BUG MODE
|-----
| IDOR / Broken Access Control
|-----
*/
if (BUG_MODE) {
    $sql = "SELECT * FROM tasks WHERE id = '$id'";
} else {
    $userId = $_SESSION["user_id"];
    $sql = "
        SELECT *
        FROM tasks
        WHERE id = '$id'
        AND user_id = '$userId'
    ";
}
$result = $conn->query($sql);
if ($result->num_rows !== 1) {
    die("Task not found.");
}
$task = $result->fetch_assoc();
$message = "";

```

```

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $title = $_POST["title"];
    $description = $_POST["description"];
    $status = $_POST["status"];
    $priority = $_POST["priority"];
    if (BUG_MODE) {
        $updateSql = "
            UPDATE tasks
            SET
                title = '$title',
                description = '$description',
                status = '$status',
                priority = '$priority'
            WHERE id = '$id'
        ";
        $conn->query($updateSql);
    } else {
        $stmt = $conn->prepare(
            "UPDATE tasks
            SET
                title = ?,
                description = ?,
                status = ?,
                priority = ?
            WHERE id = ?"
        );
        $stmt->bind_param(
            "ssssi",
            $title,
            $description,

```



```

        Title
    </label>
    <input
        type="text"
        name="title"
        class="form-control"
        value="<?= $task["title"] ?>"
    >
</div>
<div class="mb-3">
    <label class="form-label">
        Description
    </label>
    <textarea
        name="description"
        class="form-control"
        rows="5"
    ><?= $task["description"] ?></textarea>
</div>
<div class="mb-3">
    <label class="form-label">
        Status
    </label>
    <select
        name="status"
        class="form-select"
    >
        <option value="Open">
            Open
        </option>

```

```

        <option value="In Progress">
            In Progress
        </option>
        <option value="Done">
            Done
        </option>
    </select>
</div>
<div class="mb-3">
    <label class="form-label">
        Priority
    </label>
    <select
        name="priority"
        class="form-select"
    >
        <option value="Low">
            Low
        </option>
        <option value="Medium">
            Medium
        </option>
        <option value="High">
            High
        </option>
    </select>
</div>
<button class="btn btn-warning">
    Save Changes
</button>

```



```

body{
    background-color:#f5f7fa;
}
.hero{
    padding:80px 0;
}
.card-hover:hover{
    transform:translateY(-3px);
    transition:0.2s;
}
footer{
    margin-top:80px;
    padding:20px 0;
    background:#212529;
    color:#fff;
}
.feature-icon{
    font-size:42px;
}
</style>
</head>
<body>
<!-- NAVBAR -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <div class="container">
        <a class="navbar-brand fw-bold" href="index.php">
            QA TaskFlow
        </a>
        <button
            class="navbar-toggler"

```

```

type="button"
data-bs-toggle="collapse"
data-bs-target="#navbarNav"
>
    <span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav ms-auto">
        <?php if(isset($_SESSION["user_id"])): ?>
            <li class="nav-item">
                <a class="nav-link" href="dashboard.php">
                    Dashboard
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="create_task.php">
                    Create Task
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="logout.php">
                    Logout
                </a>
            </li>
        <?php else: ?>
            <li class="nav-item">
                <a class="nav-link" href="register.php">
                    Registration
                </a>
            </li>

```

```

    <li class="nav-item">
        <a class="nav-link" href="login.php">
            Login
        </a>
    </li>
<?php endif; ?>
</ul>
</div>
</div>
</nav>
<!-- HERO -->
<section class="hero">
    <div class="container">
        <div class="row align-items-center">
            <div class="col-lg-7">
                <h1 class="display-5 fw-bold mb-4">
                    Demonstration Web Application
                    for Manual QA Testing
                </h1>
                <p class="lead text-muted">
                    QA TaskFlow is a simplified task management system
                    designed specifically for demonstrating manual testing
                    processes, bug reporting and verification of
                    web application functionality.
                </p>
                <p class="text-muted">
                    The application includes user authentication,
                    task management, comments, form validation,
                    database interaction and typical client-server
                    scenarios used during software testing.
                </p>
            </div>
        </div>
    </div>
</section>

```

</p>

<p class="text-muted">

The project supports a special BUG_MODE that intentionally enables vulnerable and unstable functionality for educational testing purposes.

</p>

<?php if (BUG_MODE): ?>

<div class="alert alert-danger">

BUG_MODE ENABLED:

The application currently contains intentionally implemented defects and security vulnerabilities.

</div>

<?php else: ?>

<div class="alert alert-success">

SAFE MODE ENABLED:

The fixed and secured version of the application is currently active.

</div>

<?php endif; ?>

</div>

<div class="col-lg-5">

<div class="card shadow border-0">

<div class="card-body p-4">

<h4 class="mb-3">

Application Modules

</h4>

<ul class="list-group">

<li class="list-group-item">

User registration and authentication


```

    <li class="list-group-item">
        Task creation and editing
    </li>
    <li class="list-group-item">
        Status and priority management
    </li>
    <li class="list-group-item">
        Comment system
    </li>
    <li class="list-group-item">
        MySQL database interaction
    </li>
    <li class="list-group-item">
        Session management
    </li>
    <li class="list-group-item">
        Validation and error handling
    </li>
    <li class="list-group-item">
        Security testing scenarios
    </li>
</ul>
</div>
</div>
</div>
</div>
</div>
</section>
<!-- FEATURES -->
<section class="pb-5">

```

```

<div class="container">
  <div class="row g-4">
    <!-- MANUAL TESTING -->
    <div class="col-md-4">
      <div class="card h-100 shadow-sm border-0 card-hover">
        <div class="card-body">
          <div class="feature-icon mb-3">
            
          </div>
          <h5 class="card-title">
            Manual Testing
          </h5>
          <p class="card-text">
            Functional testing of forms,
            workflows, UI behavior, validation,
            navigation and business logic.
          </p>
        </div>
      </div>
    </div>
    <!-- SECURITY TESTING -->
    <div class="col-md-4">
      <div class="card h-100 shadow-sm border-0 card-hover">
        <div class="card-body">
          <div class="feature-icon mb-3">
            
          </div>
          <h5 class="card-title">
            Security Testing
          </h5>

```



```
<!-- TESTING OBJECTIVES -->
```

```
<section class="pb-5">
```

```
<div class="container">
```

```
<div class="card shadow-sm border-0">
```

```
<div class="card-body p-5">
```

```
<h3 class="mb-4">
```

```
    Testing Objectives
```

```
</h3>
```

```
<div class="row">
```

```
<div class="col-md-6">
```

```
<ul>
```

```
<li>
```

```
    Verify correctness of task workflows
```

```
</li>
```

```
<li>
```

```
    Validate user input processing
```

```
</li>
```

```
<li>
```

```
    Detect UI and UX defects
```

```
</li>
```

```
<li>
```

```
    Test application stability
```

```
</li>
```

```
</ul>
```

```
</div>
```

```
<div class="col-md-6">
```

```
<ul>
```

```
<li>
```

```
    Identify security vulnerabilities
```

```
</li>
```

```

        <li>
            Perform regression testing
        </li>
        <li>
            Analyze application behavior
        </li>
        <li>
            Verify access control restrictions
        </li>
    </ul>
</div>
</div>
</div>
</div>
</div>
</section>
<footer>
    <div class="container text-center">
        QA TaskFlow © 2026 <br>
        Demonstration project for manual QA testing
    </div>
</footer>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></scr
ipt>
</body>
</html>

login.php:
<?php

```

```

require_once "config.php";
require_once "db.php";
$message = "";
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $email = $_POST["email"];
    $password = $_POST["password"];
    /*
    |-----
    | BUG MODE
    |-----
    | Навмисні дефекти:
    | - SQL Injection
    | - MD5
    | - відсутність session regeneration
    |-----
    */
    if (BUG_MODE) {
        $hashedPassword = md5($password);
        $sql = "
            SELECT *
            FROM users
            WHERE email = '$email'
            AND password = '$hashedPassword'
        ";
        $result = $conn->query($sql);
        if ($result->num_rows === 1) {
            $user = $result->fetch_assoc();
            $_SESSION["user_id"] = $user["id"];
            $_SESSION["user_name"] = $user["name"];
            header("Location: dashboard.php");
        }
    }
}

```

```

        exit;
    } else {
        $message = "Invalid credentials.";
    }
} else {
    /*
    |-----
    | FIXED VERSION
    |-----
    */
    if (
        empty($email) ||
        empty($password)
    ) {
        $message = "All fields are required.";
    } else {
        $stmt = $conn->prepare(
            "SELECT * FROM users WHERE email = ?"
        );
        $stmt->bind_param("s", $email);
        $stmt->execute();
        $result = $stmt->get_result();
        if ($result->num_rows === 1) {
            $user = $result->fetch_assoc();
            if (
                password_verify(
                    $password,
                    $user["password"]
                )
            ) {

```


Login

</h2>

<?php if (!empty(\$message)): ?>

<div class="alert alert-danger">

<?= \$message ?>

</div>

<?php endif; ?>

<form method="POST">

<div class="mb-3">

<label class="form-label">

Email

</label>

<input

type="text"

name="email"

class="form-control"

>

</div>

<div class="mb-3">

<label class="form-label">

Password

</label>

<input

type="password"

name="password"

class="form-control"

>

</div>

<button class="btn btn-dark w-100">

Login


```

$name = $_POST["name"];
$email = $_POST["email"];
$password = $_POST["password"];
/*
|-----
| BUG MODE
|-----
| Навмисні дефекти:
| - відсутня серверна валідація
| - пароль хешується через MD5
| - можливість дублювання email
|-----
*/
if (BUG_MODE) {
    $hashedPassword = md5($password);
    $sql = "INSERT INTO users(name, email, password)
        VALUES('$name', '$email', '$hashedPassword')";
    if ($conn->query($sql)) {
        $message = "Registration successful!";
    } else {
        $message = "Error: " . $conn->error;
    }
} else {
    /*
    |-----
    | FIXED VERSION
    |-----
    */
    if (
        empty($name) ||

```

```

empty($email) ||
empty($password)
) {
    $message = "All fields are required.";
} elseif (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $message = "Invalid email format.";
} elseif (strlen($password) < 6) {
    $message = "Password must contain at least 6 characters.";
} else {
    $checkStmt = $conn->prepare(
        "SELECT id FROM users WHERE email = ?"
    );
    $checkStmt->bind_param("s", $email);
    $checkStmt->execute();
    $checkStmt->store_result();
    if ($checkStmt->num_rows > 0) {
        $message = "User with this email already exists.";
    } else {
        $hashedPassword = password_hash(
            $password,
            PASSWORD_DEFAULT
        );
        $stmt = $conn->prepare(
            "INSERT INTO users(name, email, password)
            VALUES(?, ?, ?)"
        );
        $stmt->bind_param(
            "sss",
            $name,
            $email,

```

```

        $hashedPassword
    );
    if ($stmt->execute()) {
        $message = "Registration successful!";
    } else {
        $message = "Registration failed.";
    }
}
}
}
}
}
}
?>
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">
<div class="container py-5">
    <div class="row justify-content-center">
        <div class="col-md-5">
            <div class="card shadow border-0">
                <div class="card-body p-4">
                    <h2 class="mb-4 text-center">
                        Registration
                    </h2>
                    <?php if (!empty($message)): ?>

```

```
<div class="alert alert-info">
    <?= $message ?>
</div>
<?php endif; ?>
<form method="POST">
    <div class="mb-3">
        <label class="form-label">
            Name
        </label>
        <input
            type="text"
            name="name"
            class="form-control"
        >
    </div>
    <div class="mb-3">
        <label class="form-label">
            Email
        </label>
        <input
            type="text"
            name="email"
            class="form-control"
        >
    </div>
    <div class="mb-3">
        <label class="form-label">
            Password
        </label>
        <input
```


ДОДАТОК Б

Демонстраційні матеріали (презентація)



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Методи ручного тестування веб-додатків
у процесі забезпечення якості програмного забезпечення

Виконав студент 4 курсу
групи ІСД-41
Онищенко Владислав Олегович
Керівник роботи
доктор філософії Жидка Ольга Валеріївна

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи дослідження методів ручного тестування веб-додатків та формування практичних рекомендацій щодо підвищення ефективності процесу тестування

Об'єкт дослідження процес забезпечення якості веб-додатків

Предмет дослідження методи ручного тестування веб-додатків та особливості їх застосування

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. аналіз теоретичних основ тестування ПЗ;
2. дослідження особливостей ручного тестування веб-додатків;
3. аналіз методів і технік ручного тестування;
4. дослідження типових дефектів веб-систем;
5. формування алгоритму ручного тестування;
6. практична реалізація тестування демонстраційного веб-додатка;
7. формування практичних рекомендацій.

3

АКТУАЛЬНІСТЬ ТА ПРАКТИЧНА ЦІННІСТЬ

- зростання складності веб-додатків;
- необхідність забезпечення якості;
- велика кількість сценаріїв взаємодії;
- ручне тестування залишається важливим навіть за наявності автоматизації.



4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

- **Функціональні:**
 - реєстрація та авторизація користувачів;
 - робота із задачами та коментарями;
 - взаємодія з базою даних;
 - перевірка введених даних;
 - режим **BUG_MODE** для тестування помилок.
- **Нефункціональні:**
 - коректна робота у браузерях;
 - простий та зрозумілий інтерфейс;
 - стабільність роботи системи;
 - можливість використання для ручного тестування.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



6

МЕТОДИ ТА ТЕХНІКИ РУЧНОГО ТЕСТУВАННЯ



- Комбінування формалізованих і дослідницьких методів тестування дозволяє підвищити ефективність виявлення дефектів у веб-додатках.

7

УЗАГАЛЬНЕНИЙ АЛГОРИТМ РУЧНОГО ТЕСТУВАННЯ



- Запропонований алгоритм забезпечує системний підхід до організації процесу ручного тестування веб-додатків.

8

ГОЛОВНА СТОРІНКА

QA TaskFlow

RegistrationLogin

Demonstration Web Application for Manual QA Testing

QA TaskFlow is a simplified task management system designed specifically for demonstrating manual testing processes, bug reporting and verification of web application functionality.

The application includes user authentication, task management, comments, form validation, database interaction and typical client-server scenarios used during software testing.

The project supports a special **BUG_MODE** that intentionally enables vulnerable and unstable functionality for educational testing purposes.

SAFE MODE ENABLED: The fixed and secured version of the application is currently active.

Application Modules

User registration and authentication

Task creation and editing

Status and priority management

Comment system

MySQL database interaction

Session management

Validation and error handling

Security testing scenarios

Manual Testing

Functional testing of forms, workflows, UI behavior, validation, navigation and business logic.

Security Testing

Demonstration of common vulnerabilities: SQL Injection, Stored XSS, Broken Access Control, weak authentication and validation flaws.

Bug Reporting

Preparation of test cases, bug reports, regression testing and verification of fixes.

9

ПАНЕЛЬ ЗАДАЧ

Task Dashboard

Welcome, John Smith

Create TaskLogout

ID	Title	Status	Priority	Author	Actions
1	Fix login page validation	In Progress	High	John Smith	EditDeleteComments
2	Update dashboard UI	To Do	Medium	John Smith	EditDeleteComments

10

ФОРМА СТВОРЕННЯ ЗАДАЧІ

Create Task

Title

Test Task Title

Description

Test Task Description

Priority

High

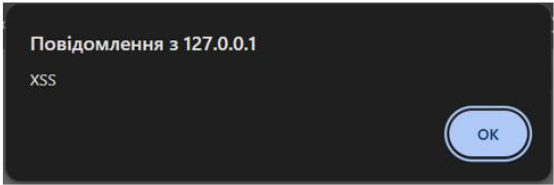
Create

Back

11

РЕЗУЛЬТАТИ ТЕСТУВАННЯ ТА ВИЯВЛЕНІ ДЕФЕКТИ

Дефект	Результат
SQL Injection	Виявлено
Stored XSS	Виявлено
Некоректна валідація	Виявлено
Проблеми із сесіями	Виявлено
Відсутність перевірки вводу	Виявлено



Registration

Registration successfull

Name

User

Email

randomemail

Password

Register

[Already have an account?](#)

Login

Email

admin@taskflow.local OR TRUE #

Password

Login

[Create account](#)

12

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ

- використання risk-based testing;
- пріоритизація тестових сценаріїв;
- комбінування exploratory testing із test-cases;
- використання структурованої тестової документації;
- адаптивний підхід до тестування;
- регулярне regression testing.

Адаптивний підхід дозволяє змінювати стратегію тестування залежно від рівня ризику функціональності та етапу розробки.

13

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ

Традиційний підхід	Запропонований підхід
○ несистемне формування тестових сценаріїв;	✓ структурований алгоритм тестування;
○ відсутність пріоритизації перевірок;	✓ використання risk-based testing;
○ висока залежність від людського фактору;	✓ комбінування exploratory testing і test-cases;
○ ризик пропуску критичних дефектів;	✓ покращення тестового покриття;
○ складність повторного тестування.	✓ підвищення ефективності regression testing;
	✓ зменшення ризику пропуску дефектів.

Використання систематизованого та адаптивного підходу до ручного тестування дозволяє підвищити якість перевірки веб-додатків і забезпечити більш ефективне виявлення дефектів.

14

ВИСНОВКИ

1. досліджено методи ручного тестування;
2. сформовано алгоритм ручного тестування веб-додатків;
3. реалізовано демонстраційний веб-додаток;
4. проведено тестування та виявлено дефекти;
5. сформовано практичні рекомендації;
6. підтверджено ефективність адаптивного підходу.

ДЯКУЮ ЗА УВАГУ!
