МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
Циклова комісія *комп'ютерних систем та мереж*
(повна назва циклової комісії)

Допустити до захисту
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
_____ *Ірина КРАВЧУК*
(підпис) (ім'я, ПРІЗВИЩЕ)
« 10 » 06 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬО-ПРОФЕСІЙНОГО СТУПЕНЯ ФАХОВИЙ МОЛОДШИЙ БАКАЛАВР

Тема: *Основи тестування програмного забезпечення (software testing)*

Група: *321* Спеціальність: *123 «Комп'ютерна інженерія»*

Здобувач освіти _____ *Кирило ТОРОС*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Олександр ГРИНЧЕНКО*
(підпис) (ім'я, ПРІЗВИЩЕ)

Консультант з оформлення пояснювальної записки _____ *Оксана ОСАДЧА*
(підпис) (ім'я, ПРІЗВИЩЕ)

Кривий Ріг 2025 р.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Презентація Microsoft PowerPoint

6. Консультанти розділів роботи (проекту)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Узгодження теми кваліфікаційної роботи*	*15.03.2025*	*Виконано*
2	*Огляд літератури*	*02.04.2025*	*Виконано*
3	*Розділ 1.Основи тестування програмного забезпечення*	*01.05.2025*	*Виконано*
4	*Розділ 2.Види та методи тестування програмного забезпечення*	*22.05.2025*	*Виконано*
5	*Розділ 3.Практичне застосування тестування та аналіз результатів*	*30.05.2025*	*Виконано*
6	*Оформлення пояснювальної записки*	*02.06.2025*	*Виконано*
7	*Попередній захист кваліфікаційної роботи*	*02.06.2025-06.06.2025*	*Виконано*
8	*Захист роботи*		

Здобувач освіти _____ *Кирило ТОРОС*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Олександр ГРИНЧЕНКО*
(підпис) (ім'я, ПРІЗВИЩЕ)

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Відділення *комп'ютерної та програмної інженерії*
Циклова комісія *комп'ютерних систем та мереж*
Освітньо-професійний ступінь *фаховий молодший бакалавр*
Спеціальність *123 «Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
(підпис) *Ірина КРАВЧУК* (ім'я, ПРІЗВИЩЕ)
« *10* » *03* 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ОСВІТИ

Торосу Кирилу Івановичу
(прізвище, ім'я, по батькові)

1.Тема роботи *Основи тестування програмного забезпечення (software testing)*

Керівник роботи *Гринченко Олександр Сергійович, викладач вищої категорії*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по коледжу від « *04* » *04* 2025 року № *50-ст*

2. Строк подання здобувачем освіти роботи з _____ по _____

3. Вихідні дані до роботи *Аналіз програмного забезпечення для тестування, Методи тестування ПЗ.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

Розділ 1.Основи тестування програмного забезпечення

Розділ 2.Види та методи тестування програмного забезпечення

Розділ 3.Практичне застосування тестування та аналіз результатів

StrikePlagiarism.com

Дата звіту 6/9/2025
Дата редагування ---

Документ прийнятий

Звіт подібності

метадані

Назва організації
Ukrainian national aviation university
Заголовок
Торос К_321_2025_КПІ.docx.docx
Автор Науковий керівник / Експерт
ТоросГринченко О
підрозділ
Криворізький Фаховий коледж

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

КП 1

КЦ

25	9647	76103
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	1
Інтервали	A→	0
Мікропробіли	¶	1
Білі знаки	ß	0
Парафрази (SmartMarks)	a	12

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://mate.academy/blog/qa/static-dynamic-testing/	19 0.20 %
2	Poiasniuvalna_zapyska_2024_Shtenzova [redacted] 11/25/2024 National Technical University "Kharkiv Polytechnic Institute" students papers (National Technical University "Kharkiv Polytechnic Institute" students papers)	12 0.12 %
3	https://mate.academy/blog/qa/static-dynamic-testing/	11 0.11 %

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Основи тестування програмного забезпечення (*software testing*)» містить: 69 сторінок, 29 рисунків, 9 таблиць, 15 використаних джерел.

AGILE, DEVOPS, IT, UI, UX, STLC, API, QA, TESTRAIL, JUNIT, SELENIUM, POSTMAN, ПЗ, JIRA, TESTLINK, TESTNG, ORANGEHRM, WHITE-BOX TESTING, BLACK-BOX TESTING, GRAY-BOX TESTING, ТЕСТ-КЕЙС, CYPRESS, PLAYWRIGHT, IDE, SWAGGER

Мета роботи: метою кваліфікаційної роботи є системне вивчення основ тестування програмного забезпечення, зокрема його принципів, методів, видів та інструментів, а також аналіз практичного застосування тестування на прикладі реального програмного продукту з метою підвищення якості ПЗ.

Актуальність роботи: У сучасному *IT*-середовищі якість програмного забезпечення є критично важливою. Помилки у програмних продуктах можуть призвести до серйозних наслідків: фінансових втрат, витоку даних, зниження довіри користувачів. Зростаюча складність ПЗ, вимоги до швидкого випуску продуктів та висока конкуренція на ринку зумовлюють необхідність ефективного тестування. Також актуальність теми зростає через інтеграцію новітніх підходів (*Agile, DevOps*) і технологій (автоматизація, штучний інтелект) у процес тестування. Вивчення основ тестування ПЗ є важливим для формування професійної компетентності майбутніх *IT*-фахівців.

Об'єкт дослідження: Процес забезпечення якості програмного забезпечення в межах життєвого циклу його розробки.

Предмет дослідження: Теоретичні основи, методи, види та інструменти тестування програмного забезпечення, а також особливості їх практичного застосування на прикладі системи *OrangeHRM*.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

API (Application Programming Interface) — інтерфейс прикладного програмування, набір визначень для взаємодії програмних компонентів.

CI/CD (Continuous Integration / Continuous Delivery або Continuous Deployment) — безперервна інтеграція та доставка/розгортання; практики автоматизації всіх етапів життєвого циклу ПЗ.

IDE (Integrated Development Environment) — інтегроване середовище розробки; програмне забезпечення, що забезпечує інструменти для написання, тестування та налагодження коду.

IT (Information Technology) — інформаційні технології; галузь, що охоплює обробку, зберігання та передачу інформації за допомогою комп'ютерних систем.

MVC (Model–View–Controller) — архітектурний шаблон, що розділяє програму на три взаємопов'язані компоненти: модель, представлення та контролер. *SDLC (Software Development Life Cycle)* — життєвий цикл розробки програмного

забезпечення; етапи створення ПЗ — від ідеї до виводу на ринок. *STLC (Software Testing Life Cycle)* — життєвий цикл тестування програмного забезпечення; послідовність етапів, що визначають процес тестування. *UAT (User Acceptance Testing)* — користувацьке приймальне тестування; фінальний етап перевірки ПЗ перед впровадженням.

UI (User Interface) — користувацький інтерфейс; засоби взаємодії користувача з програмою або системою.

BDD (Behavior Driven Development) — розробка, орієнтована на поведінку; методологія, що фокусується на описі очікуваної поведінки системи мовою, зрозумілою бізнес-користувачам.

TDD (Test Driven Development) — розробка, керована тестами; підхід, коли спочатку пишуться тести, а вже потім — код, який має ці тести пройти.

ВСТУП

У сучасному світі інформаційних технологій програмне забезпечення стало невід'ємною частиною більшості аспектів життєдіяльності — від освіти й медицини до фінансових систем, безпеки та національної інфраструктури. В умовах постійного ускладнення програмних продуктів зростає значення забезпечення їхньої надійності, стабільності та безпечності, що безпосередньо залежить від якості тестування. Наявність помилок у програмі може призвести до серйозних фінансових, репутаційних або навіть соціальних втрат. Саме тому тестування програмного забезпечення виступає критично важливою стадією життєвого циклу ПЗ.

Актуальність теми обумовлюється потребою у високоякісному програмному забезпеченні, яке відповідає очікуванням користувачів, вимогам замовника та технічним стандартам. Зважаючи на стрімкий розвиток індустрії розробки ПЗ, особливо важливою є підготовка фахівців, які не тільки володіють інструментами розробки, а й мають глибоке розуміння процесів верифікації та валідації продукту. Вивчення основ тестування дозволяє сформувати базис для професійного зростання в цій галузі, зокрема для ролей *QA*-інженерів, тестувальників, аналітиків тощо.

Предметом дослідження є методи, інструменти та практики тестування, що застосовуються для виявлення дефектів, оцінки якості та забезпечення відповідності продукту встановленим вимогам.

Метою кваліфікаційної роботи є дослідження фундаментальних понять, видів і методів тестування програмного забезпечення, а також аналіз інструментів і практик, що застосовуються в сучасному *IT*-середовищі.

Для досягнення мети було сформульовано такі завдання:

- Проаналізувати основні види тестування: ручне, автоматизоване, функціональне, нефункціональне, модульне, інтеграційне тощо.
- Ознайомитись з базовими термінами, такими як кейс, баг, тест-план, регресія.

- Провести огляд популярних інструментів, які використовуються в тестуванні: *Postman, Selenium, JUnit, TestRail* тощо.
- Дослідити практичні приклади та підходи до побудови тестової документації.
- Показати значення якості тестування для ефективного програмного продукту.

Методи дослідження, застосовані в роботі, включають аналіз літературних джерел, узагальнення практичних кейсів, вивчення документації, систематизацію матеріалів та порівняльну характеристику інструментів.

Практична значущість отриманих результатів полягає в можливості використання напрацьованих знань для формування навчальних курсів для початківців у сфері тестування, покращення практичної підготовки майбутніх тестувальників, а також для створення методичних рекомендацій у межах спеціальності.

Таким чином, тема кваліфікаційної роботи є не лише актуальною з науково практичної точки зору, а й має безпосереднє прикладне значення для підготовки конкурентоздатного *IT*-фахівця на ринку праці.

РОЗДІЛ 1

ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО

ЗАБЕЗПЕЧЕННЯ 1.1 Теоретичні основи тестування ПЗ

Тестування програмного забезпечення — це техніка контролю якості, що передбачає перевірку відповідності фактичної поведінки програми її очікуваним результатам за допомогою ретельно підібраного набору тестів. Воно включає процес виявлення помилок і дефектів, а також оцінку програмних компонентів за різними критеріями: відповідність технічним вимогам, коректність обробки вхідних даних, продуктивність, зручність використання, сумісність із іншими програмами та операційними системами, а також відповідність задачам користувача. Оскільки якість є суб'єктивним поняттям, тестування не гарантує абсолютної безпомилковості, а лише дозволяє порівняти фактичний стан програмного продукту зі специфікацією. Важливо відрізняти тестування від загального забезпечення якості, яке охоплює ширший спектр процесів розробки. Сучасний розвиток *IT* змушує компанії дедалі активніше використовувати тестування для мінімізації ризиків, пов'язаних із помилками, які можуть призвести до фінансових і репутаційних втрат. Формування теоретичної бази тестування здійснювалось за значного внеску таких фахівців, як Борис Бейзер, Гленн Майерс і Рекс Блек, які систематизували підходи і методології в цій галузі [12].

Ідея тестування виникла ще на ранніх етапах розвитку програмування, коли обсяги коду були невеликими, але вже тоді потреба в перевірці коректності роботи програм була очевидною. З роками, коли системи ставали дедалі складнішими, постало питання не лише у виявленні помилок, а й у розумінні їхніх причин, профілактиці та забезпеченні стабільності роботи ПЗ при масштабуванні, оновленнях або збільшенні навантаження [10].

Однією з важливих концепцій у теоретичних основах тестування є поділ тестування на два ключових підходи:

- Статичне тестування — аналіз артефактів (документації, коду, вимог тощо) без фактичного виконання програми. Сюди входять рецензії, огляди, аналіз коду,

перевірка специфікацій.

• Динамічне тестування — безпосередній запуск програми з метою виявлення помилок у її поведінці під час роботи.

У межах цих підходів сформувались також рівні тестування, які відображають послідовність перевірки ПЗ на різних етапах (рисунок 1.1): 1. Модульне тестування (*Unit Testing*) — перевірка окремих функцій, класів або методів. Здійснюється зазвичай розробниками.

2. Інтеграційне тестування (*Integration Testing*) — тестування взаємодії між модулями.

3. Системне тестування (*System Testing*) — повна перевірка цілого застосунку відповідно до технічних вимог.

4. Приймальне тестування (*Acceptance Testing*) — оцінка готового продукту замовником або кінцевими користувачами.

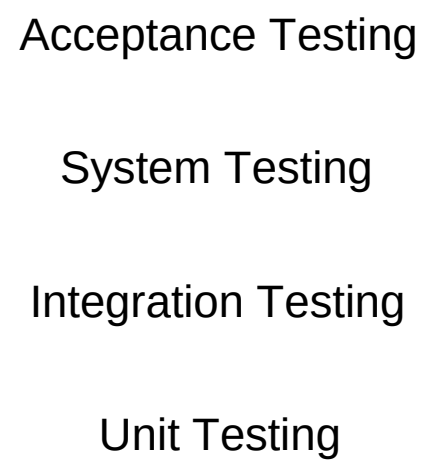


Рисунок 1.1 – Піраміда рівнів тестування: від модульного до приймального

Ще одним важливим поняттям є рівні доступу до внутрішньої логіки програми, на основі яких виділяють:

• Тестування «чорної скриньки» (*Black-box testing*) — фокус на перевірці зовнішньої поведінки системи, без знання внутрішньої реалізації. • Тестування

«білої скриньки» (*White-box testing*) — тестування внутрішньої логіки, алгоритмів, гілок коду.

• Сіре тестування (*Gray-box testing*) — проміжний підхід, коли тестувальник має обмежене уявлення про внутрішню структуру системи.

Також у теорії тестування важливою є класифікація за цілями:

• Функціональне тестування — перевірка відповідності поведінки ПЗ вимогам.

• Нефункціональне тестування — оцінка таких характеристик, як продуктивність, зручність використання, безпека.

• Регресійне тестування — перевірка, що нові зміни не зламали раніше працездатний функціонал.

• Ретестинг — повторна перевірка вже виправлених дефектів.

1.2 Визначення та основні цілі тестування ПЗ

У процесі створення програмного забезпечення виникає безліч питань: чи правильно працюють усі функції? Чи відповідає продукт очікуванням замовника? Чи здатен він витримати нестандартні сценарії використання, не втрачаючи стабільності? Для того щоб дати відповіді на ці запитання, у сфері розробки застосовується спеціальна практика — тестування програмного забезпечення. Це процес, спрямований на виявлення помилок у програмі, перевірку її працездатності, відповідності заданим вимогам і загального рівня якості створеного продукту [6].

Тестування — це не лише про пошук багів. Це ширше поняття, яке включає: • перевірку функціональності;

• оцінку зручності користування (юзабіліті);

• аналіз продуктивності;

• виявлення ризиків.

Основні цілі тестування:

1. Виявлення дефектів до того, як програму почнуть використовувати користувачі. Найбільша мета — знайти помилки раніше, ніж вони викличуть проблеми в реальному використанні. Наприклад, краще виявити, що кнопка «Оформити замовлення» не працює під час тестування, ніж коли клієнт намагається купити товар.

2. Перевірка відповідності між фактичною та очікуваною поведінкою програми. Якщо в технічному завданні сказано, що при введенні неправильного пароля користувач має отримати повідомлення «Невірний пароль», а в реальності програма просто закривається — це проблема, яку тестування має виявити.

3. Оцінка якості продукту. Якість — це не тільки відсутність багів. Це і стабільність, і зручність інтерфейсу, і сумісність з іншими системами. Наприклад, програма може працювати ідеально на *Windows 10*, але взагалі не запускатись на *Windows 11* — і це також питання до тестування.

4. Зменшення ризиків. Уявімо, що система обробки платежів має невеликий баг, який іноді подвоює суму списання. Якщо не протестувати такі речі — наслідки можуть бути катастрофічними, особливо у фінансових чи медичних системах.

5. Підвищення впевненості в продукті. Ретельно протестована програма дає впевненість розробникам, менеджерам і замовнику, що продукт дійсно готовий до використання.

Приклад із реального життя:

Уявімо розробку мобільного банкінгу. Якщо не протестувати систему логіну, то після релізу може виявитись, що користувачі не можуть увійти в акаунт при слабкому інтернеті. Тестувальник перевірив би цей кейс заздалегідь — і проблема не дійшла б до користувача.

Щоб виконати всі вищезазначені цілі, тестувальники застосовують різні типи тестування. Ось кілька з них, які найчастіше використовуються: • Функціональне тестування — перевірка, чи відповідає програма функціональним вимогам. Наприклад, чи можна зареєструватися, увійти, додати товар до кошика.

• Нефункціональне тестування — охоплює все, що не стосується конкретних

функцій: перевірку швидкодії, стабільності, безпеки. Наприклад, скільки користувачів може одночасно працювати із системою без збоїв.

• Регресійне тестування — дозволяє переконатися, що нові зміни в коді не зламали вже існуючу функціональність. Це особливо важливо при частих оновленнях продукту.

• Тестування безпеки — оцінка того, наскільки система захищена від зовнішніх загроз. Актуально для будь-яких веб-застосунків, онлайн-банкінгу тощо. • Тестування зручності (юзабіліті) — визначає, наскільки інтуїтивно зрозумілий і зручний інтерфейс програми. Іноді гарно працюючий продукт відлякує через складний або нелогічний інтерфейс.

• Сумісність — перевірка, чи працює програма на різних пристроях, *ОС,* браузерах тощо.

1.3 Принципи тестування програмного забезпечення

Коли мова заходить про тестування програмного забезпечення, важливо розуміти, що це не просто набір інтуїтивних дій, а процес, побудований на чітких принципах. Ці принципи були сформовані на основі практичного досвіду інженерів, які роками вдосконалювали підходи до перевірки якості програм.

Один із фундаментальних принципів тестування полягає в усвідомленні того, що будь-яке програмне забезпечення майже завжди містить помилки. Завдання тестувальника — не довести, що система ідеальна, а виявити ті місця, де вона поводиться неправильно або нестабільно. Навіть якщо під час тестування не виявлено жодної помилки, це зовсім не означає, що їх не існує. Можливо, перевірка просто не охопила певні граничні випадки або нетипові сценарії використання.

Неможливість повного тестування — ще одна ключова ідея. Уявімо програму з сотнями функцій і тисячами можливих комбінацій вхідних даних. Перевірити кожен варіант фізично неможливо, особливо в обмежених часових рамках. Тому

завжди доводиться приймати рішення: що перевіряти першочергово, де зосередити зусилля, які частини програми є найважливішими або найбільш вразливими. У цьому контексті особливу цінність набуває ідея раннього тестування. Чим раніше починається перевірка — ще на стадії аналізу вимог чи підготовки технічної документації — тим більше шансів виявити серйозні недоліки до того, як вони «вростуть» у систему і стануть дорогими для виправлення. Часто дешевше відкоригувати вимогу, ніж переробляти частину готового функціоналу. З практики також відомо, що помилки схильні до концентрації — це явище, яке іноді називають «ефектом Парето». Інакше кажучи, більшість проблем виникає в незначній частині коду. Це може бути пов'язано з тим, що певні модулі складніші за інші, частіше змінюються або реалізовані менш досвідченими розробниками. Тестувальники, які виявляють такі зони ризику, можуть значно підвищити ефективність перевірки, зосередившись саме на них.

Суттєвим є й той факт, що повторне виконання одних і тих самих тестів з часом втрачає сенс. Якщо перевірка не змінюється, вона виявляє все менше нових помилок. Програмне забезпечення «звикає» до цих сценаріїв, і тести починають лише підтверджувати вже відоме. Щоб цього уникнути, потрібно регулярно переглядати тест-кейси, доповнювати їх новими прикладами та орієнтуватись на зміну поведінки користувачів.

Крім того, слід пам'ятати, що тестування завжди залежить від контексту. Підхід, ефективний для мобільного застосунку, може бути непридатним для складної корпоративної системи. В одних випадках важлива швидкість, в інших — безпека, точність або масштабованість. Відповідно, техніки тестування, інструменти та пріоритети обираються з урахуванням специфіки конкретного проєкту.

І, нарешті, навіть безпомилкова з технічної точки зору програма може бути повністю непридатною для користувача. Якщо система не вирішує реальних проблем або реалізує функції, які нікому не потрібні, — це провал, незважаючи на відсутність багів. Тестування має зважати не лише на код, а й на загальну корисність, інтуїтивність і відповідність очікуванням.

Таким чином, принципи тестування — це не просто теорія. Вони формують

ментальність якісного підходу до розробки програм, дозволяють ефективно виявляти проблеми та підтримувати програмне забезпечення на високому рівні.

1.4 Ключові терміни та концепції тестування

У сфері тестування програмного забезпечення існує низка базових понять, без яких важко повноцінно орієнтуватися в процесах перевірки якості продукту. Ці терміни не просто зручні ярлики — вони відображають сутність подій, процесів і проблем, які виникають під час розробки й тестування. Знання цієї термінології дозволяє ефективніше комунікувати в команді, правильно інтерпретувати документацію, звіти та формулювати завдання.

Баг (*bug*) — це найпоширеніше слово в арсеналі будь-якого тестувальника. Його вживають для позначення будь-якої помилки або дефекту в програмі, що призводить до неправильної або неочікуваної поведінки. Наприклад, якщо натиснути кнопку «Оплатити» в інтернет-магазині, а програма зависає або перекидає на помилкову сторінку — це баг. Важливо, що баг може бути як критичним (наприклад, призводити до втрати даних), так і малозначущим (некоректне відображення тексту чи кольору).

Помилка (*error*) — це дія або недолік, допущений людиною. Наприклад, розробник написав неправильну формулу або переплутав змінні. Помилка сама по собі може не проявлятися одразу, але вона стає причиною появи дефекту [12].

Дефект (*defect/fault*) — це технічне втілення помилки в коді чи документації. Він є потенційним джерелом майбутнього бага, якщо не буде вчасно знайдений і усунений. Наприклад, якщо замість перевірки «менше або дорівнює» у коді використовується просто «менше», і це спричиняє некоректне визначення меж — це дефект [12].

Тест-кейс (*test case*) — це документ або набір кроків, який описує, що саме, як і за яких умов потрібно перевірити в програмі. У ньому вказано початкові умови, послідовність дій користувача, очікуваний результат. Наприклад, тест-кейс для

перевірки логіну користувача включає введення правильного логіна й пароля,

натискання кнопки «Увійти» і перевірку того, що відбувається перехід у профіль. Тестовий набір (*test suite*) — це група пов’язаних тест-кейсів, які об'єднані за певною ознакою (наприклад, перевірка функціональності кошика, реєстрації або замовлення). Такий підхід дозволяє системно охоплювати певні блоки програми [12].

Тестове середовище (*test environment*) — це сукупність програмного забезпечення, обладнання, налаштувань, баз даних і мережевої конфігурації, які створюють умови для проведення тестування. Ідея в тому, щоб змоделювати або відтворити реальне середовище, в якому буде працювати продукт. Наприклад, для вебзастосунку це може бути конкретна операційна система, браузер і версія серверного ПЗ.

Тестовий сценарій (*test scenario*) — це більш загальне формулювання перевірки певного функціоналу. Наприклад, «перевірити можливість зміни пароля». У межах одного сценарію може бути кілька тест-кейсів, які охоплюють різні варіанти (успішна зміна, неправильний старий пароль, занадто простий новий тощо) [6].

Регресійне тестування (*regression testing*) — це перевірка вже протестованих частин програми після внесення змін, щоб упевнитися, що новий код не порушив наявну функціональність. Це один із найважливіших типів тестування, бо часто після виправлення однієї проблеми виникають нові.

Пріоритет і серйозність (*priority & severity*) — поняття, які допомагають оцінити значущість бага. Пріоритет вказує, наскільки терміново його слід виправити, а серйозність — наскільки критичним є його вплив на систему.

Покриття тестами (*test coverage*) — це метрика, яка показує, яка частина коду або функціоналу була перевірена під час тестування. Це важливий показник ефективності тестування.

Ці терміни формують базис, на якому будується вся подальша діяльність тестувальника. Розуміння й правильне використання цієї термінології критично важливе не лише для якісного виконання тестування, але й для ефективної взаємодії

з командою розробників, аналітиків, менеджерів і замовників. Детальне пояснення

наведених термінів представлено в таблиці 1.4.

Таблиця 1.4 – Приклади термінів та їх пояснення

Термін	**Пояснення**	**Приклад**
Помилка	Людський фактор: неправильна дія, що призводить до дефекту в коді.	Програміст написав if (a = b) замість if (a == b).
Дефект	Технічне втілення помилки в коді або логіці.	У вікні входу не працює перевірка на пусте поле логіна.
Баг	Наслідок дефекту, що проявляється під час роботи програми.	При введенні порожнього логіна система зависає або видає помилку 500.
Тест-кейс	Документ, що описує умови, дії та очікуваний результат тестування.	Ввести логін і пароль → натиснути “Увійти” → очікується перехід у профіль користувача.
Тест-сценарій	Загальна перевірка функціоналу без деталізації кроків.	Перевірити процес авторизації користувача.
Тестове середовище	Умови, в яких запускається тест (система, браузер, версії тощо).	Windows 10, Google Chrome 122, тестова база даних, симулятор платіжної системи.
Регресійне тестування	Перевірка, що нові зміни не порушили стару функціональність.	Після оновлення дизайну перевірити, що кнопка “Купити” все ще працює правильно.
Покриття тестами	Відсоток коду чи функціоналу, охоплений тестами.	Із 100 функцій протестовано лише 65 — покриття 65%.

1.5 Життєвий цикл тестування (*STLC*) та його етапи

Життєвий цикл тестування програмного забезпечення (*Software Testing Life Cycle, STLC*) — це набір логічно послідовних етапів, які проходить команда тестувальників, аби забезпечити якість продукту. *STLC* допомагає структурувати

тестування, зробити його контрольованим, прогнозованим і ефективним. Кожен крок у цьому циклі має свою мету, вхідні/вихідні дані та конкретні результати [7].

Хоча в різних компаніях ці етапи можуть мати деякі відмінності, загалом *STLC* включає такі основні стадії:

1. Аналіз вимог

Це етап, коли тестувальники уважно вивчають вимоги до продукту (технічне завдання, специфікації, прототипи) і з'ясовують, що саме потрібно перевіряти. На цьому етапі також можна виявити потенційні помилки чи суперечності в документації ще до початку розробки.

Наприклад: якщо в техзавданні написано, що пароль має містити мінімум 6 символів, але в іншому місці згадується 8 — тестувальник зверне на це увагу і допоможе уникнути помилки ще до кодування

2. Планування тестування

Ціль цього етапу — сформувати чітке розуміння того, що, як і коли буде тестуватись. Тут створюється тест-план, де прописуються ресурси, інструменти, стратегія, ризики, обсяг робіт тощо.Це своєрідна дорожня карта для всієї роботи з тестування.

3. Розробка тестових випадків (тест-кейсів)

Після планування переходять до створення тестів. Тестувальники розробляють тест-кейси — детальні сценарії, які описують крок за кроком, що потрібно зробити, з якими даними, і який повинен бути очікуваний результат. 4. Підготовка тестового середовища

Цей етап полягає в тому, щоб налаштувати умови, в яких будуть запускатися тести: сервери, бази даних, браузери, мобільні емулятори, *API*-моки тощо. Це важливо, адже від середовища часто залежить поведінка програми. 5. Виконання тестування

На цьому етапі тестувальники запускають тести згідно з планом і ретельно фіксують всі результати. Якщо виявляються дефекти (баги) — їх реєструють у системі трекінгу (наприклад, *Jira*) і передають на виправлення. 6. Завершення тестування і звітність

Коли тестування завершене, команда складає фінальний звіт, у якому

описується, скільки тестів було виконано, скільки пройдено/провалено, які дефекти

були знайдені і скільки з них виправлено. Це дозволяє прийняти зважене рішення: чи можна запускати продукт у реліз, чи ще треба доопрацювати. Чому *STLC* — це не просто формальність?

Використання життєвого циклу тестування — це не бюрократія, а спосіб організувати процес так, щоб:

• не було пропущено критичних перевірок;

• зменшити ризик дефектів у продакшн-версії;

• економити час, виявляючи проблеми ще до того, як код написано. • У

реальних проектах *STLC* часто інтегрується з життєвим циклом розробки (*SDLC*), працює в спринтах, і навіть частково автоматизується. Але загальна структура лишається актуальною.

1.6 Роль тестування в життєвому циклі розробки програмного забезпечення (*SDLC*)

Тестування є критично важливою частиною життєвого циклу розробки програмного забезпечення (*SDLC*), оскільки воно забезпечує якість, надійність і відповідність продукту вимогам користувачів. Розглянемо, як тестування інтегрується на різних етапах *SDLC*:

1. Аналіз вимог

На цьому етапі тестувальники беруть участь у вивченні та уточненні вимог до програмного забезпечення. Вони перевіряють вимоги на повноту, узгодженість та однозначність, що дозволяє виявити потенційні проблеми ще до початку розробки. Це сприяє зменшенню ризику виникнення дефектів у подальших фазах проекту.

2. Проєктування

Під час проектування архітектури та дизайну системи тестувальники співпрацюють з розробниками для забезпечення тестованості продукту. Вони визначають стратегії тестування, вибирають інструменти та підходи, а також готують тестові сценарії, що відповідають специфікаціям проекту.

3. Розробка

На етапі розробки тестувальники можуть проводити статичний аналіз коду, перевіряти відповідність стандартам кодування та брати участь у рев'ю коду. Це дозволяє виявити помилки на ранніх стадіях і зменшити витрати на їх виправлення в майбутньому.

4. Тестування

Цей етап включає виконання різних видів тестування, таких як: • Модульне тестування: перевірка окремих компонентів системи на коректність роботи.

• Інтеграційне тестування: оцінка взаємодії між модулями та компонентами системи.

• Системне тестування: перевірка всієї системи на відповідність вимогам та очікуванням користувачів.

• Тестування прийняття користувачем (*UAT*): підтвердження того, що система відповідає бізнес-вимогам і готова до впровадження.

5. Впровадження

Перед запуском продукту в експлуатацію тестувальники проводять фінальні перевірки, включаючи регресійне тестування, щоб переконатися, що нові зміни не вплинули негативно на існуючий функціонал. Вони також перевіряють готовність системи до роботи в реальному середовищі.

6. Обслуговування

Після впровадження продукту тестувальники продовжують моніторинг його роботи, виявляють та фіксують нові дефекти, проводять тестування оновлень і патчів. Це забезпечує стабільність та надійність системи протягом усього її

життєвого циклу [7].

На рисунку 1.6 схематично зображено інтеграцію тестування в кожну фазу *SDLC* у вигляді кільцевої моделі. Розглянемо, як тестування вписується в ці етапи.

Рисунок 1.6 – Життєвий цикл тестування та розробки програмного забезпечення (*SDLC*)

РОЗДІЛ 2
ВИДИ ТА МЕТОДИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Класифікація видів тестування

Тестування програмного забезпечення можна класифікувати за кількома критеріями. Ці класифікації допомагають визначити, які методи та підходи застосовувати залежно від задачі, рівня доступу до коду, мети тестування та інших факторів. Класифікація зазвичай здійснюється за двома основними критеріями: за рівнем доступу до коду та за метою тестування.

2.1.1 За рівнем доступу до коду

Цей критерій класифікації базується на тому, чи має тестувальник доступ до вихідного коду програми. Залежно від цього, існує три основних типи тестування: Білі скриньки (*White-box testing*)

Білі скриньки, або структурне тестування, передбачає повний доступ до вихідного коду програми. Тестувальник може переглядати, аналізувати та розуміти всі внутрішні механізми роботи програми, що дозволяє йому створювати тестові сценарії, орієнтуючись на логіку та структуру коду. Основні підходи в цьому виді тестування:

• Кодова покритість: перевірка, наскільки повно код покритий тестами (наприклад, за допомогою покриття гілок або умов).

• Тестування шляхів виконання: оцінка всіх можливих шляхів виконання програми.

• Тестування помилок: перевірка обробки винятків і неправильних вхідних даних.

Біле тестування дозволяє виявляти помилки на найглибшому рівні, але воно потребує значних знань про внутрішню структуру програми та може бути складним для великих і складних систем [6].

Сірі скриньки (*Gray-box testing*)

Сіре скриньки поєднує елементи білого та чорного тестування. Тестувальник має частковий доступ до вихідного коду або внутрішньої архітектури системи, але не в повному обсязі. Це дає можливість комбінувати знання про систему з підходами тестування з боку користувача. Таке тестування є корисним у тих випадках, коли потрібно виявити не тільки дефекти функціональності, але й оцінити взаємодію різних компонентів системи.

Наприклад, тестувальник може знати про структуру даних, але не мати доступу до всього коду, що дозволяє виявити недоліки в інтеграції різних частин системи [6].

Чорні скриньки (*Black-box testing*)

Чорні скриньки є найпоширенішим видом тестування. В цьому випадку тестувальник не має жодного доступу до вихідного коду програми. Його завдання полягає в тому, щоб перевірити, чи працює програма згідно з вимогами та специфікаціями, на основі зовнішнього функціонування системи. Тестувальник фокусується на поведінці програми, вводячи різні дані та перевіряючи результати, не знаючи, як саме обробляється цей процес всередині.

Чорне тестування особливо ефективне для перевірки інтерфейсів, взаємодії компонентів та взаємодії з користувачем, оскільки воно дозволяє перевірити кінцеву функціональність [6].

2.1.2 За метою тестування

Інший важливий критерій класифікації тестування — це мета тестування. В залежності від мети, тестування можна поділити на функціональне та нефункціональне. Цей поділ схематично представлений на рисунку 2.1.

Функціональне тестування (*Functional Testing*)

Це тестування, яке орієнтоване на перевірку того, чи відповідає система її функціональним вимогам, визначеним на етапі планування або проектування. Тестувальник перевіряє, чи працюють всі функції системи належним чином, тобто чи виконується кожна окрема функціональна можливість згідно з вимогами.

Функціональне тестування може включати:

• Юніт-тестування (*Unit Testing*): перевірка окремих одиниць коду або модулів[15].

• Інтеграційне тестування (*Integration Testing*): перевірка взаємодії між модулями або компонентами.

• Системне тестування (*System Testing*): перевірка всієї системи в цілому. Це один із найважливіших видів тестування, оскільки він перевіряє, чи виконує програмне забезпечення свої основні функції [2].

Нефункціональне тестування (*Non-Functional Testing*)

Нефункціональне тестування фокусується не на функціях системи, а на її якісних характеристиках. Метою нефункціонального тестування є перевірка того, як система працює з точки зору таких аспектів, як швидкодія, безпека, зручність використання тощо[6]. Основні види нефункціонального тестування

• Тестування продуктивності (*Performance Testing*): перевірка того, як система працює під різними навантаженнями, включаючи стрес-тестування та тестування на витривалість.

• Тестування безпеки (*Security Testing*): оцінка того, наскільки система захищена від атак або несанкціонованого доступу.

• Тестування зручності використання (*Usability Testing*): перевірка, чи є система зручною для кінцевого користувача.

• Тестування сумісності (*Compatibility Testing*): перевірка, чи працює система на різних пристроях, операційних системах або браузерах [2].

Рисунок 2.1 – Схема класифікацій функціонального та нефункціонального тестування

2.2 Методи тестування

Методи тестування — це підходи до реалізації процесу перевірки програмного забезпечення з урахуванням цілей, ресурсів, типу ПЗ та вимог замовника. Вибір конкретного методу визначає використовувані інструменти, рівень деталізації тест-кейсів, ступінь автоматизації та підхід до аналізу ризиків .

Для узагальнення та візуалізації основних характеристик методів тестування програмного забезпечення, у таблиці 2.1 представлено їхню порівняльну характеристику.

Таблиця 2.1 – Порівняльна характеристика методів тестування

Метод тестування	**Опис**	**Переваги**	**Недоліки**	**Коли використовуєтьс я**

Ручне тестування	Тестування, яке виконується тестувальником без використання автоматичних інструментів	Гнучкість, можливість швидко виявити неочевидні помилки, добре підходить для *UI*-тестів	Часозатратне, людський фактор, низька повторюваність	На початкових етапах, при нестабільних вимогах, для креативного тестування
Автоматизоване тестування	Тестування, яке виконується з використанням спеціальних інструментів і скриптів	Швидке виконання, можливість багаторазового повторення, висока точність	Початкові витрати на налаштування, не завжди ефективне при *UI*-тестуванні	Регресійне тестування, часті повторювані перевірки, *CI/CD* процеси
Тестування на основі ризиків	Фокус на перевірці найкритичніших і найбільш ризикованих частин системи	Оптимізація ресурсів, зосередженість на важливому, ефективне при обмеженнях	Може бути пропущено менш критичні помилки, залежить від правильності оцінки ризиків	Проекти з обмеженими строками або ресурсами, складні системи з критичними модулями
Дослідницьке тестування	Інтуїтивне та креативне тестування без попередньо визначених сценаріїв	Виявляє нестандартні помилки, висока гнучкість, підходить для нових продуктів	Важко відтворити результати, вимагає досвідченого тестувальника	Коли вимоги не повністю визначені, для швидкої оцінки якості

2.2.1 Ручне тестування

Ручне тестування (*manual testing*) — це процес, під час якого тестувальник самостійно виконує тест-кейси без використання автоматизованих інструментів [6]. Цей метод передбачає глибоке розуміння логіки продукту, уважність і здатність виявляти як очевидні, так і приховані помилки.

Основні переваги:

• Ідеально підходить для коротких проєктів або коли необхідна гнучкість;

• Можливість швидкої адаптації до змін інтерфейсу або логіки програми;

• Ефективне для юзабіліті-тестування та дослідницького підходу.

Недоліки:

• Трудомісткий процес;

• Високий ризик людської помилки;

• Обмежена масштабованість при великій кількості тест-кейсів.

2.2.2 Автоматизоване тестування

Автоматизоване тестування (*automated testing*) — це метод, при якому тести створюються у вигляді сценаріїв або програмного коду і виконуються автоматично з використанням спеціалізованих інструментів (наприклад, *Selenium, JUnit, TestNG, Cypress*) [6].

Переваги:

• Швидке виконання великої кількості тестів;

• Повторне використання тестових сценаріїв;

• Ефективне в регресійному тестуванні, коли однакові дії перевіряються багаторазово;

• Зменшує ймовірність помилки за рахунок стандартизованого виконання.

Обмеження:

• Висока вартість початкової реалізації;

• Потреба у технічній кваліфікації;

• Не придатне для всіх типів тестування (наприклад, юзабіліті або креативних перевірок).

2.2.3 Тестування на основі ризиків

Тестування на основі ризиків (*risk-based testing*) — це стратегія, при якій тестування спрямовується на ті частини системи, які мають найвищу ймовірність виникнення критичних помилок або можуть спричинити найбільші втрати.
Принципи:

• Визначення бізнес- або технічних ризиків;

• Пріоритизація функціоналу за критичністю;

• Виділення ресурсів передусім на високоризиковані області.

Цей метод особливо корисний, коли обмежено час чи ресурси, і потрібно сконцентруватися на найважливішому. Наприклад, у банківському застосунку більше уваги приділяється обробці транзакцій, ніж налаштуванням профілю.

2.2.4 Дослідницьке тестування

Дослідницьке тестування (*exploratory testing*) — це підхід, при якому тестувальник одночасно вивчає програму, проектує тести та виконує їх, не маючи заздалегідь створених тест-кейсів. Метод базується на знаннях, досвіді та інтуїції спеціаліста.[1]

Ключові характеристики:

• Мінімальне формальне планування;

• Швидке виявлення помилок завдяки гнучкості;

• Часто використовується в умовах, коли документація відсутня або неповна.

Особливо ефективне у випадках, коли продукт ще не стабільний або має часті зміни. Наприклад, у стартап-проєктах, де немає повного технічного завдання, цей метод дозволяє знаходити критичні баги швидше, ніж традиційне тестування.

РОЗДІЛ 3
ПРАКТИЧНЕ ЗАСТОСУВАННЯ ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Огляд інструментів для тестування програмного забезпечення

Розвиток програмного забезпечення супроводжується зростанням складності інформаційних систем, що, своєю чергою, вимагає ефективних і точних методів перевірки їхньої якості. У цьому контексті важливу роль відіграють інструменти для тестування — спеціалізовані програмні засоби, що дозволяють автоматизувати, оптимізувати та стандартизувати процес виявлення дефектів у ПЗ.

Інструменти тестування поділяються за напрямами використання: автоматизоване функціональне тестування, юніт-тестування, тестування продуктивності, тестування інтерфейсів прикладного програмування (*API*), тестування безпеки, а також засоби для керування тестовими артефактами та звітності. На рисунку 3.1 узагальнено цю класифікацію, що візуалізує основні типи інструментів і сфери їхнього застосування.

Залежно від типу тестування, використовуються як універсальні інструменти, що підтримують кілька підходів і технологій, так і вузькоспеціалізовані рішення, орієнтовані на конкретні типи продуктів, мови програмування або тестові середовища. Наприклад, існують окремі засоби для вебдодатків, мобільних платформ або десктопного ПЗ. Деякі інструменти інтегруються з системами контролю версій, *CI/CD*-пайплайнами, таск-трекерами та іншими компонентами сучасного *DevOps*-середовища.

Також важливим є поділ інструментів за способом використання — одні працюють у вигляді локальних програм або бібліотек, інші — як вебінтерфейси або хмарні платформи. Поширення гнучких методологій розробки (*Agile, DevOps*)

стимулює інтеграцію засобів тестування безпосередньо в процес безперервної доставки, де тести виконуються автоматично на кожному етапі збірки або релізу продукту.

Крім технічних характеристик, важливу роль відіграють також аспекти зручності, продуктивності, підтримки спільноти та ліцензування. У багатьох випадках команди поєднують кілька інструментів, створюючи цілісну інфраструктуру тестування відповідно до потреб конкретного проєкту.

Таким чином, ефективне використання інструментів для тестування є ключовою умовою забезпечення високої якості програмного забезпечення, особливо в умовах високої динаміки розробки та зростаючих вимог до стабільності й продуктивності програмних продуктів.

Рисунок 3.1 – Класифікація інструментів тестування за типами та застосуванням

3.1.1 Інструменти для автоматизованого функціонального тестування

Автоматизоване функціональне тестування є одним з найбільш популярних та важливих напрямків у сфері забезпечення якості програмного забезпечення.

Воно дозволяє тестувати функціональність програмного продукту з високою швидкістю та ефективністю, забезпечуючи при цьому повторюваність тестів та зниження людського фактора. Для цього існує ряд інструментів, що надають можливість автоматизації функціональних тестів, таких як *Selenium, Cypress* та *Playwright.*

Selenium є одним з найпоширеніших і найбільш використовуваних інструментів для автоматизації тестування веб-додатків. Він підтримує різні мови програмування, такі як *Java, Python, C#, Ruby*, і може інтегруватися з популярними фреймворками для тестування, такими як *JUnit ma TestNG*. Selenium дозволяє створювати автоматизовані скрипти для тестування веб-сторінок на різних браузерах, забезпечуючи високий рівень точності та ефективності тестування. Однією з особливостей *Selenium* є його здатність працювати з реальними браузерами, що дозволяє перевіряти не лише функціональність, а й сумісність з різними версіями браузерів.

Cypress є ще одним потужним інструментом для автоматизованого функціонального тестування веб-додатків, але він має деякі суттєві відмінності від *Selenium*. Cypress спеціалізується на тестуванні сучасних *JavaScript*-додатків і має вбудовану підтримку асинхронного тестування, що дозволяє писати тести швидше. Цей інструмент працює безпосередньо в браузері, що дозволяє здійснювати тестування на рівні реального користувача і значно прискорює процес тестування. *Cypress* підтримує інтеграцію з іншими інструментами тестування, що дає можливість будувати повноцінні стратегії тестування для веб-додатків.

Playwright, подібно до *Cypress*, є новітнім інструментом для автоматизації тестування веб-додатків, який дозволяє працювати з різними браузерами, включаючи *Chromium, WebKit* та *Firefox*. Одна з основних переваг *Playwright* — це його здатність підтримувати кілька браузерів одночасно, що робить його надзвичайно корисним для тестування багатоплатформних додатків. *Playwright* також підтримує автоматизацію сценаріїв, що включають взаємодію з елементами

на сторінці, створення складних тестових випадків і навіть тестування мобільних додатків через емуляцію мобільних браузерів.

Ці інструменти дозволяють автоматизувати не тільки базові функціональні тести, але й більш складні сценарії, що включають перевірку взаємодії між різними компонентами веб-додатка, а також перевірку його роботи в умовах реальних навантажень. Вибір інструменту залежить від конкретних вимог проекту, рівня підтримки різних браузерів, мови програмування, яка використовується в проекті, а також від технічних вимог щодо швидкості та ефективності тестування.

Проте автоматизація тестування — це не лише вибір інструменту, але й створення відповідної інфраструктури для тестування. Важливо зазначити, що автоматизоване тестування потребує належної підготовки тестових скриптів, що займає певний час і ресурси, однак, у довгостроковій перспективі цей процес дозволяє значно зменшити витрати часу та коштів на тестування, особливо при наявності постійних змін в коді програмного забезпечення.

Автоматизоване функціональне тестування є важливим інструментом у забезпеченні якості програмного забезпечення, і вибір відповідного інструменту є критично важливим для ефективності тестування. Коротка підбрірка інструментів ,описаних раніше, є в таблиці 3.1.

Таблиця 3.1 – Інструменти для автоматизованого функціонального тестування

Інструмент	Мова програмування	Особливості	Підтримка браузерів
Selenium	*Java, Python, C#, Ruby*	Інтеграція з *JUnit, TestNG*. Підтримка реальних браузерів	*Chrome, Firefox, Safari, Edge* тощо
Cypress	*JavaScript*	Асинхронне тестування, запуск у браузері	Працює безпосередньо у *Chromium*-браузерах

Playwright	*JavaScript, Python, C#*	Підтримка кількох браузерів одночасно, емуляція мобільних	*Chromium, Firefox, WebKit*

3.1.2 Інструменти для юніт-тестування (залежно від мови програмування)

Юніт-тестування є важливою частиною процесу забезпечення якості програмного забезпечення. Воно спрямоване на тестування окремих частин програмного коду (зазвичай функцій або методів) для виявлення помилок на ранніх етапах розробки. Інструменти для юніт-тестування допомагають автоматизувати процес перевірки правильності роботи програмного коду, що зменшує ймовірність виникнення дефектів у готовому продукті.

Одним з найбільш поширених інструментів для юніт-тестування є *JUnit*, який використовують програмісти на *Java*. Цей фреймворк дозволяє розробникам писати і виконувати тести для окремих частин програмного коду, автоматично перевіряючи їх на коректність. *JUnit* інтегрується з багатьма середовищами розробки (*IDE*), що дозволяє легко тестувати код під час розробки та виявляти помилки на ранніх етапах [5].

Для *C#* найпопулярнішим інструментом є *NUnit*, який також підтримує тестування окремих одиниць коду. *NUnit* дає можливість створювати та виконувати тести для різних типів методів, а також має підтримку різних фреймворків для автоматизованого тестування. За допомогою *NUnit* розробники можуть забезпечити високу якість програмного забезпечення, виявляючи помилки до того, як код потрапить у продакшн середовище.

Python пропонує своє рішення для юніт-тестування у вигляді *unittest*. Цей інструмент інтегрується з іншими бібліотеками *Python*, що дає змогу автоматизувати процес тестування програм. Він підтримує різноманітні стратегії тестування, зокрема можливість тестування як окремих функцій, так і цілих класів. У *Python* також широко використовується бібліотека *pytest*, яка пропонує

зручний синтаксис і покращену інтеграцію з іншими інструментами розробки [4].

Для програмування на *JavaScript* в основному використовуються інструменти *Mocha* та *Jest*. *Mocha* — це фреймворк для тестування, який дозволяє створювати різноманітні тестові сценарії, а *Jest*, розроблений *Facebook*, забезпечує

швидке і просте юніт-тестування з фокусом на можливість тестувати як клієнтські, так і серверні частини додатків [11].

Вибір інструменту для юніт-тестування залежить від мови програмування та потреб проекту. Всі зазначені інструменти сприяють покращенню якості програмного забезпечення, скорочуючи час на виявлення помилок і дозволяючи розробникам швидше виправляти дефекти, що забезпечує стабільність і надійність готового продукту.

Застосування таких інструментів у реальних проєктах дозволяє забезпечити контроль за якістю на кожному етапі розробки, що є важливою складовою процесу розробки програмного забезпечення, особливо в умовах швидких змін і постійних оновлень. Таблиця 3.2 містить підбрірку інструментів та їх опис.

Таблиця 3.2 – Інструменти для юніт-тестування

Інструмент	Мова програмування	Особливості	Інтеграції / Додаткові можливості
JUnit	*Java*	Автоматичне тестування методів	Інтеграція з *IDE*
NUnit	*C#*	Підтримка тестування методів різного типу	Підтримка автоматизації
unittest	*Python*	Інтеграція з іншими бібліотеками	Тестування функцій і класів
pytest	*Python*	Зручний синтаксис	Інтеграція з *CI/CD*

Mocha	*JavaScript*	Гнучкість у створенні сценаріїв	Можна використовувати з *Chai*
Jest	*JavaScript*	Фокус на швидке тестування	Тестування клієнтських і серверних частин

3.1.3 Інструменти для тестування *API*

У процесі розробки сучасного програмного забезпечення все більшу роль відіграють прикладні програмні інтерфейси (*API*), які забезпечують взаємодію між окремими компонентами систем або між зовнішніми сервісами. Тестування *API* є критично важливим етапом, оскільки помилки в логіці взаємодії, обробці запитів або валідації даних можуть призвести до порушень функціональності всієї програми або її окремих модулів[8]. Для забезпечення ефективного тестування *API* використовуються спеціалізовані інструменти, серед яких найбільш поширеними є *Postman* та *Swagger*.

Postman — це один із найпопулярніших інструментів для тестування *RESTful API*. Його інтерфейс дозволяє швидко формувати *HTTP*-запити різних типів (*GET, POST, PUT, DELETE* тощо), налаштовувати заголовки, тіло запиту, авторизацію, а також переглядати відповіді від сервера у структурованому вигляді. Окрім базового функціоналу, *Postman* дозволяє автоматизувати тестування шляхом створення колекцій запитів і написання скриптів перевірки (*assertions)* за допомогою *JavaScript*. Наприклад, можна задати умову, що код відповіді має бути 200, а тіло відповіді має містити певне поле. Це спрощує процес регресійного тестування *API* та забезпечує повторне використання сценаріїв. На практиці Postman часто використовується як інструмент не тільки для тестувальників, а й для розробників, оскільки дозволяє швидко перевірити працездатність створених кінцевих точок *API* [9].

Swagger, у свою чергу, — це інструмент, який орієнтований насамперед на опис, документування та тестування *REST API*. Основою є використання

специфікації *OpenAPI*, яка описує структуру та поведінку *API* у форматі *JSON* або *YAML*. Це дозволяє як створювати документацію, зрозумілу для розробників, так і генерувати клієнтський або серверний код автоматично. Одним із компонентів *Swagger* є *Swagger UI* — веб-інтерфейс, що дозволяє взаємодіяти з *API* безпосередньо з браузера: надсилати запити, переглядати відповіді, виявляти помилки. Наприклад, у складних системах мікросервісної архітектури *Swagger*

дозволяє централізовано документувати всі сервіси, що істотно полегшує тестування і підтримку [13].

Окрім згаданих інструментів, у сфері тестування *API* також застосовуються такі рішення, як *SoapUI* для тестування *SOAP* і *REST API* з підтримкою сценаріїв на *Groovy*, або *Insomnia*, що пропонує гнучке управління змінними середовища та інтеграцію з *CI*/*CD*-процесами[3]. Проте саме *Postman* і *Swagger* залишаються стандартом де-факто завдяки своїй простоті, зручності та широким можливостям інтеграції у процес розробки.

Реальні кейси використання цих інструментів можна спостерігати у компаніях, що розробляють веб-додатки або мобільні сервіси, де *API* слугує посередником між клієнтським інтерфейсом та бекендом. Наприклад, у системах онлайн-банкінгу або електронної комерції критично важливо, щоб *API* коректно обробляв запити на перевірку балансу, оплату або авторизацію користувача. Тестування таких сценаріїв у *Postman* дозволяє швидко виявляти помилки, що можуть мати серйозні наслідки для користувачів і бізнесу.

Отже, інструменти для тестування *API* є невіддільною частиною процесу забезпечення якості програмного забезпечення, особливо в умовах активного використання сервіс-орієнтованих архітектур. Їх застосування дозволяє не лише ефективно перевіряти функціональність *API*, а й забезпечує прозорість, повторюваність і автоматизацію тестування на всіх етапах життєвого циклу розробки ПЗ. Назва та означення описаних інструментів для тестування *API* представлено у таблиці 3.3.

Таблиця 3.3 – Інструменти для тестування *API*

Назва інструменту	Призначення	Ключові можливості	Тип ліцензії	Особливості
Postman	Тестування *RESTful API*	Швидке створення *HTTP*-запитів, автоматизація тестування, підтримка сценаріїв перевірки	Безкоштовний	Інтерфейс для тестувальників і розробників, підтримка колекцій запитів

Продовження таблиці 3.3

Swagger	Опис і тестування *REST API*	Документування *API*, генерація клієнтського/серверного коду, інтеграція з *OpenAPI*	Безкоштовний	*Swagger UI* для тестування безпосередньо в браузері
SoapUI	Тестування *SOAP* і *REST API*	Підтримка сценаріїв на *Groovy*, тестування веб-сервісів	Безкоштовний/ Комерційний	Підтримка *SOAP* і *REST API*, інтеграція з *CI*/*CD*
Insomnia	Тестування *API*	Гнучке управління змінними середовища, інтеграція з *CI*/*CD*	Безкоштовний/ Комерційний	Зручний інтерфейс, підтримка різних типів запитів *API*

3.1.4 Інструменти для тестування продуктивності

Тестування продуктивності є критично важливою складовою забезпечення якості програмного забезпечення, особливо в умовах зростаючих вимог до швидкості, масштабованості та надійності сучасних *IT*-систем. Метою такого тестування є визначення, як система поводиться під навантаженням, наскільки вона стабільна при інтенсивному використанні, і чи відповідає заданим вимогам продуктивності. Для цього використовуються спеціалізовані інструменти, які дозволяють моделювати навантаження, аналізувати час відгуку, виявляти вузькі місця в архітектурі додатку та прогнозувати його поведінку в умовах пікової

активності

Одним із найпопулярніших інструментів для тестування продуктивності є *Apache JMeter*. Це безкоштовний, відкритий інструмент, який дозволяє створювати складні сценарії навантаження для вебдодатків, *API*, баз даних та інших сервісів. Його інтерфейс дає змогу задавати кількість віртуальних користувачів, їхню поведінку, параметри запитів, а також збирати метрики — зокрема, середній час відповіді, відсоток помилок, пропускну здатність тощо[14]. Наприклад, у разі розгортання нового інтернет-магазину *JMeter* можна використати для симуляції тисяч одночасних користувачів, які додають товари до кошика, оформлюють замовлення та перевіряють статус доставки.

Іншим широко застосовуваним інструментом є *LoadRunner*, який належить компанії *Micro Focus*. На відміну від *JMeter, LoadRunner* є комерційним продуктом і надає розширені можливості — зокрема, підтримку великої кількості протоколів (*HTTP*/S, *SOAP, REST, Citrix, SAP, Oracle* тощо), гнучкі засоби моніторингу, детальну аналітику та інтеграцію з іншими інструментами забезпечення якості. *LoadRunner* часто використовується у великих корпоративних проєктах, де важлива точність результатів, збереження історії навантажень і відповідність вимогам безпеки та відповідності стандартам. Наприклад, у банківському секторі цей інструмент дозволяє перевірити, як система онлайн-платежів поводиться під час пікового навантаження — наприклад, наприкінці місяця, коли значна кількість клієнтів здійснює операції одночасно.

Слід зазначити, що обидва інструменти можуть працювати в різних режимах — стрес-тестування (перевірка меж витривалості системи), навантажувального тестування (моделювання реального сценарію використання) та тестування стабільності (довготривала робота під навантаженням). Таким чином, вони дають змогу не лише знайти «вузькі місця», але й оцінити витривалість усієї інфраструктури, зокрема серверів, баз даних, балансувальників навантаження та мережевих компонентів.

У сучасному *IT* середовищі також набирають популярності хмарні рішення для тестування продуктивності, такі як *BlazeMeter* (який базується на *JMeter*), *Gatling* або навіть інтеграції з *CI*/*CD* пайплайнами, які дозволяють запускати

продуктивні тести після кожної зміни в коді. Це робить тестування продуктивності частиною автоматизованого контролю якості, що особливо важливо в умовах *Agile* розробки та безперервної інтеграції. Підбірка інструментів з їх описом представлена у таблиці 3.4.

Таблиця 3.4 – Інструменти для тестування продуктивності

Назва інструменту	Призначення	Ключові можливості	Тип ліцензії	Особливості
Apache JMeter	Тестування продуктивності веб-додатків та *API*	Симуляція навантаження, моніторинг часу відповіді, збір метрик	Безкоштовний	Підтримка великої кількості протоколів, відкритий код
LoadRunner	Тестування продуктивності в корпоративних середовищах	Підтримка різних протоколів, детальна аналітика, інтеграція з іншими інструментами	Комерційний	Висока точність результатів, використовується для великих проєктів
BlazeMeter	Хмарне тестування продуктивності на основі *JMeter*	Інтеграція з *CI/CD*, симуляція навантаження в хмарі	Комерційний	Гнучке масштабування тестів, підходить для *Agile* розробки
Gatling	Тестування продуктивності веб-додатків	Підтримка сценаріїв, аналіз результатів	Безкоштовний/Комерційний	Чистий синтаксис, інтеграція з *CI/CD*

3.1.5 Інструменти для управління тестуванням та дефектами

Управління тестуванням і дефектами є невід’ємною складовою процесу

забезпечення якості програмного забезпечення. На практиці тестування охоплює не лише перевірку коду, а й організацію тестових активностей, відслідковування прогресу, фіксацію результатів і ведення обліку помилок. Для цього використовуються спеціалізовані інструменти, які забезпечують ефективну взаємодію між командами, прозорість процесів і аналітичну підтримку прийняття рішень.

Одним із найпоширеніших інструментів у сфері управління тестуванням є *TestRail*. Цей сервіс дозволяє структурувати тестові сценарії, створювати тестові

набори, відстежувати результати виконання тестів та аналізувати покриття функціональності. *TestRail* зручний тим, що забезпечує централізовану систему зберігання даних, гнучку організацію тестових кампаній та інтеграцію з іншими інструментами (наприклад, *Jira, Jenkins*), що особливо цінно в умовах *CI/CD* процесів.

Для фіксації, класифікації та відстеження дефектів ключовим інструментом є *Jira*. Хоча *Jira* є багатофункціональним трекером задач, саме її гнучкість у налаштуванні робочих процесів (*workflow*) і підтримка *Agile*-методологій зробили її стандартом де-факто для команд розробників і тестувальників. Кожен дефект у Jira можна детально описати, призначити відповідального, задати пріоритетність, а також відслідковувати його життєвий цикл — від створення до закриття. Це дозволяє підтримувати високу прозорість та контроль над якістю продукту на всіх етапах розробки.

Крім цього, існують комплексні системи, що поєднують можливості як для управління тестами, так і для роботи з дефектами. Прикладом може бути *TestLink* — система з відкритим кодом, яка дозволяє створювати тест-плани, проводити мануальне тестування та реєструвати результати. Хоча вона менш зручна в порівнянні з комерційними аналогами, її перевага — у гнучкості, простоті розгортання та відсутності ліцензійних витрат, що важливо для невеликих команд або освітніх цілей.

Для команд, які працюють в умовах високої автоматизації, важливими є інтеграції інструментів управління тестами з системами *CI/CD*. Наприклад, інтеграція *Jira* з *Jenkins* або *GitLab* дозволяє автоматично оновлювати статуси

задач після проходження тестів. Це знижує навантаження на тестувальників і мінімізує ризик помилок у документації процесу.

Реальний приклад — у великому аутсорсинговому проєкті з розробки мобільного застосунку для банку, команда тестувальників використовувала зв'язку *Jira* + *TestRail*. Всі баги фіксувалися в *Jira*, а всі тести зберігалися й виконувалися через *TestRail*. Завдяки інтеграції між цими інструментами кожен невдалий тест

кейс автоматично створював дефект у *Jira*, що значно підвищувало швидкість

реакції та зменшувало ручну роботу. Підбірка інструментів для управління тестування та дефектами представлена в таблиці 3.5.

Таблиця 3.5 – Інструменти для управління тестування та дефектами

Інструмент	Основна функціональність	Особливості	Інтеграції	Приклад використання
TestRail	Управління тестуванням, відстеження результатів, аналіз покриття	Централізоване зберігання даних, гнучка організація тестових кампаній	*Jira, Jenkins*	Управління тестами та інтеграція з іншими інструментами для *CI/CD*
Jira	Управління задачами та дефектами, відстеження життєвого циклу дефектів	Підтримка *Agile* методологій, гнучкість у налаштуванні workflow	*TestRail, Jenkins, GitLab*	Фіксація та відстеження дефектів у команді розробників та тестувальників
TestLink	Створення тест планів, мануальне тестування	Відкритий код, простота розгортання	Не потребує спеціалізованих інструментів для інтеграції	Використовується у малих командах або в освітніх цілях

3.2 Практичне застосування тестування на прикладі *ORANGEHRM*

Для закріплення теоретичних знань та демонстрації практичної значущості тестування було обрано відкритий програмний продукт *OrangeHRM*. Це модульна система для управління людськими ресурсами, яка широко використовується як у малих, так і в середніх компаніях. Завдяки відкритому коду, доступній документації та активному ком'юніті, *OrangeHRM* є зручною платформою для вивчення різних підходів до тестування.

Особливість цього продукту полягає в його багатофункціональності: система включає модулі для управління персоналом, відпустками, табелями обліку робочого часу, рекрутингу та звітності. Це дозволяє охопити різні аспекти тестування — від функціонального та інтеграційного до нефункціонального

тестування продуктивності й безпеки. Такий вибір забезпечує всебічну перевірку навичок та застосування різних стратегій тестування в умовах, наближених до реальних бізнес-вимог.

Завдяки архітектурі, побудованій за принципами *MVC* (*Model–View–Controller*), *OrangeHRM* також надає можливість аналізу впливу змін на різні частини системи, що є важливим у процесі розробки стратегії тестування та забезпечення її цілісності.

3.2.1 Опис програмного продукту та його ключових функціональностей

OrangeHRM — це модульний веб-застосунок для управління людськими ресурсами, розроблений на *PHP* із базою даних *MySQL*. Інтерфейс побудовано на *HTML, CSS* і *JavaScript,* а архітектура — за патерном *MVC*. Система підтримує рольову модель доступу (адміністратор, менеджер, співробітник), що забезпечує розділення прав та безпечну роботу з різними модулями. На рисунку 3.2 представлено інтерфейс головної сторінки.

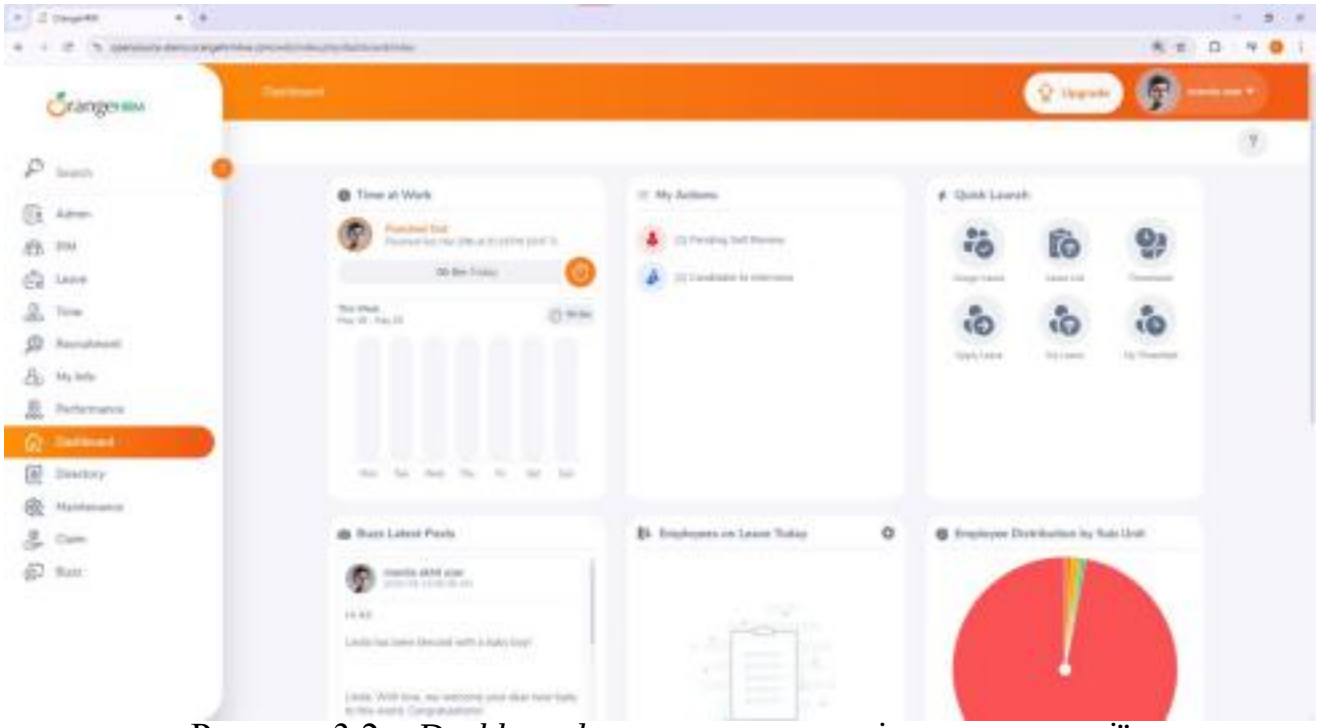


Рисунок 3.2 – *Dashboard*, головна панель після авторизації

Модуль персоналу (*PIM*) відповідає за збереження й обробку даних співробітників. Форма додавання нового працівника містить обов’язкові поля для персональних і посадових даних, що дозволяє проводити тестування валідації форм і коректності форматів. На рисунку 3.3 приклад створення форми працівника.

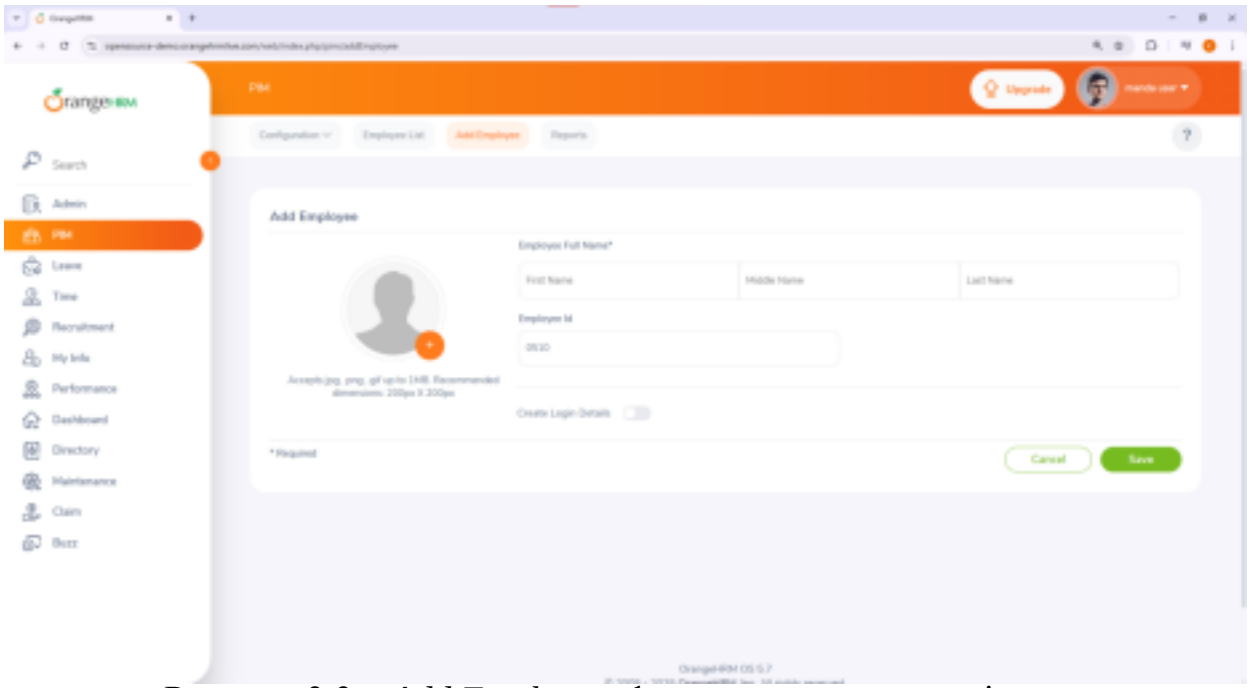


Рисунок 3.3 – *Add Employee*, форма створення працівника

Після створення записів інформація відображається в табличному вигляді зі можливістю фільтрації, пошуку та сортування за різними параметрами (ім’я, відділ, посада тощо). Це зручно для перевірки функціональності сортування та пошукових алгоритмів. Таблицю зі списком працівників зображенно на рисунку 3.4

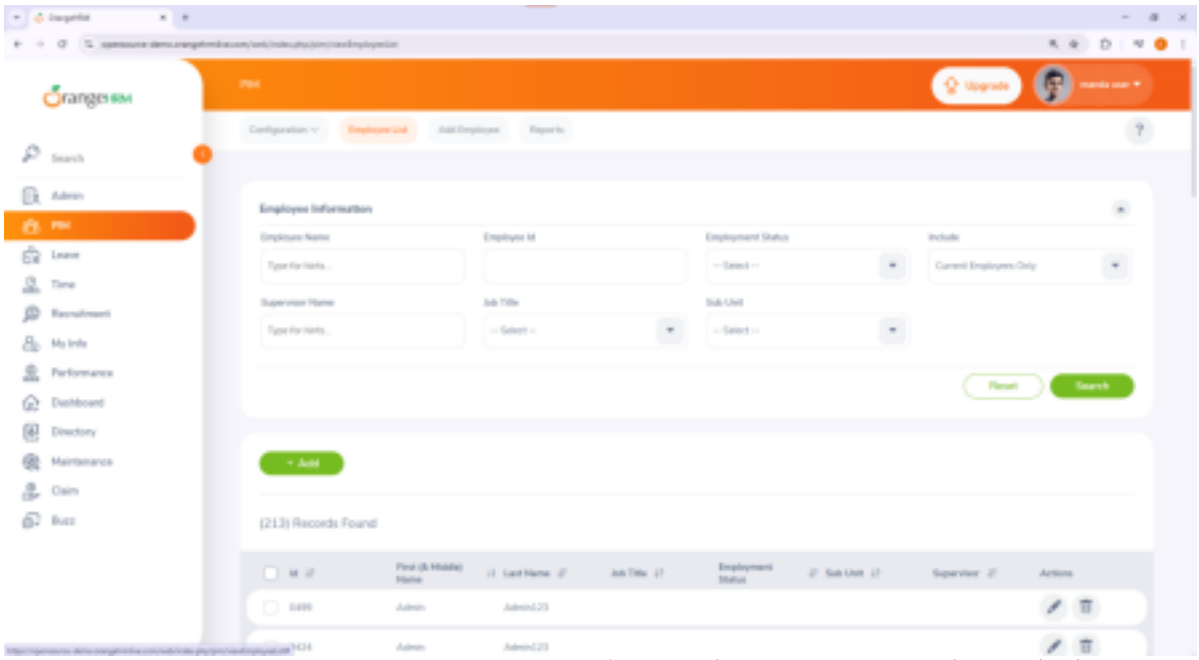


Рисунок 3.4 – *Employee List*, таблиця зі списком працівників із фільтрацією/пошуком

Модуль відпусток (*Leave*) дозволяє співробітникам створювати заявки, а менеджерам — їх погоджувати або відхиляти. У цій частині важливо тестувати бізнес-правила обчислення залишку днів відпустки та обмеження ролей. Приклад показаний на рисунку 3.5

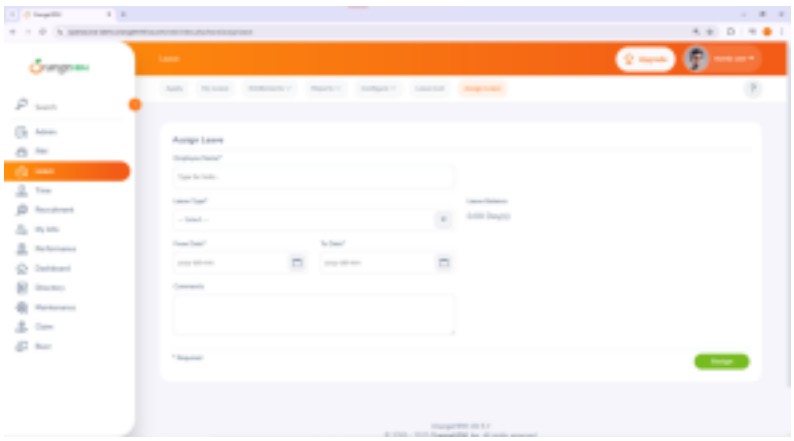


Рисунок 3.5 – *Leave Request Form* форма подачі заявки на відпустку

Для набору персоналу використовується модуль рекрутингу (*Recruitment*), що включає публікацію вакансій, обробку заявок кандидатів і планування інтерв'ю.

Складні залежні поля в цій формі вимагають особливої уваги під час функціонального тестування. Форма створення вакансії на рисунку 3.6.

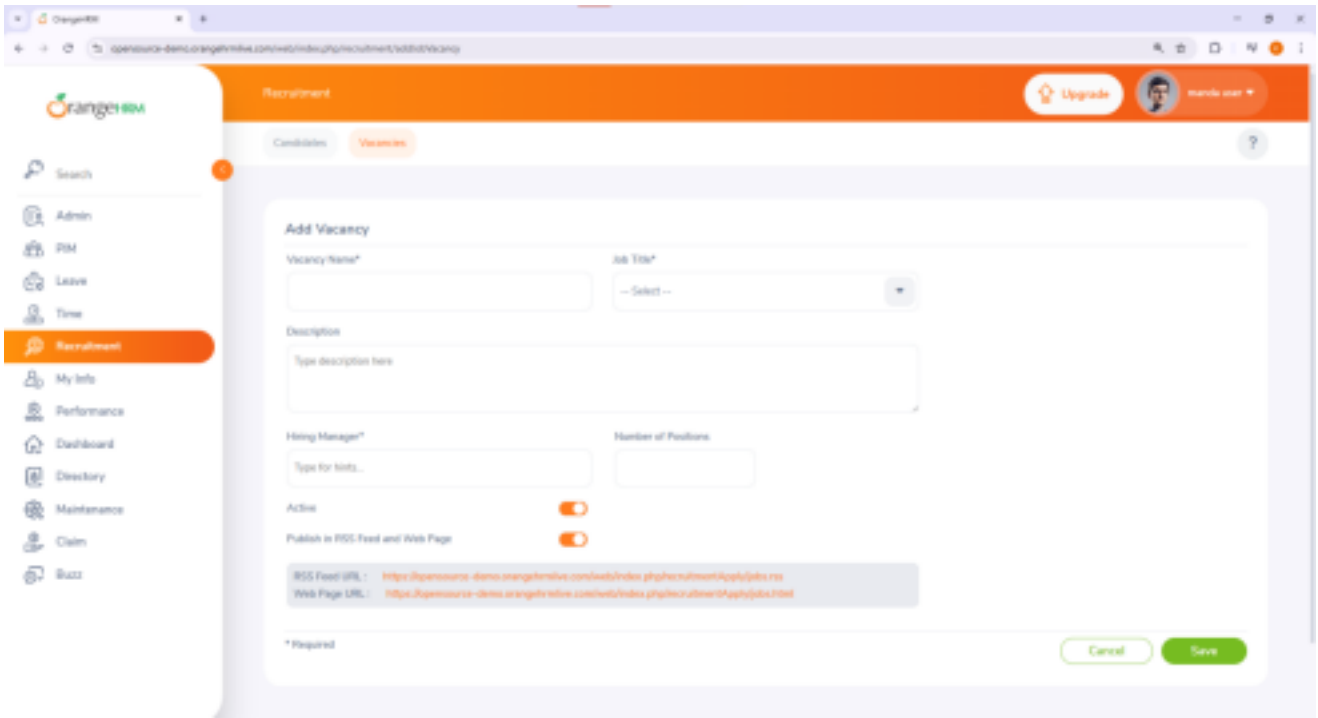


Рисунок 3.6 – *Vacancy Creation Form* ,форма створення вакансії

3.2.2 Розробка стратегії та плану тестування для обраного продукту

Розробка стратегії тестування починається з чіткого визначення цілей та меж тестування: в рамках *OrangeHRM* необхідно підтвердити коректність ключових бізнес-процесів, передбачених модулями персоналу, відпусток, рекрутингу та експорту даних. На цьому етапі формується розуміння того, які функціональні блоки потребують першочергової уваги через їхню критичність для користувачів і вплив на бізнес-логіку. Наприклад, оскільки некоректне підраховування відпусткових днів може призвести до юридичних наслідків, модуль *Leave* отримує високий пріоритет, а модуль експорту списку працівників сприймається як другорядний.

У плані тестування визначаються необхідні типи перевірок для кожного

модуля. Для *PIM*-модуля це насамперед функціональні сценарії додавання, редагування та видалення записів із валідацією полів. Модуль рекрутингу

передбачає тестування взаємозалежних полів у формі створення вакансії та процесів обробки заявок кандидатів. Нефункціональні аспекти, такі як продуктивність при масиві даних понад тисячу записів у таблиці «*Employee List*», відносяться до окремих завдань. Також важливо заздалегідь обумовити критерії завершення тестування — наприклад, досягнений відсоток пройдених тестів чи відсутність критичних дефектів протягом двох тестових циклів.

При складанні календарного плану для *OrangeHRM* доречно скористатися підходом спринтового тестування, виділивши кожному модулю по дві–три діб. Перший спринт відводиться на налаштування тестового середовища, яке включає розгортання копії системи на окремому сервері, наповнення її тестовими даними та створення облікових записів із ролями «*Admin*», «*Manager*» і «*Employee*». Другий спринт зосереджується на виконанні функціональних тестів для модулів *PIM* та *Leave* із застосуванням тест-кейсових сценаріїв, а третій — на інтеграції та нефункціональних перевірках, зокрема на навантаження та експортуванні звітів.

Документування тестової активності здійснюється в обраному інструменті керування тестами (наприклад, *TestRail*), де описуються передумови, кроки виконання, очікуваний результат та фактичний стан. Такий підхід дозволяє відстежувати прогрес, швидко реагувати на виявлені дефекти й забезпечувати прозорість процесу для всіх учасників проєкту.

Таким чином, стратегія та план тестування *OrangeHRM* базуються на пріоритетах бізнес-логіки, чіткому розподілі часу між модулями та детальному документуванні кожного етапу перевірок.

3.2.3 Проведення різних видів тестування

Проведення різних видів тестування передбачає послідовне виконання набору тестів, які охоплюють функціональне, нефункціональне та інтеграційне тестування програмного забезпечення. Спочатку здійснюється функціональне тестування для перевірки коректності роботи основних функцій системи

відповідно до вимог. Далі виконуються нефункціональні тести, зокрема тестування

продуктивності, безпеки та зручності використання, щоб оцінити якість системи у реальних умовах. Інтеграційне тестування підтверджує правильну взаємодію між модулями та зовнішніми компонентами. Весь процес супроводжується фіксацією результатів, аналізом виявлених помилок та їх пріоритезацією для подальшого виправлення.

Функціональні тести:

Назва тест-кейсу: Додавання нового працівника

• Передумови: Активний обліковий запис з роллю *Admin*; відкрита форма *Add Employee*

• Кроки для виконання: Ввести коректні дані в обов’язкові поля (ПІБ, ІНН, дата народження, *email*), натиснути *Save*

• Очікуваний результат: Переадресація на сторінку профілю з відображенням введених даних; новий запис з’являється в *Employee List* •

Фактичний процес і результат:

На рисунку 3.7 показано процес заповнення форми додавання працівника.

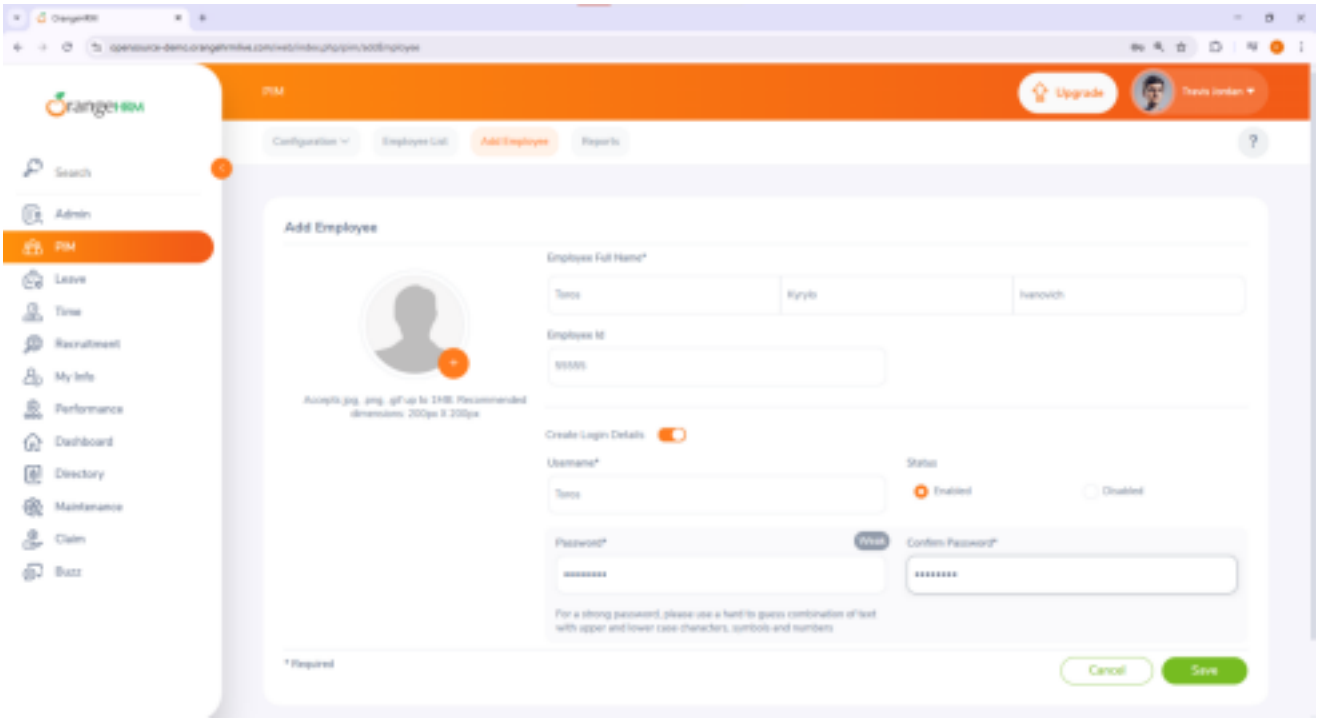


Рисунок 3.7 – У вкладці “*Add Employee*” ввод конкретних даних та створення облікового запису

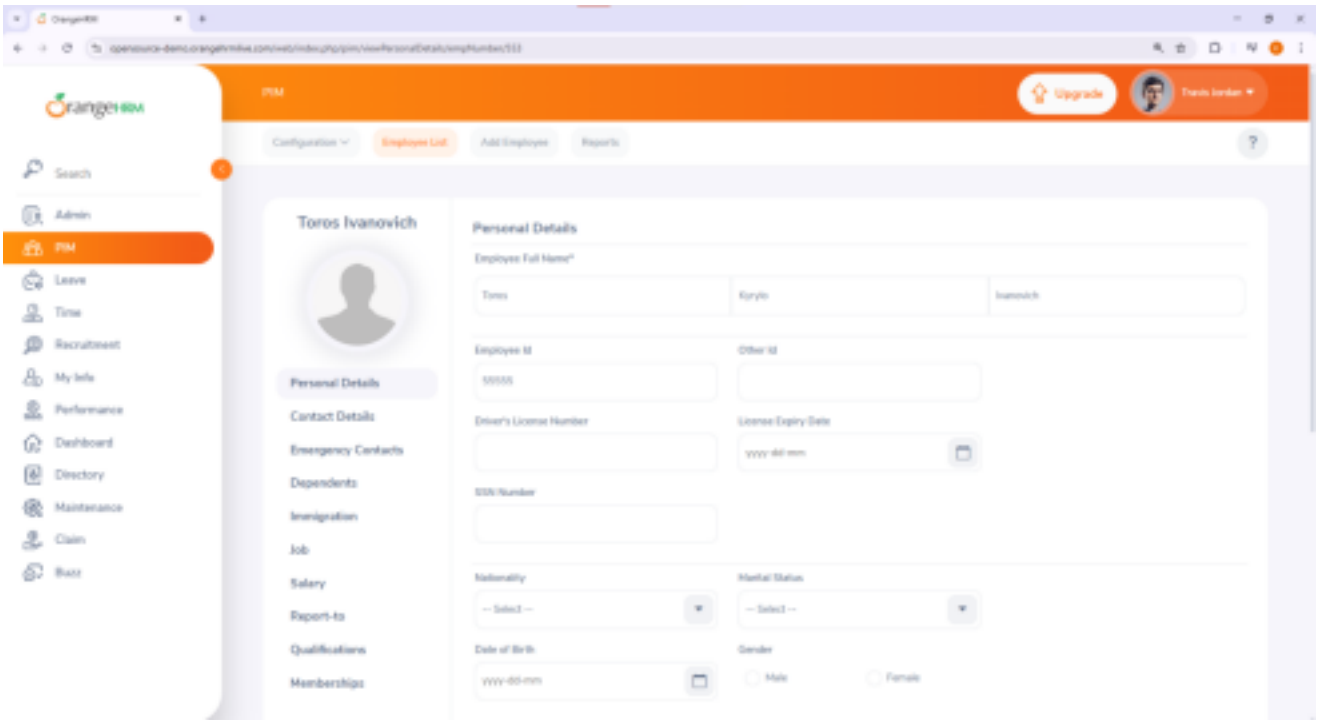


Рисунок 3.8 – Переадресація на сторінку профіля з відображенням повної інформації та можливістю редагування

Новий запис успішно з’являється у списку працівників, що видно на рисунку 3.9.

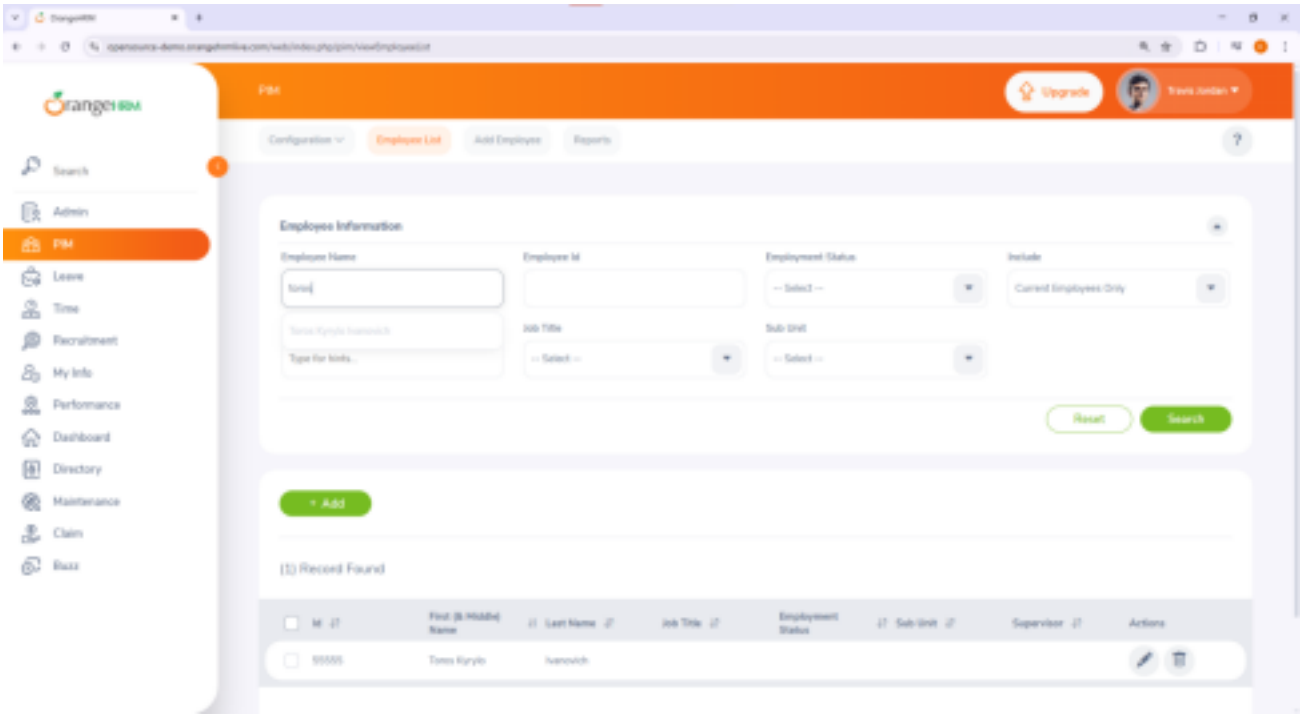


Рисунок 3.9 – Запис в *Employee List*

Статус: Пройшов.

Назва кейсу: Редагування інформації про працівника

• Передумови : Існує профіль співробітника; доступ до форми редагування • Кроки для виконання : Відкрити профіль, натиснути *Edit*, змінити дані, натиснути Save

• Очікуваний результат : У профілі й у списку працівників зміни відображаються коректно

• Фактичний процес і результат:

На рисунку 3.10 зображено створення тестового профілю для редагування.

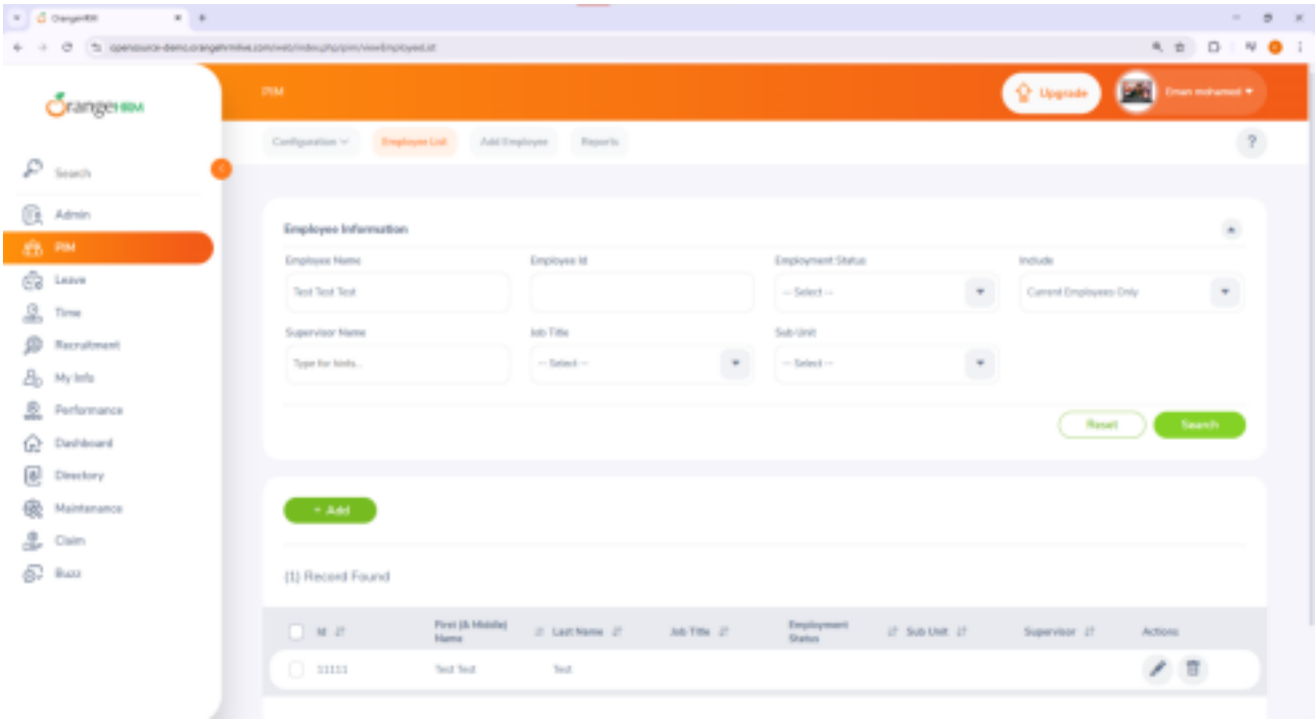


Рисунок 3.10 – Створення тестового профілю

Далі проводиться редагування даних профілю, що ілюструє рисунок 3.11.

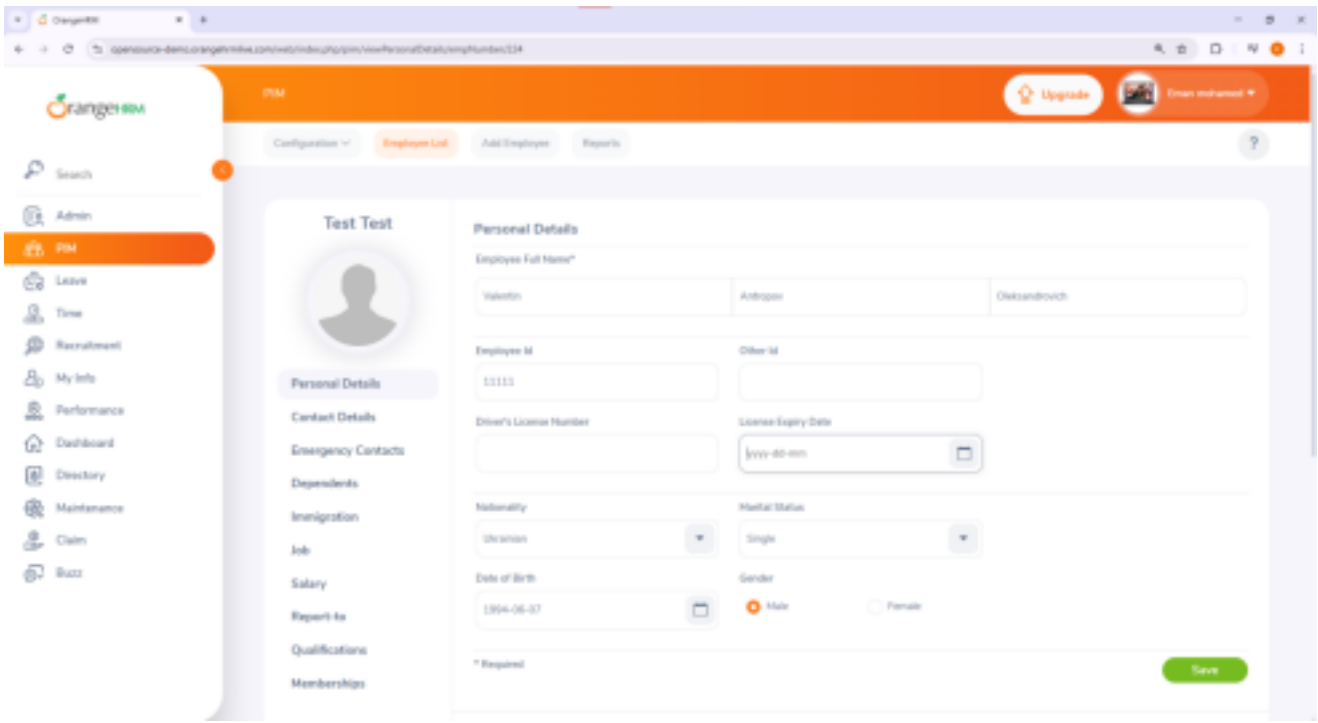


Рисунок 3.11 – Заповнення профілю іншими даними та підтвердження кнопкою “*Save*”

Після редагування виконано перевірку змін за допомогою пошуку, як показано на рисунку 3.12.

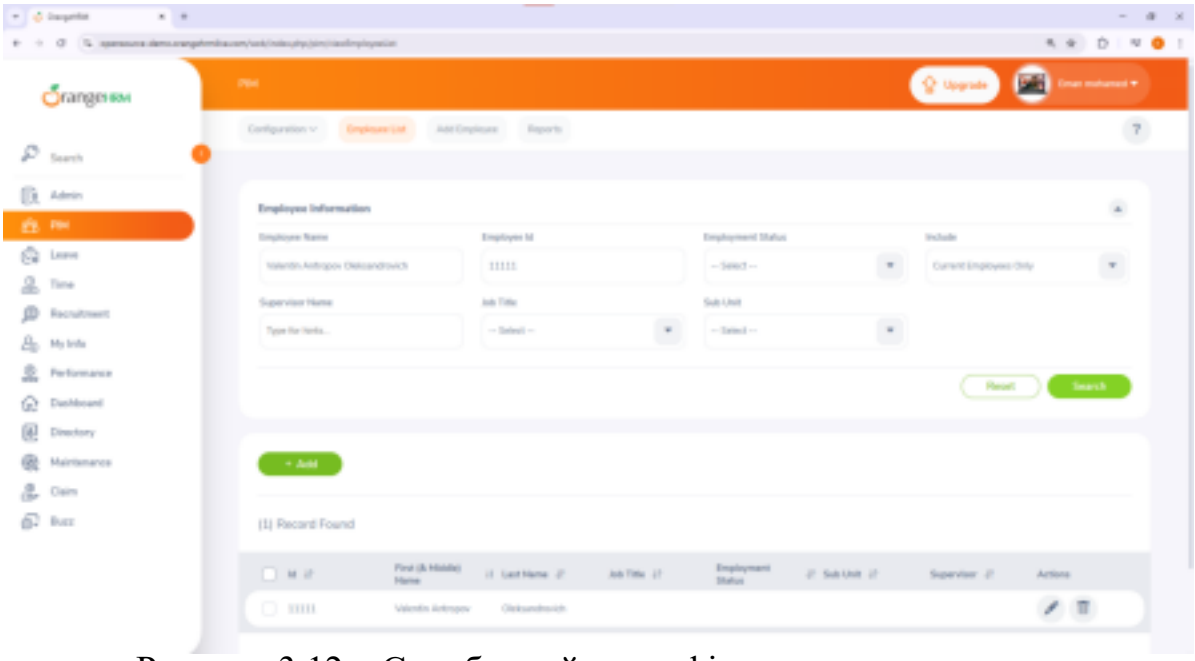


Рисунок 3.12 – Спроба знайти профіль за новими даними.

Статус: Пройшов.

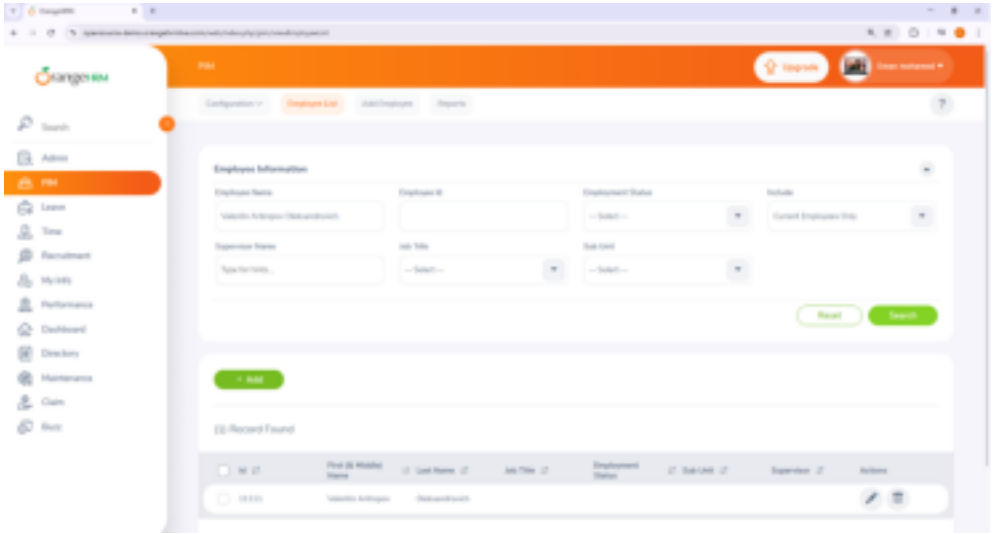
Назва кейсу: Видалення профілю працівника

• Передумови : Принаймні один запис у *Employee List*; роль *Admin* • Кроки для виконання : У списку працівників натиснути *Delete* на обраному записі, підтвердити видалення

• Очікуваний результат : Запис зникає з таблиці; при переході за прямим *URL* на профіль — повідомлення “*Record not found*”

• Фактичний процес і результат:

Перед видаленням здійснюється підготовка профілю (рисунок



3.13).

Рисунок 3.13 – Підготовка профілю для видалення

Далі виконуєтьсяпідтвердження дії, наведене на рисунку 3.14.

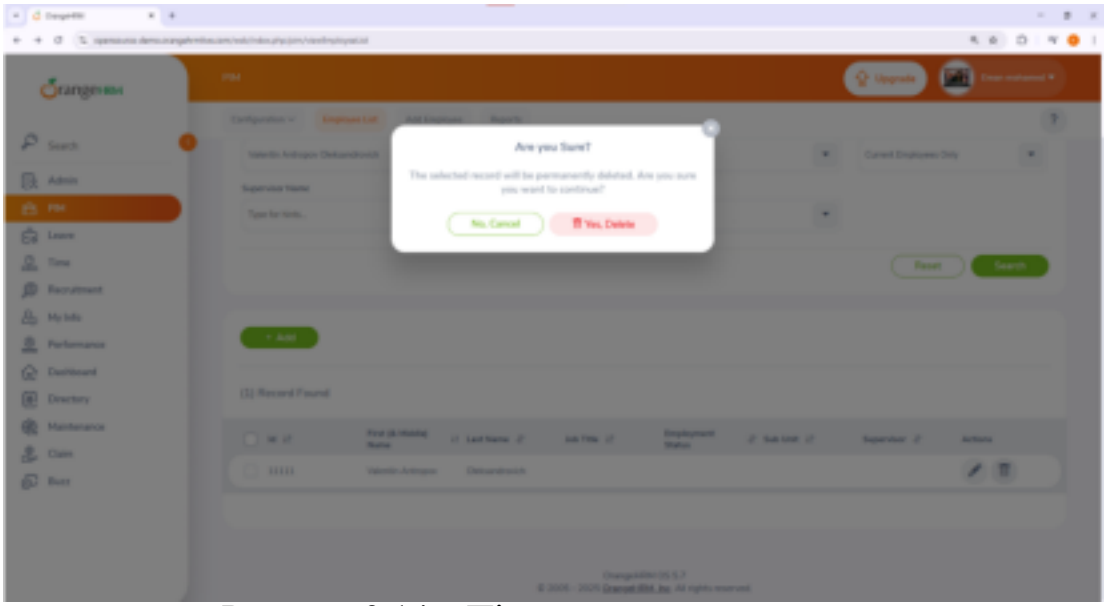


Рисунок 3.14 – Підтвердження видалення

Після видалення проведено спробу знайти запис за попередніми даними (рисунок 3.15а) і перевірено доступ до профілю за *URL* (рисунок 3.15б).

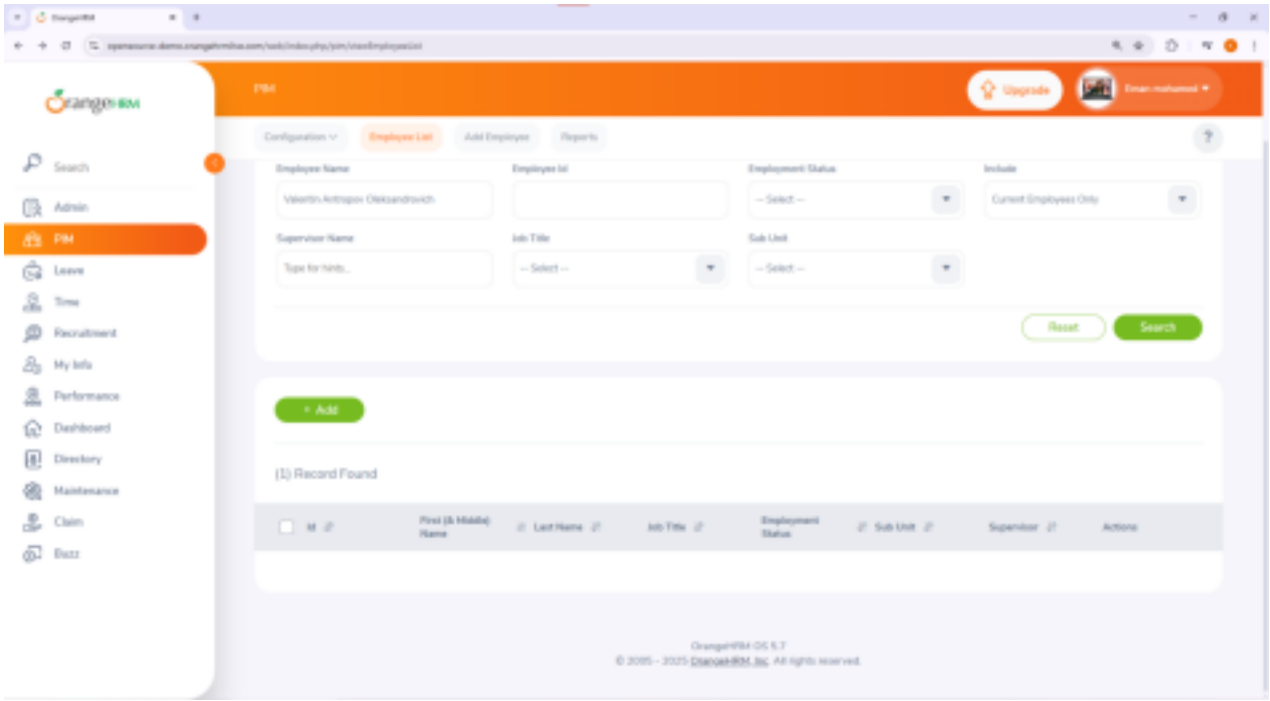


Рисунок 3.15 (а) – Пошук працівника за старими даними

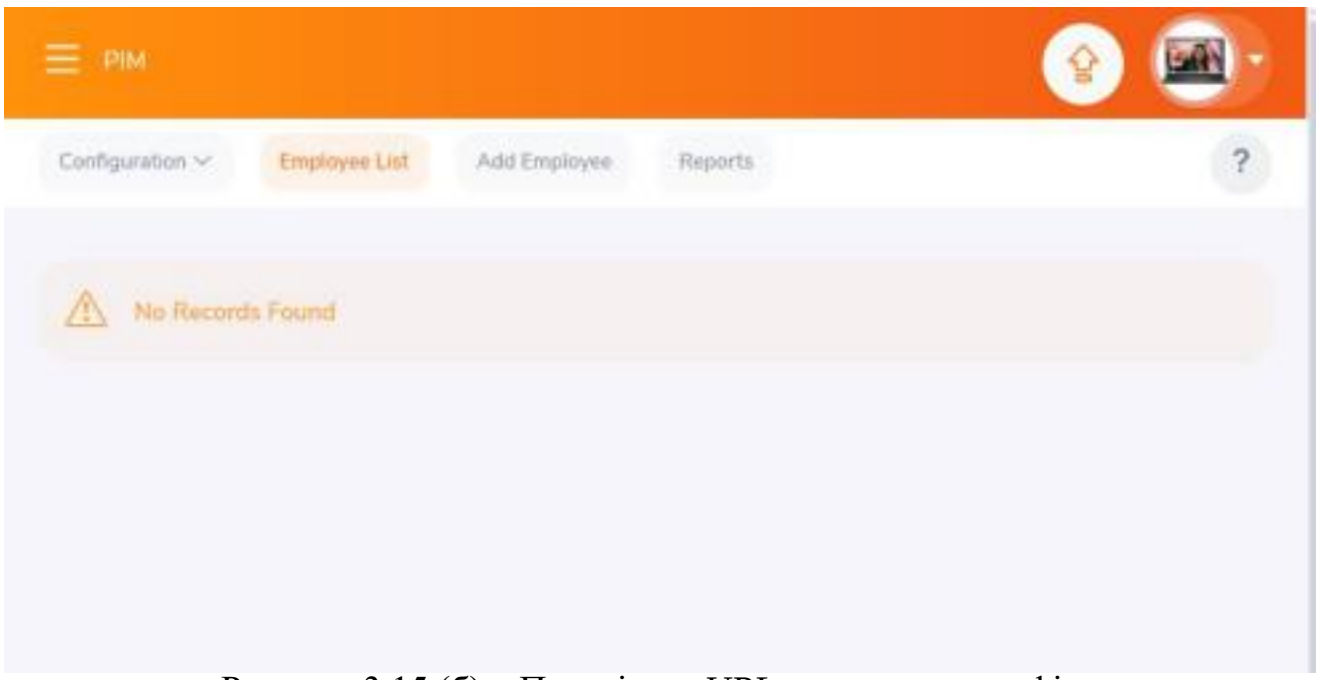


Рисунок 3.15 (б) – Перехід по *URL* посиланню профіля

Статус: Пройшов.

Нефункціональні тести :

Назва кейсу: Обмеження доступу для ролі *Employee*

• Передумови: Увійти як звичайний працівник (роль *Employee*)

• Кроки для виконання:

• Увійти в систему;Спробувати перейти до модулів "*Admin*", "*Recruitment*","*PIM*"

• Очікуваний результат: Доступ заборонено, пункти меню недоступні або повідомлення “*Access Denied*”

• Фактичний процес і результат:

На рисунку 3.16 наведено інтерфейс після входу з облікового запису працівника.

Рисунок 3.16 – Початкова сторінка при вході з звичайного аккаунта

Спроба доступу до вкладки *Admin* завершилась відмовою, що показано на рисунку 3.17.

Рисунок 3.17 – Вкладка *Admin*

На рисунку 3.18 ілюструється аналогічна ситуація для вкладки *PIM*

Рисунок 3.18 – Вкладка *PIM*, не дозволило зайти до налаштуваннь та перекинуло на власний робочий профіль

Результати переходу до модуля *Recruitment* відображено на рисунку 3.19.

Рисунок 3.19 – Вкладка *Recruitment*

Статус: Пройшов.

Назва кейсу: Відображення інтерфейсу в різних браузерах

• Передумови: Відкрити систему у *Edge*

• Кроки для виконання:

• Відкрити *OrangeHRM* у кожному браузері;Перевірити меню, таблиці, кнопки

• Очікуваний результат: Інтерфейс відображається однаково, без спотворень чи помилок

• Фактичний процес і результат:

На рисунку 3.20 представлено загальний вигляд інтерфейсу у браузері *Edge*.

Рисунок 3.20 – Інтерфейс при заході з браузеру *Edge* виглядає коректно

Рисунок 3.21 демонструє вигляд сторінки після авторизації.

Рисунок 3.21 – Інтерфейс сторінки після авторицї

На рисунках 3.22–3.24 показано відображення різних вкладок.

Рисунок 3.22 – Адмін панель

Рисунок 3.23 – Вкладка *PIM*

Рисунок 3.24 – Вкладка *Leave*

Статус: Пройшов.

Назва кейсу: Поведінка системи при великій кількості записів у списку працівників

• Передумови: У базі даних щонайменше 100 записів про працівників

• Кроки для виконання:

• Увійти як *Admin*;Перейти до *PIM* → *Employee List*;Пролистати список працівників до кінця

• Очікуваний результат: Сторінка не зависає, пагінація працює, час відгуку ≤ 3 сек

• Фактичний процес і результат:

На рисунку 3.25 відображено сторінку *PIM* із великим обсягом записів.

Рисунок 3.25 – Вкладка *PIM*

У вкладці *PIM* існує 272 облікових записи, при тестуванні цієї сторінки я намагався виявити баг шляхом навантажування, а саме швидко змінював фільтри, сторінки, швидко листав сторінку, але на моїй машині проблем не виявилось Статус: Пройшов.

3.2.4 Аналіз результатів тестування

У процесі тестування функціоналу системи *OrangeHRM* було виконано 6 тест-кейсів, що охоплювали як позитивні, так і негативні сценарії. Більшість функціональних тестів показали стабільну роботу системи. Помітні недоліки виявлено при тестуванні багатокористувацьких сценаріїв (редагування записів одночасно).

Ключові спостереження:

- *UI* взаємодія інтуїтивно зрозуміла, але не завжди обробляє граничні випадки.
- Виявлено відсутність обробки виключень при паралельному редагуванні.

- Система в загальному демонструє стабільність, однак потребує додаткових

оптимізацій для високонавантажених сценаріїв.

3.2.5 Документування виявлених дефектів та складання звіту про тестування

Після проведення функціонального та нефункціонального тестування *OrangeHRM*, результати були систематизовані у вигляді формального звіту (таблиця 3.6), що включає:

- зведену таблицю тест-кейсів,
- реєстр виявлених дефектів,
- підсумковий висновок щодо якості системи на момент тестування.

Таблиця 3.6 – Зведенна таблиця виконання тест-кейсів

№	Назва тест кейсу	Тип тесту	Статус	Примітка
1	Додавання нового працівника	Функціональний	Пройшов	
2	Редагування інформації про працівника	Функціональний	Пройшов	
3	Видалення профілю працівника	Функціональний	Пройшов	
4	Обмеження доступу для ролі *Employee*	Нефункціональний	Пройшов	
5	Відображення інтерфейсу в різних браузерах	Нефункціональний	Пройшов	*Egde,Chrome,Firefox*
6	Поведінка системи при великій кількості записів	Продуктивність	Пройшов	Перевірено на 272 записах

У ході тестування критичних або блокуючих помилок не виявлено, однак були

зафіксовані декілька непринципових *UX*-зауважень та пропозицій щодо поліпшення, які записані у таблиці 3.7.

Таблиця 3.7 – Таблиця пропозицій щодо поліпшення продукту

ID	**Назва дефекту**	**Тип**	**Опис**	**Серйозність**	**Статус**
DEF 01	Неочевидна кнопка “*Save*” при редагуванні	*UX/UI*	Кнопка зливається з фоном, не завжди помітна	Низька	Відкрито
DEF 02	Відсутнє повідомлення при видаленні профілю	*UX/ Functionality*	Після видалення немає підтвердження дії	Середня	Відкрито
DEF 03	Роль *Employee* бачить пункт меню “*PIM*”	*UX*	Хоча доступ обмежено, сам пункт залишено видимим	Низька	Відкрито

За результатами тестування жодного дефекту, що блокує роботу системи або викликає критичні помилки, не виявлено.

Підсумковий звіт про тестування

- Кількість виконаних тест-кейсів: 6
- Пройдено успішно: 6
- Кількість виявлених дефектів: 3 (низька або середня критичність) •

Якість реалізації функціоналу: Задовільна

- Рекомендації: Оптимізувати інтерфейс редагування профілю, додати системні повідомлення після видалення запису, приховати неактуальні пункти меню для ролі *Employee.*

3.3 Проблеми та перспективи розвитку тестування програмного забезпечення

У цьому підрозділі розглянуто ключові проблеми, що постають перед

сучасним тестуванням програмного забезпечення, а також напрямки його розвитку з урахуванням новітніх підходів, технологій та інструментів.

3.3.1 Актуальні проблеми в сфері тестування ПЗ

Однією з головних проблем сучасного тестування є необхідність забезпечення якості складних розподілених систем. Багато сучасних застосунків мають архітектуру, що включає в себе десятки, а то й сотні окремих компонентів, які взаємодіють через *API*, черги повідомлень, бази даних тощо. Такі системи важко перевіряти в повному обсязі, особливо коли йдеться про інтеграційне чи енд-ту-енд тестування.

Ще однією проблемою є робота з великими даними. Тестування програм, що обробляють великі обсяги інформації, потребує спеціалізованих підходів: тестові набори мають бути не лише великими, а й репрезентативними, а також реалістичними. Оцінити продуктивність, перевірити коректність алгоритмів та модулів у таких умовах — складне завдання.

Значну складність також становить тестування мікросервісної архітектури. Оскільки кожен сервіс є самостійним компонентом, тестувальникам доводиться перевіряти не лише функціональність кожного модуля, але й узгодженість і надійність взаємодії між ними. Особливу увагу тут слід приділяти симуляції відмов, мережевих затримок, а також питанням конфігурації середовища.

3.3.2 Актуальні проблеми

Однією з головних проблем сучасного тестування є необхідність забезпечення якості складних розподілених систем. Багато сучасних застосунків мають архітектуру, що включає в себе десятки, а то й сотні окремих компонентів, які взаємодіють через *API*, черги повідомлень, бази даних тощо. Такі системи важко перевіряти в повному обсязі, особливо коли йдеться про інтеграційне чи енд-ту-енд тестування.

Ще однією проблемою є робота з великими даними. Тестування програм, що обробляють великі обсяги інформації, потребує спеціалізованих підходів: тестові

набори мають бути не лише великими, а й репрезентативними, а також

реалістичними. Оцінити продуктивність, перевірити коректність алгоритмів та модулів у таких умовах — складне завдання.

Значну складність також становить тестування мікросервісної архітектури. Оскільки кожен сервіс є самостійним компонентом, тестувальникам доводиться перевіряти не лише функціональність кожного модуля, але й узгодженість і надійність взаємодії між ними. Особливу увагу тут слід приділяти симуляції відмов, мережевих затримок, а також питанням конфігурації середовища.

Крім того, в епоху *DevOps* та *CI/CD* критично важливими є стабільність і швидкість тестів. Нестабільні (*flaky*) тести, що дають різні результати за однакових умов, уповільнюють процес розробки і знижують довіру до автоматизації. А необхідність швидкого зворотного зв’язку змушує оптимізувати тестові набори та автоматизувати рутинні перевірки.

3.3.3 Роль тестування в *agile і devops*

У методологіях *Agile* і *DevOps* тестування стало не лише етапом, а безперервним процесом. Тестувальники інтегруються в команди розробників, активно беруть участь у плануванні, аналізі вимог, створенні автоматизованих тестів ще до написання коду. Часто використовується підхід *TDD* (*Test-Driven Development*) або *BDD* (*Behavior-Driven Development*), що забезпечують краще розуміння очікуваної поведінки системи.

У *DevOps* особлива увага приділяється безперервній інтеграції та доставці (*CI/CD*). Тестування має відбуватись на кожному етапі — від перевірки окремого модуля до фінального релізу. Тут зростає роль автоматизації, оскільки ручне тестування стає надто повільним.

3.3.4 Вплив штучного інтелекту

Одним з найбільш перспективних напрямів є впровадження штучного інтелекту та машинного навчання в процес тестування. *AI* використовується для

автоматичної генерації тест-кейсів, виявлення аномалій у поведінці системи, пріоритезації тестів, прогнозування можливих зон із дефектами. Це дозволяє скоротити час і підвищити ефективність виявлення помилок.

Деякі інструменти вже використовують *ML* для аналізу журналів логів, історії змін у коді, минулих дефектів — і на основі цього формують найбільш критичні сценарії для перевірки.

У майбутньому тестування програмного забезпечення ставатиме ще більш інтегрованим, автоматизованим і "розумним". Основні напрями розвитку включають:

- Поширення *low-code/no-code* рішень для автоматизації тестів;
- Поява нових фреймворків з вбудованою підтримкою *AI/ML*;
- Розвиток хмарних платформ для масштабного та паралельного тестування;
- Удосконалення інструментів спостереження (*observability*) та метрик для швидшої діагностики помилок;
- Акцент на безпеку (*Security Testing, DevSecOps*) та відповідність нормативам.

Таким чином, хоча сучасне тестування стикається з рядом проблем — від технічних до організаційних — воно також має великі перспективи завдяки технологічному прогресу, новим підходам до розробки та впровадженню інтелектуальних систем.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи на тему «Вивчення основ тестування програмного забезпечення» було всебічно проаналізовано теоретичні засади, практичні підходи та інструменти, що використовуються в сучасному процесі забезпечення якості програмних продуктів. Актуальність теми зумовлена стрімким зростанням складності програмного забезпечення, високими вимогами до його надійності, а також необхідністю скорочення часу виходу продукту на

ринок без втрати якості.

У першому розділі розглянуто фундаментальні поняття тестування ПЗ, зокрема його цілі, принципи та ключові терміни. Було визначено, що тестування є не лише етапом перевірки, але й важливою частиною усього життєвого циклу розробки, яка дозволяє виявляти дефекти на ранніх стадіях та знижувати витрати на їх усунення. Проаналізовано життєвий цикл тестування (*STLC*) та його місце у загальному процесі *SDLC*.

У другому розділі проведено детальну класифікацію видів і методів тестування. Зокрема, було розмежовано тестування за рівнем доступу до коду, за цілями, а також окремо охарактеризовано функціональні й нефункціональні аспекти тестування. Практичні приклади, такі як юніт-тестування модулів або навантажувальне тестування вебзастосунків, дозволили краще проілюструвати застосування відповідних видів тестування на практиці. Також було висвітлено як традиційні, так і сучасні методи тестування — ручне, автоматизоване, ризик орієнтоване та дослідницьке.

У третьому розділі увагу зосереджено на реальному інструментарії, що використовується в професійній діяльності тестувальника. Було детально розглянуто низку популярних інструментів, таких як *Selenium, Postman, JMeter, Jira* тощо, а також наведено приклади їх застосування. На прикладі тестування системи *OrangeHRM* було розроблено і реалізовано повноцінну стратегію тестування, що включала функціональні та нефункціональні кейси, ручне виконання тестів, а також аналіз результатів і документування виявлених дефектів. Цей приклад підтвердив

теоретичні положення та продемонстрував практичну цінність структурованого підходу до тестування.

Окрему увагу приділено сучасним викликам і тенденціям в галузі, зокрема труднощам, що виникають при тестуванні мікросервісів, *Big Data* чи розподілених систем. Розглянуто вплив методологій *Agile* і *DevOps*, які потребують більш гнучкого та безперервного підходу до тестування. Також проаналізовано перспективи, пов’язані з інтеграцією штучного інтелекту, зокрема в аспектах автоматизації, самонавчання систем тестування та прогнозування дефектів.

Підсумовуючи, слід зазначити, що якісне тестування — це не ізольований процес, а складова всього життєвого циклу розробки, яка потребує аналітичного мислення, технічної підготовки та гнучкості у підходах. Здобуті в ході дослідження знання мають як теоретичне, так і практичне значення для формування компетентного фахівця у сфері забезпечення якості програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Ad-hoc* і дослідницьке тестування [Електронний ресурс] // *IT-Notes*. – Режим доступу: https://www.it-notes.wiki/software-testing/ad-hoc-and-exploratory testing/, (дата звернення: 21.05.2025)

2. Види тестування програмного забезпечення [Електронний ресурс] // *Lemon School*. – Режим доступу: https://lemon.school/blog/vydy-testuvannya-programnogo zabezpechennya, (дата звернення: 14.05.2025)

3. Використання *SoapUI* в тестуванні [Електронний ресурс] // *QATestLab Training Center*. – Режим доступу: https://training.qatestlab.com/blog/technical articles/soapui-in-testing/?utm_source=chatgpt.com, (дата звернення: 10.06.2025)

4. Документація модуля *unittest (Python)* [Електронний ресурс]. – Режим доступу: https://docs.python.org/uk/3.13/library/unittest.html, (дата звернення: 21.05.2025)

5. *JUnit*: приклади тестування на *Java* [Електронний ресурс] // *Mate Academy*. – Режим доступу: https://mate.academy/blog/qa/junit-java-testing-examples/, (дата звернення : 16.05.2025)

6. Навчальний посібник з якості ПЗ та тестування [Електронний ресурс]. – Режим доступу: https://dspace.wunu.edu.ua/bitstream/316497/39773/1/%d0%9d%d0%b0%d0%b2%d1%87%d0%b0%d0%bb%d1%8c%d0%bd%d0%b8%d0%b9%20%d0%bf%d0%be%d1%81%d1%96%d0%b1%d0%bd%d0%b8%d0%ba%20%d0%b7%20%d1%8f%d0%ba%d0%be%d1%81%d1%82%d1%96%20%d0%9f%d0%97%20%d1%82%d0%b0%20%d1%82%d0%b5%d1%81%d1%82%d1%83%d0%b2%d0%b0%d0%bd%d0%bd%d1%8f%20(1).pdf (дата звернення: 21.04.2025)

7. Основи тестування програмного забезпечення: методичні матеріали

[Електронний ресурс]. – Режим доступу: https://knute.edu.ua/file/Mg==/0a8805eb667ec41f49257ca4b95ca304.pdf#page=91, (дата звернення: 23.04.2025)

8. Переваги *API*-тестування [Електронний ресурс] // *QAGROUP*. – Режим доступу: https://qagroup.com.ua/publications/perevagy-api-testing/, (дата звернення: 21.05.2025)

9. Постман: що це таке і як ним користуватися [Електронний ресурс] // *GoIT*. – Режим доступу: https://goit.global/ua/articles/postman-shcho-tse-take-i-iak-nym korystuvatysia/, (дата звернення: 21.05.2025)

10. Приклад лекції з тестування [Електронний ресурс]. – Режим доступу: https://financial.lnu.edu.ua/wp-content/uploads/2019/09/LEKTSIYA-1.pdf, (дата звернення: 21.04.2025)

11. Розробка вебдодатків *JavaScript* [Електронний ресурс] // *Foxminded*. – Режим доступу: https://foxminded.ua/rozrobka-veb-dodatkiv-javascript/, (дата звернення: 21.05.2025)

12. Слюсаренко О. Ю. Теоретичні основи тестування програмного забезпечення [Електронний ресурс] / О. Ю. Слюсаренко. – Режим доступу: https://eprints.cdu.edu.ua/1482/1/testyvan.pdf, (дата звернення : 21.04.2025)

13. *Swagger*: що це таке і для чого використовують [Електронний ресурс] // *Foxminded*. – Режим доступу: https://foxminded.ua/shcho-tak-swagger/, (дата звернення: 21.05.2025)

14. Що таке *JMeter* і як його використовувати в тестуванні [Електронний ресурс] // *QATestLab Training Center*. – Режим доступу: https://training.qatestlab.com/blog/technical-articles/using-jmeter-in-testing/, вільний (дата звернення: 21.05.2025)

15. Що таке юніт-тестування (*unit testing*) [Електронний ресурс] // *PMTips*. – Режим доступу: https://pmtips.com.ua/qna/shcho-take-iunit-testuvannia-unit-testing, (дата звернення : 21.05.2025)

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

РЕЦЕНЗІЯ
на кваліфікаційну роботу

випускника спеціальності: 123 «Комп'ютерна інженерія»

відділення: комп'ютерної та програмної інженерії

циклова комісія: комп'ютерних систем та мереж

Кирило ТОРОС
(ім'я, прізвище)

1. Актуальність теми: Обрана тема кваліфікаційної роботи «Вивчення основ тестування програмного забезпечення (software testing)» є актуальною.

2. Кваліфікаційна робота відповідає темі, затвердженій наказом.

3. Завдання на виконання кваліфікаційної роботи виконано у повному обсязі.

4. В результаті виконання кваліфікаційної роботи був проведений аналіз основ тестування ПЗ та практичне застосування отриманих знань на прикладі готового продукту

5. Якість виконання пояснювальної записки та ілюстративного (графічного) матеріалу відповідає вимогам Державних стандартів.

6. В кваліфікаційній роботі зроблений акцент на дані отримані з відкритих джерел.

7. Кваліфікаційна робота заслуговує оцінку «добре».

Рецензент _____
(науковий ступінь, посада)

«01» 06 2025 р. _____ Анна ДРОЗДОВА
(підпис) (ім'я, прізвище)

З рецензією ознайомлений _____ Кирило ТОРОС
(підпис) (ім'я, прізвище)

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

ВІДГУК
керівника кваліфікаційної роботи

випускника спеціальності: 123 «Комп'ютерна інженерія»

відділення: комп'ютерної та програмної інженерії

циклова комісія: комп'ютерних систем та мереж

Кирило ТОРОС
(ім'я, прізвище)

1. *Кваліфікаційна робота на тему «Вивчення основ тестування програмного забезпечення (software testing)» виконана в ініціативному порядку.*
2. *Метою кваліфікаційної роботи є ознайомлення з основами тестування ПЗ та використання цих навичок на реальних продуктах.*
3. *Кваліфікаційна робота відповідає темі, затвердженій наказом начальника коледжу.*
4. *Кваліфікаційна робота виконана здобувачем освіти самостійно.*
5. *Здобувач освіти показав високі вміння роботи з літературними джерелами, аналіз теоретичного та практичного матеріалу, приймання обґрунтованих рішень, застосовування сучасних комп'ютерних інформаційних технологій.*
6. *Кирило ТОРОС показав достатній рівень дотримання вимог державних стандартів при виконанні кваліфікаційної роботи в цілому та оформленні пояснювальної записки .*
7. *Рівень виконаної кваліфікаційної роботи заслуговує оцінку «добре», відповідає набутих випускником знань, умінь та навичок, вимогам освітньої характеристики фахівця і можливість присвоєння йому кваліфікації фахівця освітнього ступеня «Бакалавр» спеціальності 123 «Комп'ютерна інженерія».*

Керівник кваліфікаційної роботи

«01» 06 2025 р. _____ Олександр ГРИНЧЕНКО
(підпис) (ім'я, прізвище)