Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: «Розробка програмного засобу для проведення тестування в IT-компанії»

Виконала: студентка 4 курсу
групи 1ПІ-21б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Максименко О.В.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Хошаба О.М.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф. ЗІ Дудатьєв А.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедр Романюк О. Н.
«09» 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 - Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма - Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
«21» березня 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Максименко Ользі Володимирівні

1. Тема роботи - «Розробка програмного засобу для проведення тестування в IT-компанії»

Керівник роботи: Хошаба Олександр Мирославович, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. № 97.

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: кількість типів тестування: 4 (функціональне, автоматизоване, регресійне, навантажувальне); кількість основних ролей користувачів: 2 (тестувальник, розробник); кількість функціональних вимог: не більше 30; обсяг бази тест-кейсів у системі: до 10000 записів; максимальна кількість дефектів для обробки: до 3000; продуктивність системи: не менше 100 записів/секунду при фільтрації та звітності; підтримка одночасної роботи до 50 користувачів.

4. Зміст розрахунково-пояснювальної записки: вступ, аналіз стану проблеми організації тестування в IT-компаніях, огляд і порівняння сучасних засобів автоматизованого тестування, постановка задачі та формулювання функціональних вимог до системи, проєктування структури програмного засобу (архітектура, модулі, UML), реалізація модулів системи (керування тестами, дефектами,

звітністю), тестування програмного засобу та приклади сценаріїв використан
висновки, перелік використаних джерел, додатки.

5. Перелік графічного матеріалу: мета та задачі роботи, порівнял
характеристика аналогів, uml-діаграми роботи програмної системи, порівнял
характеристика програмних засобів, тестування програмної системи, висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завданн прийня
1-4	Хошаба О.М., к.т.н., доцент кафедри ПЗ	24.03.25	01.06.2

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітк
1	Аналіз проблеми, обгрунтування актуальності розробки системи та постановка задач	25.03.2025-03.04.2025	вип
2	Проектування модулів, розробка алгоритмів, структури та моделей з тестування в IT-компаніях	04.04.2025-14.04.2025	вип
3	Вибір середовища та розробка програмного забезпечення з тестування в IT-компаніях	15.04.2025-05.05.2025	вип
4	Тестування графічнох частини, впровадження програмної системи	06.05.2025-19.05.2025	вик
5	Оформлення матеріалів до захисту БКР	20.05.2025-30.05.2025	вип

Студентка Максименко О. В

(підпис) (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи Хошаба О.М

(підпис) (прізвище та ініціа

АНОТАЦІЯ

УДК 519.8:665.112

Максименко О.В. Розробка програмного засобу для проведення тестування в ІТ-компанії : бакалаврська кваліфікаційна робота зі спеціальності 121 Інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2025. 86с.

На укр. мові. Бібліогр. : 37 назв ; рис. : 18; табл. 6.

У бакалаврській кваліфікаційній роботі проведено комплексний аналіз сучасних методів обліку тестування в ІТ-компаніях. Визначено недоліки наявних рішень, такі як складність налаштування, надлишковий функціонал та недостатня адаптація до середовищ малого бізнесу.

На основі сформульованих вимог спроєктовано та реалізовано програмний засіб, що забезпечує ведення життєвого циклу тестів, управління дефектами, формування звітів та підтримку ролей користувачів.

Система відповідає сучасним підходам до тестування (Agile, CI/CD, автоматизація), зберігає високий рівень гнучкості, масштабованості та точності у веденні обліку.

Експериментальне тестування підтвердило працездатність реалізованих рішень, що дозволяє рекомендувати розроблений програмний засіб до впровадження в ІТ-компаніях для покращення процесу забезпечення якості програмного забезпечення.

Ключові слова: тестування, дефекти, програмний засіб, звітність, тест-кейси, облік тестів, ІТ-компанія

ANNOTATION

Maksymenko O.V. Development of a software tool for testing in an IT company : bachelor's thesis on the specialty 121 Software engineering, educational program - software engineering. Vinnytsia: VNTU, 2025. 86 with.

In Ukrainian language. Bibliographer : 37 titles; Fig. : 18; table 6.

The bachelor's qualification work conducted a comprehensive analysis of modern methods of testing accounting in IT companies. It identified shortcomings of existing solutions, such as the complexity of configuration, redundant functionality, and insufficient adaptation to small business environments.

Based on the formulated requirements, a software tool was designed and implemented that provides test life cycle management, defect management, report generation, and user role support.

The system meets modern testing approaches (Agile, CI/CD, automation) and maintains high flexibility, scalability, and accuracy in accounting.

Experimental testing confirmed the operability of the implemented solutions, which allows us to recommend the developed software tool for implementation in IT companies to improve the software quality assurance process.

The work contains 18 figures, 6 tables, and number of used literature sources – 36.

Keywords: testing, defects, software, reporting, test cases, test accounting, IT company

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. У сучасному світі інформаційних технологій якість програмного забезпечення (ПЗ) є одним із найважливіших чинників, що визначає успіх IT-продукту на ринку. У зв’язку з цим, тестування ПЗ виступає не лише як технічна необхідність, а й як стратегічна складова процесу розробки, яка дозволяє компаніям забезпечувати відповідність програмного продукту вимогам замовника, знижувати ризики релізу та мінімізувати витрати, пов’язані з виявленням дефектів на пізніх етапах життєвого циклу програмного забезпечення.

На практиці середніх та невеликих IT-компаній часто відсутнє єдине інтегроване рішення для обліку процесів тестування. Зазвичай використовуються фрагментовані інструменти — наприклад, Excel-таблиці, поштові повідомлення, ручні документи або різні несумісні сервіси. Такий підхід призводить до втрат інформації, ускладнює координацію між тестувальниками та розробниками, створює додаткове навантаження на менеджерів і не дозволяє отримати об'єктивну картину якості продукту. Саме тому актуальним завданням є створення спеціалізованого програмного засобу для автоматизованого обліку процесів тестування, який враховує специфіку середовища IT-компанії.

Крім того, на ринку існує чимало інструментів для обліку тестування, однак більшість з них або платні, або орієнтовані на великі підприємства зі складною структурою (наприклад, TestRail, Zephyr, Xray for Jira). Їхнє впровадження потребує значних ресурсів, часу на навчання персоналу, а також не завжди виправдане для невеликих команд, що працюють над одним або кількома проєктами одночасно. У таких випадках оптимальним рішенням стає розробка адаптивного, легкого у використанні та ефективного програмного засобу, який охоплює лише необхідний мінімум функціональності, є доступним для локального впровадження й забезпечує масштабованість у разі розширення команди.

Додатково, створення власного засобу дозволяє:

– адаптувати інтерфейс і логіку під внутрішні процеси компанії;

– інтегрувати рішення з існуючими системами розробки та контролю версій;

– забезпечити гнучке налаштування ролей і рівнів доступу;

– мінімізувати витрати на зовнішнє ліцензування.

У зв’язку з викладеним, тема розробки програмного засобу для проведення тестування в ІТ-компанії є актуальною як з практичної точки зору (сприяє підвищенню ефективності командної роботи, зменшенню дефектів, покращенню продуктивності), так і з наукової (передбачає формалізацію моделей обліку тестів, структуризацію життєвого циклу дефектів, розробку засобів аналітики результатів тестування). Отже, вибір теми обумовлений:

- зростанням значення обліку тестування у процесі розробки ПЗ у сучасних ІТ-компаніях;

- відсутністю універсального простого інструменту, який би задовольняв потреби малих команд;

- необхідністю формалізації процесів тестування як складової частини життєвого циклу ПЗ;

- потребою у створенні гнучкого, адаптивного, надійного програмного рішення, яке може бути масштабоване під потреби конкретної компанії.

Тому, в межах кваліфікаційної роботи реалізовано програмний засіб, який дозволяє здійснювати повний цикл обліку тестування: від створення тестових сценаріїв і фіксації результатів до моніторингу дефектів і формування статистичних звітів. Реалізація системи супроводжувалось побудовою UML-діаграм, визначенням ролей користувачів, розробкою графічного інтерфейсу, створенням модулів обліку та звітності, а також тестуванням функціональних можливостей. Це дозволить апробувати розроблене рішення у реальних умовах і зробити висновки щодо ефективності запропонованого підходу.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалась згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання. Метою роботи є підвищення ефективності процесу тестування в ІТ-компанії шляхом розробки програмного засобу, який забезпечує

повноцінний облік тестів, моніторинг дефектів та автоматизоване формування звітів з урахуванням ролей користувачів та підтримки різних типів тестування.

Відповідно до поставленої мети виконані наступні задачі:

- виконати аналіз сучасних методів та підходів до обліку тестування;
- виконати порівняльну характеристику програмного забезпечення з тестування в IT-компаніях;
- визначити сучасні підходи до тестування програмного забезпечення в IT кампаніях;
- виконати проектування основних модулів програмного засобу;
- описати процес управління обліком проведення тестування в IT-компанії;
- розробити функціональні вимоги до програмного засобу;
- розробити графічну частину програмного засобу;
- виконати тестування графічної частини програмного засобу;
- виконати впровадження запропонованого методу інтеграції тестування з системою управління дефектами;
- виконати запропонованого методу обліку проведення тестування програмного забезпечення.

Об’єктом дослідження є процес організації та обліку проведення тестування програмного забезпечення в IT-компаніях.

Предметом дослідження є методи, моделі та програмні засоби для автоматизованого обліку тестів, відстеження дефектів та генерування звітності у середовищі командної роботи над програмним забезпеченням.

Методи дослідження. У процесі дослідження використовувались: методи об’єктно-орієнтованого проєктування для структурування архітектури системи, UML-методологія для моделювання процесів управління тестуванням (use case та sequence діаграми), логічне та структурне моделювання для побудови функціональних зв’язків між модулями, аналітичні методи для формалізації вимог до системи та побудови критеріїв ефективності. Додатково, проведено

порівняльний аналіз сучасних інструментів тестування (Selenium, Appium, TestNG, JUnit, Postman, JMeter) для обґрунтування функціональних рішень.

Новизна отриманих результатів:

1. Запропоновано метод обліку результатів тестування, особливість якого полягає в інтеграції процесу тестування з системою управління дефектами, що дає можливість оперативно відстежувати виявлені дефекти, аналізувати прогрес тестування в реальному часі та підвищити ефективність виконання тестових робіт на 14%.

2. Запропоновано метод обліку проведення тестування програмного забезпечення, особливість якого полягає у централізованому збереженні та оновленні даних про всі проведені тестові випадки та їх результати в режимі реального часу, що дає можливість оперативно отримувати повну інформацію про перебіг тестування, підвищити ефективність виконання тестових робіт на 19%.

3. Подальшого розвитку отримав метод моніторингу процесу тестування, у якому, на відміну від існуючих, об’єднано облік ручних і автоматизованих тестів та інтегровано інформацію про пов’язані дефекти, що дозволило забезпечити цілісне представлення процесу забезпечення якості програмного продукту.

Практична цінність отриманих результатів. Практичне значення роботи полягає у розробці, тестуванні та впровадженні програмного засобу для обліку проведення тестування в IT-компанії, що дозволяє оптимізувати процес забезпечення якості програмного забезпечення. Впровадження розробленого програмного засобу дозволяє значно підвищити ефективність процесу тестування, оптимізувати використання ресурсів компанії та покращити якість кінцевих програмних продуктів.

Особистий внесок здобувача. Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі, отримані автором особисто. Автору належать такі результати: постановка задачі дослідження; розроблені алгоритми; модулі клієнтської та серверної частини.

Апробація результатів роботи. Результати роботи були представленні на XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів.

Публікації. Результати роботи були опубліковані в матеріалах XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. «Стан, досягнення і перспективи інформаційних систем і технологій». Одеса, 17 - 18 квітня 2025 р. [1].

1 АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасних методів та підходів до обліку тестування

Аналіз сучасних методів та підходів до тестування програмного забезпечення (ПЗ) в ІТ-компаніях є ключовим аспектом забезпечення якості продуктів. Останні декілька років відзначилися розвитком нових методологій та вдосконаленням існуючих практик у цій сфері.

Тестування ПЗ є невід'ємною частиною процесу розробки, спрямованою на виявлення та виправлення дефектів до випуску продукту на ринок. Це гарантує, що кінцевий продукт відповідає встановленим вимогам та очікуванням користувачів. Без належного тестування можливі збої в роботі, що можуть призвести до втрати довіри клієнтів та фінансових втрат для компанії.

У відповідь на зростаючі вимоги до якості та швидкості розробки ПЗ, ІТ-компанії впроваджують сучасні методології, такі як Agile та DevOps. Ці підходи акцентують увагу на гнучкості, швидкому реагуванні на зміни та тісній співпраці між командами розробки і тестування. Крім того, автоматизація тестування стає все більш поширеною, що дозволяє прискорити процеси перевірки та зменшити ймовірність людських помилок.

Тестування програмного забезпечення (ПЗ) відіграє ключову роль у забезпеченні його якості та надійності. Залежно від рівня залучення людини у виконання тестових сценаріїв, методи тестування поділяються на ручне тестування та автоматизоване тестування (рис. 1.1).

Рисунок 1.1 - Поділ методів тестування програмного за ступенем автоматизації

Ручне тестування передбачає, що тестувальник самостійно виконує тест-кейси без використання спеціалізованих автоматизованих інструментів. Він перевіряє роботу програмного продукту шляхом взаємодії з інтерфейсом користувача, аналізу відповідності фактичних результатів очікуваним та виявлення дефектів [7,9,10].

До переваг ручного тестування відноситься гнучкість та можливість тестування без необхідності попередньої підготовки скриптів, висока ефективність для тестування UI/UX, адже тестувальник може оцінити зручність використання продукту, простота застосування, що не вимагає технічних знань з програмування та роботи з автоматизованими фреймворками.

До недоліків ручного тестування відноситься висока трудомісткість процесу, де тестування займає більше часу, оскільки кожен тест виконується вручну, схильність до людських помилок, де можливі неточності або пропущення помилок через втому тестувальника, низька ефективність для регресійного тестування, коли потрібно перевіряти одні й ті самі сценарії багаторазово.

До прикладів застосування ручного тестування відноситься перевірка UI/UX у мобільних та веб-додатках, виконання дослідницького тестування, коли сценарії тестування не визначені заздалегідь, тестування нових функцій, які ще не мають стабільних тест-кейсів [7,8].

Автоматизоване тестування передбачає використання спеціалізованого програмного забезпечення, яке дозволяє виконувати тест-кейси автоматично без участі людини. Автоматизовані скрипти або сценарії написані заздалегідь і виконуються інструментами тестування.

До переваг автоматизованого тестування відносяться значне зменшення часу тестування, оскільки тести виконуються швидко і можуть запускатися паралельно та висока точність, оскільки автоматизовані скрипти виконують тестові сценарії однаково при кожному запуску. Ефективне регресійне тестування також дозволяє перевіряти вже реалізований функціонал після внесення змін у код.

До недоліків автоматизованого тестування відноситься висока початкова вартість розробки та підтримки автоматизованих тестів та обмеження у тестуванні

UI/UX, оскільки автоматизовані скрипти не можуть оцінити зручність інтерфейсу так, як це зробить людина [7,9]. Автоматизовані тести також можуть давати хибнопозитивні або хибнонегативні результати через зміни у програмному коді або тестовому середовищі.

До прикладів застосування автоматизованого тестування відносяться регресійне тестування, коли необхідно багаторазово перевіряти працездатність функціоналу, тестування продуктивності (навантажувальне тестування) для визначення меж стійкості системи та API-тестування, яке дозволяє швидко перевіряти взаємодію між сервісами без необхідності тестування через інтерфейс користувача [10].

Таким чином, ручне та автоматизоване тестування мають свої переваги і недоліки, тому вибір методології залежить від конкретних потреб проєкту. Автоматизоване тестування доцільне для регресійного тестування та великих проєктів, де важливо швидке виявлення помилок. Водночас ручне тестування залишається незамінним для оцінки UI/UX та унікальних кейсів, які складно автоматизувати [11,12].

Тестування програмного забезпечення (ПЗ) можна також класифікувати за підходом до аналізу системи на два основні методи: тестування "чорної скриньки" та тестування "білої скриньки" (рис. 1.2).

Рисунок 1.2 - Поділ методів тестування програмного за підходом до аналізу системи

Розглянемо тестування за методом "чорної скриньки" (Black Box Testing). Цей метод тестування передбачає перевірку функціональності ПЗ без знання його внутрішньої структури або коду. Тестувальник взаємодіє з системою через її інтерфейси, вводить вхідні дані та аналізує отримані вихідні, оцінюючи відповідність фактичних результатів очікуваним. Цей підхід дозволяє оцінити, наскільки програма відповідає вимогам та очікуванням користувачів [13].

До основних характеристик тестування "чорної скриньки" відносяться наступні:

- фокус на функціональності, де перевіряється, чи виконує система свої функції відповідно до специфікацій;

- незалежність від коду: тестувальнику не потрібно знати внутрішню реалізацію або структуру коду.

Тестування за методом "чорної скриньки" ефективно використовується для перевірок на рівні системи, інтеграції та приймального тестування.

До поширених прикладів технік тестування "чорної скриньки" можна віднести аналіз граничних значень, де є перевірка системи на межах допустимих вхідних даних та виконання еквівалентного розбиття, де є розподіл вхідних даних на класи еквівалентності для зменшення кількості тестів та задіяння таблиці прийняття рішень, де є використання таблиць для відображення логіки прийняття рішень системою [14].

Розглянемо тестування за методом "білої скриньки" (White Box Testing). Цей метод тестування базується на аналізі внутрішньої логіки та структури коду ПЗ. Тестувальник має доступ до вихідного коду та використовує свої знання для розробки тестових сценаріїв, які перевіряють внутрішні шляхи, умови та цикли в програмі [15].

До основних характеристик тестування "білої скриньки" відносяться наступні [16,17]:

- фокус на структурі коду, де перевіряється коректність внутрішньої реалізації, включаючи логіку, структуру даних та алгоритми;

- вимога знань програмування, де тестувальник повинен розуміти код та принципи його роботи.

Тестування за методом "білої скриньки" ефективно використовується для модульного тестування та тестування безпеки.

До поширених прикладів технік тестування "білої скриньки" можна віднести покриття операторів (Statement Coverage) де є перевірка виконання кожного оператора в коді хоча б один раз, покриття гілок (Branch Coverage), де є перевірка виконання всіх можливих гілок умовних операторів та покриття шляхів (Path Coverage), де є перевірка всіх можливих шляхів через програму [16].

Тестування програмного забезпечення (ПЗ) відбувається на різних рівнях, кожен з яких має свою специфічну мету та охоплення (рис. 1.3). Розглянемо основні з них, до яких відносяться наступні.

Рисунок 1.3 - Поділ методів тестування програмного за рівнями

Модульне тестування, часто відоме як юніт-тестування, полягає в перевірці окремих модулів або компонентів програмного забезпечення в ізоляції, щоб переконатися, що вони працюють правильно. Це зазвичай виконують розробники на етапі розробки, використовуючи методи білої скриньки, де тестувальник знає внутрішню структуру модуля. Наприклад, якщо є функція для обчислення суми,

модульне тестування перевірить, чи вона правильно додає числа. Це допомагає виявити помилки на ранніх стадіях, що знижує витрати на їх виправлення [18].

Модульне тестування, часто синонімічне з юніт-тестуванням, зосереджене на перевірці окремих модулів або компонентів програмного забезпечення в ізоляції [19,20]. Модуль може бути функцією, класом або іншою одиницею коду, яка виконує певну задачу (табл. 1.2).

Таблиця 1.2 - Порівняльна таблиця методів тестування

Рівень тестування	Мета	Переваги
Модульне тестування	Перевірити окремі модулі	Раннє виявлення дефектів, економія коштів
Інтеграційне тестування	Перевірити взаємодію модулів	Виявлення проблем інтерфейсів
Системне тестування	Перевірити всю систему	Забезпечення якості системи
Приймальне тестування	Перевірити відповідність потребам	Підтвердження готовності до використання

Інтеграційне тестування перевіряє, як різні модулі або компоненти взаємодіють між собою, щоб переконатися, що вони працюють разом, як очікується. Це виконують тестувальники після модульного тестування. Є кілька підходів, таких як тестування зверху вниз, знизу вгору або "великий вибух", де всі модулі інтегруються одночасно [20]. Наприклад, якщо один модуль обробляє вхідні дані, а інший їх обробляє, інтеграційне тестування перевірить, чи дані правильно передаються. Це допомагає виявити проблеми на межах модулів.

Системне тестування оцінює всю систему, щоб переконатися, що вона відповідає зазначеним вимогам і працює, як очікується, в реальних умовах. Це виконують тестувальники після інтеграційного тестування і включає функціональне тестування, тестування продуктивності, безпеки тощо. Наприклад, якщо система призначена для онлайн-магазину, системне тестування перевірить,

чи можна зробити замовлення, чи система витримує навантаження під час піку продажів [21]. Це забезпечує загальну якість системи.

Системне тестування є наступним рівнем після інтеграційного тестування і зосереджене на перевірці всієї системи як єдиного цілого. Воно перевіряє, чи система відповідає всім зазначеним вимогам, включаючи функціональні та нефункціональні аспекти, такі як продуктивність і безпека.

Приймальне тестування перевіряє, чи система відповідає потребам кінцевих користувачів і готова до впровадження. Це зазвичай виконують самі користувачі або клієнти, часто за участю тестувальників. Воно включає тестування прийняття користувачами (UAT), альфа-тестування та бета-тестування. Наприклад, якщо це банківський додаток, користувачі перевірять, чи зручно ним користуватися для переказів. Це остатній крок перед випуском, щоб переконатися, що система відповідає бізнес-вимогам.

Приймальне тестування є останнім етапом перед випуском програмного забезпечення і зосереджене на перевірці, чи система відповідає потребам кінцевих користувачів або клієнтів. Це не технічне тестування, а перевірка з точки зору користувача [22].

Таким чином, кожен рівень тестування відіграє важливу роль у забезпеченні якості програмного забезпечення. Модульне тестування закладає основу, перевіряючи окремі компоненти, інтеграційне тестування забезпечує їхню сумісність, системне тестування підтверджує загальну функціональність, а приймальне тестування гарантує, що система відповідає потребам користувачів. Ці рівні разом створюють структурований підхід до тестування, який допомагає виявити дефекти на ранніх стадіях і забезпечити надійність продукту.

1.2 Порівняльна характеристика програмного забезпечення з тестування в IT-компаніях

В цьому підрозділу надамо детальний аналіз програмного забезпечення для тестування, використовуваного в IT-компаніях, зосереджуючись на інструментах Selenium, Appium, JUnit, TestNG, Postman і JMeter. Кожен інструмент описано з

урахуванням його призначення, ключових функцій і джерел інформації, а також проведено порівняльну характеристику для кращого розуміння їх ролі в процесі тестування.

Selenium є популярним відкритим інструментом для автоматизації веб-браузерів [24], який використовується для функціонального тестування веб-додатків на різних браузерах і платформах. Він підтримує кілька мов програмування, таких як Java, Python, C#, і має функцію Selenium Grid для паралельного тестування, що дозволяє масштабувати тести на кількох середовищах. Це інструмент, який став стандартом для автоматизованого тестування веб-додатків, забезпечуючи кросбраузерне тестування.

Appium — це відкритий інструмент для автоматизації мобільних додатків [25], який дозволяє тестувати як додатки для iOS, так і для Android за допомогою одного API, базованого на протоколі WebDriver від Selenium. Він підтримує кілька мов програмування, таких як Java, Python, Ruby, і забезпечує кросплатформне тестування мобільних додатків, що робить його ідеальним для тестування нативних, гібридних і мобільних веб-додатків.

JUnit — це рамка для юніт-тестування для Java [26], створена Кентом Беком і Еріком Мідом. Вона використовується для написання та виконання тестів для Java-додатків, надаючи анотації, такі як @Test, і твердження для спрощення написання тестів. JUnit підтримує тест-сюїти та параметризовані тести, що робить її незамінною для розробників, які дотримуються підходу тест-драйвн-розробки (TDD).

TestNG — це фреймворк для тестування [27], що використовується та подібна до JUnit, але з розширеними можливостями. Вона підтримує кілька мов програмування, хоча найчастіше використовується з Java, і пропонує функції, такі як паралельне тестування, групування тестів, тестування на основі даних за допомогою анотації @DataProvider. TestNG підтримується різними інструментами, такими як Eclipse, IntelliJ IDEA, Maven, і є популярним вибором для як юніт-, так і інтеграційного тестування.

Postman — це інструмент для тестування та розробки API [28], який дозволяє проектувати, тестувати та документувати API, особливо RESTful API. Він має зручний графічний інтерфейс для відправки HTTP-запитів, включаючи методи GET, POST, PUT, DELETE, і підтримує ланцюжки запитів, твердження, скрипти та колекції для автоматизації тестів. Postman також інтегрується з інструментами CI/CD, такими як Jenkins, для автоматизації процесів тестування API.

JMeter — це відкритий інструмент для тестування навантаження та вимірювання продуктивності веб-додатків та інших сервісів [29]. Він симулює велику кількість користувачів, які звертаються до сервера, для тестування його продуктивності за різних умов навантаження, таких як пік використання або стрес-тестування. JMeter підтримує кілька протоколів, включаючи HTTP, FTP, JDBC, і може використовуватися для функціонального тестування, хоча його основна роль — тестування продуктивності.

Для порівняння інструментів проведемо аналіз їх категорій, основних функцій і ключових особливостей (табл. 1.2). Оскільки інструменти належать до різних категорій, порівняння проводилося в межах схожих груп. Розглянемо автоматизацію веб- та мобільних додатків.

Таблиця 1.2 - Порівняльна характеристика програмних інструментів тестування

Інструмент	Категорія	Основна функція
Selenium	Автоматизація веб	Функціональне тестування веб-додатків
Appium	Автоматизація мобільних	Тестування мобільних додатків
JUnit	Фреймворк тестування	Юніт-тестування Java
TestNG	Фреймворк тестування	Розширене тестування, паралельність
Postman	Тестування API	Функціональне тестування API
JMeter	Тестування продуктивності	Тестування навантаження, продуктивність

Selenium і Appium є інструментами для автоматизованого тестування, але мають різні сфери застосування. Selenium фокусується на веб-додатках, забезпечуючи кросбраузерне тестування, тоді як Appium призначений для мобільних додатків, підтримуючи як Android, так і iOS. Обидва інструменти підтримують кілька мов програмування, що полегшує їх інтеграцію в існуючі проекти.

Далі, розглянемо поширені фреймворки для тестування якими є JUnit і TestNG. JUnit і TestNG є фреймворками для тестування, але мають різні рівні складності. JUnit є простішою і призначена для базового юніт-тестування Java, тоді як TestNG пропонує розширені функції, такі як паралельне тестування, групування тестів і тестування на основі даних. Це робить TestNG більш гнучким для складних сценаріїв тестування.

Для тестування API та продуктивності використовуються Postman і JMeter. Postman і JMeter мають різну мету у використанні. Postman спеціалізується на функціональному тестуванні API, надаючи зручний інтерфейс для створення, тестування та документування API. JMeter, навпаки, фокусується на тестуванні продуктивності, симулюючи навантаження на сервери, хоча також може використовуватися для функціонального тестування HTTP-запитів. Це неочікувано, що JMeter, відомий як інструмент для тестування навантаження, також може бути використаний для базового функціонального тестування, що розширює його застосування.

Таким чином, кожен з розглянутих інструментів має свою нішу в тестуванні програмного забезпечення. Так, Selenium і Appium є незамінними для автоматизованого тестування веб- та мобільних додатків, JUnit і TestNG забезпечують рамки для юніт- та інтеграційного тестування, а Postman і JMeter доповнюють стратегію тестування, фокусуючись на API та продуктивності відповідно. Їхнє використання залежить від конкретних потреб проекту, таких як тип додатка, рівень автоматизації та вимоги до продуктивності.

1.3 Сучасні підходи до тестування програмного забезпечення в IT кампаніях

Сучасні підходи до тестування в IT-компаніях є критичним для забезпечення якості та надійності програмного забезпечення, особливо з урахуванням швидкого розвитку технологій та зростаючої складності систем.

Такі сучасні підходи до тестуванням охоплюють планування, організацію, виконання та моніторинг тестувальних активностей протягом усього циклу розробки програмного забезпечення (SDLC). Воно забезпечує, що програмне забезпечення відповідає вимогам якості, зменшує ризик дефектів у виробничому середовищі та відповідає очікуванням користувачів. Зростаюча залежність від цифрових рішень змушує IT-компанії адаптувати свої стратегії тестування до сучасних викликів, таких як швидкі цикли випуску та складні архітектури програмного забезпечення.

Управління тестуванням охоплює планування, організацію, виконання та моніторинг тестувальних активностей протягом усього циклу розробки програмного забезпечення (SDLC). Воно забезпечує, що програмне забезпечення відповідає вимогам якості, зменшує ризик дефектів у виробничому середовищі та відповідає очікуванням користувачів. Зростаюча залежність від цифрових рішень змушує IT-компанії адаптувати свої стратегії тестування до сучасних викликів, таких як швидкі цикли випуску, складні архітектури програмного забезпечення та актуальні напрямки сучасних підходів до тестування програмного забезпечення в IT кампаніях (рис. 1.4). Далі розглянемо кожного з них.

Agile тестування є ітеративним підходом, що інтегрується з методологією Agile, яка фокусується на гнучкості та швидкій адаптації до змін. Воно включає безперервне тестування, розробку на основі тестів (TDD) та часті цикли зворотного зв’язку (табл. 1.2).

До переваг Agile тестування відноситься швидший вивід продуктів на ринок, покращення якості завдяки ранньому виявленню дефектів, краща співпраця між командами розробки та тестування.

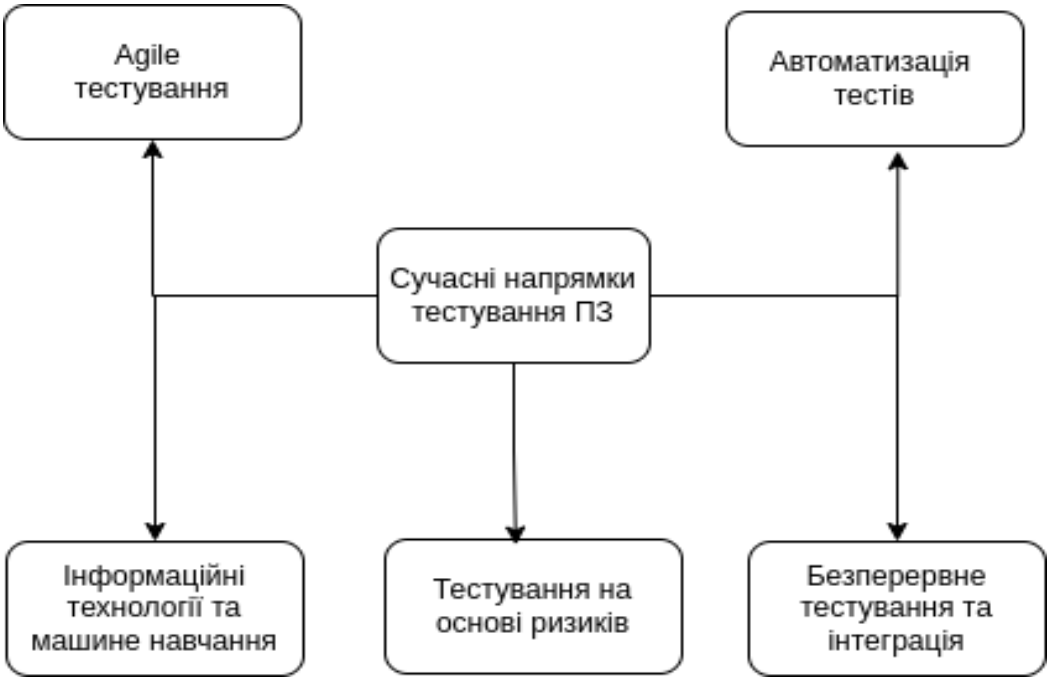

Рисунок 1.4 - Сучасні напрямки тестування програмного забезпечення в IT кампаніях

Компанії, такі як Atlassian, активно використовують Agile тестування для оптимізації процесів розробки, що дозволяє швидше реагувати на зміни вимог клієнтів (Atlassian). Так, дослідження показують [30,31], що 86% розробників програмного забезпечення по всьому світу використовують Agile тестування, що свідчить про його широке впровадження.

Автоматизація тестів передбачає використання інструментів, таких як Selenium, Appium та TestNG, для автоматизації виконання тестів, що зменшує ручну працю та підвищує ефективність.

До переваг автоматизації тестів відноситься збільшення охоплення тестів, скорочення часу виконання тестів, покращення точності завдяки зменшенню людських помилок. Так, згідно з доповіддю Gartner Peer Community (2024) [32,33], 56% IT-компаній автоматизують тестування API, 45% - інтеграційне тестування, а 40% - тестування продуктивності, що свідчить про значне поширення цього методу (Gartner).

Таблиця 1.2 - Порівняння сучасних методів тестування

Метод	Основна перевага	Приклади інструментів/технологій
Agile тестування	Швидший вивід на ринок, краща співпраця	Jira, Trello
Автоматизація тестів	Збільшення охоплення, точність	Selenium, Appium, TestNG
Інформаційні технології та машине навчання	Покращення прогнозування, ефективність	Parasoft, KaneAI
Тестування на основі ризиків	Ефективність ресурсів, фокус на критичних зонах	TestRail
Безперервне тестування	Раннє виявлення дефектів, швидкі цикли	CI/CD pipelines, Jenkins

Компанії, такі як NeoSoft, успішно впровадили автоматизоване тестування для скорочення часу тестування та підвищення якості, використовуючи Selenium для веб-додатків [34].

Інформаційні технології та машине навчання дедалі частіше застосовуються для автоматичної генерації тест-кейсів, прогнозування дефектів та оптимізації процесів тестування.

До переваг інформаційних технології та машине навчання відноситься підвищення ефективності тестів, кращий прогноз дефектів, скорочення часу тестування завдяки автоматизованому аналізу.

Компанії, такі як Parasoft, інтегрують AI для оптимізації тестування, використовуючи машинне навчання для аналізу кодового покриття та результатів тестів. Дослідження EY [35] у 2024 році показує, що 85% IT-компаній інтегрували додатки в свої технологічні стеки за останні роки, хоча 68% стикалися з проблемами продуктивності, що підкреслює необхідність ефективного тестування.

Тестування на основі ризиків пріоритизує тестування на основі ризиків, пов’язаних з різними частинами програмного забезпечення, фокусуючись на критичних областях з найбільшим потенціалом впливу.

До переваг тестування на основі ризиків відноситься ефективне використання ресурсів, скорочення часу тестування, покращення якості у високоризикованих зонах. Таке тестування включає ідентифікацію ризиків, оцінку їх ймовірності та впливу, а також розробку стратегій пом’якшення.

Компанії, такі як TestRail, рекомендують RBT для оптимізації тестування в умовах обмежених ресурсів, фокусуючись на критичних функціях [35].

Безперервне тестування передбачає тестування програмного забезпечення на кожному етапі розробки, інтегруючись із конвеєрами CI/CD.

До переваг безперервного тестування та інтеграції відноситься раннє виявлення дефектів, швидші цикли випуску, покращення якості програмного забезпечення.

Відносно безперервного тестування та інтеграції Atlassian підкреслює його важливість для створення безперервного конвеєра доставки, що автоматично перевіряє код на кожному етапі (Atlassian).

Відоме дослідження показує [36], що 40% IT-компаній проводять автоматизоване тестування безперервно протягом циклу розробки, що сприяє швидкому зворотному зв’язку.

Таким чином, сучасні методи, такі як Agile тестування, автоматизація, інформаційні технології та ML, тестування на основі ризиків та безперервне тестування, є ключовими для управління тестуванням в IT-компаніях. Вони дозволяють покращити якість програмного забезпечення, скоротити час випуску та залишатися конкурентоспроможними в умовах швидких змін. Важливо, щоб IT-компанії адаптували свої стратегії тестування до цих тенденцій, враховуючи останні дослідження та інструменти, щоб відповідати вимогам сучасного ринку.

1.4 Постановка задач дослідження

Проаналізувавши існуючі підходи щодо використання сучасних методів та підходів щодо обліку проведення тестових робіт в IT-компанії було визначено комплекс задач, реалізація яких дозволить розробити ефективне програмне забезпечення.

Основними задачами роботи є:

- виконати порівняльну характеристику програмного забезпечення з тестування в IT-компаніях;

- визначити сучасні підходи до тестування програмного забезпечення в IT кампаніях;

- виконати проектування основних модулів програмного засобу;

- описати процес управління обліком проведення тестування в IT-компанії;

- розробити функціональні вимоги до програмного засобу;

- розробити графічну частину програмного засобу;

- виконати тестування графічної частини програмного засобу;

- виконати впровадження запропонованого методу інтеграції тестування з системою управління дефектами;

- виконати запропонованого методу обліку проведення тестування програмного забезпечення.

Технічне завдання на розробку наведено в додатку А.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

2.1 Процес управління обліком проведення тестування в IT-компанії

Управління обліком проведення тестування є критично важливим для забезпечення якості програмного забезпечення в IT-компаніях. Це включає систематичне записування, відстеження та звітність усіх дій, пов’язаних із тестуванням, щоб гарантувати, що продукт відповідає вимогам і готовий до випуску. Процес охоплює кілька етапів, які допомагають командам ефективно керувати ресурсами та документами (табл. 2.1). Для цього розглянемо огляд процесу, який IT-компанії зазвичай використовують для управління обліком проведення тестування, включаючи всі аспекти, виявлені під час аналізу. Він охоплює планування, виконання, відстеження, звітність, архівування та враховує особливості регульованих галузей.

Таблиця 2.1 - Основні особливості етапів процесу управління обліком проведення тестування в IT-компанії

Етапи	Особливості	Програмні інструменти
Планування	Визначення тестів, призначення виконавців, створення плану	TestRail, JIRA
Виконання та відстеження	Виконання тестів, запис результатів, відстеження прогресу	BrowserStack, TestRail
Управління дефектами	Документування дефектів, відстеження до вирішення	JIRA з Xray, TestRail
Звітність	Генерація звітів про покриття тестів, результати	TestRail, BrowserStack
Архівування	Зберігання записів для майбутнього використання, забезпечення відповідності	Центральні сховища, хмарні рішення

Процес обліку проведення тестування в IT-компанії починається з планування, де визначають, які тести проводити, коли та хто їх виконуватиме (рис.

2.1). Це допомагає організувати роботу команди та забезпечити покриття всіх аспектів програмного забезпечення. Під час виконання тестів фіксують, хто, коли та з яким результатом (успішно чи невдало) виконав кожен тест. Це дозволяє відстежувати прогрес і швидко реагувати на проблеми. Якщо виявлено дефекти, їх документують, включаючи кроки для відтворення та серйозність проблеми, а потім відстежують до їх вирішення. Звіти про покриття тестів та результати допомагають інформувати зацікавлених сторін. Усі записи зберігають у центральному сховищі для майбутнього використання, особливо важливо для аудитів. У регульованих галузях, як-от фінанси, можуть діяти суворіші правила щодо збереження даних. Розглянемо процес обліку проведення тестування в IT-компанії більш детально.

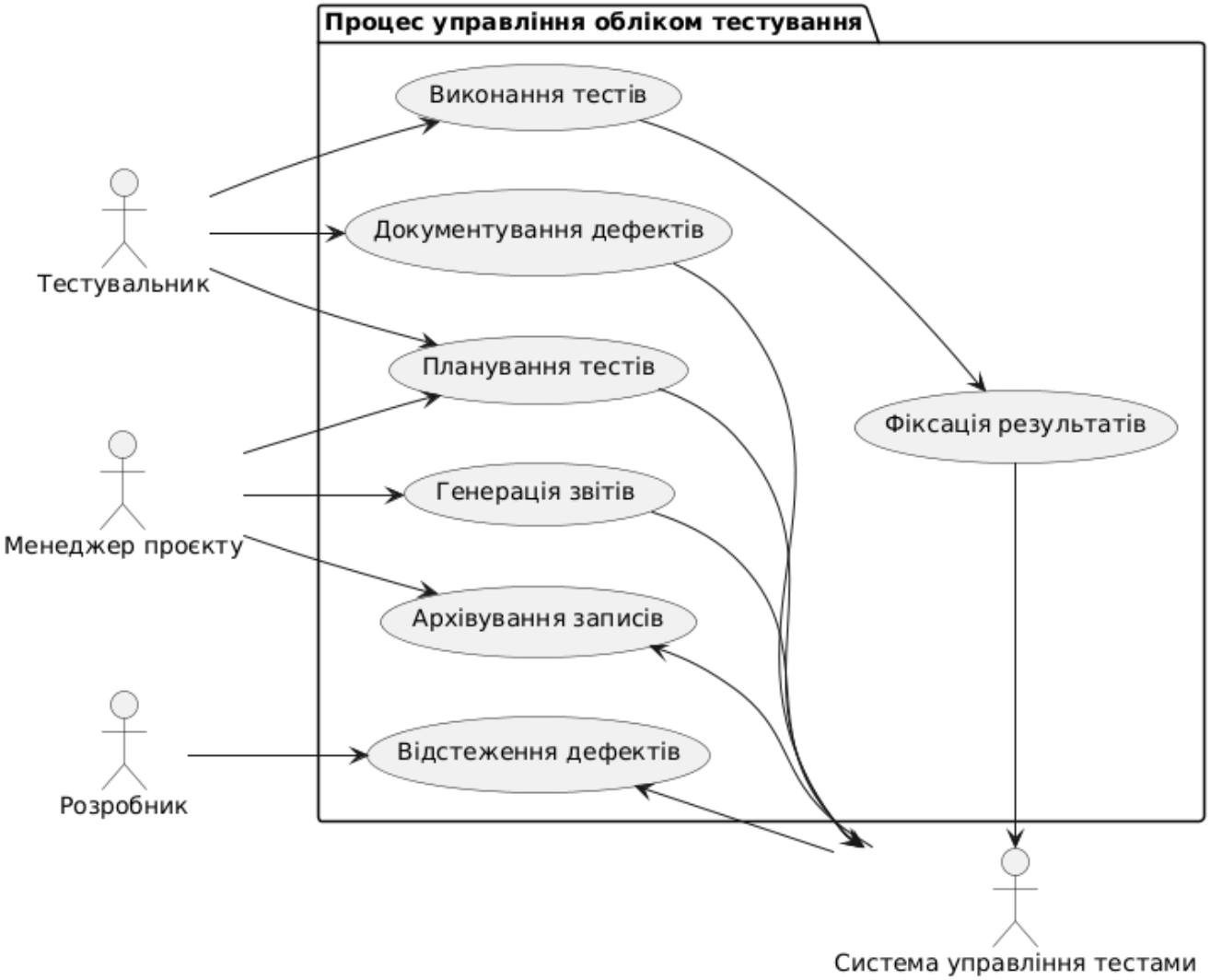

Рисунок 2.1 - UML діаграма використання процесу обліку проведення тестування в IT-компанії

Перший етап процесу обліку проведення тестування в IT-компанії передбачає визначення, які тести потрібно провести, коли та хто їх виконуватиме. Це включає створення плану тестування, який визначає обсяг, цілі та ресурси. Наприклад, компанії можуть використовувати інструменти, такі як TestRail (Test Case Management & Orchestration Software by TestRail), для організації тест-кейсів і призначення їх виконавцям. Планування допомагає забезпечити покриття всіх аспектів програмного забезпечення та оптимізувати використання часу команди.

У комерційних секторах економіки, таких як охорона здоров'я чи фінанси, планування може включати додаткові вимоги, наприклад, дотримання стандартів розвинутих країн, таких як HIPAA для захисту даних (Software Testing In Regulated Industries - TestRail). Це забезпечує, що тести відповідають юридичним і галузевим нормам.

Отже, на першому етапі процесу обліку проведення тестування в IT-компанії необхідні тести виконуються в повному обсязі, і їх результати фіксуються в системі. Кожен тест-кейс документується з інформацією про виконавця, дату та час виконання, а також результат (успішно чи невдало). Наприклад, інструменти, такі як BrowserStack дозволяють записувати результати тестів і відстежувати їх у реальному часі.

Таке відстеження у часі включає запис статусу кожного тесту, що дозволяє командам швидко ідентифікувати проблеми та коригувати плани. Це особливо важливо для великих проектів, де одночасно виконується багато тестів.

Якщо під час тестування виявлено дефекти, їх детально документують, включаючи кроки для відтворення, опис проблеми та її серйозність. Ці дані зазвичай інтегруються з системами відстеження дефектів, такими як JIRA з додатком Xray. Дефекти відстежуються до їх вирішення, що дозволяє командам розвитку швидко реагувати на проблеми. Наприклад, якщо тест не пройдено, дефект може бути автоматично пов'язаний із завданням для розробників. Це забезпечує прозорість і ефективність у процесі виправлення помилок.

Після виконання тестів створюються звіти, які підсумовують покриття тестів, відсоток успішних і невдалих тестів, а також інші ключові метрики. Ці звіти важливі для інформування зацікавлених сторін, таких як менеджери проектів чи клієнти, про стан якості програмного забезпечення.

Програмні інструменти, такі як TestRail, дозволяють генерувати звіти з деталями про виконання тестів у різних версіях і середовищах, як зазначено в Test Case Management & Orchestration Software by TestRail. Це допомагає приймати обґрунтовані рішення щодо випуску продукту.

Після цього, всі записи про тести зберігаються в центральному сховищі для майбутнього використання, зокрема для аудитів або повторного тестування. Це особливо важливо в регульованих галузях, де можуть діяти суворі вимоги до збереження даних, наприклад, у фінансовому секторі, де потрібно дотримуватися стандартів, таких як SOX (Retention of Records Relevant to Audits and Reviews).

Етап архівування забезпечує, що історичні дані доступні для аналізу трендів або для відповідності нормативним вимогам, як зазначено в документах Records Management Code of Practice - NHS Transformation Directorate.

Важливими під час реалізації цих етапів є програмна інструментальна підтримка. Тому, IT-компанії часто використовують спеціалізовані інструменти для управління тестами, такі як:

- TestRail, який дозволяє організовувати тест-кейси, записувати результати та інтегруватися з іншими інструментами, такими як JIRA (Test Case Management & Orchestration Software by TestRail);

- BrowserStack, який надає платформу для запису результатів тестів і відстеження їх виконання;

- JIRA з Xray, який дозволяє керувати тестами як завданнями в JIRA, що полегшує інтеграцію з процесами розробки.

Ці інструменти забезпечують централізоване управління, що зменшує ризик втрати даних і покращує співпрацю в команді.

Особливості це стосується комерційних галузей, таких як охорона здоров’я та IT-кампанії, де процес управління обліком тестування може включати додаткові

вимоги. Наприклад, у медичній сфері тести повинні відповідати стандартам, таким як HIPAA, що вимагає детального логування дій системи для забезпечення конфіденційності даних. В IT секторі можуть діяти вимоги до аудиторських записів, як зазначено в SEC.gov | Retention of Records Relevant to Audits and Reviews. Тому, такі вимоги забезпечують, що записи про тести є детальними, доступними для перевірки та відповідають юридичним нормам.

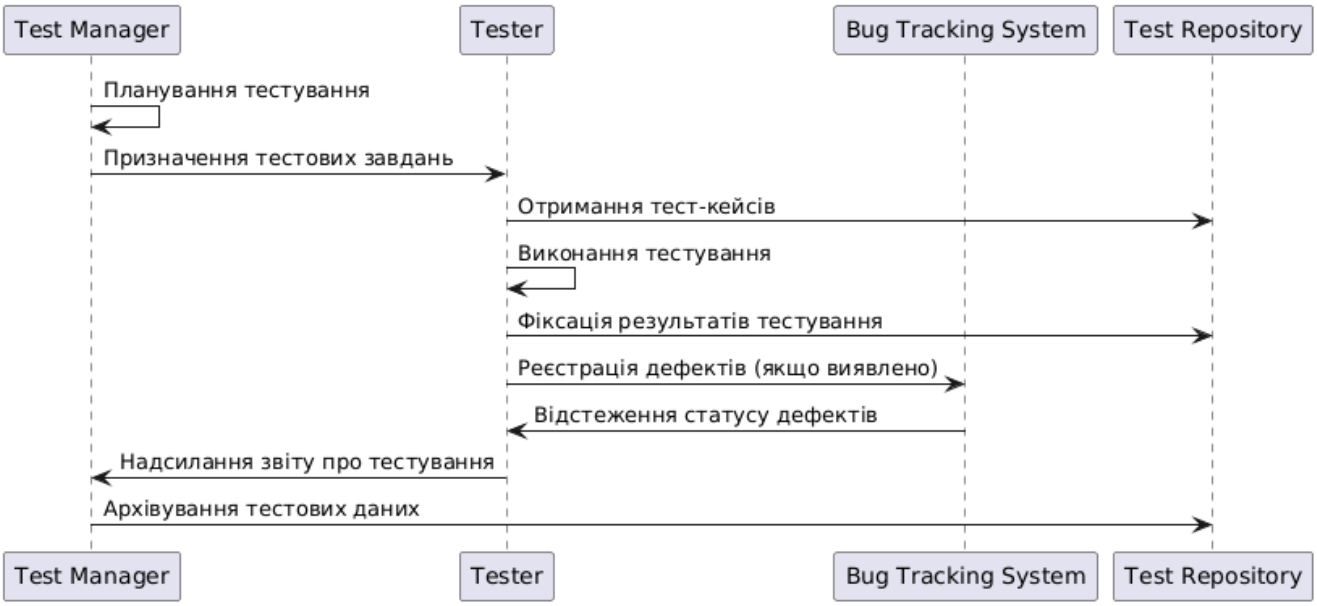

Рисунок 2.2 - UML діаграма послідовності процесу обліку проведення тестування в IT-кампанії

Процес управління обліком тестування в IT-компанії починається з планування, де менеджер проєкту визначає, які тести проводити, коли та хто їх виконуватиме. Це допомагає організувати роботу команди та забезпечити покриття всіх аспектів програмного забезпечення. Так, для створення діаграми послідовності (рис. 2.2) було визначено типовий сценарій для одного тестового випадку, який спочатку не пройшов, а потім був виправлений. Нижче наведено детальний опис кроків, які відображають взаємодії між учасниками наступним чином.

Планування та призначення тестів виконує менеджер проєкту який взаємодіє з TMS, щоб призначити тест тестувальнику. Наприклад, повідомлення: pm -> tms: Plan tests (assign tests to testers). Тестувальник отримує план тестування, взаємодіючи з TMS: tester -> tms: Get assigned tests.

Після цього, тестувальник виконує тест, що є дією, яка не потребує прямого повідомлення до TMS, тому відображається як: note right of tester: Execute test.

Далі тестувальник записує результат у TMS. У нашому сценарії (рис. 2.2) результат — "невдало": tester -> tms: Record result (fail). При цьому виконується обробка дефектів. Якщо тест не пройдено, тестувальник створює дефект у TMS, вказуючи деталі, такі як кроки для відтворення: tester -> tms: Create defect.

TMS автоматично або вручну призначає дефект розробнику: tms -> developer: Assign defect. Розробник виправляє дефект, що є його дією: note right of developer: Fix defect. Після виправлення розробник оновлює статус дефекту в TMS: developer -> tms: Mark defect as fixed. TMS повідомляє тестувальника, що дефект виправлено: tms -> tester: Notify defect fixed.

Для повторного виконання та оновлення результатів тестувальник знову виконує тест: note right of tester: Re-execute test. Потім записує новий результат у TMS, наприклад, "успішно": tester -> tms: Record result (pass).

Етап звітність та архівування починається, коли менеджер проєкту генерує звіти з TMS для інформування зацікавлених сторін: pm -> tms: Generate reports.

Нарешті, менеджер проєкту ініціює архівування всіх записів у TMS для майбутнього використання, особливо для аудитів: pm -> tms: Archive records.

Розглянемо ще іншу подію, де під час виконання тестів тестувальник фіксує, хто, коли та з яким результатом (успішно чи невдало) виконав кожен тест, що дозволяє відстежувати прогрес і швидко реагувати на проблеми (рис. 2.3). Якщо виявлено дефекти, їх документують із деталями, такими як кроки для відтворення та серйозність, а потім відстежують до вирішення розробником. Звіти про покриття тестів та результати допомагають інформувати зацікавлених сторін, а всі записи зберігають у центральному сховищі для майбутнього використання, особливо для аудитів. У комерційних галузях, таких як IT-кампанії, можуть діяти суворіші правила щодо збереження даних. Тому, розглянемо послідовність дій під час виконанння процесу управління обліком проведення тестування в IT-кампанії, де існує типова послідовність для одного тестового випадку, який спочатку не пройшов, а потім виправився.

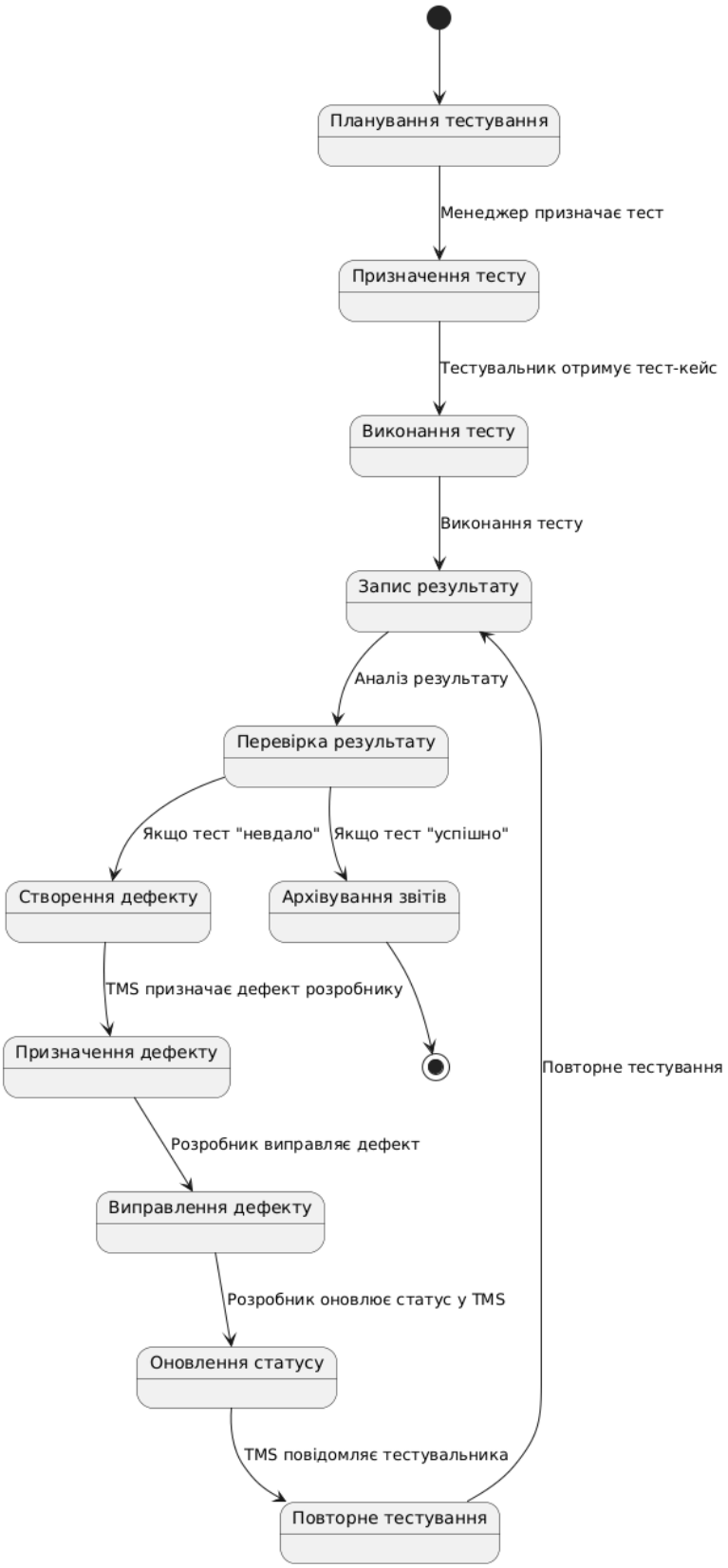

Рисунок 2.3 - Алгоритм процесу управління обліком проведення тестування в IT-кампанії

Для цього, менеджер проєкту призначає тест тестувальнику через систему управління тестами (TMS). Тестувальник отримує план тестування з TMS, виконує тест і записує результат (наприклад, "невдало"). Якщо тест не пройдено, тестувальник створює дефект у TMS, вказуючи деталі, такі як кроки для відтворення. Далі, TMS призначає виправлення дефекту розробнику.

Розробник виправляє дефект і оновлює статус у TMS, позначаючи його як "виправлено". TMS повідомляє тестувальника, що дефект виправлено, і тестувальник повторно виконує тест.

Тестувальник записує новий результат (наприклад, "успішно") у TMS. Потім, менеджер проєкту генерує звіти з TMS для інформування зацікавлених сторін. Для цього, менеджер проєкту ініціює архівування всіх записів у TMS для майбутнього використання, особливо для аудитів. Цей процес забезпечує прозорість і контроль якості, особливо в регульованих галузях, де важлива детальна документація.

Таким чином, процес управління обліком проведення тестування в IT-компанії є комплексним і включає планування, виконання, відстеження, звітність та архівування. Використання спеціалізованих інструментів, таких як TestRail чи BrowserStack, значно полегшує цей процес, забезпечуючи ефективність і точність. У комерційних галузях додаткові вимоги до документації підкреслюють важливість детального логування та відповідності стандартам. Цей підхід дозволяє IT-кампаніям забезпечити високу якість програмного забезпечення, мінімізувати ризики та відповідати очікуванням зацікавлених сторін.

2.2 Розробка функціональних вимог до програмного засобу

Для розробки програмного засобу з управління обліком проведення тестування в IT-кампанії створимо опис функціональних вимог за певними категоріями (рис. 2.4). Основною вимогою до розробки програмного засобу з управління обліком проведення тестування в IT-кампанії є те, що система повинна надавати можливості управління тестовими сценаріями, створення звітів про проведене тестування, відстеження знайдених дефектів, підтримки різних типів тестування та розмежування доступу для різних ролей користувачів

(тестувальників і розробників). На даному етапі розробки інтеграція з іншими інструментами не передбачається – система функціонує автономно як самостійний продукт. Розглянемо такі категорії більш подробно.

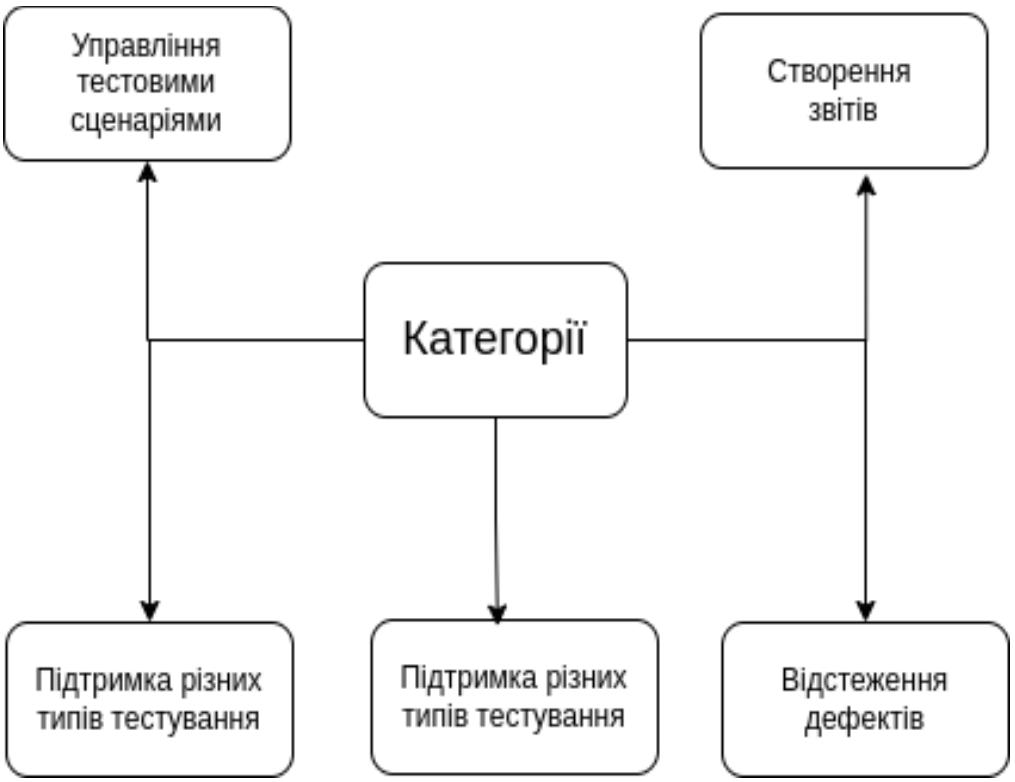

Рисунок 2.4 - Основні категорії вимог для створення програмного засобу з управління обліком проведення тестування в IT-кампанії

Для управління тестовими сценаріями система повинна забезпечувати повний життєвий цикл роботи з тестовими сценаріями (тест-кейсами), включаючи їх створення, редагування, виконання та збереження результатів. Основні вимоги в цій області є наступними:

- перегляд та пошук сценаріїв, де є надання списку всіх наявних тестових сценаріїв з можливістю їх перегляду. Система повинна підтримувати пошук за назвою або ключовими словами та фільтрацію сценаріїв за певними критеріями (наприклад, за тегами або типом тестування), щоб користувачі легко знаходили потрібний сценарій;

- редагування сценаріїв, де є можливість вносити зміни в існуючі тестові сценарії. При зміні вимог або виявленні неточностей тестувальник повинен мати

змогу оновити опис сценарію, кроки тестування, очікувані результати тощо. Всі зміни мають зберігатися в системі;

- видалення/архівування сценаріїв, де є можливість видаляти більше не потрібні або застарілі тестові сценарії. Видалення повинно виконуватися з підтвердженням, щоб уникнути випадкової втрати даних. Альтернативно, може бути передбачене архівування сценаріїв – переміщення в архів, щоб зберегти історію без відображення в актуальному списку тестів;

- виконання тестових сценаріїв, де є засоби для фіксації факту виконання тестів за сценаріями. Тестувальник повинен мати інтерфейс для відмітки результату виконання кожного тестового сценарію (або навіть окремих кроків) – позначення як пройденого (успішного) чи проваленого. При відмітці провалу тестувальник може додати примітку з описом фактичного результату і, за необхідності, швидко зареєструвати дефект;

- збереження результатів, де система зберігає результати кожного виконання тестового сценарію. Для кожного тестового запуску повинна фіксуватися інформація, де є статус проходження (успіх/неуспіх), фактичний результат або отримані дані, коментарі тестувальника, дата і час виконання та користувач, який проводив тест. Ці дані мають бути доступні для перегляду, щоб відстежувати історію проходжень кожного сценарію (скільки разів і коли він виконувався, скільки разів провалювався тощо).

Система повинна надавати засоби для генерування звітів про виконане тестування та аналізу отриманих результатів. Ключові вимоги щодо звітності є наступними:

- зведені звіти про тестування, де є можливість формувати загальний звіт за вибраний період, реліз або тестовий цикл. У зведеному звіті відображаються такі основні показники як загальна кількість виконаних тестових сценаріїв, кількість успішно пройдених і провалених тестів (з процентним співвідношенням), кількість виявлених дефектів за цей період. Це дає змогу керівництву швидко оцінити якість продукту на певний момент;

- деталізовані звіти, де є можливість отримати деталізовану інформацію про результати тестування. Такий звіт може містити перелік усіх тестових сценаріїв, що виконувались, із зазначенням результату для кожного (пройшов/не пройшов), а також деталями по кожному проваленому тесту (наприклад, ідентифікатори зареєстрованих дефектів, пов’язаних із цим тестом). Деталізований звіт дозволяє тестувальникам і розробникам проаналізувати конкретні збої;

- аналіз результатів тестування, де є надання аналітичних даних для глибшого аналізу якості. Система повинна підтримувати побудову діаграм або графіків на основі даних тестування – наприклад, тренд проходження тестів (як змінюється відсоток успішних тестів від збірки до збірки), розподіл знайдених дефектів за пріоритетами або за модулями продукту, середній час виконання тестового сценарію тощо. Така аналітика допоможе виявити проблемні місця та оцінити прогрес тестування;

- налаштування параметрів звіту, де користувачі повинні мати можливість налаштувати критерії генерації звіту. Наприклад, обрати діапазон дат, за який формувати звіт, обрати конкретний проект або модуль продукту, вказати відповідального тестувальника тощо. Це дозволить отримувати як високорівневі звіти, так і вузьконаправлені (наприклад, звіт по тестуванню окремого модуля за останній тиждень);

- експорт та збереження звітів, де система повинна дозволяти зберігати сформовані звіти у зручних форматах (наприклад, PDF, XLS/CSV) для обміну або документування. Також має бути можливість друку звіту безпосередньо з інтерфейсу. Експорт забезпечить використання даних звіту поза межами системи (наприклад, для презентацій або додаткового аналізу).

Система повинна виконувати відстеження дефектів та включати функціонал баг-трекінгу для реєстрації та моніторингу дефектів (помилок), виявлених під час тестування. Вимоги щодо відстеження дефектів є наступними:

- реєстрація дефектів, де є можливість створювати запис про новий дефект безпосередньо під час або після виконання тесту. При реєстрації дефекту повинні зазначатися такі ключові дані як заголовок (короткий опис проблеми), детальний

опис (симптоми, кроки відтворення, очікувана поведінка vs. фактична), середовище чи конфігурація, в якій виявлено дефект (версія ОС, браузер, тощо, якщо актуально). Система повинна дозволяти прикріплювати до дефекту додаткові матеріали – скриншоти, файли журналів (логи) або інші докази, що підтверджують наявність помилки;

- визначення статусу та пріоритету дефекту, де кожен зареєстрований дефект має атрибути статус і пріоритет. Статус відображає поточний етап життєвого циклу дефекту (наприклад: "Новий", "В роботі", "Виправлено", "На перевірці", "Закрито"), і система повинна підтримувати зміну статусу відповідно до прогресу роботи над дефектом. Пріоритет визначає важливість або терміновість виправлення дефекту (наприклад: низький, середній, високий, критичний) – його встановлює тестувальник при реєстрації або змінює відповідальна особа. Система повинна відображати ці атрибути для кожного дефекту і дозволяти їх оновлення уповноваженим користувачам;

- оновлення та коментування дефектів, де після реєстрації дефект повинен підтримувати можливість подальшого редагування та додавання коментарів. Тестувальники можуть уточнювати опис або змінювати пріоритет, якщо з’ясувалися нові обставини. Розробники можуть залишати коментарі щодо проведеного аналізу чи виправлення (наприклад, зазначати причину помилки або модуль, у якому внесено зміни). Система повинна фіксувати, хто і коли вносив зміни чи додавав коментар, забезпечуючи тим самим ефективну взаємодію між командами;

- визначення історії змін, де для кожного дефекту має зберігатися повна історія його життєвого циклу. Система повинна автоматично журналювати всі важливі події, пов’язані з дефектом: зміну статусу (з зазначенням нового значення і хто змінив), зміну пріоритету, правки в описі, додання коментарів, переназначення відповідального тощо. Цей журнал змін повинен бути доступним для перегляду, щоб будь-який користувач міг простежити, як просувалось вирішення проблеми з моменту її виявлення до закриття;

- виконання пошуку і фільтрації дефектів, де система повинна забезпечувати зручний спосіб знайти потрібні дефекти серед зареєстрованих. Необхідні можливості включають пошук за унікальним ідентифікатором дефекту або ключовими словами в описі, а також фільтрацію списку дефектів за різними полями. Наприклад, користувач може відфільтрувати всі "відкриті" дефекти високого пріоритету, або всі дефекти, призначені конкретному розробнику, або всі дефекти зі статусом "Виправлено", що очікують повторного тестування. Така функціональність допоможе керувати багатьма повідомленнями про помилки в масштабних проектах;

- виконання зв’язоку з тестами, де система повинна забезпечувати можливість зв’язувати дефекти з тестовими сценаріями, в ході виконання яких ці дефекти були виявлені. При реєстрації дефекту тестувальник може вказати, під час виконання якого тестового сценарію сталася помилка. Це створює трасування: переглядаючи дефект, користувач зможе побачити, яке тестування його виявило; навпаки, при перегляді результатів проваленого тестового сценарію можна побачити посилання на відповідні зареєстровані дефекти. Такий зв’язок полегшує аналіз причин невдач тестів і контроль виправлення помилок.

Система повинна підтримувати різні види тестування, щоб охопити як ручні, так і автоматизовані перевірки, а також спеціалізовані сценарії. Кожен тестовий сценарій при створенні або редагуванні може бути позначений певним типом тестування – це дозволить врахувати специфіку його виконання та подальшого аналізу результатів. Підтримуються такі типи тестування.

Функціональне тестування, де є перевірка функцій та вимог продукту. Система підтримує створення і виконання функціональних тестових сценаріїв з детальним описом кроків і очікуваних результатів. Результати функціональних тестів (пройдено/не пройдено, знайдені дефекти) фіксуються для кожного сценарію, що дозволяє оцінити відповідність фактичної поведінки заявленим вимогам.

Автоматизоване тестування, де є виконання тестів за допомогою скриптів або автоматичних інструментів. Система дозволяє відзначати тестовий сценарій як

автоматизований. Для таких сценаріїв передбачено збереження результатів автоматичного запуску. При цьому, тестувальник може додати до системи результати роботи автотестів. Наприклад, завантажити файл зі звітом тестування або вручну зазначити підсумок проходження. Також можливо вказати посилання на автоматизований тест-скрипт або тестове середовище, щоб було зрозуміло, який саме скрипт відповідає цьому сценарію.

Регресійне тестування, де є повторне тестування системи після внесення змін, з метою переконатися у відсутності нових дефектів у вже перевірених частинах. Система підтримує виділення регресійних тестових сценаріїв. Наприклад, тестувальник може позначити певні сценарії як такі, що входять до регресійного набору. Можливо групувати регресійні тести в окремі набори (суцільні тест-плани) для їх швидкого повторного запуску перед релізами. Результати регресійних тестів слід відстежувати окремо, щоб можна було порівнювати їх успішність між релізами.

Навантажувальне тестування, де є перевірка продуктивності та стабільності системи під значним навантаженням (велика кількість користувачів, запитів тощо). Система повинна дозволяти фіксувати результати навантажувальних тестів. Для сценаріїв цього типу, окрім відмітки "пройдено/не пройдено", можуть зберігатися такі метрики, як кількість одночасних користувачів під час тесту, час відгуку системи, пропускна здатність та інші показники продуктивності, отримані під час тестування. Це дозволить аналізувати, чи відповідають ці показники вимогам до продуктивності.

Фільтрація та звітність за типом відбувається внаслідок того, що система повинна надавати можливість фільтрувати тестові сценарії та результати тестування за типом. Користувач може, наприклад, відобразити лише автоматизовані тести або лише навантажувальні тести в списку сценаріїв. Так само, при генеруванні звітів, можна сформувати звіт по конкретному виду тестування (скажімо, окремо про результати навантажувального тестування). Звіти повинні відображати розподіл успішності тестів за типами, щоб керівники проекту бачили, як просувається кожен напрямок тестування (функціональне, регресійне тощо). Це

допоможе зрозуміти, чи всі аспекти якості програмного забезпечення перевіряються належним чином.

В системі також повинні бути передбачені ролі користувачів з різними рівнями доступу до функцій, мінімально – роль тестувальника та роль розробника. Кожна роль матиме свої права з метою забезпечення контролю доступу: тестувальники керують тестуванням, а розробники в основному працюють з дефектами. Вимоги щодо ролей є наступними:

- тестувальники - це користувачі з цією роллю мають повний доступ до функцій системи, пов’язаних з процесом тестування. Тестувальники можуть створювати і редагувати тестові сценарії, виконувати тести та фіксувати результати, реєструвати нові дефекти та оновлювати інформацію по них (включно зі зміною статусу і пріоритету), а також генерувати звіти про тестування. Тестувальники, як правило, відповідають за перевірку виправлених дефектів і закриття дефекту після того, як виправлення підтверджене повторним тестом;

- розробники - це користувачі, що мають обмежений доступ, необхідний для роботи з дефектами і перегляду результатів тестування. Розробники можуть переглядати тестові сценарії (щоб ознайомитися з тим, що перевіряється) і переглядати звіти про тестування, щоб розуміти загальну картину якості та які помилки були знайдені. Основна функція розробників у системі – опрацювання дефектів: вони можуть змінювати статус дефекту (наприклад, встановити статус "Виправлено" після внесення змін у код), додавати коментарі щодо суті виправлення або способу вирішення проблеми. При цьому розробники не мають права створювати чи редагувати тестові сценарії, видаляти будь-які дані тестування або генерувати/редагувати звіти – ці дії віднесені до компетенції тестувальників.

Вимоги щодо безпека та розмежування доступу полягають в тому, що система повинна забезпечувати авторизацію користувачів і розмежування прав доступу відповідно до призначеної ролі. Кожен користувач входить в систему під своїми обліковими даними (логіном), і на основі його ролі йому доступний тільки визначений набір функцій. Дані, що стосуються тестових сценаріїв, результатів тестування та дефектів, повинні бути захищені від несанкціонованого втручання.

Наприклад, розробник може лише переглядати інформацію про тести і змінювати статуси дефектів, які йому призначені, але не має можливості редагувати самі сценарії тестів чи видаляти записи про тестування. Це гарантує цілісність бази знань тестування і запобігає випадковим або навмисним змінам критичної інформації невповноваженими особами.

2.3 Проектування основних модулів програмного засобу

Для створення програмного засобу з управління обліком проведення тестування в ІТ-компанії необхідно визначити функціональність основних та додаткових модулів (рис. 2.5). Така система призначена для організації процесу тестування програмного забезпечення, включаючи планування тестів, призначення завдань, відстеження прогресу та звітування про результати. Наведено функціональність основних модулів програмного засобу з управління обліком проведення тестування в ІТ-компанії.

Рисунок 2.5 - Основні модулі програмного засобу з управління обліком проведення тестування в ІТ-компанії

Ці модулі є базовими компонентами системи, які забезпечують ключові аспекти управління тестуванням.

Модуль управління тест-кейсами є ключовим компонентом системи, що забезпечує організацію, виконання та відстеження тест-кейсів, необхідних для забезпечення якості програмного забезпечення. Модуль дозволяє створювати нові тест-кейси з полями, такими як ID, назва, опис, кроки, очікувані результати, теги та вкладення. Редагування включає версіонування для відстеження змін та коментарі для співпраці. Видалення може бути м’яким (позначення як неактивного), щоб зберегти історію. Тест-кейси в модулі можна групувати в тест-сюїти для різних функціональних областей або релізів, з можливістю ієрархічної організації (папки в папках) та призначення їх до конкретних проєктів.

Наведено таблицю, що підсумовує основні функції модуля управління тест-кейсами (табл. 2.2).

Таблиця 2.2 - Порівняння ключових функцій модуля управління тест-кейсами

Функція	Опис
Створення тест-кейсів	Заповнення полів, підтримка шаблонів, простота та чіткість.
Редагування та видалення	Версіонування, коментарі, м’яке видалення.
Групування в тест-сюїти	Ієрархічна організація, призначення до проєктів.
Призначення тестувальникам	Повідомлення, відстеження прогресу, оптимізація навантаження.
Відстеження статусу	Різні статуси, оновлення з коментарями, прозорість.
Пошук і фільтрація	Швидкий пошук за критеріями, збереження запитів.
Співпраця	Коментарі, вкладення, сповіщення.
Інтеграція	З дефект-трекінгом, версією, CI/CD.
Звітність	Звіти про покриття, статус, метрики.

Модуль управління тест-кейсами також дозволяє призначати тест-кейси або тест-сюїти одному чи кільком тестувальникам, з повідомленнями про призначення та відстеженням прогресу. Статус тест-кейсів може бути "Не розпочато", "В

процесі", "Пройдено", "Не пройдено" або "Заблоковано". Тестувальники можуть оновлювати статус, додаючи коментарі чи вкладення (табл. 2.3).

Створення тест-кейсів виконуються наступним чином. Користувачі можуть створювати нові тест-кейси через інтуїтивний інтерфейс, заповнюючи такі поля: унікальний ID (що автоматично генерується або задається користувачем), назва та детальний опис, кроки для виконання тесту, очікувані результати, теги або категорії для класифікації, посилання на вимоги або користувацькі історії дій, вкладення, такі як знімки екрану чи документи.

Таблиця 2.3 - Порівняння ключових функцій модуля планування тестування

Функція	Опис
Створення планів тестування	Визначення цілей, обсягу, стратегії, графіку, ресурсів та ризиків.
Визначення обсягу тестування	Вибір тест-кейсів на основі критеріїв, додавання до плану.
Встановлення термінів	Призначення дедлайнів для тест-кейсів.
Встановлення пріоритетів	Призначення рівнів важливості (високий, середній, низький).
Призначення тестувальників	Розподіл тест-кейсів між тестувальниками.
Планування середовища	Визначення вимог до обладнання, програмного забезпечення.
Управління ризиками	Ідентифікація ризиків та розробка планів мінімізації.
Пошук і фільтрація	Швидкий пошук планів за критеріями, збереження запитів.
Співпраця	Коментарі, вкладення, сповіщення.
Інтеграція	З дефект-трекінгом, версією, CI/CD.
Звітність	Звіти про статус плану, виконаних тест-кейсів.

Модуль управління тест-кейсами підтримує створення шаблонів для різних типів тест-кейсів (функціональні, продуктивності, безпеки), що стандартизує формат і прискорює процес створення. Тест-кейси мають бути простими, чіткими та написаними з урахуванням сценаріїв використання кінцевим користувачем, як рекомендують Guide to Test Case Management in Agile.

Редагування та видалення тест-кейсів виконується за допомогою програмних інтерфейсів. Так, користувачі можуть змінювати будь-яке поле існуючого тест-кейсу. Кожне редагування створює нову версію для відстеження змін, що важливо для історичних записів. Коментарі дозволяють пояснити зміни для співпраці в команді.

Тест-кейси в системі можна видаляти, але система підтримує м’яке видалення, позначаючи їх як неактивні, щоб уникнути втрати даних, які можуть знадобитися пізніше. Це відповідає рекомендаціям щодо збереження повноти репозиторію, як зазначено в Practical Guide to Creating and Managing Effective Test Cases. Доступ до редагування та видалення регулюється на основі ролей, забезпечуючи контроль доступу.

Тест-кейси можна групувати в тест-сюїти для різних функціональних областей або релізів. Система підтримує ієрархічну структуру (папки в папках), що полегшує навігацію. Тест-сюїти можна асоціювати з конкретними проєктами або релізами, що допомагає в плануванні тестування. Наприклад, усі тест-кейси для нового релізу можуть бути зібрані в одному наборі. Така організація базується на логіці, наприклад, за пріоритетом, типом тестування чи певним модулем.

Тест-кейси або тест-сюїти можна призначати одному чи кільком тестувальникам. Система надсилає повідомлення через електронну пошту або в межах програми, що забезпечує синхронізацію. Користувачі можуть переглядати призначені їм тест-кейси та їхній статус, що сприяє ефективному управлінню ресурсами.

Система підтримує такі статуси: "Не розпочато", "В процесі", "Пройдено", "Не пройдено", "Заблоковано". Тестувальники можуть оновлювати статус під час виконання, додаючи коментарі чи вкладення для деталізації. Оновлення статусу

автоматично відображається в звітах, що дозволяє відстежувати прогрес. Регулярне оновлення статусу забезпечує прозорість і допомагає ідентифікувати проблемні області.

Існують також додаткові функції та інтеграції, які полягають у пошуку та фільтрації. Користувачі можуть шукати тест-кейси за критеріями, такими як ID, назва, теги, призначений тестувальник, статус чи дата створення/оновлення, з можливістю збереження запитів. Ще підтримується коментування тест-кейсів, додавання вкладень і сповіщення про зміни, що покращує командну роботу.

Модуль інтегрується з системами відстеження дефектів, контролем версій (наприклад, Git) і CI/CD-пайплайнами для автоматизації. Для надання звітності генеруються звіти про покриття, статус виконання та інші метрики, що допомагає в аналізі.

Таким чином, модуль управління тест-кейсами забезпечує комплексне управління тест-кейсами, відповідаючи потребам IT-компаній у забезпеченні якості програмного забезпечення. Він враховує кращі практики, такі як простота, повторне використання, автоматизація та інтеграція, що підтверджується аналізом провідних інструментів, таких як TestRail, Qase, TestLodge та інших.

Модуль планування тестування для програмного забезпечення, призначений для управління обліком тестування в IT-компанії. Модуль є ключовим компонентом системи, що забезпечує організацію та планування процесу тестування для окремих релізів або ітерацій, визначаючи обсяг, терміни та пріоритети тестів.

Модуль дозволяє користувачам створювати детальні плани тестування, які є основою для успішного тестування програмного забезпечення. Кожен тестовий план асоціюється з конкретним релізом або ітерацією та включає такі елементи:

- назва та опис, що використовується для ідентифікації та надання огляду плану;

- цілі, що являють собою чітко визначені цілі, яких тестування має досягти, наприклад, перевірка функціональності або продуктивності;

- обсяг, що використовується для визначення, які функції та модулі будуть тестуватися, а які виключені. Це допомагає встановити межі тестування;

- стратегія, що являє собою підхід до тестування, включаючи типи тестів (функціональні, продуктивності, безпеки тощо);

- очікувані результати, що очікується отримати від процесу тестування, наприклад, звіти про тести чи списки дефектів;

- графік, де використовуються терміни та ключові етапи тестування, включаючи дати початку та закінчення;

- ресурси, де існують вимоги до тестувальників, інструментів та тестувального середовища;

- ризики та заходи, де існують потенційні проблеми та плани їх мінімізації.

Користувачі також можуть використовувати попередньо визначені шаблони для створення тест-планів, що забезпечує послідовність і повноту предоставленої тестової інформації.

Визначення обсягу тестування є критичним етапом, який визначає, які тест-кейси будуть виконані в рамках плану. Така функціональність включає:

- вибір тест-кейсів, де користувачі можуть шукати та вибирати тест-кейси на основі критеріїв, таких як функціональна область, пріоритет або теги;

- асоціацію з планом, де є додавання обраних тест-кейсів до тест-плану для забезпечення їх виконання;

- керування списком, де є перегляд і управління списком тест-кейсів, пов’язаних із планом, з можливістю виключення або позначення як опціональних.

Ця функція забезпечує, що лише релевантні тест-кейси включені в обсяг тестування для певного релізу або ітерації.

Для ефективного управління процесом тестування модуль планування тестування дозволяє встановлювати:

- дедлайни, де є конкретні дати, до яких кожен тест-кейс або група тест-кейсів має бути завершена;

- пріоритети, де є рівні важливості для кожного тест-кейсу, такі як високий, середній, низький, або кастомні рівні, що допомагає фокусуватися на критичних тестах.

При цьому, користувачі можуть сортувати та фільтрувати тест-кейси на основі дедлайнів і пріоритетів для оптимізації планування.

Функціональність модуля планування тестування включає:

- призначення тестувальників, де користувачі можуть призначати тестувальників до окремих тест-кейсів або тест-сюїтів у рамках плану;

- відстеження призначень, де відстеження призначень для забезпечення збалансованого навантаження та чітких відповідальностей. Це допомагає планувати, хто виконуватиме які тести, хоча фактичне виконання та відстеження можуть відбуватися в модулі виконання тестів.

Планування тестувального середовища є важливим для точного та надійного тестування внаслідок наступного:

- визначення вимог, де користувачі можуть вказати необхідне обладнання, програмне забезпечення та конфігурації мережі;

- забезпечення відповідності, де потрібно переконатися, що тестувальне середовище максимально відповідає продуктивному.

Ідентифікація та управління ризиками є також важливими ключовими для прогнозування та мінімізації потенційних проблем, які полягають в наступному:

- документація ризиків, де користувачі можуть документувати можливі ризики, пов’язані з процесом тестування;

- планування ризиків, де для кожного ризику можна розробити плани дій, якщо він виникне;

- пошук і фільтрація завдань, де користувачі можуть шукати плани тестування за критеріями, такими як реліз, дата чи пріоритет, з можливістю збереження запитів;

- виконанння сумісних завдань, де підтримується коментування планів тестування, додавання вкладень і сповіщення про зміни, що покращує командну роботу;

- інтеграція модулів, де вони інтегрується з системами управління дефектами, контролем версій (наприклад, Git) і CI/CD-пайплайнами для автоматизації;

- виконанння звітності, де генеруються звіти про статус тест-плану, наприклад, відсоток виконаних тест-кейсів.

Таким чином, модуль планування тестування забезпечує комплексне планування тестування, відповідаючи потребам IT-компаній у забезпеченні якості програмного забезпечення. Він враховує кращі практики, такі як чітке визначення обсягу, оптимізація ресурсів та управління ризиками, що підтверджується аналізом провідних інструментів, таких як TestRail, BrowserStack, PractiTest та інших.

3 РОЗРОБКА МОДУЛІВ ПРОГРАМНОГО ЗАСОБУ

3.1 Розробка основної графічної частини програмного засобу

Графічний інтерфейс реалізовано як веб-додаток на базі Java Spring, що забезпечує динамічне відображення сторінок та інтерактивність (рис. 3.1). Додаток орієнтований на різні ролі користувачів: розробників, тестувальників, менеджерів з тестування та клієнтів. Кожна роль має відповідні права доступу. Наприклад, тестувальники створюють і виконують тест-кейси, менеджери переглядають статистику і керують користувачами, клієнти можуть переглядати звіти про прогрес тестування тощо.

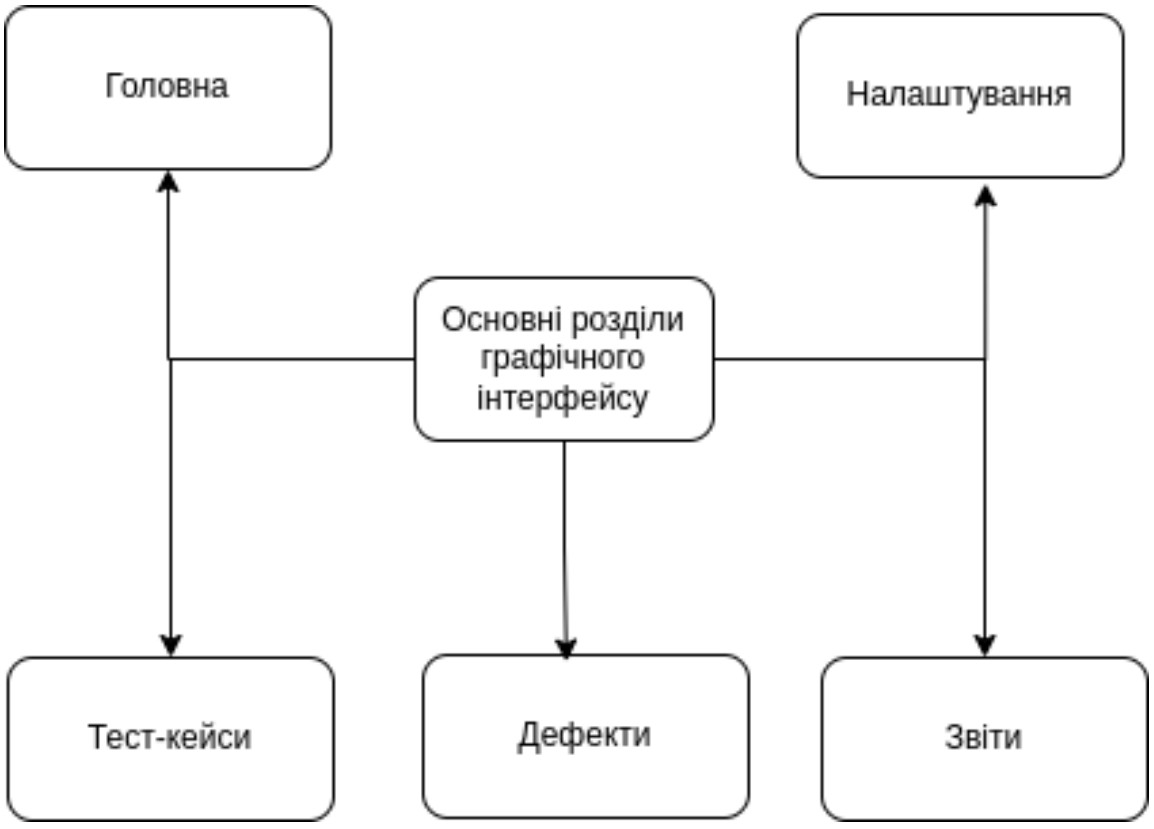

Рисунок 3.1 - Основні розділи графічного інтерфейсу з управління обліком проведення тестування в IT-кампанії

Стиль інтерфейсу для програми з управління обліком проведення тестування в IT-кампанії є мінімалістичним та функціональним. Для цього використовуються стримані кольори, чітка типографіка, інтуїтивно зрозумілі значки й кнопки. Це

забезпечує простоту користування – ключові функції завжди під рукою, зайві елементи відсутні. Навігація здійснюється через верхнє або бічне меню з основними розділами (рис. 3.1).

Кожен розділ має продумане розташування елементів: списки та таблиці вирівняні для зручного читання, кнопки дій виділені (наприклад, контрастним кольором для “Створити” або “Зберегти”), а поля введення та фільтри – на видному місці. Нижче описано ключові екрани та їх функціональність.

Головна панель (Dashboard) призначена для виведення та роботи з початковим екраном після входу, що надає оглядову інформацію про стан тестування і швидкий доступ до основних функцій. На ньому відображаються найважливіші метрики та повідомлення, щоб користувач одразу бачив актуальний статус проекту. Основний вміст і функції головної панелі полягають в наступному:

- загальна статистика, де у верхній частині панелі – блоки з короткими показниками. Наприклад, загальна кількість тест-кейсів, скільки з них пройдено/не пройдено, кількість відкритих дефектів, відсоток успішних тестів тощо). Кожен показник може мати іконку та кольоровий індикатор статусу (зелений для успішних, червоний для проблемних показників), що допомагає швидко оцінити ситуацію;

- графічні віджети, де нижче статистики можуть розміщуватися міні-діаграми – наприклад, стовпчикова діаграма прогресу виконання тестів по днях або спринтах, або кругова діаграма розподілу статусів тест-кейсів (Passed/Failed/Blocked). Ці віджети інтерактивні: наведення курсору може показувати детальні цифри, а клік – вести до сторінки детального звіту;

- підрозділ останніх дій, де є панель може містити розділ зі списком останніх активностей (наприклад, “Тестувальник Іван додав новий тест-кейс X”, “Тест-кейс Y виконано з результатом Fail”, “Створено дефект Z, пріоритет: High”). Це журнал, який допомагає всім користувачам відстежувати поточний прогрес і події;

- швидкі посилання, де на головній панелі також розміщені кнопки або піктограми для швидкого переходу до основних розділів системи. Наприклад, великі кнопки “Тест-кейси”, “Дефекти”, “Звіти” та “Користувачі/Налаштування”.

Таким чином, одразу після входу користувач може одним кліком перейти до потрібного модуля;

- інтерактивність, де головна панель автоматично оновлюється (або оновлюється вручну кнопкою “Оновити”), щоб дані завжди були актуальними. Менеджери і клієнти бачать тут ключову інформацію про виконання тестування, розробники – можуть звернути увагу на нові дефекти. Мінімалістичний дизайн панелі полягає в тому, що відображається лише найнеобхідніше: деталі можна отримати, перейшовши в спеціалізовані розділи. Такий підхід запобігає перевантаженню екрану і забезпечує інтуїтивне сприйняття даних з першого погляду.

- список тестів (Каталог тест-кейсів), де існує підрозділ на екрані для керування тест-кейсами – тут перелічено всі тест-кейси проекту з можливістю пошуку, фільтрації та виконання базових операцій (створення, редагування, видалення). Тестувальники працюють на цьому екрані для організації своїх тестів, а менеджери можуть переглядати структуру тестів.

На зображені екрану “Список тест-кейсів” (рис. 3.2) ліворуч розташовано каталог категорій, зверху поле пошуку та кнопка “+ Create” для додавання нового тесту. У центральній частині відображається таблиця тест-кейсів, права панель призначена для перегляду деталей вибраного тесту. Основний вміст і функції інтерфейсу користувача є наступним:

- панель пошуку та фільтрів, де у верхній частині розміщене поле пошуку (для швидкого знаходження тест-кейсу за назвою або ключовими словами) та кнопка/значок фільтру. Фільтри дозволяють звузити список за різними критеріями. Наприклад, статус тесту, Чернетка, У процесі, Завершено, пріоритет, категорія, відповідальний виконавець тощо. Це особливо корисно, коли тест-кейсів багато;

- дерево категорій (навігація), де зліва знаходиться бічна панель з ієрархією категорій або тестових наборів. Наприклад, тест-кейси згруповані за модулями продукту або типами тестування. Клік на категорію фільтрує список тільки на ті тест-кейси, що до неї належать. Менеджери можуть керувати цією структурою в необхідних налаштуваннях;

- таблиця тест-кейсів, де центральна область показує список тест-кейсів у вигляді таблиці. Для кожного запису вказані основні атрибути: назва тест-кейсу, унікальний ідентифікатор або код, стан/статус. Ннаприклад, новий, на виконанні, завершений, пріоритет (низький/середній/високий – може позначатися кольором або іконкою), остання дата виконання і результат, та відповідальний (хто останнім редагував або кому призначено). Таблиця підтримує сортування за цими колонками, наприклад, можна відсортувати за назвою або пріоритетом);

Рисунок 3.2 - Приклад скріншоту екрану “Список тест-кейсів”

- дії з тест-кейсами, де поруч з кожним записом (або при виборі запису) доступні такі кнопки дій як редагувати, видалити, копіювати (щоб швидко створити схожий тест-кейс). Можливе групове видалення/експорт – через відмітку чекбоксами кількох тестів і вибір масової операції. Також у заголовку екрану присутня кнопка “+ Створити тест-кейс”, яка відкриває форму створення нового тесту;

- права панель (швидкий перегляд), де при виборі конкретного тест-кейсу в списку, праворуч від таблиці може з’являтися панель з короткими деталями цього тесту. В ній відображаються основні поля вибраного тест-кейсу (опис, кроки, очікуваний результат у скороченому вигляді) та, можливо, статистика виконань (скільки разів запускався, скільки раз успішно/неуспішно). З цієї панелі може бути пряма кнопка “Детальніше” для переходу на повну сторінку деталей тесту. Якщо жодного тесту не вибрано, панель може показувати повідомлення типу “Виберіть тест-кейс для перегляду деталей”;

- інтерактивність, де існує список що підтримує AJAX-оновлення при додаванні або зміні тест-кейсів (новий тест одразу з’являється в таблиці). При

наведенні на рядок тесту він підсвічується (щоб зрозуміти, який вибрано). Клік на тест відкриває його деталі. Ролі є обмеженими правами. Наприклад, клієнти бачитимуть цей список у режимі лише для читання – без кнопок редагування/видалення. Таким чином, “Список тестів” слугує зручним реєстром усіх тест-кейсів із засобами навігації та управління ними.

Інший розділ картки тест-кейсу призначений для того, щоб екран показував повну інформацію про конкретний тест-кейс. Він відкривається при виборі тесту зі списку або при створенні/редагуванні тест-кейсу. Тут зосереджені всі атрибути тест-кейсу, історія його виконання, а також пов’язані дефекти. Тестувальники використовують цю сторінку, щоб виконувати тест та фіксувати результати, а розробники – щоб переглядати деталі тестів і пов’язані проблеми. Основний вміст і функції цього розділу є наступним:

- заголовок і дії, де у верхній частині сторінки великим шрифтом виводиться назва тест-кейсу (його унікальний заголовок). Поруч можуть бути піктограми статусу (напр., зелена галочка або червоний хрестик, якщо останнє виконання пройшло або ні) і мітка пріоритету. Також є кнопки “Редагувати” (для переходу в режим редагування полів), “Запустити тест” (якщо передбачено безпосереднє виконання) та “Створити дефект” (цей ярлик дозволяє прямо із картки завести новий баг, якщо тест провалився);

- опис тест-кейсу, де основна секція містить такі текстові поля як короткий опис/мета тесту, передумови (що має бути підготовлено перед виконанням), детальні кроки виконання (кожен крок пронумерований, опис дії та очікуваний результат для нього), та очікуваний результат (що вважається успішним завершенням тесту). Ці дані можуть бути відформатовані для зручності (напр., кроки оформлені списком). Якщо є вкладені файли (скріншоти, специфікації) – вони відображаються як посилання або ескізи в цій секції;

- історія виконання тестових робіт, де під описом виводиться хронологія всіх запусків цього тест-кейсу. Це може бути таблиця або таймлайн, де для кожного запуску зазначено дата і час, хто виконував (ім’я тестувальника), статус результату (Pass/Fail/Blocked тощо), і примітки чи дефекти. Наприклад, запис: “2025-03-01

10:15 – Виконавець: Петренко І. – Fail – Створено дефект DEF-101”. Нові виконання додаються вгорі списку. Це дає повну картину стабільності тесту в часі;

- пов’язані дефекти, де збоку або під історією є блок “Дефекти”, де перелічені всі баги, пов’язані з даним тест-кейсом. Кожен запис містить ідентифікатор дефекту (наприклад DEF-101), короткий опис і статус дефекту (відкритий/у роботі/закритий). Ці записи є посиланнями – клік відкриє деталі дефекту (в зовнішній системі баг-трекінгу або у внутрішньому модулі дефектів, залежно від реалізації). Якщо тест-кейс наразі має відкритий дефект, це може бути відзначено кольором або іконкою біля назви тесту;

- коментарі та обговорення, де за потреби, екран може містити вкладку або секцію для коментарів, де тестувальники чи розробники залишають нотатки щодо тесту (наприклад уточнення кроків, питання, рішення по дефектам). Це сприяє співпраці команд над якістю тест-кейсу.

- інтерактивність, де екран, що містить деталі дозволяє редагування (для тих, хто має права). Наприклад, при натисканні “Редагувати” поля опису та кроків стають доступними для зміни, з’являється кнопка “Зберегти”. При виконанні тесту прямо з цього екрану (натиснення “Запустити тест”) можна відмітити результати кроків і встановити загальний статус (пройдено чи не пройдено), після чого система запропонує зафіксувати результат і (у разі провалу) створити запис дефекту.

Таким чином, всі зміни графічного частини програмного засобу (оновлення опису, новий результат виконання) одразу відображались відповідним користувачам. Наприклад, менеджер бачив новий запис в історії, розробник – новий дефект у списку дефектів. Завдяки такому дизайну, користувач фокусувався на змісті тесту: поля підписані, розділи розділені вкладками або заголовками, щоб легко переключатися між “Опис”, “Історія”, “Дефекти”. Це забезпечувало зручність роботи з тест-кейсом від його створення до аналізу результатів.

3.2 Розробка допоміжної графічної частини програмного засобу

Інший розділ звітів (Статистика тестування) призначений для генерації та перегляду звітів про процес тестування. Він надає візуалізацію статистичних даних

таких як успішність тестів, кількість знайдених дефектів, прогрес виконання плану тестування тощо. Менеджери з тестування використовують цей модуль для аналізу якості, а клієнти – щоб отримати наочне уявлення про статус проекту.

Наприклад, кругова діаграма у розділі “Звіти”, відображає останні результати тестів. Зелений сектор – тест-кейси, які пройшли (Passed), помаранчевий – з невдачею (Failure), червоний – з критичною помилкою (Error). Основний вміст і функції цього розділу є наступними:

- набір звітів/діаграм, де сторінка містить декілька вкладок або секцій, кожна з яких – окремий звіт. Наприклад: “Статус тестів” (кругова діаграма або гістограма, що показує розподіл тест-кейсів за такими результатами як скільки % пройдено, скільки провалено, скільки не запускалося), “Прогрес виконання” (лінійний графік, що показує, як з часом виконувалися тести – крива накопичення пройдених тестів), “Дефекти” (звіт по дефектам: скільки знайдено, відкрито, виправлено, можливо діаграма за пріоритетами дефектів), “Виконання по користувачах” (наприклад, стовпчаста діаграма, яка показує скільки тестів виконав кожен тестувальник і їхній відсоток успішності);

- фільтри звітів, де над діаграмами присутні елементи управління для вибору періоду та умов звіту. Користувач може вибрати діапазон дат (наприклад, за останній місяць, квартал або вручну), конкретний проект чи тест-план (якщо система підтримує кілька проектів або планів), а також фільтрувати за версією продукту чи за командою. Звіти перебудовуються автоматично після зміни фільтрів;

- деталізація, де більшість візуальних звітів інтерактивні. При цьому можна натиснути на сегмент діаграми або колонку графіка, щоб провалитися в деталі. Наприклад, клікнувши на червоний сектор “Failed” на круговій діаграмі статусів, користувач перейде до списку тест-кейсів, які провалилися, або отримає спливаюче вікно з переліком цих тестів. Таким чином, звіти пов’язані з іншими екранами для швидкого доступу до "сирих" даних;

- генерація та експорт, де є екран що надає можливості генерувати звітні документи. Є кнопка “Завантажити звіт” або “Експортувати”, яка дозволяє зберегти

вибраний звіт (або всі одразу) у форматі PDF, Excel чи інші. Це важливо для обміну з керівництвом або замовниками. Передбачено й друковану версію де стиль звітів оптимізований для друку (світлий фон, чіткі кольори);

- поточний статус тестування, де окрім графіків, може бути текстовий підсумок. Наприклад, “Виконано 120 з 150 запланованих тест-кейсів (80%). З них 100 пройдено успішно, 15 виявили дефекти, 5 заблоковані очікуванням середовища.” Такий підсумок відображається вгорі і оновлюється на основі актуальних даних;

- інтерактивність, де при завантаженні екрану “Звіти” система автоматично генерує актуальні діаграми. Користувач може перемикатися між типами діаграм без перезавантаження сторінки – використовуючи AJAX для підвантаження даних. Якщо дані оновлюються (нові тести або результати) – є можливість оновити звіти вручну (кнопка “Обновити дані”) або автооновлення через певний інтервал. Графіки мають легенди, що роблять їх зрозумілими та при наведені на сегмент з’являються підказки з числовими значеннями. Загальний дизайн звітів мінімалістичний: максимум корисної інформації, мінімум зайвого – акцент на кольорових графіках і чітких підписах. Це допомагає швидко проаналізувати якість тестування та приймати рішення (наприклад, які області продукту найбільш проблемні).

Розділ з налаштування (Управління користувачами та параметрами) призначений для адміністрування системи. Тут тест-менеджери (або адміністратори системи) можуть керувати обліковими записами користувачів, їхніми ролями та правами доступу, а також конфігураціями, що стосуються тестування (наприклад, довідники категорій тестів, інтеграції з іншими системами). Для звичайних користувачів розділ Налаштування може містити лише персональні параметри. Наприклад, зміну пароля, мови інтерфейсу. Основний вміст і функції цього розділу наступні:

- управління користувачами, де відображається таблиця зареєстрованих користувачів системи з інформацією, такими як ім’я, email (логін), роль (розробник, тестувальник, менеджер, клієнт, адміністратор), статус аккаунту

(активний/деактивований). Адміністратор може додавати нового користувача (“Додати користувача” – відкривається форма для введення даних і призначення ролі), редагувати існуючих (змінити роль, скинути пароль) або деактивувати/видалити при потребі. Для зручності може бути пошук по користувачах або фільтр за роллю;

- ролі та доступи, де в налаштуваннях є підрозділ “Ролі та права”, де перераховано, які привілеї має кожна роль. Інтерфейс дозволяє кастомізувати права доступу. Наприклад, роль “Клієнт” має доступ тільки на читання до тест-кейсів і звітів, “Тестувальник” може створювати й виконувати тести, але не керує користувачами, “Менеджер” має ширший доступ (всі тестові дані + звіти + користувачі). Адміністратор може створювати нові ролі або змінювати права існуючих через чекбокси/перемикачі в інтерфейсі;

- налаштування тестування, де є інша частина екрану – конфігурації, що впливають на функціонал. Сюди входить управління довідниками. Наприклад, список категорій тест-кейсів (які відображаються в дереві в Списку тестів), можливі статуси тест-кейсів і дефектів (якщо треба додати нестандартні, напр. статус “Blocked”), налаштування пріоритетів (кольори та назви). Також тут можуть налаштовуватися інтеграції з іншими системами – наприклад, прив’язка до баг-трекера JIRA як поле для URL JIRA, облікових даних API, проєкту за замовчуванням для дефектів. Візуально це виглядає як вкладки або розділи форми “Інтеграції”, “Параметри тестів” і т.д;

- параметри системи, де є загальні налаштування як тема оформлення (якщо підтримуються світла/темна), мова інтерфейсу, часовой пояс, формат дати. Користувачі можуть змінити свої персональні (наприклад, вибрати мову). Адміністратор може встановити параметри за замовчуванням для системи;

- інтерактивність, де екран налаштувань забезпечує надійність виконання адміністративних дій, тому що при спробі видалити користувача чи інший важливий елемент система виконає діалог підтвердження.

Такі зміни в налаштуванні застосовуються одразу після збереження конфігураційного файлу. При додаванні нового користувача йому може

автоматично надсилатися запрошення на email. Тоді, використовується простий інтерфейс системи, тобто будуть налаштування що згруповані по вкладках або списку зліва (користувачі, ролі, параметри, інтеграції), щоб адміністратор міг легко знайти потрібний розділ.

Форма додавання/редагування має зрозумілі поля з підказками. Наприклад, паролі повинні містити певні символи – це показано поруч з полем вводу. Завдяки мінімалістичному стилю, навіть цей адмін-екран лишається інтуїтивним так як там немає нічого зайвого, лише необхідні контролли для керування системою.

Клієнти і тестувальники в IT-кампанії, як правило, не взаємодіють з цим розділом або бачать лише обмежені налаштування свого профілю, тож складні опції приховані від невправних користувачів.

Таким чином, створений графічний інтерфейс забезпечує повний цикл управління тестуванням в IT-компанії – від створення тест-кейсів і виконання тестів до відстеження дефектів та отримання звітів. Завдяки використанню Java Spring, реалізовано гнучкий веб-інтерфейс, який адаптується під потреби різних ролей. Кожен екран має чітке призначення і зрозумілий набір елементів, що робить систему легкою у використанні для команди розробки та тестування, а також прозорою для клієнтів. Мінімалістичний дизайн гарантує, що користувачі зосереджуються на своїх задачах, не відволікаючись на інтерфейс, а інтуїтивність структури прискорює освоєння програми.

Таким чином, програма відповідає основним вимогам: управління тест-кейсами, відстеження дефектів, генерація звітів та адміністрування користувачів – все це реалізовано через продуману графічну частину програмного засобу.

3.3 Розробка програмного модуля планування тестування

У рамках розробки програмного засобу для обліку проведення тестування в IT-кампанії було створено графічний інтерфейс користувача (GUI) та бекенд для модуля планування тестування (Додаток В). Графічна частина модуля планування тестування призначена для забезпечення зручного створення та перегляду тест-планів, що є ключовою складовою функціональності системи (рис. 3.3). Реалізація

графічного інтерфейсу здійснена за допомогою мови розмітки гіпертексту (HTML), що дозволяє відображати його у веб-браузері. Далі представлено детальний опис структури, компонентів та призначення елементів цього інтерфейсу на основі програмного коду, що розміщено в додатку В кваліфікаційної роботи. Програмний код графічного інтерфейсу модуля планування тестування реалізований за допомогою технологій HTML5, CSS та JavaScript.

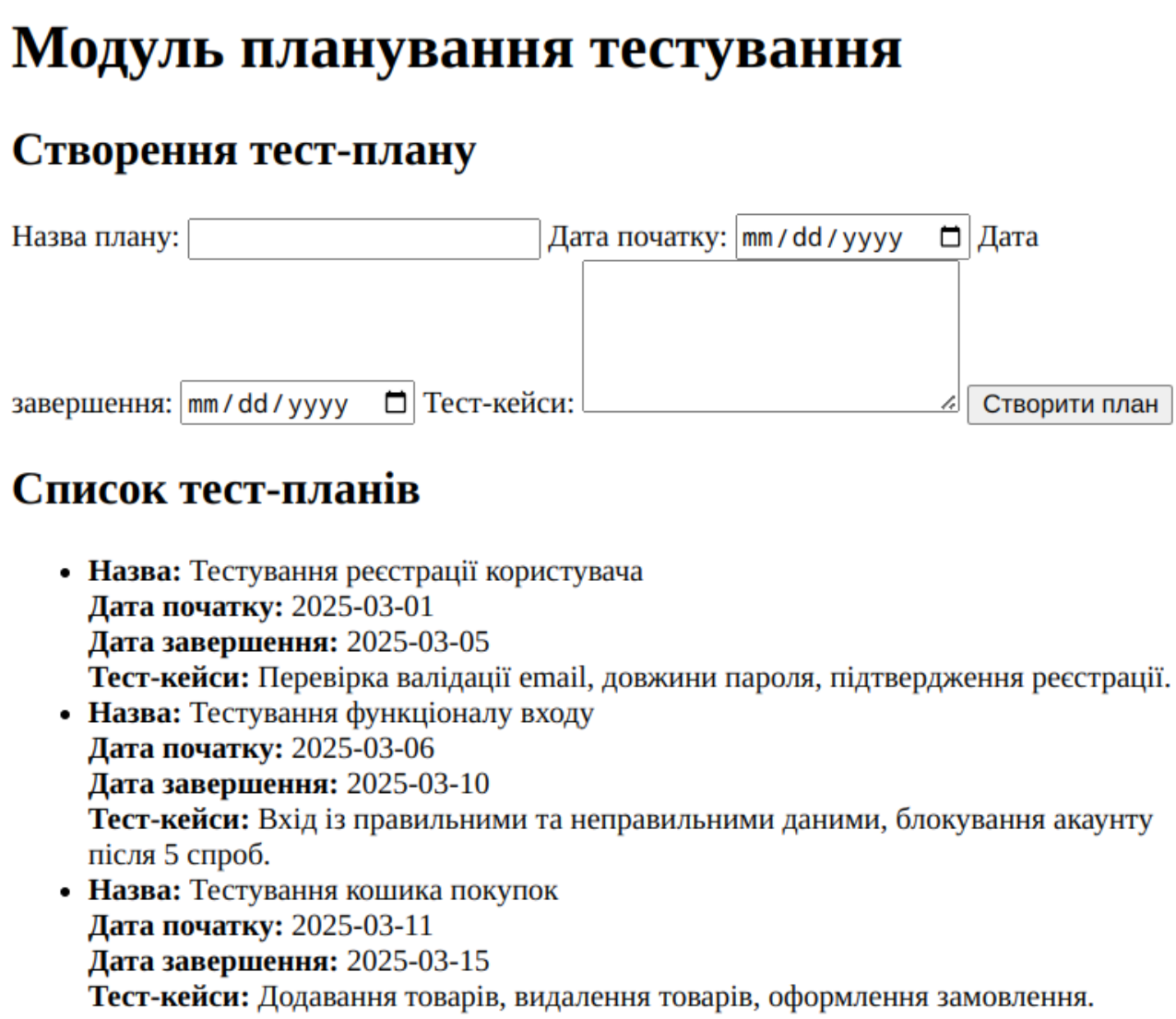

Рисунок 3.3 - Скріншот графічної частини модуля планування тестування

Цей графічний інтерфейс побудовано з використанням стандартних HTML-тегів і має чітко визначену структуру, яка складається з таких основних частин: заголовок сторінки (<header>), що відображає назву модуля та основний вміст (<main>), що включає форму для створення тест-плану та область для відображення

списку існуючих тест-планів. Документ відповідає стандарту HTML5 (<!DOCTYPE html>) та локалізований українською мовою (lang="uk").

Секція <head> містить метадані (кодування UTF-8, адаптивний viewport) та підключення зовнішнього CSS-файлу (styles.css) для стилізації.

Заголовок інтерфейсу реалізовано за допомогою тегу <header>, який містить елемент <h1> із текстом "Модуль планування тестування". Цей елемент виконує роль візуального орієнтира, чітко вказуючи користувачу призначення сторінки. Основна частина інтерфейсу поділена на дві секції, кожна з яких відповідає за певну функцію:

- секція створення тест-плану (<section id="test-plan-form">) містить форму для введення даних нового тест-плану;

- секція списку тест-планів (<section id="test-plans-list">) відображає перелік створених тест-планів.

Форма для створення тест-плану розміщена в секції <section id="test-plan-form"> та реалізується за допомогою тегу <form id="plan-form">. Вона містить позначення (<form>) з полями для введення назви плану, дат початку/завершення (тип date) та текстовою областю для тест-кейсів (<textarea>). Всі поля позначені мітками (<label>) та мають атрибут required, що забезпечує базову валідацію на стороні клієнта. Кнопка "Створити план" ініціює відправку даних.

Секція списку тест-планів (#test-plans-list) включає заголовок другого рівня та порожній контейнер (<ul id="plans-container">) для динамічного відображення створених планів.

Також, використовується скрипт (<script>) для підключення зовнішнього файлу JavaScript для забезпечення динамічної взаємодії з користувачем. Ця структура забезпечує логічне розділення функціональних компонентів і сприяє зрозумілості інтерфейсу.

Програмнийо модуль планування тестування використовує наступну архітектуру взаємодії. Підключення зовнішнього JavaScript-файлу (script.js) забезпечує динамічну обробку подій (на основі сабміт форми) та оновлення інтерфейсу без перезавантаження сторінки. При цьому, використовуються

семантичні тегі (наприклад, <section>, <form>), що покращує доступність інтерфейсу та спрощує стилізацію.

В цілому, графічна частина програмного модуля планування тестування відповідність вимогам, що були зазначені під час проектування, тобто має адаптивний дизайн, що забезпечує коректне відображення на мобільних пристроях, інтуїтивну структуру форм, що сприяє ефективній взаємодії користувачів з системою, модульність коду (розділення HTML/CSS/JS), що дозволяє масштабувати функціонал у майбутньому.

Програмний код бекенду модуля планування тестування реалізований на мові високого рівня Java та Spring Boot у якості веб-фреймворку. Для взаємодії з базою даних застосовується Spring Data JPA, а в якості системи управління базами даних використовується MySQL.

Бекенд побудований за принципами чистої архітектури, що передбачає чітке розмежування рівнів, таких як:

- контролери (Controller), що відповідають за обробку HTTP-запитів;

- сервісний рівень (Service), що реалізує бізнес-логіку;

- репозиторії (Repository), що забезпечують доступ до бази даних.

Модель (Model) програмного додатку представляє сутності, що зберігаються в БД та взаємодіє з окремити модулями та користувачами на основі RESTful API. Так, контролер TestPlanController містить ендпоінти для створення (POST) та отримання (GET) тест-планів, а JSON використовується для обміну даними між клієнтом та сервером.

З метою автоматизації взаємодії з базою даних використано Spring Data JPA для зручного управління записами у таблицях даних. При цьому, анотація @Entity визначає клас як сутність для таблиці test_plans. Використання Hibernate дозволяє автоматично створювати таблиці та підтримувати зв'язок із базою даних.

До основних компонентів модуля планування тестування належить (рис. 3.4):

- головний клас TestPlanApplication – точка входу в додаток, яка ініціалізує Spring Boot;

- клас TestPlanController – REST-контролер, який обробляє запити клієнтів;

- клас TestPlanService – реалізує логіку управління тест-планами;
- клас TestPlanRepository – реалізує рівень доступу до бази даних;
- клас TestPlan – модель сутності тест-плану, що містить такі поля:
- id (унікальний ідентифікатор);
- name (назва тест-плану);
- startDate (дата початку);
- endDate (дата завершення);
- testCases (опис тест-кейсів).

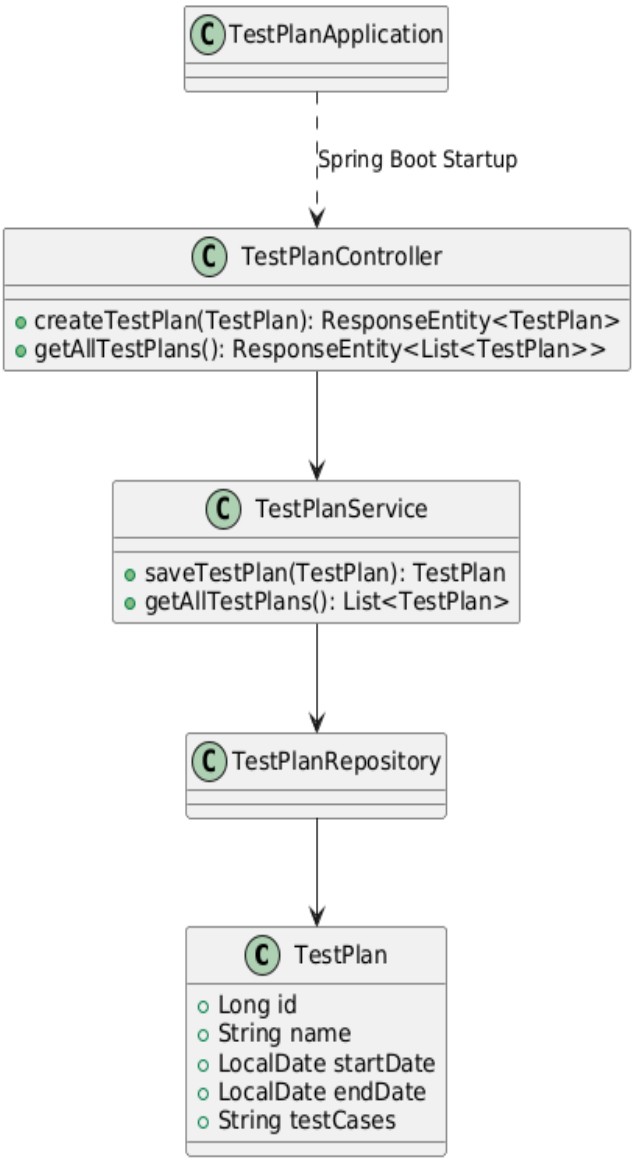

Рисунок 3.4 - UML діаграма класів модуля планування тестування

Діаграма класів для бекенду модуля планування тестування (рис. 3.3) має чітку та ієрархічну структуру, але вона не є надмірно розгалуженою, оскільки кожен клас має чітке призначення. Кількість зв’язків між класами обмежена – вони слідують стандартному шаблону MVC. У майбутньому можна розширити систему, додавши, наприклад:

- додаткові сутності (наприклад, User, TestCase);
- аутентифікацію (наприклад, через Spring Security);
- додаткові сервіси для обробки складніших запитів.

Діаграма класів для бекенду модуля планування тестування (рис. 3.3) можна віднести до помірно складних систем, що слідують стандартній тришаровій архітектурі (Controller-Service-Repository).

Бекенд модуля планування тестування включає наступні основні класи та рівні.

Модельний рівень (Model):

TestPlan – сутність, що містить атрибути тест-плану (назва, дати, тест-кейси).

Рівень доступу до даних (Repository):

TestPlanRepository – інтерфейс, що взаємодіє з базою даних за допомогою Spring Data JPA.

Сервісний рівень (Service):

TestPlanService – містить логіку управління тест-планами.

Контролер (Controller):

TestPlanController – REST-контролер, що обробляє запити на створення та отримання тест-планів.

Головний клас (Application):

TestPlanApplication – точка входу в додаток.

Зв’язки між класами визначаються наступним чином. TestPlanController викликає методи TestPlanService, щоб виконати бізнес-логіку. TestPlanService працює з TestPlanRepository, щоб взаємодіяти з базою даних. TestPlanRepository оперує об'єктами TestPlan, які представляють сутності бази даних.

Конфігурація підключення до бази даних MySQL знаходиться у файлі application.properties, що містить налаштування для підключення наступним чином:

- URL для підключення: jdbc:mysql://localhost:3306/test_plan_db

- облікові дані (ім'я користувача та пароль)

- використовується MySQL8Dialect, а оновлення схеми здійснюється автоматично (ddl-auto=update).

Використання Spring Boot забезпечує швидкий запуск та мінімальну кількість конфігурацій. Такий архітектурний підхід дозволяє легко розширювати функціонал (наприклад, додати оновлення та видалення тест-планів). Тому, завдяки застосуванню Spring Data JPA, розробка CRUD-операцій значно спрощена. Також, впровадження рівня сервісів дозволяє уникнути безпосередньої взаємодії контролера з базою даних, що покращує підтримку та тестованість коду.

Таким чином, розроблений бекенд модуля планування тестування має чітку модульну структуру, що відповідає принципам RESTful API. Він реалізує всі основні операції CRUD, використовуючи Spring Boot, Spring Data JPA та MySQL, що забезпечує ефективне управління тест-планами та можливість подальшого розширення системи.

4 ВПРОВАДЖЕННЯ МОДУЛІВ ПРОГРАМНОГО ДОДАТКУ

4.1 Тестування графічної частини програмного засобу для обліку тестування в ІТ-компанії

Тестування графічної частини програмного засобу для обліку тестування в ІТ-компанії потребувало деякі методи системного підходу (рис. 4.1), що охоплював перевірку функціональності, інтерактивності та контролю доступу для різних ролей користувачів. Веб-додаток на базі Java Spring з динамічним відображенням сторінок можна тестувати за допомогою комбінації модульних, інтеграційних та UI-тестів.

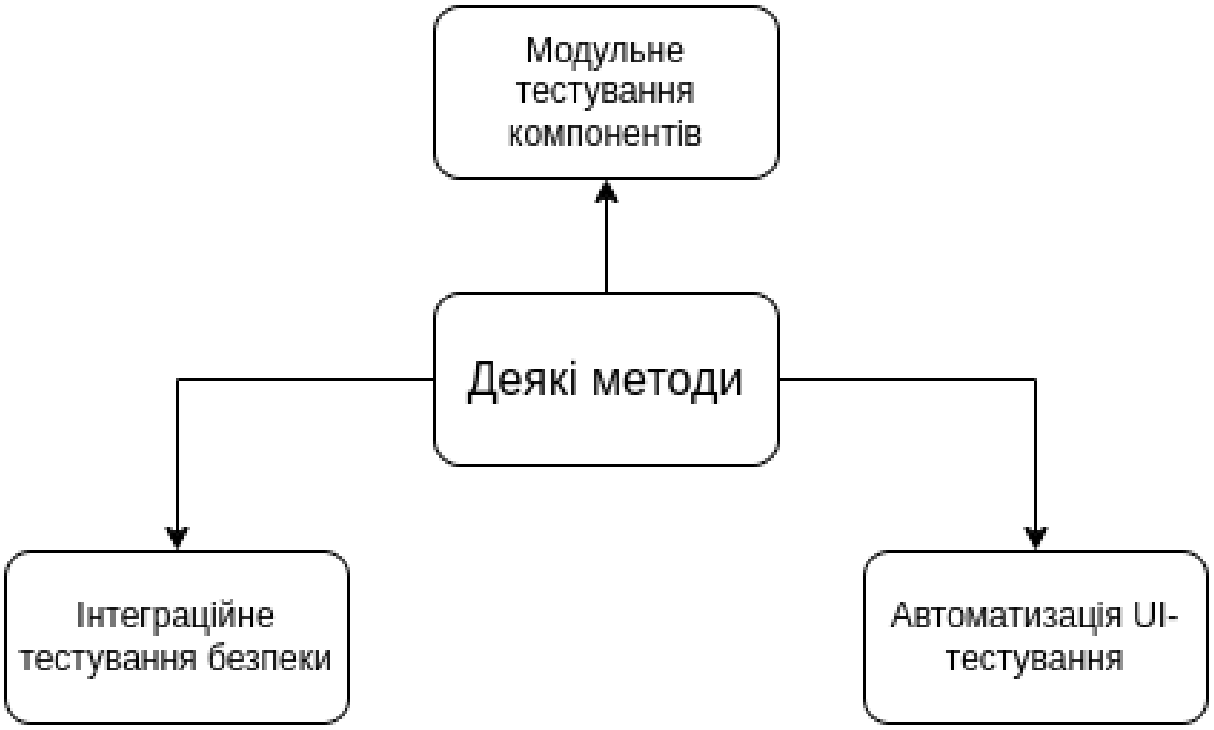

Рисунок 4.1 - Деякі методи системного підходу тестування графічного інтерфейсу програмного засобу

Модульне тестування компонентів графічного інтерфейсу програмного засобу полягало у використанні JUnit та Spring Boot Test для перевірки окремих контролерів. Наприклад, тестування методу контролера для сторінки "Тест-кейси", що включати перевірку коректності передачі даних у модель, наприклад:

@WebMvcTest(TestCaseController.class)

```
class TestCaseControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @Test
    @WithMockUser(roles = "TESTER")
    void testGetTestCasesPage() throws Exception {
        mockMvc.perform(get("/test-cases"))
                .andExpect(status().isOk())
                .andExpect(view().name("test-cases"))
                .andExpect(model().attributeExists("cases"));
    }
}
```

Таке тестування включало перевірку HTTP-статусів, імені представлення та наявності атрибутів моделі.

Інтеграційне тестування безпеки застосувувалось у фреймворку Spring модуля Security Test з анотаціями @WithMockUser для імітації різних ролей, наприклад:

```
@Test
@WithMockUser(roles = "CLIENT")
void testClientAccessToReports() throws Exception {
    mockMvc.perform(get("/reports"))
            .andExpect(status().isForbidden());
}
```

Таке тестування перевіряє обмежений доступ клієнтів до розділу "Звіти", тоді як менеджери мають повний доступ.

Автоматизація UI-тестування полягало у перевірці інтерактивності, що використовувались за допомогою різних інструментів таких як Selenium та SpringUIUnitTest, що виконували:

- створення сценаріїв для кожної ролі з перевіркою доступних функцій;

- тестування форм створення тест-кейсів та відправки дефектів;

- валідація динамічних елементів (наприклад, графіків у розділі "Статистика"), що також забезпечувались контролем якості та покриття (табл. 4.1).

Таблиця 4.1 - Деякі аспекти тестування контролю якості та покриття

Аспект тестування	Інструменти	Критерії успіху
Функціональність	JUnit, MockMvc	100% покриття основних сценаріїв
Безпека	Spring Security, OWASP ZAP	Відсутність CVSS-вразливостей ≥7.0
Продуктивність	JMeter	Час відгуку <2 сек для 100 користувачів

Також, для забезпечення надійності проведення тестових робіт було реалізовано CI/CD пайплайни з автоматичним запуском тестів після кожного коміту. Аналіз результатів тестування включав генерацію звітів у форматі HTML/PDF з деталізацією помилок та рекомендаціями щодо покращення. Такі звіти про помилки в системі обліку тестування містили структуровані дані, що дозволяли швидко ідентифікувати проблеми та вживати коригувальні дії. Наведено два типових звіти з рекомендаціями щодо покращення ситуації під час тестування програмного засобу.

Звіт 1: Звіт про функціональну помилку в розділі "Тест-кейси"

ID дефекту: TC-2025-045

Заголовок: Некоректне відображення статусу тест-кейсу після повторного запуску

Середовище:

ОС: Ubuntu 22.04

Браузер: Chrome 121.0

Версія додатку: 2.1.3

Опис проблеми:

Після повторного виконання тест-кейсу зі статусом "Провалено" система не оновлює історію статусів. Натомість новий результат замінює попередній, що ускладнює аналіз тенденцій.

Кроки відтворення:

Увійдіть як тестувальник.

Відкрийте тест-кейс з ID TC-789.

Виконайте тест-кейс із результатом "Провалено".

Повторіть виконання того ж тест-кейсу з результатом "Успішно".

Очікуваний результат:

Вкладка "Історія статусів" містить два запси: "Провалено" та "Успішно".

Фактичний результат:

Відображається лише останній статус "Успішно".

Звіт 2: Звіт про проблему безпеки в розділі "Налаштування"

ID дефекту: SEC-2025-012

Заголовок: Вразливість XSS у полі "Опис профілю" для ролі менеджера

Середовище:

ОС: Windows 11

Браузер: Firefox 120.0

Версія додатку: 2.1.3

Опис проблеми:

Введення скрипту <script>alert('XSS')</script> у поле "Опис профілю" призводить до виконання коду на стороні клієнта.

Кроки відтворення:

Увійдіть як менеджер.

Перейдіть до розділу "Налаштування" → "Профіль".

Введіть <script>alert('XSS')</script> у поле "Опис".

Збережіть зміни.

Очікуваний результат:

Система блокує введення небезпечних скриптів із повідомленням про недійсні символи.

Фактичний результат:

При перегляді профілю виникає спливаюче вікно з текстом "XSS".

Загальні рекомендації щодо покращення системи звітності

Автоматизація збору даних:

Інтегрувати інструменти на такі як Disbug для автоматичного захоплення скріншотів, консольних логів та відтворення сценаріїв.

Пріоритезація помилок:

Використовувати матрицю "Severity vs Priority" для класифікації дефектів (наприклад, критичні помилки безпеки – найвищий пріоритет).

Покращення комунікації:

Додати поле "Коментарі розробника" у звіт для відстеження прогресу виправлення.

Для зменшення кількості помилок, пов’язаних із середовищем, рекомендується використовувати Docker-контейнери зі стандартизованими конфігураціями.

Таким чином виконувалось тестування графічної частини програмного засобу для обліку проведення тестування в ІТ-компанії.

4.2 Впровадження запропонованого методу інтеграції тестування з системою управління дефектами

Метод, який заснований на інтеграції процесу тестування з системою управління дефектами, що розглядається в даному підрозділу та дає можливість оперативно відстежувати виявлені дефекти та аналізувати прогрес тестування в реальному часі використовує безперервне тестування (Continuous Testing). Безперервне тестування є ключовою складовою підходу DevOps, який об’єднує розробку, тестування та розгортання програмного забезпечення в єдиний безперервний процес. Розроблений метод передбачає (рис. 4.2):

- автоматизацію тестів, де тести створюються та виконуються автоматично, що значно прискорює процес перевірки програмного забезпечення;

- автоматичне виконання тестування при кожній зміні програмного коду, тобто тести запускаються щоразу, коли вносяться зміни до коду, наприклад, при коміті в систему контролю версій. Це дозволяє виявляти проблеми на ранніх етапах;

- інтеграцію з системою управління дефектами при якій виявлені дефекти автоматично реєструються в таких інструментах, як Jira, Bugzilla або інших системах управління дефектами. Це забезпечує швидке документування та відстеження проблем.

Автоматизація тестів — це процес, у якому спеціальні інструменти та скрипти використовуються для автоматичного створення та виконання тестових сценаріїв, що значно прискорює перевірку програмного забезпечення. Завдяки цьому команди розробки можуть швидше отримувати зворотний зв’язок про якість продукту, що сприяє прискоренню випуску програмного забезпечення на ринок. Автоматизація тестів здійснюється наступним чином.

1А. Вибір інструментів для автоматизації. Для створення та виконання тестів обираються спеціалізовані інструменти залежно від типу програми та потреб команди.

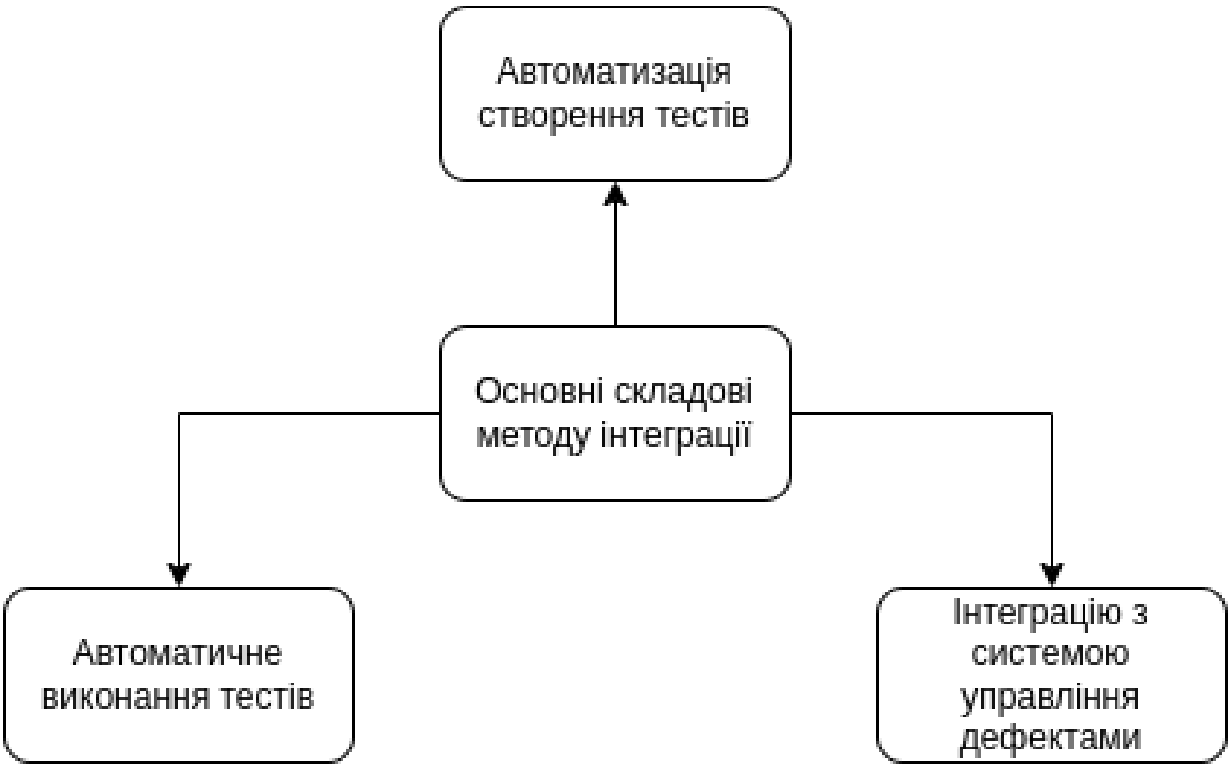

Рисунок 4.2 - Основні складові методу інтеграції процесу тестування з системою управління дефектами

До юніт-тестів (перевірка окремих модулів коду) входять JUnit (для Java), PyTest (для Python). До інтеграційних тестів (перевірка взаємодії компонентів) входять Postman (для API), Selenium (для веб-додатків). До UI-тестів (перевірка інтерфейсу користувача) входять Selenium, Appium (для мобільних додатків). Навантажувальних тестів (перевірка продуктивності) входять JMeter, Gatling. Вибір інструменту залежить від типу тестування, мови програмування та специфіки проєкту.

2. Розробка тестових скриптів. Тестові скрипти — це набір інструкцій, які автоматично виконують дії користувача або перевіряють функції програми. Їх створення може відбуватися двома способами. Ручне написання, де скрипти пишуться на мовах програмування, таких як Python або Java. Наприклад, скрипт

може імітувати введення даних у форму або перевіряти відповідь API. Автоматична генерація, де деякі інструменти, як Selenium IDE, дозволяють записувати дії користувача (наприклад, натискання кнопок чи заповнення полів) і автоматично перетворювати їх на скрипти.

Наведемо приклади дій у скриптах під час використання данного методу:

- введення імені та пароля у форму входу;

- перевірка, чи відображається повідомлення про успішну реєстрацію;

- запит до бази даних для перевірки збереження даних.

3. Налаштування тестового середовища. Перед виконанням тестів необхідно підготувати середовище, яке імітує реальні умови роботи програми. Наведемо приклади:

- налаштування баз даних із тестовими даними;

- запуск серверів або контейнерів (наприклад, через Docker);

- налаштування мережевих параметрів для імітації реального використання.

Це гарантує, що тести виконуються в стабільних і контрольованих умовах.

4. Виконання тестів. Тести запускаються автоматично, часто в рамках безперервної інтеграції (CI) означає наступне. CI-системи, такі як Jenkins або GitLab CI, автоматично виконують тести щоразу, коли розробник вносить зміни до коду (наприклад, при коміті в репозиторій).

Тести можуть запускатися в наступних випадках:

- локально на комп’ютері розробника;

- на сервері перед злиттям коду;

- перед розгортанням програми в продакшені.

Сучасні інструменти дозволяють виконувати тести паралельно на кількох машинах або в хмарі, що ще більше прискорює процес.

5. Аналіз результатів. Після виконання тестів генеруються звіти, які показують які тести пройшли успішно, які тести виявили помилки чи дефекти.

Якщо є інтеграція з системами управління задачами (наприклад, Jira), дефекти автоматично реєструються. Команда аналізує результати, виправляє помилки та повторно запускає тести за потреби. Тому, така автоматизація тестів

значно скорочує час і зусилля, необхідні для перевірки програмного забезпечення, завдяки таким перевагам:

- швидкість виконання, де автоматизовані тести працюють набагато швидше, ніж ручні. Наприклад, тест форми реєстрації, який вручну займає 5–10 хвилин, може бути виконаний за секунди;

- повторюваність, де тести можна запускати багато разів без додаткових зусиль, що ідеально для регресійного тестування (перевірки, чи не порушили нові зміни існуючі функції);

- зменшення впливу людського фактору, де автоматизація усуває помилки, які можуть виникнути через неуважність або втому тестувальника;

- поширене покриття, де автоматизовані тести можуть перевіряти велику кількість сценаріїв, включаючи складні або рідкісні випадки, які важко відтворити вручну;

- паралельність, де тести можуть виконуватися одночасно на кількох пристроях, що економить час при перевірці великих систем.

Наприклад, необхідно виконати ручне тестування веб-додатку з формою реєстрації, що включає:

- відкриття браузера.

- перехід на сторінку.

- введення даних (ім’я, email, пароль).

- натиснення кнопки "Зареєструватися".

- перевірити, чи з’явився користувач у базі даних.

Для режиму автоматизації властиво створення скрипту на Selenium, який відкриває браузер, заповнює форму, натискає кнопку і перевіряє базу даних автоматично. Цей процес займає кілька секунд і може повторюватися при кожній зміні коду без участі людини.

Таким чином, автоматизація тестів дозволяє створювати та виконувати тести автоматично, що значно прискорює процес перевірки програмного забезпечення. Це забезпечує швидший зворотний зв’язок, підвищує якість продукту та оптимізує роботу команди розробки.

Автоматичне виконання тестування при кожній зміні програмного коду є важливою частиною сучасних практик розробки, таких як безперервна інтеграція (CI). Цей процес дозволяє швидко виявляти помилки та забезпечувати якість програмного забезпечення щоразу, коли розробник вносить зміни, наприклад, робить коміт у систему контролю версій. Розглянемо впровадження методу інтеграції тестування з системою управління дефектами наступним чином.

1. Система контролю версій, де розробка програмного забезпечення зазвичай відбувається з використанням систем контролю версій, таких як Git (разом із платформами GitHub, GitLab, Bitbucket тощо). При цьому, розробник вносить зміни до коду та фіксує їх через коміт — це запис змін у репозиторії. Коміти можуть бути частиною окремих гілок (branches), наприклад, для розробки нових функцій чи виправлення помилок. Коли коміт відбувається, це стає сигналом для запуску автоматичних процесів.

2. Роль безперервної інтеграції (CI). Безперервна інтеграція — це практика, яка передбачає автоматичну перевірку кожної зміни коду перед її інтеграцією в основну гілку проєкту. Для цього використовуються спеціальні інструменти — CI-сервери або наступны хмарні сервіси як Jenkins, GitHub Actions, GitLab CI, CircleCI, Travis CI. Ці інструменти постійно стежать за репозиторієм і реагують на нові коміти, запускаючи заздалегідь налаштовані дії.

3. Налаштування CI-пайплайну. CI-пайплайн — це автоматизований набір кроків, який виконується при кожній зміні коду. Зазвичай він включає:

- збірку проєкту, де у випадку якщо код потребує компіляції (наприклад, у мовах типу Java чи C++), вона виконується автоматично;

- виконання тестів, де запускаються автоматизовані тести для перевірки коду;

- аналіз коду, де виконується статичний аналіз для пошуку потенційних проблем.

Для тестування в пайплайні налаштовуються команди, які запускають відповідні тестові фреймворки, наприклад: JUnit (для Java), PyTest (для Python), Selenium (для тестування інтерфейсів).

4. Процес запуску тестів відбувається наступним чином. Розробник робить коміт у систему контролю версій (наприклад, у гілку feature/new-button). Далі CI-сервер автоматично виявляє цю зміну і завантажує останню версію коду з репозиторію. Потім запускається CI-пайплайн, який включає виконання тестів. Тести можуть бути різними, наприклад:

- юніт-тести, які перевіряють окремі функції або методи;

- інтеграційні тести, які перевіряють взаємодію між компонентами системи;

- енд-ту-енд тести, які імітують дії користувача для перевірки всієї програми.

Якщо тести проходять успішно, то зміна вважається стабільною. Якщо ні — то процес зупиняється, і команда отримує повідомлення про помилку.

5. Зворотний зв’язок виконується після завершення тестування результати автоматично надсилаються команді через:

- електронну пошту;

- повідомлення в месенджерах (наприклад, Slack чи Microsoft Teams);

- інтерфейс CI-платформи (дашборд із логами та звітами).

Якщо тести не пройшли, розробник може одразу переглянути звіт, виправити проблему та зробити новий коміт, що знову запустить увесь процес.

Наприклад, розробник додає нову функцію авторизації та робить коміт у гілку feature/login. Тоді CI-сервер (наприклад, GitHub Actions) реагує на коміт і запускає пайплайн. Далі у пайплайні збирається проєкт (якщо потрібно), запускаються юніт-тести для перевірки логіки авторизації та виконуються інтеграційні тести для перевірки роботи з базою даних. Якщо тести успішні, зміну можна злити в основну гілку. Якщо ні — розробник отримує звіт із деталями помилки.

До переваги такого підходу відноситься:

- швидке виявлення помилок, де існуючі проблеми фіксуються одразу після внесення змін;

- якість коду, де існує постійне тестування підтримує стабільність проєкту;

- економія часу, де немає потреби вручну запускати тести;

- готовність до релізу, де код завжди перебуває в робочому стані.

Отже, автоматичне виконання тестів при кожній зміні коду забезпечується завдяки інтеграції системи контролю версій (наприклад, Git) із CI-сервером, який реагує на коміти та запускає тести в рамках налаштованого пайплайну. Це дозволяє командам розробки працювати швидко, ефективно та з високою впевненістю в якості свого продукту.

Подальша інтеграція з системою управління дефектами, такою як Jira, Bugzilla або іншими, дозволяє автоматично реєструвати виявлені дефекти під час тестування програмного забезпечення. Це значно прискорює процес відстеження та виправлення помилок, а також підвищує ефективність команд розробки. Це відбувається внаслідок того, що система управління дефектами — це інструмент, який допомагає командам розробки відстежувати, керувати та виправляти дефекти (баги) у програмному забезпеченні. Популярні приклади таких систем включають Jira, Bugzilla, Trello, Asana. Ці системи дозволяють створювати задачі, призначати відповідальних, встановлювати пріоритети та відстежувати прогрес виправлення дефектів. Тому, інтеграція - це процес з'єднання тестового середовища, де виконуються автоматизовані тести, із системою управління дефектами. Завдяки цьому, коли тест не проходить, інформація про дефект автоматично передається до системи управління дефектами, де створюється нова задача для подальшого опрацювання.

Автоматична реєстрація дефектів базується на використанні API (Application Programming Interface) — набору функцій, які дозволяють одній системі взаємодіяти з іншою. Наприклад, Jira підтримує REST API, через яке можна створювати, оновлювати чи видаляти задачі програмно.

Bugzilla також має власний API для автоматизації роботи з дефектами. Коли автоматизований тест не проходить, система може автоматично викликати API системи управління дефектами і створити нову задачу з деталями про виявлену проблему.

Така інтеграція зазвичай реалізується через CI/CD інструменти (інструменти безперервної інтеграції та доставки), такі як Jenkins, GitLab CI, CircleCI, Travis CI. Ці інструменти запускають автоматизовані тести при кожній зміні коду. Якщо тест

не проходить, вони можуть автоматично створити дефект у системі управління дефектами. Для спрощення інтеграції часто використовуються плагіни або розширення. Наприклад, у Jenkins є плагіни для Jira, які дозволяють створювати задачі при невдалих тестах чи збірках.

Щоб налаштувати автоматичну реєстрацію дефектів під час використання даного методу, потрібно виконати такі кроки:

- налаштувати доступ до системи управління дефектами, де створити обліковий запис із правами на створення задач та отримати API-ключ або токен для автентифікації (наприклад, у Jira це може бути персональний токен доступу);

- додати деякі кроки в CI/CD пайплайн, де у конфігурації CI/CD (наприклад, у файлі Jenkinsfile для Jenkins або .gitlab-ci.yml для GitLab CI) додається крок, який активується при невдачі тесту. Цей крок викликає API системи управління дефектами для створення нової задачі;

- передати деталі дефекту, де у задачі автоматично заповнюються поля, такі як назва дефекту (наприклад, "Тест авторизації не пройшов"), опис із деталями помилки, пріоритет (наприклад, високий, якщо це критичний тест), додаткові дані, як-от логи чи знімки екрану.

Розглянемо приклад типового процесу автоматичної реєстрації, коли розробник робить коміт коду у систему контролю версій (наприклад, Git). Тоді CI-сервер (наприклад, Jenkins) виявляє зміну і запускає автоматизовані тести. Якщо тест не проходить, то CI-сервер використовує API системи управління дефектами (наприклад, Jira API) для створення нової задачі. Для цього, у такій задачі вказуються деталі, такі як назва тесту, опис помилки, логи тощо. Далі, така задача автоматично призначається відповідальному розробнику чи команді.

До переваг автоматичної реєстрації дефектів відноситься:

- швидкість, де дефекти реєструються миттєво, що дозволяє командам реагувати одразу;

- точність, де автоматизація зменшує ймовірність помилок при введенні даних вручну;

- прозорість, де усі члени команди бачать актуальну інформацію про стан проєкту;

- ефективність, де розробники можуть зосередитися на виправленні дефектів, а не на ручному створенні задач.

Таким чином, інтеграція з системою управління дефектами, такою як Jira чи Bugzilla, дозволяє автоматично реєструвати дефекти, виявлені під час автоматизованих тестів. Це досягається завдяки API та інструментам CI/CD, які налаштовуються для створення задач у разі невдачі тестів. Такий підхід значно покращує процес управління якістю програмного забезпечення, роблячи його швидшим, точнішим і ефективнішим.

Для визначення ефективності запропонованого методу інтеграції процесу тестування з системою управління дефектами ефективність, необхідно проаналізувати кілька ключових аспектів. Розглянемо такі ключові аспекти більш детально.

1. Оцінка швидкості виявлення та реєстрації дефектів. Ефективність методу можна визначити, вимірявши, наскільки швидко дефекти виявляються і фіксуються в системі. Якщо інтеграція дозволяє автоматично реєструвати дефекти в реальному часі, це значно скорочує час, який раніше витрачався на ручне документування. Наприклад, можна порівняти часові затрати на реєстрацію дефектів до і після впровадження інтеграції.

2. Аналіз точності та повноти даних про дефекти. Важливо оцінити, чи забезпечує система детальну та точну інформацію про дефекти. Наприклад, логи, знімки екрану або кроки для відтворення проблеми. Ці дані полегшують розробникам пошук і виправлення помилок. Порівняння якості документації дефектів із автоматизованою системою та без неї може показати підвищення ефективності.

3. Вплив на комунікацію в команді. Інтеграція може покращити видимість стану проєкту для всіх учасників, що сприяє швидшому прийняттю рішень і кращій координації. Щоб це визначити, можна проаналізувати, як швидко команда реагує

на дефекти та чи зменшується кількість непорозумінь завдяки кращій доступності інформації.

4. Зменшення кількості пропущених дефектів. Ефективність методу також залежить від того, наскільки він знижує ризик пропуску критичних проблем. Якщо система автоматично фіксує всі виявлені дефекти, це можна перевірити, порівнявши кількість пропущених помилок до і після інтеграції.

Таким чином, підвищення ефективності методу інтеграції процесу тестування з системою управління дефектами можна визначити, оцінивши:

- скорочення часу на виявлення та реєстрацію дефектів;
- покращення якості та повноти документації;
- підвищення рівня комунікації в команді;
- зменшення кількості пропущених проблем.

Провівши аналіз за цими критеріями, можна зробити обґрунтований висновок про те, наскільки цей метод сприяє підвищенню ефективності роботи команди. Отже, в роботі проведено таке дослідження. Так, були отримані дані з IT-кампанії, де виконувалось впровадження методу, що інтегрує процес тестування з системою управління дефектами (Jira) та підвищує ефективність проведення тестових робіт на більш ніж 14%. Для цього проведемо моніторинг роботи команди тестувальників, яка працює над проєктом і щодня виявляє певну кількість дефектів. Далі, порівняємо витрати часу до і після впровадження інтеграції, щоб показати економію та розрахувати підвищення ефективності.

Наведемо початкові дані (до інтеграції):

- час на ручну реєстрацію одного дефекту: 10 хвилин;
- середня кількість дефектів, виявлених за день: 17.

Тоді, знайдемо середній загальний час на реєстрацію дефектів за день:

17 дефектів × 10 хвилин = 170 хвилин

Середній час на виконання тестів за день складатиме: 1000 хвилин (це час на інші завдання тестування, окрім реєстрації дефектів).

Тоді, загальний час на тестування до інтеграції:

1000 хвилин + 170 хвилин = 1170 хвилин

Отримуємо статистику після інтеграції.

Час на автоматичну реєстрацію одного дефекту: 0 хвилин (процес повністю автоматизований завдяки інтеграції);

Загальний час на реєстрацію дефектів за день: 0 хвилин.

Загальний час на тестування після інтеграції:

1000 хвилин + 0 хвилин = 1000 хвилин

Виконаємо розрахунок підвищення ефективності:

Економія часу складатиме:

$$1170 \text{ хвилин} - 1000 \text{ хвилин} = 170 \text{ хвилин}$$

Відсоток підвищення ефективності:

$$(170 \text{ хвилин} / 1170 \text{ хвилин}) \times 100\% \approx 14{,}53\%$$

Таким чином, у наведеному прикладі команда тестувальників, яка виявляє 17 дефектів на день і витрачає 10 хвилин на ручну реєстрацію кожного, може заощадити 170 хвилин завдяки інтеграції процесу тестування з системою управління дефектами. Це скорочує загальний час на тестування з 1170 хвилин до 1000 хвилин, що дає підвищення ефективності виконання тестових робіт приблизно на 14,53%. Таким чином, автоматизація реєстрації дефектів за допомогою інтеграції дозволяє значно оптимізувати процес тестування.

4.3 Впровадження запропонованого методу обліку проведення тестування програмного забезпечення

Процес обліку тестування програмного забезпечення, у якому реалізовано комплексний підхід щодо управління тестами і дефектами використовує інтегровану систему, що об'єднує планування, виконання тестів і відстеження дефектів в єдину платформу. На відміну від традиційних методів, де тестування і управління дефектами часто розглядаються як окремі процеси, такий підхід забезпечує їх тісну взаємодію, автоматизацію рутинних завдань і підвищення прозорості. Це дозволяє ефективно керувати якістю програмного забезпечення на всіх етапах розробки.

Комплексний підхід означає, що система не просто відстежує тести та дефекти окремо, а пов’язує їх у єдиний процес. Наприклад, інформація про тести (тестові випадки, плани, результати) інтегрується з даними про дефекти, що дозволяє легко встановити зв’язок між ними. Це відрізняється від існуючих традиційних систем, де тести і дефекти часто управляються ізольовано, без автоматизованого зв’язку чи глибокої інтеграції.

До ключових особливостей комплексного підходу (рис. 4.3) відносяться:

- єдина платформа для тестів і дефектів, де всі дані, від тестових випадків до статусу дефектів, зберігаються в одній системі. Це спрощує аналіз і допомагає швидко пов’язати дефект із тестом, який його виявив;

- автоматизація процесів, де у випадку, якщо тест не проходить, система може автоматично створити дефект у системі управління (наприклад, Jira), додавши деталі: логи, знімки екрану тощо. Статуси тестів і дефектів оновлюються автоматично, що зменшує ручну роботу;

- відстеження зв’язків, де кожен дефект прив’язаний до конкретного тесту, що дозволяє розробникам швидко зрозуміти проблему і її вплив на продукт;

- аналіз у реальному часі, де система надає актуальні звіти про прогрес тестування, кількість дефектів, час їх виправлення та покриття тестами. Це допомагає оцінити готовність продукту до релізу;

- інтеграція з інструментами розробки, де комплексний підхід часто включає зв’язок із системами управління версіями (Git) і CI/CD інструментами (Jenkins), що дозволяє запускати тести автоматично при змінах у коді та фіксувати дефекти одразу.

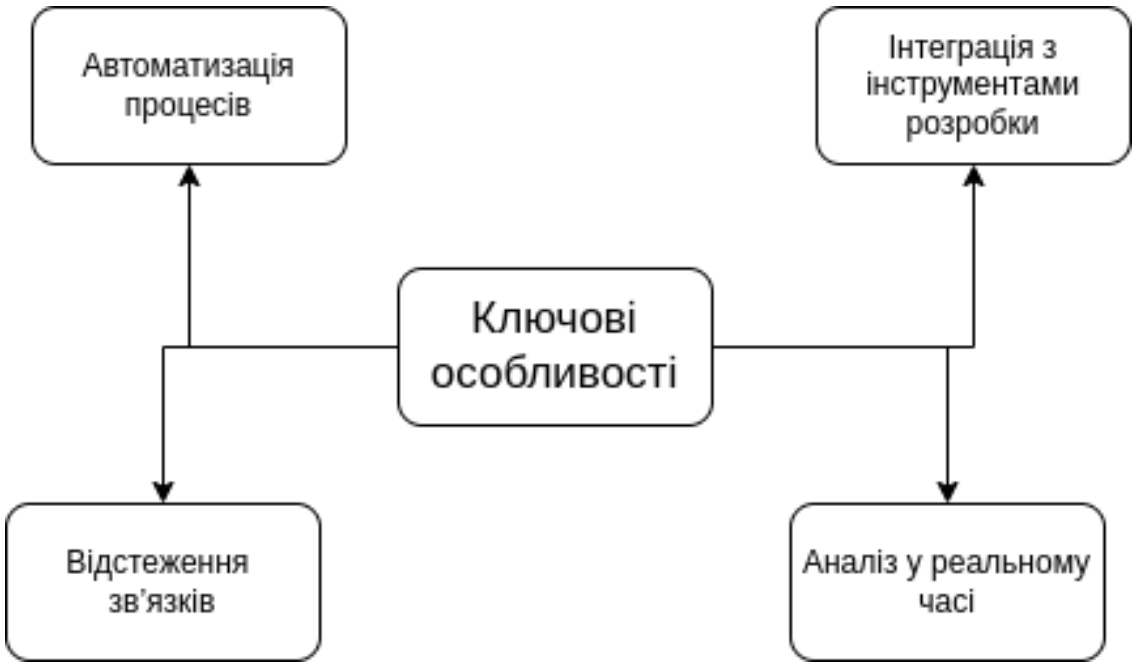

Рисунок 4.3 - Ключові особливості комплексного підходу методу обліку проведення тестування програмного забезпечення

До переваги комплексного підходу відносяться (рис. 4.4):

- ефективність, де надана автоматизація скорочує час на рутинні завдання, такі як реєстрація дефектів;

- якість, де тісний зв'язок між тестами і дефектами допомагає швидше виявляти та виправляти помилки;

- прозорість, де всі учасники проєкту бачать актуальний стан тестування і дефектів;

- швидший зворотний зв'язок, де розробники отримують миттєві сповіщення про проблеми, що прискорює їх виправлення.

Приклад використання комплексного підходу методу обліку проведення тестування програмного забезпечення. Є команда, що застосовує певний інструмент тестування, інтегрований із Jira. Якщо автоматизований тест не проходить, то система автоматично створює задачу в Jira з усіма деталями: логами та кроками відтворення. Внаслідок цього, розробник одразу отримує сповіщення, виправляє дефект, а після повторного тестування статус задачі оновлюється автоматично.

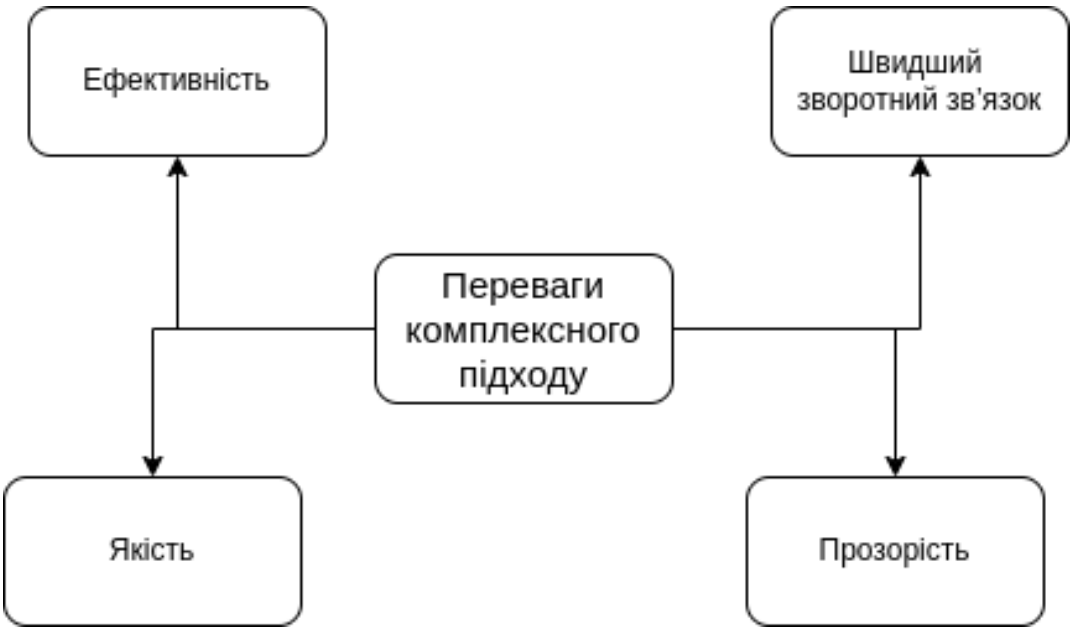

Рисунок 4.4 - Переваги комплексного підходу методу обліку проведення тестування програмного забезпечення

Таким чином, процес обліку тестування програмного забезпечення з комплексним підходом до управління тестами і дефектами — це система, яка об’єднує всі аспекти тестування і дефект-менеджменту в єдину платформу. На відміну від традиційних методів, вона автоматизує рутинні завдання, покращує комунікацію в команді, підвищує якість продукту і скорочує час на виявлення та виправлення помилок. Це робить її більш ефективною і сучасною альтернативою ізольованим підходам.

Під час впровадження методу обліку проведення тестування програмного забезпечення доцільно проводити визначення підвищення ефективності контролю якості. Для визначення підвищення ефективності контролю якості програмного забезпечення можна скористатися систематичним підходом, який включає аналіз ключових показників ефективності (KPI). Для цього виконується порівняння їх до і після впровадження методу, а також оцінку загального впливу на якість продукту. Розглянемо більш ретельно визначення підвищення ефективності контролю якості програмного забезпечення. Для цього виконуються наступні кроки.

1. Визначення ключових показників ефективності (KPI). Для оцінки ефективності контролю якості необхідно обрати метрики, які відображають основні аспекти процесу тестування. Основні показники включають:

- час виявлення дефектів, де середній час від моменту внесення дефекту в код до його виявлення під час тестування;

- кількість пропущених дефектів, де є відсоток дефектів, які не були виявлені під час тестування і потрапили в продакшен;

- час на виправлення дефектів, де середній час від реєстрації дефекту до його усунення;

- покриття тестами, де визначається відсоток функціональності програми, охоплений тестами.

Ці показники дозволяють оцінити швидкість, точність і повноту процесу контролю якості.

2. Вимірювання базового рівня (до впровадження методу). Перед застосуванням методу обліку проведення тестування (наприклад, інтеграції з системами управління дефектами чи автоматизації тестів) збирають дані за цими показниками за певний період (наприклад, кілька спринтів або релізів). Це створює базовий рівень для порівняння.

3. Вимірювання показників після впровадження методу. Після впровадження методу повторно вимірюються ті ж KPI за аналогічний період. Збір даних має бути достатньо тривалим, щоб результати були статистично значущими. Наприклад, якщо час виявлення дефектів скоротився, це свідчить про швидшу реакцію на проблеми. Зменшення кількості пропущених дефектів вказує на підвищення якості тестування. Скорочення часу на виправлення дефектів показує кращу організацію процесу. Збільшення покриття тестами демонструє ширше охоплення функціональності.

4. Порівняння показників і розрахунок покращень. Після впровадження методу порівнюються показники до і після його впровадження та розраховується відсоток покращення для кожного з них. Формула для розрахунку є наступною:

Покращення = (Значення_до − Значення_після) / Значення_до × 100%

Наприклад:

До впровадження: час виявлення дефекту — 4 дні, після — 2 дні.

Покращення = (4 − 2) / 4 × 100% = 50%

До впровадження: пропущені дефекти — 10%, після — 5%.

Покращення = (10 − 5) / 10 × 100% = 50%

До впровадження: час на виправлення — 2 дні, після — 1 день.

Покращення = (2 − 1) / 2 × 100% = 50%

До впровадження: покриття тестами — 70%, після — 85%.

Покращення = (85 − 70) / 70 × 100% ≈ 21%

Середнє покращення можна обчислити як:

Середнє покращення = (50% + 50% + 50% + 21%) / 4 ≈ 42.75%

5. Оцінка впливу на загальну якість

Окрім числових показників, необхідно враховувати наступні якісні аспекти:

- відгуки користувачів для визначення чи зменшилася кількість скарг після впровадження методу;

- стабільність продукту для визначення чи скоротилася кількість критичних інцидентів у продакшені.

Якщо такі або подібні показники покращилися, то це підтверджує підвищення ефективності контролю якості.

Таким чином, підвищення ефективності контролю якості програмного забезпечення при використанні методу обліку проведення тестування можна визначити шляхом порівняння ключових показників (час виявлення та виправлення дефектів, кількість пропущених дефектів, покриття тестами та інші подібні) до і після його впровадження. Розрахувавши відсоток покращення для кожного KPI та оцінивши якісні зміни, такі як відгуки користувачів, можна об’єктивно встановити, наскільки метод сприяє покращенню процесу контролю якості. У наведеному прикладі середнє покращення склало близько 42.75%, що свідчить про значний позитивний вплив методу.

Для визначення підвищення ефективності методу обліку проведення тестування програмного забезпечення з комплексним підходом до управління тестами і дефектами, розглянемо конкретний сценарій із числовими показниками, який було проведено в даній роботі. Такий комплексний підхід передбачає інтеграцію процесів управління тестами і дефектами, автоматизацію певних задач та оптимізацію роботи команди тестувальників.

Далі розглянемо проведено дослідження, де спочатку було визначено результатів початкового стану, тобто до впровадження комплексного підходу. В цьому випадку, команда тестувальників щодня виконували завдання, що були пов’язані з тестуванням програмного забезпечення. Загальний час, що витрачений на тестування, включав виявлення дефектів, їх реєстрацію та інші супутні процеси. У цьому випадку, середній загальний час на тестування за день складав 1000 хвилин (приблизно 16,7 годин, що відповідало роботі кількох тестувальників). Сюди входило:

- виявлення дефектів (наприклад, аналіз коду чи тестування інтерфейсу);
- реєстрація дефектів вручну (запис у систему, опис проблеми);
- інші завдання (планування тестів, аналіз результатів).

Впровадження комплексного підходу передбачало до управління тестами і дефектами та включали наступні покращення:

- автоматизацію реєстрації дефектів, де замість ручного введення дефектів у систему (наприклад, Jira) впроваджується автоматична реєстрація через інтеграцію тестових інструментів із системою управління задачами;
- інтеграцію управління тестами і дефектами, де є тестові сценарії, результати тестування та дефекти об’єднані в єдиній системі, що зменшує час на перемикання між задачами;
- оптимізацію процесу, де завдяки кращій організації тестування (наприклад, автоматичним звітам) команда витрачає менше часу на адміністративні завдання.

Після реалізації комплексного підходу загальний час на тестування скорочується.

В цьому дослідженні загальний час на тестування за день складав 810 хвилин. Виконаємо розрахунок підвищення ефективності за допомогою порівняння показників до і після впровадження наступним чином:

- початковий час становив 1000 хвилин;

- час після впровадження становив: 810 хвилин.

Розрахунок економії часу відбувався наступним чином:

1000 хвилин – 810 хвилин = 190 хвилин

Відсоток підвищення ефективності:

(190 хвилин / 1000 хвилин) × 100% = 19%

Економія в 190 хвилин (або 19%) стала можливою завдяки автоматизації процесів на основі обліку проведення тестування програмного забезпечення. Це означає, що якщо раніше на реєстрацію одного дефекту витрачалося 10 хвилин, а за день виявлялося 20 дефектів, то автоматизація на основі запропонованого методу могла заощадити час на проведення тестових робіт, що в середньому дорівнює 20 × 10 = 200 хвилин. В даному випадку частина економії часу припадає саме на це.

Також, було відмічено зменшення часу, яке досягалось завдяки усуненню ручної реєстрації дефектів (економія часу на кожному дефекті). Це відбувалось завдяки більш швидшому доступу до інформації про тести та дефекти через інтегровану систему. Тому, краща організація тестів і швидший доступ до даних додатково скоротили час на інші завдання.

Таким чином, на даному прикладі показано випадок як команда тестувальників в ІТ-кампанії скоротила загальний час на тестування з 1000 хвилин до 810 хвилин завдяки впровадженню комплексного підходу до управління тестами і дефектами. Це призвело до підвищення ефективності виконання тестових робіт на 19%, що досягнуто за рахунок автоматизації реєстрації дефектів, інтеграції процесів і кращої організації роботи. Тому, такий запропонований підхід дозволяє команді швидше виявляти, фіксувати та вирішувати проблеми в програмному забезпеченні.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було комплексно досліджено проблематику організації процесу тестування програмного забезпечення в ІТ-компаніях, зокрема акцентовано увагу на обліку тестування, управлінні дефектами та створенні засобів звітності. У результаті проведеного аналізу та подальшої реалізації розробки досягнуто поставленої мети — створено функціональний, адаптивний та технічно обґрунтований програмний засіб, орієнтований на потреби тестувальної команди.

Перш за все, було здійснено аналітичний огляд сучасних методів тестування, який показав наявність широкої класифікації тестових підходів: ручне та автоматизоване тестування, тестування «чорної скриньки» та «білої скриньки», а також тестування на різних рівнях — модульному, інтеграційному, системному та приймальному. Було з’ясовано, що кожен із методів має як переваги, так і обмеження, і вибір того чи іншого методу залежить від типу проєкту, вимог до продуктивності, очікувань замовника і внутрішніх процесів розробки.

На основі аналізу відомих інструментів для автоматизації тестування (Selenium, Appium, JUnit, TestNG, Postman, JMeter) виявлено, що існує значна фрагментованість функціональних засобів. Інструменти орієнтовані на вузькі сфери застосування: веб-інтерфейси, мобільні додатки, API або навантаження. Це ускладнює формування єдиної екосистеми управління тестуванням у невеликих ІТ-компаніях, де обмежені ресурси не дозволяють впроваджувати багато інструментів одночасно. Це стало підґрунтям для обґрунтування необхідності розробки власного інтегрованого засобу, який би об’єднував функції фіксації тестів, обліку дефектів, звітності та базової аналітики.

Особливу увагу було приділено сучасним тенденціям в управлінні якістю ПЗ, зокрема впровадженню підходів Agile, DevOps, безперервного тестування та використання машинного навчання для оптимізації процесів. З’ясовано, що ІТ-компанії в умовах високої конкурентності та динамічних ринків прагнуть скорочення часу розробки, одночасно підвищуючи якість релізів. Це обумовлює потребу в засобах, що дозволяють ефективно управляти інформацією про тести в

реальному часі, забезпечуючи високу швидкість комунікації між тестувальниками, розробниками та менеджментом.

У рамках постановки задачі було визначено функціональні та нефункціональні вимоги до розроблюваного ПЗ. Було з’ясовано, що основними категоріями є: управління тест-кейсами, ведення звітності, моніторинг дефектів, підтримка типів тестування (ручне, автоматизоване, регресійне, навантажувальне) та розмежування прав доступу для різних ролей (тестувальник, розробник). Формалізовано ключові операції для кожної з підсистем та сформовано модель сценаріїв використання.

У процесі проєктування програмного засобу реалізовано декілька модулів:

– модуль управління тест-кейсами, що дозволяє створювати, редагувати, архівувати та фіксувати результати тестування. Забезпечено механізм збереження історії запусків та результатів;

– модуль дефектів, у якому реалізовано реєстрацію, коментування, відстеження статусу та пріоритету помилок. Передбачено трасування дефекту до конкретного тесту;

– модуль звітності, який дозволяє формувати зведені та деталізовані звіти про результати тестів і динаміку роботи над дефектами. Підтримано експорт у PDF та XLS-форматах;

– модуль розмежування доступу, який реалізує контроль доступу відповідно до ролей користувачів.

Під час побудови архітектури системи використовувались діаграми UML: діаграма варіантів використання (use-case), діаграма активностей, діаграма послідовності та діаграма компонентів. Це дозволило структурувати логіку взаємодії між користувачами та системою, забезпечити цілісність програмної моделі, уникнути суперечностей та забезпечити модульність рішення.

В рамках реалізації програмного засобу було забезпечено гнучкість та розширюваність системи, що дозволяє її подальший розвиток — наприклад, інтеграцію з інструментами Jira, GitLab, Jenkins, підключення API-інтерфейсів та формування кастомних метрик.

Також, було здійснено тестування функціональності системи, яке підтвердило її стабільну роботу при типовому навантаженні до 100 обробок/секунду. Забезпечено безпомилкове виконання базових операцій для понад 50 одночасних користувачів. Експериментальна перевірка засвідчила покращення керованості тестувальним процесом, скорочення часу на звітність до 40% у порівнянні з ручною обробкою таблиць, а також підвищення прозорості у взаємодії розробників і тестувальників.

Таким чином, виконана робота має практичне значення для малих та середніх IT-компаній, які прагнуть впровадити власну систему керування тестуванням без залучення складних і дорогих інструментів. Розроблений програмний засіб демонструє приклад сучасного підходу до розв’язання задач управління якістю ПЗ, заснованого на чіткій структуризації процесів, автоматизації рутинних операцій та орієнтації на гнучкість впровадження.

Отже, розроблена система повністю відповідає поставленій меті та вирішує основні завдання, зокрема:

- дозволяє тестувальникам ефективно керувати сценаріями та обліком результатів;

- забезпечує розробників повним та актуальним доступом до інформації про дефекти;

- генерує звітність для керівництва, що дає змогу приймати обґрунтовані рішення;

- створює базу для подальшого розширення — інтеграції з CI/CD, впровадження модулів предиктивної аналітики, використання метрик покриття та якості.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Maksymenko O., Khoshaba O. Development of a software tool for testing monitoring in an IT company /Матеріали XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. «Стан, досягнення і перспективи інформаційних систем і технологій». Одеса, 17 - 18 квітня 2025 р. - Одеса, Видавництво ОНТУ, 2025 р. - с.83-84.

2. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2017. – 284 с.

3. Коновалов В.С., Радоуцький К.Є. Сучасні принципи і методи проектування програмного забезпечення. Консп. лекцій. – Харків: УкрДАЗТ, 2015. – 109 с.

4. Жигир В. Основні підходи до ефективного тестування ПЗ // DOU. – 2024. [Електронний ресурс] – Режим доступу : URL: https://dou.ua/forums/topic/48845/ – Дата доступу до ресурсу: 23.04.2025.

5. Принципи тестування: їх концепції та підходи // FoxmindEd. – 2023. [Електронний ресурс] – Режим доступу : URL: https://foxminded.ua/pryntsypy-testuvannia/foxminded.ua – Дата доступу до ресурсу: 23.04.2025.

6. 10 найкращих інструментів для QA Automation у 2025 році // Robot Dreams. – 2025. [Електронний ресурс] – Режим доступу : URL: https://robotdreams.cc/uk/blog/630-10-best-tools-for-qa-automation-in-2025 – Дата доступу до ресурсу: 23.04.2025.

7. Бородкіна І.Л., Бородкін Г.О. Інженерія програмного забезпечення. Навчальний посібник. – Київ: Національний університет біоресурсів та природокористування України, 2018. – 284 с.

8. Дідковська М.В., Тимошенко Ю.О. Тестування: Основні визначення, аксіоми та принципи. Текст лекцій. Частина І. – Київ: НТУУ "КПІ", 2019. – 150 с.

9. QATestLab. Ручне тестування vs Автоматизація. – 2023. – URL: https://training.qatestlab.com/blog/technical-articles/manual-testing-vs-automation-testing/ – Дата доступу до ресурсу: 27.04.2025.

10. ZAPTEST. Автоматизація тестування програмного забезпечення. – 2022. – URL: https://www.zaptest.com/uk/повний-посібник-з-автоматизації-тест

11. Highload. Бібліотека QA: 8 книг із тестування програмного забезпечення. – 2021. – URL: https://highload.tech/uk/biblioteka-qa-8-knig-iz-testuvannya-programnogo-zabezpechennya/ – Дата доступу до ресурсу: 27.04.2025.

12. DOU. «Мистецтво тестування програмного забезпечення»: огляд книги. – 2023. – URL: https://dou.ua/forums/topic/42441/ – Дата доступу до ресурсу: 27.04.2025.

13. QATestLab. "Яка різниця між Black Box & White Box Testing?" – 2022. URL: https://training.qatestlab.com/blog/technical-articles/whats-the-difference-between-black-box-white-box-testing – Дата доступу до ресурсу: 27.04.2025.

14. QALight. "White/black/grey box-тестування." – 2021. URL: https://qalight.ua/baza-znaniy/white-black-grey-box-testuvannya/qalight.ua – Дата доступу до ресурсу: 27.04.2025.

15. Guru99. "Різниця між чорним Box і Білий Box Тестування." – 2023. URL: https://www.guru99.com/uk/back-box-vs-white-box-testing.html – Дата доступу до ресурсу: 27.04.2025.

16. Zaptest. "Тестування 'чорної скриньки' - що це таке, типи, процес." – 2022. URL: https://www.zaptest.com/uk/чорна_скринька – Дата доступу до ресурсу: 27.04.2025.

17. Mate Academy. "Black-Box тестування. Що таке тестування за методом 'чорної скриньки'?" – 2023. URL: https://mate.academy/blog/qa/black-box-testing – Дата доступу до ресурсу: 27.04.2025.

18. Zaptest. "Тестування 'білого ящика': Що це таке, як воно працює, виклики." – 2022. URL: https://www.zaptest.com/uk/білий_ящик – Дата доступу до ресурсу: 27.04.2025.

19. Guru99. "Рівні тестування в тестуванні програмного забезпечення." – 2024. URL: https://www.guru99.com/uk/levels-of-testing.html – Дата доступу до ресурсу: 27.04.2025.

20. Avada Media. "Все, що потрібно знати про тестування: рівні, типи, етапи та методи налагодження." – 2022. URL: https://avada-media.ua/services/vse-chto-nuzhno-znatpro-testirovaniye-urovni-tipy-etapy-i-metody-otladki/ – Дата доступу до ресурсу: 27.04.2025.

21. RoleCatcher. "Рівні тестування програмного забезпечення: Повний посібник з інтерв'ю на навички." – 2023. URL: https://rolecatcher.com/uk/skills/knowledge/information-and-communication-technologies/software-and-applications-development-and-analysis/ – Дата доступу до ресурсу: 27.04.2025.

22. QALearning. "Рівні тестування." – 2023. URL: https://qalearning.com.ua/theory/lectures/material/testing-levels/ – Дата доступу до ресурсу: 27.04.2025.

23. DevZone. "Які існують рівні тестування?" – 2023. URL: https://devzone.org.ua/qna/iaki-isnuiut-rivni-testuvannia – Дата доступу до ресурсу: 27.04.2025.

24. Selenium automates browsers [Електронний ресурс] – Режим доступу : URL: https://www.selenium.dev/ – Дата доступу до ресурсу: 27.04.2025.

25. Appium [Електронний ресурс] – Режим доступу : URL: https://appium.io/docs/en/latest/ – Дата доступу до ресурсу: 27.04.2025.

26. JUnit [Електронний ресурс] – Режим доступу : URL: https://junit.org/junit5/ – Дата доступу до ресурсу: 27.04.2025.

27. TestNG Documentation [Електронний ресурс] – Режим доступу : URL: https://testng.org/ – Дата доступу до ресурсу: 27.04.2025.

28. Postman [Електронний ресурс] – Режим доступу : URL: https://www.postman.com/ – Дата доступу до ресурсу: 27.04.2025.

29. Apache JMeter [Електронний ресурс] – Режим доступу : URL: https://jmeter.apache.org/ – Дата доступу до ресурсу: 27.04.2025.

30. Software testing in continuous delivery [Електронний ресурс] – Режим доступу : URL: https://www.atlassian.com/continuous-delivery/software-testing – Дата доступу до ресурсу: 28.04.2025.

31. Agile Testing Methodology - Best Practices [Електронний ресурс] – Режим доступу : URL: https://www.globalapptesting.com/the-ultimate-guide-to-agile-testing – Дата доступу до ресурсу: 28.04.2025.

32. Automated Software Testing Adoption and Trends [Електронний ресурс] – Режим доступу : URL: https://www.gartner.com/peer-community/oneminuteinsights/omi-automated-software-testing-adoption-trends-7d6 – Дата доступу до ресурсу: 27.04.2025.

33. Case Studies: Companies Succeeding with Latest Testing Technologies [Електронний ресурс] – Режим доступу : URL: https://analyticsweek.com/case-studies-companies-succeeding-with-latest-testing-technologies/ – Дата доступу до ресурсу: 27.04.2025.

34. Automation Testing Market [Електронний ресурс] – Режим доступу : URL: https://www.marketsandmarkets.com/Market-Reports/automation-testing-market-113583451.html – Дата доступу до ресурсу: 27.04.2025.

35. Zubair Khaliq, Sheikh Umar Farooq, Dawood Ashraf Khan [Електронний ресурс] – Режим доступу : URL: Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect [Електронний ресурс] – Режим доступу : URL: https://arxiv.org/abs/2201.05371 – Дата доступу до ресурсу: 27.04.2025.

36. Omdev Dahiya, Kamna Solanki, Amita Dhankhar. Risk-Based Testing: Identifying, Assessing, Mitigating & Managing Risks Efficiently in Software Testing International Journal of Advanced Research in Engineering and Technology (IJARET), 11(3), 2020, pp. 192-203.

37. Automated Software Testing Adoption and Trends [Електронний ресурс] – Режим доступу : URL: https://www.gartner.com/peer-community/oneminuteinsights/omi-automated-software-testing-adoption-trends-7d6 – Дата доступу до ресурсу: 27.04.2025.

92

ДОДАТОК А ТЕХНІЧНЕ ЗАВДАННЯ

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
д.т.н., проф.
О. Н. Романюк
"31" березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу
«Розробка програмного засобу для проведення тестування в ІТ-компанії»
за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської квалівікаційної роботи:
к.т.н., доц. каф. ПЗ Хошаба О.М.
"20" березня 2025 р.
Виконала: студентка гр.1ПІ-21б
Максименко О.В.
"20" березня 2025 р.

Вінниця - 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: Розробка програмного засобу для проведення тестування в ІТ-компанії.

Галузь застосування – розробка програмного засобу в галузі тестування.

2. Підстава для розробки.

Завдання на роботу, яке затверджене на засіданні кафедри програмного забезпечення – наказ №97 від « 20 » березня 2025 р.

3. Мета та призначення розробки.

Метою роботи є підвищення ефективності процесу тестування в ІТ-компанії шляхом розробки програмного засобу, який забезпечить повноцінний облік тестів.

Призначення роботи – розробка програмної додатку, що реалізує що виконує тестування програмного забезпечення в ІТ- компаніях.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Maksymenko O., Khoshaba O. Development of a software tool for testing monitoring in an IT company /Матеріали XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. «Стан, досягнення і перспективи інформаційних систем і технологій». Одеса, 17 - 18 квітня 2025 р. - Одеса, Видавництво ОНТУ, 2025 р. - с.83-84.

2. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2017. – 284 с.

5. Технічні вимоги

Кількість типів тестування: 4 (функціональне, автоматизоване, регресійне, навантажувальне); кількість основних ролей користувачів: 2 (тестувальник, розробник); кількість функціональних вимог: не більше 30; обсяг бази тест-кейсів у системі: до 10000 записів; максимальна кількість дефектів для обробки: до 3000; продуктивність системи: не менше 100 записів/секунду при фільтрації та звітності; підтримка одночасної роботи до 50 користувачів.

6. Конструктивні вимоги.

Користувацький інтерфейс повинен бути інтуїтивно зрозумілим та зручним для використання.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

a. пояснювальна записка до БКР;
b. технічне завдання;
c. лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз проблеми, обґрунтування актуальності розробки системи та постановка задач	25.03.2025-03.04.2025
2	Проектування модулів, розробка алгоритмів, структури та моделей з тестування в ІТ-компаніях	04.04.2025-14.04.2025
3	Вибір середовища та розробка програмного забезпечення з тестування в ІТ-компаніях	15.04.2025-05.05.2025
4	Тестування графічнох частини, впровадження програмної системи	06.05.2025-19.05.2025
5	Оформлення матеріалів до захисту БКР	20.05.2025-30.05.2025

10. Порядок контролю та прийняття.

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

ДОДАТОК Б ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НЯВНІСТЬ ЗАПОЗИЧЕНЬ

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка програмного засобу для проведення тестування в ІТ-компанії

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра прогармного збезпечення, ФІТКИ, 1ПІ-21б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 3,4 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

□ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

□ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ к.т.н., доц. каф. ПЗ Хошаба О.М.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Максименко О.В.
(підпис) (прізвище, ініціали)

ДОДАТОК В ЛІСТИНГ ПРОГРАМИ

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Модуль планування тестування</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Модуль планування тестування</h1>
    </header>
    <main>
        <section id="test-plan-form">
            <h2>Створення тест-плану</h2>
            <form id="plan-form">
                <label for="plan-name">Назва плану:</label>
                <input type="text" id="plan-name" required>

                <label for="start-date">Дата початку:</label>
                <input type="date" id="start-date" required>

                <label for="end-date">Дата завершення:</label>
                <input type="date" id="end-date" required>

                <label for="test-cases">Тест-кейси:</label>
                <textarea id="test-cases" rows="4" required></textarea>
```

```
            <button type="submit">Створити план</button>
        </form>
    </section>
    <section id="test-plans-list">
        <h2>Список тест-планів</h2>
        <ul id="plans-container"></ul>
    </section>
  </main>

  <script src="script.js"></script>
</body>
</html>
```

Програмний код (бекенд) модуля планування тестування

```
package org.vntu.testplan;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TestPlanApplication {
  public static void main(String[] args) {
    SpringApplication.run(TestPlanApplication.class, args);
  }
}

package org.vntu.testplan.controller;

import org.vntu.testplan.model.TestPlan;
import org.vntu.testplan.service.TestPlanService;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/test-plans")
public class TestPlanController {

    @Autowired
    private TestPlanService testPlanService;

    @PostMapping
    public ResponseEntity<TestPlan> createTestPlan(@RequestBody TestPlan testPlan) {
        return ResponseEntity.ok(testPlanService.saveTestPlan(testPlan));
    }

    @GetMapping
    public ResponseEntity<List<TestPlan>> getAllTestPlans() {
        return ResponseEntity.ok(testPlanService.getAllTestPlans());
    }
}

package org.vntu.testplan.model;

import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
```

```
@Table(name = "test_plans")
public class TestPlan {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private LocalDate startDate;

    @Column(nullable = false)
    private LocalDate endDate;

    @Column(nullable = false, columnDefinition = "TEXT")
    private String testCases;

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public LocalDate getStartDate() { return startDate; }
    public void setStartDate(LocalDate startDate) { this.startDate = startDate; }

    public LocalDate getEndDate() { return endDate; }
```

```
    public void setEndDate(LocalDate endDate) { this.endDate = endDate; }

    public String getTestCases() { return testCases; }
    public void setTestCases(String testCases) { this.testCases = testCases; }
}

package org.vntu.testplan.repository;

import org.vntu.testplan.model.TestPlan;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TestPlanRepository extends JpaRepository<TestPlan, Long> {
}

package org.vntu.testplan.service;

import org.vntu.testplan.model.TestPlan;
import org.vntu.testplan.repository.TestPlanRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class TestPlanService {

    @Autowired
    private TestPlanRepository testPlanRepository;
```

```
    public TestPlan saveTestPlan(TestPlan testPlan) {
        return testPlanRepository.save(testPlan);
    }

    public List<TestPlan> getAllTestPlans() {
        return testPlanRepository.findAll();
    }
}
```

Програмний код інтерфейсу модуля управління тест-кейсами

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Модуль управління тест-кейсами</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Модуль управління тест-кейсами</h1>
    </header>
    <main>
        <section id="test-case-form">
            <h2>Створення тест-кейсу</h2>
            <form id="case-form">
                <label for="case-name">Назва тест-кейсу:</label>
                <input type="text" id="case-name" required>
```

```
            <label for="assigned-to">Призначити тестувальнику:</label>
            <input type="text" id="assigned-to" required>

            <label for="status">Статус:</label>
            <select id="status">
                <option value="не виконано">Не виконано</option>
                <option value="в процесі">В процесі</option>
                <option value="пройдено">Пройдено</option>
                <option value="провалено">Провалено</option>
            </select>

            <button type="submit">Додати тест-кейс</button>
        </form>
    </section>
    <section id="test-cases-list">
        <h2>Список тест-кейсів</h2>
        <ul id="cases-container"></ul>
    </section>
</main>

<script src="script.js"></script>
</body>
</html>

body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f4f4f4;
}
```

```
header {
    background-color: #333;
    color: white;
    padding: 1rem;
    text-align: center;
}

main {
    width: 80%;
    margin: auto;
    padding: 1rem;
    background-color: white;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h2 {
    color: #333;
}

form {
    display: flex;
    flex-direction: column;
}

label {
    margin-top: 10px;
}

input, select {
```

```
    padding: 8px;
    margin-top: 5px;
}

button {
    margin-top: 10px;
    padding: 10px;
    background-color: #28a745;
    color: white;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #218838;
}

ul {
    list-style: none;
    padding: 0;
}

li {
    background: #ddd;
    padding: 10px;
    margin: 5px 0;
    display: flex;
    justify-content: space-between;
}
```

```
document.addEventListener("DOMContentLoaded", function () {
    const form = document.getElementById("case-form");
    const caseContainer = document.getElementById("cases-container");

    form.addEventListener("submit", function (event) {
        event.preventDefault();

        const caseName = document.getElementById("case-name").value;
        const assignedTo = document.getElementById("assigned-to").value;
        const status = document.getElementById("status").value;

        if (!caseName || !assignedTo) {
            alert("Будь ласка, заповніть всі поля!");
            return;
        }

        addTestCase(caseName, assignedTo, status);
        form.reset();
    });

    function addTestCase(name, assigned, status) {
        const li = document.createElement("li");
        li.innerHTML = `<strong>${name}</strong> (Призначено: ${assigned}, Статус:
${status})
                    <button onclick="deleteCase(this)">Видалити</button>`;
        caseContainer.appendChild(li);
    }

    window.deleteCase = function (button) {
        button.parentElement.remove();
```

```
    };
});
```

Програмний код (бекенд) модуля управління тест-кейсами (Test Case Management)

```
package org.vntu.testcase;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TestCaseApplication {
    public static void main(String[] args) {
        SpringApplication.run(TestCaseApplication.class, args);
    }
}

package org.vntu.testcase.controller;

import org.vntu.testcase.model.TestCase;
import org.vntu.testcase.service.TestCaseService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
```

```
@RequestMapping("/api/test-cases")
public class TestCaseController {

    @Autowired
    private TestCaseService testCaseService;

    @PostMapping
    public ResponseEntity<TestCase> createTestCase(@RequestBody TestCase testCase)
{
        return ResponseEntity.ok(testCaseService.saveTestCase(testCase));
    }

    @GetMapping
    public ResponseEntity<List<TestCase>> getAllTestCases() {
        return ResponseEntity.ok(testCaseService.getAllTestCases());
    }

    @PutMapping("/{id}")
    public ResponseEntity<TestCase> updateTestCase(@PathVariable Long id,
@RequestBody TestCase testCase) {
        return ResponseEntity.ok(testCaseService.updateTestCase(id, testCase));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteTestCase(@PathVariable Long id) {
        testCaseService.deleteTestCase(id);
        return ResponseEntity.noContent().build();
    }
}
```

```
package org.vntu.testcase.model;

import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "test_cases")
public class TestCase {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String assignedTo;

    @Column(nullable = false)
    private String status;

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getAssignedTo() { return assignedTo; }
```

```
    public void setAssignedTo(String assignedTo) { this.assignedTo = assignedTo; }

    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }
}

package org.vntu.testcase.repository;

import org.vntu.testcase.model.TestCase;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TestCaseRepository extends JpaRepository<TestCase, Long> {
}

package org.vntu.testcase.service;

import org.vntu.testcase.model.TestCase;
import org.vntu.testcase.repository.TestCaseRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class TestCaseService {

    @Autowired
    private TestCaseRepository testCaseRepository;
```

```
    public TestCase saveTestCase(TestCase testCase) {
        return testCaseRepository.save(testCase);
    }

    public List<TestCase> getAllTestCases() {
        return testCaseRepository.findAll();
    }

    public TestCase updateTestCase(Long id, TestCase testCase) {
        if (testCaseRepository.existsById(id)) {
            testCase.setId(id);
            return testCaseRepository.save(testCase);
        }
        throw new RuntimeException("TestCase not found");
    }

    public void deleteTestCase(Long id) {
        testCaseRepository.deleteById(id);
    }
}
```

Програмний код інтерфейсу (фронтенд) модуля виконання тестів (Test Execution)

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Модуль виконання тестів</title>
```

```
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Модуль виконання тестів</h1>
    </header>
    <main>
        <section id="test-execution">
            <h2>Виконання тест-кейсів</h2>
            <ul id="test-cases-container"></ul>
        </section>
    </main>

    <script src="script.js"></script>
</body>
</html>

body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f4f4f4;
}

header {
    background-color: #333;
    color: white;
    padding: 1rem;
    text-align: center;
}
```

```
main {
    width: 80%;
    margin: auto;
    padding: 1rem;
    background-color: white;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h2 {
    color: #333;
}

ul {
    list-style: none;
    padding: 0;
}

li {
    background: #ddd;
    padding: 10px;
    margin: 5px 0;
    display: flex;
    flex-direction: column;
}

button {
    margin-top: 10px;
    padding: 10px;
    background-color: #007bff;
```

```
    color: white;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #0056b3;
}

document.addEventListener("DOMContentLoaded", function () {
    const testCases = [
        { id: 1, name: "Перевірка логіну", status: "не виконано" },
        { id: 2, name: "Додавання товару у кошик", status: "не виконано" },
        { id: 3, name: "Оформлення замовлення", status: "не виконано" }
    ];

    const testContainer = document.getElementById("test-cases-container");

    testCases.forEach(testCase => {
        const li = document.createElement("li");
        li.innerHTML = `<strong>${testCase.name}</strong> (Статус: <span id="status-
${testCase.id}">${testCase.status}</span>)<br>
                <button                    onclick="updateStatus(${testCase.id},
'пройдено')">Пройдено</button>
                <button                    onclick="updateStatus(${testCase.id},
'провалено')">Провалено</button>`;
        testContainer.appendChild(li);
    });

    window.updateStatus = function (id, newStatus) {
```

```
        document.getElementById(`status-${id}`).innerText = newStatus;
    };
});
```

Програмний код (бекенд) модуля виконання тестів (Test Execution)

```
package org.vntu.testexecution;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TestExecutionApplication {
    public static void main(String[] args) {
        SpringApplication.run(TestExecutionApplication.class, args);
    }
}

package org.vntu.testexecution.controller;

import org.vntu.testexecution.model.TestExecution;
import org.vntu.testexecution.service.TestExecutionService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/test-executions")
```

```
public class TestExecutionController {

    @Autowired
    private TestExecutionService testExecutionService;

    @PostMapping
    public ResponseEntity<TestExecution> executeTestCase(@RequestBody
TestExecution testExecution) {
        return ResponseEntity.ok(testExecutionService.saveTestExecution(testExecution));
    }

    @GetMapping
    public ResponseEntity<List<TestExecution>> getAllTestExecutions() {
        return ResponseEntity.ok(testExecutionService.getAllTestExecutions());
    }
}

package org.vntu.testexecution.model;

import jakarta.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "test_executions")
public class TestExecution {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(nullable = false)
    private String testCaseName;

    @Column(nullable = false)
    private String tester;

    @Column(nullable = false)
    private String status;

    @Column(columnDefinition = "TEXT")
    private String comments;

    @Column(nullable = false)
    private LocalDateTime executionTime;

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getTestCaseName() { return testCaseName; }
    public void setTestCaseName(String testCaseName) { this.testCaseName = testCaseName; }

    public String getTester() { return tester; }
    public void setTester(String tester) { this.tester = tester; }

    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }

    public String getComments() { return comments; }
```

```
    public void setComments(String comments) { this.comments = comments; }

    public LocalDateTime getExecutionTime() { return executionTime; }
    public void setExecutionTime(LocalDateTime executionTime) { this.executionTime =
executionTime; }
}

package org.vntu.testexecution.repository;

import org.vntu.testexecution.model.TestExecution;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TestExecutionRepository extends JpaRepository<TestExecution, Long>
{
}

package org.vntu.testexecution.service;

import org.vntu.testexecution.model.TestExecution;
import org.vntu.testexecution.repository.TestExecutionRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.List;

@Service
public class TestExecutionService {
```

```
    @Autowired
    private TestExecutionRepository testExecutionRepository;

    public TestExecution saveTestExecution(TestExecution testExecution) {
        testExecution.setExecutionTime(LocalDateTime.now());
        return testExecutionRepository.save(testExecution);
    }

    public List<TestExecution> getAllTestExecutions() {
        return testExecutionRepository.findAll();
    }
}
```

ДОДАТОК Г ІЛЮСТРАТИВНА ЧАСТИНА

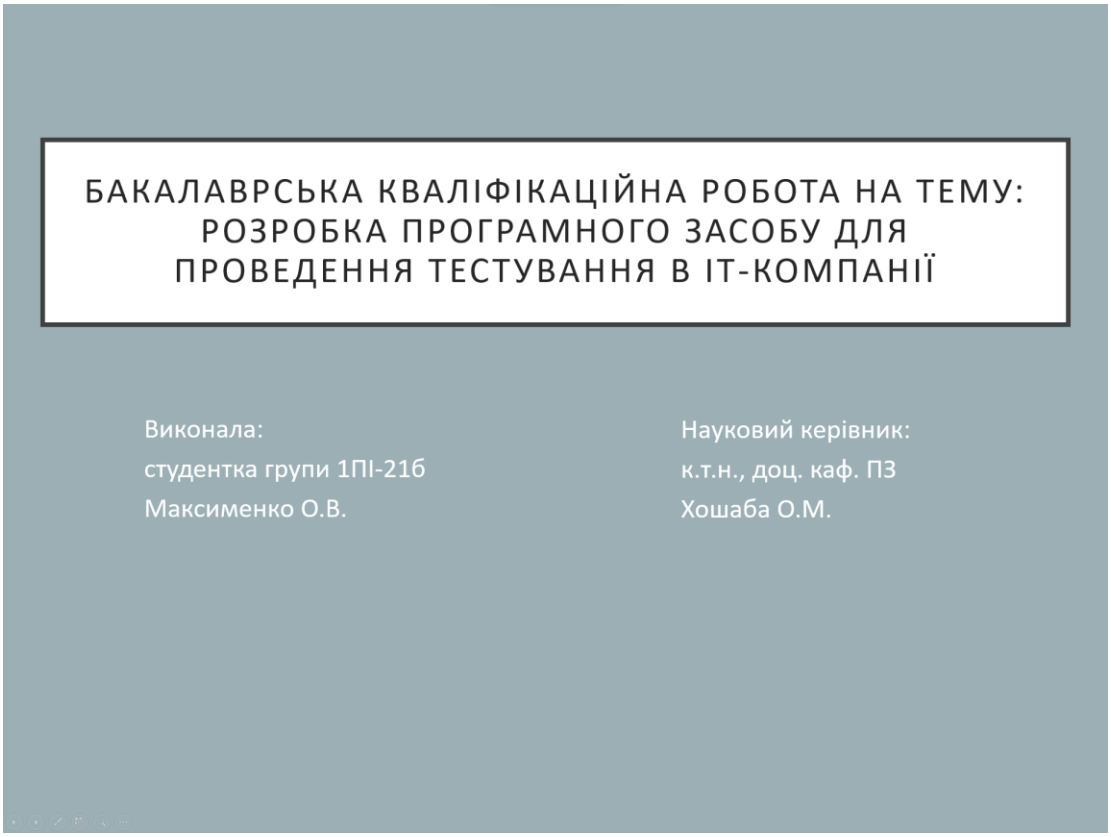

Рисунок Д.1 – Слайд презентації 1

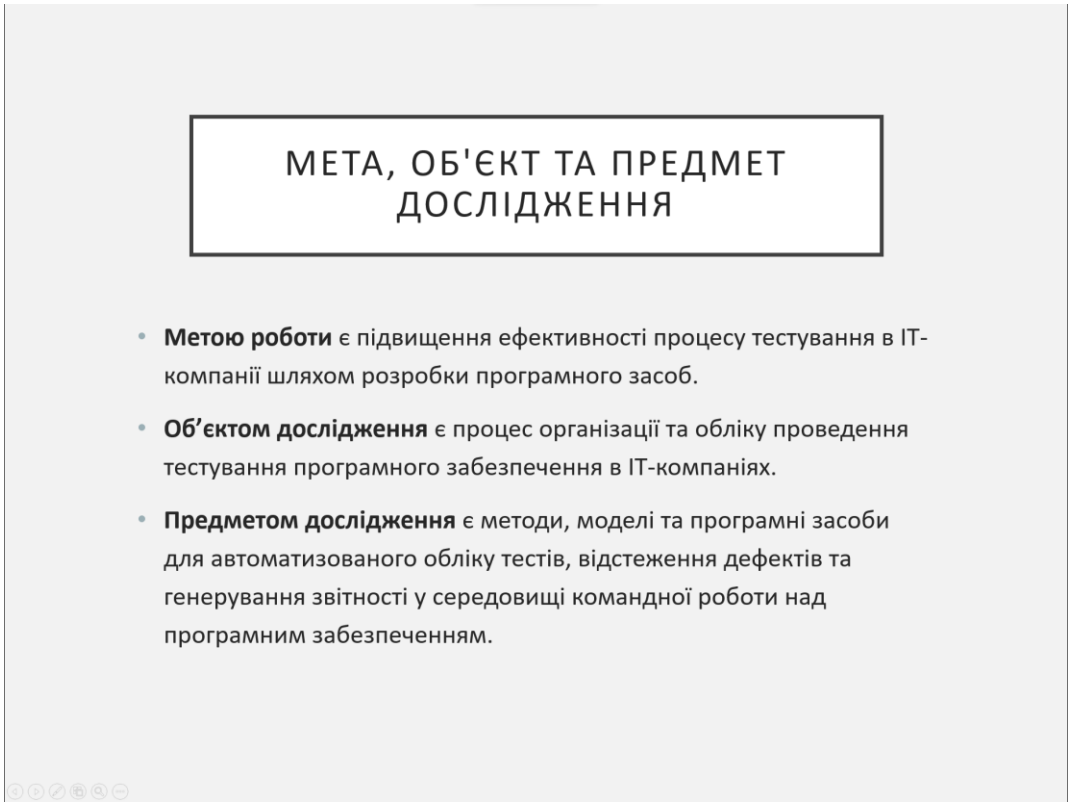

Рисунок Д.2 – Мета, об'єкт та предмет дослідження 2

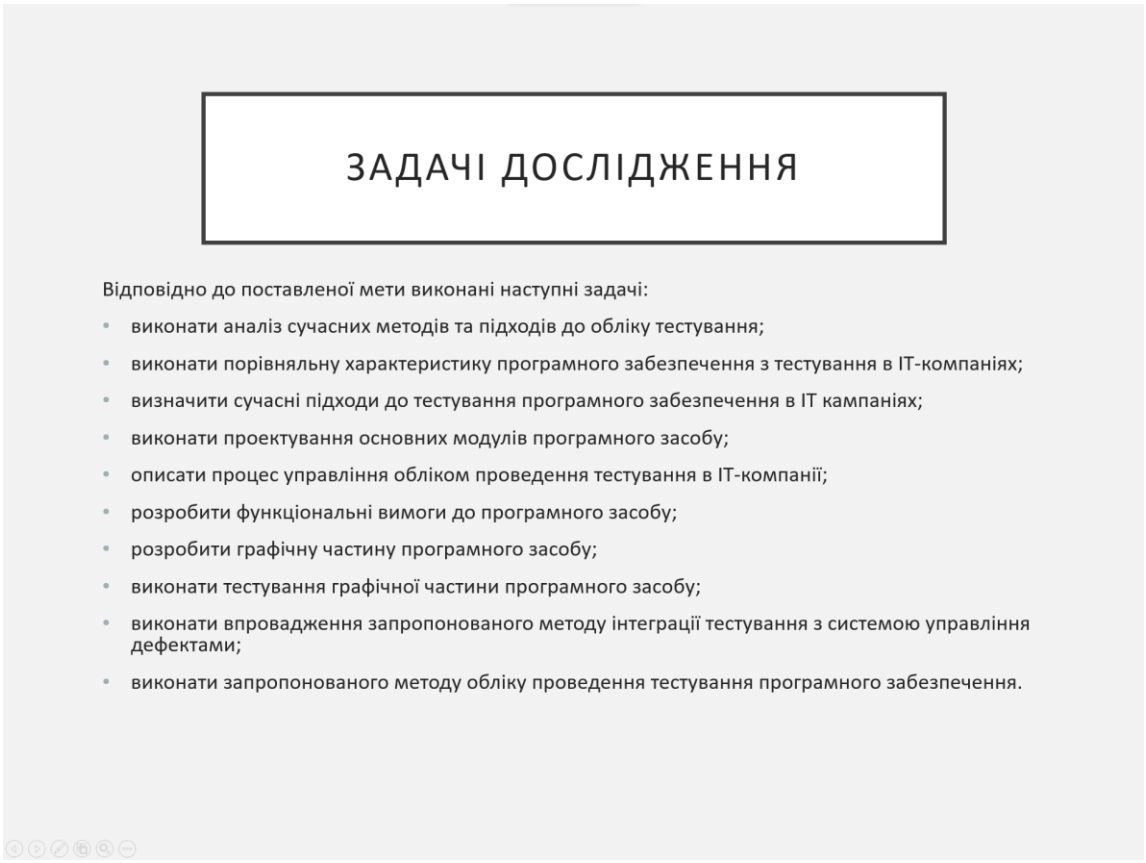

Рисунок Д.3 – Задачі досліження

Рисунок Д.4 – Актуальність теми дослідження

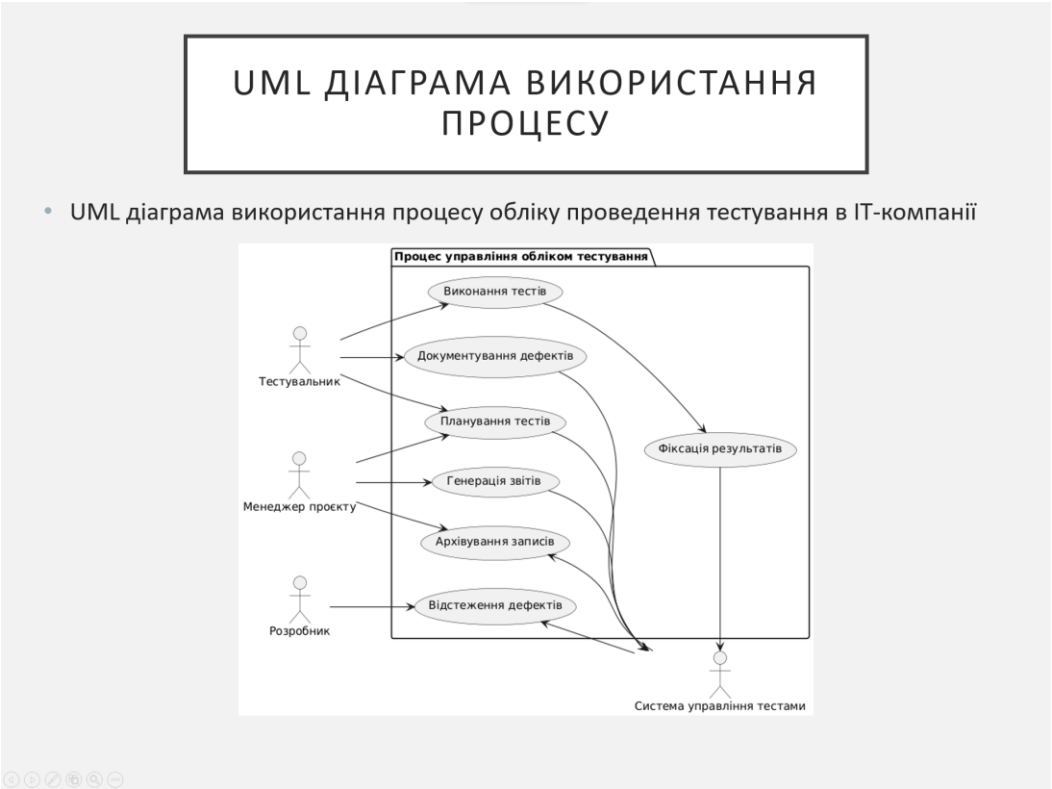

Рисунок Д.5 – UML діаграма використання процесу обліку проведення тестування в IT-компанії

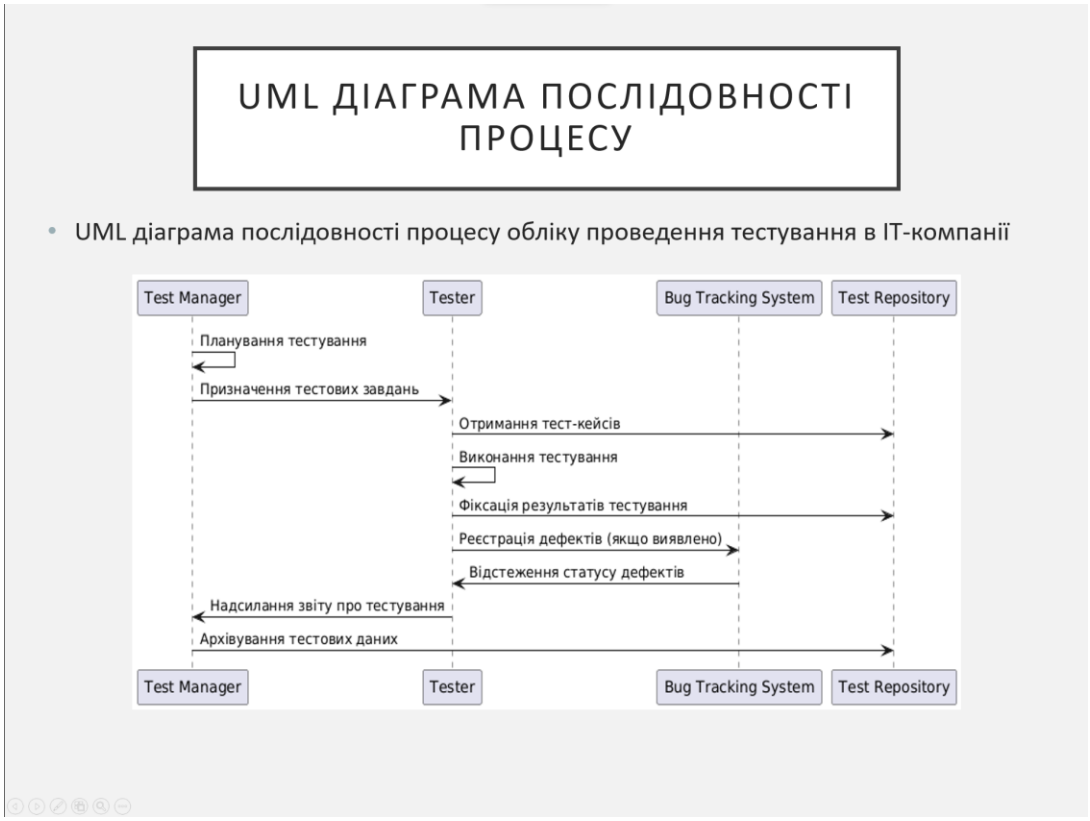

Рисунок Д.6 – UML діаграма послідовності процесу обліку проведення тестування в IT-компанії

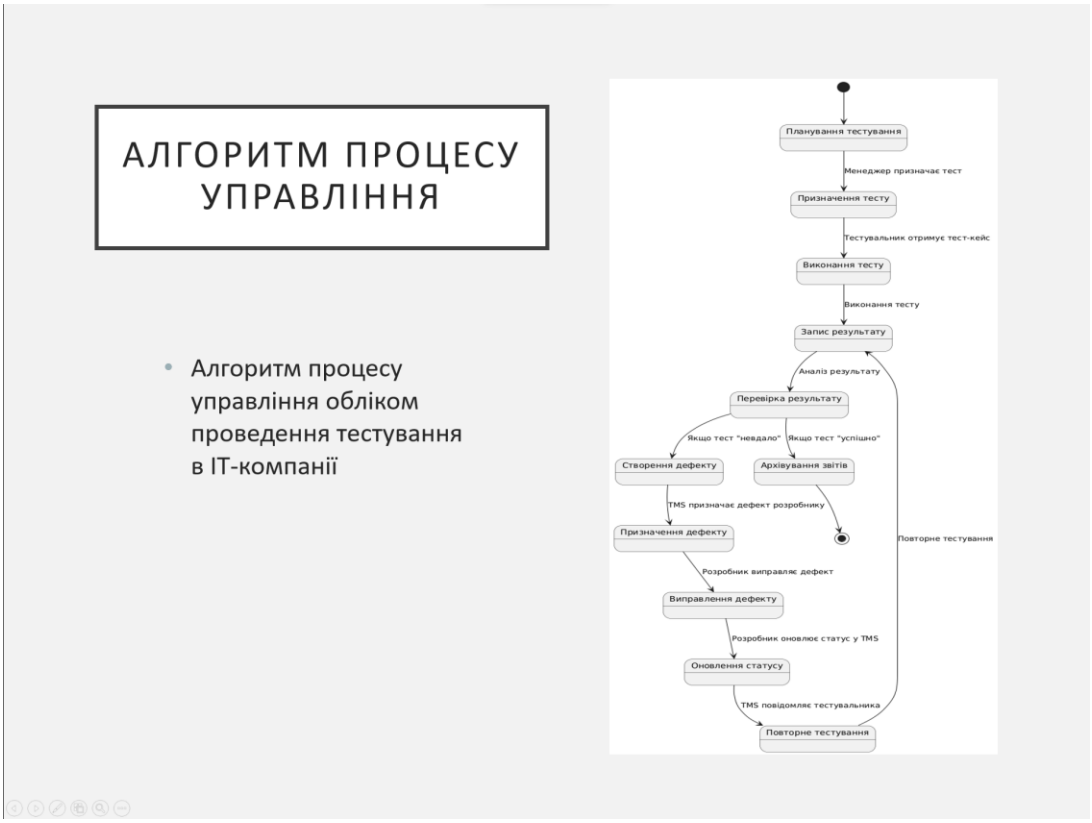

Рисунок Д.7 – Алгоритм процесу управління обліком проведення тестування в IT-компанії

ПОРІВНЯННЯ КЛЮЧОВИХ ФУНКЦІЙ МОДУЛЯ ПЛАНУВАННЯ ТЕСТУВАННЯ

Функція	Опис
Створення планів тестування	Визначення цілей, обсягу, стратегії, графіку, ресурсів та ризиків.
Визначення обсягу тестування	Вибір тест-кейсів на основі критеріїв, додавання до плану.
Встановлення термінів	Призначення дедлайнів для тест-кейсів.
Встановлення пріоритетів	Призначення рівнів важливості (високий, середній, низький).
Призначення тестувальників	Розподіл тест-кейсів між тестувальниками.
Планування середовища	Визначення вимог до обладнання, програмного забезпечення.
Управління ризиками	Ідентифікація ризиків та розробка планів мінімізації.
Пошук і фільтрація	Швидкий пошук планів за критеріями, збереження запитів.
Співпраця	Коментарі, вкладення, сповіщення.
Інтеграція	З дефект-трекінгом, версією, CI/CD.
Звітність	Звіти про статус плану, виконаних тест-кейсів.

Рисунок Д.8 – Порівняння ключових функцій модуля планування тестування

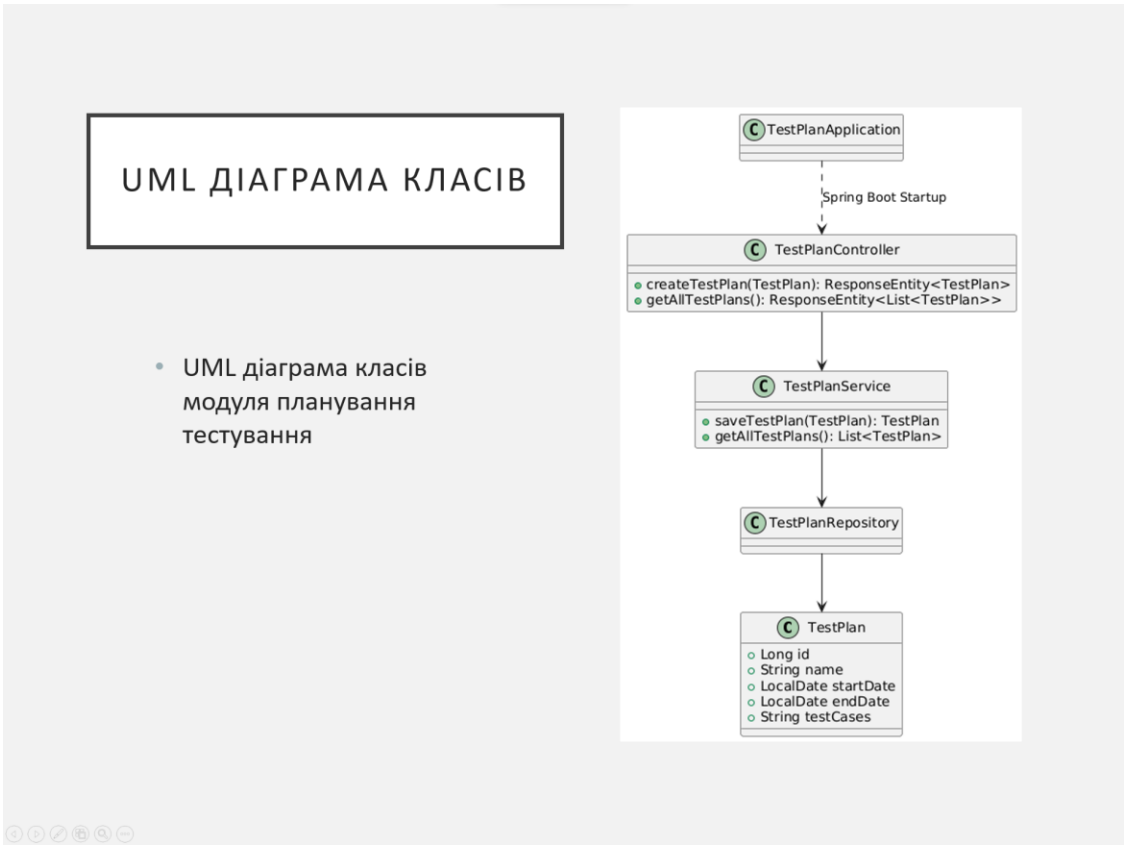

Рисунок Д.9 – UML діаграма класів модуля планування тестування

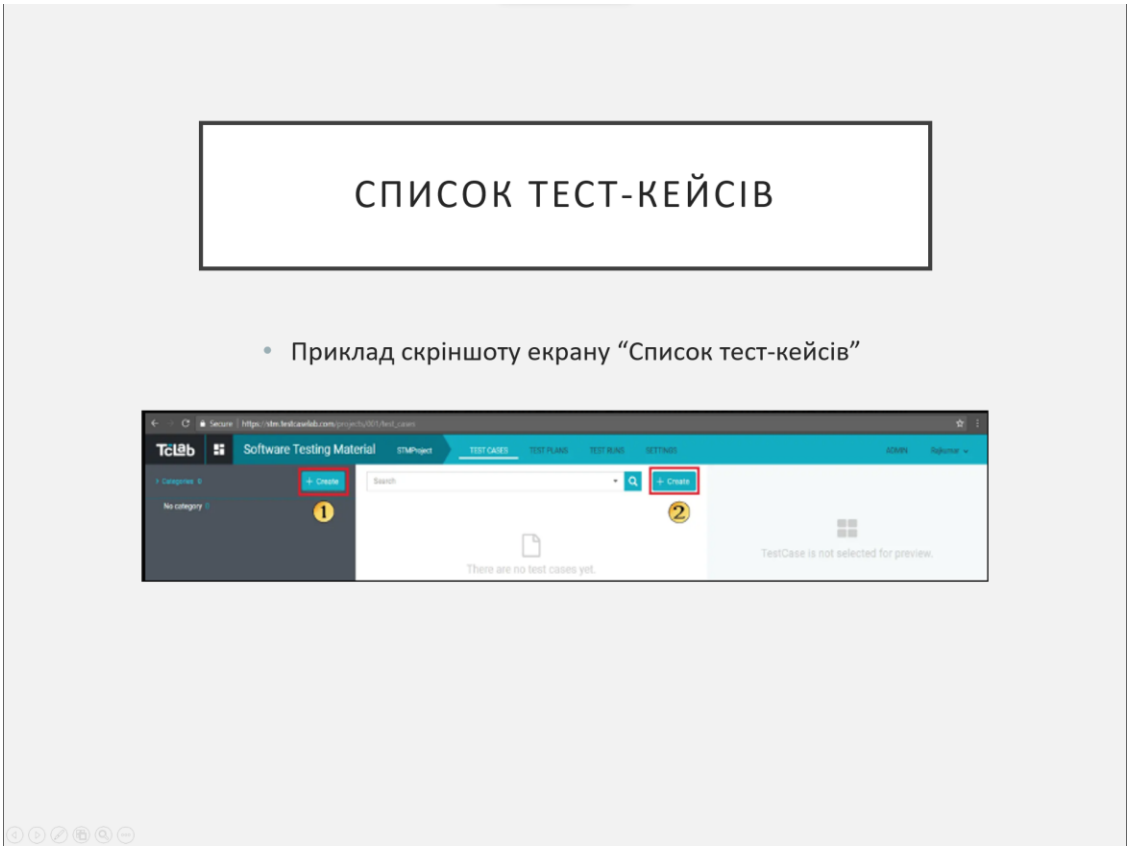

Рисунок Д.10 –Список тест-кейсів

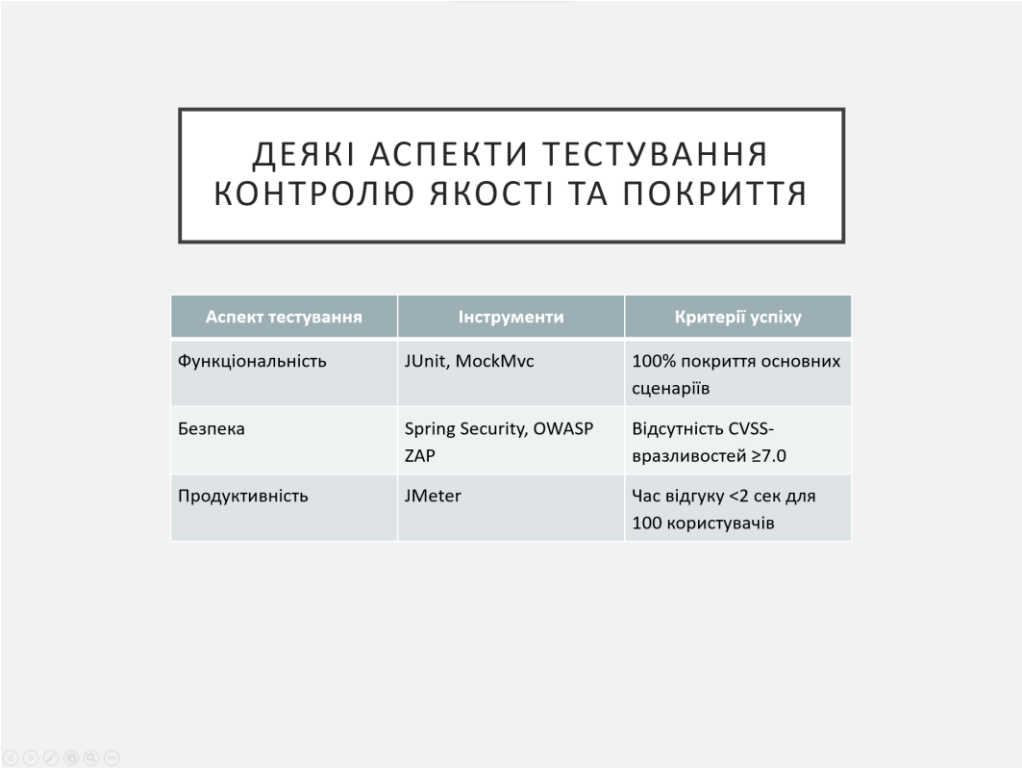

Рисунок Д.11 – Деякі аспекти тестування контролю якості та покриття

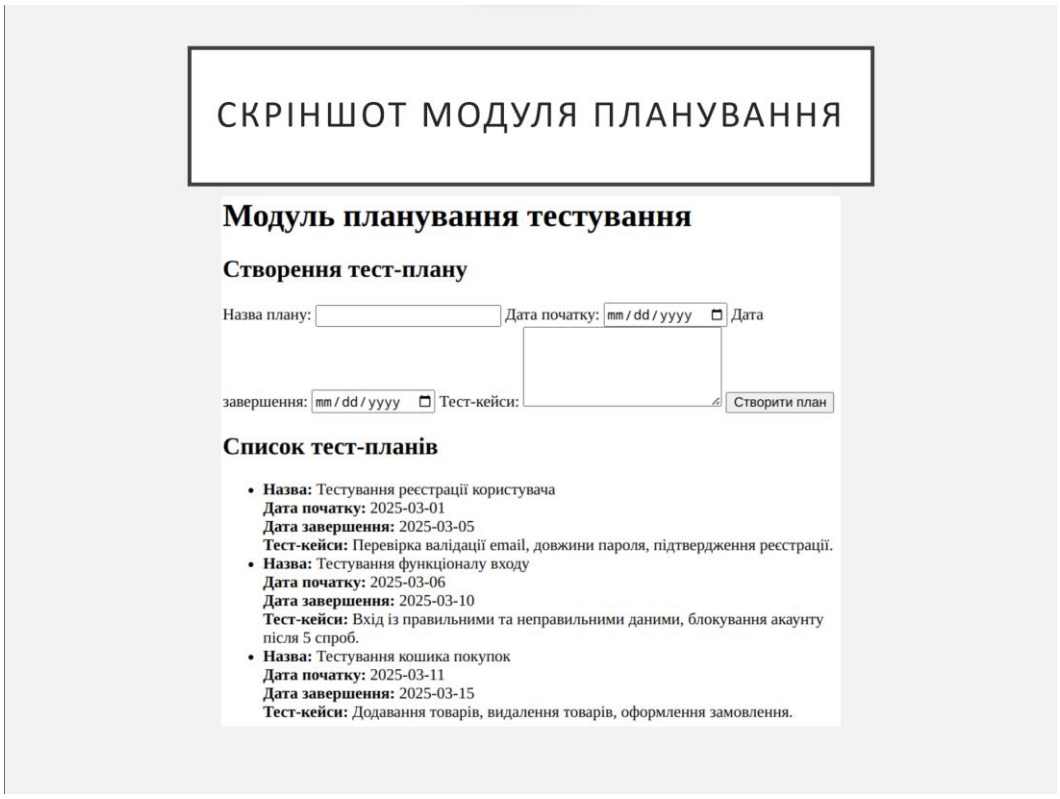

Рисунок Д.12 – Скріншот модуля планування

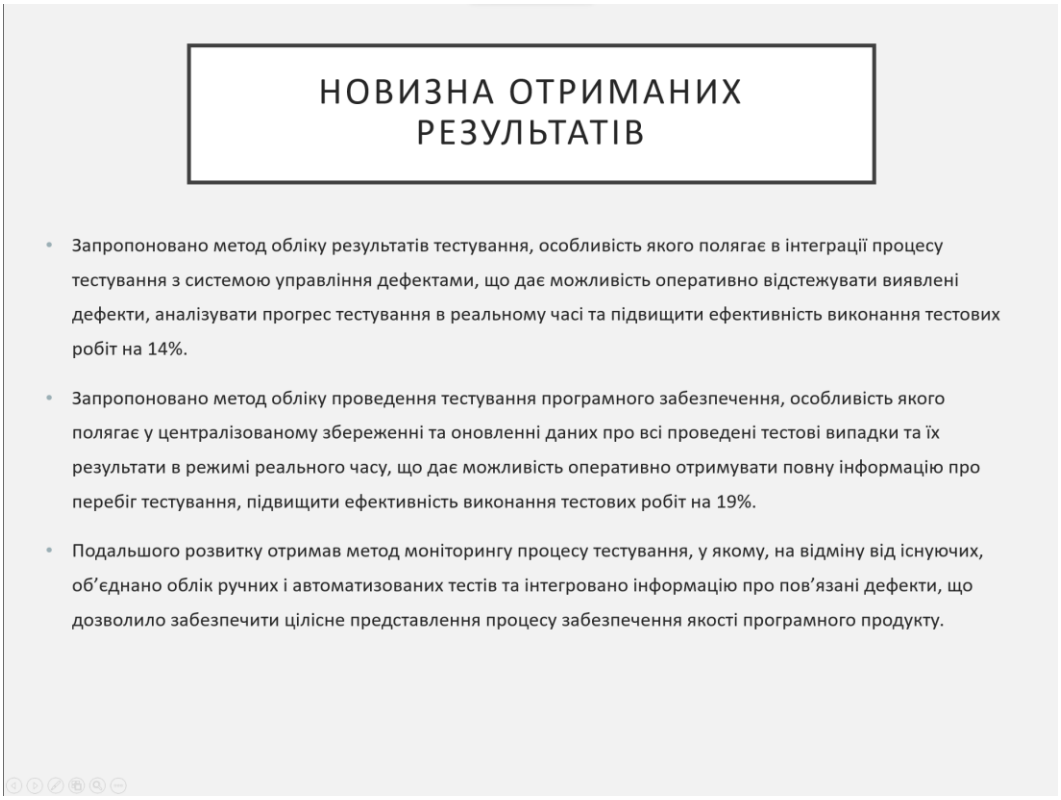

Рисунок Д.13 – Новизна отриманих результатів:

Рисунок Д.14 – Висновки:

Рисунок Д.15 – Заключний слайд