МІНІСТЕРСТВО ВНУТРІШНІХ СПРАВ УКРАЇНИ
ЛЬВІВСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ ВНУТРІШНІХ СПРАВ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ УПРАВЛІННЯ, ПСИХОЛОГІЇ ТА БЕЗПЕКИ

Кафедра інформаційних технологій

МОДЕЛЮВАННЯ ПРОЦЕСУ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота
здобувача вищої освіти
Валерія Головчанського

Науковий керівник:
доцент, кандидат технічних наук
Любомир ФЛУД

Рецензент:

_____
вчене звання, науковий ступінь

_____
(Ім’я ПРІЗВИЩЕ рецензента)

Кваліфікаційна робота допущена до захисту
«___» _____ 2026 р., протокол № ____

Завідувач кафедри інформаційних технологій
_____ Олег ЗАЧЕК
(підпис)

Львів
2026

АНОТАЦІЯ

Головчанський В. Моделювання процесу тестування програмного забезпечення. – Рукопис.

Дослідження на здобуття освітнього ступеня «бакалавр» за спеціальністю 126 «Інформаційні системи та технології». – Львівський державний університет внутрішніх справ, МВС України, Львів, 2026.

Бакалаврська робота присвячена створенню моделей, застосування яких сприятиме забезпеченню якості проектування програмного забезпечення, важливою складовою якого є моделювання процесу тестування програмних компонент та операційної системи в цілому. В ході виконання бакалаврської роботи виокремлено фактори впливу на якість програмного забезпечення, створено графічну модель зв'язків між факторами у вигляді семантичної мережі, що стала підставою синтезування моделі пріоритетного впливу факторів на якість програмного забезпечення. Визначальним фактором встановлено тестування програмного забезпечення, що обумовило визначення методів та засобів розроблення автоматизованого тестування програмних компонент та автоматизацію процесу створення звіту програмного засобу. У результаті виконання роботи створено програмний продукт, який перевіряє наявність програмних компонент на певній операційній системі, та забезпечує автоматизоване підготування звіту.

Ключові слова: програмне забезпечення, фактор, семантична мережа, автоматизована система, тестування, операційна система, програмний продукт.

ABSTRACT

Holovchanskyi V. Modeling the Software Testing Process. – Manuscript.

Research for obtaining a bachelor's degree in specialty 126 «Information systems and technologies». – Lviv State University of Internal Affairs, MIA of Ukraine, Lviv, 2026.

The bachelor's qualifying work is devoted to the creation of models that contribute to ensuring the quality of software design, an important part of which is the modeling of the software components and operating system testing process as a whole.

In the course of the work, the factors affecting software quality were identified, and a graphical model of the relationships between these factors was developed in the form of a semantic network. This model served as the basis for synthesizing a priority influence model of factors on software quality. Software testing was determined as the key factor, which predetermined the selection of methods and tools for automated testing of software components and automation of the software report generation process.

As a result, a software product was created that verifies the presence of software components on a particular operating system and ensures the automated preparation of reports.

Keywords: software, factor, semantic network, automated system, testing, operating system, software product.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПЗ	Програмне забезпечення
AQA	Automation Quality Assurance (Автоматизоване забезпечення якості)
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
OS	Operating System (Операційна система)
QA	Quality Assurance (Забезпечення якості)
QC	Quality Control (Контроль якості)
SDLC	Software Development Life Cycle (Життєвий цикл розробки програмного забезпечення)
STLC	Software Testing Life Cycle (Життєвий цикл тестування програмного забезпечення)
UML	Unified Modeling Language (Уніфікована мова моделювання)

ЗМІСТ

ВСТУП

В умовах швидкого розвитку інформаційних технологій важливим завданням є забезпечення стабільності та відповідності компонентів операційного середовища вимогам програмного забезпечення. Багато інформаційних систем вимагають наявності певних сервісів, пакетів та конфігурацій операційної системи. Відсутність або неправильна робота таких компонентів може призвести до збоїв у роботі ПЗ.

У роботі програмних компонентів та операційної системи існує поняття залежностей. Навіть якщо програма, чи операційна система написана правильно, це не гарантує її коректної роботи, так як зазвичай всі компоненти, так чи інакше, пов’язанні між собою. І відсутність хоча б одного програмного компоненту, може викликати помилку всієї роботи системи. І не достатньо тільки перевірити чи усі компоненти встановлені, а також який з них відсутній, і чи операційна система має змогу встановити його. Також необхідно створювати звіт, у якому буде це все структурно вказано, для якомога швидшого вирішення проблеми. Особливо актуально це в великих компаніях які розробляють всій власний продукт, і де час дуже дорогоцінний, так як в компанії йде не тільки витрати на те, що її продукт не працює, а також і на працівників, які очікують результатів перевірок.

Мета дослідження – створення моделей, застосування яких сприятиме забезпеченню якості програмного забезпечення, важливою складовою якого є моделювання процесу тестування програмних компонент та програмного продукту в цілому.

Для досягнення мети проєкту необхідно виконати такі **завдання**:

– Провести аналіз предметної області, методологій розроблення програмного забезпечення та принципів тестування;

– Виокремити фактори впливу на якість програмного забезпечення та розробити семантичну мережу зв’язків між ними;

– Спроєктувати матрицю попарних порівнянь факторів та розрахувати вагові значення пріоритетів факторів;

– Сформувати вимоги до інформаційної системи;

– Побудувати комплекс UML-діаграм;

– Розробити програмний комплекс згідно вимог;

– Провести налаштування тестового середовища та виконання тесту;

Об’єктом дослідження процес тестування конфігурації операційної системи, що реалізує перевірку ключових системних служб, компонентів та мережевої доступності з використанням фреймворку pytest та інструменту Allure.

Предметом дослідження виступають методи, алгоритми та програмні засоби.

У роботі використано комплекс **методів** що поєднує теоретичні та практичні методи. Теоретичну основу склали методи системного аналізу, структурно-функціонального моделювання та аналізу науково-технічної літератури. Практична частина базується на експериментальному методі, методі програмної реалізації та автоматизованого тестування.

Практична значущість полягає в автоматизації процесу перевірки відповідності мінімальних системних вимог операційних систем Windows та Linux. Запропоноване рішення дозволить перевірити наявність необхідних програмних компонент для коректної роботи як сторонніх програм, так і операційної системи, з можливістю генерації звіту виконаної роботи.

Структура роботи. Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел. Обсяг основного тексту роботи складає 52 сторінок, 37 рисунків, 10 таблиць і 26 бібліографічних джерела. Загальний обсяг роботи – 61 сторінок.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ВІДОМОСТІ ТА ПРИНЦИПИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Аналіз предметної області

Життєвий цикл створення програмного забезпечення (Software Development Life Cycle, SDLC) це впорядкована послідовність процесів, що охоплює всі етапи розроблення програмного продукту – від формування вимог до підготовки підсумкової документації та оцінювання результатів виконаної роботи. Залежно від обраної методології кількість і зміст етапів можуть відрізнятися, але загальна логіка процесу залишається незмінною: кожен наступний крок виконується лише після завершення попереднього [1, 2].

У межах процесу тестування особливе значення мають вхідні та вихідні критерії. Вхідний критерій визначає умови, за яких дозволяється розпочати виконання певного етапу, тоді як вихідний підтверджує його успішне завершення. Наприклад, проведення тестування можливе лише після підготовки працездатного тестового середовища, яке є результатом попередньої стадії.

Аналіз вимог є першим етапом дослідження та оцінювання вимог до програмного продукту. Формулювання вимог повинні бути однозначними, зрозумілими для всіх учасників проєкту та не містити суперечностей [1,3]. Нечіткі визначення на кшталт «система має бути зручною» або «пароль повинен бути достатньо надійним» ускладнюють процес реалізації та можуть стати причиною помилкової інтерпретації функціональних потреб.

Під час цього етапу здійснюється:

- визначення необхідних видів тестування;
- встановлення пріоритетності перевірок;

- формування початкової матриці відповідності вимог і тестів (RTM);
- аналіз вимог до середовища, у якому виконуватиметься тестування.

Результатом є сформована базова структура документації для подальшого контролю відповідності реалізованого функціоналу встановленим вимогам.

Планування тестування. Наступним кроком виступає підготовка плану тестування де визначаються ресурси, часові витрати та підхід до перевірки якості програмного продукту [6]. На цій стадії формується стратегія тестування, обираються інструменти та розподіляються обов’язки між учасниками команди.

Основні роботи включають:

- розроблення тестової стратегії;
- добір програмних засобів тестування;
- оцінювання трудових і технічних ресурсів;
- визначення ролей та зон відповідальності.

Підсумком стає підготовлений план тестування або набір документованих правил, що регламентують подальший процес перевірки програмного забезпечення.

Проєктування тестової документації. На цьому етапі створюється набір тестових сценаріїв та тест-кейсів, які використовуватимуться для перевірки функціональності системи [1]. Визначаються та готуються дані, необхідні для моделювання різних умов експлуатації програмного продукту.

До основних завдань належать:

- підготовка тестових сценаріїв;
- проєктування автоматизованих перевірок;
- перегляд і вдосконалення створених тестів;
- формування тестових наборів даних.

Результатом є повний комплект тестової документації та підготовлені дані

Підготовка тестового середовища. Для забезпечення коректності перевірок створюється окреме середовище, максимально наближене до реальних умов експлуатації системи. Це дозволяє уникнути впливу поточних змін у програмному коді та не перешкоджати роботі кінцевих користувачів.

У межах даного етапу:

- визначається конфігурація апаратних і програмних засобів;
- виконується налаштування інфраструктури;
- завантажуються тестові дані;
- проводиться первинна перевірка працездатності середовища за допомогою smoke-тестування.

Результатом є повністю готове середовище для виконання тестових процедур.

Проведення тестування. Після завершення підготовчих робіт здійснюється безпосередня перевірка функціонування програмного забезпечення відповідно до затвердженої документації. У процесі тестування фіксуються результати виконання тестів та виявлені дефекти.

Основні дії:

- виконання тестових сценаріїв;
- реєстрація знайдених помилок;
- актуалізація RTM;
- повторна перевірка функціоналу після усунення дефектів.

Результатом є сформована база виявлених помилок, оновлена документація та оцінка стану програмного продукту.

Завершення тестового циклу Фінальна стадія передбачає аналіз ефективності проведених робіт та оцінювання досягнутих результатів. Команда узагальнює отримані показники якості та формує рекомендації щодо вдосконалення процесів тестування у майбутньому.

На цьому етапі:

- аналізуються критерії завершення тестування;

- розраховуються показники ефективності;
- готується підсумкова звітність;
- здійснюється класифікація виявлених дефектів за критичністю та пріоритетністю.

Результатом стають статистичні показники якості програмного продукту, аналітичні звіти та рекомендації щодо подальшого вдосконалення процесу розроблення.

1.2. Принципи тестування

Тестування програмного забезпечення базується на сукупності фундаментальних принципів, що визначають ефективний підхід до оцінювання якості програмних продуктів [4]. Ці принципи сформовані на основі багаторічного практичного досвіду та застосовуються незалежно від типу системи або методології розробки.

До ключових принципів належать:

- Тестування підтверджує наявність дефектів, але не доводить їх повну відсутність. Навіть якщо під час перевірок не було знайдено помилок, це не гарантує абсолютної бездоганності програмного продукту.
- Повне тестове покриття є недосяжним. Через величезну кількість можливих комбінацій вхідних даних, середовищ виконання та сценаріїв використання неможливо перевірити всі варіанти роботи системи.
- Раннє виявлення проблем знижує витрати на їх усунення. Помилки, знайдені на етапі аналізу вимог або проєктування, виправляються значно швидше та дешевше, ніж дефекти, виявлені після завершення розробки.
- Дефекти мають тенденцію концентруватися в окремих частинах системи. Практика показує, що переважна частина помилок зазвичай локалізується в невеликій кількості найбільш складних модулів.

- Тестові сценарії потребують регулярного оновлення. Повторне використання однакових перевірок поступово знижує їхню ефективність і ускладнює виявлення нових дефектів.
- Методи тестування залежать від специфіки проєкту. Підходи, ефективні для веб-застосунків, можуть бути непридатними для вбудованих систем, мобільних додатків або програм критичного призначення.
- Відсутність помилок не гарантує успіху продукту. Навіть технічно бездоганне програмне забезпечення може не відповідати потребам користувачів або бізнес-вимогам.

Під час оцінювання якості програмного забезпечення можуть перевірятися такі характеристики:

- відповідність функціональним та нефункціональним вимогам;
- коректність обробки вхідних даних;
- продуктивність та швидкодія;
- зручність використання;
- сумісність із різними платформами та середовищами;
- відповідність очікуванням замовника.

Основною метою тестування є виявлення дефектів та зниження ризиків експлуатації програмного забезпечення. Отримані результати дозволяють надати об'єктивну оцінку якості продукту та сформувати рекомендації щодо його вдосконалення [6].

Тестування може виконуватися на різних стадіях життєвого циклу програмного забезпечення. У традиційних моделях розробки основний обсяг перевірок проводиться після завершення реалізації функціоналу. Натомість сучасні гнучкі методології передбачають інтеграцію тестування безпосередньо в процес розробки, що сприяє швидшому виявленню та усуненню дефектів [7].

У сфері забезпечення якості програмного забезпечення прийнято розрізняти кілька професійних напрямів:

QC (Quality Control) — контроль якості готового програмного продукту шляхом виконання перевірок та аналізу результатів;

QA (Quality Assurance) — забезпечення якості через побудову та вдосконалення процесів розроблення;

AQA (Automation Quality Assurance) — автоматизація тестування та підтримка інструментів автоматизованої перевірки.

Таблиця 1.1

Основні відмінності позиції в ролі тестувальника

Quality Assurance	Quality Control	Тестування
Комплекс заходів, який охоплює всі технологічні аспекти на всіх етапах розробки, запуску та впровадження в експлуатацію програмних систем для забезпечення необхідного рівня якості програмного продукту	Процес контролю відповідності системи, що розробляється, вимогам, що виставлені до неї	Процес, що відповідає безпосередньо за створення та проходження тест-кейсів, знаходження та локалізацію дефектів і т.д.
Фокус в більшій мірі на процеси та засоби, ніж на безпосереднє виконання тестування системи	Фокус на виконання тестування шляхом запуску програми з метою визначення дефектів із використанням затверджених процесів та засобів	Фокус на виконання тестування як такого
Процесно-орієнтований підхід	Продуктно-орієнтований підхід	Продуктно-орієнтований підхід
Превентивні міри	Коректуючий процес	Превентивний процес
Підмножина процесів Software Test Life Cycle – циклу тестування ПЗ	Підмножна процесів QA	Підмножина процесів QC

Ієрархічно тестування є складовою контролю якості, а контроль якості, своєю чергою, входить до ширшої концепції забезпечення якості. Фахівці з автоматизованого тестування поєднують функції QA та розробляють автоматизовані тести, які інтегруються в процеси безперервної інтеграції та доставки програмного забезпечення (CI/CD).

Важливе місце в сучасній розробці займають поняття SDLC (Software Development Life Cycle) та STLC (Software Testing Life Cycle), які забезпечують системний підхід до створення, перевірки та супроводу програмних продуктів протягом усього періоду їх життєвого циклу [11].

1.3. Аналіз існуючих методології розробки програмного забезпечення

Життєвий цикл програмного забезпечення містить наступні етапи [6–9]:

– Планування;
– Дизайн;
– Розробка;
– Тестування;
– Випуск продукту;
– Підтримка.

Найпоширеніші SDLC моделі:

- Ітеративна модель (iterative model);
- Спіральна модель (spiral model);
- V–модель (v–model);
- Каскадна модель (waterfall model);
- Гнучка модель (agile model).

Методології розроблення програмного забезпечення – це комплекс принципів, підходів, моделей та інструментів, які використовуються протягом усього життєвого циклу створення програмного продукту. Її завданням є впорядкування процесів проєктування, програмування, тестування та супроводу програмних систем з метою підвищення ефективності роботи команди та забезпечення належної якості кінцевого результату.

Будь-яка методологія базується на трьох ключових складових:

- сукупності принципів, які визначають загальну філософію процесу розроблення;

- наборі взаємопов’язаних моделей, методів і практик, що реалізують обраний підхід;
- системі термінів і понять, необхідних для формалізації процесів та забезпечення єдиного розуміння між учасниками проєкту.

На етапі створення програмного коду поняття методології часто ототожнюють із парадигмою програмування, яка визначає спосіб організації програмної логіки та структуру програмного продукту [1–9].

У сучасній індустрії програмного забезпечення застосовується багато підходів до організації процесу розроблення. Найпоширенішими серед них є каскадна модель (Waterfall), прототипування (Prototyping), ітеративно-інкрементна модель (Iterative and Incremental Development), спіральна модель (Spiral Model), швидка розробка застосунків (RAD), екстремальне програмування (XP), а також різноманітні гнучкі методології сімейства Agile [1–9].

Одним із найвідоміших представників Agile-підходів є методологія Extreme Programming (XP). Значний внесок у її популяризацію зробив її автор — Кент Бек, який не лише сформував концепцію XP, а й сприяв широкому впровадженню її практик у програмній індустрії.

В основі XP лежать чотири базові цінності:

- ефективна комунікація між учасниками проєкту;
- постійний зворотний зв’язок;
- прагнення до максимальної простоти рішень;
- готовність до змін та сміливість у прийнятті технічних рішень.

На основі цих принципів сформовано комплекс практик, що регламентують організацію процесу розроблення. Частина таких практик існувала й раніше, проте XP об’єднала їх у єдину систему, де кожен елемент підсилює ефективність інших [1–5].

Особливістю методології є еволюційний підхід до проєктування систем. Розробка виконується невеликими ітераціями, під час яких

реалізується лише поточна функціональність без надмірного планування майбутніх можливостей. Такий підхід забезпечує високу гнучкість, швидку адаптацію до змін вимог та можливість постійного вдосконалення програмного коду шляхом рефакторингу.

Розроблення програмного забезпечення з відкритим вихідним кодом

Open Source-проєкти мають власну специфіку організації процесу розроблення. Управління такими проєктами здійснює один або декілька координаторів, які відповідають за підтримку репозиторію та прийняття рішень щодо інтеграції змін у програмний код.

Учасники спільноти можуть пропонувати власні покращення, виправлення помилок або нові функції у вигляді патчів чи змін до репозиторію. Після перевірки та аналізу запропоновані модифікації можуть бути включені до основної гілки проєкту.

Перевагою відкритої моделі розроблення є можливість залучення великої кількості фахівців до тестування та вдосконалення програмного забезпечення. Завдяки цьому пошук помилок відбувається значно швидше, а процес розвитку системи не обмежується ресурсами окремої організації.

Крім того, модель Open Source дозволяє ефективно розподіляти завдання між учасниками проєкту відповідно до їхнього рівня підготовки. Частина розробників займається архітектурними рішеннями та реалізацією функціональності, тоді як інші зосереджуються на тестуванні, аналізі та усуненні дефектів.

Методологія Scrum

Серед сучасних гнучких підходів до управління проєктами особливе місце займає Scrum. Дана методологія орієнтована на організацію роботи команди в умовах постійних змін та високої невизначеності вимог.

Основою Scrum є поділ процесу розроблення на короткі цикли роботи – спринти. Кожен спринт має фіксовану тривалість і завершується створенням працездатного інкременту програмного продукту.

Перед початком спринту команда визначає перелік функціональних можливостей, які необхідно реалізувати. Після затвердження завдань вимоги не повинні змінюватися до завершення поточного циклу роботи.

Для забезпечення прозорості процесів проводяться щоденні короткі зустрічі (Daily Scrum), під час яких учасники обговорюють виконану роботу, поточні труднощі та плани на найближчий період. Такий механізм дозволяє своєчасно виявляти проблеми та підтримувати високий рівень взаємодії між членами команди.

Головна увага в Scrum приділяється безперервному плануванню, контролю виконання завдань та адаптації до змін. Завдяки цьому методологія добре поєднується з іншими Agile-практиками, зокрема принципами екстремального програмування, що дозволяє підвищити якість програмного продукту та скоротити терміни його створення.

РОЗДІЛ 2
СИСТЕМНИЙ АНАЛІЗ ФАКТОРІВ ВПЛИВУ НА ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Поняття якості є одним із ключових критеріїв оцінювання будь-якого продукту або процесу його створення. Під якістю зазвичай розуміють ступінь відповідності певного об’єкта встановленим вимогам, нормативам, стандартам або очікуванням користувачів. У сфері програмної інженерії якість програмного забезпечення характеризує рівень відповідності програмного продукту визначеним функціональним, технічним та експлуатаційним вимогам.

На відміну від матеріальних виробів, програмне забезпечення часто розробляється відповідно до потреб конкретного замовника або певної групи користувачів. Саме тому критерії його оцінювання можуть суттєво відрізнятися залежно від сфери застосування, особливостей бізнес-процесів та очікувань кінцевих користувачів. У зв’язку з цим поняття якості програмного забезпечення охоплює не лише технічні характеристики продукту, а й рівень задоволення потреб осіб, які використовуватимуть систему в реальних умовах експлуатації.

2.1. Вихідні поняття та характеристики якості програмного забезпечення

Характеристики якості програмного забезпечення.

Міжнародний стандарт ISO/IEC 25002:2011 визначає підходи до оцінювання внутрішніх і зовнішніх характеристик якості програмного забезпечення.

Зовнішні характеристики описують властивості програмного продукту, які можуть бути оцінені під час його функціонування. Для їх вимірювання використовуються спеціальні показники та метрики, що

дозволяють кількісно визначити рівень відповідності програмного забезпечення встановленим вимогам. До таких показників можуть належати швидкодія системи, час реакції окремих модулів, пропускна здатність та інші параметри, які оцінюються під час тестування або експлуатації програмного продукту.

Внутрішні характеристики застосовуються на ранніх етапах життєвого циклу програмного забезпечення та використовуються для контролю якості проміжних результатів розроблення. Вони відображають особливості архітектури, структури програмного коду, організації модулів та інших внутрішніх компонентів системи. Для їх оцінювання використовуються внутрішні метрики, що дають змогу визначити рівень відповідності програмного продукту вимогам ще до початку його безпосереднього тестування.

Зовнішні та внутрішні характеристики відображають погляд розробників і замовників на якість програмного продукту. Водночас для кінцевого користувача особливе значення має експлуатаційна якість, яка характеризує загальну ефективність використання програмного забезпечення в реальних умовах роботи. Такий підхід дозволяє оцінювати не окремі властивості системи, а сукупний результат її функціонування.

Стандарт ISO/IEC 25010:2011 визначає модель якості продукту містить 8 характеристик якості: [20].

1. Функціональна придатність (Functional Suitability)

Характеризує здатність програмного забезпечення коректно реалізовувати функції, необхідні користувачам для виконання поставлених завдань.

2. Ефективність продуктивності (Performance Efficiency)

Здатність ПЗ виконувати функції з мінімальними витратами ресурсів (часу, процесора, пам’яті, мережі) за визначених умов. Визначає, наскільки швидко та економно працює система.

3. Сумісність (Compatibility)

Здатність ПЗ працювати разом з іншими системами та продуктами в одному середовищі без конфліктів. Визначає, наскільки добре ПЗ «уживається» та обмінюється даними з іншими програмами.

4. Зручність використання (Usability)

Здатність ПЗ бути зрозумілим, легким у вивченні та комфортним у використанні для цільових користувачів. Визначає, наскільки приємно та ефективно людина може працювати з програмою.

5. Надійність (Reliability)

Здатність ПЗ виконувати задані функції стабільно протягом певного часу в заданих умовах. Оцінює надійність програмного забезпечення – частоту відмов і здатність до відновлення після відмов.

6. Безпека (Security)

Здатність ПЗ захищати дані та інформацію від несанкціонованого доступу, використання, зміни чи знищення. Визначає, наскільки добре система оберігає конфіденційність, цілісність і доступність даних.

7. Підтримуваність (Maintainability)

Здатність ПЗ бути ефективно модифікованим, вдосконаленим та виправленим після впровадження. Визначає, наскільки легко та дешево розробникам підтримувати, виправляти баги і розвивати систему.

8. Переносимість (Portability)

Здатність ПЗ бути перенесеним і адаптованим до роботи в іншому середовищі (інша операційна система, апаратна платформа, браузер тощо). Визначає, наскільки легко ПЗ можна перемістити на нові технології або платформи.

Стандарт ISO/IEC 25010:2023 містить 9 характеристик якості: [21].

1. Функціональна придатність (Functional Suitability).

Здатність ПЗ в певних умовах вирішувати задачі, потрібні користувачам. Визначає, що саме робить ПЗ, які задачі воно вирішує.

2. Ефективність продуктивності (Performance Efficiency).

Здатність ПЗ виконувати функції швидко та з оптимальним використанням ресурсів (процесор, пам’ять, мережа, енергія). Визначає, наскільки ефективно та економно працює система.

3. Сумісність (Compatibility).

Здатність ПЗ працювати разом з іншими системами, обмінюватися даними та співіснувати без негативного впливу. Визначає, наскільки добре ПЗ інтегрується та взаємодіє з іншими продуктами.

4. Здатність до взаємодії (Interaction Capability).

4. Здатність ПЗ забезпечувати ефективну, інтуїтивну, приємну та інклюзивну взаємодію користувачів з системою. Визначає, наскільки зручно, доступно та комфортно працювати з ПЗ різним категоріям користувачів.

5. Надійність (Reliability)

Здатність ПЗ стабільно та безпомилково виконувати функції протягом заданого часу в визначених умовах. Визначає, наскільки системі можна довіряти, наскільки рідко виникають збої та як швидко вона відновлюється.

6. Безпека (Security).

Здатність ПЗ захищати дані та інформацію від несанкціонованого доступу, модифікації, витоку чи знищення. Визначає рівень захисту системи від кіберзагроз та несанкціонованих дій.

7. Підтримуваність (Maintainability)

Здатність ПЗ бути ефективно модифікованим, вдосконаленим, виправленим та протестованим. Визначає, наскільки легко та економно розробники можуть підтримувати й розвивати систему після впровадження.

8. Гнучкість (Flexibility).

Здатність ПЗ адаптуватися, переноситися та масштабуватися в різних середовищах і платформах. Визначає, наскільки легко ПЗ працює на різних пристроях, операційних системах та під різним навантаженням.

9. Безпечність (Safety).

Здатність ПЗ функціонувати без створення неприйнятних ризиків для людей, майна чи навколишнього середовища. Визначає рівень безпечності системи, особливо в критичних застосуваннях (медицина, транспорт, оборонна сфера тощо).

Моделі якості

Стандарт ISO/IEC 25010:2011 встановлює модель якості програмного забезпечення з 8 характеристиками, які стали основним еталоном протягом понад десяти років: функціональна придатність, ефективність продуктивності, сумісність, зручність використання, надійність, безпека, підтримуваність та переносимість. У новій редакції ISO/IEC 25010:2023 модель розширено до 9 характеристик, зробивши її сучаснішою та адаптованою до реалій сучасної розробки. Головні відмінності полягають у перейменуванні двох характеристик: «Зручність використання» (Usability) стала Здатністю до взаємодії (Interaction Capability) з суттєвим розширенням акценту на інклюзивність, самоописовість, залученість користувача та допомогу; «Переносимість» (Portability) трансформувалася в Гнучкість (Flexibility) з доданням масштабованості. Найважливішою зміною стало введення нової самостійної характеристики Безпечність (Safety), яка визначає відсутність неприйнятних ризиків для людей, майна та навколишнього середовища. Крім того, в оновленій моделі з’явилися нові підхарактеристики (наприклад, Resistance у Security, Faultlessness у Reliability), а загальний підхід став більш орієнтованим на інклюзивність, стійкість до сучасних кіберзагроз, критичні системи та великомасштабні рішення. Таким чином, версія 2023 року значно краще відповідає поточним викликам розробки ПЗ, особливо в державному, оборонному та критичному секторах [20, 21].

Суть завдання бакалаврської роботи полягає у встановленні рівнів пріоритетноті впливу перерахованих лінгвістичних характеристик (названих у роботі факторами) на якість проектування програмного забезпечення. Розв’язання поставленого завдання здійснюється на інформаційному рівні

використовуючи моделі оцінювання, що базуються на аналізі відносної важливості факторів. Основою такого підходу є попарне порівняння досліджуваних характеристик, представлених у вигляді лінгвістичних змінних, з метою встановлення їхнього впливу на кінцеву якість програмного продукту.

Ключовим є визначення пріоритетності впливу кожного фактора на процес проєктування. Даний підхід характеризується універсальністю, оскільки дозволяє працювати як із параметрами, що мають кількісне представлення, і з якісними характеристиками, які складно формалізувати та описуються природною мовою. Завдяки цьому забезпечується можливість комплексного врахування різнорідних чинників у процесі оцінювання якості програмного забезпечення. [25].

2.2. Семантична мережа факторів якості програмного забезпечення

На основі наведеного вище якість програмного забезпечення Q відповідно до стандарту [21] залежить від множини восьми основних характеристик (Безпеку (Security) та Безпечність (Safety) умовно об’єднаємо), формальний запис якої матиме вигляд:

$$X = \{ x_1; x_2; ...; x_8 \} \tag{2.1}$$

Для подальшого дослідження останню характеристику «переносимість» замінимо важливою на наш погляд характеристикою «рівень тестування». Остаточно залежність якості програмного забезпечення від множини характеристик запишемо у вигляді функції

$$Q = F(x_1; x_2; ...; x_8), \tag{2.2}$$

де: x_1 – функціональна придатність, x_2 – ефективність, x_3 – здатність до взаємодії, x_4 – надійність, x_5 – сумісність, x_6 – захищеність, x_7 – підтримуваність, x_8 – рівень тестування. Вказані характеристики назвемо факторами впливу на якість програмного забезпечення.

Сукупність математичного позначення, лінгвістичного трактування та мнемонічних назв вказаних факторів подамо у такому вигляді:

$$X = \begin{Bmatrix} x_1 - \textit{функціональна придатність}(ФП); \\ x_2 - \textit{ефективність}(ЕФ); \\ x_3 - \textit{зручність використання}(ЗВ); \\ x_4 - \textit{надійність}(НД); \\ x_5 - \textit{сумісність}(СМ); \\ x_6 - \textit{захищеність}(ЗХ); \\ x_7 - \textit{супроводжуваність}(СП); \\ x_8 - \textit{рівень тестування}(РТ) \end{Bmatrix} \quad (2.3)$$

На підставі отриманих вихідних даних запроектуємо графічну модель зв’язків між факторами у вигляді семантичної мережі (рис. 2.1). У вершинах мережі задаємо лінгвістичні трактування факторів, стрілки вказують наявність і напрям залежності між ними. При потребі можуть використовуватися умовні позначення на стрілках, які можна перевести у типи впливів чи залежностей з відповідною ваговою метрикою.

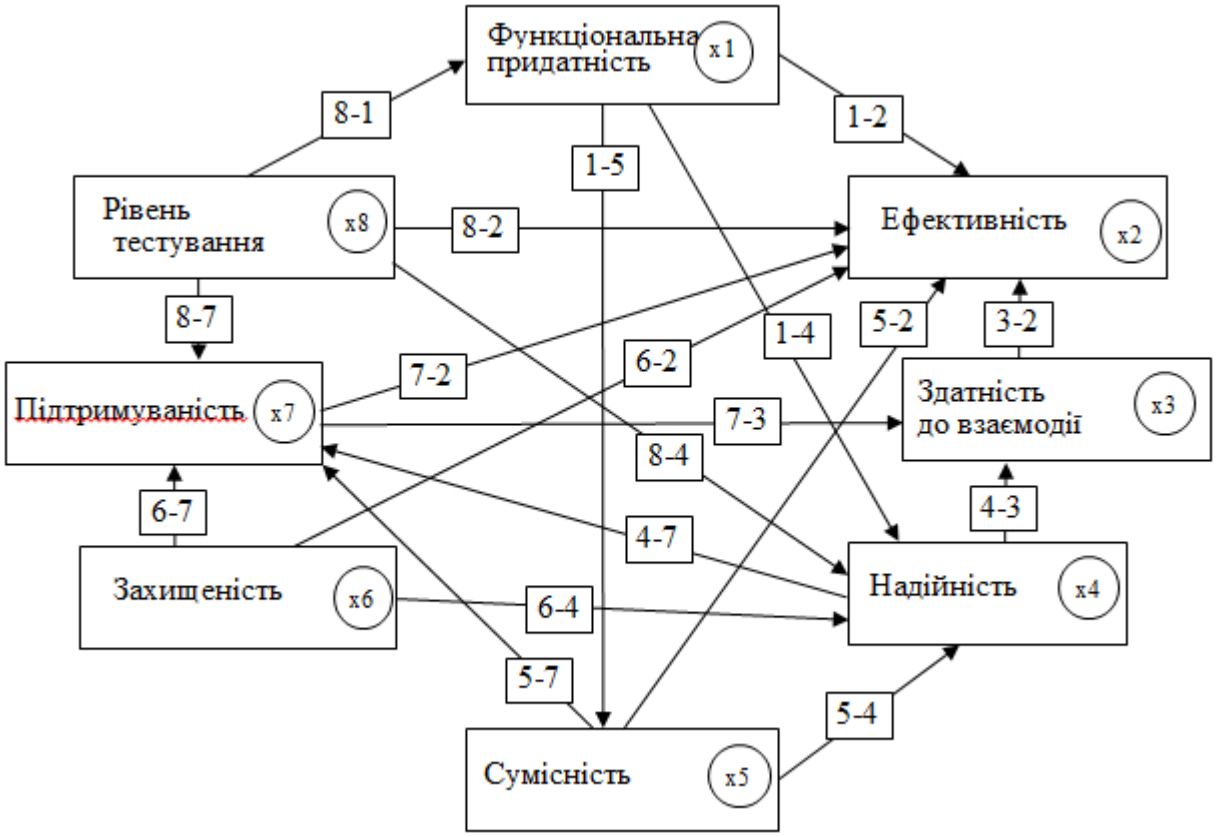

Рис. 2.1. Мережа зв’язків між факторами якості програмного забезпечення

2.3. Модель рівнів пріоритетності дії факторів

Встановлення рівнів пріоритетності факторів здійснюється на підставі методу математичного моделювання ієрархій [25]. Відповідно до даного підходу формується матриця досяжності, яка є математичним представленням взаємозалежностей між факторами, відображеними в семантичній мережі. Формування матриці здійснюється за правилом:

$$b_{ij} = \begin{cases} 1, \textit{якщо з вершини } i \textit{ можна попасти у вершину } j; \\ 0 \textit{ в іншому випадку}. \end{cases} \quad (2.1)$$

Таблиця 2.1

Матриця досяжності

	X1 ФП	X2 ЕФ	X3 ЗВ	X4 НД	X5 СМ	X6 ЗХ	X7 СП	X8 РТ
X1, ФП	1	1	0	1	1	0	0	0
X2, ЕФ	0	1	0	0	0	0	0	0
X3, ЗВ	0	1	1	0	1	0	0	0
X4, НД	0	0	1	1	0	0	1	0
X5, СМ	0	1	0	1	1	0	1	0
X6, ЗХ	0	1	0	1	0	1	1	0
X7, ПД	0	1	1	0	0	0	1	0
X8, РТ	1	1	0	1	0	0	1	1

Практично вершина x_j ($j = 1, 2, ..., 8$) семантичної мережі рис. 2.1 вважається досяжною відносно вершини x_i ($i = 1, 2, ..., 8$), якщо існує хоча б один маршрут переходу між ними, незалежно від кількості проміжних вершин. Таким чином враховуються як прямі, так і непрямі зв'язки між факторами. Такі досяжності назвемо прямими (безпосередніми) та опосередкованими впливами.

У результаті аналізу для кожної вершини формується множина досяжних вершин $D(w_i)$. Аналогічно визначається множина вершин-попередниць $P(w_i)$, до якої входять усі вершини, з яких можливо досягти розглядувану вершину.

Перетин цих множин визначається співвідношенням:

$$Z(w_i) = D(w_i) \cap P(w_i), \tag{2.2}$$

Фактори, що належать до вершин, які не досягаються з інших вершин поточного рівня X, формують відповідний рівень пріоритетності. При цьому обов'язково повинна виконуватися умова:

$$P(w_i) = Z(w_i). \tag{2.3}$$

Для визначення рівнів пріоритетності факторів використовуємо матрицю досяжності і залежності (2.2) і (2.3), на основі чого будуємо першу ітераційну таблицю розрахунку рівнів важливості факторів.

У другому стовпці таблиці розміщуються множини досяжних вершин $D(w_i)$, які визначаються за одиничними елементами відповідних рядків матриці досяжності. У третьому стовпці відображаються множини вершин-попередниць $P(w_i)$, сформовані за одиничними елементами стовпців цієї матриці. Збіг елементів зазначених множин дозволяє встановити фактори, що належать до одного рівня пріоритетності.

Таблиця 2.2

i	$D(w_i)$	$P(w_i)$	$D(w_i) \cap P(w_i)$
1	1,2,4,5	1,8	1
2	2	1,2,3,5,6,7,	2
3	2,3	3,4,7	3
4	3,4,7	1,4,5,6,8	4
5	2,4,5,7	1,5	5
6	2,4,6,7	6	6
7	2,3,7	4,5,6,7,8	7
8	1,2,4,7,8	8	8

З аналізу даних табл. 2.2 видно, що умова збігу виконується для факторів 6 – «захищеність» та 8 – «рівень тестування». Відповідно до використаного методу саме ці фактори утворюють найвищий рівень ієрархії та мають найбільший вплив на процес проєктування програмного забезпечення.

Згідно з методикою ієрархічного моделювання [26, 27] із подальших розрахунків вилучаються рядки, що відповідають знайденим факторам, а також усуваються їхні номери з множин досяжності та попередництва. Після цього формується наступна ітераційна таблиця.

Таблиця 2.3

i	$D(w_i)$	$P(w_i)$	$D(w_i) \cap P(w_i)$
1	1,2,4,5	1	1
2	2	1,2,3,5,7	2
3	2,3	3,4,7	3
4	3,4,7	1,4,5	4
5	2,4,5,7	1,5	5
7	2,3,7	4,5,7	7

З табл. 2.3 отримаємо фактор наступного рівня впливу: 1 – функціональна придатність. Після його вилучення процедура повторюється аналогічним чином.

Результатом наступного кроку є побудова нової таблиці.

Таблиця 2.4

i	$D(w_i)$	$P(w_i)$	$D(w_i) \cap P(w_i)$
2	2	2,3,5,7	2
3	2,3	3,4,7	3
4	3,4,7	4,5	4
5	2,4,5,7	5	5
7	2,3,7	4,5,7	7

Таблиця 2.4 виокремлює фактор 5 – сумісність програмного забезпечення.

Після дій, описаних вище, отримаємо табл. 2.5.

Таблиця 2.5

i	$D(w_i)$	$P(w_i)$	$D(w_i) \cap P(w_i)$
2	2	2,3,7	2
3	2,3	3,4,7	3
4	3,4,7	4	4
7	2,3,7	4,7	7

Таблиця 2.5 закріпляє наступний рівень за фактором 4 – надійність ПЗ.

Повторення попередніх кроків приведе до наступного ранжування факторів якості програмного забезпечення: 7 – підтримуваність; 3 – здатність до взаємодії; 2 – ефективність.

У підсумку отримується багаторівнева ієрархічна структура, яка відображає ступінь впливу факторів на якість програмного забезпечення.

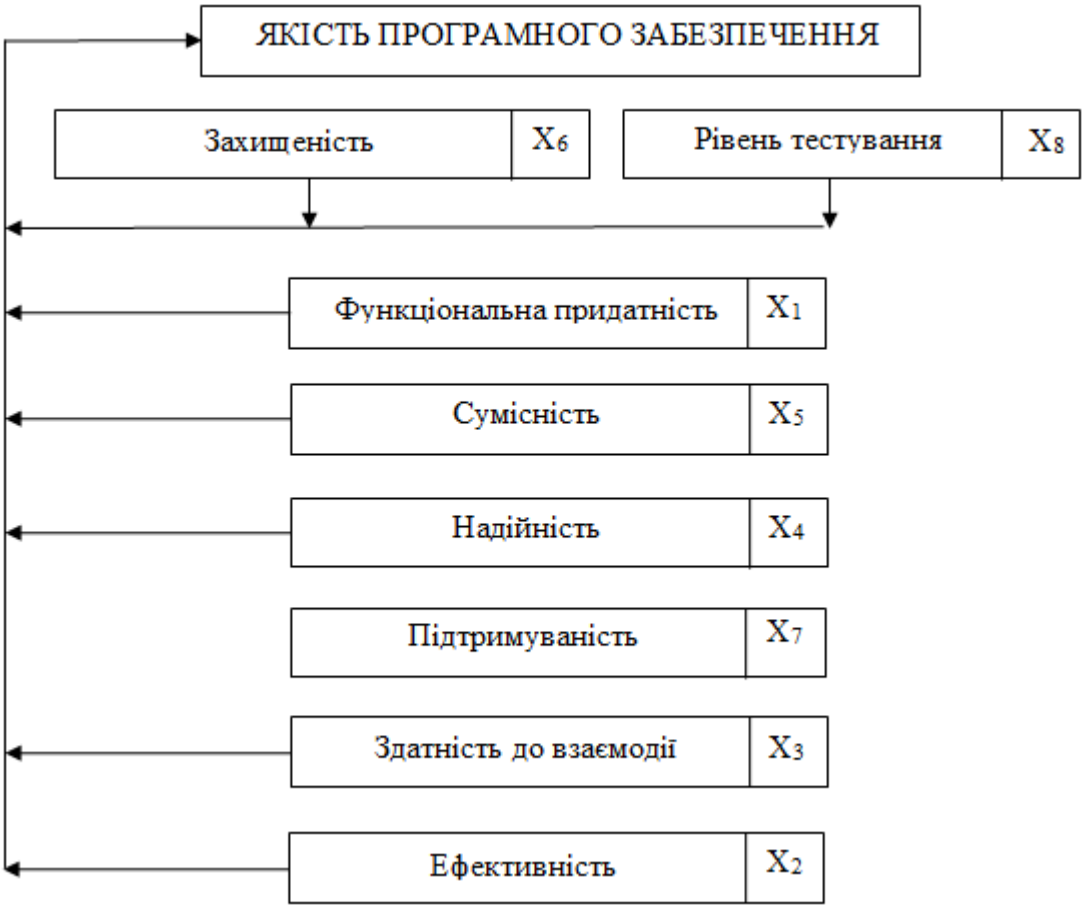

Рис. 2.2 Багаторівнева модель пріоритетності факторів, що впливають на якість програмного забезпечення

Побудована модель відображає узагальнені експертні оцінки щодо факторів у процесі створення програмного продукту. Як видно з рис. 2.2., найбільшу вагу мають фактори «захищеність» та «рівень тестування», в залежності від яких вибудовується ієрархічна «піраміда» факторів – дій. Практика роботи IT–фірм в основному підтверджує цю теоретично обґрунтовану тезу, що свідчить про достовірність результатів проведеного дослідження [26].

2.4. Оптимізація моделі факторів впливу на якість ПЗ

Наступним етапом дослідження, що має практичне значення для бакалаврської роботи, є визначення кількісних вагових коефіцієнтів факторів якості програмного забезпечення. Для цього використовується метод попарного порівняння, який дозволяє встановити рівень переваги одного фактора над іншим та сформувати матрицю попарних порівнянь [26,27].

Застосування цього підходу забезпечує не тільки перевірку узгодженості експертних оцінок щодо пріоритетності факторів, а й дає можливість отримати числову характеристику адекватності взаємозв’язків між ними, відображених у побудованій семантичній моделі. Таким чином здійснюється перехід від якісного аналізу до кількісної оцінки впливу факторів на якість програмного забезпечення.

Допоміжною основою для проведення експертних оцінювань слугують семантична мережа взаємозв’язків факторів (рис. 2.1) та побудована багаторівнева ієрархічна модель їх пріоритетності (рис. 2.2). Якщо декілька факторів належать до одного рівня ієрархії, перевага надається тому фактору, який має більшу кількість зв’язків з іншими елементами системи. У випадку однакової кількості зв’язків остаточне рішення приймається на підставі додаткових експертних суджень.

Результатом попарного порівняння є квадратна обернено-симетрична матриця, подальше опрацювання якої за алгоритмом дозволяє визначити вагові коефіцієнти факторів. Отримані значення використовуються для побудови оптимізованої моделі впливу факторів на процес розроблення ПЗ та якість кінцевого програмного продукту.

Для формування шкали пріоритетності будується квадратна матриця попарних порівнянь, порядок якої відповідає кількості аналізованих факторів. Алгоритм її побудови полягає у порівнянні вагових характеристик

факторів, розташованих у першому стовпці матриці, з вагами факторів верхнього рядка [26,27].

У кожній клітинці матриці фіксується числове значення переваги фактора рядка над фактором стовпця. Очевидно, що діагональні елементи такої матриці завжди дорівнюють одиниці, оскільки кожен фактор порівнюється сам із собою.

Для спрощення процедури оцінювання використовується шкала відносної важливості, запропонована Томасом Сааті [26, 27] (табл. 2.6). Відповідно до цієї шкали кожній парі факторів (наприклад k_1 і k_2), присвоюється числовий коефіцієнт, який відображає ступінь переваги одного фактора над іншим (k_1, k_2)..

Таблиця 2.6

Шкала відносної важливості об’єктів

Оцінка важливо	Критерії порівняння	Пояснення щодо вибору критерію
1	Об’єкти рівноцінні	Відсутність переваги k_1 над k_2
3	Один об’єкт дещо переважає інший	Існує підстава наявності слабкої переваги k_1 над k_2
5	Один об’єкт переважає інший	Існує підстава наявності суттєвої переваги k_1 над k_2
7	Один об’єкт значно переважає інший	Існує підстава присутності явної переваги k_1 над k_2
9	Один об’єкт абсолютно переважає інший	Абсолютна перевага k_1 над k_2 не викликає сумніву
2,4,6,8	Компромісні проміжні значення	Допоміжні порівняльні оцінки

Нижня трикутна частина матриці заповнюється оберненими значеннями елементів верхньої частини. Таким чином, якщо в комірці $A=(a_{ij})$ знаходиться значення 3, то в симетричній комірці $a_{ji}=1/a_{ij}$ буде

записано значення (1, 1/3, 1/5, 1/7, 1/9.). Для оцінювання незначних відмінностей між факторами використовуються проміжні значення 2, 4, 6 та 8 разом з їх оберненими величинами.

Вибір максимальної оцінки, що дорівнює дев'яти, обумовлений результатами досліджень у галузі прийняття рішень. Встановлено, що для надійного розрізнення альтернатив a_{ij} достатньо п'яти основних рівнів переваги: рівнозначна, слабка, суттєва, дуже сильна та абсолютна. З урахуванням проміжних градацій формується дев'ятибальна шкала оцінювання.

Після проведення всіх попарних порівнянь формується остаточна матриця.

Таблиця 2.7

Матриця попарних порівнянь

	X1 ФП	X2 ЕФ	X3 ЗВ	X4 НД	X5 СМ	X6 ЗХ	X7 ПД	X8 РТ
X1, ФП	1	7	6	4	3	1/2	5	1/3
X2, ЕФ	1/7	1	1/3	1/5	1/6	1/8	1/4	1/9
X3, ЗВ	1/6	3	1	1/4	1/5	1/7	1/3	1/8
X4, НД	1/4	5	4	1	1/3	1/5	3	1/6
X5, СМ	1/3	6	5	3	1	1/4	4	1/5
X6, ЗХ	2	8	7	5	4	1	6	1/2
X7, ПД	1/5	4	3	1/3	1/4	1/6	1	1/7
X8, РТ	3	9	8	6	5	2	7	1

Для визначення вектора пріоритетів використовується метод аналізу ієрархій Сааті [26, 27]. Розрахунки виконуються за допомогою програмного засобу інтерфейс якого наведено на рис. 2.3.

На першому етапі визначається головний власний вектор матриці попарних порівнянь $X(x_1, x_2, ..., x_n)$. Одним із найбільш поширених способів його знаходження є обчислення середнього геометричного елементів кожного рядка матриці:

$$x_i = \sqrt[n]{a_{i1} \cdot a_{i2} \cdot \ldots \cdot a_{in}} \quad i = \overline{1,n}. \tag{2.4}$$

У результаті розрахунків отримано головний власний вектор:

$$X = (2{,}127;\ 0{,}220;\ 0{,}332;\ 0{,}799;\ 1{,}251;\ 3{,}008;\ 0{,}512;\ 4{,}165).$$

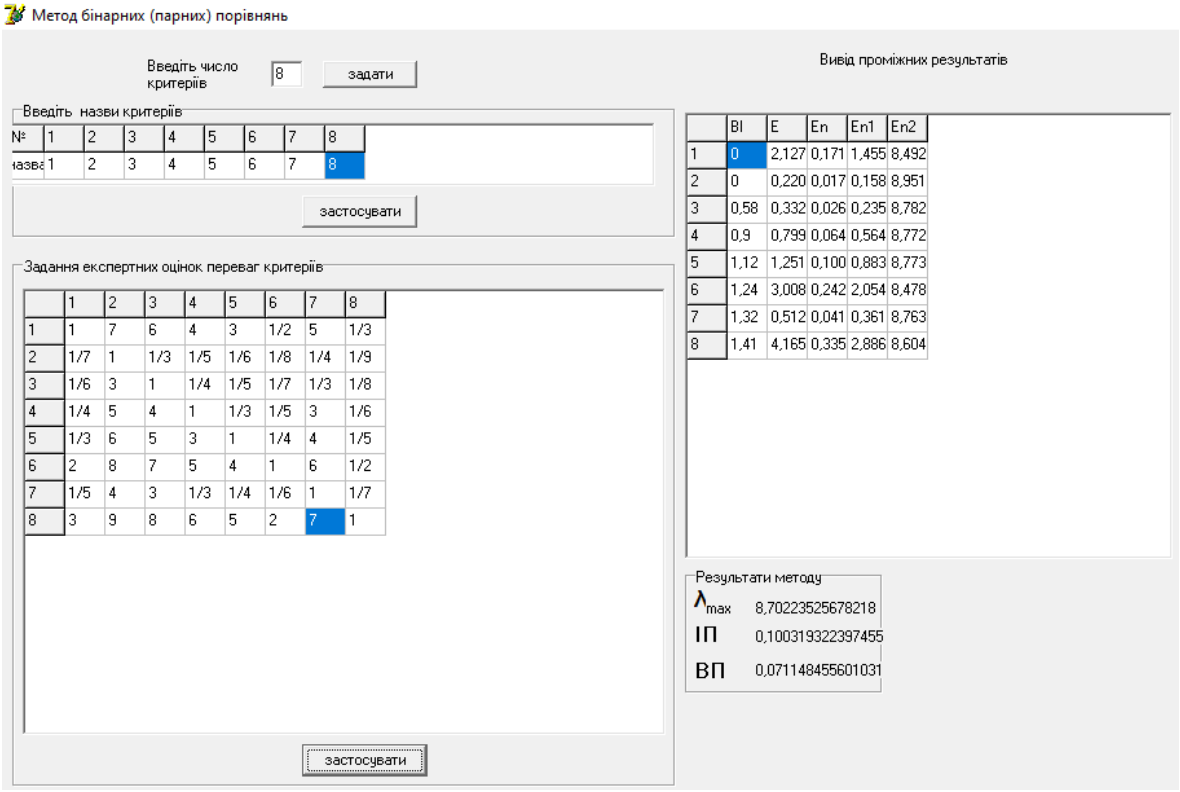

Рис. 2.3. Результат опрацювання матриці

Наступним кроком є нормалізація отриманого вектора. Для цього кожна його компонента ділиться на суму всіх компонент:

$$x_{i\ норм} = \frac{\sqrt[n]{a_{i1} \cdot a_{i2} \cdot \ldots \cdot a_{in}} \quad i = \overline{1,n}}{\sum_{i=1}^{n} \sqrt[n]{a_{i1} \cdot a_{i2} \cdot \ldots \cdot a_{in}}}. \tag{2.5}$$

Після виконання нормалізації отримуємо (опція E_n):

$$X_{норм} = (0{,}171;\ 0{,}017;\ 0{,}026;\ 0{,}064;\ 0{,}100;\ 0{,}242;\ 0{,}041;\ 0{,}335),$$

Отриманий вектор відображає кількісні пріоритети досліджуваних факторів та є формальним результатом процедури оцінювання.

Для більш наочного подання вагових коефіцієнтів можна застосувати масштабування, наприклад шляхом множення значень на 1000:

$$X_{норм} \times k = (171;\ 17;\ 26;\ 64;\ 100;\ 242;\ 41;\ 335).$$

Далі виконується оцінювання узгодженості експертних суджень. Для цього матриця попарних порівнянь множиться на нормалізований вектор $X_{норм}$ пріоритетів, у результаті чого отримується вектор:

$$X_{норм1} = (1{,}455;\ 0{,}158;\ 0{,}235;\ 0{,}564;\ 0{,}883;\ 2{,}054;\ 0{,}361;\ 2{,}886).$$

Після цього визначаються компоненти допоміжного вектора шляхом ділення відповідних елементів вектора $X_{норм1}$ на елементи нормалізованого вектора $X_{норм}$:

$$X_{норм2} = (8{,}492;\ 8{,}951;\ 8{,}782;\ 8{,}772\ 8{,}773;\ 8{,}478;\ 8{,}783;\ 8{,}604).$$

Максимальне власне значення матриці визначається як середнє арифметичне компонент вектора $\lambda_{\max} = 8{,}70$.

$$IU = (\lambda_{\max} - n)/(n - 1). \qquad (2.6)$$

За результатами аналізу матриці попарних порівнянь, отриманими в процесі її обробки (рис. 2.5), для досліджуваного випадку було визначено значення індексу узгодженості $IU = 0{,}10$.

Для оцінювання достовірності отриманих результатів розраховане значення індексу узгодженості зіставляється з нормативним показником, який називають випадковим індексом (WI). Його величина залежить від розмірності матриці, тобто від кількості факторів або альтернатив, що беруть участь у процедурі попарного порівняння. Під випадковим індексом розуміють середнє значення індексу узгодженості, отримане для випадково сформованих обернено-симетричних матриць, елементи яких генеруються відповідно до дев'ятибальної шкали оцінювання. Експертні оцінки вважаються достатньо узгодженими у тому випадку, якщо розрахований

індекс узгодженості не перевищує 10 % від еталонного значення випадкового індексу для матриці відповідного порядку.

Довідкові значення випадкового індексу для матриць різної розмірності, що відповідає різній кількості порівнюваних об’єктів, наведено нижче.

Таблиця 2.8

Значення випадкового індексу для матриць різного порядку

Кількість об'єктів	3	4	5	6	7	8	9	10	11	12
Еталонне значення індексу	0,58	0,90	1,12	1,24	1,32	1,41	1,45	1,49	1,51	1,54

Порівнявши обчислене значення індексу узгодженості з нормативним табличним значенням для матриці, що містить вісім об’єктів, та перевіривши виконання умови , одержуємо: $IU < 0,1 \times WI$, $0,10 < 0,1 \times 1,41$. Дотримання зазначеної умови свідчить про коректність отриманого результату та підтверджує адекватність побудованої моделі .

Крім того, якість отриманих результатів додатково оцінюється за допомогою відношення узгодженості, яке визначається за формулою: $WU = IU/WI$. Для розглянутого випадку маємо $WI = 1,41$, то відповідно $WU = 0,07$ Результати попарних порівнянь є прийнятними за умови виконання нерівності $WU \le 0,1$. Отримане значення підтверджує достатній рівень збіжності розрахункового процесу та свідчить про належну узгодженість експертних оцінок, використаних під час побудови матриці попарних порівнянь факторів.

У ході проведеного дослідження для всіх розглянутих факторів були визначені нормалізовані вагові коефіцієнти, які характеризують ступінь їхнього впливу на якість програмного забезпечення. Значення ваг

сформовано на основі головного власного вектора матриці попарних порівнянь, що дозволило отримати оптимізовані оцінки пріоритетності факторів та забезпечити їх відповідність структурі взаємозв'язків, відображеній у вихідній графічній моделі.

У разі, якщо значення індексу узгодженості або відношення узгодженості не відповідають встановленим вимогам, необхідно повторного аналізу початкової схеми зв'язків між факторами та коригування експертних оцінок, використаних у матриці попарних порівнянь. Фактично це передбачає розв'язання оберненої задачі з подальшою перевіркою отриманих результатів за критеріями узгодженості. У підсумку розраховані вагові коефіцієнти слугують основою для побудови оптимізованої моделі пріоритетного впливу факторів на якість програмного забезпечення.

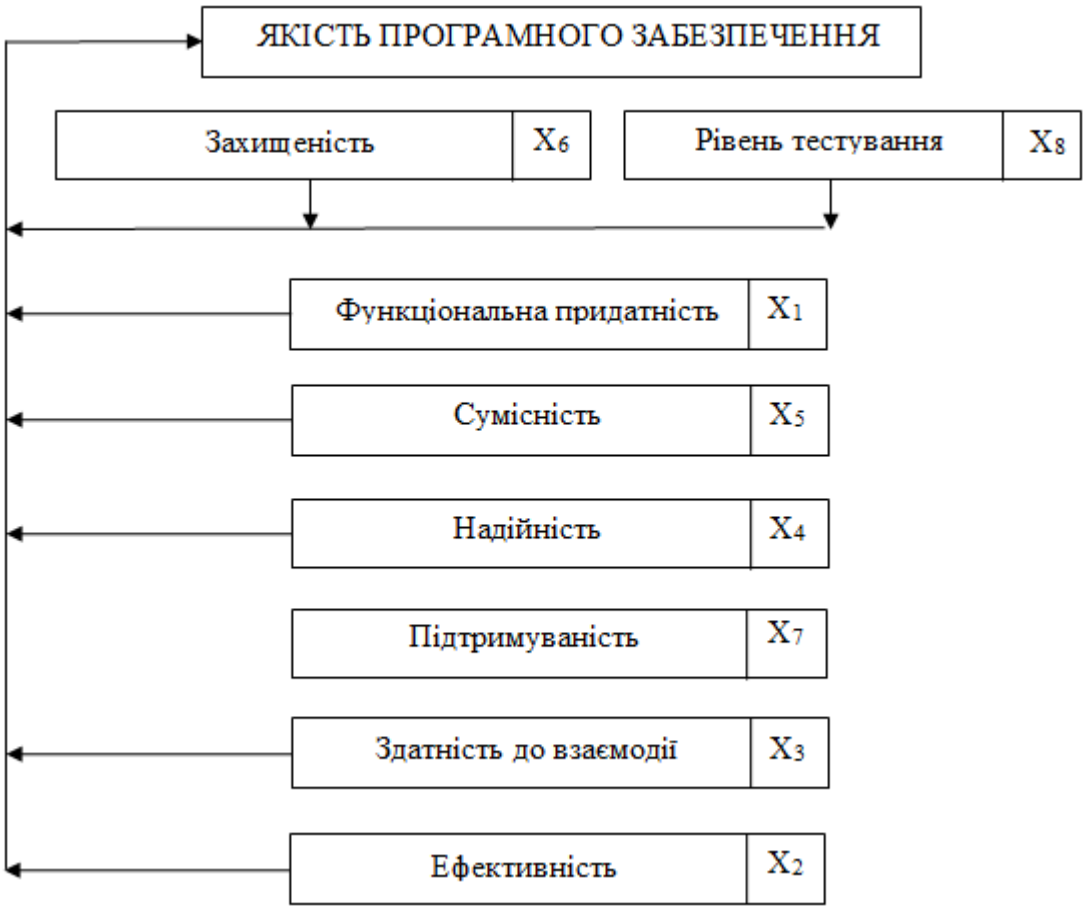

Рис. 2.4. Багаторівнева модель пріоритетного впливу факторів на якість програмного забезпечення

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО КОМПЛЕКСУ ДЛЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Структура програмного комплексу

3.1.1. Діаграма діяльності

Щоб описати структуру програмного застосунку для тестування програмних компонент зобразимо її за допомогою діаграм діяльності UML (рис. 3.1.). Діаграма діяльності ілюструє послідовність виконання основного процесу автоматизованої перевірки відповідності системних вимог операційної системи. Вона відображає логіку роботи програмного забезпечення від моменту запуску тесту до завершення з формуванням результатів.

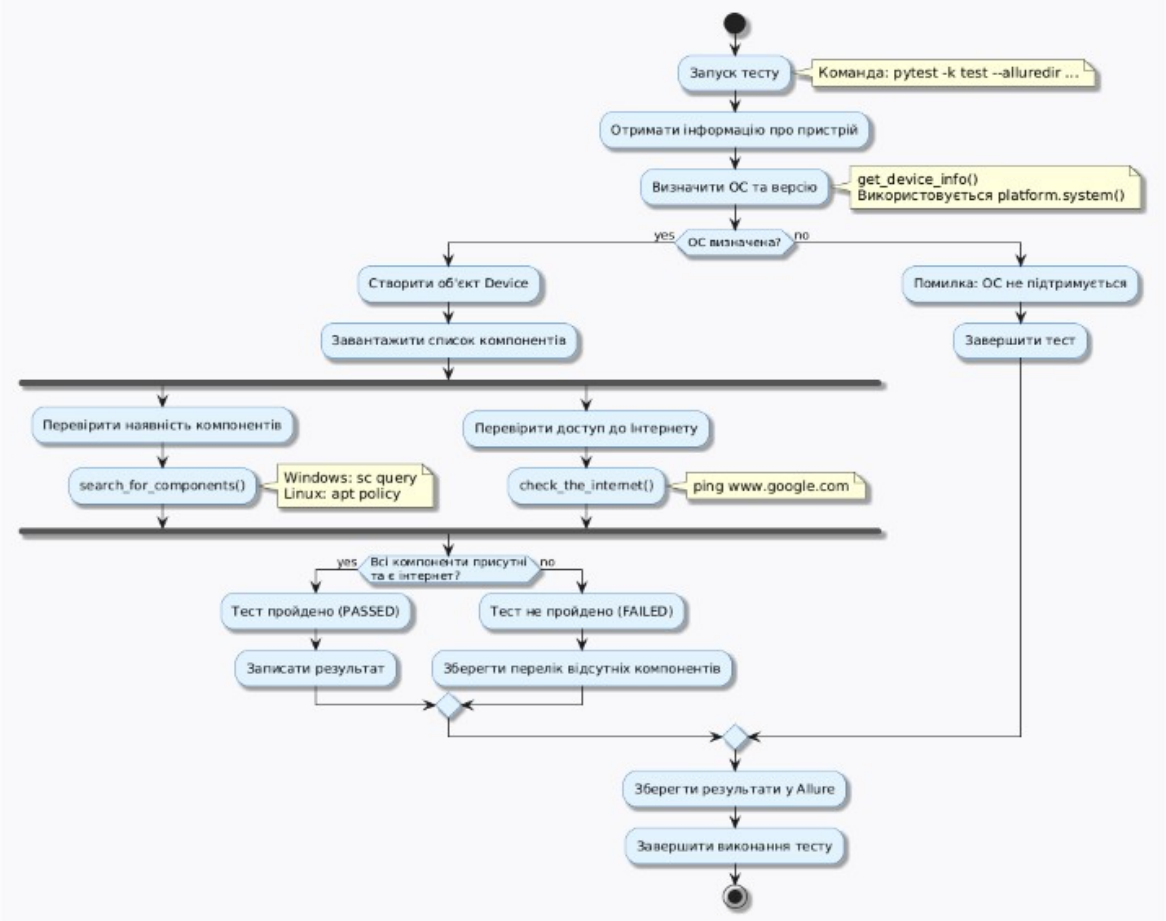

Рис. 3.1 Діаграма діяльності

Опис процесу:

Початок процесу. Процес починається з запуску тестів за допомогою команди pytest. Користувач запускає тести з кореневої директорії проєкту, вказуючи ключі -k test та --alluredir для збереження результатів у відповідну директорію.

Отримання інформації про пристрій. Викликається функція get_device_info() (файл util/environment/env.py). За допомогою модулів platform та sys визначається тип операційної системи (Windows або Linux) та її версія. Назва ОС нормалізується у верхній регістр для подальшого порівняння з константами.

Перевірка підтримки операційної системи. Приймається рішення: якщо операційна система розпізнана і підтримується (Windows або Linux), процес продовжується. У разі виявлення непідтримуваної ОС тест завершується з помилкою.

Створення об’єкта Device. Створюється екземпляр класу Device (файл util/clients/device_client.py), до якого передаються дані про операційну систему та її версію.

Паралельне виконання перевірок (Fork). Процес розгалужується на дві паралельні гілки:

Перевірка наявності необхідних компонентів – викликається функція search_for_components().

Залежно від ОС виконуються системні команди: Windows: sc query;

Linux: apt policy

Відсутні компоненти накопичуються у списку missed_components.

Перевірка доступу до мережі – викликається функція check_the_internet(), яка виконує команду ping www.google.com.

Прийняття рішення про результат тесту. Після завершення обох перевірок проводиться аналіз результатів:

Якщо всі необхідні компоненти присутні та є доступ до Інтернету – тест вважається успішним (PASSED).

В іншому випадку тест вважається невдалим (FAILED), а система зберігає перелік відсутніх компонентів для подальшого аналізу.

Збереження результатів. Результати тестування (успіх/невдача, логи виконання, перелік проблем) записуються у форматі, сумісному з Allure.

Завершення процесу. Виконання тесту завершується. Звіт можна переглянути за допомогою команди allure serve.

Основні елементи діаграми:

Початковий і кінцевий стани – позначають початок і завершення процесу. Дії – прямокутники зі скругленими кутами, що описують конкретні операції. Рішення (ромб) – точки розгалуження логіки (підтримувана ОС, результат перевірки). Паралельне виконання (Fork/Join) – дозволяє показати незалежну перевірку компонентів та доступу до Інтернету. Нотатки – містять важливі технічні деталі (назви функцій, використовувані команди).

3.1.2. Діаграма варіантів використання

Діаграма варіантів використання відображає основні функціональні можливості розробленої системи (рис. 3.2). Головний актор — Користувач (Тестувальник), який ініціює процес перевірки. Використані зв’язки <> показують обов’язкові підпроцеси (визначення ОС, перевірка компонентів, інтернету), а <> — опціональне розширення (перегляд Allure-звіту).

Рис. 3.2 Діаграма варіантів використання

Діаграма ілюструє взаємодію користувача (тестера або системного адміністратора) з системою. Основними варіантами використання є визначення середовища, перевірка компонентів, перевірка мережі та генерація звітів. Діаграма показує, що варіант «Виконати повний цикл тестування» включає в себе інші сценарії.

3.2. Аналіз і вибір методів, алгоритмів та інструментів

У рамках кваліфікаційної роботи розроблено програмне забезпечення для автоматизованої перевірки відповідності мінімальних системних вимог операційної системи (Windows або Linux) шляхом верифікації наявності ключових компонентів та доступу до мережі. Для ефективної реалізації поставлених завдань було проведено аналіз і здійснено вибір методів, алгоритмів та інструментів, що забезпечують надійність, розширюваність, зручність тестування та генерацію звітності.

1. Вибір технологічного стеку. Одним із ключових етапів проектування програмного забезпечення є вибір мови програмування, який суттєво впливає на ефективність реалізації, підтримку та масштабованість рішення. У рамках бакалаврської роботи для розробки системи

автоматизованої перевірки компонентів та середовища операційних систем Windows і Linux обрано мову програмування Python.

Вибір Python обумовлений сукупністю факторів, що роблять його оптимальним інструментом для вирішення поставлених завдань.

Python вирізняється високим рівнем крос-платформенності. Мова забезпечує стабільну роботу на різних операційних системах без суттєвих змін у коді завдяки стандартним бібліотекам platform, os, sys та subprocess. Це особливо важливо для даного проєкту, метою якого є перевірка компонентів як на Windows, так і на Linux [16, 17].

Python володіє однією з найбільш розвинених екосистем для автоматизованого тестування та системної автоматизації. Центральним фреймворком тестування обрано pytest, який підтримує потужний механізм фікстур, параметризацію тестів, маркери та плагінну архітектуру. Для генерації детальних інтерактивних звітів використовується Allure Framework разом з адаптером allure-pytest.

Python характеризується простотою синтаксису, високою читабельністю коду та швидкістю розробки. Це дозволяє зосередитись на логіці бізнес-задач, а не на низькорівневих деталях реалізації, що є важливим у науково-дослідній роботі.

Мова забезпечує високий рівень безпеки та надійності при виконанні системних команд завдяки модулю subprocess, який надає повний контроль над процесами, обробку помилок і таймаути.

Для організації автоматизованого тестування обрано фреймворк pytest 9.0.3. Перевагами даного вибору є:

Для генерації інтерактивних звітів використано Allure 2.41.0 — сучасний інструмент, який дозволяє створювати детальні, візуально привабливі звіти з ієрархією тестів, логами виконання, скріншотами (за потреби) та фільтрами. Allure інтегрується з pytest через відповідний плагін і значно підвищує наочність результатів тестування.

Контроль версій здійснюється за допомогою Git 2.54.0, що є стандартом для академічних та комерційних проєктів.

2. Архітектура програмного забезпечення. Проєкт побудовано за принципами модульності та розділення відповідальностей (Separation of Concerns):

Директорія requirements — містить конфігураційні дані (набори компонентів для Windows та Linux у форматі set). Використання множин забезпечує швидку перевірку наявності елементів (O(1) середній час доступу).

Директорія testcases — містить тестову логіку (test_cycle.py), допоміжні модулі (helpers.py, constants.py) та конфігурацію Pytest (conftest.py).

Директорія util — інкапсулює бізнес-логіку:

clients/device_client.py — клас Device, що представляє сутність перевірюваного пристрою (патерн Data Transfer Object / Entity).

environment/env.py — модуль збору інформації про середовище виконання.

Таке розділення забезпечує високу тестоздатність, підтримуваність та можливість розширення (додавання нових ОС або компонентів).

3. Алгоритми та методи перевірки. Збір інформації про операційну систему реалізовано у функції get_device_info():

Використання кросплатформових модулів platform та sys;

Нормалізація назви ОС (метод .upper());

Витяг версії з урахуванням особливостей кожної платформи (для Windows — platform.version(), для Linux — парсинг sys.version).

Алгоритм перевірки компонентів (search_for_components):

Ітеративна перевірка кожного компонента за допомогою системних команд (sc query для Windows, apt policy для Linux);

Накопичення відсутніх компонентів у списку missed_components;

Повернення результату у зрозумілому для тестового фреймворку форматі (порожній список = успіх).

Перевірка доступу до інтернету (check_the_internet):

Використання команди ping www.google.com через os.system(). Попри простоту, даний метод є достатнім для базової перевірки мережевої доступності.

Використання pytest-fixture у conftest.py дозволяє створювати інстанс Device один раз на сесію тестування, забезпечуючи ефективність та консистентність даних.

Таблиця 3.1

Обґрунтування проектних рішень

Аспект	Обране рішення	Обґрунтування
Мова та фреймворк	Python + pytest	Швидкість розробки, зріла екосистема тестування
Звітність	Allure	Висока наочність, інтерактивність, інтеграція з CI/CD
Архітектура	Модульна з чітким розділенням	Підтримуваність, розширюваність
Перевірка компонентів	Системні команди + парсинг результатів	Пряма взаємодія з ОС без додаткових залежностей
Кросплатформенність	platform + умовні гілки	Мінімальний код, максимальна надійність
Конфігурація	Константи в окремих класах та sets	Читабельність та швидкість

Використання екосистеми Python у складі фреймворку pytest та інструменту Allure дозволило створити сучасне, гнучке та зручне в експлуатації програмне рішення для автоматизованої перевірки системного середовища. Обрані засоби забезпечують високу якість коду, надійність роботи на різних операційних системах, а також відповідність сучасним стандартам у сфері автоматизованого тестування та DevOps-практик.

3.3. Фреймворки для автоматизованого тестування

1. JBehave є одним із поширених інструментів тестування для платформи Java, який реалізує підхід BDD (Behavior-Driven Development – розробка, орієнтована на поведінку). Дана концепція виникла як подальший розвиток методологій TDD (Test-Driven Development) та ATDD (Acceptance Test-Driven Development), поєднуючи їхні переваги з акцентом на поведінку системи з точки зору кінцевого користувача.

Головною метою BDD є покращення взаєморозуміння між учасниками проєкту шляхом використання зрозумілих сценаріїв поведінки системи. Такий підхід спрощує процес створення тестів, робить його більш наочним та дозволяє ефективніше організувати співпрацю між розробниками, тестувальниками, бізнес-аналітиками та замовниками.

Основними компонентами платформи JBehave є:

- JBehave Core;
- JBehave Web.
- Вимоги для встановлення JBehave

Для використання фреймворка рекомендується інтеграція із середовищем розробки Eclipse. Для коректної роботи необхідна наявність JDK версії 1.7 або новішої, а також сучасної версії Eclipse IDE. Крім цього, можуть знадобитися додаткові бібліотеки та JAR-файли.

Фреймворк JBehave має низку переваг, серед яких:

- підтримка принципів поведінкового тестування;

- покращення взаємодії між командами, що працюють над спорідненими проєктами;
- єдиний формат специфікацій для всіх учасників процесу розроблення;
- підвищення прозорості процесів контролю якості;
- спрощення відстеження вимог та результатів тестування;
- зручність формування тестової документації.

Завдяки використанню стандартизованих сценаріїв усі учасники проєкту мають однакове бачення функціональних вимог та очікуваної поведінки програмного продукту.

Недоліки:

Основним обмеженням JBehave є висока залежність від ефективності комунікації між учасниками проєкту. У разі виникнення непорозумінь між розробниками, тестувальниками, керівниками або замовниками можуть з’являтися неоднозначності у вимогах та інтерпретації поведінкових сценаріїв.

Недостатня узгодженість дій між учасниками команди здатна призвести до появи програмних дефектів або реалізації функціональності, що не відповідає очікуванням замовника.

За принципом роботи JBehave має багато спільного з платформою Serenity. Досить поширеним є їх спільне використання, оскільки Serenity забезпечує зручні механізми побудови звітів та додаткові можливості для реалізації BDD-підходу.

2. JUnit належить до найвідоміших відкритих фреймворків для тестування програмного забезпечення мовою Java. Він виконує тестування, що дозволяє контролювати якість окремих компонентів програми та своєчасно виявляти помилки на ранніх етапах розроблення.

Фреймворк широко застосовується для створення автоматизованих тестів, перевірки працездатності окремих методів і класів, а також для реалізації регресійного тестування після внесення змін до програмного коду.

Ефективним є використання JUnit у поєднанні з Selenium WebDriver, що дає можливість автоматизувати тестування вебзастосунків та контролювати коректність роботи інтерфейсу користувача.

До основних можливостей JUnit належать:

- підтримка тестових наборів (Test Suites);
- використання фікстур для підготовки тестового середовища;
- набір спеціалізованих класів і анотацій;
- підтримка механізму запуску тестів (Test Runners).

Фреймворк розроблений для екосистеми Java та сумісний із більшістю популярних середовищ розробки, серед яких:

- Eclipse;
- NetBeans;
- IntelliJ IDEA;
- Maven;
- Ant.

Завдяки цьому створення, запуск та супровід модульних тестів можуть здійснюватися безпосередньо у середовищі розробки.

Сучасна версія JUnit 5 забезпечує підтримку нових механізмів тестування, розширені можливості роботи з винятковими ситуаціями та сумісність із тестами, створеними у попередніх версіях фреймворка.

Для роботи з JUnit можуть використовуватися Java 5 та новіші версії платформи.

Серед недоліків JUnit варто відзначити відсутність вбудованої підтримки тестування залежностей між тестовими сценаріями. Для реалізації подібної функціональності зазвичай застосовується фреймворк TestNG.

Отже, JUnit є одним із найпоширеніших інструментів модульного тестування для Java-застосунків. За функціональними можливостями він має багато спільного з TestNG, проте відрізняється підходами до організації та виконання тестів.

3.4. Розробка програмного комплексу

Як уже зазначалось у даній бакалярській роботі буде використовуватись наступне:

1. Python – 3.14.8
2. Pytest – 9.0.3
3. Git – 2.54.0. windows.10
4. Allure – 2.41.0

Алгоритм процесу міститиме такі кроки:

Дане ПЗ буде запускатись на операційній системі Windows або Linux, перевіряти яка поточна операційна система, та її версія, перевіряти чи мінімальні компоненти для правильної роботи встановлені, чи є можливість виходу в інтернет, в разі необхідності формування звіту по виконаних тестах.

Структура проекту наступна:

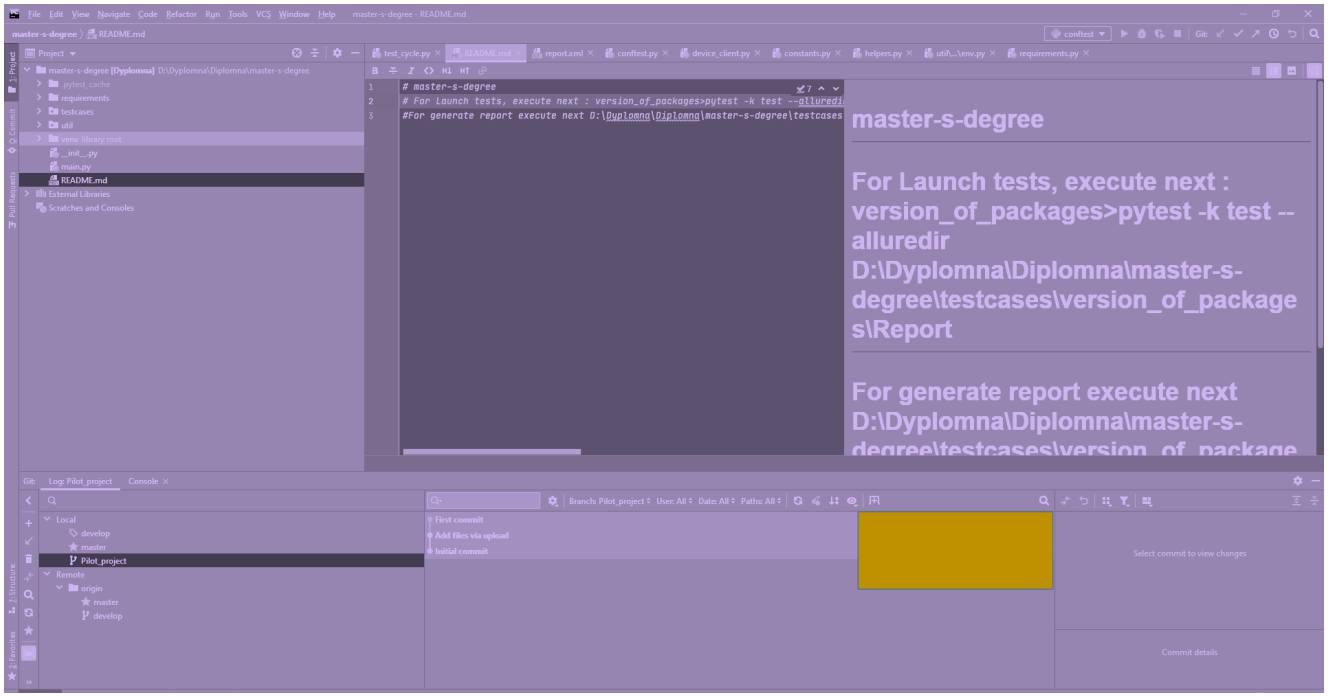

Рис.3.3. Структура розроблюваного проекту

Requiremets – у директиві requirements буде знаходиться файл, в якому буде перелік компонентів як для платформи Windows так і для Linux.

Testcases – у директиві test–cases буде знаходитись модулі для виконання тесту, такі як conftest.py, helpers.py, constans.py, самий файл з тестами test_cycle.py, та директива Report, в якій будуть зберігатись результати виконання в Allure репорті.

Util – директива, в якій буде створюватись сутність пристрою.

Environment – директива, в якій буде братись інформація про пристрій

Також в корені проекту, буде файл README, в якому буде опис програмного забезпечення, та інструкція для запуску

Requirements

Дана директива має в собі файл requirements.py, який в собі має 4 константи:

```
WINDOWS_COMPONENTS_CORRECT
WINDOWS_COMPONENTS_INCORRECT
LINUX_COMPONENTS_CORRECT
LINUX_COMPONENTS_INCORRECT
```

Ці константи будуть в типі Set, та будуть містити список компонент. Для Windows це буде:

```
'Dhcp',
'Dnscache',
'KtmRm',
'SysMain',
'Audiosrv',
'WUAUSERV'
```

Для Linux:

```
'bash'
'netcat'
```

Також інші константи необхідні для відображення невдалого тесту для Windows:

```
'Dhcp',
'Dnscache',
'KtmRm',
'Linux',
'SysMain',
'Audiosrv',
'WUAUSERV'
```

Linux:

```
'bash'
'netcat'
'Windows'
```

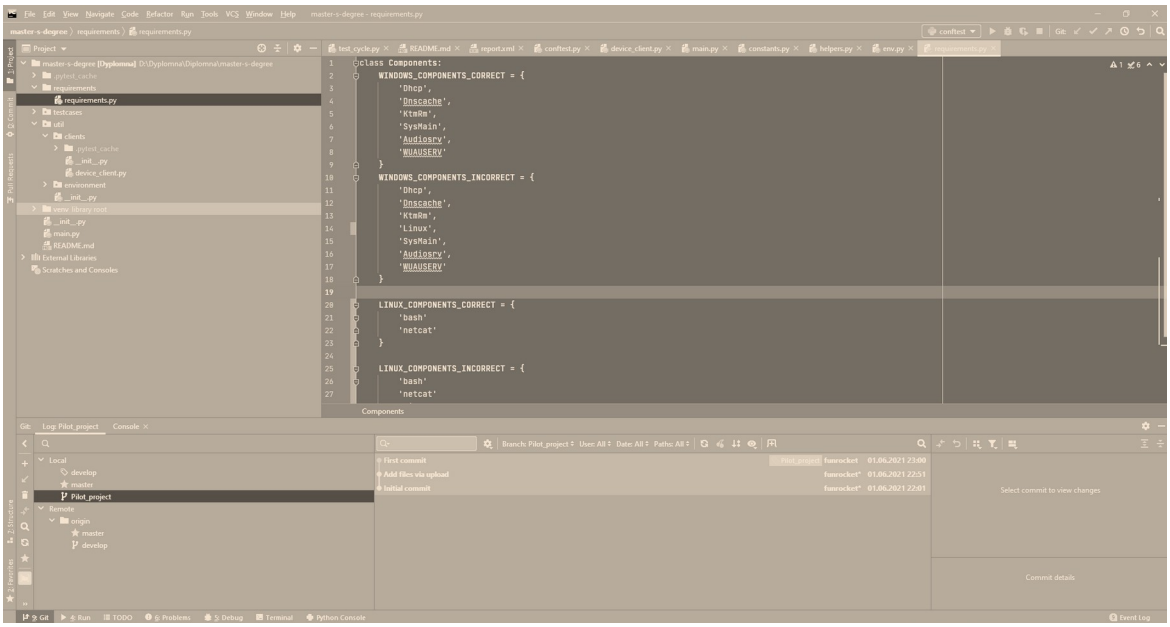

Рис. 3.4. Директива Requirements з файлом переліку компонент

Testcases

Дана директива має в собі наступні файли:

Conftest.py, constants.py, helpers.py а також директиву version_of_packages де знаходить файл test_cycle.py.

У файлі conftest.py задано наступне: імпорт pytest фреймворка, сутності пристрою з файлу util.clients.device_client а також util.environment.env функція get_gevice_info. Основний функціонал

наступний, за допомогою pytest заданий декоратор pytest.fixture. Це створенно для того, щоб задати нашій сутності пристрою певні параметри. Вона буде доступна протягом користувацьної сесії, а також буде автоматично виконуватись.

Послідовність дій наступна: в змінну env_os задається результат виконання get_device_info, потім в інстанс класу Device задаються параметри про пристрій, через розділення словника, де за першим індексом знаходиться назва операційної системи, а в другому її версія. Після виконання створення інстансу пристрою, за допомогою методу yield повертаємо наш створений обєкт. Yield використовується для того, щоб можна було багаторазово використати наш законфігурований пристрій.

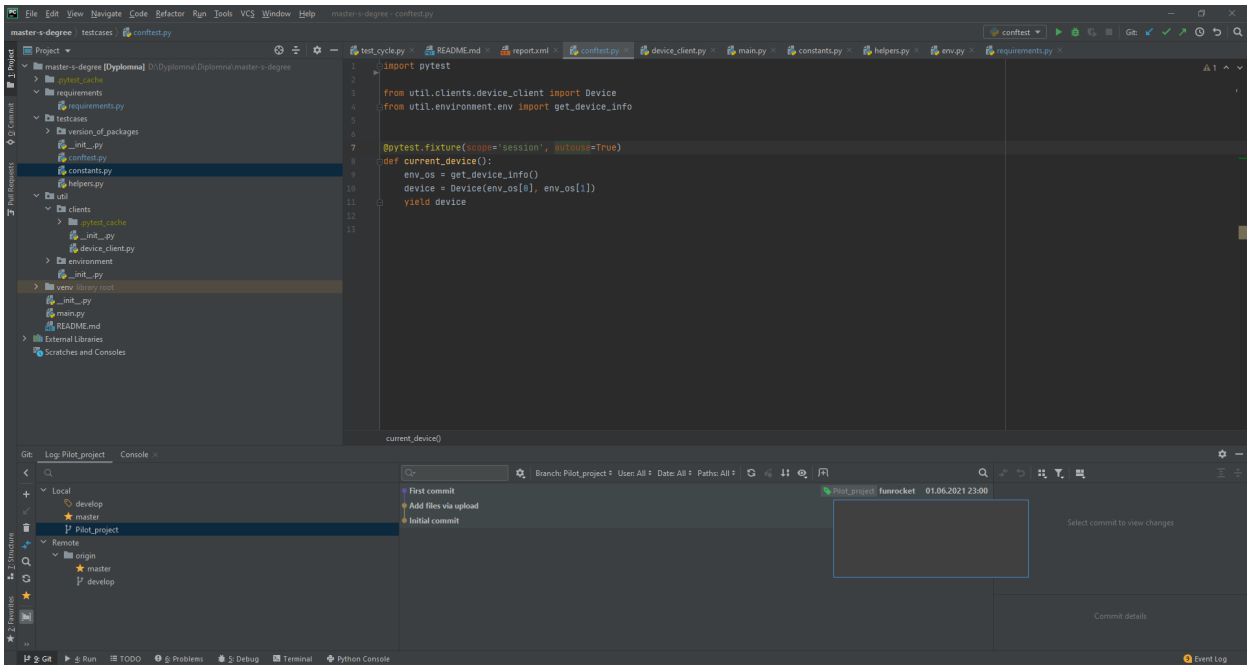

Рис. 3.5. Директива Testcases з файлами conftest.py, constants.py, helpers.py для виконання тесту

У файлі constants.py створено 2 класи, Windows та Linux, в яких існують по 2 константи : OS та Version. В випадку Windows це

```
class Windows:
    WINDOWS_OS = 'WINDOWS'
    WINDOWS_VERSION = 10
```

В випадку Linux це

```
class                                                    Linux:
             LINUX_OS                =                 'LINUX'
 LINUX_VERSION = 5
```

У файлі helpers.py виконується імпорт os модуля, та створено 2 функції search_for_components та check_the_internet.

У випадку check_the_internet, повертається значення (True або False), і використовується метод модуля os а саме system, якому ми передаємо наступну команду:

```
ping www.google.com
```

У випадку search_for_components в функцію передається сутність пристрою, та список компонентів. Під час виконання створюється пуста колекція, яка називається:

```
missed_components
```

яка буде зберігати відсутні компоненти. Після цього проходить перевірка на операційну систему, в залежності від поточного значення операційної системи, відбувається ітерація компонент, за допомогою os.system.

```
apt policy
```

Або у випадку Windows

```
sc query
```

Якщо компонент не знайдемо він додається до колекції, за допомогою

```
missed_components.append(_)
```

Під кінець ітерації повертає колекцію з значенням False, якщо всі компоненти були знайдені, або в вигляді колекції, яка буде мати значення True.

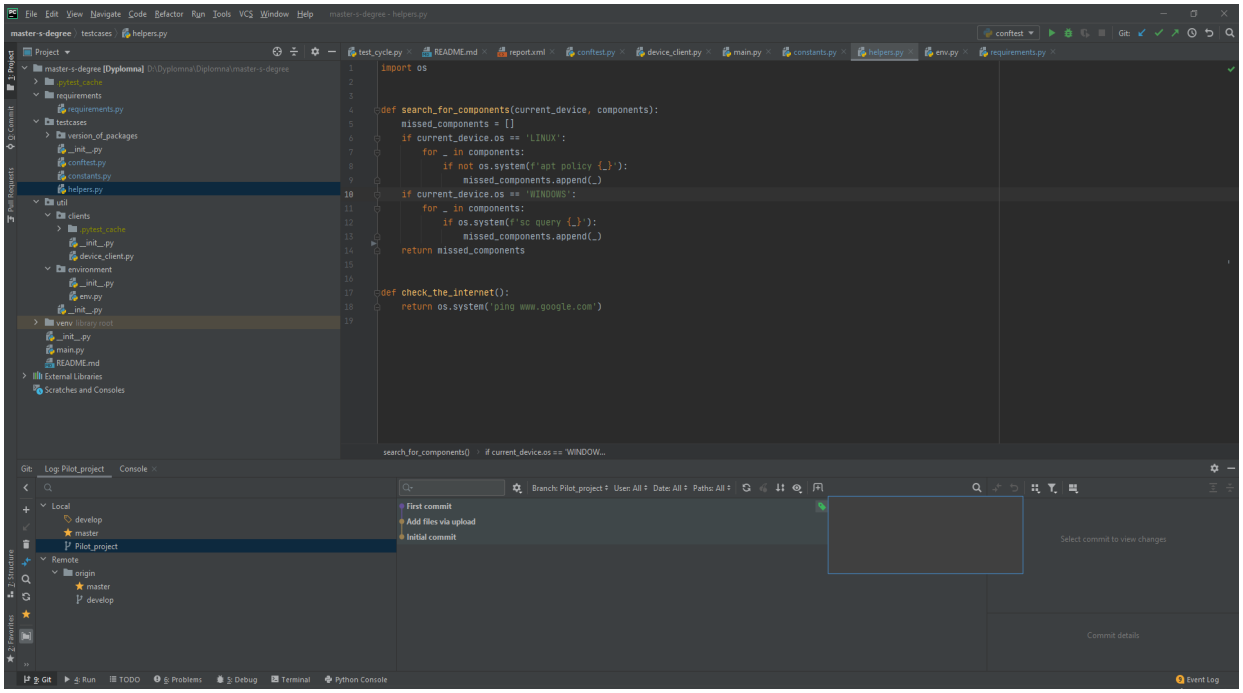

Рис. 3.6. Директива Testcases з файлом helpers.py для виконання тесту

Util

Дана директива має наступні під директиви:

– Clients

– Environment

У директиві Clients в нас знаходиться файл device_client.py, який виступає в ролі інстансу пристрою. У даному файлі оголошенно клас Device, та метод __init__(), який приймає наступні аргументи os та version. Дані аргументи задаються вже в самий клас Device через метод self:

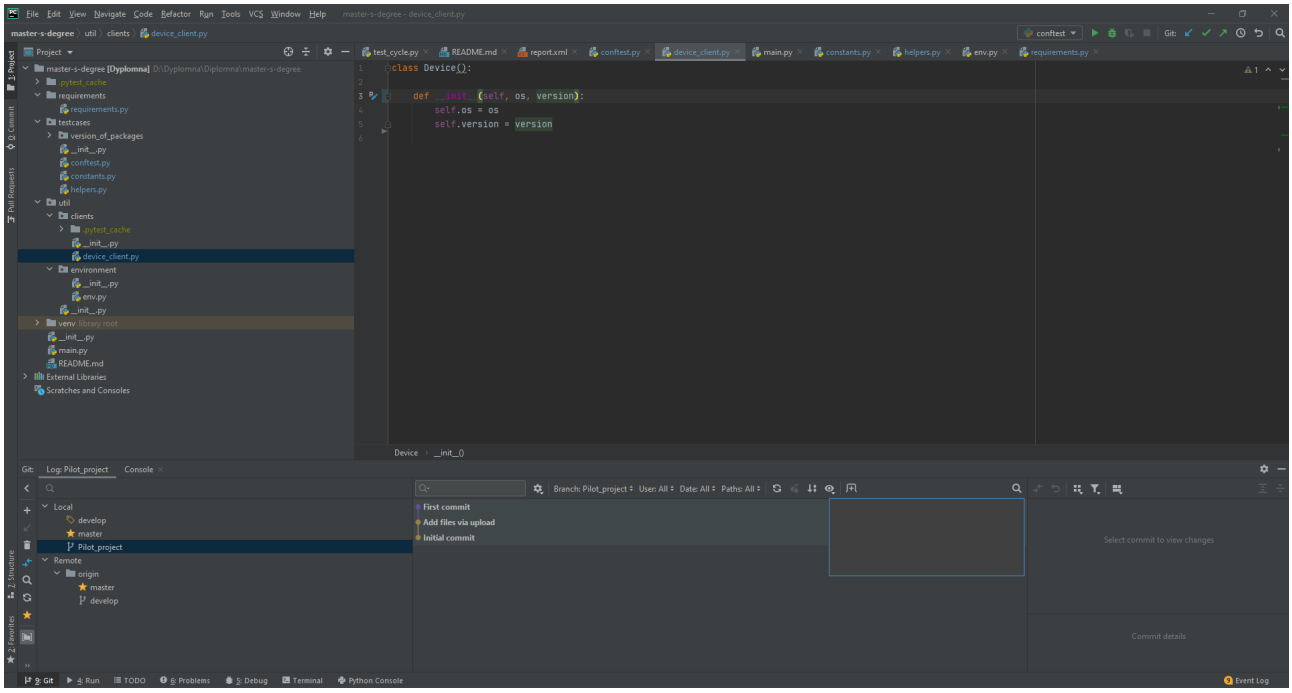

Рис. 3.7. Директива Util з піддирективою Clients і файлом device_client.py, з допомогою якого створюється сутність пристрою

У директиві Environment створенно файл env.py. У цьому файлі відбувається імпорт модулів os та platform, а також констант. Функція get_device_info в файлі виконує наступне:

У змінну operation_system задається значення поточної операційної системи, за допомогою метода platform.system(), також для цього значення ми використовуємо метод upper(), для того, щоб значення operation_system було одного формату як і в константах. Потім за допомогою if/elif перевіряємо чи це у нас Windows, чи Linux. У випадку Windows відбувається наступне:

```
if operation_system == Windows.WINDOWS_OS:
    version = platform.version()
 version = version[0:4]
```

У змінну version задається значення результату виконання метода platform.version(), а після цього переоприділяється, через задання тільки 4 цифр.

У випадку Linux відбувається наступне:

```
elif operation_system == Linux.LINUX_OS:
        version = sys.version
    version = version.split('Linux-')[1]
 version = version[0:4]
```

Як і у випадку Windows, у змінну version задається значення результату виконання метода, але вже sys.version, після цього, нам необхідно відділити слова Linux від значення версію, за допомогою split(), а після цього, як і у випадку Windows, ми беремо 4 цифри.

Після виконання функції, вона нам поверне значення змінної operation_system та version

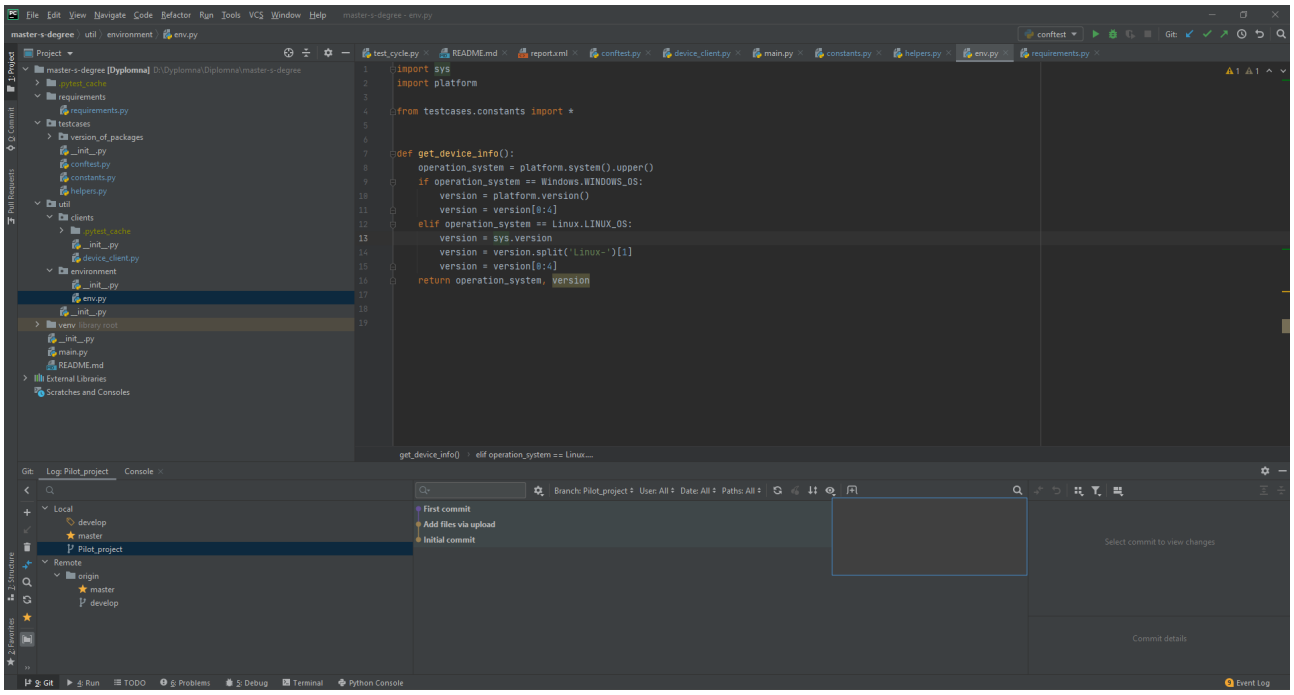

Рис. 3.8. Директива Util з піддирективою Environment і файлом env.py, з допомого якого відбувається імпорт модулів

Виконання програми:

Для того щоб запустити програму необхідно з кореня проекту викликати термінал та запустити наступний командний рядок

pytest –k test —alluredir ~\master–s–degree\testcases\version_of_packages\Report

Після запуску даної команди запуститься виконання нашої програми, яке буде виглядати наступним чином:

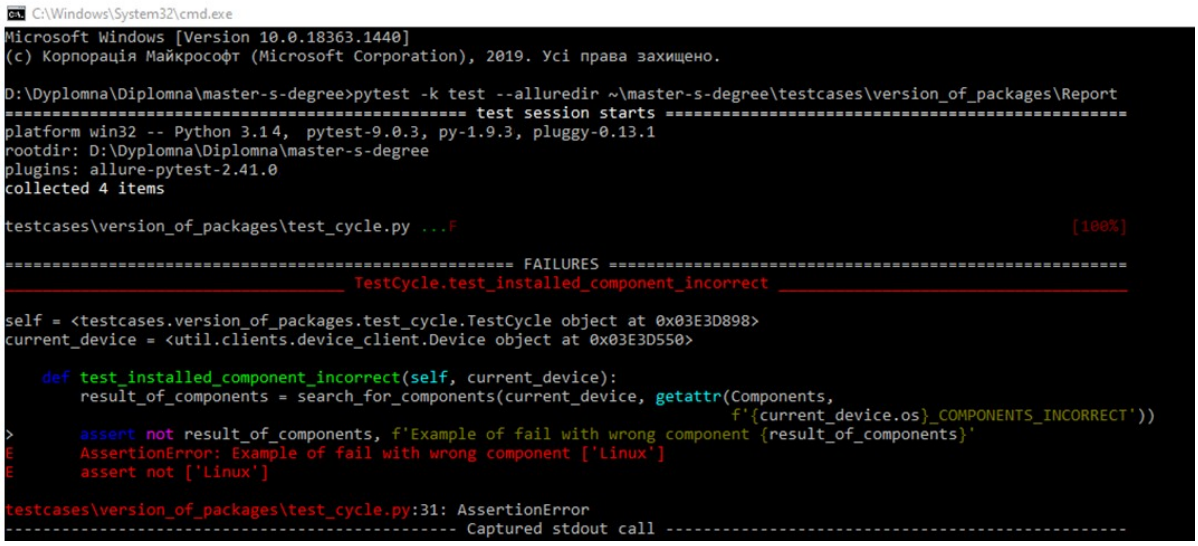

Рис. 3.9. запустити програму.

Після виконання усіх тестів, має з’явитись наступне повідомлення:

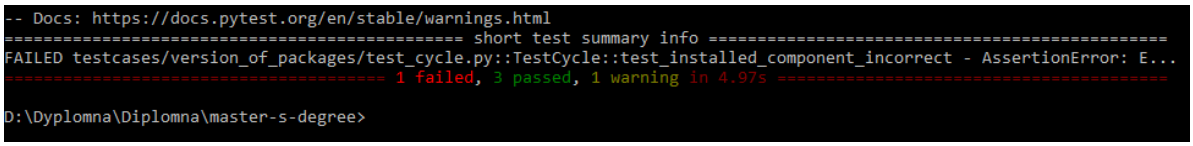

Рис. 3.10 Повідомлення виконання тестів

Де failed неуспішне виконання, passed успішне. Також у директиві Report повинні з’явитись файли для подальшого використання їх, для створення Allure репорту

Allure репорт

Для створення Allure репорту необхідно зробити наступне:

У терміналі, з корення проекту, запустити наступну команду:

allure serve ~\master–s–degree\testcases\version_of_packages\Report

Після запуску у терміналі повинно з’явитись наступне повідомлення:

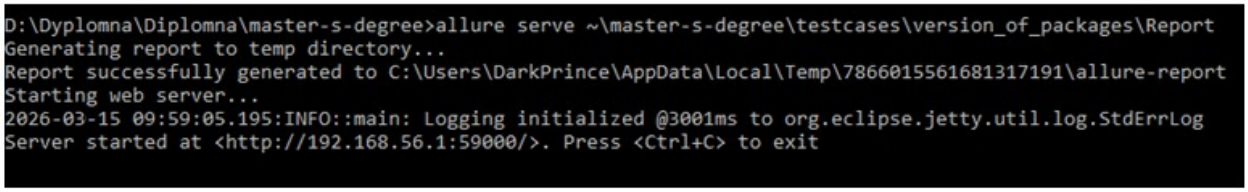

Рис. 3.11 Запуск у терміналі

Після цього, нас повинно перенести у Браузер з відкритим Allure репортом, який повинен виглядати наступним чинном:

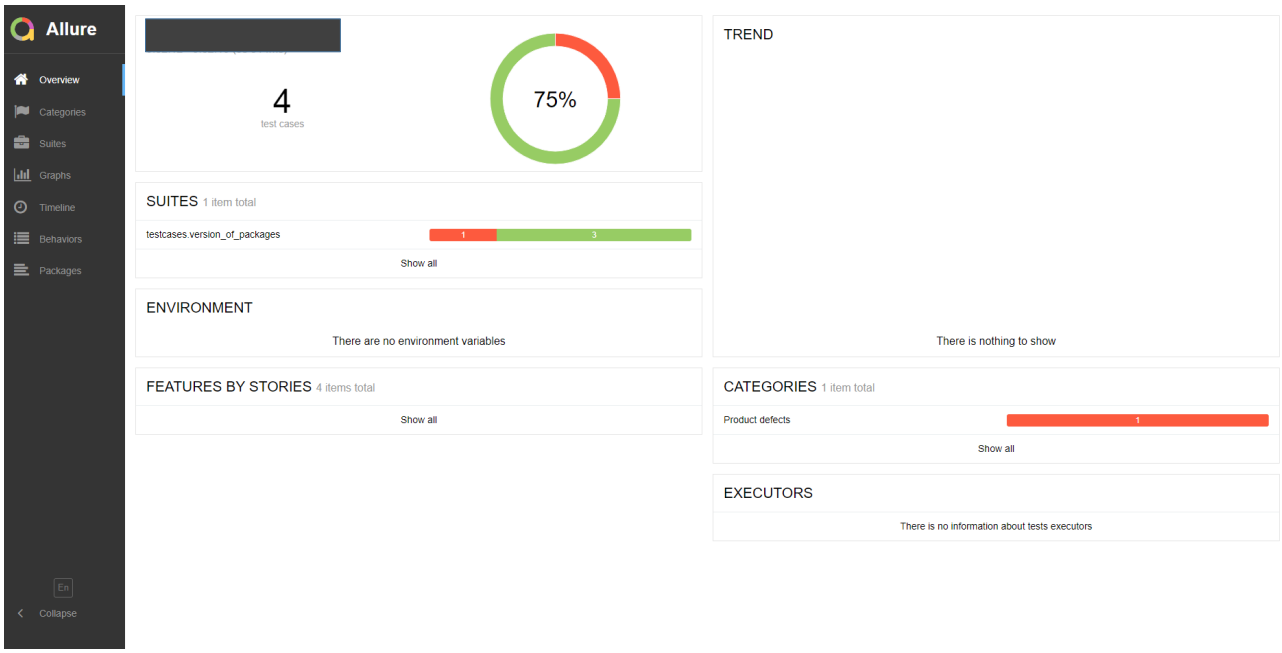

Рис. 3.15 Allure репорт

У даному репорті показано дата виконання:

ALLURE REPORT 3/15/2026

Час виконання:

9:52:12 – 9:52:16 (3s 514 ms)

Кількість тестів та відсоток успішних\неуспішних тестів.

Якщо ми натиснемо на Show All у категорії SUITES, нас перенесе на наступну сторінку:

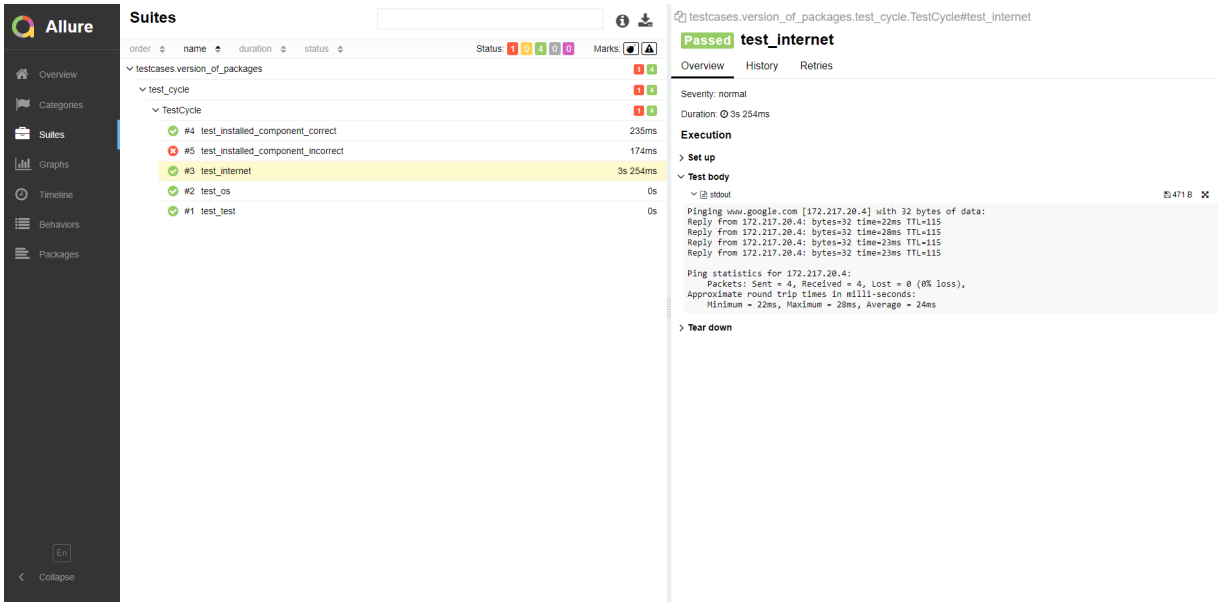

Рис 3.16 Show All

На даній сторінці є більш детальна інформація про виконання, така як:

– Назва тесту
– Скільки часу виконувався кожен тест
– Який з тестів неуспішний
– Test body показує нам результат виконання команди

Також на цій сторінці у нас є фільтри для відбору по певним категоріям. Наприклад тільки успішні або тільки неуспішні.

Розроблений програмний комплекс успішно реалізує повний цикл автоматизованої перевірки: визначення типу та версії операційної системи, верифікацію критичних системних компонентів, перевірку мережевої доступності та генерацію детальних інтерактивних звітів за допомогою Allure. Проведено тестування програмного продукту в реальних середовищах Windows та Linux, що підтвердило його працездатність, стабільність та відповідність поставленим вимогам.

Реалізоване рішення є кросплатформенним, модульним, легко розширюваним та може бути використане як самостійний інструмент діагностики системного середовища, так і як складова частина більш комплексних систем автоматизованого тестування та CI/CD.

ВИСНОВКИ

У бакалаврській роботі вирішено актуальне науково-практичне завдання моделювання процесу тестування програмного забезпечення.

Мета дослідження, яка полягала у створенні моделей та програмних засобів, що сприяють підвищенню якості програмного забезпечення через ефективне моделювання та автоматизацію процесу тестування системних компонентів, була повністю досягнута.

У ході виконання роботи вирішено всі поставлені завдання:

Проведено аналіз предметної області, методологій розроблення програмного забезпечення та принципів тестування, розглянуто моделі життєвого циклу розробки (SDLC) та ключові стандарти якості (ISO/IEC 25010).

Виокремлено фактори впливу на якість програмного забезпечення, побудовано семантичну мережу їх взаємозв’язків та багаторівневу ієрархічну модель пріоритетності.

Запроєктовано матрицю попарних порівнянь факторів та розраховано їх вагові коефіцієнти за методом аналізу ієрархій Сааті, що дозволило встановити найвищий пріоритет факторів «Захищеність» та «Рівень тестування».

Сформовано вимоги до інформаційної системи та розроблено комплекс UML-діаграм (діаграму варіантів використання та діаграму діяльності).

Розроблено програмний комплекс для автоматизованої перевірки системних вимог операційних систем Windows та Linux на основі Python 3.14.8, фреймворку pytest 9.0.3 та інструменту генерації звітів Allure 2.41.0.

Реалізовано модульну архітектуру проєкту з чітким розділенням відповідальностей, кросплатформенний збір інформації про середовище,

алгоритми перевірки наявності системних компонентів та мережевої доступності.

Проведено налаштування тестового середовища та успішне виконання тестів з генерацією інтерактивних Allure-звітів.

Наукова новизна роботи полягає в комплексному поєднанні теоретичного моделювання факторів якості програмного забезпечення з практичною реалізацією інструменту автоматизованої діагностики системного середовища.

Практична значущість розробленого програмного забезпечення полягає в автоматизації та об’єктивізації процесу верифікації системних вимог, що дозволяє суттєво скоротити час діагностики середовища, зменшити вплив людського фактора та підвищити надійність розгортання програмних продуктів. Програмний продукт може бути використаний розробниками, DevOps-інженерами, системними адміністраторами, а також у навчальному процесі при вивченні дисциплін з автоматизованого тестування та системного програмування.

Перспективи подальших досліджень пов’язані з розширенням функціональності системи на підтримку інших операційних систем, впровадженням додаткових рівнів перевірок (апаратних ресурсів, залежностей пакетів), інтеграцією з системами моніторингу та вдосконаленням алгоритмів аналізу результатів тестування за допомогою методів машинного навчання.

Таким чином, виконана робота підтверджує ефективність запропонованого підходу до автоматизації тестування системних компонентів та вносить певний внесок у забезпечення якості сучасного програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Канер С. Тестування програмного забезпечення : пер. з англ. / С. Канер, Дж. Фолк, Е. Кек Нгуєн. – К. : ДіаСофт, 2015. – 544 с.

2. Software Testing Life Cycle (STLC) GeeksForGeeks. URL: https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/

3. Iryna T. Research results of functional, white box and smoke testing methods for mobile applications / T. Iryna, K. Maksym // Trends in science and practice of today. – 2021. – Vol. 5. – P. 418.

4. Principles of Software Testing with Examples URL: https://www.guru99.com/software-testing-seven-principles.html

5. Коробейник А. Н. Короткі основи тестування програмного забезпечення / А. Н. Коробейник. – Київ : Директ-лайн, 2018.

6. Тестування програмного забезпечення URL: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення

7. Котляров В. П. Основи тестування програмного забезпечення / В. П. Котляров. – М. : БІНОМ, 2016.

8. Patton R. Software Testing / R. Patton. – 2nd ed. – Sams Publishing, 2015.

9. Engineering Trustworthy Secure Systems / Ron Ross, Mark Winstead, Michael McEvilley // NIST Special Publication 800-160. – Vol. 1, Rev. 1. URL: https://doi.org/10.6028/NIST.SP.800-160v1r1

10. Allen L. Advanced Penetration Testing for Highly-Secured Environments / L. Allen, K. Cardwell. – 2nd ed. – Packt Publishing, 2016.

11. Рівні тестування Quality Assurance Group. URL: https://www.quality-assurance-group.com/rivni-testuvannya/

12. Мова програмування. URL: https://uk.wikipedia.org/wiki/Мова_програмування

13. Andrews M. How to Break Web Software / M. Andrews, J. A. Whittaker. – Addison-Wesley, 2006.

14. Операційні системи : навчальний посібник / І. М. Федотова-Півень, І. В. Миронець, О. Б. Півень [та ін.] ; за ред. В. М. Рудницького ; Черкаський держ. технол. ун-т. – Харків : ТОВ «ДІСА ПЛЮС», 2019. – 216 с.

15. Operating Systems and You: Becoming a Power User. URL: https://www.coursera.org/learn/os-power-user/home/welcome (

16. The Python Standard Library — Platform Access.Python Documentation. – 2025. – URL: https://docs.python.org/3/library/platform.html

17. Summerfield M. Programming in Python 3 / M. Summerfield. – 2nd ed. – Addison-Wesley, 2024.

18. Minimal Python Package Structure . Python Packaging User Guide. URL: https://python-packaging.readthedocs.io/en/latest/minimal.html

19. pip – The Python Package. PyPI. – URL: https://pypi.org/project/pip/

20. ISO/IEC 25010:2011. Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models. – URL: https://www.iso.org/standard/35733.html

21. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. – URL: https://www.iso.org/standard/78176.html

22. pytest: helps you write better programs. Pytest.org. – URL: https://pytest.org

23. Allure Report. URL: https://allurereport.org/

24. Different Levels of Testing in Software Testing. Reqtest. URL: https://reqtest.com/testing-blog/different-levels-of-testing/

25. Шаховська Н. Б. Проектування інформаційних систем : навч. посібник / Н. Б. Шаховська, В. В. Литвин. – Львів : Магнолія-2006, 2022. – 380 с.

26. Зеленський К. Х. Моделювання систем : навч. посібник / К. Х. Зеленський, Є. А. Настенко, В. А. Павлов. – Київ : КПІ ім. Ігоря Сікорського, 2022. – 295 с.