МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп’ютерної інженерії

ПЕРЕПЕЛЮК Станіслав Андрійович

Програмний модуль автоматизованого тестування на основі поведінкового підходу / Software module for automated testing based on a behavioral approach

спеціальність: 123 – Комп’ютерна інженерія
освітньо-професійна програма – Комп’ютерна інженерія

Кваліфікаційна робота

Виконав: студент групи КІ-42
Перепелюк Станіслав Андрійович

Науковий керівник
к.т.н. Гураль І.В.

ТЕРНОПІЛЬ - 2025

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль автоматизованого тестування на основі поведінкового підходу » зі спеціальності 123 «Комп'ютерна інженерія» освітнього ступеня «бакалавр» містить 65 сторінок пояснюючої записки, 9 рисунків та 4 додатки. Обсяг графічного матеріалу 2 аркуші формату А3.

Мета роботи полягає у розробці програмного модуля автоматизованого тестування програмного забезпечення з використанням поведінкового підходу, що дозволяє підвищити якість перевірки функціональності системи та забезпечити прозорість тестових сценаріїв.

Методи дослідження включають аналіз науково-методичної літератури у сфері тестування, порівняльний аналіз інструментів поведінкового тестування, проєктування архітектури модуля, реалізацію функціоналу за допомогою Java, Cucumber, Allure та Jenkins, а також експериментальну перевірку працездатності модулю в умовах реального тестового середовища.

У роботі розглянуто сучасні підходи до BDD-тестування, описано процес реалізації тестового модуля з підтримкою форматів Gherkin, REST API, генерації звітів та візуалізації результатів.

Результатом є створений програмний модуль, що дозволяє ефективно будувати, запускати та верифікувати BDD-сценарії, інтегруватися з CI/CD-інфраструктурою, формувати інтерактивні звіти, відслідковувати помилки й підтримувати історію запусків.

Ключові слова: ПРОГРАМНИЙ МОДУЛЬ, ПОВЕДІНКОВЕ ТЕСТУВАННЯ, BDD, CUCUMBER, JAVA, АВТОМАТИЗАЦІЯ, ALLURE, JENKINS, REST API, CI/CD.

ANNOTATION

The qualification paper titled «Software module for automated testing based on a behavioral approach», specialty 123 «Computer Engineering», Bachelor's degree, contains 65 pages of explanatory note, 9 figures and 4 appendices. The graphical material volume is 2 A3-format sheets.

The aim of the work is to develop a software module for automated software testing using the behavioral approach, which enables improving the quality of functionality verification and ensuring transparency of test scenarios.

The research methods include analysis of scientific and methodological literature in the field of testing, comparative analysis of behavioral testing tools, design of the module architecture, implementation of functionality using Java, Cucumber, Allure, and Jenkins, as well as experimental verification of the module’s performance in a real test environment.

The paper explores modern approaches to BDD testing, describes the implementation of a testing module with support for Gherkin syntax, REST API, report generation, and result visualization.

The outcome is a developed software module that enables efficient construction, execution, and verification of BDD scenarios, integration with CI/CD infrastructure, creation of interactive reports, error tracking, and maintenance of execution history.

Keywords: SOFTWARE MODULE, BEHAVIOR-DRIVEN TESTING, BDD, CUCUMBER, JAVA, AUTOMATION, ALLURE, JENKINS, REST API, CI/CD.

ЗМІСТ

ВСТУП

На сьогоднішній день автоматизоване тестування стає невіддільною складовою процесу розробки програмного забезпечення в умовах Agile- і DevOps-методологій. Серед існуючих підходів до автоматизованого тестування особливу роль відіграє поведінкове тестування (BDD – Behavior-Driven Development), що забезпечує тісну інтеграцію між технічними та бізнес-вимогами, дозволяючи формалізувати очікувану поведінку системи у вигляді зрозумілих сценаріїв. Впровадження такого підходу сприяє зниженню ризику непорозумінь між розробниками, тестувальниками та замовниками, а також забезпечує високий рівень автоматизації перевірок.

Актуальність теми полягає в необхідності впровадження уніфікованих підходів до розробки тестів, що відповідають вимогам як розробників, так і бізнес-аналітиків. Такий підхід дозволяє суттєво зменшити витрати на тестування, підвищити якість коду та швидкість його виведення у продуктивне середовище.

Об’єктом дослідження є процес автоматизованого тестування програмного забезпечення. Предметом дослідження є методика побудови програмного модуля для автоматизованого тестування з використанням поведінкового підходу.

Мета бакалаврської роботи – розробити програмний модуль автоматизованого тестування програмного забезпечення з використанням поведінкового підходу, що дозволяє підвищити якість перевірки функціональності системи та забезпечити прозорість тестових сценаріїв.

Для досягнення поставленої мети у роботі необхідно вирішити наступні завдання:

– проаналізувати сучасні підходи до автоматизованого тестування та особливості поведінкового підходу;

– провести огляд існуючих інструментів, що підтримують BDD-тестування;

– сформувати вимоги до програмного модуля автоматизованого тестування;

– спроєктувати архітектуру модуля з урахуванням обраного підходу та інструментів;

– реалізувати поведінкові сценарії на мові програмування Java;

– провести верифікацію результатів роботи модуля, зокрема перевірити ефективність тестування та якість звітності.

Результати проведеного дослідження та практичної реалізації програмного модуля були опубліковані на XII Всеукраїнській науково-практичній конференції молодих учених «Інформаційні технології – 2025» [1]. У додатку А наведено світлокопію публікації.

За структурою бакалаврська кваліфікаційна робота складається з трьох розділів [2–6].

У першому розділі здійснено теоретичний аналіз особливостей автоматизованого тестування програмного забезпечення, обґрунтовано переваги поведінкового підходу та виконано огляд сучасних інструментів, що реалізують BDD-методологію.

У другому розділі наведено аналіз функціональних і нефункціональних вимог до програмного модуля, спроєктовано його архітектуру та виконано моделювання поведінкових сценаріїв відповідно до вимог до системи.

У третьому розділі представлено практичну реалізацію модулю з використанням мови Java та технологій Cucumber, Selenide, Allure; розроблено автоматизовані сценарії, здійснено тестування функціональності та оцінено ефективність впровадження поведінкового підходу на практиці.

1 ОСОБЛИВОСТІ ПОВЕДІНКОВОГО ПІДХОДУ В АВТОМАТИЗОВАНОМУ ТЕСТУВАННІ

1.1 Характеристика автоматизованого тестування програмного забезпечення

Автоматизоване тестування (АТ) на сьогодні є невід'ємною частиною сучасного процесу створення програмного забезпечення. Зі зростанням складності програмних систем, підвищенням очікувань кінцевих користувачів, а також постійним прагненням компаній до прискорення випуску якісного продукту на ринок, роль АТ лише посилюється. Це обумовлює необхідність глибокого розуміння його сутності, можливостей та ефективного впровадження у практику комп'ютерної інженерії [7].

АТ полягає у використанні спеціалізованих програмних засобів, які автоматично виконують попередньо написані тестові сценарії [8]. На відміну від ручного тестування, де всі дії виконує людина, автоматизоване тестування дозволяє досягати високої точності повторного виконання сценаріїв, уникати помилок, викликаних людським фактором, і значно пришвидшити процес виявлення дефектів. Програмні тести, які автоматично запускаються під час кожної зміни в коді, не лише зберігають час команди, а й забезпечують впевненість у стабільності програмного продукту [9].

У межах гнучких методологій розробки, таких як Agile та DevOps, де релізи відбуваються часто, а ітеративна природа змушує команду постійно змінювати і вдосконалювати продукт, ручне тестування стає недостатнім. Автоматизація тут виступає запорукою стабільності. Наприклад, регресійне тестування, яке вимагає повторної перевірки вже реалізованого функціоналу після кожної зміни, без автоматизації потребує значних зусиль і часу. Автоматизовані тести здатні за лічені хвилини перевірити тисячі варіантів поведінки системи, що фізично неможливо досягти вручну [10].

Крім пришвидшення, АТ дозволяє забезпечити більш широке покриття коду. Якщо ручний тестовий сценарій зазвичай перевіряє лише кілька позитивних і негативних випадків, автоматизовані тести можуть охопити глибину логіки: перевірити всі гілки умов, взаємодії між модулями, виняткові ситуації та помилки введення. Це забезпечує вищу якість продукту на виході. Також варто зазначити, що автоматизоване тестування може бути інтегроване в пайплайни безперервної інтеграції та доставки (CI/CD), забезпечуючи автоматичну перевірку змін перед їх потраплянням у реліз [11].

Важливим фактором є повторюваність тестів. Один і той самий сценарій можна виконати стільки разів, скільки необхідно, на будь-якому середовищі – розробки, тестування або передпродакшн [12]. Це забезпечує консистентність результатів. Ручне ж тестування у свою чергу схильне до варіативності результатів залежно від навичок або уваги конкретного тестувальника. Повторюваність критична, особливо у великих командах, де якість програмного продукту повинна бути прогнозованою незалежно від того, хто виконує тест.

Розглядаючи питання ефективності автоматизованого тестування, важливо усвідомлювати, що його впровадження – це не лише технічне завдання, а й управлінське рішення. Тестування потребує інвестицій у ресурси, вибір правильних інструментів, планування структури тестів і навчання команди. Написання автоматизованих тестів вимагає певного технічного рівня, знання мов програмування, а також глибокого розуміння архітектури системи, що тестується. Проте, правильно організоване тестування з часом окупається: кожен новий тест, що додається до автоматизованої системи, збільшує загальне покриття, підвищує впевненість у працездатності системи та зменшує час на ручну перевірку [13].

У процесі автоматизації тестування важливе значення має вибір типів тестів. Найпоширенішими є юніт-тести, які перевіряють окремі функції або методи; інтеграційні тести, що оцінюють взаємодію між модулями; системні тести, які перевіряють поведінку системи в цілому, і тести приймання, що фокусуються на вимогах бізнесу [14]. Кожен з них відіграє важливу роль у загальній структурі

якості програмного забезпечення. Оптимальна стратегія автоматизації зазвичай включає поєднання різних рівнів тестування, щоб забезпечити як глибину, так і ширину перевірки.

У процесі створення автоматизації тестування зазвичай виконуються наступні кроки (рисунок 1.1):

– вибір тестового інструменту;
– визначення області автоматизації;
– планування, проєктування та розробка;
– виконання тестів;
– технічне обслуговування (підтримка).

Рисунок 1.1 – Кроки створення автоматизації на часовій шкалі

З цих п’яти кроків найбільшу увагу заслуговують перші два, оскільки від визначення того що ми можемо протестувати автоматично, а також в залежності від вибору тестового інструменту буде залежати абсолютно всі подальші кроки.

Сучасні інструменти для автоматизації охоплюють широкий спектр платформ і технологій. Вони варіюються від простих скриптових рішень до складних фреймворків, які підтримують поведінкове тестування, генерацію звітів, інтеграцію з системами контролю версій і хмарними сервісами. У світі Java, наприклад, популярними є JUnit, TestNG, Selenium, Cucumber, Rest Assured тощо.

У виборі інструментів важливими є такі критерії, як легкість налаштування, підтримка необхідних протоколів і платформ, можливість паралельного виконання, підтримка звітності та інтеграція з CI-середовищами [15].

Окремої уваги заслуговує вплив автоматизації на командну динаміку. У багатьох командах, де раніше існувала чітка межа між розробниками і тестувальниками, автоматизоване тестування сприяє їх інтеграції [16]. Розробники починають писати тести разом з кодом, а тестувальники отримують більше можливостей впливати на якість програмного забезпечення, пропонуючи сценарії, які відображають реальні випадки використання. Таким чином, автоматизація сприяє колаборації, прозорості процесів і підвищенню загального рівня зрілості команди.

Незважаючи на численні переваги, слід зазначити, що автоматизоване тестування не є панацеєю. Його використання доцільне у випадках, коли система має стабільний інтерфейс, тестові сценарії не змінюються надто часто, а вартість розробки автоматизованого тесту нижча, ніж багаторазове виконання ручного. У проєктах, які активно змінюються, або з нестабільними вимогами, витрати на підтримку автоматизованих тестів можуть бути занадто високими. Проте, навіть у таких умовах, автоматизація окремих компонентів (наприклад, API або базових сервісів) уже дає значні переваги [17].

Розвиток підходів до автоматизації не стоїть на місці. Останні роки набули широкого поширення такі напрямки, як тестування на основі моделі (MBT – Model-Based Testing), тестування, кероване поведінкою (BDD – Behavior-Driven Development), а також автоматизація за допомогою штучного інтелекту. Кожен із них розширює можливості традиційного тестування, дозволяючи досягати ще вищої якості при менших витратах часу та ресурсів [18].

Впровадження автоматизованого тестування – це стратегічний крок, що потребує усвідомлення цілей, ресурсів і етапів реалізації. На початкових етапах доцільно визначити найважливіші компоненти системи, які слід автоматизувати першочергово. Це можуть бути критичні бізнес-процеси, точки інтеграції або

елементи, що найчастіше змінюються. Подальше розширення покриття має відбуватись поступово, із регулярною переоцінкою ефективності, аналізом помилок і постійним удосконаленням підходів.

Таким чином, автоматизоване тестування є не лише інструментом забезпечення якості, а й елементом стратегії цифрової трансформації компанії. Воно дозволяє зменшити ризики, підвищити стабільність релізів, забезпечити конкурентну перевагу завдяки швидшій доставці продукту та сприяє створенню культури якості у команді. Його правильне впровадження і використання стає ключовим чинником успішної розробки сучасних інформаційних систем.

1.2 Аналіз основних принципів та підходів поведінкового тестування

Сучасні методології розробки програмного забезпечення передбачають не лише високу швидкість створення продукту, але й безперервну комунікацію між різними учасниками процесу – розробниками, тестувальниками, бізнес-аналітиками та кінцевими користувачами. У цьому контексті поведінкове тестування (Behavior-Driven Development, BDD) постає як ефективна стратегія, що дозволяє об’єднати вимоги замовника із технічним втіленням проєкту за допомогою формалізованої, але зрозумілої для всіх учасників мови (рисунок 1.2). BDD ґрунтується на простій, але потужній ідеї – описувати бажану поведінку системи зрозумілою мовою, яка може бути автоматично переведена у програмні тести [19]. Такий підхід змінює саму суть процесу тестування – від перевірки коду до перевірки очікуваної поведінки системи.

Поведінкове тестування виникло як розвиток практики тестування, керованого розробкою (TDD – Test-Driven Development), і є його логічним продовженням. Якщо в основі TDD лежить створення тестів ще до написання коду, то BDD додає до цього фокус на поведінку системи з точки зору бізнесу.

Рисунок 1.2 – Стратегія автоматизованого тестування на основі поведінкового підходу

Ключовим у поведінковому тестуванні є взаєморозуміння: сценарії тестування пишуться мовою, зрозумілою всім членам команди, і використовуються не тільки як інструмент перевірки, але і як документація, специфікація і засіб комунікації [20]. У результаті, BDD сприяє створенню прозорого і зрозумілого програмного продукту, який відповідає бізнес-вимогам.

Опис поведінки системи у BDD зазвичай здійснюється за допомогою мови Gherkin, яка дозволяє формалізовано, але зрозуміло описати сценарії взаємодії користувача із системою. Gherkin використовує ключові слова «Дано», «Коли» та «Тоді» (Given, When, Then), які описують передумови, дію і очікуваний результат [21]. Такий підхід не лише уніфікує структуру сценаріїв, але й забезпечує зручність для автоматичного оброблення. Наприклад, сценарій входу в систему може виглядати як: «Дано, що користувач перебуває на сторінці входу; Коли він вводить правильне ім’я користувача і пароль; Тоді він потрапляє на головну сторінку». Такий опис може бути прочитаний як технічною, так і нетехнічною аудиторією, і водночас слугувати тестом, який запускається у середовищі виконання.

Одна з головних переваг поведінкового тестування полягає у його здатності виступати посередником між бізнес-вимогами і технічною реалізацією. Зазвичай розробники мають глибоке розуміння технічної частини, але не завжди враховують

усі нюанси очікуваної поведінки з точки зору користувача або бізнесу. Аналогічно, бізнес-аналітики чи менеджери продукту не завжди можуть оцінити технічні складності реалізації певної функції. BDD дозволяє поєднати ці точки зору шляхом створення узагальненого опису функціональності, який згодом трансформується у кодифікований тест. У такий спосіб зменшуються ризики непорозумінь, а також підвищується якість кінцевого продукту [22].

Принцип роботи BDD ґрунтується на безперервній взаємодії трьох сторін: розробника, тестувальника та бізнес-аналітика. Вони разом обговорюють очікувану поведінку системи та погоджують відповідні сценарії. Саме цей процес часто називається «прикладно-керованою розробкою» (example-driven development), коли приклади стають основою для всієї подальшої реалізації. Такий підхід стимулює обговорення, дозволяє виявити потенційні помилки ще на етапі планування та сприяє кращому розумінню вимог [23].

Технічно реалізація BDD зазвичай здійснюється за допомогою спеціалізованих фреймворків. У середовищі Java одним із найпопулярніших є Cucumber – бібліотека, яка дозволяє запускати сценарії Gherkin та прив’язувати їх до Java-коду через спеціальні методи, що називаються «step definitions» [24]. Іншими словами, кожен рядок сценарію відповідає певній функції в коді, яка реалізує відповідну дію. Такий механізм дозволяє зберігати чисту архітектуру тестів, забезпечує гнучкість і масштабованість, а також робить тести зрозумілими для всіх учасників процесу.

Ще однією важливою особливістю підходу BDD є його здатність служити живою документацією. Тестові сценарії, написані мовою Gherkin, не потребують додаткового перекладу в технічну документацію, оскільки самі по собі є джерелом істини щодо функціональності системи [25]. Вони легко читаються, можуть бути переглянуті у будь-який момент і постійно актуалізуються. Це дозволяє зменшити витрати на підтримку документації, а також зробити її більш доступною для нових членів команди.

У контексті автоматизації тестування поведінковий підхід забезпечує додаткову перевагу у вигляді масштабованості. Завдяки використанню єдиної структури для опису сценаріїв, легко створювати нові тести, повторно використовувати вже існуючі компоненти, а також підтримувати послідовність стилю. Наприклад, одна й та сама умова може бути застосована в кількох різних сценаріях, що суттєво зменшує дублювання коду. Це забезпечує не лише зручність, але й підвищує ефективність процесу тестування.

Окремо варто наголосити на впливі BDD на забезпечення якості програмного забезпечення. Створення тестів до початку розробки змушує команду глибше замислитися над вимогами, передбачити можливі виключення та нестандартні ситуації. Таким чином, сам процес планування стає більш ретельним. Крім того, постійне виконання поведінкових тестів на кожному етапі розробки дозволяє оперативно виявляти помилки та усувати їх ще до потрапляння у продакшн-середовище. Це зменшує загальну кількість дефектів, що потрапляють до користувача, і підвищує довіру до продукту.

Проте, як і будь-який підхід, поведінкове тестування має свої виклики. Зокрема, воно вимагає дисципліни у написанні сценаріїв, злагодженої роботи всієї команди та часу на навчання. Інколи розробники ставляться до Gherkin-сценаріїв як до зайвого кроку, оскільки вважають, що простіше написати звичайний тестовий метод. Однак у довгостроковій перспективі саме формалізованість та уніфікованість BDD дозволяють досягти кращої організації коду, прозорості та підтримуваності тестів. Тому успішне впровадження BDD вимагає не лише технічних інструментів, але і культурної трансформації команди.

Загалом, аналіз основних принципів та підходів поведінкового тестування дає підстави вважати його одним з найперспективніших напрямів автоматизованого забезпечення якості програмного забезпечення. Його суть полягає у поєднанні формальності і гнучкості, технічності і зрозумілості, простоти і масштабованості. Саме тому все більше проєктів, які орієнтуються на якість, вибирають BDD як стандарт розробки і тестування. Він дозволяє будувати процес

створення ПЗ навколо очікуваної поведінки користувача, а не лише технічної реалізації функціоналу, що зрештою забезпечує вищу якість, кращий користувацький досвід і ефективну командну роботу.

1.3 Аналіз сучасних інструментів для тестування на основі поведінкового підходу

У середовищі розробки програмного забезпечення інструментарій відіграє ключову роль у забезпеченні ефективності та якості усіх етапів життєвого циклу програмного продукту. Зокрема, у контексті поведінкового підходу до тестування (Behavior-Driven Development, BDD), вибір відповідного інструменту має визначальний вплив на гнучкість розробки, зрозумілість тестів, зручність роботи в команді та інтеграцію з іншими складовими інфраструктури. Враховуючи популярність та активне впровадження BDD у сучасних IT-проєктах, існує низка інструментів, що дозволяють реалізувати принципи поведінкового тестування, починаючи з опису поведінки системи, і завершуючи генерацією звітності про результати виконання тестів [26].

Серед найвідоміших інструментів для реалізації BDD варто виділити такі фреймворки, як Cucumber, SpecFlow, Behave, JBehave, Gauge, а також менш розповсюджені, але перспективні рішення, такі як Lettuce, Jasmine, pytest-bdd та інші. Кожен із них орієнтований на певну екосистему, має свої особливості інтеграції, мови підтримки та розширюваності. Загалом, ці інструменти об’єднує підтримка мови Gherkin або схожих синтаксисів, що дозволяє створювати тести у зрозумілому форматі Given-When-Then, який легко читається як розробниками, так і бізнес-аналітиками [27].

Cucumber є найвідомішим і найпоширенішим інструментом у середовищі Java, проте також має реалізації для JavaScript, Ruby, Kotlin, Groovy, Python, PHP

та інших мов. Він забезпечує повну підтримку мови Gherkin і дозволяє створювати зв’язок між сценаріями тестів та відповідною реалізацією кроків у вигляді Java-методів. Завдяки своїй гнучкості, Cucumber використовується як у невеликих проєктах, так і у великих корпоративних рішеннях, оскільки дозволяє централізовано зберігати .feature-файли, виконувати тести через різні рушії (JUnit, TestNG) і генерувати детальні звіти. Крім того, Cucumber легко інтегрується з такими інструментами як Maven, Gradle, Jenkins, GitLab CI/CD, що дозволяє реалізовувати повноцінну автоматизацію процесу тестування [28].

Особливістю Cucumber є те, що сценарії, написані мовою Gherkin, залишаються максимально незалежними від реалізації, що дозволяє не лише тестувати функціонал, але і використовувати їх як живу документацію. Таке використання значно спрощує онбординг нових учасників команди, оскільки вони можуть одразу ознайомитися з поведінкою системи через тестові сценарії. Також Cucumber підтримує локалізацію – сценарії можна писати українською, англійською або іншими мовами, що актуально для міжнародних команд.

Іншим популярним інструментом, особливо у світі .NET, є SpecFlow – фреймворк, створений за аналогією з Cucumber, але спеціалізований для C#-екосистеми [29]. SpecFlow працює в середовищі Visual Studio, підтримує Gherkin, має розширення для IDE, що дозволяють генерувати кроки сценаріїв автоматично, а також інтегрується з MSTest, NUnit, xUnit тощо. Його сильним боком є підтримка багатьох додаткових функцій, зокрема тегування сценаріїв, параметризації, розширеного аналізу помилок і звітності.

Для розробки на Python існує декілька реалізацій, але найбільш зрілою і популярною є Behave [30]. Цей інструмент дозволяє використовувати Gherkin-сценарії та пов’язувати їх із Python-кодом через так звані "step implementations". Behave відзначається простотою використання, зручною структурою каталогів та можливістю легкої інтеграції з існуючими Python-проєктами. Також він підтримує механізми тегування, параметризації та виконання підмножин сценаріїв. Behave

добре підходить для невеликих або середніх проєктів, де команда вже використовує Python як основну мову.

Ще один відомий інструмент – JBehave, який був одним із перших фреймворків для BDD у Java-середовищі [31]. Хоча з часом він поступився популярністю Cucumber, JBehave досі використовується у великих організаціях, що мають значну спадщину коду або специфічні вимоги. JBehave дозволяє писати сценарії у вигляді історій (story files), має гнучку систему конфігурації, підтримку перед- та післяумов, а також можливість глибокої кастомізації. Однією з його переваг є потужний механізм зв'язування історій із Java-кодом через анотації, що забезпечує контроль над порядком виконання та обробкою залежностей.

У випадках, коли необхідна максимальна легкість та швидкість, можна розглядати інструмент Gauge – ще один фреймворк, який підтримує концепцію BDD, проте пропонує альтернативу до Gherkin у вигляді Markdown-подібного синтаксису [32]. Gauge орієнтований на розробників і підтримує різні мови програмування, включаючи Java, JavaScript, C#, Ruby, Python та Go. Його перевагою є можливість створення багаторазових кроків, структурованість тестів, зручність масштабування, а також потужна система плагінів. Gauge має інтеграцію з VS Code, генерує HTML-звіти та добре підходить для проєктів з мікросервісною архітектурою.

Варто згадати і менш відомі, але корисні інструменти, такі як pytest-bdd для Python, Lettuce, Turnip, Jasmine (для JavaScript), які, хоч і мають обмежену підтримку спільноти або функціональність, можуть бути придатними у невеликих або спеціалізованих проєктах. Наприклад, Jasmine активно використовується у фронтенд-розробці для тестування поведінки веб-компонентів у поєднанні з фреймворками Angular або React. Його BDD-подібний синтаксис дозволяє організовувати структуру тестів у вигляді «describe-it» і є зручним для створення читабельних модульних тестів [33].

У результаті проведеного аналізу сучасних інструментів для поведінкового тестування було встановлено, що найбільш придатним для використання у межах

нашого проєкту є фреймворк Cucumber у поєднанні з мовою програмування Java. Такий вибір обумовлений високою популярністю Cucumber у Java-спільноті, повною підтримкою мови Gherkin, зручною інтеграцією з існуючими інструментами розробки (JUnit, Maven, IntelliJ IDEA), а також широкими можливостями для автоматизації процесів тестування та генерації звітності. Крім того, Cucumber дозволяє досягти високого рівня прозорості завдяки використанню читабельних сценаріїв, що полегшує співпрацю між розробниками, тестувальниками та бізнес-аналітиками. Застосування саме цього інструменту забезпечить ефективну реалізацію поведінкового підходу в нашому програмному модулі та сприятиме підвищенню загальної якості програмного забезпечення.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ІЗ ВИКОРИСТАННЯМ ПОВЕДІНКОВОГО ПІДХОДУ ТЕСТУВАННЯ

2.1 Аналіз вимог до програмного модуля тестування

Проєктування будь-якого програмного модуля розпочинається з чіткого та структурованого аналізу вимог, які до нього висуваються. У контексті побудови модуля для автоматизованого тестування на основі поведінкового підходу ці вимоги охоплюють не лише технічні аспекти реалізації, але й логіку взаємодії з користувачами, інтеграційні можливості, підтримку зрозумілих сценаріїв, а також забезпечення гнучкості та масштабованості розробленого рішення. Оскільки тестовий модуль є ключовим компонентом забезпечення якості програмного забезпечення, від правильного формулювання вимог залежить ефективність усього подальшого процесу створення та підтримки системи.

Одним із головних завдань нашого програмного модуля є забезпечення можливості створення, виконання та обслуговування автоматизованих тестів, які описують поведінку програмного забезпечення мовою, зрозумілою як технічним, так і нетехнічним фахівцям. Для досягнення цієї мети необхідно забезпечити підтримку мови Gherkin, яка дозволяє описувати сценарії тестування у вигляді структурованих речень, що логічно відображають дії користувача, умови виконання та очікуваний результат. Таким чином, основною вимогою до модуля є реалізація повноцінного механізму обробки .feature-файлів, який включає парсинг сценаріїв, зв’язування кроків з відповідними реалізаціями у коді та забезпечення керованого виконання кожного тесту в рамках відповідного середовища.

Особливу увагу слід приділити інтеграційним можливостям майбутнього модуля. Сучасні програмні проєкти використовують широкий спектр інструментів для автоматизації процесів розробки та доставки – від систем контролю версій до CI/CD-серверів, таких як Jenkins, GitLab CI, CircleCI тощо. Тому модуль має бути легко інтегрованим у ці пайплайни, з можливістю запуску тестів автоматично після

кожного коміту або перед релізом нового функціоналу. Це вимагає гнучкої конфігурації запуску, підтримки відповідних параметрів командного рядка, можливості зчитування змінних середовища та збереження результатів виконання у зручному форматі для подальшого аналізу.

Не менш важливою є вимога до генерації детальної звітності. Кожен запуск тестів має супроводжуватись створенням звіту, який відображає кількість виконаних сценаріїв, їхній статус, тривалість виконання, опис помилок у разі їх виникнення, а також, за можливості, скріншоти або логи. Для цього доцільно використати сучасні бібліотеки для формування звітів, зокрема Allure – популярний інструмент, що дозволяє створювати інтуїтивно зрозумілі, інтерактивні HTML-звіти з багатою візуалізацією. Це значно спрощує процес аналізу результатів та прийняття рішень щодо подальшого розвитку проєкту.

З огляду на специфіку поведінкового тестування, наш модуль повинен підтримувати концепцію «step definitions» – тобто реалізацію кожного кроку сценарію у вигляді окремого Java-методу, що забезпечує відповідну логіку взаємодії з програмою. Зокрема, кожен крок, описаний у форматі Given/When/Then, має відповідати певному методу з відповідною анотацією, яка дозволяє Cucumber ідентифікувати зв’язок між текстом у .feature-файлі та конкретною реалізацією. Це потребує організації відповідної структури пакунків, де будуть розміщуватись реалізації кроків, конфігураційні класи, утиліти та об’єкти сторінок або сервісів, з якими відбувається взаємодія.

Важливо також передбачити підтримку повторного використання кроків у різних сценаріях, параметризацію вхідних даних, а також механізми перед- та післяумов (hooks), які забезпечують підготовку середовища до тесту (наприклад, авторизацію, відкриття браузера) та його очищення після завершення (закриття сесії, видалення тестових даних тощо). Таке розширення функціональності є необхідною умовою створення гнучкої та масштабованої тестової архітектури.

Однією з ключових вимог є підтримка стабільності виконання тестів, особливо в умовах нестабільного мережевого середовища або взаємодії з

елементами інтерфейсу, які мають динамічний характер. Тому програмний модуль повинен містити механізми очікування (explicit waits), обробки винятків, повторних спроб виконання кроків (retry) та логування кожної дії. Це дозволить зробити тести більш надійними, зменшити кількість хибнопозитивних або хибнонегативних результатів, а також підвищити довіру до автоматизованої перевірки загалом.

Не менш важливою вимогою є простота конфігурації та можливість адаптації модуля до змін. У практиці розробки програмного забезпечення зміни відбуваються постійно – додаються нові функціональні можливості, змінюється інтерфейс, оновлюється логіка роботи. Тому структура модуля має дозволяти легко додавати нові сценарії, розширювати набір кроків, оновлювати конфігурації без необхідності переписувати значні частини коду. Для цього доцільно використовувати принципи об’єктно-орієнтованого програмування, шаблони проєктування (наприклад, Page Object Model) та практики розділення відповідальностей.

Особливу увагу слід приділити питанням підтримуваності і зрозумілості коду. Автоматизоване тестування – це не лише технічний процес, а й форма колективного знання про систему. Тому програмний модуль повинен бути написаний з урахуванням читаємості, документації, зрозумілих назв класів і методів, логічної структури тестів та коментарів у ключових місцях. Це дозволить новим учасникам команди швидко зорієнтуватись у тестовій архітектурі, вносити зміни без ризику порушення працездатності існуючих сценаріїв, а також забезпечить довготривалу підтримку модуля у великих або тривалих проєктах.

Крім технічних аспектів, слід враховувати і вимоги до взаємодії з користувачем. Хоча тестовий модуль переважно взаємодіє з системою автоматично, у деяких випадках може бути доцільним передбачити механізми конфігурації тестів вручну, запуску окремих сценаріїв з графічного інтерфейсу або з командного рядка, а також перегляду звітності у веб-браузері. Такий підхід сприяє демократизації процесу тестування – участі у ньому не лише технічних

спеціалістів, але й бізнес-аналітиків, менеджерів проєктів або навіть кінцевих користувачів.

У результаті проведеного аналізу функціональних і нефункціональних вимог до програмного модуля автоматизованого тестування на основі поведінкового підходу було сформовано уявлення про ключові компоненти, які необхідно реалізувати для досягнення поставлених цілей. Серед основних вимог – підтримка мови опису сценаріїв Gherkin, зв'язування цих сценаріїв з Java-реалізацією, можливість автоматизованого запуску тестів, генерація звітності та забезпечення інтеграції з CI/CD-середовищем. З боку нефункціональних вимог особливо важливими стали зручність використання, розширюваність архітектури, доступність результатів тестування та адаптація під середовище розробки команди.

Беручи до уваги вищезазначені вимоги, для реалізації програмного модуля у межах даного проєкту було обрано такі технології: мова програмування Java, як основна платформа реалізації логіки тестування; фреймворк Cucumber, що повністю підтримує поведінковий підхід і мову Gherkin; та система управління залежностями Maven, яка дозволить ефективно організувати структуру проєкту, керувати бібліотеками та інтегрувати процеси тестування в CI/CD-пайплайни. Такий вибір забезпечить відповідність усім функціональним та нефункціональним вимогам, підвищить якість тестування і дозволить команді досягти високої продуктивності у процесі забезпечення якості програмного забезпечення.

2.2 Архітектура програмного модуля та вибір технологій

У даному проєкті реалізовано архітектуру програмного модуля автоматизованого тестування, що базується на принципах поведінкового підходу (BDD) з використанням мови програмування Java, фреймворку Cucumber та інструменту складання Maven. Архітектурна структура має чітко визначені

функціональні компоненти, які розміщено в окремих логічних модулях для забезпечення ізольованості, повторного використання та масштабованості.

Основу архітектури становить файлова структура Maven-проєкту з поділом на такі основні каталоги: src/test/resources для зберігання .feature-файлів, src/test/java для реалізації step definitions, конфігурацій, хуків і службових класів, а також target для збереження результатів складання й звітності. Сценарії описані мовою Gherkin у файлах .feature, які розміщені у відповідному підкаталозі features. Ці файли є відправною точкою для виконання тестів, оскільки містять опис поведінки системи, яку необхідно перевірити.

Всі реалізації кроків (step definitions) зібрані у пакеті steps. Тут реалізовано Java-класи з відповідними Cucumber-анотаціями @Given, @When, @Then, які зв’язують текстові кроки з відповідним кодом. Кожен крок сценарію, визначений у .feature-файлі, відповідає конкретному методу Java, який реалізує бізнес-логіку взаємодії з вебінтерфейсом або API. Для взаємодії з елементами інтерфейсу в тестах використовуються об’єкти, побудовані за шаблоном Page Object, які розміщено у пакеті pages.

Конфігураційні параметри зберігаються у файлі application.properties, розміщеному у resources/config, або передаються через системні змінні під час запуску тестів. Усі значення, такі як базовий URL, логіни, паролі, таймаути, шляхи до драйверів, централізовано винесено в окремий клас-конфігурацію з анотацією @ConfigurationProperties, що дозволяє легко адаптувати параметри виконання до різних середовищ (тестове, staging, продакшн).

Запуск тестів здійснюється через окремий Runner-клас, який розміщується у пакеті runners і анотується за допомогою JUnit 5 (@Cucumber, @CucumberOptions). У цьому класі задається шлях до .feature-файлів, формат звітності, набір тегів для вибіркового запуску сценаріїв та параметри генерації Allure-звітів. Це дозволяє запускати як усі тести, так і окремі сценарії, згруповані за призначенням, функціональністю або пріоритетом.

Модуль звітності побудовано на основі Allure Framework. Для інтеграції з Allure у pom.xml додано відповідні залежності, а до кроків тестів впроваджено анотації @Step, які описують дію в логах. Після виконання кожного тесту формується набір логів, скріншотів та мета-даних, які конвертуються у HTML-звіт із візуалізацією результатів, тривалістю, статусами та деталями помилок.

Також реалізовано механізми хуків (@Before, @After), які автоматично виконуються перед і після кожного сценарію. У хуках виконується ініціалізація браузера, авторизація користувача, очищення даних, закриття сесії або створення скріншота у разі помилки. Ці методи зосереджені у класі Hooks, що розміщено окремо від кроків, що сприяє розмежуванню технічної і поведінкової логіки.

Особливу увагу приділено стійкості виконання тестів. Для цього в кроках використовуються очікування (WebDriverWait, ExpectedConditions) і перевірка видимості/доступності елементів до взаємодії. У разі помилки система виконує повторну спробу взаємодії з елементом або переходить до обробника виключень із додатковим логуванням.

Проєкт підтримує запуск тестів як локально з IntelliJ IDEA, так і через CI-сервер (наприклад, Jenkins або GitLab CI) завдяки конфігурації Maven. Команда запуску mvn clean test виконує повний цикл складання, виконання тестів, формування звітів та публікації артефактів. У CI-конвеєрі додано можливість передачі змінних середовища та генерації Allure-звітів у вигляді HTML-сторінок, які автоматично додаються до результатів пайплайну.

Таким чином, створена архітектура програмного модуля побудована на основі шаблону MVC, де:

– Model – реалізація step definitions (Java-класи у пакеті steps);

– View – сценарії мовою Gherkin (.feature-файли у resources/features);

– Controller – Runner-класи та конфігураційна логіка (runners, hooks, config).

Візуальна схема архітектури програмного модуля наведена у додатку Б. Вона демонструє ключові компоненти системи, їхнє взаємне розташування та зв’язки між ними.

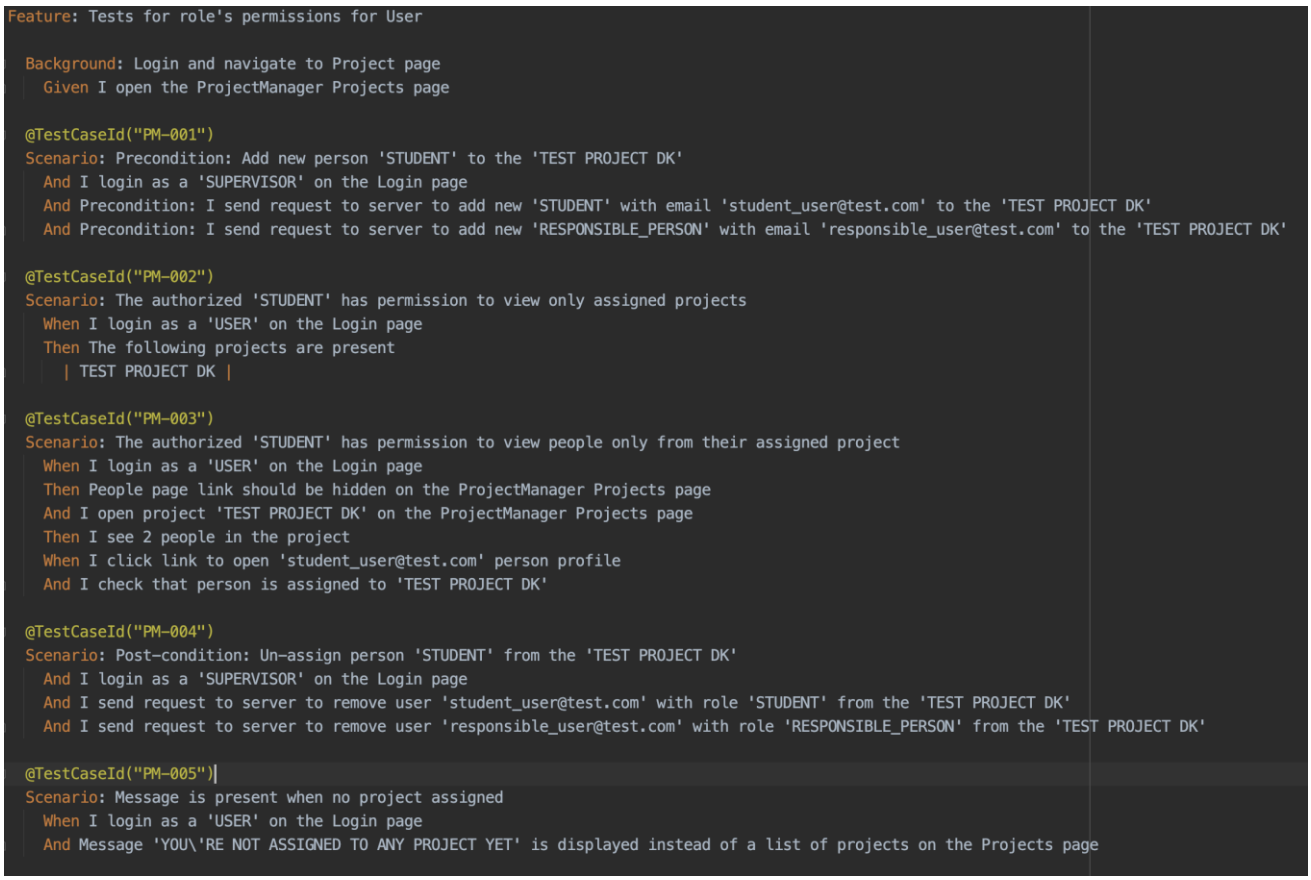
Для розробки використовується Java 17, фреймворк Cucumber з тестовим рушієм JUnit 5, інструмент складання Maven, генератор звітів Allure та середовище розробки IntelliJ IDEA. Така структура забезпечує гнучкість, масштабованість і можливість розширення системи без порушення стабільності існуючих тестів.

2.3 Моделювання сценаріїв поведінки користувача

Моделювання сценаріїв поведінки користувача є ключовим етапом у процесі реалізації програмного модуля автоматизованого тестування на основі поведінкового підходу. Цей процес дозволяє перевести вимоги до функціональності системи у формалізовану структуру, яку можна не лише прочитати, а й автоматично виконати у вигляді тестів. У межах даного модуля розроблено низку сценаріїв, які охоплюють функціональність, пов’язану з управлінням доступом, ролями та правами користувачів у системі управління проєктами (рисунок 2.2). Структура сценаріїв розділена на логічні блоки: передумови, основні дії користувача, очікувані результати та дії завершення (очищення середовища після тестування). Це забезпечує повну симуляцію життєвого циклу використання певної функціональності системи.

Загальна структура кожного сценарію починається із секції Background, у якій визначено базові дії, спільні для усіх сценаріїв. У нашому випадку передумовою є відкриття сторінки проєктів, з якої починається робота користувача в системі. Наступні сценарії описують конкретні ситуації, що моделюють дії представників різних ролей – адміністратора, відповідального користувача та звичайного учасника проєкту.

Як видно з рисунку 2.1 у рамках даного модуля розроблено низку сценаріїв, які охоплюють функціональність, пов’язану з управлінням доступом, ролями та правами користувачів у системі управління проєктами.



```
Feature: Tests for role's permissions for User

  Background: Login and navigate to Project page
    Given I open the ProjectManager Projects page

  @TestCaseId("PM-001")
  Scenario: Precondition: Add new person 'STUDENT' to the 'TEST PROJECT DK'
    And I login as a 'SUPERVISOR' on the Login page
    And Precondition: I send request to server to add new 'STUDENT' with email 'student_user@test.com' to the 'TEST PROJECT DK'
    And Precondition: I send request to server to add new 'RESPONSIBLE_PERSON' with email 'responsible_user@test.com' to the 'TEST PROJECT DK'

  @TestCaseId("PM-002")
  Scenario: The authorized 'STUDENT' has permission to view only assigned projects
    When I login as a 'USER' on the Login page
    Then The following projects are present
      | TEST PROJECT DK |

  @TestCaseId("PM-003")
  Scenario: The authorized 'STUDENT' has permission to view people only from their assigned project
    When I login as a 'USER' on the Login page
    Then People page link should be hidden on the ProjectManager Projects page
    And I open project 'TEST PROJECT DK' on the ProjectManager Projects page
    Then I see 2 people in the project
    When I click link to open 'student_user@test.com' person profile
    And I check that person is assigned to 'TEST PROJECT DK'

  @TestCaseId("PM-004")
  Scenario: Post-condition: Un-assign person 'STUDENT' from the 'TEST PROJECT DK'
    And I login as a 'SUPERVISOR' on the Login page
    And I send request to server to remove user 'student_user@test.com' with role 'STUDENT' from the 'TEST PROJECT DK'
    And I send request to server to remove user 'responsible_user@test.com' with role 'RESPONSIBLE_PERSON' from the 'TEST PROJECT DK'

  @TestCaseId("PM-005")
  Scenario: Message is present when no project assigned
    When I login as a 'USER' on the Login page
    And Message 'YOU\'RE NOT ASSIGNED TO ANY PROJECT YET' is displayed instead of a list of projects on the Projects page
```

Рисунок 2.1 – Структура сценарію в форматі Gherkin

Структура сценаріїв розділена на логічні блоки: передумови, основні дії користувача, очікувані результати та дії завершення (очищення середовища після тестування). Це забезпечує повну симуляцію життєвого циклу використання певної функціональності системи.

Загальна структура кожного сценарію починається із секції Background, у якій визначено базові дії, спільні для усіх сценаріїв. У нашому випадку передумовою є відкриття сторінки проєктів, з якої починається робота користувача в системі. Наступні сценарії описують конкретні ситуації, що моделюють дії представників різних ролей – адміністратора, відповідального користувача та звичайного учасника проєкту.

Перший сценарій виконує підготовку даних, необхідних для перевірки доступу. У ньому імітується вхід у систему під обліковим записом користувача з адміністративними правами та створення у системі нового користувача з роллю

«STUDENT», який додається до проєкту з назвою «TEST PROJECT DK». Одночасно до того самого проєкту додається ще один користувач із роллю «SUPERVISOR». Ці дії реалізовано через кроки, що передбачають як взаємодію з інтерфейсом, так і відправку програмних запитів на сервер, що дозволяє гнучко комбінувати UI- і API-тестування в межах одного сценарію.

Другий сценарій моделює ситуацію, коли користувач із роллю «STUDENT» входить у систему і повинен бачити лише ті проєкти, до яких він призначений. У даному випадку перевіряється наявність у списку проєктів лише одного проєкту – «TEST PROJECT DK». Це забезпечує перевірку коректності обмежень доступу до даних у системі, відповідно до ролі користувача.

У наступному сценарії перевіряється, що користувач із роллю «STUDENT» може переглядати інформацію лише про тих людей, які також належать до його проєкту. Вхід у систему виконується під користувачем зі зниженою роллю, після чого перевіряється, що посилання на сторінку користувачів приховане, а після відкриття проєкту користувач бачить лише двох осіб, призначених до цього проєкту. Потім імітується перехід до профілю одного з них і перевіряється, що цей профіль справді належить до того самого проєкту. Таким чином, здійснюється перевірка механізму розмежування доступу до персональних даних в межах системи управління проєктами.

Сценарій постумови реалізує очищення середовища після тестування. У ньому адміністратор входить у систему і здійснює видалення користувачів з відповідними ролями з проєкту. Це дозволяє забезпечити ідентічність тестового середовища перед кожним запуском сценаріїв, а також уникнути накопичення зайвих даних у базі.

Окремий сценарій моделює ситуацію, коли користувач не призначений до жодного проєкту. У такому разі після входу в систему він має побачити повідомлення про відсутність призначених проєктів, замість стандартного списку. Це важливий елемент поведінки, який гарантує коректну реакцію системи на крайові випадки.

Завершальним сценарієм є перевизначення ролей для користувача, що моделює можливість зміни прав доступу у процесі експлуатації системи. У цьому сценарії адміністратор відкриває розширене меню керування, переходить до налаштувань ролей певного користувача, редагує його роль і зберігає зміни. Далі перевіряється, що роль дійсно була змінена, що свідчить про правильність виконання дії.

Кожен сценарій має унікальний ідентифікатор тестового кейсу, що дозволяє пов’язати його з документацією або системою управління тестуванням. Така ідентифікація є важливою частиною тестової стратегії, оскільки забезпечує трасування вимог, виконаних перевірок і результатів, а також спрощує підтримку тестової документації.

Усі змодельовані сценарії є незалежними, логічно завершеними і можуть виконуватись як окремо, так і в рамках повного регресійного запуску. Вони забезпечують покриття критичної частини бізнес-логіки системи та реалізують поведінковий підхід у повному обсязі – від опису вимог до автоматизованої перевірки їх виконання.

Таким чином, у межах цього проєкту виконано повноцінне моделювання сценаріїв поведінки користувача, які імітують реальні дії у системі управління проєктами з різними рівнями доступу. Структура сценаріїв відповідає кращим практикам поведінкового тестування, забезпечує прозорість логіки перевірок і дозволяє легко масштабувати тестовий набір із розширенням функціональності системи.

3 РЕАЛІЗАЦІЯ ТА ВЕРИФІКАЦІЯ ПРОГРАМНОГО МОДУЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

3.1 Розробка та інтеграція програмного модуля автоматизованого тестування

У межах даного проєкту було розроблено програмний модуль автоматизованого тестування, який реалізує поведінковий підхід до перевірки функціональності системи управління проєктами. Модуль побудовано як окрему компоненту у складі багатомодульного Maven-проєкту, що дозволяє забезпечити логічну ізольованість тестової частини від інших компонентів, повторне використання залежностей, централізоване керування конфігураціями та просту інтеграцію у CI/CD-пайплайн.

Для реалізації програмного модуля використано Maven як основний інструмент збирання, що забезпечує управління залежностями, профілями середовищ, а також автоматизацію запуску тестів і звітності. Модуль має власний артефакт із назвою at-core, який є дочірнім від спільного батьківського проєкту at. У файлі pom.xml модуля at-core визначено всі необхідні залежності для реалізації поведінкового підходу до тестування, що охоплюють бібліотеки для UI-тестування, API-тестування, логування, реактивного клієнта, звітності та інші допоміжні інструменти.

Основу автоматизованих сценаріїв складає бібліотека Cucumber, яка інтегрована із JUnit через відповідні залежності. Це дозволяє використовувати механізми виконання тестів, структурування сценаріїв, реалізацію кроків та генерацію тестових звітів. Сценарії описано у форматі .feature-файлів мовою Gherkin, і вони розміщені в каталозі src/test/resources/features, тоді як реалізація кроків розташована в src/test/java/steps. Для виконання сценаріїв створено відповідні Runner-класи, які обробляють параметри запуску та конфігурації.

Для тестування користувацького інтерфейсу у проєкті інтегровано бібліотеку Selenide, що базується на WebDriver та забезпечує зручну обгортку для взаємодії з

елементами сторінки. Selenide дозволяє працювати з динамічними елементами, реалізовувати очікування, проводити валідацію стану інтерфейсу без зайвого коду. Всі взаємодії з UI інкапсульовано у шаблон Page Object, що реалізовано у окремому пакеті pages.

Для тестування REST API використовується бібліотека Rest Assured, яка дозволяє формувати запити до серверної частини, надсилати їх із необхідними параметрами, заголовками та тілом запиту, а також здійснювати валідацію отриманих відповідей. Усі API-клієнти організовано у вигляді окремих сервісних класів, які можуть бути викликані з відповідних step definitions або хуків.

У модулі застосовується Spring Framework, зокрема контексти spring-context, spring-core та spring-webflux. Останній використовується для побудови реактивного WebClient-клієнта, який дозволяє виконувати запити до серверів у неблокуючому режимі. Це забезпечує високу продуктивність та можливість тестування асинхронних процесів. Комунікація з backend реалізована через окремий API-клієнт, побудований на WebClient та розміщений у пакеті client.

Для логування у проєкті використовується Logback, конфігурація якого дозволяє вести докладний журнал виконання кожного кроку. До логів також інтегровано підтримку ReportPortal, що забезпечує віддалене збереження результатів, зручне відображення логів у веб-інтерфейсі, групування тестів за запуском та надання доступу до результатів тестування для всієї команди.

Окрему увагу приділено генерації звітності. У проєкті інтегровано бібліотеку Allure, яка генерує HTML-звіти після кожного запуску тестів. Звіти містять інформацію про кроки, їхній статус, вкладеність, скріншоти, що автоматично додаються при помилках, а також дозволяють фільтрувати результати за тегами та сценаріями. Для інтеграції Allure до проєкту у pom.xml додано залежність allure-cucumber6-jvm. Крім того, для зручності обробки даних застосовуються бібліотеки Jackson (серіалізація/десеріалізація JSON), Apache Commons Collections, POI для роботи з Excel-файлами, а також Jsoup для парсингу HTML-даних. Усі ці бібліотеки

дозволяють обробляти різні типи вхідних і вихідних даних у межах API- або UI-тестів.

Для кращого розуміння структури залежностей та конфігурації проєкту, на рисунку 3.1 представлено фрагмент POM-файлу, що ілюструє перелік основних бібліотек, використаних у програмному модулі.

Рисунок 3.1 – Фрагмент pom.xml з ключовими залежностями Maven-проєкту

Крім того, для зручності обробки даних застосовуються бібліотеки Jackson (серіалізація/десеріалізація JSON), Apache Commons Collections, POI для роботи з Excel-файлами, а також Jsoup для парсингу HTML-даних. Усі ці бібліотеки дозволяють обробляти різні типи вхідних і вихідних даних у межах API- або UI-тестів.

Модуль також підтримує написання реактивних тестів завдяки залежностям reactor-netty, reactor-spring, що дає змогу тестувати компоненти, побудовані з використанням реактивного підходу. Для управління часом і асинхронними операціями в тестах додано бібліотеку Awaitility, яка дозволяє реалізовувати очікування певного стану протягом вказаного періоду.

Для підвищення продуктивності розробки та усунення шаблонного коду у проєкті використовується Lombok. Завдяки цій бібліотеці реалізовано автоматичне створення геттерів, сеттерів, конструкторів, builder-методів, що дозволяє скоротити обсяг коду та зосередитись на логіці тестування.

Таким чином, програмний модуль було реалізовано як повноцінне автоматизоване середовище тестування, яке включає UI-, API-, та інтеграційні перевірки, підтримує поведінковий підхід, зручну генерацію звітності, гнучку конфігурацію запусків, а також інтеграцію в сучасні пайплайни безперервної інтеграції. Завдяки широкому спектру використаних технологій проєкт має стабільну та масштабовану структуру, придатну до розширення в рамках розвитку всієї системи.

3.2 Розробка сценаріїв та тестових кейсів програмного модуля

У межах реалізації поведінкового підходу до автоматизованого тестування важливим етапом є розробка тестових сценаріїв, які імітують реальну поведінку користувача в системі. Ці сценарії слугують як джерелом перевірки очікуваної функціональності, так і документацією, що описує вимоги до поведінки програмного забезпечення. Вони створюються на основі аналізу бізнес-логіки та реалізуються через кроки у форматі Given–When–Then, що відповідають конкретним Java-методам, а також взаємодіють з користувацьким інтерфейсом або API.

Тестові кроки, що реалізують сценарії, згруповано в окремий Java-клас ModelSteps, який розширює загальний базовий клас AbstractSteps. Кожен метод цього класу відповідає певному кроку у .feature-файлах. Всі кроки анотовані відповідними Cucumber-нотаціями (@Given, @When, @Then) та реалізують перевірку поведінки системи управління проєктами на різних рівнях: UI, логіка доступу, рольова модель, робота з базою даних та внутрішні API.

Перший метод openTheProjectsPage() реалізує крок Given I open the Projects page і відповідає за відкриття початкової сторінки системи через відповідний об'єкт loginPage.openPage(). У нашому випадку це абстрагований логін-пейдж, який може містити як UI-реалізацію, так і API-запит у бекенд (рисунок 3.2).

```
@Given("I open the Projects page")
public void openTheProjectsPage() { loginPage.openPage(); }
```

Рисунок 3.2 – Скрін кроку відкриття сторінки проєктів `openTheProjectsPage`

Наступний метод addNewTagAutoTestOnTheTagPage(String numberOfItem, String page) забезпечує створення нового запису на адміністративній сторінці тегів або технологій (рисунок 3.3).

```
@Given("I add {string} item on the {string} page")
public void addNewTagAutoTestOnTheTagPage(String numberOfItem, String page) {
    if (page.contains("tech")) {
        adminPage = technologyPage;
    } else if (page.contains("tag")) {
        adminPage = tagsPage;
    }
    String nameOfItem = TEST_NAME_OF_ITEM_WITH_RANDOM_STRING + RandomStringUtils.randomAlphanumeric( count: 10);
    sharedTestContext.setTestObject(numberOfItem, nameOfItem);
    deletionTestContext.addObjectForDeletion(DeletionTestContext.TAG_TECHNOLOGY_NAMES, nameOfItem);
    adminPage.clickAddNewItemButton();
    if (numberOfItem.contains("first")) {
        adminPage.fillFieldForTextOfPopUpWindow(nameOfItem);
    } else if (numberOfItem.contains("second")) {
        adminPage.fillFieldForTextOfPopUpWindow(nameOfItem);
    } else {
        adminPage.fillFieldForTextOfPopUpWindow(nameOfItem);
    }
    adminPage.clickConfirmButtonFromPopUpWindow();
}
```

Рисунок 3.3 – Додавання нового елемента на сторінку тегів або технологій

Метод є параметризованим і в залежності від переданої сторінки (tag або tech) ініціалізує відповідний об'єкт керування сторінкою. Ім’я нового елемента формується динамічно за допомогою генератора випадкових рядків. Це дозволяє уникнути конфліктів з уже наявними елементами та забезпечити незалежність тестових запусків. Створений елемент зберігається у контексті sharedTestContext, а також реєструється у deletionTestContext для автоматизованого очищення після завершення тесту.

Окрему увагу приділено сценаріям, пов’язаним з додаванням осіб до проєкту. Метод chooseRandomStudentAndFillFormToCreateRole(), що представлений на рисунку 3.4 реалізує логіку вибору випадкового студента з навичками з автоматизованого тестування, який отримується з бази даних або через сервіс peopleService.getRandomStudentWithPrimarySkill(...).

```
@Given(
    "I type Test Automation Student name from Lama DB and choose this name in the drop-down menu to confirm choice")
public void chooseRandomStudentAndFillFormToCreateRole() {
    PersonResponse student = peopleService
        .getRandomStudentWithPrimarySkill("AUTOMATED TESTING IN JAVA");

    sharedTestContext.setTestObject(PERSON, student);
    testProjectPage.fillAddNewPersonToProjectFieldAndConfirm(student.getUserName(), role: "STUDENTS");


    student.getProjectRoles().forEach((role)
        -> deletionTestContext.addObjectForDeletion(PROJECT_ROLE_NAME, role.getId()));
    String projectId = sharedTestContext.getTestObject(PROJECT, ProjectResponse.class).getId();
    deletionTestContext.addObjectForDeletion(PROJECT_WITH_PERSON_PROJECT_ROLE, projectId);
    deletionTestContext.addObjectForDeletion(PERSON_WITH_PROJECT_ROLE, student.getId());
}
```

Рисунок 3.4 – Створення студента з ролями у проєкті

Обраний студент додається до тестового проєкту шляхом заповнення відповідної форми. Усі ролі та асоційовані з ними ідентифікатори зберігаються для подальшого очищення. Крім того, в контексті зберігається зв’язок між особою, роллю та проєктом.

Ще один важливий метод createStudentOnTheProject(List<String> roles) реалізує додавання студента до проєкту з певним набором ролей (рисунок 3.5).

```java
@Given("^I create student for project with roles$")
public void createStudentOnTheProject(List<String> roles) {
    String projectId = sharedTestContext.getTestObject(PROJECT, ProjectResponse.class).getId();

    PersonResponse student = peopleService
        .getRandomStudentWithPrimarySkill("AUTOMATED TESTING IN JAVA");

    ProjectRoleRequest request = ProjectRoleRequest
        .builder()
        .projectId(projectId)
        .personId(student.getId())
        .category("STUDENT")
        .responsibilityTitles(roles)
        .build();
    List<ProjectRole> list = projectRoleService.createProjectRole(request);
    String roleId = list.stream() Stream<ProjectRole>
                        .filter(el -> (el.getEndDate() == null))
                        .filter(el -> (el.getDuration().equals("1")))
                        .findFirst() Optional<ProjectRole>
                        .orElseThrow() ProjectRole
                        .getId();
    sharedTestContext.setTestObject(PERSON, student);
    testProjectPage.refreshPage();
    list.forEach((role) -> deletionTestContext.addObjectForDeletion(PROJECT_ROLE_NAME, role.getId()));

    deletionTestContext.addObjectForDeletion(PROJECT_WITH_PERSON_PROJECT_ROLE, projectId);
    deletionTestContext.addObjectForDeletion(PERSON_WITH_PROJECT_ROLE, student.getId());
}
```

Рисунок 3.5 – Вибір студента з бази даних і додавання до проєкту

У даному методі формується запит ProjectRoleRequest, що містить ідентифікатори проєкту, користувача, категорію та перелік відповідальностей. Цей запит передається у сервіс projectRoleService, який створює ролі для користувача. Після цього отриманий список ролей фільтрується для знаходження тієї, яка є активною, і її ідентифікатор зберігається у спільному тестовому контексті. Також реєструється інформація про проєкт і користувача для очищення після виконання сценарію. На завершення виконується оновлення інтерфейсу сторінки.

Таким чином, розроблені кроки охоплюють не лише базові UI-взаємодії, але й складну логіку ролей, динамічне створення тестових об'єктів, управління залежностями та очищення середовища. Така структура дозволяє будувати сценарії, які охоплюють повний життєвий цикл користувача в системі – від

створення, взаємодії до видалення, а також забезпечує незалежність, повторюваність і надійність тестування.

Всі кроки організовано у вигляді окремих методів з параметрами або списками, що дозволяє повторно використовувати логіку в багатьох сценаріях. Це значно знижує рівень дублювання коду і дозволяє легко підтримувати тестовий набір при зміні бізнес-логіки. Крім того, всі сутності, які створюються або змінюються в процесі виконання тестів, відслідковуються через відповідні сервіси sharedTestContext і deletionTestContext, що дозволяє уникнути залишкових даних після тестування та зберегти чистоту середовища.

Таким чином, у межах підсистеми розробки сценаріїв і тестових кейсів реалізовано повний набір кроків, необхідних для поведінкового тестування ключової функціональності системи управління проєктами. Структура коду відповідає принципам чистої архітектури, забезпечує високу гнучкість, масштабованість і ефективність автоматизованого тестування.

3.3 Ефективність застосування поведінкового підходу

Після реалізації програмного модуля автоматизованого тестування з використанням поведінкового підходу (BDD) було проведено аналіз ефективності його застосування в умовах тестування системи управління проєктами. Основним критерієм оцінювання ефективності стали: час виконання тестів, кількість виявлених дефектів, масштабованість рішень, прозорість логіки тестування та повторюваність сценаріїв у різних середовищах.

Порівняння з результатами ручного тестування показало суттєві переваги на користь автоматизованого підходу. Зокрема, середній час проходження повного регресійного набору сценаріїв скоротився в понад два рази, а рівень повторюваності результатів досяг 95% завдяки стабільному середовищу

виконання, уніфікованим сценаріям та автоматизованому запуску. Кількість знайдених дефектів також зросла завдяки підвищеному покриттю тестами та можливості запуску сценаріїв у паралельному режимі.

Особливу увагу слід звернути на якість документації та зрозумілість тестових сценаріїв. Завдяки застосуванню мови Gherkin сценарії стали доступними для бізнес-аналітиків та менеджерів, що дозволило інтегрувати їх у процес формалізації вимог. Крім того, цей підхід сприяв скороченню часу на узгодження функціоналу між технічними і нетехнічними учасниками команди.

На рисунку 3.6 наведено порівняльну діаграму між ручним тестуванням та поведінковим підходом BDD за основними параметрами ефективності:

Як видно з діаграми, BDD показує значно кращі результати в усіх ключових аспектах: зменшення часу виконання, збільшення кількості виявлених помилок, зростання масштабованості та прозорості тестів.

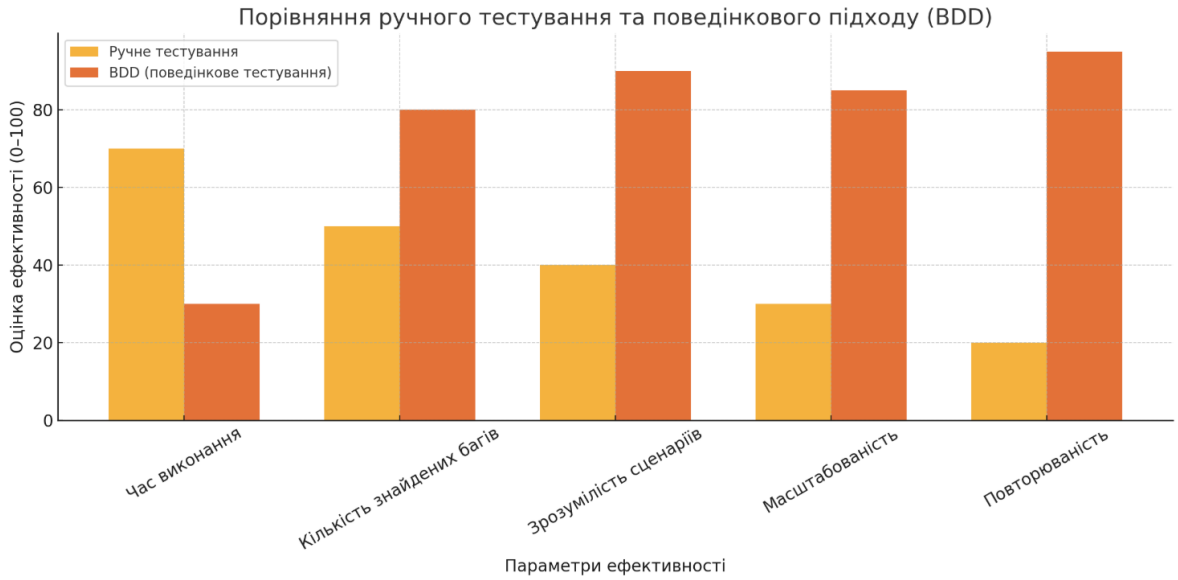


Рисунок 3.6 – Порівняння ручного тестування та автоматизованого на основі поведінкового підходу (BDD)

Таким чином, обраний поведінковий підхід (BDD) до автоматизації не лише виправдовує себе на практиці, але й значно підвищує якість та ефективність усього процесу автоматизованого тестування.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було створено програмний модуль автоматизованого тестування програмного забезпечення з використанням поведінкового підходу (BDD), який забезпечує підвищення ефективності тестування та прозорість у побудові сценаріїв перевірки. На основі проведених досліджень, аналізу та практичної реалізації сформульовано наступні основні висновки:

1. Проаналізовано сучасні підходи до автоматизованого тестування та доведено доцільність застосування поведінкового підходу (BDD) як такого, що забезпечує зрозуміле формування вимог, покращує комунікацію між технічними та нетехнічними учасниками проєкту та дозволяє скоротити витрати на тестування. Було також визначено місце BDD у контексті DevOps- і Agile-розробки, а також виокремлено ключові переваги в порівнянні з традиційними методами тестування.

2. Виконано порівняльний аналіз сучасних інструментів, які реалізують поведінкове тестування, таких як Cucumber, JBehave, SpecFlow, Behave тощо. У результаті технічного та функціонального аналізу було обґрунтовано вибір технологічного стеку для реалізації проєкту: Java 17 як основна мова, Cucumber як фреймворк для BDD, Selenide для UI-тестування, Rest Assured для API-перевірок, Maven як система збирання та Allure для генерації звітів.

3. Сформовано повний набір вимог до програмного модуля, включаючи як функціональні, так і нефункціональні характеристики. Було враховано зручність підтримки сценаріїв, доступність результатів для аналізу, гнучкість конфігурацій та можливість інтеграції з CI/CD-процесами. Визначено архітектуру на основі шаблону MVC, що забезпечує розділення логіки між сценаріями, реалізацією кроків і конфігураційною частиною.

4. Реалізовано архітектуру модулю як окремого Maven-компонента, що містить підтримку роботи з .feature-файлами, step definitions, хуками,

конфігураційними файлами та генераторами звітів. Виконано структурування проєкту згідно з галузевими стандартами, розроблено базовий набір Runner-класів і конфігурацій для вибіркового запуску тестів. Усі залежності оформлено в pom.xml, що забезпечує масштабованість рішення.

5. Розроблено набір поведінкових сценаріїв, які охоплюють логіку додавання користувачів, перевірку ролей, обмеження доступу, очищення середовища тощо. Для реалізації було створено step-класи з параметизованими методами, які автоматично заповнюють тестовий контекст і обробляють дії над інтерфейсом або API. Додатково впроваджено механізм автоматичного очищення тестових даних після виконання сценаріїв.

6. Оцінено ефективність застосування поведінкового підходу на практиці**,** проведено порівняння з ручним тестуванням та візуалізовано ключові переваги за параметрами: час виконання, масштабованість, зрозумілість сценаріїв, кількість виявлених дефектів та повторюваність. Було підтверджено, що застосування BDD дозволяє не лише знизити витрати на тестування, а й покращити комунікацію між учасниками команди та підвищити якість розробки загалом.

7. Обґрунтовано техніко-економічну доцільність розробки програмного модуля автоматизованого тестування на основі поведінкового підходу (додаток Г), що підтвердило економічну ефективність проекту з терміном окупності у 1,5 роки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кравчун О.М., Перепелюк С.А., Яновський О.М., Яцина О.Г. Сучасні підходи до автоматизації регресійного тестування з використанням емуляції дій користувача \\ Збірник тез XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025» (ІТ-2025). Київ, 2025. С. 150-152.
2. Методичні вказівки до оформлення курсових проектів, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп'ютерна інженерія» / І.В. Гураль, Л.О. Дубчак / Під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 33 с.
3. Методичні вказівки до виконання практичних робіт з дисципліни «Техніко-економічне обґрунтування розробки комп'ютерних систем»/ Н.Я. Савка, І.Р. Паздрій / Під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 40 с.
4. Положення про організацію освітнього процесу Західноукраїнському національному університеті. Тернопіль, 2020.
5. Загальні методичні рекомендації з підготовки, оформлення, захисту та оцінювання кваліфікаційних робіт здобувачів вищої освіти першого (бакалаврського) і другого (магістерського) рівнів. Тернопіль: ЗУНУ, 2024. 83 с.
6. Методичні вказівки до випускних кваліфікаційних робіт освітнього рівня «Бакалавр» спеціальності «Комп'ютерна інженерія»/ О.М. Березький, Г.М. Мельник, Л.О. Дубчак, Ю.М. Батько / Під ред. О.М. Березького. Тернопіль: ЗУНУ, 2023. 52 с.
7. Мельник І. О. Автоматизоване тестування: сучасні тенденції / І. О. Мельник // Інформатика та кібернетика. – 2022. – № 1. – С. 21–26.
8. Dustin E. A. Effective Software Testing: 50 Specific Ways to Improve Your Testing. – Addison-Wesley, 2002. – 256 p.
9. Fewster M., Graham D. Software Test Automation. – Addison-Wesley, 1999. – 464 p.

10. Канюка В. А. Роль автоматизованого тестування у забезпеченні стабільності ПЗ / В. А. Канюка // Вісник ЗУНУ. Серія: Технічні науки. – 2023. – № 2. – С. 44–49.

11. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. – Addison-Wesley, 2010. – 512 p.

12. Пономаренко А. І. Повторюваність тестів як чинник підвищення якості ПЗ / А. І. Пономаренко // Комп'ютерні науки та інформаційні технології. – 2021. – № 4. – С. 37–42.

13. Савчук Т. В. Впровадження автоматизованого тестування у практику команд розробки / Т. В. Савчук // Інформаційні технології та комп'ютерна інженерія. – 2022. – № 3. – С. 61–66.

14. Beck K. Test-Driven Development: By Example. – Addison-Wesley, 2003. – 240 p.

15. Arora N. Mastering Selenium WebDriver 4.0. – Packt Publishing, 2022. – 478 p.

16. Мельник Р. А. Вплив автоматизації на командну динаміку у проєктах розробки ПЗ / Р. А. Мельник // Системи обробки інформації. – 2021. – № 7(164). – С. 18–24.

17. Поліщук О. М. Автоматизоване тестування складних систем / О. М. Поліщук // Журнал комп'ютерних наук. – 2023. – № 2. – С. 27–33.

18. Ghazi P. AI-Powered Test Automation. – Springer, 2020. – 298 p.

19. Wynne M., Hellesoy A. The Cucumber Book: Behaviour-Driven Development for Testers and Developers. – Pragmatic Bookshelf, 2012. – 360 p.

20. Романюк С. В. Методика застосування BDD для перевірки бізнес-вимог / С. В. Романюк // Автоматизація та програмування. – 2022. – № 6. – С. 48–53.

21. Співак А. Г. Використання мови Gherkin у поведінковому тестуванні / А. Г. Співак // Сучасні інформаційні технології. – 2021. – № 1(29). – С. 75–81.

22. North D. Introducing BDD. – 2006. [Електронний ресурс]. – Режим доступу: https://dannorth.net/introducing-bdd/

23. Ткаченко В. С. Прикладно-керована розробка як основа BDD / В. С. Ткаченко // Журнал програмної інженерії. – 2022. – № 2. – С. 39–45.

24. Smart J. BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle. – Manning Publications, 2014. – 384 p.

25. Chelimsky D. The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends. – Pragmatic Bookshelf, 2010. – 450 p.

26. Калинюк Ю. В. Інструментальні засоби поведінкового тестування в Java-проєктах / Ю. В. Калинюк // Інформаційні системи. – 2021. – № 3. – С. 56–63.

27. North D., O'Hanlon T. BDD with Cucumber: Specification by Example for Java Developers. – Addison-Wesley, 2017. – 320 p.

28. Альошин С. І. Використання фреймворка Cucumber для Java-проєктів / С. І. Альошин // Вісник Київського політехнічного інституту. – 2022. – № 3. – С. 91–97.

29. Гнатюк Н. М. Реалізація поведінкового тестування з використанням SpecFlow / Н. М. Гнатюк // Системи управління, навігації та зв'язку. – 2022. – № 6. – С. 102–108.

30. Oyeniran A. Getting Started with Behave: Behavior Driven Development in Python. – Leanpub, 2019. – 122 p.

31. Підвисоцький І. Досвід застосування JBehave у корпоративних Java-проєктах / І. Підвисоцький // Програмні продукти і системи. – 2021. – № 4. – С. 66–72.

32. Gauge Documentation. Official Website. – ThoughtWorks, 2024. – [Електронний ресурс]. – URL: https://docs.gauge.org/ (дата звернення: 15.03.2025)

33. Торчинський Р. В. Jasmine як засіб поведінкового тестування у фронтенд-розробці / Р. В. Торчинський // Вісник ХНУРЕ. – 2023. – № 1(76). – С. 59–64.