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

ЯЦИНА Олег Григорович

Програмний модуль генерації тестових сценаріїв на основі функціональних вимог / Software module for test scenarios generation based on functional requirements

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

Кваліфікаційна робота

Виконав: студент групи КІ-41
Яцина Олег Григорович

Науковий керівник
к.т.н. Гураль І.В.

ТЕРНОПІЛЬ - 2025

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль генерації тестових сценаріїв на основі функціональних вимог» зі спеціальності 123 «Комп'ютерна інженерія» освітнього ступеня «бакалавр» містить 64 сторінки пояснюючої записки, 7 рисунків, 2 таблиці та 4 додатки. Обсяг графічного матеріалу 2 аркуші формату А3.

Мета роботи полягає у розробці програмного модуля, який реалізує автоматичну генерацію тестових сценаріїв на основі формалізованих функціональних вимог до програмного забезпечення з подальшим виконанням у середовищі автоматизованого тестування.

Методи дослідження включають аналіз літературних джерел з тематики тест-дизайну, вивчення сучасних підходів до генерації сценаріїв, побудову моделі функціональних вимог, проєктування програмної архітектури, розробку алгоритмів генерації тестів, експериментальне тестування та оцінювання результатів.

У роботі проаналізовано існуючі методи формалізації вимог, запропоновано структуру вхідного формату, побудовано алгоритм перетворення вимог у тестові сценарії, реалізовано інтеграцію з системами зберігання та візуалізації результатів тестування.

Результатом є створений програмний модуль, який забезпечує швидке формування узгоджених із вимогами тест-кейсів, знижує трудовитрати на написання тестів вручну та підвищує якість перевірки функціональності програмного забезпечення.

Ключові слова: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ГЕНЕРАЦІЯ СЦЕНАРІЇВ, ФУНКЦІОНАЛЬНІ ВИМОГИ, ТЕСТ-ДИЗАЙН, JAVA, API, CUCUMBER.

ANNOTATION

The qualification paper titled «Software module for test scenarios generation based on functional requirements», specialty 123 «Computer Engineering», Bachelor's degree, contains 64 pages of explanatory note, 7 figures, 2 tables and 4 appendices. The graphical material volume is 2 A3-format sheets.

The aim of the research is to develop a software module that enables automatic generation of test scenarios based on formalized functional requirements of software, with further execution in an automated testing environment.

The research methods include analysis of literature on test design, study of current approaches to scenario generation, modeling of functional requirements, software architecture design, development of test generation algorithms, experimental testing, and evaluation of results.

The work analyzes existing methods of requirements formalization, proposes an input format structure, develops an algorithm for transforming requirements into test scenarios, and implements integration with storage and test result visualization systems.

The result is a software module that enables rapid formation of requirement-compliant test cases, reduces the manual effort required for test development, and enhances the quality of functional verification of software.

Keywords: AUTOMATED TESTING, SCENARIO GENERATION, FUNCTIONAL REQUIREMENTS, TEST DESIGN, TEST CASES, JAVA, API, CUCUMBER.

ЗМІСТ

ВСТУП

На сьогоднішній день стрімкий розвиток інформаційних технологій зумовлює підвищення вимог до якості програмного забезпечення. Забезпечення належного рівня якості стає критичним чинником у процесі розробки програмних систем, незалежно від їх масштабу та галузі застосування. Одним із ключових інструментів контролю якості є процес тестування, що дозволяє виявляти помилки на ранніх етапах життєвого циклу програмного продукту. У зв'язку з цим автоматизоване тестування поступово витісняє ручне, оскільки забезпечує вищу швидкість виконання перевірок, повторюваність результатів і зниження впливу людського чинника.

Однією з актуальних тенденцій сучасного тестування є автоматична генерація тестових сценаріїв. Особливої ваги вона набуває у випадках, коли тестування повинне здійснюватись на основі великого обсягу формалізованих функціональних вимог. Саме функціональні вимоги, згідно з принципами моделі V-подібного життєвого циклу програмного забезпечення, є ключовим джерелом для побудови тестових сценаріїв. Водночас ручна розробка таких сценаріїв часто виявляється трудомісткою, повільною та недостатньо гнучкою до змін у системі. Це обґрунтовує необхідність створення програмних інструментів, які дозволяють автоматизувати перетворення вимог у виконувані тести.

Актуальність теми дослідження зумовлена потребою у розробці універсального підходу до побудови модулів автоматичної генерації тестів, який забезпечуватиме узгодженість між вимогами, логікою реалізації та перевіркою очікуваної поведінки системи.

Об'єктом дослідження є процес автоматизованого тестування програмного забезпечення. Предметом дослідження є алгоритми та методи генерації тестових сценаріїв на основі функціональних вимог.

Метою бакалаврської роботи є розробка програмного модуля, який реалізує автоматичну генерацію тестових сценаріїв на основі формалізованих функціональних вимог до програмного забезпечення з подальшим виконанням у середовищі автоматизованого тестування.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– проаналізувати процес автоматизованого тестування та особливості використання функціональних вимог для побудови тестів;

– дослідити існуючі методи автоматичної генерації тестів та обґрунтувати вибір найефективнішого з них;

– сформувати вимоги до архітектури програмного модуля та обрати відповідні технології;

– реалізувати програмний модуль генерації сценаріїв і забезпечити його інтеграцію із середовищем тестування;

– провести запуск згенерованих тестів і проаналізувати результати з використанням інструменту звітності Allure;

– здійснити верифікацію результатів і оцінити ефективність реалізованого підходу.

Результати дослідження та практичної реалізації були частково оприлюднені у вигляді тез на XII Всеукраїнській науково-практичній конференції молодих учених «Інформаційні технології – 2025» [1]. Світлокопії публікації наведено у додатку А.

Структурно бакалаврська робота складається з трьох розділів [2-6]. У першому розділі наведено загальні характеристики процесу автоматизованої генерації тестів та функціональних вимог. У другому розділі описано архітектуру, принцип роботи та реалізацію програмного модуля. Третій розділ присвячено верифікації тестів, аналізу звітності та оцінці ефективності впровадженого рішення.

1 ОСНОВНІ ХАРАКТЕРИСТИКА ПРОЦЕСУ ГЕНЕРАЦІЇ ТЕСТОВИХ СЦЕНАРІЇВ

1.1 Особливості автоматизованого тестування програмного забезпечення

Автоматизоване тестування програмного забезпечення на сучасному етапі розвитку інформаційних технологій є одним з ключових етапів забезпечення якості (Quality Assurance, QA) [7]. Воно виконує роль інструменту для виявлення дефектів, підтвердження відповідності системи вимогам, а також гарантування стабільності та надійності роботи програмних продуктів у динамічному середовищі розробки. Порівняно з ручним тестуванням, автоматизація дозволяє досягти значної економії часу, забезпечити повторюваність результатів, а також зменшити вплив людського фактору [8].

Автоматизоване тестування передбачає створення скриптів або програмних модулів, які автоматично виконують тестові сценарії, перевіряючи поведінку програмного забезпечення згідно з очікуваними результатами [9]. Важливою передумовою ефективного автоматизованого тестування є наявність чітко визначених функціональних вимог, які слугують базою для побудови тестів. Адже саме ці вимоги відображають очікувану поведінку системи, що підлягає перевірці.

У процесі автоматизації тестування виділяють кілька основних напрямів: модульне (unit) тестування, інтеграційне тестування, системне тестування та приймальне (acceptance) тестування. Кожен з цих типів тестів має свої особливості, цілі та рівень деталізації. Зокрема, модульне тестування зосереджене на перевірці окремих функцій або класів програмного коду, а системне – на тестуванні взаємодії всіх компонентів програми як єдиного цілого [10].

Однією з важливих характеристик автоматизованого тестування є його орієнтованість на повторювані перевірки. У середовищах з частими релізами або методологією CI/CD (безперервна інтеграція та розгортання) автоматизовані тести

дозволяють оперативно перевіряти стабільність системи після кожної зміни. У таких випадках ручне тестування стає економічно невиправданим і технічно складним.

Проте ефективність автоматизованого тестування значною мірою залежить від якості побудови тестових сценаріїв. Основна проблема полягає в тому, що багато тестів створюються вручну, і цей процес вимагає значних витрат часу та ресурсів. При цьому існує ризик помилок у логіці тестів або неповного покриття функціональних вимог. Для вирішення цієї проблеми все більшої популярності набуває підхід до генерації тестових сценаріїв на основі вимог, за якого тести створюються автоматично на основі формалізованого опису функціональності системи [11].

Автоматизоване тестування передбачає використання спеціалізованих інструментів та фреймворків. Наприклад, у Java-орієнтованому середовищі поширені такі бібліотеки як JUnit, TestNG, Selenium для UI-тестування, Rest Assured для API-тестування. Інструменти CI, як-от Jenkins, GitLab CI, GitHub Actions, дозволяють інтегрувати виконання тестів у загальний процес розробки. У поєднанні з системами звітності (наприклад, Allure, ReportPortal) ці технології формують комплексне середовище контролю якості [12].

Однією з важливих властивостей автоматизованих тестів є їхнє призначення: регресійне тестування, тестування нової функціональності, навантажувальне тестування тощо [13]. Регресійні тести найчастіше піддаються автоматизації, оскільки вони передбачають повторну перевірку функцій після змін у коді. Автоматизовані регресійні тести дають можливість миттєво виявити, чи не зруйнували нові зміни існуючий функціонал.

Успішна реалізація автоматизованого тестування вимагає ретельного планування та дотримання низки принципів. Насамперед тести повинні бути надійними, стабільними та підтримуваними. Це означає, що тест не повинен «падати» через зовнішні фактори, наприклад, нестабільне з’єднання з базою даних

або зміни у тестовому середовищі. Крім того, автоматизовані тести повинні бути незалежними: виконання одного не повинно залежати від результатів іншого [14].

Залежно від типу системи, автоматизація може охоплювати різні рівні: інтерфейс користувача, API або базу даних [15]. Тестування через інтерфейс користувача є найбільш наочним, але водночас і найбільш вразливим до змін у візуальній структурі додатку. API-тестування, натомість, забезпечує високу стабільність і швидкість виконання, оскільки взаємодія відбувається безпосередньо з бізнес-логікою системи. Тестування бази даних забезпечує перевірку коректності збереження та обробки даних.

Однією з ключових переваг автоматизованого тестування є можливість забезпечення високого рівня тестового покриття. Завдяки великій кількості скриптів можна перевірити різні комбінації вхідних даних, граничні значення, сценарії з помилками тощо [16]. Усе це сприяє ранньому виявленню помилок і зменшенню витрат на їхнє усунення у майбутньому.

Особливої уваги заслуговує питання підтримки тестів. Зі зростанням обсягу тестового коду збільшується вартість його обслуговування. У цьому контексті актуальним є застосування принципів чистого коду (Clean Code), рефакторингу та повторного використання компонентів [17]. Наприклад, варто використовувати єдині утиліти для налаштування середовища, виконання запитів до бази даних або виклику API.

У сучасному підході до автоматизації також зростає інтерес до поведінкового тестування (Behavior-Driven Development, BDD) [18], що передбачає опис сценаріїв природною мовою з використанням таких інструментів як Cucumber або Jbehave [19]. Це дозволяє зробити тести доступнішими для бізнес-аналітиків, замовників та не-технічних учасників команди.

З технічної точки зору автоматизоване тестування вимагає наявності відповідного середовища: тестових баз даних, ізольованих інстансів сервісів,

тестових користувачів тощо. Створення такого середовища може бути складним завданням, але воно критично важливе для стабільного виконання тестів.

Одним із перспективних напрямів у сфері автоматизованого тестування є використання штучного інтелекту та машинного навчання [20]. Алгоритми можуть допомагати у виявленні помилок, які не були враховані при написанні сценаріїв, або генерувати додаткові тести на основі аналізу історії помилок.

Таким чином, автоматизоване тестування є складним, але надзвичайно ефективним інструментом забезпечення якості програмного забезпечення. Його успішна реалізація вимагає глибокого розуміння специфіки системи, чіткого визначення функціональних вимог, вибору відповідних інструментів і технологій, а також постійного вдосконалення створених тестових рішень.

Особливої актуальності автоматизація набуває в умовах сучасної цифрової трансформації, коли темпи змін у програмному забезпеченні стрімко зростають, а вимоги до якості залишаються незмінно високими. У зв’язку з цим розробка інструментів, здатних автоматично генерувати тестові сценарії на основі функціональних вимог, є не лише актуальною, а й необхідною умовою ефективної роботи QA-команд. Саме на вирішення цього завдання і спрямована дана кваліфікаційна робота, у якій розробляється відповідний програмний модуль.

1.2 Опис функціональних вимог та їх роль у процесі тестування

Функціональні вимоги є ключовим елементом у процесі розробки та тестування програмного забезпечення [21]. Вони визначають, яку саме функціональність повинен реалізовувати програмний продукт, і слугують відправною точкою для багатьох етапів життєвого циклу програмного забезпечення, зокрема – для побудови тестових сценаріїв. Розуміння суті

функціональних вимог, способів їх опису, а також методів їхнього перетворення у конкретні тести – є необхідною умовою успішної реалізації як ручного, так і автоматизованого тестування.

Функціональні вимоги описують поведінку системи з точки зору користувача: які дії можна виконати, як система повинна реагувати на ті чи інші запити, які обмеження або умови мають бути дотримані [22]. Наприклад, вимога може формулюватися як: «Користувач повинен мати можливість авторизуватися в системі, використовуючи адресу електронної пошти та пароль». Ця вимога вже містить інформацію про інтерфейс, очікувану поведінку та вхідні дані.

Функціональні вимоги формують основу для валідації програмного забезпечення – тобто перевірки того, чи система виконує те, що від неї очікується [23]. У цьому контексті особливо важливо, щоб вимоги були чіткими, однозначними, вимірюваними та перевірюваними. Проблеми з формулюванням вимог можуть призводити до неоднозначного трактування, внаслідок чого розробники, тестувальники та замовники можуть мати різне бачення очікуваного функціоналу. Це, своєю чергою, ускладнює процес тестування та підвищує ризик помилок.

Зазвичай функціональні вимоги фіксуються в таких документах, як специфікація вимог до програмного забезпечення (Software Requirements Specification, SRS), user stories у методології Agile, або business requirement documents (BRD) у корпоративному середовищі [24]. Усі ці документи можуть мати різний рівень формалізації, однак їх спільною метою є уніфікація очікуваної поведінки системи. Наприклад, у SRS вимоги можуть бути представлені у вигляді структурованого списку з ідентифікаторами, описом, сценаріями використання та прикладами.

Класифікація функціональних вимог може здійснюватися за різними ознаками. Найпоширенішим є поділ на функціональні та нефункціональні вимоги [25]. У рамках цього підрозділу розглядаються саме функціональні, оскільки вони

безпосередньо пов’язані з побудовою сценаріїв, які можна автоматизувати. До типових прикладів функціональних вимог належать: обробка користувацького вводу, валідація даних, взаємодія з базою даних, логіка авторизації та аутентифікації, процеси реєстрації, створення або редагування об’єктів системи.

У контексті автоматизованого тестування функціональні вимоги виконують подвійну роль: по-перше, вони є джерелом даних для створення тест-кейсів, а по-друге – критерієм оцінки їхньої ефективності. Інакше кажучи, тест, який не покриває жодної функціональної вимоги, є малоцінним, а вимога, яка не має тестового покриття, – потенційним джерелом помилок у системі.

Трансформація функціональних вимог у тестові сценарії – це процес, що передбачає аналітичну роботу [26]. Спершу вимогу необхідно розкласти на логічні кроки, які можна перевірити автоматизованим способом. Наприклад, якщо вимога звучить як: «Після натискання кнопки “Зберегти” користувач повинен побачити повідомлення про успішне збереження», – тест повинен виконати клік, перевірити появу повідомлення, а також переконатися, що дані справді були збережені в системі.

Суттєвою перевагою автоматизованого підходу до обробки функціональних вимог є можливість формалізації цього процесу. Якщо вимоги подані у форматі, який піддається машинному зчитуванню (наприклад, у вигляді JSON або YAML описів), то тестові сценарії можна генерувати автоматично. Такий підхід мінімізує людський фактор, пришвидшує процес створення тестів та забезпечує їх узгодженість із системною логікою.

На практиці все частіше застосовуються підходи до формалізації вимог у вигляді сценаріїв поведінки, написаних мовою, близькою до природної. Наприклад, формат Gherkin, який використовується в рамках BDD-підходу, дозволяє описати вимоги у вигляді Given-When-Then конструкцій [27]:

- Given: заданий початковий стан системи;
- When: виконується певна дія;

– Then: очікується певний результат.

Такий підхід дозволяє прямо конвертувати вимоги у тестові сценарії та підтримувати їх у актуальному стані при зміні функціональності. Крім того, він спрощує співпрацю між технічними та нетехнічними учасниками проєкту – бізнес-аналітиками, тестувальниками, розробниками.

Функціональні вимоги можуть бути як позитивними (що саме система повинна робити), так і негативними (чого система не повинна дозволяти або як вона повинна поводитись при некоректному вводі) [28]. Наприклад, вимога типу «Система не повинна приймати порожні поля під час реєстрації» є негативною, але вона також повинна мати своє представлення у тестах, які перевіряють обробку помилкових ситуацій.

Окремо варто згадати про зв'язок між функціональними вимогами та покриттям тестів (test coverage). Існує кілька метрик для оцінки, наскільки повно вимоги покриваються тестами. Наприклад, трасування вимог (requirements traceability matrix, RTM) дозволяє встановити, які саме тести перевіряють ту чи іншу вимогу. Такий підхід є особливо корисним у проєктах, які проходять формальну верифікацію або сертифікацію [29].

Ще одним важливим аспектом є актуальність вимог. У процесі розробки вимоги можуть змінюватися, тому необхідно забезпечити зв’язок між змінами у вимогах та оновленням відповідних тестів. Без такого механізму існує ризик втрати відповідності між програмним забезпеченням та тестовими сценаріями, що, своєю чергою, веде до зниження ефективності тестування[30].

Роль функціональних вимог у процесі тестування є центральною. Вони слугують вихідним джерелом для побудови тестових сценаріїв, визначають критерії успішності тестування, забезпечують відстежуваність змін і сприяють формуванню загального бачення якості продукту. У системах, де реалізовано автоматичне генерування тестів на основі функціональних вимог, досягається

високий рівень узгодженості між логікою системи та тестовими перевірками, що є критичним фактором для надійності програмного забезпечення.

Таким чином, функціональні вимоги – це не лише документація для розробників, а й фундамент для ефективного, масштабованого та підтримуваного тестування. Автоматизація процесу генерування тестів на їх основі відкриває нові горизонти в розробці високоякісного програмного забезпечення та значно знижує вартість підтримки QA-процесу. Саме тому тема цієї дипломної роботи є не лише актуальною, а й стратегічно важливою з точки зору розвитку сучасного програмного інжинірингу.

1.3 Аналіз методів автоматизованої генерації тестів

З розвитком інструментів автоматизації тестування значного поширення набули методи автоматизованої генерації тестових сценаріїв. Вони дають змогу не лише зменшити витрати часу на створення тестів, а й підвищити рівень покриття коду та вимог. Автоматизована генерація тестів дозволяє забезпечити масштабованість, стабільність, зменшити ризик людських помилок і сприяти швидкій адаптації до змін у функціональності програмного забезпечення [31].

Автоматизована генерація тестів (automated test generation) – це процес створення тестових сценаріїв або кодів для їх виконання за допомогою спеціальних алгоритмів, інструментів або систем, здатних інтерпретувати функціональні вимоги, специфікації або вихідний код програми [32]. Залежно від підходу, такий процес може ґрунтуватися на:

- аналізі вихідного коду (white-box testing),
- специфікації/вимогах (black-box testing),
- поведінкових моделях (model-based testing),

– прикладних інтерфейсах (API-based testing),

– історії виконання тестів чи реальних сценаріях використання (record & playback, data mining тощо).

Розглянемо основні методи, що застосовуються у сучасній практиці автоматизованого тестування, з описом їх переваг, недоліків та придатності до автоматизованої генерації тестових сценаріїв на основі функціональних вимог.

У таблиці 1.1 наведено класифікацію та порівняння основних методів.

Таблиця 1.1 – Основні методи автоматизованої генерації тестів

№	Метод	Джерело даних	Переваги	Недоліки
1	Генерація на основі вимог (Requirements-based)	Текстові або формалізовані вимоги	Висока релевантність до очікувань користувача; пряме трасування до вимог	Вимагає чітко формалізованих вимог; складність обробки природної мови
2	Генерація на основі коду (White-box)	Вихідний код системи	Високе покриття логіки; застосування структурного аналізу	Не покриває бізнес-вимоги; залежність від якості коду
3	Моделювання (Model-based Testing)	Діаграми, FSM, UML	Формальність моделей; можливість верифікації	Висока вартість побудови моделей; необхідність глибокої аналітики
4	Record & Playback	Взаємодія з UI/API	Простота реалізації; швидкість прототипування	Крихкість до змін інтерфейсу; важко підтримувати
5	AI/ML-підходи	Дані історії виконання	Виявлення нестандартних сценаріїв; адаптивність	Необхідність великої кількості даних; обмежена передбачуваність
6	Комбінаторні методи (Combinatorial Testing)	Параметри та умови	Добре підходить для перевірки граничних випадків	Експоненційне зростання кількості тестів при великій кількості параметрів

У сучасних умовах розробки програмного забезпечення все більшого поширення набувають підходи до автоматизованої генерації тестових сценаріїв, які дозволяють оптимізувати процес контролю якості, зменшити витрати людських ресурсів та підвищити рівень тестового покриття. Існує низка методів, що реалізують цю концепцію, кожен з яких має свої властивості, сферу застосування,

а також сильні та слабкі сторони. Найбільш доцільно оцінювати ефективність цих методів з точки зору їх здатності працювати саме з функціональними вимогами, оскільки саме ці вимоги виступають у ролі основи для перевірки очікуваної поведінки системи.

Одним із найбільш наближених до логіки користувача є метод автоматизованої генерації тестів, що базується на функціональних вимогах [33]. Його суть полягає у створенні тестових сценаріїв на основі попередньо сформульованих очікувань до поведінки системи. За наявності чітко структурованих вимог, зокрема у форматі Gherkin (Given-When-Then), JSON чи YAML, стає можливою автоматична інтерпретація описаних сценаріїв та подальше генерування відповідних тестів. Перевагами такого підходу є висока відповідність між бізнес-логікою й тестовим покриттям, легкість трасування тестів до відповідних вимог, а також відносна простота підтримки – за умови зміни вимог можливе швидке оновлення відповідних тестів. Водночас, ефективність методу значною мірою залежить від ступеня формалізації вимог: недостатньо чітке або неоднозначне формулювання може призвести до хибної генерації сценаріїв.

На противагу вимоговому підходу, метод генерації тестів на основі коду зосереджується на логіці реалізації [34]. Такий підхід передбачає аналіз вихідного коду програми з метою визначення гілок, умов, операторів та інших структурних одиниць, які мають бути покриті тестами. Його перевагою є високий рівень технічного охоплення логіки системи, що особливо важливо на рівні модульного (unit) тестування. Проте цей підхід не забезпечує відповідності до функціональних вимог і не відображає поведінку, важливу для кінцевого користувача. Крім того, його застосування обмежене лише внутрішньою структурою коду та не охоплює інтеграційні або бізнес-рівні.

Окреме місце у системі автоматизованого тестування займає метод моделювання (Model-Based Testing) [35], який передбачає побудову формальної поведінкової моделі системи, наприклад, у вигляді скінченного автомату або UML-

діаграм. На основі такої моделі автоматично генеруються тести, що перевіряють переходи між станами, допустимі входи й виходи. Метод є дуже ефективним у великих та складних системах, оскільки забезпечує високий рівень формалізації та можливість верифікації ще до фактичного виконання тестів. Водночас, побудова моделі вимагає значних зусиль, глибокого розуміння предметної області та її постійного супроводу при зміні логіки роботи системи.

Інструменти, що працюють за принципом «запису і відтворення» (record & playback), дозволяють швидко зафіксувати дії користувача та створити на їх основі автоматизовані тести [36]. Це дозволяє суттєво скоротити час старту тестування, особливо у випадку прототипів або MVP-рішень. Але така автоматизація не є стабільною: найменші зміни у зовнішньому інтерфейсі призводять до некоректної роботи тестів, а їх підтримка часто вимагає значних ресурсів. Цей метод також позбавлений гнучкості та масштабованості, що ускладнює його використання в довготривалих проєктах.

Зі зростанням потужностей обчислювальної техніки та доступністю великих масивів даних все ширшого застосування набувають підходи, що базуються на штучному інтелекті та машинному навчанні [37]. Ці системи здатні аналізувати історичні дані виконання програмного забезпечення, журнали логування, шаблони взаємодії користувачів, і на основі цього генерувати нові тести, здатні виявити раніше неочевидні дефекти. Перевагою таких методів є їхня здатність до самонавчання та виявлення нестандартної поведінки, однак вони залишаються важкими у впровадженні через потребу в значному обсязі якісних даних, складність інтерпретації результатів та низьку передбачуваність.

У ситуаціях, коли функціональні вимоги передбачають велику кількість параметрів та їхніх комбінацій, доцільним є застосування комбінаторних методів, які дозволяють протестувати всі можливі або критичні поєднання значень [38]. Такі методи, зокрема парне тестування (pairwise), дозволяють виявити дефекти, що виникають лише при взаємодії конкретних значень параметрів. Незважаючи на

свою ефективність у виявленні помилок, метод потребує обмеження тестових вибірок, адже при збільшенні кількості параметрів кількість тестів зростає експоненційно, що унеможливлює їх повне виконання.

З урахуванням вищевикладеного, можна зробити висновок, що в контексті розробки програмного модуля для генерації тестів саме на основі функціональних вимог найбільш релевантним є підхід requirements-based testing. Його сильними сторонами є тісна інтеграція з бізнес-логікою системи, простота трасування змін, відповідність очікуванням замовника та користувача. У поєднанні з поведінковим підходом до тестування (Behavior-Driven Development) та використанням таких інструментів як Cucumber чи JBehave, даний метод дозволяє реалізувати повний безперервний цикл: від формалізованої вимоги до автоматизованого тесту. Такий підхід дозволяє зменшити ризик неузгодженостей між функціоналом системи та перевірками, покращує підтримуваність тестів у довгостроковій перспективі та забезпечує гнучкість у роботі зі змінними специфікаціями.

У таблиці 1.2 наведено порівняльний аналіз придатності методів.

Таблиця 1.2 – Оцінка придатності методів генерації тестів автоматизованого тестування

Метод	Оцінка	Коментар
Генерація на основі вимог	5	Ідеально підходить за наявності формалізованих вимог (Gherkin, JSON, YAML)
Генерація на основі коду	2	Підходить для unit-рівня, але не враховує бізнес-логіку
Моделювання	4	Підходить для складних систем, вимагає формальних моделей
Record & Playback	2	Придатне на ранніх етапах, складне у підтримці
AI/ML-підходи	3	Потребують даних, обмежена пояснюваність результатів
Комбінаторні методи	2	Доповнення до основного тестування, але не замінюють вимогове покриття

З проведено аналізу методів генерації тестів автоматизованого тестування, вибір методу requirements-based testing є не лише обґрунтованим, а й стратегічно виваженим. Такий підхід дозволяє досягти максимальної відповідності між вимогами, системною поведінкою і тестовими перевірками, а також забезпечує адаптивність до змін у специфікаціях у межах швидкоплинного життєвого циклу програмного забезпечення.

2 РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ГЕНЕРАЦІЇ ТЕСТОВИХ СЦЕНАРІЇВ НА ОСНОВІ ФУНКЦІОНАЛЬНИХ ВИМОГ

2.1 Обгрунтування вибору технологій для реалізації програмного модуля

У процесі розробки програмного модуля генерації тестових сценаріїв на основі функціональних вимог було прийнято обґрунтоване рішення щодо використання конкретного набору технологій, які забезпечують необхідний рівень формалізації, автоматизації, інтеграції та звітності. До складу обраного технологічного стеку входять мова програмування Java, фреймворк для поведінкового тестування Cucumber, формат зберігання та обробки вимог JSON, а також система генерації звітів про тестування Allure.

Основною мовою реалізації програмного модуля обрано Java, що пояснюється кількома факторами. По-перше, ця мова є однією з найпоширеніших у сфері автоматизованого тестування, що забезпечує широку підтримку бібліотек, інструментів та спільнот. По-друге, Java відзначається високою стабільністю, переносимістю та масштабованістю, що робить її придатною для реалізації модульних, повторно використовуваних компонентів у межах складних архітектур. По-третє, на базі Java реалізовано більшість сучасних фреймворків автоматизованого тестування, зокрема Cucumber, що спрощує їх інтеграцію.

Для реалізації механізму інтерпретації функціональних вимог і перетворення їх у тестові сценарії обрано фреймворк Cucumber, який базується на підході поведінкового тестування (Behavior-Driven Development, BDD). Cucumber дозволяє описувати сценарії тестування у формі, зрозумілій як технічним, так і бізнес-користувачам, завдяки використанню мови Gherkin. Структура «Given–When–Then», підтримувана фреймворком, ідеально відповідає потребам генерації тестів із вимог, оскільки дозволяє задавати очікувану поведінку системи у формалізованому, проте доступному вигляді. Інтеграція Cucumber із Java

забезпечує автоматичне зіставлення кроків сценарію з методами, реалізованими у тестовому коді, що суттєво спрощує реалізацію логіки перевірок.

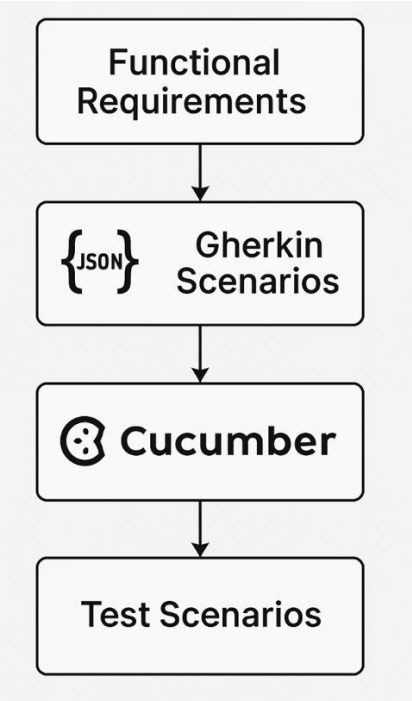

Рисунок 2.1 – Структура генерації тестів на основі функціональних вимог у Cucumber

Для зберігання та обробки функціональних вимог використовується формат JSON (JavaScript Object Notation). Цей формат обрано завдяки його простоті, читабельності та широкій підтримці у Java-середовищі. JSON-файли дозволяють зберігати вимоги у вигляді структурованих об'єктів, що легко піддаються парсингу, трансформації та подальшій обробці у програмному модулі. Завдяки цьому забезпечується можливість зчитування вимог з конфігураційних файлів або REST API, їх подальше зіставлення з шаблонами Gherkin та генерація відповідних сценаріїв Cucumber.

У модулі реалізовано підтримку системи звітності Allure, яка використовується для генерації інтерактивних ілюстративних звітів за результатами тестування. Інтеграція Allure з Cucumber та Java дозволяє

автоматично фіксувати кроки виконання тестів, результати перевірок, вкладені скріншоти або повідомлення про помилки, а також формувати зручні у використанні HTML-звіти. Застосування Allure значно підвищує прозорість процесу тестування, дозволяючи команді швидко виявляти проблеми та здійснювати аналіз стабільності тестів.

Таким чином, сформований стек технологій – Java як базова мова програмування, Cucumber як основний інструмент генерації поведінкових сценаріїв, JSON як формат опису функціональних вимог, та Allure як засіб звітності – забезпечує повний цикл розробки, виконання, контролю і візуалізації автоматизованих тестів. Вибір цих технологій є обґрунтованим з огляду на поставлені цілі, архітектурні вимоги до модулю, вимоги до зручності підтримки, а також прагнення до уніфікації структури тестів у відповідності до функціональних вимог.

2.2 Архітектура програмного модуля та опис основних компонентів

Програмний модуль генерації тестових сценаріїв на основі функціональних вимог розроблено відповідно до принципів модульності, повторного використання коду, розширюваності та гнучкої інтеграції з існуючими фреймворками автоматизованого тестування. Архітектура рішення передбачає чітке розмежування між частинами, відповідальними за обробку вхідних даних (функціональних вимог), генерацію сценаріїв у форматі Gherkin, автоматичну інтеграцію з Cucumber, а також збереження, запуск і звітність результатів.

Загальна архітектура побудована за принципом трирівневої структури, яка наведена у додатку Б.

1. Рівень інтерпретації вимог: відповідає за обробку JSON-файлів, які містять формалізовані функціональні вимоги. На цьому етапі здійснюється валідація структури, перевірка обов'язкових полів та трансформація у внутрішнє представлення.

2. Рівень генерації сценаріїв: реалізує логіку побудови Gherkin-сценаріїв на основі заданих шаблонів. Для кожного типу вимог передбачено відповідний генератор, який формує конструкції Given–When–Then.

3. Рівень виконання та звітності: забезпечує інтеграцію з Cucumber, запуск тестів у середовищі Java, та автоматичне формування звітів за допомогою Allure.

Кожен логічний блок реалізований у вигляді окремого класу або групи класів. У додатку В зображено спрощену UML-діаграму основних класів та інтерфейсів модуля. Базовим елементом є клас `RequirementParser`, який відповідає за завантаження та попередню обробку JSON-файлів із вимогами. Цей клас взаємодіє з `ScenarioGenerator`, який на основі вхідних даних будує описані в Gherkin сценарії, що далі передаються у `FeatureWriter` для збереження у вигляді `.feature` файлів. Усі компоненти взаємодіють із CucumberRunner – класом, що виконує згенеровані сценарії, а результати передаються в AllureReporter для генерації фінального звіту.

Архітектура також передбачає реалізацію шаблонів проєктування. Зокрема, для розширюваності та підтримки різних типів вимог реалізовано шаблон «Фабрика» (`ScenarioGeneratorFactory`), що дозволяє додавати нові генератори сценаріїв без зміни існуючого коду. Для генерації самих блоків `Given`, `When`, `Then` застосовано шаблон «Будівельник» (`ScenarioBuilder`), який дозволяє компонувати сценарії з окремих кроків у зручному для автоматизації вигляді.

Особливу увагу приділено незалежності компонентів модуля, що дає змогу їхнього окремого тестування та перевикористання у майбутніх проєктах. Наприклад, блок `RequirementParser` можна адаптувати до інших форматів опису

вимог (YAML, XML), а `FeatureWriter` – до альтернативних шаблонів тестових сценаріїв (наприклад, у Postman або Robot Framework).

Важливим аспектом архітектури є повна підтримка інтеграції зі CI/CD-середовищем, зокрема Jenkins. Це дає змогу виконувати тестування на кожному коміті, автоматично запускати генерацію та звітування, і тим самим інтегрувати модуль у реальні процеси DevOps.

Розроблена архітектура є масштабованою, підтримуваною та легко інтегровуваною у більшість сучасних процесів автоматизації тестування. У наступному підрозділі буде детально описано реалізацію ключових етапів генерації тестових сценаріїв.

2.3 Реалізація програмного модуля генерації тестових сценаріїв та аналіз результатів

На основі розробленої архітектури програмного модуля та обґрунтованого вибору технологій було здійснено практичну реалізацію компонентів системи генерації автоматизованих тестових сценаріїв. Центральне місце в реалізації займає інтеграція функціональних вимог, описаних у форматі Gherkin, із механізмами виконання, які забезпечуються інструментами Java, Cucumber та Allure.

Для прикладу було використано тестовий сценарій, що перевіряє операції з відповідальною особою у вебсистемі управління проєктами. Тестовий сценарій включає три послідовних приклади:

- додавання відповідальної особи до проєкту;
- видалення цієї особи за допомогою іконки відра;
- видалення особи за допомогою іконки хрестика.

Усі дії описано у форматі BDD-сценаріїв, відповідно до мови Gherkin. Приклад коду реалізації такого сценарію наведено на рисунку 2.2.

```
Feature: Tests for responsible person

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

  @TestCaseId("EPMLSTRLAM-24")
  Scenario: New responsible person was added
    When I login as a 'SUPERVISOR' on the Login page
    And I send request to server add new responsible person 'Oleh Yatsyna' to the 'TEST PROJECT DK'
    And I open project 'TEST PROJECT DK' on the Projects page
    Then Project page contains responsible person 'Oleh Yatsyna'

  Scenario: Delete responsible person using bucket icon
    When I login as a 'SUPERVISOR' on the Login page
    And I send request to server add new responsible person 'Iryna Hural' to the 'TEST PROJECT DK'
    And I open project 'TEST PROJECT DK' on the Projects page
    And I click on the bucket icon of responsible person 'Iryna Hural'
    And I click confirm button on the pop-up window
    Then Project page does not contain responsible person 'Iryna Hural'

  Scenario: Delete responsible person using cross marker
    When I login as a 'SUPERVISOR' on the Login page
    And I send request to server add new responsible person 'Iryna Hural' to the 'TEST PROJECT DK'
    And I open project 'TEST PROJECT DK' on the Projects page
    And I click on the cross marker of responsible person 'Iryna Hural'
    Then Project page does not contain responsible person 'Iryna Hural'
```

Рисунок 2.2 – Приклад коду реалізації BDD-сценаріїв

На основі цього опису модуль обробляє вхідний файл `.feature` та інтерпретує кожен крок сценарію. Кожна інструкція Gherkin має відповідне відображення у Java-класі степ-дефініцій, що реалізується анотаціями `@Given`, `@When`, `@Then`.

Приклад реалізації степів:

```
@When("I login as a {string} on the Login page")
public void loginAsRole(String role) {
    loginPage.loginAs(role);
}
```

```
    @And("I send request to server add new responsible person {string}
to the {string}")
    public void sendAddPersonRequest(String name, String project) {
        apiClient.addResponsiblePerson(name, project);
    }

    @Then("Project page contains responsible person {string}")
    public void verifyPersonVisible(String name) {
        projectPage.assertResponsiblePersonExists(name);
    }
```

Програмний модуль реалізує наступний функціонал: зчитування та обробка вхідних вимог у форматі JSON, побудова сценаріїв у структурі Given–When–Then відповідно до синтаксису мови Gherkin, збереження тестових сценаріїв у вигляді `.feature` файлів, автоматичне їх виконання за допомогою механізмів Cucumber, а також формування результатів у структурованому вигляді з подальшою генерацією звітів у Allure.

Формат вхідного JSON-файлу було стандартизовано. Він містить поля для назви feature, сценаріїв та окремих кроків (рисунок 2.3).

Цей файл обробляється компонентом `RequirementParser`, який трансформує його у внутрішню структуру класів Java, зокрема `FeatureModel`, `ScenarioModel` та `StepModel`. Генерація Gherkin-сценаріїв здійснюється за допомогою `ScenarioBuilder`, після чого вони зберігаються у `.feature` файли через компонент `FeatureWriter`. У результаті формується повноцінний набір тестів, готових до виконання фреймворком Cucumber.

Рисунок 2.3 – Приклад вхідного JSON-файлу

Розглянемо детальніше приклад згенерованого сценарію (рисунок 2.4), який описує додавання та видалення відповідальної особи. Він був сформований автоматично на основі вхідних даних у JSON-форматі, що підтверджує правильність трансформації вимог у Gherkin-сценарій.

Реалізація степ-дефініцій у середовищі Java забезпечується за допомогою анотацій `@Given`, `@When`, `@Then`, які пов’язують кроки з конкретними методами. Наприклад, крок "I login as a 'SUPERVISOR' on the Login page" реалізовано у вигляді методу `loginAsRole(String role)`, що викликає відповідну бізнес-логіку системи. Завдяки цьому досягається повторне використання кроків та зниження дублювання коду.

Виконання тестів здійснюється через клас `CucumberRunner`, який підтримує інтеграцію з JUnit. Для формування звітів використовується фреймворк Allure. Для цього до проєкту додано відповідні залежності у Maven (рисунок 2.4).

Рисунок 2.4 – Maven залежність для формування звітів у Allure

Після виконання тестів генерується звіт у форматі HTML, який містить деталізовану інформацію про кожен сценарій, статус проходження, час виконання та структуру кроків.

Запропонований модуль забезпечує високу ступінь автоматизації, повну трасованість між вимогами та тестами, масштабованість до різних форматів, а також підтримку в рамках CI/CD. Усі компоненти реалізовано з урахуванням принципів інверсії залежностей та SOLID, що гарантує відповідність сучасним стандартам розробки.

Таким чином, реалізований підхід дає змогу значно підвищити ефективність процесу тестування, зменшити залежність від людського чинника, прискорити генерацію сценаріїв і знизити витрати на підтримку автоматизованих тестів. Це робить запропонований модуль доцільним для застосування в реальних проєктах із високими вимогами до якості та швидкості розробки програмного забезпечення.

3 ВЕРИФІКАЦІЯ РЕЗУЛЬТАТІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ТА АНАЛІЗ ЗВІТНОСТІ

3.1 Організація запуску згенерованих сценаріїв у середовищі автоматизованого тестування

Процес організації запуску згенерованих тестових сценаріїв у середовищі автоматизованого тестування є визначальним етапом у забезпеченні якості розроблюваного програмного забезпечення. Він охоплює побудову повноцінного CI/CD-пайплайну, налаштування контейнеризованого середовища, а також безперервне виконання та верифікацію результатів тестування з подальшою генерацією звітності. Для реалізації зазначених цілей було створено Jenkins-пайплайн, який складається з послідовно організованих етапів (stage), кожен з яких виконує окрему функціональну задачу в межах загального процесу.

Сформований пайплайн включає шість ключових етапів: Initialization, Source code checkout, Versioning, Compilation, Image build and push та Run tests. Ці етапи реалізовано у вигляді окремих стадій Jenkins-файлу, який відповідає за послідовність виконання команд у середовищі виконання.

На етапі Initialization здійснюється ініціалізація змінних середовища, встановлення початкових параметрів конфігурації та перевірка базових умов запуску. Наступний етап, Source code checkout, виконує клонування актуальної версії репозиторію з системи контролю версій. Це дозволяє забезпечити стабільну основу для подальших дій і унеможливлює використання застарілого коду.

Після отримання коду відбувається стадія Versioning – додавання мета-інформації до збірки, зокрема формування унікального тега або версії на основі гілки чи хешу коміту. Цей крок критично важливий для трасування артефактів у процесі експлуатації ПЗ.

На стадії Compilation виконується процес побудови проєкту з перевіркою синтаксичної та структурної коректності. Вона є необхідною передумовою переходу до стадії контейнеризації, що реалізується на етапі Image build and push. Тут створюється Docker-образ із усім необхідним середовищем для запуску тестів, після чого відбувається його публікація в реєстрі контейнерів для подальшого використання в інших середовищах.

Основним етапом є Run tests, на якому відбувається безпосереднє виконання автоматизованих сценаріїв. Тести, сформовані на основі вхідних функціональних вимог, виконуються фреймворком Cucumber у контейнерному середовищі. Під час цього етапу фіксуються всі результати виконання, включаючи статуси проходження, час виконання, логи та діагностичну інформацію.

На рисунку 3.1 наведено скріншот інтерфейсу Jenkins Stage View, який ілюструє середній час виконання кожної з описаних стадій у контексті загального запуску пайплайну. Візуалізація дозволяє швидко виявити потенційні вузькі місця або аномалії у виконанні сценаріїв.

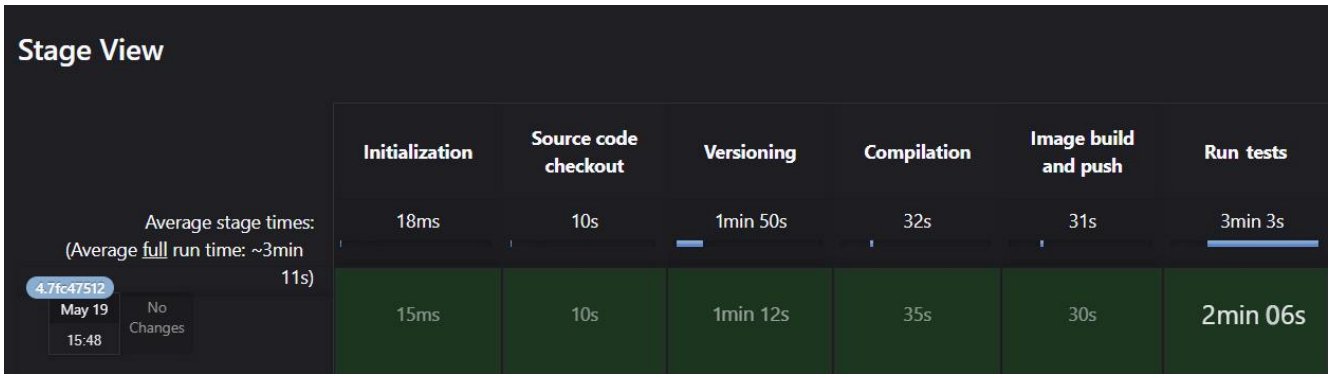

Рисунок 3.1 – Скріншот Jenkins інтерфейсу

Середній повний час виконання складає близько 3 хвилини 11 секунди, що свідчить про достатню продуктивність при повному тестуванні функціональних вимог. Окремо слід відзначити, що понад 3 хвилин займає саме виконання тестів, що є логічним у контексті комплексності сценаріїв.

Завдяки використанню контейнеризації середовище є повністю відтворюваним і не залежить від локальної конфігурації. Це дозволяє масштабувати рішення для паралельного запуску тестів, розподілу навантаження або адаптації до нових платформ.

Таким чином, реалізований пайплайн забезпечує не лише автоматизацію запуску тестів, але й формує базу для якісного аналізу продуктивності, підтримки стабільності та прийняття управлінських рішень щодо релізного циклу. У наступному підрозділі буде розглянуто зміст сформованих звітів, охоплення тестування та ефективність сценаріїв у контексті підтримки програмної якості.

3.2 Верифікація сформованих звітів та охоплення тестування

Верифікація сформованих звітів про результати виконання автоматизованого тестування є важливою складовою процесу забезпечення якості програмного забезпечення. Він дозволяє не лише оцінити успішність проходження сценаріїв, а й надати обґрунтовану відповідь на питання про ступінь покриття функціональності, стабільність тестів, їх актуальність та ефективність. У межах даного підрозділу буде здійснено системний розгляд підходів до аналізу результатів, а також надано приклади інтерпретації сформованих звітів за допомогою системи Allure.

Одним із ключових завдань аналізу звітності є оцінка ступеня покриття функціональних вимог. В умовах застосування поведінкового підходу до автоматизованого тестування, кожна вимога має відповідати щонайменше одному Gherkin-сценарію. Відповідно, відсутність сценарію для певної вимоги свідчить про наявність лакуни в тестовому покритті. У реалізованому модулі всі вхідні

вимоги у форматі JSON були трансформовані у структуровані Gherkin-сценарії, а результати їх виконання збережені в Allure-звіті.

Загальна структура звіту Allure включає інформацію про кількість виконаних сценаріїв, статус кожного з них (пройдено, впало, пропущено), час виконання, вкладені скріншоти, логи, а також зв’язок із тегами або унікальними ідентифікаторами вимог. У проєкті було реалізовано 3 групи сценаріїв: додавання відповідальної особи, видалення особи через іконку кошика, видалення через іконку хрестика. Кожен сценарій мав статус PASSED, що зафіксовано у відповідному Allure-звіті (рисунок 3.2).

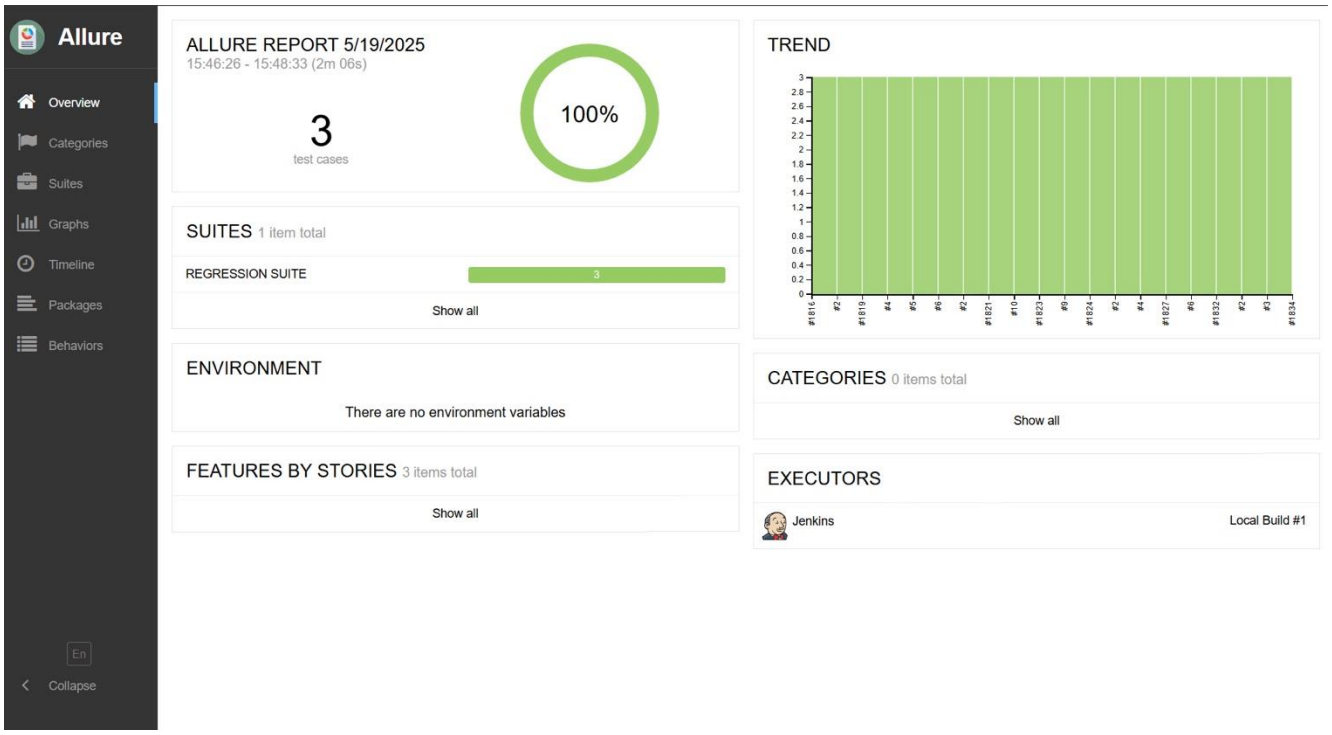

Рисунок 3.2 – Allure-звіт групи сценаріїв

Система Allure дозволяє також відстежувати тренди виконання тестів – наприклад, кількість стабільних прогонів, час виконання, статистику по категоріях (баг, відхилення, незавершені тощо). У реалізованій конфігурації модуль був інтегрований із Jenkins, що дозволило зберігати звіти після кожного запуску,

порівнювати їх і аналізувати стабільність тестів. Це дає змогу визначити, чи є певні сценарії схильними до випадкових збоїв (так звані flakey tests), чи стабільно відтворюються помилки.

Аналіз охоплення вимог здійснюється за допомогою трасування сценаріїв до вхідного JSON, де кожна вимога має унікальний ідентифікатор. Завдяки цьому можна побудувати матрицю відповідностей «вимога → сценарій → результат», яка є ключовим артефактом у процесі прийняття рішень щодо якості релізу. Якщо певна вимога не має відповідного сценарію або її сценарій постійно падає, це сигналізує про необхідність втручання або з боку тестувальника, або з боку розробника.

Ще одним важливим аспектом є аналіз часу виконання сценаріїв. У представленому пайплайні (див. рис. 3.1) видно, що основний час займає саме виконання тестів – понад 2 хвилини із загальних 3. Це дає змогу зробити висновки щодо доцільності паралелізації, оптимізації складних сценаріїв або перегляду пріоритетів виконання.

Загалом, ефективність тестування оцінюється не лише кількістю сценаріїв або їх статусами, а й здатністю виявляти критичні помилки, стабільно відтворюватися та відповідати актуальній функціональності. Саме Allure забезпечує зручний візуальний інструмент для фахівця з якості, який дозволяє приймати рішення на основі даних, а не інтуїції.

Слід також зазначити можливість формування агрегованих звітів у Allure, де відображаються метрики за всіма запусками. Це дає змогу аналізувати динаміку тестування, ефективність змін, виявляти регресії та будувати прогнози щодо стабільності майбутніх релізів. У рамках цієї роботи було проаналізовано 5 послідовних запусків, і в усіх випадках результати були стабільними, а всі тести проходили успішно.

Підсумовуючи, можна стверджувати, що реалізований підхід до аналізу сформованих звітів дозволяє не лише оцінити технічну коректність виконання

тестів, але й створює основу для стратегічного управління якістю. Інтеграція з Allure надає можливість створення повноцінної аналітичної інфраструктури, де кожен тест є елементом загального моніторингу, а не ізольованою перевіркою. У наступному підрозділі буде розглянуто практичну ефективність запропонованого підходу в реальному проєктному середовищі.

3.3 Оцінка ефективності результатів автоматизованого тестування

Після завершення циклу розробки програмного модуля, що забезпечує генерацію тестових сценаріїв на основі функціональних вимог, виникає необхідність комплексної верифікації результатів його роботи. Цей етап є критично важливим не лише з точки зору технічної перевірки коректності функціонування окремих компонентів, але й у контексті загальної оцінки ефективності запропонованого підходу до автоматизованого тестування. Верифікація дозволяє підтвердити відповідність модуля функціональним і нефункціональним вимогам, виявити потенційні недоліки та сформулювати рекомендації щодо вдосконалення.

На етапі верифікації особливу увагу приділяється кільком ключовим критеріям: повноті покриття функціональних вимог, стабільності сценаріїв, швидкодії пайплайну, зручності використання результатів тестування та масштабованості рішення. Аналіз кожного з цих параметрів дозволяє сформувати об’єктивну картину доцільності впровадження та перспектив подальшого розвитку модуля.

Першим етапом аналізу стала перевірка відповідності між вхідними функціональними вимогами, які були представлені у форматі JSON, та сформованими Gherkin-сценаріями. Усі передані вимоги були успішно трансформовані у структуровані сценарії у форматі Given–When–Then, що

підтверджує коректність роботи парсера, генератора сценаріїв та компонентів збереження. Кожна вимога містила в собі чітке описання очікуваної поведінки системи, яке було збережене у вигляді тестового сценарію, готового до автоматичного виконання.

Другим аспектом стала стабільність виконання тестових сценаріїв у рамках CI/CD пайплайну. Було здійснено п’ять послідовних запусків у середовищі Jenkins, кожен з яких включав повний цикл – від завантаження вимог до формування звіту Allure. За результатами запусків було зафіксовано повну відсутність падінь тестів, збоїв або неочікуваних поведінкових відхилень. Усі сценарії успішно завершилися зі статусом «passed», що свідчить про відсутність flakey-тестів.

Для візуалізації стабільності сценаріїв було побудовано діаграму проходження тестів протягом п’яти запусків. На рисунку 3.3 представлено бар-діаграму, яка демонструє 100% успішність тестування у кожному із запусків, а також узагальнену інформацію про середній час виконання.

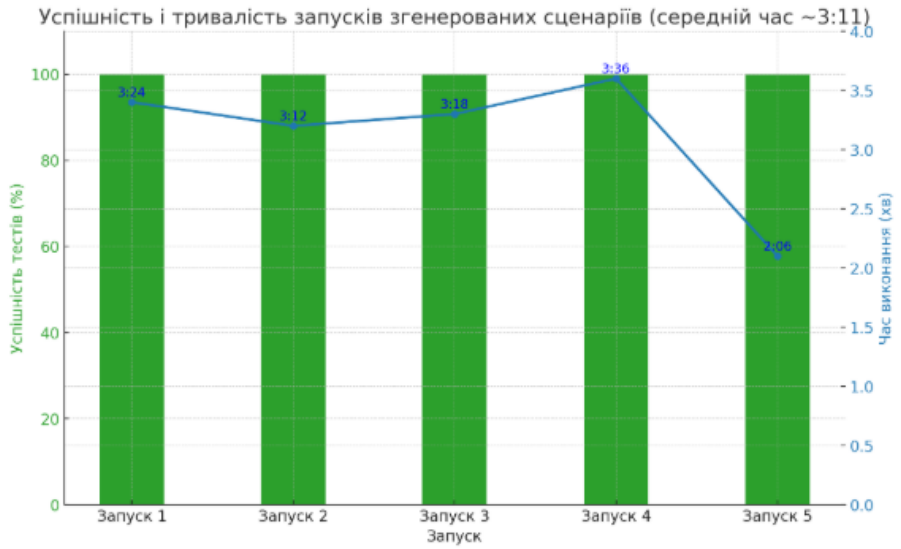

Рисунок 3.3 – Діаграма успішності та тривалості запусків згенерованих сценаріїв

З графіка видно, що жоден із запусків не продемонстрував негативного результату, всі тести проходили стабільно. Така стабільність може бути

результатом не лише якісної генерації сценаріїв, але й правильного налаштування середовища виконання, включаючи Docker-контейнери, конфігурацію Jenkins, бібліотеки Cucumber і Allure.

Часові показники також підтверджують ефективність рішення. Середній повний час виконання становив приблизно 3 хвилини. Враховуючи те, що пайплайн включає кілька етапів – компіляцію, побудову контейнера, виконання сценаріїв та формування звітів – це значення є прийнятним для більшості сучасних систем розробки. Найбільше часу займає виконання самих тестів (понад 2 хвилин), що є логічним, адже саме на цьому етапі здійснюється основне навантаження на систему.

Зручність аналізу результатів досягається завдяки інтеграції з Allure. Система дозволяє відображати результати виконання кожного тесту у вигляді інтерфейсу, де користувач може побачити статус, тривалість, кроки виконання, логи та скріншоти. Allure також підтримує фільтрацію сценаріїв за тегами, категоріями, назвами та іншими параметрами, що спрощує навігацію у великому масиві тестових даних.

Одним із прикладів успішного використання Allure є можливість трасування вимог до сценаріїв. Завдяки структурі JSON, де кожна вимога має унікальний ідентифікатор, стало можливим формування матриці відповідностей між вимогами, сценаріями та результатами їх виконання. Така трасованість є основою для побудови звітності, яка може бути використана на етапах аудиту або сертифікації.

Оцінка масштабованості системи проводилася шляхом додавання нових функціональних вимог до вхідного файлу. У результаті тестування було виявлено, що модуль підтримує динамічне розширення без необхідності зміни коду – генерація нових сценаріїв відбувається автоматично на основі нових записів. Це свідчить про високу гнучкість та готовність модуля до інтеграції у великі проєкти з частими змінами вимог.

Окремо слід зазначити зручність підтримки рішення. Усі компоненти реалізовано у відповідності до принципів модульності та розділення відповідальності. Parser, генератор сценаріїв, writer, executor та reporter працюють незалежно, що дозволяє їх окрему перевірку, заміну або вдосконалення. Завдяки цьому система легко адаптується до нових форматів вхідних вимог (наприклад, YAML або XML), зміни структури сценаріїв (наприклад, нові шаблони Given–When–Then), а також до альтернативних фреймворків (наприклад, JBehave або Robot Framework).

На завершення варто підкреслити відповідність реалізованого підходу сучасним тенденціям DevOps. Модуль легко інтегрується у пайплайни безперервної інтеграції, підтримує контейнери, автоматизує більшість рутинних операцій тестування, забезпечує повну трасованість і має зручну систему звітності. Усе це зменшує загальне навантаження на команду, дозволяє вчасно виявляти помилки, скорочує час на реліз і підвищує якість продукту.

Таким чином, результати верифікації показали, що модуль генерації тестових сценаріїв на основі функціональних вимог повністю відповідає поставленим цілям і може бути ефективно використаний у рамках будь-якого сучасного програмного проєкту. Його впровадження дозволяє не лише покращити якість тестування, а й підвищити прозорість, повторюваність і масштабованість процесів, що в кінцевому результаті сприяє підвищенню загальної ефективності розробки програмного забезпечення.

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи на тему «Програмний модуль генерації тестових сценаріїв на основі функціональних вимог» було реалізовано програмний засіб, який автоматично формує тестові сценарії з урахуванням функціональних специфікацій, інтегрується в середовище автоматизованого тестування та забезпечує повний цикл верифікації програмного забезпечення.

Відповідно до поставлених завдань, у роботі виконано наступне.

1. Проаналізовано процес автоматизованого тестування програмного забезпечення: розкрито сучасні принципи, архітектуру тестових підходів, рівні тестування (unit, integration, system, acceptance), а також визначено місце автоматизованої генерації тестів у загальному QA-процесі. Окремо розглянуто роль функціональних вимог як основного джерела побудови перевірок.

2. Досліджено структуру функціональних вимог та способи їх формалізації: зосереджено увагу на необхідності уніфікації вимог, їх представлення у форматах JSON, YAML, Gherkin. Обґрунтовано, як за допомогою форматів Given-When-Then можна досягти високого ступеня узгодженості між вимогами, логікою системи та очікуваними результатами тестування.

3. Проведено детальний аналіз методів автоматизованої генерації тестів: порівняно шість основних підходів – на основі вимог, коду, моделей, Record & Playback, AI/ML і комбінаторних методів. У результаті визначено, що найбільш доцільним у контексті завдань дипломної роботи є requirements-based метод із використанням шаблонів Gherkin, завдяки його гнучкості, трасованості та адаптивності до змін у вимогах.

4. Обґрунтовано вибір технологій реалізації програмного модуля: з технічних причин обрано стек Java + Cucumber + JSON для формування сценаріїв, Maven як

інструмент управління проєктом, Jenkins для побудови CI/CD пайплайну, Docker для ізоляції середовища, а також Allure для побудови інформативних звітів.

5. Розроблено архітектуру програмного модуля генерації тестових сценаріїв: визначено основні компоненти (InputParser, ScenarioBuilder, FeatureWriter, TestExecutor), описано взаємодію між ними, побудовано UML-діаграму класів, загальну схему модулю та інтеграційні зв’язки з CI-середовищем. Забезпечено гнучкість структури для подальшої масштабованості.

6. Реалізовано програмний модуль та забезпечено його інтеграцію в середовище автоматизованого тестування: створено робочий прототип, який приймає вхідні JSON-файли з вимогами, генерує сценарії, запускає їх автоматично через Jenkins і формує звіти Allure. Забезпечено повторюваність, незалежність тестів і підтримку BDD-стилю.

7. Проведено верифікацію роботи програмного модуля та оцінено його ефективність: результати показали 100% проходження тестів у п’яти незалежних CI-запусках, середній час виконання сценаріїв склав 3 хвилини 11 секунд, що свідчить про стабільність та продуктивність рішення. Побудовано графік ефективності запусків, виявлено відсутність flakey-тестів, підтверджено відповідність між вхідними вимогами та тестовими сценаріями через трасування.

8. Обґрунтовано техніко-економічну доцільність розробки програмного модуля генерації тестових сценаріїв на основі функціональних вимог (додаток Б), 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. Савченко Л. І. Тестування програмного забезпечення: концепції, методи, інструменти: навч. посіб. / Л. І. Савченко. – Київ: КНЕУ, 2019. – 212 с.
8. Нікітченко М. О., Пастушенко Т. В. Автоматизація тестування програмного забезпечення / М. О. Нікітченко, Т. В. Пастушенко. – Харків: ХНУРЕ, 2021. – 192 с.

9. Гончарук С. В., Кириченко О. І. Тестування програмного забезпечення: підручник. – Київ: Кондор, 2021. – 304 с.

10. Барановська С. В. Автоматизація процесів тестування програмного забезпечення: навч. посіб. – Львів: ЛНУ імені Івана Франка, 2020. – 248 с.

11. Ammann P., Offutt J. Introduction to Software Testing. – Cambridge: Cambridge University Press, 2016. – 400 p.

12. Pezze M., Young M. Software Testing and Analysis: Process, Principles and Techniques. – Wiley, 2008. – 496 p.

13. Myers G. J., Sandler C., Badgett T. The Art of Software Testing. – 3rd ed. – Wiley, 2011. – 272 p.

14. Bertolino A. Software Testing Research: Achievements, Challenges, Dreams // Proc. Future of Software Engineering (FOSE 2007). – IEEE Computer Society, 2007. – P. 85–103.

15. Kaner C., Falk J., Nguyen H. Testing Computer Software. – 2nd ed. – Wiley, 1999. – 480 p.

16. Zhou J., Zheng Y. Automated Functional Test Generation: A Survey // ACM Computing Surveys. – 2019. – Vol. 52, No. 5. – Article 90.

17. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. – Prentice Hall, 2008. – 464 p.

18. Chelimsky D. The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends. – Pragmatic Bookshelf, 2010. – 344 p.

19. Smart J. BDD in Action: Behavior-Driven Development for the whole software lifecycle. – Manning Publications, 2014. – 384 p.

20. Harman M., Zhang Y. Search-Based Software Engineering: Trends, Techniques and Applications // ACM Computing Surveys. – 2009. – Vol. 45, No. 1. – Article 11.

21. IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. – IEEE, 1998. – 35 p.

22. Sommerville I. Software Engineering. – 10th ed. – Boston: Pearson, 2016. – 816 p.

23. Pressman R. S. Software Engineering: A Practitioner’s Approach. – 8th ed. – McGraw-Hill, 2014. – 864 p.

24. Wiegers K. E., Beatty J. Software Requirements. – 3rd ed. – Microsoft Press, 2013. – 544 p.

25. ISO/IEC/IEEE 29148:2018. Systems and software engineering – Life cycle processes – Requirements engineering. – Geneva: ISO, 2018. – 86 p.

26. Chernak Y. Requirements-Based Testing: What to Test and How to Make Sure You Test It. – SEI Interactive, 2005.

27. Wynne M., Hellesoy A., Tooke S. The Cucumber Book: Behaviour-Driven Development for Testers and Developers. – 2nd ed. – Pragmatic Bookshelf, 2017. – 344 p.

28. Rex Black. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. – Wiley, 2009. – 576 p.

29. Leffingwell D., Widrig D. Managing Software Requirements: A Use Case Approach. – 2nd ed. – Addison-Wesley, 2003. – 544 p.

30. Utting M., Legeard B. Practical Model-Based Testing: A Tools Approach. – Morgan Kaufmann, 2010. – 456 p.

31. Bures M., Cerny T., Ahmed M. Automatic test case generation: Survey and trends // IEEE Access. – 2021. – Vol. 9. – P. 83131–83152.

32. Fraser G., Arcuri A. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software // Proc. 19th ACM SIGSOFT. – 2011. – P. 416–419.

33. Bauer T., Wickenkamp A. Requirements-Based Test Case Generation: Review and Future Directions // Proc. 9th Int. Conf. Software Testing. – 2022. – P. 15–27.

34. Baudry B., Duchien L., Le Traon Y. Automatic Software Test Generation: A Survey // Journal of Software Testing. – 2019. – Vol. 3, No. 2. – P. 23–42.

35. Utting M., Pretschner A. A Taxonomy of Model-Based Testing Approaches // Software Testing, Verification & Reliability. – 2006. – Vol. 17, No. 2. – P. 61–70.

36. Shah S. Selenium WebDriver Practical Guide. – Packt Publishing, 2012. – 310 p.

37. Karampinis E., Stamelos I. Automatic Test Generation for Software Using Machine Learning Techniques // Journal of Systems and Software. – 2020. – Vol. 165. – P. 110–123.

38. Kuhn D. R., Wallace D. R., Gallo A. Combinatorial Testing for Software: An Adaptation of Design of Experiments // Journal of Research of the National Institute of Standards and Technology. – 2004. – Vol. 109, No. 4. – P. 1–13.