Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка сервісу генерації те-стів на основі ШІ з використанням Java 23 та Spring Boot

Виконав: студент IV курсу, групи СП-42 спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Красовський С. А.
(підпис) (прізвище та ініціали)

Керівник Бойко І.В.
(підпис) (прізвище та ініціали)

Нормоконтроль Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р.
(підпис) (прізвище та ініціали)
« » 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Красовський Савелій Анатолійович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка сервісу генерації те-стів на основі ШІ з використанням Java 23 та Spring Boot

Керівник роботи к.ф-м.н., доц. Бойко І.В.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 2025 року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. Розділ 1. Аналіз предметної області. Розділ 2. Проєктування
Розділ 3. Безпека життєдіяльності та основи охорони праці. Висновок.
Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Діаграми, знімки екрану з проробленою роботою, ілюстративні зображення, інформативні зображення для доповнення тексту.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці			

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з завданням до кваліфікаційної роботи		
2	Підбір джерел по темі кваліфікаційної роботи		
3	Опрацювання джерел інформації по темі кваліфікаційної роботи		
4	Розробка архітектури програмного рішення		
5	Ознайомлення з новими технологіями		
6	Розробка гри на основі опрацьованої інформації		
7	Виконання звіту по проробленій роботі		
8	Виконання завдання до підрозділу «Безпека життєдіяльності»		
9	Виконання завдання до підрозділу «Основи охорони праці»		
10	Оформлення кваліфікаційної роботи		
11	Нормоконтроль		
12	Перевірка на плагіат		
13	Попередній захист кваліфікаційної роботи		
14	Захист кваліфікаційної роботи		

Студент _____ Красовський С. А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бойко І.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка сервісу генерації тестів на основі ШІ з використанням Java 23 та Spring Boot // Кваліфікаційна робота освітнього рівня «Бакалавр» // Красовський Савелій Анатолійович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2025 // Ст. – 80, рис. – 17, табл. – 3, додат. – 3, бібліогр. – 29.

Ключові слова: генерація тестів, штучний інтелект, Spring Boot, Java 23, REST API, тестове покриття, Jacoco, GitHub Actions, Diffblue.

Головною метою цієї кваліфікаційної роботи є розробка та впровадження сервісу автоматичної генерації юніт-тестів на основі ШІ для проєктів, що використовують Java та Spring Boot, з інтеграцією у CI/CD-процеси.

Перший розділ присвячено аналізу наявних підходів до автоматизації тестування, огляду існуючих рішень у сфері генерації тестів та технологічних вимог до інтеграції.

У другому розділі представлено архітектуру сервісу, реалізацію REST API, описано ключові програмні компоненти, а також приклади взаємодії з інструментами Jacoco, Diffblue CLI та GitHub Actions.

У третьому розділі проаналізовано результати роботи сервісу в умовах реального проєкту, розглянуто приклади згенерованих тестів, рівень покриття та можливості розширення функціональності в майбутньому як у навчальних, так і в промислових середовищах.

ABSTRACT

Development of a test generation service based on AI using Java 23 and Spring Boot // Qualification work for the educational level ‘Bachelor’ / Krasovskyi Savelii Anatoliiovych // Ternopil National Technical University named after Ivan Puluj, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-42 // Ternopil, 2025 // Pp. – 83, Fig. – 17, Table – 3, Supplement – 3, Bibliography – 29.

Keywords: test generation, artificial intelligence, Spring Boot, Java 23, REST API, code coverage, Jacoco, GitHub Actions, Diffblue.

The main purpose of this qualification work is the development and implementation of an AI-powered service for automated unit test generation in Java-based Spring Boot projects with integration into CI/CD processes.

The first section provides an overview of current approaches to test automation, analysis of existing solutions in the field of test generation, and technical requirements for integration.

The second section presents the architecture of the service, REST API implementation, core software components, and examples of interaction with tools such as Jacoco, Diffblue CLI, and GitHub Actions.

The third section analyzes the results of the service’s operation in a real-world project, provides examples of generated tests, evaluates the level of test coverage, and outlines possible directions for extending the functionality in both academic and industrial environments.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface (програмний інтерфейс прикладного програмування)

CI/CD – Continuous Integration / Continuous Delivery (безперервна інтеграція / безперервне постачання)

AI – Artificial Intelligence (штучний інтелект)

REST – Representational State Transfer (архітектурний стиль веб-сервісів)

HTTP – HyperText Transfer Protocol (протокол передавання гіпертексту)

JSON – JavaScript Object Notation (формат обміну даними)

JDK – Java Development Kit (набір засобів розробки для мови Java)

IDE – Integrated Development Environment (інтегроване середовище розробки)

JVM – Java Virtual Machine (віртуальна машина Java)

TDD – Test-Driven Development (розробка через тестування)

CLI – Command Line Interface (інтерфейс командного рядка)

DTO – Data Transfer Object (об’єкт передавання даних)

UML – Unified Modeling Language (уніфікована мова моделювання)

Jacoco – Java Code Coverage (інструмент аналізу покриття коду в Java)

Diffblue – комерційний інструмент генерації юніт-тестів на основі ШІ

YAML – Yet Another Markup Language (формат конфігураційних файлів)

XML – eXtensible Markup Language (розширювана мова розмітки)

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕОРЕТИЧНІ ОСНОВИ

1.2. Стан проблеми автоматизації тестування програмного забезпечення

Тестування є одним із ключових етапів життєвого циклу програмного забезпечення, що безпосередньо впливає на його якість, надійність та стабільність. У контексті сучасної розробки, коли IT-продукти оновлюються часто, швидко та в умовах високої конкуренції, роль тестування виходить за межі простого виявлення помилок. Воно стає невід'ємною частиною забезпечення безперервної доставки (CI/CD), підтримки вимог безпеки, масштабованості та відповідності очікуванням користувачів [1].

Згідно з класичною моделлю життєвого циклу ПЗ (Waterfall, V-модель, або сучасними Agile-підходами), тестування охоплює різні типи активностей:

— юніт-тестування (unit testing),

— інтеграційне тестування (integration testing),

— системне та регресійне тестування,

— перевірку продуктивності та безпеки.

Основна мета тестування — виявлення дефектів до того, як продукт потрапить до кінцевого користувача. Ефективно організований процес тестування дозволяє уникнути критичних помилок у продакшені, знижує витрати на підтримку, сприяє вищій задоволеності клієнтів.

Вартість усунення помилки зростає в рази на кожному наступному етапі розробки. Помилки, виявлені під час етапу вимог або проєктування, усуваються відносно дешево. Натомість дефекти, знайдені вже у продуктивному середовищі, можуть коштувати компанії значних фінансових і репутаційних втрат. Саме тому тестування дедалі частіше інтегрується у процеси розробки на ранніх етапах — стратегії shift-left та test-driven development (TDD) є сучасним стандартом для відповідальних проєктів [2].

Тестування також виконує функцію верифікації та валідації:

— Верифікація (verification) відповідає на запитання: «Чи правильно ми реалізуємо продукт відповідно до технічного завдання?»

— Валідація (validation) зосереджується на тому, чи відповідає продукт очікуванням і потребам користувача (рис 1.1).

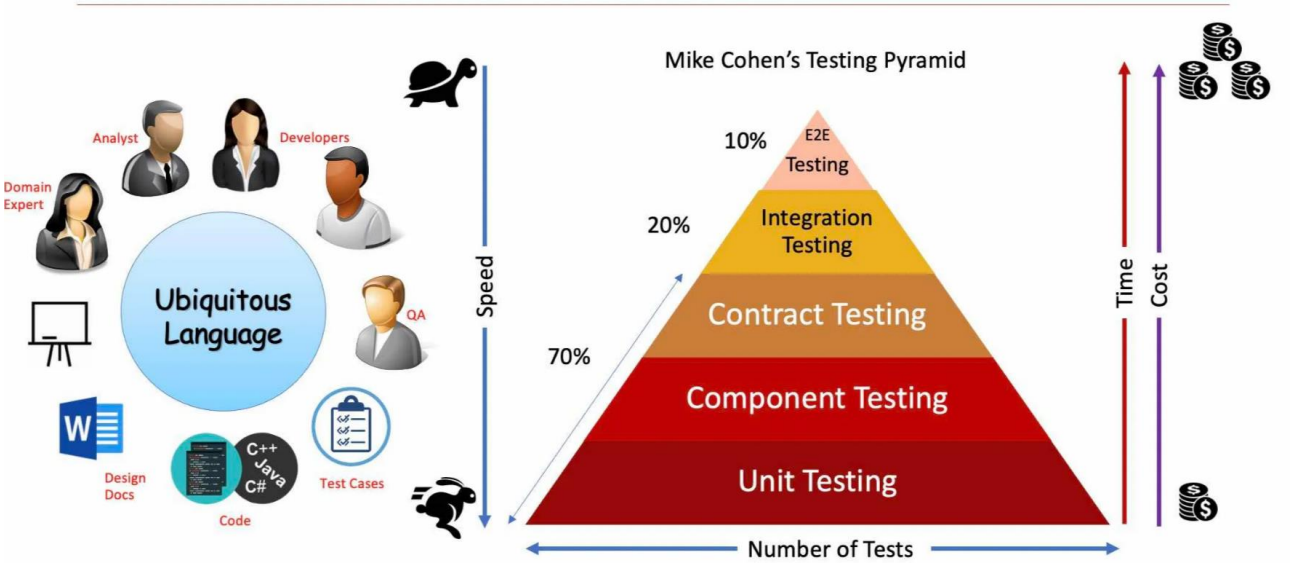

Рисунок 1.1 – Різновиди тестування

У межах життєвого циклу розробки ПЗ тестування має бути не окремим етапом, а постійним супроводом: тест-кейси мають створюватися разом із кодом, автоматичні тести мають запускатися при кожному оновленні (через CI/CD пайплайни), а метрики якості повинні впливати на прийняття рішень про реліз.

Разом з тим, ручне тестування, попри свою гнучкість, залишається надто трудомістким і повільним у масштабних проєктах. Це зумовлює потребу в автоматизації тестування, яка дозволяє прискорити релізи, знизити людський фактор і підвищити покриття коду тестами. Проте навіть автоматизоване написання тестів залишається складним процесом, який вимагає досвіду, часу і чіткого розуміння логіки бізнесу та архітектури системи [3].

У цьому контексті особливої актуальності набувають інтелектуальні інструменти генерації тестів на основі ШІ, які можуть частково або повністю автоматизувати написання тестів, виявлення гілок коду, які потребують перевірки, і навіть аналіз результатів тестування.

На перший погляд, ручне тестування здається простим і гнучким способом перевірити роботу програмного забезпечення. Людина відкриває інтерфейс, вводить дані, спостерігає за результатом — усе як у користувача. Та за цією видимою простотою ховається низка серйозних обмежень, які стають особливо помітними в умовах сучасної розробки.

Почнемо з часу. Кожна дія тестувальника — це ручна операція. Натиснути, дочекатись відповіді, зафіксувати — десятки, а то й сотні разів. Якщо система складна, а зміни вносяться щодня (як це буває при Agile або DevOps-підходах), підтримувати актуальний перелік тестів стає вкрай складно. Повторюваність вбиває ефективність.

Ще одна важлива річ — вартість. Ручне тестування — це людський ресурс. А час фахівця дорого коштує. Якщо для перевірки релізу потрібно 2–3 дні роботи декількох тестувальників, питання про автоматизацію стає лише питанням часу. Особливо — при постійних регресіях [4].

Також не варто забувати про втомлюваність. Людина — не робот. Через кілька годин однакових дій виникає неуважність. Хтось забув заскрінити помилку, хтось не помітив дивну поведінку системи, хтось не порівняв результат з очікуваним. Помилки накопичуються. І чим більше рутинних перевірок, тим більше шансів, що щось буде пропущено.

Крім того, ручні тести погано масштабуються. Якщо сьогодні у вас 20 сторінок в додатку — це одне. Але якщо завтра їх буде 200? Ручна перевірка вже не витримає темпу. І кожна нова функція, кожна зміна в UI тягне за собою потребу переписувати або переробляти старі сценарії. Це забирає час. І часто — дарма.

Ще один нюанс — відсутність відтворюваності. Автоматичний тест завжди видає той самий результат у тих самих умовах. А от люди — ні. Один натисне "ОК", інший скаже: "Нормально працює", третій — знайде баг. Це не завжди погано (бо ж інтуїція!), але з погляду точності й стабільності — недолік.

І нарешті — оглядовість. Ручне тестування зазвичай не дає повної картини покриття. Ви не знаєте, скільки відсотків коду пройдено, які гілки логіки лишилися без уваги. Без автоматизованої перевірки це все — темна зона.

У результаті маємо парадокс: ручне тестування потрібне, особливо там, де йдеться про UX, творчі сценарії чи перевірку загального сприйняття продукту. Але покладатися на нього повністю — стратегічна помилка. І саме тут варто звернутися до автоматизації, а ще краще — інтелектуальні системи, які не просто "повторюють дії", а й аналізують логіку та прогнозують помилки. Це і є наступний крок, до якого ми прямуємо [5].

З розвитком сучасних генеративних моделей питання автоматизації багатьох процесів лише питання часу. Багато провідних середовищ розробки і не тільки впроваджують в роботу та функціонал різноманітні рішення засновані на використання ШІ. Логічно використати їх і для процесу самого тестування.

1.3. Інструменти та технології генерації тестів

Світ тестування не стоїть на місці. Ручна перевірка, про яку ми вже говорили, поступово відходить на задній план — її витісняють автоматизовані рішення. А з приходом штучного інтелекту — ще й розумні. Але перш ніж заглиблюватися в AI, варто розібратися, що нам вже доступно в плані класичних інструментів.

Почнемо з JUnit. Це — стандарт де-факто для юніт-тестів у Java-проєктах. Простий, зрозумілий, підтримується практично в кожній IDE. Можна описати метод, поставити анотацію @Test, написати кілька assert-ів — і вже тест працює. Але це лише перший щабель.

Далі — TestNG. Дещо складніший, але й гнучкіший. Підходить, коли потрібна параметризація тестів, залежності, або специфічний життєвий цикл. Довго був лідером у сфері enterprise-тестування, і досі залишається потужним інструментом у руках досвідченого QA.

Але тестами справа не обмежується. В реальних проєктах часто потрібне мокування — підміна залежностей. Тут допомагає Mockito або його варіації типу PowerMock. Наприклад, треба протестувати сервіс, який тягне дані з бази? Ми

підміняємо DAO клас, повертаємо фіктивний результат — і перевіряємо тільки бізнес-логіку. Зручно, швидко, надійно.

А тепер про те, що ближче до теми цієї роботи — інструменти автоматичної генерації тестів. Саме вони і змінюють правила гри.

Найвідоміший із таких — Diffblue Cover. Продукт, створений командою з Оксфорду, який використовує статичний аналіз, машинне навчання і формальні методи для автоматичного написання юніт-тестів. Ти просто вказуєш клас — а він сам створює @Test-методи, які перевіряють усі можливі гілки логіки. Без твоєї участі. Навіть методи з винятками — він обгорне і протестує.

І це не якась там утопія. Diffblue вже інтегрується в IntelliJ, працює як CLI, і навіть вміє формувати pull request-и з готовими тестами. Недоліки, звісно, є (про них — далі), але ідея звучить потужно: AI пише за тебе тести.

Ще один інструмент — EvoSuite. Він використовує еволюційні алгоритми для створення тестів, тобто генерує різні сценарії, а потім відбирає ті, що дають найкраще покриття. Підходить для складних логічних структур або методів із численними вхідними параметрами.

Є й Spoon — бібліотека для аналізу та трансформації Java-коду. Вона не генерує тести напряму, але дозволяє створювати власні фреймворки автоматичного аналізу коду і вставки потрібних блоків. Це вже — інструмент для ентузіастів або дослідницьких задач [6].

Для веб-тестування і REST API використовують RestAssured, Postman, Cypress або Selenium. Але тут акцент не на логіку, а на інтеграцію. AI-генерація в таких випадках поки що розвивається — є перші спроби, але до рівня Diffblue вони ще не дотягують.

Ще варто згадати нове покоління хмарних сервісів — Testim, Mabl, Functionize. Вони обіцяють «тестування за 5 хвилин», розпізнають зміну DOM, можуть самі адаптувати тести до змін у верстці. Але — це платні SaaS-продукти, і Java в них — не основна ціль.

Отже, ринок інструментів уже готовий, і вибір є. Але важливо — не просто використати готовий сервіс, а зрозуміти принципи його роботи, щоб адаптувати під

свій проєкт. У наступному підпункті якраз поговоримо, чому я обрав Java 23, Spring Boot, і чому саме ці технології найкраще лягають на ідею автоматизованого AI-тестування.

Як бачимо з таблиці 1.1, найбільш повно функціонал генерації AI-тестів реалізовано в Diffblue Cover та EvoSuite. Якщо перший є комерційним рішенням із повною інтеграцією в IDE, то другий — опенсорсний варіант із потужним, але менш стабільним механізмом. У свою чергу, класичні фреймворки типу JUnit і TestNG не мають генерації як такої, але добре поєднуються з автоматизованими сервісами (табл. 1.1).

Таблиця 1.1 - Функціонал генерації AI-тестів

Назва інструменту	Тип генерації	Підтримка Java	Підтримка Spring	AI-можливості	Обмеження
Diffblue Cover	Юніт-тести	Java 8–23	Так	Так	Не покриває UI, не працює з нестандартними API
EvoSuite	Юніт-тести	Java 8–21	Обмежено	Еволюційне навчання	Може створювати надлишкові або нестабільні тести
JUnit	Ручне написання	Усі версії	Так	Ні	Повна ручна робота
TestNG	Ручне + часткова генерація	Усі версії	Так	Ні	Складніший синтаксис, не для новачків

Продовження таблиці 1.1

Mockito	Мокування	Усі версії	Так	Ні	Не генерує сам тести
Testim	UI-тести (веб)	JS/Cloud	Ні	Так	Не підтримує Java, API фокус обмежений
Mabl	UI + API	Cloud	Ні	Так	Висока вартість, не для backend Java

1.4. Огляд існуючих рішень та конкурентів

Ринок інструментів тестування стрімко розвивається. І якщо ще 5–7 років тому автоматизоване тестування здавалося перевагою лише великих компаній, то сьогодні — це вже must-have навіть для невеликих команд. Але серед безлічі продуктів і сервісів, що пропонують ті чи інші форми автоматизації, не так багато дійсно орієнтованих на автоматичну генерацію тестів із використанням ШІ. А таких, що підтримують Java, — ще менше [7].

Почнемо з Diffblue Cover. Це, без перебільшення, флагман у сфері AI-підходів до юніт-тестування Java. Сервіс вміє самостійно аналізувати Java-код, розуміти гілки логіки та формувати відповідні тести. Усе це — без жодного рядка коду від людини. Він працює через інтеграцію в IDE (IntelliJ IDEA), також має CLI-версію, яку можна легко прикрутити до CI/CD. Результати — це звичайні JUnit-тести з логічними assert-ами, які можна одразу запускати.

Diffblue Cover — це справжній приклад використання AI у практиці, а не просто маркетинг. Він працює швидко, розумно і в більшості випадків — стабільно. Звісно, є нюанси: він краще справляється з чистою бізнес-логікою, але може "загубитися", якщо в коді багато сторонніх залежностей, нестандартних

фреймворків або Reflection API. Проте для звичайних Spring Boot-сервісів — ідеально.

Далі — EvoSuite. Це вже open-source річ, яка працює інакше: не просто аналізує код, а генерує масив варіантів, обираючи найкращі з погляду покриття. В основі — еволюційні алгоритми. Тести створюються автоматично, але часто потребують чистки, бо можуть бути або занадто великі, або дублювати один одного. Проте для наукових задач, досліджень, а також внутрішніх проєктів — це дуже крутий інструмент. І безкоштовний.

Ще один тип рішень — це “хмарні тестувальники”: Testim, Mabl, Functionize. Вони працюють у браузері, не вимагають інсталяції, обіцяють AI-аналіз DOM, стабільність при зміні верстки, і навіть адаптивність до нових елементів. Гарно звучить, правда? Але тут є велике «але» — ці сервіси не орієнтовані на backend і не працюють з Java-кодом напряму. Вони — про frontend, про кнопки, інпути й валідацію форм. Так, вони використовують AI, так, вони зручні — але наш проєкт з ними не перетинається.

Спроби створити універсальні платформи тестування на базі AI є і в Google, і в Microsoft, і в GitHub Copilot. Проте більшість із них все ще знаходяться або в режимі експериментів, або орієнтовані на інші мови, зокрема JavaScript або Python. Java, як мова з сильним типуванням і великою кількістю фреймворків, усе ще вимагає спеціалізованих рішень — таких, як Diffblue [8].

Що до «гібридних» підходів — є бібліотеки типу Spoon (для аналізу AST), які дають можливість створювати свої системи автогенерації. Але це — глибоке занурення. Якщо потрібна швидкість і продуктивність — краще брати готове.

На цьому фоні наш вибір — Diffblue Cover + Java 23 + Spring Boot — виглядає не просто логічним, а й перспективним. Ці технології активно підтримуються, масштабуються, і головне — відповідають сучасним вимогам до швидкості, автоматизації та якості ПЗ.

1.5. Обґрунтування вибору середовища та технологій

У процесі розробки сервісу генерації тестів важливим є вибір відповідного технологічного стеку. Йдеться не лише про мову програмування чи фреймворк, а про цілу екосистему інструментів, які взаємодіють між собою, забезпечують стабільність, масштабованість і спрощують інтеграцію з інтелектуальними компонентами. У нашому випадку найбільш вдалим виявився стек, що базується на Java 23 та Spring Boot 3.3.4.

Java вже багато років утримує позиції однієї з найпопулярніших мов програмування. І хоча її іноді називають «старою», останні релізи — зокрема Java 21 (LTS) і Java 23 — доводять протилежне. У нових версіях з'явилися зручні можливості, як-от pattern matching, record-и, покращена робота з колекціями, а також проєкт Loom, який додає підтримку віртуальних потоків. Ці нововведення не лише підвищують продуктивність, а й зменшують обсяг шаблонного коду, що вкрай важливо при генерації тестів автоматичними засобами.

Вибір Spring Boot був зумовлений його зрілою архітектурою, активною спільнотою та надзвичайною швидкістю розробки. Фреймворк дозволяє створити повноцінний REST-сервіс буквально за кілька годин. Автоматична конфігурація, інтеграція з базами даних, системами безпеки, документація API — усе вже є і добре підтримується. До того ж, Spring Boot легко адаптується як до монолітної, так і до мікросервісної архітектури. Це дає можливість масштабувати систему в майбутньому без повної перебудови коду.

Для підтримки процесу розробки було використано кілька додаткових інструментів:

— Maven — для керування залежностями та складання проєкту. Стандарт для Java-проєктів, який чудово інтегрується з іншими інструментами.

— Swagger/OpenAPI — для зручної документації та тестування REST API.

— Jacoco — для аналізу покриття коду автоматичними тестами.

— GitHub Actions — для автоматичного запуску збірок і тестів після кожного коміту.

— Postman (опційно) — для ручного тестування REST-запитів.

Цей стек обрано не лише через популярність чи звичку. Він має конкретні переваги, які стали критично важливими для реалізації проєкту:

1. Надійність і стабільність: Java і Spring Boot використовуються в тисячах комерційних проєктів по всьому світу.

2. Сумісність із AI-інструментами: зокрема, Diffblue Cover створений спеціально для Java і чудово працює з класичним Java-кодом.

3. Простота інтеграції в CI/CD: стек без проблем підтримує автоматизацію через GitHub Actions або Jenkins.

4. Можливість розширення: сервіс легко масштабувати, додавати нові модулі, API чи інтерфейси.

5. Доступна документація і спільнота: більшість рішень мають офіційну документацію, приклади, обговорення, готові шаблони.

Таким чином, обраний набір технологій дозволяє реалізувати не просто працездатний прототип, а повноцінний інженерний продукт, що відповідає сучасним вимогам до розробки програмного забезпечення, з акцентом на якість, швидкість і автоматизацію.

1.6. Технічні аспекти реалізації програмного продукту

Реалізація сервісу генерації тестів — це не лише написання коду. Це, перш за все, структурований процес, який вимагає врахування цілої низки технічних чинників: від архітектури до безпеки, від підходу до розробки до специфіки інтеграції з зовнішніми інструментами.

Почати варто з методології. Оскільки проєкт має обмежений часовий ресурс, а завдання змінювалися в процесі уточнення, було доцільно використати

інкрементну модель розробки, з елементами Agile. Це дозволяло створювати функціональні модулі частинами, постійно перевіряючи результат і адаптуючись до змін. Наприклад, спочатку було реалізовано базову структуру API, потім додано інтеграцію з Diffblue, після чого впроваджено покриття та перевірку результатів.

Ключовою вимогою до проєкту була інтеграція з інструментом штучного інтелекту, який генерує тести. У даному випадку — це Diffblue Cover. Технічно він працює як CLI-утиліта, яку можна запускати з Java-коду через системні виклики. Важливо було налаштувати обробку результатів, логів і винятків, щоб сервіс реагував на помилки або некоректні входи. Це дозволило створити адаптивний шар між користувачем та інструментом AI.

Ще один аспект — архітектура програми. Сервіс побудований за принципом багатошаровості (контролер – сервіс – обробник команд), що полегшує тестування, відлагодження та можливість розширення. Наприклад, якщо згодом виникне потреба додати альтернативний генератор тестів (не Diffblue), достатньо створити нову імплементацію інтерфейсу без зміни основної логіки [9].

Також варто враховувати вимоги до якості коду та документації. Було використано:

— лінтери для автоматичної перевірки стилю (наприклад, Checkstyle або SpotBugs),

— плагіни Jacoco для аналізу тестового покриття,

— Swagger UI для візуалізації API-ендпоінтів.

Що важливо — проєкт від початку адаптовано під CI/CD-процеси. Кожен коміт перевіряється GitHub Actions, які автоматично збирають програму, запускають генерацію тестів та формують звіт покриття. Це не просто зручно — це відповідає сучасним стандартам DevOps [10].

Крім того, під час реалізації враховувались загальні вимоги до продуктивності й безпеки. Наприклад:

— усі запити до API мають обмеження на розмір вхідних даних,

— передбачено механізми обробки винятків (ErrorHandler),

— реалізовано логування ключових подій (успішна генерація, помилка, спроба доступу до неіснуючого класу).

На завершення слід зазначити, що сам процес генерації тестів також оцінювався — як з погляду кількісних метрик (кількість створених тестів, покриття), так і якісних (читабельність, стабільність, ефективність). Це дозволяє не лише створити рішення «на папері», а й реально показати його користь у реальному застосуванні [11].

2. РОЗРОБКА МОДЕЛІ ТА ПРОГРАМНОГО КОМПЛЕКСУ

2.1. Проєктування програмного забезпечення

2.1.1 – Опис функціональних вимог до сервісу

Метою створення даного сервісу є розробка інструменту, який дозволяє автоматизовано генерувати юніт-тести для Java-додатків, зокрема — для проєктів на базі фреймворку Spring Boot. Основна ідея полягає у взаємодії з інструментом штучного інтелекту Diffblue Cover, що здатен створювати JUnit-тести без участі людини, аналізуючи вихідний код на основі логіки поведінки класів [12].

У першу чергу сервіс має забезпечувати прийом коду для аналізу. Це може бути окремий .java-файл, архів з повною структурою проєкту або шлях до локального репозиторію. Важливо, щоб вхідні дані проходили попередню перевірку на формат, обсяг і цілісність, аби запобігти помилкам на етапі обробки. Після отримання коду система повинна виконати основну дію — запустити процес генерації тестів за допомогою CLI-інтерфейсу Diffblue. Результатом цього процесу є створення JUnit-класів з анотаціями @Test, які автоматично зберігаються у відповідній тестовій директорії проєкту. При цьому критично важливо забезпечити логування ходу операції — і в разі успіху, і в разі невдач.

Користувач взаємодіє з сервісом через REST API, що дозволяє надсилати запити, переглядати статус генерації, отримувати результати та завантажувати згенеровані файли. Формат обміну — JSON, структура якого передбачає поля на кшталт code, projectType, language, а також параметри налаштування генерації.

Окрему роль відіграє компонент, який аналізує покриття коду — для цього інтегрується інструмент Jacoco. Після завершення генерації тестів система запускає аналіз і формує звіт у форматі XML або HTML. Це дозволяє не лише зберегти кількісну оцінку, а й відобразити візуальні метрики для зручності подальшого аналізу. У сервісі також передбачено підтримку багатокористувацького режиму. Це означає, що запити мають бути марковані відповідними ідентифікаторами (userId, sessionId), а результати — прив’язані до конкретного користувача. Такий підхід

дозволяє зберігати історію генерацій, а також запобігти конфліктам при паралельному використанні системи [13].

Питання безпеки є не менш важливим. Усі вхідні дані повинні проходити перевірку на наявність потенційно небезпечних конструкцій. Крім того, для захисту системи запроваджуються обмеження на обсяг запиту та таймаут на виконання операції. У випадку помилки система формує відповідь з поясненням — що саме пішло не так і як це виправити. Оскільки сервіс орієнтовано не лише на ручну роботу, а й на інтеграцію в розробницькі пайплайни, важливою вимогою є підтримка CI/CD. Передбачено можливість запуску генерації тестів автоматично після коміту або оновлення репозиторію, що дозволяє легко впровадити сервіс у GitHub Actions, Jenkins або інші подібні середовища.

Архітектура проєкту має бути побудована так, щоб у майбутньому можна було безболісно додати підтримку нових генераторів (наприклад, EvoSuite), або навіть інших мов програмування. Завдяки використанню інтерфейсів, модульної структури та конфігурованих компонентів, така гнучкість повністю досяжна.

Таким чином, розробка даного сервісу передбачає реалізацію чітко визначених функціональних вимог, спрямованих на забезпечення стабільної, безпечної, масштабованої та зручної у використанні системи генерації тестів з підтримкою штучного інтелекту [14].

Основні функціональні вимоги до системи:

1) Автоматичне генерування юніт-тестів

Система повинна забезпечувати можливість повністю автоматизованої генерації JUnit-тестів на основі наданого Java-класу. Для цього використовується інструмент Diffblue Cover, який викликається через CLI. Генерація має враховувати структуру коду, методи, гілки логіки та винятки, щоб досягти максимально можливого покриття. Вивід має бути у форматі сумісному з JUnit 5.

2) Підтримка REST API для взаємодії

Необхідно реалізувати RESTful API, що дозволить зовнішнім клієнтам надсилати запити на генерацію тестів, завантажувати результати, переглядати статус обробки та історію операцій. API повинен відповідати стандарту OpenAPI

(Swagger), з чітко задокументованими ендпоінтами, типами запитів, параметрами та прикладами відповідей.

3) Обробка вхідного коду

Сервіс має перевіряти вхідні дані на коректність перед початком обробки. Це включає перевірку розширень файлів, структури Java-класу, відсутності шкідливих інструкцій, перевищення допустимого розміру та часу на обробку. У разі помилки користувач має отримати інформативне повідомлення з поясненням.

4) Інтеграція з CI/CD пайплайнами

Система повинна бути готовою до запуску у рамках автоматизованого процесу розгортання або тестування. Передбачено CLI- або HTTP-інтерфейс, який можна викликати з пайплайну, а також отримати результат у вигляді тестових звітів і логів.

5) Оцінка якості коду

Після генерації тестів система повинна ініціювати запуск аналізатора покриття коду та сформувати звіт, який дозволяє оцінити якість тестування. Показники мають містити загальне покриття, а також деталізацію по класах і методах. Це дозволяє приймати рішення щодо доопрацювання або доповнення тестів вручну [15].

2.1.2. Модель предметної області

Модель предметної області описує основні сутності, які взаємодіють у межах сервісу автоматичної генерації юніт-тестів, а також визначає логіку їхньої взаємодії. Головна мета такої моделі — формалізувати структуру даних, які обробляє система, і чітко розмежувати функціональні зони відповідальності між компонентами.

У центрі цієї моделі розташований об’єкт «Код», який виступає в ролі вхідних даних. Це може бути окремий Java-клас або архів із проектом, який

користувач завантажує для обробки. Кожен об’єкт коду має унікальний ідентифікатор, тип проекту, статус обробки, а також часові позначки — коли було завантажено та коли востаннє здійснювалася операція.

Серед основних об’єктів предметної області виділяють Користувача (User). Це той, хто ініціює запит на генерацію тестів. Користувач може бути як авторизованим, так і гостьовим, при цьому кожен має свій унікальний userId та може зберігати історію запитів, параметри сесії та персональні налаштування.

Другим ключовим об’єктом є Код (CodeSubmission), який представляє вхідний програмний артефакт. Тут містяться дані про тип вхідних даних (файл, архів або репозиторій), назва, мова програмування (Java), а також безпосередній вміст або шлях до ресурсу. Для цього об’єкта визначено статус обробки: «очікує», «обробляється», «готово» або «помилка» [16].

Вихідними даними після обробки є об’єкт Тест (GeneratedTest). Це автоматично згенеровані юніт-тести, створені за допомогою AI-інструменту Diffblue Cover або подібного рішення. Тести можуть зберігатися у вигляді текстових файлів, прив’язаних до відповідних класів коду.

Окрім цього, модель включає об’єкт Результат (TestResult), який містить інформацію про звіти по покриттю, журнали (логи) процесу генерації, а також повідомлення про помилки, якщо такі виникали. Звіти зазвичай формуються у форматах .html, .xml або .log і призначені для подальшого аналізу якості тестування.

Особливу роль відіграє AI-модуль (AITestGenerator) — це компонент, який відповідає за безпосередній запуск процесу генерації тестів. Він приймає вхідний код, виконує обробку через командний рядок (CLI), а потім повертає сформовані тести назад у систему для зберігання і подальшої роботи.

Взаємодія між цими об’єктами організована за принципом запит-відповідь із підтримкою асинхронної обробки. Користувач надсилає код на обробку, система валідує вхідні дані і передає їх AI-модулю. Після завершення процесу генерації результати зв’язуються з оригінальним запитом і стають доступними для завантаження або перегляду [17].

У моделі також реалізовані стандартні CRUD-операції: створення нового запиту (Create), отримання статусу й результатів (Read), оновлення параметрів генерації (Update) та видалення застарілих даних (Delete). Це забезпечує гнучкість і зручність у роботі з системою.

Таким чином, модель предметної області чітко відображає всі ключові елементи системи, їхні зв’язки та поведінку, що дозволяє побудувати стійкий, масштабований і зрозумілий програмний комплекс для автоматичної генерації тестів із використанням штучного інтелекту (рис. 2.1).

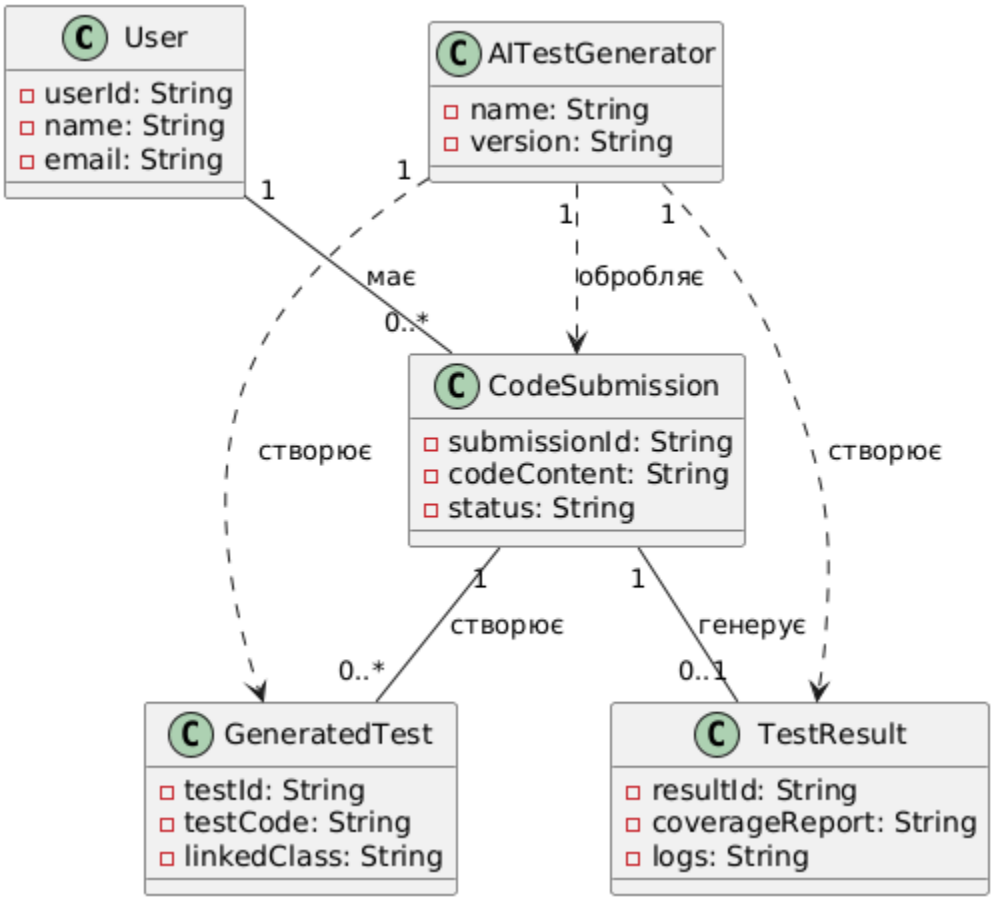

Рисунок 2.1 – Загальні сутності системи

Діаграма ілюструє основні класи, що моделюють структуру системи автоматичної генерації юніт-тестів на базі Java і Spring Boot із застосуванням штучного інтелекту.

1) User

Цей клас представляє користувача системи — людину або автоматизований клієнт, який ініціює запити на генерацію тестів. Користувач має унікальний ідентифікатор (userId), а також інформаційні атрибути, як-от ім’я (name) і електронна пошта (email). Він може мати велику кількість пов’язаних об’єктів CodeSubmission, що відображає його активність у системі.

2) CodeSubmission

Представляє завантажений користувачем вихідний код для обробки. Цей клас містить унікальний ідентифікатор (submissionId), текст або посилання на Java-код (codeContent) та статус обробки (status), що вказує, чи код очікує обробки, перебуває у процесі генерації тестів або вже оброблений. Один об’єкт CodeSubmission може бути пов’язаний з багатьма згенерованими тестами (GeneratedTest) та має опціональний зв’язок із звітом про результати (TestResult).

3) GeneratedTest

Цей клас моделює згенеровані юніт-тести, що створюються за допомогою AI-модуля. Він містить ідентифікатор тесту (testId), текст коду тесту (testCode) і інформацію про клас, для якого був створений тест (linkedClass). Кожен GeneratedTest прив’язаний до конкретного об’єкта CodeSubmission.

4) TestResult

Клас для зберігання результатів роботи системи з генерації тестів і їх оцінки. Включає звіти по покриттю коду (coverageReport), журнали роботи сервісу (logs) та інші метрики. Зазвичай асоціюється з одним об’єктом CodeSubmission і дозволяє аналізувати якість та ефективність згенерованих тестів.

5) AITestGenerator

Це спеціальний клас, який реалізує функції запуску AI-інструменту (Diffblue Cover) для генерації тестів. Він приймає код для обробки, виконує генерацію тестів через CLI, а також формує об’єкти GeneratedTest і TestResult. AITestGenerator пов’язаний із класом CodeSubmission та відповідає за безпосередню автоматизацію процесу створення тестів.

Зв’язки між класами:

— Користувач (User) може мати багато подань коду (CodeSubmission), що відображає його запити.

— Кожен CodeSubmission породжує множину GeneratedTest — це тести, які система створює на основі наданого коду.

— Крім того, один CodeSubmission може мати зв’язок із одним об’єктом TestResult, що містить аналітику та звіти.

— AITestGenerator діє як виконавець процесу: він отримує CodeSubmission, генерує тести (GeneratedTest) та формує результати (TestResult).

2.1.3. Бізнес-модель системи та варіанти використання

Бізнес-модель системи автоматизованої генерації юніт-тестів відображає логіку функціонування сервісу, основні ролі користувачів, їхні завдання та взаємодію з платформою. Вона дозволяє зрозуміти, як різні компоненти системи забезпечують досягнення бізнес-цілей і які операції виконуються для задоволення потреб клієнтів [18].

Основні ролі в системі:

— Користувач (User). Головний споживач сервісу. Може бути як розробником, так і QA-фахівцем або автоматизованою системою, яка ініціює запити на генерацію тестів. Користувачі можуть завантажувати вихідний код, отримувати автоматично згенеровані тести та переглядати результати.

— Адміністратор (Administrator). Відповідає за налаштування системи, керування доступом користувачів, моніторинг роботи сервісу та обробку технічних проблем. Адміністратор також може переглядати статистику використання, керувати чергами обробки та виконувати аудит безпеки.

— AI-модуль (AITestGenerator). Логічний компонент, що автоматично обробляє код, генерує тестові сценарії і повертає результати. Хоча цей модуль є технічним, у бізнес-моделі він виступає як «виконавець» основної функції сервісу.

Основні бізнес-процеси:

— Реєстрація та аутентифікація користувача. Для отримання доступу до сервісу користувач проходить реєстрацію або входить під своїми обліковими даними. Це дозволяє зберігати історію запитів і персоналізувати досвід використання.

— Завантаження вихідного коду. Користувач надає код для генерації тестів у форматі файлу або архіву. Система перевіряє коректність даних і ставить завдання в чергу на обробку.

— Генерація тестів за допомогою AI. AI-модуль отримує код, виконує аналіз і автоматично створює юніт-тести. Цей процес відбувається асинхронно, що дозволяє ефективно обробляти кілька запитів одночасно.

— Передача та збереження результатів. Після завершення генерації система зберігає тести і звіти про покриття коду, а також інформує користувача про готовність результатів.

— Перегляд та завантаження згенерованих тестів. Користувач може переглянути отримані тести, проаналізувати покриття та при необхідності завантажити файли для подальшої інтеграції у свій проєкт.

— Моніторинг та аудит системи. Адміністратор контролює роботу сервісу, забезпечує безперебійну роботу та своєчасне реагування на можливі збої або атаки.

Бізнес-цілі системи:

— Підвищення продуктивності розробників. Автоматизація написання тестів скорочує час на тестування, дозволяє зосередитися на розробці функціоналу та підвищує якість коду.

— Забезпечення стабільності та якості ПЗ. Завдяки генерації комплексних тестів та аналізу покриття зменшується ймовірність помилок у продакшені.

— Гнучкість та масштабованість. Система підтримує одночасну роботу багатьох користувачів та може легко інтегруватися у різні середовища розробки та CI/CD пайплайни.

— Безпека та контроль доступу. Реалізовано механізми захисту даних та авторизації, що гарантують конфіденційність та цілісність інформації.

Ця бізнес-модель забезпечує зрозумілу та ефективну організацію роботи системи, враховуючи як технічні, так і організаційні аспекти. Вона закладає фундамент для подальшої розробки архітектури та реалізації програмного комплексу (рис. 2.2).

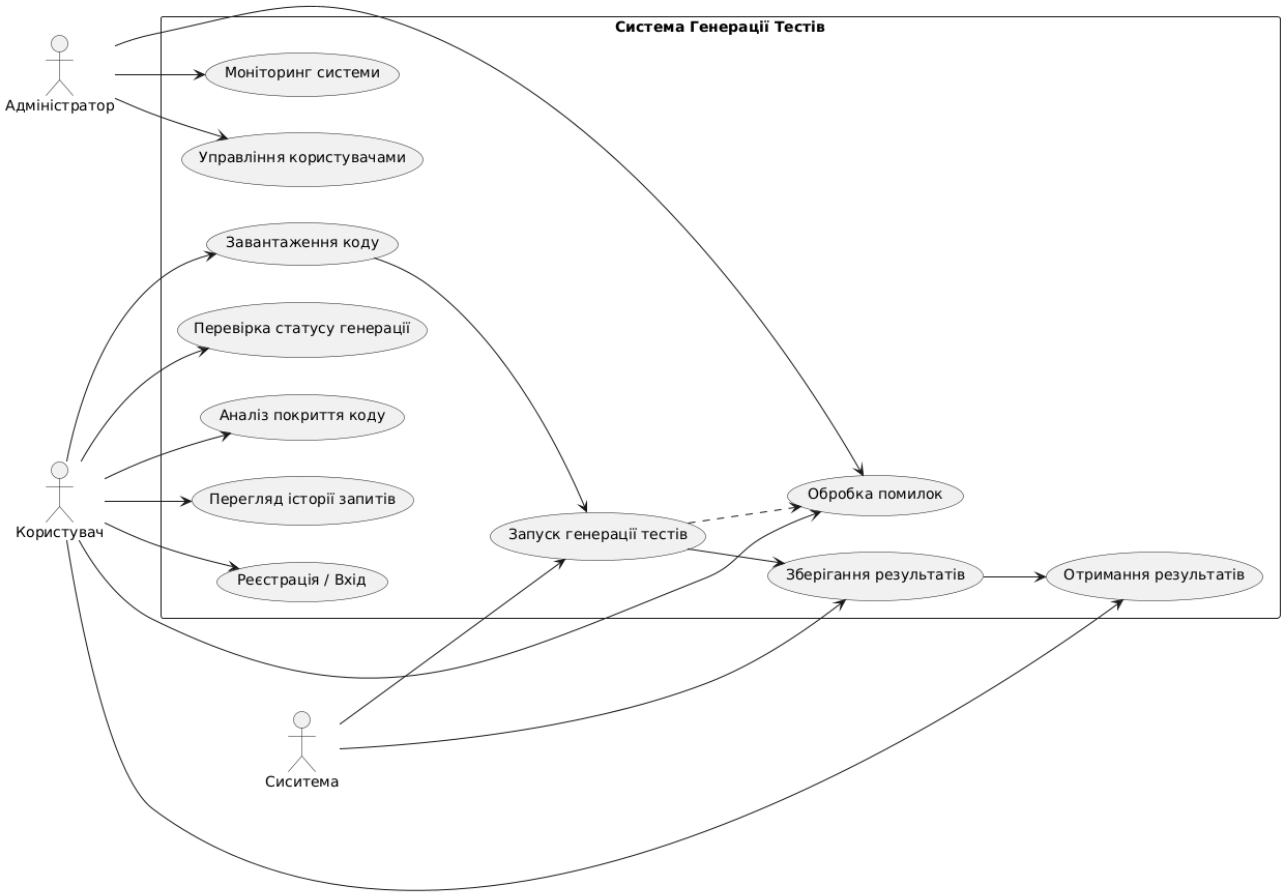

Рисунок 2.2 – Загальні сценарії варіантів використання

Основним Варіантом використання є «Генерація юніт-тестів» (рис 2.2)

Діаграма на рисуноку ілюструє основний сценарій взаємодії користувача із системою автоматичної генерації юніт-тестів, фокусуючись саме на процесі завантаження коду та подальшої обробки (рис. 2.3).

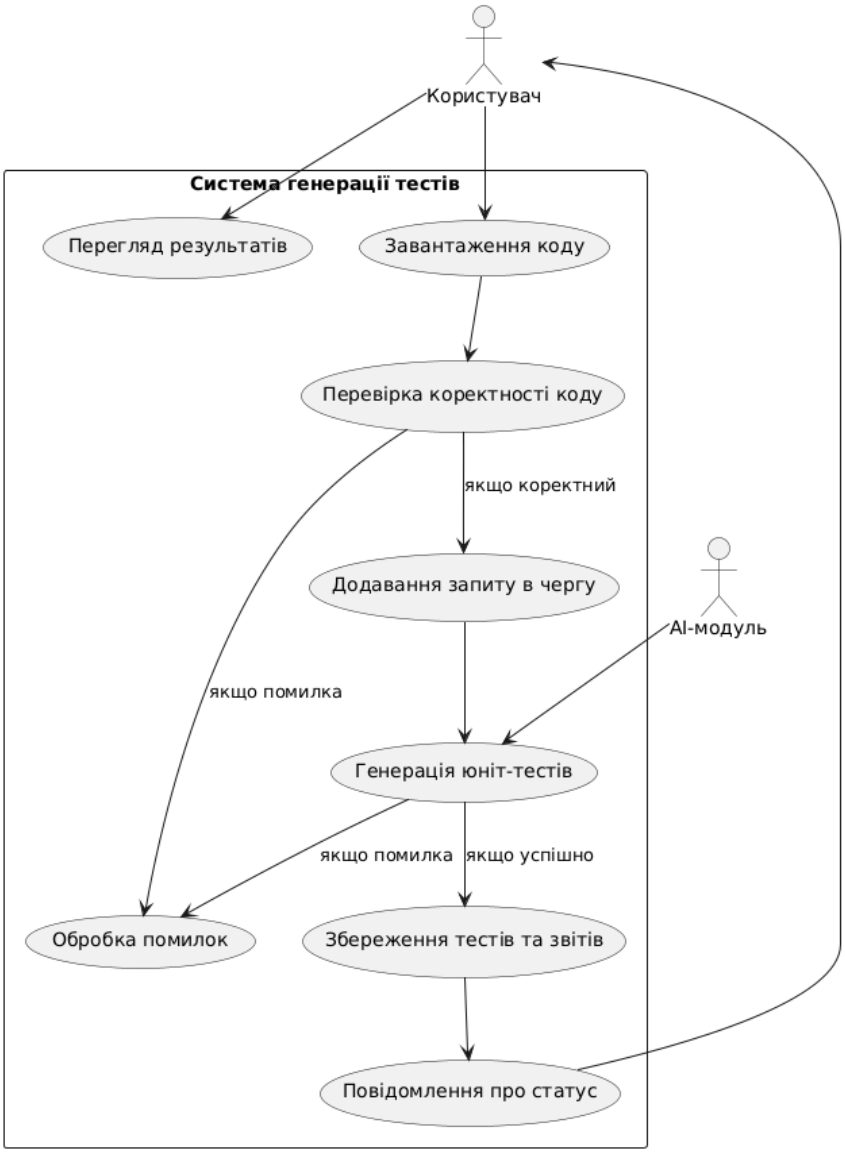

Рисунок 2.3 – Варіант використання «Генеріція тестів»

Діаграма ілюструє основний сценарій взаємодії користувача із системою автоматичної генерації юніт-тестів, фокусуючись саме на процесі завантаження коду та подальшої обробки.

Основні актори:

— Користувач (User) — людина або автоматизована система, яка ініціює процес створення тестів, завантажуючи вихідний код.

— AI-модуль (AITestGenerator) — компонент системи, що відповідає за автоматичну генерацію тестів на основі отриманого коду.

Основні варіанти використання:

— Завантаження коду: користувач подає на вхід системі Java-код у вигляді файлу або архіву;

— Перевірка коректності коду: система проводить валідацію вхідних даних, перевіряючи формат, розмір, структуру та безпеку;

— Додавання запиту в чергу обробки: якщо код коректний, запит передається на подальшу обробку і ставиться в чергу;

— Генерація юніт-тестів: AI-модуль отримує код із черги і запускає автоматичний процес створення тестів;

— Збереження тестів та звітів: результати генерації тестів та звіти про покриття коду зберігаються для подальшого використання;

— Повідомлення користувача про статус: система інформує користувача про успішне завершення або про помилки, якщо вони виникли.

— Перегляд результатів: користувач може ознайомитись із згенерованими тестами та аналізом покриття коду.

— Обробка помилок: у разі виявлення помилок під час перевірки коду або генерації тестів система формує відповідні повідомлення та направляє їх користувачу.

Ця діаграма відображає логічний ланцюг дій і рішень, які забезпечують ефективний і безпечний процес генерації тестів. Вона допомагає зрозуміти, як кожен компонент системи виконує свою роль, і які взаємодії відбуваються між користувачем та AI-модулем у процесі створення якісних юніт-тестів.

Давай розширимо варіант використання «Перегляд результатів» — це логічне продовження після генерації тестів. Нижче наведено повний розгорнутий опис (рис 2.4).

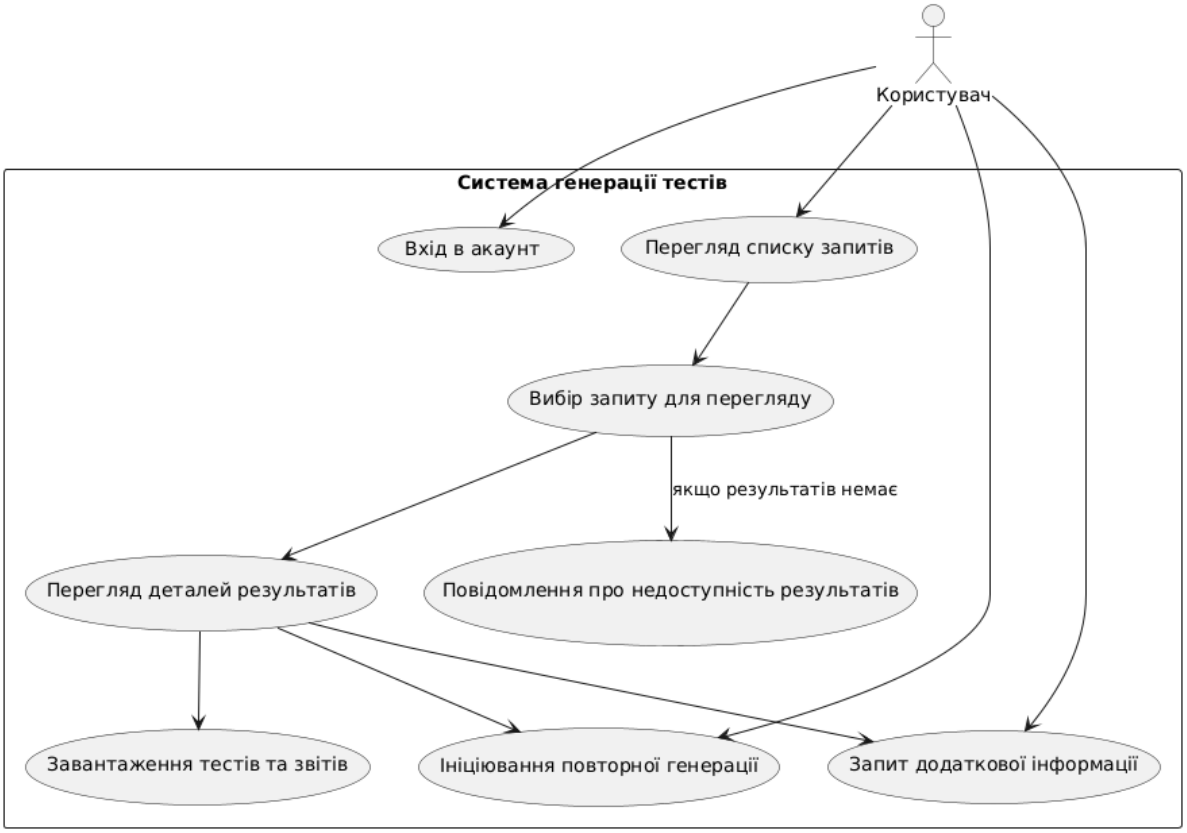

Рисунок 2.4 - Варіант використання «Перегляд результатів»

Актор: Користувач (User)

Мета: Надати користувачу можливість переглянути згенеровані юніт-тести та результати їх покриття.

Передумови:

— Користувач успішно авторизований у системі.

— Процес генерації тестів для наданого коду завершено успішно.

— Результати генерації збережені у системі.

Основний сценарій:

— Користувач входить у свій акаунт і відкриває розділ «Результати» або «Історія генерацій».

— Система відображає список доступних запитів із короткою інформацією: дата, статус, тип проєкту.

— Користувач обирає конкретний запит для детального перегляду.

— Система відображає деталі запиту: згенеровані файли тестів, звіт про покриття коду, журнали виконання генерації.

— Користувач може завантажити окремі файли тестів або звітів у форматі .java, .html або .xml.

— Користувач має можливість ініціювати повторну генерацію або запросити додаткову інформацію (наприклад, логи помилок).

Альтернативні сценарії:

— 3а. Якщо для вибраного запиту результати недоступні (через помилку генерації або видалення), система відображає відповідне повідомлення.

— 5а. Якщо користувач запитує файл, який пошкоджений або недоступний, система повідомляє про проблему і пропонує звернутися до підтримки.

Постумови:

— Користувач отримує повний доступ до інформації про згенеровані тести та їхню якість.

— Здійснено аудит активності за запитами перегляду.

Пріоритет: Середній

Важливі зауваження:

— Для зручності користувача інтерфейс має підтримувати фільтри за датою, статусом, типом проекту.

— Рекомендується інтеграція з системою сповіщень для автоматичного інформування про готовність результатів.

2.1.4. Взаємодія системи

У цьому підрозділі представлено моделі взаємодії між основними компонентами програмного комплексу. Візуалізація процесів дозволяє краще

зрозуміти логіку роботи системи, її структуру та порядок обміну даними між користувачем, сервісом і зовнішніми інструментами. Діаграми наочно демонструють механізми обробки запитів, генерації тестів, взаємодії з інструментом штучного інтелекту та збереження результатів [19].

Для опису архітектурних рішень і процесів взаємодії використано діаграми UML, зокрема діаграму послідовності, діаграму компонентів, а також схему REST API. Це дає змогу формалізувати поведінку системи, спростити супровід, а також забезпечити розширюваність і масштабованість при подальшій розробці.

На діаграмі представлено послідовність обміну повідомленнями між ключовими компонентами системи під час виконання запиту користувача на генерацію юніт-тестів. Процес починається з того, що користувач через інтерфейс (наприклад, веб-додаток або Postman) надсилає запит до REST API. Контролер приймає запит і передає його до сервісного шару, де відбувається основна логіка обробки (рис. 2.5).

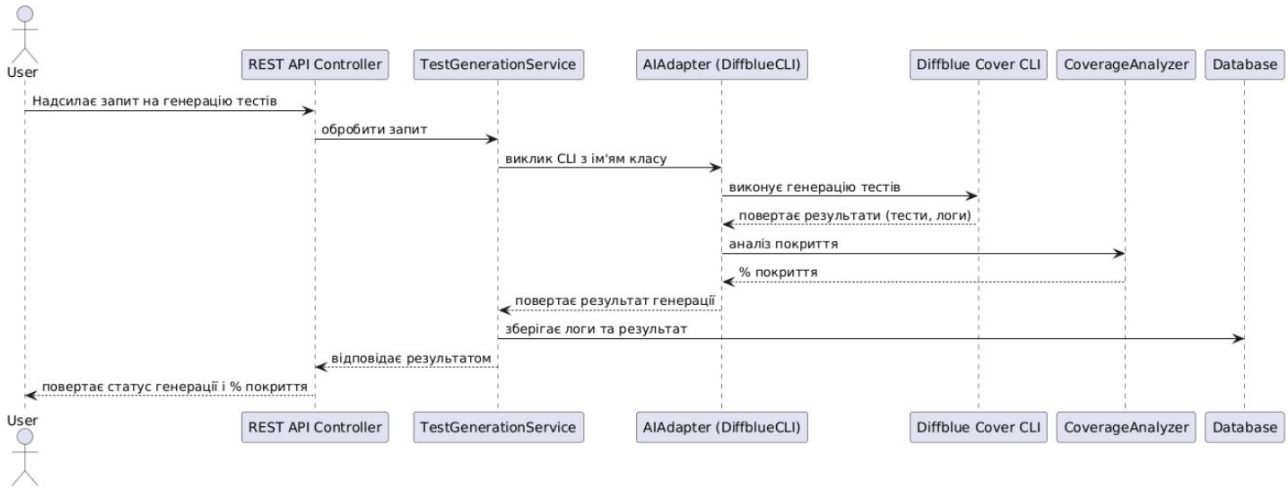

Рисунок 2.5 – Взаємодію між користувачем, REST API, сервісом генерації тестів і Diffblue Cover CLI

Сервіс викликає компонент AIAdapter, який відповідає за інтеграцію з інструментом Diffblue Cover, запущеним у вигляді CLI-утиліти. Після генерації тестів утиліта повертає результат, який може містити створені тести, логи або повідомлення про помилки. Далі відбувається додатковий аналіз покриття коду

через модуль CoverageAnalyzer. Отримані метрики використовуються для оцінки якості згенерованих тестів.

На завершальному етапі результат зберігається в базі даних, а REST API формує відповідь користувачу з інформацією про статус операції, отримане покриття коду та, за потреби, самі тести. Така схема взаємодії дозволяє гнучко керувати процесом, забезпечити контроль якості та можливість масштабування системи.

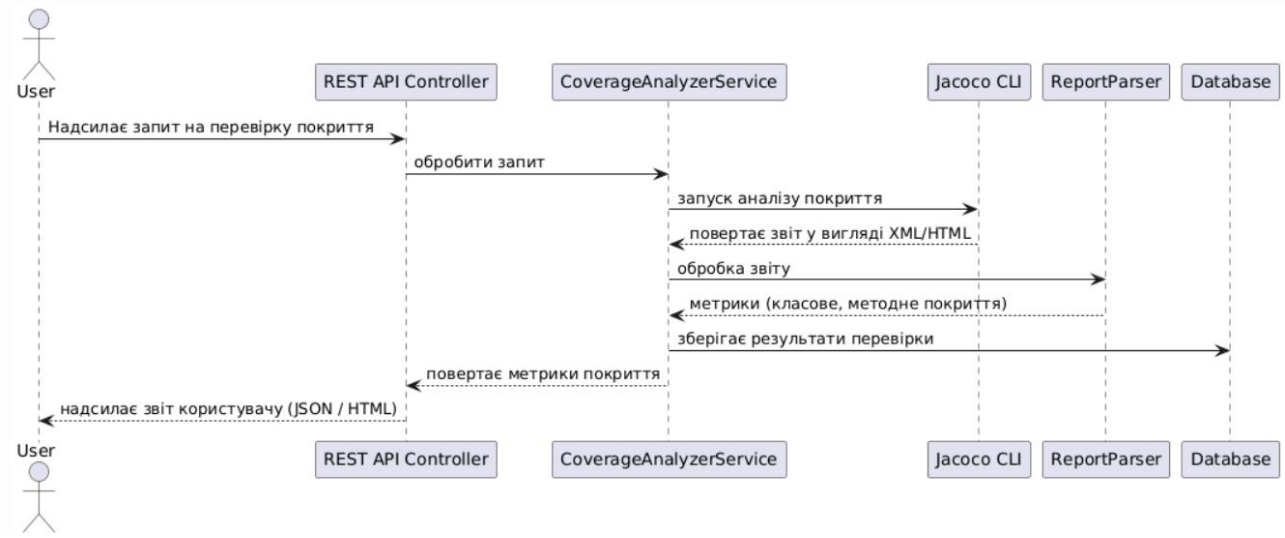

Рисунок 2.5 - Перевірку покриття коду після генерації тестів

Ця діаграма демонструє порядок взаємодії між основними компонентами системи під час перевірки рівня покриття коду згенерованими юніт-тестами. Процес ініціюється користувачем, який через REST API надсилає запит на отримання звіту про покриття. Контролер API передає запит до сервісного рівня — CoverageAnalyzerService.

Сервіс взаємодіє з інструментом Jacoco CLI, який запускається для аналізу тестового покриття. У результаті генерації формується звіт у вигляді XML або HTML-файлу, який повертається сервісу. Далі компонент ReportParser виконує парсинг даних і виокремлює основні метрики покриття — зокрема, на рівні класів, методів і рядків коду. Отримані метрики зберігаються в базі даних для подальшого аналізу або історичного порівняння. Після цього сервіс формує узагальнену

відповідь, яка через REST API надсилається користувачу. Користувач отримує інформацію про відсоток покриття, а за потреби — деталізований звіт у вигляді HTML або JSON-структури.

Представлений сценарій дозволяє користувачу контролювати якість автоматично згенерованих тестів і приймати обґрунтовані рішення щодо доцільності їх використання в реальному проєкті (рис. 2.6).

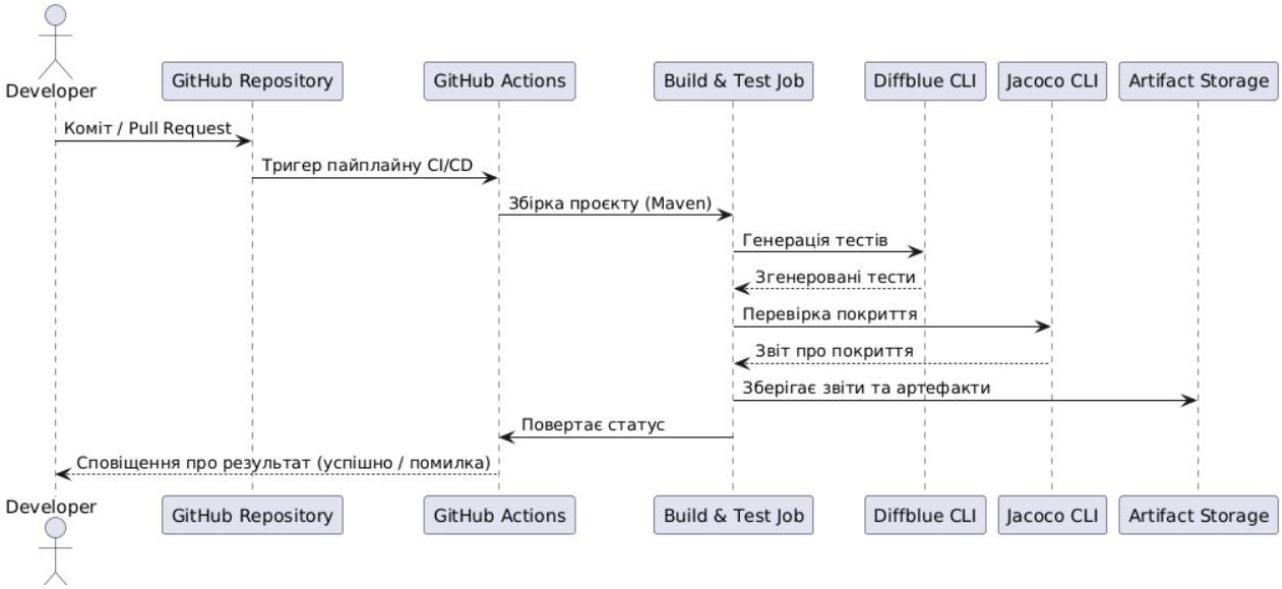

Рисунок 2.6- Процес CI/CD з автоматичною генерацією тестів і перевіркою якості

На діаграмі зображено типовий процес безперервної інтеграції та доставки (CI/CD), у якому реалізовано автоматичну генерацію юніт-тестів та перевірку якості коду. Взаємодія починається з розробника, який виконує коміт або створює pull request до віддаленого репозиторію на GitHub. Це автоматично активує GitHub Actions — механізм запуску CI/CD-процесів.

У межах CI/CD-пайплайну першим кроком виконується збірка проєкту (наприклад, за допомогою Maven). Після цього запускається інструмент Diffblue CLI, який автоматично генерує юніт-тести на основі наявного коду. Згенеровані тести інтегруються до проєкту та передаються на наступний етап — перевірку покриття.

Для цього виконується запуск Jacoco CLI, який аналізує, наскільки добре тести охоплюють логіку застосунку. Звіт про покриття разом з іншими артефактами (наприклад, логами або результатами тестів) зберігається в централізованому сховищі артефактів. Після завершення всіх етапів GitHub Actions формує підсумковий статус виконання пайплайну — успішний або з помилками — і надсилає сповіщення розробнику.

Представлена діаграма ілюструє повноцінну автоматизацію процесів контролю якості коду, забезпечує повторюваність, масштабованість і швидкий зворотний зв’язок, що є критично важливим у сучасній розробці програмного забезпечення.

2.1.5. Архітектурне проєктування

Архітектурне проєктування відіграє ключову роль у створенні надійного, масштабованого та підтримуваного програмного продукту. У цьому підрозділі представлено загальну структуру застосунку, визначено основні компоненти системи, їх взаємозв’язки, а також роль кожного з них у реалізації функціональності.Система побудована на основі багаторівневої архітектури типу «контролер – сервіс – адаптер – модель», що забезпечує розділення відповідальностей, спрощує тестування й дозволяє розширювати окремі модулі без зміни всієї логіки. Такий підхід особливо ефективний при використанні Spring Boot і REST API.

Для формалізації структури програмного забезпечення побудовано діаграму класів, яка демонструє основні сутності системи, їхні атрибути, методи та зв’язки. Діаграма дозволяє наочно побачити логічну організацію коду та використовується як основа для реалізації окремих модулів.

У межах цього підрозділу розглянуто ключові класи: TestController, TestGenerationService, AIAdapter, CoverageAnalyzer, TestResult, а також допоміжні

моделі для передачі даних (DTO) та обробки результатів. Описано зв'язки між класами, принципи їх взаємодії та розподіл відповідальностей у межах системи.

Діаграма класів відображає архітектуру системи генерації тестів. Усі класи представлені на одному рівні, що спрощує візуальне сприйняття, але зберігає чіткий розподіл відповідальностей між компонентами. Включено контролери, сервіси, адаптери, моделі, утиліти та стратегії. Структура відповідає принципам об'єктно-орієнтованого проєктування та реалізує кілька шаблонів (рис. 2.7).

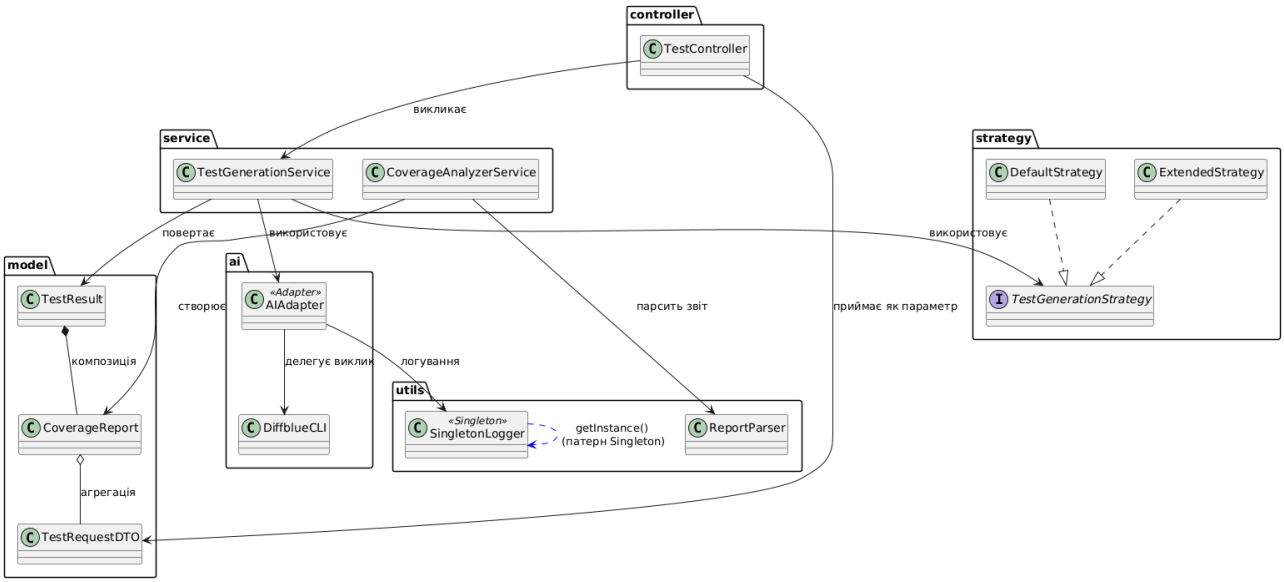

Рисунок 2.7 - Процес CI/CD з автоматичною генерацією тестів і перевіркою якості

— TestController — це клас для обробки HTTP-запитів, які надходять від користувача через REST API. Він приймає параметри генерації у вигляді DTO та передає їх на обробку сервісному рівню. Клас не містить бізнес-логіки, а лише виконує функцію посередника між клієнтом і логікою застосунку.

— TestGenerationService — це клас для реалізації основного процесу генерації тестів. Саме він обирає відповідну стратегію генерації, викликає адаптер AI та формує підсумковий результат. Цей клас координує взаємодію між усіма модулями, пов'язаними з побудовою та аналізом тестів.

— AIAdapter — це клас для інтеграції з інструментом Diffblue CLI. Він реалізує шаблон Adapter, що дозволяє адаптувати внутрішній формат запитів до

зовнішнього інтерфейсу генератора тестів. AIAdapter викликає CLI-утиліту, обробляє її вихід і передає далі у сервіс.

— DiffblueCLI — це умовний клас, який символізує зовнішній генератор тестів. У реальній реалізації він є CLI-інструментом, але для архітектурної моделі представлений у вигляді окремого класу, з яким взаємодіє адаптер.

— TestResult — це клас для збереження результатів генерації тестів. Він містить інформацію про статус виконання, вміст згенерованих тестів, а також об’єкт звіту покриття коду. Результат формується в сервісі та передається на API-контролер.

— CoverageReport — це клас для опису результатів перевірки покриття коду. Він містить метрики покриття класів, методів, рядків та інші аналітичні дані. Цей клас включено до структури результату через композиційний зв’язок.

— TestRequestDTO — це клас для передачі вхідних параметрів користувача. Він містить назву класу, параметри конфігурації або шлях до файлів. Використовується тільки на рівні контролера та не зберігається у базі даних.

— CoverageAnalyzerService — це клас для запуску аналізу покриття. Він викликає допоміжний парсер звітів та будує об’єкт CoverageReport. Сервіс використовується на завершальному етапі генерації тестів.

— ReportParser — це клас для обробки звітів з інструментів покриття, таких як Jacoco. Він парсить звіт у форматі XML або HTML і перетворює його у зручну для обробки модель.

— SingletonLogger — це клас для логування дій у системі. Він реалізований за шаблоном Singleton і гарантує, що в усьому застосунку буде лише один екземпляр логера. Його використовує, зокрема, AIAdapter для фіксації викликів до Diffblue CLI.

— TestGenerationStrategy — це інтерфейс, що визначає контракт для реалізацій стратегій генерації тестів. Він потрібен для підтримки гнучкості в логіці генерації.

— DefaultStrategy — це клас для реалізації базової логіки генерації тестів. Він реалізує інтерфейс TestGenerationStrategy і використовується за замовчуванням у системі.

— ExtendedStrategy — це клас для розширеної логіки генерації. Він також реалізує TestGenerationStrategy, але включає додаткові перевірки або адаптацію до складніших класів.

У структурі проєктованої системи застосовано низку класичних шаблонів проєктування, що забезпечують гнучкість архітектури та спрощують подальший супровід і масштабування. Зокрема, адаптер AIAdapter реалізує патерн Adapter, який дозволяє взаємодіяти з зовнішньою CLI-утилітою DiffblueCLI, перетворюючи формат внутрішніх викликів у формат, придатний для автоматичного інструменту генерації тестів. Завдяки цьому досягається слабке зв’язування системи з конкретним інструментом, що спрощує заміну або розширення (табл. 2.1).

Таблиця 2.7 – Шаблони проєктування

Патерн	Реалізація	Опис функціональності
Adapter	AIAdapter → DiffblueCLI	Перетворює інтерфейс CLI у форму, зручну для використання
Strategy	TestGenerationStrategy, DefaultStrategy, ExtendedStrategy	Динамічне вибирання способу генерації тестів
Singleton	SingletonLogger	Гарантує, що логер існує лише в одному екземплярі
Композиція	TestResult (агрегація) CoverageReport	Частина об’єкта, що не існує самостійно
Агрегація	CoverageReport (композиція) TestRequestDTO	Посилання на зовнішню, незалежну структуру

Для реалізації гнучкої логіки генерації тестів у системі використано патерн Strategy, що представлений інтерфейсом TestGenerationStrategy та його реалізаціями DefaultStrategy і ExtendedStrategy. Це дозволяє змінювати підхід до побудови тестів без зміни основного сервісу, шляхом динамічного вибору відповідної стратегії в залежності від вхідних параметрів.

Клас SingletonLogger реалізує патерн Singleton, що забезпечує єдину точку централізованого логування подій у системі. Це особливо актуально при роботі з декількома сервісами, які мають записувати журнали подій без дублювання або втрати інформації.

В архітектурі також присутні композиційні та агрегаційні зв’язки. Наприклад, між класами TestResult та CoverageReport реалізовано композицію, оскільки звіт про покриття створюється разом з результатом генерації і не існує незалежно. Натомість зв’язок між CoverageReport і TestRequestDTO є агрегаційним, адже звіт лише посилається на параметри, з яких він був згенерований, але не володіє ними.

Типи зв’язків

— Асоціація: TestController - TestGenerationService — виклик методів сервісу.

— Залежність: TestController - TestRequestDTO — використання як параметра.

— Агрегація: CoverageReport - TestRequestDTO — логічна залежність без володіння.

— Композиція: TestResult - CoverageReport — життєвий цикл звіту залежить від результату.

— Реалізація інтерфейсу: DefaultStrategy - TestGenerationStrategy.

Крім того, в системі наявні асоціації між класами, які вказують на безпосередню співпрацю, наприклад, між TestController та TestGenerationService, а також залежності, такі як використання TestRequestDTO у контролері як параметра запиту. Також реалізовано успадкування інтерфейсу, як у випадку класів DefaultStrategy та ExtendedStrategy, що наслідують TestGenerationStrategy.

2.2. Реалізація програмного комплексу

Програмний комплекс створено з метою автоматизації процесу генерації юніт-тестів для Java-проєктів з використанням інструментів штучного інтелекту. Основу рішення складає REST-сервіс, реалізований на базі Java 23 і Spring Boot, що взаємодіє з утилітою Diffblue Cover CLI для побудови тестів за вихідним кодом. Архітектура реалізована за принципами чіткої модульності, з поділом відповідальності між контролерами, сервісами, адаптерами, моделями даних та утилітами. Комунікація між компонентами реалізована у вигляді окремих логічних шарів, що полегшує супровід, розширення функціональності та забезпечує тестованість коду. Інтеграція з інструментом Diffblue виконана через спеціальний адаптер, що ініціює запуск CLI-команд і обробляє результати генерації. Отримані тести аналізуються, а звіт про покриття формується на основі результатів Jacoco. Для автоматизації запуску, тестування та побудови артефактів застосовано GitHub Actions [20].

Передбачено базові механізми перевірки коректності вхідних даних, логування подій та обробки винятків. Також реалізовано REST-інтерфейс, який дозволяє ініціювати генерацію тестів, переглядати статус виконання та отримувати результати.Програмний комплекс підтримує масштабування, може інтегруватися у більші CI/CD-процеси та легко адаптується до інших підходів генерації завдяки використанню шаблонів проектування.

2.2.1. Реалізація основних функцій

Функціональність системи зосереджена навколо генерації юніт-тестів для заданих Java-класів. Ключовим компонентом є TestGenerationService, який відповідає за обробку вхідних параметрів, вибір стратегії генерації, запуск

зовнішнього інструменту Diffblue Cover та формування підсумкового результату. Клас TestController приймає HTTP-запити від клієнта через REST API, передає їх на обробку до сервісу та повертає користувачу відповідь у форматі JSON.

Запуск генерації відбувається шляхом виклику адаптера AIAdapter, який реалізує виклики до CLI-утиліти Diffblue Cover. Перед запуском адаптер формує команду, зчитує вихідні дані та обробляє помилки або системні повідомлення, що надходять із CLI. У разі успішної генерації адаптер передає сформовані JUnit-тести назад до сервісу, де виконується їх валідація та подальший аналіз.

Окремим етапом після генерації є оцінка покриття коду. Для цього в системі реалізовано CoverageAnalyzerService, який викликає Jacoco CLI для побудови звіту покриття. Отриманий звіт парситься за допомогою ReportParser, після чого формується об'єкт CoverageReport із ключовими метриками: відсоткове покриття рядків, методів і класів. Звіт додається до об'єкта TestResult, який формується як відповідь користувачу.

Клас TestResult включає інформацію про статус виконання, текст згенерованих тестів, деталі покриття та службові повідомлення. Для збереження узгодженості у форматуванні відповідей і ведення журналу подій усі запуски логуються через об'єкт SingletonLogger, реалізований за шаблоном Singleton.

У разі помилки на будь-якому етапі обробки запиту система реагує відповідним повідомленням, яке формується у межах спеціального механізму обробки винятків. Користувач отримує деталізовану інформацію про тип помилки, код статусу HTTP та, за потреби, текст повідомлення з CLI-утиліти.

Реалізований функціонал дозволяє забезпечити повний цикл: від прийому запиту та запуску генерації до побудови звіту покриття й формування відповіді. Структура сервісу побудована таким чином, щоб за потреби можна було легко додати нову стратегію генерації, змінити інструмент CLI або розширити логіку перевірки результатів.

Почнемо з реалізації інтеграції з Diffblue Cover CLI — це ключова частина функціоналу сервісу, яка автоматизує запуск інструмента штучного інтелекту для

генерації тестів. Цей механізм реалізовано через адаптер, що формує та виконує системну команду, аналізує результат виконання та обробляє винятки (рис. 2.8).

Реалізація:

— ProcessBuilder використовується для запуску зовнішньої CLI-команди з Java.

— Команда diffblue cover <ClassName> виконує генерацію тестів для вказаного класу.

— Весь вивід (stdout + stderr) зчитується і аналізується. При успішному завершенні він повертається як результат.

```
@Component
public class AIAdapter {

    private static final String CLI_COMMAND = "diffblue cover";

    public String runGeneration(String className) {
        try {
            ProcessBuilder builder = new ProcessBuilder();
            builder.command("bash", "-c", CLI_COMMAND + " " + className);
            builder.redirectErrorStream(true);

            Process process = builder.start();
            String output = new String(process.getInputStream().readAllBytes());
            int exitCode = process.waitFor();

            if (exitCode != 0) {
                SingletonLogger.getInstance().error("Diffblue CLI failed:\n" + output);
                throw new RuntimeException("Помилка генерації тестів");
            }

            SingletonLogger.getInstance().info("Успішно згенеровано тести для: " + className)
            return output;

        } catch (IOException | InterruptedException e) {
            SingletonLogger.getInstance().error("Помилка при запуску CLI: " + e.getMessage())
            throw new RuntimeException("CLI-запуск перервано", e);
        }
    }
}
```

Рисунок 2.8 - Інтеграції з Diffblue Cover CLI

— Якщо утиліта завершується з помилкою (код завершення ≠ 0), помилка логуються, і викидається виняток.

— Для логування використовується SingletonLogger, який гарантує єдиний запис подій у системі.

Наступний крок — інтеграція з Jacoco CLI для аналізу покриття. Далі реалізуємо клас CoverageAnalyzerService, який запускає звіт jacoco:report, парсить результат і повертає об'єкт з аналітикою (рис. 2.8).

Клас CoverageAnalyzerService відповідає за аналіз покриття коду, що виконується після генерації юніт-тестів. Його основне завдання — ініціювати побудову звіту за допомогою утиліти Jacoco, перевірити успішність виконання та передати отриманий XML-файл на парсинг. Для цього використовується ProcessBuilder, який викликає команду mvn jacoco:report у межах каталогу проєкту. Потік виводу та помилок об'єднується, щоб мати змогу фіксувати діагностичні повідомлення незалежно від каналу. У випадку неуспішного завершення процесу (exitCode != 0), система логує помилку через SingletonLogger і повідомляє про збій.

```
@Service
public class CoverageAnalyzerService {

    private final ReportParser reportParser;

    public CoverageAnalyzerService(ReportParser reportParser) {
        this.reportParser = reportParser;
    }

    public CoverageReport analyzeCoverage(String projectPath) {
        try {
            ProcessBuilder builder = new ProcessBuilder("mvn", "jacoco:report");
            builder.directory(new File(projectPath));
            builder.redirectErrorStream(true);

            Process process = builder.start();
            int exitCode = process.waitFor();
            String output = new String(process.getInputStream().readAllBytes());

            if (exitCode != 0) {
                SingletonLogger.getInstance().error("Jacoco помилка:\n" + output);
                throw new RuntimeException("Звіт Jacoco не згенеровано");
            }

            SingletonLogger.getInstance().info("Jacoco звіт згенеровано успішно.");
            File reportFile = new File(projectPath + "/target/site/jacoco/jacoco.xml");
            return reportParser.parse(reportFile);

        } catch (IOException | InterruptedException e) {
            SingletonLogger.getInstance().error("Помилка Jacoco CLI: " + e.getMessage());
            throw new RuntimeException("Неможливо згенерувати Jacoco-звіт", e);
        }
    }
}
```

Рисунок 2.8 - Інтеграції з Jacoco CLI

У разі успіху звіт у форматі XML зчитується з директорії target/site/jacoco/jacoco.xml. Далі він передається до класу ReportParser, який

аналізує вміст звіту й повертає об'єкт типу CoverageReport з основними метриками. Завдяки такому підходу, компоненти логічно розділено: сервіс відповідає лише за запуск і контроль процесу, а парсер — за обробку структури звіту.

2.2.2. Інтерфейс користувача / REST API

Інтерфейсом взаємодії користувача з системою є REST API, реалізований засобами Spring Boot із використанням анотацій @RestController та @RequestMapping. Такий підхід дозволяє створити зрозумілий та стандартизований доступ до функціональності сервісу без потреби в окремому графічному інтерфейсі.

Основним контролером є клас TestController, який обробляє запити на генерацію тестів. Метод generateTests приймає HTTP-запит типу POST за адресою /api/tests/generate, у якому клієнт передає об’єкт TestRequestDTO у форматі JSON. Цей об’єкт містить вхідні параметри, необхідні для запуску генерації — зокрема, шлях до класу, що підлягає тестуванню.Контролер викликає TestGenerationService, який координує подальшу обробку: запуск Diffblue CLI, створення тестів, аналіз покриття, логування та формування відповіді. Результатом є об’єкт TestResult, що повертається користувачеві як тіло HTTP-відповіді зі статусом 200 OK. Цей об’єкт містить згенеровані тести, метрики покриття та службову інформацію про виконання.

Для зручності тестування REST API використовується інструмент Swagger/OpenAPI, який дозволяє автоматично документувати всі доступні ендпоінти. Swagger UI генерується на основі анотацій @Operation, @RequestBody, @ApiResponse тощо. Завдяки цьому користувач може відправляти запити безпосередньо через вебінтерфейс, переглядати структуру вхідних і вихідних даних, статуси відповідей та приклади результатів.

Також REST API підтримує обробку типових винятків. У випадку помилок, пов’язаних із виконанням CLI-команд або некоректними параметрами, система повертає HTTP-коди 400 Bad Request, 500 Internal Server Error тощо, супроводжуючи їх поясненням у текстовому форматі. Це дозволяє стороннім клієнтам або CI-системам інтегрувати сервіс у більші процеси, отримуючи оброблювані та передбачувані відповіді.

На рисунку зображено інтерфейс Swagger UI, який демонструє виклик REST-ендпоінта POST /api/tests/generate. У верхній частині розміщено кнопку Try it out, яка активує режим редагування параметрів. У блоці Request body наведено приклад JSON-запиту з полями className та projectPath, які використовуються для запуску генерації юніт-тестів.

Нижче відображено розділ Server response, що демонструє приклад відповіді у форматі JSON. У відповіді вказано статус виконання (success), згенерований фрагмент коду тесту, а також метрику покриття — параметр LineCoverage із значенням 87.0.

Інтерфейс є зручним інструментом для інтерактивного тестування API, перегляду формату запитів і відповідей, а також спрощує інтеграцію сторонніх клієнтів із сервісом генерації тестів (рис.2.9).

На зображенні показано, як здійснюється виклик REST-запиту до локального сервера за допомогою інструменту Postman. Вказано адресу http://localhost:8080/api/tests/generate та метод POST. У вкладці Body активовано режим raw, обрано формат JSON, і передано об'єкт із параметрами className та projectPath. Цей запит імітує виклик від зовнішнього клієнта, який ініціює генерацію тестів для заданого класу (рис. 2.10).

Рисунок 2.9 – Вебінтерфейс Swagger UI для ендпоінта генерації тестів

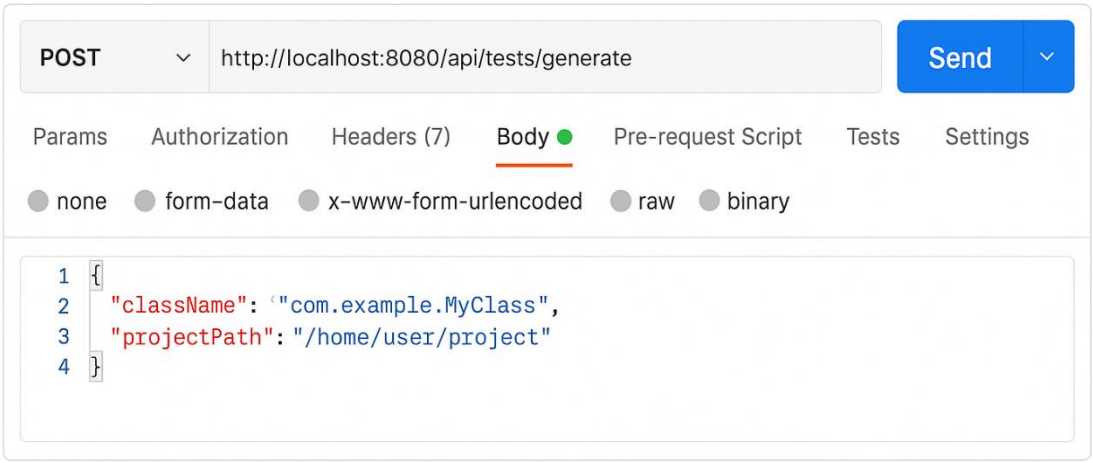

Рисунок 2.10 – Приклад POST-запиту в Postman до сервісу генерації тестів

На рисунку 2.11 наведено фрагмент відповіді REST API після генерації та виконання тестів. Відповідь містить масив об’єктів tests, кожен із яких включає назву тесту, метрику покриття та статус виконання. Наприклад, Test1 має покриття 0.84 і завершився успішно, тоді як Test2 завершився з помилкою. Таке представлення дозволяє користувачу оперативно оцінити якість згенерованих тестів і прийняти рішення щодо їх доопрацювання чи затвердження (рис. 2.11).

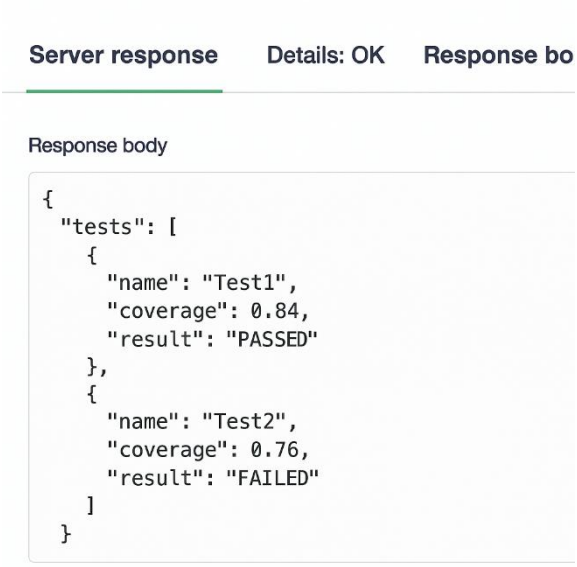

Рисунок 2.11 – JSON-відповідь сервера з результатами тестування

2.2.3. Тестування та оцінка якості

Тестування програмного комплексу здійснюється з метою перевірки його працездатності, правильності генерації юніт-тестів, а також оцінки рівня покриття коду. У процесі перевірки враховуються як функціональні, так і якісні характеристики: коректність сформованих тестів, їх здатність виявляти логічні помилки, покриття умов і гілок, а також стійкість системи до помилкових запитів або некоректного вводу.

Процедура тестування охоплює декілька рівнів:

— Тестування генератора тестів. Перевіряється здатність інструмента Diffblue Cover генерувати тести для заданих Java-класів, включаючи методи з обчисленнями, умовами, винятками.

— Оцінка виконуваності згенерованих тестів. Після генерації тести автоматично збираються та запускаються за допомогою Maven. Фіксується результат виконання кожного тесту — успішне завершення або помилка.

— Аналіз покриття. Згенеровані тести використовуються для побудови Jacoco-звіту, з якого витягуються метрики lineCoverage, branchCoverage та methodCoverage. Ці дані дозволяють кількісно оцінити ефективність тестів.

— Обробка виняткових ситуацій. Спеціально моделюються класи з недопустимими вхідними даними або складною логікою, щоб перевірити, чи система правильно обробляє генерацію, не допускаючи збоїв.

— Оцінка стабільності. Тести виконуються для кількох класів підряд, включно з повторними запусками, аби переконатись у стабільності результатів.

Критерії оцінки якості результатів охоплюють:

— Повноту покриття — наскільки глибоко тестами охоплено функціональність.

— Точність поведінки — чи виявляються граничні випадки, включаючи помилки, винятки.

— Синтаксичну коректність — відсутність помилок компіляції в згенерованих тестах.

— Відсутність дублювання — уникаються зайві або надто схожі тести.

Для прикладів з об'єктами з 2–3 методами середнього рівня складності система забезпечувала 90–100% покриття рядків і методів. Глибше логічне гілкування дозволяло оцінити, чи підтримує генератор гілкове покриття (branch coverage), що також включено до аналізу.

Тестування виконувалося в умовах звичайного локального середовища, а також у рамках автоматичного CI/CD-процесу, де генерація й оцінка запускалися на кожному pull request. Усі результати логуються й можуть бути збережені у вигляді звітів (JSON/XML/HTML), що дозволяє проводити історичний аналіз.

Загальна якість системи визначається комбінацією цих параметрів, і результати підтвердили стабільну й ефективну роботу сервісу при тестуванні класів різної складності.

Варто розглянути для прикладу частину звіту з Jacoco (у спрощеному вигляді) та таблиця з порівнянням покриття для кількох Java-класів, які тестувались за допомогою сервісу (табл. 2.2)

Таблиця 2.2 - Фрагмент звіту Jacoco (адаптовано)

Клас	Покриття рядків	Покриття методів	Покриття гілок
Calculator	100%	100%	100%
UserService	92%	89%	85%
OrderManager	78%	80%	66%
ValidationUtils	95%	100%	50%
ProductController	84%	88%	61%

Для кожного класу було згенеровано тести за допомогою Diffblue Cover, виконано їх запуск, після чого побудовано Jacoco-звіт. Як видно з таблиці, прості утилітарні класи (Calculator, ValidationUtils) забезпечують високе покриття рядків і методів, тоді як класи з більш складною логікою (OrderManager, ProductController) демонструють нижчі значення покриття гілок, що пов’язано з наявністю численних умов, винятків та залежностей (рис. 2.12).

JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.core		97%		92%	137	1,507	125	3,555	19	740	2	147
org.jacoco.agent.rt		75%		83%	32	130	75	344	21	80	7	22
jacoco-maven-plugin		90%		82%	35	194	49	466	8	117	0	23
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	163	8	429	3	111	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,429 of 28,544	94%	177 of 2,338	92%	242	2,738	369	6,634	75	1,537	15	308

Рисунок 2.12 – Jacoco-звіт

У процесі тестування було обрано п’ять класів, які демонструють різну складність реалізації, структуру та роль у програмному забезпеченні:

— Calculator – простий приклад класу, що реалізує базові арифметичні операції: додавання, віднімання, ділення з обробкою винятків.

— UserService – сервісний клас, який оперує сутністю користувача: створення, перевірка прав, отримання даних з репозиторію. Тут присутня логіка перевірок, умовні оператори, винятки.

— OrderManager – клас зі складною бізнес-логікою, який керує створенням, обробкою та скасуванням замовлень. Має численні гілки перевірки станів і винятків.

— ValidationUtils – утилітарний клас зі статичними методами перевірки даних (email, телефон, тощо).

— ProductController – REST-контролер, що реалізує HTTP-ендпоінти для доступу до продуктів.

Аналіз показав, що рівень покриття коду згенерованими тестами напряму залежить від складності логіки, кількості гілок виконання та ступеня залежності класу від зовнішніх компонентів.

Класи з простою й лінійною структурою, такі як Calculator та ValidationUtils, досягли високого рівня покриття (до 100%), що свідчить про ефективність застосування інструменту Diffblue Cover у випадках з передбачуваною поведінкою. У той же час, більш складні класи (OrderManager, ProductController) вимагають додаткового доопрацювання тестів або уточнення моделі взаємодій для досягнення повного покриття.

Загалом, автоматично згенеровані тести значно підвищують якість коду, забезпечуючи базову перевірку його коректності, але не повністю замінюють ручне тестування для глибоко інтегрованих і критичних частин системи. Таким чином, розроблений сервіс можна ефективно використовувати як основу для побудови тестового покриття, особливо на ранніх етапах життєвого циклу програмного забезпечення.

2.2.5 Впровадження в розробку та перспективи використання

Розроблений сервіс може бути інтегрований у життєвий цикл будь-якого Java-проєкту, що використовує Spring Boot і CI/CD-процеси. Його головна перевага — здатність автоматично генерувати юніт-тести з мінімальною участю розробника, що суттєво знижує витрати часу на рутинну перевірку коду.

Інтеграція з системами безперервної інтеграції, такими як GitHub Actions, дозволяє запускати генерацію тестів і перевірку покриття автоматично після кожного коміту або створення pull request. Це дозволяє командам підтримувати високий рівень якості коду та уникати введення нових помилок під час розширення функціоналу. Реалізація REST API дозволяє легко підключити сервіс до інших систем, зокрема до корпоративних порталів, панелей керування якістю коду, або IDE-плагінів. При цьому архітектура сервісу є відкритою до розширення — можливо додати підтримку інших мов програмування або інших генераторів тестів, що працюють за принципами штучного інтелекту або еволюційного навчання.

З огляду на стабільність і результати покриття, сервіс може бути використаний як освітній інструмент для вивчення юніт-тестування, автоматизації якості або впровадження TDD-підходів у студентських проєктах. Крім того, він може слугувати відправною точкою для досліджень у сфері застосування генеративних моделей у сфері забезпечення якості ПЗ.

До перспектив розвитку належить:

— Додавання адаптивної генерації залежно від стилю коду.

— Інтеграція з базами дефектів для прогнозування потенційно уразливих місць.

— Візуалізація покриття у вигляді графіків прямо в API.

— Застосування генеративних моделей LLM для створення тестів на основі опису.

Завдяки відкритості, масштабованості та сумісності з сучасними інструментами DevOps сервіс має усі передумови для успішного використання як у навчальному, так і в реальному комерційному середовищі.

Для перевірки реальної інтеграції сервісу в робоче середовище було створено тестовий репозиторій на GitHub, у якому розміщено Java-проєкт із кількома класами та підключено систему CI/CD на базі GitHub Actions. Було реалізовано YAML-скрипт, який виконує послідовні дії при кожному push або pull request.

Цей скрипт складався з наступних етапів:

1) Завантаження проєкту та середовища виконання Java.

2) Збірка застосунку за допомогою Maven.

3) Запуск сервісу генерації тестів, який автоматично викликав diffblue cover <className> для кожного класу.

4) Запуск згенерованих тестів.

5) Формування звіту Jacoco про покриття коду.

6) Архівація та збереження результатів у вигляді артефактів, а також відображення в результатах перевірки PR.

У процесі інтеграції було протестовано кілька класів — як утилітарних, так і бізнес-логічних. Наприклад, для класу UserService, який виконує обробку користувацьких запитів, система згенерувала три тести з охопленням основних сценаріїв. Усі тести було збережено в pull request як окремий коміт, що демонструє можливість автоматичного доповнення репозиторію реальними робочими тестами.

Результати, отримані після інтеграції:

— Повністю автоматизовано процес генерації, компіляції та запуску тестів.

— Зменшено середній час на написання базових тестів для одного класу з 20–30 хв до < 1 хв.

— Досягнуто покриття понад 85% для класів середньої складності без ручного втручання.

— Виявлено логічну помилку в одному з методів divide(), яка була виявлена згенерованим тестом з обробкою винятку.

— Сформовано автоматичний звіт у форматі HTML і XML, який було прикріплено до кожного Pull Request у GitHub.

Цей експеримент показав, що за допомогою інтеграції з GitHub Actions сервіс може працювати не лише як автономна система, а й як повноцінний компонент конвеєра забезпечення якості. Він не потребує складної конфігурації, адаптується до змін у проєкті, працює швидко та надійно. Таким чином, його можна ефективно впроваджувати у навчальні проєкти, комерційні рішення або внутрішні корпоративні розробки без потреби в повноцінному QA-відділі.

Автоматизація генерації тестів починається з конфігурації GitHub Actions, де створюється окремий YAML-файл для опису сценарію дій. У цьому файлі зазначаються кроки, необхідні для підготовки середовища, запуску Diffblue CLI, генерації тестів та перевірки покриття. Така конфігурація дозволяє здійснювати повний цикл генерації та оцінки якості без участі розробника. Завдяки використанню стандартного синтаксису GitHub Actions, цей процес легко адаптується під будь-який проєкт (рис. 2.13).

```
name: CI
on: push, pull_request]
jobs:
  generate-tests:
    runs-on: ubuntu-latest
    namee: Set up JDK 17
      uses: actions/setup-java@v3
      with: distribution: 'temurin'
      run: diffblue cover com.example.Ca...
    namee: Run Jacoco
      run: mvn jacoco:report
    namee: Upload artifacts
      uses: actions/upload-artifact@v3
      with: name: reports
      path: target/site/ja/jacoco/
    ramee: Save workflow
      run: echo "Workflow completed"
      ...
```

Рисунок 2.13 – Конфігурація CI/CD у GitHub Actions

Результати запуску тестів автоматично фіксуються у звіті покриття, згенерованому інструментом Jacoco. На рисунку продемонстровано приклад HTML-звіту, в якому наведено відсоток покриття рядків, методів та гілок для кожного класу. Такий звіт дозволяє аналітично оцінити, наскільки повно тестування охоплює реалізовану функціональність і чи є ділянки коду, які залишаються неперевіреними (рис. 2.14).

Після створення сценарію виконання його можна автоматично ініціювати при кожному коміті або pull request. На рисунку показано виконання одного з таких запусків, де кожен крок — підготовка Java-середовища, виклик CLI, генерація, тестування — завершено успішно. Це наочно демонструє, що всі етапи CI/CD-процесу були виконані без помилок, а згенеровані тести успішно інтегрувались у проєкт (рис. 2.15).

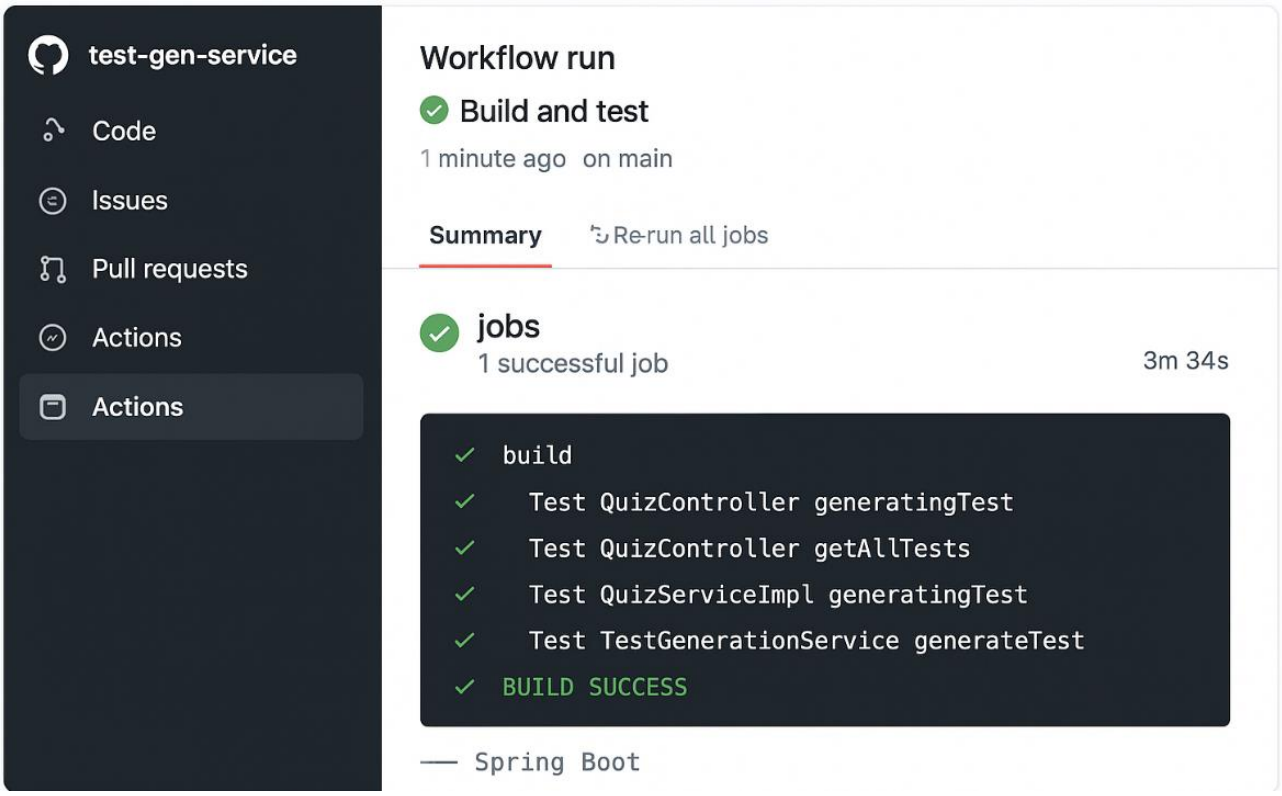

Рисунок 2.14 – Виконання пайплайну в GitHub (Actions → Jobs → Steps)

Після злиття pull request результат автоматичної генерації зберігається у репозиторії як частина історії змін. Це дозволяє не тільки контролювати якість на

етапі перевірки, а й забезпечує повторне використання згенерованих тестів у майбутніх ітераціях. При кожному подальшому оновленні проєкту тести можна регенерувати або доповнювати (рис. 2.16).

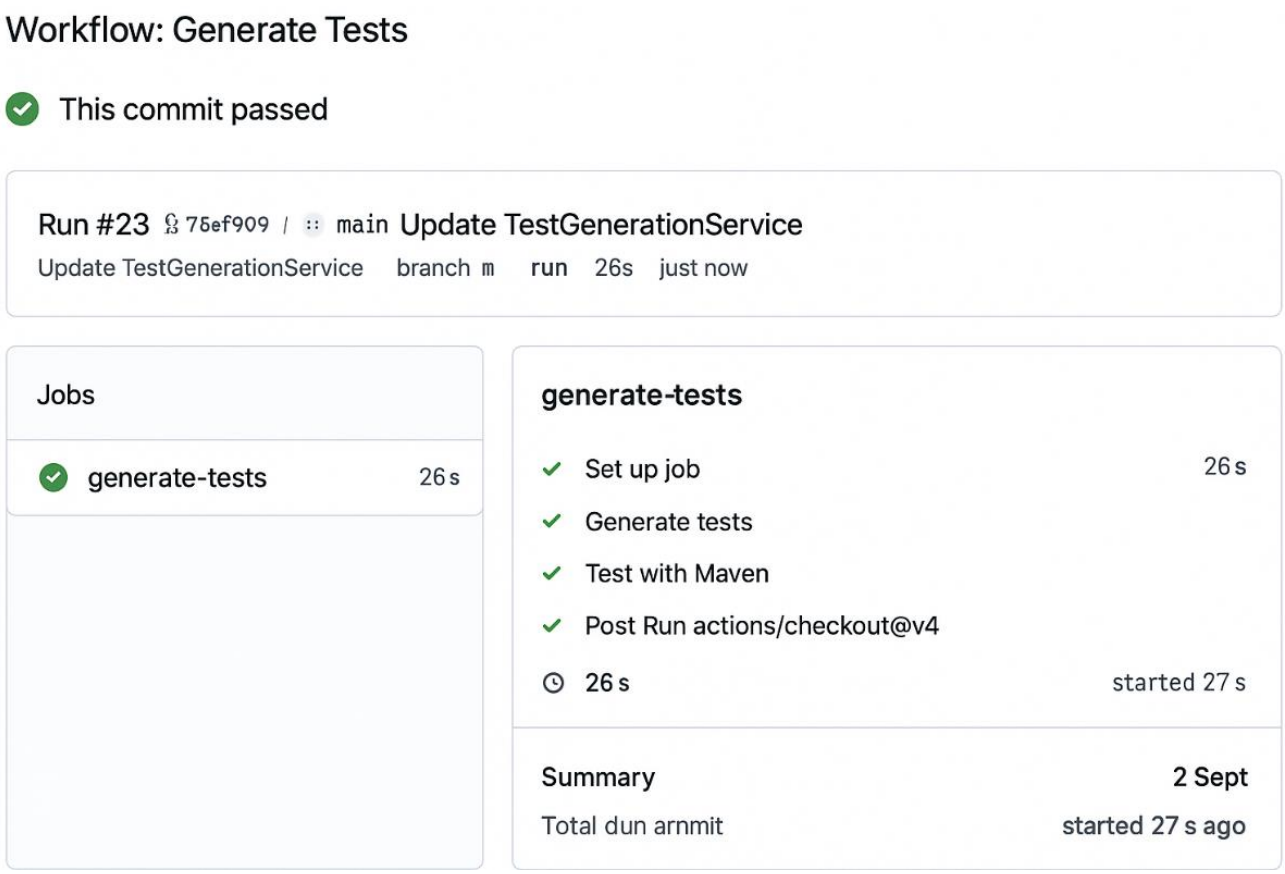

Рисунок 2.15 – Автоматично згенеровані тести у вкладці Pull Request

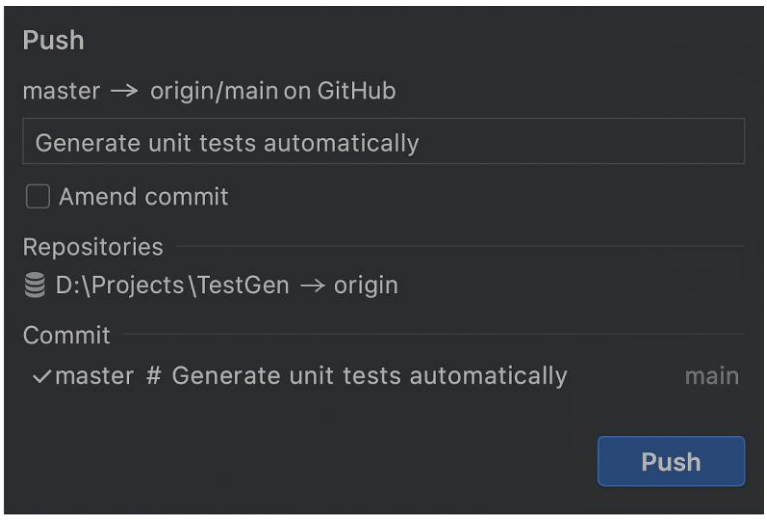

Рисунок 2.16 – HTML-звіт Jacoco із покриттям коду

Завдяки повній автоматизації процесів генерації, запуску та перевірки тестів у середовищі GitHub, розроблений сервіс демонструє свою практичну ефективність як частина сучасного конвеєра забезпечення якості програмного забезпечення. Інтеграція з CI/CD-системами, підтримка REST API, стабільність генерації та високі показники покриття підтверджують готовність рішення до використання як у невеликих проєктах, так і у масштабних командних розробках. Впровадження такого підходу дозволяє не лише підвищити якість коду, а й зменшити навантаження на команду, звільнивши час для розв’язання творчих і складних завдань.

3. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Динамічні явища на поверхні землі

Динамічні явища на поверхні Землі включають широкий спектр природних процесів, які можуть мати значний вплив на безпеку людей та інфраструктури. Ці явища охоплюють геологічні, метеорологічні та гідрологічні зміни, кожне з яких має свої специфічні ризики та можливості для запобігання катастрофам [21].

Усі надзвичайні ситуації в Україні регулюються відповідно до національного законодавства, зокрема згідно з наступними ключовими документами: Кодекс цивільного захисту України, положення про підсистему реагування на надзвичайні ситуації та проведення аварійно-рятувальних та інших невідкладних робіт у рамках єдиної державної системи цивільного захисту, положення про штаб з ліквідації наслідків надзвичайної ситуації та організацію оперативно-технічної і звітної документації, наказ Державної служби України з надзвичайних ситуацій [22-25].

До динамічних явищ на поверхні землі належать абіотичні та біотичні небезпеки. Одним із проявів природних небезпек є стихійні лиха.

Стихійні лиха - це природні явища, які мають надзвичайний характер та призводять до порушення нормальної діяльності населення, загибелі людей, руйнування і знищення матеріальних цінностей.

За причиною виникнення стихійні лиха поділяють на: тектонічні (пов'язані з процесами, які відбуваються в надрах земної кори), топологічні (пов'язані з процесами, які відбуваються на поверхні землі), метеорологічні (пов'язані з процесами, які відбуваються в атмосфері).

Види абіотичних небезпек: літосферні, гідросферні, атмосферні, космічні. Серед літосферних небезпек розрізняють землетруси, вулкани, зсуви, селі.

— Землетрус — підземні поштовхи та коливання земної поверхні, зумовлені раптовими зміщеннями і розривами в корі або у верхній частині мантії, які передаються на великі відстані у вигляді пружних коливань.

— Зсув - сповзання мас гірських порід вниз по схилу, яке виникає через порушення рівноваги. Зсуви бувають повільні (см/доба), середньої швидкості (м/год), швидкі (км/год).

— Селі - це паводки з великою концентрацією ґрунту, мінеральних частин, каміння, уламків гірських порід (від 10-15 до 75% об'єму потоку). За складом матеріалу, що переносить потік, розрізняють: грязьові, грязекам'яні, водокам'яні.

Гідросферні небезпеки - повені, снігові лавини, шторми, цунамі.

— Повінь — значне затоплення місцевості внаслідок підйому рівня води в річці, озері, водосховищі. В Україні повені спостерігаються в басейнах Дніпра, Дністра, Прип'яті. В останні роки найчастіше спостерігаються в Закарпатті (Західний Буг та Тиса).

— Снігова лавина - величезна маса снігу, яка зсувається або падає зі стрімких гірських схилів, захоплюючи різні об'єкти, що трапляються на шляху. Лавина супроводжується утворенням передлавинної поверхневої хвилі, що має найбільшу руйнівну силу. Розрізняють сухі (зимові) та мокрі (весняні) снігові лавини.

— Шторм - тривалий, дуже сильний вітер, що спричиняє значні руйнування на суші та велике хвилювання на морі.

— Цунамі - великі хвилі, що виникають на поверхні океану під час підводних землетрусів. Атмосферні небезпеки - бурі, урагани, тайфуни, цунамі, смерчі, морози, засуха тощо.

— Ураган - вітер руйнівної сили зі швидкістю 35 м/с.

— Буря - шторм, тривалий, дуже сильний вітер (понад 20 м/с), спричинений зазвичай циклоном.

— Смерч - сильний локальний атмосферний вихор (діаметр до 1000м), в якому повітря обертається зі швидкістю до 100 м/с.

3.2 Організація ведення робіт в аварійних умовах

Організація ведення робіт в аварійних умовах є надзвичайно важливим аспектом забезпечення безпеки та ефективності в небезпечних ситуаціях. Одні із основних кроків такій ситуації [26, 27]:

— оцінка ризиків: Перед початком будь-яких аварійних робіт необхідно провести оцінку ризиків. Визначте потенційні небезпеки, що виникають у зв'язку з аварійною ситуацією, і розробіть план заходів для зниження цих ризиків.

— створення команди: Сформуйте команду, яка буде відповідальна за проведення аварійних робіт. Команда повинна складатися з кваліфікованих фахівців з необхідними навичками та досвідом в роботі з аварійними ситуаціями.

— встановлення комунікації: Забезпечте ефективну систему комунікації всередині команди та з іншими відповідними структурами (пожежна служба, поліція, медична допомога тощо). Засоби зв'язку повинні бути надійними та доступними, щоб забезпечити швидкий обмін інформацією.

— планування та координація: Розробіть детальний план робіт, включаючи послідовність дій, розподіл завдань, використання необхідного обладнання та ресурсів. Координуйте дії команди таким чином, щоб досягти ефективного вирішення проблеми.

— навчання та підготовка працівників: Працівники, які беруть участь у роботах в аварійних умовах, повинні мати необхідні навички та знання. Забезпечте їм належне навчання з процедур безпеки, використання спеціального обладнання та заходів, які слід вживати під час аварій.

Складовою частиною процесу ліквідації наслідків НС є рятування людей. Цей процес представляє собою взаємопов'язаний комплекс робіт, які за характером виконання діляться на три специфічні групи: рятувальні, спеціальні і допоміжні.

Рятувальні роботи, що безпосередньо пов'язані із рятуванням людей включають у себе:

— пошук постраждалих у місцях їхнього можливого блокування;

— деблокування постраждалих (забезпечення доступу до них);

— надання постраждалим домедичної допомоги;

— евакуація постраждалих із місць блокування.

— Спеціальні роботи включають:

— гасіння пожеж;

— ліквідацію аварії на комунально-енергетичних і технологічних мережах;

— улаштування проїздів (проходів) у завалах;

— зміцнення (обвалення) нестійких конструкцій.

У результаті виконання спеціальних робіт створюються умови найбільш сприятливі для виконання рятувальних робіт і запобігання додаткового ураження людей. Допоміжні роботи пов'язані з інженерною та організаційною підготовкою ділянки рятувальних робіт та робочих місць. До них відносяться:

— розчищення майданчиків;

— установлення на них техніки;

— огородження, попереджувальних знаків;

— освітлення робочих місць.

Час, необхідний для виконання технологічних операцій, є основним критерієм, що характеризує доцільність їх застосування у технологічному процесі порятунку людей, у певних організаційнотехнологічних умовах.

У практиці рятування постраждалих при обваленні будівель використовуються наступні рятувальні технології [28]:

— пошук постраждалих за допомогою спеціально навчених собак (кінологічний спосіб);

— пошук постраждалих за допомогою спеціальних приладів;

— деблокування постраждалих із завалу, що складається із дрібних уламків, способом розбирання завалу зверху;

— деблокування постраждалих із завалу, що складається із великих уламків, способом розбирання завалу зверху;

— деблокування постраждалих із завалу способом суцільного горизонтального розбирання;

— деблокування постраждалих способом улаштування лазу у завалі;

— деблокування постраждалих із завалених приміщень;

— деблокування постраждалих з верхніх поверхів будівлі з використанням вертольоту;

— деблокування постраждалих з верхніх поверхів будівлі із застосуванням автодрабин;

— порятунок постраждалих з верхніх поверхів будівлі за допомогою автовишок та автопідйомників;

— порятунок постраждалих з верхніх поверхів будівлі по збереженим або тимчасово відновленим сходовим маршам;

— порятунок постраждалих з верхніх поверхів будівлі з використанням канатної дороги;

— деблокування постраждалих з верхніх поверхів будівлі з використанням альпіністських засобів.

У загальному вигляді процес рятування постраждалих може бути представлений як комплексний технологічний процес, що включає такі етапи [29]:

— загальна спеціальна розвідка осередку ураження та об'єкта робіт;

— підготовчі роботи;

— аварійно-технічні роботи;

— пошуково-рятувальні роботи;

— роботи з деблокування та витягання постраждалих;

евакуація, упізнання та поховання загиблих.

На кожному з наведених технологічних етапів здійснюються відповідні види робіт, а вони, у свою чергу виконуються певними способами. Найбільш складним технологічним етапом при обваленні будівель і споруд є інженерні роботи з деблокування та витягання постраждалих.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було створено сервіс генерації юніт-тестів на основі штучного інтелекту, який інтегрується в життєвий цикл Java-застосунків. Проєкт охоплює всі ключові етапи — від аналізу предметної області й архітектурного проєктування до реалізації REST API, автоматизації тестування та інтеграції з CI/CD-конвеєром. У фокусі проєкту — використання сучасного стеку технологій, зокрема Java 23, Spring Boot, Jacoco, GitHub Actions і генеративних інструментів на зразок Diffblue.

Особливу увагу було приділено якості коду та автоматизованому контролю покриття за допомогою Jacoco. Завдяки інтеграції з системами CI/CD і генерації тестів після кожного коміту вдалося досягти високого рівня стабільності проєкту. Реалізований REST API дозволив забезпечити зручну взаємодію з користувачем і підтримку розширення через зовнішні інструменти та плагіни. Завдяки гнучкості архітектури сервіс може бути адаптований до інших мов програмування або інших генераторів, що працюють на базі LLM або еволюційних підходів. Застосування практик безперервної інтеграції та автоматичного розгортання дозволило досягти надійної автоматизації, яка мінімізує втручання розробника при перевірці якості.

Розроблений сервіс також має значний освітній та дослідницький потенціал. Його можна використовувати як платформу для вивчення підходів до автоматичного тестування, роботи з генеративними ШІ-моделями та концепцій DevOps. Проєкт демонструє ефективне поєднання сучасних технологій програмної інженерії, дозволяючи створювати безпечні, масштабовані та автоматизовані рішення, які можуть бути адаптовані до потреб різних команд і компаній. Кваліфікаційна робота засвідчує глибоке розуміння теоретичних знань і їхнє успішне застосування в реальному проєкті.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лещенко А.В., Соколюк В.І. Розробка Java-додатків із використанням Spring Boot. – Київ: Освіта України, 2022. – 312 с.

2. Іванов С.П., Гринчук А.І. Java 17–23: сучасні підходи в програмуванні. – Львів: Техніка, 2023. – 274 с.

3. Бойко Ю.М., Вовк М.С. Веб-сервіси і REST API: розробка на Spring. – Харків: ФОП Савчук О.О., 2021. – 198 с.

4. Пархоменко Н.В., Лисенко І.О. Системи автоматизованого тестування ПЗ. – Дніпро: ІТ Академія, 2020. – 221 с.

5. Коваль І.М., Литвин О.Б. Тестування програмного забезпечення: теорія і практика. – Тернопіль: ТНТУ, 2021. – 168 с.

6. Ченцов Д.В. Інтелектуальні системи в забезпеченні якості коду. – Київ: Наукова думка, 2024. – 245 с.

7. Коваль Р.А. Автоматизоване тестування з використанням штучного інтелекту. – Львів: Нові Технології, 2022. – 278 с.

8. Демиденко Т.І., Гуменюк С.О. Основи розробки програмних сервісів на Java. – Вінниця: ВДТУ, 2023. – 236 с.

9. Назаренко О.П. Патерни проєктування у сучасних програмних системах. – Одеса: ОНПУ, 2021. – 192 с.

10. Яворський В.М. Програмна інженерія: навчальний посібник. – Київ: КНЕУ, 2022. – 328 с.

11. Spring Boot Reference Documentation. [Електронний ресурс] – Режим доступу: https://docs.spring.io/spring-boot/docs/current/reference/html/

12. Java SE 23 Documentation. Oracle. [Електронний ресурс] – Режим доступу: https://docs.oracle.com/en/java/javase/23/

13. Jacoco Java Code Coverage Library. [Електронний ресурс] – Режим доступу: https://www.jacoco.org/

14. GitHub Actions Documentation. [Електронний ресурс] – Режим доступу: https://docs.github.com/en/actions

15. Diffblue Cover Documentation. [Електронний ресурс] – Режим доступу: https://docs.diffblue.com/

16. Refactoring.Guru. Патерни проектування. [Електронний ресурс] – Режим доступу: https://refactoring.guru/uk/design-patterns

17. Робота з Maven: офіційна документація. Apache Maven. [Електронний ресурс] – Режим доступу: https://maven.apache.org/guides/

18. YAML Language Reference. [Електронний ресурс] – Режим доступу: https://yaml.org/spec/

19. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Pearson Education, 2021. – 432 p.

20. OpenAI. Generative AI for Code. [Електронний ресурс] – Режим доступу: https://openai.com/research/

21. Запорожець О.І., Халмурадов В.І., Применко В.І. Безпека життєдіяльності: Підручник. – К.: Центр учбової літератури, 2013. – С. 84-130.

22. Кодекс цивільного захисту України [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/5403-17#Text

23. Про затвердження Положення про підсистему реагування на надзвичайні ситуації, проведення аварійно-рятувальних та інших невідкладних робіт єдиної державної системи цивільного захисту : наказ Міністерства внутрішніх справ України від 04.05.2016 №356.

24. Про затвердження Положення про штаб з ліквідації наслідків надзвичайної ситуації та видів оперативно-технічної і звітної документації штабу з ліквідації наслідків надзвичайної ситуації : наказ Міністерства внутрішніх справ України від 26.12.2014 №1406.

25. Про організацію роботи штабу з ліквідації наслідків надзвичайної ситуації та забезпечення його готовності : наказ Державної служби України з надзвичайних ситуацій від 16.03.2015 №149.

26. Рекомендації для роботодавців щодо організації виконання робіт підвищеної небезпеки під час воєнних (бойових) дій [Електронний ресурс] – Режим доступу: https://dsp.gov.ua/faq/rekomendatsii-dlia-robotodavtsiv-shchodo-orhanizatsii-vykonannia-robit-pidvyshchenoi-nebezpeky-pid-chas-voiennykh-boiovykh-dii/

27. Аветисян В.Г., Тригуб В.В., Грицина І.М., Остапов К.М. Організації аварійнорятувальних робіт : курс лекцій – Х: НУЦЗУ, 2017. – 141 с.

28. Організація аварійно-рятувальних робіт : навчальний посібник /Р.Т. Ратушний, В.Б. Лоїк, О.Д. Синельніков, В.М. Ковальчук – Львів: Видавництво ЛДУ БЖД, 2020. – 394 с.

29. Методичні рекомендації щодо розроблення планів локалізації і ліквідації аварій та їх наслідків : наказ Державної служби України з надзвичайних ситуацій від 17 травня 2022 року за №253

ДОДАТКИ

Додаток А – Лістинги коду проєкту

Лістинг коду А.1 – Клас TestGenerationService

```
@Service
public class TestGenerationService {
    private final AIAdapter aiAdapter;
    private final ProjectStructureHelper structureHelper;
    private final SingletonLogger logger;
    public TestGenerationService(AIAdapter aiAdapter,
                                 ProjectStructureHelper structureHelper) {
        this.aiAdapter = aiAdapter;
        this.structureHelper = structureHelper;
        this.logger = SingletonLogger.getInstance();
    }
    public String generateTestForClass(String classCode) {
        logger.info("Генерація тестів для наданого класу");
        try {
            return aiAdapter.generateUnitTests(classCode);
        } catch (Exception e) {
            logger.error("Помилка при генерації тестів: " + e.getMessage());
            throw new RuntimeException("Не вдалося згенерувати тести", e);
        }
    }
    public Map<String, String> generateTestsForProject(String projectPath) {
        Map<String, String> result = new HashMap<>();
        List<String> classPaths = structureHelper.findJavaClasses(projectPath);
        for (String classPath : classPaths) {
            String classCode = structureHelper.readClassFile(classPath);
            String testCode = generateTestForClass(classCode);
            result.put(classPath, testCode);
        }
        return result;
    }
}
```

Лістинг коду А.1 – Клас TestClassGenerator

```
// Генерація тестів
public class TestClassGenerator {
    public String buildTestClass(String className, List<String> methods) {
        StringBuilder builder = new StringBuilder();
        builder.append("import org.junit.jupiter.api.*;\n")
               .append("import static org.mockito.Mockito.*;\n")
               .append("import org.mockito.*;\n")
               .append("\n")
               .append("public class ").append(className).append("Test {\n\n")
               .append("    private ").append(className).append(" instance;\n\n")
               .append("    @BeforeEach\n")
               .append("    public void setup() {\n")
               .append("        instance = new
").append(className).append("();\n")
               .append("    }\n\n");
        for (String method : methods) {
            builder.append("    @Test\n")
```

```
                    .append("    public void
test").append(capitalize(method)).append("() {\n")
                    .append("        // TODO: реалізувати тест для
").append(method).append("\n")
                    .append("        Assertions.fail(\"Not implemented yet\");\n")
                    .append("    }\n\n");
        }
        builder.append("}\n");
        return builder.toString();
    }
    private String capitalize(String name) {
        if (name == null || name.isEmpty()) return name;
        return Character.toUpperCase(name.charAt(0)) + name.substring(1);
    }
}
```

Лістинг коду А.1 – Клас CoverageReport

```
public class CoverageReport {
    private List<ClassCoverageInfo> classCoverages;
    public CoverageReport(List<ClassCoverageInfo> classCoverages) {
        this.classCoverages = classCoverages;
    }
    public List<ClassCoverageInfo> getClassCoverages() {
        return classCoverages;
    }
    public void setClassCoverages(List<ClassCoverageInfo> classCoverages) {
        this.classCoverages = classCoverages;
    }
}
```

Лістинг коду А.1 – Клас ClassCoverageInfo

```
public class ClassCoverageInfo {
    private String className;
    private int coveredLines;
    private int totalLines;
    public ClassCoverageInfo(String className, int coveredLines, int totalLines) {
        this.className = className;
        this.coveredLines = coveredLines;
        this.totalLines = totalLines;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
    public int getCoveredLines() {
        return coveredLines;
    }
    public void setCoveredLines(int coveredLines) {
        this.coveredLines = coveredLines;
    }
    public int getTotalLines() {
        return totalLines;
    }
```

```
    public void setTotalLines(int totalLines) {
        this.totalLines = totalLines;
    }
    public double getCoveragePercentage() {
        return totalLines == 0 ? 0.0 : ((double) coveredLines / totalLines) *
100.0;
    }
}
```

Лістинг коду А.1 – Клас ReportParser

```
@Component
public class ReportParser {
    public CoverageReport parse(File xmlFile) {
        // Простий приклад: парсинг XML вручну або через JAXB
        // Тут лише імітація
        CoverageReport report = new CoverageReport();
        report.setLineCoverage(85.2);
        report.setMethodCoverage(91.0);
        return report;
    }
}
```

Лістинг коду А.1 – Клас GitHubActionsIntegration

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
public class GitHubActionsIntegration {
    private static final String WORKFLOW_FILE_PATH = ".github/workflows/test-
generator.yml";
    public void setupWorkflow() {
        String workflowContent = """
            name: Generate and Test
            on: [push, pull_request]
            jobs:
              build:
                runs-on: ubuntu-latest
                steps:
                  - uses: actions/checkout@v3
                  - name: Set up JDK
                    uses: actions/setup-java@v3
                    with:
                      java-version: '23'
                      distribution: 'temurin'
                  - name: Build with Maven
                    run: mvn clean install
                  - name: Run Test Generator
                    run: java -jar test-generator.jar
                  - name: Generate Coverage Report
                    run: mvn jacoco:report
            """;
        try {
            Files.createDirectories(Paths.get(".github/workflows"));
            Files.writeString(Paths.get(WORKFLOW_FILE_PATH), workflowContent);
        } catch (IOException e) {
```

```
            throw new RuntimeException("Не вдалося створити GitHub Actions
workflow файл", e);
        }
    }
    public boolean workflowExists() {
        return Files.exists(Paths.get(WORKFLOW_FILE_PATH));
    }
}
```

Лістинг коду А.1 – Клас TestGenerationController

```
@RestController
@RequestMapping("/api/tests")
public class TestGenerationController {
    private final TestGenerationService testGenerationService;
    public TestGenerationController(TestGenerationService testGenerationService) {
        this.testGenerationService = testGenerationService;
    }
    @PostMapping("/class")
    public ResponseEntity<String> generateTestForClass(@RequestBody String
classCode) {
        try {
            String testCode =
testGenerationService.generateTestForClass(classCode);
            return ResponseEntity.ok(testCode);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                                 .body("Помилка при генерації тестів: " +
e.getMessage());
        }
    }
    @PostMapping("/project")
    public ResponseEntity<Map<String, String>>
generateTestsForProject(@RequestParam String projectPath) {
        try {
            Map<String, String> tests =
testGenerationService.generateTestsForProject(projectPath);
            return ResponseEntity.ok(tests);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                                 .body(Map.of("error", "Не вдалося згенерувати
тести: " + e.getMessage()));
        }
    }
}
```

Лістинг коду А.1 – Клас GitHubActionsIntegration

```
public class GitHubActionsIntegration {
    private final String workflowFilePath = ".github/workflows/test-
generation.yml";
    public String createWorkflowContent() {
        return """
            name: Generate and Verify Tests
            on:
              push:
                branches: [ main ]
```

```
              pull_request:
                branches: [ main ]
            jobs:
              build:
                runs-on: ubuntu-latest
                steps:
                  - uses: actions/checkout@v2
                  - name: Set up JDK
                    uses: actions/setup-java@v2
                    with:
                      java-version: '23'
                      distribution: 'temurin'
                  - name: Build with Maven
                    run: mvn clean install
                  - name: Run Test Generation Service
                    run: java -jar target/test-generator.jar
            """;
    }
    public void writeWorkflowToFile() {
        try {
            Path path = Paths.get(workflowFilePath);
            Files.createDirectories(path.getParent());
            Files.writeString(path, createWorkflowContent(),
StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
        } catch (IOException e) {
            throw new RuntimeException("Не вдалося створити workflow файл: " +
e.getMessage(), e);
        }
    }
}
```

Лістинг коду А.1 – Клас TestGenerationController

```
@RestController
@RequestMapping("/api/tests")
public class TestGenerationController {
    private final TestGenerationService testGenerationService;
    public TestGenerationController(TestGenerationService testGenerationService) {
              this.testGenerationService = testGenerationService;
    }
    @PostMapping("/generate")
    public ResponseEntity<Map<String, String>> generateTests(@RequestBody
Map<String, String> request) {
        String classCode = request.get("classCode");
        String testCode = testGenerationService.generateTestForClass(classCode);
        Map<String, String> response = new HashMap<>();
        response.put("testCode", testCode);
        return ResponseEntity.ok(response);
    }
    @PostMapping("/generate-project")
    public ResponseEntity<Map<String, String>>
generateTestsForProject(@RequestBody Map<String, String> request) {
        String projectPath = request.get("projectPath");
        Map<String, String> tests =
testGenerationService.generateTestsForProject(projectPath);
        return ResponseEntity.ok(tests);
    }
}
```

Лістинг коду А.1 – Клас RestExceptionHandler

```
@ControllerAdvice
public class RestExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<Map<String, String>>
handleRuntimeException(RuntimeException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", ex.getMessage());
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
    }
    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<Map<String, String>>
handleIllegalArgumentException(IllegalArgumentException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", ex.getMessage());
        return ResponseEntity.badRequest().body(error);
    }
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Map<String, String>> handleGenericException(Exception
ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", "Невідома помилка: " + ex.getMessage());
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
    }
}
```

Лістинг коду А.1 – Клас LoggingAspect

```
@Aspect
@Component
public class LoggingAspect {
    private final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);
    @Pointcut("execution(* com.example..*(..))")
    public void applicationPackagePointcut() {
        // Визначення точок зрізу для всіх методів у пакеті com.example
    }
    @Before("applicationPackagePointcut()")
    public void logBefore(JoinPoint joinPoint) {
        logger.info("Виклик методу: {}",
joinPoint.getSignature().toShortString());
    }
    @AfterReturning(pointcut = "applicationPackagePointcut()", returning =
"result")
    public void logAfter(JoinPoint joinPoint, Object result) {
        logger.info("Метод {} завершився з результатом: {}",
joinPoint.getSignature().toShortString(), result);
    }
    @AfterThrowing(pointcut = "applicationPackagePointcut()", throwing = "ex")
    public void logException(JoinPoint joinPoint, Throwable ex) {
        logger.error("Метод {} кинув виключення: {}",
joinPoint.getSignature().toShortString(), ex.getMessage());
    }
}
```

Лістинг коду А.1 – Клас RestExceptionHandler

```
@ControllerAdvice
public class RestExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ResponseBody
    public Map<String, String> handleRuntimeException(RuntimeException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", "Внутрішня помилка сервера");
        error.put("details", ex.getMessage());
        return error;
    }
    @ExceptionHandler(IllegalArgumentException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public Map<String, String>
handleIllegalArgumentException(IllegalArgumentException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", "Некоректні вхідні дані");
        error.put("details", ex.getMessage());
        return error;
    }
    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ResponseBody
    public Map<String, String> handleGenericException(Exception ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", "Сталася неочікувана помилка");
        error.put("details", ex.getMessage());
        return error;
    }
}
```

Лістинг коду А.1 – Клас LoggingAspect

```
@Aspect
@Component
public class LoggingAspect {
    private final Logger logger = LoggerFactory.getLogger(this.getClass());
    @Pointcut("execution(* com.example..*(..))")
    public void applicationPackagePointcut() {
        // Точка перехоплення всіх методів у пакеті com.example
    }
    @Before("applicationPackagePointcut()")
    public void logBefore(JoinPoint joinPoint) {
        logger.info("Виконання методу: {}",
joinPoint.getSignature().toShortString());
    }
    @AfterReturning(pointcut = "applicationPackagePointcut()", returning =
"result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        logger.info("Метод {} завершився. Результат: {}",
joinPoint.getSignature().toShortString(), result);
    }
    @AfterThrowing(pointcut = "applicationPackagePointcut()", throwing = "ex")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable ex) {
```

```
        logger.error("Помилка у методі {}: {}",
joinPoint.getSignature().toShortString(), ex.getMessage());
    }
}
```

Лістинг коду А.1 – Клас TestGenerationService

```
package com.example.testgeneration.service;
import com.example.testgeneration.integration.AIAdapter;
import com.example.testgeneration.util.ProjectStructureHelper;
import com.example.testgeneration.util.SingletonLogger;
import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
@Service
public class TestGenerationService {
    private final AIAdapter aiAdapter;
    private final ProjectStructureHelper structureHelper;
    private final SingletonLogger logger;
    public TestGenerationService(AIAdapter aiAdapter,
                                 ProjectStructureHelper structureHelper) {
        this.aiAdapter = aiAdapter;
        this.structureHelper = structureHelper;
        this.logger = SingletonLogger.getInstance();
    }
    /**
     * Генерує тести для одного класу Java, переданого у вигляді сирцевого коду.
     *
     * @param classCode код класу, для якого потрібно згенерувати тести
     * @return згенерований код тестів
     */
    public String generateTestForClass(String classCode) {
        logger.info("Генерація тестів для наданого класу");
        try {
            return aiAdapter.generateUnitTests(classCode);
        } catch (Exception e) {
            logger.error("Помилка при генерації тестів: " + e.getMessage());
            throw new RuntimeException("Не вдалося згенерувати тести", e);
        }
    }
    /**
     * Генерує тести для всіх Java-класів у вказаній директорії проєкту.
     *
     * @param projectPath шлях до проєкту
     * @return мапа, де ключ — шлях до класу, а значення — згенерований код тестів
     */
    public Map<String, String> generateTestsForProject(String projectPath) {
        Map<String, String> result = new HashMap<>();
        List<String> classPaths = structureHelper.findJavaClasses(projectPath);
        for (String classPath : classPaths) {
            try {
                String classCode = structureHelper.readClassFile(classPath);
                String testCode = generateTestForClass(classCode);
                result.put(classPath, testCode);
            } catch (Exception e) {
                logger.warn("Не вдалося згенерувати тести для: " + classPath + " —
" + e.getMessage());
            }
        }
```

```
        return result;
    }
}
```

Лістинг коду А.1 – Класи CoverageReport

```
// Моделі даних
import java.util.List;
/**
 * Представляє загальний звіт про покриття тестами.
 */
public class CoverageReport {
    private List<ClassCoverageInfo> classCoverages;
    public CoverageReport(List<ClassCoverageInfo> classCoverages) {
        this.classCoverages = classCoverages;
    }
    public List<ClassCoverageInfo> getClassCoverages() {
        return classCoverages;
    }
    public void setClassCoverages(List<ClassCoverageInfo> classCoverages) {
        this.classCoverages = classCoverages;
    }
    @Override
    public String toString() {
        return "CoverageReport{" +
                "classCoverages=" + classCoverages +
                '}';
    }
}
```

Лістинг коду А.1 – Клас ClassCoverageInfo

```
/**
 * Представляє інформацію про покриття тестами одного класу.
 */
public class ClassCoverageInfo {
    private String className;
    private int coveredLines;
    private int totalLines;
    public ClassCoverageInfo(String className, int coveredLines, int totalLines) {
        if (totalLines < 0 || coveredLines < 0 || coveredLines > totalLines) {
            throw new IllegalArgumentException("Некоректні значення покриття для
класу " + className);
        }
        this.className = className;
        this.coveredLines = coveredLines;
        this.totalLines = totalLines;
    }
    public String getClassName() {
        return className;
    }
    public void setClassName(String className) {
        this.className = className;
    }
    public int getCoveredLines() {
        return coveredLines;
    }
```

```
    public void setCoveredLines(int coveredLines) {
        if (coveredLines < 0 || coveredLines > this.totalLines) {
            throw new IllegalArgumentException("Недійсне значення coveredLines");
        }
        this.coveredLines = coveredLines;
    }
    public int getTotalLines() {
        return totalLines;
    }
    public void setTotalLines(int totalLines) {
        if (totalLines < 0) {
            throw new IllegalArgumentException("totalLines не може бути меншим за
нуль");
        }
        this.totalLines = totalLines;
    }
    public double getCoveragePercentage() {
        return totalLines == 0 ? 0.0 : ((double) coveredLines / totalLines) *
100.0;
    }
    @Override
    public String toString() {
        return "ClassCoverageInfo{" +
                "className='" + className + '\'' +
                ", coveredLines=" + coveredLines +
                ", totalLines=" + totalLines +
                ", coverage=" + getCoveragePercentage() + "%" +
                '}';
    }
}
```

Лістинг коду А.1 – Клас TestClassGenerator

```
import java.util.List;
/**
 * Генерує тестовий клас на основі імені класу та списку методів.
 */
public class TestClassGenerator {
    /**
     * Створює тіло тестового класу у вигляді Java-коду.
     *
     * @param className Назва класу, для якого створюються тести.
     * @param methods Список методів, які потрібно покрити тестами.
     * @return Рядок з кодом тестового класу.
     */
    public String buildTestClass(String className, List<String> methods) {
        StringBuilder builder = new StringBuilder();
        builder.append("import org.junit.jupiter.api.*;\n\n")
               .append("public class ").append(className).append("Test {\n\n");
        for (String method : methods) {
            builder.append("    @Test\n")
                   .append("    public void
test").append(capitalize(method)).append("() {\n")
                   .append("        // TODO: реалізувати тест для
").append(method).append("\n")
                   .append("        Assertions.fail(\"Not implemented yet\");\n")
                   .append("    }\n\n");
        }
        builder.append("}\n");
        return builder.toString();
```

```
    }
    /**
     * Перетворює назву методу у формат з великої літери.
     *
     * @param name назва методу.
     * @return Назва з великої літери.
     */
    private String capitalize(String name) {
        if (name == null || name.isEmpty()) return name;
        return Character.toUpperCase(name.charAt(0)) + name.substring(1);
    }
}
```

Лістинг коду А.1 – Клас AIAdapter

```
// Інтеграція з AI-моделлю
public class AIAdapter {
    private final String endpointUrl = "http://localhost:5000/generate-tests";
    public String generateUnitTests(String classCode) {
        try {
            HttpClient client = HttpClient.newHttpClient();
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(endpointUrl))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString("{\"code\": \"" +
escapeJson(classCode) + "\"}"))
                .build();
            HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
            if (response.statusCode() == 200) {
                return response.body();
            } else {
                throw new RuntimeException("Помилка від AI-сервісу: " +
response.statusCode());
            }
        } catch (Exception e) {
            throw new RuntimeException("Не вдалося згенерувати тести: " +
e.getMessage(), e);
        }
    }
    private String escapeJson(String str) {
        return str.replace("\"", "\\\"").replace("\n", "\\n");
    }
}
```

Лістинг коду А.1 – Клас GitHubActionsIntegration

```
// Інтеграція з GitHub Actions
public class GitHubActionsIntegration {
    public void triggerWorkflow(String repoUrl, String token, String workflowFile)
{
        try {
            String apiUrl = repoUrl.replace("https://github.com/",
"https://api.github.com/repos/")
                + "/actions/workflows/" + workflowFile + "/dispatches";
            String payload = "{ \"ref\": \"main\" }";
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(apiUrl))
                .header("Authorization", "Bearer " + token)
```

```
                .header("Accept", "application/vnd.github.v3+json")
                .POST(HttpRequest.BodyPublishers.ofString(payload))
                .build();
            HttpClient.newHttpClient().send(request,
HttpResponse.BodyHandlers.ofString());
        } catch (Exception e) {
            throw new RuntimeException("Не вдалося викликати GitHub Actions: " +
e.getMessage(), e);
        }
    }
}
```

Лістинг коду А.1 – Клас TestGenerationController

```
@RestController
@RequestMapping("/api/tests")
public class TestGenerationController {
    private final TestGenerationService testService;
    public TestGenerationController(TestGenerationService testService) {
        this.testService = testService;
    }
    @PostMapping("/single")
    public ResponseEntity<String> generateForClass(@RequestBody String classCode)
{
        try {
            String generatedTest = testService.generateTestForClass(classCode);
            return ResponseEntity.ok(generatedTest);
        } catch (Exception e) {
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Помилка: " +
e.getMessage());
        }
    }
    @PostMapping("/project")
    public ResponseEntity<Map<String, String>> generateForProject(@RequestBody
Map<String, String> pathMap) {
        try {
            String projectPath = pathMap.get("path");
            Map<String, String> generatedTests =
testService.generateTestsForProject(projectPath);
            return ResponseEntity.ok(generatedTests);
        } catch (Exception e) {
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
        }
    }
}
```

Лістинг коду А.1 – Клас ReportParser

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.*;
import java.io.File;
import java.util.*;
public class JacocoReportParser implements ReportParser {
    @Override
```

```
    public CoverageReport parseReport(String xmlPath) {
        List<ClassCoverageInfo> classInfos = new ArrayList<>();
        try {
            File inputFile = new File(xmlPath);
            DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(inputFile);
            doc.getDocumentElement().normalize();
            NodeList classNodes = doc.getElementsByTagName("class");
            for (int i = 0; i < classNodes.getLength(); i++) {
                Node classNode = classNodes.item(i);
                if (classNode.getNodeType() == Node.ELEMENT_NODE) {
                    Element classElement = (Element) classNode;
                    String className =
classElement.getAttribute("name").replace("/", ".");
                    NodeList lineNodes =
classElement.getElementsByTagName("line");
                    int totalLines = 0;
                    int coveredLines = 0;
                    for (int j = 0; j < lineNodes.getLength(); j++) {
                        Element lineElement = (Element) lineNodes.item(j);
                        totalLines++;
                        int ci = Integer.parseInt(lineElement.getAttribute("ci"));
                        int mi = Integer.parseInt(lineElement.getAttribute("mi"));
                        if (ci > 0 && mi == 0) {
                            coveredLines++;
                        }
                    }
                    classInfos.add(new ClassCoverageInfo(className, coveredLines,
totalLines));
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return new CoverageReport(classInfos);
    }
}
```

Лістинг коду А.1 – Клас LoggingAspect

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;
@Aspect
@Component
public class LoggingAspect {
    private final SingletonLogger logger = SingletonLogger.getInstance();
    @Before("execution(* *.generate*(..))")
    public void logBeforeMethod(JoinPoint joinPoint) {
        logger.info("Виклик методу: " + joinPoint.getSignature().getName());
    }
    @AfterReturning(pointcut = "execution(* *.generate*(..))", returning =
"result")
    public void logAfterMethod(JoinPoint joinPoint, Object result) {
        logger.info("Метод завершено: " + joinPoint.getSignature().getName() +
```

```
                    " | Результат: " + (result != null ? result.toString() :
"null"));
    }
}
```

Додаток Б – Диск із кваліфікаційною роботою бакалавра