Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет Комп'ютерні технології і системи

Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка
до кваліфікаційної роботи
ОС магістра

на тему: Дослідження властивостей генерації тестів методами білої скриньки
за освітньою програмою: Інженерія програмного забезпечення

Виконав: студент групи: ПЗ2421 Володимир МАКСИМЧУК

Керівник: Віктор ШИНКАРЕНКО

Нормоконтролер: Світлана ВОЛКОВА

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Дніпро – 2026

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems

Department Computer information technology

Explanatory Note
to Master's Thesis

on the topic: Research on Test Generation Properties Using White-Box Methods

according to educational curriculum: Software engineering

in the Speciality: 121 Software engineering

Done by the student of the group PZ2421: Volodymyr MAKSYMCHUK

Scientific Supervisor: Viktor SHINKARENKO

Normative controller : Svitlana VOLKOVA

Dnipro – 2026

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: «Комп'ютерні технології і системи»
Кафедра: «Комп'ютерні інформаційні технології»
Рівень вищої освіти: магістр
Освітня програма: «Інженерія програмного забезпечення»
Спеціальність: «121 Інженерія програмного забезпечення»
(шифр та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри _____
_____ Вадим ГОРЯЧКІН

Дата _____

ЗАВДАННЯ

на кваліфікаційну роботу Магістр

студенту: Володимиру МАКСИМЧУКУ

1. Тема роботи: «Дослідження властивостей генерації тестів методами білої скриньки»

Керівник роботи: Віктор ШИНКАРЕНКО

затверджені наказом від "02" жовтня 2025 р. № 1401

2. Строк подання студентом роботи: 05.01.2026

3. Вихідні дані до роботи: згенеровані набори тестів за допомогою генеративних моделей

4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):

4.1 Аналітична частина: огляд методів тестування ПЗ та принципів контролю якості.

4.2 Основна частина: експеримент із оцінки якості генерації тестів моделями та аналіз результатів.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	09.10.2025 – 23.10.2025	
2	Аналіз сучасного стану вирішення обраної задачі та програмно-апаратного забезпечення	24.10.2025 – 08.11.2025	
3	Збір вимог до програми, вибір засобів штучного інтелекту для дослідження	10.11.2025 – 16.11.2025	30%
4	Зовнішнє та внутрішнє проектування	18.11.2025 – 09.12.2025	
5	Розробка програмного забезпечення	11.12.2025 – 17.12.2025	60%
6	Тестування та налагодження програмного забезпечення	20.12.2025 – 26.12.2025	
7	Проведення експериментального дослідження та аналіз результатів	26.12.2025 – 04.01.2026	
8	Оформлення пояснювальної записки	04.01.2025 – 06.01.2026	
9	Розробка демонстраційних матеріалів	06.01.2026 – 11.01.2026	100%
10	Подання кваліфікаційної роботи до кафедри	16.01.2026	
11	Подання кваліфікаційної роботи до кафедри	24.01.2026	

Студент _____ Володимир МАКСИМЧУК

Керівник роботи _____ Віктор ШИНКАРЕНКО

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра:

117с., 8 рис., 2 табл., 30 джерел, 4 додатки.

Об’єкт дослідження – процеси автоматичної генерації тестових сценаріїв за допомогою сучасних методів штучного інтелекту.

Предмет дослідження – генеративні нейромережеві моделі та набори даних, що визначають точність і релевантність згенерованих тестів.

Мета роботи – дослідити сучасні генеративні підходи до автоматичного створення тестів, оцінити їх ефективність та визначити оптимальні рішення для практичного використання у контролі якості програмного забезпечення.

Методи дослідження – експериментальне тестування генеративних моделей на різних наборів даних, порівняльний аналіз якості згенерованих тестів за показниками точності, повноти та релевантності.

Отримані результати – розроблено програмний застосунок для навчання та тестування генеративних моделей, що дозволяє оцінювати якість автоматично згенерованих тестових сценаріїв. Проведено експериментальне дослідження та визначено показники ефективності моделей для різних типів завдань контролю якості ПЗ.

Ключові слова: ГЕНЕРАТИВНА МОДЕЛЬ, АВТОМАТИЧНЕ ТЕСТУВАННЯ, ТЕСТОВИЙ СЦЕНАРІЙ, ШТУЧНИЙ ІНТЕЛЕКТ, QA.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI — Artificial Intelligence, штучний інтелект

DL — Deep Learning, глибинне навчання

ML — Machine Learning, машинне навчання

NLP — Natural Language Processing, обробка природної мови

QA — Quality Assurance, контроль якості програмного забезпечення

SUT — System Under Test, система, що тестується

API — Application Programming Interface, прикладний програмний інтерфейс

CI/CD — Continuous Integration / Continuous Deployment, безперервна інтеграція та розгортання

UI — User Interface, інтерфейс користувача

UX — User Experience, користувацький досвід

TP — Test Plan, план тестування

TC — Test Case, тестовий сценарій

LLM — Large Language Model, велика мовна модель

NN — Neural Network, нейронна мережа

ВСТУП

Програмна інженерія пройшла тривалий шлях еволюції − від перших спроб систематизації процесу створення програмного забезпечення до появи комплексних фреймворків і методологій, які визначають сучасний вигляд галузі. Упродовж останніх десятиліть розробники з усього світу стикалися з численними проблемами, що виникали під час розробки складних програмних систем: від надмірної складності архітектурних рішень до помилок, які важко виявити на етапі тестування. Саме ці труднощі сприяли формуванню принципів, які сьогодні складають основу сучасних практик створення програмного забезпечення − від планування вимог до безперервного тестування й розгортання.

Зі зростанням масштабів та складності програмних систем зросла й роль тестування як одного з ключових механізмів забезпечення надійності та стабільності програмних продуктів. У сучасному світі розробники мають справу з великою кількістю абстракцій, бібліотек і залежностей, створених різними командами, що підвищує ризик виникнення несумісностей, логічних помилок або неочікуваної поведінки системи. Задля мінімізації таких ризиків у процес розробки інтегруються практики тестування, що дозволяють виявити потенційні дефекти ще до етапу релізу.

Серед багатьох підходів особливе місце посідає тестування методом “білого ящика”, яке ґрунтується на аналізі внутрішньої структури програмного коду. На відміну від «чорного ящика», що перевіряє лише зовнішню поведінку системи, метод «білого ящика» дозволяє оцінити повноту покриття коду, перевірити правильність логіки виконання та забезпечити глибше розуміння функціонування програми. Такий підхід дає змогу не лише підвищити якість програмного забезпечення, але й оптимізувати процес розробки, скоротивши кількість прихованих дефектів і витрати на їх виправлення

1 ЗНАЧЕННЯ ТА МЕТОДОЛОГІЯ РОБОТИ

1.1 Актуальність роботи

На момент написання цієї роботи у сфері тестування існує велика кількість усталених практик і фреймворків, спрямованих на підвищення якості програмного коду. Такі інструменти, як NUnit, JUnit, PyTest, xUnit та інші, давно стали стандартом у професійній розробці, проте їх ефективність значною мірою залежить від правильності застосування та контексту використання. З розвитком DevOps-практик та CI/CD-підходів процес тестування дедалі частіше автоматизується, що вимагає глибшого розуміння методів побудови та генерації тестів.

Попри різноманіття інструментів, проблема залишається − в умовах інформаційного шуму складно об’єктивно оцінити якість і доцільність використання певного методу чи засобу автоматичної генерації тестів. Багато оглядів зосереджені на технічних аспектах, але не дають системного порівняльного аналізу ефективності. Саме тому дослідження, яке поєднує теоретичне розуміння методів і їхнє практичне застосування на прикладі конкретного програмного продукту, є актуальним і цінним як для початківців, так і для досвідчених фахівців з тестування.

Дана робота має на меті не лише узагальнити сучасні підходи до тестування методом «білого ящика», а й надати власну оцінку їхньої ефективності в контексті реального програмного проекту. Результати дослідження можуть бути використані як методологічна основа для впровадження практик автоматизованого тестування в нових або вже існуючих програмних продуктах.

1.2 Об’єкт та предмет дослідження

Для перевірки ефективності методів генерації тестів за допомогою інструментів штучного інтелекту було відібрано набір публічних кодових баз та

окремих фрагментів програмного коду, що відрізняються за структурою, складністю й функціональним призначенням. До аналізу включено як прості алгоритмічні модулі, так і компоненти, що реалізують елементи бізнес-логіки або взаємодію між об’єктами.

Об’єктом дослідження виступає процес автоматизованого тестування, орієнтований на оцінку якості покриття коду згенерованими тестами. Предметом дослідження є показники ефективності цих тестів, що визначаються через метрики покриття та аналітичну оцінку результатів.

Особлива увага приділяється застосуванню аналітичних методів і математичного апарату, що використовується у тестовій аналітиці для вимірювання якості коду − таких як щільність покриття, глибина тестування та коефіцієнт виявлення логічних залежностей. Це дозволяє не лише кількісно оцінити ступінь покриття, але й підтвердити отримані результати аналітичним шляхом, забезпечуючи об’єктивність і відтворюваність висновків дослідження.

1.3 Мета й завдання дослідження

Метою роботи є дослідження методів підвищення надійності програмного забезпечення та мінімізації непередбачуваної поведінки компонентів шляхом упровадження практик автоматизованої генерації тестів із використанням інструментів штучного інтелекту. Особливу увагу приділено аналізу якості покриття коду − як кількісному (метрики рядків, гілок і умов), так і якісному (повнота логічних перевірок, релевантність згенерованих даних, продуктивність тестування тощо).

Для досягнення поставленої мети визначено такі основні завдання дослідження:

1. відібрати набір публічних кодових баз і фрагментів коду різного типу (алгоритмічні, структурні, бізнес-орієнтовані) для проведення експериментального аналізу;

2. застосувати інструменти генеративного тестування, орієнтовані на методи «білого ящика», для автоматичного створення тестових сценаріїв;

3. зібрати та проаналізувати метрики покриття коду, включаючи структурне й логічне покриття;

4. провести аналітичну оцінку отриманих результатів із використанням математичного апарату якості програмного коду, що застосовується в аналітичному тестуванні;

5. сформулювати висновки та практичні рекомендації щодо використання генеративних методів тестування для підвищення ефективності контролю якості програмного забезпечення.

1.4 Методи дослідження

Для реалізації експерименту застосовано комбінований практико-аналітичний підхід, що передбачає оцінку якості автоматично згенерованих тестів на основі аналізу різнорідних фрагментів коду з відкритих репозиторіїв. Обрані кодові ділянки охоплюють різні аспекти програмної логіки − від базових алгоритмічних структур до компонентів, пов’язаних із бізнес-правилами та взаємодією між об’єктами.

1.5 Підготовка дослідного матеріалу

Для забезпечення репрезентативності вибірки сформовано набір фрагментів коду, які відрізняються за складністю, архітектурним підходом і стилем реалізації. Це дозволяє оцінити роботу генеративних інструментів у різних умовах − від елементарних методів до комплексних класів і сервісів.

1.6 Автоматична генерація тестів

Процес створення тестів здійснювався із залученням сучасних генеративних систем, зокрема GitHub Copilot та GPT-моделей від Microsoft/OpenAI, які застосовують підхід «білого ящика». Вони аналізують вихідний код, будують модель його логіки, визначають можливі комбінації

вхідних параметрів і формують тестові сценарії, орієнтовані на досягнення максимально можливого покриття. Сформовані тести виконуються в середовищі стандартних фреймворків модульного тестування, що дає змогу зібрати об’єктивні показники їх ефективності.

1.7 Збір та обробка метрик

Під час експерименту фіксуються кількісні метрики, зокрема Line Coverage, Branch Coverage і Condition Coverage, що відображають ступінь охоплення коду тестами. Дані метрики збираються за допомогою інструментів покриття (coverage tools), які автоматично формують звіти та візуалізують результати у вигляді репортів і графічних діаграм. Додатково проводиться логічний аналіз на основі побудови блок-схем та формалізованих моделей програмного потоку. Для цього використовується апарат математичної логіки, зокрема елементи формальної верифікації (формальні методи доказу коректності), який традиційно застосовується для підтвердження повноти тестування та логічної узгодженості коду.

1.8 Аналітична перевірка результатів

Отримані метрики інтерпретуються за допомогою математичних моделей якості програмного коду − обчислюються коефіцієнти щільності покриття, логічної насиченості та ефективності виявлення потенційних помилок. Такий підхід забезпечує не лише емпіричну, але й аналітично обґрунтовану оцінку ефективності згенерованих тестів.

1.9 Узагальнення результатів

Проведений аналіз дозволяє виявити переваги й обмеження генеративних інструментів у контексті тестування коду різної складності. На основі отриманих результатів сформульовано практичні рекомендації щодо доцільності використання Copilot- і GPT-подібних систем у задачах аналітичного тестування для підвищення якості та надійності програмного забезпечення.

Висновки до розділу 1

Отримані результати мають практичну цінність для фахівців у сфері тестування, адже дозволяють чітко окреслити межі застосування ручного й автоматизованого підходів до генерації тестів. Проведений експеримент показує, у яких випадках використання інструментів «білого ящика» є виправданим, а коли доцільніше покладатися на ручне проектування тестів.

Робота може бути корисною для команд, які прагнуть підвищити рівень автоматизації процесів тестування, а також для розробників, що бажають глибше зрозуміти взаємозв’язок між архітектурою програмного продукту та якістю його тестового покриття. Крім того, результати дослідження можуть стати базою для подальших експериментів у сфері генерації тестів на основі статичного аналізу коду, машинного навчання чи гібридних підходів.

У підсумку, проведене дослідження сприяє розвитку культури тестування в сучасній розробці, підкреслюючи важливість системного підходу до оцінки якості програмного забезпечення та обґрунтованого вибору інструментів і методів тестування.

2 ТЕСТУВАННЯ У ПРОГРАМНІЙ ІНЖЕНЕРІЇ

2.1 Загальний контекст контролю якості в програмній інженерії

У сучасній програмній інженерії неможливо уявити повноцінний процес розробки програмного забезпечення без системного та багаторівневого контролю якості реалізації. Контроль якості є невід'ємною складовою життєвого циклу програмного продукту та охоплює всі етапи — від формування вимог і проєктування архітектури до впровадження, супроводу та подальшого розвитку системи. Будь-яка програмна система, незалежно від її масштабу, архітектурної складності чи сфери застосування, повинна проходити ретельну перевірку на відповідність бізнес-вимогам, функціональним і нефункціональним обмеженням, а також загальноприйнятим стандартам надійності, безпеки та продуктивності.

Найбільш поширеним і традиційним підходом у сфері забезпечення якості програмного забезпечення залишається ручне тестування або ручне написання автоматизованих тестів. Такий підхід зазвичай базується на заздалегідь визначених тестових сценаріях і специфікаціях, які формуються представниками бізнесу, аналітиками або власниками продукту [10, 11, 14]. Ручне тестування дозволяє гнучко реагувати на зміни вимог, оцінювати поведінку системи з точки зору кінцевого користувача та виявляти дефекти, які складно формалізувати у вигляді автоматичних перевірок.

Історично тестування програмного забезпечення починалося з повністю ручних підходів, коли розробник або тестувальник перевіряв функціональність системи шляхом безпосереднього виконання сценаріїв використання та фіксації отриманих результатів. На ранніх етапах розвитку програмної інженерії такі перевірки часто виконувалися без чіткої методології, спираючись переважно на досвід і інтуїцію фахівця. Основними принципами цього етапу були інтуїтивність, поступовість і орієнтація на поведінкову логіку користувача, а також на очевидні сценарії взаємодії з системою.

Зі зростанням складності програмних продуктів і збільшенням кількості можливих сценаріїв використання ручні підходи почали демонструвати свої обмеження. Зокрема, вони виявилися недостатньо масштабованими, залежними від людського фактору та складними з точки зору відтворюваності результатів. Це, у свою чергу, стимулювало розвиток формалізованих методів тестування, стандартизацію тестової документації та поступовий перехід до автоматизованих інструментів контролю якості, які доповнюють, але не повністю замінюють ручне тестування.

Таке тестування мало високу вартість і низьку повторюваність: кожен тестовий цикл вимагав значних людських ресурсів, а результати могли бути суб’єктивними [11, 13, 14].

Перехід до напівавтоматизованого тестування стався із розвитком скриптових засобів. Тестувальники почали створювати власні скрипти, що імітували дії користувача або перевіряли окремі функції. Прикладом можуть бути перші утиліти на основі Batch, Perl, Python або VBScript, які дозволяли записувати послідовність дій і повторювати їх автоматично. Цей етап став основою для появи автоматизованих фреймворків тестування.

Однак ручне тестування має низку принципових недоліків. По-перше, воно є аддитивним, тобто орієнтованим на окремі сценарії, а не на повну верифікацію поведінки системи. По-друге, немає гарантії, що перевірений модуль діятиме однаково в усіх контекстах виконання, особливо за змін вхідних даних чи при оновленні залежностей. Часті модифікації функціоналу або логіки призводять до ризику втрати консистентності між очікуваними та фактичними результатами.

Ці проблеми зумовлюють потребу в жорсткішому контракті між розробником і замовником, що забезпечує автоматизовану перевірку очікуваної поведінки системи без залучення людського фактора. Саме тому у критично важливих бізнес-секціях застосунків поступово відмовляються від виключно ручного тестування, віддаючи перевагу аналітичним методам аналізу коду та

контекстним типам тестування (інтеграційному, енд-ту-енд, тестуванню сценаріїв користувача) [12, 14].

На практиці розробники часто створюють допоміжні інструменти, так звані адмін-панелі тестувальників, які дозволяють емітувати поведінку системи через конфігураційні налаштування. Такі рішення підвищують якість ручного тестування, але не замінюють потреби в автоматизації.

2.2 Автоматизоване модульне тестування як напрям розвитку

Методи, розглянуті у цій роботі, охоплюють питання створення автоматизованого модульного тестування, зокрема генерації тестів програмними засобами. Ідея полягає в тому, що програма може сама ініціювати виконання певних ділянок коду з різними наборами вхідних параметрів, відстежуючи результат і порівнюючи його з очікуваними умовами.

Цей підхід дозволяє досягнути більш глибокого контролю логіки програми та оцінити ступінь її тестового покриття. У контексті цього дослідження методи автоматизованого тестування трактуються як програмні механізми, що відтворюють виконання коду в контрольованому середовищі, де кожен набір вхідних даних відповідає певному гілкуванню або оператору програми [3, 10].

Подальший розвиток методів привів до створення структурованих фреймворків, які стандартизували процес автоматичного виконання тестів. З’явилися класичні інструменти − JUnit, NUnit, xUnit, TestNG, які впровадили базові концепції:

1. організацію тестів у класи та методи;
2. ізольованість середовища виконання;
3. автоматичну перевірку результатів через механізми Assert;
4. можливість групування тестів і повторного запуску;
5. інтеграцію з CI/CD-процесами.

Ці системи заклали фундамент першої генерації автоматизації, коли тестові сценарії створювались вручну, але виконувались автоматично. З’явилися

також допоміжні інструменти для створення Data-driven тестів − наприклад, JUnitParams або CsvSource у середовищах .NET [6] і Java, що дозволяли варіювати вхідні дані без зміни тестового коду.

Така методологія особливо актуальна при застосуванні методів білого ящика, які спираються на аналіз вихідного коду. Вона не замінює тестування бізнес-вимог, але доповнює його, дозволяючи точно оцінити, які частини коду залишаються неперевіреними, і які шляхи виконання не були активовані під час тестів.

2.2.1 Виникнення генеративних засобів і шаблонних підходів

Згодом акцент змістився на автоматизацію створення самих тестів. Це призвело до появи інструментів, які будують тестові шаблони на основі аналізу структури вихідного коду або специфікацій.

Такі системи, як EvoSuite (для Java) чи Pex (для .NET), використовували алгоритми пошуку шляхів виконання коду для генерації тестів, що максимально покривають умови та гілки. Принцип їх роботи ґрунтується на символічному виконанні (symbolic execution) − коли змінні програми замінюються математичними виразами, а умови розгалуження аналізуються через обмеження (constraints). Результатом цього аналізу є набір тестових вхідних даних, які забезпечують проходження всіх можливих шляхів виконання [6].

Паралельно розвивалися системи побудови тестових шаблонів на основі опису поведінки (Behavior-Driven Development, BDD). Інструменти, як Cucumber або SpecFlow, дозволили описувати тести природною мовою у форматі Given–When–Then, що спростило зв’язок між вимогами й тестами. Таким чином, тестування почало поєднувати програмну структуру з лінгвістичними формулюваннями.

2.2.2 Перехід до інтелектуальних систем аналізу

З появою великих обсягів коду та необхідністю аналізу контексту виникла потреба у семантичному підході, який розуміє зміст і призначення програм. Цей рівень став основою для мовних та алгоритмічних моделей, що поєднують програмний аналіз із машинним навчанням.

Такі системи, як DeepCode, Snyk Code, SonarLint і Amazon CodeWhisperer, навчаються на великих корпусах відкритого коду та можуть не лише виявляти потенційні помилки, а й пропонувати тести, які відтворюють типові сценарії помилок.

Особливе місце займають LLM-моделі (Large Language Models) − GitHub Copilot, ChatGPT, Code Llama, які оперують лінгвістичною моделлю коду. Вони не просто аналізують синтаксис, а будують семантичне подання задачі, розуміючи контекст функцій, їх взаємозв'язки, вхідні дані й очікувані результати.

На основі цього формується абстрактна задача: "перевір, що функція робить саме те, що описано", − після чого модель підбирає конкретні інструменти реалізації тестів (наприклад, xUnit, Jest або PyTest) і генерує відповідний код [2, 7].

Таким чином, сучасні ІІ-системи працюють на вищому рівні абстракції − вони не конструюють тест покроково, а відтворюють логічну структуру валідації поведінки програми, спираючись на знання про типові патерни реалізації.

2.3 Тестування як інструмент підвищення надійності системи

Тестування програмного забезпечення є невід'ємною частиною життєвого циклу будь-якої системи. Воно виконує одразу дві функції:

1. контроль якості реалізації;
2. забезпечення стійкості до помилок.

Раннє виявлення дефектів дозволяє суттєво знизити вартість їх усунення, спростити подальшу підтримку та скоротити ризики збоїв у продакшн-середовищі.

У найширшому сенсі тестування можна розглядати як контракт між концепцією та реалізацією, який підтверджує, що компонент системи поводиться відповідно до визначених вимог. Однією з найвідоміших методологій у цьому контексті є Test-Driven Development (TDD) [3] − підхід, при якому процес розробки починається не з написання коду, а з написання тестів. Такий підхід створює перевірюваний каркас вимог і гарантує, що подальші зміни не порушують логіку системи [3, 4, 5, 12].

Організація тестування передбачає визначення об’єктів перевірки, постановку цілей, обмежень, вибір вхідних даних і метрик контролю. Типові метрики − це покриття тестами (coverage), цикломатична складність коду, кількість сценаріїв, що охоплені тестами [5, 11].

2.4 Типологія тестування: чорна та біла скринька

Залежно від рівня абстракції виділяють два ключових підходи − тестування чорної скриньки та тестування білої скриньки.

Тестування «чорної скриньки» базується на аналізі бізнес-вимог без доступу до вихідного коду. Метою є перевірка функціональності з точки зору користувача − чи виконує система потрібні дії за заданими сценаріями. Цим здебільшого займаються фахівці QA, які використовують тест-кейси, створені на основі специфікацій [12, 14].

Натомість тестування «білої скриньки» здійснюється розробником або технічним тестувальником. Воно передбачає глибоке занурення у вихідний код і перевірку його логічних шляхів. Цей тип тестування включає кілька рівнів:

1. модульне тестування, де перевіряються незалежні компоненти системи;

2. інтеграційне тестування, спрямоване на взаємодію модулів;

3. тестування користувацьких сценаріїв − для перевірки бізнес-логіки у контексті реальних запитів;

4. розподілене тестування, необхідне для асинхронних або подійних систем.

На практиці обидва підходи комбінуються. «Біла скринька» гарантує повне покриття логіки коду, а «чорна» − відповідність бізнес-очікуванням.

2.5 Аналітичне тестування та формальна верифікація

Подальший розвиток технологій зумовив появу аналітичних методів, які базуються на побудові формальних моделей виконання програм.

У деяких галузях − зокрема у військових, авіаційних, наукових або космічних системах − недостатньо просто виконати тести з різними наборами вхідних даних. Необхідно формально довести коректність програми для всього діапазону можливих умов. Для цього застосовується аналітичне тестування, або формальна верифікація [11].

Суть методу полягає у побудові математичної моделі виконання коду з перевіркою істинності умов у кожній гілці виконання. Це дозволяє підтвердити правильність роботи навіть без фізичного запуску кожного тесту. Такий підхід вимагає використання логічних інваріантів, побудови графів потоку керування (Control Flow Graph, CFG) та аналізу станів змінних [10, 13, 14, 16].

У цих підходах використовуються:

1. графи потоку управління (Control Flow Graphs, CFG), які відображають структуру переходів між операторами;

2. матриці залежностей та інваріанти виконання, що описують логічні зв’язки між змінними;

3. методи формальної верифікації, які перевіряють, чи задовольняє програма заданим логічним властивостям (correctness proofs).

У таких системах тести не просто «покривають» код, а підтверджують його логічну узгодженість. Приклади таких інструментів: Frama-C для C-коду, SPIN для моделювання систем, Z3 від Microsoft Research − для розв’язання логічних обмежень у процесі аналізу програм.

Цей етап умовно можна назвати математичною автоматизацією тестування − коли генерація тестів ґрунтується не на емпіричних правилах, а на формальному доказі того, що програма поводиться правильно для всіх допустимих вхідних даних.

2.6 Відлагодження та профілювання як частина процесу перевірки

Коли програма поводиться неочікувано, необхідно провести детальний аналіз причин. Відлагодження (debugging) є інструментом локалізації помилок і відновлення причинно-наслідкового ланцюга. Під час аналізу фіксуються умови збою, вхідні дані, контекст і час виконання. Виявлені гіпотези перевіряються через повторне відтворення проблеми або построкове виконання коду.

У складних або асинхронних системах доцільно застосовувати профілювання − збір телеметрії, статистики виконання, аналіз часу обробки запитів, а також побудову часових шкал навантаження. Це дозволяє виявити «вузькі місця», які не є явними дефектами, але можуть спричинити нестабільність під навантаженням [10, 14, 15, 16].

2.7 Інваріанти, вибір об’єктів тестування та формування даних

Найчастіше об’єктами тестування стають бізнес-правила, обчислювальні сервіси та допоміжні компоненти, логіка яких часто змінюється під час розвитку системи. Такі ділянки коду називають інваріантними в тому сенсі, що, незважаючи на зміни реалізації, очікуваний результат залишається незмінним [12].

Для таких компонентів автоматичне тестування є особливо корисним: після кожної зміни чи розширення можна перевірити, що система зберегла

консистентність результатів. Це створює передумови для підтримки стабільності під час рефакторингу.

Під час розробки тестів аналізуються допустимі вхідні дані (обмежники), визначаються граничні значення та варіанти виконання умовних операторів. На основі цього формується мінімальний, але достатній набір тестових випадків для досягнення повного покриття [14, 16].

Набір вхідних даних визначається через аналіз коду та логіки функцій. Сформовані значення подаються у вигляді параметрів тестів − одиничних або множинних. У разі відхилення результату від очікуваного це сигналізує або про дефект у реалізації, або про недостатність покриття тестами.

Висновки до розділу 2

Аналіз сучасних підходів до тестування показує, що забезпечення надійності програмного забезпечення є багаторівневим процесом, який включає як ручні, так і автоматизовані методи, а також елементи формальної верифікації.

Проблема полягає в пошуку оптимального балансу між ручним тестуванням, автоматизацією та аналітичними методами, що забезпечують контроль поведінки системи на всіх рівнях − від окремої функції до бізнес-процесу в цілому.

Методи, розглянуті в цій роботі, фокусуються саме на автоматичній генерації тестів методом білого ящика, що дозволяє досягти найвищого рівня охоплення коду та скоротити трудомісткість процесу перевірки. Цей підхід спрямований не лише на підвищення якості коду, а й на формування нової парадигми контролю надійності в розробці програмних систем.

3 РОЗРОБКА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ДОСЛІДЖЕННЯ

У межах експериментальної частини дослідження було розроблено комплекс інструментальних засобів, що забезпечують повний цикл автоматизованого тестування − від вибору об’єктів аналізу до збору й інтерпретації метрик покриття коду. Основна мета створення цього комплексу полягала в побудові стандартизованого середовища, придатного для дослідження ефективності генеративних засобів тестування у реальних умовах програмного розвитку.

3.1 Вибір дослідного матеріалу

Як базу для дослідження обрано низку відкритих програмних проектів із публічних репозиторіїв (зокрема GitHub), що відповідають загальноприйнятим правилам і стандартам написання коду у відповідних предметних областях. Під час відбору враховувались такі критерії:

1. наявність чітко структурованої архітектури з розділенням шарів бізнес-логіки, даних і представлення;
2. застосування принципів SOLID та єдиних підходів до іменування, форматування й побудови класів;
3. наявність у коді алгоритмічних, сервісних і інтеграційних компонентів, що дозволяє охопити різні типи функціональності;
4. відкритість ліцензії, яка дозволяє використовувати фрагменти коду для наукових і навчальних цілей.

Таке різноманіття об’єктів дослідження забезпечило можливість порівняти ефективність генерації тестів у різних контекстах − від простих процедурних реалізацій до складних об’єктно-орієнтованих систем із багаторівневою логікою.

3.2 Вибір тестового фреймворку

Для реалізації експерименту було обрано xUnit − сучасний, модульний і гнучкий фреймворк модульного тестування для .NET-платформи. Його переваги полягають у підтримці автоматизованих сценаріїв запуску, високій сумісності з CI/CD-процесами та інтеграції з інструментами збору метрик покриття [21].

xUnit побудований на основі принципу Fact/ Theory, який дозволяє створювати як статичні, так і параметризовані тести. Такий підхід зручний для дослідження генеративних моделей, оскільки надає змогу перевіряти не окремі результати, а множини вхідних комбінацій, згенерованих штучним інтелектом [21].

Додатковими перевагами xUnit є:

1. ізольованість середовища виконання тестів − кожен тест виконується у власному контексті, що забезпечує чистоту результатів;
2. підтримка асинхронності − важливо для сучасних сервісних програм, які активно використовують async/await;
3. гнучка система фікстур (Fixtures), що дозволяє ініціалізувати загальні залежності, не дублюючи код;
4. інтеграція з Coverlet та ReportGenerator, які використовуються для збору та візуалізації метрик покриття.

3.3 Організація тестового середовища

Для кожного відібраного проекту створюється окремий тестовий модуль, що оформлюється як незалежний .NET-проект типу xUnit Test Project. Такий підхід забезпечує ізоляцію від основного коду та дозволяє запускати тести автономно, без втручання в логіку основного застосунку.

Запуск тестів здійснюється двома способами:

1. через командний рядок за допомогою утиліти dotnet test, що підтримує передачу параметрів збірки, рівнів логування й шляхів до звітів;

2. через сценарії автоматизації (скрипти), які забезпечують послідовний запуск генерації тестів, виконання й збір метрик у єдиному циклі.

Результати виконання тестів збираються у форматі TRX або XML, що дає можливість інтегрувати їх у подальший процес аналітичної обробки.

3.4 Автоматична генерація тестів

Генерація тестових сценаріїв здійснюється за допомогою інструментів на основі штучного інтелекту − GitHub Copilot та GPT-моделей, інтегрованих у середовище Visual Studio та VS Code. Ці системи аналізують код за принципами методу «білого ящика», визначаючи гілки, цикли та логічні умови, які потребують перевірки. На основі цього формується початковий набір тестів, орієнтований на досягнення максимально можливого рівня покриття.

Згенеровані тести доповнюються структурами ініціалізації, загальними фікстурами, перевірками (Assert) і винятковими сценаріями, що відповідають вимогам до модульного тестування. Додатково впроваджується механізм data-driven testing − тестування на основі наборів даних, що дозволяє оцінити поведінку системи при різних вхідних параметрах.

3.5 Система збору метрик покриття

Для збору кількісних показників використовується Coverlet − інструмент аналізу покриття коду, що інтегрується з xUnit. Coverlet дозволяє вимірювати:

1. Line Coverage − відсоток виконаних рядків коду;
2. Branch Coverage − охоплення логічних гілок;
3. Condition Coverage − оцінку умовних виразів.

Отримані дані експортуються у форматі XML, після чого обробляються за допомогою ReportGenerator, який формує інтерактивні HTML-звіти.

Візуалізація результатів включає структуру проекту, кількість тестів, відсоток покриття для кожного файлу й класу, а також графічне виділення непокритих ділянок коду [22].

Такі звіти дозволяють дослідити не лише кількість охопленого коду, але й простежити закономірності між структурою програми та складністю досягнення повного покриття.

3.6 Верифікаційний аналіз і підтримка формальних методів

Для підвищення точності оцінки було реалізовано додатковий модуль аналітичної обробки, що дозволяє зіставляти кількісні метрики покриття з формальними моделями поведінки коду. На цьому етапі використовуються блок-схеми, діаграми класів і послідовностей, які будуються на основі вихідного коду.

Формальна верифікація проводиться за спрощеними принципами аналітичного тестування − шляхом перевірки узгодженості між покритими шляхами виконання та логічними залежностями, описаними в формальних моделях. Це дає можливість визначити не лише «кількість» виконаного коду, а й «якість» тестового охоплення з погляду структурної логіки.

3.7 Автоматизація процесу

Усі етапи − від генерації тестів до побудови звітів − об’єднані в єдиний конвеєр, який можна запускати автоматично через командний рядок або системи безперервної інтеграції (наприклад, GitHub Actions чи Azure DevOps Pipelines) [29, 30].

Сценарії автоматизації забезпечують:

1. компіляцію тестових проектів;
2. послідовний запуск тестів;
3. збір результатів покриття;
4. формування звітів та їх експорт у форматі HTML.

Завдяки цьому система дозволяє багаторазово повторювати експерименти з однаковими умовами, що є необхідною вимогою для достовірності наукових результатів.

Додатково створюється програма для напівавтоматичного створення тестів та збору метрик, яка дозволяє оператору:

1. контролювати генерацію тестів вручну або частково автоматично;
2. обирати моделі генерації коду та перевіряти якість створених тестів;
3. збирати кількісні метрики по покриттю та результатам виконання тестів;
4. формувати зведені звіти для подальшого аналізу.

Такий підхід дозволяє оператору зберігати контроль над процесом, оскільки не всі генеративні моделі здатні створювати коректний і виконуваний код тестів. Багаторазове повторення експериментів у однакових умовах забезпечує достовірність наукових результатів.

3.8 Інструментальна програма для напівавтоматичної генерації тестів

У рамках дослідження створено інструментальну програму для напівавтоматичної генерації тестів, збору метрик покриття коду та формування звітів.

Призначення програми:

1. автоматизувати процес створення тестів для існуючих проектів;
2. забезпечити збір кількісних метрик покриття коду;
3. формувати звіти у зручному форматі для подальшого аналізу;
4. надати оператору контроль за виконанням конвеєра та якістю генерованого коду.

Архітектурні особливості:

1. реалізація як WinForms (.NET) додаток;
2. структура типова для MVP/MVC: розділення логіки генерації тестів, UI та формування звітів;

3. поля для вибору проекту, директорій для тестів і звітів, а також прапорці для активації етапів конвеєра;
4. кнопка запуску конвеєра, яка послідовно виконує обрані етапи;
5. журнал виконання для відстеження стану процесу.

Використані технології:

1. .NET 6/7;
2. WinForms для користувацького інтерфейсу;
3. xUnit для виконання тестів;
4. Coverlet для збору метрик покриття;
5. ReportGenerator для формування HTML-звітів;
6. інтеграція з генеративними AI-системами (Copilot / GPT) для автоматизації створення тестів.

Програма є напівавтоматичною та керованою оператором, що дозволяє контролювати процес та уникати помилок при роботі з кодом, який генерується AI-моделями.

Висновки до розділу 3

Розроблений інструментальний комплекс забезпечує повний цикл автоматизованого дослідження якості покриття коду. Поєднання xUnit [30], Coverlet, ReportGenerator та генеративних систем на основі AI (Copilot / GPT) створює універсальне середовище, у якому можна відтворити всі основні практики сучасного тестування:

1. структурний аналіз коду;
2. автоматичну побудову тестів;
3. виконання тестів у відокремленому середовищі;
4. збір кількісних метрик і формування звітів;
5. порівняння результатів із формальними моделями логіки.

Такий підхід дозволяє не лише оцінити потенціал генеративних технологій у створенні якісних тестів, але й надати науково обґрунтовану базу

для подальших досліджень у сфері аналітичного тестування програмного забезпечення.

4 ПІДГОТОВКА ДОСЛІДЖЕННЯ

4.1 Вибір методу дослідження

Оскільки дослідження присвячене оцінці якості покриття коду тестами, згенерованими за допомогою інструментів штучного інтелекту, методика включає практичну частину, засновану на аналізі різних фрагментів програмного коду. Для перевірки ефективності таких інструментів обираються окремі ділянки коду, що відрізняються за своїм призначенням і складністю – від алгоритмічних компонентів до елементів бізнес-логіки. Такий підхід дає змогу оцінити, наскільки згенеровані тести здатні забезпечити повноту покриття різних типів функціональності.

Методика дослідження передбачає емпіричний аналіз, який полягає у створенні тестових наборів за допомогою генеративних моделей та подальшому зборі й порівнянні метрик покриття коду. Для цього застосовуються допоміжні інструменти, що дозволяють автоматично генерувати тестові дані, виконувати тести й збирати статистику результатів їх виконання.

У межах експерименту аналізується:

1. повнота покриття коду тестами (рівень виконання рядків, гілок, умов);
2. якість згенерованих тестів у контексті адекватності перевірок;
3. трудомісткість підготовки тестових сценаріїв;
4. узгодженість результатів при зміні контексту виконання.

Для виконання дослідження обрано підхід, заснований на використанні XUnit-фреймворку, який є стандартом у екосистемі .NET і дозволяє організувати структуру тестів у вигляді ізольованих методів із передбачуваними результатами. XUnit підтримує атрибутивну модель опису тестів, що спрощує інтеграцію з інструментами автоматичного аналізу й CI/CD-системами.

Крім того, у роботі розглядаються підходи до автоматичної генерації тестів за допомогою інструментів на основі штучного інтелекту (зокрема, GitHub

Copilot), які формують вихідний тестовий код на підставі контексту функції або класу. Це дозволяє оцінити різницю між людською логікою побудови тестів і машинною евристикою, а також перевірити, наскільки автоматичні методи здатні відтворити інваріанти бізнес-логіки без втручання розробника.

4.2 Використані інструменти

Для виконання дослідження застосовується програмне середовище Microsoft .NET 8 із мовою програмування C#. У якості тестового фреймворку обрано XUnit, що підтримує механізми виконання паралельних тестів, параметризованих викликів і фікстур.

Додатково використовуються інструменти для автоматизації створення тестів, збору метрик покриття коду та формування аналітичних звітів.

Платформа .NET 8 − це сучасне середовище розробки, яке підтримує кросплатформене виконання, високу продуктивність і інтеграцію з широким спектром бібліотек [6].

У межах цього дослідження .NET 8 використовується для реалізації веб-API сервісу прогнозування погоди, а також для організації тестового середовища, сумісного з фреймворком XUnit і системами збору метрик (Coverlet, ReportGenerator).

Перевагою даної платформи є її модульна архітектура, що дозволяє ізолювати бізнес-логіку, сервісні компоненти й тестові проекти, забезпечуючи чистоту експерименту при порівнянні результатів ручного та автоматизованого тестування.

Мова C# була обрана завдяки своїй строгій типізації, об’єктно-орієнтованій природі та глибокій інтеграції з інструментами екосистеми .NET. Для дослідження важливим є не лише зручність у реалізації бізнес-логіки, а й можливість аналітичного тестування через доступ до внутрішніх структур програми. C# забезпечує підтримку відображення типів (reflection), що дозволяє інструментам аналізу покриття коду ефективно збирати інформацію про виконувані ділянки, методи та класи під час тестування.

XUnit − це сучасний фреймворк для модульного тестування у середовищі .NET, що базується на принципах інверсії керування та атрибутивному підході до визначення тестів. Його особливістю є мінімалістичність і гнучкість, а також відсутність необхідності у складних конфігураціях для виконання тестових наборів.

Фреймворк підтримує:

1. ізольоване виконання тестів у паралельному режимі;
2. параметризовані тести;
3. механізм фікстур для підготовки контексту тестового середовища;
4. інтеграцію з CI/CD-платформами.

Використання XUnit дозволяє легко порівнювати результати виконання тестів, згенерованих вручну та автоматично, у єдиному форматі звітів [6, 7].

GitHub Copilot − це інструмент з використанням штучного інтелекту, спільно розроблений компаніями GitHub та OpenAI для допомоги програмістам у процесі написання коду. Він інтегрується з популярними середовищами розробки, зокрема Visual Studio Code, Visual Studio, JetBrains IDE та Neovim, і надає можливість автоматичного доповнення коду, генерації шаблонів функцій і навіть написання тестів.

Основою Copilot є модель OpenAI Codex − модифікована версія архітектури GPT-3, спеціально навчена на відкритих репозиторіях GitHub, які містять сотні гігабайтів реального вихідного коду. Copilot аналізує контекст коду, коментарі й сигнатури функцій, після чого пропонує варіанти коду або тестових методів, що підходять для поточної задачі. Цей інструмент використовується у дослідженні для створення автоматично згенерованих тестів, які потім порівнюються з ручними зразками, щоб оцінити рівень адекватності та повноти покриття [2, 8].

Coverlet − це утиліта з відкритим кодом, яка виконує збір метрик покриття коду для проектів .NET. Вона інтегрується з фреймворком XUnit та

формує результати у вигляді XML-звітів формату Cobertura, що дозволяє зчитувати й обробляти дані іншими інструментами. Для її використання проект модульних тестів повинен мати підключений пакет coverlet.collector або coverlet.console (через NuGet), після чого під час виконання тестів автоматично обчислюються показники покриття рядків, гілок і методів. Цей інструмент забезпечує основу для кількісного аналізу, дозволяючи об’єктивно оцінити, які частини програмного коду були протестовані [6, 7, 23].

ReportGenerator − це інструмент для візуалізації результатів аналізу покриття, який перетворює XML-звіти (зокрема формату Cobertura) у зручні HTML-документи з графічним відображенням статистики. Завдяки ReportGenerator можна побачити, які саме класи, методи або рядки коду були виконані під час тестування, а які залишилися неохопленими. Отримані звіти використовуються для порівняльного аналізу ручних і автоматично згенерованих тестів, оскільки вони дозволяють візуально оцінити структуру покриття та виявити ділянки із недостатньою кількістю перевірок.

Такий набір інструментів дозволяє не лише зібрати сирі метрики, але й інтерпретувати їх у контексті архітектури системи, виявляючи ділянки, які залишилися непокритими тестами або перевірені недостатньо детально.

4.3 Загальна характеристика програмних модулів та їх формалізація

Для експериментального дослідження було створено проект типу Class Library, у якому розміщені різні програмні модулі з відкритих репозиторіїв Git. Ці модулі виступають об’єктами тестування та дозволяють оцінити ефективність генерації та виконання тестів для різних алгоритмічних сценаріїв.

Серед розглянутих модулів виділяються:

1. HammingLab — модуль, який реалізує обчислення зовнішнього добутку векторів, додавання матриць та множення матриці на вектор. Цей код включає складну комбінацію циклів, вкладених обчислень та операцій з масивами, що робить його репрезентативним для тестування покриття.

2. Balanced — модуль для перевірки збалансованості дужок у рядку. Містить просту роботу зі стеком і умовними перевірками; цей модуль підходить для базового покриття та демонстрації генерації тестів.

3. Duplicates — модуль, що видаляє повторювані елементи з масиву. Містить колекції HashSet та List; хоча алгоритм простий, він включає операції додавання та перевірки, що мають сенс для тестування покриття.

4. Frequency — модуль підрахунку частоти символів у рядку. Алгоритм містить ітерацію по елементам рядка та умовне оновлення словника, що робить його придатним для покриття базовими тестами.

5. Palindrome — модуль перевірки, чи є рядок паліндромом. Простий алгоритм з циклом та умовною перевіркою; може бути покритий тестами з базовими граничними випадками.

6. Search — модуль бінарного пошуку в відсортованому масиві. Алгоритм містить циклічну перевірку з умовами, що дозволяє аналізувати контрольні точки виконання.

7. SortedAscending — модуль перевірки, чи відсортований масив за зростанням. Простий лінійний алгоритм, який можна тестувати базовими сценаріями.

8. Subarray — модуль пошуку максимальної суми підмасиву (алгоритм Кадане). Включає цикли та умовні операції з накопиченням значень, що робить його репрезентативним для аналізу покриття.

Таким чином, у проєкті присутні модулі різного рівня складності:

1. ті, що необхідно покрити тестами через наявність циклів, умов, рекурсії та складних структур даних (HammingLab, Search, Subarray, QuickSort аналоги);

2. і ті, що мають простий код і не вимагають складного покриття, але корисні для демонстрації генерації тестів та базового аналізу (Balanced, Duplicates, Frequency, Palindrome, SortedAscending).

Формалізація коду здійснюється шляхом перетворення його на абстрактне структурне подання, яке дозволяє будувати графи потоку керування

(CFG) і визначати ділянки, найбільш схильні до помилок, для подальшого автоматизованого тестування.

4.4 Кандидати на тестування та принципи побудови тестів

Одним із центральних етапів експерименту є процес ідентифікації кандидатів для тестування. У класичній теорії тестування, що мала значний розвиток у межах досліджень Microsoft та спільнотою LLVM, програма розглядається як множина базових блоків. Теоретична коректність перевірки досягається лише тоді, коли всі оператори, умови, переходи та цикли були виконані хоча б один раз. На практиці це означає, що тестування має охоплювати всі суттєві фрагменти CFG, включаючи рідкісні або нетривіальні шляхи.

Вибір кандидатів базується на аналізі операторів та побудові умов, які дозволяють активувати альтернативні гілки виконання. Для QuickSort такими кандидатами є, наприклад, гілки порівняння елементів, оператори переміщення покажчиків, розподіл масиву та рекурсивні виклики. Для BFS — це умови перевірки наявності суміжних вузлів, обробка вже відвіданих вершин, логіка черги та порядок обходу.

Сучасні ІІ-системи здатні синтезувати тести, використовуючи подібні принципи. Генерація тестів базується на тому, що модель ідентифікує контрольні точки програми — місця, в яких поведінка може змінюватися залежно від вхідних даних. Модель використовує семантичні патерни й структурні інваріанти, щоб сформувати набір тестів, які стимулюють програму пройти максимальну кількість незалежних шляхів. Наприклад, для QuickSort генерується набір вхідних масивів із типово впорядкованими, зворотно впорядкованими, повторюваними та хаотичними значеннями, що дозволяє активувати різні траєкторії рекурсивного розбиття. Для BFS можуть створюватися графи з ізольованими вершинами, деревоподібні структури, графи із циклами або з високою розгалуженістю.

Таким чином, ІІ-генератори тестів не лише перевіряють правильність отриманого результату, а й забезпечують семантичне покриття, що є суттєво

глибшим за традиційні підходи, орієнтовані на механічну активацію інструкцій. Оскільки модель аналізує код як семантично пов’язаний об’єкт, вона здатна формувати тести, які підтверджують або спростовують визначені інваріанти виконання. У результаті створюється повноцінна система верифікації, що поєднує структурний та змістовний підходи.

4.5 Організація процесу виконання модульних тестів

Усі згенеровані за допомогою мовних моделей, ChatGPT 5.1 (повсякденна версія) та Copilot Smart GPT-5, тести були реалізовані на основі фреймворку xUnit. Вони розміщуються у окремих проектах, які віддзеркалюють назву моделі, що їх згенерувала. Така структура дозволяє в майбутньому виконувати порівняльний аналіз. Виконання тестів відбувалося за допомогою стандартного тестового раннера .NET SDK, що дозволяє автоматизувати збір метрик та повторюваність експерименту.

Результати виконання тестів використовувалися не лише для отримання інформації про коректність роботи модулів, а й для генерації формальних показників покриття. Це дозволяє оцінити внутрішню повноту тестів, різницю у поведінці алгоритмів під час тестування, а також виявити потенційні зони невизначеності або недостатнього охоплення.

4.6 Збір метрик покриття та особливості їх інтерпретації

Для аналізу якості тестового набору використовувався інструмент Coverlet, що інтегрується з MSBuild [24] та xUnit. Coverlet формує результати у стандартизованому XML-поданні, яке включає кілька ключових показників.

Першим з них є покриття рядків, яке визначає частку виконаних інструкцій. Цей показник відображає загальну активність прогону, але не завжди враховує специфіку альтернативних гілок. Другим є покриття гілок, яке показує, чи були активовані всі логічні розгалуження. Для алгоритмів із багатьма умовами цей показник є критичним, оскільки саме він визначає, наскільки повно охоплена логічна структура. Третім є покриття методів, яке демонструє, чи вдалося тестам

активувати кожну вершину високого рівня в графі викликів. Нарешті, значущим показником є цикломатична складність, що визначає кількість незалежних шляхів у програмі. Для складних алгоритмів, таких як QuickSort, цей параметр може бути досить високим, що вимагає пропорційно більшу кількість тестів [25].

Ці метрики дозволяють розглядати тестування як процес, що може бути строго формалізований, а не як набір інтуїтивних практик. Порівнюючи їх між різними моделями ІІ у майбутніх експериментах, можна буде встановити, які з них створюють більш повні та ефективні сценарії [26, 27].

4.7 Побудова аналітичного дашборду та інтерпретація результатів

Після отримання первинних метрик файли з даними були передані інструменту ReportGenerator, який здійснює агрегування показників і створення HTML-дашборду. Даний дашборд є ключовим елементом експерименту, оскільки забезпечує наочне представлення результатів. Він дозволяє аналізувати покриття на рівні проекту, простору імен, класу та окремого методу, а також містить візуальні інструменти, такі як теплові карти, діаграми й дерева складності. Завдяки цьому дослідник отримує цілісну картину стану тестування, може виявляти прогалини та здійснювати порівняння між різними тестовими наборами.

ReportGenerator перетворює XML-дані Coverlet на внутрішній об’єкт, який агрегує всі показники, обчислює відсоткові співвідношення та структурує інформацію. Графічні представлення дозволяють миттєво оцінити критичні ділянки та визначити сегменти, що потребують додаткового охоплення.

4.8 Результати експерименту

Програмна інженерія як науково-прикладна дисципліна пройшла тривалий шлях розвитку − від процедурного програмування до сучасних моделей, орієнтованих на верифікацію, тестування й доведення коректності.

У процесі еволюції було сформовано низку методів формального опису програм, серед яких логіко-математичні моделі, графи потоку управління, блок-схеми та інші візуальні представлення.

Такі моделі дозволяють розглядати програму не лише як послідовність інструкцій, а як систему переходів між станами, де кожен фрагмент коду виконує певну функціональну роль.

У контексті даного дослідження тестове покриття розглядається як відображення реального виконання програми на її абстрактну модель. Кожна метрика покриття (рядків, гілок, методів) може бути інтерпретована як часткове відображення від множини усіх можливих шляхів виконання до множини фактично протестованих. Це дозволяє не лише оцінити кількісні показники, а й побудувати структурну модель якості тестування − своєрідну блок-схему, у якій лінії (Line Coverage) відповідають виконаним інструкціям, розгалуження (Branch Coverage) − перевіреним логічним переходам, а вузли (Method Coverage) − активованим функціональним блокам програми.

З практичного боку результати експерименту базуються на кількісних метриках, отриманих за допомогою інструментів Coverlet і ReportGenerator. Основними показниками є:

1. Line Coverage − частка виконаних інструкцій від загальної кількості рядків коду, що беруть участь у виконанні;
2. Branch Coverage − частка перевірених умовних переходів, що демонструє рівень охоплення логічних розгалужень;
3. Method Coverage − співвідношення протестованих методів до їх загальної кількості.

Для кожного класу або модуля формується окрема статистика, що відображає, наскільки повно тести охоплюють поведінку системи на різних рівнях абстракції.

Отримані дані представляються у вигляді HTML-звіту, який дозволяє візуально співвіднести елементи вихідного коду з рівнем їхнього покриття.

Графічна інтерпретація нагадує блок-схему виконання, у якій кольорове маркування показує, які ділянки коду були активовані під час тестування (виконані), а які залишились неохопленими.

Таким чином, результати експерименту демонструють не лише кількісні аспекти (відсоток покриття), але й якісні − структуру виконання програми, її логічну цілісність і ступінь верифікованості через тестування.

Порівняння ручно створених тестів із автоматично згенерованими дає змогу виявити закономірності у способі охоплення коду, визначити слабкі ділянки тестової логіки та зробити висновки щодо ефективності генеративних методів.

Висновки до розділу 4

Отримані метрики формують аналітичну основу для подальшої оцінки ефективності автоматично згенерованих тестів. На цьому етапі важливо не лише зафіксувати числові значення покриття, але й інтерпретувати їх у контексті структури програми: які саме логічні ділянки були охоплені, наскільки повно реалізовано перевірку умов і переходів, та як це відображається на рівні формальної коректності.

Порівняння виконується не лише за кількісними показниками − відсотком покриття, кількістю виявлених дефектів чи часом виконання тестів, − а й з використанням формальних методів аналітичної верифікації. Такий підхід передбачає застосування елементів математичної логіки для перевірки відповідності між структурою покриття (за даними coverage-метрик) і фактичною якістю тестування, тобто здатністю тестів підтверджувати коректність роботи логічних залежностей у коді.

Для інтерпретації кількісних результатів застосовується візуальна аналітична звітність − HTML-дашборд, згенерований на основі XML-звітів (наприклад, із xUnit або Coverlet). Такий формат дозволяє зіставляти кількісні метрики покриття з формалізованими моделями логіки програми, аналізувати відповідність між рівнем покриття та реальним ступенем перевірки поведінки системи.

Оцінка якісних аспектів тестування проводиться окремо − на основі аналізу формальних методів опису та верифікації програм, які моделюють потік виконання та взаємодію між компонентами у відокремлених контекстах. Порівняння кількісних метрик покриття з результатами формальної верифікації дозволяє встановити, наскільки згенеровані тести охоплюють ключові поведінкові сценарії системи та забезпечують узгодженість між рівнем покриття та структурною логікою програми.

Узагальнення результатів верифікації дозволяє оцінити, наскільки генеративна модель на основі ШІ здатна не лише забезпечити високі показники покриття, але й підтвердити якісну коректність коду через логічну повноту та узгодженість тестових сценаріїв.

5 ОПИС ЕКСПЕРИМЕНТУ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

У процесі дослідження було розглянуто можливості сучасних засобів автоматизованого тестування в екосистемі .NET, із фокусом на генеративні моделі штучного інтелекту та інструменти збору метрик покриття. Основна увага приділялася формалізації вихідного коду, визначенню критеріїв вибору модулів для тестування, а також побудові модульних тестів на основі моделей ChatGPT 5.1 та Copilot Smart GPT-5. Отримані показники покриття, зібрані засобами Coverlet і ReportGenerator, стали базою для аналізу точності, повноти та якості тестових наборів.

Концептуально дослідження спирається на припущення про те, що якість модульного тестування можна визначати незалежно від людського фактора, використовуючи суто формальні метрики: покриття рядків і гілок, цикломатичну складність та активовані шляхи виконання. Для оцінки роботи тестових генераторів були обрані програмні модулі різної природи — від детермінованих алгоритмів до елементів бізнес-логіки, що дозволило сформувати репрезентативну модель поведінки програмних систем.

5.1 Опис процесу проведення експериментального дослідження

У межах експериментального дослідження використовувалися програмні фрагменти, що репрезентують різні рівні структурної та семантичної складності програмного коду. До складу експериментальної вибірки увійшли алгоритмічні реалізації класичних задач сортування та пошуку, модулі з елементами доменної логіки, а також компоненти, що здійснюють обробку колекцій, перетворення даних та оновлення внутрішнього стану об’єктів.

Окрему групу становили утилітарні функції, орієнтовані на лінійну обробку вхідних даних, які не містять складних розгалужень, але є корисними для оцінки базового рівня тестового покриття. Загалом у межах експерименту використовувалося близько двадцяти програмних модулів, розміщених у

каталозі *Source*. Приклад файлової структури проекту наведено на відповідному рисунку.

Вихідний програмний код представлено у вигляді окремих файлів мовою C#, отриманих із відкритих публічних репозиторіїв. Кожен файл містить завершену реалізацію одного логічного модуля (наприклад, перевірка збалансованості дужок), який розглядається як незалежний об’єкт тестування.

5.2 Організація процесу генерації та виконання модульних тестів

На основі зазначених програмних модулів здійснювалася автоматизована генерація модульних тестів за допомогою спеціалізованої користувацької програми. Програма реалізує напівавтоматичний підхід і дозволяє оператору вказати шлях до файлу програмного коду, обрати генеративну модель та ініціювати процес створення тестів.

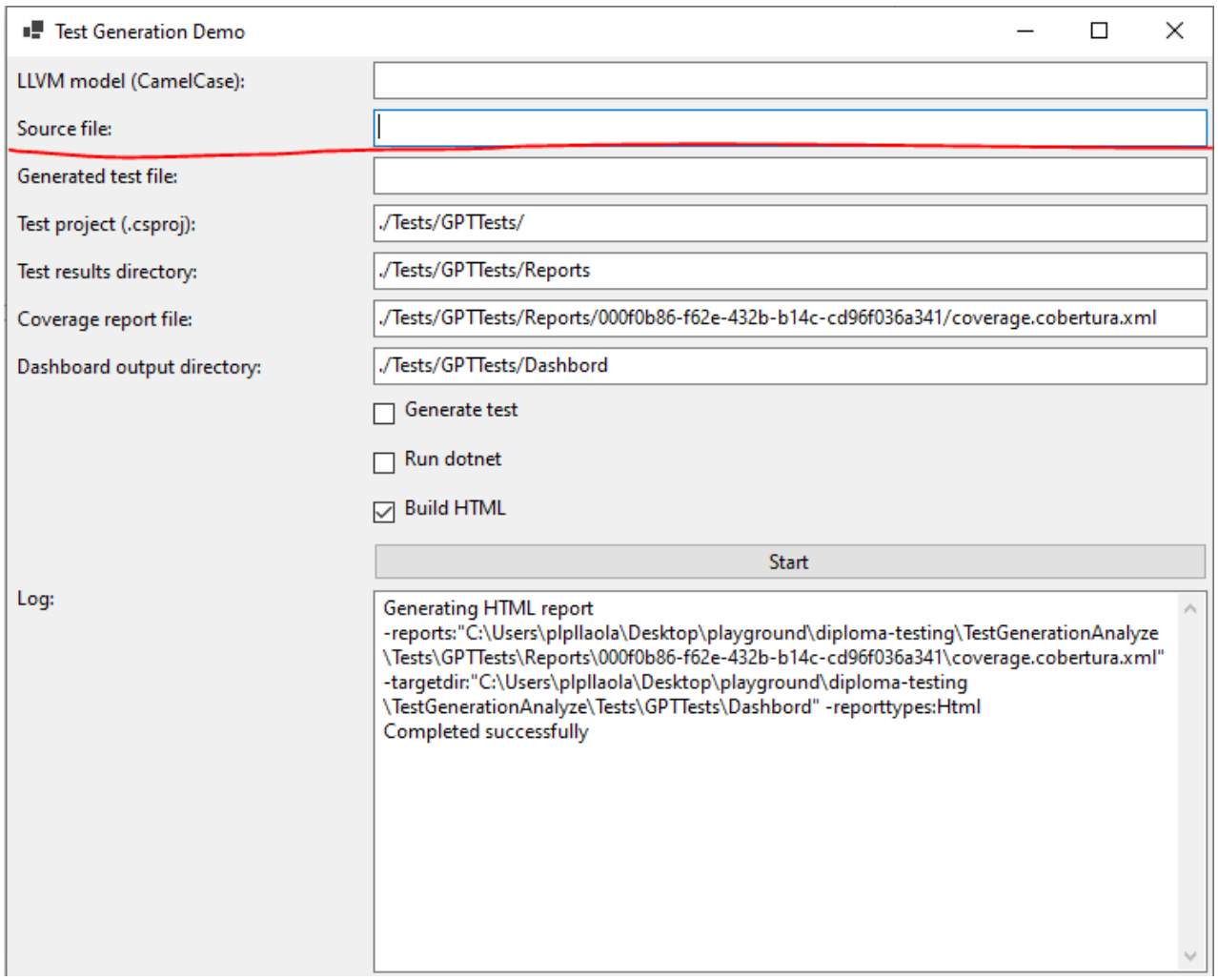

Рисунок 5.1 – Інтерфейс програми генерації для визначення файла з кодом

Усі тести, згенеровані моделями ChatGPT-5.1 та Copilot Smart GPT-5, були реалізовані із використанням фреймворку xUnit і розміщені в окремих тестових проектах, ідентифікованих за назвою моделі-генератора.

Така сегментація забезпечила ізольованість тестових наборів і створила умови для коректного порівняльного аналізу результатів. Файлова структура тестів організована за моделями та дозволяє однозначно співвідносити згенерований код із джерелом його походження.

Генерація тестів виконується через клієнтські виклики до відповідних генеративних моделей. За генерацію тестів відповідає рядок інтерфейсу Generated Test File з додатково заповненим рядком шляху до файла программи

на основі якого будується тест також потрібно ввести назву генеративної моделі а також прапор генерування тестів.

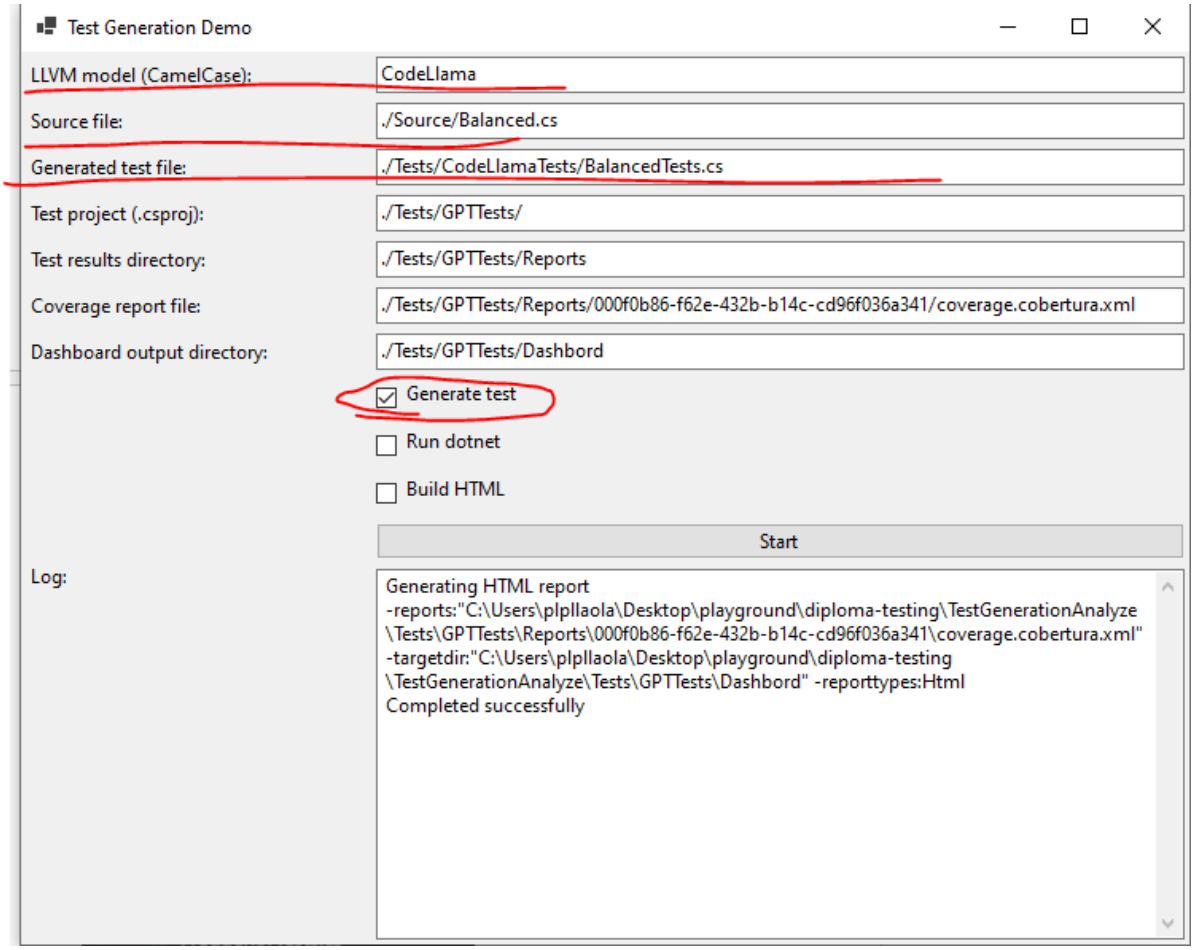

Рисунок 5.2 − Інтерфейс програми генерації для генерації тестів

Для генерації тестів використовувалась програма що використовує виклики до генеративних моделей у вигляді клієнтів.

У разі, якщо згенерований код містить синтаксичні або логічні помилки, оператор може змінити вхідні інструкції (prompts), обрати іншу модель або, за необхідності, виконати мінімальні правки для забезпечення коректного запуску тестів. Такий підхід обумовлений тим, що не всі моделі гарантують генерацію повністю працездатного тестового коду.

5.3 Організація процесу виконання модульних тестів

Виконання модульних тестів здійснювалося за допомогою стандартного раннера .NET SDK, що забезпечило автоматизацію процесу тестування та повторюваність експериментальних умов. Користувацька програма містить

набір інструкцій для виконання необхідних консольних операцій, зокрема компіляції тестових проектів і запуску тестів із підключенням інструментів збору покриття.

За генерацію тестів відповідає рядок інтерфейсу Generated Test File з додатково заповненим рядком шляху до файла программы на основі якого будується тест також потрібно ввести назву генеративної моделі а також прапор генерування тестів

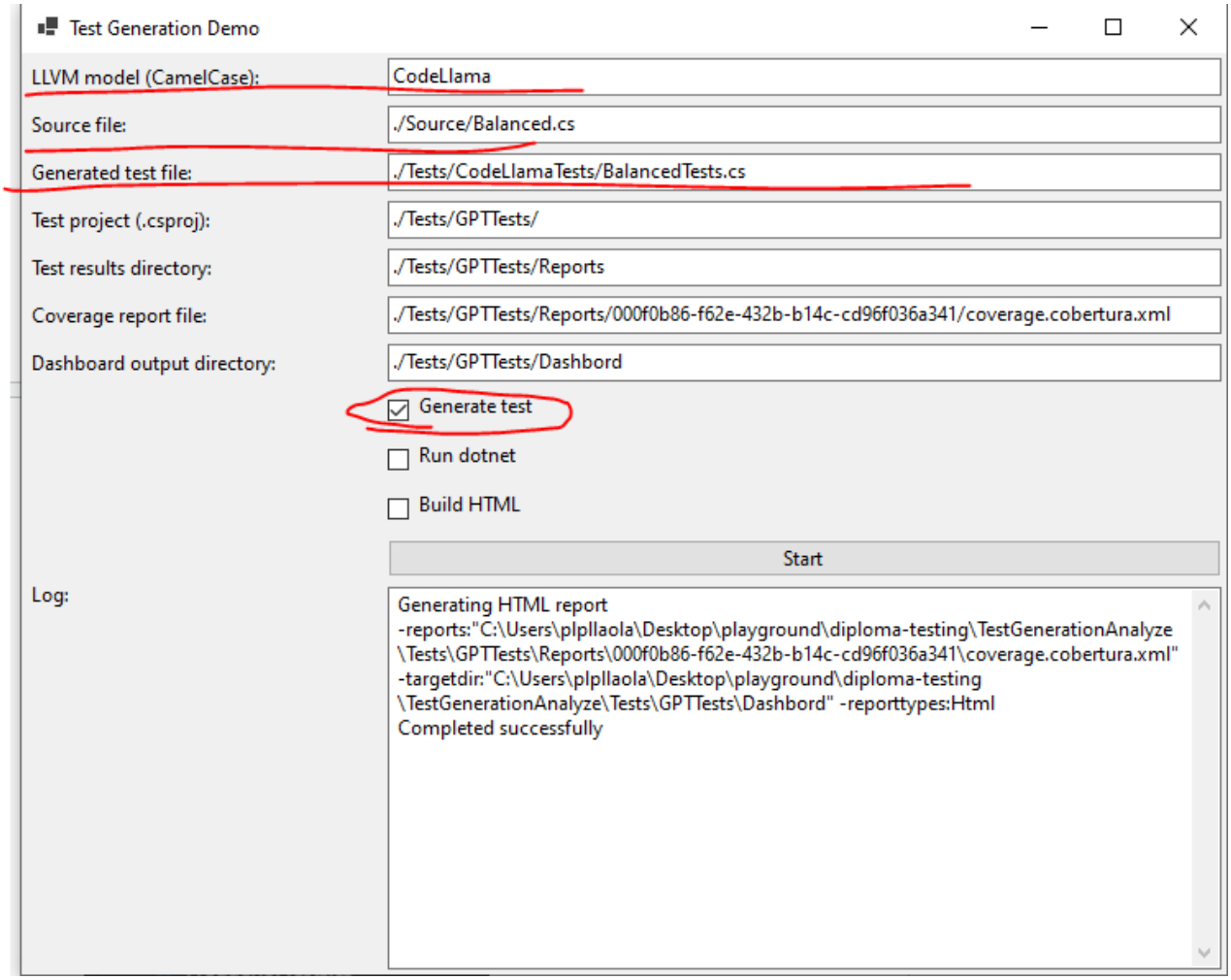

Рисунок 5.3 − Інтерфейс програми генерації для генерації тестів

У результаті чого створюється відповідний файл тесту згенерований обраною моделью

Для отримання метрик покриття оператор задає шлях до тестового проєкту відповідної моделі та ініціює запуск тестів із відповідним прапором. У результаті формується файл покриття у форматі Cobertura XML, який зберігається у каталозі результатів тестування. Для тестів, згенерованих

моделлю ChatGPT-5.1, такі файли містять детальну інформацію про покриття, включно з великою кількістю проаналізованих рядків коду.

У програмі реалізований набір інструкцій який дозволяє виконувати консольні операції

Для генерації файлу метрик покриття потрібно ввести шлях до проекта тестів моделі для якої ми хочему отримати метріки та додати флаг run dotnet

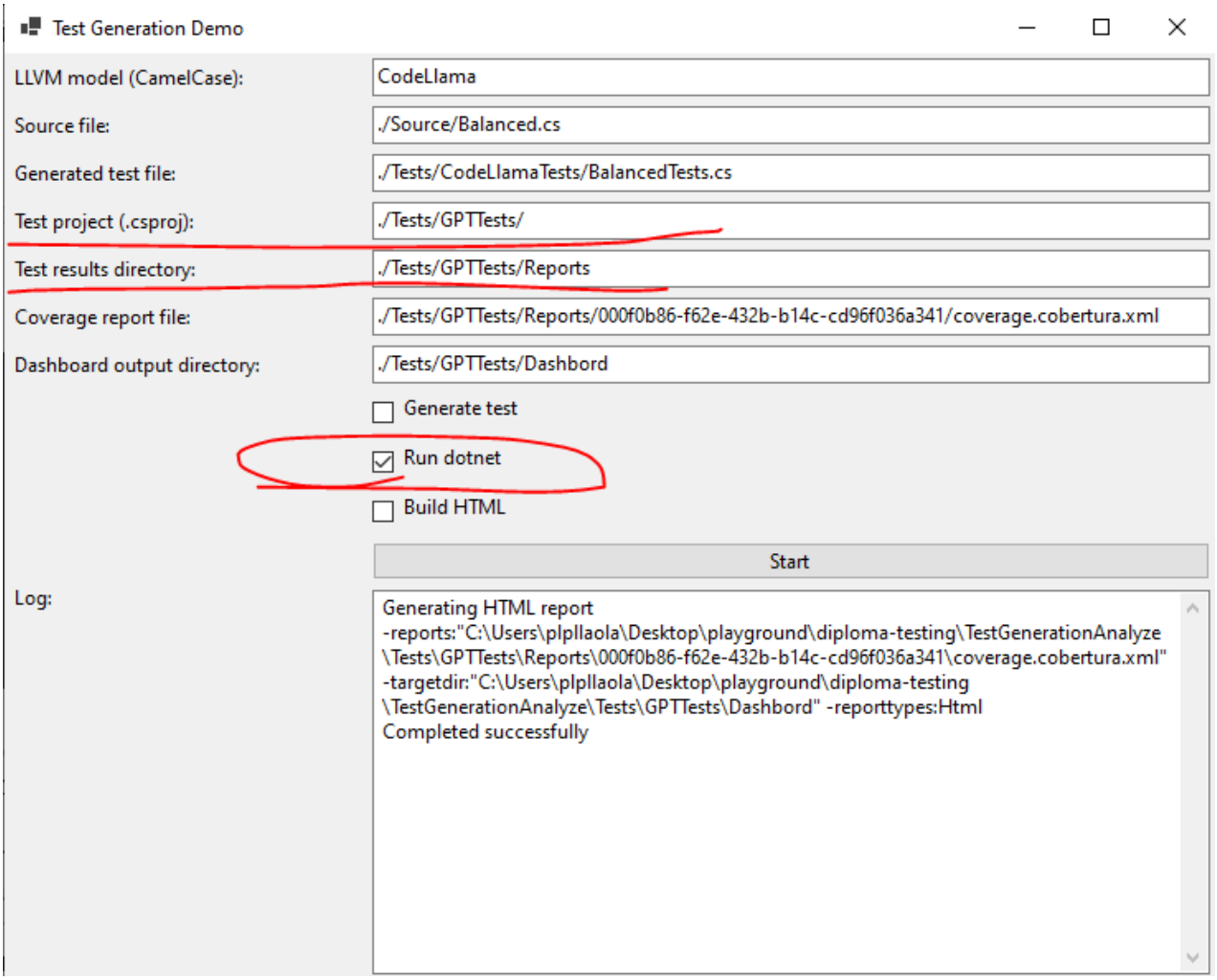

Рисунок 5.4 − Інтерфейс програми для генерації файла з метриками покриття

Це дозволяє створити файл метрик покриття по шляху вказанному у полі test results directory у вигляду файлу coverage.cobertura.xml у папці з айди генерацією.

Файл для тесті на основі chatgpt-5.1 повертає такий файл метрик покриття з кількістю рядків більше 1000.

Отримані файли метрик передавалися інструментам аналізу покриття для подальшої обробки. На їх основі формувалися узагальнені показники якості тестування, що дозволили оцінити повноту охоплення коду, особливості поведінки алгоритмів під час тестування та виявити потенційні зони недостатнього покриття.

Для генерації звітів оператор зазначає шлях до файлу покриття та каталог, у який має бути згенерований HTML-звіт. У результаті формується набір артефактів у вигляді веб-дашборда, що наочно відображає результати тестування. Дашборд дозволяє переглядати як загальні метрики покриття, так і детальне теплове представлення окремих файлів програмного коду.

Для генерації результатів потрібно вказати шлях до файлу метрик покриття та шлях куди буде генеруватись штмл дашборд а також встановити флаг генерації репорту

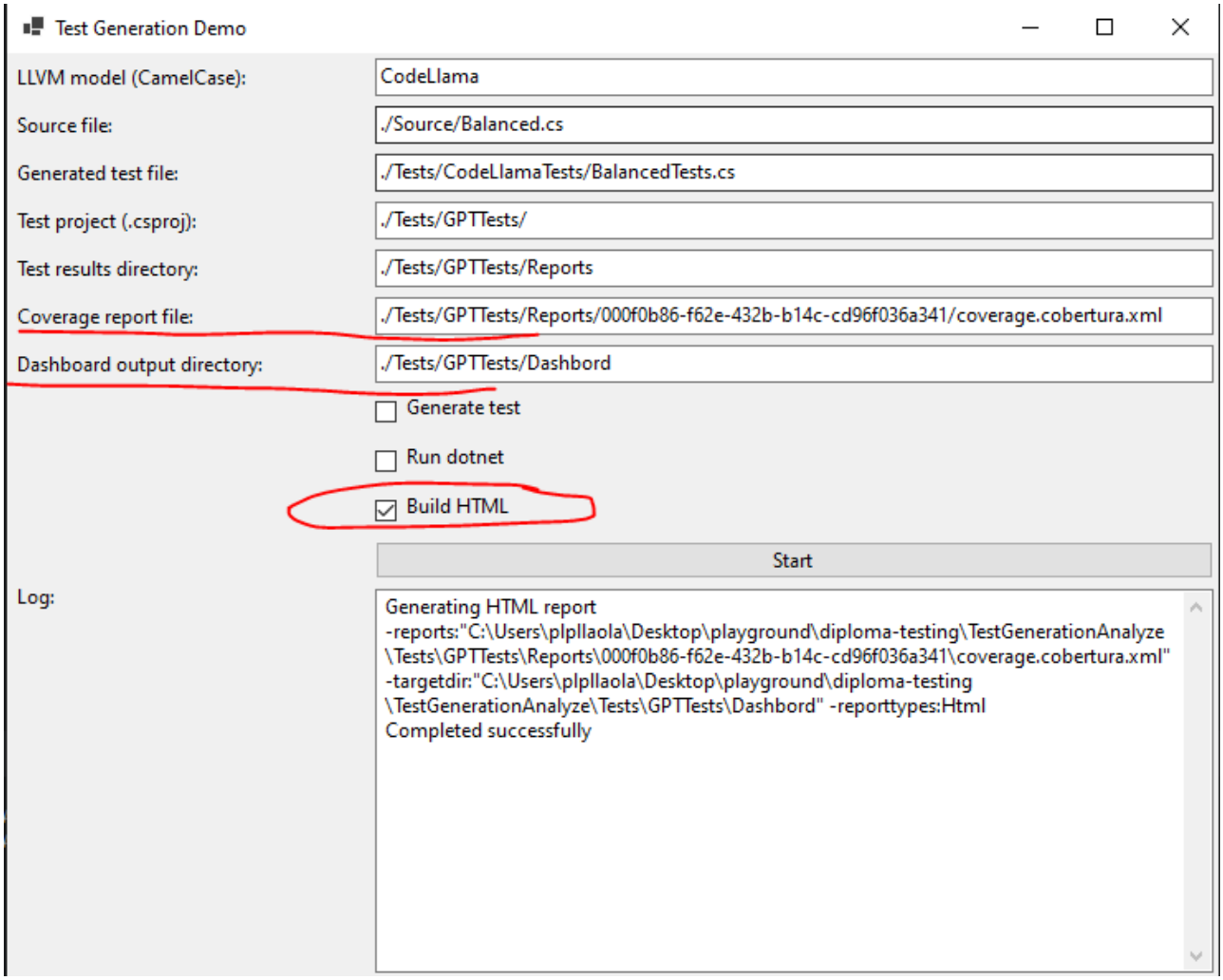

Рисунок 5.5 − Інтерфейс програми для генерації дашборда покриття

За результатом генерації отримано артефакти у вигляді файлів веб сайту дашборда по вказаному у програмі шляху.

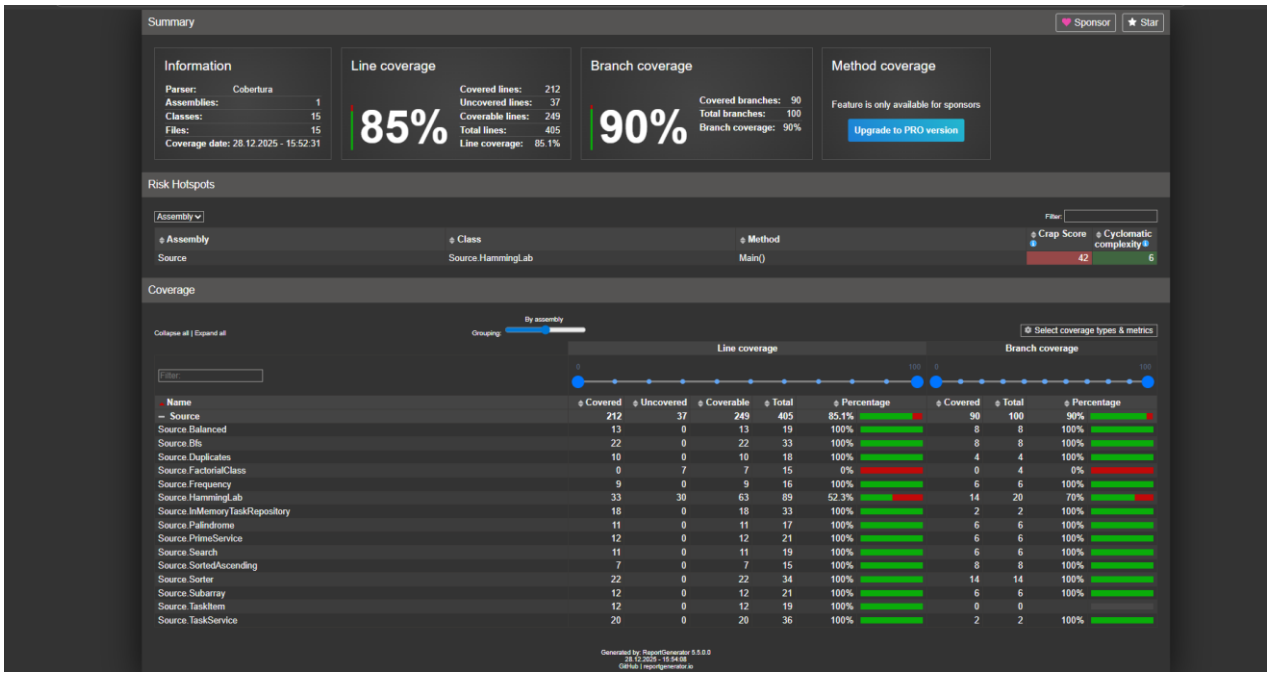

Рисунок 5.6 − Головна сторінка дашборда покриття усіх файлів

Дашборд теплового покриття окремого файла відкривається при натисканні на рядок у таблиці якості покриття по файлам

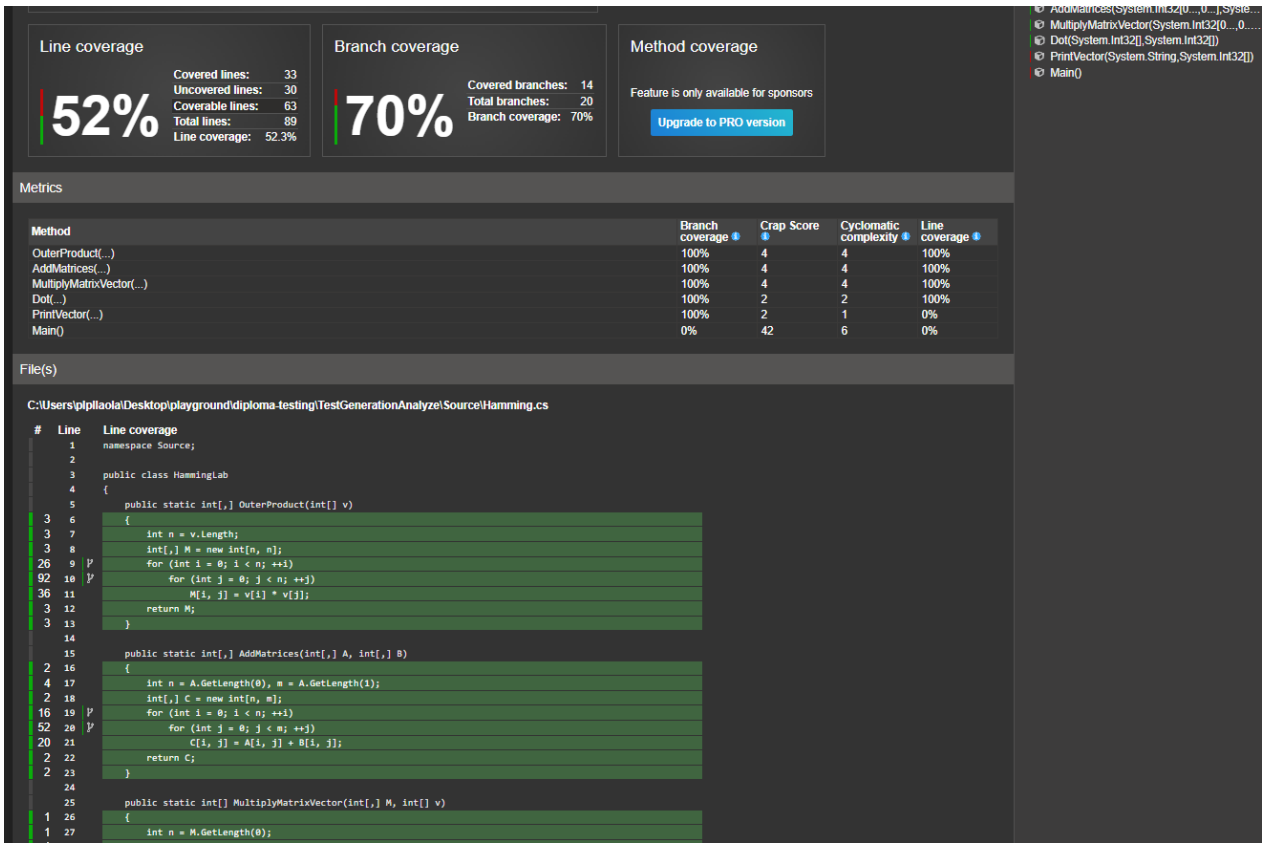

Рисунок 5.7 − Сторінка теплового покриття окремого файла

5.4 Основні результати порівняльного аналізу

В рамках проведеного дослідження було здійснено порівняльну оцінку двох мовних моделей: ChatGPT 5.1 (повсякденна версія) та Copilot Smart GPT-5. Для аналізу використовувалися програмні артефакти, зібрані за допомогою MSBuild .NET, а також звіти, сформовані засобами Report Generator.

У ході експериментального дослідження, виконаного на базі двох обмежених програмних модулів, не було виявлено відмінностей за ключовими технічними метриками: успішністю компіляції, повнотою покриття умов та загальною коректністю виконання тестів. В обох випадках — як для ChatGPT 5.1, так і для Copilot Smart GPT-5 — було досягнуто рівень покриття коду більше 80% якщо враховувати методи що не можуть бути протестовані або не мають це на меті.

Таблиця 5.1 – Результати для ChatGPT 5.1

name	line covered	line uncovered	line covarebl e	line total	line precent age	branch covere d	branc h total	branch covareble
total	212	37	249	405	85.1%	90	100	90%
balanced	13	0	22	33	100%	8	8	100%
dublicates	10	0	10	18	100%	4	4	100%
factorial	0	7	7	15	0%	0	4	0%
frequence	9	0	9	16	100%	6	6	100%
hamming	33	30	63	89	52.3%	14	20	70%
inMemoryT askReposito ry	18	0	18	33	100%	2	2	100%
palindrome	11	0	11	17	100%	6	6	100%

Таблиця 5.1 – Результати для Copilot Smart GPT-5

primeService	12	0	12	21	100%	6	6	100%
Search	11	0	11	19	100%	6	6	100%
sorter	22	0	22	34	100%	14	14	100%
sortedAscending	7	0	7	15	100%	8	8	100%
subarray	12	0	12	21	100%	6	6	100%
taskItem	12	0	12	19	100%	0	0	
taskservice	20	0	20	36	100%	2	2	100%

Таблиця 5.2 – Результати для Copilot Smart GPT-5

name	line covered	line uncovered	line covarebl e	line total	line precent age	branch covere d	branc h total	branchco vareble
total	219	30	249	405	87.9%	93	100	93%
balanced	13	0	13	19	100%	8	8	100%
dublicates	22	0	22	33	100%	8	8	100%
factorial	7	0	7	15	100%	4	4	100%
frequence	9	0	9	16	100%	6	6	100%
hamming	33	30	63	89	52.3%	14	20	70%
inMemoryTaskRepository	18	0	18	33	100%	2	2	100%

Таблиця 5.2 – Результати для Copilot Smart GPT-5

palindrome	11	0	11	17	100%	6	6	100%
primeService	12	0	12	21	100%	6	6	100%
Search	11	0	11	19	100%	6	6	100%
sorter	22	0	22	34	100%	14	14	100%
sortedAscending	7	0	7	15	100%	8	8	100%
subarray	12	0	12	21	100%	5	6	83.3%
taskItem	12	0	12	19	100%	0	0	
taskservice	20	0	20	36	100%	2	2	100%

Це свідчить про те, що обидві моделі здатні забезпечити мінімально необхідний гарантований рівень функціональної верифікації для модулів невеликої складності.

Водночас різниця проявилася у підходах до формальної побудови тестового набору.

Copilot Smart GPT-5 автоматично сформував ширший набір тестових сценаріїв, охоплюючи більшу кількість варіацій та граничних ситуацій. Це потенційно забезпечує вищу надійність тестування у випадках, коли алгоритм працює з розширеним доменом вхідних даних або містить складні схеми розгалуження.

ChatGPT 5.1, навпаки, продемонстрував ітеративний стиль: базовий набір тестів формувався на початковому етапі, а подальше розширення пропонувалося лише у відповідь на уточнення або зміну вимог. Таким чином,

модель орієнтується на оптимізацію та поступове удосконалення, що відповідає сучасним підходам до адаптивної розробки програмного забезпечення.

З точки зору методів формальної верифікації, збільшена кількість тестових сценаріїв, яку генерує Copilot Smart GPT-5, може бути сприйнята як надмірність або дублювання перевірок. У класичному тестовому аналізі таке дублювання вважається небажаним через можливе зниження ефективності та зростання обсягу супроводу тестового набору.

Однак у інженерній практиці високонадійних систем подібна надмірність частіше трактується як додатковий шар захисту. Якщо домен вхідних даних важко передбачити або якщо алгоритм містить складні рекурсії, вкладені цикли чи потенційні неявні переходи, надлишкове тестове покриття розглядається як спосіб мінімізувати ризик появи прихованих помилок. У цьому контексті Copilot демонструє більш консервативну стратегію забезпечення коректності — він прагне «запечатати» усі можливі сценарії виконання, навіть якщо частина з них дублює логіку попередніх тестів.

ChatGPT 5.1, у свою чергу, можна охарактеризувати як модель, що орієнтується на структуровану оптимізацію тестового простору. Вона не намагається одразу покрити весь можливий спектр випадків, а формує тестовий набір таким чином, щоб він відповідав сформульованим вимогам і міг масштабуватися у разі зміни або уточнення задачі. Цей підхід особливо цінний у середовищі, де тестування здійснюється в режимі швидкого прототипування або у контексті інкрементальної розробки.

Узагальнюючи, обидві моделі демонструють валідні стратегії тестового синтезу: Copilot Smart GPT-5 — як система з акцентом на повноту та консервативну надійність, а ChatGPT 5.1 — як інструмент адаптивного, контекстно орієнтованого формування тестових сценаріїв.

Висновки розділу 5

Проведений експеримент продемонстрував, що поєднання структурної формалізації коду, автоматизованої генерації тестів і детального аналізу метрик

покриття формує цілісний і надійний підхід до верифікації програмних модулів. Завдяки застосуванню ІІ-генераторів стало можливим отримати тести, що відтворюють інваріанти алгоритмів та забезпечують широке семантичне покриття. Інструменти Coverlet і ReportGenerator дозволили сформувати стандартизований набір даних і візуалізацій, що відкривають можливість для подальших порівняльних експериментів між різними моделями ШІ.

Отримані результати закладають методологічну основу для дослідження ефективності автоматичних генераторів тестів та подальшого вдосконалення систем контролю якості програмного забезпечення. При цьому необхідно формулювати правильні промти з чітким описом задачі та контексту, адже деякі помилки є допустимими — наприклад, неоднозначність класів через простори імен, особливо у випадках взаємопов’язаних програмних модулів і складних залежностей. У таких ситуаціях важливо передбачати можливість мінімальних правок оператором, якщо це потрібно для досягнення коректності та узгодженості тестів.

Обидві моделі забезпечують порівнянний рівень якості за формальними метриками складання та покриття. Тим не менш, різниця в стратегіях тестування свідчить про те, що Copilot Smart GPT-5 схильний створювати ширший набір тестів, що може бути корисним для критично важливих систем, де надійність має першочергове значення. Водночас ChatGPT 5.1 залишається ефективним інструментом для ітеративної розробки, пропонуючи вдосконалення за потреби. Таким чином, вибір між моделями може визначатися не стільки формальними метриками, скільки практичним контекстом використання: Copilot переважніше у сценаріях, де потрібна максимальна надійність, а ChatGPT — для гнучкої та адаптивної розробки.

ЗАГАЛЬНІ ВИСНОВКИ

У кваліфікаційній роботі магістра виконано комплексне дослідження сучасних підходів до автоматичної генерації тестових сценаріїв із використанням методів штучного інтелекту. Актуальність теми зумовлена зростанням складності програмних систем, підвищеними вимогами до якості програмного забезпечення та необхідністю скорочення витрат часу й ресурсів на процеси тестування.

У ході роботи проаналізовано існуючі підходи до автоматизованого тестування програмного забезпечення, зокрема методи, що базуються на генеративних нейромережевих моделях. Визначено їх переваги та обмеження з точки зору точності, повноти та релевантності згенерованих тестових сценаріїв.

Розроблено програмний застосунок для навчання, тестування та оцінювання генеративних моделей, який забезпечує автоматичну генерацію тестових сценаріїв і підтримує експериментальне порівняння результатів на різних наборах даних. Архітектура застосунку дозволяє масштабування та адаптацію до різних типів завдань контролю якості програмного забезпечення.

Проведено експериментальне дослідження ефективності генеративних моделей на різних наборах даних, що імітують типові сценарії тестування програмних систем. За результатами експериментів встановлено, що якість автоматично згенерованих тестів значною мірою залежить від структури та репрезентативності навчальних даних, а також від налаштувань і типу використаної моделі.

Отримані результати підтверджують доцільність використання генеративних моделей у процесах автоматизованого тестування, особливо на етапах первинного покриття функціональності та регресійного тестування. Застосування запропонованого підходу дозволяє підвищити ефективність контролю якості програмного забезпечення, зменшити трудомісткість тестування та забезпечити більш системний підхід до формування тестових сценаріїв.

Практичне значення роботи полягає у можливості використання розробленого програмного застосунку в діяльності команд контролю якості програмного забезпечення, а також у навчальному процесі під час підготовки фахівців у галузі інформаційних технологій. Напрямами подальших досліджень є розширення наборів даних, інтеграція гібридних моделей та вдосконалення метрик оцінювання якості згенерованих тестових сценаріїв.

СПИСОК ПОСИЛАНЬ

1. Jurafsky, D. Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition [Text] / D. Jurafsky, J. H. Martin. – 3rd ed. (draft). – 2023. – 700 p.
2. Large language model [Електронний ресурс] // Wikipedia. – Режим доступу: URL: https://en.wikipedia.org/wiki/Large_language_model. – Загл. з екрана. – Дата звернення: 02.01.2025.
3. Nielson, B. Automated Testing for Developers [Text] / B. Nielson, A. Skou // BRICS, CISS Conference. – Aarhus, 2002. – 12 p.
4. Разработка через тестирование [Електронний ресурс] // Wikipedia. – Режим доступу : URL: https://ru.wikipedia.org/wiki/Разработка_через_тестирование. – Загл. з екрана. – Дата звернення: 02.01.2025.
5. Beck, K. Test-driven development : by example [Text] / Kent Beck. – Boston : Addison-Wesley, 2003. – 240 p.
6. .NET documentation [Electronic resource] // Microsoft Learn. – Access mode : URL: https://learn.microsoft.com/en-us/dotnet/. – Title from screen. – Date of access: 02.01.2025.
7. xUnit documentation [Electronic resource]. – Access mode : URL: https://xunit.net/?tabs=cs. – Title from screen. – Date of access: 02.01.2025.
8. GitHub Copilot : Your AI pair programmer [Electronic resource] // GitHub. – Access mode : URL: https://github.com/features/copilot. – Title from screen. – Date of access: 02.01.2025.
9. Graham, D. Foundations of Software Testing : ISTQB Certification [Text] / D. Graham, E. van Veenendaal, I. Evans, R. Black. – 1st ed. – London : Cengage Learning, 2008. – 304 p.
10. Jorgensen, P. C. Software Testing : A Craftsman’s Approach [Text] / Paul C. Jorgensen. – 4th ed. – Boca Raton : CRC Press, 2013. – 634 p.

11. Myers, G. J. The Art of Software Testing [Text] / G. J. Myers, C. Sandler, T. Badgett. – 3rd ed. – Hoboken : John Wiley & Sons, 2011. – 256 p.
12. Collin, M. Mastering Selenium WebDriver 3.0 [Text] / Mark Collin. – 1st ed. – Birmingham : Packt Publishing, 2018. – 320 p.
13. Ammann, P. Introduction to Software Testing [Text] / Paul Ammann, Jeff Offutt. – 2nd ed. – Cambridge : Cambridge University Press, 2016. – 350 p.
14. OWASP Testing Guide [Text]. – 4th ed. – OWASP Foundation, 2014. – 396 p.
15. ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. – Geneva : ISO, 2011. – 34 p.
16. IEEE Std 1012-2016 IEEE Standard for System, Software, and Hardware Verification and Validation. – New York : IEEE, 2016. – 152 p.
17. ISO/IEC/IEEE 29119 Software Testing. Parts 1–4. – Geneva : ISO, 2015.
18. GitHub – xunit/xunit : xUnit.net is a free, open source unit testing tool for .NET [Electronic resource]. – Access mode : URL: https://github.com/xunit/xunit. – Title from screen. – Date of access: 02.01.2025.
19. GitHub – danielpalme/ReportGenerator [Electronic resource]. – Access mode : URL: https://github.com/danielpalme/ReportGenerator. – Title from screen. – Date of access: 02.01.2025.
20. NuGet Gallery : dotnet-coverage 18.3.1 [Electronic resource]. – Access mode : URL: https://www.nuget.org/packages/dotnet-coverage. – Title from screen. – Date of access: 02.01.2025.
21. Using MSBuild [Electronic resource] // Microsoft Learn. – Access mode : URL:https://learn.microsoft.com/ru-ru/visualstudio/msbuild/walkthrough-using-msbuild. – Title from screen. – Date of access: 02.01.2025.
22. Code coverage in Visual Studio [Electronic resource] // Microsoft Learn. – Access mode : URL: https://learn.microsoft.com/ru-ru/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested. – Title from screen. – Date of access: 02.01.2025.

23. Creating an MSBuild project file from scratch [Electronic resource] // Microsoft Learn. – Access mode : URL: https://learn.microsoft.com/ru-ru/visualstudio/msbuild/walkthrough-creating-an-msbuild-project-file-from-scratch. – Title from screen. – Date of access: 02.01.2025.
24. What’s new in MSBuild 17.0 [Electronic resource] // Microsoft Learn. – Access mode : URL: https://learn.microsoft.com/ru-ru/visualstudio/msbuild/whats-new-msbuild-17-0. – Title from screen. – Date of access: 02.01.2025.
25. What’s new in MSBuild 17.0 [Electronic resource] // Microsoft Learn. – Access mode : URL: https://learn.microsoft.com/ru-ru/visualstudio/msbuild/whats-new-msbuild-17-0. – Title from screen. – Date of access: 02.01.2025.
26. MSBuild [Електронний ресурс] // Wikipedia. – Режим доступу : URL: https://ru.wikipedia.org/wiki/MSBuild. – Загл. з екрана. – Дата звернення: 02.01.2025.
27. xUnit [Електронний ресурс] // Wikipedia. – Режим доступу : URL: https://ru.wikipedia.org/wiki/XUnit. – Загл. з екрана. – Дата звернення: 02.01.2025.

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ
Перший проректор Українського
державного
університету науки і технологій
Анатолій РАДКЕВИЧ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01537–01–ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Віктор ШИНКАРЕНКО
Виконавець
_____Володимир МАКСИМЧУК
Нормоконтролер
_____Світлана ВОЛКОВА

2026

ЗАТВЕРДЖЕНО

44165850.01537–01–ЛЗ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Технічне завдання

Листів 13

2026

АНОТАЦІЯ

Документ 44165850.01395–01–ЛЗ «АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ» входить до складу програмної документації на програмний засіб, що реалізує експериментальну платформу для аналізу покриття коду модульними тестами згенерованими різними генеративними моделями.

У даному документі представлено технічне завдання на розробку програмного засобу. Програмний засіб реалізовано з використанням сучасних серверних веб-технологій. Орієнтовний обсяг оперативної пам’яті, що використовується програмою, залежить від обраної моделі доступу до даних та становить до 124 ГБ. Обсяг відеопам’яті залежить від моделі та становить до 84 ГБ. Конфігурація апаратного забезпечення залежить від обраної моделі та способу доступу до генеративної моделі та може потребувати мережеві адаптери в залежності від способа доступу до генеративних моделей. Програмний засіб призначений для роботи у системі Windows.

ЗМІСТ

ВСТУП

Програмний засіб для аналізу покриття коду модульними тестами згенерованими різними генеративними моделями має на меті експериментальне вивчення впливу різних моделей генерації модульних тестів методом білої скриньки на якість генерації а саме покриття коду модульними тестами та працездатністю тестів.

Використання різних моделей, зокрема локальних та хмарних з різною потужністю моделі дозволяє оцінити інваріанти для аналізу покриття коду модульними тестами. Автоматизація процесу вимірювання цих показників дозволяє об’єктивно оцінити ефективність згенерованих модульних тестів.

Розроблений програмний засіб може застосовуватися у навчальних і дослідницьких цілях для аналізу генерації.

1 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 27.12.2023 №1173ст ректора Українського державного університету науки і технологій “Про призначення наукових керівників та затвердження тем магістерських робіт” за спеціальністю 121 “Інженерія програмного забезпечення» факультету “Комп’ютерних технологій і систем” по кафедрі “Комп’ютерні інформаційні технології”.

Тема дипломної роботи – “АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ”. Керівник – Віктор ШИНКАРЕНКО.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення:

1. реалізація програмного засобу для аналізу покриття коду модульними тестами, згенерованими різними генеративними моделями;
2. підтримка генерації модульних тестів методом «білої скриньки» з використанням як локальних, так і хмарних моделей різної потужності;
3. автоматизований запуск, валідація працездатності та оцінювання якості згенерованих модульних тестів;
4. збір, фіксація та агрегація метрик покриття коду (line/branch/condition тощо) і показників коректності виконання тестів;
5. порівняльний аналіз ефективності різних генеративних моделей та конфігурацій генерації тестів.

Експлуатаційне призначення:

1. зменшення часу, необхідного для експериментального дослідження впливу різних генеративних моделей на покриття коду та працездатність модульних тестів;
2. отримання відтворюваних, об’єктивних і порівнюваних результатів завдяки автоматизації процесів генерації та вимірювання;
3. автоматизований збір телеметричних даних без ручного втручання, що мінімізує суб’єктивні похибки;
4. можливість масштабування експериментів шляхом зміни наборів моделей, параметрів генерації та цільових проектів;
5. застосування у навчальних і дослідницьких цілях для аналізу якості генерації модульних тестів і вивчення інваріантів покриття коду.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Програма повинна надавати можливість:

1. підключення та використання різних генеративних моделей для автоматизованої генерації модульних тестів методом «білої скриньки»;
2. вибору типу генеративної моделі:
    a. локальні моделі;
    b. хмарні моделі;
    c. моделі з різною обчислювальною потужністю та конфігураціями;
3. автоматизованого запуску згенерованих модульних тестів для цільового програмного коду;
4. вимірювання показників покриття коду (покриття рядків, гілок, умов тощо);
5. перевірки працездатності згенерованих тестів та фіксації помилок виконання;
6. збору та агрегації метрик якості генерації модульних тестів;
7. порівняльного аналізу результатів, отриманих від різних генеративних моделей;
8. збереження результатів експериментів для подальшого аналізу та відтворення.

3.2 Вимоги до надійності

Вимоги до надійності наступні:

1. відсутність критичних збоїв під час генерації, виконання та аналізу модульних тестів;
2. коректна обробка помилок генерації тестів, некоректного коду або некоректних результатів виконання;
3. інформування користувача про помилки, збої або нестандартні ситуації у процесі аналізу;

4. стабільність та відтворюваність результатів при повторному виконанні експериментів за однакових умов.

3.3 Вимоги експлуатації

Для стабільної та безперервної роботи програмного засобу необхідно забезпечити відповідні умови експлуатації обчислювального середовища. Обчислювальна система повинна функціонувати в умовах, що забезпечують стабільну роботу апаратного забезпечення та мінімізують ризик збоїв.

Експлуатація програмного засобу передбачає наявність користувача або адміністратора з базовими навичками роботи з інструментами тестування, системами контролю версій та засобами автоматизації тестування, який здатний здійснювати налаштування експериментів, запуск аналізу та інтерпретацію отриманих результатів.

3.4 Вимоги до складу та параметрів технічних засобів

Апаратне забезпечення, необхідне для роботи:

1. процесор з тактовою частотою не менше 3.2 ГГц;
2. не менше 124 ГБ оперативної пам’яті;
3. не менше 120 ГБ вільного дискового простору;
4. доступ до мережі Інтернет (для використання хмарних генеративних моделей).

3.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення, необхідне для роботи:

1. операційна система Windows 10/11;
2. середовище виконання для запуску та тестування програмного коду;
3. фреймворк для модульного тестування відповідної мови програмування;
4. інструменти вимірювання покриття коду;
5. засоби логування та збереження результатів експериментів.

3.6 Вимоги до маркування і упаковки

Програмний продукт та супровідна документація повинні бути захищені від механічних і кліматичних впливів.

Програма повинна мати етикетку, на якій зазначено:

1. назву програмного засобу;
2. версію програми;
3. ім’я розробника;
4. рік розробки.

Приклад маркування :

Module Testing Coverage Analyzer

Розробник: _____

УДУНТ, кафедра КІТ

2026

3.7 Вимоги до транспортування та зберігання

Транспортування повинно проводитись в упаковці та забезпечувати цілісність продукту.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу програмної документації повинні входити:

1. технічне завдання;
2. пояснювальна записка;
3. текст програми;
4. інструкція користувача.

Програмна документація повинна відповідати вимогам чинних стандартів ДСТУ.

5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Таблиця А.1 – Стадії та етапи розробки

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	09.10.2025 – 23.10.2025	
2	Аналіз сучасного стану вирішення обраної задачі та програмно-апаратного забезпечення	24.10.2025 – 08.11.2025	
3	Збір вимог до програми, вибір засобів штучного інтелекту для дослідження	10.11.2025 – 16.11.2025	30%
4	Зовнішнє та внутрішнє проектування	18.11.2025 – 09.12.2025	
5	Розробка програмного забезпечення	11.12.2025 – 17.12.2025	60%
6	Тестування та налагодження програмного забезпечення	20.12.2025 – 26.12.2025	
7	Проведення експериментального дослідження та аналіз результатів	26.12.2025 – 04.01.2026	
8	Оформлення пояснювальної записки	04.01.2025 – 06.01.2026	
9	Розробка демонстраційних матеріалів	06.01.2026 – 11.01.2026	100%
10	Подання кваліфікаційної роботи до кафедри	16.01.2026	
11	Подання кваліфікаційної роботи до кафедри	24.01.2026	

6 ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ

Контроль виконання здійснює керівник розробки Віктор Шинкаренко. Прийом здійснюється уповноваженою комісією.

7 БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем: методичні вказівки до дипломного проектування та лабораторних робіт/уклад.: Ю.М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38 с

ДОДАТОК Б

Інструкція користувача

ЗАТВЕРДЖУЮ
Перший проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Інструкція користувача

ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01537–01 І3 01–ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Віктор ШИНКАРЕНКО
Виконавець
_____Володимир МАКСИМЧУК
Нормоконтролер
_____Світлана ВОЛКОВА

2026

ЗАТВЕРДЖЕНО

44165850.01537–01 І3 01–ЛЗ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Інструкція користувача

Листів 15

2026

ВСТУП

1.1 Область застосування

Програмний засіб для аналізу покриття коду модульними тестами, згенерованими різними генеративними моделями, застосовується для експериментального дослідження якості автоматичної генерації модульних тестів методом «білої скриньки». Програмне забезпечення призначене для оцінювання впливу різних генеративних моделей, зокрема локальних і хмарних моделей різної потужності, на рівень покриття коду та працездатність згенерованих тестів.

Програмний засіб може використовуватися у навчальній та науково-дослідній діяльності для вивчення підходів до автоматизованого тестування програмного забезпечення, а також для порівняльного аналізу ефективності сучасних генеративних моделей у задачах генерації модульних тестів. Крім того, програмний засіб може застосовуватися під час дослідження та вдосконалення процесів забезпечення якості програмного коду в процесі розроблення програмних систем.

1.2 Опис можливостей

Програмний засіб надає користувачу можливість проводити експериментальне дослідження якості модульних тестів, згенерованих різними генеративними моделями, та отримувати кількісні показники покриття коду і коректності тестування. Основні можливості програмного засобу:

1. підключення та використання різних генеративних моделей для генерації модульних тестів методом «білої скриньки»;
2. вибір типу генеративної моделі:
    a. локальні моделі;
    b. хмарні моделі;
    c. моделі з різними параметрами та обчислювальною потужністю;
3. автоматизована генерація модульних тестів для заданого програмного коду;

4. запуск і перевірка працездатності згенерованих модульних тестів;
5. вимірювання показників покриття коду (покриття рядків, гілок, умов тощо);
6. збір та аналіз метрик якості згенерованих тестів;
7. порівняльний аналіз результатів генерації модульних тестів різними моделями;
8. збереження результатів експериментів у вигляді структурованих даних для подальшого аналізу та узагальнення.

2 ПРИЗНАЧЕННЯ І УМОВИ ЗАСТОСУВАННЯ

2.1 Функціональне і експлуатаційне призначення

Функціональне призначення розробки:

1. реалізація програмного засобу для автоматизованої генерації модульних тестів методом «білої скриньки» з використанням різних генеративних моделей;
2. запуск і виконання згенерованих модульних тестів у середовищі .NET із використанням фреймворку xUnit;
3. вимірювання та фіксація показників покриття коду модульними тестами;
4. перевірка працездатності згенерованих тестів і виявлення помилок виконання;
5. формування структурованих звітів про результати тестування та рівень покриття коду;
6. порівняльний аналіз ефективності генерації модульних тестів різними генеративними моделями.

Експлуатаційне призначення розробки:

1. отримання відтворюваних і об’єктивних результатів оцінювання якості згенерованих модульних тестів;
2. автоматизований збір і формування звітних даних без ручного втручання користувача;
3. можливість масштабування експериментів за рахунок зміни параметрів генерації тестів і вибору генеративної моделі;
4. застосування у навчальній та дослідницькій діяльності для аналізу якості автоматизованого тестування програмного коду.

2.2 Умови застосування

Для успішного застосування програми необхідно дотримання таких умов:

1. процесор з тактовою частотою не менше 3.2 ГГц;
2. не менше 124 ГБ оперативної пам’яті;
3. не менше 1 ТБ дискового простору для збереження кешу та звітів;
4. доступ до мережі Інтернет (для роботи з генеративними моделями); операційна система Windows 10/11;
5. платформа .NET для виконання програмного коду мовою C#;
6. фреймворк модульного тестування xUnit;
7. генератор звітів для формування результатів аналізу покриття коду; генеративна модель LLaMA;
8. контейнеризація середовища виконання за допомогою Docker.

3 ПІДГОТОВКА ДО РОБОТИ

3.1 Склад і зміст дистрибутивного носія даних

Для роботи з програмним засобом аналізу покриття коду модульними тестами, згенерованими генеративними моделями, необхідне таке програмне забезпечення та компоненти:

1. Docker — для контейнеризованого запуску генеративної моделі та допоміжних сервісів;
2. середовище виконання .NET — для запуску аналізованого проекту та модульних тестів мовою C#;
3. фреймворк модульного тестування xUnit — для виконання згенерованих тестів;
4. генератор звітів — для формування звітів про покриття коду та результати тестування;
5. генеративна модель LLaMA — для автоматизованої генерації модульних тестів;
6. файл конфігурації Docker (Dockerfile або docker-compose.yml) — для опису параметрів запуску контейнерів.

Дистрибутивний комплект містить

1. виконуваний файл програмного засобу;
2. конфігураційні файли Docker;
3. модуль генерації модульних тестів;
4. модуль збору метрик покриття коду;
5. модуль генерації звітів;
6. супровідну документацію користувача.

3.2 Порядок завантаження даних і програми

Перед початком роботи необхідно виконати такі дії:

1. Встановити Docker відповідно до офіційної документації.
2. Переконатися у доступності середовища .NET та фреймворку xUnit.

3. Отримати дистрибутивний комплект програмного засобу.
4. Запустити програмний засіб або відповідний Docker-контейнер.

3.3 Порядок запуску програмного засобу

Для запуску експериментального середовища необхідно:

1. Запустити програмний засіб або контейнер із генеративною моделлю LLaMA.
2. У користувацькому інтерфейсі вибрати файл або папку з вихідним програмним кодом, для якого необхідно згенерувати модульні тести.
3. Вибрати папку, у якій будуть збережені згенеровані модульні тести.
4. Задати назву генерації або файлу модульних тестів.
5. Обрати генеративну модель та параметри генерації тестів.
6. Запустити процес генерації модульних тестів.
7. Після завершення генерації автоматично виконуються модульні тести з використанням xUnit.
8. Виконується збір метрик покриття коду та працездатності тестів.
9. Генерується звіт з результатами аналізу.

3.4 Порядок перевірки працездатності

Для перевірки працездатності програмного засобу необхідно виконати такі дії:

1. Переконатися, що генеративна модель та програмний засіб успішно запущені.
2. Виконати генерацію модульних тестів для невеликого тестового проекту.
3. Переконатися у створенні файлів модульних тестів у вибраній папці.
4. Запустити автоматичне виконання тестів і перевірити відсутність критичних помилок.
5. Переконатися у коректному зборі метрик покриття коду.
6. Перевірити формування звіту з результатами генерації та тестування.

У разі успішного виконання зазначених дій програмний засіб вважається готовим до повноцінного використання для аналізу покриття коду модульними тестами, згенерованими різними генеративними моделями.

4 ВИКОНАННЯ ПРОГРАМИ

Виконайте запуск програми вказавши шлях до файлу програмного коду, обрати генеративну модель обрати шлях куди буде згенеровано тестовий файл натиснути прапор Generate test та натиснути кнопку старт цу ініціює процес створення тестів.

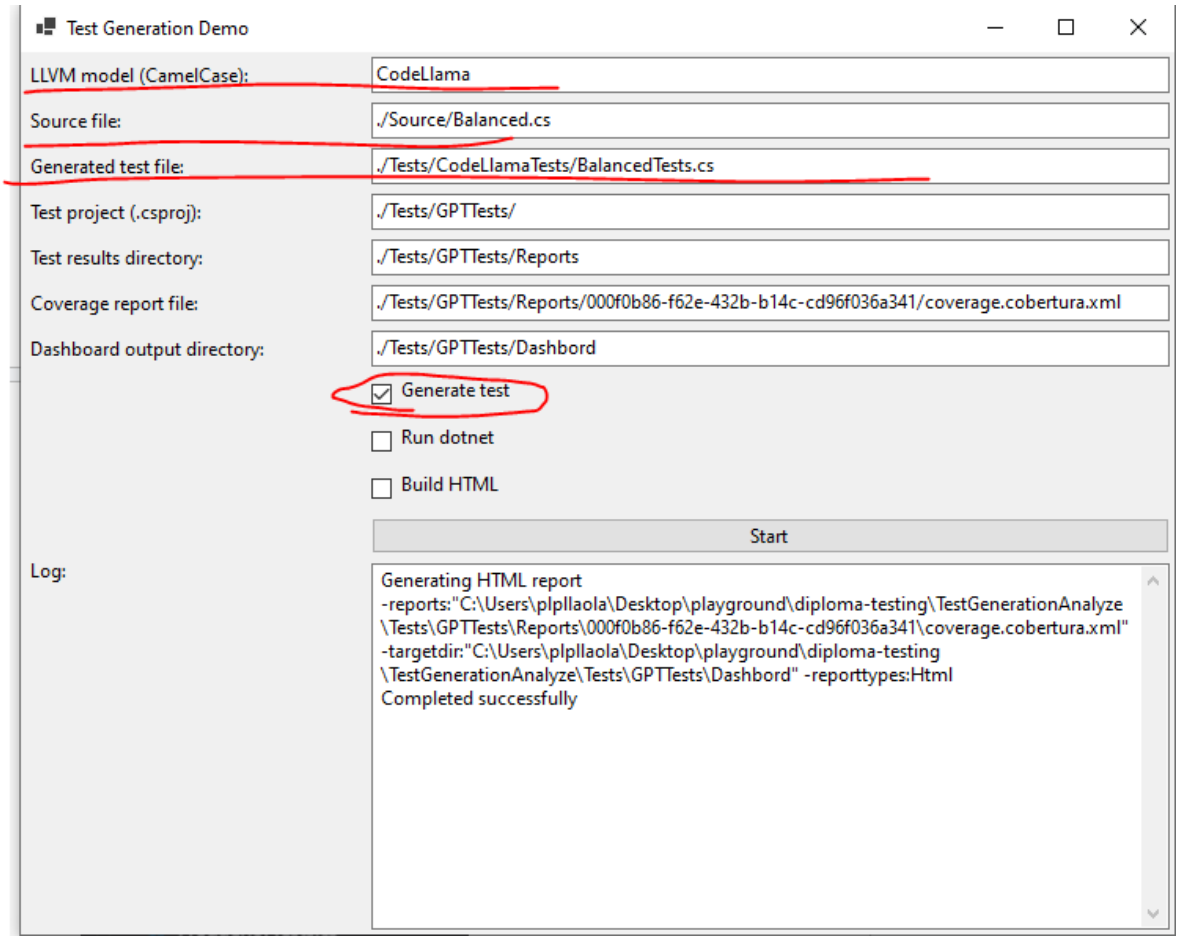

Рисунок 4.1 – Інтерфейс програми генерації для визначення файла з кодом

Генерація тестів виконується через клієнтські виклики до відповідних генеративних моделей. За генерацію тестів відповідає рядок інтерфейсу Generated Test File з додатково заповненим рядком шляху до файла программи на основі якого будується тест також потрібно ввести назву генеративної моделі а також прапор генерування тестів.

У разі, якщо згенерований код містить синтаксичні або логічні помилки, оператор може змінити вхідні інструкції (prompts), обрати іншу модель або, за необхідності, виконати мінімальні правки для забезпечення коректного запуску тестів. Такий підхід обумовлений тим, що не всі моделі гарантують генерацію повністю працездатного тестового коду.

Для отримання метрик покриття оператор задає шлях до тестового проекту відповідної моделі та ініціює запуск тестів із відповідним прапором. У результаті формується файл покриття у форматі Cobertura XML, який зберігається у каталозі результатів тестування. Для тестів, згенерованих моделлю ChatGPT-5.1, такі файли містять детальну інформацію про покриття, включно з великою кількістю проаналізованих рядків коду.

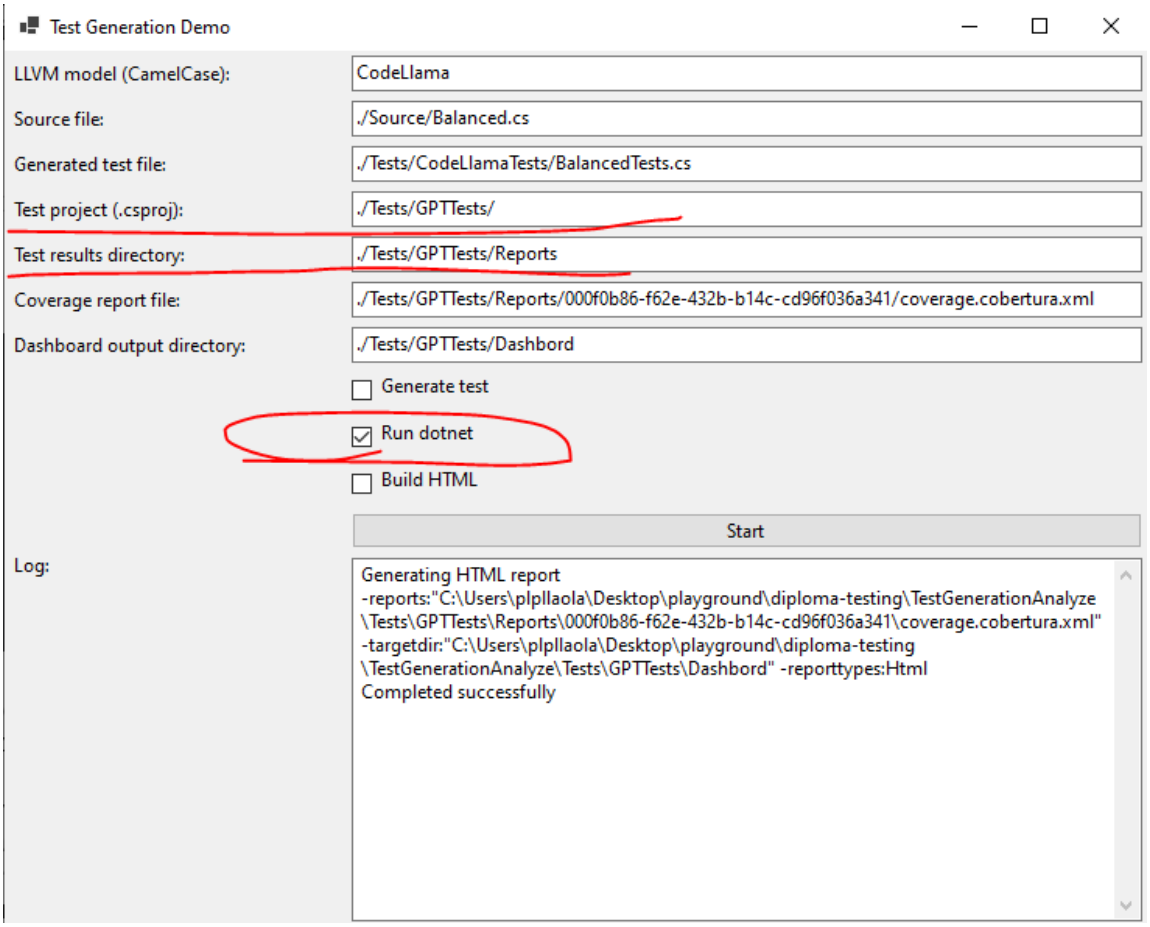

Рисунок 4.2 – Інтерфейс програми для генерації файла з метриками покриття

Отримані файли метрик передавалися інструментам аналізу покриття для подальшої обробки. На їх основі формувалися узагальнені показники якості тестування, що дозволили оцінити повноту охоплення коду, особливості поведінки алгоритмів під час тестування та виявити потенційні зони недостатнього покриття.

Для генерації звітів зазначте шлях до файлу покриття та каталог, у який має бути згенерований HTML-звіт. У результаті формується набір артефактів у вигляді веб-дашборда, що наочно відображає результати тестування. Дашборд дозволяє переглядати як загальні метрики покриття, так і детальне теплове представлення окремих файлів програмного коду.

Для генерації результатів вкажіть шлях до файлу метрик покриття та шлях куди буде генеруватись штмл дашборд а також встановити флаг генерації репорту

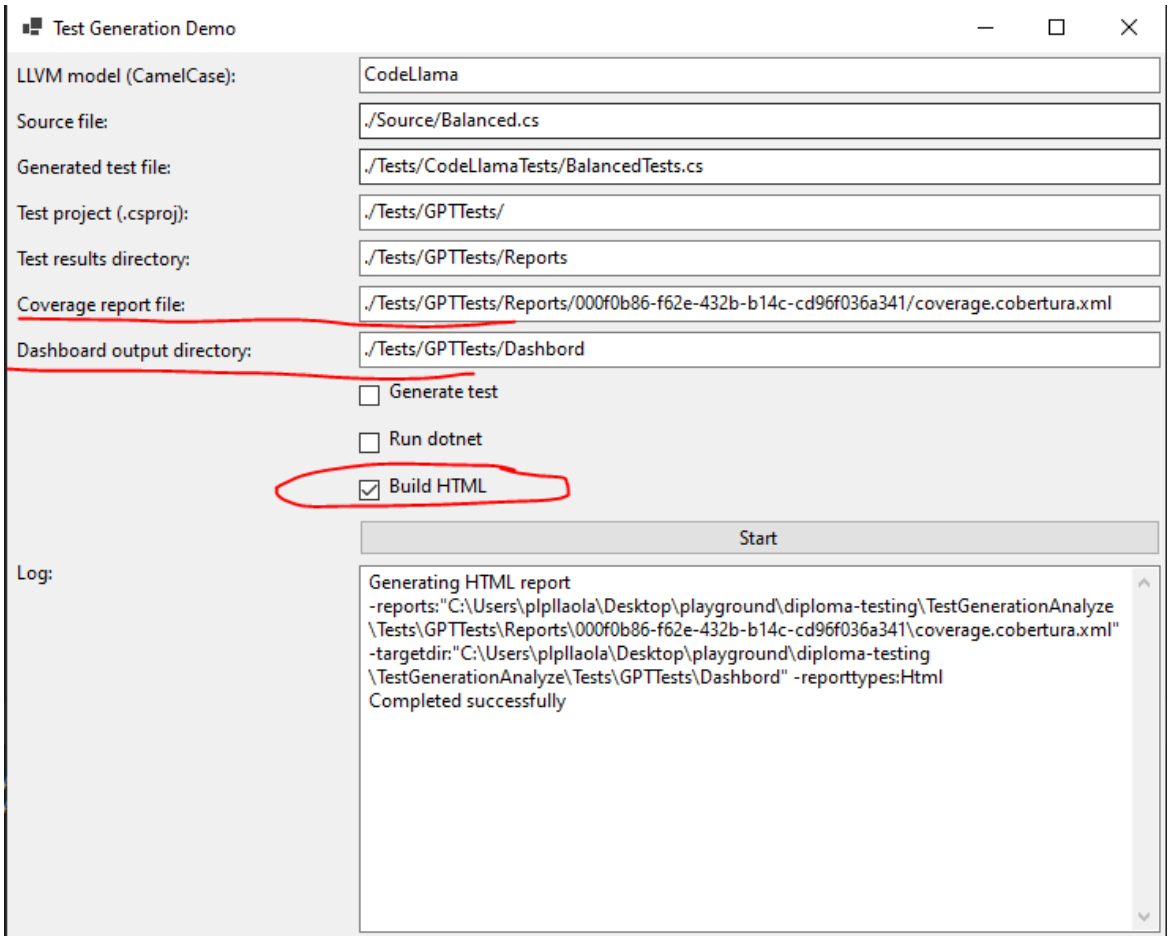

Рисунок 4.3 – Інтерфейс програми для генерації дашборда покриття

Отримайте артефакти у вигляді файлів веб сайту дашборда по вказаному у програмі шляху.

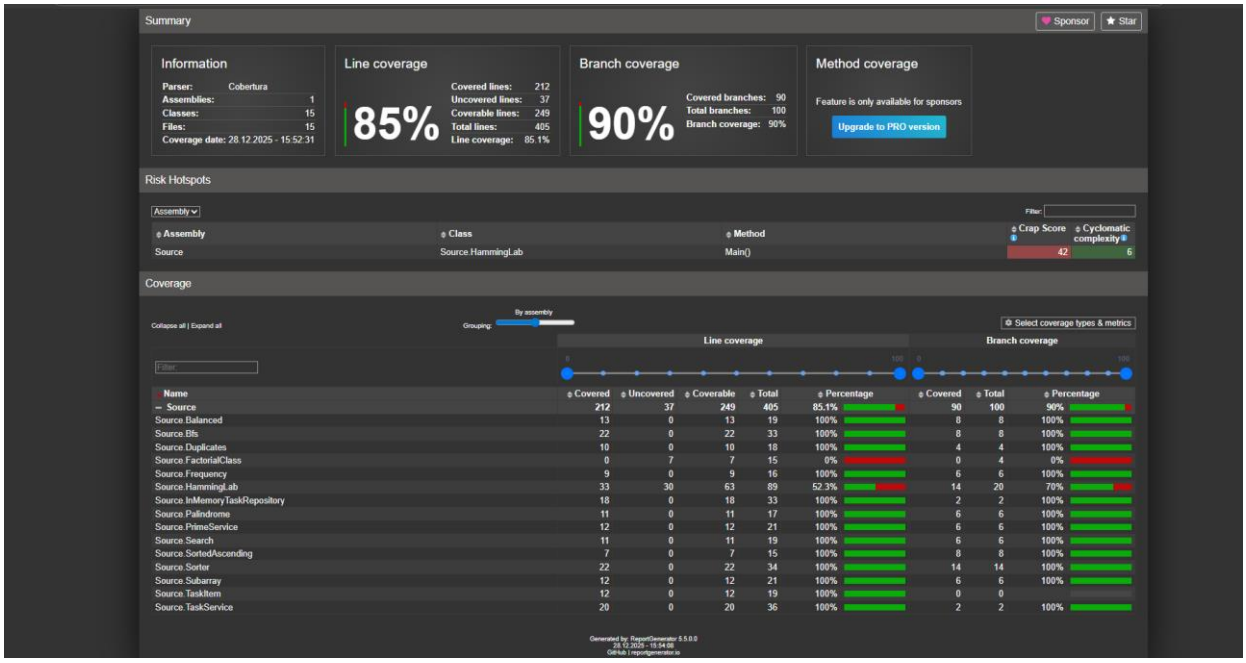

Рисунок 4.4 − Головна сторінка дашборда покриття усіх файлів

Дашборд теплового покриття окремого файла відкривається при натисканні на рядок у таблиці якості покриття по файлам

5 РЕКОМЕНДАЦІЇ ПО ЗАСТОСУВАННЮ

Для успішного досвіду роботи з програмою слід ознайомитись з керівництвом користувача та мати мінімальні навички роботи з комп’ютером.

6 ПОВІДОМЛЕННЯ

У таблиці Б.1 наведено повідомлення користувачу, які можуть з’явитись у процесі роботи з програмою.

Таблиця Б.1 – Повідомлення користувачу

Повідомлення	Опис	Рекомендації
Помилка при генеруються файли метрик	Не виконується запуск тестів	Перевірити тестовий проект на наявність дефектів у згенерованому коді
Виліт програми	Проблеми навантаження	Збільшити кількість апаратної потужності пристрою або вибрати меншу за обсягами модель

8. ПІДСТАВА ДЛЯ РОЗРОБКИ

Основою для розробки є наказ проректора Українського державного університету науки і технології Радкевич А.В. «Про затвердження тем та призначення керівників дипломних проектів» №____ ст від _____ року.

Тема проекту: “АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ».

Керівник дипломного проекту: Горячкін В. М.

ДОДАТОК В

Текст програми

ЗАТВЕРДЖУЮ
Перший проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Текст програми

ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01537–01 12 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Віктор ШИНКАРЕНКО
Виконавець
_____Володимир МАКСИМЧУК
Нормоконтролер
_____Світлана ВОЛКОВА

2026

ЗАТВЕРДЖЕНО

44165850.01537–01 12 01–ЛЗ

АНАЛІЗАТОР ГЕНЕРАЦІЇ ТЕСТІВ МЕТОДАМИ БІЛОЇ СКРИНЬКИ

Текст програми

Листів 4

2026

ЗМІСТ

Program.cs

Цей код являє собою настільний додаток на платформі .NET, реалізований із використанням Windows Forms та архітектурного підходу, близького до MVC.

Model відповідає за збереження та обробку даних, а також за бізнес-логіку додатку. У моделі інкапсульовані структури даних, валідація та операції, які не залежать від користувацького інтерфейсу.

View реалізована у вигляді WinForms-форм та контролів. Вона відповідає виключно за відображення даних та взаємодію з користувачем (кнопки, поля вводу, таблиці, події інтерфейсу), не містить бізнес-логіки.

namespace client;

Контролер виступає посередником між View і Model: обробляє дії користувача (події форм), ініціює виклики методів моделі та оновлює подання відповідно до отриманих результатів.

Комунікація між шарами побудована таким чином, щоб мінімізувати зв’язаність: View не працює безпосередньо з логікою даних, а Model не залежить від UI. Код орієнтований на розширюваність, тестованість та розподіл відповідальностей між компонентами додатку.

```
using System;
using System.Diagnostics;
using System.IO;
//./Tests/GPTTests/BfsTest3.cs ./Source/Bfs.cs TinyLlama
static class Program
{
   //[STAThread]
   static void Main()
   {
      ApplicationConfiguration.Initialize();
      Application.Run(new MainView());
```

```
    }
}

// ====================== MODEL ====================
class AppModel
{
    public string LLVMModel { get; set; }
    public string SourceFilePath { get; set; }             // ./Source/Bfs.cs
    public string GeneratedTestFilePath { get; set; }      //
./Tests/GPTTests/BfsTests.cs
    public string TestProjectPath { get; set; }            //
./Tests/GPTTests/GPTTests.csproj
    public string TestResultsDir { get; set; }             // ./Tests/GPTTests/Reports
    public string CoverageReportPath { get; set; }         //
./Tests/GPTTests/Reports/**/coverage.cobertura.xml
    public string DashboardOutputDir { get; set; }         // ./Dashboard

    public bool CreateNewTest { get; set; }
    public bool RunTests { get; set; }
    public bool BuildHtmlReport { get; set; }

    public bool Success { get; set; }
    public string Error { get; set; }

    public string RootDir
    {
        get
        {
            var exeDir = AppContext.BaseDirectory;
            var clientDir = Path.GetFullPath(Path.Combine(exeDir, "..", "..", ".."));
```

```
            return Path.GetFullPath(Path.Combine(clientDir, ".."));
        }
    }

    public string ResolvedSourceFilePath => Resolve(SourceFilePath);
    public string ResolvedGeneratedTestFilePath => Resolve(GeneratedTestFilePath);
    public string ResolvedTestProjectPath => Resolve(TestProjectPath);
    public string ResolvedTestResultsDir => Resolve(TestResultsDir);
    public string ResolvedCoverageReportPath => Resolve(CoverageReportPath);
    public string ResolvedDashboardOutputDir => Resolve(DashboardOutputDir);

    private string Resolve(string path)
    {
        if (string.IsNullOrWhiteSpace(path))
            throw new InvalidOperationException("Path is empty");

        path = path.Trim().Trim("");

        var combined = Path.IsPathRooted(path)
            ? path
            : Path.Combine(RootDir, path);

        return Path.GetFullPath(combined);
    }
}


// ==================== CONTROLLER ====================
class AppController
{
```

```
private readonly AppModel _model;
private readonly Action<string> _log;

public AppController(AppModel model, Action<string> log)
{
    _model = model;
    _log = log;
    LlmClient.Initialize(_log);
}

public async Task Start()
{
    try
    {
        if (_model.CreateNewTest)
        {
            _log("Reading input file");
            var source = File.ReadAllText(_model.ResolvedSourceFilePath);

            _log("Processing source (integration point)");
            var tests = await GenerateTests(source, _model.LLVMModel);
            var testFile = _model.ResolvedGeneratedTestFilePath;

            _log("Getting generated tests");
            File.WriteAllText(testFile, tests);
        }

        if (_model.RunTests)
        {
            _log("Running dotnet test");
```

```
            _log($"test {_model.ResolvedTestProjectPath} --collect:\"XPlat Code
Coverage\" --results-directory:\"{_model.ResolvedTestResultsDir}\"");
            RunProcess("dotnet", $"test {_model.ResolvedTestProjectPath} --
collect:\"XPlat Code Coverage\" --results-
directory:\"{_model.ResolvedTestResultsDir}\"");
        }
        //./Tests/GPTTests/Bsftst2.cs ./Source/Bfs.cs
./Tests/GPTTests/Reports/7c73d34c-15ce-449a-b983-
450b01941e49/coverage.cobertura.xml
        if (_model.BuildHtmlReport)
        {
            _log("Generating HTML report");
            _log($"-reports:\"{_model.ResolvedCoverageReportPath}\" -
targetdir:\"{_model.ResolvedDashboardOutputDir}\" -reporttypes:Html");
            RunProcess("reportgenerator", $"-
reports:\"{_model.ResolvedCoverageReportPath}\" -
targetdir:\"{_model.ResolvedDashboardOutputDir}\" -reporttypes:Html");
        }

        _model.Success = true;
        _log("Completed successfully");
    }
    catch (Exception ex)
    {
        _model.Success = false;
        _model.Error = ex.Message;
        _log("Error: " + ex.Message);
    }
}
```

```
private async Task<string> GenerateTests(string code, string model)
{
    var generatedTest = "";
    var prompt = BuildTestPrompt(code, model);

    switch(model)
    {
        case "TinyLlama":
            generatedTest = await LlmClient.TinyLlama.GenerateAsync(prompt);
            break;

        case "CodeLlama":
             generatedTest = await LlmClient.CodeLlama.GenerateAsync(prompt);
             break;

        case "GptOss120b":
            generatedTest = await LlmClient.GptOss120b.GenerateAsync(prompt);
            break;

        default:
            throw new Exception("No model found");
    }

    _log(generatedTest);

    return generatedTest;
}

private void RunProcess(string file, string args)
{
```

```
    var p = new Process();
    p.StartInfo.FileName = file;
    p.StartInfo.Arguments = args;
    p.StartInfo.RedirectStandardOutput = true;
    p.StartInfo.RedirectStandardError = true;
    p.StartInfo.UseShellExecute = false;
    p.Start();
    p.WaitForExit();
}

private string BuildTestPrompt(string sourceCode, string modelName)
{
    // Общие правила для всех моделей
    string rules = """
        Rules:
        - Use xUnit for testing
        - Tests must compile
        - Tests may be trivial (pass/fail checks)
        - Do not include comments or explanations
        - Use namespace: GeneratedTests
        - Each test class should be named AutoGeneratedTests
        - Include at least one [Fact] per logical branch
        """;

    // Генерация конкретного промпта
    string prompt = modelName switch
    {
        "TinyLlama" => $"""
        Generate a SIMPLE xUnit test file in C# based on the following source code.
Return ONLY valid C# code.
```

```
        {rules}
        Source code:
        {sourceCode}
        """,

            "CodeLlama" => $"""
        Generate xUnit tests for the following C# code. Only valid C# code. No explanations.
        {rules}
        Source code:
        {sourceCode}
        """,

            "GptOss120b" => $"""
        You are an AI code generator. Generate fully functional xUnit test classes in C# for the following code. Only output C# code, no text explanations.
        {rules}
        Source code:
        {sourceCode}
        """,

            _ => $"""
        Generate xUnit tests for the following C# code. Only output valid C# code, no comments.
        Source code:
        {sourceCode}
        """
    };

    return prompt;
```

```
    }

}

// ==================== VIEW ====================
class MainView : Form
{
    TextBox txtLlvmModel = new() { Dock = DockStyle.Fill };
    TextBox txtSourceFile = new() { Dock = DockStyle.Fill };
    TextBox txtGeneratedTest = new() { Dock = DockStyle.Fill };
    TextBox txtTestProject = new() { Dock = DockStyle.Fill };
    TextBox txtTestResults = new() { Dock = DockStyle.Fill };
    TextBox txtCoverage = new() { Dock = DockStyle.Fill };
    TextBox txtDashboard = new() { Dock = DockStyle.Fill };

    CheckBox chkCreateTest = new() { Text = "Generate test for source file" };
    CheckBox chkRunTests = new() { Text = "Run dotnet test" };
    CheckBox chkBuildReport = new() { Text = "Build HTML coverage report" };

    Button btnStart = new() { Text = "Start", Dock = DockStyle.Fill };
    TextBox logBox = new() { Multiline = true, Dock = DockStyle.Fill, ScrollBars =
ScrollBars.Vertical };

    public MainView()
    {
        Text = "Test Generation Demo";
        Width = 750;
        Height = 600;

        var layout = new TableLayoutPanel
```

```
        {
            Dock = DockStyle.Fill,
            ColumnCount = 2,
            RowCount = 12,
            AutoSize = true
        };

        layout.ColumnStyles.Add(new ColumnStyle(SizeType.Absolute, 220));
        layout.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 100));

        AddRow(layout, "LLVM model (CamelCase):", txtLlvmModel);
        AddRow(layout, "Source file:", txtSourceFile);
        AddRow(layout, "Generated test file:", txtGeneratedTest);
        AddRow(layout, "Test project (.csproj):", txtTestProject);
        AddRow(layout, "Test results directory:", txtTestResults);
        AddRow(layout, "Coverage report file:", txtCoverage);
        AddRow(layout, "Dashboard output directory:", txtDashboard);

        layout.Controls.Add(chkCreateTest, 1, layout.RowCount++);
        layout.Controls.Add(chkRunTests, 1, layout.RowCount++);
        layout.Controls.Add(chkBuildReport, 1, layout.RowCount++);

        layout.Controls.Add(btnStart, 1, layout.RowCount++);

        layout.Controls.Add(new Label { Text = "Log:", Dock = DockStyle.Fill }, 0,
layout.RowCount);
        layout.Controls.Add(logBox, 1, layout.RowCount++);

        Controls.Add(layout);
```

```
        btnStart.Click += StartBtn_Click;
    }

    private void AddRow(TableLayoutPanel panel, string label, Control control)
    {
        int row = panel.RowCount++;
        panel.RowStyles.Add(new RowStyle(SizeType.AutoSize));
        panel.Controls.Add(new Label { Text = label, Dock = DockStyle.Fill, TextAlign
= ContentAlignment.MiddleLeft }, 0, row);
        panel.Controls.Add(control, 1, row);
    }
    //./Tests/GPTTests/Reports/411feaf3-3e47-4750-9036-
ebf4d32abe0f/coverage.cobertura.xml
    //./Tests/GPTTests/Dashbord
    //./Tests/GPTTests/
    //./Tests/GPTTests/Reports
    private async void StartBtn_Click(object sender, EventArgs e)
    {
        logBox.Clear();

        var model = new AppModel
        {
            LLVMModel = txtLlvmModel.Text,
            SourceFilePath = txtSourceFile.Text,
            GeneratedTestFilePath = txtGeneratedTest.Text,
            TestProjectPath = txtTestProject.Text,
            TestResultsDir = txtTestResults.Text,
            CoverageReportPath = txtCoverage.Text,
            DashboardOutputDir = txtDashboard.Text,
```

```
            CreateNewTest = chkCreateTest.Checked,
            RunTests = chkRunTests.Checked,
            BuildHtmlReport = chkBuildReport.Checked
        };

        var controller = new AppController(model, Log);
        await controller.Start();

        if (!model.Success)
            MessageBox.Show(model.Error ?? "Unknown error");
    }

    private void Log(string msg)
    {
        logBox.AppendText(msg + Environment.NewLine);
    }
}
```

HttpClient.cs

Цей код реалізує патерн HTTP-клієнта з базовим абстрактним рівнем та спеціалізованими реалізаціями для роботи з зовнішніми API генеративних моделей.

Базовий HTTP-клієнт інкапсулює загальну інфраструктурну логіку взаємодії через HTTP: ініціалізацію та конфігурацію з'єднання, управління базовим URL, заголовками, авторизацією, серіалізацією та десеріалізацією запитів і відповідей, а також централізовану обробку помилок та мережевих винятків. Він надає уніфіковані методи для виконання HTTP-запитів (GET, POST тощо), не прив’язуючись до конкретного API чи формату корисного навантаження.

Похідні клієнти успадковуються від базового клієнта та реалізують конкретні API-контракти. У них визначаються специфічні для кожної генеративної моделі ендпоїнти, структури запитів і відповідей, параметри генерації та додаткові правила обробки результатів. Ці клієнти використовують функціональність базового класу, перевизначаючи або розширюючи її лише в тих місцях, де потрібна специфіка конкретного API.

Таке розділення дозволяє:

1. ізолювати загальну мережеву та транспортну логіку від доменної логіки API;
2. спростити додавання нових клієнтів для інших генеративних моделей без зміни існуючого коду;
3. підвищити тестованість за рахунок можливості мокування базового клієнта;
4. забезпечити єдиний контракт взаємодії з різними зовнішніми сервісами.

Архітектура орієнтована на розширюваність, повторне використання коду та зменшення зв'язаності між компонентами, що використовують API генеративних моделей, та конкретними реалізаціями HTTP-клієнтів.

namespace client;

```
using System.Net.Mime;
using System.Text.Json;
using System.Threading.Tasks;
using System.Text;
using System.Net.Http;

public class Request
{
   public string Url { get; set; } = "";
   public object? Body { get; set; } = null;
   public Dictionary<string, string>? Headers { get; set; } = null;
```

```
    public HttpMethod Method { get; set; } = null!;
}

public class HttpClientWrapper
{
    private readonly HttpClient _http;
    private readonly Action<string> _log;

    public HttpClientWrapper(Action<string> log)
    {
        _http = new HttpClient();
        _log = log ?? (_ => { });
    }

    public async Task<T> SendAsync<T>(Request req)
    {
        var requestMessage = new HttpRequestMessage
        {
            RequestUri = new Uri(req.Url),
            Method = req.Method ?? (req.Body != null ? HttpMethod.Post :
HttpMethod.Get)
        };

        if (req.Body != null)
        {
            string json = JsonSerializer.Serialize(req.Body);
            requestMessage.Content = new StringContent(json, Encoding.UTF8,
MediaTypeNames.Application.Json);
            _log("Request Body: " + json);
        }
```

```
        if (req.Headers != null)
        {
            foreach (var kv in req.Headers)
            {
                requestMessage.Headers.TryAddWithoutValidation(kv.Key, kv.Value);
              _log($"Header: {kv.Key} = {kv.Value}");
            }
        }

       _log("Sending request to: " + req.Url);

        var response = await _http.SendAsync(requestMessage);

        if (!response.IsSuccessStatusCode)
        {
            var errorBody = await response.Content.ReadAsStringAsync();
            throw new Exception(!string.IsNullOrWhiteSpace(errorBody) ? errorBody :
response.ReasonPhrase);
        }

        var respText = await response.Content.ReadAsStringAsync();
       _log("Response status: " + response.StatusCode);
       _log("Response body: " + respText);

        if (!response.IsSuccessStatusCode)
        {
            throw new Exception(!string.IsNullOrWhiteSpace(respText) ? respText :
response.ReasonPhrase);
        }
```

```
        return JsonSerializer.Deserialize<T>(respText)!;
    }
}

public static class LlmClient
{
    private static HttpClientWrapper _client;

    public static void Initialize(Action<string> log)
    {
        _client = new HttpClientWrapper(log);
    }

    // ================== TinyLlama ==================
    public static class TinyLlama
    {
        private static readonly string Url = "http://localhost:11434/v1/responses";

        public static async Task<string> GenerateAsync(string prompt)
        {
            var request = new Request
            {
                Url = Url,
                Body = new
                {
                    model = "tinyllama",
                    input = prompt,
                    stream = false,
                }
```

```
        };

        var resp = await _client.SendAsync<JsonElement>(request);

        // извлекаем только поле "response" как строку
        if (resp.TryGetProperty("output", out var outputProp) &&
            outputProp.ValueKind == JsonValueKind.Array &&
            outputProp.GetArrayLength() > 0)
        {
            var first = outputProp[0];
            if (first.TryGetProperty("content", out var contentProp) &&
                contentProp.ValueKind == JsonValueKind.Array &&
                contentProp.GetArrayLength() > 0)
            {
                var textProp = contentProp[0];
                if (textProp.TryGetProperty("text", out var textValue))
                {
                    return textValue.GetString() ?? "";
                }
            }
        }

        return "";
    }
}

// ================== CodeLlama ==================
public static class CodeLlama
{
    private static readonly string Url = "http://localhost:11435/v1/responses";
```

```
public static async Task<string> GenerateAsync(string prompt)
{
    var request = new Request
    {
        Url = Url,
        Body = new
        {
            model = "codellama",
            input = prompt,
            stream = false,
        }
    };

    var resp = await _client.SendAsync<JsonElement>(request);

    if (resp.TryGetProperty("output", out var outputProp) &&
        outputProp.ValueKind == JsonValueKind.Array &&
        outputProp.GetArrayLength() > 0)
    {
        var first = outputProp[0];
        if (first.TryGetProperty("content", out var contentProp) &&
            contentProp.ValueKind == JsonValueKind.Array &&
            contentProp.GetArrayLength() > 0)
        {
            var textProp = contentProp[0];
            if (textProp.TryGetProperty("text", out var textValue))
            {
                return textValue.GetString() ?? "";
            }
```

```
        }
    }

    return "";
  }
}

// ================== GPT OSS 120B ==================
public static class GptOss120b
{
  private static readonly string Url = "http://localhost:11436/v1/responses";

  public static async Task<string> GenerateAsync(string prompt)
  {
    var request = new Request
    {
      Url = Url,
      Body = new
      {
        model = "gpt-oss-120b",
        input = prompt,
          stream = false,
      }
    };

    var resp = await _client.SendAsync<JsonElement>(request);

    // извлекаем текст из "output[0].content[0].text"
    if (resp.TryGetProperty("output", out var outputProp) &&
      outputProp.ValueKind == JsonValueKind.Array &&
```

```
            outputProp.GetArrayLength() > 0)
        {
            var first = outputProp[0];
            if (first.TryGetProperty("content", out var contentProp) &&
                contentProp.ValueKind == JsonValueKind.Array &&
                contentProp.GetArrayLength() > 0)
            {
                var textProp = contentProp[0];
                if (textProp.TryGetProperty("text", out var textValue))
                {
                    return textValue.GetString() ?? "";
                }
            }
        }

        return "";
    }
  }
}
```

ДОДАТОК Г

Аналіз ефективності тестування програмного забезпечення методом “білої скриньки” в умовах зростаючої складності програмних систем

Максимчук В.С., Шинкаренко В.І, Український державний університет науки і технологій, Україна

Програмна інженерія пройшла тривалий період еволюції: від перших спроб формалізувати процес створення програмного забезпечення до появи комплексних практик, що визначають сучасний вигляд галузі. Зі збільшенням масштабів і складності програмних систем суттєво зросли вимоги до якості, надійності та передбачуваності програмного продукту. У цих умовах роль тестування на різних етапах життєвого циклу ПЗ стала ключовою складовою забезпечення стабільності та безпомилковості систем.

Однією з найбільш ефективних методик є тестування за принципом “білої скриньки”, яке базується на аналізі внутрішньої структури коду. На відміну від методів “чорної скриньки”, що досліджують зовнішню поведінку системи, підхід “білої скриньки” забезпечує можливість оцінити повноту покриття, перевірити коректність логічних розгалужень, виявити приховані дефекти та оптимізувати архітектурні рішення на ранніх етапах. Зростання кількості залежностей, сторонніх бібліотек та абстракцій у сучасній розробці ще більше підсилює потребу в таких інструментах.

Метою роботи є практико-аналітичне дослідження ефективності використання методів “білої скриньки” для автоматизованої генерації тестів, а також визначення меж застосування ручного та автоматизованого підходів у контексті різних типів кодових фрагментів.

Для досягнення цієї мети здійснено аналіз якості автоматично згенерованих тестів на основі різнорідних ділянок коду, відібраних з відкритих репозиторіїв. Вибірка охоплює алгоритмічні структури, компоненти з бізнес-логікою, а також елементи взаємодії між об’єктами. Оцінювання проводилося з позиції повноти покриття, здатності виявляти приховані дефекти та відповідності тестів реальним сценаріям виконання.

Дослідження показало, що методи “білої скриньки” демонструють високу ефективність у виявленні логічних помилок та оптимізації складних розгалужень, однак їх результативність значною мірою залежить від контексту та специфіки програмної логіки. Водночас сучасні інструменти автоматизованої генерації тестів, що базуються на аналізі внутрішньої структури коду, продемонстрували здатність з високою точністю покривати різнорідні ділянки програмного забезпечення. За результатами експерименту ці моделі забезпечили стабільні показники покриття та коректно ідентифікували ділянки коду, критичні для контролю логічної цілісності.

Отримані результати мають практичну цінність для інженерів з тестування та команд розробки, які прагнуть підвищити рівень автоматизації та покращити точність тестового покриття. Дослідження формує підґрунтя для подальшого вдосконалення підходів на основі статичного аналізу, машинного навчання та гібридних методик, здатних об’єднувати сильні сторони автоматизованої та ручної генерації тестів.

Таким чином, робота підкреслює важливість системного підходу до вибору інструментів тестування та демонструє, що сучасні моделі аналізу коду можуть забезпечувати високу точність покриття при мінімальних витратах на ручне втручання. Це сприяє формуванню зрілої культури тестування, підвищує надійність програмного забезпечення та розширює можливості автоматизації у сучасній розробці.