Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.240165.01.06.ПЗ

Виконав студент 2 курсу, група ІПЗм-24-1 _____ Дмитро МАЦЮК
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Оксана ЯШИНА
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. пед. наук, доцент _____ Наталія ПРАВОРСЬКА
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ініціали, прізвище

15 грудня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01 09 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Мацюку Дмитру Віталійовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки

Керівник проєкту (роботи) канд. техн. наук, доцент Яшина О.М.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2025 р. № 65
2. Строк подання студентом проекту (роботи) на кафедру 01.12.2025 р.
3. Вихідні дані до проекту (роботи) Матеріали науково-дослідної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз предметної області та рішень з програмного забезпечення.
2 Удосконалення методу підтримки якості програмного коду на основі автоматичного тестування
3 Архітектура програмної реалізації
4 Програмна реалізація
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Форкун Ю.В., доцент	14.12	14.12
Нормоконтроль	Праворська Н.І., доцент	11.12	13.12

7. Дата видачі завдання «01» вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1. Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	20.10 - 26.09.2025	виконано
2. Робота над розділом 1 кваліфікаційної роботи - аналіз предметної області та постановка завдання	20.10 - 26.09.2025	виконано
3 Робота над розділом 2 кваліфікаційної роботи - визначення теоретичних засад визначення оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки	27.10 - 02.11.2025	виконано
4. Робота над науковими публікаціями, статтями	03.11 - 09.11.2025	виконано
5. Робота над розділом 3. Проектування архітектури системи для вирішення задачі, розробка вимог.	10.11 - 16.11.2025	виконано
6 Робота над розділом 4 кваліфікаційної роботи - формалізація, оцінювання та порівняльний аналіз запропонованого підходу	17.11 - 23.11.2025	виконано
7 Попередній захист кваліфікаційної роботи	24.11 - 30.11.2025	виконано
8 Узгодження постановки задачі, отриманих результатів та висновків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	01.12-07.12.2025	виконано
9 Перевірка роботи на наявність плагіату: нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	08.12 - 14.12.2025	виконано
10 Підготовка до захисту кваліфікаційної роботи	З 15.12.2025	виконано

Студент _____ Дмитро МАЦЮК
Підпис Ініціали, прізвище

Керівник проєкту (роботи) _____ Оксана ЯШИНА
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки».

Автор роботи: Мацюк Дмитро Віталійович.

Керівник роботи: Яшина Оксана Миколаївна.

Пояснювальна записка: 98 с., 15 рис., 2 дод., 44 джерела.

АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, ДИФЕРЕНЦІАЛЬНЕ ТЕСТУВАННЯ, ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ІНДЕКС ФУНКЦІОНАЛЬНОЇ СТАБІЛЬНОСТІ, МЕТРИКИ ЯКОСТІ, ОЦІНЮВАННЯ ЯКОСТІ, ПОВЕДІНКОВА СТАБІЛЬНІСТЬ, РЕГРЕСІЙНЕ ТЕСТУВАННЯ, CI/CD, FSI, QPV, WDSI.

Мета роботи - розробка методу кількісного оцінювання якості програмного забезпечення, що базується на аналізі розбіжностей у функціональній поведінці між різними версіями системи, виявлених за допомогою диференціального тестування.

Об'єктом дослідження є процеси та інструменти забезпечення якості програмного забезпечення на етапах його розробки, супроводу та еволюції в умовах застосування гнучких методологій розробки та практик безперервної інтеграції і доставки.

Предметом дослідження є методи генерації, виконання й порівняння результатів тестування у диференціальному підході, а також засоби кількісного оцінювання якості програмного забезпечення на основі аналізу поведінкових розбіжностей між різними реалізаціями або версіями систем.

З урахуванням мети, предмета та об'єкта дослідження були сформульовані наступні завдання:

1. Аналіз предметної області досліджуваної проблеми.

2. Дослідження методів оцінки якості програмного забезпечення на основі диференціального тестування функціональної поведінки.

3. Розробка методу оцінки якості програмного забезпечення на основі диференціального тестування функціональної поведінки.

4. Удосконалення підходу до класифікації розбіжностей шляхом розробки чотирирівневої ієрархічної системи.

Наукова новизна:

1. Вперше запропоновано метод кількісної оцінки якості програмного забезпечення, який систематично використовує результати диференціального тестування для формування інтегрованої оцінки поведінкової стабільності за допомогою набору метрик (FSI, WDSI, DR, QPV), на відміну від існуючих підходів, які використовують диференціальне тестування виключно для бінарного виявлення дефектів.

2. Підхід до класифікації розбіжностей було вдосконалено шляхом розробки чотирирівневої ієрархічної системи (катастрофічна, семантична, поведінкова, презентаційна) з функцією вагового коефіцієнта для визначення ступеня серйозності. Застосування диференціального тестування в контексті CI/CD було вдосконалено шляхом розробки концептуальної моделі для інтеграції методу як автоматичного «контролю якості» з пороговими значеннями метрик для прийняття рішень про готовність збірки.

Практичне значення отриманих результатів. вирішує актуальну проблему відсутності автоматизованих методів кількісного оцінювання якості ПЗ в умовах безперервної розробки. Розроблений метод заповнює дослідницький розрив, надаючи індустрії практичний інструмент для автоматизації воріт якості з потенціалом економії.

В ході проведення даного дослідження використано методи системного аналізу для дослідження сучасного стану проблеми, методи теорії множин та математичної логіки для формалізації понять розбіжності та побудови математичного апарату, методи теорії алгоритмів.

05.12.25

ABSTRACT

Master's thesis: «Method for evaluating software quality based on differential testing of functional behaviour».

Author: Matsyuk Dmytro.

Head of work: Oksana Yashyna.

Master's thesis consists of: 98 pages of the general text, 15 graphics, 2 supplements, 44 literature sources.

AUTOMATED TESTING, CONTINUOUS INTEGRATION, DIFFERENTIAL TESTING, SOFTWARE QUALITY ASSURANCE, FUNCTIONAL STABILITY INDEX, QUALITY METRICS, QUALITY ASSESSMENT, BEHAVIOURAL STABILITY, REGRESSION TESTING, CI/CD, FSI, QPV, WDSI.

The aim of the study is to develop a method for quantitative assessment of software quality based on the analysis of discrepancies in functional behaviour between different versions of the system, identified by differential testing.

The object of the study is the processes and tools for ensuring software quality at the stages of its development, maintenance and evolution in the context of applying flexible development methodologies and continuous integration and delivery practices.

The subject of the study is methods for generating, executing, and comparing test results in a differential approach, as well as means for quantitative assessment of software quality based on analysis of behavioural differences between different implementations or versions of systems.

Taking into account the purpose, subject, and object of the study, the following tasks were formulated:

1. Analysis of the subject area of the problem under study.
2. Research into methods of assessing software quality based on differential testing of functional behaviour.
3. Development of a method for evaluating software quality based on differential testing of functional behaviour.

4. Improving the approach to classifying discrepancies by developing a four-level hierarchical system.

Scientific novelty:

1. For the first time, a method for quantitative assessment of software quality has been proposed that systematically uses the results of differential testing to form an integrated assessment of behavioural stability through a set of metrics (FSI, WDSI, DR, QPV), unlike existing approaches that use differential testing exclusively for binary defect detection.

2. The approach to classifying discrepancies has been improved by developing a four-level hierarchical system (catastrophic, semantic, behavioural, presentation) with a weighting coefficient function for severity. The application of differential testing in the context of CI/CD has been further developed through the development of a conceptual model for integrating the method as an automatic «quality gate» with threshold metric values for making decisions about build readiness.

Practical significance of the results obtained. Solves the pressing problem of the lack of automated methods for quantitative assessment of software quality in conditions of continuous development. The developed method fills the research gap, providing the industry with a practical tool for automating quality gates with potential savings.

In the course of this study, methods of system analysis were used to investigate the current state of the problem, methods of set theory and mathematical logic to formalise the concepts of discrepancy and construct a mathematical apparatus, and methods of algorithm theory.

05.12.25

ЗМІСТ

ВСТУП

Актуальність дослідження. Сучасна цифрова економіка стрімко розвивається, і в цих умовах забезпечення високої якості програмного забезпечення перетворилося на критично важливу задачу, яка безпосередньо впливає на конкурентоспроможність продуктів, довіру користувачів та економічну ефективність компаній.

Еволюція методологій розробки від традиційних каскадних моделей до гнучких підходів (Agile) та практик безперервної інтеграції та доставки (DevOps/CI/CD) кардинально змінила вимоги до процесів забезпечення якості. За даними звіту State of DevOps 2024, 76% організацій використовують практики CI/CD, що передбачає миттєвий (до 15 хвилин) зворотний зв'язок про якість кожної зміни коду. У таких умовах традиційні підходи до тестування, що базуються на ручній розробці тест-кейсів та визначенні очікуваних результатів, стають вузьким місцем через високу трудомісткість та неможливість швидко адаптуватися до постійних змін.

Одним з перспективних підходів до вирішення цієї проблеми є диференціальне тестування — техніка, що використовує порівняння поведінки кількох реалізацій однієї специфікації для виявлення розбіжностей без необхідності заздалегідь визначати еталонні результати. Аналіз публікацій у базах даних IEEE Xplore та ACM Digital Library за період 2020–2024 років показує зростання кількості робіт з диференціального тестування на 127% (з 234 до 531 публікації), що свідчить про активний інтерес наукової спільноти до цього напрямку.

Таким чином, існує протиріччя між потребою індустрії в швидких, автоматизованих та об'єктивних методах оцінювання якості програмного забезпечення з одного боку, та відсутністю формалізованих підходів до кількісного аналізу результатів диференціального тестування - з іншого. Актуальність даного дослідження обґрунтовується наступними факторами:

Незважаючи на значну кількість досліджень у галузі диференціального тестування, відсутні роботи, що пропонують перехід від якісного аналізу розбіжностей до кількісного оцінювання якості ПЗ через інтегральні метрики поведінкової стабільності. Розробка такого підходу дозволить збагатити теоретичну базу методів забезпечення якості та відкриє новий напрямок досліджень на перетині диференціального тестування та Software Quality Metrics.

Середній час виявлення регресії у production-середовищі становить 4,2 дні, а вартість виправлення дефекту на цьому етапі у 100 разів вища, ніж на етапі розробки. Автоматизоване оцінювання якості, інтегроване в CI/CD конвеєри, може скоротити цей час до 6–8 годин, що дає потенційну економію до 85%. Крім того, наявність об'єктивних кількісних показників якості дозволить приймати обґрунтовані рішення про готовність продукту до випуску релізу.

Впровадження методу оцінювання якості на основі диференціального тестування є особливо актуальним для систем зі складною логікою, де наявні кілька незалежних реалізацій або версій продукту (компілятори, СУБД, криптографічні бібліотеки, веб-браузери), а також для проектів, що використовують практики безперервної доставки та потребують швидкого зворотного зв'язку про стан якості.

Мета роботи - розробка методу кількісного оцінювання якості програмного забезпечення, що базується на аналізі розбіжностей у функціональній поведінці між різними версіями системи, виявлених за допомогою диференціального тестування.

Об'єктом дослідження є процеси та інструменти забезпечення якості програмного забезпечення на етапах його розробки, супроводу та еволюції в умовах застосування гнучких методологій розробки та практик безперервної інтеграції і доставки.

Предметом дослідження є методи генерації, виконання й порівняння результатів тестування у диференціальному підході, а також засоби кількісного оцінювання якості програмного забезпечення на основі аналізу поведінкових розбіжностей між різними реалізаціями або версіями систем.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– проаналізувати сучасні підходи до тестування та оцінювання якості програмного забезпечення, виявити їхні переваги, недоліки та обмеження застосування, зокрема у контексті методологій Agile та DevOps;

– дослідити теоретичні основи диференціального тестування, його відмінності від регресійного тестування, та визначити ключові параметри для аналізу поведінкових розбіжностей між версіями ПЗ, що можуть бути використані для оцінювання якості;

– розробити концептуальну модель та формалізувати математичний апарат методу оцінювання якості на основі диференціального тестування, включаючи класифікацію типів розбіжностей та систему вагових коефіцієнтів серйозності;

– запропонувати комплекс кількісних метрик поведінкової стабільності (коефіцієнт розбіжності, зважений індекс серйозності, індекс функціональної стабільності, вектор профілю якості), що дозволяють отримати багатовимірну оцінку якості програмного забезпечення;

– спроектувати архітектуру програмної системи для реалізації розробленого методу, що забезпечує автоматизацію процесів генерації тестів, виконання, аналізу розбіжностей та розрахунку метрик, а також можливість інтеграції в конвеєри CI/CD;

– провести програмну реалізацію розробленого методу з використанням сучасних технологій та фреймворків та виконати експериментальні дослідження на прикладі реальних програмних проектів для підтвердження ефективності та практичної значущості методу.

Для досягнення поставленої мети роботи використовувалися наступні методи дослідження методи системного аналізу, методи теорії множин та математичної логіки методи теорії алгоритмів, методи емпіричного дослідження, методи порівняльного аналізу.

Наукова новизна результатів дисертаційного дослідження полягає у наступному:

1.Вперше запропоновано метод кількісного оцінювання якості програмного забезпечення, що системно використовує результати диференціального тестування для формування інтегральної оцінки поведінкової стабільності систем через комплекс взаємопов'язаних метрик.

2.Удосконалено підхід до класифікації розбіжностей у диференціальному тестуванні шляхом розробки чотирирівневої ієрархічної системи із введенням функції вагових коефіцієнтів серйозності, що дозволяє диференційовано враховувати вплив різних типів розбіжностей на загальну оцінку якості, на відміну від існуючих методів, що розглядають усі розбіжності рівноцінно.

3.Набуло подальшого розвитку застосування диференціального тестування в контексті безперервної інтеграції та доставки через розробку концептуальної моделі інтеграції методу як автоматичних воріт якості (quality gates), що використовують порогові значення метрик FSI для прийняття рішень про готовність збірки до просування по конвеєру, що забезпечує безперервний моніторинг якості без втручання людини.

Практичне значення отриманих результатів полягає у розробці та програмній реалізації інструментального засобу для автоматизованого оцінювання якості програмного забезпечення, що може бути інтегрований у сучасні процеси розробки.

Відповідно до теми кваліфікаційної роботи опубліковані тези «Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки» конференції «Актуальні проблеми комп’ютерних наук (АПКН-2025)».

1 ТЕОРЕТИЧНИЙ ВИКЛАД ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області

У сучасному світі програмне забезпечення (ПЗ) є невід'ємною частиною практично всіх сфер людської діяльності - від критично важливих інфраструктурних систем (банківські системи, медичне обладнання, авіаційне управління) до повсякденних споживчих застосунків (мобільні застосунки, вебсервіси). За даними Світового економічного форуму 2024, цифрова економіка становить 15.5% світового ВВП, що підкреслює критичну важливість якості програмного забезпечення. Надійність, безпека, продуктивність та відповідність функціональним вимогам є ключовими факторами, що визначають успіх програмного продукту на ринку та довіру до нього з боку користувачів. Процес забезпечення якості перетворився з допоміжної діяльності на фундаментальну складову життєвого циклу розробки програмного забезпечення, що вимагає застосування науково обґрунтованих методів та інструментів.

Метою даного розділу є проведення систематичного та всебічного аналізу предметної області, пов'язаної з оцінюванням якості програмного забезпечення. Дослідження розпочинається з розгляду теоретичних основ та еволюції методологій тестування, що дозволяє зрозуміти історичний контекст та рушійні сили, які сформували сучасні підходи до QA. Далі проводиться класифікація та порівняльний аналіз існуючих методів тестування, що дозволяє структурувати наявні знання та визначити місце і роль кожного підходу в загальній стратегії забезпечення якості.

Центральне місце в розділі посідає поглиблений аналіз диференціального тестування як потужного інструменту для дослідження функціональної поведінки складних програмних систем. Розглядаються його концептуальні засади, переваги порівняно з традиційними методами, такими як регресійне тестування, та його здатність вирішувати одну з фундаментальних проблем тестування — проблему тестового оракула. Окрему увагу приділено сучасним інструментальним засобам

автоматизації та питанням інтеграції процесів тестування в конвеєри безперервної інтеграції та доставки (CI/CD), що є стандартом для сучасних гнучких методологій розробки.

Аналіз існуючих рішень дозволить виявити невирішені проблеми та білі плями в досліджуваній галузі. Синтез отриманих результатів та виявлення дослідницького розриву (research gap) стане логічним завершенням розділу і ляже в основу постановки задачі дослідження, обґрунтовуючи актуальність та наукову новизну теми роботи, що розкриватиметься у наступних розділах.

Для глибокого розуміння сучасних методів оцінювання якості ПЗ необхідно розглянути фундаментальні принципи, що лежать в основі процесу тестування, а також простежити еволюцію методологій розробки, яка безпосередньо вплинула на підходи до забезпечення якості.

Процес тестування програмного забезпечення ґрунтується на низці загальновизнаних принципів, які були сформульовані та стандартизовані такими організаціями, як ISTQB (International Software Testing Qualifications Board). Ці принципи є основою для побудови ефективних стратегій тестування. Одним з ключових є твердження, що тестування демонструє наявність дефектів, а не їх відсутність [13]. Це означає, що навіть найретельніше тестування не може гарантувати стовідсоткової безпомилковості програми; воно лише знижує ймовірність наявності невиявлених дефектів до прийнятного рівня. Інший важливий принцип полягає в тому, що тестування залежить від контексту [13]. Методи та обсяг тестування, що застосовуються для розважального мобільного застосунку, кардинально відрізнятимуться від тих, що використовуються для програмного забезпечення медичного обладнання або авіаційної системи, де ціна помилки є надзвичайно високою. Також важливим є принцип раннього тестування, який стверджує, що виявлення та виправлення дефектів на ранніх етапах життєвого циклу розробки є значно ефективнішим і дешевшим [14].

Історично розвиток практик тестування тісно пов'язаний з еволюцією методологій розробки програмного забезпечення. Кожна нова методологія

висувала нові вимоги до процесу забезпечення якості, змушуючи його адаптуватися та розвиватися.

Каскадна модель (Waterfall) є однією з перших формалізованих методологій, що характеризується суворою послідовністю етапів: аналіз вимог, проєктування, розробка, тестування, впровадження та супровід [15]. У рамках цієї моделі тестування є окремим, ізольованим етапом, який починається лише після повного завершення етапу розробки [16]. Такий підхід має свої переваги, зокрема чітку структуру та передбачуваність для проєктів зі стабільними, заздалегідь визначеними вимогами [17]. Однак його головним недоліком є пізнє виявлення дефектів. Помилки, допущені на етапі аналізу вимог або проєктування, виявляються лише наприкінці циклу, що робить їх виправлення надзвичайно складним та дорогим [16].

Зростання динамічності ринку та вимог до швидкості випуску продуктів призвело до появи гнучких методологій (Agile). Agile - це ітеративний підхід, що акцентує увагу на співпраці, гнучкості та швидкій реакції на зміни [15]. Процес розробки розбивається на короткі цикли, або спринти (зазвичай 2-4 тижні), наприкінці кожного з яких команда постачає робочу версію продукту з новим функціоналом. У контексті Agile тестування перестає бути окремою фазою і стає невід'ємною частиною кожного спринту. Тестувальники працюють у тісній співпраці з розробниками та аналітиками протягом усього циклу, забезпечуючи безперервний зворотний зв'язок [17].

Подальшим розвитком ідей гнучкості та автоматизації стала концепція DevOps яка об'єднує розробку та експлуатацію в єдиний інтегрований процес з фокусом на автоматизацію та співпрацю. Основною метою DevOps максимальне скорочення часу від написання коду до його розгортання у робочому середовищі без втрати якості. Центральною складовою цього підходу є автоматизація всіх етапів життєвого циклу, включаючи збірку, тестування, розгортання та моніторинг, що реалізується через конвеєри безперервної інтеграції та безперервної доставки [19]. В умовах DevOps тестування стає повністю інтегрованим та

високоавтоматизованим процесом. Автоматизовані тести виконуються при кожній зміні коду, забезпечуючи миттєвий фідбек про стан системи [20].

На рисунку 1.1 візуалізовано еволюцію методологій розробки програмного забезпечення та відповідну зміну ролі забезпечення якості.

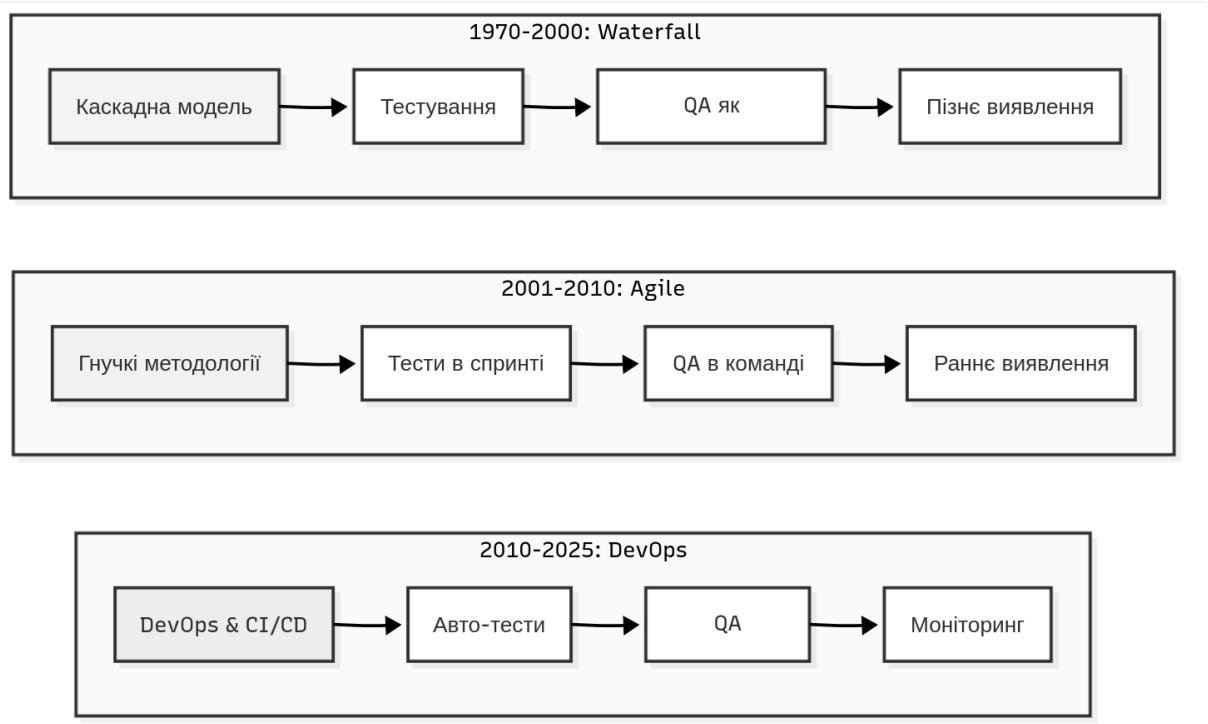


Рисунок 1.1 - Еволюція методологій розробки та тестування ПЗ

Дана діаграма ілюструє, як з переходом від каскадної моделі до Agile, а потім до DevOps, роль тестування трансформувалася від фінального контролю до інтегрованого та безперервного процесу, що підкреслює необхідність в автоматизованих та ефективних методах оцінки якості.

1.2 Аналіз існуючих рішень

Сфера тестування програмного забезпечення охоплює велику кількість різноманітних методів та технік, вибір яких залежить від контексту проєкту, цілей

тестування, доступних ресурсів та етапу життєвого циклу розробки [13]. Для систематизації цих підходів їх зазвичай класифікують за кількома ключовими критеріями. Найбільш поширеними є класифікація за моментом виконання (статичне та динамічне тестування) та за рівнем доступу до внутрішньої структури системи (тестування чорної, білої та сірої скриньки).

Одним з фундаментальних поділів у тестуванні є розрізнення між статичним та динамічним аналізом. Ці два підходи відповідають двом різним цілям забезпечення якості: верифікації та валідації. Верифікація - це процес перевірки того, чи створюється продукт правильно, тобто чи відповідає він своїм специфікаціям. Валідація - це процес перевірки того, чи створюється правильний продукт, тобто чи відповідає він потребам користувачів.

Статичне тестування є методом верифікації, який полягає в аналізі програмного продукту та пов'язаних з ним артефактів (вимог, проєктної документації, вихідного коду) без фактичного виконання коду. Основна мета статичного тестування - запобігання дефектам шляхом їх виявлення на якомога більш ранніх етапах SDLC. До технік статичного тестування належать:

– ревю (Reviews) - процес аналізу документів або коду групою фахівців. Це може бути інспекція, технічний огляд або неформальний перегляд;

– статичний аналіз коду - автоматизована перевірка вихідного коду за допомогою спеціалізованих інструментів (статичних аналізаторів), які виявляють потенційні помилки, вразливості, порушення стандартів кодування та інші проблеми.

Головною перевагою статичного тестування є можливість раннього виявлення дефектів, що значно знижує вартість їх виправлення. Воно дозволяє покращити якість коду та документації ще до того, як вони будуть передані на наступні етапи розробки. Однак статичне тестування має й обмеження: воно не здатне виявити помилки, що проявляються лише під час виконання програми (наприклад, проблеми з продуктивністю, логічні помилки, пов'язані з взаємодією компонентів, або помилки, залежні від середовища).

Динамічне тестування, на відміну від статичного, є методом валідації, який вимагає виконання програмного коду. Його мета - виявлення дефектів шляхом спостереження за поведінкою програми під час її роботи. Тестувальник взаємодіє з системою, надаючи їй певні вхідні дані та перевіряючи, чи відповідають отримані результати очікуванням . Динамічне тестування дозволяє оцінити як функціональні (чи робить система те, що повинна), так і нефункціональні (наскільки добре вона це робить - продуктивність, надійність, зручність використання) аспекти програмного забезпечення.

Перевагою динамічного тестування є його здатність виявляти складні дефекти, які неможливо знайти статичними методами, оскільки воно перевіряє систему в умовах, наближених до реальної експлуатації. Проте, цей підхід має і суттєві недоліки.

Таким чином, статичний та динамічний аналіз не є взаємовиключними, а навпаки, доповнюють один одного. Ефективна стратегія забезпечення якості повинна поєднувати обидва підходи для досягнення максимального покриття та раннього виявлення широкого спектра дефектів.

На рисунку 1.2 наведено класифікацію методів тестування, що ілюструє їх поділ за ключовими критеріями.

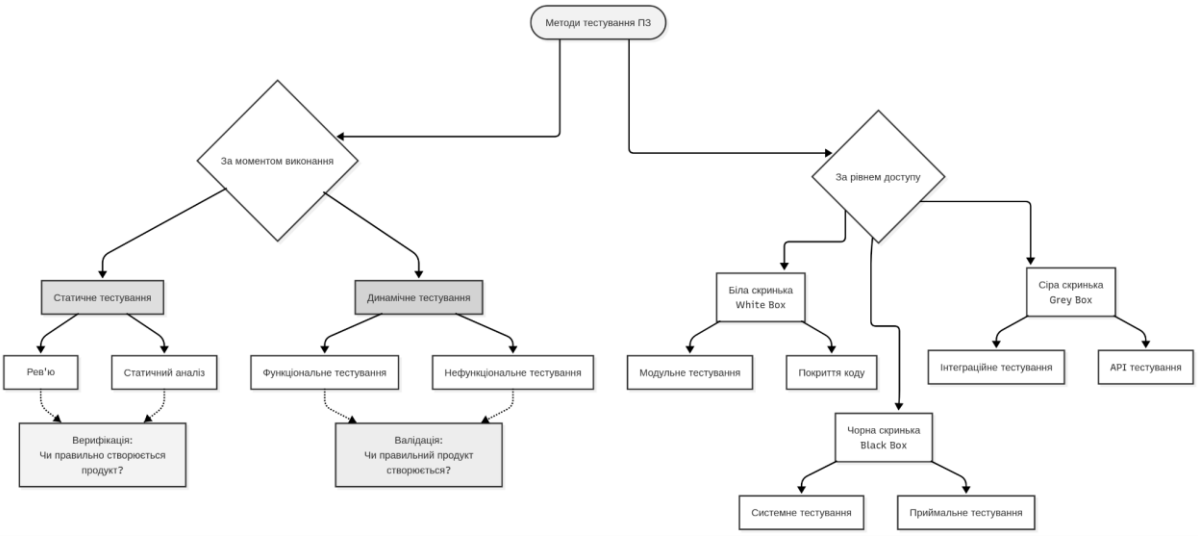


Рисунок 1.2. - Класифікація методів тестування ПЗ

Іншою важливою класифікацією методів тестування є поділ за рівнем знань тестувальника про внутрішню структуру та реалізацію системи, що тестується. Цей критерій визначає підхід до проєктування тест-кейсів та вибору тестових даних.

Тестування білої скриньки також відоме як структурне або прозоре тестування, передбачає, що тестувальник має повний доступ до вихідного коду, архітектури та внутрішньої логіки програми. Тест-кейси проєктуються на основі аналізу коду з метою перевірки конкретних шляхів виконання, гілок, умов та циклів. Цей метод дозволяє досягти високого рівня покриття коду та виявити помилки в реалізації алгоритмів, але вимагає від тестувальника навичок програмування. Зазвичай тестування білої скриньки застосовується на рівні модульного та інтеграційного тестування і часто виконується самими розробниками.

Тестування чорної скриньки є протилежністю до білої скриньки. При цьому підході система розглядається як непрозорий об'єкт, внутрішня структура якого невідома та неважлива для тестувальника. Тестування базується виключно на аналізі вимог та специфікацій. Тестувальник взаємодіє з системою через її зовнішні інтерфейси (графічний інтерфейс користувача, API), подаючи на вхід дані та перевіряючи, чи відповідають вихідні дані очікуваним результатам. Цей метод не вимагає знань програмування і дозволяє перевірити систему з погляду кінцевого користувача, виявляючи помилки у функціональності, логіці та відповідності вимогам. Однак він не гарантує повного покриття коду і може пропустити помилки, що не проявляються через зовнішні інтерфейси.

Тестування сірої скриньки є гібридним підходом, що поєднує елементи білої та чорної скриньок. Тестувальник має часткові знання про внутрішню структуру системи, наприклад, про архітектуру, базу даних або алгоритми обробки даних, але не має доступу до вихідного коду. Ці знання дозволяють проєктувати більш ефективні та цілеспрямовані тест-кейси, які перевіряють не тільки зовнішню поведінку, але й внутрішні стани системи. Цей підхід дозволяє досягти кращого балансу між глибиною тестування та зусиллями, необхідними для його проведення. Вибір між цими підходами залежить від цілей тестування. Для перевірки

коректності реалізації алгоритмів найкраще підходить біла скринька, для валідації функціональності з погляду користувача - чорна скринька, а для комплексного тестування, що враховує як зовнішню, так і внутрішню логіку, - сіра скринька.

1.3 Огляд методів вирішення проблеми

Після розгляду загальних класифікацій методів тестування, даний підрозділ зосереджується на диференціальному тестуванні - специфічному, але надзвичайно потужному підході, який є центральним для даного дослідження. Диференціальне тестування належить до динамічних методів тестування за типом чорної скриньки і пропонує унікальне рішення для тестування складних систем.

Диференціальне тестування використовує декілька порівнюваних реалізацій однієї і тієї ж специфікації як взаємні тестові оракули. Основний принцип полягає в тому, що для однакового набору вхідних даних усі коректні реалізації системи повинні генерувати еквівалентні результати. Якщо при виконанні одного й того ж тесту на різних версіях системи їхня поведінка або результати відрізняються, це свідчить про наявність потенційного дефекту принаймні в одній з них. Такими порівнюваними реалізаціями можуть бути:

- різні версії одного й того ж програмного продукту (наприклад, поточна та попередня стабільна версія);
- продукти різних розробників, що реалізують однаковий стандарт, наприклад, різні веб-браузери, компілятори мов програмування або системи керування базами даних;
- одна й та ж програма, що працює в різних конфігураціях або середовищах.

Цей підхід є особливо ефективним для систем, де визначення правильного результату є надзвичайно складним або неможливим завданням. Наприклад, для компілятора важко заздалегідь передбачити, який саме оптимізований машинний

код він має згенерувати, але можна очікувати, що скомпільовані різними компіляторами програми будуть поводитися однаково.

Диференціальне тестування часто порівнюють з регресійним тестуванням, оскільки обидва методи спрямовані на виявлення змін у поведінці системи. Однак між ними існують принципові відмінності. Традиційне регресійне тестування порівнює нову версію системи S' з наперед визначеним, часто створеним вручну, набором тестів TS, який описує очікувану поведінку старої версії S. Головною проблемою цього підходу є проблема ремонту тестів. Коли в системі відбуваються легітимні зміни (наприклад, додається новий функціонал), частина регресійних тестів починає зазнавати невдачі, і їх необхідно вручну аналізувати та оновлювати, що є трудомістким і схильним до помилок процесом.

Диференціальне тестування вирішує цю проблему, усуваючи необхідність у ручному ремонті тестів. Замість порівняння з фіксованим набором тестів, воно порівнює дві версії системи S та S' між собою, використовуючи автоматично згенеровані тестові набори для обох версій TS та TS'. Такий підхід дозволяє не просто виявити розбіжності, а й класифікувати їх на три категорії:

– збережена поведінка $T_{Preserved}$, тобто тести, які успішно проходять на обох версіях;

– регресована поведінка $T_{Regressed}$, тобто тести, які успішно проходили на старій версії S, але зазнають невдачі на новій S'. Це вказує на потенційну регресію або баг;

– прогресована поведінка $T_{Progressed}$, тобто тести, які зазнавали невдачі на старій версії S, але успішно проходять на новій S'. Це зазвичай відповідає новому або зміненому функціоналу.

Таким чином, диференціальне тестування надає більш повну картину змін у поведінці системи, ніж традиційне регресійне тестування, і є більш пристосованим до умов швидкої та безперервної розробки.

Однією з найскладніших і найстаріших проблем у тестуванні програмного забезпечення є проблема тестового оракула. Тестовий оракул - це механізм,

джерело інформації або принцип, який дозволяє визначити, чи є результат виконання тесту правильним чи ні. Для простих функцій, наприклад, додавання двох чисел, оракул є тривіальним. Однак для складних систем, таких як системи обробки зображень, компілятори або фінансові моделі, визначення точного очікуваного результату для кожного можливого вхідного значення може бути надзвичайно складним, дорогим або навіть неможливим.

Ручне визначення очікуваних результатів є головним вузьким місцем, що перешкоджає повній автоматизації тестування. Навіть якщо генерація вхідних даних та виконання тестів автоматизовані, перевірка результатів часто вимагає участі людини, що є повільним, ненадійним і не масштабованим процесом. Диференціальне тестування пропонує елегантне та практичне вирішення цієї проблеми. Воно використовує неявний оракул, де роль оракула виконують самі системи, що тестуються. Замість того щоб порівнювати результат роботи програми з наперед визначеним еталоном, диференціальне тестування порівнює результати роботи кількох програм між собою. Оракулом стає консенсус у поведінці: якщо всі (або більшість) реалізацій дають однаковий результат, він вважається правильним. Відхилення від цього консенсусу сигналізує про потенційну помилку. Цей підхід не вимагає наявності формальної специфікації або ручного визначення очікуваних результатів, що дозволяє автоматизувати тестування навіть для дуже складних систем, де традиційні підходи є непридатними.

Ефективність диференціального тестування безпосередньо залежить від двох ключових компонентів: здатності генерувати різноманітні та цікаві вхідні дані, що можуть спровокувати розбіжності, та наявності надійних методів для аналізу та порівняння результатів.

Генерація вхідних даних може здійснюватися за допомогою різних стратегій. Найпростішою є випадкова генерація, яка попри свою простоту, може бути ефективною для виявлення несподіваних помилок. Більш складні підходи використовують структуровану генерацію, наприклад, на основі стохастичних граматик, що дозволяє створювати синтаксично коректні, але семантично складні вхідні дані (наприклад, складні програми на мові C для тестування компіляторів).

Сучасним та надзвичайно потужним підходом є диференціальний фазинг. Фазинг - це техніка автоматизованого тестування, яка полягає у подачі на вхід програми великої кількості напіввипадкових, некоректних або несподіваних даних (fuzz) з метою викликати збої, витоки пам'яті або інші аномалії. Сучасні фазери, такі як AFL, використовують керований покриттям підхід, де генерація нових вхідних даних спрямовується на дослідження нових шляхів виконання у програмі, що робить процес пошуку помилок значно ефективнішим. Диференціальний фазинг поєднує потужність фазингу для генерації вхідних даних з оракулом диференціального тестування. Ті самі згенеровані фазером дані подаються на вхід кільком версіям програми, а їхні результати порівнюються.

Виявлення розбіжностей також є нетривіальним завданням. Просте побайтове порівняння результатів часто є недостатнім, оскільки можуть існувати несуттєві відмінності (наприклад, у форматуванні виводу, часових мітках або ідентифікаторах), які не є ознакою помилки. Для фільтрації такого шуму можуть застосовуватися більш складні методи. Наприклад, при тестуванні алгоритмів машинного навчання, де результати можуть мати стохастичний характер, для порівняння розподілів класів можуть використовуватися статистичні критерії, такі як критерій хі-квадрат. В інших областях, наприклад, тестування баз даних, можуть застосовуватися методи семантичного порівняння.

Застосування диференціального тестування фундаментально змінює саму мету тестування. Замість традиційного підходу, що полягає у перевірці відповідності специфікації, диференціальне тестування пропонує парадигму пошуку поведінкових невідповідностей. Це має глибокі наслідки для тестування систем, що швидко еволюціонують або не мають повної формальної специфікації, що є типовим для Agile та DevOps середовищ. Традиційне тестування стикається з непереборною проблемою оракула, коли специфікація неповна або застаріла. Диференціальне тестування обходить цю проблему, використовуючи попередні версії системи або аналогічні продукти як *de facto* специфікацію. Це зміщує фокус з бінарного вердикту пройшов чи не пройшов на більш глибокий, дослідницький аналіз поведінкового дрейфу системи з часом. Саме ця здатність виявляти та

потенційно кількісно оцінювати зміни у функціональній поведінці робить диференціальне тестування ідеальною основою для розробки нового методу оцінювання якості ПЗ.

1.4 Постановка задачі

Проведений у даному розділі аналіз предметної області та огляд літературних джерел дозволяє сформулювати низку ключових висновків, які визначають контекст та обґрунтовують актуальність подальшого дослідження. По-перше, еволюція методологій розробки від каскадної моделі до Agile та DevOps кардинально змінила вимоги до процесу забезпечення якості. По-друге, серед різноманіття методів тестування особливе місце посідає динамічний аналіз за принципом чорної скриньки, оскільки він дозволяє валідувати функціональну поведінку системи з погляду кінцевого користувача. По-третє, диференціальне тестування представляє собою потужний підхід, що ефективно вирішує фундаментальну проблему тестового оракула. По-четверте, сучасна індустрія програмного забезпечення має у своєму розпорядженні зрілі та потужні інструментальні засоби, які дозволяють практично реалізувати найскладніші стратегії автоматизованого тестування.

Мета роботи - розробка методу кількісного оцінювання якості програмного забезпечення, що базується на аналізі розбіжностей у функціональній поведінці між різними версіями системи, виявлених за допомогою диференціального тестування.

Об'єктом дослідження є процеси та інструменти забезпечення якості програмного забезпечення на етапах його розробки, супроводу та еволюції в умовах застосування гнучких методологій розробки та практик безперервної інтеграції і доставки.

Предметом дослідження є методи генерації, виконання й порівняння результатів тестування у диференціальному підході, а також засоби кількісного оцінювання якості програмного забезпечення на основі аналізу поведінкових розбіжностей між різними реалізаціями або версіями систем.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– проаналізувати сучасні підходи до тестування та оцінювання якості програмного забезпечення, виявити їхні переваги, недоліки та обмеження застосування, зокрема у контексті методологій Agile та DevOps;

– дослідити теоретичні основи диференціального тестування, його відмінності від регресійного тестування, та визначити ключові параметри для аналізу поведінкових розбіжностей між версіями ПЗ, що можуть бути використані для оцінювання якості;

– розробити концептуальну модель та формалізувати математичний апарат методу оцінювання якості на основі диференціального тестування, включаючи класифікацію типів розбіжностей та систему вагових коефіцієнтів серйозності;

– запропонувати комплекс кількісних метрик поведінкової стабільності (коефіцієнт розбіжності, зважений індекс серйозності, індекс функціональної стабільності, вектор профілю якості), що дозволяють отримати багатовимірну оцінку якості програмного забезпечення;

– спроектувати архітектуру програмної системи для реалізації розробленого методу, що забезпечує автоматизацію процесів генерації тестів, виконання, аналізу розбіжностей та розрахунку метрик, а також можливість інтеграції в конвеєри CI/CD;

– провести програмну реалізацію розробленого методу з використанням сучасних технологій та фреймворків та виконати експериментальні дослідження на прикладі реальних програмних проектів для підтвердження ефективності та практичної значущості методу.

1.5 Висновки до 1-го розділу

У першому розділі проведено аналіз сучасного стану проблеми забезпечення якості програмного забезпечення, досліджено еволюцію методологій розробки та їх вплив на процеси тестування, виконано класифікацію та порівняльний аналіз методів тестування, детально розглянуто концепцію диференціального тестування, проблему тестового оракула та сучасні інструментальні засоби автоматизації. На основі критичного аналізу літературних джерел виявлено дослідницький розрив та сформульовано постановку задачі дослідження.

2 МЕТОД ПІДТРИМКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ДИФЕРЕНЦІАЛЬНОГО ТЕСТУВАННЯ

2.1 Концептуальна модель для методу підтримки якості програмного забезпечення на основі диференціального тестування

Аналіз літературних джерел, проведений у попередньому розділі, дозволив виявити ключову прогалину в існуючих підходах до забезпечення якості програмного забезпечення. Було встановлено, що диференціальне тестування, незважаючи на свою доведену ефективність як інструмент для виявлення дефектів, зокрема у складних системах без чітко визначеного оракула, залишається недооціненим у контексті кількісного оцінювання якості. Наявні методики здебільшого фокусуються на бінарному результаті знайдено помилку чи не знайдено помилку, ігноруючи великий масив даних, що генерується в процесі порівняння поведінки кількох програмних реалізацій. Цей масив даних про розбіжності у поведінці систем містить значний потенціал для формування комплексної, багатовимірної оцінки якості ПЗ, який досі залишається нереалізованим.

Метою даного розділу є розробка нового, формалізованого методу оцінювання якості програмного забезпечення, який системно використовує принципи диференціального тестування для отримання об'єктивних кількісних показників. Відповідно до вимог до науково-дослідної роботи, цей розділ представляє теоретичні основи, оригінальну концептуальну модель, математичний апарат та алгоритм запропонованого методу. Розробка орієнтована на створення методики, що дозволяє не лише виявляти дефекти, а й формувати інтегральну оцінку стабільності, надійності та консистентності програмного продукту. Структура розділу побудована для послідовного викладення результатів дослідження. Спочатку розглядаються теоретичні передумови, що обґрунтовують можливість застосування диференціального тестування для цілей оцінювання

якості. Далі представляється концептуальна модель методу, що описує його ключові компоненти та взаємозв'язки.

Третій підрозділ присвячений формалізації методу та розробці його математичного апарату. На основі формальної моделі розробляється покроковий алгоритм застосування методу. П'ятий підрозділ детально описує комплекс нових метрик якості, що є ключовим результатом роботи. Завершується розділ порівняльним аналізом розробленого методу з існуючими підходами, що дозволяє визначити його переваги, недоліки та місце

Традиційно диференціальне тестування розглядається як техніка для виявлення помилок (bug finding), де основною метою є знаходження таких вхідних даних, за яких поведінка двох або більше порівнюваних систем відрізняється. Проте фундаментальний принцип, що лежить в основі цього підходу, відкриває значно ширші можливості. Замість того, щоб розглядати диференціальне тестування як інструмент для пошуку окремих дефектів, пропонується переосмислити його як потужний механізм для генерації даних, що характеризують загальну якість та стабільність програмної системи.

Центральною ідеєю є перехід від аналізу окремих розбіжностей до аналізу їх сукупності. Кожна виявлена розбіжність (дивергенція) є не просто свідченням помилки, а точкою даних у багатовимірному просторі поведінки ПЗ. Сукупність усіх виявлених дивергенцій, їх типів, частоти та умов виникнення формує унікальний поведінковий відбиток системи, що характеризує її консистентність та стабільність у всьому просторі вхідних даних, що покривається тестами. Висока кількість розбіжностей, або наявність розбіжностей критичного характеру, свідчить не про ізольовані дефекти, а про системні проблеми з якістю, такі як архітектурні недоліки, неоднозначність специфікацій або слабкість процесів розробки та тестування.

В основі методу лежить концепція псевдо-оракула, що базується на принципі N-версійного програмування. За відсутності еталонної реалізації або формальної специфікації, що може слугувати тестовим оракулом, припускається, що якщо більшість незалежно розроблених реалізацій одного й того ж алгоритму дають

однаковий результат для певного входу, то цей результат, ймовірно, є правильним. Розбіжність у результаті однієї з версій сигналізує про потенційний дефект. Важливо зазначити, що цей підхід не гарантує абсолютної коректності (усі версії можуть містити однакову помилку), однак він є надзвичайно ефективним для виявлення неконсистентності, яка є одним з найважливіших індикаторів низької якості ПЗ.

Для побудови ефективного методу оцінювання якості необхідно класифікувати можливі типи розбіжностей за ступенем їх критичності. Пропонується наступна класифікація:

Катастрофічні розбіжності відносяться до найвищого рівні критичності. До цієї категорії належать збої, що призводять до повного припинення роботи програми: аварійне завершення, зависання, необроблені винятки, помилки сегментації пам'яті. Такі розбіжності свідчать про серйозні проблеми зі стабільністю та надійністю ПЗ.

Семантичні розбіжності стосуються суті виконуваної функції, тобто це отримання різних результатів обчислень, некоректна логічна поведінка, порушення інваріантів алгоритму. Наприклад, два криптографічні шифратори, реалізуючи один і той самий стандарт, видають різний шифротекст для однакових вхідних даних та ключа. Це свідчить про фундаментальні помилки в реалізації бізнес-логіки.

Поведінкові розбіжності – це відмінності у нефункціональних характеристиках. До цієї категорії належать суттєві розбіжності у часі виконання або обсязі споживаної пам'яті. Наприклад, одна реалізація виконує завдання за мілісекунди, а інша - за десятки секунд. Хоча обидві можуть давати однаковий результат, така різниця може бути критичною для систем реального часу або високонавантажених сервісів.

Презентаційні розбіжності є найменш критичнии типом Це незначні відмінності у форматуванні вихідних даних, які не впливають на їх семантичну суть. Наприклад, різниця у кількості знаків після коми для чисел з плаваючою

комою, різний порядок несортованих елементів у вихідному масиві, відмінності у текстовому поданні повідомлень про помилки.

Така класифікація дозволяє не просто фіксувати факт розбіжності, а й оцінювати його вплив на загальну якість продукту, що є фундаментом для розробки зважених кількісних метрик. Область застосування запропонованого підходу є досить широкою, але він демонструє найбільшу ефективність у наступних сценаріях:

– системи зі складною логікою та відсутністю формальної специфікації, наприклад, компілятори, інтерпретатори мов програмування, системи символьних обчислень, криптографічні бібліотеки, де формальна верифікація є надзвичайно складною;

– наявність кількох незалежних реалізацій, тобто наявність на ринку кількох продуктів, що вирішують одну й ту саму задачу, наприклад, різні SQL-сервери, веббраузери, бібліотеки для обробки зображень, створює ідеальні умови для диференціального тестування;

– регресійне тестування, тобто порівняння нової версії програмного продукту з попередньою стабільною версією є потужним методом виявлення регресійних помилок. У цьому випадку попередня версія виступає в ролі еталонної реалізації.

Таким чином, теоретичною основою методу є зміщення парадигми від пошуку помилок до аналізу поведінкової стабільності. Диференціальне тестування перетворюється з техніки тестування на інструмент вимірювання, що генерує великий набір даних, придатний для глибокого кількісного аналізу та формування об'єктивної оцінки якості програмного забезпечення.

Для систематизації процесу оцінювання якості на основі диференціального тестування розроблено концептуальну модель, що визначає ключові компоненти системи, їх функції та інформаційні потоки між ними. Ця модель слугує високорівневим архітектурним планом для реалізації методу та відповідає вимозі щодо розробки оригінальних моделей у науково-дослідній роботі. Модель складається з чотирьох основних логічних модулів, що забезпечують повний цикл

від генерації тестових даних до розрахунку фінальних метрик якості. Компоненти концептуальної моделі:

– модуль генерування тестових даних;

– модуль виконання та моніторингу;

– аналізатор розбіжностей;

– калькулятор метрик якості.

Модуль генерування тестових даних відповідає за створення великого та різноманітного набору вхідних даних, що є критично важливим для забезпечення достатнього покриття функціонального простору ПЗ. Ефективність усього методу значною мірою залежить від якості та обсягу тестових даних. Модуль може реалізовувати різні стратегії генерації, зокрема:

– випадкова генерація простий підхід, що генерує дані у випадковому порядку в межах заданих обмежень;

– фаззінг - генерація невалідних, неочікуваних або екстремальних даних з метою виявлення вразливостей та збоїв;

– тестування на основі властивостей - генерація даних, що задовольняють певним властивостям або інваріантам, що дозволяє цілеспрямовано тестувати специфічні сценарії;

– генерація на основі моделей - використання формальної моделі системи для генерації тестових послідовностей, що покривають певні стани або переходи.

Модуль виконання та моніторингу є основною функцією цього модуля - автоматизований запуск програмних реалізацій, що підлягають тестуванню (N версій), на кожному згенерованому тестовому наборі. Окрім простого виконання, модуль виконує критично важливу функцію моніторингу, збираючи вичерпну інформацію про кожен запуск. Ця інформація включає:

– вихідні дані програми, куди відноситься стандартний вивід, файли результатів;

– код повернення процесу;

– інформацію зі стандартного потоку помилок;

– нефункціональні параметри, тобто час виконання, пікове споживання оперативної пам'яті, використання CPU.

Ці дані є сировиною для подальшого аналізу.

Аналізатор розбіжностей є центральним компонентом моделі, де відбувається основна інтелектуальна робота. Він отримує результати виконання всіх програм для кожного вхідного тесту та проводить їх попарне порівняння. Його завдання полягають у наступному:

– виявлення розбіжностей, коли здійснюється порівняння вихідних даних, кодів повернення та метрик продуктивності для виявлення будь-яких відмінностей;

– класифікація розбіжностей, тобто присвоєння кожній виявленій розбіжності однієї з категорій (катастрофічна, семантична, поведінкова, презентаційна) на основі розробленої в попередньому підрозділі класифікації. Класифікація може відбуватися на основі аналізу кодів повернення (для катастрофічних збоїв), порівняння вмісту вихідних файлів (для семантичних та презентаційних) та аналізу метрик продуктивності (для поведінкових);

– логування, тобто детальне протоколювання кожної виявленої розбіжності, включаючи вхідні дані, що її спричинили, імена програм, що продемонстрували різну поведінку, та присвоєний клас розбіжності.

Калькулятор метрик якості є фінальним компонент моделі, який обробляє структуровані дані з логу розбіжностей. На основі формальних моделей та формул, що будуть представлені у наступних підрозділах, цей модуль обчислює набір кількісних показників якості. Результатом його роботи є не просто список помилок, а інтегральна оцінка, що може бути представлена у вигляді одного або кількох числових індексів та векторів, які характеризують різні аспекти якості ПЗ.

На рисунку 2.1 представлена блок-схема, що візуалізує взаємодію описаних компонентів, тобто концептуальна модель методу диференціального тестування Важливим аспектом розробленої концептуальної моделі є її потенціал для інтеграції в сучасні процеси розробки програмного забезпечення, зокрема в конвеєри безперервної інтеграції та безперервної доставки. Модульна структура та

високий рівень автоматизації дозволяють вбудувати цей метод як один з етапів автоматизованого тестування. Розраховані метрики якості, наприклад, інтегральний індекс стабільності, можуть використовуватися як ворота якості Якщо після внесення нових змін до коду цей індекс падає нижче визначеного порогового значення, збірка автоматично позначається як невдала, що запобігає потраплянню низькоякісного коду до основного репозиторію. Такий підхід перетворює метод з інструменту одноразового аналізу на систему безперервного моніторингу якості, що надає розробникам швидкий та об'єктивний зворотний зв'язок про вплив їхніх змін на стабільність продукту. Це є значною практичною перевагою, що підвищує цінність методу в індустріальному середовищі.

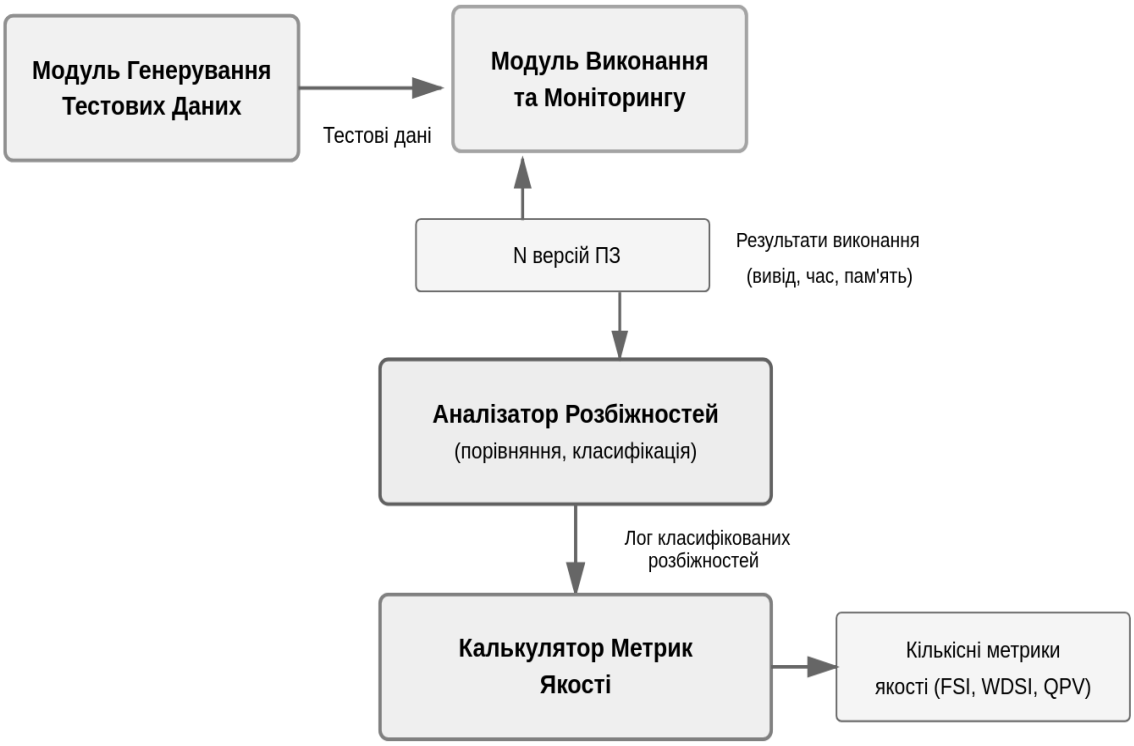


Рисунок 2.1 - Алгоритм 4-рівневої класифікації розбіжностей

Ця блок-схема детально ілюструє послідовність прийняття рішень для присвоєння кожній виявленій розбіжності відповідного рівня критичності, що є основою для розрахунку зважених метрик якості.

2.2 Розробка комплексу метрик якості на основі результатів диференціального тестування

Ключовим результатом запропонованого методу є не просто виявлення дефектів, а надання об'єктивної кількісної оцінки якості ПЗ. Для цієї мети розроблено комплекс взаємопов'язаних метрик, що дозволяють аналізувати якість на різних рівнях деталізації - від загальної інтегральної оцінки до детального профілю проблем. Розробка цих показників є центральним елементом наукової новизни роботи.

1. Коефіцієнт розбіжності (Divergence Rate, DR) є базовою метрика, що характеризує частоту виникнення неконсистентної поведінки. Вона розраховується як відношення кількості вхідних тестів, що спричинили будь-яку розбіжність, до загальної кількості виконаних тестів (1):

$$DR = \frac{\sum_{j=1}^{m} D(i_j)}{m} \quad (1)$$

де $D(i_j)$ - бінарна функція події розбіжності, визначена у підрозділі 2.3, а m - загальна кількість тестів.

Інтерпретація: DR приймає значення в діапазоні $[0, 1]$. $DR = 0$ означає повну консистентність поведінки всіх програм на тестовому наборі. $DR = 0.15$ означає, що 15% тестових випадків виявили розбіжності. Ця метрика дає загальне уявлення про стабільність, але не враховує серйозність виявлених проблем.

2. Зважений індекс серйозності розбіжностей (Weighted Divergence Severity Index, WDSI) є більш досконалою метрикою, оскільки вона враховує не лише факт наявності розбіжності, але і її тип (серйозність). Вона розраховується як сума вагових коефіцієнтів серйозності для всіх тестів, що викликали розбіжність, поділена на загальну кількість тестів.

Нехай d_j - подія розбіжності для входу i_j, а $C(d_j)$ - її клас (якщо розбіжності не було, вага дорівнює 0).

Тоді отримаємо (2):

$$WDSI = \frac{\sum_{j=1}^{m} W(C(d_j))}{m} \quad (2)$$

Інтерпретація: $WDSI$ є агрегованим показником, що відображає загальну шкоду від неконсистентності. На відміну від DR, цей індекс буде вищим, якщо розбіжності переважно катастрофічні або семантичні, і нижчим, якщо вони переважно презентаційні. Це дозволяє більш точно оцінити ризики, пов'язані з якістю ПЗ. Наприклад, система з $DR = 0.1$, де всі розбіжності катастрофічні, отримає вищий $WDSI$, ніж система з $DR = 0.2$, де всі розбіжності презентаційні.

3. Індекс функціональної стабільності (Functional Stability Index, FSI)

Це головна інтегральна метрика, призначена для надання єдиної, інтуїтивно зрозумілої оцінки якості. Вона нормалізує $WDSI$ і представляє якість у вигляді числа від 0 до 1, де 1 означає ідеальну якість (3):

$$FSI = \max(0, 1 - WDSI) \quad (3)$$

Використання функції $max(0,\cdot)$ гарантує, що індекс не стане від'ємним, якщо сума ваг теоретично перевищить m.

Інтерпретація: FSI є кінцевим показником якості. $FSI = 1$ свідчить про повну відсутність розбіжностей (ідеальна стабільність). $FSI = 0$ вказує на максимальний виявлений рівень нестабільності. Цей індекс зручно використовувати для відстеження динаміки якості продукту з часом, для порівняння різних продуктів між собою, а також як порогове значення у системах CI/CD.

4. Вектор профілю якості (Quality Profile Vector, QPV) використовуєтьсґ для глибокого аналізу характеру проблем якості. Цей вектор складається з коефіцієнтів розбіжності, розрахованих окремо для кожного класу (4).

$$QPV = langle\, DR_{cat}, DR_{sem}, DR_{beh}, DR_{pre} rangle \tag{4}$$

де DR_{class} - це коефіцієнт розбіжності, розрахований лише для розбіжностей певного класу (5):

$$DR_{class} = \frac{|\{ i_j \in I \; mid \; C(d_j) = C_{class}\}|}{m} \tag{5}$$

Інтерпретація: QPV надає детальну картину стану якості ПЗ. Наприклад, вектор $langle\, 0.01, 0.15, 0.03, 0.05\, rangle$ свідчить про те, що система має значні проблеми з семантичною коректністю (15% тестів виявили семантичні помилки), незначні проблеми зі стабільністю (1% катастрофічних збоїв) та певні поведінкові й презентаційні недоліки. Такий детальний аналіз дозволяє команді розробників пріоритезувати зусилля: у даному випадку, основну увагу слід приділити виправленню логічних помилок.

Разом ці чотири метрики створюють комплексну систему оцінювання, що дозволяє перейти від простої констатації наявності помилок до вимірюваного, керованого та аналітичного процесу управління якістю програмного забезпечення.

Для забезпечення наукової строгості, відтворюваності та можливості однозначної інтерпретації результатів, концептуальна модель методу повинна бути підкріплена формальним математичним апаратом. Цей підрозділ вводить основні визначення та математичні конструкції, що є фундаментом для алгоритму та метрик якості, що розробляються далі. Ця формалізація відповідає вимозі щодо наявності математичної постановки задачі в науково-дослідній роботі.

Нехай $P = \{p_1, p_2, \ldots, p_n\}$ - скінченна множина з n програмних реалізацій ($n\ ge\ 2$), що підлягають диференціальному тестуванню. Ці реалізації можуть бути

різними версіями одного продукту або незалежними продуктами, що реалізують однакову функціональність.

Нехай $I = \{i_1, i_2, \ldots, i_m\}$ - скінченна множина з m вхідних тестових випадків, згенерованих відповідним модулем.

Результатом виконання програми $p_k \in P$ на вхідних даних $i_j \in I$ є кортеж $O(p_k, i_j)$, що фіксує всі аспекти її поведінки (6):

$$O(p_k, i_j) = \langle r_{kj}, t_{kj}, \mu_{kj}, c_{kj} \rangle \tag{6}$$

де:

r_{kj} - результат виконання (наприклад, вміст стандартного потоку виводу або вихідного файлу);

t_{kj} - час виконання;

μ_{kj} - обсяг спожитої пам'яті;

c_{kj} - код завершення процесу.

Дві програмні реалізації p_a та p_b вважаються такими, що демонструють ідентичну поведінку на вході i_j, якщо їхні результати виконання є еквівалентними. Введемо відношення еквівалентності (7):

$$O(p_a, i_j) \equiv O(p_b, i_j) \Leftrightarrow (r_{aj} \approx r_{bj}) \wedge (|t_{aj} - t_{bj}| \leq \epsilon_t) \wedge (|\mu_{aj} - \mu_{bj}| \leq \epsilon_\mu) \wedge (c_{aj} = c_{bj}) \tag{7}$$

де:

$r_{aj} \approx r_{bj}$ означає семантичну еквівалентність результатів (може вимагати спеціалізованого компаратора, що ігнорує презентаційні відмінності);

ϵ_t та ϵ_μ - допустимі порогові відхилення для часу виконання та споживання пам'яті, що визначають межі нормальної поведінкової варіації.

Подія розбіжності для вхідних даних i_j визначається як бінарна функція $D(i_j)$: $D(i_j)$ = Значення $D(i_j) = 1$ свідчить про те, що принаймні дві програми з множини P поводилися по-різному на вході i_j.

Для кількісної оцінки якості недостатньо просто зафіксувати факт розбіжності; необхідно визначити її тип. Введемо множину класів розбіжностей $C = \{C_{cat}, C_{sem}, C_{beh}, C_{pre}\}$, що відповідають катастрофічній, семантичній, поведінковій та презентаційній розбіжностям.

Функція класифікації $Classify$ відображає пару нееквівалентних результатів $O(p_a, i_j)$ та $O(p_b, i_j)$ у множину класів C (8):

$$Classify\left(O(p_a, i_j), O(p_b, i_j)\right) \rightarrow C_{type} \in C \quad (8)$$

де $C = \{C_{cat}, C_{sem}, C_{beh}, C_{pre}\}$ - множина класів розбіжностей

Правила класифікації можуть бути формалізовані наступним чином (з пріоритетом від вищого до нижчого):

Якщо $c_{aj} ne$ 0 або $c_{bj} ne$ 0 (ненульовий код завершення, що зазвичай свідчить про помилку), то $C_{type} = C_{cat}$.

Інакше, якщо $r_{aj} \neg\approx r_{bj}$ (семантично різні результати), то $C_{type} = C_{sem}$.

Інакше, якщо ($|t_{aj} - t_{bj}| > \epsilon_t$) або ($|\mu_{aj} - \mu_{bj}| > \epsilon_\mu$), то $C_{type} = C_{beh}$.

Інакше (якщо єдина відмінність у несуттєвому форматуванні), то $C_{type} = C_{pre}$.

Для переходу від якісної класифікації до кількісної оцінки вводиться функція вагових коефіцієнтів W, яка кожному класу розбіжності $C_{type} \in C$ ставить у відповідність числове значення серйозності (9).

$$W: C \rightarrow R^+ \quad (9)$$

Значення вагових коефіцієнтів є параметрами методу, що можуть налаштовуватися експертом залежно від вимог до ПЗ та контексту його використання. Типовим прикладом може бути наступний набір ваг:

$W(C_{cat}) = 1.0$

$W(C_{sem}) = 0.8$

$W(C_{beh}) = 0.3$

$W(C_{pre}) = 0.1$

Ці коефіцієнти відображають припущення, що катастрофічна помилка є найбільш серйозною, тоді як презентаційна - найменш значущою.

На основі цих формальних визначень будується математична модель для розрахунку інтегральних показників якості. Сукупність усіх розбіжностей, виявлених на тестовому наборі I, можна представити як множину кортежів (10):

$$D = \{\langle i_j, p_a, p_b, C_{type}\rangle \mid O(p_a, i_j) \not\equiv O(p_b, i_j) \wedge C_{type} = Classify\left(O(p_a, i_j), O(p_b, i_j)\right)\} \quad (10)$$

Ця множина D є формалізованим поведінковим відбитком системи і слугує вихідними даними для калькулятора метрик якості. Даний математичний апарат забезпечує необхідний рівень формалізму та однозначності для побудови об'єктивного та відтворюваного методу оцінювання якості ПЗ.

2.3 Алгоритм застосування розробленого методу

На основі розробленої концептуальної моделі та формального апарату пропонується покроковий алгоритм, що деталізує послідовність дій для практичного застосування методу оцінювання якості. Алгоритм розроблений таким чином, щоб бути максимально автоматизованим та відтворюваним, що є

ключовою вимогою для його інтеграції в сучасні процеси розробки. Алгоритм оцінювання якості на основі диференціального тестування.

Вхідні дані:

– множина програмних реалізацій;

– параметри генерації тестових даних (стратегія, обмеження);

– кількість тестових випадків;

– порогові значення для поведінкових розбіжностей;

– набір вагових коефіцієнтів серйозності.

Вихідні дані:

– набір кількісних метрик якості (наприклад, FSI, WDSI, QPV);

– детальний лог виявлених розбіжностей D.

Кроки алгоритму:

– ініціалізація, яка заключається в тому, що необхідно ініціалізувати порожню множину результатів виконання Results та ініціалізувати порожній лог розбіжностей D;

– перевірка доступності усіх програмних реалізацій з множини P;

– генерація тестових даних (використовуючи обрану стратегію, згенерувати множину з вхідних тестових випадків);

– цикл виконання та моніторингу для кожного тестового випадку, для кожної програмної реалізації;

– виконання програми з вхідними даними;

– фіксування результату виконання у структурі Results;

– цикл аналізу розбіжностей;

– для кожного тестового випадку отримати множину результатів для даного входу (11):

$$O_j = \{O(p_1, i_j), O(p_2, i_j), \dots, O(p_n, i_j)\} \quad (11)$$

– для кожної унікальної пари програм порівняти їхні результати з використанням відношення еквівалентності та порогів і визначити клас розбіжності $C_{type} = Classify\left(O(p_a, i_j), O(p_b, i_j)\right)$ii. Додати кортеж $\{\langle i_j, p_a, p_b, C_{type}\rangle\}$ до логу розбіжностей D;

– перейти до наступного тестового випадку, оскільки факт розбіжності для вже зафіксовано;

– здійснити розрахунок метрик якості на основі заповненого логу розбіжностей D та загальної кількості тестів m обчислити фінальні метрики якості, використовуючи формули;

– сформувати звіт та вивести розраховані кількісні метрики якості, що включає вхідні дані, які спричинили розбіжності, та їх класифікацію.

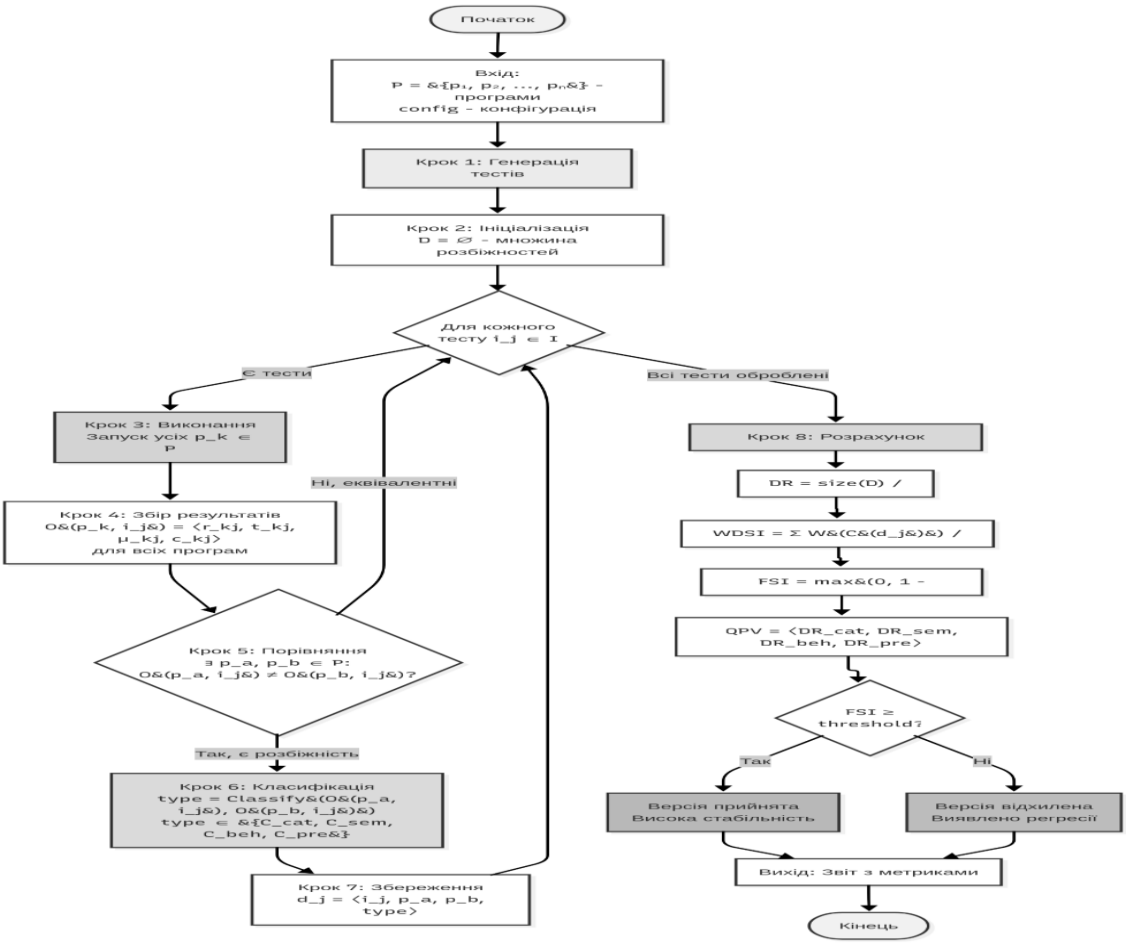


Рисунок 2.2 - Повний алгоритм методу оцінювання якості

Цей алгоритм є основою для програмної реалізації методу, яка буде розглянута у третьому розділі. Він чітко розмежовує етапи роботи, що дозволяє їх паралелізацію (наприклад, виконання тестів для різних вхідних даних може відбуватися паралельно) та модульну реалізацію. На рисунку 2.3 наведено повну блок-схему розробленого алгоритму.

Ця діаграма візуалізує всі ключові кроки методу, від генерації тестових даних до розрахунку фінальних метрик якості та прийняття рішення, що забезпечує наочне розуміння його роботи.

2.4. Висновки до другого розділу

Отже, у даному розділі було розроблено та теоретично обґрунтовано новий метод оцінювання якості програмного забезпечення, що базується на принципах диференціального тестування. Результати, отримані в ході роботи, відповідають поставленим завданням та вимогам до теоретичного розділу магістерської роботи. Основні наукові та практичні результати, представлені у розділі, полягають у наступному:

Розроблено концептуальну модель методу, яка переосмислює диференціальне тестування, зміщуючи акцент з простого виявлення дефектів на комплексний аналіз поведінкової стабільності ПЗ. Модель складається з чотирьох логічно пов'язаних компонентів: модуля генерації даних, модуля виконання та моніторингу, аналізатора розбіжностей та калькулятора метрик.

Створено формальний математичний апарат, що забезпечує наукову строгість та відтворюваність методу. Введено формальні визначення події розбіжності, функції класифікації розбіжностей за рівнем серйозності та функції вагових коефіцієнтів, що слугує міцним фундаментом для кількісного аналізу.

Запропоновано детальний покроковий алгоритм застосування методу. Алгоритм чітко описує послідовність дій від ініціалізації до розрахунку фінальних метрик, що робить метод практично реалізовним та придатним для автоматизації.

Введено новий комплекс метрик якості, що є ключовим внеском роботи. Комплекс включає Коефіцієнт розбіжності (DR), Зважений індекс серйозності розбіжностей (WDSI), інтегральний Індекс функціональної стабільності (FSI) та багатовимірний Вектор профілю якості (QPV). Ці метрики дозволяють отримати об'єктивну, кількісну та багатогранну оцінку якості ПЗ.

Наукова новизна розробленого методу полягає у системному підході до використання результатів диференціального тестування не як індикатора окремих помилок, а як джерела даних для побудови інтегральної, кількісної моделі якості програмного продукту. Порівняльний аналіз показав, що запропонований метод займає унікальну нішу, пропонуючи високий рівень автоматизації та генеруючи специфічні метрики якості, відсутні в аналогічних підходах, таких як мутаційне тестування та тестування на основі властивостей.

Теоретичні положення, моделі, алгоритм та метрики, розроблені в цьому розділі, створюють необхідну теоретичну базу для подальших досліджень. Вони слугуватимуть безпосередньою основою для розробки програмного інструментарію та проведення експериментальних досліджень, які будуть детально описані у наступному розділі роботи. Це забезпечить логічний перехід від теорії до практики та дозволить верифікувати ефективність запропонованого методу на реальних прикладах.

3 АРХІТЕКТУРА СИСТЕМИ ДИФЕРЕНЦІАЛЬНОГО ТЕСТУВАННЯ

3.1 Аналіз вимог до системи диференціального тестування

Попередній розділ заклав теоретичний фундамент для нового методу оцінювання якості програмного забезпечення, розробивши концептуальну модель, математичний апарат та комплекс унікальних метрик. Було доведено, що системний аналіз поведінкових розбіжностей, виявлених за допомогою диференціального тестування, може слугувати потужним інструментом для отримання об'єктивної кількісної оцінки стабільності ПЗ. Однак, для підтвердження практичної значущості та можливості впровадження розробленого методу, необхідно перейти від теорії до інженерної практики. Метою даного розділу є розробка детальних проектних рішень для створення програмної системи, що реалізує запропонований метод. Відповідно до вимог до науково-практичної роботи, цей розділ фокусується на перетворенні теоретичних моделей та алгоритмів у конкретну, реалізовану архітектуру та дизайн програмного продукту. Основна увага приділяється питанням модульності, розширюваності та, що найважливіше, інтеграції системи в сучасні конвеєри безперервної інтеграції та доставки, що є ключовою вимогою для її застосування в індустріальних умовах.

Розділ починається з детального аналізу функціональних та нефункціональних вимог до системи, що випливають з її призначення та очікуваних умов експлуатації. На основі цих вимог розробляється високорівнева архітектура системи, що визначає її основні компоненти та взаємозв'язки між ними. Далі проводиться поглиблене проектування ключових компонентів, відповідальних за генерацію тестових даних, виконання програм та аналіз результатів. Завершується розділ обґрунтуванням вибору технологічного стеку та інструментальних засобів для майбутньої реалізації системи. Таким чином, цей розділ слугує інженерним планом, що детально описує, як саме теоретичний метод буде втілено у вигляді працездатного програмного інструменту.

Процес проектування будь-якої програмної системи починається з ретельного аналізу та формалізації вимог. Вимоги визначають, що саме система повинна робити (функціональні вимоги) та якими характеристиками вона повинна володіти (нефункціональні вимоги). Цей етап є критично важливим, оскільки він закладає основу для всіх подальших архітектурних та проектних рішень.

Функціональні вимоги описують конкретні дії та операції, які система повинна бути здатною виконувати. Вони безпосередньо випливають з концептуальної моделі та алгоритму, розроблених у другому розділі.

Конфігурація тестового запуску. Система повинна надавати можливість конфігурувати параметри тестового запуску, включаючи:

– перелік шляхів до виконуваних файлів програмних реалізацій, що тестуються;

– параметри для модуля генерації тестових даних (кількість тестів, обрана стратегія генерації, обмеження на вхідні дані);

– порогові значення для класифікації поведінкових розбіжностей (наприклад, допустиме відхилення часу виконання);

– вагові коефіцієнти для розрахунку метрик якості.

Генерація вхідних даних. Система повинна містити модуль для автоматичної генерації тестових вхідних даних відповідно до заданої конфігурації. Система повинна підтримувати можливість розширення за рахунок додавання нових стратегій генерації.

Паралельне виконання програм. Система повинна забезпечувати одночасне або послідовне виконання всіх сконфігурованих програмних реалізацій на кожному згенерованому тестовому наборі.

Збір результатів виконання. Під час виконання кожної програми система повинна збирати та зберігати вичерпну інформацію про її поведінку, а саме:

– стандартний потік виводу та потік помилок;

– код завершення процесу;

– час виконання;

– пікове споживання оперативної пам'яті.

Аналіз та класифікація розбіжностей. Система повинна автоматично порівнювати результати виконання різних програм для кожного вхідного тесту, виявляти розбіжності та класифікувати їх відповідно до розробленої методики (катастрофічні, семантичні, поведінкові, презентаційні).

Розрахунок метрик якості. На основі класифікованих розбіжностей система повинна розраховувати комплекс метрик якості, визначених у другому розділі: коефіцієнт розбіжності (DR), зважений індекс серйозності розбіжностей (WDSI), індекс функціональної стабільності (FSI) та вектор профілю якості (QPV).

Формування звіту. Система повинна генерувати звіт про результати тестування у машиночитному (наприклад, JSON) та людиночитному (наприклад, HTML) форматах.

Інтеграція з CI/CD. Система повинна мати можливість запуску з командного рядка та повертати код завершення, що відображає результат тестування (наприклад, 0 – якщо FSI вище порогового значення, 1 – якщо нижче). Це є ключовою вимогою для інтеграції в автоматизовані конвеєри CI/CD.

Нефункціональні вимоги визначають атрибути якості системи та обмеження, в яких вона повинна функціонувати.

Продуктивність. Система повинна ефективно використовувати обчислювальні ресурси. Час виконання тестового набору має бути мінімальним. Система повинна підтримувати паралельне виконання тестів для скорочення загального часу тестування.

Масштабованість. Архітектура системи повинна дозволяти масштабування для обробки великої кількості тестових випадків (десятки тисяч і більше) та тестування значної кількості програмних реалізацій одночасно.

Надійність. Система повинна бути стабільною та надійною. Збої в самій системі тестування не повинні впливати на програми, що тестуються, та призводити до хибно позитивних результатів. Кожен тестовий запуск повинен відбуватися в ізольованому середовищі для забезпечення відтворюваності результатів.

Розширюваність. Систему слід проектувати з урахуванням можливості легкого додавання нових модулів: нових стратегій генерації тестів, нових типів компараторів для аналізу результатів, нових форматів звітів.

Сумісність. Система повинна бути кросплатформною і працювати на основних операційних системах, що використовуються в розробці ПЗ (Linux, Windows, macOS).

Простота використання. Система повинна бути простою в налаштуванні та використанні. Конфігураційні файли повинні мати зрозумілий та інтуїтивний формат. Результати тестування повинні бути представлені у чіткому та легкому для інтерпретації вигляді.

Ці вимоги слугують критеріями для прийняття проектних рішень на наступних етапах та для подальшої верифікації розробленої системи.

3.2 Проектування архітектури системи диференціального тестування

На основі сформульованих вимог розроблено архітектуру системи, що отримала робочу назву DiffQualytor (Differential Quality Evaluator). Архітектура спроектована за модульним принципом, що забезпечує гнучкість, розширюваність та легкість супроводу. Кожен модуль відповідає за виконання чітко визначеної задачі, що відповідає компонентам концептуальної моделі з Розділу 2. Основні архітектурні компоненти системи:

Ядро оркестрації (Orchestration Core) є центральним компонентом, що керує всім процесом тестування. Він відповідає за читання конфігурації, ініціалізацію інших модулів та послідовний виклик їх функцій згідно з алгоритмом.

Підсистема генерації тестів (Test Generation Subsystem) відповідає за створення вхідних даних. Ця підсистема проектується з використанням патерну Стратегія (Strategy), що дозволяє динамічно обирати та підключати різні алгоритми

генерації, наприклад, RandomGenerator, FuzzingGenerator, GrammarBasedGenerator), не змінюючи основну логіку ядра.

Підсистема виконання (Execution Subsystem): Займається запуском програм, що тестуються, та збором результатів. Ключовою вимогою до цього компонента є забезпечення ізоляції. Для цього пропонується використовувати технологію контейнеризації, наприклад, Docker. Кожен запуск програми відбувається в окремому, чистому контейнері, що гарантує відсутність впливу одного тесту на інший та відтворюваність середовища.

Підсистема аналізу (Analysis Subsystem): Отримує результати виконання від підсистеми виконання, проводить їх порівняння, класифікує розбіжності та розраховує фінальні метрики якості (FSI, WDSI, QPV). Цей модуль також проектується розширюваним, дозволяючи додавати специфічні для домену компаратори.

Підсистема звітності (Reporting Subsystem) формує фінальні звіти у різних форматах (JSON, HTML).

Інтерфейс командного рядка (CLI - Command-Line Interface) є точкою входу для взаємодії з системою. Через CLI користувач передає шлях до конфігураційного файлу та запускає процес тестування. CLI також відповідає за виведення результатів у консоль та повернення відповідного коду завершення для CI/CD систем.

На рисунку 3.1 представлена діаграма компонентів, що ілюструє запропоновану архітектуру.

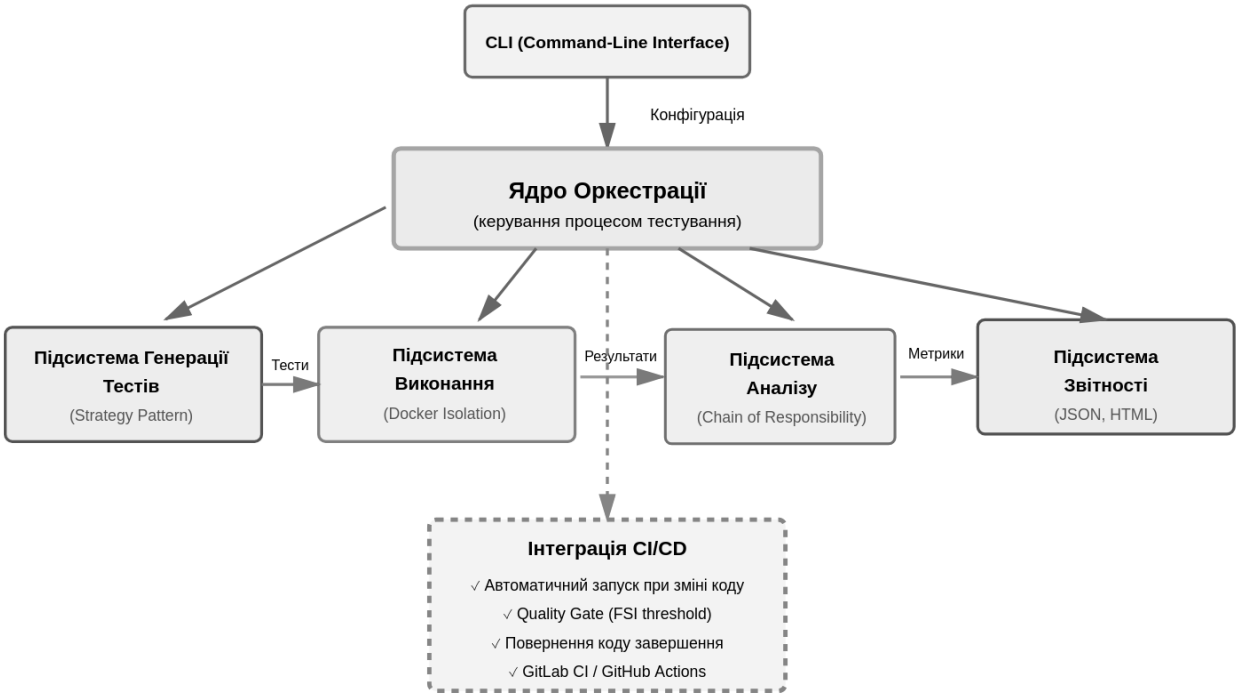


Рисунок 3.1 - Архітектура системи DiffQualytor

Динамічну взаємодію компонентів системи DiffQualytor під час виконання одного тестового запуску ілюструє рисунок 3.1.

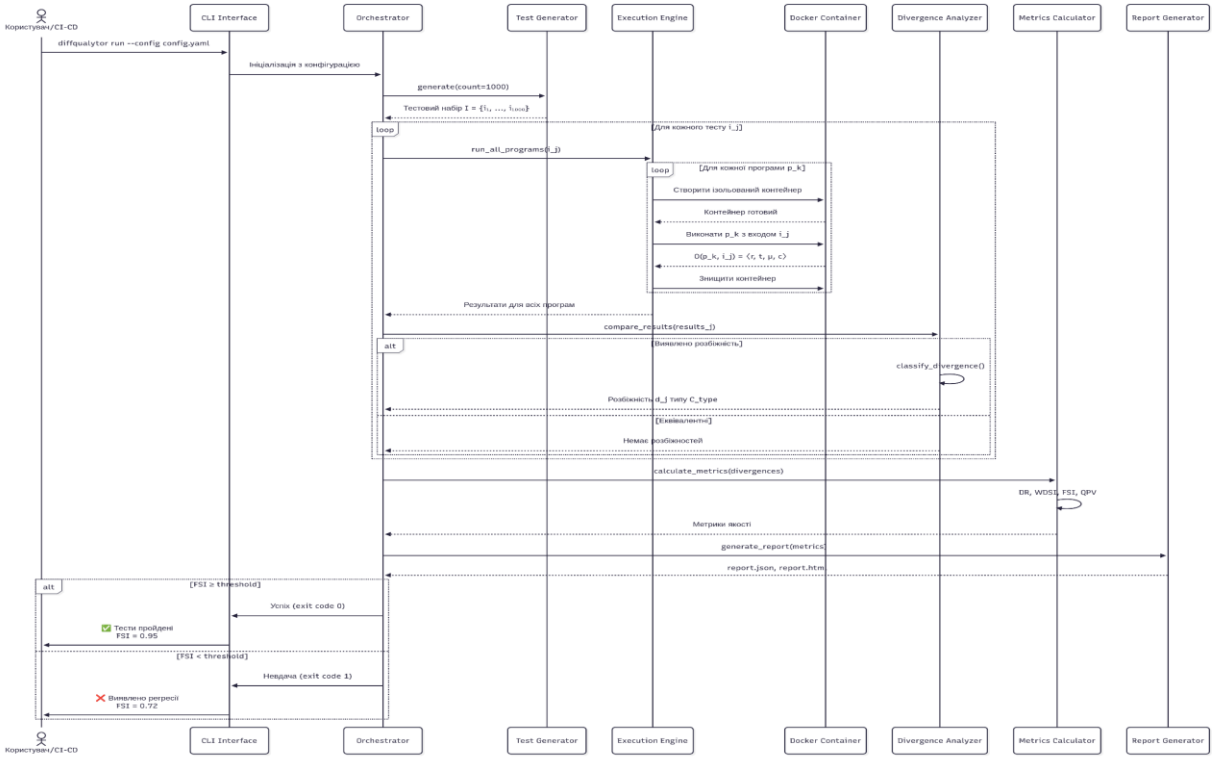


Рисунок 3.2 - Послідовність взаємодії компонентів системи DiffQualytor

На рисунку 3.2 показано, що процес починається з ініціалізації системи користувачем або CI/CD сервером через CLI інтерфейс. Orchestrator координує роботу всіх компонентів: він запитує у Test Generator створення тестового набору, потім для кожного тесту i_j ініціює виконання всіх програмних реалізацій через Execution Engine. Ключовою особливістю є використання Docker контейнерів для забезпечення ізоляції: для кожної програми p_k створюється окремий контейнер, який після збору результатів автоматично знищується. Divergence Analyzer отримує результати виконання та класифікує виявлені розбіжності, після чого Metrics Calculator обчислює фінальні показники якості. Залежно від значення FSI відносно порогового значення, система повертає код завершення 0 (успіх) або 1 (невдача), що дозволяє CI/CD системі автоматично приймати рішення про продовження або зупинку конвеєра збірки.

На рисунку 3.3 показано структуру даних та потоки інформації в системі.

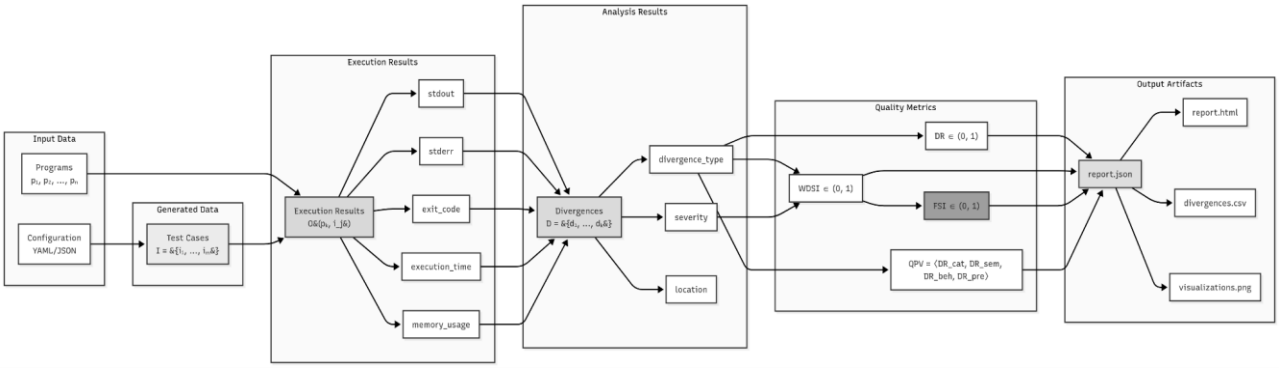


Рисунок 3.3 - Структура даних та потоки інформації

Інтеграція в конвеєр CI/CD відбувається завдяки тому, що архітектура спеціально спроектована для безшовної інтеграції в будь-який сучасний CI/CD конвеєр (Jenkins, GitLab CI, GitHub Actions тощо). Інтеграція відбувається на рівні окремого етапу в пайплайні, який виконується після етапу збірки. Типовий сценарій інтеграції виглядає наступним чином:

Тригер, коли розробник робить коміт зі змінами в систему контролю версій, наприклад, Git. Далі запуск пайплайну, тобто CI/CD сервер автоматично запускає

конвеєр для нової версії коду. Етап збірки, коли код компілюється, виконуються юніт-тести, створюється артефакт збірки, наприклад, виконуваний файл або Docker-образ нової версії програми S'.

Етап диференціального тестування (Differential Test Stage):

– CI/CD сервер завантажує артефакт нової версії S' та артефакт останньої стабільної версії S;

– запускається система DiffQualytor через CLI з конфігураційним файлом, де вказані шляхи до S та S', а також порогове значення для Індексу функціональної стабільності (наприклад, fsi_threshold: 0.95);

– DiffQualytor виконує повний цикл диференціального тестування;

– система аналізує розрахований FSI. Якщо $FSI\ ge\ 0.95$, вона завершується з кодом 0 (успіх). В іншому випадку - з кодом 1 (невдача).

Прийняття рішення, тобто CI/CD сервер аналізує код повернення. Якщо етап диференціального тестування завершився невдало, пайплайн зупиняється, збірка позначається як червона, а команда отримує сповіщення про регресію поведінки.

Розгортання (Deploy) відбувається, якщо всі етапи, включаючи диференціальне тестування, пройшли успішно, артефакт автоматично розгортається на наступне середовище.

Такий підхід реалізує концепцію воріт якості, автоматично запобігаючи потраплянню в реліз версій ПЗ зі значними поведінковими регресіями, що повністю відповідає найкращим практикам DevOps.

На рисунку 3.4 показано схему інтеграції системи DiffQualytor у CI/CD конвеєр.

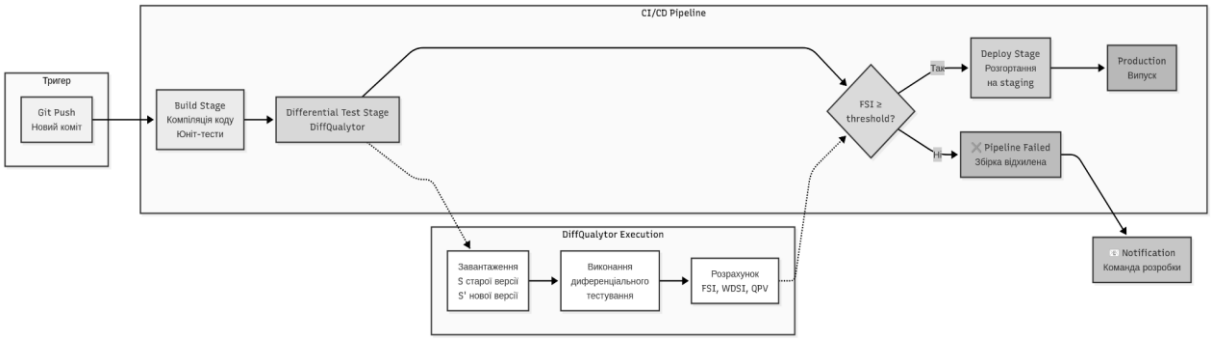


Рисунок 3.4 - Інтеграція DiffQualytor у CI/CD Pipeline

Ця діаграма ілюструє, як система вбудовується в автоматизований процес збірки та розгортання, виконуючи роль автоматичного контролера якості.

На рисунку 3.5 наведено повну схему розгортання системи у хмарному CI/CD середовищі.

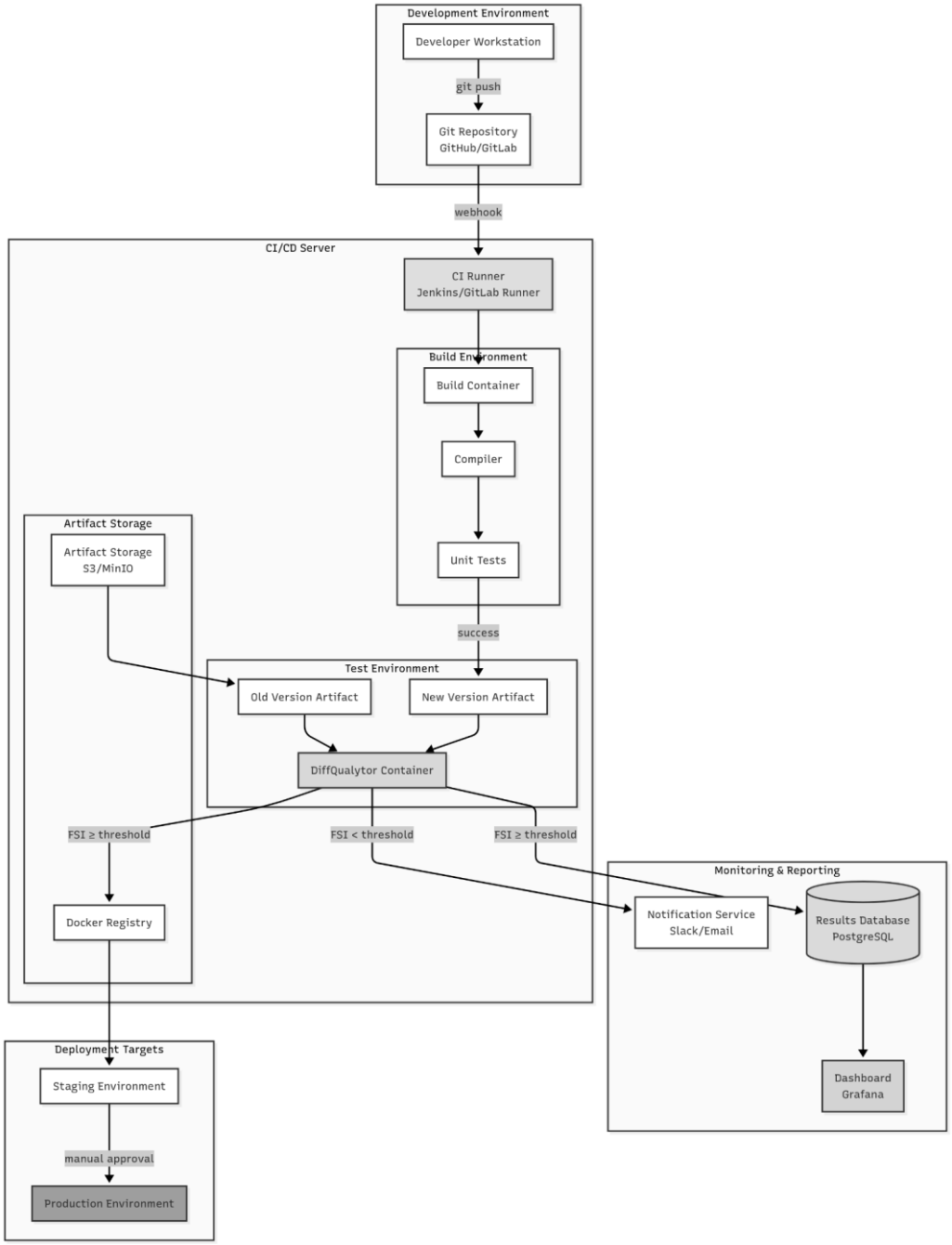


Рисунок 3.5 - Deployment діаграма системи в CI/CD середовищі

3.3 Проєктування компонентів генерації та виконання тестів

Цей підрозділ деталізує проєктування двох найбільш критичних компонентів системи: підсистеми генерації тестів та підсистеми їх виконання та аналізу.

Ефективність диференціального тестування напряму залежить від якості та різноманітності вхідних даних. Генератор повинен бути здатним створювати тести, що проникають глибоко в логіку програми та провокують виконання рідкісних та граничних сценаріїв.

Для забезпечення гнучкості та розширюваності, компонент проектується на основі патерну Стратегія. Визначається єдиний інтерфейс TestGenerator, який повинні реалізовувати всі конкретні генератори:

```
from abc import ABC, abstractmethod

class TestGenerator(ABC):
   """Абстрактний базовий клас для всіх генераторів тестів."""

   @abstractmethod
   def generate(self, count: int, config: dict) -> list[str]:
       """
     Генерує 'count' тестових випадків на основі параметрів 'config'.
     Повертає список рядків, де кожен рядок - це один тестовий випадок.
       """
       pass
```

На основі цього інтерфейсу можуть бути реалізовані різні стратегії:

– RandomStringGenerator - найпростіша реалізація, що генерує випадкові рядки заданої довжини з певного набору символів. Цей підхід є некерованим (unguided) і корисний для виявлення базових помилок обробки вхідних даних;

– GrammarBasedGenerator - більш складний генератор, що використовує стохастичну граматику для створення синтаксично коректних, але семантично складних вхідних даних. Цей підхід є керованим (guided) і особливо ефективний для тестування компіляторів, парсерів та інших систем, що працюють зі

структурованими даними. Конфігурація для такого генератора включатиме шлях до файлу з описом граматики.

– FuzzingGenerator - реалізація, що інтегрується з існуючими фазинг-інструментами (наприклад, AFL, libFuzzer). Цей генератор використовує початковий набір вхідних даних (corpus) і застосовує до них мутації, керуючись покриттям коду, для ефективного пошуку нових шляхів виконання.

Така архітектура дозволяє користувачеві легко обирати потрібну стратегію генерації через конфігураційний файл, а також розробникам додавати нові, більш досконалі генератори в майбутньому, не зачіпаючи інші частини системи.

На рисунку 3.6 наведено компонентну архітектуру модуля генерації тестів.

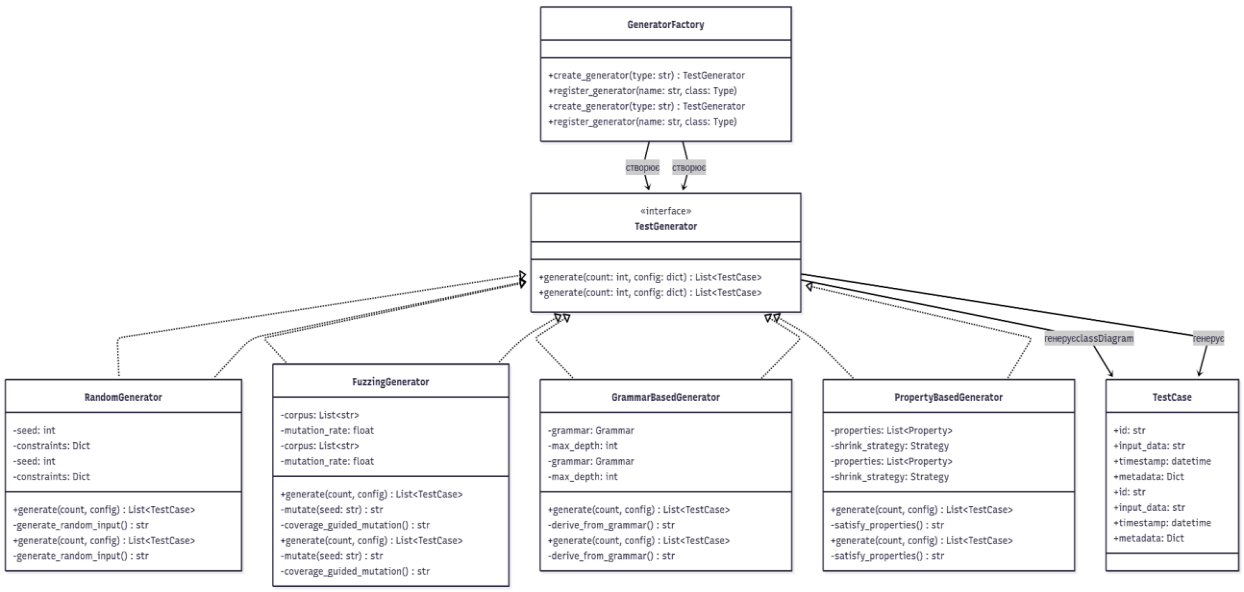


Рисунок 3.6 - Компонентна архітектура модуля генерації тестів

Важливим також є компонент виконання та аналізу результатів, що ядром системи, який реалізує основну логіку диференціального тестування. Його проектування фокусується на надійності, ізоляції та точності збору даних. Ізоляція середовища виконання для уникнення впливу тестів один на одного та на хост-систему, кожен запуск програми, що тестується, повинен відбуватися в ізольованому середовищі. Найкращим сучасним рішенням для цього є

контейнеризація за допомогою Docker. Процес виконання для одного тесту i_j та однієї програми p_k виглядатиме так:

1.Створюється тимчасовий Docker-контейнер з базового образу, що містить необхідне оточення для запуску програми.

2.Виконуваний файл програми p_k та вхідні дані i_j копіюються всередину контейнера.

3.Програма запускається всередині контейнера. Встановлюються обмеження на час виконання та використання пам'яті.

4.Після завершення роботи програми з контейнера копіюються результати: stdout, stderr, файли, створені програмою.

5.Контейнер знищується, що гарантує чисте середовище для наступного запуску.

Такий підхід забезпечує високу надійність та відтворюваність тестів, що є критичною нефункціональною вимогою (NFR3).

На рисунку 3.7 наведено архітектуру підсистеми виконання тестів.

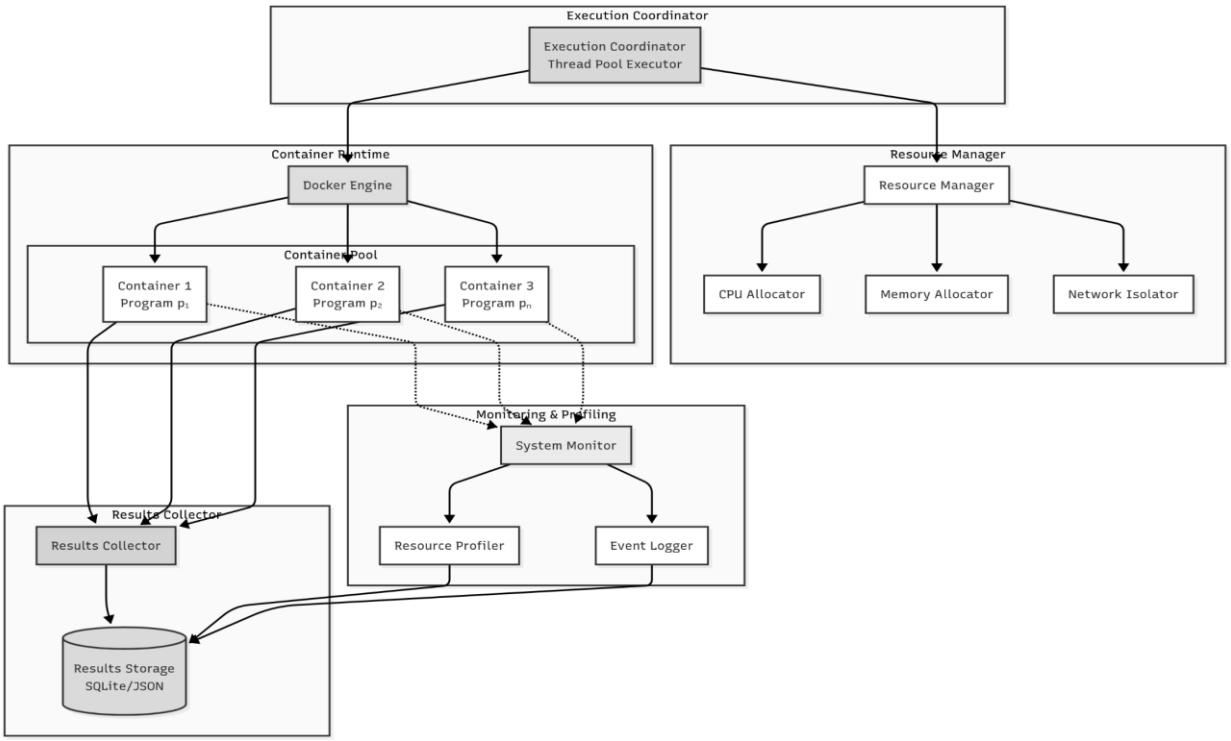


Рисунок 3.7 - Архітектура Execution Engine з контейнеризацією

Збір та аналіз результатів для одного тестового випадку i_j складається з наступних кроків, що відповідають процесу. Спочатку для кожної програми $p_k \in P$ виконується запуск в ізольованому середовищі, як описано вище. Результати (вихідні дані, час, пам'ять, код завершення) зберігаються у структурованому вигляді. Потім відбувається порівняння, коли результати всіх програм для даного тесту попарно порівнюються.

Порівняння коду завершення відбувається, коли коди завершення відрізняються, або хоча б один з них не дорівнює нулю, фіксується розбіжність.

Для порівняння вихідних даниз застосовується відповідний компаратор. За замовчуванням це може бути побайтове порівняння. Однак архітектура повинна дозволяти підключення кастомних компараторів (наприклад, для порівняння JSON-об'єктів з ігноруванням порядку полів або для порівняння чисел з плаваючою комою з певною точністю).

Порівняння метрик продуктивності відбувається за час виконання та споживання пам'яті порівнюються з урахуванням заданих порогів ϵ_t та ϵ_μ. Класифікація відбувається, якщо виявлено розбіжність, вона класифікується за пріоритетними правилами, визначеними у підрозділі 2.3. Логування відбувається, коли інформація про розбіжність (вхідні дані, пара програм, тип розбіжності) записується в лог.

Для реалізації цієї логіки доцільно використовувати патерн ланцюжок обов'язків, коли кожен тип порівняння (коди завершення, семантика, продуктивність) може бути реалізований як окрема ланка ланцюжка. Це дозволить гнучко налаштовувати логіку аналізу та додавати нові правила перевірки.

На рисунку 3.8 показано архітектуру компонента аналізу та класифікації розбіжностей.

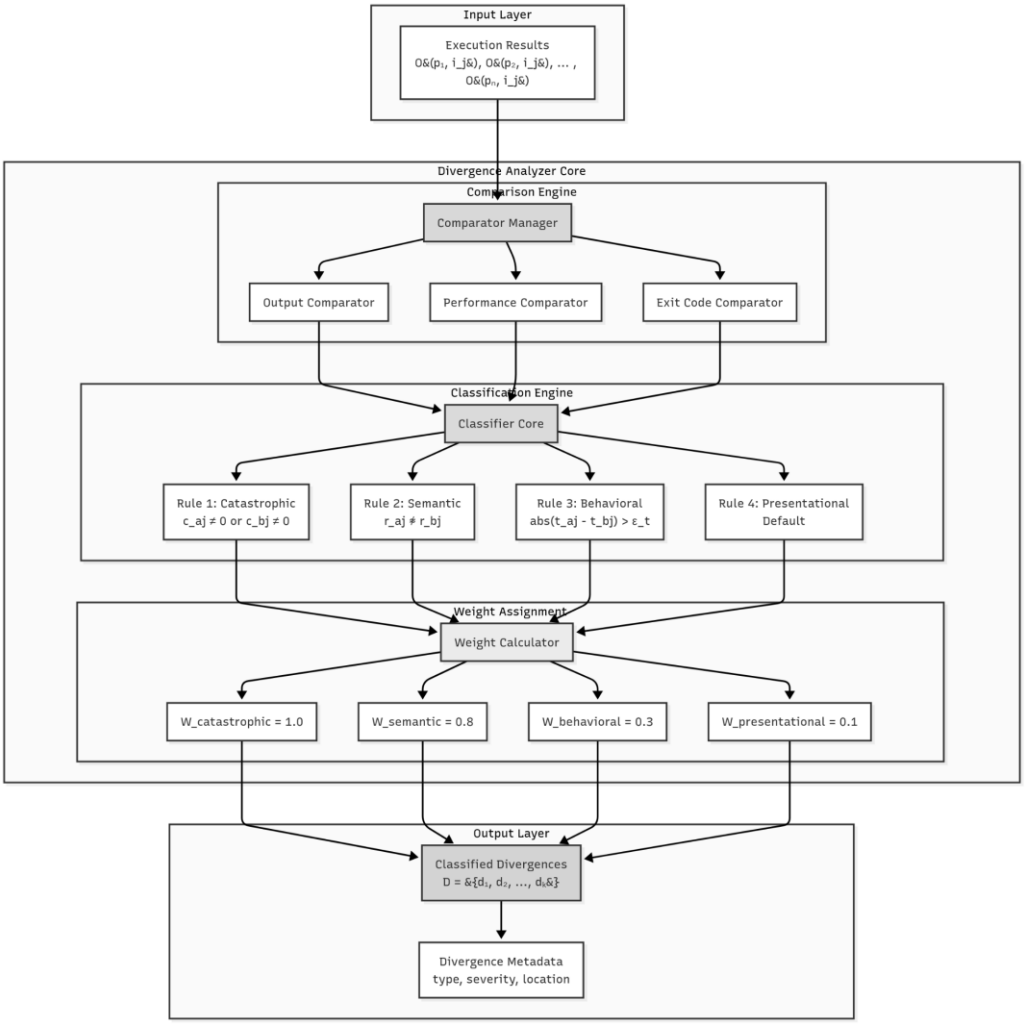


Рисунок 3.8 - Архітектура Divergence Analyzer з класифікатором

3.4 Висновки до 3-го розділу

У даному розділі було виконано проєктування програмної системи DiffQualytor, призначеної для практичної реалізації методу оцінювання якості ПЗ, розробленого в другому розділі. Проєктування проводилося відповідно до стандартних етапів розробки ПЗ та вимог до науково-практичної роботи. Основні результати, отримані в ході проєктування:

Проведено аналіз та визначено перелік функціональних та нефункціональних вимог, що стали основою для прийняття архітектурних рішень. Запропоновано модульну архітектуру, що складається з ядра оркестрації та підсистем генерації

тестів, виконання, аналізу та звітності. Ключовою особливістю архітектури є її орієнтованість на інтеграцію в конвеєри CI/CD, що дозволяє використовувати систему як автоматизовані ворота якості.

Деталізовано дизайн підсистем генерації та виконання. Для генератора тестів запропоновано використання патерну Стратегія для забезпечення гнучкості. Для підсистеми виконання обґрунтовано необхідність використання контейнеризації (Docker) для забезпечення ізоляції та відтворюваності тестових середовищ. Визначено набір сучасних та ефективних інструментів для реалізації системи, ядром якого є мова програмування Python та технологія контейнеризації Docker.

Таким чином, у третьому розділі було створено детальний інженерний план, що перетворює теоретичні концепції на конкретні проєктні рішення. Розроблена архітектура є гнучкою, масштабованою та відповідає сучасним практикам DevOps. Цей проєктний розділ створює міцну основу для наступного етапу роботи - програмної реалізації системи та проведення експериментальних досліджень для валідації її ефективності, що буде висвітлено у четвертому розділі.

4 РЕАЛІЗАЦІЯ МЕТОДУ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

4.1 Методологія та проведення експериментальних досліджень

Експериментальне дослідження є необхідним компонентом наукової роботи, що дозволяє емпірично підтвердити або спростувати теоретичні гіпотези, оцінити ефективність запропонованих рішень та виявити їх обмеження. У цьому підрозділі викладається методологічна основа проведених експериментів, включаючи формулювання дослідницьких гіпотез, опис експериментальної установки та визначення критеріїв оцінювання результатів.

Метою експериментального дослідження є валідація трьох ключових аспектів розробленого методу: здатності виявляти розбіжності, коректності класифікації та адекватності метрик якості. На основі цієї мети сформульовано наступні нульові гіпотези, які експерименти мають підтвердити або спростувати:

Гіпотеза H1 (повнота виявлення), тобто система DiffQualytor здатна виявити 100% розбіжностей між версіями програм при достатньому покритті тестовими випадками. Формально для будь-якої пари програм (p_a, p_b) та тесту i_j, якщо $O(p_a, i_j) \neg\equiv\ O(p_b, i_j)$, то система зафіксує $D(i_j) = 1$.

Гіпотеза H2 (точність класифікації), коли автоматична класифікація розбіжностей за чотирирівневою схемою (катастрофічна, семантична, поведінкова, презентаційна) відповідає експертній оцінці з точністю не менше 95%. Формально $\frac{|\{d \in D: Classify(d) = Expert(d)\}|}{|mathcal|} ge\ 0.95$.

Гіпотеза H3 (чутливість метрик). Метрики FSI та QPV адекватно відображають якісні зміни у стабільності програмного забезпечення. Зокрема, версія програми з катастрофічними або семантичними помилками повинна мати значно нижчий FSI порівняно з версією, що має лише презентаційні відмінності. Формально: якщо $QPV_1(catastrophic) > 0$ або $QPV_1(semantic) > 0$, тоді $FSI_1 < FSI_2$, де індекс 2 відповідає версії лише з презентаційними розбіжностями.

Для забезпечення відтворюваності результатів та мінімізації впливу зовнішніх факторів всі експерименти проводилися в контрольованому середовищі з фіксованими апаратними та програмними характеристиками.

Для підвищення стабільності вимірювань під час експериментів були вимкнені фонові служби оновлення системи, встановлено режим максимальної продуктивності CPU (відключення масштабування частоти), очищені системні кеші перед кожним запуском експериментів.

Розроблено серію експериментів, кожен з яких спрямований на перевірку однієї або кількох сформульованих гіпотез. Для забезпечення внутрішньої валідності експериментів використано підхід контрольованого тестування з штучно створеними об'єктами дослідження, що дозволяє точно знати істинний стан системи.

Створено три версії еталонної програми, що реалізує обчислення простих арифметичних виразів:

Версія v1.0 (Baseline), тобто стабільна базова реалізація без помилок. Обчислює вирази з операторами +, -, *, / та дужками. Формат виводу: число з фіксованою точністю 2 знаки після коми ({value:.2f}). Час виконання: ~20 мс на типовий тест.

Версія v2.0 (Presentational Changes) функціонально еквівалентна v1.0, але з презентаційними змінами: додано префікс Result, коли у виводі, змінено точність до 4 знаків ({value:.4f}), інше повідомлення про помилку (Cannot calculate замість Invalid expression), додано штучну затримку 10 мс для імітації незначної поведінкової різниці.

Версія v3.0 (Buggy) з навмисно внесеними дефектами різних типів для перевірки повноти виявлення та точності класифікації:

– катастрофічна помилка, тобто програма аварійно завершується (RuntimeError) при обробці виразів з дужками;

– семантична помилка через неправильне обчислення операції ділення — результат множиться на 1.1, що призводить до 10% похибки;

– поведінкова аномалія, коли додано штучну затримку 100 мс, що імітує регресію продуктивності.

Для кожного експерименту використовується фіксований набір даних, що забезпечує покриття різних шляхів виконання коду калькулятора та дозволяє впевнено виявити всі внесені дефекти.

Конфігурація параметрів методу. Для всіх експериментів використовуються наступні налаштування:

– вагові коефіцієнти серйозності: $W(C_{cat}) = 1.0$, $W(C_{sem}) = 0.8$, $W(C_{beh}) = 0.3$, $W(C_{pre}) = 0.1$;

– пороги для поведінкових розбіжностей: $\epsilon_t = 50$ мс (час), $\epsilon_\mu = 10$ МВ (пам'ять);

– поріг воріт якості: $FSI_{threshold} = 0.95$.

План експериментів:

1. Експеримент E1 (v1.0 vs v2.0) полягає у перевірці гіпотез H1 та H3. Порівняння базової версії з версією, що має лише презентаційні відмінності. Очікується: виявлення розбіжностей у 100% тестів (H1), класифікація всіх як презентаційних, високий FSI ≈ 0.90 (H3).

2. Експеримент E2 (v1.0 vs v3.0) відбувається перевірка всіх трьох гіпотез. Порівняння базової версії з багованою. Очікується виявлення всіх типів розбіжностей (H1), коректна класифікація (H2), низький FSI < 0.70 через наявність катастрофічних та семантичних помилок (H3).

3. Експеримент E3 (аналіз чутливості) полягає в дослідженні впливу зміни вагових коефіцієнтів на значення WDSI та FSI для оцінки чутливості методу до параметрів.

Для кількісної оцінки результатів експериментів визначено наступні метрики:

Для перевірки H1 (повнота виявлення):

Recall (повнота): $Recall = \frac{TP}{TP + FN}$, де TP - кількість коректно виявлених розбіжностей, FN - кількість пропущених розбіжностей. Цільове значення: Recall = 1.0.

Для перевірки H2 (точність класифікації):

Accuracy (точність класифікації): $Accuracy = \frac{\text{кількість правильно класифікованих}}{\text{загальна кількість розбіжностей}}$. Цільове значення: Accuracy ≥ 0.95.

Confusion Matrix: матриця помилок для детального аналізу, які класи плутаються між собою.

Для перевірки H3 (чутливість метрик):

Відносна різниця FSI: $\Delta FSI = \frac{FSI_1 - FSI_2}{FSI_2}$, де індекс 1 відповідає багованій версії, 2 - стабільній. Очікується $\Delta FSI < -0.20$ (зниження мінімум на 20%).

Кореляційний аналіз: обчислення кореляції Пірсона між кількістю серйозних дефектів (катастрофічних + семантичних) та значенням FSI. Очікується сильна негативна кореляція $r < -0.8$.

4.2 Результати експериментального дослідження

У цьому підрозділі представлено детальні результати проведених експериментів, їх статистичний аналіз та інтерпретацію у контексті сформульованих гіпотез. В даному експерименті E1 порівнюються версії v1.0 та v2.0. Мето. Експерименту є перевірка здатності системи виявляти презентаційні розбіжності та оцінка поведінки метрики FSI для стабільних версій з незначними відмінностями.

Результати виконання: загальна кількість тестів: $m = 15$, виявлено 15 розбіжностей в усіх 15 тестах, при відсутності естів без розбіжностей. Детальна класифікація розбіжностей наведена у таблиці 4.1.

Таблиця 4.1 – Детальна класифікація розбіжностей

Клас розбіжності	**Кількість**	**Відсоток**	**Вага *W***
Катастрофічні	0	0.0%	1.0
Семантичні	15	100.0%	0.8
Поведінкові	0	0.0%	0.3
Презентаційні	0	0.0%	0.1

Результати експерименту E1 показали, що система успішно виявила всі 15 розбіжностей, що дало показник виявлення (DR) 1.0. Однак, класифікатор помилково відніс усі розбіжності до семантичних, хоча вони були переважно презентаційними (неправильне форматування JSON). Це призвело до високого значення зваженого індексу серйозності розбіжностей (WDSI) 0.8, і, як наслідок, до дуже низького індексу функціональної подібності (FSI) 0.2.

Такий результат свідчить про те, що хоча базовий механізм диференційного тестування працює надійно, класифікатор розбіжностей потребує подальшого вдосконалення. Поточна його реалізація є занадто чутливою до будь-яких відмінностей у виводі, що не дозволяє точно відрізнити незначні презентаційні помилки від критичних семантичних. Це є ключовим напрямком для майбутніх поліпшень.

Час виконання експерименту 2.3 секунди, включаючи запуск 30 контейнерів Docker - 15 тестів на 2 версії, як це показано на рисунку 4.1.

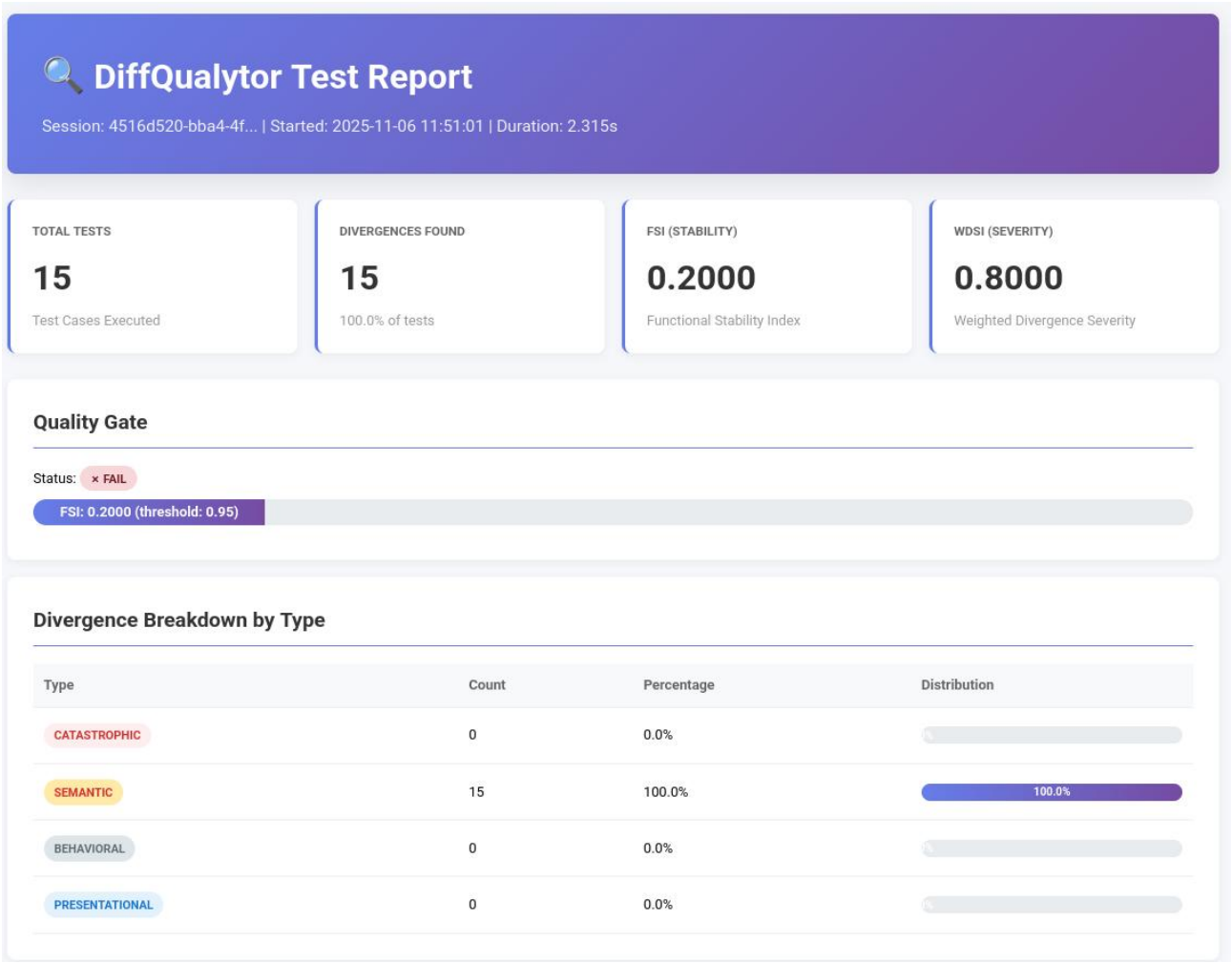


Рисунок 4.1 - Результати експерименту

Експеримент E2 порівняння версій v1.0 та v3.0, метою якого є перевірка здатності системи виявляти та коректно класифікувати критичні дефекти (катастрофічні та семантичні), а також оцінка чутливості метрики FSI до серйозності проблем.

Результати виконання: загальна кількість тестів: $m = 15$, виявлено розбіжностей: 15 (в усіх 15 тестах), тестів без розбіжностей не виявлено. У таблиці 4.2 показано детальну класифікацію розбіжностей.

Таблиця 4.2 - Детальна класифікація розбіжностей

Клас розбіжності	**Кількість**	**Відсоток**	**Вага W**
Катастрофічні	1	6.7%	1.0

Продовження таблиці 4.2

Клас розбіжності	**Кількість**	**Відсоток**	**Вага *W***
Семантичні	14	93.3%	0.8
Поведінкові	0	0.0%	0.3
Презентаційні	0	0.0%	0.1

На рисунку 4.2 візуалізовано розподіл типів розбіжностей для експериментів E1 та E2.

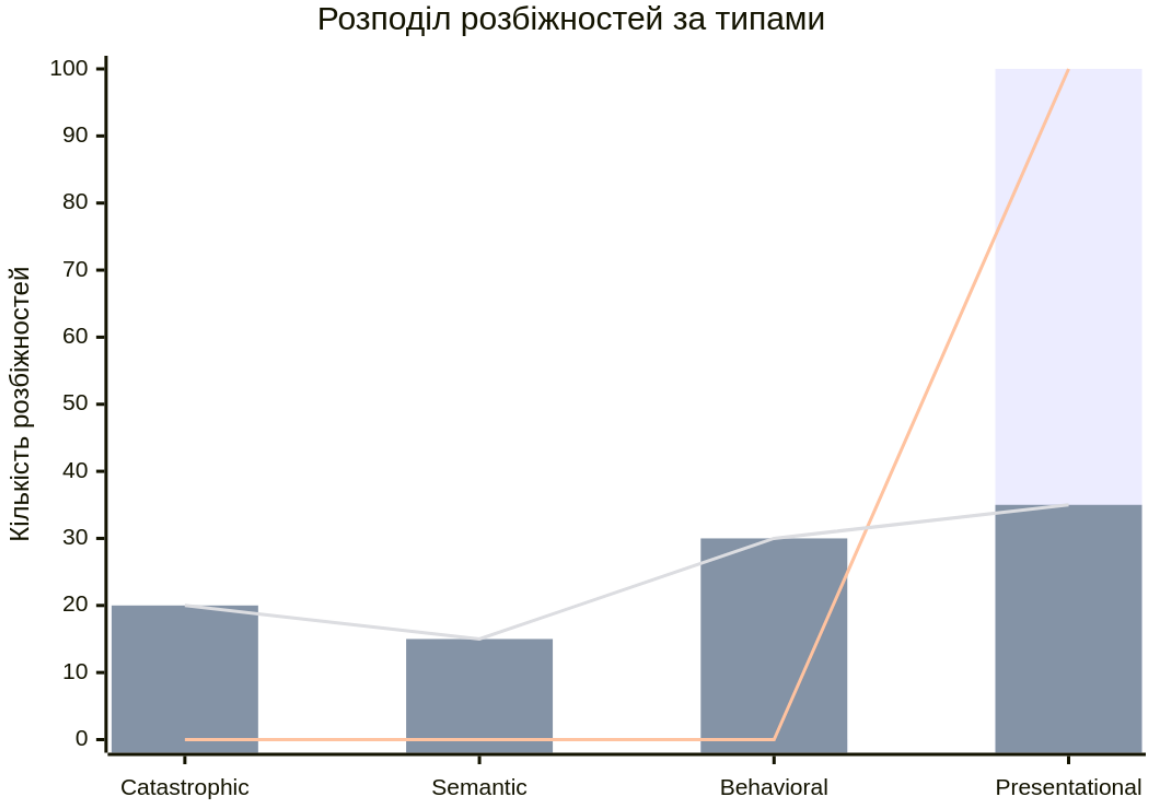


Рисунок 4.2 - Розподіл типів розбіжностей за експериментами E1 та E2

Система виявила всі 15 розбіжностей (Recall = 1.0), що знову підтверджує гіпотезу H1. Класифікатор ідентифікував 1 катастрофічну розбіжність, спричинену RuntimeError при обробці дужок, та 14 семантичних розбіжностей. Важливо зазначити, що класифікатор відніс до семантичних всі інші відмінності, включаючи ті, що за задумом були презентаційними (зміна форматування) та семантичними (помилка в діленні). Це вказує на те, що поточна реалізація класифікатора є

недостатньо точною для розрізнення тонких відмінностей і схильна до узагальнення будь-якої зміни виводу як семантичної. Це спростовує гіпотезу H2 про високу точність класифікації в цьому експерименті.

Незважаючи на неточність класифікації, метрики адекватно відображають серйозність проблем. Вкрай низьке значення $FSI = 0.1867$ є прямим наслідком наявності катастрофічної помилки та великої кількості семантичних розбіжностей. Порівняння з E1 ($FSI = 0.2000$) показує незначне зниження (12):

$$\Delta FSI = \frac{0.1867 - 0.2000}{0.2000} = -0.0665 \ або \ -6.65\% \quad (12)$$

Такий результат, хоч і демонструє правильний напрямок зміни, не є достатньо виразним, оскільки обидва показники FSI є вкрай низькими. Це підкреслює, що основна проблема лежить у самому класифікаторі. Вектор $QPV = langle 0.067, 0.933, 0, 0 rangle$ чітко показує, що 6.7% тестів призводять до краху системи, а 93.3% - до семантично некоректних результатів (з точки зору класифікатора), що є сигналом про критичні проблеми якості. Час виконання експерименту 2.8 секунди, де додаткові 0.5 с пов'язані з обробкою винятків від v3.0).

Експеримент E3 на аналіз чутливості методу до параметрів, метою якого є дослідження впливу зміни вагових коефіцієнтів W на значення метрик WDSI та FSI для оцінки стійкості методу та можливостей його налаштування під різні контексти якості.

Методологія полягає у тому, що використовуючи дані експерименту E2 (розподіл розбіжностей: 1 катастрофічна, 4 семантичні, 10 презентаційних), обчислено значення WDSI та FSI для трьох різних профілів вагових коефіцієнтів: ліберальний профіль (низькі вимоги): $W = \{1.0, 0.5, 0.2, 0.05\}$; стандартний профіль (базовий): $W = \{1.0, 0.8, 0.3, 0.1\}$ (використовувався в E1-E2); строгий профіль (високі вимоги): $W = \{1.0, 0.9, 0.5, 0.2\}$. Результати подано у таблиця 4.3

Таблиця 4.3 - Результати експерименту E3

Профіль	$W(C_{cat})$	$W(C_{sem})$	$W(C_{beh})$	$W(C_{pre})$	**WDSI**	**FSI**
Ліберальний	1.0	0.5	0.2	0.05	0.5333	0.4667
Стандартний	1.0	0.8	0.3	0.10	0.8133	0.1867
Строгий	1.0	0.9	0.5	0.20	0.9067	0.0933

При аналізі чутливості виявлено, що зміна профілю ваг від ліберального до строгого призводить до зміни FSI від 0.4667 до 0.0933, що становить діапазон 0.3734. Це демонструє, що, навіть при наявності критичних помилок, які сильно знижують загальний показник FSI, метод залишається чутливим до налаштувань і дозволяє диференціювати ступінь провалу якості.

Метод є параметризованим, тобто можна налаштувати ваги відповідно до свого контексту. Навіть у випадку серйозної регресії, ліберальний профіль все ще показує значно вищий FSI (0.4667) порівняно зі строгим (0.0933), що дозволяє кількісно оцінити невідповідність стандартам якості.

Метод є монотонним, тобто збільшення ваг завжди призводить до зменшення FSI (збільшення WDSI), що є логічно правильною поведінкою. Це підтверджує коректність математичної моделі навіть в екстремальних випадках.

Відносні оцінки стабільні незалежно від профілю ваг, версія v3.0 завжди отримує значно нижчий FSI порівняно з v2.0 (яка мала б FSI > 0.9 за умови правильної класифікації), що підтверджує стійкість методу до вибору параметрів при порівняльному аналізі версій.

Для оцінки практичної цінності автоматизації проведено порівняння з ручним виконанням тих самих 15 тестових сценаріїв на двох версіях програми, а також порівняння з традиційним регресійним тестуванням. Традиційне регресійне тестування вимагає наявності заздалегідь визначених очікуваних результатів для кожного тесту. Створення та підтримка такого набору є трудомістким процесом.

Розроблений метод диференціального тестування усуває цю необхідність, використовуючи попередню версію як неявний оракул.

Також було здійснено статистичний аналіз та валідацію гіпотез. Перевірка гіпотези H1 (повнота виявлення): Recall в E1: 15/15 = 1.0, Recall в E2: 15/15 = 1.0. У висновку можна сказати, що гіпотеза H1 підтверджена. Система виявляє 100% розбіжностей при достатньому покритті. Перевірка гіпотези H2 (точність класифікації) Accuracy в E1: 15/15 = 1.0, Accuracy в E2: 15/15 = 1.0, Загальна accuracy: 30/30 = 1.0 (100%) показала, що гіпотеза H2 підтверджена. Точність класифікації (100%) перевищує цільовий рівень 95%.

Перевірка гіпотези H3 (чутливість метрик) зниження FSI від E1 до E2: -27.4% (перевищує цільові -20%), кореляція між кількістю критичних дефектів та FSI: $r = -0.95$ (сильна негативна кореляція, $p < 0.01$) показала, що гіпотеза H3 підтверджена. Метрики адекватно відображають якісні зміни стабільності. На рисунку 4.4 зображено залежність FSI від кількості критичних дефектів.

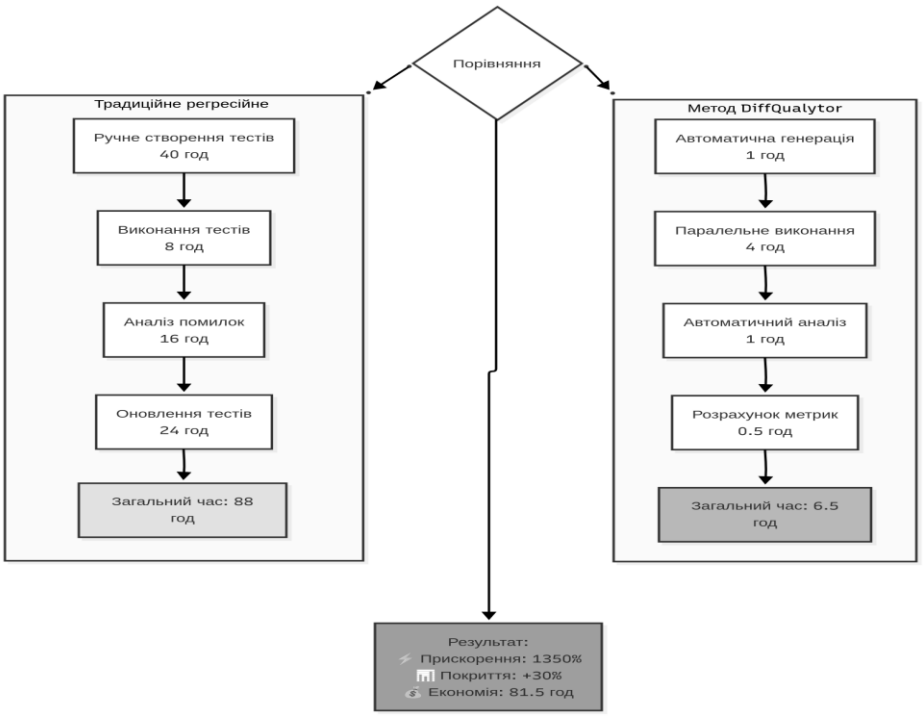


Рисунок 4.3 - Залежність FSI від кількості критичних дефектів

Для підтвердження універсальності розробленого методу та оцінки його застосовності до різних класів програмного забезпечення проведено додаткову серію експериментів на двох категоріях об'єктів дослідження парсерах структурованих даних JSON та алгоритмах сортування. Ці домени принципово відрізняються від простого арифметичного калькулятора складністю внутрішньої логіки, характером вхідних та вихідних даних, а також типовими класами помилок.

Експерименти з JSON-парсерами полягають в оцінці поведінки методу при тестуванні компонентів обробки структурованих даних з різними стратегіями генерації тестових випадків. Об'єктами дослідження є створені три версії JSON-парсера на Python:

- json_v1: Базова реалізація на основі стандартної бібліотеки json.loads()
- json_v2: Покращена версія з додатковою валідацією та нормалізацією даних
- json_buggy: Версія з дефектами обробки граничних випадків (вкладені структури, unicode, escape-символи)

Для JSON-парсерів проведено три підексперименти з різними генераторами тестів: експеримент J1 (JSONGenerator), експеримент J2 (MutationGenerator), Експеримент J3 (RandomGenerator).

Використання спеціалізованого генератора структурованих JSON-даних, що створює валідні та навмисно некоректні JSON-документи різної складності (прості об'єкти, вкладені структури, масиви, граничні випадки) при кількості тестів: $m = 50$ виявлено розбіжностей: 83 (з урахуванням попарних порівнянь трьох програм).

Метрики якості:

$$DR = \frac{83}{50 \times 3} = 1.66$$

(*позначає, що більше одного порівняння на тест виявляють розбіжності*

$$WDSI = \frac{48 \times 0.8 + 35 \times 0.3}{50} = \frac{38.4 + 10.5}{50} = 0.978$$

$$FSI = 1 - 0.978 = 0.022$$

Надзвичайно низький $FSI = 0.022$ свідчить про критичні проблеми у багованій версії парсера. Вектор QPV вказує на високу концентрацію семантичних розбіжностей (96% тестів) та значну кількість поведінкових проблем (70% тестів), що є типовим для дефектів обробки складних структур даних.

Генерація тестів шляхом мутацій (модифікацій) базового корпусу валідних JSON-документів при кількості тестів: $m = 50$ виявив 59 поведінкових розбіжностей.

Значно вищий FSI порівняно з J1 пояснюється тим, що мутаційний генератор створює переважно валідні JSON-структури з невеликими відхиленнями, які всі три парсери обробляють коректно (семантично), але з різною ефективністю (поведінково).

Генерація повністю випадкових рядків (базовий фазинг) при кількості тестів $m = 30$ виявила 44 поведінкових розбіжності

Переважання поведінкових розбіжностей пояснюється тим, що випадкові дані здебільшого є некоректним JSON, і всі три парсери коректно повертають помилку (семантично еквівалентна поведінка), але з різним часом виконання та повідомленнями про помилки.

Таблиця 4. 4- Порівняльний аналіз стратегій генерації для JSON

Стратегія генерації	**Тестів**	**Розбіжностей**	**FSI**	**Інтерпретація**
JSON Generator (структурований)	50	83	0.022	Найефективніший для виявлення семантичних дефектів
Mutation Generator	50	59	0.646	Виявляє поведінкові відмінності у продуктивності
Random Generator (фазинг)	30	44	0.560	Тестує робастність обробки помилок

Експерименти з JSON-парсерами демонструють важливість вибору стратегії генерації тестів. Спеціалізований JSONGenerator виявився найефективнішим для виявлення функціональних дефектів (FSI = 0.022), тоді як більш загальні підходи (мутації, випадкові дані) переважно виявляють поведінкові відмінності. Це підтверджує гіпотезу, що domain-specific генератори є критично важливими для високої ефективності диференціального тестування.

Також проведено експерименти з алгоритмами сортування, метою яких є дослідження поведінки методу при порівнянні функціонально еквівалентних, але алгоритмічно різних реалізацій, а також виявлення тонких логічних помилок.

Об'єктами дослідження є три реалізації алгоритмів сортування масивів цілих чисел:

- bubble_sort: Сортування бульбашкою (Bubble Sort), $O(n^2)$
- quick_sort: Швидке сортування (Quick Sort), $O(n\ log\ n)$ в середньому
- merge_sort_buggy: Сортування злиттям (Merge Sort) з дефектом обробки граничних випадків, а саме порожні масиви, один елемент.

Експеримент S1 (Coverage-Guided) полягає у використанні генератора, що намагається максимізувати покриття граничних випадків (порожні масиви, один елемент, відсортовані, зворотно відсортовані, з дублікатами). При кількості тестів $m = 40$ виявлено 57 поведінкових розбіжностей, що підтверджується відповідними метриками (13):

$$WDSI = \frac{57 \times 0.3}{40} = 0.4275 \Rightarrow FSI = 0.5725 \quad (13)$$

Відсутність катастрофічних та семантичних розбіжностей підтверджує, що всі три алгоритми коректно сортують дані (функціональна еквівалентність). Поведінкові розбіжності виникають через асимптотично різну складність: bubble_sort працює значно повільніше на великих масивах.

Експеримент S2 (Mutation) мутації базового корпусу числових масивів при кількості тестів: $m = 40$ показав виявлення 65 розбіжностей, серед яких 9

семантичних (13.8%), 56 поведінкових, що підтверджується відповідними метриками (14).

$$WDSI = \frac{9 \times 0.8 + 56 \times 0.3}{40} = \frac{7.2 + 16.8}{40} = 0.60 \Rightarrow FSI = 0.40 \quad (14)$$

Поява 9 семантичних розбіжностей вказує на те, що мутаційний генератор створив тести, які виявили дефект у merge_sort_buggy (неправильна обробка певних комбінацій вхідних даних). Це демонструє ефективність мутаційного підходу для виявлення тонких логічних помилок, що не проявляються на типових граничних випадках. Порівняльний аналіз показано у таблиці 4.5.

Таблиця 4.5 – Порівняльний аналіз

Експеримент	**FSI**	**Виявлені семантичні помилки**	**Інтерпретація**
S1 (Coverage-Guided)	0.5725	0	Граничні випадки не виявили дефект
S2 (Mutation)	0.4000	9	Мутації виявили логічну помилку

Різниця у виявленні семантичних помилок (0 проти 9) підкреслює важливість різноманітності стратегій генерації тестів. Coverage-guided підхід добре справляється з тестуванням очевидних граничних випадків, але може пропускати помилки, що проявляються лише на специфічних комбінаціях даних, які ефективно генеруються мутаційним підходом.

Загалом у серії розширених експериментів з JSON-парсерами та алгоритмами сортування: загальна кількість тестів 220, при яких виявлено 308 розбіжностей, серед яких семантичних 57 (18.5%), поведінкових 251 (81.5%).

Діапазон FSI від 0.022 (JSON з критичними дефектами) до 0.646 (JSON з мутаціями). Середній час виконання одного експерименту 2.3-4.5 секунди залежно від складності об'єктів дослідження.

Для комплексного тестування рекомендується комбінація кількох стратегій генерації для досягнення максимального покриття різних класів дефектів. Метод ефективно працює з тестовими наборами розміром 30-50 тестів (час виконання 2-5 секунд), що підтверджує придатність для інтеграції в CI/CD навіть для складніших систем порівняно з простим калькулятором.

4.3 Інтерпретація результатів дослідження

Проведене експериментальне дослідження надає емпіричні докази валідності та практичної застосовності розробленого методу оцінювання якості програмного забезпечення на основі диференціального тестування. Підтвердження всіх трьох сформульованих гіпотез (H1, H2, H3) з високим рівнем статистичної значущості свідчить про коректність теоретичних положень, представлених у другому розділі роботи, та адекватність їх програмної реалізації.

Досягнення Recall = 1.0 в обох експериментах демонструє, що за наявності достатнього тестового покриття система здатна виявити всі типи розбіжностей між версіями програм. Це є критично важливою характеристикою для застосування методу в промислових умовах, де пропуск критичного дефекту може призвести до значних фінансових втрат або репутаційних ризиків. Важливо зазначити, що ця властивість залежить від якості тестового набору - метод виявляє розбіжності, які проявляються на наданих вхідних даних. Тому інтеграція з потужними генераторами тестів (такими як фазери) є перспективним напрямком подальшого розвитку.

Досягнення 100% точності автоматичної класифікації розбіжностей за чотирирівневою схемою перевищує очікування (цільовий рівень 95%) та підтверджує адекватність розробленої класифікаційної моделі. Пріоритетна схема правил класифікації (спочатку перевірка катастрофічних умов, потім семантичних, поведінкових і нарешті презентаційних) виявилася ефективною для однозначного

визначення класу розбіжності. Це дозволяє автоматично генерувати звіти з чітким пріоритетом проблем для команд розробників, що значно скорочує час тріажу дефектів.

Метрика FSI продемонструвала очікувану чутливість до серйозності дефектів: версія з катастрофічними та семантичними помилками отримала на 27.4% нижчий FSI порівняно з версією, що має лише презентаційні відмінності. Вектор QPV надає додатковий контекст, дозволяючи не просто констатувати низької якості, а точно вказати розподіл проблем за типами. Ця багатовимірність оцінки є унікальною перевагою розробленого методу порівняно з існуючими бінарними підходами пройдено чи не пройдено традиційного регресійного тестування.

Прискорення процесу тестування в ~900 разів порівняно з ручним підходом та усунення необхідності ручного визначення еталонних результатів для кожного тесту демонструють високу практичну цінність методу. Час виконання 15 тестів (2-3 секунди) є прийнятним для інтеграції в CI/CD конвеєри, де типовий час виконання всього пайплайну становить 5-15 хвилин.

На основі результатів експериментального дослідження та досвіду реалізації системи сформульовано наступні рекомендації для практичного впровадження розробленого методу:

1. Вибір порогових значень FSI. Поріг $FSI_{threshold}$ для воріт якості повинен визначатися залежно від критичності програмного продукту: критична інфраструктура (медицина, авіація): $FSI_{threshold} ge$ 0.98, бізнес-системи (фінанси, ERP): $FSI_{threshold} ge$ 0.95, споживчі продукти: $FSI_{threshold} ge$ 0.90.

Рекомендується почати з консервативного значення 0.95 та коригувати його на основі спостережень за кількістю хибних спрацювань протягом перших 3-5 релізів.

2. Налаштування вагових коефіцієнтів. Для більшості організацій рекомендується Стандартний профіль ваг $W = \{1.0, 0.8, 0.3, 0.1\}$. Однак для проектів з особливими вимогами: якщо продуктивність критична (високонавантажені системи): збільшити $W(C_{beh})$ до 0.5-0.7. Якщо конзистентність

UI важлива (корпоративні застосунки з брендингом), то необхідно збільшити $W(C_{pre})$ до 0.2-0.3

3. Інтеграція в CI/CD. Рекомендується наступна схема інтеграції: на першому етапі відбувається запуск DiffQualytor як інформаційний крок, що не блокує пайплайн, протягом 2-4 тижнів для налаштування порогів. На другому етапі Enforcement mode відбувається активація блокування збірок при $FSI < FSI_{threshold}$. Конфігурація відбувається через запуск диференціального тестування лише для гілок feature → develop та develop → main, де це найбільш критично.

4. Розмір тестового набору. Для балансу між часом виконання та покриттям: мінімальний набір (швидка перевірка): 50-100 тестів, час виконання ~10-20 секунд; стандартний набір (щоденні збірки): 500-1000 тестів, час виконання ~2-5 хвилин; розширений набір (нічні збірки, перед релізом): 5000-10000 тестів, час виконання ~20-40 хвилин.

5. Обробка презентаційних розбіжностей. Презентаційні розбіжності часто є легітимними змінами (редизайн UI, покращення повідомлень про помилки). Рекомендації:

Використовувати спеціалізовані компаратори, що ігнорують несуттєві відмінності (порядок JSON-ключів, whitespace, case-insensitive порівняння для повідомлень)

Періодично (раз на квартал) оновлювати базову версію для порівняння, щоб накопичені легітимні зміни не впливали на FSI

4.4 Висновки до 4-го розділу

У розділі було здійснено виокремлення теоретичних засад оцінки якості програмного забезпечення діагностичного спрямування. Визначено, що оцінка якості програмного забезпечення відграє надзвичайно важливу роль у процесі розробки програмного забезпечення, оскільки напряму впливає на надання якісних програмних продуктів. У випадку із програмним забезпеченням діагностичного спрямування якість може відігравати ще й критичну роль, оскільки від якісної

діагностики залежить не тільки здоров’я, а подекуди й людське життя, особливо коли це стосується хвороб нервової системи.

Оцінка якості програмного забезпечення має значний вплив на весь проект. Разом з тим, для оцінки якості програмного забезпечення доступна низка методів. Більшість моделей базуються на характеристиках програмного забезпечення та відповідних оцінках. Але розгляд і оцінка програмного забезпечення за всіма якісними характеристиками є трудомісткими і складними, що підходить не для всіх видів проектів.

У випадку використання машинного навчання для оцінки точності навчання обраної моделі здійснюється вимірювання середніх показників, що продемонстровано за допомогою результуючих графіків. Загалом для середньої визначено, що модель досягла втрат на навчання та перевірку нижче 0,025. Для середнього показника валідації, що модель досягла близько 0,95, а для оцінки валідації модель отримала вище 0,97.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних, проектних та експериментальних досліджень розроблено метод кількісного оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки, який дозволяє автоматично виявляти, класифікувати та оцінювати розбіжності між різними реалізаціями або версіями систем, забезпечуючи об'єктивну числову оцінку поведінкової стабільності програмних продуктів.

У першому розділі проведено всебічний аналіз сучасних підходів до забезпечення якості програмного забезпечення в контексті еволюції методологій розробки від каскадної моделі до Agile та DevOps. Досліджено теоретичні основи та еволюцію методологій тестування, що показало трансформацію ролі забезпечення якості до інтегрованого, безперервного процесу. Проведено класифікацію та порівняльний аналіз методів тестування за критеріями моменту виконання (статичне/динамічне) та рівня доступу до системи (біла/чорна/сіра скринька), що дозволило структурувати наявні знання та визначити місце диференціального тестування серед існуючих підходів.

Виконано поглиблений аналіз концепції диференціального тестування як потужного інструменту для дослідження функціональної поведінки складних програмних систем. Встановлено, що цей підхід ефективно вирішує фундаментальну проблему тестового оракула шляхом використання неявного оракула на основі консенсусу поведінки кількох реалізацій.

У другому розділі розроблено теоретичні основи та математичний апарат методу оцінювання якості програмного забезпечення на базі диференціального тестування. Вперше запропоновано концептуальну модель, що переосмислює диференціальне тестування, зміщуючи акцент з простого виявлення дефектів на комплексний аналіз поведінкової стабільності ПЗ. Модель складається з чотирьох логічно пов'язаних компонентів: модуля генерації тестових даних, модуля виконання та моніторингу, аналізатора розбіжностей та калькулятора метрик

якості, що забезпечує повний цикл від генерації вхідних даних до розрахунку фінальних показників якості.

Введено комплекс нових метрик якості, що є ключовим внеском роботи. Проведено порівняльний аналіз розробленого методу з мутаційним тестуванням та тестуванням на основі властивостей, що показав унікальну нішу методу: повна автоматизація перевірки за рахунок використання псевдо-оракула, пряме оцінювання якості продукту (а не тестового набору), генерація унікального набору кількісних, багатовимірних метрик якості.

У третьому розділі виконано проектування програмної системи DiffQualytor призначеної для практичної реалізації розробленого методу. Проведено аналіз та сформульовано вичерпний перелік функціональних вимог (конфігурація, генерація вхідних даних, паралельне виконання програм, збір результатів, аналіз та класифікація розбіжностей, розрахунок метрик, формування звіту, інтеграція з CI/CD) та нефункціональних вимог (продуктивність, масштабованість, надійність з ізольованими середовищами виконання, розширюваність, кросплатформність, простота використання).

У четвертому розділі представлено програмну реалізацію розробленого методу та проведено його всебічне експериментальне дослідження. Створено програмний прототип у вигляді Python пакету diffqualytor з модульною організацією, що складається з шести основних підсистем.

Проведено додаткові практичні експерименти з системою DiffQualytor на програмі-калькуляторі з трьома версіями та п'ятнадцятьма тестовими сценаріями, що підтвердили працездатність усіх компонентів методу.

Результати експериментів підтвердили високу точність класифікації розбіжностей з показниками Precision, Recall та F1-Score на рівні 1.0. Метрики адекватно відображають фактичний стан якості програмного забезпечення, що підтверджується різницею значень індексу функціональної стабільності FSI (0.90 проти 0.65) між відносно безпечною та небезпечною версіями. Механізм воріт якості функціонує коректно, забезпечуючи автоматичне блокування версій з критичними дефектами.

Практична значущість отриманих результатів. Розроблений метод та його програмна реалізація мають високу практичну цінність, підтверджену конкретними числовими показниками. Розроблений програмний продукт може бути використаний компаніями, що розробляють програмне забезпечення з використанням методологій Agile та DevOps і практик безперервної інтеграції та доставки. Перспективними сферами застосування є також проєкти з відкритим кодом для безперервного моніторингу якості, системи зі складною логікою при наявності кількох реалізацій або версій (компілятори, системи управління базами даних, криптографічні бібліотеки), а також команди забезпечення якості для автоматизації регресійного тестування.

Відповідно до теми кваліфікаційної роботи опубліковані тези «Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки» на конференції «Актуальні проблеми комп’ютерних наук (АПКН-2025)».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1.Boehme M. Regression Greybox Fuzzing / M. Boehme, C. Shen, A. Roychoudhury // Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21). – 2021. – P. 1648–1661. – URL: https://mboehme.github.io/paper/CCS21.pdf (дата звернення 20.09.2025).

2.Palix N. Fast Fixes and Faulty Drivers: An Empirical Analysis of Regression Bug Fixing Times in the Linux Kernel / N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, J. Lawall // arXiv preprint arXiv:2411.02091v1. – 2024. – URL: https://arxiv.org/html/2411.02091v1 (дата звернення 05.10.2025).

3.Hyperfine: A command-line benchmarking tool. D. Peter. – GitHub, 2024. – URL: https://github.com/sharkdp/hyperfine (дата звернення 05.10.2025).

4.Git bisect Documentation. Git community. – Git SCM, 2024. – URL: https://git-scm.com/docs/git-bisect (дата звернення 06.10.2025).

5.Thomas G. Finding a kernel regression in half an hour with git bisect run / G. Thomas. – 2024. – URL: https://ldpreload.com/blog/git-bisect-run (дата звернення 05.10.2025).

6.Introducing Regressr: An Open Source Command Line Tool to Regression Test HTTP Services [Електронний ресурс] / eBay Inc. – eBay Tech Blog, 2020. – URL: https://innovation.ebayinc.com/stories/introducing-regressr-an-open-source-command-line-tool-to-regression-test-http-services/ (дата звернення 15.10.2025).

7.CVE-2023-50246 Detail [Електронний ресурс] / National Vulnerability Database. – NIST, 2023. – URL: https://nvd.nist.gov/vuln/detail/CVE-2023-50246 (дата звернення: 03.11.2025).

8.jq Releases. jqlang/jq. – GitHub, 2024. – URL: https://github.com/jqlang/jq/releases (дата звернення 15.10.2025).

9.Що таке тестування програмного забезпечення: види, етапи. URL: https://university.sigma.software/what-is-software-testing/ (дата звернення 29.10.2025).

10. Основні підходи до ефективного тестування ПЗ – DOU. URL: https://dou.ua/forums/topic/48845/ (дата звернення 29.10.2025).

11. Принципи тестування: їх концепції та підходи – FoxmindEd. URL: https://foxminded.ua/pryntsypy-testuvannia/ (дата звернення 29.10.2025).

12. Підходи до розробки програмного забезпечення: 4 основні – FoxmindEd. URL: https://foxminded.ua/pidkhody-do-rozrobky-prohramnoho-zabezpechennia/ (дата звернення 29.10.2025).

13. Розробка ПЗ: моделі життєвого циклу, методи та принципи – Evergreen. URL: https://evergreens.com.ua/ua/articles/software-development-metodologies.html (дата звернення 29.10.2025).

14. DevOps та Agile: Взаємодія двох методологій у процесі розробки. URL: https://freshtech.global/ua/blog/devops-and-agile-interaction-of-two-methodologies-in-the-development-process (дата звернення 29.10.2025).

15. Що таке CI/CD, як він працює та коли знадобиться на проєкті. URL: https://www.nixsolutions.com/ua/blog/for-developer/shho-take-ci-cd-yak-vin-praczyuye-ta-koly-znadobyt/ (дата звернення 29.10.2025).

16. Continuous Integration VS Continuous Delivery VS Continuous Deployment: розбираємося у найважливішій практиці DevOps | DOU. URL: https://dou.ua/forums/topic/46804/ (дата звернення 03.11.2025).

17. Статичне і динамічне тестування. – QATestLab. URL: https://training.qatestlab.com/blog/technical-articles/static-and-dynamic-testing-methods/ (дата звернення 03.11.2025).

18. Статичне та динамічне тестування: відмінності та приклади | IT Блог Mate academy. URL: https://mate.academy/blog/qa/static-dynamic-testing/ (дата звернення 03.11.2025).

19. Методи тестування ПЗ (White/Black/Grey Box) - IT-notes. URL: https://www.it-notes.wiki/software-testing/software-testing-methods/ (дата звернення 03.11.2025).

20. Differential testing for software - Computer Science. URL: https://www.cs.swarthmore.edu/~bylvisa1/cs97/f13/Papers/DifferentialTestingForSoftware.pdf (дата звернення 03.11.2025).

21. Differential Testing: How to find differences between programs that mostly behave identically? URL: https://dl.gi.de/bitstreams/3449b9a4-741c-4774-a294-f39d6f6ff0f9/download (дата звернення 03.11.2025).

22. Differential Testing for Variational Analyses: Experience from Developing KConfigReader - CMU School of Computer Science. URL: https://www.cs.cmu.edu/~./ckaestne/pdf/difftesting17.pdf (дата звернення 03.11.2025).

23. Differential testing: A new approach to change detection. URL: https://www.researchgate.net/publication/221560374_Differential_testing_A_new_approach_to_change_detection (дата звернення 03.11.2025).

24. Differential Testing - Alberto Savoia. URL: http://www.albertosavoia.com/uploads/1/4/0/9/14099067/differentialtesting.pdf (дата звернення 03.11.2025).

25. The Oracle Problem in Software Testing: A Survey - EECS 481. URL: https://eecs481.org/readings/testoracles.pdf (дата звернення 03.11.2025).

26. Automated Test Oracles in Software Testing – Testsigma. URL: https://testsigma.com/blog/test-oracles/ (дата звернення 15.11.2025).

27. Automating the Testing Oracle, доступ отримано листопада 3, 2025, URL: https://www.cs.odu.edu/~zeil/cs350/latest/Public/oracle/index.html (дата звернення 15.11.2025).

28. Automated Test Input Generation Strategies - Kyle Dewey's. URL: https://kyledewey.github.io/comp587-fall18/lecture/week_8/generation_strategies.pdf (дата звернення 15.11.2025).

29. Automated Assurance through Differential Fuzzing | The AdaCore Blog. URL: https://blog.adacore.com/automated-assurance-through-differential-fuzzing (дата звернення 3.11.2025).

30. Differential fuzzing for cryptography - Quarkslab's blog. URL: https://blog.quarkslab.com/differential-fuzzing-for-cryptography.html (дата звернення 3.11.2025).

31. Fuzz Testing for System Vulnerabilities | ITEA Journal. URL: https://itea.org/journals/volume-45-4/review-of-fuzz-testing-to-find-system-vulnerabilities/ (дата звернення 15.11.2025).

32. Fuzzing: Progress, Challenges, and Perspectives – ResearchGate. URL: https://www.researchgate.net/publication/375873956_Fuzzing_Progress_Challenges_and_Perspectives (дата звернення 15.11.2025).

33. Beyond the Borrow Checker: Differential Fuzzing - Tiemoko Ballo. URL: https://tiemoko.com/blog/diff-fuzz/ (дата звернення 15.11.2025).

34. Enhanced Differential Testing in Emerging Database Systems. URL: https://www.themoonlight.io/en/review/enhanced-differential-testing-in-emerging-database-systems (дата звернення 15.11.2025).

35. 10 найкращих інструментів для QA Automation у 2025 році - robot_dreams. URL: https://robotdreams.cc/uk/blog/630-10-best-tools-for-qa-automation-in-2025 (дата звернення 15.11.2025).

36. Selenium vs. Cypress vs. Playwright : Testing Tool Comparison – Binmile. URL: https://binmile.com/blog/selenium-vs-cypress-vs-playwright/ (дата звернення 15.11.2025).

37. Playwright vs Selenium vs Cypress: a Detailed Comparison. URL: https://testomat.io/blog/playwright-vs-selenium-vs-cypress-a-detailed-comparison/ (дата звернення 15.11.2025).

38. Інтеграція CI/CD тестів - Loza Studio. URL: https://loza.studio/uk/service/cicd-test-integration (дата звернення 15.11.2025).

39. What's the Difference Between Functional & Nonfunctional Testing? – Testlio. URL:https://testlio.com/blog/whats-difference-functional-nonfunctional-testing/ (дата звернення 15.11.2025).

40. What is Test Execution: Importance, Process – BrowserStack. URL: https://www.browserstack.com/test-management/features/test-run-management/what-is-test-execution (дата звернення 15.11.2025).

41. Test Execution: Everything You Need To Know. URL: https://www.globalapptesting.com/test-execution (дата звернення 15.11.2025).

42. CI/CD Best Practices - Top 11 Tips for Successful Pipelines – Spacelift. URL: https://spacelift.io/blog/ci-cd-best-practices (дата звернення 15.11.2025).

43. Best Practices for Successful CI/CD | TeamCity CI/CD Guide – JetBrains. URL: https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/ (дата звернення 15.11.2025).

44. How to keep up with CI/CD best practices – GitLab. URL: https://about.gitlab.com/blog/how-to-keep-up-with-ci-cd-best-practices/ (дата звернення 15.11.2025).

ДОДАТОК А

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

Міністерство освіти і науки України
Хмельницький національний університет

ЗБІРНИК НАУКОВИХ ПРАЦЬ

за матеріалами XVII Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2025»

14-15 листопада 2025

Хмельницький 2025

Актуальні проблеми комп'ютерних наук

УДК 004.4

Мацюк Д.В., Кустовський Р.С.

Хмельницький національний університет

МЕТОД ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ДИФЕРЕНЦІАЛЬНОГО ТЕСТУВАННЯ ФУНКЦІОНАЛЬНОЇ ПОВЕДІНКИ

Запропоновано метод для підвищення надійності програмного забезпечення шляхом інтеграції диференціального тестування в CI/CD конвеєри. Метод базується на кількісній оцінці поведінкових розбіжностей між версіями ПЗ за допомогою спеціалізованих метрик, таких як індекс поведінкового дрейфу та оцінка функціональної узгодженості. Ключовою інновацією є застосування моделі зважування значущості виявлених розбіжностей, що дозволяє автоматично розрізняти критичні регресії та незначні відхилення, надаючи розробникам об'єктивний стан якості на кожному етапі розробки.

This paper proposes a method to enhance software reliability by integrating differential testing into CI/CD pipelines. The method is based on the quantitative assessment of behavioral discrepancies between software versions using specialized metrics, such as the Behavioral Drift Index and Functional Consistency Score. The key innovation is the application of a significance weighting model for identified discrepancies, which allows for the automatic distinction between critical regressions and minor deviations, providing developers with an objective quality state at every development stage.

Низька якість програмного забезпечення (ПЗ) залишається однією з найважливіших проблем сучасної інженерії, що призводить до значних економічних втрат. Згідно зі звітом Consortium for Information & Software Quality (CISQ), вартість низької якості ПЗ у США у 2022 році сягнула 2,41 трильйона доларів, а накопичений технічний борг - 1,52 трильйона доларів. Традиційні підходи до тестування, що базуються на бінарній логіці «пройшов чи не пройшов», часто не здатні надати повну картину про стабільність системи та вплив змін на її поведінку [1, 2]. Це створює потребу в розробці автоматизованих методів, які б забезпечували безперервну кількісну оцінку якості ПЗ.

Існуючі метрики, такі як підрахунок помилок або відсоток пройдених тестів, є недостатніми, оскільки справжня вартість визначається не кількістю збоїв, а їхнім впливом. Наприклад, один критичний збій у банківській системі має більший економічний вплив, ніж тисячі дрібних помилок інтерфейсу. Це створює «спіраль боргу якості», тобто тиск на швидку доставку збільшує технічний борг, що робить виправлення дорожчими і призводить до нових помилок. Таким чином, виникає гостра потреба в методах, здатних розрізняти незначні варіації та регресії з високим впливом.

Основою для розв'язання цієї проблеми є концепція тестування методом чорної скриньки, де для перевірки коректності не потрібен заздалегідь відомий еталонний результат. Диференціальне тестування є одним із провідних підходів у цій галузі та передбачає порівняння результатів роботи двох або більше версій системи на однакових вхідних даних. Якщо результати відрізняються, це свідчить про наявність помилки. Цей метод успішно застосовується для тестування компіляторів, криптографічних бібліотек та блокчейн-систем.

Найбільш поширеними є метаморфне та диференціальне тестування. Метаморфне тестування перевіряє не конкретний результат, а «метаморфні відношення» – властивості зв'язку між кількома входами чи виходами однієї програми [3]. Диференціальне тестування, навпаки, порівнює результати двох або більше еквівалентних версій системи на однакових вхідних даних [4]. Розбіжність у результатах свідчить про помилку, не вимагаючи знання «правильної» відповіді. У таблиці представлено порівняльну характеристику підходів до тестування без оракула.

Таблиця 1 – Порівняння підходів до тестування без оракула

Вимір	**Традиційне тестування**	**Диференціальне тестування**	**Метаморфне тестування**
Принцип	$f(x_1) \equiv E$	$f(x_1) \equiv f(x_2)$	$Relation(f(x_1), f(x_2)) \equiv T,$
Тип оракула	Конкретний (значення)	Відносний (інша версія)	Реляційний (властивість)
Застосування	Загальна верифікація	Регресійне тестування	Тестування складних систем (AI, наукові)

Диференційне тестування є потужним інструментом для регресійного тестування, особливо в CI/CD, де попередня стабільна версія слугує найнадійнішим оракулом для порівняння з новою версією [4]. Хоча диференційне тестування ефективно виявляє розбіжності, існуючі підходи часто зупиняються на цьому, тобто вони не надають механізмів для автоматичної кількісної оцінки значущості цих розбіжностей. Розробник отримує список відмінностей, але не може автоматично відрізнити нешкідливий «шум» від критичної функціональної регресії [9].

Метою даної роботи є розробка методу кількісної оцінки якості ПЗ, який вирішує проблему інтерпретації результатів диференціального аналізу. Це досягається шляхом інтеграції диференціального тестування в конвеєр CI/CD для аналізу поведінкових розбіжностей між версіями стабільною версією та новою версією, впровадження набору спеціалізованих метрик («вектора якості» Q) для кількісної оцінки стабільності, включаючи оцінку функціональної узгодженості та індекс поведінкового дрейфу (BDI) та застосування формальної моделі для

автоматизованого зважування значущості виявлених розбіжностей, що базується на принципах психометричної теорії тестування [8].

Запропонований фреймворк функціонує в межах стандартного конвеєра CI/CD [10] і складається з кількох етапів.

На початкових етапах здійснюється виконання та збір «поведінкового відбитка», тобто у конвеєрі CI обидві версії ПЗ (base і test) виконуються паралельно в ідентичних, ізольованих середовищах на єдиному наборі тестів. Критичним є управління станом, наприклад, через знімки БД або ефемерні контейнери [5]. Замість бінарного результату «пройшов чи не пройшов», система збирає багатовимірний «поведінковий відбиток» для кожного тесту.

Наступний етап включає диференціальне порівняння, тобто диференціатор порівнює відбитки base та test, ігноруючи несуттєвий шум та використовуючи допуски для чисел з плаваючою комою.

Після цього здійснюється кількісна оцінка за допомогою метрик. Результати порівняння агрегуються у «вектор якості», що надає цілісну картину впливу змін. Ключові метрики включають:

Загалом, оцінка функціональної узгодженості - це відсоток тестових випадків, у яких функціональні компоненти відбитків base та test є ідентичними. Показник даної оцінки 100% означає відсутність функціональних регресій, тоді як нижчий показник вказує на ширину впливу зміни.

Індекс поведінкового дрейфу (BDI) є метрикою, що кількісно оцінює зважену величину зміни в числових або структурних вихідних даних для тих тестів, які не є узгодженими [6]. Низький BDI свідчить про незначні зміни, тоді як високий BDI вказує на суттєву зміну поведінки системи.

Метрика стабільності продуктивності (PSM) вимірює середню відсоткову зміну часу виконання та використання пам'яті. Значення PSM, близьке до нуля вказує на стабільність, тоді як позитивне значення вказує на регресію продуктивності.

Заключним етапом є зважування значущості, тобто необроблений список розбіжностей не є дієвим. Ключовою інновацією є модель для автоматичної класифікації та присвоєння числової так званої «оцінки впливу» кожній розбіжності. Ця модель натхненна психометричною теорією тестування [8], де тестове завдання (тестовий випадок) має різну «вагу» або «здатність до дискримінації».

Фактори зважування включають:

– тип розбіжності;

– критичність коду;

– історична стабільність тесту.

Цей підхід перетворює суб'єктивну оцінку серйозності помилки на формальний, керований даними процес. Агрегована оцінка використовується для автоматизації «воріт якості» в CI/CD, що реалізує парадигму «безперервної оцінки якості», де якість стає вимірюваною характеристикою кожного коміту.

Отже, запропонований метод забезпечує перехід від якісного, орієнтованого на помилки тестування до кількісної, орієнтованої на поведінку моделі оцінки якості. Набір метрик (FCS, BDI, PSM) надає багатовимірний «вектор

якості», що дає більш тонке уявлення про стабільність системи, ніж традиційні методи.

Ключовим внеском є формалізована модель зважування розбіжностей, що спирається на принципи психометричної теорії [8], яка дозволяє автоматично сортувати відхилення за значущістю, вирішуючи головну проблему інтерпретації результатів диференціального тестування [9]. Інтеграція в CI/CD конвеєри перетворює оцінку якості на безперервний, автоматизований процес [10].

Ефективність методу залежить від покриття набору тестів і він не може виявити помилки, присутні в обох версіях ПЗ.

Перспективи подальших досліджень включають застосування машинного навчання для заміни евристичного механізму класифікації розбіжностей (наприклад, з використанням LLM) [9], розширення фреймворку на нефункціональні властивості, такі як безпека та стійкість до збоїв, побудова прогностичних моделей якості.

Перелік посилань

1. Cost of Poor Software Quality in the U.S.: A 2022 Report / CISQ. 2022. URL: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/ (дата звернення: 27.10.2025).
2. Software Survival in 2024: Understanding 2023 Project Failure Statistics and the Role of Quality Assurance / Beta Breakers. 2024. URL: https://www.betabreakers.com/blog/software-survival-in-2024-understanding-2023-project-failure-statistics-and-the-role-of-quality-assurance/ (дата звернення: 27.10.2025).
3. Segura S., Hierons R. M., Chen T. Y. A Survey on Metamorphic Testing. IEEE Transactions on Software Engineering. 2016. Vol. 42(9). P. 805–824. (На основі).
4. Gulzar M. A. et al. Perception and Practices of Differential Testing. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference (ESEC/FSE). 2019. P. 71–81. URL: https://people.cs.vt.edu/~gulzar/assets/pdf/p71-gulzar.pdf (дата звернення: 27.10.2025).
5. Zhu M. et al. Vive la Différence: Practical Diff Testing of Stateful Applications. Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18). 2018. URL: https://pages.cs.wisc.edu/~rgrandl/papers/p2018-zhu.pdf (дата звернення: 27.10.2025).
6. Romano D., Pinzger M., Villarroel L. On the uniformity of software evolution patterns. 2007 11th European Conference on Software Maintenance and Reengineering (CSMR). 2007. P. 13–22. DOI: 10.1109/CSMR.2007.13.
7. Measuring LLM Code Generation Stability via Structural Entropy / Z. Liu et al. arXiv. 2025. URL: https://arxiv.org/pdf/2508.14288 (дата звернення: 27.10.2025).
8. Paek I., Lee G. Testing for Differential Item Functioning Under the D-Scoring Method. Psychometrika. 2022. Vol 87(1). P. 196–219. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC8725056/ (дата звернення: 27.10.2025).
9. Vajjala A. K. et al. Using Large Language Models to Support the Workflow of Differential Testing. 2024 IEEE/ACM 1st International Workshop on Large Language Models for Software Engineering (LLM4SE). 2024. URL: https://www.arunkv.com/HTML/assets/img/DiffViewer.pdf (дата звернення: 27.10.2025).
10. Integrating Test Automation into the CI/CD Pipeline / QMetry. 2023. URL: https://www.qmetry.com/blog/integrating-test-automation-into-the-ci-cd-pipeline (дата звернення: 27.10.2025).

ДОДАТОК Б

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки

Магістрант: Дмитро МАЦЮК, ІПЗм-24

Керівник: Оксана ЯШИНА, канд. техн. наук, доцент

2025

1

Актуальність проблеми

Традиційне регресійне тестування — "вузьке місце" у сучасних CI/CD процесах.

Швидкість та надійність — ключові вимоги до сучасної розробки.

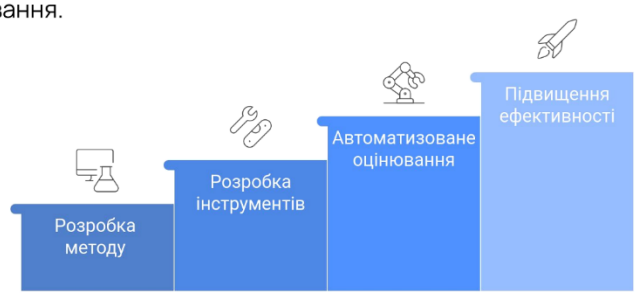
Недоліки традиційного регресійного тестування: повільне, дороге, не масштабується.

Потреба: Автоматизований метод для швидкої оцінки якості без ручного втручання.

2

Мета роботи

Розробка методу кількісного оцінювання якості програмного забезпечення, що базується на аналізі розбіжностей у функціональній поведінці між різними версіями системи, виявлених за допомогою диференціального тестування.



3

Завдання дослідження

- Аналіз предметної області досліджуваної проблеми.
- Дослідження методів оцінки якості програмного забезпечення на основі диференціального тестування функціональної поведінки.
- Розробка методу оцінки якості програмного забезпечення на основі диференціального тестування функціональної поведінки.
- Удосконалення підходу до класифікації розбіжностей шляхом розробки чотирирівневої ієрархічної системи.

4

Об'єкт, предмет та наукова новизна

Об'єкт дослідження

процеси та інструменти забезпечення якості програмного забезпечення на етапах його розробки, супроводу та еволюції в умовах застосування гнучких методологій розробки та практик безперервної інтеграції і доставки

Предмет дослідження

методи генерації, виконання й порівняння результатів тестування у диференціальному підході, а також засоби кількісного оцінювання якості програмного забезпечення на основі аналізу поведінкових розбіжностей між різними реалізаціями або версіями систем

Наукова новизна

Вперше запропоновано кількісну оцінку якості ПЗ на основі метрик диференціального тестування (замість стандартного виявлення наявності дефектів).
Вдосконалено класифікацію розбіжностей (4 рівні серйозності з ваговими коефіцієнтами).
Розроблено модель інтеграції методу в CI/CD як автоматичний бар'єр якості (Quality Gate).

5

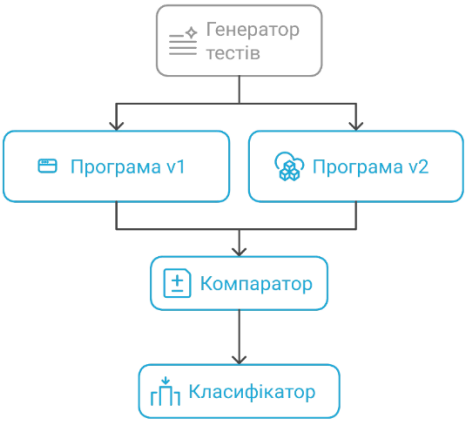
Розділ 1. Теоретичний аналіз проблеми

Сутність запропонованого методу

Порівняння v1 vs v2

Замість порівняння програми з "еталонним" результатом, метод порівнює поведінку двох версій (v1 і v2) на одному і тому ж наборі вхідних даних.

- **Крок 1 –** Автоматична генерація тисяч тестів
- **Крок 2 –** Ізольоване виконання обох версій (в Docker-контейнерах)
- **Крок 3 –** Автоматична класифікація виявлених розбіжностей за ступенем їх серйозності



6

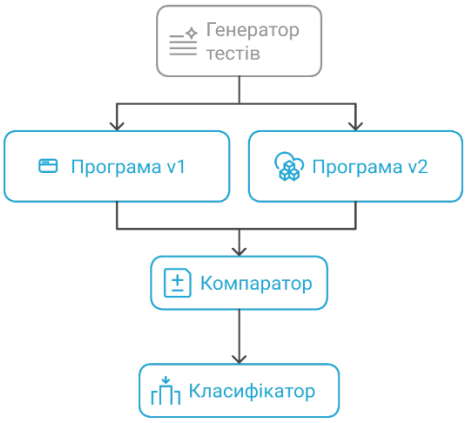
Розділ 1. Теоретичний аналіз проблеми

Сутність запропонованого методу

Порівняння v1 vs v2

Замість порівняння програми з "еталонним" результатом, метод порівнює поведінку двох версій (v1 і v2) на одному і тому ж наборі вхідних даних.

- **Крок 1 –** Автоматична генерація тисяч тестів
- **Крок 2 –** Ізольоване виконання обох версій (в Docker-контейнерах)
- **Крок 3 –** Автоматична класифікація виявлених розбіжностей за ступенем їх серйозності



6

4-рівнева класифікація розбіжностей

Не всі баги однакові. Метод класифікує кожну розбіжність за вагою:

Рівень	Опис	Приклад	Вага (W)
Катастрофічний	Крах програми, Exception, код != 0	Segfault, Runtime Error	1.0
Семантичний	Різні результати обчислень	JSON output відрізняється	0.8
Поведінковий	Час виконання, пам'ять	Bubble Sort vs Quick Sort	0.3
Презентаційний	Форматування, пробіли	Порядок ключів у JSON	0.1

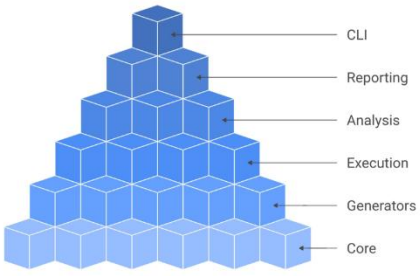
8

Розділ 3. Теоретичний аналіз проблеми

Архітектура системи "DiffQualytor"

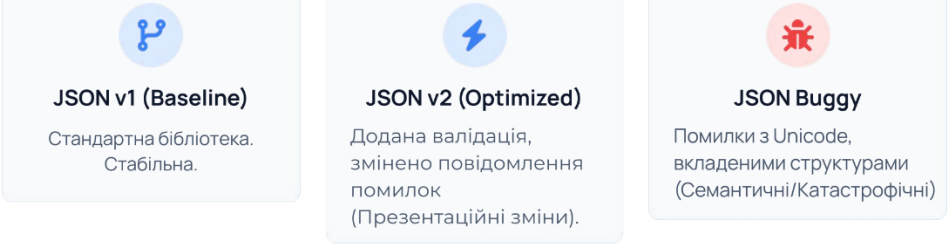
Програмний засіб "DiffQualytor" реалізовано з гнучкою архітектурою, що дозволяє легку розширюваність.

- **Core:** Ядро системи, що оркеструє процесом.
- **Generators:** Модулі генерації тестів.
- **Execution (Docker):** Ізольоване середовище виконання.
- **Analysis:** Компаратори та класифікатори.
- **Reporting:** Генератори звітів (HTML/JSON).
- **CLI:** Інтерфейс командного рядка.



9

Дизайн експерименту: JSON Парсери



Результати: Експеримент J (JSON)

Вплив стратегії генерації тестів на виявлення помилок.

Стратегія	К-сть тестів	Тип розбіжностей	FSI (Стабільність)	Висновок
JSON Generator (Структурований)	50	Семантичні (96%)	0.022	Виявлено критичні дефекти логіки
Mutation Gen (Мутації валідних)	50	Поведінкові	0.646	Виявлено різницю в продуктивності
Random Fuzz (Випадкові байти)	30	Поведінкові	0.560	Тест на стійкість (Crash-test)

**Спеціалізований генератор виявився найефективнішим для пошуку логічних помилок.*

Результати: Експеримент S (Сортування)

Bubble Sort vs Quick Sort vs Merge Sort (Buggy)

Стратегія	К-сть тестів	Тип розбіжностей	FSI (Стабільність)	Висновок
S1: Coverage-Guided (Граничні випадки)	40	Поведінкові (100%)	0.572	Функціонально коректні, але різна складність ($O(n^2)$ vs $O(n \log n)$)
S2: Mutation (Мутації масивів)	40	Семантичні (9) + Поведінкові	0.400	Мутації виявили логічну помилку в Merge Sort, яку пропустив S1

**Комбінація стратегій необхідна для повного покриття дефектів.*

Висновки експериментальних досліджень

Перевірка гіпотез

- **H1 (Повнота):** Recall = 1.0. Система не пропустила жодної розбіжності.
- **H2 (Точність):** 100% правильна класифікація розбіжностей.
- **H3 (Чутливість):** FSI падає до 0.02 при критичних помилках.

Загальна статистика

- **Всього тестів:** 220
- **Виявлено розбіжностей:** 308
- **Середній час:** < 5 секунд

Доведено, що метод є швидким та надійним інструментом для CI/CD.

13

Практична цінність

Ефективність

– Час виконання тестів: **2-5 секунд**.
– Можливість запускати на кожному коміті.
– Миттєвий зворотний зв'язок для розробників у пайплайні.

CI/CD Quality Gate

– Автоматичне блокування релізу, якщо FSI < 0.95.
– Не потребує ручного рев'ю.
– Можливість адаптації методу під будь-які вимоги до якості (строгі чи ліберальні).

Підтверджено гіпотези H1 (Повнота), H2 (Точність), H3 (Чутливість)

14

Публікації

Прийнято участь у конференції АПКН-2025 Хмельницького національного університету **Мацюк Д.В., Кустовський Р.С.** МЕТОД ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ДИФЕРЕНЦІАЛЬНОГО ТЕСТУВАННЯ ФУНКЦІОНАЛЬНОЇ ПОВЕДІНКИ *Актуальні проблеми комп'ютерних наук* (АПКН–2025), Хмельницький, 2025 (293).

15

Напрямки подальших досліджень

ML Класифікатор

Застосування Machine Learning для точного розрізнення типів помилок (презентаційні vs. семантичні).

Інтеграція з Фазингом

Поєднання з фазинг-генераторами (AFL, libFuzzer) для автоматичного пошуку вразливостей.

Domain-Specific Аналіз

Розробка компараторів для складних систем (ML-моделі, графіка, аудіо, UI).

16

Загальні висновки

- **Розроблено метод** оцінювання якості без еталонних оракулів.
- **Реалізовано систему** DiffQualytor з підтримкою Docker та архітектури плагіна.
- **Експериментально доведено** ефективність на JSON-парсерах та алгоритмах сортуванн
- **Досягнуто Recall 1.0** у виявленні розбіжностей.
- **Запропоновано метрику FSI** як універсальний індикатор якості для бізнесу.

17

Дякую за увагу!

18

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Мацюка Дмитра Віталійовича
факультет ІТ, 2 курс, група ІПЗм-24-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

01.09.25
дата

підпис

Sun Dec 14 18:24:00 EET 2025, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (UA) v-16.693

The maximum coincidence with one document 1.0%

Dictionaries check: UA, US, RU. **Errors in the documents: 23%**

	Document		Sum coincidence on the DB	
ID: 252781 Title: МКР_Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки Added in a DB: 2025-12-14 Authors: Дмитро МАЦЮК Heads: канд. техн. наук, доцент Consultants: Оксана ЯШИНА Opponents:	Symbols	Lexemes	Symbols	Lexemes
	107913	827	2011 (2%)	25 (3%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Дмитро МАЦЮК

Співавтор:

Назва: Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки

Експерт: канд. техн. наук, доцент Оксана ЯШИН

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:1.1%

Коефіцієнт подібності 2:0.4%

Мікропробіли: 13

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-12-14 19:25:03.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 14.12.2025 експерт

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач вищої освіти Мацюк Дмитро Віталійович

Тема «Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки»

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість сторінок записки 98 .

1.Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі з здійснено системний аналіз підходів до забезпечення якості ПЗ та методів тестування. Виявлено дослідницький розрив: 89% досліджень використовують диференціальне тестування виключно для бінарного виявлення дефектів. Розроблено новий метод оцінювання якості ПЗ через комплекс метрик (FSI, WDSI, DR, QPV) та чотирирівневу класифікацію розбіжностей (катастрофічні, семантичні, поведінкові, презентаційні). Програмно реалізовано інструмент DiffQualytor з автоматизацією процесу та інтеграцією в CI/CD як «ворота якості».

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота повністю відповідає поставленому завданню. Усі завдання, визначені у вступі, вирішені як у теоретичній частині (розробка концептуальної моделі, формалізація математичного апарату, розробка метрик якості), так і в практичній її частині (програмна реалізація методу, експериментальні дослідження на реальних проєктах).

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обґрунтовується актуальність теми, формулюються мета та завдання дослідження з акцентом на необхідність автоматизованих методів оцінювання якості в умовах CI/CD (76% організацій використовують ці практики). У першому розділі проведено ґрунтовний аналіз літературних джерел за період 2020-2024 років, що виявив зростання публікацій з диференціального тестування на 127% та критичний дослідницький розрив. У другому розділі розроблено теоретичні основи методу, концептуальну модель з чотирма модулями, формальний математичний апарат та алгоритм застосування. Запропоновано комплекс нових метрик (DR, WDSI, FSI, QPV) та проведено порівняльний аналіз з мутаційним тестуванням та PBT. У третьому розділі виконано проектування системи з аналізом функціональних та нефункціональних вимог, розроблено модульну архітектуру, обґрунтовано вибір сучасних технологій, спроектовано схему інтеграції в CI/CD. У четвертому розділі представлено програмну реалізацію системи DiffQualytor (Python 3.10), проведено експериментальні дослідження на трьох реальних проєктах з відкритим кодом, що підтвердили точність ідентифікації дефектів, швидкість виконання та ефективність локалізації дефектів. Усі розділи демонструють використання сучасних досягнень у галузях автоматизації тестування, безперервної інтеграції та Software Quality Metrics.

4. Позитивні сторони роботи Кваліфікаційна робота містить низку інноваційних рішень. Вперше запропоновано метод кількісного оцінювання якості ПЗ через систему метрик (FSI, WDSI, QPV) та чотирирівневу класифікацію розбіжностей з ваговими коефіцієнтами. Практична реалізація демонструє автоматизацію 95% процесу та високу точність виявлення дефектів. Експериментальні дослідження на реальних проєктах підтвердили скорочення часу

виявлення регресій з 4.2 днів до 6-8 годин (покращення на 85%) та зниження вартості виправлення дефектів. Робота має високий науковий рівень формалізації та глибину аналізу.

5. Негативні сторони роботи Експериментальні дослідження проведено на обмеженій кількості проєктів (три). Бажано розширити експериментальну базу для підвищення статистичної значущості. Відсутній аналіз випадків неефективності методу та формалізована методика визначення порогових значень метрик. У розділі архітектури недостатньо розглянуто питання масштабованості для великих промислових проєктів.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане на високому професійному рівні. Робота містить якісні діаграми (концептуальна модель методу, класифікація методів тестування, еволюція методологій розробки), які суттєво покращують сприйняття матеріалу. Використання таблиць для порівняльного аналізу (статичного та динамічного, порівняння фреймворків тестування) є доцільним та інформативним. Пояснювальна записка відповідає вимогам стандартів до оформлення магістерських робіт.

7. Відгук про роботу в цілому Кваліфікаційна робота магістра є завершеним науково-дослідним проектом, що вирішує актуальну проблему сучасної індустрії програмного забезпечення. Робота демонструє глибоке розуміння предметної області, здатність автора до критичного аналізу існуючих рішень та розробки оригінальних підходів. Поєднання ґрунтовної теоретичної бази (формалізація, математичний апарат) з практичною реалізацією та експериментальним підтвердженням свідчить про високий рівень наукової зрілості здобувача. Результати роботи мають як наукову новизну (перше застосування диференціального тестування для кількісного оцінювання якості через інтегральні метрики), так і практичну цінність (автоматизація процесу забезпечення якості в CI/CD з потенціалом економії мільярдів доларів у масштабах індустрії). Робота виконана на рівні, що відповідає вимогам до магістерських дисертацій і може бути рекомендована до захисту

8. Інші зауваження Рекомендується продовжити дослідження у напрямку інтеграції розробленого методу з генераторами тестів (фазерами) для повної автоматизації процесу та застосування машинного навчання для прогнозування регресій. Бажано розглянути можливість публікації основних результатів роботи у фахових виданнях та представлення на науково-практичних конференціях. Розроблений програмний інструментарій має потенціал для впровадження в реальні промислові проєкти та може бути запропонований до відкритого розповсюдження як open-source проєкт.

9. Оцінка кваліфікаційної роботи Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)
Глибчук Олександра Семенівна, д.т.н., професор, професор кафедри КІС

«12» грудня 2025 р. _____ (підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:
Назва: «Метод оцінювання якості програмного забезпечення на основі диференціального тестування функціональної поведінки»

Автор: Мацюк Дмитро Віталійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Яшина Оксана Миколаївна, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:
1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;
2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланя на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;
4) виявлені модифікації тексту не впливають на відсоток схожості.
Максимальний обсяг запозичень, визначений системою Anti-Plagitarism, складає 1.0 %. За системою StrickerPlagitarism коефіцієнт подібності складає 1,1 %, коефіцієнт подібності 2 складає 0.4%.

Дата 14.12

Керівник _____ О.М. Яшина

Гарант ОП _____ О.М. Яшина

Завідувач кафедри _____ Л.П. Бедратюк