https://doi.org/10.31891/2307-5732-2026-365-48
УДК 681.518:004.415.53

КОВТКО АНДРІЙ
Тернопільський національний технічний університет ім. І. Пулюя
https://orcid.org/0009-0003-0083-0514
e-mail: kovtko773@gmail.com
САВКІВ ВОЛОДИМИР
Тернопільський національний технічний університет ім. І. Пулюя
https://orcid.org/0000-0001-9141-4804
e-mail: v.b.savkiv@gmail.com

АРХІТЕКТУРА АВТОМАТИЗОВАНОЇ СИСТЕМИ ГЕНЕРАЦІЇ ТЕСТІВ

Розглянуто проблему автоматизації створення модульних тестів у сучасних програмних системах. Незважаючи на широке використання модульного тестування, розроблення ефективних тестових сценаріїв залишається трудомістким процесом, а високі показники покриття коду не завжди гарантують здатність тестового набору виявляти реальні дефекти програмної логіки. Одним із перспективних напрямів підвищення ефективності тестування є використання великих мовних моделей для автоматизованої генерації тестів.

Метою роботи є розроблення архітектури автоматизованої системи генерації модульних тестів, що поєднує можливості генеративного штучного інтелекту та мутаційного тестування як механізму оцінювання та вдосконалення тестового набору. Запропонована система передбачає інтеграцію результатів мутаційного аналізу у цикл автоматизованої генерації тестів з використанням великих мовних моделей, що забезпечує формування механізму зворотного зв'язку для покращення якості тестів. У роботі описано архітектуру запропонованої автоматизованої системи тестування програмного забезпечення, котра містить модулі генерації тестів, адаптації звітів мутаційного тестування, аналізу результатів та ітеративного вдосконалення тестового набору. Особливу увагу приділено проблемі уніфікації форматів звітів різних фреймворків мутаційного тестування для забезпечення їх подальшого використання у процесі формування запитів до мовних моделей.

Запропонований підхід створює передумови для підвищення ефективності автоматизованого тестування програмного забезпечення та може бути застосований у різних програмних середовищах і технологічних стеках.

Ключові слова: *модульне тестування, мутаційне тестування, автоматизована генерація тестів, великі мовні моделі.*

KOVTKO ANDRII, SAVKIV VOLODYMYR
Ternopil Ivan Puluj National Technical University

ARCHITECTURE OF AN AUTOMATED TEST GENERATION SYSTEM

The growing complexity of contemporary software systems necessitates more advanced approaches to ensuring their reliability and quality. Although unit testing remains a fundamental practice in software engineering, the development of effective test cases is still resource-intensive, while traditional metrics such as code coverage do not adequately reflect the fault-detection capability of test suites.

Recent advancements in large language models (LLMs) have enabled the automated generation of test cases; however, the consistency and effectiveness of such tests remain limited without additional validation mechanisms. In this context, the present study introduces an architecture for an automated unit test generation system that integrates generative artificial intelligence with mutation testing techniques.

Unlike conventional approaches, mutation testing is employed not only as an evaluation metric but also as a structured feedback mechanism within an iterative test generation cycle. Mutation analysis results, including information about surviving mutants, mutation operators, and code locations, are utilized to enrich LLM prompts and guide the refinement of test cases.

The proposed architecture incorporates modules for test generation, mutation testing integration, report normalization, and iterative improvement of the test suite. Special attention is given to the standardization of mutation testing reports generated by heterogeneous frameworks, enabling their consistent processing and reuse within the system.

The developed approach has the potential to enhance the effectiveness of automated testing processes and demonstrates applicability across diverse software environments and technological stacks.

Keywords: *unit testing, mutation testing, automated test generation, large language models.*

Стаття надійшла до редакції / Received 11.02.2026
Прийнята до друку / Accepted 11.03.2026
Опубліковано / Published 28.05.2026

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Зростання складності сучасних програмних систем суттєво підвищує вимоги до забезпечення їхньої якості та надійності. Одним із ключових інструментів контролю якості програмного забезпечення є модульне тестування, яке дозволяє виявляти дефекти на ранніх етапах розроблення та підвищує стабільність програмних систем. Водночас створення ефективних модульних тестів є трудомістким процесом, що потребує значних часових і людських ресурсів, а високі показники покриття коду тестами не завжди гарантують здатність тестового набору виявляти реальні дефекти програмної логіки.

У зв'язку з цим актуальним науковим і практичним завданням є пошук та розроблення методів і засобів автоматизації процесу створення якісних тестів та підвищення їхньої ефективності. Останніми роками для цієї мети активно використовуються великі мовні моделі, здатні генерувати тестові сценарії на основі аналізу програмного коду. Однак якість таких тестів часто є нестабільною і потребує додаткових механізмів оцінювання та вдосконалення. У цьому контексті перспективним напрямом досліджень є інтеграція мутаційного тестування

як інструмента аналізу ефективності тестів у процес автоматизованої генерації тестів, що дозволить сформувати механізм зворотного зв'язку для покращення результатів роботи інтелектуальних систем тестування.

Аналіз досліджень та публікацій

Тестування є одним із ключових етапів життєвого циклу розроблення програмного забезпечення. Воно передбачає перевірку відповідності програмного продукту визначеним вимогам та здатності системи формувати очікувані результати. Ефективна реалізація цього етапу є необхідною умовою забезпечення високої якості та надійності програмного продукту [1].

У міру зростання складності програмних систем [2] підвищуються і вимоги до процесів тестування, що зумовлює його перетворення на один із пріоритетних напрямів досліджень у галузі програмної інженерії [3]. Розвиток інноваційних підходів до тестування програмного забезпечення є важливим не лише для підвищення якості кінцевого продукту, але й для оптимізації використання ресурсів розроблення, оскільки, за різними оцінками, витрати на тестування можуть становити від 30% до 50% загального бюджету проєкту розроблення програмного забезпечення [4].

У роботі [5] проведено дослідження впливу автоматизації тестування на якість програмного забезпечення у відкритих проєктах, що використовують безперервну інтеграцію (CI). Автори проаналізували значну вибірку репозиторіїв з відкритим кодом та дослідили взаємозв'язок між рівнем автоматизації тестування, практиками CI та показниками якості програмного продукту. Результати дослідження свідчать, що підвищення рівня автоматизації тестування позитивно корелює зі зменшенням кількості дефектів і підвищенням стабільності програмного забезпечення.

У дослідженні також показано, що використання автоматизованих тестів у середовищах безперервної інтеграції сприяє швидшому виявленню помилок та підвищує ефективність процесу розробки. Автоматизоване тестування дозволяє регулярно перевіряти зміни у програмному коді, зменшує ризик появи регресійних дефектів та підтримує стабільність системи в умовах частих оновлень. Автори підкреслюють, що зрілість процесів автоматизації тестування є важливим фактором забезпечення якості програмного продукту та підвищення ефективності сучасних практик розроблення програмного забезпечення.

Сфера автоматизованого тестування перебуває в активному розвитку, зокрема завдяки залученню інструментів штучного інтелекту у подібних системах, які відкривають нові підходи до генерації тестів, виявлення дефектів і адаптивного управління тестовими стратегіями [6].

Модульне тестування є одним із найбільш поширених видів автоматизованого тестування та використовується у більшості сучасних програмних проєктів. Емпіричні дослідження показують, що модульне тестування є поширеною практикою у сучасній розробці програмного забезпечення, хоча рівень його використання може суттєво відрізнятися залежно від типу проєкту. Так, у масштабному дослідженні понад 50 000 open-source систем було встановлено, що значна частина проєктів використовує модульні тестові сценарії, причому більші та активніші проєкти частіше мають розвинену тестову інфраструктуру [7]. Опитування розробників також показують, що модульне тестування використовується більшістю програмістів як базова інженерна практика [8].

Попри численні переваги модульного тестування, у сучасних наукових дослідженнях виявлено низку обмежень та проблем, пов'язаних із його використанням. Дослідники зазначають, що ефективність модульного тестування значною мірою залежить від якості розроблених тестових сценаріїв та рівня підготовки розробників. Створення якісних модульних тестів є складним і ресурсоємним завданням, яке потребує глибокого розуміння логіки програмної системи та ретельного проєктування тестових сценаріїв. Зокрема, результати дослідження Microsoft Research показують, що розробка модульних тестів може займати від 20 % до 50 % часу, витраченого на створення основного функціоналу, а в окремих випадках — до 60 % загального часу розроблення програмного забезпечення [9].

Важливим аспектом, який активно досліджується, є проблема якості тестового коду. Недотримання рекомендованих практик проєктування тестів може призводити до появи так званих test smells – антипатернів тестування, що ускладнюють підтримку тестового набору та знижують ефективність виявлення дефектів. До найбільш поширених антипатернів належать, зокрема, надмірна кількість тверджень у тесті без чіткої структури (Assertion Roulette), відсутність перевірочних тверджень (Missing Assert), порожні тести (Empty Test), перевірка надто великої кількості функціональних елементів у межах одного тесту (Eager Test) та використання складної умовної логіки у тестових сценаріях [10].

Емпіричні дослідження показують, що антипатерни тестування мають тенденцію накопичуватися у процесі еволюції програмних систем. Наприклад, у роботі [11], де було проаналізовано вісім великих Java-проєктів з відкритим кодом, встановлено, що на кожен усунутий випадок тестового антипатерну з часом виникає в середньому два нових. Наявність таких антипатернів може призводити до хибного уявлення про якість тестового набору, зокрема через високі показники покриття коду, які не завжди відображають реальну здатність тестів виявляти дефекти. У поєднанні з орієнтацією на числові показники покриття коду та обмеженими часовими ресурсами це може призводити до зниження якості тестових наборів у реальних програмних проєктах [12].

Формулювання цілей статті

Мета даної роботи – розроблення архітектури автоматизованої системи генерації модульних тестів з використанням інструментів генеративного штучного інтелекту та мутаційного тестування як механізму оцінювання та вдосконалення тестового набору. Для подібної системи потрібно сформулювати основні технічні

вимоги та розробити загальні правила та принципи її функціонування. Запропонована автоматизована система генерації тестів має забезпечити інтеграцію результатів аналізу мутаційного тестування у цикл генерування тестових сценаріїв з використанням великих мовних моделей.

Для досягнення мети даної роботи розроблювана архітектура автоматизованої системи тестування програмного забезпечення повинна складатись з модулів генерації тестів, адаптації звітів мутаційного тестування, аналізу результатів та ітеративного вдосконалення тестового набору. Для досягнення цієї мети формати звітів різних фреймворків мутаційного тестування потрібно уніфікувати з метою їх подальшого використання у процесі формування запитів до мовних моделей.

Загальна ціль даного дослідження – це підвищення ефективності системи автоматизованого тестування програмного забезпечення та забезпечення можливостей її використання для різних програмних середовищ і технологічних стеків.

Виклад основного матеріалу

Попри стрімку еволюцію засобів автоматизованої генерації тестових сценаріїв, їхнє впровадження в цикли розробки програмного забезпечення (ПЗ) супроводжується низкою критичних обмежень. Емпіричні дослідження вказують на суттєву гетерогенність, неоднорідність якості згенерованих тестів. Ефективність автоматизованих інструментів тестування визначається сукупністю чинників, серед яких визначальними є архітектурна специфіка та типи модулів, що підлягають верифікації, рівень покриття існуючої кодової бази та загальний ступінь дотримання принципів «чистого коду» (clean code). Практичний досвід свідчить про поширеність випадків, коли тести, сформовані за допомогою штучного інтелекту (ШІ), демонструють формальну працездатність, проте не забезпечують коректної перевірки функцій та класів або очікуваної поведінки об’єктів. Це явище нівелює надійність автоматизованого процесу тестування та вимагає впровадження додаткових механізмів верифікації самих тестів.

Одним із найбільш релевантних методів об’єктивного оцінювання якості згенерованих тестів є мутаційне тестування [13]. Мутаційне тестування вважається «золотим стандартом» перевірки тестів, його впровадження не є масовим через кілька серйозних бар'єрів.

Основна причина обмеженого впровадження тестування на мутаціях є високі затрати ресурсів та часу. Мутаційне тестування вимагає витрату величезних обчислювальних ресурсів (Time Expense). Для прикладу, якщо є набір зі 100 тестів і інструмент створює 500 мутантів, системі доведеться запустити тести 50 000 разів, кожен тест проти кожного мутанта. На великих проектах це може розтягнути час збірки (CI/CD) з хвилин до годин або навіть днів.

Для впровадження мутаційного тестування існує проблема еквівалентних мутантів (Manual Effort), коли зміна коду не змінює його логіку, наприклад, заміна «i++» на «++i» у циклі for. Тести пройдуть без виявлення мутацій, бо кінцевий результат однаковий. Як наслідок, розробнику доведеться вручну переглядати сотні таких звітів, щоб зробити висновок, – тест неякісний чи мутація була «пустою». Це призводить до суттєвих витрат робочого часу.

Мутаційне тестування подеколи складно інтегрувати до існуючих тестових сценаріїв. Мутаційні фреймворки потребують глибокого налаштування під конкретний стек технологій, вони часто конфліктують із великими монолітами або складними залежностями в legacy-коді.

Обмеження на інтеграцію мутаційного тестування накладає так званий «шум» у звітах. Якщо в проекті вже є нестабільні тести (flaky tests), мутаційне тестування стає майже неможливим, бо результати будуть постійно хибними.

Але попри суттєві обмеження мутаційне тестування є обов’язковим до впровадження у проєктах, де ціна помилки критична, особливо це стосується наступних галузей і сфер застосування, – медичне обладнання та програмне забезпечення, авіакосмічна галузь, сфера фінансово технічних та банківських транзакцій, критичні open-source бібліотеки, якими користуються мільйони і.т.п.

Тестування на мутації – це потужний метод тестування на основі помилок, який використовується для оцінки якості набору тестів та тестових сценаріїв. Він передбачає внесення невеликих, навмисних змін до вихідного коду (Mutants), з метою перевірки, чи можуть існуючі тести їх виявити. Зазвичай процес мутаційного тестування складається з наступних основних етапів:

1. Генерація мутантів (Mutant Generation).

На цьому початковому етапі оригінальна програма модифікується для створення кількох версій, які називаються мутантами . Ці зміни виконуються за допомогою операторів мутації, які імітують поширені помилки програмування. Для прикладу, заміна арифметичних операторів (зміна «+» на «-»), заміна операторів логічних функцій (зміна && на ||), заміна реляційних операторів (зміна «>»на «>=»), видалення оператора, повне видалення рядка коду.

2. Виконання базового тесту (Baseline Test Execution).

Перед тестуванням мутантів оригінальна (немутована) програма запускається на існуючому наборі тестів. Це гарантує, що всі тести пройдуть на правильній версії коду. Якщо будь-які тести тут не пройдуть, набір тестів або код необхідно виправити, перш ніж продовжувати, оскільки мутаційне тестування передбачає успішне проходження базового рівня.

3. «Страта» мутантів (Mutant Execution).

Кожен згенерований мутант виконується для набору тестів. Мета полягає в тому, щоб хоча б один тестовий випадок завершився невдачею, що сигналізує про те, що набір тестів успішно виявив зміну в коді. Результат зведеться до двох тестових випадків:

- «Мутант убитий» – принаймні один тест не пройшов. Це бажаний результат.
- «Мутант вижив» – усі тести пройдено успішно, незважаючи на зміну коду. Це вказує на потенційну прогалину в охопленні набору тестів.

4. Ідентифікація еквівалентних мутантів (Identification of Equivalent Mutants).

Іноді мутація змінює синтаксис коду, але не його поведінку (наприклад, зміна i < 10 на i <= 9 у цілочисельному циклі). Такі мутанти називаються еквівалентними мутантами . Оскільки вони функціонально ідентичні оригінальній програмі, жоден тест ніколи не зможе їх знищити. Ідентифікація та видалення їх із загальної кількості часто є ручним або евристичним процесом.

5. Розрахунок балу мутації (Mutation Score Calculation).

Заключним етапом мутаційного тестування є розрахунок показника мутації (Mutation Score, MS), який відображає ефективність набору тестів. Чим вищий показник, тим надійніші тести.

Оцінка розраховується за формулою

$$MS = \left(\frac{K}{M - E} \right) \times 100\%,$$

де K – кількість убитих мутантів, M – загальна кількість згенерованих мутантів, E – кількість еквівалентних мутантів.

Концептуально, мутаційне тестування за рахунок семантичної модифікації вихідного коду шляхом внесення штучних дефектів – «мутацій» дозволить виконати наступні важливі процедури:

- Імітацію дефектів через впровадження операторів мутації, що моделюють типові помилки розробників.
- Оцінювання чутливості, якщо після внесення змін тестовий набір не фіксує збій у роботі програми («мутант виживає»), це є індикатором недостатньої здатності тестів виявляти помилки.
- Верифікацію логіки сценаріїв тестування, використання мутацій дозволяє ідентифікувати «поверхневі» тести, які забезпечують формальне покриття рядків коду (line coverage), але ігнорують перевірку граничних умов та логічних розгалужень.

Таким чином, мутаційне тестування дозволить ефективно виявляти неякісні або поверхневі тести, значно підвищить точність і глибину верифікації логіки програмних модулів. Використання мутаційного тестування на ранніх етапах життєвого циклу програмного забезпечення дозволить не лише своєчасно виявляти програмні дефекти, але суттєво покращить загальне тестове покриття.

Тому пропонується концептуальна архітектура інтегрованої системи генерації та вдосконалення модульних тестів автоматизованої системи, яка базується на синергії можливостей великих мовних моделей (LLM) та функціональних можливостей мутаційного аналізу. Процес оптимізації є ітеративним та охоплює наступні етапи:

1. Генеративна фаза. Використання LLM, натренованих на великих масивах синтаксичних структур та тестових шаблонів, для синтезу первинних модульних тестів. Це дозволяє охопити значний діапазон тестових сценаріїв, включно з тими, котрі залишаються поза увагою під час ручного тестування, мінімізуючи вплив людського фактора.
2. Фаза впровадження мутацій. Вихідний код піддається контрольованим змінам, мутаціям за допомогою спеціалізованих фреймворків для розрахунку мутаційного індексу. Це дозволить провести оцінювання здатності згенерованих тестів виявляти навмисно впроваджені дефекти.
3. Аналітична ітерація та рефакторинг. Результати виконання тестів на мутованому коді передаються назад до моделі ШІ. На основі отриманих даних модель здійснює коригування та доповнення тестових сценаріїв з метою підвищення їх ефективності.
4. Адаптивне, ітеративне навчання. Реалізація замкненого циклу зворотного зв'язку дозволяє системі накопичувати досвід щодо специфічних вразливостей конкретної архітектури, поступово підвищуючи ефективність генерації.
5. Термінація, зупинка процесу. Процедура оптимізації модульних тестів завершується при досягненні встановлених константних показників якості (наприклад, досягнення цільового рівня *Mutation Score*), коли подальші ітерації не демонструють статистично значущого покращення результатів.

Очікувані результати та практична цінність запропонованої загальної архітектури інтегрованої системи генерації та вдосконалення модульних тестів. Інтеграція генеративного ШІ з методиками мутаційного аналізу забезпечує досягнення наступних ключових науково-практичних результатів:

- Екстенсивне та інтенсивне покращення ефективності тестування. Формування набору тестів, що характеризуються високою здатністю до виявлення латентних дефектів та адекватною обробкою граничних сценаріїв. Такий підхід сприяє ретельнішій перевірці поведінки програмних модулів навіть у складних або непередбачуваних умовах.
- Оптимізація часових витрат. Автоматизація валідації якості тестів у поєднанні з ітеративним механізмом їх удосконалення суттєво знижує навантаження на інженерів з тестування (QA), вивільняючи ресурс для вирішення основних завдань, мінімізуючи витрати часу на аналіз і доопрацювання тестів.

− Підвищення якості та надійності ПЗ. Рання ідентифікація дефектів сприяє зниженню сукупної вартості програмного продукта (TCO) за рахунок зниження витрат на усунення помилок, які виявляються лише на пізніх стадіях. В загальному це сприяє покращенню загальної стабільності і надійності програмного забезпечення.

− Технологічна інваріантність та універсальність у застосуванні. Запропонована архітектура є універсальною та може бути адаптована до різних парадигм програмування та технологічних стеків. Автоматизована система генерації тестів розробляється з можливостями підтримки різних мов програмування та інструментів тестування що дозволить її застосування до широкого спектру проєктів – від невеликих бібліотек до масштабних розподілених систем.

Сучасний інструментарій мутаційного тестування охоплює більшість поширених мов програмування, проте існуючі фреймворки характеризуються суттєвою неоднорідністю, гетерогенністю механізмів генерації та фільтрації мутантів, а також відрізняються структурами вихідних звітів. Зокрема, фреймворк Stryker (для мов JavaScript/TypeScript) формує деталізовані звіти у форматах JSON та HTML, що містять статуси мутантів і посимвольне підсвічування модифікованих фрагментів вихідного коду. Натомість інструмент PIT (для Java) базується на іншій моделі представлення результатів, агрегуючи дані за класами, методами та типами мутантів у форматах XML та HTML із розрахунком таких метрик, як Line Coverage, Mutation Score та Test Strength. Отже, попри подібність семантичного наповнення звітів, розбіжності в їхній структурі створюють перешкоди для подальшої автоматизованої обробки даних.

З огляду на зазначену варіативність, уніфікований аналіз результатів мутаційного тестування потребує попередньої стандартизації даних. Ефективна інтеграція великих мовних моделей (LLM) у процеси генерації та вдосконалення тестів можлива лише за умови приведення різних форматів звітів до єдиного проміжного представлення (Intermediate Representation). Обов'язковими атрибутами уніфікованого формату визначено ідентифікатор мутанта, фрагменти вихідного та модифікованого коду, статус виконання, відомості про відповідні тестові сценарії або покриття, а також контекст програмного модуля. Стандартизація забезпечує узгоджену інтерпретацію метрик, мінімізує втрати інформації під час парсингу та оптимізує процес формування запитів (prompt engineering). У результаті, використання структурованих даних та результатів їх порівнянних дозволяє LLM достовірно аналізувати якість тестового покриття, ідентифікувати критичні вразливості та генерувати адаптовані тестові набори незалежно від первинного джерела даних.

З метою забезпечення масштабованості та архітектурної гнучкості розроблено модульну структуру автоматизованої системи генерації тестів із використанням інструментів великих мовних моделей (LLM), яка наведена на рис. 1.

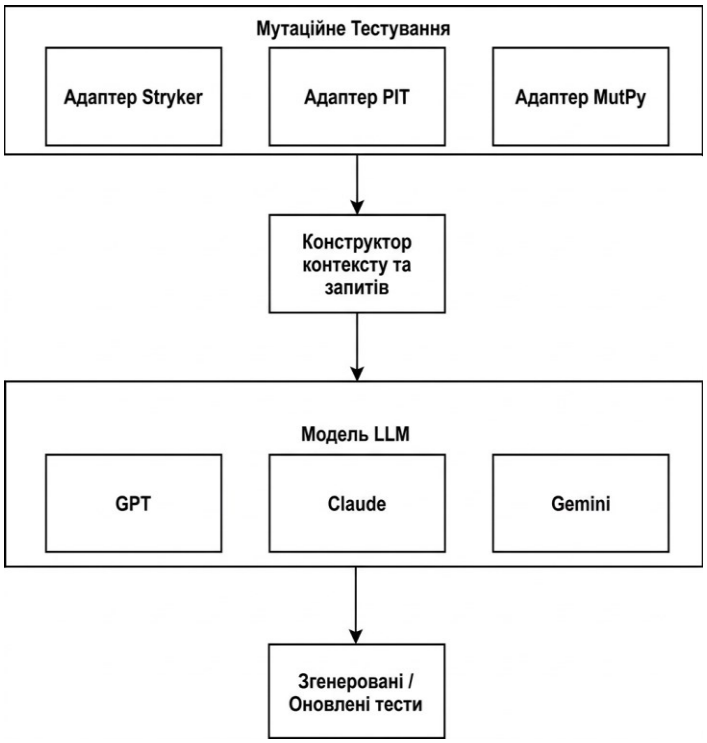

Рис. 1 Автоматизована система генерації модульних тестів

Модульна структура автоматизованої системи генерації тестів з використанням інструментів LLM містить наступні функціональні складові:

1. Модулі адаптерів, призначені для стандартизації звітів від різних фреймворків мутаційного тестування. Використання адаптерів дозволяє оперативно розширювати функціонал системи шляхом інтеграції нових інструментів тестування без зміни логіки основних компонентів.

2. Модуль формування запитів, котрий здійснює генерацію запитів до LLM на основі уніфікованих даних, отриманих з адаптерів і додаванням результатів аналізу контексту програмних модулів.

3. Блок взаємодії з LLM, що забезпечує опрацювання запитів із підтримкою декількох моделей залежно від специфіки поставленого завдання.

4. Контур ітеративного вдосконалення, який передбачає отримання оновленого набору тестів та їх передачу на початковий етап для верифікації та повторного оцінювання за допомогою мутаційного аналізу.

Запропонована архітектура та відповідна методика забезпечують можливість застосування різних фреймворків мутаційного тестування та моделей великих мовних моделей (LLM) із мінімальною модифікацією інших компонентів системи, що свідчить про її універсальність, адаптивність і придатність до масштабування.

Висновки з даного дослідження
і перспективи подальших розвідок у даному напрямі

У роботі проаналізовано роль модульного тестування як базового механізму забезпечення якості програмного забезпечення та окреслено його основні переваги й пов'язані з ним проблеми. Показано, що, попри високу ефективність на ранніх етапах розроблення, модульне тестування потребує значних ресурсів і є вразливим до появи так званих test smells, які можуть знижувати ефективність тестового набору.

Встановлено, що сучасні генеративні моделі штучного інтелекту здатні автоматизувати процес створення тестів, однак якість таких тестів залишається нестабільною. Для розв'язання цієї проблеми запропоновано модифікований підхід, що поєднує використання генеративного штучного інтелекту з мутаційним тестуванням. Запропоновано архітектуру автоматизованої системи генерації модульних тестів, яка включає етапи генерації тестів, проведення мутаційного аналізу та ітеративного вдосконалення тестового набору. Передбачається, що інтеграція зазначених підходів дає змогу виявляти слабкі місця у тестовому покритті, підвищувати глибину перевірки програмної логіки та зменшувати витрати на тестування, що робить запропоноване рішення перспективним для застосування у масштабних програмних проєктах. Для підтвердження цієї гіпотези необхідні подальші емпіричні дослідження після практичної реалізації описаної архітектури.

Подальші дослідження планується спрямувати на вивчення можливостей використання результатів мутаційного тестування як додаткового контексту для генерації тестів. Крім того, передбачається провести оцінювання сучасних великих мовних моделей з метою визначення найбільш придатних для задач генерації тестів, а також розглянути можливість створення спеціалізованої моделі, адаптованої до потреб запропонованої автоматизованої системи генерації модульних тестів.

Література

1. Myers G. J., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. Hoboken : John Wiley & Sons, 2011. 256 p.

2. Zhang M. Uncertainty in Software Development Projects: A Review of Recent Studies // Systems. 2025. Vol. 13, No. 8. Article 650. DOI: 10.3390/systems13080650.

3. Escalante-Viteri A., Mauricio D. Artificial Intelligence in Software Testing: A Systematic Review of a Decade of Evolution and Taxonomy // Algorithms. 2025. Vol. 18, No. 11. Article 717. DOI: 10.3390/a18110717

4. Zhou Z. Q., Sinaga A., Susilo W., Zhao L., Cai K. Y. A cost-effective software testing strategy employing online feedback information // Information Sciences. 2018. Vol. 422. P. 318–335. DOI: 10.1016/j.ins.2017.08.088

5. Wang Y., Mäntylä M. V., Liu Z., Markkula J. Test automation maturity improves product quality — Quantitative study of open-source projects using continuous integration // Journal of Systems and Software. 2022. Vol. 188. Article 111259. DOI: 10.1016/j.jss.2022.111259

6. Alshahwan N., Chheda J., Finogenova A., Gokkaya B., Harman M., Harper I., Marginean A., Sengupta S., Wang E. Automated unit test improvement using large language models at Meta // Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24). 2024. P. 185–196.

7. Kochhar P. S., Bissyandé T. F., Lo D., Jiang L.Adoption of Software Testing in Open Source Projects: A Preliminary Study on 50,000 Projects // Proceedings of the European Conference on Software Maintenance and Reengineering. 2013.DOI: 10.1109/CSMR.2013.48

8. Aniche M., Treude C., Zaidman A. How developers engineer test cases: An observational study // IEEE Transactions on Software Engineering. 2022. Vol. 48, No. 12. P. 4952–4969. DOI: 10.1109/TSE.2021.3064912

9. Williams L., Kudrjavets G., Nagappan N. On the effectiveness of unit test automation at Microsoft // Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE 2009). 2009. P. 81–89.

10. Panichella A., Panichella S., Fraser G., Zaidman A., van Deursen A., Di Penta M. Test smells 20 years later: Detectability, validity and reliability // Empirical Software Engineering. 2022. Vol. 27. Article 170. DOI: 10.1007/s10664-022-10207-5

11. Kim D.J. An empirical study on the evolution of test smell // Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020). 2020. P. 3.

12. Inozemtseva L., Holmes R. Coverage is not strongly correlated with test suite effectiveness // Proceedings of the 36th International Conference on Software Engineering. 2014. DOI: 10.1145/2568225.2568271.

13. Jia Y., Harman M. An analysis and survey of the development of mutation testing // IEEE Transactions on Software Engineering. 2011. Vol. 37, No. 5. P. 649–678.

References

1. Myers G. J., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. Hoboken : John Wiley & Sons, 2011. 256 p.

2. Zhang M. Uncertainty in Software Development Projects: A Review of Recent Studies // Systems. 2025. Vol. 13, No. 8. Article 650. DOI: 10.3390/systems13080650.

3. Escalante-Viteri A., Mauricio D. Artificial Intelligence in Software Testing: A Systematic Review of a Decade of Evolution and Taxonomy // Algorithms. 2025. Vol. 18, No. 11. Article 717. DOI: 10.3390/a18110717

4. Zhou Z. Q., Sinaga A., Susilo W., Zhao L., Cai K. Y. A cost-effective software testing strategy employing online feedback information // Information Sciences. 2018. Vol. 422. P. 318–335. DOI: 10.1016/j.ins.2017.08.088

5. Wang Y., Mäntylä M. V., Liu Z., Markkula J. Test automation maturity improves product quality — Quantitative study of open-source projects using continuous integration // Journal of Systems and Software. 2022. Vol. 188. Article 111259. DOI: 10.1016/j.jss.2022.111259

6. Alshahwan N., Chheda J., Finogenova A., Gokkaya B., Harman M., Harper I., Marginean A., Sengupta S., Wang E. Automated unit test improvement using large language models at Meta // Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24). 2024. P. 185–196.

7. Kochhar P. S., Bissyandé T. F., Lo D., Jiang L.Adoption of Software Testing in Open Source Projects: A Preliminary Study on 50,000 Projects // Proceedings of the European Conference on Software Maintenance and Reengineering. 2013.DOI: 10.1109/CSMR.2013.48

8. Aniche M., Treude C., Zaidman A. How developers engineer test cases: An observational study // IEEE Transactions on Software Engineering. 2022. Vol. 48, No. 12. P. 4952–4969. DOI: 10.1109/TSE.2021.3064912

9. Williams L., Kudrjavets G., Nagappan N. On the effectiveness of unit test automation at Microsoft // Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE 2009). 2009. P. 81–89.

10. Panichella A., Panichella S., Fraser G., Zaidman A., van Deursen A., Di Penta M. Test smells 20 years later: Detectability, validity and reliability // Empirical Software Engineering. 2022. Vol. 27. Article 170. DOI: 10.1007/s10664-022-10207-5

11. Kim D.J. An empirical study on the evolution of test smell // Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020). 2020. P. 3.

12. Inozemtseva L., Holmes R. Coverage is not strongly correlated with test suite effectiveness // Proceedings of the 36th International Conference on Software Engineering. 2014. DOI: 10.1145/2568225.2568271.

13. Jia Y., Harman M. An analysis and survey of the development of mutation testing // IEEE Transactions on Software Engineering. 2011. Vol. 37, No. 5. P. 649–678.