Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи тестування програмного забезпечення на базі платформи Android

(тема)

Виконав:

студент II курсу, групи КСМм-23-1

Семко В.В.
(прізвище, ініціали)

Спеціальність

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Комп’ютерні системи та мережі
(повна назва освітньої програми)

Керівник: ас. Кравченко П. О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерні системи та мережі
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Семку Владиславу Вікторовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи тестування програмного забезпечення на базі платформи Android

затверджена наказом по університету від “ 22 ” листопада 2024 р. № 1237 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20 січня 2025 р.

3. Вхідні дані до роботи

Методи тестування програмного забезпечення;
Модульне тестування; Інтеграційне тестування; Функціональне тестування;
Платформа Android;
Загальна модель тестування;
Програмне середовище PyCharm;
Мова програмування Python.

4. Перелік питань, що потрібно опрацювати у роботі

Вступ.
1. Аналіз предметної області та постановка задач дослідження.
2. Методи тестування програмного забезпечення на базі платформи Android.
3. Реалізація та дослідження методів тестування програмного забезпечення

Висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Демонстраційні матеріали. Плакати – 13 арк.

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз завдання та пошук літератури	26.11.2024-01.12.2024	
2	Огляд існуючих методів тестування	02.12.2024-10.12.2024	
3	Аналіз технічних засобів для реалізації	11.12.2024-20.12.2024	
4	Програмна реалізація досліджуваних методів	21.12.2024-26.12.2024	
5	Опрацювання результатів дослідження	27.12.2024-31.12.2024	
6	Оформлення ПЗ	01.01.2025-10.01.2025	

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ ас. Кравченко П. О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 78 с., 21 рис., 1 табл., 1 дод., 13 джерел.

МЕТОДИ ТЕСТУВАННЯ, ПЛАТФОРМА ANDROID, ТИПОВА МОДЕЛЬ ТЕСТУВАННЯ, ПРОЦЕС ТЕСТУВАННЯ, МОДУЛЬНЕ ТЕСТУВАННЯ, ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ, ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ.

Об’єктом дослідження є процес тестування Android застосунків.

Предметом дослідження є методи тестування програмного забезпечення на базі платформи Android.

Метою кваліфікаційної роботи є дослідження існуючих методів тестування програмного забезпечення на базі платформи Android, проведення порівняльного аналізу та надання рекомендацій щодо ефективного застосування досліджуваних методів в залежності від специфіки проектів.

У ході виконання кваліфікаційної роботи було проведено дослідження методів тестування програмного забезпечення на базі платформи Android. Було проаналізовано наступні методи тестування: метод модульного тестування, метод інтеграційного тестування та метод функціонального тестування. В ході дослідження було розглянуто їх математичні моделі, а також основні переваги та недоліки обраних методів.

У результаті проведеного дослідження було наведено порівняльний аналіз обраних методів тестування за параметром покриття коду та часом виконання та наведені рекомендації щодо їх використання.

ABSTRACT

Master’s thesis: 78 pages, 21 figures, 1 tables, 1 appendices, 13 sources.

TESTING METHODS, ANDROID PLATFORM, TYPICAL TESTING MODEL, TESTING PROCESS, MODULE TESTING, INTEGRATION TESTING, FUNCTIONAL TESTING.

The object of the study is the process of testing Android applications.

The subject of the study is software testing methods for the Android platform.

The purpose of the qualification work is to study existing software testing methods for the Android platform, conduct a comparative analysis and provide recommendations for the effective application of the studied methods depending on the specifics of the projects.

During the qualification work, a study of software testing methods for the Android platform was conducted. The following testing methods were analyzed: the module testing method, the integration testing method and the functional testing method. During the study, their mathematical models were considered, as well as the main advantages and disadvantages of the selected methods.

As a result of the research, a comparative analysis of the selected testing methods was provided by the code coverage parameter and execution time, and recommendations for their use were provided.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

API – прикладний програмний інтерфейс (англ., Application Programmong Interface)

HTTP – протокол передачі гіпертексту (англ., Hypertext Transfer Protocol)

IBM – американська електронна корпорація (англ., International Business Machines)

SDLC – життєвий цикл розробки програмного забезпечення (англ., Software Development Life Cycle)

TDD – розробка керована тестуванням (англ., Test Driven Development)

UI – інтерфейс користувача (англ., User Interface)

ВСТУП

В наш час важко уявити людину, яка б не користувалася смартфоном. Вони стали невід'ємною частиною нашого життя завдяки великій кількості функцій, які вміщають в собі. В сучасному смартфоні є все, що потрібно в повсякденному житті: ви можете проводити своє дозвілля переглядаючи фільми чи граючи в ігри, проводити тренування завдяки фітнес-застосункам, чи працювати з телефону в будь-якій точці світу, де є доступ до інтернету.

Проте, на жаль, багато з нас зіштовхувались з великою кількістю помилок під час користування телефоном: від невеликих помилок в інтерфейсі чи роботі застосунків, до непередбачуваного закриття програм, в результаті чого користувачі можуть втрачати дані. На сьогоднішній день близько 70% смартфонів працюють на платформі Android, яка є найпопулярнішою у світі операційною системою з понад 2,5 мільярдами активних користувачів у багатьох країнах світу. Вона була розроблена як альтернатива з відкритим існуючим тоді iPhone і Palm OS, та стала найпопулярнішою мобільною операційною системою на початку 2010-х років.

З великою популярністю виникає і багато проблем. Напевно кожен з нас стикався із низькою швидкістю роботи чи поганою сумісністю застосунків на різних версіях смартфону. Фрагментація пристроїв в Android стосується великої кількості телефонів із різними розмірами екрана, роздільною здатністю, апаратними комбінаціями та версіями програмного забезпечення. Це створює багато труднощів для розробників, які повинні перевірити свої програми на сумісність з великою кількістю пристроїв, кожен з яких має свої особливості, що в свою чергу має велику трудомісткість. Дана проблема може сильно вплинути на взаємодію користувачів з програмами. Якщо застосунок погано оптимізовано для певного пристрою, то вона може

працювати неналежним чином: будуть з’являтись помилки в її роботі, баги інтерфейсу та багато іншого.

Зважаючи на це, якість програмного забезпечення, яке розробляється для цієї платформи відіграє важливу роль в її функціонуванні. Ефективне тестування Android-застосунків дозволяє забезпечити високу стабільність, продуктивність та безпеку, що, в свою чергу, підвищує рівень задоволення користувачів та зменшує ризики [1]. В умовах постійного оновлення програмного забезпечення, його тестування стає складнішим процесом, який вимагає використання сучасних методів як мануального, так і автоматизованого тестування, кожен з яких має свої особливості. Окрім стандартних методів тестування, в середовищі Android широкої популярності набрали спеціалізовані інструменти та фреймворки для автоматизації тестування програмного забезпечення, які можуть використовуватись в залежності від потреб розробника.

Метою даної роботи є аналіз основних методів тестування програмного забезпечення на базі платформи Android, дослідження популярних інструментів автоматизації тестування та надання рекомендацій щодо вибору методів тестування в залежності від конкретної специфіки застосунків, що розроблятимуться. В ході дослідження буде розглянуто основні методи тестування програмного забезпечення, описані їх переваги та недоліки, а також визначено в яких випадках кожен з методів буде більш ефективним.

В результаті дослідження буде створено рекомендації щодо оптимальних методів тестування залежно від особливостей Android-застосунку, що покращить якість та надійність програмного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ ТА ПОСТАНОВКА ЗАДАЧ

Android – одна з найпопулярніших операційних систем у світі, що займає перше місце на ринку мобільних пристроїв [2]. Завдяки відкритому вихідному коду і широкій підтримці з боку виробників смартфонів, таких як Samsung, Xiaomi, Huawei та багатьох інших, Android має більше 70% ринку мобільних операційних систем (рисунок 1.1). Його популярність обумовлена також гнучкістю, багатством функцій і великою кількістю застосунків.

Рисунок 1.1 – Частка ринку мобільних операційних систем у всьому світі

Тестування Android застосунків відіграє дуже важливу роль під час їх розробки. Сьогодні Android 14.0 є найновішою та найпопулярнішою версією, але все ще використовуються застарілі версії, такі як 4.4 і 5.1. На жаль, вони можуть не підтримувати деякі мобільні функції та програми. Старіші пристрої також мають різну роздільну здатність екрана, що вимагає категоричного проектування певних функцій.

Сьогодні існує багато методів тестування, які можна класифікувати за різними ознаками, такими як:

- за рівнем тестування: дана класифікація стосується етапів розробки, на яких виконують тестування;
- за ступенем автоматизації: залежить від того, чи використовується автоматизація під час тестування;

- за методами доступу до коду: залежить від того, чи має тестувальник доступ до коду;

- за підходом до тестування;

- за цілями тестування: класифікується за тим, що саме перевіряється під час тестування.

В залежності від класифікації методів можна зрозуміти, які підходи є найбільш доцільними для певних етапів розробки програмного забезпечення, а застосування різних методів для всебічного тестування застосунку дозволяє виявити більшість помилок перед випуском продукту, завдяки чому забезпечується краща якість розроблюваних застосунків.

Тестування має важливе значення для забезпечення сумісності застосунків у різноманітних екосистемах пристроїв і для того, щоб користувачі могли максимально використовувати застосунок. Воно гарантує, що застосунок функціонує належним чином на різних пристроях і операційних системах.

Нижче наведено кілька причин, чому тестування Android є таким важливим:

- гарантія якості: тестування гарантує якість програми, виявляючи помилки та дефекти на ранніх стадіях процесу розробки, таким чином зменшуючи витрати та час, необхідні для їх подальшого усунення;

- взаємодія з користувачем: тестування гарантує, що програма є інтуїтивно зрозумілою, функціональною та пропонує чудову взаємодію з користувачем;

- сумісність: тестування Android гарантує сумісність програми з різними пристроями та операційними системами, що розширює потенційну аудиторію програми та базу користувачів;

- безпека: тестування допомагає виявити недоліки безпеки та проблеми, які можуть поставити під загрозу дані споживачів.

Тестування гарантує, що програма відповідає очікуванням користувачів щодо якості та продуктивності. Таким чином, тестування є

життєво важливим для розробки якісних, безпечних і функціональних застосунків, які забезпечують чудовий досвід роботи з різними пристроями та операційними системами.

1.1 Основні принципи та етапи тестування застосунків

Тестування – це процес перевірки чи оцінки системи або її компонентів за допомогою ручних або автоматизованих засобів для перевірки того, чи задовольняють вони заданим вимогам. Основною метою тестування є переконатися, що програмний продукт працює так як повинен та забезпечує позитивний користувацький досвід. Зазвичай тестування займає близько половини часу і роботи, необхідних для створення функціональної програми.

Хоча тестування на перший погляд і здається зрозумілим по своїй меті, воно пройшло велику еволюцію підходів через призму змін у «психічному житті» тестувальників. Загалом, виокремлюють 5 основних фаз розвитку тестування починаючи з 1950-х років [3]:

- фаза 0 (до 1956 року: орієнтована на налагодження): тестування не відрізнялося від налагодження, і не вважалося окремою дисципліною;
- фаза 1 (1957-1978 роки: орієнтована на демонстрацію): тестування орієнтоване на демонстрацію того, що програма працює, проте такий підхід виявився не ефективним, бо з часом знаходив нові помилки;
- фаза 2 (1979-1982 роки: орієнтована на руйнування): тестування було спрямоване на те, щоб показати, що програма не працює, що теж виявилось неефективним, оскільки в такому випадку програма може бути ніколи не випущена, бо завжди можна знайти нову помилку;
- фаза 3 (1983-1987 роки: орієнтована на оцінку): тестування виконувалося для оцінки ризиків. Метою такого тестування було зниження ризику непрацездатності до допустимого рівня, а не доведення програми до повної безпомилковості;

- фаза 4 (з 1988 року по наш час: орієнтована на профілактику): тепер тестування використовується для знаходження найбільш проблемних частин коду та мінімізації трудовитрат на тестування.

Кожен процес тестування, як і обраний метод залежить від конкретно вибраної ситуації та специфікації застосунку, що тестується. Проте, в загальному вигляді можна виділити наступні етапи, які будуть виконуватись під час проведення тестів (рисунок 1.2) [4]:

- аналіз вимог: на даному етапі тестувальник вивчає вимоги до продукту, що розробляється, досліджує їх на логічність та однозначність, знаходить слабкі місця в тестах та виявляє можливі ризики під час тестування;

- планування тестування: на даному етапі створюється стратегія тестування та визначаються інструменти та підходи, що будуть використовуватися;

- розробка тестових скриптів: створюються конкретні сценарії тестування або набори дій, за допомогою яких тестувальник перевірятиме різні функції застосунку;

- виконання тестування: відбувається запуск тестів згідно з сценаріями, розробленими в 3 пункті, а також відбувається фіксація результатів;

- аналіз результатів: проводиться оцінка відповідності результатів тестування до заданих вимог, виявлені помилки документуються, класифікуються та передаються розробникам для усунення;

- повторне тестування: для перевірки виправлення дефектів відбувається повторне тестування, в результаті якого формується звіт.

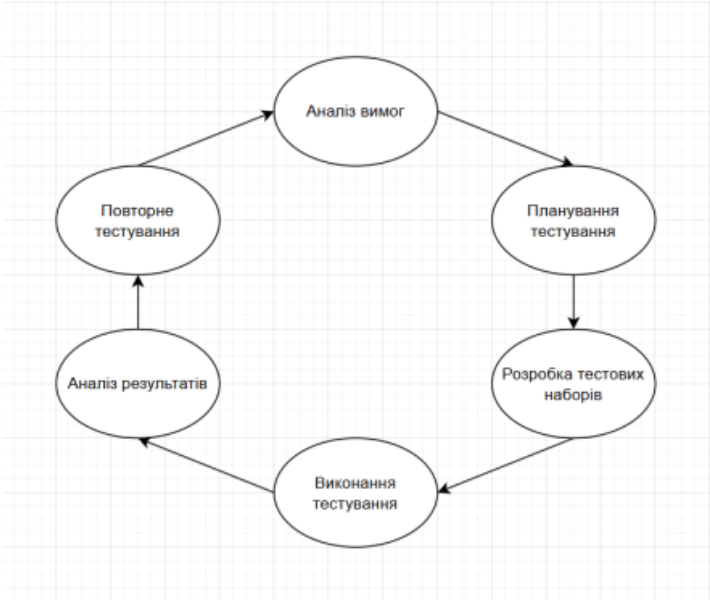

Рисунок 1.2 – Життєвий цикл тестування

Для успішного тестування любого застосунку потрібно також дотримуватись та усвідомлювати основні принципи тестування [5]. Принципи тестування – це основні підходи, завдяки яким кожне тестування може бути проведене більш ефективно. Їх дотримання є важливим на будь-якому етапі життєвого циклу ПЗ. На сьогодні можна виокремити сім принципів тестування:

- тестування показує наявність дефектів, а не їх відсутність: суть цього принципу полягає в тому, що навіть якщо всі проведені тести були успішні, це не є гарантією того, що в програмі відсутні помилки;

- вичерпне тестування неможливе: даний принцип можна вважати логічним продовженням першого. Неможливо провести тестування, яке б перевірило всі можливі комбінації сценаріїв. Замість того, щоб перевіряти всі можливі варіанти використання, проводиться аналіз ризиків та розстановка приорітетів;

- раннє тестування економить гроші та час: тестування повинне проводитись якомога раніше в життєвому циклі розробки ПЗ. Компанія IBM провела дослідження, яке показало: те, що коштує один долар, щоб

виправити на етапі проєктування, може коштувати п’ятнадцять на етапі тестування та сотню, якщо це буде виявлено на продакшені (рисунок 1.3);

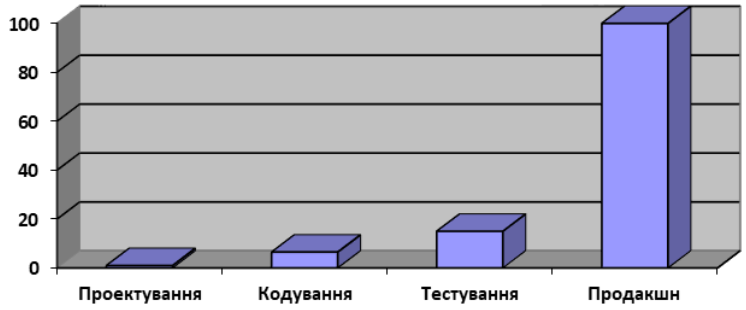

Рисунок 1.3 – Відносна вартість усунення дефектів

- скупчення дефектів: даний принцип стверджує, що більшість дефектів зазвичай зосереджена в невеликій частині коду. Його також можна назвати принципом 80%/20%: 80% дефектів зазвичай знаходять в 20% коду. Принцип скупчення дефектів є важливим інструментом для оптимізації процесу тестування. Застосування цього принципу дозволяє ефективніше виявляти дефекти і підвищувати загальну якість програмного продукту;

- парадокс пестициду: даний принцип бере свою назву з сільського господарства та описує ситуацію, коли повторне використання одних і тих самих тестів стає менш ефективним. Зазвичай він виникає через старіння існуючих тестів або еволюцію дефектів. Регулярний аналіз і оновлення тестових випадків є ключовими факторами для успішної боротьби з даною проблемою;

- тестування залежить від контексту: даний принцип вказує нам, що не існує універсального підходу і кожне тестування проводиться по-різному, враховуючи всі нюанси контексту;

- відсутність помилок оманлива: іноді програмне забезпечення, яке на етапі тестування не показало помилок, може бути непридатним для його використання. Зазвичай таке відбувається, коли було неправильно визначені вимоги до тестування, або просто не зручне для користування.

1.2 Огляд інструментів для тестування Android застосунків

На сьогодні існує великий спектр інструментів для тестування, які допомагають автоматизувати процеси перевірки та допомагають знаходити помилки під час розробки застосунку. Вибір правильної програми для виконання тестових сценаріїв залежить від вимог проекту, зручності для розробника та тестувальника, підтримки різних мов програмування, можливості швидко створювати звіти про тестування та багато інших особливостей. Гарно підібраний інструмент для тестування може знизити витрати на розробку в довгостроковій перспективі, зекономити час та автоматизувати виконання тестів там, де це можливо, а також уникнути основних проблем тестування, таких як фрагментація пристроїв та різні мобільні інтерфейси [2].

Одним з найбільш відомих інструментів для тестування Android застосунків є Appium. Даний інструмент кросплатформенний, може використовуватись як для iOS, так і Android платформ, що робить його універсальним рішенням. Appium працює на базі Selenium WebDriver, що забезпечує йому підтримку багатьох мов програмування, таких як Java, C#, Python та інші. Це полегшує інтеграцію з існуючими фреймворками та дає можливість автоматизації тестів мовами, якими володіють розробники. Ще однією особливістю цього інструменту є відсутність модифікації вихідного коду, що значно спрощує процес автоматизації тестування.

Appium працює по принципу клієнт-серверної архітектури. Основою інструменту є сервер, який розгортається локально чи в хмарі, а клієнти надсилають команди на даний сервер. В ролі клієнта виступають клієнтські бібліотеки, які створюються на різних мовах програмування. Якщо потрібно створити нову тестову платформу на одній з доступних мов програмування, достатньо загрузити клієнтські бібліотеки на даній мову програмування. В ролі сервера виступає HTTP-сервер, куди відправляються всі запити. Сервер

створено на базі Node JS, і для його ініціалізації потрібне попереднє встановлення.

Знаючи архітектуру даного інструменту, можна описати принцип його роботи:

- встановлюється Appium та налаштовується сервер REST API;
- створюється структура використовуючи клієнтські бібліотеки на необхідній мові;
- відправляється запит на сервер, який передає його на мобільний застосунок, де виконується відповідна дія;
- зворотній зв’язок надходить у форматі HTTP відповідей (для цього сервер використовує фреймворк чи платформу автоматизації).

Appium має ряд переваг, які можуть спонукати розробника до вибору саме цього інструменту тестування, зокрема:

- стандартний API для всіх платформ: для тестування Android та iOS працює єдиний API, який можна налаштовувати під себе;
- гнучкість та кросплатформеність: він працює на різних платформах, таких як Windows, Mac, Linux, його можна використовувати для тестування застосунків на iOS та Android;
- підтримка багатьох мов програмування;
- вільне використання тестового простору: завдяки використанню даного інструменту тестувальник може вибрати будь-який фреймворк та впровадити його в поточний проект.

Проте, не зважаючи на ряд переваг, він має і певні недоліки:

- не підходить для роботи з тестовою платформою Android версії 4.2 і нижче;
- не найкращий для тестування гібридних застосунків, оскільки можуть виникнути проблеми при переключенні з веб-застосунку в нативний та навпаки;

- погана продуктивність при тестування застосунків з великою кількістю зображень, оскільки функціонал по пошуку і розпізнаванню образів обмежений.

Ще одним популярним інструментом тестування є Espresso. Розроблений компанією Google та інтегрований у Android Studio, використовується для UI-тестування нативних застосунків. Основним призначенням Espresso є швидке, безпечне та стабільне тестування взаємодії користувача з елементами інтерфейсу Android-застосунків.

Завдяки легкому синтаксису даний інструмент максимально доступний для розробників та тестувальників з різним рівнем досвіду. Espresso спеціалізується на автоматизації тестування UI-інтерфейсу та автоматично виконує тести синхронізуючись з головним потоком UI-застосунку. Завдяки такій синхронізації, фреймворк самостійно чекає, поки елемент з’явиться чи зникне на екрані, перш ніж продовжувати тестування, що допомагає уникнути затримок пов’язаних з так званою «гонкою» за ресурсами. Espresso є частиною екосистеми Android, завдяки чому легко інтегрується в Android Studio і працює без сторонніх бібліотек. Це забезпечує хорошу зручність для розробників, оскільки вони можуть писати і запускати тести прямо в середовищі розробки.

Архітектура Espresso побудована на концепції «модель, дії і твердження» (Model-Action-Assertion). Кожен тестовий сценарій має три основні компоненти: ViewMatchers знаходить елементи інтерфейсу, ViewActions описує дії, які потрібно виконати з елементом, ViewAssertions дозволяє перевірити результат дії.

Основними перевагами Espresso можна виділити наступне:

- швидке тестування: завдяки інтеграції з AndroidStudio даний фреймворк працює швидше ніж більшість інших фрейморків для UI-тестування;

- надійність: завдяки автоматичній синхронізації з UI-потоком тести виконуються стабільно;

- мінімалістичність: зрозумілий та мінімалістичний синтаксис спрощує написання тестів;

- безпека та стабільність: оскільки він працює в межах процесу застосунку, тестування стає більш захищеним та стабільним.

Espresso має і ряд недоліків:

- обмеженість у використанні: працює лише з нативними застосунками на Android та не підтримує кроспратформеність;

- залежність від коду: не може працювати з функціями, які вимагають взаємодії з різними процесами.

1.3 Загальна класифікація методів тестування

Методи тестування можна класифікувати за багатьма критеріями, такими як доступ до коду, об’єктом тестування чи підходом до тестування. На сьогодні існує велика кількість методів, кожен з яких має свої особливості та краще всього підходить для вирішення певних завдань.

Наприклад, один метод краще підходить для тестування окремих модулів в застосунку, а інший – для перевірки взаємодії модулів між собою. Під час вибору методу тестування важливу роль відіграють такі аспекти, як об’єкт тестування, його технічні характеристики та специфічні особливості, і в залежності від цих факторів обирається найбільш відповідний метод чи їх комбінація.

На сьогодні одна з найпоширеніших класифікацій методів тестування складається з 3 категорій (рисунок 1.4) [6]:

- за ступенем автоматизації;
- за доступом до коду;
- за цілями тестування.

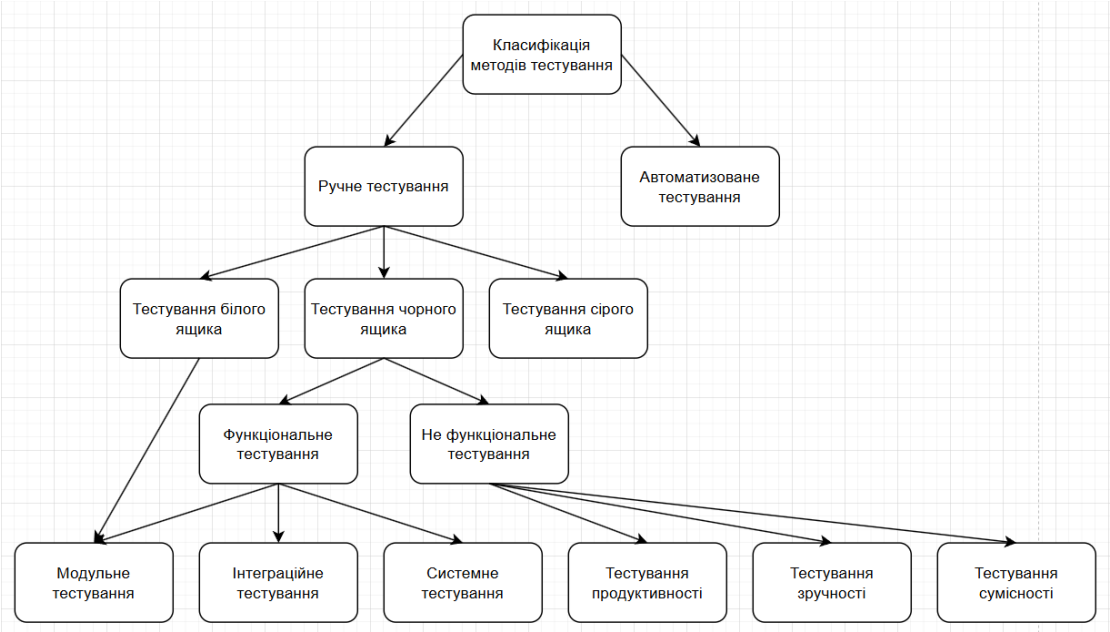

Рисунок 1.4 – Загальна класифікація методів тестування

Дана класифікація є зручною для розуміння, однак вона не охоплює всі можливі методи тестування. Постійна еволюція програмного забезпечення з кожним роком створює нові методи тестування, а специфіка проектів може вимагати нестандартних методів тестування, які не підходять до стандартної класифікації. Також варто зазначити, що кожен з методів можна розділити на безліч підтипів в залежності від рівня абстракції та комбінації методів.

1.3.1 Класифікація за ступенем автоматизації

На сьогодні існує два різних типи тестування програмного забезпечення за типом виконання, які використовуються в галузі та мають свої переваги та недоліки.

Ручне тестування – це техніка тестування програмного забезпечення, яка виконується тестувальниками без використання автоматизованих інструментів чи скриптів. Під час ручного тестування спеціаліст виконує тестові сценарії вручну, дотримуючись набору попередньо визначених тестів, щоб знайти помилки, оцінити зручність користування та інші аспекти програми. В даному методі тестери створюють тестові випадки для кодів,

тестують програмне забезпечення та дають остаточний звіт про це програмне забезпечення. Ручне тестування – це самий низькорівневий та простий тип тестування, який не вимагає великої кількості знань, проте він займає багато часу, оскільки воно виконується людьми, і існує ймовірність людських помилок.

Основними перевагами ручного тестування є:

- швидкий зворотній зв’язок: найчастіше використовується для тестування графічного інтерфейсу, де виявляє велику кількість помилок;
- дешевизна: не вимагає складних навичок чи певних інструментів;
- відсутність коду: для даного типу тестування не потрібно знати програмування, тому він легкий для нових тестувальників;
- ефективний під час змін: ручне тестування дуже зручне, коли вносяться незаплановані зміни в програму, бо воно легко адаптується.

Автоматизоване тестування – це техніка тестування, під час якої тестувальник розробляє тестові сценарії та використовує спеціалізоване програмне забезпечення або інструменти автоматизації для тестування застосунків. Даний підхід дозволяє значно прискорити процес тестування, особливо коли він застосовується для повторюваних процесів. Автоматизація тестування є найбільш ефективною, коли потрібно виконати регресійне тестування (перевірку стабільності системи після внесення змін) та перевірити функціональність застосунку протягом його життєвого циклу.

Основними перевагами автоматизованого тестування є:

- спрощення виконання тестів: автоматизоване тестування можна залишати без нагляду, що дозволяє контролювати тестування лише в кінці процесу. Це спрощується процес загального тестування;
- підвищує надійність тестів: даний метод працює по заздалегідь розробленому алгоритму, завдяки чому гарантує рівномірне тестування всієї програми, завдяки чому забезпечує кращу якість кінцевого продукту;
- більший обсяг покриття: за допомогою автоматизації можна створювати та перевіряти більшу кількість тестів. В результаті розробник

отримує більше охоплення тестуванням і знаходить більшу кількість помилок;

- мінімізація людської взаємодії: завдяки автоматизації всього процесу тестування зменшується до мінімуму фактор людської помилки.

1.3.2 Класифікація за доступом до коду

Класифікація за доступом до коду поділяється на три категорія: тестування білого, чорного та сірого ящика.

Тестування білого ящика – це метод тестування програмного забезпечення, яка передбачає тестування внутрішньої структури застосунку. В даному випадку тестер має доступ до вихідного коду і використовує ці знання для розробки тестових сценаріїв, які перевіряють правильність роботи програмного забезпечення на рівні коду. Методи тестування білої скриньки аналізують внутрішні структури, використані структури даних, внутрішній дизайн, структуру коду та роботу програмного забезпечення, а не лише функціональність, як у тестуванні чорної скриньки.

Основними перевагами даного методу є:

- ретельність: перевіряється весь код та структура застосунку завдяки доступу до коду;
- оптимізація коду: допомагає оптимізувати код та видалити помилки;
- раннє виявлення дефектів: завдяки доступу до коду, тестування може початись раніше, оскільки не потребує користувацьких інтерфейсів;
- інтеграція з SDLC: можна легко розпочати в життєвому циклі розробки застосунку;
- виявлення комплексних дефектів: за допомогою даної техніки можна знайти дефекти, які неможливо виявити за допомогою інших методів тестування.

Тестування чорного ящика – це метод тестування, в процесі якої тестувальник не зацікавлений у внутрішніх деталях застосунку, його

структурі, програмному коді, а зосереджується на перевірці його функціональності. Іншими словами, тестувальника цікавлять лише вхідні та вихідні дані системи, а не те, як вони обробляються в системі. Даним метод тестування широко використовується для кінцевого контролю якості програмного забезпечення та його відповідності до заданих вимог.

Основними перевагами даного методу є:

- простота реалізації: тестеру не потрібно мати великого рівня знань в програмуванні, щоб реалізувати даний метод;
- ефективність при тестування великих систем: зосереджується на кінцевих функціональних результатах, що зручно при роботі з великою кількістю функцій;
- тестування з позиції користувача: тести виконуються з точки зору користувача, завдяки чому перевіряється зручність та інтуїтивність застосунку;
- повторюваність: тестові випадки легко відтворюються.

Тестування сірого ящика – це метод тестування, який являє собою комбінацію технік білого та чорного ящика. У цьому підході тестувальник має частковий доступ до внутрішньої структури, коду чи логіки програми, завдяки чому він має змогу ефективніше планувати та виконувати тести, проте воно є не настільки глибоким, як в тестуванні білого ящика. Даний метод тестування зазвичай використовується для комплексних систем, де важливо врахувати як її функціональність, так і деякі внутрішні аспекти.

Основними перевагами даного методу є:

- чіткість цілей: користувачі та розробники мають чіткі цілі під час тестування;
- тестування з точки зору користувача: в основному виконуються з точки зору користувача;
- невисокі вимоги до навичок програмування;
- покращена якість: завдяки комбінації двох методів покращується якість тестування.

1.3.3 Класифікація за цілями тестування

Класифікація за цілями тестування зазвичай поділяється на два типи: функціональне та не функціональне та відносяться до техніки тестування чорної скриньки.

Функціональне тестування – це метод тестування програмного забезпечення, під час якого система перевіряється на відповідність функціональним вимогам. Даний метод тестування гарантує, що вимоги чи специфікації задовольняються програмою. Він зосереджений на моделюванні фактичного використання системи, перевірці конкретних функцій та можливостей програми без оцінки інших характеристик. До основних задач функціонального тестування можна віднести перевірку правильності виконання основних функцій, перевірку логіки програми та інтеграції між різними компонентами.

Основними перевагами даного методу є:

- застосунок без помилок: функціональне тестування гарантує високоякісний продукт без помилок;
- задоволеність користувачів: гарантує виконання всіх вимог;
- зосередженість на специфікаціях: функціональне тестування зосереджено на специфікаціях відповідно до використання клієнтом;
- висока якість продукту: функціональне тестування гарантує безпеку та покращує якість продукту.

Функціональне тестування в свою чергу поділяється на наступні категорії: модульне, інтеграційне та системне тестування.

Модульне тестування - це метод тестування окремих блоків або компонентів програмного застосунку. Зазвичай даний метод використовується для забезпечення належної роботи окремих одиниць програмного забезпечення. Модульні тести зазвичай автоматизовані та призначені для перевірки певних частин коду, наприклад певної функції чи методу. Модульне тестування виконується на найнижчому рівні процесу

розробки програмного забезпечення, де окремі одиниці коду тестуються окремо. Модульне тестування в основному включено у тестування білої та чорної скриньки.

Інтеграційне тестування – це метод тестування, в якому відбувається перевірка того, як різні модулі або компоненти програмного застосунку взаємодіють один з одним. Він використовується для виявлення та вирішення будь-яких проблем, які можуть виникнути під час об’єднання різних модулів програмного забезпечення. Інтеграційне тестування зазвичай виконується після модульного тестування і використовується для перевірки того, що різні модулі програмного забезпечення працюють разом належним чином.

Системне тестування – це метод тестування програмного забезпечення, який оцінює загальну функціональність і продуктивність повного та повністю інтегрованого програмного рішення. Він перевіряє, чи відповідає система зазначеним вимогам і чи підходить вона для доставки кінцевим користувачам. Цей тип тестування виконується після інтеграційного тестування.

Нефункціональне тестування – це метод тестування, який використовується для перевірки нефункціональних вимог програми. В даному методі перевіряються всі аспекти, які не відносяться до функціонального тестування, тобто перевіряє нефункціональні атрибути системи та характеристики програм, які не стосуються конкретних функцій. До таких характеристик відноситься продуктивність, зручність інтерфейсу, безпека та інші. Нефункціональне тестування допомагає зрозуміти готовність програми до роботи в реальних умовах. Підсумовуючи, можна сказати, що функціональне тестування відповідає на питання «що робить система», а нефункціональне – «як вона це робить».

Основними перевагами даного методу є:

- покращена продуктивність: перевіряє продуктивність системи та визначає вузькі місця в ній;
- економія часу: займає менше часу, ніж інші методи;

- покращує взаємодію з користувачем: оскільки стосується нефункціональних аспектів, таких як зручність використання, то покращує взаємодію користувача з програмою;

- більш безпечний продукт: оскільки включає в себе тестування безпеки, то робить продукт більш захищеним від атак.

Нефункціональне тестування в свою чергу поділяється на наступні категорії: тестування зручності, сумісності та продуктивності.

Тестування продуктивності – це тип тестування який гарантує належну роботу програмного забезпечення за очікуваного навантаження. Це метод тестування, який виконується для визначення продуктивності системи з точки зору чутливості, реактивності та стабільності за певного робочого навантаження. це тип тестування програмного забезпечення, який зосереджується на оцінці продуктивності та масштабованості системи або програми. Метою тестування продуктивності є виявлення вузьких місць, вимірювання продуктивності системи за різних навантажень і умов і переконання, що система може обробляти очікувану кількість користувачів або транзакцій.

Тестування сумісності – це тип тестування, яке відноситься до категорії нефункціонального тестування та виконується на програмі для перевірки її сумісності (працездатності) на різних платформах/середовищах. Це тестування проводиться лише тоді, коли програма стає стабільною. Даний метод тестування має на меті перевірити функціональність розробленого програмного забезпечення на різних програмних і апаратних платформах, мережевих браузерах тощо. Тестування на сумісність є дуже важливим з точки зору виробництва продукту та впровадження, оскільки воно виконується, щоб уникнути майбутніх проблем із сумісністю.

Тестування зручності – це тип тестування, який перевіряє застосунок на його зручність для використання користувачами. Метою даного методу є виявлення проблем в інтерфейсі застосунку, оцінка інтуїтивності

використання та загального враження користувачів від продукту, що тестується.

1.4 Постановка задач

В ході виконання кваліфікаційної роботи можна виділити наступні задачі дослідження, які повинні бути виконані в рамках даної роботи:

- дослідити існуючу класифікацію методів тестування програмного забезпечення на базі платформи Android;
- обрати методи, які доцільно використовувати для тестування Android застосунків;
- дослідити обрані методи тестування;
- реалізувати та провести порівняльний аналіз досліджуваних методів тестування;
- провести аналіз отриманих результатів;
- надати рекомендації щодо використання досліджуваних методів тестування.

2 МЕТОДИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Типова модель тестування на базі платформи Android

Забезпечення хорошої якості Android-застосунків являється одним із ключових аспектів під час його розробки. Тестування в цьому процесі відіграє вирішальну роль, оскільки допомагає знаходити та усувати помилки, а також гарантує відповідність застосунку вимогам і очікуванням кінцевих користувачів. Процес тестування включає формальні процедури: підготовку вхідних даних, передбачення очікуваних результатів, виконання тестових сценаріїв і спостереження за результатами. Помилки можуть виникати на будь-якому з цих етапів, тому тестування спрямоване на забезпечення стабільної та ефективної роботи застосунку. Тестування допомагає не лише виявити недоліки, але й забезпечити відповідність програмного продукту вимогам, знизити ризики його використання та підвищити задоволеність кінцевих користувачів.

Ефективна організація тестування Android-застосунків базується на трьох ключових моделях: моделі середовища, моделі програми та моделі очікуваних помилок (рисунок 2.1). Ці моделі дозволяють врахувати як зовнішні фактори, так і внутрішню архітектуру застосунку [3].

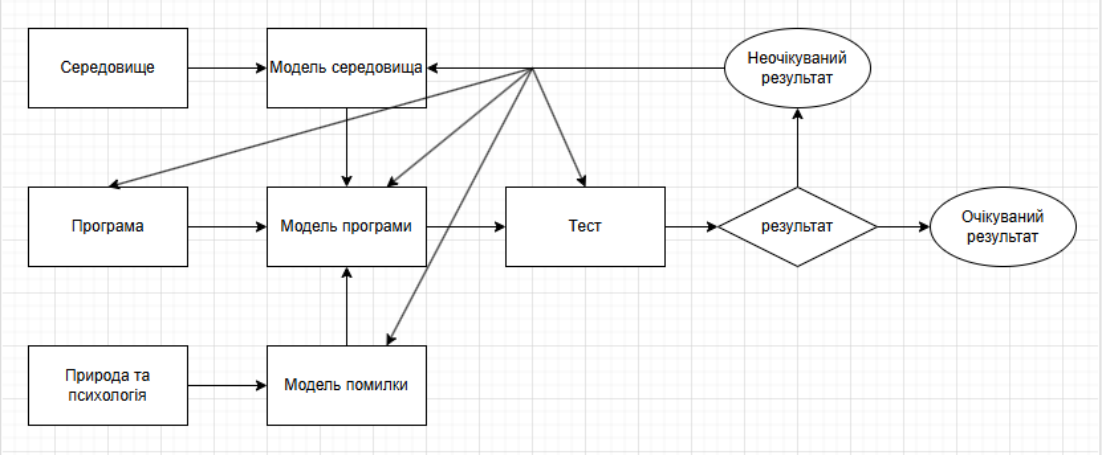

Рисунок 2.1 – Загальна модель тестування Android-застосунку

Модель середовища описує всі зовнішні фактори, що можуть впливати на роботу Android-застосунку. Вона включає різноманітне апаратне забезпечення (смартфони, планшети, пристрої з різними характеристиками) і програмне забезпечення (різні версії Android, бібліотеки, залежності). Тестування в межах цієї моделі спрямоване на перевірку роботи програми в різних умовах експлуатації. Для Android-застосунків важливим аспектом є перевірка роботи на різних пристроях і в різних умовах:

- використання пристроїв із різними розмірами екрану та роздільною здатністю;
- робота у різних мережевих умовах (Wi-Fi, мобільний зв'язок, режим офлайн);
- сумісність із різними версіями операційної системи Android.

Модель програми визначає, як Android-застосунок спроектований і працює. Вона охоплює такі аспекти, як архітектура застосунку, бізнес-логіка, взаємодія компонентів та обробка даних.

У випадку Android-застосунків важливо враховувати:

- реалізацію взаємодії між компонентами через internet або інші механізми комунікації;
- логіку обробки даних у фонових задачах або потоках.

Модель програми дозволяє зосередитись на тестуванні як функціональних, так і нефункціональних аспектів застосунку, зокрема продуктивності та зручності використання.

Модель очікуваних помилок фокусується на прогнозуванні типових помилок, що можуть виникнути під час розробки Android-застосунку. Вона базується на таких аспектах:

- аналіз типових проблем, наприклад, помилок адаптації під різні пристрої чи версії Android;
- оцінка ризиків у ключових сценаріях використання, таких як реєстрація користувача чи обробка платежів;

- урахування можливих помилок користувача, наприклад, некоректного введення даних.

Прогнозування помилок дозволяє зосередити тестування на найбільш критичних аспектах, таких як стабільність роботи, швидкодія та захист даних.

Однак для практичної реалізації такої концепції потрібно застосовувати конкретні методи тестування. Базовими та найбільш поширеними методами тестування зазвичай виступають модульне тестування, інтеграційне тестування та системне тестування. Вони дозволяють перевірити програму на всіх рівнях, від найменшої її частини до повної системи.

Модульне тестування фокусується на перевірці окремих частин застосунку, які виконують конкретні функції. Ці частини, які називаються модулями, зазвичай є найменшими одиницями програмного коду, наприклад, окремі класи, методи або функції. Основна мета модульного тестування – впевнитися, що кожен модуль працює правильно і виконує свої задачі відповідно до заданих вимог.

Інтеграційне тестування зосереджується на процес перевірки правильності взаємодії між модулями після їх індивідуального тестування. Даний тип тестування допомагає виявити проблеми, які виникають при комбінації компонентів, навіть якщо кожен з них окремо працює без помилок.

Системне тестування забезпечує перевірку всього застосунку в умовах, максимально наближених до реальних, включаючи тестування його продуктивності, безпеки та стійкості до помилок.

Після того як програма пройде різні етапи тестування, особливу увагу потрібно приділити аналізу та виправленню помилок виявлених у процесі тестування. Однак важливо не лише виявляти помилки, а й розуміти їх наслідки:

$$EI = F * (C + I + A), \tag{2.1}$$

де EI – Error impact (важливість помилки, представляється числовим значенням у відносній шкалі);

F – Frequency (частота помилки, зазвичай вимірюється у відсотках);

C – Correction cost (вартість виправлення, зазвичай вимірюється в грошовому еквіваленті);

I – Installation cost (вартість встановлення, зазвичай вимірюється в грошовому еквіваленті, який враховує витрати на оновлення ПЗ та додаткових ресурсів);

A – Associated costs (супутні витрати, зазвичай вимірюються в грошових одиницях, в деяких випадках може вимірюватись як репутаційні втрати, які будуть представлені в умовній шкалі числовим значенням).

Важливість кожної помилки залежить від частоти, вартості виправлення, вартості встановлення та наслідків:

- частота враховує наскільки часто виникає та чи інша помилка. Більше уваги зазвичай приділяється помилкам, які трапляються найчастіше;

- вартість виправлення помилки залежить від моменту її виявлення, чим пізніше її знайдуть, тим дорожче її усунути. Зазвичай вартість виправлення складається з витрат на виявлення помилки та витрат на її виправлення;

- вартість встановлення залежить від масштабу програми. Для програми одного користувача витрати будуть невеликими, тоді як для великих програм значно вищими. Іноді виправлення однієї помилки та її розповсюдження обходиться дорожче, ніж розробка нової системи;

- наслідки помилок варіюються від незначних до катастрофічних, відновлення після серйозних помилок може бути катастрофічним.

2.2 Метод модульного тестування

Завжди існує перший крок на шляху до мети: ти вперше написав програму, вперше провалився в роботі чи досягнув успіху. Так і в сфері тестування програмного забезпечення першим, базовим методом тестування являються модульні тести. Вони являються основою піраміди тестування Майка Кона (рисунок 2.2) [7].

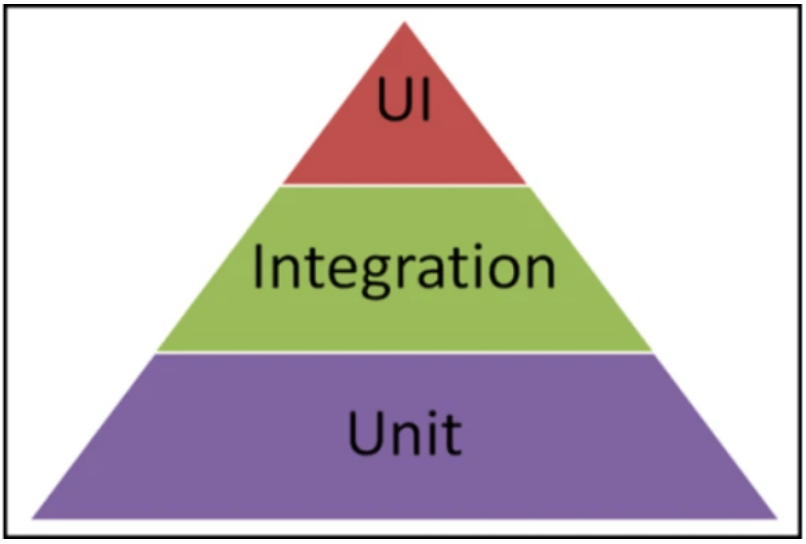

Рисунок 2.2 – піраміда тестування Майка Кона

Модульне тестування не є новою концепцією в розробці програмного забезпечення . Вона існує з перших днів мови програмування Smalltalk у 1970-х роках, і вона знову і знову підтверджує себе як один із найкращих способів покращити якість коду. Кент Бек представив концепцію модульного тестування в Smalltalk, і згодом вона перейшла у інші мови програмування, ставши постійною практикою під час розробки програмного забезпечення.

Модульний тест – це автоматизована частина коду, яка викликає одиницю роботи, що тестується, а потім перевіряє деякі припущення про один кінцевий результат цієї одиниці [8]. Він легко пишеться та швидко працює, заслуговує на довіру, читається та підтримується, стабільний у своїх результатах, доки виробничий код не змінився. Ці тести зазвичай

намагаються максимально ізолювати від зовнішньої логіки за допомогою моків, заглушок та шпигунів.

Одиницею роботи називають суму дій, які виконуються між викликом загальнодоступного методу в системі та окремим помітним кінцевим результатом тестування цієї системи. Помітний кінцевий результат можна спостерігати, не дивлячись на внутрішній стан системи, а лише через її публічний API та поведінку. Кінцевим результатом одиниці роботи може бути наступне:

- викликаний публічний метод повертає значення (функцію, яка не є пустою);
- існує помітна зміна стану або поведінки системи до і після виклику, яку можна визначити без опитування приватного стану (приклад: система може увійти раніше неіснуючим користувачем або змінити її властивості);
- існує звернення до системи третьої сторони, над якою тест не має контролю, і ця система третьої сторони не повертає жодного значення, або будь-яке значення, що повертається цією системою, ігнорується (приклад: виклик сторонньої системи журналювання, написаної не вами).

Для імітації залежностей в тестуванні часто використовують заглушки, моки та шпигуни. Заглушка може підмінити реальний компонент і забезпечити стабільну відповідь, яка не змінюється, незалежно від обставин. Це спрощений підхід, що дозволяє зосередитися на тестуванні основного коду без врахування сторонніх факторів.

Моки, на відміну від заглушок, не лише імітують залежності, але й дають змогу відстежувати, як саме вони були використані. Вони дозволяють перевірити, чи викликався метод, які параметри передавались і скільки разів це відбувалось, що дає змогу оцінити взаємодію між різними компонентами.

Шпигуни поєднують елементи моків і реальної логіки. Вони забезпечують виконання оригінального коду, паралельно збираючи дані про його використання. Це дозволяє аналізувати роботу залежностей, не втрачаючи при цьому функціональності.

Модульне тестування є важливою складовою процесу розробки програмного забезпечення, оскільки дозволяє перевіряти окремі частини програми – модулі – на наявність помилок ще на етапі їх створення. Це забезпечує надійність і правильність роботи кожного компонента системи. Зазвичай модульне тестування реалізується через підхід, відомий як TDD (Test-Driven Development), що означає "розробка через тестування". У рамках цього підходу тестування виконується до того, як код буде написаний, і базується на принципі "чорної скриньки", де відомо тільки, які вхідні дані ми подаємо та які результати хочемо отримати на виході.

Алгоритм модульного тестування можна представити, як послідовність кроків для перевірки окремих модулей програми, зокрема функцій чи методів, на їх правильність чи відповідність вимогам.

В загальному вигляді, послідовність кроків модульного тестування матиме наступний вигляд (рисунок 2.3):

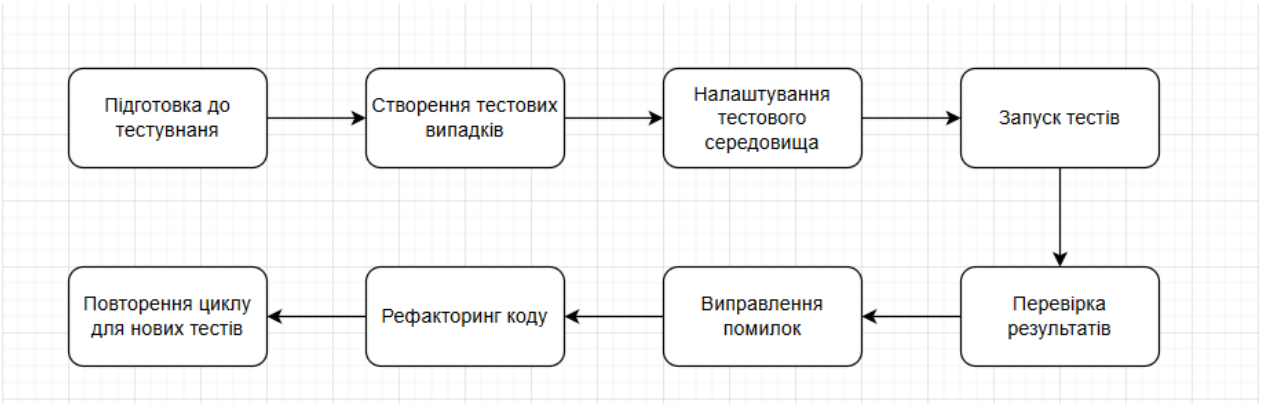

Рисунок 2.3 – Послідовність кроків методу модульного тестування

Даний рисунок демонструє загальну послідовність кроків методу модульного тестування, який складається з 8 ключових кроків:

- крок 1: підготовка до тестування; на даному кроці відбувається ознайомлення з вимогами до тестованого коду та ідентифікація тестованих компонентів;

- крок 2: створення тестових випадків; після того, як були визначені компоненти, що будуть тестуватись, розробляються тести, які будуть описувати бажану поведінку на основі заданих вхідних даних;

- крок 3: налаштування тестового середовища; відбувається підготовка необхідних вхідних даних та ізолювання тестованих модулів від інших частин програми;

- крок 4: запуск тестів; відбувається запуск написаних тестів з відповідними вхідними даними та нотуються результати після виконання тестованого коду;

- крок 5: перевірка результатів; на даному кроці порівнюються отримані результати з очікуваними (так званими assert, заздалегідь заданими значеннями), після чого, якщо результати співпадають – тест пройдено, у випадку відмінностей – тест не пройшов;

- крок 6: виправлення помилок; якщо тест не пройшов, то розробник знаходить і виправляє помилки в коді, після чого повторно запускає тестування (крок 4);

- крок 7: рефакторинг коду; після того, як всі тести були пройдені успішно, код можна покращувати шляхом рефакторингу (покращення структури без зміни функціональності). Після рефакторингу необхідно знову повернутись до кроку 4, щоб перевірити, чи не порушила оптимізація коду роботу програми;

- крок 8: повторення циклу; після того, як перший набір тестів було успішно виконано, додаються нові тести для перевірки інших функцій.

Узагальнюючи попередній алгоритм, можна виділити три основні етапи в модульному тестуванні, які допомагають забезпечити логічну структуру тестів, зрозумілу й зручну для аналізу результатів:

- етап 1: завдання тестових даних (так звані фікстури); на цьому етапі знаходяться підготовчі кроки, розробник налаштовує тестове середовище та створює всі необхідні дані чи залежності, які будуть використовуватись в тесті;

- етап 2: використання тестованого коду (виклик тестованого методу); це етап виконання логіки, яку потрібно протестувати. Розробник викликає метод чи функцію із заздалегідь підготовленими даними, ізольованими від сторонніх залежностей;

- етап 3: перевірка результатів та звірка з очікуваннями; на даному етапі відбувається перевірка результатів виконання тестованого коду відносно очікувань. Для цього використовують так звані твердження (assert), які порівнюють повернені значення, стан системи та взаємодії із залежностями.

2.2.1 Математична модель модульного тестування

Процес написання модульних тестів у контексті розробки програмного забезпечення можна описати наступним чином [9]. Для початку потрібно навести деякі визначення, зокрема:

- первинний юніт-тест – це мінімальний блок коду, який тестує основну функцію певної сутності для звичайних вхідних даних;

- вторинний юніт-тест – схожий на первинний, проте працює в граничних умовах (максимальні чи мінімальні значення, порожні чи нульові дані);

- простір R_p – це кінцевий набір значущих даних (ті дані, які можуть бути використані для виконання певних завдань) середовища програмування. Іншими словами його можна назвати сукупністю існуючих екземплярів будь-яких типів даних, доступних на фіксованій платформі розробки, куди можна віднести як примітивні типи (числа, рядки), так і складні типи (масиви, списки, об’єкти класів);

- функція $f(x): R_p \to R_p$ – це певна послідовність коду, яка виконується над даними x із простору R_p. Тобто, функція приймає на вхід

дані x з простору R_p (число, масив, список), і повертає певний результат, який також належить простору R_p. Наприклад, $f(x)$ може бути методом класу, що приймає на вхід x (число) і повертає результат виконання певного алгоритму у тому ж вигляді (числовому).

У підсумку можна сказати наступне:

- простір R_p – це сукупність усіх даних, доступних для обробки на певній платформі;
- функція $f(x)$ – це процес, який приймає вхідні дані з простору R_p і повертає результат, який також належить до цього простору.

Визначення первинного юніт-тесту виглядатиме наступним чином, нехай:

$$z = h(x) \tag{2.2}$$

де z – результат виконання тесту;

h – функція тесту;

x – вхідні дані.

Для фіксованого значення x_0 результат виконання тесту виглядатиме наступним чином:

$$z_0 = h(x_0) \tag{2.3}$$

де z_0 – результат виконання тесту з тестовими даними x_0;

x_0 – певне фіксоване значення тестових даних;

$h(x_0)$ – функція тесту, яка приймає вхідні дані x_0 і виконує операції для перевірки коду.

Тепер необхідно визначити функцію $a(z_0)$, яка буде перевіряти результат z_0 , порівнюючи його з очікуваним результатом: повертати 0 (якщо z_0 не коректне) або 1 (якщо z_0 коректне). Тобто ми беремо певні тестові дані x_0, виконуємо з ними певні маніпуляції в вигляді $h(x_0)$ та отримуємо z_0. Після цього ми створюємо assert (еквівалент ручної перевірки через функцію $a(z_0)$) для отриманих даних z_0 і перевіряємо правильність тесту.

Таким чином, тест виконує наступні дії:

- бере тестові дані x_0;
- виконує певні дії $h(x_0)$, отримуючи результат z_0;
- виконує перевірку результату за допомогою $a(z_0)$.

Підсумовуючи вище сказане, алгоритм написання тесту виглядатиме наступним чином:

- визначення тестових даних і перевірка результатів;
- розробка функціонального коду;
- створення моків та допоміжного коду;
- рефакторинг.

2.3 Метод інтеграційного тестування

Інтеграційне тестування – це метод тестування програмного забезпечення, який зосереджений на перевірці взаємодії та обміну даними між різними компонентами або модулями програмного застосунку [10]. Метою інтеграційного тестування є виявлення будь-яких проблем або помилок, які виникають, коли різні компоненти поєднуються та взаємодіють один з одним (рисунок 2.4).

Рисунок 2.4 – Інтеграційне тестування

Інтеграційне тестування зазвичай виконується після модульного тестування та перед тестуванням системи. Це допомагає виявляти та вирішувати проблеми інтеграції на ранніх етапах циклу розробки, зменшуючи ризик виникнення більш серйозних і дорогих проблем у майбутньому.

Інтеграційне тестування є важливим, оскільки воно перевіряє, чи окремі програмні модулі або компоненти правильно працюють разом як цілісна система. Це гарантує, що інтегроване програмне забезпечення функціонує належним чином, і допомагає виявити будь-які проблеми сумісності чи зв’язку між різними частинами системи. Виявляючи та вирішуючи проблеми інтеграції на ранній стадії, тестування інтеграції сприяє загальній надійності, продуктивності та якості програмного продукту.

На сьогодні існує чотири підходи до виконання інтеграційного тестування (рисунок 2.5):

- інтеграційне тестування великого вибуху;
- інтеграційне тестування знизу вгору;
- низхідне інтеграційне тестування;
- змішане інтеграційне тестування.

Рисунок 2.5 – Підходи до інтеграційного тестування

Одним із найпростіших, але водночас найменш структурованих способів є тестування великого вибуху. У цьому випадку всі модулі системи об'єднуються одразу і тестуються як єдине ціле. Такий підхід найчастіше обирають для невеликих проєктів або у ситуаціях, коли відсутній чіткий план поступової інтеграції. Хоча цей метод не потребує значної підготовки, він часто ускладнює пошук і виправлення помилок через одночасне тестування всіх компонентів.

Інший підхід, відомий як тестування знизу вгору, базується на послідовній інтеграції модулів. Спочатку перевіряються компоненти нижчого рівня, а потім до них поступово додаються модулі вищого рівня. Для забезпечення взаємодії використовуються спеціальні драйвери, які допомагають передавати дані та керувати роботою модулів. Такий спосіб дозволяє послідовно знижувати залежності між компонентами системи, поступово замінюючи тестові драйвери реальними модулями.

На противагу цьому існує низхідне тестування, яке розпочинається з модулів верхнього рівня і поступово охоплює нижчі компоненти. Якщо певні модулі ще не готові, їхню поведінку моделюють за допомогою заглушок. Таким чином, цей підхід забезпечує перевірку функціональності основних частин системи ще до завершення роботи над усіма її складовими.

Особливим є змішане тестування, яке об’єднує переваги двох попередніх підходів. Завдяки паралельній роботі з модулями верхнього та нижнього рівнів цей метод усуває недоліки як "знизу вгору", так і "зверху вниз". Центральні компоненти системи перевіряються останніми, коли вже протестовані всі крайні рівні. Такий гібридний підхід забезпечує більш збалансоване тестування і дозволяє оптимізувати процес інтеграції.

Інтеграційне тестування включає в себе декілька ключових кроків, які допомагають забезпечити правильну інтеграцію окремих модулів в єдину систему, та має наступний вигляд (рисунок 2.6):

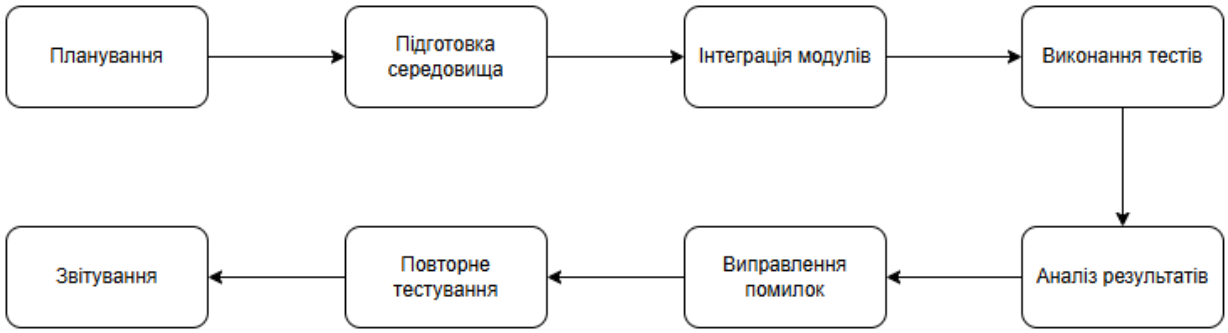

Рисунок 2.6 – Послідовність кроків методу інтеграційного тестування

- крок 1: планування; визначаються цілі тестування (що саме потрібно перевірити, які модулі), обирається стратегія тестування, створюється план тестування та підготовка тестових сценаріїв;
- крок 2: підготовка середовища; відбувається налаштування тестового середовища, розробляються заглушки для модулів, які ще не інтегровані;
- крок 3: інтеграція модулів; згідно з обраною стратегією тестування, відбувається покрокова інтеграція модулів чи груп модулів до основної системи;
- крок 4: виконання тестів; виконуються тестові сценарії для перевірки взаємодії між інтегрованими модулями та фіксуються результати;

- крок 5: аналіз результатів; відбувається аналіз даних, отриманих під час тестування, та, за необхідності, уточнення тестових сценаріїв для перевірки нових чи незапланованих випадків;

- крок 6: виправлення помилок; знайдені помилки передаються для виправлення розробникам;

- крок 7: повторне тестування; після виправлення помилок проводиться додаткове тестування для перевірки загальної стабільності системи;

- крок 8: звітування; готується звіт про результат інтеграційного тестування, включаючи в себе опис проведених перевірок, знайдених помилок, та загальну оцінку готовності системи до наступних етапів розробки.

2.3.1 Математична модель інтеграційного тестування

З технологічної точки зору, інтеграційне тестування є логічним продовженням модульного тестування [11]. Воно також працює з інтерфейсами модулів і підсистем, потребуючи створення тестового середовища, включно із заглушками (Stub) для заміни відсутніх модулів. Основна відмінність між модульним і інтеграційним тестуванням полягає в їхніх цілях, тобто у видах дефектів, які потрібно знайти. Ці цілі визначають вибір вхідних даних та методів аналізу. На етапі інтеграційного тестування часто використовуються методи перевірки покриття інтерфейсів, наприклад, аналіз викликів функцій або методів, а також аналіз використання інтерфейсних об’єктів, таких як глобальні ресурси чи засоби комунікації, що надаються операційною системою.

В загальному вигляді інтеграційне тестування можна пояснити використовуючи наступну структуру програми (рисунок 2.7), яка включає в себе модулі К, В1, В2, В11, В12, В21, В22, які пройшли етап модульного тестування.

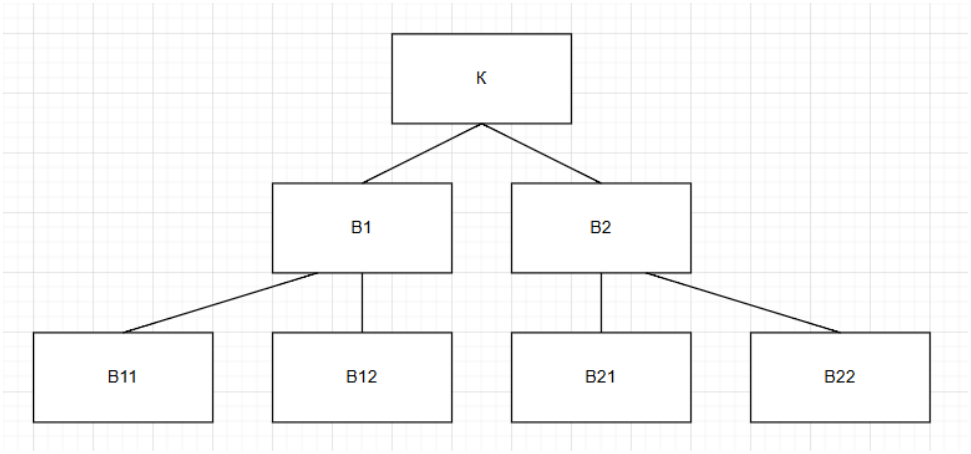

Рисунок 2.7 – Структура програми для інтеграційного тестування

Інтеграційне тестування перевіряє міжмодульні зв’язки, які існують в структурі програми. Зазвичай використовується інкрементальний підхід, під час якого відбувається поступове нарощення модулів програми, що тестується. Даний підхід поділяється на два типи: «зверху вниз» і «знизу вгору», кожен з яких має свій порядок тестування модулів програми. Для рисунку 2.7 алгоритм тестування модулів програми при тестуванні «зверху вниз» буде виглядати наступним чином:

1) К>XYK – тестується головний модуль К, розробляється тестовий набір XYK;

2) В1>XY1 – тестується перший підмодуль В1, розробляється тестовий набір XY1;

3) В11>XY11 – тестується підмодуль В11, який є підмодулем В1, для нього розробляється тестовий набір XY11;

4) В12>XY12 – тестується підмодуль В12, який є підмодулем В1, для нього розробляється тестовий набір XY12;

5) В2>XY2 – далі тестуємо другий підмодуль В2, для нього розробляється тестовий набір XY2;

6) В21>XY21 – тестується підмодуль В21, який є підмодулем В2, для нього розробляється тестовий набір XY21;

7) В22>XY22– тестується підмодуль В22, який є підмодулем В2, для нього розробляється тестовий набір XY22.

Спрощуючи, можна виділити наступні кроки тестування:

- тестування починається з головного модуля К;

- після цього тестуються підмодулі першого рівня В1 і В2. Воно включає в себе тестування їх функціонування окремо та їх взаємодії з головним модулем;

- далі тестуються підмодулі другого рівня В11, В12, В21, В22, основною метою якого є перевірка їх взаємодії з відповідними модулями першого рівня.

Більшість складних програм зазвичай має модульно-ієрархічну структуру, тобто система розбивається на невеликі частини, кожна з яких стає частиною більшого модуля. В таких випадках майже завжди тестування проводиться за критерієм покриття гілок $C1$. Це означає, що кожен модуль та перехід між ними повинні бути протестовані хоча б раз за весь хід тестування. Даний підхід забезпечує перевірку всіх можливих шляхів виконання програми. Математична модель тестування програми з урахуванням критерію покриття гілок $C1$ виглядає наступним чином :

$$M(P, C1) = ENij, \tag{2.4}$$

де E – множина переходів (дуг);

Nij – вхідні вершини графа модуля (позначають точки, з яких починається виконання тестів);

$M(P, C1)$ – математична модель тестування програми з урахуванням шляхів тестування P та критерію покриття гілок $C1$;

M – математична модель програми чи модуля;

P – шляхи, якими можна пройти в графі модуля чи програми (послідовність вузлів, які описують всі можливі маршрути виконання програми);

$C1$ – критерій покриття гілок.

Оцінка складності тестування модуля за критерієм С1 (топологічна складність Мак-Кейба) виглядає наступним чином:

$$V(P, C1) = q + k_{in}, \tag{2.5}$$

де q – число бінарних виборів для умов розгалуження;

k_{in} – число входів графу;

$V(P, C1)$ – оцінка топологічної складності тестування, яка показує яка кількість тестів необхідна для покриття всіх гілок програми, чим більше значення – тим складніше тестування.

У випадку, коли програма «Р» складається з «р» модулів, під час їх інтеграції в комплекс ми отримуємо ієрархічну (рисунок 2.7) модель програми, критерієм тестування якої на інтеграційному рівні використовується критерій покриття гілок $C1$. В такому випадку складність інтеграційного тестування всієї програми Р за критерієм покриття гілок $C1$ буде виражена наступною формулою:

$$V(P, C1) = \sum V(Mod_i, C1) - k_{in} + k_{ext} = e - n - k_{ext} + k_{out} = q + k_{ext}, (\forall Mod_i \in \mathrm{P}) \tag{2.6}$$

де n – кількість вершин (станів, операцій чи пунктів виконання) в графі;

e – число дуг (перехідних станів) в графі;

q – кількість умовних розгалуджень (умовні оператори, такі як if/else, додають додаткову складність, оскільки кожне розгалудження потрібно тестувати);

k_{in} – кількість вхідних точок графа, з яких починається виконання (під час інтеграції це загальна кількість вхідних точок для всіх модулів програми);

k_{out}– число виходів з графів (точки завершення виконання модулів або точки переходу до іншого модуля);

k_{ext} – число точок входу, які можуть бути викликані ззовні (це точки, які з'єднують програму з зовнішнім середовищем, такі як користувацькі інтерфейси);

$V(P, C1)$ – топологічна складність програми для тестування за критерієм С1, визначається як сума значень $V(Mod_i, C1)$ кожного окремого модуля;

$\sum V\,(Mod_i, C1)$ – сумарна складність модулів Mod_i, які входять до складу програми «Р».

2.4 Метод функціонального тестування

Функціональне тестування – це метод тестування програмного забезпечення, який відноситься до групи методів системного тестування, та спрямований на перевірку того, чи правильно виконуються всі функції системи згідно з її вимогами та специфікаціями. Це включає перевірку правильності обробки вводу та виводу, відповідності результатів очікуванням користувачів, а також перевірку взаємодії різних компонентів системи для забезпечення їх бездоганної роботи в рамках загальної архітектури програмного забезпечення.

Функціональне тестування відноситься до тестування чорної скриньки, тобто внутрішня структура системи не розглядаються під час виконання тестів [12]. Тестувальники оцінюють систему виключно на основі її функціональної специфікації, вхідних та вихідних даних.

Функціональне тестування зазвичай фокусується на наступних аспектах:

- коректність виконання функцій (чи робить система те, що повинна робити);
- взаємодія з користувачем (чи зручний та зрозумілий інтерфейс);
- відповідність вимогам (чи реалізує система всі заплановані функціональні можливості).

В якості вхідних даних використовуються компоненти, які вже пройшли попереднє модульне та інтеграційне тестування, і всі тестові випадки досягли своєї кінцевої точки без відкритих чи критичних помилок. Вхідні вимоги виглядають наступним чином:

- план тестування авторизований та підписаний;
- виконання тестових сценаріїв і тестових випадків відбувається відповідно до підготовки;
- структура тестових сценаріїв відповідає стратегії впровадження;
- для кожної нефункціональної вимоги мають бути доступні тестові випадки;
- тестові набори мають бути готові до виконання.

На виході тестувальник отримує наступні стандарти:

- виконаний кожен тест;
- не залишилося відкритих критичних, пріоритетних чи пов’язаних з безпекою проблем;
- в разі, якщо залишились помилки низького чи середнього пріоритету, розробник виправляє їх за згодою клієнта;
- тестувальник повинен надати звіт про вихідний результат тестів для обліку.

Найбільшу ефективність функціональне тестування показує у випадках, коли заздалегідь розроблено чіткий документ з вимогами, який містить в собі всі зміни та розуміння використання програми в режимі реального часу. Алгоритм тестування функціональності має наступний вигляд (рисунок 2.8) [13]:

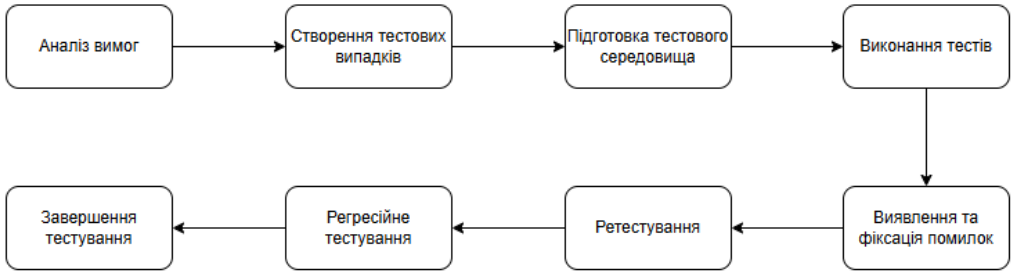

Рисунок 2.8 – Послідовність кроків методу функціонального тестування

Пояснюючи рисунок 2.8, можна виділити наступні етапи, які виконує кожен тестувальник під час функціонального тестування:

- крок 1: аналіз вимог; спочатку вивчаються вимоги та специфікації, щоб зрозуміти функціональні можливості системи, на основі вимог створюються тестові сценарії;

- крок 2: створення тестових випадків; після визначення тестових сценаріїв створюються конкретні тестові випадки, які мають конкретні дані для тестування кожної функції, визначаються вхідні дані та очікувані результати;

- крок 3: підготовка тестового середовища; відбувається налаштування тестового середовища, інсталяція програм та інструментів для тестування;

- крок 4: виконання тестових випадків; відбувається запуск тестів, де перевіряються практичні результати з очікуваними результатами, відбувається документація отриманих даних;

- крок 5: виявлення та фіксація помилок; якщо тест виявляє невідповідність, фіксується помилка і створюється звіт про неї, який направляється розробникам для виправлення;

- крок 6: ретестування; після виправлення помилок відбуваєтсья повторне тестування дял перевірки, чи дійсно проблема була вирішена;

- крок 7: регресійне тестування; після виконання всіх тестів проводиться регресійне тестування, щоб перевірити, чи зміни не вплинули на існуючі функції системи;

- крок 8: завершення тестування; після виконання всіх тестових випадків готується підсумковий звіт з результатами тестування, в якому описуються виявлені дефекти та загальний стан програми, а також надаються певні рекомендації щодо виправлення дефектів.

2.4.1 Математична модель функціонального тестування

Загальну математичну модель функціонального тестування можна представити як комплексну структуру, яка буде охоплювати декілька аспектів, зокрема модель тестування, модель помилок, покриття тестування та аналіз результатів.

Опис системи виглядатиме наступним чином:

- система S складається з множини компонентів $S = \{M_1, M_2, \ldots, M_n\}$, в якій M_i – функціональні модулі системи;
- стан системи X визначається через сукупність змінних $X = \{x_1, x_2, \ldots, x_k\}$, де x_i – значення певного параметра в системі;
- для тестування використовуються тестові сценарії $T = \{t_1, t_2, \ldots, t_m\}$, в якому t_i – певний набір кроків, що перевіряють систему за визначеним алгоритмом;

Поведінка системи відображається функцією, яка моделює реакцію системи на тестовий сценарій T:

$$f: T * X > Y, \tag{2.8}$$

де f – функція, що описує поведінку системи під час тестування;

T – множина тестових сценаріїв для перевірки функціональності системи;

X – множина станів або значень параметрів в системі, які описують поточний стан системи;

Y – множина результатів, отриманих після тестування (успіх чи невдача, певне числове значення чи стан системи після тестування);

Жодна система не є ідеальною, тому завжди будуть виникати помилки. Ймовірність помилки в компоненті можна знайти за наступною формулою:

$$P_{error}(M_i) = 1 - P_{success}(M_i), \quad (2.9)$$

де $P_{success}(M_i)$ – ймовірність правильної роботи компонента системи;

$P_{error}(M_i)$ – ймовірність того, що в компоненті M_i виникне помилка.

Ймовірність виявлення помилки тестом t_j у компоненті M_i знаходиться за наступною формулою:

$$p_{detect}(t_j, M_i) = g(t_j, M_i), \quad (2.10)$$

де $p_{detect}(t_j, M_i)$ – ймовірність того, що тест t_j виявить помилку в компоненті M_i;

$g(t_j, M_i)$ – функція, що описує ймовірність виявлення помилки, враховуючи покриття тестуt_j і характеристику компонента M_i.

Оцінка впливу помилки має наступний вигляд:

$$W_{eror}(M_i) = a_i * P_{error}(M_i), \quad (2.11)$$

де $W_{eror}(M_i)$ – оцінка впливу помилки в компоненті M_i;

a_i – коефіцієнт критичності помилки в компоненті;

$P_{error}(M_i)$ – ймовірність виявлення помилки в компоненті M_i.

Одною з метрик ефективності тестування можна використати так звану модель покриття тестування, яка визначається як відношення перевірених компонентів до загальної кількості компонентів:

$$C = \frac{M_{tested}}{M_{total}}, \tag{2.12}$$

де C – коефіцієнт покриття тестування, який показує, яка частина компонентів була перевірена під час тестування;

M_{tested} – кількість перевірених компонентів;

M_{total} – загальна кількість компонентів.

Ще одна з метрик ефективності полягає в успішності тестування, загальну формулу якої можна продемонструвати наступним чином:

$$P_{success} = \prod_{j=1}^{m} P_{success}(t_j), \tag{2.13}$$

де $P_{success}$ – ймовірність успішного виконання усіх тестів;

m – кількість тестів, що виконуються;

$P_{success}(t_j)$ – ймовірність того, що тест t_j буде виконано успішно.

Також можна використати формулу для розрахунку середньої кількості помилок, яка виглядатиме наступним чином:

$$AvgErrors = \frac{\sum_{i=1}^{n} W_{error}(M_i)}{n}, \tag{2.14}$$

де n – загальна кількість компонентів у системі, що тестується;

$W_{error}(M_i)$ – вага помилки в компоненті. В даному параметрі враховується критичність кожної помилки;

$\sum_{i=1}^{n} W_{error}(M_i)$ – сума значущості помилок знайдених в системі;

$AvgErrors$ – середня кількість помилок в системі.

3 РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ МЕТОДІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис загальної характеристики дослідження

У рамках кваліфікаційної роботи було формалізовано три методи тестування програмного забезпечення: модульне, інтеграційне та функціональне. Дані методи доповнюють один одного та забезпечують комплексний підхід під час перевірки програмного забезпечення, а також допомагають поетапно виявляти помилки під час розробки. Дослідження буде проводитись на основі реального прикладу (реалізації програми калькулятор з 4 основними операціями), для якого буде розроблено декілька тестів кожним з методів, проаналізовано результати виконання тестів та наведено порівняльні таблиці, в яких зазначено основні аспекти для кожного з методів.

3.2 Вибір засобів для проведення дослідження

Для проведення дослідження було використано стаціонарний комп’ютер на базі центрального процесора Ryzen 5500, 16 гігабайтів оперативної пам’яті та графічного адаптера RX 6600, що забезпечують гарну продуктивність у повсякденній роботі та виконанні програмних тестів.

В якості мови розробки було обрано мову програмування Python та середу розробки PyCharm, які дають змогу створити тестову програму калькулятора, додати в неї розроблені тести та виконати їх. Інтерфейс застосунку PyCharm з розробленим кодом калькулятора.та окремими файлами для написання тестів має наступний вигляд (рисунок 3.1):

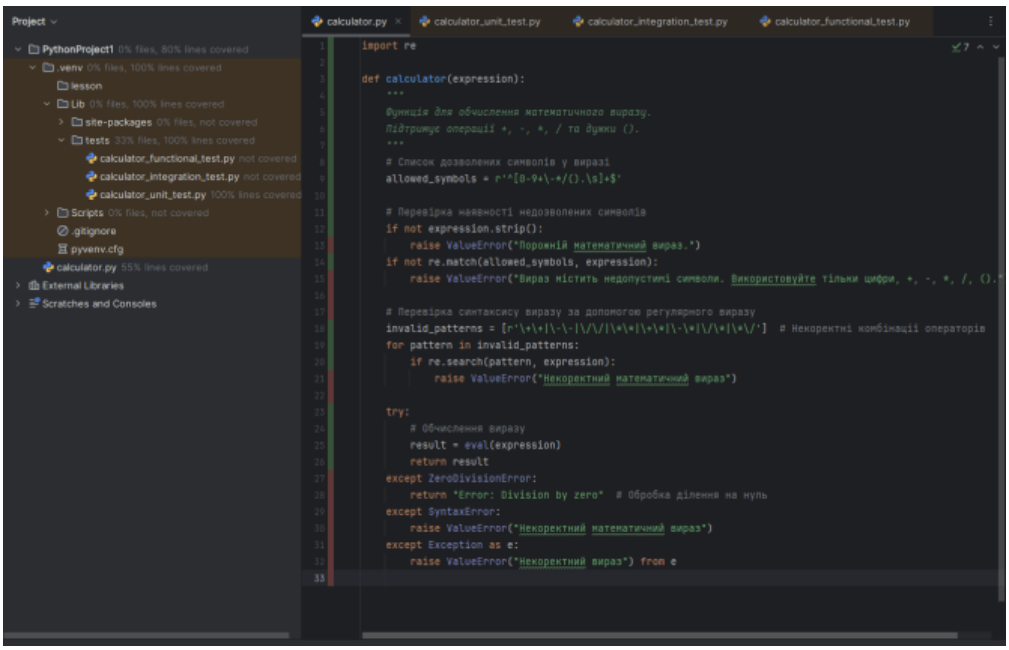

Рисунок 3.1 – Інтерфейс застосунку PyCharm

Для роботи з кодом та проведення тестів було використано наступні модулі:

- unittest – це стандартна бібліотека Python, яка використовується для виконання тестів, з її допомогою можна структурувати тести, автоматизувати перевірки та управляти станом системи під час тестування;

- time – даний модуль використовується для вимірювання часу виконання коду, в нашому випадку час виконання розроблених тестів;

- coverage – дана бібліотека призначена для вимірювання покриття коду тестами, з її допомогою можна проаналізувати, скільки рядків коду було виконано під час тестування, знайти ділянки коду які не виконувались, а також згенерувати звіт, в якому можна побачити результати тестування.

3.3 Вибір критеріїв для порівняння

У ході виконання кваліфікаційної роботи, досліджувані методи тестування будуть порівнюватись за наступними критеріями:

- покриття коду – даний критерій показує, яку частину коду покриває кожен із методів і дає змогу зрозуміти, які частини не були перевірені;
- час виконання – даний критерій демонструє швидкість виконання кожного з тестів;
- мета тестування – даний критерій демонструє які саме проблеми чи частини коду перевіряють досліджувані методи;
- об’єкт тестування – даний критерій демонструє, що саме виступає об’єктом тестування для того чи іншого методу;
- типи помилок – даний критерій демонструє, які саме помилки зазвичай знаходить кожен з методів тестування.

3.4 Експериментальне дослідження методів тестування програмного забезпечення

Для проведення експериментального дослідження було написано програмний код застосунку «калькулятор» та в окремих файлах створено три види тестових сценаріїв: модульні, інтеграційні та функціональні.

В результаті виконання кожного з наборів тестів ми отримуємо результати, в яких можемо побачити назву кожного тесту, час його виконання, результат тесту, загальний результат виконання всіх тестів, а також можемо дослідити покриття основної програми тестовим кодом завдяки підтримці використання модуля coverage. Результати кожного з тестів будуть перевірятись за допомогою оператора assert, який має в собі очікуваний результат та порівнює його з отриманими під час тестування результатами, і в разі помилки видає повідомлення AssertionError.

Для дослідження модульного тестування буде використано тестовий сценарій (файл calculator_unit_test.py), в якому створено 4 тести для перевірки базових функцій калькулятора, в результаті виконання якого ми отримаємо дані про успіх чи невдачу тестування, а також час виконання кожного з тестів (рисунок 3.2).

```
Ran 4 tests in 0.001s

OK
Результат тесту test_divide: 2.0
__main__.CalculatorTest.test_divide завершено за 0.000721 секунд
Результат тесту test_minus: 4
__main__.CalculatorTest.test_minus завершено за 0.000050 секунд
Результат тесту test_multi: 16
__main__.CalculatorTest.test_multi завершено за 0.000046 секунд
Результат тесту test_plus: 4
__main__.CalculatorTest.test_plus завершено за 0.000039 секунд
Wrote XML report to C:/Users/Vladislav/AppData/Local/JetBrains/PyCharm2024.3/coverage/

Process finished with exit code 0
```

Рисунок 3.2 – Результат виконання модульних тестів

Під час виконання модульного тестування було проведено 4 тести, виведені назви та результати обчислень кожного з тестів, час їх виконання, а також загальний результат (Ran 4 tests in 0.001s, OK), який повідомляє нам, що тести пройшли успішно та виводить загальний час виконання всіх тестів.

Для оцінки покриття тестами основного застосунку ми можемо переглянути результат виконання модуля coverage, який виводиться окремо (рисунок 3.3).

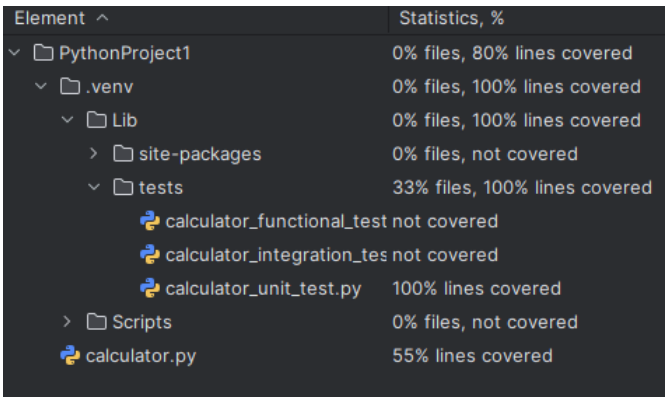

Рисунок 3.3 – Покриття модульними тестами коду розробленого застосунку

Аналізуючи рисунок 3.3, ми можемо побачити, що код модульного тестування був виконаний повністю (файл calculator_unit_test.py – 100% covered) і покрив 55% коду калькулятора (файл calculator.py – 55% covered).

Такий результат є цілком очікуваний, оскільки модульні тести перевіряють найменші частини програми та не охоплюють її загальний функціонал. Для збільшення покриття коду тестами можна створювати більшу кількість тестів, враховувати граничні випадки, тестувати виключення, проте це значно збільшить час, необхідний для розробки і виконання даних тестів, що не завжди є оптимальним варіантом. Варто також врахувати, що досягнення 100% покриття не завжди необхідно, оскільки високий відсоток покриття не гарантує відсутність багів. Існують частини коду, які рідко використовуються, тому їх тестування буде неефективним.

Покриття коду є важливим показником якості тестування, проте це лише інструмент, а не мета сама по собі, необхідно знаходити баланс між якістю тестування та наявними ресурсами, а також звертати увагу на специфіку проекту, що тестується.

Для дослідження інтеграційного тестування було розроблено окремий тестовий сценарій (файл calculator_integration_test.py), в якому було створено 2 інтеграційні тести, один з яких перевіряє загальні взаємозв’язки між модулями програми, а інший – взаємозв’язок модулів на граничних випадках.

В результаті інтеграційного тестування ми можемо побачити звіт про його успішність, час виконання тестів та загальний результат (рисунок 3.4).

Рисунок 3.4 – Результат виконання інтеграційних тестів

Інтеграційні тести були виконані успішно (Ran 2 tests in 0.002s, OK), виведені результати кожного з тестів, а для оцінки покриття коду використано модуль coverage (рисунок 3.5).

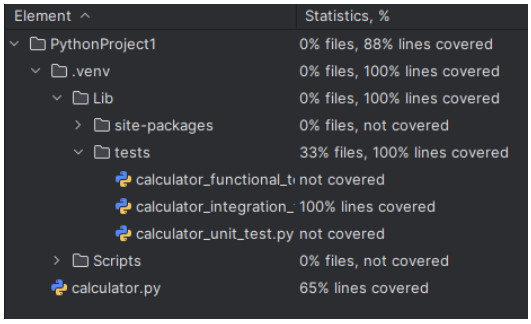

Рисунок 3.5 – Покриття інтеграційними тестами коду розробленого застосунку

Аналізуючи результати інтеграційного тестування, можна помітити, що час виконання тестів значно виріс, оскільки відбувається перевірка взаємодії багатьох модулів всередині застосунку, кожен тест в собі має більше тестових сценаріїв та відбувається перевірка ряда виключень на правильну взаємодію з основною частиною програми.

Варто також зазначити, що написання інтеграційних тестів займає більше часу, чим модульних, оскільки потрібно врахувати загальну структуру програми та правильно поєднати тестування модулів, що не потрібно під час модульного тестування.

Покриття коду тестами зросло на 10% (до 65%), завдяки перевірці більшої кількості модулів, проте в програмі все ще існують частини коду, які не перевіряються.

Для дослідження функціонального тестування було розроблено окремий тестовий сценарій (файл calculator_functional_test.py), в якому було створено 6 тестів, кожен з яких перевіряє певну частину функціональності розробленого застосунку (коректність математичних виразів, поділ на нуль, крайові випадки, синтаксичні помилки, недопустимі символи та порожні

вирази), а також перевірється виклик вийнятків у разі помилок під час тестування (рисунок 3.6).

```
Результат: 0
Результат: 0
Результат: 1000
Результат: 5
Результат: 5.0
__main__.CalculatorFunctionalTest.test_edge_cases завершено за 0.001024 секунд
Результат: Порожній математичний вираз
__main__.CalculatorFunctionalTest.test_empty_expression завершено за 0.000517 секунд
Результат: Вираз містить недопустимі символи (10+@5)
Результат: Вираз містить недопустимі символи (sin(30))
__main__.CalculatorFunctionalTest.test_invalid_symbols завершено за 0.000062 секунд
Результат: Некоректний математичний вираз (2++2)
Результат: Некоректний математичний вираз (5*/2)
__main__.CalculatorFunctionalTest.test_invalid_syntax завершено за 0.000051 секунд
Результат: 15
Результат: 7
Результат: 18
Результат: 5.0
Результат: 20
Результат: 5.0
__main__.CalculatorFunctionalTest.test_valid_expressions завершено за 0.000166 секунд
Результат: Error: Division by zero
__main__.CalculatorFunctionalTest.test_zero_division завершено за 0.000054 секунд
Wrote XML report to C:/Users/Vladislav/AppData/Local/JetBrains/PyCharm2024.3/coverage/
......
----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK
```

Рисунок 3.6 – Результат виконання функціональних тестів

Функціональне тестування має в собі набагато більший набір тестів, ніж два попередні методи, оскільки головною метою такого тестування є перевірка коректності роботи програми в ситуаціях, наближених до реальних, а також переконатися, що програма відповідним чином перевіряє помилки чи крайові випадки.

Тестування пройшло успішно (ran 6 tests in 0.002s), ми можемо побачити результат кожного тесту та час його виконання.

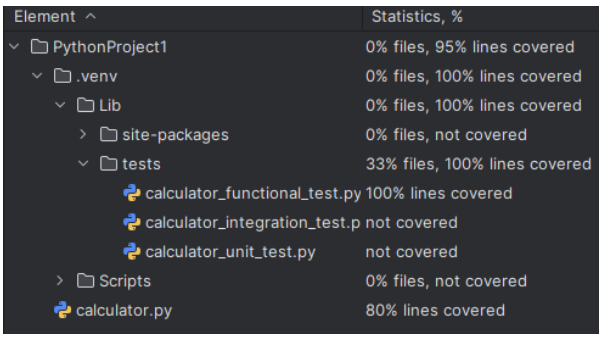

Рисунок 3.7 – Покриття функціональними тестами коду розробленого застосунку

На рисунку 3.7 можна побачити, що покриття коду тестами знову зросло, оскільки в функціональному тестуванні програма перевіряється в наближених до реальних умовах і зачіпає певні частини коду, які не були протестовані раніше.

3.5 Аналіз результатів дослідження

Метод модульного тестування являється найбільш простим у реалізації, оскільки тестує найменші компоненти програми та забезпечує швидкий зворотній зв’язок щодо базових функцій програми. Написання тестів просте у реалізації та швидко дає результати, оскільки зосереджене на дрібних частинах коду.

Модульні тести, як правило, швидкі у виконанні, але мають низьке покриття коду тестами, оскільки перевіряють невеликі частини коду. Основною метою такого тестування являється знаходження помилок в окремих функціях програми.

В нашому випадку загальний час виконання модульних тестів виглядає наступним чином:

- перевірка модуля ділення відбулася за 0.00071 секунду;
- перевірка модуля віднімання відбулася за 0.00005 секунд;
- перевірка модуля множення відбулася за 0.00004 секунди;

- перевірка модуля додавання відбулася за 0.00003 секунди;

Загальне покриття коду розробленого застосунку тестами становить 55%.

Модульне тестування являється найбільш ефективним у використанні на ранніх етапах розробки застосунку, оскільки займає мало часу, легке в реалізації та дає швидкий взаємозв’язок.

Модульне тестування найбільш доцільно використовувати на початкових етапах розробки для перевірки окремих модулів чи функцій на етапі їх створення, а також під час рефакторингу коду, щоб переконатися, що базові функції залишились працездатними після внесення змін. Зазвичай використовуються для перевірки критичних функцій, які повинні працювати безпомилково, а також зручні для їх автоматизації.

Метод інтеграційного тестування являється дещо складнішим у реалізації, оскільки вимагає розуміння архітектури програми та зв’язків між модулями, завдяки чому написання таких тестів повільніше від модульних, складніше в реалізації та не дає такого швидкого результату.

Інтеграційні тести, як правило, повільніші у своєму виконанні, але охоплюють більшу частину коду програми під час тестування. Основною метою таких тестів являється перевірка взаємодії між модулями програми, і відповідно знаходження помилок у їх взаємодії, які не виявляються під час модульного тестування. Зазвичай інтеграційне тестування виконується після модульного тестування, щоб перевірити взаємодію всіх протестованих модулів між собою.

В нашому випадку було виконано два інтеграційні тести, час виконання яких виглядає наступним чином:

- інтеграційний тест з кількома операціями виконано за 0.00126 секунд;

- інтеграційний тест з крайовими випадками виконано за 0.00015 секунд.

Покриття тестуванням зросло на 10% у порівнянні з модульними тестами, та становить 65%. Хоча покриття коду і зросло, частина коду все ще залишилась неперевіреною.

Інтеграційне тестування найбільш доцільно використовувати після завершення розробки кількох модулів для перевірки їх взаємодії між собою, а також після інтеграції нових компонентів в систему, щоб переконатися в їх правильній взаємодії між існуючими частинами програми. Рекомендується проводити інтеграційне тестування при додаванні нового функціоналу, який взаємодіє з декількома модулями, а також для модулів, які мають багато залежностей.

Метод функціонального тестування являється найбільш складним у реалізації, оскільки потрібно перевірити багато функціональних особливостей застосунку, що в свою чергу потребує написання більшої кількості тестів. Даний метод тестування імітує реальний сценарій використання завдяки чому може охопити більше коду та дозволяє знаходити помилки в реальних умовах роботи програми.

Функціональне тестування зазвичай відбувається на готовій програмі чи великих частинах програми, щоб перевірити її в цілому, як вона працює для кінцевого користувача. Трудомісткість даного методу значно більша в порівнянні з попередніми, оскільки тестові сценарії повинні охопити весь застосунок. Час виконання таких тестів може мати велику похибку в залежності від складності програми, що тестується: до швидкого виконання тестів на малих програмах, до дуже великого в порівнянні з іншими на складних застосунках.

Функціональне тестування спрямоване на знаходження помилок в логіці програми та її поведінці в реальних умовах, зокрема і на коректну обробку помилок, та використовується на завершальних етапах розробки.

В нашому випадку було створено 6 тестових сценаріїв, які перевіряли різні аспекти програми, час виконання яких виглядає наступним чином:

- тестування коректності математичних виразів виконано за 0.00005 секунд;
- тестування ділення на нуль виконано за 0.00005 секунд;
- тестування крайових випадків, в тому числі і з нулем виконано за 0.00102 секунди;
- тестування на синтаксичні помилки виконано за 0.00016 секунд;
- тестування на ввод недопустимих символів виконано за 0.00006 секунд;
- тестування на взаємодію з порожніми виразами виконано за 0.00051 секунду.

Покриття коду тестами зросло до 80%, що дає найкращий результат в порівнянні з іншими, проте не є ідеальним, оскільки залишились частини коду, які не було протестовано. Досягнення 100% покриття коду на невеликому проекті цілком можливо, проте не є доцільним, оскільки потрібно створювати набагато більше тестів, і не всі частини коду мають необхідність в тестуванні, варто враховувати специфіку кожного проекту та виділяти час на тестування найбільш критичних частин програми.

Функціональне тестування зазвичай виконується для тестування реальних сценаріїв використання, щоб перевірити коректну обробку помилок, після внесення великих змін до коду, щоб впевнитись, що програма працює як і очікувалось, а також перед релізом продукту для перевірки підтримки всіх заявлених функцій. Рекомендується виконувати перед релізом програми чи великим оновленням, а також в комбінації з іншими методами тестування для охоплення всіх можливих сценаріїв.

На рисунку 3.8 можна побачити порівняння досліджуваних методів тестування за критерієм покриття коду.

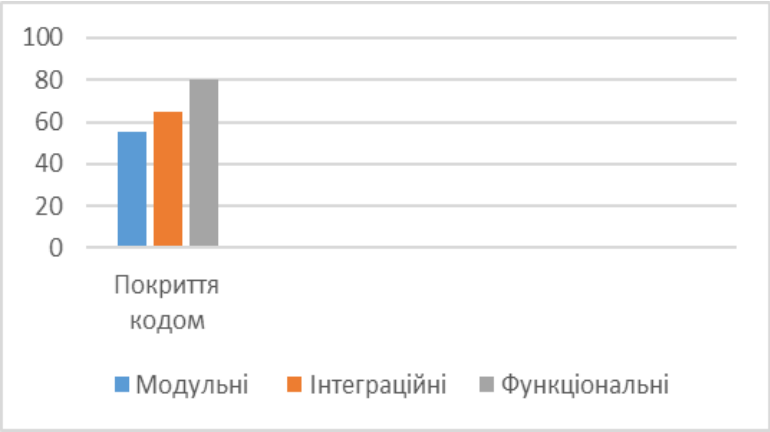

Рисунок 3.8 – Порівняння методів тестування за критерієм покриття коду

Модульне тестування має найменше покриття коду тестами (55%), інтеграційне тестування займає друге місце (65%), а функціональне тестування має найбільше покриття (80%).

Ще одним важливим критерієм для порівняння досліджуваних методів є час виконання тестів (рисунок 3.9).

Рисунок 3.9 – Порівняння досліджуваних методів тестування за часом виконання тестів

Аналізуючи рисунок 3.9 можна побачити, що, як правило, модульні тести виконуються найшвидше (від 30 до 710 мікросекунд), що є логічним, оскільки вони перевіряють окремі базові функції. Найшвидшим є тестування додавання (30 мікросекунд), а найповільнішим ділення (710 мікросекунд).

Інтеграційні тести, як правило, мають середній час виконання (від 150 до 1260 мікросекунд), бо вони перевіряють взаємодію багатьох компонентів між собою. В нашому випадку, використання багатьох операцій значно збільшує час виконання тесту (1260 мікросекунд) і вибивається із загальної логіки, оскільки задіює в собі велику кількість операцій, що значно вповільнює його виконання.

Функціональні тести зазвичай займають більше часу ніж модульні та інтеграційні, проте в деяких випадках все ж являються швидшими від інтеграційних (в нашому випадку всі функціональні тести швидші ніж інтеграційне тестування багатьох операцій, що зазвичай є виключенням). Найдовше виконується перевірка крайових випадків (1020 мікросекунд), бо тут можуть викликатись додаткові вийнятки, які збільшують час обробки.

У таблиці 3.1 наведена порівняльна характеристика досліджуваних методів тестування.

Таблиця 3.1 – Порівняльна характеристика досліджуваних методів тестування

	Модульне тестування	Інтеграційне тестування	Функціональне тестування
1	2	3	4
Мета тестування	Перевірка роботи окремих модулів	Перевірка взаємодії між модулями програми	Перевірка програми в цілому, її працездатності та функціональності для користувача
Об’єкт тестування	Найменші одиниці програми (модулі, класи чи функції)	Групи модулів та їх взаємодія між собою	Готова програма або її великі частини в реальних сценаріях використання

Продовження таблиці 3.1

1	2	3	4
Типи помилок	Локальні дефекти, такі як помилки в реалізації алгоритмів чи невірні операції	Виявляються помилки взаємодії між модулями	Відсутність або некоректна функціональність програми, непередбачувані сценарії роботи
Покриття коду тестами	Найменше (в нашому випадку 55%)	Середнє (в нашому випадку 65%)	Найбільше (в нашому випадку 80%)
Час виконання	Найшвидший, бо перевіряється малий обсяг коду	Помірний, бо тестується взаємодія модулів	Найповільніший, бо тестується багато сценаріїв

Аналізуючи таблицю 3.1 можна зробити висновок, що модульне тестування зосереджується на перевірці окремих частин програми, таких як модулі, функції або класи, щоб переконатися, що вони працюють правильно. Цей тип тестування виявляє локальні помилки в алгоритмах і логіці, виконуючи тести найшвидше, але з найменшим покриттям коду.

Інтеграційне тестування перевіряє, як різні модулі програми взаємодіють між собою. Воно спрямоване на виявлення помилок у цій взаємодії, займає більше часу, ніж модульне тестування, і має середнє покриття коду.

Функціональне тестування орієнтоване на перевірку всієї програми або її великих частин у реальних сценаріях використання. Його мета – впевнитися, що програма працює відповідно до вимог користувача. Цей тип тестування охоплює найбільшу частину коду і є найповільнішим, оскільки перевіряє різні сценарії роботи програми.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи були досліджені методи тестування програмного забезпечення на базі платформи Android, такі як: метод модульного тестування, метод інтеграційного тестування та метод функціонального тестування.

У результаті проведеного дослідження було наведено порівняльний аналіз обраних методів тестування, що дає змогу побачити їх основні переваги та недоліки, а також розробити рекомендації щодо їх застосування залежно від ситуації та специфіки проекту.

Метод модульного тестування показав найшвидший час виконання (від 0,00003 до 0,00071 секунд) та являється найлегшим в реалізації, оскільки не вимагає від розробника знання внутрішньої структури програми, та стосується найменших одиниць коду. Використання даного методу дає змогу знаходити помилки на ранніх етапах розробки та покращити процес рефакторингу коду, проте він має обмежене покриття коду (55%) та може створити відчуття хибної безпеки, коли всі модульні тести виконуються, проте програма матиме помилки.

Метод інтеграційного тестування показав велику розбіжність у часі виконання (від 0,00015 до 0,00126 секунд), проте, як правило, він буде другий за швидкодією після модульного тестування. Дана розбіжність під час досліду пояснюється тим, що в інтеграційному тестування в кожному тестовому кейсі знаходиться більший набір тестів, кожен з яких може викликати інші залежності, що впливає на час виконання. Метод інтеграційного тестування складніший в реалізації, оскільки розробник повинен розуміти загальну структуру застосунку для написання інтеграційних тестів та правильного пов’язання модулів між собою, проте він забезпечує кращу стабільність системи завдяки перевірці взаємозв’язків в програмі. Інтеграційне тестування забезпечує більше покриття коду (65%),

проте написання інтеграційних тестів займає більше часу та залежить від готовності створених модулів.

Метод функціонального тестування, як правило, найповільніший (від 0,00005 до 0,00102 секунд), оскільки має перевіряти функціональність системи в цілому, не звертаючи увагу на логіку програми чи внутрішінй код. Функціональні тести порівнюють фактичний результат із очікуваними та імітують дії майбутнього користувача для перевірки основних функцій програми. Функціональне тестування проводиться безпосередньо перед випуском застосунку для перевірки якості продукту та знаходження критичних дефектів, які можуть впливати на роботу основних функцій програми. Зазвичай має найбільше охоплення коду тестами (80%), оскільки перевіряє всі очікувані функції в застосунку, проте найскладніший в реалізації.

Модульне тестування найкраще використовувати на початкових етапах розробки для перевірки окремих функцій або блоків коду. Воно допомагає швидко виявити помилки в логіці програми та є ефективним для ізольованих модулів. Зазвичай використовується в невеликих проектах у поєднанні з функціональним тестуванням.

Інтеграційне тестування застосовується після модульного, коли модулі об’єднуються в єдину систему. Воно допомагає виявити помилки у взаємодії компонентів. Зазвичай використовується у великих проектах у поєднанні з іншими методами, щоб забезпечити надійність застосунку та правильну взаємодію модулів між собою.

Функціональне тестування орієнтоване на перевірку відповідності програми вимогам та очікуванням користувача. Воно імітує реальне використання та допомагає переконатися, що ключова функціональність працює правильно. Даний метод тестування може використовуватись як у невеликих проектах, так і в складних, проте він не завжди необхідний, оскільки він складний та дорогий в реалізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Семко В.В., Кравченко П.О. Методи тестування програмного забезпечення та їх актуальність / Проблеми інформатизації. Тези доповідей дванадцятої міжнародної НТК. – Баку: ІСУ АР; Харків: НТУ «ХПІ»; Харків: ХНУРЕ; Харків: НАУ «ХАІ»; Бельсько-Бяла: УТіГН, 2024. – 21-22 листопада 2024. – Том 2. – С. 61.

2. Sourojit Das. What is Android Testing: Types, Tools, and Best Practices [Електронний ресурс] / Sourojit Das. – 2023. – Режим доступу до ресурсу: https://www.browserstack.com/guide/what-is-android-testing.

3. SOFTWARE TESTING METHODOLOGIES [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://mrcet.com/downloads/digital_notes/CSE/III%20Year/Software%20Testing%20Methodologies.pdf.

4. Етапи тестування програмного забезпечення [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/software-testing-stages/.

5. Veronika Z. Основні підходи до ефективного тестування ПЗ [Електронний ресурс] / Zhyhyr Veronika. – 2024. – Режим доступу до ресурсу: https://dou.ua/forums/topic/48845/.

6. Types of Software Testing [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/types-software-testing/.

7. Констянтин. Java Unit Testing: методики, поняття, практика [Електронний ресурс] / Констянтин. – 2023. – Режим доступу до ресурсу: https://javarush.com/ua/groups/posts/uk.2500.java-unit-testing-metodiki-ponjattja-praktika.

8. OSHEROVE R. The Art of Unit Testing, Second Edition [Електронний ресурс] / ROY OSHEROVE. – 2013. – Режим доступу до ресурсу: https://github.com/wuzhouhui/misc2/blob/master/%5BThe.Art.of.Unit.Testing(2nd%2C2013.11)%5D.Roy.Osherove.pdf.

9. Optimizing the process of creating unit tests [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/en/articles/49434/.

10. Integration Testing – Software Engineering [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/software-engineering-integration-testing/.

11. Інтеграційне тестування [Електронний ресурс] – Режим доступу до ресурсу: https://elib.lntu.edu.ua/sites/default/files/elib_upload/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F/page14.html.

12. System Testing – Software Engineering [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/system-testing/.

13. What is System Testing: It\'s Types With Best Practices [Електронний ресурс] – Режим доступу до ресурсу: https://www.lambdatest.com/learning-hub/system-testing.