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 перший (бакалаврський)

Автоматизація тестування під час проєктування мобільних застосунків за технологією Agile
(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Павло МИКИТА
(власне ім’я, прізвище)

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

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

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: доц. Наталія БОЛОГОВА
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

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

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

Освітня програма Комп’ютерна інженерія
(повна назва)

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

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Микиті Павлу Ярославовичу
(прізвище, ім’я, по батькові)

1. Тема Автоматизація тестування під час проєктування мобільних застосунків за технологією Agile

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) технологія: Agile; 2) методики: Record and Play та Scripting; 3) мова програмування: Java; 4) середовище розробки: Android Studio.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз проблеми та огляд існуючих рішень;
2) аналіз існуючих технологій Agile;
3) аналіз вимог до створення застосунку;
4) розробка рішення реалізації застосунку;
5) програмна реалізація застосунку;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій 15 слайдів

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	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Аналіз існуючих технологій Agile	31.05.25-03.06.25	
3	Розробка рішення реалізації застосунку	04.06.25-07.06.25	
4	Програмна реалізація програми	08.06.25-09.06.25	
5	Оформлення матеріалів кваліфікаційної	10.06.25-11.06.25	
	роботи		
6	Подання кваліфікаційної роботи керівникові та	12.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Наталія БОЛОГОВА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 72 с., 19 рис., 3 табл., 1 дод., 9 джерел.

МОБІЛЬНИЙ ЗАСТОСУНОК, AGILE, API, ІНТЕРФЕЙС, ANDROID, GUI, ПРОГРАМНА РЕАЛІЗАЦІЯ.

Метою кваліфікаційної роботи є розробка ефективної методики автоматизованого тестування мобільних застосунків, створених за технологією Agile.

У ході виконання кваліфікаційної роботи об’єктом дослідження є процес тестування мобільних застосунків, що створюються з використанням технології Agile. Предметом дослідження виступає методика автоматизації тестування мобільних додатків у рамках Agile-підходу.

У зв’язку з цим, актуальність дослідження полягає в необхідності створення методики, яка дозволить ефективно автоматизувати як API-тестування, так і тестування графічного інтерфейсу мобільних додатків у рамках Agile-проєктів.

ABSTRACT

Bachelor’s thesis: 72 pages, 19figures, 3 tables, 1 appendices, 72 sources.

MOBILE APPLICATION, AGILE, API, INTERFACE, ANDROID, GUI, SOFTWARE IMPLEMENTATION.

The major goal of this thesis is the qualification work is to develop an effective methodology for automated testing of mobile applications created using Agile technology.

In order to the course of the qualification work, the object of research is the process of testing mobile applications created using Agile technology. The subject of research is the methodology for automating the testing of mobile applications within the Agile approach.

In this regard, the relevance of the research lies in the need to create a methodology that will allow for the effective automation of both API testing and graphical interface testing of mobile applications within Agile projects.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

МЗ – мобільний застосунок

ОС – операційна система

ПЗ – програмне забезпечення

Agile – гнучка методологія розробки (англ., Agile Software Development)

API – набір правил, протоколів та інструментів (англ., Application Programming Interface)

BDD – процес створення (англ., Behavior-Driven Development)

DSL –предметно-орієнтована мова програмування (англ. Domain-specific language)

ВСТУП

На сьогодні технологія Agile набуває все більшого поширення в сфері розробки програмного забезпечення, зокрема – мобільних застосунків. Водночас постає актуальне питання ефективного тестування таких систем у межах коротких ітерацій.

У практиці Agile-проєктів найчастіше для автоматизації тестування використовується підхід Scripting, який добре зарекомендував себе у випадках тестування API. Проте, коли йдеться про створення автотестів для графічного інтерфейсу (GUI), цей метод виявляється недостатньо ефективним. Як наслідок – перевірка GUI здебільшого здійснюється вручну.

В умовах стислих ітерацій Agile вдається охопити лише базові перевірки інтерфейсу, що є прийнятним для веб- або десктоп-додатків, орієнтованих на обмежений перелік платформ чи браузерів. Але мобільні додатки мають значно ширший спектр варіантів: різні мобільні операційні системи, версії, типи пристроїв та їх конфігурації. Обмеження тестування лише базовими перевірками призводить до того, що численні помилки GUI залишаються непоміченими і потрапляють у продакшен. Такі дефекти часто виявляються самими користувачами, що негативно впливає на репутацію продукту й може призвести до його комерційної невдачі.

У зв’язку з цим, актуальність дослідження полягає в необхідності створення методики, яка дозволить ефективно автоматизувати як API-тестування, так і тестування графічного інтерфейсу мобільних додатків у рамках Agile-проєктів.

Об’єктом дослідження є процес тестування мобільних застосунків, що створюються з використанням технології Agile. Предметом дослідження виступає методика автоматизації тестування мобільних додатків у рамках Agile-підходу.

Мета роботи – дослідити та розробити ефективну методику

автоматизованого тестування мобільних застосунків, створених за технологією Agile.

Гіпотеза дослідження: використання запропонованої методики дозволить значно підвищити ефективність процесу тестування мобільних додатків, створених за методологією Agile.

Для досягнення поставленої мети необхідно реалізувати такі завдання:

- провести аналіз існуючих методів і видів тестування програмного забезпечення та оцінити їхню придатність для мобільних додатків;
- розглянути наявні методики та інструменти автоматизації тестування і визначити можливість їх застосування до мобільних систем;
- розробити власну методику автоматизації тестування мобільних додатків у контексті Agile.

1 МЕТОДОЛОГІЧНІ ЗАСАДИ ТЕСТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

1.1 Специфіка мобільних застосунків

Мобільний застосунок – програмне забезпечення, призначене для роботи на смартфонах, планшетах та інших мобільних пристроях, розроблене для конкретної платформи [1].

Існує кілька типів мобільних застосунків (МЗ) [18]:

- нативні;
- браузерні (мобільні веб-додатки);
- гібридні.

Нативні МЗ створюються до роботи з конкретної платформою. Їхній код пишеться мовами програмування, які є «рідними» для мобільних платформ. Для Android це Java, для iOS – Swift чи Objective-C. Такі МЗ фізично встановлюються на мобільний пристрій та поширюються через магазини мобільних додатків.

Браузерні МЗ – це оптимізовані версії веб-застосунків, спочатку розроблених для ПК. МЗ цього типу є мультиплатформенними – запускаються через браузер на мобільному пристрої з будь-якою операційною системою (ОС) та не використовують його програмне забезпечення (ПЗ).

Гібридні МЗ поєднують у собі риси перших двох типів - вони використовують веб-технології, але вимагають установки на пристрій і мають доступ до його функцій. Гібридні МЗ використовують вбудовану оболонку програми, яка містить веб-подання (web-view) для запуску веб-програми всередині програми для конкретної платформи.

Незалежно від типу, МЗ має ряд особливостей, які відрізняють його від веб- та настільних додатків. Далі будуть розглянуті ключові моменти [2-4]:

- взаємодія з основними функціями устрою-комунікатора. Смартфон – це, в першу чергу, телефон і жодні програми не повинні блокувати можливість прийняти/здійснити дзвінок, отримати/надіслати sms;

- робота з додатком у змінних умовах (наприклад: зміна рівня зовнішнього освітлення, нестабільний зв'язок з Інтернетом, перемикання між Wi-Fi і 3/4G, витрата заряду батареї пристрою, переведення програми у фоновий режим);

- короткий цикл розробки. Це викликано необхідністю часто випускати оновлення для забезпечення сумісності з новішими моделями мобільних девайсів та операційних систем для них;

- велика різноманітність мобільних пристроїв, розмірів та діагоналей екранів, версій ОС, на яких може бути використана програма, та графічних оболонок (для Android);

- сильна конкуренція серед виробників мобільного програмного забезпечення, через яку у користувачів формуються високі очікування.

Виходячи з цього, користувачі оцінюють якість мобільного додатка за такими критеріями як

- коректне виконання заявлених у назві та описі завдань,

- інтуїтивна зрозумілість та хороша швидкість відгуку всіх елементів управління,

- безперебійна робота за будь-яких умов.

Незручність використання, несумісність з популярними моделями девайсів, функціональні помилки, проблеми графічного інтерфейсу та інші недоліки призводять до негативних відгуків та різкого зниження рейтингу програми [5].

Тому якість МЗ – необхідна умова його затребуваності та конкурентоспроможності. Важливо виділити саме ті тести, які є найбільш критичними для конкретного додатка з метою скорочення витрат компанії та зниження ризику появи помилок.

Складність не дозволить провести всі можливі тести, тому слід

використовувати пріоритети зон тестування, серед яких можна виділити найбільш критичні [6-8]:

- інтерфейс – необхідно переконатися, що всі елементи мають зручний розмір, в додатку немає порожніх екранів, підтримує стандартні жести;

- апаратні ресурси – потрібно ретельно перевіряти обробку проблемних ситуацій (наприклад: встановлення програми при нестачі пам'яті, недостатній обсяг пам'яті для роботи програми в активному або фоновому режимі);

- перевірки різних версій ОС та дозволів екрану (наприклад: коректність відображення елементів на AMOLED- та retina дисплеї, у ландшафтній та портретній орієнтації, неможливість встановлення програми на девайс з непідтримуваною версією ОС);

- реакція на зовнішні переривання – в першу чергу це вхідні дзвінки та смс, перехід девайсу в сплячий режим, push-сповіщення інших програм, підключення додаткових пристроїв, відключення та включення Wi-Fi та мобільного інтернету;

- зворотний зв'язок з користувачем – відгук елементів на дії користувача повинен бути зрозумілим та своєчасним, реакція кнопок на натискання повинна відповідати їхньому стану (активна, натиснена, заблокована), при спробі видалити дані повинні з'являтися запобіжні алерти з можливістю скасувати дії;

- платний контент – вартість повинна відповідати функціоналу, що надається, покупки не повинні губитися при

- оновлення програми і так далі;

- локалізація – сюди відносяться максимальна кількість символів, які можна ввести в поля, що заповнюються, коректність перекладу, формат відображення дат і специфічних для конкретної мови символів;

- оновлення – основні перевірки – збереження даних користувача, функціонування урізаних версій програми, створених для роботи з більш старими версіями ОС;

- відповідність правилам та угодам конкретної ОС – необхідно перевіряти коректність назви та опису, формат настановного файлу, підтримку вимог різних магазинів додатків (для Android) [9];

- обробка випадкових та непередбачуваних подій – мобільні девайси часто виявляються в умовах, в яких отримують хаотичну інформацію (наприклад, коли відбувається розблокування пристрою, що лежить у кишені), тому має адекватно обробляти випадкове введення [8];

- імітація реальних умов використання – необхідно перевіряти роботу мобільного додатка за нестабільного зв'язку з інтернетом.

Перелік критичних зон може скорочуватися чи розширюватись залежно від специфіки конкретного МЗ. Наприклад, для програми, яка розробляється для використання в конкретній країні, не потрібно проводити тестування локалізації.

1.2 Методи та види тестування мобільних застосунків

Базові принципи тестування сформульовані у класичних книгах із тестування [8]. Автори виділяють два підходи до тестування програмних продуктів – метод чорної скриньки та скляної (білої) скриньки.

Тестування за методом «прозорої коробки» (білого ящика) орієнтоване на аналіз внутрішньої логіки програми. Основною метою є виявлення не синтаксичних, а логічних помилок, які складніше локалізувати й усунути. Такі дефекти не завжди помітні зовні, але можуть мати критичний вплив на роботу системи. Пошук подібних помилок дозволяє точніше діагностувати причини збоїв, однак не забезпечує повної оцінки якості програмного забезпечення (ПЗ).

При застосуванні методу білого ящика тестування відбувається в штучному середовищі, без врахування реальних умов використання. Такий підхід не охоплює користувацький досвід, включаючи ергономіку інтерфейсу, зручність навігації чи естетичні аспекти GUI, що є важливими

для кінцевих користувачів.

Крім того, даний метод фокусується на реалізованих функціях, тому існує ризик того, що нереалізовані або втрачені вимоги залишаться поза увагою.

На відміну від цього, метод чорної скриньки передбачає тестування без знання внутрішньої структури програмного продукту. Основна увага приділяється реакції програми на різні вхідні дані. Тестувальники, які виконують таке тестування, зазвичай працюють з відкритими інтерфейсами: користувацьким або програмним (API), і звіряють фактичну поведінку програми з очікуваними результатами, що описані в технічних вимогах.

Такий підхід орієнтований на виявлення помилок, які мають зовнішній прояв, зокрема некоректне функціонування, порушення в обробці даних або помилки в логіці користувацьких сценаріїв. Внутрішні механізми програми перевіряються лише опосередковано – через аналіз наслідків їхньої роботи.

Сильна сторона методу чорної скриньки – орієнтація на поведінку з точки зору кінцевого користувача. Це дозволяє виявляти найбільш критичні та помітні збої. Водночас, недоліком є обмежене розуміння внутрішніх процесів, що ускладнює пошук джерела проблеми.

Обидва підходи – білий та чорний ящик – не суперечать одне одному, а навпаки, можуть бути ефективно скомбіновані.

Тому у сучасній практиці часто застосовується метод сірого ящика, який поєднує переваги обох підходів. Тестувальник при цьому має уявлення про внутрішню архітектуру додатку, але також враховує й типові дії користувача. Це дозволяє ефективно перевіряти як зовнішній вигляд та поведінку програми, так і її внутрішні функціональні блоки.

Сіре тестування дозволяє точніше локалізувати проблеми – як на рівні коду, так і в контексті користувацького сценарію, що призвів до помилки. Такий підхід сприяє швидшій і точнішій оцінці дефектів, їх пріоритетності та потенційного впливу на систему в цілому.

У межах цього методу перевірки роботи МЗ зазвичай застосовуються

різні види тестування. Залежно від мети їх можна поділити на три групи [7]:

- функціональні,
- нефункціональні,
- пов'язані із змінами.

Функціональне тестування передбачає перевірку того, що ті чи інші функції реалізовані в даній версії програми та працюють відповідно до вимог.

Нефункціональне тестування спрямоване на перевірку дисфункцій МЗ. Нижче розглянуто основні види дисфункції тестування.

Інсталяційне (настановне) тестування спрямоване на виявлення проблем, що впливають на встановлення МЗ. Включає перевірку таких ситуацій як установка в новому середовищі (нова модель девайсу або версія ОС), оновлення поточної версії, зміна встановленої версії на старішу, повторне встановлення після невдалої спроби, видалення програми.

Тестування зручності використання (usability) покликане визначити, наскільки функціонал програмного продукту зрозумілий користувача і подобається йому. МЗ можна вважати зручним для використання, якщо користувач робить те, що йому здається найбільш правильним і при цьому у нього не виникає жодних питань та сумнівів у своїх діях [5]. При роботі з МЗ не повинно виникати «тупикових ситуацій», коли користувач не повністю розуміє, що відбувається у додатку та (або) не може контролювати те, що відбувається.

Тестування інтерфейсу передбачає перевірку того, наскільки коректно працює інтерфейс користувача і його компоненти [2].

Тестування безпеки перевіряє, чи здатне МЗ протистояти діям зловмисникам – спробам доступу до конфіденційної інформації чи запровадження шкідливого коду.

Тестування сумісності дозволяє визначити здатність програми працювати в конкретному оточенні (модель і конфігурація девайса, версія ОС, обладнання, що підключається – зовнішня клавіатура, гарнітура). Сюди

можна віднести перевірку коректного функціонування програми у різних орієнтаціях конкретного пристрою, перевірку модулів програми, їх дизайну щодо відповідності конкретної ОС [3].

Тестування продуктивності зводиться до вивчення того, з якою швидкістю додаток реагує на навантаження різного характеру та інтенсивності – звичайну, зростаючу та стабільно високу (стресову).

Тестування локалізації проводиться для перевірки якості та коректності адаптації МЗ до роботи певною мовою, а також з урахуванням культурних особливостей (наприклад формат дати, специфічні для мови роздільники та символи, одиниці зміни ваги, відстані).

За ступенем автоматизації тестування поділяється на ручне та автоматизоване. У першому випадку всі перевірки проводяться вручну, у другому – тест-кейси частково чи повністю виконує спеціалізований інструментальний засіб. При цьому тестувальник не виключається із процесу повністю – він займається розробкою тестових сценаріїв, підготовкою даних, аналізом та оцінкою результатів виконання, підготовкою звітів про знайдені помилки.

Також тестування мобільних програм може класифікуватися за суб'єктом виконання. Коли продукт уже зібраний воєдино, але ще надто «сирий» для демонстрації зовнішнім користувачам, всередині компанії-розробника проводиться альфа-тестування.

До виконання залучаються як професійні випробувачі, так і співробітники інших відділів організації. Альфа-тестування застосовується для перевірки життєздатності ідеї проекту та відстеження найбільш критичних помилок у коді МЗ [7].

Бета-тестування передбачає активну участь замовника і/або кінцевих користувачів у процесі перевірки продукту. Програмне забезпечення передається тестувальникам на бета-етапі, коли воно вже має достатній рівень стабільності, але може містити дефекти, які можна виявити лише під час використання програми в реальному середовищі.

У разі закритого бета-тесту доступ до додатку надається обмеженій групі осіб. У випадку відкритого тестування долучитися до перевірки можуть усі охочі. Закрите тестування дозволяє краще контролювати перелік користувачів, які взаємодіють із додатком, тоді як відкрите охоплює ширшу аудиторію, забезпечує більший обсяг фідбеку та дозволяє перевірити, як система поводиться при значному навантаженні на сервер.

Перевірка мобільного застосунку здійснюється після кожної зміни в коді – усунення помилок, додавання або видалення функціональних можливостей.

Димове тестування включає короткий набір базових перевірок, які дозволяють пересвідчитись у можливості встановлення, запуску та мінімального функціонування програми після компіляції нової або оновленої версії.

Регресійна перевірка виконується після змін у програмі або середовищі її виконання. Її мета – упевнитися, що раніше реалізовані функції продовжують працювати правильно.

Тестування збирання (build verification) допомагає визначити, чи відповідає нова збірка встановленим критеріям якості, що необхідні для початку повноцінного тестування.

Санітарне тестування, як один із різновидів регресійного, фокусується на перевірці окремої функції, чиї характеристики мають повністю відповідати вимогам специфікації.

За ступенем глибини перевірки як функціональне, так і нефункціональне тестування можна класифікувати на два підвиди – перевірку критичного шляху і розширене тестування.

Тестування критичного шляху зосереджено на тих частинах функціональності, які найчастіше використовуються користувачами під час стандартної взаємодії з програмою [5].

Щодо мобільних додатків, до критичного шляху зазвичай відносять етапи інсталяції та запуску, процес авторизації, навігацію між основними

екранами, а також дії, пов’язані з виконанням головної функції конкретного застосунку – наприклад, обробка або збереження даних певного типу, а також надсилання відповідних запитів.

На рисунку 1.1 показано суть тестування критичного шляху [5].

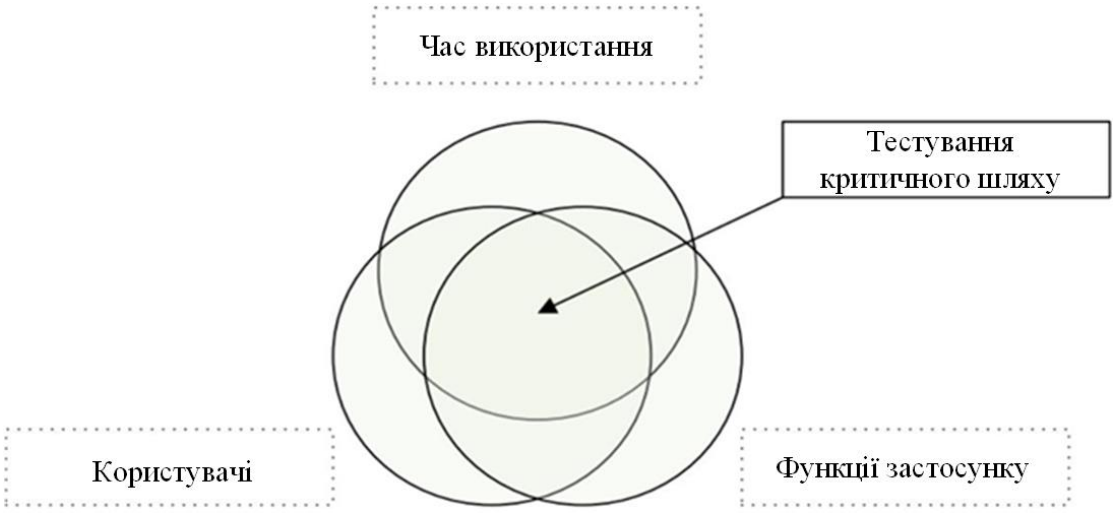

Рисунок 1.1 – Суть тестування критичного шляху

На відміну від тестування критичного шляху, розширене тестування охоплює повний обсяг функціональності, яка була зафіксована у вимогах до програмного забезпечення. Такий підхід базується на пріоритетності: спочатку перевіряються найбільш значущі функції, а менш важливі – у подальшому. Якщо є достатньо часу та ресурсів, у процесі тестування охоплюються навіть кейси з найнижчим пріоритетом.

Крім того, у межах розширеного тестування можуть виконуватись перевірки нестандартних, рідкісних і нетипових сценаріїв використання додатку, які зазвичай не входять до зони уваги при тестуванні основного функціоналу.

Залежно від типу тест-кейсів, що використовуються, усі описані вище методи перевірки можуть бути віднесені або до позитивного, або до негативного тестування.

Позитивне тестування зосереджене на тому, щоб перевірити правильну роботу додатку у випадку виконання користувачем усіх інструкцій, а також

при введенні коректних даних і дотриманні сценаріїв, передбачених документацією.

З огляду на вищевикладене, можна зробити висновок, що метод тестування сірої скриньки повною мірою відповідає специфіці перевірки мобільного ПЗ. Його використання дає можливість одночасно аналізувати як зовнішній вигляд інтерфейсу, так і логіку роботи серверної частини програми.

Однак варто враховувати, що застосування цього методу передбачає широкий обсяг перевірок, що, у свою чергу, веде до значних витрат часу й потребує достатнього залучення тестувальників.

1.3 Особливості процесу тестування мобільних застосунків на проєктах Agile

Більшість мобільних додатків розробляються в динамічному середовищі, де постійно виникає потреба у вдосконаленні функціоналу, оновленні інтерфейсу, адаптації наявного коду до нових версій мобільних операційних систем, а також у задоволенні зростаючих очікувань користувачів.

У зв’язку з цим під час створення мобільних застосунків дедалі частіше застосовується підхід Agile, який орієнтований на швидке впровадження працездатного функціоналу, його регулярне оновлення та активну взаємодію з клієнтом на всіх етапах розробки.

Agile або гнучка методологія розробки (agile software development) – група методологій розробки програмного забезпечення, заснованих на ітеративній поетапній розробці, де вимоги та рішення розвиваються за допомогою співробітництва між самоорганізуються між функціональними командами [4].

Ключові принципи методології описані в Agile Manifesto [9] – програмному документі спільноти Agile Alliance, розробленому в лютому

2001 року.

В основі Agile лежать 12 принципів:

- «найвищий пріоритет – задоволення потреб замовника, яке досягається завдяки регулярному та ранньому постачанню цінного програмного забезпечення;

- зміна вимог вітається навіть на пізніх стадіях розробки;

- працюючий продукт слід випускати якнайчастіше, оптимальна періодичність – від кількох тижнів до кількох місяців;

- протягом усього проекту розробники та представники бізнесу мають щоденно працювати разом;

- над проектом мають працювати вмотивовані професіонали. щоб роботу було зроблено, створіть умови, забезпечте підтримку та повністю довіртеся їм;

- безпосереднє спілкування є найбільш практичним та ефективним способом обміну інформацією як із самою командою, так і всередині команди;

- діючий продукт – основний показник прогресу;

- інвестори, розробники та користувачі повинні мати можливість підтримувати постійний ритм безкінечно;

- постійна увага до технічної досконалості та якості проектування підвищує гнучкість проекту;

- простота як мистецтво мінімізації надмірної роботи;

- найкращі вимоги, архітектурні та технічні рішення народжуються у команд, що самоорганізуються.

Команда повинна систематично аналізувати можливі способи покращення ефективності та відповідно коригувати стиль своєї роботи» [2].

На рисунку 1.2 [4] показана різниця в організації процесу тестування ПЗ із застосуванням каскадної методології та Agile.

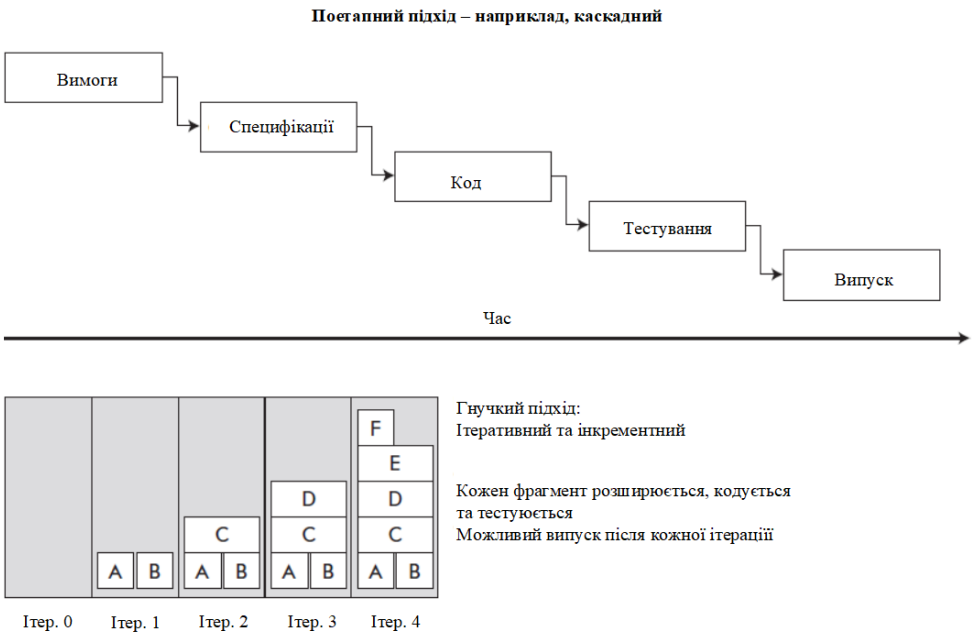

Рисунок 1.2 – Різниця в організації процесу тестування при застосуванні традиційної та гнучкої методологій

У разі використання каскадної мотодології розробки програмне забезпечення випускається одразу з повним обсягом запланованих функцій. Натомість при впровадженні гнучких підходів функціональність реалізується поетапно: на кожному етапі до вже наявних можливостей поступово додаються нові компоненти, які проходять окреме тестування в межах кожної ітерації.

Agile-підхід до тестування передбачає наступні зміни у роботі QA-команди [4]:

- тестування перестає бути ізольованою фазою у створенні ПЗ та активно застосовується на всіх стадіях життєвого циклу (ЖЦ) продукту – починаючи з планування. Таким чином, QA-фахівці можуть

- своєчасно виявляти найпростіші у виправленні помилки (неточності, неоднозначні деталі та інші проблеми документації), скласти уявлення про програмний продукт задовго до написання коду та виявити його потенційно слабкі місця;

- обсяг тестової документації скорочується до мінімуму: на зміну докладним тест-кейсам приходять більш високорівневі та універсальні тест-плани та чек-листи. Формат чек-листа дозволяє не розписувати перевірки досконально, за рахунок цього документацію простіше підтримувати в актуальному стані, а у роботі тестувальника збільшується частка дослідницького тестування;

- на всіх стадіях розробки підтримується зворотний зв'язок між фахівцями з тестування та рештою членів команди (розробники, бізнес-аналітики, дизайнери, проектний менеджер). Завдяки цьому у кожного з учасників проекту формується повніше і всебічно бачення продукту;

- швидка віддача від тестування – знайдені баги підлягають оперативному виправленню, що дозволяє підтримувати «чистоту коду» та уникнути накопичення застарілого та складно підтримуваного коду;

тестування – невід'ємна частина критерію готовності: ступінь готовності ПЗ визначається з урахуванням кількості, пріоритету та серйозності виявлених проблем. Наприклад, критерієм готовності МЗ до випуску може бути відсутність у продукті дефектів з пріоритетом вище «незначного» (Minor) або «середнього» (Normal).

Застосування тестування на всіх стадіях ЖЦ додатку створює умови для реалізації методології BDD (скор. від англ. Behavior-driven development, дослівно «розробка через поведінку»), яка заснована на поєднанні у процесі розробки суто технічних інтересів та бізнесу [3].

Відповідно до цієї методології, для спілкування членів проектної команди між собою використовується предметно-орієнтований мова. Основу останнього представляють конструкції з природної мови, зрозумілі нефахівцеві та описують поведінку програмного продукту та очікувані

результати.

Під очікуваним результатом розуміється поведінка ПЗ, що становить цінність бізнесу. Для його опису використовується специфікація поведінки (англ. behavioral specification), яка має таку структуру:

- заголовок - опис бізнес-мети в умовному способі;
- короткий опис користувальницької історії із зазначенням ролі користувача;
- один або кілька сценаріїв, кожен з яких розкриває ситуацію користувача поведінки.

Такий підхід дозволяє формувати єдине розуміння продукту усіма учасниками процесу розробки. За рахунок цього BDD дозволяє сформулювати вимоги до продукту, які зроблять його технічно реалізованим, корисним з погляду бізнесу та зручним для кінцевого користувача.

Взаємодія за принципами BDD показано на рисунку 1.3 [4].

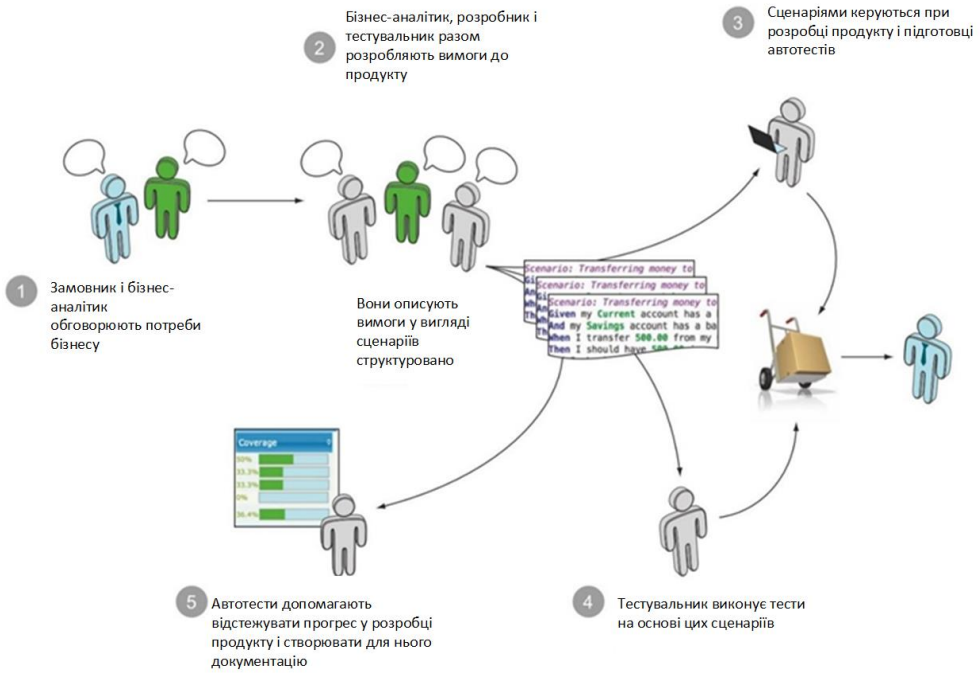

Рисунок 1.3 – Застосування методології BDD розробки ПЗ

Методологія BDD протиставляється традиційному підходу до організації взаємодії всередині проектної команди, у якому джерелом

помилок у роботі продукту часто стають різні трактування вимог бізнес-аналітиками, розробниками та тестувальниками.

Традиційний підхід показано на рисунку 1.4 [4].

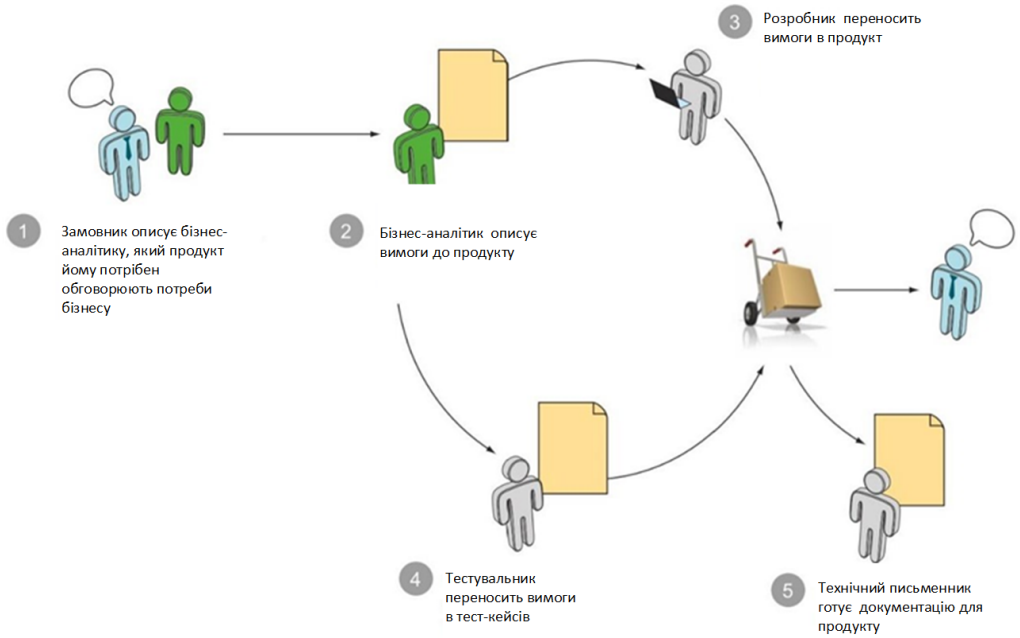

Рисунок 1.4 – Традиційний підхід до організації взаємодії усередині проектної команди

Перелічені принципи та зміни зачіпають усі стадії розробки програми:

- проектування. Відмова від вичерпного опису всіх аспектів програмного забезпечення дозволяє швидко вносити зміни до документації без шкоди для цілісності загальної ідеї програми. Основні завдання QA-фахівця на етапі проектування – тісне спілкування з бізнес-аналітиками, вивчення проектної документації та підготовка тестових сценаріїв;

- технологія. За рахунок частого випуску нових версій МЗ постійно еволюціонує та не втрачає актуальності, коли виходять нові моделі смартфонів та планшетів, мобільних ОС та «оболонок». Тестувальники виконують ручні перевірки, які спрямовані на пошук суттєвих (блокуючих, критичних та важливих) помилок. Паралельно створюються автоматизовані функціональні випробування;

- завершення ітерації. Активне спілкування із замовником дозволяє своєчасно отримувати зворотний зв'язок та, як наслідок, оперативно усувати помилки, забезпечувати узгодженість роботи всіх складових додатка. На цій стадії діяльність QA-фахівців зосереджена на відтворенні та аналізі проблем, які знайдені та зафіксовані на етапах проектування та розробки.

Отже, застосування гнучкої методології дозволяє побудувати процес розробки МЗ з урахуванням специфіки додатків цього.

2 ПІДХОДИ ДО РЕАЛІЗАЦІЇ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ У ПРОЄКТАХ ЗА МЕТОДОЛОГІЄЮ AGILE

2.1 Сфера застосування автоматизованого тестування

Автоматизоване тестування це процес використання програмних інструментів, які запускають нещодавно розроблене програмне забезпечення або оновлення через низку тестів для виявлення потенційних помилок кодування, вузьких місць та інших перешкод продуктивності. Засоби автоматизації тестування програмного забезпечення виконують такі функції:

Впровадження та виконання тестів:

- аналіз результатів;
- порівняння результатів з очікуваними результатами;
- формування звіту про продуктивність програмного забезпечення розробки.

Під час тестування нового програмного забезпечення або оновлень програмного забезпечення ручне тестування може бути дорогим і виснажливим. Тоді як автоматизовані тести менш дорогі та займають менше часу.

Автоматизовані тести можуть допомогти швидше виявляти збої з меншим ризиком людської помилки. Крім того, їх легше запускати кілька разів для кожної зміни або до досягнення бажаних результатів.

Автоматизація також прискорює процес виведення програмного забезпечення на ринок. Автоматизація дозволяє проводити ретельне тестування в певних областях, щоб ви могли вирішити загальні проблеми, перш ніж переходити до наступного етапу.

Піраміда автоматизації тестування допомагає зрозуміти, як часто потрібно виконувати кожен тип тесту.

Піраміда автоматизації тестування поділяє тестування на чотири рівні.

Нижній шар представляє тести, які ви повинні виконувати найчастіше. Рівні стають меншими, чим ближче вони до вершини піраміди, тобто тести, які вам слід виконувати рідше.

Ось типи тестів, які вам слід виконати за пірамідою автоматизації тестування, від найбільшого до найменшого:

- модульні тести;
- інтеграційні тести;
- тести API;
- тести інтерфейсу користувача.

1. Одиниця. Модульне тестування передбачає розбиття програмного забезпечення для розробки на легкозасвоювані одиниці для виявлення будь-яких помилок або проблем із продуктивністю.

Модульне тестування допомагає виявити помилки до того, як процес розробки програмного забезпечення зайде занадто далеко. Цей тип тестування відбувається на ранніх стадіях розробки програмного забезпечення, виявляючи та вирішуючи проблеми перед тим, як почати тестування.

Модульне тестування – це той тип тестування, який вам слід проводити найчастіше, оскільки він гарантує, що всі найдрібніші компоненти програмного забезпечення працюють правильно, перш ніж інтегрувати їх у ціле.

2. Інтеграція. Після того, як ви перевірили, чи кожен окремий компонент програмного забезпечення працює правильно, настав час об’єднати їх, щоб визначити, чи всі вони працюють разом. Інтеграційні тести перевіряють взаємодію компонентів, у тому числі в межах однієї програми.

Важливо, щоб усі інтегровані компоненти правильно взаємодіяли з програмним забезпеченням або зовнішніми службами, наприклад веб-службами. Таким чином, більшість людей вирішує створити базу даних для інтеграційного тестування, щоб перерахувати всі можливі сценарії.

Оскільки під час модульного тестування ви усунете більшість помилок

у коді, вам не доведеться проводити інтеграційне тестування так часто.

3. API.Тестування прикладного програмного інтерфейсу (API) перевіряє, чи можуть два окремі програмні компоненти спілкуватися один з одним за різних обставин.

Деякі типи тестування API включають:

- валідаційне тестування;
- функціональне тестування;
- тестування безпеки;
- тестування навантаження.

4. Інтерфейс користувача. Тестування інтерфейсу користувача (також відоме як тестування GUI) гарантує, що програмне забезпечення працює з різними інтерфейсами користувача, такими як операційні системи, браузери та інші місця, де з ним взаємодіють кінцеві користувачі. Тестування інтерфейсу користувача оцінює такі функції, як функціональність, візуальний дизайн, продуктивність і зручність використання. На щастя, тестування автоматизації інтерфейсу користувача позбавляє від необхідності купувати кілька пристроїв для тестування.

Автоматизація тестування інтерфейсу користувача враховує досвід кінцевого користувача та допомагає формувати програмне забезпечення відповідно до цієї взаємодії. Структура автоматизації тестування інтерфейсу користувача повинна включати сценарії тестування, пов’язані з вузькими місцями системи та процесу.

Оскільки всі попередні етапи тестування повинні були виявити та усунути більшість проблем, які могли виникнути в програмному забезпеченні, тестування інтерфейсу користувача має бути найменш трудомістким. Інструменти автоматизації інтерфейсу користувача економлять ще більше часу.

Основними критеріями успішного процесу автоматизації тестування є виявлення помилок програмного забезпечення та їх усунення до того, як проект перейде до іншої фази або досягне кінцевого користувача. Успішний

процес автоматизації тестування займає менше часу та створює програмне забезпечення, яке поводиться та надає функціональні можливості належним чином.

Незважаючи на всі плюси автоматизації, варто пам'ятати і про її потенційні ризики. Часті зміни функціоналу, коду, структури баз даних та інших компонентів програми мають на увазі регулярне оновлення написаних раніше автотестів. В результаті автоматизація тестування МЗ за часом нерідко можна порівняти з проведенням ручних перевірок.

Всупереч назві автоматизоване тестування не є повністю автономним процесом і має на увазі активну участь людини.

Виходячи зі сказаного вище, ефективність роботи команди тестування багато в чому залежить від того, які саме завдання було вирішено автоматизувати та як ця автоматизація була проведена [4,7,9].

Будь-який додаток (у тому числі – мобільний) можна умовно поділити на три рівні:

- рівень компонентів (Unit layer) включає код МЗ (наприклад, змінні, функції, методи, бібліотеки);

- рівень функціональності (Functional layer) – це бізнес-логіка програми, тобто практичний результат роботи, для отримання якого він створений;

- рівень графічного інтерфейсу користувача (GUI layer) включає компоненти МЗ, видимі кінцевому користувачеві (наприклад, екрани, кнопки списки).

Автоматизація приносить максимальний ефект процесу тестування, якщо вона зачіпає всі рівні МЗ, що тестується. На рисунку 2.1 показано, які типи автотестів відповідають усім рівням програми.

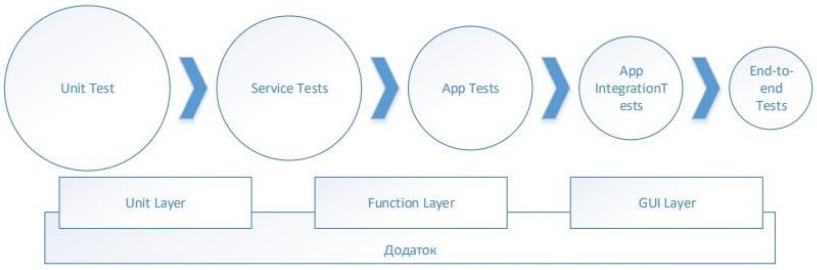

Рисунок 2.1 – Типи автотестів для різних рівнів програми

Автоматизація починається лише на рівні компонентів. Автоматизовані юніт-тести (Unit Tests) створюються для кожної нової можливості доданої в додаток. Вони дозволяють швидко виявляти помилки у коді.

Наступний ступінь – рівень функціональності. Йому відповідають сервісні автотести (Service Tests), спрямовані на тестування класів, що утворюють компонент у складі нового функціоналу. Такі тести запускаються лише після успішного завершення юніт-тестування.

Це тести, призначені для перевірки функціональності "у чистому вигляді". Зазвичай вони запускаються лише на рівні API без використання графічного інтерфейсу. Якщо потрібно протестувати взаємодію із зовнішніми сервісами, які не можуть гарантувати надання даних або з якихось причин недоступні, використовуються емулятори зовнішніх сервісів.

На рівні інтерфейсу виконуються автотести для перевірки програми в цілому (App Tests), інтеграції програм (App Integration Tests) та повні сценарні тести (End-to-End Test).

Автотести для перевірки програми в цілому відрізняються глибиною опрацювання та великим обсягом. Їхня мета – переконатися в коректності роботи всього додатку. Якщо програма має об'ємний функціонал, для тестування вона може бути розбита на кілька окремих програм, що надають користувачеві різні можливості.

В цьому випадку також застосовуються автотести інтеграції додатків, призначені для перевірки взаємодії «додатків всередині додатка» і коректності перемикання між ними.

Повні сценарні тести являють собою автоматизовані GUI-тести, які запускаються для всієї системи, відтворюють типові шляхи користувача або повні сценарії взаємодії.

2.2 Алгоритм автоматизації тестування

Успішна система автоматизації тестування буде дотримуватись наступного процесу.

Крок 1: Визначте цілі тесту.

Позначте, чого ви хочете досягти за допомогою тестування, перш ніж вибирати будь-які тести для запуску. Таким чином, ви не витрачаєте час на обробку на безглузді результати.

Крок 2: визначте пріоритетність тестування.

Встановлення списку пріоритетів для тестування дає змогу спершу зосередитися на найважливіших сферах і рухатися вниз до найменш важливих.

Крок 3: Кросплатформна застосовність.

Важливо перевірити, чи програмне забезпечення працює з різними операційними системами, браузерами та пристроями.

Крок 4: Легкість тестування.

Тести мають бути багаторазовими, застосовними до інших програм або здатними швидко адаптуватися до інших сценаріїв. Таким чином, ви не винаходите колесо, ініціюючи процеси тестування.

Крок 5: Оптимізовані комунікації.

Переконайтеся, що кожен, хто має внести свій внесок у тестування, зробив це та що інформація доступна в загальному місці. Складання чіткої карти того, хто повинен бути залучений до кожного тесту та результатів, може усунути надмірності або скасувати чиюсь важку роботу.

Крок 6: Гарантія якості [6].

2.3 Методики автоматизації тестування МЗ

Насамперед чим вибрати інструментальне засіб, необхідно визначитись із методикою автоматизації.

В даний час застосовуються чотири методики:

- запис та відтворення скриптів,
- написання сценарію,
- тестування під керуванням даними,
- тестування з урахуванням ключових слів.

Запис та відтворення скриптів (Record and Play) – передбачає використання утиліт для запису дій користувача у програмі. Програма перетворює запис на код і генерує автотести, які згодом виконуються без участі людини.

Основні переваги – простота застосування, не потрібні знання в галузі програмування.

Створення нових автотестів після внесення будь-яких змін до інтерфейсу МЗ. Зазвичай ця методика активно застосовується щодо димового тестування, і навіть одноразового прогону однотипних тестів у різних оточеннях.

Написання сценарію (Scripting) – методика полягає у використанні тестових сценаріїв, написаних мовами, спеціально розроблених для автоматизації тестування ПЗ [8]. Основна відмінність від методики «запис та відтворення скриптів» – код тестів створюється людьми, а не програмою.

Це потребує серйозних тимчасових та фінансових витрат, оскільки розробкою займаються програмісти чи тестувальники з високою кваліфікацією. З іншого боку – такі тести простіше підтримувати та масштабувати, оскільки під час написання коду вручну автор враховує можливі зміни у назвах структурних елементів МЗ, а також може погодити ці зміни з рештою учасників проектної команди.

Тестування під керуванням даними (Data-driven testing). Це методологія

створення скриптів та їх верифікації на основі даних, що містяться у базі даних чи сховищі. Використовується у випадках, коли потрібно реалізовувати однотипні перевірки різних комбінацій вхідних даних.

Тестування на основі ключових слів (Keyword-based testing) – методика написання автоматизованих тестових сценаріїв, що використовує файли, що подаються на вхід, не тільки для зберігання тестових даних і очікуваних результатів, але і ключових слів, що відносяться до тестованого додатку. Ключові слова інтерпретуються спеціальними процедурами, що викликаються з керуючого сценарію даного тесту [7].

Тести, підготовлені у межах цього підходу, є не програмний код, а послідовність дій зі своїми параметрами. Як і перший підхід дозволяє створювати автотести тестувальникам, які не мають навичок програмування.

При цьому автотести під керуванням ключовими словами стабільнішими і легшими в підтримці, ніж тести типу Record and Play. Для опису Keyword-based тестів використовуються ключові слова, для реалізації застосовуються фреймворки.

При виборі методики автоматизації тестування для конкретного продукту потрібно враховувати такі фактори: особливості предметної галузі та тип МЗ, а також методологію розробки програми.

2.4 Інструменти автоматизації тестування МЗ

Автоматизація тестування призводить до ускладнення схеми цього процесу. Схема ручного тестування включає три компоненти – тести, додаток та тестувальник, який виступає як «посередник» між ними. Він перетворює кроки тест-кейсів у дії з тим чи іншим інтерфейсом програми (API, GUI, Net та інші).

Тому, щоб автоматизувати тести, недостатньо використовувати єдиний інструмент, функції якого обмежуються запуском. Також необхідні інструментальні засоби, які формуватимуть тест-кейси, взаємодіятимуть з

інтерфейсами програми, що тестується, та іншими інструментами автоматизації [8].

Тому схема автоматизованого тестування включає комплекс інструментів. Залежно від функціональних можливостей та механізму роботи їх можна розділити на кілька груп [21]:

- драйвери,
- надбудови,
- фреймворки запуску,
- комбайни.

Вибір конкретних інструментів залежить від типу МЗ, тестування якого потрібно автоматизувати. Існують крос-платформні засоби автоматизації та спеціалізовані програми, призначені для роботи з конкретною платформою. Перші застосовуються для автоматизації МЗ будь-яких типів, другі – для автоматизації нативних та гібридних МЗ.

На рисунку 3.2 [9] показано зміну схеми тестування під час його автоматизації, а також позначено місце кожного з перерахованих вище типів інструментів у процесі автоматизованого тестування.

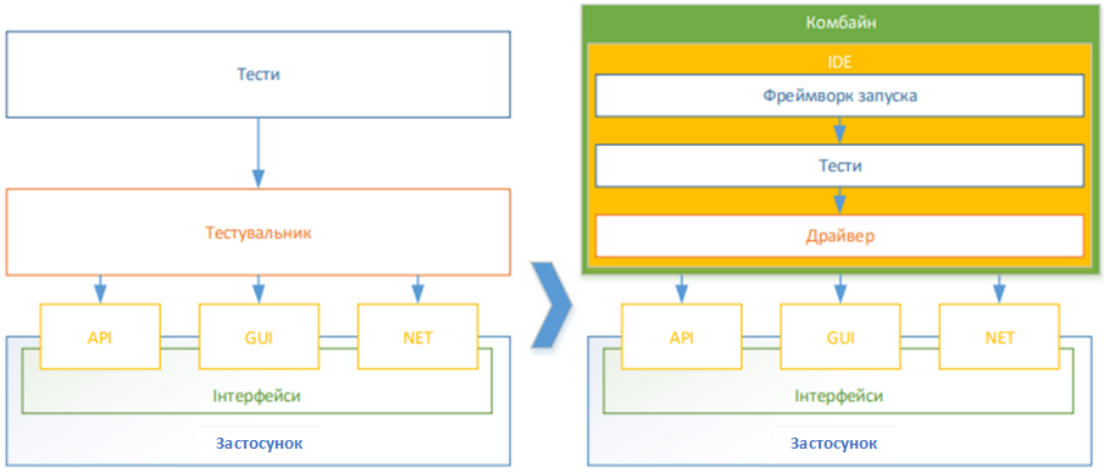

Рисунок 3.2 – Зміна схеми тестування під час автоматизації

У цьому контексті драйвер – це «програма, що видає себе за апаратне забезпечення та забезпечує зв'язок між програмами за стандартним

інтерфейсом. Іншими словами, цей інструмент надає API для одного з інтерфейсів програми».

Наприклад, драйвер для графічного інтерфейсу сприймає команди через API і відправляє їх в МЗ, що тестується, де команди перетворюються на відповідні дії з графічними елементами.

При тестуванні нативних та гібридних МЗ використовуються GUI-драйвери XCTest для iOS, UIAutomator та Espresso для Android. Для створення

Автотести графічного інтерфейсу для браузерних МЗ застосовується Selenium WebDriver.

XCTest підтримує версії iOS від 9.0 та вище. Для створення автотестів потрібен доступ до вихідного коду МЗ, що тестується. Тести для цього драйвера пишуться тими ж мовами, що й iOS-додатки – Swift та Objective-C.

У базовій комплектації XCTest тести можна запустити лише на симуляторі, але за допомогою сторонніх утиліт це можна зробити і на реальних пристроях. За допомогою рекордера, вбудованого в інтерфейс XCode, XCTest дозволяє записувати GUI-тести, знаходити графічні елементи та їх властивості.

UIAutomator дозволяє працювати з версіями Android, починаючи з Android 4.3 (API level 18), і не вимагає впровадження свого коду в проект.

Утиліта UIAutomator Viewer взаємодіє з програмами, запущеними на емуляторі або на реальному девайсі, отримує дані про GUI-елементи та показує їх локатори.

Espresso підтримує версії Android починаючи з Android 2.3.3 (API level 10) і призначений для тестування методом білої скриньки. До того ж утиліта не може самостійно працювати з системою Android та іншими додатками. Для запуску драйвера потрібен доступ до вихідного коду МЗ. Espresso можна використовувати разом із UIAutomator, поєднуючи в одному тесті команди обох інструментів.

3 ФОРМУВАННЯ МЕТОДИКИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ МЗ В УМОВАХ AGILE-ПРОЄКТУ

3.1 Розроблення підходу до автоматизації тестування мобільних застосунків

Успішна система автоматизації тестування буде дотримуватись наступного процесу:

Крок 1: Визначте цілі тесту.

Позначте, чого ви хочете досягти за допомогою тестування, перш ніж вибирати будь-які тести для запуску. Таким чином, ви не витрачаєте час на обробку на безглузді результати.

Крок 2: визначте пріоритетність тестування.

Встановлення списку пріоритетів для тестування дає змогу спершу зосередитися на найважливіших сферах і рухатися вниз до найменш важливих.

Крок 3: Кросплатформна застосовність.

Важливо перевірити, чи програмне забезпечення працює з різними операційними системами, браузерами та пристроями.

Крок 4: Легкість тестування.

Тести мають бути багаторазовими, застосовними до інших програм або здатними швидко адаптуватися до інших сценаріїв. Таким чином, ви не винаходите колесо, ініціюючи процеси тестування.

Крок 5: Оптимізовані комунікації.

Переконайтеся, що кожен, хто має внести свій внесок у тестування, зробив це та що інформація доступна в загальному місці. Складання чіткої карти того, хто повинен бути залучений до кожного тесту та результатів, може усунути надмірності або скасувати чиюсь важку роботу.

Крок 6: Гарантія якості.

Для перевірки результатів важливо використовувати команду QA. Використання групи тестування якості усуває ймовірність пропустити важливі помилки в кінцевому продукті.

Найбільше помилкове уявлення про автоматизоване тестування полягає в тому, що воно виправляє все для кожного програмного забезпечення для розробки. Це переконання призводить до наступних неправильних припущень:

1. Автоматизація замінює ручне тестування

Найкраща аналогія щодо автоматизації, яка замінює ручні завдання, походить від помилкової ідеї, що посудомийні машини можуть викорінити все ручне миття посуду. Однак завжди є посуд, який потребує ручного миття.

Ця ж концепція стосується автоматизованого тестування програмного забезпечення. Автоматизація прискорює звичайні сценарії тестування та зменшує навантаження на тестування. Однак це не усуває потреби в ручних тестерах, особливо на тому етапі усунення несправностей, коли розробник може краще визначити джерела помилок.

2. Автоматизація усуває помилки

Навіть найкращі тести не усунуть помилки чи збої системи. Деякі недоліки в коді властиві процесу. Інші помилки кодування активуються лише в дуже конкретних сценаріях. Використання автоматизованого тестування схоже на те, як світлофори роблять перехрестя набагато безпечнішими, але вони не усувають аварії, вузькі місця чи затори.

3. Для розробки автоматизації потрібен досвід

Хоча деякі автоматизовані тести є складнішими та потребують досвідченого розробника, багато пакетів тестування дозволяють новачкам писати прості автоматизовані тести.

Про що слід пам’ятати до, під час і після процесу автоматизації тестування

Як і у випадку з будь-якою системою тестування, певні припущення та реальність завжди повинні враховуватися:

1. Тестування не все виправить

Тестування – це спосіб виявлення проблем за допомогою роботизованого автоматизованого процесу. Це не одноразове рішення, яке не допоможе визначити кожну проблему. Необхідно повторне тестування, доки кожен компонент не запрацює належним чином.

2. Поспіх викликає помилки

Поспішне тестування загрожує цілісності тесту. Переконайтеся, що ви завершили кожен тест, якщо ви взагалі збираєтеся його виконувати. Зупинка до того, як вона досягне кінця, оскільки ви припускаєте, що вона дасть позитивні результати, може призвести до сюрпризів, які ви не захочете пізніше.

3. Навіть у тестах є помилки

Іноді тест може мати помилку, яка виявляється лише за певних обставин. Під час перегляду результатів пам’ятайте про можливість помилок тестування та виявляйте будь-які аномалії [1].

Надбудовою називається програма, яка взаємодіє з додатком через один або кілька драйверів, підвищує зручність їх використання або розширює їх можливості:

- модифікація поведінки драйвера без зміни API (наприклад: валідація даних, очікування виконання дії протягом заданого часу);
- підвищення рівня абстракції API шляхом спрощення складних команд, реалізації альтернативних стилів програмування тощо;
- уніфікація драйверів через надання єдиного інтерфейсу для них, наприклад, для використання одного і того ж коду тестів для використання з програмою на iOS та Android.

Найбільш відомими надбудовами є Appium та Calabash.

Appium підходить для автоматизованого тестування МЗ незалежно від платформи, типу програми та версії системи. Програма підтримує описані раніше драйвера XCTest, UIAutomator та Espresso, надає можливості:

- писати автотести будь-якою популярною мовою програмування, у тому числі – «нерідними» для МЗ мовами;
- створювати та запускати тести для будь-яких типів МЗ (нативні, гібридні, веб);
- працювати з будь-яким тестовим фреймворком;
- тестувати програми без доступу до коду.

Фреймворк запуску (далі – фреймворк), на відміну від драйверів та налаштувань, не є прошарком між тестами та МЗ. Це програма, яка служить для формування та запуску набору тестів, а також збору результатів їх виконання.

Комбайн – це утиліта, що поєднує в собі драйвери, фреймворки та можливості розробки. В автоматизованому тестуванні найчастіше використовуються комбайни Xamarin, UITest, Squish та Ranorex.

Xamarin є сервісом для розробки (в основному мовою C#) та тестування МЗ. Цей комбайн має інструменти автоматизації тестування, а також власні ферми мобільних пристроїв.

Ranorex дозволяє тестувати МЗ на емуляторах та реальних пристроях. Тести для нього пишуться мовами C# та VB.NET. Доступний лише для Windows, має рекордер для тестів.

Squish має власний рекордер і IDE. Мови для написання тестів: Ruby, Perl, Python, JavaScript.

3.2 Методика автоматизації тестування на Agile-проектах та оцінка її застосовності для МЗ

Технологія Agile передбачає частий випуск нових версій програми, що визначають специфіку тестування на Agile-проектах. На підготовку тестової документації, виконання тестів та аналіз результатів виділяється набагато менше часу, ніж на проектах із традиційним підходом до розробки.

Тому більшість проведених тестувальниками перевірок пов'язані з

реалізацією нових функціональних можливостей, розробка і тестування виконуються паралельно. Це дозволяє від початку ітерації закласти забезпечення якості та знизити ризик того, що нові можливості порушать роботу існуючого функціоналу.

Одним із способів оптимізації процесу тестування стає застосування автотестів. При цьому основною метою автоматизації стає можливість швидко оцінити стан програми та оперативно передати цю інформацію розробникам. На Agile-проект автоматизоване тестування, як і тестування в цілому, є не ізольованим завданням або етапом у роботі над додатком, а безперервним процесом, який вписаний у всі стадії життєвого циклу ПЗ.

Для забезпечення якісного зворотного зв'язку автотести повинні виконуватися часто та швидко, а їх результати – бути достовірними та достатньо деталізованими [26].

Мінімальним критерієм готовності нової версії програми до випуску вважається відсутність регресійних дефектів. Тому одним із ключових моментів автоматизації тестування на Agile-проектах є частий запуск регресійних тестів [3].

Як правило, автотести для регресії включають кілька наборів тест-кейсів, які відрізняються за кількістю та ступенем деталізації тест-кейсів:

- пакет «димових» автотестів для перевірки того, що програма успішно завантажується та запускається, а також перевірки кількох ключових сценаріїв роботи з програмою. "Димовий" набір запускається при кожному розгортанні програми;

- пакет функціональних автотестів застосовується для детальнішої перевірки роботи програми, зазвичай включає кілька тестових наборів для різних цілей. Такі набори при необхідності запускаються в різних оточеннях і перевіряють стабільність роботи програми. Подібні автотести запускаються кілька разів на день;

- пакет регресійних автотестів призначений для тестування програми як єдиного цілого. Така перевірка дозволяє переконатися, що різні частини

програми, які звертаються до інших програм, різних баз даних, сторонніх бібліотек та зовнішніх ресурсів, працюють коректно.

Цей набір призначений не для перевірки всіх можливостей додатки (їх робота раніше перевірена функціональними автотестами), а для тестування переходів з одного стану в інший, а також найбільш популярних сценаріїв користувача.

Для підготовки перерахованих автотестів використовують методику Scripting. Технічно вона дозволяє створювати автотести для графічного інтерфейсу. Але останні мають ряд особливостей, через які написання таких тестів з використанням Scripting стає неоптимальним рішенням.

Прийнято говорити про такі недоліки автотестів GUI:

- крихкість – для визначення графічних елементів та взаємодії з ними у тестах прописуються локатори, тому після зміни існуючих елементів або заміни їх новими тести перестають працювати;
- обмеженість тестування – графічний інтерфейс який завжди дозволяє тестувальнику повністю перевірити функціональність, оскільки він надає недостатньо деталей, необхідні верифікації;
- відносно низька швидкість виконання (проти юніт і API тестами) – оскільки тести проводяться через GUI, час завантаження графічних елементів помітно збільшує загальний час виконання тестів, у результаті зростає час отримання зворотний зв'язок;
- найменша окупність – через перераховані вище проблеми, автотести графічного інтерфейсу стають невигідними з фінансової точки зору.

В умовах Agile автотести GUI можуть втратити актуальність ще до першого запуску. Ручне створення автотестів нових графічних елементів займає багато часу, а виконання відкладається до наступної ітерації. На той момент графічний інтерфейс програми знову може змінитися, і написані раніше автотести доведеться переробляти.

Тому на Agile-проектах часто відбувається відмова від GUI-тестів на користь тестування на рівні API [3, 14], де автотести можна запустити одразу

після юніт-тестів.

Такий підхід до автоматизації допустимий для веб- та десктоп-додатків, які розраховані на роботу з обмеженою кількістю браузерів та платформ.

Але МЗ проектуються під різні мобільні платформи, версії їх операційних систем та конфігурації девайсів. Ручне тестування МЗ дозволяє перевірити лише базові сценарії та лише у найбільш популярних оточеннях.

Тому автоматизації тестування, яка не охоплює перевірки графічного інтерфейсу, не застосовується більшість мобільних додатків. Для МЗ, які розробляються за технологією Agile, відсутність автотестів GUI особливо критична.

Типова тривалість ітерації на Agile-проекті становить два тижні. За цей час неможливо протестувати графічний інтерфейс МЗ в результаті багато дефектів GUI потрапляють в продакшен-версію МЗ і виявляються кінцевими користувачами.

Для вирішення цієї проблеми необхідно розробити нову методику автоматизації тестування, яка дозволить автоматизувати тестування як API, так і графічного інтерфейсу МЗ.

Загальний принцип цієї методики можна сформулювати так: використання Scripting для створення тестів API, Record and Play для автотестів GUI.

Методику «запис і відтворення» має сенс застосовувати для тестування нових або екранів МЗ, що часто змінюються. Використання програм-драйверів дозволяє автоматизувати тестування нових

елементів GUI відразу після їхньої розробки, в рамках однієї ітерації. Також

Ця методика дозволяє швидко підготувати нові автотести для покриття позапланових доробок у дизайні або його кардинальній зміні.

Ручне створення автотестів нових екранів займає більше часу, ніж генерація коду з допомогою драйвера, і передбачає виконання автотестів

лише у наступній ітерації. До цього доведеться тестувати нові екрани вручну і, як наслідок, обмежитись перевіркою основних сценаріїв.

У результаті виникає подвійний ризик. По-перше, після введення нової версії МЗ в експлуатацію, користувачі можуть виявити значні дефекти в сценаріях, не покритих ручними тестами. По-друге, після запуску автотестів у наступній ітерації може з'ясуватись, що їх необхідно доопрацювати. У цьому випадку фактичне застосування автотестів відкладається ще на два тижні.

Тому методика «написання сценарію» більше підходить для автоматизованої перевірки API та функціональностей, які не змінюються або змінюються незначно.

Під час підготовки автотестів GUI необхідно переконатися, що вони мінімально перетинаються з функціональними автотестами. Останні повинні покривати більшу частину позитивних та негативних сценаріїв, спрямованих на перевірку взаємодії із сервером, базою даних та різними зовнішніми сервісами.

Автотести графічного інтерфейсу не повинні перетинатися із функціональними перевірками. Виняток – тести, пов'язані із формуванням коректних запитів при взаємодії користувача з різними контролами.

Паралельна розробка та запуск автотестів для API та GUI допомагає максимально швидко оцінити його якість та виявити найбільші дефекти всіх типів – структурні помилки в коді, проблеми функціоналу та графічний інтерфейс. Друга перевага використання комбінованої методики – можливість комплексно аналізувати та лагодити проблеми, пов'язані одночасно з функціоналом та компонентами GUI. Це дозволяє уникнути ситуації, коли дефекти на відповідних рівнях виявляються і виправляються асинхронно, через що ремонт на стороні бек-енду може призвести до появи нових проблем графічного інтерфейсу і навпаки.

4 ПЕРЕВІРКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОЇ МЕТОДИКИ

4.1 Апробація методики автоматизації тестування МЗ

4.1.1 Аналіз предметної області тестованого МЗ та умови проведення експерименту

Апробацію розробленої методики автоматизації проведено на нативному МЗ для iOS та Android, яке розробляється за методологією Scrum із застосуванням BBD-підходу.

У роботі використано таке визначення Scrum: «одна з гнучких технологій, що дозволяє жорстко фіксовані і невеликі за часом ітерації, звані спринтами (sprints), надавати кінцевому користувачеві працюючий продукт із новими бізнес-можливостями, котрим визначено найбільший пріоритет» [9].

МЗ призначене для адміністрування інформаційної системи (ІС) мережі книгарень. Це розрахована на багато користувачів додаток для отримання контрольованого доступу до ІС і виконання операцій, набір яких залежить від ролі користувача.

У застосунку визначено такі ролі:

- власник мережі магазинів (Owner);
- головний адміністратор мережі магазинів (Main administrator);
- адміністратор філії (Department administrator);
- співробітник магазину (Employee).

Нижче наведено основні сутності, представлені в структурі даних МЗ:

- користувач (User);
- роль (Role);
- покупець (Customer);
- книга (Book);
- магазин (Department);

- видавництво (Publishing office);
- замовлення від покупця (Order);
- постачання від видавництва (Supply).

На рисунку 4.1 показані варіанти використання МЗ для користувача з участю власника магазину.

Рисунок 4.1 – Варіанти використання М для власника магазину

На рисунку 4.2 показані представлені варіанти використання користувача з участю головного адміністратора.

Рисунок 4.2 – Варіанти використання МЗ для головного адміністратора

На рисунку 4.3 представлена діаграма варіантів використання користувача «адміністратор філії».

Рисунок 4.3 – Варіанти використання МЗ для адміністратора філії

На рисунку 4.4 показано діаграма варіантів використання для користувача «співробітник магазину».

Рисунок 4.4 – Варіанти використання МЗ для працівника магазину

Експеримент проводився після реалізації в МЗ нової функціональності, яка доступна всім користувачам МЗ – можливості перенесення книг зі списку поставки до списку книг магазину, який відображається на відповідному екрані (Book list). Для цієї функціональності було додано новий екран «Доставлені книги» (Delivered books) та кілька нових запитів:

- отримання списку доставлених книг;
- отримання конкретної книги зі списку доставлених книг;
- додавання книги зі списку доставлених книг до списку книг магазину.

Інші умови проведення експерименту описані у таблиці 4.1.

Таблиця 4.1 – Умови проведення експерименту

Методика автоматизації	Record and Play	Scripting
Платформа	iOS, Android	iOS, Android
Метод тестування	Метод сірої скриньки	Метод сірої скриньки
Рівень МЗ	GUI	API
Тестована частина системи	Фронт-енд	Бек-енд
Вид тестування	Графічний інтерфейс, сумісності	Нової функціональності, регресійне, безпеки

Блок-схема алгоритму проведення автоматизованого тестування із застосуванням розробленої методики представлена на рисунку 4.5.

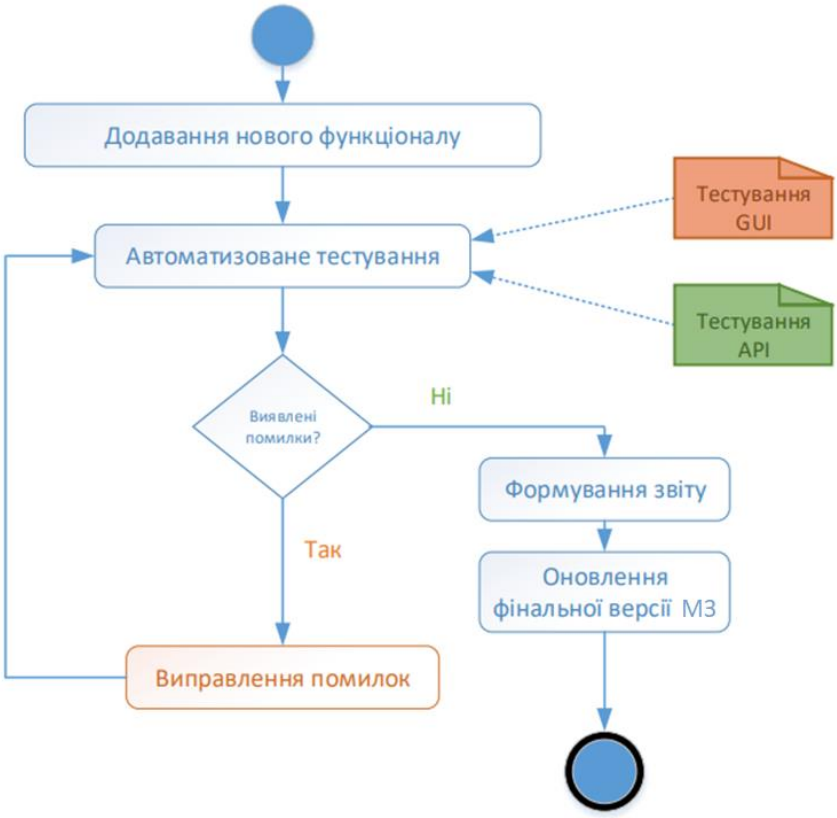

Рисунок 4.5 – Алгоритм проведення тестування із застосуванням розробленої методики

Далі описано апробацію розробленої методики для тестування GUI та API.

4.2 Апробація методик автоматизації для тестування GUI

Для запису автотестів із застосуванням методики Record and Play для МЗ використовують спеціальні драйвери, вбудовані в платформні засоби розробки. Для Android це драйвер Espresso у складі Android Studio, для iOS – драйвер XCTest, вбудований у Xcode.

Алгоритм створення таких автотестів однаковий для обох платформ:

Крок 1: у платформній ІСР вибрати реальний пристрій або емулятор для запуску програми, що тестується;

Крок 2: включити запис дій користувача, виконати вручну потрібний

тест;

Крок 3: переконатися, що запису, створеної драйвером, немає зайвих кроків;

Крок 4: після завершення запису вибрати опцію, яка запускає генерацію коду автотесту;

Крок 5: запустити автоматичне виконання підготовленого тесту та переконатися, що він працює коректним;

Крок 6: за допомогою фреймворку Allure згенерувати звіт про виконання автотесту.

У таблиці 4.2 детальніше описано оточення, яке було створено для підготовки та запуску автотестів із застосуванням методики Record and Play для iOS та Android.

Таблиця4.2 – Опис оточення для роботи з автотестами Record and Play

Платформа	iOS	Android
Операційна система	macOS	Windows
Застосунок	.swift файли з вихідним кодом програми, що виконується ipa.файл	.java файли з вихідним кодом програми, що виконується apk.файл
Мова розробки програми	Swift	Java
Середовище розробки	Xcode 11.4.1	Android Studio 3.6.6
Інструмент автоматизації	Надбудова Appium версія 1.15.1	Надбудова Appium 1.15.1
	Драйвер XCTest	Драйвери Espresso 3.2.0 та UIAutomator2

На рисинку 4.6 показані результати запуску автотестів Android Studio.

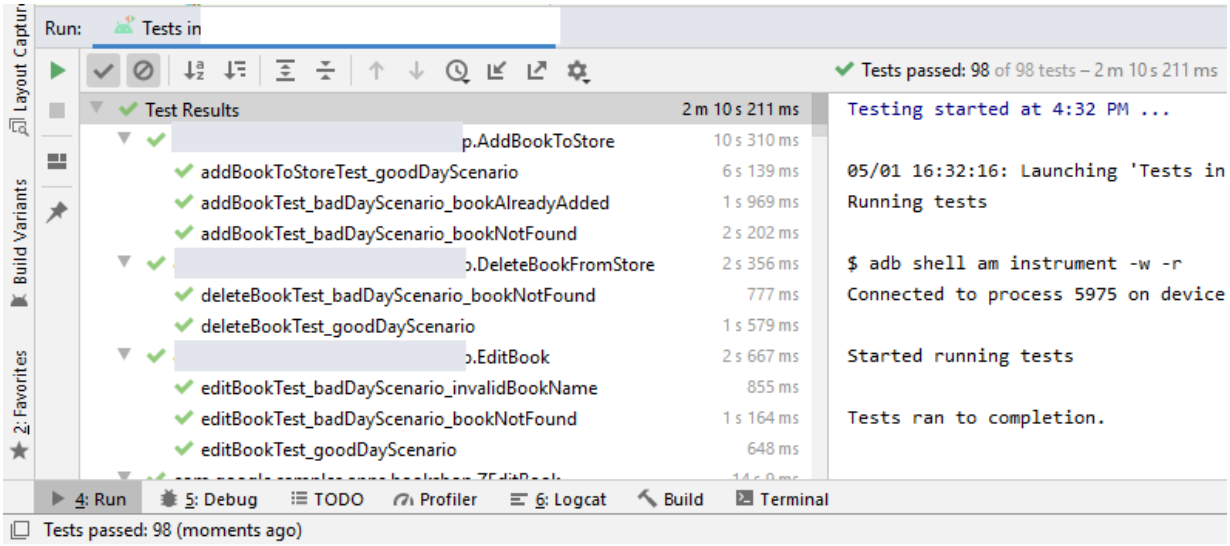

Рисунок 4.6 – результати запуску автотестів Record and Play в Android Studio

На рисунку 4.7 показано фрагмент автотесту, записаного у xCode.

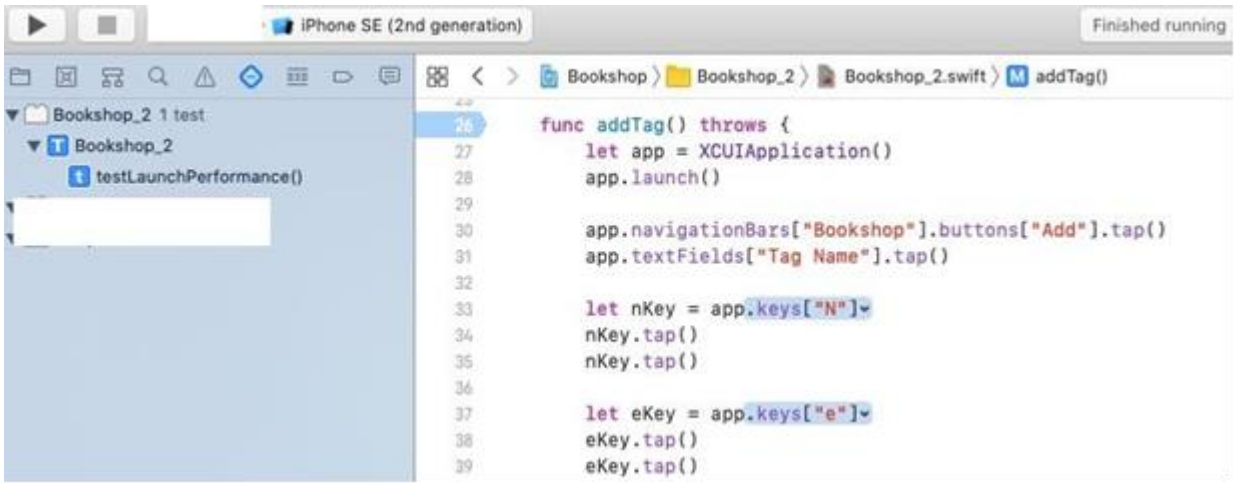

Рисунок 4.7 – робота з автотестами Record and Play у xCode

Приклад автотесту, записаного за допомогою Espresso у Android Studio.

Алгоритм роботи з Appium має такий вигляд.

Крок 1: запустити сервер Appium;

Крок 2: у налаштуваннях Appium вказати можливості (Capabilities), які будуть використані при запуску сесії – платформу, версію ОС, ім'я реального девайса або емулятора, назву потрібного драйвера та шлях до файлу для

тестованого МП;

Крок 3: розпочати сесію. Після підключення програми до девайсу та запуску МП перейти на потрібний екран та вибрати графічний елемент, для якого потрібно дізнатися локатор та (або) ідентифікатор.

Крок 4: скопіювати дані елемента раніше записані автотести, в яких задіяний цей елемент;

Крок 5: запустити оновлений тест та переконатися, що він працює коректно.

На рисунок 4.8 показано конфігурацію Appium для запуску драйвера UIAutomator.

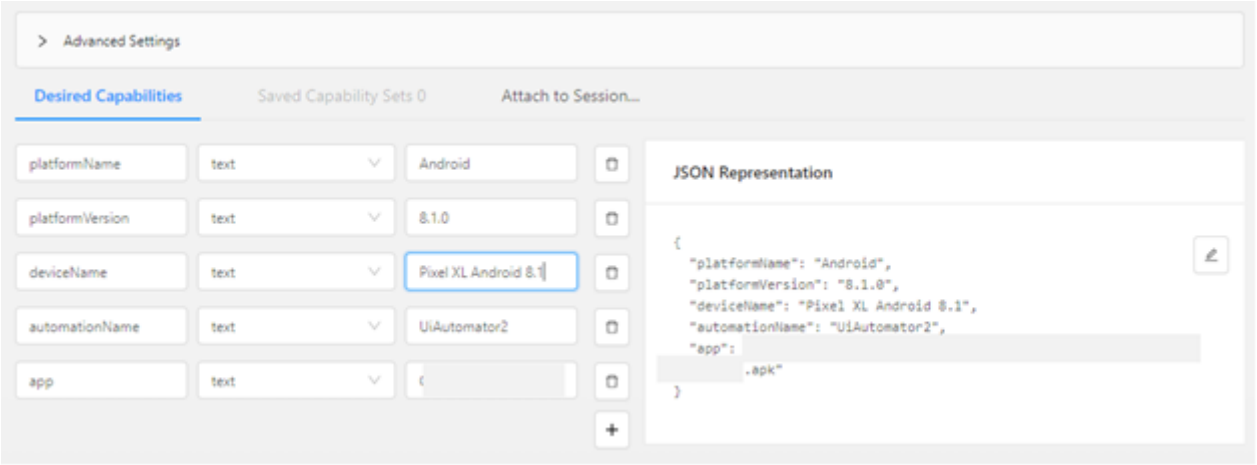

Рисунок 4.8 – Конфігурація Appium для запуску драйвера UIAutomator

На рисунок 4.9 показано відображення локаторів графічних елементів в інтерфейсі програми UIAutomator.

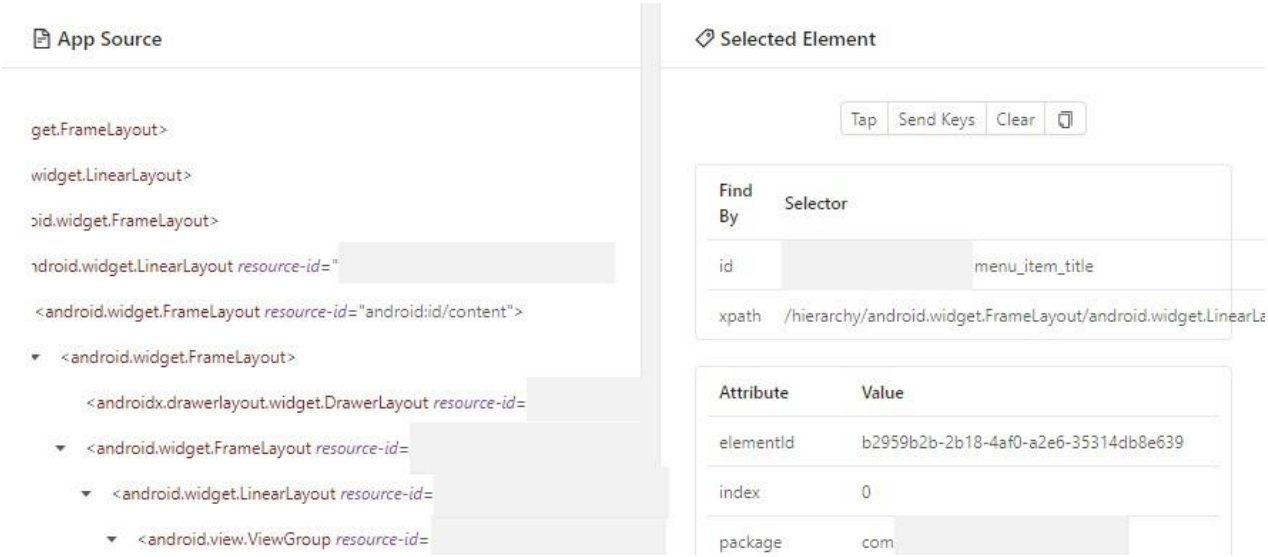

Рисунок 4.9 – Відображення локаторів елементів GUI в UIAutomator

Автотести, створені із застосуванням методики Record and Play, були виконані на всіх тестових пристроях із матриці девайсів, підготовленої під час проектування МП.

Це дозволило переконатися, нові елементи графічного інтерфейсу коректно функціонують усім популярних оточеннях.

4.3 Апробація методики для автоматизації тестування API

Код бекенду мобільного додатку (МЗ) є незалежним від мобільної платформи, що робить автотести API універсальними для операційних систем iOS та Android. Для виконання цих автотестів застосовується середовище розробки, яке підтримує мову програмування, використану для написання коду бекенду.

Відповідно до концепції BDD (Behavior-Driven Development), процес створення автотестів за методологією Scripting стартує з формулювання тестового сценарію. Опис сутностей та функціональних можливостей тестованого МЗ здійснюється з використанням ключових слів предметно-орієнтованої мови (DSL). Ця мова є однаково зрозумілою для розробників, фахівців з тестування та бізнес-аналітиків [6].

Оскільки бекенд МЗ, який досліджувався в експерименті, був реалізований на Java, для апробації методики було обрано середовище IntelliJ IDEA. Підготовку тестового сценарію в IntelliJ IDEA ілюструє рисунок 4.10.

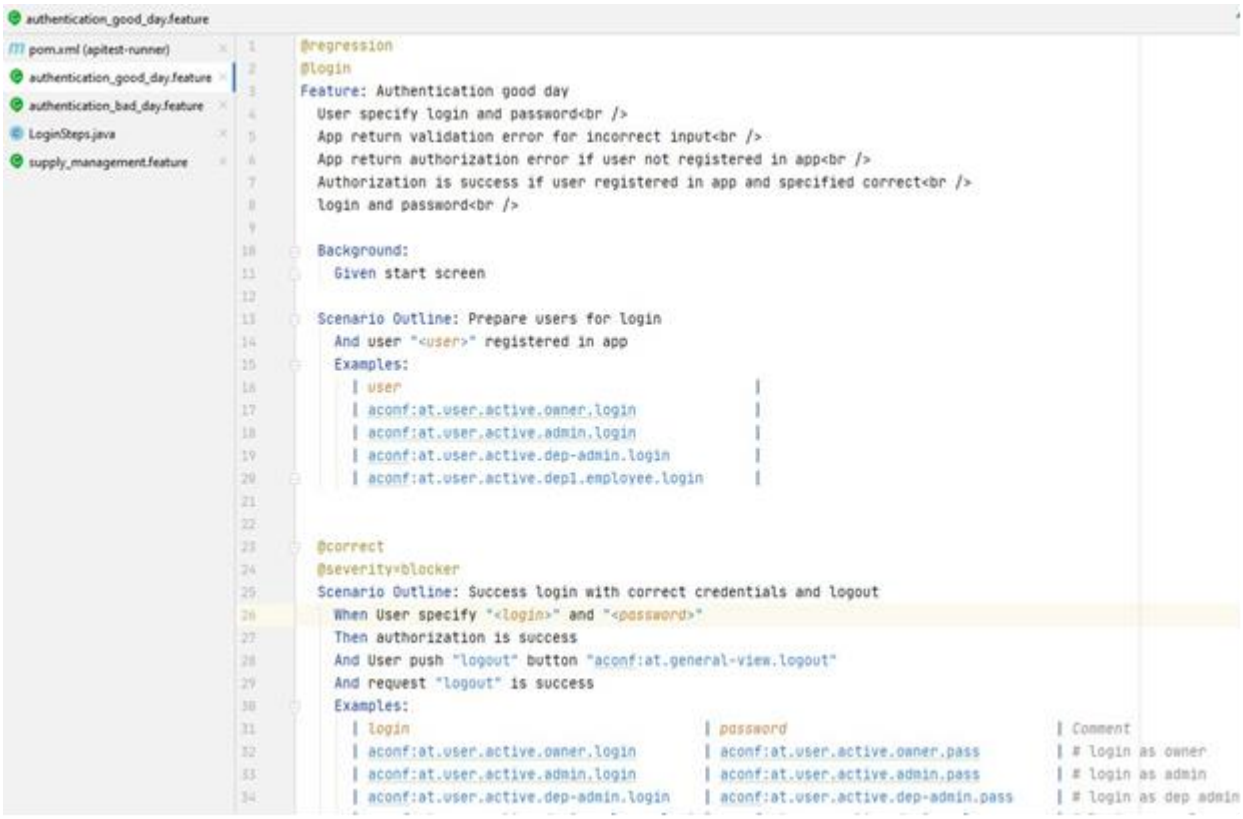

Рисунок 4.10 – Підготовка тестового сценарію IntelliJ IDEA

Алгоритм автоматизованого тестування із застосуванням методики Scripting складається з наступних кроків.

Крок 1: в ІСР створити сценарій тестування предметно-орієнтованою мовою Gherkin, сумісному з фреймворком Cucumber [2];

Крок 2: написати код автотесту з урахуванням підготовленого сценарію;

Крок 3: за допомогою фреймворку складання Maven налаштувати конфігурацію запуску сценаріїв;

Крок 4: запустити отриманий автотест серед розробки;

Крок 5: за допомогою фреймворку Allure згенерувати звіт про виконання автотесту.

Опис оточення для підготовки та запуску автотестів Scripting наведено в таблиці 4.3.

Таблиця 4.3 – Опис оточення для роботи з автотестами Scripting

Операційна система	Windows
Методологія	BDD
Середовище розробки	IntelleJ IDEA
Мова написання сценаріїв	Gherkin заснований
Мова написання коду автотестів	Java 8
Інструмент автоматизації	Фреймворк Cucumber 5.5.0, бібліотека cucumber-java 1.2.5,
	Фреймворк автоматичного збирання Maven 3.6.1

Далі наведено приклад тестового сценарію перевірки авторизації (Authentication) в МЗ з регресійного набору тестів. У сценарій включені як позитивні (correct), і негативні (fail) кейси.

Приклад більшого тестового сценарію для регресійного тестування наведено в додатку А. Нижче представлений код тесту, написаний на основі сценарію Authentication на мові Java8:

4.4 Результати застосування та оцінка ефективності розробленої методики

Результати виконання автоматизованого тестування МЗ, отримані під час апробації розробленої методики, представлені у таблиці 4.4.

Код бекенду мобільного додатку (МЗ) є незалежним від мобільної платформи, що робить автотести API універсальними для операційних систем iOS та Android. Для виконання цих автотестів застосовується середовище розробки, яке підтримує мову програмування, використану для

написання коду бекенду.

Таблиця 4.4 – Звіт про виконання автоматизованого тестування

Перевірка	Опис перевірки	Результат тестування
Тестування GUI	Перевірка наявності нових елементів графічного інтерфейсу та їх функціонування	Відповідає специфікації
Тестування GUI	Перевірка працездатності нових елементів GUI на різних пристроях	Виявленонезначні дефекти на деяких комбінаціях пристрій + платформа + версія ОС
Тестування API	Перевірка API-частини нової функціональності на відповідність специфікації	Відповідає специфікації
	Регресійна перевірка раніше створених функціональностей, порушених при додаванні нової функціональності	Виявленонезначні дефекти
	Тестування безпеки передачі даних під час надсилання нових запитів	Виявленонезначні дефекти, пов'язані з порушенням рольової моделі

Застосування методики дозволило:

- запустити автотести GUI на всіх тестових пристроях та завдяки цьому виявити дефекти, специфічні для конкретних девайсів та версій ОС;

- виконати тестування API з використанням комбінаторних даних, що допомогло оперативно виявити проблеми під час проходження менш

частотних сценаріїв, пов'язані з додаванням функціональності.

Для оцінки ефективності застосованої методики зроблено порівняння показників тестових метрик, отриманих до і після апробації розробленої методики. У результаті проведеного аналізу було виявлено такі зміни:

- тестове покриття збільшилося на 25% для всіх тестових пристроїв та на 60% для кожного конкретного пристрою;
- кількість не пройдених тест-кейсів зменшилася з 827 до 307 для всіх девайсів та з 1938 до 704 для окремих девайсів;
- кількість виявлених помилок збільшилася в 6 разів - з 12 до 72. Але зростання відбулося в основному за рахунок дефектів з незначним та тривіальним пріоритетом. Їхнє число склало 35 і 26 відповідно.

Така динаміка свідчить про те, що сценарії критичного шляху були вкриті до застосування автоматизації. Застосування розробленої методики дало змогу збільшити тестове покриття за рахунок запуску тестів на всіх тестових пристроях та виконання менш частотних сценаріїв, які при ручному тестуванні не перевірялися або перевірялися не повністю.

На рисунку 4.11 представлена діаграма, що ілюструє стартові та підсумкові показники метрики тестове покриття для всього набору тестових пристроїв і для кожного окремого пристрою.

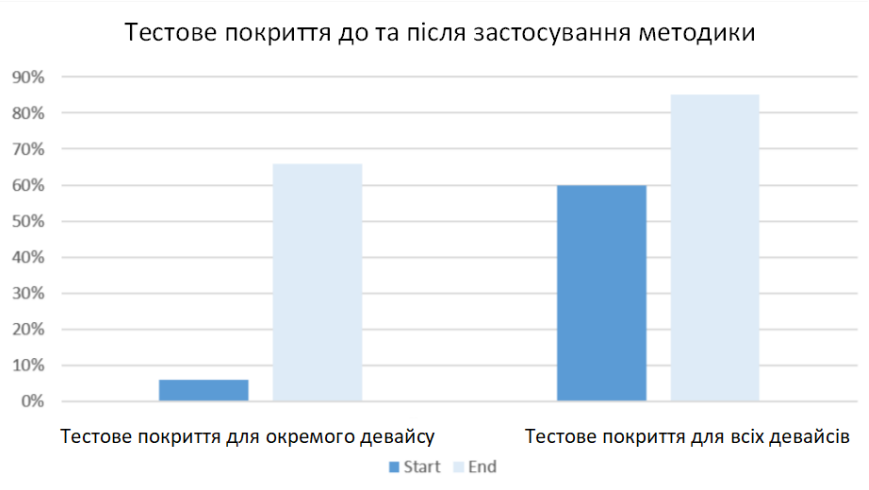

Рисунок 4.11 – Тестове покриття до та після застосування методики

На рисунку 4.12 представлений фрагмент звіту про виконання всіх підготовлених наборів автотестів API після ремонту виявлених дефектів, згенерований за допомогою фреймворку Allure.

Рисунок 4.12 – Фрагмент звіту про виконання автотестів API

На рисунку 4.13 показано фрагмент звіту про виконання конкретного автотесту.

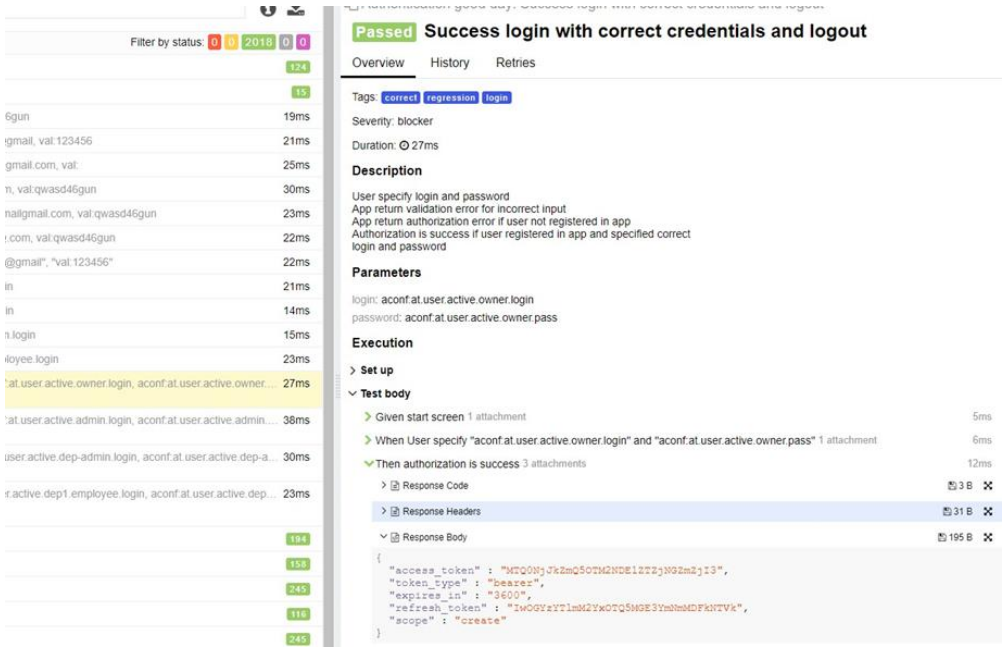

Рисунок 4.13 – Звіт про виконання автотесту «авторизація у МЗ»

Характер та якість змін доводять ефективність застосованої методики автоматизації тестування. Впровадження автотестів дозволило перевіряти більшість сценарних розгалужень протягом двотижневої Scrum-ітерації.

Таким чином, апробацію методики вважатимуться успішною.

ВИСНОВКИ

Метою цієї кваліфікаційної роботи є аналіз і створення ефективної методики автоматизації тестування мобільних застосунків (МЗ), що розробляються за підходом Agile.

У результаті проведеного дослідження було отримано такі ключові висновки та результати:

1.Проведено огляд літературних джерел за тематикою дослідження, який засвідчив недостатню кількість праць, присвячених питанням автоматизації тестування мобільних застосунків в умовах Agile-розробки.

2. Проаналізовано методику Scripting, що використовується для автоматизованого тестування в Agile-проєктах. Результати аналізу показали, що ця методика не забезпечує повного охоплення вимог до тестування мобільних додатків.

3. Запропоновано та реалізовано методику автоматизації тестування мобільних застосунків у рамках Agile, яка базується на комбінації двох підходів: використання методики Record and Play для тестування GUI та методики Scripting – для API-тестування.

4. Для реалізації методики були застосовані відповідні драйвери, інтегровані в середовища розробки Android Studio та Xcode, а також фреймворк Cucumber, який використовувався у поєднанні з середовищем IntelliJ IDEA.

З метою оцінки ефективності розробленого підходу було виконано порівняльний аналіз метрик тестування, отриманих до та після впровадження автоматизації.

Результати продемонстрували збільшення загального тестового покриття мобільного додатка на 25% для всіх тестових пристроїв та на 60% — для кожного окремого пристрою. Крім того, кількість виявлених дефектів зросла з 12 до 72, переважно за рахунок виявлення помилок із низьким

пріоритетом.

Ці зміни свідчать про суттєве підвищення ефективності процесу тестування після впровадження розробленої методики.

У межах роботи розроблено ефективну методику автоматизованого тестування мобільних застосунків, орієнтовану на застосування в Agile-проєктах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ministry of Health of Ukraine. (n.d.). Mobile phone, smartphone and mobile applications. https://moz.gov.ua/uk/mobilnij-telefon-smartfon-ta-mobilni-zastosunki.

2. Zhang, H., Wu, H., & Rountev, A. (2016). Automated test generation for detection of leaks in Android applications. Proceedings of the IEEE 11th International Workshop on Automated Software Testing, 64–70. https://doi.org/10.1109/AST.2016.7503712.

3. Jabbarvand, R., Sadeghi, A., Bagheri, H., & Malek, S. (2016). Energy-aware test-suite minimization for Android apps. Proceedings of the International Symposium on Software Testing and Analysis, 425–436. https://doi.org/10.1145/2931037.2931055.

4. Zhang, T., Gao, J., Cheng, J., & Uehara, T. (2015). Compatibility testing service for mobile applications. Proceedings of the IEEE Symposium on Service-Oriented System Engineering, 179–186. https://doi.org/10.1109/SOSE.2015.33.

5. Yeh, C.-C., Lu, H.-L., Chen, C.-Y., Khor, K.-K., & Huang, S.-K. (2014). CRAXDroid: Automatic Android system testing by selective symbolic execution. Proceedings of the IEEE 8th International Conference on Software Security and Reliability – Companion, 140–148. https://doi.org/10.1109/SERE-C.2014.31.

6. Adamsen, C. Q., Mezzetti, G., & Møller, A. (2015). Systematic execution of Android test suites in adverse conditions. Proceedings of the International Symposium on Software Testing and Analysis, 83–93. https://doi.org/10.1145/2771783.2771792.

7. Gao, J., Tsai, W.-T., Paul, R., Bai, X., & Uehara, T. (2014). Mobile testing-as-a-service (MTaaS): Infrastructures, issues, solutions and needs. Proceedings of the 15th International Symposium on High-Assurance Systems

Engineering, 158–167. https://doi.org/10.1109/HASE.2014.30.

8. Agile Alliance. Manifesto for Agile Software Development [Електронний ресурс]. – Режим доступу: https://agilemanifesto.org/.

9. GeeksforGeeks. What is Behavior-Driven Development.(BDD)? [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/ behavioral-driven-development-bdd-in-software-engineering/.