НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.434

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних та інформаційних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Програмне забезпечення для автоматичного тестування GUI на Android»

Виконав (-ла):
студент (-ка) ІІ курсу, групи ІП-21мн
Дубовик Андрій Павлович _____

Керівник:
доцент кафедри ІПІ, к. т. н., доц.,
Фіногенов Олексій Дмитрович _____

Рецензент:
доцент каф. НГІ та КГ, к.т.н., доц.,
Яблонський Петро Миколайович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних та інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2024р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Дубовик Андрій Павлович

1. Тема дисертації «Програмне забезпечення для автоматичного тестування GUI на Android», науковий керівник дисертації Фіногенов Олексій Дмитрович, к. т. н., доцент, затверджені наказом по університету від «23» березня 2024 р. № 1445-с.

2. Термін подання студентом дисертації «27» травня 2024 р.

3. Об’єкт дослідження – процеси забезпечення якості програмного забезпечення.

4. Предмет дослідження – автоматизоване генерування тестових сценаріїв для графічного інтерфейсу, розробленого з використанням декларативної парадигми.

5. Перелік завдань, які потрібно розробити – аналіз існуючих рішень та виявлення їх недоліків; формулювання вимог; огляд та аналіз методів та алгоритмів, що застосовуються під час вирішення аналогічних задач; проектування ПЗ; написання ПЗ; тестування розробленого ПЗ; оцінка ефективності запропонованого рішення.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 7 таблиць, 15 рисунків.

7. Орієнтовний перелік публікацій – дві публікації

8.Дата видачі завдання «15» березня 2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Ознайомлення з предметною областю, вивчення літератури	30.11.2022	
2	Аналіз наявних методів та засобів автоматизованого тестування	28.02.2023	
3	Постановка та формалізація задачі дослідження	15.03.2023	
4	Аналіз вимог до програмного забезпечення	29.03.2023	
5	Розробка методів та засобів для розв’язання поставлених задач	31.05.2023	
6	Розробка архітектури програмного забезпечення	25.10.2023	
7	Виконання експериментальних досліджень	20.12.2023	
8	Оформлення пояснювальної записки	20.04.2024	
9	Подання дисертації на попередній захист	11.05.2024	
10	Подання дисертації на рецензію	16.05.2024	
11	Подання дисертації на захист	27.05.2024	

Студент Андрій ДУБОВИК

Науковий керівник Олексій ФІНОГЕНОВ

РЕФЕРАТ

Магістерська дисертація: 104 с., 15 рис., 7 табл., 3 додатки, 68 джерел

Актуальність теми. Тестування є важливим етапом розробки програмного забезпечення та здійснюється або вручну, або за допомогою спеціального програмного забезпечення. Проте ручне тестування зазвичай є довготривалим, вимагає виділення додаткового часу, тому актуальною задачею є дослідження та створення методів та засобів для автоматизації тестування. Графічний інтерфейс користувача є однією з найважливіших частин додатку, оскільки саме від нього залежить кінцеве ставлення користувача до програмного продукту. З часом, технології, що використовуються для розробки інтерфейсів, втрачають актуальність, а їм на заміну приходять нові. В контексті розробки Android-додатків такою технологією є декларативна парадигма програмування, яка тепер застосовується для розробки інтерфейсів та мова програмування Kotlin. Актуальність цих досліджень полягає у тому, що, існуючі інтегровані середовища та інструменти розробки не підтримують автоматичне генерування коду модульних тестів для GUI, розроблених за допомогою мови програмування Kotlin та нової парадигми, що унеможливлює використання цих інструментів під час проведення тестування, тож вимагає проектування нових засобів, що відповідали б вимогам сучасності.

Мета досліджень. Розширення рамок використання інтегрованих середовищ розробки під Android шляхом підтримки автоматизованого генерування коду сценаріїв модульних тестів графічного інтерфейсу для декларативної парадигми, що дозволить зменшити час на розробку тестів.

Для досягнення поставленої мети було сформовано **наступні завдання**:

– дослідити наявні методи, алгоритми та підходи до побудови сценаріїв автоматизованого тестування графічного інтерфейсу користувача;

– дослідити наявні рішення та засоби, та виявити їх недоліки;

– удосконалити методи та засоби генерування модульних тестів для досягнення максимального покриття коду, розширивши їх на підтримку декларативної парадигми;

– провести проектування та розробити плагін до інтегрованого середовища розробки Android Studio для автоматичної генерації коду сценаріїв unit-тестів;

– провести оцінку ефективності запропонованого рішення.

Об’єкт досліджень. Програмне забезпечення для автоматичної генерації коду тестів GUI Android.

Предмет досліджень. Процеси забезпечення якості програмного забезпечення.

Методи досліджень. Емпіричні дослідження, системний аналіз, абстрагування, структурно-генетичний аналіз, розробка програмного забезпечення.

Наукова новизна даної роботи полягає в удосконаленні методу створення сценаріїв автоматизованого тестування, що відрізняється від існуючих аналогів врахуванням всіх можливих викликів елементів інтерфейсу та комбінацій множин даних та наявністю підтримки автоматизованого генерування коду модульних тестів GUI, розробленого з використанням декларативної парадигми та фреймворку Jetpack Compose, чого раніше не було в середовищах розробки додатків під ОС Android.

Практична новизна полягає у створенні плагіну модульного тестування для IDE Android Studio, що дає можливість збільшити покриття компоненту тестами та значно зекономити час розробників.

Апробація. Результати роботи доповідались на VI Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології SoftTech-2023».

Публікації. Результати роботи над магістерською дисертацією опубліковані на V та VI Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології SoftTech-2023».

Ключові слова: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, GUI, ANDROID, KOTLIN, JETPACK COMPOSE

ABSTRACT

Topic: «Software for automatic GUI testing on Android»

Master's degree thesis: 104 pages, 15 figures, 7 tables, 3 attachments, 68 sources

Relevance of the topic. Testing is an important stage of software development and it is performed either manually or using the special software. However, manual testing is usually time-consuming and requires additional time, so it is an actual task to research and create methods and tools for automating testing. The graphical user interface is the crucial part of the application, since it determines the final attitude of the user to the software product. Over time, the technologies used to develop interfaces lose their relevance and are replaced by new ones. In the context of Android application development, such technology is the declarative programming paradigm, which is now used for interface development along with the Kotlin programming language. The relevance of this research lies in the fact that existing IDEs and tools do not support the automatic generation of unit test code for GUIs, developed using the Kotlin and the new paradigm, which makes it impossible to use these tools during testing, so it requires the design of new tools that would meet the current requirements.

Research objective. Expanding the scope of using IDEs for Android by supporting automated code generation of the GUI unit-tests for the declarative paradigm, which will reduce the time for tests development.

The following tasks were formulated to achieve this research objective:

- study the existing methods, algorithms and approaches to building automated testing scenarios for the graphical user interface;

- study existing solutions and tools and identify their shortcomings;

- improve the methods and tools for generating unit tests to achieve maximum code coverage by extending them to support the declarative;

- design and develop a plugin for the IDE Android Studio for automatic code generation of unit test scenarios;

- evaluate the effectiveness of the proposed solution.

The object of research. Software for automatic generation of Android GUI test code.

The subject of research. Software quality assurance processes.

The scientific innovation of the obtained results lies in improvement of the automated testing scenarios creation method, that differs from existing analogues by taking into account all possible interface element events and combinations of data sets, and by the availability of support for automated GUI unit test code generation, developed using the declarative paradigm and the Jetpack Compose framework, which was not previously available in Android application development environments.

The practical innovation lies in the creation of a unit testing plugin for the Android Studio IDE, which allows testing and generating tests the declarative paradigm of GUI development of Android applications and makes it possible to increase the coverage of a component with tests and significantly save developers' time.

Approbation. The results of the work were presented at the VI International Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies SoftTech-2024".

Publications. The results of the work on the master's thesis were published at the V and VI International Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies SoftTech-2024".

Keywords: AUTOMATED TESTING, GUI, ANDROID, KOTLIN, JETPACK COMPOSE

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ	– програмне забезпечення;
GUI	(graphic user interface) – графічний інтерфейс користувача;
API	(application programming interface) – програмний інтерфейс;
IDE	(integrated development environment) – інтегроване середовище розробки;
AC	(Acceptance Criteria) – критерії прийняття;
DI	(Dependency Injection) – вбудовування залежностей;
UI	(User Interface) – інтерфейс користувача;
RL	– глибоке навчання з підкріпленням;
ОС	– операційна система;
Android	– операційна система для мобільних пристроїв;
Activity	– це компонент додатка Android, який відображає користувацький інтерфейс на екрані пристрою та обробляє взаємодію з користувачем.

ВСТУП

Протягом останніх десятиліть спостерігається швидкий розвиток мобільних технологій, що призводить до випуску на ринок безлічі нових мобільних додатків. Це пояснюється здатністю мобільних додатків задовольняти різноманітні потреби користувачів та надавати доступ до онлайн сервісів у будь-який час та у будь-якому місці.

Одним з основних аспектів якості мобільних додатків є графічний інтерфейс, оскільки від його якості залежить загальне враження користувача від використання додатку та досвід користування. Через велику різноманітність пристроїв, конфігурацій та розмірів екранів, тестування графічного інтерфейсу додатків для Android стає викликом.

Unit тестування використовує ітеративний підхід і добре зарекомендувало себе у покращенні результатів розробки. Проведення тестів на ранніх стадіях життєвого циклу розробки є найефективнішою стратегією для виявлення та помилок. Нижче наведено кілька ключових переваг unit-тестування:

1. підвищення гнучкості в Agile процесах: Agile методології покладаються на ефективні, повторювані та автоматизовані набори тестів, щоб гарантувати, що кожна ітерація проходить безперебійно, без перешкод у вигляді тривалих циклів тестування;

2. підвищення якості та безпеки: хоча команди розробників програмного забезпечення визнають важливість тестування для забезпечення відповідності програмного забезпечення вимогам, вони часто не проводять належного тестування або відкладають його на пізніші етапи розробки;

3. зниження витрат у довгостроковій перспективі: дефекти програмного забезпечення, виявлені після релізу, можуть спричинити значно більші витрати порівняно з тими, що були виявлені під час розробки.

Незважаючи на ці переваги, розробники продовжують стикатися з проблемами при проведенні unit-тестування, незважаючи на їх прагнення до покращення результатів. Ці проблеми включають в себе наступне:

– створення тестів вимагає додаткових зусиль і часто сприймається як нудна робота, а балансування між створенням комплексного набору тестів та цілями і термінами проекту створює дилему для команд розробників;

– обслуговування тестів виявляється дорогим: подібно до коду, модульні тести потребують підтримки, оскільки будь-які зміни в кодовій базі можуть вимагати відповідних коригувань у відповідних тестах;

– імітація та ізоляція блоків, що тестуються, викликає труднощі та займає багато часу, а належна ізоляція блоків, що тестуються, має вирішальне значення, але її досягнення вимагає імітації залежностей, що може бути трудомістким процесом.

Щоб скористатися перевагами комплексного модульного тестування, команди розробників програмного забезпечення повинні вирішити проблеми, пов'язані зі створенням, ізоляцією та підтримкою тестів, а автоматизація слугує рішенням.

З часом, технології, що використовуються для розробки інтерфейсів, втрачають актуальність, а їм на заміну приходять нові. Прикладом такої технології в контексті розробки Android-додатків є декларативна парадигма програмування, яка тепер застосовується і до розробки інтерфейсів, та мова програмування Kotlin. І хоча поява обох технологій значно спростила розробку графічного інтерфейсу користувача, та, на жаль, існуючі інтегровані середовища та інструменти розробки не підтримують автоматичне генерування коду модульних тестів для графічних інтерфейсів користувача, розроблених за допомогою мови програмування Kotlin та декларативної парадигми.

Тому метою даної роботи є розширення рамок використання інтегрованих середовищ розробки під Android шляхом підтримки автоматизованого генерування коду сценаріїв модульних тестів графічного інтерфейсу для декларативної парадигми, що дозволить зменшити час на розробку тестів.

Актуальність цих досліджень полягає у тому, що, поява нових мов та парадигм програмування з часом унеможливлює розширення використання

наявних інструментів, тож вимагає проектування нових засобів, що відповідали б вимогам сучасності.

Об’єктом дослідження є програмне забезпечення для автоматичної генерації коду тестів GUI Android.

Предметом дослідження є процеси забезпечення якості програмного забезпечення.

Для досягнення мети магістерської дисертації необхідно виконати наступні завдання:

– дослідити наявні методи, алгоритми та підходи до побудови сценаріїв автоматизованого тестування графічного інтерфейсу користувача;

– дослідити наявні рішення та засоби, та виявити їх недоліки;

– розширити рамки використання інтегрованих середовищ розробки під Android шляхом підтримки коду сучасних мов програмування та автоматизації генерації коду сценаріїв unit-тестів для них;

– провести проектування та розробити плагін до інтегрованого середовища розробки Android Studio для автоматичної генерації коду сценаріїв unit-тестів;

– провести оцінку ефективності запропонованого рішення.

1 ОГЛЯД МЕТОДІВ ТА ДОСЛІДЖЕНЬ

1.1 Опис предметної області

З часів появи операційної системи Android для розробки інтерфейсів користувача використовувався імперативний підхід. Він включав в себе використання XML-файлів, де описувались макети екранів, управління взаємодією та наслідування, де кожен компонент інтерфейсу мав наслідуватись від абстрактного класу View. Хоча цей метод був популярним, він мав свої недоліки. Розробники часто стикалися з наступними проблемами:

1. велика кількість коду: використання XML для розмітки користувацького інтерфейсу вимагає великої кількості коду, а розмітка UI може займати значний дисковий простір, особливо у великих проектах;
2. складність в обслуговуванні коду: робота з великими XML-файлами і безліччю View-компонентів є складною, а пошук і виправлення помилок або внесення змін є трудомістким;
3. обмежене перевикористання: за традиційного підходу важко домогтися високого ступеня перевикористання компонентів інтерфейсу через складність структури XML і View-компонентів;
4. труднощі анімацій і складних ефектів: для реалізації складних анімацій і ефектів потрібно більше зусиль, і часто це робиться з використанням Java або Kotlin-коду, що ускладнює їхнє створення і підтримку;
5. незручність роботи з адаптивністю: створення адаптивних макетів для різних розмірів екранів і орієнтацій може бути складним і вимагати великої кількості ресурсів.

Ускладнення проектів з часом змушувало розробників витрачати все більше часу на їх підтримку та впровадження нового функціоналу. Тому, для вирішення цих проблем компанія Google представила новий фреймворк для розробки графічного інтерфейсу користувача – Jetpack Compose [3]. Його перевагами стали:

– декларативний підхід: Jetpack Compose надає декларативний спосіб визначення користувацького інтерфейсу, тобто розробник описує, який вигляд має мати інтерфейс у кінцевому стані, а ядро Compose займається рендерингом;

– менше коду: функціональний стиль дає змогу створювати UI з меншою кількістю коду порівняно з традиційним XML / View-підходом;

– легке перевикористання: функціональні компоненти легко використовувати повторно завдяки їхній декларативній природі, що сприяє створенню більш чистого і модульного коду;

– простота роботи з анімаціями: Compose надає більш зручні засоби для роботи з анімаціями і складними ефектами, що робить їх більш доступними і підтримуваними;

– покращена підтримка адаптивності: Compose спрощує створення адаптивних інтерфейсів за допомогою функцій, таких як Modifier.fillMaxSize, що автоматично адаптуються до розмірів екрана;

– вищий рівень абстракції: функціональний підхід надає вищий рівень абстракції, що спрощує розуміння та підтримання коду, особливо для нових розробників, які можуть бути більш знайомі з декларативним стилем;

– використання мови Kotlin: Jetpack Compose використовує Kotlin, яка стає дедалі популярнішою та офіційно рекомендується компанією Google для розробки Android-додатків. Використання мови Kotlin надає такі додаткові переваги, як-от безпечніший і зручніший синтаксис, а також можливості роботи з розширеними функціональними можливостями мови.

Представлення Jetpack Compose вирішило проблеми традиційного підходу і внесло революційні зміни у побудову графічного інтерфейсу Android-додатків, полегшуючи розробку та обслуговування коду і роблячи його більш зрозумілим та ефективним.

Composable-функції представляють собою ключовий елемент Jetpack Compose, дозволяючи описувати інтерфейс додатка як функціональні компоненти [4]. Однак, на відміну від традиційних Views, Composable-функції позбавлені унікальних ідентифікаторів, що ускладнює процес тестування.

Відсутність таких явних ідентифікаторів може зробити генерацію тестових сценаріїв менш ефективною, а також ускладнює реалізацію тестових випадків.

Враховуючи важливість тестування для сучасних додатків, компанія Google надає необхідний фреймворк для написання тестів та тестування Composable-функцій. Проте, тестування Composable-функцій вимагає розуміння їхньої структури та взаємодій, що відрізняється від традиційних підходів. Це означає, що для генерації тестів необхідно більше аналізу та розуміння самого коду Compose, а не лише його візуального представлення.

Kotlin вважається сучасною, виразною та безпечнішою мовою програмування у порівнянні з Java [5]. Окрім більш лаконічного синтаксису, Kotlin відрізняється від Java у кількох аспектах, включаючи нульові та ненульові типи, data-класи та виведення типів. Варто зазначити, що Kotlin було створено з акцентом на сумісність з Java, що дозволяє легко викликати код на Java з Kotlin і навпаки. Зокрема, на платформі Android, Kotlin компілюється в той самий байт-код, що і Java, забезпечуючи повну сумісність між двома мовами.

Після того, як Kotlin стала офіційно підтримуваною мовою програмування для Android, вона пережила сплеск популярності. У 2018 році вона стала найбільш швидкозростаючою мовою на GitHub і зберегла свою позицію четвертої найпопулярнішої мови у наступних роках [6]. За даними Google, майже 60% з 1000 найкращих додатків для Android містять код на Kotlin [7], тоді як AppBrain повідомляє про частку ринку 75,95% серед 500 найкращих додатків у США та загальну частку 15,03%, при цьому понад 125 000 додатків використовують Kotlin [8].

Дійсно, ці цифри є вражаючими для відносно молодої мови. Однак, схоже, що Kotlin недостатньо представлений у публікаціях спільноти інженерів-програмістів, особливо в дослідженнях, присвячених Android. Для ілюстрації цього, було проведено дослідження, аналогічне до [9], на згадки про Kotlin або Java у публікаціях, присвячених Android, на авторитетних конференціях (зокрема, ICSE, MSR, SANER та MOBILESoft) та у наукових статтях між 2018 та 2023 роками, результати якого зведено у табл. 1.

Таблиця 1.1 – Згадки про мов програмування Kotlin та Java у публікаціях

	2018	2019	2020	2021	2022	2023
Java	129	118	100	125	99	76
Kotlin	2	1	5	15	17	20

Графічне представлення даного дослідження наведено на рис. 1.1.

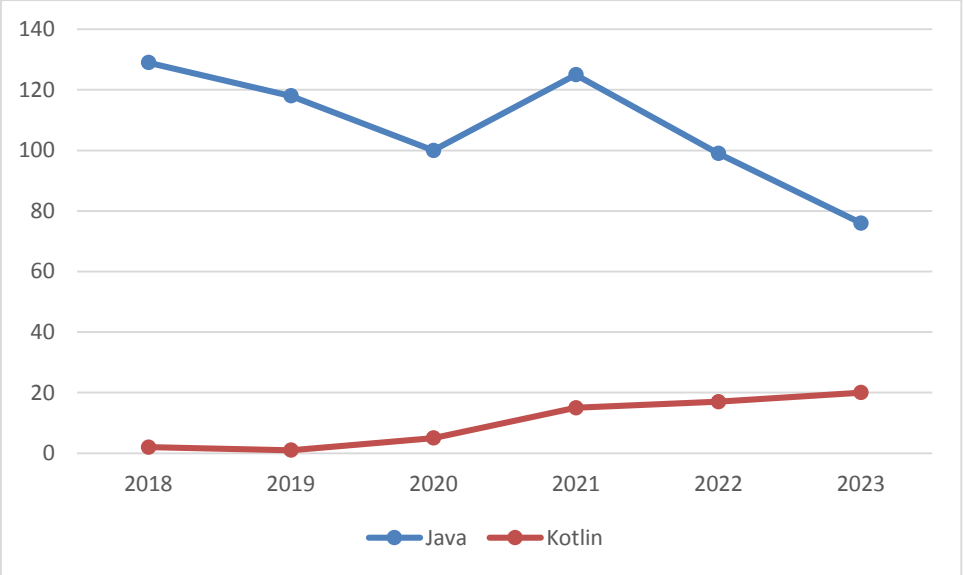

Рисунок 1.1 – Графік кількості згадок мов програмування Kotlin та Java

Як показують результати, Kotlin згадується у публікаціях стосовно Android-розробки все ще досить нечасто. При цьому чітко простежується тренд на зростання кількостей публікацій та згадок цієї мови програмування. Це підкреслює, що Kotlin часто ігнорується навіть тоді, коли його актуальність може бути значною.

На противагу Kotlin, Java згадується набагато частіше, проте останнім часом тренд на її використання помітно зменшується на користь Kotlin.

1.2 Огляд різновидів тестування

Тестування мобільних додатків стало невід'ємною частиною розробки ПЗ. Загалом тестування графічного інтерфейсу мобільних додатків поділяється на

такі види тестування, як тестування білого, чорного та сірого ящиків, оскільки ці види тестування відіграють критичну роль у забезпеченні якості та надійності додатків. Розглянемо ці різновиди тестування графічного інтерфейсу Android-додатків, а також детально розглянемо призначення, методи, переваги та недоліки кожного виду тестування і опишемо приклади інструментів та підходів, які можуть бути використані для кожного з них.

1.2.1 Тестування "білого ящика"

Тестування "білого ящика" або тестування на рівні вихідного коду – відноситься до методів тестування програмного забезпечення, де внутрішня структура та реалізація програмного коду вивчається та аналізується для розробки тестів. Основна мета полягає в перевірці правильності роботи окремих компонентів програми. У контексті графічного інтерфейсу Android-додатків, тестування білого ящика включає в себе аналіз джерел коду, модулів та функцій, щоб перевірити їхню правильність та працездатність.

Основні методи тестування "білого ящика" включають:

– модульне тестування: застосування технік, які дозволяють тестувати окремі модулі або функції програми без залежності від інших компонентів;

– unit-тестування: написання тестів для перевірки коректності роботи окремих функцій чи методів. Для Android-додатків це може включати тестування логіки бізнес-шару додатку або методів управління даними;

– метод покриття коду: цей метод визначає рівень покриття програмного коду тестами. У контексті графічного інтерфейсу Android-додатків, це означає, що кожен елемент інтерфейсу, функція та модуль коду додатка має бути протестований;

– тестування границь: цей метод визначає значення вхідних даних, які використовуються для тестування, так, щоб вони включали як найбільше, так і найменше значення. У контексті графічного інтерфейсу Android-додатків, це може означати тестування елементів інтерфейсу з різними розмірами та роздільною здатністю екрану;

– тестування шляхів програми (англ. path testing): цей метод тестування оцінює різні шляхи в програмі та перевіряє їхню правильність. У контексті графічного інтерфейсу Android-додатків, це може означати перевірку різних шляхів взаємодії користувача з додатком.

Основними інструментами розробки під час виконання тестування білого ящика виступають інтегровані середовища розробки з можливістю виконання та налагодження тестів, такі, як Android Studio.

До переваг тестування білого ящику можна віднести:

– глибоке тестування коду: дозволяє виявити помилки, які можуть залишитися невидимими за інших методів тестування;

– раннє виявлення помилок: дозволяє виявити та виправити помилки на ранніх етапах розробки, що може значно скоротити час витрачений на налагодження.

До недоліків можна віднести:

– обмежену охоплюваність: не всі аспекти програми можуть бути покриті тестами на рівні вихідного коду;

– значну кількість коду: може знадобитися значна кількість тестового коду для охоплення всіх можливих сценаріїв.

До найпопулярніших інструментів платформи Andriod, що дозволяють розробляти код тестів білого ящика, відносять:

– JUnit [10]: фреймворк для написання та виконання тестів мовою програмування Java.

– Mockito [11]: бібліотека для створення та використання підроблених (англ. mock) об'єктів під час модульного тестування.

– Robolectric [12]: фреймворк для тестування Android-додатків на рівні вихідного коду без необхідності запуску на справжньому пристрої.

1.2.2 Тестування "чорного ящика"

Тестування чорного ящика спрямоване на перевірку функціональності додатка без знання внутрішньої реалізації та відповідає методам тестування

програмного забезпечення, де внутрішня структура програми не розглядається. Замість цього, тестування зосереджується на функціональності програми та виходить з припущення, що тестувальник не знає деталей її внутрішньої реалізації.

Основні методи тестування чорного ящика включають:

– функціональне тестування: цей метод перевіряє, чи працює програма згідно з очікуваним функціоналом. У контексті графічного інтерфейсу Android-додатків, це може включати тестування функціональності елементів інтерфейсу, таких як кнопки, поля введення та меню;

– тестування введення/виведення (англ. I/O testing): цей метод тестує вхідні та вихідні дані програми, перевіряючи, чи обробляє вона коректно введені дані та чи виводить відповідний результат. У контексті графічного інтерфейсу Android-додатків, це може означати тестування коректності введення користувача через сенсорний екран та виведення відповідних результатів;

– тестування користувацьких сценаріїв: створення тестових сценаріїв, які відображають поведінку користувача під час використання додатка;

– автоматизоване тестування інтерфейсу користувача: використання інструментів, таких як Appium [13] або Espresso [14], для автоматизації тестування графічного інтерфейсу додатка.

До переваг тестування чорного ящика можна віднести:

– зовнішній погляд: тестування з точки зору кінцевого користувача дозволяє виявити проблеми, які можуть бути невидимі з точки зору розробника;

– відсутність знання внутрішньої реалізації: тестування не потребує глибокого розуміння внутрішньої структури програми.

До недоліків можна віднести:

– обмежене покриття: тестування з точки зору кінцевого користувача може не виявити всі можливі проблеми з функціональністю;

– високі витрати на час: ручне тестування може бути часо- і ресурсомістким процесом, особливо для великих додатків.

З найпопулярніших інструментів платформи Andriod, що дозволяють виконувати тестування чорного ящика, можна виокремити наступні:

– Appium [13]: цей інструмент автоматизованого тестування дозволяє тестувати графічний інтерфейс Android-додатків на різних пристроях і платформах;

– Espresso [14]: це фреймворк для автоматизованого тестування інтерфейсу користувача Android-додатків. Він надає зручний API для виконання тестів, які взаємодіють з елементами інтерфейсу;

– UI Automator [15]: інструмент для створення тестів, які взаємодіють з різними елементами інтерфейсу користувача Android-додатків на рівні користувацького інтерфейсу.

1.2.3 Тестування "сірого ящика"

Тестування "сірого ящика" поєднує в собі елементи як тестування "білого", так і "чорного ящиків". Воно використовує знання про внутрішню структуру програми, але також перевіряє її функціональність без детального аналізу коду. У контексті графічного інтерфейсу Android-додатків, це означає, що тестування "сірого ящика" використовує комбінацію аспектів тестування "білого" та "чорного ящиків" для забезпечення якості додатків.

Основні методи тестування "сірого ящика" включають:

– тестування з урахуванням специфіки (англ. domain testing): цей метод тестує програму з урахуванням особливостей області застосування. У контексті графічного інтерфейсу Android-додатків, це означає, що тестування враховує специфіку взаємодії з користувачем через сенсорний екран та інші функції пристрою;

– тестування реального середовища (англ. real environment testing): цей метод тестує програму в реальних умовах використання. У контексті графічного інтерфейсу Android-додатків, це може включати тестування додатка на різних пристроях та версіях операційної системи Android;

– інтеграційне тестування: перевірка взаємодії різних компонентів додатка та їх спільної роботи;

– тестування сценаріїв: тестування програми за допомогою певних сценаріїв використання, які можуть включати в себе як користувальницькі дії, так і системні взаємодії;

– компонентне тестування: перевірка роботи окремих компонентів додатка, таких як сервіси, ресурси, мережеві запити тощо.

До переваг тестування сірого ящика відносять:

– поєднання переваг: дозволяє використовувати переваги як тестування білого, так і тестування чорного ящика.

– більша охоплюваність: дозволяє тестувати не лише окремі компоненти, але і їх взаємодію.

До недоліків тестування сірого ящика відносять:

– складність: таке тестування вимагає певного рівня знань про внутрішню структуру програми для ефективного проведення тестування.

– високі витрати на ресурси: таке тестування може вимагати значних ресурсів, особливо при великих обсягах програмного коду.

До інструментів для виконання тестування сірого ящика можна віднести:

– Calabash [16]: цей фреймворк для автоматизованого тестування мобільних додатків дозволяє виконувати тести як на Android, так і на iOS;

– MonkeyRunner [17]: цей інструмент для автоматизованого тестування Android-додатків надає можливість скриптованого взаємодії з додатком на рівні користувацького інтерфейсу;

– Robotium [18]: цей фреймворк для автоматизованого тестування Android-додатків дозволяє легко створювати тести на рівні інтерфейсу користувача.

Загалом, тестування "білого", "чорного" та "сірого ящиків" є критично важливими методами тестування в контексті розробки графічного інтерфейсу Android-додатків. Кожен з описаних методів має свої переваги та обмеження, і застосування комплексного підходу до тестування може значно підвищити

якість та ефективність процесу розробки мобільних додатків. Ретельне тестування допомагає уникнути помилок та недоліків, що можуть виникнути в процесі використання додатків користувачами [19].

1.3 Огляд моделей тестування

Для ефективного тестування необхідно мати чітку стратегію, яка допомагає підтримувати стабільність та надійність програми. Однією з найпоширеніших та ефективних стратегій тестування є класична піраміда тестування, введена Майком Коном [20].

Піраміда тестування – це модель, яка визначає співвідношення між різними рівнями тестування, базуючись на їх вартості, складності та часових затратах. Ця модель розроблена з метою оптимізації тестових зусиль та мінімізації ризиків перед випуском продукту. Вона складається з трьох рівнів:

1. нижній рівень або модульне тестування: на нижньому рівні піраміди розташовані тести одиниць або модулів. Ці тести перевіряють правильність роботи окремих функцій, методів та класів програми на найнижчому рівні. Вони зазвичай написані розробниками під час написання коду та використовуються для швидкого виявлення та виправлення помилок;

2. середній рівень (функціональне або сервісне тестування): на цьому рівні піраміди розташовані функціональні тести. Ці тести перевіряють правильність роботи окремих функцій та модулів програми. Вони включають у себе тестування API, бізнес-логіки та інші ключові функції продукту;

3. верхній рівень (UI тестування): на вершині піраміди знаходяться тести графічного інтерфейсу користувача. Ці тести перевіряють коректність роботи графічного інтерфейсу програми та взаємодію з користувачем. Вони спрямовані на перевірку функціональних можливостей програми з погляду кінцевого користувача.

Класична піраміда тестування показує, що найбільша кількість тестів повинна бути сконцентрована на нижніх рівнях, тобто на одиницях та функціональних тестах. Це дозволяє виявити помилки на ранніх етапах розробки

та зменшити їх вплив на більш високих рівнях. На відміну від цього, UI тести, як правило, менш кількісно представлені, але вони перевіряють інтерфейс з точки зору кінцевого користувача, що робить їх не менш важливими. Розглянемо детальніше нижній та верхній рівні класичної піраміди, тобто модульне тестування та тестування графічного інтерфейсу.

1.3.1 Модульне тестування

Модульне тестування є фундаментальним рівнем тестування програмного забезпечення, який дозволяє перевіряти правильність роботи окремих модулів або компонентів програми. Це невід'ємна складова процесу розробки, спрямована на перевірку окремих компонентів (модулів) програми на відповідність очікуваному функціоналу. Модульне тестування дозволяє виявити помилки на ранніх етапах розробки, забезпечуючи високу якість коду та покращуючи його надійність та є основою класичної піраміди тестування.

Ізоляції компонентів системи під час модульного тестування можна досягти за допомогою заміни залежних компонентів, тобто створення спеціального програмного компонента, який при тестуванні використовується замість реальних компонентів, що викликаються вихідним компонентом або в якійсь мірі залежать від нього. При цьому компонент, що тестується, не залежить від інших компонентів.

Для досягнення ізоляції компонентів під час модульного тестування, використовуються методи, спрямовані на імітацію поведінки справжнього компонента так, щоб інші компоненти системи не помітили підміни під час тестування. Серед них розрізняють [21]:

1. підробні об'єкти (англ. mock): це об'єкти-замінники, які використовуються для заміни реальних об'єктів та компонентів у модульних тестах, які можуть моделювати різні стани та поведінку компонента в залежності від тестових умов. Вони відтворюють поведінку реальних об'єктів і дозволяють ізолювати компоненти від інших частин системи, які можуть бути необхідні для

їхньої роботи. Замінники дозволяють точно контролювати взаємодію між компонентами та перевіряти їхню правильність;

2. порожні компоненти (англ. dummy): це компоненти-замісники, які нічого не роблять, або просто повертають фіктивні або статичні значення. Використовуються там, де потрібно передати об'єкт, але його функціональність не є значущою для тестування;

3. фіктивні об'єкти (англ. fake): це об'єкти, які реалізують спрощену версію функціоналу реальних об'єктів для використання у модульних тестах. Вони дозволяють ізолювати тестований компонент від зовнішніх залежностей, що можуть бути складними для налаштування або управління. Наприклад, фіктивна база даних може зберігати дані у пам'яті, а не у реальній базі даних, що полегшує тестування;

4. порожні методи (англ. stubs): це порожні методи або функції, що використовуються для заміни реального функціоналу під час модульного тестування та повертають заздалегідь визначені значення відповідно до вхідних параметрів. Вони дозволяють викликати тестований код без необхідності виконання реальних операцій, таких як доступ до бази даних або мережеві запити. Використовуються для заміни реальних компонентів, які ще не реалізовані або недоступні для тестування;

5. вбудовування залежностей (DI): Застосування принципу ін’єкції залежностей дозволяє вставляти залежності в тестований компонент через конструктор, метод або властивість. Це дозволяє використовувати mock-об'єкти або stubs-функції для ізоляції тестованого компонента від його залежностей;

6. об'єкти-шпигуни (англ. spy): це об'єкти, що здатні реєструвати те, як вони викликаються у тестованому компоненті, та параметри, з якими вони були викликані. Вони дозволяють перевіряти правильність викликів та взаємодії між компонентами.

Під час використання перерахованих методів може збиратись різноманітна статистика, наприклад, виклики функцій або параметрів компонента, типи параметрів, аргументи, що були передані під час виклику, різні стани об’єкту,

контексту виконання додатка та ін. Все це може аналізуватись для перевірки поведінки програмованої системи та для створення тестового сценарію.

Модульне тестування є хоч і базовим, проте одним з найважливіших етапів під час розробки програмного забезпечення. Використання методів імітування компонентів дозволяє ефективно тестувати програму у різних умовах та взаємодії з іншими компонентами. Ізоляція компонентів допомагає зменшити ризики виникнення помилок та непередбачених поведінок та змоделювати поведінку графічного інтерфейсу за різних станів, за допомогою передачі різних параметрів ізольованим блокам.

1.3.2 Тестування користувацького інтерфейсу

Користувацький інтерфейс визначає спосіб взаємодії користувача з програмним продуктом, тому тестування його функціональності та ефективності важливо для забезпечення високої якості програмного забезпечення. Необхідно підкреслити, що успішне тестування користувацького інтерфейсу не обмежується лише виявленням помилок, а також включає в себе забезпечення того, що інтерфейс відповідає потребам і очікуванням кінцевих користувачів. Такий підхід допомагає підвищити задоволеність користувачів та забезпечити успішну експлуатацію програмного забезпечення. У цьому розділі ми дослідимо підхід до тестування інтерфейсу, в контексті класичної піраміди тестування.

Застосування класичної піраміди тестування для тестування користувацького інтерфейсу передбачає використання різних стратегій для кожного з рівнів піраміди. Деякі з найбільш ефективних стратегій включають:

– автоматизацію тестів на рівні модулів (англ. units);
– тестування інтеграції;
– GUI-тести.

Стратегія автоматизації тестів на рівні модулів передбачає використання автоматизованих тестів для перевірки окремих компонентів інтерфейсу. Вона дозволяє виявляти та виправляти помилки на ранніх етапах розробки, що значно зменшує витрати на подальше тестування.

Стратегія тестування інтеграції передбачає перевірку взаємодії між різними частинами інтерфейсу, щоб забезпечити їх правильну роботу як єдиного цілого. Це допомагає уникнути проблем, пов'язаних зі змінами в одному компоненті, якщо ці зміни можуть вплинути на інші компоненти додатку, що залежать від того, що розглядається.

Стратегія тестування графічного інтерфейсу передбачає розробку та виконання тестів, які перевіряють користувацький інтерфейс в цілому. Вони дозволяють забезпечити правильну роботу програми з точки зору кінцевого користувача та виявити будь-які проблеми, пов'язані з інтерфейсом

Залежно від мови програмування, типу інтерфейсу та фреймворку, використаного для його розробки, тестування зазвичай зводиться до підміни компонентів бізнес-логіки на mock-об’єкти та написання розробником unit-тестів до абстрагованих компонентів користувацького інтерфейсу. Проте, розмежування компонентів системи, абстрагування одне від одного та виокремлення потребує достатньої підготовки розробника та виходить далеко не завжди, тому часом тестування виконується у наскрізному режимі. Цей тип інтеграційного тестування передбачає запуск тестування роботи системи у середовищі та режимі, що наближені до реальної праці, наскільки це можливо.

Більшість систем не потребує тестування всіх можливих сценаріїв та достатнім є перевірити, що виконання критичних сценаріїв використання додатку є безпомилковим. Ця перевірка виконується тестувальниками в режимі ручного тестування, та зазвичай гарантує перевірку більшості бізнес-процесів використання додатку та їх аспектів, а також працездатність системи, відповідно до поставлених вимог до програмного забезпечення, тому проведення додаткового тестування граничних випадків не є доцільним.

1.3.3 Розширена модель тестування

Більшість сучасних продуктів включають багато компонентів, які, в свою чергу можуть складатись з підкомпонентів, тож більше сценарії тестування мають охоплювати більшу кількість шарів, аніж класична піраміда тестування,

для чого була введена розширена піраміда тестування.

Розширена піраміда тестування – це підхід до організації тестування програмного забезпечення, який розширює класичну піраміду тестування, розроблену Майклом Коеном [20]. Основна ідея полягає в тому, щоб рівноважно використовувати різні види тестів на різних рівнях абстракції програми залежно від їхньої вартості та корисності.

Якщо класична піраміда тестування складається з лише трьох рівнів, то розширена піраміда додає додаткові рівні та типи тестів, такі як API-тести, модульні тести, функціональні тести, регресійні тести та інші.

Головна мета розширеної піраміди тестування - це забезпечити більш швидке та ефективне виявлення помилок у програмному забезпеченні за допомогою автоматизованих тестів на різних рівнях абстракції, збільшивши при цьому швидкість розробки та зменшивши вартість тестування.

Основні її рівні включають:

– ручне тестування на нижньому рівні;
– рівень модульних тестів;
– рівень API-тестів;
– рівень інтеграційних тестів;
– рівень UI-тестів.

Ручне тестування на нижньому рівні спрямоване на залучення інженерів з тестування та включає в себе такі підходи ручного тестування, таке як дослідницьке тестування, користувацькі сценарії та інші техніки для перевірки функціональності та взаємодії з користувачем.

На рівні модульних тестів створюються автоматизовані тести, тобто на рівні окремих модулів програми або компонентів. Ці тести перевіряють правильність роботи конкретних функціональних частин програми.

На рівні API-тестів розробляються та виконуються автоматизовані тести, що перевіряють взаємодію між різними компонентами програми через API. Ці тести перевіряють, як компоненти взаємодіють один з одним.

На рівні інтеграційних тестів перевіряють взаємодію між різними компонентами програми у контексті програми як цілісної системи за допомогою автоматизованих тестів.

Користувацький інтерфейс програми, включаючи його візуальний вигляд та коректність взаємодії з користувачем перевіряють за допомогою автоматизовані тестів на рівні UI-тестів.

Розширеної піраміда тестування має чимало важливих переваг над класичною пірамідою, серед яких слід окреслити наступні:

– швидше виявлення помилок, бо завдяки використанню автоматизованих тестів, помилки можуть бути виявлені раніше у вибудованому процесі розробки;

– зменшення ризику непродуктивних помилок, бо автоматизація дозволяє швидко та ефективно виконувати тести, зменшуючи ризик помилок, пов'язаних з ручним тестуванням;

– ефективне використання ресурсів, бо автоматизація дозволяє використовувати тести протягом тривалого часу без необхідності постійного участі людей.

Проте розширена піраміда тестування має і деякі недоліки, серед яких можна назвати наступні:

– складність реалізації та підтримки автоматизованих тестів, бо налагодження автоматизованих тестів може вимагати значних зусиль та часу;

– необхідність високої компетентності, оскільки розробка та підтримка автоматизованих тестів вимагає спеціалізованих знань та навичок.

Розширена піраміда тестування застосовується частіше в порівнянні з класичною пірамідою тестування, особливо під час тестування мобільних додатків, оскільки вона дозволяє швидше виявляти помилки та забезпечує більшу стабільність і надійність програмного забезпечення у довгостроковій перспективі. Автоматизовані тести дозволяють випробовувати програмне забезпечення на різних пристроях та платформах швидко та ефективно.

1.4 Аналіз досліджень

Стрімке зростання за останнє десятиліття кількості наукових робіт, присвячених тематиці автоматичної генерації тестів, описується в дослідженнях [22] та [23], що провели систематичне дослідження літератури (англ. systematic mapping study) та дослідили зростання інтересу наукової спільноти до теми тестування графічного інтерфейсу користувача Android-додатків.

Загалом розрізняють два види тестування: це функціональне та нефункціональне тестування. Функціональне тестування стосується функціональних можливостей програми. В контексті мобільних додатків – це тестування інтерфейсу додатка, функціональних можливостей сервісу та API. Тестування інтерфейсу користувача – це різновид функціонального тестування [24].

Базуючись на дослідженні [25] розглянемо інструменти тестування "білого", "сірого" та "чорного ящика". Тестування "білого ящика" розглядає внутрішню структуру програми, оскільки процес тестування базується на наявності вихідного коду програми. Існує кілька методів тестування білого ящика, як-от AGRippin, EvoDroid, MonkeyImprover, Monkey++, ACRT. На вхід вони приймають вихідний код Android-додатку.

Тестування "чорного ящика" не враховує наявність вихідного коду програми, однак використовує інтерфейси. Існує кілька підходів до тестування за допомогою чорної скриньки, як-от CrawlDroid, Stoat, GATS, MobiGUITAR, Monkey, Dynodroid, ADAPTDROID. На вхід вони отримують APK-файл додатку.

Тестування "сірого ящика" – це комбінація тестування чорного та білого ящика. Таке тестування базується на структурі програми та вимогах. Наприклад, Sapienz отримує на вхід APK-файл програми, розпаковує його, щоб отримати вихідний код. Однак цей підхід також можна застосувати, якщо наявний вихідний код.

Інші підходи, як от TEGDroid [26], намагаються врахувати як контекст, так і GUI події для генерації тест-кейсів. У своєму дослідженні, автори

використовують статичний аналіз байткоду та аналіз файлів XML для визначення подій і контексту.

Dynodroid [27], в свою чергу розглядає Android-додаток, як керовану подіями програму, що взаємодіє зі своїм середовищем за допомогою послідовності подій через інфраструктуру Android. Dynodroid легко відстежує реакцію програми на події (event), використовуючи їх для створення наступних подій. Dynodroid також дозволяє чергувати події від машин, що генерують велику кількість простих вхідних даних, з подіями від користувача.

Дослідники розробили різні інструменти для автоматизації генерації тестів для Android-додатків. В «Додатку А» в таблиці А.1 наведено класифікацію інструментів генерації тестів для Android за такими ознаками:

1. метод;
2. підхід до генерації тестових кейсів;
3. вхідні дані;
4. середовище тестування;
5. артефакти тесту;
6. основний підхід.

Деякі з існуючих підходів, такі як Exerciser Monkey [28], випадковим чином генерують події, що імітують взаємодію користувача, однак мають обмежену ефективність, оскільки не враховують структуру самого додатка, а лише виконують випадкові або передбачувані дії. Підхід використовує Reinforcement Learning [28, 29, 30], щоб навчити систему генерувати тести, які задовольняють визначені Linear-time Temporal Logic (LTL) специфікації. Важливим аспектом є те, що цей підхід дозволяє комбінувати взаємодії як від користувачів, так і від машин, покращуючи при цьому ефективність генерації.

В роботі [29] був запропонований підхід тестового охоплення на основі Q-Learning та був запропонований інструмент DroidbotX для створення тестових випадків графічного інтерфейсу користувача для Android-додатків, щоб максимізувати охоплення інструкцій, методів і активностей (Activities).

Результати продемонстрували, що адаптація Q-Learning з дослідженням верхньої довірчої межі є досить успішною.

1.4.1 Автоматизоване тестування графічного інтерфейсу користувача (GUI) за допомогою Q-Learning

В роботі [31] було представлено AutoBlackTest, який є піонерським рішенням для тестування графічного інтерфейсу на основі Q-Learning, пристосованим для настільного програмного забезпечення на Java. Спочатку AutoBlackTest фіксує абстрактне представлення поточного стану графічного інтерфейсу і будує на його основі поведінкову модель. Ця модель оновлюється відповідно до нового стану інтерфейсу і безпосередньо виконаних дій. Далі ця поведінкова модель керує вибором наступної дії для виконання, і таким чином знову запускає ітеративний процес. Інший інструмент, TESTAR [32], використовує принципи Q-Learning для створення послідовностей тестів графічного інтерфейсу, в першу чергу орієнтованих на веб-додатки. Ефективність алгоритму Q-Learning особливо очевидна у поєднанні з чітко визначеним набором параметрів, що призводить до значного підвищення продуктивності.

GunPowder [33] – це інструмент генерації тестових даних, призначений для генерації тестових даних на основі пошуку з використанням глибокого навчання з підкріпленням. Розроблений спеціально для додатків мовою C, GunPowder проходить три окремі фази: інструментарій, виконання та оцінка придатності. На етапі інструментації він спочатку включає допоміжні коди для полегшення контролю та моніторингу виконання програми. Після цього програма будується і виконується, що завершується застосуванням алгоритму машинного навчання на третьому етапі для генерації тестових даних. Наразі GunPowder підтримує функцію оцінки придатності, спрямовану на покращення покриття гілок. Хоча це не підходить для тестування додатків Android, інші дослідницькі проекти використовують методи RL для таких цілей [34,35,36].

В роботі [35] було описано інструмент автоматизованої генерації тестових кейсів для додатків Android, який використовує Q-Learning і модель Маркова для описання unit-тестування додатків. Цей інструмент розпізнає і вивчає відповідну поведінкову модель, а потім використовує її для генерації тестових кейсів. Процес тестування складається з виконання послідовності подій, які називаються "епізодами", протягом фіксованої кількості ітерацій. Після завершення епізоду інструмент вибирає випадковий стан з тих, що були раніше, щоб розпочати наступну фазу. Однак цей інструмент має кілька обмежень, зокрема, він зосереджений виключно на генеруванні подій інтерфейсу користувача, нехтуючи діями, спричиненими системними подіями.

В роботі [36] було представлено інструмент автоматизованої генерації тестових кейсів на основі Q-Learning для додатків Android, створений на основі фреймворків Appium [22] та UI Automator [24]. На етапі генерації тестових кейсів інструмент обирає події з найвищими значеннями Q з доступного набору подій у кожному стані. Процедура генерації тестових кейсів тісно пов'язана з попередніми дослідженнями [35], поділяючи процес на епізоди, де стани, використані в попередніх епізодах, слугують основою для запуску нових епізодів. Автори визначають стан як множину, що охоплює унікальні доступні дії.

AndroFrame [34] – це інструмент дослідження на основі Q-Learning, розроблений для генерації тестових кейсів. Відрізняючись від випадкових підходів, він здійснює навігацію через графічний інтерфейс на основі попередньо наближеного розподілу ймовірностей, який відповідає визначеній меті тесту. Цей інструмент будує Q-матрицю, що відображає ймовірності досягнення мети тесту, а потім використовує її для визначення подальших дій. Однак AndroFrame демонструє непослідовне покриття дій і підтримує виключно одноцільові функції придатності, де кожне виконання має на меті покращити покриття активностей або пошук збоїв.

1.4.2 Автоматизоване тестування графічного інтерфейсу з навчанням з підкріпленням

AimDroid [37] – це інструмент генерації тестових кейсів на основі моделей, розроблений спеціально для Android-додатків. Він використовує випадковий підхід, керований RL, що складається з двох основних дій. Спочатку він проводить пошук вшир, щоб виявити недосліджені дії, поміщаючи кожну знайдену дію в "клітку". Далі він інтенсивно досліджує ці дії, використовуючи алгоритми нечіткої логіки, керований RL. Процес тестування сегментований на епізоди, кожен з яких генерує обмежену кількість подій, концентруючись на певній активності, пригнічуючи переходи від однієї активності до іншої. Крім того, AimDroid використовує алгоритм RL під назвою SARSA, щоб дізнатися про здатність подій, які можуть відкривати нові дії, "заглядати вперед" і вибирати події, які з більшою ймовірністю можуть викликати нові види діяльності або аварії. Однак AimDroid має певні обмеження. Наприклад, він відключає переходи між активностями, що потенційно може призвести до несправностей, які виникають через життєві цикли робіт. Також AimDroid не має можливості вивчати другу найкращу подію на вибір; він покладається виключно на подію на основі SARSA, вибираючи всі інші події випадковим чином.

Humanoid [38] був розроблений спільно з DroidBot [39], метою якого було вивчення взаємодії користувачів з Android-додатками. Humanoid використовує модель графічного інтерфейсу для розуміння та аналізу поведінки додатку, що тестується, надаючи перевагу компонентам інтерфейсу, з якими взаємодіє людина. Інструмент працює в два основних етапи:

1. на етапі автономного навчання використовується модель глибокої нейронної мережі для розуміння взаємозв'язку між контекстами графічного інтерфейсу та взаємодією з користувачем;

2. етап онлайн-тестування включає в себе модель переходу інтерфейсу користувача для автоматизованого unit-тестування. Тут Humanoid використовує як модель переходу до інтерфейсу, так і модель взаємодії, щоб визначити тип тестових даних, які буде використовувати.

Однак, Humanoid має певні обмеження. Він не дає помітного розширення покриття порівняно з іншими інструментами, і йому не вистачає можливості використовувати текстову інформацію в додатку для генерації тестових кейсів.

1.4.3 Підходи до автоматизованого тестування графічного інтерфейсу

З'явилося багато методологій для оптимізації побудови тестових кейсів та дослідження додатків Android, зокрема:

1. випадкові підходи;
2. методи, засновані на моделях;
3. методології систематичного тестування.

Випадкове тестування є однією з найпоширеніших методик виявлення несправностей системного рівня в додатку. Цей метод спрямований на генерацію потоку непередбачуваних подій, хоча часто не дозволяє всебічно дослідити всі функціональні можливості програми [40]. Хоча в основному його використовують для стрес-тестування, випадкове тестування ефективно генерує події, що сприяло його широкому поширенню. Зокрема, Android Monkey [41] являє собою інструмент тестування графічного інтерфейсу методом чорного ящика, що входить до складу Android Software Development Kit (SDK). Відомий своєю простотою та широкою сумісністю з різними платформами Android, Android Monkey став популярним серед багатьох додатків на ринку [42,43].

Тим не менш, Android Monkey забирає багато часу при створенні довгих послідовностей подій. Ці послідовності часто містять надлишкові події, які повторюють навігацію між активностями програми, а також непродуктивні події, які взаємодіють з неінтерактивними областями на екрані [37, 44, 45]. Вирішуючи цю проблему, в роботі [46] представили доповнення до Android Monkey, спрямоване на автоматичне виявлення збоїв інтерфейсу користувача. Інший підхід, Dynodroid [47], використовує два евристичні алгоритми для вибору релевантних подій на основі поточного стану програми, що повторюються через цикл спостереження-вибір-виконання. SmartMonkey [48] використовує методологію випадкового тестування, запропоновану в [40], щоб мінімізувати

кількість тестових кейсів і час, необхідний для виявлення початкової несправності. Він будує модель переходів додатку через випадкову взаємодію і генерує тестові кейси за допомогою техніки випадкового обходу, що складаються з послідовностей як користувацьких, так і системних подій.

Тестування на основі моделей ґрунтується на графовій моделі, яка відображає взаємодію користувача з графічним інтерфейсом програми. Ця модель може бути створена вручну або автоматично, на основі специфікацій програми, що тестується, таких як код або конфігураційні файли XML, або шляхом безпосередньої взаємодії з програмами. Дослідження на основі моделей може бути спрямоване на конкретні недосліджені області з використанням систематичних стратегій, таких як дослідження в глибину, дослідження в ширину або гібридні підходи [49], або з використанням стохастичних моделей [50]. Однак методи, що базуються на моделях, стикаються з проблемами, пов'язаними з неточністю моделювання. Зокрема, динамічна поведінка графічних інтерфейсів може призвести до неточностей у моделі або до некоректної інтерпретації стану через недетерміновані зміни в графічних інтерфейсах.

Отже, підхід, заснований на моделях, може не помічати змін, вважаючи певні події несуттєвими, і, як наслідок, відхилятися від дослідницького шляху. Зокрема, модель графічного інтерфейсу, що охоплює лише обмежений спектр потенційних поведінкових просторів, може знизити ефективність тесту. В роботі [51] було описано дослідження багаторівневого представлення станів, що виявило факт суттєвого впливу різних рівнів абстракції на ефективність інструменту моделювання. A3E [49] використовує дві стратегії дослідження: глибинне дослідження, яке систематично аналізує додатки в реальних умовах без доступу до вихідного коду, і цільове дослідження, яке визначає пріоритети діяльності, що походять від початкової діяльності на статичному графі переходів діяльності. Однак A3E представляє кожну активність як окремий стан, не враховуючи можливість існування активностей у різних станах, що призводить

до того, що певна поведінка програми може бути проігнорована через неповне дослідження всіх станів дій.

Orbit [52] виконує статичний аналіз вихідного коду програми, щоб генерувати відповідні події, які підтримуються програмою. Однак його метод дослідження використовує простий підхід "спочатку вглиб", який передбачає перезапуск програми з початкового стану, щоб відкотитися до попередніх станів. PUMA [53] інтегрує загальний UI Automator [15], що використовує схожий з Android Monkey випадковий підхід, хоча і з відмінним дизайном, використовуючи динамічний аналіз, щоб викликати зміни оточення під час виконання програми. Stoat [50] проводить тестування стохастичної моделі у два етапи. Спочатку він будує імовірнісну модель, динамічно досліджуючи та аналізуючи взаємодію з графічним інтерфейсом додатку. Згодом виконується уточнення моделі станів, використовуючи вибірку Гіббса, і генеруються тести на основі оптимізованої моделі, щоб максимізувати покриття коду та активності. З іншого боку, Ape [54] застосовує підхід динамічного моделювання для покращення початкової моделі графічного інтерфейсу, використовуючи інформацію про час виконання.

Систематичне тестування використовує передові методи, такі як символьне виконання та еволюційні алгоритми для отримання цільових вхідних даних. Перевага цього методу полягає в тому, що він дозволяє використовувати вихідний код для створення тестів, виявляючи раніше недосліджену поведінку програми. Thor [55] працює, піддаючи існуючі тестові кейси негативним умовам. Однак Thor не генерує нові тестові кейси; натомість він вставляє послідовності подій в існуючі, не змінюючи результати початкового тестового кейсу. З іншого боку, CrashScope [56] автоматизує виявлення збоїв у тестованих додатках за допомогою гібридного підходу, що поєднує методи систематичного дослідження та статичного аналізу.

AppDoctor [57] використовує стратегію наближеного виконання для прискорення процесів тестування та автоматичної класифікації більшості звітів на помилки та хибні спрацьовування. Тим не менш, пропонуючи можливість

відтворення помилок і надання розробникам стекових трас, цей підхід вимагає повторного трасирування збоїв, щоб відфільтрувати хибнопозитивні спрацьовування, і не надає детальних і вичерпних звітів. SIG-Droid [58], з іншого боку, є інструментом автоматизованої генерації системного вводу для додатків Android, що поєднує методи аналізу програм з символьним виконанням для досягнення широкого покриття коду. Sapienz [59] використовує повністю автоматизовану багатоцільову методологію тестування на основі пошуку, використовуючи генетичні алгоритми для оптимізації тестових послідовностей, щоб максимізувати покриття коду і виявлення помилок при мінімізації довжини послідовності. Sapienz досліджує компоненти додатків за допомогою специфічних графічних інтерфейсів та складних послідовностей вхідних подій, що відповідають заздалегідь визначеному шаблону, що інкапсулює досвід тестувальників. Таким чином, він розширює покриття коду за рахунок конкатенації атомарних подій [60].

Якщо описані вище методи розглядають та пропонують засоби для генерування тестів для традиційних підходів розробки, то за останні декілька років індустрія Android-розробки, а саме, розробки інтерфейсу, зміщується з традиційної імперативної до декларативної (або функціональної) парадигми, тобто описані вище існуючі підходи втрачають свою актуальність.

Всі описані інструменти прямо чи опосередковано намагаються вирішити проблему генерації коду тестів або сценаріїв unit-тестування, проте вони не враховують сучасних викликів Android-розробки, мова про які піде в наступному розділі.

1.5 Аналіз існуючих рішень

Хоча в попередньому розділі було наведено досить багато інструментів та методів, більшість з них не сформовані, як плагіни або розширення до існуючих інтегрованих середовищ розробки під Android, оскільки спрямовані на окреме встановлення, поставляються, як самостійний продукт, або взагалі недоступні для загального використання. Тому розглянемо найпопулярніші інструменти,

доступні на ринку плагінів, для автоматичної генерації коду функціональних unit-тестів графічного інтерфейсу користувача, що використовують підхід тестування білого ящика. Серед таких плагінів варто виділити Symflower, Parasoft Jtest та EvoSuite.

1.5.1 Symflower

Symflower – це генератор модульних тестів Java, що автоматично створює шаблони та набори тестів JUnit з широким покриттям для додатків Java, Spring та Spring Boot, а також надає діагностику коду в режимі реального часу [61].

Інструмент допомагає прискорити розробку, підвищити якість коду та спростити процес тестування, завдяки інтеграції в популярні IDE. Не порушуючи робочий процес, функції генерації тестів Symflower дозволяють розробникам переключити свій час з нудного тестування на більш практичні завдання. Symflower допомагає організаціям скоротити час виходу на ринок, зменшити кількість вразливостей та збільшити покриття тестів – без ручної роботи.

До переваг Symflower можна віднести:

– автоматична генерація коду тестів;

– підтримка JUnit 4 та JUnit 5 [10] для Java;

– можливість створення тестових наборів, що протестують різні шляхи виконання;

– діагностика: виключення можна виявляти та відтворювати під час виконання;

– підтримка популярних інструментів для збірки: Gradle та Maven;

– підтримка створення mock-об’єктів за допомогою Mockito [11].

При цьому Symflower має наступні недоліки:

– не підтримується мова програмування Kotlin (лише мова програмування Java);

– не підтримується декларативна парадигма програмування (лише імперативна парадигма);

– відсутня підтримка генерування тестів для Jetpack Compose;

– є платним.

1.5.2 Parasoft Jtest

Parasoft Jtest – це потужний генератор тестів, розроблений під мову програмування Java, що надає повний набір засобів статичного аналізу та методів тестування, які можна використовувати для перевірки відповідності стандартам безпеки (OWASP, CWE, CERT, PCI DSS тощо) та власним стандартам кодування за допомогою вбудованих або визначених користувачем правил, виявлення проблем під час виконання без запуску коду, виявлення дублювання коду, а також для розуміння складності та структури коду [62].

Jtest використовує найсучасніший механізм синтаксичного аналізу коду Java для статичного аналізу та розуміння коду, що тестується, та знаходження дефектів коду, на які вказують порушення правил. Jtest надає розширені можливості для перетворення статичного аналізу на підтримуваний елемент процесу розробки, такі як придушення небажаних результатів, визначення пріоритетів і автоматичне призначення результатів розробникам для швидшого виправлення помилок та багато іншого.

До безумовних переваг Jtest можна віднести:

– вбудовані тестові конфігурації: попередньо визначені набори правил, прив'язані до конкретних стандартів кодування або рекомендацій, дозволяють користувачам швидко і зручно виконувати статичний аналіз, спрощуючи дотримання галузевих вимог і стандартів безпеки;

– категорії правил: кожне правило належить до певної категорії, наприклад, «Оптимізація», «Безпека», «Винятки» або «API», щоб допомогти користувачам швидко зрозуміти, як правила можуть відповідати їхнім пріоритетам тестування;

– рівні серйозності (англ. severity): кожному правилу присвоюється рівень серйозності, щоб допомогти користувачам краще зрозуміти потенційний вплив порушення правила;

– використовує понад 40 загальноприйнятих у галузі метрик коду для перевірки якості коду;

– підтримка спеціалізованих засобів пошуку помилок: наприклад, виняток нульового вказівника, витік ресурсів, тупикові ситуації, ділення на нуль, вихід індексу масиву за межі та багато іншого;

– інтеграція зі всіма сучасними IDE та гнучкість використання: статичний аналіз коду можна виконувати в IDE (Eclipse, IntelliJ та VS Code), з командного рядка або за допомогою плагінів системи збірки (Jenkins, Maven, Gradle) для автоматизації та сценаріїв безперервної інтеграції. До результатів аналізу можна отримати негайний доступ (в IDE або у вигляді HTML/XML/PDF звітів), інтегрувати з системами CI, такими як GitHub, GitLab і Azure DevOps, або отримати доступ до Parasoft DTP для пост-обробки, аналізу від збірки до збірки, звітів про відповідність вимогам і розширеної аналітики з пріоритетами машинного навчання;

До недоліків даного плагіну, аналогічно до Symflower, можна віднести:

– не підтримується мова програмування Kotlin (лише мова програмування Java);

– не підтримується декларативна парадигма програмування (лише імперативна парадигма);

– відсутня підтримка генерування тестів для Jetpack Compose;

– є платним.

1.5.3 EvoSuite

EvoSuite – це інструмент, який автоматично генерує тестові кейси з твердженнями для класів, написаних на Java. Для цього EvoSuite застосовує новий гібридний підхід, який генерує та оптимізує цілі набори тестів для задоволення критерію покриття. Для створених тестових наборів EvoSuite пропонує вірогіднісні оракули, додаючи невеликі та ефективні набори тверджень, які стисло описують поточну поведінку; ці твердження дозволяють розробнику виявити відхилення від очікуваної поведінки та зафіксувати поточну поведінку, щоб захиститися від майбутніх дефектів, які порушують цю поведінку [63].

EvoSuite автоматично генерує набори тестів JUnit для класів Java, орієнтуючись на критерії покриття коду, такі як покриття гілок. Він використовує еволюційний підхід на основі генетичного алгоритму для створення наборів тестів. Для покращення читабельності згенеровані модульні тести мінімізуються, а до тестів додаються регресійні твердження, які відображають поточну поведінку класів, що тестуються.

До переваг EvoSuite можна віднести:

– генерація тестів JUnit 4 для вибраних класів;

– оптимізація різних критеріїв покриття, таких як тестування ліній, гілок, виходів та мутацій;

– мінімізація тестів: залишаються лише ті, що сприяють досягненню покриття;

– тести виконуються в "пісочниці", щоб запобігти потенційно небезпечним операціям;

– підтримується віртуальна файлова система;

– підтримується віртуальна мережа;

– має плагін Maven, який можна використовувати для генерації нових тестових кейсів як частину збірки, що дозволяє запускати з серверів безперервної інтеграції з мінімальними накладними витратами на конфігурацію;

– має плагін EvoSuite для IntelliJ.

Серед недоліків EvoSuite слід відзначити:

– не підтримується мова програмування Kotlin (лише мова програмування Java);

– не підтримується декларативна парадигма програмування (лише імперативна парадигма).

1.6 Постановка проблеми

Проведений огляд методів та засобів до автоматичної генерації коду тестів графічного інтерфейсу користувача показав, що наявні підходи до генерування тестів для платформи Android працюють лише з мовою програмування Java та

підтримують лише імперативну парадигму, що не вирішує проблему генерування коду тестів для програм, розроблених мовою програмування Kotlin та з використанням сучасної декларативної парадигми.

Тому для розв’язання вказаних проблем були сформовані наступні задачі розробки для даної магістерської дисертації:

1. розширити рамки використання інтегрованих середовищ розробки під Android шляхом підтримки мови програмування Kotlin для автоматизації генерації коду unit-тестів графічного інтерфейсу, розробленого з використанням фреймворку Jetpack Compose;

2. розробити плагін для автоматизованого генерування коду unit-тестів графічного інтерфейсу для інтегрованого середовища розробки Android Studio;

3. провести дослідження ефективності запропонованого інструменту автоматизованого генерування тестів в інтегрованому середовищі розробки Android Studio.

2 РОЗРОБКА МЕТОДУ ГЕНЕРУВАННЯ ТЕСТІВ ДЛЯ COMPOSABLE-ФУНКЦІЙ

Розробка будь-якого методу взаємодії з програмним кодом починається з розгляду аспектів представлення коду у вигляді дерева та опису семантичної моделі коду для подальшого аналізу та використання. Абстрактне синтаксичне дерево (AST) та семантична модель коду є ключовими концепціями у сфері програмування та компіляції програмного забезпечення. Розглянемо їхні визначення та ролі в аналізі та оптимізації коду.

2.1. Абстрактне синтаксичне дерево та семантична модель коду

Абстрактне синтаксичне дерево (AST, рис. 2.1) є структурою даних, що представляє синтаксичну структуру програми, але без зайвих деталей, що присутні у сирому вихідному коді.

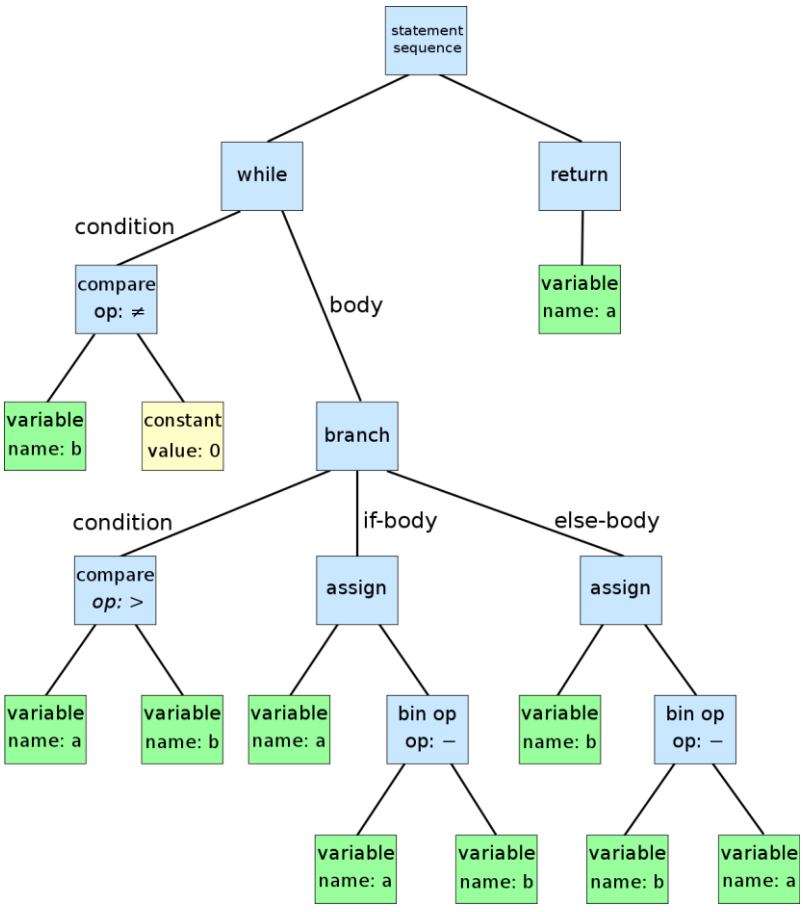

Рисунок. 2.1 – Приклад абстрактного синтаксичного дерева

AST є результатом синтаксичного аналізу (парсингу) вихідного коду і відображає його у вигляді дерева, де вузлами є оператори та операнди, а ребра представляють відношення між ними. Кожен вузол дерева відповідає конструкції мови програмування, такі як оператори, змінні, виклики функцій тощо.

AST зазвичай використовується компіляторами під час перетворення вихідного коду програми у виконуваний або іншу форму, таку як машинний код або байт-код. Також використання AST є поширеним підходом під час генерування програмного коду. Компілятори ж використовують AST для розуміння семантики програми та вирішення задач, таких як оптимізація коду, виявлення помилок та генерація оптимізованого вихідного коду.

Семантична модель коду визначає семантику програми та відображає її структуру та функціональність. Вона йде далі від простого синтаксису та враховує значення та зв'язки між елементами програми. Семантична модель може бути представлена у вигляді додаткових даних, які супроводжують AST або як окрема структура даних.

Під час аналізу коду семантична модель використовується для розуміння, як програма працює, та вирішення складних задач, таких як виявлення залежностей між змінними, перевірка типів даних та виявлення потенційних помилок у логіці програми. Крім того, вона може використовуватися для автоматичного генерування документації, рефакторингу коду та виявлення можливостей для оптимізації.

Семантична модель коду може мати різний вигляд в залежності від мови програмування та інструментів, що використовуються. У найпростіших випадках вона може представляти собою набір таблиць символів, які зберігають інформацію про ідентифікатори, типи даних та інші атрибути програми. У складніших випадках це може бути багаторівнева структура, що включає в себе аналіз потоку управління, виклики функцій, структури даних, токени та інші аспекти програми.

Застосування семантичної моделі коду включає в себе аналіз та перетворення програмного коду для поліпшення його читабельності,

продуктивності та надійності. Це може бути використано для підтримки інструментів розробки, таких як редактори коду з функцією автодоповнення, системи керування версіями, рефакторингу коду та автоматичного виявлення помилок.

У підсумку, абстрактне синтаксичне дерево та семантична модель коду є важливими інструментами у сфері програмування, які допомагають аналізувати, розуміти та оптимізувати програми. Вони використовуються компіляторами та іншими інструментами розробки для автоматизації процесів розробки програмного забезпечення та підвищення продуктивності програмістів.

В контексті задачі, що розглядається, будуть використовуватись нативні засоби мови програмування Kotlin для перетворення коду Kotlin-файлів на AST-дерево, такі, як Kotlin Compiler та Kotlin Reflect, і далі отримане дерево буде аналізуватись та генеруватимуться відповідні тестові сценарії.

2.2. Особливості Composable-функцій

Розглянемо підхід декларативної парадигми в ретроспективі, базуючись на [64]. Історично ієрархія графічного інтерфейсу в Android представляється у вигляді дерева віджетів. Оскільки стан програми змінюється через взаємодію з користувачем, ієрархію інтерфейсу потрібно оновлювати, щоб відображати поточні дані. Найпоширеніший спосіб оновлення інтерфейсу – пройтись по дереву за допомогою функцій на кшталт findViewById() і змінити вузли за допомогою виклику методів на кшталт button.setText(String), container.addChild(View) або img.setImageBitmap(Bitmap). Ці методи змінюють внутрішній стан віджета.

Маніпулювання віджетами вручну збільшує ймовірність помилок. Якщо фрагмент даних відображається в декількох місцях, легко забути оновити один з view-віджетів, що його відображає. Також легко створити несанкціоновані стани, коли два оновлення конфліктують у непередбачуваний спосіб. Наприклад, оновлення може спробувати встановити значення вузла, який щойно було видалено з інтерфейсу користувача. Загалом, складність супроводу програмного

забезпечення зростає зі збільшенням кількості представлень (view), які потребують оновлення.

Декларативна модель інтерфейсу значно спрощує розробку, пов'язану з побудовою та оновленням користувацьких інтерфейсів. Техніка працює за рахунок концептуальної регенерації всього екрану з нуля, після чого вносяться лише необхідні зміни. Такий підхід дозволяє уникнути складності ручного оновлення ієрархії представлень зі станом. Compose – це декларативний фреймворк інтерфейсу користувача.

Однією з проблем регенерації всього екрану є те, що це потенційно дорого з точки зору часу, обчислювальних потужностей та використання батареї. Щоб зменшити ці витрати, Compose розумно вибирає, які частини інтерфейсу потрібно перемалювати у певний момент часу.

За допомогою Compose можна створити користувацький інтерфейс, визначивши набір функцій, які можна компонувати, які отримують дані і виводять елементи інтерфейсу. Простим прикладом є віджет Greeting, який отримує рядок і виводить текстовий віджет, що відображає привітання (рис. 2.2).

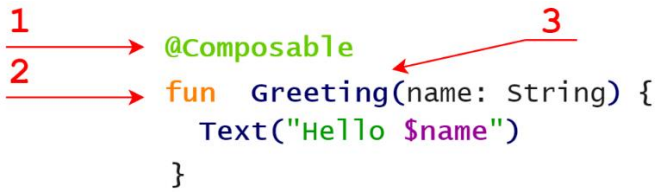

Рисунок 2.2 – Схема composable-функції: 1 – анотація;
2 – ключове слово мови Kotlin для визначення функцій;
3 – назва функції.

Розглянемо цю функцію детальніше:

1. функція має анотацію @Composable (рис. 2.2, 1): усі composable-функції мають бути анотованими; за допомогою цієї анотації компілятор Compose отримує повідомлення про те, що ця функція перетворює передані їй аргументи в елементи користувацького інтерфейсу;

2. функція отримує дані: складані функції можуть приймати параметри, які дозволяють логіці програми описувати інтерфейс користувача. У цьому випадку наш віджет приймає рядок String, щоб він міг привітати користувача за іменем;

3. функція відображає текст в інтерфейсі: Вона робить це шляхом виклику складеної функції Text, яка фактично створює текстовий елемент інтерфейсу. Складані функції створюють ієрархію інтерфейсу, викликаючи інші складані функції;

4. функція нічого не повертає: складеним функціям, що емітують інтерфейс, не потрібно нічого повертати, оскільки вони описують бажаний стан екрана, а не створюють віджети інтерфейсу.

5. функція є швидкою, ідемпотентною і не має побічних ефектів, тобто:

5.1 функція поводиться однаково при багаторазовому виклику з одним і тим же аргументом і не використовує інші значення, такі як глобальні змінні або виклики random().

5.2 функція описує інтерфейс користувача без будь-яких побічних ефектів, таких як зміна властивостей або глобальних змінних.

У багатьох імперативних об'єктно-орієнтованих інструментаріях інтерфейсу інтерфейс ініціалізується створенням дерева віджетів. Часто це робиться шляхом роздування XML-файлу макета. Кожен віджет зберігає свій власний внутрішній стан і надає методи отримання та встановлення, які дозволяють логіці програми взаємодіяти з віджетом.

У декларативному підході Compose віджети не зберігають стан (для цього є окремі функції remember) і не мають функцій setter або getter. Фактично, composable-функції не є об'єктами. Інтерфейс оновлюється шляхом викликів тієї самої функції, що компонується, з різними аргументами (рис. 2.3). Це дозволяє легко передавати зберігання та управління станом таким архітектурним патернам, як ViewModel. Потім, composable-функції відповідають за перетворення поточного стану програми в інтерфейс кожного разу, коли оновлюються видимі дані.

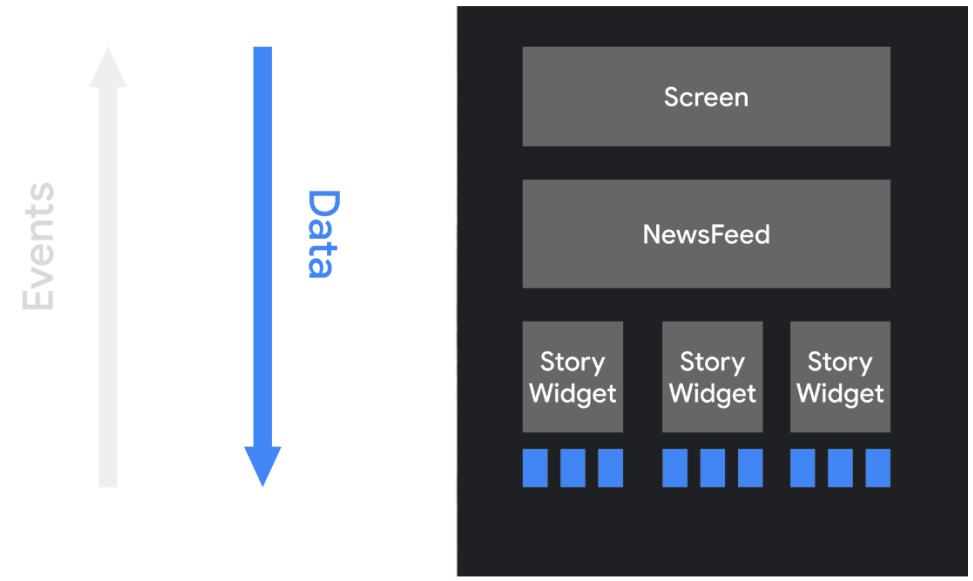

Рисунок. 2.3 – Демонстрація декларативної парадигми: передача даних зверху вниз [64]

Коли користувач взаємодіє з інтерфейсом, інтерфейс генерує події, такі як onClick (рис. 2.4).

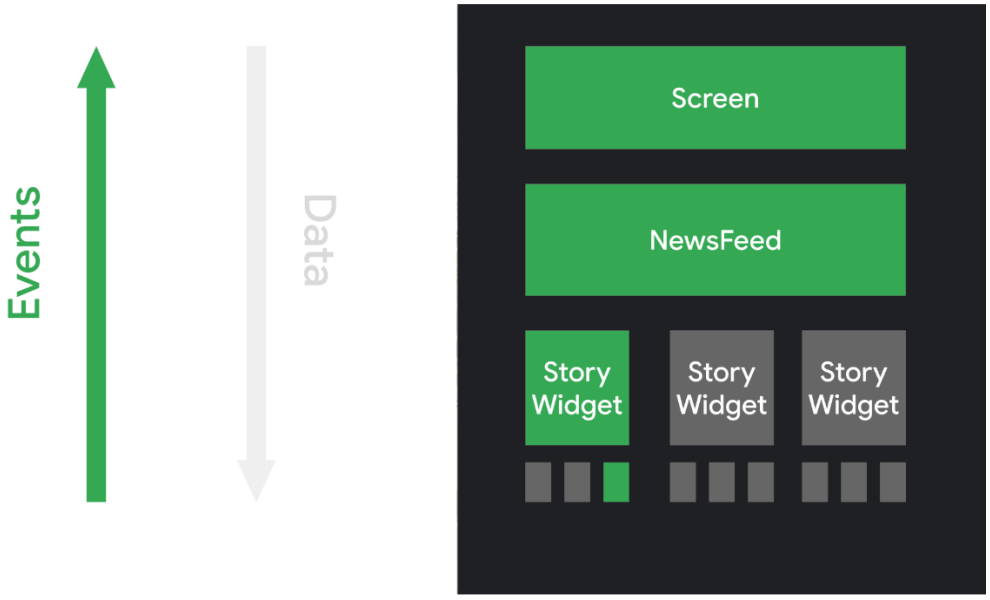

Рисунок 2.4 – Передача події знизу вверх [64]

Ці події повинні емітувати ту логіку програми, яка може змінити стан програми. Коли стан змінюється, composable-функції викликаються знову з

новими даними. Це призводить до перемальовування елементів інтерфейсу – цей процес називається рекомпозицією.

Розглянемо тепер більш складну composable-функцію (рис. 2.5).

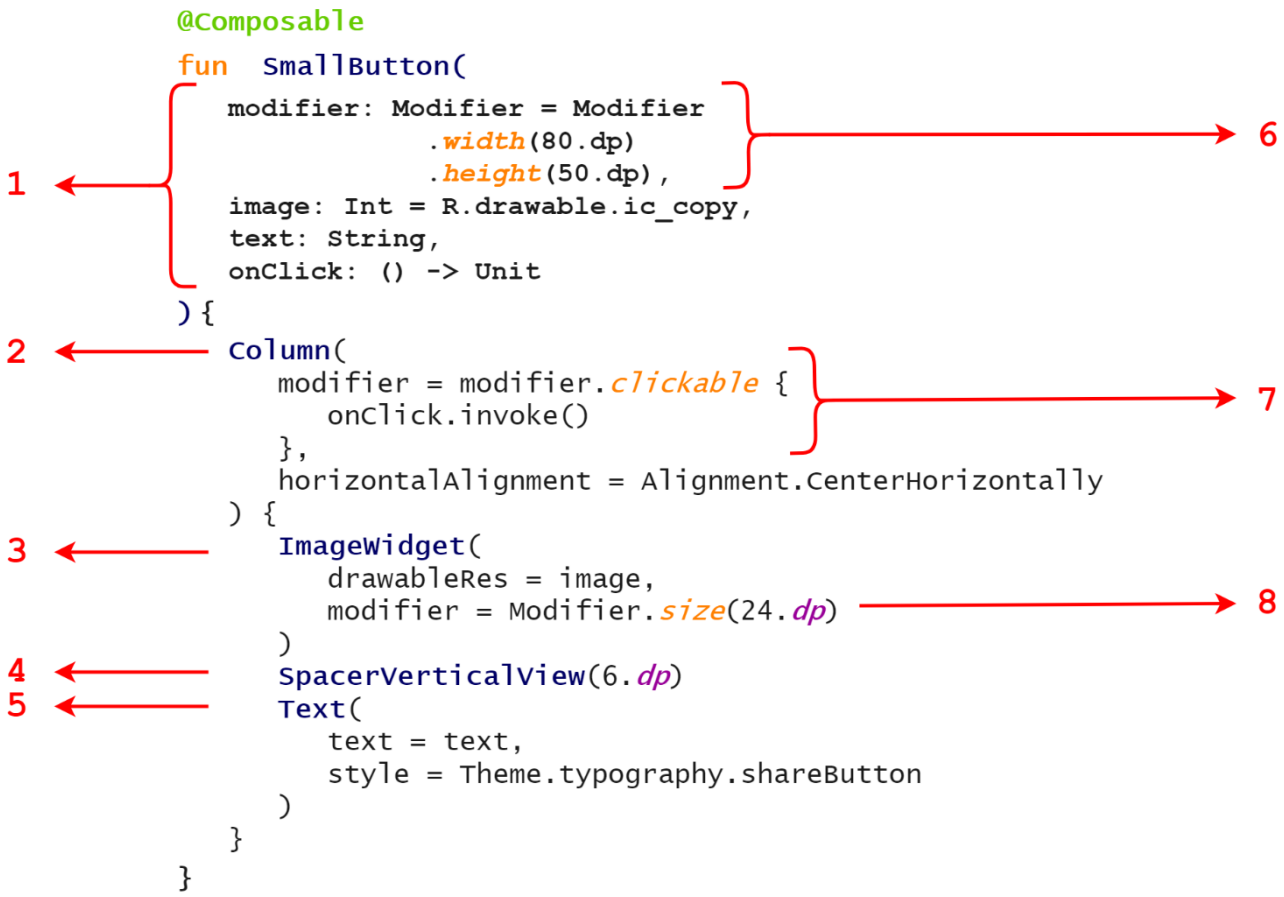

Рисунок 2.5 – Розширена схема composable-функції

Функція SmallButton представляє собою звичайну функцію мови програмування Kotlin та складається з наступних складових:

– анотація @Composable (рис. 2.2, 1);

– ключове слово fun, що дозволяє вказати мові Kotlin, що визначається функція (рис. 2.2, 2);

– назва функції (рис. 2.2, 3);

– список параметрів (рис. 2.5, 1);

– тіло функції.

Тіло функції SmallButton складається з викликів інших composable-функцій, а саме, функції колонки Column 2, що приймає, як дочірні, виклики

функцій віджета картинки ImageWidget 3, пробілу SpacerVerticalView 4 та віджета для відображення тексту Text 5.

Варто звернути увагу, на блоки коду, позначені цифрами 6, 7 та 8 (рис. 2.5). Ці блоки коду представляють собою виклик модифікаторів. Модифікатори дозволяють, як виходить з їх назви, змінювати, прикрашати або доповнювати composable-функції [65]. Модифікатори дають змогу робити такі речі:

– змінювати розмір, компонування, поведінку та зовнішній вигляд функції, що рекомпозується;

– додавати інформацію (наприклад, мітки доступності);

– обробляти користувацьке введення;

– додавати високорівневі взаємодії (наприклад, зробити елемент клікабельним, прокручуваним, перетягуваним або масштабованим).

Порядок викликів функцій у модифікатора є важливим. Оскільки кожна функція вносить зміни до модифікатора, повернутого попередньою функцією, послідовність впливає на кінцевий результат. Кілька модифікаторів можна об'єднати в ланцюжок, щоб прикрасити або доповнити композицію. Цей ланцюжок створюється за допомогою інтерфейсу Modifier, який представляє собою впорядкований, незмінний список окремих Modifier.

Модифікатор Modifier представляє собою патерн Builder, який дозволяє задати віджету ті параметри відображення, які вимагаються. Всі ці аспекти варто врахувати при розробці алгоритму тестування composable-функцій.

2.3. Тестування Composable-функцій

Тестування інтерфейсів або екранів використовується для перевірки правильної поведінки Compose-коду, покращуючи якість додатку шляхом виявлення помилок на ранніх стадіях розробки. Compose надає набір тестових API для пошуку елементів, перевірки їхніх атрибутів та виконання дій користувача [66].

Тестування інтерфейсу, створеного за допомогою Compose, відрізняється від тестування інтерфейсу на основі View. Інструментарій View-based UI чітко

визначає, що таке View. Вигляд займає прямокутний простір і має властивості, такі як ідентифікатори, позиція, поля, відступи тощо. У Compose лише деякі елементи інтерфейсу можна компонувати в ієрархію інтерфейсу, тому потрібен інший підхід до узгодження елементів інтерфейсу.

Тести інтерфейсу користувача у Compose використовують семантику для взаємодії з ієрархією інтерфейсу. Семантика, як випливає з назви, надає значення фрагменту інтерфейсу. У цьому контексті "фрагмент інтерфейсу" може означати будь-що, від одного composable-елемента до повноекранного. Дерево семантики створюється разом з ієрархією інтерфейсу і описує її (рис. 2.6).

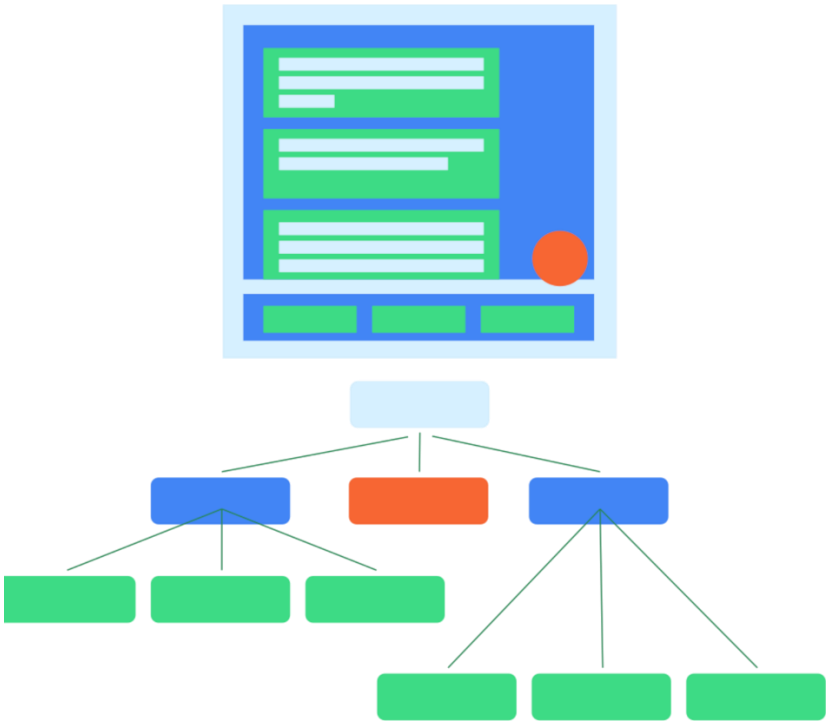

Рисунок 2.6 – Типова ієрархія інтерфейсу користувача
та його дерево семантики [66]

Для початку тестування, необхідно вказати правило тестування, яке міститься в класі ComposeTestRule і має реалізацію для Android під назвою AndroidComposeTestRule. За допомогою цього правила можна встановити композицію вмісту або отримати доступ до активності. Правила створюються за допомогою заводських функцій createComposeRule або

createAndroidComposeRule, якщо потрібен доступ до активності. Типовий UI-тест для Compose виглядає, як показано на лістингу нижче [66]:

```
class MyComposeTest {
    @get:Rule val composeTestRule = createComposeRule()

    @Test
    fun myTest() {
        composeTestRule.setContent {
            MyAppTheme {
                MainScreen(uiState = fakeUiState, /*...*/)
            }
        }

        composeTestRule.onNodeWithText("Continue").performClick()

        composeTestRule.onNodeWithText("Welcome").assertIsDisplayed()
    }
}
```

ComposeTestRule дозволяє запустити активність, яка відображає будь-який елемент композиції: повний додаток, окремий екран або невеликий елемент. Це хороша практика для перевірки правильності інкапсуляції composable-компонентів і їхньої незалежної роботи, що дозволяє спростити і зробити більш цілеспрямованим тестування інтерфейсу користувача.

Існує три основні способи взаємодії з елементами:

– пошук (finder): дозволяє вам вибрати один або декілька елементів (або вузлів у дереві семантики), щоб зробити твердження або виконати дії над ними.

– твердження (assertions): використовуються для перевірки того, що елементи існують або мають певні атрибути.

– дії (actions): вводять імітовані користувацькі події над елементами, такі як кліки або інші жести.

Для пошуку вузла можна використовувати onNode та onAllNodes для вибору одного або декількох вузлів відповідно, але також можна скористатися зручними пошукачами для найпоширенішого пошуку, такими як onNodeWithText, onNodeWithContentDescription тощо. Наприклад, для вибору одного вузла можна виконати наступну команду:

```
composeTestRule.onNode(hasText("Button"))
```

або для вибору багатьох вузлів:

```
composeTestRule.onAllNodes(hasText("Button"))
```

Перевіряти твердження можна, викликаючи assert() на SemanticsNodeInteraction, що повертається шукачем з одним або декількома matchers, наприклад:

```
composeTestRule
    .onNode(matcher)
    .assert(hasText("Button"))
```

Щоб застосувати дію до вузла, необхідно викликати функцію perform<тип дії>(), наприклад:

```
composeTestRule.onNode(...).performClick()
```

Загалом, фреймворк Jetpack Compose надає достатній API для написання коду модульних тестів графічного інтерфейсу, розробленого з використанням composable-функцій. На відміну від звичного підходу, коли графічний інтерфейс представляється у вигляді об’єктів або віджетів, декларативний інтерфейс користувача – є функціями, а це накладає певні обмеження та додаткові вимоги на генерування коду модульних тестів і в наступному підрозділі розглянемо алгоритм для цього.

2.4. Алгоритм генерування тестів Composable-функцій

Враховуючи описані особливості composable-функцій та API для тестування, що надається Compose, можемо визначити ключові моменти, необхідні для генерування тесту. Серед них варто виділити наступні:

– перевірка на наявність в файлі коду Kotlin composable-функцій: оскільки це звичайна функція, composable-функції можна включати до декларування класів проте викликати їх можна буде лише з Compose-контексту;

– визначення модифікаторів: необхідно отримати перелік модифікаторів, що передаються в composable-функцію та перевірити їх на

наявність клікабельних елементів, або елементів, що підтримують введення тексту;

– визначення умов: коректно згенерований тест, має включати в себе комбінації різних умовних операторів, якщо такі є в коді composable-функції та можуть бути змінені через вхідні параметри;

– визначення контексту тіла composable-функції;

– ізоляція станів: якщо об’єкти стану передаються в тіло composable-функції як параметр, то мають бути створені mock-об’єкти для імітації стану та забезпечення ізоляції модулів під час тестування;

– побудова рекурсивним аналізом низхідного графа викликів вкладених функцій та множини комбінацій вхідних даних;

– генерування всіх можливих сценаріїв для даного інтерфейсу на основі всіх можливих комбінацій імітованих значень для заданої composable-функції.

Мовою псевдокоду загальний алгоритм генерування тесту буде наступним:

```
start, Functions[]
if файл містить composable-функцію then
    Functions = EXTRACT(composable-функції)
    for each f ∈ Functions do
        Modifiers[], Conditions[], Mocks[]
        Modifiers = EXTRACT(модифікатори composable-функції)
        if (composable-функція має умови виклику) then
            Conditions = EXTRACT(умови)
        if (composable-функція має об’єкти стану) then
            Mocks = CREATE(mock об’єкти станів)
        GENERATE_TEST(Modifiers, Conditions, Mocks)
end
```

Описаний алгоритм представлено у вигляді узагальненої блок-схеми на рис. 2.7. Кожен крок алгоритму має багато інкапсульованих деталей, більшість яких потребує більш детального висвітлення.

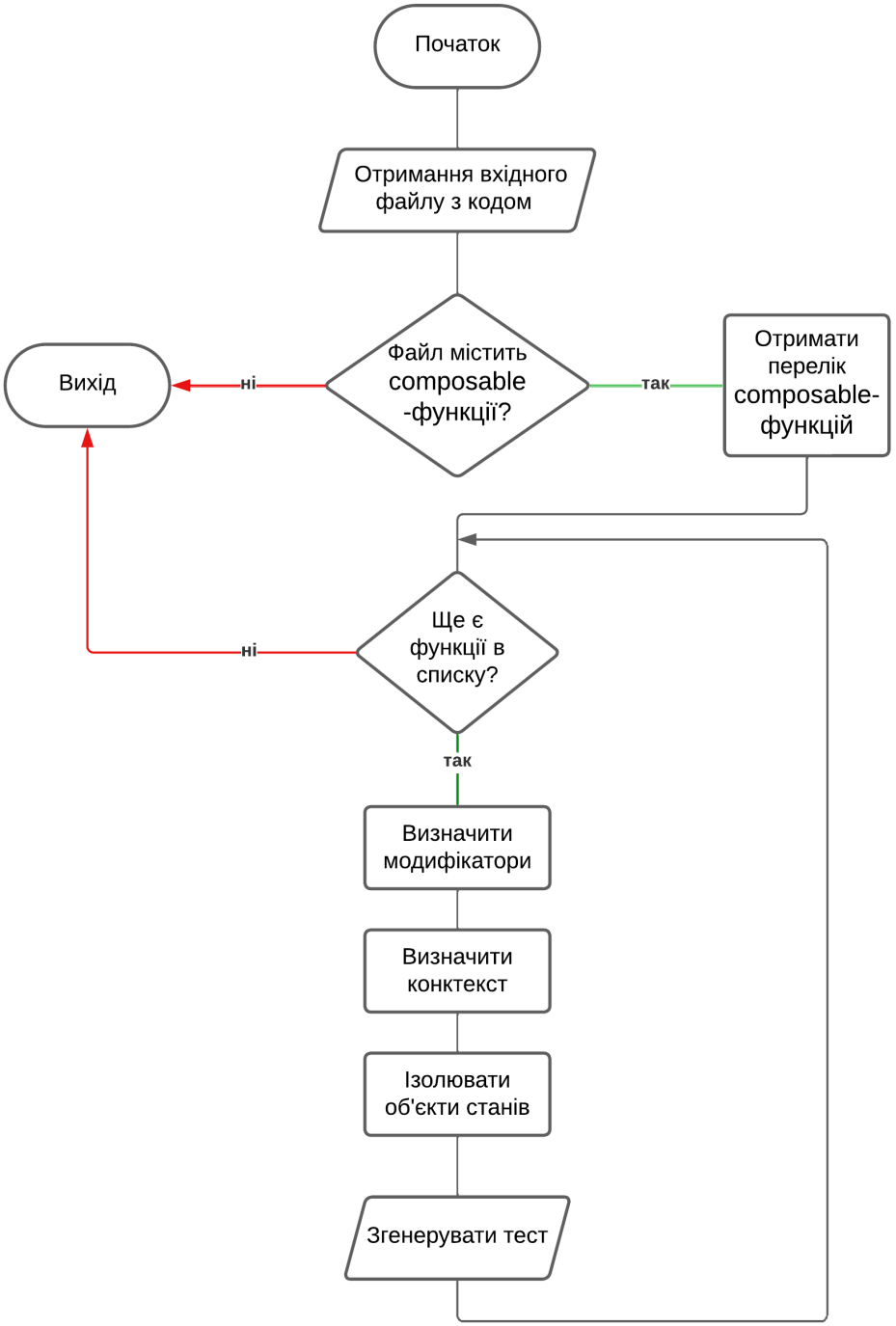

Рисунок 2.7 – Загальна блок-схема алгоритму

Розглянемо детальніше такі кроки алгоритму, як визначення умов відображення, визначення контексту, ізоляцію станів, побудову графа викликів та генерування всіх можливих сценаріїв, а також формалізацію правил до генерування коду для тестування окремих елементів інтерфейсу, розроблених з використанням composable-функцій.

2.4.1 Визначення умов відображення

Оскільки composable-функції є звичайними функціями мови Kotlin, при створенні інтерфейсу можуть використовуватись звичні для програмістів конструкції мови програмування, такі як умовні оператори та оператори множинного вибору. Під час генерування тестових сценаріїв ця особливість composable-функцій має враховуватись.

Розглянемо, як приклад лістинг програмного коду:

```
@Composable
fun MessageContainer(
    isTextShown: Boolean,
    onShowMessageClick: () -> Unit
) {
    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
    ) {
        if (isTextShown) {
            Text(
                text = "Your message is shown",
                modifier = Modifier.testTag("textMessage")
            )

            Spacer(modifier = Modifier.height(16.dp))
        }

        Button(onClick = onShowMessageClick) {
            Text(text = "Show message")
        }
    }
}
```

На даному лістингу функція MessageContainer представляє собою колонку, що відображає текст повідомлення та кнопку, що дозволяє його показати. Тобто для повноцінного тестування даної функції необхідно згенерувати два тестові сценарії: перший з передачею аргументу true, а другий – зі значенням false для параметру isTextShown.

Аналогічно для операторів множинного вибору: для повноцінного тестування ізольованого компонента необхідно, що кількість тестових сценаріїв дорівнювала кількості можливих варіантів для оператора множинного вибору (в контексті мови програмування Kotlin це оператор when).

2.4.2 Визначення контексту

Оскільки інтерфейс є багатофункціональним та складається з різноманітних компонентів, для повноцінного тестування необхідно враховувати контекст тіла функції. Під час тестування composable-функцій мають виконуватись наступні вимоги врахування контексту:

1. якщо в тілі composable-функції використовуються діалоги, спадні списки, перемикачі, галочки та інші інтерактивні елементи інтерфейсу або частина інтерфейсу є анімованою, то варто враховувати цей контекст, очікуючи завершення анімації та відкриття цих об’єктів, керуючи умовами відкриття/закриття;
2. якщо об’єкт не надає можливості керувати умовами відкриття чи закриття та зберігає стан всередині себе, то код тесту має містити перевірку того факту, що об’єкт відображається на екрані;
3. якщо елемент є клікабельним, то код тесту має містити клік по даному елементу;
4. якщо елемент представляє собою перемикач, то код тесту має містити перевірку, що елемент перебуває в групі та перемикання одного перемикача змінює стан інших;
5. якщо виклики деякий composable-функцій обгорнуті у анімовані віджети і якщо стан відображення може керуватись через параметри тестованої функції, необхідно згенерувати тести з різними умовами відображення;
6. якщо функція представляє собою текстове поле для введення паролю, необхідно упевнитись в тому, що символи не відображаються після введення тексту: в коді тесту має здійснюватись введення символів та перевірка їх наявності та відображення;
7. якщо функція представляє собою текстове поле або надає можливість для введення тексту, то необхідно з’ясувати тип клавіатури, що використовується. ОС Android Text містить наступні 12 типів клавіатур, що нативно підтримуються системою:

    7.1 Number: для введення числових значень;

7.2 Phone: для введення телефонних номерів;

7.3 DateTime: для введення дат і часу;

7.4 TextPassword: для введення паролів;

7.5 TextEmailAddress: для введення електронних адрес;

7.6 TextMultiLine: для введення багаторядкового тексту;

7.7 TextAutoComplete: для введення з пропозиціями автозавершення;

7.8 TextCapSentences: для введення з автоматичним використанням великої першої літери речень;

7.9 TextCapWords: для введення з автоматичним використанням великої першої літери слів;

7.10 TextCapCharacters: для автоматичного використання великих літер у всіх символах під час введення;

7.11 NumberDecimal: для введення десяткових чисел;

7.12 NumberSigned: для введення числових значень зі знаком.

Під час генерування коду тестів необхідно поділити їх на дві категорії: числові та символьні. До числових відносяться Number, Phone, DateTime, NumberDecimal та NumberSigned, до символьних – всі інші. Тому для повноцінного тестування composable-функції, що є текстовим полем необхідно:

– в разі якщо тип клавіатури не встановлено, то він автоматично визначається як символьний, тож код тесту має містити імітацію введення будь-яких значень;

– якщо тип клавіатури відноситься до числової, необхідно згенерувати позитивний сценарій із введенням чисел та додати перевірку на наявність введених символів у полі, та негативний сценарій: у текстове поле додається рядок випадкового тексту, що складається з літер, та додається перевірка на відсутність цих символів у полі після введення.

Аналогічно до врахування випадків відображення під час аналізу коду, під час аналізу контексту генеруються всі можливі випадки на основі переданих параметрів у функції.

2.4.3 Ізоляція станів

Рекомендованим підходом до створення інтерфейсу за допомогою composable-функцій є збереження стану у ViewModel та передача об’єктів чи даних стану, як аргументи у функцію. Це є важливою перевагою розробки інтерфейсу із використанням фреймворку Jetpack Compose, оскільки така передача параметрів дозволяє ізолювати компоненти та впливати на їх поведінку, або змінювати її, передачею інших аргументів у функцію.

Оскільки функція може приймати об’єкти різного типу, як об’єкти стандартних методів Kotlin SDK, так і додані розробником, то під час ізолювання компоненту необхідно враховувати всі можливі варіанти комбінацій даних, які можуть бути передані в функцію, тобто мають бути створені mock-об’єкти для імітації стану та stub-функції, що змінюють поля переданого об’єкта стану, якщо це передбачено в проекті.

Загальний алгоритм створення імітованих даних залежно від переданого типу мовою псевдокоду наведено нижче:

```
ПРОЦЕДУРА generateMock(<тип об’єкту>) ПОВЕРТАЄ [можливі варіанти]:
    ЯКЩО <тип об’єкту> = String ТОДІ:
        ПОВЕРНУТИ [ порожній рядок, рядок з випадковими символами ]
    ЯКЩО <тип об’єкту> = NUMBER ТОДІ:
        ПОВЕРНУТИ [ випадкові від’ємне число, нуль, додатне число ]
    ЯКЩО <тип об’єкту> = Collection ТОДІ:
        ПОВЕРНУТИ [ порожню колекцію, колекцію з імітованими об’єктами ]
    ЯКЩО <тип об’єкту> <= складний об’єкт> ТОДІ:
        ДЛЯ КОЖНОГО поля generateMock(<тип поля>)
            ПОВЕРНУТИ [згенерованих списків варіантів для кожного поля]
```

Аналогічно, для лямбда-функцій, що передаються, як параметри у Composable-функцію, та не потребують повернення даних, необхідно згенерувати порожні лямбда-функції.

Як було сказано вище, документація Google рекомендує не зберігати стан всередині composable-функцій, а, натомість, переміщувати його у ViewModel. Тож досить частим є патерн, коли у функцію передається об’єкт та його callback-функція. Розглянемо наступний лістинг коду:

```
@Composable
fun CustomTextField(
    value: String,
    onValueChange: (String) -> Unit
) {
    // …

    TextField(
        value = value,
        onValueChange = onValueChange
    )
}
```

На цьому лістингу програмного коду показано передачу параметра value в середину composable-функції та передачу його callback-функції onValueChange. Принцип роботи callback-функції полягає в тому, що під час її виклику вона зберігає стан переданого значення, наприклад, у ViewModel або в базу даних, а нове збережене значення пропагується функції CustomTextField у вигляді параметру value. Такий випадок має враховуватись під час генерування тесту, та має створюватись подібна лямбда функція, що зберігає стан змінної за допомогою механізмів remember, що надаються фреймворком Jetpack Compose, та передаватись в тестовану функцію.

2.4.4 Множини викликів та даних

Тіло composable-функції складається з викликів інших функцій та виразів, умовних операторів, циклів тощо. Ці структурні блоки створюють розгалуження виконання коду, а наявність у цих функцій різних наборів вхідних параметрів створює ще більше розгалужень при врахуванні всіх можливих варіантів.

Розглянемо лістинг програмного коду (див. с. 62).

Даний код представляє собою екран для відображення таких даних користувача, як ім’я, вік та аватар. Екран розбито на три частини: картку відображення деталей користувача, поле введення електронної пошти та кнопку відправки даних. Картка даних користувача складається з віджета для відображення фотографії аватара користувача, двох віджетів відображення тексту та кнопки для переходу в режим редагування.

Лістинг 2.1

```
data class ProfileViewData(
    val name: String,
    val age: Int,
    val avatarSrc: String
)

@Composable
fun Screen(
    user: ProfileViewData,
    isUserDataValid: Boolean,
    email: String,
    onEmailChange: (String) -> Unit,
    onEditClick: () -> Unit,
    onSubmitClick: () -> Unit
) {
    ProfileCard(
        user = user,
        isValid = isUserDataValid,
        onEditClick = onEditClick
    )
   TextField(
        value = email,
        onValueChange = onEmailChange,
        keyboardOptions = KeyboardOptions.Default.copy(
            capitalization = KeyboardCapitalization.Sentences,
            autoCorrect = false,
            keyboardType = KeyboardType.Email,
            imeAction = ImeAction.Done
        )
   )
    if (isUserDataValid) {
        Button(onClick = onSubmitClick) {
            Text(text = "Submit")
        }
    }
}

@Composable
fun ProfileCard(
    user: ProfileViewData,
    onEditClick: () -> Unit
) {
    Column {
        CoilImage(model = avatarSrc)
        Text(text = user.name)
        Text(text = user.age.toString())
        Button(onClick = onEditClick) {
            Text(text = "Edit")
        }
    }
}
```

Прорахуємо кількість різних станів для полів класу ProfileViewData, що передається у функцію ProfileCard:

– для поля "name" кількість станів 2: порожній рядок та рядок випадкових символів;

– для поля "age" кількість станів 1: оскільки не відбувається перевірок та значення одразу виходиться на екран, то достатньо згенерувати будь-яке число для імітації даних;

– для поля "avatarSrc" кількість станів 3: порожній рядок, рядок випадкових символів та валідне посилання.

Для обчислення кількості всіх можливих комбінацій цих станів без повторень можна скористатися принципом множення, який стверджує, що якщо ми маємо n_1 способів вибрати щось з першої множини, n_2 способів вибрати щось із другої множини і так далі, то всього існує $n_1 \times n_2 \times ...$ способів вибрати один елемент з кожної множини.

Отже, для функції ProfileCard кількість розгалужених варіантів складе 6. Якщо розглянути кількість розгалужень для функції Screen, то, відповідно до попереднього обчислення, кількість розгалужень дерева варіантів вхідних даних для неї складе 24, оскільки крім об’єкту user вона приймає параметр isUserDataValid, що може набувати двох значень: true чи false. Дерево розгалужень варіантів вхідних даних представлено на рис. 2.8.

Крім різноманітних наборів даних, необхідно розглянути наявність у елементів різного набору типів взаємодії, що також створює додаткові відгалуження.

Загалом інтерфейс додатку складається з таких структурних елементів, як текстові поля, кнопки, перемикачі, галочки, спадні списки, діалоги та елементи відображення графічної інформації та веб-сторінок. Якщо розділити їх типи за взаємодією користувача, то можна класифікувати їх в наступні категорії:

– введення тексту (текстові поля);

– клікабельні елементи (перемикачі, галочки, спадні списки, діалоги).

Також всі такі елементи можна віднести до категорії компонентів, що відображають певну інформацію, тобто мають бути перевірені на видимість на екрані.

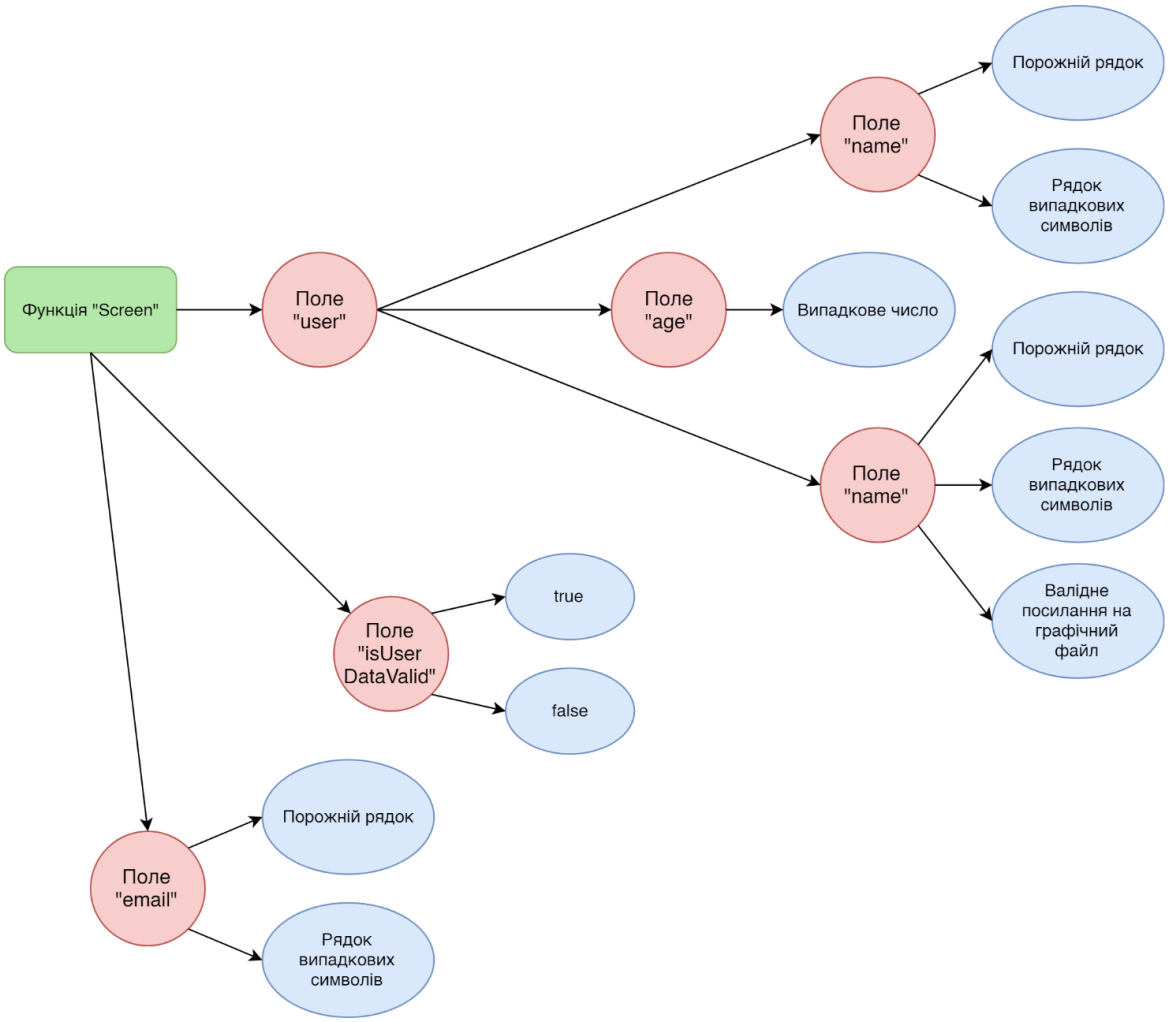

Рисунок 2.8 – Дерево розгалужень варіантів вхідних даних

Описані типи взаємодії користувача з елементами інтерфейсу при накладанні на стек викликів composable-функцій створюють дерево різновидів подій взаємодії користувача з екраном.

Розглянемо стани взаємодії, яких можуть набувати певні елементи взаємодії:

1. елементи інтерфейсу для введення тексту залежно від типу клавіатури, що застосовується:

1.1 числова клавіатура: перший сценарій позитивний із введенням числових символів, другий – негативний, із введенням буквених символів;

1.2 символьна клавіатура: якщо тип клавіатури Email, тобто для введення електронної пошти, то два сценарії: позитивний – введення

валідної пошти, негативний – введення випадкового набору символів; якщо поле не для введення електронної пошти, то єдиний позитивний сценарій із введенням випадкового набору символів (введення порожнього рядка не несе жодного смислового навантаження для тестування поля введення тексту).

2. клікабельні елементи інтерфейсу: два стани – з виконанням кліку та без нього;

3. для елементів перевірки видимості: єдиний сценарій – виконання перевірки елементу на відображення на екрані.

Маючи подібну формалізацію правил для генерації імітованих подій користувача на екрані, можемо перейти до генерування тестових сценаріїв.

2.4.5 Генерування сценарію

Правила, описані в двох попередніх підрозділах, створюють множину змінної довжини, яка може бути точно визначена лише для конкретного набору composable-функцій.

Загалом, для розрахунку загальної кількості комбінацій сценаріїв взаємодії без повторів, знову застосовуємо принцип множення та можемо вивести наступну формулу:

$$N_{\text{c}} = \prod_{i=1}^{N_B} n_i \ , \tag{2.1}$$

де N_{c} – кількості комбінацій сценаріїв взаємодії без повторів,

N_B – кількість елементів інтерфейсу, до яких застосовується генерування сценарію тестування;

n_i – кількість сценаріїв тестування, які можна застосувати до поточного елемента інтерфейсу, що тестується.

Розглянемо лістинг коду, який було наведено вище (див. с. 61-62). Функція Screen складається з функції ProfileCard текстового поля TextField та

клікабельного елемента Button.

Для текстового поля TextField вказано тип клавіатури Email, тобто кількість сценаріїв тестування для нього дорівнює 2. Для клікабельного елементу Button, в свою чергу, необхідно виконати 2 сценарії: з виконанням кліку та без нього, проте елемент Button обернено в умовний оператор, тож кількість сценаріїв саме для цього елементу збільшується вдвічі.

Аналогічно, функція ProfileCard складається з віджету для відображення на екрані зображення та двох текстових полів, для яких застосовується лише сценарії перевірки на видимість, а оскільки це лише один сценарій, то він не створює розгалуження та не впливатиме на загальну кількість комбінацій сценаріїв виконання тестування даного екрану. Елемент Button всередині функції ProfileCard має 2 сценарії: виконання кліку та без кліку. Зведемо їх у таблицю 2.1.

Таблиця 2.1 – Кількість подій користувача для інтерфейсу, що тестується

Елемент інтерфейсу	TextField	Button	CoilImage	Text	Text	Button
Кількість сценаріїв	2	4	1	1	1	2

Відповідно до формули (2.1) розраховуємо кількість імітованих подій користувача, які буде згенеровано:

$$2 \times 4 \times 1 \times 1 \times 1 \times 2 = 16$$

Тобто 16 комбінацій тестових сценарії буде застосовано для лістингу, що розглядається. Візуалізація вказаної множини тестових сценаріїв відповідно до графу викликів функцій для лістингу, що розглядається, наведено на рис. 2.9.

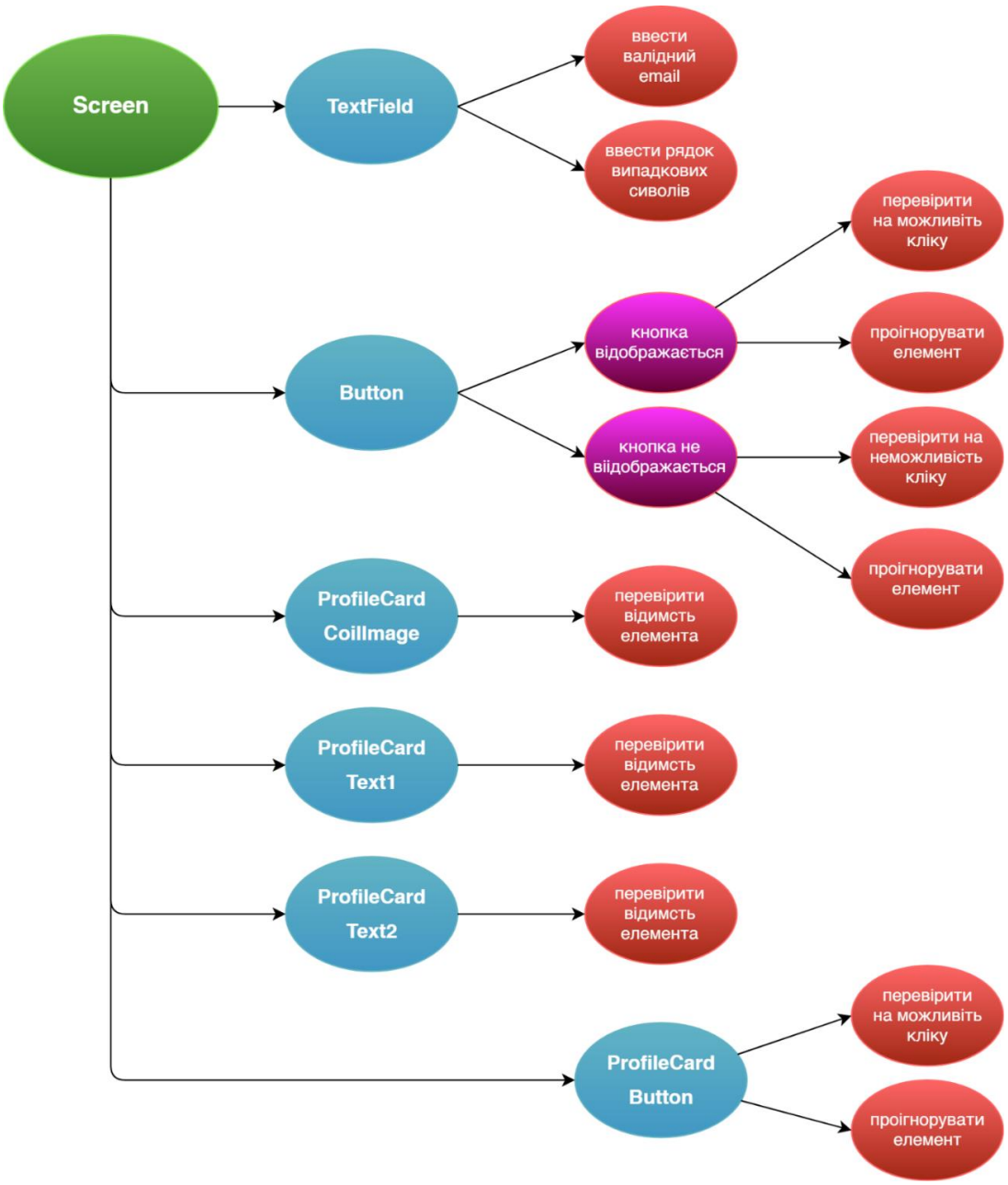

Рисунок 2.9 – Множина тестових сценаріїв для кожного елемента інтерфейсу

Враховуючи кількість комбінацій вхідних даних (див. с. 63) та розрахованих множин імітованих подій користувача, можемо розрахувати загальну кількість тестових сценаріїв:

$$N_з = 16 \times 24 = 384$$

Тобто 384 тестові сценарії можна згенерувати для лістингу коду 2.1. Описаний підхід має певні недоліки, наприклад, надмірність кількості тестових

сценаріїв, проте дану проблему можна вирішити методами налаштувань глибини сценарію, а також не включаючи певні події користувача до генерування сценарію, що значно зменшить їх кількість. Проте детальніше ці умови будуть описані у розділі програмної реалізації.

2.5 Висновки до розділу

В даному розділі було розглянуто принципи декларативної парадигми, рекомпозицій, особливості composable-функцій, їх роботи та особливості їх тестування за допомогою фреймворку Compose API для тестування, що постачається компанією Google. Даний API буде використаний під час розробки програмного забезпечення для генерації тестів графічного інтерфейсу.

Також у цьому розділі було описано загальний алгоритм генерування тестових сценаріїв, що враховує особливості побудови та роботи composable-функцій. Було описано особливості перевірок умов видимості елементів інтерфейсу, формалізовано правила ізолювання станів, створення імітованих об’єктів, генерації множин вхідних даних та застосування множин сценаріїв тестування до різнотипних елементів інтерфейсу.

Описаний метод генерування тестових сценаріїв відрізняється від аналогів тим, що покриває всі можливі множини подій користувача та вхідних даних, що дозволяє наблизити покриття коду тестами до 100%.

Використовуючи формалізовані правила та описаний алгоритм, можемо перейти до аналізу вимог та безпосередньої розробки програмного забезпечення, що реалізуватиме описаний алгоритм для генерації тестових сценаріїв для composable-функцій.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Оскільки фундаментальною основною плагіну для автоматизованого генерування коду тестів графічного інтерфейсу є мова програмування Kotlin та її декларативний підхід, то, відповідно, головною задачею розробки плагіну є перетворення вхідного файлу в абстрактне синтаксичне дерево, його аналіз та генерування тесту. Для розв’язання цієї задачі було обрано мову програмування Kotlin, IDE Intellij Idea та компілятор Kotlin Compiler Embeddable. Обґрунтуванням такого вибору є той факт, що мова даний набір надає потужні інструменти для розробки плагінів до програмних продуктів Intellij Idea та Android Studio та є рекомендованим компанією Intellij.

Починаючи з конференції Google I/O у 2019 році Kotlin став першою мовою компанії Google, на яку виходять офіційні модулі, бібліотеки та плагін для мобільної розробки під Android [67].

Відповідно до офіційної статистики на момент написання магістерської дисертації, понад 50% професійних Android розробників використовують Kotlin як основну мову, тоді як лише 30% використовують Java як основну мову [68]. 70% розробників, основною мовою яких є Kotlin, стверджують, що Kotlin робить їх роботу більш продуктивною. Згідно з внутрішніми даними Google, додатки, створені з використанням Kotlin, на 20% рідше виходять з ладу [67, 68]. І найголовнішою перевагою мови Kotlin є те, що вона нативно підтримує Jetpack Compose – рекомендований компанією Google сучасний інструментарій для створення нативного інтерфейсу користувача на Kotlin для Android.

Таким чином, мова програмування Kotlin, інтегроване середовище розробки Intellij Idea, компілятор Kotlin Compiler Embeddable та Kotlin Reflect API є найкращим вибором цільового середовища для створення засобу плагіну для автоматичного генерування тестів графічного інтерфейсу.

3.1 Аналіз вимог до програмного забезпечення

Ґрунтуючись на результатах досліджень наявних підходів, методів, рішень для автоматизації генерування тестів для графічного інтерфейсу користувача для додатків під Android, їх характеристик та недоліків, можем визначити перелік характеристик, яким має відповідати плагін для автоматичної генерації тестів:

– ефективність: плагін має коректно використовувати ресурси системи під час генерації тестів, проводити генерування у фоновому режимі, ефективно застосовуючи апаратні та програмні засоби;

– переносимість: плагін повинен легко встановлюватись на популярних інтегрованих середовищах розробки (Intellij Idea, Android Studio);

– повторюваність: плагін повинен надавати можливість багаторазового генерування тестів без необхідності внесення будь-яких змін або виконання додаткових конфігурацій;

– версіонування: плагін повинен забезпечувати облік згенерованих тестів графічного інтефейсу;

– гнучкість: плагін повинен надавати можливість програмісту змінювати згенеровані тести.

Сформулюємо функціональні та нефункціональні вимог до плагіну для генерації коду unit-тестів графічного інтерфейсу Android-додатку.

Функціональні вимоги:

1. Плагін повинен генерувати тести для коду написаного мовою програмування Kotlin.

2. Плагін повинен аналізувати вміст відкритого файлу та ідентифікувати Composable-функції.

3. Для кожної Composable-функції плагін повинен автоматично генерувати код unit-тесту.

4. Під час генерації тестів мають враховуватись параметри Modifier для кожної Composable-функції.

5. Сценарії тестів повинні охоплювати введення даних, перевірки видимості елементів та створення mock-об'єктів для імітації поведінки.

6. Плагін повинен інтегруватися з Android Studio та бути доступним через контекстне меню відкритого файлу.

7. Плагін повинен надавати можливість налаштування параметрів генерації тестів, таких як шляхи до папок для зберігання тестових файлів, шаблони коду для тестування тощо.

8. Плагін повинен забезпечувати можливість автоматичного оновлення тестів при зміні Composable-функцій у вихідних файлах.

Нефункціональні вимоги:

1. Код unit-тестів, який генерується плагіном, має бути читабельним і відповідати стандартам тестування.

2. Плагін повинен легко встановлюватись та бути простим у використанні, з мінімальною кількістю кроків для активації.

3. Розробка плагіну повинна відповідати стандартам розробки плагінів для Android Studio і бути добре документованою.

4. Плагін повинен бути стійким і надійним, без збоїв або падінь, які можуть вплинути на роботу IDE.

5. Інтерфейс користувача плагіну має бути інтуїтивно зрозумілим і зручним у використанні для розробників Android.

6. Плагін повинен бути добре документованим, з високоякісною документацією, яка пояснює всі можливості та функції плагіну.

7. Розробка плагіну повинна дотримуватися принципів тестування, включаючи автоматичне тестування самого плагіну для перевірки його працездатності та стабільності.

8. Плагін повинен бути сумісним з останніми версіями Android Studio і забезпечувати швидке оновлення для вирішення можливих проблем сумісності з майбутніми версіями IDE.

3.2 Сценарії використання

Ґрунтуючись на визначених вимогах до плагіну, можемо сформувати перелік сценаріїв використання (use cases) та визначити акторів.

Оскільки плагін призначений для роботи з єдиним користувачем – розробником, який працює в IDE зі встановленим плагіном, то єдиним актором в даній системі буде виступати «Android розробник».

Сформуємо перелік use cases для даної системи (див. табл. 3.1). До них належатимуть:

– автоматичне генерування unit-тесту для Composable-функції;
– налаштування параметрів генерації тестів;
– оновлення тестів після змін в Composable-функціях;
– перегляд та редагування згенерованого коду тестів;
– видалення згенерованих тестів.

Таблиця 3.1 – Список use case для системи, що розробляється

Актор	Android розробник
Use Case	Автоматичне генерування unit-тесту для Composable-функції
Ідентифікатор	UC-1
Опис	Розробник відкриває файл з Composable-функцією у Android Studio та викликає плагін через контекстне меню. Плагін аналізує файл, ідентифікує Composable-функцію та автоматично генерує код unit-тесту для неї.
Use Case	Налаштування параметрів генерації тестів
Ідентифікатор	UC-2
Опис	Розробник встановлює параметри генерації тестів, такі як шляхи до папок для зберігання тестових файлів та шаблони коду для тестування. Плагін використовує ці налаштування при генерації тестів.

Продовження таблиці 3.1

Use Case	Оновлення тестів після змін в Composable-функціях
Ідентифікатор	UC-3
Опис	Після зміни Composable-функції у вихідних файлах, розробник викликає плагін для оновлення відповідного unit-тесту. Плагін автоматично оновлює тест, враховуючи зміни у Composable-функції.
Use Case	Перегляд та редагування згенерованого коду тестів
Ідентифікатор	UC-4
Опис	Після генерації коду тесту розробник відкриває файл відповідного згенерованого unit-тесту.
Use Case	Видалення згенерованих тестів
Ідентифікатор	UC-5
Опис	Після генерації тесту розробник видаляє файл відповідного згенерованого unit-тесту.

Опишемо кожен з даних use case відповідно до стандартів оформлення use case у форматі User Story + AC з використанням формату GIVEN-WHEN-THEN для критеріїв прийняття (див. табл. 3.2).

Таблиця 3.2. Use Case в форматі User Story + AC

Use Case	UC-1: Автоматичне генерування unit-тесту для Composable-функції
User Story	Як розробник Android, я хочу мати можливість автоматично генерувати unit-тест для Composable-функції у моєму проекті, щоб забезпечити якісну тестування моєї UI-логіки.
Acceptance Criteria	GIVEN що я відкриваю файл з Composable-функцією у Android Studio, WHEN я викликаю плагін через контекстне меню, THEN плагін аналізує файл, ідентифікує Composable-функцію і автоматично генерує код unit-тесту для неї.
Use Case	UC-2: Налаштування параметрів генерації тестів
User Story	Як розробник Android, я хочу мати можливість налаштувати параметри генерації тестів для плагіну, щоб відповідати вимогам мого проекту.
Acceptance Criteria	GIVEN що я маю можливість налаштувати параметри генерації тестів у налаштуваннях плагіну, WHEN я встановлюю шляхи до папок для зберігання тестових файлів та шаблони коду для тестування, THEN плагін використовує ці налаштування при генерації тестів.

Продовження таблиці 3.2

Use Case	UC-3: Оновлення тестів після змін в Composable-функціях
User Story	Як розробник Android, я хочу мати можливість автоматичного оновлення тестів після внесення змін у Composable-функції, щоб забезпечити актуальність моїх тестів.
Acceptance Criteria	GIVEN що я змінив Composable-функцію у вихідному файлі мого проекту, WHEN я викликаю плагін для оновлення тестування через контекстне меню, THEN плагін автоматично оновлює відповідний unit-тест, враховуючи зміни у Composable-функції.
Use Case	UC-4: Перегляд та редагування згенерованого коду тестів
User Story	Як розробник Android, я хочу мати можливість переглядати та редагувати згенерований код unit-тесту, щоб вносити корективи або доповнювати його за потребою.
Acceptance Criteria	GIVEN що я викликав плагін для генерації тесту через контекстне меню, WHEN тест був згенерований, THEN я можу відкрити та переглянути згенерований код тесту в Android Studio, а також внести в нього зміни за необхідності.
Use Case	UC-5: Видалення згенерованих тестів
User Story	Як розробник Android, я хочу мати можливість видаляти згенеровані тести, якщо вони більше не потрібні або стали застарілими.
Acceptance Criteria	GIVEN що я маю згенеровані тести для Composable-функцій, WHEN тести більше не потрібні або стали застарілими, THEN я можу видалити їх через контекстне меню у Android Studio, і вони будуть видалені з мого проекту.

Використовуючи визначені вимоги, варіанти використання, історії користувачів та критерії прийняття, можемо перейти до розробки архітектури проектованого плагіну.

Перелік описаних варіантів використання та критерії їх прийняття, доцільно звести до діаграми прецедентів (див. рис. 3.1).

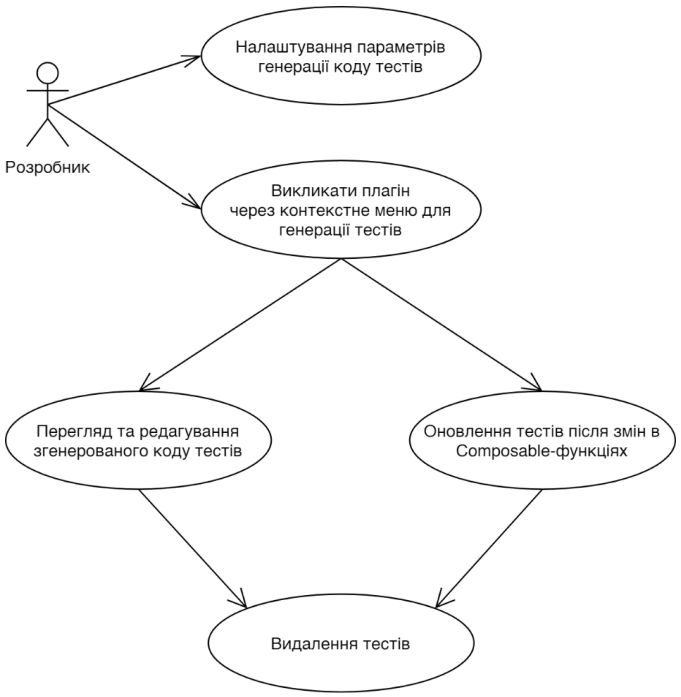

Рисунок 3.1 – Діаграма прецедентів

Відповідно до таблиці 3.1, розробнику мають бути доступні наступні варіанти використання:

– налаштовування параметрів генерації коду;
– виклик плагіну через контекстне меню;
– перегляду та видалення згенерованих тестів;
– оновлення згенерованого коду.

Цей перелік варіантів використання відповідає поставленим задачам та визначеним функціональним та нефункціональним вимогам. Інші можливі варіанти використання не розглядаються на даному етапі через недоцільність.

3.3 Архітектура програмного забезпечення

Базуючись на переліку визначених вимог до програмного забезпечення, що розробляється, та можливих архітектурних рішеннях для засобів автоматизованого генерування коду модульних тестів графічного інтерфейсу, було розроблено архітектуру плагіну, що зображена на діаграмі компонентів (рис. 3.2).

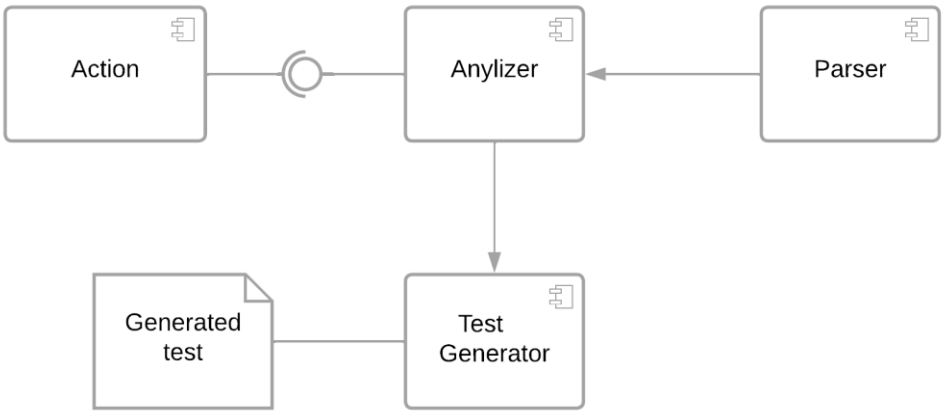

Рисунок 3.2 – Діаграма компонентів

Архітектурне рішення складається з наступних компонентів:

– вхідна та вихідна точка плагіну Action: пов’язує всі компоненти архітектури між собою, відповідає за обробку файлу, перевірку чи цей файл є файлом коду мовою програмування Kotlin та чи містить такий файл Composable-функції;

– парсер: вбудований інструмент компілятора Kotlin, що конвертує код з файлу в абстрактне синтаксичне дерево;

– аналізатор: компонент аналізу абстрактного синтаксичного дерева коду та пошуку в ньому вузлів (node), що мають бути протестовані, враховуючи контекст (напр., в разі тестування діалогу, якщо він обернений в умовний блок, то необхідно додати два тести, передавши умову true та false, щоб упевнитись в правильності відображення діалогу на екрані);

– генератор тесту: компонент, що відповідає за створення файлу тесту, та генерування коду тесту відповідно до списку вузлів та умов, що були передані аналізатором;

– програмний контекст: програмне оточення системи.

Ґрунтуючись на розробленій діаграмі компонентів та враховуючи особливості реалізації обраних бібліотек, можемо перейти до проектування діаграми класів.

Спрощена діаграма класів для плагіну, що розробляється, наведена в «Додаток Б» на рис. Б.1. Головним класом плагіну є ProcessKtFileAction, в який

композиційно вбудовується TestGenerator. Об’єкт TestGenerator не може існувати або виконувати аналіз файлів без класу ProcessKtFileAction, оскільки той передає йому контекст виконання, тобто поточний проект та обраний файл. Далі в TestGenerator композиційно вбудовуються об’єкти класів FileManager та CodeAnalyzer.

Головною сутністю, в яку перетворюється проаналізована composable-функція є FunctionTest, яка зберігає такі дані, як тег тестованого об’єкту інтерфейсу, параметри, типи сценаріїв, які можна застосувати до функції та ін.

3.4. Реалізація функціоналу програмного забезпечення

Як було визначено на попередніх кроках ядром додатку є Action, який представлено за допомогою ProcessKtFileAction. Єдиним його методом є actionPerformed, який викликається з контексту редактора IntellijIdea або Android Studio та передає відкритий на редагування файл плагіну.

З класу ProcessKtFileAction управління обраним файлом передається класу TestGenerator, який приймає обраний файл через метод generateTest. Класс TestGenerator має дві композиційні залежності із класами CodeAnalyzer та FileManager, оскільки об’єкти створюються всередині класа TestGenerator та знищуються після знищення об’єкта TestGenerator.

CodeAnalyzer приймає файл та містить методи для знаходження composable-функцій, аналізу контексту, створення mock-об’єктів. Він повертає список сформованих тестів FunctionTest для кожної знайденої функції.

Результатом виконання методу analyze класу CodeAnalyzer є дерево у форматі графу, що являє собою структуру викликів функцій, методів тестування, які можна до них застосувати, та імітованих об’єктів, для реалізації імітування та ізоляції різних компонентів тестованого інтерфейсу.

CodeAnalyzer визначає перелік модифікаторів, що передаються в composable-функцію та перевіряє їх на наявність клікабельних елементів, або елементів, що підтримують введення тексту. Якщо в тілі composable-функції використовуються діалоги, спадні списки, перемикачі, галочки та інші

інтерактивні елементи інтерфейсу або частина інтерфейсу є анімованою, то CodeAnalyzer враховує цей контекст, очікуючи завершення анімації та відкриття цих об’єктів, керуючи умовами відкриття/закриття. Якщо в тілі composable-функції є звичайні конструкції мови Kotlin, такі як умовні оператори та оператори множинного вибору, CodeAnalyzer визначає їх та, в разі можливості, враховує це, змінюючи вхідні параметри для composable-функції. Якщо об’єкти стану передаються в тіло composable-функції як параметр, то CodeAnalyzer створює mock-об’єкти для імітації стану та забезпечення ізоляції модулів під час тестування.

Клас FileManager відповідає за створення файлів тестів для кожної composable-функції, перетворення сформованого об’єкту FunctionTest в код та запис його в файл. Створення тестів відбувається на основі генерування всіх можливих варіантів тестових сценаріїв для даного набору тестових параметрів.

Інші процеси роботи плагіну є стандартними для будь-якого з наявних плагінів для інтегрованого середовища розробки Android Studio: це виклик через контекстне меню редактора коду, генерування коду тесту для обраного файлу, запуск згенерованого коду тесту на виконання та перевірка результатів виконання згенерованих сценаріїв тестування графічного інтерфейсу.

Останнім етапом роботи розробленого плагіну для автоматизованого генерування модульних тестів є вивільнення використаних ресурсів системи та завершення виконання плагіну. Після завершення роботи плагін передає управління інтегрованому середовищу розробки Android Studio. Код згенерованих тестів при цьому зберігаються у відповідному каталозі проекту та є доступним розробнику для редагування. Якщо виникає помилка, то плагін звільняє ресурси та припиняє свою роботу, надсилаючи повідомлення про помилку генерації тесту користувачу.

3.5 Оптимізація генерування тестів

Оскільки на практиці зазвичай ніхто не тестує усі можливі випадки для екранів, а генерування всіх можливих комбінацій може бути часо- та

ресурсозатратною задачею, необхідним є введення додаткових механізмів оптимізації генерування тестів

Для оптимізації тестування та уникнення генерування зайвого коду, було введено конфігураційний файл у форматі JSON, що дозволяє змінювати параметри генерування. Даний конфігураційний файл складається з переліку параметрів, до яких входять наступні: "excludeTags", "globalRules" та "forNode".

Параметр "excludeTags" дозволяє виключити тестування для елементів інтерфейсу з відповідними тестовими тегами. Параметр "globalRules" дозволяє встановити глобальні правила для генерування тестів та включати або виключати з генерування певні підходи. Параметр "forNode" дозволяє встановлювати окремі правила для генерування для певного елементу інтерфейсу з відповідним тегом.

Загальний вигляд файлу наведено на лістингу нижче:

```
{
    "excludeTags": [],
    "globalRules": {
        "applyClickIgnore": true,
        "useEmptyStrings": false,
        "useNegativeNumbers": false,
        "testConditionals": true
    },
    "forNode": [
        {
            "nodeTag": <тег ноди>,
            "rules": {
                // ….
            }
        }
    ]
}
```

Даний механізм оптимізації дозволяє розробнику керувати процесом генерування тестів та надає досить гнучкий механізм для цього.

3.6 Висновки до розділу

Архітектурне рішення та реалізація ПЗ, описані в даному розділі, відповідають функціональним та нефункціональним вимогам, поставленим у

розділі 3.1.

Розроблене програмне забезпечення дозволяє автоматизовано генерувати код модульних тестів графічного інтерфейсу користувача і, на відміну від наявних аналогів, підтримує тестування інтерфейсу розробленого з використанням декларативної парадигми, а саме, фреймворку Jetpack Compose, та мови програмування Kotlin, автоматично визначає всі можливі сценарії та правила тестуванні, які можна застосувати до визначеного набору вхідних параметрів для обраної Composable-функції.

Крім того, розроблений пропонує додаткові інструменти для проведення модульного тестування, а саме створення фіктивних даних стану та функцій для реалізації вимоги ізоляції компонент одна від одної.

4 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Порядок проведення дослідження та середовище

Розробка будь-якого нового продукту завершується тестуванням, а проведене дослідження має підтверджуватись результатами експериментів. Оскільки наявні на ринку інструменти та плагіни для генерування коду функціональних модульних тестів графічного інтерфейсу не підтримують декларативну парадигму та мову програмування Kotlin, а фреймворк Jetpack Compose розроблений лише для Kotlin та не підтримує Java, ми не можемо провести порівняння швидкодії інструментів для автоматизованого генерування коду тестів. Тому для оцінки ефективності розробленого рішення пропонується провести наступні дослідження:

– визначити швидкодію генерування тестових сценаріїв;

– розрахувати економію часу розробника на генерування отриманої кількості тестових сценаріїв.

Для проведення дослідження використовувалось апаратне забезпечення з такими характеристиками:

– процесор: 2-ядерний Intel® Core™ i5-5300U з частотою 2.29 ГГц;

– оперативна пам’ять (RAM): 16 ГБ SODIMM 1600 МГц;

– графічний процесор: Intel® HD Graphics 5500;

– операційна система: 64-розрядна Windows 8.1 Professional.

4.2 Швидкодія генерування тестів

Визначальною характеристикою для будь-якого інструменту для розробника є його швидкодія, оскільки будь-який такий інструмент призначений для прискорення та полегшення виконання роботи розробника.

Як було визначено у параграфі 2.4.5, для лістингу програмного коду, що розглядався (див. с. 63), кількість тестових сценаріїв склала 384. Результати часу, який зайняло автоматизоване генерування вказаної кількості сценаріїв наведено

у таблиці 6.

Також було проведено ряд тестів, під час яких було протестовано роботу розробленого плагіну на різних проектах та наборах composable-функцій. Результати експерименту подано в таблиці 3.3 та на рис. 3.3.

Таблиця 3.3 – Результати експериментальних досліджень

Кількість вхідних параметрів	50	120	384	458	600
Час на генерування, с	17	24	36	44	52

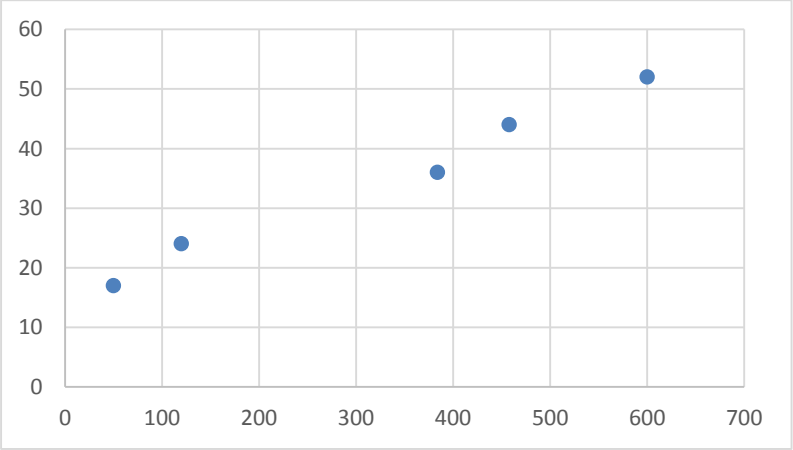

Рисунок 3.3 – Результати експериментального дослідження

Як видно, результати не ідеально лінійні, оскільки швидкодія генерування залежить не тільки від самого генерування та запису в файл, а й від складності коду, що аналізується та кількості вкладень, проте загальний тренд чітко простежується.

4.3 Економія часу на написання тестових сценаріїв

Оскільки ми не можемо порівняти швидкодію результату зі швидкодією існуючих аналогів, ми можемо порівняти їх зі швидкістю написання аналогічного тесту середньостатистичним Android-розробником. Код прикладу згенерованого сценарію для лістингу 1, що розглядався, наведено на лістингу

нижче:

```
package io.jctest.app
import androidx.compose.ui.test.assertIsDisplayed
import androidx.compose.ui.test.junit4.createComposeRule
import androidx.compose.ui.test.onNodeWithText
import androidx.compose.ui.test.performClick
import androidx.test.ext.junit.runners.AndroidJUnit4
import io.jctest.app.ui.screen. HomeScreen
import io.jctest.app.ui.theme.JcTestTheme
import org.junit.Rule import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.Mockito.mock import org.mockito.Mockito.whenever

@RunWith(AndroidJUnit4::class)
class HomeScreenTest {

    @get: Rule
    val composeTestRule = createComposeRule()

    @Test
    fun `test`() {
        val val profileViewDataMock = mock(ProfileViewData::class.java)
        whenever(profileViewDataMock.name).thenReturn("John")
        whenever (profileViewDataMock.age).thenReturn(30)
        whenever (profileViewDataMock.avatarSrc).thenReturn("path/")

        val isUserDataValid = true
        val onEditClick: () -> Unit = {}
        val onSubmitClick: () -> Unit = {}

        composeTestRule.setContent {
            val email by remember {
                mutableStateOf("")
            }

            val onEmailChange: (String) -> Unit = { email = it }

            Screen(
                user = profileViewDataMock,
                isUserDataValid = isUserDataValid,
                email = email,
                onEmailChange = onEmailChange,
                onEditClick = onEditClick,
                onSubmitClick = onSubmitClick
            )

        composeTestRule.onNodeWithText("coilImage").assertIsDisplayed()
        composeTestRule.onNodeWithText("text1").assertIsDisplayed()
        composeTestRule.onNodeWithText("text2").assertIsDisplayed()
        composeTestRule.onNodeWithText("editButton").performClick()
        composeTestRule.onNodeWithText("textfield"). performTextInput("examp
        composeTestRule.onNodeWithText("submitButton").performClick()
    }
}
```

Якщо швидкість написання коду середньостатистичного програміста складає 200-250 символів/хв і якщо відкинути частину коду з імпортами, то код тесту, складає 1507 символів, тобто навіть за написання тесту зі швидкістю 250

символів/хв програміст напише його за 6 хв. При цьому якщо врахувати, що кількість часу, необхідна розробнику на аналіз коду, продумування тестового сценарію, продумування імітованих даних тощо, і задати цей час в 4 хв, то мінімальний час написання цього простого тесту складе 10 хв.

Порівняти отримані дані наочно можна, звівши їх в таблицю (табл. 3.4) та представивши у вигляді стовпчикових діаграм (рис. 3.4).

Таблиця 3.4 – Порівняння часу на написання тестів

	Програміст	**Розроблений плагін**
Кількість написаних тестів	1	384
Час, с	600	36

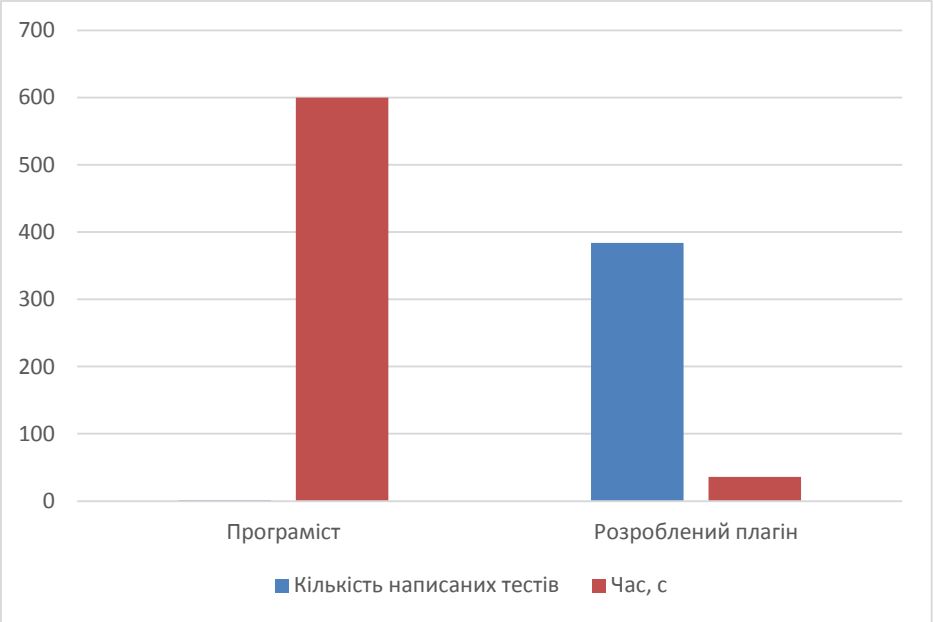

Рисунок 3.4 – Порівняння часу на написання тестів

Загалом, економію часу можна виразити наступною формулою:

$$E = \frac{(t_H + t_A) \times N_K}{t_{Nk}} \quad (3.1)$$

де E – економія часу;

t_H – час, що витрачає розробник на написання тесту;

t_A – час, що витрачається розробником на аналіз коду, ізоляцію блоків та створення імітованих об’єктів;

N_K – кількість комбінацій множини подій користувача та імітованих даних;

t_{Nk} – час, що витрачається плагіном на генерування тестових сценаріїв для всіх комбінацій множини імітованих подій та даних.

Як видно з результатів (табл. 3.4, рис. 3.4), плагін значно прискорює написання тестових сценаріїв та за 3.5% часу, що витрачає програміст, дозволяє швидко покривати в 384 рази більше сценаріїв, тобто всі можливі тестові сценарії для заданого екрану.

4.3 Висновки до розділу

В даному розділі було досліджено швидкодію запропонованого алгоритму для генерування коду модульних тестових сценаріїв та економію часу для розробника при використанні запропонованого плагіну.

Як показали експериментальні дослідження, економія часу розробника на написання модульних тестів при використанні пропонованого плагіну, хоч і залежить від складності коду, що тестується, проте є дуже значною.

В порівнянні з аналогами, розглянутими в параграфах 1.4-1.5, що спрямовані за своєю більшістю на генерування випадкових подій або обмеженої кількості подій, розроблений плагін спрямований на врахування всіх можливих комбінацій подій користувача та різновидів вхідних даних. Плагін не потребує знань тестування та навичок написання тестів та дозволяє швидко генерувати тестові сценарії для компонентів інтерфейсу, покриваючи всі можливі сценарії.

ВИСНОВКИ

В даній магістерській дисертації розроблено методи та засоби для автоматизованого генерування коду сценаріїв модульного тестування, які розширюють можливості використання сучасних інтегрованих середовищ розробки, шляхом розширення підтримки таких сучасних підходів до розробки Android-додатків, як декларативна парадигма розробки інтерфейсів, мова програмування Kotlin та фреймворк Jetpack Compose.

В процесі виконання роботи було досліджено сучасні моделі та види тестування, підходи до опису сценаріїв автоматизованого тестування. Дослідження наукових праць, доступних та наявних на ринку інструментів автоматичної генерації коду тестів для інтегрованих середовищ розробки під Android виявило, що абсолютна більшість з існуючих інструментів не підтримує сучасні підходи до розробки інтерфейсів, такі як декларативна парадигма, тобто не задовольняють нагальних потреб розробників у підтримці мови програмування Kotlin та підтримці декларативної парадигми.

Для розв'язання даної проблеми запропоновано удосконалений метод генерування коду модульних тестів, який передбачає:

- парсинг вихідного коду інтерфейсу та його структуризацію у вигляді дерева викликів composable-функцій, що рекурсивно та по низхідній враховує виклики інших composable-функцій;
- аналіз стану composable-функцій та врахування контексту виклику, з ізолюванням блоків стану та callback-методів у виглядів імітованих mock-об’єктів та stub-функцій;
- генерування всіх можливих варіантів сценаріїв для визначеного набору аргументів та дерева викликів;
- генерування позитивних та негативних тестових сценаріїв.

З метою реалізації запропонованих рішень був розроблений плагін для автоматизованого генерування модульних тестів для інтегрованого середовища розробки Android Studio, який дозволяє генерувати тести для мови

програмування Kotlin та виконувати згенеровані сценарії тестування графічного інтерфейсу за допомогою системи автоматичного збирання Gradle.

Для розробки плагіну було використано мову програмування Kotlin та інтегроване програмне середовище Intellij Idea. Все перераховане є широко вживаними інструментами серед розробників по всьому світу.

Результати роботи над магістерською дисертацією опубліковані на V та VI Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології SoftTech-2023» [1, 2].

Наукова новизна даної роботи полягає в удосконаленні методу створення сценаріїв автоматизованого тестування, що відрізняються від існуючих аналогів врахуванням всіх можливих викликів елементів інтерфейсу та комбінацій множин даних та наявністю підтримки автоматизованого генерування коду модульних тестів графічного інтерфейсу користувача, розробленого з використанням декларативної парадигми та фреймворку Jetpack Compose, чого раніше не було в середовищах розробки додатків під ОС Android. Розроблене ПЗ дає можливість збільшити покриття компоненту тестами та значно зекономити час розробників, оскільки для підтримки тестів достатньо перегенерувати їх в автоматичному режимі. При цьому надано можливість використовувати плагін безкоштовно та розширювати його іншим розробникам

Таким чином, дана робота не тільки вносить практичну новизну в область розробки мобільних додатків на Android, але й відкриває нові можливості для розробників у забезпеченні якості та надійності ПЗ, тобто усі поставлені завдання для роботи виконані повною мірою, а поставлена мета – досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дубовик А. П. Генерація тестів GUI Android-додатків: аналіз сучасних підходів та виклики декларативної парадигми. Інженерія програмного забезпечення і передові інформаційні технології (Soft Tech-2023): матеріали V Міжнародної наук.-практ. конф. молодих вчених та студентів, м. Київ, 19-21 грудня 2023 року, НТУУ «Київський політехнічний інститут імені Ігоря Сікорського», 2023. С. 108-113.

2. Дубовик А. П. Особливості генерування коду модульних тестів GUI Android-додатків, розроблених з використанням декларативної парадигми (Soft Tech-2024): матеріали VI Міжнародної наук.-практ. конф. молодих вчених та студентів, м. Київ, 20-21 травня 2024 року, НТУУ «Київський політехнічний інститут імені Ігоря Сікорського», 2024.

3. Jetpack Compose UI – App Development Toolkit. Android Official Documentation. [Електронний ресурс] // Режим доступу: https://developer.android.com/jetpack/compose (дата звернення: 01.12.2023)

4. Catalin Ghita, Kickstart Modern Android Development with Jetpack and Kotlin: Enhance your applications by integrating Jetpack and applying modern app architectural concepts, Packt Publishing, 2022.

5. Google, "Develop android apps with kotlin", [Електронний ресурс] // Режим доступу:: https://developer.android.com/kotlin, 2020, (дата звернення: 01.12.2023).

6. GitHub, "The state of the octoverse: Top languages", 2019, [Електронний ресурс] // Режим доступу: https://octoverse.github.com/#top-languages (дата звернення: 01.12.2023).

7. D. Winer, "Android’s commitment to kotlin", [Електронний ресурс] // Режим доступу: https://android-developers.googleblog.com/2019/12/androids-commitment-to-kotlin.html (дата звернення: 15.12.2023).

8. AppBrain, "Android statistics / android libraries / kotlin", [Електронний ресурс] // Режим доступу: https://www.appbrain.com/stats/libraries/details/kotlin

/kotlin (дата звернення: 15.12.2023)

9. G. Hecht and A. Bergel, "Quantifying the adoption of Kotlin on Android stores: Insight from the bytecode", 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), Madrid, Spain, 2021, pp. 94-98, doi: 10.1109/MobileSoft52590.2021.00019.

10. JUnit5: Official Documentation [Електронний ресурс]. Режим доступу: https://junit.org/junit5/.

11. Mockito framework site [Електронний ресурс]. Режим доступу: https://site.mockito.org/.

12. Robolectric: Official Documentation [Електронний ресурс]. Режим доступу: https://robolectric.org/.

13. Appium: Official Documentation [Електронний ресурс]. Режим доступу: https://appium.io/docs/en/latest/.

14. Espresso: Official Android Documentation [Електронний ресурс]. Режим доступу: https://developer.android.com/training/testing/espresso.

15. UI Automator: Official Android Documentation [Електронний ресурс]. Режим доступу: https://developer.android.com/training/testing/other-components/ui-automator.

16. Calabash: Automated Functional testing for Android using cucumber [Електронний ресурс]. Режим доступу: https://github.com/calabash/calabash-android.

17. MonkeyRunner: Official Android Documentation [Електронний ресурс]. Режим доступу: https://developer.android.com/studio/test/monkeyrunner

18. Robotium: Android UI Testing [Електронний ресурс]. Режим доступу: https://github.com/RobotiumTech/robotium.

19. Beizer B. Software Testing Techniques / Boris Beizer., 1990. – 580 p. – (2nd Edition).

20. Mike Cohn. Succeeding with Agile. Software Development Using Scrum: // Mike Cohn. Addison-Wesley Professional. 2010. – 512 p.

21. Fields J. Working Effectively with Unit Tests // Jay Fields. Leanpub.

2015. – 347 p.

22. Kong, Pingfan, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé and Jacques Klein. “Automated Testing of Android Apps: A Systematic Literature Review.” IEEE Transactions on Reliability 68 (2019): pp. 45-66.

23. Nie, Liming, Kabir S. Said, Lingfei Ma, Yaowen Zheng and Yangyang Zhao. “A systematic mapping study for graphical user interface testing on mobile apps.” IET Softw. 17 (2023): pp. 249-267.

24. Samir, Amira & Amin, Huda & Badr, Nagwa. (2022). A survey on automated user interface testing for mobile applications. [Електронний ресурс]: pp. 126-136. 10.21608/ijicis.2022.98138.1124.

25. Usman, Asmau & Ibrahim, Noraini & Anka, Salihu. (2020). TEGDroid: Test Case Generation Approach for Android Apps Considering Context and GUI Events. International Journal on Advanced Science, Engineering and Information Technology [Електронний ресурс]. 10. 16. 10.18517/ijaseit.10.1.10194.

26. Machiry, Aravind & Tahiliani, Rohan & Naik, Mayur. (2013) [Електронний ресурс]: Dynodroid: An input generation system for Android apps. pp. 224-234. 10.1145/2491411.2491450.

27. J. Doyle, T. Saber, P. Arcaini and A. Ventresque, "Improving Mobile User Interface Testing with Model Driven Monkey Search" [Електронний ресурс]: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto de Galinhas, Brazil, 2021, pp. 138-145, doi: 10.1109/ICSTW52544.2021.00034.

28. Köroğlu, Yavuz & Sen, Alper. (2020) [Електронний ресурс]: Functional test generation from UI test scenarios using reinforcement learning for android applications. Software Testing, Verification and Reliability. pp. 31. 10.1002/stvr.1752.

29. Yasin, Husam & Ab hamid, Siti hafizah & Raja Yusof, Raja. (2021) [Електронний ресурс]: DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning. Symmetry. pp. 13. 310. 10.3390/sym13020310.

30. Kabir S. Said, Liming Nie, Adekunle A. Ajibode, and Xueyi Zhou. 2021 [Електронний ресурс]: GUI testing for mobile applications: objectives, approaches

and challenges. In Proceedings of the 12th Asia-Pacific Symposium on Internetware (Internetware '20). Association for Computing Machinery, New York, NY, USA, pp. 51–60. https://doi.org/10.1145/3457913.3457931

31. Mariani, L.; Pezze, M.; Riganelli, O.; Santoro, M. Autoblacktest: Automatic black-box testing of interactive applications. In Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, 17–21 April 2012; pp. 81–90.

32. Esparcia-Alcázar, A.I.; Almenar, F.; Martínez, M.; Rueda, U.; Vos, T. Q-learning strategies for action selection in the TESTAR automated testing tool. In Proceedings of the 6th International Conferenrence on Metaheuristics and Nature Inspired Computing (META 2016), Marrakech, Morocco, 27–31 October 2016; pp. 130–137.

33. Kim, J.; Kwon, M.; Yoo, S. Generating test input with deep reinforcement learning. In Proceedings of the IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST), Gothenburg, Sweden, 28–29 May 2018; pp. 51–58.

34. Koroglu, Y.; Sen, A.; Muslu, O.; Mete, Y.; Ulker, C.; Tanriverdi, T.; Donmez, Y. QBE: QLearning-based exploration of android applications. In Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Luxembourg, 18–22 March 2013; pp. 105–115.

35. Vuong, T.A.T.; Takada, S. A reinforcement learning based approach to automated testing of Android applications. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Lake Buena Vista, FL, USA, 5 November 2018; pp. 31–37.

36. Adamo, D.; Khan, M.K.; Koppula, S.; Bryce, R. Reinforcement learning for Android GUI testing. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Lake Buena Vista, FL, USA, 5 November 2018; pp. 2–8.

37. Gu, T.; Cao, C.; Liu, T.; Sun, C.; Deng, J.; Ma, X.; Lü, J. Aimdroid: Activity-insulated multi-level automated testing for android applications. In Proceedings of the IEEE International Conference on Software Maintenance and

Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 103–114.

38. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. A Deep Learning based Approach to Automated Android App Testing. arXiv 2019, arXiv:1901.02633.

39. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. DroidBot: A lightweight UI-guided test input generator for Android. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 23–26.

40. Chen, T.Y.; Kuo, F.-C.; Merkel, R.G.; Tse, T. Adaptive random testing: The art of test case diversity. J. Syst. Softw. 2010, 83, pp. 60–66.

41. Google. UI/Application Exerciser Monkey|Android Developers. [Електронний ресурс] // Режим доступу: https://developer.android.com/studio/test /monkey.

42. Wang, W.; Li, D.; Yang, W.; Cao, Y.; Zhang, Z.; Deng, Y.; Xie, T. An empirical study of android test generation tools in industrial cases. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 738–748.

43. Mahmood, R.; Mirzaei, N.; Malek, S. Evodroid: Segmented evolutionary testing of android apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 11 November 2014; pp. 599–609.

44. Clapp, L.; Bastani, O.; Anand, S.; Aiken, A. Minimizing GUI event traces. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 422–434.

45. Zheng, H.; Li, D.; Liang, B.; Zeng, X.; Zheng, W.; Deng, Y.; Lam, W.; Yang, W.; Xie, T. Automated test input generation for android: Towards getting there in an industrial case. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, Argentina, 20–28 May 2017; pp. 253–262.

46. Hu, C.; Neamtiu, I. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test,

Waikiki, Honolulu, HI, USA, 23–24 May 2011; pp. 77–83.

47. Machiry, A.; Tahiliani, R.; Naik, M. Dynodroid: An input generation system for android apps. In Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 224–234.

48. Haoyin, L. Automatic android application GUI testing—A random walk approach. In Proceedings of the International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, India, 22–24 March 2017; pp. 72–76.

49. Azim, T.; Neamtiu, I. Targeted and depth-first exploration for systematic testing of android apps. In Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, Indianapolis, IN, USA, 26–31 October 2013; pp. 641–660.

50. Su, T.; Meng, G.; Chen, Y.; Wu, K.; Yang, W.; Yao, Y.; Pu, G.; Liu, Y.; Su, Z. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 245–256.

51. Baek, Y.-M.; Bae, D.-H. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 238–249.

52. Yang, W.; Prasad, M.R.; Xie, T. A grey-box approach for automated GUI-model generation of mobile applications. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering, Rome, Italy, 16–24 March; pp. 250–265.

53. Hao, S.; Liu, B.; Nath, S.; Halfond, W.G.; Govindan, R. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, Bretton Woods, NH, USA, 16–19 June 2014; pp. 204–217.

54. Gu, T.; Sun, C.; Ma, X.; Cao, C.; Xu, C.; Yao, Y.; Zhang, Q.; Lu, J.; Su, Z. Practical GUI testing of Android applications via model abstraction and refinement.

In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 269–280.

55. Adamsen, C.Q.; Mezzetti, G.; Møller, A. Systematic execution of android test suites in adverse conditions. In Proceedings of the International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 13–17 July 2015; pp. 83–93.

56. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. Automatically discovering, reporting and reproducing android application crashes. In Proceedings of the IEEE International Conference on Software Testing, Verification And Validation (ICST), Chicago, IL, USA, 11–15 April 2016; pp. 33–44.

57. Hu, G.; Yuan, X.; Tang, Y.; Yang, J. Efficiently, effectively detecting mobile app bugs with appdoctor. In Proceedings of the Ninth European Conference on Computer Systems, Amsterdam, The Netherlands, 14–16 April 2014; p. 18.

58. Mirzaei, N.; Bagheri, H.; Mahmood, R.; Malek, S. Sig-droid: Automated system input generation for android applications. In Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, USA, 2–5 November 2015; pp. 461–471.

59. Mao, K.; Harman, M.; Jia, Y. Sapienz: Multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 94–105.

60. Yasin, Husam N., Siti Hafizah Ab Hamid, and Raja Jamilah Raja Yusof. 2021. "DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning" Symmetry 13, no. 2: 310. https://doi.org/10.3390/sym13020310

61. Symflower for IntelliJ IDEA - Smart Unit Test Generator for Java. Official Website. [Електронний ресурс] // Режим доступу: https://symflower.com/en/ (дата звернення: 12.02.2024)

62. AI-Powered Java Testing Tool - Boost Productivity, Official Website. [Електронний ресурс] // Режим доступу: https://www.parasoft.com/products/para soft -jtest/ (дата звернення: 13.02.2024)

63. EvoSuite | Automatic Test Suite Generation for Java, Official Website.

[Електронний ресурс] // Режим доступу: https://www.evosuite.org/ (дата звернення: 15.02.2024)

64. Google, Thinking in Compose. Official Documentation. [Електронний ресурс] // Режим доступу: https://developer.android.com/jetpack/compose/mental-model (дата звернення: 18.02.2024)

65. Google, Compose modifiers. Official Documentation. [Електронний ресурс] // Режим доступу: https://developer.android.com/jetpack/compose/modifiers (дата звернення: 20.02.2024)

66. Google, Testing your Compose layout. Official Documentation. [Електронний ресурс] // Режим доступу: https://developer.android.com/jetpack/compose/testing (дата звернення: 22.02.2024)

67. Google, Android’s Kotlin-first approach. Official Documentation. [Електронний ресурс] // Режим доступу: https://developer.android.com/kotlin/first (дата звернення: 22.02.2024)

68. Kotlin for Android. Official Documentation. [Електронний ресурс] // Режим доступу: https://kotlinlang.org/docs/android-overview.html (дата звернення: 22.02.2024)

ДОДАТОК А

Таблиця А.1 – Огляд інструментів генерації тестових кейсів

№	Засіб	Підхід	Підхід до генерації тестових кейсів	Вхідні дані	Середовище тестування	Тестові артефакти	Основа
1	APE	Динамічний аналіз	Model-based	User	Real Device, Emulator	Crash, coverage reports	Monkey
2	Humanoid	Динамічний аналіз	Deep Q Network	User, System	Real Device, Emulator	Log, coverage report	Droidbot
3	AndroFrame	Динамічний аналіз	Q-Learning-Based	User, System	Real Device, Emulator	Log	-
4	(Adamo, Khan, et al.)	Динамічний аналіз	Q-Learning-	User	Emulator	Coverage report	Appium
5	(Vuong and Takada)	Динамічний аналіз	Q-Learning-	User	Real Device, Emulator	Log, coverage report	UI Automator
6	AimDroid	Динамічний аналіз	Model-based/SARSA	User	Real Device, Emulator	Crash, coverage reports	Monkey
7	DroidBot	Динамічний аналіз	Model-based	User, System	Real Device, Emulator	Log, coverage reports	-
8	SmartMonkey	Динамічний аналіз	Random based	User, System	-	-	Monkey
9	Stoat	Гібридний	Model-based	User, System	Real Device, Emulator	Log, crash and coverage reports	A3E
10	Sapienz	Гібридний	Search-based/Random	User, System	Emulator	Log, crash and coverage report	Monkey
11	Sig-Droid	Статичний аналіз	Systematic	User	Emulator	Log	Java PathFinder
12	AppDoctore	Гібридний	Random based	User	Real Device	Log	Monkey
13	DroidCrawle	Динамічний аналіз	Model-based	User	Emulator	Coverage report	-
14	Dynodroid	Динамічний аналіз	Guided/Random	User, System	Emulator	Coverage and crash reports	Monkey Runner
15	ORBIT	Динамічний аналіз	Model-based	User	Real Device, Emulator	-	-
16	A3E-Targeted	Статичний аналіз	Systematic	User, System	Real Device, Emulator	-	Troyd
17	Monkey	Динамічний аналіз	Random-based	User, System	Real Device, Emulator	Log	-

ДОДАТОК Б

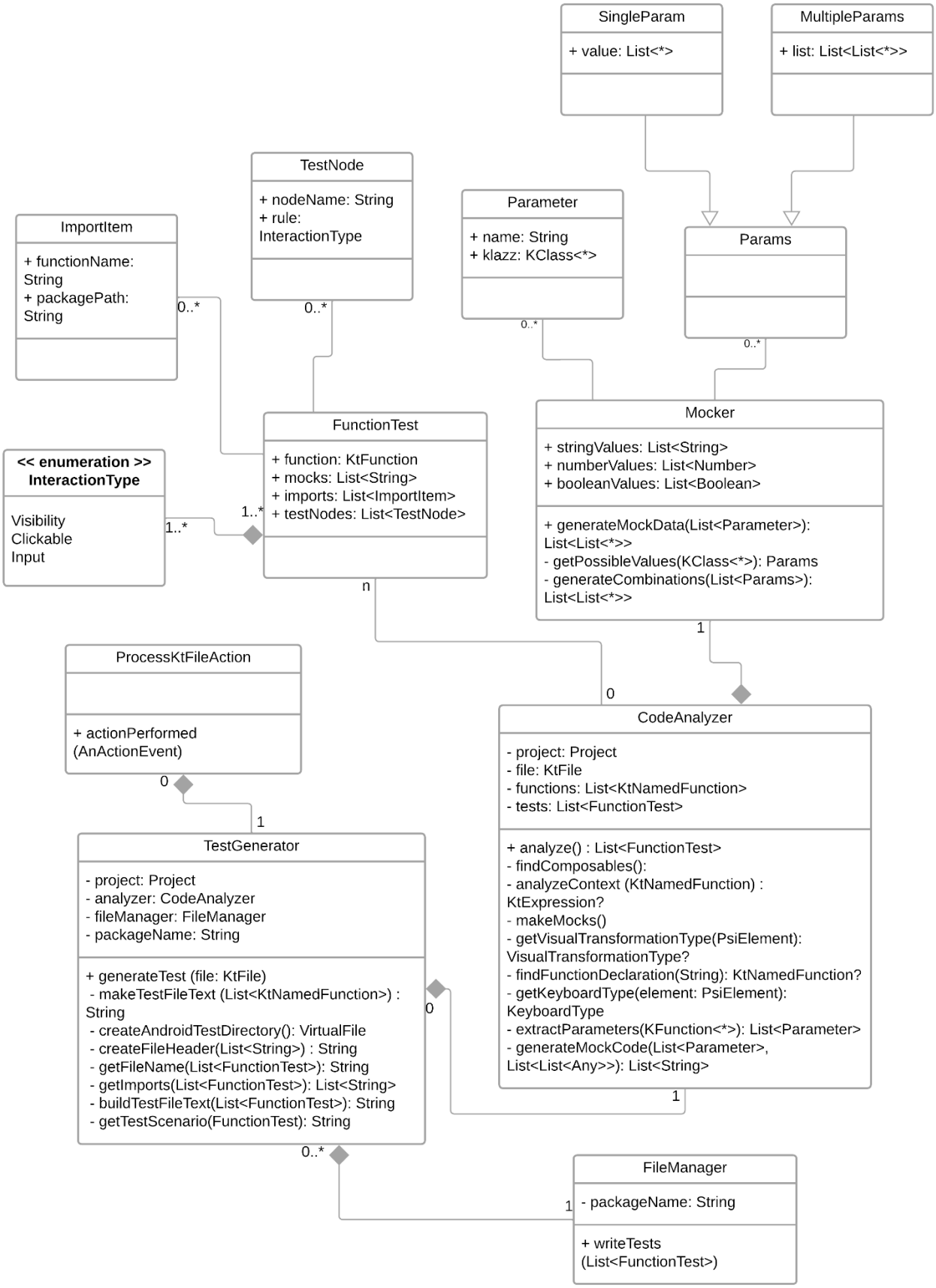

Рисунок Б.1 – Діаграма класів

ДОДАТОК В

```
package com.ordinarythinker.jcat.generator

import com.intellij.openapi.project.Project
import com.intellij.psi.*
import com.intellij.psi.search.GlobalSearchScope
import com.intellij.psi.util.PsiTreeUtil
import com.ordinarythinker.jcat.enums.InteractionType
import com.ordinarythinker.jcat.enums.KeyboardType
import com.ordinarythinker.jcat.enums.ParameterType
import com.ordinarythinker.jcat.enums.VisualTransformationType
import com.ordinarythinker.jcat.models.FunctionTest
import com.ordinarythinker.jcat.models.Parameter
import com.ordinarythinker.jcat.models.TestNode
import com.ordinarythinker.jcat.settings.Settings
import com.ordinarythinker.jcat.utils.*
import com.ordinarythinker.jcat.utils.Cons.clickables
import com.ordinarythinker.jcat.utils.Cons.textFields
import com.ordinarythinker.jcat.utils.Cons.visibles
import org.jetbrains.kotlin.descriptors.CallableDescriptor
import org.jetbrains.kotlin.descriptors.ClassDescriptor
import org.jetbrains.kotlin.descriptors.FunctionDescriptor
import org.jetbrains.kotlin.idea.caches.resolve.analyze
import org.jetbrains.kotlin.idea.stubindex.KotlinFullClassNameIndex
import org.jetbrains.kotlin.psi.*
import org.jetbrains.kotlin.resolve.BindingContext
import org.jetbrains.kotlin.resolve.calls.callUtil.getResolvedCall
import org.jetbrains.kotlin.resolve.calls.model.ResolvedCall
import org.jetbrains.kotlin.resolve.descriptorUtil.fqNameSafe
import org.jetbrains.kotlin.resolve.lazy.BodyResolveMode
import org.jetbrains.kotlin.resolve.source.getPsi
import org.jetbrains.kotlin.types.KotlinType
import org.jetbrains.kotlin.utils.IDEAPluginsCompatibilityAPI

class CodeAnalyzer(
    private val project: Project,
    private val file: KtFile,
) {
    private val mocker = Mocker(project)
    private val settings: Settings = Settings.init(project)

    private val functions = mutableListOf<KtNamedFunction>()
    private val tests = mutableListOf<FunctionTest>()

    fun analyze(): List<FunctionTest> {
        findComposableDeclarations()

        functions.forEach { function ->
            val parameters = extractParameters(function)
            val mocks = mocker.generateMockData(parameters)
            val imports = mutableListOf<String>()
            imports.addAll(mocker.imports)
            val fqName = function.fqName?.asString()
            fqName?.let { imports.add(it) }

            val subFunctions = findComposableFunctionsInBody(function)
            val interactionsForScreen = mutableListOf<List<TestNode>>()
            subFunctions.forEach { uiComponent ->
                val result = defineScenarios(uiComponent)
                interactionsForScreen.addAll(result)
            }
```

```
            tests.add(
                FunctionTest(
                    function = function,
                    mocks = mocks,
                    imports = imports,
                    scenarios = interactionsForScreen
                )
            )
        }
        return tests
    }

    private fun findComposableDeclarations() {
        file.acceptChildren(object : PsiElementVisitor() {
            override fun visitElement(element: PsiElement) {
                if (element is KtNamedFunction) {
                    val annotation = element.annotationEntries.find {
it.isComposableAnnotation() }
                    if (annotation != null) {
                        functions.add(element)
                    }
                }
                super.visitElement(element)
            }
        })
    }

    private fun findComposableFunctionsInBody(function: KtNamedFunction):
List<KtCallExpression> {
        val callExpressions =
PsiTreeUtil.findChildrenOfType(function.bodyBlockExpression,
KtCallExpression::class.java)
        return callExpressions.filter { func ->
            val calleeExpression = func.calleeExpression
            if (calleeExpression is KtNameReferenceExpression) {
                val functionName = calleeExpression.getReferencedName()
                println(functionName)
            }

            isComposableFunction(func)
        }
    }

    @OptIn(IDEAPluginsCompatibilityAPI::class)
    private fun isComposableFunction(callExpression: KtCallExpression): Boolean
{
        return try {
            val bindingContext = callExpression.analyze(BodyResolveMode.FULL)
            val referenceExpression = callExpression.calleeExpression as?
KtReferenceExpression
            val resolvedCall: ResolvedCall<out CallableDescriptor>? =
referenceExpression.getResolvedCall(bindingContext)
            val resultingDescriptor = resolvedCall?.resultingDescriptor
            if (resultingDescriptor is FunctionDescriptor) {
                val annotations = resultingDescriptor.annotations
                annotations.any {
                    it.fqName?.asString()?.contains("Composable")
                        ?: false
                }
            } else false
        } catch (e: Exception) {
            false
        }
    }
```

```
    @OptIn(IDEAPluginsCompatibilityAPI::class)
    fun findFunctionDeclaration(callExpression: KtCallExpression):
KtNamedFunction? {
        val bindingContext = callExpression.analyze(BodyResolveMode.FULL)
        val referenceExpression = callExpression.calleeExpression as?
KtReferenceExpression ?: return null
        val resolvedCall: ResolvedCall<out CallableDescriptor>? =
referenceExpression.getResolvedCall(bindingContext)
        val resultingDescriptor = resolvedCall?.resultingDescriptor ?: return
null
        if (resultingDescriptor is FunctionDescriptor) {
            val declaration = resultingDescriptor.source.getPsi()
            if (declaration is KtNamedFunction) {
                return declaration
            }
        }
        return null
    }

    private fun defineScenarios(ktCallExpression: KtCallExpression):
List<List<TestNode>> {
        val interactions = defineInteractionType(ktCallExpression)
        return getAllCombinations(interactions)
    }

    private fun defineInteractionType(ktCallExpression: KtCallExpression):
List<List<TestNode>> {
        val calleeExpression = ktCallExpression.calleeExpression
        val name = if (calleeExpression is KtNameReferenceExpression) {
            val functionName = calleeExpression.getReferencedName()
            println(functionName)
            functionName
        } else null

        val testTag = retrieveTestTagFromComposable(ktCallExpression)
        val nodes = mutableListOf<List<TestNode>>()

        val tryRecursively = {
            val funDeclaration = findFunctionDeclaration(ktCallExpression)
            funDeclaration?.let { declaration ->
                val calls = findComposableFunctionsInBody(declaration)
                val interactions = mutableListOf<List<TestNode>>()
                calls.forEach { call ->
                    val interaction = defineInteractionType(call)
                    interactions.addAll(interaction)
                }
            }
        }

        if (name != null) {
            if (testTag != null) {
                if (!settings.excludeTags.contains(testTag)) {
                    when (name) {
                        in textFields -> {
                            val keyboardType = getKeyboardType(ktCallExpression)
                            val testNoInput = settings.forNode.firstOrNull {
it.nodeTag == testTag }?.rules?.useEmptyStrings
                                ?: settings.globalRules.useEmptyStrings

                            val inputs = mutableListOf<TestNode>()
                            if (testNoInput) {
                                inputs.add(
                                    TestNode(
                                        testTag = testTag,
                                        rule = InteractionType.Input.NoInput
```

```
                                    )
                                )
                            }

                            when (keyboardType) {
                                KeyboardType.Number, KeyboardType.Phone,
KeyboardType.Decimal, KeyboardType.NumberPassword -> {
                                    inputs.add(
                                        TestNode(
                                            testTag = testTag,
                                            rule =
InteractionType.Input.NumberInput()
                                        )
                                    )
                                }
                                KeyboardType.Text, KeyboardType.Password -> {
                                    inputs.add(
                                        TestNode(
                                            testTag = testTag,
                                            rule =
InteractionType.Input.RandomStringInput()
                                        )
                                    )
                                }
                                KeyboardType.Email -> {
                                    inputs.add(
                                        TestNode(
                                            testTag = testTag,
                                            rule =
InteractionType.Input.RandomStringInput()
                                        )
                                    )
                                    inputs.add(
                                        TestNode(
                                            testTag = testTag,
                                            rule =
InteractionType.Input.ValidEmailInput()
                                        )
                                    )
                                }
                            }
                            nodes.add(inputs)
                        }

                        in clickables -> {
                            val testNoClicks = settings.forNode.firstOrNull {
it.nodeTag == testTag }?.rules?.applyClickIgnore
                                ?: settings.globalRules.applyClickIgnore

                            val clicks = mutableListOf<TestNode>()
                            if (testNoClicks) {
                                clicks.add(
                                    TestNode(
                                        testTag = testTag,
                                        rule = InteractionType.Clickable.NoClick
                                    )
                                )
                            }
                            clicks.add(
                                TestNode(
                                    testTag = testTag,
                                    rule =
InteractionType.Clickable.PerformClick
                                )
                            )
```

```
                            nodes.add(clicks)
                        }

                        in visibles -> {
                            nodes.add(
                                listOf(
                                    TestNode(
                                        testTag = testTag,
                                        rule = InteractionType.Visibility
                                    )
                                )
                            )
                        }

                        else -> {
                            tryRecursively.invoke()
                        }
                    }
                }
            } else {
                tryRecursively.invoke()
            }
        }

        return nodes
    }

    private fun retrieveTestTagFromComposable(callExpression: KtCallExpression):
String? {
        val valueArguments = callExpression.valueArguments
        for (arg in valueArguments) {
            val argExpression = arg.getArgumentExpression()
            if (argExpression != null && argExpression is
KtDotQualifiedExpression) {
                val receiverExpression = argExpression.receiverExpression
                val selectorExpression = argExpression.selectorExpression
                if (receiverExpression.text == "Modifier" && selectorExpression
is KtCallExpression) {
                    val testTagCallExpression = selectorExpression
                    if (testTagCallExpression.calleeExpression?.text ==
"testTag") {
                        val testTagValueArgument =
testTagCallExpression.valueArguments.firstOrNull()
                        val testTagValue =
testTagValueArgument?.getArgumentExpression()?.text
                        return testTagValue
                    }
                }
            }
        }
        return null
    }

    private fun getVisualTransformationType(element: PsiElement):
VisualTransformationType? {
        when (element) {
            is KtNamedFunction -> {
                // Check function declaration parameters
                val visualTransformationParameter = element.valueParameters.find
{
                    it.name == "visualTransformation"
                }
                visualTransformationParameter?.defaultValue?.text?.let {
defaultValue ->
                    if (defaultValue.contains("PasswordVisualTransformation")) {
```

```
                        return VisualTransformationType.Password
                    }
                }
            }
            is KtCallExpression -> {
                // Check function call arguments
                val functionReference = element.reference
                val resolvedFunction = functionReference?.resolve() as?
KtNamedFunction
                    ?: return null // Couldn't resolve function reference

                val visualTransformationParameter =
resolvedFunction.valueParameters.find {
                    it.name == "visualTransformation"
                }
                visualTransformationParameter?.defaultValue?.text?.let {
defaultValue ->
                    if (defaultValue.contains("PasswordVisualTransformation")) {
                        return VisualTransformationType.Password
                    }
                }
            }
        }
        return null
    }

    private fun getKeyboardType(callExpression: KtCallExpression): KeyboardType
{
        val valueArguments = callExpression.valueArguments
        for (arg in valueArguments) {
            val argExpression = arg.getArgumentExpression()
            if (argExpression != null && argExpression is
KtDotQualifiedExpression) {
                val receiverExpression = argExpression.receiverExpression
                val selectorExpression = argExpression.selectorExpression

                if (receiverExpression is KtDotQualifiedExpression &&
selectorExpression is KtCallExpression) {
                    val keyboardTypeArg =
selectorExpression.valueArguments.firstOrNull {
                        val expr = it.getArgumentExpression()

                        expr is KtDotQualifiedExpression &&
expr.receiverExpression.text == "KeyboardType"
                    }

                    keyboardTypeArg?.let { typeArg ->
                        val valueExpression = typeArg.getArgumentExpression()

                        if (valueExpression != null && valueExpression is
KtDotQualifiedExpression) {
                            val selector =
valueExpression.selectorExpression?.text
                            return try {
                                selector?.let { s ->
                                    KeyboardType.valueOf(s)
                                } ?: KeyboardType.Text
                            } catch (e: Exception) {
                                KeyboardType.Text
                            }
                        }
                    }
                }
            }
        }
```

```
        return KeyboardType.Text
    }

    fun extractParameters(function: KtNamedFunction): List<Parameter> {
        val parameters = mutableListOf<Parameter>()

        function.valueParameters.forEach { parameter ->
            val paramName = parameter.name ?: return@forEach
            val paramTypeRef = parameter.typeReference ?: return@forEach
            val bindingContext = paramTypeRef.analyze(BodyResolveMode.PARTIAL)
            val kotlinType = bindingContext[BindingContext.TYPE, paramTypeRef]
?: return@forEach
            val paramClass = findClassByName(kotlinType)

            if (paramClass != null) {
                parameters.add(Parameter(name = paramName, type = paramClass))
            }
        }

        return parameters
    }

    private fun findClassByName(kotlinType: KotlinType): ParameterType? {
        val classDescriptor = kotlinType.constructor.declarationDescriptor as?
ClassDescriptor
        val fqName = classDescriptor?.fqNameSafe

        return if (fqName != null) {

            if (fqName.asString().startsWith("kotlin.Function")) {
                return ParameterType.Function
            }

            val project = file.project
            val psiClass = KotlinFullClassNameIndex.getInstance()
                .get(fqName.asString(), project,
GlobalSearchScope.allScope(project))
                .firstOrNull() as? KtClassOrObject
            if (psiClass != null) {
                ParameterType.Type(psiClass as KtClass)
            } else null
        } else null
    }
}
```

UNICHECK
by Turnitin

Ім'я користувача:
Лісовиченко Олег Іванович

ID перевірки:
1016286000

Дата перевірки:
27.05.2024 08:18:01 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
27.05.2024 08:26:22 EEST

ID користувача:
76913

Назва документа: **ІП-21мн_Дубовик_ПЗ**

Кількість сторінок: **80** Кількість слів: **15234** Кількість символів: **119816** Розмір файлу: **1.40 MB** ID файлу: **1016079886**

2.18%
Схожість

Найбільша схожість: **0.41%** з Інтернет-джерелом (https://ela.kpi.ua/bitstream/123456789/39709/1/Dukhin_magistr.pdf)

1.57% Джерела з Інтернету	239	Сторінка 82
1.27% Джерела з Бібліотеки	259	Сторінка 83

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	5