Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Сябро Івана Володимировича*
(ПІБ)

академічної групи *121М-19-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

на тему: *Розробка програмного забезпечення з метою дослідження ефективності фреймворка підміни даних для тестування Flutter додатків.*

_____________________ *Сябро І.В.*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	Проф. Алексєєв М.О.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
________ І.М. Удовик
(підпис) (прізвище, ініціали)
« » ________ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-19-1* *Сябро Івану Володимировичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка програмного забезпечення з метою дослідження ефективності фреймворка підміни даних для тестування Flutter додатків.*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процеси автоматизованого тестування мобільних додатків.

Предмет досліджень – методи модульного тестування Flutter-додатків з використанням спеціального фреймворка підміни даних.

Мета роботи – підвищення швидкості тестування Flutter-додатків за допомогою використання фреймворка підміни даних.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна отриманих результатів кваліфікаційної роботи визначається тим, що удосконалені існуючі методи модульного тестування Flutter-додатків з використанням спеціального mock фреймворка

Практична цінність результатів полягає в тому, що розроблені та вдосконалені в роботі інструменти тестування мобільних Flutter-додатків дозволяють зменшити час на написання та зробити більш зручну підтримку модульних тестів, що в свою чергу підвищить якість. На основі аналізу поточних

рішень створено зручний набір інструментів тестування, який дозволить використовувати більш лаконічний опис логіки програмного продукту, що тестується.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2020-30.09.2020
Побудова модульних тестів для Flutter додатку з використанням фреймворка підміни даних та пошук оптимальних шляхів вирішення задачі удосконалення швидкості виконання модульних тестів	01.10.2020-31.10.2020
Створення на базі існуючих інструментів власного інструменту для підміни даних у модульному тестуванні з метою вирішення задачі збільшення швидкості виконання модульних тестів	01.11.2020-17.12.2020

Завдання видав ____________ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ____________ *Сябро І.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *18.12.2020*

РЕФЕРАТ

Пояснювальна записка: 106 с., 44 рис., 3 дод., 60 джерел.

Об'єкт дослідження: процеси автоматизованого тестування мобільних додатків.

Предмет дослідження: методи модульного тестування Flutter-додатків з використанням спеціального фреймворка підміни даних.

Мета кваліфікаційної роботи: підвищення швидкості тестування Flutter-додатків за допомогою використання фреймворка підміни даних.

Методи дослідження. Для виконання поставлених завдань були використані методи модульного тестування програмного забезпечення, а також виконано аналіз і наукове узагальнення літературних джерел по результатам досліджень.

Наукова новизна отриманих результатів кваліфікаційної роботи визначається тим, що удосконалені існуючі методи модульного тестування Flutter-додатків з використанням спеціального фреймворка підміни даних.

Практичне значення роботи. Розроблені та вдосконалені в ході кваліфікаційної роботи інструменти тестування мобільних Flutter-додатків дозволяють зменшити час на написання та зробити більш зручну підтримку модульних тестів, що в свою чергу підвищить якість та швидкість виконання.

У розділі «Економіка» проведено розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПО і тривалості його розробки.

Список ключових слів: мобільні додатки, модульне тестування, фреймворк підміни даних, Flutter, Dart, mockito.

ABSTRACT

Explanatory note: 106 p., 44 fig., 3 applications, 60 sources.

Object of research: Processes of automated mobile applications testing.

Subject of research: Unit testing methods of Flutter applications with mock framework usage.

Purpose of the qualification work: Increasing the speed of Flutter applications testing by using the mock framework.

Research methods. To perform the tasks unit testing methods were used, also the analysis and scientific generalization of literature sources on the original research were performed.

Originality of research is determined by the fact that the existing methods of unit testing of Flutter applications using a special mock framework have been improved.

Practical value of the results consists in testing tools for mobile Flutter applications which were developed and improved during the qualification work and allow to reduce the time for writing and making more convenient maintenance of unit tests, which in turn increases their quality and speed of execution.

In the Economics section the complexity and costs of software development were calculated, also the duration of the actual development was calculated.

Keywords: mobile applications, unit testing, mock framework, Flutter, Dart, mockito.

ЗМІСТ

ВСТУП 10
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ 13
1.1. Проблеми кросплатформної розробки 13
1.2. Переваги Fluter 15
1.2.1 Відсутність моста JavaScript 15
1.2.2 Тривалість компіляції 16
1.2.3 Обмін кодом 16
1.2.4 Підтримка програмного продукту 16
1.3. Як Flutter працює 17
1.3.1 Віджети у Flutter 19
1.3.2 Компонування віджетів 20
1.3.3 Віджети зі станом та без 21
1.4. Відрисовка у Flutter 24
1.4.1 Розміщення дерева віджетів 26
1.4.2 Стуктурування віджетів 27
1.4.3 Фактичне відображення віджетів на екрані 28
1.5. Висновки до першого розділу 29
РОЗДІЛ 2. ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ 30
2.1. Продуктивність і якість у програмному забезпеченні 31
2.2. Мета тестування 32
2.3. Етапи прийняття тестування 33
2.3.1 Етап неприйняття тестування 34
2.3.2 Етап доведення відсутності проблем 34
2.3.3 Етап доведення наявності проблем 35
2.3.4 Етап зниження ризику 35
2.3.5 Етап ментального стану 36
2.4. Проектування тестів 37

2.5. Особливості тестування мобільних додатків 38
2.5.1 Очікування користувачів від мобільних додатків 38
2.5.2 Мобільність і мережі даних 40
2.5.3 Мобільні пристрої 41
2.5.4 Випуск нових версій мобільних продуктів 43
2.5.5 Тестування мобільного програмного забезпечення 45
2.6. Процес тестування мобільних додатків 45
2.6.1 Мануальне та автоматизоване тестування 46
2.6.2 Звичайне тестування 47
2.7. Мок об’єкти та крихкість тестів 49
2.7.1 Типи двійників у тестах 50
2.7.2 Мок як інструмент та як тестовий двійник 52
2.7.3 Перевірка виклику методів 53
2.7.4 Спільне використання моків та заглушок 55
2.7.5 Відношення моків та заглушок до запитів і команд 56
2.8. Спостережувальна поведінка та деталі реалізації 57
2.8.1 Публічне API та спостережувальна поведінка 58
2.8.2 Витік деталей реалізації 60
2.8.3 Внутрішні та зовнішні взаємодії систем 62
2.9. Висновки до другого розділу 64
РОЗДІЛ 3. ПІДВИЩЕННЯ ШВИДКОСТІ ТЕСТУВАННЯ FLUTTER-ДОДАТКІВ ЗА ДОПОМОГОЮ ФРЕЙМВОРКА ПІДМІНИ ДАНИХ 67
3.1. Модульне тестування 67
3.1.1 Підміна залежностей за допомогою mockito 70
3.1.2 Верифікація викликів та робота з локальним сховищем 73
3.2. Використання власного інструменту для роботи з мережею 76
3.3. Висновки до третього розділу 79
РОЗДІЛ 4. ЕКОНОМІКА 81
4.1. Визначення трудомісткості розробки програмного забезпечення 81
4.2. Витрати на створення програмного забезпечення 84

4.3. Маркетингові дослідження ... 86
4.4. Економічна ефективність ... 89
ВИСНОВКИ ... 93
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 94
Додаток А. ЛІСТИНГ ПРОГРАМИ ... 99
Додаток Б. ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ ... 105
Додаток В. ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ ... 106

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

SDK – Software Development Kit;

API – Application Programming Interface;

JSON – JavaScript Object Notation;

HTTP – HyperText Transfer Protocol;

QA – Quality Assurance;

БД – База даних;

ПЗ – Програмне забезпечення.

ВСТУП

Актуальність роботи. Останні декілька років у мобільній розробці набуває популярності крос платформне рішення Flutter. Flutter – це набір інструментів користувальницького інтерфейсу для створення гарних, нативно компільованих застосунків для мобільних, веб та десктопних платформ з єдиною кодовою базою та використанням мови програмування Dart.

Головні переваги Flutter:

– можливість оновлювати візуальне відображення застосунку на екрані за лічені мілісекунди за допомогою гарячого перезавантаження зі збереженням стану (Stateful Hot Reload);
– завдяки багатому набору цілком настроюваних віджетів створення нативного інтерфейсу займає декілька хвилин;
– швидке впровадження нового функціоналу з фокусом на користувальницький досвід;
– багаторівнева архітектура дозволяє робити гнучкі настройки, що в свою чергу призводить до надзвичайно швидкої отрисовки додатку та виразних і гнучких елементів дизайну;
– Flutter-віджети включають в себе усі критичні відмінності між платформами, такі як навігація, прокрутка сторінок, іконки та шрифти;
– використовує сучасну мову розробки Dart;
– код на Flutter компілюється у нативний машинний код з використанням нативних Dart компіляторів, що надає високу продуктивність, таку ж саму, як у нативних мобільних додатках.

Під час розробки програмних систем та додатків необхідно переконатися, що створюваний програмний продукт працює належним чином. Цей процес називається контролем якості (Quality Assurance) або просто тестуванням програмного забезпечення.

Одним із видів тестування є автоматизоване модульне тестування або юніт тестування (unit testing). Модульне тестування представляє собою метод

тестування програмного забезпечення, який включає в себе тестування індивідуальних елементів вихідного коду – наборів одного або кількох програмних модулів разом із асоційованим набором контрольних даних, використовуваних та операційних процедур, з метою перевірки їх на придатність до використання.

Перевагою такого методу тестування є надзвичайно велика швидкість та можливість автоматизованого запуску об’ємної кількості тестів без участі людини. Інколи при юніт тестуванні виникає проблема зовнішньої залежності, через яку тест неможливо виконати у тестовому середовищі. Одним із способів вирішення цієї проблеми є використання мок об’єктів (mock objects), які імітують поведінку реальних об’єктів, необхідних при роботі тесту, але недоступних через обмеження тестових систем.

Мета роботи – підвищення швидкості тестування Flutter-додатків за допомогою використання фреймворка підміни даних.

Об’єкт досліджень – процес автоматизованого тестування мобільних додатків

Предмет досліджень – методи модульного тестування Flutter-додатків з використанням спеціального фреймворка підміни даних.

Методи дослідження – для виконання поставлених завдань були використані методи модульного тестування програмного забезпечення, а також виконано аналіз і наукове узагальнення літературних джерел по результатам досліджень.

Наукова новизна отриманих результатів кваліфікаційної роботи визначається тим, що удосконалені існуючі методи модульного тестування Flutter-додатків з використанням спеціального фреймворка підміни даних.

Практична цінність результатів полягає в тому, що розроблені та вдосконалені в роботі інструменти тестування мобільних Flutter-додатків дозволяють зменшити час на написання та зробити більш зручну підтримку модульних тестів, що в свою чергу підвищить якість. На основі аналізу поточних рішень створено зручний набір інструментів тестування, який дозволить

використовувати більш лаконічний опис тестуємої логіки програмного продукту.

Особистий внесок автора:

1. Наукові результати роботи отримані автором самостійно;
2. Вибір методів досліджень і технологій реалізації;
3. Дослідження і систематизація знань про існуючі методики тестування мобільних додатків, в тому числі із використанням фреймворків для підміни даних;
4. Розробка власного рішення, направленого на збільшення швидкості модульного тестування мобільних Flutter додатків;
5. Оцінка отриманих результатів.

Структура та обсяг роботи. Робота складається з вступу, трьох розділів і висновків. Містить 110 сторінок друкованого тексту, в тому числі 96 сторінок тексту основної частини з 44 рисунками, списку використаних джерел з 60 найменуваннями на 5 сторінках, 4 додатка на 9 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ

1.1. Проблеми кросплатформної розробки

Вибір інструментів та технологій для написання мобільного додатку.

На початку створення нового мобільного додатку перед розробником постає питання, які технології та набір інструментів обрати у певній ситуації, щоб вони максимально чітко відповідали потребам програмного продукту.

Перший вибір – це нативна розробка для платформ iOS та Android. Такий підхід надає максимальний контроль, відладочні інструменти та теоретично найбільшу продуктивність. З точки зору роботодавця, або компанії, ще скоріш за все означає, що код повинен бути написаний двічі під кожну платформу. Ймовірно роботодавець повинен наймати дві різні команди розробників, серед яких будуть різні люди з різними рівнями навичок. Це, в свою чергу, потребує додаткової комунікації, оскільки навряд чи розробники із різних команд зможуть з легкістю допомагати один одному.

Другий вибір – це кросплатформені, засновані на ДжаваСкріпт (JavaScript) набори інструментів, такі як Веб Віджети (Web Views) та Реакт Нейтів (ReactNative). Це доволі непоганий вибір, який дозволяє позбутися проблем, що виникають в ході нативної розробки. Будь який розробник з команди, який має сучасні навички JavaScript, наприклад спеціаліст по FrontEnd, може допомогти вирішити проблему з програмним кодом. Це одна із головних причин, чому великі компанії, такі як Microsoft і Facebook використовують React Native у своїх продуктах [1].

Але і у такого підхіду є свої недоліки. Найбільший із них – це міст Java Script. Перші кросплатформні мобільні додатки були усього напросто веб віджетами, що запускалися на движку браузерної отрисовки WebKit. Вони буквально були вкладеними веб сторінками, поміщеніми у мобільний додаток.

Проблема полягала у тому, що взаємодія з елементами веб сторінки напряму була дуже повільною та користувальницький досвід залишав бажати кращого.

Деякі платформи вирішили цю проблему, побудувавши так званий Java Script міст, що дозволяє напряму спілкуватися із нативним кодом. Це набагато продуктивніше, ніж веб віджети, оскільки це вирішує проблему комунікації з елементами веб сторінок, але це все ще не ідеальний підхід. Кожного разу, коли мобільному застосунку потрібно напряму комунікувати із движком отрисовки, він повинен бути скомплікований у нативний код, щоб перейти міст JavaScript. Під час однієї простої взаємодії, міст повинен бути перетнутий двічі. Перший раз під час переходу із платформи до застосунку, і другий раз із застосунки до платформи, як зображено на рисунку 1.1.

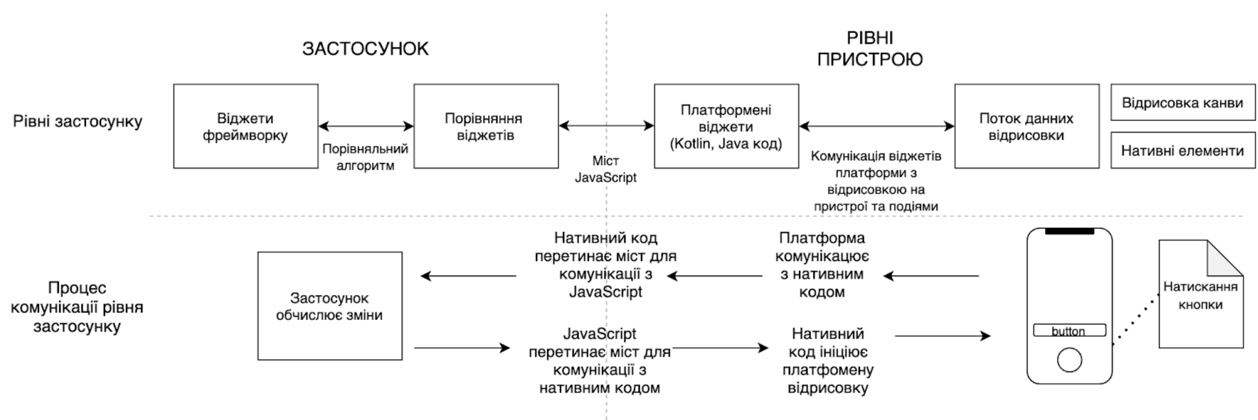

Рис. 1.1. Процес комунікації фреймворків JavaScript із мобільними застосунками

Flutter компілюється напряму у ARM код під час створення сборки для кінцевого користувача. В свою чергу, ARM – архітектура та сімейство мобільних процесорів, що використовуються у мобільних девайсах. Flutter також поставляється зі своїм власним движком відрисовки. Це означає, що Flutter працює нативно та не потребує необхідності перетинати будь який міст. Він напряму комунікує з нативними елементами та контролює кожен піксель на екрані пристрою нативно. У порівнянні з JavaScript мостом, комунікація Flutter-застосунку має набагато менше кроків (рис. 1.2).

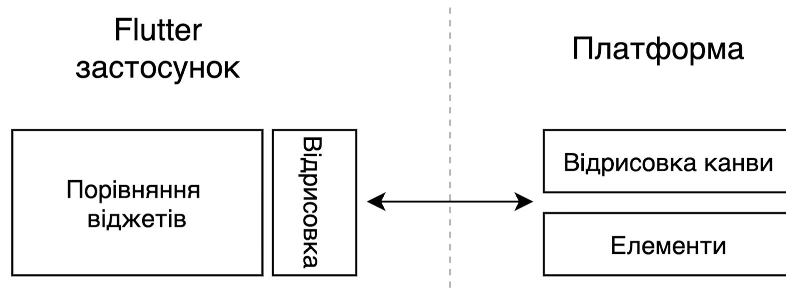

Рис. 1.2. Flutter платфома, у контексті моста JavaScript

Міст JavaScript, це звичайно, велике досягнення сучасного програмування, але він має дві великі проблеми. Перша заключається у складності відладки. Якщо під час виконання програми виникає помилка, вона повинна бути відстежена через міст JavaScript та знайдена у JavaScript коді. Друга проблема – це продуктивність. Міст JavaScript доволі витратний: кожного разу, коли користувач взаємодіє з інтерфейсом, наприклад скролить сторінку, ця подія повинна бути відправлена через міст до самого JavaScript застосунку. У результаті користувач отримує підторможування інтерфейсу.

Більшість вищеперелічених кросплатформних проблем були вирішені у Flutter.

1.2. Переваги Flutter

1.2.1. Відсутність моста JavaScript

Міст JavaScript, що використовується у більшості крос платформених рішень, є значним вузьким горлечком у продуктивності застосунків. Навігація не є плавною, застосунок не завжди працює швидко і його важко налагоджувати.

Flutter відповідає фактичному нативному коду та відрисовується за допомогою движка, що використовується у Chrome, його назва Skia, так що потреба у конвертації Dart коду, який використовує флатер, відпадає. Це означає, що застосунок не буде мати жодних проблем з продуктивністю при роботі на пристроях користувачів.

1.2.2. Тривалість компіляції

У мобільній розробці зазвичай тривалість компіляції доволі велика. Вона може займати від декількох до десяти хвилин при розробці Android додатків та від п'яти до п'ятнадцяти хвилин при розробці iOS додатків. Тривалість напряму залежить від важкості проекту, і чим більше коду, тим більше часу потрібно, щоб його скомпілювати. Під час розробки програмного забезпечення це може значно зменшити швидкість розробки, в результаті чого продукт буде випущений пізніше запланованих строків.

У Flutter час на повну компіляцію (ту, що виконується перший раз, та займає набагато більше часу, ніж наступні), зазвичай займає менше ніж 30 секунд. Час на наступні компіляції займає менше однієї секунди, завдяки можливості гарячого перезавантаження.

1.2.3. Обмін кодом

При використанні Flutter та Dart, десктоп, веб та мобільні додатки можуть мати однаковий код, оскільки Flutter підтримує всі вищеперелічені платформи. На практиці це означає, що різні команди розробників можуть надавати допомогу один одному, значно зменшуючи час на розробку програмного продукту.

1.2.4. Підтримка програмного продукту

Під час виправлення помилок у роботі додатку, прибирання помилки на одній платформі означає, що на всіх інших платформах вона також не буде проявлятися. Це значно скорочує час на поліпшення застосунків на всіх платформах. Тільки в деяких дуже специфічних випадках може проявлятися баг (помилка, від англійського bug), на Flutter Android додатках, що не присутній в iOS додатках (та навпаки). Але зазвичай ці проблеми не є критичними, та

представляють собою незначні косметичні проблеми, такі як розмірі тесту та вирівнювання шрифтів, тому вони є тривіальними з точки зору трудозатрат на їх виправлення.

1.3. Як Flutter працює

З точки зори високого рівня абстракції, Flutter це реактивна, декларативна та складаєма із різних рівнів віджетів бібліотека, що нагадує ReactJS на веб, за винятком того, що Flutter також є повноцінним движком відрисовки. Інтерфейс користувача будується шляхом компонування широкого набору біль малих елементів, що називаються віджетами. Весь графічний інтерфейс користувача представлений віджетами, які в свою чергу всього на всього Dart класи, що описують, як вони повинні відображатися на екрані. Структури, стилі, кнопки, списки, навіть анімації – усе, що бачить користувач під час роботи застосунку є віджетами. Слід зазначити, що окрім віджетів у Flutter є також контролери, потоки даних, результати відложеного виконання задач та багато інших елементів. Це декілька відрізняється від слогану “Усе є віджетом”, який можна знайти не тільки на різних куточках інтернету, але й і на офіційному сайті Flutter https://flutter.dev/.

Припустимо, необхідно побудувати застосунок, схожий з соціальними мережами та має доволі стандартний функціонал. Він відображає список постів, які складаються із зображення, тексту під зображенням та мають кнопки “подобається”, “коментувати” та “поділитися”. Список, зображення, текст, кнопки і все інше на екрані є віджетами. На рисунку 1.3 відображено, як приблизно міг би виглядати код для таких віджетів. Інші класи, які скоріш за все будуть присутні у такому додатку це класі специфічної бізнес-логіки, яка не має відношення до Flutter.

Віджети складаються із віджетів, які в свою чергу також складаються з вкладених віджетів. Деякі віджети мають стан. Наприклад, віджет що відстежує кількість лайків під постом. Коли стан віджета змінюється, фреймворк отримує

про це повідомлення та порівнює опис нового дерева віджетів зі старим описом та змінює лише ті віджети, що потребують необхідності. У прикладі із соціальним додатком, коли користувач натискає кнопку “подобається” (кнопка у вигляді серця), йде оновлення внутрішнього стану, що примушує Flutter перемалювати усі віджети, що залежать від цього стану. В нашому випадку, це текстовий віджет кількості лайків та стан кнопки. На рисунку 1.4 схематично зображено, як віджети можуть виглядати до і після натискання віджета “подобається”, що має назву IconButton.

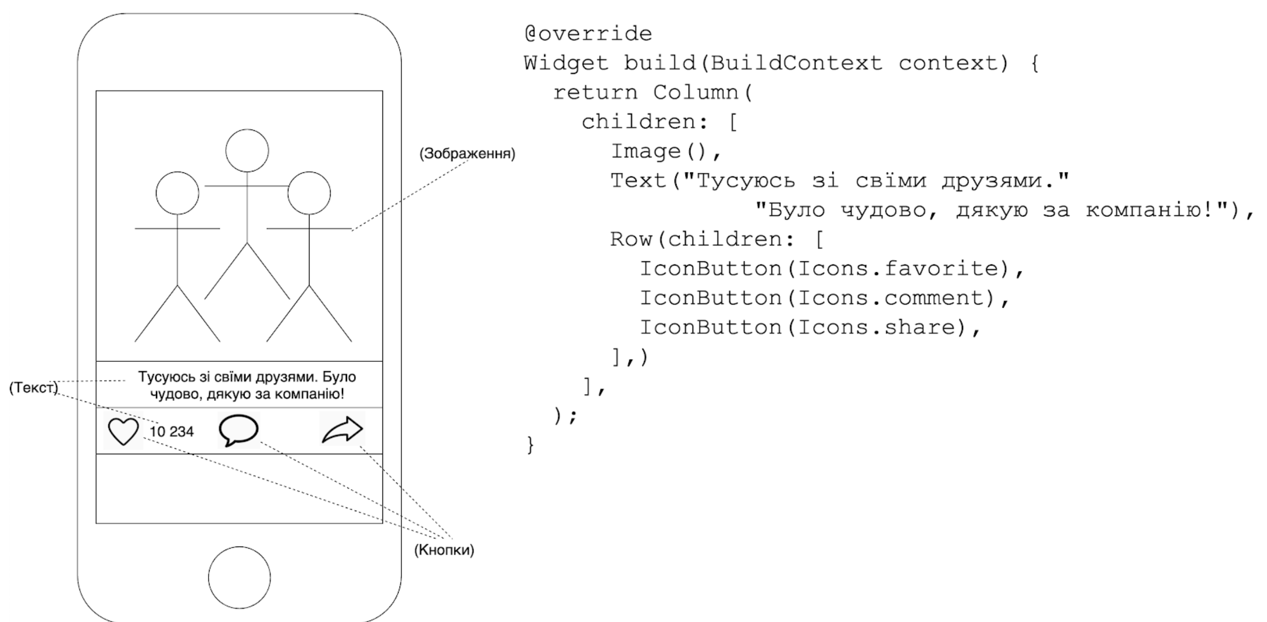

Рис. 1.3. Схематичне використання Flutter віджетів

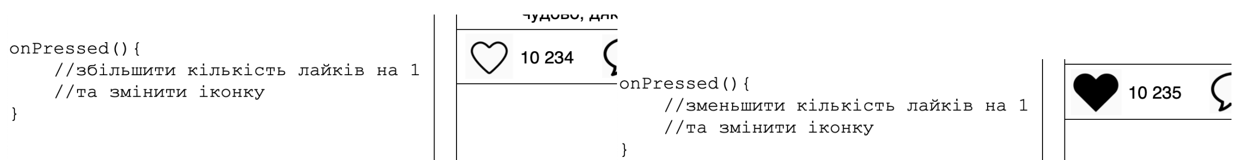

Рис. 1.4. Результат виконання методу setState, що оновлює відображувану кількість лайків та колір кнопки

Ці дві ідеї, віджети та оновлення стану, є головними для Flutter та ключовими для розробників.

1.3.1. Віджети у Flutter

Усе є віджетом – така основна ідея Flutter. Звичайно, це не означає, що у Flutter немає інших типів об’єктів. Але більшу частину часу розробник проводить у роботі з віджетами. Flutter не має спеціальних класів по типу моделей (models), або моделей уявлення (view models) для роботи із віджетами.

Віджет може визначати будь який аспект уявлення застосунку. Деякі віджети, такі як Column (колонка), визначають характеристики позиціонування. Інші менш абстрактні, та представляють собою структурні елементи, такі як текст (TextField) або зображення (Image). Кореневий елемент застосунку також є віджетом.

Повертаючись до прикладу із соціальним застосунком, на рисунку 1.5 відображено, яким приблизно може бути код для побудування віджетів розміщення. На рисунку 1.6 відображено код структурних елементів.

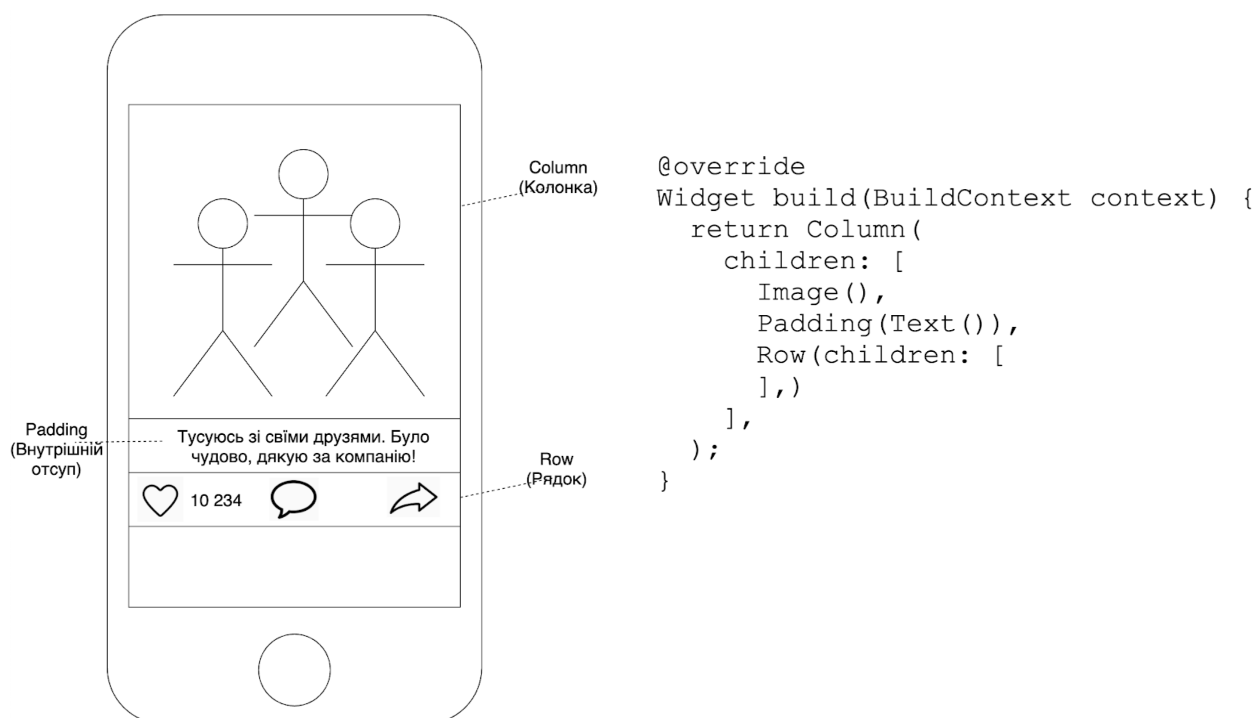

Рис. 1.5. Приблизний код для побудування віджетів розміщення

Звичайно, віджетів набагато більше, ніж відображено на рисунку, тому що вони визначають розміщення, стилі, анімації і так далі.

Найчастіше використовуваними віджетами є:

- структурні: Text, TextField, Button;
- розміщення: Row, Column, Stack;
- анімації: FadeInImage, CurvedAnimation;
- позиціонування та вирівнювання: Right, Padding;
- стилі: TextStyle, Color.

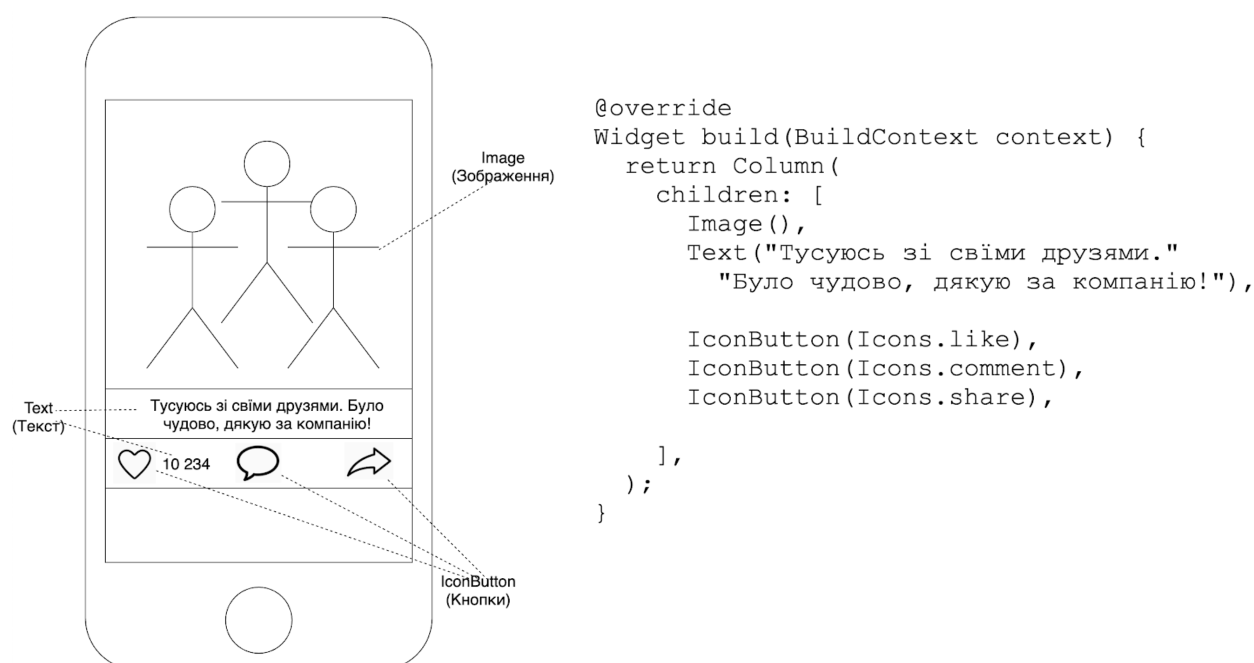

Рис. 1.6. Приблизний код для будування структурних віджетів

1.3.2. Компонування віджетів

Flutter віддає перевагу компонуванню на відміну від спадкування класів, що дозволяє з легкістю робити власні, унікальні віджети. На практиці це означає, що у Flutter для створення власного віджету не потрібно успадковувати клас цього віджету від іншого. Замість такого запису, який може доволі часто бути зустрінутим у Android та iOS, зображеного на рисунку 1.7, використовується компонування завдяки обгортанню віджету Image у інші віджети (рис. 1.8).

У віджетів є певні властивості та методи життєвого циклу. Найголовніший метод у віджетів – це метод build(). Він має бути присутнім у кожному Flutter

віджеті, тому що саме він описує його, повертаючи як результат виконання інші віджети.

```
class ShareButton extends Button
{
}
```

Рис. 1.7. Типове об'явлення класу власного віджету у Android та iOS

```
class ShareButton extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: [
        IconButton(Icons.share),
      ],
    );
  }
}
```

Рис. 1.8. Об'явлення класу власного віджету у Flutter

1.3.3. Віджети зі станом та без

Більшість віджетів підпадає під дві категорії: віджети, що мають стан (stateful) та ті, що не мають (stateless). Для розробників головна відмінність полягає у тому, що StatelessWidget використовується тоді, коли втрата інформації що зберігається у ньому, допустима. Весь стан віджета або конфігурація передається всередину та його задача полягає лише у відображенні інтерфейсу. Його життєвий цикл залежить від зовнішніх подій та він не керує коли потрібно додавати, або перестроювати віджет у дереві віджетів. Замість нього це робить фреймворк.

У прикладі із соціальним додатком, віджет ShareButton є stateless віджетом. Йому не потрібно керувати станом та він не має потреби знати про будь яку іншу частину дерева віджетів. Задача цього віджета полягає лише у тому, щоб чекати

натискання користувачем та виконувати відповідну функцію при виконанні такої події. Це не означає, що ShareButton ніколи не зможе змінити свій стан. У процесі виконання можна зробити так, щоб замість іконки відображався текст, наприклад “поділитися”. Для цього інший віджет може передати у ShareButton об’єкт типу строка (String), який буде представляти собою текст для відображення. Віджет буде оновлюватися разом із строкою тексту, переданного у нього. Таким чином віджет реагує на нову інформацію.

Прикладом StatefulWidget у соціальному застосунку є LikesCount віджет, тому що він керує невеликою кількістю даних стану, що відстежують кількість лайків під постом. У кожного StatefulWidget завжди є відповідний об’єкт State (стан). State має спеціальні методи, такі як setState, що ініціюють перерисовку на стороні Flutter. Об’єкти State мають подовжений життєвий цикл у порівнянні з віджетами. Окрім можливості ініціювати перерисовку у Flutter, вони також можуть отримувати повідомлення про необхідність її виконання, наприклад, при оновленні відповідного Stateful віджета у результаті зовнішніх подій.

Повертаючись до прикладу з соціальним застосунком, приблизно так може виглядати код методу build для stateful віджета LikesCount (рис. 1.9):

```
@override
  Widget build(BuildContext context) {
    return Padding(
      padding: EdgeInsets.all(16),
      child: Row(
        children: [
          IconButton(
              icon: Icon(hasLike
                  ? Icons.favorite_border
                  : Icons.favorite_border_sharp),
              onPressed: () {
                setState(() {
                  hasLike = !hasLike;
                  if (hasLike) {
                    likesCount++;
                  } else {
                    likesCount--;
                  }
                });
              }),
          Text("$likesCount")
        ],
      ),
    );
  }
```

Рис. 1.9. Приклад методу build для власного stateful віджету LikesCount

У методі onPressed віджету IconButton є виклик методу setState, який виконується кожного разу, коли користувач натискає кнопку “подобається”. Цей метод доступний, оскільки stateful віджети мають доступ до деяких методів із базового класу State. Метод може оновити будь яку частину віджета за проханням та ініціювати перерисовку на стороні Flutter саме тих віджетів, що залежать від зміненого стану, як зображено на рис 1.4.

Процес побудови та оновлення віджетів називається життєвим циклом. На рисунку 1.10 зображено життєвий цикл StatefulWidget.

Життєвий цикл віджету LikesCount може виглядати наступним чином. При першому відображені сторінки Flutter створює об’єкт, який в свою чергу створює об’єкт стану (State), що відповідає цьому віджету.

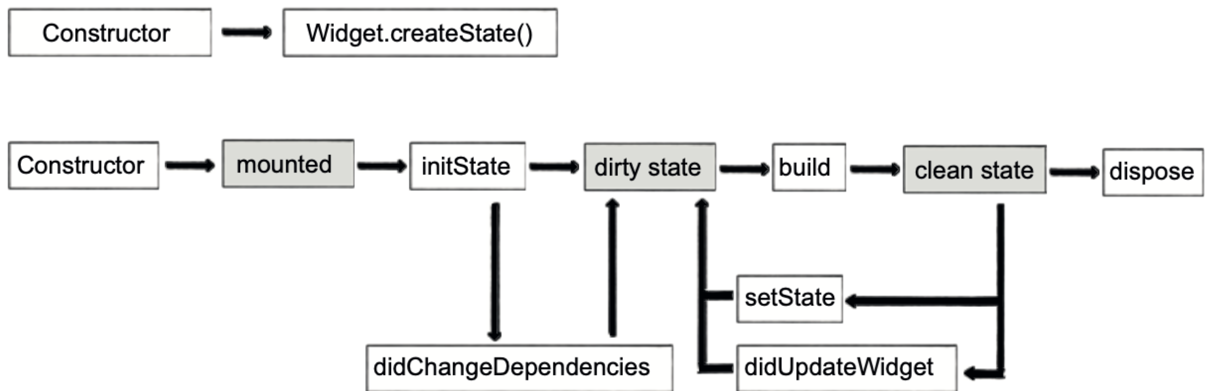

Рис. 1.10. Життєвий цикл stateful віджета

Після створення віджету фреймворк робить виклик методу initState. Одразу за ініціалізацією стану, Flutter виконує побудову віджету, викликаючи метод build, та відрисовує його. Тепер віджет відображається на екрані, та очікує одну із трьох наступних подій:

– користувач переходить до іншої частини застосунку, у такому разі об’єкт стану може бути знищено;
– віджет у дереві віджетів, що є зовнішнім по відношенню до даного, змінив конфігурацію, від якої залежить стан цього віджету. У такому разі об’єкт стану отримує виклик методу didUpdateWidget та виконує перерисовку при необхідності. Прикладом цього може бути

ситуація, коли пост було видалено, та віджет, що є вищим по відношенню до даного у ієрархії віджетів наказує попередньому змінити стан на недоступний, оскільки можливості поставити лайк посту вже немає;

– користувач натиснув на віджет, що викликало setState та оновило внутрішній стан віджета. При цьому також Flutter перебудовує та перерисовує віджет.

1.4. Відрисовка у Flutter

Найбільша перевага Flutter полягає у процесі, який він виконує мільйони часів на день. Процес, у ході якого Flutter побудовує та перебудовує застосунок. У будь-який момент часу Flutter застосунок є скомпонованим із безлічі віджетів, що формують своєобразне дерево. Рисунок 1.11 відображає спрощений приклад того, як може виглядати дерево віджетів для сторінки, що включає в себе віджет LikesCount.

У цьому дереві віджет PostItem є Stateful віджетом, і кожен із його дочірніх елементів залежить його стану. Коли стан віджета PostItem оновлюється, у піддереві віджетів з цієї точки запускається процес перерисовки [3].

Віджети у Flutter є реактивними. Вони реагують на інформацію з зовнішнього джерела, або з методу setState, і Flutter перебудовує відповідні частини інтерфейсу. С високого рівня абстракції, цей процес виглядає так:

1. Користувач натискає на кнопку;
2. Застосунок робить виклик методу setState у відповідному методу обратного зв’язку (слухачеві) кнопки – Button.onPressed;
3. Flutter знає, яку частину інтерфейсу потрібно перебудувати, оскільки стан кнопки позначений як брудний (dirty);
4. Новий віджет займає місце старого у дереві віджетів;
5. Flutter відрисовує нове дерево віджетів.

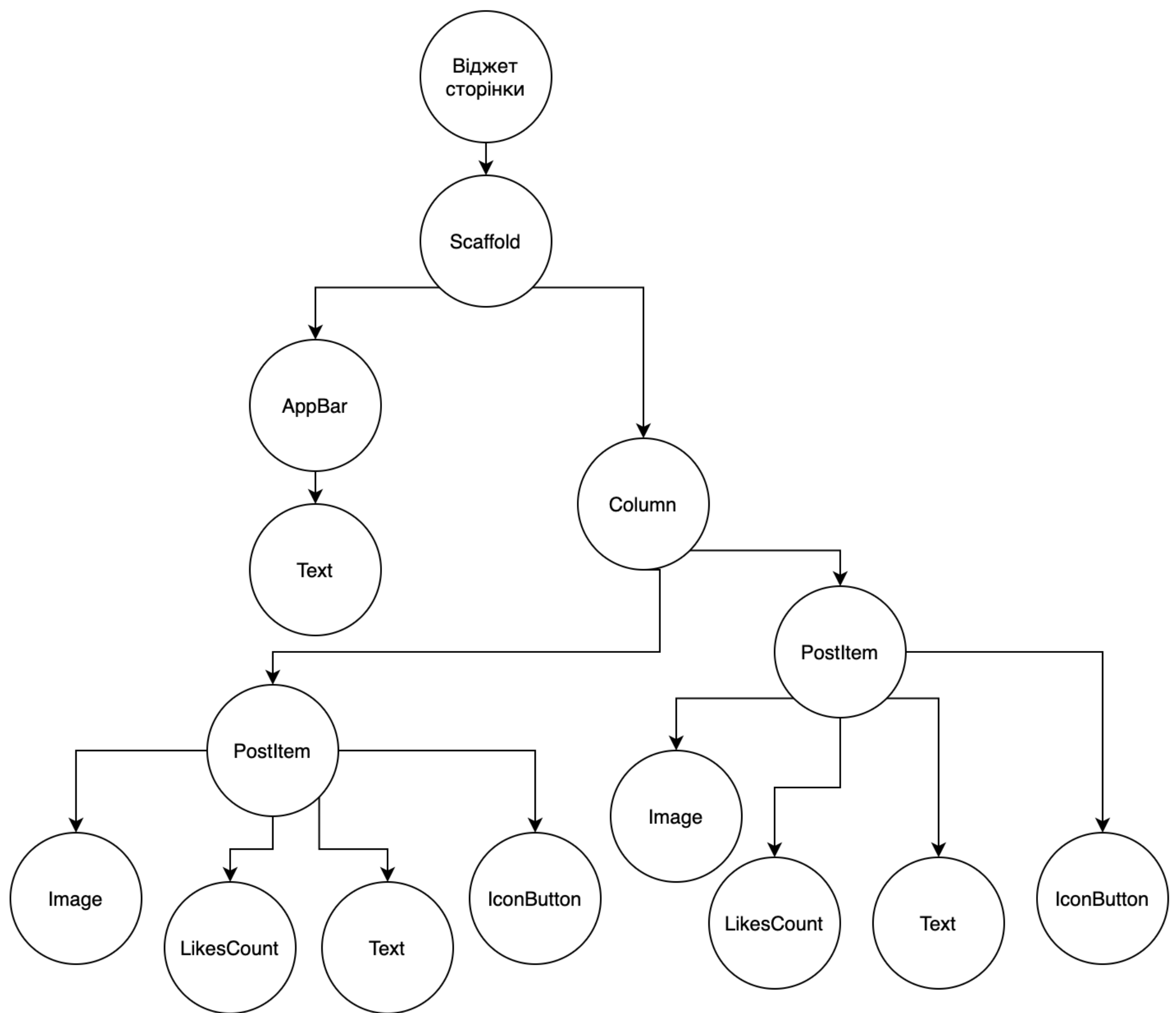

Рис. 1.11. Спрощений приклад дерева віджетів на сторінці

Після того, як Flutter отримує інформацію про нові віджети, він готовий до відрисовки. Крок відрисовки представляє собою серію інших послідовних кроків. Рисунок 1.12 відображає кроки, що виконує Flutter під час відрисоки, та акцентує увагу на кроці 3.

Flutter починає відрисовку із запуску тікерів анімації. Тікер – це спеціальний об'єкт, що контролює процес роботи анімації. Наприклад, при скролінгу списку донизу стартова позиція кожного елементу поступово змінюється до кінцевої позиції, що створює плавну анімацію. Цей процес контролюється анімаційними тікерами, які відповідають за час, коли елемент повинен рухатися. Під час анімації перерисовка відбувається на кожному кадрі, що приблизно дорівнює 60 разів на секунду.

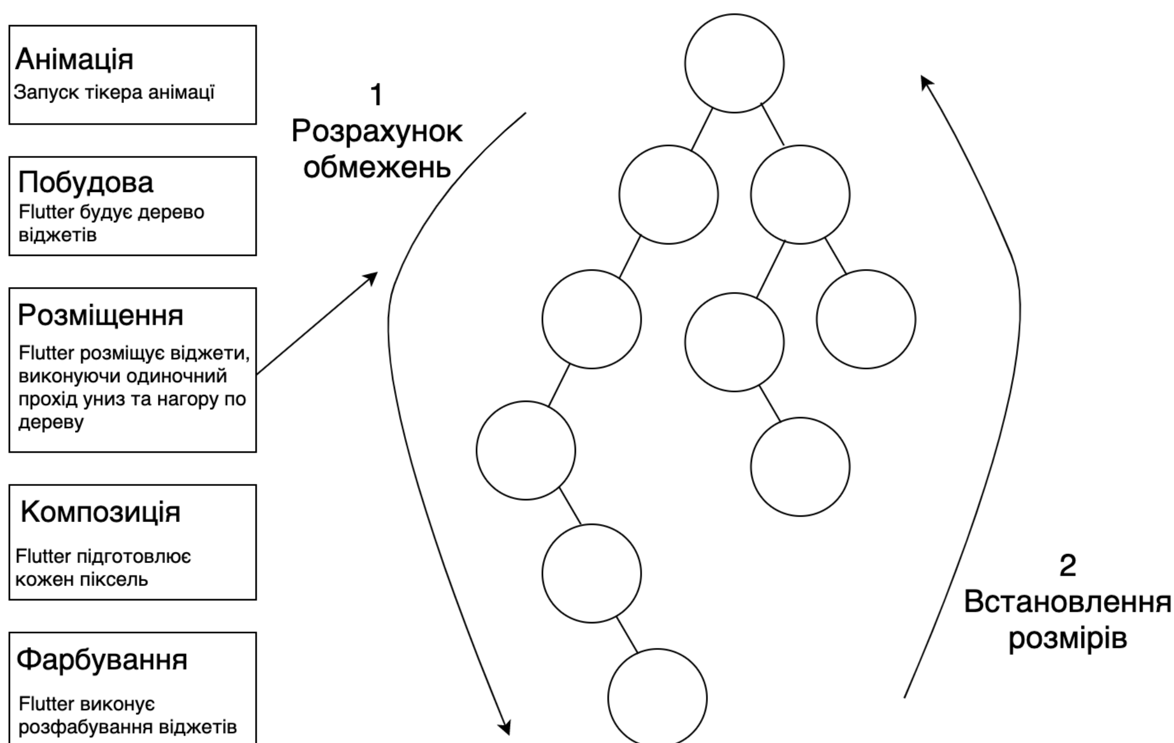

Рис. 1.12. Ключові кроки процесу відрисовки

1.4.1. Розміщення дерева віджетів

Після запуску анімацій Flutter виконує побудування усіх віджетів та конструює дерево віджетів. Від віджетами слід розуміти дані та конфігурації, що описують, як елементі повинні відображатися на екрані. Насправді, при побудуванні кнопки для дерева віджетів створення прямокутника із текстом у ньому створюється не на цьому етапі, це пізніший крок. Віджети лише обробляють конфігурацію елементів, що в кінцевому етапі будуть відображені на екрані.

Після компонування дерева віджетів Flutter починає їх розміщення. За лінійний час він один раз проходиться по дереву віджетів зверху до низу. Слід зазначити, що лінійний час означає велику швидкість. На шляху униз Flutter збирає інформацію про позицію віджетів. У Flutter обмеження на розміщення та розмір задається батьківськими елементами для дочірніх. Обмеження (constraints) стосуються позиціонування, наприклад, інформація про мінімально та максимально допустимі розміри віджетів. Обмеження необхідні для того, щоб

віджети не кофліктували один із одним при відрисовці. При зворотному шляху при проході дерева віджетів, кожен віджет знає свої обмеження, завдяки цьому він може повідомити батьківському елементу його актуальні розміри та позицію. Таким чином віджети розміщуються з урахуванням інших віджетів, один відносно одного.

У прикладі з соціальним застосунком коли користувач натискає кнопку “подобається” у віджеті LikesCount і стан віджета оновлюється з новою кількістю лайків та іконкою, Flutter проходить униз по дереву віджетів та LikesCount повідомляє кнопці із текстовому полю їх обмеження, але це ще не є дійсними розмірами. Алгоритм виконується циклічно для спадкових елементів віджета, доки не буде досягнуто листових (крайніх) елементів дерева. Після досягнення крайніх елементів алгоритмом усі віджети знають свої обмеження розмірів. На зворотному шляху алгоритму вони безпечно займають правильну кількість місця на правильній позиції. Рисунок 1.13 відображає цей процес на дереві віджетів. Слід зазначити, що завдяки лише одному проходу по дереву віджетів Flutter додатки дуже є продуктивними.

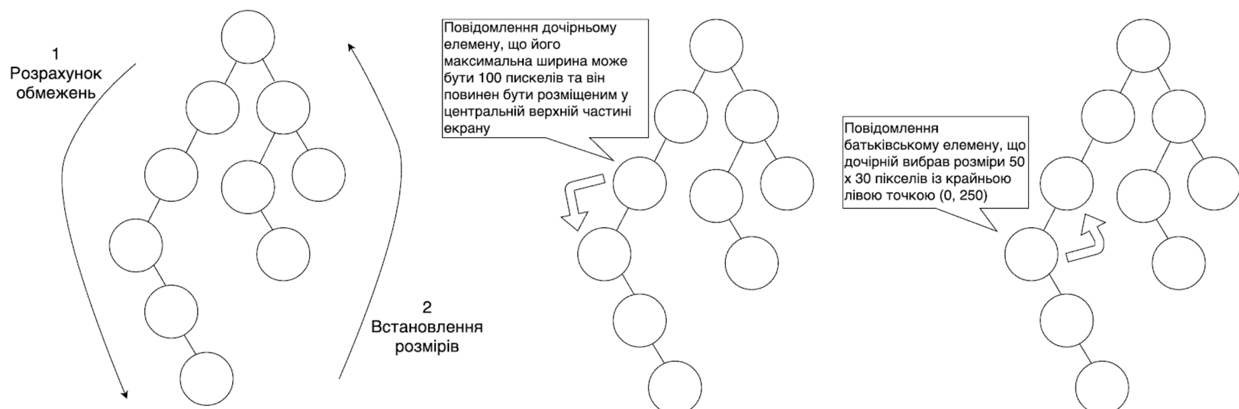

Рис. 1.13. Процес розміщення віджетів

1.4.2. Структурування віджетів

На цьому кроці кожен віджет розміщений та знає, що він не конфліктує з іншими віджетами, тому Flutter може намалювати віджети. Але віджети все ще не растризовані та не намальовані фізично на екрані пристрою. Під час

структурування Flutter назначає віджетам їх реальні координати на екрані, що відповідають фізичним пікселям. Цей крок відділений від кроку малювання по причині того, що структуровані віджети можуть бути використані повторно при необхідності. Це дуже корисно у випадках, коли на екрані є список елементів, що скролиться, як зображено на рисунку 1.14. Замість того, щоб повторно будувати новий елемент кожного разу, як крайній елемент списку з'являється або пропадає з екрану при скролі, Flutter вставляє вже раніше підготовлений елемент, який при цьому побудований та намальований.

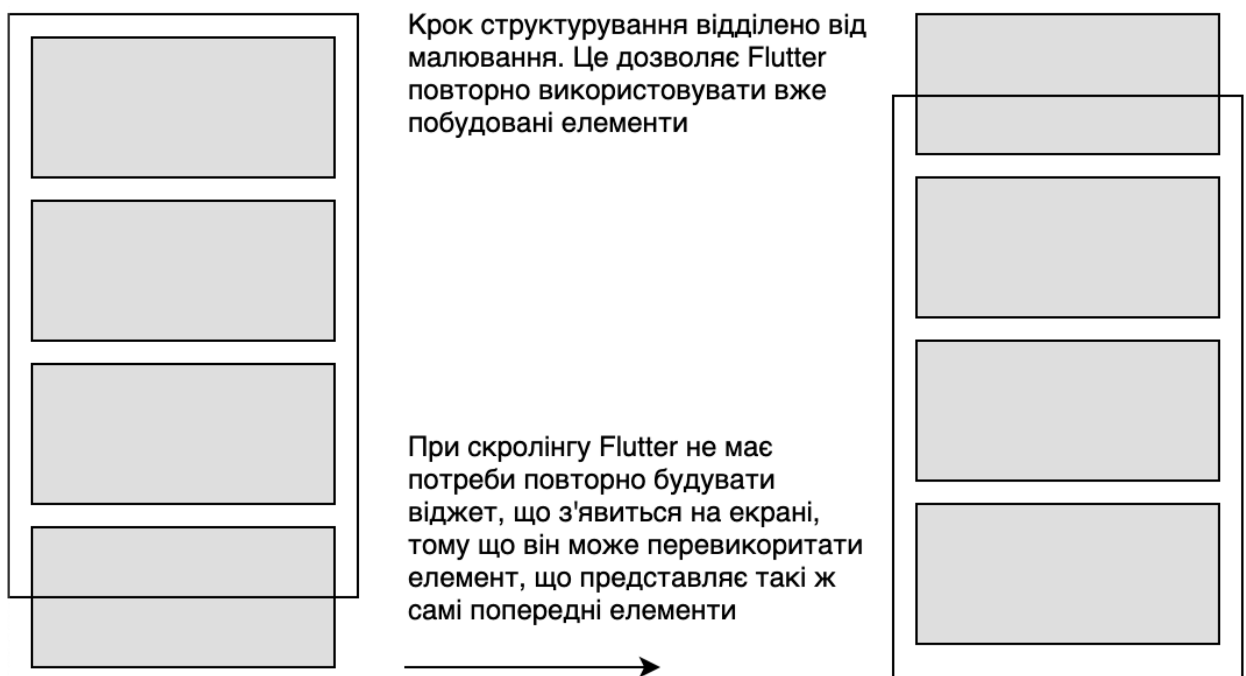

Рис. 1.14. Приклад повторного використання віджетів під час кроку структурування

1.4.3. Фактичне відображення віджетів на екрані

Після структурування движок комбінує дерево віджетів у відображуване представлення та наказує операційній системі відобразити його. Це називається разтризацією, і це останній крок.

На цьому відрисовка завершується. Для підсумування, основні ідеї Flutter полягають у наступному:

- усе є віджетом;
- Flutter є реактивним;
- обмеження віджетів визначаються батьківськими елементами для дочірніх;
- об’єкти стану (State) мають подовжений у порівнянні з віджетами життєвий цикл, та доволі часто використовуються повторно.

1.5. Висновки до першого розділу

Flutter фреймфорк бере на себе дуже великий об’єм роботи для досягнення високої ефективності та створення гарного користувальницького досвіду. Оскільки віджети, що описує програміст, ще не є фактичними елементами користувальницького інтерфейсу, а лише їх метаданими, Flutter доволі легко справляється із великою вложеністю елементів, що в свою чергу мінімізує вірогідність виникнення проблем з продуктивністю. Яким би розумним фреймворк не був, він не може застерегти від помилок бізнес логіки. Це може бути ситуація, коли на стороні відображення інтерфейсу усе працює правильно, але дані, передані у нього, є помилковими. В результаті цього взаємодія користувача з додатком або ускладнюється, або взагалі становиться неможливою. Для знаходження та вирішення таких проблем використовується процес тестування програмного продукту.

РОЗДІЛ 2
ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

При створенні якісно працюючого програмного продукту щонайменьше половина робочого часу витрачається на тестування. Невелика кількість програмістів вподобає тестування і навіть менша – розробку тестів. Особливо, коли розробка тестів та тестування займає більше часу, ніж процес написання коду та створення програми. При цьому у деяких розробників може складатися враження, що якщо тестування не виявило помилок, то процес написання тестів був лише марно втраченим часом.

Існує міф, що якби програмісти були дійсно талановитими у програмуванні, то необхідність у виправленні помилок відпала. Якби програмісти могли ідеально сконцентруватися у процесі роботи, використовували лише чисту архітектуру, таблиці рішень, всебічно придумували логіку взаємодії різних частин програми, тоді багів взагалі б не існувало. І оскільки баги все ж таки є, тоді програмісти погано виконують свою роботу, тому вони повинні почуватися винними за це, а тестування і розробка тестів є результатом признання програмістами цього фактора.

Програмісти, як і всі інші люди, не є досконалими, тому допускання помилок при написанні програми – це нормально. По статистиці при написанні програми, на сотню операторів існує від однієї до трьох помилок. Розробників, що пишуть програму без багів не існує. Дуже важливо приймати цей факт та розуміти, що кожен програміст допускає помилки у процесі роботи. Тому програмісти повинні не почуватися винними за свої помилки, а робити усе можливе, щоб запобігати їх виникненню у релізних версіях програм та знаходити їх якнаймога швидше.

2.1. Продуктивність і якість у програмному забезпеченні

Розглянемо процес виробництва масового продукту, що не є програмним забезпеченням. Яка б не була вартість розробки, вона складає невелику частину від сумарної вартості при амортизації великого обігу продукції. У процесі виробництва кожна виробнича стадія піддається контролю якості та тестуванню як окремих компонентів, так і продукту у цілому перед відправкою. Якщо на будь-якій стадії виявляються недоліки, у такому разі частина продукту буде або вилучена, або відправлена на повторне доопрацювання та корекцію. Продуктивність складальної лінії вимірюється сумую, затрачено на матеріали, повторну роботу, видалені компоненти, а також сумую, витраченою на контроль якості та тестування. Існує компроміс між витратами на контроль якості та витратами на виробництво. При приділенні недостатньої кількості уваги контролю якості, процент браку буде високим, що в свою чергу зробить високою собівартість продукції. Навпаки, якщо тестування настільки добре, що всі недоліки виявляються відразу ж після їх виникнення то таке тестування буде коштувати великих затрат, що знову ж таки й призведе до збільшення собівартості. Розробники виробничого процесу намагається встановити такий рівень контролю якості та тестування, що мінімізує чисту собівартість при досягненні допустимої якості. Витрати на тестування та контроль якості для виробничих товарів можуть можуть складати лише 2% при виробництві масових продуктів широкого використання та складати до 80% у таких продуктів як космічні кораблі або літаки, де виникнення помилок призводить до загрози життя людини.

Відношення між продуктивністю та якістю програмного забезпечення дуже сильно відрізняється від вищезгаданих виробничих продуктів. Вартість виробництва копії програмного забезпечення дуже мала. Вона включає у себе вартість фізичного диска носія (або взагалі не включає, якщо програма завантажена за допомогою веб сервісу) та декілька хвилин комп'ютерного часу. Тому можна сказати що процес фактичного виробництва займає мізерну частину

у порівнянні із розробкою програмного забезпечення. Процес підтримки програмного продукту відрізняється від підтримки фізичних продуктів. Насправді це не підтримка, а розширена розробка, у процесі якої вдосконалення розробляються та доставляються кінцевим користувачам, а дефекти видаляються. Найбільшу частину вартості програмного забезпечення складає є вартість багів: вартість знаходження, вартість виправлення, вартість розробки тестів що знаходять їх та вартість запуску цих тестів [14].

2.2. Мета тестування

Тестування та розробка тестів, як частина контролю якості, повинні також приділяти увагу запобіганню багів. При цьому тестування та розробка тестів не запобігають багам. Вони повинні мати можливість знаходити симптоми, що спричиняються багами. Тести також повинні надавати чітку діагностику, щоб баги могли бути з легкістю виправлені. Запобігання багів є першочерговою метою тестування. Відвернена помилка краще ніж та, що була знайдена та виправлена, оскільки якщо помилка відвернена, тоді й немає потреби у коді, якій би її виправляв. Більш того, немає потреби у повторному тестуванні, щоб доказати що програміст дійсно виправив помилку, ніхто не відчуває незручностей через допущений баг, та робота йде за розкладом.

Кращим засобом, ніж тестування, та одним із найбільш відомих інструментів запобігання помилок, є розробка тестів. Під час створення корисного тесту помилки можуть бути знайдені та відвернуті навіть до того як вони були написані. Дійсно, якщо в ході проектування програми продумувати те, як той чи інший функціонал буде тестуватися, то велика частина помилок просто відпаде цьому етапі. Це може бути застосовано не тільки на етапі проектування, але і також на етапах специфікації, написання коду та інших. В ідеальних умовах це могло бути настільки добрим засобом у запобіганні багів, що в тестуванні не було б потреби, оскільки всі баги у такому разі були б знайдені і виправлені на процесі проектування.

Але, на жаль, не можна досягти такого ідеального стану речей. Незважаючи на всі наші старання, баги все одно будуть існувати через людську натуру розробників. У доповнення, якщо тестування не досягає своєї першочергової мети – запобігання помилок, воно повинно досягати своєї наступної мети – виявлення помилок. Помилки далеко не завжди очевидні. Вони проявляються при відхиленні поведінки від очікуваної. В процесі розробки тесту повинні документуватися очікування, процедура тестування у деталях та результати тестів – все що схильно до помилок. Але знання лише про те, що програма не працює належним чином, не дає уявлення про суть помилки. У різних помилок можуть бути різні прояви, в той час як і у однієї помилки безліч симптомів. Симптоми і причини можуть бути визначені тільки завдяки використанню великої кількості маленьких, але деталізованих тестів.

2.3. Етапи прийняття тестування

Щоб відповісти на питання, яку кінцеву мету ставить перед собою тестування, можна привести п’ять стадій, в ході яких відношення розробників до цього процесу поступово змінюється:

1. Розробник не бачить різниці між тестуванням і налагодженням та вважає, що окрім як допоміжний інструмент для налагодження, тестування не має сенсу.

2. Тестування потрібно для того, щоб показати що програмне забезпечення працює.

3. Тестування потрібно для того, щоб показати що програмне забезпечення не працює.

4. Тестування програмного забезпечення не ставить перед собою мету будь що доказувати. Воно потрібне для того, щоб знизити ризик ймовірності виникнення проблем в ході роботи програми до допустимого значення.

5. Тестування не є певною дією. Це ментальна дисципліна, завдяки якій програмне забезпечення містить мінімальну кількість помилок при мінімальних трудовитратах на тестування.

2.3.1. Етап неприйняття тестування

На цьому етапі розробник просто не бачить різниці між налагодженням і тестуванням та відноситься до тестування, як до чогось чужого та непотрібного. Здебільшого це розробники, які або тільки починають писати код та все ще не знають про важливість та необхідність тестування, або які вже мають досвід у програмуванні і якимось чином все ще не почали застосовувати тестування в ході розробки. Розробники, що відносяться до другого типу, можуть знаходити безліч причин за якими, як вони вважають, у написанні тестів немає необхідності. Це можуть бути такі причини як:

- тут нічого тестувати;
- це зайва трата часу;
- це і так відмінно працює;
- тестування це занадто складно.

Якщо така думка домінує у компанії, це означає що там не може бути ефективного тестування, як і контролю якості, як і самої якості.

2.3.2. Етап доведення відсутності проблем

Після усвідомлення того, що тестування все ж необхідно для розробки програмного забезпечення, розробник зазвичай починає вважати, що єдина мета тестування – це всього лише довести, що програмне забезпечення працює правильно. Це судження є помилковим, оскільки достатньо лише одного неправильно проведеного тесту, щоб показати що програма не працює належним чином. У той же час нескінченна кількість тестів не зможе довести зворотне. Таким чином мета, яку ставить перед собою розробник на цьому етапі, є

недосяжною. Недосяжною ця мета є через те, що ймовірність показати що програмне забезпечення працює зменшується в міру того як збільшується кількість тестування. Таким чином чим більше програмне забезпечення тестується, тим більше є вірогідність знаходження помилки у ньому. Тому слід прийняти факт, що помилки не виявляються тільки в тому випадку коли тестування немає. Але у такому разі вони все ж таки з’являються, при тому на релізних версіях програми у кінцевих користувачів.

2.3.3. Етап доведення наявності проблем

Після досягнення третьої фази, розробники тестів починають працювати проти тих, хто розробляє програмний продукт. Зазвичай, це різні люди – тестувальники та програмісти. Але це може бути і саморуйнівний підхід лише однієї людини, програміста. Дана фаза призводить до глибоких, всеохоплюючих тестів.

Наявність всього лише одного невдало пройденого тесту досягає мети, поставленої в даному етапі. Незважаючи її досягнення, залишаються деякі проблеми. Тест демонструє помилку, після чого програміст її виправляє, далі розробник тесту створює і виконує інший тест, націлений на демонстрацію іншої помилки. Третя фаза призводить до нескінченної послідовності надмірно прискіпливих тестів. Доводячи до крайнощів, цей процес ніколи не закінчується, у результаті чого створюється надійне програмне забезпечення, яке ніколи не виходить на ринок.

2.3.4. Етап зниження ризику

На цьому етапі відбувається прийняття принципів статистичного контролю якості. Завдяки знаходженню та виправленню помилок тестування покращує продукт. При проходженні тесту якість продукту не змінюється, чого не можна сказати про уявлення розробників щодо якості продукту. Кожен

успішно або неуспішно пройдений тест зменшує кількість ризику, який програмісти відчувають по відношенню до програмного продукту. При збільшенні кількості тестів тести стають більш глибокими, всеохоплюючими та тестують крайні ситуації поведінки системи. Це в свою чергу дає більше впевненості в продукті. При достатній впевненості продукт можна випускати в світ [19].

2.3.5. Етап ментального стану

На останньому етапі знання того що тестування може і що не може, поєднане зі знанням про те, що робить програмне забезпечення таким, щоб його можна було протестувати, призводить до програмного забезпечення, яке не потребує великої кількості тестування для досягнення цілей попередніх етапів. Здатність програмного забезпечення бути протестованим є метою з двох причин. Перша і очевидна причина полягає в тому що це призводить до зменшення кількості часу який потрібно витратити на перевірку ПЗ. Друга і більш важлива причина полягає в тому, що код який можна протестувати, має меншу кількість багів ніж код який складно протестувати. Поєднані разом, ці два фактори посилюють вплив один одного і тим самим значно збільшують продуктивність.

Слід звернути увагу що всі вищеперелічені цілі повинні підсумовуватися. Налагодження покладається на тестування як на інструмент для дослідження гіпотетичних причин симптомів помилок. Для того щоб “зламати” програмне забезпечення існує багато засобів, однак далеко не всі вони мають зв'язок з функціональними вимогами до програмного продукту. Таким чином мета третьої фази може ніколи не бути завершеною. “Зламувати” програмне забезпечення не має особливого сенсу до тих пір, поки немає реальних доказів того що воно працює. І нарешті одного лише ментального стану недостатньо – навіть таке програмне забезпечення, яке ідеально складено з можливістю тестування, все одно повинно відлагоджуватися, повинно працювати і має бути стійким до спроб його “зламати”.

2.4. Проектування тестів

Незважаючи на те, що програмісти, тестувальники і менеджери знають що код повинен бути спроектован і протестований, більшість з них не знають про те, самі тести також повинні бути спроектовані і протестовані в ході не менш ретельного і контрольованого процесу, ніж процес написання коду. Занадто часто тест-кейси застосовуються без попереднього аналізу програмних вимог або структури. Таке проектування тестів, якщо його взагалі можна назвати проектуванням, всього лише серія випадкових імпровізацій що не документуются перед або після виконання тестів. Оскільки вони не були формально спроектовані, вони не можуть бути в точності повторенні, і ніхто не впевнений, з якої причини вони були написані: через виникнення помилки, або через вимоги системи або взагалі без будь-якої на те причини. Після ймовірного виправлення помилки ніхто не впевнений в тому, що повторний тест був ідентичний тому тесту, в ході якого було знайдено баг. Довільні тести корисні під час налагодження, коли їх головною метою є допомога при знаходженні помилки. Але довільні тести все ж таки робляться для підтримки налагодження і не є повноцінною заміною для спроектованих тестів [22].

Стадія проектування тестів в програмуванні повинна бути явно вказана. У процесі розробки, який можна було б описати як "проектування, написання коду, тестування, налагодження", тестування повинне проводитися після завершення кожного кроку і перед переходом до наступного. Надання кроку проектування тестів явного місця в схемі процесу дає більше видимості цій непомітний на перший погляд половині роботи, яку досить часто очікують побачити на етапі “тестування і налагодження”. Це знижує ймовірність того що проектування тестів буде недооцінене за умов обмеженого бюджету, стисненого розкладу і наявності смутної надії на те, що можливо цього разу, одного єдиного разу, система буде створена без помилок.

2.5. Особливості тестування мобільних додатків

Деякі розробники можуть вважати що мобільне тестування не відрізняється від тестування інших програмних продуктів. На їхню думку мобільний пристрій, будь то телефон або планшет, всього лише той же комп'ютер, але з маленьким екраном. Глянувши на екран свого смартфона деякі можуть побачити лише маленький комп'ютер з крихітними іконками на екрані, інші ж – персональний комп'ютер з великою кількістю сенсорів і можливостей вода, який містить всі приватні дані.

Насправді ж смартфон – це дійсно персональний комп'ютер який зберігає величезну кількість персональних даних, такі як електронна пошта, SMS, фотографії, музика, відео та інші. Ці дані можуть бути отримані в будь-який момент незалежно від того, де знаходиться власник смартфона. Також він може бути використаний як навігатор або інформаційна система для отримання інформації про оточення. З цієї причини користувачі очікують від додатків, що вони будуть надійними, швидкими і простими у використанні.

2.5.1. Очікування користувачів від мобільних додатків

Головним випробуванням і об'єктом, на якому потрібно сфокусуватися мобільним командам, являється користувач. Факт того, що у кожного користувача унікальні очікування, ускладнює розробку і доставку “правильного” додатку кінцевим користувачам. Як показує статистика і звіти, очікування користувачів щодо мобільних додатків набагато вище ніж щодо браузерних. Велика кількість звітів і опитувань стверджує що приблизно 80% користувачів видаляє додатки після першого використання. Основними чотирма причинами видалення є: поганий дизайн, складність у використанні, довгий час завантаження і аварійне завершення роботи додатку відразу після установки. Приблизно 60% користувачів видалять додаток, який вимагає реєстрації і більш ніж половина користувачів очікує, що застосунок запуститься менш ніж за 2

секунди. Якщо додаток витрачає на це більше часу – його видаляють. Виходячи з цього стає ясно, що у користувачів мобільних додатків дійсно високі очікування щодо зручності користування, продуктивності і надійності.

В даний момент більш ніж 2 мільйони додатків доступно у магазинах додатків від великих компаній. Велика частина додатків виконують однакові завдання, а це означає що завжди існує як мінімум один додаток-конкурент, що дає можливість користувачам легко завантажити інший додаток всього лише в один клік. При розробці і тестуванні мобільного додатка важливо звертати увагу на такі моменти:

- збір інформації про можливу цільову аудиторію;
- опитування користувачів про їхні потреби;
- додаток має вирішити проблему користувача;
- зручність користування має величезну роль;
- додаток має бути надійним і стійким до відмов;
- додаток має бути продуктивним;
- додатки повинні гарно виглядати.

Звичайно існує безліч інших речей, які також слід брати до уваги, але при дотриманні вище описаних пунктів користувачі швидше за все будуть задоволені додатком. У воєнно-морських сил США є акронім, що використовується під час проектування – “KISS”, що розшифровується як “не ускладнюй” (keep it simple). Він стверджує, що для підтримання високого рівня надійності системи, вона не повинна бути занадто ускладнена, оскільки чим більше частин має система, тим більша вірогідність виникнення помилки. Популярність даної фрази не залишилась без уваги у сфері програмування, та у 70х роках вона сформувала принцип KISS. Цей принцип дійсно дуже корисний для програмних продуктів. Не слід просто так додавати в програмне забезпечення ту чи іншу функцію. Підтримання його невеликим, легким і простим в більшості випадків це ключ до задоволених користувачів.

2.5.2. Мобільність та мережі даних

Інше випробування, з яким доводиться стикатися мобільним додаткам більше, ніж програмному забезпеченню що запускається на комп'ютерах, це факт того що користувачі постійно пересуваються під час використання додатків, і при цьому додатки потребують наявності Інтернет з'єднання для отримання даних з сервера та надання користувачу різних оновлень та інформації.

Мобільні додатки потрібно тестувати в реальному житті, у реальному оточенні, де потенційний користувач буде їх використовувати. Наприклад, якщо потрібно тестувати мобільний додаток для сервісу таксі, який повинен давати можливість записатися на поїздку, подивитися маршрут, здійснити оплату, то ці функції повинні бути протестовані в машині таксі. Інакше не можна гарантувати те, що функціонал буде працювати так як і очікувалося.

Звичайно є такі ділянки у додатку, які можна протестувати в офісних умовах, такі як правильна побудова користувальницького інтерфейсу або можливість установки додатка, але так само є такі моменти, як відображення руху машини під час їзди або вирішення і відстеження конфліктних ситуацій між пасажиром і водієм. Швидкість Інтернет-з'єднання може змінюватися в залежності від району міста, в якому в даний момент знаходиться таксі. Ймовірно що ближче до центру міста швидкість Інтернет-з'єднання буде стабільно високою і при цьому вона буде низькою на околицях. Слід подумати про те, що трапиться з додатком під час поганого Інтернет-з'єднання або за його відсутності. Чи буде воно успішно продовжувати роботу або аварійно завершиться? Складно відповісти на це питання в офісних умовах. При мобільному тестуванні потрібно бути мобільним і бути підключеним до мобільних мереж даних.

З перерахованого вище видно що вкрай важливо тестувати додаток в реальному оточенні і проводити тести у в мережах даних з різними пропускними здатностями, оскільки пропускна здатність може надавати величезний ефект на додаток. Низька пропускна здатність мережі може призвести до несподіваних

повідомлень про помилки, також перехід від високої до низької пропускної здатності може призвести до проблем з продуктивністю або сповільненої роботи додатку [25].

2.5.3. Мобільні пристрої

Якщо взяти сучасний смартфон в руки і подивитися на нього, то речі які потраплять під погляд будуть наступними: це тачскрін, кілька фізичних кнопок, гніздо для зарядки, роз'єм для навушників і камера. На цьому ймовірно все – рідко коли на сучасному телефоні є більш ніж чотири фізичні кнопки.

Під час коли поняття "мобільний телефон" стало синонімом слова "смартфон" важливо пам'ятати, що раніше були інші типи мобільних телефонів, так звані "прості телефони" і "просунуті телефони" у яких було набагато більше фізичних кнопок для здійснення дзвінків або написання повідомлень. Список можливостей традиційного “простого” телефону сильно обмежений, це можливість зробити дзвінок, написати повідомлення або зберегти контакт в телефонну книгу. “Просунуті” телефони вже мають ігри, календар або навіть базову версію web-браузера з можливістю підключення до мережі Інтернет. Але в рамках функціоналу і розширюваності всі ці телефони є базовими, оскільки вони не надають користувачам можливості встановлювати додатки або оновлювати програмне забезпечення легким чином до нової версії. Два цих типу телефонів все ще доступні для покупки але починаючи з 2013 року більшу частину проданих мобільних телефонів складали смартфони і така тенденція з кожним роком все збільшується.

Телефони, які використовуються зараз є зовсім іншими на відміну від старих телефонів. Сучасні смартфони є невеликими, але при цьому повноцінними комп'ютерами з великою кількістю можливостей в рамках апаратного і програмного забезпечення. В їх розпорядженні такі різноманітні датчики, як датчик освітленості, датчик наближення, датчик прискорення, GPS і багато інших. Також у всіх сучасних смартфонах є наявність wi-fi модуля, NFC,

передньої та задньої камер і можливість підключення до Інтернету через мережі сотового зв'язку. Також у смартфоні є набір інших різноманітних функцій і можливостей, що залежить від платформи і виробника.

З точки зору програмного забезпечення, смартфон надає велику кількість прикладних програмних інтерфейсів, які також називаються API, що активно використовуються сторонніми додатками, тим самим ще збільшуючи можливості смартфону. Дві основні мобільної платформи iOS і Android, мають величезну кількістю комбінацій програмного і апаратного забезпечення, з якими доводиться стикатися в ході тестування. Наявність такої великої кількості комбінацій називається фрагментацією. Дана фрагментація є великою темою для обговорень і являє собою ще одну проблему, яка виникає в ході мобільного тестування [31].

Неможливо протестувати додаток на всіх існуючих в світі комбінаціях апаратного і програмного забезпечення. Ускладнює цю задачу також те що тестувати необхідно на реальних пристроях в реальному середовищі. Тому важливо докладати зусилля на те щоб замість тестування на великій кількості різних пристроїв тестувати на невеликій, але правильній частині пристроїв.

Але як сформувати поняття про правильні пристрої? Чи має цей пристрій відноситися до певної платформи, чи бути останнім флагманом виробника або ж мати останню версію програмного забезпечення? Щоб визначити які пристрої найкраще підходять для тестування слід розуміти що всі застосунки унікальні, і мають свої вимоги, проблеми та клієнтську аудиторію. Беручи до уваги вищесказане, наступні пункти можуть відповісти на питання, які ж мобільні пристрої є "правильними":

- цільова аудиторія;
- рік типового користувача;
- найпопулярніша платформа серед користувачів;
- найпопулярніший пристрій;
- версія програмного забезпечення, що встановлене на більшості пристроїв;

– які сенсори пристрою використовує застосунок;
– комунікація застосунку із зовнішнім оточенням;
– головна функція застосунку.

Звичайно ж цей перелік далеко не вичерпаний, проте при визначенні і розумінні перерахованих вище пунктів пошук необхідних пристроїв стає значно легшим.

2.5.4. Випуск нових версій мобільних продуктів

У світі мобільних пристроїв кожен виробник виробляє як мінімум один, а то й кілька флагманів на рік, це не враховуючи середньобюджетні пристрої, яких може бути випущено до десяти нових за рік. Велика частина цих пристроїв, а особливо флагмани отримують нові можливості. Це означає нові можливості для певних груп користувачів та нові сценарії використання. Більшою мірою це може бути застосовано до пристроїв на операційній системі Android, оскільки кожен рік випускаються нові пристрої з новою версією операційної системи, яка включає у себе новий функціонал та нові інструменти прикладного програмування інтерфейсів. Протягом одного лише року випускається декілька оновлень платформи, що включають в себе новий функціонал або виправлення помилок. У процесі контролю якості необхідно бути переконаним, що мобільний застосунок працює коректно на найостанніх версіях програмного та апаратного забезпечення.

Відповідаючи на питання, чи потрібно тоді купувати кожен новий телефон, що виходить на ринок або постійно оновлювати програмне забезпечення до останньої версії, слід приймати до уваги цільову аудиторію та специфікації додатка. Якщо цільова аудиторія завжди використовує найостаннішу модель флагмана від певного виробника, значить скоріш за все, вона оновиться до останньої моделі пристроїв, як тільки вона вийде на ринок. У такому разі слід заздалегідь подбати, щоб додаток працював коректно на цих нових пристроях, а також на останніх версіях програмного забезпечення. Але не слід нехтувати і

старими версіями платформ, оскільки не усі користувачі завжди оновлюють свої пристрої. Деяких влаштовує їх версія операційної системи, а деякі просто не знають як її оновити. Тому слід перевіряти коректність роботи мобільного додатку і на старих версіях операційних платформ, оскільки на них можуть бути відсутні деякі нові API, що присутні у нових версіях.

Хорошим засобом для вирішення вищеописаних проблем є покупка нових телефонів з самою останньою версією операційної системи. Але це досить дорого і не у кожного проекту є такий бюджет, який може дозволити собі покупку нових пристроїв, які будуть використовуватися лише кілька місяців. Рішенням цієї проблеми може бути оренда пристроїв. Існують різні сервіси які дають в оренду як фізичні пристрої, так і віддалений доступ до пристроїв через хмарні сервіси. Розумним підходом може бути наявність невеликої кількості пристроїв які, збігаються з найчастішими пристроями цільової аудиторії, наприклад у кількості 15 штук, які таким чином будуть покривати до 90% користувальницьких сценаріїв. Для вирішення інших, не критичних 10% сценаріїв та проблем, можна використовувати оренду пристроїв.

Підсумовуючи вищесказане, для охоплення групи стрімкого росту програмного та апаратного забезпечення мобільних пристроїв, можна керуватися наступною стратегією:

- слідкувати за ринком мобільних пристроїв та відповідним програмним забезпеченням;
- дізнаватися про новий функціонал операційних систем, який збираються випустити на ринок;
- перевіряти наявність нових пристроїв у цільової користувальницької аудиторії за допомогою збору статистичних даних;
- якщо бюджет дозволяє, купувати нові пристрої, якщо ж ні, тоді брати в оренду.

2.5.5. Тестування мобільного програмного забезпечення

Із описаного вище можна зробити висновок, що тестування мобільного програмного забезпечення дійсно відрізняється від тестування десктопних або веб платформ. Вплив фізичних пристроїв на мобільні додатки надає набагато більший ефект в порівнянні з ефектом комп'ютера на звичайні програми. Наявність такої величезної кількості смартфонів на ринку змушує сфокусуватися на апаратних особливостях фізичних пристроїв. Також слід брати до уваги те, що користувачі часто перебувають у русі при використанні мобільних додатків і мереж даних, тому при тестуванні мобільних застосунків необхідно відтворювати дані умови. Очікування користувачів мають не менш важливу роль в житті мобільного застосунку, тому їх також слід враховувати. Слідкуючи принципу KISS, не потрібно перенавантажувати додаток безліччю непотрібних користувачу функцій, а слід фокусуватися на цільовому функціоналі, та робити його простим, швидким та зручним у використанні [34].

2.6. Процес тестування мобільних додатків

При виборі пристрою для тестування програми є три опції, це симулятор, емулятор та реальний пристрій.

Прикладом емулятора мобільного пристрою, що є десктопним додатком, який транслює інструкції вихідного коду скомпільованого додатку таким чином, що додаток може бути запущений на десктопному комп'ютері, є Андроід Емулятор (Android Emulator). Завдяки тому, що емулятор поводиться в точності як фізичний мобільний пристрій і має таку ж операційну систему, тестувальники і розробники мають можливість тестувати або здійснювати налагодження додатків на ньому. Оскільки додаток в такому випадку запущений на комп'ютері, не всі фізичні властивості мобільного пристрою можуть бути проемульовані, наприклад сенсори або жести торкання. Але на ранній стадії розробки емулятори

можуть бути дуже корисними, оскільки вони дозволяють досить швидко отримати уявлення та результат реалізованого функціоналу.

Другий варіант, симулятор, є менш складним програмним забезпеченням, яке симулює невеликий набір апаратних наборів пристрою і його поведінку. На відміну від емуляторів, симулятори має схожість тільки з цільової платформою і симулюють апаратне забезпечення реального пристрою, що робить їх набагато швидше емуляторів. У симуляторах відсутня можливість тестування специфічних апаратних елементів пристрою. Однак користь від симулятора також досить висока, як і від емулятора. Основна різниця між емуляторами та симуляторами полягає в тому, що симулятор намагається скопіювати поведінку мобільного пристрою у той час як емулятор намагається скопіювати цілком всю внутрішню архітектуру мобільного пристрою і таким чином більш точно імітує поведінку цільової платформи. Прикладом мобільного симулятору є iOS симулятор від Apple.

Оскільки користувачі під час реального використання програми знаходяться в більшості випадків у русі, емулятори та симулятори повинні використовуватися на ранніх етапах розробки для перевірки базового функціоналу додатка, в той час як реальні пристрої повинні використовуватися ближче до середини та завжди на кінцевій стадії розробки.

2.6.1. Мануальне та автоматизоване тестування

Під час тестування програми виникає досить важливе питання. Чи повинно здійснюватися тестування виключно за допомогою автоматизованих тестів або ж тільки за допомогою мануальних тестів, або це ж повинен бути комбінований підхід. Вибір рішення залежить від функціоналу додатка. Одного лише автоматизованого тестування може бути недостатньо. Це може бути неефективно через те, що не всі все специфічні функції, пов'язані з мобільним пристроєм, можуть бути автоматизовані. В офісних умовах досить складно протестувати функціонал, пов'язаний з даними про місцезнаходження або ж

іншими даними, що отримуються сенсорами мобільного пристрою з оточення. Виходячи зі складності виникає результат того що програмні помилки будуть потрапляти у релізні версії кінцевих користувачів. У наведеному випадку правильним рішенням є використання мануального тестування. Однак використання тільки мануального тестування недостатньо у більшості випадків. Мануального тестування може бути достатньо тільки для простих додатків з дуже обмеженим функціоналом або ж для додатків, які з'являються в магазині додатків на обмежений період часу. У всіх інших випадках необхідно комбінувати мануальне і автоматизоване тестування. Перед виконанням автоматизації завжди необхідно проводити мануальное тестування. Кожен новий функціонал повинен бути протестований в ручному режимі на різних пристроях. Після проходження мануального тестування можна визначати, які частини функціоналу вимагають автоматизації [37].

2.6.2. Звичайне тестування

Крім тестування функціоналу специфічного для мобільних додатків також необхідно тестувати додаток таким чином, яким тестується Web або десктопні програми. Необхідно проектувати тест-кейси, управляти наборами тестових даних і звичайно ж запускати тести.

На рисунку 2.1 відображені типові дії у процесі контролю якості програмного продукту. Існує два типи контролю якості, з фокусом на продукт та з фокусом на процес. На стадії продукту виконується пошук помилок, в той час як на стадії процесу робиться усе можливе для недопускання нових помилок у програмному забезпеченні. Під час фази фокуса на процесі аналізуються методи, мова написання, інструменти, стандарти і процеси за допомогою яких розробляється програмне забезпечення. На цьому етапі люди, що відповідають за контроль якості, повинні слідкувати щоб раніше прийняті підходи до розробки продукту були дотримані. Вони повинні допомагати розробникам і іншим

членам команди дотримуватися заданого процесу для того, щоб запобігати виникненню помилок.

Рис. 2.1. Методи контролю якості програмного продукту

Фаза фокусу на продукті складається з статичного та динамічного тестування. Під час статичного тестування програмне забезпечення не запускається. Розробники і тестувальники повинні виконувати перегляд процесу розробки програмного забезпечення за допомогою або перегляду коду перед його відправкою у репозиторій або шляхом перегляду документів і специфікацій перед початком розробки. Під час статичної фази код додатка також може перевірятися за допомогою статичних аналізаторів коду для перевірки його на відповідність стандартам написання і наявності базових помилок.

Під час динамічного тестування відбувається запуск коду програмного продукту для того, щоб оцінити його поведінку під час роботи. Динамічне тестування ділиться на тестування чорним ящиком і тестування білим ящиком. Тестування методом білого ящика передбачає тестування зі знанням внутрішньої роботи структури, класів і методів програмного забезпечення. Зазвичай тестування методом білого ящика проводиться розробниками, які тестують свій код на модульному рівні за допомогою спеціальних інструментів тестування, таких як JUnit. Тестування методом білого ящика включає в себе такі техніки, як

тестування потоку контролю і потоку даних, тестування розгалужень, покриття операторів, умов та функцій.

Тестування методом чорного ящика передбачає відсутність знань про внутрішню структуру методів і класів. Цей підхід зазвичай виконується тестувальниками програмного забезпечення, які знають про те, що програмне забезпечення має зробити, але не знають про те, як воно це виконує. Важливою складовою тестування методом чорного ящика є написання тест-кейсів. Існують різні техніки при проектуванні правильних тест кейсів, наприклад техніка граничних значень, техніка таблиці прийняття рішень і техніка еквівалентного поділу класів. Техніка граничних значень свідчить, що помилки можуть проявлятися при граничних умовах і поведінках при роботі системи. Такі помилки можуть призводити до аварійного завершення роботи, зависання додатку та іншої некоректної поведінки. Таблиці рішень допомагають розбити функціонал додатку на багату кількість малих частин, при яких валиві частини функціоналу підпадають під фокус. Еквівалентний поділ класів – це техніка, що допомагає розробити тест-кейси які знаходять проблемні класи. Дана техніка дозволяє зменшити кількість необхідних тест-кейсів. Даний підхід зазвичай використовується для перевірки точок вводу даних у додатку.

2.7. Мок об’єкти та крихкість тестів

Існує досить багато суперечок на тему використання мок об'єктів в тестуванні. Одні вважають що мок об'єкти – це настільки чудовий інструмент, що їх необхідно застосовувати мало не в кожному тесті. Інші вважають, що мок об'єкти призводять до крихкості тестів і намагаються зовсім їх не використовувати. Обидві точки зору справедливі і мають місце бути, оскільки мок об'єкти дійсно досить часто призводять до крихкості тестів. Крихкі тести складно переносять рефакторинг. Але також є досить багато ситуацій в яких використання мок об'єктів є вигідним, а іноді і просто необхідним підходом [45].

2.7.1. Типи двійників у тестах

Термін “тестовий двійник” є всеохоплюючим терміном, що описує всі види нерелізних, штучних залежностей в тестах. Поняття цього терміна можна порівняти з каскадером у фільмах. Найбільшою користю від використання тестових двійників є полегшення тестування. Дані об'єкти передаються в тестуюму систему замість реальних об'єктів, які можуть бути складні в створенні або мати проблеми при роботі у тестовому оточенні.

Існує п’ять видів тестових двійників: холості, заглушки, шпигуни, мок-об’єкти та фальшивки. Незважаючи на таку велику різноманітність, вони можуть бути розділені на два типи: моки і заглушки (рис. 2.2).

Рис. 2.2. Поділення тестових двійників на мок та заглушки

Відмінність між даними двома категоріями полягає в наступному. Мок об'єкти допомагають емулювати і досліджувати вихідну поведінку та вихідні взаємодії. Під час таких взаємодій тестова система звертається до її залежностей для зміни їх стану. Заглушки ж допомагають емулювати вхідні взаємодії. Під час таких взаємодій тестована система звертається до своїх залежностей для отримання вхідних даних (рис. 2.3).

Тестовий двійник, що емулює відправку HTTP запиту, зовнішньої вихідної взаємодії, яка призводить до побічного ефекту на HTTP сервері, являється мок об’єктом. Навпаки, отримання даних з локального сховища є прикладом вхідної взаємодії, яка не призводить до побічних ефектів. Об’єкт, що емулює таку поведінку, називається заглушкою.

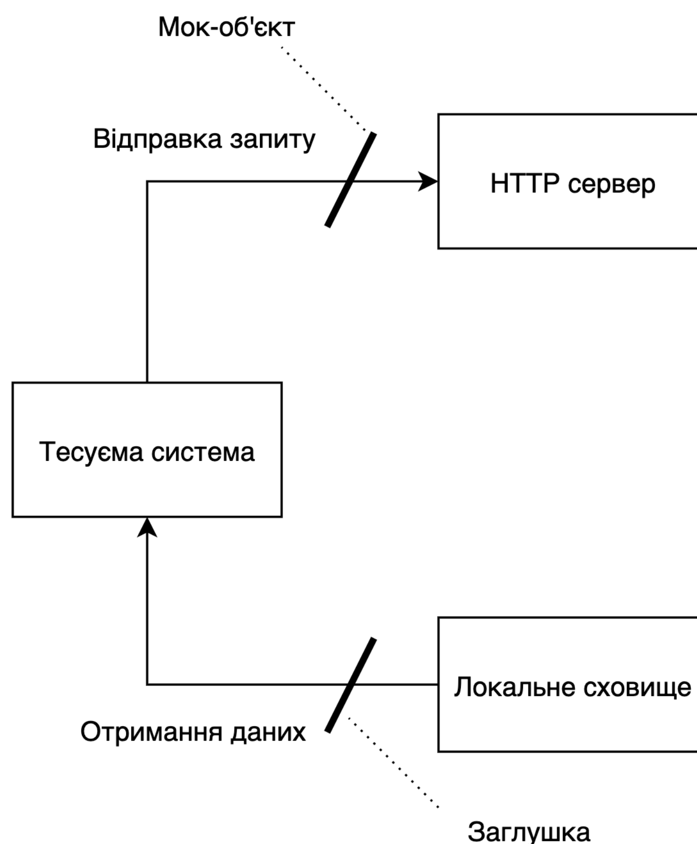

Рис. 2.3. Відмінності між мок-об’єктами та заглушками

Всі інші відмінності між п'ятьма варіантами є незначними деталями реалізації. Наприклад, шпигуни виконують ту ж роль що і моки. Різниця полягає в тому, що шпигуни пишуться вручну, в той час як моки створюються за допомогою мок-фреймворків. С іншого боку, заглушка, холостий і підробка відрізняються рівнем функціоналу, який вони містять у собі. Холостий об'єкт є простим захардкоженим значенням, таким як значення null або рядок. Він використовується для задоволення сигнатури методу тестованої системи і не має відношення до фінального результату. Заглушка є повномасштабною залежністю, яка налаштовується таким чином, щоб повертати різні значення при різних умовах. Фальшивка виконує в більшості випадків ту ж саму роботу що і заглушка. Різниця у причини її створення, оскільки фальшивка зазвичай створюється для заміни залежності, яка все ще не існує.

Крім різниці у вхідних і вихідних взаємодіях, основна різниця між моками і заглушками полягає в тому, що моки допомагають емулювати і досліджувати

взаємодію між тестованою системою і її залежностями, тоді як заглушки всього лише допомагають емулювати ці взаємодії.

2.7.2. Мок як інструмент та як тестовий двійник

Термін “мок” може використовуватися по-різному в залежності від ситуації. Деякі люди помилково називають моками всіх тестових двійників, в той час мок є лише одним з підтипів цих двійників. Моками також можуть називатися класи, що належать бібліотекам, які виконують підміну (мок) даних. Ці класи допомагають створювати мок об'єкти, але самі по собі вони не є моками. На рисунку 2.4 зображено приклад таких об’єктів.

```
class MockSender extends Mock implements
Sender {}

class Sender {
  void sendData(String data){
    //
  }
}

test('send email', () {

      var mock = MockSender();
      var director = SendDirector(mock);
      director.setHeaders("Key: Value");

      verify(mock.sendData("Key:
Value")).called(1);
    });
```

Рис. 2.4. Використання мок класів із мок бібліотеки для створення мока

В даному прикладі використовується Mock клас із бібліотеки mockito для Flutter. Цей клас є інструментом, який дозволяє створити тестового двійника – мок об’єкт. Іншими словами клас “Mock” є мок-інструментом, в той час як екземпляр цього класу “mock” є тестовим мок-двійником. Доволі важливо розділяти поняття мок як інструменту з моком, як тестовим двійником, тому що

мок як інструмент може використовуватися для створення як мок-об'єктів так і заглушок (рис. 2.5). Тест у наступному прикладі також використовує Mock клас, але сам екземпляр класу є заглушкою, а не моком.

```
class MockDataProvider extends Mock
implements LocalDataProvider {}

class LocalDataProvider {
  String getPersonalName(){

  }
}

var stub = MockDataProvider();

when(stub.getPersonalName()).thenReturn("Ivan
Siabro");

      var director = StorageDirector(stub);
      var snapshot =
director.createUserDataSnapshot();

      expect(snapshot.data, "Ivan Siabro");
```

Рис. 2.5. Використання мок класів із мок бібліотеки для створення заглушки

Даний тестовий двійник емулює вхідну взаємодію – виклик методу, який надає тестуємій системі вхідні дані.

2.7.3. Перевірка виклику методів

Як було сказано раніше, моки допомагають емулювати і перевіряти вихідні взаємодії між тестованою системою і її залежностями, в той час як заглушки допомагають тільки емулювати вхідні взаємодії, але не перевіряти їх. З цього випливає, що взаємодія з заглушками ніколи не повинна перевірятися. Звернення тестованої системи до заглушки не є частиною кінцевого результату, який виробляє система. Такий виклик є лише засобом для отримання кінцевого

результату. Заглушка надає вхідні дані, на основі яких тестована система надалі генерує вихідний результат.

Перевірка взаємодії з заглушками є загальним антипаттерном, який призводить до крихкості тестів. Єдиним правильним шляхом для уникнення хибнопозитивних тестів і підвищення їх здатності до рефакторингу є підхід, при якому тести перевіряють лише кінцевий результат, а не деталі реалізації. Хибнопозитивні тести – це такі тести, які видають позитивний результат при проходженні, але містять у собі помилку, в результаті чого перевірка не є істиною. На рисунку 2.6 зображено частину раніше наведеного прикладу перевірки взаємодії за допомогою мока.

```
verify(mock.sendData("Key: Value")).called(1);
```

Рис. 2.6. Перевірка взаємодії за допомогою мока

Така перевірка відповідає фінальному результату, який має значення для бізнеса. Відправка даних об’єктом Sender є тим кінцевим результатом, у якому бізнес зацікавлений. В той же час, виклик методу getPersonalName(), що зображено на рисунку 2.5, взагалі не є результатом. Це лише внутрішні деталі реалізації того, як тестуєма система збирає необхідні дані для створення снапшоту. Таким чином перевірка даних викликів призведе до крихкості тестів. Не повинно мати значення як тестована система отримує кінцевий результат до тих пір, поки цей результат є коректним. На рисунку 2.7 наведено приклад такого крихкого тесту.

У наведеному прикладі є перевірка виклику методу getPersonalName(), який є лише деталлю реалізації. Практика перевірки речей, які не є частиною кінцевого результату, називається надмірною специфікацією. В більшості випадків надмірна специфікація виникає якраз таки при перевірці взаємодій. Перевірка взаємодій з заглушками є недоліком, який дуже легко помітити, оскільки тести не повинні перевіряти ніяких взаємодій з заглушками. З моками

трохи складніше, оскільки не всі використання моків призводять до крихкості тестів, але багато з них можуть призвести до цього.

```
var stub = MockDataProvider();
when(stub.getPersonalName()).thenReturn("Ivan Siabro");

var director = StorageDirector(stub);
var snapshot = director.createUserDataSnapshot();

expect(snapshot.data, "Ivan Siabro");
verify(stub.getPersonalName()).called(1)
```

Рис. 2.7. Приклад перевірки взаємодії заглушки

2.7.4. Спільне використання моків та заглушок

В деяких випадках потрібно створити такого тестового двійника, який проявляє якості одночасно як і мока, так і заглушки, як зображено на рисунку 2.8.

```
var marketMock = MarketMock();
when(marketMock.hasFood(Food.Burger, 3)).thenReturn(false);
var buyer = Buyer();
bool bought = buyer.buy(marketMock, Food.Burger, 3);
expect(bought, false);
verifyZeroInteractions(marketMock.decreaseFood(Product.Burger, 3));
```

Рис. 2.8. Приклад спільного використання моків та заглушок

Об'єкт marketMock виконує два завдання: повертає заготовлену відповідь і перевіряє, що тестована система виконала виклик методу. Однак в даному випадку присутні два різних метода. Тест встановлює відповідь для методу hasFood(), але потім перевіряє виклик методу decreaseFood(). Таким чином взаємодія з заглушками не перевіряється, і порушення правила немає.

Тестовим двійник, який виконує роль мока і заглушки, все одно називається моком, а не заглушкою. Називається він так не тільки з причини

того, що потрібно вибрати одне ім'я, а й через те, що мок є більш важливим об'єктом, ніж заглушка.

2.7.5. Відношення моків та заглушок до запитів і команд

Поняття моків і заглушок тісно пов'язане з принципом поділу команд і запитів. Принцип поділу команди і запитів стверджує що кожен метод повинен бути або командою або запитом, але не двома елементами одночасно. Як показано на рисунку 2.9, команди це такі методи, які ведуть до побічних ефектів і не повертають значень. До побічних ефектів належать зміна стану об'єкта або зміна змісту файлу файлової системи і так далі. Запити є протилежністю описаного, вони позбавлені побічних ефектів і повертають значення.

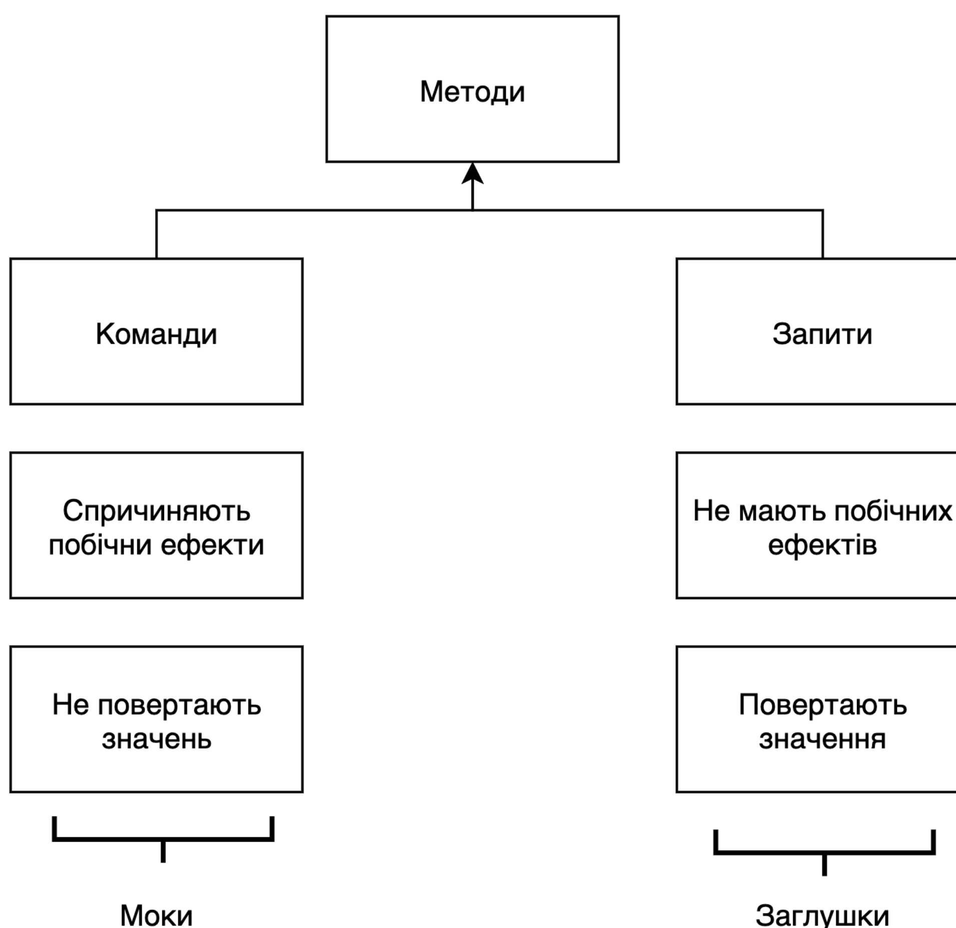

Рис. 2.9. Розділення моків та заглушок відповідно до принципу команд та запитів

Дотримуючись цього принципу метод, який призводить до побічного результату, повинен повертати тип даних void. І якщо метод повертає значення він повинен бути позбавлений побічних ефектів. Код, який дотримується такого чистого поділу є набагато легшим для читання. Глянувши на сигнатуру методу, можна сказати що він робить, не заглиблюючись при цьому в деталі реалізації.

Звичайно дотримання цього принципу не завжди є можливим. Завжди існують методи, в яких є сенс породити побічний ефект і повернути значення. Класичним прикладом може бути метод видалення елементу зі списку, list.remove(). Цей метод видаляє елемент зі списку, та якщо дія виконана успішно, повертає значення true. Але все ж таки це гарна ідея дотримуватися принципу розділення команд та запитів, коли це можливо.

Тестові двійники, які заміщають команди, є моками. Відповідно тестові двійники, які заміщають запити, є заглушками. У прикладі, зображеному на рисунку 2.10, sendData() є командою, побічна дія якої — відправка даних. Тестовий двійник, що замінює таку команду, є моком. З іншого боку, getPersonalName() є запитом, що повертає значення та не змінює стан локального сховища. Тому відповідний тестовий двійник називається заглушкою.

```
var mock = MockSender();
verify(mock.sendData("Key: Value")).called(1);

var stub = MockDataProvider();
when(stub.getPersonalName()).thenReturn("Ivan Siabro");
```

Рис. 2.10. Різниця між моками та заглушками

2.8. Спостережувальна поведінка та деталі реалізації

Для роз'яснення як моки впливають на крихкість тестів, потрібно спочатку пояснити, що взагалі призводить до крихкості тестів. У юніт тестів є чотири атрибута якості: захист від регресій, стійкість до рефайторингу, швидкий зворотній зв’язок та легкість у підтримці. Крихкість тестів відноситься до

стійкості до рефакторингу. Метрика стійкості до рефакторингу є досить важливою характеристикою, оскільки у даній метрики є дві сторони однієї медалі. Підвищення даної методики до ступеня, в якому тест все ще вважається модульний тестом і не переходить в категорію наскрізного тестування є хорошою дією. Наскрізне тестування, свою чергу, не дивлячись на те, що має високу стійкість до рефакторингу, в більшості випадків досить складне в підтримці.

Однією з головних причин, чому тести дають хибнопозитивні результати і при цьому не мають стійкість до рефакторингу, є те, що вони пов'язані з деталями реалізації коду. Для того щоб уникнути такого зв'язування, необхідно перевіряти кінцевий результат коду, його спостережувальну поведінку, і віддаляти тести від деталі реалізації настільки, наскільки це можливо. Тести повинні тільки фокусуватися на тому, що трапилося, але не на тому, як це трапилося.

2.8.1. Публічне API та спостережувальна поведінка

Увесь код, що підпадає під реліз, тобто знаходиться у фінальній кінцевій версії програмного забезпечення, можна розділити на два аспекти:

- публічне та приватне API;
- спостережувальна поведінка та деталі реалізації.

У цих двох аспектах категорії не перетинаються, так само як метод не може бути одночасно публічним і приватним. Схожим чином код є або деталями внутрішньої реалізації або частиною спостережуваної поведінки системи, але не тим і тим одночасно.

Прикладом даного механізму може бути використання слів public і private в мові програмування Java. За допомогою private можна помітити будь який член класу та сховати його від клієнтського коду, зробивши таким чином його приватним API даного класу. У мові програмування Dart використовується символ підкреслення для позначення приватного API.

Різниця між спостережуваною поведінкою і внутрішніми деталями реалізації є більш тонкою. Щоб код вважався спостережуваною поведінкою системи, він повинен виконувати одну з наступних двох дій:

- надавати операцію, яка допомагає клієнтській стороні досягти одну з його цілей. Операція – це метод, що виконує обчислення або призводить до стороннього ефекту, або це дві дії одночасно;
- надає стан, що допомагає клієнтській стороні досягти певної мети. Станом вважається стан системи.

Код, що не виконує вищеперелічених дій, називається деталями реалізації. Слід звернути увагу на те, що що код є спостережувальною поведінкою в залежності від того, хто є клієнтської стороною і які у неї цілі. Для того щоб бути частиною спостережуваної поведінки, у коду повинен бути прямий зв'язок як мінімум з однією такої метою. Термін "клієнт" може відноситися до різних речей в залежності від того, де знаходиться код. Типовим прикладом може бути клієнтський код з тієї ж кодової бази, зовнішня програма або користувальницький інтерфейс.

Публічне API системи має збігатися з її спостережуваною поведінкою і все її деталі реалізації повинні бути приховані від клієнтів. API, що задовольняє таким потребам, вважається правильно спроектованим, як зображено на рисунку 2.11.

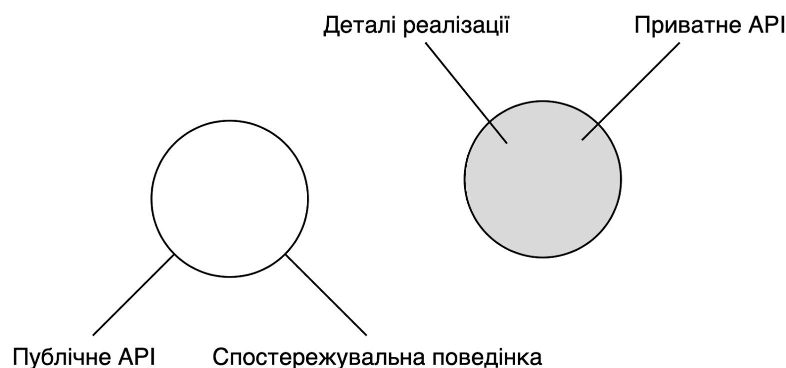

Рис. 2.11. Приклад правильно спроектованої системи

Але доволі часто публічне API системи виходить за межі її спостережувальної поведінки та надає доступ до деталей реалізації. Як зображено на рисунку 2.12, у такому разі деталі реалізації системи починають підпадати під публічне API.

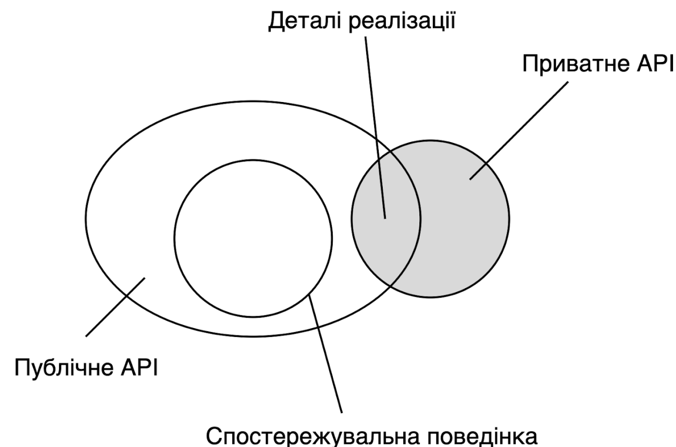

Рис. 2.12. Вихід публічного API за межі спостережувальної поведінки та витік деталей реалізації

2.8.2. Витік деталей реалізації

Прикладом витоку деталей реалізації може бути наступний код, зображений на рисунку 2.13.

У даному прикладі клієнтською стороною виступає ReportController клас, що користується класом Report у методі changeReportCaption(). Метою даного методу є зміна заголовку звіту. Причина, по якій клас Report є неправильно спроектованим, є наявність публічного методу trimCaptionIfNeed(), який є лише внутрішніми деталями реалізації зміни заголовку звіта. Властивість caption допомагає клієнту досягти певної мети – змінити заголовок звіту. В свою чергу, trimCaptionIfNeed() не має прямого зв’язку з кінцевою метою клієнта, а викликається клієнтом тільки для того, щоб задовольнити правило зміни заголовку звіта.

Для того, щоб класи вважалися правильно спроектованими, необхідно переписати їх, як зображено на рисунку 2.14.

```
class Report {

  String caption;

  String trimCaptionIfNeed(String caption){
    String trimmedCaption = caption.trim();
    if (trimmedCaption.length > 10){
      return trimmedCaption.substring(0, 10);
    }
    return trimmedCaption;
  }

}

class ReportController {
  void changeReportCaption(String reportKey,
String caption){
    Report report =
loadReportFromLocalStorage(reportKey);

    String trimmedCaption =
report.trimCaptionIfNeed(caption);
    report.caption = trimmedCaption;

    updateReportInLocalStorage(report);
  }
}
```

Рис. 2.13. Приклад витіку деталей реалізації

```
class Report {

  String _caption;
  set caption(String caption){
    _caption = trimCaptionIfNeed(caption);
  }

  String _trimCaptionIfNeed(String caption){
    String trimmedCaption = caption.trim();
    if (trimmedCaption.length > 10){
      return trimmedCaption.substring(0, 10);
    }
    return trimmedCaption;
  }

}

class ReportController {
  void changeReportCaption(String reportKey,
String caption){
    Report report =
loadReportFromLocalStorage(reportKey);

    report.caption = caption;

    updateReportInLocalStorage(report);
  }
}
```

Рис. 2.14. Версія класу Report з правльно зпроектованим API

У новому прикладі властивість caption стала приватною, оскільки вона не використовується клієнтським кодом. Замість цього з’явився публічний сетер, який робить виклик приватного методу _trimCaptionIfNeed() та змінює ім’я. У такому варіанті кінцева мета клієнта досягається лише однією операцією, що також зменшує можливу кількість помилок у клієнтському коді та зменшує кількість необхідних для тестування методів.

2.8.3. Внутрішні та зовнішні взаємодії систем

Так само, як метод _trimCaptionIfNeed() є деталями внутрішньої реалізації роботи класу Report, який не представляє цінності для клієнтського коду ReportController. Мета останнього – лише змінити заголовок у звіті. Під час створення модульного тесту необхідно перевіряти фінальну роботу, тобто змінений заголовок звіту, report.caption, але не роботу методу _trimCaptionIfNeed() або його виклик.

Припустимо, що під час збереження звіту потрібно також відправити його на HTTP сервер, як зображено на рисунку 2.15.

Сервер у даному випадку є зовнішньою системою та спостережуваною поведінкою, до якої призводить клас ReportController, та яка має значення для бізнесу. Ця логіка відбувається у методі _sendReportToServer() та є зовнішньою взаємодією. Для відправки використовується метод sendHttpCall() класу NetworkApi. Під час написання модульного тесту раціонально перевіряти, чи був успішним виклик до цього сервісу, оскільки відправка даних на сервер є частиною контракту взаємодії між системами.

Окрім відправки на сервер, є також збереження звіту у локальне сховище – метод _updateReportInLocalStorage(). Він виконує збереження даних за допомогою методу saveData() класу LocalStorage. По відношенню до класу ReportController ця дія також є зовнішньою, але дані, що зберігаються у локальному сховищі, доступні тільки лише для даного застосунку, або для класу

ReportController. Тому у рамках взаємодії різних систем, збереження у локальне сховище є лише внутрішніми деталями реалізації системи.

```
class ReportController(this._networkApi) {

  NetworkApi _networkApi;

  void changeReportCaption(String reportKey, String caption){
    Report report = loadReportFromLocalStorage(reportKey);

    report.caption = caption;

    _updateReportInLocalStorage(report);
    _sendReportToServer();
  }

  void _sendReportToServer(Report report){
    _networkApi.sendHttpCall("update_report", report.toJson())
  }

  void _updateReportInLocalStorage(Report report){
    _localStorage.saveData("report", report.toJson())
  }

}

class NetworkApi {
    sendHttpCall(String action, String data)
}

class LocalStorage {
    saveData(String table, String data)
}
```

Рис. 2.15. Клас ReportController з прикладом зовнішньої взаємодії

Доцільно перевірити, що після виклику методу changeReportCaption() був виклик класу NetworkApi.sendHttpCall() с параметрами “report” та “report.toJson()”, оскільки це є частиною спостережувальної поведінки, результатом якої буде новий запис про звіт на видаленому HTTP сервері, як зображено на рисунку 2.16.

```
var mockApi = MockNetworkApi()
var controller = ReportController(mockApi)
var report = loadReportById("123")
controller.changeReportCaption(report, "New Caption")
verify(mockApi.sendHttpCall("report", report.toJson)).called(1)
```

Рис. 2.16. Перевірка виклику мок об’єкта для зовнішньої системи

Навпаки, перевіряти виклик методу _updateReportInLocalStorate() не має сенсу, оскільки він не є частиною контракту взаємодій між системами та може бути замінене на іншу реалізацію в будь-який момент без шкоди для загальної роботи. При перевірці виклику даного методу модульний тест втратить стійкість до рефакторингу, оскільки при його проведенні внутрішні деталі реалізації можуть бути змінені.

2.9. Висновки до другого розділу

Тестування та контроль якості є невід'ємною складовою якісного програмного продукту. Є три типа тестування: модульне, інтеграційне та мануальне. Мануальне тестування, як і інтеграційне, здебільшого проводиться тестувальниками, в той час як модульне – програмістами. На жаль, необхідність модульного тестування інколи недооцінюється як розробниками, так і бізнесом. Але крім розуміння, що тестування важливе, потрібно також розуміти його кінцеву мету. Тестування потрібно не для того, щоб довести, що програмний продукт на сто відсотків працездатний та позбавлений помилок або навпаки довести наявність помилок, оскільки першу мету неможливо досягти, а друга мета досягається досить легко, та не є кінцевою метою бізнесу. Тестування потрібне для зниження кількості ризику, пов’язаного з випуском програмного забезпечення в обіг до сприйнятливого значення. При відношенні до тестування як до ментальної дисципліни та проектуванні програми з урахуванням її здатності до тестування призводить до створення такого програмного забезпечення, яке має меншу кількість помилок та більш легше у підтримці, оскільки деякі можливі недоліки відсікаються на стадії проетування.

Тестування мобільних додатків має свої особливості, за рахунок дуже швидкого розвитку мобільних платформ та створенню нових мобільних пристроїв. Великі компанії щонайменше раз на рік виводять на ринок нову версію флагманского пристрою, що має нові апаратні можливості. Крім флагманів за один рік виходить також багато середньо бюджетних та низько бюджетних пристрої, які також можуть мати певні нові властивості. Виробники мобільного платформного програмного забезпечення, Google та Apple, раз на рік створюють нову версію програмного забезпечення, яка містить новий функціонал та зміни у роботі старих API. Досягти ідеальної роботи застосунку на всіх пристроях неможливо, але слід брати до уваги пристрої цільової аудиторії, та досягати доброї роботи додатку саме на них. Оскільки купувати кожен новий пристрій дуже дорого, можна користуватися послугами оренди пристроїв, як фізичних так і хмарних сервісів.

Модульне тестування, крім запуску локально на машині програміста, може також виконуватися на виділеному сервері, за рахунок чого при кожній зміні коду, що підтримує систему контролю версій, тести автоматично запускаються. Таким чином створюється процес постійної підтримки та перевірки працеспроможності програмного коду. Але не всі модулі програмного продукту можна зручно та швидко покрити модульними тестами в первісному вигляді. Деякі взагалі не підлягають тестуванню у тестовому середовищі. Для вирішення таких проблем існують мок-фреймворки, або фреймворки підміни даних, які мають можливість замінити певні екземпляри класів на штучно створені. Такі екземпляри імітують поведінку реальних об’єктів таким чином, що система продовжує вважати, що працює з реальними. При цьому тестування коду, яке було б або дуже повільним, або взагалі неможливим, стає реальним та зручним. Прикладом цього може бути встановлення зв’язку із сервером, що може сильно сповільнювати тести або доступ до локального сховища, який неможливий у тестовому середовищі.

Мок об’єкти поділяються на дві великі групи – заглушки та моки. Заглушки використовуються для емуляції вхідних взаємодій із внутрішніми

залежностям системи для отримання даних вводу. Моки служать для емуляції зовнішніх взаємодій системи із зовнішніми залежностями та для перевірки таких взаємодій. При роботі з заглушками важливо не допускати помилок, у ході яких під час тесту перевіряються внутрішні деталі реалізації, але не кінцевий результат. Такий підхід призводить до крихкості тестів та робить їх нестійкими до рефакторингу, оскільки під час рефакторингу внутрішні деталі реалізації можуть бути змінені, в результаті чого доведеться переписувати тест. Також перевірка деталей реалізації замість кінцевого результату може призвести до наявності хибно позитивних тестів, результат проходження який буде позитивний при наявності помилок у системі. Тому важливо перевіряти кінцеві результати, а у випадку з перевіркою виклику методів – лише такі методи, що змінюють стан зовнішньої системи та виклик яких є контрактом у роботі між системами.

РОЗДІЛ 3
ПІДВИЩЕННЯ ШВИДКОСТІ ТЕСТУВАННЯ FLUTTER-ДОДАТКІВ ЗА ДОПОМОГОЮ ВИКОРИСТАННЯ ФРЕЙМВОРКА ПІДМІНИ ДАНИХ

3.1. Модульне тестування

Для першого прикладу було взято тест модуля, який встановлює зв'язок з мережею Інтернет для отримання даних про користувача та сворює для нього привітання. Тест складається з класів:

- UserPageController, клас високого рівня абстракції, з методом для вітання користувача;
- UserRepository, репозиторій для користувача, займається завантаженням даних;
- NetworkApi, клас для виконання HTTP запитів;
- UserNetworkModel, містить інформацію про користувача.

Схема класів наведена на рисунку 3.1.

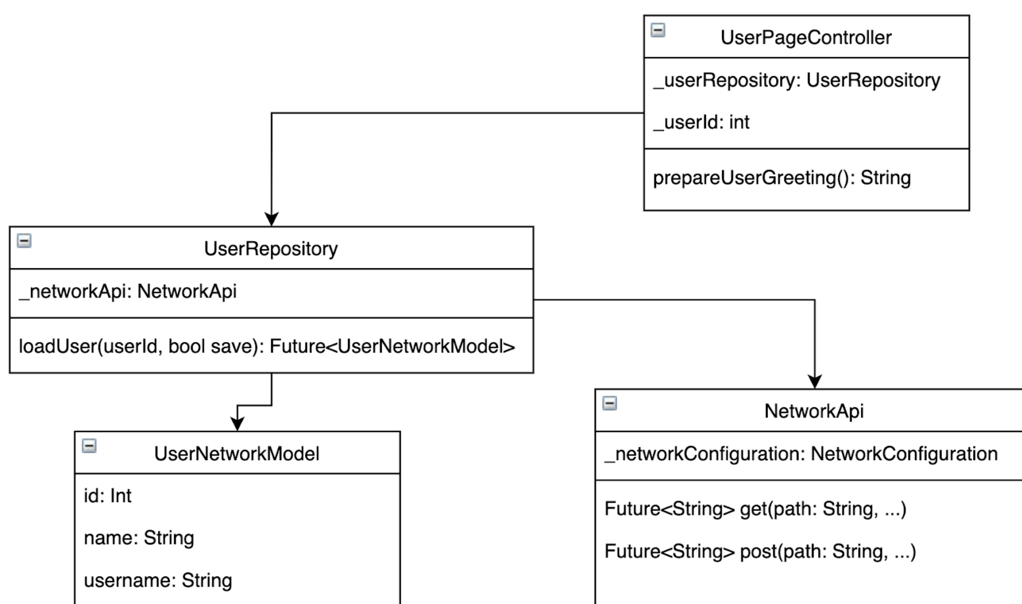

Рис. 3.1. Схема класів першого модульного тесту

Код тесту зображено на рисунку 3.2. Клас UserPageController отримує ідентифікатор користувача, для якого потрібно створити привітання. Далі йде виклик методу prepareUserGreeting(), який формує привітання. Привітання формується із фрази “Hello, dear $user!” та імені користувача. Після цього текст привітання перевіряється за допомогою методу expect(), який приймає актуальний та очікуваний результати. Якщо співпадають, тест проходить успішно, якщо ж ні – завершується з помилкою.

```
test('Check greeting only network', () async {
      var userPageController = UserPageController(1);
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Leanne Graham!");
    });
```

Рис. 3.2. Тест для перевірки методу prepareUserGreeting()

Метод prepareUserGreeting() в свою чергу звертається до об’єкту _userRepository, та викликає у нього loadUser() з раніше переданим ідентифікатором користувача. Метод loadUser() використовує об’єкт _networkApi для завантаження даних з мережі Інтернет, викликаючи метод get() – networkApi.get("users/$userId"), як зображено на рисунку 3.3.

```
Future<UserNetworkModel> loadUser(int userId) async {
    var result = await networkApi.get("users/$userId");
    UserNetworkModel userNetworkModel =
         UserNetworkModel.fromJson(jsonDecode(result));
    return userNetworkModel;
  }
```

Рис. 3.3. Метод завантаження користувача у репозиторії

Даний тест має зовнішню залежність, а саме звертання до веб вервісу https://jsonplaceholder.typicode.com/ для отримання даних про користувача. Швидкість виконання методу може бути змінною та залежить від швидкості Інтернет з’єднання та швидкості роботи самого сервісу. Нижче на рисунку 3.4

наведено діаграму залежності швидкості виконання тесту від типу Інтернет мережі.

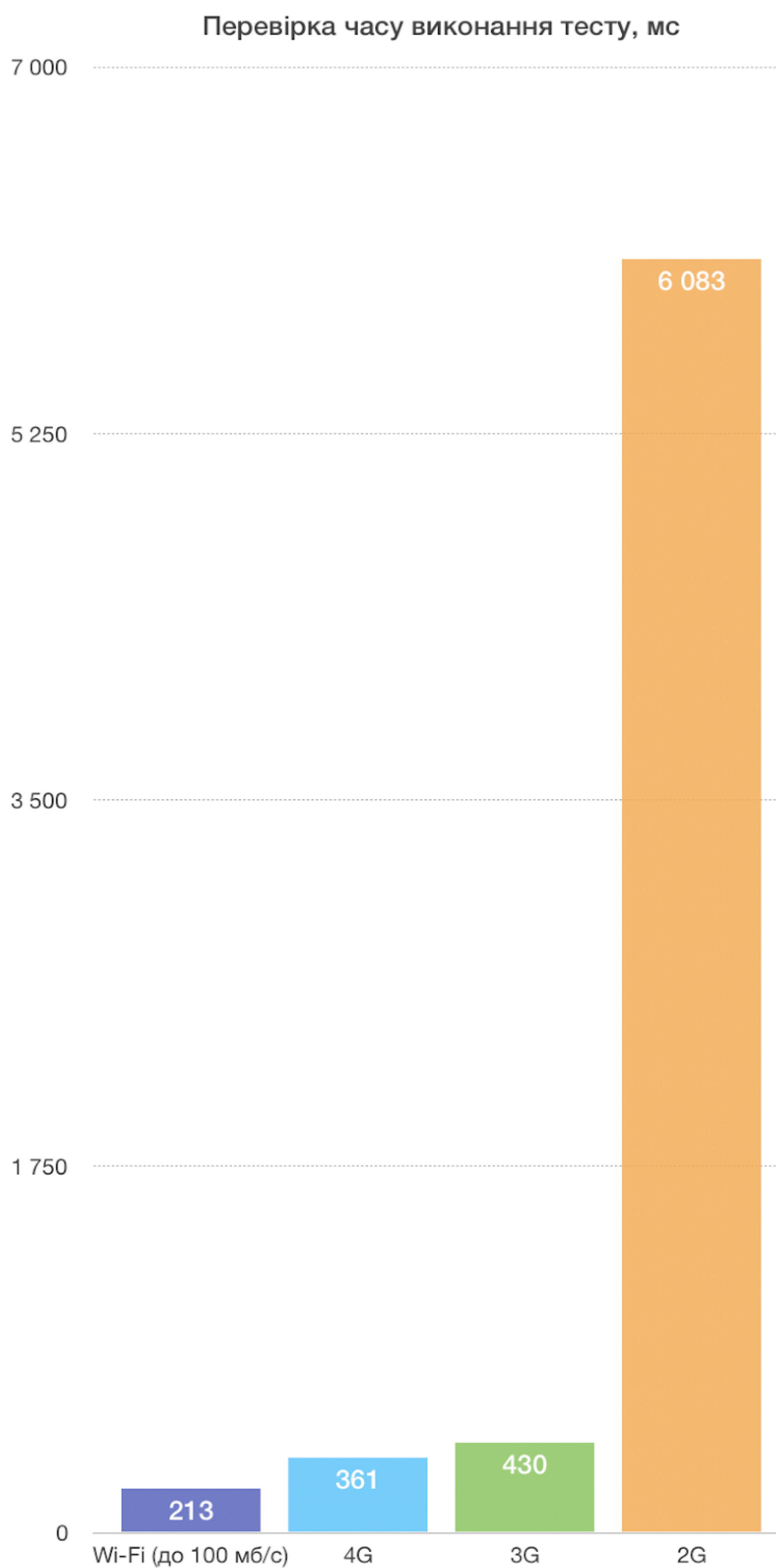

Рис. 3.4. Залежність затраченого часу на виконання тесту від типу мережі

При замірах швидкості було проведено по п’ять тестів для кожного типу мережі та було взято середнє значення. Таким чином, середня швидкість виконання тесту склала:

– 213 мс для Wi-Fi мережі;
– 361 мс для мережі 4G;
– 430 мс для мережі 3G;
– 6083 мс для мережі 2G.

Із наведених результатів видно, що чим менше максимальна швидкість мережі, тим довше виконується тест. Під час використання мережі 2G тривалість виконання тесту збільшується приблизно у 15 разів в порівнянні із 3G мережею, у 20 разів у порівнянні із 3G, та у 30 разів у порівнянні із Wi-Fi мережею. Якщо ж перші три результати є довгими, але все ж таки прийнятними, то останній результат катастрофічно сповільнює роботу тестів. Слід зауважити, що під час перевірки результатів з використанням 2G мережі, приблизно лише один із трьох разів був успішним, в той час як інші призводили до завершення тесту з помилкою через обрив зв’язку.

3.1.1. Підміна залежностей за допомогою mockito

Для вирішення даної проблеми та виконання тесту за константний період часу можна використовувати мок фреймворк mockito, який дозволяє створити мок-об’єкт, що буде імітувати роботу класу NetworkApi, та брати заздалегідь підготовлені дані замість звертання до стороннього сервісу через мережу Інтернет. На рисунку 3.5 наведено нову версію тесту.

Для створення мок-об’єкту інструментами mockito необхідно створити клас, що розширює клас Mock та реалізує інтерфейс NetworkApi – class MockNetworkApi extends Mock implements NetworkApi {}, та настроїти його за допомогою методів when() та thenAnswer().

```
class MockNetworkApi extends Mock implements NetworkApi {}

test('Check greeting only network with Mock', () async {
      var mockNetwork = MockNetworkApi();
      when(mockNetwork.get("users/1",
              queryParameters: anyNamed("queryParameters"),
              headers: anyNamed("headers"),
              config: anyNamed("config")))
          .thenAnswer((_) async =>
              """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

      var userPageController = UserPageController(1,
          userRepository: UserRepository(networkApi: mockNetwork));
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Leanne Graham!");
    });
```

Рис. 3.5. Версія тесту перевірки методу prepareUserGreeting() з використанням мок-об'єктів

Метод when() приймає виклик методу, який необхідно підмінювати. Під час створення конфігурації можно також вказати правила аналізу параметрів методу. Найпростіший варіант – це просто передати сире значення ("users/1"), на яке фреймворк буде орієнтуватися під час перевірки параметрів, як це зроблено з параметром path методу NetworkApi.get(). Також існують спеціальні об'єкти – матчери, які описують правила співставлення об'єктів. За допомогою функції argThat(matcher()) можна комбінувати різні правила співставлення параметрів, наприклад:

- startsWith("123") та endsWith("123"), параметр повинен починатися або завершуватися значенням "123";
- isNull/isNotNull, повинен дорівнювати або не дорівнювати значенню null;
- hasLength, параметр повинен мати властивість length.
- Окрім встановлених матчерів, можна створювати свої, розширюючи клас Matcher та реалізовуючи метод matches().
- Після конфігурування методу when() потрібно задати відповідь за допомогою одного із наступних параметрів:

– thenReturn, повертає передане у нього значення, використовується у більшості випадків;
– thenAnswer, повертає значення, що обгорнуто в об’єкт Future. Використовується для асинхронних операцій;
– thenThrow, ініціює виклик виключення. Потрібен для імітації помилок.

Після конфігурації об’єкт mockNetworkApi передається у UserRepository, що в свою чергу передається у UserPageController. Тепер під час виклику методу prepareUserString() клас UserPageController працює з мок об’єктом, який повертає результат за константний проміжок часу та не залежить від проблем у роботі мережі. Результати замірів швидкості виконання зображені на рисунку 3.6.

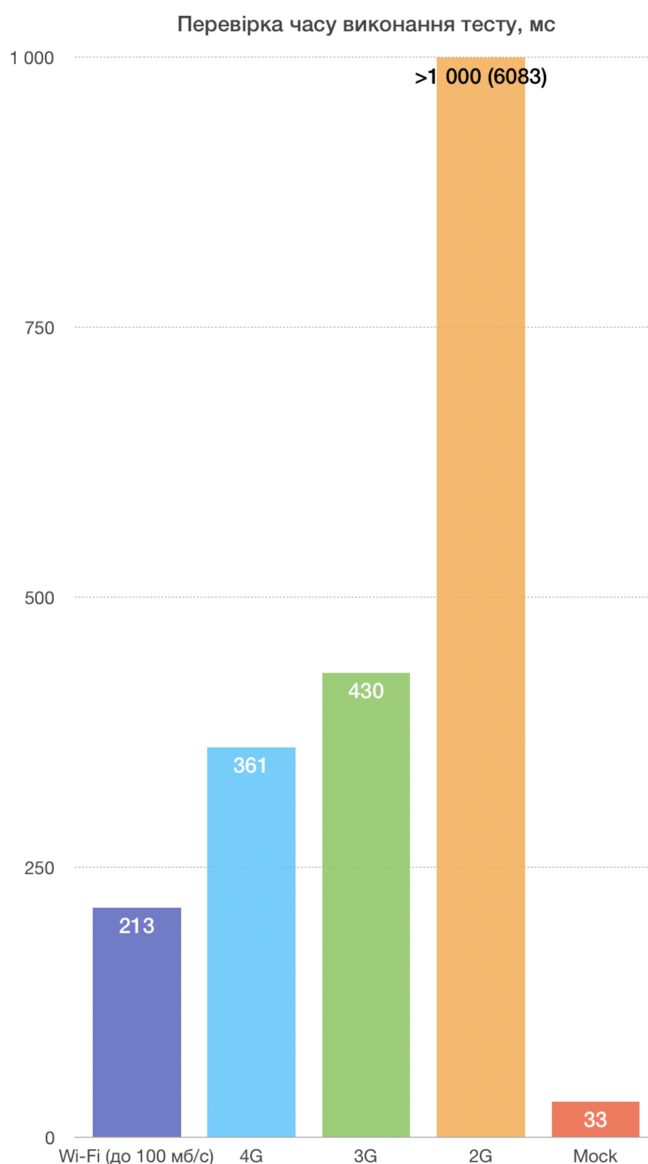

Рис 3.6. Час, витрачений на тест з використанням мок-об’єкту

Час, затрачений на виконання тесту із мок об’єктом склав лише 33 мілісекунди, що майже у 6,5 разів швидше, ніж у порівнянні з мережею Wi-Fi, в 11 разів у порівнянні з 4G, у 13 з 3G, та у 184 рази швидше, ніж у порівнянні з 2G мережею. Слід зазначити, що на рисунку 3.6 значення затраченого часу при 2G мережі зображено схематично та обмежене значенням у 1000 мс для того, щоб колонка, відповідна за роботу з мок об’єктом була помітна. При реальному значенні в 6083 мс, колонка зі значенням 33 мс відображається як маленька лінія.

3.1.2. Верифікація викликів та робота з локальним сховищем

Другий тест перевіряє коректність роботи методу оновлення даних користувача. Він включає в себе роботу з локальним сховищем, базою даних SQLite. Початковий код тесту зображено на рисунку 3.7.

```
test('Check if user update success', () async {
      var userPageController = UserPageController(1);
      await userPageController.updateUserName("Taras Shevchenko");
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Taras Shevchenko!");
    });
```

Рис. 3.7. Початковий код тесту для перевірки методу оновлення

Виклик методу updateUserName() повинен змінити ім’я користувача у системі та відправити відповідний запит на сервер. У даному випадку це є бізнес вимогою та спостережуваною поведінкою, тому і підлягає тестуванню. Код методу зображено на рисунку 3.8.

У тесті необхідно перевірити, що запрос на оновлення даних був успішно відправлений, та що під час нового виклику prepareUserGreeting() привітання включає оновлене ім’я користувача. Для збереження даних у локальне сховище метод updateUserName() викликає наступний метод – DBProvider.db.saveUser(user), який є лише деталями реалізації, а не кінцевою метою, тому верифікувати його виклик не доречно.

```
Future<int> updateUser(String newName, int userId) async {
    var user = await loadUserCacheOnly(userId);
    if (user == null){
      user = await loadUser(userId);
    }
    user.name = newName;
    await networkApi.post("users/$userId/update", body: user);
    var saveResult = await dbProvider.saveUser(user);
    return saveResult;
  }
```

Рис. 3.8. Код методу оновлення імені користувача

Правильно зробити перевірку результату методу prepareUserGreeting(), тому що якщо він включає ім'я оновленого користувача, це автоматично значить, що процес оновлення був успішним. І навпаки, успішний виклик методу збереження даних у локальне сховище – DBProvider.db.saveUser(user), не означає, що процес оновлення був успішним, оскільки метод оновлення користувача складається з декількох кроків, під час кожного з яких може виникнути помилка. Також в разі рефакторингу та зміни методу збереження користувача на іншу реалізацію, код тесту потрібно буде змінювати, якщо тест буду включати перевірку методу saveUser().

При запуску тесту в початковому вигляді, як зображено на рисунку 3.7 він завершується з помилкою MissingPluginException, оскільки сховище SQLite недоступно у тестовому середовищі. Для вирішення даної проблеми використовуємо мок-об'єкт класу DBProvider, та передаємо його у конструктор класу UserRepository. Клас UserRepository з новими властивостями та методом зображений на рисунку 3.9.

Тест, підготовлений до роботи у тестовому середовищі виглядає наступним чином, як зображено на рисунку 3.10.

Спочатку конфігурується мок об'єкт DbProvider, який імітує зберігання та отримання даних через за рахунок збереження їх в оперативній пам'яті. Це методи saveUser() та getUser(). Далі конфігурується мок для NetworkApi, який співпадає з моком першого тесту.

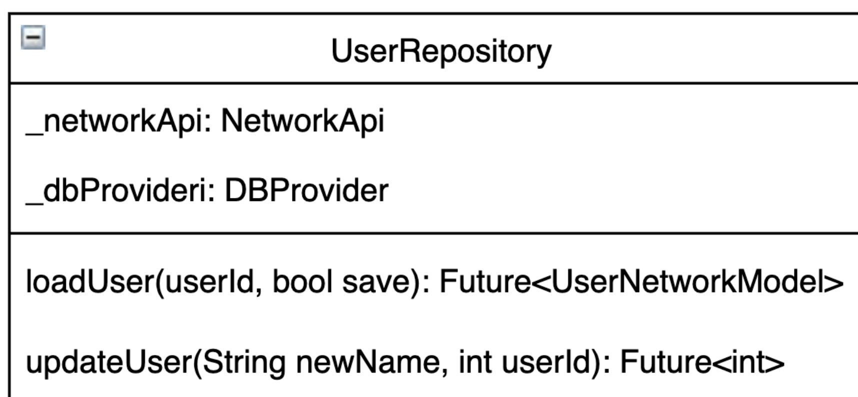

Рис. 3.9. Клас UserRepository з новими властивостями та методом

```
test('Check if user update success', () async {
      //встановлення мок-об'єкту для сховища
      var mockDbProvider = MockDbProvider();
      UserNetworkModel user;
      when(mockDbProvider.saveUser(any)).thenAnswer((realInvocation){
        user = realInvocation.positionalArguments[0];
        return Future.value(1);
      });
      when(mockDbProvider.getUser(1)).thenAnswer((realInvocation) =>
Future.value(user));

      //встановлення мок-об'єкту для роботи з мережею
      var mockNetworkApi = MockNetworkApi();
      when(mockNetworkApi.get("users/1",
          queryParameters: anyNamed("queryParameters"),
          headers: anyNamed("headers"),
          config: anyNamed("config")))
          .thenAnswer((_) async =>
      """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

      //ініціалізація userRepository мок об'єктами
      var userRepository = UserRepository(networkApi: mockNetworkApi, dbProvider:
mockDbProvider);

      var userPageController = UserPageController(1, userRepository: userRepository);
      var firstGreeting = await userPageController.prepareUserGreeting();
      expect(firstGreeting, "Hello, dear Leanne Graham!");

      await userPageController.updateUserName("Taras Shevchenko");
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Taras Shevchenko!");

      verify(mockNetworkApi.post("users/1/update", body: argThat(equals(user), named:
"body")));
    });
```

Рис. 3.10. Код тесту для перевірки методу оновлення с моками

Після конфігурації тест успішно проходить та доводить, що метод prepareUserGreeting() працює належним чином: спочатку він виводить

привітання для старого імені користувача – “Leanne Graham”, а потом для нового – “Taras Shevchenko”.

Але тест був би неповним без перевірки на останньому рядку, який верифікує, що метод оновлення даних був відправлений через мережу на сервер. За допомогою verify() проходить верифікація виклику методу NetworkApi.post() зі значенням параметра path рівному “users/1/update” та значенням параметра body рівному об’єкту user. Даний метод підлягає обов’язковій верифікації, тому що він є зовнішньою взаємодією та спостережуваною поведінкою.

3.2. Використання власного інструменту для роботи з мережею

Під час дослідження та використання фреймворку mockito було розроблено інструмент, що підмінює конфігурацію класу роботи з мережею на локальну. Даний інструмент дозволяє визначати логіку імітації роботи мережевих запитів для класу NetworkApi. Він має методи для відправки post та get запитів та поле з конфігурацією NetworkConfiguration, яка реалізує відправку запитів. Схема класів NetoworkConfirguration та NetworkApi зображена на рисунку 3.11.

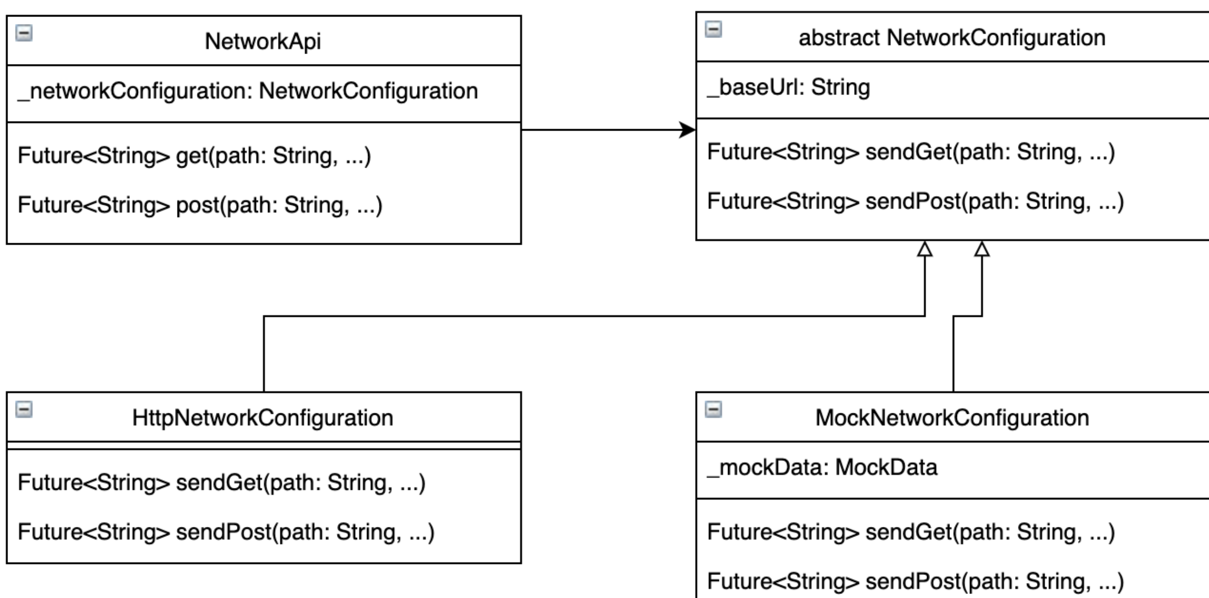

Рис. 3.11. Схема класів NetworkApi та NetworkConfiguration

NetworkConfiguration є абстрактним класом що містить базові методи sendGet() та sendPost() але не реалізує їх. HttpNetworkConfiguration розширює клас NetworkConfiguration та виконує HTTP запити за заданими параметрами. MockNetoworkConfiguration також розширює базовий клас NetworkConfiguration, але замість відправки реальних запитів на сервер бере дані з об’єкту MockData.

На даний момент для конфігурації передбачено методи – response() та stubs(). Метод response() є доволі простим у використанні та приймає шлях і відповідь, яку необхідно повернути. Приклад використання методу response() зображено на рисунку 3.12.

```
mock.response(
    path: "users/1",
    response: """{"id":1,"name":"Leanne Graham","username":"Bret"}""");
```

Рис. 3.12. Приклад використання методу response()

Метод stubs() дозволяє встановити більш широку конфігурацію. Він також приймає параметр path, шлях, для якого потрібно встановити заглушки та параметр groupScenarioBuilder, який приймає список сценаріїв. На даний момент існує три типи відповідей:

– success, успішна відповідь, приймає json рядок;
– error, відповідь з помилкою, приймає об’єкт HttpExceptionWithStatus;
– successObject, такий же як success, але замість рядка приймає об’єкт.

Приклад використання зображено на рисунку 3.13.

Кожен з трьох типів сценаріїв має обов’язковий параметр “when”. У ньому необхідно задати критерії спрацьовування сценарію, передавши функцію bool Function(Scenario), яка повертає булеве значення та приймає об’єкт даного сценарію для більш зручного конфігурування. Окрім аналізу зовнішніх даних, доступні також наступні параметри, що надсилаються з даним HTTP запитом:

– Map<String, dynamic> queries, набір query параметрів;

– Map<String, String> headers, набір header параметрів;

– String body, тіло запиту у форматі рядку;

– Map<String, dynamic> bodyJson, тіло запиту у форматі json.

```
mock.stubs(
    path: "users/1/update",
    groupScenarioBuilder: GroupScenarioBuilder.scenarios([
      Scenario.success(
          when: (it) =>
              it.bodyJson["id"] == 1 &&
              it.bodyJson["name"] == "Taras Shevchenko",
          response: """{"status":"success"}"""), // Scenario.success
      Scenario.error(
          when: (it) => it.header["Auth"] == "Auth Value",
          exception: HttpExceptionWithStatus(
              400, "CustomMessage bad request")), // HttpExceptionWithStatus, Scenario.error
      Scenario.successObject(
          when: (it) =>
              it.query["type"] == 1 &&
              it.query["action"] == "Update",
          object: UserNetworkModel(
              id: 3,
              name: "Ivan Siabro",
              username: "african_grey_parrot")) // UserNetworkModel, Scenario.successObject
    ])); // GroupScenarioBuilder.scenarios
```

Рис. 3.13. Використання сценаріїв success, error та successObject

Після конфігурації об’єкту mockBuilder він передається в клас MockNetworkConfiguration, який в свою чергу до класу NetworkApi, як зображено на рисунку 3.14.

```
MockBuilder builder = (MockData mock) {
  mock.response(
      path: "users/1",
      response: """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

  mock.stubs(
      path: "users/1/update",
      groupScenarioBuilder: GroupScenarioBuilder.scenarios(...)); // GroupScenarioBuilder.scenarios
};

var mockNetworkApi = NetworkApi(MockNetworkConfiguration(
    "https://anyurl.com",
    mockBuilder: builder)); // MockNetworkConfiguration, NetworkApi
```

Рис. 3.14. Ініціалізація об’єкту NetworkApi мок конфігурацією

Подальше використання об’єкту NetowrkApi не відрізняється від звичайного використання, оскільки система використовує спільний інтерфейс конфігурацій. При цьому окрім такої ж великої швидкості, як і з використанням фреймворку mockito, дане рішення має декларативний та більш лаконічний підхід, за рахунок чого процес написання та підтримки тестів стає зручнішим.

3.3. Висновки до третього розділу

В ході роботи над даним розділом було застосовано на практиці фреймворк для підміни даних mockito та розроблено інструмент для імітації відправки HTTP запитів NetworkApi разом з MockNetworkConfiguration. Було доведено, що в залежності від специфіки тестуємої системи використання мок-об’єктів дозволяє значно збільшити швидкість виконання модульних тестів, а в деяких випадках навіть зробити можливим перевірку таких модулів системи, які в початковому вигляді непридатні для роботи в тестовому середовищі. При перевірці модуля, що включає роботу з мережею, тривалість виконання тесту при використанні мок-об’єкту склала всього 33 мс, проти мінімального значення у 213 мс при швидкій мережі та 6083 мс при повільній. Приріст швидкості складає відповідно від 6,5 до 184 разів, в залежності від швидкості мережі. Крім підвищеної швидкості виконання, мок інструмент також дозволяє проводити верифікацію виклику методів з необхідними параметрами, що в свою чергу перевіряє коректність спостережувальної поведінки модуля, яка є частиною контракту взаємодії між системами.

Застосування розробленого у процесі написання роботи інструменту для підміни даних мережевих HTTP запитів дозволяє використовувати декларативний та лаконічний підхід опису мережевої логіки. Завдяки цьому процес написання мок об’єктів для роботи з мережею становиться легшим та швидшим. Декларативний підхід дозволяє зробити тести більш простими у підтримці та розвитку.

Фреймворки для підміни даних є потужним інструментом, що дозволяє збільшувати швидкість виконання, роботи можливим тестування раніше непридатних для тестування модулів та проводити верифікацію викликів. Але дані інструменти потрібно використовувати з розумом. При описані логіки мок об’єктів потрібно слідкувати за тим, щоб не створювати непослідовної поведінки у роботі системи, що може призвести до хибно позитивних тестів. Рекомендується дотримуватися правила, що якщо систему можна протестувати без мок об’єктів, або приріст швидкості від їх використання у даній системі не є значним, то у такому разі не потрібно їх використовувати. Попри це правило, кожна тестуєма система є унікальною, з унікальними потребами, логікою роботи та реалізацією цієї логики. Тому перед використанням мок-об’єктів у тестах важливо розуміти, які саме переваги будуть отримані та які проблеми будуть вирішені.

РОЗДІЛ 4
ЕКОНОМІКА

Одним з головних етапів при розробці ПЗ є визначення трудомісткості та розрахунок витрат на створення програмного продукту. В даному розділі наведено розрахунок витрат на розробку програми з паралельними обчисленнями.

4.1. Визначення трудомісткості розробки програмного забезпечення

Початкові дані:

1. передбачуване число операторів – 350;
2. коефіцієнт складності програми – 1,5;
3. коефіцієнт корекції програми в ході її розробки – 0,12;
4. годинна заробітна плата програміста, грн/год – 190;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,4;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 0,95;
7. вартість машино-години ЕОМ, грн/год – 40.

Нормування роботи в процесі створення ПЗ істотно ускладнюється в силу творчого характеру роботи програміста, тому трудомісткість розробки ПЗ може бути розрахована на основі системних моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин,}$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p),$$

де q - передбачуване число операторів;

c - коефіцієнт складності програми;

p - коефіцієнт корекції програми в ході її розробки.

$$Q = 350 * 1{,}5 * (1 + 0{,}12) = 588, \text{ людино-годин.}$$

Витрати праці на вивчення опису задачі tи визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_и = \frac{Q * B}{(75..85) * k} \text{ людино-годин}$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_и = \frac{588 * 1{,}4}{80 * 0{,}95} = \frac{823{,}2}{76} = 10{,}8 \text{ людино-годин}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20...25) * k} \text{ людино-годин}$$

$$t_a = \frac{588}{20 * 0{,}95} = 30{,}9 \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20…25) * k} \text{ людино-годин}$$

$$t_n = \frac{588}{24 * 0{,}95} = 25{,}7 \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5) * k} \text{ людино-годин}$$

$$t_{отл} = \frac{588}{5 * 0{,}95} = 123 \text{ людино-годин}$$

– за умови комплексного налагодження завдання:

$$t_{отл}^{k} = 1{,}5 * t_{отл}, \text{ людино-годин,}$$

$$t_{отл}^{k} = 1{,}5 * 123 = 184{,}5 \text{ людино-годин.}$$

Витрати праці на підготовку документації:

$$t_{\partial} = t_{\partial p} + t_{\partial o}$$, людино-годин,

де $t_{\partial p}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{\partial p} = \frac{Q}{15..20 * k} \text{людино} - \text{годин}$$

$$t_{\partial p} = \frac{588}{18 * 0,95} = 34 \text{ людино-годин}$$

$t_{\partial o}$ - трудомісткість редагування, печатки й оформлення документації:

$$t_{\partial o} = 0,75 * t_{\partial p}, \text{ людино-годин},$$

$$t_{\partial o} = 0,75 * 34 = 25,5, \text{ людино-годин},$$

$$t_{\partial} = 34 + 25,5 = 59,5, \text{ людино-годин}.$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 50 + 10,8 + 30,9 + 25,7 + 123 + 59,5 = 300, \text{ людино-годин}.$$

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми *Зз/п* і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{по} = З_{зп} + З_{мв}, \text{ грн.}$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп} = t * C_{пр}, \text{ грн,}$$

де t - загальна трудомісткість, людино-годин;

$C_{пр}$ - середня годинна заробітна плата програміста, грн/година.

$$З_{зп} = 300 * 190 = 57000, \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження програми:

$$З_{мв} = t_{отл} * C_{мч}, \text{ грн,}$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год,

$C_{мч}$ - вартість машино-години ЕОМ, грн/год,

$C_{мч}$ = 35, грн/год.

$$З_{мв} = 123 * 40 = 4920, \text{ грн.}$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП:

$$K_{по} = 57000 + 4920 = 61920, \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k * F_p}, \text{ міс ,}$$

де B_k - число виконавців (приймається 1),

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$$T = \frac{300}{1 * 176} = 1{,}7 \text{ міс.}$$

4.3. Маркетингові дослідження

В ході роботи над кваліфікаційною роботою було досліджено ефективність застосування фреймворка підміни даних для тестування мобільних Flutter додатків та розроблено власний вузькоспеціалізований інструмент, який дозволяє підмінювати дані для виконання мережевих HTTP запитів. Незважаючи на те, що розроблене рішення є вузьконаправленим, воно має доволі велику сферу застосування, оскільки його можна використовувати при розробці та тестуванні усіх Flutter-додатків, що передають та отримують дані із мережі. Нижче наведено список популярних застосунків, що написані за допомогою Flutter SDK та використовують передачу та отримання даних з мережі:

- Google Pay, сервіс від Google для оплати онлайн та в магазинах. Дозволяє додавати банківські картки до акаунту для подальшої оплати товарів та послуг як в магазинах так і онлайн;
- Realtor.com Real Estate: Homes for Sale and Rent - сервіс для послуг продажі та оренди житла, який надає можливості віртуальних турів по нерухомості, інтерактивного пошуку на мапі, калькулятора іпотеки та гнучкого підбору житла по заданим критеріям;
- Hamilton, музикальний застосунок для покупки білетів на концерти, доступу до музикальних відео та караоке;

- eBay Motors: Buy & Sell Car, сервіс для покупки/продажу легкових та грузових автомобілів з можливостями сканування VIN коду та автомобільних номерів, чату та інтеграція з платіжними системами;
- Insight Timer, медитаційний застосунок, надає доступ до медитацій та заспокійливих музикальних композицій, а також надає практичні рекомендації, як долати стрес та безсоння.

Наразі єдиним інструментом, що може бути застосованим для підміни даних у модульному тестуванні є офіційний фреймворк від розробників Flutter - mockito, який був розглянутий у практичній частині даної кваліфікаційної роботи. Він володіє широким набором можливостей підміни даних різноманітних модулів та класів, але при використанні його для підміни мережевих запитів синтаксис описання логіки може бути дещо громіздким. Рішення, розроблене у ході кваліфікаційної роботи має декларативний та лаконічний синтаксис опису підміни логіки відправки мережевих HTTP запитів, а також надає можливість виконувати реальні запити.

Потенційними користувачами даного розробленого рішення можуть бути середні і великі як аутсорс так продуктові компанії, що використовують Flutter SDK для розробки застосунків та впроваджують тести під час розробки. Акцент робиться на середніх та великих компаніях, тому що зазвичай маленькі компанії, що займаються розробкою програмного забезпечення не мають достатньо ресурсів для впровадження тестів. Такими ресурсами є:

- наявність додатково закладеного часу менеджером проекту на тестування програмного продукту. Додатковий час, окрім більш пізніх строків реалізації продукту означає збільшену матеріальну вартість розробки. Маленькі компанії знаходяться у жорсткій конкуренції, та зазвичай не можуть дозволити собі закладати додатковий час на тестування продукту через страх, що замовник відмовиться від їх послуг та обере іншу компанію, яка зробить те ж завдання швидше та дешевше;

- кваліфікаційні розробники, що мають досвід у розробці ефективних модульних тестів. Модульне тестування не є базовим навиком, що вимагається від програміста при прийомі на роботу. Початкові програмісти зазвичай не дуже цікавляться модульним тестуванням, та віддають перевагу вивченню нових технологій розробки та створення програм. Невеликі програмні продукти, життєвий цикл який складає не більше півроку, дійсно можуть бути вдало створенні без використання модульних тестів, але у середніх та крупних проектах, що розробляється роками та мають велику кількість спадкового коду, неможливо досягти прийнятної якості без використання модульних тестів;
- серверне обладнання, яке за допомогою спеціальних програм під’єднано до системи контролю версій та виконує запуск тестів при кожній зміні коду. Це дозволяє автоматизувати процес перевірки якості програмного продукту та заощадити час. Серверне обладнання зазвичай доволі дорого коштує, тому не всі маленькі компанії можуть його собі дозволити.

Виходячи з вищесказаних критиріїв, дане рішення для підміни відправки мережевих запитів може бути використане наступними компаніями без значних змін:

- Make it in Ukraine, Агенція з підбору персоналу та працевлаштування в Україні, пропонує пошук роботи для спеціалістів у галузі високих технологій, дизайну та маркетингу, та надає можливість віддаленої роботи з усього світу;
- EPAM, команди EPAM обслуговують клієнтів у понад 25 країнах Північної Америки, Європи, Азії та Австралії. EPAM є визнаним лідером ринку у багатьох категоріях серед провідних незалежних незалежних дослідницьких агентств і була однією з чотирьох технологічних компаній, які кожного року публікувались у списку Forbes 25 найбільш швидкозростаючих публічних технологічних компаній з 2013 року;

- i3 Engineering, інноваційна продуктова компанія в галузі розумного будинку та автоматизації. Займається впровадженням передових технологій у повсякденне життя та модулів, які контролюють пристрої розумного будинку;
- United Software, створює та керує спеціальними командами для бізнесу. Займається поєднанням зрілого бізнесу та динамічних стартапів з унікальними місцевими талантами;
- SSA Group, компанія, що займається аутсорсингом програмного забезпечення та веб-розробки, створена в 2007 році в Харкові, Україна. Має шість офісів, розташованих у Харкові, Києві, Дніпрі та Львові. Досвід компанії полягає у розробці веб-програм, мобільних і настільних додатків, а також дизайні інтерфейсу користувача / інтерфейсу користувача, QA та DevOps.
- Ciklum, це глобальна компанія з цифрових рішень, згідно за списком Fortune 500 та одна із швидкозростаючих організацій по всьому світу, була заснована у 2002 році. Має офіси у таких странах, як Україна, Польща, Білорусь, Пакистан, Іспанія.

4.4. Економічна ефективність

У цьому підрозділі наведено розрахунок економічного ефекту від впровадження на підприємстві розробленого програмного забезпечення, який зображено на таблиці 4.1.

За основу розрахунку очікуваного економічного ефекту від впровадження інструменту для розробки та тестування ПЗ виробництва програмного забезпечення взято зниження кількості часу, витрачаємого на запуск модульних тестів та їх підтримку і, як наслідок, економію коштів, що витрачаються під час розробки та підтримки програмного продукту.

Швидкість виконання модульних тестів, що використовують роботу з мережею та інструмент для підміни даних отримують приріст швидкості від 6,5

до 184 разів, що було досліджено в третьому розділі даної кваліфікаційної роботи. Але слід зауважити, що кінцевий приріст швидкості залежить від відсоткової частини тестів, що використовують модулі для роботи з мережею та їх загальної кількості. Чим більший процент таких тестів та чим більша їх загальна кількість, тим вигідніше буде запровадити даний інструмент.

Крупні підприємства з розробки мобільних компаній мають в середньому 300 модульних тестів у проекті, з яких приблизно 75 відсотків тестують модулі, що пов'язані з мережею, це приблизно 225 тестів. Більшість тестів включають в себе лише один запит у мережу, але деякі складаються з двох, трьох і навіть більшої кількості послідовних запитів. Середня кількість запитів у мережу на тест складає 1,25, середня кількість затраченого часу на комунікацію з мережею відповідно 1250 мс. Знаючи кількість тестів та зекономлений час можемо порахувати заощаджений час при одному запуску тестів, який складає 4 хвилини та 40 секунд. У середній команді знаходиться 10 розробників, кожен з який робить зміні у системі контролю версій приблизно по 3 рази на день. Знаючи кількість запуску тестів та зекономлений час на одному тесті, можемо порахувати кількість зекономленого у день, який складає 140 хвилин при 8 годинному робочому дні. При середній вартості машинного часу, що складає 40 грн/год, отримуємо економію 93 грн на день або 23343 грн на рік.

Окрім економії машинного часу отримуємо також економію робочого часу програмістів, оскільки тести також запускаються у ході розробки та до впровадження змін за допомогою системи контролю версій. З урахуванням середньої заробітної плати програміста, 200 грн/год, кількості запущених тестів на день, що включають роботу з мережею, 250, та зекономленого часу на один такий тест, що дорівнює 1250 мс, отримуємо, отримаємо 5 хвилин робочого часу на одного програміста на день, або 50 хвилин при середній кількості команди у 10 розробників. Грошова економія складає 166 грн на день або 41666 грн на рік.

Таблиця 4.1

Розрахунок чистих грошових надходжень від розробки ПЗ

Показники, грн	За роками						Усього за 5 років	Середнє за 5 років
	0	1	2	3	4	5		
Інвестиції на ПЗ	61920	-	-	-	-	-	61920	12384
Витрати до впровадження ПЗ	-	93201	93201	93201	93201	93201	466055	93201
- на машинний час	-	33466	33466	33466	33466	33466	167333	33466
- на робочий рас програмістів	-	59735	59735	59735	59735	59735	298675	59735
Витрати після впровадження ПЗ	-	28192	28192	28192	28192	28192	14960	28192
- на машинний час	-	10123	10123	10123	10123	10123	50615	10123
- на робочий рас програмістів		18069	18069	18069	18069	18069	90345	18069
- на придбання щорічних ліцензій	-	1200	1200	1200	1200	1200	6000	1200
Економія	-	65009	65009	65009	65009	65009	325045	65009
Амортизація	-	12384	12384	12384	12384	12384	61920	12384
Чисті грошові надходження	-	52625	52625	52625	52625	52625	263125	52625
Коефіцієнт дисконтування	-	0,87	0,75	0,65	0,57	0,49	-	-
Дисконтові грошові надходження	-	45783	39468	34206	29996	25786	175239	35047

Коефіцієнти економічної ефективності

Чиста поточна вартість доходів:

$$NPU = 175239 - 61920 = 113319 \text{ грн} > 0$$

Строк окупності:

$$T = 61920/35047 = 1{,}7 \text{ років}$$

Індекс прибутковості:

$$ІП = 175239/61920 = 2{,}8$$

Показник економічної ефективності (NPU - чиста поточна вартість доходів за роки реалізації впровадження (3-5 років) складе 113319 грн тобто відповідає умовам ефективності, тому що NPU > 0.

Середній строк окупності капвкладень складе 1,7 рік.

Індекс прибутковості за 5 років складе 2,8, тобто ІП > 1, проект варто прийняти.

Таким чином, показник ефективності свідчить про те, що дане впровадження є економічно вигідним.

ВИСНОВКИ

Кваліфікаційна робота присвячена питанню дослідження та збільшення швидкості виконання модульних тестів при розробці мобільних Flutter додатків за допомогою використання фреймворків для підміни даних.

У ході роботи було проаналізовано головні властивості та відмінності Flutter SDK від інших кросплатформенних рішень, розглянуто головні особливості специфіки тестування мобільних додатків, роль модульних тестів і критерії їх ефективного створення. Також досліджено ефективність використання фреймворків для підміни даних при роботі модульних тестів та створено власний інструмент для підміни даних мережевих HTTP запитів.

Використання фреймворків для підміни даних при розробці мобільних додатків на Flutter дозволяє суттєво збільшити швидкість виконання тестів, а в деяких випадках взагалі зробити спроможним для тестування такий модуль програми, який в початковому вигляді неможливо протестувати з причини відсутності необхідних класів та програмних залежностей у тестовому оточенні.

Розроблений в ході кваліфікаційної роботи інструмент для підміни даних мережевих HTTP запитів також дозволяє отримати приріст швидкості виконання модульних тестів від 6,5 до 184 разів в залежності від швидкості мережі, за рахунок відсутності затримки на комунікацію з мережею. Дане рішення може застосовуватись у середніх та крупних компаніях з розробки програмного забезпечення для економії часу та коштів при проходженні модульних тестів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dabit N. React Native in Action / N. Dabit – New York: Manning Publications, 2019. – 8 с.

2. Marco L. N. Beginning Flutter: A Hands On Guide to App Development / L. N. Marco – Birmingham: Wrox, 2019. – 77 с.

3. Windmill E. Flutter in Action / E. Windmill – New York: Manning Publications, 2019. – 25 с.

4. Weinberg G. Perfect Software: And Other Illusions about Testing / G. Weinberg – New York:Dorset House, 2008. – 94 с.

5. Whittaker J. Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design / J. Whittaker – Boston: Addison-Wesley Professional, 2009. – 172 с.

6. Homès B. Fundamentals of Software Testing / B. Homès – London:Wiley-ISTE, 2012. – 303 с.

7. Graham D. Software Test Automation: Effective Use of Test Execution Tools / D. Graham, M. Fewster – Boston: Addison-Wesley Professional, 2000. – 487 с.

8. Whittaker J. How to Break Software: A Practical Guide to Testing / J. Whittaker – New York:Pearson, 2002. – 74 с.

9. Riley T. Beautiful Testing: Leading Professionals Reveal How They Improve Software / T. Riley, A. Goucher – Sebastopol: O'Reilly Media, 2009. – 296 с.

10. Mili A. Software Testing: Concepts and Operations / A. Mili, F. Tchier – Hoboken: Wiley, 2015. – 84 с.

11. Meszaros G. xUnit Test Patterns: Refactoring Test Code / G. Meszaros – Boston: Addison-Wesley, 2007. – 614 с.

12. Kaczanowski T. Practical Unit Testing with JUnit and Mockito / T. Kaczanowski – Krakow: Tomasz Kaczanowski, 2013. – 306 с.

13. Spillner A. Software Testing Foundations: A Study Guide for the Certified Tester Exam / A. Spillner, T. Linz, H. Schaefer – Santa Barbara: Rocky Nook, 2011. – 53 с.

14. Beizer B. Software Testing Techniques, Second Edition / B. Beizer – New York: Van Nostrand Reinhold Co, 1990. – 19 с.

15. Craig R. Systematic Software Testing / R. Craig, S. Jaskiel – Norwood: Artech House, 2002. – 484 с.

16. Dustin E. Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance / E. Dustin, J. Rashka, J. Paul – Boston: Addison-Wesley Professional, 1999. – 289 с.

17. Kaner C. Lessons Learned in Software Testing: A Context-Driven Approach / C. Kaner, J. Bach, B. Pettichord – Hoboken: Wiley, 2001. – 183 с.

18. Crispin L. Agile Testing: A Practical Guide for Testers and Agile Teams / L. Crispin – Boston: Addison-Wesley Professional, 2008, – 91 с.

19. Graham D. Foundations of Software Testing: ISTQB Certification / D. Graham, E. Veenendaal, I. Evans, R. Black – Boston: Cengage Learning Emea, 2008. – 142 с.

20. Hendrickson E. Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing / E. Hendrickson – Eau Claire: Pragmatic Bookshelf, 2013. – 86 с.

21. Бек К. Экстремальное программирование: разработка через тестирование / К. Бек – Санкт-Петербург: Питер, 2017. – 197 с.

22. Hambling B. Software Testing: An ISTQB-ISEB Foundation Guide / B. Hambling, P. Morgan, A. Samaroo, G. Thompson, P. Williams – London: British Computer Society, 2010. – 22 с.

23. Мартин. Р. Чистый Код: Создание, анализ и рефакторинг / Р. Мартин – Санкт-Петербург: Питер, 2013. – 381 с.

24. Фримен Э. Head First. Паттерны проектирования. Обновленное юбилейное издание / Э. Фримен, Э. Робсон, К. Сьерра, Б. Бейтс – Санкт-Петербург: Питер, 2018. – 56 с.

25. Клейн Т. Дневник охотника за ошибками. Путешествие через джунгли проблем безопасности программного обеспечения / Т. Клейн – Москва: ДМК Пресс, 2013. – 210 с.

26. Уиттакер Д. Как Тестируют в Google / Д. Уиттакер, Д. Арбон, Д. Кароло – Санкт-Петербург: Питер, 2014. – 132 с.

27. Винниченко И. В. Автоматизация процессов тестирования / И. В. Винниченко – Санкт-Петербург: Питер, 2005. – 54 с.

28. Куликов С.С. Тестирование Программного Обеспечения / С.С. Куликов – Минск: Четыре Четверти, 2017. – 257 с.

29. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер – Санкт-Петербург: Питер, 2004. – 73 с.

30. Блэк Р. Ключевые процессы тестирования / Р. Блэк. – Москва: Лори, 2006. – 336 с.

31. Copeland L. A Practitioner's Guide to Software Test Design / L. Copeland – Norwood: Artech House, 2004. – 185 с.

32. Калбертсон Р. Быстрое тестирование / Р. Калбертсон, К. Браун – Москва: Вильямс, 2001. – 35 с.

33. Синицын С. В. Верификация программного обеспечения / С. В. Синицын, Н. Ю. Налютин – М.: БИНОМ, 2008. – 345 с

34. Boehm B. Software engineering economics / B. Boehm – NJ: Prentice Hall, 1981. – 639 с.

35. Boehm B. Software Cost Estimation with COCOMO II / B. Boehm – NJ: Prentice Hall, 2000. – 235 с.

36. Brooks F. The Mythical Man-Month: Essays on Software Engineering / F. Brooks – MA: Addison-Wesley, 1982. – 79 с.

37. DeMarco T. Peopleware: Productive Projects and Teams / T. DeMarco, T. Lister – NewYork: Dorset House Publishing, 1987. – 196 с.

38. Humphrey W. Managing the Software Process / W. Humphrey – MA: Addison-Wesley, 1990. – 38 с.

39. Humphrey W. Introduction to the Personal Software Process / W. Humphrey – MA: Addison-Wesley, 1997. – 463 c.

40. Humphrey W. Introduction to the Team Software Process / W. Humphrey – MA: Addison-Wesley, 2000. – 185 c.

41. Capers J. Programming Productivity / J. Capers – New York: McGraw-Hill, 1986. – 294 c.

42. Edward K. Software Testing in the Real World: Improving the Process / K. Edward – MA:Addison-Wesley, 1995. – 128 c.

43. Koomen T. Test Process Improvement / T. Koomen, P. Martin – MA:Addison-Wesley, 1999. – 237 c.

44. Lewis W. Software Testing and Continuous Quality Improvement / W. Lewis – FL: Auerbach, 2000. – 160 c.

45. Khorikov V. Unit Testing: Principles, Practices, and Patterns / V. Khorikov – New York: Manning Publications, 2019. – 105 c.

46. Paulk M. The Capability Maturity Model: Guidelines for Improving the Software Process / M. Paulk, C. Weber, B. Curtis – MA: Addison-Wesley, 1995. – 48 c.

47. Paulk M. Using the Software CMM with Small Projects and Small Organizations / M. Paulk – Hershey, PA: Idea Group Publishing, 1999. – 69 c.

48. Perry W. Effective Methods for Software Testing / W. Perry – New York: Wiley, 2000. – 52 c.

49. Perry W. Surviving the Top Ten Challenges of Software Testing / W. Perry, R. Randall – New York: Dorset House Publishing, 1997. – 189 c.

50. Pfleeger S. Software Engineering: Theory and Practice / S. Pfleeger – Saddle River, NJ: Prentice Hall, 2001. – 141 c.

51. Pressman R. Software Engineering: A Practitioner's Approach / R. Pressman. – New York: McGraw-Hill, 1997. – 100 c.

52. Schulmeyer G. Verification and Validation of Modern Software-Intensive Systems / G. Schulmeyer, M. Garth – Upper Saddle River, NJ: Prentice Hall, 2000. – 306 c.

53. Дейтел П. Android для разработчиков / П. Дейтел, Х. Дейтел, А. Уолд. – Санкт-Петербург: Питер, 2016. – 237 с.

54. Murphy M. The Busy Coder's Guide to Android Development / M. Murphy – New York: CommonsWare, 2012. – 583 с.

55. Ханг В. Оптимизация производительности приложений для iOS / В. Ханг – Москва: ДМК-Пресс, 2013, 106 с.

56. Макконнелл С. Совершенный Код / С. Макконнелл – Санкт-Петербург: БХВ-Петербург, 2016. – 734 с.

57. Майкл Ф. Эффективная работа с унаследованным кодом / Ф. Майкл. – Москва: Вильямс, 2016. – 275 с.

58. Гамма Э. Рефакторинг. Улучшение существующего кода / Э. Гамма, М. Фаулер, К. Бек – Москва: Символ Плюс, 2008. – 385 с.

59. Kaner C. Testing Computer Software / C. Kaner, J. Falk, H. Q. Nguyen – 2nd Edition. – Hoboken: Wiley, 1999. – 324 с.

60. Myers G. J. The Art of Software Testing / G. J. Myers, C. Sandler, T. Badgett – Hoboken: Wiley, 2011. – 173 с.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

Головний файл запуску модульних тестів, main.dart

```
main() {
  DBProvider mockDb(){
    var mockDbProvider = MockDbProvider();
    UserNetworkModel user;
    when(mockDbProvider.saveUser(any)).thenAnswer((realInvocation){
      user = realInvocation.positionalArguments[0];
      return Future.value(1);
    });
    when(mockDbProvider.getUser(1)).thenAnswer((realInvocation) => Future.value(user));
    return mockDbProvider;
  }

  group('Ivan Siabro tests', () {
    test('Check greeting only network no mock', () async {
      var userPageController = UserPageController(1);
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Leanne Graham!");
    });

    test('Check greeting only network with Mock', () async {
      var mockNetwork = MockNetworkApi();
      when(mockNetwork.get("users/1",
              queryParameters: anyNamed("queryParameters"),
              headers: anyNamed("headers"),
              config: anyNamed("config")))
          .thenAnswer((_) async =>
              """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

      var userPageController = UserPageController(1,
          userRepository: UserRepository(networkApi: mockNetwork));
      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Leanne Graham!");
    });

    test('Check if user update success', () async {
      var mockDbProvider = MockDbProvider();
      UserNetworkModel user;
      when(mockDbProvider.saveUser(any)).thenAnswer((realInvocation){
        user = realInvocation.positionalArguments[0];
        return Future.value(1);
      });
      when(mockDbProvider.getUser(1)).thenAnswer((realInvocation) => Future.value(user));

      var mockNetworkApi = MockNetworkApi();
      when(mockNetworkApi.get("users/1",
          queryParameters: anyNamed("queryParameters"),
          headers: anyNamed("headers"),
          config: anyNamed("config")))
          .thenAnswer((_) async =>
      """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

      var userRepository = UserRepository(networkApi: mockNetworkApi, dbProvider:
mockDbProvider);

      var userPageController = UserPageController(1, userRepository: userRepository);
      var firstGreeting = await userPageController.prepareUserGreeting();
      expect(firstGreeting, "Hello, dear Leanne Graham!");

      await userPageController.updateUserName("Taras Shevchenko");
      var res = await userPageController.prepareUserGreeting();
```

```
      expect(res, "Hello, dear Taras Shevchenko!");

      verify(mockNetworkApi.post("users/1/update", body: argThat(equals(user), named:
"body")));
    });

    test('Check greeting only network with Mock own', () async {
      MockBuilder builder = (MockData mock) {
        mock.response(
            path: "users/1",
            response: """{"id":1,"name":"Leanne Graham","username":"Bret"}""");

        mock.stubs(
            path: "users/1/update",
            groupScenarioBuilder: GroupScenarioBuilder.scenarios([
              Scenario.success(
                  when: (it) =>
                      it.bodyJson["id"] == 1 &&
                      it.bodyJson["name"] == "Taras Shevchenko",
                  response: """{"status":"success"}"""),
              Scenario.error(
                  when: (it) => it.headers["Auth"] == "Auth Value",
                  exception: HttpExceptionWithStatus(
                      400, "CustomMessage bad request")),
              Scenario.successObject(
                  when: (it) =>
                      it.queries["type"] == 1 &&
                      it.queries["action"] == "Update",
                  object: UserNetworkModel(
                      id: 3,
                      name: "Ivan Siabro",
                      username: "african_grey_parrot"))
            ]));
      };

      var mockNetworkApi = NetworkApi(MockNetworkConfiguration(
          "https://anyurl.com",
          mockBuilder: builder));

      var userRepository = UserRepository(networkApi: mockNetworkApi, dbProvider: mockDb());

      var userPageController = UserPageController(1, userRepository: userRepository);
      var firstGreeting = await userPageController.prepareUserGreeting();
      expect(firstGreeting, "Hello, dear Leanne Graham!");

      await userPageController.updateUserName("Taras Shevchenko");

      var res = await userPageController.prepareUserGreeting();
      expect(res, "Hello, dear Taras Shevchenko!");
    });
  });
}

class MockNetworkApi extends Mock implements NetworkApi {}

class MockPrinter extends Mock implements Printer {}

class MockDbProvider extends Mock implements DBProvider {}
```

Клас комунікації з мережею NetworkApi

```
class NetworkApi {
  NetworkConfiguration networkConfiguration;

  NetworkApi(this.networkConfiguration);

  Future<String> get(String path,
      {Map<String, String> queryParameters,
```

```
      Map<String, String> headers,
      Function(NetworkConfiguration networkConfiguration) config}) async {
    return await networkConfiguration.sendGet(
        path, queryParameters, headers, config);
  }

  Future<String> post(String path,
      {Map<String, String> queryParameters,
      Map<String, String> queryHeaders,
      dynamic body,
      Function(NetworkConfiguration networkConfiguration) config}) async {
    if (body is String) {
      body = body as String;
    } else {
      body = jsonEncode(body);
    }
    return await networkConfiguration.sendPost(
        path, queryParameters, queryHeaders, body as String, config);
  }
}
```

Клас MockNetworkConfiguration для підміни даних комунікації з мережею

```
typedef MockBuilder = Function(MockData mockData);

class MockNetworkConfiguration extends NetworkConfiguration {
  MockNetworkConfiguration(String baseUrl,
      {Map<String, String> baseHeaders,
      InInterceptor inInterceptor,
      OutInterceptor outInterceptor,
      MockBuilder mockBuilder})
      : super(baseUrl,
            baseHeaders: baseHeaders,
            inInterceptor: inInterceptor,
            outInterceptor: outInterceptor){
    mockBuilder(mockData);
  }

  MockData mockData = MockData();

  @override
  Future<String> sendGet(
      String path,
      Map<String, String> queryParameters,
      Map<String, String> queryHeaders,
      Function(NetworkConfiguration networkConfiguration) config) async {
    return getResponseFromMap(ParamsHolder(
      path: path,
      headers: queryHeaders,
      queries: queryParameters,
    ));
  }

  @override
  Future<String> sendPost(
      String path,
      Map<String, String> queryParameters,
      Map<String, String> queryHeaders,
      String body,
      Function(NetworkConfiguration networkConfiguration) config) async {
    return getResponseFromMap(ParamsHolder(
      path: path,
      headers: queryHeaders,
      queries: queryParameters,
      jsonBody: body
    ));
  }

  Future<String> getResponseFromMap(ParamsHolder requestParams){
```

```
    mockData.setRequestParams(requestParams);

    Scenario scenario = mockData._getResponse(requestParams);
    String scenarioName;
    if (scenario?._scenarioName.isNullOrEmpty)
      scenarioName = "";
    else
      scenarioName = "scenario: ${scenario?._scenarioName}";
    var messageRes = scenarioName + "req: $requestParams";
    print("");
    print(messageRes);



    try {
      if (scenario == null){
        String message = " Path: /${requestParams.path} -> scenario not defined for params
$requestParams";
        return Future.error(message);
      } else {
        if (scenario._exception != null){
          return Future.error(scenario._exception);
        } else {
          Future<String> futureResult;
          if (scenario._requestCode == 200){
            String responseBody = scenario._response;
            futureResult = Future.sync(() => responseBody);
          } else {
            Exception httpException = HttpExceptionWithStatus(
              scenario._requestCode,
              scenario._response ?? "response without body");
            futureResult = Future.error(httpException);
          }
          return futureResult;
        }
      }
    } catch (e){
      return Future.error(e);
    }
  }

  @override
  NetworkConfiguration copy() {
    var config = MockNetworkConfiguration(
        baseUrl,
        baseHeaders: baseHeaders,
        inInterceptor: inInterceptor,
        outInterceptor: outInterceptor);
    config.mockData = mockData;
    return config;
  }
}

class HttpExceptionWithStatus extends HttpException {
  const HttpExceptionWithStatus(this.statusCode, String message, {Uri uri})
      : super(message, uri: uri);
  final int statusCode;
}

class MockData {

  Map<String, List<Scenario>> _stubsData = Map();

 void setRequestParams(ParamsHolder paramsHolder){
   if (paramsHolder != null){
     _stubsData[paramsHolder.path]?.forEach((e) => e._params = paramsHolder);
   }
 }

 Scenario _getResponse(ParamsHolder key){
   var stubs = _stubsData[key.path];
   if (stubs == null) return null;
   if (stubs.length == 1 && stubs[0]._predicate == null) return stubs[0];
   return stubs.firstWhere((element) {
     if (element._predicate == null)
       throw Exception(" you must write When expression");
     return element._predicate(element);
```

```
    });
  }

   void response({@required String path, @required String response}) {
    String key;
    if (path.startsWith("/")){
      key = path.substring(1);
    } else {
      key = path;
    }
    List<Scenario> stubs = _stubsData[key];
    List<Scenario> newStubs = [Scenario.response(response)];
    if (stubs == null){
      _stubsData[key] = newStubs;
    } else {
      stubs.addAll(newStubs);
    }
   }

   void stubs({@required String path, @required GroupScenarioBuilder groupScenarioBuilder}){
    String key;
    if (path.startsWith("/"))
      key = path.substring(1);
    else
      key = path;

    List<Scenario> mockData = groupScenarioBuilder.stubs;
    List<Scenario> currentStubs = _stubsData[key];
    if (currentStubs == null){
      _stubsData[key] = mockData;
    } else {
      currentStubs.addAll(mockData);
    }
   }
  }

  class GroupScenarioBuilder {
    List<Scenario> stubs = [];

    void scenario(Scenario scenario){
      stubs.add(scenario);
    }

    GroupScenarioBuilder();

    GroupScenarioBuilder.scenarios(this.stubs);

  }

  class Scenario {
    String _scenarioName;
    String _response;
    int _responseCode = 200;
    ParamsHolder _params;
    Map<String, dynamic> get queries => _params?.queries ?? Map();
    Map<String, String> get headers => _params?.headers ?? Map();
    String get body => _params?.jsonBody ?? "";
    Map<String, dynamic> get bodyJson =>
        _params?.jsonBody != null ? jsonDecode(_params.jsonBody) : Map();

    Exception _exception;
    int _delay;
    Predicate _predicate;
    Action _action;
    int _requestCode;

    Scenario.response(this._response){
      _requestCode = _responseCode;
    }

    Scenario(){
      _requestCode = _responseCode;
    }
```

```
  Scenario.success({@required Predicate when, @required String response}){
    this._predicate = when;
    this._response = response;
    if (response != null && response.isNotEmpty){
      _requestCode = _responseCode;
    }
  }

  Scenario.successObject({@required Predicate when, @required dynamic object}){
    this._predicate = when;
    this._response = jsonEncode(object);
    if (_response != null && _response.isNotEmpty){
      _requestCode = _responseCode;
    }
  }

  Scenario.error({@required Predicate when, @required Exception exception}){
    this._predicate = when;
    this._exception = exception;
  }

  void when(Predicate predicate){
    this._predicate = predicate;
  }

  void setResponse(String response){
    this._response = response;
  }

  void setAction(Action action){
    if (_predicate == null) _predicate = (_) => true;
    this._action = action;
  }

  void delayMillis(int millis){
    this._delay = millis;
  }

  void throwException(Exception exception){
    this._exception = exception;
  }

}

typedef Predicate = bool Function(Scenario);
typedef Action = String Function();


class ParamsHolder {
  String path;
  Map<String, String> headers;
  Map<String, String> queries;
  String jsonBody;

  ParamsHolder({this.path, this.headers, this.queries, this.jsonBody});

  @override
  String toString() {
    return 'ParamsHolder{path: $path, headers: $headers, queries: $queries, jsonBody:
$jsonBody}';
  }
}
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

на тему: «Розробка програмного забезпечення з метою дослідження ефективності фреймворка підміни даних для тестування Flutter додатків.»

студента групи 121м-19-1 Сябро Івана Володимировича

Керівник економічного розділу

доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Кваліфікаційна_робота_Сябро.doc	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Кваліфікаційна_робота_Сябро.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми
Презентація	
Презентація_Сябро.ppt	Презентація до кваліфікаційної роботи