Міністерство освіти і науки України
Харківський національний університет
радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Метод порівняння архітектур для Андроід на прикладі Viper та Mvp
(тема)

Виконав: студент 2 курсу, групи ІПЗм-17-1
спеціальності 121 - Інженерія програмного забезпечення
(код і повна назва спеціальності)
спеціалізації Інженерія програмного забезпечення
(повна назва спеціалізації)

Гуменюк В.В.
(прізвище, ініціали)
Керівник проф. Дудар З.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Дудар З.В.
(підпис) (прізвище, ініціали)

2019 р.

Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
Кафедра _____ Програмної інженерії _____
Рівень вищої освіти_____другий (магістерський) _____
Спеціальність _____121– Інженерія програмного забезпечення_____
(код і повна назва)
Спеціалізація ___Інженерія програмного забезпечення____
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри __Дудар З.В.
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові_____*Гуменюку Владиславу Володимировичу*_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту)*_____Метод порівняння архітектур для Андроід на прикладі Viper та Mvp*_____
затверджена наказом по університету від ___квітня__ 2019 р. № Ст
2. Термін подання студентом роботи до екзаменаційної комісії _20_ _червня_ 2019 р.
3. Вихідні дані до роботи *Дослідити існуючі архітектури програмних додатків та зокрема мобільних додатків, дослідити існуючі методи порівняння архітектур програмних додатків. Розробити новий метод порівняння мобільних архітектур. Провести за розробленим методом порівняння двох архітектур мобільних додатків для Android: MVP та VIPER.*
4. Зміст пояснювальної записки (перелік питань, що потрібно розробити) *мета роботи, аналіз предметної галузі, огляд наукових методологій та підходів, що використовувались, опис проектування ПЗ, план тестування ПЗ та аналіз дослідної експлуатації. Додатки: а) таблиці з результатами досліджень, б) таблиці з результатами тестування, в) слайди презентації, г) електронні матеріали до проекту на CD.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) *зображення схем побудови мобільних додатків існуючих архітектур, зображення та таблиці, що відображають результати порівняння двох архітектур мобільних додатків для Android: MVP та VIPER.*

КАЛЕНДАРНИЙ ПЛАН

	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
	Об'єктний аналіз поставленої задачі	25-02-2019	виконано
	Аналіз предметної галузі	12-03-2019	виконано
	Аналіз існуючих рішень	15-04-2019	виконано
	Дослідження існуючих методів	28-05-2019	виконано
	Пошук та вирішення недоліків алгоритмів та систем	15-05-2019	виконано
	Підготовка пояснювальної записки.	20-06-2019	виконано
	Підготовка презентації та доповіді	01-06-2019	виконано
	Попередній захист	06-06-2019	виконано
	Нормоконтроль, рецензування		виконано
0	Занесення диплома в електронний архів		виконано
1	Допуск до захисту у зав. кафедри		виконано

Дата видачі завдання «22» січня 2019 р.

Студент _____
(підпис)

Керівник роботи _____ ___проф. Дудар З.В.__
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 80 с., 8 рис., 7 табл., 3 додатки, 20 джерел.

АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МОБІЛЬНА РОЗРОБКА, ANDROID, MVP, VIPER, ПОРІВНЯННЯ АРХІТЕКТУРИ.

Об’єктом дослідження є архітектури мобільних додатків Android, основні характеристики архітектур та методи їх порівняння.

Метою роботи є розробка методу порівняння архітектур проектів розробки для мобільної платформи Android та порівняння двох архітектур VIPER та MVP.

Методи розробки базуються на систематичному вивченню літератури, мові програмування Java, Kotlin, модульному та інтеграційному тестуванні.

У результаті роботи створено метод порівняння архітектур та застосовано його для порівняння архітектур Android застосувань VIPER та MVP.

SOFTWARE ARCHITECTURES, MOBILE DEVELOPMENT, ANDROID, MVP, VIPER, ARCHITECTURE COMPARISON

The object of research is the architecture of Android mobile applications, the main characteristics of architectures and the methods of comparing them.

The aim of the work is to develop a method for comparing design architectures for the Android mobile platform and to compare the two VIPER and MVP architectures.

Methods of development are based on systematic study of literature, Java programming language, Kotlin, modular and integration testing.

Results - a method for comparing architectures was created and applied to compare the Android architecture of the VIPER and MVP applications.

ЗМІСТ

ВСТУП

Кожний програмний додаток будується з використанням певної архітектури. Різні архітектури пропонують різні переваги, але мають різні слабкості. Вибір архітектури програмних додатків часто залежить від конкретних вимог проекту та властивостей його середовища. Програмні архітектури зазвичай мають різні властивості, щоб відповідати різним потребам. Деякі архітектури показують кращу продуктивність на великих проектах, але вимагають набагато більше часу для реалізації. Інші архітектури забезпечують велику універсальність у регулюванні витрат на структуру коду, але важче покрити тестами.

Дизайн програмної архітектури - це область знань, яка постійно розвивалася протягом останніх десяти років для поліпшення якості програмних систем і пошуку архітектурних рішень для нових завдань [1]. У майбутньому архітектурні знання будуть ще більш важливими, оскільки нові системи потребують нових підходів до розвитку. Постійно впроваджуються нові програмні архітектури. Ці архітектури призначені для вирішення проблем існуючих архітектурних рішень. Але розуміння переваг і недоліків нової архітектури в порівнянні з більш консервативними слід оцінювати науковими дослідженнями. Порівняння двох програмних архітектур вимагає винайдення методу. Архітектура зазвичай описується набором властивостей, таких як змінюваність або продуктивність. Методи, які призначені для порівняння програмних архітектур, зазвичай покладаються на порівняння цих властивостей. Однак просто порівняти властивості недостатньо. Результати оцінки властивостей повинні бути проаналізовані та пояснені таким чином, що можуть бути використані для прийняття рішення про вибір архітектури для конкретного проекту.

Боб Мартін описав основні принципи чистої архітектури в своїй книзі «Чистий код» [2]. Архітектура повинна розділяти шари для підвищення легкості підтримки і дотримуватися принципів SOLID. SOLID - це абревіатура імен принципів. Це принцип принципу єдиної відповідальності, принцип відкритого /

закритого принципу, принцип заміщення Лісков, принцип сегрегації інтерфейсу, принцип інверсії залежності. Основна мета цих принципів полягає в тому, щоб зробити програмне забезпечення більш зрозумілим, легким у підтримуванні і гнучким.

Операційна система Android має складну структуру. У поєднанні зі специфікою того, як програми повинні бути побудовані, він додає деякі обмеження для архітектури програмних додатків.

Операційна система Android побудована на декількох основних компонентах і шарах. Верхній шар - прикладний шар. Всі програми виконуються на цьому рівні. Щоб отримати доступ до функцій операційної системи, програми використовують API шару Application Framework. Структура додатків відповідає за надання даних пристроїв і систем через API до додатків. Він збирає дані з різних датчиків і обладнання. Для цього він використовує системні бібліотеки та середовище виконання програми Android.

Системні бібліотеки Android і середовище виконання програми Android Runtime знаходяться на одному рівні і відповідають за надання основних функцій. Ці бібліотеки забезпечують доступ до апаратних засобів, наприклад, датчиків світла або акселерометра. Вони надають класи, які дозволяють керувати засобами масової інформації, наприклад відтворення аудіо чи відео. Вони надають доступ до інтерфейсів для відображення тексту на екрані, малювання графіки. Найбільш базовий шар - ядро Linux. Це ядро операційної системи. Він містить драйвери апаратних компонентів і працює з ними.

Операційна система Android є найпопулярнішою платформою для мобільних розробників, яка на сьогодні становить близько 80% ринку [3]. Отже, дослідження в цій галузі розробки програмного забезпечення будуть корисними для багатьох розробників. На Play Market постійно публікуються нові програми для Android.

Програми для Android зараз розробляються з використанням архітектури MVP. Але є нова архітектура VIPER, що йде від розробки iOS. Вона адаптується для розробки Android. Обидві ці архітектури використовуються для виконання

рекомендацій Боб Мартін для створення чистої архітектури для мобільних проектів. Боб Мартін представив розділення шарів для архітектури та оголосив основні принципи та обов'язки для кожного шару. Нижче наведено шари:

— рівень доступу до даних. Класи в цьому шарі відповідають за зберігання і отримання об'єктів предметної області;

— шар бізнес-логіки. Класи в цьому шарі відповідають за будь-які логічні операції над об'єктами предметної області;

— презентаційний шар. Класи в цьому шарі відповідають за створення інтерфейсу, видимого для користувачів.

Класи в рівні презентації повинні мати доступ до рівня бізнес-логіки і не повинні знати про рівень доступу до даних. Класи в шарі бізнес-реєстрації повинні працювати тільки з використанням рівня доступу до даних. Рівень доступу до даних повинен бути незалежним від будь-якого іншого шару. Архітектура додатків має високий вплив на загальну якість програмного забезпечення. Архітектура називається однією з істотних вимог до якості програмного забезпечення [4].

Існує не багато статей, що описують архітектуру VIPER, і немає емпіричних даних і ретельного аналізу, щоб зрозуміти, що архітектура більше підходить для окремих проектів з різною специфікою. Розробники все ще сумніваються, чи потрібно переробити проекти в нову архітектуру і які переваги вони отримують. Робота має на меті заповнити дослідницький розрив архітектури VIPER та порівняти її з більш традиційною архітектурою MVP.

1 ПОСТАНОВКА ЗАДАЧІ

Властивості програмних систем і прикладних програм сильно залежать від архітектурного підходу, на якому вони побудовані. Через це вибір правильної архітектури для виконання всіх потреб є надзвичайно важливим завданням. Відновлення програми завжди займає багато часу та ресурсів. Отже, впевненість у прийнятті правильного архітектурного рішення є дуже важливим питанням у сучасному світі IT-індустрії.

Змінюваність системи програмного забезпечення описує, наскільки добре система може бути зрозуміла, відремонтована і розширена. Технічне обслуговування є основною частиною процесу розвитку і займає більше ресурсів і часу, ніж будь-які інші частини цього процесу. Це означає, що одним з вимог до гарної архітектури є створення високопродуктивного проекту. Підвищення ремонтопридатності часто є основним критерієм вибору архітектури.

Висока продуктивність системи або програми також важлива у багатьох проектах. Вибір правильного архітектурного підходу значною мірою сприяє досягненню бажаної продуктивності. Розуміння плюсів і мінусів різних архітектурних підходів має ґрунтуватися на емпіричних дослідженнях, але для порівняння різних архітектур додатків Android, зокрема до ВІПЕР, не так багато досліджень, оскільки це зовсім нове.

Програми для Android представляють особливий інтерес через відносно нову технологію та швидке зростання [5]. Кожен програмний проект побудований на архітектурі. Щоб зрозуміти, як правильно підібрати архітектуру для проекту з певними вимогами та обмеженнями, відповіді на такі запитання будуть:

— яку архітектуру слід використовувати для розробки програми з урахуванням можливих обмежень, тобто часу, вартості, бажаної продуктивності;

— яку архітектуру слід використовувати для розробки додатків конкретного розміру, тобто застосування невеликого прототипу, середнього розміру, великого застосування з великим планом розширення;

— яка архітектура краще відповідає принципам SOLID;

— яка архітектура призводить до чистого коду;

— яка архітектура краще підходить для модульного застосування;

— яка архітектура краще підходить для бібліотек?

Існує декілька типів проектів Android, які визначаються кількістю доданих у порівнянні випадків. Випадок використання - це набір дій, які визначають взаємодію між користувачем і системою для досягнення мети. Малий проект - це проект, який має 0-6 різних випадків використання. Середній проект має 7-15 випадків використання. Великий проект має понад 15 випадків використання.

Таблиця 1.1 - Дослідницькі питання

Питання дослідження 1 (RQ1)	Which Android architecture is more maintainable, MVP or VIPER?
Питання дослідження 2 (RQ2)	Which Android architecture leads to better performance, MVP or VIPER?
Питання дослідження 3 (RQ3)	Which Android architecture fits better for different types of projects, MVP or VIPER?

Бібліотека - це тип проекту, який вирішує одну або дві проблеми і використовується іншими проектами програмного забезпечення.

Модульний проект - це тип проекту, який зазвичай є великим і побудований як набір невеликих проектів, які можуть функціонувати самостійно.

1.1 Метод дослідження

Огляд літератури різних порівняльних робіт [9] [12] показує, що при порівнянні різних архітектур вони покладаються в основному на одну основну властивість - ремонтопридатність. Для такого порівняння використовуються методи, засновані на сценаріях. Клементс, Казман і Клейн пропонують керівництво щодо того, як оцінювати архітектуру програмного забезпечення на основі наступних методів [6]:

— ATAM: метод аналізу компромісів в архітектурі;

— SAAM: метод аналізу архітектури програмного забезпечення;

— ARID: активні огляди проміжних конструкцій.

Ці методи оцінюють властивості архітектур з цілями, які пов'язані з бізнесом. Однією з таких цілей є скорочення витрат, наприклад. Для цієї тези немає прямих ділових цінностей. Отже, зниження витрат не може бути оцінено безпосередньо. Такі ж обмеження описані в інших наукових роботах, які оцінюють архітектури [14].

Метод, який буде використовуватися, схожий на метод, який фактично використовується для порівняння алгоритмів. Набір випадків спочатку буде реалізовано з використанням обох архітектур. Потім буде вибрано набір показників. Ці метрики включають в себе модифікацію, перевірочність, споживання пам'яті і час, необхідний для виконання випадку використання. Тоді будуть оцінені всі метрики для архітектур і дані метрики будуть зібрані за допомогою того ж сценарію та налаштування.

Для оцінки архітектур розроблено набір випадків використання. Кожен випадок використання складається з декількох типових дій для мобільних додатків, таких як перемикання екранів, введення тексту, доступ до бази даних тощо. Кожен випадок використання реалізується з використанням архітектурних підходів MVP і VIPER. Потім тести виконуються в кожному випадку використання, і дані метрики збираються. Для кожного випадку використання

логіка програми залишається незмінною, середовище виконання тесту постійне. Єдина відмінність - це архітектура.

1.2 Науковий вклад

Збір даних метрик та аналіз результатів дадуть відповіді на запитання, зазначені в розділі 1.3. Показники - це продуктивність, перевірочність і модифікованість архітектури, реалізовані в простих сценаріях використання. Визначені метрики та зібрані дані будуть достатніми для визначення переваг і недоліків кожної архітектури. Результат цієї дисертації також включає:

— опис двох архітектурних підходів до створення власних додатків Android;

— визначення способів порівняння архітектур, включаючи визначення відповідних метрик;

— дані метрики збору для десяти різних сценаріїв використання Android, реалізованих з використанням двох різних архітектур;

— порівняння двох архітектур на основі зібраних даних;

— пропозиції щодо вибору цих архітектурних підходів для різних завдань.

Результати тези створюють емпіричні дані щодо порівняння архітектури андроїда VIPER та архітектури MVP. Пов'язана робота не містить жодних даних, що описують вибір потрібної архітектури для різних типів проектів.

1.3 Цільові групи

Цільовою групою цього дослідження є професійні розробники програмного забезпечення, особливо розробники мобільних додатків. Вони матимуть вигоду,

знаючи відповіді на ключові дослідницькі питання цієї тези. Можливість спиратися на емпіричні дані може мати вирішальне значення при прийнятті рішення про архітектуру для вибору нового мобільного додатка. Це рішення вплине на витрати на розробку проектів андроїд-проектів та їх утримання, а також на час, необхідний для завершення розробки.

Дисертація описує метод порівняння мобільних архітектур, тому дослідники є другою цільовою групою. Порівняння мобільних архітектур не є тривіальним завданням. Наукова робота визначає ключові властивості для архітектур і пояснює ці властивості, представляючи їх як набір конкретних метрик, які можна виміряти. У дисертації також описано, як інтерпретувати результати цих вимірювань. Подальші дослідження в цій області можуть використовувати методи, описані в цій дисертації, для порівняння інших мобільних архітектур або для створення інших методів для порівняння архітектури.

1.4 Структура роботи

У Розділі 2 детальніше описуються основи проблеми. У ній пояснюється, що цікаво про цю проблему, чому її слід досліджувати і як буде використано рішення на практиці. Розділ 3 описує науковий підхід, який використовується для відповіді на проблему дослідження. У ньому міститься пояснення формальних методів дослідження, які використовуються для відповіді на дослідницькі питання. Метод поділяється на етапи, які пояснюються в цьому розділі. Розділ 4 містить деталі впровадження експерименту. Він містить список зовнішніх рамок, які були інтегровані для проведення експерименту. У розділі 5 Оцінки показані та описані результати експерименту. Він показує результати кожного етапу експерименту, аналіз і зроблені висновки. Висновки та майбутня робота Розділ 6 містить загальний висновок на тему дисертації та можливі шляхи подальшого вдосконалення.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Аналіз предметної області

Боб Мартін представляє спосіб розробки архітектури програмного забезпечення з використанням трьох шарів компонентів програмного забезпечення [2]. Ці шари ділять компоненти системи програмного забезпечення за обов'язками і залежностями. Розрізняють три основні шари.

Перший - це шар даних. Цей шар складається з класів і контрактів з основною метою виконання функцій зберігання даних. Ці класи відповідають за написання, читання, оновлення та видалення даних з місцевих або зовнішніх джерел. Ці джерела можуть включати бази даних, зовнішні служби, файлову систему та інше. Реалізація цих шарів залежить від операційної системи і протоколів, обраних для цієї системи або зовнішніх служб.

Другий - шар бізнес-логіки. Цей шар складається з класів і контрактів, які описують алгоритми основних функцій системи. Ці компоненти відповідають за виконання бізнес-логіки. Це включає в себе такі операції, як пошук, фільтрація, обчислення різних показників. Компоненти шару бізнес-логіки залежать від контрактів, які поширюються шаром даних. Ці класи використовують компоненти шару даних для роботи з даними. Реалізація компонентів на цьому шарі не повинна залежати від операційної системи або будь-якої іншої зовнішньої властивості.

Третій рівень - це рівень презентації. Класи і контракти цього шару відповідають за реалізацію інтерфейсів для взаємодії з користувачем. Користувач тут - це людина або зовнішня служба. У випадку людини інтерфейс користувача реалізується за допомогою командного рядка або графічних компонентів. Якщо користувач є зовнішньою службою, програмне забезпечення системи описує API. Реалізація компонента на рівні презентації розбивається на дві групи. Перша група компонентів відповідає за фактичне відображення графічних елементів. Ця група залежить від середовища, яке запускає програмне забезпечення. Інша група

відповідає за виконання логіки презентації та обробки подій перегляду. Ці події натискання на кнопку, жести на тач-скріні,та інше. Ця група не залежить від зовнішніх властивостей. Вона використовує шар бізнес-логіки для виклику відповідних функцій.

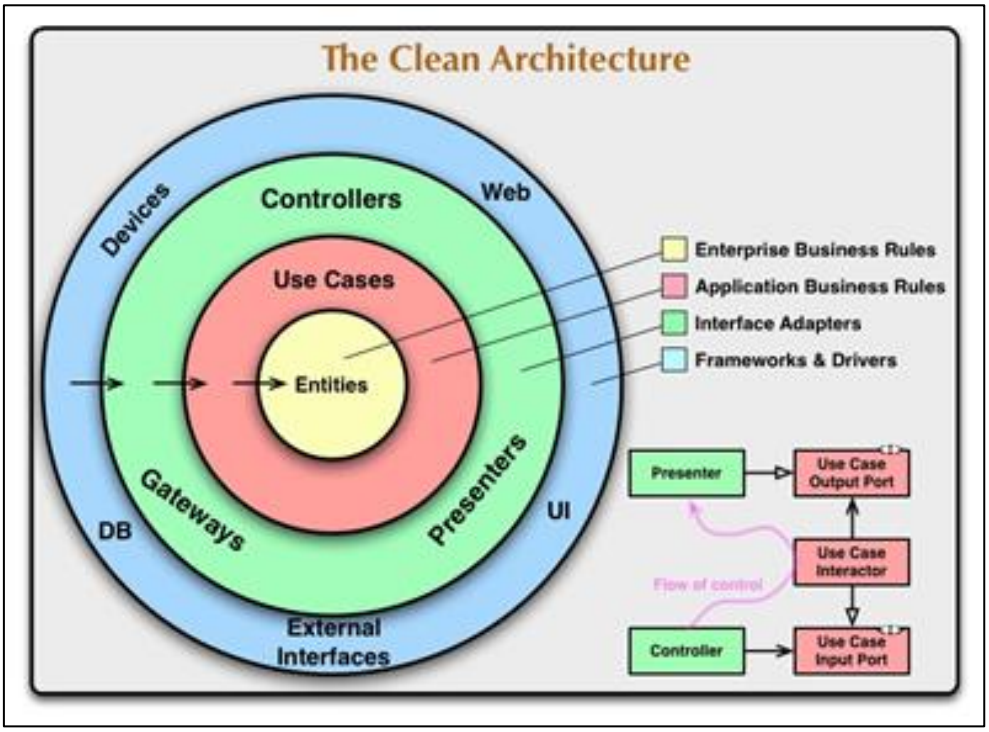

Рисунок 2.1 – Архітектурні шари компонентів

Обидві архітектури відповідають базовим принципам. До них відносяться принципи SOLID. SOLID - це абревіатура, яка складається з перших літер назв принципів. Ці принципи сформульовані для встановлення основних правил для гарного проектування системи.

Принцип 1. Принцип єдиної відповідальності. Це правило говорить, що кожен компонент повинен мати тільки одну причину для зміни. Це означає, що він має тільки одну відповідальність, і єдиною причиною зміни компонента є необхідність змінити його відповідальність.

Принцип 2. Принцип відкрито / закрито. Це правило стверджує, що кожен компонент повинен бути відкритим для розширення і закритий до модифікації. Це

означає, що додаткові функціональні можливості повинні бути реалізовані за допомогою розширення існуючих компонентів, а не зміни їх.

Принцип 3. Принцип заміщення Лісков. Це правило стверджує, що об'єкти в системі програмного забезпечення повинні мати можливість бути замінені їх підтипами, не впливаючи на правильність роботи системи. Це означає, що якщо об'єкт має підтип, цей підтип повинен реалізовувати всі контракти базового компонента.

Принцип 4. Принцип сегрегації інтерфейсів. Це правило стверджує, що контракти компонентів повинні бути мінімальними і описувати лише один набір правил. Компоненти програмної системи працюють на безлічі інтерфейсів, які називаються контрактами. Ці контракти визначають обов'язки компонента, не пов'язані з конкретною реалізацією. Це правило вводить спосіб структурування контрактів. Контракти повинні бути мінімальними. Поведінка компонента повинна описуватися, скоріше, складом малих контрактів, ніж одним великим контрактом.

Принцип 5. Принцип інверсії залежності. Це правило говорить, що компонент повинен залежати від абстракцій, а не від конкретних реалізацій. Цей принцип легко реалізується за допомогою інтерфейсів і контрактів.

Ці принципи спрямовані на встановлення деяких основних правил проектування архітектури. Вони визначають контракти як основний спосіб зв'язку між компонентами. Це призводить до більшої абстракції при реалізації системи програмного забезпечення. Багато абстракцій означає, що нові компоненти легше інтегруються в систему, оскільки єдиною вимогою до нового компонента є виконання контракту.

Зміна компонента у великій системі програмного забезпечення є складним завданням, оскільки існують інші компоненти, які залежать від нього. Використання контрактів означає, що жоден компонент не залежить від фактичної реалізації іншого компонента, але він залежить від контракту. Це означає, що зміна одного компонента не вплине на правильність системи, якщо вона виконує контракт.

Операційна система Android побудована на декількох основних компонентах і шарах. Верхній шар - прикладний шар. Всі програми виконуються на цьому рівні. Для доступу до функцій операційної системи API використовує API для шару Application Framework. Цей шар також містить власні програми. Ці програми забезпечуються реалізацією операційної системи. Прикладами рідних додатків є веб-браузер і програма електронної пошти.

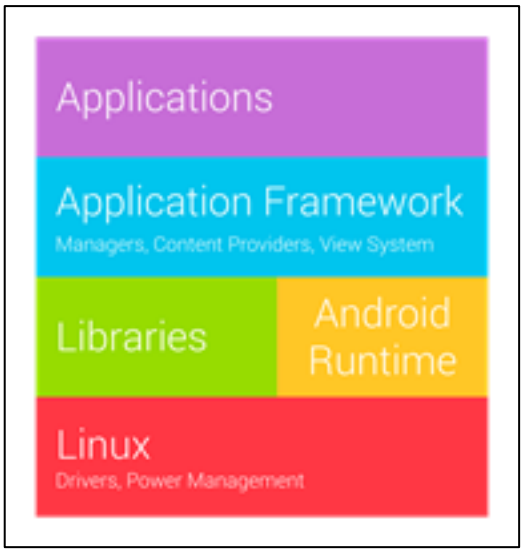

Рисунок 2.2 – Архітектура операційної системи Android

Каркас додатків відповідає за надання пристрою і системних даних через API до додатків. Він збирає дані з різних датчиків і обладнання. Цей шар створює середовище для запуску програм. Для цього він використовує різні служби Android. Деякі з цих служб:

— менеджер дій (містить стек діяльності);

— менеджер ресурсів (забезпечує доступ до ресурсів програми, наприклад, зображень і рядків);

— менеджер розташування (дозволяє програмам отримувати оновлення даних про місцезнаходження);

— менеджер повідомлень (забезпечує застосування засобами для відображення повідомлень);

— система перегляду (створює компоненти перегляду).

Для цього використовуються системні бібліотеки та середовище Android Runtime. Системні бібліотеки Android і середовище виконання програми Android Runtime знаходяться на одному рівні і відповідають за надання основних функцій і викликів процедур. Бібліотеки Android надають доступ до всіх основних функцій. Деякі бібліотеки:

— android.text (використовується для відображення тексту на екрані);

— android.media (використовується для відтворення відео та аудіо);

— android.database (забезпечує доступ до бази даних);

— android.hardware (використовується для доступу до апаратних датчиків).

Найбільш базовий шар - ядро Linux. Це ядро операційної системи, воно містить драйвери до апаратних компонентів і працює з ними. Ядро обробляє примітивну багатозадачність, керування живленням, управління процесами.

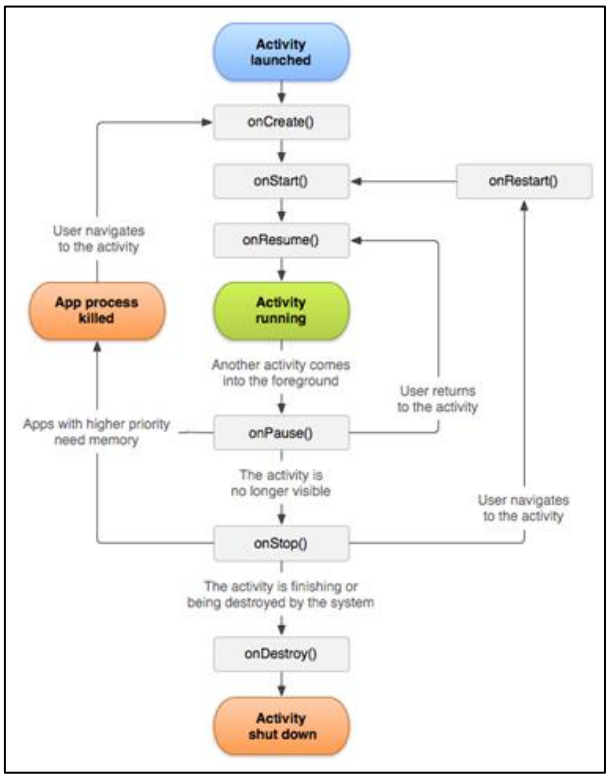

Рисунок 2.3 - Життєвий цикл активності

Додаток для андроїда складається з базових компонентів. Ці компоненти є активністю, фрагментом, службою, приймачем мовлення та постачальниками вмісту. Кожен компонент має свій власний життєвий цикл.

Активність є основним компонентом у створенні андроїд-додатків. Це єдиний компонент, який має графічний інтерфейс. Фрагменти є частиною діяльності, а також мають графічний інтерфейс. Активність зазвичай являє собою один екран програми. Фрагменти використовуються для поліпшення інтерфейсу користувача всередині діяльності.

Додаток для Android - це набір активностей, які змінюють один одного. Це необхідно враховувати при розробці архітектури Android. Активність має певний життєвий цикл і різні дії і виклики повинні виконуватися на різних етапах цього життєвого циклу.

Метод onCreate викликається при створенні активності. Основні компоненти повинні бути визначені і ініціалізовані на цьому кроці. Метод onResume викликається, коли користувач зможе почати взаємодію з графічним інтерфейсом. Метод onPause викликається, коли активність більше не видно. Метод onDestroy називається безпосередньо перед тим, як діяльність знищується.

Архітектура програмного застосування для Android повинна враховувати це, оскільки активність є єдиним інструментом для роботи з графічним інтерфейсом, і використання його неправильно призведе до помилок. Наприклад, виклик активності, що вже знищено, призведе до збою програми. Компоненти архітектури завжди повинні бути синхронізовані зі станом класів Android.

2.2 Аналіз існуючих архітектурних рішень

Існують три найпопулярніші мобільні архітектури, що використовуються в проектах Android в сучасній розробці додатків. Ці архітектури є MVC (Model-View-Controller), MVP (Model-View-Presenter) і MVVM (Model-View-ViewModel).

MVC - це скорочення від Model-View-Controller. Ця архітектура описує три основні компоненти: модель, перегляд і контролер. Модель відповідає за визначення структури даних і оновлює додаток для відображення поточного стану

даних. Контролер в цій архітектурі містить логіку управління. Він несе відповідальність за отримання оновлень від моделі та сповіщує про ці оновлення. В'ю відображає інтерфейс користувача. Він обробляє всі вхідні події користувача і відправляє його до контролера. Структура цієї архітектури показана на малюнку 2.4.

Основною особливістю архітектури Model-View-Controller є те, що Model безпосередньо пов'язана з View, щоб мати можливість змінити її відповідно до поточного стану даних. Контролер легкий і лише повідомляє модель про зміни даних, які спрацьовують користувачем. View у цій архітектурі має велику кількість обов'язків, оскільки він повинен обробляти всі введені користувачем дані, передавати їх до контролера та обробляти всі оновлення з моделі та контролера.

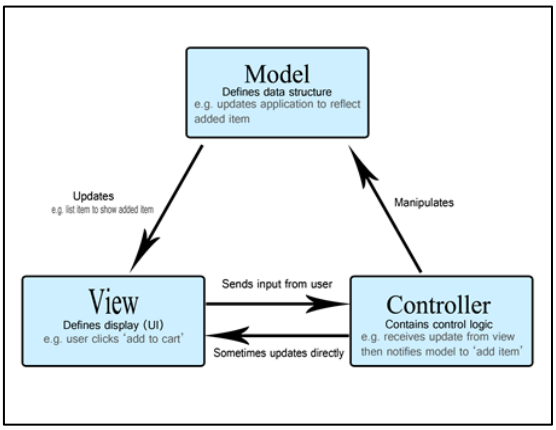

Рисунок 2.4 – Архітектура Model-View-Controller

Архітектура MVC була концепцією, що дозволяла паралельно розробляти програмний додаток, оскільки різні частини архітектури могли бути розроблені різними людьми одночасно. Це також був спосіб зробити компоненти програмних систем багаторазовими. Перегляд може бути змінений без будь-яких змін у моделі або контролері. Те ж саме стосується контролера і моделі. Проте прямий зв'язок між моделлю і в'ю створює цикл залежностей, що означає, що одну і ту ж модель важко використовувати з різними в'ю. Вона вимагає того, щоб або реалізувати

абстрактний контракт, або створити різні моделі для різних поглядів. Розроблено нові архітектури для вирішення цієї проблеми.

MVVM - це скорочення від Model-View-ViewModel. Модель представляє шар доступу до даних. Вона описує структуру даних системи програмного забезпечення. Він отримує оновлення з компонента ViewModel і оновлює стан цього компонента. В'ю відображає інтерфейс користувача. Він також обробляє вхідні події користувача і направляє його до компонента ViewModel. ViewModel - це компонент, який щільно приєднаний до в'ю. Він зберігає поточний стан в'ю та даних. Він відповідає за оновлення в'ю, обробку вхідних даних користувача, отриманих з в'ю, і оновлення даних у відповідь на цей вхід. Структура цієї архітектури показана на малюнку 2.5.

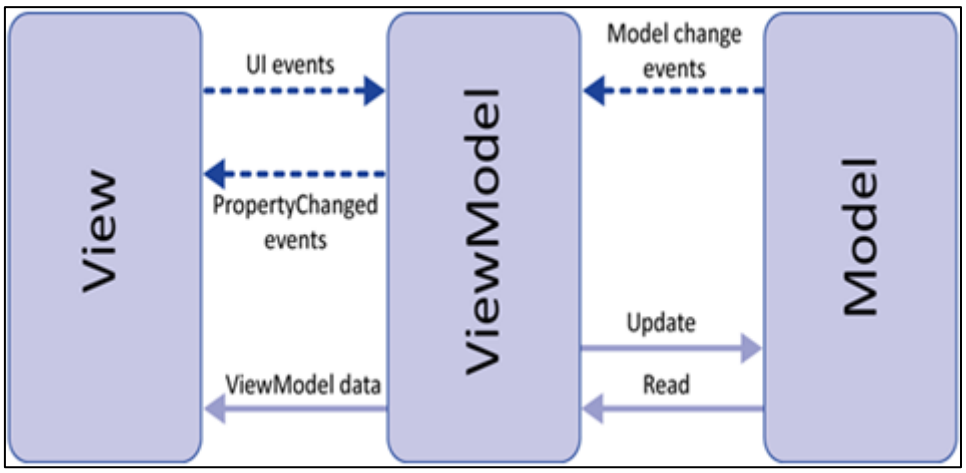

Рисунок 2.5 – Архітектура Model-View-ViewModel

Ця архітектура вимагає спеціальної технології зв'язування, щоб правильно функціонувати. Прив'язка потрібна для безпосереднього з'єднання стану даних, збережених у моделі перегляду, до перегляду. XAML (Extensible Application Markup Language) - це технологія прив'язки, реалізована Microsoft для ініціалізації структурованих значень і об'єктів. Вона є частиною .Net Framework і використовується в WPF (Windows Presentation Foundation). Ця структура використовується в основному для розробки додатків магазинів Windows і мобільних додатків для Windows Phone.

Google розробила власну структуру, яка реалізує технологію прив'язки, необхідну для архітектури MVVM. Бібліотека прив'язки даних описана на офіційному сайті розробників Android. Ця бібліотека є відносно новим і досі невивченим інструментом, як і підхід MVVM у розробці програмного забезпечення для Android. Ця архітектура повністю залежить від однієї конкретної бібліотеки не тільки для зручності, але й для функціонування.

2.3 Архітектура MVP

MVP - це скорочення від Model-View-Presenter. Ця архітектура найчастіше використовується у розробці додатків Android. Такий підхід дозволив зберегти переваги MVC як здатність до паралельного розвитку і виправляв її проблеми. Модель MVP не підключена до View і стало можливим повторне використання моделі без будь-яких обмежень. Це особливо добре, тому що модель представляє механізм, який надає дані в системі. Це означає, що незалежна модель може використовуватися окремо. Наприклад, можна реалізувати фонову службу Android, яка не має інтерфейсу користувача для програми, використовуючи ту ж саму модель, яка використовується в роботі програми з графічним інтерфейсом.

MVP є більш традиційною архітектурою [5]. MVP - Model-View-Presenter. Модель тут відповідає за зберігання об’єктів предметної області та виконання логічних операцій на них.

Модель зазвичай відноситься до шару бізнес-логіки. Рівень бізнес-логіки забезпечує контракти компонентів, які отримують дані системних систем і виконують логіку програми на ньому. Рівень доступу до даних також є частиною моделі, оскільки він безпосередньо відповідає за виконання основних операцій зберігання, таких як створення, читання, оновлення, видалення. Перегляд відповідає за створення інтерфейсу користувача. Вона містить логіку для ініціалізації кожного компонента інтерфейсу користувача в потрібному місці

екрану та обробки подій вводу користувача. Ці компоненти інтерфейсу користувача є кнопками, мітками, скролерами, випадаючими списками. І події, що вводяться користувачем, є клацаннями, прокрутками, свайпамми. Презентер відповідає за обробку всіх зворотних викликів з перегляду і використовує модель для отримання даних, необхідних перегляду для створення правильного інтерфейсу користувача. Він є посередником між поглядом і моделлю. Презентер отримує всі вхідні події користувача з перегляду і здійснює виклики моделі. Він також отримує відповіді від моделі та оновлює перегляд.

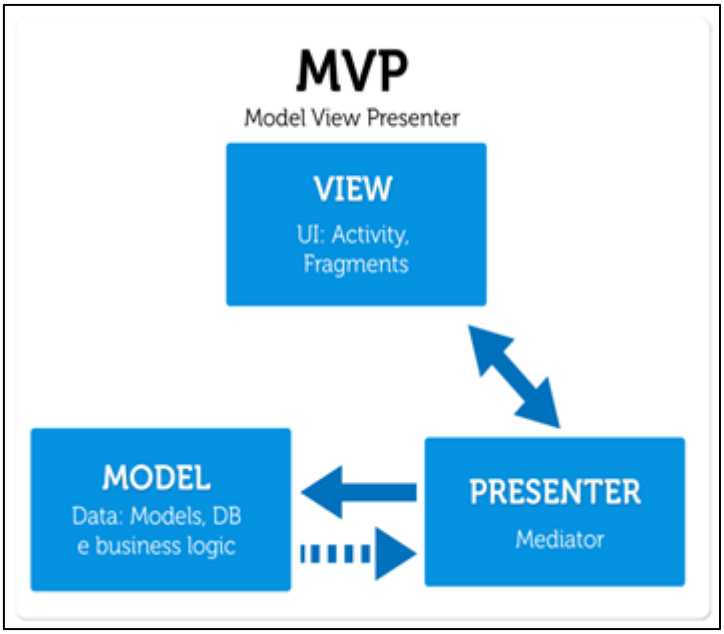

Рисунок 2.6 – Архітектура Model-View-Presenter

Однією з великих переваг архітектури Model-View-Presenter є те, що вона описує три рівні абстракції. Це полегшує налагодження програми для Android. Поділ бізнес-логіки та логіки доступу до даних також дозволяє здійснювати модульне тестування.

Архітектура Model-View-Presenter в Android забезпечує краще розмежування відповідальностей компонентів. Він переміщує бізнес-логіку та логіку доступу до даних з активіті та фрагментів. Активіті і фрагменти - це специфічні для платформи класи, а бізнес-логіка повинна бути незалежною від платформи. Архітектура програмного забезпечення Model-View-Presenter робить

програму більш багаторазовою. Презентер, перегляд і модель з'єднані за допомогою контрактів, що робить їх незалежними від конкретних реалізацій. Кожен презентер може бути легко замінений іншими презентерами, які реалізують іншу логіку, якщо вони виконують той же контракт.

2.4 Архітектура VIPER

Архітектура VIPER відносно нова. VIPER означає View-Interactor-Presenter-Entity-Router. Перегляд відповідає за різні елементи інтерфейсу користувача, наприклад, прогрес-бар, кнопки або діалогові вікна. Перегляд також обробляє події вводу користувача. Потім події вводу користувача передаються відповідним методам презентера.

Presenter обробляє всі зворотні виклики з перегляду і використовує interactor для отримання даних, необхідних перегляду для створення правильного інтерфейсу користувача. Він також виконує всю логіку подання, наприклад, приймає рішення про показ або приховування панелі прогресу, або показує діалогове вікно.

Інтерактор відповідає за виконання будь-яких логічних операцій на об'єктах предметної області. Interactor являє собою одну логічну операцію, наприклад, пошук або фільтрацію.

Презентер зазвичай використовує кілька взаємодіючих інтеракторів. Інтерактор використовує компоненти моделі для отримання об'єктів предметної області та виконання операцій бізнес-логіки на цих даних. Модель відповідає за зберігання об’єктів предметної області. Вона забезпечує інтерфейс для доступу до сховища даних і виконує основні операції, такі як створення, читання, оновлення і видалення. Маршрутизатор відповідає за навігацію між різними сценами інтерфейсу користувача. Доступ до маршрутизатора здійснюється презентером.

Велика перевага архітектури програмного забезпечення View-Interactor-Presenter-Entity-Router полягає в тому, що вона забезпечує хороше покриття модульними тестами. Головною особливістю архітектури програмного забезпечення VIPER є вільний зв'язок між компонентами. Кожен компонент призначений для того, щоб мати найменший діапазон обов'язків і виконувати найменший набір дій. Це стає у нагоді під час впровадження модульних тестів для проекту. Кожен компонент легко перевірити, тому що всі залежності можна підмінити моками.

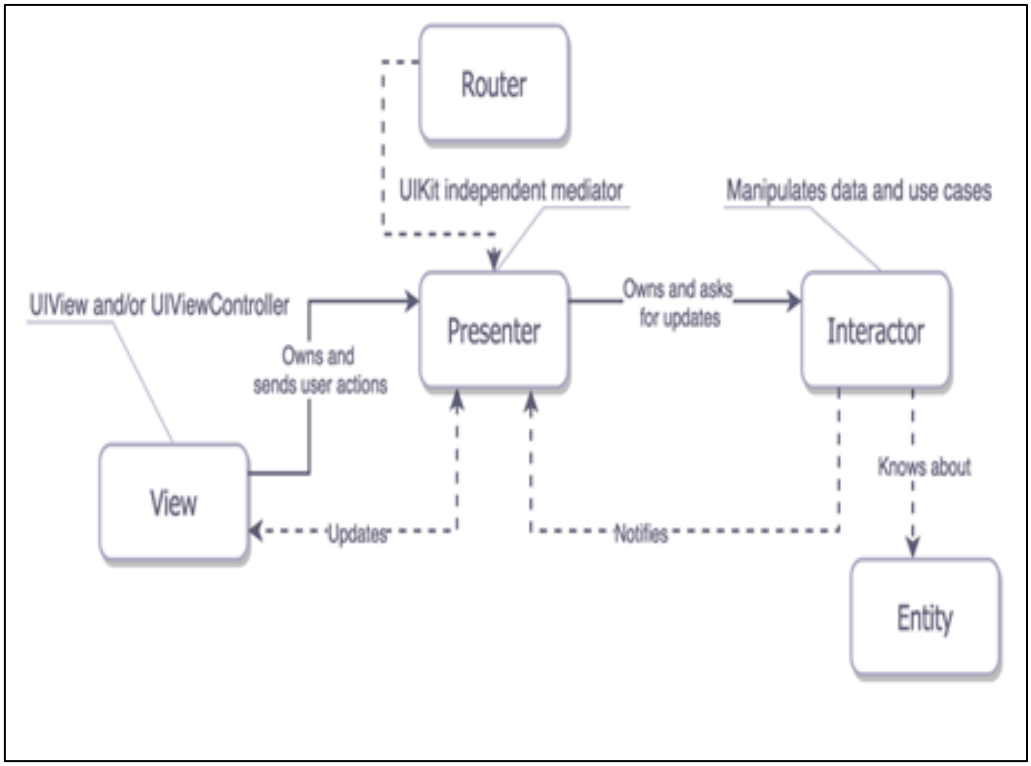

Рисунок 2.7 – Архітектура View-Interactor-Presenter-Entity-Router

Архітектура View-Interactor-Presenter-Entity-Router спочатку була спробою застосувати принципи SOLID на платформі iOS. Перейшовши на платформу Android, архітектура зберегла орієнтацію на дотримання основних принципів проектування архітектури.

Архітектура програмного забезпечення VIPER має незалежні модулі, що дозволяє легко працювати над програмою великій команді розробників. Спочатку реалізується основний каркас архітектури, потім конкретні модулі призначені для

реалізації різним розробникам. Всі реалізації компонентів засновані на одному каркасі, тому модулі проекту виглядають однотипно.

Кодова база проекту з використанням архітектури VIPER виглядає дуже схоже. Це дозволяє легко зрозуміти проект для нових членів команди, які знайомі з архітектурою VIPER. Код не складно зрозуміти для людини, яка знайома з VIPER, але не знайома з певною платформою. Наприклад, розробник android здатний зрозуміти і iOS-проект, тому що компоненти схожі і структура однакова.

Обидві архітектури, MVP і VIPER, дотримуються принципів базової архітектури. Існує декілька відмінностей у структурі та відповідальностях компонентів, що призводить до різної реалізації проекту та різниці в таких властивостях, як змінюваність та легкість покриття тестами.

MVP і VIPER мають подібні компоненти. Але вони мають різні обов'язки. Презентер у MVP відповідає за обробку зворотних викликів з перегляду, виконання бізнес-логіки та отримання даних з моделі. Архітектура VIPER визначає окремий компонент, interactor, для виконання бізнес-логіки, тому презентер є більш простим. Перегляд у MVP відповідає за побудову інтерфейсу користувача, а також навігацію між різними видами. VIPER представляє окремий компонент, маршрутизатор, який обробляє всю навігацію. Він зовсім не підключений до перегляду, він діє за командою презентера.

2.5 Сценарії використання програм для Android

Сценарій використання андроїд-додатків - це набір основних дій, які необхідно виконати для завершення одного випадку використання. Ці дії описані з точки зору застосування. Основні дії: натиснути кнопку, отримати доступ до сховища даних, відкрити новий екран, ввести текст, вибрати елемент зі списку, перемкнути вкладки тощо.

Найбільш популярні програми для Android зазвичай мають невеликий або середній розмір сценаріїв використання. Малі - сценарії використання, які складаються з п'яти основних дій. Сценарії використання середнього масштабу складаються з п'яти-десяти основних дій. Це можна переглядати в найпопулярніших додатках для Android.

Twitter є одним з найпопулярніших додатків для Android. Ця програма дозволяє користувачам публікувати короткі повідомлення. Найпопулярнішим сценарієм використання тут є розміщення твітів. Він складається з наступних основних дій:

— натиснути кнопку, щоб почати твітувати;

— відкрити новий екран твітів;

— тип повідомлення;

— натисніть кнопку "твіт";

— пост-твіт на сервері (доступ до сховища даних).

Ще одним популярним сценарієм використання у програмі Twitter є читання каналу користувача. Основні дії тут такі:

— відкрити екран подачі даних користувача;

— завантажувати твіти в використання feed (доступ до сховища даних);

— прокручування для перегляду твітів.

Ці сценарії є найбільш популярними сценаріями використання у програмі. Один з цих сценаріїв має три основні дії, інший - п'ять. Обидва ці сценарії використання вважаються малими через кількість дій.

Іншим прикладом популярної програми Android є Netflix. Ця програма дозволяє користувачеві переглядати різні телепрограми онлайн. Найбільш популярним сценарієм тут є відкриття останнього епізоду телевізійного шоу, який користувач регулярно спостерігає. Цей сценарій складається з наступних основних дій:

— завантажувати нещодавно переглянуті ТВ-шоу (доступ до сховища даних);

— натисніть на піктограму телевізійного шоу;

— відкрити новий екран для перегляду відео;

— завантажувати останній епізод (доступ до сховища даних).

Ще одним популярним сценарієм є пошук телепередачі і початок його перегляду. Нижче наведено перелік основних дій:

— відкритий екран пошуку;

— введіть назву телешоу;

— завантажувати список ТВ-шоу, що відповідає пошуковому запиту (доступ до сховища даних);

— натисніть на піктограму телевізійного шоу;

— відкрити новий екран для передачі телевізійного шоу;

— завантаження ТВ-шоу (доступ до сховища даних).

Ці сценарії використання найбільш часто використовуються в додатку Netflix, оскільки вони відповідають його головній ідеї. Ці сценарії складаються з чотирьох і шести основних дій. Ця кількість дій відповідає малим і середнім сценаріям використання.

Календар - це програма, яка встановлюється на кожному мобільному пристрої. Основними сценаріями використання тут є перегляд подій поточного дня / тижня та створення нового події. Перегляд подій за тиждень - це сценарій використання, який складається з наступних основних дій:

— показувати екран зі списком завдань;

— завантажувати всі події на поточний тиждень (доступ до сховища даних);

— прокручування, щоб переглянути всі події.

Іншим популярним сценарієм використання є створення нової події в календарі. Вона має такі основні дії:

— натисніть кнопку створення;

— відкрити новий екран створення події;

— вибрати дату події;

— вибрати час події;

— введіть назву події;

— введіть опис події;

— натисніть кнопку "Зберегти";

— створити нову подію в базі даних (отримати доступ до сховища даних).

Перший сценарій використання має лише три основні дії, а другий складається з восьми. Ці сценарії використання описують основну мету застосування. Один з основних сценаріїв використання календаря невеликий, а один - середнього розміру.

Інший мобільний додаток, присутній на будь-якому пристрої, - це електронна пошта. Ця програма надає користувачеві можливість переглядати та надсилати електронні листи. І це найбільш популярні сценарії використання для цієї програми. Щоб переглянути електронну пошту, потрібно виконати такі дії:

— відкритий екран списку повідомлень;

— завантажувати всі листи (доступ до сховища даних);

— вибрати одну електронну пошту;

— відкрийте новий екран для перегляду електронної пошти;

— завантажувати вибрану електронну пошту (доступ до сховища даних).

Сценарій використання нового листа складається з основних кроків, наведених нижче:

— натисніть кнопку "нова електронна пошта";

— відкрити новий екран для створення електронного листа;

— введіть адресу одержувача;

— введіть назву;

— введіть текст повідомлення електронної пошти;

— натисніть кнопку "Надіслати".

Обидва ці сценарії використання мають кількість основних дій не більше шести. Програма, яка дозволяє користувачеві надсилати та отримувати електронні листи, є найосновнішою для кожного мобільного пристрою. Перелічені сценарії використання не містять складних обчислень.

Youtube є одним з найпопулярніших і найпопулярніших додатків для мобільних пристроїв Android. Найбільш популярним сценарієм використання у

цій програмі є перегляд відео. Щоб знайти перегляд і переглянути його, потрібно виконати такі основні дії:

— натисніть кнопку "пошук";

— відкритий екран пошуку;

— введіть пошуковий запит;

— завантажувати відео (доступ до джерела даних).

Цей сценарій доступу до даних, що включає пошук, складається з семи основних дій з точки зору програми. Ці сценарії використання використовують мережу, але не мають складних обчислень або надмірної багатозадачності. Найбільш популярні сценарії використання в основному мають на меті дозволити користувачеві швидше досягти цілей.

Приклад цих п'яти популярних додатків Android з сотнями мільйонів завантажень показує, що найбільш популярні сценарії використання складаються з невеликої кількості основних дій. Це означає, що сценарії використання малого та середнього розміру мають особливий інтерес при оцінці архітектур додатків.

2.6 Огляд літератури

Існуючі дослідження на цю тему в основному складаються з різних наукових робіт, що описують порівняння інших архітектур android або методів оцінки архітектур android.

Tian Lou зробив дослідження в цій області в своїй роботі "Порівняння Android Native App Architecture - MVC, MVP і MVVM" [6]. Для порівняння цих архітектур вибрано ключові показники. Ці показники - продуктивність, змінюваність і легкість покриття тестами. Зібрані дані пізніше порівняли та проаналізували, щоб визначити кращий архітектурний підхід.

Аналіз і експерименти показують, що MVP і MVVM мають кращу легкість покриття тестами, змінюваність (низький рівень зв'язку) і продуктивність

(витрачаючи менше пам'яті). Тобто MVP і MVVM краще, ніж MVC за вибраними трьома критеріями. Але для MVP і MVVM немає доказів, які свідчать про те, що він перевершує інший. Ці дві архітектури мають подібну продуктивність, тоді як MVP забезпечує кращу змінюваність, а MVVM забезпечує краще покриття тестами[6].

"Структурування додатків для Android" Anthony Madhvani [7] описує способи поєднання різних архітектурних підходів (MVP, MVVM, MVC) з різними шаблонами проектування, а також фреймворками для створення програми для Android. Ця робота також описує, яку архітектуру вибрати для різних проектів відповідно до специфіки проекту.

Hugo Kallstrom у своїй роботі під назвою "Збільшення ремонтопридатності для додатків для Android" порівнює різні архітектури (MVP, MVC і MVVM), щоб визначити, який є більш ремонтопридатним, змінюваним і легким у покритті тестами [8].

Аналіз уразливостей Android та сучасних методів експлуатації [9] описує важливість розробки додатків з гарною архітектурою, щоб зробити її менш можливою. Основні принципи архітектури:

— змінити замість будівлі до останнього;

— поділ проблем;

— принцип єдиної відповідальності;

— ідентифікувати компоненти та згрупувати їх у логічних шарах;

— не повторюйте функціональність.

Ці принципи в основному вимагають підтримки архітектури. Змінюватися замість будівництва до останнього означає, що проект застосувань значно зміниться і важливо розробити його таким чином, щоб зменшити витрати на його зміну. Наступні принципи описують, як це зробити. Перекриття функціональних можливостей компонентів, багаторазові обов'язки компонентів, повторення функціональних можливостей ускладнять розуміння проекту та його зміну.

Цей розділ охоплює основні знання про архітектуру програмного забезпечення, його властивості та мотивацію. Архітектура програмного

забезпечення розроблена таким чином, щоб структурувати проект програмного додатку. Це важливо, щоб допомогти компаніям з розробки програмного забезпечення краще досягти своїх цілей. Компанії-розробники програмних додатків прагнуть забезпечити програмне забезпечення в найкоротші терміни з найменшими витратами і високою якістю. Ці вимоги перетворилися на визначення найбільш важливих властивостей архітектури програмного забезпечення. Це ремонтопридатність і продуктивність. Ремонтопридатність описується як композиція змінюваності і легкості покриття тестами. У цій главі також пояснюються дві архітектури, які порівнюються в цій роботі: Model-View-Presenter і View-Interactor-Presenter-Entity-Router. Ці архітектури мають подібні компоненти, але обов'язки цих компонентів відрізняються.

Огляд літератури описує тези та статті, що стосуються порівняння архітектур програмного забезпечення та властивостей архітектур. Архітектури порівнюються на основі їх властивостей. Головні архітектурні властивості спочатку оцінюються на простих і базових прикладах додатків. Ці властивості потім порівнюються і аналізуються. Властивості архітектури визначають, чи підходить архітектура для одного типу проекту або іншого проекту. Наприклад, стартапи зазвичай вимагають, щоб перша версія програмної системи була розроблена якомога швидше. Це означає, що такі властивості, як продуктивність або перевірочність, можуть бути принесені в жертву для більшої змінюваності.

2.7 Резюме

Розділ, присвячений предметній області, охоплює основні знання про архітектуру програмного забезпечення, його властивості та мотивацію. Архітектура програмного забезпечення розроблена таким чином, щоб структурувати проект програмного забезпечення. Це важливо, щоб допомогти компаніям з розробки програмного забезпечення краще досягти своїх цілей.

Компанії-розробники програмного забезпечення прагнуть забезпечити програмне забезпечення в найкоротші терміни з найменшими витратами і високою якістю. Ці вимоги перетворилися на визначення найбільш важливих властивостей архітектури програмного забезпечення. Це ремонтопридатність і продуктивність. Ремонтопридатність описується як композиція модифікованості і перевіряемості. У цій главі також пояснюються дві архітектури, які порівнюються в цій тезі: Model-View-Presenter і View-Interactor-Presenter-Entity-Router. Ці архітектури мають подібні компоненти, але обов'язки цих компонентів відрізняються.

Огляд літератури описує тези та статті, що стосуються порівняння архітектур програмного забезпечення та властивостей архітектур. Архітектури порівнюються на основі їх властивостей. Головні архітектурні властивості спочатку оцінюються на простих і базових прикладах додатків. Ці властивості потім порівнюються і аналізуються. Властивості архітектури визначають, чи підходить архітектура для одного типу проекту або іншого проекту. Наприклад, стартапи зазвичай вимагають, щоб перша версія програмної системи була розроблена якомога швидше. Це означає, що такі властивості, як продуктивність або перевірочність, можуть бути принесені в жертву для більшої модифікованості.

3 МЕТОД

Порівняння архітектур - це емпіричне дослідження. Методи, які використовуються для порівняння архітектур, ґрунтуються на експериментальних дослідженнях. Експериментальне дослідження проводиться у вигляді набору тестів. Для цього створюється стабільне середовище, розробляються тести. Мета експерименту - довести наукову теорію з визначеними залежними і незалежними змінними.

Використовуваний метод аналогічний методу, який використовується для порівняння алгоритмів. Приклади програм, які слідують архітектурам, спочатку будуть реалізовані, буде вибрано набір показників. Тоді всі метрики для архітектур будуть зібрані з використанням того ж сценарію та налаштування. Для оцінки архітектур розроблено набір випадків використання.

Кожен випадок використання складається з декількох дій, таких як перемикання екранів, набору тексту, доступу до бази даних і т.д. Це також мотивує розмір і складність сценаріїв використання тестів.

Дослідження вимірює такі властивості, як продуктивність, тестованість і змінюваність. Основною властивістю архітектури проекту є ремонтопридатність. Тестируемость і змінюваність сприяють цьому властивості, тому вони будуть вимірюватися. Продуктивність повинна вимірюватися, оскільки архітектурні вимоги свідчать про те, що має бути належне управління пам'яттю, а інтерфейс користувача не повинен впливати на будь-які відставання або затримки.

3.1 Науковий підхід

Дослідження починається з огляду літератури. Цей метод використовується для встановлення передумов дослідження і вибору основного методу і засобів

реалізації експерименту, який допомагає відповісти на дослідницькі питання. Результатом систематичного огляду літератури в області архітектури програмного забезпечення та порівняння мобільної архітектури є конкретно визначені основні властивості архітектури та загальний підхід до порівняння архітектур.

На основі огляду літератури експеримент розрахований на оцінку архітектур програмного забезпечення Android. Програмні архітектури, MVP і VIPER, оцінюються шляхом вимірювання набору властивостей для обох архітектур у стабільному середовищі. Дані, отримані в експерименті, аналізуються для порівняння архітектур і відповіді на дослідницькі питання. Прямого порівняння властивостей архітектур достатньо, щоб відповісти на дослідницькі питання.

3.2 Опис метода

Числові дані повинні бути зібрані для всіх метрик для обох архітектур, тому використовується контрольований метод експерименту. Для проведення експерименту вибрано стабільне і постійне середовище виконання. Це фізичний пристрій Android, таблиця 3.1 містить технічні характеристики пристрою.

Таблиця 3.1 - Технічна специфікація пристрою

Device name	LG Nexus 5X
Operating system	Android 8.1.0
Chipset	Qualcomm MSM8992 Snapdragon 808
CPU	Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57)
GPU	Adreno 418

Кінець таблиці 3.1

RAM	2Gb
Resolution	1080 x 1920 pixels, 16:9

Незалежною змінною в цьому експерименті є те, що зараз використовується випадок використання. Залежні змінні - це показники продуктивності, змінюваність і легкість покриття тестами. Показники продуктивності - це споживання пам'яті та час, необхідний для виконання сценарію використання. Всі дії користувача виконуються за допомогою автоматизованого тестування, тому вони завжди беруть постійний час. Це означає, що натискання кнопки або прокручування списку завжди займає стільки ж часу. Тут потрібна автоматизація, щоб переконатися, що єдиним фактором, що впливає на вимірювання показників продуктивності, є розробка архітектури програмного забезпечення.

Показники змінюваності - це кількість класів, які потрібно додати для реалізації функції, кількість класів, які необхідно редагувати для реалізації функції, і ряд рядків коду, які необхідно додати або змінити для реалізації функції. Ці метрики описують обсяг механічної роботи, необхідний для реалізації функції. Реалізації всіх випадків використання обох архітектур покладаються на ті самі рамки. Це означає, що кількість коду залежить тільки від фактичного дизайну архітектури та кількості компонентів, які ця архітектура має виконувати.

Тестопридатність описується метрикою тестового покриття. Тестове покриття показує, яка частина загальної кількості класів в проекті покрита тестами, тобто виконана в тестовій середовищі для перевірки правильності. Найбільш поширені підходи в галузі розробки програмного забезпечення були обрані для реалізації тестових випадків. Модульні тести використовуються для перевірки кожного окремого компонента. Інтеграційні тести реалізуються для перевірки правильності роботи цілих випадків використання, що вимагає декількох компонентів для спільної роботи. Тести користувача інтерфейсу гарантують, що графічні компоненти відображаються правильно і реагують на дії користувача. Числові дані, зібрані в цьому експерименті, пізніше

використовуються для порівняння та аналізу архітектур для забезпечення якісних результатів.

Кілька типів проектів включені в порівняння, щоб краще визначити, яка архітектура буде відповідати цим типам програмних проектів. Кожен тип проекту має різні ваги для кожного показника, зібраного в експерименті.

Найбільш важливим показником для малого та середнього проекту є кількість класів, які необхідно додати для реалізації функції. Малі проекти в основному є прототипами, які необхідно реалізувати в найкоротші терміни. Показники продуктивності та тестованості не є важливими, оскільки проект, ймовірно, не буде містити жодної складної бізнес-логіки.

Тестопридатність і продуктивність є важливими показниками для великих проектів. Тестопридатність полегшує реалізацію нових функцій і гарантує, що не вплине на правильність реалізації попередніх функцій. Важливо також і кількість класів, які необхідно редагувати для реалізації нової функції, оскільки великі проекти зазвичай містять багато класів і змінюють їх можуть вплинути на правильність системи.

Бібліотеки програмного забезпечення оцінюють такі метрики, як кількість класів, які необхідно додати для реалізації функції та кількості рядків коду. Бібліотеки використовуються як частини для реалізації інших проектів і потребують невеликого розміру, оскільки проекти зазвичай використовують декілька бібліотек, і в результаті їх застосування може бути досить важким.

3.3 Надійність і дійсність метода

Метою експерименту є порівняння архітектур шляхом виконання випадків використання, побудованих з використанням цих архітектур. Продуктивність вимірюється в термінах часу виконання і використовуваної пам'яті. Зібрані результати не будуть мати абсолютних значень, але будуть цінними відносно

один до одного. Бізнес-логіка буде такою ж, тому вона не вплине на результати порівняння, і єдина річ в додатках буде способом організації коду. Архітектури описують компоненти і відносини між компонентами, тому вони не залежатимуть від особи, яка реалізує архітектуру. Логіка застосування, логіка презентації та логіка доступу до даних залишаються однаковими для сценаріїв використання тестових випадків, тому ці фактори не впливатимуть на результати порівняння. Це означає, що архітектури можуть порівнюватися, а спосіб реалізації кожного конкретного класу не вплине на результати порівняння.

Основною метою цього методу є збір даних про однакові показники та використання тієї ж установки для різних архітектур. Це означає, що вимірювання є дійсними для порівняння метрик для архітектур, але не представляють абсолютних значень для бенчмарку архітектури.

Швидкість введення та натискання кнопки керування є постійною, оскільки вони виконуються за допомогою автоматизованої структури. Швидкість малювання і завантаження є постійною, оскільки налаштування пристрою є постійним. Єдиним фактором, що впливає на обсяг пам'яті, необхідний для програми та часу, є кількість операцій, які необхідно завантажити, що відрізняється для архітектур додатків.

Результати можуть не бути репрезентативними для великих проектів з дуже складною бізнес-логікою, оскільки випадки використання зразків не є великими. Крім того, ці результати не будуть дійсними для ігрових додатків, оскільки вони розроблені з використанням різного підходу і мають різні цілі.

Також такий спосіб оцінки архітектурної перевірки може ввести в оману сам по собі. Тести можуть не бути хорошими і можуть вводити в оману. Високий рівень тестового покриття як основна мета архітектури проекту призводить до більшої уваги до тестування, а не до досягнення бажаної поведінки. І реалізація правильної поведінки архітектурних компонентів є головною метою будь-якої архітектури.

Експеримент виконується на сценаріях малого використання. Ці сценарії використання складаються з невеликого до середнього числа простих кроків. Ці

дії є типовими діями, які виконуються під час роботи програми. Ці дії відкривають нові екрани, натискання кнопок, введення тексту, завантаження даних, виконання алгоритмів, доступ до бази даних та інших. Сценарії використання, представлені в експерименті, складаються з відкриття двох / трьох екранів, доступу до бази даних, завантаження даних, збереження даних, натискання кнопок і введення тексту. Більшість сценаріїв використання для мобільних додатків не великі. Наприклад, є популярне додаток під назвою Instagram. Це дозволяє користувачеві розміщувати фотографії, переглядати фотографії друзів і подібні фотографії. Подібно до фото - це один із сценаріїв використання для цієї програми. З точки зору програми вона складається з наступних дій: отримати користувальницький канал зі списку (отримати доступ до сховища даних), прокрутити канал до фотографії, натиснути подібну кнопку, надіслати запит на сервер (отримати доступ до даних) зберігання). Вона складається тільки з чотирьох основних дій і не потребує зміни екранів.

Іншим сценарієм є розміщення фотографії. Дії тут такі: натисніть кнопку, щоб почати публікацію, відкрийте новий екран, виберіть фотографію, відкрийте новий екран, виберіть фільтр, застосуйте вибраний фільтр, відкрийте новий екран, введіть підпис, натисніть кнопку "повідомлення", створити повідомлення на сервері (отримати доступ до сховища даних). Цей сценарій використання довший, ніж попередній, і складається з відкриття трьох нових екранів. Більшість додатків складаються як з короткого, так і середнього сценаріїв використання. Проте розробники великих додатків намагаються мінімізувати кількість дій і екранів, необхідних для завершення сценарію використання, оскільки користувачі, як правило, втрачають або втрачають бажання завершити сценарій.

Експеримент проводиться на малих і середніх сценаріях використання. Результати порівняння можуть не бути репрезентативними для будь-якого андроїд-програми, оскільки тестові сценарії мають певні обмеження. Тестові сценарії складаються з невеликої або середньої кількості дій (до десяти). Тестові сценарії не роблять багато паралельних обчислень. Він використовує

багатопоточність для виконання бізнес-логіки і доступу до сховища даних, але не має складних обчислень.

Експеримент проводиться лише на десяти різних сценаріях використання. Ці тестові сценарії представляють найпоширеніші сценарії в мобільних додатках, оскільки містять основні дії. Більша кількість тестових сценаріїв підвищить достовірність результатів. Впровадження нових тестових сценаріїв є предметом майбутніх досліджень. Метою роботи є порівняння двох архітектур андроїд-додатків, реалізованих для найбільш базових додатків і сценаріїв використання. Ці сценарії використання є основою програми, а потім доповнюються більш складними. Виконання архітектури в додатках з такими функціями, як виконання складних обчислень, великих сценаріїв використання або робота з великою кількістю даних, може бути досліджено в подальшому.

Достовірність у вимірюванні показників продуктивності досягається тим, що кілька разів виконується тестування. Споживання пам'яті та час, необхідний для виконання сценарію використання, вимірюються на одному і тому ж наборі сценаріїв використання в одному тестовому середовищі. Один тест складається з 500 ітерацій. Цей тест проводився два рази, і вимірювання порівнювалися з тим, щоб результати не змінювалися з часом.

Щоб гарантувати, що результати порівняння архітектури можуть бути надійними, всі тести виконуються, і всі метрики збираються в десяти різних сценаріях використання. Ті ж результати порівняння для кожного сценарію використання означають, що ці результати мають внутрішню надійність узгодженості.

Всі сценарії використання реалізовані однією особою, що є недоліком достовірності дослідження. Метрики для оцінки архітектур були розроблені з метою мінімізації ефекту цього недоліку на кінцевий результат. Отримані вимірювання для обох архітектур залишаються цінними в порівнянні, але не як абсолютні значення. Запрошуючи більше розробників програмного забезпечення з досвідом розробки Android для реалізації тих самих сценаріїв використання, можна підвищити надійність результатів.

4 ДЕТАЛІ РЕАЛІЗАЦІЇ

Всі випадки використання реалізовані у вигляді андроїд-додатків. Ці програми вимагають мінімального рівня API 16 для Android, що відповідає Android 4.1 Jelly Beans. 99% ринку Android працює на Android версії 4.1 або вище, згідно з офіційними панелями Android [20]. Додаток для Android реалізовано на мові програмування Java. Блокові тести реалізовані в Java, використовуючи рамки тестування JUnit і Mockito. Тести інтерфейсу інтерфейсу реалізовані на мові програмування Kotlin з використанням тестової бази Espresso.

Ці рамки використовуються при реалізації тестових сценаріїв використання для зручності. Такий же набір платформ Android використовується для розробки випадків використання як VIPER, так і MVP. Ці фреймворки використовуються розробниками додатків Android у величезній кількості додатків. Java є найбільш використовуваною мовою програмування для розробки додатків Android. Kotlin є мовою програмування, яка є відносно новою, але офіційно підтримується Google як мова для розробки додатків для Android з більшістю офіційних посібників, що містять приклади в Java і Kotlin.

Реалізація додатків базується на декількох рамках. Google Room використовується для реалізації рівня доступу до даних за допомогою шаблону об'єктів доступу до даних. Кадр 2 використовується для реалізації інжекції залежності. Рамки RxJava і RxAndroid використовуються для дотримання принципів реактивного програмування. Вибір фреймворків для роботи з базою даних або реалізація ін'єкцій залежностей не є важливим. Головне - дотримуватися одного і того ж підходу і використовувати ті самі рамки при реалізації сценаріїв тестування для обох архітектур.

На рисунку 4.1 показаний інтерфейс користувача декількох екранів у реалізованих випадках використання. Користувальницький інтерфейс використовує сучасний підхід в розробці, розроблений компанією Google, що має назву Material design. Навігація між основними екранами здійснюється у вигляді

нижньої навігаційної панелі. Кнопка з плаваючою дією використовується для підняття уваги користувачів до основних дій на екрані, таких як створення нової нотатки або категорії. Меню використовується для представлення набору керуючих дій для кожного екрана, наприклад збереження або видалення.

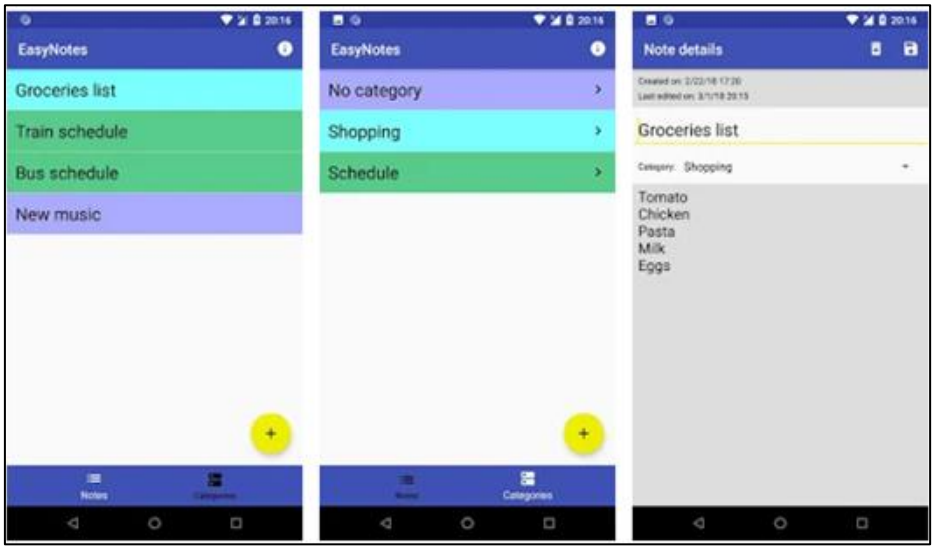

Рисунок 4.1 - Інтерфейс коистувача

Тести розбиті на категорії. Перша категорія представлена одиничними випробуваннями. Ці тести реалізовані з використанням мови програмування Java. Основна мета модульних тестів полягає в тому, щоб гарантувати, що кожен компонент програмної системи працює правильно. Ці компоненти є класами в шарі доступу до даних, шару домену та класах шару презентації, які виконують логіку перегляду. Щоб переконатися, що класи, від яких залежить конкретний компонент, не впливають на результати тесту, ці залежності знущаються за допомогою спеціального тестування Mockito.

Друга категорія - це інтеграційні тести. Ці тести реалізовані з використанням мови програмування Java. Основна мета інтеграційних тестів полягає в тому, щоб переконатися, що компоненти працюють правильно разом. Це означає, що ці тести охоплюють конкретні потоки даних і сценарії діяльності. Як тільки кожен компонент покривається блоковими тестами, результати

інтеграційних тестів залежать тільки від правильності реалізації конкретного сценарію.

Третя категорія - тести інтерфейсу користувача. Ці тести реалізовані з використанням мови програмування Kotlin. Основна мета тестів інтерфейсу користувача полягає в тому, щоб забезпечити відображення кожного графічного компонента в потрібний час і реагувати на дії користувача. Ці тести охоплюють класи презентаційного рівня, які відповідають за відображення графічних компонентів. Всі три категорії охоплюють різний набір класів і створюють різні тестові покриття. Рамка тестування Jacoco використовується для об'єднання звіту про випробувальне покриття для всього проекту.

Випадки використання Android реалізовані за допомогою "Дії та фрагменти". Діяльність - це основний компонент інтерфейсу користувача Android. Це компонент, який відображає графічні компоненти на екрані і дозволяє користувачеві працювати з цими компонентами. Фрагмент зазвичай забезпечує поведінку або частину інтерфейсу користувача для Activity. Компоненти інтерфейсу користувача, такі як кнопки, перегляди тексту та інші, розділені між активністю та фрагментами. Нижня панель навігації відображається у вікні "Діяльність", оскільки вона постійно залишається на екрані. Панель дій (верхня панель) також приєднана до активності, але меню параметрів змінюється відповідно до активного фрагмента. До фрагментів додаються інші компоненти інтерфейсу користувача. Окремі фрагменти містять такі графічні компоненти, як списки, перегляди тексту, редагування текстів, зображень і кнопок.

Для доступу до бази даних використовується Google Room. Room - це структура, розроблена компанією Google для реалізації шаблону об'єкта доступу до даних. Цей шаблон пропонує створити окремий об'єкт для виконання операцій створення, читання, оновлення, видалення для кожного об'єкта в базі даних. G також надає інструменти для оновлення версії бази даних та реалізації міграції даних.

Всі операції виконуються у фоновому потоці. Це необхідно, щоб уникнути замерзання або відставання в інтерфейсі користувача. Структура RxAndroid

використовується для реалізації механізму асинхронних викликів і синхронного виконання зворотних викликів. Цей фреймворк покладається на робочий потік для виконання всіх складних обчислень або операцій і повертається до основного потоку програми, коли виконується робота.

Техніка ін'єкції залежності широко використовується в реалізації проекту. Цей підхід визначає об'єкт, який надає всі інші системні об'єкти та компоненти з необхідними залежностями. Ін'єкція залежностей - це спосіб реалізації принципу Inversion of Control. Оскільки проект використовує багато абстракцій і класи залежать від контрактів (інтерфейсів), а не від конкретних реалізацій, об'єкти забезпечуються всіма залежностями зовнішнім компонентом. Об'єкти отримують залежності, які підходять контрактам, не спеціально знаючи про виконання цих контрактів. Dagger 2 - це структура, яка обробляє ін'єкції залежностей. Процесор Dagger отримує набір спеціальних класів, званих компонентами, які забезпечують реалізації контрактів, від яких залежать класи всередині системи. Потім процесор Dagger отримує спеціальні інтерфейси, звані модулями, які описують з'єднання компонентів кинджала з класами в програмній системі. Процесор Dagger автоматично генерує класи інжекторів для забезпечення програмної системи всіма необхідними залежностями від часу компіляції.

Графічний інтерфейс користувача реалізований з використанням XML-макетів. Один XML-файл містить опис графічних компонентів, які будуть відображатися на екрані. Він містить всі властивості і вказує ідентифікатори для всіх компонентів, щоб вони могли бути пізніше адресовані з коду. Butterknife - це структура, яка автоматично, використовуючи Java-анотації, пов'язує об'єкти Java з XML-описами графічних компонентів.

5 ОПИС ЕКСПЕРИМЕНТУ ТА ОЦІНКА РЕЗУЛЬТАТІВ

У розділі опису експерименту та оцінки результатів наведено детальний опис виконання експерименту. Реалізація експерименту складалася з набору випадків використання, які виконували основні операції в додатку Android. Кожен варіант використання був розроблений для представлення різних типів сценаріїв використання. Ці сценарії складаються з різної кількості простих дій. Деякі дії логічні і вимагають більше часу для виконання, інші прості і легкі. Деякі дії є чистими апаратними обчисленнями, інші - діями введення користувача. Ця глава також містить таблиці з вимірами, отриманими в експерименті. Ці дані аналізуються, щоб відповісти на дослідницькі питання. Архітектури оцінюються в контексті вимірювань і порівнюються з відповідними дослідницькими питаннями.

Програмні архітектури оцінюються як композиція з трьох властивостей. Такими властивостями є змінюваність, перевірочність і продуктивність. Показники змінюваності - це кількість класів, які потрібно додати для реалізації функції, кількість класів, які необхідно редагувати для реалізації функції, і ряд рядків коду, які необхідно додати або змінити для реалізації функції. Тестуваність описується метрикою тестового покриття. Показники продуктивності - це споживання пам'яті та час, необхідний для виконання сценарію використання.

Продуктивність є важливим показником для оцінки архітектури програмного забезпечення. Погана продуктивність призводить до того, що користувач має поганий досвід використання програмного забезпечення через можливі лаги та зависання в інтерфейсі користувача. Погана ефективність також призведе до тривалого очікування користувача, коли програма відкриває нові екрани або завантажує дані. Очікування відповіді після введення користувача також є наслідком поганої продуктивності. Ці дії розглядаються як основні і часті в роботі програмної системи. Таким чином, продуктивність повинна бути одним з пріоритетів у розробці гарної програмної архітектури.

Змінюваність - це здатність проекту оновлюватися. Вона включає як реалізацію нових функцій, так і виправлення помилок. Хороша модифікація призведе до меншого часу та ресурсів, витрачених на додавання нових функцій. Виправлення помилок в існуючому коді проекту неминуче і хороша модифікація полегшує пошук джерела помилок.

Тестуваність оцінюється за допомогою метрики тестового покриття. Цей показник показує, скільки компонентів системи може бути покрито тестами. Існує кілька типів тестів: модульні тести, інтеграційні тести та тести для інтерфейсу користувача. Всі ці тести реалізовані і комбінований звіт про випробування покриття є основою для оцінки тестованості.

5.1 Опис експерименту

Використання випадків для оцінки архітектур реалізовано в домені простого додатка приміток. Є можливості для створення, редагування та видалення нотаток, створення, редагування та видалення категорій нотаток і додавання приміток до категорій.

Наступні випадки використання (дужки містять список дій):

— створити нотатку (відкрити вікно створення нотатки, ввести назву, опис, вибрати категорію, натиснути кнопку «зберегти», створити нотатку в базі даних);

— редагувати нотатку (читати всі нотатки з бази даних, відкривати екран редагування нотатки, завантажувати нотатку з бази даних, змінювати назву, змінювати опис, натискати кнопку «зберегти», оновлювати примітку в базі даних);

— створити категорію (виберіть вкладку «категорії», відкрийте екран категорій створення, введіть назву, виберіть колір, натисніть кнопку «зберегти», створіть категорію в базі даних);

— категорія редагування (прочитати всі категорії з бази даних, відкрити екран категорій редагування, завантажити категорію з бази даних, змінити назву, змінити колір, натиснути кнопку «зберегти», оновити категорію в базі даних);

— видалити нотатку (прочитати всі нотатки з бази даних, відкрити вікно редагування, натиснути кнопку «видалити», підтвердити видалення в діалоговому вікні, видалити нотатку з бази даних);

— видалити категорію (вкладка відкритих категорій, прочитати всі категорії з бази даних, відкрити екран категорій редагування, натиснути кнопку «видалити», підтвердити видалення в діалоговому вікні, видалити категорію з бази даних);

— створити нотатку категорії (вкладка відкритих категорій, прочитати всі категорії з бази даних, вибрати категорію, відкрити екран створення нотатки, введіть назву, опис, натисніть кнопку «зберегти», створіть нотатку в базі даних);

— змінити нотатку категорії (вкладка відкритих категорій, прочитати всі категорії з бази даних, вибрати категорію, прочитати всі нотатки з бази даних, відкрити екран редагування записів, завантажити нотатку з бази даних, змінити назву, змінити опис, натиснути кнопку «зберегти», оновити ноту в базі даних);

— видалити нотатку з категорії (вкладка відкритих категорій, прочитати всі категорії з бази даних, вибрати категорію, прочитати всі нотатки з бази даних, відкрити вікно редагування, натиснути кнопку «видалити», підтвердити видалення в діалоговому вікні, видалити нотатку з бази даних);

— діалогове вікно показу інформації (натисніть кнопку «інформація», відкрийте діалогове вікно інформації).

Всі ці випадки використання складаються з менших кроків. Ці кроки є звичайними для різних випадків використання, а різні випадки використання - різні комбінації цих дій. Дії або взаємодія користувача, або доступ до інфраструктури додатків. Показники збираються для кожного з цих 10 випадків використання. Метрики - це продуктивність, змінюваність і перевірочність.

Стандарт ISO / IEC 25010 представляє модель якості програмного забезпечення, де продуктивність групується в три категорії; потенціал, часове

поведінка та використання ресурсів [7]. Продуктивність буде вимірюватися за допомогою Android Profiler, розробленого Google. Він був розроблений для вимірювання різних областей продуктивності для додатків Android, таких як CPU, GPU, пам'ять, акумулятор або мережа. Ці дані можна контролювати під час процесу його збору.

Продуктивність розраховується як набір двох вимірів, зібраних одночасно під час виконання випадку використання. Це час відгуку та споживання пам'яті. Аналогічний набір дій виконується для кожного випадку використання для обох архітектур. Всі набори дій будуть відповідати різним діям користувача. Кожен набір виконується і метрики збираються для обох архітектур 1000 разів, щоб зменшити помилку.

Тести продуктивності реалізовані за допомогою тестування Espresso. Швидкість введення та натискання елементів керування є постійною, оскільки вони виконувалися за допомогою автоматизованої структури. Швидкість малювання і завантаження є постійною, оскільки налаштування пристрою є постійним. Єдиним фактором, що впливає на час, є кількість операцій, які необхідно завантажити, що відрізняється для архітектур додатків.

З моделі якості програмного забезпечення ISO / IEC 25010 змінюваність визначається як "Ступінь, до якої продукт або система можуть бути ефективно і ефективно змінені без введення дефектів або погіршення існуючої якості продукції" [7]. Модифікація описується трьома основними вимірами. Для оцінки модифікованості архітектури проекту слід відповісти на наступні питання:

— Які класи потрібно модифікувати і скільки?

— Які нові класи потрібно додати і скільки?

— Скільки нових рядків коду (LOC) потрібно додати або видалити?

Ці показники будуть зібрані для кожної функції у випадках використання з архітектурами MVP та VIPER. Випадки використання поділяються на малі завдання, тому вони атомні і можуть представляти різні аспекти розробки мобільних додатків і створювати велику кількість зібраних значень, щоб краще зрозуміти глобальну тенденцію.

ISO / IEC 25010 визначає тестируемость як «Ступінь ефективності та ефективності, з якою можуть бути встановлені критерії випробувань для системи, продукту або компонента, і можуть бути проведені випробування для визначення того, чи були ці критерії виконані» [7]. Кілька років тому не було достатньо інструментів для реалізації хороших тестів для проекту мобільних додатків [11]. І це був справжній виклик, оскільки модульне тестування вже було однією з найважливіших частин процесу забезпечення якості. Але в останні роки стало доступним безліч інструментів для тестування з відкритим кодом, таких як Robotium [18] або Mockito [19].

Один із способів вимірювання тестованості структури коду - це метрика, яка називається тестовим покриттям коду. Він показує, які класи покриваються тестами, які методи і частини коду можна перевірити. Ці вимірювання агрегуються для всього проекту і можуть розглядатися як відсоток всієї кодової бази, що охоплюється тестами. Ця метрика має свої плюси і мінуси. Одне з його переваг полягає в тому, що він показує, наскільки добре структуровані класи та обов'язки розподілені між класами.

Високе тестове покриття означає, що архітектура проекту має низьку зв'язок, а класи мають певну роль у системі. Це призведе до кращого розуміння системою розробником і полегшить зміну системи. Це призводить до більшої ремонтопридатності.

Однак такий спосіб оцінки архітектури може ввести в оману сам по собі. Тести можуть не бути хорошими і можуть вводити в оману. Високий рівень тестового покриття як основна мета архітектури проекту призводить до більшої уваги до тестування, а не до досягнення бажаної поведінки.

І реалізація правильної поведінки архітектурних компонентів є головною метою будь-якої архітектури.

Покриття випробування збирають як поєднання одиниці випробувань і покриття ui-тесту. Блокові тести реалізовані за допомогою фреймворків Junit і Mockito.

Тести інтерфейсу використовуються в рамках тестування Espresso. Звіти об'єднані за допомогою інструмента Jacoco.

Для вимірювання модифікованості, всі випадки застосування розбиті на функції. Можливості є одиницями розробки програмного забезпечення. Кількість функцій є однією з метрик, що описує складність проекту. Ці функції наведено нижче:

— рівень інфраструктури (база даних, дао);

— доменний рівень (доменні моделі та абстракції);

— базова архітектура (моделі шарів і абстракції)

— головний екран (контейнер головного вікна)

— створювати нотатку (створювати екран нотаток, вставляти нотатки interactor, маршрутизатор);

— перегляд примітки (екран нотатки, читання нотатки взаємодії, маршрутизатор);

— редагувати нотатку (редагувати екран нотаток, оновити нотатку взаємодії, маршрутизатор);

— видалення нотатки (видалення нотатки взаємодії, маршрутизатора);

— перегляд списку всіх нотаток (екран списку нотаток, читання всіх нотаток взаємодії, маршрутизатора);

— створити категорію (створити екран категорій, вставити категорію interactor, маршрутизатор);

— категорія редагування (екран редагування категорій, взаємодіючий клас категорії оновлення, маршрутизатор);

— видалити категорію (видалити категорію взаємодія, маршрутизатор);

— встановити категорію для нотатки (елемент вибору категорії, оновити взаємодію нотатки);

— переглядати нотатки за категоріями (екран категорій, читати всі категорії взаємодію, маршрутизатор).

5.2 Оцінка результатів

Для кожної функції збираються метрики змінюваності. Ці метрики - це кількість класів, які необхідно створити для реалізації функції, кількість класів, які необхідно модифікувати, щоб реалізувати функцію, і кількість рядків коду, які необхідно додати або змінити для реалізації функції.

Таблиця 5.1 - Кількість класів, які необхідно створити для реалізації функції.

Feature	VIPER	MVP
Infrastructure layer	5 classes	5 classes
Domain layer	5 classes	5 classes
Base architecture	21 classes	24 classes
Main screen	8 classes	4 classes
Create note	9 classes	5 classes
View note	7 classes	7 classes
Edit note	2 classes	0 classes
Delete note	2 classes	0 classes
View list of all notes	10 classes	6 classes
Create category	9 classes	5 classes
Edit category	9 classes	5 classes
Delete category	2 classes	0 classes
Set category for a note	1 class	1 class
View notes by category	10 classes	6 classes

У таблиці 5.1 показано кількість класів, які необхідно додати для реалізації кожної особливості для обох архітектур.

Архітектура VIPER вимагає створення більшої кількості класів для реалізації кожної конкретної функції, а потім MVP

Таблиця 5.2 - Кількість класів, які потрібно редагувати для реалізації функції.

Feature	VIPER	MVP
Infrastructure layer	0 classes	0 classes
Domain layer	0 classes	0 classes
Base architecture	0 classes	0 classes
Main screen	4 classes	4 classes
Create note	7 classes	8 classes
View note	4 classes	6 classes
Edit note	4 classes	5 classes
Delete note	5 classes	6 classes
View list of all notes	5 classes	5 classes
Create category	5 classes	8 classes
Edit category	5 classes	6 classes
Delete category	4 class	5 classes
Set category for a note	6 class	6 class
View notes by category	8 classes	9 classes

Таблиця 5.2 показує кількість класів, які необхідно редагувати для реалізації кожної конкретної функції для обох архітектур. Архітектура VIPER вимагає редагування менших класів для реалізації кожної конкретної функції, ніж MVP. Це пояснюється більшою кількістю класів з чітко визначеними обов'язками та меншою зв'язком між компонентами.

Таблиця 5.3 - Кількість рядків коду, які потрібно додати або відредагувати для реалізації функції.

Feature	VIPER	MVP
Infrastructure layer	216 LOC	216 LOC
Domain layer	270 LOC	270 LOC
Base architecture	631 LOC	659 LOC
Main screen	242 LOC	185 LOC
Create note	356 LOC	336 LOC
View note	278 LOC	266 LOC
Edit note	102 LOC	74 LOC
Delete note	110 LOC	80 LOC
View list of all notes	293 LOC	267 LOC
Create category	336 LOC	300 LOC
Edit category	390 LOC	344 LOC
Delete category	51 LOC	35 LOC
Set category for a note	123 LOC	107 LOC
View notes by category	452 LOC	405 LOC

Таблиця 5.3 показує кількість рядків коду, які необхідно додати або відредагувати для реалізації кожної конкретної функції для обох архітектур. Архітектура VIPER вимагає в середньому написати на 10% більше рядків коду для реалізації кожної конкретної функції, ніж MVP. Повні таблиці зі списками класів можна подивитися в додатку.

Результати роботи були зібрані за допомогою автоматизованого тестування основи Espresso для Android. Швидкість введення та натискання елементів керування була незмінною, оскільки виконувалася за допомогою автоматизованої системи. Швидкість нанесення та завантаження була постійною, оскільки установка пристрою була постійною. Єдиним фактором, що впливає на час виконання і розподіл пам'яті, є кількість операцій, які потрібно було завантажувати, що відрізняється для архітектур додатків. Для вимірювання середньої величини було зроблено 1000 вимірювань, щоб переконатися, що помилка незначна. Всі вимірювання були розділені на два тести, які складалися з 500 разів виконання.

Таблиця 5.4 - Час виконання робіт.

Nexus 5X 8.1.0, time(ms)	Create note	Edit Note	Create category	Edit category	Create category note	Edit category note	Delete note	Delete category	Delete category note	Show info
Viper test 1	2920	2766	2320	2453	3554	2740	1401	1600	1824	1057
Mvp test 1	3142	3000	2479	2651	3774	2935	1534	1690	2011	1100
Viper test 2	2912	2726	2288	2519	3574	2762	1443	1622	1884	1095
Mvp test 2	3182	2992	2471	2711	3752	2905	1500	1688	2033	1186
Viper avg.	2916	2746	2309	2486	3564	2751	1422	1611	1854	1076
Mvp avg.	3162	2996	2475	2681	3763	2920	1517	1689	2027	1143

У таблиці 5.4 показано час виконання для кожного випадку використання, реалізованого обома архітектурами. Випадки застосування були виконані в стабільному середовищі 1000 разів, а таблиця показує середню. Це число тестових виконання вибирається для зменшення впливу можливих помилок або спайків продуктивності навколишнього середовища. Було проведено два тести, кожен з яких складався з 500 разів виконання.

Таблиця 5.5 - Виділення пам'яті для випадків використання.

Nexus 5X 8.1.0, memory (mb)	Create note	Edit Note	Create category	Edit category	Create category note	Edit category note	Delete note	Delete category	Delete category note	Show info
Viper test 1	120.9	105.3	86.6	103.5	129.0	104.5	80.5	83.9	85.3	59.8
Mvp test 1	112.1	121.2	110.1	112.7	142.1	120.0	88.2	100.1	99.8	66.0
Viper test 2	120.3	107.3	87.8	98.7	127.6	122.5	80.3	82.1	89.1	61.8
Mvp test 2	108.1	119.6	115.5	132.7	148.1	129.4	92.2	88.1	99.6	67.0
Viper avg.	120.6	106.3	87.2	101.2	128.3	113.5	80.4	83.0	87.2	60.8
Mvp avg.	110.1	120.4	112.8	122.7	145.1	124.7	90.2	94.1	99.7	66.5

Таблиця 5.5 показує виділення пам'яті для кожного випадку використання, реалізованого обома архітектурами. Виділення пам'яті було виміряно за допомогою профілю Android. Цей інструмент був реалізований для вимірювання експлуатаційних властивостей додатків.

Результати тестування можна побачити на рисунках 1 і 2 у додатку. Вони показують звіти про тестові покриття для проектів VIPER і MVP, які містять всі випадки використання. MVP та VIPER мають випробувальне покриття у 85%. Це означає, що VIPER і MVP добре перевірені, і немає ніякої різниці між цими двома архітектурами. Причина в тому, що обидві архітектури описують компоненти системи, які не повинні бути великими. Обидві ці архітектури мають на меті забезпечити високу когезію компонентів і хорошу настройку для модульного тестування.

Зібрані метрики про модифікацію показують, що архітектура VIPER вимагає створення більшої кількості класів, але менше класів для редагування для реалізації кожної функції. Кожен клас являє собою один компонент, описаний архітектурою. Компоненти можуть бути розділені на більше класів, але правило "один компонент - один клас" було розроблено для того, щоб зробити змінні змінні значущими. Це показує, що класи архітектури VIPER є меншими і менш складними. Це призводить до кращого розуміння класів.

Архітектура VIPER вимагає створення більшої кількості класів, оскільки існує більше різних компонентів з різними обов'язками. Кожна функція з новим екраном вимагає створення нових контрактів на перегляд, презентатор і маршрутизатор, а також реалізацію для них. Для кожної функції, що приносить до програми нові правила бізнес-логіки, необхідно створити новий договір взаємодії та реалізації. Нові функції також приносять зміни до існуючих маршрутизаторів.

Що стосується MVP, кожна нова функція з новим екраном потребує нових контрактів перегляду та презентації, а також реалізацій для тих, які будуть створені. Нові правила бізнес-логіки в основному вносять зміни до існуючих фасадів і не вимагають створення нових класів.

Однак архітектура MVP вимагає додавання менших рядків коду для кожної функції. Це означає, що для впровадження нової функції в архітектурі MVP потрібно менше часу. Це особливо важливо для проектів із строгими термінами і величезною кількістю робіт, які потрібно зробити, наприклад, стартапів.

VIPER вимагає більше рядків коду через більшу кількість класів і компонентів, що об'єднує створення архітектури. Кожен клас, окрім фактичних алгоритмів, описує залежності і відносини цього класу. Ці описи можуть повторюватися від класу до класу, і це відбувається тому, що загальна кількість рядків коду, в даному випадку, вище. MVP в шарі бізнес-логіки в основному спираються на великі фасади, які містять багато методів. Але залежність у цих фасадах зазвичай не повторюється.

Архітектура VIPER показує кращі результати роботи. Програма, побудована за допомогою архітектури VIPER, займає менше часу, щоб пройти через сценарії використання програм, ніж MVP. Архітектура VIPER потребує меншої кількості пам'яті, що виділяється для застосування операційною системою, ніж MVP.

Кращі результати роботи архітектури VIPER досягаються через менший розмір класів. Менші класи потребують менше пам'яті для завантаження в пам'ять. Бізнес-логіка, що виконується в сценаріях тестового використання, однакова для обох архітектур, тому кількість інструкцій також майже однакова. Але завдяки програмній архітектурі MVP, що має більш важкі класи, вони

потребують більше часу для завантаження в пам'ять. Це пояснює той факт, що сценарії використання тестових сценаріїв, розроблені з використанням архітектури програмного забезпечення MVP, потребують більше часу для виконання.

RQ1. Яка архітектура Android є більш доступною для обслуговування, MVP або VIPER?

Достовірність оцінюється як склад тестованості і модифікованості. Архітектура VIPER показала кращі результати у тестованості, але різниця була незначною. Архітектура MVP вимагала меншої кількості LOC для реалізації кожної функції в середньому на 10%.

Ряд рядків коду, які потрібно додати або змінити, є метрикою, яка менш важлива для змінюваності. Більше число класів у архітектурі програмного забезпечення VIPER, ніж у архітектурі MVP, насправді є гарною справою. Більша кількість класів означає, що вони мають менший набір обов'язків. Це означає, що знайти можливу помилку в системі програмного забезпечення простіше. Великі класи важче зрозуміти та дослідити.

Однак менша кількість рядків коду, які потрібно додати або змінити для реалізації функції, означає, що швидше реалізувати цю функцію таким чином. Архітектура програмного забезпечення MVP дозволяє команді розробників реалізувати проект швидше, ніж якщо б вони використовували архітектуру програмного забезпечення VIPER.

RQ2. Яка архітектура Android призводить до кращої продуктивності, MVP або VIPER?

Продуктивність оцінюється шляхом вимірювання часу, необхідного для завершення сценарію використання та споживання пам'яті. VIPER покращився з обох цих показників. VIPER показав кращі результати в обох вимірах продуктивності в тестовому середовищі. Тестова середовище - це пристрій Android із апаратними повноваженнями середнього діапазону. Ці результати можуть відрізнятися для інших пристроїв. Різниця менш значуща на більш потужних пристроях. Якщо основним ринком андроїд-додатків проекту є більш

потужні флагманські пристрої, то різниця в продуктивності не повинна бути основним критерієм для вибору архітектури. Однак, якщо основний ринок проекту андроїд-додатків також спирається на менш потужні пристрої, різниця в споживанні пам'яті може стати вирішальною для великих і складних додатків. В основному, різниця незначна, оскільки в основному мобільні додатки не великі, а останні Android-пристрої всіх цінових діапазонів мають потужне обладнання.

RQ3. Яка архітектура Android краще підходить для різних типів проектів, MVP або VIPER?

Архітектура MVP вимагає менше ліній коду, що пишуться, ніж VIPER, тому розробнику потрібно менше часу для завершення програми. Це також призводить до зниження вартості. Архітектура VIPER демонструє кращу продуктивність і перевірочність, тому ця архітектура буде кращою, якщо продуктивність і якість програми є більш важливими, ніж вартість і час.

Архітектура MVP буде кращою, ніж VIPER для малих проектів, оскільки різниця в продуктивності не буде видно, але витрати і час споживання будуть нижчими. Архітектура VIPER буде кращою, ніж MVP для середніх і великих масштабних проектів, оскільки продуктивність буде важливою проблемою, коли буде виконано багато бізнес-логіки. Архітектура VIPER призводить до більш чистого коду, ніж MVP, оскільки класи менші, а обов'язки розділені краще між класами. Архітектура VIPER краще підійде для модульних додатків, оскільки ці програми великі, мають складну структуру і багато в чому залежить від продуктивності. Архітектура MVP краще підійде для бібліотек, оскільки результуючий розмір бібліотеки є важливим, а проект MVP матиме менше рядків коду.

Розмір є важливим для бібліотек, оскільки проекти зазвичай використовують багато з них, і ці проекти намет стають дуже важкими, через великі розміри використовуваних рамок.

5.3 Підсумки

У цьому розділі наведено відомості про експеримент. Вона починається з опису показників, які необхідно зібрати, і мотивує цей вибір. Ці показники - це змінюваність, тестованість і продуктивність. Ці метрики збираються для обох архітектур на сценаріях використання тестів. Тестові сценарії використання виконуються в стабільному середовищі. Перша частина цього розділу присвячена поясненню деталей експерименту. Він містить опис усіх тестових сценаріїв використання. Потім він описує вибрані метрики та пояснює спосіб збирання цих показників.

Друга частина розділу містить результати експерименту. На ній наведені таблиці з результатами вимірювань. Архітектура VIPER показала кращі результати з обох показників ефективності. Тестуваність архітектури MVP і VIPER знаходиться на одному рівні. Результати модифікації показали, що як MVP, так і VIPER мають переваги в різних вимірах. Потім на запитання на відповіді на основі результатів вимірювань. Результати показують, що обидві архітектури є життєздатними і краще підходять для різних типів проектів розробки програмного забезпечення.

ВИСНОВКИ ТА МАЙБУТНІ ДОСЛІДЖЕННЯ

Дослідницька робота має на меті визначити, чи краще нова архітектура VIPER, ніж більш консервативна архітектура MVP. Для досягнення цієї мети починається вивчення основних концепцій проектування архітектури та вивчення структури обох архітектур для визначення подібності та відмінностей.

Огляд літератури показав, що дослідження з питань порівняння архітектури андроїда є присутнім. Дане дослідження описує різні підходи до порівняння архітектур. Ці підходи спрямовані на оцінку архітектури в декількох ключових показниках, а потім аналізують результати.

Порівняння архітектури починається з визначення трьох основних властивостей. Такими властивостями є ремонтопридатність, продуктивність і перевірочність. Потім кожне властивість пояснюється шляхом встановлення критеріїв для оцінки якості. Зрештою, всі вимірювання порівнюються і аналізуються.

Оцінка змінюваності складається з трьох ключових вимірювань. Ряд класів, які необхідно створити для реалізації функції, кількість класів, які необхідно модифікувати для реалізації функції, і кількість рядків коду, які необхідно додати або змінити для реалізації функції, є вимірами для оцінки модифікованості. VIPER вимагав створити більше класів для розробки функцій, тоді як менше класів було змінено, ніж архітектура MVP. Для реалізації функції потрібно додавати або змінювати на 10% більше рядків коду.

Споживання пам'яті та час, необхідний для запуску сценарію використання, є двома вимірами для оцінки продуктивності архітектури. Виконання тестових сценаріїв було повністю автоматизованим, щоб забезпечити, щоб інші фактори, крім архітектурних, не впливали на результати. VIPER показав кращі результати за обома вимірами. Використання випадків, реалізованих за допомогою архітектури MVP, вимагало більше часу для виконання і витрачання більшої кількості пам'яті.

Тестуваність оцінюється за допомогою вимірювального тестового покриття. Покриття випробувань вимірюють, поєднуючи як випробування на пристрій, інтеграцію та інтерфейс користувача. Архітектура VIPER і MVP показали однакові результати.

6.1 Висновки

У підсумку обидві архітектури є життєздатними. Для обох архітектур було зібрано набір метрик, що описують їх властивості, такі як змінюваність, тестуваність, ремонтопридатність і продуктивність. Ці показники показали, що одна архітектура має кращу в одній метриці, але гірше в іншій. Це означає, що не існує абсолютного найкращого рішення для кожної ситуації, і архітектура повинна завжди вибиратися відповідно до потреб і обмежень проекту. Ці обмеження можуть стосуватися вартості або часу розробки, виконання програми або якості коду. Проекти також можуть мати дуже різну природу, наприклад, мобільний додаток з інтерфейсом користувача, або деякі рамки для мобільного розвитку. Всі ці відмінності впливають на те, яку архітектуру вибрати для проекту.

Ця дисертація надає інформацію для двох популярних архітектур андроїд-проекту MVP і VIPER. Вибір правильної архітектури для проекту буде легше, коли на основі цього дослідження. Це особливо корисно в промисловості, оскільки кількість нових андроїд-додатків, фреймворків і бібліотек постійно зростає.

Результати цього дослідження не застосовні до інших областей розробки програмного забезпечення. Але методи, розроблені в цій дисертації для порівняння двох мобільних архітектур, можуть бути використані для порівняння будь-яких інших архітектур, не тільки для мобільних додатків, але й для інших

типів проектів. Цей метод може бути використаний для створення еталону мобільних архітектур.

Яка архітектура Android є більш доступною для ремонту, MVP або VIPER? Зібрані показники показують, що MVP вимагає додавання менших класів для реалізації нової функції, але більше класів для редагування. Різниця в метриці тестованості була незначною. Результати показують, що на це питання немає прямої відповіді, і всі ці показники слід враховувати при виборі архітектури.

Яка архітектура Android призводить до кращої продуктивності, MVP або VIPER? Архітектура VIPER має кращі результати в обох показниках продуктивності. Це означає, що архітектура Viper приводить до кращої продуктивності, ніж архітектура MVP.

Яка архітектура Android краще підходить для різних типів проектів, MVP або VIPER? Архітектура MVP краще підходить для невеликих масштабних проектів і бібліотек. Архітектура VIPER надає переваги для середніх і великих проектів, а також для модульних проектів.

6.2 Майбутні дослідження

Майбутня робота може включати більше показників архітектури для порівняння. Існують метрики, які можуть представляти злиття між класами і зв'язок між класами. Ці метрики також можуть бути розраховані для модулів. Порівняння цих показників може дати більше інформації про предмет ремонтопридатності та складності проекту, який реалізується за допомогою специфічної архітектури.

Можна було б більше застосувати випадки для розслідування архітектур. Подальша робота може включати порівняння архітектури на дуже великих і складних випадках використання.

Збір метрик, згаданих у цій тезі, на великих комерційних проектах дозволить підвищити точність результатів. Це також дозволить створити стабільний тест для архітектур Android.

Вимірювання продуктивності на великій кількості пристроїв Android значно покращить розуміння обох архітектур і допоможе визначити способи підвищення продуктивності цих архітектур. Це також може бути крок до створення нової архітектури, яка вирішить проблеми MVP і VIPER і сильно вплине на дизайн Android-додатків.

Архітектура Android є відносно новим і активним полем знань, тому нові архітектури з'являться кожні кілька років. Порівняння архітектури буде проблемою, і методи, описані в цій тезі, можуть бути корисними. Порівняння нових архітектур з більш консервативними є точкою майбутніх досліджень.

ПЕРЕЛІК ПОСИЛАНЬ

1. Tian Lou. A Comparison of Android Native App Architecture – MVC, MVP and MVVM // Helsinki: Aalto University, Master's Thesis, 2016.

2. Anthony Madhvani. Structuring Android Applications // Kortrijk: The Hogeschool West Vlaanderen College, Bachelor's Thesis, 2016.

3. Hugo Kallstrom. Increasing Maintainability of Android Applications // Umea : Umea University, Department of Computing Science, Master's Thesis, 2016.

4. Jussi Koskinen. Software maintenance costs // Jyvaskyla: University of Jyvaskyla, 2010.

5. Douglas Gilstrap. "Ericsson mobility report on the pulse of the networked society," // Ericsson, 2016. URL: https://www.slideshare.net/EricssonLatinAmerica/ericsson-mobility-report-november-2016. (дата звернення: 25.01.2018).

6. Paul Clements, Rick Kazman, and Mark Klein // Evaluating software architectures. Boston: Addison-Wesley, 2003.

7. International Organization of Standardization // "ISO/IEC 25010:2011", /International Organization of Standardization, 2011. URL: https://www.iso.org. (дата звернення: 20.01.2018).

8. Robert C Martin. Clean code: a handbook of agile software craftsmanship. London: Pearson Education, 2009.

9. A. Patidar and U. Suman, "A survey on software architecture evaluation methods" in Computing for Sustainable Global Development (INDIACom). 2nd International Conference on, Indore, 2015.

10. L. Chen, M. A. Babar and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," in IEEE Software, vol. 30, no. 2, pp. 38 - 45, 2013.

11. M. E. Joorabchi, V. B. C. Univ. of British Columbia, A. Mesbah and P. Kruchten, "Real Challenges in Mobile App Development," // Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, 2013.

12. M. Mattsson, H. Grahn and F. Mårtensson, "Software architecture evaluation methods for performance, maintainability, testability, and portability," // Second International Conference on the Quality of Software Architectures, 2006.

13. Douglas Gilstrap. “Ericsson mobility report on the pulse of the networked society,” // Ericsson, 2015. URL: https://www.slideshare.net/Ericsson/ericsson-mobilityreportjune2015. (дата звернення: 25.01.2018).

14. Mahfuzul Huda, Dr. Y.D.S. Arya, and Dr. M.H. Khan. “Measuring testability of object oriented design: A systematic review,” // International Journal of Scientific Engineering and Technology vol. 3 no.10, pp1313-1319, 2014.

15. Himanshu Shewale, Sameer Patil, Vaibhav Deshmukh, Pragya Singh. Analysis of Android vulnerabilities and modern exploitation techniques // ICTACT journal on communication technology, march 2014.

16. M. Rizwan Jameel Qureshi, Fatima Sabir. “A comparison of model view controller and model view presenter,” // Science International (Lahore), vol. 25(1), pp. 7-9, 2013.

17. Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, Muhammad Ali Babar. “10 years of software architecture knowledge management: Practice and future,” // Journal of Systems and Software, vol. 116, pp. 191-205, June 2016.

18. GitHub, Inc. “Robotium Android testing framework.” URL: https://github.com/RobotiumTech/robotium. (дата звернення: 25.01.2018)

19. Szczepan Faber, Brice Dutheil, Rafael Winterhall. “Mockito Android testing framework.” URL: http://site.mockito.org/. (дата звернення: 25.01.2018)

20. Google LLC, “Official Android dashboards.” URL: https://developer.android.com/about/dashboards/. (дата звернення: 25.02.2018)