Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Модель мобільного застосунку iOS з додаванням базових сервісів та розширень
(тема)

Виконав:

студент II курсу, групи КСМм-20-1

Оцевик В.А.
(прізвище, ініціали)

Спеціальність

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

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

Освітня програма

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

Керівник: доц. Філімончук Т.В.
(посада, прізвище, ініціали)

Допускається до захисту

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

2021 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

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

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

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

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

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Оцевику Владиславу Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Модель мобільного застосунку iOS з додаванням базових сервісів та розширень

затверджена наказом по університету від “ 05 ” листопада 2021 р. № 1656 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 13 грудня 2021 р.

3. Вхідні дані до роботи архітектурний шаблон – MVVM; патерн програмування – Coordinator; мова програмування – Swift; платформа – iOS; середовище розробки – Xcode.

4. Перелік питань, що потрібно опрацювати у роботі

1. Переваги та недоліки існуючих архітектур програмного забезпечення
2. Огляд інструментарію для розробки мобільного застосунку iOS
3. Принципи об’єктно-орієнтованого програмування SOLID
4. Складові моделі мобільного застосунку iOS
5. Огляд сервісів та розширень, які доцільно додати до мобільного застосунку iOS

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 11 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд архітектур програмного забезпечення	09.11.21-10.11.21	
2	Огляд інструментарію для розробки	11.11.21-12.11.21	
	мобільного застосунку iOS		
3	Огляд принципів об'єктно-орієнтованого	13.11.21-18.11.21	
	програмування SOLID		
4	Розробка моделі мобільного застосунку iOS	19.11.21-22.11.21	
5	Перелік базових сервісів та розширень, які	23.11.21-03.12.21	
	доцільно додати до мобільного застосунку iOS		
6	Оформлення матеріалів кваліфікаційної роботи	04.12.21-07.12.21	
7	Подання кваліфікаційної роботи керівникові	08.12.21-09.12.21	
	та її попередній захист		
8	Подання роботи на рецензування	10.12.21-11.12.21	

Дата видачі завдання 08 листопада 2021 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Філімончук Т.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 64 с., 10 рис., 1 дод., 20 джерел.

МОБІЛЬНА РОЗРОБКА, ШАБЛОН ЗАСТОСУНКУ, АРХІТЕКТУРА, ПАТЕРНИ, SOLID, IOS, СЕРВІСИ, ТЕСТУВАННЯ

Метою кваліфікаційної роботи є розробка моделі мобільного застосунку під платформу IOS для побудови комерційних додатків з надійною та легко масштабованою архітектурою.

У ході виконання кваліфікаційної роботи було розроблено модель мобільного застосунку IOS з додаванням базових сервісів та розширень. Модель, що розроблено, призначена для спрощення розробки мобільного додатку. Вона відповідає рекомендаціям Apple, що до розробки мобільних додатків під платформу IOS, та включає в себе використання патернів та сучасних засобів розробки.

ABSTRACT

Master’s thesis: 64 pages, 10 figures, 1 appendice, 20 sources.

MOBILE DEVELOPMENT, APPLICATION TEMPLATE, ARCHITECTURE, PATTERNS, SOLID, IOS, SERVICE, TESTING.

The major goal of this thesis is to develop a model of mobile application for the IOS platform to build commercial applications with a reliable and easily scalable architecture.

During the certification work, a model of the iOS mobile application was developed with the addition of basic services and extensions. The developed model is designed to simplify the development of a mobile application. The model meets Apple’s recommendations for the development of mobile applications for the iOS platform, and includes the use of patterns and modern development tools.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – інтерфейс прикладного програмування (англ, Application Programming Interface)

DI – стиль налаштування об’єкта (англ, Dependency Injection DI)

IB – додаток від Apple для операційної системи Mac OS (англ, Interface Builder)

MVVM – архітектурний шаблон (англ, Model-View-ViewModel)

SPM – інструмент для керування залежностями проекту написаного на Swift (англ, Swift Package Manager)

UIKit – фреймворк, що забезпечує необхідну інфраструктуру для програм iOS або tvOS

Xcode – інтегроване середовище розробки (IDE) програмного забезпечення для платформ macOS, iOS, watchOS та tvOS, розроблене корпорацією Apple

ВСТУП

Важливість мобільних телефонів у нашому повсякденному житті та діяльності, безперечно, безмежна. Це пов’язано з тим, що зараз відбувається величезна трансформація в тому, що мобільні телефони більше не є звичайним пристроєм зв’язку, яким вони були раніше. Завдяки різноманітним неймовірним функціям та можливостям, які пропонують мобільні телефони, він став колосальною точкою уваги як для окремих осіб, так і для компаній. Сукупний прогрес мобільних технологій, доступність і доступ до високошвидкісного інтернету та чудовий комунікативний інтерфейс у цих пристроях призводять до цілого рівня нового та інноваційного досвіду мобільних обчислень. Це стало можливим завдяки розробці мобільних додатків.

Модель мобільного застосунку – це каркас мобільного застосунку, який зобов’язує будувати додаток відповідаючи певній архітектурі.

Архітектура мобільного додатка є найважливішою частиною розробки мобільних додатків. Архітектура мобільного додатка – це набір різних структурних елементів та їхніх інтерфейсів, на основі яких можна розробляти добре структуровані та зручні мобільні додатки. Перш ніж приступити до написання коду, важливо отримати чітке уявлення про те, які функції повинні бути вбудовані в програму, щоб вона відповідала потребам клієнтів.

Усі ресурси, вкладені в створення бездоганної мобільної архітектури, неодмінно окупляться в майбутньому. Розробники зможуть ефективно виправляти неполадки, легко масштабувати програму та швидко додавати плагіни. Такі принципи розробки програмного забезпечення, як SOLID, KISS та DRY, слід враховувати при розробці архітектури програми, щоб уникнути проблем у майбутньому та досягти найкращих результатів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Розробка програмного забезпечення – це як будівництво будинку. Перше і фундаментальне, що потрібно зробити – це закласти міцний фундамент. Довговічність та надійність, серед багатьох інших речей, значною мірою залежать від цієї основи. У розробці мобільних додатків модель, по якій його буде спроектовано, відіграє ключову роль та впливатиме на розмір, структуру, масштабованість, ремонтопридатність проекту та значною мірою відзначається на часі, необхідному для розробки застосунку.

Модель додатку – це набір патернів, які будуть застосовуватись під час проектування та розробки додатку [1]. Патерни відрізняються за рівнем складності, деталізації та охоплення системи, що проектується. Самі низькорівневі та прості патерни – це ідіоми, але вони не універсальні, оскільки застосовні тільки в рамках однієї мови програмування.

Найуніверсальніші – це архітектурні патерни, які можна реалізувати практично на будь-якій мові. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Немає універсального правила для розробки мобільних додатків, тому і немає універсальної основи. Це пов’язано з тим, що на розробку додатку впливає безліч факторів:

- визначення типу пристрою;
- простота використання;
- сценарії пропускної здатності;
- інтерфейс користувача;
- правильний метод навігації;
- оновлення в режимі реального часу.

Визначення типу пристрою – це також важливий фактор. Існують різні категорії смартфонів, які потрібно визначити та врахувати. Окрім категорії або типу пристрою, слід звернути увагу на багато інших речей: розмір екрану, роздільну здатність та характеристики процесора. Крім того, слід

також врахувати ємність сховища та наявність фреймворка інструментів розробки.

Фактор простоти використання пов’язаний з подальшою підтримкою мобільного додатка, тобто обов’язково завжди враховувати вартість його підтримки. Варто відзначити, що кращий код – це код, який ніколи не був написаний. І чим менше у проекті коду, тим менше помилок може виникнути. Також важливо враховувати сценарії пропускної спроможності цільової аудиторії мобільного додатку. Можуть бути моменти, коли зв’язок буде нульовим.

Під час створення мобільного додатку мережа цільової аудиторії є важливим фактором: якщо додаток працює повільно – вони відмовляться від нього. Потрібно враховувати і споживання електроенергії облікового запису, і механізм доступу до проектування, і швидкість. Крім того, слід обирати найкращі програмні протоколи та обладнання для мобільного додатка. Тобто потрібно налаштувати всі ці речі для повільного та переривчастого підключення до Інтернету.

Для мобільних додатків важливу роль відіграє інтерфейс користувача. Зручний та продуманий інтерфейс користувача дозволяє легко та комфортно взаємодіяти з додатком. Інтерфейс програми повинен бути простим та креативним відповідно до попиту цільової аудиторії [2].

Вибір правильного методу навігації є важливим та вирішальним. Це важливий аспект розробки мобільних додатків. Метод навігації повинен відповідати вимогам програми та уподобанням клієнтів. Вибір найкращого варіанту після аналізу різних методів навігації може привести до успіху. Деякі з популярних методів навігації включають складну панель навігації, контролер вкладок та модальний контролер. Навігація на основі жестів заохочує користувача використовувати комбінації пальців. За допомогою цього методу користувач може викликати дію на екрані.

При розробці мобільного додатку також слід вирішити, потрібно створювати оновлення в режимі реального часу або натискаючи кнопку.

Розробник може вирішити цей фактор, пам’ятаючи про свою аудиторію. Оновлення в режимі реального часу може коштувати дорого, але це переконлива функція.

Визначають чотири основні типи розробки мобільних додатків: MVC, MVP, MVVM та Viper. Кожен з них, як правило, має піддерево варіацій використання. Та навіть при найкращому підборі технології до певного проекту вони не завжди задовольняють потребам при розробці. Їх архітектура може бути основою для побудови більш універсальної та гнучкої моделі застосунку [3].

Зважаючи на все вищезазначене, можливо згенерувати вимоги, які слід враховувати при розробці моделі мобільного додатку:

- розділення проблем: цей принцип означає, що різні частини коду повинні бути розділені. Це допомагає при повторному використанні цих сутностей, систематизації розробки, легкому налагодженню та ізоляції частин, які можуть у подальшому часто змінюватись, щоб вони не впливали на інші;

- тестування: корисно писати тестові приклади та мати можливість окремо тестувати функціональні можливості. Таким чином розробник уникне виявлення проблем під час застосування додатку, що зазвичай означає виправлення на тривалий час;

- надійність: добре підібрана архітектура має вирішальне значення для створення стабільних програм без серйозних суперечностей, оскільки вона визначає, як частини коду взаємодіють між собою;

- масштабованість: додаток повинен мати міцну основу для накопичення нових функцій та змін відповідно до потреб бізнесу. Він також має бути придатним для майбутніх реконструкцій (наприклад, нових бібліотек, операційних систем) технологій програмування;

- налагодження та простота використання: хороша архітектура спрощує написання та читання коду. Це також може призвести до низьких витрат на обслуговування.

2 АРХІТЕКТУРА, ЯК ОСНОВНИЙ КОМПОНЕНТ МОДЕЛІ ДОДАТКУ

2.1 Поняття та цілі архітектури програмного забезпечення

Архітектура системи описує її основні компоненти, їх взаємозв’язки (структури) і те, як вони взаємодіють один з одним. Архітектура та дизайн програмного забезпечення (ПЗ) включають декілька факторів: бізнес-стратегію, атрибути якості, людську динаміку, дизайн та IT-середовище (рисунок 2.1).

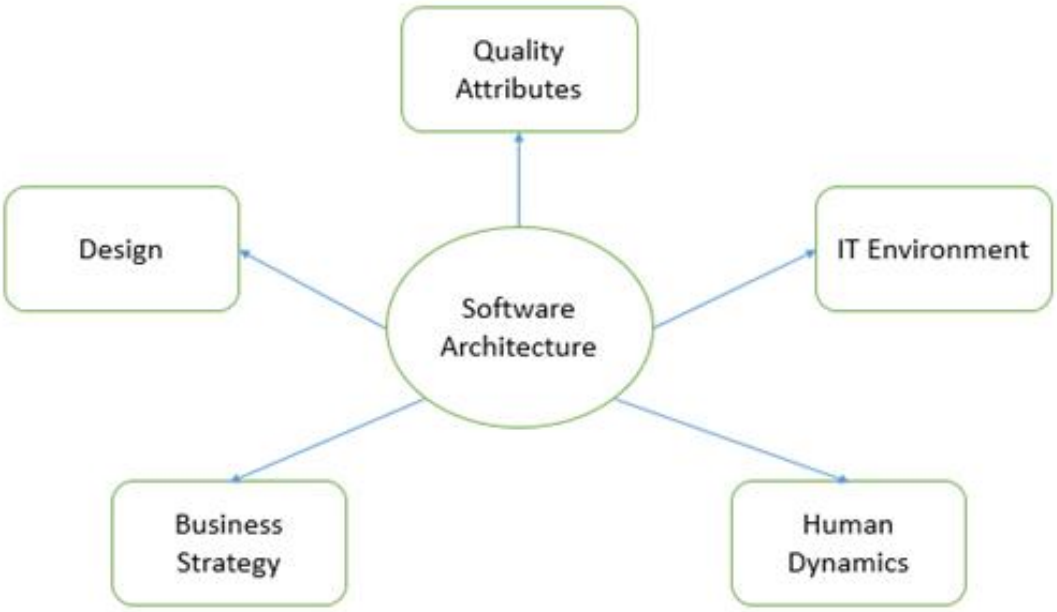

Рисунок 2.1 – Схема архітектури додатку

Можливо розділити архітектуру та дизайн програмного забезпечення на дві окремі фази: архітектуру програмного забезпечення та проектування програмного забезпечення. В архітектурі нефункціональні рішення складаються та відокремлюються функціональними вимогами. У проектуванні виконуються функціональні вимоги.

Архітектура служить планом системи, вона забезпечує абстрагування для управління складністю системи, встановлення механізму зв’язку та координації між компонентами. Також вона визначає структуроване рішення,

що відповідає всім технічним та експлуатаційним вимогам, одночасно оптимізуючи загальні якості, такі як продуктивність та безпека. Архітектура передбачає набір важливих рішень щодо організації, що пов’язано із розробкою програмного забезпечення, і кожне з цих рішень може мати значний вплив на якість, продуктивність та загальний успіх кінцевого продукту (рисунок 2.2).

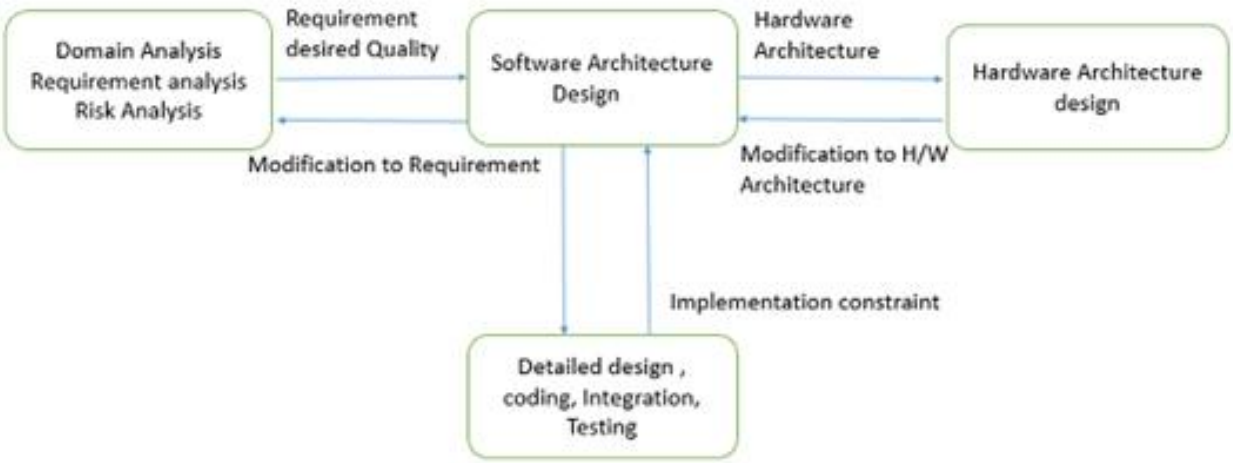

Рисунок 2.2 – Схема розробки архітектури застосунку

Розробка програмного забезпечення забезпечує план проектування, який описує елементи системи: як вони підходять та працюють разом, щоб виконати вимоги системи. Цілі створення плану проектування наступні:

- обговорити системні вимоги та визначити очікування клієнтів, маркетологів та управлінського персоналу;

- виступати в якості плану під час процесу розробки додатку;

- розробка документації з виконання окремих задач, які було оговорено, включаючи детальне проектування, кодування, інтеграцію та тестування.

Складати такий план слід обов’язково і тому, що в подальшому це заощадить час на виправлення помилок. Також на попередньому етапі слід здійснити аналіз предметної галузі, вимог та ризиків. І тільки після того необхідно переходити до детального проектування, кодування, інтеграції та тестування [4].

Основною метою архітектури є визначення вимог, які впливають на

структуру програми. Добре продумана архітектура зменшує бізнес-ризики, що пов’язані зі створенням технічного рішення, і будує міст між бізнесом та технічними вимогами.

Архітектура програмного забезпечення має такі обмеження:

- відсутність інструментів та стандартизованих засобів представлення архітектури;
- відсутність методів аналізу для прогнозування того, чи призведе архітектура до реалізації, яка відповідає вимогам клієнтів;
- недостатня поінформованість щодо важливості архітектурного проектування для розробки програмного забезпечення;
- нерозуміння ролі розробника програмного забезпечення та нечітка комунікація між зацікавленими сторонами;
- відсутність розуміння процесу проектування, досвіду проектування та оцінки дизайну.

2.2 Аналіз існуючих архітектур

При створенні мобільних або веб-додатків, можливо, доведеться зіткнутися з різними архітектурними шаблонами дизайну, залежно від типу та поведінки проекту. Перед запуском проекту завжди потрібно визначити найкращу архітектуру, яка залежить від вимог проекту та її поведінки. Для цього треба знати різні шаблони проектування та архітектури.

Розглянемо деякі із них: Model-View-Controller (MVC) , Model-View-Presenter (MVP) , Model-View-ViewModel (MVVM), VIPER (View-Interactor-Presenter-Entity-Router) та ін.

Архітектура дуже важлива для проекту. Вона допомагає мінімізувати дублювання коду, запобігає сильному зв’язуванню та стандартизує загальний засіб написання коду, який забезпечує загальне рішення для ситуації, що повторюється при розробці програмного забезпечення [5].

2.2.1 Архітектура Model-View-Controller

Model-View-Controller (MVC) – це найпопулярніша архітектура додатку. ЇЇ рекомендує компанія Apple тому що вона зрозуміла та легка у застосуванні. У більшості прикладів коду на офіційних веб-сайтах Apple або Swift програми створюються з використанням архітектури MVC. Це архітектурний патерн додатку за замовчуванням, який використовується у всіх проектах iOS, MacOS та watchOS.

MVC – це засіб організації коду, який передбачає виділення блоків, що відповідають за вирішення різних задач. Один блок відповідає за дані програми, інший відповідає за зовнішній вигляд, а третій контролює роботу програми.

Компонентами MVC виступають наступні складові (рисунок 2.3):

- модель (model) – компонент, який відповідає за дані, а також визначає структуру програми;
- подання (view) – компонент, який відповідає за взаємодію з користувачем. Тобто код компонента view визначає зовнішній вигляд програми та засоби її використання;
- контролер (controller) – компонент, який відповідає за зв’язок між model та view. Код компонента controller визначає, як реагує сайт на дії користувача. По суті, це мозок MVC-програми.

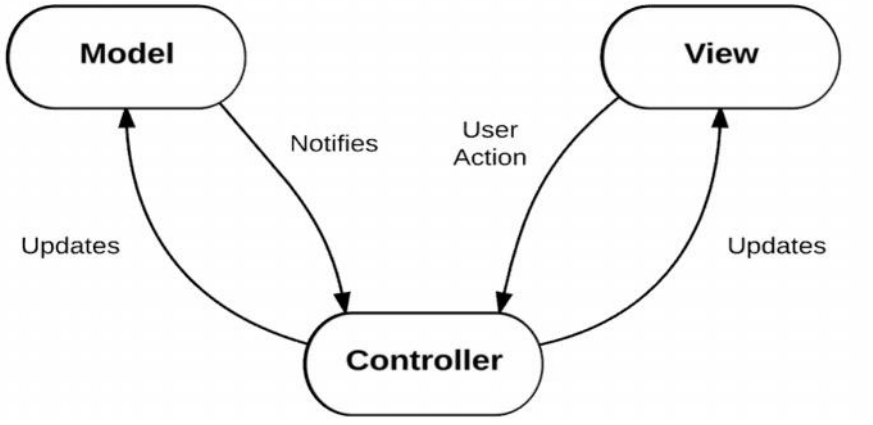

Рисунок 2.3 – Структура MVC

Додаток запускається з ініціалізації контролерів представлення верхнього рівня. Потім ці контролери завантажують та налаштовують представлення з даними із моделі. Контролер може або створювати власні моделі явно, або мати доступ до них. Якщо кільком контролерам потрібно отримати доступ до однієї моделі, можна використовувати UIDocument, щоб зробити це простіше.

Контролер в архітектурі MVC відповідає за багато речей. Він отримує всі подання з представлення, обробляє всю логіку взаємодії з користувачем, відправляє зміни до моделі, отримує зміни від моделі і, нарешті, застосовує ці зміни до представлення. Це призводить до того, що файли контролера дуже великі і їх важко розбити на частини в моделі MVC. Тому вважається, що це є недоліком використання архітектури MVC.

MVC – це підхід до проектування програми, який передбачає виділення коду в блоки: модель, представлення та контролер. Контролер обробляє запити, що входять. Модель дістає з бази даних інформацію, необхідну для виконання конкретних запитів. Подання визначає результат запиту, який отримує користувач [6].

2.2.2 Архітектура Model-View-Presenter

Model-View-Presenter (MVP) – шаблон проектування, похідний від MVC, який використовується в основному для побудови інтерфейсу користувача.

Елемент Presenter в даному шаблоні бере на себе функціональність посередника (аналогічно контролеру в MVC) і відповідає за управління подіями інтерфейсу користувача (наприклад, використання миші).

Модель MVP складається із наступних компонентів (рисунок 2.4):

- View відповідає за відображення інтерфейсу та керування макетом, а також надає інтерфейс для оновлення представлення та збору даних у Presenter;

- Presenter відповідає за отримання подання від View, оновлення подання через інтерфейс, що надається View, та керування моделлю;

- Model забезпечує модель даними та здійснює маніпулювання даними.

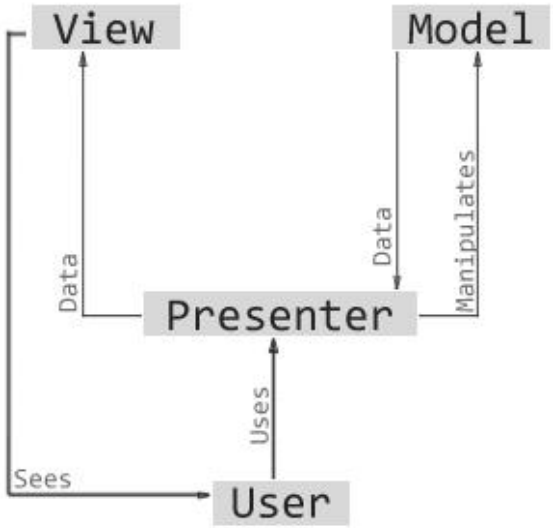

Рисунок 2.4 – Структура MVP

В iOS UIView та UIViewController об’єднуються, щоб сформувати подання в MVP. UIView відповідає за відображення елементів, UIViewController відповідає за макет та композицію інтерфейсу та перенаправляє події в Presenter. Тому в MVP бізнес-логіка розміщується в Presenter, який відповідає за координацію уявлення та моделі. Через поділ View статус Presenter є контрольованим, і він менш сприйнятливий до зовнішніх впливів під час тестування.

Використовувати MVP в iOS дуже просто. Недоліком є те, що додатковий рівень інтерфейсу для ізоляції збільшуватиме обсяг коду.

2.2.3 Архітектура Model-View-ViewModel

Патерн MVVM (Model-View-ViewModel) дозволяє відокремити логіку програми від візуальної частини (подання) та складається з трьох компонентів (рисунок 2.5): моделі (Model), моделі уявлення (ViewModel) та уявлення (View).

Цей патерн був представлений Джоном Госсманом у 2005 році як модифікація шаблону Presentation Model і був спочатку націлений на розробку додатків WPF. І хоча зараз цей патерн вийшов за межі WPF і застосовується в різних технологіях, у тому числі при розробці під Android, iOS, проте WPF є досить показовою технологією, яка розкриває можливості даного патерну.

Модель описує дані, що використовуються в додатку. Моделі можуть містити логіку, безпосередньо пов’язану з цими даними, наприклад, логіку валідації властивостей моделі. У той же час модель не повинна містити жодної логіки, пов’язаної з відображенням даних та взаємодією з візуальними елементами керування.

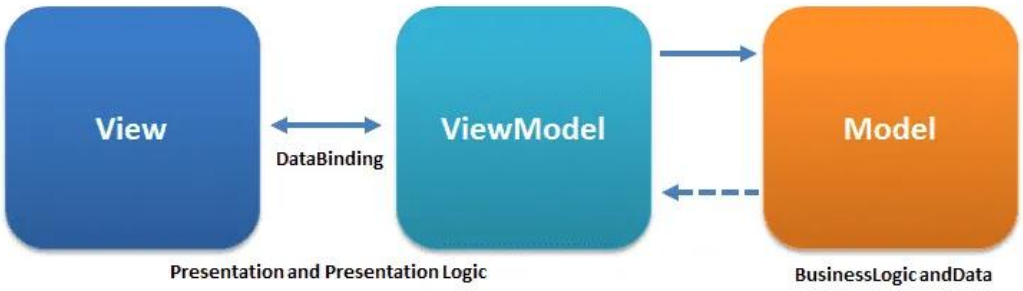

Рисунок 2.5 – Структура MVVM

Нерідко ця модель реалізує інтерфейси INotifyPropertyChanged або INotifyCollectionChanged, які дозволяють повідомити систему про зміни властивостей моделі. Завдяки цьому полегшується прив’язка до подання, хоча знову ж таки – пряма взаємодія між моделлю та поданням відсутня.

View або подання визначає візуальний інтерфейс, через який користувач взаємодіє з програмою. Стосовно WPF уявлення – це код xaml, який визначає інтерфейс у вигляді кнопок, текстових полів та інших візуальних елементів. Подання не обробляє події за рідкісним винятком, а виконує дії в основному за допомогою команд.

ViewModel або модель подання пов’язує модель та подання через механізм прив’язування даних. Якщо в моделі змінюються значення

властивостей, при реалізації моделлю інтерфейсу NotifyPropertyChanged, автоматично йде зміна даних, що відображаються в поданні, хоча безпосередньо модель та уявлення не пов’язані.

ViewModel також містить логіку з отримання даних з моделі, які потім передаються у подання. І також VewModel визначає логіку щодо оновлення даних у моделі. Оскільки елементи уявлення, тобто візуальні компоненти типу кнопок, не використовують події, уявлення взаємодіє з ViewModel за допомогою команд [7].

Підсумком застосування патерну MVVM є функціональний поділ програми на три компоненти, які простіше розробляти та тестувати, а також надалі модифікувати та підтримувати.

2.2.4 Архітектура VIPER

VIPER (View, Interactor, Presenter, Entity і Routing) – це адаптація Clean Architecture для додатків iOS. VIPER характеризується чіткими обов’язками, тонкою деталізацією та чіткими відносинами ізоляції, які можуть принести багато переваг:

- якісне тестування: тестування інтерфейсу користувача та бізнес-логіки може виконуватися окремо;
- легко повторювати: кожна частина несе одну відповідальність, і розробник можете чітко знати, де має бути розміщений новий код;
- ступінь ізоляції високий, а ступінь зчеплення низький: код одного модуля не може легко вплинути на інший модуль;
- легко працювати в команді: кожна частина має чіткий поділ праці, легко уніфікує стиль коду під час спільної роботи та може швидко перейняти код інших людей.

На рисунку 2.6 наведено структуру VIPER. Модуль View надає повний вигляд додатку. Він відповідає за поєднання, макет та оновлення виду. Також в його обов’язки входить надання модулю Presenter інтерфейсу для

подальшого оновлення подання.

Модуль Presenter отримує та обробляє події з модуля View, а також отримує запит на бізнес-логіку від модулю Interactor. Він надає модулю Interactor дані у поданні, отримує та обробляє події зворотного виклику даних від модулю Interactor. Важливою є функція повідомлення модуля View про оновлення [8].

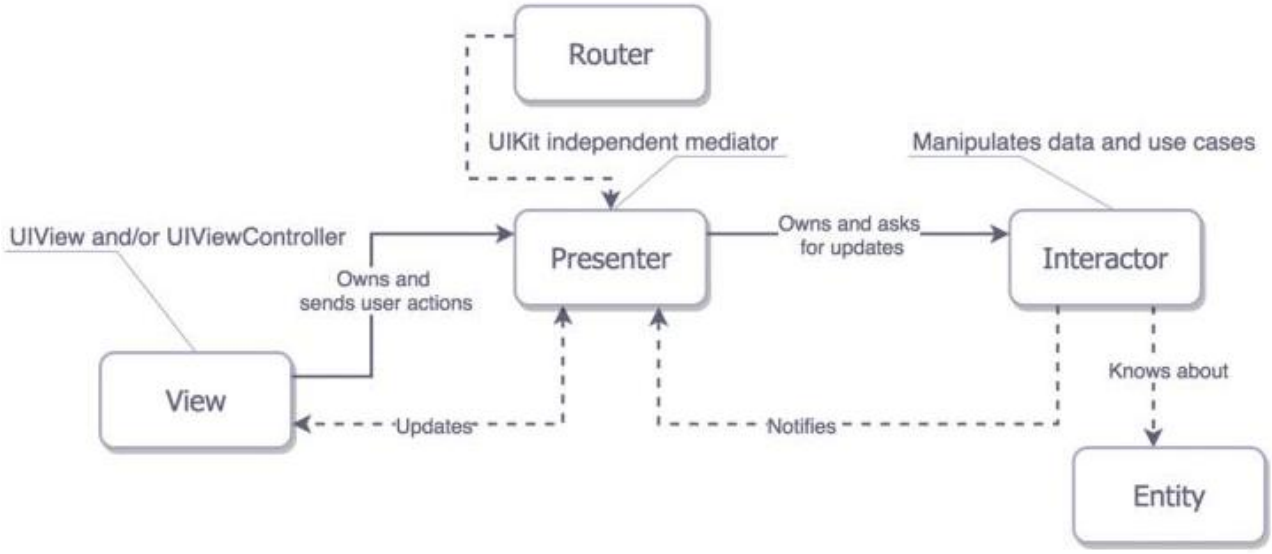

Рисунок 2.6 – Структура VIPER

Модуль Router забезпечує функцію переходу між уявленнями, зменшуючи зв’язок між модулями та ініціалізує різні модулі VIPER.

Модуль Interactor підтримує основні функції бізнес-логіки та надає модулю Presenter існуючі бізнес-сценарії використання. Також Interactor підтримує, отримує та оновлювати сутність, коли відбувається подія, а також повідомляє доповідача

Модуль Entity містить базові об’єкти моделі, які використовуються в модулі Interactor. Наведений поділ також відповідає принципу єдиної відповідальності. Модуль Interactor відповідає за бізнес-логіку, модуль Presenter відповідає за взаємодії людини та додатка, оброблює жести, натискання та інше. В свою чергу модуль View відповідає за візуальний вигляд додатку, що розробляється.

3 ОБГРУНТУВАННЯ ІНСТРУМЕНТАРІЮ ДЛЯ РОЗРОБКИ МОДЕЛІ

3.1 Інтегроване середовище розробки Xcode

Xcode – це IDE (інтегроване середовище розробки), яке створене компанією Apple, для розробки програмного забезпечення для macOS, iOS, watchOS та tvOS. Це єдиний інструмент, який офіційно підтримується, для створення та публікації програм у Apple App Store, що призначено для використання початківцями та досвідченими розробниками.

Xcode включає в себе всі інструменти, що необхідні для створення програм в рамках одного програмного пакета, а саме, текстовий редактор, компілятор та систему збірки. За допомогою Xcode можна писати, компілювати та налагоджувати програму, а після закінчення написання додатку є змога надіслати її в Apple App Store [9]. Xcode містить багато інструментів, які допомагають пришвидшити процес розробки, тому досвідчені розробники можуть створювати програми швидше, а новачки стикаються з меншою плутаниною та клопотами, щоб створити чудовий додаток.

Як редактор коду, Xcode підтримує велику різноманітність мов програмування – C, C++, Objective-C, Objective-C++, Java, AppleScript, Python, Ruby, ResEdit та Swift; використовує моделі програмування Cocoa, Carbon та Java.

IDE використовується для надання єдиного вікна для роботи, яке буде містити в собі все необхідне для розробника. IDE має перевірку вихідного коду та функцію автозаповнення, що значно полегшує написання вихідного коду. Також при створенні нового проекту, можно обрати один із доступних шаблонів, щоб отримати інфраструктуру для розширення. Ці функції корисні для початківців, оскільки вони дають опору для продовження навчання. Досвідчені розробники знайдуть ці функції корисними для оптимізації робочого процесу та пришвидшення процесу розробки додатків.

3.1.1 Редактор Interface Builder

Редактор Interface Builder (рисунок 3.1) у Xcode спрощує розробку повного інтерфейсу користувача без написання будь-якого коду. Інтерфейс додатку можливо побудувати лише перетягаючи кнопки, текстові поля та інші об’єкти на полотно дизайну, щоб створити функціонуючий інтерфейс користувача.

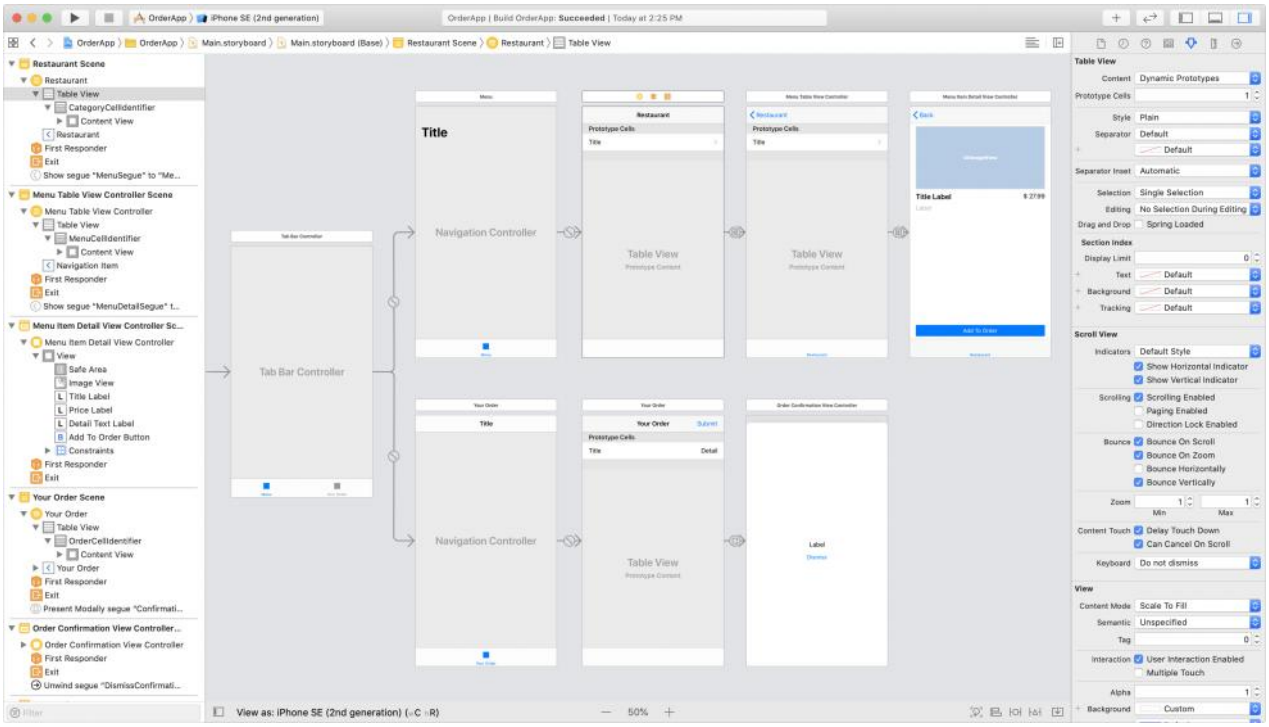

Рисунок 3.1 – Interface Builder

Оскільки Cocoa та Cocoa Touch створені за допомогою шаблону Model-View-Controller, легко самостійно проектувати інтерфейси, окремо від їх реалізацій. Інтерфейси користувача насправді є архівованими об’єктами Cocoa або Cocoa Touch (збережені як файли .nib), а macOS та iOS динамічно створюватимуть зв’язок між інтерфейсом користувача та кодом під час запуску програми.

Повна програма iOS складається з кількох представлень, через які користувач переміщується [10]. Зв’язки між цими представленнями визначаються розкадровками, які показують повне уявлення про потік

додатка.

Конструктор розкадрування Interface Builder дозволяє легко створювати та проектувати нові уявлення, а також об’єднувати їх разом, щоб створити повний інтерфейс користувача, готовий до використання коду користувача.

Xcode включає в себе:

- контролер перегляду таблиці;
- контролер перегляду колекції;
- контролер навігації;
- контролер панелі вкладок;
- контролер перегляду сторінки;
- контролер перегляду GLKit.

Асистент допомагає під час редагування файлу Storyboard або .xib, щоб швидко підключити елементи керування інтерфейсу користувача до коду, який реалізує їх поведінку [11]. Якщо ще немає написаного коду, Xcode запропонує створити заглушку для дії (метод запуску) або вихід (змінна для зберігання даних), яка забезпечить логіку інтерфейсу.

І iOS, і macOS містять потужну систему компонування під назвою Auto Layout з чудовою підтримкою, яка вбудована в Interface Builder. Автоматичний макет заснований на ідеї, що кожен об’єкт у інтерфейсі може визначити обмеження, щоб контролювати, як він реагує на батьківський перегляд та інші елементи керування інтерфейсом. Наприклад, можна визначити, чи буде кнопка залишатися певного розміру чи розгортатися, щоб вмістити більший текст під час відображення іншою мовою.

Interface Builder може автоматично створювати всі обмеження, забезпечуючи набір сумісних правил. Також можна безпосередньо керувати обмеженнями, щоб визначити точний пріоритет кожного, визначаючи, як програма працюватиме на різних розмірах екрана, при повороті або запуску в нових мовах. Режим попереднього перегляду використовується для швидкого перегляду інтерфейсу у різних ситуаціях, не запускаючи програму, що значно

прискорює ітеративний процес проектування [12]. Можна переглядати додаток у книжковому або альбомному форматі, у попередній версії iOS, на екранах різних розмірів тощо.

3.1.2 Фреймфорк для тестування XCTest

XCTest полегшує написання тестів на різних рівнях абстракції. Хороша стратегія тестування поєднує кілька типів тестів, щоб максимізувати переваги кожного. Зазвичай використовується «пірамідальний» розподіл тестів (рисунок 3.2). XCTest додає велику кількість швидких, добре ізольованих модульних тестів, щоб охопити логіку додатка, меншу кількість тестів інтеграції, щоб продемонструвати, що інтегровані частини правильно з’єднані між собою, і тести інтерфейсу користувача, щоб підтвердити правильну поведінку типових випадків використання.

Тести інтерфейсу користувача є остаточним індикатором того, що додаток працює для користувачів так, як очікується, але вони виконуються довше, ніж інші види тестів [13]. Піраміда тестів поєднує тести високої точності, які демонструють, що користувачі можуть виконувати свої завдання, із чітко сфокусованими тестами, які дають швидкий відгук про правильність логіки додатка та вплив внесених змін.

На додаток до піраміди тестів, пишуть тести продуктивності, щоб забезпечити охоплення регресією критично важливих для продуктивності областей коду, щоб дізнатися про процес визначення коду, який має критичне значення для продуктивності.

Кожен модульний тест має підтверджувати очікувану поведінку одного шляху через метод або функцію у проекті. Щоб охопити кілька шляхів, необхідно написати по одному тесту для кожного зі сценаріїв. Наприклад, якщо функція отримує необов’язковий параметр, необхідно написати тест, і у якому параметр дорівнює нулю, і тест, у якому він приймає ненульове значення.

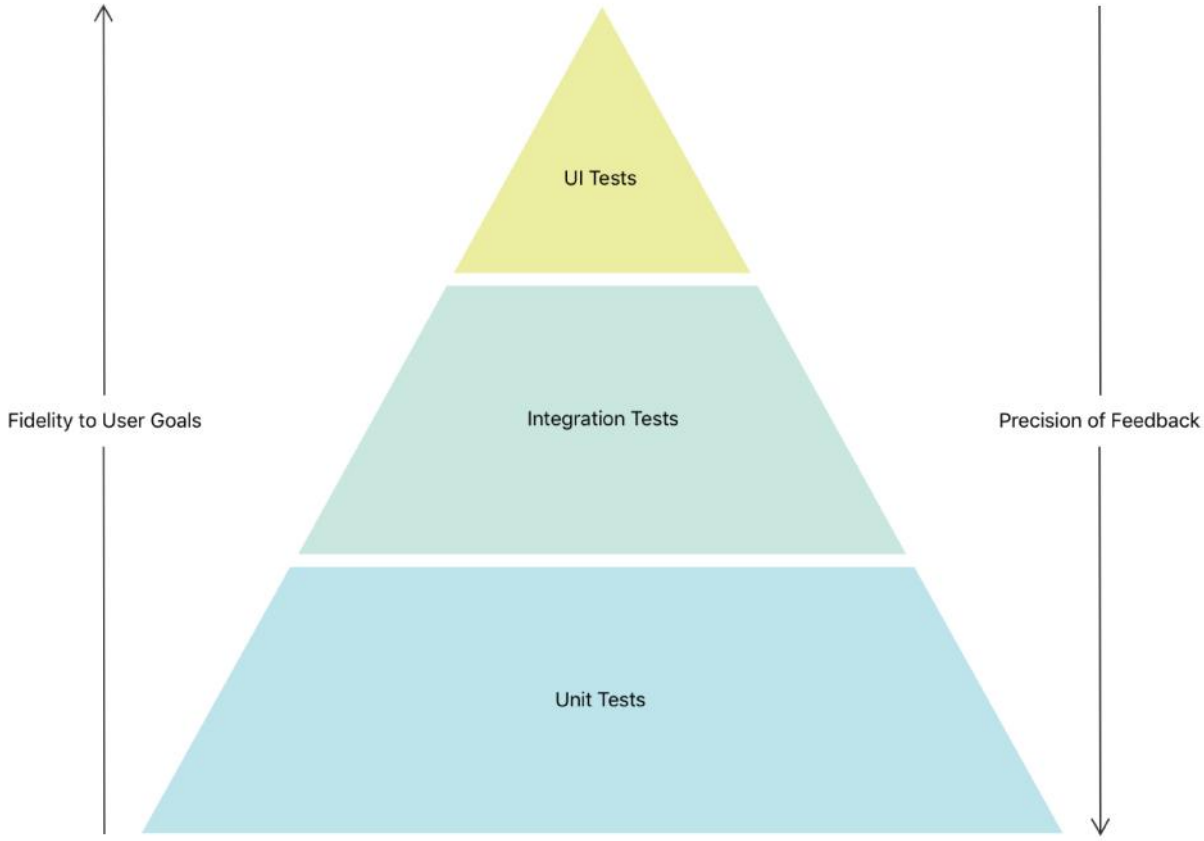

Рисунок 3.2 – Пірамідальний розподіл тестів

Обравши клас або функцію для тестування створюється підклас XCTestCase, який містить тести для цього класу чи функції. Додається метод до підкласу XCTestCase, який не приймає аргументів та повертає Void, даючи методу назву, що починається зі слова «test».

Метод тестування містити три кроки в порядку, описаному нижче:

- зміна залежностей на «заглушки», які легко налаштувати, щоб гарантувати, що тести виконуються швидко;
- виклик методу або функції, що тестується, використовуючи параметри та властивості, які налаштовуються на попередньому етапі;
- застосування тестових тверджень у фреймворку XCTest, для порівняння поведінки коду, який використовуєте на етапі дії, з очікуваннями розробника щодо того, що має статися. Будь-яке твердження, умова якого хибна, призводить до невдачі тесту. Отриманий метод тестування зображений на рисунку 3.3.

```
class MyAPITests : XCTestCase {
  func testMyAPIWorks() {
    // Arrange: create the necessary dependencies.
    // Act: call my API, using the dependencies created above.
    XCTAssertTrue(/* … */, "The result wasn't what I expected")
  }
}
```

Рисунок 3.3 – Метод тестування

3.2 Мова програмування Swift

Swift – це безпечна, швидка та інтерактивна мова програмування, яка поєднує в собі найкраще в сучасному мовному мисленні з мудрістю широкої інженерної культури Apple та різноманітними внесками її спільноти з відкритим кодом. Компілятор оптимізовано для продуктивності, а мова оптимізована для безпечної розробки.

Swift дружня до новачків: це мова програмування промислової якості, така ж виразна та приємна, як і мова сценаріїв. Playground – це навчальний інструмент для роботи зі Swift, який дозволяє експериментувати з кодом та відразу бачити результати, не витрачаючи час на створення та запуск програми [14].

Swift визначає великі класи поширених помилок програмування, приймаючи сучасні шаблони програмування:

- змінні завжди ініціалізуються перед використанням;
- індекси масиву перевіряються на наявність помилок за межами;
- цілі числа перевіряються на переповнення;
- додаткові параметри гарантують, що нульові значення обробляються явно;
- управління пам’яттю здійснюється автоматично;
- обробка помилок дозволяє контролювати відновлення після несподіваних збоїв.

Swift-код зібрано й оптимізовано, щоб отримати максимальну віддачу

від сучасного обладнання. Синтаксис та стандартна бібліотека були розроблені на основі принципу керування, згідно з яким очевидний засіб написання коду також має бути найкращим. Його поєднання безпеки та швидкості робить Swift чудовим вибором для всього, починаючи з «Hello, world!» на всю операційну систему. Swift – це узгодження шаблонів із сучасним легким синтаксисом, що дозволяє чітко та стисло висловлювати складні ідеї. У результаті код не тільки легше писати, але також легше читати та підтримувати.

3.3 Менеджер пактів SPM

Менеджер пакетів Swift – це інструмент для керування розповсюдженням коду Swift. Він інтегрований із системою збірки Swift для автоматизації процесу завантаження, компіляції та зв’язування залежностей. Менеджер пакетів входить до Swift 3.0 і вище. SPM організовує код у модулі. Кожен модуль визначає простір імен та забезпечує контроль доступу, до частин коду що можна використовувати поза модулем [15].

Програма може мати весь свій код в одному модулі, або вона може імпортувати інші модулі як залежності. Окрім кількох системних модулів, таких як Darvin у macOS або Glibc у Linux, більшість залежностей вимагають завантаження та побудови коду для використання.

Окремий модуль, який вирішує певну проблему, можна повторно використовувати в інших ситуаціях. Наприклад, модуль, який надає функціональні можливості для створення мережних запитів, може використовуватися спільно між програмою для обміну фотографіями та програмою погоди. Використання модулів дозволяє будувати поверх коду інших розробників, а не самостійно реалізовувати ту ж функціональність.

Пакет складається з Swift-вихідних файлів та файлу маніфесту. Файл маніфесту, який називається Package.swift, визначає назву пакета та його вміст за допомогою PackageDescription модуля.

Пакет має одну або кілька цілей. Кожна ціль визначає продукт та може визначати одну або кілька залежностей. Taget може створювати або бібліотеку, або запускати код з файлу як свій. Бібліотека містить модуль, який може бути імпортований іншим кодом Swift. Код який запущено є програмою, яка може працювати в операційній системі.

Залежність складається з відносної або абсолютної URL-адреси джерела пакунка та набору вимог до версії пакунка, яку можна використовувати. Роль менеджера пакетів полягає в тому, щоб зменшити витрати на координацію шляхом автоматизації процесу завантаження та створення всіх залежностей для проекту. Це рекурсивний процес: залежність може мати власні залежності, кожна з яких також може мати залежності, утворюючи графік залежностей. Менеджер пакетів завантажує та створює все, що потрібно для задоволення всього графіка залежностей.

3.4 Принципи об’єктно-орієнтованого програмування SOLID

SOLID – це абревіатура п’яти основних принципів проектування в об’єктно-орієнтованому програмуванні: Single responsibility, Open-closed, Liskov substitution, Interface segregation та Dependency inversion. У перекладі українською: принципи єдиної відповідальності, відкритості/закритості, підстановки Барбари Лисків, поділу інтерфейсу та інверсії залежностей [16].

Абревіатура SOLID була запропонована Робертом Мартіном, автором кількох книг, широко відомим у спільноті розробників. Ці принципи дозволяють будувати масштабовані та супроводжувані програмні продукти зі зрозумілою бізнес-логікою. Розглянемо докладніше принципи:

- Single responsibility – принцип єдиної відповідальності;
- Open-closed – принцип відкритості/закритості;
- Liskov substitution – принцип підстановки Барбари Лисков;
- Interface segregation – принцип поділу інтерфейсу;
- Dependency inversion – принцип інверсії залежностей.

3.4.1 Принцип єдиної відповідальності

Принцип єдиної відповідальності (The Single Responsibility Principle або SRP) – один із п’яти основних принципів об’єктно-орієнтованого програмування та проектування, які були сформульовані Робертом Мартіном.

Принцип декларує, що кожен об’єкт повинен мати один обов’язок і цей обов’язок повинен бути повністю інкапсульований у клас, а всі його сервіси повинні бути спрямовані виключно на забезпечення цього обов’язку.

Дотримання принципу полягає зазвичай у декомпозиції складних класів, які роблять відразу багато речей, на прості, відповідальність яких дуже спеціалізована. Але також і об’єднанні в окремий клас однотипної функціональності, яка може бути розподіленою за багатьма класами, може розглядатися як дотримання цього принципу[17].

Проектування класів із спрямованістю забезпечення єдиної обов’язки спрощує подальші модифікації та супровід, оскільки простіше розібратися у одному блоці функціональності, ніж розплутувати складні взаємозв’язки між різними функціональними блоками. Також при модифікації логіки в одному місці застосування знижуються ризики виникнення проблем в інших «несподіваних» його місцях.

Слідування SRP дуже корисна практика з погляду повторного використання коду. Складні об’єкти з комплексними залежностями зазвичай дуже складно використовувати повторно, якщо потрібна лише частина реалізованого в них функціоналу. А невеликі класи з чітко окресленим функціоналом, навпаки, простіше використовувати повторно, оскільки вони не надмірні та рідко тягнуть у себе істотний обсяг залежностей.

Найбільш яскравим анти-патерном, що порушує принцип єдиної відповідальності, є використання God-об’єктів, які «надто багато знають» або «надто багато вміють». Виникають такі «божественні об’єкти» зазвичай через любов розробників до абстракції – якщо зводити абстракцію в абсолют, то

цілком можна будь-який об’єкт реального світу відобразити у додатку у вигляді екземпляра якогось універсального класу. На словах це навіть може виглядати логічно, але практично майже завжди це призводить до проблем супроводжуваності. Зазвичай такі об’єкти стають центральною частиною системи, модифікація яких вкрай складна, оскільки стає дуже складно передбачити, як змінення коду для вирішення поточного завдання може позначитися на раніше реалізованій функціональності.

Насправді, як і будь-які інші принципи, SRP вимагає свідомого та осмисленого застосування. Надмірна декомпозиція може виявитися шкідливою, якщо вона призводить до більшої складності або ускладнює супровід.

Наприклад, патерн ActiveRecord, що часто використовується у фреймворках, порушує принцип єдиної відповідальності. ActiveRecord реально поєднує у собі дуже багато функціональних можливостей і часто змішує бізнес-логіку та роботу із шаром зберігання. При цьому використання ActiveRecord часто є зручним та доцільним [18]. На цьому прикладі стає зрозумілим, що SRP – це не догма, а порушення цього принципу цілком може бути логічним та доцільним.

3.4.2 Принцип відкритості/закритості

Принцип відкритості/закритості (The Open Closed Principle або OCP) – принцип, який декларує, що програмні сутності (класи, модулі, функції тощо) мають бути відкриті до розширення, але закриті зміни. Це означає, що ці сутності можуть змінювати свою поведінку без зміни їхнього вихідного коду.

У цьому контексті відкритість для розширення – це можливість додати для класу, модуля або функції нову поведінку, якщо потреба у цьому виникне, а закритість для змін – це заборона зміни вихідного коду програмних сутностей. На перший погляд, це звучить складно та

суперечливо. Але якщо розібратися, то принцип цілком логічний.

Дотримання принципу OCP полягає в тому, що програмне забезпечення змінюється не через зміну існуючого коду, а через додавання нового коду. Тобто створений код залишається «недоторканим» та стабільним, а нова функціональність впроваджується або через успадкування реалізації, або через використання абстрактних інтерфейсів та поліморфізм.

Принцип відкритості/закритості Мейєра ґрунтується на ідеї, що розроблена спочатку реалізація класу надалі не модифікується (хіба що виправляються помилки), а будь-які зміни здійснюються через створення нового класу, який зазвичай успадковується від початкового. Згідно з визначенням Мейєра реалізація інтерфейсу може бути успадкована та перевикористана, але інтерфейс може змінитися в новій реалізації.

Пізніше було сформульовано поліморфний принцип відкритості/закритості. Він ґрунтується на суворій реалізації інтерфейсів та на спадкуванні від абстрактних базових класів або на поліморфізмі. Створений спочатку інтерфейс має бути закритий для модифікацій, а нові реалізації як мінімум відповідають цьому первісному інтерфейсу, але можуть підтримувати інші, більш розширені.

3.4.3 Принцип підстановки Барбари Лисков

Принцип підстановки Барбари Лисков (The Liskov Substitution Principle або LSP) – це принцип, який у формулюванні Роберта Мартіна декларує, що функції, які використовують базовий тип, повинні мати можливість використовувати підтипи базового типу, не знаючи про це. Оригінальне визначення Барбари Лисков більш формальне та помітно складніше для сприйняття: «У тому випадку, якщо q(x) – властивість, вірна по відношенню до об’єктів x якогось типу T, то властивість q(y) теж буде вірною щодо ряду об’єктів y, які ставляться до типу S, у своїй S – підтип деякого типу T».

Дотримання принципу LSP у тому, що з побудови ієрархій

успадкування спадкоємці, що створюються, мають коректно реалізовувати поведінку базового типу. Тобто якщо базовий тип реалізує певну поведінку, то ця поведінка має бути коректно реалізована і для її спадкоємців.

LSP перегукується з контрактним програмуванням, визначаючи точні, формальні та версифіковані описи інтерфейсів. І інтерфейси, що реалізуються спадкоємцями, повинні відповідати контракту інтерфейсів базового класу.

Спадкоємець класу доповнює, але не замінює поведінку базового класу. Тобто будь-де у програмі заміна базового класу на клас-спадкоємець не повинна викликати проблем. Якщо з якихось причин не виходить, то найімовірніше має місце або некоректна реалізація, або неправильно обрана абстракція для наслідування.

Дотримання принципу підстановки Барбари Лиск дозволяє гарантувати, що будь-який створений підклас буде без проблем використовуватися раніше реалізованими модулями, які працювали з надкласом. І це значно спрощує розширення функціональних можливостей системи.

Але LSP, як і будь-який інший принцип, не є догмою. І іноді дотримання цього принципу при побудові архітектури може призводити до ресурсомісткішої реалізації, ніж робота з порушенням цього принципу. Але як і з будь-якими іншими правилами, треба усвідомлювати можливі наслідки порушення.

3.4.4 Принцип поділу інтерфейсу

Принцип поділу інтерфейсу (The Interface Segregation Principle або ISP) – це принцип, який у формулюванні Роберта Мартіна декларує, що клієнти не повинні залежати від методів, які вони не використовують. Тобто, якщо якийсь метод інтерфейсу не використовується клієнтом, то зміни цього методу не повинні призводити до необхідності внесення змін до клієнтського

коду.

Дотримання принципу ISP полягає у створенні інтерфейсів, які досить специфічні та вимагають лише необхідний мінімум реалізацій методів. Надлишкові інтерфейси, навпаки, можуть вимагати від класу, що реалізує, створення великої кількості методів, причому навіть таких, які не мають сенсу в контексті класу.

У чомусь принцип поділу інтерфейсу перегукується з принципом єдиної відповідальності – інтерфейси не повинні бути надмірно «товстими», якщо раптом у додатку формується об’ємний інтерфейс, тобто висока ймовірність, що так відбувається через те, що в цьому інтерфейсі занадто багато різних відповідальності, а значить логічніше провести декомпозицію складного інтерфейсу на кілька простих.

Принцип поділу інтерфейсу знижує складність підтримки та розвитку програми. Чим простіше та мінімалістичніше інтерфейс, що використовується, тим менш ресурсомісткою є його реалізація в нових класах, тим менше причин його модифікувати, але і у разі модифікації вона буде значно простішою.

3.4.5 Принцип інверсії залежностей

Принцип інверсії залежностей (The Dependency Inversion Principle або DIP) – це принцип, який декларує, що модулі верхніх рівнів не повинні залежати від нижніх модулів рівнів, а обидва типи модулів повинні залежати від абстракцій; самі абстракції не повинні залежати від деталей, а деталі повинні залежати від абстракцій.

Дотримання принципу інверсії залежностей «примушує» реалізовувати високорівневі компоненти без вбудовування залежностей від конкретних низькорівневих класів, що, наприклад, сильно спрощує заміну залежностей, що використовуються для цілей тестування.

Наприклад, реалізуємо зберігання документів у веб-програмі. На

перший погляд, здається логічним додати залежність від модулів роботи з файловою системою безпосередньо до класу, який відповідає за високорівневу роботу з цими документами. Але в перспективі така залежність може створити проблеми – наприклад, користувачу потрібно зберігати дані не лише на диску, а й у хмарі. Якщо ж в залежність виведено рівень абстракції (інтерфейсу), розробнику буде достатньо реалізувати функціонал роботи з хмарою, відповідний раніше створеному інтерфейсу роботи з файлами [19].

Принцип інверсії залежностей часто спрощує дотримання принципу підстановки Барбари Лисков. Виділення абстракцій та вбудовування залежностей від них знижує ймовірність того, що у новому класі розробник не повністю реалізує контракт базового класу, який він розширює у рамках нового.

4 МОДЕЛЬ ЗАСТОСУНКУ, ЩО РОЗРОБЛЯЄТЬСЯ

4.1 Огляд моделі застосунку

Модель застосунку описує шаблони та методи, які використовуються для проектування та створення програми. Модель дає карту розвитку та методи, яких слід дотримуватися під час створення програми, щоб у підсумку вийшов добре структурований додаток.

Після аналізу предметної області і вже існуючих моделей застосунку можна спроектувати модель, яка буде максимально надійною та легкою в побудові. Такою моделлю можна назвати комбінацію архітектури Model-View-ViewModel, а також шаблону Coordinator (рисунок 4.1). Ще необхідно зазначити додання базових сервісів та розширень і застосування особливостей розробки додатків IOS, такі як змога використання кастомного шаблону модулю чи додатку та ін.

Модель додатку може бути побудована на основі множини M, яка містить в собі інші елементи та множини (4.1):

$$M = \{ AD, AC, F, Sc, Ser, CE \} \quad (4.1)$$

де AD (AppDelegate) – об’єкт делегата додатка, який керує загальною поведінкою додатку;

AC(AppCoordinator) – об’єкт, який відповідає за навігацію додатку, створює та налаштовує необхідний потік;

F (FlowCoordinator) – об’єкт, який обробляє навігацію в потоці та налаштовує сцени;

Sc (Scene) – модуль, який відповідає одному екрану;

Ser (Service) – об’єкти, через які виконується взаємодія сцени з певним інтерфейсом;

CE (Common Extension) – розширення базових класів UIKit та Foundation з метою збільшення їх функціоналу та для заощадження часу

написання програми та зменшення написання кількості коду. Також це можуть бути розширення класів користувача з метою розбиття коду на логічні частини для поліпшення читання або для розділення на декілька файлів з метою запобігання масивних файлів.

Вхідною точкою в додаток є AD(AppDelegate). Делегат програми фактично є кореневим об’єктом програми, і він працює разом із UIApplication для керуванням та деякими взаємодіями з системою. Як і UIApplication, об’єкт UIKit створює об’єкт делегата програми на початку циклу запуску програми, тому він завжди присутній.

Об’єкт делегата програми використовуються для виконання таких завдань:

- ініціалізація центральних структур даних додатка;

- налаштування сцен програми;

- відповідь на сповіщення, що надходять із-за меж додатка, як-от попередження про низький рівень пам’яті, сповіщення про завершення завантаження тощо;

- відповідь на події, які націлені на саму програму та не стосуються сцен, переглядів або контролерів перегляду додатка;

- реєстрація для будь-яких необхідних служб під час запуску, наприклад, служби Apple Push Notification.

В свою чергу AD створює AC (AppCoordinator) передаючи туди необхідні дані та залежності. AC відповідає за навігацію додатку, створює та налаштовує необхідний потік. Одразу після свого створення AC, на основі отриманих даних обирає, який потік необхідно запустити та запускає його, також передаючі в нього необхідні дані та залежності.

FC обробляє навігацію в потоці та налаштовує сцени.

Sc – це модуль, який відповідає одному екрану та описується множиною (4.2):

$$Sc = \{ M, V, VM \} \quad (4.2)$$

де: M – дані, з якими працює програма;

V – візуальні елементи інтерфейсу користувача. У iOS контролер представлення невіддільний від концепції представлення.

VM – оновлює дані, з якими працює сцена та інтерфейс користувача відповідно до них.

Ser – це об’єкти, через які виконується взаємодія сцени з певним інтерфейсом будь то мережа, база даних чи взагалі робота з апаратною частиною девайсу такі як камера, TuchID та інше. Множина Ser описуються кортежем (4.3):

$$Ser = \{NS, KS\}, \quad (4.3)$$

де: NS (NetworkServise) – сервіс, який відповідає за отримання даних з API;

KS (KingfisherService) – сервіс, який відповідає за завантаження зображення по URL-адресі.

4.2 Навігація у застосунку (модуль навігації)

За навігацію застосунку відповідає координатор. Координатори є чудовим інструментом, оскільки вони звільняють ViewController від відповідальності, якої він не повинні нести. Це допомагає дотримуватися принципу єдиної відповідальності, що робить ViewController набагато компактнішим та легшим для повторного використання.

ViewController не повинен знати, який ViewController був перед ним, який з них повинен бути наступним або, які залежності він повинен передати.

Це має бути просто обгортка навколо View/Subviews на екрані та обробляти лише речі, пов’язані з UIKit. Будь-які дії, що стосуються навігації на ViewController, повинні надсилатися координатору для належної обробки.

Координатор отримує події що сталися від модулю ViewModel через CoordinatorDelegate.

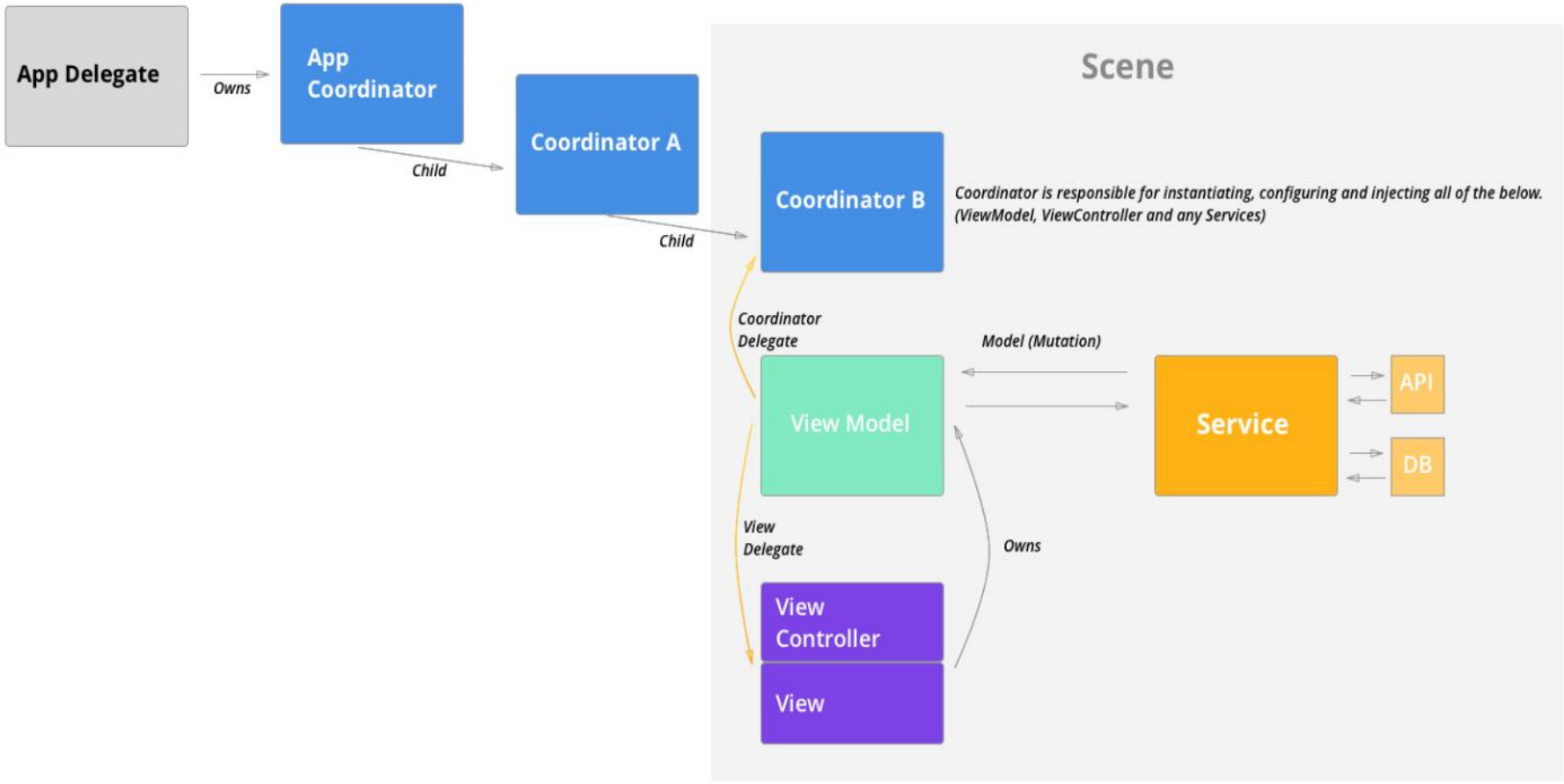

Рисунок 4.1 – Схема архітектури MVVM+C

Абстрактний базовий клас, від якого успадковуються всі координатори наведено у лістенгу 4.1.

Лістинг 4.1 – Абстрактний клас координатора

```
class Coordinator {
    private(set) var childCoordinators: [Coordinator] = []
    func start() {
        preconditionFailure("This method needs to be overriden
by concrete subclass.")
    }
    func finish() {
        preconditionFailure("This method needs to be overriden
by concrete subclass.")
    }
    func addChildCoordinator(_ coordinator: Coordinator) {
        childCoordinators.append(coordinator)
    }
    func removeChildCoordinator(_ coordinator: Coordinator) {
        if let index = childCoordinators.index(of: coordinator)
{
            childCoordinators.remove(at: index)
        } else {
            Print("Couldn't remove coordinator: \(coordinator).
It's not a child coordinator.")
        }
    }
    func removeAllChildCoordinatorsWith<T>(type: T.Type) {
        childCoordinators = childCoordinators.filter { $0 is T
== false }
    }
    func removeAllChildCoordinators() {
        childCoordinators.removeAll() }
}
extension Coordinator: Equatable {
    static func == (lhs: Coordinator, rhs: Coordinator) -> Bool
{
        return lhs === rhs     }
}
```

Додаток має складатися з кількох координаторів, по одному для кожної сцени. Але він завжди повинен мати один «головний» AppCoordinator, який буде належати делегату програми.

Наступний код (лістинг 4.2) є прикладом базового AppCoordinator. Усі інші координатори в додатку будуть дочірніми від цього координатора.

Лістинг 4.2 – Клас AppCoordinator

```
class AppCoordinator: Coordinator {
     // MARK: - Properties
    let window: UIWindow?
    lazy var rootViewController: UINavigationController = {
        return UINavigationController(rootViewController:
UIViewController())
    }()
    let apiClient: ApiClient = {
        let configuration = URLSessionConfiguration.default
        configuration.httpAdditionalHeaders = ["Content-Type":
"application/json; charset=utf-8"]
        let apiClient = ApiClient(configuration: configuration)
        return apiClient
    }()
    // MARK: - Coordinator
    init(window: UIWindow?) {
        self.window = window
    }
    override func start() {
        guard let window = window else {
            return
        }
        window.rootViewController = rootViewController
        window.makeKeyAndVisible()    }
    override func finish() {
    }
}
```

AppCoordinator має три важливі частини:

- Properties: це властивості, якими володіє цей об’єкт. У цьому випадку, оскільки це AppCoordinator, він повинен мати властивість UIWindow. Він також володіє кореневим та іншими ViewController, а також будь-якими залежностями, які йому знадобляться, як у цьому випадку ApiClient;

- Init method: тут повинні бути передані будь-які залежності, які знадобляться у цьому координаторі. У лістингу 4.2, оскільки це перший координатор, єдина залежність, яка потрібна, – це UIWindow;

- Start/Finish method – повинні перевизначити методи Start() та Finish(). Метод запуску є найважливішим, це те, де розробник представляє свій інтерфейс користувача на екрані. У цьому випадку розробник встановляє кореневий контролер подання на вікно.

У лістингу 4.3 наведено запуск додатку з AppDelegate за допомогою AppCoordinator.

Лістинг 4.3 – Запуск додатку за допомогою AppCoordinator

```
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    var appCoordinator: AppCoordinator!

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        window = UIWindow(frame: UIScreen.main.bounds)
        appCoordinator = AppCoordinator(window: window)
        appCoordinator.start()
        return true
    }
}
```

Також додатку знадобиться принаймні один FlowCoordinator для обробки всіх сцен у додатку.

Приклад такого типу координатора наведено у лістингу 4.4. Він буде схожий на AppCoordinator, але він матиме більше обов’язків, залежно від того, наскільки складною є сцена.

Лістинг 4.4 – Реалізація класу FlowCoordinator

```
class SearchInputCoordinator: Coordinator {
    // MARK: - Properties

    let rootViewController: UITabBarController
    let rootNavigationController: UINavigationController = {
        let navVC = UINavigationController()
        navVC.tabBarItem = .search
        return navVC     }()

    let storyboard = UIStoryboard(named: "Search")
    let apiClient: ApiClient

    // MARK: VM / VC's
    lazy var searchInputViewModel: SearchInputViewModel! = {
        let viewModel = SearchInputViewModel()
        viewModel.coordinatorDelegate = self
        return viewModel    }()

    var locationSearchViewModel: LocationSearchViewModel {
        let placeService = PlaceApiService(apiClient: apiClient,
plistClient: PlistClient())
        let viewModel = LocationSearchViewModel(service:
placeService)
```

```
        viewModel.coordinatorDelegate = self
        return viewModel    }
    // MARK: - Coordinator
    init(rootViewController: UITabBarController, apiClient:
ApiClient) {
        self.rootViewController = rootViewController
        self.apiClient = apiClient    }
    override func start() {
        let searchInputVC: SearchInputViewController =
storyboard.instantiateViewController()
        searchInputVC.viewModel = searchInputViewModel
        rootNavigationController.setViewControllers([searchInput
VC], animated: false)
        rootViewController.setViewControllers([rootNavigationCon
troller], animated: false)    }
    override func finish() {
        // Clean up any view controllers. Pop them of the
navigation stack for example.
        delegate?.didFinish(from: self)    }
}
```

FlowCoordinator дуже схожий на AppCoordinator, але має деякі відмінності:

- Properties: посилання на розкадровку, яка пов’язана з цим координатором/сценою, а також з усіма іншими контролерами представлення та залежностями;

- Init, Root View Controller: цей координатор ініціалізується кореневим ViewController, а не вікном, як у AppCoordinator;

- Init, Api Client: ApiClient також передається в ініціали заторі;

- Start: тип кореневого «контейнера», який розробник ініціалізує, визначає, що він збираємося робити в методі Start();

- Finish: тут йде очистка всіх ViewControllers, які є в стеку навігації.

Навігація є головною метою роботи координаторів. Як правило додаються розширення до координатора для обробки всієї навігації, маючи метод для кожної навігації, яка може статися (лістинг 4.5).

Лістинг 4.5 – Обробка навігації в розширенні класу FlowCoordinator

```
extension SearchInputCoordinator {
    func goToLocationSearch(searchType: LocationSearchType, from
controller: UIViewController) {
```

```
        let viewController: LocationSearchViewController =
storyboard.instantiateViewController()
        viewController.viewModel = locationSearchViewModel
        controller.present(viewController, animated: true,
completion: nil)
    }

    func goToSearch(withState state: SearchInputState, from
controller: UIViewController) {
        let searchCoordinator =
SearchCoordinator(rootViewController: rootNavigationController,
apiClient: apiClient, searchInput: validatedState)
        searchCoordinator.delegate = self
        addChildCoordinator(searchCoordinator)
        searchCoordinator.start()    }
}
```

Іншою важливою частиною координатора є зворотні виклики делегатів, які надходять від ViewModel та інших дочірніх координаторів. По-перше, є можливість отримання зворотних викликів від координатора потоку. Якщо координатор потоку повинен повідомити про щось, у нього буде протокол делегату, який необхідно йому передати при його створенні (лістинг 4.6).

Лістинг 4.6 – Приклад делегата координатора потоку

```
extension SearchInputCoordinator: SearchCoordinatorDelegate {
    func didFinish(from coordinator: SearchCoordinator) {
        removeChildCoordinator(coordinator)
    }
}
```

Інший тип зворотних викликів, які можуть надходити, надходять від ViewModel (лістинг 4.7).

Лістинг 4.7 – Приклад делегата моделі перегляду

```
extension SearchInputCoordinator:
SearchInputViewModelCoordinatorDelegate {
    func didSelectOrigin(from controller: UIViewController) {
        goToLocationSearch(searchType: .origin, from:
controller)
    }

    func didSelectDestination(from controller: UIViewController,
fromPlace place: Place) {
        goToLocationSearch(searchType: .destination(from:
place), from: controller)   }
}
extension SearchInputCoordinator:
```

```
LocationSearchViewModelCoordinatorDelegate {
    func didSelect(place: TripPlace, from controller:
UIViewController) {
        if place.type == .origin {
            searchInputViewModel.state.origin = place
        } else if place.type == .destination {
            searchInputViewModel.state.destination = place     }
        controller.dismiss(animated: true, completion: nil)    }
    func didSelectClose(from viewModel: LocationSearchViewModel,
from controller: UIViewController) {
        controller.dismiss(animated: true, completion: nil)    }
}
```

Ці зворотні виклики дають знати, чи хоче користувач перейти до іншої частини програми, і звідси викликаються методи навігації goTo (лістинг 4.5). ViewController, до якого надходить даний запит, є параметром у цих методах делегата. Це значно полегшує показ контролерів перегляду поверх них, не зберігаючи посилання, або шукати контролер перегляду в ієрархії.

4.3 Модуль сцени

Сцена – це модуль який відповідає одному екрану, має такі основні компоненти як ViewController/View, ViewModel.

ViewController/View – це всі специфічні методи UIKit/UIViewController і все, що безпосередньо стосується переглядів та підпоказань.

ViewModel – це вся бізнес-логіка, наприклад, мережні запити, перевірка, підготовка даних моделі для презентації тощо.

Цей поділ допомагає контролеру представлення дотримуватися принципу єдиної відповідальності. Тепер контролер перегляду, який є частиною UIKit, оскільки це підклас UIViewController, має справу лише з матеріалами UIKit.

Це залишає ViewModel повністю очищеним від UIKit та дозволить зосередитися на основній бізнес-логіці, дотримуючись SRP. Також це значно полегшує роботу з кодом та покриття тестами.

ViewController матиме відносини один до одного з ViewModel. Буде

одна ViewModel на один ViewController, і ViewController буде мати посилання на свою ViewModel. ViewModel передається у ViewController за допомогою додавання залежностей. У моделі що розробляється, ViewModel буде створена координатором потоку та передана до ViewController.

Першим кроком для створення ViewModel є визначення протоколу (лістинг 4.8). ViewModel має базуватися на протоколі, тому що коли ViewModel вводиться залежністю у контролер перегляду, не повинно бути залежності від конкретного типу. Це дозволить поекспериментувати з різними реалізаціями для тестування тих чи інших цілей.

Лістинг 4.8 – Протоколи ViewModel

```
protocol LocationSearchViewModelType {
    weak var viewDelegate: LocationSearchViewModelViewDelegate?
{ get set }
    // Data Source
    var shouldShowHeader: Bool { get }
    var headerText: String { get }
    func numberOfItems() -> Int
    func itemFor(row: Int) -> PlaceViewDataType
    // Events
    func start()
    func searchFor(text: String)
    func didSelectRow(_ row: Int, from controller:
UIViewController)
    func didSelectClose(from controller: UIViewController)}
protocol LocationSearchViewModelCoordinatorDelegate: class {
    func didSelect(place: TripPlace, from controller:
UIViewController)
    func didSelectClose(from viewModel: LocationSearchViewModel,
from controller: UIViewController)}
protocol LocationSearchViewModelViewDelegate: class {
    func updateScreen()
    func updateState(_ state: ViewControllerState)  }
```

Зазвичай має бути три протоколи:

- YourNameViewModelType – це протокол, який має реалізувати модель ViewModel. Він складається з двох основних частин. Джерело даних, яке матиме всі відформатовані дані, необхідно контролеру перегляду, і події, які ViewController надсилатиме під час будь-якої дії користувача;

- YourNameViewModelCoordinatorDelegate – цей протокол делегату дозволить висвітлювати будь-які дії, які повинен виконувати координатор потоку. Цей делегат повинен бути встановлений координатором, коли він створює кожну відповідну ViewModel;

- YourNameViewModelViewDelegate – цей протокол делегата дозволить спілкуватися з ViewController. Наприклад, коли є нові дані, надсилається повпідмовлення до ViewController, щоб він міг оновити свій екран.

У лістингу 4.9 є три важливі частини:

- кілька властивостей, які включають і наших делегатів, і моделі даних, і сервіси;

- метод ініціалізації, в якому повинні отримати всі необхідні залежності. У лістингу 4.9 це сервіс Place, який дозволить виконувати мережні запити;

- приклад мережних запитів, та оновлення ViewController через viewDelegate після їх виконання.

Лістинг 4.9 – Реалізація ViewModel

```
class LocationSearchViewModel {
    // MARK: - Delegates
    weak var coordinatorDelegate:
LocationSearchViewModelCoordinatorDelegate?
    weak var viewDelegate: LocationSearchViewModelViewDelegate?
    // MARK: - Properties
    fileprivate let service: PlaceService
    fileprivate var places: [Place] = []
    fileprivate var cachedPlaces: [Place] = []
    fileprivate var isSearching = false
    // MARK: - Init
    init(service: PlaceService)   {
        self.service = service    }
    func start() {
        getPlacesFromCache()      }
    // MARK: - Network
    func getPlacesFromCache() {
        // Use SERVICE class to get places from cache
        // ...
        self.cachedPlaces = places
        viewDelegate?.updateScreen()    }
```

```
    func getPlaces(text: String) {
        // Use SERVICE class to make request
        // ...
        self.places = places
        viewDelegate?.updateScreen()    }
}
```

Крім усієї бізнес-логіки, як-от мережних запитів у лістингу 4.10 реалізовано протокол ViewModel.

Лістинг 4.10 – Реалізація протоколу ViewModel

```
extension LocationSearchViewModel: LocationSearchViewModelType {
    // MARK: - Data Source
    var shouldShowHeader: Bool {
        return !isSearching    }
    var headerText: String {
        return "Search"    }
    func numberOfItems() -> Int {
        return isSearching ? places.count : cachedPlaces.count
    }
    func itemFor(row: Int) -> PlaceViewDataType {
        let place = isSearching ? places[row] :
prefetchedPlaces[row]
        return PlaceViewData(place: place)      }
    // MARK: - Events
    func searchFor(text: String) {
        guard !text.characters.isEmpty else {
            isSearching = false
            return                          }
        places = []
        viewDelegate?.updateState(.loading(animated: false))
        isSearching = true
        getPlaces(text: text)               }
    func didSelectRow(_ row: Int, from controller:
UIViewController) {
        let place = isSearching ? places[row] :
cachedPlaces[row]
        coordinatorDelegate?.didSelect(place, from: controller)
    }
    func didSelectClose(from controller: UIViewController) {
        coordinatorDelegate?.didSelectClose(from: self, from:
controller)    }    }
```

Методи джерела даних оброблятимуть форматування будь-яких даних моделі, які може запитати ViewController. ViewModel не повинна надсилати будь-який об’єкт моделі безпосередньо на шар перегляду, тому зазвичай

відправляються лише відформатовані данні або інші структури, такі як ViewData.

Методи подій будуть викликані ViewController, коли є дія користувача, і потрібно щось з цим зробити. Іноді потрібно буде розпочати мережний запит, а іноді, якщо це питання навігації, потрібно буде передати його координатору через координатор Delegate. Також слід додати реалізацію ViewController в зв’язці з ViewModel (лістинг 4.11).

Лістинг 4.11 – Реалізація ViewController

```
class LocationSearchViewController: UIViewController {
    // MARK: - Properties
    var viewModel: LocationSearchViewModelType! {
        didSet {
            viewModel.viewDelegate = self        }
    }
    // MARK: - Outlets
    @IBOutlet weak var searchField: UITextField!
    @IBOutlet weak var tableView: UITableView!
    // MARK: - UIViewController
    override func viewDidLoad() {
        super.viewDidLoad()
        viewModel.start()
        setup()                }
    // MARK: - Setup
    func setup() {
        searchTypeLabel.text = viewModel.searchTypeText
        searchField.attributedPlaceholder =
viewModel.searchPlaceholder    }
    // MARK: - Action's
    func searchFor(text: String) {
        viewModel.searchFor(text: text)    }
    @IBAction func didSelectClose(_ sender: Any) {
        viewModel.didSelectClose(from: self)     }
}
// MARK: - ViewModel Delegate
extension LocationSearchViewController:
LocationSearchViewModelViewDelegate {
    func updateScreen() {
        tableView.reloadData()    }
    func updateState(_ state: ViewControllerState) {
        self.state = state                          }
}
```

Є сильне посилання на ViewModel і є прослуховувач didSet, який

готовий встановити себе як viewDelegate, як тільки координатор установить його. Використовується посилання на viewModel, щоб отримати всі необхідні форматовані дані та надіслати їх при будь-якій відповідній події. І, нарешті, реалізація протоколу ViewDelegate, щоб ViewModel могла повідомляти, коли потрібно оновити екран.

4.4 Сервісні модулі

Сервісний рівень – це шаблон дизайну, який існує протягом десятиліть. Він заснований на концепції, що система повинна бути розділена на різні шари, з різними обов’язками на кожному з цих шарів. Рівень сервісу має бути з’єднанням із зовнішнім світом, будь то база даних, віддалений сервер, з’єднання Bluetooth тощо. На цьому рівні послуг має бути декілька служб, і кожна з них повинна мати єдину відповідальність за роботу з певним типом послуги або ресурсу.

По-перше, необхідно визначити базову функціональність сервісу, від якої в майбутньому будуть унаслідуватись конкретні сервіси (лістинг 4.12).

Лістинг 4.12 – Базовий клас Service та допоміжні компоненти для створення сервісів

```
let ServiceRegistry = ServiceRegistryImplementation()
protocol Service {
    var serviceName : String { get }
    func register()                }
extension Service {
    func register() {
        ServiceRegistry.add(service: self)    }}
final class LazyService : Service {
    let serviceName : String
    lazy var serviceGetter : (() -> Service) = {
        if self.service == nil {
            self.service = self.implementationGetter()        }
        return self.service!    }
    private var implementationGetter : (() -> Service)
    private var service : Service? = nil
    init(serviceName : String, serviceGetter : @escaping (() ->
```

```
Service)) {
        self.serviceName = serviceName
        self.implementationGetter = serviceGetter   }}

struct ServiceRegistryImplementation {
    private static var serviceDictionary : [String :
LazyService] = [:]

    func add(service: LazyService) {
        if
ServiceRegistryImplementation.serviceDictionary[service.serviceN
ame] != nil {
            print("WARNING: service \(service.serviceName) is
already registered.")        }

ServiceRegistryImplementation.serviceDictionary[service.serviceN
ame] = service    }

    func add(service: Service) {
        add(service: LazyService(serviceName:
service.serviceName, serviceGetter: { service }))    }

    func serviceWith(name: String) -> Service {
        guard let resolvedService =
ServiceRegistryImplementation().get(serviceWithName: name) else
{ fatalError("Error: SOAService \(name) is not registered via
ServiceRegistry.")       }
        return resolvedService    }

    private func get(serviceWithName name: String) -> Service? {
        return
ServiceRegistryImplementation.serviceDictionary[name]?.serviceGe
tter()    }}
```

Кожен сервіс повинен мати унікальну реалізацію методів serviceName() та register(), які відповідають за створення сервісу та збереження його імені. Клас ServiceRegistryImplementation, відповідає за реєстрацію сервісу в AppDelegate. Після завершення базової реалізації можна визначити конкретні сервіси додатку. NetworkService – це сервіс, який відповідає за отримання данних з API (лістинг 4.13).

Лістинг 4.13 – Реалізація класу NetworkService

```
import ObjectMapper
private let networkServiceName = "NetworkService"
extension NetworkRegistryImplementation {
    var quoteService: QuoteService {
        get {
            return serviceWith(name: networkServiceName) as!
NetworkService        }    }}

protocol NetworkService: Service {
```

```
    func getQuotes(count: Int, success: @escaping (Int, [Quote])
-> (), failure: @escaping (Int) -> ())  }
extension NetworkService {
    var serviceName: String {
        get {
            return networkServiceName        }    }
    func getQuotes(count: Int, success: @escaping (Int, [Quote])
-> (), failure: @escaping (Int) -> ()) {
        let urlString =
SimpsonsRequestConfigurator.configureURLString(Endpoints.QUOTES,
"count", "\(count)")
        let client = NetworkClient(baseUrl: BaseURLs.simpsons)
        client.getArray(urlString: urlString, success: { (code,
arrayOfQuotes) in
            success(code, arrayOfQuotes)
        }) { (code) in
            failure(code)        }    } }
internal class NetworkServiceImplementation: NetworkService {
    internal static func register() {
        ServiceRegistry.add(service: LazyService(serviceName:
networkServiceName, serviceGetter: {
            QuoteServiceImplementation()  }))
    }     }
```

KingfisherService (лістинг 4.14) – це сервіс, який відповідає за отримання даних зображення з URL-адреси. Він реалізується аналогічно до NetworkService.

Лістинг 4.14 – Реалізація класу KingfisherService

```
import UIKit
import Kingfisher
private let kingfisherServiceName = "KingfisherService"
protocol KingfisherService: Service {
    func loadImageFrom(urlString: String, success: @escaping
(Data) -> (), failure: @escaping (KingfisherError) -> ())}
extension KingfisherService {
    var serviceName: String {
        get {       kingfisherServiceName    }    }
    func loadImageFrom(urlString: String, success: @escaping
(Data) -> (), failure: @escaping (KingfisherError) -> ()) {
        guard let url = URL(string: urlString) else {
            return       }
        ImageDownloader.default.downloadImage(with: url,
options: nil, progressBlock: nil) { result in
            switch result {
            case .success(let value):
```

```
                print("Data: \(value.originalData)")
                success(value.originalData)
            case .failure(let error):
                print("Error: \(error)")
                failure(error)  }  }  }  }
class KingfisherServiceImplementation: KingfisherService {
    static func register() {
        ServiceRegistry.add(service: LazyService(serviceName:
kingfisherServiceName, serviceGetter: { () -> Service in
            return KingfisherServiceImplementation()   }))
    }     }
extension ServiceRegistryImplementation {
    var kingfisherService: KingfisherService {
        get {
            return serviceWith(name: kingfisherServiceName) as!
KingfisherService     }  }  }
```

4.5 Розширення елементів UIKit та класів користувача

Swift оперує концепцією, що називається Extension, яка дозволяє розширювати будь-які існуючі Class, Protocolабо Object та додати нову функціональність, не маючи доступ до вихідного коду. Найпоширенішим прикладом такої можливості є розширення візуальних компонентів UIKit.

Елементи, які часто використовуються UIKit як UICollectionView, UITableView надто залежать від делегата шаблону, щоб передати дані, які будуть брати участь у їх побудові. Також це створює проблему в MV-X архітектурах. Таким чином, розробник може перемістити реалізацію для DataSource, Delegate UICollectionView в окреме розширення, щоб очистити основний ViewController. Для цього необхідно створити новий файл з назвою Extension [ім’я ViewController]. У лістингу 4.15 наведено розширення ViewController за допомогою реалізації методів DataSource та Delegate для UICollectionView.

Лістинг 4.15 – Розширення ViewController за допомогою методів DataSource та Delegate для UICollectionView

```
extension ViewController : UICollectionViewDataSource ,
UICollectionViewDelegate , UICollectionViewDelegateFlowLayout  {
      func collectionView(_ collectionView: UICollectionView,
```

```
numberOfItemsInSection section: Int) -> Int {
        return 5    }
    func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell =
collectionView.dequeueReusableCell(withReuseIdentifier: "cell",
for: indexPath) as! PrivateOrderCell
        cell.item = data[indexPath.row]
        return cell    }
    func collectionView(_ collectionView: UICollectionView,
layout collectionViewLayout: UICollectionViewLayout,
sizeForItemAt indexPath: IndexPath) -> CGSize {
        let size  = collectionView.frame
        return  CGSize(width: size.width, height: 100)     }
    func collectionView(_ collectionView: UICollectionView,
layout collectionViewLayout: UICollectionViewLayout,
insetForSectionAt section: Int) -> UIEdgeInsets {
        return UIEdgeInsets(top: 0, left: 0, bottom: 0, right:
0)    }
    func collectionView(_ collectionView: UICollectionView,
layout collectionViewLayout: UICollectionViewLayout,
minimumInteritemSpacingForSectionAt section: Int) -> CGFloat {
        return 0    }
    func collectionView(_ collectionView: UICollectionView,
layout collectionViewLayout: UICollectionViewLayout,
minimumLineSpacingForSectionAt section: Int) -> CGFloat {
        return 10    }   }
```

Також під час розробки програми зазвичай часто буде використовуватися одна і та ж сама група кольорів та шрифтів. І може виникнути необхідність змінити їх під час розробки проекту. Щоб мати змогу не створювати колір чи шрифт у всіх місцях, де він потрібен та мати змогу легко замінити їх виправивши код лише в одному місці, можна створити обчислювану властивість, що виходить за межі UIColor (лістинг 4.16).

Лістинг 4.16 – Розширення

```
extension UIColor {
   static var mainColor : UIColor {
        return UIColor(hexString: "B2924E")    }
    static var appGray : UIColor {
        return UIColor(hexString: "424244")    }
}
```

ВИСНОВОК

Тема розробки моделі мобільного застосунку під платформу iOS є досить цікавою та представляє широке поле для подальших досліджень в галузі розробки мобільного ПЗ. Специфіка даного сегмента полягає в тому, що розробка iOS-додатків повинна проводитися з урахуванням особливостей мобільних пристроїв: відмінностями інтерфейсу, розміром екрану, сенсорним управлінням. Актуальність теми підкреслюється широким спектром можливостей для втілення ідей у вигляді мобільного додатку.

В роботі проведено аналіз існуючих моделей додатку, зроблено оцінку використаних та досліджених підходів, визначено основні переваги та недоліки створеного рішення. Також в роботі були проаналізовао основні інструменти для розробки мобільного програмного забезпечення під платформу iOS. Розглянуто основні засоби розробки, доведено та обгрунтовано правильність вибору цих засобів для проектування. На основі проведеного аналізу встановлено, що найкращим середовищем розробки є Xcode, а мовою програмування – Swift [20]. Адже вони найбільше підходять для розробки додатків під платформу iOS.

Менеджером залежностей було обрано SPM через нативність та інтегрованість в екосистему. На відміну від Cocoapods, більше не обов’язково мати workspace для роботи над проектом із залежностями. Додавши нову залежність, не потрібно робити pod install та перезбирати весь проект. Зручно використовувати локальні залежності також не змінюється структура проекту та швидшевиконується збірка проекту.

Окрему увагу було приділено вибору методу навігації та предачі залежностей в додатку так як це є однією з основних його частин. На основі порівняння був обраний патерн Coordinator для навігації, так як він будує відкрити для масштабування логіку та передача залежностей через DI шо робить додаток максиально надійним через побудову відповідно до SOLID, KISS та DRY принципів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Принципы SOLID, о которых должен знать каждый разработчик. [Електронний ресурс]. Режим доступу: https://medium.com/webbdev/solid-4ffc018077da.

2. Android vs. iOS [Електронний ресурс]. Режим доступу: https://www.diffen.com/difference/ Android_vs_iOS.

3. iOS App Life Cycle. [Електронний ресурс]. Режим доступу: https://medium.com/@neroxiao/ios-app-life-cycle-ec1b31cee9dc.

4. iOS View Controller Life Cycle. [Електронний ресурс]. Режим доступу: https://medium.com/good-morning-swift/ios-view-controller-life-cycle-2a0f02e74ff5.

5. PYPL (Popularity of Programming Language Index). [Електронний ресурс]. Режим доступу: http://pypl.github.io/PYPL.html.

6. Swift vs Objective-C. [Електронний ресурс]. Режим доступу: https://medium.com/swiftify/swift-vs-objective-c-comparison-32aba9dad4e3.

7. A brief post about what Firebase is all about, and it’s new NoSQL Database. [Електронний ресурс]. Режим доступу: https://hackernoon.com/ introduction-to-firebase-218a23186cd7.

8. Нотт Мэтью. Beginning Xcode: Swift Edition, 2014, 20 с.

9. About an Xcode IDE for Apple application development, 2018. [Електронний ресурс]. Режим доступу: https://medium.com/worst-ios-developer/ab8out-an-xcode-ide-for-apple-application-development-68b161088e53.

10. Василий Усов. Swift. Основы разработки приложений под iOS и macOS, 2017, 355 с.

11. Cocoa Application Competencies for iOS. [Електронний ресурс]. Режим доступу: https://developer.apple.com/library/archive/documentation/ General/Conceptual/Devpedia-CocoaApp/Storyboard.html.

12. Шон Льюис, Майк Данн. Native Mobile Development, 2019, 188 с.

13. Sketch Tutorial for iOS Developers. [Електронний ресурс]. Режим

доступу: https://www.raywenderlich.com/1379-sketch-tutorial-for-ios-developers.

14. Рене Кашо, Джош Берлін. Advanced iOS App Architecture (Second Edition): Real-World App Architecture in Swift, 2017, 500 с.

15. Interface Builder Built-In. [Електронний ресурс]. Режим доступу: https://developer.apple.com/ xcode/interface-builder.

16. Роберт Мартін. Чистий код, 2014, 30 с.

17. Грей Марлин. Adaptive Code: Agile Coding with Design Patterns and SOLID Principles , 2016, 150 с.

18. Роберт Вайт. Оптимізація мобільних застосунків, 2018, 223 с.

19. Чарльз Скот, Барбара Ленз. Ідеальний програміст, 2011, 328 с.

20. Філімончук Т.В., Оцевик В.А. Модель мобільного застосунку iOS. Проблеми інформатизації: Тези доповідей дев’ятої міжнародної науково-технічної конференції. Том 2, секція 4. Черкаси – Харків – Баку – Бельсько-Бяла. 2021. С. 98.