Міністерство освіти і науки України НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Розробка мобільного додатку для ведення обліку фінансів мовою Swift для платформи iOS

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
Борозенний С. О.
(підпис)
“____” _____ 2022 р.

Виконала студентка
Волошенюк І. М.
“____” _____ 2022 р.

Київ 2022

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,

Доцент., к. ф.-м. н. С.С.Гороховський (підпис)

„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу студентці Волошенюк Ірині Михайлівні факультету інформатики 4-го курсу ТЕМА Розробка мобільного додатку під iOS з використанням Swift

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Розділ 1. Аналіз предметної області та постановка завдання
6. Розділ 2. Теоретичні відомості
7. Розділ 3. Опис реалізації програмного продукту
8. Висновки
9. Список використаних джерел

Дата видачі “___” _____ 2022 р. Керівник Борозенний С. О. (підпис)

Завдання отримала Волошенюк І. М. (підпис)

Тема: Принципи роботи технології блокчейн та її властивості

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи.	15.10.2020	
2.	Огляд технології	15.11.2020	
3.	Написання основної частини	20.03.2021	
4.	Розробка додатку	20.04.2021	
5.	Завершення написання текстової частини курсової	25.04.2021	
6.	Корегування роботи згідно із зауваженнями керівника	02.05.2021	
7.	Створення презентації	15.05.2021	

Студенту Волошенюк І. М. (підпис)

Керівник Борозенний С. О. (підпис)

“___” _____ 2022 р

АНОТАЦІЯ

Дана курсова робота присвячена розробці мобільного додатку для операційної системи iOS з використанням мови програмування Swift. У ході роботи було досліджено особливості розробки додатків для мобільної платформи. Окрім того, було проаналізовано наявні інструменти та архітектурні патерни для якомога ефективнішої реалізації поставленої задачі. На основі отриманих знань, було розроблено мобільний застосунок для ведення обліку фінансів.

Ключові слова: мобільний застосунок, iOS, Swift, Core Data, UIKit

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ ТЕРМІНІВ І СКОРОЧЕНЬ

ORM – англ. “Object-Relation Mapping”.

MVC – англ. “Model View Controller”.

MVP – англ. “Model View Presenter”.

MVVM – англ. “Model View ViewModel”.

MVC+R – англ. “Model View Controller + Router”.

MVVM+R – англ. “Model View ViewModel + Router”.

UI – англ. “User Interface”.

HIG – англ. “Human Interface Guidlines”.

ОС – операційна система

iOS – англ. “iPhone Operating System”.

WatchOS – англ. “Watch Operating System”.

MacOS – англ. “Mac Operating System”.

iPadOS – англ. “iPad Operating System”.

ARM – англ. “Advanced RISC Machine”.

IPv6 – англ. “Internet Protocol version 6”.

СКБД – “Систе́ма управління ба́зами да́них”.

SQL – англ. “Structured Query Language”.

NoSQL – англ. “No Structured Query Language”.

JSON – англ. “JavaScript Object Notation”.

GCD – англ. “Grand Central Dispatch”.

API – англ. “Application Programming Interface”.

APNs – англ. “Apple Push Notification service”.

POP – англ. “Protocol-Oriented Programming”.

ARC – англ. “Automatic Reference Counting”.

Вступ

Протягом багатьох століть гроші були невід'ємною частиною життя людей. Вони супроводжують нас на кожному кроці, роблячи наше життя простішим. З появою грошей, з'явилась і проблема: як краще розпоряджатись ними, щоб позбутись надмірних витрат та мати загальну картину видатків. Розвиток технологій надав можливість для спрощення та автоматизації ведення обліку витрат.

Ще одним нерозлучним компаньйоном людини, з недавніх пір, став мобільний телефон. Його портативність та продуктивність дозволили йому стати ідеальним помічником. Завдяки тому, що він завжди є під рукою, це ідеальний рішення для відслідковування надходжень та витрат.

Об'єктом дослідження стало створення додатку під мобільну платформу iOS.

Предмет дослідження - це особливості роботи операційної системи iOS їхній вплив на створення застосунків для неї, характеристики мови програмування Swift, види баз даних, архітектурних патернів та фреймворків для створення графічного інтерфейсу. Крім того було розглянуто патерни, відмінності практичного використання парадигм ООП і ПОП та особливості роботи з нотифікаціями.

Метою даної курсової роботи стало створення мобільного додатку під платформу iOS для спрощення відслідковування грошового потоку та ведення обліку витрат та надходжень.

У ході роботи описано результати дослідження актуальності додатків для введення обліку фінансів. Зокрема, було проаналізовано болі користувачів та особливості сприйняття інформації людиною. На основі цього, було виділено функціонал, що повинен містити додаток. На рівні з функціональними вимогами, це покликано вирішити проблеми користувача, пов'язані зі складністю відслідковування грошового потоку.

Другий розділ присвячено огляду теоретичних відомостей, що призначено для пошуку найбільш ефективного способу реалізації технічного завдання,

поставленого в попередньому розділі. Особливості роботи платформи iOS було розглянуто з точки зору 4 ключових понять: життєвий цикл додатка, особливості роботи графічного інтерфейсу в багатопоточному середовищі, забезпечення стабільності роботи та безпеки, за допомогою Watchdog та Sandbox. Оскільки, єдиним легальним способом встановлення додатку на iOS є завантаження з магазину App Store, також було розглянути вимоги, встановлені до публікованих додатків.

Останній розділ описує практичні аспекти створення додатку, обґрунтування обраних технологій та особливості роботи з ними.

Розділ 1. Аналіз предметної області та постановка завдання

1.1 Актуальність та аналіз предметної області

Сьогодні мобільний телефон став невід'ємною частиною життя кожної людини. Першочергова задача мобільних пристроїв – підтримка зв'язку з оточенням – зараз відійшла на другий план. Телефони стали нашими компаньйонами та асистентами. Згідно з дослідженнями від лютого 2021 року, близько 50% людей проводять 5-6 годин за смартфонами. І дана цифра збільшується з кожним роком. [1]

Ще однією річчю щоденного вжитку є гроші. Від купівлі жуйки в кіоску, до придбання комп'ютерних ігор – наше життя сповнене безлічі витрат. Серед такого різноманіття дуже легко загубитись та не помітити як велика кількість грошей витрачається неефективно. Особливо питання ведення фінансів ускладнюється, коли у людини з'являється сім'я та діти. Дана курсова робота призначена, щоб допомогти уникнути зайвих витрат та збільшити контроль над власними фінансами шляхом надання зручного механізму внесення щоденних витрат з подальшим переглядом статистики. У якості платформи було обрано мобільні пристрої. Завдяки тому, що смартфони завжди під рукою, ведення фінансів стає легше, оскільки користувач може внести витрату не відходячи від каси. Так само, він у будь-який момент може проаналізувати поточні витрати та прийняти рішення щодо доцільності потенційної покупки.

З кожним днем з'являються все нові бізнес-моделі та способи отримання прибутку. Сьогодні популярним способом розрахунку за послуги є підписка. Замість одноразового платежу, людина зобов'язується раз на певний термін платити фіксовану суму. Зачасту, списання коштів відбувається автоматично. Це видається зручним, якщо людина має декілька підписок. Але якщо їх багато це стає справжньою проблемою. В один момент, людина може прийти до того, що не розуміє за що у неї знімаються гроші з картки. Для уникнення подібних ситуацій,

даний додаток містить механізм для попередження користувача про регулярний платіж.

Особливості роботи мозку людину призводять до того, що розпізнавання візуальної інформації відбувається у 30 разів легше ніж тексту. [2] [3] Таким чином, користувачам набагато легше оперувати з графіками і діаграмами, ніж з табличними даними. Завдяки візуалізації даних, людина може легше порівнювати різні статті витрат та порівнювати об’єм витрачених коштів порівняно з аналогічними періодами в минулому, що допомагає не витрачати непомірну кількість власних коштів на неважливі категорії та заощаджувати (якщо це необхідно).

Іноді нам необхідно ввести облік не лише власних витрат або відділити витрати за певний період (відпустка, відрядження тощо), для легшого подальшого аналізу. Для цього додаток передбачає створення декількох бюджетів та легке переключення між ними. Таким чином, усі витрати можуть бути розділені не лише по витратам, але і по бюджетам.

1.2 Аналіз існуючих рішень та конкурентів

Створення конкурентоспроможного проєкту починається з аналізу ринку та пошуку потенційних аналогів. Завдяки цьому, є можливість проаналізувати переваги та недоліки існуючих рішень та розробити якісно нове рішення. Також при аналізі конкурентів, особливу увагу було приділено плавності роботи інтерфейсу та наявності багів. Для дослідження було обрано безкоштовні застосунки, що входять в топ-3 популярних додатків для ведення обліку фінансів магазину App Store. Далі будуть перечисленні основні переваги на недоліки кожного застосунку.

1.2.1 Coin Keeper

Застосунок Coin Keeper має рейтинг 4.7 та найбільшу кількість відгуків серед розглянутих мною додатків. До переваг застосунку можна віднести зручний механізм drag-and-drop (з англ. «потягни-та-відпусти»), що дозволяє автоматично встановлювати джерело та категорію для витрати. Також є зручний екран для перегляну усіх джерел надходжень та категорій. При створенні транзакції можна вказати суму, дату, тег та час наступного платежу, якщо вона є повторюваною. Вибір валюти та додання коментаря недоступні. Функціональні можливості безкоштовної версії додатка дуже обмежені: відсутня як загальна статистика, так й історія транзакцій. На мою думку, подібна обмеженість можливостей робить застосунок непридатним для ефективного аналізу витрат.

1.2.2 Money Lover

Money Lover, з рейтингом 4.7, пропонує найбільшу кількість можливостей серед описаних мною конкурентів. Цікавою особливістю даного застосунку є те, що при створенні транзакції, її можна приховати з загальної статистики та історії. Велика кількість функціональності призвела до серйозного недоліку – інтерфейс додатку виглядає перевантаженим та не інтуїтивно зрозумілим. Також до недоліків я б віднесла нав’язування допомоги при початковому налаштуванні бюджетів, яку неможливо пропустити. Судячи з відгуків, надмірна кількість можливостей, що надає застосунок, замість того, щоб приваблювати користувачів, відштовхує їх. Це спричинено тим, що тривіальні речі, такі як додання транзакції чи перегляд історії, займають набагато більше часу порівняно з іншими подібними додатками.

1.2.3 Coinpath

Додаток Coinpath має рейтинг 4.4 та приваблює своїм лаконічним дизайном, виконаним в чорно-білій гамі, та плавними анімаціями. Окрім безпосереднього ведення обліку витрат шляхом додання транзакцій, застосунок надає можливість гнучкого пошуку по історії. На жаль, статистика доступна лише в межах одного дня, що не дозволяє зрозуміти загальну картину витрат. Окрім цього, додаток дозволяє створювати декілька бюджетів та легко між ними переключатись. Даний застосунок буде гарним вибором для людей, що віддають перевагу лаконічності. Функціонал додатку дозволяє легко вести облік витрат, але не підходить для ситуацій, коли користувач має справу з повторюваними транзакціями або різними валютами.

1.3 Постановка завдання курсової роботи

Завданням даної курсової роботи є створення мобільного додатку для iOS, що дозволить вирішити болі користувача. При аналізі конкурентів, було виявлено, що при порушені балансу між функціональністю та зручністю інтерфейсу, користувацький досвід стрімко погіршується. У ході дослідження ринку та потреб користувачів, було виділено наступні можливості, що призначені вирішити болі користувачів:

- створення транзакцій з можливістю вказання суми, категорії, часу та коментаря;
- створення категорій з вказанням назви, ліміту (певна сума грошей, яку ми плануємо витратити) та іконки;
- відображення історії транзакцій з фільтрами, такими як: тип (витрати/надходження) та час (день/місяць/весь час);
- відображення статистики в кінці місяця у вигляді діаграми чи графіка;
- регулярні платежі та нагадування;

- створення декількох бюджетів;
- мультивалютність, з «підтягуванням» актуального курсу валют.

Також при реалізації застосунку накладаються певні вимоги:

- дизайн повинен бути інтуїтивно зрозумілим та розробленим згідно стандартів HIG;
- дизайн повинен бути адаптивним та коректно відображатись на усіх підтримуваних моделях при портретній та альбомній орієнтації екрана; при тому елементи не мають накладатись одне на одного, бути менше, зазначеного в HIG, розміру та «розміщуватись» за межами екрану;
- робота додатку повинна бути плавною, без надмірного споживання ресурсів пристрою.

Для ефективної реалізації додатку, необхідно дослідити особливості мови програмування Swift та розробки для платформи iOS в цілому. Окрім цього, потрібно проаналізувати існуючі архітектурні патерни для мобільних додатків та обрати найбільш оптимальний. Виходячи з особливостей застосунку, також потрібно вибрати базу даних та фреймворк для створення графічного інтерфейсу.

1.4 Висновки до розділу 1

У даному розділі було проаналізовано актуальність такої теми, як ведення обліку витрат та виділено проблеми, що вона здатна вирішити. Для кращого розуміння потреб користувача та збільшення конкурентоспроможності додатку було розглянуто наявні на ринку рішення. У ході дослідження, було виявлено, що більшість конкурентів не мають достатньої кількості функціональних можливостей для вирішення усіх болей користувача або мають надто складний інтерфейс, що ускладнює взаємодію з ним. Таким чином, було виділено можливості, які повинен надавати додаток та сформульовано функціональні вимоги

Розділ 2. Теоретичні відомості

2.1 Особливості розробки для платформи iOS

Технічні характеристики кожної платформи породжують низку особливостей пов'язаних з розробкою застосунків для них. Апаратні обмеження змушують шукати підходи для більш ефективної реалізації кожної задачі. Компактний розмір мобільних пристроїв призводить до обмеженого розміру акумулятора, гіршої теплопровідності та вентиляції, а компактний розмір процесора та логічних ядер призводить до більш скромних показників продуктивності. Також при розробці застосунків, необхідно враховувати певні нюанси роботи операційної системи та взаємодії користувача зі смартфоном. Так, застосунки що працюють у фоновому режимі, отримують менше ресурсів, ніж та, з якою в даний момент часу активно взаємодіє користувач. Крім того, серйозні обмеження накладаються на енергоефективність програми, оскільки смартфон, окрім того, що зазвичай не має акумулятора великої ємності, ще може працювати при великих перепадах температур (наприклад, зимою і влітку), що значно погіршує ємність енергоносія.

Додаткові обмеження може накладати операційна система. Так, розробка додатків для Android, має певні відмінності від створення програм для iOS. Це пов'язано як з особливостями реалізації кожної з платформ, так і корпоративними обмеженнями. Прикладом таких критеріїв може слугувати HIG – збірка рекомендацій при створені графічного інтерфейса, що має на меті покращення користувацького досвіду шляхом надання правил щодо розміру, розміщення, кількості, кольору тощо графічних елементів, що має спростити їх візуальне розпізнавання користувачем. [7]

2.1.1 Життєвий цикл додатка

Концептуальна різниця розробки для iOS починається з того, що програми прийнято називати додатками. Це пов'язано з тим, що операційна система є самостійною і головною, і ми в праві лише додавати певний функціонал до неї. Взаємодія додатку з ОС відбувається декларативно. Ми можемо підписуватись на певні події та декларувати бажані дії. Операційна система сама обирає як, в який момент часу і що зробити. Те саме стосується і графічного інтерфейсу. При використанні AutoLayout, ми лише можемо вказувати відносне положення елементів та їхній пріоритет. Залежно розміру дисплею та орієнтації, ОС самостійно приймає рішення щодо того, як краще відмалювати UI.

Важливим елементом є життєвий цикл програми. Включно з iOS 12, ми можемо працювати зі станами або зі сценами. Ці два підходи відрізняються лише переходами між станами програми, самі ж стани лишаються однаковими. Розглянемо більш новий підхід.

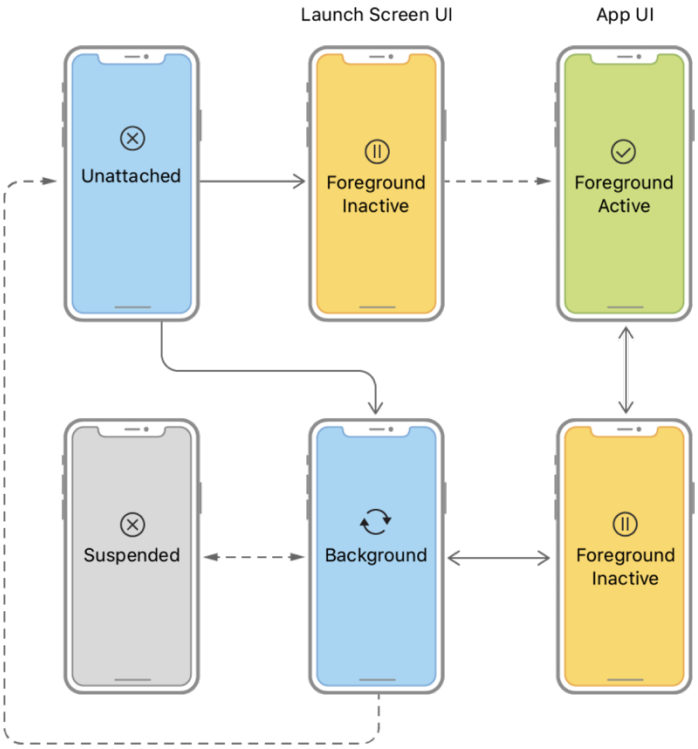

Рисунок 1.1 Стани програми у iOS 12+

Виділяють наступні стани:

- Unattached – стан програми, що не працює, код не завантажено в оперативну пам’ять,
- Foreground Inactive – стан програми, що готується до відображення; на даному етапі взаємодія з користувачем ще не відбувається; даний етап слугує для підготовки додатку до взаємодії з користувачем,
- Foreground Active - стан програми, що працює; обробляються дотики і жести користувача,
- Background – стан програми, що працює к фоні; при цьому інтерфейс не відображається, але програма може ще виконуватись певний час,
- Suspended – стан програми, що не працює; код застосунку досі завантажено в оперативну пам’ять.

Для кожного додатку, згідно з його станом, виділяється певна кількість оперативної пам’яті та процесорного часу. При спробі виходу за цей ліміт, ОС попередить програму, надіславши memoryWarning. Якщо програма не відреагує на нього та не зупинить ресурсоємні процеси, її буде аварійно завершено. Найбільше ресурсів виділяється додатку, що перебуває у стані Foreground Active (до iOS 12 - Active). Саме з ним користувач активно взаємодіє в даний момент часу.

При виході з стану Foreground Active, програма мусить вивільнити ресурси та підготуватись до завершення роботи (зберегти зміни до бази даних, завершити сесію, закрити сокет тощо). Це пов’язано з тим, що додатки на iOS не виконуються у фоні постійно, задля економії ресурсів та заряду акумулятора. Крім цього, необхідно бути готовим до того, що перехід у фоновий режим може відбуватись неочікувано – наприклад, при отримані вхідного дзвінка. Якщо додаток містить чутливу інформацію (банківський застосунок, тощо), при переході у фоновий режим, він має підготувати програму до snapshot (з англ. «моментальний знімок») – знімку інтерфейсу, що буде відображатись у tray (з англ. «піднос, корзинка»). Якщо це, на приклад, банківська програма, то необхідно приховати значення

балансу та транзакції, щоб убезпечити користувача від випадкової демонстрації приватної інформації.

Якщо програма не запущена, за потреби, оперативна система може ненадовго викликати її у фоновому режимі для обробки певної події (зміна геолокації, push-нотифікація тощо). Перелік дозволених дій у фоновому режимі обмежений. Це зроблено задля безпеки та збереження приватності користувача. Таким чином, жодна програма, що працює у фоні, не має можливості знімати фото/відео чи робити будь-які інші дії, що б могли нашкодити користувачу.

2.1.2 Багатопоточність

При створенні мобільного застосунку, постає необхідність роботи в багатопотоковому середовищі. Завдяки цьому досягається більш ефективне використання процесорного часу. Також, що не менш важливо, головний потік, що відповідає за відображення графічного інтерфейсу, може бути вивільнений від виконання затратних операцій, що дозволяє досягти плавної роботи UI.

Для роботи з операціями та чергами виконання мова Swift надає бібліотеку Grand Central Dispatch (GCD) для декларативної роботи в багатопотоковому середовищі. [9] Розробник може створювати черги самостійно або використовувати наперед створені глобальні черги DispatchQueue. Для легшої роботи, можна організовувати задачі в групи – DispatchGroup. Вони дозволяють синхронізувати виконання тасків, та спростити роботу з ними.

Важливою частиною розробки застосунку являється правильна робота з головною чергою DispatchQueue.main, оскільки вона відповідає за відмалювання UI. Виконання

Також варто враховувати, що додавати задачі в чергу виконання можна синхронно та асинхронно, за допомогою методів sync та async відповідно. При синхронному виконанні, задача додається з кінця черги. Після цього, поточний потік блокується, в очікуванні виконання задачі. При асинхронному виконанні,

різниця полягає в тому, що поточний потік не блокується. Варто бути обережним при роботі з головним потоком, оскільки виклик метода sync на ньому призведе до аварійного завершення роботи.

При розподілені задач по чергам виконанням, необхідно враховувати пріоритет. GCD надає 6 різних пріоритетів для налаштування гнучкої роботи з потоками: userInteractive, userInitiated, default, utility, background, unspecified. Процесорний час розподіляється згідно пріоритету. Завдяки цьому, ми можемо розподілити завдання згідно їх важливості.

Особливістю DispatchQueue є те, що додану задачу неможливо відмінити. Для більш гнучкої роботи існує OperationQueue, що є надбудовою над DispatchQueue. Вона використовує машину станів для відображення різних можливих станів задачі та переходів між ними. Операція може мати один із 5 станів: pending, ready, executing, finished, cancelled. При тому порядок виконання задач залежить від пріоритету та степені їх готовності. [10]

2.1.3 Підтримувані операційні системи та їх версії

При розробці графічного інтерфейсу, варто враховувати перелік пристроїв, що зможуть використовувати ваш додаток. Таким чином, розміщаючи елементи UI та налаштовуючи їхнє положення, необхідно передбачити як вони будуть виглядати на екранах з різною діагоналлю. Окрім цього, необхідно враховувати, що додаток може працювати при різній орієнтації.

Ще одним важливим моментом є підтримувані операційні системи та їх версії. Екосистема Apple дозволяє розробляти один універсальний додаток, що буде працювати на iPhone, iPad та Mac. Підтримка MacOS досягається завдяки проєкту Catalyst (з англ. «каталізатор») – платформи, що дозволяє розробляти додатки за допомогою мобільних фреймворків UIKit та SwiftUI, які нативно працюють на десктопі. До цього, для створення десктопних застосунків, був доступний лише фреймворк AppKit. З появою комп’ютерів Apple, що

використовують архітектуру ARM – ту ж, що використовується у мобільних пристроях – підтримка мобільних додатків на MacOS стала ще актуальнішою. [5]

При розробці універсальних додатків, необхідно приділити велику увагу дизайну. У зв'язку з різним розміром дисплеїв телефонів, планшетів та комп'ютерів, підхід до відображення інформації теж різниться. Якщо використати дизайн, притаманний для iPhone, при створені додатку для iPad, ми зіткнемось з проблемою, що графічні елементи, які виглядали елегантно і доречно на телефоні, на значно більшому екрані виглядають по-дитячому незграбними. У випадку iPadOS також необхідно врахувати, що дана операційна система підтримує одночасне використання декількох додатків (технологія Split Screen). Тобто, перед розробником та дизайнером постає ще складніша задача - змусити програму виглядати гарно та зручно, навіть коли вона займає не 100% екрану. При роботі з комп'ютерами, необхідно передбачити те, що дисплей не є сенсорним, а отже, і взаємодія з контентом відбувається по-іншому. Різняться також і жести, що використовуються для сенсорного екрану мобільного пристрою та тачпаду. [8]

2.1.4 Безпека: Sandbox і Watchdog

Якщо попередні особливості, у тій чи іншій мірі, стосуються усіх мобільних платформ, то наступна характеристика притаманна лише iOS та iPadOS – кожний додаток працює у так званому sandbox (з англ. «пісочниця»). Sandbox – це контейнер, що відділяє дисковий простір доступний програмі, від решти пам'яті. Це створено з огляду на безпеку. Таким чином, жоден додаток не здійснити несанкціонований доступ до файлів, що не належать йому. Це захищає систему від різноманітного шкідливого програмного забезпечення та несанкціонованого поширення користувацьких даних з іншими програмами. Також обмежується доступ до налаштувань, мережний та апаратних ресурсів. При видалені додатку, користувач також видає усі дані, що містились в його sandbox. Це дозволяє переконатись, що пам'ять пристроя не буде заповнена даними, що вже не

використовуються. При необхідності, об'єм пам'яті, що виділена для sandbox-у конкретного додатку може бути збільшено. Максимальний розмір sandbox обмежений лише загальним об'ємом пам'яті на пристрої. [4]

На варті користувацького досвіду та безпеки стоїть Watchdog (з англ. «сторожовий пес») – спеціальна програма, що слідкує за роботою усіх додатків. До його обов'язків входить:

- перевірка дотримання мінімальних стандартів якості,
- забезпечення плавної роботи додатка,
- перевірка кількості використаних ресурсів,
- перевірка використовуваних ресурсів у фоні.

При будь-якій забороненій дії, Watchdog аварійно завершить програму. Саме він не дозволяє несанкціонований доступ до ресурсів у фоні та слідкує за тим, щоб процесорний час та пам'ять були правильно розподілені між усіма запущеними додатками.

2.1.5 Публікація в App Store

Ще одним етапом захисту є власна перевірка Apple. На відміну від операційної системи Android, додатки для платформи iOS можуть бути завантажені лише з офіційного магазину App Store. Перед публікацією кожної програми, її ретельно перевіряють на дотримання стандартів якості та внутрішніх критеріїв. Усі вимоги висвітлені на офіційному сайті Apple. За спробу обдурити систему перевірки, викрадення даних користувача, плагіат, зловживання системою рейтинга тощо, програму буде негайно видалено з магазину, а розробнику – заборонено публікувати. Таким чином, усі додатки перед публікацією проходять автоматичну та мануальну перевірку на дотримання 5 ключових критеріїв: безпека, продуктивність, бізнес, дизайн та законність.

Почнемо з найважливішого параметру – безпека. Усі користувачі, що встановлюють додатки з App Store, можуть бути впевненими – жодна програма не зможе нашкодити ні їхньому пристрою, ні їм самим. Окрім очевидної перевірки на рахунок того, чи правильно застосунок зберігає чутливу інформацію і чи не передає її третім лицям, перевірці підлягає і контент: чи не містить він образливої, шокуючої, огидної інформації. Найбільшої модерації підлягають додатки призначені для дитячої аудиторії. Корпоративна політика Apple не допускає наявності жодного контенту, що може якимось чином нашкодити фізичному або психологічному здоров'ю дитини. Крім того, передбачається, що дитина не повинна мати можливість якимось чином переходити з даного додатка на сторонній сайт та/або програму, здійснювати платежі тощо. Особливістю програм для дітей є те, що навіть якщо розробник більше не хоче позиціонувати власний додаток у категорії Kids (з англ. «дитячий»), він мусить у подальших оновленнях продовжувати задовільняти критерії додатку для дітей. Ця вимога покликана захистити неповнолітніх користувачів, що вже встановили даний додаток.

Наступним не менш важливим критерієм є продуктивність. Філософія компанії передбачає, що користування пристроєм має відбуватись плавно, без жодних підвисань та багів, незалежно від дій користувача. Для забезпечення цього, кожний додаток перевіряється на дотримання мінімальних стандартів якості. Додатки повинні бути енергоефективними, не використовувати надмірно ресурси пристрою, провокувати перегрівання тощо. Більш того, жодна програма не повинна потребувати/спричиняти перезавантаження пристрою та вносити зміни до системних файлів. Також, як ми говорили раніше, кожен додаток мусить функціонувати в рамках створеного системою контейнера Sandbox, та не має права читати/писати дані поза його межами. Наявність вірусів, троянів та будь-якого шкідливого програмного забезпечення неприпустима. Накладаються вимоги і на мережні технології: додатки повинні повністю функціонувати на стандарті IPv6. Окрім того, Apple закликає, наскільки це можливо, створювати універсальні додатки, що працюють як на iPhone так і iPad.

Більшість додатків створюються з метою отримання вигоди. App Store надає широкий перелік можливості для монетизації вашого застосунку. Однак, усі бізнес-моделі підлягають ретельній перевірці і, у разі неочевидної схеми отримання вигоди, можуть бути відхиленими. Будь-які платежі повинні проходити в самому додатку за допомогою вбудованого механізму від Apple. Перехід по стороннім посиланням, банківські перекази, сканування QR-кодів для здійснення оплати суворо заборонено. Правила App Store допускають використання криптовалют у додатках. Це можуть бути як крипто-гаманці, біржі чи інші програми, що створені для зберігання, продажу та обміну криптовалют. Однак, майнінг – процес отримання криптовалют в обмін на використання процесорного, дискового простору або будь-якого іншого ресурса пристроя – строго заборонений. Крім тощо, не допускається отримання криптовалюти за виконання завдань, завантаження сторонніх додатків, публікація контенту до соціальних мереж тощо.

Ще однією допустимою бізнес-моделлю є підписка. Згідно з правилами App Store, підписка повинна діяти на усіх пристроях користувача, де встановлено даний додаток. Після оплати підписки, людина повинна мати доступ до оплаченого контенту без потреби виконувати будь-які додаткові дії, як от виставлення гарного рейтингу застосунку. Перед тим, як запропонувати користувачу оплатити підписку, необхідно надати йому вичерпну інформацію, щодо правил користування та контенту, що він отримає. У будь-який момент часу, користувач повинен мати змогу відмінити або відновити підписку. Крім цього, додаток не повинен жодним чином примушувати користувача виставляти програмі гарний рейтинг, завантажувати сторонні застосунки тощо.

Розробник має право розміщувати рекламу у додатку. Однак, на неї теж накладаються певні обмеження. Реклама має відповідати встановленому віковому цензу. Заборонено розміщувати будь-які рекламні оголошення в клавіатурах, віджетах та сповіщеннях. Таргетування також підлягає під певні правила: заборонено використовувати чутливу інформацію, таку як медичні показники, (наприклад, отримані від HealthKit API), інформацію про навчальний заклад тощо.

Усі рекламні банери повинні містити достатньо велику кнопку закриття, щоб користувач міг легко на неї натиснути.

Не менш прискіпливо перевіряють дизайн додатку. Розробник сам визначає скільки часу і зусиль він прагне приділити візуальній складовій програмі, але кожна програма мусить відповідати мінімальним стандартам якості, що встановлені App Store. Це стосується не лише публікації, а і подальшої підтримки додатку. Застосунок, що припинив працювати, перестав відповідати оновленим стандартам якості та/або надають погіршений досвід (degraded experience) можуть бути видаленими. Окрім цього існує цілий перелік правил і рекомендацій. По-перше, не допускається жоден плагіат. Також застосунок повинен мати певну розважальну, інформативну чи навчальну цінність. Кожен додаток самостійний і не має вимагати встановлення сторонньої програми чи контенту для своєї роботи. Якщо програма надає можливість авторизації за допомогою стороннього сервіса, наприклад Twitter, Facebook, LinkedIn, Amazon, Google тощо, вона також повинна надавати механізм для авторизації з використанням акаунту Apple, на рівні з рештою сервісів.

Крім власне внутрішніх правил Apple, додаток повинен не протирічити законодавству країн, на території яких він доступний. Окрім цього, усі додатки зобов'язуються містити посилання на правила приватності, яких вони дотримуються: детальний опис даних, як вони збираються, використовуються та поширюються зі сторонніми сервісами. Якщо додаток ділиться будь-якою інформацією з сторонніми сервісами, повинна бути гарантія, що вони також дотримуються правил зберігання на використання користувацьких даних. Програми повинні запрошувати лише дані, що необхідні для їхнього функціонування. Усі додатки мають поважати налаштування приватності, встановлені користувачем і не намагатись будь-яким чином їх обійти. Крім цього, перед публікацією необхідно пересвідчитись, що додаток містить лише вашу власну інтелектуальну власність, або контент, на який ви маєте ліцензію. Також не допускається сприяння незаконному поширенню контенту та зберігання,

конвертація або зберігання медіа файлів зі сторонніх сервісів без явної авторизації. [6]

2.2 Особливості мови програмування Swift

При створені нативних застосунків для платформи iOS розробник може обрати в якості мови програмування Objective-C та Swift.

Objective-C розроблена у 1988 році та являє собою надбудовою над мовою C. Завдяки підтримці ООП та використаню динамічного рантайму (runtime), вона швидко здобула популярність. У 2014 році Apple презентувала альтернативу – Swift. Щоб не позбавляти розробників великої кількості фреймворків (серед який Cocoa та Cocoa Touch) та інструментів, розроблених для Objective-C, у Swift додали повну сумісність з попередньою мовою. Також є підтримка змішаних проєктів, написаних з використанням обох мов.

На відміну від Objective-C, Swift - власна розробка компанії. Попри те, що концептуально дві мови схожі, Swift привнесла в розробку на iOS важливі зміни. Одна із ключових переваг - це продуктивність. Нова мова в 2.6 разів швидша за Objective-C. Зміни торкнулись також і синтаксису. Він став прощений, з’явилось більше «синтаксичного цукру», що допомогло збільшити читабельність коду. Крім цього, у мові Swift відмовились від розділення на інтерефейс (header file) та файли імплементації (implementation file). Це спрощує підтримку та внесення змін у код, оскільки необхідно редагувати лише один файл.

Завдяки вищеописаним перевагам, Swift завоював визнання серед спільноти програмістів та швидко витіснив попередника. На сьогоднішній момент, це основна мова розробки для платформ Apple. [11][12]

Swift – це крос-платформна, open-source, компільована мова. Ключовою її відмінністю, порівняно з конкурентами, є безпека. При розробці мовою Swift, в основному, ми користуємось відсилками, подібно до того як це було прийнято в Objective-C. Однак, сама робота з відсилками повністю переосмислена. Для

позначення відсутності значення у Swift використовується літерал nil. Це є поле .none для обгортки Optional конкретного типу, на відміну від Objective-C, де nil позначав посилання на неіснуючий об'єкт. Завдяки цьому нововведенню, значення nil може набувати усі опціональні типи, не лише відсилки. Для розгортання опціональної змінної у мові Swift наявні конструкції «if let/var» та «guard let/var». [13][14]

На швидкодію мови сильно впливає обрана модель управляння пам'яттю. Swift використовує ARC (англ. "Automatic Reference Counting"). Принцип роботи наступний: під час виконання програми будується граф об'єктів, для відстеження наявності посилань між об'єктами. При тому кожен елемент графу має власний лічильник, що позначає кількість посилань на даний об'єкт. Коли лічильник набуває значення 0, об'єкт автоматично видаляється. Завдяки даному підходу, вивільнення пам'яті відбувається поступово, а не за кумулятивним принципом, як при використанні збиральників сміття (garbage collector). При цьому видалення об'єктів відбувається автоматично – це відповідальність Swift - що значно полегшує розробку, порівняно з використанням ручного керування пам'яттю та розумних указників (smart pointers), що застосовувались в Objective-C. Таким чином, при правильному використанні ARC, розробник отримує швидкодію в поєднанні з автоматизованістю при роботі з пам'яттю.

Розуміючи особливості роботи ARC необхідно бути особливо обережним, щоб не допустити витоку пам'яті. Найбільш небезпечним є наявність замкненого кола (retain cycle), що утворюється при взаємному посиланні двох об'єктів одне на одного. Таким чином, їхні лічильники завжди будуть більше рівні 1 і ARC не зможе видалити їх з пам'яті. Щоб уникнути цього, Swift надає різні види відсилок: strong, weak, unowned(safe), unowned(unsafe). Найчастіше використовуються перші два. Основною відмінністю weak від strong є те, що він не збільшує значення лічильника об'єкта. Тому прийнятою практикою, при посиланні об'єктів одне на одного, одну з відсилок позначати як weak. [15]

З метою оптимізації та збільшення продуктивності, при роботі з колекціями та рядками використовується механізм Copy-on-Write (з англ. “Копіювання-при-Записі”). Завдяки цьому, справжнє копіювання об’єктів відбувається лише при внесенні змін до дублікату – до того часу, копія посилається на комірку пам’яті оригіналу. [16]

Оскільки, Swift підтримує ООП, мова містить також і механізм наслідування. На відміну від деяких мов, таких як Java, Swift не має базового універсального класу. Тому додаток може містити безліч базових класів за потреби. Також Swift не має абстрактних та віртуальних методів. Це обмеження можна обійти задекларувавши метод з порожнім тілом. Процес ініціалізації проходить в 2 фази. Спочатку відбувається ініціалізація усіх полів від дочірнього класу до батьківського. Після того, як усі збережувальні властивості проініціалізовані, наступає наступна фаза, під час якої відбувається доналаштування. На цьому етапі, властивості та методи доступні для роботи. Swift надає 2 види ініціалізаторів: призначені (designated) та для зручності (convenience). Обмеження на кількість ініціалізаторів не накладається. [17] [18]

Як і мови сімейства C, функції у Swift мають свій тип. Це дозволяє передавати їх у якості параметрів до вищого порядку, повертати як результат виконання, присвоювати змінним та вкладати в інші функції. Також розробнику доступні замикання (closure), що являють собою функції, що не мають імені. Замикання дозволяють захоплювати значення, що містяться в зовнішньому контексті. Завдяки цьому, їх часто використовуються для колбеків (callback). Наприклад, при використанні голосового помічника Siri, якщо користувач хоче використати функції іншої програми (замовити таксі, подивитись погоду тощо), системі необхідно запустити дану програму ненадовго в фоновому режимі. Після виконання запиту, Siri демонструє результат виконання. Однак, якщо запит потребує багато часу на обробку, це може призвести до «заторможування» інтерфейсу Siri. У попередніх розділах було зазначено, що Watchdog слідкує за тим, щоб усі програми працювали плавно, і в іншому випадку, додаток буде аварійно

завершено. Щоб уникнути цього, Siri передає в сторонню програму замикання. Після того, як запит було оброблено, Siri викликає замикання, аби він надіслав результат. Таким чином, зникає потреба чекати на виконання функції і інтерфейс залишається доступ для взаємодії з користувачем. [19][20]

Ще однією особливістю Swift є використання непрозорих типів (opaque type), які створені для фреймворку SwiftUI, і доступні з версії iOS 13. Завдяки ним, ми можемо абстрагуватись від конкретного типу, що повертає функція. У цьому контексті, нам важливо лише який протокол задовольняє результат. Однак, компілятор точно знає який тип повертається. Відмінністю від узагальнення є те, що всередині функції не відбувається абстрагування від конкретного типу. [21]

2.3 Особливості вибору та роботи з базами даних

Для користувацького досвіду неймовірно важлива швидкодія застосунку. Однією із запорук цього є правильна реалізація рівня роботи з даними. Тому вибір оптимальної бази даних чи фреймворку є запорукою ефективної роботи додатку.

Особливості мобільної платформи окрім іншого накладають певні обмеження на збереження даних. При виборі бази даних, необхідно, в першу чергу, відповісти на два питання: який об’єм даних ми плануємо зберігати та для яких операційних систем буде випущено наш додаток.

2.2.1 SQLite

Подібно до решти популярних платформ, iOS дозволяє працювати з СКБД SQLite. Дана реляційна база даних, є легким (lightweight) та потужним інструментом, що дозволяє швидко створювати сутності та зв’язки між ними [22] Завдяки тому, що ця база даних вбудована в iOS, це дозволяє нам позбутись накладних витрат при її додаванні до проєкту. Також SQLite існує на ринку 22 роки

– першу версію СКБД було випущено 1 серпня 2000 року. Це означає, що за ці роки навколо даної технології вже утворилась спільнота, що полегшує пошук необхідних навчальних матеріалів та полегшує зневадження застосунку. Однак, попри свою довгу історію, вона залишається актуальною донині. [32] На сьогоднішній день, це є найбільш популярна база даних [23].

До переваг SQLite можна віднести:

- вбудованість в iOS,
- cross-platform,
- open-source,
- ефективна робота з даними, що мають складну структуру,
- потокобезпечність,
- низький поріг входження,
- повністю безкоштовна.

На жаль, SQLite не позбавлена недоліків. Серед них:

- відсутність оптимізації для роботи з великими об’ємами інформації,
- розробник відповідальний за життєвий цикл зав’язків між сутностями,
- відсутність підтримки одночасної роботи декількох користувачів (file locking design),
- відсутність вбудованого шифрування,
- нижча швидкість роботи порівняно з Realm і Core Data,
- рішення від сторонньої компанії. [24]

SQLite, у першу чергу, хороший вибір для додатків, що не потребують збереження великої кількості даних (not content-rich). Також дана база даних, гарне рішення для розробників, що вже мали попередній досвід з реляційними базами даних, оскільки вони не потребуватимуть додаткового часу для опанування технології.

2.2.2 Realm

На відміну від попередньої СКБД, Realm не основана на ORM (Object-Relation Mapping). Замість цього вона використовує власний движок, що вирізняється швидкістю та простотою. Серед інших баз даних, Realm в першу чергу виділяє гарна підтримка крос-платформи.

На відміну від Core Data, поріг входження для Realm низький. Це стосується як збереження даних (Realm самостійно опікується збереженням змін) та і створення нових сутностей. Оскільки дана СКБД створена сторонніми розробниками, після додання об'єм програми збільшиться приблизно на 13 Мб.

Переваги:

- висока швидкість [27],
- cross-platform,
- open-source,
- низький поріг входження,
- високий степінь вбудованого захисту даних.
- підтримка вбудованих типів даних Swift,
- ефективне використання пам'яті,
- підтримка інтеграції сторонніх сервісів, таких як AWS тощо[26][34]

Недоліки:

- рішення від сторонньої компанії,
- потребує додаткового дискового простору.

Таким чином, якщо застосунок планується для різних платформ, найкращим рішенням буде використання Realm. Крім того простота та висока швидкість роботи з даними, робить Realm хорошим вибором для додатків, що не потребують переваг Core Data.

2.2.3 Firebase

Firebase – мобільна backend-as-a-service. Це керована подіями NoSQL база даних розроблена компанією Google, що зберігає та синхронізує дані в реальному часі. Вона містить 3 основні сервіси: база даних в реальному часі, автентифікація користувача та хостинг. Хоча Firebase і підтримує роботу в оф лайн режимі, основний її профіль – робота онлайн. Збереження даних на пристрій слугує для уникнення ситуацій, коли інтернет з'єднання зникло і користувач більше немає доступу до контенту. Однак, це не робить Firebase гарною оф лайн базою даних. Це випливає з її технічних характеристик – Firebase хмарний сервіс.

Для кращого розуміння переваг і недоліків даної технології, необхідно проаналізувати відмінність SQL баз даних від NoSQL. Нереляційні бази даних використовують динамічні схеми, на відміну від реляційних. Це дозволяє обробляти великі за розміром або неструктуровані (такі як JSON та документи) дані більш ефективно, але в той же час погіршує роботу з сутностями, що мають складну структуру. SQL краще підходить для роботи з багаторядковими транзакціями. Також на відміну від реляційних баз даних, яка є вертикально масштабованою, нереляційні - масштабуються горизонтально.

Переваги:

- підтримка декількох користувачів одночасно,
- підтримка роботи як з інтернет з'єднанням, так і без,
- ефективна робота з великим об'ємом інформації,
- дозволяє синхронізувати дані між різними платформами,
- наявна підтримка зберігання даних як на пристрої, так і на хмарі,
- ефективна робота з неструктурованими даними,
- обширна і гарно написана документація

Недоліки:

- повний доступ до усіх функцій коштує $24.99 на місяць,

- неефективна робота з сутностями, що мають складну структуру та зв'язки,
- сильна прив'язка до Google сервісів. [33]

Завдяки технічним особливостями Firebase, закладених при її створені, вона є ідеальним рішенням для роботи з великими об'ємами інформації, що потребують хмарного зберігання. Швидка синхронізація між пристроями, навіть з різними операційними системами, дозволяє ефективно використовувати Firebase для кросплатформених застосунків. На відміну від Realm, основний профіль роботи даної бази даних – зберігання та синхронізація даних, що знаходяться на хмарі.

2.2.4 Core Data

Найпопулярнішим вибором для роботи з даними у мобільному застосунку на iOS є фреймворк Core Data, розроблений компанією Apple. Він представляє собою надбудову над базою даних. Тип бази даних чи особливості її реалізації нас в цьому випадку не турбують, оскільки при роботі з фреймворком ми абстрагуємось від бази даних – вона служить лише для збереження даних і розробник напряму з нею не взаємодіє. Вся робота відбувається через Core Data, що представляє дані у вигляді object graph (графа об'єктів), де кожен зв'язок між сутностями представлений ребром графа. Core Data бере на себе відповідальність за керування життєвого циклу об'єктів у графі. За допомогою середовища розробки Xcode, ми можемо переглянути граф об'єктів нашого додатку.

Попри високий поріг входження, Core Data завдяки сильній оптимізації (кешування, використання lazy (лінивого) завантаження даних та продумане управління пам'яттю) дозволяє отримати великий виграш у швидкодії, при цьому позбавляючи розробника необхідності оптимізувати SQL-запити тощо. Також до переваг фреймворку можна віднести лаконічність і структурованість коду, порівняно з використанням SQLite без жодних допоміжних фреймворків чи

бібліотек. [26] Серед цікавих особливостей Core Data є можливість відміни останньої дії. [27]

Переваги:

- швидкість і оптимізація,
- підтримка Apple,
- висока степінь вбудованого захисту даних,
- життєвий цикл об’єктів в графі – відповідальність Core Data,
- можливість візуалізувати об’єктний граф,
- проста міграція [28]

Недоліки:

- високий поріг входження,
- типи даних, що використовуються в Core Data, відрізняються від типів Swift, що потребує додаткової уваги при розробці додатку

Попри усі недоліки, Core Data протягом тривалого часу лишається «золотим стандартом» роботи з даними на iOS. Він поєднує в собі переваги використання високорівневих обгорток для баз даних з швидкодією нативних бібліотек.

Оскільки, мій додаток передбачає поступове накопичення великої кількості даних, мною було обрано Core Data. Тому я б хотіла детальніше розглянути фреймворк Core Data та особливості роботи з ним.

При роботі з Core Data ми маємо справу з 4 компонентами: Persistent container, Managed object model, Managed object context та Store coordinator. Взаємозв’язок між проілюстровано на рисунку нижче.

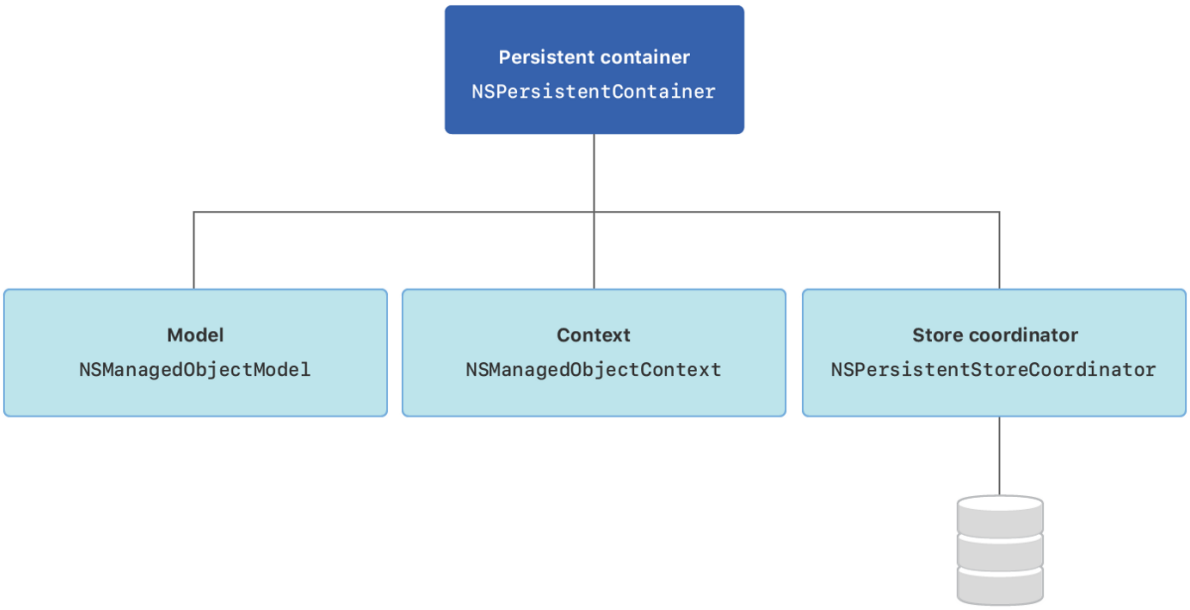

Рисунок 2.2.4.1 Структура модулів фреймворку Core Data

Persistent container являє собою найвищий рівень абстракції, що енкапсулює в собі найважливіші компоненти фреймворку. На наступному рівні знаходиться Managed Object Model, завдяки якому фреймворк мапить записи (records) з persistent store в managed objects. Саме з останніми ми маємо справу при розробці застосунку. Таким чином, Managed Object Model є представленням усіх сутностей нашого додатку вкупі з їхніми зв'язками та полями. По-суті, це є програмною реалізацією .xcdatamodeld файлу, що дозволяє нам редагувати та видаляти наші дані.

Managed Object Context є тою вхідною точкою, з якою взаємодіє бізнес-логіка додатку. Він зберігає посилання на persistent store і дозволяє нам доступитись до нього кожен раз коли нам необхідно створити, редагувати чи видалити сутність. Persistent Store Coordinator – це місток між компонентами фреймворку та базою даних. Він виконує функцію серіалізації та десеріалізації даних для бази даних. Таким чином, «під капотом» усі дані можуть зберігатись в реляційній базі даних, але «наверх» ми отримаємо об'єктний граф.

На жаль, не існує ідеальної універсальної бази даних, яка б могла задовільнити усі потреби. Кожна СКБД має свої переваги і недоліки. Тому вибір

бази даних необхідно починати з написання переліку вимог. Так, для зберігання невеликої кількості даних гарним вибором буде SQLite. Як і всі реляційні бази даних, вона ефективно працює з даними, що мають складну структуру. Для кросплатформних застосунків, що зберігають дані у внутрішнє сховище пристроя, хорошим рішенням буде обрати Realm. Порівняно з рештою мобільних баз даних, вона є найшвидшою. Однак, оскільки вона не інтегрована до iOS, на відміну від SQLite, її додання до проєкту потребує додаткового дискового простору. Ще одним рішенням для кросплатформених застосунків є Firebase. Також вона підходить для додатків, що потребують збереження даних на хмарне сховище. Завдяки тому, що це NoSQL база даних, вона ефективно справляється з обробкою неструктурованих даних великого розміру. Core Data – найпопулярніше рішення для додатків, що потребують оф лайн збереження великих об'ємів даних. Оскільки це лише фреймворк, розробник може обирати базу даних залежно від характеристик майбутнього застосунку та своїх потреб. Робота ж з самим фреймворком залишається такою ж: кодування та декодування даних в сховище – відповідальність Core Data. Ми працюємо з графом об'єктів, що зберігає зв'язки у вигляді ребер графу. Завдяки вбудованому в Xcode інструменту, ми можемо переглядати object graph для нашого додатку. Серед недоліків Core Data – це найскладніша технологія серед усіх вищеописаних.

2.4 Особливості вибору архітектури

Архітектура є надзвичайно важливим елементом при розробці застосунку на будь-якій платформі. Вибір доречного архітектурного патерну та правильна його реалізація, дозволить мінімізувати проблеми при подальшій розробці та підтримці додатку. Крім того, модульність і структурованість проєкта дозволяє зберігати код читабельним та полегшує зневадження. Відділення бізнес-логіки від решти

компонентів покращує написання unit-тестування, що робить додаток більш стабільним та надійним.

2.4.1 MVC

До недавнього часу, стандартом у виборі архітектури для більшості мобільних додатків була архітектура MVC (Model View Controller). При такому підході, проєкт містить 3 модулі: Model, View та Controller. Рівень Model відповідає за рівень роботи з даними (data access layer), рівень View – за відображення, Controller – посередник між Model та View рівнями, який дозволяє змінювати Model, у відповідь на дії користувача та оновлення View при зміні Model. Очікується, що Model та View рівні не знають одне про одного і вся взаємодія відбувається через рівень Controller-а, однак на практиці ми отримуємо сильну зв'язність (coupling) між рівнем Controller та View. [35]

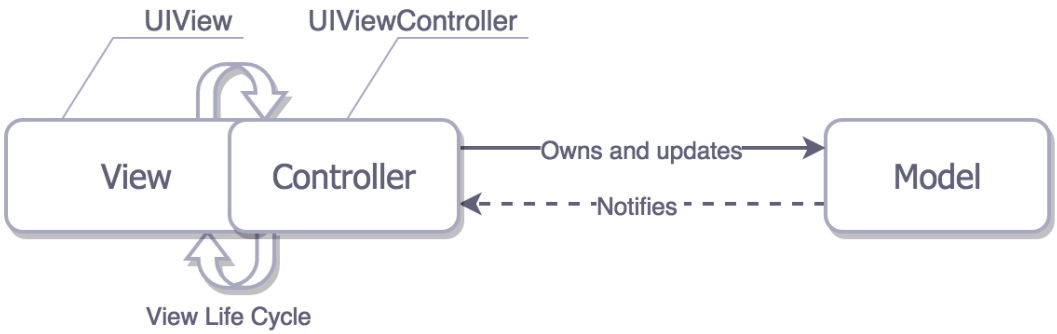

Рисунок 2.4.1.1 Реальна реалізація патерну MVC [3]

Однак з часом все більш явними ставали недоліки даного підходу – «роздування» рівня controller, що позбавляє код структурованості та ускладнює подальшу підтримку застосунку. [34] Щоб позбутись цієї проблеми було розроблено низку нових архітектурних патернів, серед яких MVP, MVVM, MVVM+R, Viper, Redux тощо.

2.4.2 MVP

Щоб подолати проблему надмірної зв'язності (coupling) рівнів Controller та View, було створено патерн MVP (View Model Presenter). Однак, при цьому існує прямий зв'язок (binding) між View та Model та Presenter виконує частину роботи по обробці дій користувача, та зміни View, що є поганим прикладом розділення сфери відповідальності. [35]

2.4.3 MVVM

Останній і найкращий представник із сімейства MV(X) архітектурних патернів – MVVM (Model View ViewModel). Він подібний до MVP, однак не містить прямого зв'язку між View та Model.

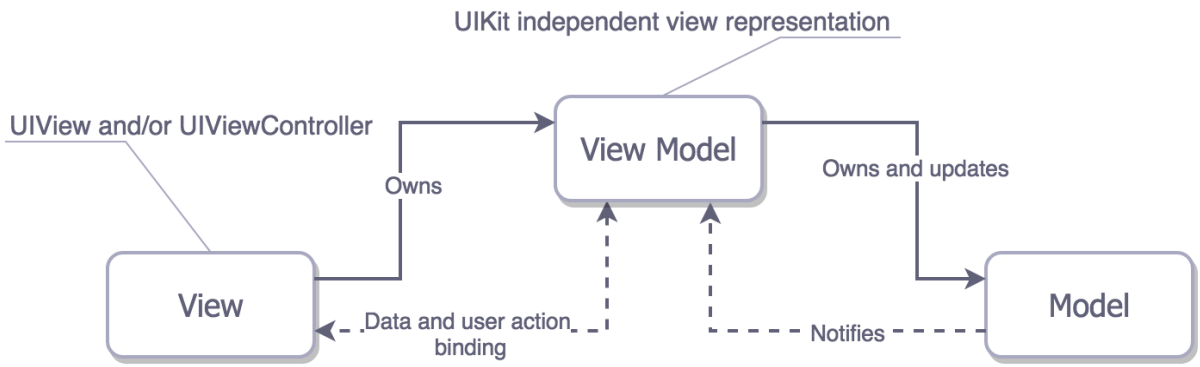

Рисунок 2.4.3.1 Архітектура MVVM [3]

Незалежно від вибору архітектури MVC чи MVVM, ми можемо додати додатковий компонент Router. Завдяки ньому ми можемо інкапсулювати логіку зникання, появи та змінення екранів, оскільки це може бути частиною бізнес-логіки. Назва такого патерну MVC+R та MVVM+R відповідно.

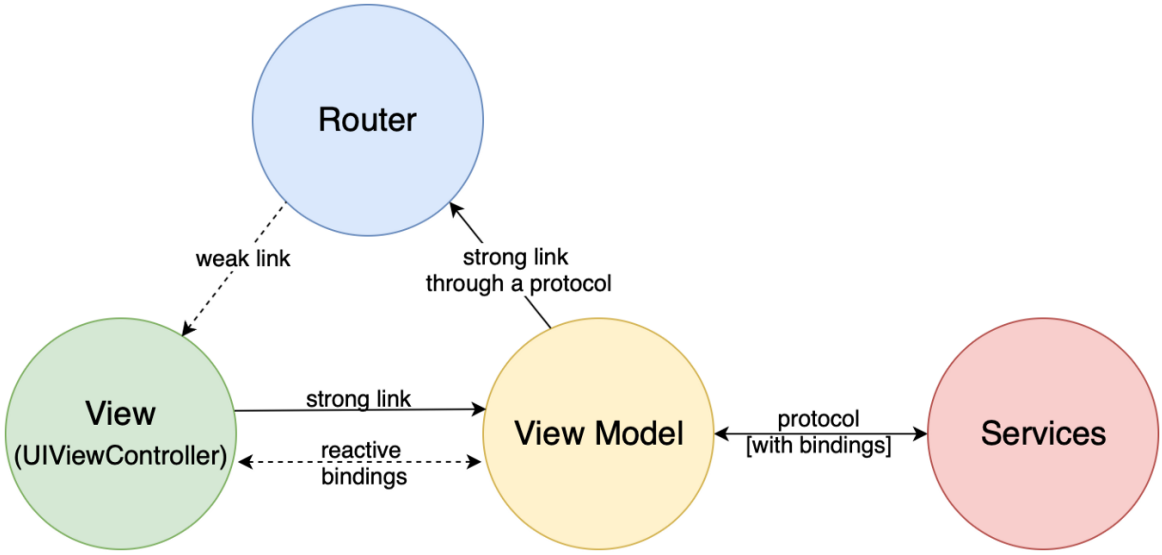

Рисунок 2.4.3.2 Взаємозв'язок компонентів архітектури MVVM+R

2.4.3 Viper, RIB, Redux

Viper архітектура розроблена спеціально для додатків під iOS. Вона являє собою розширення архітектури MVVM+R. На відміну від свого попередника, у архітектурі Viper рівень ViewModel розділено на Interactor та Presenter. До сфери відповідальності першого входить взаємодія з сутностями (domain layer), до другого – підготовка компонентів до відображення на рівні View.

Окрім вищеописаних, існують ще інші архітектури, такі як Redux і RIB. Перша прийшла в мобільну розробку із веб і створена для реактивного програмування. RIB – варіація MVVM+R, у якому компонент ViewModel перейменовано у Interactor та додано додатковий компонент Builder, що призначений для ініціалізації та зв'язування усіх компонентів. [33]

2.5 Особливості розробки графічного інтерфейсу

2.5.1 UIKit

Існує два фреймворки для побудови графічного інтерфейсу мовою Swift – UIKit та SwiftUI. На сьогоднішній день, UIKit – це основний фреймворк для розробки додатків під платформи iOS, iPadOS та tvOS. Основою для його створення слугував фреймворк Cocoa Touch, написаний мовою Objective-C. Оскільки, він є нащадком фреймворку Cocoa, що призначений для macOS, деякі компоненти графічного інтерфейсу є подібними.

При роботі з графічними елементами UIKit, розробник послуговується 3 видами сутностей: UIView, UIWindow та CALayer. Зоною відповідальності UIView є прямокутна область екрану, вмістом якої він керує. До зони відповідальності UIView входить: відмалювання вмісту, його анімація, розміщення вкладених view. Також він обробляє дотики та жести користувача. UIView є базовим класом для усіх графічних елементів фреймворку та визначає їхню основну поведінку. UIWindow - вікно програми. Також є нащадком UIView. Варто зазначити, що розробник напряму не керує відмалюванням інтерфейсу, оскільки за це відповідає UIKit. Робота з фреймворком відбувається декларативно. Таким чином система сама приймає рішення про рендеринг та розміщення елементів. [45]

Кожна UIView має поле layer типу CALayer, що призначена для ефективного рендерингу його вмісту. CALayer – компонент фреймворку Core Animation. У задачі CALayer входить зберігання кешу, для оптимізації рендерингу своєї view та застосування просунутих графічних ефектів, таких як: тінь, градієнт, додання бортику тощо. Причина розділення графічних елементів на UIView та CALayer полягає в тому, що відмалювання контенту не залежить від розміщення об’єкту і навпаки. Кожна UIView має кореневий CALayer, що може містити декілька підшарів. [46] [47]

UIKit надає різноманітні інструменти для побудови UI, залежно від вподобань програміста та рівня його знань. Для початківців є простий механізм drag-and-drop («потягни і відпусти») для додання нових елементів та візуальний інструмент для налаштування їхнього положення та параметрів у вкладці Interface Builder. У цей же час досвідчені програмісти можуть обирати між інструментом Interface Builder та створенням UI за допомогою коду. Однак, це призводить до деяких проблем. Хоча створення нових елементів графічного інтерфейсу відбувається тривіально, правильне їхнє налаштування потребує певних знань та досвіду.

Попри те, що UIKit існує на ринку майже 10 років, він не є застарілим: спільнота продовжує активно підтримувати фреймворк, незважаючи на нового конкурента. Це призводить до двох позитивних наслідків: велика кількість навчальних матеріалів та широкий вибір бібліотек.

Важливим елементом розробки є зневаджування. Серед переваг UIKit – наявність інструмента для зневаджування - Debug View Hierarchy, що дозволяє переглядати візуальні шари програми у реальному часі.

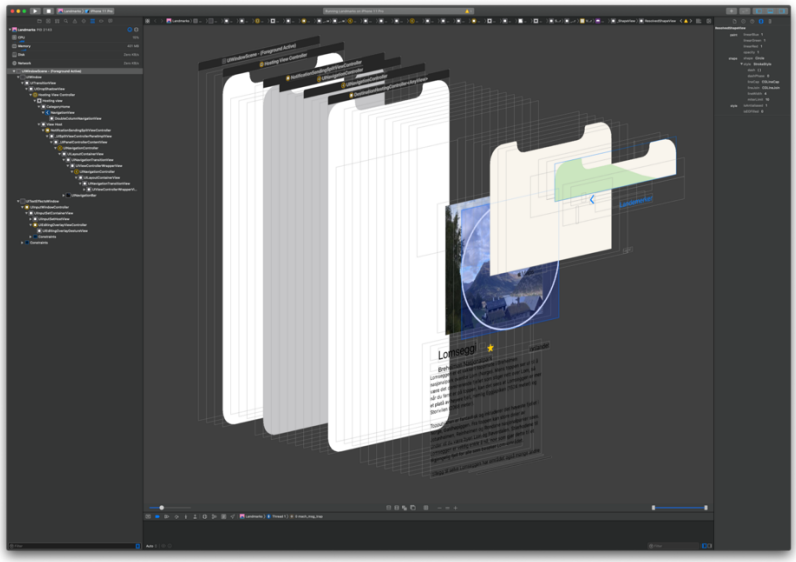

Рисунок 2.5.1.1 Приклад Debug View Hierarchy [42]

Таким чином, ми можемо виділити наступні переваги:

- більша стабільність – фреймворк існує вже 9 років [38];
- легший в освоєнні, порівняно з SwiftUI;
- варіативність: UI може бути створено за допомогою Interface Builder, кодом та змішано;
- більша підтримка: велика кількість бібліотек, як розроблених Apple, так і сторонніх.

Недоліки:

- складніше створення UI кодом порівняно з SwiftUI;
- старий фреймворк – в майбутньому Apple може повністю змістити фокус в сторону SwiftUI і UIKit позбудеться активної підтримки. [39]

2.5.2 SwiftUI

SwiftUI являє собою нову альтернативу UIKit. Завдяки декларативному підходу, фреймворк дозволив якісно переглянути підхід до створення графічного інтерфейсу. Тепер на заміну громіздкому Interface Builder-у прийшов інструмент Live Preview, що дозволяє в реальному часі переглядати виконання програми прямо в середовищі розробки Xcode, без потреби запускати окремо емулятор. Ця функція особливо актуальна для старих комп’ютерів, що не мають великої кількості оперативної пам’яті. Окрім цього, тепер увесь UI створюється за допомогою коду.

Разом з анонсом SwiftUI було представлено фреймворк Combine. Даний фреймворк надає декларативний Swift API для обробки асинхронних подій за допомогою патерну Observer («Слухач»). Таким чином, при зміні даних, графічний інтерфейс буде перемальовано автоматично.[43] Саме поєднання даних двох фреймворків дозволяє збільшити структурованість та модульність коду. [44]

З випуском iOS 14, Apple додали підтримку віджетів. Для цього було розроблено фреймворк WidgetKit, який доступний лише для SwiftUI. Таким чином, ми можемо очікувати, що з часом буде з’являтись все більше і більше інструментів

та технологій, що будуть використовувати лише новий фреймворк. Також при створені додатків для WatchOS (операційної системи для годинників), розробники на SwiftUI можуть використовувати переваги нативних бібліотек та графічних елементів. [42]

Переваги:

- декларативність
- підтримка «змішаних» проєктів: компоненти написані на UIKit можуть бути легко інтегрованими у проєкт на SwiftUI та навпаки
- підтримка реактивного програмування
- наявність функції Live Preview у Xcode, що надає можливість швидкого перегляду виконання програми
- SwiftUI автоматично опікується несуперечністю розміщення компонентів [37]
- наявність єдиного «джерела правди» для графічного інтерфейсу: весь UI задається за допомогою коду
- фреймворк Combine
- автоматична підтримка темної теми

Недоліки:

- нестабільність
- відсутність підтримки iOS нижче 13
- обмежене покриття API [40]

Випускаючи фреймворк SwiftUI Apple провели «роботу над помилками» та постарались виправити наявні проблеми і надати концептуально новий спосіб побудови UI. Це дозволило зменшити кількість boilerplate коду (код, що багато разів повторюється без змін) та пришвидшити розробку. Попри те що, більшість експертів сходять в думці, що за SwiftUI майбутнє [40], станом на 2022 рік

обмеженість функціоналу та нестабільність роблять цей фреймворк поганим вибором для додатків enterprise рівня.

Підсумовуючи, UIKit – надійний фреймворк, з активною підтримкою як від спільноти, так і від Apple. При використанні UIKit розробник може обирати між створенням графічного інтерфейсу за допомогою коду чи візуальних інструментів. Однак, з цього випливає і важливий недолік – немає єдиного «джерела правди». При роботі у змішаному стилі, особливо з великими проєктами, дуже легко втратити загальну картину взаємодії та розміщення елементів. Для того щоб подолати не лише цей недолік, було розроблено фреймворк SwiftUI. Він привніс дві серйозні зміни в процес розробки: відтепер увесь графічний інтерфейс створюється за допомогою коду та реактивно (при умові використання фреймворку Combine). Також порівняно з UIKit, новий фреймворк надає набагато більшу степінь декларативності. Завдяки цьому, Apple досягли того, що одні і ті ж речі при використанні SwiftUI потребують набагато меншої кількості коду.

На сьогоднішній день, більшість додатків для платформи iOS написані з використанням фреймворку UIKit. Дана динаміка буде зберігатись ще декілька років, поки не відбудеться плавний перехід в сторону SwiftUI. На жаль, він ще не є досить стабільним для розробки застосунків enterprise рівня. Ще одним серйозним мінусом є обмежене покриття API, порівняно з UIKit.

Щоб зробити перехід на SwiftUI більш плавним та органічним, Apple додала підтримку змішаних проєктів, написаних з використанням обох фреймворків. Це досягається завдяки наступному механізму: елемент SwiftUI обгортається, за допомогою UIHostingController, так, що решта елементів взаємодіє з ним як з «рідним» і vice versa. Таким чином, у вже існуючі додатки може бути легко інтегровано компоненти написані з використанням SwiftUI. Завдяки цьому, застосунки написані на UIKit можуть використовувати переваги нового фреймворку, без необхідності бути повністю переписаними з його використанням.

Для розробки практичної частини курсової було обрано фреймворк UIKit.

2.6 Висновки до розділу 2

Таким чином, при створені додатку, необхідно серйозно підійти до дослідження особливостей кожної з підтримуваних платформ. При розробці для мобільних пристроїв, варто ощадливо ставитись до використання ресурсів. Це, в першу чергу, стосується роботи в фоні. Використання надмірної кількості пам'яті та/або процесорного часу неодмінно призведе до аварійного завершення програми Watchdog-ом, який створений для контролю за роботою додатку. Це зроблено з метою забезпечення плавної роботи системи та безпеки. Окрім цього, для перешкоджання несанкціонованих дій, кожен додаток розміщується в у власному контейнері sandbox, що не дозволяє йому доступатись до чужих файлів, налаштувань та системних файлів.

При розробці для комп'ютерів та планшетів варто приділити особливу увагу дизайну, оскільки додаток, що гарно виглядає на iPhone не обов'язково буде так само гарно виглядати на іншій платформі. Крім цього, необхідно врахувати різницю у способах взаємодії з пристроєм та наявність особливих функції (наприклад, Split Screen), що не доступні для інших операційних систем.

Перед фінальним етапом – публікацією додатку – необхідно переконатись, що застосунок як задовольняє мінімальні стандарти якості, так і не протирічить правилам магазину та чинному законодавству країн, які підтримуються вашою програмою.

При виборі інструментів та фреймворків варто враховувати, що не існує універсального рішення і обирати необхідно виходячи з особливостей конкретного проєкту та вимог. Популярними рішеннями є SQLite, Realm, Firebase та Core Data. SQLite поєднує в собі легкість та переваги реляційних баз даних, що робить його хорошим рішенням для додатків, що потребують збереження лише невеликої кількості даних. Найкращим крос-платформним оф лайн сховищем для застосунків на iOS є Realm. Серед його переваг: швидка робота з даними, ефективне використання пам'яті та open-source. Для крос-платформних додатків, що

переважно працюють з Інтернет підключенням існує Firebase. Завдяки тому, що це нереляційна база даних, обробка великого об'єму інформації відбувається дуже швидко. Також, він підтримує роботу декількох користувачів одночасно з синхронізацією між різними платформами. Однак, на відміну від попередніх рішень, він не є безкоштовним. Core Data – це фреймворк, розроблений Apple, що створений для зберігання даних на пам'яті пристроя. Абстрагування від бази даних дозволяє використовувати різні рішення, залежно від потреб проєкту, залишаючи той самий синтаксис роботи з даними. Завдяки високій степені оптимізації, ефективне зберігання та обробка великої кількості даних. Тому, на сьогоднішній день, Apple рекомендує використовувати Core Data для усіх додатків, що розробляються лише для однієї платформи та не потребують збереження даних на хмару.

Вибір архітектури – надзвичайно важливий етап створення будь-якого застосунку. Зараз на зміну популярному в минулому архітектурному патерну MVC прийшов MVVM, що дозволяє позбутись високої зв'язності компонентів Controller та View. Він є стандартом при розробці з використанням фреймворку SwiftUI. Крім того, існує архітектура Viper, розроблена спеціально для iOS. Її відмінність від попереднього патерну в тому, що рівень ViewModel розділено на Interacter та Presenter, для того щоб відмежувати логіку відображення екранів від взаємодії з сутностями. Варто враховувати, що інколи для досягнення найкращого результату, необхідно вносити зміни до вже існуючих рішень, тим самим адаптуючи їх до конкретних потреб та вимог. Так компанія Uber, при розробці однойменного додатку додала до архітектурного патерну MVVM+R компонент Builder для ініціалізації та зв'язування усіх компонентів.

Не менш важливим є вибір фреймворку для розробки графічного інтерфейсу. Так, розробник може обирати між UIKit та SwiftUI. Кожен із фреймворків демонструє власний підхід до створення додатків. Однак, у них є і спільні риси: декларативність, активна підтримка Apple та підтримка інтеграції елементів іншого фреймворку. На жаль, SwiftUI досі не є достатньо надійним та містить багато багів,

що ускладнює його використання для великих додатків. Також, через те що це більш молодий фреймворк, кількість бібліотек та матеріалів набагато скромніша, порівняно з UIKit.

Розділ 3. Опис реалізації програмного продукту

3.1 Постановка технічного завдання.

Технічним завданням даної курсової роботи є створення мобільного додатку для ведення обліку фінансів на базі операційної системи iOS. Застосунок повинен забезпечити зручне внесення витрат та надходжень, з можливістю їхнього подальшого перегляду у вигляді історії та статистики.

При створені додатку необхідно враховувати особливості життєвого циклу мобільних застосунків та коректно обробляти зміну станів. Також накладається вимога на плавну роботу застосунку, оскільки у зворотному випадку, це може призвести до аварійного завершення програми, що негативно вплине на користувацький досвід. Створений додаток має коректно відображатись на усіх пристроях, з операційною системою iOS, незалежно від роздільної здатності та розміру екрану. З метою забезпечення ефективного використання пам’яті та швидкодії, потрібно враховувати особливості моделі управління пам’яттю мови Swift та не допустити memory leak.

Додаток повинен бути розробленим з використанням оптимальних інструментів та технологій. Не допускається використання застарілих бібліотек та фреймворків, оскільки це ускладнить подальшу підтримку додатку. На основі виконаного в попередньому розділі аналізу, у якості сховища було обрано фреймворк Core Data. Для створення графічного інтерфейсу має використовуватись фреймворк UIKit.

3.2 Особливості реалізації та використаних технологій

3.2.1 Обґрунтування вибору архітектури та парадигм

Після аналізу технічного завдання стало очевидно, що додаток буде містити велику кількість вікон, а отже і велику кількість ViewController-ів. Щоб уникнути «роздування» ViewController та зменшення зв'язності компонентів, у якості архітектури було обрано MVVM.

MVVM - це оптимальний вибір, що поєднує в собі простоту сімейства архітектурних патернів MV(X) та ефективність складніших архітектур, таких як Viper, Redux та RIB. Бізнес логіка додатка не потребує зникання/появи екрану відповідно до зміни внутрішнього автомату станів, тому було вирішено відмовитись від додавання компоненту Router до архітектури.

Як було вказано в попередніх розділах, мова Swift підтримує мультипарадигменність. Однак, Apple рекомендує віддавати перевагу більш новій парадигмі ПОП (Протокольно-орієнтоване програмування) замість ООП (Об'єктно-орієнтоване програмування). Замість розширення функціональності класу через наслідування, як це прийнято в ООП, ми можемо розширювати (створити extension) існуючу сутність. З точки зору архітектури додатку, це є більш безпечним рішенням, оскільки ми можемо позбутись складних та нашарованих абстракцій. З точки зору семантики, використання розширення позбавляє нас проблеми приналежності дочірнього класу до батьківського.

Створювати розширення можна не лише для класів, а і для структур (struct) та переліків (enum). Оскільки, розширювати можна не лише власні, а і «чужі» сутності, накладається вимога, що розширення не може впливати на розмір сутності (не може містити збережуваних властивостей). При тому розширення має доступ до усіх властивостей та методів свого типу.

У даній курсовій extension були використанні для розширення функціональності вбудованих типів Swift та UIKit. Наприклад, для полегшення

фільтрування транзакцій за часом було додано розширення до типу Date, що дозволяє знайти перший та останній день в місяці.

```
extension Date {
    func startOfMonth() -> Date {
        return Calendar.current.date(from: Calendar.current.dateComponents([.year, .month], from:
            Calendar.current.startOfDay(for: self)))!
    }

    func endOfMonth() -> Date {
        return Calendar.current.date(byAdding: DateComponents(month: 1, day: -1), to:
            self.startOfMonth())!
    }
}
```

Рисунок 3.2.1.1 Приклад розширення вбудованого типу

Окрім цього, ПОП надає нам протоколи (protocol). За допомогою них ми можемо задекларувати вимоги, які ми накладаємо на тип, що буде відповідати (conform) даному протоколу. Він може вимагати наявність властивостей, методів та ініціалізаторів. Зважаючи на те, що ми лише декларуємо вимоги, протокол не може містити реалізації в своєму тілі. Якщо існує потреба відділити реалізацію від типу, що буде відповідати протоколу, можна створити реалізацію за замовченням за допомогою розширення.

При розробці ViewController-ів виникає потреба використовувати велику кількість протоколів. Наприклад, якщо View містить список елементів (UITableView), нам необхідно реалізувати протокол UITableViewDataSource, що відповідає за надання view джерела даних та UITableViewDelegate, що використовується для керуванням вибору, видалення та зміни порядку елементів. [47]

```
extension SettingsViewController:UITableViewDataSource{
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return budgets.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "BudgetListItem", for: indexPath)
            as! BudgetTableViewCell
        cell.configure(for: budgets[indexPath.row])
        return cell
    }
}
```

Рисунок 3.2.1.2 Приклад розширення власного класу для задовільнення протоколу

3.2.2 Опис практичної реалізації обраних патернів

Делегування – популярний патерн при розробці застосунків мовою Swift. Для того, щоб гарантувати наявності певної функціональності у делегата, ми інкапсулюємо повноваження делегата, за допомогою протокола. [48]

Протоколи та патерн делегування допомагають налаштувати взаємодію між об'єктами. Завдяки використанню патерну делегування, «дочірній» об'єкт не потребує знати конкретний тип «батьківського» об'єкту. Прибравши прямий зв'язок між елементами, ми можемо зменшити зв'язність об'єктів. Також це підвищує можливість повторного використання коду.

Одним із екранів мого додатку є екран налаштування, який містить список усіх бюджетів, які створив користувач. При натиску на комірку списку, відбувається перехід на екран для перегляду інформації про обраний бюджет. Також користувач може редагувати та видалити бюджет. Для того, щоб повідомити попередній екран про зміни, ми створюємо протокол, який він має конформити. У ViewController-і екрану деталей ми додаємо делегати. Завдяки тому, що протоколи можна використовувати замість типу, ми можемо абстрагуватись від конкретного ViewController-а.

```
protocol BudgetDetailsViewControllerDelegate: AnyObject {
    func finishedEditing(budget:Budget)
    func deleteBudget(budget:Budget)
}
```

Рисунок 3.2.2.1 Приклад створення протоколу для налагодження взаємодії між екранами

Ще одним популярним патерном, що використовується у Swift є target-action (з англ. «ціль-дія»). Він дозволяє викликати певну дію на певному об'єкті.

У цій роботі патерн target-action було використано при створенні випадаючого списку (drop-down menu). Оскільки, це функція, що прийшла в мову Swift із Objective-C, функцію, що викликається в селекторі (selector), необхідно позначати як @objc, щоб зробити її доступною для рантайму Objective-C. При використання синтаксису #selector, параметри перевіряються на етапі компіляції.

```
let button = UIBarButtonItem(title: "Done", style: .plain, target: self, action:
    #selector(self.action))
```

Рисунок 3.2.2.2 Приклад використання патерну target-action

3.2.3 Особливості використання фреймворку Core Data

При роботі з Core Data, перша річ з якою стикаєшся це кодогенерація (codegen). Після створення сутностей у вікні редактором, наступним кроком є генерація їхніх класів. Починаючи з Xcode 8, розробник має 3 варіанти: Manual/None, Class Definition та Category/Extension. Як випливає із назви, при Manual/None редактор не генерує файли, а покладає це на плечі розробника. Дана опція створена для використання разом з системою контролю версій. Недоліком є те, що усі зміни внесені через редактор мають бути вручну продубльовані у відповідні класи сутностей. Class Definition створює класи при першому запуску програми. Вона гарно підходить для задач, коли не потрібно відслідковувати зміни та можна доручити синхронізацію та підтримування цілісності даних системі. Останній варіант Category/Extension при запуску додатку генерує розширення класу сутності. Ця опція створена для відслідковування ієрархії класів. [49]

Оскільки, при розробці важливу роль приділялось використанню системи контролю версій, у якості варіанту кодогенерації було обрано Manual/None.

При ітеративній розробці неймовірно важко передбачити майбутні зміни. Внесення правок в бізнес-логіку, часто призводить до зміни моделі даних. Для полегшення внесення правок, Core Data надає механізм легких міграцій (lightweight migrations). Для проведення міграції необхідно мати оригінальну модель даних та змінену. Вони потрібні для створення трансформаційної моделі, що призначена для конвертації даних зі старого сховища в нове. Міграція відбувається в 3 етапи. Спочатку копіюються усі об’єкти. Далі вони трансформуються, відповідно до заданих правил. На заключному етапі, відбувається валідація даних. Окрім легкої міграції, існує також ручна (manual). Вона надає розробнику більший контроль над

процесом перетворення об’єктів, але потребує написання більшої кількості коду. [50] При розробці даного застосунку, переважно використовувались легкі міграції.

Усі сутності, що створюються в межах Core Data, є нащадками NSManagedObject. Тому при створені запитів, у якості результату ми отримує об’єкти типу NSManagedObject, які далі можемо приводити до бажаного типу. Завдяки цьому, ми можемо створювати універсальну реалізацію запиту на відвантажування даних, абстрагуючись, в межах функції, від конкретного типу.

```
func fetchAll(entityName: String) throws -> [NSManagedObject]{
    var result = [NSManagedObject]()
    do {
        let request = NSFetchRequest<NSFetchRequestResult>(entityName: entityName)
        let records = try context.fetch(request)
        if let records = records as? [NSManagedObject] {
            result = records
        }
    } catch {
        throw error
    }
    return result
}
```

Рисунок 3.2.3.1 Приклад запиту з використанням NSManagedObject

3.2.4 Особливості роботи з нотифікаціями

Важливим елементом даного додатку є push-нотифікації. Вони використовуються для нагадування користувачу про регулярні платежі. При створенні або редагуванні транзакції, користувач може вказати, що вона регулярна та вказати часовий інтервал у днях.

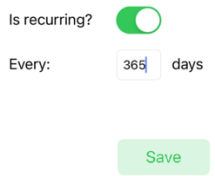

Рисунок 3.2.4.1 Приклад додання повторюваної транзакції

Push-нотифікації – це повідомлення, що бачить користувач, навіть коли активно не взаємодії з додатком. Ці повідомлення надсилаються через Apple Push Notification service (APNs) та можуть містити текст, прикріплені медія, грати звук, додавати бейдж (badge) на іконку застосунку або містити інтерактивні елементи, для взаємодії з користувачем без відкриття додатку. Ще одним видом push-нотифікацій є «тихі» повідомлення, що слугують для виконання завданнь у фоновому режимі. У загальному, push-нотифікації можна поділити на локальні та віддалені. Оскільки, усі нотифікації нашого додатку мають виключно персональний характер, було використано локальні нотифікації. [51]

Після iOS 10, на зміну старому фреймворку UILocalNotification, прийшов новий – UILocalNotification. У ньому переглянуто логіку взаємодії з користувачем. Зокрема, для створення нотифікації, спочатку необхідно отримати його дозвіл. Таким чином, він убезпечується від отримання безлічі непотрібних йому повідомлень. Для запиту на дозвіл використовується метод requestAuthorization(). При тому, необхідно також вказати, які види нотифікацій ми плануємо показувати. Доступно 4 вида: badge, sound, alert та carPlay.

На прикладі нижче зображено приклад запиту на дозвіл демонстрації push-нотифікацій та оновлення чисельного бейджу програми.

```
private func requestAuthorization(completionHandler: @escaping (_ success: Bool) -> ()) {
    // Request Authorization
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .badge]) { (success, error) in
        if let error = error {
            print("Request Authorization Failed (\(error), \(error.localizedDescription))")
        }
        completionHandler(success)
    }
}
```

Рисунок 3.2.4.2 Приклад запиту на дозвіл демонстрації push-нотифікацій

Наступним кроком є створення безпосередньо нотифікації. Окрім, заголовку, підзаголовку, основного тексту та звуку, розробник також може вказати ідентифікаційний код. Це особливо важливо при створенні повторюваної нотифікації. Після цього, необхідно створити трігер. Завдяки ньому User Notifications фреймворк буде знати коли необхідно показувати нотифікацію. У

якості трігеру може інтервал часу (UNTimeIntervalNotificationTrigger), конкретна дата і час (UNCalendarNotificationTrigger) або зміна геопозиції (UNLocationNotificationTrigger). [52]

У подальшому, якщо виникає потреба скасувати нотифікацію, ми можемо скористатить методом removePendingNotificationRequests, вказавши ідентифікаційний код. [53] Для того щоб позбутись необхідності зберігати десь масив з усіма id нотифікації та відповідної транзакції, замість генерації нових id, було викристано ідентифікаційний код транзакції, пов'язану з цим повідомленням.

3.2.5 Особливості роботи з URLSession

Для реалізації мультивалютності додатку, необхідно було використовувати стороннє API для отримання даних про актуальний курс. Для завантаження та вивантаження даних в мережу, мова Swift передбачено клас URLSession.

Розробнику доступно три види завдань:

- URLSessionDataTask – дозволяє працювати з GET запитами;
- URLSessionUploadTask – призначений для завантаження даних в мережу, з використанням POST або PUT запитів;
- URLSessionDownloadTask – створений для завантаження файлів з віддаленого сховища на тимчасовий носій. [54][55]

До недавнього часу, єдиним способом роботи з URLSession був GCD. У 2021 році Apple презентувала Concurrency, який привніс в мову Swift концепт async/await. Головний задум полягає в тому, щоб замість передавання у функцію колбеків (callback) – блоку коду, що виконається після завершення асинхронної задачі – функція позначається async, а доступ до результату відбувається за допомогою ключового слова await. Головна перевага такого підходу полягає в лаконічності коду та зменшенні вкладеності. Відмінність між двома способами роботи з URLSession проілюстровано нижче.

```
func getCurrencyList(completion: @escaping (_ data: [Currency]?, _ error: Error?) -> ()){
    let url = NSURL(string:
        "https://currency-converter5.p.rapidapi.com/currency/list?format=json&language=en")! as URL
    let request = NSMutableURLRequest(url: url, cachePolicy: .useProtocolCachePolicy,
        timeoutInterval: 10.0)
    request.httpMethod = "GET"
    request.allHTTPHeaderFields = headers

    let dataTask = URLSession.shared.dataTask(with: request as URLRequest, completionHandler: {
        (data, response, error) -> Void in
        if (error != nil) {
            completion(nil, error)
            return
        } else {
            guard let data = data else {
                completion(nil, nil)
                return
            }
            do{
                let json = try JSONDecoder().decode([Currency].self, from: data)
                completion(json, nil)
            } catch {
                completion(nil, error)
            }
        }
    })
    dataTask.resume()
}
```

Рисунок 3.2.5.1 Приклад запиту з використанням URLSession та колбеку

```
func getCurrencyListAsync() async throws -> [Currency]{
    let url = NSURL(string:
        "https://currency-converter5.p.rapidapi.com/currency/list?format=json&language=en")! as URL
    let request = NSMutableURLRequest(url: url, cachePolicy: .useProtocolCachePolicy,
        timeoutInterval: 10.0)
    request.httpMethod = "GET"
    request.allHTTPHeaderFields = headers
    let (data, _) = try await URLSession.shared.data(for: request as URLRequest)

    return try JSONDecoder().decode([Currency].self, from: data)
}
```

Рисунок 3.2.5.2 Приклад запиту з використанням URLSession та Concurrency

Подібно до GCD, в контексті Swift Concurrency ми теж маємо справу з задачами (Task), яким можемо призначати пріоритети. Однак, робота з задачами була дещо переосмислена: тепер таска може мати структурну або неструктурну належність. Різниця між ними проявляється при скасування задачі. Відміна структурованої таски призведе до скасування усіх її дочірніх задач, у той час

відміна неструктурованої таски не буде мати ефекту на інші задачі, оскільки вона не має «батька».

Swift Concurrency також дозволяє досягти вищої степені безпеки на етапі виконання та більш ефективного використання потоків. При роботі з GCD, блокування одного потоку призводить до «піднімання» нового, що частот призводить до ситуації, яка називається вибух потоків (thread explosion). Оскільки, створення кожного нового потоку та перемикання контексту призводить до збільшення накладних витрат, у Concurrency відмовились від використання потоків. Замість цього, використовуються континуації (continuation) – абстракція над реальними потоками. Завдяки цьому, блокування однієї задачі не призводить до блокування всього потоку, а лише її абстракції. [56]

3.3 Опис розробки мобільного застосунку та принципи роботи

При першому запуску додатку, користувачу буде запропоновано створити акаунт та перший бюджет. Бізнес логіка накладає обмеження на створення транзакцій, категорій та джерел, що не належать жодному бюджету. При тому, взаємна приналежність між сутностями забезпечується завдяки зв’язкам. Сутність бюджет містить в собі 3 зв’язки один-до-багатьох. Правилом видалення встановлено Cascade – при видалені бюджету, усі її підпорядковані екземпляри теж видаляються.

Attributes

Attribute	Type
S mainCurrency	String
B isCurrent	Boolean
S name	String

Relationships

Relationship	Destination	In...
M categories	Category	budge
M sources	Source	budge
M transactions	Transaction	budge

Рисунок 3.3.1 Поля та зв'язки сутності бюджет

Створюючи бюджет, користувач може вказувати назву та основну валюту. При переході на екран вже існуючого бюджету, відображається статистика по загальній сумі витрат та надходжень за весь час. Також користувач може ознайомитись з кількість категорій, джерел та транзакцій, що належать вказаному бюджету.

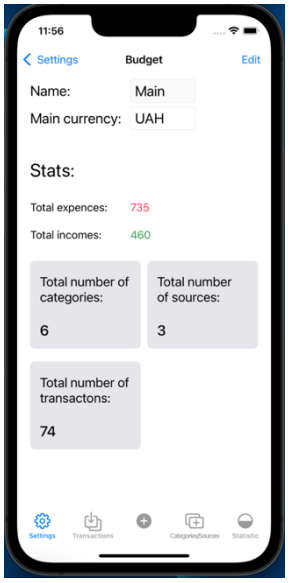

Рисунок 3.3.2 Екран бюджету

Функцією, з якою найчастіше взаємодіє користувач, це додавання транзакцій. При створені транзакції він може обирати вид транзакції (витрата/надходження), категорію для витрат та джерело для надходжень. Аби спростити вибір категорії та джерела, було реалізовано випадаючий список.

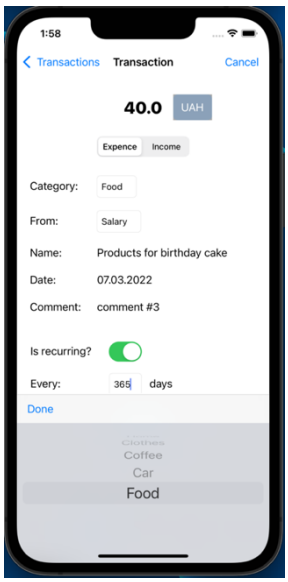

Рисунок 3.3.3 Приклад вибору категорії із випадаючого списку

Також, незалежно від виду транзакції, є можливість вибрати, гроші з якого джерела були витрачені. Завдяки цьому, автоматично перераховується доступна сума грошей відповідного джерела. Окрім цього, для транзакції можна встановлювати ім’я, дату, коментар та інтервал часу, якщо вона повторювана.

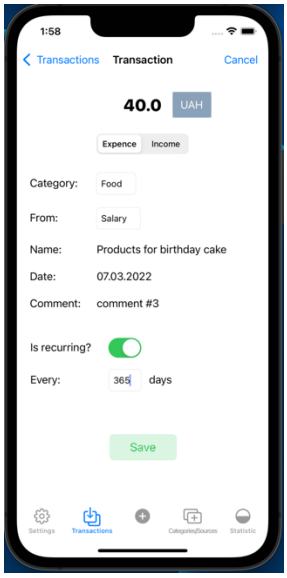

Рисунок 3.3.4 Приклад створення транзакції

Додаток також дозволяє переглядати історію транзакцій. Для покращення користувацького досвіду, є можливість фільтрування транзакцій як за видом, так і за часом. Кожна транзакція в списку, окрім назви, суми та дати, має спеціальну

кольорову мітку, яка відповідає кольору його категорії/джерела. Це спрощує візуальне сприйняття інформації.

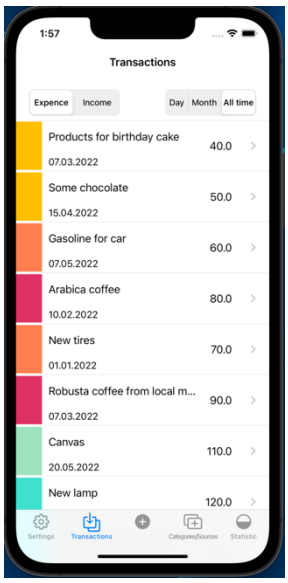

Рисунок 3.3.5 Список транзакцій з застосованими фільтрами

Не залежно від того, чи це категорія чи джерело, користувач може встановлювати назву та колір. На рисунку нижче зображено приклад створеної категорії.

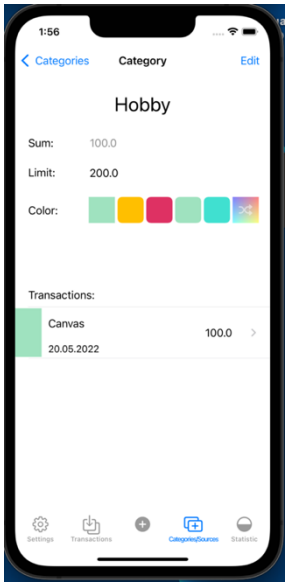

Рисунок 3.3.6 Екран категорії

Для збільшення контролю над фінансами, користувач може також додавати ліміти для категорій та джерел. У загальному списку, відображається сумарна шкала лімітів та сум транзакцій кожного джерела та категорії.

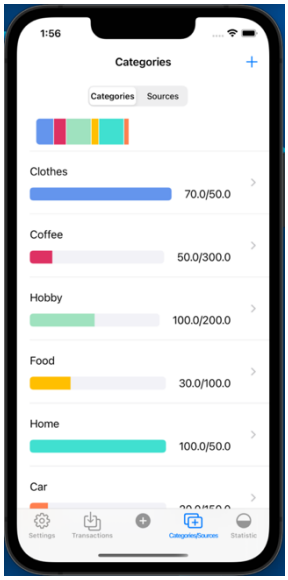

Рисунок 3.3.7 Екран списку категорії

При створенні додатку особлива увага приділялась екрану статистики. Для кращої візуалізації даних, статистика демонструється у вигляді діаграми розподілу витрат за категоріями/джерелами та графіку порівняння загальної суми витрат/надходжень за місяці. Статистику можна переглядати як за витратами, так і за надходженнями.

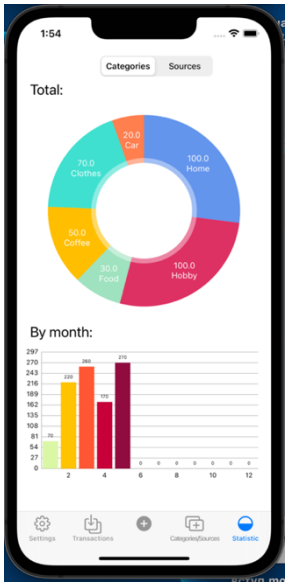

Рисунок 3.3.8 Екран статистики

При аналізі конкурентів, було виявлено, що більшість з них не підтримують мультивалютність. При створенні транзакції, користувач може обрати валюту. Для коректного обрахування загальної суми витрат та статистики, на етапі обрахунку, усі транзакції переводяться в основну валюту бюджету, яка була вказана при його створені. Для того щоб пришвидшити роботу додатку, підвантаження нового курсу відбувається раз на день. Система зберігає курси використовуваних валют. Перерахунок загальної суми у категоріях та джерелах відбувається у 2 випадках: користувач додав/редагував/видалив транзакцію або курс основної та валют транзакцій змінився відносно попереднього дня. Те саме стосується і статистики.

Для полегшення вибору потрібної валюти, наявний механізм пошуку.

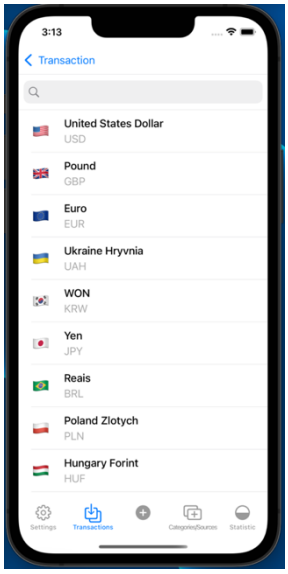

Рисунок 3.3.9 Екран вибору валюти транзакції

3.4 Висновки до розділу 3

Підсумовуючи, у даному розділі було поставлено технічне завдання та вимоги до готового продукту, на основі, отриманих у ході огляду теоретичних відомостей, знань.

Окрім цього на прикладах було розглянуто особливості реалізації проєкту. Серед іншого, було досліджено різницю використання парадигм ООП і ПОП в мові Swift, описано принцип дії та реалізацію двох найпопулярніших патернів при розробці на iOS - делегування та target-action. Також було розкрито нюанси роботи з фреймворком Core Data та нотифікації (UILocalNotification). На прикладі завантаження актуального курсу валют, було розглянуто особливості використання URLSession. При тому було проведено порівняння реалізації завантаження списку усіх валют з застосуванням GCD та Swift Concurrency. Усі вищенаведені пункти супроводжувались ілюстративними матеріалами, у вигляді уривків коду, що були використані при розробці практичної частини даної курсової роботи.

Останнім висвітленим аспектом був опис безпосередньої реалізації додатку та принципів його дії.

Висновки

У ході роботи було досліджено нюанси створення додатків для платформи iOS та виділено особливості, які необхідно було врахувати під час реалізації практичної частини. Завдяки цьому, вдалось уникнути багатьох проблем при проєктуванні та безпосередньому створенню додатку. Велику увагу було приділено методам забезпечення стабільної роботи та безпеки, для досягнення якомога кращого користувацького досвіду.

Ключовим моментом при створенні додатку є вибір архітектури. Саме він визначає наскільки легко буде розробляти та підтримувати застосунок в подальшому. Неправильний вибір може призвести не лише до порушення принципів розробки (SOLID, DRY тощо), а і до неможливості ефективної реалізації певних бізнес вимог, що призведе до низької стабільності та швидкодії програми. Також вибір архітектурного патерну може негативно вплинути на зневаджування додатку. Деякі архітектури не дозволяють повністю відділити бізнес-логіку застосунку від інших компонентів, що спричиняє низький рівень покриття тестами. Для розробки практичної частини використовувалась архітектура MVVM. Ключовою її перевагою є низька зв’язність компонентів. У роботі даний патерн програмування себе гарно зарекомендував. При створенні додатку, що передбачає велику кількість екранів та переходів між ними, важливо не допускати «роздування» ViewController-а або будь-якого іншого компоненту, оскільки це ускладнює впровадження нової функціональності.

Завдяки тому, що мова Swift активно розвивається і підтримується, існує безліч опцій при виборі сховища даних. Оскільки, пріоритетом при розробці була швидкодія, для роботи з даними було обрано фреймворк Core Data. Складність в освоєнні даного фреймворку компенсується продуктивністю кінцевого продукту. Ще одним приємним бонусом є те, що додання Core Data в проєкт не призводить до збільшення розміру файла програми.

Для ефективної реалізації графічного інтерфейсу додатку було детально розглянуто ключові переваги та недоліки фреймворків UIKit та SwiftUI. З огляду на більшу стабільність, було обрано UIKit. Широкий перелік можливостей, що надає UIKit дозволила гнучко налаштовувати UI, відповідно до вимог бізнес-логіки. На мою думку, недоліком даного фреймворку є складність створення власних UIView, особливо порівняно з SwiftUI.

Оскільки було розроблено мобільний додаток для ведення обліку фінансів, що містить інструменти для аналізу та відслідковування грошового потоку, мета даної курсової роботи була успішно досягнута.

Даний проєкт має потенціал для подальшого розвитку. У першу чергу, це публікація застосунку у магазині App Store, що зробить застосунок доступним для скачування звичайним користувачам. Також серед можливих векторів розвитку, можна розглянути додання механізму накопичення грошей та можливості експорту даних.

На мою думку, ринок мобільної розробки для платформи iOS має дуже позитивну тенденцію розвитку. За 7 років свого існування, мова Swift вже дійшла до 5.6 версії та продовжує стрімко розвиватись, щоб не відставати від сучасних трендів розробки. Те саме стосується і фреймворків для створення графічного інтерфейсу. Все більше додатків на UIKit інтегрують компоненти, написані з використанням SwiftUI. На жаль, SwiftUI поки не є настільки ж стабільним як його попередник, однак вже почався плавний перехід на новий фреймворк. Можна очікувати, що скоро він майже повністю витіснить UIKit, як це колись відбулось з Objective-C та Swift.

Список використаних джерел

1. How much time on average do you spend on your phone on a daily basis? [Електронний ресурс] – https://www.statista.com/statistics/1224510/time-spent-per-day-on-smartphone-us/ Дата доступу: 18.05.2022.
2. 15 statistics that prove the power of data visualization [Електронний ресурс] – https://blog.csgsolutions.com/15-statistics-prove-power-data-visualization Дата доступу: 18.05.2022.
3. 6 Types of Visual Content You Need to Use in Your Marketing Campaigns [Електронний ресурс] – https://neilpatel.com/blog/visual-content-you-need-to-use-in-your-marketing-campaign/ Дата доступу: 18.05.2022.
4. App Sandbox [Електронний ресурс] - https://developer.apple.com/documentation/security/app_sandbox Дата доступу: 18.05.2022.
5. Mac Catalyst [Електронний ресурс] – https://appleinsider.com/inside/mac-catalyst Дата доступу: 18.05.2022.
6. App Store Review Guidelines [Електронний ресурс] – https://developer.apple.com/app-store/review/guidelines/#intellectual-property Дата доступу: 18.05.2022.
7. Human Interface Guidelines [Електронний ресурс] – https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/ Дата доступу: 18.05.2022.
8. The Missing Guide for Mac Catalyst Apps [Електронний ресурс] – https://www.craft.do/maccatalyst-guide/b/54CEAE5D-E849-4222-B1DB-4D0898FF4FB4/Design Дата доступу: 18.05.2022.
9. Dispatch [Електронний ресурс] – https://developer.apple.com/documentation/DISPATCH Дата доступу: 18.05.2022.

10. Operation Queue [Електронний ресурс] – https://developer.apple.com/documentation/foundation/operationqueue Дата доступу: 18.05.2022.
11. About Objective-C [Електронний ресурс] – https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html Дата доступу: 18.05.2022.
12. Swift vs. Objective-C: A Look at iOS Programming Languages [Електронний ресурс] – https://www.upwork.com/resources/swift-vs-objective-c-a-look-at-ios-programming-languages Дата доступу: 18.05.2022.
13. Optional [Електронний ресурс] – https://developer.apple.com/documentation/swift/optional Дата доступу: 18.05.2022.
14. The Basics [Електронний ресурс] – https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID330 Дата доступу: 18.05.2022.
15. ARC [Електронний ресурс] – https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html Дата доступу: 18.05.2022.
16. Copy-on-Write Mechanisms [Електронний ресурс] https://medium.com/@lucianoalmeida1/understanding-swift-copy-on-write-mechanisms-52ac31d68f2f Дата доступу: 18.05.2022.
17. Inheritance [Електронний ресурс] https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html Дата доступу: 18.05.2022.
18. Initialization [Електронний ресурс] https://docs.swift.org/swift-book/LanguageGuide/Initialization.html Дата доступу: 18.05.2022.
19. Functions [Електронний ресурс] https://docs.swift.org/swift-book/LanguageGuide/Functions.html Дата доступу: 18.05.2022.

20. Why would you want to use closures [Електронний ресурс] https://www.hackingwithswift.com/quick-start/understanding-swift/why-would-you-want-to-use-closures-as-parameters Дата доступу: 18.05.2022.
21. Opaque Types [Електронний ресурс] https://docs.swift.org/swift-book/LanguageGuide/OpaqueTypes.html Дата доступу: 18.05.2022.
22. What is the difference between Core Data and SQLite [Електронний ресурс] https://cocoacasts.com/what-is-the-difference-between-core-data-and-sqlite/ Дата доступу: 18.05.2022.
23. SQLite [Електронний ресурс] http://sqlite.org/mostdeployed.html Дата доступу: 18.05.2022.
24. Pros and cons of SQLite [Електронний ресурс] https://www.trustradius.com/products/sqlite/reviews?qs=pros-and-cons#reviews Дата доступу: 18.05.2022.
25. Core Data vs Realm [Електронний ресурс] https://agilie.com/blog/coredata-vs-realm-what-to-choose-as-a-database-for-ios-apps Дата доступу: 18.05.2022.
26. What is Core Data [Електронний ресурс] https://cocoacasts.com/what-is-core-data Дата доступу: 18.05.2022.
27. Core Data [Електронний ресурс] https://developer.apple.com/documentation/coredata Дата доступу: 18.05.2022.
28. Core Data or Realm [Електронний ресурс]https://cocoacasts.com/core-data-or-realm Дата доступу: 18.05.2022.
29. A light intro into Core Data [Електронний ресурс] https://betterprogramming.pub/a-light-intro-to-core-data-part-un-e344f9d1528 Дата доступу: 18.05.2022.
30. SQLite with Swift tutorial [Електронний ресурс] https://www.raywenderlich.com/6620276-sqlite-with-swift-tutorial-getting-started#toc-anchor-001 Дата доступу: 18.05.2022.
31. Firebase vs MySQL [Електронний ресурс] https://www.integrate.io/blog/firebase-vs-mysql/ Дата доступу: 18.05.2022.

32. Realm [Електронний ресурс] https://stackshare.io/realm Дата доступу: 18.05.2022.
33. The best architecture for iOS [Електронний ресурс] https://oleksandr-stepanov.medium.com/the-best-architecture-for-ios-app-does-it-even-exist-3af357ac62e7 Дата доступу: 18.05.2022.
34. MVP architecture pattern [Електронний ресурс] https://saad-elloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3 Дата доступу: 18.05.2022.
35. iOS architecture patterns [Електронний ресурс] https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52 Дата доступу: 18.05.2022.
36. SwiftUI vs UIKit [Електронний ресурс] https://steelkiwi.medium.com/swiftui-vs-uikit-benefits-and-drawbacks-6a540cced684 Дата доступу: 18.05.2022.
37. Will SwiftUI kill UIKit [Електронний ресурсhttps://topdevs.org/blog/will-swiftui-kill-uikit Дата доступу: 18.05.2022.
38. SwiftUI vs UIKit [Електронний ресурс] https://www.bluelabellabs.com/blog/swiftui-vs-uikit/ Дата доступу: 18.05.2022.
39. Answering the big question. Should you learn SwiftUI or UIKit [Електронний ресурс] https://www.hackingwithswift.com/quick-start/swiftui/answering-the-big-question-should-you-learn-swiftui-uikit-or-both Дата доступу: 18.05.2022.
40. Utilizing the debug view hierarchy [Електронний ресурс] https://medium.com/stacc/utilizing-the-debug-view-hierarchy-to-better-understand-your-apps-ui-c8655f6d34e9 Дата доступу: 18.05.2022.
41. Learning SwiftUI or UIKit [Електронний ресурс] https://bignerdranch.com/blog/learning-apples-swiftui-or-uikit-which-one-is-right-for-you-right-now/ Дата доступу: 18.05.2022.
42. Combine [Електронний ресурс] https://developer.apple.com/documentation/combine Дата доступу: 18.05.2022.
43. Swift Combine [Електронний ресурс] https://peterfriese.dev/posts/swift-combine-love/ Дата доступу: 18.05.2022.

44. UIView [Електронний ресурс] https://developer.apple.com/documentation/uikit/uiview Дата доступу: 18.05.2022.
45. CALayer tutorial for iOS [Електронний ресурс] https://www.raywenderlich.com/10317653-calayer-tutorial-for-ios-getting-started Дата доступу: 18.05.2022.
46. What is CALayer [Електронний ресурс] https://www.hackingwithswift.com/example-code/calayer/what-is-calayer Дата доступу: 18.05.2022.
47. UITableView [Електронний ресурс] https://developer.apple.com/documentation/uikit/views_and_controls/table_views Дата доступу: 18.05.2022.
48. Delegate pattern in Swift [Електронний ресурс] https://dilloncodes.com/delegate-pattern-in-swift Дата доступу: 18.05.2022.
49. Core Data code generation [Електронний ресурс] https://www.swiftdevjournal.com/core-data-code-generation/ Дата доступу: 18.05.2022.
50. Lightweight migrations in Core Data [Електронний ресурс] https://www.raywenderlich.com/7585-lightweight-migrations-in-core-data-tutorial Дата доступу: 18.05.2022.
51. Push notifications [Електронний ресурс] https://www.raywenderlich.com/11395893-push-notifications-tutorial-getting-started Дата доступу: 18.05.2022.
52. Local notifications with the UserNotification framework [Електронний ресурс] https://cocoacasts.com/local-notifications-with-the-user-notifications-framework Дата доступу: 18.05.2022.
53. Scheduling notifications [Електронний ресурс] https://www.hackingwithswift.com/read/21/2/scheduling-notifications-unusernotificationcenter-and-unnotificationrequest Дата доступу: 18.05.2022.

54. URLSession [Електронний ресурс] https://developer.apple.com/documentation/foundation/urlsession Дата доступу: 18.05.2022.
55. URLSession tutorial [Електронний ресурс] https://www.raywenderlich.com/3244963-urlsession-tutorial-getting-started Дата доступу: 18.05.2022.
56. Concurrency [Електронний ресурс] https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html Дата доступу: 18.05.2022.