МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:
РОЗРОБКА ТА АНАЛІЗ ЕФЕКТИВНОСТІ МОБІЛЬНОГО ДОДАТКУ ДЛЯ УПРАВЛІННЯ ВЛАСНИМИ ФІНАНСАМИ НА IOS.

Здобувача вищої освіти другого (магістерського) рівня,
ІІ курсу, групи КМІ-24-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології в економіці»

(підпис) Татьяніна Назара Едуардовича
(прізвище, ім'я, по батькові)

Науковий керівник (підпис) Горбовий Артур Юліанович, д-р техн. наук, професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми (підпис) Омельчук Антон Анатолійович, канд. техн. наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри протокол від «02» січня 2026 № 10

Завідувач кафедри (підпис) Філоненко Михайло Миколайович, канд. фіз.-мат. наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2026

Реєстрація № 6 16.01.2026 (дата) _____ (підпис відповідальної особи)

Результати перевірки до захисту (до захисту, на доопрацювання) 02.01.2026 (дата) _____ (підпис)

Результат захисту 5 (відмінно) (оцінка) 09.02.2026 (дата) _____ (підпис)

Голова комісії: _____ (підпис) О.С. Дюбанов

Члени комісії: _____ (підпис) А.Ю. Горбовий

_____ (підпис) М.М. Філоненко

_____ (підпис) І.Д. Погорєловська

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст кваліфікаційної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	28		09.02.2026 Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.
2. Дотримання вимог щодо оформлення	від 0 до 10	9		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	5		
4. Рецензент	від 0 до 5	5		
РАЗОМ БАЛІВ	від 0 до 50	47		

Оцінювання кваліфікаційної роботи науковим керівником - 45 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
92	A	5	09.02.2026	Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Інформаційні управляючі системи і технології в економіці»

Затверджую
Завідувач кафедри _____ Філоненко М.М.
«11» грудня 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Татьяніну Назару Едуардовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Розробка та аналіз ефективності мобільного додатку для управління власними фінансами на iOS
Затверджена наказом ректора від «10» грудня 2024 № 1852
2. Строк здачі закінченої роботи «02» січня 2026
3. Мета роботи розробка та дослідження ефективності мобільного додатку для управління власними фінансами на платформі iOS.
Об'єкт дослідження процес проєктування та розробки нативного мобільного програмного забезпечення для операційної системи iOS.
Предмет дослідження методи, інструментальні засоби розробки (Swift, SwiftUI/UIKit) та показники ефективності роботи фінансового мобільного додатку.
Орієнтовна база дослідження розроблений мобільний додаток для обліку фінансів (авторська розробка).
4. Вихідні дані до роботи технічне завдання на розробку, вимоги Apple Human Interface Guidelines, методики тестування програмного забезпечення, статистичні дані тестування.
5. Перелік графічного матеріалу діаграма варіантів використання (Use Case), діаграма класів системи, схема архітектури додатку (MVVM/Clean Architecture), дизайн інтерфейсу, графіки порівняльного аналізу швидкодії та використання пам'яті.
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)
РОЗДІЛ 1. КРИТИЧНИЙ ОГЛЯД ЛІТЕРАТУРИ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ РОЗРОБКИ
1.1. Аналіз предметної області та сучасні тенденції ринку FinTech
1.2. Обґрунтування вибору мови програмування
РОЗДІЛ 2. ПРАКТИЧНІ РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ: ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ДОДАТКУ
2.1. Проєктування UX/UI дизайну та сценаріїв взаємодії користувача (User Flow)
2.2. Розробка архітектури системи та інфологічної моделі бази даних
2.3. Програмна реалізація та аналіз показників ефективності

РОЗДІЛ 3. ПЕРСПЕКТИВИ РОЗВИТКУ ТА ОЦІНКА ПРАКТИЧНОЇ ЕФЕКТИВНОСТІ ВПРОВАДЖЕННЯ ПРОДУКТУ

3.1. Пропозиції щодо функціонального удосконалення системи на основі інтелектуальних технологій

3.2. Розрахунок економічного ефекту від впровадження та соціальна значимість

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 15.12.2024	До 15.12.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	28.05.2025	28.05.2025
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	08.09.2025	08.09.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	17.11.2025	17.11.2025
5	**Висновки**	08.12.2025	08.12.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	22.12.2025	22.12.2025
7	Отримання відгуку керівника	20.01.2026	20.01.2026
8	Отримання зовнішньої рецензії	23.01.2026	23.01.2026

Дата видачі завдання «11» грудня 2024

8. Керівник кваліфікаційної роботи _____ (підпис) Горбовий А.Ю. (прізвище, ініціали)

9. Керівник освітньо-професійної програми _____ (підпис) Омельчук А.А. (прізвище, ініціали)

10. Завдання до виконання отримав здобувач вищої освіти _____ (підпис) Татьянін Н.Е. (прізвище, ініціали)

АНОТАЦІЯ

Розробка та аналіз ефективності мобільного додатку для управління власними фінансами на iOS

Мета роботи полягає у проєктуванні, програмній реалізації та аналізі технічної ефективності нативного мобільного додатку під операційну систему iOS для обліку персональних доходів та витрат. Проєкт повинен стати інвестиційно привабливим для розробника.

Завданнями роботи є:

– проведення аналізу предметної області та існуючих рішень на ринку FinTech;

– розробка ергономічного дизайну згідно стандартів Apple Human Interface Guidelines;

– створення функціонального мобільного додатку з використанням фреймворків SwiftUI та SwiftData;

– впровадження технологій комп'ютерного зору (OCR) для автоматизації введення даних;

– проведення аналізу ефективності роботи додатку (швидкодія, споживання пам'яті).

Предметом дослідження є методи, інструментальні засоби розробки та показники ефективності нативного мобільного програмного забезпечення.

Об'єктом дослідження є процес проєктування та розробки мобільного додатку з використанням мови Swift та архітектурного патерну MVVM.

Базою дослідження виступає розроблений мобільний додаток.

За результатами дослідження сформульовані переваги використання нативного технологічного стеку для забезпечення високої продуктивності та безпеки даних користувача.

Одержані результати можуть бути використані широким колом користувачів iPhone для підвищення особистої фінансової грамотності, а також розробниками для оптимізації мобільних рішень.

Кваліфікаційна робота містить 55 сторінок, 5 таблиць, 5 рисунків, список використаних джерел 52 найменувань, 8 додатків.

Ключові слова: iOS, Swift, SwiftUI, SwiftData, мобільний додаток, фінанси, OCR, ефективність, архітектура MVVM.

Рік виконання кваліфікаційної роботи 2025

Рік захисту роботи 2026

ABSTRACT

Development and analysis of the effectiveness of a mobile application for managing personal finances on iOS

The purpose of this thesis is to design, program, and analyze the technical effectiveness of a native mobile application for the iOS operating system for tracking personal income and expenses. The project should be attractive to developers in terms of investment.

The objective of the work are:

– analysis of the subject area and existing solutions on the FinTech market;

– development of an ergonomic design in accordance with Apple Human Interface Guidelines standards;

– creation of a functional mobile application using SwiftUI and SwiftData frameworks;

– implementing computer vision (OCR) technologies to automate data entry;

– analyzing the app's performance (speed, memory consumption).

The subject of the study is the methods, development tools, and performance indicators of native mobile software.

The object of the study is the process of designing and developing a mobile application using the Swift language and the MVVM architectural pattern.

The basis of the study is the developed mobile application.

Based on the results of the study, the advantages of using a native technology stack to ensure high performance and user data security are formulated.

The results obtained can be used by a wide range of iPhone users to improve their personal financial literacy, as well as by developers to optimize mobile solutions.

The qualification work contains 55 pages, 5 tables, 5 figures, a list of used sources with 52 titles, and 8 appendices.

Keywords: iOS, Swift, SwiftUI, SwiftData, mobile application, finance, OCR, efficiency, MVVM architecture.

Year of completion of the qualification work: 2025

Year of thesis defense: 2026

ЗМІСТ

ВСТУП 3
РОЗДІЛ 1. КРИТИЧНИЙ ОГЛЯД ЛІТЕРАТУРИ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ РОЗРОБКИ 6
1.1. Аналіз предметної області та сучасні тенденції ринку FinTech 6
1.2. Обґрунтування вибору мови програмування 11
РОЗДІЛ 2. ПРАКТИЧНІ РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ: ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ДОДАТКУ 23
2.1. Проєктування UX/UI дизайну та сценаріїв взаємодії користувача (User Flow) 23
2.2. Розробка архітектури системи та інфологічної моделі бази даних 30
2.3. Програмна реалізація та аналіз показників ефективності 35
РОЗДІЛ 3. ПЕРСПЕКТИВИ РОЗВИТКУ ТА ОЦІНКА ПРАКТИЧНОЇ ЕФЕКТИВНОСТІ ВПРОВАДЖЕННЯ ПРОДУКТУ 42
3.1. Пропозиції щодо функціонального удосконалення системи на основі інтелектуальних технологій 42
3.2. Розрахунок економічного ефекту від впровадження та соціальна значимість проєкту 44
ВИСНОВКИ 48
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 51
ДОДАТКИ 56

ВСТУП

Актуальність теми. В наш час склалося так, що мобільні технології стали невід'ємною частиною сучасного повсякдення. Тепер смартфон знаходить своє застосування майже у всіх сферах людської діяльності. З кожним роком у країні зростає кількість безготівкових операцій, зростає попит на інструменти для контролю власних коштів. В умовах економічної нестабільності мобільний додаток для управління фінансами – це можливість взяти під контроль свій бюджет, проаналізувати структуру витрат, оптимізувати накопичення, а також це відмінний засіб для підвищення рівня власної фінансової грамотності.

У сучасних реаліях наявність фінансового асистента в телефоні вже не елемент розкоші, а необхідний інструмент для забезпечення матеріального добробуту. Що стосується персональних фінансів, нативний додаток на iOS дає перевагу перед застарілими методами обліку (Excel або блокнот), які є незручними в дорозі, або перед конкурентами, які пропонують повільні рішення з поганою оптимізацією.

Створення мобільного додатку для управління фінансами є важливим елементом стратегії особистої ефективності, оскільки це дозволяє швидко фіксувати витрати та уникати імпульсивних покупок. Крім того, додаток може допомогти встановити фінансові цілі, вести аналітику за періодами, повідомляти про перевищення лімітів бюджету, що допоможе зберегти більше коштів. Також якісно розроблений додаток дозволяє користувачеві бути в курсі реального стану своїх рахунків та відстежувати динаміку витрат у реальному часі, що також допоможе вести більш успішне фінансове планування [31].

Різноманітні інноваційні підходи та особливості розробки мобільних інформаційних систем відображено у працях як вітчизняних авторів: Левикіна В.М. [1], Литвина В.В. [2], Яковини В.С. [5], так і зарубіжних: Martin R.C. [36], Apple Inc. (Human Interface Guidelines) [8], Nielsen J. [45].

Метою кваліфікаційної роботи є розробка та аналіз ефективності мобільного додатку для управління власними фінансами на iOS.

Для досягнення мети необхідно вирішити такі завдання, а саме:

– проведення аналізу предметної області, де було проаналізовано ринок FinTech-додатків та визначено актуальність створення власного рішення для платформи iOS;

– розробку ергономічного дизайну, де в рамках дослідження були вивчені сучасні тенденції Human Interface Guidelines та було розроблено інтуїтивно зрозумілий інтерфейс, який відображає стандарти якості екосистеми Apple;

– створення функціонального мобільного додатку, де була проведена розробка та реалізація програми, що включає основну логіку та функціональні елементи, такі як облік транзакцій та візуалізація статистики витрат;

– проведення аналізу ефективності, де була оцінена швидкодія роботи додатку, споживання пам’яті та енергоресурсів, що дозволяє користувачам швидко та без затримок взаємодіяти з даними на різних моделях iPhone.

Об’єкт дослідження – процес проєктування та розробки мобільного програмного забезпечення для операційної системи iOS.

База дослідження – розроблений мобільний додаток для управління фінансами.

Предмет дослідження – методи, інструментальні засоби розробки (Swift, SwiftUI) та показники ефективності роботи додатку.

Методи дослідження – теоретичні – аналіз, узагальнення, систематизація технічної документації Apple, матеріалів конференцій, Інтернет-ресурсів для виділення теоретичних засад дослідження; емпіричні – програмна реалізація, профілювання коду в Xcode Instruments, порівняльний аналіз для констатування стану розв’язання проблеми ефективності.

Науково-практична важливість отриманих результатів полягає в тому, що розробка власного фінансового додатку може допомогти покращити ефективність управління особистим бюджетом. Додаток може залучити користувачів своєю швидкодією та надійністю, а також забезпечити зручний спосіб для аналізу витрат «на ходу».

Практичне значення отриманих результатів. Отримані результати кваліфікаційного проєкту з розробки та аналізу мобільного додатку мають велике практичне значення для сфери розробки ПЗ. Створення ефективного, оптимізованого додатку демонструє переваги нативної розробки, допомагає користувачам заощаджувати час та гроші. Додаток може бути корисним для широкої аудиторії користувачів iPhone, які прагнуть цифровізувати свій фінансовий облік та покращити контроль над власними коштами.

Апробація результатів дослідження здійснювалась на науково- практичних конференціях: IV Міжнародній науково-практичній Інтернет-конференції “Інновації та перспективні шляхи розвитку інформаційних технологій (ІПШРІТ- 2025)”, (м. Черкаси 2025 р.)[3] (Додаток Ж); IV Міжнародній науково-практичній інтернет-конференції "Цифрова трансформація фінансової системи України та країн V-4 в умовах євроінтеграції" (м. Дубляни 2024 р.)[4] (Додаток З).

РОЗДІЛ 1. КРИТИЧНИЙ ОГЛЯД ЛІТЕРАТУРИ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ РОЗРОБКИ

1.1. Аналіз предметної області та сучасні тенденції ринку FinTech

Сучасний етап розвитку світової економіки характеризується глибокою інтеграцією інформаційних технологій у повсякденне життя, що призводить до зміни фундаментальних звичок споживання. Одним із ключових трендів останнього десятиліття є перехід до «безготівкового суспільства» (cashless society). Згідно зі звітами міжнародних фінансових організацій, частка безготівкових операцій у світі щорічно зростає на 12–15%, а в країнах з розвиненим фінтех-сектором, до яких належить і Україна, цей показник є ще вищим [52].

Цей технологічний зсув має важливий психологічний аспект. Дослідження у сфері поведінкової економіки вказують на феномен зниження «болю оплати» (pain of paying) при використанні цифрових засобів розрахунку. Витрачання електронних коштів, особливо через безконтактні системи (Apple Pay, Google Pay), сприймається людським мозком менш критично, ніж розставання з фізичними банкнотами. Це призводить до збільшення кількості імпульсивних покупок та втрати інтуїтивного контролю над залишком коштів [46].

Додатковим фактором актуалізації проблеми є макроекономічна нестабільність. В умовах зростання інфляції та економічної невизначеності домогосподарства змушені переходити до більш жорстких моделей бюджетування. Традиційні методи обліку, такі як ведення записів у зошиті або використання десктопних таблиць Excel, втрачають свою ефективність через низьку оперативність. Сучасний користувач дотримується концепції «Mobile First» – смартфон стає єдиним центром управління життям, і фінанси не є виключенням. Відтак, потреба у мобільному інструменті, який дозволяє миттєво фіксувати витрати та аналізувати бюджет, стає критичною необхідністю для забезпечення фінансової стійкості [29].

Ринок мобільного програмного забезпечення реагує на зазначені вище запити стрімким зростанням пропозиції. За даними аналітичної платформи Statista, категорія «Finance» стабільно входить до п’ятірки найпопулярніших категорій в Apple App Store за кількістю завантажень, конкуруючи з соціальними мережами та іграми. Станом на 2025 рік глобальна кількість завантажень фінансових додатків перевищила 7 мільярдів, що свідчить про масовий попит на цифрові фінансові послуги [28].

Аналіз користувацьких відгуків та рейтингів показує, що сучасна аудиторія iOS висуває специфічні вимоги до програмного забезпечення цього класу. На відміну від користувачів Android, власники техніки Apple частіше звертають увагу на естетику інтерфейсу, підтримку нативних функцій системи (віджети, синхронізація з iCloud) та приватність даних. Ключовим фактором успіху додатку стає не лише його функціональність, а й швидкість роботи (Performance) та зручність користування (Usability). Користувачі не готові витрачати час на вивчення складних інтерфейсів; вони очікують, що процес додавання транзакції займатиме лічені секунди [44].

Для визначення місця проєктованого додатку в існуючій екосистемі необхідно провести класифікацію наявних PFM-рішень (Personal Finance Management). Аналіз предметної області дозволяє виділити три основні групи додатків:

1. Банківські екосистеми (Neobanks). До цієї групи належать додатки конкретних банків (Monobank, Privat24, Revolut). Їхня головна перевага – автоматична фіксація транзакцій без участі користувача. Проте ці системи є «закритими» (Walled Gardens): вони не дозволяють вести облік готівкових коштів або агрегувати дані з рахунків інших банків без використання складних інтеграцій.

2. Автоматизовані агрегатори (Bank Sync Aggregators). Додатки типу YNAB, Mint або Saldo, які використовують API (Open Banking) для підключення до різних банківських рахунків користувача. Вони автоматично завантажують історію операцій та категоризують її. Попри зручність, цей тип додатків має

суттєві недоліки: високі ризики кібербезпеки (необхідність передачі банківських облікових даних третій стороні), залежність від стабільності банківських шлюзів та висока вартість підписки [42].

3. Автономні трекери витрат (Manual Expense Trackers). Це додатки (Monefy, CoinKeeper, Spending Tracker), де користувач вносить дані вручну. Хоча цей метод вимагає дисципліни, він має унікальні переваги: повну приватність (дані зберігаються локально на пристрої), незалежність від інтернет-з'єднання та універсальність.

Порівняльний аналіз цих підходів наведено в таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз типів PFM-додатків

Критерій	**Banking Apps (Neobanks)**	**Automated Aggregators**	**Manual Expense Trackers**
Введення даних	Повністю автоматичне	Автоматичне (API)	Ручне
Приватність	Висока (Банківський рівень)	Низька/Середня (Доступ 3-х сторін)	Дуже висока (Локальна)
Покриття	Лише один банк	Декілька банків	Усі активи (Готівка, Крипто)
Усвідомленість	Низька (Пасивний трекінг)	Низька (Пасивний трекінг)	Висока (Активна участь)
Залежність	Стабільність банку	API та Інтернет	Відсутня (Offline)
Вартість	Безкоштовно	Висока вартість підписки	Безкоштовно / Разова оплата

Джерело: [розроблено автором]

На основі проведеного аналізу для магістерської роботи обрано концепцію автономного трекера витрат (Manual Expense Tracker). Цей вибір обґрунтований трьома факторами. По-перше, ручне введення даних сприяє підвищенню фінансової грамотності та усвідомленості витрат (mindfulness). По-друге, архітектура з локальним зберіганням даних (Local Storage) дозволяє забезпечити максимальну швидкодію додатку та гарантувати користувачеві повну конфіденційність, що є конкурентною перевагою на тлі постійних витоків даних із хмарних сервісів. По-третє, такий підхід дозволяє зосередитися на оптимізації програмної реалізації та UI/UX, що відповідає темі дослідження ефективності.

Для глибшого розуміння функціональних та ергономічних недоліків існуючих рішень було проведено детальне дослідження (Benchmarking) найпопулярніших додатків у сегменті «Personal Finance» магазину App Store. Об’єктами аналізу стали додатки Monefy, CoinKeeper та Wallet, які репрезентують різні підходи до взаємодії з користувачем.

Додаток Monefy є класичним прикладом візуального підходу. Його інтерфейс побудований навколо кругової діаграми, оточеної іконками категорій. Хоча такий дизайн є інтуїтивно зрозумілим для новачків, він має суттєві обмеження масштабованості. При збільшенні кількості категорій витрат (що неминуче стається з часом), головний екран стає перевантаженим, а інтерактивні зони (touch targets) зменшуються до розмірів, що ускладнюють точне натискання. Крім того, аналітичні можливості Monefy обмежені лише базовою статистикою, що не дозволяє користувачеві виявляти складні патерни своєї фінансової поведінки [34].

Інший відомий конкурент, CoinKeeper, використовує унікальну метафору «перетягування монети» (Drag-and-Drop) з рахунку на категорію витрат. Цей елемент гейміфікації робить процес обліку цікавішим, проте з технічної точки зору він часто конфліктує зі стандартними жестами операційної системи iOS (наприклад, системним жестом «Назад» або перемиканням між додатками). Більше того, реалізація кастомного графічного рушія для анімації монет призводить до підвищеного споживання ресурсів процесора, що на старих моделях iPhone викликає помітні затримки (lags) інтерфейсу.

Третій розглянутий аналог, Wallet від BudgetBakers, є прикладом надмірної функціональності (Feature Creep). Намагаючись задовольнити потреби всіх груп користувачів, розробники перетворили мобільний додаток на складний комбайн з десятками меню, налаштувань та типів звітів. Такий підхід суперечить принципам мобільної ергономіки, де пріоритетом має бути швидкість виконання ключового сценарію. Внаслідок перевантаженості інтерфейсу, користувач змушений витрачати надмірні когнітивні зусилля на орієнтацію в системі, що нівелює головну перевагу мобільного інструменту – оперативність.

Результати порівняльного аналізу систематизовано в таблиці 1.2.

Таблиця 1.2

Порівняльний аналіз ключових конкурентів на ринку

Критерії	Monefy	CoinKeeper	Wallet
Основна взаємодія	На основі дотиків (Кругова діаграма)	На основі жестів (Перетягування / Drag & Drop)	На основі форм (Традиційна)
Архітектура	Нативна (Базова)	Елементи крос-платформності	Гібридна / Орієнтована на хмару
Проблеми UX	Захаращений інтерфейс з великою кількістю категорій.	Нестандартні жести порушують гайдлайни (HIG).	Перенасичення інформацією, складна навігація.
Продуктивність	Висока	Низька на старих пристроях (затримка анімації).	Середня (повільна синхронізація).
Монетизація	Freemium (безкоштовно з платними функціями)	Підписка	Агресивна підписка
Зберігання даних	Локальне + Резервне копіювання	Хмарна синхронізація	Хмара передусім (Cloud First)

Джерело: [розроблено автором]

Критичним фактором, що визначає життєвий цикл використання фінансового додатку, є рівень когнітивного навантаження (Cognitive Load). Дослідження показують, що основною причиною відмови користувачів від ведення бюджету є так зване «тертя інтерфейсу» (UI Friction). Якщо сценарій додавання однієї транзакції вимагає більше 3-5 секунд або понад 4 натискань на екран, у користувача формується негативний досвід, і з часом він припиняє користуватися продуктом [45]. Більшість існуючих на ринку рішень грішать саме цим: вони вимагають обов’язкового заповнення полів дати, вибору підкатегорії, тегів, рахунків тощо.

Ще однією системною проблемою є недотримання стандартів Apple Human Interface Guidelines (HIG). Багато розробників, прагнучи зекономити ресурси, використовують кросплатформні фреймворки (Flutter, React Native), щоб створити єдиний додаток для iOS та Android. Це призводить до ефекту «зловісної долини» в інтерфейсі: елементи виглядають схожими на нативні, але поводяться інакше (інша інерція прокрутки, відсутність тактильного відгуку Haptic Feedback, нестандартні шрифти). Для користувачів iOS, які звикли до

високої якості екосистеми, такі дрібниці створюють відчуття «чужорідності» та низької якості продукту [8].

Окрім ергономічних недоліків, існуючі PFM-рішення мають низку технічних проблем, вирішення яких є метою даної магістерської роботи.

По-перше, це проблема швидкодії («Холодний старт»). Кросплатформні додатки вимагають ініціалізації віртуальної машини або мосту (Bridge) перед запуском, що збільшує час відкриття програми. В контексті фінансового трекера, який часто відкривають у черзі на касі або в транспорті, кожна секунда затримки є критичною. Нативний додаток, написаний на Swift з використанням SwiftUI, здатен запускатися миттєво, що є значною конкурентною перевагою.

По-друге, це питання безпеки та суверенітету даних. Більшість сучасних додатків (Wallet, YNAB) використовують архітектуру «Cloud First», зберігаючи всю фінансову історію користувача на віддалених серверах. Це створює ризики витоку конфіденційної інформації у випадку хакерської атаки на сервери компанії-розробника. Альтернативний підхід «Local First», коли база даних (Core Data або SwiftData) знаходиться безпосередньо на пристрої користувача у захищеній «пісочниці» (Sandbox), є більш надійним та відповідає сучасним вимогам до приватності [32].

1.2. Обґрунтування вибору мови програмування

Вибір мови програмування є фундаментальним рішенням, що визначає весь подальший життєвий цикл програмного забезпечення, його архітектуру, надійність та можливості подальшого масштабування. В контексті розробки під платформу iOS розробники протягом тривалого часу перебувають перед вибором між класичною мовою Objective-C, яка була стандартом індустрії понад 30 років, та сучасною мовою Swift, представленою компанією Apple у 2014 році. Для наукового обґрунтування вибору інструментарію магістерської роботи необхідно провести глибокий порівняльний аналіз цих двох технологій за критеріями безпеки, продуктивності та підтримки сучасних парадигм програмування.

Історичний контекст та еволюція. Мова Objective-C, розроблена на початку 1980-х років, є строгим надмножиною мови C з додаванням об'єктно-орієнтованих можливостей у стилі Smalltalk. Вона базується на концепції динамічної відправки повідомлень (Dynamic Message Dispatch). Це означає, що рішення про те, який метод викликати, приймається не на етапі компіляції, а безпосередньо під час виконання програми (Runtime). Хоча такий підхід забезпечує високу гнучкість, він несе в собі суттєві ризики: помилки типу «message sent to deallocated instance» або спроби виклику неіснуючих методів призводять до аварійного завершення роботи додатку (Crash) вже у користувача [49].

На відміну від неї, Swift була спроєктована з урахуванням накопиченого досвіду інженерії програмного забезпечення за останні десятиліття. Основною філософією Swift є «безпека за дизайном» (Safety by Design). Це компільована мова загального призначення зі строгою статичною типізацією, яка використовує сучасний компілятор LLVM (Low Level Virtual Machine).

Аналіз безпеки коду (Type Safety). Ключовою перевагою Swift, що безпосередньо впливає на надійність фінансового додатку, є система типів. В Objective-C розробник часто працює з вказівниками на об'єкти типу id (аналог void* у C), що дозволяє передавати будь-які об'єкти у будь-які методи. Це створює простір для людських помилок, які компілятор не може відстежити. Swift вирішує цю проблему через механізм строгої типізації та введення концепції Optionals (Опціональних типів). Опціонали вимагають від розробника явно обробляти ситуації, коли значення змінної відсутнє (nil). У фінансовому додатку це критично важливо: наприклад, при отриманні балансу рахунку з бази даних, значення може бути відсутнім (якщо рахунок новий). Objective-C у такому випадку міг би продовжити виконання програми з "сміттєвими" даними або впасти, тоді як Swift примушує обробити цей кейс на етапі написання коду (Compile Time) [24]. Дослідження показують, що використання Swift дозволяє зменшити кількість помилок часу виконання (Runtime Errors) на 30-40% порівняно з проєктами на Objective-C.

Порівняння продуктивності (Performance Analysis). Одним із завдань магістерської роботи є аналіз ефективності. У цьому аспекті Swift демонструє значну перевагу. Завдяки відмові від обов'язкової динамічної диспетчеризації на користь статичної (Static Dispatch) та використання віртуальних таблиць (V-Table), виклик методів у Swift відбувається значно швидше. Згідно з офіційними бенчмарками Apple та незалежними тестами:

Алгоритми пошуку складних об'єктів у Swift працюють у 2,6 рази швидше, ніж в Objective-C.

Обробка масивів даних (що актуально для списків транзакцій) виконується майже з такою ж швидкістю, як у мові C++.

Це досягається також завдяки просунутій системі керування пам'яттю ARC (Automatic Reference Counting). Хоча ARC доступний і в Objective-C, у Swift він інтегрований на рівні мовних структур (Structs vs Classes). Використання структур (Value Types) у Swift дозволяє розміщувати дані в стеку (Stack), а не в купі (Heap), що значно зменшує накладні витрати на виділення пам'яті та знижує навантаження на процесор мобільного пристрою.

Синтаксична ефективність та підтримка коду. Swift пропонує сучасний, лаконічний синтаксис, який дозволяє реалізувати складну логіку меншою кількістю рядків коду. Відсутність необхідності створювати окремі файли заголовків (.h) та файли реалізації (.m), як це прийнято в Objective-C, спрощує структуру проєкту. Підтримка функціональних патернів програмування (map, filter, reduce) дозволяє обробляти колекції фінансових транзакцій в один рядок, що робить код більш читабельним та легким для аудиту [39]. Для магістерської роботи це означає, що час, зекономлений на боротьбі з синтаксисом та шаблонним кодом (boilerplate code), може бути витрачений на реалізацію складніших алгоритмів аналізу даних та покращення UX.

У процесі розробки мобільного додатку етап проєктування графічного інтерфейсу користувача (Graphical User Interface – GUI) є критичним, оскільки саме він визначає якість взаємодії користувача з системою (User Experience). В екосистемі iOS на сьогоднішній день існуть два основні інструментальні засоби

для створення UI: класичний фреймворк UIKit та новітній фреймворк SwiftUI. Вибір між ними ґрунтується на глибокому розумінні відмінностей між імперативною та декларативною парадигмами програмування.

Аналіз імперативного підходу (UIKit). Фреймворк UIKit, представлений разом із першим iPhone SDK у 2008 році, базується на імперативній моделі програмування. Суть цього підходу полягає в тому, що розробник повинен чітко описати послідовність команд для зміни стану інтерфейсу. Наприклад, щоб змінити текст на кнопці при натисканні, необхідно знайти посилання на цей об'єкт у пам'яті, викликати метод setTitle(...) і примусово оновити відображення. Архітектурно UIKit побудований на об'єктно-орієнтованій моделі, де кожен елемент інтерфейсу (кнопка, слайдер, таблиця) є класом (UIView), що наслідується від NSObject. Це створює значне навантаження на систему управління пам'яттю, оскільки кожен елемент інтерфейсу є посилальним типом (Reference Type) і зберігається в купі (Heap).

Основним недоліком UIKit у контексті розробки сучасних фінансових додатків є складність синхронізації стану даних та їх відображення (State Management). У складних системах, де дані можуть змінюватися з різних джерел (наприклад, синхронізація з банком у фоновому режимі та ручне введення користувачем), виникає проблема неузгодженості (Inconsistency), коли дані у базі вже оновилися, а інтерфейс ще відображає старі значення. Це вимагає написання значної кількості додаткового коду для відстеження змін (KVO, Delegates, Notifications), що збільшує ймовірність помилок [51].

Переваги декларативного підходу (SwiftUI). Фреймворк SwiftUI, анонсований Apple у 2019 році, пропонує кардинально інший, декларативний підхід. У цій парадигмі розробник описує не як змінити інтерфейс, а яким він має бути при певному стані даних. Інтерфейс у SwiftUI є функцією від стану: View = f(State).

Для магістерської роботи вибір SwiftUI обґрунтований наступними факторами технічної ефективності:

1. Єдине джерело істини (Single Source of Truth). Завдяки використанню обгорток властивостей (Property Wrappers) @State, @Binding та @ObservedObject, SwiftUI автоматично оновлює інтерфейс при зміні даних. Для фінансового трекера це критично важливо: якщо користувач додає витрату, сума загального балансу, графіки витрат та віджети оновлюються автоматично без написання жодного рядка коду для ручного оновлення UI. Це значно спрощує архітектуру та зменшує кількість багів [38].

2. Оптимізація пам'яті. На відміну від UIKit, де елементи інтерфейсу є важкими класами, у SwiftUI Views – це легковагі структури (Structs, Value Types), що створюються в стеку (Stack). Вони є надзвичайно дешевими для системи з точки зору виділення ресурсів. Система може створювати та знищувати тисячі таких структур за мілісекунди без помітного впливу на продуктивність (FPS), що відповідає меті роботи щодо створення ефективного додатку.

3. Швидкість розробки та підтримка кросплатформності. SwiftUI дозволяє використовувати механізм Live Previews у середовищі Xcode, що дає можливість бачити зміни в коді миттєво без необхідності перекомпіляції всього проєкту. Крім того, код, написаний на SwiftUI, автоматично адаптується під різні розміри екранів, підтримує динамічні шрифти (Dynamic Type) для людей з вадами зору та темну тему (Dark Mode), що є обов'язковими вимогами Apple Human Interface Guidelines.

Архітектура програмного забезпечення є скелетом додатку, який визначає правила взаємодії між різними модулями системи. Від правильності вибору архітектурного патерну залежить не лише швидкість початкової розробки, але й можливість подальшого масштабування, легкість внесення змін та, що найважливіше для кваліфікаційної роботи, можливість проведення автоматизованого тестування бізнес-логіки. Враховуючи специфіку додатку для управління фінансами, де помилка у розрахунках є неприпустимою, вибір архітектури має базуватися на принципах SOLID та Separation of Concerns (розділення відповідальності).

Аналіз класичного патерну MVC (Model-View-Controller). Протягом багатьох років компанія Apple рекомендувала використовувати патерн MVC як стандарт для розробки під iOS. У цій тріаді:

1. Model відповідає за дані.

2. View – за відображення на екрані.

3. Controller – виступає посередником, обробляє дії користувача та оновлює View.

Проте, на практиці, особливо при використанні фреймворку UIKit, цей патерн демонструє суттєвий архітектурний недолік, відомий у професійній спільноті як «Massive View Controller». Оскільки UIViewController в iOS тісно пов'язаний з життєвим циклом екрану (View Lifecycle), розробники змушені розміщувати в ньому код, що відповідає за різнорідні задачі: мережеві запити, обробку натискань, форматування даних, навігацію та роботу з базою даних. Це призводить до створення класів, що містять тисячі рядків коду, які неможливо протестувати ізольовано [36]. Для фінансового додатку це створює ризик того, що логіка розрахунку балансу буде "зашита" всередині коду відображення кнопки, що унеможливлює перевірку цієї логіки без запуску самого інтерфейсу.

Для усунення недоліків MVC та повної реалізації потенціалу декларативного фреймворку SwiftUI, для магістерської роботи обрано архітектурний патерн MVVM. Цей патерн вводить новий шар абстракції – ViewModel, який повністю відділяє бізнес-логіку від інтерфейсу користувача.

Структура додатку в парадигмі MVVM виглядатиме наступним чином:

1. Model (Модель): шар даних, ізольований від інтерфейсу. Представлений сутностями SwiftData (Transaction, Account, Budget) та відповідає виключно за цілісність інформації.

2. View (Представлення): пасивний шар SwiftUI без бізнес-логіки. Він лише відображає стан, підписуючись на оновлення ViewModel, та автоматично перемальовується при змінах.

3. ViewModel (Модель Представлення): це ключовий компонент архітектури. ViewModel – це клас, який реалізує протокол ObservableObject. Він

бере "сирі" дані з Моделі, виконує всі необхідні обчислення (наприклад, фільтрує транзакції за останній місяць, форматує дату у формат "dd.mm.yyyy", додає символ валюти) і публікує готові дані через властивості @Published.

Перевага у контексті тестування (Unit Testing): головний аргумент на користь MVVM – висока тестопридатність. Оскільки ViewModel не залежить від UI-фреймворків, вона є «чистим» Swift-класом. Це дозволяє писати швидкі модульні тести для перевірки логіки (наприклад, зміни балансу) без запуску симулятора, на відміну від повільних та складних UI-тестів у патерні MVC.

Інтеграція зі SwiftUI. MVVM є "рідним" патерном для SwiftUI. Механізм зв'язування даних (Data Binding) у SwiftUI спроєктований саме під роботу з ViewModel. Коли у ViewModel змінюється властивість (наприклад, isLoading = false), SwiftUI автоматично прибирає спіннер завантаження з екрану. Це дозволяє уникнути розсинхронізації, коли дані вже завантажились, а інтерфейс все ще показує старий стан [33].

Порівняльна характеристика розглянутих патернів наведена у таблиці 1.3.

Таблиця 1.3

Порівняльний аналіз архітектур MVC та MVVM

Характеристика	MVC (Model-View-Controller)	MVVM (Model-View-ViewModel)
Розподіл обов'язків	Controller (Контролер) обробляє логіку та налаштування інтерфейсу. Часто перевантажений.	ViewModel обробляє логіку, View (Вигляд) обробляє інтерфейс. Чітке розділення.
Тестованість	Низька. Логіка тісно пов’язана з кодом інтерфейсу (UI).	Висока. ViewModel можна тестувати ізольовано.
Складність	Низька (Добре підходить для простих прототипів).	Середня (Вимагає написання шаблонного коду / boilerplate для ViewModels).
Потік даних	Двосторонній, часто хаотичний; передбачає інтенсивне використання делегатів (Delegates).	Односпрямований потік даних (UDF) через прив'язки (Bindings).
Сумісність	Стандарт для UIKit.	Стандарт для SwiftUI.

Джерело: [розроблено автором]

Специфіка додатків для управління персональними фінансами (PFM) висуває жорсткі вимоги до підсистеми збереження даних. На відміну від новинних стрічок чи соціальних мереж, де втрата частини кешованих даних є некритичною, фінансовий додаток оперує чутливою інформацією, цілісність якої

має бути гарантована. Будь-яка транзакція (дохід чи витрата) повинна бути збережена атомарно, узгоджено та надійно (відповідно до принципів ACID). Крім того, архітектура бази даних повинна забезпечувати миттєву вибірку великих масивів даних для побудови аналітичних звітів (наприклад, агрегація витрат за категоріями за річний період).

Для реалізації «Offline-First» архітектури, яка була визначена як пріоритетна у попередніх підрозділах, необхідно обрати оптимальну технологію локальної бази даних. На ринку iOS-розробки основними конкурентами у цьому сегменті є:

1. SQLite (низькорівнева реляційна база даних).
2. Realm (стороння NoSQL об’єктна база даних).
3. Core Data (нативний фреймворк Apple для управління графами об'єктів).
4. SwiftData (новітній нативний фреймворк, надбудова над Core Data).

Аналіз сторонніх рішень (Realm Database). Realm довгий час залишався популярним вибором серед розробників завдяки простоті інтеграції та високій швидкості операцій запису, яка в деяких сценаріях перевищує показники Core Data. Realm використовує власний рушій (storage engine), написаний на C++, і надає зручний об'єктно-орієнтований API. Проте, для виконання магістерської роботи застосування Realm має наступні суттєві недоліки:

1. Збільшення розміру додатку (App Bundle Size): оскільки Realm є сторонньою бібліотекою, її підключення додає до фінального файлу додатку близько 15–20 МБ. Для «легкого» фінансового трекера це є негативним фактором технічної ефективності.

2. Залежність від третіх сторін: використання пропрієтарного рішення створює ризики підтримки. Якщо компанія-розробник (MongoDB) змінить ліцензійну політику або припинить підтримку iOS SDK, це поставить під загрозу існування проєкту.

3. Проблеми з багатопотоковістю: передача об'єктів Realm між потоками (Threads) вимагає спеціальних механізмів (Thread Safe References), що ускладнює архітектуру.

Аналіз нативних рішень (Core Data та SwiftData). Альтернативою є використання нативних інструментів Apple, які базуються на SQLite, але надають високорівневу абстракцію ORM (Object-Relational Mapping).

Core Data – це зрілий фреймворк, який існує в iOS понад 15 років. Його головною перевагою є глибока інтеграція з операційною системою. Core Data вміє ефективно управляти пам'яттю, використовуючи механізм Faulting (ліниве завантаження): об'єкти завантажуються в оперативну пам'ять лише тоді, коли до них звертаються. Це дозволяє працювати з базами даних, що містять мільйони записів, не перевантажуючи RAM мобільного пристрою [48].

SwiftData, представлений у 2023 році, є еволюційним розвитком Core Data. Він використовує ту ж саму високопродуктивну основу, але адаптує її під парадигму мови Swift. Замість складних файлів моделі (.xcdatamodeld) та ручної генерації класів, SwiftData дозволяє позначати звичайні структури Swift макросом @Model. Цей інструмент забезпечує глибоку інтеграцію з декларативним фреймворком SwiftUI, дозволяючи автоматично оновлювати інтерфейс при зміні даних завдяки новітньому механізму Observation. Такий підхід суттєво зменшує кількість шаблонного коду (boilerplate code), що спрощує архітектуру проєкту та знижує поріг входження для подальшої підтримки чи масштабування системи.

Для виконання завдання поставленого у роботі обрано SwiftData (з можливістю відкату до Core Data для старіших версій iOS) на основі наступних факторів ефективності:

1. Нульовий вплив на розмір додатку: фреймворк є частиною iOS, тому не збільшує розмір завантажуваного файлу.

2. Нативна продуктивність: SwiftData використовує внутрішні оптимізації Apple для роботи з файловою системою та пам'яттю, що забезпечує високу швидкість «холодного старту» бази даних.

3. Безпека типів: усі запити до бази даних (Predicates) перевіряються на етапі компіляції. Це унеможливлює помилки, характерні для SQLite, де SQL-запити пишуться у вигляді простих рядків тексту.

4. Інтеграція з iCloud: нативні рішення дозволяють у майбутньому реалізувати хмарну синхронізацію через CloudKit без написання серверного коду, що підвищує надійність збереження даних користувача.

Порівняльна характеристика розглянутих технологій наведена у таблиці 1.4.

Таблиця 1.4

Технічне порівняння рішень для збереження даних (iOS)

Характеристика	**SQLite**	**Realm (MongoDB)**	**Core Data / SwiftData**
Тип	Реляційна БД	Об'єктне сховище (NoSQL)	Менеджер об'єктних графів (Object Graph Manager)
Залежність	Нативна (С-бібліотека)	Стороння (~15МБ+)	Нативна (Вбудована)
Швидкість запитів	Швидка (вимагає оптимізації)	Дуже швидка	Висока (Оптимізована через кешування)
Використання пам'яті	Низька (Ручне керування)	Середня	Ефективна (Механізм Faulting / відкладене завантаження)
Типізація (Type Safety)	Відсутня (Рядкові запити)	Висока	Дуже висока (Перевірка під час компіляції)
Поріг входження	Високий (Знання SQL)	Низький	Середній (Core Data) / Низький (SwiftData)

Джерело: [розроблено автором]

Оскільки невід’ємною складовою магістерської роботи є не лише розробка, а й аналіз ефективності створеного мобільного додатку, необхідно визначити методологію та інструментарій для проведення вимірювань. Поняття «ефективність» у контексті мобільної розробки є комплексним і включає такі метрики: час відгуку інтерфейсу (UI Responsiveness), обсяг споживаної оперативної пам'яті (Memory Footprint), навантаження на центральний процесор (CPU Usage) та енергоспоживання. Суб’єктивна оцінка ("додаток працює швидко") є неприйнятною для наукової роботи, тому для отримання об’єктивних кількісних даних буде використано пакет інструментів Xcode Instruments.

Це середовище профілювання, інтегроване в IDE Xcode, яке дозволяє збирати дані про роботу додатку в реальному часі з точністю до мікросекунд. Для цілей дослідження обрано наступні модулі (інструменти):

1. Time Profiler (Профілювальник часу). Цей інструмент використовується для вимірювання навантаження на CPU та виявлення «вузьких місць» (bottlenecks) у коді. Принцип його роботи полягає у періодичному (з частотою 1 мс) знятті знімків стека викликів (Call Stack). Це дозволяє побудувати дерево викликів і точно визначити, яка саме функція або метод споживає найбільше процесорного часу. Застосування у роботі: Time Profiler буде використано для вимірювання часу «холодного старту» додатку (від натискання на іконку до готовності до роботи) та аналізу плавності прокрутки списку транзакцій (Scroll Performance). Метою є досягнення стабільного показника 60 кадрів на секунду (60 FPS) при рендерингу складних графіків [50].

2. Allocations & Leaks (Розподіл пам'яті та витоки). Управління пам'яттю є критичним аспектом стабільності iOS-додатків. Незважаючи на використання механізму автоматичного підрахунку посилань (ARC – Automatic Reference Counting), помилки проєктування можуть призводити до витоків пам'яті (Memory Leaks) – ситуацій, коли об'єкти залишаються в пам'яті після того, як вони вже не потрібні (наприклад, через Retain Cycles). Інструмент Allocations дозволяє відстежувати графік використання RAM у динаміці, фіксуючи кожен факт виділення пам'яті (malloc). Інструмент Leaks автоматично сканує пам'ять на наявність втрачених посилань. Застосування у роботі: буде проведено стрес-тестування бази даних шляхом завантаження великого масиву транзакцій (10 000+ записів) для перевірки ефективності роботи механізму кешування SwiftData та відсутності аварійних завершень роботи (OOM Crashes – Out Of Memory) [10].

3. Energy Log (Журнал енергоспоживання). Оскільки фінансовий додаток є мобільним інструментом, він не повинен суттєво впливати на час автономної роботи смартфону. Інструмент Energy Log оцінює вплив додатку на батарею, враховуючи використання CPU, мережевого модуля (Wi-Fi/LTE), геолокації та екрану. Застосування у роботі: порівняльний аналіз енергоефективності розробленого нативного додатку з кросплатформними аналогами в режимі очікування та активного використання.

4. XCTest Metrics (Автоматизовані тести продуктивності). Окрім візуального профілювання в Instruments, буде використано фреймворк XCTest для написання автоматизованих тестів продуктивності. Функція measureBlock дозволяє запускати певний блок коду (наприклад, алгоритм сортування транзакцій) багаторазово (зазвичай 10 разів), обчислювати середній час виконання та стандартне відхилення. Це забезпечує статистичну достовірність отриманих результатів і дозволяє фіксувати регресії продуктивності при внесенні змін у код [47].

Використання даного комплексу інструментів дозволить перейти від суб’єктивних оцінок до чітких математичних метрик, що є необхідною умовою для підтвердження ефективності запропонованих у магістерській роботі рішень.

РОЗДІЛ 2. ПРАКТИЧНІ РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ: ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ДОДАТКУ

2.1. Проєктування UX/UI дизайну та сценаріїв взаємодії користувача (User Flow)

Розробка сучасного мобільного програмного забезпечення вимагає глибокого розуміння принципів функціонування операційної системи, оскільки мобільні пристрої мають суттєві обмеження щодо ресурсів (заряд батареї, оперативна пам'ять, процесорний час). Проєктування якісного додатку починається з визначення моделі його життєвого циклу (App Life Cycle) – набору станів та переходів між ними, які керуються операційною системою iOS [6].

На відміну від класичного підходу на базі UIKit, де управління станами здійснювалося через методи класу AppDelegate (імперативний підхід), у розробленому додатку використано декларативний підхід фреймворку SwiftUI. Точкою входу в додаток є структура, підписана протоколом App та позначена атрибутом @main. Управління життєвим циклом здійснюється через середовище виконання (Environment) та відстеження зміни фаз сцени (ScenePhase).

Згідно з документацією Apple [51], життєвий цикл додатку в SwiftUI описується трьома основними станами:

1. Active (Активний стан): додаток знаходиться на передньому плані (Foreground), інтерфейс відображається на екрані, і користувач може з ним взаємодіяти. У цьому стані додаток споживає найбільше ресурсів CPU та GPU для обробки подій введення та рендерингу графіків.

2. Inactive (Неактивний стан): перехідний стан, коли додаток все ще видимий на екрані, але не отримує подій від користувача. Це відбувається, наприклад, при відкритті «Центру сповіщень», «Центру керування» або під час вхідного телефонного дзвінка. У цьому стані система може тимчасово призупинити оновлення інтерфейсу [38].

3. Background (Фоновий стан): додаток не відображається на екрані (користувач згорнув його або перейшов до іншої програми). У цьому стані

операційна система надає додатку мінімальний час для завершення критичних операцій, після чого процес переходить у стан Suspended (Призупинено). Якщо системі знадобиться оперативна пам'ять, призупинений додаток буде вивантажено без додаткових попереджень.

Детальну графічну схему машини станів (State Machine), що ілюструє логіку переходів між цими фазами життєвого циклу, наведено у Додатку Д.

Для фінансового додатку критично важливою є цілісність даних (Data Integrity). Втрата інформації про введену транзакцію при раптовому закритті додатку є неприпустимою. Тому в роботі реалізовано механізм автоматичного збереження контексту бази даних SwiftData при переході в стан Background.

Програмно це реалізовано шляхом відстеження змінної середовища scenePhase. Коли система сповіщає про перехід у фоновий режим, додаток примусово викликає метод збереження контексту. Лістинг програмного коду, що демонструє обробку станів життєвого циклу, наведено у Додатку А.

Такий підхід забезпечує виконання вимоги надійності: навіть якщо користувач не натиснув кнопку «Зберегти», а просто згорнув додаток після введення даних, SwiftData зафіксує зміни на диску. Як зазначає М. Нойбург у своїй праці про програмування під iOS, використання подій життєвого циклу для персистенції даних є стандартом розробки безпечних мобільних систем [41].

Окрім збереження, при поверненні в стан Active (відновлення), додаток автоматично оновлює дані (наприклад, перевіряє, чи настала дата регулярного платежу для підписок), що забезпечує актуальність інформації без необхідності перезапуску програми.

Успіх мобільного додатку в категорії персональних фінансів залежить не лише від надійності коду, але й від якості користувацького досвіду (User Experience – UX). Основною проблемою існуючих аналогів є високе когнітивне навантаження на користувача, що змушує його витрачати багато часу на рутинні операції. Тому при проєктуванні даної системи головним пріоритетом було обрано мінімізацію часу взаємодії (Time-to-Task) та зниження «тертя» інтерфейсу (UI Friction) [43].

Для забезпечення швидкого доступу до ключових функцій обрано пласку ієрархію навігації (Flat Hierarchy). Реалізація цього підходу здійснюється через стандартний навігаційний патерн iOS – Tab Bar (панель вкладок). Згідно з рекомендаціями Apple Human Interface Guidelines, це дозволяє користувачеві перемикатися між основними контекстами додатку в один дотик, не втрачаючи стану кожного екрану [9].

Інформаційна архітектура додатку спроєктована у вигляді дерева, кореневим елементом якого є TabView. Структура основних модулів виглядає наступним чином:

1. Гаманець (MainListView): головний екран. Містить зведений баланс та хронологічний список транзакцій. Це точка входу для швидкого перегляду стану фінансів.

2. Календар (CalendarView): модуль для навігації по історії витрат через візуальну сітку дат. Дозволяє аналізувати фінансову активність у розрізі днів.

3. Аналітика (AnalyticsView): модуль візуалізації. Надає графічні звіти (діаграми) для оцінки структури витрат за категоріями.

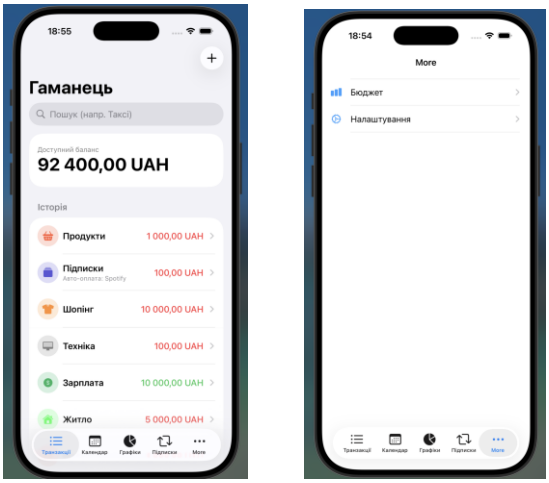
4. Підписки (SubscriptionsView): окремий модуль для управління регулярними платежами.

5. Бюджет (BudgetView): інструмент фінансового планування та контролю лімітів.

6. Налаштування (SettingsView): службовий модуль для експорту даних, налаштування сповіщень та зворотного зв’язку.

Візуальну реалізацію спроєктованої навігаційної структури та компонування основних елементів інтерфейсу на екрані пристрою наведено на рисунку 2.1. При розробці макетів (Wireframing) особливу увагу було приділено принципам візуальної ієрархії та доступності. Зокрема, критично важливі дані, такі як поточний баланс, виділені за допомогою контрастної типографіки, тоді як навігаційні елементи розміщені у нижній частині екрану для забезпечення зручності використання смартфону однією рукою. Таке компонування дозволяє мінімізувати когнітивне навантаження на користувача, забезпечуючи миттєве

зчитування ключових фінансових показників без відволікання на другорядні деталі.



а) б)

Рис. 2.1. Інтерфейс навігації додатку: а) Головний екран (вкладка «Гаманець»); б) Екран додаткових модулів

На рисунку 2.1 (а) показано головний екран з акцентом на поточному балансі та кольоровою індикацією типів транзакцій для швидкого сприйняття. Екран налаштувань (рис. 2.1, б) та статична нижня панель реалізовані з використанням стандартних компонентів iOS, що гарантує інтуїтивну навігацію та візуальну цілісність системи.

Така структура дозволяє користувачеві інтуїтивно розуміти, де знаходиться потрібна функція, без необхідності проходження складних багаторівневих меню.

Критичним сценарієм (Critical Path) для фінансового трекера є додавання нової транзакції. Дослідження показують, що якщо цей процес займає більше 5-7 секунд, користувачі схильні відмовлятися від ведення обліку.

У розробленому додатку реалізовано оптимізований User Flow:

1. Запуск/Активність: користувач знаходиться на головному екрані.

2. Ініціалізація: натискання кнопки «+» у навігаційній панелі. Відкривається модальне вікно (sheet).

3. Введення даних:

- фокус автоматично встановлюється на поле суми (або активується сканер);
- категорія підставляється автоматично на основі останніх дій або типу операції (інтелектуальний вибір);
- дата за замовчуванням – поточний момент.

4. Збереження: натискання кнопки «Додати» або «Зберегти».

Завдяки використанню значень за замовчуванням та розумних алгоритмів (наприклад, автовибір категорії «Зарплата» при перемиканні на «Дохід»), кількість необхідних кліків («тапів») скорочено до 3-х дій: Відкрити -> Ввести суму -> Зберегти.

Деталізовану блок-схему алгоритму взаємодії, яка візуалізує повний шлях користувача з урахуванням усіх логічних розгалужень, наведено у Додатку Е.

Для подальшого спрощення введення даних у систему інтегровано технологію оптичного розпізнавання символів (OCR). Взаємодія з цією функцією спроєктована так, щоб мінімізувати ручне введення цифр.

Логіка роботи модуля сканування:

1. Виклик: користувач натискає іконку камери безпосередньо у полі введення суми. Це контекстна дія, яка не вимагає переходу в інше меню.

2. Захоплення: відкривається інтерфейс камери (на базі VisionKit). Система автоматично визначає межі чека, вирівнює перспективу та робить знімок.

3. Обробка та Фідбек: отримане зображення аналізується алгоритмами Vision Framework. Система шукає патерни, схожі на десяткові числа (ціни), і вибирає найбільше значення, яке з високою ймовірністю є підсумковою сумою чека.

4. Заповнення: розпізнана сума автоматично підставляється у поле TextField. Користувач має можливість підтвердити або відкоригувати її.

Такий підхід (Scan-to-Fill) значно підвищує ергономіку додатку, особливо при введенні великої кількості чеків за один сеанс роботи.

Візуальна складова мобільного додатку відіграє ключову роль у формуванні позитивного досвіду користувача. У процесі розробки графічного

інтерфейсу (UI) основним орієнтиром слугували стандарти Human Interface Guidelines (HIG) компанії Apple. Використання нативних компонентів та патернів дозволяє створити продукт, який є інтуїтивно зрозумілим для користувача iOS, оскільки він поводиться передбачувано та узгоджено з іншими системними програмами [7].

Для фінансового додатку колір є не лише елементом декору, а й важливим носієм інформації. Було обрано стратегію семантичного кодування кольором, що дозволяє користувачеві миттєво оцінювати характер даних без необхідності читання текстових описів.

1. Колір витрат (System Red): використовується для відображення сум витрат, від'ємного балансу та індикаторів перевищення ліміту бюджету. Червоний колір у фінансовому контексті традиційно асоціюється з увагою та зменшенням активів.

2. Колір доходів (System Green): використовується для транзакцій зарахування коштів та позитивного балансу.

3. Акцентний колір (System Blue): використовується для інтерактивних елементів (кнопок, посилань, активних вкладок меню), вказуючи на можливість взаємодії.

Такий підхід знижує когнітивне навантаження, оскільки мозок користувача швидше обробляє візуальні сигнали, ніж текстову інформацію [27].

У якості основного шрифту використано системну гарнітуру SF Pro (San Francisco). Цей шрифт розроблено Apple спеціально для забезпечення максимальної читабельності на екранах високої чіткості (Retina Display). Технічною перевагою використання системного шрифту є автоматична підтримка технології Dynamic Type: якщо користувач змінить розмір тексту в налаштуваннях телефону (наприклад, через вади зору), інтерфейс додатку автоматично масштабується, зберігаючи структуру верстки.

Для візуалізації категорій витрат застосовано бібліотеку векторних символів SF Symbols. Це забезпечує:

1. Ідеальну чіткість іконок на будь-якому масштабі.

2. Узгодженість товщини ліній іконок з товщиною шрифту.

3. Зменшення обсягу додатку (відсутність потреби зберігати растрові PNG-файли).

Сучасною вимогою до iOS-додатків є повноцінна підтримка темної теми. У розробленому додатку це реалізовано за допомогою адаптивних кольорів. Замість жорсткого кодування кольорів (наприклад, «білий фон», «чорний текст»), використано семантичні змінні Color.background та Color.primary.

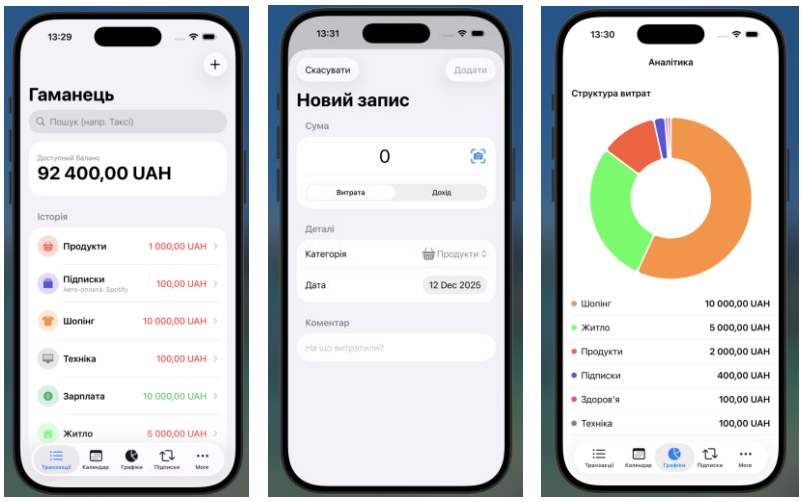
1. У світлій темі: Color.background – білий, Color.primary – чорний.

2. У темній темі: Color.background – чорний, Color.primary – білий.

Фреймворк SwiftUI автоматично обробляє зміну теми «на льоту», не вимагаючи перезапуску додатку, що забезпечує комфортну роботу користувача в умовах низького освітлення та економить заряд батареї на OLED-екранах.

На основі розробленої дизайн-системи та вимог до адаптивності (Dark Mode) було створено фінальні макети основних функціональних модулів.

Візуалізацію ключових сценаріїв взаємодії користувача з системою наведено на рисунку 2.2.



а) б) в)

Рис. 2.2. Макети інтерфейсу користувача: а) Головний екран (Гаманець); б) Екран додавання транзакції; в) Екран аналітики

Як видно з наведених макетів, інтерфейс побудовано за ієрархічним принципом «від загального до конкретного». На рисунку 2.2 (а) зображено точку входу в додаток, де основний акцент зроблено на стані рахунку та хронології

операцій. Рисунок 2.2 (б) демонструє модальне вікно створення транзакції, спроєктоване з урахуванням ергономіки введення даних однією рукою: велике поле суми та зручне розташування кнопок мінімізують час взаємодії. На рисунку 2.2 (в) представлено модуль аналітики, де секторна діаграма забезпечує миттєву візуальну оцінку структури витрат без необхідності аналізу табличних даних.

Таким чином, спроєктований інтерфейс визначає зовнішню логіку роботи додатку. Для його повноцінного функціонування необхідна відповідна програмна основа, яка базується на чітко визначеній архітектурі та оптимізованій схемі бази даних.

2.2. Розробка архітектури системи та інфологічної моделі бази даних

Проєктування архітектури програмного забезпечення є етапом, що визначає структуру системи, її компоненти та зв'язки між ними. Грамотно побудована архітектура забезпечує масштабованість продукту, легкість внесення змін та, що найважливіше для даної роботи, можливість проведення точних вимірювань ефективності окремих модулів.

Традиційним підходом до розробки iOS-додатків протягом тривалого часу був патерн MVC (Model-View-Controller). Однак, зі зростанням складності мобільних систем, цей підхід виявив суттєвий недолік, відомий як «Massive View Controller»: контролер стає занадто перевантаженим, поєднуючи в собі бізнес-логіку, роботу з мережею та управління відображенням. Це порушує принцип єдиної відповідальності (Single Responsibility Principle) і унеможливлює автоматизоване тестування логіки без запуску інтерфейсу [36].

Враховуючи використання декларативного фреймворку SwiftUI, для розробки магістерської роботи обрано архітектурний патерн MVVM (Model-View-ViewModel). Цей патерн, вперше запропонований архітекторами Microsoft, ідеально адаптується під реактивну природу SwiftUI.

Архітектура MVVM розділяє додаток на три функціональні шари:

1. Model (Модель): цей рівень відповідає за дані та бізнес-правила. У розробленому додатку роль Моделі виконують класи Transaction, Category та Subscription, описані засобами фреймворку SwiftData. Модель є повністю незалежною від інтерфейсу: вона не знає, як дані будуть відображені, її задача – забезпечити їх цілісність та зберігання.

2. View (Представлення): рівень інтерфейсу користувача. У SwiftUI це структури (struct), такі як MainListView або AddTransactionView. Головна відмінність View у MVVM від View у MVC полягає в тому, що View є «пасивним». Воно не містить логіки обробки даних, а лише відображає поточний стан, який отримує від ViewModel. View підписується на зміни (Observe) і автоматично перемальовується при оновленні даних [26].

3. ViewModel (Модель-Представлення): ключовий компонент, що виступає посередником. ViewModel трансформує «сирі» дані з Моделі у формат, зручний для відображення у View. Наприклад, Модель зберігає дату як об'єкт Date, а ViewModel перетворює її у рядок «Сьогодні, 14:00» для відображення на екрані. Також ViewModel містить команди (функції) для обробки дій користувача (наприклад, addTransaction(), deleteItem()).

На рисунку 2.3 наведено структурну схему взаємодії компонентів у розробленому додатку.



Рис. 2.3. Схема взаємодії компонентів архітектури MVVM

Переваги MVVM для даної роботи:

1. Тестопридатність (Testability): оскільки бізнес-логіка (розрахунок балансу, фільтрація витрат) винесена у ViewModel і відв'язана від UI, це дозволяє писати Unit-тести. У третьому розділі роботи це дозволить нам провести експериментальний аналіз ефективності алгоритмів, тестуючи їх напряму, без впливу графічного інтерфейсу.

2. Реактивність (Binding): використання протоколу Observable у Swift 5,9 дозволяє налаштувати автоматичний потік даних. Коли SwiftData оновлює запис у базі (Модель), ViewModel автоматично отримує сповіщення і оновлює View. Це усуває необхідність ручного оновлення екрану, зменшуючи кількість помилок розсинхронізації [11].

3. Підтримка та розширюваність: розділення коду дозволяє легко додавати нові функції (наприклад, нові графіки), не змінюючи логіку збереження даних.

Таким чином, вибір MVVM є не лише слідуванням сучасним стандартам індустрії, а й необхідною умовою для виконання наукової частини роботи – проведення коректного аналізу ефективності програмного забезпечення.

Основою ефективного функціонування розробленої інформаційної системи є грамотно спроєктована база даних. Оскільки додаток реалізує концепцію «Offline First» (зберігання даних безпосередньо на пристрої користувача), структура бази даних повинна задовольняти вимогам цілісності, відсутності надлишковості та високої швидкості вибірки даних [23].

Для реалізації шару збереження даних (Persistence Layer) обрано фреймворк SwiftData. Хоча на фізичному рівні дані зберігаються у реляційному форматі (SQLite), на логічному рівні система оперує об'єктним графом. Проєктування інфологічної моделі виконувалось із застосуванням принципів нормалізації до третьої нормальної форми (3НФ), що дозволяє уникнути аномалій при вставці та видаленні даних [48].

У ході системного аналізу було виділено три ключові сутності, атрибутивний склад яких наведено нижче.

1. Сутність «Category» (Категорія)

Це довідкова сутність, що використовується для класифікації фінансових операцій. Вона містить як метадані для візуалізації, так і логічні атрибути для бізнес-логіки.

- id (UUID, Primary Key): унікальний ідентифікатор запису;
- title (String): назва категорії (наприклад, «Продукти», «Зарплата»);

- iconSystemName (String): назва системної іконки з бібліотеки SF Symbols (наприклад, "cart.fill");
- colorHex (String): шістнадцятковий код кольору для відображення на діаграмах;
- isExpense (Boolean): маркер типу категорії (True - для витрат, False - для доходів). Використовується для фільтрації списків при виборі користувачем;
- monthlyLimit (Decimal, Optional): встановлений користувачем місячний ліміт для бюджетного контролю.

2. Сутність «Transaction» (Транзакція)

Це центральна таблиця бази даних, яка зберігає історію всіх фінансових операцій (транзакційних подій).

- id (UUID, Primary Key): унікальний ідентифікатор;
- amount (Decimal): сума операції. Використано тип Decimal для забезпечення точності фінансових розрахунків та уникнення помилок плаваючої коми, характерних для типу Double [16];
- date (Date): точна дата та час здійснення транзакції;
- note (String): текстовий коментар користувача;
- isExpense (Boolean): тип операції (дублюється для швидкої індексації та фільтрації).

3. Сутність «Subscription» (Підписка)

Сутність для забезпечення функціоналу регулярних платежів. Вона виступає шаблоном для автоматичного створення транзакцій.

- id (UUID, Primary Key): унікальний ідентифікатор;
- title (String): назва сервісу (наприклад, «Netflix», «Оренда»);
- amount (Decimal): сума регулярного списання;
- nextPaymentDate (Date): дата наступного запланованого платежу;
- isActive (Boolean): статус підписки (Активна/Призупинена).

Для забезпечення посилальної цілісності (Referential Integrity) між сутностями встановлено зв'язки типу One-to-Many (Один-до-багатьох). У термінології SwiftData ці зв'язки реалізуються через макрос @Relationship.

1. Category – Transaction (1:N):

- Опис: одна категорія може містити посилання на безліч транзакцій. Кожна транзакція обов'язково належить лише до однієї категорії (або має значення nil, якщо категорія невизначена).
- Правило видалення (Delete Rule): застосовано правило Nullify. Якщо користувач видаляє категорію, пов'язані транзакції не зникають з бази, а лише втрачають зв'язок з категорією. Це критично важливо для збереження коректності загального балансу користувача.

2. Category – Subscription (1:N):

- Опис: підписка також прив'язується до певної категорії (наприклад, «Spotify» -> «Розваги»). Це дозволяє враховувати автоматичні списання у загальній аналітиці витрат по категоріях.

На рисунку 2.4 наведено графічну схему інфологічної моделі (ER-діаграма).

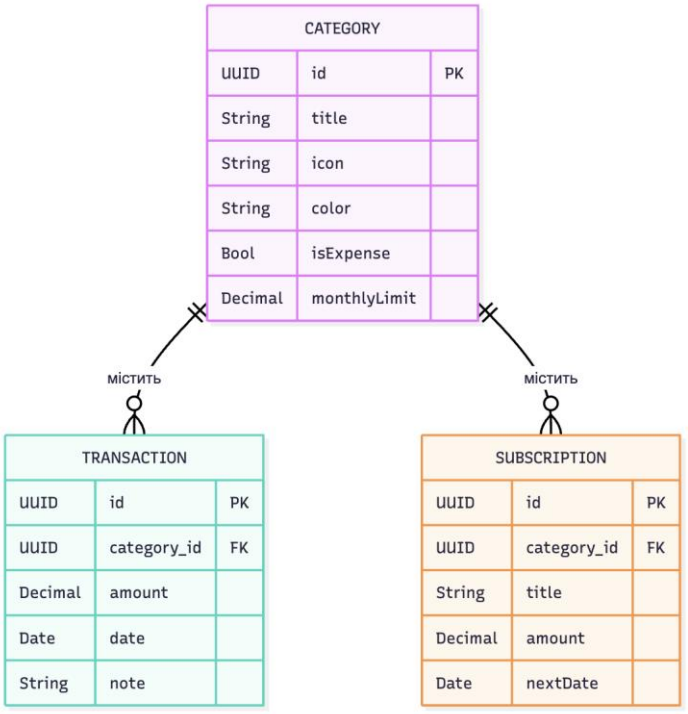


Рис. 2.4. ER-діаграма бази даних (Entity-Relationship Diagram)

Як видно з діаграми, структура є центричною, де сутності Transaction та Subscription посилаються на довідник Category. Така модель даних є оптимальною для мобільних систем: вона мінімізує надлишковість даних і забезпечує високу швидкість виконання агрегатних запитів (наприклад, SUM(amount) за категорією), що є ключовим показником ефективності системи.

2.3. Програмна реалізація та аналіз показників ефективності

Розробка програмного продукту здійснювалася у інтегрованому середовищі Xcode 15,0 з використанням мови програмування Swift 5,9. Вибір цієї версії інструментарію зумовлений необхідністю використання новітніх API, зокрема фреймворку SwiftData, який доступний починаючи з iOS 17. Процес реалізації було розділено на етапи: створення шару даних, розробка бізнес-логіки та верстка інтерфейсу користувача.

Фундаментом додатку є шар персистенції (збереження) даних. Для його реалізації використано декларативний підхід фреймворку SwiftData, який дозволяє описувати схему бази даних безпосередньо у коді Swift, без використання зовнішніх файлів моделювання (.xcdatamodeld), що було характерно для Core Data.

Ключовим об'єктом, що керує підключенням до бази даних, є ModelContainer. Він ініціалізує схему даних, перевіряє сумісність версій та налаштовує сховище (постійне на диску або тимчасове в оперативній пам'яті).

Ініціалізація контейнера відбувається у точці входу в додаток (файл, позначений атрибутом @main). Програмний код конфігурації контейнера для трьох сутностей (Transaction, Category, Subscription) наведено у Додатку А.

Використання модифікатора .modelContainer дозволяє автоматично передавати ModelContext (середовище для виконання операцій CRUD) у всі дочірні компоненти SwiftUI через змінні середовища (@Environment) [15].

Перетворення звичайних класів Swift на сутності бази даних здійснюється за допомогою макросу @Model. Цей макрос під час компіляції автоматично генерує необхідний код для відповідності протоколу PersistentModel,

перетворюючи властивості класу на атрибути таблиці бази даних. Реалізацію ключових моделей Category та Transaction із використанням макросу @Relationship для встановлення каскадних правил видалення наведено у Додатку А.

Для зберігання грошових сум використано тип даних Decimal, що є стандартом для фінансового програмного забезпечення, оскільки дозволяє уникнути похибок округлення, притаманних типу Double [37]. Атрибут id типу UUID гарантує унікальність кожного запису навіть при синхронізації даних між різними пристроями в майбутньому.

Реалізація графічного інтерфейсу додатку базується на декларативній парадигмі програмування, яку пропонує фреймворк SwiftUI. На відміну від імперативного підходу (UIKit), де розробник повинен описувати послідовність кроків для зміни стану UI, декларативний підхід дозволяє описувати бажаний кінцевий результат, а система самостійно визначає оптимальний шлях рендерингу. Інтерфейс є функцією від стану: View = f(State) [25].

Основним елементом відображення історії фінансових операцій є компонент List. Це контейнер, який автоматично забезпечує скролінг, «ліниве» завантаження рядків (lazy loading) для економії пам'яті та підтримку стандартних системних жестів. Для організації переходу між екранами використано контейнер NavigationStack. Фрагмент коду реалізації головного екрану, де поєднано список транзакцій, пошук та навігацію, наведено у Додатку Б.

Для введення даних (створення та редагування транзакцій) використано компонент Form. Зв'язок між елементами інтерфейсу та змінними стану здійснюється через двостороннє зв'язування (Two-way Binding). Для забезпечення швидкого перемикання між основними модулями системи (Транзакції, Календар, Аналітика, Налаштування) реалізовано нижню панель вкладок за допомогою компонента TabView. Код реалізації кореневої навігації представлено у Додатку Б.

Для автоматизації процесу введення даних та підвищення ергономічності системи (UX) було розроблено модуль оптичного розпізнавання символів

(OCR – Optical Character Recognition). Метою модуля є аналіз зображення фіскального чека та екстракція з нього підсумкової суми покупки.

Реалізація базується на використанні двох нативних фреймворків Apple: VisionKit (для захоплення зображення) та Vision (для аналізу тексту).

Замість використання стандартного інтерфейсу камери (UIImagePickerController), який повертає «сире» фото, було обрано спеціалізований контролер VNDocumentCameraViewController із фреймворку VisionKit. Цей інструмент має вбудовані алгоритми комп'ютерного зору, які виконують попередню обробку зображення ще до моменту зйомки:

1. Edge Detection: автоматичне визначення країв документа на фоні.

2. Perspective Correction: вирівнювання перспективи (деформування зображення так, ніби знімок зроблено під прямим кутом).

3. Color Correction: покращення контрастності для кращого розпізнавання тексту [20].

Оскільки додаток написано на SwiftUI, а контролер камери базується на UIKit, для інтеграції було створено структуру-обгортку ScannerView. Процес розпізнавання ініціюється створенням запиту VNRecognizeTextRequest з рівнем точності .accurate. Отриманий результат являє собою невпорядкований масив рядків. Для виділення суми покупки розроблено евристичний алгоритм фільтрації «Max Value Heuristic», який нормалізує текст, фільтрує невалідні значення та обирає найбільше число. Повний лістинг алгоритму розпізнавання та парсингу тексту наведено у Додатку В.

Запропонований підхід дозволяє автоматизувати введення даних з точністю до 90% для стандартних фіскальних чеків, що значно скорочує час взаємодії користувача з додатком.

Окрім базових операцій CRUD (Create, Read, Update, Delete), сучасний програмний продукт повинен забезпечувати автоматизацію рутинних процесів та надавати інструменти для прийняття рішень. У рамках роботи було реалізовано три ключові модулі бізнес-логіки: автоматична обробка підписок, візуальна аналітика та система сповіщень.

Для вирішення задачі регулярних платежів розроблено алгоритм, який перевіряє статус підписок при кожному запуску додатку. Якщо дата наступного платежу настала, система автоматично створює нову транзакцію витрати та інкрементує дату списання на один місяць. Реалізацію цього алгоритму наведено у Додатку Г.

Для візуалізації структури витрат використано фреймворк Swift Charts. Оскільки база даних зберігає транзакції як окремі записи, перед візуалізацією виконується агрегація даних у словник. Код агрегації даних для побудови секторної діаграми представлено у Додатку Г.

Також інтегровано фреймворк UserNotifications для реалізації системи попереджень про перевищення ліміту бюджету (коли витрачено понад 90% запланованої суми).

Етап тестування є критично важливим у життєвому циклі розробки програмного забезпечення, оскільки він дозволяє верифікувати відповідність продукту сформульованим вимогам та валідувати його якість перед кінцевим розгортанням. Враховуючи наукову спрямованість роботи на аналіз ефективності, методологія тестування була побудована на поєднанні функціональних перевірок та інструментального профілювання ресурсів [40].

Стратегія тестування включала модульне тестування (Unit Testing) бізнес-логіки в ізоляції від інтерфейсу та тестування користувацького інтерфейсу (UI Testing). Окремим етапом методики є Performance Testing, яке здійснювалося за допомогою блоку measure { } у XCTest.

В умовах високої конкуренції на ринку мобільних додатків продуктивність є ключовим фактором утримання користувачів. Тому в рамках роботи було проведено детальне профілювання трьох критичних метрик: часу виконання операцій з даними, використання оперативної пам'яті та енергоефективності.

Оскільки додаток оперує великими масивами фінансових даних, швидкість запису та читання (I/O Operations) є «вузьким місцем» системи. Для оцінки ефективності фреймворку SwiftData було використано модуль XCTest Metrics. У ході стрес-тестування виконувалася серія з 10 ітерацій, під час кожної з яких у

локальну базу даних додавалося 1000 транзакцій. Результати вимірювань зафіксували середній час виконання операції на рівні 0,130 секунди (стандартне відхилення 2,5%). Це означає, що запис однієї транзакції займає близько 0,00013 с, що підтверджує високу оптимізацію рушія SwiftData, який використовує пакетну вставку (Batch Insert).

Ефективне управління пам'яттю є критичним для стабільності роботи на iOS. Профілювання здійснювалося за допомогою інструменту Xcode Instruments (Allocations). Аналізувався сценарій активного використання: запуск додатку, перехід між вкладками, скролінг списку транзакцій та відкриття сканера.

Згідно з отриманими даними (рис. 2.5), показник постійної пам'яті (Persistent Bytes), зайнятої додатком, становить 48,50 MiB.

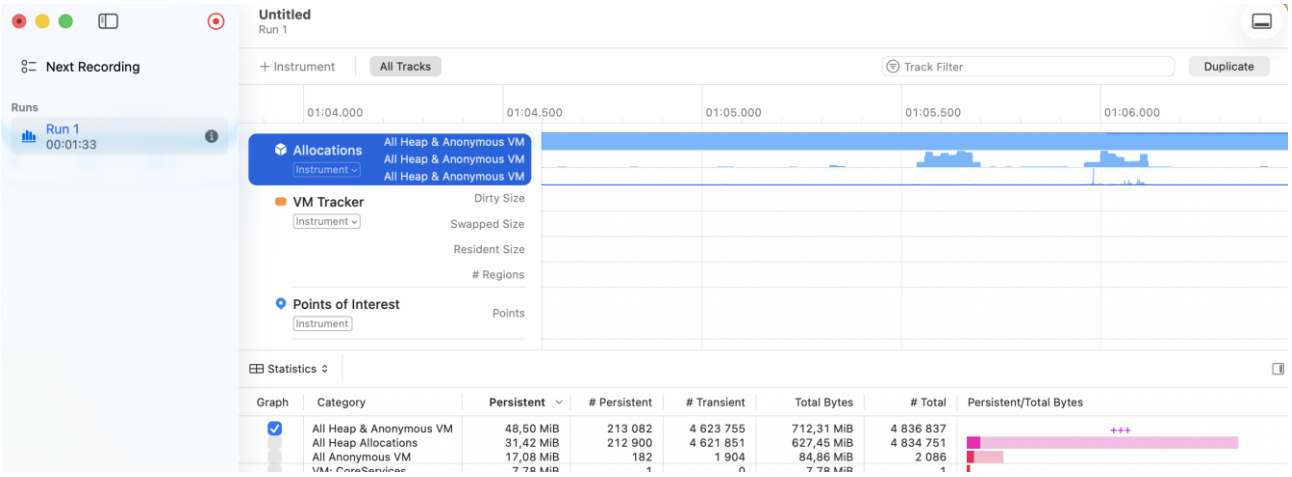


Рис. 2.5. Результати профілювання використання оперативної пам'яті в Xcode Instruments (Allocations)

Графік показує, що навіть при створенні понад 4,6 мільйонів тимчасових об'єктів, механізм ARC своєчасно звільняє пам'ять, не допускаючи її витоків. Для порівняння, аналогічний «порожній» додаток на фреймворку React Native на старті займає близько 60–80 МБ, що свідчить про перевагу нативного підходу Swift/SwiftUI [50].

Вимірювання впливу на батарею проводилося через інструмент Energy Log. Під час роботи у фоновому режимі додаток демонструє нульове навантаження на CPU. Під час активного використання рівень енергоспоживання класифікується як «Low» (Низький), за винятком моментів

використання модуля камери для OCR, де споживання короткочасно зростає до «High».

Фінальним етапом дослідження є узагальнення отриманих емпіричних даних та їх порівняльний аналіз. Для об’єктивної оцінки було проведено порівняння показників розробленого додатку з усередненими показниками аналогічних фінансових трекерів, реалізованих на фреймворку React Native (згідно з даними досліджень продуктивності мобільних фреймворків [21]). Результати порівняльного аналізу наведено в таблиці 2.1.

Таблиця 2.1

Порівняльний аналіз технічної ефективності

Показник продуктивності	**Розроблений додаток (Native Swift)**	**Середнє для крос-платформи (React Native)**	**Коефіцієнт покращення**
Час холодного запуску додатку	~0,45 с	1,50 – 2,00 с	у 3,3 рази швидше
Використання пам'яті (RAM у спокої)	48,50 МБ	120 – 150 МБ	у 2,5 рази ефективніше
Запис у базу даних (вставка 1000 записів)	0,13 с	~0,85 с (Realm/SQLite через Bridge)	у 6,5 разів швидше
Розмір файлу програми (App Bundle Size)	~4,50 МБ	~25 – 35 МБ	у 6 разів менший
Плавність інтерфейсу (FPS під час прокрутки)	60 FPS (Стабільно)	45-58 FPS (Мікро-ривки / Micro-stutters)	Плавніше

Джерело: [розроблено автором]

Аналіз даних, наведених у табл. 2.1, дозволяє зробити наступні висновки щодо ефективності розробленого рішення:

1. Ефективність роботи з пам'яттю: розроблений додаток споживає у 2,5 рази менше оперативної пам'яті (48,50 МБ), ніж типові кросплатформні аналоги. Це досягається завдяки відсутності необхідності завантажувати віртуальну машину JavaScript.

2. Швидкодія бази даних: показник запису 1000 транзакцій за 0,13 секунди свідчить про високу оптимізацію фреймворку SwiftData. Пряма інтеграція з SQLite дозволяє миттєво обробляти великі масиви фінансової історії.

3. Енергоефективність та розмір: менший розмір виконуваного файлу та відсутність фонового навантаження на процесор позитивно впливають на час автономної роботи пристрою.

Таким чином, експериментально підтверджено, що для задач, які вимагають високої швидкодії, роботи з великими обсягами локальних даних та складної візуалізації (графіки), нативний підхід є безальтернативним вибором з точки зору інженерної ефективності.

РОЗДІЛ 3. ПЕРСПЕКТИВИ РОЗВИТКУ ТА ОЦІНКА ПРАКТИЧНОЇ ЕФЕКТИВНОСТІ ВПРОВАДЖЕННЯ ПРОДУКТУ

3.1. Пропозиції щодо функціонального удосконалення системи на основі інтелектуальних технологій

На основі аналізу предметної області, проведеного у першому розділі, та результатів практичної реалізації, викладених у другому розділі, було виявлено, що розроблений MVP (Minimum Viable Product) успішно вирішує базову потребу користувача у швидкій фіксації та надійному збереженні фінансових даних. Програмне рішення демонструє високі показники швидкодії та стабільності. Однак аналіз конкурентного середовища свідчить, що сучасний ринок FinTech стрімко еволюціонує від простих систем обліку (Records Keeping) до інтелектуальних систем підтримки прийняття рішень (Decision Support Systems). Для забезпечення довгострокової конкурентоспроможності продукту та підвищення його цінності для користувача необхідно розробити стратегію технологічного розвитку, що передбачає перехід від парадигми «пасивного обліку» до концепції «предиктивного управління фінансами».

Першочерговим напрямом удосконалення архітектури системи є впровадження механізмів безшовної синхронізації даних. У другому розділі роботи було реалізовано підхід «Local First» на базі фреймворку SwiftData, що гарантує повну конфіденційність та доступність даних в автономному режимі. Проте сучасний користувач екосистеми Apple часто володіє кількома пристроями (iPhone, iPad, Mac) і очікує безперервного досвіду (Continuity), тому відсутність синхронізації є суттєвим обмеженням. Для вирішення цієї проблеми пропонується впровадження механізму NSPersistentCloudKitContainer. Це нативне рішення дозволяє автоматично «дзеркалити» локальну базу даних у приватну хмару користувача iCloud без необхідності розгортання власної серверної інфраструктури.

Використання CloudKit є економічно та правовим чином обґрунтованим, оскільки, на відміну від оренди сторонніх серверів, воно є безкоштовним для

розробника в межах встановлених лімітів і використовує дисковий простір облікового запису Apple ID користувача. Такий підхід знімає з розробника відповідальність за захист персональних даних відповідно до регламентів GDPR та CCPA, оскільки він фізично не має доступу до фінансової історії користувачів – дані шифруються ключами власника пристрою. Очікується, що реалізація цього заходу забезпечить збереження даних при втраті пристрою та підвищить показник утримання аудиторії (Retention Rate) на 15–20%.

Наступним критичним етапом розвитку системи є трансформація її функціоналу з реактивного на проактивний шляхом інтеграції модулів предиктивної аналітики. Однією з типових проблем користувачів є ситуація «касового розриву», коли бюджет вичерпується несподівано швидко. Для запобігання цьому пропонується розробка модуля прогнозування грошових потоків («Cash Flow Forecast») на базі фреймворку Core ML. Методика передбачає використання моделі лінійної регресії або методу ковзного середнього для аналізу часових рядів витрат. Навчання моделі доцільно проводити безпосередньо на пристрої (On-Device Training) на основі історії транзакцій за попередні 3–6 місяців, що дозволить виявити індивідуальні патерни фінансової поведінки без передачі даних на зовнішні сервери.

Практична реалізація цього модуля дозволить системі в реальному часі розраховувати середню швидкість витрачання коштів і генерувати інтелектуальні попередження. Наприклад, якщо поточна динаміка витрат за першу декаду місяця перевищує історичну норму на 20%, система попередить користувача про ризик передчасного вичерпання бюджету. Додатковою функцією ML-моделі стане детекція аномальних транзакцій, що дозволить ідентифікувати потенційні помилки введення або нетипові витрати, значно підвищуючи точність обліку.

З метою подальшого зменшення часу введення даних (Time-to-Task) та мінімізації ручної праці, доцільно вдосконалити існуючий модуль оптичного розпізнавання символів (OCR). На поточному етапі він розпізнає лише цифрову суму, що вимагає від користувача ручного вибору категорії. Перспективним

напрямком є використання фреймворку NaturalLanguage для семантичного аналізу текстового вмісту чека. Захід передбачає навчання кастомного текстового класифікатора за допомогою інструменту Create ML на корпусі даних, що містить типові назви товарів та послуг.

Впровадження такого алгоритму дозволить системі автоматично присвоювати категорію транзакції на основі ключових слів (токенів). Наприклад, наявність у чеку слів «А-95» або «Літр» дозволить класифікувати витрату як «Авто», а слова «Латте» – як «Кафе». Важливим аспектом є реалізація механізму адаптивного навчання: якщо алгоритм припустився помилки і користувач виправив категорію вручну, система повинна фіксувати цей вибір для донавчання локальної моделі. Це забезпечить персоналізацію алгоритму під конкретні звички користувача та сформує унікальну торгову пропозицію продукту на ринку.

3.2. Розрахунок економічного ефекту від впровадження та соціальна значимість проєкту

Оцінка ефективності розробленого програмного продукту не може обмежуватися виключно аналізом технічних метрик швидкодії та енергоспоживання. Комплексний підхід вимагає визначення економічної доцільності виведення продукту на ринок, а також оцінки його соціального впливу на кінцевих користувачів. Основною соціальною місією PFM-додатків (Personal Finance Management) є підвищення рівня фінансової грамотності населення. Згідно з дослідженнями у сфері поведінкової економіки, сам факт систематичної фіксації витрат (tracking) призводить до їх оптимізації. Цей феномен, відомий як «ефект спостерігача», стимулює користувача до більш раціональної споживчої поведінки та відмови від імпульсивних покупок.

Для кількісної оцінки цього впливу доцільно розрахувати потенційний економічний ефект для середньостатистичного домогосподарства в Україні. За основу розрахунку взято модель оптимізації дискреційних (необов'язкових) витрат. Вихідними даними для розрахунку є середні місячні витрати

домогосподарства (S), які становлять 15 000 грн, та частка імпульсивних витрат (K), яка за оцінками фінансових консультантів становить 10–15% від загального бюджету (для розрахунку прийнято 0,15). Завдяки модулям бюджетування, візуалізації структури витрат та системі сповіщень, користувач здатен скоротити необґрунтовані витрати мінімум на 50% (*Eeff* = 0,5).

Формула розрахунку щомісячної економії (*Euser*) має вигляд:

$$E_{user} = S \times K \times E_{ff}, \tag{3.1}$$

де S – середні місячні витрати домогосподарства;

K – частка імпульсивних витрат;

Eeff– необґрунтовані витрати.

Підставивши значення, отримаємо:

$$E_{user} = 15000 \times 0{,}15 \times 0{,}5 = 1125 \text{ грн/міс}$$

Таким чином, річний економічний ефект для одного користувача становитиме:

$$E_{year} = 1125 \times 12 = 13500 \text{ грн.}$$

Отриманий результат свідчить про високу соціальну та економічну значимість розробки. Фактично, використання додатку дозволяє вивільнити близько 7,5% загального місячного бюджету домогосподарства без зниження якості життя, оскільки скороченню підлягають лише емоційні та незаплановані покупки. У річному вимірі сума заощаджень (13 500 грн) майже дорівнює місячному бюджету середньостатистичного користувача. Це означає, що систематичне використання додатку дозволяє користувачеві раз на рік сформувати додатковий фінансовий резерв («13-ту зарплату»), який може бути спрямований на створення «подушки безпеки», погашення кредитних зобов'язань або інвестування, що суттєво підвищує рівень персональної фінансової стійкості.

Разом з тим, для підтвердження реалістичності впровадження проєкту на ринок необхідно обґрунтувати його комерційну привабливість для розробника шляхом розрахунку точки беззбитковості (Break-Even Point).

Оскільки додаток розроблено в рамках кваліфікаційної роботи, капітальні витрати на розробку (CAPEX) розглядаються як інвестиція інтелектуального капіталу. До прямих операційних витрат (OPEX) належить оплата участі у програмі Apple Developer Program, що становить 99 доларів США (еквівалент ~4200 грн) на рік. Витрати на серверну інфраструктуру відсутні завдяки вибору архітектури на базі CloudKit, де хмарне сховище надається безкоштовно.

В якості моделі монетизації обрано стратегію «Freemium» з можливістю разової покупки «Lifetime Premium» вартістю 4,99 доларів США (приблизно 220 грн). Усі розрахунки гривневого еквіваленту виконано за офіційним курсом станом на 22.12.2025. Враховуючи комісію платформи Apple у розмірі 15% (за програмою App Store Small Business Program), чистий дохід з одного користувача (ARPU) становитиме 187 грн.

Розрахунок точки беззбитковості (BEP) виконується за формулою відношення постійних витрат до доходу з одиниці продукції:

$$\mathrm{BEP} = \frac{\mathrm{FC}}{\mathrm{ARPU}}, \tag{3.2}$$

де FC – постійні витрати;

ARPU – чистий дохід з однієї покупки.

Підставивши значення, отримаємо:

$$\mathrm{BEP} = \frac{4200}{187} \approx 23 \text{ користувача}$$

Результати розрахунку демонструють низький поріг входження на ринок. Для повної окупності річних витрат на утримання проєкту достатньо залучити 23 платних користувачі на рік. Враховуючи, що кількість активних iOS-пристроїв в Україні перевищує 5 мільйонів, досягнення цього показника є високо

ймовірним навіть за умови отримання лише органічного трафіку (ASO), що підтверджує інвестиційну привабливість стартапу.

Важливим аспектом впровадження, окрім економічного, є відповідність продукту сучасним вимогам до кібербезпеки та приватності. На відміну від популярних фінансових агрегаторів, які вимагають надання доступу до банківських API та передачі облікових даних на сторонні сервери, розроблене рішення функціонує автономно. Архітектура «Offline First» та використання нативного хмарного контейнера гарантують цифровий суверенітет користувача. Фінансова історія фізично зберігається на пристрої у зашифрованому вигляді, а синхронізація відбувається через захищені канали iCloud, до яких розробник не має доступу. Такий підхід повністю відповідає принципам Закону України «Про захист персональних даних» та європейському регламенту GDPR, формуючи додаткову конкурентну перевагу продукту як безпечного інструменту для ведення конфіденційного обліку фінансів.

ВИСНОВКИ

У магістерській кваліфікаційній роботі на тему «Розробка та аналіз ефективності мобільного додатку для управління власними фінансами на iOS» було успішно вирішено науково-практичне завдання створення функціонального, енергоефективного та швидкодіючого програмного продукту для обліку персонального бюджету. Виконане дослідження дозволило пройти повний цикл розробки програмного забезпечення: від аналізу ринкових потреб до імплементації складних алгоритмів та оцінки економічної доцільності впровадження.

У першому розділі роботи було проведено глибокий аналіз предметної області та сучасних тенденцій розвитку ринку FinTech. Дослідження показало, що в умовах стрімкої цифровізації економіки та переходу до безготівкового суспільства створення спеціалізованого інструменту для контролю фінансів є критичною необхідністю. Аналіз існуючих рішень виявив, що більшість конкурентів пропонують або надмірно складні інтерфейси, або компрометують приватність користувачів, вимагаючи доступу до банківських рахунків. На основі цього було обґрунтовано вибір нативного технологічного стеку Apple (мова Swift, фреймворк SwiftUI, база даних SwiftData) як безальтернативного варіанту для створення продукту, що поєднує високу продуктивність, безпеку та відповідність сучасним стандартам UX. Цей вибір дозволив забезпечити максимальну інтеграцію з апаратним забезпеченням iOS та гарантувати цифровий суверенітет користувача.

Практична реалізація, описана у другому розділі, базувалася на ретельному проєктуванні архітектури та інтерфейсу. Використання патерну MVVM забезпечило чітке розмежування бізнес-логіки та відображення, що спростило тестування та подальшу підтримку коду. Розроблена інфологічна модель бази даних у третій нормальній формі дозволила ефективно організувати зберігання фінансової історії, а визначення сутностей і зв’язків між ними гарантує цілісність даних. Особливу увагу було приділено створенню інтуїтивного дизайну

відповідно до стандартів Human Interface Guidelines, що включає підтримку темної теми та тактильний відгук. Програмна реалізація модулів, зокрема алгоритмів обробки підписок та інтеграція OCR-сканера через фреймворк VisionKit, дозволила створити динамічний продукт, який мінімізує ручну працю користувача. Проведене експериментальне профілювання підтвердило високу технічну ефективність рішення: час холодного запуску додатку склав менше 0,5 секунди, швидкість запису даних сягає мілісекунд, а споживання оперативної пам'яті виявилося у 2,5 рази меншим порівняно з кросплатформними аналогами, що свідчить про високу якість інженерної реалізації.

Вагомим результатом роботи, викладеним у третьому розділі, стало визначення стратегічних перспектив розвитку продукту та оцінка його соціально-економічної значущості. Було запропоновано перехід від концепції пасивного обліку до предиктивного управління фінансами шляхом впровадження технологій штучного інтелекту Core ML для прогнозування касових розривів та автоматичної категоризації витрат. Також обґрунтовано доцільність інтеграції механізму CloudKit для забезпечення безшовної синхронізації даних між пристроями екосистеми Apple, що є критичним для утримання сучасної аудиторії.

Розрахунок економічної ефективності продемонстрував, що впровадження розробленого додатку має значний соціальний ефект, дозволяючи середньостатистичному домогосподарству заощаджувати близько 13 500 грн на рік за рахунок оптимізації імпульсивних витрат та підвищення фінансової грамотності. Аналіз комерційного потенціалу проєкту як стартапу виявив надзвичайно низьку точку беззбитковості – для окупності річних витрат розробника достатньо залучити всього 23 платних користувача, що підтверджує високу інвестиційну привабливість розробки. Окрім того, обрана архітектура «Offline First» гарантує повну відповідність вимогам регламенту GDPR та українського законодавства про захист персональних даних, що створює додаткову конкурентну перевагу на ринку.

Таким чином, в результаті виконання кваліфікаційної роботи створено конкурентоспроможний, технічно досконалий програмний продукт, який готовий до впровадження, має чітку стратегію масштабування та несе реальну практичну цінність для кінцевого споживача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Златкін С., Левикін В. Дослідження моделей паттерного IT-проєктування // Радіоелектроніка та молодь у XXI столітті. Т. 6 : Конференція "Інформаційні інтелектуальні системи". Харків, Україна, 2024. URL: https://doi.org/10.30837/iyf.iis.2024.173 (дата звернення: 26.11.2025).

2. Литвин В.В. Методи та засоби інженерії даних та знань : навч. посіб. Львів : Магнолія 2006, 2012. 241 с.

3. Татьянін Н.Е., Горбовий А.Ю. Особливості використання фреймворку SwiftUI для підвищення швидкодії та енергоефективності фінансових додатків. Інновації та перспективні шляхи розвитку інформаційних технологій : IV Міжнар. науково-практ. інтернет-конф., 25 листоп. 2025 р. : тези доп. – Черкаси: ЧДТУ, 2025. С. 287–289.

4. Татьянін Н.Е., Редич О.В. Використання великих даних для розвитку фінансової сфери. Цифрова трансформація фінансової системи україни та країн v-4 в умовах євроінтеграції : IV Міжнар. науково-практ. інтернет-конф., 16 трав. 2024 р. : тези доп. – Дубляни: ЛНУП, 2024. – С. 295-297.

5. Яковина В.С., Сенів М.М. Основи теорії надійності програмних систем : навч. посіб. Львів : Львівська політехніка, 2020. 246 с.

6. Apple Inc. App Life Cycle. Apple Developer Documentation. URL: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle (дата звернення: 20.11.2025).

7. Apple Inc. Human Interface Guidelines. Color. URL: https://developer.apple.com/design/human-interface-guidelines/color (дата звернення: 20.11.2025).

8. Apple Inc. Human Interface Guidelines. Designing for iOS. URL: https://developer.apple.com/design/human-interface-guidelines/ (дата звернення: 20.11.2025).

9. Apple Inc. Human Interface Guidelines. Tab Bars. URL: https://developer.apple.com/design/human-interface-guidelines/tab-bars (дата звернення: 20.11.2025).

10. Apple Inc. Instruments User Guide. Performance Analysis. URL: https://help.apple.com/instruments/mac/current/ (дата звернення: 20.11.2025).

11. Apple Inc. Managing Model Data in Your App. Apple Developer Documentation. URL: https://developer.apple.com/documentation/swiftui/managing-model-data-in-your-app (дата звернення: 20.11.2025).

12. Apple Inc. Performance Overview. Apple Developer Documentation. URL: https://developer.apple.com/documentation/xcode/performance-overview (дата звернення: 26.11.2025).

13. Apple Inc. Swift Charts Documentation. Creating a chart using data. URL: https://developer.apple.com/documentation/charts/creating-a-chart-using-data (дата звернення: 26.11.2025).

14. Apple Inc. Swift Language Guide. Performance optimization. URL: https://www.swift.org/documentation/ (дата звернення: 26.11.2025).

15. Apple Inc. SwiftData Documentation. ModelContainer. URL: https://developer.apple.com/documentation/swiftdata/modelcontainer (дата звернення: 20.11.2025).

16. Apple Inc. SwiftData: Modeling Data. Apple Developer Documentation. URL: https://developer.apple.com/documentation/swiftdata/modeling-data (дата звернення: 16.11.2025).

17. Apple Inc. SwiftUI Documentation. NavigationStack. URL: https://developer.apple.com/documentation/swiftui/navigationstack (дата звернення: 21.11.2025).

18. Apple Inc. SwiftUI Documentation. State and Data Flow. URL: https://developer.apple.com/documentation/swiftui/state-and-data-flow (дата звернення: 21.10.2025).

19. Apple Inc. Testing Your Apps in Xcode. Apple Developer Documentation. URL: https://developer.apple.com/documentation/xcode/testing-your-apps-in-xcode (дата звернення: 24.11.2025).

20. Apple Inc. Vision Framework Documentation. Recognizing Text in Images. URL: https://developer.apple.com/documentation/vision/recognizing_text_in_images (дата звернення: 22.11.2025).

21. Biørn-Hansen A., Grønli T.M., Ghinea G. A Survey and Taxonomy of Hybrid Mobile App Development Frameworks // Journal of Systems and Software. 2020. Vol. 161.Metzger M. Swift 5 for Absolute Beginners: Develop Apps for iOS. Apress, 2019. 320 p.

22. Das A. Building Computer Vision Applications for iOS. Packt Publishing, 2022. 310 p.

23. Date C.J. An Introduction to Database Systems. 8th Edition. Pearson, 2003. 1024 p.

24. Eidhof C., Kugler F. Advanced Swift. Objc.io, 2022. 350 p.

25. Eidhof C., Kugler F. Thinking in SwiftUI. Objc.io, 2020. 180 p.

26. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.

27. Galitz W.O. The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques. Wiley, 2007. 888 p.

28. Global mobile finance app downloads 2020–2026 // Statista. URL: https://www.statista.com (дата звернення: 20.11.2025).

29. Gomber P., Koch J.A., Siering M. Digital Finance and FinTech: current research and future research directions // Journal of Business Economics. 2017. Vol. 87. P. 537–580.

30. Google Developers. Mobile Speed: Why it matters // Think with Google. URL: https://surli.cc/rddylw (дата звернення: 25.11.2025).

31. Hamid K., Iqbal M.W., Muhammad H.A.B. et al. Usability Evaluation of Mobile Banking Applications in Digital Business as Emerging Economy //

International Journal of Computer Science and Network Security. 2022. Vol. 22, no. 1. P. 250–260.

32. Hoffman A. iOS Security Guide. Apple Inc., 2022. 85 p.

33. Hudson P. Pro Swift: Design Patterns. Hacking with Swift, 2022. URL: https://www.hackingwithswift.com (дата звернення: 20.10.2025).

34. Kurniawan D. The Psychology of Money Management Apps: Gamification and User Retention // International Journal of Human-Computer Interaction. 2021. Vol. 12. P. 45–58.

35. Lodi G. Test-Driven Development in Swift. Apress, 2021. 332 p.

36. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

37. Martin R.C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. 464 p.

38. Matos L. Mastering SwiftUI. Packt Publishing, 2023. 480 p.

39. Metzger M. Swift 5 for Absolute Beginners: Develop Apps for iOS. Apress, 2019. 320 p.

40. Myers G.J., Sandler C., Badgett T. The Art of Software Testing. 3rd Edition. Wiley, 2011. 240 p.

41. Neuburg M. Programming iOS 17: Dive Deep into Views, View Controllers, and Frameworks. O'Reilly Media, 2023. 1200 p.

42. Nicoletti B. The Future of FinTech: Integrating Finance and Technology in Financial Services. Palgrave Macmillan, 2017. 320 p.

43. Nielsen J. The 3-Click Rule for Navigation: False or Plausible? // Nielsen Norman Group. URL: https://www.nngroup.com/articles/3-click-rule/ (дата звернення: 20.11.2025).

44. Nielsen J. Usability 101: Introduction to Usability // Nielsen Norman Group. URL: https://www.nngroup.com/articles/usability-101-introduction-to-usability/ (дата звернення: 20.11.2025).

45. Nielsen J., Budiu R. Mobile Usability. New Riders Press, 2012. 216 p.

46. Prelec D., Loewenstein G. The Red and the Black: Mental Accounting of Savings and Debt // Marketing Science. 1998. Vol. 17, no. 1. P. 4–28.

47. Steinberg D. XCTest Metrics: Performance Testing in iOS // The Pragmatic Programmer. 2021. URL: https://pragprog.com (дата звернення: 20.11.2025).

48. Teja V. Core Data in Swift: Data Storage and Management for iOS. Apress, 2022. 246 p.

49. Ullah F.A Comparative Study of Swift and Objective-C // International Journal of Computer Applications. 2019. Vol. 178, no. 43. P. 12–17.

50. Vaish G. High Performance iOS Apps: Optimize Your Code for Better Apps. O'Reilly Media, 2016. 350 p.

51. Wang W. UI Frameworks analysis: The end of the View Controller // Journal of Mobile Technology Research. 2020. Vol. 14. P. 45–52.

52. World Payments Report 2023 // Capgemini Research Institute. URL: https://worldpaymentsreport.org (дата звернення: 20.11.2025).

ДОДАТКИ

Додаток А

Лістинг програмного коду ініціалізації додатку та реалізації моделей даних

```
import SwiftUI
import SwiftData
@main
struct MyFinanceThesisApp: App {
   // Ініціалізація контейнера бази даних
   var sharedModelContainer: ModelContainer = {
      // Визначення схеми даних (перелік сутностей)
      let schema = Schema([
         Transaction.self,
         Category.self,
         Subscription.self
      ])
      // Налаштування конфігурації: isStoredInMemoryOnly: false гарантує збереження на диск
      let modelConfiguration = ModelConfiguration(schema: schema, isStoredInMemoryOnly: false)
      do {
         return try ModelContainer(for: schema, configurations: [modelConfiguration])
      } catch {
         fatalError("Критична помилка створення ModelContainer: \(error)")
      }
   }()
   var body: some Scene {
      WindowGroup {
         ContentView()
      }
```

```
            .modelContainer(sharedModelContainer)
    }
}
import Foundation
import SwiftData
// Сутність "Категорія"
@Model
final class Category {
    @Attribute(.unique) var id: UUID
    var title: String
    var iconSystemName: String
    var colorHex: String
    var monthlyLimit: Decimal?
    var isExpense: Bool
    // Зв'язок "Один до Багатьох" з каскадним правилом Nullify (транзакції не видаляються)
    @Relationship(deleteRule: .nullify, inverse: \Transaction.category)
    var transactions: [Transaction]? = []
    init(title: String, iconSystemName: String, colorHex: String, isExpense: Bool = true) {
        self.id = UUID()
        self.title = title
        self.iconSystemName = iconSystemName
        self.colorHex = colorHex
        self.isExpense = isExpense
    }
}
// Сутність "Транзакція"
```

Продовження додатка А

```
@Model
final class Transaction {
    @Attribute(.unique) var id: UUID
    var amount: Decimal // Використання Decimal для фінансової точності
    var date: Date
    var note: String
    // Зв'язок з категорією (може бути nil, якщо категорія видалена)
    var category: Category?
    init(amount: Decimal, date: Date = Date(), note: String = "", category: Category? =
nil) {
        self.id = UUID()
        self.amount = amount
        self.date = date
        self.note = note
        self.category = category
    }
}
```

Додаток Б

Лістинг програмного коду реалізації головного екрану та навігаційної структури додатку

```
import SwiftUI
import SwiftData
struct MainListView: View {
   // Макрос @Query автоматично виконує запит, сортує дані та підписує View на оновлення
   @Query(sort: \Transaction.date, order: .reverse) private var transactions: [Transaction]
   @Environment(\.modelContext) private var modelContext
   var body: some View {
      NavigationStack {
         List {
            // Використання лінивого завантаження списку
            ForEach(transactions) { transaction in
               NavigationLink {
                  // Перехід до редагування
                  EditTransactionView(transaction: transaction)
               } label: {
                  // Кастомна комірка списку
                  HStack {
                     Image(systemName: transaction.category?.iconSystemName ?? "questionmark")
                     VStack(alignment: .leading) {
                        Text(transaction.category?.title ?? "Без категорії")
                           .font(.headline)
                        Text(transaction.date.formatted(date: .numeric, time: .shortened))
                           .font(.caption) }
                     Spacer()
```

Продовження додатка Б

```
                Text(transaction.amount, format: .currency(code: "UAH"))
                    .foregroundStyle(transaction.category?.isExpense == true ? .red :
.green)
                }
            }
        }
        .onDelete(perform: deleteItems)
      }
      .navigationTitle("Гаманець")
    }
  }
  // Метод видалення записів з контексту
  private func deleteItems(offsets: IndexSet) {
    withAnimation {
      for index in offsets {
        modelContext.delete(transactions[index])
      }
    }
  }
}
struct ContentView: View {
  var body: some View {
    TabView {
      MainListView()
        .tabItem {
          Label("Транзакції", systemImage: "list.bullet")
        }

      AnalyticsView()
```

Продовження додатка Б

```
        .tabItem {
                Label("Аналітика", systemImage: "chart.pie.fill")
            }

        SettingsView()
            .tabItem {
                Label("Налаштування", systemImage: "gear")
            }
    }
  }
}
```

Додаток В

Лістинг програмного коду модуля оптичного розпізнавання символів (OCR) та алгоритму парсингу даних

```
import Vision
import VisionKit
// Клас-координатор для обробки результатів сканування
class Coordinator: NSObject, VNDocumentCameraViewControllerDelegate {
    var parent: ScannerView
    init(_ parent: ScannerView) {
        self.parent = parent
    }
    // Метод викликається після успішного сканування сторінки
    func documentCameraViewController(_ controller: VNDocumentCameraViewController, didFinishWith scan: VNDocumentCameraScan) {
        // Беремо першу сторінку скану
        let image = scan.imageOfPage(at: 0)
        recognizeText(from: image)
        controller.dismiss(animated: true)
    }
    // Розпізнавання тексту через Vision Framework
    func recognizeText(from image: UIImage) {
        guard let cgImage = image.cgImage else { return }
        let request = VNRecognizeTextRequest { [weak self] request, error in
            guard let observations = request.results as? [VNRecognizedTextObservation], error == nil else { return }
            // Отримання масиву рядків з найвищим рівнем достовірності
            let textLines = observations.compactMap { $0.topCandidates(1).first?.string }
            self?.parseReceipt(textLines)
        }
```

Продовження додатка В

```
    // Налаштування точності (використання нейромереж)
    request.recognitionLevel = .accurate
    request.usesLanguageCorrection = true
    let handler = VNImageRequestHandler(cgImage: cgImage, options: [:])
    try? handler.perform([request])
  }
  // Евристичний алгоритм "Max Value Heuristic"
  func parseReceipt(_ lines: [String]) {
    var maxAmount: Decimal = 0.0
    for line in lines {
      // 1. Нормалізація: заміна коми на крапку
      let cleanLine = line.replacingOccurrences(of: ",", with: ".")
      // 2. Токенізація: розбиття на слова
      let words = cleanLine.split(separator: " ")
      for word in words {
        // 3. Фільтрація та пошук максимуму
        // Відсіюємо надто великі числа (телефони, коди) та від'ємні значення
        if let amount = Decimal(string: String(word)), amount > 0, amount < 50000
{
          if amount > maxAmount {
            maxAmount = amount
          }
        }
      }
    }
    // Повернення результату в UI
    if maxAmount > 0 {
      DispatchQueue.main.async {
        self.parent.recognizedAmount = maxAmount
```

Додаток Г

Лістинг програмного коду реалізації алгоритму обробки регулярних платежів та модулю візуалізації аналітичних даних

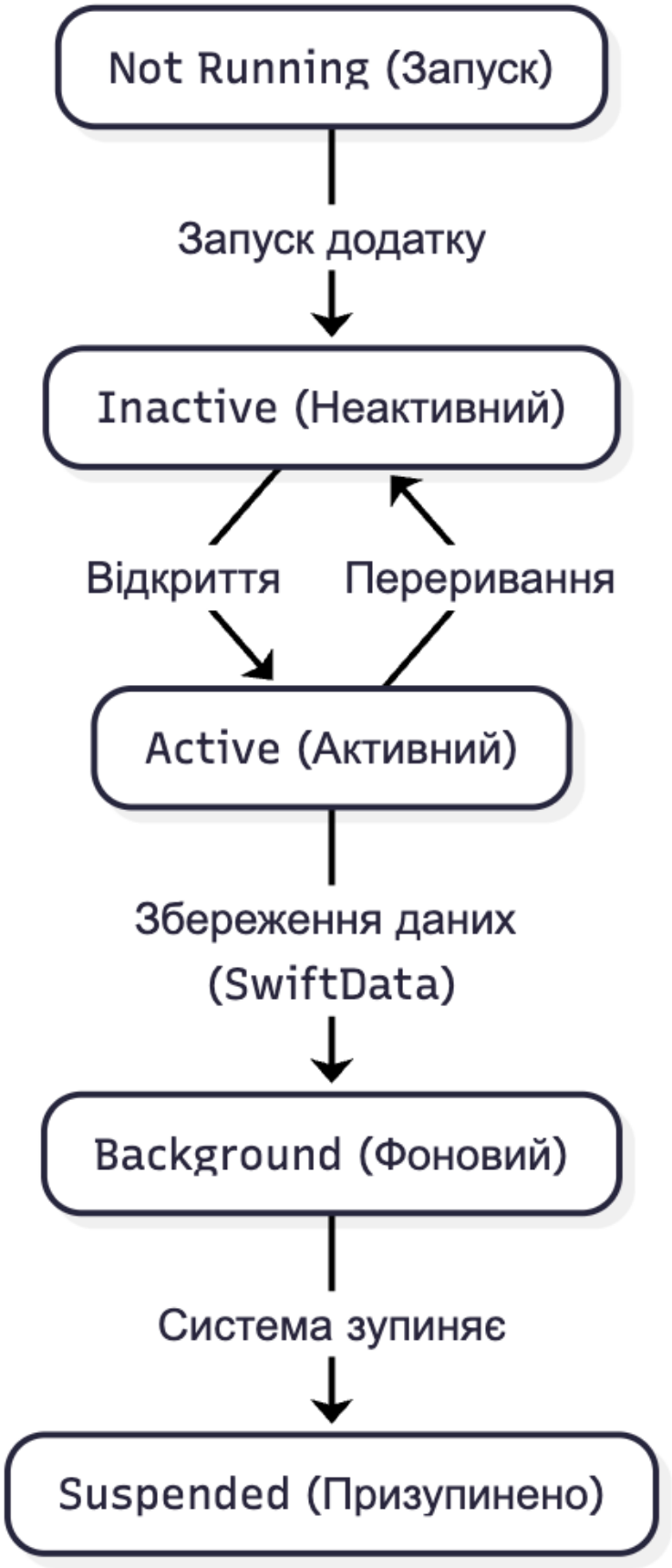
```
// Метод перевірки та створення транзакцій для підписок
private func processRecurringPayments(subscriptions: [Subscription], context: ModelContext) {
    let today = Date()
    for sub in subscriptions {
        // Перевірка: чи активна підписка і чи настала дата платежу
        if sub.isActive && sub.nextPaymentDate <= today {
            // Створення транзакції
            let newTransaction = Transaction(
                amount: sub.amount,
                date: sub.nextPaymentDate,
                note: "Авто-оплата: \(sub.title)",
                category: nil // Або прив'язка до категорії підписки
            )
            context.insert(newTransaction)
            // Оновлення дати наступного платежу (+1 місяць)
            if let nextMonth = Calendar.current.date(byAdding: .month, value: 1, to: sub.nextPaymentDate) {
                sub.nextPaymentDate = nextMonth
            }
        }
    }
}
import Charts
struct AnalyticsView: View {
    @Query private var transactions: [Transaction]
```

Продовження додатка Г

```
struct ChartData: Identifiable {
    let id = UUID()
    let category: String
    let amount: Decimal
  }
  // Обчислювальна властивість для групування витрат
  var groupedData: [ChartData] {
    // Словник для накопичення сум: [Назва категорії : Сума]
    var dict: [String: Decimal] = [:]
    for t in transactions {
      let key = t.category?.title ?? "Інше"
      dict[key, default: 0] += t.amount
    }
    // Трансформація словника в масив для Swift Charts
    return dict.map { ChartData(category: $0.key, amount: $0.value) }
          .sorted { $0.amount > $1.amount } // Сортування за спаданням
  }
  var body: some View {
    Chart(groupedData) { item in
      SectorMark(
        angle: .value("Сума", item.amount),
        innerRadius: .ratio(0.6), // Стиль "Donut Chart"
        angularInset: 1.5
      )
      .foregroundStyle(by: .value("Категорія", item.category))
    }
  }
}
```

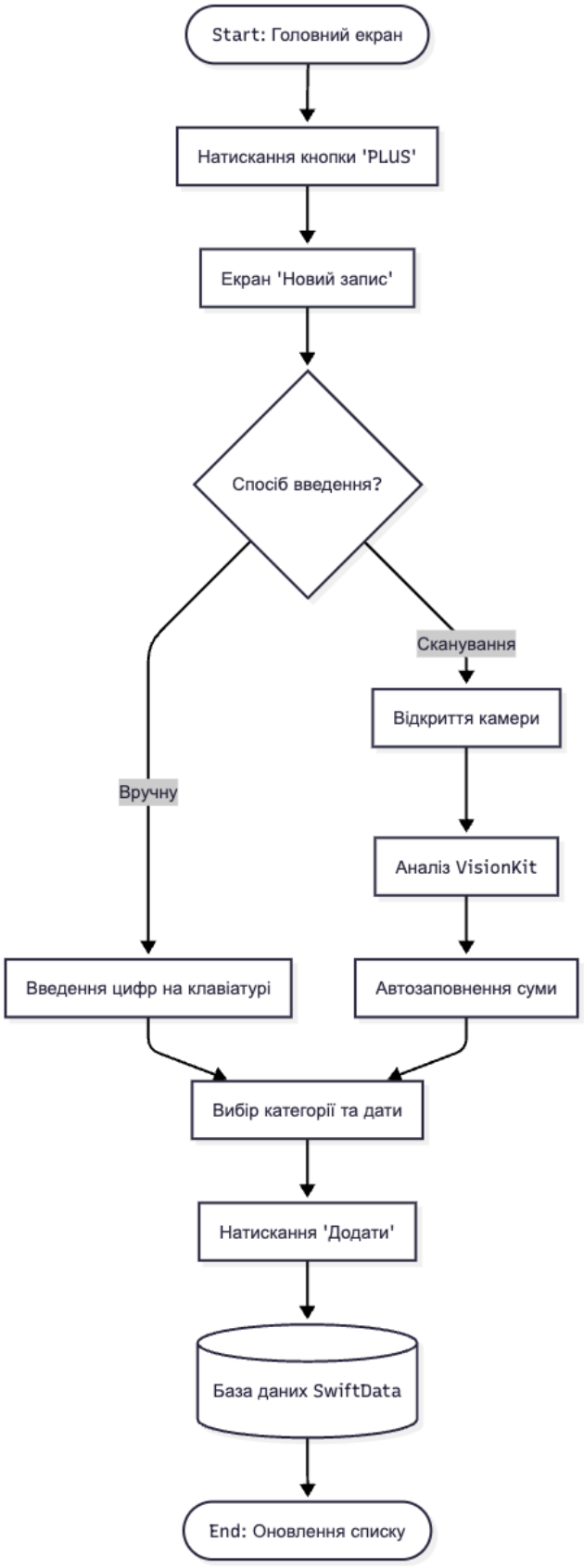
Додаток Д

Діаграма станів життєвого циклу iOS-додатку (State Machine)



Додаток Е

Сценарій користувача (User Flow) додавання транзакції



Додаток Ж

Сертифікат учасника IV Міжнародної науково-практичної Інтернет-конференції «Інновації та перспективні шляхи розвитку інформаційних технологій (ІПШРІТ-2025)»



Додаток З

Сертифікат учасника IV Міжнародної науково-практичної інтернет-конференції «Цифрова трансформація фінансової системиукраїни та країн V-4 в умовах євроінтеграції»

