Міністерство освіти і науки України
Національний університет «Києво-Могилянська Академія»
Кафедра мультимедійних систем

КУРСОВА РОБОТА
освітній ступінь - бакалавр
на тему:
Розробка застосунку для управління фінансами на платформі iOS.

Студента 3 курсу
спеціальність 121 «Інженерія
програмного забезпечення»
Катаскіна Олександра Олександровича
Науковий керівник:
Борозенний Сергій Олександрович
Кількість балів: _____

Члени комісії:

_____ _____
(підпис) (прізвище та ініціали)

_____ _____
(підпис) (прізвище та ініціали)

_____ _____
(підпис) (прізвище та ініціали)

Київ – 2025

## Зміст

**Анотація**

Робота присвячена створенню застосунку для управління фінансами. У роботі проведений аналіз потреб користувача, огляд існуючих рішень. Виведений список найбільш необхідних функцій, які має забезпечувати застосунок, а також проаналізовано відповідність існуючих рішень цим умовам.

Розроблено застосунок, який відповідає вимогам у попередніх пунктах роботи. Як платформу обрано iOS, інтерфейс реалізовано з використанням SwiftUI. Досліджено найбільш оптимальні способи реалізації вимог до застосунку, таких як безпека, збереження даних тощо. Аргументовано вибір використаних технологій, їх відповідність критеріям завдання.

Текстова частина роботи містить власне дослідження проблеми, її аналіз, огляд існуючих рішень, а також опис процесу розробки власного рішення.

# Вступ

## Постановка проблеми та актуальність теми

В умовах розвитку цифрової економіки та збільшення кількості джерел прибутку та витрат популярності набувають застосунки для управління особистими фінансами.

Зручність мобільних застосунків обумовлена розвитком банківської системи в Україні. Monobank – один з перших банків, що запровадив повноцінне управління рахунком через мобільний застосунок. Він дозволяє не тільки здійснювати платежі, а й переглядати статистику витрат, створювати регулярні платежі тощо. Зручність такого рішення доводить популярність банку – понад 9,5 мільйонів користувачів [1]. На момент початку 2025 року майже всі українські банки вже мають схожі додатки для керування рахунком.

Проблема полягає в тому, що, частіше за все, українці мають більше одного банківського рахунку. Про це говорить статистика Національного банку України – 193 млн рахунків на 84 млн користувачів [2]. Також не слід виключати готівкові кошти як один з основних способів користування грошима. Отже, актуальним є створення одного застосунку для контролю усіх електронних і фізичних рахунків.

Іншою проблемою є зручність використання застосунку для управління фінансами. Основним способом вводу витрат є ручний ввід, також, оскільки в Україні досі популярні паперові фіскальні чеки, актуальним є впровадження зручної системи імпорту витрат з фотографій. Окрім цього інноваційним рішенням є впровадження автоматичного імпорту транзакцій напряму з банків.

## Об’єкт і предмет дослідження

Об’єктом дослідження є процес управління особистими фінансами за допомогою мобільних застосунків.

Предметом дослідження є методи і технології розробки застосунку для управління фінансами, зокрема засоби інтеграції банківських даних та інструменти аналізу фінансів (бюджетування, розпізнавання тексту зображень тощо).

**Мета роботи і завдання**

Метою цієї роботи є розробка застосунку, який вирішить описані проблеми. Для досягнення зазначеної мети необхідно вирішити такі завдання:

1. Дослідити потреби користувачів застосунків для управління фінансами.
2. Дослідити можливості інтеграції з банківськими системами.
3. Розробити архітектуру застосунку, спроектувати інтерфейс користувача, структуру збереження даних.
4. Реалізувати основні функціональні модулі застосунку: аутентифікацію, захищене зберігання конфіденційних даних, операції з транзакціями, рахунками, категоріями витрат та бюджетами, облік регулярних платежів, імпорт та експорт даних.
5. Розробити компонент сканування чеків із використанням технологій розпізнавання тексту та машинного навчання.
6. Проаналізувати результати роботи застосунку та сформування висновки щодо досягнення мети проекту.

# РОЗДІЛ 1. Аналіз предметної області.

## 1.1 Огляд сфери персональних фінансів

Сфера персональних фінансів охоплює управління особистими коштами, бюджетування, контроль витрат і заощаджень для досягнення особистих фінансових цілей [3].

За оцінками Fact.MR, глобальний ринок мобільних застосунків для управління особистими фінансами стрімко зростає – з приблизно $2,9 млрд у 2024 році очікується збільшення до $12,58 млрд у 2034 році (середньорічний приріст близько 15,8%) [4]. Глобальна популярність смартфонів забезпечує аудиторію для таких застосунків, тому вони стають невід'ємним інструментом фінансового планування для багатьох людей. Так, у Великій Британії до 65% власників смартфонів встановили принаймні один фінансовий застосунок для роботи з грошима [5].

Застосунки для управління фінансами пропонують значно більше, ніж фіксацію витрат і доходів. Вони забезпечують централізований контроль фінансового стану користувача через інформаційні панелі, що відображають баланси рахунків, структуру витрат за категоріями, прогрес заощаджень тощо. Багато популярних рішень інтегруються з банківськими рахунками, що дозволяє автоматично завантажувати транзакції для економії часу користувача. Використовуються алгоритми категоризації витрат та інструменти аналізу, які допомагають виявити фінансові звички та надати персоналізовані поради. [6]  Серед інноваційних рішень - застосування штучного інтелекту для прогнозування фінансових показників і надання рекомендацій [7].

Уміння аналізувати власні фінанси є критично важливим. У 2018 році рівень фінансової грамотності українців оцінено лише у 11,6 балів з 21 можливого [8]. Війна та пов'язані з нею економічні виклики також не

дозволяють ігнорувати планування бюджету. Багато людей змушені уважніше ставитися до витрат, заощаджень та резервів на непередбачувані обставини [9].

Застосунки для управління фінансами, окрім власне обліку витрат, мають функцію персонального асистента, що підвищує фінансову грамотність та дисципліну користувачів за рахунок надання інформації у зручному та зрозумілому вигляді, обраховуючи певну статистику замість користувача. Важливо зазначити, що подібні застосунки не гарантують зміни фінансового становища, проте є ефективним інструментом контролю запланованих фінансових цілей.

Отже, огляд сфери персональних фінансів демонструє, що наразі є потреба у покращенні фінансової обізнаності, а мобільні застосунки – просте і ефективне рішення, адже є легкодоступним та надає достатньо інструментів, щоб підвищити обізнаність користувачів про власні фінанси. Тобто, застосунки для управління фінансами повинні мати не тільки облік витрат, а ще й виконувати функцію персонального асистента за рахунок надання інформації у зрозумілому вигляді.

### 1.2 Потреби користувачів та проблеми фінансового обліку

Використання застосунку для управління фінансами не дасть жодних результатів, якщо він не відповідає реальним потребам людей та не вирішує їх проблеми. На основі усних опитувань, власних вражень, а також огляду існуючих рішень були виявлені такі потреби:

- **Простота і швидкість використання.** Застосунок має забезпечувати зручний процес додавання транзакцій. Уся статистика має надаватися у інтуїтивно-зрозумілому вигляді.

- **Автоматизація збору даних.** Значна частина фінансових операцій проходить в електронній формі. Застосунок має надавати змогу автоматично отримувати транзакції з банківських рахунків. Це значно зменшує кількість транзакцій, які треба вводити вручну. Для іншої категорії транзакцій, де неможливо отримати виписку з банку, можливістю автоматизації є сканування тексту з фотографій (чеки, виписки тощо).
- **Підтримка та кількох рахунків та валют.** Користувачі часто оперують різними рахунками (банківські картки, готівка, електронні гаманці) і валютами. Тому застосунок повинен мати можливість вести облік кількох гаманців і конвертувати валюти за поточним курсом. В українських реаліях це особливо актуально, оскільки курс валют не є стабільним [10].
- **Аналітика і візуалізація даних.** Власне фіксація витрат не надає особливих переваг над електронними таблицями чи паперовим обліком. Важливою частиною застосунку є зрозумілі звіти: графіки витрат за категоріями, діаграми балансу тощо. Візуалізація допомагає виявити, на чому варто сфокусуватися, як краще спланувати бюджет тощо. Такий підхід дозволить приймати більш обґрунтовані фінансові рішення.
- **Бюджетування.** Важливо не лише відстежувати минулі витрати, а й планувати наперед. Тому сучасні застосунки для управління фінансами мають можливість встановити бюджет на категорії, рахунки тощо.
- **Регулярні платежі.** Регулярне введення однакових витрат не є зручним. Важливим є надання можливості встановити термін повторення таких витрат. Це зменшує ризик пропустити платіж і полегшує облік.

- **Безпека та конфіденційність.** Інформаціє про фінанси є достатньо чуттєвою, особливо, якщо вона взята з реальних банківських рахунків. Застосунок для своєї роботи має зберігати ключі доступу до цих рахунків, а також повну картину фінансових операцій користувача. Необхідно забезпечити шифрування ключів доступу, а також можливість біометричної автентифікації для захисту від несанкціонованого доступу.
- **Низька вартість.** Багато популярних застосунків для управління фінансами поширюються за підпискою або є цілком платними, що може стримувати користувачів. Надто висока вартість підписки на потрібні функції може стати причиною відмови від застосунку, навіть якщо він зручний.

Невідповідність переліченим потребам може призвести до відтоку користувачів. Статистика підтверджує цю тенденцію: за даними J.D. Power, 67% користувачів видаляють або перестають активно використовувати фінансовий застосунок уже протягом першого місяця використання, якщо вважають його функціональність недостатньою [11].

Підсумовуючи, для успіху застосунку для управління фінансами необхідно максимально орієнтуватися на потреби аудиторії. Застосунок має зібрати всі фінансові дані користувача в одному місці, забезпечити зручний ввід транзакцій та можливість автоматизації цього процесу, надати зрозумілу картину витрат і доходів.

### 1.3 Особливості українського ринку

Під час розробки застосунку важливо врахувати специфіку середовища, в якому він буде застосовуватися. Український ринок персональних фінансів має ряд особливостей, що впливають на вимоги до функціоналу та просування продукту. Історично склалося, що значна

частина застосунків для управління фінансами, якими користувалися українці, була іноземного походження (зокрема, російського). За даними дослідження Saldo Apps та сервісу ASObot, приблизно 50% нових встановлень таких застосунків досі припадає на додатки, розроблені в росії [12]. Більшість із них маскуються під міжнародні або локальні продукти: мають українську локалізацію, інтеграцію з українськими банками. Через це багато користувачів не усвідомлюють ризиків і продовжують ними користуватися навіть після початку повномасштабного вторгнення. За перші 18 місяців українці сплатили понад $250 000 на користь російських фінансових застосунків [12]. В умовах війни така ситуація стала критично неприйнятною як з погляду національної безпеки, так і етичних міркувань. Тому на ринку є чітка потреба у заміщенні подібних застосунків альтернативами.

Ще однією особливістю українського ринку є висока частка безготівкових розрахунків та популярність онлайн-банкінгу. За результатами дослідження Mastercard, 51% українців готові користуватися виключно цифровим банкінгом [13]. Це означає, що потенційні користувачі застосунків для управління фінансами вже мають певний досвід управління грошима з мобільних пристроїв.

Open Banking в Україні на момент початку 2025 року ще очікується: відповідно до Закону України «Про платіжні послуги», з 1 серпня 2025 року усі банки зобов’язані відкрити публічні API для доступу до інформації про рахунки за згодою клієнтів [14]. Це відкриває великі можливості для персональних фінансових застосунків: можна буде автоматично отримувати дані з різних банків.

Наразі існує неофіційне відкрите API від Monobank для клієнтів і сторонніх розробників. В межах цієї роботи воно буде використовуватися як приклад можливості автоматично вносити платежі. [15].

Отже, український ринок персональних фінансів нині характеризується високою часткою безготівкових розрахунків, а також високою часткою іноземних, часто російських, застосунків для управління фінансами. Завданням є витіснити російські продукти з міркувань безпеки. Висока частка безготівкових розрахунків, в свою чергу, означає, що користувач вже має приклади можливого функціоналу із застосунків, що належать банкам. Це дозволяє інтуїтивно зрозуміти інтерфейс застосунку для управління фінансами, оскільки він теж оперує транзакціями і рахунками.

## 1.4 Огляд існуючих рішень

### 1.4.1. You Need a Budget (YNAB)

Застосунок, орієнтований на концепцію нульового бюджету. YNAB вимагає від користувача планувати кожну гривню доходу наперед, розподіляючи кошти по категоріях ще до здійснення витрат. YNAB підтримує синхронізацію з банківськими рахунками (переважно західних фінансових установ) та роботу на мобільних пристроях і у веб-версії. Цей інструмент є платним – використовується модель підписки ($14,99 на місяць або $109 на рік) [6].

### 1.4.2. Mint

Це безкоштовний застосунок, що вміє автоматично збирати дані з банківських рахунків, кредитних карток та інших фінансових джерел і класифікувати транзакції за категоріями. Mint популярний у США завдяки інтеграції з банками та сервісами, проте для українських користувачів його функціональність обмежена, оскільки локальні банки не

підтримуються. Застосунок надає можливості планування бюджету та налаштування сповіщень. Монетизація реалізована шляхом показу реклами.

### 1.4.3. Spendee

Spendee працює за freemium-моделлю: базові можливості доступні безкоштовно, але для розширених функцій (таких як підключення банків чи спільні «гаманці») потрібна платна підписка. Застосунок підтримує різні валюти – можна створювати кілька гаманців у різних валютах і переглядати витрати з автоматичною конвертацією коштів. Spendee також дозволяє налаштовувати власні категорії витрат, додавати фотографії або геолокацію до транзакцій, передбачена функція підключення банківських рахунків. Також застосунок має спільні бюджети [16] [17].

### 1.4.4. Money Lover

Застосунок, який отримав відзнаку «App of the Day» на різних платформах [18]. Він надає користувачам можливість додавати транзакції, розділяти їх по категоріях, відображати статистику у вигляді графіків та діаграм. Money Lover підтримує створення бюджетів, сповіщення та ведення кількох гаманців. Важливою перевагою є синхронізація даних між пристроями. Також є режим Travel Mode – підтримка різних валют з актуальними курсами обміну [18]. Застосунок має freemium модель монетизації [19]. Платна версія дозволяє вести необмежену кількість гаманців, експортувати дані, відключити рекламу тощо. Також існує окрема послуга Linked Wallets – автоматичне підключення до банківських рахунків, яка доступна за окрему щомісячну плату і наразі підтримує банки переважно в країнах Південно-Східної Азії [19]. Для українських банків така інтеграція недоступна, тож із функціоналу залишається лише ручне введення або імпорт даних.

### 1.4.5 Monefy

Застосунок відображає структуру витрат у вигляді діаграми і не перевантажений зайвими функціями. Базова версія безкоштовна, а розширена (Monefy Pro) доступна за одноразову плату. Monefy здобув популярність серед українських користувачів завдяки простоті використання та відсутності необхідності оформлювати підписку.

## 1.5 Аналіз функціональності існуючих рішень

Для аналітичного порівняння варто виділити ключові можливості застосунків: спосіб внесення транзакцій, інтеграцію з банківськими рахунками, підтримку кількох валют, можливості бюджетування та планування, аналітику та візуалізацію даних, безпеку та захист даних, а також модель монетизації.

### 1.5.1 Спосіб додавання транзакцій та інтеграція з банками.

Усі розглянуті застосунки дозволяють вручну додавати транзакції, обирати категорію, суму, дату та коментар. Однак для підвищення зручності деякі застосунки впроваджують автоматизацію: імпорт даних з банківських систем або платіжних повідомлень. Наприклад, Mint та Spendee підключаються до банківських акаунтів. **Money Lover** має сервіс Linked Wallet, але перелік підтримуваних банків обмежений кількома країнами [19].

### 1.5.2. Підтримка кількох валют та робота з валютними курсами.

Spendee дозволяє створювати окремі гаманці в різних валютах і автоматично конвертувати суми за потреби [17], Money Lover має режим Travel Mode з оновленням курсів [18], Monefy також дає змогу вибирати валюту для кожного рахунку.

### 1.5.3. Бюджети та фінансове планування.

YNAB фактично побудовано довкола бюджетування, що змушує планувати всі витрати наперед. Spendee дозволяє створювати місячні бюджети на категорії та відстежувати їх [17]. Money Lover підтримує бюджети для кожної категорії і відображає попередження при наближенні до ліміту.

### 1.5.4. Аналітика та візуалізація даних.

Більшість застосунків надають користувачу засоби проаналізувати його фінансові звички через діаграми, графіки, звіти. Стандартом є кругова діаграма витрат по категоріях за місяць, графіки доходів/витрат по місяцях, графіки залишків. Monefy фокусується на простих діаграмах, Mint пропонує більш просунуті. Spendee у простій формі показує, скільки відсотків бюджету зайняла та чи інша стаття витрат. [17].

### 1.5.5. Безпека та захист даних.

Кожен застосунок приділяє увагу безпеці. Поширеними заходами є автентифікація за біометрією або PIN, шифрування даних як на пристрої, так і при передачі даних через мережу. YNAB зберігає дані переважно у хмарі, інші, переважно, локально.

### 1.5.6. Цінова модель та доступність.

Mint повністю безкоштовний – розробники заробляють на рекламі чи партнерських пропозиціях. Є застосунки з freemium-моделлю, де безкоштовно доступний обмежений набір функцій (Spendee, Money Lover). Вартість підписки у різних сервісів коливається: наприклад, YNAB найдорожчий ($15/міс)[6], Money Lover пропонує преміум за одноразовий платіж ($20) [19], а Monefy Pro коштує лише декілька доларів одноразовим платежем.

### 1.6 Висновки та постановка власного рішення

Аналіз існуючих рішень показав, що, незважаючи на велику кількість застосунків на ринку, кожен з них покриває певні специфічні вимоги, тобто немає єдиного монополіста, який задовольняє потреби всіх користувачів. Зокрема, виявлено, що:

- Більшість популярних рішень не підтримують автоматичне отримання транзакцій з українських банків або мають дуже обмежену підтримку.
- Платні моделі доступу до повного функціоналу роблять просунуті функції недоступними для тих, хто не готовий оформлювати підписку. Таким чином, безкоштовні версії недостатньо зручні, а платити за преміум готові далеко не всі.

Розробка застосунку в рамках цієї роботи ставить за мету заповнити виявлені прогалини та усунути недоліки конкурентів.

# РОЗДІЛ 2. Аналіз використаних технологій. Обґрунтування їх вибору.

## 2.1 Мова програмування та платформа

Для розробки застосунку обрано мову програмування Swift та платформу iOS. Swift – це сучасна мова від Apple, що поєднує високу продуктивність компільованого коду з безпечністю пам’яті, а також має зручний синтаксис. Завдяки використанню компілятора LLVM Swift-код транслюється у високоефективний машинний код, оптимізований для апаратного забезпечення [20].

Мова розроблялася з акцентом на безпеку: Swift усуває цілі класи помилок, що пов’язані з небезпечною роботою з пам’яттю через механізми автоматичного керування пам’яттю (ARC) та систему опціоналів [20].

Платформа iOS надає широкі можливості для створення безпечних та інтегрованих рішень. Мобільна операційна система від Apple має вбудовані засоби захисту даних: шифрування на рівні апаратного забезпечення, ізольоване середовище виконання додатків. Це є суттєвим для застосунку, що працює з чуттєвою інформацією.

## 2.2. Користувацький інтерфейс

Наразі існують 2 основних фреймворки для розробки користувацького інтерфейсу для систем iOS: UIKit та SwiftUI.

### 2.2.1 UIKit

UIKit – популярний фреймворк від Apple для розробки користувацького інтерфейсу. Підтримує мови Swift та Objective-C.

Цей фреймворк – безпечний та надійний вибір для будь-яких застосунків під iOS, оскільки існує з 2008 року [21] та має широкий вибір функціоналу під будь-які потреби розробників.

Із мінусів – фреймворк написаний під імперативну парадигму [22], тому змушує писати порівняно більшу кількість коду, аніж декларативний SwiftUI. Також, через зрілість фреймворку, основною структурою програми є MVC – Model View Controller [23]. Цей підхід є застарілим, оскільки існують кращі альтернативи (наприклад MVVM – Model View ViewModel).

### 2.2.2. SwiftUI

SwiftUI представлений у 2019 році [21]. Цей фреймворк є більш інноваційним, оскільки написаний для використання, здебільшого, в реактивній парадигмі. Стиль викладення коду декларативний, що дозволяє перенести відповідальність за створення та оновлення інтерфейсу на компілятор – розробник тільки описує необхідні елементи.

Головною перевагою над UIKit є повна інтеграція фреймворку Combine, що дозволяє легко підтримувати принцип «одного джерела правди».

Серед недоліків – незрілість фреймворку призводить до порівняно невеликого функціоналу. Також декларативний стиль викладення коду призводить до меншої гнучкості налаштувань інтерфейсу, порівняно з імперативним UIKit.

### 2.2.3. Обґрунтування вибору SwiftUI

Для застосунку в рамках цього проекту був обраний фреймворк SwiftUI. Аргументовано це тим, що головним пріоритетом є повна синхронізація даних між різними екранами (одне джерело правди), а також необхідність використовувати Combine. Наразі немає необхідності у складних елементах, що можуть бути реалізованими тільки з

використанням UIKit, тому можливостей SwiftUI вистачить для застосунку.

### 2.3. Реактивна парадигма. Використання фреймворку Combine.

Реактивна парадигма – підхід до розробки, що дозволяє оперувати не станами програми, а окремими подіями. Головним принципом є розповсюдження подій по програмі та відповідна реакція на них – певна обробка інформації. З цього випливає, що реактивний підхід є повністю асинхронним. Головні переваги такі [24]:

- При великому потоку даних код легше підтримувати, він є більш зрозумілим, оскільки усі дані йдуть з одного місця.
- При великій кількості асинхронних запитів до мережі, системних компонентів та інших ресурсів реактивний підхід дозволяє легко синхронізувати інтерфейс з вхідними даними.
- Принципова відсутність станів програми зменшує кількість можливих помилок, полегшує тестування.

Вибір реактивного підходу аргументовано такими вимогами

- Дані у застосунку завжди надходять асинхронно: з бази даних або з мережі (з банківських рахунків). Реалізація синхронізації даних за допомогою подій набагато простіша та надійніша, аніж за допомогою асинхронних присвоєнь, оскільки вони не завжди є безпечними в контексті роботи з пам’яттю.
- Робота з транзакціями, категоріями, рахунками призводить до великої кількості різних вікон програми. Є необхідність синхронізувати інформацію при зміні даних. Реактивний підхід, як і в минулому пункті, дозволяє зробити це набагато простіше і надійніше, аніж імперативний.

Наразі є 2 основних фреймворки, що використовують реактивний підхід: Combine та RxSwift. Серед них очевидним вибором є Combine, оскільки він є розробкою Apple та написаний під системи iOS, що робить його більш надійним рішенням через відсутність проблем з версіями фреймворку. Також перевагою є швидкодія [25]:

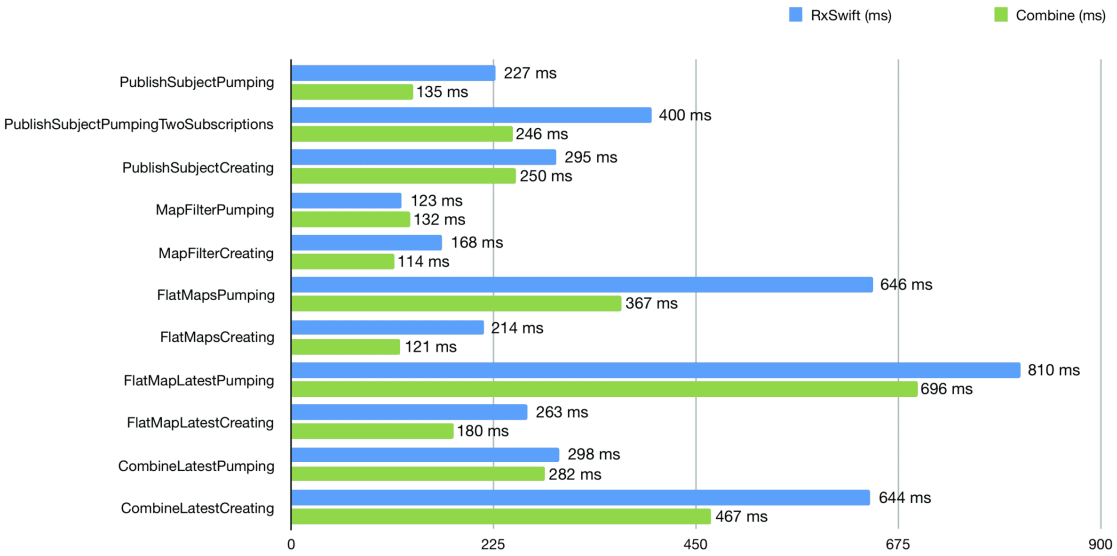


*Рисунок 2.3 – порівняння швидкодії RxSwift та Combine*

Отже, для підтримки «одного джерела правди» та більш надійної синхронізації даних всередині програми було обрано реактивний підхід замість класичного імперативного.

## 2.4. Використання паттерну MVVM при розробці застосунку.

Патерн MVVM передбачає розподіл додатку на три основні компоненти: Model, View і ViewModel:

- Model: це дані та бізнес-логіка застосунку – сутності (транзакції, рахунки, категорії), служби (робота з API, збереження даних) та інші невидимі користувачу частини.

- View: це інтерфейс користувача – екрани, сформовані з SwiftUI View, які відображають стан моделі і передають дії користувача (натискання кнопок, введення даних).
- ViewModel: проміжний шар, що зв’язує View та Model. ViewModel отримує від модельного шару необхідні дані, обробляє їх та надає View готовий до показу стан. Також ViewModel приймає події від View і викликає відповідні методи Model.

Головна ідея MVVM – ізолювати логіку представлення і підготовки даних у ViewModel, залишивши View шаром відображення. У класичному MVC для iOS контролери представлення (ViewController) часто розросталіся до величезних розмірів, містячи і логіку відображення, і бізнес-логіку – від чого виник термін Massive View Controller. MVVM розв’язує цю проблему, делегуючи всю логіку у ViewModel [26].

Оскільки застосунок використовує реактивну парадигму за допомогою фреймворку Combine, використання MVVM також допомагає правильно налаштувати потік подій. Так ViewModel, зазвичай, містить такі об’єкти, як Publisher. Publisher – елемент фреймворку Combine. Він може приймати події і надсилати їх відповідно. У Model можуть міститися запити до мережі або бази даних, що, відповідно, надсилають події до Publisher’ів. View, в свою чергу, підписується на оновлення Publisher’ів, що дозволяє без зайвих присвоєнь оновлювати дані як тільки поступила нова подія.

Таким чином, використання паттерну MVVM у зв’язці з Combine допомагає інкапсулювати логіку, відображення і дані, а також налаштувати зв’язки між ними за допомогою асинхронних подій.

## 2.5. Зберігання даних: CoreData та Keychain.

Застосунок в рамках роботи оперує наступними даними: транзакції, категорії транзакцій, рахунки, бюджети. Для цих даних зручним є використання бази даних. Тут очевидний вибір пав на CoreData – інтегроване рішення, що не потребує окремих залежностей.

Окремо застосунок працює із реальними банківськими даними, що потребує ключі доступу. Зберігати ключі доступу в базі даних – небезпечно, тому використовується інше інтегроване сховище – Keychain.

### 2.5.1. CoreData

CoreData не є окремою базою даних – це лише граф об’єктів. CoreData використовує базу даних SQLite, натомість надає можливість працювати з об’єктами напряму в середовищі розробки, що відкидає необхідність створювати окремі моделі для сутностей, що зберігаються в базі даних.

Вибір CoreData аргументований зручністю використання: не потрібно налаштовувати зовнішні залежності, є можливість оперувати створеними в CoreData об’єктами, що теж полегшує процес розробки.

### 2.5.2. Keychain

Keychain – це зашифроване сховище, доступне додатку через API Security framework. Дані в Keychain шифруються з використанням надійних алгоритмів (AES-256) і можуть бути прочитані тільки тим додатком, який їх помістив [27]. Використання Keychain гарантує, що навіть у разі компрометації файлової системи пристрою зловмисник не отримає доступу до ключів і паролів, оскільки для розшифрування потрібні системні ключі, пов’язані з апаратним забезпеченням.

Для спрощення роботи із цим сховищем використано бібліотеку KeychainAccess, що надає зручний інтерфейс для збереження ключ-значення. [28].

### 2.5.3. Шифрування даних та конфіденційність

У iOS всі файли додатків за замовчуванням захищені механізмом *Data Protection*. Це означає, що коли пристрій заблоковано (екран вимкнено і встановлено пароль), ключі шифрування файлової системи недоступні, і всі файли практично знаходяться у зашифрованому стані. Для підвищення безпеки додаток може примусово встановити для файлів бази даних найвищий клас захисту – NSFileProtectionComplete, при якому файл шифрується ключем, похідним від пароля користувача та унікального ідентифікатора пристрою [29]. У такому режимі дані залишаються зашифрованими до моменту, коли користувач розблокує пристрій.

В рамках роботи використовується стандартний захист iOS для файлів бази (Level CompleteUnlessOpen за замовчуванням), що забезпечує баланс між безпекою та зручністю. Доступ до бази можливий навіть коли додаток працює у фоні, але дані захищені, коли додаток неактивний.

Крім того, реалізовано додатковий рівень захисту доступу до самого застосунку. При запуску користувач проходить біометричну автентифікацію – Face ID. Це засіб захисту від несанкціонованого доступу: навіть якщо телефон розблоковано, відкриття додатку потребує підтвердження особи власника. Технологічно це реалізовано через фреймворк LocalAuthentication, який надає простий інтерфейс для запиту біометричних даних.

### 2.6. Інтеграція з банківськими API

Однією з ключових функцій застосунку є автоматичний імпорт транзакцій із зовнішніх фінансових сервісів. У рамках роботи реалізовано інтеграцію тільки з API Monobank, оскільки, як зазначено в першому пункті роботи, наразі реалізація Open Banking API ще не увійшла в силу.

Теоретично інтеграція з банківським API зводиться до виконання HTTP-запитів до сервера банку і обробки відповіді у форматі JSON. Це вимагає вирішення питань авторизації, формату даних, обмежень на кількість запитів та безпеки передачі інформації.

#### 2.6.1. Принципи роботи REST API у Monobank open API.

Monobank надає розробникам відкритий REST API, який дозволяє отримувати інформацію про клієнта, баланс рахунків, історію транзакцій, курси валют тощо/ Взаємодія побудована наступним чином: користувач (власник рахунку) отримує у своєму кабінеті Monobank персональний токен доступу (спеціальний рядок-ключ). Цей токен використовується додатком при кожному запиті як засіб автентифікації – передається в HTTP-заголовку X-Token. Таким чином, сервер Monobank може ідентифікувати запит як такий, що надходить від певного клієнта, і надіслати дані саме по його рахунках.

Обмін даними відбувається у форматі JSON. Відповідь на запит містить поля з інформацією. Для запиту балансу та списку рахунків – це JSON-об’єкт з іменами клієнта, переліком рахунків (номер, валюта, поточний баланс, кредитний ліміт тощо); для запиту виписки транзакцій – масив об’єктів, де кожен об’єкт описує транзакцію (дата, час, сума, валюта, опис, категорія та інші атрибути) [30].

Monobank встановлює деякі обмеження та особливості для свого API. З міркувань навантаження, певні методи можна викликати не частіше визначеного інтервалу. Отримання інформації про клієнта (/personal/client-info) дозволено не більше ніж раз на 60 секунд [30]. Історія транзакцій надається за період до місяця одним запитом, але можна робити запити по черзі для різних інтервалів, щоб отримати повну історію. Також Monobank підтримує механізм webhook-сповіщень: розробник може зареєструвати URL, на який банк надсилатиме повідомлення при кожній новій транзакції.

### 2.7. Використання машинного навчання та OCR (Apple Vision, Core ML)

Ще однією можливістю застосунку є автоматичне додавання витрат шляхом сканування зображень. Це реалізовано з використанням технологій комп'ютерного зору (OCR) та машинного навчання для аналізу тексту. Теоретично реалізовано 2 функції: розпізнавання тексту на зображенні (Optical Character Recognition) та інтелектуальне опрацювання розпізнаного тексту з метою вилучення потрібної інформації за допомогою навченої моделі.

#### 2.7.1. OCR з використанням Apple Vision.

Вбудований фреймворк Vision дозволяє виконувати задачі комп'ютерного зору локально на пристрої. Vision включає і готовий функціонал для розпізнавання тексту на зображеннях, це робиться через клас VNRecognizeTextRequest. Цей клас виконує аналіз зображення та повертає знайдені текстові фрагменти (ряди символів) [31].

Фактично Apple Vision реалізує OCR повністю на пристрої, використовуючи вбудовані моделі машинного навчання, оптимізовані для iOS. Перевагою цього підходу є те, що зображення чеків не

відправляються на сторонні сервери для обробки – вся обробка здійснюється локально, що забезпечує конфіденційність даних користувача, оскільки фінансова інформація з чеків нікуди не надсилається.

Алгоритм додавання транзакції через сканування можна реалізувати наступним чином:

- Зображення імпортується в додаток з галереї. Для цього використовується PhotosUI – вбудований фреймворк, що дозволяє додати інтерфейс галереї з можливістю обирати елементи.
- Зображення передається у Vision-фреймворк, де запускається VNRecognizeTextRequest. Модель OCR після обробки повертає набір розпізнаних рядків тексту та їх координати на зображенні.
- Результати роботи VNRecognizeTextRequest аналізуються – за допомогою навченої моделі можна виділити назву магазину, що виготовив чек тощо.

### 2.7.2. Аналіз тексту за допомогою машинного навчання.

Отримавши сирий текст з чека, застосунок має визначити такі дані: суму транзакції, дату операції, назву магазину. Частково це можна зробити простим пошуком шаблонів: сума – це найбільше число з двома десятковими знаками, дата – рядок формату dd.mm.yyyy.

Однак, для пошуку назви магазину недоцільно використовувати шаблони, адже вони не мають специфічного формату. Саме тому використано машинне навчання. CoreML – технологія, що дозволяє інтегрувати моделі машинного навчання в застосунок. Модель для

CoreML легко створити за допомогою Create ML. Цей інструмент дозволяє навчити модель класифікації тексту на основі інформації, що належить різним категоріям [32].

Інтеграція моделі у застосунок здійснюється шляхом додавання моделі (файлу .mlmodel) до проєкту Xcode. Після цього автоматично генерується клас Swift для роботи з нею.

# РОЗДІЛ 3. Реалізація застосунку.

У цьому розділі описано структуру та реалізацію основного й допоміжного функціоналу застосунку.

## 3.1. Модель бази даних.

Застосунок, в першу чергу, має оперувати транзакціями. Транзакція може мати категорію та повинна мати рахунок, з якого вона зроблена. Також транзакція може бути повторюваною, це також треба врахувати.

Описаний набір даних доцільно зберігати в базі даних. Як вже було визначено в попередньому пункті, для реалізації обрано CoreData. Відповідно, так виглядає фінальна модель бази даних застосунку:

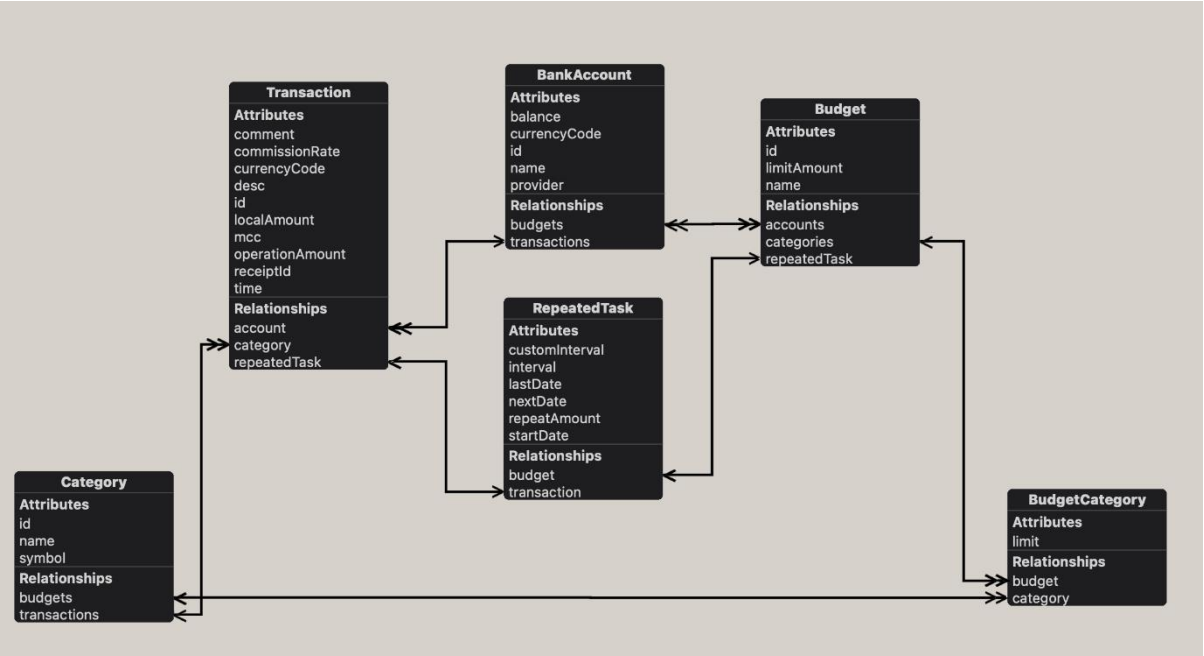


*Рисунок 3.1 – Схема бази даних застосунку*

Було спроектовано таку базу даних, щоб кожен об'єкт мав необхідні реляції, які допоможуть класифікувати його в повній мірі.

Так транзакція має account – рахунок, до якого вона прив'язана. Також транзакція опціонально має категорію.

Бюджет має рахунки, на які він поширюється, а також категорії. Для зручності було створено сутність BudgetCategory – зв’язок між категорією та бюджетом. Така логіка пояснюється тим, що бюджет може мати категорії, кожна з яких має свій ліміт.

Також бюджет та транзакція опціонально мають RepeatedTask – своєрідну підписку, що зберігає інтервал та кількість повторень бюджету чи транзакції. Ця логіка аргументована тим, що необхідно зберігати точну інформацію про регулярні дії.

Варто зазначити, що всі реляції є двосторонніми, тобто, умовно, account містить багато transactions, тоді як кожна transaction містить один account. Подібна логіка поширюється на інші реляції.

Як вже було сказано в попередньому пункті, існують такі дані, які не варто зберігати в базі даних. До них відносяться ключі доступу до банківських рахунків. Саме тому не було створено об’єктів CoreData для цього.

### 3.2. Застосування бази даних в застосунку.

#### 3.2.1. CRUD операції з сутностями CoreData.

CoreData дозволяє напряму використовувати сутності в коді програми. Таким чином, без зайвих перетворень можна реалізувати CRUD операції – Create, Read, Update, Delete.

Створення сутності відбувається за допомогою ініціалізації об’єкту в програмі з відповідним контекстом CoreData. Атрибути записується за допомогою присвоєння значень. Збереження сутності відбувається за допомогою збереження контексту. Приклад створення сутності категорії:

```swift
// MARK: - Create
func createCategory(name: String,
                    symbol: String) {
    let category = Category(context: context)
    category.id = UUID()
    category.name = name
    category.symbol = symbol

    saveContext()
}
```

*Рисунок 3.2.1.1 – Метод додавання категорії*

```swift
// MARK: - Save
private func saveContext() {
    do {
        if context.hasChanges {
            try context.save()
        }
    } catch {
        print("Failed to save context:", error)
    }
}
```

*Рисунок 3.2.1.2 – Метод збереження контексту*

Отримання сутностей відбувається за допомогою функції fetch. Для застосування фільтрів можна використати предикат. Приклад отримання сутностей:

```swift
// MARK: - Read
func fetchCategory(_ id: UUID) -> Category? {
    let request: NSFetchRequest<Category> = Category.fetchRequest()
    request.predicate = NSPredicate(format: "id == %@", id as CVarArg)
    return try? context.fetch(request).first
}

func fetchCategories() -> [Category] {
    let request: NSFetchRequest<Category> = Category.fetchRequest()
    return (try? context.fetch(request)) ?? []
}
```

*Рисунок 3.2.1.3 – Метод отримання категорії*

Оновлення сутностей схоже на створення – відбувається присвоєння значень атрибутів. Також необхідно зберегти контекст:

```
// MARK: - Update
func updateCategory(_ category: Category,
                    name: String? = nil,
                    symbol: String? = nil,
                    budgets: [BudgetCategory]? = nil,
                    transactions: [Transaction]? = nil
) {

    if let name { category.name = name }
    if let symbol { category.symbol = symbol }
    if let budgets { category.budgets = NSSet(array: budgets) }
    if let transactions { category.transactions = NSSet(array: transactions) }

    saveContext()
}
```

*Рисунок 3.2.1.4 – Метод оновлення категорії*

Видалення сутностей відбувається за допомогою метода delete:

```
// MARK: - Delete
func deleteCategory(_ category: Category) {
    context.delete(category)
    saveContext()
}
```

*Рисунок 3.2.1.5 – Метод видалення категорії*

Таким чином, CoreData надає прості методи, що дозволяють здійснювати всі основні операції з сутностями.

### 3.2.2. Застосування CoreData з Combine.

SwiftUI надає зручний інтерфейс для відображення та інших операцій із сутностями. За допомогою @FetchRequest можна отримувати сутності напряму з бази даних. Однак, такий спосіб порушує принципи інкапсуляції, а також викликає певні труднощі у плануванні архітектури застосунку [33].

Аргументується це тим, що CoreData – не єдине можливе джерело даних в застосунку. Існують інші бази даних під iOS. Наприклад – Realm. При зміні бази даних з тих чи інших причин доведеться повністю змінювати як логіку застосунку, так і відображення даних. Саме через це було прийнято рішення не використовувати @FetchRequest.

Замість @FetchRequest популярним рішенням є імплементація CoreData Publisher'а – зв'язку між CoreData та Combine. Як вже було описано, Publisher дозволяє як приймати, так і надсилати події. Фреймворк надає можливість створювати власні Publisher'и, що і було зроблено для імплементації необхідної логіки. Прикладом є імплементація від Bassem Quolta [34]. Цей підхід і був використаний у рамках роботи.

### 3.2.3. Інкапсуляція роботи з базою даних.

У попередньому пункті описано, що робота з сутностями напряму порушує принципи інкапсуляції, отже доцільно виокремити роботу з базою даних в окремий клас.

У застосунку було створено репозиторій для кожної сутності. Репозиторій має такі функції:

- CRUD операції з сутністю.
- Publisher з усіма екземплярами сутності.

Такий набір функцій пояснюється вимогами застосунку. База даних постійно оновлюється під час роботи програми, оскільки користувач, частіше за все, заходить до застосунку для того, щоб додати нові витрати. Враховуючи це, а також те, що для застосунку у реактивній парадигмі необхідно налаштувати потік даних, і були створені Publisher'и, що завжди містять актуальний стан бази даних. Вони надсилають події оновлення усім підписникам, а також зберігають актуальний масив значень.

Збереження стану не є чисто реактивним підходом, однак є виправдною мірою, оскільки без цього логіка застосунку значно ускладниться.

### 3.3. Автентифікація за допомогою Face ID.

Необхідною мірою безпеки, що убезпечить чутливу інформацію застосунку є автентифікація з використанням біометрії. Усі сучасні пристрої iPhone підтримують технології Face ID – 3D сканер обличчя.

Apple надає зручний вбудований інтерфейс для роботи з автентифікацією за допомогою фреймворку LocalAuthentication:

```
func unlock() async {
    guard !isUnlocked else { return }

    let reason = "Authenticate to access Finance Manager"
    do {
        if try await context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason: reason) {
            DispatchQueue.main.async { [weak self] in
                self?.isUnlocked = true
                self?.context.invalidate()
            }
```

*Рисунок 3.3 – Робота з LocalAuthentication*

Таким чином була реалізована локальна автентифікація, що дозволяє точно впевнитися у тому, що застосунком користується реальний власник пристрою.

### 3.4. Робота з банківськими API.

Як вже було зазначено, робота зі сторонніми API зводиться до надсилання запитів та очікування відповіді у вигляді JSON файлу. Наразі є 2 популярних методи реалізації такої логіки: вбудований URLSession та бібліотека Alamofire.

Alamofire – велика бібліотека, яка має безліч фукнціоналу, що спрощує роботу із запитами до мережі [35]. Для проекту в рамках роботи такий інструмент є надлишковим, тому було прийнято рішення використовувати вбудований URLSession.

Все, що необхідно для запиту – створити URLRequest з необхідним посиланням, HTTP методом та ключем доступу:

```
var request = URLRequest(url: self.baseURL.appendingPathComponent("personal/client-info"))
request.httpMethod = "GET"
request.setValue(token, forHTTPHeaderField: "X-Token")
```

*Рисунок 3.4.1 – Створення URLRequest*

Запити за допомогою URLSession є асинхронними, тому постало питання правильної обробки інформації, що надходить у відповідь. Так як в рамках роботи використовується Combine, був використаний спеціальний Publisher – dataTaskPublisher. Цей Publisher повертає дату або помилку тоді, коли виконається запит і прийде відповідь від серверу [36]. Також існує спеціальний метод, що дозволяє у випадку отримання інформації відразу її декодувати у потрібну структуру. Таким чином, асинхронну логіку запитів легко реалізувати в контексті реактивної парадигми з використанням подій:

```
return URLSession.shared.dataTaskPublisher(for: request)
    .tryMap { data, response -> Data in
        guard let httpResponse = response as? HTTPURLResponse else {
            throw MonobankError.invalidResponse
        }
        guard 200..<300 ~= httpResponse.statusCode else {
            if httpResponse.statusCode == 401 {
                throw MonobankError.serverError("Unauthorized - check your token")
            } else {
                throw MonobankError.httpStatus(httpResponse.statusCode)
            }
        }
        return data
    }
    .decode(type: ClientInfo.self, decoder: JSONDecoder())
```

*Рисунок 3.4.2 – Реалізація обробки запиту*

У запиті використовується ключ доступу – токен. Як вже зазначено, ключі доступу зберігаються у Keychain, тому іншим завданням є налаштування потоку даних: від запиту у Keychain до отримання відповіді з серверу.

Так функція loadToken() -> AnyPublisher<String, Error> {…} робить запит у Keychain, а повертає Publisher із даними або помилкою. Передати отримані дані далі по ланцюгу можна за допомогою оператора .flatMap, що приймає на вхід дані, а повертає не оброблену відповідь, а новий Publisher [37]. Це дозволяє використати ланцюг Publisher'ів для налаштування логіки передачі даних до ViewModel без зайвих змінних та присвоєнь:

```
func fetchAccounts() -> AnyPublisher<[Account], Error> {
    return loadToken()
        .flatMap { token -> AnyPublisher<[Account], Error> in
```

*Рисунок 3.4.3 – Застосування .flatMap*

Оскільки дані із запиту необхідно декодувати, це робиться за допомогою новоствореної структури – моделі даних для роботи з API. На рисунку вище такою є Account. Такі структури повинні реалізовувати протокол Decodable [38]. Цей протокол дозволяє ініціалізувати структурі із JSON об'єкту:

```
struct Account: Codable {
    let id, sendID: String
    let currencyCode: Int
    let cashbackType: String
    let balance, creditLimit: Int
    let maskedPan: [String]
    let type, iban: String

    enum CodingKeys: String, CodingKey {
        case id
        case sendID = "sendId"
        case currencyCode, cashbackType, balance, creditLimit, maskedPan, type, iban
    }
}
```

*Рисунок 3.4.4 – Застосування Decodable*

Використовувати структури, призначені для роботи з API, в інших частинах застосунку – порушення інкапсуляції. Тому для усіх основних даних, якими оперує застосунок, створені окремі класи чи структури:

```
class AccountModel: Identifiable, Equatable, Hashable {
    static func == (lhs: AccountModel, rhs: AccountModel) -> Bool {
        lhs.id == rhs.id
    }

    func hash(into hasher: inout Hasher) {
        hasher.combine(id)
    }

    let id: String
    let name: String
    var balance: Int64
    var currency: Int32
    var provider: AccountProvider?

    init(id: String, name: String, balance: Int64, currency: Int32, provider: AccountProvider? = nil) {
        self.id = id
        self.name = name
        self.balance = balance
        self.currency = currency
        self.provider = provider
    }
```

*Рисунок 3.4.5 – Приклад інкапсульованої моделі*

Такі класи і структури незалежні від сутностей бази даних чи JSON об’єктів з API. Натомість, вони мають методи ініціалізації, що дозволяють конвертувати сторонні об’єкти в необхідний клас чи структуру:

```
init?(_ account: BankAccount) {
    guard let id = account.id, let name = account.name else { return nil }

    self.id = id
    self.name = name
    self.balance = account.balance
    self.currency = account.currencyCode
    if let providerString = account.provider,
       let provider = AccountProvider(rawValue: providerString) {
        self.provider = provider
    }
}

init(_ externalAccount: Account) {
    self.id = externalAccount.id
    self.name = "Monobank \(externalAccount.type)"
    self.balance = Int64(externalAccount.balance)
    self.currency = Int32(externalAccount.currencyCode)
    self.provider = .monobank
}
```

*Рисунок 3.4.6 – Методи ініціалізації у моделі*

Для підтримки інкапсуляції уся робота з API, як і репозиторії для бази даних, винесена в окремий клас. Таким чином, ViewModel може використовувати екземпляр цього класу для виклику необхідних методів, однак не буде впливати на власне перебіг запитів.

## 3.5. Проста реалізація зчитування інформації з чеків.

Галузь комп’ютерного зору та машинного навчання надає безмежні можливості, однак вимагає багато часу та компетентності для розробки потрібних функцій. Apple має свої фреймворки для роботи з цими галузями: Vision та CoreML. Ці фреймворки мають безліч можливостей, однак, в цей же час, надають порівняно простий інтерфейс для реалізації базових функцій. Такими, у випадку цієї роботи, є зчитування інформації з фотографії, а також класифікація тексту.

### 3.5.1. Реалізація зчитування інформації з фотографій.

Одразу варто зазначити, що, як і робота з мережею і базою даних, обробка фотографій виконується асинхронно, що виправдовує налаштування потоку даних за допомогою Combine: був створений Publisher, який отримує на вхід зображення, а повертає масив зчитаних рядків.

Власне зчитування відбувається за допомогою VNImageRequestHandler, що приймає на вхід зображення. Цей клас має метод .perform, що запускає сканування з відповідним запитом. Запит – об'єкт класу VNRecognizeTextRequest [31].

Після отримання результатів сканування з масиву результатів береться необхідна інформація: власне текст та його координати.

### 3.5.2. Реалізація класифікації тексту з масиву рядків.

Деякі поля чеку можна проаналізувати звичайними методами – наприклад, перевірити відповідність певним правилам. Так сума може бути найбільшим числом, а дата – стрічкою формату dd.mm.yyyy тощо. Для демонстрації роботи CoreML була обрана класифікація назви магазину, що виробив чек. Ця інформація не може бути легко класифікована звичайними алгоритмами, тому виправдано використання машинного навчання.

CoreML потребує навчену модель, яка і буде визначати, чи рядом підходить під критерій (в цьому випадку – чи є він назвою магазину). Ця модель була створена за допомогою програми Create ML. Цей застосунок дозволяє створювати різні моделі, в тому числі класифікатор тексту [39].

Для навчання класифікатора необхідні тестові дані. У випадку цього проекту тестові дані були обмежені 62 фотографіями чеків різних магазинів. Для демонстрації моделі такої кількості цілком достатньо, однак для повноцінної роботи функції необхідно надати набагато більше екземплярів для навчання моделі.

Текст чеків розділений на рядки, кожен з яких підписаний або як «Store», або як «Other». Усі рядки розділені у співвідошенні 6 до 1 для тренування і тестування моделі відповідно:

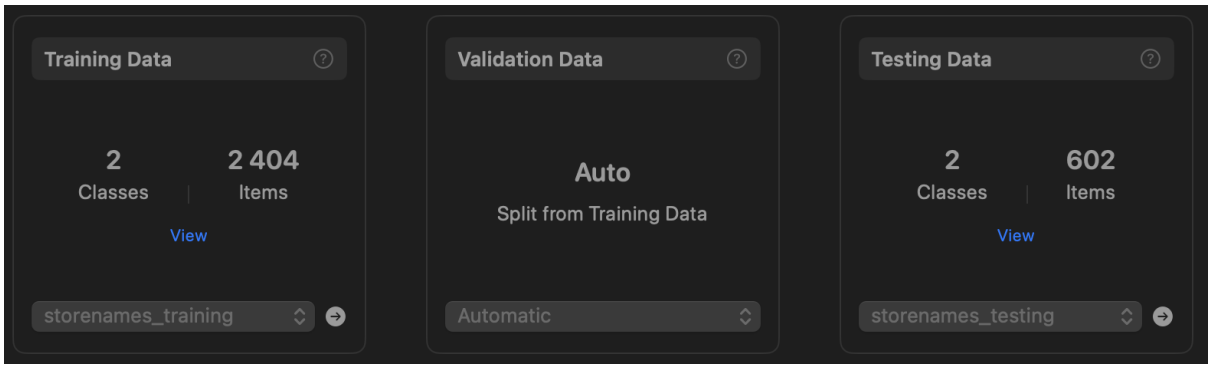


*Рисунок 3.5.2.1 – Меню створення класифікатора*

Таким чином, застосунок Create ML дозволив створити демонстраційну модель, яка з порівняно високою точністю класифікує текст як «назву магазину» або як «інший текст».

Використання моделі у коді відбувається за допомогою методу prediction:

```
func isStoreName(_ line: String) -> Bool {
    guard
        let output = try? model.prediction(text: line),
        output.label == "Store"
    else { return false }

    return true
}
```

*Рисунок 3.5.2.2 – Використання класифікатора в коді*

### 3.6. Реалізація інтерфейсу.

Головною функцією застосунку є облік транзакцій. У попередніх пунктах вже був описаний алгоритм роботи з базою даних, стороннім API та розпізнаванням тексту на зображеннях. Для того, щоб додати можливість працювати з цими методами, необхідно створити користувацький інтерфейс.

#### 3.6.1. Інтерфейс ручного вводу транзакції.

Для ручного вводу була реалізована двокрокова форма. Перший екран надає можливість ввести необхідну інформацію, а саме суму та рахунок транзакції. Другий екран допоміжний – дозволяє опціонально налаштувати додаткові поля транзакції, такі як категорія, опис, примітка, а також додати регулярне повторення.

У SwiftUI подібна логіка двох екранів реалізована за допомогою допоміжної View з двома станами TransactionStep:

```
enum TransactionStep {
    case amount
    case details
}

struct ManualTransactionFlowView: View {
    @Environment(\.dismiss) var dismiss

    @StateObject var vm = ManualTransactionViewModel()
    @State private var step: TransactionStep = .amount

    var body: some View {
        NavigationView {
            Group {
                switch step {
                case .amount:
                    AmountStepView(viewModel: vm) {
                        step = .details
                    }

                case .details:
                    DetailsStepView(viewModel: vm,
                                    onPrevious: { step = .amount },
                                    onSave: {
                        vm.saveTransaction()
                        dismiss()
                    })
                }
            }
```

*Рисунок 3.6.1.1 – Код інтерфейсу ручного вводу*

Відповідно, при зміні значення step зміниться і відображений екран. Так виглядає власне інтерфейс додавання транзакції:

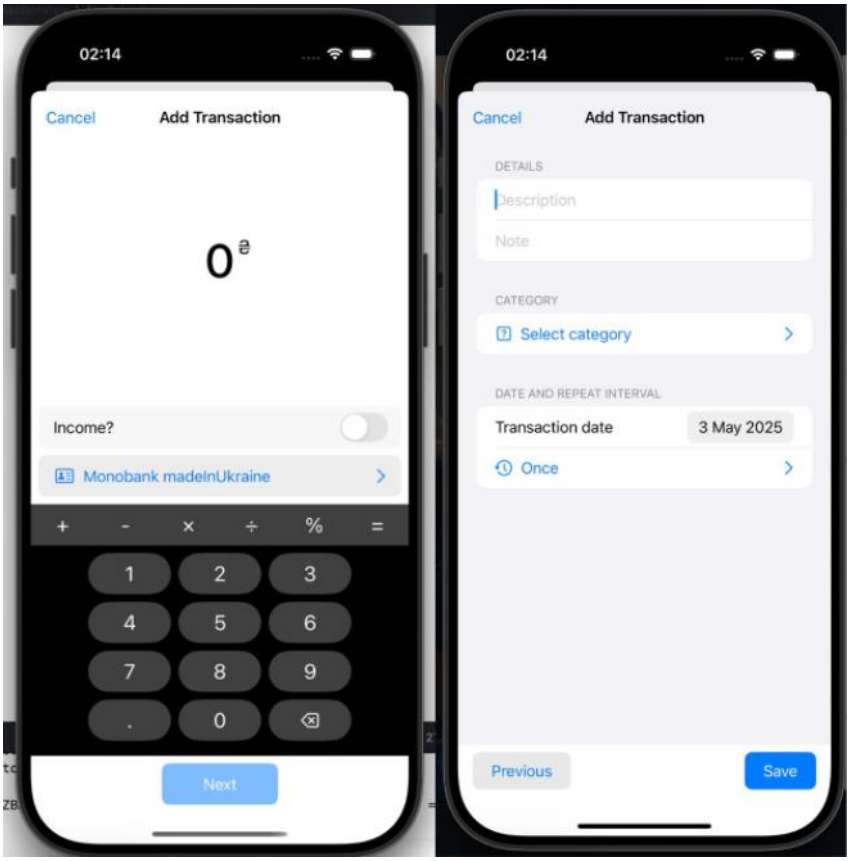


*Рисунок 3.6.1.2 – Інтерфейс ручного вводу*

Робота з усіма полями відбувається у ViewModel. ViewModel містить у собі усі поля, а також посилання на репозиторій. При збереженні транзакції відповідний метод ViewModel викликає метод репозиторію для збереження транзакції.

### 3.6.2. Інтерфейс додавання зовнішніх рахунків.

Для автоматичного імпорту транзакцій передбачений інтерфейс, що містить список доступних сервісів. Як зазначалось раніше, в рамках проекту інтегроване API від Monobank. Інтерфейс виглядає наступним чином:

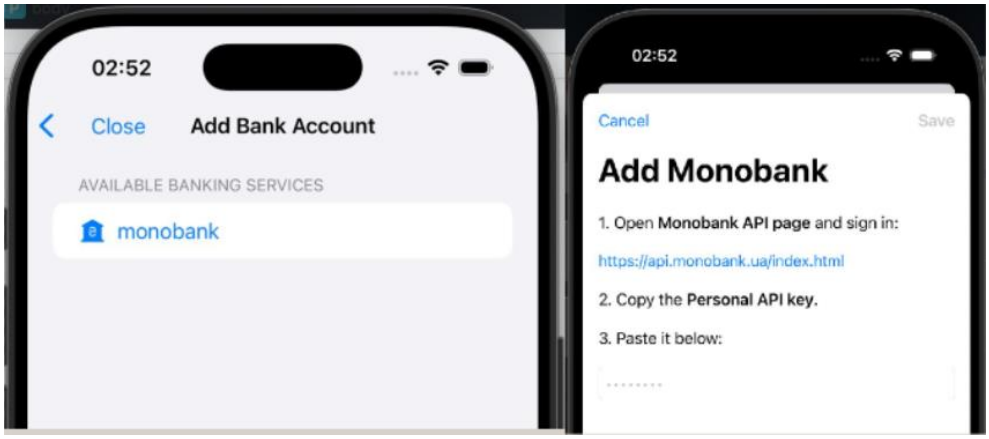


*Рисунок 3.6.2 – Інтерфейс додавання зовнішніх рахунків*

Для токену використовується SecureField, що приховує введені символи. Такий підхід є стандартом для полей, що містять чуттєву інформацію.

### 3.6.3. Інтерфейс сканування чеків.

Інтерфейс розділений на кілька екранів:

- Екран-список - має кнопки для управління інтерфейсом, а також відображає поточний стан імпорту:

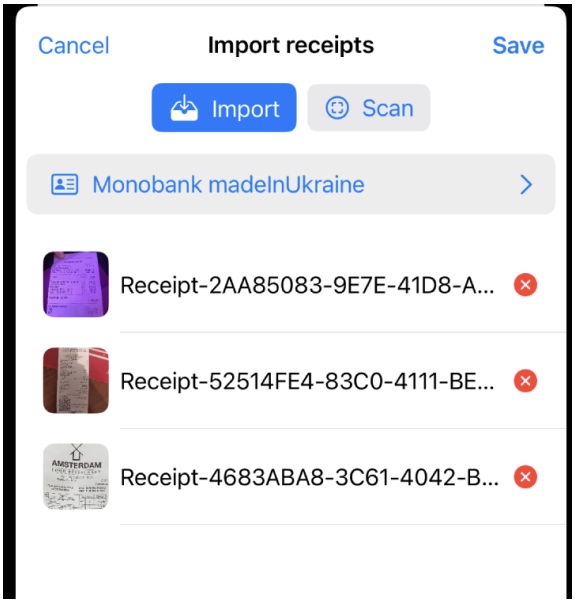


*Рисунок 3.6.3.1 – Інтерфейс списку чеків*

Після сканування також відображає інформацію з кожного чеку:

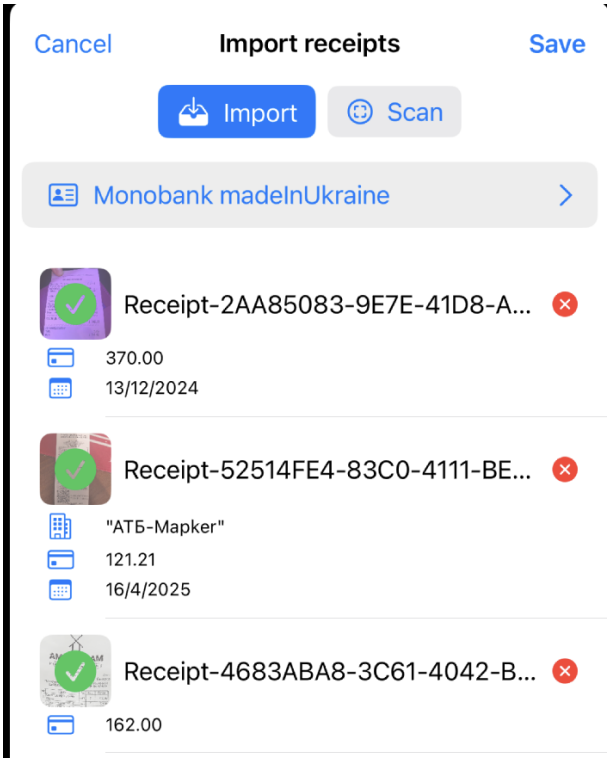


*Рисунок 3.6.3.2 – Приклад роботи сканування*

- Екран імпорту зображень – вбудоване рішення від Apple, надає можливість обрати одну чи кілька фотографій з галереї.
- Екран перегляду відсканованої фотографії – надає можливість передивитися та редагувати інформацію методом настику на екран, різні типи інформації підсвічені різними кольорами:

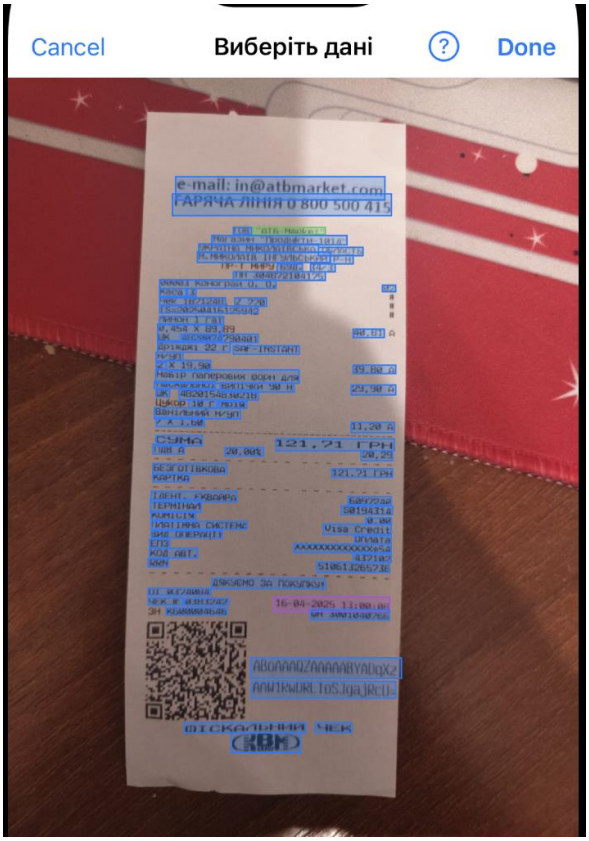


*Рисунок 3.6.3.3 – Інтерфейс екрану перегляду чека*

Як тільки сканування завершено, є можливість зберегти всю інформацію в застосунку. Використовується метод create з репозиторію.

### 3.6.4. Інтерфейс редагування та видалення транзакції.

Логіка редагування та видалення транзакції схожа на додавання:

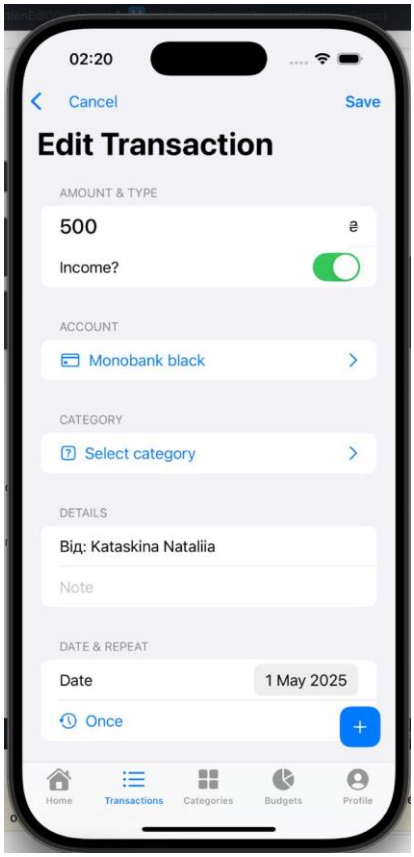


*Рисунок 3.6.4 – Інтерфейс редагування транзакції*

Користувач має можливість редагувати всі поля транзакції. ViewModel, у свою чергу, запустить відповідний метод репозиторію та збереже дані у базі даних. Також кнопка Delete запустить метод видалення транзакції.

### 3.6.5. Інтерфейс перегляду транзакцій.

Перегляд транзакцій реалізований за допомогою відфільтрованого списку у SwiftUI. Екран містить поле для фільтру по назві, даті і категорії:

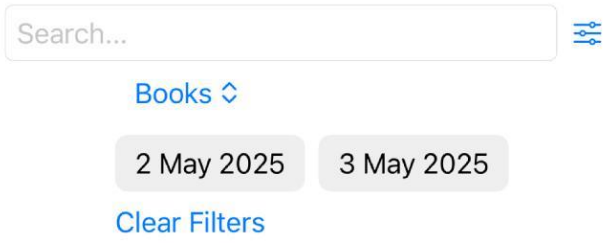


*Рисунок 3.6.5.1 – Інтерфейс фільтрації*

ViewModel містить метод, який фільтрує транзакції, отримані з репозиторію, після чого передає їх до View. Відповідно так виглядає відображення всіх транзакцій:

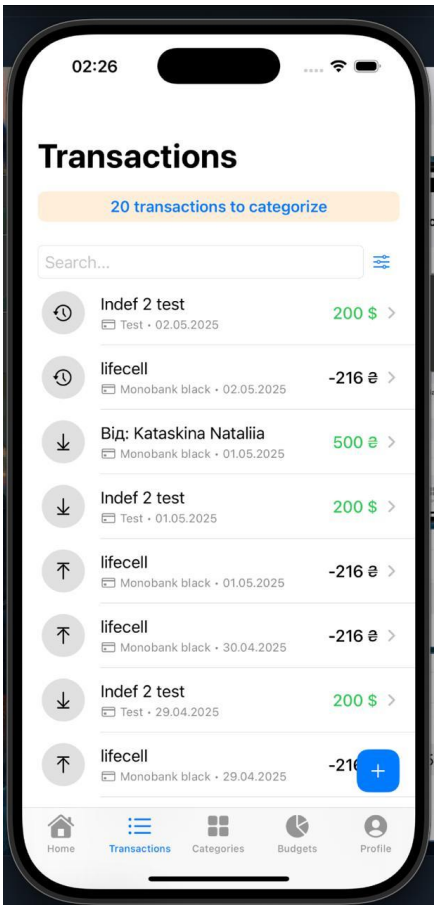


*Рисунок 3.6.5.2 – Інтерфейс перегляду транзакцій*

Продуваний інтерфейс категоризації витрат, які ще не мають категорії. При натиску відкривається меню, що представляє собою чергу з транзакції, яка просувається при визначенні категорії.

При натиску на саму транзакцію в списку відкривається меню редагування. При натиску на кнопку «+» відкривається меню додавання нової транзакції.

### 3.6.6. Інтерфейс категорій транзакцій.

Оскільки транзакції мають категорію, був створений інтерфейс перегляду та додавання нових категорій. Виглядає він так:

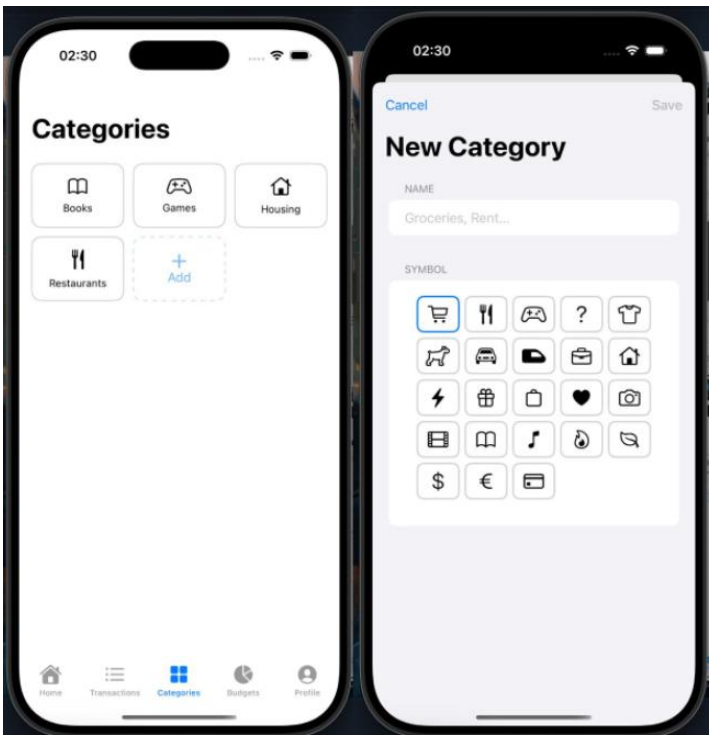


*Рисунок 3.6.6 – Інтерфейси категорій*

### 3.6.7. Інтерфейс бюджетування.

Важливою функцією застосунку є бюджетування, для цього створений окремий інтерфейс додавання бюджету, який містить такі поля: ліміт, назва, рахунки, інтервал, категорії, ліміти кожної категорії:

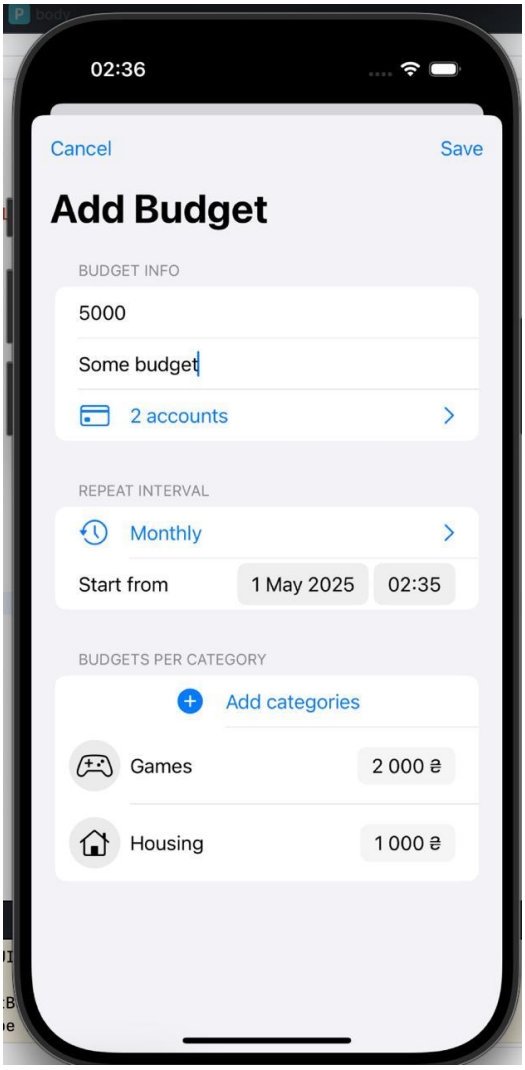


*Рисунок 3.6.7.1 – Інтерфейс додавання бюджету*

Схожим чином реалізований інтерфейс редагування бюджету – відрізняється тільки наявністю кнопки видалення бюджету.

Перегляд бюджетів також реалізований списком. Кожен елемент виглядає наступним чином:

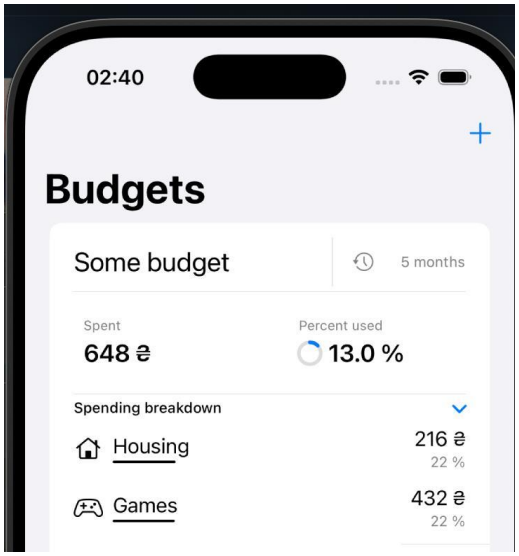


*Рисунок 3.6.7.2 – Інтерфейс списку бюджетів*

### 3.6.8. Інтерфейс рахунків.

Як і інші елементи програми, рахунки відображаються списком, а також мають інтерфейс додавання і редагування. Виглядають вони ось так:

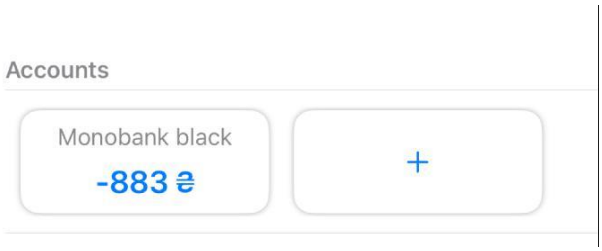


*Рисунок 3.6.8.1 – Інтерфейс списку рахунків*

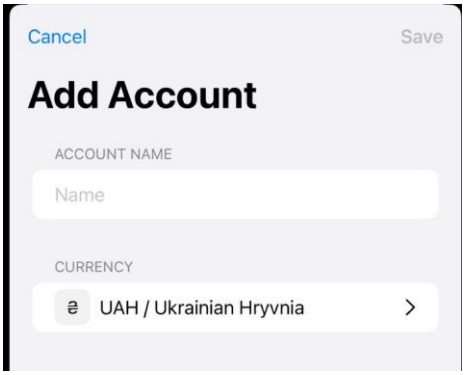


*Рисунок 3.2.1.1 – Інтерфейс додавання рахунку*

Так як рахунки містять в собі список транзакцій, також розроблено інтерфейс, який відображає транзакції виключно обраного рахунку:

### 3.6.9. Інтерфейс регулярних транзакцій та бюджетів

Транзакції та бюджети можуть повторюватися, тому для них реалізовано спеціальний інтерфейс, який дозволяє обрати необхідні налаштування повтору:

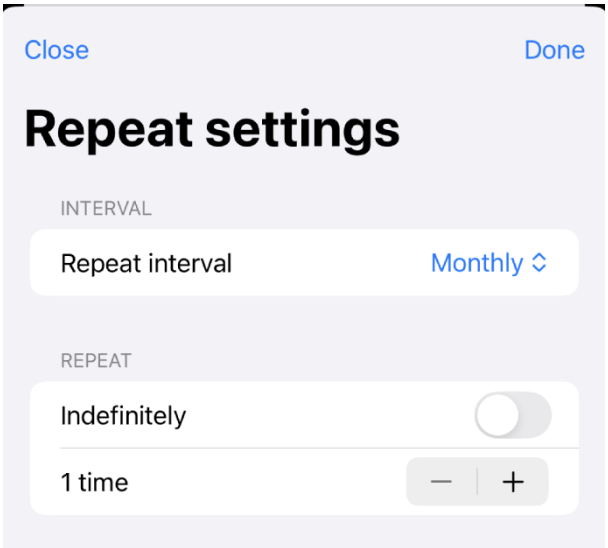


*Рисунок 3.6.9 – Інтерфейс налаштування повторів*

## 3.7. Висновки розробки застосунку.

Отже, в рамках цієї роботи був розроблений застосунок, що відповідає мінімальним вимогам, які описані в попередніх пунктах роботи. Варто зазначити, що даний застосунок є прототипом, який не готовий до використання кінцевим користувачем.

Головною задачею було дослідити можливі функції застосунків для управління фінансами та процес створення цих функцій. Так був повністю налагоджений процес управління транзакціями, а також реалізовано прототипи методів автоматизацій вводу транзакцій.

Завданням було і налагодження відповідної архітектури для застосунку, що оперує даними, які постійно змінюються і потребують збереження водночас. Так було реалізовано застосунок в реактивній парадигмі, що дозволило реалізувати «одне джерело правди» і налагодити потік даних з багатьох джерел: база даних, запити до мережі, запити до аналізатора тексту тощо. Для більшої структурованості було використано паттерн MVVM, що дозволив ефективно інкапсулювати логіку та забезпечити можливість розширення застосунку без переписування існуючих елементів.

## Висновки

Дослідження предметної області показало, що українські користувачі потребують єдиного мобільного середовища для обліку всіх безготівкових і готівкових розрахунків. Аналіз популярних фінансових застосунків виявив відсутність рішення, яке б задовільнило більшість виставлених вимог. Отже, висунуто вимоги до власного рішення проблеми: імпорт транзакцій напряму з банків, сканування транзакцій з фотографій, підтримка багатьох валют, бюджетування, забезпечення безпечного зберігання даних.

З огляду на виставлені вимоги аргументовано вибір стеку: Swift, SwiftUI, Combine, Core Data, Keychain, Apple Vision, CoreML. В якості шаблону використано MVVM + Repository.

Створено прототип iOS-застосунку, що реалізує такий набір функцій:

- CRUD-операції з транзакціями, рахунками, категоріями та бюджетами
- Інтеграцію Monobank open API
- OCR-сканер чеків і текстовий класифікатор, що використовує машинне навчання
- Face ID-аутентифікація
- Потік даних у реактивній парадигмі.

**Перспективи вдосконалення роботи:**

- **Розвиток інтеграції з банками** – після впровадження Open Banking API (01.08.2025 р.) з’явиться можливість інтеграції інших українських банків.

- **Покращення класифікації тексту** – наразі реалізовано концепт сканування чеків, однак, збільшивши кількість зразків для тренування моделі, можна досягти повної автоматизації процесу сканування з достатньою для використання точністю.
- **Покращення UI/UX застосунку** – наразі реалізовано базові функції програми, однак відкритим залишається питання реалізації зручного інтерфейсу, відображення звітів, графіків тощо.

Отже, у процесі виконання роботи створено прототип застосунку для управління фінансами, що пропонує методи заповнення прогалин існуючих рішень, підтверджено працездатність обраної архітектури й закладено основу для подальшого масштабування. Розробка демонструє, що поєднання SwiftUI, Combine, Core Data, Vision та Core ML дає змогу реалізувати застосунок, що повною мірою охоплює процес управління фінансами.

**Список використаних джерел**


1. Дашборд monobank. *monobank – мобільний банк*. URL: https://monobank.ua/dashboard (дата звернення: 05.05.2025).
2. Національний банк України. Скільки рахунків відкрито в Україні?. *Facebook*. URL: https://www.facebook.com/NationalBankOfUkraine/posts/скільки-рахунків-відкрито-в-україні-проаналізували-дані-станом-на-початок-цього-/1076364317867547/ (дата звернення: 05.05.2025).
3. Що таке особисті фінанси і як ними правильно управляти: від комуналки до донатів на перемогу | Національний університет «Львівська політехніка». | *Національний університет «Львівська політехніка»*. URL: https://lpnu.ua/news/shcho-take-osobysti-finansy-i-iak-nymy-pravylno-upravliaty-vid-komunalky-do-donativ-na (дата звернення: 05.05.2025).
4. Personal Finance Mobile App Market Statistics | 2034. *Market Research Reports | Business Intelligence Consulting - Fact.MR*. URL: https://www.factmr.com/report/personal-finance-mobile-app-market (date of access: 05.05.2025).
5. Finance Apps: Definition, Types, Features, Development. *Intelivita UK*. URL: https://www.intelivita.co.uk/blog/finance-apps (date of access: 05.05.2025).
6. McMullen L. The Best Budget Apps for 2025. *NerdWallet*. URL: https://www.nerdwallet.com/article/finance/best-budget-apps (date of access: 05.05.2025).
7. AI in Finance: Applications, Examples & Benefits | Google Cloud. *Google Cloud*. URL: https://cloud.google.com/discover/finance-ai (date of access: 05.05.2025).

8. Вітка Ю. Фінансова грамотність та фінансовий добробут українців: який стан та що робити?. *Національний банк України*. URL: https://events.bank.gov.ua/nbuexpress2019/src/files/Фінансова%20грамотність.pdf (дата звернення: 05.05.2025).
9. Самойлюк М. Трекер економіки України під час війни. *Центр економічної стратегії*. URL: https://ces.org.ua/tracker-economy-during-the-war/ (дата звернення: 05.05.2025).
10. Курс НБУ - USD (долар США). *Мінфін*. URL: https://index.minfin.com.ua/ua/exchange/nbu/curr/usd/ (дата звернення: 05.05.2025).
11. Lee S. 10 Must-Try Features in Personal Finance Apps Today. *Number Analytics - AI Statistical Software*. URL: https://www.numberanalytics.com/blog/must-try-features-personal-finance-apps-today (date of access: 05.05.2025).
12. Бондар К. 50% фінансових застосунків в Україні є російськими, – дослідження - Bazilik Media. *Bazilik Media*. URL: https://bazilik.media/50-finansovykh-zastosunkiv-v-ukraini-ie-rosijskymy-doslidzhennia/ (дата звернення: 05.05.2025).
13. Mastercard. Дослідження Mastercard: 51% українців готові користуватися виключно цифровим банкінгом. *Mastercard*. URL: link (дата звернення: 05.05.2025).
14. Open Banking: що це таке і як вплине на фінтех-ринок України. *Fondy. Сучасні платіжні рішення для вашого бізнесу*. URL: https://fondy.ua/uk/knowledge/open-banking/ (дата звернення: 05.05.2025).
15. Monobank open API. *Monobank*. URL: https://api.monobank.ua/docs/index.html (date of access: 05.05.2025).

16. Money Manager & Budget Planner | Spendee. *Money Manager & Budget Planner | Spendee*. URL: https://www.spendee.com/ (date of access: 05.05.2025).
17. Spendee. The Ultimate Guide to Spendee Subscriptions. *Medium*. URL: https://spendeeapp.medium.com/the-ultimate-guide-to-spendee-subscriptions-e00d5a261d14 (date of access: 05.05.2025).
18. How to be a financially successful person? | Money Lover. *Moneylover*. URL: https://moneylover.me/ (date of access: 05.05.2025).
19. JSC F. Money Lover: Expense Manager. *App Store*. URL: https://apps.apple.com/us/app/money-lover-expense-manager/id486312413 (date of access: 05.05.2025).
20. Swift - Apple Developer. *Apple Developer*. URL: https://developer.apple.com/swift/ (date of access: 05.05.2025).
21. Jeroen L. UIKit vs. SwiftUI - Choosing the right framework!. *Scalable Feeds, Chat, & Video - Powerful APIs and Components by Stream*. URL: https://getstream.io/blog/uikit-vs-swiftui/ (date of access: 05.05.2025).
22. Muhamad. UIKit vs. SwiftUI: Imperative vs. Declarative Paradigms. *Medium*. URL: https://medium.com/@muhamad99t/uikit-vs-swiftui-imperative-vs-7f627ba79b7d (date of access: 05.05.2025).
23. About App Development with UIKit | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/uikit/about-app-development-with-uikit (date of access: 05.05.2025).
24. Uğur G. What is Reactive Programming in Swift?. *Medium*. URL: https://gayeugur.medium.com/what-is-reactive-programming-in-swift-c35fd9fd4af0 (date of access: 05.05.2025).
25. Combine vs. RxSwift: Should you switch to Combine?. *QuickBird Studios*. URL: https://quickbirdstudios.com/blog/combine-vs-rxswift/ (date of access: 05.05.2025).

26. Monteiro T. N. MVVM in iOS: A Clean Approach to Scalable Apps. *LinkedIn: Log In or Sign Up*. URL: https://www.linkedin.com/pulse/mvvm-ios-clean-approach-scalable-apps-thiago-nunes-monteiro-izzkf (date of access: 05.05.2025).
27. Securely Managing User Details in the Keychain on iOS: Part I Web Development, Software, and App Blog | 200OK Solutions. *Web Development, Software, and App Blog | 200OK Solutions*. URL: https://200oksolutions.com/blog/securely-managing-user-details-in-the-keychain-on-ios-part-i/ (date of access: 05.05.2025).
28. GitHub - kishikawakatsumi/KeychainAccess: Simple Swift wrapper for Keychain that works on iOS, watchOS, tvOS and macOS. *GitHub*. URL: https://github.com/kishikawakatsumi/KeychainAccess (date of access: 05.05.2025).
29. Data Protection classes. *Apple Support*. URL: https://support.apple.com/en-om/guide/security/secb010e978a/web (date of access: 05.05.2025).
30. GitHub - siomochkin/monobank-open-api-documentation: Monobank Open API (v2303) Documentation. *GitHub*. URL: https://github.com/siomochkin/monobank-open-api-documentation (date of access: 05.05.2025).
31. Recognizing Text in Images | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/vision/recognizing-text-in-images (date of access: 05.05.2025).
32. Creating a text classifier model | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/createml/creating-a-text-classifier-model (date of access: 05.05.2025).

33. Team Kodeco. SwiftUI Cookbook, Chapter 5: Integrate Core Data With SwiftUI. *kodeco.com*. URL: https://www.kodeco.com/books/swiftui-cookbook/v1.0/chapters/5-integrate-core-data-with-swiftui (date of access: 05.05.2025).
34. Qoulta B. iOS Core Data With Combine. *Medium*. URL: https://medium.com/swlh/ios-core-data-with-combine-c80373c5484 (date of access: 05.05.2025).
35. GitHub - Alamofire/Alamofire: Elegant HTTP Networking in Swift. *GitHub*. URL: https://github.com/Alamofire/Alamofire (date of access: 05.05.2025).
36. dataTaskPublisher(for:) | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/foundation/urlsession/datataskpublisher(for:)-5kiir (date of access: 05.05.2025).
37. flatMap(maxPublishers:_:) | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/combine/publisher/flatmap(maxpublishers:_:)-3k7z5 (date of access: 05.05.2025).
38. Encoding and Decoding Custom Types | Apple Developer Documentation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/foundation/encoding-and-decoding-custom-types (date of access: 05.05.2025).
39. Create ML - Machine Learning - Apple Developer. *Apple Developer*. URL: https://developer.apple.com/machine-learning/create-ml/ (date of access: 05.05.2025).