Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Бакалавр
(назва освітнього ступеня)

на тему: **Розробка та тестування програмного застосунку на основі платформи .NET для обліку та планування особистих фінансових витрат**

Виконала: студентка IV курсу, групи СП-42
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) Шульга А.О. (прізвище та ініціали)

Керівник (підпис) Петрик М.Р. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю.М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М.Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль 2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
__________ Петрик М.Р.
(підпис) (прізвище та ініціали)
« 06 » квітня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Студенту Шульзі Анастасії Олександрівні
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка та тестування програмного застосунку на основі платформи .NET для обліку та планування особистих фінансових витрат

Керівник роботи Петрик Михайло Романович, доктор фізико-математичних наук, професор
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 06 » квітня 20 26 року № ______

2. Термін подання студентом завершеної роботи 22.06.2026

3. Вихідні дані до роботи Методичні вказівки, рекомендації кафедри, наукові публікації, технічна документація з розробки програмного забезпечення

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз предметної області
2. Проєктування застосунку для обліку та планування фінансових витрат
3. Реалізація та тестування застосунку
4. Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, слайдів)
Ілюстрації, схеми, діаграми, інтерфейси реалізації застосунку, слайди презентації: актуальність теми; мета та завдання роботи, аналіз існуючих рішень; типовий цикл роботи з фінансами; сценарії взаємодії користувача з системою; функціональні та нефункціональні вимоги; архітектура застосунку; діаграма класів підсистеми транзакцій; моделювання поведінки системи; проєктування бази даних; реалізація структури рішення; організація роботи з даними та бізнес-логікою; екрани реалізованого застосунку; тестування системи; висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	к.т.н., доцент Мариненко С.Ю.		
Нормоконтроль	к.т.н., доц. каф. ПІ Стоянов Ю.М.		

7. Дата видачі завдання 6 квітня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Розробка технічного завдання*	*6.04 – 12.04*	*виконано*
2.	*Робота над першим розділом «Аналіз предметної області»*	*13.04 – 26.04*	*виконано*
3.	*Робота над другим розділом «Проєктування застосунку для обліку та планування особистих фінансових витрат Spendly»*	*27.04 – 03.05*	*виконано*
4.	*Робота над третім розділом «Реалізація та тестування застосунку Spendly»*	*04.05 – 17.05*	*виконано*
5.	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*18.05 – 24.05*	*виконано*
6.	*Оформлення пояснювальної записки і графічного матеріалу*	*25.05 – 7.06*	*виконано*
7.	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*	*виконано*
8.	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*	*виконано*
9.	*Захист кваліфікаційної роботи бакалавра*		

Студент ______________ Шульга А.О.
(підпис) (прізвище та ініціали)

Керівник роботи ______________ Петрик М.Р.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка та тестування програмного застосунку на основі платформи .NET для обліку та планування особистих фінансових витрат // Кваліфікаційна робота освітнього рівня «Бакалавр» // Шульга Анастасія Олександрівна // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп’ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2026 // с. – 73 , рис. – 32, табл. – 7, додат. – 1, бібліогр. – 35.

Ключові слова: Ключові слова: .NET, WPF, MVVM, MediatR, Entity Framework Core, SQLite, облік витрат, особисті фінанси, фінансова аналітика, бюджет.

У кваліфікаційній роботі бакалавра виконано проєктування та реалізацію програмного застосунку для обліку та планування особистих фінансових витрат. Продукт призначений для ведення фінансової історії користувача, аналізу доходів і витрат та контролю бюджету за категоріями.

У ході роботи проведено аналіз предметної області та існуючих рішень, визначено їх переваги й недоліки, сформовано вимоги до системи та обґрунтовано архітектурний підхід. Застосунок реалізовано на платформі .NET як настільний продукт з графічним інтерфейсом. Спроєктовано багатошарову архітектуру з розподілом на рівні представлення, бізнес-логіки та доступу до даних. Для організації взаємодії компонентів використано підхід, близький до CQRS. Функціональність застосунку охоплює створення, редагування та видалення фінансових операцій, їх фільтрацію і пошук, ведення категорій доходів і витрат, формування аналітичних показників та візуалізацію даних за допомогою графіків.

Об’єктом дослідження є процес управління особистими фінансами користувача. Предметом дослідження є методи та програмні засоби реалізації систем обліку й аналізу фінансових даних.

ABSTRACT

Development and Testing of a .NET-Based Software Application for Personal Financial Expense Tracking and Planning // Shulha Anastasiia Oleksandrivna // Ternopil Ivan Puluj National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, Group SP-42 // Ternopil, 2026 // p. – 73, fig. – 29, tab. – 7, app. – 1, bibl. – 35.

Keywords: .NET, WPF, MVVM, MediatR, Entity Framework Core, SQLite, expense tracking, personal finance, financial analytics, budgeting.

In this bachelor's qualification thesis, the design and implementation of a software application for tracking and planning personal financial expenses are presented. The developed product is intended for maintaining a user's financial history, analyzing income and expenses, and controlling the budget by categories.

During the research, the subject area and existing solutions were analyzed, their advantages and disadvantages were identified, system requirements were defined, and the architectural approach was justified. The application is implemented on the .NET platform as a desktop application with a graphical user interface. A multi-layered architecture has been designed, including separation into presentation, business logic, and data access layers. A CQRS-like approach is used to organize interaction between components. The functionality of the application includes creating, editing, and deleting financial transactions, filtering and searching them, managing income and expense categories, generating analytical indicators, and visualizing data using charts.

The object of research is the process of personal financial management. The subject of research is methods and software tools for implementing systems for financial data tracking and analysis.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

.NET – Microsoft .NET (платформа для розробки програмного забезпечення)

WPF – Windows Presentation Foundation (технологія створення настільних застосунків для Windows)

MVVM – Model–View–ViewModel (архітектурний шаблон розділення інтерфейсу та логіки застосунку)

CQRS – Command Query Responsibility Segregation (патерн розділення операцій читання та зміни даних)

UI – User Interface (інтерфейс користувача)

EF Core – Entity Framework Core (фреймворк доступу до баз даних для .NET)

SQLite – Structured Query Language Lite (вбудована реляційна система керування базами даних)

CRUD – Create, Read, Update, Delete (основні операції роботи з даними)

LINQ – Language Integrated Query (технологія виконання запитів до даних у .NET)

UML – Unified Modeling Language (уніфікована мова моделювання програмних систем)

CSV – Comma-Separated Values (текстовий формат зберігання табличних даних)

KPI – Key Performance Indicator (ключовий показник ефективності)

PFM – Personal Finance Management (управління особистими фінансами)

AES – Advanced Encryption Standard (стандарт шифрування даних)

2FA – Two-Factor Authentication (двофакторна автентифікація)

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10

1.1 Загальна характеристика предметної області та огляд існуючих рішень на сучасному ринку 10

1.2 Актанти системи та діаграма варіантів використання 17

1.3 Функціональні та нефункціональні вимоги до застосунку 18

1.4 Висновок до першого розділу 25

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ ДЛЯ ОБЛІКУ ТА ПЛАНУВАННЯ ОСОБИСТИХ ФІНАНСОВИХ ВИТРАТ SPENDLY 27

2.1 Архітектура системи та діаграми класів 27

2.2 Моделювання поведінки системи: діаграми послідовності, діяльності та станів 31

2.3 Проєктування бази даних 36

2.4 Висновок до другого розділу 40

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ SPENDLY 42

3.1 Реалізація шарів даних та бізнес-логіки 42

3.2 Опис екранів та інтерфейсу застосунку 48

3.3 Тестування системи 53

3.4 Висновки до третього розділу 60

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ 62

4.1 Актуальність безпеки життєдіяльності людини 67

4.2 Охорона праці 64

ВИСНОВКИ 66

ПЕРЕЛІК ДЖЕРЕЛ 68

ВСТУП

Щоденна активність людини охоплює численні фінансові операції: оплату послуг, грошові перекази, придбання товарів. Із зростанням кількості дій контроль витрат ускладнюється, а відстеження структури потребує більше часу. Частина користувачів не веде облік узагалі. Інші фіксують витрати епізодично. У результаті бюджет планують із похибками, фінансова картина втрачає цілісність.

Сучасні програмні продукти підтримують роботу з фінансами. Водночас чимало застосунків перевантажують інтерфейс або вимагають постійного доступу до мережі. У деяких умовах інтернет недоступний. Дані потрібні негайно. Паралельно виникають ризики для фінансової інформації, адже користувачі оперують персональними відомостями. Користувач очікує простий інструмент без надлишкових функцій. Зручність взаємодії визначає ефективність роботи. Інтерфейс має швидко приводити до результату: внесення витрат або перегляд статистики. Окремою вимогою виділяється автономна робота без зовнішніх сервісів. Перелічені умови формують підхід до розроблення систем обліку витрат.

Метою кваліфікаційної роботи є створення й тестування програмного застосунку на платформі .NET для обліку та планування особистих витрат. Розроблений продукт зберігає фінансові записи й надає користувачу інструменти їх аналізу. Користувач переглядає структуру витрат і відстежує зміни в обраному часовому інтервалі. Особливу увагу приділено зручності інтерфейсу та логіці взаємодії. Реалізація поставленої мети передбачає низку завдань: аналіз предметної області та огляд наявних рішень; формування вимог до програмного продукту; проєктування архітектури застосунку; створення модулів для роботи з транзакціями, бюджетом і аналітичними даними; побудову системи зберігання; розроблення засобів візуалізації; тестування роботи програми.

Кожен етап формує підсумковий результат. Система об’єднує введення даних, їх структуризацію та подальший аналіз. Компоненти працюють узгоджено. Рішення цілісне.

Об’єкт дослідження кваліфікаційної роботи охоплює взаємодію користувача з особистими фінансами, включно з обліком доходів і витрат та подальшим аналізом.

Предмет дослідження визначає підходи до створення програмних систем для обліку, аналізу й відображення фінансових даних на платформі .NET.

У роботі використано архітектуру MVVM разом із CQRS-подібною організацією логіки. Така побудова розділяє інтерфейс і обробку даних. Розробник отримує зрозумілу структуру коду. Застосунок поєднує облік і аналітику в одному середовищі. Користувач бачить не лише перелік операцій. Дані групуються за категоріями. Динаміка витрат відображається наочно. Контроль бюджету стає прозорим.

Результатом роботи є повноцінний програмний застосунок. Користувач додає записи, переглядає статистику та аналізує витрати. Подальший розвиток можливий. Перспективні напрями включають інтеграцію з банківськими сервісами та розширення аналітичних інструментів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

На сьогоднішній день нерегулярний облік витрат і занепокоєність щодо збереження персональних даних залишаються головними викликами для користувачів сучасних фінансових інструментів.

У першому розділі розглядаються наявні рішення на ринку, виявляються їхні переваги та обмеження, а також формуються вимоги до застосунку Spendly.

1.1 Загальна характеристика предметної області та огляд існуючих рішень на сучасному ринку

Предметна область особистого фінансового менеджменту охоплює кілька взаємопов'язаних задач: фіксацію доходів і витрат, розподіл операцій за категоріями, контроль виконання бюджету та аналіз динаміки за обраний період. У науковій літературі вона визначається через процеси планування, бюджетування, оцінювання та управління фінансами задля досягнення поставлених цілей.

Програмні рішення підтримують перелічене через транзакції, категорії, фільтри, звіти та бюджети. Найгострішою практичною проблемою стає нерегулярність ведення записів, а не брак відповідних інструментів. Витрати відкладаються на потім, дрібні операції губляться, щоденне документування вимагає стійкої звички. Опитування серед 60 студентів STMIK Mardira Indonesia підтвердило факт кількісно: 61,4% назвали несистематичний облік головною перешкодою, 21,4% – брак часу на фіксацію [1].

Окремо важливим залишається захист персональних даних: фінансові записи належать до чутливої категорії інформації, тому підхід до зберігання суттєво впливає на довіру до застосунку. Мобільний і настільний формати орієнтовані на різні сценарії роботи з фінансами. Оперативна фіксація витрат одразу після покупки є типовим мобільним сценарієм, що формує звичку регулярного обліку. Перегляд сотень записів, налаштування складних фільтрів,

побудова звітів за кілька місяців – задачі, де настільна програма відчутно зручніша. Дослідження Stefanov et al. (2024) підтверджує: ядро подібних систем будується навколо реляційної бази з таблицями транзакцій і категорій, а звітність та графіки витрат входять до обов'язкового функціонального мінімуму [2].

Наведений контекст безпосередньо пояснює вибір технологічного стеку в роботі. Для наочного відображення структури витрат і відхилень від бюджету добре підходить настільний формат на .NET із WPF. Середовище нативно підтримує таблиці з фільтрацією, форми введення та графіки. Патерн MVVM розділяє логіку обліку та інтерфейс, що спрощує тестування компонентів і подальший розвиток застосунку. Локальне зберігання засобами SQLite і EF Core виключає залежність від мережі й узгоджується з вимогами до захисту фінансових даних користувача.

Ринок застосунків для аналізу особистих фінансів давно вийшов за межі простого списку витрат. Частина продуктів працює як банківський застосунок із вбудованою статистикою, частина вчить користувача бюджетувати, інша частина збирає рахунки з різних банків і будує ширшу фінансову картину. Для дипломної теми важливо дивитися не лише на перелік функцій. Не менше значать модель зберігання даних, робота без мережі, глибина аналітики та рівень контролю над приватною інформацією. Наукові публікації теж показують практичний ефект таких інструментів: використання фінансових застосунків підвищує регулярність обліку, покращує фінансову дисципліну та робить поведінку користувача більш усвідомленою [3].

YNAB будує роботу не навколо банку, а навколо бюджетного методу. Сервіс підключає рахунки до банку, автоматично імпортує операції, синхронізує дані між комп’ютером, телефоном і планшетом навіть у режимі offline, підтримує спільне ведення бюджету, цілі, планувальник боргу та звіти за витратами і net worth [4]. Продукт орієнтується на користувача, який готовий регулярно працювати з категоріями й планувати витрати наперед. Тут важить дисципліна. Виграш полягає у глибокому бюджетуванні, сильній аналітиці та

прозорому розділенні пріоритетів. Додатковий плюс дають безрекламна модель, шифрування, 2FA та акцент на захисті даних.

У слабких місцях картина теж проста: YNAB просить від користувача більше уваги, ніж банківський застосунок. Частину рішень людина підтверджує сама. Логіка "кожен долар має роботу" підходить не всім. Для швидкого пасивного перегляду витрат сервіс інколи видається занадто вимогливим, зате для бюджетного планування випереджає більшість банківських рішень.

Головну сторінку сервісу продемонстровано на рисунку 1.1.

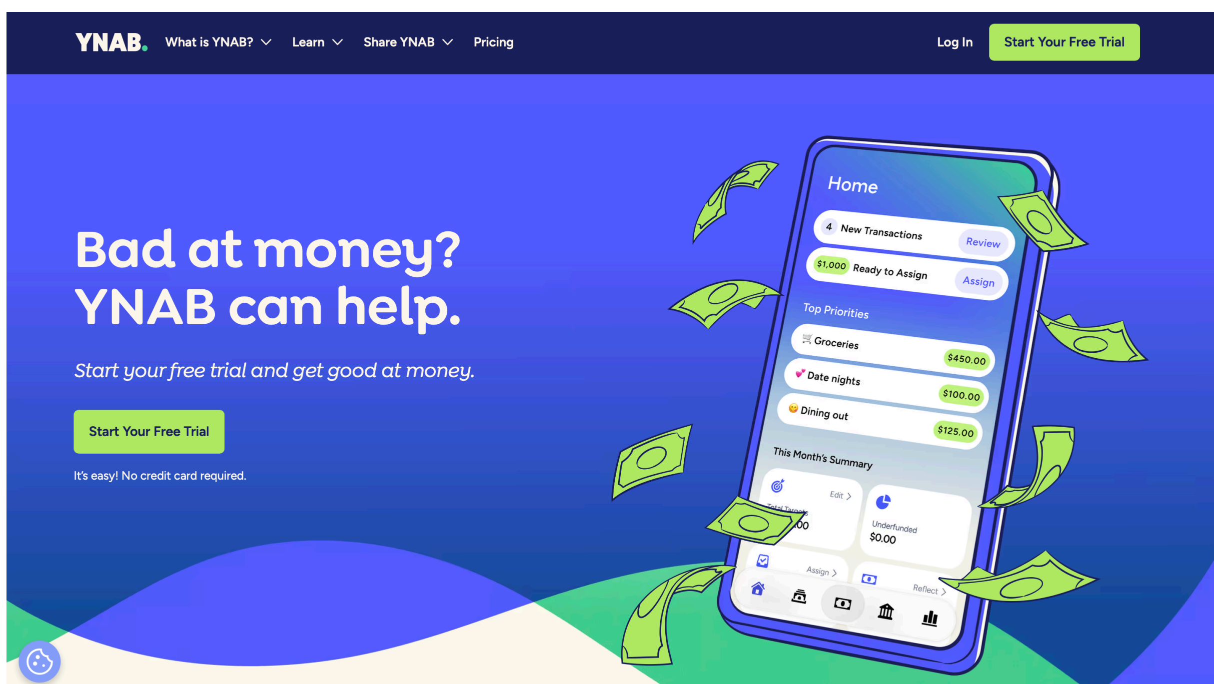

Рисунок 1.1 – Головна сторінка сервісу YNAB [4]

Wallet by BudgetBakers. Wallet ближче до класичних PFM-платформ. Продукт поєднує гнучкі бюджети, автоматичну категоризацію, нагадування про платежі, cash-flow-аналітику, детальні графіки, сімейний режим, мультивалютність, offline mode і синхронізацію між web, iOS та Android; окремо сервіс підключає понад 15 000 банків і тримає дані в актуальному стані [5]. Для користувача з кількома рахунками або регулярними витратами в різних валютах такий варіант виглядає сильним. Один екран збирає багато джерел. Перевага Wallet полягає у балансі між автоматизацією й ручним контролем: сервіс не

замикає людину на одному банку, але й не змушує вести все вручну. До плюсів належать 256-bit AES, 2FA, декларація відповідності GDPR, cloud backup і сімейне спільне використання [5].

Слабкі сторони теж помітні. Повний комфорт із bank sync найчастіше пов’язаний із платним функціоналом, а хмарне зберігання зменшує локальний контроль над записами порівняно з desktop-first програмами. Аналітика сильна, проте велика кількість функцій інколи перевантажує новачка вже на старті [5].

Головну сторінку сервісу продемонстровано на рисунку 1.2.

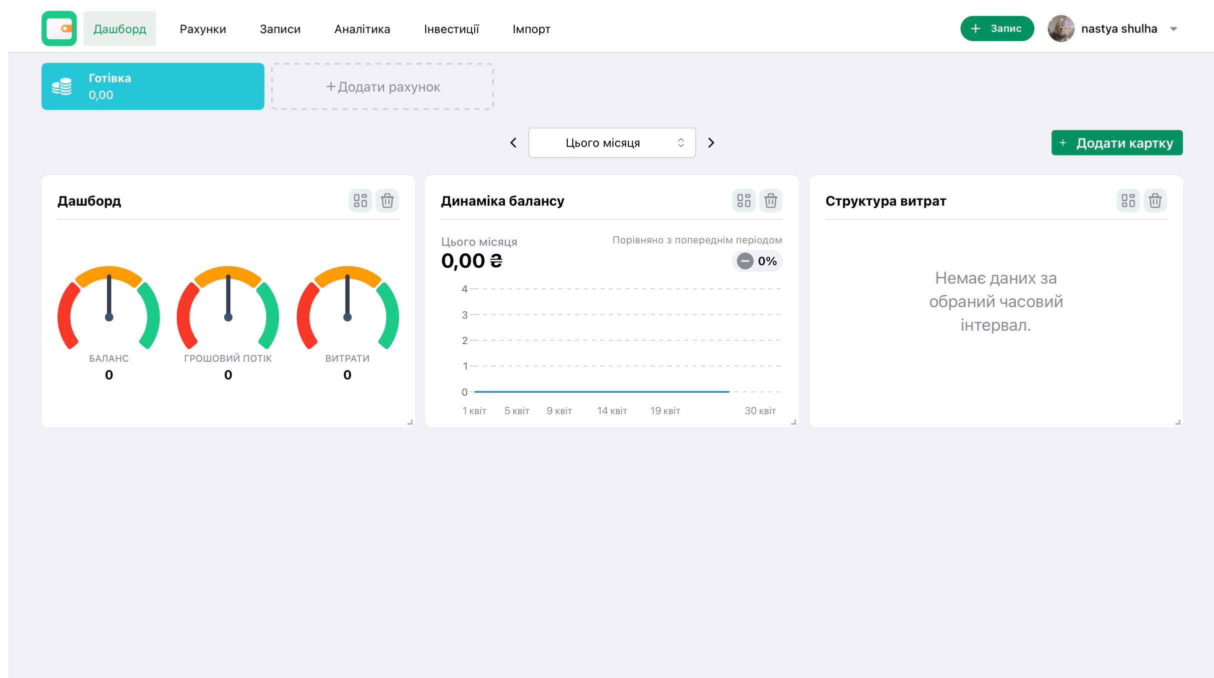

Рисунок 1.2 – Головна сторінка сервісу Wallet by BudgetBakers [5]

Money Manager Ex представляє інший полюс ринку. Проєкт працює як безкоштовна open-source desktop-програма, веде рахунки й валюти, показує транзакції в одному списку, підтримує пошук, фільтри, сортування і дає загальний погляд на фінансову картину [6]. Технічна модель тут інша. MMEX використовує SQLite-файл, а синхронізацію будує за принципом Bring Your Own Cloud: користувач сам обирає Dropbox, OneDrive, Google Drive, Nextcloud або інший канал і сам контролює конфлікти файлів між пристроями [6]. Для людей, які цінують автономну роботу та контроль над даними, підхід виглядає дуже

переконливо. Спочатку офлайн. Потім синхронізація. У такому сценарії продукт помітно сильніший за багато мобільних сервісів.Водночас ручної роботи більше: банк не підтягує транзакції так безшовно, як cloud-first рішення, а інтерфейс місцями виглядає стримано і навіть трохи старомодно. Зате саме MMEX найкраще показує, чому настільні рішення не зникли з ринку: локальна база, прозора архітектура і повний контроль над файлом даних для частини користувачів важать більше, ніж яскравий інтерфейс [6].

Головну сторінку сервісу продемонстровано на рисунку 1.3.

Рисунок 1.3 – Головна сторінка сервісу Money Manager Ex [6]

Quicken Simplifi by Quicken. Quicken Simplifi орієнтується на користувача, якому мало простого списку витрат. Офіційний опис робить акцент на бюджеті, звітах, real-time alerts, projected cash flows, інвестиціях і плануванні майбутніх рішень; сервіс працює у web та mobile і продається за моделлю підписки [7]. Через таку конфігурацію продукт ближчий до персонального фінансового центру, ніж до звичайного витратника. Він добре підходить людям, які хочуть бачити не лише минулі витрати, а й найближчий cash flow, резерв на рахунках і місце інвестицій у загальній картині. Саме тут лежить головна перевага Simplifi.

Для кваліфікаційної роботи такий підхід важливий, бо показує сучасний попит не лише на облік, а й на прогноз.

Слабкі місця теж зрозумілі: продукт покладається на хмарну інфраструктуру, не пропонує локально-орієнтовану модель настільної програми і підштовхує користувача до підписки. Отже, Simplifi корисний радше як функціональний орієнтир, а не як архітектурний шаблон для автономного desktop-застосунку [7].

Головну сторінку сервісу продемонстровано на рисунку 1.4.

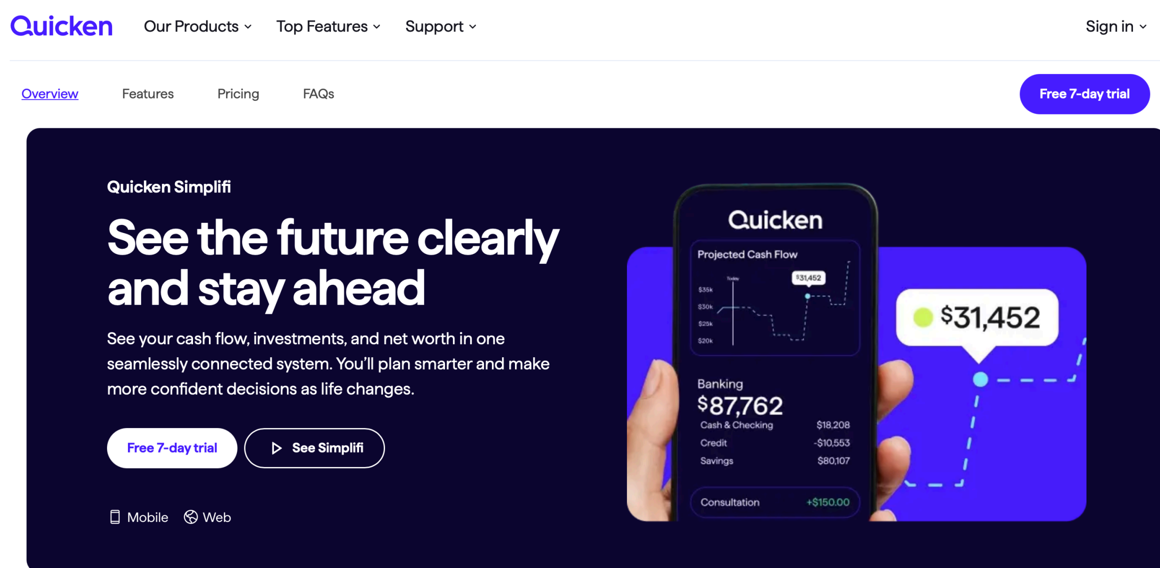

Рисунок 1.4 – Головна сторінка сервісу Quicken Simplifi by Quicken [7]

Порівняння показує доволі чіткий поділ ринку. Банківські продукти на зразок monobank і Revolut виграють в автоматичності, але програють у локальному контролі та гнучкості багатобанкового обліку. YNAB і Simplifi сильніше працюють із поведінкою користувача, цілями та прогнозом, але користувач приймає cloud-first модель і, як правило, модель підписки. Wallet займає проміжну позицію: агрегація, bank sync, аналітика, сімейні сценарії та часткова офлайн-робота зібрані в одному продукті. Money Manager Ex, навпаки, найкраще показує цінність desktop-first підходу: локальна база, автономність і

контроль над даними не втратили актуальності. Для кваліфікаційної роботи найцікавішими орієнтирами виглядають MMEX і Wallet. Перший підказує, як будувати автономний настільний інструмент. Другий показує, якого рівня зручності та аналітики вже чекає сучасний користувач. Банківські сервіси теж важливі. Вони нагадують, що автоматичний імпорт і проста статистика сильно підвищують щоденну залученість.

Нижче у таблиці 1.1 подано зведене порівняння ключових властивостей.

Таблиця 1.1 – Порівняння сучасних рішень у сфері бюджетування та обліку особистих фінансів

Рішення	**Ключові функції**	**Підтримка offline**	**Модель sync / зберігання**	**Приватність**
1	**2**	**3**	**4**	**5**
YNAB	бюджетування, цілі, bank import, спільний бюджет, борг, звіти	так	хмарна синхронізація між пристроями, імпорт із банків	ad-free, 2FA, шифрування
Wallet	бюджети, авто-категоризація, графіки, cash flow, сімейний доступ, bank sync	так, частково	хмарна синхронізація, банк-синк, backup	AES-256, 2FA, GDPR/ISO-підхід
Money Manager Ex	рахунки, валюти, транзакції, фільтри, звіти, локальний файл даних	так, повноцінно	локальна SQLite-база, користувацька хмара за потреби	максимальний локальний контроль
Quicken Simplifi	бюджет, звіти, alerts, cash-flow forecast, інвестиції, планування	обмежена	cloud-based web/mobile	приватність залежить від сервісної політики

Схема на рисунку 1.5 узагальнює типовий користувацький цикл, який реалізують більшість сучасних PFM-рішень і який доречно врахувати під час проєктування власного настільного застосунку.

Рисунок 1.5 – Схема типового користувацього циклу додатків для бюджетування та управління фінанасами

1.2 Актанти системи та діаграма варіантів використання

UML-актор (актант) описує роль, яка знаходиться поза межами програмної системи та взаємодіє з нею. Роль виконують користувачі, інші програмні продукти або технічні пристрої. Взаємодія відбувається через обмін даними або запитами, після чого система повертає результат обробки [8]. Окремий варіант

використання відображає конкретну ціль актора. Для досягнення результату актор виконує послідовність дій. Кожен сценарій пов'язаний із реальною задачею.

Діаграма варіантів використання поєднує окремі сценарії в межах однієї моделі. Така модель формує уявлення про можливості застосунку та ролі, що взаємодіють із системою [9]. Візуальне подання спрощує сприйняття. Зв'язки між сценаріями стають очевидними.

Spendly являє собою настільний застосунок на платформі .NET, який підтримує облік і планування особистих фінансів. У програмі відсутні механізми реєстрації та авторизації. Після запуску користувач одразу починає роботу. Інтерфейс відкривається без додаткових дій.

У моделі взаємодії присутній один актор – Користувач. Він додає фінансові операції, змінює записи або видаляє їх. Далі він працює з категоріями і рахунками, формує структуру витрат. За потреби користувач встановлює обмеження для категорій і перевіряє їх виконання. Окрім цього, користувач переглядає аналітичні дані. Він відкриває графіки, аналізує звіти, оцінює зміни витрат у часі. Дані доступні у різних представленнях. Користувач також імпортує записи з файлів або експортує їх для подальшого використання. Впорядковані дані про варіанти використання представлено у таблиці 1.2

Таблиця 1.2 – Основні варіанти використання застосунку Spendly

ID	**Назва**	**Короткий опис**	**Передумови**	**Основний сценарій**	**Очікуваний результат**
1	**2**	**3**	**4**	**5**	**6**
US-01	Перегляд дашборду	Користувач бачить поточний фінансовий стан: скільки витрачено, зароблено і залишилось за місяць.	Застосунок запущено.	1)Користувач відкриває розділ «Dashboard». 2)Переглядає підсумкові картки за поточний місяць. 3)Аналізує графік «Доходи / Витрати» за 6 місяців і кругову діаграму категорій.	Відображається фінансова зведення місяця з графіками та останніми операціями.
US-02	Додавання транзакції	Користувач фіксує нову операцію: витрату, дохід або переказ між рахунками.	Застосунок запущено.	1)Користувач натискає «+ Транзакція» 2) Вводить необхідні дані. 3)Натискає «Зберегти транзакцію».	Операція збережена; баланс рахунку та підсумки дашборду оновлюються.

Продовження таблиці 1.2

1	2	3	4	5	6
US-03	Редагування транзакції	Користувач виправляє помилку або уточнює дані раніше введеної операції.	У системі є хоча б одна транзакція.	Користувач відкриває розділ «Транзакції». Вибирає запис і натискає іконку редагування. Змінює потрібні поля у формі. Зберігає оновлені дані.	Запис оновлено; агрегати бюджету й аналітики перераховуються.
US-04	Видалення транзакції	Користувач прибирає зайвий або помилковий запис із журналу операцій.	У системі є хоча б одна транзакція.	1)Користувач вибирає транзакцію зі списку. 2)Натискає іконку видалення. 3)Підтверджує дію у діалозі.	Запис вилучено; баланси та звіти перераховуються.
US-05	Фільтрація та пошук транзакцій	Користувач звужує список операцій: шукає конкретний запис або групує за типом, категорією чи датою.	Розділ «Транзакції» відкрито.	1)Користувач вводить текст у поле пошуку або натискає фільтри. 2) Заповнює необхідні поля. 3)Список оновлюється в реальному часі.	На екрані тільки відфільтровані записи; підсумки «Загальні витрати» і «Загальні доходи» перераховуються.

Продовження таблиці 1.2

1	2	3	4	5	6
US-06	Керування категоріями	Користувач додає власні категорії або прибирає непотрібні.	Форма транзакції відкрита або розділ категорій доступний.	1)Користувач відкриває список категорій у формі транзакції. 2)Вибирає існуючу категорію або додає нову. 3)Зберігає зміни.	Список категорій оновлено; нова категорія доступна під час введення наступної транзакції.
US-07	Експорт транзакцій	Користувач вивантажує дані у CSV. Для резервної копії або аналізу в сторонньому інструменті.	У системі є транзакції.	1)Користувач натискає «Експорт CSV» у розділі «Транзакції». 2)Вибирає діапазон або формат. 3)Застосунок генерує та зберігає файл.	Файл із транзакціями збережено на пристрої.
US-08	Налаштування ліміту категорії	Користувач задає планову суму витрат для категорії, щоб контролювати бюджет протягом місяця.	Існують категорії витрат.	1)Користувач переходить у розділ «Бюджет». 2)Додає ліміт або вибирає наявний. 3)Вводить суму ліміту	Ліміт збережено; прогрес-бар категорії відображає поточне використання відносно плану.

Продовження таблиці 1.2

1	2	3	4	5	6
US-09	Перегляд прогресу бюджету	Користувач перевіряє, скільки від плану вже витрачено і які категорії наближаються до межі.	Встановлено хоча б один ліміт.	1)Користувач відкриває розділ «Бюджет». 2)Переглядає картки «Загальний бюджет», «Витрачено», «Залишилось». 3)Аналізує прогрес-бари по кожній категорії.	Наочна картина виконання бюджету; категорії з перевищенням виділені попередженням.
US-10	Перегляд аналітики	Користувач вивчає динаміку своїх фінансів: як змінювались витрати по днях, які категорії з'їдають найбільше і який прогноз до кінця місяця	У системі є транзакції.	1) Користувач переходить у розділ «Аналітика». 2)Вибирає період: місяць / квартал / рік / свій діапазон. 3)Переглядає графік витрат і порівняння з попередніми місяцями. 4)Аналізує розбивку по категоріях і розподіл між рахунками.	Відображаються інтерактивні графіки, прогноз до кінця місяця і топ витратних категорій.

Практика розробки фінансових застосунків показує характерну особливість: основну взаємодію виконує саме користувач. Інші ролі або зовнішні сервіси часто не залучаються, особливо у локальних рішеннях [10]. Діаграма варіантів використання відображає центр цієї взаємодії –Користувача. Від нього йдуть зв'язки до кожного сценарію. Структура виглядає простою. Проста модель легше читається і аналіз також спрощується. Діаграму варіантів використання представлено на рисунку 1.6.

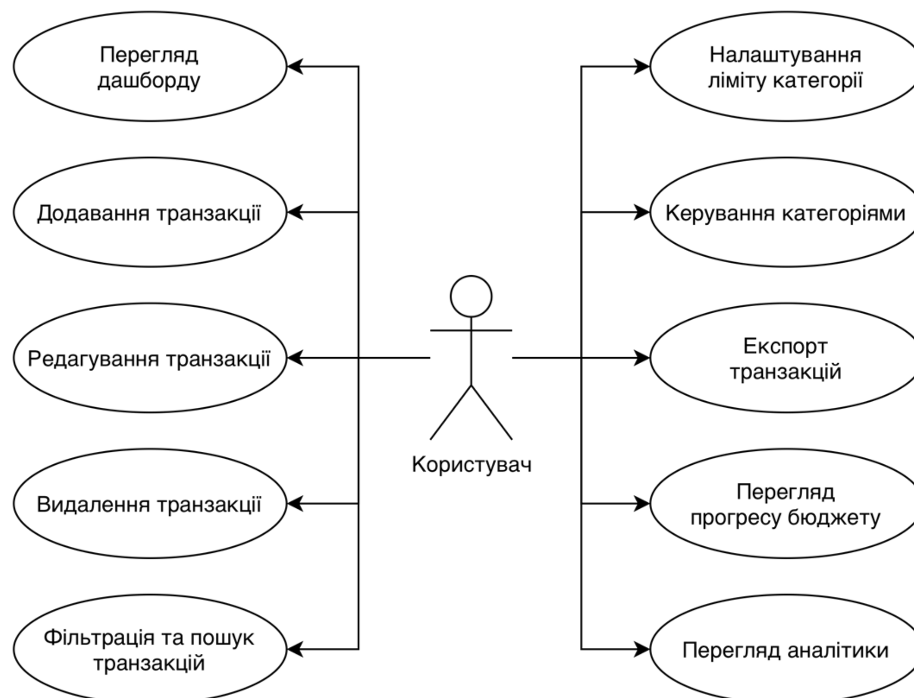

Рисунок 1.6 – Діаграма варіантів використання для актора «Користувач»

1.3 Функціональні та нефункціональні вимоги до застосунку

Функціональні вимоги. У науковій літературі функціональні вимоги розглядаються як опис операцій, які система виконує в різних сценаріях взаємодії з користувачем [11]. До таких операцій належать збереження транзакцій, розрахунок балансу та побудова звітів. Повний і узгоджений перелік функціональних вимог формується на ранніх етапах проєктування. Саме він визначає межі розробки. Перелік функціональних вимог подано у таблиці 1.3.

Таблиця 1.3 – Функціональні вимоги до застосунку Spendly

ID	Назва вимоги	Опис
1	**2**	**3**
FR-01	Додавання транзакцій	Користувач вводить фінансову операцію із зазначенням суми, типу, категорії та дати.
FR-02	Редагування та видалення	Параметри наявного запису змінюються або запис вилучається без перезавантаження екрана.
FR-03	Пошук і фільтрація	Операції відбираються за датою, категорією, типом або рахунком (також одночасно за кількома критеріями).
FR-04	Сортування та пагінація	При великому обсязі записів інтерфейс упорядковує їх за датою чи сумою і показує порціями.
FR-05	Експорт даних	Користувач формує CSV-файл з обраними транзакціями для подальшої обробки або передачі.
FR-06	Керування рахунками	Окремі фінансові рахунки («Готівка», «Картка» тощо) налаштовуються і ведуться незалежно.
FR-07	Керування категоріями	Користувач додає або змінює категорії витрат і доходів та встановлює до них обмеження.
FR-08	Бюджетування	Ліміти витрат фіксуються для визначених періодів; застосунок відображає стан їх виконання.
FR-09	Розрахунок показників	Модуль щоразу перераховує баланс, підсумки доходів і витрат та відсоток виконання бюджету при зміні даних.
FR-10	Візуалізація даних	Динаміка витрат і доходів розкладається по категоріях і часових відрізках на графіках і діаграмах.
FR-11	Інтерфейс користувача	До головних дій кілька кліків максимум; навігація не вимагає попереднього навчання.

Нефункціональні вимоги описують якісні характеристики системи та умови експлуатації. Вони не додають нових функцій, проте визначають обмеження й рівень якості. Йдеться про швидкодію, безпеку, стабільність і зручність використання. Дослідження трактують такі вимоги як властивості системи або обмеження її функціонування [11].

Нижче наведено основні характеристики якості програмного застосунку у табл. 1.4. Вони визначають поведінку системи під навантаженням.

Таблиця 1.4 – Нефункціональні вимоги до застосунку Spendly

ID	**Назва вимоги**	**Опис**
1	**2**	**3**
NFR-01	Продуктивність	При типовому навантаженні запит обробляється за 1-2 секунди. Користувач не чекає [8].
NFR-02	Масштабованість	Коли обсяг даних або кількість облікових записів зростає, переносити базу на потужніший хост можна без зміни бізнес-логіки.
NFR-03	Надійність	Застосунок працює стабільно, фіксує помилки у логах і запобігає втраті даних.
NFR-04	Безпека	Розробник захищає персональні дані через хешування паролів і контроль доступу за ролями [8].
NFR-05	Зручність використання	Інтерфейс допомагає виконувати основні дії швидко й без перевантаження зайвими елементами.
NFR-06	Сумісність	Застосунок функціонує у середовищі Windows відповідно до вимог WPF і задокументованих мінімальних вимог.
NFR-07	Підтримуваність	Архітектура коду залишається зрозумілою та придатною до розширення завдяки принципам SOLID [8].
NFR-08	Логування	Система записує події для аналізу та пошуку помилок, що скорочує час на діагностику збоїв.
NFR-09	Надійність зберігання	База даних зберігає цілісність інформації навіть у разі збоїв завдяки властивостям ACID.
NFR-10	Доступність	Застосунок залишається працездатним протягом більшої частини часу (не менше 99% робочого часу).
NFR-11	Якість коду	Розробник дотримується стандартів і підтримує чисту структуру модулів, що спрощує майбутні зміни [2].

Обидві групи вимог впливають на результат. Призначення системи розкривається через функціонал, а довіра користувача – через те, наскільки

стабільно й безпечно він працює. Нехтування хоча б однією групою веде до погіршення продукту, тому розробник опрацьовує їх паралельно [12].

1.4 Висновок до першого розділу

Особистий фінансовий менеджмент давно потребує спеціалізованих інструментів, що підтверджують дослідження у розділі 1. PFM-застосунки реально змінюють поведінку: користувачі починають краще контролювати витрати й розуміти власні фінанси [13].

Розглядаючи додаток Spendly, проглядається типова архітектура: підтримка додавання й редагування транзакцій, категоризації, бюджетування та звітності, зі збереженням простоти інтерфейсу при цьому. Модель акторів спирається на єдиного Користувача. Жодних зовнішніх ролей, сторонніх сервісів чи інтеграцій. Авторизація й управління обліковими записами не потрібні при такому підході. Застосунок реалізує концепцію local-first: фінансові дані живуть на пристрої користувача і не потрапляють у хмару.

Дослідження приватності бюджетних додатків зафіксувало, що близько 60% популярних сервісів передають дані третім сторонам [14]. Локальне зберігання усуває цей ризик повністю. Архітектурні наслідки прості, але критичні. Офлайн-режим і локальна база даних знімають необхідність адміністрування серверів, оскільки немає паролів, немає автентифікації, нижчий бар'єр входу [14].

Натомість розробник отримує інше завдання: надійно захистити файл із даними на диску. Платформа .NET надає для цього перевірені механізми (зокрема, AES через бібліотеку System.Security.Cryptography). Microsoft рекомендує симетричні алгоритми для шифрування великих обсягів даних у потоках [15]. Спираючись на ці рекомендації, Spendly має шифрувати файл бази даних або задіювати криптографічні сервіси операційної системи. Кожен сценарій роботи обертається навколо однієї особи, тому інтерфейс мусить відповідати цій логіці. Під час введення нової операції користувач бачить чотири

поля: суму, категорію, рахунок і дату. Дослідники підтверджують: бюджетні застосунки отримують позитивну оцінку саме тоді, коли вони прозорі й інтуїтивні [13]. Spendly показує підсумки за категоріями, прогрес бюджету й історію операцій, що й формує аналітичну цінність продукту.

Подальший розвиток застосунку вимагає кількох конкретних кроків. По-перше, надійний механізм резервного копіювання. Оскільки програма локальна, втрата пристрою означає втрату всіх даних. По-друге, підтримка форматів CSV і JSON для експорту та імпорту зі сторонніх банківських систем.

Корисним доповненням стали б швидкі шаблони для регулярних витрат або OCR-сканування квитанцій. Дизайн із єдиним користувачем спрощує реалізацію, але звужує можливості. Без хмарної синхронізації мобільний доступ і спільний бюджет із близькими недоступні. Продуктивність обмежена ресурсами одного пристрою. Для особистих потреб це прийнятно, для сімейного обліку вже може стати проблемою.

Підсумовуючи: Spendly відповідає базовим потребам користувача і виграє у приватності та простоті [14]. Майбутній розвиток проекту має бути спрямований на розширення підтримки форматів, резервне копіювання та підвищення безпеки зі стандартними механізмами .NET. Збереження принципів локального зберігання і приватності поряд із зручним і надійним інтерфейсом стане орієнтиром для наступних ітерацій

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ ДЛЯ ОБЛІКУ ТА ПЛАНУВАННЯ ОСОБИСТИХ ФІНАНСОВИХ ВИТРАТ SPENDLY

Архітектурне проєктування є визначальним етапом розробки програмного забезпечення, оскільки прийняті на цьому етапі рішення безпосередньо впливають на якість, супроводжуваність і розширюваність системи.

У другому розділі розглядаються архітектура застосунку Spendly, поведінкові UML-діаграми та структура бази даних, що у сукупності формують повну проєктну модель системи.

2.1 Архітектура системи та діаграми класів

Архітектура й модель даних є відправною точкою будь-якого серйозного застосунку. Від того, як розмежовані компоненти, залежить якість тестування і здатність системи розвиватись без накопичення технічного боргу. Дослідники підтверджують це: архітектура програмного забезпечення відіграє ключову роль на всіх етапах – від проєктування до підтримки [16]. На рисунку 2.1 продемонстровано архітектуру проєкту Spendly.

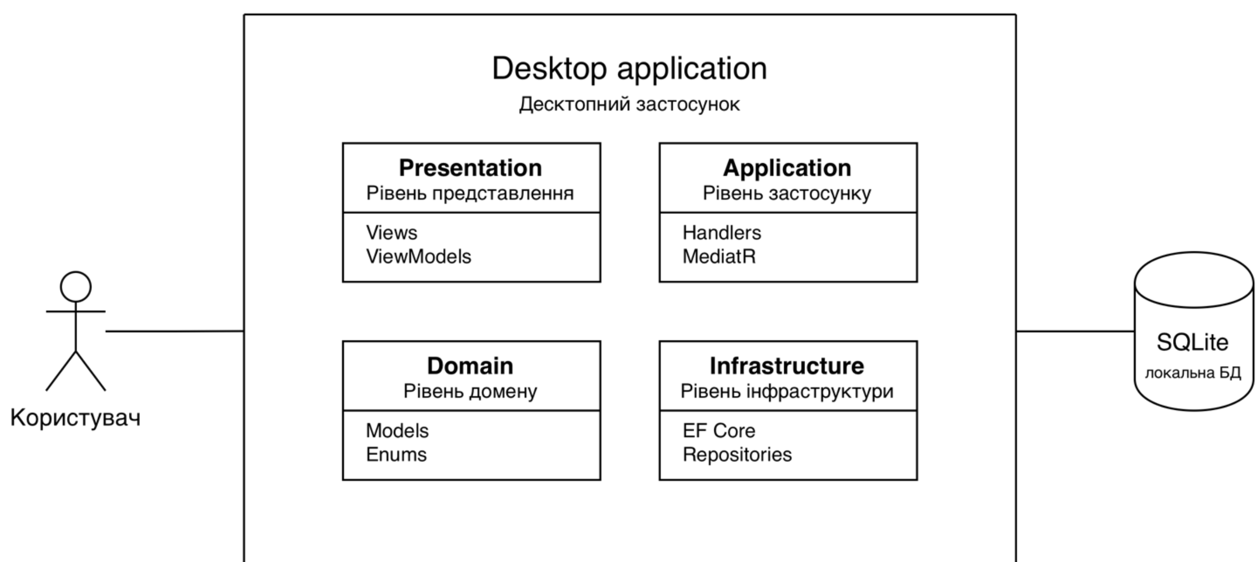

Рисунок 2.1 – Архітектура проєкту Spendly

Графічний інтерфейс побудований з використанням патерну MVVM (Model-View-ViewModel), де логіка представлення структурно відділена від UI, а не умовно. MVVM добре вивчений у літературі саме завдяки тестованості й незалежності інтерфейсу від бізнес-логіки [17]. ViewModel у WPF-додатках працює посередником між View і Model. Жоден із шарів не знає деталей реалізації іншого. Поєднання MVVM і шарової архітектури надає системі гнучкості й спрощує розширення у майбутньому. 2.1 Архітектура системи та діаграми класів Чотири шари утворюють основу Spendly: інтерфейс користувача, доменна модель, доступ до даних і прикладна логіка. Кожен шар відокремлений від решти. UI реалізований на WPF із патерном MVVM: окремі ViewModels і Views взаємодіють через двосторонній байндинг.

Шар прикладної логіки (Spendly.Application) організований за принципом CQRS з бібліотекою MediatR. Команди й запити проходять через окремі хендлери, що полегшує додавання нових операцій з транзакціями чи аналітикою. Для роботи з базою даних використано Entity Framework Core із класами-сутностями, контекстом SpendlyDbContext і патерном Repository разом з Unit of Work.

Згідно з підходом Domain-Driven Design, патерн Repository ізолює доменну модель від деталей збереження даних [18], тому доменна логіка не залежить від специфіки СУБД. Між сутностями (Account, Category, Transaction) та інфраструктурним шаром сформовані чіткі зв'язки, що полегшують підтримку й тестування коду.

Нижче на рис. 2.2-2.8 продемонстровано основні класи та взаємозв'язки підсистеми отримання та відображення транзакцій системи Spendly.

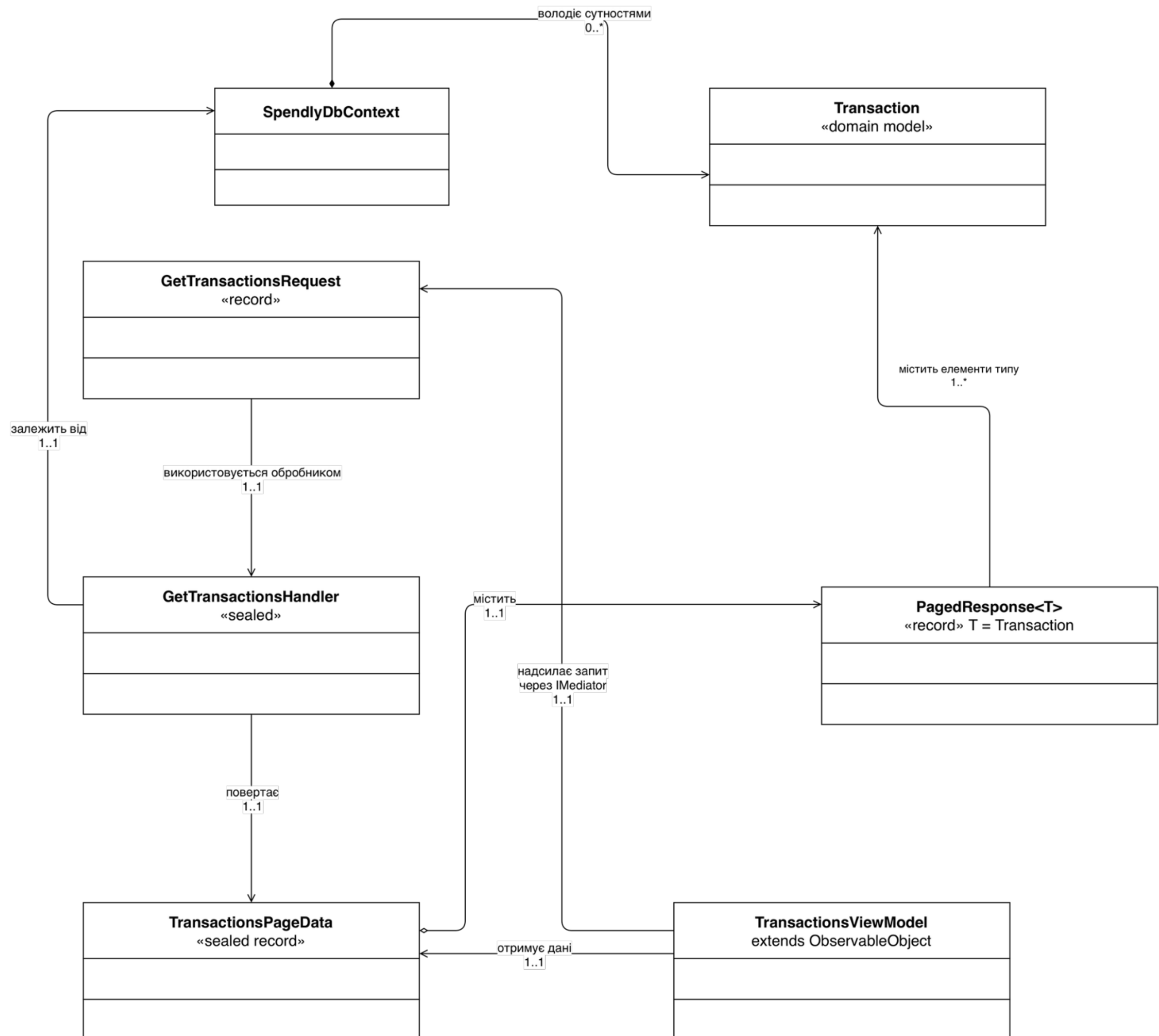

Рисунок 2.2 – Діаграма ієрархії класів UML підсистеми транзакцій системи обліку та планування особистих фінансів Spendly

GetTransactionsRequest (див. рис. 2.3) є чистим DTO-запитом, що реалізує інтерфейс IRequest і передає параметри фільтрації до шару застосунку. Клас містить необов'язкові поля Type (перелічення TransactionType: Expense або Income), Month (DateOnly для фільтрації за місяцем), CategoryId та AccountId (ідентифікатори для звуження вибірки) і рядок SearchTerm для повнотекстового пошуку. Поля пагінації Page (за замовчуванням 1) і PageSize (за замовчуванням 10) визначають розмір сторінки результату. Клас не містить методів – його єдина відповідальність полягає у транспортуванні критеріїв запиту.

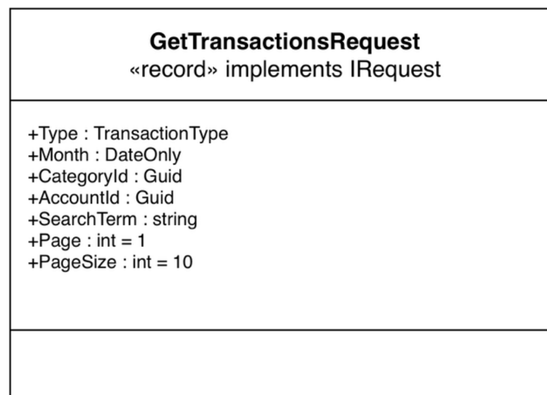

Рисунок 2.3 – Детальний опис класу GetTransactionsRequest

GetTransactionsHandler (див. рис. 2.4) є запечатаним класом (sealed), що реалізує інтерфейс IRequestHandler і містить приватне поле _dbContext типу SpendlyDbContext. Метод Handle приймає запит і токен скасування та повертає Task<TransactionsPageData>. Усередині методу послідовно застосовуються фільтри до IQueryable: за Type, за діапазоном дат місяця (UTC), за CategoryId та AccountId, а також за допомогою EF.Functions.Like по полю SearchTerm. Після фільтрації виконуються агрегатні запити SumAsync для обрахунку загальних витрат і доходів, а потім – вибірка з пагінацією через Skip та Take.

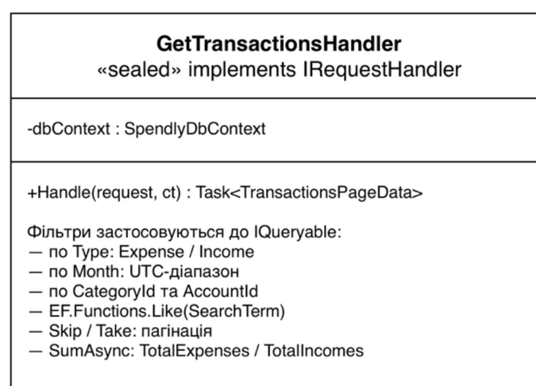

Рисунок 2.4 – Детальний опис класу GetTransactionsHandler

Transaction (див. рис. 2.5) є доменною моделлю і центральною сутністю підсистеми. Клас містить унікальний ідентифікатор Id типу Guid, поле AccountId

(прив'язка до рахунку), необов'язковий CategoryId, десяткове Amount (сума операції), DateUtc (мітка часу у форматі UTC), перелічення Type (Expense або Income) і рядок Comment для приміток. Навігаційні властивості Account та Category пов'язують транзакцію з відповідними об'єктами. Саме Transaction є основою для побудови аналітики, фільтрації та формування історії операцій.

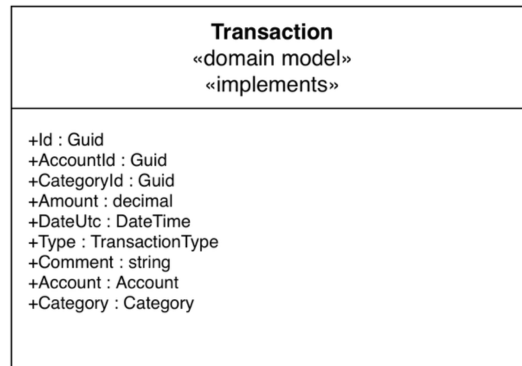

Рисунок 2.5 – Детальний опис класу Transaction

PagedResponse<T> (див. рис. 2.6) є узагальненим record-класом, параметризованим типом Transaction. Містить колекцію Items (IReadOnlyList<T>), загальну кількість записів TotalCount, поточну сторінку Page і розмір сторінки PageSize. Обчислювані властивості TotalPages, HasPreviousPage та HasNextPage надають ViewModel готову інформацію про стан пагінації без додаткових обчислень на стороні UI.

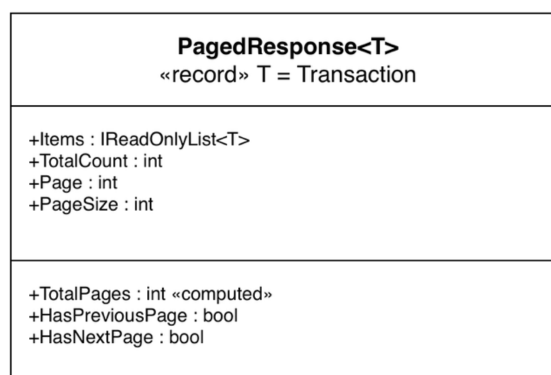

Рисунок 2.6 – Детальний опис класу PagedResponse<T>

TransactionsPageData (див. рис. 2.7) є запечатаним record-класом – результатом виконання Handle у GetTransactionsHandler. Містить поле Page типу PagedResponse<Transaction>, а також агрегати TotalExpenses і TotalIncomes типу decimal. Клас не містить методів і є чистим DTO, що передає дані від шару застосунку до ViewModel.

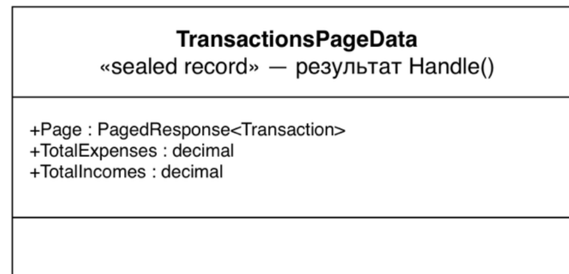

Рисунок 2.7 – Детальний опис класу TransactionsPageData

TransactionsViewModel (див. рис. 2.8) розширює ObservableObject з бібліотеки CommunityToolkit.Mvvm і є центральним класом представлення для екрану транзакцій. Взаємодія з шаром застосунку відбувається через приватне поле _mediator типу IMediator, а доступ до довідникових даних – через _categoryQueries та _accountQueries. ViewModel зберігає стан фільтрації (SelectedTypeFilter, SelectedMonth, SelectedCategoryId, SelectedAccountId, SearchText), стан пагінації (CurrentPage, TotalPages) і агрегати (TotalExpenses, TotalIncome). Метод InitializeAsync ініціалізує довідники та завантажує першу сторінку. Приватний метод LoadTransactions формує GetTransactionsRequest з поточних фільтрів і надсилає його через Mediator. DebounceSearchAsync забезпечує затримку 300 мс перед пошуком, щоб уникнути зайвих запитів під час введення. Методи GoToNextPage, GoToPreviousPage та ExportCsv позначені атрибутом RelayCommand і доступні для прив'язки у XAML-представленні.

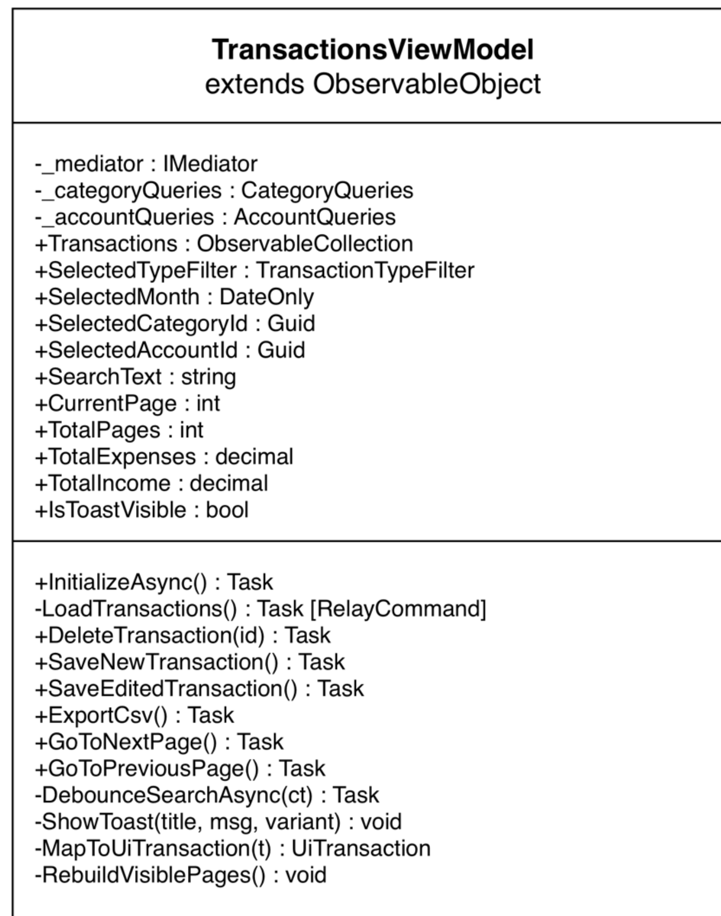

Рисунок 2.8 – Детальний опис класу TransactionsViewModel

Отже, підсистема транзакцій у Spendly побудована з чітким розподілом відповідальностей між запитом, обробником, доменною моделлю, DTO-результатом і ViewModel.

2.2 Моделювання поведінки системи: діаграми послідовності, діяльності та станів

Статичні структурні схеми не розкривають, як система поводиться під час роботи. UML (Unified Modeling Language) закриває потребу в цьому через поведінкові нотації, зокрема діаграми послідовності, діяльності та станів. Кожна з них фіксує окремий зріз динаміки: хронологію повідомлень між об'єктами, алгоритмічний потік управління та зміни станів компонента впродовж його життєвого циклу [19]. Через непрямість взаємодій у MVVM-системах поведінку

важко простежити лише за кодом, виникає потреба у окремій моделі. Ларман встановив: якщо поведінкові діаграми будують на етапі проєктування, а не постфактум, кількість архітектурних помилок скорочується до 40% [20].

У Spendly логіку обробки фінансових операцій розподілено між шарами Domain, Application та Infrastructure. Побудовані діаграми дають змогу простежити повний шлях кожного запиту.

Учасників взаємодії на діаграмах послідовності позначають вертикальні лінії, повідомлення між ними відображені горизонтальними стрілками. Пунктирні стрілки вказують на зворотне повернення управління. Хронологія читається зверху вниз.

На рисунку 2.7 показано послідовність додавання транзакції. Користувач заповнює форму в AddTransactionView і натискає Save. Після цього подія надходить до AddTransactionViewModel. Метод SaveCommand.Execute() ініціює асинхронний виклик await _mediator.Send(request, ct) із об'єктом CreateTransactionRequest. Завдяки патерну Mediator ViewModel не залежить від конкретного обробника – шари Presentation та Application повністю розв'язані. CreateTransactionHandler отримує запит, перетворює модель у доменну сутність через Riok.Mapperly і зберігає результат викликом SpendlyDbContext.SaveChangesAsync(). Після успішного запису ViewModel показує toast-повідомлення та закриває форму.

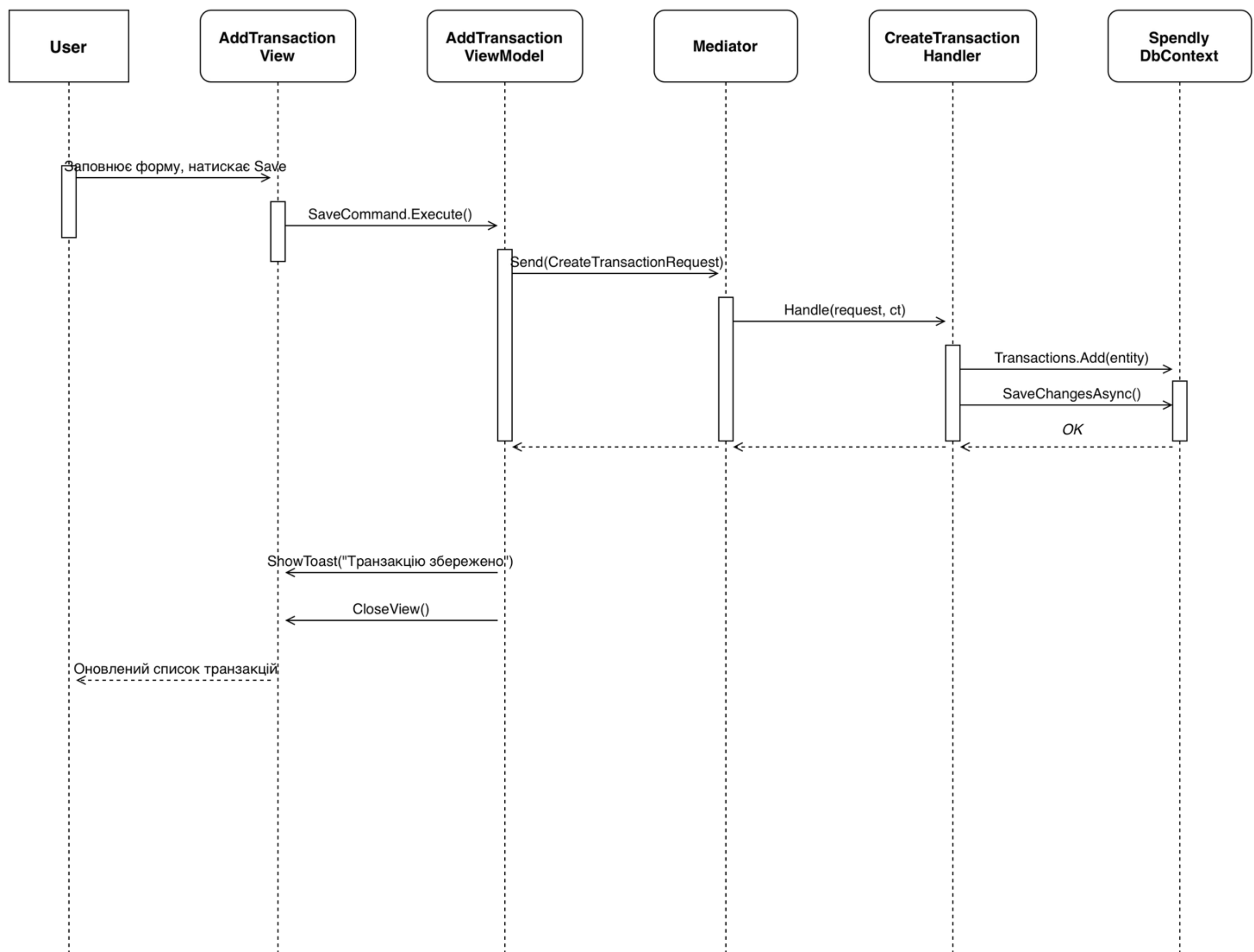

Рисунок 2.7 – Діаграма послідовності операції додавання транзакції

Від звичайної блок-схеми діаграма діяльності відрізняється підтримкою паралельного виконання та стандартизованою семантикою UML. Ромби позначають вузли прийняття рішень (decision nodes), кола – початковий і кінцевий стани. Потік може розгалужуватись і зливатись.

Рисунок 2.8 ілюструє алгоритм обробника ExportCsvHandler. Обробник отримує ExportCsvRequest із необов'язковими фільтрами: тип транзакції (Type), місяць (Date), категорія та рахунок. Для кожного ненульового фільтра до базового IQueryable додається умова WHERE (підхід зберігає єдиний запит до бази замість кількох послідовних). Після побудови запиту застосунок перевіряє наявність директорії і за потреби створює її. Відфільтровані транзакції передаються у CsvWriter.WriteRecordsAsync() з бібліотеки CsvHelper, після чого обробник повертає повний шлях до збереженого файлу. Поетапне застосування

фільтрів через ланцюжок IQueryable мінімізує обсяг даних, що завантажуються з бази, і підвищує продуктивність при великих наборах транзакцій.

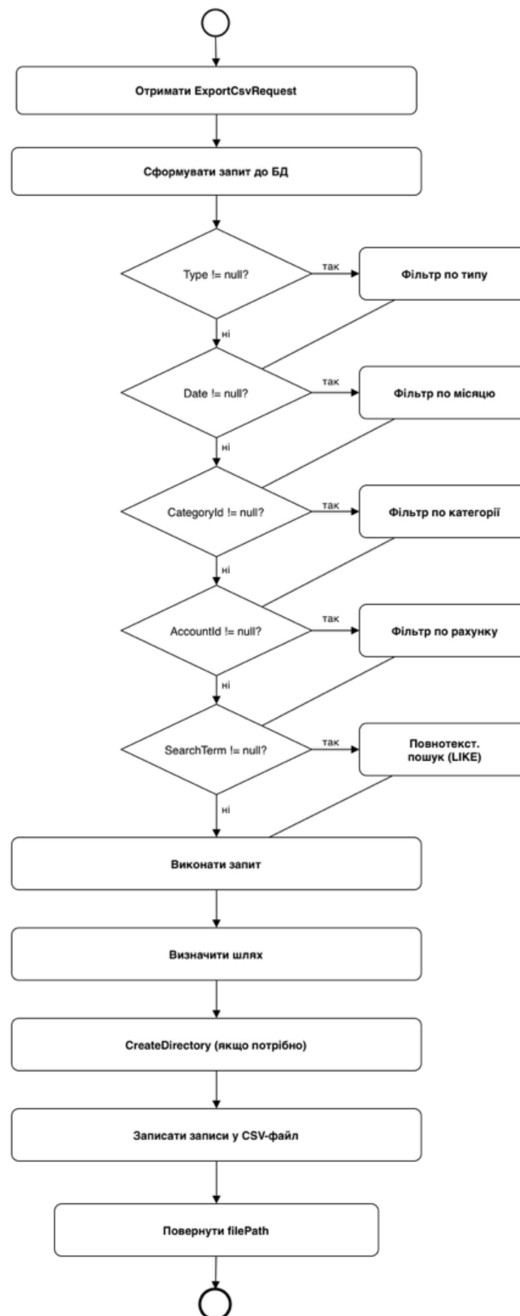

Рисунок 2.8 – Діаграма діяльності для алгоритму обробника ExportCsvHandler

Діаграма станів моделює можливі стани об'єкта та переходи між ними під дією подій або умов. У MVVM-архітектурі стан інтерфейсу у будь-який момент має бути детермінованим, а переходи – передбачуваними. Недетермінований UI є джерелом регресій.

Запуск застосунку ініціює завантаження: ViewModel надсилає запит до GetTransactionsHandler і переходить у стан Завантаження (IsLoading = true). Отримавши дані, компонент переходить у стан Відображення, коли користувач бачить список транзакцій і може з ним працювати. Зміна фільтра (місяць, тип, категорія, рахунок) є самопереходом у тому самому стані з перезапуском запиту; повного перезавантаження ViewModel при цьому не відбувається. Зі стану Відображення відкриваються три переходи: до Форми додавання (AddCommand), Форми редагування (EditCommand) або Підтвердження видалення (DeleteCommand). Підтвердження дії у будь-якому з них повертає систему до стану Завантаження з оновленням даних. Скасування повертає назад до Відображення без змін. Пунктирні стрілки на діаграмі позначають умовні переходи: лише за умови успішного збереження. На рисунку 2.9 зображено діаграму станів для TransactionsViewModel.

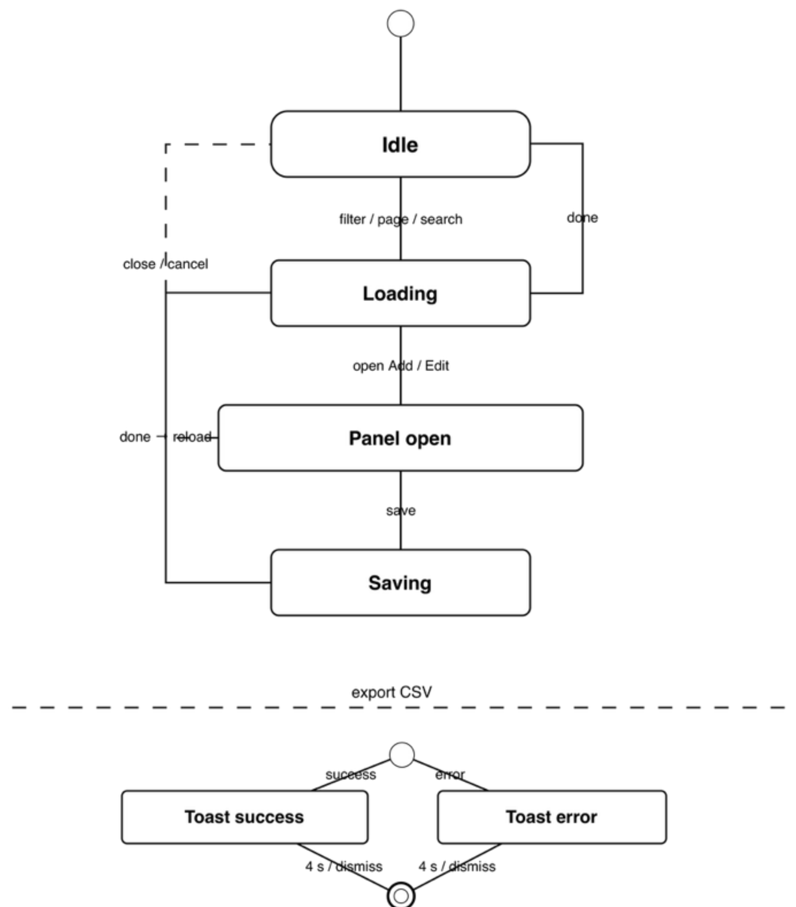

Рисунок 2.9 – Діаграма станів для TransactionsViewModel

Три побудовані діаграми охоплюють різні виміри динаміки проєкту Spendly. Діаграма послідовності підтверджує коректне застосування патерну Mediator. Алгоритм фільтрації й збереження CSV-файлу формалізує діаграма діяльності, що важливо для верифікації бізнес-логіки. TransactionsViewModel у будь-який момент перебуває в одному з чітко визначених станів. Детерміновану поведінку інтерфейсу підтверджує діаграма станів.

Сукупне використання трьох типів відповідає специфікації OMG UML 2.5, яка наголошує на потребі поєднувати взаємодіяльнісне, процедурне та стано-машинне моделювання для повноти документації складних систем [21].

2.3 Проєктування бази даних

Структура сховища визначає продуктивність запитів, цілісність даних і здатність системи розвиватися. Причому ці три аспекти взаємопов'язані й не можна оптимізувати один, ігноруючи інші. Реляційна модель тримає домінуючу позицію у фінансових застосунках саме через строгі механізми цілісності та зрілу екосистему інструментів. Нормалізація схеми і стратегія індексування впливають на продуктивність навіть при малих обсягах даних. Це явище підтверджується у дослідженнях із проєктування баз даних [22].

Для Spendly обрано SQLite з цілком прагматичної причини: однокористувацький desktop-інструмент не потребує паралельних підключень і окремого сервера. Файл бази зберігається у директорії користувача без жодної конфігурації. Схему при цьому спроєктовано провайдеронезалежно. При переході на PostgreSQL структура таблиць не змінюється, лише рядок підключення [23].

Усього чотири таблиці: accounts, categories, category_limits і transactions. Схема нормалізована до 3НФ. Транзитивні залежності відсутні, кожна таблиця моделює рівно один об'єкт предметної області.

Сутність accounts (рахунки користувача). Рахунок фіксує джерело або отримувача коштів. Первинний ключ id має тип UUID – він гарантує унікальність

незалежно від СУБД і не видає кількість записів у таблиці, на відміну від автоінкрементного числа.

Поле type приймає одне з трьох рядкових значень: Cash, Card, Savings. Баланс зберігається у balance з фіксованою десятковою точністю.

Таблиця 2.1 – Сутність accounts

Стовпець	**Тип**	**Обмеження**	**Опис**
1	**2**	**3**	**4**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор рахунку
name	TEXT	NOT NULL	Назва рахунку
balance	DECIMAL	NOT NULL	Поточний баланс
type	TEXT(32)	NOT NULL	Тип рахунку

Сутність categories (категорії операцій). Категорії дають змогу користувачу класифікувати транзакції за напрямками. З першого запуску застосунок наповнює таблицю 17 записами: Groceries, Sports, Entertainment, Cafes, Health, Transport, Rent, Subscriptions, Education, Clothing, Gifts, Travel, Insurance, Taxes, Investments, Salary, Other.

Надалі користувач самостійно розширює перелік. У полі name побудовано унікальний індекс, тому дублікати блокуються на рівні бази, незалежно від того, чи перевіряє їх логіка застосунку.

Таблиця 2.2 – Сутність categories

Стовпець	**Тип**	**Обмеження**	**Опис**
1	**2**	**3**	**4**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор
name	TEXT(64)	NOT NULL, UNIQUE	Назва категорії

Сутність category_limits (бюджетні ліміти). Ліміт прив'язується до категорії у відношенні «один-до-одного». Ліміти винесено в окрему таблицю свідомо: categories зберігає стабільну класифікацію, а category_limits записує змінні користувацькі налаштування, які можуть з'являтись і зникати незалежно.

Об'єднувати їх в одній таблиці означало б порушити принцип єдиної відповідальності на рівні схеми.

Таблиця 2.3 – Сутність category_limits

Стовпець	**Тип**	**Обмеження**	**Опис**
1	**2**	**3**	**4**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор ліміту
category_id	UUID	FK – categories.id, UNIQUE	Посилання на категорію
amount	DECIMAL	NOT NULL	Сума місячного ліміту

Сутність transactions (фінансові транзакції). Транзакції є найбільшою таблицею і головним об'єктом аналітичних запитів. Кожен запис прив'язаний до рахунку обов'язково, в той час як категорія не прив’язана.

Поле type зберігається цілим числом: 0 – дохід, 1 – витрата. Числовий тип фільтрується швидше за рядковий при зростанні кількості рядків. Дата записується у UTC (без прив'язки до локального часового поясу клієнта). Точність amount задана явно: 18 значущих цифр, два знаки після коми. Для побутових фінансів це є достатньо великим запасом.

По account_id і category_id побудовано окремі некластерні індекси. Вони безпосередньо прискорюють вибірку транзакцій за рахунком і щомісячну агрегацію витрат за категорією, що покриває два найчастіші сценарії у роботі аналітичних обробників.

Таблиця 2.4 – Сутність transactions

Стовпець	**Тип**	**Обмеження**	**Опис**
1	**2**	**3**	**4**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор транзакції
account_id	UUID	FK – accounts.id, NOT NULL	Рахунок-джерело/отримувач
category_id	UUID	FK – categories.id, NULL	Категорія (опційно)
amount	DECIMAL(18,2)	NOT NULL	Сума транзакції
date_utc	DATETIME	NOT NULL	Дата і час у форматі UTC
type	INTEGER	NOT NULL	0 – дохід, 1 – витрата
comment	TEXT(512)	NOT NULL	Коментар до транзакції

Нормалізація схеми бази даних. Надлишковість у реляційній схемі породжує аномалії оновлення, що означає ситуації, коли одна логічна зміна вимагає правок у кількох місцях одночасно. Нормалізація усуває цю проблему через послідовне приведення схеми до більш суворих форм, кожна з яких будується на попередній [22].

Перша нормальна форма (1НФ): атрибути атомарні, рядки унікальні. У Spendly жоден стовпець не зберігає списків чи складених значень: comment у transactions містить один рядок, type у accounts також містить одне значення переліку. UUID-ключі гарантують унікальність рядків. Вимога виконана.

Друга нормальна форма (2НФ) забороняє часткові залежності. Неключовий атрибут не може залежати лише від частини складеного ключа. У Spendly всі первинні ключі прості: один стовпець id. Часткова залежність за таких умов апріорі неможлива. Атрибути amount, date_utc, type і comment у transactions залежать виключно від ідентифікатора запису. Вимога виконана.

Третя нормальна форма (3НФ) виключає транзитивні залежності між неключовими атрибутами. Розглянемо таблицю transactions: вона містить account_id і category_id як зовнішні ключі, проте жодного атрибута, похідного від них. Назва рахунку зберігається в accounts, назва категорії у categories. Кожна таблиця описує рівно свою сутність і нічого зайвого. Вимога виконана.

Практичним наслідком стає те, що перейменування рахунку вноситься одним записом у accounts.name і миттєво стає актуальним для всіх транзакцій через зовнішній ключ. Денормалізована схема вимагала б оновлення кожного рядка transactions окремо, і будь-який збій залишив би дані в неузгодженому стані.

Чотири таблиці покривають предметну область без надлишкових структур. UUID як тип ключа, диференційовані стратегії каскадного видалення і точкове індексування на стовпцях фільтрації. Кожне рішення продиктоване конкретною вимогою. Нормалізація до 3НФ усуває аномалії оновлення і лишає схему відкритою до розширення без перебудови наявних таблиць [22].

На рисунку 2.10 наведено ER-діаграму сутностей.

Accounts – transactions (1:N). Рахунок накопичує довільну кількість транзакцій. Зовнішній ключ account_id є обов'язковим, оскільки транзакція без рахунку втрачає фінансовий зміст, тому видалення рахунку каскадно видаляє всі його записи. categories – transactions (1:0..N). Зв'язок необов'язковий. Транзакція без категорії є повноцінним записом. При видаленні категорії поле category_id обнуляється, самі транзакції не зачіпаються. categories – category_limits (1:0..1). Категорія може мати один ліміт або жодного. Видалення категорії автоматично прибирає ліміт через каскад.

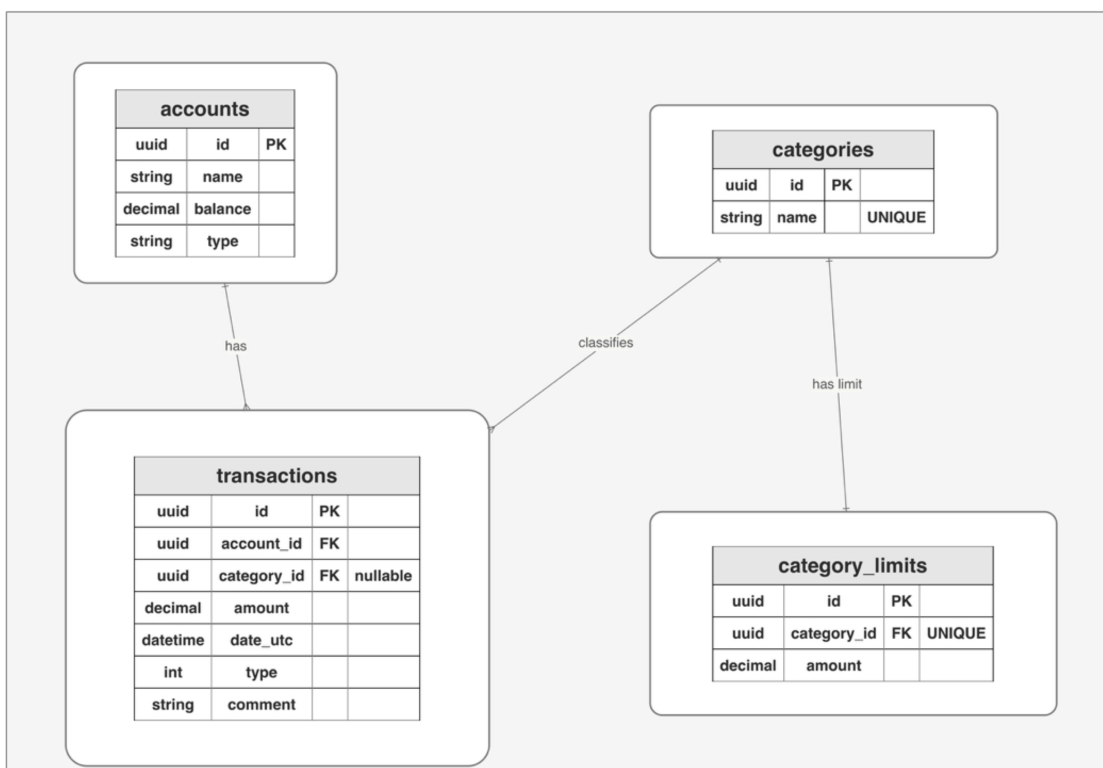

Рисунок 2.10 – ER-діаграма сутностей системи

2.4 Висновок до другого розділу

Розділ 2 присвячено проєктуванню застосунку Spendly, починаючи вибором архітектури і закінчуючи побудовою фізичної структури бази даних. Прийняті рішення формують узгоджену систему, у якій кожен компонент має чітку роль і підтримує стабільну роботу фінансового застосунку.

У підпункті 2.1 розглянуто архітектурну основу Spendly. Застосунок побудовано з поділом на рівні інтерфейсу користувача, прикладної логіки, доменної моделі та доступу до даних. Такий підхід зменшує зв'язність між модулями й спрощує подальший супровід. Графічний інтерфейс реалізовано на WPF із використанням MVVM: View відповідає за відображення стану, а ViewModel керує логікою взаємодії. MediatR координує обмін між шарами, а обробники команд і запитів містять основну бізнес-логіку. Entity Framework Core у поєднанні з Repository та Unit of Work відокремлює доменну модель від механізмів збереження даних. Основу моделі формують класи Account, Category, Transaction і CategoryLimit, які описують рахунки, категорії, фінансові операції та бюджетні обмеження.

У підпункті 2.2 подано поведінкову модель системи. Діаграми послідовності, діяльності та станів показують, як Spendly реагує на дії користувача, обробляє запити та змінює стан ViewModel. Зокрема, модель створення транзакції демонструє шлях від введення даних до збереження запису в базі, а діаграма діяльності пояснює логіку виконання операцій, включно з експортом у CSV. Такі моделі допомагають перевірити сценарії роботи ще до повної реалізації продукту.

У підпункті 2.3 описано структуру бази даних. Реляційна схема містить таблиці accounts, categories, transactions і category_limits, які відтворюють доменну модель та пов'язані зовнішніми ключами. Нормалізація до третьої нормальної форми зменшує надлишковість даних, а індекси для полів фільтрації пришвидшують виконання запитів. SQLite добре підходить для локального фінансового застосунку, оскільки не потребує мережевого підключення й зберігає простоту розгортання. Отже, проєктні рішення розділу 2 формують передбачувану й керовану систему. Архітектура задає взаємодію компонентів, база даних підтримує цілісність інформації, а поведінкові моделі пояснюють логіку роботи Spendly у ключових сценаріях.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ SPENDLY

Реалізація програмного забезпечення є етапом, на якому архітектурні та проєктні рішення отримують конкретне технічне втілення, а їх коректність підтверджується через систематичне тестування.

У третьому розділі описується програмна реалізація компонентів застосунку Spendly, структура та функціональність інтерфейсу користувача, а також стратегія і результати автоматизованого тестування розробленої системи.

3.1 Реалізація шарів даних та бізнес-логіки

Архітектура застосунку Spendly спирається на принципи Clean Architecture. Система має кілька незалежних шарів, між якими залежності спрямовані від зовнішніх компонентів до внутрішніх [23]. Кожен шар має окрему зону відповідальності. Завдяки поділу коду компоненти легше тестувати, замінювати та підтримувати без порушення роботи всієї системи.

Додаток Spendly містить дев'ять проєктів, які формують чотири логічні шари. Найглибший шар Domain зберігає доменні моделі Transaction, Account, Category, а також перелічувальні типи даних TransactionType і AccountType.

Domain не має залежностей від інших проєктів. Application зосереджує бізнес-логіку у вигляді обробників команд і запитів. Для роботи йому потрібні лише Domain та інтерфейси, оголошені для інфраструктурного рівня. Infrastructure/Data працює з даними через Entity Framework Core і містить конкретні класи репозиторіїв. Presentation представлений WPF-застосунком. UI звертається до внутрішніх шарів, однак внутрішні шари не містять знань про інтерфейс користувача.

Принцип інверсії залежностей у Spendly проявляється через розділення контрактів і реалізацій. Application оголошує інтерфейси ITransactionsRepository, IAccountsRepository, IUnitOfWork, а Infrastructure передає конкретні реалізації через механізм ін'єкції залежностей [24].

Усі залежності проекту можна побачити на діаграмі залежностей між проектами на рисунку 3.1.

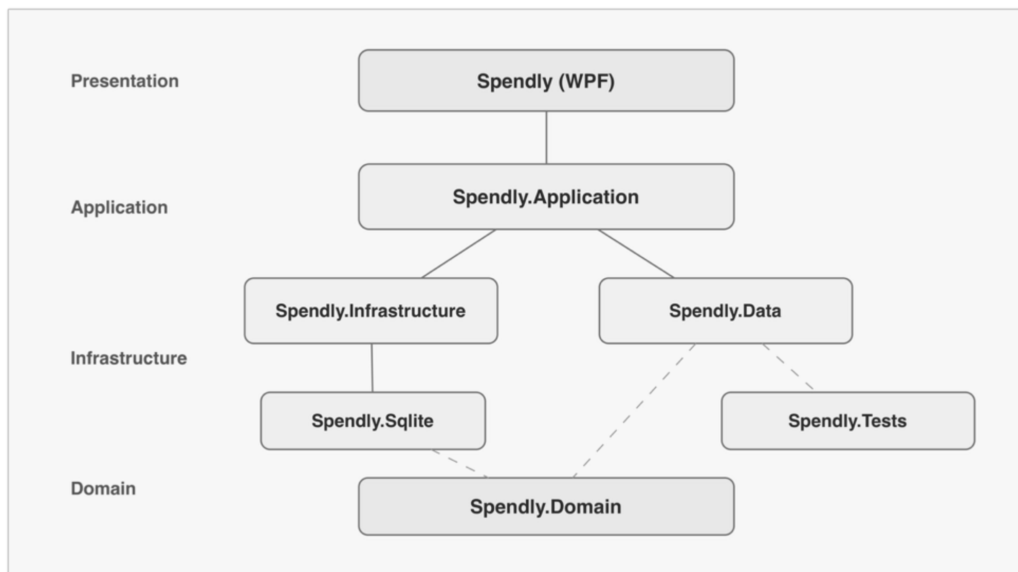

Рисунок 3.1 – Діаграма залежностей між проектами рішення Spendly

Доменний шар містить три ключові моделі: Transaction, Account і Category. Модель Transaction є центральною сутністю системи. Вона пов'язана з рахунком (Account) через зовнішній ключ AccountId та з категорією (Category) через nullable CategoryId, що дозволяє транзакціям існувати без категорії, що демонструє лістинг 3.1:

Лістинг 3.1 – Програмний код

```
public class Transaction
{
    public Guid Id { get; set; }
    public Guid AccountId { get; set; }
    public Guid? CategoryId { get; set; }
    public decimal Amount { get; set; }
    public DateTime DateUtc { get; set; }
    public TransactionType Type { get; set; }
    public string Comment { get; set; } = "";
    public Account Account { get; init; } = null!;
    public Category? Category { get; init; }
}
```

Зберігання дати у форматі UTC (DateUtc) є свідомим рішенням, яке унеможливлює помилки, пов'язані з часовими поясами при фільтрації та агрегації даних. Перелік TransactionType містить два значення Expense та Income, що є достатнім для реалізації повного обліку особистих фінансів.

Додатково шар Infrastructure містить сутність CategoryLimitEntity з унікальним індексом на полі CategoryId, що на рівні бази даних гарантує: одна категорія може мати щонайбільше один бюджетний ліміт. Зв'язок між Category та CategoryLimit реалізований як «один до нуля-або-одного» через конфігурацію EF Core.

Доступ до даних реалізований через Entity Framework Core 9 із провайдером SQLite. База даних зберігається локально у директорії %AppData%\Spendly\spendly.db, що забезпечує роботу застосунку без потреби в мережевому підключенні. Центральним елементом є клас SpendlyDbContext, наведений у лістингу 3.2.

Лістинг 3.2 – Програмний код

```
public class SpendlyDbContext(
    DbContextOptions<SpendlyDbContext> options,
    IProviderModelBuilder? providerModelBuilder)
    : DbContext(options)
{
    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.ApplyConfigurationsFromAssembly(
            typeof(SpendlyDbContext).Assembly);
        providerModelBuilder?.Configure(modelBuilder);
    }
}
```

Конфігурація EF-сутностей виконана через окремі класи, що реалізують IEntityTypeConfiguration<T>. Зокрема, TransactionEntityConfig налаштовує точність decimal-поля (HasPrecision(18, 2)), максимальну довжину коментаря (512 символів) та каскадну поведінку при видаленні. DeleteBehavior.SetNull для

зв'язку з категорією (щоб видалення категорії не знищувало транзакції) та DeleteBehavior.Cascade для зв'язку з рахунком.

Репозиторії (AccountsRepository, TransactionsRepository) реалізують інтерфейси з Spendly.Infrastructure.Interfaces і виконують лише операції додавання, оновлення та видалення сутностей без збереження змін, оскільки цю відповідальність несе клас UnitOfWork (див. лістинг 3.3).

Лістинг 3.3 – Програмний код

```
public class UnitOfWork(SpendlyDbContext dbContext) :
IUnitOfWork
{
    public async Task CompleteAsync(CancellationToken ct)
        => await dbContext.SaveChangesAsync(ct);
}
```

Патерн Unit of Work забезпечує атомарність операцій: кілька змін у різних репозиторіях фіксуються єдиним викликом SaveChangesAsync, що відповідає транзакційній природі фінансових операцій [25]. Схему взаємодії репозиторіїв продемонстровано на рисунку 3.2.

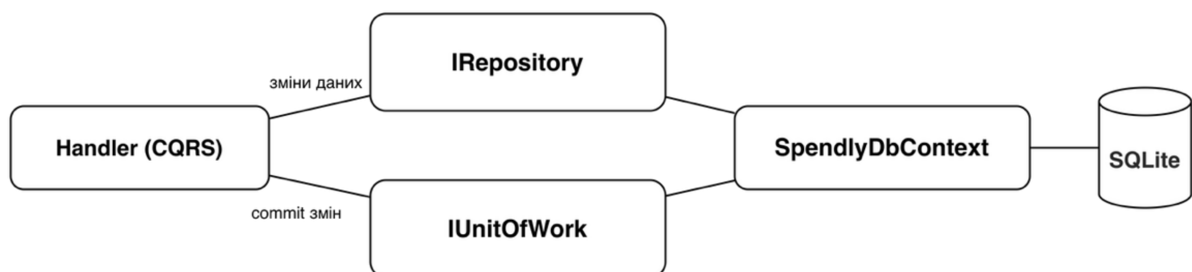

Рисунок 3.2 – Схема взаємодії репозиторіїв, Unit of Work та DbContext

У застосунку Spendly бізнес-логіка побудована за патерном CQRS із використанням бібліотеки MediatR 14 [26]. Кожна дія має пару «запит та обробник»: запит описує намір користувача або системи, а обробник містить алгоритм виконання. Усі обробники реєструються через сканування збірки одним викликом (див. лістинг 3.4):

Лістинг 3.4 – Програмний код

```
services.AddMediatR(options =>
    options.RegisterServicesFromAssemblyContaining(
        typeof(ServiceCollectionExtensions)));
```

Застосунок має 10 обробників, які охоплюють основні функціональні області. До транзакцій належать Create, Update, Delete, Get і ExportCsv; до дашборду: GetDashboardData; до аналітики: GetAnalyticsData та GetAnalyticsTrendData; до бюджетів належить GetBudgetsData.

Найбільшу кількість логіки містить GetDashboardDataHandler. Обробник бере один набір транзакцій і на його основі формує кілька результатів: KPI поточного місяця з порівнянням до попереднього, дані для стовпчастого графіка за шість місяців, розподіл витрат за категоріями для donut-діаграми та список транзакцій для таблиці. Запит до бази виконується один раз. Після завантаження даних обробник агрегує їх у пам’яті засобами LINQ, тому кількість звернень до бази не зростає під час побудови дашборду.

Обробник GetTransactionsHandler реалізує серверну пагінацію: запит GetTransactionsRequest наслідує абстрактний запис PagedRequest з властивостями Page та PageSize, а відповідь обертається у типізований PagedResponse<T>. Фільтрація за типом, місяцем, категорією, рахунком та текстовим пошуком виконується на рівні SQL через EF.Functions.Like, що важливо для продуктивності при великій кількості записів.

Для маппінгу між EF-сутностями та доменними моделями застосовуються два підходи: ручний маппер TransactionsMapper у вигляді статичного класу з методами-розширеннями та автоматичний code-generation маппер AccountsMapper на основі бібліотеки Riok.Mapperly. Mapperly генерує код маппера під час компіляції, не використовуючи рефлексію в рантаймі, що забезпечує нульові накладні витрати на продуктивність [27].

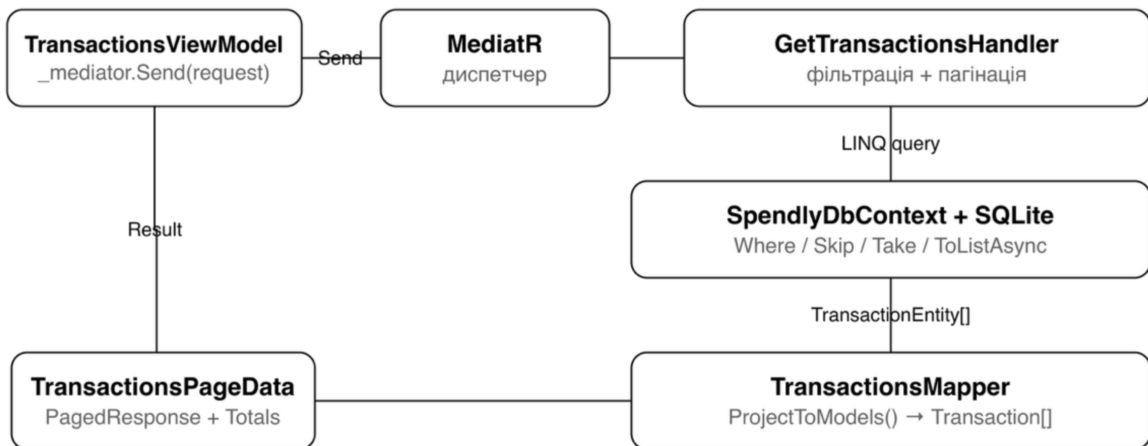

Рисунок 3.3 – Потік виконання запиту GetTransactionsRequest через шари системи

Реєстрація всіх сервісів виконана через патерн Extension Methods. Кожен шар надає метод розширення IServiceCollection, що інкапсулює власну конфігурацію. В App.xaml.cs ці методи викликаються послідовно (див. лістинг 3.5):

Лістинг 3.5 – Програмний код

```
services.AddInfrastructure();    // репозиторії, UoW, Queries
services.AddApplication();       // MediatR + handlers
services.AddSqlite();            // DbContext + SQLite provider
```

Після старту застосунку виконується ініціалізація бази даних через IDbInitializer.InitializeAsync(), яка застосовує всі незастосовані міграції та гарантує наявність рахунку типу Cash за замовчуванням. Міграції керовані кодом (code-first) і зберігаються у проекті Spendly.Sqlite. На момент реалізації наявні три міграції: Initial (створення таблиць accounts, categories, transactions), AddDefaultCategories (вставка 17 стандартних категорій із фіксованими GUID), AddLimits (додавання таблиці category_limits з унікальним індексом).

Фіксовані GUID у міграції категорій є свідомим рішенням: вони дозволяють безпечно повторно застосовувати міграцію та використовувати ці ідентифікатори в тестах без додаткового пошуку по базі.

Тестування шарів Application та Infrastructure реалізоване у проекті Spendly.Tests з використанням xUnit v3, FluentAssertions та провайдера EF InMemory. Кожен тест отримує власний ізольований контекст бази даних через TestDbContextFactory.Create(), що створює SpendlyDbContext з унікальною InMemory базою. Це гарантує повну ізоляцію тестів між собою.

Підготовка тестових даних централізована у класі TestDataBuilder, що реалізує патерн Builder для тестів: замість дублювання логіки створення сутностей у кожному тесті клас надає виразні методи AddAccount(), AddCategory(), AddExpense(), AddIncome(), AddLimit(). Це підвищує читабельність тестів та спрощує їх підтримку [25].

Покриття включає 35 тест-кейсів для всіх ключових сценаріїв: CRUD-операції над транзакціями, розрахунок KPI дашборду (включно з порівнянням місяців та заповненням нулями відсутніх місяців у 6-місячному графіку), алгоритм ризик-аналізу бюджету (перевірка порогів 75%/100%), розрахунок часток витрат за рахунками та фільтрація CSV-експорту за комбінацією параметрів.

Таким чином, реалізація шарів даних та бізнес-логіки застосунку Spendly демонструє практичне застосування принципів Clean Architecture, патернів Repository, Unit of Work та CQRS, а чіткий поділ відповідальностей між шарами забезпечує тестованість кожного компонента в ізоляції.

3.2 Опис екранів та інтерфейсу застосунку

Інтерфейс Spendly побудований на базі Windows Presentation Foundation (WPF) і охоплює чотири основні екрани: Dashboard, Transactions, Analytics та Budgets. Між екранами користувач переміщується через бічну навігаційну панель (Sidebar). Вона містить лише назву застосунку та чотири пункти меню, без зайвих елементів. Світла кольорова схема формує єдине візуальне середовище: тло у світлиї відтінках, за акцентні кольори обрано фіолетовий, бірюзовий і рожевий. Контрастність кольорових пар відповідає когнітивним

вимогам до сприйняття числових даних на темному тлі: людське зорове сприйняття краще розрізняє об'єкти, коли яскравість тла і елемента відрізняється щонайменше на 50% [28].

Мінімалістична навігація є продуманим рішенням. За рекомендаціями щодо UX аналітичних застосунків, скорочення навігаційних кроків прямо знижує когнітивне навантаження і пришвидшує виконання типових завдань [29].

Екран Dashboard. Dashboard є головною точкою огляду фінансового стану за обраний місяць. У правому верхньому куті розміщений Month Picker: натискаючи на нього, користувач перемикається між місяцями без переходу на окремий екран.

Верхній ряд займають чотири KPI-картки.Income (month) показує загальний дохід і відсоткову зміну відносно попереднього місяця (наприклад, 34,2%. Expenses (month) дублює логіку, але для витрат: 7,0% у січні 2026). Balance рахується як різниця між доходами та витратами і додатково показує частку від місячного доходу. Четверта картка, Transactions, виводить кількість операцій за місяць.

Кольорова мітка при кожному показнику змінює відтінок залежно від напрямку тренду. Зростання доходів відображено зеленим, а зростання витрат показано червоним. Семантичне кольорове кодування спирається на вроджені асоціації людського сприйняття: зелений сигналізує про безпеку, червоний повідомляє про загрозу, тому інтерпретація відбувається без свідомого читання числа [28].

Під картками розміщено два графіки. Ліворуч знаходиться стовпчаста діаграма «Incomes / Expenses» за шість місяців, побудована бібліотекою LiveChartsCore: паралельні стовпці бірюзового (доходи) та червоного (витрати) кольорів. Праворуч розміщується donut-графік «By categories» із легендою для читання: категорії відсортовані за спаданням частки, тому Rent (32%) і Investments (28%) опиняються вгорі без додаткового налаштування.

Нижню частину екрана займає компактна таблиця «Recent transactions» (три останні записи з іконкою категорії, назвою, підкатегорією, сумою і датою).

Кнопка «Show all» веде до повного списку. Весь екран загалом продемонстровано на рисунку 3.4

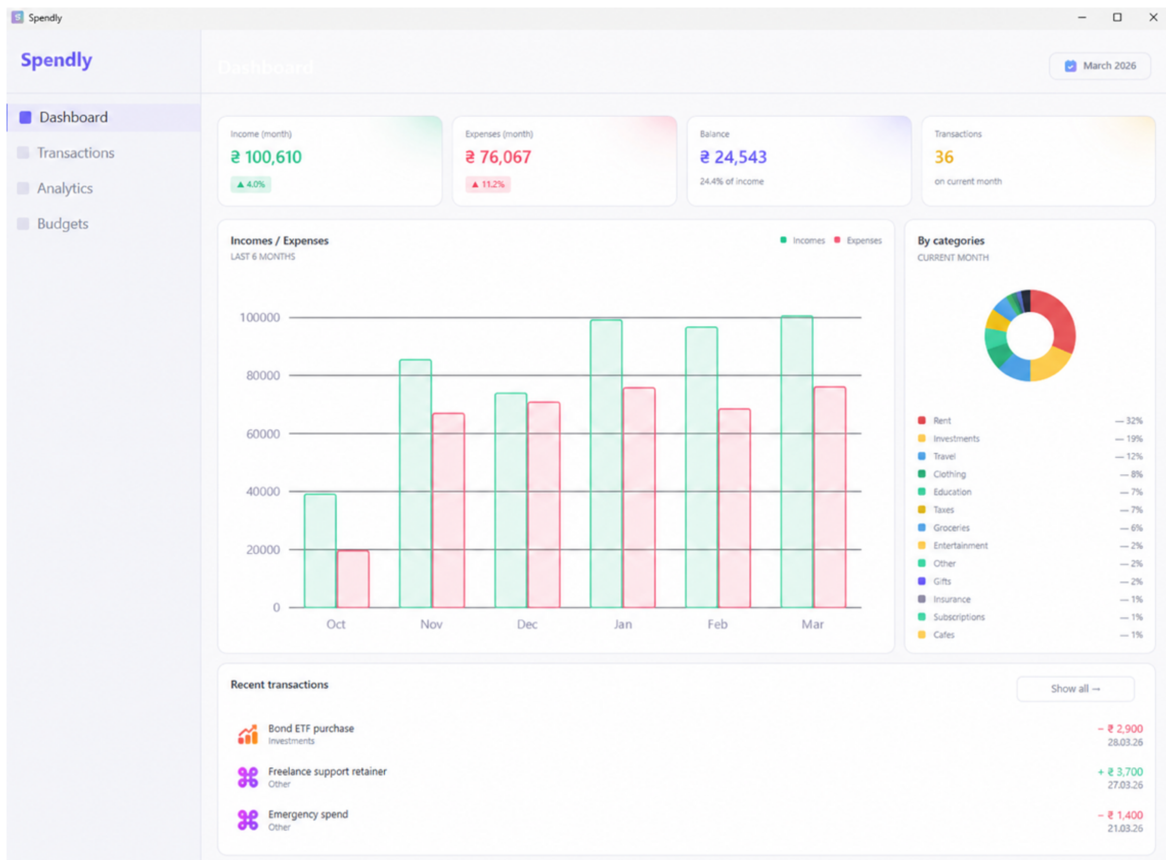

Рисунок 3.4 – Екран Dashboard застосунку Spendly

Екран Transactions. Transactions (див. рис. 3.5) охоплює повний цикл управління операціями: перегляд, пошук, фільтрацію, редагування та видалення. Панель у верхній частині групує фільтри за типом транзакції (All / Expenses / Incomes), часом, категорією і рахунком. Поруч знаходиться поле повнотекстового пошуку по коментарю, назві рахунку та категорії. Два окремих елементи управління розміщені праворуч: «Export CSV» і «New».

Одразу під фільтрами розміщено три лічильники. Records shown показує кількість знайдених записів після застосування поточних фільтрів. Total expenses і Total income перераховуються автоматично при будь-якій зміні критеріїв відбору.

Основний вміст екрана складає таблиця з колонками Name / Category, Amount, Account, Date та Actions. Сума забарвлена: плюс і зелений маркер позначає дохід, мінус і червоний означає витрату. Пагінація виводить рядок «Showing 1-10 of 254» і навігаційні кнопки. Форми додавання і редагування відкриваються в окремих Views і містять поля для типу, рахунку, категорії, суми, дати та коментаря з валідацією на рівні ViewModel.

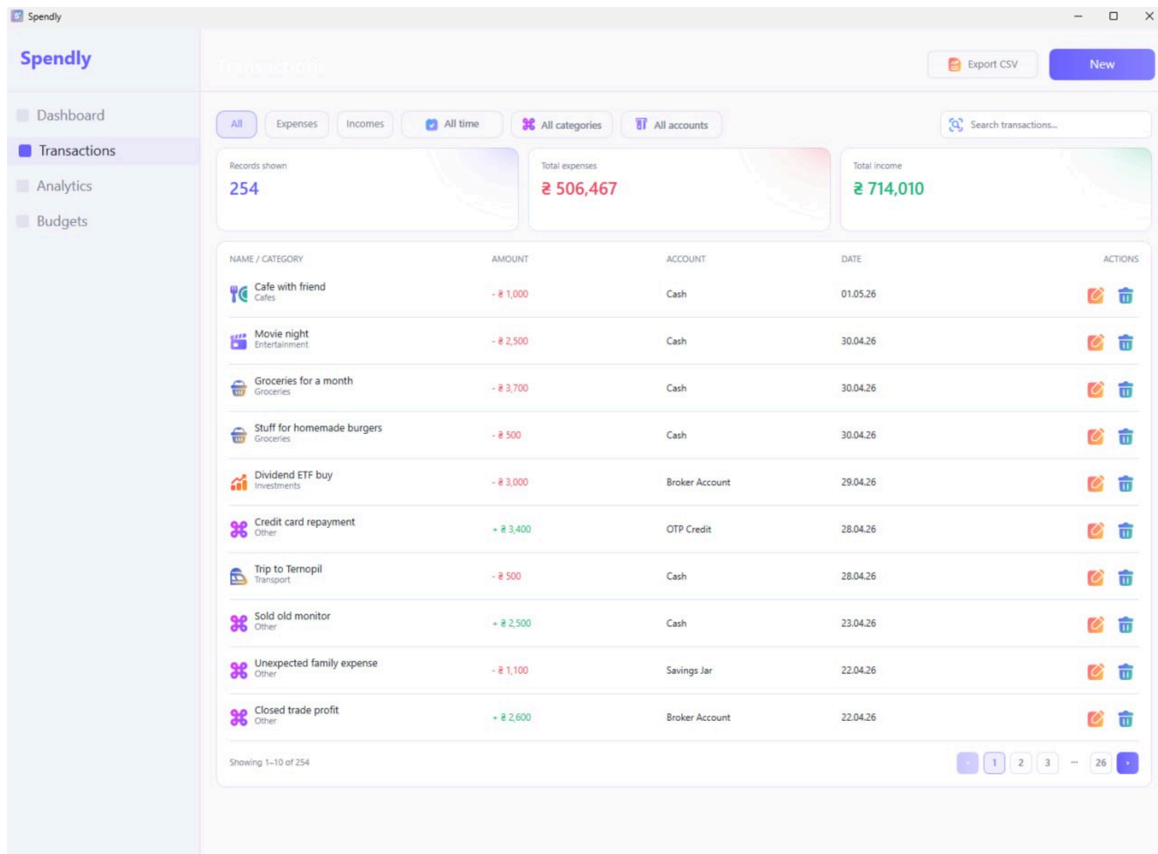

Рисунок 3.5 – Екран Transactions застосунку Spendly

Екран Analytics. Analytics є найбільш інформаційно щільним екраном застосунку. Він поділений на дві секції: General і Account breakdown.

У секції General розміщено чотири картки з різним семантичним навантаженням. Перша сигналізує про перевитрату: або виводить категорію-порушника з фактичною і граничною сумою, або показує «No overspending this month». Друга картка (Underused) вказує на категорію з найменшим відсотком використання ліміту; у прикладі на скриншоті Groceries використана лише на ₴0

з ₴8,000. Третя картка («All good / Risk») видає попередження про категорії у зоні 75-99% від встановленого ліміту або підтверджує відсутність ризиків. Четверта, Forecast by end of month, передбачена архітектурою, проте у поточній версії повертає «₴»: алгоритм прогнозування на даний момент не реалізований.

Нижче знаходиться горизонтальна панель категорій і лінійний графік. Вибравши Cafes, Education або будь-яку іншу категорію зі списку, користувач отримує динаміку витрат за шість місяців у вигляді плавної кривої. Виявлення сезонних патернів стає можливим без додаткової обробки даних.

Account breakdown показує, між якими рахунками розподілені витрати. Дві картки («Top spending account» і «Second by spending») виводять назву рахунку, місячну суму і частку від загальних витрат. Під ними горизонтальна барна діаграма охоплює всі рахунки: Cash 44%, Broker Account 25%, PrivatBank 18%, Savings Jar 7%, OTP Credit 6%. Кожна смуга пофарбована окремим кольором і підписана відсотком (рис. 3.6).

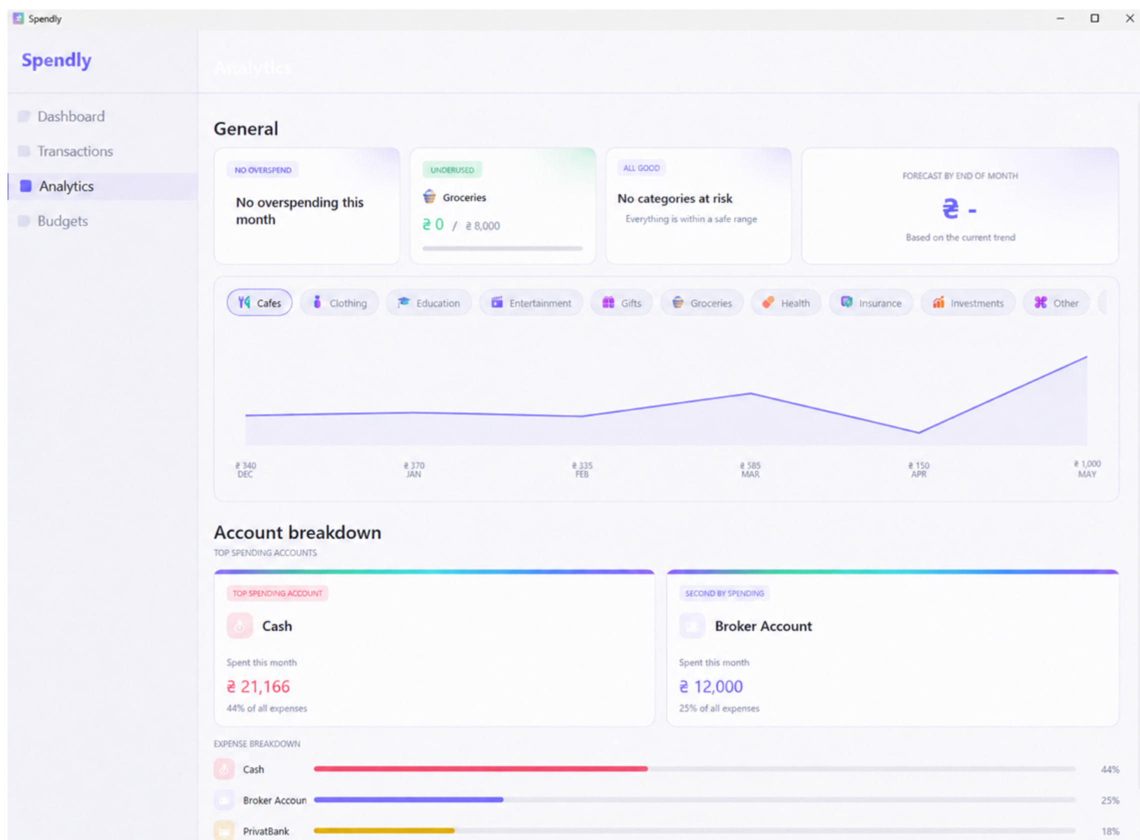

Рисунок 3.6 – Екран Analytics застосунку Spendly

Екран Budgets. Budgets відповідає за бюджетні ліміти в розрізі категорій. Три KPI-картки вгорі фіксують: Total budget (сума всіх лімітів), Spent (фактичні витрати за підконтрольними категоріями), Remaining (невикористаний залишок).

Блок "Overall budget progress" агрегує стан усього бюджету в один горизонтальний прогрес-бар з позначкою відсотка використання. Під ним розміщується окремий рядок для кожної категорії: іконка, назва, співвідношення витрат до ліміту, прогрес-бар і текст на зразок «40%, Remaining ₴1,500». Колір смуги несе практичне значення: зелений показує нормальний рівень, жовтий наближається до межі, червоний означає перевищення. Триступенева шкала відповідає принципу преатентивного сприйняття: мозок фіксує колірний сигнал небезпеки раніше, ніж читає цифру [29]. Кнопка «New limit» у верхньому куті відкриває форму нового ліміту. Іконка кошика праворуч від кожного рядка видаляє обраний ліміт.

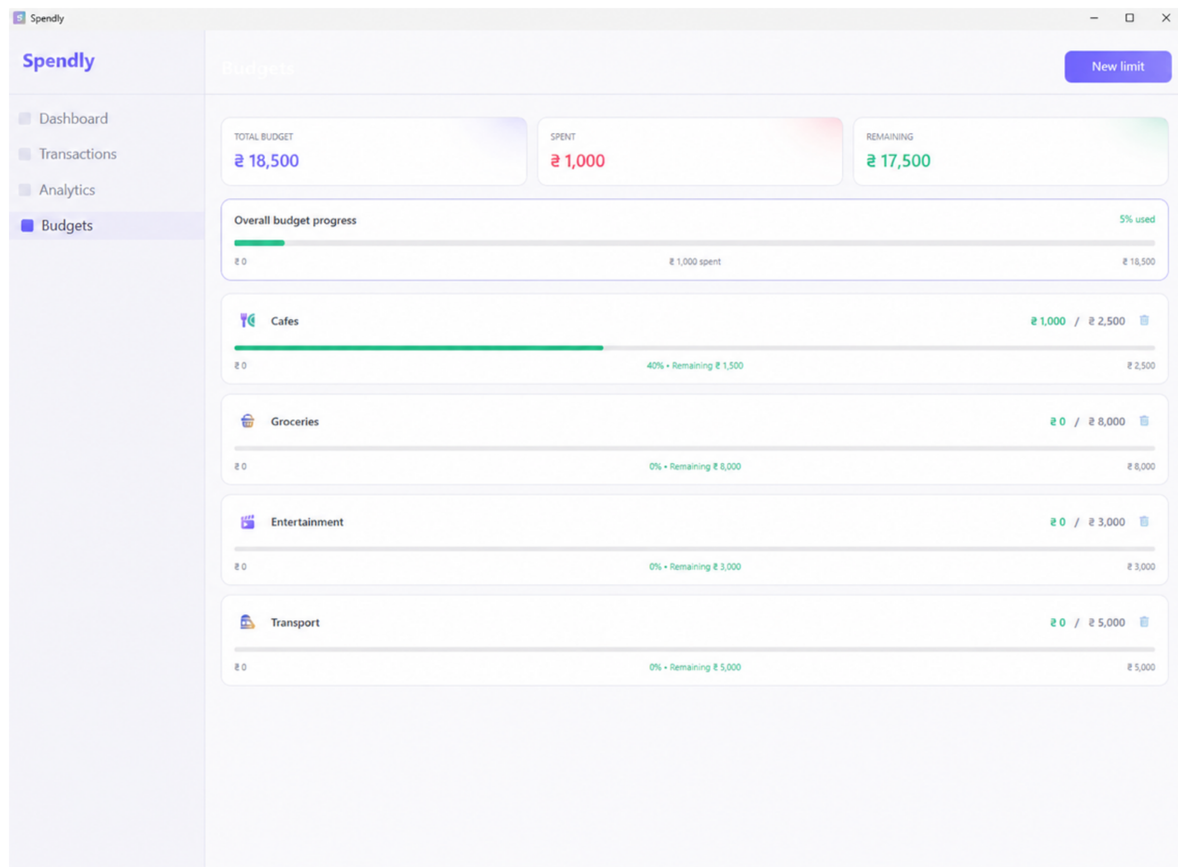

Рисунок 3.7 – Екран Budgets застосунку Spendly

Spendly витримує єдину дизайн-систему на всіх чотирьох екранах: темна схема, уніфіковані картки із заокругленими кутами, SVG-іконки категорій і стале кольорове кодування: бірюзовий/зелений для доходів, рожевий/червоний для витрат. Консистентність скорочує час адаптації нових користувачів [].

3.3 Тестування системи

Автоматизоване тестування у проекті Spendly виконує подвійну функцію: підтверджує коректність реалізованої логіки та захищає від регресій при подальшому розвитку коду. Проект дотримується стратегії модульного тестування (див. рис. 3.8) з елементами інтеграційного на рівні бази даних [30].

Інструменти та підхід до тестування. Тестовий проект Spendly.Tests побудований на трьох інструментах. xUnit v3 організовує та запускає тести. Він є де-факто стандартом для .NET завдяки паралельному виконанню та зручній моделі життєвого циклу. FluentAssertions замінює стандартні Assert-виклики виразними твердженнями: замість Assert.Equal(expected, actual) розробник пише actual.Should().Be(expected), що робить повідомлення про помилки значно інформативнішими. EF Core InMemory дає змогу виконувати запити через реальний SpendlyDbContext без підключення до SQLite-файлу. База існує в пам'яті лише на час тесту.

Принциповою рисою підходу є тестування обробників у повній зв'язці з базою даних, а не в ізоляції через mock-об'єкти. Хендлер перевіряється разом із LINQ-запитами, маппінгом сутностей і поведінкою EF Core при конкретних умовах. Науковці відносять подібний підхід до інтеграційного тестування шару застосування. Він виявляє цілий клас помилок, які mock-репозиторії просто не здатні зафіксувати [31].

Ізоляцію між тестами забезпечує TestDbContextFactory, продемонстрований у лістингу 3.6. Кожен виклик Create() формує окремий InMemory-контекст із унікальним ім'ям:

Лістинг 3.6 – Програмний код

```
public static class TestDbContextFactory
{
    public static SpendlyDbContext Create()
    {
        var options = new
DbContextOptionsBuilder<SpendlyDbContext>()
            .UseInMemoryDatabase(Guid.NewGuid().ToString())
            .Options;

        return new SpendlyDbContext(options, null);
    }
}
```

Guid.NewGuid().ToString() як ім'я бази гарантує чистий стан для кожного тесту незалежно від порядку запуску та паралельності.

Підготовку тестових даних централізує клас TestDataBuilder, побудований за патерном Builder. Fluent-інтерфейс із методами AddAccount(), AddCategory(), AddExpense(), AddIncome(), AddLimit() усуває дублювання коду між тестами. Кожен метод зберігає сутність безпосередньо в контекст і повертає builder для ланцюгового виклику (див. лістинг 3.7):

Лістинг 3.7 – Програмний код

```
var builder = await new TestDataBuilder(dbContext)
    .AddAccount("Готівка")
    .AddCategory("Їжа")
    .AddExpense(500m, month: new YearMonth(2026, 3))
    .AddExpense(300m, month: new YearMonth(2026, 3))
    .AddIncome(2000m, month: new YearMonth(2026, 3))
    .BuildAsync();
```

Читаючи підготовку тесту, одразу зрозуміло, який сценарій моделюється. Немає необхідності у зануренні в деталі створення об'єктів. Принцип DRY тут не просто декларація, а практичний результат.

Тести модуля Analytics. Аналітичний модуль концентрує найскладнішу бізнес-логіку застосунку. Саме тут транзакції перетворюються на висновки: алгоритм визначає перевитрачені та недовикористані категорії, виявляє ризикові

статті витрат і розраховує частки по рахунках. Помилка в жодному з цих розрахунків не спричинить виняткової ситуації, вона лише тихо спотворить дані на екрані. Через цю специфіку модуль отримав найбільше тестів: дев'ять, розподілених між двома тест-класами (рис. 3.8).

AnalyticsPageHandlerTests охоплює шість тестів:

– Should_Calculate_Account_Shares: два рахунки з витратами 300 і 700 грн повинні отримати частки 30% і 70% відповідно;

– Should_Ignore_Income_In_Accounts: при розрахунку часток враховуються виключно транзакції типу Expense. Включення доходів спотворило б відсотки;

– Should_Order_By_Amount_Descending: категорії у звіті сортуються від найбільшої суми витрат до найменшої;

– Should_Return_Empty_Overspent_When_No_Category_Exceeded_Limit: якщо жодна категорія не перевищила ліміт, список overspent повертається порожнім, а не null;

– Should_Return_Overspent_And_Underused_Categories: ключовий тест: одночасно перевіряється найбільш перевитрачена категорія (витрати > ліміт) та найменш використана (найнижчий відсоток від ліміту);

– Should_Return_Risk_Items: категорії, що перевищили 75% ліміту, але не досягли 100%, мають потрапити до списку ризиків.

AnalyticsTrendHandlerTests перевіряє три сценарії:

– Should_Filter_By_Category: при передачі конкретного CategoryId відповідь містить дані лише по ній;

– Should_Ignore_Income: тренди будуються виключно на витратах;

– Should_Return_6_Months_Data: відповідь завжди містить рівно шість точок. Відсутні місяці заповнюються нулями, а не пропускаються.

{} Analytics (9 tests) Success
AnalyticsPageHandlerTests (6 tests) Success
Should_Calculate_Account_Shares Success
Should_Ignore_Income_In_Accounts Success
Should_Order_By_Amount_Descending Success
Should_Return_Empty_Overspent_When_No_Category_Exceeded_Limit Success
Should_Return_Overspent_And_Underused_Categories Success
Should_Return_Risk_Items Success
AnalyticsTrendHandlerTests (3 tests) Success
Should_Filter_By_Category Success
Should_Ignore_Income Success
Should_Return_6_Months_Data Success

Рисунок 3.8 – Результати виконання тестів модуля Analytics

Тести модуля Budgets. Бюджетний модуль розраховує прогрес витрат відносно встановлених лімітів по категоріях. Від його коректності залежить основна практична цінність застосунку: хибний прогрес формує у користувача неправдиве відчуття контролю над фінансами. Чотири тести охоплюють базовий розрахунок і граничні випадки, оскільки саме в них найчастіше ховаються приховані помилки (рис. 3.9).

– Should_Calculate_Budget_And_Spend: для категорії з лімітом 1000 грн і витратами 600 грн хендлер повертає Limit = 1000, Spend = 600, Progress = 0.6;

– Should_Ignore_Income_Transactions: транзакції типу Income не входять до суми витрат при розрахунку прогресу;

– Should_Ignore_Transactions_From_Other_Months: витрати з інших місяців не впливають на прогрес поточного. Без фільтрації по місяцю ліміт виглядав би перевищеним некоректно;

– Should_Return_Zero_Spend_When_Category_Has_No_Transactions: категорія з лімітом, але без жодної транзакції за місяць, повертає Spend = 0 і Progress = 0. Виняткова ситуація не виникає.

Budgets (4 tests) Success
GetBudgetsDataHandlerTests (4 tests) Success
Should_Calculate_Budget_And_Spend Success
Should_Ignore_Income_Transactions Success
Should_Ignore_Transactions_From_Other_Months Success
Should_Return_Zero_Spend_When_Category_Has_No_Transactions Success

Рисунок 3.9 – Результати виконання тестів модуля Budgets

Тести модуля Dashboard. Дашборд є першим екраном після запуску застосунку. Він агрегує результати кількох незалежних обчислень: KPI місяця, шестимісячний графік, розбивку по категоріях і список транзакцій. Складність полягає не в кожному обчисленні окремо, а в їх спільній поведінці при різних вхідних параметрах, зокрема, коли вибраний місяць не передано. Чотири тести перевіряють саме ці точки нестабільності (рис. 3.10):

– Handle_Should_Return_6_Month_Points_And_Fill_Missing_Months_With_Zero. Транзакції навмисно створено лише для двох із шести місяців. Відповідь повинна містити рівно шість точок, де чотири відсутніх місяці заповнено нулями. Без заповнення графік відображав би лише місяці з даними;

– Handle_Should_Return_All_Time_Kpi_When_Date_Is_Null. При selectedMonth = null хендлер агрегує всі транзакції без часового фільтру та повертає загальний баланс за весь час;

– Handle_Should_Return_Categories_Only_For_Expenses_With_Category. До donut-діаграми потрапляють лише витрати з призначеною категорією. Транзакції без категорії та всі доходи ігноруються;

– Handle_Should_Return_Selected_Month_Kpi_And_Current_Month_Transactions. Комплексний сценарій: KPI розраховується для вибраного місяця з порівнянням до попереднього, а таблиця транзакцій показує записи поточного системного місяця.

Dashboard (4 tests) Success
GetDashboardDataHandlerTests (4 tests) Success
Handle_Should_Return_6_Month_Points_And_Fill_Missing_Months_With_Zero Success
Handle_Should_Return_All_Time_Kpi_When_Date_Is_Null Success
Handle_Should_Return_Categories_Only_For_Expenses_With_Category Success
Handle_Should_Return_Selected_Month_Kpi_And_Current_Month_Transactions Success

Рисунок 3.10 – Результати виконання тестів модуля Dashboard

Тести модуля Transactions. Сторінка транзакцій формує фундамент усього застосунку. Без коректної роботи операцій читання, запису і фільтрації жоден інший модуль не може функціонувати правильно. Окремої уваги заслуговує CSV-експорт: єдина функція, що виходить за межі застосунку і формує файл для зовнішнього використання. Дев'ять тестів розподілено між п'ятьма тест-класами. Кожна операція перевіряється у штатному сценарії та в умовах некоректних вхідних даних (рис. 3.11).

CreateTransactionHandlerTests Handle_Should_Create_Transaction – після виклику хендлера транзакція присутня в базі, а поля Amount, Type, CategoryId, Comment, DateUtc відповідають вхідним даним. DeleteTransactionHandlerTests

Handle_Should_Delete_Existing_Transaction – після видалення транзакція не знаходиться в базі.

Handle_Should_Do_Nothing_When_Transaction_Does_Not_Exist – спроба видалити неіснуючий запис не кидає виняток і не змінює стан бази. ExportCsvHandlerTests

Handle_Should_Create_Csv_File_With_Filtered_Transactions – хендлер застосовує фільтри (місяць, тип, категорія) і записує у файл лише відповідні транзакції у коректному форматі. GetTransactionsHandlerTests

Handle_Should_Filter_By_Month_Category_And_Account – з бази з транзакціями для різних місяців, категорій і рахунків повертаються лише ті, що відповідають усім трьом критеріям одночасно.

Handle_Should_Filter_By_Type_And_Calculate_Totals – TotalExpenses та TotalIncome у відповіді розраховуються коректно навіть при активному фільтрі по типу.

Handle_Should_Return_Paged_Transactions_Ordered_By_Date_Descending – при Page = 2, PageSize = 3 повертаються правильні записи, відсортовані від найновішої дати до найстарішої. UpdateTransactionHandlerTests

Handle_Should_Do_Nothing_When_Transaction_Does_Not_Exist – оновлення неіснуючого запису завершується без виняткової ситуації.

Handle_Should_Update_Existing_Transaction – після оновлення поля Amount, Comment, CategoryId відображають нові значення.

{} Transactions (9 tests) Success
CreateTransactionHandlerTests (1 test) Success
Handle_Should_Create_Transaction Success
DeleteTransactionHandlerTests (2 tests) Success
Handle_Should_Delete_Existing_Transaction Success
Handle_Should_Do_Nothing_When_Transaction_Does_Not_Exist Success
ExportCsvHandlerTests (1 test) Success
Handle_Should_Create_Csv_File_With_Filtered_Transactions Success
GetTransactionsHandlerTests (3 tests) Success
Handle_Should_Filter_By_Month_Category_And_Account Success
Handle_Should_Filter_By_Type_And_Calculate_Totals Success
Handle_Should_Return_Paged_Transactions_Ordered_By_Date_Descending Success
UpdateTransactionHandlerTests (2 tests) Success
Handle_Should_Do_Nothing_When_Transaction_Does_Not_Exist Success
Handle_Should_Update_Existing_Transaction Success

Рисунок 3.11 – Результати виконання тестів модуля Transactions

Проект містить 26 тестів у чотирьох модулях: Analytics (9), Transactions (9), Budgets (4), Dashboard (4). Усі виконуються успішно. Скріншоти тестувань у середовищі JetBrains Rider підтверджують це (рис. 3.8-3.11).

Кожен тест дотримується структури AAA: спочатку TestDataBuilder готує дані, потім хендлер отримує запит, після – FluentAssertions перевіряє результат. Покриття не обмежується «щасливим шляхом». Неіснуючі ідентифікатори,

порожні вибірки, транзакції поза обраним місяцем – усі ці граничні умови є окремими тест-кейсами, адже більшість прихованих дефектів саме там [30].

3.4 Висновки до третього розділу

Розділ 3 присвячено практичній реалізації застосунку Spendly. Реалізовані рішення підтверджують архітектурні та проєктні рішення, ухвалені у розділі 2, і демонструють їх працездатність у реальному коді.

У підпункті 3.1 описано структуру та реалізацію системи. Дев'ять проєктів рішення розподілені між чотирма шарами Clean Architecture: Domain зберігає доменні моделі без зовнішніх залежностей, Application зосереджує бізнес-логіку через десять обробників MediatR, Infrastructure реалізує доступ до даних через Entity Framework Core 9 із провайдером SQLite, а WPF-застосунок формує шар представлення.

У підпункті 3.2 описано чотири реалізовані екрани застосунку. Dashboard агрегує KPI поточного місяця, шестимісячну діаграму доходів і витрат, donut-графік категорій та таблицю останніх транзакцій. Transactions охоплює повний CRUD-цикл з комбінованою фільтрацією, пагінацією та CSV-експортом відфільтрованих даних. Analytics виявляє перевитрачені, недовикористані та ризикові категорії, відображає шестимісячний тренд по обраній категорії і розподіл витрат між рахунками. Budgets керує лімітами по категоріях і візуалізує прогрес через триколірну шкалу прогрес-барів.

У підпункті 3.3 описано стратегію тестування. Проект Spendly.Tests містить 26 тестів у чотирьох модулях: Analytics (9), Transactions (9), Budgets (4), Dashboard (4). Кожен тест отримує ізольований InMemory-контекст через TestDbContextFactory і використовує TestDataBuilder для виразної підготовки сценарію. Усі 26 тестів успішно проходять у середовищі JetBrains Rider. Покриття охоплює не лише штатні сценарії, а й граничні умови: неіснуючі ідентифікатори, порожні вибірки, транзакції поза обраним місяцем і відсутні місяці у графіках.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

У сучасних умовах розвитку інформаційних технологій питання безпеки життєдіяльності та охорони праці залишаються невід'ємною складовою професійної діяльності розробника програмного забезпечення. У даному розділі розглянуто основні чинники небезпеки, що виникають у процесі роботи з комп'ютерною технікою, а також нормативно-правові вимоги щодо організації безпечних умов праці.

4.1 Актуальність безпеки життєдіяльності людини

Безпека життєдіяльності є міждисциплінарною галуззю знань, що вивчає небезпеки, які загрожують людині в процесі її існування та трудової діяльності, і розробляє методи захисту від них. В умовах сучасного технологічного суспільства питання безпеки набувають особливої актуальності, оскільки зростання рівня технічної оснащеності виробництва супроводжується появою нових ризиків для здоров'я та життя працівників [32].

Людина протягом своєї діяльності перебуває у взаємодії з навколишнім середовищем – природним, техногенним і соціальним. Кожне з цих середовищ є потенційним джерелом небезпек, що можуть реалізуватися у вигляді нещасних випадків, професійних захворювань або надзвичайних ситуацій. Відповідно до сучасних концепцій безпеки життєдіяльності, основним об'єктом захисту є людина, а головною метою – збереження її здоров'я, працездатності та життя.

Особливе місце серед чинників небезпеки займають ризики, пов'язані з професійною діяльністю. Розробка програмного забезпечення, попри відсутність важкої фізичної праці, формує специфічний комплекс загроз: тривала робота за монітором спричиняє зорове стомлення, вимушена статична поза негативно впливає на опорно-руховий апарат, а високе когнітивне навантаження є передумовою психоемоційного вигорання [34].

Зорове навантаження є одним із ключових ризиків для користувачів ПК. Тривала робота з монітором призводить до підвищеного навантаження на очі внаслідок мерехтіння зображення, низького контрасту, незадовільної яскравості екрана та неправильного розміщення джерел освітлення. Монітор слід розташовувати на відстані 600–700 мм від очей оператора, а його верхній край має знаходитись на рівні або дещо нижче горизонтальної лінії зору. Рівень освітленості робочої поверхні при роботі з відеотерміналами має відповідати ДБН В.2.5-28:2018 і становити не менше 300–500 лк для роботи з документами та 200 лк для роботи виключно з екраном.

Необхідно уникати прямого потрапляння природного або штучного світла на поверхню монітора, що спричиняє відблиски та підвищує зорове стомлення. Кожні 20 хвилин роботи з монітором рекомендується переводити погляд на віддалені об'єкти протягом 20 секунд. Статичне навантаження на опорно-руховий апарат виникає внаслідок тривалого перебування в сидячому положенні з мінімальною руховою активністю.

За таких умов підвищується навантаження на хребет, особливо в поперековому та шийному відділах, а також на м'язи плечового пояса та передпліч. Робоче місце оператора має забезпечувати висоту робочої поверхні 680–800 мм із можливістю регулювання, кут між стегном і тулубом у діапазоні 90–110 градусів та опору для попереку з регульованою висотою. Крісло має бути регульованим за висотою сидіння, кутом нахилу спинки та підлокітниками, що знижують статичне напруження м'язів плечового пояса при тривалій роботі з клавіатурою.

Дотримання цих вимог є необхідною умовою профілактики захворювань опорно-рухового апарату, зокрема остеохондрозу та тунельного синдрому – поширених професійних захворювань серед користувачів ПК [35]. Психоемоційне навантаження формується в умовах розв'язання складних аналітичних і архітектурних завдань, що створює передумови для виникнення хронічного стресу та синдрому емоційного вигорання.

Науково обґрунтованим засобом його профілактики є регламентовані перерви: при восьмигодинному робочому дні оператор ПК робить перерви по 10–15 хвилин кожні дві години роботи з екраном, а загальний час інтенсивної роботи з ВДТ не повинен перевищувати чотирьох годин на зміну. Мікроклімат виробничого приміщення також є важливим чинником безпеки життєдіяльності. Для приміщень з комп'ютерною технікою встановлюється оптимальна температура в холодний період 22–24°C, у теплий – 23–25°C, відносна вологість – 40–60%.

Рівень шуму у приміщеннях для розумової праці не повинен перевищувати 50 дБА, оскільки вище цієї межі у працівників знижується концентрація уваги та погіршується самопочуття [35].

Таким чином, актуальність безпеки життєдіяльності в контексті розробки програмного забезпечення визначається сукупністю фізичних, ергономічних та психоемоційних ризиків, що супроводжують цей вид діяльності. Системний підхід до їх усунення полягає у правильній організації робочого місця, дотриманні санітарно-гігієнічних норм і режиму праці та відпочинку.

4.2 Охорона праці

Охорона праці охоплює правові, соціально-економічні, організаційно-технічні, санітарно-гігієнічні та лікувально-профілактичні заходи, спрямовані на збереження здоров'я і працездатності людини в процесі трудової діяльності. Закон України «Про охорону праці» покладає на роботодавця обов'язок створювати в кожному структурному підрозділі умови праці, що відповідають нормативно-правовим актам, і дотримуватись вимог законодавства щодо прав працівників [32].

Правову основу галузі формують також Конституція України, яка гарантує громадянам право на безпечні й здорові умови праці, Кодекс законів про працю та Закон «Про загальнообов'язкове державне соціальне страхування». Практична організація охорони праці на підприємстві охоплює навчання

персоналу, розслідування нещасних випадків і професійних захворювань, контроль умов праці на робочих місцях. Мікроклімат, шум, вібрація та освітлення – основні фізичні чинники виробничого середовища, що безпосередньо впливають на стан здоров'я оператора ПК.

Санітарні норми встановлюють для приміщень з комп'ютерною технікою оптимальну температуру в холодний період 22–24°С, у теплий – 23-25°С, відносну вологість – 40–60%. Допустимий рівень шуму не перевищує 50 дБА: вище цієї межі у працівників знижується концентрація уваги та погіршується самопочуття. Вібрація для комп'ютерних робочих місць зазвичай не є критичним чинником, але підлягає обов'язковому контролю. Освітлення робочого місця регламентує ДБН В.2.5-28:2018 [33]. Недостатнє або нерівномірне освітлення, блиски на екрані монітора – прямий шлях до зорової втоми й головного болю. Горизонтальна поверхня столу при роботі з екранними пристроями потребує освітленості 300–500 лк.

Пожежна безпека комп'ютерних приміщень забезпечується відповідно до Правил пожежної безпеки в Україні та ДБН В.1.1-7-2016. Приміщення з ПК належать до категорії В за пожежною небезпекою. Роботодавець зобов'язаний підтримувати евакуаційні шляхи у вільному стані, встановити первинні засоби пожежогасіння й системи оповіщення.

Захист від електромагнітних полів і випромінювання регулюють Норми радіаційної безпеки України НРБУ–97/Д–2000. Сучасні монітори виробляють з урахуванням міжнародних стандартів і генерують мінімальний рівень випромінювання. Попри це, тривала робота без перерв залишається небажаною. НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями» [34] зобов'язує роботодавця провести оцінку ризиків на робочих місцях з ВДТ, вжити заходів для їх зниження та організувати медичні огляди операторів. Безперервна робота з екраном – не більше двох годин, а загальний час за монітором упродовж зміни – не більше шести годин.

ВИСНОВКИ

КРБ присвячено проєктуванню та реалізації настільного застосунку Spendly для обліку й планування особистих фінансових витрат. За підсумками виконаної роботи сформульовано наступні висновки.

У першому розділі досліджено предметну область особистого фінансового менеджменту й розглянуто наявні ринкові рішення. Аналіз підтвердив: головною практичною проблемою залишається не брак інструментів, а нерегулярність ведення записів. Порівняння YNAB, Wallet by BudgetBakers, Money Manager Ex і Quicken Simplifi виявило чіткий поділ між cloud-first і local-first підходами. Орієнтирами для дипломної роботи стали MMEX як автономний desktop-інструмент і Wallet як еталон сучасної аналітики. На основі аналізу сформульовано функціональні та нефункціональні вимоги, побудовано модель актора й визначено десять ключових варіантів використання системи.

Другий розділ охоплює повне проєктування застосунку. Архітектурну основу Spendly складає чотиришарова Clean Architecture із патерном MVVM на рівні представлення та CQRS на прикладному рівні. Чотири ключові сутності: Account, Category, Transaction і CategoryLimit, описують предметну область. Поведінку системи формалізовано через три типи UML-діаграм: діаграма послідовності підтвердила коректне застосування патерну Mediator при додаванні транзакції, діаграма діяльності зафіксувала алгоритм фільтрації та CSV-експорту, а діаграма станів TransactionsViewModel (детерміновану поведінку інтерфейсу). Реляційну схему бази даних нормалізовано до третьої нормальної форми. Чотири таблиці: accounts, categories, transactions і category_limits, покривають предметну область без надлишкових структур, а індекси на полях account_id і category_id прискорюють найчастіші аналітичні запити.

Третій розділ описує практичну реалізацію та тестування. Дев'ять проєктів рішення розподілено між шарами відповідно до принципу інверсії

залежностей: Domain не має зовнішніх залежностей, Application оголошує інтерфейси, Infrastructure надає конкретні реалізації. Десять обробників MediatR охоплюють усі функціональні сценарії. Найвантажнішим є GetDashboardDataHandler. Він виконує один SQL-запит і агрегує результати в пам'яті, уникаючи проблеми N+1. Три EF-міграції послідовно формують схему бази даних із 17 стандартними категоріями. Застосунок має чотири екрани. Dashboard агрегує KPI поточного місяця та графіки на одному екрані; Transactions охоплює повний CRUD-цикл із комбінованою фільтрацією, пагінацією та CSV-експортом; Analytics виявляє перевитрачені, недовикористані й ризикові категорії та відображає шестимісячний тренд; Budgets керує лімітами й візуалізує прогрес через триколірну шкалу. Автоматизоване тестування охопило 26 тест-кейсів у чотирьох модулях – Analytics (9), Transactions (9), Budgets (4), Dashboard (4). Усі тести проходять у JetBrains Rider, а покриття включає граничні умови: неіснуючі ідентифікатори, порожні вибірки, транзакції поза обраним місяцем і відсутні точки у шестимісячних графіках.

Spendly зберігає фінансові дані локально у SQLite-файлі без залежності від мережі чи хмарного сервісу. Для користувача це означає: жодна персональна фінансова інформація не залишає пристрій. Перевага суттєва, особливо порівняно з хмарними конкурентами.

Подальший розвиток проєкту передбачає кілька напрямків. Алгоритм прогнозування витрат до кінця місяця архітектурно закладено, але у поточній версії не реалізовано. Локальне зберігання потребує надійного механізму резервного копіювання, інакше втрата пристрою означає повну втрату даних. Підтримка імпорту транзакцій із банківських CSV-виписок автоматизує введення даних. Шифрування файлу бази засобами System.Security.Cryptography захистить інформацію на рівні файлової системи. Послідовна реалізація цих кроків наблизить Spendly до рівня комерційних PFM-рішень, зберігши головну перевагу – повний локальний контроль над фінансовими даними.

ПЕРЕЛІК ДЖЕРЕЛ

1. Fitria D., Mustari F. F., Retnowaty A. T., Subagja R. I. Design of Personal Finance Management Application Based on Mobile App (Case Study of Students of STMIK Mardira Indonesia). Informatics Management, Engineering, and Information System Journal. 2025. Vol. 3, No. 1. P. 33–56. DOI: https://doi.org/10.56447/imeisj

2. Stefanov T., Stefanova M., Varbanova S., Temelkov S. Personal Finance Management Application. TEM Journal. 2024. Vol. 13, No. 3. P. 2066–2075. DOI: https://doi.org/10.18421/TEM133-34

3. French D., McKillop D., Stewart E. The effectiveness of smartphone apps in improving financial capability. The European Journal of Finance. 2020. Vol. 26, No. 4–5. P. 302–318. DOI: https://doi.org/10.1080/1351847X.2019.1639526

4. YNAB. Features / YNAB. Official website. URL: https://www.ynab.com

5. BudgetBakers. Features – Powerful Financial Tools / BudgetBakers. Official website. URL: https://budgetbakers.com/en/ (дата звернення: 25.04.2026).

6. Money Manager Ex. Free, easy-to-use, personal finance software / Money Manager Ex. Official project website. URL: https://moneymanagerex.org

7. Quicken. Plans & Pricing for Quicken Simplifi / Quicken Inc. Official website. URL: https://www.quicken.com

8. UML Use Case Diagram Tutorial / Lucidchart. URL: https://www.lucidchart.com/pages/tutorial/uml-use-case-diagram

9. Use Case Diagrams / uml-diagrams.org. URL: https://www.uml-diagrams.org/use-case-diagrams.html

10. UML Project Beginner: Personal Finance App / Visual Paradigm Skills. URL:https://skills.visual-paradigm.com/docs/uml-basics-diagrams-for-beginners/uml-projects-beginners/uml-project-beginner-personal-finance-app/

11. Гобов Д. А., Шевченко Н. Ю. Визначення архітектури вимог до IT-рішення як бізнес-аналітичного продукту. Сучасний стан наукових досліджень та технологій в промисловості. 2024. № 1 (27). С. 26–38. DOI: https://doi.org/10.30837/ITSSI.2024.27.026

12. Омельченко Д. В. Управління нефункціональними вимогами до програмних продуктів у сфері фінансових технологій. Економіка та суспільство. 2025. Вип. 73. DOI: https://doi.org/10.32782/2524-0072/2025-73-57

13. Waliszewski K., Warchlewska A. How we can benefit from personal finance management applications during the COVID-19 pandemic? The Polish case. Entrepreneurship and Sustainability Issues. 2021. Vol. 8(4). P. 103–121. URL: https://ideas.repec.org/a/ssi/jouesi/v8y2021i4p103-121.html

14. Good for your wallet but not for your privacy: 60% of 20 popular budgeting apps share your data / Incogni. 2026. URL: https://blog.incogni.com/budgeting-apps-research/

15. Encrypting data – .NET / Microsoft Learn. URL: https://learn.microsoft.com/dotnet/standard/security/encrypting-data (дата звернення: квітень 2026).

16. Su Y. et al. Software Architecture Patterns for Modern Systems. arXiv. 2025. arXiv:2507.14554v2. URL: https://arxiv.org/html/2507.14554v2

17. Fuksa J., Speth S., Becker C. Evaluation of Software Architecture Patterns. arXiv. 2025. arXiv:2504.18191v1. URL: https://arxiv.org/html/2504.18191v1

18. Repository Pattern and Domain-Driven Design / Microsoft Learn. URL: https://learn.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

19. Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. 2nd ed. Boston: Addison-Wesley, 2004. 721 p.

20. Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd ed. Upper Saddle River: Prentice Hall, 2004. 703 p.

21. Object Management Group. OMG Unified Modeling Language Specification, Version 2.5.1. OMG Document № formal/2017-12-05. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF

22. Churcher C. Beginning Database Design: From Novice to Professional. 2nd ed. New York: Apress, 2012. 266 p.

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

24. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 395 p.

25. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002. 533 p.

26. Richardson C. Microservices Patterns: With examples in Java. Shelter Island: Manning Publications, 2018. 520 p.

27. Riok.Mapperly: Source generator based object mapper for .NET / GitHub. URL: https://github.com/riok/mapperly

28. Johnson J. Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines. 3rd ed. Burlington: Morgan Kaufmann, 2020. 314 p.

29. Few S. Information Dashboard Design: Displaying Data for At-a-Glance Monitoring. 3rd ed. Burlingame: Analytics Press, 2021. 260 p.

30. Osherove R. The Art of Unit Testing: With examples in C#. 2nd ed. Shelter Island: Manning Publications, 2013. 296 p.

31. Freeman S., Pryce N. Growing Object-Oriented Software, Guided by Tests. Boston: Addison-Wesley, 2009. 384 p.

32. Желібо Є.П. Безпека життєдіяльності : підручник / В. В. Зацарний. Київ : Каравела, 2023. 344 с.

33. ДБН В.2.5-28:2018 «Природне і штучне освітлення». – К.: Мінрегіон України, 2018. – 133 с.

34. Жидецький В.Ц. Охорона праці користувачів комп’ютерів : підручник. Львів : Афіша, 2020. 176 с.

35. ДСанПін 3.3.2.007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами (ВДТ) електронно-обчислювальних машин».

ДОДАТКИ

ДОДАТОК А

Тези доповіді на конференції

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет імені Івана Франка
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет
Чернівецький національний університет імені Юрія Федьковича
Вроцлавський економічний університет (Польща)
Університет технологій та економіки імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

IX МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

24-25 квітня 2026 р.

(збірник тез конференції)

Тернопіль 2026

УДК 004.41
Шульга А.
Тернопільський національний технічний університет імені Івана Пулюя

ВИКОРИСТАННЯ ПАТЕРНУ MEDIATOR (MEDIATR) У WPF‑ЗАСТОСУНКАХ ДЛЯ ОБЛІКУ ТА ПЛАНУВАННЯ ФІНАНСІВ

Науковий керівник: д.ф.-м.н., професор Петрик Михайло Романович

Shulha A.
Ternopil Ivan Puluj National Technical Univercity

USING THE MEDIATOR PATTERN (MEDIATR) IN WPF APPLICATIONS FOR FINANCE TRACKING AND PLANNING

Функції персонального фінансового обліку зазвичай швидко ростуть від простого введення витрат до аналітики (підсумки за період, розподіл за категоріями, відстеження лімітів), що збільшує кількість взаємодій між UI, правилами обробки даних і сховищем. Якщо ці взаємодії залишити «розсипаними» по ViewModel, проєкт стає складнішим у супроводі: кожне розширення функціональності створює нові залежності й точки можливої помилки.

У WPF‑архітектурі на базі MVVM ViewModel часто виступає координатором дій користувача. Ризик полягає в тому, що ViewModel починає накопичувати не лише стан екрана, а й «важкі» обчислення, правила валідації та безпосередню роботу з БД. Наслідки типові: по‑перше, складніше змінювати логіку, не зачіпаючи UI; по‑друге, важче тестувати бізнес‑сценарії без запуску інтерфейсу; по‑третє, зростає кількість прихованих залежностей між екранами (наприклад, коли різні ViewModel напряму викликають одні й ті самі сервіси даних).

Патерн Mediator («Посередник») описує підхід, за якого взаємодії між об'єктами переносяться в один компонент‑посередник, щоб зменшити хаотичні залежності між класами [2]. У .NET це зручно реалізувати через MediatR, який підтримує сценарій request/response (команди й запити), а також забезпечує пошук і виклик відповідного handler‑а за типом запиту [1].

У WPF‑застосунку архітектурний «ланцюжок» виглядає так: ViewModel → IMediator.Send(...) → IRequest<T> → IRequestHandler<TRequest,TResponse> [1]. Практична інтерпретація для задач обліку фінансів включає команди (зміна даних): AddExpenseCommand, DeleteExpenseCommand, UpdateBudgetLimitCommand — повертають, наприклад, Unit або ідентифікатор створеного запису та запити (читання/аналітика): GetMonthlyTotalsQuery, GetCategoryBreakdownQuery — повертають агреговані значення або DTO для графіків і звітів.

MediatR інтегрується через стандартний контейнер залежностей .NET: сервіси реєструються у IServiceCollection, а потім резолвляться через IServiceProvider. Microsoft підкреслює, що DI в .NET базується саме на реєстрації сервісів у колекції та подальшому отриманні їх через контейнер, що напряму підтримує ін'єкцію залежностей у конструктор [3]. На практиці це означає: ViewModel отримує IMediator, handler отримує FinanceDbContext та інші сервіси, а композиція проєкту залишається централізованою й прозорою.

Представлення сценаріїв обліку фінансів як окремих request/handler дає відчутний «організаційний» ефект: функціональність структурується за діями, а не за

екранами, і кожен сценарій має чітку точку входу та відповідальність. Патерн «Посередник» зменшує прямі залежності, а MediatR у .NET надає для цього готові контракти й механізм диспетчеризації запитів [2]. Для даних, що зберігаються в SQLite через EF Core, це додатково означає, що бізнес‑логіка читається/тестується в одному місці (handler), а UI не «заражається» деталями збереження [4].

Література:

1. MediatR (офіційний репозиторій). GitHub. URL: https://Https://github.com/LuckyPennySoftware/MediatR.
2. Посередник (Mediator). Refactoring Guru. URL: https://refactoring.guru/uk/design-patterns/mediator.
3. Dependency injection in .NET. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection/overview.
4. .NET dependency injection. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection/overview.