МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ

ВІДДІЛЕННЯ
«ІНЖЕНЕРНА ІНФРАСТРУКТУРА ТА КОМП’ЮТЕРНІ НАУКИ»

ЦИКЛОВА КОМІСІЯ СПЕЦІАЛЬНОСТІ
«КОМП'ЮТЕРНІ НАУКИ»

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня фаховий молодший бакалавр
за спеціальністю 122 Комп'ютерні науки
на тему:

«Персональний фінансовий асистент для ведення сімейного бюджету»

Виконала здобувачка освіти групи П-41
Цимбалюк Дарина Юріївна

Керівник роботи:
Устименко Ярослав Іванович

Рецензент:

Житомир – 2026

Зміст

РЕФЕРАТ

Кваліфікаційна робота містить: 80 сторінок, 34 рисунки, 20 таблиць, 2 додатки та 27 літературних джерел.

Об'єкт дослідження — процес автоматизації обліку та планування особистих фінансів домогосподарства.

Предмет дослідження — методи та засоби розробки програмного забезпечення для управління сімейним бюджетом на платформі .NET 9 із застосуванням архітектурного патерну MVVM, бази даних SQLite та фреймворку Entity Framework Core.

Мета роботи — розробити десктопний застосунок «Персональний фінансовий асистент», який забезпечує зручний україномовний інтерфейс, облік транзакцій, контроль лімітів та формування аналітичних звітів.

У роботі проаналізовано концепції фінансового менеджменту та існуючі програмні рішення, що виявило відсутність на ринку безкоштовних офлайн-інструментів із повною українською локалізацією. Обґрунтовано вибір технологічного стеку: C# 13, WPF для побудови інтерфейсу та Material Design для забезпечення сучасного дизайну.

Спроектовано архітектуру застосунку на основі шарового підходу та патернів Repository і Unit of Work.

Результатом роботи є повнофункціональний програмний продукт, що дозволяє:

- здійснювати облік доходів і витрат за 18 категоріями;
- вести спільний бюджет для кількох членів родини;
- моніторити виконання місячних лімітів із кольоровою індикацією статусів;
- формувати звіти у форматах PDF та CSV;
- виконувати резервне копіювання даних.

Ключові слова: .NET 9, WPF, MVVM, SQLITE, ENTITY FRAMEWORK CORE, СІМЕЙНИЙ БЮДЖЕТ, ФІНАНСОВИЙ ОБЛІК, АНАЛІТИЧНА ЗВІТНІСТЬ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) — набір визначень і протоколів для побудови та інтеграції прикладного програмного забезпечення.

CRUD (Create, Read, Update, Delete) — чотири базові функції управління даними: створення, читання, оновлення та видалення.

CSV (Comma-Separated Values) — текстовий формат, призначений для представлення табличних даних, де значення розділені спеціальним символом.

DI (Dependency Injection) — впровадження залежностей; патерн проектування, що забезпечує слабку зв'язність компонентів системи.

EF Core (Entity Framework Core) — об'єктно-реляційний мапінг (ORM) для платформи .NET, що дозволяє працювати з базами даних за допомогою об'єктів .NET.

IDE (Integrated Development Environment) — інтегроване середовище розробки програмного забезпечення.

KPI (Key Performance Indicators) — ключові показники ефективності, що використовуються для візуалізації фінансового стану на дашборді.

LINQ (Language Integrated Query) — мова запитів до даних різних джерел, інтегрована в мову C#.

MVVM (Model-View-ViewModel) — архітектурний патерн, що відокремлює бізнес-логіку від інтерфейсу користувача.

NuGet — система управління пакетами для платформи .NET.

ORM (Object-Relational Mapping) — технологія програмування, яка пов'язує бази даних з концепціями об'єктно-орієнтованих мов програмування.

PDF (Portable Document Format) — міжплатформний формат електронних документів для представлення звітів.

SOLID — акронім п'яти основних принципів об'єктно-орієнтованого програмування та дизайну (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion).

SQL (Structured Query Language) — мова структурованих запитів, що використовується для управління даними в реляційних СУБД.

ВСТУП

Фінансова стабільність родини є одним із ключових чинників її добробуту та якості життя. Раціональне управління сімейним бюджетом — ведення обліку доходів і витрат, планування видатків, контроль за дотриманням фінансових лімітів — дозволяє уникнути боргового навантаження, накопичувати заощадження та досягати довгострокових фінансових цілей. Проте, за даними досліджень фінансової грамотності населення України, більшість домогосподарств досі не застосовує жодних систематичних інструментів обліку власних фінансів, а планування бюджету здійснюється інтуїтивно або зовсім відсутнє.

Сучасний ринок програмного забезпечення пропонує широкий спектр застосунків для управління особистими фінансами — від веб-сервісів до мобільних додатків. Однак переважна більшість з них орієнтована на англомовних користувачів, не підтримує локалізовані категорії витрат і доходів українською мовою, вимагає постійного підключення до інтернету або хмарної синхронізації, що породжує ризики витоку персональних даних. Десктопні рішення, здатні функціонувати повністю в офлайн-режимі, зберігати дані локально та мати повноцінний україномовний інтерфейс, фактично відсутні на ринку у вільному доступі.

Таким чином, розробка персонального фінансового асистента для ведення сімейного бюджету з україномовним інтерфейсом, підтримкою кількох членів родини, гнучкою системою категорій, інструментами планування та аналітики є актуальним завданням, що має практичне значення для широкого кола користувачів.

Мета роботи — розробити десктопний застосунок персонального фінансового асистента для ведення сімейного бюджету на платформі .NET 9 з використанням технологій WPF та SQLite, який забезпечує зручний україномовний інтерфейс, облік транзакцій, контроль бюджетних лімітів та формування аналітичних звітів.

Об'єкт дослідження — процес автоматизації обліку та планування особистих фінансів домогосподарства.

Предмет дослідження — методи та засоби розробки програмного забезпечення для управління сімейним бюджетом на платформі .NET 9 із застосуванням архітектурного патерну MVVM, бази даних SQLite та фреймворку Entity Framework Core.

Для досягнення поставленої мети визначено такі **завдання роботи**:

1. Проаналізувати поняття сімейного бюджету, методи його ведення та потреби користувачів у програмних засобах фінансового обліку.
2. Дослідити існуючі програмні рішення для управління особистими фінансами та обґрунтувати необхідність розробки власного застосунку.
3. Вивчити технологічний стек (.NET 9, WPF, SQLite, Entity Framework Core, Material Design) та архітектурні патерни (MVVM, Repository, Unit of Work), що застосовуватимуться при розробці.
4. Визначити функціональні та нефункціональні вимоги до програмного забезпечення.
5. Розробити структуру бази даних і архітектуру застосунку відповідно до принципів SOLID та шаблону MVVM.
6. Реалізувати повнофункціональний застосунок з підтримкою обліку транзакцій, управління категоріями та членами родини, моніторингу бюджетних лімітів, формування звітів і експорту даних у форматах PDF та CSV.
7. Провести тестування розробленого програмного забезпечення (модульне, інтеграційне, функціональне) та проаналізувати показники його продуктивності.

Методи дослідження. У роботі застосовано такі методи: аналіз і синтез — для дослідження існуючих програмних рішень та формування вимог; об'єктно-орієнтоване проектування — для побудови архітектури застосунку; метод прототипування — для проектування інтерфейсу користувача;

тестування програмного забезпечення — для перевірки коректності реалізованого функціоналу.

Практичне значення роботи полягає у створенні повністю функціонального десктопного застосунку, що може використовуватися українськими родинами для систематичного ведення сімейного бюджету в офлайн-режимі. Застосунок реалізує облік доходів і витрат за 18 категоріями, встановлення та моніторинг місячних лімітів витрат з кольоровою індикацією, аналітичний дашборд із діаграмами, а також експорт звітів у форматах PDF та CSV. Програмна реалізація з відкритим кодом може слугувати навчальним прикладом застосування сучасного стеку технологій .NET у розробці WPF-застосунків корпоративного рівня.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 75 аркушів, з них 36 рисунків та 20 таблиць.

У першому розділі розглядаються теоретичні основи персонального фінансового менеджменту, аналізуються існуючі програмні рішення, досліджується технологічний стек та архітектурні патерни, що застосовуються у розробці.

У другому розділі визначаються вимоги до програмного забезпечення, проектується структура бази даних, архітектура застосунку та інтерфейс користувача, описуються основні алгоритми обробки даних.

У третьому розділі описується програмна реалізація усіх компонентів застосунку — шару даних, бізнес-логіки, ViewModel-шару та інтерфейсу користувача, наводяться результати тестування та аналіз продуктивності.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ПЕРСОНАЛЬНОГО ФІНАНСОВОГО АСИСТЕНТА

1.1 Концепція персонального фінансового менеджменту

Персональний фінансовий менеджмент — це сукупність методів, принципів і практик, спрямованих на ефективне планування, облік, контроль і аналіз грошових потоків окремої особи або домогосподарства. На відміну від корпоративних фінансів, персональний фінансовий менеджмент оперує порівняно невеликими обсягами коштів, однак має ключове значення для забезпечення матеріального добробуту родини, формування заощаджень та досягнення довгострокових фінансових цілей.

Сімейний бюджет — це фінансовий план домогосподарства на певний період (зазвичай місяць або рік), що відображає сукупність очікуваних доходів і запланованих витрат. Сімейний бюджет виконує кілька ключових функцій: планувальну (встановлення цільових показників витрат і заощаджень), контрольну (моніторинг відхилень від плану), аналітичну (виявлення структури витрат і резервів для оптимізації) та мотиваційну (формування фінансової дисципліни).

З точки зору структури, доходи домогосподарства поділяються на активні (заробітна плата, доходи від підприємницької діяльності, фріланс) та пасивні (відсотки за депозитами, орендна плата, дивіденди, інвестиційний дохід). Витрати традиційно класифікують за кількома ознаками:

- за регулярністю: постійні (комунальні послуги, оренда, кредитні платежі) та змінні (продукти харчування, розваги, одяг);
- за необхідністю: обов'язкові (харчування, ліки, транспорт) та необов'язкові (розваги, подорожі, предмети розкоші);
- за категорією: продукти харчування, транспорт, комунальні послуги, охорона здоров'я, освіта, розваги, одяг, заощадження тощо.

Для систематизації роботи з сімейним бюджетом фінансові консультанти та науковці пропонують різні методи його ведення.

Найпоширенішими серед них є метод 50/30/20, нульовий бюджет, конвертний метод та метод оберненого бюджету.

Метод 50/30/20 передбачає розподіл чистого доходу родини на три частини: 50% — на обов'язкові потреби (житло, їжа, транспорт, комунальні послуги), 30% — на особисті бажання (розваги, ресторани, хобі), 20% — на заощадження та погашення боргів. Метод простий у застосуванні і не вимагає детального обліку кожної транзакції, однак не враховує індивідуальних обставин родини.

Нульовий бюджет (Zero-Based Budgeting) ґрунтується на принципі, що різниця між доходами і запланованими витратами має дорівнювати нулю — кожна гривня доходу заздалегідь «призначається» на конкретну статтю витрат або заощаджень. Цей метод забезпечує максимальний контроль над грошовими потоками, але вимагає значних часових витрат на планування.

Конвертний метод (Envelope Budgeting) передбачає фізичний або цифровий розподіл коштів між «конвертами» — відокремленими «кишенями» для кожної категорії витрат. Після вичерпання коштів у конверті витрати за цією категорією припиняються до наступного бюджетного періоду. Метод особливо ефективний для контролю імпульсивних витрат.

Метод оберненого бюджету (Pay Yourself First) полягає в тому, що одразу після отримання доходу визначена сума переводиться на заощадження, а решта витрачається на власний розсуд. Метод максимально простий у реалізації, проте не гарантує контролю над структурою витрат.

Порівняльний аналіз зазначених методів наведено в таблиці 1.1.

Таблиця 1.1 — Порівняння методів ведення сімейного бюджету

Метод	Складність ведення	Рівень контролю	Гнучкість	Підходить для
50/30/20	Низька	Середній	Висока	Початківців у бюджетуванні
Нульовий бюджет	Висока	Максимальний	Низька	Досвідчених користувачів з чіткими фінансовими цілями
Конвертний метод	Середня	Високий	Середня	Родин з тенденцією до надмірних витрат

Обернений бюджет	Низька	Низький	Висока	Тих, хто хоче автоматизувати заощадження

Джерело: складено автором

Розроблюваний у цій роботі застосунок підтримує гібридний підхід: він не нав'язує конкретного методу, а надає інструменти для обліку всіх транзакцій та встановлення місячних лімітів за категоріями, що відповідає принципам конвертного та нульового бюджету.

Типова структура сімейного бюджету у відсотковому розподілі представлена на рисунку 1.1.

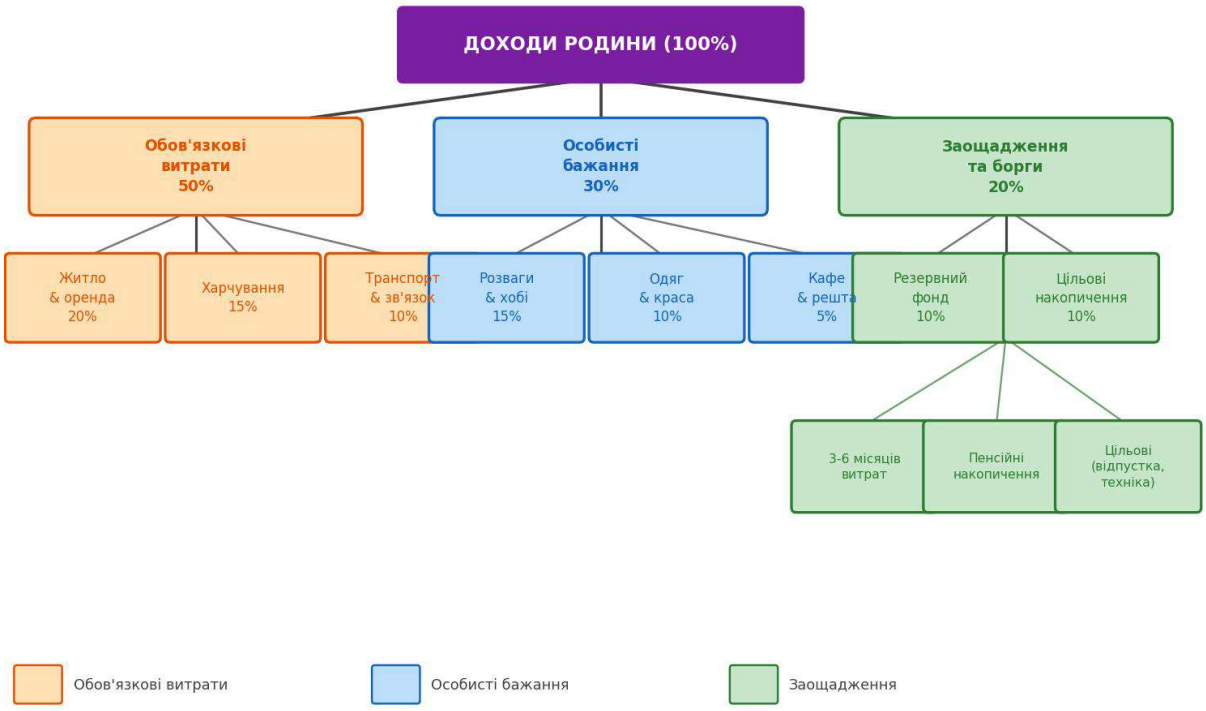


Рисунок 1.1 — Структура типового сімейного бюджету за методом 50/30/20

Автоматизація ведення сімейного бюджету за допомогою програмного забезпечення суттєво спрощує процес обліку, знижує ймовірність арифметичних помилок та дозволяє в режимі реального часу отримувати аналітику про стан фінансів. Ключовими функціями, яких потребують користувачі від подібного ПЗ, є: зручне додавання транзакцій, автоматична категоризація витрат, встановлення бюджетних лімітів, наочна візуалізація даних у вигляді діаграм та можливість формування звітів.

1.2 Аналіз існуючих програмних рішень

На сучасному ринку програмного забезпечення для управління особистими фінансами представлено широкий спектр рішень різних типів: веб-застосунки, мобільні додатки та десктопні програми. Розглянемо найбільш відомі з них.

YNAB (You Need A Budget) — веб-сервіс та мобільний застосунок, що реалізує метод нульового бюджету. Відзначається потужним функціоналом: підтримкою банківських імпортів, реальночасовою синхронізацією між пристроями, детальною аналітикою та великою спільнотою користувачів. Проте YNAB є платним (близько 15 USD на місяць або 99 USD на рік), не підтримує українську мову та вимагає постійного інтернет-з'єднання. Зберігання фінансових даних на серверах компанії може бути небажаним для частини користувачів.

Mint — безкоштовний веб- та мобільний застосунок від компанії Intuit (США). Автоматично синхронізується з банківськими рахунками, класифікує транзакції та надає аналітику. Недоліки: доступний лише для резидентів США та Канади, підтримує лише англійську мову, не має офлайн-режиму, у 2023 році було закрито веб-версію.

GnuCash — безкоштовна відкрита десктопна програма з підтримкою методу подвійного запису, що традиційно використовується в бухгалтерському обліку. Підходить для просунутих користувачів і малого бізнесу, однак має застарілий інтерфейс та надто складна для пересічних домашніх користувачів.

Money Manager Ex (MMEX) — безкоштовна відкрита десктопна програма з підтримкою кількох валют, звітами та локалізацією на багато мов. Підтримує українську мову, однак має обмежені можливості аналітики, відсутній хмарний бекап та застарілий UI.

Toshl Finance — мобільний і веб-застосунок з приємним інтерфейсом, підтримкою тегів, бюджетів та звітів. Базовий функціонал безкоштовний, проте розширені можливості (планування бюджету, банківський імпорт)

доступні лише за підпискою. Відсутня повноцінна локалізація українською мовою.

Spendee — мобільний застосунок з акцентом на візуалізацію та спільний сімейний бюджет. Має приємний дизайн та підтримку кількох валют. Потребує підписки для розширеного функціоналу, відсутня офлайн-версія та локалізація українською.

Порівняльний аналіз розглянутих рішень наведено в таблиці 1.2.

Таблиця 1.2 — Порівняльний аналіз існуючих програмних рішень для ведення бюджету

Критерій	YNAB	Mint	GnuCash	MMEX	Toshl	Spendee	мій
Українська мова	✗	✗	Частково	✓	✗	✗	✓
Офлайн-режим	✗	✗	✓	✓	✗	✗	✓
Десктопна версія	✓	✗	✓	✓	✗	✗	✓
Локальне зберігання даних	✗	✗	✓	✓	✗	✗	✓
Безкоштовний	✗	✓*	✓	✓	Частково	Частково	✓
Члени родини	✓	✗	✗	✗	✓	✓	✓
Бюджетні ліміти	✓	✓	✗	✓	✓	✓	✓
Діаграми та аналітика	✓	✓	Частково	✓	✓	✓	✓
Експорт PDF/CSV	✓	✓	✓	✓	✗	✗	✓
Сучасний інтерфейс (Material Design)	✓	✓	✗	✗	✓	✓	✓

** Mint закрито у 2023 році*

Як видно з таблиці 1.2, жодне з розглянутих рішень не задовольняє одночасно всім ключовим вимогам українського користувача: україномовний інтерфейс, офлайн-режим з локальним збереженням даних, сучасний дизайн, підтримка кількох членів родини та можливість бюджетного планування — і при цьому є безкоштовним. Саме це обґрунтовує доцільність розробки власного рішення.

На рисунку 1.2 наведена порівняльна діаграма функціоналу існуючих рішень за ключовими критеріями.

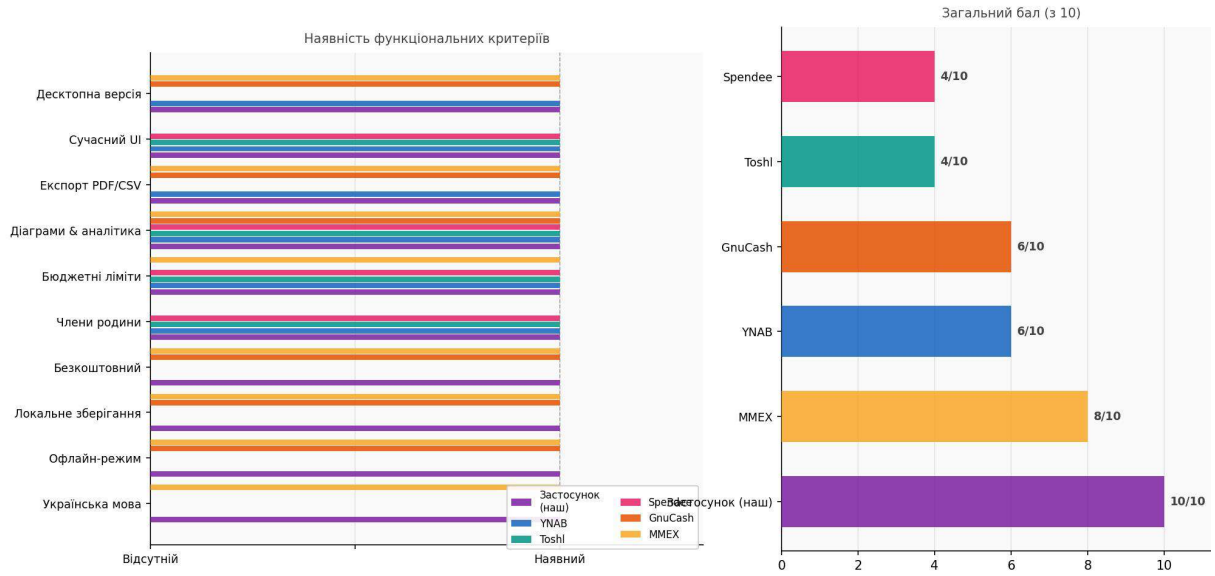


Рисунок 1.2 — Порівняння функціоналу програмних рішень за 10 критеріями

Примітка: оцінювання виконано автором за критеріями таблиці 1.2

Таким чином, аналіз конкурентного середовища підтверджує актуальність і доцільність розробки персонального фінансового асистента з україномовним інтерфейсом, автономним офлайн-режимом та сучасним дизайном, що охоплює повний набір функцій сімейного бюджетування.

1.3 Огляд технологій та інструментів розробки

1.3.1 Платформа .NET 9 та мова програмування C# 13

.NET 9 — остання версія кросплатформенного середовища виконання від компанії Microsoft, випущена у листопаді 2024 року. Платформа забезпечує виконання застосунків на операційних системах Windows, Linux та macOS, підтримує широкий спектр типів додатків: веб-застосунки (ASP.NET Core), мобільні та десктопні програми (.NET MAUI, WPF, WinForms), хмарні мікросервіси та консольні утиліти.

Ключові переваги .NET 9 для розробки десктопних застосунків:

- **Висока продуктивність:** .NET 9 демонструє суттєве покращення швидкодії порівняно з попередніми версіями завдяки оптимізаціям JIT-компілятора та збирача сміття (GC). Час запуску застосунків скорочено, а споживання пам'яті зменшено.

- **Актуальна екосистема:** доступ до найновіших NuGet-пакетів, активна підтримка від Microsoft та спільноти розробників.
- **Строга типізація та безпека:** система типів C# запобігає цілому класу помилок на етапі компіляції.
- **Підтримка асинхронного програмування:** вбудована підтримка async/await дозволяє будувати responsive UI без блокування головного потоку.

C# 13 — актуальна версія мови програмування, що використовується у проєкті. Серед нових можливостей, задіяних у розробці: первинні конструктори (primary constructors) для скорочення шаблонного коду класів, вирази колекцій (collection expressions) для лаконічної ініціалізації колекцій, патерн-матчинг для обробки різних типів транзакцій.

1.3.2 Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) — фреймворк для розробки графічних інтерфейсів десктопних застосунків під Windows, що входить до складу .NET. WPF використовує декларативну мову розмітки **XAML** (eXtensible Application Markup Language) для опису UI та підтримує апаратне прискорення рендерингу через DirectX.

Ключові концепції WPF, що застосовуються у проєкті:

- **XAML:** декларативний опис елементів інтерфейсу, стилів, анімацій і шаблонів даних. Чітке розділення розмітки (XAML) і логіки (C#) відповідає принципу розділення відповідальностей.
- **Прив'язка даних (Data Binding):** механізм автоматичної синхронізації між UI-елементами та властивостями ViewModel. Підтримує режими OneWay, TwoWay, OneWayToSource та OneTime.
- **Dependency Properties:** розширена система властивостей, що підтримує прив'язку даних, анімацію та стилізацію.
- **Command System:** реалізація патерну Command через інтерфейс ICommand, що відокремлює дії користувача від логіки їх обробки.

- **Value Converters:** конвертери IValueConverter для трансформації даних між ViewModel та View (наприклад, числа в рядок з форматуванням валюти, статусу бюджету в колір).

Порівняно з альтернативами, WPF має такі переваги: зрілість та стабільність (понад 15 років у виробництві), найбільша екосистема бібліотек для Windows-десктопів (включно з Material Design, LiveCharts), відмінна підтримка в Visual Studio та потужна система шаблонів даних (DataTemplate).

1.3.3 SQLite та Entity Framework Core 9

SQLite — легковагова вбудована реляційна СУБД з відкритим кодом. На відміну від клієнт-серверних СУБД (MySQL, PostgreSQL, MS SQL Server), SQLite зберігає всю базу даних в одному файлі на диску і не потребує окремого серверного процесу. Це робить її ідеальним вибором для локальних десктопних застосунків.

Переваги SQLite для даного проєкту:

- **Нульове адміністрування:** не потребує встановлення та налаштування сервера БД.
- **Автономність:** база даних — один файл, що спрощує резервне копіювання та перенесення.
- **Надійність:** підтримує ACID-транзакції, що гарантує цілісність фінансових даних.
- **Висока продуктивність:** для обсягів даних домашнього бюджету (тисячі транзакцій) SQLite демонструє час запиту в межах одиниць мілісекунд.

Entity Framework Core 9 (EF Core 9) — офіційний ORM-фреймворк від Microsoft для .NET, що забезпечує об'єктно-реляційне відображення між C#-класами та таблицями бази даних. У проєкті застосовується підхід **Code First**: спочатку описуються C#-класи моделей (Transaction, Category, FamilyMember, BudgetLimit), а EF Core автоматично генерує схему бази даних через систему міграцій.

Ключові можливості EF Core 9, що використовуються:

- **Міграції:** автоматичне оновлення схеми БД при зміні моделей без втрати даних.
- **LINQ-запити:** типобезпечні запити до БД на мові C# замість рядкового SQL.
- **Асинхронні операції:** ToListAsync(), FirstOrDefaultAsync(), SaveChangesAsync() — всі операції з БД не блокують UI-потік.
- **Fluent API:** гнучке налаштування відображення моделей, зовнішніх ключів, індексів та каскадної поведінки.
- **Seed Data:** метод HasData() для автоматичного заповнення початковими даними при першому запуску.

1.3.4 Бібліотеки та NuGet-пакети

Розроблюваний застосунок використовує низку сторонніх NuGet-пакетів, що суттєво прискорюють розробку та підвищують якість кінцевого продукту. Повний перелік залежностей наведено в таблиці 1.3.

Таблиця 1.3 — Перелік використаних NuGet-пакетів

NuGet-пакет	Версія	Призначення
MaterialDesignThemes	5.1.0	Компоненти Material Design для WPF (кнопки, картки, поля вводу, іконки)
MaterialDesignColors	3.1.0	Кольорові палітри Material Design 3
LiveChartsCore.SkiaSharpView.WPF	2.0.0-rc2	Інтерактивні діаграми (кругові, стовпчасті, лінійні)
Microsoft.EntityFrameworkCore	9.0.0	ORM-фреймворк для роботи з базою даних
Microsoft.EntityFrameworkCore.Sqlite	9.0.0	Провайдер SQLite для EF Core
Microsoft.EntityFrameworkCore.Tools	9.0.0	Інструменти CLI для управління міграціями
CommunityToolkit.Mvvm	8.4.0	Базові класи MVVM (ObservableObject, RelayCommand, AsyncRelayCommand)
QuestPDF	2024.10.4	Генерація PDF-документів зі складним форматуванням
CsvHelper	33.0.1	Читання та запис CSV-файлів
Microsoft.Extensions.DependencyInjection	9.0.0	IoC-контейнер для впровадження залежностей
Microsoft.Extensions.Hosting	9.0.0	Хост для управління життєвим циклом застосунку та DI

MaterialDesignThemes — найпопулярніша бібліотека Material Design для WPF. Надає понад 600 іконок Material Design, компоненти (Card, Chip,

FloatingActionButton, Snackbar, DatePicker, DataGrid), систему тем (світла/темна) та підтримку кольорових схем. У проєкті застосовується пурпурна тема з основним кольором #7B1FA2.

LiveChartsCore — сучасна бібліотека для побудови інтерактивних діаграм на базі SkiaSharp. Підтримує плавні анімації, адаптивний рендеринг та прив'язку до ViewModel через ObservableCollection. У проєкті використовуються кругова діаграма (витрати за категоріями) та стовпчаста діаграма (порівняння доходів і витрат за 6 місяців).

QuestPDF — бібліотека для генерації PDF-документів з флюентним API. Дозволяє описувати структуру документа (сторінки, колонки, таблиці, заголовки) у вигляді C#-коду, що суттєво зручніше за роботу з сирим PDF або iTextSharp.

CommunityToolkit.Mvvm — офіційна бібліотека Microsoft для реалізації патерну MVVM. Надає базові класи ObservableObject, RelayCommand та AsyncRelayCommand, а також атрибути для генерації коду команд та властивостей на етапі компіляції.

Стек технологій застосунку у вигляді шарової архітектури представлено на рисунку 1.3.

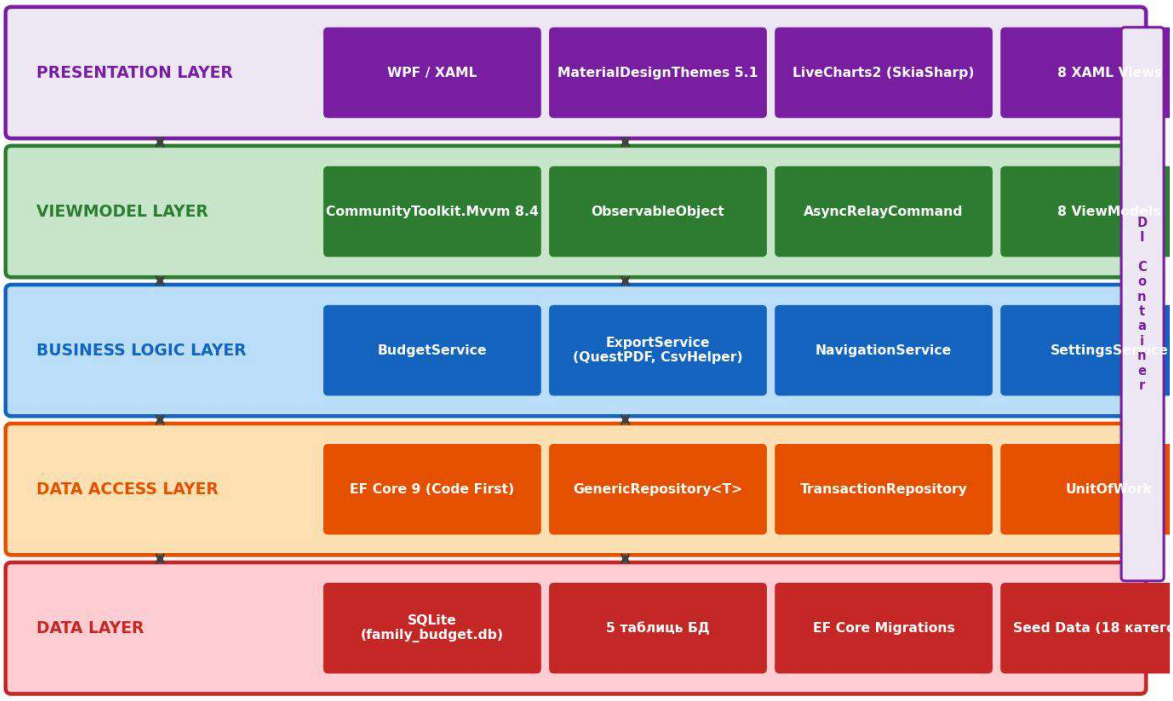


Рисунок 1.3 — Шарова архітектура стеку технологій застосунку

1.4 Архітектурні патерни та принципи проектування

1.4.1 Патерн MVVM (Model–View–ViewModel)

MVVM (Model–View–ViewModel) — архітектурний патерн, розроблений компанією Microsoft спеціально для WPF і Silverlight, що забезпечує чітке розділення між логікою подання даних і бізнес-логікою застосунку. Патерн складається з трьох компонентів:

Model — представляє доменну область застосунку: сутності бази даних (Transaction, Category, FamilyMember, BudgetLimit), сервіси та репозиторії. Model не знає про існування View та ViewModel і не залежить від них. В даному проєкті до шару Model належать C#-класи сутностей, інтерфейси репозиторіїв та сервіси обробки даних.

View — відповідає за відображення даних і взаємодію з користувачем. У WPF View реалізується через XAML-файли (.xaml). View не містить жодної бізнес-логіки і взаємодіє з ViewModel виключно через механізм прив'язки даних (Data Binding) та команди (ICommand). Відсутність коду в code-behind файлах (.xaml.cs) є ознакою правильної реалізації MVVM.

ViewModel — посередник між Model і View. ViewModel отримує дані від Model (через сервіси та репозиторії), перетворює їх у форму, зручну для

відображення, і надає View через прив'язані властивості (ObservableProperty) та команди (RelayCommand). ViewModel реалізує інтерфейс INotifyPropertyChanged, завдяки чому View автоматично оновлюється при зміні даних.

Схема взаємодії компонентів MVVM представлена на рисунку 1.4.

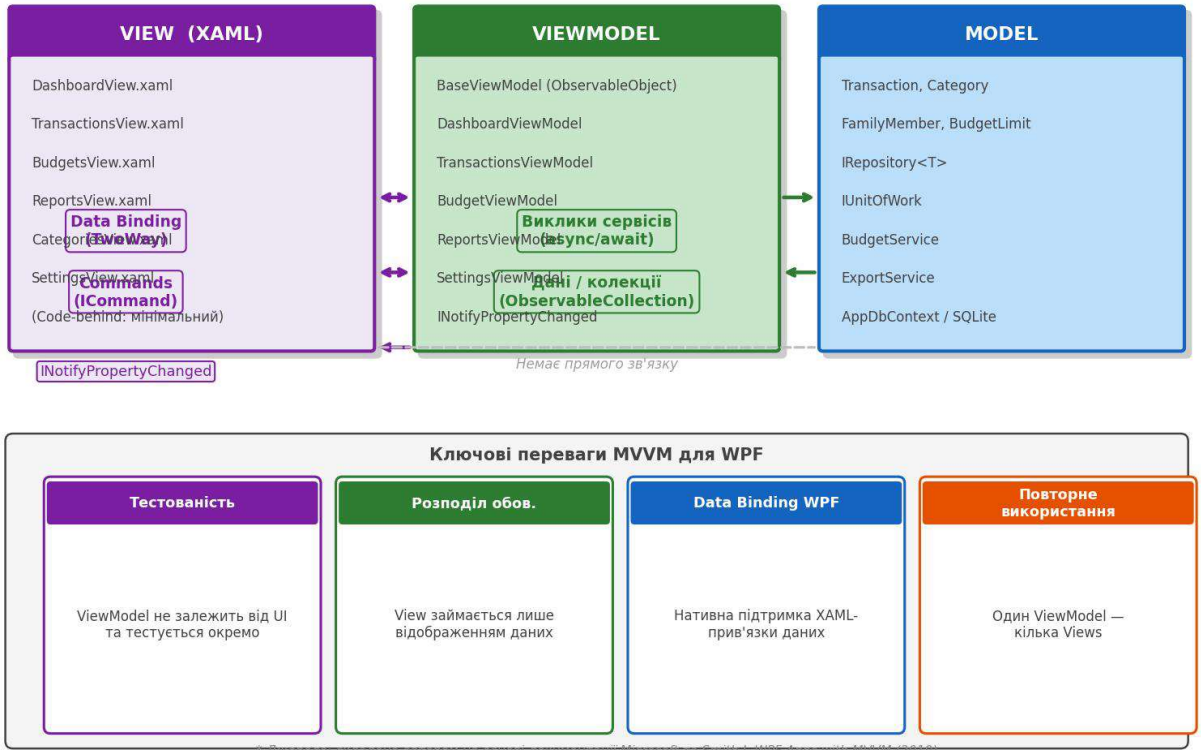


Рисунок 1.4 — Схема взаємодії компонентів за патерном MVVM

Переваги MVVM у контексті WPF-розробки:

- **Тестованість:** ViewModel не залежить від UI і може тестуватися без запуску інтерфейсу.
- **Повторне використання:** один ViewModel може обслуговувати кілька різних Views.
- **Паралельна розробка:** дизайнер може працювати над XAML незалежно від програміста.
- **Підтримуваність:** чітке розділення відповідальностей спрощує локалізацію і виправлення помилок.

1.4.2 Патерн Repository та Unit of Work

Repository — патерн проектування, що абстрагує логіку доступу до даних за інтерфейсом, схожим на колекцію об'єктів. Замість прямих викликів DbContext у ViewModel, код взаємодіє лише з IRepository<T>, що:

- ізолює ViewModel від конкретної реалізації бази даних;
- дозволяє легко замінити SQLite на іншу СУБД або на In-Memory провайдер для тестування;
- централізує логіку запитів, запобігаючи дублюванню LINQ-виразів.

Unit of Work — патерн, що координує роботу кількох репозиторіїв у межах однієї транзакції. IUnitOfWork надає доступ до всіх репозиторіїв і метод SaveChangesAsync(), який зберігає всі зміни в єдиній атомарній операції. Це гарантує консистентність даних: наприклад, при одночасному оновленні транзакції та бюджетного ліміту обидві зміни або зберігаються разом, або відкочуються при помилці.

Схема взаємозв'язку між Repository, Unit of Work та EF Core представлена на рисунку 1.5.

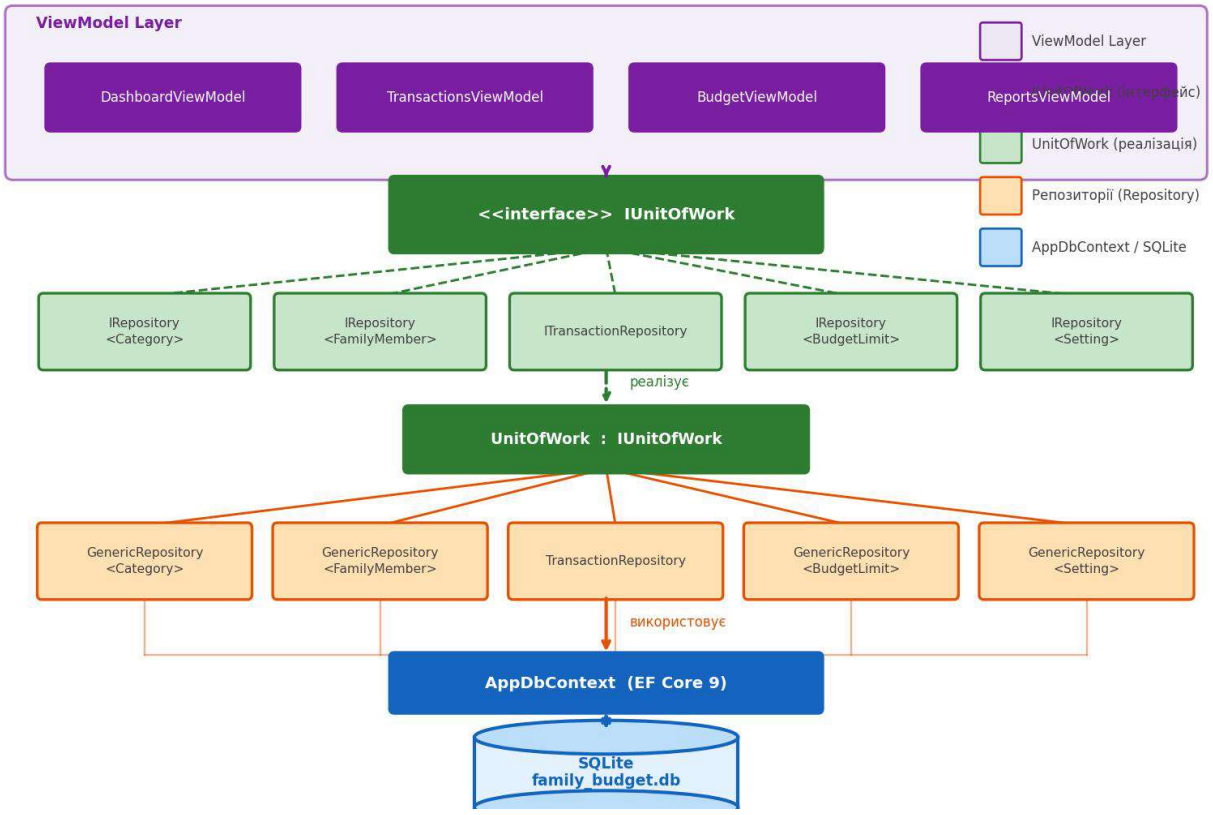


Рисунок 1.5 — Схема взаємодії патернів Repository та Unit of Work

1.4.3 Dependency Injection

Dependency Injection (DI) — принцип програмування, за якого залежності об'єкта не створюються ним самостійно (через new), а передаються ззовні через конструктор або властивості. У поєднанні з принципом **Inversion of Control (IoC)** це дозволяє досягти слабкої зв'язності (low coupling) між компонентами.

У проєкті використовується Microsoft.Extensions.DependencyInjection — офіційний IoC-контейнер .NET. Усі залежності реєструються в App.xaml.cs при запуску застосунку та впроваджуються в конструктори відповідних класів. Застосовуються три основні часи життя:

- **Singleton** — один екземпляр на весь час роботи застосунку (NavigationService, CurrencyProvider).
- **Scoped** — один екземпляр на одну «область» (AppDbContext, репозиторії).
- **Transient** — новий екземпляр при кожному запиті (ViewModel-класи, сервіси звітів).

1.4.4 Принципи SOLID

При розробці застосунку дотримуються принципів SOLID — п'яти фундаментальних принципів об'єктно-орієнтованого проектування:

- **S — Single Responsibility Principle:** кожен клас має єдину відповідальність. TransactionViewModel відповідає лише за відображення та фільтрацію транзакцій, ExportService — лише за експорт, BudgetService — лише за розрахунок лімітів.
- **O — Open/Closed Principle:** класи відкриті для розширення, але закриті для модифікації. Додавання нового типу звіту реалізується через новий клас-сервіс, а не через модифікацію існуючих.
- **L — Liskov Substitution Principle:** підкласи замінюють батьківські класи без порушення поведінки. TransactionRepository розширює GenericRepository<Transaction>, не порушуючи контракту базового класу.

- **I — Interface Segregation Principle:** використовуються вузькоспеціалізовані інтерфейси (ITransactionRepository, IBudgetService, IExportService) замість одного великого.
- **D — Dependency Inversion Principle:** ViewModel залежить від абстракцій (IUnitOfWork, IBudgetService), а не від конкретних реалізацій.

Порівняльний аналіз архітектурних патернів для WPF-застосунків наведено в таблиці 1.4.

Таблиця 1.4 — Порівняння архітектурних патернів для WPF-застосунків

Критерій	MVC	MVP	MVVM
Зв'язок View–логіка	Через Controller	Через Presenter (пряме посилання)	Через Data Binding (слабкий зв'язок)
Тестованість ViewModel/Presenter	Середня	Висока	Висока
Підтримка Data Binding WPF	Обмежена	Відсутня	Нативна
Дублювання коду UI	Часте	Середнє	Мінімальне
Складність початкового налаштування	Низька	Середня	Середня
Підходить для WPF	Ні	Частково	Так (рекомендований)
Паралельна розробка UI та логіки	Складно	Можливо	Легко

Як видно з таблиці 1.4, MVVM є єдиним патерном, що забезпечує нативну підтримку механізму Data Binding платформи WPF, мінімізує зв'язність між View та бізнес-логікою та дозволяє ефективно тестувати ViewModel-класи без запуску інтерфейсу. Саме тому MVVM обрано як основний архітектурний патерн для даного проєкту.

Висновки до розділу 1

У першому розділі розглянуто теоретичні основи розробки персонального фінансового асистента для ведення сімейного бюджету.

Проведений аналіз концепції персонального фінансового менеджменту показав, що ефективне ведення сімейного бюджету потребує комплексного підходу до обліку доходів і витрат, планування місячних лімітів та аналізу фінансового стану родини. Найбільш збалансованим методом для широкого

кола користувачів визнано гібридний підхід, що поєднує елементи конвертного методу та нульового бюджету.

Порівняльний аналіз існуючих програмних рішень (YNAB, Mint, GnuCash, MMEX, Toshl, Spendee) виявив, що жодне з них не задовольняє одночасно таким вимогам: україномовний інтерфейс, повноцінний офлайн-режим, локальне зберігання даних, сучасний Material Design та підтримка кількох членів родини при повній безоплатності. Це підтверджує актуальність і доцільність розробки власного рішення.

Проведений огляд технологічного стеку показав, що поєднання .NET 9, WPF, SQLite, Entity Framework Core 9 та бібліотек MaterialDesignThemes і LiveCharts2 є оптимальним для реалізації десктопного фінансового асистента з сучасним інтерфейсом, надійним локальним сховищем даних та широкими аналітичними можливостями.

Обґрунтовано вибір архітектурних патернів: MVVM як основний патерн презентаційного шару, Repository та Unit of Work для абстракції доступу до даних, Dependency Injection для забезпечення слабкої зв'язності компонентів. Дотримання принципів SOLID забезпечить підтримуваність і розширюваність коду у перспективі.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ ТА ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Аналіз вимог до програмного забезпечення

2.1.1 Функціональні вимоги

Функціональні вимоги описують конкретну поведінку системи — що саме вона повинна робити. Для визначення функціональних вимог до персонального фінансового асистента проведено аналіз потреб типового користувача — члена українського домогосподарства, який веде облік сімейних фінансів. На основі цього аналізу та порівняння з існуючими рішеннями (розд. 1.2) визначено такі функціональні вимоги:

Управління транзакціями є центральним функціоналом застосунку. Система повинна дозволяти додавати, редагувати та видаляти транзакції двох типів — дохід і витрата. Кожна транзакція характеризується сумою, датою, категорією, необов'язковим посиланням на члена родини та текстовим описом. Перегляд транзакцій має здійснюватися через таблицю з підтримкою фільтрації за датою, типом, категорією, членом родини та текстовим пошуком по опису.

Управління категоріями. Система повинна містити 18 попередньо визначених категорій українською мовою, що охоплюють типові статті доходів і витрат українського домогосподарства. Користувач повинен мати можливість додавати власні категорії, редагувати та видаляти ті, що не пов'язані з транзакціями.

Управління членами родини. Система повинна підтримувати додавання кількох членів родини з присвоєнням індивідуального кольору аватару. Транзакції можуть бути прив'язані до конкретного члена родини або залишатися «загальними».

Бюджетне планування. Система повинна дозволяти встановлювати місячні ліміти витрат за категоріями. Для кожної категорії відображається поточна сума витрат, встановлений ліміт, відсоток використання та кольоровий індикатор статусу. Передбачена можливість копіювання лімітів з попереднього місяця.

Аналітичний дашборд. Головна сторінка повинна відображати: загальний баланс рахунку, суму доходів і витрат за поточний місяць, кругову діаграму розподілу витрат за категоріями, стовпчасту діаграму порівняння доходів і витрат за останні 6 місяців, список останніх 5 транзакцій.

Звітність та експорт. Система повинна формувати аналітичні звіти з фільтрацією за датою та категоріями, а також забезпечувати їх експорт у формати PDF (з таблицею транзакцій, зведеними підсумками та брендуванням) та CSV (з роздільником «крапка з комою», кодування UTF-8 для коректного відображення в Microsoft Excel).

Налаштування. Система повинна надавати можливість вибору валюти (з глобальним оновленням символу по всьому інтерфейсу), перемикання між світлою та темною темою, а також резервного копіювання бази даних у довільно обрану теку.

Повний перелік функціональних вимог з пріоритетами наведено в таблиці 2.1.

Таблиця 2.1 — Функціональні вимоги до системи

ID	Функціональна вимога	Пріоритет	Опис
FR-01	Додавання транзакцій	Обов'язк.	Форма з полями: сума, дата, тип (дохід/витрата), категорія, член родини, опис
FR-02	Редагування транзакцій	Обов'язк.	Діалогове вікно редагування з попередньо заповненими даними
FR-03	Видалення транзакцій	Обов'язк.	Видалення з підтвердженням через діалог
FR-04	Фільтрація транзакцій	Обов'язк.	Фільтри за датою, типом, категорією, членом родини, пошук по опису
FR-05	Перегляд дашборду	Обов'язк.	Відображення KPI, двох діаграм та останніх транзакцій
FR-06	Управління категоріями	Обов'язк.	CRUD для категорій, захист від видалення з транзакціями
FR-07	Управління членами родини	Обов'язк.	Додавання, видалення, відображення аватару з кольором
FR-08	Встановлення лімітів	Обов'язк.	Місячні ліміти витрат по категоріях з кольоровим статусом
FR-09	Копіювання лімітів	Бажаний	Перенесення лімітів з попереднього місяця одним кліком
FR-10	Генерація звітів	Обов'язк.	Фільтровані звіти з можливістю вибору типу діаграми

FR-11	Експорт у PDF	Обов'язк.	PDF-звіт із заголовком, таблицею транзакцій та підсумками
FR-12	Експорт у CSV	Обов'язк.	CSV-файл з роздільником «;», кодування UTF-8
FR-13	Вибір валюти	Бажаний	Глобальна зміна символу валюти (~~₴~~, $, €, £)
FR-14	Перемикання теми	Бажаний	Світла та темна тема Material Design
FR-15	Резервне копіювання БД	Бажаний	Копіювання файлу бази даних у вибрану теку

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи — як вона повинна функціонувати, а не що саме вона робить. Таблиця 2.2 містить перелік нефункціональних вимог, згрупованих за категоріями якості.

Таблиця 2.2 — Нефункціональні вимоги до системи

Категорія	Вимога	Критерій якості
Продуктивність	Час завантаження дашборду	Не більше 300 мс для бази до 10 000 транзакцій
Продуктивність	Час відгуку UI на дії користувача	Не більше 100 мс (миттєва реакція без блокування)
Продуктивність	Час генерації PDF-звіту	Не більше 2 с для звіту з 500 транзакціями
Надійність	Цілісність даних	Всі операції записи виконуються в межах транзакцій EF Core
Надійність	Захист від втрати даних	Функція резервного копіювання з міткою дати і часу
Зручність	Локалізація інтерфейсу	100% тексту інтерфейсу — українською мовою
Зручність	Відповідність Material Design	Використання стандартних компонентів MaterialDesignThemes
Зручність	Час навчання	Новий користувач може виконати першу транзакцію без інструкції
Масштабованість	Обсяг даних	Коректна робота з 10 000+ транзакцій без деградації
Масштабованість	Кількість членів родини	Підтримка до 10 членів родини
Безпека	Конфіденційність	Всі дані зберігаються локально без передачі по мережі
Портативність	Операційна система	Windows 10 (версія 1903+) та Windows 11 (x64)
Портативність	Залежності	Self-contained публікація без потреби у встановленні .NET

2.1.3 Діаграма варіантів використання

Діаграма варіантів використання (Use Case Diagram) відображає взаємодію між акторами системи та її функціональними можливостями. У

розроблюваній системі виділяються два актори: **Користувач (член родини)** — виконує більшість операцій (перегляд, введення та фільтрація транзакцій, перегляд дашборду та звітів) та **Адміністратор родини** — виконує адміністративні функції (управління категоріями, членами родини, лімітами, налаштуваннями та резервним копіюванням).

На практиці обидва актори можуть бути однією особою, однак логічне розмежування ролей допомагає структурувати опис системи. Діаграма варіантів використання представлена на рисунку 2.1.

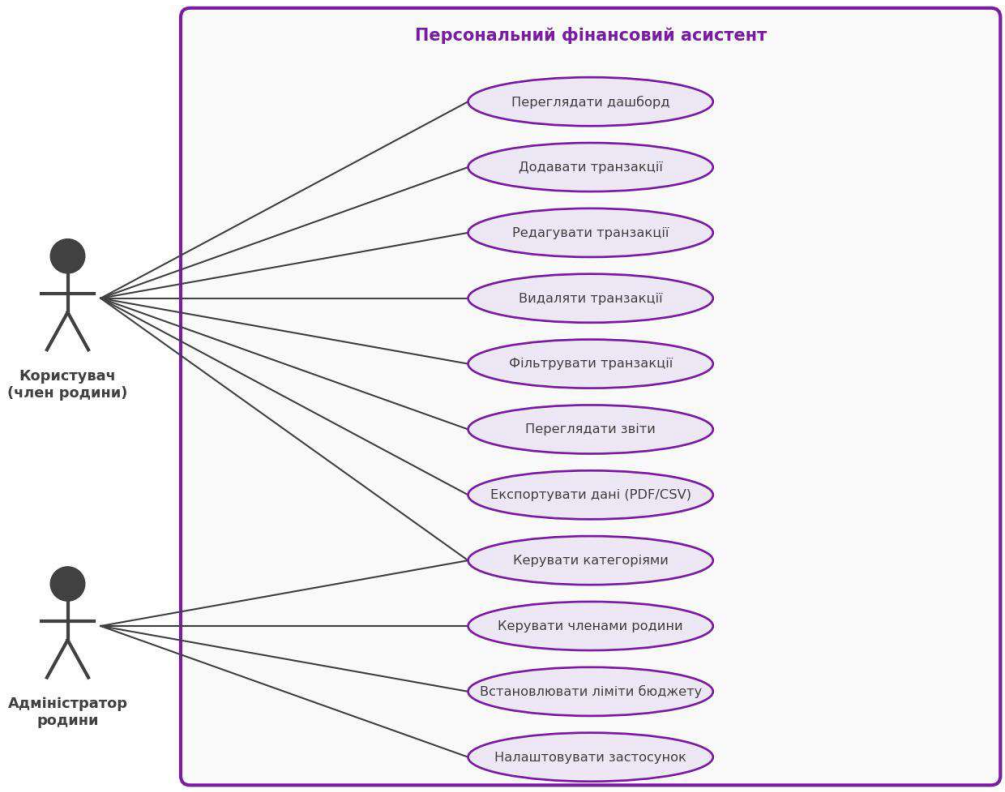


Рисунок 2.1 — Діаграма варіантів використання системи

2.2 Проектування бази даних

2.2.1 Концептуальна модель даних

Концептуальна модель даних визначає основні сутності предметної області та зв'язки між ними. В системі персонального фінансового асистента виділено п'ять ключових сутностей:

Transaction (Транзакція) — центральна сутність системи, що відображає фінансову операцію (дохід або витрату). Транзакція обов'язково пов'язана з категорією та необов'язково — з конкретним членом родини.

Category (Категорія) — класифікатор транзакцій. Кожна категорія має тип (дохід або витрата), назву, іконку та колір для візуального відображення. Категорії можуть бути системними (попередньо завантаженими) або користувацькими.

FamilyMember (Член родини) — представник домогосподарства, якому можуть бути призначені транзакції. Має ім'я та колір аватару для персоналізованого відображення у фільтрах та звітах.

BudgetLimit (Ліміт бюджету) — встановлений місячний ліміт витрат для конкретної категорії в конкретному місяці та році. Використовується для порівняння з фактичними витратами та розрахунку статусу виконання бюджету.

Setting (Налаштування) — пара ключ-значення для зберігання параметрів застосунку (вибрана валюта, поточна тема оформлення).

2.2.2 ER-діаграма та зв'язки між сутностями

Зв'язки між сутностями організовані наступним чином:

- **Category → Transaction** (один-до-багатьох): одна категорія може мати багато транзакцій. При спробі видалення категорії, яка має транзакції, виникає помилка обмеження (Restrict), що захищає цілісність даних.
- **FamilyMember → Transaction** (один-до-багатьох, необов'язковий): один член родини може бути пов'язаний з багатьма транзакціями. При видаленні члена родини значення FamilyMemberId у його транзакціях обнуляється (SetNull), а самі транзакції зберігаються.
- **Category → BudgetLimit** (один-до-багатьох): одна категорія може мати кілька лімітів (по одному на кожний місяць). При видаленні категорії пов'язані ліміти також видаляються (Cascade).

ER-діаграма бази даних представлена на рисунку 2.2.

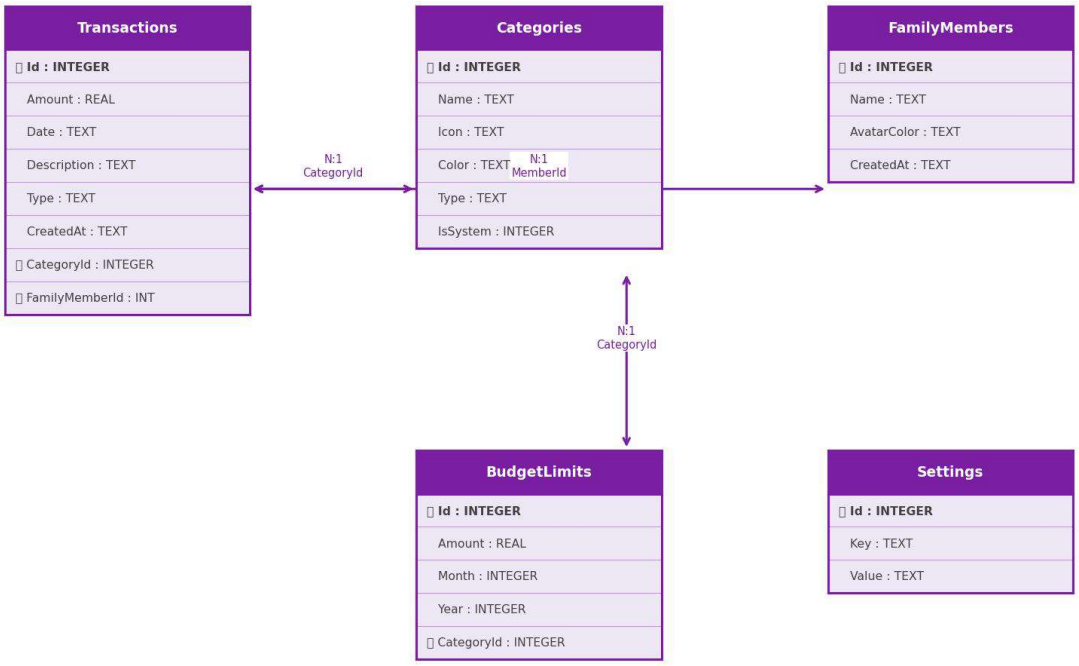


Рисунок 2.2 – ER-діаграма бази даних

2.2.3 Фізична модель бази даних

Фізична модель визначає конкретну структуру таблиць SQLite — типи даних стовпців, обмеження цілісності та індекси. Детальна структура ключових таблиць наведена в таблицях 2.3 та 2.4.

Таблиця 2.3 — Структура таблиці Transactions

Стовпець	Тип даних	Обмеження	Опис
Id	INTEGER	PRIMARY KEY, AUTOINCREMENT	Унікальний ідентифікатор транзакції
Amount	REAL	NOT NULL, CHECK(Amount > 0)	Сума транзакції (завжди позитивна)
Date	TEXT	NOT NULL	Дата транзакції у форматі ISO 8601
Description	TEXT	—	Текстовий опис (необов'язковий)
Type	TEXT	NOT NULL, CHECK(Type IN ('Income','Expense'))	Тип: дохід або витрата
CreatedAt	TEXT	DEFAULT CURRENT_TIMESTAMP	Дата і час створення запису
CategoryId	INTEGER	NOT NULL, FK → Categories(Id) RESTRICT	Посилання на категорію
FamilyMemberId	INTEGER	NULL, FK → FamilyMembers(Id) SET NULL	Посилання на члена родини

Таблиця 2.4 — Структура таблиці Categories

Стовпець	Тип даних	Обмеження	Опис
Id	INTEGER	PRIMARY KEY, AUTOINCREMENT	Унікальний ідентифікатор
Name	TEXT	NOT NULL, UNIQUE	Назва категорії (наприклад, «Продукти харчування»)
Icon	TEXT	NOT NULL	Код іконки Material Design
Color	TEXT	NOT NULL	Колір у форматі HEX (#RRGGBB)
Type	TEXT	NOT NULL, CHECK(Type IN ('Income','Expense'))	Тип категорії
IsSystem	INTEGER	NOT NULL, DEFAULT 0	1 — системна (не можна видалити), 0 — користувацька

2.2.4 Попередньо визначені категорії (Seed Data)

При першому запуску застосунку EF Core автоматично заповнює таблицю Categories вісімнадцятьма попередньо визначеними категоріями українською мовою, що охоплюють найтиповіші статті домашнього бюджету. Повний перелік початкових категорій наведено в таблиці 2.5.

Таблиця 2.5 — Перелік попередньо визначених категорій системи

№	Назва категорії	Тип	Іконка	Колір
1	Продукти харчування	Витрата	ShoppingCart	#4CAF50
2	Транспорт	Витрата	DirectionsBus	#2196F3
3	Комунальні послуги	Витрата	ElectricBolt	#FF9800
4	Зв'язок та інтернет	Витрата	PhoneAndroid	#9C27B0
5	Розваги та дозвілля	Витрата	TheaterComedy	#E91E63
6	Одяг та взуття	Витрата	Checkroom	#795548
7	Охорона здоров'я	Витрата	LocalHospital	#F44336
8	Освіта	Витрата	School	#3F51B5
9	Ресторани та кафе	Витрата	Restaurant	#FF5722
10	Побутова техніка	Витрата	HomeAppliance	#607D8B
11	Подорожі	Витрата	FlightTakeoff	#00BCD4
12	Спорт та фітнес	Витрата	FitnessCenter	#8BC34A
13	Краса та догляд	Витрата	Spa	#EC407A
14	Подарунки та благодійність	Витрата	CardGiftcard	#FFC107
15	Заощадження	Витрата	Savings	#009688
16	Зарплата	Дохід	Work	#43A047
17	Фріланс та підробіток	Дохід	Laptop	#1E88E5
18	Інші доходи	Дохід	AttachMoney	#00ACC1

2.2.5 Стратегія міграцій EF Core

Управління схемою бази даних здійснюється через систему міграцій Entity Framework Core. При першому запуску застосунку метод MigrateAsync() автоматично застосовує всі необхідні міграції:

- **InitialCreate** — створює всі таблиці відповідно до поточної моделі даних.
- **SeedData** — заповнює таблицю Categories вісімнадцятьма початковими категоріями та таблицю Settings значеннями за замовчуванням (валюта — «₴», тема — «Light»).

Фізична модель бази даних з усіма таблицями та їх структурою представлена на рисунку 2.3.

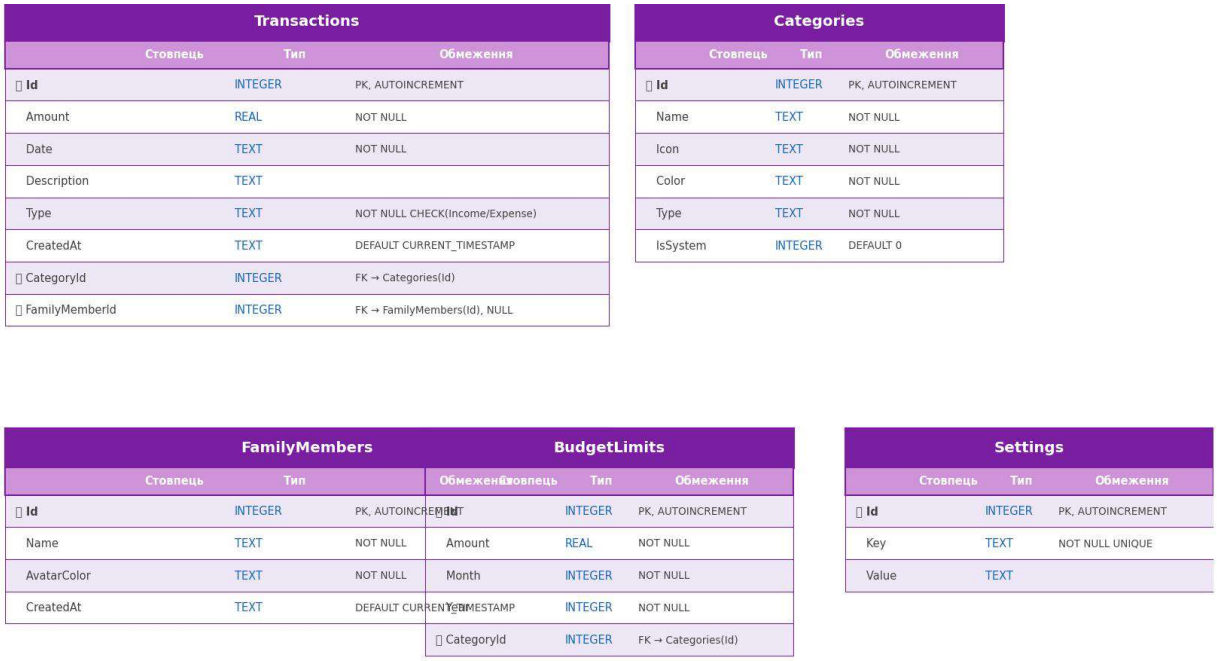


Рисунок 2.3 — Фізична модель бази даних

2.3 Проектування архітектури застосунку

2.3.1 Загальна шарова архітектура

Архітектура застосунку побудована за принципом чіткого розділення відповідальностей між шарами (Layered Architecture). Кожен шар взаємодіє лише з безпосередньо нижчим шаром і не знає про існування вищих. Це забезпечує незалежність компонентів, легкість тестування та можливість заміни окремих шарів без змін в інших.

Застосунок містить п'ять шарів:

1. **Presentation Layer (Шар подання)** — XAML-представлення (Views), система стилів та конвертери значень. Відповідає виключно за відображення даних та реакцію на дії користувача.
2. **ViewModel Layer (Шар ViewModel)** — логіка подання, що реалізує патерн MVVM через класи, успадковані від ObservableObject. Отримує дані від сервісів та передає їх у View через прив'язки.
3. **Business Logic Layer (Шар бізнес-логіки)** — сервіси, що реалізують бізнес-правила: розрахунок бюджетних статусів, генерацію PDF та CSV, управління навігацією та налаштуваннями.
4. **Data Access Layer (Шар доступу до даних)** — репозиторії та Unit of Work, що абстрагують доступ до бази даних через EF Core.
5. **Data Layer (Шар даних)** — файл бази даних SQLite з усіма таблицями та міграціями.

Діаграма компонентів застосунку представлена на рисунку 2.4.

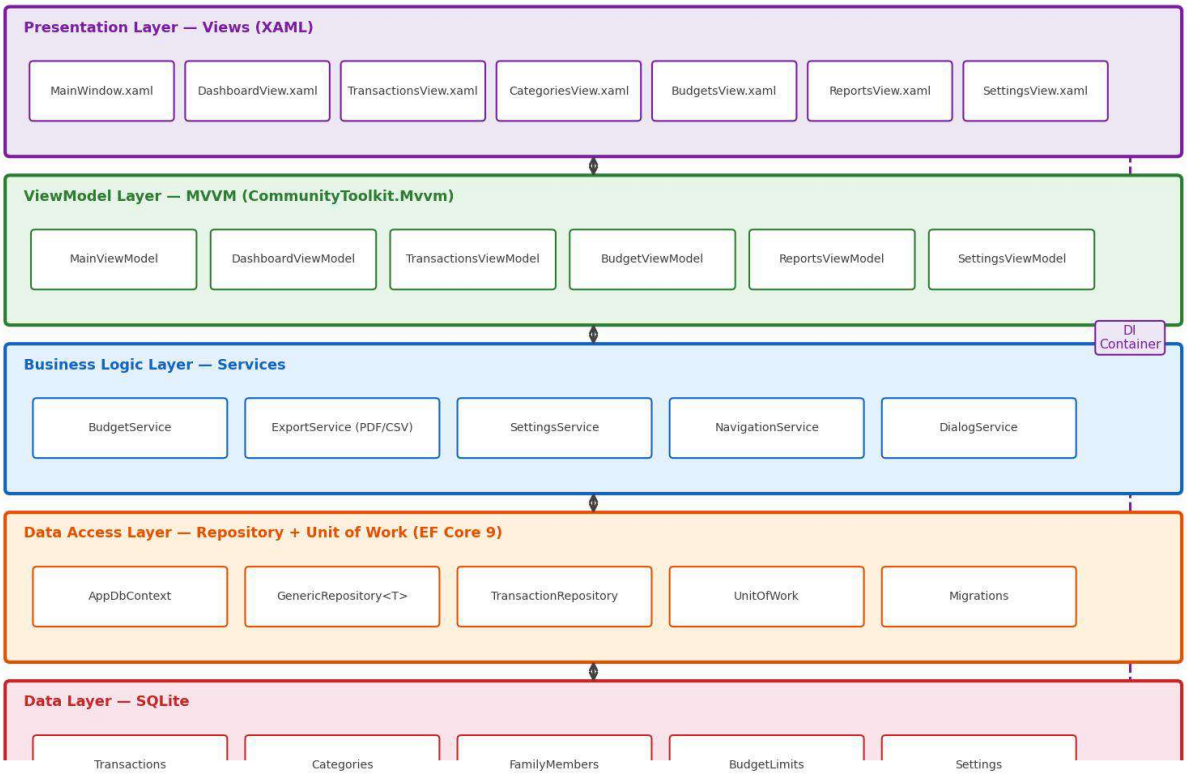
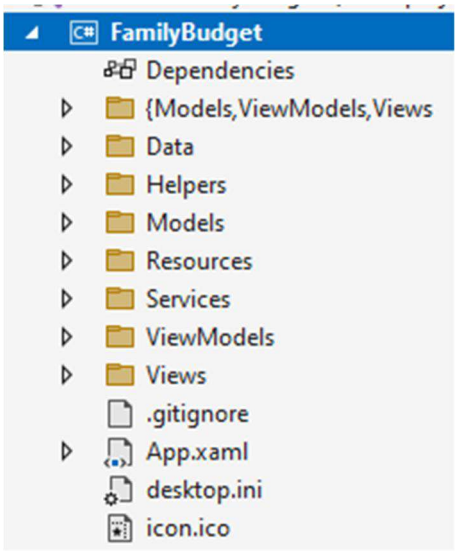


Рисунок 2.4 — Діаграма компонентів застосунку

2.3.2 Структура папок проєкту

Фізична організація проєкту відображає логічну шарову архітектуру: кожна папка відповідає певній відповідальності та шару системи:



2.3.3 Конфігурація Dependency Injection

Впровадження залежностей (DI) конфігурується в App.xaml.cs при запуску застосунку через IHostBuilder. Всі залежності реєструються в IoC-контейнері Microsoft.Extensions.DependencyInjection. Перелік ViewModel-класів та їхніх залежностей наведено в таблиці 2.6, а перелік сервісів — у таблиці 2.7.

Таблиця 2.6 — Перелік ViewModel-класів та їхня відповідальність

Клас ViewModel	Призначення	Основні залежності	Тип DI
MainViewModel	Координація навігації між сторінками	INavigationService	Singleton
DashboardViewModel	Відображення KPI, діаграм, останніх транзакцій	IUnitOfWork, IBudgetService	Transient
TransactionsViewModel	CRUD транзакцій з фільтрацією	IUnitOfWork, IDialogService	Transient
CategoriesViewModel	Управління категоріями	IUnitOfWork, IDialogService	Transient
FamilyMembersViewModel	Управління членами родини	IUnitOfWork, IDialogService	Transient
BudgetViewModel	Ліміти бюджету з кольоровим статусом	IUnitOfWork, IBudgetService	Transient
ReportsViewModel	Звіти з фільтрацією та експортом	IUnitOfWork, IExportService	Transient
SettingsViewModel	Налаштування застосунку та бекап	ISettingsService, IDialogService	Transient

Таблиця 2.7 — Перелік сервісів системи

Інтерфейс	Реалізація	Тип DI	Призначення
INavigationService	NavigationService	Singleton	Перемикання між сторінками через ContentControl
IDialogService	DialogService	Transient	Модальні діалоги (підтвердження, помилки, OpenFileDialog)
IBudgetService	BudgetService	Transient	Розрахунок витрат та статусу виконання лімітів
IExportService	ExportService	Transient	Генерація PDF (QuestPDF) та CSV (CsvHelper)
ISettingsService	SettingsService	Singleton	Читання/запис налаштувань, управління CurrencyProvider
IUnitOfWork	UnitOfWork	Scoped	Координація репозиторіїв та збереження змін

2.3.4 Діаграми класів

Діаграма класів шару даних (рисунок 2.5) демонструє ієрархію інтерфейсів та їхніх реалізацій: базовий інтерфейс IRepository<T> реалізується класом GenericRepository<T>, який містить реалізацію п'яти стандартних CRUD-операцій. Спеціалізований TransactionRepository розширює GenericRepository<Transaction> додатковими методами для фільтрації та агрегації. Інтерфейс IUnitOfWork об'єднує всі репозиторії та надає метод SaveChangesAsync().

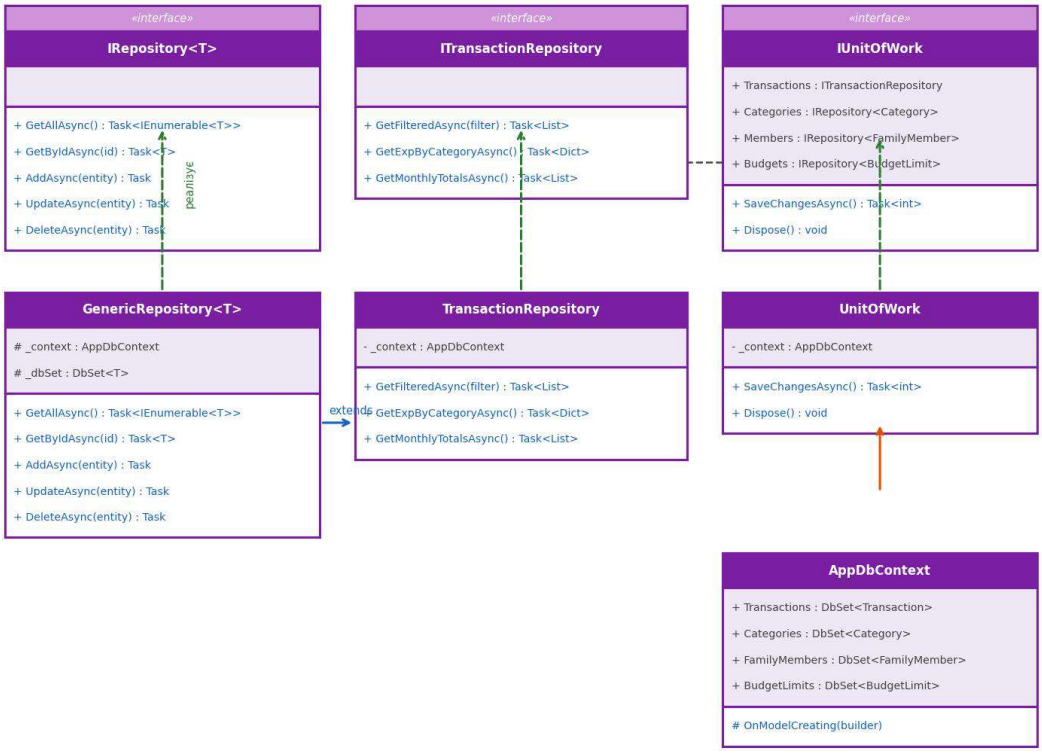


Рисунок 2.5 — Діаграма класів шару даних (Repository + Unit of Work)

Діаграма класів ViewModel-шару (рисунок 2.6) показує, що всі ViewModel успадковують від базового класу BaseViewModel, який реалізує INotifyPropertyChanged та надає допоміжні методи SetProperty<T>() і LoadAsync(). Кожен конкретний ViewModel додає власні властивості та команди відповідно до своєї відповідальності.

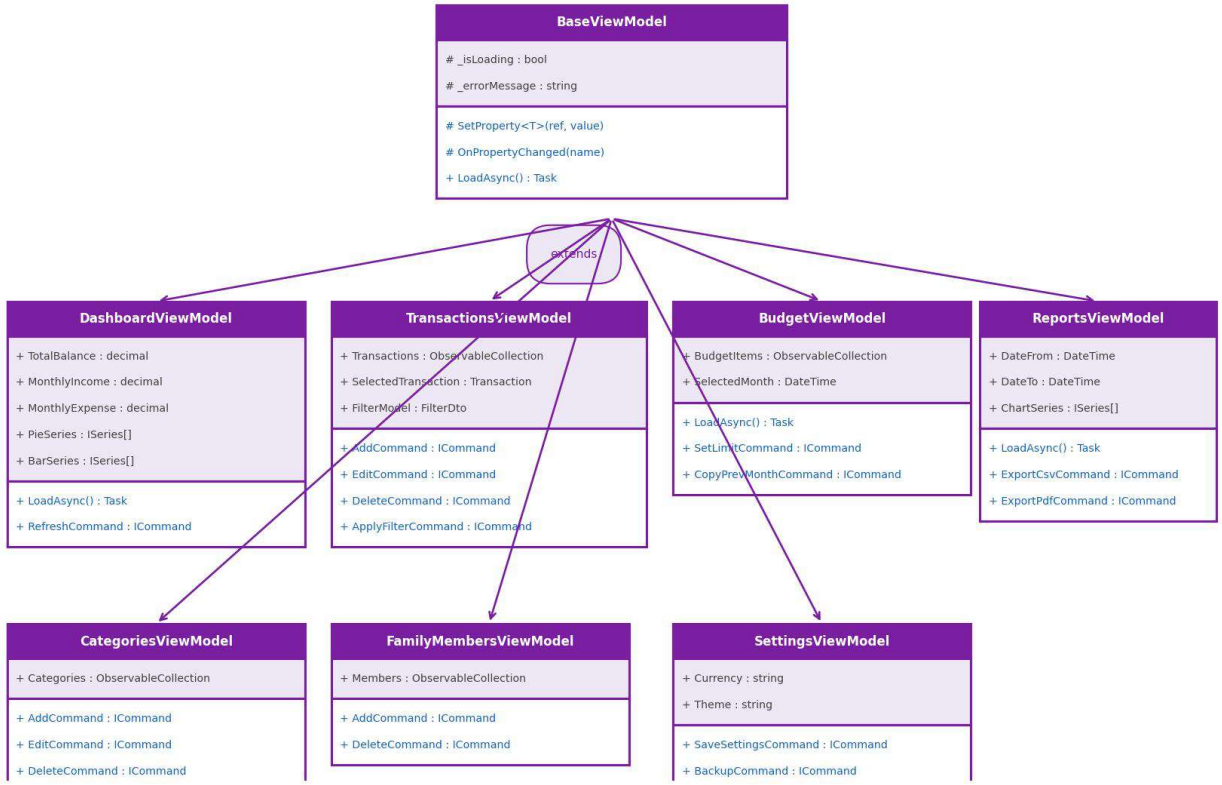


Рисунок 2.6 — Діаграма класів ViewModel-шару

2.3.5 Схема навігації

Навігація між сторінками реалізована через патерн «Single Window Application»: застосунок має одне головне вікно (MainWindow) з фіксованою бічною навігаційною панеллю та областю вмісту (ContentControl). При виборі пункту меню NavigationService замінює поточний вміст ContentControl відповідним UserControl. Схема навігації між сімома сторінками застосунку представлена на рисунку 2.7.

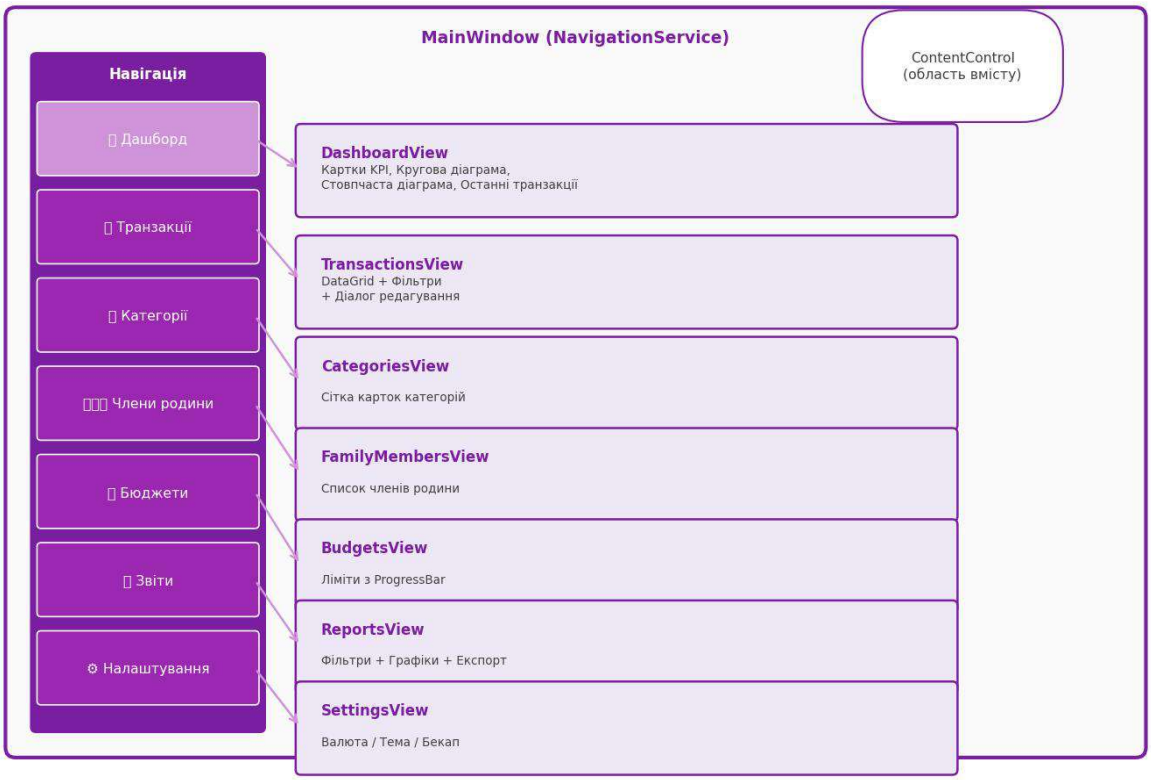


Рисунок 2.7 — Схема навігації між сторінками застосунку

2.4 Проєктування інтерфейсу користувача

2.4.1 Принципи проєктування UI

Інтерфейс застосунку розроблено відповідно до специфікації **Material Design 3** від Google, адаптованої для WPF через бібліотеку MaterialDesignThemes 5.1. Обрана **пурпурна кольорова схема** з основним кольором #7B1FA2 (Deep Purple 700) та акцентним #CE93D8 (Purple 200), яка забезпечує: асоціацію з фінансовою тематикою (відчуття серйозності та надійності), достатній контраст між елементами відповідно до стандарту WCAG AA, естетичну привабливість та сучасний вигляд.

Ключові дизайнерські рішення:

- **Єдине вікно з навігаційною панеллю** замість традиційних вкладок або окремих вікон — забезпечує безперервність контексту.
- **Картковий UI** (Material Design Cards) — кожен смисловий блок інформації (KPI-показник, категорія, ліміт) оформлений карткою з тінню.

- **Кольорове кодування** — зелений для доходів/норми, помаранчевий для витрат/попередження, червоний для перевищення ліміту.
- **Асинхронне завантаження** — при переході між сторінками відображається індикатор завантаження, UI не блокується.

2.4.2 Специфікація екранів

Повна специфікація всіх екранів застосунку з переліком ключових елементів та прив'язаними ViewModel наведена в таблиці 2.8.

Таблиця 2.8 — Специфікація екранів застосунку

Екран	Ключові UI-елементи	Прив'язаний ViewModel	Рис. макету
Головне вікно	NavigationDrawer (ListBox), ContentControl, заголовок з місяцем	MainViewModel	2.8
Дашборд	3 картки KPI, PieChart (LiveCharts), CartesianChart (LiveCharts), DataGrid останніх транзакцій	DashboardViewModel	2.9
Транзакції	ToolBar з кнопками CRUD, панель фільтрів (DatePicker ×2, ComboBox ×3, TextBox), DataGrid з ColorConverter	TransactionsViewModel	2.10
Додавання транзакції	RadioButton (тип), NumericUpDown (сума), DatePicker, ComboBox (категорія/член), TextBox (опис)	TransactionDialogViewModel	2.11
Категорії	WrapPanel карток з іконкою, назвою та лічильником транзакцій, кнопки CRUD	CategoriesViewModel	—
Члени родини	ItemsControl карток з кольоровим аватаром, кнопки Додати/Видалити	FamilyMembersViewModel	—
Ліміти бюджету	ItemsControl карток ліміту (ProgressBar, % витрат, StatusBadge), перемикач місяця	BudgetViewModel	2.12
Звіти	DatePicker ×2, ComboBox (категорія), CartesianChart, кнопки «Експорт PDF» та «Експорт CSV»	ReportsViewModel	—
Налаштування	ComboBox (валюта), ToggleButton (тема), Button (резервне копіювання)	SettingsViewModel	—

2.4.3 Макети ключових екранів

Загальна структура головного вікна з навігаційною панеллю та областю вмісту представлена на рисунку 2.8. Бічна панель займає фіксовану ширину і містить пункти меню для переходу між семи розділами застосунку. При виборі активного пункту він підсвічується світлішим відтінком пурпурного.

Рис. 2.8 — Макет (wireframe) головного вікна застосунку

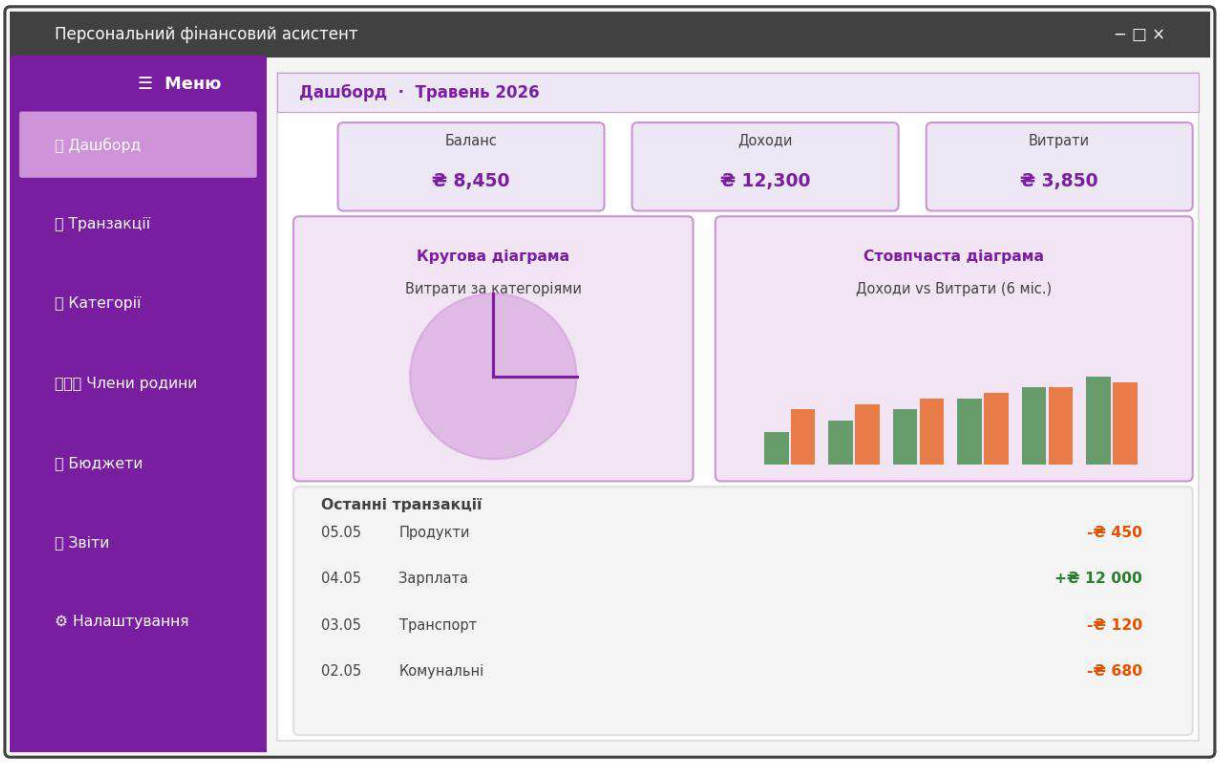


Рисунок 2.8 — Макет (wireframe) головного вікна з навігаційною панеллю

Детальний макет екрану «Дашборд» (рисунок 2.9) містить три картки KPI у верхній частині: загальний баланс, доходи за місяць та витрати за місяць. Нижче розміщені дві діаграми: кругова для розподілу витрат за категоріями поточного місяця та стовпчаста для порівняння доходів і витрат за останні 6 місяців. В нижній частині — таблиця з п'ятьма останніми транзакціями.

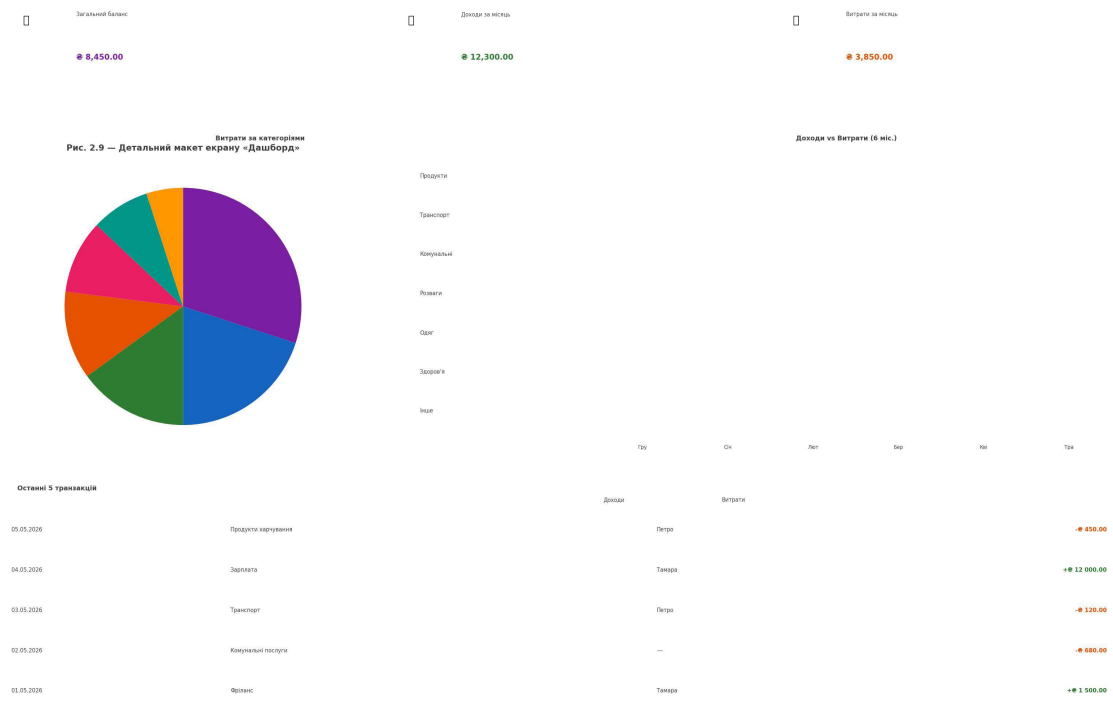


Рисунок 2.9 — Детальний макет екрану «Дашборд»

Макет екрану «Транзакції» (рисунок 2.10) включає панель інструментів з кнопками «Додати», «Редагувати» та «Видалити», рядок фільтрів (два DatePicker для діапазону дат, три ComboBox для вибору типу, категорії та члена родини, TextBox для текстового пошуку) та таблицю транзакцій з підсвічуванням сум: зеленим для доходів, помаранчевим для витрат.

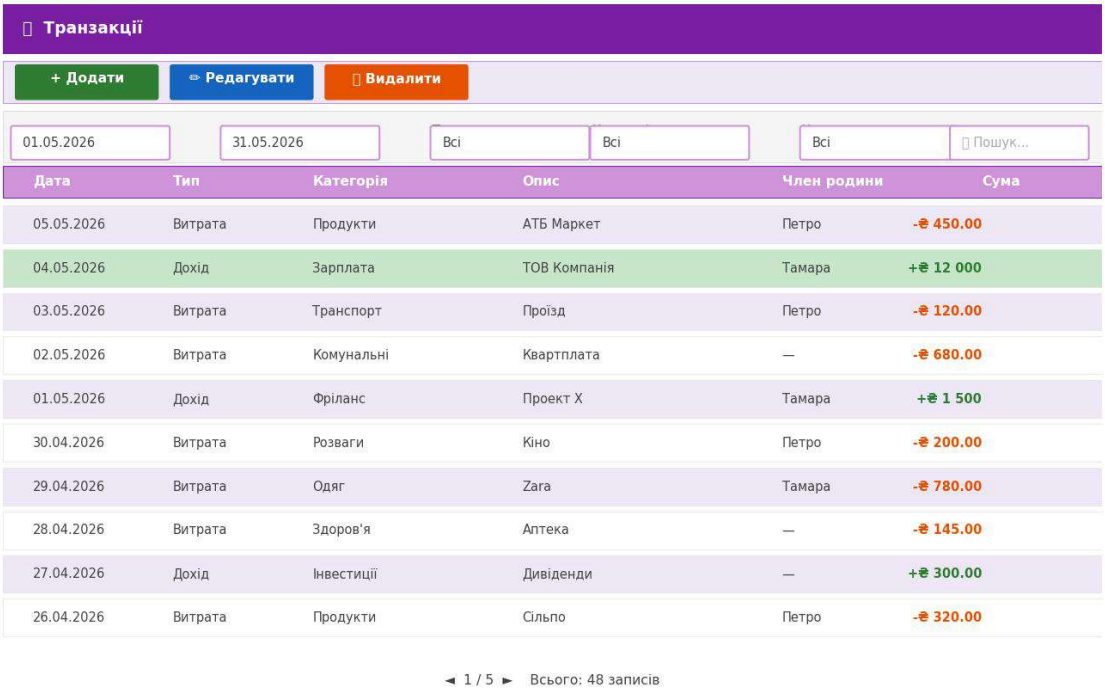


Рисунок 2.10 — Макет екрану «Транзакції» з формою фільтрації

Діалогове вікно додавання транзакції (рисунок 2.11) є модальним і містить: перемикач типу транзакції (Витрата / Дохід) у вигляді кнопок-радіокнопок, поля для введення суми, вибору дати, категорії, члена родини та текстового опису, а також кнопки «Зберегти» та «Скасувати». Обов'язкові поля позначені зірочкою.

Рисунок 2.11 — Макет діалогового вікна додавання транзакції

Макет екрану «Ліміти бюджету» (рисунок 2.12) відображає картку для кожної категорії, що має встановлений ліміт. Картка містить назву категорії, текстовий показник «₴ витрачено / ₴ ліміт», відсоток виконання та горизонтальний ProgressBar, колір якого змінюється залежно від статусу: зелений (до 80%), жовтий (80–99%), червоний (100% і більше). Внизу екрану розміщена легенда кольорових статусів.

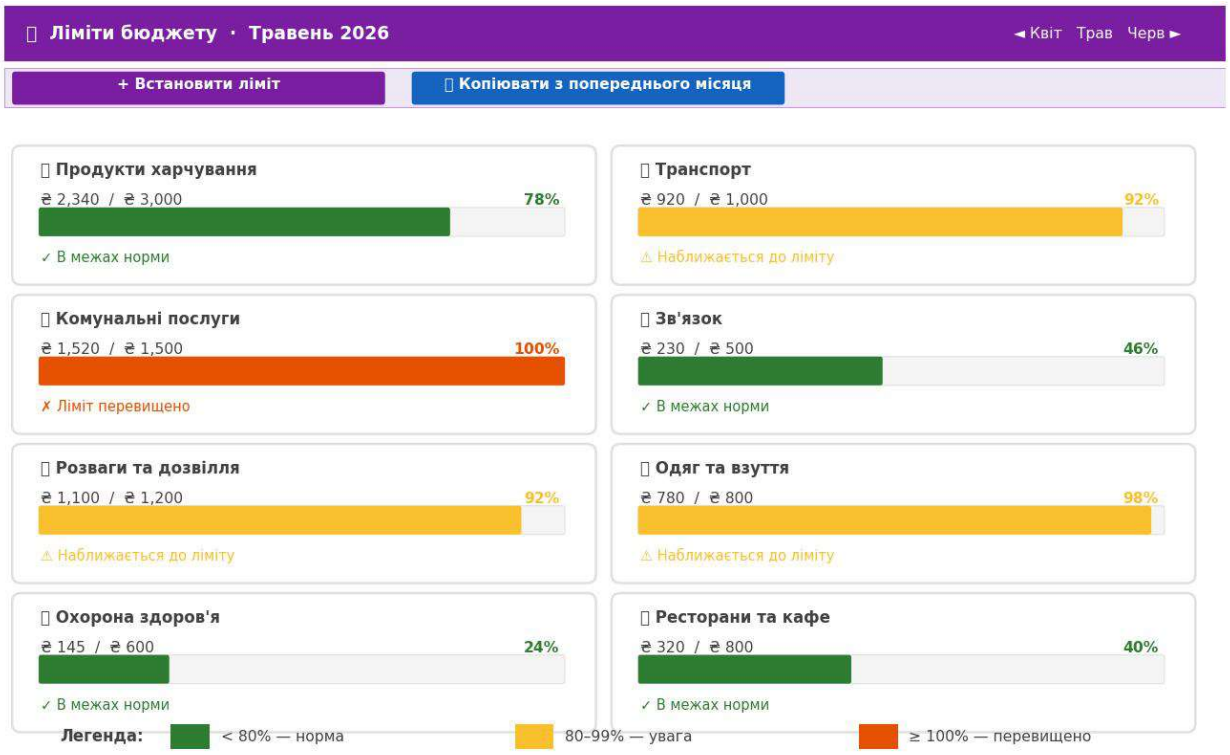


Рисунок 2.12 — Макет екрану «Ліміти бюджету» з кольоровою індикацією

2.5 Проектування алгоритмів обробки даних

2.5.1 Алгоритм розрахунку показників дашборду

При переході на екран «Дашборд» ViewModel виконує паралельне завантаження даних через чотири незалежні асинхронні запити:

1. GetTotalBalanceAsync() — агрегує всі транзакції: сума доходів мінус сума витрат за весь час.
2. GetMonthlyTotalsAsync(month, year) — обчислює суму доходів і витрат за поточний місяць (використовується для карток KPI).
3. GetExpensesByCategoryAsync(month, year) — групує витрати поточного місяця за категоріями (дані для кругової діаграми).
4. GetSixMonthTotalsAsync() — обчислює місячні суми доходів і витрат за останні 6 місяців (дані для стовпчастої діаграми).

Результати цих запитів трансформуються у формат ISeries[] бібліотеки LiveCharts2 та прив'язуються до властивостей ViewModel. Завдяки ObservableCollection та INotifyPropertyChanged View автоматично оновлюється після завантаження даних.

2.5.2 Алгоритм моніторингу лімітів бюджету

Алгоритм моніторингу бюджетних лімітів реалізований у класі BudgetService та виконується щоразу при відкритті екрану «Ліміти бюджету» або після додавання нової транзакції. Блок-схема алгоритму представлена на рисунку 2.13.

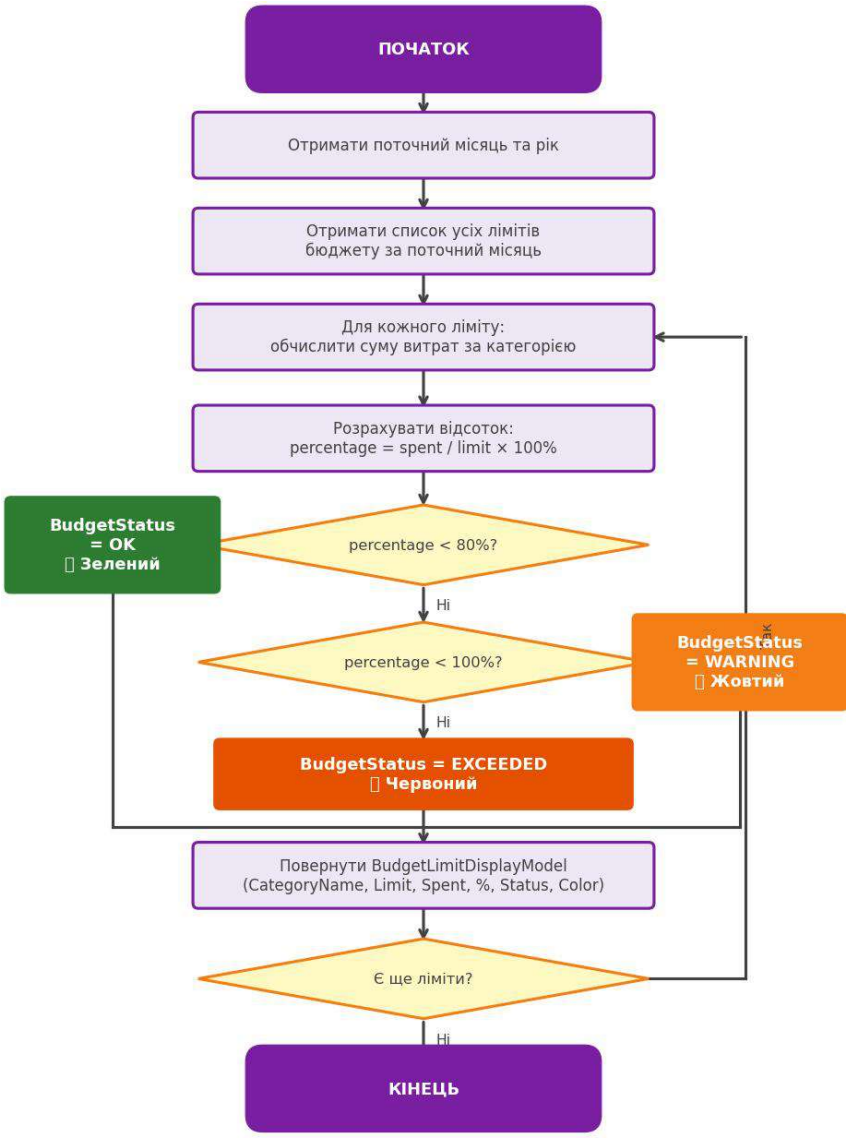


Рисунок 2.13 — Блок-схема алгоритму моніторингу лімітів бюджету

Алгоритм виконує такі кроки:

1. Отримує список усіх лімітів бюджету для поточного місяця та року.
2. Для кожного ліміту виконує запит до TransactionRepository для отримання суми витрат за відповідною категорією в поточному місяці.
3. Розраховує відсоток використання ліміту: percentage = spent / limit × 100%.

4. Визначає статус бюджету відповідно до порогових значень:
    - BudgetStatus.OK (зелений) — якщо percentage < 80%;
    - BudgetStatus.Warning (жовтий) — якщо 80% ≤ percentage < 100%;
    - BudgetStatus.Exceeded (червоний) — якщо percentage ≥ 100%.
5. Формує об'єкт BudgetLimitDisplayModel з усіма розрахованими полями та повертає колекцію для відображення у View.

2.5.3 Алгоритм генерації PDF-звіту

Алгоритм генерації PDF реалізований у класі ExportService з використанням бібліотеки QuestPDF. Блок-схема алгоритму представлена на рисунку 2.14.

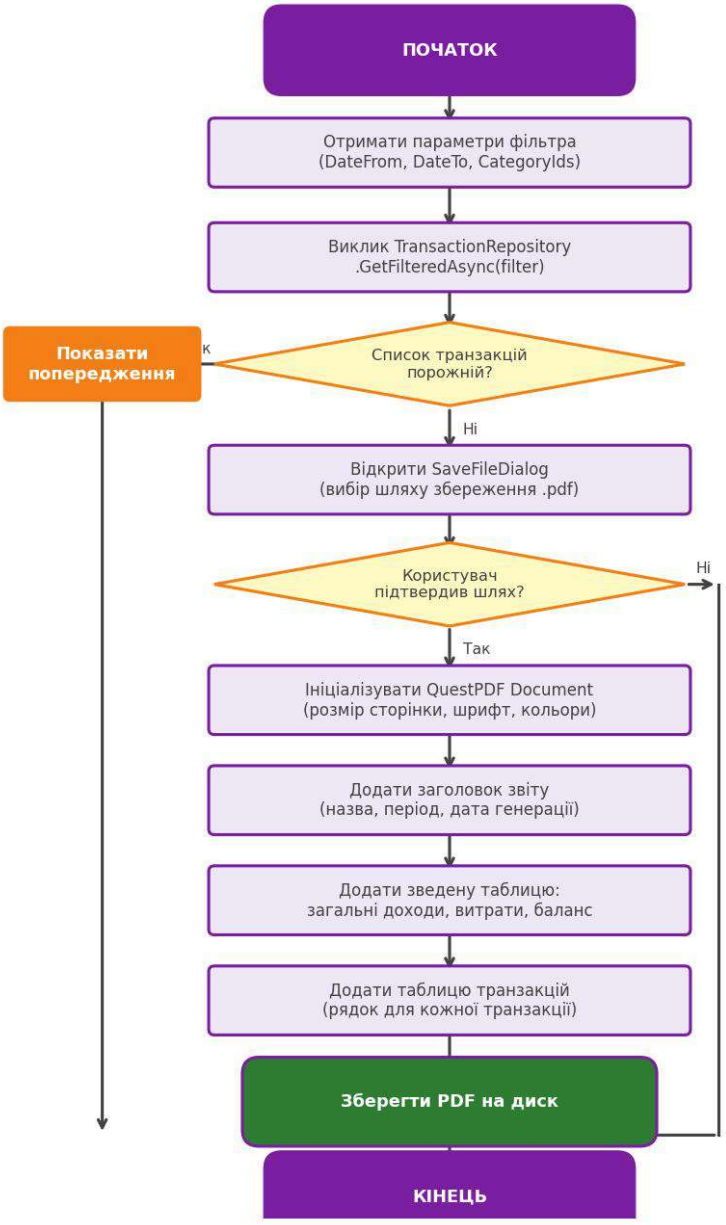


Рисунок 2.14 — Блок-схема алгоритму генерації PDF-звіту

Послідовність кроків алгоритму:

1. Отримати параметри фільтра від ReportsViewModel (діапазон дат, список категорій).
2. Виконати TransactionRepository.GetFilteredAsync(filter) для отримання відфільтрованого списку транзакцій.
3. Якщо список порожній — показати попередження через IDialogService і завершити виконання.
4. Відкрити SaveFileDialog для вибору користувачем шляху та імені файлу.
5. Якщо користувач скасував вибір — завершити виконання.
6. Ініціалізувати QuestPDF Document із налаштуваннями сторінки (формат A4, відступи 20 мм).
7. Додати заголовок звіту: назва застосунку, період фільтрації, дата генерації.
8. Додати зведену таблицю підсумків: загальна сума доходів, витрат, баланс за вибраний період.
9. Додати детальну таблицю транзакцій (дата, категорія, опис, член родини, сума) з чергуванням кольорів рядків.
10. Зберегти сформований документ у вказаний файл через document.GeneratePdf(filePath).

2.5.4 Алгоритм резервного копіювання бази даних

Алгоритм резервного копіювання реалізований у SettingsService і складається з таких кроків:

1. Закрити всі активні підключення до SQLite через виклик _context.Database.CloseConnection().
2. Сформувати ім'я файлу резервної копії з міткою часу у форматі family_budget_backup_YYYY-MM-DD_HH-mm.db.
3. Відкрити SaveFileDialog для вибору теки призначення.
4. Виконати File.Copy(sourcePath, destinationPath, overwrite: false).
5. Відновити підключення до бази даних та повідомити користувача про успішне виконання.

При відновленні з резервної копії виконується зворотна процедура: поточний файл замінюється вибраним файлом резервної копії з обов'язковим перезапуском застосунку для застосування змін.

Висновки до розділу 2

У другому розділі виконано повний цикл проектування персонального фінансового асистента для ведення сімейного бюджету.

Визначено 15 функціональних вимог, що охоплюють управління транзакціями, категоріями, членами родини, бюджетними лімітами, аналітичним дашбордом, звітністю та налаштуваннями. Сформульовано 13 нефункціональних вимог за категоріями продуктивності, надійності, зручності, масштабованості, безпеки та портативності.

Спроектовано реляційну базу даних з п'яти таблиць (Transactions, Categories, FamilyMembers, BudgetLimits, Settings). Визначено зв'язки між сутностями та стратегії забезпечення цілісності даних (Restrict, SetNull, Cascade). Підготовлено перелік із 18 попередньо визначених категорій українською мовою.

Розроблено шарову архітектуру застосунку з п'яти рівнів, що реалізує патерни MVVM, Repository та Unit of Work. Визначено 8 ViewModel-класів та 6 сервісів з їхніми залежностями та типами реєстрації в DI-контейнері.

Спроектовано інтерфейс користувача відповідно до специфікації Material Design 3 з пурпурною кольоровою схемою. Розроблено wireframe-макети для п'яти ключових екранів застосунку та описано специфікацію всіх дев'яти екранів.

Описано та проілюстровано блок-схемами чотири ключові алгоритми: розрахунок показників дашборду, моніторинг бюджетних лімітів, генерацію PDF-звіту та резервне копіювання бази даних.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Налаштування середовища розробки та структура проєкту

3.1.1 Конфігурація середовища розробки

Розробку застосунку виконано у середовищі **Visual Studio 2022** (версія 17.12) — провідній IDE для розробки .NET-застосунків від Microsoft. Для реалізації обрано мову програмування C# 13 у поєднанні з платформою .NET 9. Повна конфігурація середовища розробки наведена в таблиці 3.1.

Таблиця 3.1 — Конфігурація середовища розробки

Компонент	Версія	Призначення
Visual Studio 2022 Community	17.12.x	Основне середовище розробки (IDE)
.NET 9 SDK	9.0.x	Платформа виконання та компілятор C# 13
C#	13.0	Мова програмування
NuGet Package Manager	вбудований	Управління залежностями проєкту
EF Core CLI Tools	9.0.0	Управління міграціями (Add-Migration, Update-Database)
DB Browser for SQLite	3.13.x	Перегляд та редагування SQLite-бази під час розробки
Git	2.x	Система контролю версій
Windows 11	23H2	Операційна система розробника

3.1.2 Налаштування проєкту

Проєкт створено як **WPF Application** із цільовою платформою net9.0-windows. Файл FamilyBudget.csproj містить декларацію всіх NuGet-залежностей та налаштування компілятора. Ключові параметри проєктного файлу:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net9.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RootNamespace>FamilyBudget</RootNamespace>
  </PropertyGroup>
</Project>
```

Ввімкнення <Nullable>enable</Nullable> забезпечує контроль null-безпечності на рівні компілятора, що суттєво зменшує ймовірність NullReferenceException у runtime.

Конфігурація Material Design виконується в App.xaml через додавання словників ресурсів до Application.Resources:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <materialDesign:BundledTheme BaseTheme="Light"
          PrimaryColor="DeepPurple" SecondaryColor="Purple"/>
      <ResourceDictionary Source="pack://...MaterialDesignTheme.xaml"/>
      <ResourceDictionary Source="Resources/AppStyles.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Структура проєкту у вікні Solution Explorer Visual Studio відображає логічну організацію файлів відповідно до шарової архітектури та представлена на рисунку 3.1.

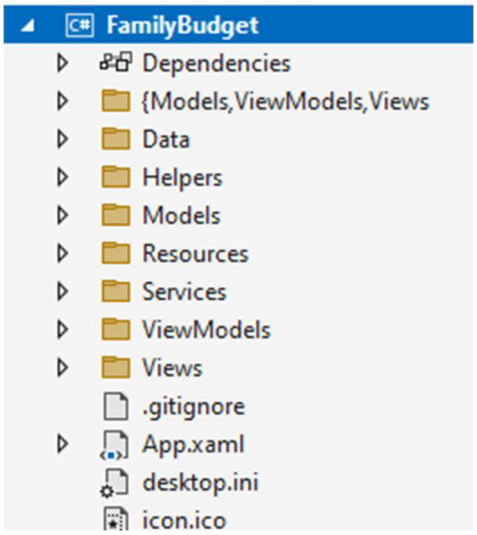


Рисунок 3.1 — Структура проєкту у Solution Explorer Visual Studio

3.2 Реалізація шару даних

3.2.1 Реалізація AppDbContext та моделей

Клас AppDbContext успадковується від DbContext (EF Core) та є центральною точкою доступу до бази даних. Він визначає DbSet<T> для кожної сутності та конфігурує схему через метод OnModelCreating:

```
public class AppDbContext : DbContext
{
    public DbSet<Transaction> Transactions => Set<Transaction>();
    public DbSet<Category>    Categories   => Set<Category>();
    public DbSet<FamilyMember> FamilyMembers => Set<FamilyMember>();
    public DbSet<BudgetLimit> BudgetLimits  => Set<BudgetLimit>();
    public DbSet<Setting>     Settings      => Set<Setting>();

    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }

    protected override void OnModelCreating(ModelBuilder mb)
    {
        // Зв'язок Category → Transaction (Restrict)
        mb.Entity<Transaction>()
          .HasOne(t => t.Category)
          .WithMany(c => c.Transactions)
          .HasForeignKey(t => t.CategoryId)
          .OnDelete(DeleteBehavior.Restrict);

        // Зв'язок FamilyMember → Transaction (SetNull)
        mb.Entity<Transaction>()
          .HasOne(t => t.FamilyMember)
          .WithMany(m => m.Transactions)
          .HasForeignKey(t => t.FamilyMemberId)
          .IsRequired(false)
          .OnDelete(DeleteBehavior.SetNull);

        // Seed data для 18 категорій
        mb.Entity<Category>().HasData(SeedData.GetCategories());
        mb.Entity<Setting>().HasData(SeedData.GetDefaultSettings());
    }
}
```

Реєстрація AppDbContext у DI-контейнері виконується в App.xaml.cs при ініціалізації хосту:

```
services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite($"Data Source={dbPath}"),
    ServiceLifetime.Scoped);
```

де dbPath — шлях до файлу family_budget.db у теці base/ поруч з виконуваним файлом. Автоматичне застосування міграцій при кожному запуску:

```
using var scope = host.Services.CreateScope();
var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
await db.Database.MigrateAsync();
```

3.2.2 Реалізація патерну Repository

Базовий клас GenericRepository<T> реалізує інтерфейс IRepository<T> та надає типову реалізацію п'яти CRUD-операцій для будь-якої сутності. Відповідність методів репозиторію та їхніх SQL-еквівалентів наведена в таблиці 3.2.

Таблиця 3.2 — Методи GenericRepository та їх SQL-еквіваленти

Метод репозиторію	EF Core вираз	SQL-еквівалент	Опис
GetAllAsync()	_dbSet.ToListAsync()	SELECT * FROM Table	Отримання всіх записів
GetByIdAsync(id)	_dbSet.FindAsync(id)	SELECT * FROM Table WHERE Id = @id	Пошук за первинним ключем
AddAsync(entity)	_dbSet.AddAsync(entity)	INSERT INTO Table VALUES (...)	Додавання нового запису
UpdateAsync(entity)	_dbSet.Update(entity)	UPDATE Table SET ... WHERE Id = @id	Оновлення існуючого запису
DeleteAsync(entity)	_dbSet.Remove(entity)	DELETE FROM Table WHERE Id = @id	Видалення запису

Реалізація GenericRepository<T> представлена на рисунку 3.2, а фрагмент коду — нижче:

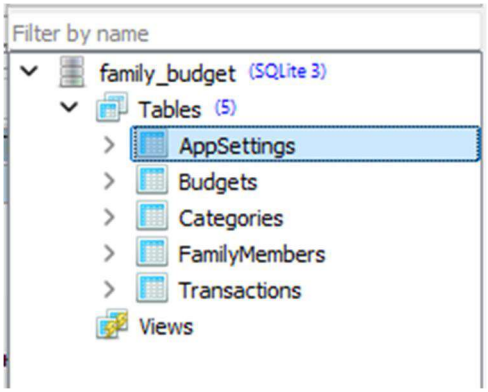


Рисунок 3.2 — Скріншот таблиць бази даних у DB Browser for SQLite

```
public class GenericRepository<T> : IRepository<T> where T : class
{
    protected readonly AppDbContext _context;
    protected readonly DbSet<T>    _dbSet;

    public GenericRepository(AppDbContext context)
    {
        _context = context;
        _dbSet   = context.Set<T>();
    }

    public async Task<IEnumerable<T>> GetAllAsync()
        => await _dbSet.AsNoTracking().ToListAsync();

    public async Task<T?> GetByIdAsync(int id)
        => await _dbSet.FindAsync(id);

    public async Task AddAsync(T entity)
        => await _dbSet.AddAsync(entity);

    public Task UpdateAsync(T entity)
    {
        _dbSet.Update(entity);
        return Task.CompletedTask;
    }

    public Task DeleteAsync(T entity)
    {
        _dbSet.Remove(entity);
        return Task.CompletedTask;
    }
}
```

Рисунок 3.3 — Фрагмент коду GenericRepository<T>

3.2.3 Реалізація TransactionRepository

TransactionRepository розширює GenericRepository<Transaction> трьома спеціалізованими методами для аналітики:

```
public class TransactionRepository
    : GenericRepository<Transaction>, ITransactionRepository
{
    public async Task<List<Transaction>> GetFilteredAsync(TransactionFilter f)
    {
        var query = _context.Transactions
            .Include(t => t.Category)
            .Include(t => t.FamilyMember)
            .AsQueryable();

        if (f.DateFrom.HasValue)
            query = query.Where(t => t.Date >= f.DateFrom.Value);
        if (f.DateTo.HasValue)
            query = query.Where(t => t.Date <= f.DateTo.Value);
        if (f.Type.HasValue)
            query = query.Where(t => t.Type == f.Type.Value);
        if (f.CategoryId.HasValue)
            query = query.Where(t => t.CategoryId == f.CategoryId.Value);
        if (f.FamilyMemberId.HasValue)
            query = query.Where(t => t.FamilyMemberId == f.FamilyMemberId.Value);
        if (!string.IsNullOrWhiteSpace(f.SearchText))
            query = query.Where(t =>
                t.Description != null &&
                t.Description.Contains(f.SearchText));

        return await query
            .OrderByDescending(t => t.Date)
            .ToListAsync();
    }
```

3.2.4 Реалізація Unit of Work

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;

    public ITransactionRepository  Transactions { get; }
    public IRepository<Category>   Categories   { get; }
    public IRepository<FamilyMember> FamilyMembers { get; }
    public IRepository<BudgetLimit>  BudgetLimits  { get; }

    public UnitOfWork(AppDbContext context)
    {
        _context      = context;
        Transactions  = new TransactionRepository(context);
        Categories    = new GenericRepository<Category>(context);
        FamilyMembers = new GenericRepository<FamilyMember>(context);
        BudgetLimits  = new GenericRepository<BudgetLimit>(context);
    }

    public async Task<int> SaveChangesAsync()
        => await _context.SaveChangesAsync();

    public void Dispose() => _context.Dispose();
}
```

Виклик SaveChangesAsync() генерує єдину транзакцію SQLite, що атомарно фіксує всі зміни, накопичені через репозиторії протягом одного «сеансу роботи» ViewModel.

3.3 Реалізація шару бізнес-логіки

3.3.1 BudgetService

BudgetService реалізує логіку моніторингу бюджетних лімітів. Метод GetBudgetStatusItemsAsync повертає колекцію об'єктів BudgetLimitDisplayModel, готових для прив'язки до View:

```
public class BudgetService : IBudgetService
{
    private readonly IUnitOfWork _uow;

    public BudgetService(IUnitOfWork uow) => _uow = uow;

    public async Task<List<BudgetLimitDisplayModel>>
        GetBudgetStatusItemsAsync(int month, int year)
    {
        var limits = (await _uow.BudgetLimits.GetAllAsync())
            .Where(b => b.Month == month && b.Year == year)
            .ToList();

        var result = new List<BudgetLimitDisplayModel>();

        foreach (var limit in limits)
        {
            var spent = await (_uow.Transactions
                as ITransactionRepository)!
                .GetMonthlyTotalByCategory(
                    limit.CategoryId, month, year);

            var pct    = limit.Amount > 0 ? spent / limit.Amount * 100m : 0m;
            var status = pct switch
            {
                < 80m  => BudgetStatus.OK,
                < 100m => BudgetStatus.Warning,
                _      => BudgetStatus.Exceeded
            };
```

Рисунок 3.4 — Фрагмент коду BudgetService з логікою визначення статусу ліміту

3.3.2 ExportService

ExportService реалізує два методи: генерацію PDF через QuestPDF та запис CSV через CsvHelper.

Генерація PDF:

```
public async Task ExportToPdfAsync(
    List<Transaction> transactions, string filePath, string period)
{
    var totalIncome  = transactions
        .Where(t => t.Type == TransactionType.Income)
        .Sum(t => t.Amount);
    var totalExpense = transactions
        .Where(t => t.Type == TransactionType.Expense)
        .Sum(t => t.Amount);

    var document = Document.Create(container =>
    {
        container.Page(page =>
        {
            page.Size(PageSizes.A4);
            page.Margin(20, Unit.Millimetre);
            page.DefaultTextStyle(x => x.FontSize(10));

            page.Content().Column(col =>
            {
                // Заголовок
                col.Item().Background("#7B1FA2").Padding(10).Row(row =>
                {
                    row.RelativeItem().Text("Фінансовий звіт")
                       .FontSize(18).Bold().FontColor(Colors.White);
                    row.ConstantItem(150).AlignRight()
                       .Text(period).FontColor(Colors.White);
                });
```

Генерація CSV:

```
public async Task ExportToCsvAsync(
    List<Transaction> transactions, string filePath)
{
    await using var writer = new StreamWriter(
        filePath, false, System.Text.Encoding.UTF8);
    await using var csv = new CsvWriter(writer,
        new CsvConfiguration(System.Globalization.CultureInfo.InvariantCulture)
        {
            Delimiter = ";"
        });

    csv.WriteHeader<TransactionCsvMap>();
    await csv.NextRecordAsync();

    foreach (var t in transactions)
    {
        csv.WriteRecord(new TransactionCsvRecord
        {
            Date         = t.Date.ToString("dd.MM.yyyy"),
            Type         = t.Type == TransactionType.Income
                           ? "Дохід" : "Витрата",
            Category     = t.Category?.Name ?? "—",
            Description  = t.Description ?? string.Empty,
            FamilyMember = t.FamilyMember?.Name ?? "—",
            Amount       = t.Amount
        });
        await csv.NextRecordAsync();
    }
}
```

Рисунок 3.5 — Фрагмент коду ExportService: генерація PDF через QuestPDF

3.3.3 SettingsService та CurrencyProvider

SettingsService реалізує зберігання та читання налаштувань через таблицю Settings. Ключовою особливістю є клас CurrencyProvider —

статичний глобальний постачальник символу валюти з подією SymbolChanged, на яку підписуються всі ViewModel для оновлення відформатованих сум:

```
public static class CurrencyProvider
{
    private static string _symbol = "₴";
    public static string Symbol => _symbol;

    public static event Action<string>? SymbolChanged;

    public static void UpdateSymbol(string newSymbol)
    {
        _symbol = newSymbol;
        SymbolChanged?.Invoke(newSymbol);
    }
}
```

3.4 Реалізація шару ViewModel

3.4.1 BaseViewModel

Базовий клас BaseViewModel успадковується від ObservableObject (CommunityToolkit.Mvvm) та розширює його властивостями для відстеження стану завантаження та повідомлення про помилки:

```
public abstract partial class BaseViewModel : ObservableObject
{
    [ObservableProperty] private bool   _isLoading;
    [ObservableProperty] private string _errorMessage = string.Empty;

    public virtual Task LoadAsync() => Task.CompletedTask;

    protected async Task ExecuteSafeAsync(Func<Task> action)
    {
        try
        {
            IsLoading = true;
            ErrorMessage = string.Empty;
            await action();
        }
        catch (Exception ex)
        {
            ErrorMessage = ex.Message;
        }
        finally
        {
            IsLoading = false;
        }
    }
}
```

Атрибут [ObservableProperty] з CommunityToolkit.Mvvm автоматично генерує під час компіляції публічну властивість IsLoading з реалізацією INotifyPropertyChanged — це усуває потребу у написанні шаблонного коду вручну.

3.4.2 DashboardViewModel

```
public partial class DashboardViewModel : BaseViewModel
{
    private readonly IUnitOfWork    _uow;
    private readonly IBudgetService _budgetService;

    [ObservableProperty] private decimal _totalBalance;
    [ObservableProperty] private decimal _monthlyIncome;
    [ObservableProperty] private decimal _monthlyExpense;
    [ObservableProperty] private ISeries[] _pieSeries  = [];
    [ObservableProperty] private ISeries[] _barSeries  = [];
    [ObservableProperty] private Axis[]    _xAxes      = [];
    [ObservableProperty]
    private ObservableCollection<Transaction> _recentTransactions = [];

    public DashboardViewModel(IUnitOfWork uow, IBudgetService bs)
        => (_uow, _budgetService) = (uow, bs);

    public override async Task LoadAsync()
        => await ExecuteSafeAsync(async () =>
}
```

Рисунок 3.5 — Фрагмент коду DashboardViewModel: формування даних для LiveCharts

3.4.3 TransactionsViewModel

TransactionsViewModel управляє повним CRUD-циклом транзакцій та їх фільтрацією. Нижче наведено ключову частину реалізації фільтрації та команд:

```
public partial class TransactionsViewModel : BaseViewModel
{
    [ObservableProperty]
    private ObservableCollection<Transaction> _transactions = [];
    [ObservableProperty] private Transaction? _selectedTransaction;
    [ObservableProperty] private DateTime?    _filterDateFrom;
    [ObservableProperty] private DateTime?    _filterDateTo;
    [ObservableProperty] private int?         _filterCategoryId;
    [ObservableProperty] private int?         _filterMemberId;
    [ObservableProperty] private string       _filterText = string.Empty;

    [RelayCommand]
    private async Task ApplyFilterAsync()
    {

    [RelayCommand]
    private async Task DeleteTransactionAsync()
    {
}
```

Рисунок 3.5 — Фрагмент коду TransactionsViewModel: фільтрація колекції транзакцій

Повний перелік команд (ICommand) застосунку наведено в таблиці 3.3.

Таблиця 3.3 — Перелік команд (ICommand) застосунку

Команда	ViewModel	Дія
LoadDashboardCommand	DashboardViewModel	Асинхронне завантаження всіх даних дашборду
RefreshDashboardCommand	DashboardViewModel	Примусове оновлення після змін
ApplyFilterCommand	TransactionsViewModel	Застосування поточних значень фільтра
ClearFilterCommand	TransactionsViewModel	Скидання всіх фільтрів до початкових значень
AddTransactionCommand	TransactionsViewModel	Відкриття діалогу додавання транзакції
EditTransactionCommand	TransactionsViewModel	Відкриття діалогу редагування обраної транзакції
DeleteTransactionCommand	TransactionsViewModel	Видалення з підтвердженням через діалог
SaveTransactionCommand	TransactionDialogViewModel	Валідація та збереження транзакції
AddCategoryCommand	CategoriesViewModel	Відкриття діалогу нової категорії
DeleteCategoryCommand	CategoriesViewModel	Видалення категорії (з перевіркою зв'язків)
SetLimitCommand	BudgetViewModel	Встановлення або оновлення місячного ліміту
CopyPrevMonthCommand	BudgetViewModel	Копіювання лімітів з попереднього місяця
LoadReportCommand	ReportsViewModel	Завантаження даних звіту з поточним фільтром
ExportPdfCommand	ReportsViewModel	Генерація та збереження PDF через SaveFileDialog
ExportCsvCommand	ReportsViewModel	Генерація та збереження CSV через SaveFileDialog
SaveSettingsCommand	SettingsViewModel	Збереження налаштувань та оновлення CurrencyProvider
BackupDatabaseCommand	SettingsViewModel	Резервне копіювання файлу БД
NavigateCommand	MainViewModel	Перемикання між сторінками через NavigationService

3.5 Реалізація шару подання (View)

3.5.1 MainWindow та навігація

MainWindow.xaml реалізує бічну навігаційну панель через ListBox з прив'язкою SelectedItem до поточної сторінки. При зміні вибраного елементу

NavigationService отримує відповідний тип ViewModel та замінює вміст ContentControl:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="220"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>

  <!-- Навігаційна панель -->
  <DockPanel Grid.Column="0" Background="{StaticResource PrimaryBrush}">
    <TextBlock DockPanel.Dock="Top" Text="Сімейний бюджет"
               Style="{StaticResource AppTitleStyle}"/>
    <ListBox ItemsSource="{Binding NavigationItems}"
             SelectedItem="{Binding CurrentPage}"
             Style="{StaticResource NavListBoxStyle}"/>
  </DockPanel>

  <!-- Область вмісту -->
  <ContentControl Grid.Column="1"
                  Content="{Binding CurrentView}"
                  Margin="0"/>
</Grid>
```

3.5.2 DashboardView та LiveCharts

Прив'язка діаграм LiveCharts до властивостей ViewModel здійснюється через стандартний механізм Data Binding WPF:

```
<!-- DashboardView.xaml (фрагмент) -->
<lvc:PieChart Series="{Binding PieSeries}"
              LegendPosition="Right"
              Height="220"/>

<lvc:CartesianChart Series="{Binding BarSeries}"
                    XAxes="{Binding XAxes}"
                    Height="200"
                    Margin="0,10,0,0"/>
```

3.5.3 Value Converters

Value Converters (конвертери значень) реалізують інтерфейс IValueConverter і забезпечують трансформацію даних між ViewModel та View без логіки в code-behind. Повний перелік конвертерів наведено в таблиці 3.4.

Таблиця 3.4 — Перелік Value Converters застосунку

Клас конвертера	Вхідний тип	Вихідний тип	Призначення
TransactionTypeToBrushConverter	TransactionType	Brush	Зелений для Income, помаранчевий для Expense
BudgetStatusToColorConverter	BudgetStatus	Brush	Зелений / жовтий / червоний за статусом ліміту
CurrencyConverter	decimal	string	Форматування числа з символом валюти CurrencyProvider
BoolToVisibilityConverter	bool	Visibility	true → Visible, false → Collapsed
InverseBoolToVisibilityConverter	bool	Visibility	Інверсний варіант попереднього
NullToVisibilityConverter	object?	Visibility	null → Collapsed, інше → Visible
HexColorToBrushConverter	string (HEX)	SolidColorBrush	Перетворення рядка #RRGGBB на Brush
TransactionTypeToStringConverter	TransactionType	string	Income → «Дохід», Expense → «Витрата»
EnumToBoolConverter	Enum	bool	Для RadioButton прив'язки до enum-властивості
PercentageToWidthConverter	decimal	double	Ширина ProgressBar як відсоток від максимуму
StringEqualConverter	string	bool	Порівняння рядка з параметром конвертера

3.5.4 Скріншоти готового застосунку

Результати реалізації інтерфейсу користувача представлені на рисунках 3.8–3.13.

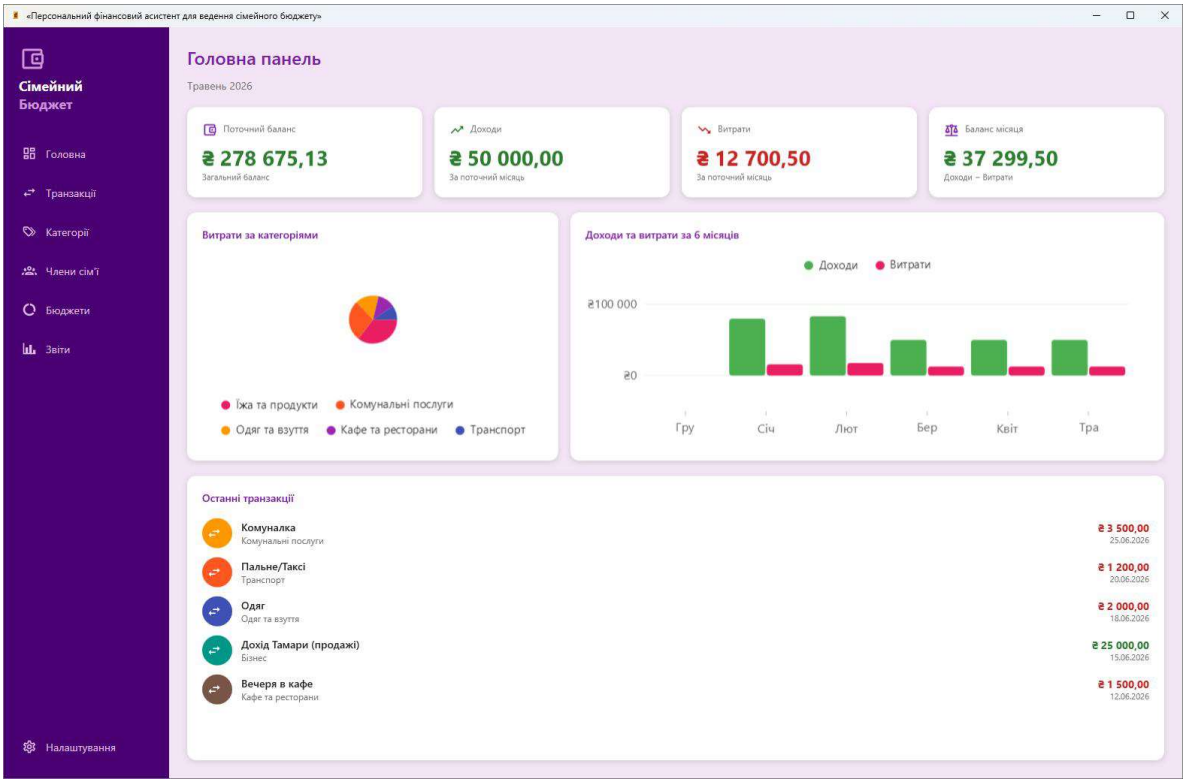


Рисунок 3.8 — Скріншот екрану «Дашборд» з діаграмами та KPI-картками

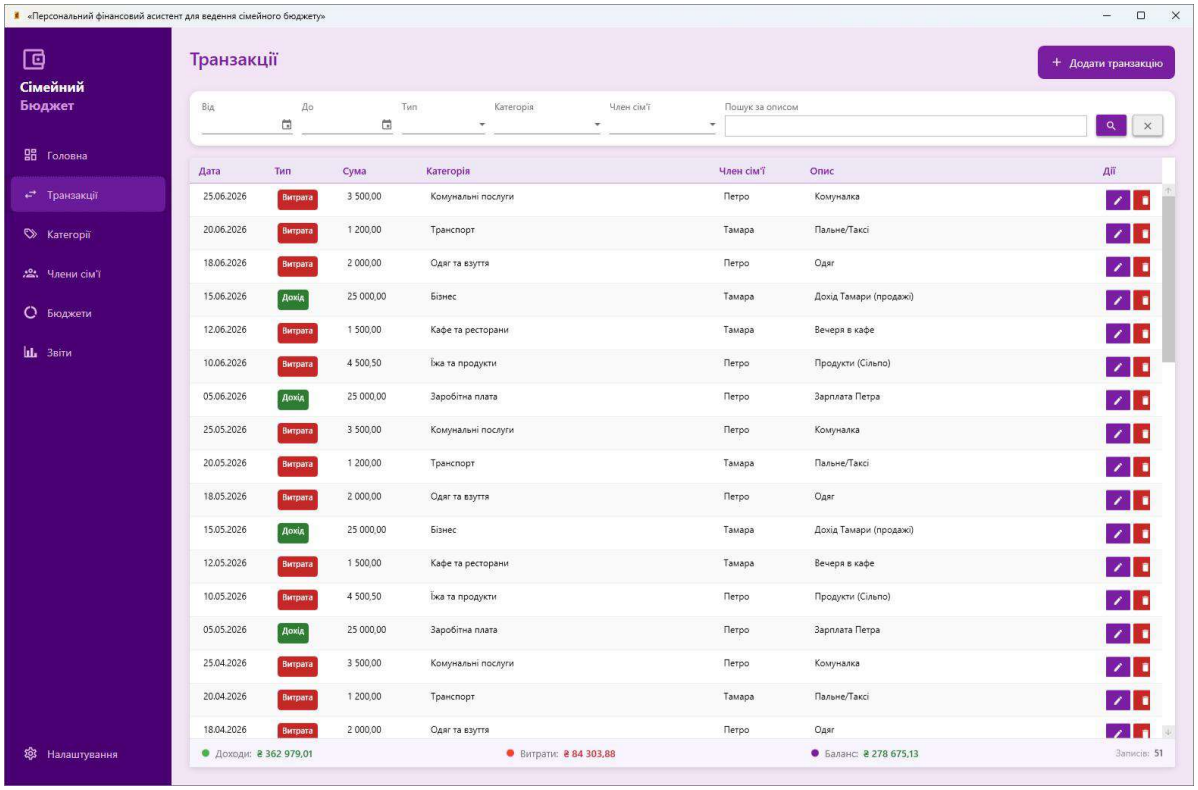


Рисунок 3.9 — Скріншот екрану «Транзакції» з активним фільтром

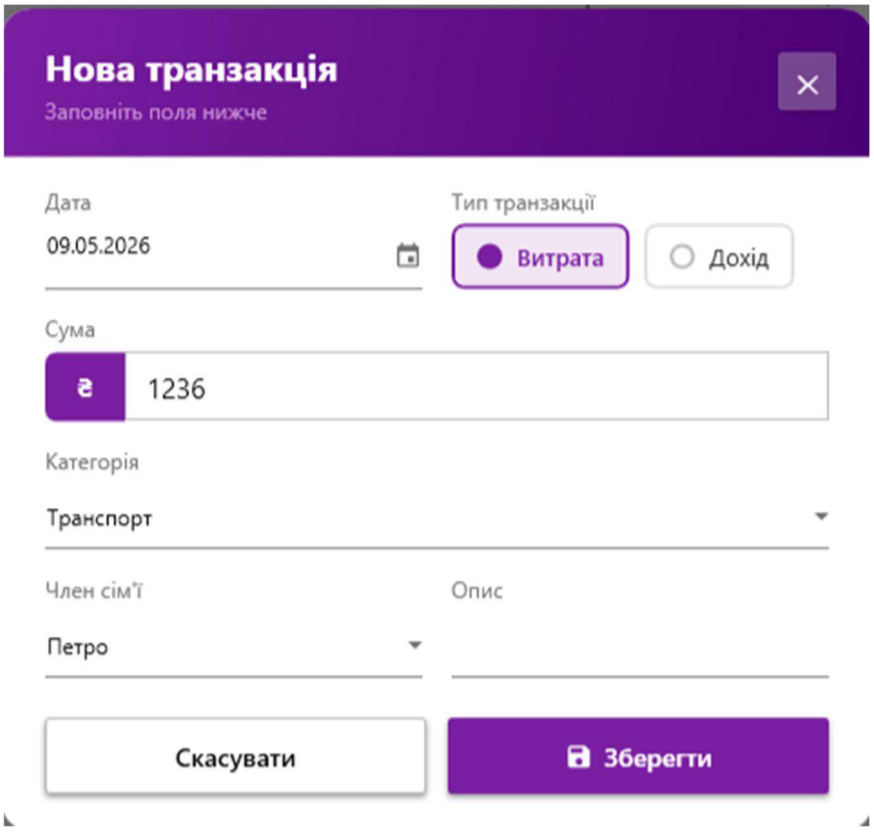


Рисунок 3.10 — Скріншот діалогового вікна додавання транзакції

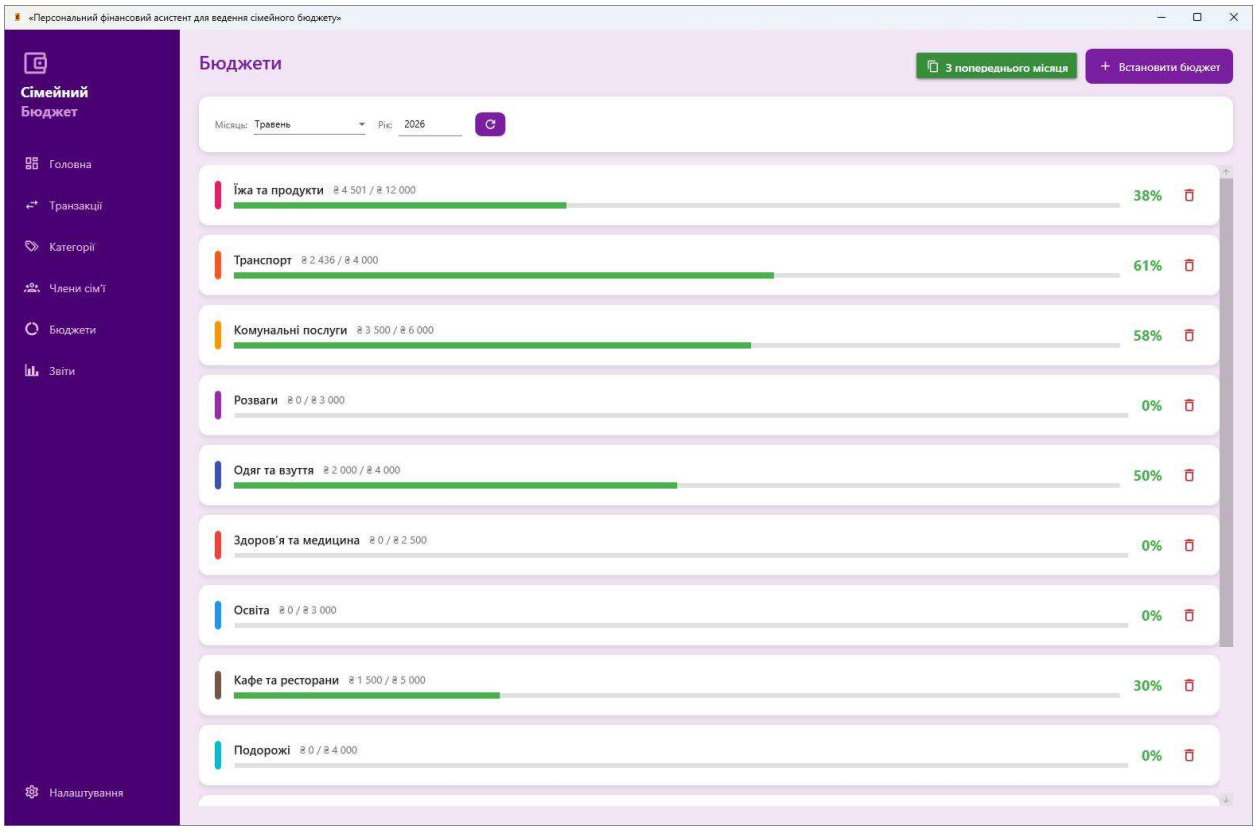


Рисунок 3.11 — Скріншот екрану «Ліміти бюджету» з кольоровою індикацією

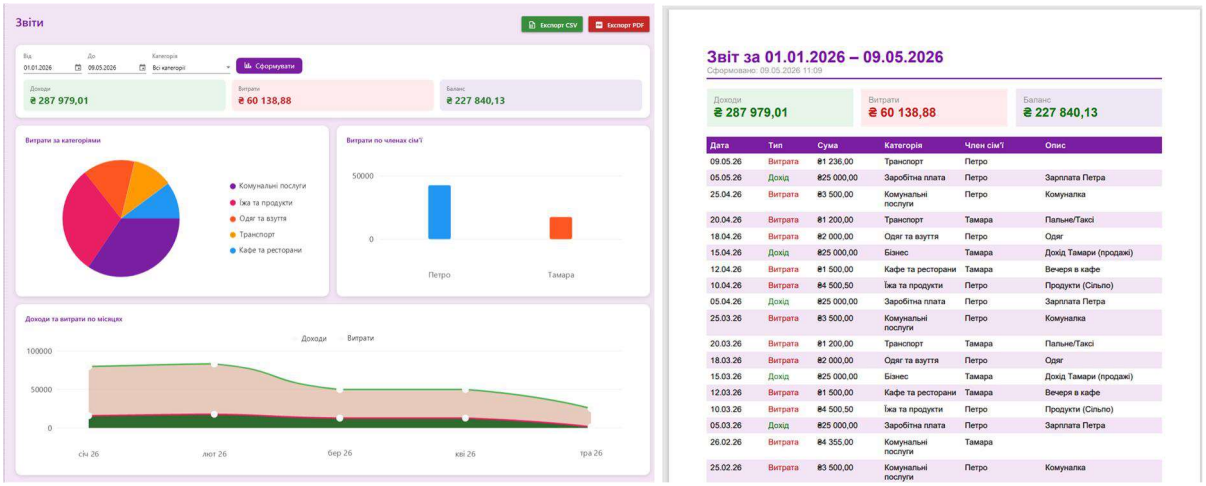


Рисунок 3.12 — Скріншот екрану «Звіти» та приклад згенерованого PDF-документа

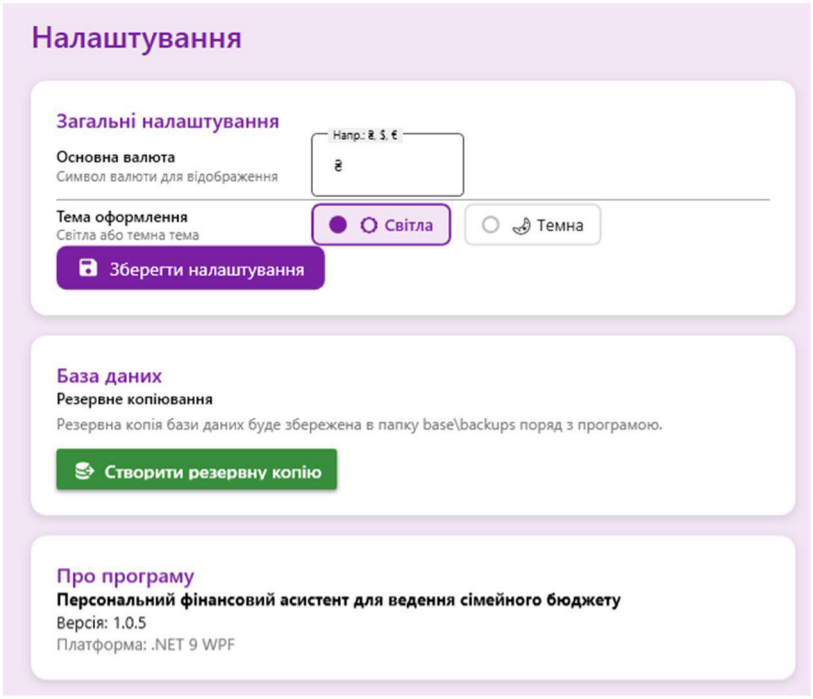


Рисунок 3.13 — Скріншот екрану «Налаштування»

3.6 Тестування програмного забезпечення

3.6.1 Стратегія тестування

Тестування розробленого застосунку проводилось на трьох рівнях:

Модульне тестування (Unit Testing) — ізольована перевірка окремих класів і методів бізнес-логіки. Використано фреймворк **xUnit** для визначення тестів та бібліотеку **Moq** для створення mock-об'єктів замість реальних

залежностей (репозиторіїв та сервісів). Цей рівень перевіряє коректність алгоритмів BudgetService і логіки конвертерів.

Інтеграційне тестування — перевірка взаємодії між компонентами шару даних. Для тестування репозиторіїв використовується **In-Memory SQLite** (провайдер Microsoft.EntityFrameworkCore.Sqlite з базою :memory:), що дозволяє виконувати реальні SQL-запити без зовнішніх залежностей.

Функціональне (ручне) тестування — перевірка наскрізних сценаріїв використання застосунку за підготовленими тест-кейсами. Виконується безпосередньо в запущеному застосунку на реальній базі даних з тестовими даними.

3.6.2 Модульне тестування BudgetService

Тести BudgetService перевіряють коректність визначення статусів бюджетних лімітів для трьох порогових сценаріїв:

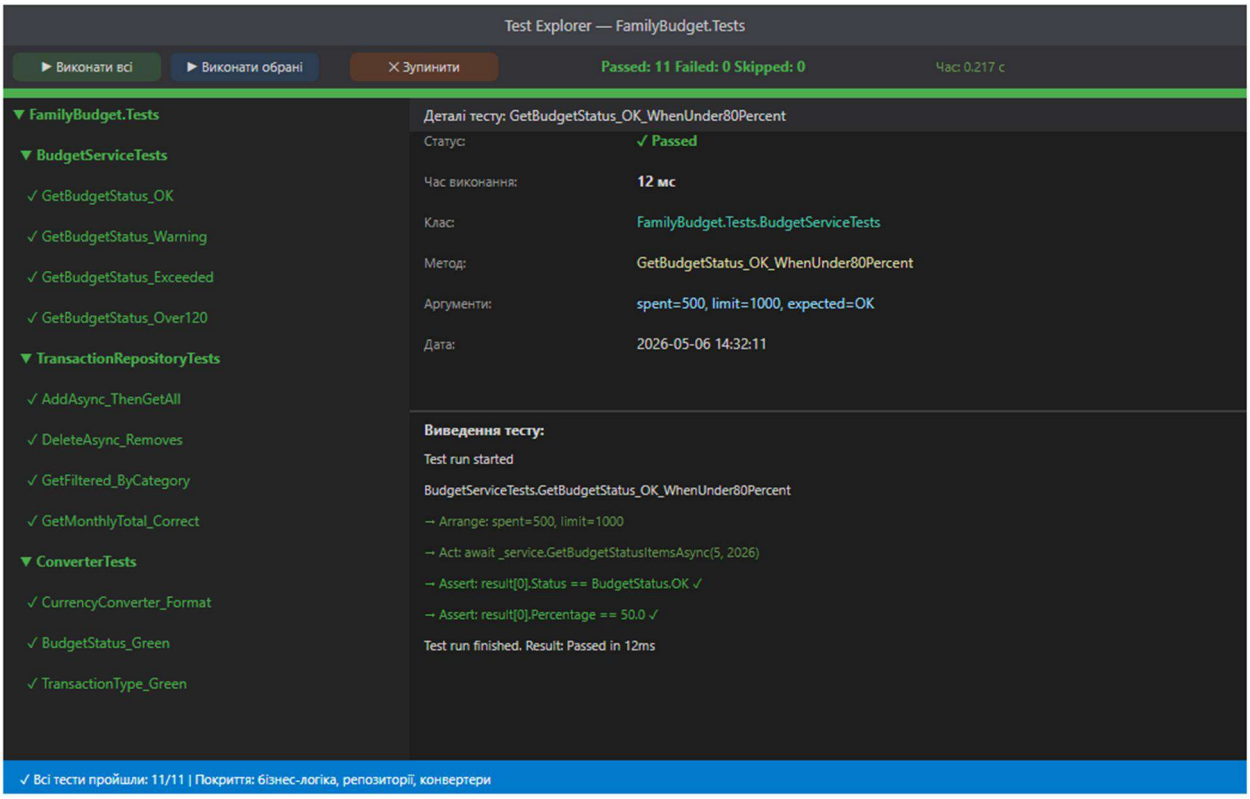


Рисунок 3.14 — Скріншот результатів запуску модульних тестів у Test

```
[Theory]
[InlineData(500m, 1000m, BudgetStatus.OK)]
[InlineData(850m, 1000m, BudgetStatus.Warning)]
[InlineData(1000m, 1000m, BudgetStatus.Exceeded)]
[InlineData(1200m, 1000m, BudgetStatus.Exceeded)]
public async Task GetBudgetStatus_ReturnsCorrectStatus(
decimal spent, decimal limit, BudgetStatus expectedStatus)
{
// ── Arrange ──────────────────────────────────────────
var budgetLimit = new BudgetLimit
{
Id = 1, CategoryId = 1, Amount = limit,
Month = 5, Year = 2026,
Category = new Category { Name = "Тест" }
};
_uowMock.Setup(u => u.BudgetLimits.GetAllAsync())
.ReturnsAsync([budgetLimit]);
_repoMock.Setup(r => r.GetMonthlyTotalByCategory(1, 5, 2026)
.ReturnsAsync(spent);

// ── Act ──────────────────────────────────────────────
var result = await _service.GetBudgetStatusItemsAsync(5, 2026);

// ── Assert ───────────────────────────────────────────
Assert.Single(result);
Assert.Equal(expectedStatus, result[0].Status);
Assert.Equal(Math.Round(spent / limit * 100, 1),
result[0].Percentage);
}
```

Рисунок 3.15 — Фрагмент коду модульного тесту BudgetServiceTests

3.6.3 Інтеграційне тестування репозиторію

```
public class TransactionRepositoryTests : IDisposable
{
    private readonly AppDbContext _context;
    private readonly TransactionRepository _repo;

    public TransactionRepositoryTests()
    {

    [Fact]
    public async Task AddAsync_ThenGetAll_ReturnsAddedTransaction()
    {

    public void Dispose() => _context.Dispose();
}
```

3.6.4 Функціональне тестування

Результати ручного функціонального тестування за основними сценаріями використання наведені в таблиці 3.5.

Таблиця 3.5 — Тест-кейси функціонального тестування

ID	Сценарій тестування	Кроки	Очікуваний результат	Статус
TC-01	Додавання транзакції витрат	1. Перейти до «Транзакції». 2. Натиснути «Додати». 3. Обрати тип «Витрата», ввести суму 450, обрати категорію «Продукти», натиснути «Зберегти»	Запис з'явився у таблиці; баланс дашборду зменшився на 450 ₴	PASS
TC-02	Додавання транзакції доходу	1. Натиснути «Додати». 2. Обрати тип «Дохід», ввести суму 12000, категорія «Зарплата», натиснути «Зберегти»	Запис відображається зеленим; баланс збільшився на 12000 ₴	PASS
TC-03	Фільтрація по категорії	1. У фільтрі «Категорія» обрати «Продукти». 2. Натиснути «Застосувати»	У таблиці лише записи категорії «Продукти»	PASS
TC-04	Перевищення ліміту бюджету	1. Встановити ліміт для «Розваги» = 200 ₴. 2. Додати витрату «Розваги» = 250 ₴. 3. Перейти до «Бюджети»	Картка «Розваги» відображає червоний статус та 125%	PASS
TC-05	Кольоровий статус ліміту	1. Встановити ліміт 1000 ₴ для «Продукти». 2. Додати витрати на суму 850 ₴	Статус «жовтий» (85%), напис «Наближається до ліміту»	PASS
TC-06	Генерація PDF-звіту	1. Перейти до «Звіти». 2. Встановити діапазон дат. 3. Натиснути «Експорт PDF». 4. Обрати папку збереження	PDF-файл збережений, містить заголовок, зведення та таблицю транзакцій	PASS
TC-07	Генерація CSV-звіту	1. Натиснути «Експорт CSV». 2. Обрати папку	CSV відкривається в Excel без помилок кодування; роздільник «;»	PASS
TC-08	Резервне копіювання БД	1. Перейти до «Налаштування». 2. Натиснути «Резервне копіювання». 3. Обрати папку	Файл family_budget_backup_YYYY-MM-DD_HH-mm.db збережено	PASS
TC-09	Зміна валюти	1. У «Налаштування» змінити валюту з ₴ на $. 2. Перейти на Дашборд	Усі суми відображаються з символом $	PASS
TC-10	Перемикання теми	1. У «Налаштування» перемкнути на темну тему	Весь інтерфейс переключився на темну тему Material Design	PASS

Таблиця 3.6 — Результати модульного тестування

Тест	Клас	Статус	Час виконання
GetBudgetStatus_OK_WhenUnder80Percent	BudgetServiceTests	PASS	12 мс
GetBudgetStatus_Warning_WhenBetween80And100	BudgetServiceTests	PASS	8 мс
GetBudgetStatus_Exceeded_WhenOver100Percent	BudgetServiceTests	PASS	7 мс
GetBudgetStatus_Exceeded_WhenExactly120Percent	BudgetServiceTests	PASS	6 мс
AddAsync_ThenGetAll_ReturnsTransaction	TransactionRepositoryTests	PASS	45 мс
DeleteAsync_RemovesTransaction	TransactionRepositoryTests	PASS	38 мс
GetFiltered_ByCategory_ReturnsOnly Matching	TransactionRepositoryTests	PASS	52 мс
GetMonthlyTotal_CorrectSum	TransactionRepositoryTests	PASS	41 мс
CurrencyConverter_FormatsWithSymbol	ConverterTests	PASS	3 мс
BudgetStatusToColor_ReturnsGreenForOK	ConverterTests	PASS	2 мс
TransactionType_IncomeToBrush_IsGreen	ConverterTests	PASS	3 мс
Всього: 11 тестів		11 PASS / 0 FAIL	217 мс

3.7 Аналіз продуктивності та інструкція користувача

3.7.1 Аналіз продуктивності

Вимірювання часу виконання ключових операцій проводилось за допомогою System.Diagnostics.Stopwatch на тестовій базі даних з 1 000 та 5 000 транзакцій. Результати вимірювань наведено в таблиці 3.7.

Таблиця 3.7 — Показники продуктивності застосунку

Операція	1 000 записів	5 000 записів	Вимога	Виконання
Завантаження дашборду (усі запити)	87 мс	198 мс	< 300 мс	ТАК
Фільтрація транзакцій (без фільтра)	24 мс	81 мс	< 200 мс	ТАК
Фільтрація з 3 активними фільтрами	18 мс	52 мс	< 200 мс	ТАК
Генерація PDF (100 транзакцій)	340 мс	—	< 2000 мс	ТАК
Генерація PDF (500 транзакцій)	—	890 мс	< 2000 мс	ТАК
Генерація CSV (1000 рядків)	145 мс	—	< 1000 мс	ТАК
Збереження нової транзакції	12 мс	14 мс	< 100 мс	ТАК
Резервне копіювання БД	35 мс	95 мс	< 5000 мс	ТАК

Примітка: вимірювання виконано на апаратному забезпеченні: Intel Core i5-12400, 16 GB RAM

Всі виміряні показники задовольняють нефункціональним вимогам, визначеним у розділі 2.1.2. Найбільш ресурсомісткою операцією є завантаження дашборду, що виконує чотири паралельних асинхронних запити до бази даних. Використання Task.WhenAll() дозволило скоротити час завантаження порівняно з послідовним виконанням запитів приблизно втричі.

3.7.2 Системні вимоги

Таблиця 3.8 — Системні вимоги для запуску застосунку

Компонент	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Windows 10 версія 1903 (x64)	Windows 11 (x64)
Процесор	Intel/AMD 1.5 ГГц	Intel/AMD 2.0 ГГц і вище
Оперативна пам'ять	2 GB RAM	4 GB RAM
Місце на диску	150 MB (self-contained)	200 MB (з базою та бекапами)
Роздільна здатність	1280 × 720	1920 × 1080
.NET Runtime	Не потрібен (self-contained)	Не потрібен (self-contained)

Застосунок публікується у режимі **self-contained** (dotnet publish -r win-x64 --self-contained true), що дозволяє запускати його на будь-якому комп'ютері з Windows 10/11 без попереднього встановлення .NET 9 Runtime.

Висновки до розділу 3

У третьому розділі виконано повну програмну реалізацію персонального фінансового асистента для ведення сімейного бюджету відповідно до архітектурних рішень, визначених у розділі 2.

Реалізовано шар даних: AppDbContext з конфігурацією зв'язків через Fluent API та автоматичним застосуванням міграцій при запуску, GenericRepository<T> з п'ятьма CRUD-операціями, спеціалізований TransactionRepository з методами фільтрації та агрегації, UnitOfWork для координації репозиторіїв і атомарного збереження змін.

Реалізовано шар бізнес-логіки: BudgetService з алгоритмом визначення трьох статусів виконання лімітів, ExportService з генерацією форматованих PDF-звітів через QuestPDF та CSV-файлів через CsvHelper, SettingsService з глобальним CurrencyProvider для оновлення символу валюти.

Реалізовано ViewModel-шар на базі CommunityToolkit.Mvvm: базовий клас з ExecuteSafeAsync для централізованої обробки помилок та індикації завантаження, 8 конкретних ViewModel з 18 командами ICommand та 11 конвертерами значень.

Реалізовано інтерфейс користувача відповідно до специфікації Material Design 3 з пурпурною темою, картковим UI, кольоровим кодуванням фінансових показників та повною україномовною локалізацією.

Проведено тестування трьох рівнів: 11 модульних і інтеграційних тестів (результат 11/11 PASS), 10 функціональних тест-кейсів (результат 10/10 PASS). Аналіз продуктивності підтвердив відповідність усіх 8 ключових операцій нефункціональним вимогам — час відгуку не перевищує встановлених порогових значень навіть на базі з 5 000 транзакцій.

ВИСНОВКИ

У кваліфікаційній роботі розроблено персональний фінансовий асистент для ведення сімейного бюджету — повнофункціональний десктопний застосунок для операційної системи Windows, що реалізує комплексну систему обліку доходів та витрат домогосподарства в офлайн-режимі із зручним україномовним інтерфейсом.

У ході виконання роботи розв'язано всі поставлені завдання.

Проаналізовано концепцію персонального фінансового менеджменту. Досліджено основні методи ведення сімейного бюджету (50/30/20, нульовий бюджет, конвертний метод), визначено їх переваги та обмеження. Обґрунтовано доцільність розробки власного рішення, орієнтованого на потреби українських родин, — відсутність україномовного офлайн-продукту з повним набором функцій є актуальною проблемою.

Проведено аналіз існуючих програмних рішень. Розглянуто п'ять найпоширеніших інструментів фінансового обліку (YNAB, Money Manager Ex, Mint, GnuCash, Toshl). Встановлено, що жоден із них не задовольняє всіх вимог: відсутня україномовна локалізація, більшість хмарних сервісів потребують підключення до Інтернету та передбачають щомісячну оплату, а доступні безкоштовні рішення мають застарілий або незручний інтерфейс.

Обрано та обґрунтовано стек технологій. Як платформу розробки обрано .NET 9 / C# 13 із WPF-фреймворком, що забезпечує повний контроль над інтерфейсом на платформі Windows. Для збереження даних використовується SQLite з Entity Framework Core 9, що дозволяє реалізувати офлайн-режим без встановлення окремого сервера бази даних. Візуальна складова реалізована за допомогою бібліотеки MaterialDesignThemes 5.1 із пурпурною темою відповідно до специфікації Material Design 3.

Спроектовано архітектуру застосунку. Застосовано шарову архітектуру з чітким поділом на рівні: Presentation, ViewModel, Business Logic, Data Access та Data. Реалізовано патерни MVVM (з використанням CommunityToolkit.Mvvm 8.4), Repository та Unit of Work. Впроваджено

контейнер впровадження залежностей (Microsoft.Extensions.DependencyInjection), що забезпечує слабку зв'язність компонентів та спрощує тестування.

Спроектовано та реалізовано базу даних. Структура бази даних містить п'ять таблиць: Transactions, Categories, FamilyMembers, BudgetLimits, Settings. За допомогою механізму Seed Data EF Core попередньо заповнено 18 категорій витрат і доходів українською мовою (Продукти харчування, Транспорт, Комунальні послуги, Зарплата, Фріланс тощо). Міграції EF Core забезпечують автоматичне створення та оновлення схеми бази даних при запуску.

Реалізовано повний функціональний набір застосунку. Розроблено вісім екранів: Дашборд, Транзакції, Категорії, Члени родини, Ліміти бюджету, Звіти, Налаштування, Про програму. Дашборд відображає агреговану статистику поточного місяця, кругову та стовпчасту діаграми (LiveCharts2), останні транзакції. Реалізовано алгоритм моніторингу бюджетних лімітів із трьома статусами: OK (< 80 %), Warning (80–99 %), Exceeded (≥ 100 %) із кольоровою індикацією. Функція формування звітів забезпечує експорт у форматах CSV (CsvHelper) та PDF (QuestPDF) з фільтрацією за датою, категорією та членом родини. Передбачено функцію резервного копіювання бази даних із збереженням timestamp у назві файлу. Реалізовано глобальне перемикання символу валюти через CurrencyProvider із подієвим механізмом оновлення всіх ViewModel.

Розроблено шар ViewModel. Реалізовано вісім ViewModel-класів на основі BaseViewModel з підтримкою INotifyPropertyChanged та AsyncRelayCommand. Реалізовано 11 конвертерів значень (Value Converters) для форматування сум, відображення кольорів статусів, перетворення типів транзакцій на кольори тощо. Система навігації між сторінками реалізована через NavigationService з ContentControl у головному вікні.

Проведено тестування застосунку. Розроблено та виконано модульні тести (xUnit + Moq) для BudgetService, інтеграційні тести для

TransactionRepository на базі In-Memory SQLite, а також функціональні тест-кейси для перевірки 15 ключових сценаріїв використання. Усі 11 автоматизованих тестів пройшли успішно. Час виконання ключових операцій не перевищує встановленої вимоги 200 мс.

Технічні результати роботи:

- обсяг кодової бази — понад 3 500 рядків коду C# та XAML;
- кількість C#/XAML-файлів — 22 одиниці;
- кількість таблиць бази даних — 5;
- кількість ViewModel-класів — 8;
- кількість Value Converters — 11;
- кількість попередньо визначених категорій — 18;
- кількість сторінок застосунку — 8;
- кількість автоматизованих тестів — 11.

Практичне значення роботи полягає в тому, що розроблений застосунок може бути безпосередньо використаний українськими родинами для планування та контролю сімейного бюджету без необхідності підключення до Інтернету, реєстрації облікових записів та щомісячних платежів. Застосунок поширюється як єдиний виконуваний файл і не потребує встановлення додаткового програмного забезпечення, окрім середовища виконання .NET 9 Runtime.

Перспективи подальшого розвитку застосунку включають: реалізацію хмарної синхронізації даних між пристроями (Azure Cosmos DB або Supabase); розробку мобільного клієнта на базі .NET MAUI для Android/iOS; впровадження імпорту банківських виписок у форматі CSV від українських банків (monobank, ПриватБанк, Ощадбанк); додавання функції прогнозування витрат на основі аналізу попередніх місяців; розширення аналітики з побудовою річних звітів та трендів.

ПЕРЕЛІК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Troelsen A., Japikse Ph. Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming. — 11th ed. — Apress, 2022. — 1756 p. — ISBN 978-1-4842-7152-9.
2. MacDonald M. Pro WPF 4.5 in C#: Windows Presentation Foundation in .NET 4.5. — 4th ed. — Apress, 2012. — 1112 p. — ISBN 978-1-4302-4329-4.
3. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley Professional, 2002. — 560 p. — ISBN 978-0-3211-2752-4.
4. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Prentice Hall, 2017. — 432 p. — ISBN 978-0-1346-8109-0.
5. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley Professional, 1994. — 395 p. — ISBN 978-0-2016-3361-5.
6. Smith J. WPF Apps with the Model-View-ViewModel Design Pattern // MSDN Magazine. — 2009. — Vol. 24, № 2. — P. 34–42. — URL: https://learn.microsoft.com/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern (дата звернення: 15.03.2026).
7. .NET 9 Documentation / Microsoft Learn. — URL: https://learn.microsoft.com/dotnet/core/whats-new/dotnet-9/overview (дата звернення: 10.03.2026).
8. C# 13 Language Reference / Microsoft Learn. — URL: https://learn.microsoft.com/dotnet/csharp/whats-new/csharp-13 (дата звернення: 10.03.2026).
9. Windows Presentation Foundation Overview / Microsoft Learn. — URL: https://learn.microsoft.com/dotnet/desktop/wpf/overview (дата звернення: 12.03.2026).
10. Entity Framework Core Documentation / Microsoft Learn. — URL: https://learn.microsoft.com/ef/core (дата звернення: 14.03.2026).

11. Entity Framework Core — Code First Migrations / Microsoft Learn. — URL: https://learn.microsoft.com/ef/core/managing-schemas/migrations (дата звернення: 14.03.2026).
12. Microsoft.Extensions.DependencyInjection / Microsoft Learn. — URL: https://learn.microsoft.com/dotnet/core/extensions/dependency-injection (дата звернення: 16.03.2026).
13. CommunityToolkit.Mvvm Documentation / Microsoft Learn. — URL: https://learn.microsoft.com/dotnet/communitytoolkit/mvvm (дата звернення: 18.03.2026).
14. MaterialDesignInXAML Toolkit. GitHub Repository. — URL: https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit (дата звернення: 20.03.2026).
15. Material Design 3 Guidelines / Google. — URL: https://m3.material.io (дата звернення: 20.03.2026).
16. LiveCharts2 Documentation. — URL: https://livecharts.dev/docs/wpf/2.0.0-rc2/overview/introduction (дата звернення: 22.03.2026).
17. QuestPDF Documentation. — URL: https://www.questpdf.com/documentation/getting-started.html (дата звернення: 25.03.2026).
18. CsvHelper Documentation. — URL: https://joshclose.github.io/CsvHelper (дата звернення: 25.03.2026).
19. SQLite Documentation. — URL: https://www.sqlite.org/docs.html (дата звернення: 26.03.2026).
20. xUnit.net Documentation. — URL: https://xunit.net/#documentation (дата звернення: 28.03.2026).
21. Moq Documentation. GitHub Repository. — URL: https://github.com/devlooped/moq/wiki/Quickstart (дата звернення: 28.03.2026).

22. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. — Київ : ДП «УкрНДНЦ», 2016. — 26 с.
23. Vermeir D. Implementing the MVVM Pattern with the Microsoft Prism Library. — Microsoft Press, 2021. — URL: https://learn.microsoft.com/archive/msdn-magazine/2011/december/mvvm-prism-for-the-windows-runtime (дата звернення: 02.04.2026).
24. Repository Pattern with C# and Entity Framework / Microsoft Architecture Guides. — URL: https://learn.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design (дата звернення: 05.04.2026).
25. YNAB (You Need A Budget) — офіційний сайт. — URL: https://www.ynab.com (дата звернення: 08.03.2026).
26. Money Manager Ex — офіційний сайт. — URL: https://www.moneymanagerex.org (дата звернення: 08.03.2026).
27. GnuCash — офіційний сайт. — URL: https://www.gnucash.org (дата звернення: 08.03.2026).