МІНІСТЕРСТВО ВНУТРІШНІХ СПРАВ УКРАЇНИ
ЛЬВІВСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ ВНУТРІШНІХ СПРАВ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ УПРАВЛІННЯ, ПСИХОЛОГІЇ ТА БЕЗПЕКИ

Кафедра інформаційних технологій

РОЗРОБКА ДОДАТКУ УПРАВЛІННЯ ОСОБИСТИМИ ФІНАНСАМИ ІЗ ВПРОВАДЖЕННЯМ КОМПОНЕНТНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ

Кваліфікаційна робота
здобувача вищої освіти
4 курсу заочної форми навчання
Катерини АНТОНЮК

Науковий керівник:
доцент, кандидат технічних наук
Андрій ГОЛОВАТИЙ

Рецензент:

_____
вчене звання, науковий ступінь

_____
(Ім'я ПРІЗВИЩЕ рецензента)

Кваліфікаційна робота допущена до захисту
«___» _____ 2026 р., протокол № ____

Завідувач кафедри інформаційних технологій
_____ Олег ЗАЧЕК
(підпис)

Львів
2026

АНОТАЦІЯ

АНТОНЮК К. Розробка додатку управління особистими фінансами із впровадженням компонентно-орієнтованої архітектури. – Рукопис.

Дослідження на здобуття освітнього ступеня «бакалавр» за спеціальністю 126 «Інформаційні системи та технології». – Львівський державний університет внутрішніх справ, МВС України, Львів, 2026.

У кваліфікаційній роботі розглянуто ключові аспекти впровадження компонентно-орієнтованої архітектури при розробці десктопного додатка для управління особистими фінансами. Для реалізації проєкту використано сучасні технології React, Electron, мову програмування TypeScript та платформу Firebase . Проаналізовано предметну область додатка, вивчено існуючі аналоги та визначено основні функціональні можливості системи. Розроблено UML-діаграми варіантів використання, діаграму класів та діаграми послідовності. Створено макети сторінок користувацького інтерфейсу. Реалізовано робочий прототип десктопного додатка, який демонструє основні процеси управління особистими фінансами.

Ключові слова: додаток, React, Electron, TypeScript, додаток управління фінансами.

ABSTRACT

ANTONYUK K. Development of a Desktop Personal Finance Management Application Using Component-Oriented Architecture

Research for obtaining a bachelor’s degree in specialty 126 «Information systems and technologies». – Lviv State University of Internal Affairs, MIA of Ukraine, Lviv, 2026.

This master's thesis explores the key aspects of implementing component-

oriented architecture for building a desktop personal finance management application. The project was developed using modern technologies: React, Electron, TypeScript, and the Firebase platform. The research includes an analysis of the subject domain, a review of existing solutions, and the definition of the system's core functional requirements. UML diagrams (use cases, class diagram, and sequence diagrams) were designed, user interface mockups were created, and a functional prototype of the desktop application was successfully implemented, demonstrating core personal finance management processes.

Keywords: application, React, Electron, TypeScript, financial management application.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API (Application Programming Interface) – прикладний програмний інтерфейс.

GUI (Graphical User Interface) – графічний інтерфейс користувача.

UML (Unified Modeling Language) – уніфікована мова моделювання.

КОА – компонентно-орієнтована архітектура.

MVC (Model-view-controller) – модель-вид-контролер.

DOM (Document Object Model) – об'єктна модель документа.

IDE (Integrated development environment) – інтегроване середовище розробки

ДНАОП – державні нормативні акти з охорони праці.

ПЗ – програмне забезпечення.

ПК – персональний комп'ютер.

ВСТУП

Ми живемо в час швидкого розвитку цифрових технології, які також змінюють суспільство та майже кожну сферу життя. Розробляються велика кількість програмних рішень для управління великими обсягами інформації. Дуже потрібні додатки для управління особистими фінансами, особливо в умовах війни, які допомагають вести облік і аналізувати доходи й витрати користувача, розумно розподіляти фінанси, планувати бюджет і визначати фінансові цілі, а також стежити за їх виконанням.

Додаток для управління особистими фінансами, реалізований у вигляді десктопної програми на базі сучасних веб-технологій, дозволяє користувачам мати повний контроль над своїми фінансами в зручному та безпечному середовищі. Такий програмний продукт є актуальним інструментом як для приватних осіб, так і для сімей, які прагнуть оптимізувати свої фінансові рішення, зменшити імпульсивні витрати та досягати поставлених фінансових цілей.

Необхідність розробки такого додатка обумовлюється швидким зростанням попиту на персональні фінансові інструменти, підвищенням вимог користувачів до зручності інтерфейсу, безпеки даних та можливості роботи в умовах обмеженого доступу до Інтернету. Подібні рішення особливо корисні в умовах нестабільної економіки, коли ефективне управління особистими коштами стає важливим фактором фінансової безпеки.

Володіючи сучасним набором функцій (автоматизований облік транзакцій, гнучке бюджетування, візуалізація даних, управління боргами та рахунками, генерація аналітичних звітів, адаптивний інтерфейс), розроблюваний додаток покликаний забезпечити ефективне управління особистими фінансами та підвищити якість фінансового планування користувачів.

Із викладеного випливає **актуальність теми** кваліфікаційної роботи – зростаюча потреба сучасного суспільства в зручних, безпечних та функціональних програмних рішеннях для управління особистими фінансами на основі компонентно-орієнтованої архітектури та сучасних фронтенд-технологій.

Аналіз останніх досліджень і публікацій. Питанням розробки програмного забезпечення для управління фінансами та застосування компонентно-орієнтованих архітектур присвячено значну кількість наукових праць українських та зарубіжних авторів. Серед них можна виділити роботи, що розглядають особливості використання React та Electron [4, 5], застосування TypeScript для підвищення надійності коду [6], а також дослідження у сфері фінансових технологій (FinTech) та Open Banking [1–3]. Однак швидкий розвиток веб-технологій, поява нових версій фреймворків, зміна підходів до архітектури десктопних додатків та підвищення вимог до безпеки й продуктивності вимагають подальшого удосконалення методів створення таких систем. У зв’язку з цим питання впровадження компонентно-орієнтованої архітектури при розробці додатку управління особистими фінансами залишається актуальним.

Метою кваліфікаційної роботи є розробка функціонального десктопного додатка для управління особистими фінансами з впровадженням компонентно-орієнтованої архітектури на базі технологій React, Electron та TypeScript.

Для досягнення поставленої мети необхідно виконати такі **завдання**:

- проаналізувати предметну область та існуючі аналоги додатків для управління особистими фінансами;
- провести структурно-функціональне моделювання системи
- розробити архітектуру програмного продукту;
- спроєктувати моделі даних, користувацький інтерфейс та основні компоненти;
- реалізувати функціональність додатка з використанням сучасних

технологій;

- здійснити тестування, оцінку якості та апробацію отриманого програмного продукту.

Об’єктом дослідження є процес розробки десктопного програмного забезпечення для управління фінансами.

Предметом дослідження виступають методи та засоби створення додатків на основі компонентно-орієнтованої архітектури з використанням React, Electron та TypeScript.

Методи дослідження. Під час проєктування додатку застосовано ряд загальноприйнятих методів дослідження, які дозволили у повній мірі досягти мету проєкту та вирішення завдань розробки. Зокрема: бібліометричний метод дозволив проаналізувати наявні наукові праці у даній сфері діяльності; використовуючи різні філософські методи проведено всебічне аналізування предмету дослідження; аналіз, синтез, індуктивний та дедуктивний методи забезпечили можливість досягнення вище зазначених завдань.

Структура роботи. Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел та додатків. Обсяг основного тексту роботи складає 40 сторінок, 24 рисунки, 14 бібліографічних джерела. Загальний обсяг роботи – 61 сторінка.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОДАТКУ УПРАВЛІННЯ ФІНАНСАМИ

1.1. Аналіз існуючих рішень для управління особистими фінансами

У сучасних умовах повномасштабної війни, економічної невизначеності та постійних змін на фінансовому ринку ефективне управління особистими фінансами є важливою частиною фінансової стабільності та добробуту. Особливо важливими стають інструменти для контролю особистого бюджету, аналізу витрат і планування фінансів. Серед них значну роль відіграють спеціалізовані програми, які автоматизують облік доходів і витрат. Щоб визначити сучасні підходи до розробки таких програм, було проаналізовано найвідоміші продуктиякі користуються попитом[1].

Одним із найпопулярніших сервісів є YNAB (рис. 1.1).

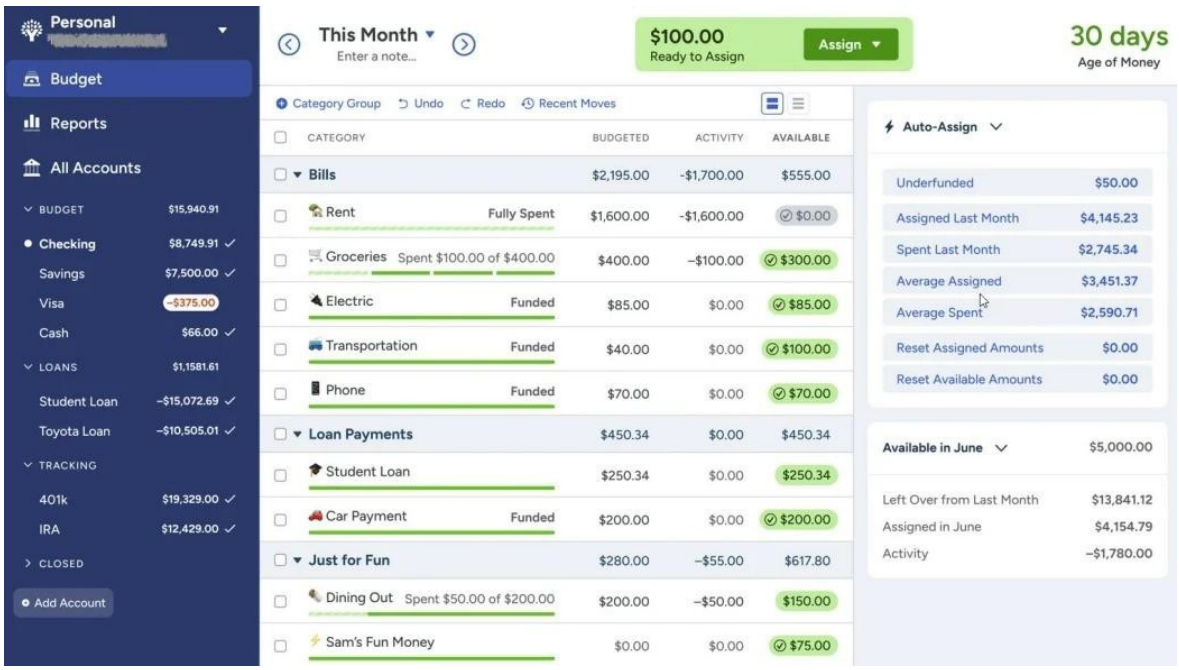

Рис. 1.1 – Інтерфейс додатку YNAB

Головна ідея цього сервісу – детальне планування бюджету та розумний розподіл коштів за категоріями. YNAB допомагає користувачам

розвивати фінансову дисципліну та краще керувати своїми ресурсами. Однак сервіс вимагає постійного ручного введення даних, а підписка коштує досить дорого. Крім того, можливості роботи з інвестиціями тут обмежені.

Ще одне популярне рішення – Empower. Він зосереджений на управлінні інвестиціями та довгостроковому фінансовому плануванні. Користувачі можуть об'єднувати дані з різних фінансових джерел і стежити за загальним станом свого капіталу. Також є можливість отримати консультації фінансових експертів. Але тут функції детального бюджетування менше розвинені, а розширені можливості доступні лише у платних версіях, що є недоліком для користувачів.

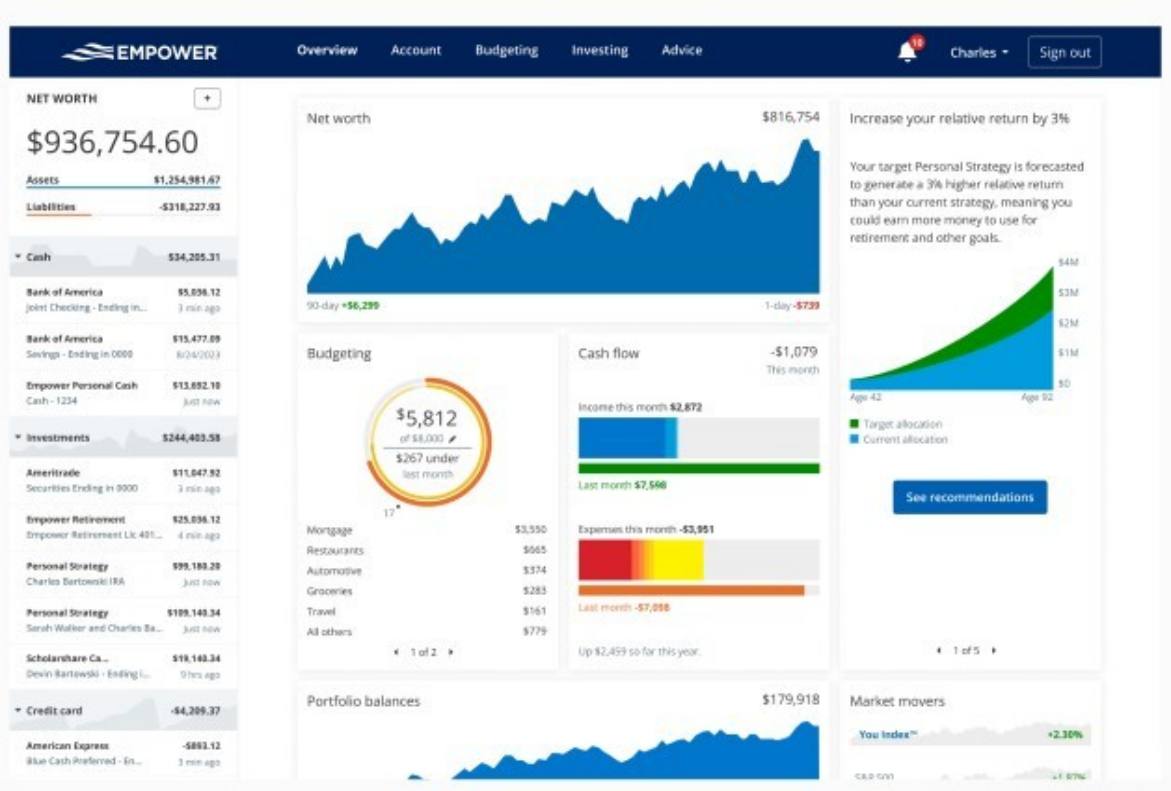

Рис. 1.2 – Інтерфейс додатку Empower

Сервіс Simplifi (рис. 1.3)підходить тим, хто шукає простий і зрозумілий інструмент для щоденного контролю фінансів. Ця програма дозволяє планувати витрати, ставити фінансові цілі і стежити за виконанням бюджету. У Simplifi інтерфейс розроблений так, щоб робота з фінансовими даними

була максимально простою. Недоліком є обмежені можливості для інвестиційного аналізу та складного бюджетування.

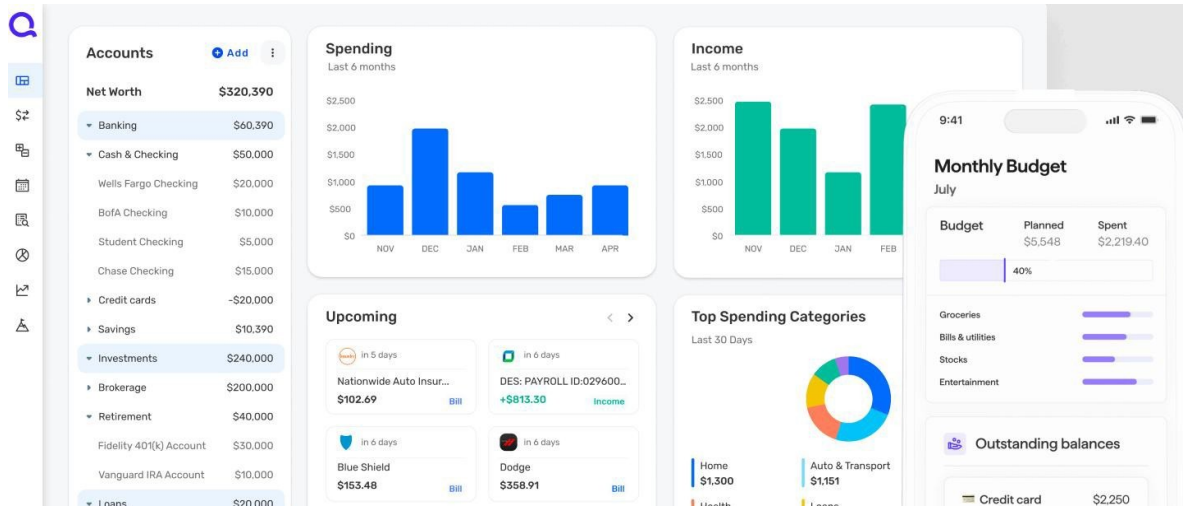

Рис. 1.3 – Інтерфейс додатку Simplifi

Rocket Money (рис 1.4) спеціалізується на контролі регулярних витрат і підписок. Система автоматично знаходить активні підписки та допомагає відмовитися від непотрібних сервісів. Також є інструменти для бюджетування, аналізу витрат і моніторингу кредитного рейтингу. Це дає користувачу зручний інструмент для оптимізації особистих фінансів.

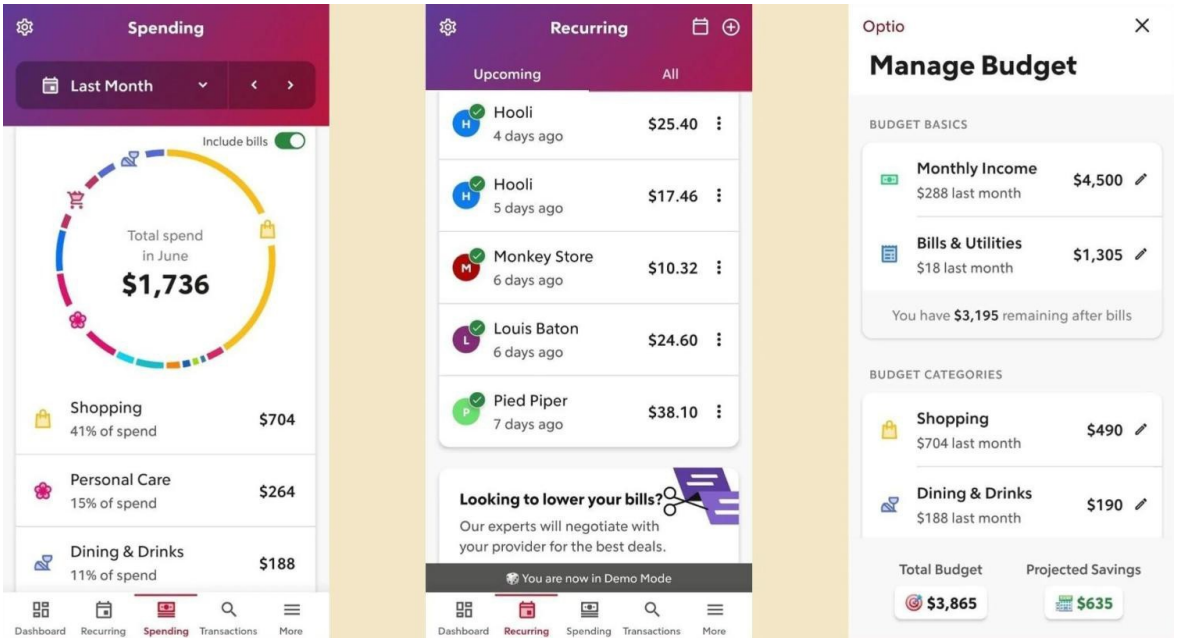

Рис. 1.4 – Інтерфейс додатку Rocket Money

Аналіз показав, що існуючі рішення мають різну спеціалізацію: одні зосереджені на бюджетуванні, інші – на інвестиціях або оптимізації витрат. Попри великий вибір програм, потрібен універсальний додаток з простим і зрозумілим інтерфейсом, який об’єднує основні фінансові інструменти.

1.2. Обґрунтування вибору напрямку дослідження

Для розробки додатку для управління, ведення та аналізу особистими фінансами було обрано формат настільного застосунку під операційну систему Windows.. Основними технологіями реалізації використано бібліотеку React, фреймворк Electron а також мову програмування TypeScript.

Компонентно-орієнтовану архітектури обрано через особливості фінансових застосунків. Вони складаються з багатьох пов’язаних модулів, таких як облік витрат і доходів, планування бюджету, статистика, аналітика та інші інструменти. Компонентно-орієнтована архітектура дозволяє розділити функції на незалежні частини, що спрощує підтримку, пришвидшує розвиток і дає змогу повторно використовувати код. Це особливо важливо для систем, які часто змінюються відповідно до потреб користувачів [3].

Вибір компонентно-орієнтованого підходу відповідає потребам сучасного ринку програмного забезпечення. Багато фінансових програм мають перевантажений інтерфейс, їх складно підтримувати і ними незручно користуватися. Тому створення продукту з продуманою архітектурою і зручним інтерфейсом є актуальним завданням.

Однією з основних технологій у розробці додатку управління особистими фінансами є бібліотека React. Бібліотеку React часто використовують для створення сучасних інтерфейсів, і вона підтримується компанією Meta та світовою спільнотою розробників. Головна ідея бібліотеки React – будувати інтерфейс з окремих незалежних компонентів,

кожен з яких відповідає за свій стан і логіку. Завдяки цьому можна створювати складні та динамічні інтерфейси.

Важливою перевагою React є використання Virtual DOM, що зменшує кількість звернень до реального DOM і підвищує швидкість роботи програми. Також бібліотека використовує синтаксис JSX, який поєднує JavaScript і HTML-подібну розмітку, роблячи код зрозумілішим і структурованим. Односпрямований потік даних між компонентами спрощує розробку і налагодження програми [4]..

React має кілька переваг: висока продуктивність, повторне використання компонентів, підтримка серверного рендерингу та універсальність для різних платформ. Серед недоліків – потрібно вивчати JSX, екосистема швидко змінюється, тому знання треба постійно оновлювати, а для маршрутизації та управління станом потрібні сторонні бібліотеки.

Ще одним інструментом для розробки додатку з використанням React на персональному комп’ютері було взято Electron. Цей фреймворк дозволяє розробляти кросплатформні десктопні застосунки на основі веб-технологій. Electron поєднує Chromium і середовище Node.js, тому можна використовувати одну кодову базу для різних операційних систем.

Крім того, Electron дозволяє швидко розробляти програми, підтримує Windows, Linux і macOS, а також дає доступ до функцій операційної системи через API Node.js. Це дає змогу інтегруватися з файловою системою, системними даними, меню та іншими компонентами ОС. Хоча і Electron використовує більше оперативної пам’яті і збільшує розмір програми через Chromium, він залишається одним із найпопулярніших інструментів для створення сучасних настільних програм[5].

Для цього додатку для управління особистими фінансами обрали мову програмування TypeScript, яка є надбудовою над JavaScript і додає статичну типізацію. Завдяки типам розробник контролює структуру даних ще під час

написання коду, це дозволяє знизити кількість помилок під час виконання програми.

Мова програмування TypeScript підвищує надійність програм, спрощує рефакторинг, покращує підтримку сучасними середовищами розробки і робить документацію зрозумілішою завдяки системі типів. Мову програмування TypeScript можна поступово впроваджувати у вже існуючі JavaScript-проєкти. Але мова програмування TypeScript має деякі недоліки: збільшується обсяг коду через типи, потрібна компіляція у JavaScript, а початковий етап розробки стає складнішим [6].

Використання TypeScript разом із React і Electron формує сучасний технологічний стек. Це дозволяє створювати якісний програмний продукт, який легко підтримувати та масштабувати в майбутньому.

Для серверної частини додатку для управління особистими фінансами обрали платформу Firebase . Вона має багато можливостей, легко інтегрується з клієнтськими технологіями та пропонує готові інструменти для створення повноцінних програм.

Платформа Firebase надає механізми автентифікації користувачів, що дозволяє організувати безпечний доступ до персональної інформації через електронну пошту, сторонні сервіси авторизації, або багатофакторну перевірку. Для зберігання даних використовується Firestore, документно-орієнтована NoSQL-база даних, яка добре працює зі складними структурами інформації та підтримує масштабування.

Найголовнішою перевагою платформи Firebase є підтримка синхронізації даних у реальному часі. Користувачі мають можливість відразу отримати оновлення інформації, що дозволяє спільно вести бюджет і відразу бачити фінансові зміни [7].

Основним середовищем розробки даного проєкту обрано редактор коду Visual Studio Code. Це безкоштовний редактор коду з відкритою архітектурою від Microsoft. Visual Studio Code поєднує високу

продуктивність, гнучкі налаштування та багато розширень, тому є одним із найпопулярніших інструментів серед розробників програмного забезпечення.

Visual Studio Code підтримує багато мов програмування та фреймворків, зокрема JavaScript, TypeScript і Node.js. Серед його можливостей є інтелектуальна система IntelliSense для автодоповнення коду, вбудовані засоби налагодження, система контролю версій і глибока персоналізація робочого середовища. Це робить його ефективним для проєктів будь-якої складності.

1.3 Технічне обґрунтування реалізації проєкту

Для організації розробки додатку для управління особистими фінансам ми обрали методологію Scrum [8], яка є гнучким підходом до управління проєктами. Короткі цикли розробки дають змогу поступово впроваджувати функціональні модулі системи, регулярно перевіряти їхню роботу та швидко реагувати на зміни вимог.

Запропонована система має модульну структуру. Кожен функціональний блок відповідає за окрему сферу фінансового управління: облік транзакцій, роботу з бюджетами, управління рахунками, формування звітів і контроль заборгованостей. Така архітектура робить компоненти незалежними і полегшує їхнє вдосконалення в майбутньому.

Компонентно-орієнтовану підхід добре підходить для використання з React, який дає змогу створювати багаторазові елементи інтерфейсу. Electron забезпечує інтеграцію програми з операційною системою, що дозволяє працювати з локальними файлами, системними повідомленнями та іншими ресурсами комп'ютера.

У результаті було сформовано технічну концепцію програмного продукту, яка базується на принципах компонентно-орієнтованої архітектури КОА, використанні сучасних веб-технологій та забезпечує можливість подальшого масштабування системи відповідно до нових потреб користувачів.

РОЗДІЛ 2
ПРОЄКТУВАННЯ МОДЕЛІ ТА ПРОГРАМНОГО КОМПЛЕКСУ

2.1. Проєктування додатку для управління особистими фінансами

Розробка програмного забезпечення на основі компонентно-орієнтованої архітектури передбачає поділ застосунку на окремі, незалежні функціональні модулі, кожен модуль виконує окремий набір завдань, має власну внутрішню логіку та взаємодіє з іншими елементами через стандартизовані інтерфейси. Така організація програмного забезпечення сприяє підвищенню гнучкості, масштабованості та спрощує подальший супровід системи.

На початковому етапі створення програмного продукту визначаються його основні функціональні складові, які далі реалізуються у вигляді окремих компонентів. Незалежність модулів дає змогу використовувати різні технологічні рішення для їх реалізації за умови дотримання встановлених правил взаємодії. У цьому випадку інтерфейс виступає механізмом узгодження між компонентами та визначає порядок обміну даними між ними [10].

Важливою характеристикою компонентно-орієнтованої архітектури є організація комунікації між модулями. Для цього можуть використовуватися виклики функцій, обмін повідомленнями та використання спільних сервісів або подій. Конкретний спосіб взаємодії обирається залежно від вимог до продуктивності, складності системи та особливостей реалізованої функціональності.

Окрему увагу приділяють управлінню життєвим циклом компонентів. До цього процесу належать створення, налаштування, оновлення та завершення роботи модулів. Для спрощення таких операцій використовуються спеціалізовані фреймворки та інструменти, які

забезпечують реєстрацію компонентів, контроль їх версій та автоматичне керування залежностями.

Для підвищення адаптивності програмного забезпечення компонентно-орієнтовані системи часто застосовуються механізми динамічного зв'язування та рефлексії. Завдяки цьому програма може підключати нові компоненти без необхідності перекомпіляції, а також аналізувати їх можливості під час виконання програми.

Таким чином, компонентно-орієнтований підхід забезпечує створення програмних продуктів, що легко масштабуються, модернізуються та підтримуються протягом тривалого часу.

Для реалізації програмного засобу було обрано архітектурний шаблон MVC (Model–View–Controller), який забезпечує чіткий розподіл відповідальності між різними частинами системи.

Концепція MVC передбачає поділ програмного забезпечення на три взаємопов’язані рівні. Модель відповідає за зберігання даних та реалізацію бізнес-логіки, представлення забезпечує відображення інформації користувачеві, а контролер виконує обробку подій та координує взаємодію між іншими складовими архітектури. Такий підхід сприяє впорядкованості коду, підвищує його читабельність і спрощує процес тестування [11].

У рамках розробки системи управління особистими фінансами використання MVC гармонійно поєднується з компонентною архітектурою. Якщо компонентний підхід забезпечує незалежність функціональних модулів, то MVC регламентує внутрішню організацію кожного з них.

Наприклад, модуль керування фінансовими операціями міститиме модель для роботи з інформацією про доходи та витрати, набір React-компонентів для відображення цих даних та контролер, який оброблятиме дії користувача під час створення, редагування або видалення транзакцій.

Фреймворк React реалізує рівень представлення, дозволяючи формувати інтерфейс на основі поточного стану застосунку. Завдяки

механізму компонентів та управлінню станом забезпечується оперативне оновлення відображуваних даних після внесення змін до моделі.

Поєднання компонентно-орієнтованої архітектури та MVC дозволяє створити добре структурований програмний продукт, окремі частини якого легко тестувати, супроводжувати та розширювати. Крім того, розділення логіки на незалежні рівні спрощує пошук та усунення помилок.

Передбачена реалізація архітектури матиме таку структуру у додатку для управління особистими фінансами:

Модель:

- класи TypeScript виступатимуть засобом представлення предметної області та зберігання даних;
- усі правила обробки інформації реалізовуватимуться в методах відповідних класів;
- взаємодія з хмарною платформою Firebase забезпечуватиме збереження, оновлення та отримання інформації через офіційний JavaScript SDK.

Представлення:

- інтерфейс користувача буде реалізовано за допомогою React-компонентів;
- компоненти відповідатимуть за відображення даних та взаємодію з користувачем;
- середовище Electron дозволить розгорнути веб-застосунок у форматі настільної програми.

Контролер:

- функції TypeScript оброблятимуть дії користувача;
- контролери координуватимуть оновлення даних та виклики бізнес-логіки;
- через них здійснюватиметься зв'язок між інтерфейсом і моделлю.

Після вибору архітектурного рішення було сформовано перелік функціональних та нефункціональних вимог до додатку.

Функціональні вимоги у додатку для управління особистими фінансами:

Система повинна забезпечувати:

- створення облікових записів та автентифікацію користувачів;
- ведення обліку доходів і витрат із розподілом за категоріями;
- формування та контроль персонального бюджету;
- перегляд історії фінансових операцій за обрані часові проміжки;
- автоматичне створення аналітичних звітів;
- графічне представлення фінансової статистики;
- облік регулярних платежів, комунальних послуг та підписок;
- реєстрацію кредитів, позик і боргових зобов'язань;
- налаштування параметрів профілю користувача;
- постановку фінансових цілей та моніторинг їх виконання;
- експорт накопичених даних.

Нефункціональні вимоги додатку для управління особистими фінансами:

Програмний продукт повинен відповідати таким характеристикам:

- швидкий запуск та оперативне виконання основних операцій;
- висока продуктивність під час роботи з даними;
- надійний механізм автентифікації користувачів;
- захист конфіденційної інформації за допомогою шифрування;
- інтуїтивно зрозумілий інтерфейс;
- зручна навігація між функціональними можливостями;
- стабільна робота без критичних помилок і збоїв.

Оскільки вимоги до програмних продуктів постійно змінюються, перелік функцій та характеристик може коригуватися на наступних етапах розробки. Потрібно постійно підтримувати актуальність моделі предметної області, це є необхідною умовою успішного розвитку інформаційної системи.

Запроєктований застосунок матиме високий рівень гнучкості завдяки використанню React, Electron, TypeScript та Firebase.

Адаптивність і кросплатформеність:

- Electron забезпечує можливість роботи програми на різних операційних

системах;

- React-компоненти легко пристосовуються до різних розмірів екранів і роздільних здатностей.

Масштабованість:

- модульна структура MVC спрощує додавання нових функцій;
- використання Firebase забезпечує інтеграцію із зовнішніми сервісами, включаючи банківські системи, платіжні сервіси та інвестиційні платформи.

Підтримка автономної роботи:

- локальне кешування даних дозволяє використовувати основні можливості програми без доступу до мережі;
- після відновлення інтернет-з’єднання здійснюється автоматична синхронізація інформації з хмарним сховищем.

У результаті проведеного аналізу було визначено основні вимоги до системи управління особистими фінансами, сформовано її архітектурну основу. Для реалізації програмного продукту обрано поєднання компонентно-орієнтованого підходу та шаблону MVC, що забезпечує високу модульність, зручність супроводу та можливість подальшого розвитку системи.

2.2. Формування бізнес-моделі програмного засобу

Одним із важливих етапів проєктування інформаційної системи є визначення способів взаємодії майбутніх користувачів із програмним продуктом. Для опису такої взаємодії в UML використовується діаграма варіантів використання (Use Case Diagram), яка дозволяє відобразити функціональні можливості системи та ролі користувачів, що беруть участь у виконанні певних операцій.

На відміну від технічних моделей, діаграма варіантів використання концентрується на функціональності програмного забезпечення з позиції

кінцевого користувача. Вона демонструє, які дії можуть виконувати актори в системі та які сервіси для цього надає програмний продукт [12].

Кожний варіант використання описує окрему бізнес-функцію або завдання, що реалізується за допомогою системи. До таких функцій можуть належати авторизація, управління фінансовими операціями, створення бюджетів чи формування аналітичних звітів. Актори, в свою чергу, представляють зовнішніх користувачів або інші системи, які взаємодіють із програмним забезпеченням.

Побудована діаграма варіантів використання для системи управління особистими фінансами наведена на рисю 2.1.

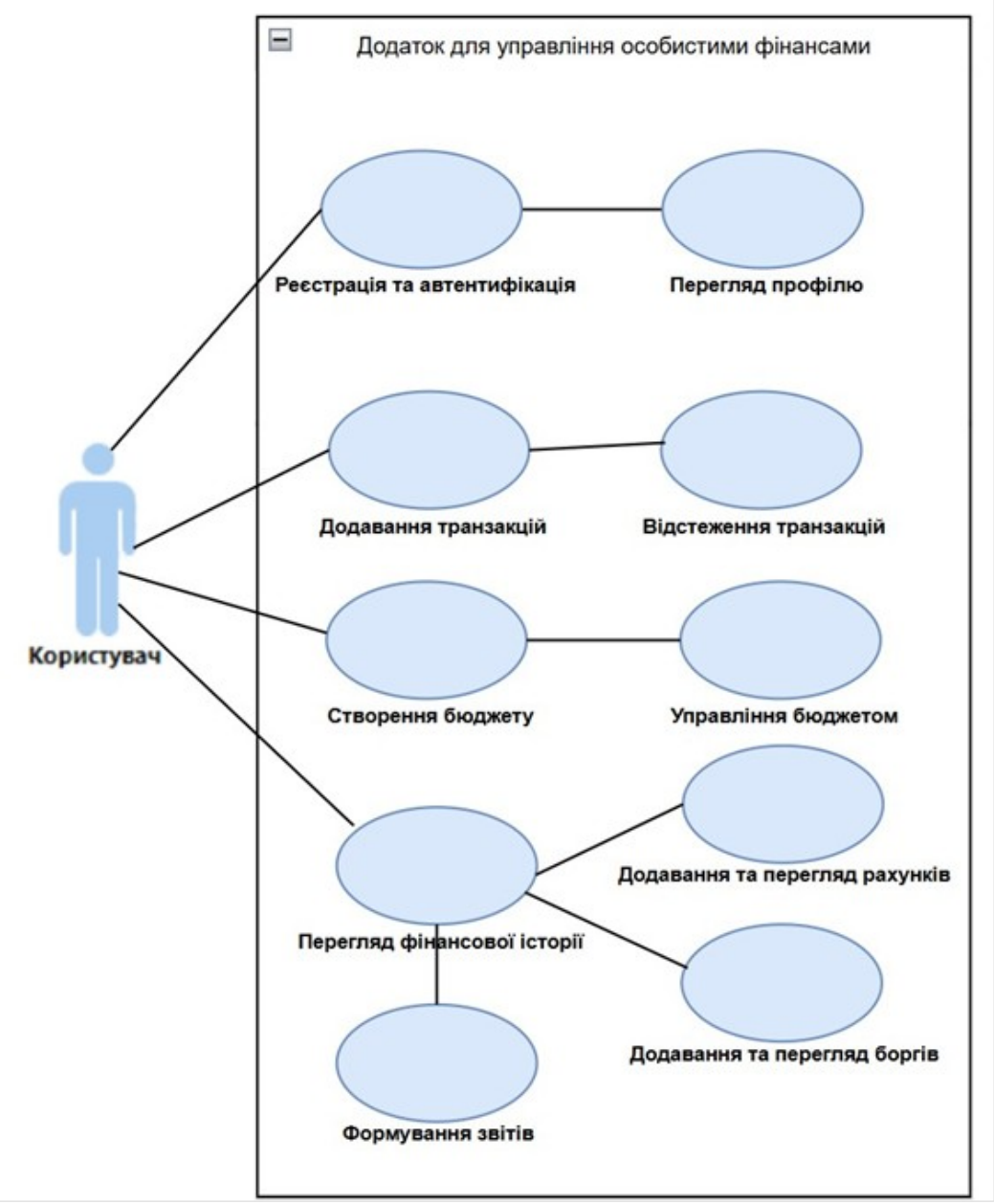

Рис. 2.1 Діаграма варіантів використання додатку

Основними складовими діаграми варіантів використання є:

- актори, що представляють користувачів системи;
- варіанти використання, які описують доступні функції;
- зв’язки між акторами та функціями, що демонструють можливість виконання певних операцій.

Після побудови діаграми варіантів використання було сформовано детальні сценарії взаємодії користувача із системою.

Сценарії виконання основних функцій:

Реєстрація та вхід до системи:

Основний сценарій:

1. Користувач запускає додаток.
2. Обирає функцію створення нового облікового запису.
3. Вводить персональні дані та облікові реквізити.
4. Система створює профіль та виконує авторизацію.

Альтернативний сценарій:

1. Користувач переходить до форми входу.
2. Вводить адресу електронної пошти та пароль.
3. Після перевірки даних система надає доступ до функціоналу застосунку.

Створення фінансової операції:

Основний сценарій:

1. Вибір функції додавання транзакції.
2. Визначення типу операції (дохід, або витрата).
3. Заповнення інформації про суму, дату та категорію.
4. Додавання коментаря до операції.
5. Збереження введених даних.

Альтернативний сценарій:

- користувач скасовує процес створення транзакції до моменту її збереження.

Перегляд історії операцій:

Основний сценарій:

1. Відкриття розділу транзакцій.
2. Отримання списку фінансових операцій.
3. Використання фільтрів за категоріями, типами або часовими проміжками.

Робота з профілем користувача:

Основний сценарій:

1. Перехід до сторінки профілю.
2. Перегляд персональної інформації.
3. Редагування налаштувань облікового запису.
4. Зміна пароля за потреби.

Формування бюджету:

Основний сценарій:

1. Відкриття інструменту планування бюджету.
2. Вибір періоду планування.
3. Встановлення лімітів для окремих категорій витрат.
4. Підтвердження та збереження бюджету.

Контроль бюджету:

Основний сценарій:

1. Перегляд створеного бюджету.
2. Аналіз поточного рівня витрат.
3. Коригування встановлених обмежень.
4. Відстеження прогресу виконання фінансового плану.

Аналіз фінансової історії:

Основний сценарій:

1. Перехід на головну сторінку.
2. Вибір історії фінансових операцій.

3. Отримання детальної інформації щодо проведених транзакцій.

Управління рахунками:

Основний сценарій:

1. Відкриття розділу рахунків.
2. Перегляд наявних фінансових рахунків.
3. Створення нового рахунку через спеціальну форму.
4. Введення необхідних параметрів.
5. Збереження та оновлення списку рахунків.
6. Перегляд детальної інформації щодо обраного рахунку.

Управління борговими зобов'язаннями:

Основний сценарій:

1. Перехід до модуля обліку боргів.
2. Перегляд існуючих записів.
3. Додавання нового боргу.
4. Введення інформації про кредитора та умови погашення.
5. Збереження даних.
6. Відстеження стану погашення заборгованості.

Створення аналітичних звітів:

Основний сценарій:

1. Відкриття модуля звітності.
2. Вибір необхідного типу звіту.
3. Налаштування параметрів аналізу.
4. Автоматичне формування звітних даних системою.

Таким чином, побудована бізнес-модель дозволила визначити основні процеси взаємодії користувача із системою та структурувати функціональні можливості майбутнього програмного продукту.

2.3 Проєктування архітектури системи

Для деталізації внутрішньої структури програмного забезпечення була розроблена діаграма класів UML. Вона використовується для опису об'єктної моделі системи та відображення взаємозв'язків між її складовими.

Кожний клас на діаграмі характеризується набором властивостей і методів, які визначають його поведінку. Зв'язки між класами демонструють способи взаємодії окремих елементів програмного забезпечення та формують загальну архітектуру системи [13].

Після аналізу вимог і функціональної структури додатку було сформовано діаграму класів, що представлена на рисунку 2.2.

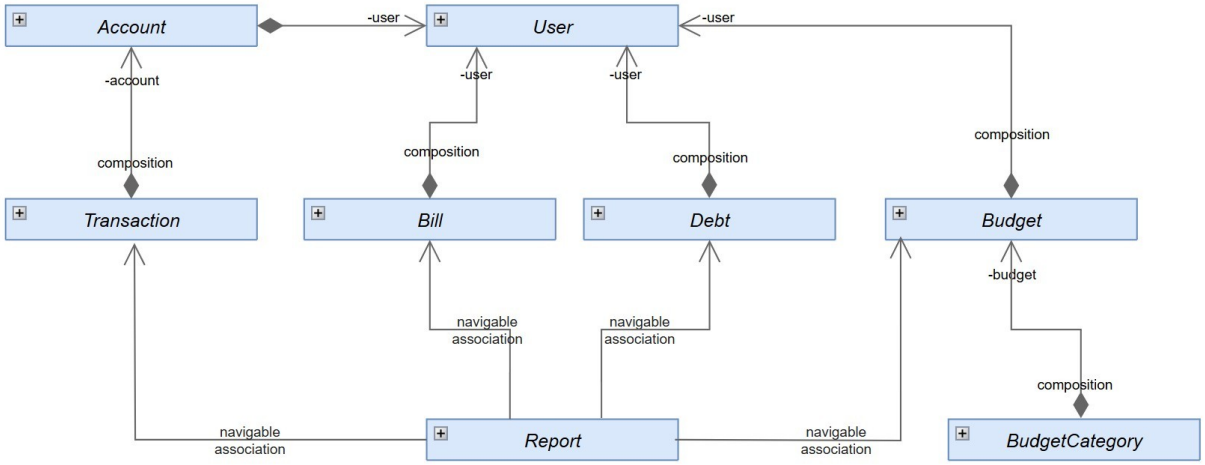

Рис. 2.2. Діаграма класів додатку

Опис основних класів

User – відповідає за управління обліковими записами користувачів. Містить персональні дані, налаштування профілю та механізми автентифікації. Основні операції класу передбачають вхід до системи, вихід із неї, оновлення персональної інформації та зміну пароля.

Account – реалізує логіку роботи з фінансовими рахунками. Зберігає інформацію про тип рахунку, валюту та поточний баланс. Дозволяє отримувати деталі рахунку та пов’язані фінансові операції.

Transaction – описує окрему фінансову операцію. Містить дані про суму, дату, категорію та тип транзакції. Передбачає можливість створення, редагування та видалення записів.

Bill – використовується для обліку регулярних платежів і рахунків. Зберігає відомості про суму платежу, постачальника послуг, дату оплати та параметри автоматичного нагадування.

Budget – відповідає за планування та контроль бюджету. Містить інформацію про доходи, витрати та період планування. Забезпечує розподіл коштів за категоріями та формування аналітики.

BudgetCategory – використовується для представлення окремих категорій бюджету. Дозволяє контролювати заплановані та фактичні витрати.

Debt – реалізує механізм управління кредитами та борговими зобов'язаннями. Забезпечує облік залишку боргу, нарахування відсотків і контроль процесу погашення.

Report – призначений для створення та збереження аналітичних звітів. Надає можливість перегляду та експорту результатів фінансового аналізу.

Розроблена діаграма класів відображає структуру програмного забезпечення, визначає взаємозв'язки між об'єктами та демонструє розподіл відповідальності між окремими модулями системи.

Для більш детального дослідження логіки роботи програмного продукту були створені діаграми послідовності. Цей тип UML-діаграм використовується для моделювання процесу обміну повідомленнями між компонентами системи під час виконання конкретного сценарію.

Діаграми послідовності дозволяють простежити часовий порядок виконання операцій, а також визначити ролі окремих об'єктів у реалізації функціональності. Такий підхід суттєво спрощує аналіз бізнес-процесів і допомагає виявляти потенційні недоліки архітектурних рішень.

Для прикладу було побудовано діаграму послідовності, що описує процес створення нової транзакції (рис. 2.3).

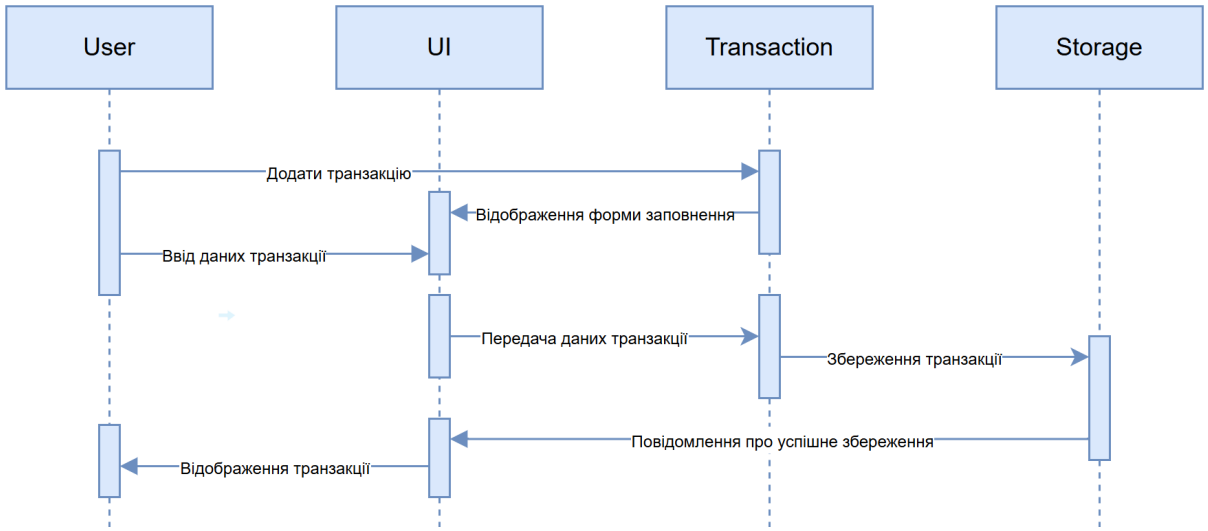

Рис. 2.3 Діаграма послідовності для сценарію "Додавання транзакцій"

Також була створена діаграма послідовності для процесу формування аналітичних звітів (рис. 2.4).

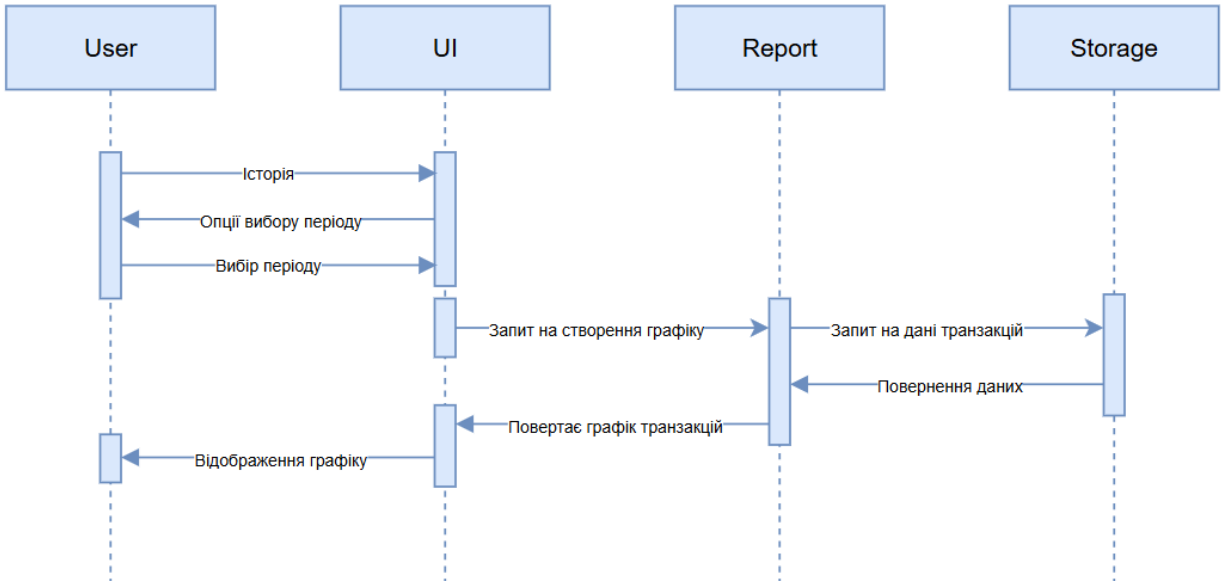

Рис. 2.4 Діаграма послідовності для сценарію "Формування звітів"

Розроблені діаграми дозволяють продемонструвати механізм взаємодії між компонентами системи, відобразити порядок виконання операцій та забезпечити краще розуміння функціонування програмного продукту.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ ДЛЯ СТВОРЕННЯ ДОДАТКУ УПРАВЛІННЯ ОСОБИСТИМИ ФІНАНСАМИ

3.1. Реалізація основних модулів додатку

При розробці програмного забезпечення реалізація особливу увагу приділяють створенню базових програмних модулів, які забезпечують реалізацію основного функціоналу системи. Саме від якості проєктування та програмної реалізації цих модулів залежить стабільність роботи застосунку, можливість його подальшого розвитку та простота технічного супроводу.

Модульний підхід дозволяє розділити систему на незалежні складові, кожна з яких виконує визначене коло завдань. Завдяки цьому зменшується дублювання коду, підвищується його читабельність, а внесення змін не потребує модифікації значної частини програмного продукту.

Реалізацію програмних компонентів було виконано в середовищі Visual Studio Code, яке забезпечує зручні засоби для розробки, тестування та налагодження застосунків на базі TypeScript (рис. 3.1).

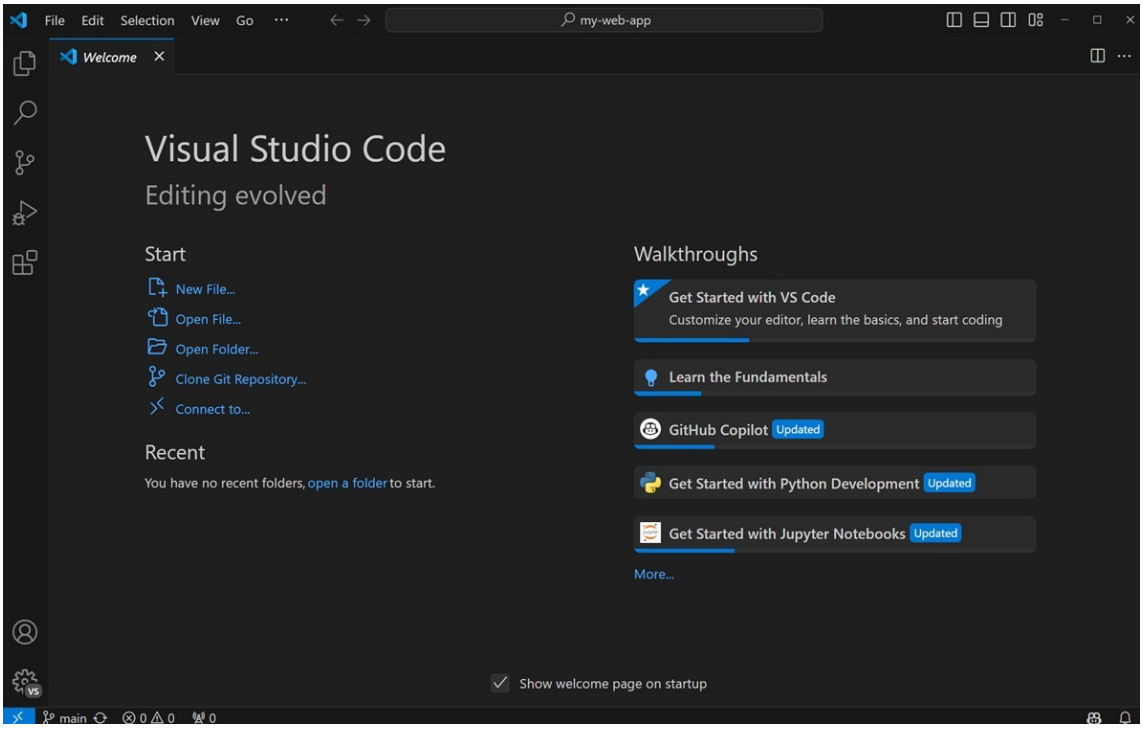

Рис. 3.1 Середовище розробки Visual Studio Code

У даному розділі розглянуто ключові модулі системи управління особистими фінансами та їх основні функціональні можливості.

Модуль керування користувачами:

Компонент **User** реалізує механізми роботи з обліковими записами користувачів. Його основне призначення у збереженні персональної інформації та забезпеченні безпечного доступу до системи.

Функціональні можливості модуля:

- зберігання даних профілю користувача;
- автентифікація та авторизація;
- оновлення персональної інформації;
- зміна облікових даних та пароля.

Для реалізації зазначеного функціоналу використовуються методи входу в систему, виходу з облікового запису, редагування профілю та оновлення параметрів безпеки (рис. 3.2.).

```
async login(): Promise<boolean> {
  try {
    // Example logic: Validate credentials against a database or API
    if (!this.email || !this.password) throw new Error("Missing credentials");
    console.log("User logged in successfully");
    return true;
  } catch (error) {
    console.error("Login failed:", error.message);
    return false;
  }
}

async changePassword(oldPassword: string, newPassword: string): Promise<boolean> {
  try {
    if (oldPassword !== this.password) throw new Error("Old password does not match");
    this.password = newPassword;
    console.log("Password changed successfully");
    return true;
  } catch (error) {
    console.error("Failed to change password:", error.message);
    return false;
  }
}
```

Рис. 3.2 Реалізація основних методів класу User

Модуль обліку фінансових операцій:

Компонент **Transaction** використовується для роботи з доходами та витратами користувача. Він забезпечує створення, редагування та видалення фінансових записів.

Основні завдання модуля:

- збереження інформації про транзакції;
- облік сум, дат та категорій операцій;
- групування транзакцій за типами;
- формування історії фінансової активності користувача.

Методи класу відповідають за додавання нових записів, зміну існуючих та їх видалення (рис. 3.3.).

```
createTransaction(): void {
  try {
    if (this.amount <= 0) throw new Error("Transaction amount must be positive");
    console.log("Transaction created successfully");
  } catch (error) {
    console.error("Failed to create transaction:", error.message);
  }
}

editTransaction(newTransactionData: Partial<Transaction>): void {
  try {
    Object.assign(this, newTransactionData);
    console.log("Transaction updated successfully");
  } catch (error) {
    console.error("Failed to update transaction:", error.message);
  }
}

deleteTransaction(): void {
  console.log("Transaction deleted successfully");
}
```

Рис. 3.3 Реалізація методів компонента Transaction

Модуль управління рахунками до сплати:

Компонент Bill призначений для обліку регулярних платежів, серед яких комунальні послуги, оренда, інтернет, мобільний зв’язок та різноманітні підписки.

Функції модуля включають:

- зберігання параметрів рахунків;
- планування майбутніх платежів;
- контроль термінів оплати;
- надсилання нагадувань користувачеві.

Для цього реалізовано механізми автоматичного планування та виконання платежів (рис. 3.4).

```
payBill(): void {
  try {
    console.log(`Bill ${this.billName} paid successfully`);
  } catch (error) {
    console.error("Failed to pay bill:", error.message);
  }
}

schedulePayment(date: Date): void {
  try {
    if (date <= new Date()) throw new Error("Scheduled date must be in the future");
    console.log(`Bill ${this.billName} scheduled for payment on ${date}`);
  } catch (error) {
    console.error("Failed to schedule payment:", error.message);
  }
}
```

Рис. 3.4 Основні методи класу Bill

Модуль бюджетування:

Компонент Budget відповідає за фінансове планування та контроль використання коштів.

Його функціональність охоплює:

- створення бюджетів на визначений період;
- розподіл фінансових ресурсів між категоріями;
- аналіз фактичних витрат;
- оцінку відхилень від запланованих показників;
- формування аналітичних звітів.

Завдяки цьому користувач може контролювати власні фінанси та ефективніше планувати витрати (рис. 3.5).

```
allocateBudget(): void {
  try {
    this.categories.forEach((category) => {
      if (category.allocatedAmount > this.totalIncome) {
        throw new Error(
          `Allocated amount for category ${category.categoryName} exceeds total income.`
        );
      }
    });
    console.log("Budget allocated successfully.");
  } catch (error) {
    console.error(`Failed to allocate budget: ${error.message}`);
  }
}

trackExpenses(): void {
  try {
    const totalTracked = this.categories.reduce(
      (sum, category) => sum + category.actualAmount,
      0
    );
    if (totalTracked > this.totalIncome) {
      console.warn("Expenses exceed total income.");
    }
    console.log("Expenses tracked successfully.");
  } catch (error) {
    console.error(`Failed to track expenses: ${error.message}`);
  }
}
```

Рис. 3.5 Реалізація методів класу Budget

Модуль обліку заборгованостей:

Компонент Debt забезпечує управління кредитами, позиками та іншими фінансовими зобов'язаннями.

До його функцій належать:

- збереження інформації про боргові зобов'язання;
- облік платежів;
- автоматичний розрахунок відсотків;
- контроль процесу погашення боргу.

Реалізовані методи дозволяють відстежувати поточний стан заборгованості та аналізувати динаміку її погашення (рис. 3.6).

```
calculateInterest(): number {
  try {
    const interest = (this.balance * this.interestRate) / 100;
    console.log(
      `Interest calculated for debt ${this.creditor}: ${interest.toFixed(2)}`
    );
    return interest;
  } catch (error) {
    console.error(`Failed to calculate interest: ${error.message}`);
    return 0;
  }
}

trackProgress(): number {
  try {
    const progress = ((this.minimumPayment / this.balance) * 100).toFixed(2);
    console.log(
      `Progress tracked for debt ${this.creditor}: ${progress}% paid`
    );
    return parseFloat(progress);
  } catch (error) {
    console.error(`Failed to track progress: ${error.message}`);
    return 0;
  }
}
```

Рис. 3.6 Основні методи класу Debt

Модуль формування звітності

Компонент **Report** відповідає за генерацію фінансової аналітики на основі накопичених даних.

Основні можливості:

- створення звітів за різними параметрами;
- відображення статистичних даних;
- побудова таблиць та графіків;
- експорт результатів аналізу до зовнішніх форматів.

Завдяки цьому користувач отримує інструменти для оцінювання власного фінансового стану та прийняття обґрунтованих рішень (рис.3.7).

```
generateReport(): void {
  try {
    if (!this.data) {
      throw new Error("No data available to generate the report.");
    }
    console.log(`${this.reportType} report generated successfully.`);
  } catch (error) {
    console.error(`Failed to generate report: ${error.message}`);
  }
}

viewReport(): void {
  try {
    console.log(`${this.reportType} report viewed:`, this.data);
  } catch (error) {
    console.error(`Failed to view report: ${error.message}`);
  }
}
```

Рис. 3.7 Основні методи класу Report

Таким чином, реалізовані модулі формують функціональне ядро системи управління особистими фінансами. Кожний компонент виконує окремий набір завдань і забезпечує роботу відповідного бізнес-процесу.

3.2. Розробка графічного інтерфейсу користувача

Однією з важливих складових сучасного програмного забезпечення є графічний інтерфейс користувача, який забезпечує зручну взаємодію людини із системою. Від якості його реалізації значною мірою залежить комфорт використання програмного продукту та швидкість виконання користувацьких операцій[14].

Під час проєктування інтерфейсу особливу увагу було приділено простоті навігації, логічному розташуванню елементів керування та візуальному представленню фінансової інформації.

Для попереднього опрацювання структури інтерфейсу було створено набір макетів основних сторінок застосунку.

Головна сторінка:

Першим було спроєктовано макет стартового екрана, який відкривається після входу користувача до системи.

На сторінці розміщено:

- панель сповіщень;
- блок профілю користувача;
- інформаційні картки із фінансовими показниками;
- графічні елементи для відображення статистики доходів і витрат.

Створений макет представлений на рис. 3.8.

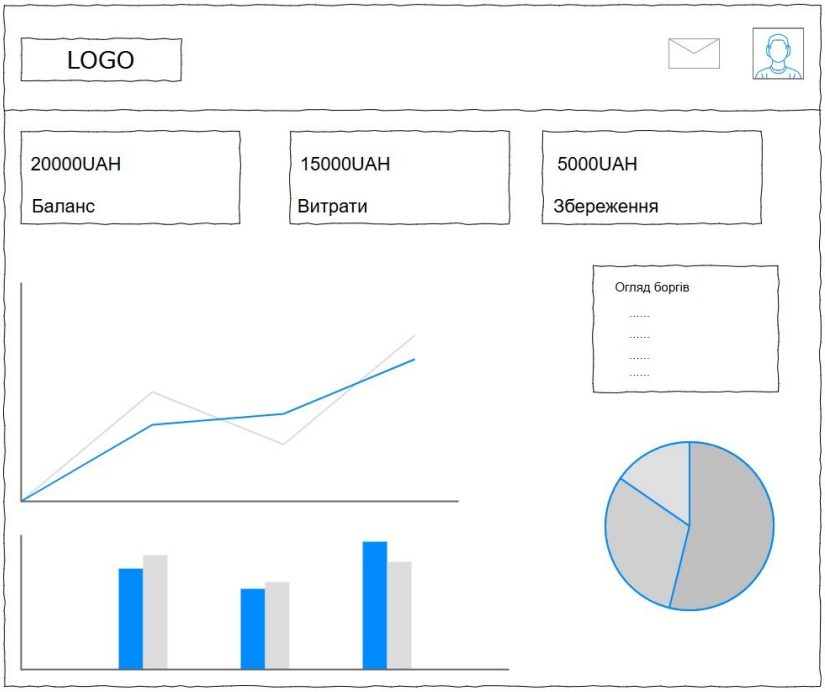

Рис. 3.8 Макет головної сторінки

Розділ управління бюджетом:

Для модуля бюджетування було створено окремий інтерфейс, який дозволяє контролювати доходи, витрати та заощадження (рис. 3.9).

На сторінці передбачено:

- блок джерел надходжень;
- налаштування фінансових лімітів;
- кругову діаграму розподілу бюджету;
- секцію накопичень.

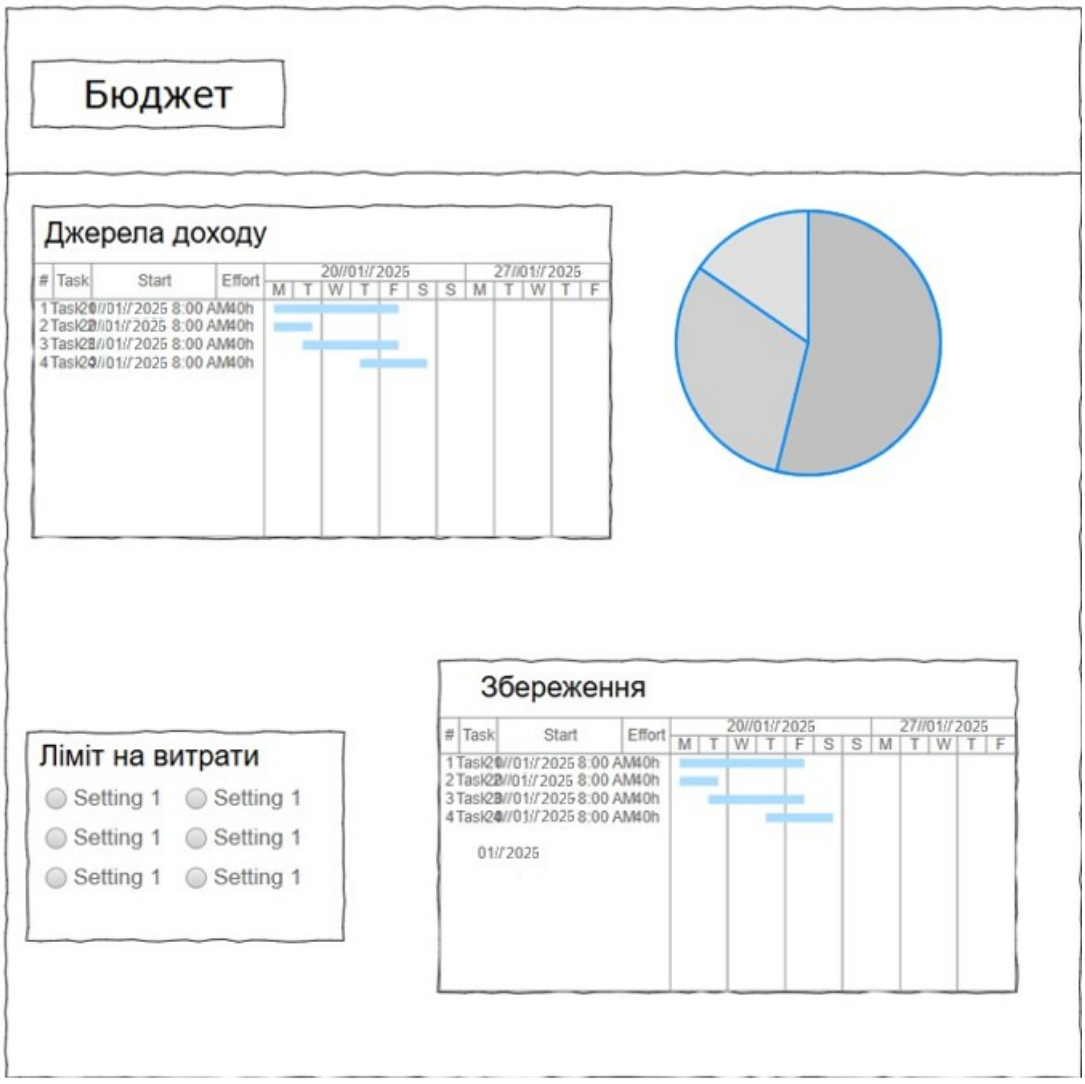

Рис. 3.9 Макет сторінки бюджету

Розділ рахунків:

Наступним етапом стало створення інтерфейсу для контролю регулярних платежів (рис. 3.10).

Сторінка містить:

- перелік рахунків;
- інформацію про статус оплат;
- графік динаміки платежів.

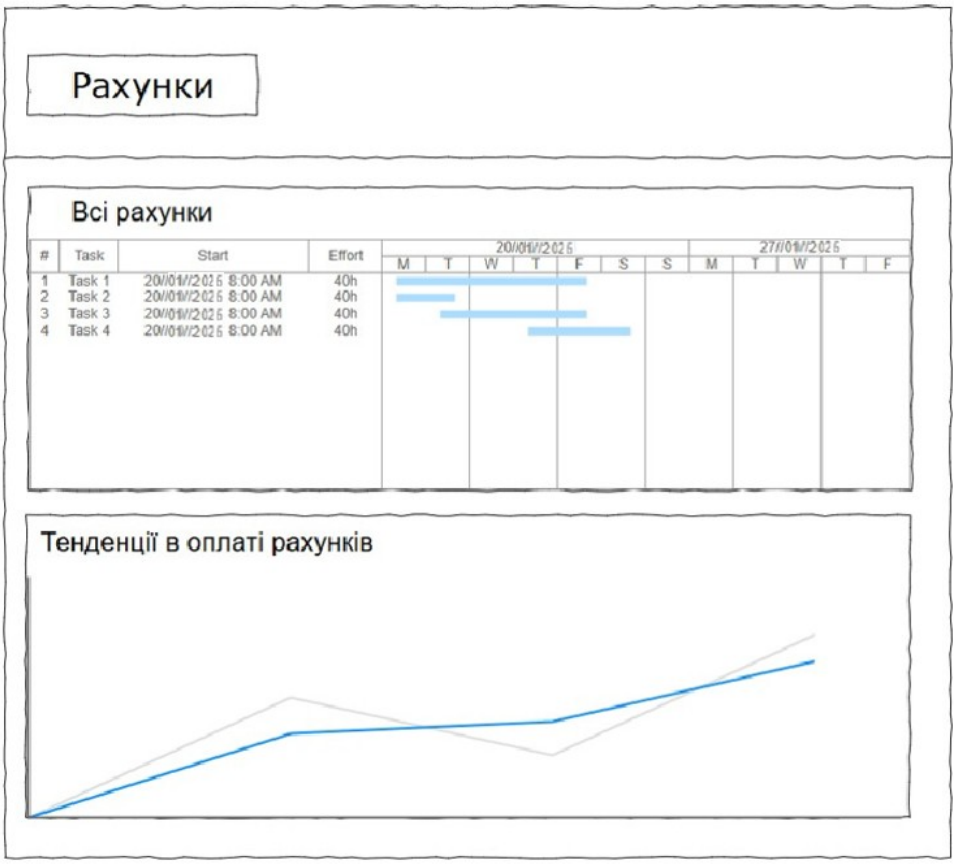

Рис. 3.10 Макет сторінки рахунків

Розділ аналітики:

Для відображення фінансової статистики було розроблено сторінку аналітики (рис. 3.11).

На ній передбачено:

- графіки доходів і витрат;
- діаграми розподілу коштів;
- аналітичні показники діяльності користувача.

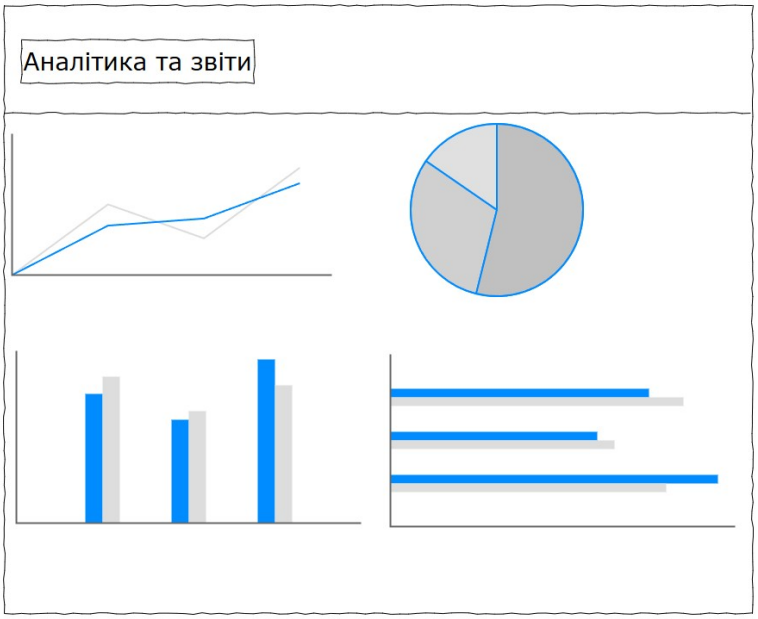

Рис. 3.11 Макет сторінки аналітики

Розділ управління боргами:

Окремий інтерфейс призначено для роботи із заборгованостями (рис. 3.12).

На сторінці відображаються:

- поточні борги;
- історія платежів;
- статистичні дані щодо процесу погашення.

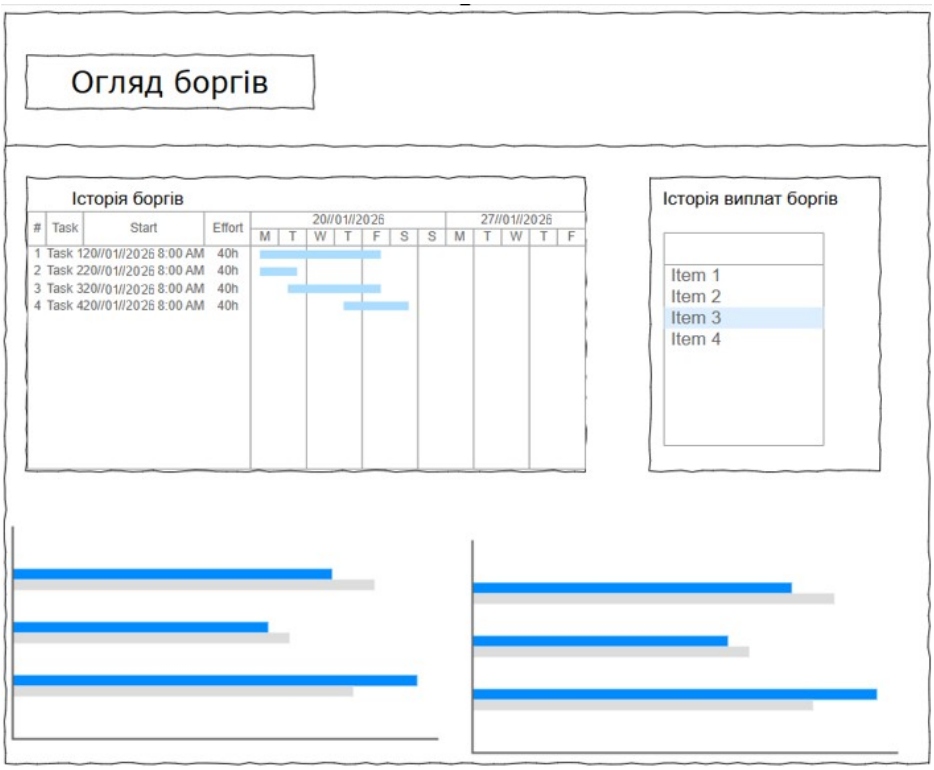

Рис. 3.12 Макет сторінки управління боргами

3.3. Тестування, оцінка якості та результат розробки додатку

Завершальним етапом створення додатку є комплексна перевірка його функціональних можливостей, стабільності роботи та відповідності поставленим вимогам. Саме тестування дозволяє підтвердити коректність реалізації бізнес-логіки, виявити можливі недоліки та оцінити готовність програмного продукту до практичного використання. У межах цього розділу виконано перевірку розробленої інформаційної системи управління особистими фінансами, проаналізовано якість її функціонування та наведено досягнуті результати роботи.

Під час перевірки було встановлено, що програмний продукт коректно функціонує в середовищі операційних систем Windows 10 та Windows 11 без виникнення критичних помилок чи втрати продуктивності. Реалізований застосунок, створений із використанням технологій React та Electron, продемонстрував високий рівень швидкодії, надійності та зручності взаємодії з користувачем.

Після запуску системи користувач потрапляє на головну сторінку, яка забезпечує доступ до основних функцій керування фінансами. Інтерфейс має навігаційну панель із розділами бюджету, рахунків, аналітики, управління заборгованістю та налаштувань. У верхній частині вікна відображаються відомості про користувача, система сповіщень та ключові фінансові індикатори, зокрема поточні витрати, баланс коштів і накопичення. Центральне місце займає графічне представлення динаміки доходів і витрат, а також таблиця боргових зобов'язань із можливістю швидкого пошуку та експорту інформації.

Загальний вигляд стартового екрана представлено на рис. 3.13.

Рис. 3.13 Початкова сторінка додатку

Нижня частина головної сторінки орієнтована на деталізований аналіз фінансової активності користувача. Тут розміщено журнал фінансових операцій із відображенням їхніх основних параметрів, а також інструменти для пошуку необхідних записів і вивантаження даних. Для наочного представлення інформації використано стовпчикову діаграму щоденних витрат та кругову діаграму структури бюджету за окремими категоріями. Такі елементи дають змогу швидко оцінити характер витрачання коштів і визначити найбільш ресурсомісткі напрями фінансової діяльності.

Відповідні компоненти інтерфейсу наведено на рис. 3.14.

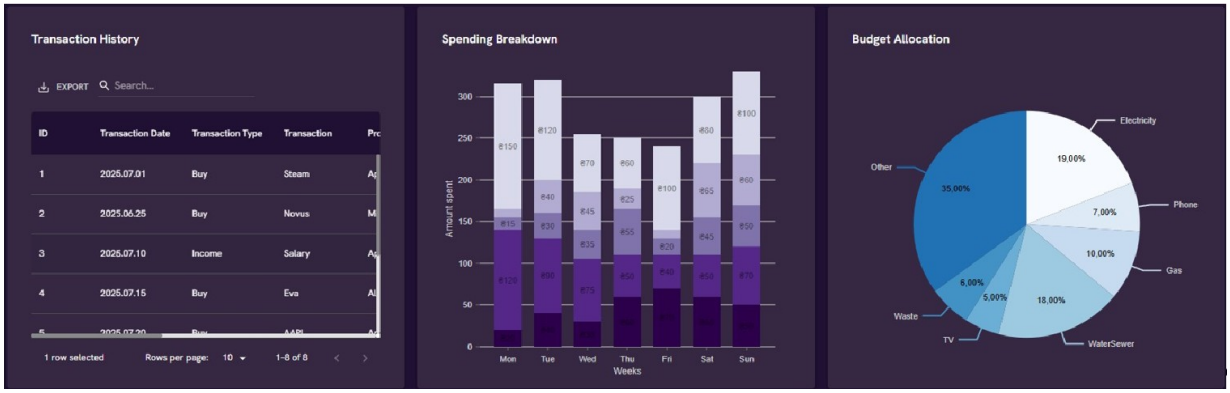

Рис. 3.14 Друга половина основної сторінки

Розділ «Бюджет» призначений для контролю джерел надходження коштів і планування фінансових ресурсів. У ньому представлено перелік усіх джерел доходу із зазначенням їх характеристик: назви, суми, способу отримання, періодичності надходження та належності до певної категорії. Для підвищення зручності роботи реалізовано функції пошуку та експорту даних.

Додатково використано кругову діаграму, яка відображає структуру формування бюджету та співвідношення окремих джерел доходів.

Вигляд сторінки подано на рис. 3.15.

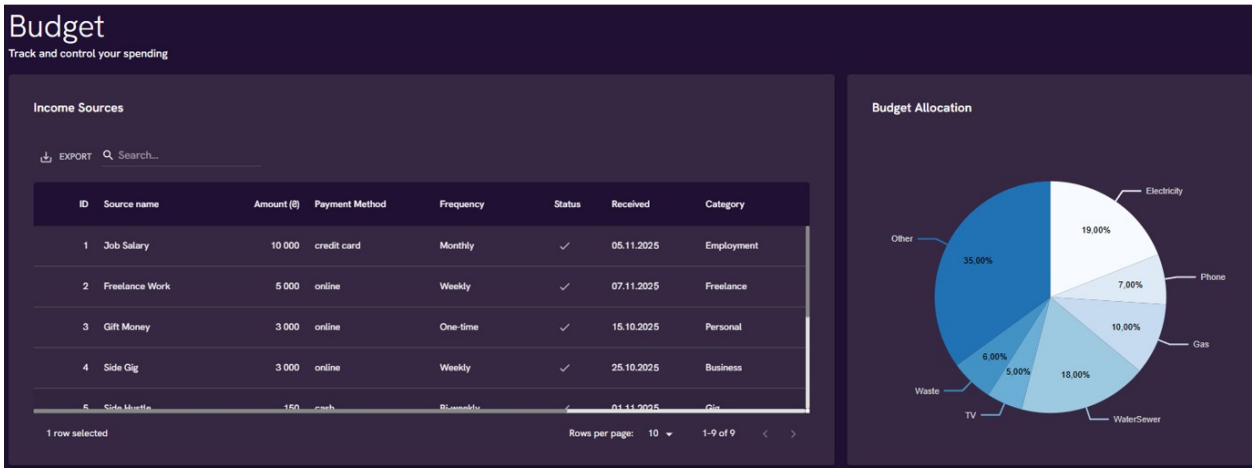

Рис. 3.15 Розділ «Бюджет»

Подальша частина розділу присвячена контролю витрат і формуванню накопичень. Ліва область інтерфейсу відображає встановлені обмеження за окремими категоріями витрат із можливістю їх редагування та створення

нових категорій. Права частина містить інструменти управління фінансовими цілями, де користувач може переглядати інформацію про накопичення, контролювати прогрес досягнення поставлених цілей та оцінювати рівень виконання запланованих показників.

Зазначені функціональні можливості продемонстровано на рис. 3.16.

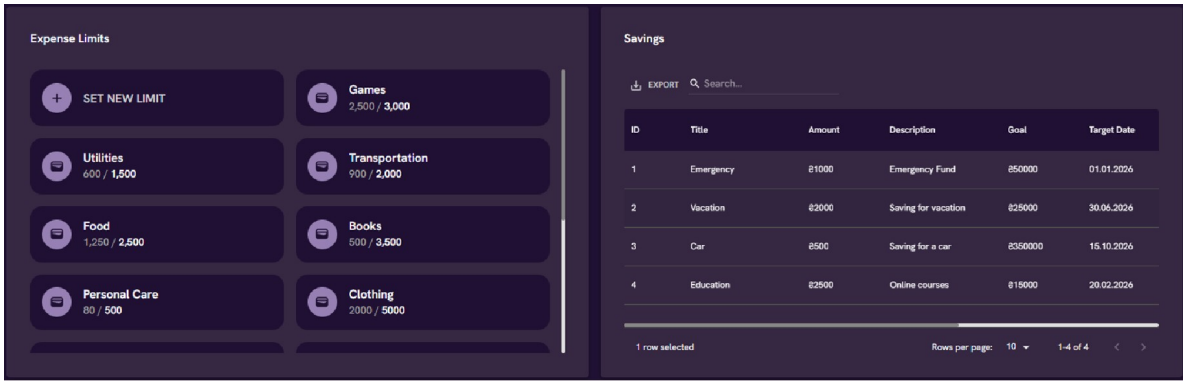

Рис. 3.16 Друга частина розділу «Бюджет»

Для контролю регулярних платежів реалізовано окремий модуль роботи з рахунками. Він містить інформацію про поточні фінансові зобов’язання, їхній статус, терміни виконання, способи оплати, вартість та категорію. Наявність інструментів фільтрації та експорту спрощує процес моніторингу платежів і дозволяє своєчасно виконувати фінансові зобов’язання.

Інтерфейс відповідної сторінки наведено на рис. 3.17.

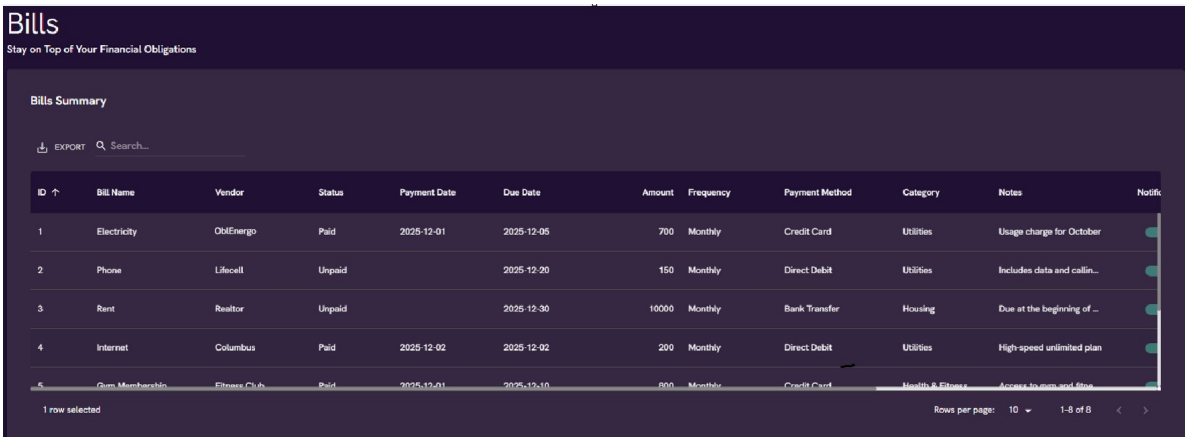

Рис. 3.17 – Рахунки на створеній сторінці

Модуль звітності та фінансового аналізу забезпечує можливість оцінювання ефективності управління коштами. Одним із його елементів є порівняльна гістограма, що демонструє співвідношення між запланованими та фактичними витратами в розрізі окремих категорій. Додатково реалізовано графік змін доходів і витрат упродовж року, який дозволяє відстежувати фінансові тенденції, аналізувати сезонні коливання та виявляти потенційні дисбаланси бюджету.

Відповідний інтерфейс показано на рис. 3.18.

Рис. 3.18 Звіти та аналіз фінансової ситуації

Окрему увагу приділено модулю управління заборгованістю. Він забезпечує централізоване відображення інформації про всі боргові зобов’язання користувача, включаючи кредиторів, види боргів, відсоткові ставки, мінімальні платежі, строки погашення та статус виконання зобов’язань. Поряд із цим реалізовано журнал платежів, який дозволяє переглядати історію погашення боргів і контролювати виконані фінансові операції.

Вигляд сторінки представлено на рис. 3.19.

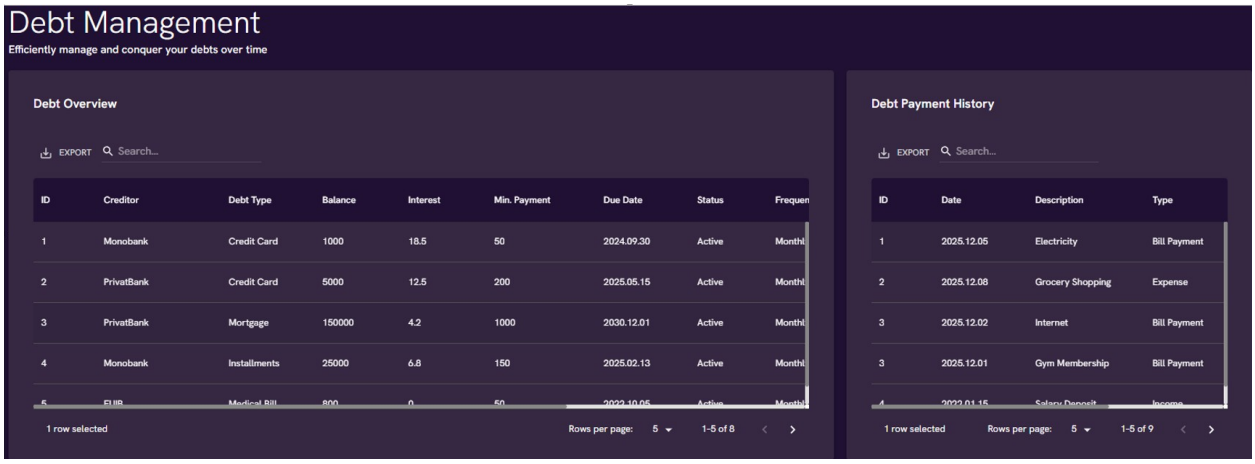

Рис. 3.19 Сторінка управління боргами

Наступна частина модуля заборгованості містить аналітичні інструменти у вигляді двох графічних візуалізацій. Перша діаграма відображає зміну співвідношення боргового навантаження до рівня доходів користувача в часовому розрізі, що дає змогу оцінити динаміку фінансових ризиків. Друга діаграма демонструє структуру загального боргу за окремими категоріями, дозволяючи визначити найбільш вагомі складові фінансових зобов’язань. Завдяки цьому користувач отримує комплексне уявлення про поточний стан власної заборгованості.

Дані елементи показано на рис. 3.20.

Рис. 3.20 Графіки щодо статистики боргів

Завершальним компонентом системи є сторінка налаштувань та керування профілем. Вона забезпечує можливість перегляду й редагування персональних даних, налаштування локалізації, параметрів відображення

дати, часу та валюти, а також керування інформацією про місцезнаходження користувача. Окремо реалізовано механізми зміни пароля та налаштування безпеки облікового запису, що сприяє підвищенню рівня захисту персональних даних.

Отже, результати тестування підтвердили працездатність і функціональну повноту розробленого програмного продукту. Усі передбачені модулі працюють коректно та забезпечують ефективне виконання поставлених завдань. Інтерфейс системи характеризується логічною структурою, зручністю навігації та наявністю сучасних засобів візуалізації даних, що значно підвищує ефективність управління особистими фінансами та спрощує процес прийняття фінансових рішень.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було проведено комплексне дослідження принципів компонентно-орієнтованої архітектури до розробки програмного забезпечення та проаналізовано можливості його використання під час створення додатку управління особистими фінансами.

Виконано аналіз сучасних технологій розробки, зокрема React, Electron та TypeScript, визначено їх переваги та доцільність застосування для створення кросплатформеного програмного продукту.

На основі проведеного аналізу було сформовано бізнес-модель системи, побудовано діаграми варіантів використання, діаграми класів та діаграми послідовностей, що дозволило формалізувати вимоги до програмного забезпечення та визначити структуру майбутнього застосунку.

Практична частина роботи охопила реалізацію основних програмних модулів, створення графічного інтерфейсу та проведення тестування готового продукту. Розроблений застосунок забезпечує повний цикл управління особистими фінансами, включаючи ведення обліку доходів і витрат, планування бюджету, контроль платежів, управління заборгованістю та формування аналітичної звітності.

Проведене тестування підтвердило працездатність програмного забезпечення, його стабільність та відповідність поставленим вимогам. Отримані результати підтвердили ефективність використання компонентно-орієнтованої архітектури при створенні складних інформаційних систем.

Подальший розвиток програмного продукту може бути пов’язаний із впровадженням двофакторної автентифікації, інтеграцією технологій відкритого банкінгу, автоматичним імпортом транзакцій із банківських сервісів та розширенням можливостей фінансової аналітики.

Поставлені завдання було виконано в повному обсязі та досягнуто мети роботи шляхом створення функціонального додатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fintech Analysis of Personal Finance App Usage among Millennials. Tika Handayani et al. Journal of Economic Education and Entrepreneurship Studies. 2024. Vol. 5, no. 2. P. 150–162.

2. Everything about Open Banking URL: до ресурсу: https://www.openbanking.org.uk/

3. Stefanyshyn, I., Pastukh, O., Stefanyshyn, V., Baran, I., Boyko, I. Robustness of AI algorithms for neurocomputer interfaces based on software and hardware technologies CEUR Workshop Proceedings. 2024. 3742. pp. 137–149.

4. Документація по React URL: https://react.dev/

5. Документація по Electron URL: https://www.electronjs.org/docs

6. Документація по TypeScript URL: https://www.typescriptlang.org/docs/

7. Документація Firebase URL: https://Firebase .google.com/docs

8. Що таке Scrum? URL: https://www.scrum.org/resources/what-scrum-module

9. Петрик М. Р., Мудрик І. Я. Проєктування програмного забезпечення на основі об’єктно-орієнтованого аналізу вимог та інструментальних засобів розробки IBM Rational Software Architect. М. Р. Петрик, І. Я. Мудрик. Тернопіль : ТНТУ ім. І. Пулюя, 2022. – 56 с.

10. Все про компонентно-орієнтовану архітектуру URL: https://www.geeksforgeeks.org/component-based- architecture-system-design/

11. Словник MVC URL: https://developer.mozilla.org/en-US/docs/Glossary/MVC

12. Основи та практичне використання UML URL: https://www.uml.org/

13. The Unified Modeling Language URL: https://www.uml-diagrams.org/

14. Hiatt J. C. React JS Foundations Building User Interfaces with ReactJS: An Approachable Guide. Wiley & Sons, Incorporated, John, 2022.

ДОДАТКИ

ДОДАТОК А

Лістинг коду розробленого додатку

Лістинг 1

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import {
   createHashRouter as createRouter,
   createRoutesFromElements as defineRoutes,
   Route,
   RouterProvider as AppRouter,
   } from "react-router-dom";

// Importing app components
import Dashboard from "./pages/Dashboard";
import Budget from "./features/budget";
import Bills from "./features/bills";
import Reports from "./features/reports";
import Debt from "./features/debt";
import Settings from "./features/settings";
import Overview from "./features/overview";
import ErrorPage from "./pages/Error";

// Define routes
const appRoutes = defineRoutes(
  <Route path="/" element={<Dashboard />}
  errorElement={<ErrorPage />}>
     <Route index element={<Overview />} />
     <Route path="budget" element={<Budget />} />
     <Route path="bills" element={<Bills />} />
```

```
    <Route path="reports" element={<Reports />} />
    <Route path="debt" element={<Debt />} />
    <Route path="investments" element={<Investments />}
    />
    <Route path="networth" element={<NetWorth />} />
    <Route path="settings" element={<Settings />} />
  </Route>
);

// Create the router
const routerConfig = createRouter(appRoutes);

// Render the application
ReactDOM.createRoot(document.getElementById("root")!).r
ender(
   <React.StrictMode>
     <AppRouter router={routerConfig} />
   </React.StrictMode>
);
```

Лістинг 2

```
export class Budget {
  private incomeSources: { id: number; name: string;
amount: number }[] = [];
  private expenseLimits: { id: number; title: string;
currentValue: number; limit: number }[] = [];
  private savings: { id: number; description: string;
amount: number }[] = [];
  private allocation: { id: string; label: string;
value: number; color: string }[] = [];
```

```
// Income Sources getIncomeSources() {
return [...this.incomeSources]; // Return a copy for
immutability
}
addIncomeSource(name: string, amount: number) {
   if (amount <= 0) throw new Error("Income amount must
be greater than 0");
   if (this.incomeSources.some((source) => source.name
=== name)) { throw new Error(`Income source "${name}"
already exists`);
   }
   this.incomeSources.push({  id:
this.generateId(this.incomeSources), name, amount });
   this.updateAllocation();
}

// Expense Limits
getExpenseLimits() {
   return [...this.expenseLimits];
}
addExpenseLimit(title: string, limit: number) {
   if (limit <= 0) throw new Error("Expense limit must
be greater than 0");
   this.expenseLimits.push({ id:this.generateId(this.exp
enseLimits),
title, currentValue: 0, limit });
   this.updateAllocation();
}
updateExpenseLimit(id: number, value: number) {
   const limit = this.expenseLimits.find((limit) =>
```

```
   limit.id === id); if (!limit) throw new
   Error(`Expense with ID ${id} not found`);
   if (value < 0) throw new Error("Expense value cannot
   be negative"); limit.currentValue = value;
   this.updateAllocation();
}

// Savings
getSavings() {
   return [...this.savings];
}
addSavings(description: string, amount: number) {
    if (amount <= 0) throw new Error("Savings amount
must be greater than 0");
    this.savings.push({
id:this.generateId(this.savings), description,
amount });
    this.updateAllocation();
}

// Budget Allocation
getAllocation() {
   return [...this.allocation];
}
private updateAllocation() {
const totalIncome =
this.calculateTotal(this.incomeSources, "amount");
   const totalSavings =
   this.calculateTotal(this.savings, "amount");
    const totalExpenses
this.calculateTotal(this.expenseLimits,
```

```
"currentValue");

    this.allocation = [
      { id: "income", label: "Income", value:
      totalIncome, color: "#4caf50"
},
      {id:"savings",label: "Savings", value:
      totalSavings,color:
"#2196f3" },
      {id:"expenses", label: "Expenses", value:
      totalExpenses, color: "#f44336" },
    ];
}
```

Лістинг 3

```
  updateBill(id: number, updatedFields:
Partial<Omit<Bill["bills"][number], "id" | "userId">>)
{
    const bill = this.bills.find((bill) => bill.id ===
    id);
    if (!bill) throw new Error(`Bill with ID ${id} not
    found`); Object.assign(bill, updatedFields);
    this.updateAllocation();
  }
// Payment History
getPaymentHistory() {
  return [...this.paymentHistory];
}
recordPayment(billId: number, amount: number) {
  const bill = this.bills.find((bill) => bill.id ===
```

```
  billId);
  if (!bill) throw new Error(`Bill with ID ${billId}
  not found`);
  if (amount <= 0) throw new Error("Payment amount must
  be greater than 0");
  this.paymentHistory.push({
    id: this.generateId(this.paymentHistory),
    billId,
    date: new Date(),
    amount,
  });
  console.log(`Payment of ${amount} recorded for bill
"${bill.billName}"`);
}
// Scheduled Payments getScheduledPayments() {
return [...this.scheduledPayments];
}
schedulePayment(billId: number, scheduledDate: Date) {
const bill = this.bills.find((bill) => bill.id ===
billId);
  if (!bill) throw new Error(`Bill with ID ${billId}
  not found`);
  if (scheduledDate <= new Date()) throw new
  Error("Scheduled date must be in the future");
  this.scheduledPayments.push({
    id: this.generateId(this.scheduledPayments),
    billId,
    scheduledDate,
  });
  console.log(`Payment for  bill  "${bill.billName}"        sch
${scheduledDate}`);
```

```
}
// Notifications & AutoPay
toggleNotification(billId: number) {
  const bill = this.bills.find((bill) => bill.id ===
  billId);
  if (!bill) throw new Error(`Bill with ID ${billId}
  not found`);
  bill.notificationEnabled = !bill.notificationEnabled;
  console.log(
     `Notifications for bill "${bill.billName}" $
     { bill.notificationEnabled ? "enabled" : "disabled"
    }`
  );
}
toggleAutoPay(billId: number) {
  const bill = this.bills.find((bill) => bill.id ===
  billId);
  if (!bill) throw new Error(`Bill with ID ${billId}
  not found`);

  bill.autoPayEnabled = !bill.autoPayEnabled;
  console.log(
     `Autopay for bill "${bill.billName}" $
     { bill.autoPayEnabled ? "enabled" : "disabled"
     }`
  );
}
// Allocation
getAllocation() {
  return [...this.allocation];
}
```

```
private updateAllocation() {
  const totalAmount = this.calculateTotal(this.bills,
"amount");
  const totalPaymentsthis.calculateTotal
  this.paymentHistory, "amount");
this.allocation = [
     {id:"bills",label:"TotalBills",value:totalAmount,c
olor: "#ff9800" },
     { id: "payments", label: "Total Payments", value:
totalPayments, color: "#4caf50" },
  ];
}
```

Лістинг 4

```
makePayment(debtId: number, amount: number) {
    const debt = this.debts.find((d) => d.id ===
    debtId);
    if (!debt) throw new Error(`Debt with ID ${debtId}
    not found`);
    if (amount <= 0) throw new Error("Payment amount
    must be greater than 0");
    if (amount  >          debt.balance)        throw  newE
    debt.balance -= amount;

    this.paymentHistory.push({
      id: this.generateId(this.paymentHistory),
      debtId,
       paymentAmount: amount,
       paymentDate: new Date(),
    });
```

```
      this.updateAllocation();
      console.log(`Payment of ${amount} made to creditor
      "${debt.creditor}"`);
    }

calculateInterest(debtId: number): number {
  const debt = this.debts.find((d) => d.id === debtId);
  if (!debt) throw new Error(`Debt with ID ${debtId}
  not found`);

  return (debt.balance * debt.interestRate) / 100;
}

trackProgress(debtId: number): string {
  const debt = this.debts.find((d) => d.id === debtId);
  if (!debt) throw new Error(`Debt with ID ${debtId}
  not found`);

  return ((debt.minimumPayment / debt.balance) *
  100).toFixed(2);
}

// Debt Overview
generateDebtOverviewData() {
  return this.debts.map((debt) => ({
  id: debt.id,
   creditor: debt.creditor,
   debtType: debt.debtType,
   balance: debt.balance.toFixed(2),
   interestRate: `${debt.interestRate}%`,
```

```
   minimumPayment: debt.minimumPayment.toFixed(2),
   dueDate: debt.dueDate.toDateString(),
  }));
}

generatePaymentHistoryData() {
  return this.paymentHistory.map((payment) => {
    const debt = this.debts.find((d) => d.id ===
    payment.debtId);
    return {
      debtId: payment.debtId,
      creditor: debt?.creditor || "Unknown",
      paymentAmount: payment.paymentAmount.toFixed(2),
      paymentDate: payment.paymentDate.toDateString(),
     };
  });
}

// Debt Ratios
calculateDebtBurdenRatio(): string {
  const totalDebt = this.calculateTotal(this.debts,
  "balance");
  const totalMinimumPayment
  this.calculateTotal(this.debts, "minimumPayment");

  return ((totalMinimumPayment / totalDebt) *
  100).toFixed(2);
 }

calculateDebtToIncomeRatio(income: number): string {
  if (income <= 0) throw new Error("Income must be
```

```
    greater than 0");

    const totalDebt = this.calculateTotal(this.debts,
    "balance"); return ((totalDebt / income) *
    100).toFixed(2);
  }
```

Лістинг 5

```
  makePayment(debtId: number, amount: number) {
generateReport(dataSources: { budgets?: Budget[]; debts?:
  Debt[] }): void {
     try {
       if (!dataSources || (!dataSources.budgets && !
       dataSources.debts)) { throw new Error("No data
       sources provided to generate the report.");
       }

        if (this.reportType === "Budget Comparison" &&
        dataSources.budgets) {
          this.data = dataSources.budgets.map((budget) =>
          ({
             category: budget.category,
             planned: budget.amount,
             actual: budget.calculateRemaining(),
          }));
       } else if (
         this.reportType === "Debt Overview" &&
         dataSources.debts
       ) {
        this.data = dataSources.debts.map((debt) => ({
```

```
          creditor: debt.creditor,
          balance: debt.balance, minimumPayment:
          debt.minimumPayment,
        }));
      } else {
        throw new Error("Unsupported report type or
        missing data source.");
      }

      console.log(`${this.reportType} report generated
      successfully.`);
    } catch (error) {
      console.error(`Failed to generate report: $
      {error.message}`);
    }
  }

viewReport(): void {
    try {
      console.log(`${this.reportType} report viewed:`,
      this.data);
    } catch (error) {
      console.error(`Failed to view report: $
      {error.message}`);
    }
  }

exportReport(format: string): void {
    try {
      if (!["PDF", "CSV"].includes(format)) {
        throw new Error("Unsupported export format.");
```

```
    }
      console.log(`${this.reportType} report exported in
${format} format.`);
    } catch (error) {
    console.error(`Failed to export report: $
    {error.message}`);
   }
 }
}
```