Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему: «Розробка кросплатформного додатку управління фінансами»

Виконав: студент групи К23-1М

Спеціальність 122 Комп'ютерні науки

Вакуленко Д.Д.
(прізвище та ініціали)

Керівник д.е.н., проф. Корнєєв М.В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів
(місце роботи)

В.о. завідувача кафедри програмного забезпечення
(посада)

к.т.н., доц. Жульковський О.О.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Вакуленко Д.Д. Розробка кросплатформного додатку управління фінансами.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 122 «Комп’ютерні науки». – Університет митної справи та фінансів, Дніпро, 2025

Об'єктом дослідження – процес автоматизації управління фінансами із використанням сучасних інформаційних технологій.

Предмет дослідження – методи та засоби розробки кросплатформних застосунків для управління фінансами.

Метою роботи є розробка кросплатформного додатку для управління фінансами, який забезпечує інтерактивний інтерфейс, підтримку сучасних технологій обробки даних і можливість адаптації до потреб користувачів.

Кваліфікаційна робота присвячена створенню кросплатформного додатку для управління фінансами. У роботі проведено аналіз сучасних інструментів для розробки додатків, обґрунтовано вибір технологій, зокрема використання Unity та C#. Запропонований додаток забезпечує можливість обліку доходів і витрат, аналізу фінансових операцій, створення бюджетів та прийняття обґрунтованих фінансових рішень. Проведено тестування програмного продукту, яке підтвердило стабільність роботи системи та відповідність функціональним вимогам.

Практична цінність роботи полягає у створенні програмного продукту, який дозволяє користувачам ефективно управляти своїми фінансами, планувати бюджет та оптимізувати витрати.

Ключові слова: кросплатформний додаток, управління фінансами, Unity, C#, автоматизація, бюджетування.

ABSTRACT

Vakulenko D.D. Development of a Cross-Platform Application Financial Management.

This project for obtaining a master's degree in speciality 122 "Computer Science." - University of Customs and Finance, Dnipro, 2025.

Object of research – the process of automating financial management using modern information technologies.

Subject of research – methods and tools for developing cross-platform applications for financial management.

Purpose of the work is to develop a cross-platform application for financial management that provides an interactive interface, supports modern data processing technologies, and adapts to user needs.

The master's thesis is dedicated to the development of a cross-platform application for financial management. The work includes an analysis of modern tools for application development, justification of the choice of technologies, particularly the use of Unity and C#. The proposed application enables users to manage income and expenses, analyze financial operations, create budgets, and make informed financial decisions. The application has been tested, confirming its system stability and compliance with functional requirements.

Practical significance of the work lies in the creation of a software product that allows users to effectively manage their finances, plan budgets, and optimize expenses.

Keywords: cross-platform application, financial management, Unity, C#, automation, budgeting.

ЗМІСТ

ВСТУП

Сучасне суспільство неможливо уявити без ефективного управління фінансами, яке стає важливим аспектом як для індивідуального добробуту, так і для економічної стабільності суспільства в цілому. З розвитком інформаційних технологій з’являються нові можливості для автоматизації фінансових процесів, що дозволяє спростити облік доходів і витрат, оптимізувати управління ресурсами та приймати обґрунтовані фінансові рішення.

Актуальність даного дослідження зумовлена необхідністю створення зручних і функціональних рішень для особистого фінансового менеджменту. У сучасних умовах, коли фінансові операції виконуються швидко та в значних обсягах, використання мобільних і кросплатформених застосунків стає оптимальним вибором для підвищення ефективності управління фінансами.

Метою роботи – розробка кросплатформеного додатку для управління фінансами, який забезпечує інтерактивний інтерфейс, підтримку сучасних технологій обробки та зберігання даних, а також можливість адаптації до потреб користувачів.

Методи дослідження – аналіз сучасних програмних засобів для розробки кросплатформених застосунків, моделювання структури додатку, методи тестування і оптимізації програмного забезпечення.

Основні завдання дослідження:

1. Проаналізувати існуючі програмні рішення для управління фінансами.
2. Обґрунтувати вибір технологій для розробки кросплатформеного додатку.
3. Розробити структуру та функціонал додатку.
4. Реалізувати додаток та провести його тестування.

Об’єктом дослідження є процес автоматизації управління фінансами із використанням сучасних інформаційних технологій.

Предметом дослідження є програмні засоби та методи розробки кросплатформених застосунків для управління фінансами.

Практична значимість роботи полягає у створенні програмного продукту, який надасть користувачам зручний інструмент для обліку доходів і витрат, аналізу фінансових операцій, планування бюджету та прийняття обґрунтованих фінансових рішень.

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 73 сторінок тексту, 49 рисунків, 2 таблиць та переліку літературних джерел з 15 найменувань.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

В епоху глобалізації та стрімкого розвитку інформаційних технологій, фінансові ринки та економічні системи стають дедалі складнішими та динамічнішими. У цьому контексті, фінансова грамотність перетворюється з корисного навику на життєво необхідну компетенцію для кожного члена суспільства. Розуміння основних принципів функціонування фінансової системи, вміння ефективно управляти власними коштами та приймати обґрунтовані фінансові рішення стають запорукою індивідуального добробуту та економічної стабільності суспільства.

Фінансова грамотність – це комплекс знань, навичок, установок та поведінкових моделей, необхідних для прийняття ефективних фінансових рішень у різних життєвих ситуаціях. Вона охоплює розуміння основних фінансових концепцій, таких як бюджетування, заощадження, інвестування, кредитування, оподаткування, страхування та управління ризиками. Організація економічного співробітництва та розвитку визначає фінансову грамотність як "поєднання фінансової обізнаності, знань, навичок, ставлення та поведінки, необхідних для прийняття обґрунтованих фінансових рішень і, зрештою, досягнення індивідуального фінансового добробуту" [1].

Фінансова грамотність відіграє ключову роль у забезпеченні фінансового благополуччя як окремих осіб, так і суспільства в цілому. Для індивіда володіння фінансовими знаннями та навичками означає підвищення рівня життя, зниження ризику фінансових труднощів, прийняття обґрунтованих інвестиційних рішень та ефективне планування майбутнього.

На рівні суспільства підвищення фінансової грамотності населення сприяє економічному зростанню, зменшенню соціальної нерівності, підвищенню стабільності фінансової системи та зменшенню навантаження на систему соціального забезпечення.

Незважаючи на очевидну важливість фінансової грамотності, рівень фінансової обізнаності населення в багатьох країнах, включаючи Україну, залишається недостатнім. Дослідження свідчать, що значна частина громадян не володіє базовими знаннями про фінансові продукти та послуги, не вміє складати особистий бюджет та планувати витрати, схильна до імпульсивних фінансових рішень. Фінансова неграмотність може призводити до надмірної заборгованості, шахрайства та фінансових зловживань, неефективного інвестування та відсутності фінансового планування.

Особистий фінансовий менеджмент – це системний підхід для управління фінансовими ресурсами індивіда або сім'ї, що передбачає аналіз поточного фінансового стану, планування доходів і витрат, постановку фінансових цілей та розробку стратегії їх досягнення. До основних інструментів належать:

- бюджетування,
- управління заощадженнями,
- інвестування,
- управління кредитами,
- страхування
- податкове планування.

Прийняття фінансових рішень часто супроводжується емоційними та психологічними факторами, які можуть впливати на раціональність вибору. Тому, важливим аспектом фінансового менеджменту є розвиток самодисципліни, емоційного інтелекту та вміння приймати зважені рішення.

Підвищення рівня фінансової грамотності населення потребує комплексного підходу, що включає інтеграцію фінансової грамотності в систему освіти, створення спеціалізованих навчальних програм та проведення просвітницьких кампаній.

Сучасні технології, такі як мобільні додатки, онлайн-банкінг та спеціалізовані веб-сервіси, значно спрощують процес управління особистими фінансами, роблячи його більш прозорим, зручним та доступним.

1.2 Аналіз аналогів представлених на ринку

Фінансові додатки для особистого обліку почали використовувати задовго до появи сучасних смартфонів. Сьогодні вони стали невід'ємною частиною управління особистими фінансами. На ринку представлені як прості програми для фіксації доходів і витрат, так і багатофункціональні інструменти, що дозволяють аналізувати інвестиції, відстежувати курс акцій і створювати складні фінансові звіти [2].

Хоча багатофункціональні додатки надають значні можливості для управління фінансами, вони мають і певні недоліки. Наприклад:

- Висока вартість. Багато з таких інструментів доступні лише на платній основі або містять внутрішні покупки.
- Складність у використанні. Вони часто орієнтовані на досвідчених користувачів, що може відлякати новачків.
- Зручність на мобільних пристроях. Багатий функціонал не завжди адаптований для комфортного використання на смартфонах або планшетах [9].

Для розуміння особливостей таких додатків їх можна класифікувати за категоріями. Наприклад, у категорії «Фінанси» виділяють програми для бухгалтерського обліку, управління витратами, інвестиційного аналізу тощо.

1.2.1 AndroMoney

AndroMoney є одним із найпопулярніших додатків у Google Play з високою оцінкою користувачів.

Програма вирізняється інтуїтивно зрозумілим інтерфейсом і багатим функціоналом, який включає (рис. 1.1):

- Облік фінансових операцій.
- Перегляд детальних звітів і аналіз витрат.

- Створення бюджетів для різних періодів (день, тиждень, місяць, рік) із можливістю встановлення лімітів на окремі категорії. У разі перевищення бюджету надходять попередження.
- Синхронізацію даних із хмарними сервісами (Dropbox, Google Drive).

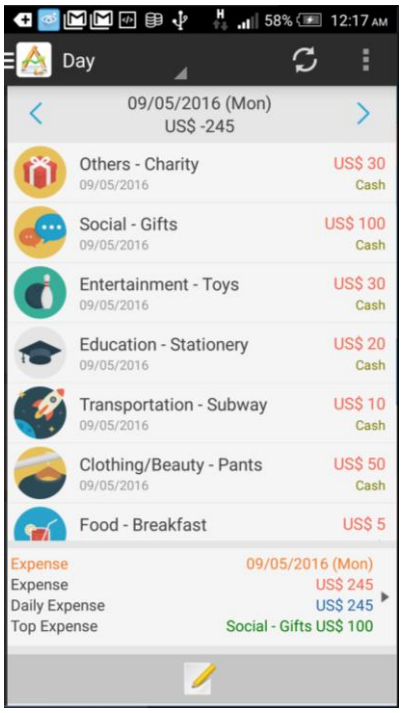

Рисунок 1.1 – AndroMoney

Особливості:

- Гнучке налаштування категорій і валют.
- Можливість переносу даних через імпорт/експорт.
- Відображення транзакцій між рахунками.

Переваги:

- Простота використання навіть для новачків.
- Наявність широкого функціоналу для аналізу даних.
- Підтримка хмарних сервісів для збереження даних.

Недоліки:

- Відсутність української мови.

- Рекламні банери (можливість видалення лише за додаткову плату).
- Застарілий інтерфейс, який може не відповідати сучасним стандартам UX/UI.
- Надлишковість функцій для базових користувачів [10].

1.2.2 Expense Manager

Expense Manager орієнтований на простоту та базове управління фінансами. Додаток, завантажений понад мільйон разів, пропонує основні функції для обліку витрат та доходів (рис. 1.2).

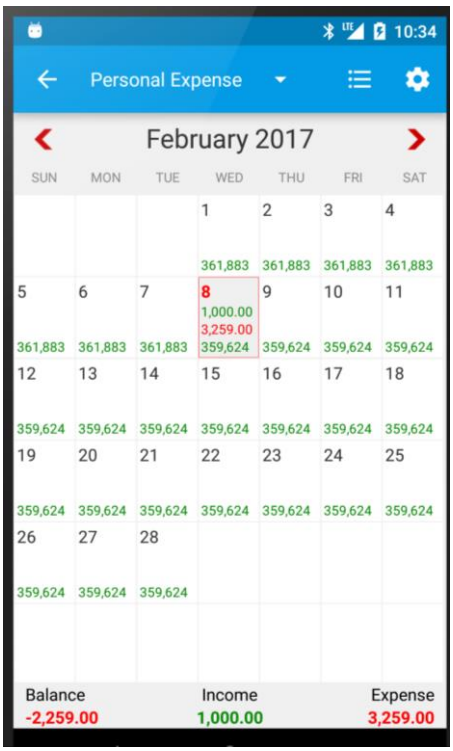

Рисунок 1.2 – Expense Manager

Функціонал:

- Попередньо встановлені категорії витрат і доходів, які можна налаштовувати.
- Відображення графіків витрат за певні періоди.
- Можливість налаштування повторюваних платежів.

Переваги:

- Інтуїтивно зрозумілий інтерфейс.
- Орієнтація на користувачів-початківців.
- Мінімум зайвих функцій.

Недоліки:

- Обмежені можливості для детального аналізу.
- Відсутність підтримки складних функцій, таких як інвестиційний облік [11].

1.2.3 Витрати

Ця програма дозволяє вести облік сімейних витрат за допомогою категорій та тегів. Користувачі можуть створювати та редагувати категорії витрат, переглядати детальну статистику за різні періоди (день, тиждень, місяць, рік) та змінювати інформацію в історії операцій (1.3).

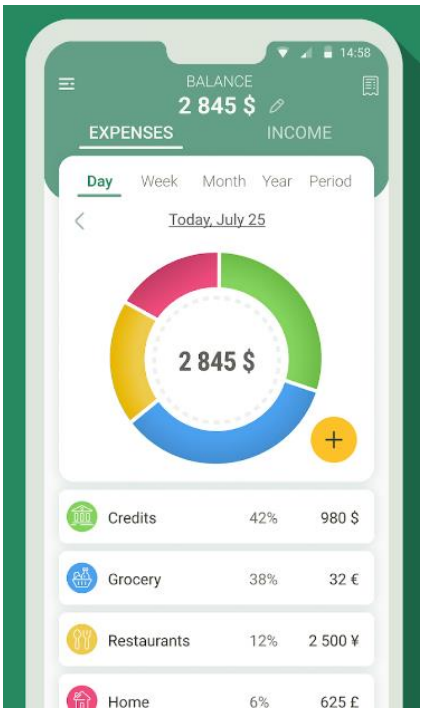

Рисунок 1.3 – Витрати

Переваги:

- Простота використання для базових завдань.
- Можливість адаптації категорій під особисті потреби.

Недоліки:

- Облік лише витрат, без функції додавання доходів чи заощаджень.
- Відсутність української мови.
- Застарілий інтерфейс та дрібний шрифт у статистиці.
- Відсутність графіків для аналізу даних.
- Додаток не оновлюється понад рік, що свідчить про зниження підтримки розробниками [15].

1.2.4 FinancePM

Finance PM пропонує значно ширший функціонал у порівнянні з базовими додатками. Програма дозволяє не тільки вести облік доходів і витрат, а й керувати боргами, планувати транзакції та налаштовувати ліміти для окремих категорій (рис. 1.4).

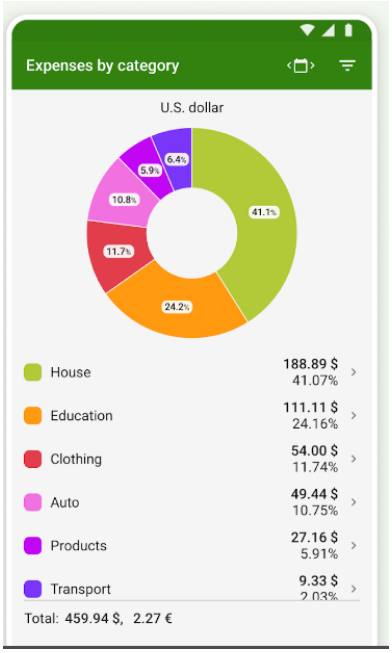

Рисунок 1.4 – FinancePM

Унікальні особливості:

- Можливість створення необмеженої кількості гаманців у різних валютах.
- Планування фінансових операцій із гнучким налаштуванням періодичності (щоденні, щомісячні, індивідуальні дні тижня чи місяця).
- Наявність шаблонів для регулярних операцій, що економить час.

Переваги:

- Гнучкість у налаштуваннях.
- Зручний функціонал для прогнозування витрат і доходів.
- Можливість налаштування інтерфейсу відповідно до особистих вподобань.

Недоліки:

- Інтерфейс не завжди інтуїтивно зрозумілий для новачків.
- Відсутність української мови [12].

1.2.5 Toshl Finance

Цей додаток вирізняється інтерактивністю та незвичайним дизайном, завдяки «помічнику-монстру», який мотивує користувачів (рис. 1.5).

Програма дозволяє кожному дню присвоювати окрему сторінку для фінансових операцій.

Основні функції:

- Облік доходів і витрат за допомогою міток замість стандартних категорій.
- Експорт звітів у PDF, Excel і Google Docs.
- Налаштування бюджетів та синхронізація з сервером.

Переваги:

- Інтерактивний дизайн, що спрощує користування.
- Можливість створювати необмежену кількість бюджетів у преміум-версії.

Недоліки:

- Відсутність української мови.
- Платна підписка ($1,99 на місяць або $19,99 на рік).
- Дизайн може здаватися надто «дитячим» для професійних користувачів.

Рисунок 1.5 – Toshl Finance

1.2.6 Monefy

Міжнародний додаток для обліку особистих фінансів із можливістю створення транзакцій, акаунтів і перегляду історії (рис. 1.6).

Особливості:

- Відображення інформації про транзакції за поточний місяць у вигляді діаграми.
- Локальне збереження даних із можливістю експорту.

Переваги:

- Простий інтерфейс із функціями базового обліку.

- Дані не передаються на сторонні сервіси, що підвищує рівень конфіденційності.

Недоліки:

- Відсутність веб-версії.
- Немає синхронізації з банками [14].

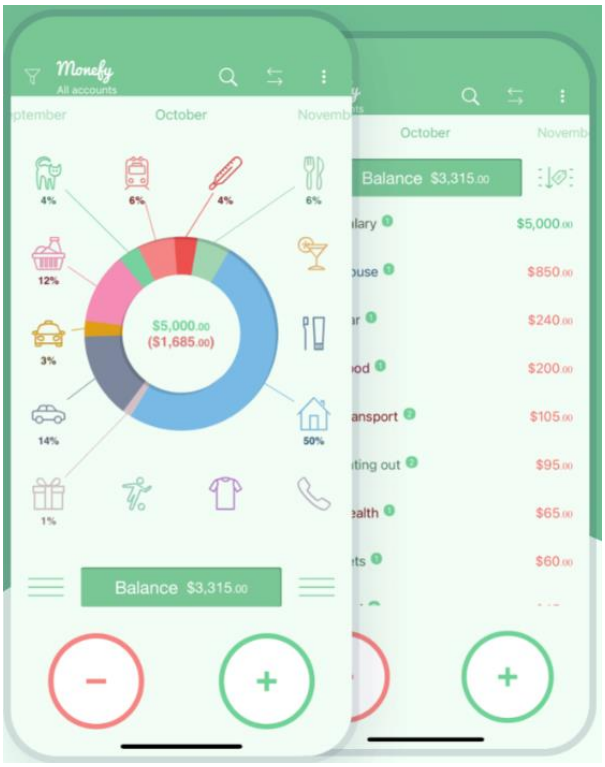

Рисунок 1.6 – Monefy

1.2.7 CoinKeeper: Budget Planner

CoinKeeper — це мобільний додаток, орієнтований переважно на ринок США, який надає користувачам можливість відстежувати та аналізувати свої фінансові потоки. Застосунок вирізняється стильним інтерфейсом і якісно реалізованою системою обліку витрат. Користувачі можуть встановлювати фінансові цілі та аналізувати дані для ефективного управління своїми фінансами [12].

На головному екрані представлена інформація про акаунти, категорії витрат та звітність. Хоча дизайн програми отримує схвальні відгуки, вона не

підтримує синхронізацію з банківськими рахунками, що може обмежувати її функціональність (рис. 1.7).

Рисунок 1.7 CoinKeeper

На головному екрані представлена інформація про акаунти, категорії витрат та звітність. Хоча дизайн програми отримує схвальні відгуки, вона не підтримує синхронізацію з банківськими рахунками, що може обмежувати її функціональність.

Основні переваги:

- Стильний дизайн із сучасними аналітичними елементами.
- Підтримка широкого спектра валют.

Основні недоліки:

- Доступний виключно як мобільний застосунок.
- Відсутність функції синхронізації з банками.
- Фокус на аналізі витрат без інтеграції функцій для доходів чи заощаджень [13].

1.2.8 Splitwise

Splitwise — це популярний додаток, доступний у веб-версії, а також у Google Play та App Store. Його ключова функція — полегшення управління спільними витратами в групах. Користувачі можуть створювати групи, додавати друзів і ділити витрати за різними сценаріями, що робить додаток ідеальним для подорожей, спільного проживання чи групових покупок (рис. 1.8).

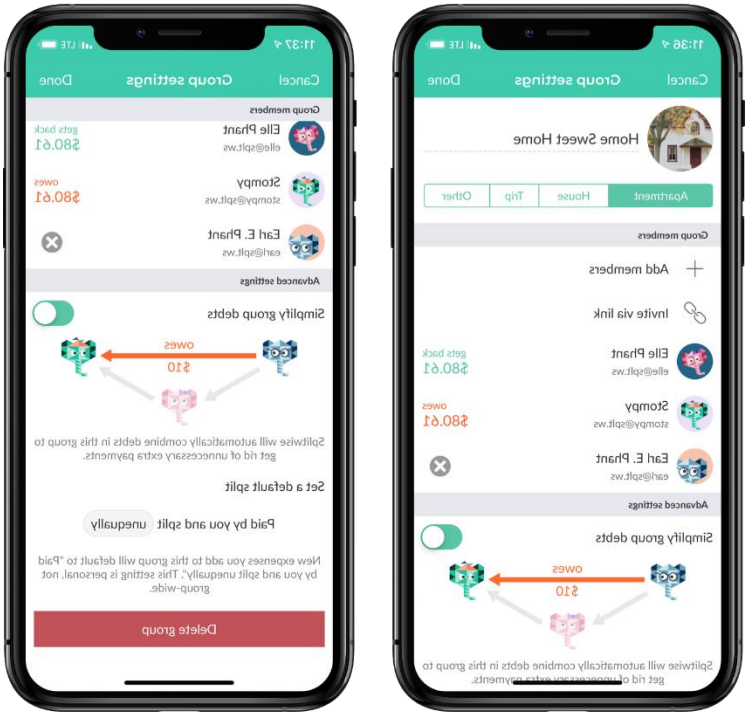

Рисунок 1.8 – Splitwise

Функціональні можливості:

- Розподіл витрат і боргів між членами групи.
- Гнучке налаштування транзакцій із можливістю розподілу за відсотками.
- Обчислення загального балансу групи.
- Пропозиції щодо оптимізації витрат і спрощення боргів.
- Налаштування повторюваних витрат.

- Підтримка роботи в офлайн-режимі для зручності в умовах відсутності доступу до мережі [12].

Унікальні переваги:

- Наявність веб-версії, що дозволяє працювати з додатком із будь-якого пристрою.
- Зручність для спільного використання у різних життєвих сценаріях, таких як подорожі чи ведення домашнього бюджету з партнерами.
- Можливість інтеграції з іншими платформами для автоматичного розрахунку та звітності.

1.2.8 Порівняння

У сучасному світі управління фінансами стає дедалі важливішим, а мобільні додатки відіграють ключову роль у цьому процесі. Вони допомагають користувачам організовувати свої витрати, відстежувати доходи, керувати бюджетом і навіть планувати заощадження.

Нижче представлена таблиця 1.1, яка порівнює основні додатки для фінансового менеджменту за їхніми функціями, перевагами, недоліками, доступними платформами та цінами. Це допоможе вибрати оптимальний інструмент відповідно до ваших потреб і фінансових цілей.

Таблиця 1.1 – Порівняння застосунків фінансового менеджменту

Додаток	Основні функції	Переваги	Недоліки	Платформа	Ціна
AndroMoney	Облік фінансів, створення бюджетів, експорт/імпорт даних	Гнучке налаштування, підтримка хмарних сервісів	Відсутність української мови, рекламні банери	Мобільний	Безкоштовний (з рекламою)
Expense Manager	Облік витрат і доходів,	Простий інтерфейс, мінімалізм	Обмежений функціонал,	Мобільний	Безкоштовний

	графіки витрат		відсутність підтримки інвестицій		
Finance PM	Управління боргами, бюджетами, шаблони транзакцій	Розширений функціонал, шаблони для повторюваних витрат	Складний для новачків, відсутність української мови	Мобільний	Безкоштовний
Toshl Finance	Мітки для витрат, бюджетування, експорт звітів	Інтерактивність, експорт у PDF/Excel	Платна підписка, відсутність української мови	Мобільний	Підписка ($1,99/місяць)
Monefy: Money Tracker	Створення транзакцій, локальне збереження даних	Простота, конфіденційність даних	Відсутність веб-версії, немає синхронізації з банками	Мобільний	Безкоштовний
CoinKeeper	Відстеження витрат, фінансові цілі, звітність	Стильний дизайн, підтримка багатьох валют	Немає синхронізації з банками, лише мобільна версія	Мобільний	Безкоштовний
Splitwise	Розподіл витрат у групах, балансування боргів	Підтримка веб-версії, зручність для групових витрат	Немає функцій для індивідуальних фінансів	Мобільний, Веб	Безкоштовний (основний функціонал)

1.3 Висновок до першого розділу

У даному розділі було проведено дослідження предметної області управління фінансами, включаючи ключові концепції, сучасні виклики та підходи до оптимізації фінансових процесів. Розглянуто основні принципи фінансової грамотності, такі як бюджетування, заощадження, інвестування,

кредитування та управління ризиками. Проаналізовано значення фінансової грамотності на індивідуальному та суспільному рівнях, а також її вплив на економічну стабільність.

Особливу увагу приділено аналізу існуючих рішень для управління фінансами, таких як AndroMoney, Expense Manager, Finance PM, Toshl Finance та інших. Вивчено функціональні можливості цих додатків, їх переваги, такі як інтуїтивно зрозумілий інтерфейс, багатий функціонал та підтримка хмарних сервісів, а також обмеження, включаючи складність для новачків, відсутність української мови та застарілий дизайн.

Метою дослідження є розробка кросплатформної системи управління фінансами.

Для досягнення поставленої мети було виконано такі завдання:

1. Проведено аналіз предметної області управління фінансами та її значення.
2. Визначено ключові принципи фінансової грамотності та її вплив на економічну стабільність.
3. Проаналізовано сучасні програмні рішення для управління фінансами.
4. Проаналізовано технології розробки кросплатформних додатків.
5. Розроблено кросплатфорний застосунок управління фінансами
6. Зібрвно та протестовано систему під операційною системою Android

Вхідними даними для аналізу стали концепції фінансового менеджменту, переваги та обмеження існуючих інструментів, а також потреби користувачів у зручних і ефективних рішеннях для управління особистими фінансами.

РОЗДІЛ 2. ПРОЕКТУВАННЯ КРОСПТФОРМЕННОГО ДОДАТКУ

2.1 Середовище розробки кросплатформенного додатку

Сучасний світ мобільних та стаціонарних технологій характеризується значним різноманіттям пристроїв та операційних систем. Розробка кросплатформенних додатків є однією з ключових тенденцій у сфері програмування, що зумовлена необхідністю створення програмного забезпечення, яке працює на різних платформах, таких як Windows, macOS, Android, iOS та інших. Актуальність розробки таких додатків полягає у можливості забезпечити доступність, економію ресурсів та широку аудиторію користувачів за допомогою єдиного кодової бази.

Кросплатформенні додатки дозволяють компаніям знизити витрати на розробку, оскільки створення окремих додатків для кожної платформи потребує значних фінансових і часових витрат. Замість цього розробники можуть використовувати єдиний кодовий базис, який компілюється або інтерпретується для роботи на різних пристроях. Такий підхід також сприяє швидшому випуску оновлень, оскільки зміни потрібно вносити лише в одну кодову базу [6].

Одним з основних підходів до створення кросплатформенних додатків є використання технологій, які дозволяють створювати програмне забезпечення з високою продуктивністю та нативним інтерфейсом. Серед найбільш популярних підходів і технологій варто відзначити наступні:

- Гібридні додатки (Hybrid Applications): Цей підхід передбачає використання веб-технологій (HTML, CSS, JavaScript) для створення інтерфейсу додатку, який працює всередині нативного контейнера. Такі додатки можуть використовувати фреймворки, як-от Apache Cordova або Ionic. Гібридні додатки забезпечують швидкий розвиток і є економічно вигідними, однак можуть мати обмежену продуктивність у порівнянні з нативними додатками.

- Нативні кросплатформенні додатки (Native Cross-Platform Applications): Технології, як-от Xamarin або .NET MAUI, дозволяють писати код мовами, такими як C# або F#, і компілювати його у нативний код для кожної платформи. Це забезпечує високу продуктивність і можливість використовувати нативні API кожної платформи. Xamarin, наприклад, широко використовується для створення потужних мобільних додатків із нативним виглядом [5].
- Кросплатформенні додатки на базі фреймворків (Cross-Platform Framework-Based Applications): Фреймворки, такі як React Native, Flutter або Unity, дозволяють створювати кросплатформенні додатки, які поєднують високу продуктивність і зручність розробки. React Native використовує JavaScript для створення інтерфейсів, а Flutter — Dart. Unity, хоча і спочатку розроблений для створення ігор, також підходить для побудови інтерактивних бізнес-додатків.
- Прогресивні веб-додатки (Progressive Web Applications, PWA): Цей підхід дозволяє створювати додатки, які працюють як звичайні веб-сайти, але можуть запускатися як додатки на мобільних пристроях і десктопах. Вони мають обмежений доступ до нативних функцій пристрою, однак є легкими у розробці та не потребують встановлення.

Кожен із підходів має свої переваги та недоліки. Наприклад, гібридні додатки є швидкими у розробці, однак можуть мати обмеження щодо продуктивності. Нативні кросплатформенні рішення, такі як Xamarin, дозволяють створювати продуктивні додатки, однак вимагають певних навичок роботи з мовами програмування, такими як C#. React Native та Flutter пропонують чудовий баланс між продуктивністю та зручністю розробки.

Обираючи середовище розробки, важливо враховувати специфіку проекту, аудиторію користувачів, необхідні функції та бюджет. Наприклад, для проектів, які вимагають високої продуктивності, рекомендується використовувати Xamarin або Flutter. Якщо ж додаток повинен бути легким та

доступним для багатьох користувачів, PWA або React Native можуть бути оптимальним вибором.

У результаті використання сучасних технологій для розробки кросплатформенних додатків дозволяє розробникам створювати продукти, які відповідають вимогам ринку, забезпечують економію ресурсів та доступ до широкої аудиторії. Такі рішення є важливим кроком у розвитку цифрових технологій і задоволенні потреб користувачів на глобальному рівні.

2.2 Особливості розгортання додатків на операційних системах

Взаємодія з різними платформами та процес розгортання є важливими аспектами розробки кросплатформенних додатків, оскільки саме від них залежить можливість додатку працювати на різноманітних пристроях та операційних системах. Основною метою взаємодії є забезпечення уніфікованого досвіду для користувачів, незалежно від платформи, а також забезпечення стабільності, продуктивності та доступу до нативних функцій кожного пристрою.

Кросплатформенні додатки розробляються з урахуванням особливостей кожної платформи. Наприклад, операційна система Android вимагає роботи з середовищем Android Studio для створення APK-файлів і використання інструментів, таких як ADB (Android Debug Bridge) для тестування. У той же час iOS-додатки потребують використання Xcode для створення IPA-файлів, а також сертифікатів і профілів для підпису додатку. Технології, як-от Xamarin або Flutter, дозволяють розробникам використовувати єдину кодову базу для створення додатків, які враховують ці специфічні вимоги.

Важливим аспектом взаємодії з платформами є доступ до нативних API. Завдяки цьому додатки можуть використовувати функції пристрою, такі як GPS, камера, сенсори або push-сповіщення. Наприклад, React Native забезпечує доступ до таких функцій за допомогою бібліотек, як-от react-native-geo-location або react-native-push-notification. У Flutter це реалізовано через

плагіни, а Xamarin пропонує використання DependencyService для платформи-специфічного коду.

Процес розгортання кросплатформенних додатків включає кілька етапів. Спочатку виконується підготовка додатку до збірки. Це включає оптимізацію ресурсоємності, адаптацію до різних розмірів екранів та тестування. Після цього додаток компілюється у формат, який підтримує цільова платформа. Для Android це APK або AAB-файли, для iOS — IPA-файли, а для веб-додатків — оптимізовані HTML, CSS та JavaScript-файли.

Наступним етапом є розгортання додатків на платформах розповсюдження. У випадку Android це Google Play Console, яка дозволяє завантажувати та керувати додатками в Google Play. Для iOS розгортання виконується через App Store Connect. Обидві платформи передбачають процес перевірки додатків на відповідність політикам та стандартам якості. Для веб-додатків розгортання зазвичай здійснюється через хостинг-платформи, такі як AWS, Firebase чи Netlify.

Для спрощення процесу розгортання існують інструменти CI/CD (безперервної інтеграції та доставки), такі як GitHub Actions, Bitrise або Jenkins. Вони автоматизують збірку, тестування та розгортання додатків, що значно скорочує час і зменшує кількість помилок. Наприклад, налаштування CI/CD дозволяє автоматично створювати збірки для Android та iOS кожного разу, коли розробник вносить зміни до репозиторію.

Ще одним важливим аспектом є підтримка оновлень. Завдяки інструментам, як-от CodePush (для React Native) або Over-the-Air Updates (для Expo), розробники можуть швидко поширювати оновлення без необхідності повторного завантаження додатку у магазини додатків. Це дозволяє оперативно виправляти помилки, додавати новий функціонал і покращувати досвід користувача.

Таким чином, взаємодія з різними платформами та процес розгортання вимагають врахування специфічних вимог кожної платформи, інтеграції з нативними API та використання сучасних інструментів для автоматизації

процесів. Використання кросплатформенних технологій дозволяє спростити ці процеси, забезпечуючи швидкість розробки, надійність роботи додатків і широкий доступ для користувачів.

2.3 Програмні засоби розробки

Розробка кросплатформенних додатків потребує ефективних програмних засобів, які дозволяють створювати програмне забезпечення для різних платформ із використанням спільної кодової бази. Ці засоби надають інструменти, фреймворки та середовища розробки, які забезпечують високу продуктивність, зручність роботи та доступ до нативних функцій пристроїв. Сучасні програмні засоби охоплюють широкий спектр рішень, серед яких розробники можуть обирати залежно від потреб проекту. Нижче наведено опис найпопулярніших засобів і їхніх можливостей.

Visual Studio є потужним середовищем розробки, яке підтримує створення кросплатформенних додатків за допомогою Xamarin та .NET MAUI. Xamarin (рис. 2.1) дозволяє писати код на C# та компілювати його у нативний код для iOS, Android і Windows. Ця технологія забезпечує доступ до нативних API та дозволяє створювати додатки з високою продуктивністю та нативним виглядом.

Рисунок 2.1 – Xamarin

.NET MAUI (Multi-platform App UI) є еволюцією Xamarin Forms і пропонує єдиний API для створення інтерфейсів користувача на різних платформах. Використання .NET MAUI (рис. 2.2) спрощує роботу з інтерфейсами, оскільки вся структура додатку організована в одному проекті. Visual Studio також надає інтегровані інструменти для налагодження, тестування та розгортання додатків [6].

Рисунок 2.2 – .NET MAUI

Flutter — це фреймворк від Google, який використовує мову Dart для створення кросплатформенних додатків. Його ключовою особливістю є використання власного рендерингового механізму, який дозволяє створювати інтерфейси користувача, що виглядають однаково на різних платформах. Flutter підтримує гаряче перезавантаження (hot reload), що значно пришвидшує процес розробки, дозволяючи розробникам одразу бачити результати змін у коді.

Flutter (рис. 2.3) має багатий набір віджетів, які можна налаштовувати, та підтримує інтеграцію з нативними модулями. Це робить його ідеальним вибором для створення додатків із високим рівнем персоналізації інтерфейсу [8].

Рисунок 2.3 – Flutter

React Native — це популярний фреймворк від Facebook, який використовує JavaScript для створення додатків із нативним виглядом і поведінкою. React Native базується на ідеї використання компонентів для побудови інтерфейсу, що спрощує розробку, налагодження та повторне використання коду.

Цей фреймворк забезпечує доступ до нативних API за допомогою модулів або плагінів і дозволяє розробляти додатки для iOS та Android одночасно. React Native відомий своєю продуктивністю, багатою екосистемою та підтримкою великої спільноти розробників.

Хоча Unity переважно асоціюється з розробкою ігор, ця платформа також використовується для створення кросплатформенних додатків, особливо інтерактивних або графічно насичених. Unity використовує мову C# і підтримує експорт на десятки платформ, включаючи мобільні пристрої, ПК та ігрові консолі.

Unity пропонує потужні інструменти для роботи з 3D та 2D-графікою, фізичними моделями та анімаціями, що робить її ідеальним вибором для додатків, які вимагають високого рівня візуалізації [4].

Apache Cordova (рис. 2.4) — це інструмент для створення гібридних додатків, який дозволяє використовувати веб-технології, такі як HTML, CSS і

JavaScript. Ці додатки працюють у нативному контейнері, що забезпечує доступ до функцій пристрою через спеціальні плагіни.

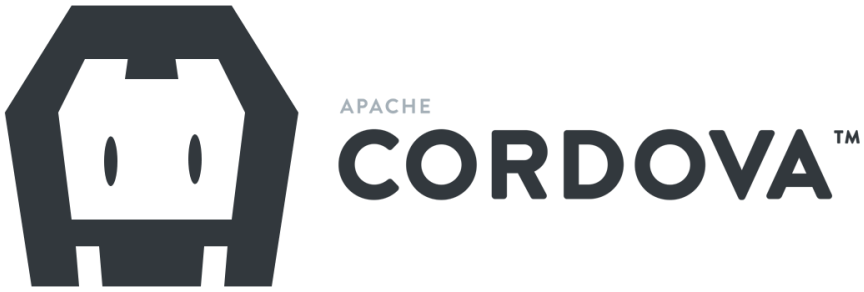

Рисунок 2.4 – Apache Cordova

Cordova є популярним вибором для швидкої розробки додатків із базовим функціоналом, але має обмеження щодо продуктивності та складності графіки.

PWA — це технологія для створення веб-додатків, які можуть працювати як нативні. Вони підтримують офлайн-режим, push-сповіщення і встановлення на головний екран пристрою. Хоча PWA не потребує додаткових інструментів для розробки, такі платформи, як Firebase, можуть значно спростити процес створення й розгортання.

Electron використовується для створення настільних кросплатформенних додатків із використанням JavaScript, HTML і CSS. Ця платформа дозволяє створювати потужні програми для Windows, macOS і Linux, які виглядають та працюють однаково на всіх платформах. Приклади додатків на Electron — Slack, Visual Studio Code і Discord [7].

Вибір програмного засобу залежить від потреб проекту. Для простих гібридних додатків підійдуть Cordova або PWA. Якщо потрібна висока продуктивність і гнучкість, ідеальними варіантами будуть Flutter або React Native. Для проектів із багатою графікою та складною логікою варто обирати Unity або Electron (табл. 2.1).

Таблиця 2.1 – Порівняльна таблиця програм розробки

Інструмент	Мова програмування	Найкраще підходить для	Переваги	Недоліки
Visual Studio з Xamarin/.NET MAUI	C#, .NET	Високопродуктивні додатки з нативним виглядом	Повний доступ до нативних API, потужне IDE, висока продуктивність	Потрібен досвід з C#, складна крива навчання
Flutter	Dart	Налаштовувані інтерфейси, швидка розробка	Hot reload, багата бібліотека UI, підтримка Google	Потрібно вивчити Dart, продуктивність може варіюватися для складних додатків
React Native	JavaScript	Нативні додатки з великою екосистемою	Велика спільнота, багаторазове використання компонентів, плагіни	Може бути повільним для складної графіки, потребує нативних модулів
Unity	C#	Інтерактивні додатки та графічно насичені ігри	Потужні інструменти графіки, підтримка багатьох платформ	Надмірний для простих додатків, високі ресурси
Apache Cordova	HTML, CSS, JavaScript	Гібридні додатки з базовим функціоналом	Легке налаштування, знайомі веб-технології	Обмежена продуктивність, не підходить для ресурсоємних додатків

Прогресивні веб-додатки (PWA)	HTML, CSS, JavaScript	Веб-додатки з офлайн-режимом	Легкість розгортання, працює на всіх платформах	Обмежена інтеграція з нативними функціями
Electron	HTML, CSS, JavaScript	Десктопні додатки з великою функціональністю	Єдина розробка для всіх платформ, велика функціональність	Високе споживання ресурсів, не підходить для легких додатків

Програмні засоби для розробки кросплатформенних додатків постійно вдосконалюються, забезпечуючи все більше можливостей для розробників. Завдяки їм стає можливим створення універсальних продуктів, які відповідають вимогам сучасних користувачів та пристроїв.

В якості середовища розробки було обрано Unity. Unity є одним із найпопулярніших і найбільш універсальних інструментів для створення кросплатформенних додатків (рис. 2.5). Спочатку розроблений як платформа для створення ігор, Unity на сьогодні використовується також для побудови інтерактивних додатків, симуляцій, візуалізації даних і навіть бізнес-рішень. Його універсальність, потужні інструменти та можливості кросплатформенності роблять Unity непоганим вибором для сучасних розробників.

Рисунок 2.5 - Unity

Однією з основних переваг Unity є його підтримка понад 25 платформ, серед яких Windows, macOS, Android, iOS, WebGL, Linux та ігрові консолі (рис. 2.6). Це дозволяє розробникам створювати один проект і експортувати його для різних пристроїв, що значно знижує витрати на розробку і забезпечує широку аудиторію користувачів. Unity також забезпечує доступ до нативних функцій кожної платформи через плагіни або API, що дозволяє використовувати всі можливості пристроїв.

Ще однією ключовою перевагою є інтуїтивний графічний інтерфейс Unity Editor. Цей інструмент дозволяє розробникам візуально створювати сцени, додавати компоненти та налаштовувати взаємодію об'єктів. Завдяки інтегрованій підтримці для сценаріїв на мові C#, Unity забезпечує високу гнучкість у розробці логіки додатку. Крім того, функція "Play Mode" дозволяє швидко тестувати додаток у реальному часі без необхідності збірки проекту.

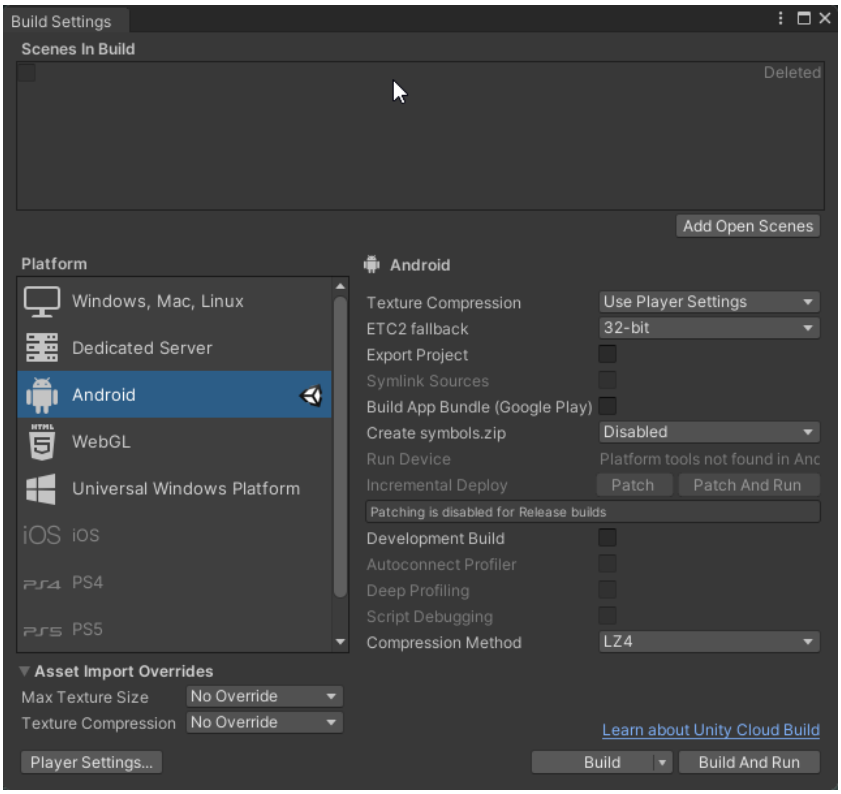

Рисунок 2.6 – Підтримувані платформи

Unity підтримує потужні інструменти для роботи з 2D і 3D-графікою, анімацією та фізикою. Це робить його ідеальним вибором для створення додатків із високим рівнем візуалізації та інтерактивності. Наприклад, розробники можуть створювати анімовані інтерфейси, тривимірні симуляції чи навіть віртуальну реальність (VR) або доповнену реальність (AR) завдяки вбудованій підтримці технологій, таких як ARKit, ARCore та Vuforia.

Unity пропонує унікальний набір інструментів, які забезпечують високу продуктивність, гнучкість і швидкість розробки. Завдяки потужному рендерингу, підтримці широкого спектру платформ і можливості інтеграції з сучасними технологіями, Unity є універсальним вибором як для новачків, так і для досвідчених розробників. Його екосистема дозволяє створювати додатки, які не лише відповідають сучасним стандартам, а й перевершують очікування користувачів.

2.4 Збереження даних

Зберігання даних є невід'ємною частиною будь-якого додатку, оскільки від цього залежить можливість зберігати, обробляти та відновлювати інформацію, яка є важливою для користувача. У контексті кросплатформенних додатків, зокрема розроблених на Unity, правильний вибір способу зберігання даних є критично важливим, оскільки він впливає на продуктивність, масштабованість і загальний користувацький досвід. Розглянемо основні технології та інструменти зберігання даних, які можуть бути використані в Unity, їхні переваги та недоліки.

Одним із найпростіших способів зберігання даних у Unity є використання PlayerPrefs (рис. 2.7). Ця вбудована система дозволяє зберігати прості типи даних, такі як рядки, числа або булеві значення, у локальному сховищі пристрою. PlayerPrefs ідеально підходить для зберігання налаштувань, наприклад, рівня звуку, теми інтерфейсу або останнього відвіданого екрана. Однак цей підхід не є придатним для складних структур даних, оскільки він обмежений простим доступом ключ-значення і не підтримує шифрування за замовчуванням, що може бути проблемою для зберігання чутливої інформації.

Рисунок 2.7 – PlayerPrefs

Для більш складних структур даних, таких як списки транзакцій або детальна інформація про користувача, можна використовувати JSON. Unity має вбудовану підтримку серіалізації об'єктів у формат JSON за допомогою класу JsonUtility (рис. 2.8). Цей підхід дозволяє зберігати дані у вигляді текстових файлів, що легко читаються як додатком, так і сторонніми програмами. JSON підходить для локального зберігання даних, а також для передачі інформації між клієнтом і сервером. Однією з переваг JSON є його легкість і універсальність, однак для великих обсягів даних він може стати менш ефективним порівняно з іншими методами.

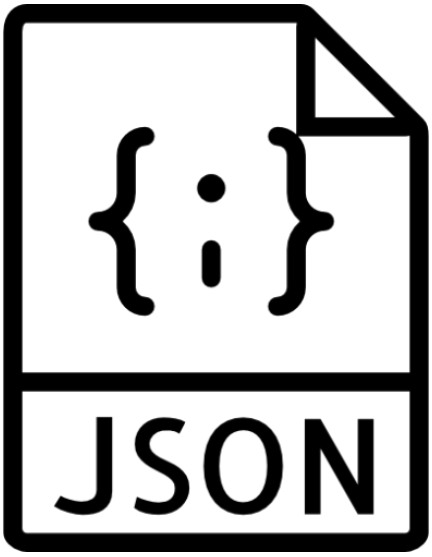

Рисунок 2.8 - Формат JSON

Для додатків, які вимагають зберігання великих обсягів структурованих даних, ідеальним вибором є використання реляційних баз даних, таких як MySQL (рис. 2.9). Це потужний інструмент для управління даними, який дозволяє виконувати складні запити та підтримує великий обсяг інформації. MySQL зазвичай використовується в поєднанні з серверною частиною додатку, яка обробляє запити клієнтів і зберігає дані в централізованому сховищі. Однак пряме використання MySQL у Unity вимагає додаткової інтеграції через сторонні бібліотеки або API.

Рисунок 2.9 – MySQL

Ще одним варіантом для зберігання даних у Unity є використання Google-таблиць як простого та доступного хмарного сховища. За допомогою API Google Sheets можна інтегрувати Unity-додаток із Google-таблицями, що дозволяє зберігати дані в хмарі без необхідності створення власного сервера. Такий підхід підходить для невеликих проєктів або додатків, які потребують спільного доступу до даних між різними користувачами.

Серед сучасних нереляційних баз даних можна виділити Firebase Realtime Database та Firestore, які ідеально підходять для кросплатформенних додатків. Firebase забезпечує синхронізацію даних у реальному часі між клієнтами, підтримує офлайн-режим і має вбудовані засоби аутентифікації користувачів. Цей сервіс є оптимальним вибором для багатокористувацьких додатків, які потребують швидкого доступу до оновлених даних.

Для локального зберігання структурованих даних у Unity популярним вибором є SQLite (рис. 2.10). Ця база даних компактна, не вимагає серверного оточення та легко інтегрується в Unity через сторонні плагіни. SQLite підходить для офлайн-додатків, які потребують роботи з великим обсягом локальних даних, наприклад, мобільних фінансових додатків.

Рисунок 2.10 – SQLite

Вибір технології залежить від потреб проєкту. Наприклад, для локального зберігання невеликих обсягів даних підійдуть PlayerPrefs або JSON. Для складних структур даних, які потребують частого оновлення, краще обрати SQLite або Firebase. Якщо ж додаток має бути масштабованим і працювати в багатокористувацькому середовищі, оптимальним вибором буде MySQL або Firestore.

Загалом Unity пропонує широкі можливості для інтеграції з різними способами зберігання даних, що дозволяє адаптувати додаток до конкретних потреб і забезпечити ефективну роботу в будь-якому середовищі. Завдяки цьому розробники можуть створювати додатки, які не лише зберігають дані, але й надають користувачам швидкий і безпечний доступ до них.

2.5 Структура кросплатформенного додатку

Для досягнення коректної роботи на більшості пристроях розробники створюють архітектуру, яка враховує особливості різних платформ, забезпечуючи високу продуктивність, гнучкість і масштабованість. Структура такого додатку складається з кількох ключових компонентів, кожен з яких виконує специфічну роль у його функціонуванні.

1. Базовий рівень (Core Layer)

Базовий рівень містить основну логіку додатку, яка є універсальною для всіх платформ. Сюди входять:

- Бізнес-логіка: управління даними, виконання обчислень, обробка введених користувачем даних.
- Моделі даних: структура об’єктів і типів даних, які використовуються додатком.
- Сервіси та утиліти: функції, які забезпечують загальну функціональність, наприклад, робота з мережею, обробка файлів, шифрування даних.

Базовий рівень зазвичай реалізується з використанням мови програмування, підтримуваної всіма цільовими платформами, наприклад, C# для Unity, Dart для Flutter або JavaScript для React Native.

2. Рівень інтерфейсу користувача (UI Layer)

Інтерфейс користувача є важливим компонентом будь-якого додатку, оскільки він визначає взаємодію користувача із системою. У кросплатформенних додатках UI Layer може бути:

- Спільним для всіх платформ: наприклад, у Flutter інтерфейс створюється за допомогою єдиної бібліотеки віджетів, яка працює на всіх пристроях.
- Специфічні для платформи: наприклад, у Xamarin.Forms інтерфейс може адаптуватися до особливостей кожної платформи, використовуючи її нативні компоненти.
- Гібридним: деякі елементи UI є спільними, а інші — реалізовані для конкретних платформ.

Крім того, інтерфейс враховує адаптивність, що дозволяє додатку коректно відображатися на екранах із різними роздільними здатностями.

3. Рівень взаємодії з платформою (Platform Integration Layer)

Цей рівень відповідає за доступ до специфічних функцій платформи, таких як камера, GPS, сенсори або push-сповіщення. Він реалізується через:

- Плагіни: наприклад, у Flutter плагіни використовуються для роботи з платформо-специфічними функціями.
- Модулі нативного коду: у React Native можна писати нативні модулі для iOS (Swift/Objective-C) та Android (Java/Kotlin), які забезпечують доступ до унікальних функцій платформи.
- Dependency Injection: у Xamarin Forms використовується DependencyService для реалізації специфічного коду, що викликається із загальної бази.

4. Рівень управління даними (Data Management Layer)

Цей рівень займається обробкою та збереженням даних. Він включає:

- Роботу з API: підключення до зовнішніх сервісів через REST або GraphQL.
- Локальне збереження даних: використання баз даних (SQLite, Realm) або систем збереження ключ-значення (PlayerPrefs, SharedPreferences).
- Кешування: зберігання тимчасових даних для підвищення продуктивності та зменшення навантаження на сервер.

5. Рівень інтеграції з мережею (Networking Layer)

У кросплатформенних додатках часто використовується інтеграція з мережею для передачі даних між клієнтом і сервером. Цей рівень забезпечує:

- Обробку запитів і відповідей: відправлення HTTP-запитів і обробку отриманих даних.
- Управління сесіями: автентифікація, зберігання токенів, підтримка стану користувача.
- Роботу з веб-сокетами: у реальному часі для інтерактивних додатків.

6. Рівень тестування та налагодження (Testing and Debugging Layer)

Тестування є важливою частиною розробки кросплатформенних додатків. Структура передбачає:

- Модульне тестування: перевірка окремих компонентів, таких як бізнес-логіка або сервіси.

- UI-тестування: автоматизоване тестування інтерфейсу для перевірки його коректної роботи на різних платформах.
- Інтеграційне тестування: перевірка взаємодії між різними частинами системи.

7. Рівень підтримки оновлень (Update and Deployment Layer)

Кросплатформенні додатки мають бути легко розгортатися та оновлюватися. Цей рівень включає:

- Систему CI/CD: автоматизація збірки, тестування та доставки додатків за допомогою інструментів, таких як GitHub Actions, Bitrise, Jenkins.
- Системи OTA (Over-the-Air): оновлення додатків без необхідності їх повторного завантаження у магазини (наприклад, CodePush для React Native).

2.6 Висновок до другого розділу

Проєктування кросплатформенного додатку є важливим і складним процесом, який вимагає врахування багатьох факторів, починаючи від вибору технологій і архітектури до забезпечення стабільної роботи на різних платформах. Основною метою такого підходу є створення програмного забезпечення, яке забезпечує однаковий користувацький досвід на всіх цільових пристроях, зберігаючи при цьому ефективність розробки та підтримки.

Вибір правильного інструменту для розробки, такого як Flutter, React Native, Unity або .NET MAUI, відіграє ключову роль у досягненні успіху проєкту. Ці технології надають розробникам можливість використовувати єдину кодову базу, що значно знижує витрати на розробку, пришвидшує впровадження нових функцій і забезпечує легкість підтримки. Крім того, вони дозволяють інтегрувати нативні функції платформ, забезпечуючи високу продуктивність і доступ до унікальних можливостей пристроїв.

Архітектура кросплатформенного додатку має бути модульною, багаторівневою та адаптованою до різних сценаріїв використання. Це включає чітке розмежування бізнес-логіки, інтерфейсу користувача, управління даними. Такий підхід забезпечує гнучкість у розробці, зручність тестування та можливість масштабування додатку.

Одним із ключових аспектів є тестування, яке повинно включати модульні, інтеграційні та UI-тести, щоб забезпечити коректну роботу додатку на всіх платформах. Інструменти CI/CD автоматизують процеси збірки, тестування та розгортання, що дозволяє швидше випускати оновлення та зменшувати кількість помилок.

Кросплатформенні додатки є потужним рішенням у сучасному світі, оскільки вони забезпечують доступ до широкої аудиторії, скорочують витрати на розробку та підтримку, а також дозволяють швидко адаптуватися до змін на ринку. Успіх проєкту залежить від ретельного планування, правильного вибору інструментів і дотримання найкращих практик розробки. Завдяки цьому підходу розробники можуть створювати додатки, які відповідають високим стандартам якості та задовольняють потреби користувачів у всьому світі.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ

3.1 Актуальність розробки

У сучасному світі управління фінансами стає дедалі важливішим як для окремих осіб, так і для організацій. Зростання кількості фінансових операцій, складність управління доходами та витратами, а також необхідність ухвалення швидких і обґрунтованих рішень потребують ефективних інструментів і систем. Саме тому розробка системи управління фінансами є надзвичайно актуальною.

Постійне зростання обсягу фінансових даних ускладнює їх обробку та аналіз. Традиційні методи управління фінансами вже не відповідають сучасним викликам, оскільки вони є повільними, громіздкими та часто не забезпечують необхідного рівня точності. У таких умовах використання автоматизованих систем управління фінансами дозволяє оптимізувати цей процес, підвищуючи ефективність прийняття рішень та знижуючи ризики.

Особливістю цієї розробки є використання платформи Unity, що дозволяє створити інтуїтивно зрозумілий інтерфейс та забезпечити високий рівень інтерактивності системи. Завдяки Unity, система управління фінансами може бути розроблена з урахуванням сучасних вимог до графіки та продуктивності, що робить її більш привабливою для користувачів. Unity також забезпечує підтримку кросплатформеності, що є важливим для доступності системи на різних пристроях.

Додатково, актуальність розробки системи управління фінансами підкреслюється розвитком цифрових технологій і поширенням мобільних додатків. Користувачі все частіше віддають перевагу рішенням, які дозволяють керувати фінансами в режимі реального часу, отримувати аналітичну інформацію, прогнозувати витрати та доходи. Таким чином, система управління фінансами повинна бути не лише функціональною, а й доступною, зручною та безпечною для користувача.

Економічна нестабільність та глобалізація фінансових ринків також сприяють актуалізації цієї теми. Для багатьох людей важливо не лише ефективно контролювати свої фінансові ресурси, але й зберігати їх у безпеці, мінімізуючи ризики та втрати. Організації, у свою чергу, потребують інструментів, які забезпечать прозорість фінансових процесів і допоможуть уникнути фінансових помилок.

Таким чином, розробка системи управління фінансами відповідає сучасним тенденціям і потребам суспільства. Це є важливим кроком на шляху до оптимізації фінансових процесів, забезпечення фінансової стабільності та підвищення рівня фінансової грамотності серед користувачів.

3.2 Структура проекту

Структура програми складається з 4 класів, які реалізують основний функціонал додатку:

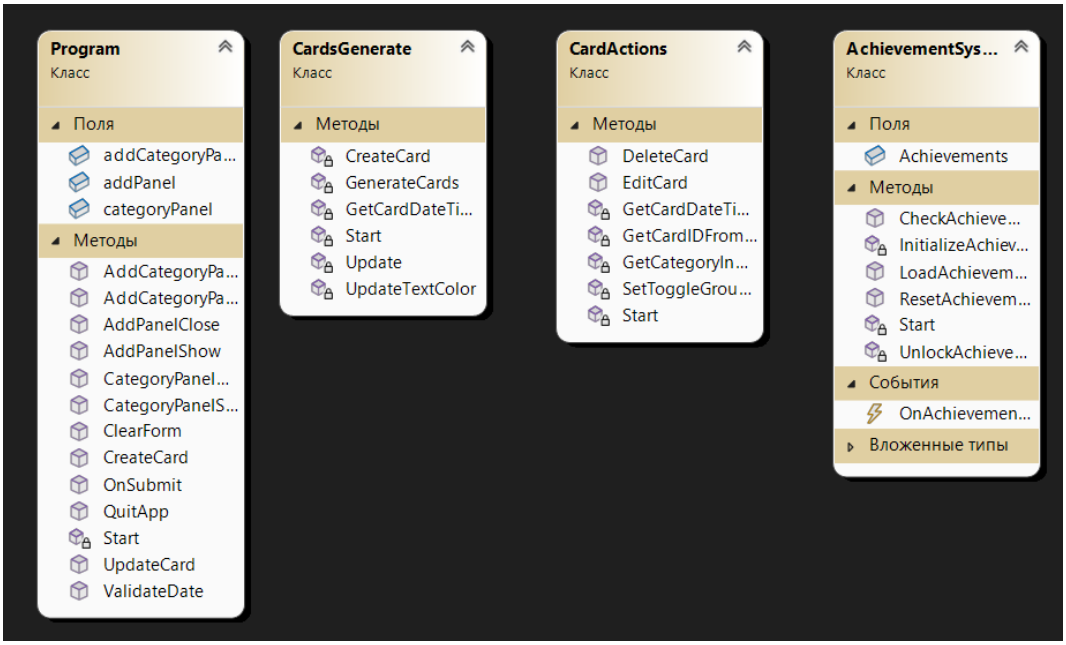

Рисунок 3.1 – Діаграма класів

Кожен клас відповідає за свій функціонал, від створення транзакцій, до зберігання користувацьких налаштувань та даних та встановлення категорій транзакції.

3.3 Розробка додатку

Розробка додатку здійснюється на платформі Unity, що забезпечує високу продуктивність, кросплатформеність та зручність у використанні. Unity надає інструменти для створення інтуїтивного інтерфейсу користувача та впровадження функціоналу, який дозволяє користувачам ефективно управляти своїми фінансами. Використання Unity також відкриває можливості для інтеграції з іншими сервісами, такими як аналітичні платформи та мобільні додатки, що розширює функціональність системи.

Перш за все, розглянемо клас ShowPanels, який відповідає за управління панелями інтерфейсу користувача. Цей клас включає в себе кілька важливих компонентів. Поля класу містять посилання на аніматори, які забезпечують відображення та анімацію різних панелей. Це дозволяє змінювати стан кожної панелі (відкрита чи закрита) у залежності від дій користувача.

Основний метод, що викликається при запуску, встановлює початкові значення для всіх панелей, закриваючи їх. Це гарантує, що при старті додатку жодна панель не буде відкрита, створюючи чітку та організовану початкову структуру інтерфейсу.

```
public class ShowPanels : MonoBehaviour
{
    public Animator addPanel;
    public Animator categoryPanel;
    public Animator addCategoryPanel;

    void Start()
    {
        addPanel.SetBool("isOpen", false);
        categoryPanel.SetBool("isOpen", false);
        addCategoryPanel.SetBool("isOpen", false);
```

Рисунок 3.2 – Клас ShowPanels

Додатково, клас включає методи (рис. 3.3) для відкриття та закриття кожної панелі окремо. Відкриття панелей супроводжується можливістю очищення форми або виконання інших дій, необхідних для підготовки інтерфейсу до роботи користувача. Закриття панелей здійснюється шляхом зміни відповідних параметрів аніматора, що забезпечує плавний перехід.

Рисунок 3.3 – Методи для відкриття та закриття панелей

Клас також має функцію для завершення роботи додатку. Цей метод дозволяє користувачам швидко закрити програму, що особливо корисно на мобільних пристроях або ПК (рис. 3.4).

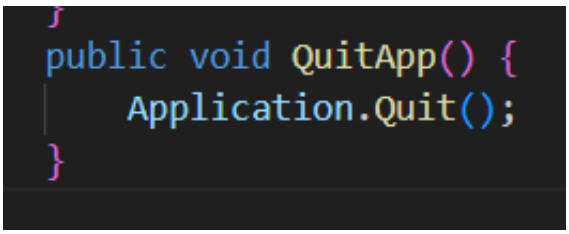

Рисунок 3.4 – Функція для завершення роботи

Загалом, цей клас є ключовим елементом, що забезпечує взаємодію користувача з інтерфейсом. Його структура та функціонал спрямовані на

створення зручного та інтуїтивного користувацького досвіду, що є важливим аспектом успішної реалізації системи.

Основний компонент класу — аніматори, які відповідають за динамічне відкриття та закриття панелей. Ці аніматори є ключовим інструментом для управління візуальними ефектами, такими як плавні переходи між станами, які покращують користувацький досвід. Вони працюють шляхом зміни параметрів стану, таких як "isOpen", які керують анімаційними переходами між відкритими та закритими станами кожної панелі. Такий підхід не лише підвищує естетичність додатку, але й забезпечує функціональність, яка адаптується до дій користувача. Використання аніматорів також спрощує розробку, дозволяючи легко налаштовувати та змінювати поведінку інтерфейсу.

Створення анімації в Unity починається з підготовки необхідних елементів сцени. Спочатку розробник додає об’єкти, які потребують анімації, такі як панелі інтерфейсу або кнопки. Потім ці об’єкти пов’язуються з аніматором — спеціальним компонентом Unity, що дозволяє налаштовувати анімаційні переходи та створювати їхню логіку.

Наступним кроком є створення анімаційних кліпів (рис. 3.5). Для цього використовується вбудований інструмент Animation в Unity, що дозволяє вказувати початкове і кінцеве положення об’єкта, його масштаб або прозорість. Анімаційні кліпи можуть бути деталізованими, наприклад, включати кілька етапів зміни властивостей об’єкта, що додає реалістичності.

Рисунок 3.5 – Створення кліпів

Коли кліпи створені, вони об’єднуються в анімаційний контролер. Контролер визначає порядок виконання анімацій, залежно від взаємодії користувача або внутрішньої логіки програми. Наприклад, при натисканні на кнопку анімаційний контролер може активувати кліп відкриття панелі.

Додатково, налаштовуються параметри (рис. 3.6), які керують станами анімації. Ці параметри, як правило, є змінними типу Boolean, Float або Integer, що змінюють свій стан під час виконання програми. За їхньою допомогою аніматор може плавно перемикатися між різними станами об’єкта.

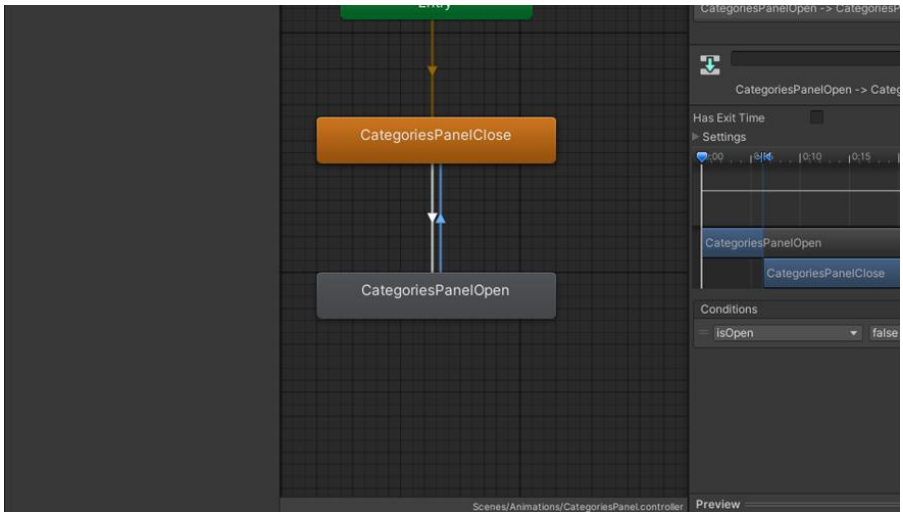

Рисунок 3.6 – Налаштування параметрів

Останнім етапом є тестування анімацій та їх оптимізація. Це включає перевірку плавності переходів, коректності логіки та узгодженості з іншими елементами інтерфейсу. У разі необхідності анімації коригуються, щоб забезпечити найкращий користувацький досвід.

Наступним потрібно розглянути клас OnValidationObject, що має в собі функціонал для обробки даних отриманих з форми за допомогою методу OnSubmit.

```
public void OnSubmit()
{
    int day = int.Parse(dayInputField.text);
    int month = int.Parse(monthInputField.text);
    int year = int.Parse(yearInputField.text);
    int hour = int.Parse(hourInputField.text);
    int minute = int.Parse(minuteInputField.text);
    DateTime dateTime = new DateTime(year, month, day
    float balanceChange = float.Parse(balanceChangeIn
    bool isPositive = GetToggleValue();
    string category = categoryDropdown.options[catego
}
```

Рисунок 3.7 – Метод OnSubmit

Процес валідації даних у методі OnSubmit починається з отримання вхідних даних із полів форми. Для цього використовуються значення текстових полів, таких як dayInputField, monthInputField, yearInputField, hourInputField та minuteInputField. Отримані значення конвертуються у відповідні числові типи за допомогою методів int.Parse та float.Parse.

Наступним етапом є створення об’єкта типу DateTime, який агрегує введені дані про дату та час. Це дозволяє забезпечити коректність введених даних, а також перевірити їхню відповідність реальним календарним значенням. Якщо введені дані некоректні, наприклад, містять значення місяця більше 12 або дня більше 31, система може повідомити про помилку користувача.

Окрім дати, перевіряються фінансові дані, такі як зміна балансу (balanceChange). Ця інформація конвертується у числовий формат і залежно від вибору користувача (позитивна чи негативна зміна) враховується у відповідних розрахунках. Для цього використовується допоміжний метод GetToggleValue, який визначає стан перемикача.

Категорія витрат або доходів обирається з випадаючого списку (categoryDropdown). Значення списку забезпечує точність класифікації введених даних, що є важливим для подальшого аналізу фінансових операцій.

Валідація завершується перевіркою всіх отриманих даних на відповідність логічним і числовим умовам. У разі виявлення помилок користувач отримує повідомлення про необхідність виправлення введених даних.

У коді для реалізації форми використовуються текстові поля (InputField), випадаючі списки (Dropdown), перемикачі (Toggle) та кнопки (Button).

Текстові поля забезпечують можливість введення даних, таких як дата, час та сума змін балансу. Ключовою особливістю є їхня здатність конвертувати введені значення у потрібні числові формати для подальшої обробки. Випадаючі списки застосовуються для вибору категорій доходів або витрат. Вони забезпечують єдину класифікацію, що полегшує аналіз фінансових даних і мінімізує можливість помилок при введенні. Перемикачі використовуються для визначення типу зміни балансу — позитивного або негативного. Це дозволяє інтуїтивно задавати напрямок фінансової операції, що спрощує роботу з формою.

Кнопки відповідають за взаємодію з формою, наприклад, підтвердження введених даних або скасування змін. Їхнє налаштування дозволяє підключати методи, які виконують валідацію або завершують певні дії у додатку.

Наступний метод ValidateDate (рис. 3.8) перевіряє, чи є значення дня, місяця та року коректними. Він приймає три параметри — день, місяць і рік, і повертає булеве значення, яке вказує на те, чи є дата валідною. Це забезпечує

захист від некоректного введення, наприклад, якщо місяць більше 12 або день перевищує кількість днів у місяці.

```
public bool ValidateDate(int day, int month, int year)
{
    try
    {
        DateTime date = new DateTime(year, month, day);
        return true;
    }
    catch
    {
        return false;
    }
}
```

Рисунок 3.8 – Метод ValidateDate

Метод намагається створити об’єкт DateTime на основі вхідних параметрів. Якщо дата є некоректною, викидається виключення, і метод повертає значення false. В іншому випадку повертається true, що означає коректність введених даних.

Цей метод інтегрується у загальну систему валідації форми. Перед тим як обробляти введені користувачем дані, система перевіряє, чи є дата дійсною. Це гарантує, що подальші операції, які залежать від дати, будуть виконані коректно та без помилок.

Метод GetToggleValue відповідає за визначення стану активного перемикача у групі перемикачів. Він перебирає всі активні перемикачі у ToggleGroup та повертає булеве значення залежно від їхнього стану.

```
foreach (Toggle toggle in toggleGroup.ActiveToggles())
{
    return toggle.name == "Plus";
}
return true;
```

Рисунок 3.9 – Метод GetToggleValue

Цей метод перевіряє, чи активний перемикач має назву "Plus". Якщо так, метод повертає true, що вказує на позитивний вибір користувача. Якщо жоден перемикач не активний або назва не відповідає "Plus", метод повертає false.

Метод CreateCard (рис. 3.10) використовується для створення нової картки із заданими параметрами. При створенні картки виконується збереження даних у системі з використанням PlayerPrefs. Збереження даних охоплює такі параметри, як сума зміни балансу, поточний баланс, категорія витрат або доходів, а також дата створення. Крім того, метод встановлює стан панелі додавання картки у закритий стан, а також викликає метод ClearForm для очищення полів введення форми. Використання PlayerPrefs дозволяє зберігати дані локально, забезпечуючи доступ до них після повторного запуску програми.

```
public void CreateCard(int ID, float change, float currentBalance, int cat
{
    PlayerPrefs.SetFloat("Card_" + ID + "_Change", change);
    PlayerPrefs.SetFloat("Card_" + ID + "_CurrentBalance", currentBalance
    PlayerPrefs.SetFloat("CurrentBalance", currentBalance + change);
    PlayerPrefs.SetInt("Card_" + ID + "_Category", categoryID);
    PlayerPrefs.SetString("Card_" + ID + "_Date", dateTimeString);
    addPanel.SetBool("isOpen", false);
    ClearForm();
}
```

Рисунок 3.10 – Метод CreateCard

Метод UpdateCard (рис. 3.11) використовується для оновлення даних існуючої картки. Цей метод дозволяє змінювати такі параметри, як сума зміни балансу, категорія, а також дата транзакції. Крім того, UpdateCard виконує перевірку наявності картки за заданим ідентифікатором, щоб уникнути помилок при оновленні. У разі відсутності картки з відповідним ідентифікатором, метод може повідомити користувача про помилку.

```
public void UpdateCard(int ID, float newChange, int newCategoryID, string newDateTimeString)
{
    if (PlayerPrefs.HasKey("Card_" + ID + "_Change"))
    {
        float currentBalance = PlayerPrefs.GetFloat("Card_" + ID + "_CurrentBalance")
        - PlayerPrefs.GetFloat("Card_" + ID + "_Change");
        PlayerPrefs.SetFloat("Card_" + ID + "_Change", newChange);
        PlayerPrefs.SetFloat("Card_" + ID + "_CurrentBalance", currentBalance + newChange);
        PlayerPrefs.SetFloat("CurrentBalance", PlayerPrefs.GetFloat("CurrentBalance")
        - PlayerPrefs.GetFloat("Card_" + ID + "_Change") + newChange);
        PlayerPrefs.SetInt("Card_" + ID + "_Category", newCategoryID);
        PlayerPrefs.SetString("Card_" + ID + "_Date", newDateTimeString);
    }
    else
    {
        Debug.LogError("Card with ID " + ID + " does not exist.");
    }
}
```

Рисунок 3.11 – Метод UpdateCard

Наступний метод GetCategoryID (рис. 3.12) дозволяє знайти ідентифікатор категорії за її назвою. Він перебирає всі існуючі категорії, збережені у PlayerPrefs, і перевіряє, чи відповідає назва категорії заданому значенню. У разі знаходження відповідності метод повертає ідентифікатор категорії. Якщо жодна категорія не відповідає назві, метод повертає значення -1, яке свідчить про відсутність відповідного запису.

```
for (int categoryID = 0; categoryID <= PlayerPrefs.GetInt("LastCategoryID"); categoryID++
{
    if (PlayerPrefs.GetString("Category_" + categoryID + "_Name") == categoryName)
    {
        return categoryID;
    }
}
return -1;
```

Рисунок 3.12 – Метод GetCategoryID

Цей метод є зручним для роботи із системою категорій у додатку. Він використовує цикл для перебору всіх можливих категорій на основі збереженого значення LastCategoryID. Такий підхід дозволяє динамічно додавати та видаляти категорії, оскільки ідентифікатори залишаються послідовними.

Метод ClearForm (рис. 3.13) виконує очищення форми введення даних, що використовується у додатку для створення чи оновлення записів.

Основною метою цього методу є скидання значень усіх текстових полів форми до початкового стану. Поля, що очищаються, включають день, місяць, рік, годину, хвилину, а також поле для введення зміни балансу. Це забезпечує користувачеві чисту форму для введення нових даних, запобігаючи перенесенню значень від попередніх операцій.

```
public void ClearForm()
{
    dayInputField.text = "";
    monthInputField.text = "";
    yearInputField.text = "";
    hourInputField.text = "";
    minuteInputField.text = "";
    balanceChangeInputField.text = "";
    PlayerPrefs.SetInt("CurrentChanging", -1);
}
```

Рисунок 3.13 –Метод ClearForm

Крім скидання текстових полів, метод встановлює значення за замовчуванням для змінної, яка відповідає за поточний вибір категорії. Це досягається через використання PlayerPrefs, де змінна "CurrentChanging" встановлюється у значення -1. Таким чином, метод ClearForm забезпечує готовність форми до нового циклу введення даних, зменшуючи ймовірність помилок, що виникають через залишкові значення.

Клас CardsGenerate відповідає за генерацію карток, які відображають фінансові операції користувача. Він реалізує функціонал, що забезпечує створення нових карток на основі збережених даних, їх оновлення при зміні інформації, а також управління візуальними характеристиками карток. Цей клас працює у тісній інтеграції з PlayerPrefs, що дозволяє зберігати і використовувати дані навіть після завершення роботи додатку.

У методі Start (рис. 3.14) виконується ініціалізація процесу генерації карток шляхом виклику методу GenerateCards. Також встановлюється

початкова позиція батьківського об’єкта карток (parent) з метою зручного розташування елементів у сцені. Цей підхід дозволяє задати базові параметри для коректного відображення карток на інтерфейсі.

```
void Start()
{
    GenerateCards();
    parent.localPosition = new Vector3(parent.localPositio
}
```

Рисунок 3.14 – Метод Start

Метод Update (рис. 3.15) реалізує механізм оновлення відображення карток у випадку, якщо у середовищі (env) встановлений прапорець NeedRefreshCards. Якщо необхідність оновлення виявлена, повторно викликається метод GenerateCards, що забезпечує динамічність у роботі з даними.

```
void Update()
{
    if(env.NeedRefreshCards) {
        GenerateCards();
    }
}
```

Рисунок 3.15 – Метод Update

Основним функціоналом класу є метод GenerateCards (рис. 3.16), який виконує очищення старих карток, перевірку даних у PlayerPrefs і створення нових карток на основі збереженої інформації. Усі дочірні об’єкти батьківського об’єкта видаляються, за винятком специфічних елементів, таких як "CardBalance" або "Category". Далі здійснюється ітерація по ідентифікаторах карток до значення LastCardID. Якщо для певної картки збережені дані, виконується додаткова перевірка її відповідності поточній

категорії. У разі відповідності викликається метод CreateCard, який створює об’єкт картки у сцені.

```
private void GenerateCards()
{
    int lastCardID = PlayerPrefs.GetInt("LastCardID");
    int currentCategory = PlayerPrefs.GetInt("CurrentCategory");
    foreach (Transform child in parent)
    {
        if (child.name != "CardBalance" && child.name != "Category")
        {
            Destroy(child.gameObject);
        }
    }
    for (int ID = 0; ID <= lastCardID; ID++)
    {
    if (PlayerPrefs.HasKey("Card_" + ID + "_Change"))
        {
            int cardCategoryID = PlayerPrefs.GetInt("Card_" + ID + "_Category");
            if (currentCategory == 0 || cardCategoryID == currentCategory)
            {
                CreateCard(ID);
            }
        }
    }
    env.NeedRefreshCards = false;
    parent.localPosition = new Vector3(parent.localPosition.x, -10000, parent.loc
}
```

Рисунок 3.16 – Метод GenerateCards

Метод CreateCard (рис. 3.17) відповідає за інстанціювання нового об’єкта картки на основі префабу cardPrefab. Він завантажує необхідні дані з PlayerPrefs, такі як сума змін, поточний баланс, категорія, дата операції та колір категорії. Ці дані відображаються у відповідних текстових і візуальних елементах картки за допомогою компонентів, таких як TextMeshProUGUI і Image. Метод також дбає про правильне форматування тексту, наприклад, додаючи "+" перед позитивними значеннями. Важливим аспектом є динамічна зміна кольору тексту залежно від балансу, що реалізовано у допоміжному методі UpdateTextColor.

```
private void CreateCard(int ID)
{
    GameObject newCard = Instantiate(cardPrefab, parent);
    newCard.name = "Card_" + ID;
    float change = PlayerPrefs.GetFloat("Card_" + ID + "_Change");
    float currentBalance = PlayerPrefs.GetFloat("Card_" + ID + "_CurrentBalance");
    int categoryID = PlayerPrefs.GetInt("Card_" + ID + "_Category");
    string date = PlayerPrefs.GetString("Card_" + ID + "_Date");
    string hexColor = PlayerPrefs.GetString("Category_" + categoryID + "_HEX");
    TextMeshProUGUI balanceChangeText = newCard.transform.Find("BalanceChange").GetComponent<T
    TextMeshProUGUI balanceCurrentText = newCard.transform.Find("BalanceCurrent").GetComponent
    TextMeshProUGUI categoryNameText = newCard.transform.Find("CategoryName").GetComponent<Tex
    TextMeshProUGUI dateText = newCard.transform.Find("Date").GetComponent<TextMeshProUGUI>();
    Image cardImage = newCard.transform.Find("Color").GetComponent<Image>();
    balanceChangeText.text = (change > 0 ? "+" : "") + change.ToString("F2") + " ₴";
    balanceCurrentText.text = "Current balance: " + currentBalance.ToString("F2") + " ₴";
    categoryNameText.text = "Category: " + GetCategoryName(categoryID);
    dateText.text = "Date: " + GetDate(date);
    cardImage.color = HexToColor(hexColor);
    UpdateTextColor(balanceChangeText, change);
    UpdateTextColor(balanceCurrentText, currentBalance);
}
```

Рисунок 3.17 – Метод CreateCard

Метод GetCardDateTime (рис. 3.18) використовується для перетворення рядка з датою у формат DateTime. Він перевіряє коректність формату введених даних і, у разі помилки, викидає виключення, що гарантує якість даних, які використовуються в додатку.

```
private DateTime GetCardDateTime(string dateTimeString)
{
    DateTime dateTime;
    if (DateTime.TryParse(dateTimeString, out dateTime))
    {
        return dateTime;
    }
    else
    {
        throw new Exception("Помилковий формат дати та часу");
    }
}
```

Рисунок 3.18 – Метод GetCardDateTime

Допоміжний метод UpdateTextColor визначає колір тексту залежно від значення балансу. Якщо баланс є позитивним, текст забарвлюється у колір, що відповідає позитивним значенням. Негативні баланси та нульові значення отримують відповідно негативний і нейтральний кольори. Це покращує сприйняття користувачем інформації про фінансові операції.

```
private void UpdateTextColor(TextMeshProUGUI balanceObj, float balance)
{
    if (balance > 0)
    {
        balanceObj.color = positiveBalanceColor;
    }
    else if (balance < 0)
    {
        balanceObj.color = negativeBalanceColor;
    }
    else
    {
        balanceObj.color = zeroBalanceColor;
    }
}
```

Рисунок 3.19 – Метод UpdateTextColor

Клас CardsGenerate є критично важливим для реалізації функціоналу управління фінансовими операціями в додатку. Він забезпечує динамічне створення, оновлення та візуалізацію карток, роблячи інтерфейс більш інформативним і зручним для користувача. Завдяки чіткому розподілу відповідальностей між методами та інтеграції з системою збереження даних PlayerPrefs, клас гарантує стабільність та гнучкість у роботі з фінансовою інформацією.

Клас реалізує функціонал редагування карток у фінансовій системі. Він забезпечує зручний інтерфейс для отримання, відображення та оновлення даних карток, інтегруючись із системою збереження даних (PlayerPrefs) та графічним інтерфейсом Unity. Ось детальний опис кожного методу та компонентів цього класу:

Перший метод класу – EditCard (рис. 3.20). Основна мета цього методу — завантаження даних картки у форму для редагування. Спочатку викликається метод GetCardIDFromName, який витягує ідентифікатор картки з її імені. Якщо ID не вдається отримати, система генерує попередження та припиняє виконання методу.

```
public void EditCard()
{
    int cardID = GetCardIDFromName();
    if (cardID == -1)
    {
        Debug.LogWarning($"Не вдалось отримати cardID з імені: {gameObject.name}");
        return;
    }
    if (PlayerPrefs.HasKey("Card_" + cardID + "_Change"))
    {
        PlayerPrefs.SetInt("CurrentChanging", cardID);
        float change = PlayerPrefs.GetFloat("Card_" + cardID + "_Change");
        float currentBalance = PlayerPrefs.GetFloat("Card_" + cardID + "_CurrentBalance");
        int categoryID = PlayerPrefs.GetInt("Card_" + cardID + "_Category");
        string dateTime = PlayerPrefs.GetString("Card_" + cardID + "_Date");
        cashValueInput.text = change.ToString();
        DateTime cardDateTime = GetCardDateTime(dateTime);
        dayInput.text = cardDateTime.ToString("dd");
        monthInput.text = cardDateTime.ToString("MM");
        yearInput.text = cardDateTime.ToString("yyyy");
        hourInput.text = cardDateTime.ToString("HH");
        minuteInput.text = cardDateTime.ToString("mm");
        SetToggleGroupValue(change > 0 ? "Plus" : "Minus");
        categoryDropdown.value = GetCategoryIndex(categoryID);
        addCardPanel.SetBool("isOpen", true);
    }
    else
    {
        Debug.LogWarning($"Карточка с ID {cardID} не знайдена.");
    }
}
```

Рисунок 3.20 – Метод EditCard

Якщо ID отримано та існують дані картки в PlayerPrefs, вони завантажуються у відповідні поля вводу форми. Зокрема, завантажується сума зміни (change), поточний баланс (currentBalance), категорія (categoryID), а також дата та час операції (dateTime). Дата й час конвертуються у формат DateTime за допомогою методу GetCardDateTime. Поля вводу для дня, місяця, року, години та хвилини заповнюються відповідними значеннями з дати. Також встановлюється стан перемикача (позитивна чи негативна зміна) через метод SetToggleGroupValue, а значення випадаючого списку категорій — через метод GetCategoryIndex.

Після заповнення форми панель редагування (addCardPanel) відкривається для користувача, що дозволяє зручно змінити необхідні дані. У разі, якщо картку знайти не вдалося, генерується відповідне попередження.

Метод GetCardIDFromName (рис. 3.21) отримує ID картки з її імені, яке має формат Card_{ID}. Якщо ім’я починається зі слова "Card_", метод виділяє числову частину, конвертує її у ціле число та повертає. У разі, якщо ім’я не

відповідає очікуваному формату або конвертація не вдалася, метод повертає значення -1. Цей метод гарантує, що ID картки буде отримано коректно, якщо назва картки відповідає стандарту.

```
private int GetCardIDFromName()
{
    string name = gameObject.name;
    if (name.StartsWith("Card_"))
    {
        string idString = name.Substring(5);
        int cardID;
        if (int.TryParse(idString, out cardID))
        {
            return cardID;
        }
    }
    return -1;
}
```

Рисунок 3.21 – Метод GetCardIDFromName

Метод GetCardDateTime (рис. 3.22) приймає рядок з датою та часом і конвертує його у формат DateTime. Якщо конвертація успішна, метод повертає отримане значення. У разі некоректного формату рядка генерується виключення з повідомленням про помилковий формат. Цей метод забезпечує валідацію та правильну обробку даних дати й часу.

```
private DateTime GetCardDateTime(string dateTimeString)
{
    DateTime dateTime;
    if (DateTime.TryParse(dateTimeString, out dateTime))
    {
        return dateTime;
    }
    else
    {
        throw new Exception("Помилковий формат дати та час
    }
}
```

Рисунок 3.22 – Метод GetCardDateTime

Метод SetToggleGroupValue відповідає за встановлення стану перемикачів у групі (toggleGroup). Він перебирає всі перемикачі, які є дочірніми елементами групи, і активує той, чия назва відповідає переданому значенню toggleName. Якщо перемикач знайдено, його стан змінюється на true, після чого цикл завершується. Метод забезпечує інтуїтивну взаємодію з інтерфейсом, дозволяючи користувачеві легко розрізняти позитивні та негативні фінансові операції.

```
private void SetToggleGroupValue(string toggleName)
{
    foreach (Toggle toggle in toggleGroup.GetComponentsI
    {
        if (toggle.name == toggleName)
        {
            toggle.isOn = true;
            break;
        }
    }
}
```

Рисунок 3.23 – Метод SetToggleGroupValue

Метод GetCategoryIndex (рис. 3.24) визначає індекс категорії у випадаючому списку (categoryDropdown) на основі переданого categoryID. Він перебирає всі опції списку, порівнюючи їх текстові значення зі збереженими у PlayerPrefs. Якщо знайдено відповідність, метод повертає індекс опції. Якщо категорія не знайдена, повертається значення 0 (індекс за замовчуванням). Цей підхід гарантує коректну синхронізацію даних категорій із графічним інтерфейсом.

```
private int GetCategoryIndex(int categoryID)
{
    for (int i = 0; i < categoryDropdown.options.Count; i++)
    {
        if (categoryDropdown.options[i].text == PlayerPrefs.Get
        {
            return i;
        }
    }
    return 0;
}
```

Рисунок 3.24 – Метод GetCategoryIndex

Компоненти робочого середовища:

PlayerPrefs: використовується для збереження та завантаження даних карток, таких як сума змін, поточний баланс, категорія, дата та час. Ця система забезпечує локальне зберігання даних, доступне навіть після перезапуску програми.

Графічні елементи форми:

- cashValueInput, dayInput, monthInput, yearInput, hourInput, minuteInput: текстові поля вводу, у які завантажуються дані картки для редагування.
- toggleGroup: група перемикачів, яка дозволяє обрати тип операції (позитивна чи негативна).
- categoryDropdown: випадаючий список, який дозволяє вибрати категорію картки.
- addCardPanel: панель, яка відкривається для редагування картки.

Останній клас AchievementSystem відповідає за створення, зберігання, перевірку та управління досягненнями у грі або додатку. Основна мета цього класу — забезпечити інтерактивність і мотивацію користувачів шляхом надання досягнень за виконання певних дій або досягнення цілей.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class AchievementSystem : MonoBehaviour
{
    // Структура для опису досягнення
    [Serializable]
    public class Achievement
    {
        public string ID;
        public string Name;
        public string Description;
        public bool IsUnlocked;
        public Action OnUnlock;
    }
```

Рисунок 3.24 – Клас AchievementSystem

Клас використовує вкладену структуру Achievement для опису окремого досягнення. Кожне досягнення має унікальний ідентифікатор (ID), назву (Name), опис (Description), булеве значення (IsUnlocked), яке вказує, чи було досягнення розблоковано, та подію (OnUnlock), яка виконується під час розблокування досягнення. Список усіх досягнень зберігається у властивості Achievements, що є списком об’єктів Achievement.

Містить також подію OnAchievementUnlocked, яка спрацьовує при розблокуванні нового досягнення. Ця подія дозволяє іншим компонентам, наприклад інтерфейсу користувача, реагувати на факт отримання досягнення.

У методі Start (рис. 3.25) викликається метод InitializeAchievements (рис. 3.25), який додає до списку Achievements декілька початкових досягнень.

```
void Start()
{
    // Ініціалізація досягнень
    InitializeAchievements();
}
private void InitializeAchievements()
{
    Achievements.Add(new Achievement
    {
        ID = "achieve_first_transaction",
        Name = "Перша транзакція",
        Description = "Виконайте першу фінансову транзакцію.",
        IsUnlocked = false,
        OnUnlock = () => Debug.Log("Досягнення розблоковано: Перша транзакція")
    });
    Achievements.Add(new Achievement
    {
        ID = "achieve_balance_1000",
        Name = "Збалансований бюджет",
        Description = "Досягніть балансу у 1000 ₴.",
        IsUnlocked = false,
        OnUnlock = () => Debug.Log("Досягнення розблоковано: Збалансований бюджет")
    });
    Achievements.Add(new Achievement
    {
        ID = "achieve_10_transactions",
        Name = "Серійний користувач",
        Description = "Виконайте 10 транзакцій.",
        IsUnlocked = false,
        OnUnlock = () => Debug.Log("Досягнення розблоковано: Серійний користувач")
    });
}
```

Рисунок 3.25 – Методи Start та InitializeAchievements

Кожне досягнення має прив’язану дію, яка виконується під час його розблокування, наприклад, виведення повідомлення у консоль.

Метод CheckAchievements (рис. 3.26) відповідає за перевірку, чи виконані умови для розблокування певного досягнення. Метод приймає

ідентифікатор досягнення (achievementID), а також опціональні параметри, такі як баланс користувача (balance) і кількість транзакцій (transactionCount).

```
public void CheckAchievements(string achievementID, float balance = 0, int transactionCount = 0)
{
    foreach (var achievement in Achievements)
    {
        if (achievement.ID == achievementID && !achievement.IsUnlocked)
        {
            switch (achievement.ID)
            {
                case "achieve_first_transaction":
                    UnlockAchievement(achievement);
                    break;
                case "achieve_balance_1000":
                    if (balance >= 1000)
                    {
                        UnlockAchievement(achievement);
                    }
                    break;
                case "achieve_10_transactions":
                    if (transactionCount >= 10)
                    {
                        UnlockAchievement(achievement);
                    }
                    break;
            }
        }
    }
}
```

Рисунок 3.26 – Метод CheckAchievements

У залежності від типу досягнення виконуються умови:

- Якщо це перша транзакція, досягнення розблоковується одразу.
- Для досягнення балансу у 1000 ₴ перевіряється, чи поточний баланс користувача досяг цього значення.
- Для досягнення 10 транзакцій перевіряється, чи їх кількість перевищує 10.

Метод UnlockAchievement (рис. 3.27) оновлює стан досягнення, встановлюючи його як розблоковане (IsUnlocked = true). Інформація про це зберігається у PlayerPrefs, що забезпечує збереження стану навіть після перезапуску програми. Метод також викликає подію OnUnlock, прив’язану до досягнення, а також глобальну подію OnAchievementUnlocked, яка інформує систему про нове розблоковане досягнення.

```
// Метод для розблокування досягнення
private void UnlockAchievement(Achievement achievement)
{
    achievement.IsUnlocked = true;
    PlayerPrefs.SetInt(achievement.ID, 1); // Збереження досягнення
    achievement.OnUnlock?.Invoke();
    OnAchievementUnlocked?.Invoke(achievement); // Сповіщення про нове досягнення
}
```

Рисунок 3.27 – Метод UnlockAchievement

Метод LoadAchievements (рис. 3.28) завантажує стан усіх досягнень із PlayerPrefs, щоб забезпечити синхронізацію даних між сесіями додатку. Скидання досягнень (ResetAchievements) видаляє всі записи у PlayerPrefs і повертає досягнення до початкового стану, де всі вони є заблокованими. Цей метод особливо корисний для тестування (рис. 3.28).

```
public void LoadAchievements()
{
    foreach (var achievement in Achievements)
    {
        achievement.IsUnlocked = PlayerPrefs.GetInt(achievement.ID, 0)
    }
}
// Очистка досягнень (для тестування)
public void ResetAchievements()
{
    foreach (var achievement in Achievements)
    {
        PlayerPrefs.DeleteKey(achievement.ID);
        achievement.IsUnlocked = false;
    }
}
```

Рисунок 3.28 – Методи LoadAchievements, ResetAchievements

3.4 Тестування

Тестування програмного забезпечення є важливим етапом у розробці додатків, який забезпечує якість, надійність та відповідність системи очікуванням користувачів. Основною метою тестування є виявлення помилок, перевірка коректності реалізації функціоналу та забезпечення зручності використання продукту. Для успішного виконання цього завдання

створюється комплексний план тестування, який включає перевірку як окремих компонентів, так і взаємодії між ними.

У межах фінансової системи тестування має особливе значення, оскільки дані, що обробляються, мають бути точними, а функції — коректними навіть у крайових випадках. Серед основних завдань тестування — перевірка роботи збереження даних у PlayerPrefs, функціонування методів редагування та створення карток, перевірка валідності введених даних, а також правильна робота досягнень і їх відображення в інтерфейсі.

Спочатку потрібно завантажити додаток. Перед користувачем з’явиться наступне вікно (рис. 3.29):

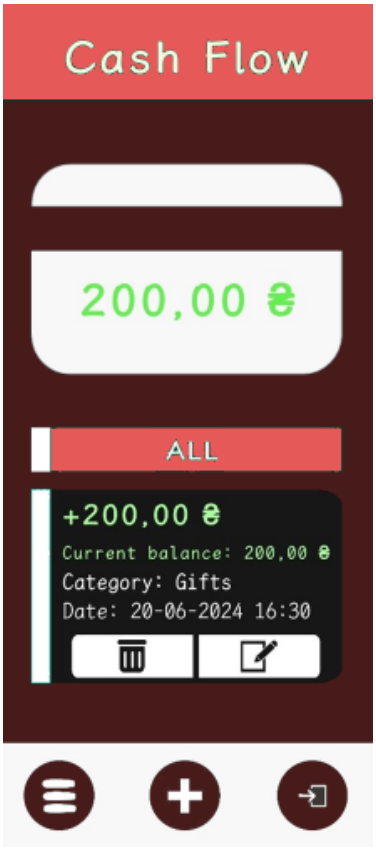

Рисунок 3.29 – Головна сторінка додатку

Для створення транзакції потрібно перейти на наступне вікно додатку. В ньому є декілька полів що використовується для зазначення суми транзакції, дата й час, категорію транзакції (рис. 3.30).

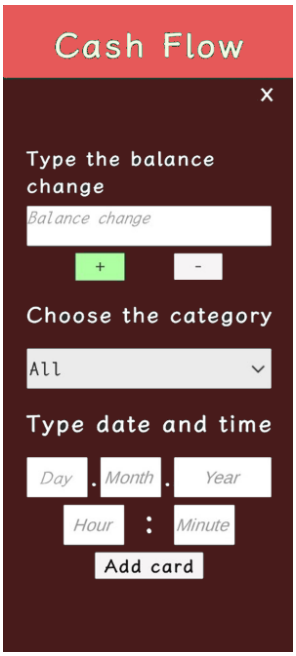

Рисунок 3.30 – Картка транзакцій

Після введення даних користувача, програма зберігає їх, та створює новий об’єкт картки.

Користувач також змінити данні транзакції включаючи або вибрати іншу категорію:

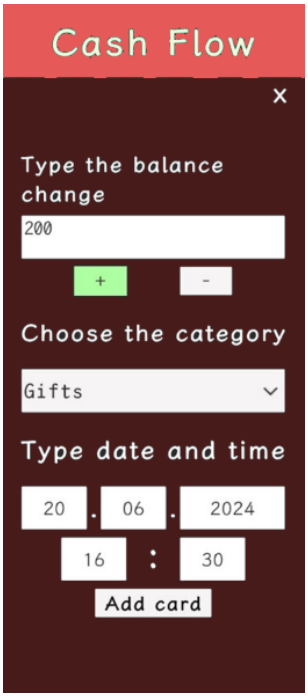

Рисунок 3.31 – Зміна даних картки транзакції

3.5 Висновок третього розділу

У третьому розділі було проведено розробку системи управління фінансами, що включала створення кросплатформенного додатку з використанням Unity. Було визначено актуальність такої розробки, враховуючи потребу у швидких та точних інструментах для аналізу фінансових операцій. Представлена структура проєкту та реалізація основних функціональних модулів, включаючи управління інтерфейсом, валідацію даних та обробку транзакцій.

Результатом стало створення інтерактивного інтерфейсу, що забезпечує зручність використання, підтримку сучасних технологій анімації та інтеграцію з базовими функціями зберігання даних. Розроблені методи та класи дозволяють користувачам ефективно контролювати фінансові операції, забезпечуючи при цьому стабільність та доступність додатку на різних платформах. Такий підхід відповідає сучасним вимогам до розробки програмного забезпечення і закладає основу для подальшого розширення функціоналу системи.

ВИСНОВОК

У ході виконання цієї роботи було проведено комплексний аналіз і розробку кросплатформеного додатка для управління фінансами. Основною метою проєкту було створення програмного забезпечення, яке забезпечує зручний інтерфейс, ефективне управління фінансами та підтримку сучасних стандартів розробки.

Було детально розглянуто принципи побудови кросплатформенних додатків, що дозволило обґрунтовано вибрати платформу Unity для реалізації проєкту. Такий підхід забезпечує високу продуктивність, доступність на різних операційних системах та можливість інтеграції з сучасними технологіями.

Вибір платформи Unity та мови програмування C# був зумовлений їхніми перевагами, зокрема:

- Кросплатформеністю – підтримкою розгортання на Android, iOS та інших платформах.
- Об’єктно-орієнтованим підходом – що сприяє структурованості коду та зручності його підтримки.
- Потужними інструментами розробки – такими як Unity Editor, що забезпечує інтерактивне тестування та візуальне створення інтерфейсу.
- Гнучкістю – можливістю налаштування функціоналу залежно від потреб користувачів.

Практичний результат роботи включає створення функціонального програмного забезпечення, яке складається з:

1. Модуля управління транзакціями, що забезпечує додавання, редагування та видалення фінансових операцій.
2. Модуля збереження даних, реалізованого за допомогою PlayerPrefs і JSON для зберігання локальних даних.
3. Інтерфейсу користувача, який забезпечує інтуїтивну взаємодію з додатком.

Розроблена система відповідає сучасним стандартам розробки програмного забезпечення, демонструє стабільну роботу в умовах різноманітних сценаріїв використання, а також забезпечує масштабованість і високий рівень безпеки.

Досягнуті результати підтверджують успішне виконання поставлених завдань, зокрема:

- Проведення аналізу сучасних програм для управління фінансами.
- Вибір оптимальних технологій для реалізації проєкту.
- Розробку програмного продукту, який забезпечує зручність і ефективність управління фінансами.

Таким чином, створений додаток може бути використаний для покращення фінансової грамотності, спрощення управління фінансами та підвищення зручності користувачів у повсякденному житті.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Chechetova, N. F. & Chechetova-Terashvili, T. M. (2019). Financial literacy as a key to the success of personal finance management. World Science, 2(10(50), 14-20. http://doi.org/10.31435/rsglobal_ws/31102019/6725.

2. Zhuravliov O. V., Simachov O. A. Statystychne doslidzhennia rynku IT-posluh v Ukraini. Statystyka Ukrainy. 2018. № 4. S. 25–33.

3. Babyre, O. V. (2017). Gamification as an instrument of persuasion in the context of environmental culture. Movni i kontseptualni kartyny svitu, (59), 21-26

4. Unity 3D – [Електронний ресурс] – Режим доступу: https://docs.unity.com

5. Офіційний документація C# – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview

6. Cross platform app frameworks in 2021 – [Електронний ресурс]. – Режим доступу: https://www.netsolutions.com/insights/cross-platform-appframeworks-in-2021

7. Розробка веб-додатків, мобільних додатків та порталів: [Електронний ресурс]. Режим доступу: http://ittel.com.ua/informacijnitexnologiyi/ rozrobka-mobilnih-dodatkiv/

8. Технології створення мобільних додатків [Електронний ресурс]. – Режим доступу: http://group-global.org/publication/63343-tehnologiisozdaniyamobilnyh-prilozheniy

9. Forbes: What Is A Financial Plan, And Why Every Adult Needs One. [Електронний ресурс]: – Режим доступу: https://www.forbes.com/sites/ lizfrazierpeck/2018/03/31/what-is-a-financial-plan-and-why-every-adult-needsone/?sh=5d15f05158be

10. Android Money. – [Електронний ресурс]: – Режим доступу: https://play.google.com/store/apps/details?id=com.money.manager.ex.android&hl=en

11. Expense Manager. – [Електронний ресурс]: – Режим доступу: https://play.google.com/store/apps/details?id=com.expensemanager&hl=en

12. Expense Tracker - FinancePM. – [Електронний ресурс]: – Режим доступу: https://play.google.com/store/apps/details?id=com.finperssaver&hl=en

13. CoinKeeper – фінанси та бюджет. – [Електронний ресурс]: – Режим доступу: https://apps.apple.com/ua/app/coinkeeper-финансы-ибюджет/id1335547405

14. Monefy – облік витрат. – [Електронний ресурс]: – Режим доступу: https://apps.apple.com/ru/app/monefy-учет-расходов/id1212024409

15. Wallet. – [Електронний ресурс]: – Режим доступу: https:// apps.apple.com/ua/app/wallet/id1093713971