НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
(назва кафедри)

_____ Голуб Б.Л.,доц.,к.т.н.
(підпис) (ПІБ)

"2" червня 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

Програмне забезпечення для планування та аналізу особистих фінансів

Спеціальність 121 – «Інженерія програмного забезпечення»
ОПП - «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ Вайганг Г.О.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи
к.е.н. ст., викладач _____ Ніколаєнко Д.В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Ніщименко А.П.
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерних наук

_____ / Голуб Б.Л., доцент, к.т.н /
підпис

"_2_" червня 2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студенту Ніщименку Антону Павловичу
Спеціальність 121 «Інженерія програмного забезпечення»
1. Тема роботи: Програмне забезпечення для планування та аналізу особистих фінансів
Затверджена наказом ректора НУБіП України № 2248 "С" від 16.12.2024

2. Термін подання завершеної роботи на кафедру 2025 . 05 . 25
рік, місяць, число

3. Вихідні дані до роботи: опис програмного забезпечення
4. Перелік питань що розглядаються:
   1. Системний аналіз предметної області.
   2. Інформаційне забезпечення.
   3. Прикладне програмне забезпечення.
   4. Рекомендації щодо впровадження та експлуатації системи
   5. Висновки.

Керівник бакалаврської кваліфікаційної роботи_____ / Ніколаєнко Д.В./
підпис ініціали та прізвище

Завдання прийняв до виконання _____ / Ніщименко А.П. /
підпис ініціали та прізвище

Дата отримання завдання 2024 . 12 . 16
рік, місяць, число

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

- API (Application Programming Interface) – інтерфейс програмування застосунків
- DB (Database) – база даних
- IDE (Integrated Development Environment) – інтегроване середовище розробки
- ORM (Object-Relational Mapping) – об'єктно-реляційне відображення
- SQL (Structured Query Language) – мова структурованих запитів
- UI (User Interface) – користувацький інтерфейс
- UX (User Experience) – користувацький досвід
- XML (eXtensible Markup Language) – розширювана мова розмітки
- SDK (Software Development Kit) – набір засобів розробки програмного забезпечення
- SQLite – легка бібліотека баз даних SQL
- Kotlin – мова програмування створена, як сучасна альтернатива Java, яка використовується для розробки під Android
- SQLite – Вбудована база даних, яка широко використовується в Android додатках
- ПЗ – програмне забезпечення
- БД – база даних

ВСТУП

Сьогодні інформаційні технології пронизують усі сфери життя, стаючи незамінним інструментом як для виконання повсякденних завдань, так і для автоматизації складних процесів. Управління особистими фінансами — одна з таких галузей, де IT-рішення дозволяють не лише спростити контроль за доходами і витратами, а й підвищити ефективність прийняття фінансових рішень.

Фінансове планування стало необхідною частиною сучасного способу життя. Люди постійно взаємодіють із грошовими потоками: формують бюджети, аналізують витрати, накопичують заощадження та ухвалюють рішення щодо інвестування. У цьому контексті програмні інструменти виконують роль цифрових помічників — вони структурують дані, автоматизують облік і допомагають побачити фінансову картину більш цілісно.

Управління особистими фінансами зазвичай складається з декількох етапів: фіксацію вхідних даних, їх аналітичну обробку, побудову бюджету, моніторинг його дотримання і, за потреби, — корекцію стратегії. Класичні підходи — наприклад, ручний запис у блокнотах чи таблицях — давно втратили актуальність: вони потребують багато часу, є схильними до помилок і не дозволяють оперативно реагувати на зміну фінансової ситуації.

Запровадження спеціалізованого програмного забезпечення дозволяє суттєво підвищити точність, швидкість та надійність фінансового обліку. Сучасні застосунки не лише автоматизують введення даних, а й надають потужні інструменти для аналізу, візуалізації та планування бюджету. Інтерфейси таких систем зазвичай інтуїтивно зрозумілі, що робить їх доступними навіть для користувачів без технічного досвіду.

Мета роботи - розробка програмного продукту, що дозволяє зручно й ефективно керувати особистим бюджетом. Система має забезпечувати

автоматизований облік доходів і витрат, аналіз фінансової інформації та підтримку інструментів для побудови й корекції бюджету.

Робота структурована наступним чином. У першому розділі подано огляд предметної області, проаналізовано існуючі програмні рішення та сформульовано основні завдання дослідження. Другий розділ присвячений побудові моделей системи: у ньому представлено діаграми варіантів використання, активностей, послідовностей і класів. Третій розділ охоплює етап проектування: створено логічну й фізичну моделі даних, визначено архітектуру системи та обґрунтовано вибір технічних засобів. У четвертому розділі описано процес впровадження системи, її тестування та ключові аспекти експлуатації програмного забезпечення.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Задача полягає у створенні програмного продукту на базі Android, який дозволить зручно та автоматизовано виконувати облік та планування бюджету користувача. Продукт має реалізовувати наступні функціональні вимоги:

- додавання нових записів про фінансові операції;
- можливість видалення вже наявних записів;
- редагування деталей фінансових транзакцій;
- перегляд списку доходів та витрат за вибраний часовий інтервал;
- вибір категорії для кожної фінансової операції;
- налаштування параметрів, пов’язаних із плануванням бюджету;
- можливість створення окремих скарбничок для збережень;
- поповнення скарбнички;
- розбиття скарбнички;
- оновлення інформації щодо фінансових дій користувача.

Інтерфейс головного екрана додатку включатиме елементи, що містять такі дані:

- порядковий номер запису;
- короткий опис операції;
- зазначення суми доходу чи витрат;
- категорія, до якої належить операція;
- дата проведення операції;
- загальна кількість фінансових записів;
- аналітика щодо витрат та доходів;
- інформація про пріоритети фінансових завдань;
- дані про зміни в бюджеті;

- графічне представлення категорій у вигляді іконки або фотографії;
- умови, за яких була здійснена операція;
- поточний стан балансу користувача.

Програмний продукт буде мати простий і зрозумілий інтерфейс, орієнтований на кінцевого користувача. У головному вікні будуть розміщені кнопки для зручного керування фінансами, що забезпечують легке користування без потреби в додаткових інструкціях.

Результатом роботи буде створений програмний продукт, що дозволить ефективно розподіляти власні кошти та збереження. Це сприятиме кращому плануванню бюджету та досягненню поставлених фінансових цілей.

1.2 Аналіз вимог до програмної системи

Аналіз вимог є одним з головних етапів у створенні будь-якого програмного продукту. Саме на цьому етапі формується уявлення про функціонал, можливості та обмеження майбутньої системи. Від того, наскільки чітко й повно окреслено вимоги, залежить відповідність розробленого ПЗ очікуванням користувачів і поставленим бізнес-цілям.

У процесі аналізу проводиться збір, фіксація та систематизація як функціональних, так і нефункціональних вимог. Окрім цього, визначаються технічні аспекти, що впливають на стабільність роботи системи, її безпечність і зручність для кінцевого користувача. Якісне опрацювання вимог — це запорука створення дійсно корисного та ефективного програмного забезпечення, орієнтованого на реальні потреби.

Нижче подано перелік ключових вимог, які було враховано під час розробки даного застосунку.

Розглянемо детальніше функціональні вимоги у табл.1.1.:

Таблиця 1.1.

Функціональні вимоги до програмної системи

№	Функціональна вимога	Опис
1	Реєстрація та аутентифікація користувачів	Користувач має можливість зареєструвати обліковий запис, вказавши своє ім’я, адресу електронної пошти та пароль, після чого може авторизуватись у системі й за потреби скористатися функцією відновлення пароля.
2	Облік доходів та витрат	Додавання, редагування, видалення записів про доходи та витрати з можливістю вказати категорію, суму, дату та опис.
3	Бюджетування	Встановлення місячного бюджету для кожної категорії витрат та перегляд залишку бюджету.
4	Фінансові звіти та аналіз	Можливість переглядати фінансові звіти про доходи та витрати за обрані періоди часу, а також отримувати наочні візуалізації у формі графіків і діаграм.

Нефункціональні вимоги визначають якісні характеристики програмного забезпечення, які не стосуються конкретного функціоналу, але безпосередньо впливають на зручність, надійність і ефективність його використання. Вони охоплюють аспекти, що стосуються продуктивності системи, її безпеки, масштабованості, сумісності з іншими системами, доступності. Саме ці вимоги визначають, наскільки комфортним буде досвід користувача та наскільки стабільно працюватиме програма в різних умовах. Розглянемо детальніше нефункціональні вимоги у табл. 1.2.:

Таблиця 1.2.

Нефункціональні вимоги програмної системи

№	Нефункціональна вимога	Опис
1	Зручність користування	Зручний та інтуїтивний інтерфейс, що забезпечує легкий доступ до основних функцій без потреби у додаткових поясненнях чи інструкціях.
2	Безпека даних	Забезпечення конфіденційності особистої інформації користувачів шляхом використання шифрування даних та обмеження доступу.
3	Надійність	Надійне функціонування системи з максимальною доступністю та мінімальними перервами в роботі.
4	Масштабованість	Можливість розширення системи у разі збільшення кількості користувачів або обсягу даних
5	Інтеграція	Здатність інтеграції з іншими системами для обміну даними
6	Відповідність стандартам	Дотримання вимог національних і міжнародних стандартів у сфері захисту даних та інформаційної безпеки.

Технічні вимоги є набором умов, які повинні бути виконані для розробки програмного продукту або системи. Вони визначають архітектуру, функціональність та інші технічні аспекти розробки ПЗ.

Основна мета цього додатку - надання користувачам зручного та ефективного інструменту для управління їхніми фінансами. Для досягнення цієї мети необхідно враховувати важливі технічні аспекти розробки, зокрема оптимізацію продуктивності, забезпечення безпеки даних та сумісність з різними пристроями і версіями операційної системи Android.

Програмне забезпечення буде розроблене з використанням мови програмування Kotlin, яка є офіційною мовою для розробки Android-додатків. Kotlin має багато переваг, таких як безпека типів, висока продуктивність і підтримка сучасних методик програмування.

Однією з ключових технічних вимог є підтримка різних версій операційної системи Android. Додаток повинен бути сумісним з широким спектром версій Android, від найстаріших до найновіших, забезпечуючи доступність для якомога більшої кількості користувачів.

Також важливим аспектом є безпека даних. Додаток повинен гарантувати захист конфіденційної фінансової інформації користувачів, зберігаючи її конфіденційність та цілісність.

Оскільки існує безліч пристроїв з різними розмірами екранів, додаток повинен мати адаптивний дизайн, що забезпечить оптимальний вигляд і функціональність на всіх типах пристроїв.

Аналіз функціональних, нефункціональних та технічних вимог до програмного забезпечення для автоматизації фінансових процесів є критично важливим етапом у розробці системи. Функціональні вимоги визначають можливості та завдання, які система повинна виконувати. Нефункціональні — встановлюють параметри якості, а технічні вимоги — архітектуру та інфраструктуру системи.

Чітке формулювання цих вимог дозволить створити програмний продукт, який ефективно виконує свої функції, забезпечуючи зручність для користувачів, надійність, безпеку даних і масштабованість.

1.3 Моделювання предметної області

Моделювання предметної області є важливим етапом у розробці програмного модуля для забезпечення інтелектуальної власності, що має вирішальне значення для побудови системи. Цей крок передбачає виявлення основних сутностей та встановлення взаємозв'язків між ними, що стає основою для подальшого проектування. У цьому розділі ми розглянемо ключові сутності та їх взаємодії в контексті нашої предметної області, що дозволить краще зрозуміти об'єкти та процеси, які реалізуються в системі.

UML (Unified Modeling Language) є стандартною мовою візуального моделювання, яка широко застосовується для опису, планування, проектування та документування програмних систем і їхніх окремих компонентів. Вона надає можливість створювати графічні, логічні та концептуальні моделі складних програмних рішень, відображаючи як структурні, так і поведінкові характеристики систем.

Мова UML ґрунтується на загальних принципах моделювання складних систем та об'єктно-орієнтованого проектування. Вона дозволяє використовувати принципи абстрагування, що спрощують процес моделювання та покращують розуміння системи. Також UML сприяє покращенню комунікації між членами команди та замовником завдяки графічному відображенню структури та поведінки системи. Розглянемо докладніше діаграму прецедентів.

Відношення між акторами та прецедентами в системі. Актори представляють сутності, що знаходяться за межами системи. Прецеденти в системі дозволяють специфікувати функціональну поведінку розроблюваної системи та отримати відповідь на запитання, що має робити системи, проте не визначають реалізацію цієї поведінки системи – не торкаються питань, яким чином відповідні функції реалізуються [7].

Діаграма прецедентів є графом, що складається з множини акторів, прецедентів (варіантів використання) обмежених границею системи (прямокутник), асоціацій між акторами та прецедентами, відношень серед

прецедентів, та відношень узагальнення між акторами [7]. Діаграми прецедентів відображають елементи моделі варіантів використання.

Для представлення предметної області було використано 2 актори та 7 прецедентів: користувач, фінансовий менеджер. Актори:

1) Користувач – особа якій необхідні операції з фінансами

2) Фінансовий менеджер.

Розглянемо детальніше на рис.1.1.

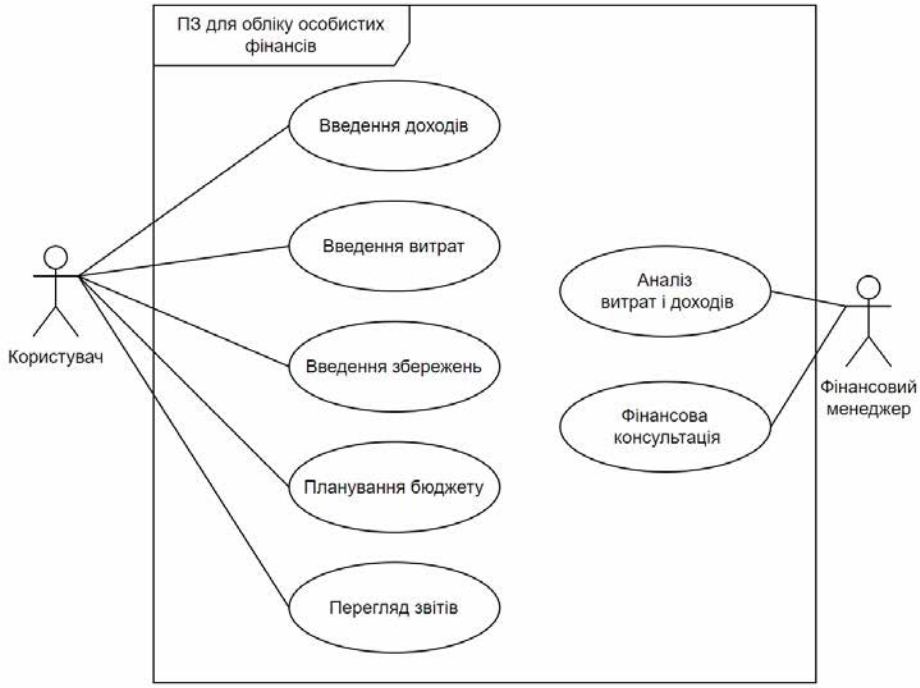

Рис.1.1 Діаграма прецедентів програмної системи

Діаграма послідовності є одним із ключових інструментів мови UML, який використовується для моделювання динаміки взаємодії між об’єктами в системі. Вона відображає, у якій послідовності відбуваються обміни повідомленнями між об'єктами в рамках певного сценарію використання. Основними елементами діаграми є об’єкти (актори або компоненти системи), їхні "лінії життя", а також повідомлення, що передаються між ними. Така візуалізація дозволяє розробникам детально простежити логіку взаємодії компонентів і зрозуміти часову послідовність подій.

Діаграми послідовності дозволяють уточнити діаграми прецедентів, детальніше описуючи логіку сценаріїв використання. Вони показують об'єкти, які взаємодіють у межах сценарію, обмінювані повідомлення та результати цих взаємодій. Діаграми послідовності мають два виміри: горизонтальний, де відображаються "лінії життя" окремих об'єктів у вигляді вертикальних ліній, та тимчасовий вимір, що показує порядок взаємодій між об'єктами.

Розглянемо детальніше діаграму послідовності на рис. 1.2.

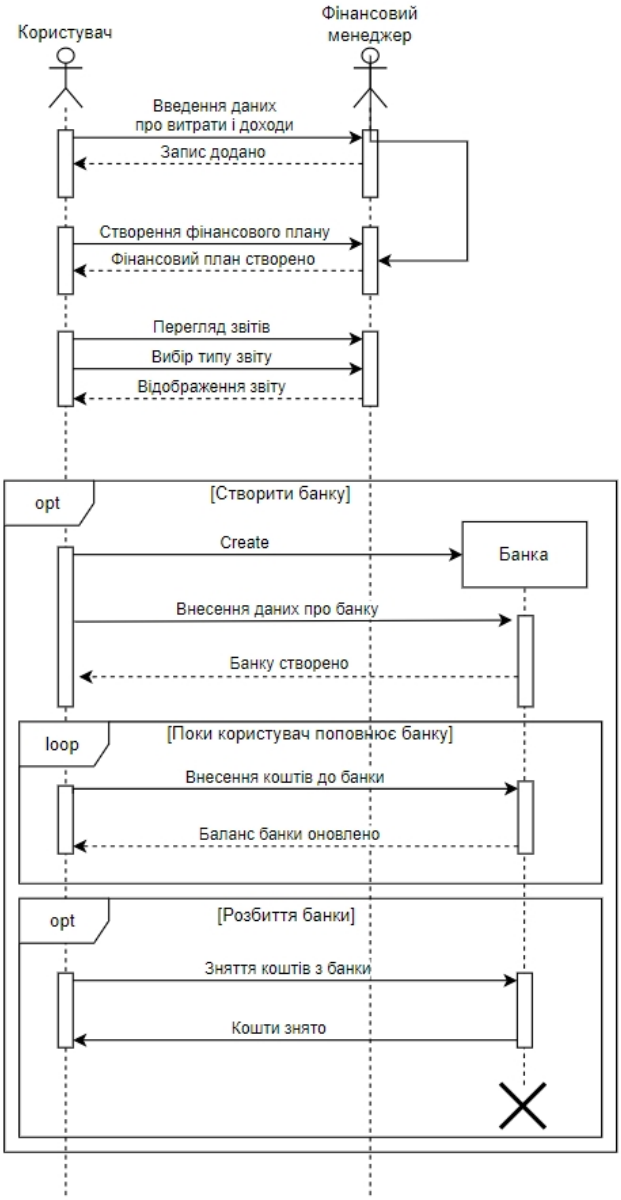

Рис. 1.2 Діаграма послідовності програмної системи

На діаграмі відображено 2 актори. Користувач має можливість вносити дані про доходи та витрати, створювати фінансовий план, керувати

скарбничками. Фінансовий менеджер виступає у ролі системи та оброблює запити користувача для планування фінансів.

Діаграма активності є графічним представленням алгоритму або бізнес-процесу, що дозволяє наочно відобразити послідовність дій, які виконує система або користувач під час виконання певної функції. Вона демонструє логіку переходів між станами, точки прийняття рішень, розгалуження та паралельне виконання процесів. У контексті розробки програмного забезпечення така діаграма допомагає краще зрозуміти внутрішню поведінку системи, оптимізувати робочі потоки та виявити потенційні місця для покращення або автоматизації.

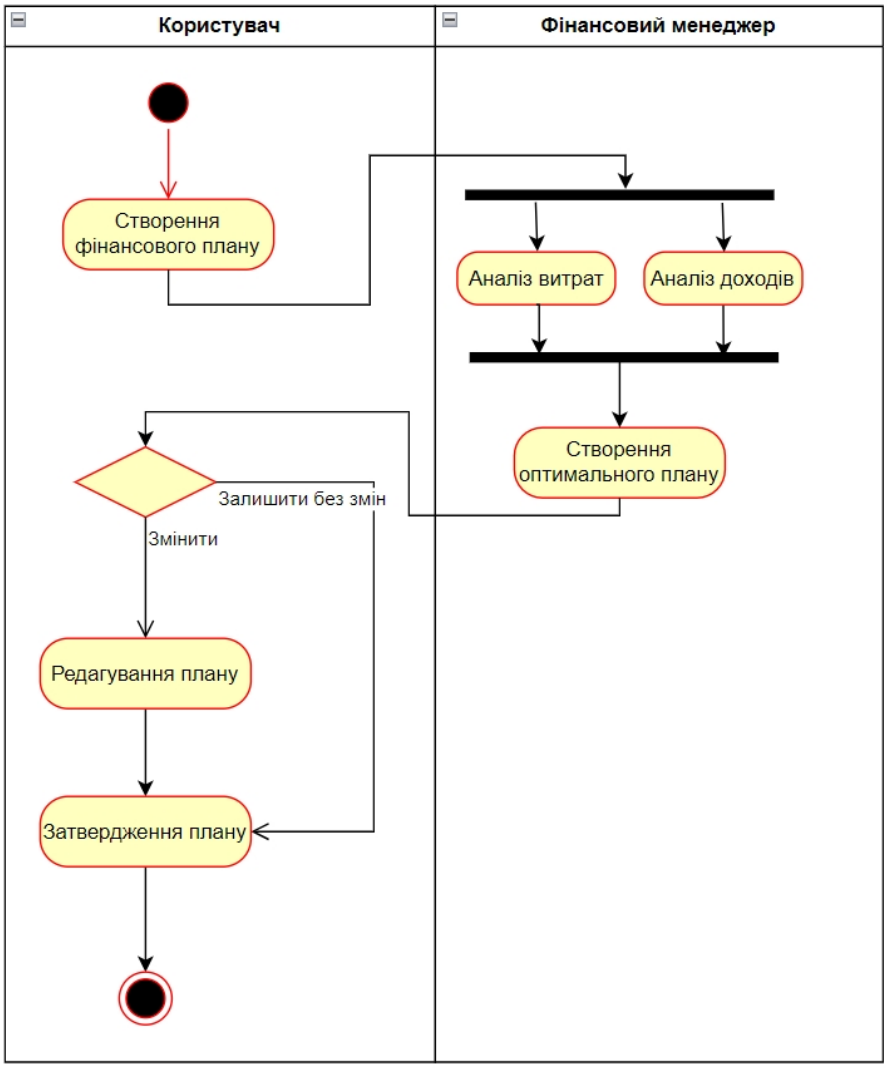

Рис.1.3 Діаграма активності програмної системи

1.4 Огляд інформаційних джерел та існуючих рішень

Огляд інформаційних джерел та існуючих рішень полягає у вивченні теоретичних матеріалів, наукових праць, технічної документації та прикладних програм, що вже реалізують подібний функціонал. Цей етап дає змогу краще зрозуміти предметну область, оцінити сучасні тенденції у розробці програмного забезпечення для фінансового планування, а також виявити переваги й недоліки конкурентних продуктів. На основі такого аналізу формуються вимоги до майбутньої системи, визначаються можливості для вдосконалення й унікальні риси, які можуть вирізнити розроблений продукт серед інших.

Monefy – додаток, що реалізовує швидке додавання доходів і витрат з поділом по категоріях. Інтерфейс орієнтований на мінімалізм, що дозволяє легко аналізувати фінансові потоки за допомогою діаграм та списків.

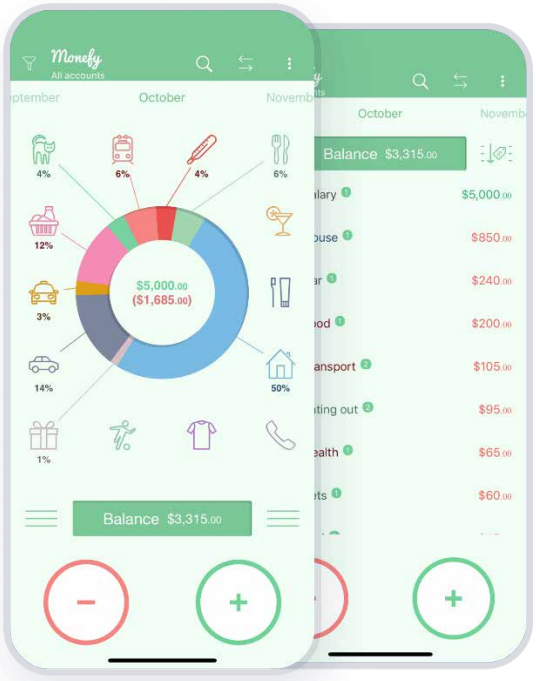

Рис.1.4 Інтерфейс програми Monefy

Основна функціональність Monefy включає наступні можливості:

1. Швидке додавання операцій: дозволяє вносити витрати та доходи за кілька секунд.

2. Візуалізація даних: демонструє фінансову активність у вигляді кругових діаграм.

3. Налаштування категорій: підтримує створення та редагування власних категорій витрат.

4. Синхронізація: забезпечує передачу даних між пристроями через Google Drive або Dropbox.

5. Підтримка валют: дозволяє працювати з кількома національними та іноземними валютами.

6. Захист доступу: реалізована функція блокування додатку паролем або PIN-кодом.

Wallet (by BudgetBakers) - пропонує автоматичне імпортування транзакцій з банківських рахунків, ведення бюджету, планування витрат та створення фінансових цілей. Додаток підтримує хмарне збереження даних і спільне використання бюджету кількома користувачами.

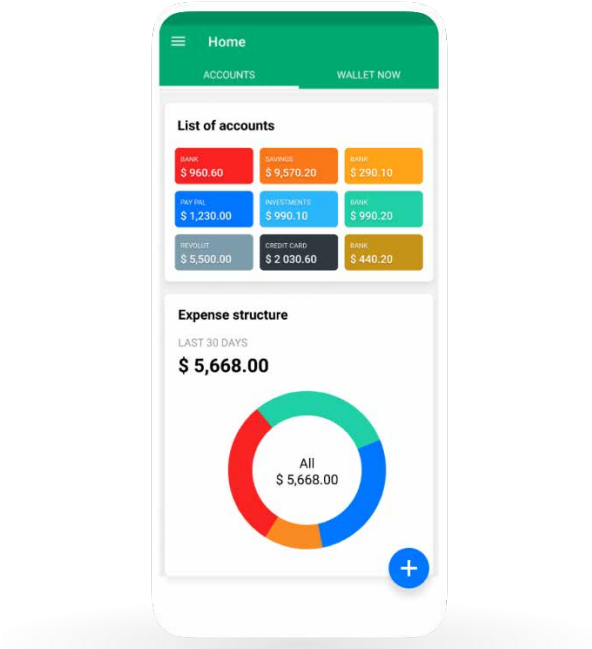

Рис.1.5 Інтерфейс програми Wallet

1. Автоматичний імпорт транзакцій: синхронізується з банками та підтягує витрати автоматично.
2. Планування бюджету: дозволяє створювати фінансові плани з лімітами на певні періоди.
3. Звіти та аналітика: формує графіки, таблиці та статистику витрат/доходів.
4. Фінансові цілі: дає змогу накопичувати кошти на конкретні потреби.
5. Спільний доступ: підтримує управління бюджетом декількома користувачами.
6. Хмарне збереження: забезпечує безпечну синхронізацію та резервне копіювання.
7. Нагадування: надсилає сповіщення про регулярні або майбутні платежі.

Money Manager - основна функціональність охоплює деталізований облік фінансів, підтримку кількох рахунків, управління активами та створення фінансових звітів. Додаток також має можливість експортувати дані та працювати з паролем для безпеки.

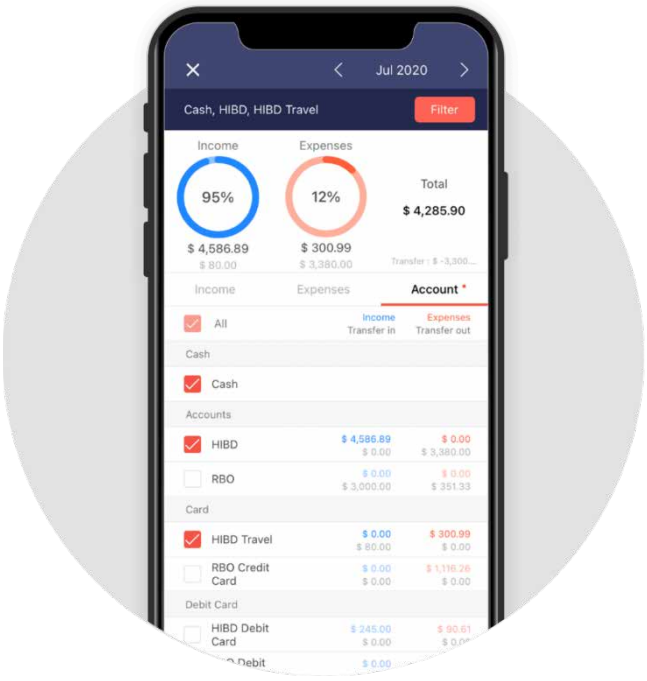

Рис.1.6 Інтерфейс програми Money Manager

1. Облік доходів і витрат: дозволяє зручно фіксувати фінансові операції з деталізацією.
2. Управління рахунками: підтримує ведення кількох гаманців, карток або банківських рахунків.
3. Групування даних: сортує транзакції за датами та періодами для зручного перегляду.
4. Візуалізація: будує графіки та діаграми фінансової активності.
5. Замітки до транзакцій: дозволяє додавати пояснення чи коментарі до кожної операції.
6. Експорт даних: підтримує вивантаження звітів у форматах Excel або CSV.
7. Безпека: забезпечує обмеження доступу через пароль або код.

1.5 Постановка завдання

Метою розробки програмного забезпечення є створення інструменту для планування, аналізу та контролю особистих фінансів користувачів. Програма повинна автоматизувати процеси введення, обліку та аналізу фінансових даних, забезпечуючи зручність, точність і доступність для користувачів.

Для реалізації поставленої мети необхідно вирішити такі основні завдання:

- Розробка інтерфейсу користувача: створити зручний та інтуїтивно зрозумілий інтерфейс для введення даних про доходи та витрати, перегляду звітів, планування бюджету та аналізу фінансової ситуації.
- Автоматизація введення фінансових даних: реалізувати функцію швидкого введення операцій з можливістю автоматичного визначення категорій витрат і доходів.
- Створення системи категоризації: передбачити можливість для користувача створювати та редагувати категорії витрат і доходів.
- Формування фінансових звітів: розробити функціонал для створення різноманітних звітів і графіків.
- Планування бюджету: забезпечити можливість встановлення фінансових лімітів на різні категорії витрат і візуалізацію їх виконання. Користувач має отримувати сповіщення про перевищення запланованих витрат.
- Безпека даних: реалізувати заходи безпеки для захисту персональних даних користувача.
- Забезпечення надійної роботи програми: реалізувати перевірку введених даних для уникнення помилок, а також тестування та оптимізацію коду для стабільної роботи програми.

Програма повинна бути легко масштабованою для додавання нових функцій у майбутньому, зокрема можливості інтеграції з іншими фінансовими сервісами або банківськими платформами.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних у вигляді ER-діаграми

ERwin — це програмне забезпечення для проектування баз даних, яке широко використовується в сфері управління даними та розробки складних інформаційних систем. Його основне призначення — створення, моделювання та управління реляційними базами даних, що дозволяє автоматизувати багато аспектів розробки і підтримки баз даних. ERwin дає змогу проектувати як логічні, так і фізичні моделі баз даних, що дозволяє забезпечити ефективне зберігання і обробку даних, а також оптимізувати структуру бази для високої продуктивності.

Однією з ключових функцій ERwin є можливість зворотного інженерства. Це дозволяє імпортувати вже існуючі бази даних і генерувати моделі, що відповідають їхній структурі. Така можливість корисна для оновлення або модернізації існуючих систем без потреби в їх повному перепроектуванні. Зворотне інженерство допомагає зберегти зв’язок між моделями і фактичними структурами баз, що забезпечує точність у відтворенні даних і знижує ризик помилок при внесенні змін.

Завдяки автоматичній генерації SQL-коду, ERwin значно прискорює процес створення бази даних. Після створення моделі, інструмент може автоматично згенерувати необхідний SQL-код для створення таблиць, індексів, зв’язків та інших елементів бази. Це дозволяє зменшити людський фактор і знизити ймовірність виникнення помилок при ручному написанні скриптів, а також забезпечує узгодженість між проектованою моделлю і реальним втіленням бази даних.

Ще однією важливою характеристикою ERwin є можливість командної роботи. Інструмент дозволяє кільком користувачам одночасно працювати над

однією моделлю бази даних, що дуже корисно для великих проектів з участю багатьох розробників і аналітиків. ERwin має вбудовані механізми для управління доступом і контролю змін, що дозволяє команді синхронізувати свою роботу і уникнути конфліктів між різними версіями моделі.

Окрім того, ERwin підтримує інтеграцію з іншими інструментами для аналізу даних, що дозволяє використовувати моделі баз даних у більш широкому контексті бізнес-аналітики. Це дає можливість покращити процес прийняття рішень на основі даних, підвищити ефективність обробки великих обсягів інформації та забезпечити більш гнучке управління даними в організації. ERwin є важливим інструментом для всіх, хто займається проектуванням баз даних, завдяки своїй здатності забезпечувати точність, ефективність і масштабованість рішень.

Створена модель програмного забезпечення для мобільного додатку з планування та аналізу особистими фінансами включає наступні вимоги:

1. Реєстрація та аутентифікація користувачів для забезпечення безпеки особистих фінансових даних.
2. Функціонал для введення та відстеження доходів та витрат з можливістю категоризації та додаткових параметрів.
3. Можливість планування та встановлення бюджетів для різних категорій витрат.
4. Генерація аналітичних звітів та графіків для візуалізації фінансових даних користувача.
5. Забезпечення можливості збереження та експорту даних для подальшого аналізу на інших пристроях чи платформах.

Розглянемо детальніше логічну модель даних на рисунку 2.1.

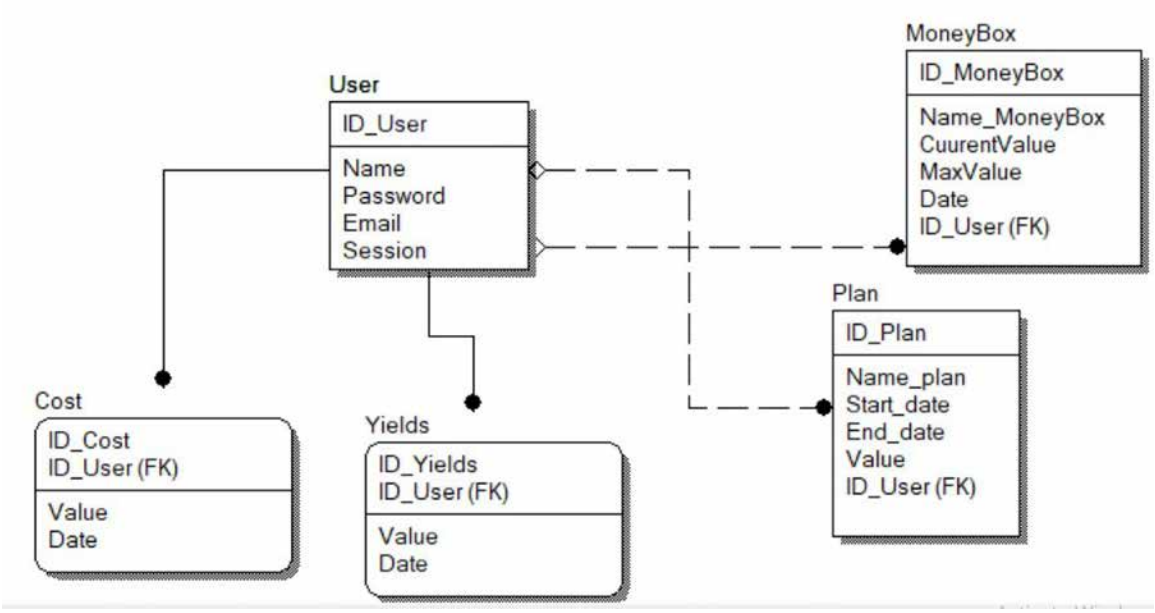

Рис.2.1 Логічна модель даних

Для побудови логічної моделі була використана ERD-модель. Відповідно цій моделі інформаційне забезпечення представлено у вигляді сутностей, кожна з яких має атрибути. Крім, того сутності пов’язані між собою.

Користувач:

- Атрибути:
  - ID_User: первинний ключ, унікальний ідентифікатор користувача.
  - Name: ім'я користувача.
  - Password: пароль користувача.
  - Email: адреса електронної пошти користувача.
  - Session: сесія користувача.

2. План:

- Атрибути:
  - ID_Plan: первинний ключ, унікальний ідентифікатор плану.
  - Name_plan: назва плану.
  - Cost: вартість плану.
  - Yields: рентабельність плану.
  - Start_date: дата початку дії плану.
  - End_date: дата закінчення дії плану.

3. Витрати:

- Атрибути:
    - ID_Cost: первинний ключ, унікальний ідентифікатор витрат.
    - Value: сума витрат.
    - Date: дата витрат.

4. Дохід:

- Атрибути:
    - ID_Yields: первинний ключ, унікальний ідентифікатор доходу.
    - Value: сума доходу.
    - Date: дата отримання доходу.

Зв'язки між сутностями:

- Користувач - План: Зв'язок "багато до одного" означає, що один користувач може мати кілька фінансових планів, але кожен план належить лише одному користувачеві. Це реалізовано через зовнішній ключ ID_User в сутності Plan, який посилається на первинний ключ ID_User в сутності User.
- План - Витрати: Зв'язок "багато до одного" показує, що один план може включати кілька витрат, але кожна витрата прив'язана лише до одного плану. Це забезпечується через зовнішній ключ ID_Plan в сутності Cost, що посилається на первинний ключ ID_Plan в сутності Plan.
- План - Дохід: Так само, як і з витратами, зв'язок "багато до одного" існує між сутностями Plan і Yields. Один план може мати кілька джерел доходу, але кожен дохід належить тільки одному плану. Це зв'язок реалізовано через зовнішній ключ ID_Plan в сутності Yields, що посилається на первинний ключ ID_Plan в сутності Plan.
- Користувач - Дохід: Зв'язок "багато до багатьох" між сутностями User і Yields відображає можливість для одного користувача отримувати дохід від кількох планів. Водночас один план може генерувати дохід для кількох користувачів. Цей зв'язок реалізується через зовнішній ключ ID_User в сутності Yields, який посилається на первинний ключ ID_User в сутності User.

2.2 Діаграма класів та кооперації

Діаграма класів є одним із важливих елементів моделювання системи, що відображає основні сутності, їхні атрибути та взаємозв'язки між ними. У контексті додатку для управління особистими фінансами, діаграма класів включає сутності, такі як User, Plan, Cost, Yields, а також їхні атрибути. Наприклад, клас User може містити атрибути, такі як ім'я, адреса електронної пошти, а клас Plan — бюджет, категорії витрат та доходів. Зв'язки між класами, такі як "багато до одного" або "багато до багатьох", визначають, як різні сутності взаємодіють між собою.

Діаграма класів (рис. 2.2) слугує для візуалізації структурної моделі системи, описуючи класи, їхні атрибути, методи та логічні зв’язки між ними. Ця діаграма допомагає визначити, які об'єкти існують у системі, як вони пов’язані між собою, і які функції виконують, що значно полегшує проектування та подальшу реалізацію. На відміну від динамічних діаграм, діаграма класів зосереджена саме на статичній структурі, без урахування часових аспектів функціонування [6].

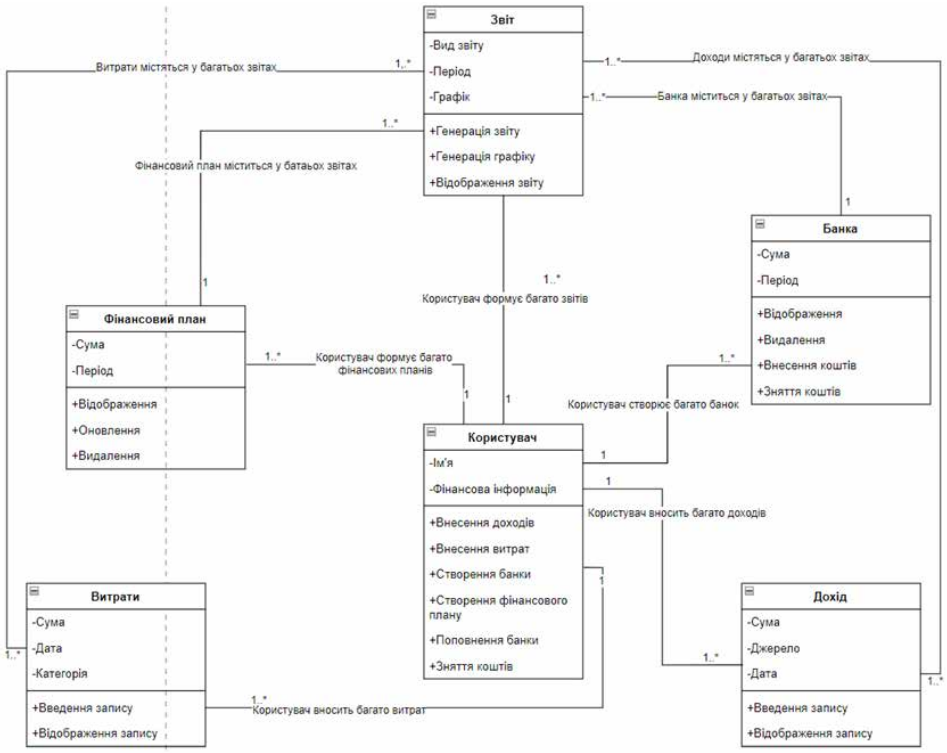

Рис.2.2 Діаграма класів додатку

Прості кооперації в UML описують взаємодії між об'єктами чи класами в рамках певного сценарію, орієнтованого на досягнення конкретної цілі. Вони є формою структурованої взаємодії, яка демонструє, як об'єкти об'єднуються для виконання спільної поведінки, не змінюючи при цьому свою внутрішню структуру. Кооперації включають ролі, які об'єкти виконують, та зв’язки, що визначають взаємодію між цими ролями, і часто використовуються для побудови діаграм взаємодії або розгортання сценаріїв поведінки системи. Нижче наведено приклади простих кооперацій на основі діаграми класів (див. рис. 2.2). Розглянемо на рис. 2.3 прості кооперації програмної системи.

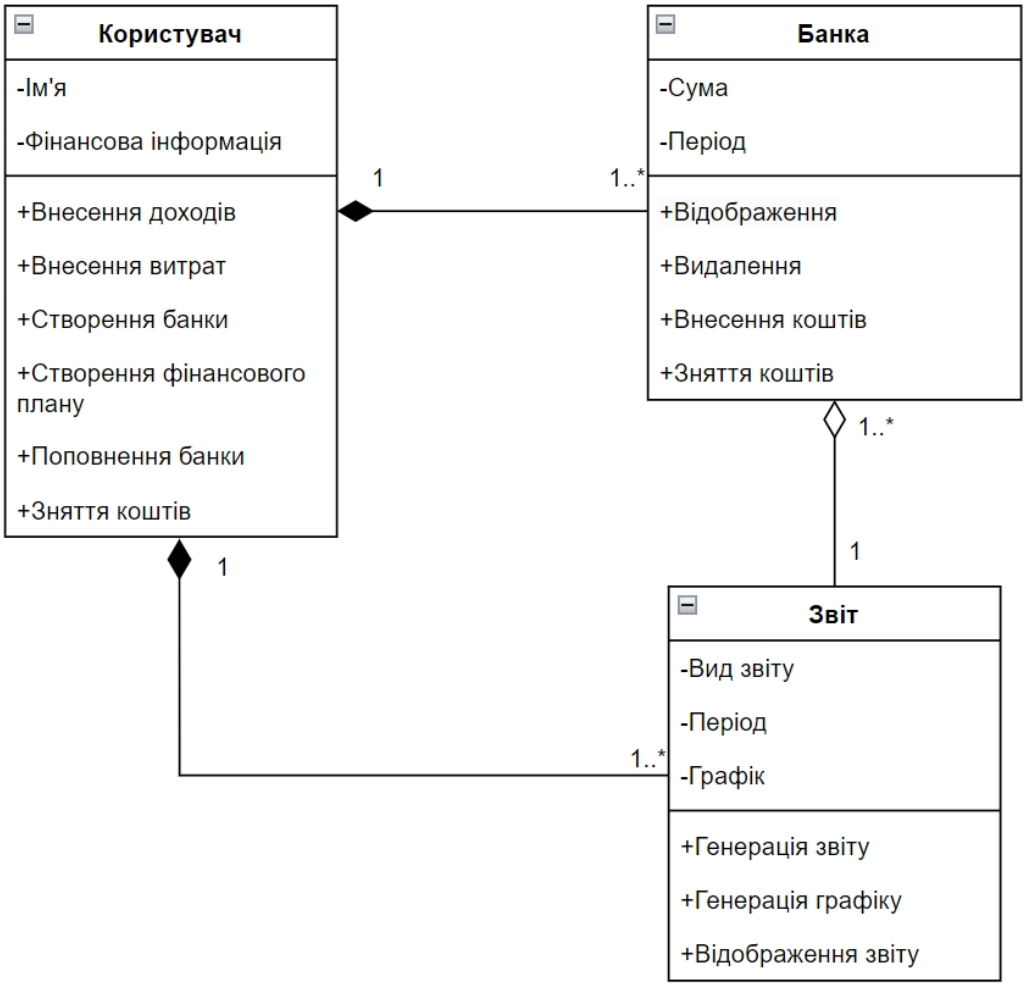

Рис. 2.3 – Проста кооперація (Користувач, Банка, Звіт)

Вище зображена проста кооперація. Вона містить 3 класи: Користувач, Банка, Звіт. Дана кооперація відображає процеси формування банки користувачем та отримання звіту. Зображені звязки:

- Користувач – Звіт: композиція – дочірній клас Звіт не може існувати без класу Користувач;
- Користувач – Банка: композиція – аналогічно до минулого зв’язку;
- Звіт – Банка: агрегація – звіт може існувати без скарбнички.

Розглянемо ще одну просту кооперацію на рис. 2.4.

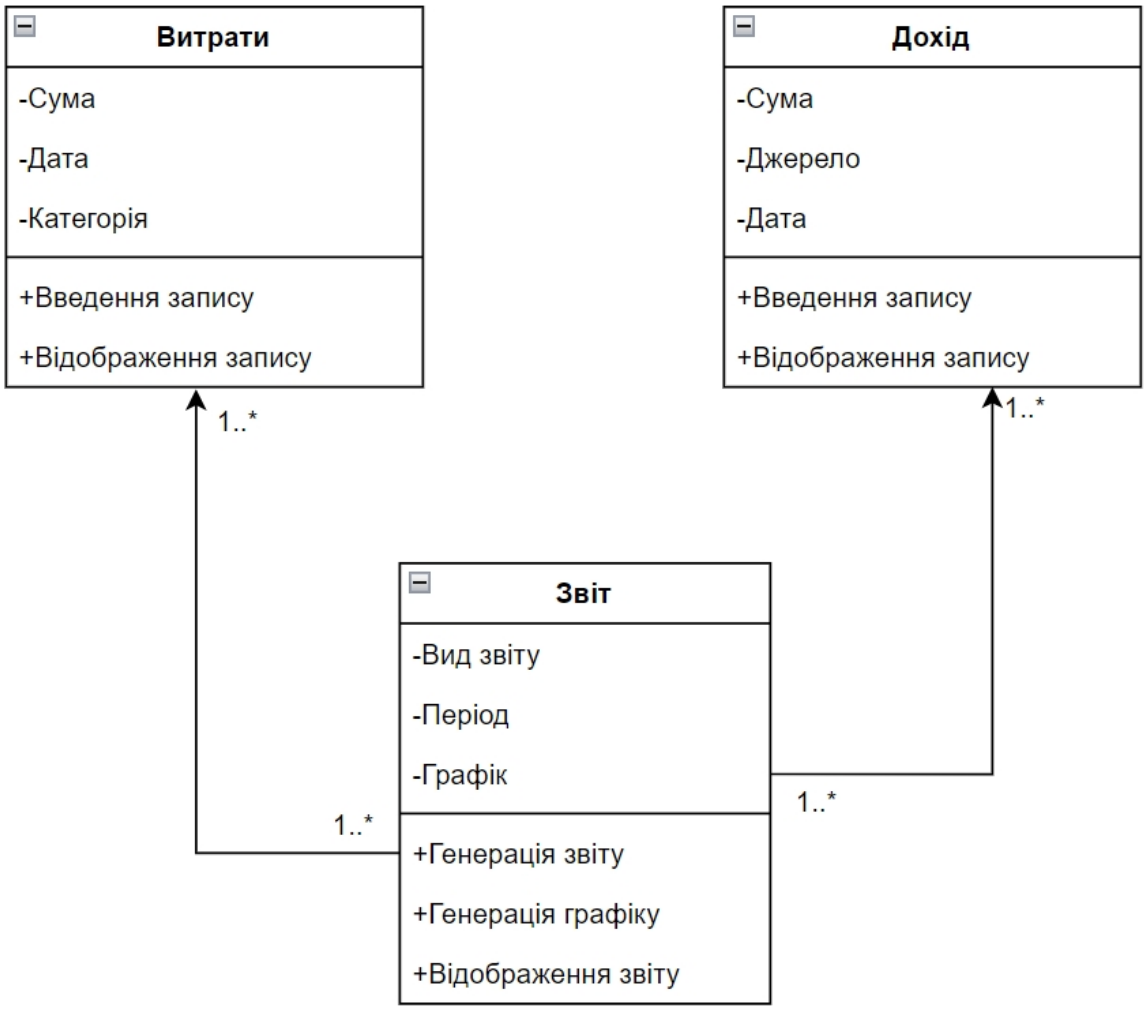

Рис. 2.4 – Проста кооперація (Дохід, Витрати, Звіт)

Наступна кооперація, яка зображена вище демонструє процес формування звітів. Вона має наступні зв’язки:

- Звіт – Витрати: агрегація;
- Звіт - Доходи: агрегація;

Розробка діаграм класів і кооперацій дозволила структуровано відобразити логіку системи та взаємозв’язки між її основними компонентами. Це допомогло чітко визначити функціональні обов’язки кожного класу, а також спростило розуміння процесів обміну інформацією між об’єктами. Такий підхід сприяє ефективному проектуванню архітектури ПЗ і знижує ризик логічних помилок на етапі реалізації.

2.3 Діаграма пакетів

Діаграма пакетів є інструментом моделювання, що використовується для логічного групування елементів системи в окремі пакети. Вона дозволяє організувати структуру проекту у вигляді модулів, де кожен пакет відповідає за певну частину функціональності або логіки. Такий підхід робить архітектуру системи більш впорядкованою та зрозумілою як для розробників, так і для аналітиків.

У контексті програмного забезпечення для планування та аналізу особистих фінансів, діаграма пакетів допомагає розділити систему на окремі функціональні блоки — наприклад, автентифікацію, управління фінансами, генерацію звітів, взаємодію з базою даних тощо. Це дозволяє спростити підтримку та розширення програми в майбутньому, адже кожен пакет може розроблятися та тестуватися незалежно.

Крім цього, діаграма чітко відображає залежності між пакетами, що дає змогу виявити зайві зв’язки або надмірну залежність одного модуля від іншого. У результаті система стає більш масштабованою, легко модифікованою та стійкою до змін, що є важливим аспектом при розробці надійного програмного продукту.

Розглянемо детальніше діаграму пакетів для програмної системи на рис. 2.5.

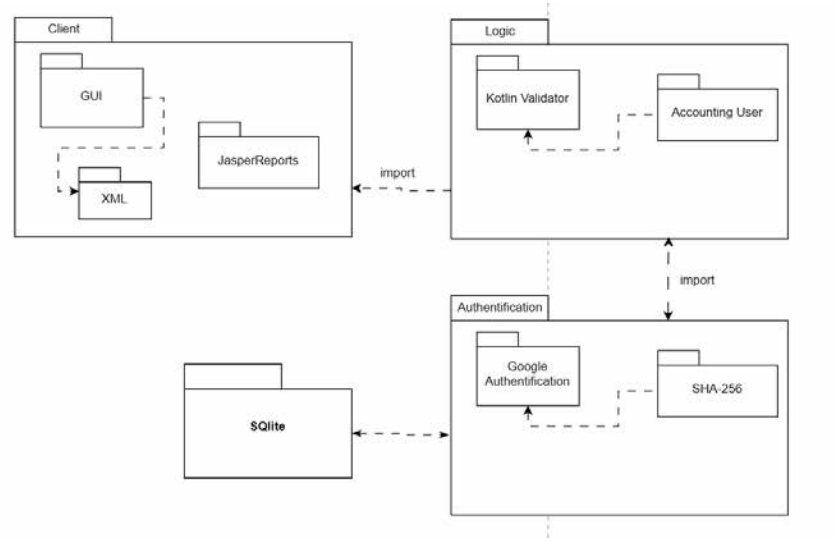

Рис. 2.5 – Діаграма пакетів програмної системи

Дана діаграма демонструє окремі пакети та зв’язки між ними:

- Client – відповідає за графічне відображення ПЗ та візуалізацію звітів.
- Logic – відповідає за логіку та алгоритми обробки запитів клієнта.
- Authentification – пакет аутентифікації в систему та шифрування.
- SQLite – пакет бази даних.

2.4 Діаграма компонентів

Діаграма компонентів є важливим інструментом візуалізації архітектури програмного забезпечення, що демонструє фізичні частини системи — її компоненти та залежності між ними. Вона фокусується не лише на логіці коду, а й на тому, як різні модулі взаємодіють між собою після компіляції. Кожен компонент представляє окрему функціональну частину системи, наприклад, бібліотеку, інтерфейс або окремий програмний модуль.

Цей тип діаграми допомагає зрозуміти структуру побудови додатку з точки зору модульності та повторного використання. Зокрема, вона дає змогу чітко побачити, які саме компоненти відповідають за взаємодію з базою даних, інтерфейсом користувача чи зовнішніми сервісами. Такий підхід спрощує підтримку й масштабування системи, особливо у великих програмних проєктах.

У контексті розробки даної системи діаграма компонентів дозволяє виявити критичні залежності між модулями, забезпечити їхню автономність і гнучкість. Це дає змогу підвищити надійність та полегшити тестування, оскільки компоненти можна розробляти й перевіряти окремо від інших частин системи.

Нижче зображено діаграму компонентів програмної системи на рис. 2.6.

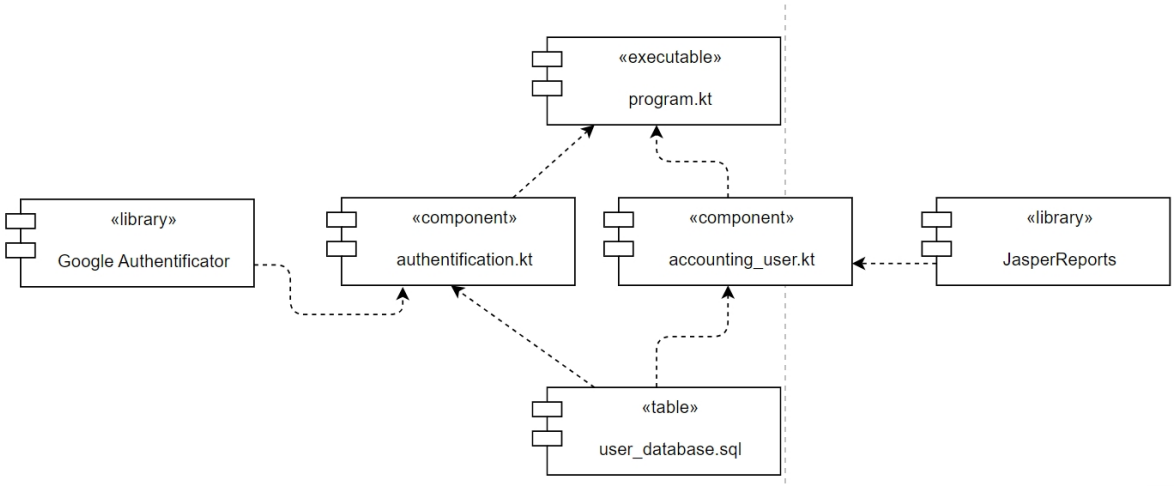

Рис.2.6. Діаграма компонентів програмної системи

Судячи з вказаної діаграми система має бути побудована з використанням модульної архітектури, де кожен компонент виконує окрему функцію. Основні компоненти — authentification.kt та accounting_user.kt — взаємодіють між собою,

а також з зовнішніми бібліотеками Google Authenticator та JasperReports та базою даних user_database.sql. Це вказує на розподілення обов’язків між компонентами: один відповідає за аутентифікацію, інший — за облік користувача.

Програма program.kt компілюється як виконуваний файл і використовує вище згадані компоненти.

Розроблена структура демонструє розділення логіки, що підвищує масштабованість, підтримуваність і безпеку додатку.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління базами даних

Для розробки вищезгаданої системи було обрано SQLite. У випадку Android-додатків вона ідеально підходить завдяки своїй легкості, автономності та мінімальним вимогам до ресурсів пристрою. SQLite не вимагає окремого сервера — вся база даних зберігається у звичайному файлі, що забезпечує просту структуру з високою продуктивністю при обробці локальних даних.

З технічної точки зору, SQLite підтримує стандарт SQL, транзакційність, індексацію та обмеження цілісності, що робить її повноцінною реляційною СУБД навіть у межах мобільного середовища. Вона дозволяє організувати надійне зберігання користувацьких даних — таких як доходи, витрати, фінансові плани — без необхідності постійного підключення до Інтернету або складної серверної інфраструктури. Це особливо корисно для додатків, які мають працювати офлайн або з нестабільним з'єднанням.

Ще однією перевагою є простота реалізації — Android SDK надає готові інструменти для роботи з SQLite через SQLiteOpenHelper, що дозволяє ефективно створювати, оновлювати та заповнювати базу даних. Крім того, існує багато бібліотек-обгорток, які спрощують розробку, автоматизують запити та допомагають уникати помилок, типових для "ручного" SQL-коду.

SQLite забезпечує повний контроль над даними на пристрої користувача, що особливо важливо з точки зору конфіденційності. Дані не передаються зовнішнім серверам без потреби і це спрощує реалізацію безпечного зберігання особистої фінансової інформації відповідно до принципів мінімізації ризиків у сфері захисту персональних даних.

3.1 Розробка інформаційної бази

Реалізацію логіки БД було виконано з допомогою мови програмування Kotlin. На рисунку 3.1 зображено запити для створення БД. Таблиці реалізації інших сутностей зображені у ДОДАТКУ А.

```
"moneyMix_db", factory, 1) {
    override fun onCreate(db: SQLiteDatabase?) {
        val createUsersQuery = """
            CREATE TABLE users (
                id INTEGER PRIMARY KEY,
                name TEXT,
                email TEXT,
                pass TEXT,
                session TEXT
            )
        """.trimIndent()
        val createLockQuery = """
            CREATE TABLE lock (
                code INTEGER
            )
        """.trimIndent()
        val createCategoryTableQuery = """
            CREATE TABLE IF NOT EXISTS category (
                id INTEGER PRIMARY KEY,
                name TEXT
            )
        """.trimIndent()
        val createYieldTableQuery = """
            CREATE TABLE IF NOT EXISTS yield (
                id INTEGER PRIMARY KEY,
                user_id INTEGER,
                value REAL,
                id_category INTEGER,
                date TEXT,
                FOREIGN KEY (user_id) REFERENCES users(id),
                FOREIGN KEY (id_category) REFERENCES category(id)
            )
        """.trimIndent()

        val createCostTableQuery = """
            CREATE TABLE IF NOT EXISTS cost (
                id INTEGER PRIMARY KEY,
                user_id INTEGER,
                value REAL,
                id_category INTEGER,
                date TEXT,
                FOREIGN KEY (user_id) REFERENCES users(id),
                FOREIGN KEY (id_category) REFERENCES category(id)
            )
```

Рис.3.1 – Створення БД розроблюваного ПЗ

Наступним етапом створення БД є ініціалізація таблиць. Її зображено на рисунку 3.2.

```
        db?.execSQL(createUsersQuery)
        db?.execSQL(createLockQuery)
        db?.execSQL(createCategoryTableQuery)
        db?.execSQL(createYieldTableQuery)
        db?.execSQL(createCostTableQuery)
        db?.execSQL(insertCategory)
        db?.execSQL(createPlanTableQuery)
        db?.execSQL(createMoneyBoxTableQuery)

    }
```

Рис.3.2 – Ініціалізація таблиць БД

На рисунку 3.2 зображено фрагмент коду Kotlin, який відповідає за виконання запитів у БД SQLite через метод execSQL використовуючи об’єкт класу db, а саме створення та ініціалізація таблиць. Кожен рядок виконує окремий запит до БД для створення окремих таблиць та зв’язків між ними.

Після створення БД важливим етапом є генерація фізичної моделі БД. Вона зображена на рисунку 3.3.

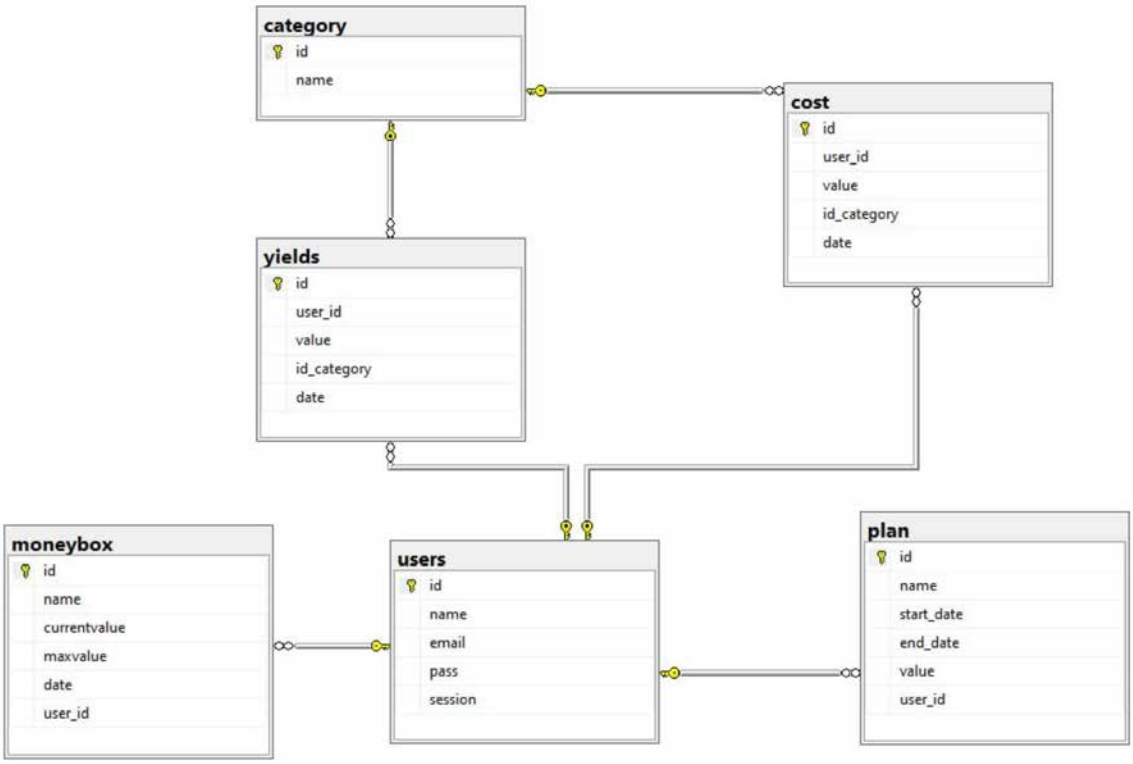

Рис.3.3 – Фізична модель даних програмної підсистеми

3.2 Вибір інструментарію для створення прикладного програмного забезпечення

Для реалізації прикладного програмного забезпечення з управління особистими фінансами було обрано поєднання таких технологій: SQLite, Kotlin, XML та Android Studio. Це рішення продиктоване поєднанням технічної доцільності, відповідності платформі Android та здатності інструментів забезпечити ефективну, стабільну і масштабовану реалізацію функціоналу.

SQLite виступає ідеальним вибором для локального зберігання даних завдяки своїй вбудованості в Android, та підтримці повноцінного SQL-синтаксису. Це дозволяє створювати цілісну структуру таблиць для користувачів, доходів, витрат, бюджетних планів без необхідності в зовнішньому серверному рішенні. База є автономною, не потребує інтернету, і забезпечує достатню швидкість обробки запитів для мобільного середовища.

Мова Kotlin обрана як основна мова програмування завдяки її сучасному синтаксису, високому рівню безпеки щодо null-значень, інтеграції з Android API та підтримці функціонального стилю програмування. Kotlin дозволяє писати компактний, читабельний і менш схильний до помилок код, що особливо важливо у розробці мобільних додатків, де стабільність — критично важлива.

XML використовується для побудови інтерфейсу користувача. Він дає змогу точно налаштувати компоненти UI, забезпечити адаптивність для різних екранів і легко вносити зміни без змішування з програмною логікою.

Android Studio обрано як основне середовище розробки, оскільки воно пропонує повний набір інструментів для Android-розробника — від візуального редактора інтерфейсу до емуляторів і дебагера. Крім того, Studio має тісну інтеграцію з Gradle, що спрощує управління залежностями, і зручні засоби для роботи з базами даних, що дозволяє ефективно тестувати й оптимізувати роботу з SQLite.

3.3 Алгоритмізація та програмування програмних модулів

Під час розробки ПЗ було реалізовано всі заплановані модулі та алгоритми. Нижче розглянемо деякі з них.

На рисунку 3.4 зображено алгоритм реєстрації користувача

```
fun addUser(user: User) : Long{
        val hashedPass = sha256(user.pass)
        val values = ContentValues()
        values.put("name", user.name)
        values.put("email", user.email)
        values.put("pass", hashedPass)

        val db = this.writableDatabase
        val success = db.insert( table: "users", nullColumnHack: null, values)
        db.close()
        return success
    } // реєстрація
```

Рис. 3.4 – Програмний код реєстрації користувача

Функція addUser виконує роль реєстрації та авторизації нового користувача. Розглянемо детальніше принцип її роботи:

1) Функція має вхідний параметр - тіло user, який містить в собі всю інформацію про користувача.
2) Першим етапом виконання функції є хешування паролю з допомогою функції sha256. Цей крок виконується для забезпечення анонімності даних користувача.
3) Створюється тіло values, в яке переносяться всі дані з user.
4) Відкриття з'єднання до БД для запису даних.
5) Виконання запиту з допомогою SQLiteHelper.
6) Закриття з’єднання з БД

Також розглянемо код алгоритму зміни паролю користувача з активною сесією на рис. 3.5.

```
fun changePassword(newPassword: String): Boolean {
    val email = getEmail()
    if (email != null) {
        val db = this.writableDatabase
        val contentValues = ContentValues()
        contentValues.put("pass", newPassword)
        val result = db.update( table: "users", contentValues,  whereClause: "email = ?", arrayOf(email))
        db.close()
        return result > 0
    }
    return false
}
```

Рис.3.5- Зміна пароля користувачем у системі

Опис роботи функції:

1) У якості параметра у функцію передається параметр newPassword
2) Визначається email user`a з активною сесією.
3) Відкриття з’єднання.
4) Виконання запиту на зміну пароля.
5) Повернення відповіді в залежності від результату.

Також розглянемо програмний код який реалізовує створення фінансового плану для користувача на рис.3.6.

```
fun insertFinancePlanData(startDate: String, endDate: String, valueCost: Double, valueYield: Double) {
    val db = this.writableDatabase

    val userId = getSessionUserId()

    val values = ContentValues().apply {
        put("start_date", startDate)
        put("end_date", endDate)
        put("value_cost", valueCost)
        put("value_yield", valueYield)
        put("user_id", userId)
    }

    db.insert( table: "financeplan",  nullColumnHack: null, values)

    db.close()
}
```

Рис. 3.6 – Фрагмент коду створення фінансового плану

Ця функція працює наступним чином:

1) Відкриває БД у режимі запису.
2) Отримує ідентифікатор користувача з активної сесії за допомогою getSessionUserId().

3) Формує об'єкт ContentValues, у який записуються: дата початку, дата завершення, значення витрат, доходів та ID користувача.
4) Виконує запит до БД на вставку даних до таблиці через метод insert.
5) Закриває з’єднання з БД.

Розглянемо наступну функцію, яка реалізовує створення скарбнички користувачем на рис. 3.7.

```
fun createMoneyBox(name: String, maxValue: Double) {
    val db = this.writableDatabase

    val userId = getSessionUserId()

    // Підготовка значень для вставки
    val values = ContentValues().apply {
        put("name", name)
        put("current_value", 0)
        put("max_value", maxValue)
        put("user_id", userId)
    }

        db.insert( table: "moneybox",  nullColumnHack: null, values)

    db.close()
}
```

Рис. 3.7 – Фрагмент коду створення скарбнички

Функція працює наступним чином:

1) Відкриває базу даних у режимі запису.
2) Отримує ідентифікатор поточного користувача через getSessionUserId().
3) Формує набір значень для нового запису скарбнички: назву, початкове значення (0), максимальну суму та ID користувача.
4) Вставляє цей запис у таблицю moneybox.
5) Закриває з'єднання з базою даних.

Інші фрагменти коду представлені у ДОДАТКУ Б.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ ПРОГРАМНОЇ СИСТЕМИ

4.1 Тестування системи

При вході в програму користувач бачить вікно авторизації. Вікно зображено на рисунку 4.1. Користувач може заповнити дані для входу та авторизуватись до системи за наявності акаунта, а також може перейти до вікна реєстрації нового облікового запису.

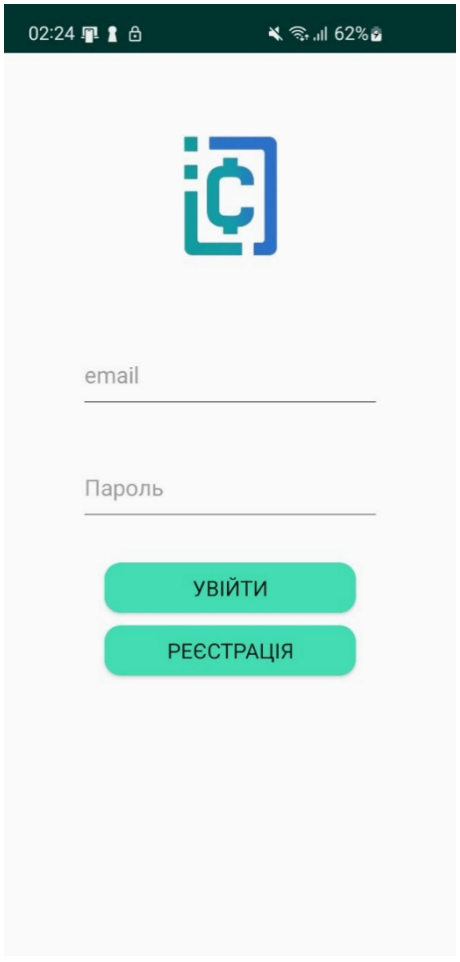

Рис. 4.1 – Авторизація у систему

Після авторизації користувач потрапляє у форму головного меню, що зображено на рисунку 4.2. На ньому користувач має багато елементів керування, таких як: створення фінансового плану, перегляд фінансового плану(за

наявності), внесення доходів, внесення витрат, перехід в меню скарбничок, перегляд загальної кількості коштів у скарбничках(за наявності), також знизу розташоване меню у вигляді горизонтальної панелі керування.

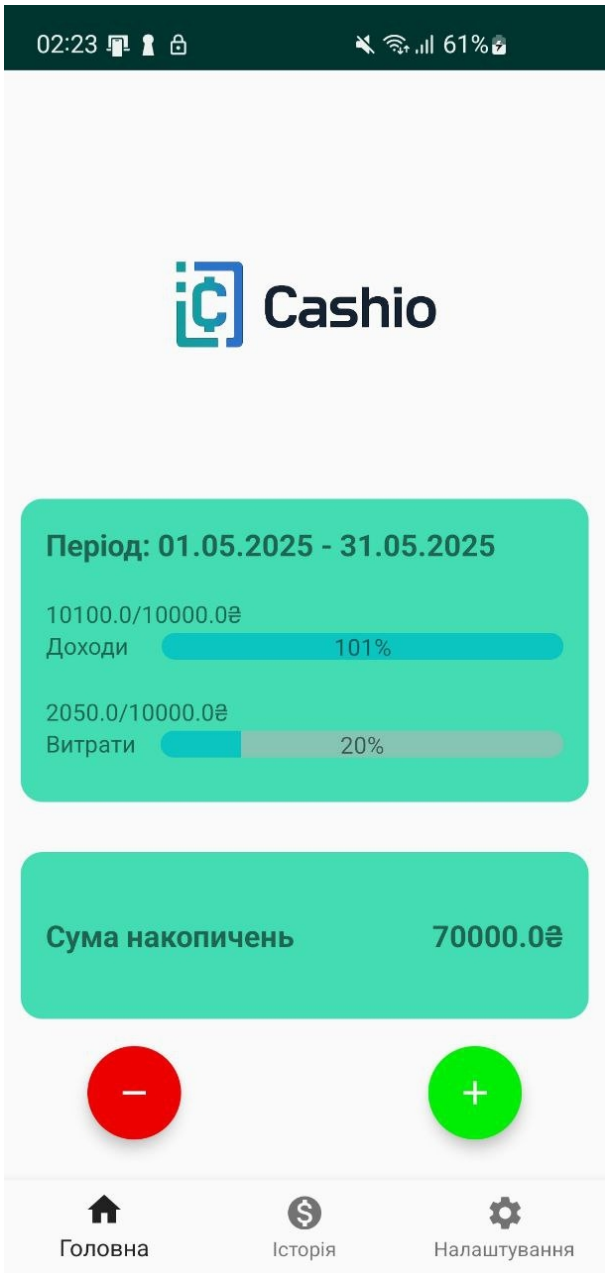

Рис. 4.2 – Інтерфейс головного меню

Перейдемо у вкладку «Історія» для перегляду внесених доходів та витрат. Інтерфейс вікна зображений на рисунку 4.3. Тут ми бачимо історію доходів та витрат у вигляді таблиці, а також можливість фільтрації історії за різними параметрами.

Рис.4.3 – Перегляд інформації про витрати і доходи

Перейдемо до вікна списку скарбничок на рис.4.4. На ньому ми бачимо список уже наявних скарбничок, а також можливість створення нової скарбнички.

Рис.4.4- Інтерфейс накопичень

При натисканні на елемент зі списку ми переходимо до інтерфейсу керування скарбничкою, що зображений на рисунку 4.5

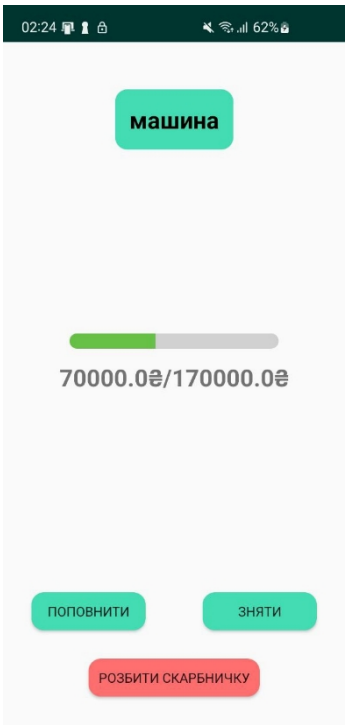

Рис.4.5 – Інтерфейс керування скарбничкою

На цій формі ми маємо такі функції: поповнення поточної скарбнички, зняття коштів зі скарбнички, а також кнопка розбиття скарбнички.

Також, на головному вікні є можливість детального перегляду прогресу фінансового плану. Дана форма зображена на рисунку 4.6.

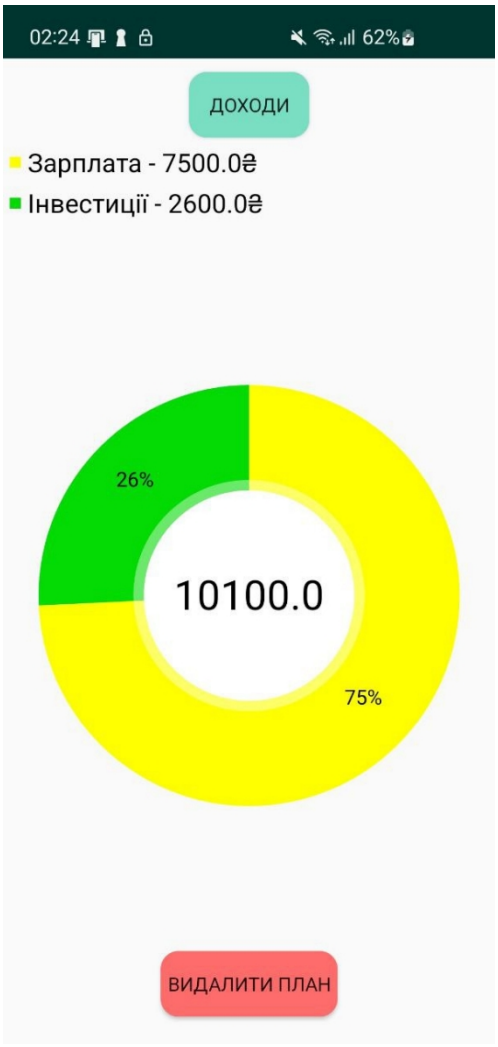

Рис.4.6 – Форма фінансового плану

Тут ми бачимо: зверху toggle-кнопка для перемикання між доходами та витратами, діаграма розподілу фінансів, легенда з подробицями діаграми, можливість видалення фінансового плану.

Також потрібно виділити увагу вікну особистого кабінету, що зображений на рисунку 4.7.

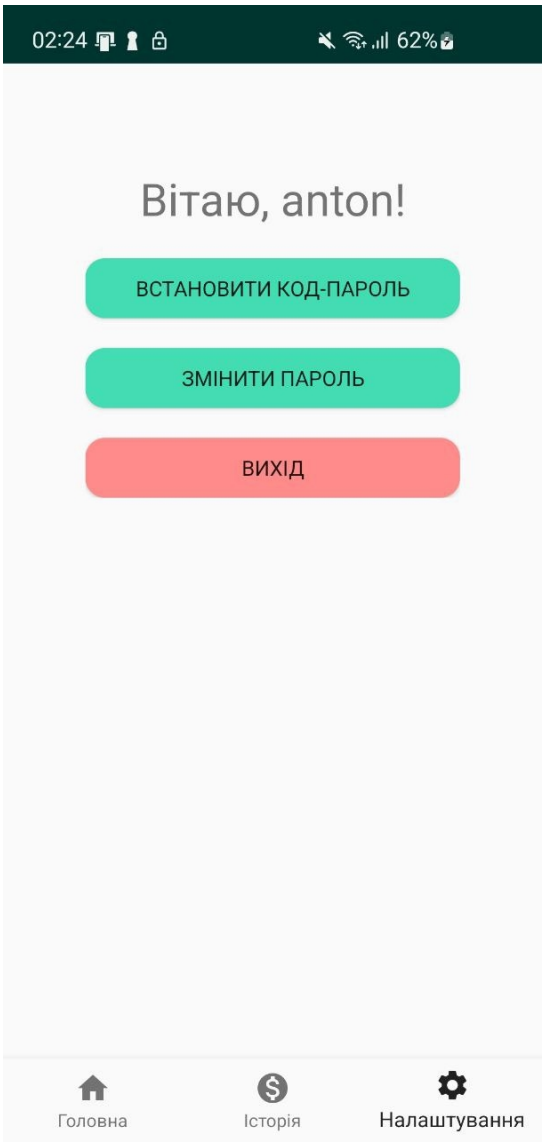

Рис.4.6 – Форма керування обліковим записом

Отже, в результаті було розглянуто основні форми інтерфейсу розроблюваного ПЗ.

4.1 Вимоги до апаратного та програмного забезпечення

Діаграма розміщення, що зображена на рисунку 4.8, відображає фізичне розташування програмних компонентів у системі, зокрема, як програмне забезпечення (вузли, компоненти) розміщується на апаратних засобах (сервери, пристрої) та які між ними зв'язки.

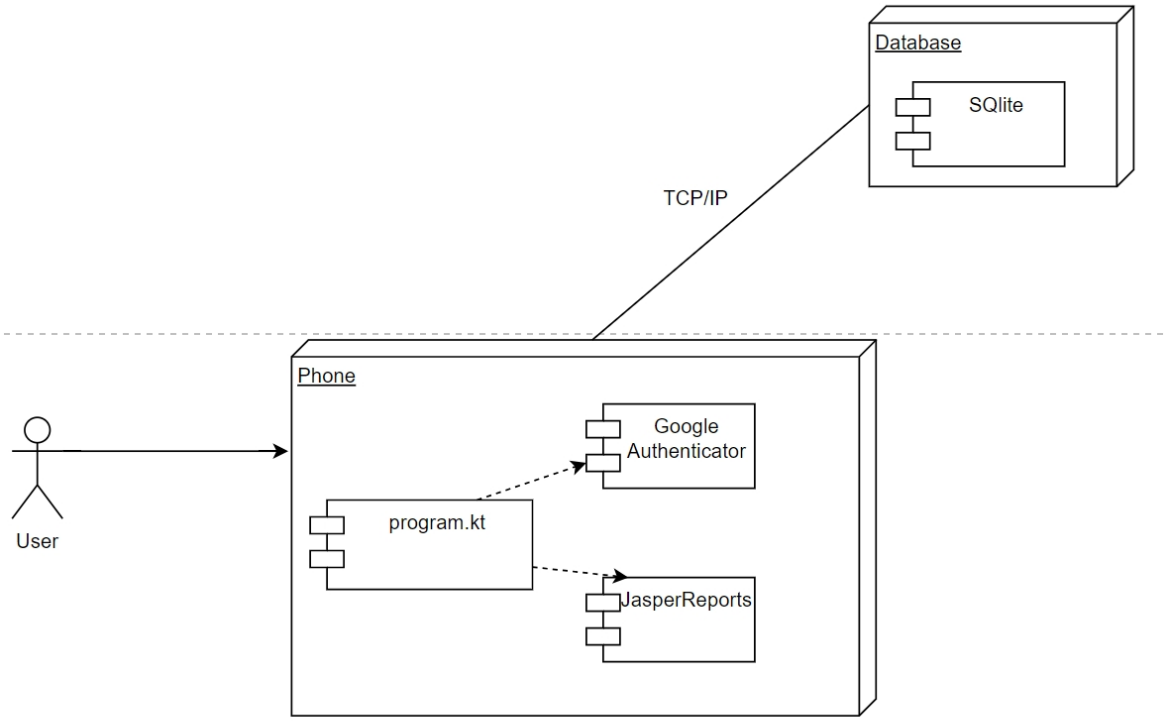

Рис.4.8 – Діаграма розміщення

Щодо вимог до розробленого ПЗ:

Мінімальні вимоги:

- Операційна система: Android 8.0 (Oreo)
- Процесор: 4-ядерний, 1.4 ГГц
- Оперативна пам’ять: 2 ГБ
- Вільне місце на пристрої: 100 МБ
- Роздільна здатність екрану: 720×1280 пікселів
- Інтернет-з’єднання: періодичне, для резервного копіювання та оновлень

- Дозволи: доступ до внутрішнього сховища, доступ до мережі, доступ до облікових записів (для автентифікації)

Рекомендовані вимоги:

- Операційна система: Android 11 або новіша
- Процесор: 8-ядерний, 2.0 ГГц і вище
- Оперативна пам’ять: 4 ГБ і більше
- Вільне місце на пристрої: 250 МБ
- Роздільна здатність екрану: Full HD (1080×1920) або вище
- Інтернет-з’єднання: стабільне для синхронізації з хмарними сервісами
- Додаткові функції: підтримка біометричної автентифікації (відбиток пальця, Face ID)

4.2 Склад інсталяційного пакету

Інсталяційний пакет розроблюваного мобільного застосунку для Android представлений у форматі APK (Android Package), який містить усі необхідні компоненти для коректної установки і запуску програми на пристроях користувачів. Цей формат забезпечує зручність розповсюдження, а також підтримку автоматичної інсталяції через системний менеджер пакетів Android. У складі APK-файлу зберігаються як вихідні двійкові ресурси програми, так і додаткові метадані, необхідні для роботи системи.

Основу пакета становить компільований код застосунку, представлений у вигляді DEX-файлів (Dalvik Executable), які містять байт-код для виконання у віртуальній машині Android Runtime. Усі Kotlin-файли, зокрема program.kt, authentification.kt, accounting_user.kt, трансформуються у відповідні DEX-структури, забезпечуючи роботу логіки додатку. Окрім основного коду, до складу інсталяційного пакета входять скомпільовані ресурси користувацького інтерфейсу, створені за допомогою XML-файлів.

Крім коду та інтерфейсу, APK-файл включає каталог res/ із графічними ресурсами, такими як іконки, кольорові схеми та стилі, що забезпечує адаптивність дизайну до різних розмірів екранів і версій Android. Окремо в каталозі assets/ можуть зберігатися локальні файли бази даних, наприклад, шаблон файлу user_database.sql, який ініціалізується під час першого запуску програми. Також до складу входять зовнішні бібліотеки — як-от Google Authenticator і JasperReports — у вигляді JAR/AAR-файлів.

Також, важливим елементом пакету є файл AndroidManifest.xml, який визначає основні параметри застосунку: його назву, іконку, дозволи, необхідні компоненти (активності, сервіси, провайдери), а також мінімальні версії ОС Android. Завдяки структурованому складу інсталяційний пакет забезпечує не лише функціональність і стабільність роботи програми, а й її безпечне розгортання на широкому спектрі мобільних пристроїв.

ВИСНОВКИ

У результаті проведеного дослідження було розроблено мобільне програмне забезпечення для ведення особистих фінансів, яке дозволяє користувачам зручно планувати доходи й витрати, формувати фінансові звіти та створювати накопичувальні цілі (скарбнички). У процесі розробки були використані сучасні інструменти — мова програмування Kotlin, Android Studio для створення інтерфейсу на XML, а також вбудована база даних SQLite для локального зберігання інформації.

Було спроектовано та реалізовано логіку роботи з користувачами, фінансовими планами, банком, звітами й базою даних. Створено відповідні діаграми UML — класів, кооперацій, компонентів, розміщення, які відображають архітектуру системи та взаємодію між її складовими.

Таким чином, отримано повнофункціональний, інтуїтивно зрозумілий застосунок, який може бути використаний кінцевими користувачами для щоденного фінансового планування. Реалізовані функції й структура програмного забезпечення дозволяють легко масштабувати проект та адаптувати його під нові вимоги.

Варто зазначити, що розроблений продукт має неабиякий потенціал та попит у різних сферах людського життя, дозволяючи полегшити рутинні справи та обов’язки, тим самим полегшуючи життя майбутніх користувачів.

СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Lea, D. Kotlin Programming: The Big Nerd Ranch Guide. Big Nerd Ranch Guides, 2018 [Електронний ресурс]. – Режим доступу: https://developer.android.com/kotlin – Назва з екрана.
2. XUnit.net. Документація щодо модульного тестування з використанням xUnit [Електронний ресурс]. – Режим доступу: https://xunit.net/ – Назва з екрана.
3. Mednieks, Z., Dornin, L., Meike, G., Nakamura, M. Programming Android. O'Reilly Media, 2012 [Електронний ресурс]. – Режим доступу: https://developer.android.com/guide – Назва з екрана.
4. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. – 533 с.
5. Togelius, J., Yannakakis, G. N. Procedural Content Generation in Games. Springer, 2016 [Електронний ресурс]. – Режим доступу: https://link.springer.com/book/10.1007/978-3-319-42716-4 – Назва з екрана.
6. Hendrikx, M., Meijer, S., Van Der Velden, J. Procedural Content Generation: A Literature Review [Електронний ресурс]. – Режим доступу: https://ieeexplore.ieee.org/document/6151041 – Назва з екрана.
7. Maze Generation Algorithms. Jamis Buck [Електронний ресурс]. – Режим доступу: https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap – Назва з екрана.
8. Stanford University. Maze Generation Algorithms [Електронний ресурс]. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2001-02/recursive-backtracking/mazes.html – Назва з екрана.
9. Smith, G., Whitehead, J. Procedural Generation: An Overview and Future Directions [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1231234.1231235 – Назва з екрана.

ДОДАТОК А

Сторінок – 1

Заповнення даних про користувачів у класі

```
            val createMoneyBoxTableQuery = """
        CREATE TABLE IF NOT EXISTS moneybox (
            id INTEGER PRIMARY KEY,
            name text,
            user_id INTEGER,
            current_value REAL,
            max_value REAL,
            FOREIGN KEY (user_id) REFERENCES users(id)
        )
    """.trimIndent()

        val createPlanTableQuery = """
    CREATE TABLE IF NOT EXISTS financeplan (
        id INTEGER PRIMARY KEY,
        user_id INTEGER,
        start_date TEXT,
        end_date TEXT,
        value_cost REAL,
        value_yield REAL,
       FOREIGN KEY (user_id) REFERENCES users(id)
    )
"".trimIndent()

        val insertCategory = """
            INSERT INTO category (id, name)
            VALUES
            ('1','Одяг'),
            ('2','Продукти та ресторани'),
            ('3','Комунальні платежі'),
            ('4','Розваги'),
            ('5','Подорож'),
            ('6','Освіта'),
            ('7','Здоров''я'),
            ('8','Житло'),
            ('9','Інвестиції'),
            ('10','Інше'),
            ('11','Зарплата'),
            ('12','Збереження'),
            ('13','Інвестиції'),
            ('14','Подарунки'),
            ('15','Інші доходи');
        """.trimIndent()
```

ДОДАТОК Б

Сторінок – 2

Фрагменти коду розроблюваного ПЗ

```
fun insertFinancePlanData(startDate: String, endDate: String,
valueCost: Double, valueYield: Double) {
    // Отримання бази даних для запису
    val db = this.writableDatabase

    // Отримання user_id зі сеансу (наприклад, через метод
getSessionUserId)
    val userId = getSessionUserId()

    // Підготовка значень для вставки
    val values = ContentValues().apply {
        put("start_date", startDate)
        put("end_date", endDate)
        put("value_cost", valueCost)
        put("value_yield", valueYield)
        put("user_id", userId)
    }

    // Вставка даних в таблицю
    db.insert("financeplan", null, values)

    // Закриття з'єднання з базою даних
    db.close()
}
```

```
fun deleteMoneyBox(moneyBoxId: Int): Boolean {
    val db = this.writableDatabase
    val deletedRows = db.delete( table: "moneybox",  whereClause: "id = ?", arrayOf(moneyBoxId.toString()))
    db.close()
    return deletedRows > 0 // Повертаємо true, якщо була видалена хоча б одна рядок
}
```

```
fun insertYield(type: String, value: Double, name_category: String, date: String) {
    val db = this.writableDatabase
    val values = ContentValues()

    val user_id = this.getSessionUserId()

    val id_category = this.getCategoryIdByName(name_category)

    values.put("user_id", user_id)
    values.put("value", value)
    values.put("id_category", id_category)
    values.put("date", date)

    db.insert(type,  nullColumnHack: null, values)
    db.close()
}
```

```
fun setCode(newCode: Int) {
    val db = this.writableDatabase
    val cursor = db.query( table: "lock", arrayOf("code"), selection: null, selectionArgs: null, groupBy: null, having: null, orderBy: null)
    val exists = cursor.moveToFirst()
    cursor.close()

    val contentValues = ContentValues().apply {
        put("code", newCode)
    }

    if (exists) {
        db.update( table: "lock", contentValues, whereClause: null, whereArgs: null)
    } else {
        db.insertWithOnConflict( table: "lock", nullColumnHack: null, contentValues, SQLiteDatabase.CONFLICT_REPLACE)
    }

    db.close()
} // встановлення код-паролю
```

```
fun getUser(email: String, password: String): Boolean{
    val hashedPass = sha256(password)
    val db = this.readableDatabase
    val result = db.rawQuery( sql: "SELECT * FROM users WHERE email = ? AND pass = ?", arrayOf(email,hashedPass))
    val success = result.moveToFirst()
    result.close()
    db.close()
    return success
} // пошук користувача в БД при спробі входу-
```

```
fun addToCurrentValue(moneyBoxId: Int, valueToAdd: Double): Boolean {
    val db = this.writableDatabase

    val query = "SELECT current_value FROM moneybox WHERE id = ?"
    val cursor = db.rawQuery(query, arrayOf(moneyBoxId.toString()))

    if (cursor.moveToFirst()) {
        val currentValueIndex = cursor.getColumnIndex("current_value")
        val currentValue = cursor.getDouble(currentValueIndex)

        if (currentValueIndex >= 0 && currentValue >= 0) {
            val updatedValue = currentValue + valueToAdd

            if (updatedValue >= 0) { // Перевіряємо, чи нове значення не менше 0
                val contentValues = ContentValues().apply {
                    put("current_value", updatedValue)
                }
                val updatedRows = db.update( table: "moneybox", contentValues, whereClause: "id = ?", arrayOf(moneyBoxId.toString()))

                db.close()
                return updatedRows > 0 // Повертаємо true, якщо була оновлена хоча б одна рядок
            } else {
                Toast.makeText(context, text: "Сума банки не може бути від'ємною", Toast.LENGTH_SHORT).show()
            }
        }
    }

    cursor.close()
    db.close()
    return false // Запис з вказаним id не знайдено або current_value містить недопустиме значення
}
```