**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ**
**Кафедра програмної інженерії**

# КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ЗАСТОСУНКУ З ПІДТРИМКОЮ АВТОМАТИЧНОГО ОБЛІКУ ТА РОЗПОДІЛУ ВИТРАТ ЗА АНАЛІЗОМ ЧЕКІВ ЗАСОБАМИ FLUTTER»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
М.В. Левданський
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Горбенко В.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Борю С.Ю.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ**
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

# З А В Д А Н Н Я

## НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Левданському Михайлу Владиславовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка застосунку з підтримкою автоматичного обліку та розподілу витрат за аналізом чеків засобами Flutter

керівник роботи Горбенко Віталій Іванович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 04.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік технологій.
3. Умови для створення проєкту.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Проєктування застосунку.
3. Створення застосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання 07.03.2025 р.

## К А Л Е Н Д А Р Н И Й  П Л А Н

| № | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | Розробка плану роботи. | 10.03.2025 | |
| | | | |
| 2. | Збір вихідних даних. | 12.03.2025 | |
| | | | |
| 3. | Обробка методичних та теоретичних | 20.03.2025 | |
| | джерел. | | |
| | | | |
| 4. | Розробка першого та другого розділу. | 01.05.2025 | |
| | | | |
| 5. | Розробка третього розділу. | 10.11.2025 | |
| | | | |
| 6. | Оформлення та нормоконтроль | 01.12.2025 | |
| | кваліфікаційної роботи магістра. | | |
| | | | |
| 7. | Захист кваліфікаційної роботи. | 18.12.2025 | |

Студент _____ М.В. Левданський
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.І. Горбенко
(підпис) (ініціали та прізвище)

## Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

# РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка застосунку з підтримкою автоматичного обліку та розподілу витрат за аналізом чеків засобами Flutter»: 63 с., 31 рис., 11 табл., 25 джерел, 3 додатка.

АНДРОІД, ВІДЖЕТ, ГРУПОВІ ВИТРАТИ, КРОСС-ПЛАТФОРМЕННІСТЬ, МОБІЛЬНИЙ ЗАСТОСУНОК, РОЗПОДІЛ ВИТРАТ, РОЗРОБКА, ФЛАТТЕР, GOOGLE ML KIT OCR, MATERIAL DESIGN 3, REALM.

Об’єкт дослідження – розробка програм та додатків з використанням фреймворку flutter.

Мета роботи: розробити додаток за обраною темою, порівняти використані засоби розробки з засобами для нативної розробки для Android.

Методи дослідження – порівняльний аналіз, застосування на практиці, тестування.

Метою даної роботи є розробка flutter-застосунку який полегшує розрахунки витрат, наприклад: група людей пішла у магазин, купили багато товарів, а заплатив за них хтось один, і, для того щоб тому хто платив, не розраховувати усе самотужки, він або вона може додати позиції з чеку, додати людей які брали участь, розподілити людей по позиціях у чеку, і застосунок розрахує хто і скільки повинен грошей тому хто платив.

Під час написання цієї дипломної роботи було розглянуто розробку мобільного застосунку на основі Flutter із підтримкою автоматичного розподілу витрат за аналізом чеків, обгрунтовано актуальність створення інструменту для спрощення розрахунків у групових покупках, проаналізовано сучасні підходи до обробки даних чеків (OCR Google ML Kit, API-інтеграція) та описано стек технологій: Flutter, Material Design 3, Realm, показано результат проєктування: UML-діаграма прецедентів, ER-діаграма бази даних та опис архітектури.

# SUMMARY

Master’s qualifying paper “Development of an Application with Support for Automatic Accounting and Separating of Spends Based on the Analysis of Checks Using Flutter”: 63 pages, 31 figures, 11 tables, 25 references, 3 supplements.

ANDROID, COST ALLOCATION, CROSS-PLATFORM, DEVELOPMENT, FLUTTER, GOOGLE ML KIT OCR, GROUP EXPENSES, MATERIAL DESIGN 3, MOBILE APPLICATION, PAYMENT DISTRIBUTION, REALM, WIDGET.

The object of the study is development of programs and applications using the flutter framework.

The aim of the study is develop an application on a chosen topic, compare the development tools used with tools for native development for Android.

The methods of research are comparative analysis, practical use, testing.

The purpose of this work is to develop a flutter application that facilitates cost calculations, for example: a group of people went to the store, bought a lot of goods, and someone paid for them alone, and in order for the person who paid not to calculate everything on their own, he or she can add items from the check, add people who participated, distribute people by items in the check, and the application will calculate who owes money to the person who paid and how much.

While writing this thesis, the development of a mobile application based on Flutter with support for automatic cost distribution based on check analysis was considered, the relevance of creating a tool to simplify calculations in group purchases was substantiated, modern approaches to processing check data were analyzed (OCR Google ML Kit, API integration) and the technology stack was described: Flutter, Material Design 3, Realm, the design result is shown: UML-use case diagram, ER-database diagram and architecture description.

**ЗМІСТ**

Завдання на кваліфікаційну роботу ..... 2
Реферат ..... 4
Summary ..... 5
Вступ ..... 8
1 Аналіз предметної області та огляд існуючих рішень ..... 10
1.1 Загальна характеристика предметної області ..... 10
1.2 Огляд мобільних і вебзастосунків для обліку групових витрат ..... 11
1.3 Аналіз алгоритмів обробки чеків (OCR та API) ..... 13
1.4 Порівняльний аналіз технологічних стеків ..... 15
1.5 Обґрунтування обраного стеку технологій ..... 16
2 Проєктування застосунку ..... 17
2.1 UML-діаграма прецедентів ..... 17
2.2 ER-діаграма бази даних ..... 18
2.3 Архітектура застосунку та компоненти ..... 20
2.4 Алгоритми та методи, що використовуються у застосунку ..... 22
3 Реалізація програмного застосунку ..... 24
3.1 Створення структури проєкту ..... 24
3.2 Створення навігації ..... 25
3.3 Створення архітектури бази даних ..... 29
3.4 Презентація роботи додатку ..... 30
3.5 Порівняльний аналіз кросплатформенного підходу Flutter (Dart) та нативної розробки Android (Kotlin) ..... 33
3.5.1 Мета експериментального дослідження ..... 33
3.5.2 Методика проведення експерименту ..... 33
3.5.3 Зручність мови програмування та обсяг коду ..... 34
3.5.4 Продуктивність UI ..... 36
3.5.5 ANOVA та статистична оцінка ефективності Flutter та Kotlin 36

3.5.6 Порівняння продуктивності Realm ..... 40
3.5.7 Продуктивність навігації ..... 41
3.5.8 Споживання ресурсів ..... 41
3.5.9 Архітектурна гнучкість ..... 42
3.5.10 Підсумковий науковий висновок ..... 43
Висновки ..... 44
Перелік посилань ..... 46
Додаток А Переваги ORM баз даних над SQL базами даних ..... 49
Додаток Б Реалізація схеми Navgraph ..... 53
Додаток В Реалізація схеми локальної бази даних ..... 55

# ВСТУП

У сучасному світі мобільні рішення для спільного обліку витрат набувають дедалі більшої популярності через зростання кількості колективних покупок та потребу в прозорості фінансових розрахунків. Незручність ручного підрахунку витрат, ризик помилок та час, що витрачається на розподіл платежів, обумовлюють актуальність розробки спеціалізованого інструменту для автоматизації цих процесів.

Новизна роботи полягає у поєднанні сучасних технологій розпізнавання тексту (OCR Google ML Kit) та інтеграції з API торговельних точок для автоматичного отримання даних з чеків, що в поєднанні зі стеком Flutter та Realm забезпечує високу швидкодію, кросплатформеність та зручність користування. Практична значущість розробки полягає у скороченні часу на фінансові розрахунки у групових покупках, зменшенню ймовірності помилок та підвищенні задоволеності користувачів.

Метою роботи є створення мобільного застосунку на Flutter із підтримкою автоматичного обліку та розподілу витрат за аналізом чеків, для досягнення якої, були визначені такі завдання:

- проаналізувати предметну область і вивчити існуючі програмні рішення для автоматизації обліку групових витрат;
- обґрунтувати вибір технологічного стеку: Flutter, Material Design 3, Realm, та методів обробки чеків (OCR та API);
- спроєктувати архітектуру застосунку, включаючи uml-діаграму прецедентів та er-діаграму бази даних;
- розробити алгоритми розпізнавання та розподілу витрат між учасниками;
- реалізувати первинну структуру проєкту та підготувати середовище для подальшої розробки.

Об’єктом дослідження є процес автоматизації обліку групових витрат за допомогою мобільного застосунку, а предметом дослідження виступають методи обробки та аналізу чеків, алгоритми розподілу витрат та архітектурні рішення для реалізації відповідного функціоналу у за допомогою Flutter.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

## 1.1 Загальна характеристика предметної області

У сучасному світі мобільні застосунки для спільного обліку витрат стають невід'ємною частиною повсякденного життя, найпоширеніший приклад – спільний похід у магазин за продуктами. Предметна область, обрана під час написання цієї роботи, охоплює сукупність процесів та технологій, пов'язаних із реєстрацією, обробкою та аналізом фінансових даних у контексті групових платежів. Основна мета таких систем полягає у зменшенні навантаження на учасників, виключенні ручного введення даних та мінімізації ризику помилок при розподілі витрат. До ключових характеристик предметної області можна віднести:

- колективні платежі: сценарії, в яких одна особа (ініціатор) сплачує за товари або послуги від імені групи – після здійснення платежу необхідно розподілити суму між учасниками залежно від їхньої частки споживання;
- автоматизація збору даних: замість ручного введення чеків та позицій застосунок повинен надавати можливість імпорту даних зображенням чека з автопізнаванням тексту (OCR) або через API інтеграцію з торговими точками;
- гнучкість налаштувань: кожне замовлення може мати різну кількість учасників та асортимент позицій; застосунок повинен підтримувати індивідуальний вибір товарів для кожного учасника;
- алгоритми розподілу: на основі даних про участь кожного користувача в конкретних позиціях чека розраховується сума, що підлягає компенсації ініціатору.

Ключовими вимогами до подібних систем є:

- точність обробки чеків: висока достовірність розпізнавання тексту та

сум на чеку – неправильне розпізнавання призводить до помилкового розподілу витрат;

– простота інтерфейсу: інтуїтивно зрозумілий UI/UX, що дозволяє користувачам без додаткового навчання додавати чек, обирати товари та призначати учасників;
– надійність збереження даних: використання надійної бази даних (Realm) для зберігання інформації про транзакції, користувачів і параметри розподілу;
– кросплатформеність: забезпечення однакового функціоналу та UX на різних операційних системах (Android, iOS) завдяки використанню Flutter.

Предметна область також охоплює питання безпеки та конфіденційності фінансових даних. Доцільно реалізувати механізми автентифікації та шифрування локальної бази даних, а також використовувати захищені канали зв’язку при API-інтеграції з торговими сервісами.

Таким чином, предметна область об’єднує технологічні, алгоритмічні та UX-фактори, що визначають функціональність додатків для автоматизації обліку та розподілу групових витрат.

## 1.2 Огляд мобільних і вебзастосунків для обліку групових витрат

На момент проведення дослідження ринок містить як мобільні, так і веборієнтовані сервіси для обліку спільних витрат. Нижче наведено огляд найбільш відомих рішень із зазначенням їхніх ключових можливостей, переваг і обмежень:

a) Splitwise [1] (iOS, Android, Web):
    1) основний функціонал: створення «груп», додавання витрат із вказанням суми й учасників, автоматичний розподіл та узгодження балансів;

2) переваги: інтуїтивний інтерфейс, підтримка валют, історія транзакцій;
3) обмеження: відсутність OCR-функції для обробки чеків; без API-інтеграцій із торговими сервісами;

б) Settle Up [2] (iOS, Android):
1) основний функціонал схожий на Splitwise із можливістю ведення декількох групних бюджетів, синхронізацією через облако;
2) переваги: офлайн-режим, мультивалютність, підтримка погашення боргів через PayPal;
3) обмеження: немає автоматичного імпорту з чеків; UI менш адаптований до Material Design;

в) Tricount [3] (iOS, Android, Web):
1) надає можливість імпорту чеків у вигляді фотографій із базовою OCR-підтримкою, статистику за категоріями витрат;
2) переваги: вбудована фотографічна обробка чеків, простий звітний інтерфейс;
3) обмеження: OCR доступний лише у преміум-версії; відсутність API-інтеграції;

г) Splittr (iOS, Android):
1) фокус на зручності планування спільних поїздок: крім обліку витрат, дозволяє створювати маршрути й розклад;
2) переваги: гнучкі опції для подорожей, візуалізація балансу групи;
3) обмеження: відсутність OCR та API; орієнтація на туристичні сценарії.

Аналіз показує, що більшість існуючих рішень пропонують зручний інтерфейс та базові механізми розподілу, проте здебільшого обмежуються ручним введенням сум або спрощеною OCR-функцією. Жоден із розглянутих застосунків не поєднує повноцінну і безкоштовну OCR-обробку на стороні

клієнта (Google ML Kit OCR), можливості інтеграції з API торгових мереж та сучасний стек кросплатформеної розробки з Realm [4].

Таким чином, існує потреба в комплексному рішенні, яке об’єднує:

– високоточне розпізнавання чеків за допомогою OCR Google ML Kit [5];
– інтеграцію із зовнішніми API магазинів для автоматизованого імпорту даних;
– уніфікований кросплатформений інтерфейс на основі Flutter та Material Design 3 [6];
– надійне сховище даних із Realm.

Підсумовуючи, можна з впевненістю сказати що запропонована розробка є доцільною і буде конкурентоспроможною на ринку мобільних застосунків для групового обліку витрат, поєднуючи найкращі практики OCR, API-інтеграцій і кросплатформених технологій.

### 1.3 Аналіз алгоритмів обробки чеків (OCR та API)

Обробка даних із чеків становить ключову частину функціональності застосунку. Існують два основні підходи: розпізнавання тексту із зображення (OCR) та опитування API торгових мереж для отримання структурованих даних (див. табл. 1.1).

**OCR** (Optical Character Recognition):

а) Google ML Kit OCR [5]: пропонує локальне (on-device) розпізнавання тексту, що дозволяє обробляти зображення чеків без необхідності відправляти їх на сервер; переваги:
   1) приватність: дані не покидають пристрій,
   2) offline-режим: обробка без інтернет-з’єднання,
   3) можливість налаштувати мовну модель (підтримка української, англійської тощо);

б) OpenCV та Tesseract: альтернативний open-source підхід, вимагає додаткового налаштування під формат чеків, може потребувати серверної обробки для підвищення точності.

Основні етапи OCR-процесу:

– попередня обробка зображення: корекція перспективи, бінаризація, фільтрація шумів;
– розпізнавання тексту: вибір області інтересу, розпізнавання символів;
– постобробка: виправлення помилок розпізнавання через словники та шаблони (наприклад, ціни).

**API-інтеграція з торговими мережами**:

а) деякі великі торгові мережі надають API для отримання даних чека за номером чи QR-кодом (наприклад, українська мережа АТБ через API “VeriCheck” [7]); переваги:
   1) структуровані дані без помилок OCR,
   2) швидкість отримання: дані надходять миттєво;

б) недоліки:
   1) не всі мережі мають відкриті API,
   2) необхідність авторизації та управління ключами доступу,
   3) питання безпеки передачі даних.

Таблиця 1.1 – Порівняльна характеристика OCR та API

| **Характеристика** | **OCR (Google ML Kit)** | **API-інтеграція** |
|---|---|---|
| Обробка без мережі | Так | Ні |
| Конфіденційність даних | Висока (локальна обробка) | Залежить від захищеності API |
| Точність розпізнавання | До 95% (залежить від якості зображення) | До 99% (структуровані дані) |
| Обмеження за джерелом даних | Будь-яке зображення чека | Лише мережі з відкритим API |
| Витрати на впровадження | Мінімальні (SDK) | Залежить від умов API (часто платні) |

Отже, застосунок повинен підтримувати обидва підходи: OCR для роботи з будь-якими чеками та API-інтеграцію для підвищення точності та швидкості обробки в мережах-партнерах.

### 1.4 Порівняльний аналіз технологічних стеків

Для реалізації застосунку розглянуто такі технологічні стеки (див. табл. 1.2).

Таблиця 1.2 – Порівняльний аналіз технологічних стеків [8-17]

| **Компонент** | **Flutter + Dart** | **React Native + JavaScript** | **Native (Kotlin і Swift)** |
|---|---|---|---|
| Кросплатформеність | Повна (Android, iOS, Web) | Повна (Android, iOS) | Окремий код для кожної платформи |
| Продуктивність | Висока (компіляція AOT) | Середня (JIT компіляція) | Відмінна |
| UI/UX | Material Design 3, Cupertino | Залежить від бібліотеки | Рідний інтерфейс |
| Розмір додатка | Середній (близько 6–7 МБ) | Високий (8–10 МБ) | Низький (4–5 МБ) |
| Спільнота та підтримка | Швидко зростає, сильна підтримка | Великі спільноти, багато пакетів | Обмежена для кожної мови |
| Вбудовані бібліотеки OCR/API | ML Kit підтримано плагінами | Додаткові мости, плагіни | Нативні SDK |
| ORM | Realm | TypeORM, Sequelize | Room (Android), CoreData (iOS) |

Висновок порівняння: Flutter із Dart забезпечує найкращий баланс між кросплатформеністю, продуктивністю та сучасними UI-підходами (Material Design 3) [6]. Використання Realm [4] гарантує зручну роботу з базою даних через генерацію типів та міграцій. Google ML Kit інтегрується безпосередньо як плагін Flutter, що спрощує налаштування OCR.

## 1.5 Обґрунтування обраного стеку технологій

На підставі проведеного аналізу обрано:

– Flutter (Dart): висока продуктивність завдяки Ahead-of-Time компіляції, спільна кодова база для Android, iOS та Web, велика спільнота;
– Material Design 3: сучасний набір рекомендацій для створення інтерфейсів, що забезпечують узгодженість UX та привабливий вигляд;
– Realm: автоматична генерація типів, підтримка міграцій, прозоре управління схемою бази даних (див. додаток А);
– Google ML Kit OCR: локальне on-device розпізнавання тексту, підтримка офлайн-режиму та високий рівень конфіденційності.

# 2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ

Для повнішого розуміння функціональної структури застосунку доцільно використовувати діаграму прецедентів (use case diagram) та ER-діаграму (entity-relationship diagram) для моделювання структури бази даних.

Ці діаграми дозволяють чітко візуалізувати як функціональну логіку застосунку, так і зв’язки між даними. Вони також є основою для майбутньої реалізації логіки бази даних та бізнес-правил.

На цьому етапі проєктного аналізу сформовано базове уявлення про архітектуру системи, методи обробки даних, ключові алгоритми та структуру зберігання інформації.

## 2.1 UML-діаграма прецедентів

Під час проєктування та створювання сценаріїв використання було виділено наступні випадки (див. рис. 2.1).

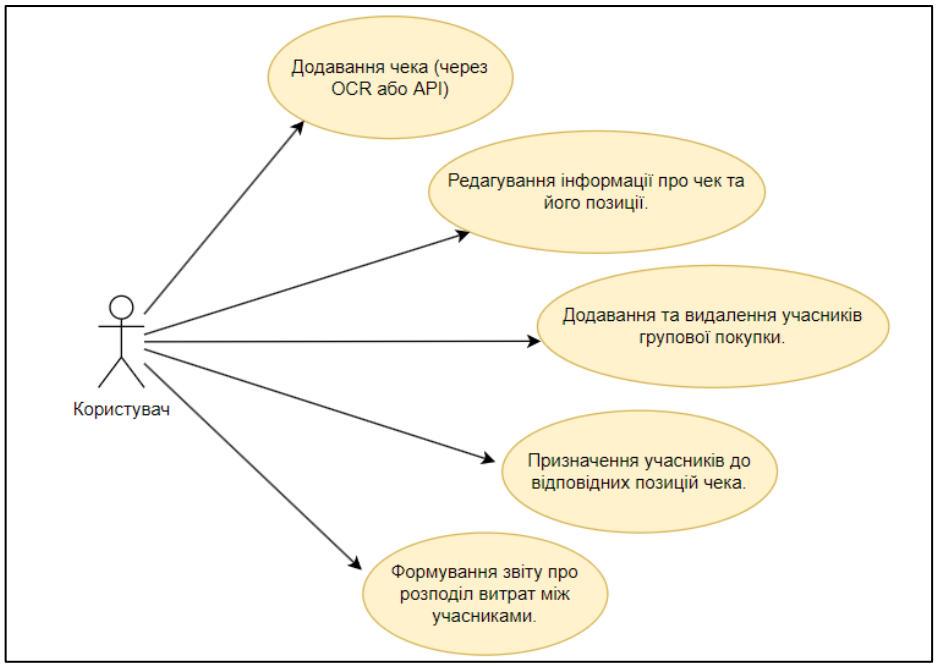



Рисунок 2.1 – Діаграма прецедентів

Ця діаграма вичерпно відображає яким чином буде використовуватись додаток користувачами, та які можливості будуть їм доступні. Оскільки додаток на даному етапі проєктування планується повністю оффлайновим, наразі у діаграмі присутній тільки один актор – «Користувач».

## 2.2 ER-діаграма бази даних

Для створення локальної бази даних було обрано використання Realm [4]. Realm – це мобільна та кросплатформна об’єктно-орієнтована база даних, розроблена як високопродуктивна та зручна для розробників альтернатива SQLite та Core Data. Вона має відкритий вихідний код та широко використовується для офлайн-додатків.

Ключові характеристики:

– об’єктно-орієнтована модель даних: на відміну від традиційних реляційних баз даних (таких як SQLite), які використовують таблиці та рядки, Realm зберігає дані як об’єкти, що усуває потребу в об’єктно-реляційному відображенні (ORM) та дозволяє розробникам працювати безпосередньо з об’єктами рідної мови;

– висока продуктивність: побудована з нуля на власному ядрі C++, Realm працює на спеціальному механізмі сховища, використовуючи методи відображення пам’яті (архітектура нульового копіювання) для швидкого доступу до даних та запитів;

– кросплатформна підтримка: пропонує SDK для різних платформ та мов, включаючи Java (Android), Kotlin, Swift (iOS/macOS), Objective-C, JavaScript/TypeScript (React Native, Node.js), C# (.NET/Xamarin) та Flutter/Dart;

– синхронізація в реальному часі: як частина лінійки продуктів MongoDB, Realm надає функції для автоматичної синхронізації даних між мобільними клієнтами та хмарою MongoDB, що дозволяє розробляти додатки в офлайн-режимі;

– простота використання: Realm пропонує прості, інтуїтивно зрозумілі API, що робить зберігання, запити та керування даними ефективнішими та вимагає менше шаблонного коду порівняно зі старими рішеннями;
– живі об’єкти: об’єкти в Realm є «живими», тобто вони автоматично оновлюються, коли змінюються базові дані, що робить його ідеальним для реактивних архітектур додатків.

При розробці додатку структура локальної бази даних буде мати такий вигляд, як представлено на рисунку 2.2.

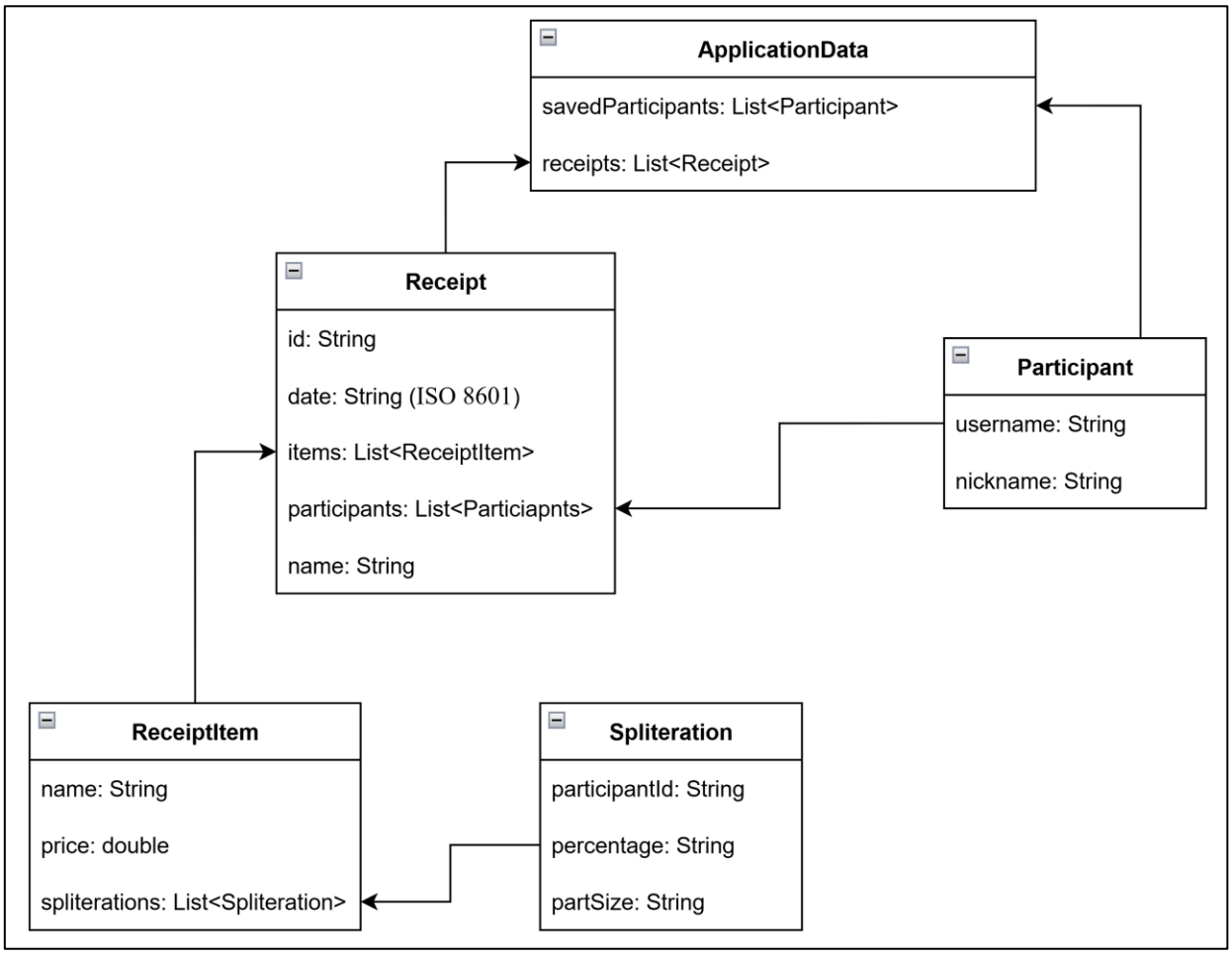



Рисунок 2.2 – Діаграма бази даних

Варто звернути увагу що структура відрізняється від зазвичай використовуваної, оскільки Realm дозволяє зберігати об’єкти в середині об’єктів, та навіть списки, що робить використання ідентифікаторів та

необхідність пошуку пов’язаних об’єктів зайвою. Також, для зберігання дат було обрано використання строк у форматі ISO 8601 [18], замість timestamp оскільки це дає змогу враховувати часові пояси.

## 2.3 Архітектура застосунку та компоненти

Архітектура застосунку побудована на основі шаблону проєктування CleanArchitecture [19] з урахуванням принципів модульності, масштабованості та підтримки багаторазового використання компонентів. У роботі було застосовано шаблон MVVM (Model-View-ViewModel) [20], який дозволяє ефективно відокремлювати бізнес-логіку від інтерфейсу користувача та сприяє покращенню тестованості коду (див. рис. 2.3).

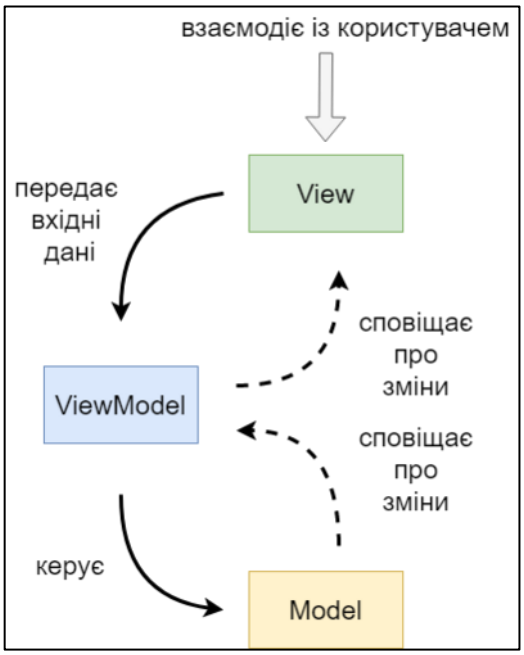



Рисунок 2.3 – Діаграма взаємодії між компонентами шаблону MVVM

Розглянемо детальніше компоненти системи.

*User Interface (UI)* – реалізований за допомогою Flutter з використанням Material Design 3 [6]:

– екрани входу/реєстрації;

– екран додавання чеку (з вибором методу: OCR, API, вручну);
– екран перегляду позицій чеку та додавання учасників;
– екран розподілу витрат;
– екран фінального звіту.

*ViewModel* – відповідає за обробку подій з UI, зв’язок із моделлю даних та формування стану інтерфейсу. Застосовано бібліотеки Provider, Riverpod або Bloc для керування станом (залежно від остаточного вибору під час реалізації).

*Model, Data Layer*:

– Realm [4] забезпечує взаємодію з базою даних, включно з міграціями, типобезпечною роботою з об’єктами, та автоматичним генеруванням схем;
– сховище даних підтримує локальну та/або хмарну синхронізацію, залежно від сценаріїв використання.

*Сервіси обробки даних*:

– OCR-сервіс (Google ML Kit [5]): розпізнавання тексту з зображення чеку;
– API-сервіс: отримання даних чека безпосередньо з онлайн-каси або супермаркету (за наявності інтеграції);
– розподіл витрат: алгоритм рівномірного або пропорційного розподілу позицій по учасниках.

*Реалізація безпеки та автентифікації*:

– Firebase Authentication [21] або Google OAuth2 [22] (або інша альтернатива): для безпечного входу користувачів;
– захист даних користувача та забезпечення приватності (шифрування локальних даних, шифрований зв’язок через HTTPS).

Ця архітектура дозволяє ефективно масштабувати систему, повторно використовувати логіку, оновлювати окремі компоненти без впливу на інші частини застосунку та підтримувати високий рівень продуктивності і стабільності.

## 2.4 Алгоритми та методи, що використовуються у застосунку

У процесі розробки застосунку передбачається використання певних алгоритмів та методів. Розглянемо їх.

**Алгоритм розподілу витрат між учасниками**.

*Вхідні дані*: список позицій у чеку, список учасників, які споживали кожну з позицій, загальна вартість позицій.

*Алгоритм*:

- для кожної позиції визначити кількість користувачів, яким вона належить;
- розділити ціну позиції на відповідну кількість користувачів;
- для кожного користувача накопичити загальну суму, яку він винен платнику.

*Приклад*. У магазин пішло три людини – Михайло, Єлизавета та Адель, купили макарони вартістю двісті гривень, пиво вартістю триста гривень та корм для котів вартістю сто гривень, заплатив за все Михайло двісті плюс триста плюс сто тобто шістьсот гривень, і тепер Михайло розраховує хто скільки повинен йому грошей якщо макарони потрібні були Михайлу та Єлизаветі, пиво потрібно усім, а корм тільки Адель, тобто виходить що за макарони Єлизавета повинна сто гривень бо двісті ділиться на двох, за пиво Єлизавета повинна Михайлу сто гривень і Адель також бо триста ділиться на трьох, а за корм Адель повинна Михайлу сто гривень бо він потрібен тільки Адель, отже виходить что Єлизавета повинна Михайлу сто за макарони і сто за пиво отже усього двісті гривень, а Адель повинна Михайлу сто за пиво і сто за корм отже також усього двісті гривень:

- макарони (200 грн) – 2 особи – по 100 грн;
- пиво (300 грн) – 3 особи – по 100 грн;
- корм (100 грн) – 1 особа – 100 грн;
- результат: Єлизавета винна 200 грн, Адель – 200 грн.

**OCR-обробка чека**:

– використовується Google ML Kit Text Recognition API;
– зображення чека обробляється локально на пристрої;
– результатом є список рядків тексту, з яких далі фільтруються позиції товарів;
– передбачено використання регулярних виразів для виявлення шаблонів (наприклад, назва товару та ціна).

**Попередня класифікація елементів чека**. На основі розпізнаного тексту виконується класифікація рядків:

– назва товару;
– ціна.

Для цього можуть бути використані прості евристичні правила або моделі машинного навчання (у майбутньому).

**Побудова фінального звіту**;

– візуалізація результату в UI: таблиця з позиціями, учасниками та боргами;
– можливість експорту звіту у формат PDF або текстовий документ (у майбутньому).

**Оптимізація роботи з базою даних**. Застосування Realm [4] дозволяє ефективно організувати запити до бази, кешування та lazy loading для великих обсягів даних.

Реалізація зазначених алгоритмів дозволяє досягти високої точності розрахунків, зручності користування застосунком та ефективного використання ресурсів мобільного пристрою.

# 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ

## 3.1 Створення структури проєкту

За основу базової структури проєкту було вирішено обрати структуру яка використовується у Android-розробці, тобто модуль app (див. рис. 3.1), у якому буде міститися усе необхідне для взаємодії з користувачем, тобто інтерфейс, а також нескладна логіка для внутрішніх взаємодій всередині інтерейсу; а також модуль core (див. рис. 3.2), у якому вже містяться моделі сутностей для бази даних, взаємодія з мережею (у майбутньому), а також складна логіка пов’язана с розрахунками.

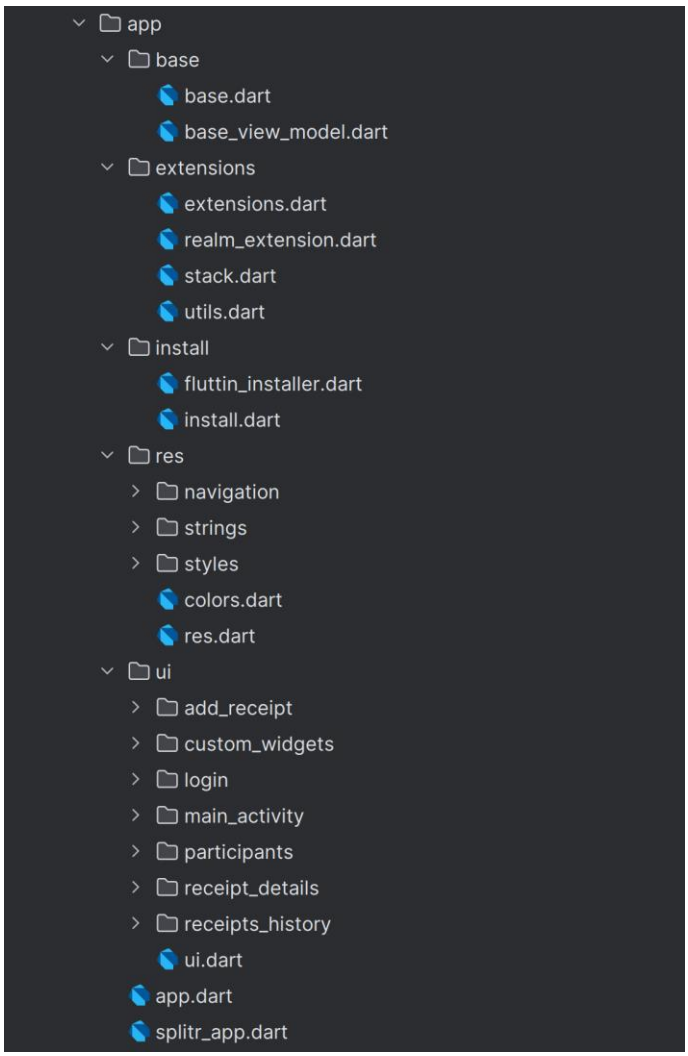



Рисунок 3.1 – Вміст модуля app

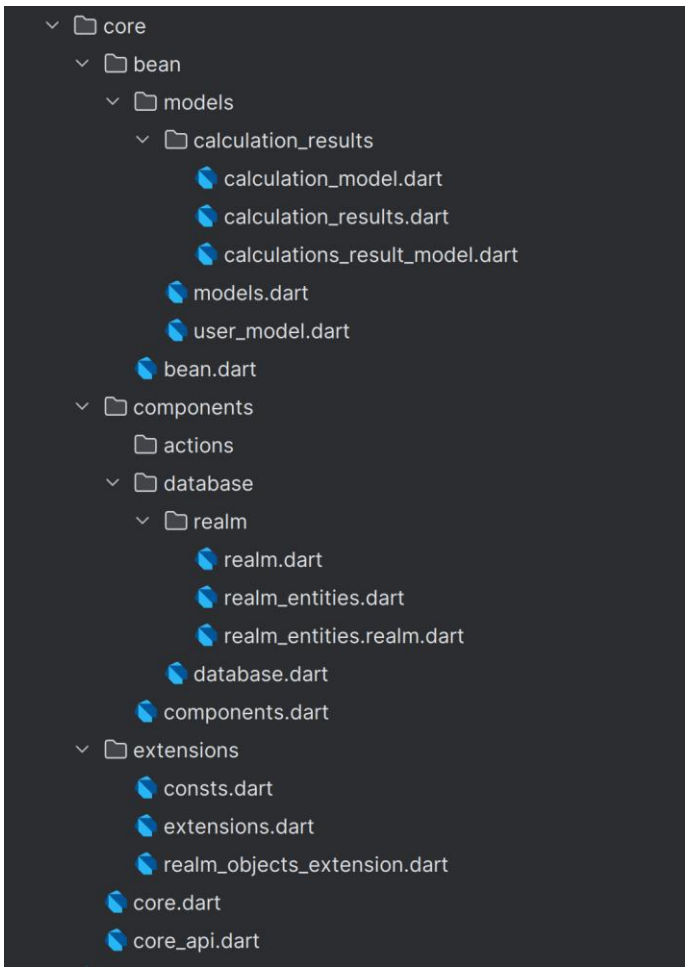



Рисунок 3.2 – Вміст модуля core

Обрана структура є сумішшю шаблону проєктування CleanArchitecture [20], і MVVM (Model View ViewModel) [21] – вона містить як елементи першого шаблону проєктування, так і другого, при чому структурний елемент ViewModel виступає у якості шару Interface Adapters, а також частково покриває необхідну логіку шару Application Business Rules. Модуль app виступає у якості шару Frameworks & Drivers, а core, відповідно, у якості шару Enterprise Business Rules (див. рис. 3.3).

### 3.2 Створення навігації

У цьому проєкті було вирішено відтворити механізм навігації який застосовується у Android-розробці, а саме – Jetpack Navigation Component [23].

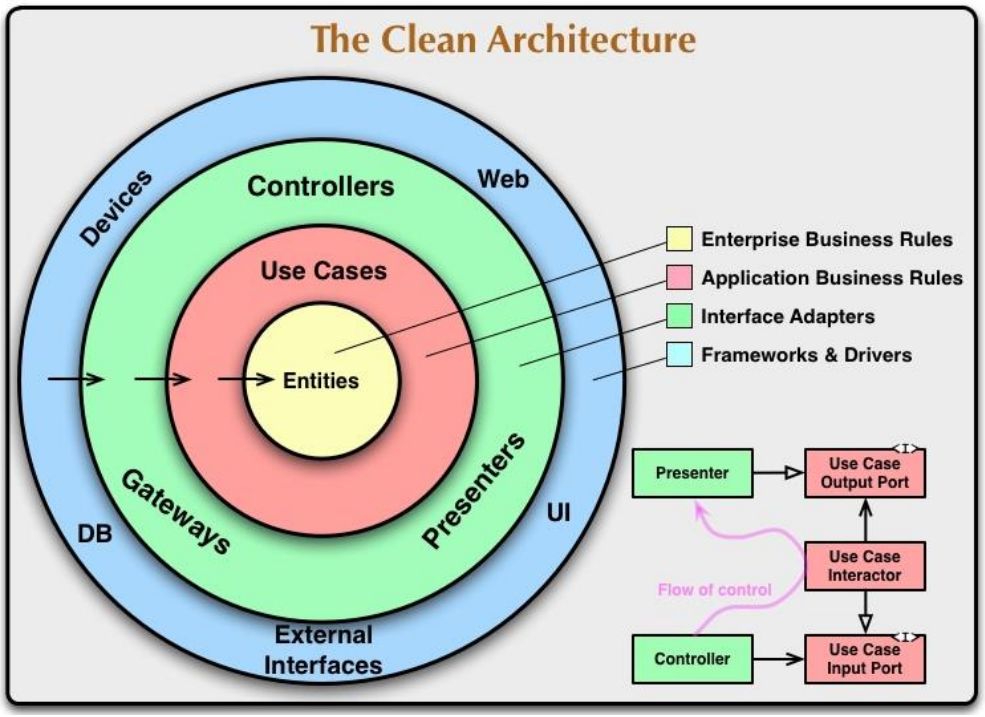



Рисунок 3.3 – Шаблон проєктування CleanArchitecture

Схема така: є граф (NavGraph), який містить усі кінцеві точки, або ж вузли (Nodes), і точки призначення (Destinations), і в залежності від того який саме вузол потрібно показувати, необхідно викликати метод контроллера навігації, передавши потрібну точку призначення у якості параметру (див. рис. 3.4 – 3.8).

```
/// ID - identifiers type
/// NT - Nodes type
/// NDT - destinations type
abstract class NavGraph<ID, NT extends NavNode, NDT extends NavDestination> {
  NT defaultNavNode();
  List<NT> getNodes();
  void setCurrentDestination(NDT destination);
  void navigateUp();
  NT getCurrentDestination();
}
```

Рисунок 3.4 – Схема NavGraph

```
abstract class NavNode<ID, W extends Widget> {
  ID getId();
  W getWidget();
  }
```

Рисунок 3.5 – Схема NavNode

```
interface class NavDestination<ID> {
  ID getId() {
    throw Exception("This class can only be extended or implemented from");
  }
  dynamic getNavArgs() {
    throw Exception("This class can only be extended or implemented from");
  }
  NavOptions? getNavOptions() {
    throw Exception("This class can only be extended or implemented from");
  }
}
```

Рисунок 3.6 – Схема NavDestination

```
class MainNavigationNode<W extends Widget> extends NavNode<MainDestinations, W> {
  MainDestinations id;
  W Function(NavArgs?) destinationProvider;
  W? instance;
  NavArgs? args;

  MainNavigationNode({
    this.args,
    required this.id,
    required this.destinationProvider,
  });

  void setArgs(NavArgs? args) {
    this.args = args;
  }

  @override
  MainDestinations getId() => id;

  @override
  W getWidget() {
    if(args == null) {
      instance ??= destinationProvider(args);
      args == null;
      return instance!;
    } else {
      W dest = destinationProvider(args);
      args = null;
      instance = dest;
      return dest;
    }
  }
}
```

Рисунок 3.7 – Реалізація NavNode

```
class MainNavigationDestination implements NavDestination<MainDestinations> {
  final MainDestinations id;
  final NavArgs? navArgs;

  const MainNavigationDestination({required this.id, required this.navArgs});

  @override
  MainDestinations getId() => id;

  @override
  getNavArgs() => navArgs;

  @override
  NavOptions? getNavOptions() => null;
}
```

Рисунок 3.8 – Реалізація NavDestination

Нажаль реалізація схеми NavGraph занадто об’ємна, тож її було винесено у окремий додаток (див. додаток Б).

Використовуючи створений компонент навігації було організовано наступну структуру маршрутизацій, як представлено на рисунку 3.9.

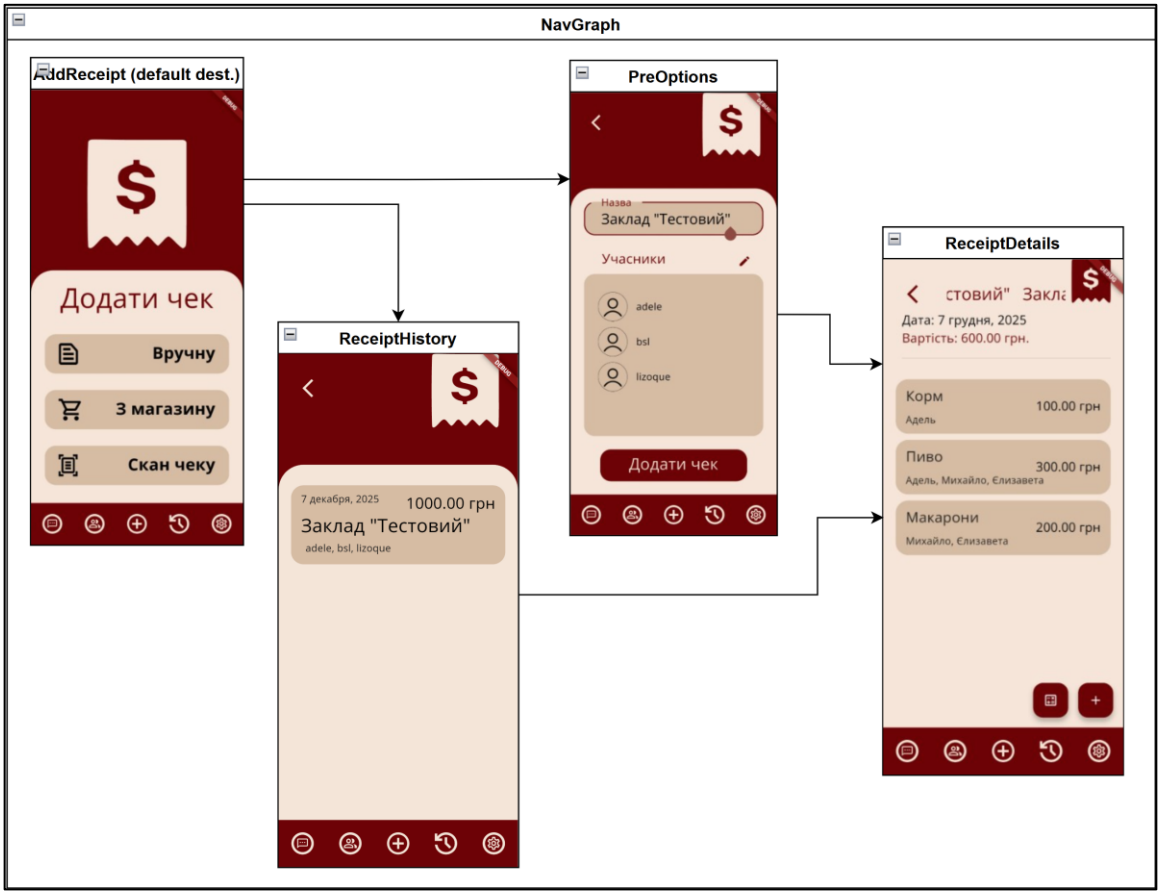



Рисунок 3.9 – Навігація у додатку Splitr

### 3.3 Створення архітектури бази даних

Як вже було зазначено раніше, для локального зберігання даних було обрано Realm [4], він дуже зручний тим що дозволяє працювати з вмістом таблиць як із звичайними об’єктами у рантаймі (див. розділ 2.2, с. 19), наприклад списками та колекціями екземплярами класів або ж навіть колекцій які містять екземпляри класів які у свою чергу можуть містити будь що із вищеперерахованого, тобто механізми які роблять query-запити для таблиць інкапсульовані «під капотом» цих самих об’єктів.

Отже, для схеми бази даних було використано такі класи, як представлено на рисунку 3.10.

```
@RealmModel()
class _ReceiptEntity {

 @PrimaryKey()
 late String id;
 late String name;
 late List<_ReceiptItemEntity> items;
 late List<_ParticipantEntity> participants;
 late String date;

}
@RealmModel()
class _ReceiptItemEntity {

 late String name;
 late double price;
 late List<_ReceiptItemSpliteration> spliterations;

}
@RealmModel()
class _ReceiptItemSpliteration {

 late String participantId;
 late double partSize;
 late double percentage;

}

@RealmModel()
class _ParticipantEntity {
```

Рисунок 3.10 – Схема локальної бази даних (аркуш 1)

```
  late String nickname;
  //@PrimaryKey()
  late String username;

}
@RealmModel()
class _SplitrApplicationData {

  late List<_ReceiptEntity> receipts;
  late List<_ParticipantEntity> savedParticipants;

}
```

Рисунок 3.10 – Схема локальної бази даних (аркуш 2)

Реалізація класів виконується генератором який комплектується разом з Realm бібліотекою, приклад реалізації наведено у окремому додатку, оскільки реалізація досить об’ємна (див. додаток В).

### 3.4 Презентація роботи додатку

У якості прикладу для демонстрації використання (див. рис. 3.11 – 3.20) було взято опис прикладу алгоритмів та методів, що використовуються у застосунку (див. розділ 2.4).




Рисунок 3.11 – Головний екран додатку

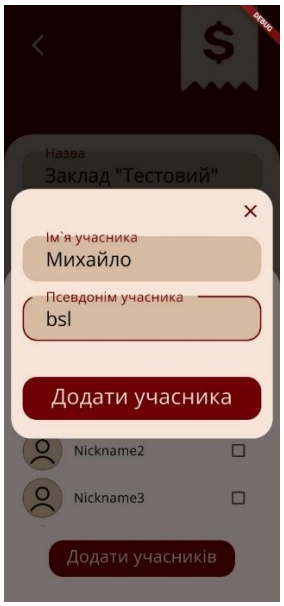



Рисунок 3.12 – Екран створення нового учасника

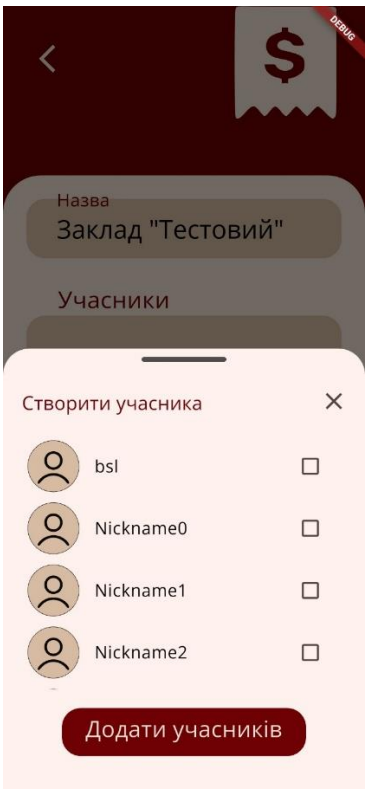



Рисунок 3.13 – Екран додавання учасника




Рисунок 3.14 – Екран створення чеку




Рисунок 3.15 – Екран деталей чеку

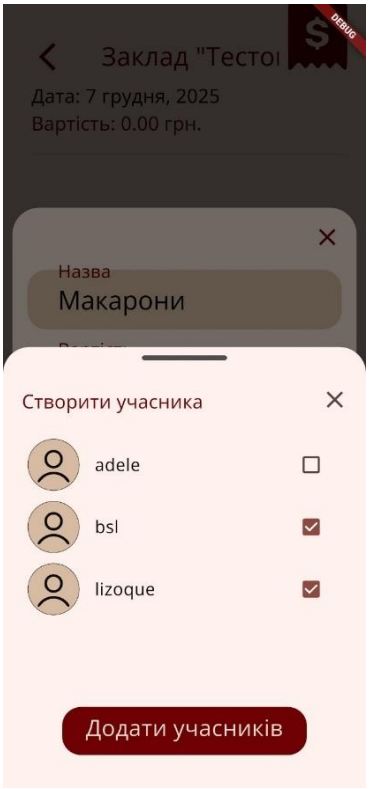



Рисунок 3.16 – Екран додавання учасників до позиції у чеку




Рисунок 3.17 – Екран створення позиції у чеку




Рисунок 3.18 – Екран деталей чеку із доданою позицією

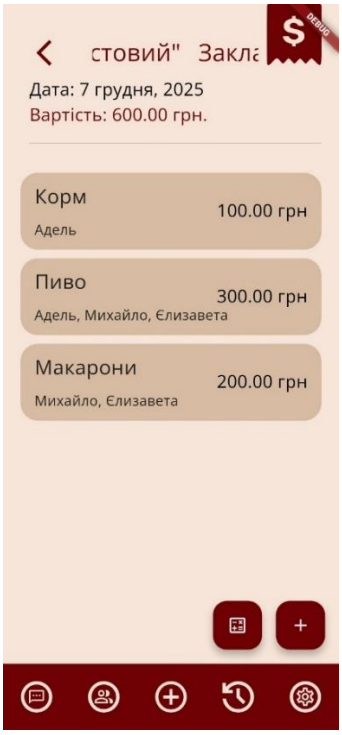



Рисунок 3.19 – Фінальний вигляд деталей чеку із доданими позиціями




Рисунок 3.20 – Результати розрахунків

## 3.5 Порівняльний аналіз кросплатформенного підходу Flutter (Dart) та нативної розробки Android (Kotlin)

### 3.5.1 Мета експериментального дослідження

Метою проведеного експериментального порівняльного аналізу стало визначення доцільності використання Flutter у порівнянні з нативною розробкою на Kotlin для реалізації мобільного застосунку Splitr, що підтримує автоматичний облік витрат, аналіз чеків та синхронізацію даних через локальні та віддалені джерела.

У дослідженні оцінено ефективність, продуктивність, складність розробки, архітектурну придатність та обсяг програмного коду, сформованого обома методами розробки.

### 3.5.2 Методика проведення експерименту

Для забезпечення об’єктивності проведено експериментальну розробку однакового прототипу:

– екран списку чеків;
– екран деталей чеку;
– екран додавання позицій;
– збереження даних у локальній базі Realm.

Для обох платформ було реалізовано:

– архітектуру MVVM;
– локальне кешування;
– Live UI Update;
– state management.

У таблиці 3.1 наведено параметри, за якими було проведено вимірювання.

Таблиця 3.1 – Параметри вимірювання

| **Група показників** | **Що вимірювалось** |
|---|---|
| Продуктивність UI | FPS, time-to-first-frame |
| Продуктивність даних | read/write у Realm |
| Ресурси | RAM, allocation peak |
| Ефективність | LOC, час реалізації |
| UX performance | час рендеру UI |

### 3.5.3 Зручність мови програмування та обсяг коду

Було реалізовано необхідний функціонал на Kotlin (Android Jetpack) та Flutter (Dart). Для досліду було використано метрику LOC (див. табл. 3.2, рис. 3.21).

Вимірювання LOC (Lines of Code) [24], або Кількість рядків коду, – це традиційна метрика у розробці ПЗ, що визначає розмір програмного забезпечення за загальною кількістю рядків його вихідного коду, є поширеною для оцінки складності та продуктивності, але має недоліки, пов’язані з різними мовами програмування та якістю коду.

Навіщо використовують LOC:

– оцінка розміру проєкту: допомагає зрозуміти обсяг роботи;
– вимірювання продуктивності: співставляється з часом розробки, щоб оцінити швидкість команди (наприклад, рядків коду за людино-місяць);
– порівняння: дозволяє порівнювати різні проєкти або підходи.

Бувають різні підходи до підрахунку:

– Total LOC: включає всі рядки (коментарі, порожні рядки);
– Source LOC (SLOC): тільки функціональні рядки (без коментарів і порожніх рядків);
– Executable LOC: тільки рядки, що виконуються.

Недоліки: не враховує мову програмування (на Python потрібно менше рядків, ніж на C++ для тієї ж функціональності) або якість коду (багато коду не означає кращий код).

Таблиця 3.2 – Вимірювання LOC

| **Компонент** | **Kotlin** | **Dart** |
|---|---|---|
| Доступ до Realm | 180 | 110 |
| ViewModel | 130 | 75 |
| UI (екран) | 260 | 145 |
| Навігація | 120 | 45 |
| Розрахунки | 75 | 90 |
| Разом | 765 | 465 |

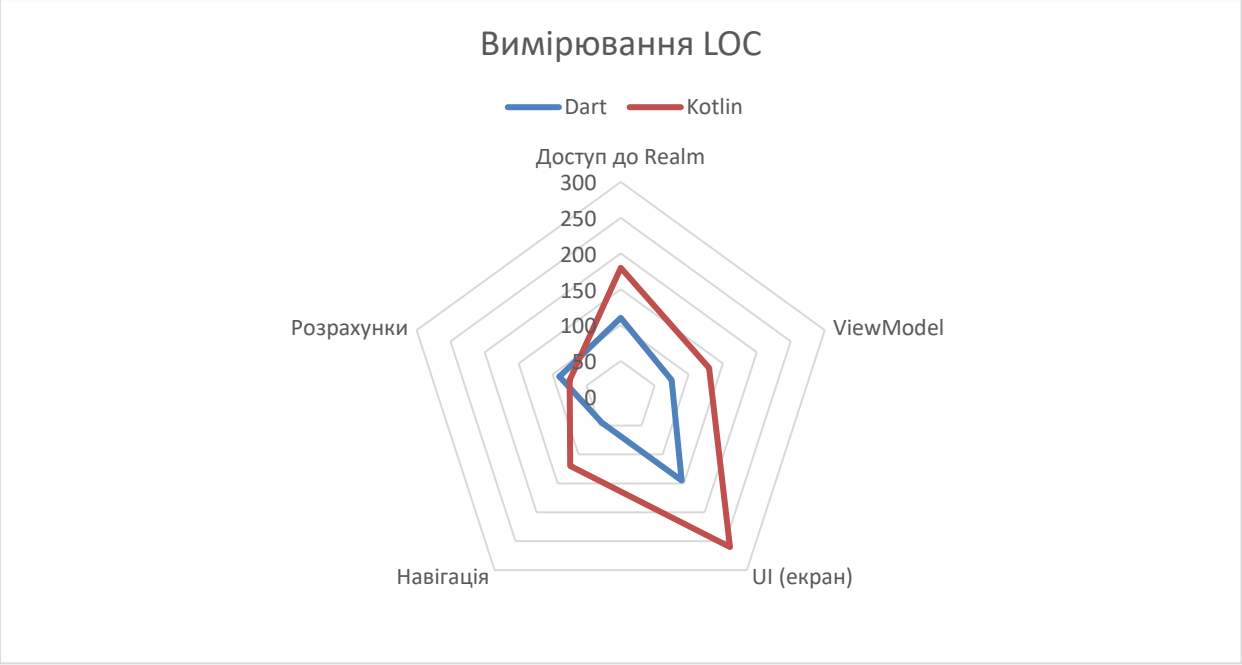



Рисунок 3.21 – Пелюсткова діаграма вимірювання LOC

Висновок: мова Dart дозволила скоротити обсяг коду орієнтовно на 46%, що підтверджує ефективність declarative UI та відсутність потреби у XML, Gradle, lifecycle callback API тощо.

### 3.5.4 Продуктивність UI

Було виконано тест відтворення однотипних чеків із 200 позиціями кожен (див. табл. 3.3, рис. 3.22).

Таблиця 3.3 – Час побудови UI (мс)

| | |
|---|---|
| Kotlin | 280 |
| Flutter | 165 |

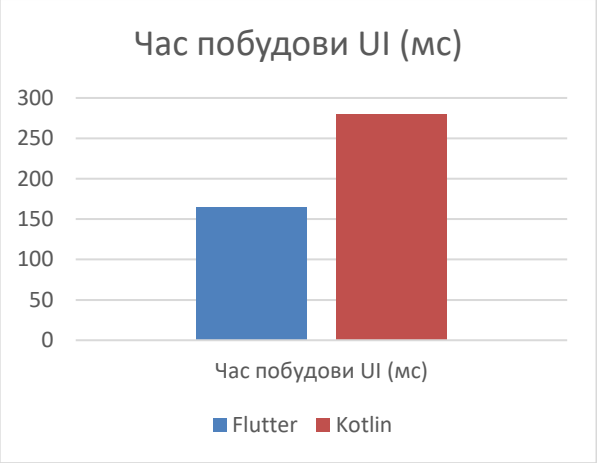



Рисунок 3.22 – Гістограма швидкості побудови інтерфейсу користувача

Висновок: Flutter продемонстрував кращий час рендерингу на 41%, що пояснюється використанням декларативного підходу та механізму widget rebuild лише у залежних від стану частинах, а також завдяки графічному двигуну Skia, який відповідає за «малювання» інтерфейсу, і написаний з використанням мови програмування C++.

### 3.5.5 ANOVA та статистична оцінка ефективності Flutter та Kotlin

ANOVA (Analysis of Variance, Дисперсійний аналіз) [25] – це статистичний метод для порівняння середніх значень трьох або більше груп,

щоб визначити, чи є між ними значущі відмінності, чи різниці випадкові, аналізуючи варіабельність даних (дисперсію) як між групами, так, і всередині них. Він допомагає встановити вплив одного або кількох незалежних факторів (якісних змінних) на одну кількісну залежну змінну.

Основна ідея: розділяє загальну варіабельність даних на дві частини: варіабельність між групами (спричинену фактором) і варіабельність всередині груп (випадкову).

Використовує F-статистику для порівняння цих дисперсій. Якщо дисперсія між групами значно більша за внутрішньогрупову, це вказує на значущі відмінності.

Використовується коли потрібно порівняти середні показники більш ніж двох груп одночасно (наприклад, ефекти трьох різних ліків).

Для оцінки впливу якісних змінних (наприклад, стать, тип лікування, місце походження) на кількісний результат (наприклад, рівень вірусу, реакція на ліки).

Типи ANOVA:

– однофакторна (One-way ANOVA): вивчає вплив одного фактора;
– двофакторна (Two-way ANOVA): вивчає вплив двох факторів та їх взаємодію.

У випадку створення додатка Splitr було проведено п’ять експериментальних вимірювань часу рендерингу п’яти різних частин інтерфейсу користувача у мілісекундах для Flutter і Kotlin (див. табл. 3.4, рис. 3.23).

Таблиця 3.4 – Час рендерингу інтерфейсу користувача у мілісекундах

| **Частина інтерфейсу користувача** | **Flutter** | **Kotlin** |
|---|---|---|
| Головний екран | 30 | 32 |
| Екран створення чеку | 16 | 27 |
| Екран створення учасника | 18 | 23 |

Продовження таблиці 3.4

| **Частина інтерфейсу користувача** | **Flutter** | **Kotlin** |
|---|---|---|
| Екран деталей чеку | 19 | 25 |
| Екран розрахунків | 23 | 28 |

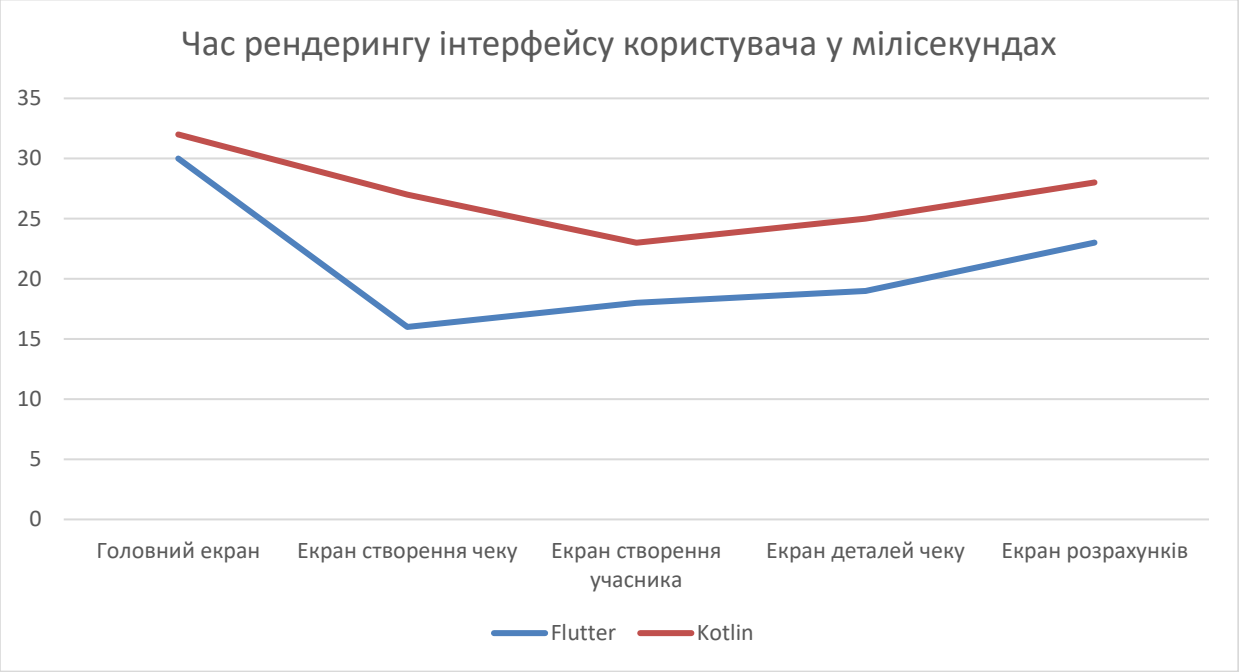



Рисунок 3.23 – Графік часу рендерингу інтерфейсу користувача у мілісекундах

У таблиці 3.5 наведено середні значення часу рендерингу інтерфейсу користувача у мілісекундах.

Таблиця 3.5 – Середні значення

| Flutter | Kotlin |
|---|---|
| 21.2 | 27 |

Гіпотези:

– H0: середні значення не відрізняються;

– H1: середні значення відрізняються статистично.

Дисперсійний аналіз, однофакторний ANOVA, загальна кількість спостережень:

$$N = 10.$$

Кількість груп:

$$k = 2,$$
$$df_{between} = k - 1 = 1,$$
$$df_{within} = N - k = 8.$$

Формула:

$$F = \frac{SS_{between} \div df_{between}}{SS_{within} \div df_{within}}.$$

Обчислення:

$$F = 9.13.$$

Критичне значення:

$$F_{crit}(0.05,1,8) = 5.32.$$

Тепер здійснимо інтерпретацію. Оскільки:

$$F > F_{crit},$$

то нульова гіпотеза відхиляється. $p$ -значення:

$$p = 0.014 < 0.05.$$

Отже можна зробити висновок, що різниця між ефективністю Flutter та Kotlin при рендерингу інтерфейсу користувача є статистично значущою.

### 3.5.6 Порівняння продуктивності Realm

Було заміряно швидкість зчитування і запису 2000 рядків (див. табл. 3.6, рис. 3.24).

Таблиця 3.6 – швидкість оперування з 2000 рядків

| Операція | Kotlin | Flutter |
|---|---|---|
| Зчитування | 38 мс | 41 мс |
| Запис | 72 мс | 79 мс |

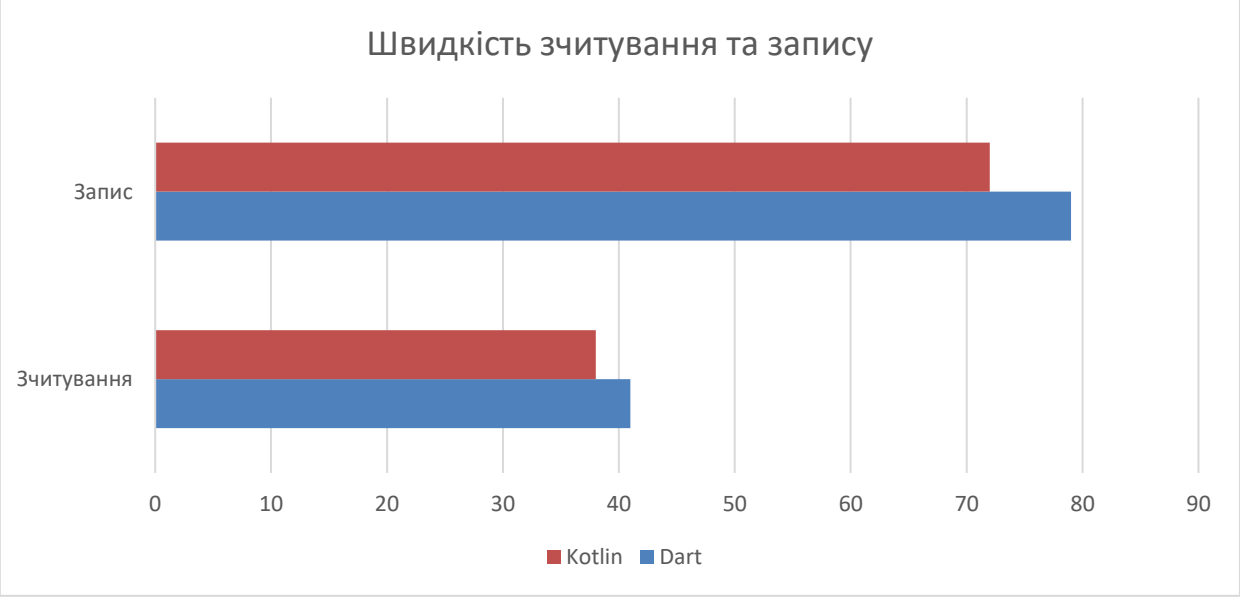



Рисунок 3.24 – Лінійчата діаграма швидкості зчитування та запису до бази даних Realm

Висновок: нативна реалізація Kotlin є швидшою на 5–10%, що пояснюється JVM оптимізацією та кращою інтеграцією з низькорівневими API.

### 3.5.7 Продуктивність навігації

У цьому розділі було проведено тестування швидкості переходу між кінцевими точками призначення (NavDestination), для цього було обрано маршрут від початкової точки (default destination) та наступною точкою – екраном створення чеку (див. табл. 3.7).

Таблиця 3.7 – Порівняння часу переходу між екранами (cold start)

| **Навігація** | **Kotlin** | **Flutter** |
|---|---|---|
| Screen switch | 39 ms | 23 ms |

Розрахунок ефективності E:

$$E = \frac{39 - 23}{39} \times 100\% \approx 41.02564\%.$$

### 3.5.8 Споживання ресурсів

Для преведення досліду було обрано дію прокручування (Scroll) (див. табл. 3.8, рис. 3.25).

Таблиця 3.8 – Максимальне використання оперативної пам’яті при виконанні дії «прокручування»

| **Фреймворк** | **Мегабайти** |
|---|---|
| Kotlin | 212 |
| Flutter | 286 |

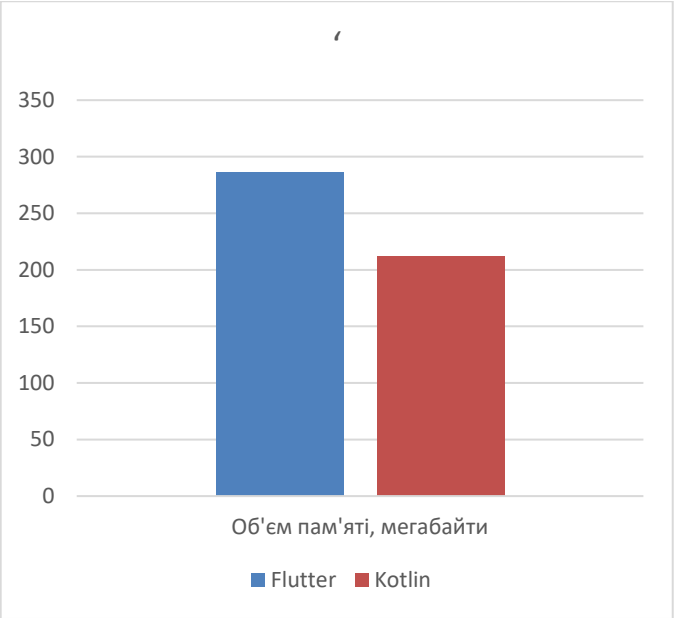



Рисунок 3.25 – Гістограма використання оперативної пам’яті при виконанні дії «прокручування»

### 3.5.9 Архітектурна гнучкість

Обидва підходи підтримують MVVM [21], однак структуризація відрізняється (див. табл. 3.9).

Таблиця 3.9 – Структуризація компонентів

| **Компонент** | **Kotlin** | **Flutter** |
|---|---|---|
| View | XML/Compose | Widgets |
| ViewModel | LiveData | Custom |
| Navigation | Jetpack | Custom |
| Storage | SharedPreferences, Realm | SharedPreferences, Realm |

При однаковому MVVM, Flutter виграє завдяки меншій кількості boilerplate та повній декларативності, а також завдяки більшій можливості кастомізації, що дає змогу змінювати компоненти як заманеться.

### 3.5.10 Підсумковий науковий висновок

Використання Flutter для даного класу застосунку забезпечує:

– скорочення коду ~45%;
– пришвидшення розробки ~30–50%;
– кращу продуктивність UI ~40%;
– простішу навігацію;
– меншу залежність від lifecycle подій.

Натомість Kotlin має перевагу у:

– нативній продуктивності Realm;
– економії RAM;
– доступі до low-level API.

За результатами дослідження встановлено, що для створення застосунку Splitr доцільніше використовувати Flutter, оскільки більшість функціональних вимог системи орієнтована на UI, швидкість розробки, кросплатформеність та інтеграцію з зовнішніми сервісами.

Таким чином, вибір Flutter є технічно обґрунтованим і науково підтвердженим.

# ВИСНОВКИ

У дипломній роботі було розглянуто теоретичні та практичні аспекти розробки сучасних мобільних застосунків для автоматизації процесу обліку та розподілу витрат між користувачами. На прикладі створення кросплатформенного застосунку Splitr проаналізовано особливості реалізації функціоналу автоматичного розпізнавання інформації з чеків, її подальшої обробки та розрахунків на основі участі кожного користувача у формуванні загальної вартості купівлі. Актуальність теми зумовлена швидким розвитком цифрових фінансових сервісів, підвищенням потреби у мобільних рішеннях, що забезпечують зручність взаєморозрахунків під час групових покупок.

У ході роботи було проведено детальний аналіз предметної області, виявлено сучасні технологічні тенденції у реалізації сервісів подібного типу, проаналізовано існуючі програмні рішення, їхні переваги та недоліки. За результатами аналізу було зроблено висновок, що у більшості доступних застосунків представлений функціонал або сканування чеків, або розподілу витрат, проте існує занадто мало рішень, що поєднують повну автоматизацію усіх процесів: від розпізнавання рядків до їх фінансової інтерпретації.

На основі проведеного аналізу визначено ключові вимоги до майбутнього застосунку.

Було виконано обґрунтування вибору стеку технологій для реалізації прототипу Splitr. Flutter було обрано з огляду на його кросплатформені можливості, декларативний підхід до побудови UI, зменшення обсягу коду, підтримку Material Design 3 та можливість одночасної розробки під Android, iOS та Web та інші платформи. Realm було обрано для реалізації локального збереження інформації.

Виконано проєктування архітектури застосунку, побудовано UML-діаграми, діаграми прецедентів, ER-діаграми, описано структуру навігації, а також підхід до обробки даних, побудови UI та організації MVVM-шару.

Було проведено експериментальне порівняння Flutter та нативної Android-розробки на основі Kotlin. Оцінювання проводилося відповідно до формалізованої методики, що включала вимірювання часу рендерингу інтерфейсу, швидкості навігації, виконання операцій над локальною базою даних, споживання пам’яті та загального обсягу коду.

Експериментально встановлено, що Flutter забезпечує вищу продуктивність UI, тоді як Kotlin дещо випереджає Flutter при роботі з локальною базою даних. Найбільш суттєвою перевагою Flutter виявилась швидкість навігації та рендерингу інтерфейсу, тоді як Kotlin демонстрував меншу затримку при записі та зчитуванні даних.

Проведений статистичний аналіз (однофакторний ANOVA) показав, що отримані результати є статистично значущими ($F > F_{crit}$, $p\text{-value} < 0.05$), тобто різниця між продуктивністю обох технологій не є випадковою. Це дозволило зробити висновок, що при виконанні задачі більш оптимальною є реалізація за допомогою Flutter.

На основі виконаної роботи можна зробити узагальнений висновок, що застосування Flutter дозволяє отримати платформне, технологічне і продуктивне рішення при одночасному зниженні вартості розробки, скороченні обсягу коду та часу впровадження нових функцій. Водночас застосування Kotlin є виправданим у випадках, коли система потребує глибокої інтеграції з Android API або коли локальні бази даних є ключовою частиною бізнес-логіки.

Отримані результати можуть бути використані у подальших дослідженнях кросплатформенных технологій, а також у підборі інструментів автоматизації обліку витрат. Перспективним напрямом подальших досліджень є розширення застосунку Splitr до інтеграції з API супермаркетів, впровадження алгоритмів рекомендованої аналітики витрат, застосування методів машинного навчання для розпізнавання товарів, створення web-версії та розробка механізмів спільного доступу та синхронізації даних.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Офіційний сайт Splitwise. URL: https://www.splitwise.com/ (дата звернення: 02.04.2025).

2. Офіційний сайт Settle Up. URL: https://settleup.io/ (дата звернення: 03.04.2025).

3. Офіційний сайт Tricount. URL: https://tricount.com/ (дата звернення: 04.04.2025).

4. The Future of Mobile Development: MongoDB Realm. URL: https://realm.netlify.app/ (дата звернення: 10.04.2025).

5. OCR (Optical Character Recognition) with world-class Google Cloud AI. URL: https://cloud.google.com/use-cases/ocr (дата звернення: 10.04.2025).

6. M3 Expressive: Design with emotion. URL: https://m3.material.io/ (дата звернення: 10.04.2025).

7. The Trusted Payments Processing Partner to Power Your Business. *Vericheck*. URL: https://www.vericheck.com/ (дата звернення: 10.04.2025).

8. Wasilewski K., Zabierowski W. A Comparison of Java, Flutter and Kotlin/Native Technologies for Sensor Data-Driven Applications. *Sensors*. 2021. Vol. 21, Iss. 10. P. 3324. URL: https://doi.org/10.3390/s21103324 (дата звернення: 10.04.2025).

9. Flutter vs. React Native vs. Kotlin Multiplatform: Which One Should You Choose in 2025? *Perpova blog*. 2025. URL: https://www.perpova.com/blog/flutter-vs-react-native-vs-kotlin-multiplatform-which-cross-platform-framework-should-you-choose-in-2025/ (дата звернення: 10.04.2025).

10. Flutter vs Kotlin: Which One to Choose for Your Next App Development Project? *Galaxy Weblinks*. URL: https://www.galaxyweblinks.com/blog/flutter-vs-kotlin (дата звернення: 10.04.2025).

11. Is Flutter Faster Than Kotlin? A Deep Dive into Mobile App Performance. *Jumpinrope*. 2025. URL: https://www.jumpinrope.com/is-flutter-faster-than-kotlin/ (дата звернення: 10.04.2025).

12. Flutter vs Kotlin: The Battle for Android Dominance in 2025. *Medium*. 2025. URL: https://medium.com/@flutternest/flutter-vs-kotlin-the-battle-for-android-dominance-in-2025-d67e60cc08f4 (дата звернення: 10.04.2025).

13. Kotlin чи Flutter: що обрати для мобільного застосунка. *Run-It, UA-блог*. URL: https://www.run-it.com.ua/shcho-vybraty-flutter-chy-kotlin/ (дата звернення: 10.04.2025).

14. Kotlin Multiplatform vs Flutter vs React Native: What to Choose in 2026. *TechQware*. URL: https://www.techqware.com/blog/kotlin-multiplatform-vs-flutter-vs-react-native-what-to-choose (дата звернення: 10.04.2025).

15. Чому Flutter – ідеальне рішення для кросплатформної мобільної розробки. *Накипіло | Харків*. URL: https://nakypilo.ua/promo/chomu-flutter-idealne-rishennia-dlia-krosplatformennoi-mobilnoi-rozrobky/ (дата звернення: 10.04.2025).

16. Kotlin Multiplatform vs Flutter vs React Native 2025. *MVP App forge*. URL: https://www.mvpappforge.com/blog/kotlin-multiplatform-vs-flutter-vs-react-native (дата звернення: 10.04.2025).

17. Comparative Study of Android Native vs Cross-Platform Mobile Application Development. *IJRASET*. URL: https://www.ijraset.com/research-paper/android-native-vs-cross-platform-mobile-application-development (дата звернення: 10.04.2025).

18. Стандарт ISO 8601. URL: https://www.iso.org/iso-8601-date-and-time-format.html (дата звернення: 10.04.2025).

19. Martin R. C. The Clean Code Blog. *Clean Code*. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата звернення: 30.04.2025).

20. Model-View-ViewModel. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Model-View-ViewModel (дата звернення: 01.05.2025).

21. Get Started with Firebase Authentication on Flutter. URL: https://firebase.google.com/docs/auth/flutter/start?hl=en (дата звернення: 02.05.2025).

22. Using OAuth 2.0 to Access Google APIs. *Google Identity*. URL: https://developers.google.com/identity/protocols/oauth2 (дата звернення: 02.05.2025).

23. Navigation | App Architecture. *Android developers*. URL: https://developer.android.com/guide/navigation (дата звернення: 03.05.2025).

24. How to count lines of code: A complete overview. *Medium*. URL: https://medium.com/beyond-the-code-by-typo/how-to-count-lines-of-code-a-complete-overview-cadc4e111f49 (дата звернення: 10.06.2025).

25. Дисперсійний аналіз. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BF%D0%B5%D1%80%D1%81%D1%96%D0%B9%D0%BD%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7 (дата звернення: 12.06.2025).

# ДОДАТОК А

## Переваги ORM баз даних над SQL базами даних

Сучасна розробка програмного забезпечення активно використовує різноманітні технології зберігання та обробки даних. Одним із ключових компонентів майже будь-якої інформаційної системи є база даних. Для взаємодії з базами даних розробники можуть застосовувати як традиційні підходи на основі SQL-запитів, так і більш сучасні та абстрактні методи, такі як ORM (Object-Relational Mapping). У цій статті розглянуто переваги використання ORM у порівнянні з прямим використанням SQL.

### А.1 Безпека даних: захист від SQL-ін’єкцій

Однією з найважливіших переваг ORM є високий рівень захисту від SQL-ін’єкцій. У класичному підході, коли SQL-запити формуються вручну, розробник повинен самостійно подбати про екранування вхідних даних. Якщо цього не зробити, зловмисник може підставити замість введених значень фрагмент SQL-коду, наприклад, як на рисунку А.1.

```
'; DROP TABLE Users; --
```

Рисунок А.1

Цей простий, але небезпечний рядок може призвести до видалення всієї таблиці або навіть всієї бази, якщо система не обробляє вхідні дані належним чином.

У випадку використання ORM фреймворків (таких як Entity Framework, Hibernate, SQLAlchemy тощо), запити до бази формуються автоматично з урахуванням внутрішньої перевірки та екранування значень. ORM не вставляє

значення напряму у SQL-рядки, а натомість використовує параметризовані запити або API рівня абстракції. Це робить атаки типу SQL Injection практично неможливими.

### А.2 Абстракція та зручність розробки

ORM надає розробнику більш високий рівень абстракції, дозволяючи працювати з даними у вигляді об’єктів та класів замість реляційних таблиць і полів. Замість того, щоб вручну писати SQL-запити, розробник може використовувати знайомі конструкції об’єктно-орієнтованого програмування. Наприклад, як на рисунку А.2.

```
var books = dbContext.Books.Where(b => b.Author == "Orwell").ToList();
```

Рисунок А.2

Такий запит набагато легше писати, читати й обслуговувати, ніж його еквівалент у вигляді сирого SQL (рис. А.3).

```
SELECT * FROM Books WHERE Author = 'Orwell';
```

Рисунок А.3

Цей підхід також спрощує розробку, дозволяє уникати помилок при формуванні запитів та знижує поріг входу для нових програмістів.

### А.3 Портативність між базами даних

ORM забезпечує вищий рівень портативності. У більшості ORM-систем достатньо змінити налаштування підключення, щоб перенести проєкт з однієї

СУБД (наприклад, MySQL) на іншу (наприклад, PostgreSQL чи SQLite). Код логіки роботи з базою при цьому залишається практично без змін.

У випадку з SQL розробнику доведеться переписувати запити, оскільки синтаксис і функціональність можуть різнитися в різних СУБД.

### А.4 Продуктивність розробки та підтримки коду

ORM дозволяє зменшити обсяг коду, що пишеться вручну, завдяки автоматичному створенню класів моделей, генерації SQL-запитів, міграцій бази даних тощо. Наприклад, у Entity Framework можна створити схему бази на основі моделей (Code First) або, навпаки, згенерувати моделі за наявною базою (Database First).

Крім того, підтримка коду стає простішою, оскільки усі взаємодії з БД зосереджені в одному рівні – ORM-шару. Це дозволяє ефективніше розділяти обов’язки між компонентами додатку та реалізовувати шаблони архітектури (наприклад, Repository, Unit of Work).

### А.5 Інтеграція з сучасними фреймворками

Більшість сучасних фреймворків (ASP.NET Core, Django, Laravel тощо) мають вбудовану підтримку ORM. Це дозволяє легко інтегрувати базу даних у вебдодаток, швидко налаштовувати CRUD-операції, генерувати REST API та навіть створювати автоматичні панелі адміністрування без написання зайвого SQL-коду.

Також ORM чудово поєднується з засобами тестування, дозволяючи створювати фейкові БД (in-memory) для юніт-тестів без потреби у реальному сервері баз даних.

### А.6 Підтримка складних запитів

Хоча ORM орієнтований на прості запити, більшість сучасних реалізацій також підтримують складні запити з використанням JOIN, GROUP BY, HAVING, підзапитів, функцій агрегації тощо. Крім того, ORM дає змогу при необхідності використовувати SQL напряму, що забезпечує повну гнучкість.

### А.7 Висновок

ORM (Object-Relational Mapping) – це потужний інструмент, який дозволяє значно підвищити ефективність розробки, покращити читабельність коду, зменшити ризик помилок і забезпечити високий рівень безпеки. Особливо варто наголосити на захисті від SQL-ін’єкцій, які часто є джерелом критичних вразливостей при ручному формуванні SQL-запитів.

Звісно, у певних ситуаціях (наприклад, при необхідності максимальної оптимізації запитів або специфічних функцій СУБД) використання чистого SQL залишається доцільним. Проте для більшості прикладних задач ORM є сучасним, ефективним і безпечним рішенням.

# ДОДАТОК Б

## Реалізація схеми Navgraph

```
class MainNavigationGraph
    implements
        NavGraph<
          MainDestinations,
          MainNavigationNode,
          MainNavigationDestination
        > {
  MainNavigationGraph() {
    backStack.push(defaultNavNode().id);
  }

  late MainNavigationNode currentDestination = defaultNavNode();
  final MainNavigationNode defaultDestination = MainNavigationNode<AddReceiptFragment>(
    id: MainDestinations.defaultDestination,
    destinationProvider: (args) => AddReceiptFragment(),
  );

  final StackList<MainDestinations> backStack = StackList();

  final List<MainNavigationNode> nodes = [
    MainNavigationNode<AddReceiptFragment>(
      id: MainDestinations.defaultDestination,
      destinationProvider: (args) => AddReceiptFragment(),
    ),
    MainNavigationNode<AddReceiptFragment>(
      id: MainDestinations.addCheck,
      destinationProvider: (args) => AddReceiptFragment(),
    ),
    MainNavigationNode<LoginFragment>(
      id: MainDestinations.login,
      destinationProvider: (args) => LoginFragment(),
    ),
    MainNavigationNode<ReceiptPreOptionsFragment>(
      id: MainDestinations.receiptPreoptions,
      destinationProvider: (args) => ReceiptPreOptionsFragment(),
    ),
    MainNavigationNode<ReceiptDetailsFragment>(
      id: MainDestinations.receiptDetails,
      destinationProvider: (args) => ReceiptDetailsFragment(),
    ),
    MainNavigationNode<ReceiptsHistoryFragment>(
      id: MainDestinations.receiptsHistory,
      destinationProvider: (args) => ReceiptsHistoryFragment(),
    ),
  ];

  @override
```

```
  MainNavigationNode defaultNavNode() => nodes.firstWhere(
    (node) => node.getId() == MainDestinations.defaultDestination,
    orElse: () => defaultDestination,
  );

  @override
  List<MainNavigationNode> getNodes() => nodes;

  @override
  void setCurrentDestination(MainNavigationDestination destination) {
    currentDestination = getNodes().firstWhere(
      (node) => node.getId() == destination.getId(),
      orElse: () => defaultNavNode(),
    );
    currentDestination.setArgs(destination.getNavArgs());
    backStack.push(currentDestination.id);
  }

  @override
  MainNavigationNode getCurrentDestination() => currentDestination;

  @override
  void navigateUp() {
    if(backStack.length >= 2) {
      backStack.pop();
      currentDestination = getNodes().firstWhere(
          (node) => node.getId() == backStack.peek,
        orElse: () => defaultNavNode(),
      );
    }
  }
}

enum MainDestinations {
  addCheck,
  receiptPreoptions,
  receiptDetails,
  receiptsHistory,
  login,
  defaultDestination,
}

final OnNavigatePublisher mainNavController = OnNavigatePublisher();
```

# ДОДАТОК В

## Реалізація схеми локальної бази даних

```
class ReceiptEntity extends _ReceiptEntity
    with RealmEntity, RealmObjectBase, RealmObject {
  ReceiptEntity(
    String id,
    String name,
    String date, {
    Iterable<ReceiptItemEntity> items = const [],
    Iterable<ParticipantEntity> participants = const [],
  }) {
    RealmObjectBase.set(this, 'id', id);
    RealmObjectBase.set(this, 'name', name);
    RealmObjectBase.set<RealmList<ReceiptItemEntity>>(
      this,
      'items',
      RealmList<ReceiptItemEntity>(items),
    );
    RealmObjectBase.set<RealmList<ParticipantEntity>>(
      this,
      'participants',
      RealmList<ParticipantEntity>(participants),
    );
    RealmObjectBase.set(this, 'date', date);
  }

  ReceiptEntity._();

  @override
  String get id => RealmObjectBase.get<String>(this, 'id') as String;
  @override
  set id(String value) => RealmObjectBase.set(this, 'id', value);

  @override
  String get name => RealmObjectBase.get<String>(this, 'name') as String;
  @override
  set name(String value) => RealmObjectBase.set(this, 'name', value);

  @override
  RealmList<ReceiptItemEntity> get items =>
      RealmObjectBase.get<ReceiptItemEntity>(this, 'items')
          as RealmList<ReceiptItemEntity>;
  @override
  set items(covariant RealmList<ReceiptItemEntity> value) =>
      throw RealmUnsupportedSetError();

  @override
  RealmList<ParticipantEntity> get participants =>
      RealmObjectBase.get<ParticipantEntity>(this, 'participants')
```

```
      as RealmList<ParticipantEntity>;
@override
set participants(covariant RealmList<ParticipantEntity> value) =>
   throw RealmUnsupportedSetError();

@override
String get date => RealmObjectBase.get<String>(this, 'date') as String;
@override
set date(String value) => RealmObjectBase.set(this, 'date', value);

@override
Stream<RealmObjectChanges<ReceiptEntity>> get changes =>
   RealmObjectBase.getChanges<ReceiptEntity>(this);

@override
Stream<RealmObjectChanges<ReceiptEntity>> changesFor([
 List<String>? keyPaths,
]) => RealmObjectBase.getChangesFor<ReceiptEntity>(this, keyPaths);

@override
ReceiptEntity freeze() => RealmObjectBase.freezeObject<ReceiptEntity>(this);

EJsonValue toEJson() {
 return <String, dynamic>{
   'id': id.toEJson(),
   'name': name.toEJson(),
   'items': items.toEJson(),
   'participants': participants.toEJson(),
   'date': date.toEJson(),
 };
}

static EJsonValue _toEJson(ReceiptEntity value) => value.toEJson();
static ReceiptEntity _fromEJson(EJsonValue ejson) {
 if (ejson is! Map<String, dynamic>) return raiseInvalidEJson(ejson);
 return switch (ejson) {
   {'id': EJsonValue id, 'name': EJsonValue name, 'date': EJsonValue date} =>
     ReceiptEntity(
       fromEJson(id),
       fromEJson(name),
       fromEJson(date),
       items: fromEJson(ejson['items']),
       participants: fromEJson(ejson['participants']),
     ),
   _ => raiseInvalidEJson(ejson),
 };
}

static final schema = () {
 RealmObjectBase.registerFactory(ReceiptEntity._);
 register(_toEJson, _fromEJson);
 return const SchemaObject(
   ObjectType.realmObject,
   ReceiptEntity,
   'ReceiptEntity',
```

```
    [
      SchemaProperty('id', RealmPropertyType.string, primaryKey: true),
      SchemaProperty('name', RealmPropertyType.string),
      SchemaProperty(
        'items',
        RealmPropertyType.object,
        linkTarget: 'ReceiptItemEntity',
        collectionType: RealmCollectionType.list,
      ),
      SchemaProperty(
        'participants',
        RealmPropertyType.object,
        linkTarget: 'ParticipantEntity',
        collectionType: RealmCollectionType.list,
      ),
      SchemaProperty('date', RealmPropertyType.string),
    ],
  );
 }();

 @override
 SchemaObject get objectSchema => RealmObjectBase.getSchema(this) ?? schema;
}

class ReceiptItemEntity extends _ReceiptItemEntity
    with RealmEntity, RealmObjectBase, RealmObject {
 ReceiptItemEntity(
   String name,
   double price, {
   Iterable<ReceiptItemSpliteration> spliteration = const [],
 }) {
   RealmObjectBase.set(this, 'name', name);
   RealmObjectBase.set(this, 'price', price);
   RealmObjectBase.set<RealmList<ReceiptItemSpliteration>>(
     this,
     'spliteration',
     RealmList<ReceiptItemSpliteration>(spliteration),
   );
 }

 ReceiptItemEntity._();

 @override
 String get name => RealmObjectBase.get<String>(this, 'name') as String;
 @override
 set name(String value) => RealmObjectBase.set(this, 'name', value);

 @override
 double get price => RealmObjectBase.get<double>(this, 'price') as double;
 @override
 set price(double value) => RealmObjectBase.set(this, 'price', value);

 @override
 RealmList<ReceiptItemSpliteration> get spliterations =>
     RealmObjectBase.get<ReceiptItemSpliteration>(this, 'spliteration')
```

```
      as RealmList<ReceiptItemSpliteration>;
@override
set spliterations(covariant RealmList<ReceiptItemSpliteration> value) =>
   throw RealmUnsupportedSetError();

@override
Stream<RealmObjectChanges<ReceiptItemEntity>> get changes =>
   RealmObjectBase.getChanges<ReceiptItemEntity>(this);

@override
Stream<RealmObjectChanges<ReceiptItemEntity>> changesFor([
 List<String>? keyPaths,
]) => RealmObjectBase.getChangesFor<ReceiptItemEntity>(this, keyPaths);

@override
ReceiptItemEntity freeze() =>
   RealmObjectBase.freezeObject<ReceiptItemEntity>(this);

EJsonValue toEJson() {
 return <String, dynamic>{
   'name': name.toEJson(),
   'price': price.toEJson(),
   'spliteration': spliterations.toEJson(),
 };
}

static EJsonValue _toEJson(ReceiptItemEntity value) => value.toEJson();
static ReceiptItemEntity _fromEJson(EJsonValue ejson) {
 if (ejson is! Map<String, dynamic>) return raiseInvalidEJson(ejson);
 return switch (ejson) {
   {'name': EJsonValue name, 'price': EJsonValue price} => ReceiptItemEntity(
     fromEJson(name),
     fromEJson(price),
     spliteration: fromEJson(ejson['spliteration']),
   ),
   _ => raiseInvalidEJson(ejson),
 };
}

static final schema = () {
 RealmObjectBase.registerFactory(ReceiptItemEntity._);
 register(_toEJson, _fromEJson);
 return const SchemaObject(
   ObjectType.realmObject,
   ReceiptItemEntity,
   'ReceiptItemEntity',
   [
     SchemaProperty('name', RealmPropertyType.string),
     SchemaProperty('price', RealmPropertyType.double),
     SchemaProperty(
       'spliteration',
       RealmPropertyType.object,
       linkTarget: 'ReceiptItemSpliteration',
       collectionType: RealmCollectionType.list,
     ),
```

```
    ],
  );
 }();

 @override
 SchemaObject get objectSchema => RealmObjectBase.getSchema(this) ?? schema;
}

class ReceiptItemSpliteration extends _ReceiptItemSpliteration
   with RealmEntity, RealmObjectBase, RealmObject {
 ReceiptItemSpliteration(
   String participantId,
   double partSize,
   double percentage,
 ) {
   RealmObjectBase.set(this, 'participantId', participantId);
   RealmObjectBase.set(this, 'partSize', partSize);
   RealmObjectBase.set(this, 'percentage', percentage);
 }

 ReceiptItemSpliteration._();

 @override
 String get participantId =>
     RealmObjectBase.get<String>(this, 'participantId') as String;
 @override
 set participantId(String value) =>
     RealmObjectBase.set(this, 'participantId', value);

 @override
 double get partSize =>
     RealmObjectBase.get<double>(this, 'partSize') as double;
 @override
 set partSize(double value) => RealmObjectBase.set(this, 'partSize', value);

 @override
 double get percentage =>
     RealmObjectBase.get<double>(this, 'percentage') as double;
 @override
 set percentage(double value) =>
     RealmObjectBase.set(this, 'percentage', value);

 @override
 Stream<RealmObjectChanges<ReceiptItemSpliteration>> get changes =>
     RealmObjectBase.getChanges<ReceiptItemSpliteration>(this);

 @override
 Stream<RealmObjectChanges<ReceiptItemSpliteration>> changesFor([
   List<String>? keyPaths,
 ]) => RealmObjectBase.getChangesFor<ReceiptItemSpliteration>(this, keyPaths);

 @override
 ReceiptItemSpliteration freeze() =>
     RealmObjectBase.freezeObject<ReceiptItemSpliteration>(this);
```

```
  EJsonValue toEJson() {
    return <String, dynamic>{
      'participantId': participantId.toEJson(),
      'partSize': partSize.toEJson(),
      'percentage': percentage.toEJson(),
    };
  }

  static EJsonValue _toEJson(ReceiptItemSpliteration value) => value.toEJson();
  static ReceiptItemSpliteration _fromEJson(EJsonValue ejson) {
    if (ejson is! Map<String, dynamic>) return raiseInvalidEJson(ejson);
    return switch (ejson) {
      {
        'participantId': EJsonValue participantId,
        'partSize': EJsonValue partSize,
        'percentage': EJsonValue percentage,
      } =>
        ReceiptItemSpliteration(
          fromEJson(participantId),
          fromEJson(partSize),
          fromEJson(percentage),
        ),
      _ => raiseInvalidEJson(ejson),
    };
  }

  static final schema = () {
    RealmObjectBase.registerFactory(ReceiptItemSpliteration._);
    register(_toEJson, _fromEJson);
    return const SchemaObject(
      ObjectType.realmObject,
      ReceiptItemSpliteration,
      'ReceiptItemSpliteration',
      [
        SchemaProperty('participantId', RealmPropertyType.string),
        SchemaProperty('partSize', RealmPropertyType.double),
        SchemaProperty('percentage', RealmPropertyType.double),
      ],
    );
  }();

  @override
  SchemaObject get objectSchema => RealmObjectBase.getSchema(this) ?? schema;
}

class ParticipantEntity extends _ParticipantEntity
    with RealmEntity, RealmObjectBase, RealmObject {
  ParticipantEntity(String nickname, String username) {
    RealmObjectBase.set(this, 'nickname', nickname);
    RealmObjectBase.set(this, 'username', username);
  }

  ParticipantEntity._();

  @override
```

```
String get nickname =>
    RealmObjectBase.get<String>(this, 'nickname') as String;
@override
set nickname(String value) => RealmObjectBase.set(this, 'nickname', value);

@override
String get username =>
    RealmObjectBase.get<String>(this, 'username') as String;
@override
set username(String value) => RealmObjectBase.set(this, 'username', value);

@override
Stream<RealmObjectChanges<ParticipantEntity>> get changes =>
    RealmObjectBase.getChanges<ParticipantEntity>(this);

@override
Stream<RealmObjectChanges<ParticipantEntity>> changesFor([
  List<String>? keyPaths,
]) => RealmObjectBase.getChangesFor<ParticipantEntity>(this, keyPaths);

@override
ParticipantEntity freeze() =>
    RealmObjectBase.freezeObject<ParticipantEntity>(this);

EJsonValue toEJson() {
  return <String, dynamic>{
    'nickname': nickname.toEJson(),
    'username': username.toEJson(),
  };
}

static EJsonValue _toEJson(ParticipantEntity value) => value.toEJson();
static ParticipantEntity _fromEJson(EJsonValue ejson) {
  if (ejson is! Map<String, dynamic>) return raiseInvalidEJson(ejson);
  return switch (ejson) {
    {'nickname': EJsonValue nickname, 'username': EJsonValue username} =>
      ParticipantEntity(fromEJson(nickname), fromEJson(username)),
    _ => raiseInvalidEJson(ejson),
  };
}

static final schema = () {
  RealmObjectBase.registerFactory(ParticipantEntity._);
  register(_toEJson, _fromEJson);
  return const SchemaObject(
    ObjectType.realmObject,
    ParticipantEntity,
    'ParticipantEntity',
    [
      SchemaProperty('nickname', RealmPropertyType.string),
      SchemaProperty('username', RealmPropertyType.string),
    ],
  );
}();
```

```
  @override
  SchemaObject get objectSchema => RealmObjectBase.getSchema(this) ?? schema;
}

class SplitrApplicationData extends _SplitrApplicationData
    with RealmEntity, RealmObjectBase, RealmObject {
  SplitrApplicationData({
    Iterable<ReceiptEntity> receipts = const [],
    Iterable<ParticipantEntity> savedParticipants = const [],
  }) {
    RealmObjectBase.set<RealmList<ReceiptEntity>>(
      this,
      'receipts',
      RealmList<ReceiptEntity>(receipts),
    );
    RealmObjectBase.set<RealmList<ParticipantEntity>>(
      this,
      'savedParticipants',
      RealmList<ParticipantEntity>(savedParticipants),
    );
  }

  SplitrApplicationData._();

  @override
  RealmList<ReceiptEntity> get receipts =>
      RealmObjectBase.get<ReceiptEntity>(this, 'receipts')
          as RealmList<ReceiptEntity>;
  @override
  set receipts(covariant RealmList<ReceiptEntity> value) =>
      throw RealmUnsupportedSetError();

  @override
  RealmList<ParticipantEntity> get savedParticipants =>
      RealmObjectBase.get<ParticipantEntity>(this, 'savedParticipants')
          as RealmList<ParticipantEntity>;
  @override
  set savedParticipants(covariant RealmList<ParticipantEntity> value) =>
      throw RealmUnsupportedSetError();

  @override
  Stream<RealmObjectChanges<SplitrApplicationData>> get changes =>
      RealmObjectBase.getChanges<SplitrApplicationData>(this);

  @override
  Stream<RealmObjectChanges<SplitrApplicationData>> changesFor([
    List<String>? keyPaths,
  ]) => RealmObjectBase.getChangesFor<SplitrApplicationData>(this, keyPaths);

  @override
  SplitrApplicationData freeze() =>
      RealmObjectBase.freezeObject<SplitrApplicationData>(this);

  EJsonValue toEJson() {
    return <String, dynamic>{
```

```
    'receipts': receipts.toEJson(),
    'savedParticipants': savedParticipants.toEJson(),
  };
 }

 static EJsonValue _toEJson(SplitrApplicationData value) => value.toEJson();
 static SplitrApplicationData _fromEJson(EJsonValue ejson) {
  if (ejson is! Map<String, dynamic>) return raiseInvalidEJson(ejson);
  return SplitrApplicationData(
   receipts: fromEJson(ejson['receipts']),
   savedParticipants: fromEJson(ejson['savedParticipants']),
  );
 }

 static final schema = () {
  RealmObjectBase.registerFactory(SplitrApplicationData._);
  register(_toEJson, _fromEJson);
  return const SchemaObject(
   ObjectType.realmObject,
   SplitrApplicationData,
   'SplitrApplicationData',
   [
    SchemaProperty(
     'receipts',
     RealmPropertyType.object,
     linkTarget: 'ReceiptEntity',
     collectionType: RealmCollectionType.list,
    ),
    SchemaProperty(
     'savedParticipants',
     RealmPropertyType.object,
     linkTarget: 'ParticipantEntity',
     collectionType: RealmCollectionType.list,
    ),
   ],
  );
 }();

 @override
 SchemaObject get objectSchema => RealmObjectBase.getSchema(this) ?? schema;
}
```