Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня
Бакалавр

(назва освітнього ступеня)

на тему: **Розробка мобільного додатку для оцінки трудомісткості задач у проєктах**

Виконав(ла): студент(ка) 4 курсу, групи СПс-41
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Стасюк С. В.
(підпис) (прізвище та ініціали)

Керівник Мудрик І. Я.
(підпис) (прізвище та ініціали)

Нормоконтроль Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри Петрик М.Р.
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____________ _______________
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Стасюку Сергію Вадимовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка мобільного додатку для оцінки трудомісткості задач у проєктах

Керівник роботи Мудрик Іван Ярославович, доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» ___________ 20___ року № ___________

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Предметна область, завдання, вимоги та специфікація, програмне рішення, методичні вказівки

4. Зміст роботи (перелік питань, які потрібно розробити)

Вступна частина

Аналіз вимог та огляд предметної області

Проєктування та конструювання

Тестування

Безпека життєдіяльності, основи охорони праці

Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

1. Тема роботи. 2. Актуальність, мета, задачі дослідження.

3. Аналіз існуючих подібних систем. 4. Функціональні та нефункціональні вимоги.

6. Варіанти використання. 7. Розробка додатку. 8. Тестування.

9. Висновки. 10. Слайди презентації.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності,	Мариненко С. Ю. канд. техн. наук,		
основи охорони праці	доцент		
Нормоконтроль	Стоянов Ю.М. доц. каф. ПІ		

7. Дата видачі завдання ______________________________

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	6.04 – 14.04	
2	Аналіз завдання	15.04 – 20.04	
3	Аналіз вимог та вибір технологій	21.04 – 10.05	
4	Проєктування архітектури та бази даних	11.05 – 20.05	
5	Розробка та тестування застосунку	21.05 – 5.06	
6	Опис ергономічних проблем безпеки життєдіяльності	6.06 – 8.06	
7	Вимоги до режимів праці і відпочинку при роботі з ВДТ	9.06 – 11.06	
8	Оформлення пояснювальної записки	12.06 – 15.06	
9	Оформлення графічного та презентаційного матеріалу	16.06 – 18.06	
10	Попередній захист	19.06 – 20.06	
11	Захист		

Студент ____________ Стасюк С.В.
(підпис) (прізвище та ініціали)

Керівник роботи ____________ Мудрик І.Я
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота бакалавра, виконав Стасюк Сергій Вадимович, студент групи СПс-41 Тернопільського національного технічного університету імені Івана Пулюя, на тему «Розробка мобільного додатку для оцінки трудомісткості задач у проєктах». Робота має обсяг 62 сторінки, включає 6 рисунків, 11 таблиць, 1 додаток та бібліографію з 32 джерел.

Ключові слова: мобільний застосунок, метод PERT, оцінювання трудовитрат, Flutter, управління проєктами, офлайн-застосунок, Clean Architecture.

Кваліфікаційна робота присвячена розробці мобільного застосунку EstiMate для платформи Android, що реалізує метод PERT (Program Evaluation and Review Technique) для структурованого оцінювання трудовитрат у розрізі проєктів та завдань.

У першому розділі проведено аналіз існуючих конкурентних рішень у сфері управління завданнями та оцінювання трудовитрат, сформульовано функціональні та нефункціональні вимоги до застосунку, обґрунтовано вибір технологічного стеку та архітектурного підходу.

У другому розділі виконано проєктування системи: побудовано діаграму варіантів використання, обрано та обґрунтовано архітектурну модель Feature-based Clean Architecture, розроблено UML-діаграми ієрархії класів та послідовності взаємодії компонентів, спроєктовано схему бази даних.

У третьому розділі описано конструювання застосунку: реалізацію шару даних на основі Drift/SQLite, управління станом через flutter_bloc/Cubit, створення користувацького інтерфейсу відповідно до специфікації Material Design 3, а також проведення модульного та автоматизованого тестування.

Об'єкт дослідження: процес оцінювання трудовитрат на розробку програмного забезпечення методом PERT.

Предмет дослідження: мобільний застосунок для автоматизації PERT-оцінювання завдань у межах проєктів.

ABSTRACT

Bachelor's qualification work completed by Stasiuk Serhii Vadymovych, student of group SPs-41 of Ternopil Ivan Puluj National Technical University, on the topic "Development of a Mobile Application for Task Effort Estimation in Projects". The work consists of 60 pages, includes 6 figures, 11 tables, 1 appendices and a bibliography of 32 references.

Keywords: mobile application, PERT method, effort estimation, Flutter, project management, offline application, Clean Architecture.

The qualification work is devoted to the development of the EstiMate mobile application for the Android platform, implementing the PERT (Program Evaluation and Review Technique) method for structured effort estimation within projects and tasks.

The first chapter analyses existing competitive solutions in the field of task management and effort estimation, formulates functional and non-functional requirements for the application, and justifies the choice of technology stack and architectural approach.

The second chapter covers system design: a use case diagram is constructed, the Feature-based Clean Architecture model is selected and justified, UML class hierarchy and sequence diagrams are developed, and the database schema is designed.

The third chapter describes the application construction: implementation of the data layer based on Drift/SQLite, state management via flutter_bloc/Cubit, creation of the user interface in accordance with the Material Design 3 specification, and the execution of unit and widget testing.

Object of research: the process of software development effort estimation using the PERT method.

Subject of research: a mobile application for automating PERT-based task estimation within projects.

ЗМІСТ

ВСТУП ... 6
1 АНАЛІЗ ВИМОГ ДО МОБІЛЬНОГО ЗАСТОСУНКУ ... 8
1.1 Огляд конкурентів ... 8
1.2 Визначення вимог до проєкту ... 11
1.3 Визначення технологій розробки, інструментів, методології та архітектури системи ... 13
2 ПРОЄКТУВАННЯ СИСТЕМИ ... 17
2.1 Вибір інструментів проєктування ... 17
2.2 Моделювання варіантів використання системи на основі вимог ... 18
2.3 Архітектурне проєктування системи ... 20
2.3.1 Вибір архітектурної моделі системи ... 20
2.3.2 Побудова UML-діаграм ієрархії класів ... 22
2.4 Детальне проєктування класів підсистем ... 24
2.5 Проєктування сценаріїв ВВ на основі UML-діаграм послідовності ... 27
3 КОНСТРУЮВАННЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ ... 30
3.1 Базова структура, налаштування проєкту та опис моделей і фасадів системи ... 30
3.2 Створення та реалізація користувацького інтерфейсу ... 34
3.3 Тестування системи ... 46
3.3.1 Модульне тестування ... 46
3.3.2 Автоматизоване тестування ... 48
4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ ... 51
4.1 Безпека життєдіяльності. Ергономічні проблеми безпеки життєдіяльності. 51
4.2 Основи охорони праці. Вимоги до режимів праці і відпочинку при роботі з ВДТ ... 53
ВИСНОВКИ ... 55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 57

ВСТУП

Ефективне управління часом є однією з ключових компетенцій у сучасній розробці програмного забезпечення та проєктному менеджменті. Однією з найбільш поширених проблем у цій сфері є неточне оцінювання трудовитрат, що призводить до зривів термінів, перевищення бюджетів та зниження якості кінцевого продукту. За даними дослідження Standish Group Chaos Report, понад 50% IT-проєктів виходять за рамки запланованих термінів або бюджетів, а основною причиною цього є саме помилки у початковому плануванні.

Для вирішення цієї проблеми широко застосовується метод PERT (Program Evaluation and Review Technique) – статистичний підхід до оцінювання тривалості завдань, що враховує три сценарії: оптимістичний, реалістичний та песимістичний. Формула очікуваного часу (O + 4R + P) / 6 дозволяє отримати зважену оцінку, яка є більш точною порівняно зі звичайним одноточковим прогнозом.

Незважаючи на доведену ефективність методу PERT, більшість існуючих мобільних інструментів для управління завданнями або не підтримують цей метод взагалі, або реалізують його лише частково. Відсутній зручний мобільний інструмент, що поєднує PERT-оцінювання, управління проєктами та аналітику в єдиному рішенні з підтримкою офлайн-роботи.

Актуальність теми зумовлена потребою у спеціалізованому мобільному застосунку для PERT-оцінювання завдань, який забезпечував би структуровану роботу з проєктами, наочну аналітику та зручний користувацький інтерфейс без залежності від мережевого з'єднання.

Мета роботи – розробити мобільний застосунок EstiMate для платформи Android з використанням фреймворку Flutter, що реалізує метод PERT для оцінювання трудовитрат у розрізі проєктів та завдань.

Для досягнення поставленої мети визначені такі завдання:

1. Провести аналіз існуючих рішень у сфері управління завданнями та оцінювання трудовитрат.
2. Сформувати функціональні та нефункціональні вимоги до застосунку.

3. Обрати технологічний стек та архітектурний підхід до розробки.
4. Спроєктувати архітектуру системи, бази даних та користувацький інтерфейс.
5. Реалізувати застосунок відповідно до спроєктованої архітектури.
6. Провести тестування розробленого програмного забезпечення.

Об'єкт дослідження – процес оцінювання трудовитрат на розробку програмного забезпечення методом PERT.

Предмет дослідження – мобільний застосунок для автоматизації PERT-оцінювання завдань у межах проєктів.

Методи дослідження: аналіз предметної області та конкурентних рішень; об'єктно-орієнтоване проєктування; UML-моделювання; методи тестування програмного забезпечення (модульне, інтеграційне); статистичний метод PERT для оцінювання часових витрат.

Практична цінність роботи полягає у створенні готового до використання мобільного застосунку, який дозволяє командам та індивідуальним розробникам структуровано вести проєкти, оцінювати завдання методом PERT та відстежувати прогрес через аналітичні інструменти. Застосунок працює повністю офлайн, що забезпечує незалежність від мережевого з'єднання.

У першому розділі проведено аналіз вимог до мобільного застосунку, а саме розглянуто існуючі конкурентні рішення, сформульовано функціональні та нефункціональні вимоги, обрано технологічний стек та визначено архітектурний підхід.

У другому розділі виконано проєктування системи, тобто побудовано діаграми варіантів використання, обрано архітектурну модель, розроблено UML-діаграми класів та послідовностей, спроєктовано структуру бази даних.

У третьому розділі описано процес конструювання застосунку, його реалізацію базової структури проєкту, шарів даних та презентації, створення користувацького інтерфейсу, а також проведення модульного та автоматизованого тестування.

1 АНАЛІЗ ВИМОГ ДО МОБІЛЬНОГО ЗАСТОСУНКУ

У цьому розділі проведено комплексний аналіз предметної області, визначено вимоги до розроблюваного застосунку та обрано технологічний стек. Аналіз починається з огляду конкурентних рішень, після чого формулюються функціональні та нефункціональні вимоги, обирається архітектурний підхід і технологічний стек.

1.1 Огляд конкурентів

Перед початком розробки було проведено аналіз існуючих програмних рішень у сфері управління завданнями та оцінювання трудовитрат. Метою аналізу є виявлення переваг і недоліків наявних інструментів, а також визначення незайнятої ніші, яку може заповнити розроблюваний застосунок. Аналіз охоплює як повнофункціональні корпоративні платформи, так і вузькоспеціалізовані утиліти, що орієнтовані виключно на обчислення PERT.

На ринку можна виділити два основних типи конкурентів: повнофункціональні системи управління проєктами та спеціалізовані PERT-калькулятори. Повнофункціональні системи орієнтовані на командну роботу та широкий набір інструментів планування, тоді як калькулятори вирішують лише одне вузьке завдання – обчислення очікуваного часу без контексту управління проєктом.

Microsoft Project є одним із найбільш комплексних інструментів управління проєктами. Він підтримує метод PERT, CPM (Critical Path Method), діаграми Ганта та широкий набір аналітичних можливостей. Застосунок дозволяє будувати ієрархічні структури робіт (WBS), призначати ресурси з урахуванням їх завантаженості, а також відстежувати відхилення між плановими та фактичними термінами. Проте застосунок орієнтований виключно на десктопні платформи (Windows), має складний інтерфейс, що вимагає спеціального навчання, а також є платним корпоративним рішенням із вартістю ліцензії від 10 доларів США на

місяць. Відсутність мобільної версії робить його непридатним для індивідуального використання або невеликих команд [3].

Jira (Atlassian) – популярна хмарна система управління проєктами, широко використовувана в IT-командах [4]. Підтримує оцінювання завдань у вигляді story points (Agile-підхід), дошки Kanban, спринти та ретроспективи. Проте метод PERT не реалізований навіть у вигляді плагіну. Застосунок потребує постійного підключення до мережі, має складну систему налаштування з десятками параметрів та розрахований переважно на командне використання з платними тарифними планами для розширених функцій. Для індивідуального розробника або малої команди без DevOps-процесів Jira є надмірно складним і фінансово недоступним рішенням.

Asana – хмарний інструмент для управління завданнями та проєктами з зручним інтерфейсом [5]. Підтримує дедлайни, пріоритети, статуси завдань і аналітику у вигляді графіків прогресу та звітів. Проте PERT-оцінювання відсутнє – є лише можливість вказати очікуваний час як звичайне текстове поле без автоматичного розрахунку. Функціональність суттєво обмежена у безкоштовному тарифному плані, а повна залежність від хмарного сервісу унеможливлює офлайн-роботу, що є критичним недоліком для розробників, які працюють у місцях з нестабільним інтернет-підключенням.

Todoist – один із найпопулярніших мобільних менеджерів завдань із простим інтерфейсом [6]. Підтримує пріоритети (чотири рівні), дедлайни, мітки та повторювані завдання. Однак Todoist не має функціональності для оцінювання трудовитрат – відсутні поля для введення часових оцінок, і аналітика обмежена лише відображенням продуктивності у вигляді кількості виконаних завдань. Застосунок не підтримує поняття проєкту у контексті PERT-планування, а платна підписка Todoist Pro необхідна для більшості корисних функцій.

Мобільні PERT-калькулятори (різні застосунки у Google Play, як-от «PERT Calculator», «Three Point Estimator») реалізують безпосередньо формулу PERT для розрахунку очікуваного часу, проте є виключно утилітарними – не мають системи проєктів, управління завданнями, відстеження статусів чи аналітики. Користувач

вводить три числа, отримує результат розрахунку, і на цьому функціональність вичерпується. Такі застосунки не зберігають жодної історії оцінок, не дозволяють групувати завдання за проєктами і не надають ніякої аналітики щодо точності попередніх прогнозів. Фактично це лише калькулятори без контексту, що не вирішують задачу управління проєктом в цілому.

Порівняльний аналіз розглянутих рішень наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз конкурентних рішень

Критерій	**Microsoft Project**	**Jira**	**Asana**	**Todoist**	**PERT-калькулятори**	**EstiMate**
PERT-оцінювання	+	–	–	–	+	+
Управління проєктами	+	+	+	–	–	+
Управління завданнями	+	+	+	+	–	+
Аналітика	+	+	+	частково	–	+
Офлайн-режим	+	–	–	частково	+	+
Мобільна платформа	–	+	+	+	+	+
Безкоштовне	–	частково	частково	частково	+	+
Простота	–	–	+	+	+	+

З таблиці 1.1 видно, що жоден із розглянутих конкурентів не поєднує всі ключові характеристики: PERT-оцінювання, управління проєктами та завданнями, аналітику, офлайн-режим і мобільну платформу в одному безкоштовному рішенні. Microsoft Project підтримує PERT, але не є мобільним рішенням і є платним. Jira та Asana забезпечують управління проєктами, але вимагають постійного підключення до мережі та не підтримують PERT. PERT-калькулятори є мобільними та офлайн, але позбавлені будь-якого контексту управління. Саме це поєднання є головною конкурентною перевагою розроблюваного застосунку EstiMate.

1.2 Визначення вимог до проєкту

На основі проведеного аналізу конкурентів та потреб цільової аудиторії (розробники програмного забезпечення, проєктні менеджери, студенти) сформульовано функціональні та нефункціональні вимоги до застосунку. Вимоги збиралися шляхом аналізу недоліків існуючих рішень та типових сценаріїв використання інструментів планування в IT-командах.

Функціональні вимоги визначають, що саме повинна робити система:

1. Управління проєктами – створення, редагування та видалення проєктів. Кожен проєкт має назву, необов'язковий опис і дедлайн. Головний екран відображає список проєктів з поточним прогресом виконання.
2. Архівування проєктів – завершені або призупинені проєкти переводяться до архіву без видалення даних з можливістю відновлення через перемикач на головному екрані.
3. Управління завданнями – створення, редагування та видалення завдань у межах проєкту. Кожне завдання має назву, опис, пріоритет і статус. Видалення проєкту автоматично видаляє всі пов'язані завдання.
4. PERT-оцінювання – для кожного завдання вводяться оптимістична, реалістична та песимістична оцінки часу. Очікуваний час розраховується автоматично: $T = (O + 4R + P) / 6$. Оптимістична та песимістична оцінки є необов'язковими.
5. Пріоритизація завдань – три рівні пріоритету (низький, середній, високий), що відображаються візуальним індикатором і використовуються як параметр сортування.
6. Відстеження статусу – три статуси завдання (заплановане, у процесі, завершене). Відсоток виконання відображається у картці проєкту та детальному перегляді.
7. Дедлайни та сповіщення – push-сповіщення за 24 години до дедлайну та в момент його настання з урахуванням часового поясу пристрою і автоматичним перплануванням після перезавантаження.

8. Аналітика – відсоток виконання, розподіл завдань за пріоритетами та статусами, загальний очікуваний час, оптимістичний і песимістичний прогнози завершення проєкту.
9. Налаштування – вибір теми (світла, темна, системна), мови інтерфейсу (українська, англійська) та одиниць часу (години, дні). Налаштування зберігаються між сесіями та застосовуються без перезапуску.
10. Часові пресети – набір швидких значень (0,5, 1, 2, 4, 8 годин) для введення оцінок, що скорочує кількість ручних операцій.

Нефункціональні вимоги визначають якісні характеристики системи та обмеження на її реалізацію:

1. Офлайн-режим – повна функціональність без підключення до мережі. Усі дані зберігаються локально у реляційній базі даних SQLite, жодна функція не залежить від інтернет-з'єднання.
2. Продуктивність – час відгуку на дії користувача (навігація, збереження, розрахунок PERT) не перевищує 1 секунди. Анімації та переходи між екранами виконуються зі стабільною частотою 60 кадрів на секунду.
3. Сумісність – підтримка Android версії 6.0 (API 23) та вище, що охоплює понад 95% активних Android-пристроїв за статистикою Google Play станом на 2024 рік.
4. Зручність використання – інтерфейс відповідає специфікації Material Design 3. Всі ключові дії доступні не більш ніж за три натискання від головного екрану.
5. Локалізація – підтримка української та англійської мов з миттєвим перемиканням у налаштуваннях без перезапуску застосунку.
6. Надійність – дані не втрачаються при закритті або перезавантаженні пристрою. Всі зміни зберігаються транзакційно, міграції схеми бази даних гарантують збереження даних при оновленні застосунку.

Сукупність визначених вимог формує повну специфікацію застосунку, достатню для переходу до етапу проєктування архітектури та вибору технологічного стеку.

1.3 Визначення технологій розробки, інструментів, методології та архітектури системи

Вибір технологічного стеку здійснювався з урахуванням сформованих вимог, насамперед вимоги офлайн-роботи, підтримки Android-платформи, швидкості розробки та якості кінцевого продукту. Для кожного компонента стеку розглядалися альтернативи та обґрунтовувався кінцевий вибір на основі чітких технічних критеріїв.

Основним фреймворком для розробки обрано Flutter версії 3.5+ від компанії Google. Flutter є крос-платформним фреймворком, що дозволяє створювати нативні мобільні застосунки з єдиної кодової бази [7]. Він використовує власний рушій рендерингу (Skia/Impeller), що забезпечує стабільно високу продуктивність на рівні 60/120 FPS, незалежно від платформи. Мова програмування Dart є статично типізованою з підтримкою null safety, що зменшує кількість помилок на етапі компіляції та підвищує надійність коду. Для даного проєкту Flutter обрано завдяки: підтримці Material Design 3 з коробки без зовнішніх залежностей, розвиненій екосистемі пакетів (pub.dev налічує понад 40 000 пакетів), зрілості інструментів налагодження (Flutter DevTools) та можливості потенційного розширення на iOS у майбутньому без переписування бізнес-логіки.

Альтернативами Flutter розглядалися нативна розробка на Kotlin (Android SDK) та React Native. Нативна розробка на Kotlin забезпечує найвищу продуктивність, однак обмежує проєкт лише платформою Android і збільшує трудовитрати на реалізацію UI-компонентів. React Native є популярним крос-платформним фреймворком, але поступається Flutter за продуктивністю UI-рендерингу через JavaScript-міст між логікою та нативними компонентами. Враховуючи ці фактори, Flutter є оптимальним вибором для даного проєкту.

Для локального зберігання даних використано бібліотеку Drift версії 2.22.1 – реактивний ORM для SQLite на Flutter/Dart. Drift забезпечує типобезпечні SQL-запити через Dart API, автоматичну генерацію коду (DAO, таблиці) через build_runner, реактивні потоки (Stream<List<T>>) для автоматичного оновлення UI

при зміні даних, підтримку версійних міграцій схеми бази даних [8]. Порівняльний аналіз бібліотек для локального зберігання даних наведено у таблиці 1.2.

Таблиця 1.2 – Порівняльний аналіз бібліотек для локального зберігання даних

Критерій	**sqflite**	**Hive**	**Drift**
Тип сховища	SQLite (реляційне)	NoSQL (key-value)	SQLite (реляційне)
Типобезпека запитів	–	+	+
Реактивні потоки	–	частково	+
Міграції схеми	вручну	обмежено	+ (версійні)
Зовнішні ключі / JOIN	+	–	+
Генерація коду (DAO)	–	–	+

З таблиці 1.2 видно, що Drift є єдиною бібліотекою, яка одночасно забезпечує типобезпечні запити, реактивні потоки, повноцінну підтримку реляційної моделі (зовнішні ключі, JOIN) та автоматичну генерацію DAO. Саме ці властивості є критичними для EstiMate, оскільки реляційний зв'язок між таблицями Projects і Tasks вимагає підтримки зовнішніх ключів із каскадним видаленням, а реактивні потоки Stream<List<T>> забезпечують автоматичне оновлення UI при будь-якій зміні даних. Додатковою перевагою є версійна система міграцій MigrationStrategy, що гарантує збереження даних користувача при оновленні застосунку.

Для управління станом застосунку використано бібліотеку flutter_bloc версії 9.0.0 із застосуванням патерну Cubit [9]. Cubit є спрощеною версією BLoC (Business Logic Component), що не потребує визначення окремих класів подій. Перевагами є чітке розділення бізнес-логіки та представлення, immutable-стани на базі Equatable, зручне тестування логіки незалежно від UI, підтримка реактивного зв'язку з потоками даних Drift. Порівняльний аналіз підходів до управління станом наведено у таблиці 1.3.

Таблиця 1.3 – Порівняльний аналіз підходів до управління станом у Flutter

Критерій	**setState / ChangeNotifier**	**BLoC**	**Cubit (flutter_bloc)**
Розділення логіки від UI	Слабке	Відмінне	Відмінне
Кількість boilerplate-коду	Мінімальна	Висока	Середня
Тестованість	Низька	Висока	Висока
Складність впровадження	Низька	Висока	Середня
Підтримка Equatable	–	+	+
Підходить для проєкту	–	надлишково	+

З таблиці 1.3 видно, що Cubit є оптимальним вибором для даного проєкту оскільки він забезпечує повне розділення логіки від UI та високу тестованість, однак вимагає суттєво менше boilerplate-коду порівняно з повноцінним BLoC. Застосування setState або ChangeNotifier є недостатнім з огляду на необхідність підписки на реактивні потоки Drift та складну логіку фільтрації й сортування завдань.

Для впровадження залежностей (Dependency Injection) використано бібліотеку GetIt версії 8.0.3 – service locator для Dart/Flutter [10]. GetIt забезпечує централізовану реєстрацію та надання залежностей (база даних, репозиторії, кубіти, сервіси) без необхідності передавати їх через конструктори по всьому дереву віджетів. Реєстрація залежностей виконується одноразово в функції setupDependencies() при запуску застосунку, після чого будь-яка залежність доступна через виклик sl<T>(), де T – тип необхідного об'єкта. Такий підхід спрощує тестування, оскільки mock-об'єкти можуть бути зареєстровані замість реальних реалізацій.

Навігацію між екранами реалізовано з використанням GoRouter версії 14.6.3 – офіційно рекомендованого рішення для декларативної навігації у Flutter [11]. GoRouter забезпечує маршрутизацію на основі URL-шляхів, передачу параметрів

через path parameters та підтримку вкладеної навігації. Декларативний підхід дозволяє описати всю навігаційну структуру застосунку в одному файлі router.dart, що спрощує розуміння архітектури та полегшує додавання нових маршрутів.

Для реалізації push-сповіщень про дедлайни проєктів використано бібліотеку flutter_local_notifications версії 17.2.4 у поєднанні з timezone та flutter_timezone для коректного планування сповіщень з урахуванням часового поясу пристрою [12]. Бібліотека підтримує точне планування сповіщень (exact alarm) на Android 12 та вище, що є обов'язковою умовою для надійного нагадування про дедлайни.

Архітектура системи. Застосунок побудовано за принципами Feature-based Clean Architecture – підходу, що поєднує ідеї чистої архітектури Роберта Мартіна з організацією коду за функціональними модулями [13]. Відмова від явного шару use cases обґрунтована простотою бізнес-логіки застосунку де єдиний нетривіальний розрахунок (формула PERT) реалізований у невеликій утиліті PertCalculator. Натомість система організована у два основних шари – даних (data) та представлення (presentation) [14].

Кодова база організована наступним чином, що шар даних (data/) містить визначення таблиць бази даних, DAO-об'єкти та репозиторії, що надають реактивні потоки, шар представлення (features/) організований за функціональними модулями – projects, tasks, analytics, settings, кожен із яких містить власні Cubit та екрани та спільні компоненти (shared/core/) включають конфігурацію DI, маршрутизацію, теми, константи та утиліти. Ключовий принцип організації – однонаправленість залежностей коли шар представлення може залежати від шару даних, але не навпаки. Шар даних не містить жодних посилань на Flutter-компоненти або UI-логіку, що дозволяє тестувати його ізольовано.

У процесі розробки застосовано ітеративний підхід із поступовим нарощуванням функціональності – від базової структури проєкту до реалізації UI та аналітики. Такий підхід дозволив своєчасно виявляти архітектурні проблеми та коригувати рішення на ранніх етапах. Для контролю версій використано Git із атомарними комітами.

2 ПРОЄКТУВАННЯ СИСТЕМИ

У цьому розділі виконано комплексне проєктування системи EstiMate. На основі визначених у розділі 1 вимог побудовано UML-діаграми, обрано архітектурну модель та спроєктовано структуру бази даних і класів підсистем.

2.1 Вибір інструментів проєктування

Для проєктування системи EstiMate використовувався набір інструментів, орієнтованих на UML-моделювання та документування архітектури. Вибір кожного інструменту обґрунтовувався конкретними технічними вимогами – насамперед можливістю версіонування артефактів у Git та зручністю підтримки діаграм разом із кодом.

Основним інструментом для побудови UML-діаграм обрано PlantUML – текстово-орієнтовану мову опису діаграм [15]. На відміну від графічних редакторів (Lucidchart, draw.io, Microsoft Visio), PlantUML дозволяє описувати діаграми у вигляді простого тексту, що забезпечує версіонування через Git, повторне використання фрагментів та зручне редагування без мишки. Текстовий підхід також виключає проблему «застарілих діаграм» – зміни в коді одразу відображаються у відповідних .puml-файлах в репозиторії. Для рендерингу використовувався онлайн-сервіс PlantText (planttext.com), що підтримує весь набір діаграм стандарту UML 2.5 без необхідності локального встановлення Java-середовища.

Для проєктування схеми бази даних застосовано підхід Schema-as-Code: схема описується безпосередньо через Drift-таблиці у Dart-коді, що одночасно є документацією і реалізацією. Це усуває розрив між ERD-діаграмою та реальною схемою бази даних, типовий для підходів із окремими артефактами.

У ході проєктування побудовано чотири типи UML-діаграм: діаграму варіантів використання (Use Case Diagram) для специфікації функціональних вимог, діаграму пакетів (Package Diagram) для відображення архітектурних шарів

та напрямку залежностей, діаграми класів (Class Diagram) для деталізації структури підсистем, та діаграми послідовності (Sequence Diagram) для опису ключових сценаріїв взаємодії компонентів.

2.2 Моделювання варіантів використання системи на основі вимог

На основі функціональних вимог, визначених у розділі 1, побудовано діаграму варіантів використання. Єдиним актором системи є Користувач, оскільки застосунок не має ролей і не взаємодіє з backend-сервером – уся логіка виконується локально на пристрої.

Варіанти використання згруповано у п'ять пакетів відповідно до функціональних підсистем: управління проєктами, управління завданнями, PERT-оцінювання, аналітика та налаштування. Детальний перелік варіантів використання у розрізі пакетів наведено у таблиці 2.1.

Таблиця 2.1 – Варіанти використання системи EstiMate за пакетами

Пакет / Підсистема	Варіанти використання
Управління проєктами	Створити проєкт; Редагувати проєкт; Видалити проєкт; Архівувати проєкт; Відновити проєкт; Встановити дедлайн
Управління завданнями	Створити завдання; Редагувати завдання; Видалити завдання; Змінити пріоритет; Змінити статус
PERT-оцінювання	Ввести оцінки O/R/P; Отримати PERT-результат; Переглянути живе прев'ю; Застосувати пресет часу
Аналітика	Переглянути прогрес проєкту; Переглянути часові оцінки; Переглянути розподіл за статусами; Переглянути розподіл за пріоритетами
Налаштування	Змінити тему; Змінити мову; Змінити одиниці часу; Отримати push-сповіщення про дедлайн

Між варіантами визначено два типи залежностей [16]. Залежність <<include>> означає обов'язкове включення: «Створити завдання» завжди включає «Ввести оцінки O/R/P», а введення оцінок – «Отримати PERT-результат». Залежність <<extend>> описує необов'язкове розширення: встановлення дедлайну

є розширенням створення проєкту, а планування сповіщення – розширенням встановлення дедлайну.

Загалом система містить 20 варіантів використання, що відповідають 10 функціональним вимогам з підрозділу 1.2. Центральний ланцюжок «Створити завдання, Ввести оцінки O/R/P, Отримати PERT-результат» відображає основну цінність застосунку.

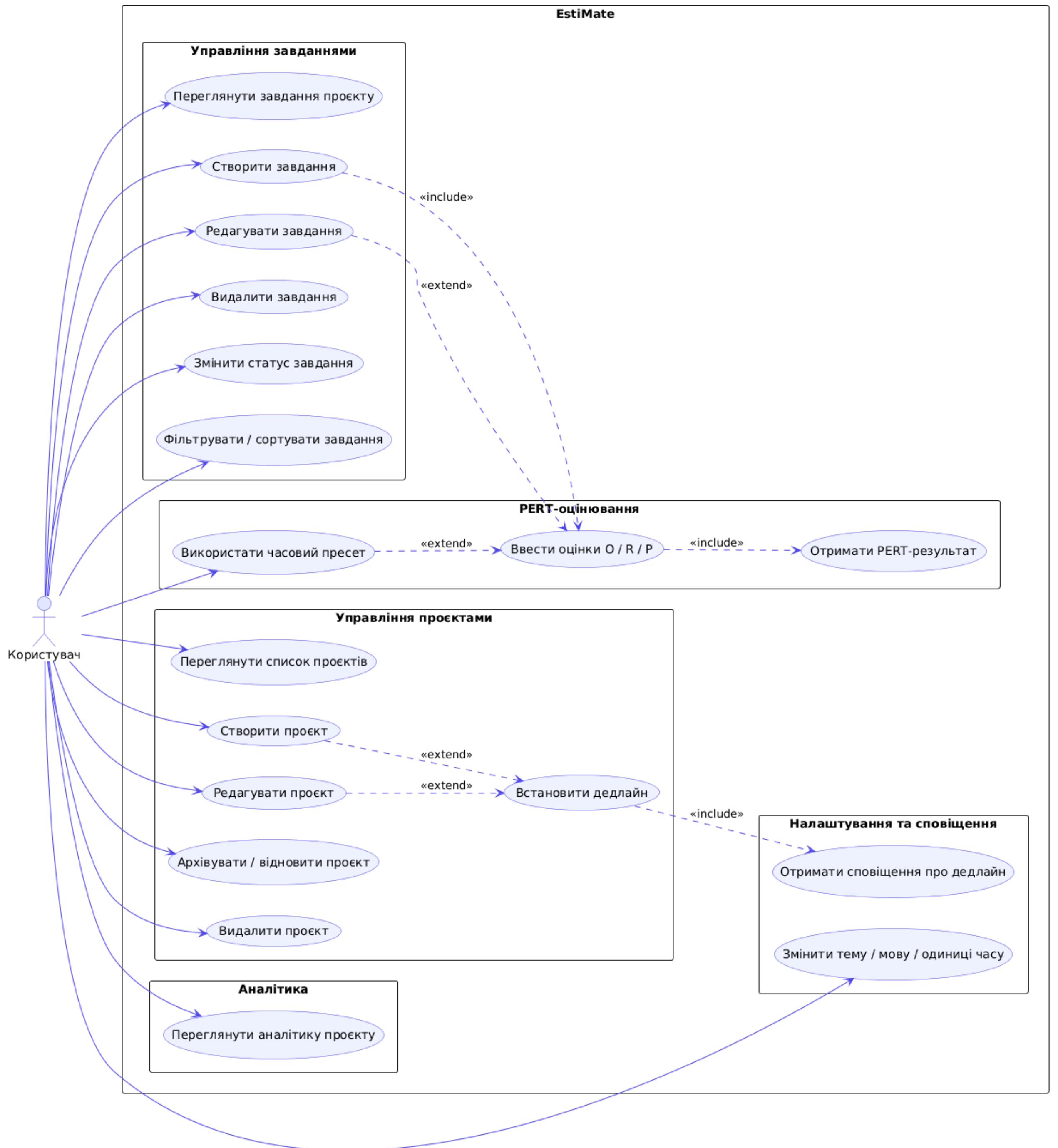


Рисунок 2.1 – Діаграма варіантів використання системи EstiMate

2.3 Архітектурне проєктування системи

Архітектурне проєктування системи охоплює вибір архітектурної моделі та побудову UML-діаграм ієрархії класів. Обидва аспекти тісно пов'язані і архітектурна модель визначає правила організації класів, а діаграми класів унаочнюють ці правила на конкретних прикладах.

2.3.1 Вибір архітектурної моделі системи

Вибір архітектурного підходу є одним із ключових рішень, що визначає масштабованість, тестованість та підтримуваність проєкту. Для EstiMate розглядалися три кандидати:

1. MVC (Model-View-Controller).
2. MVVM (Model-View-ViewModel).
3. Feature-based Clean Architecture.

Порівняльний аналіз підходів наведено у таблиці 2.2.

Таблиця 2.2 – Порівняльний аналіз архітектурних підходів

Критерій	**MVC**	**MVVM**	**Feature-based Clean Architecture**
Розділення відповідальності	Слабке	Добре	Відмінне
Тестованість	Низька	Середня	Висока
Масштабованість	Низька	Середня	Висока
Складність впровадження	Низька	Середня	Середня
Ізоляція модулів	Відсутня	Часткова	Повна
Підходить для Flutter	Частково	Добре	Відмінно

MVC є найпростішим підходом, однак у Flutter він призводить до надмірного розростання View-компонентів, оскільки Controller часто стає змішаним з логікою

відображення. MVVM є кращим варіантом і природно реалізується через ValueNotifier або ChangeNotifier, проте не забезпечує чіткого розмежування між джерелом даних та бізнес-логікою при зростанні проєкту. Як видно з таблиці 2.2, Feature-based Clean Architecture перевершує обидві альтернативи за критеріями тестованості, масштабованості та ізоляції модулів.

Обрана архітектура Feature-based Clean Architecture є адаптацією принципів Роберта Мартіна для Flutter [17]. Класична чиста архітектура передбачає поділ системи на чотири рівні абстракції, а саме сутності, варіанти використання, адаптери інтерфейсів та зовнішні фреймворки. У даному проєкті явний шар Use Cases не вводиться через просту бізнес-логіку, де єдиний нетривіальний розрахунок (формула PERT) реалізований у невеликій утиліті PertCalculator. Натомість система організована у два основних шари даних (data) та представлення (presentation), що є достатнім для поточного масштабу.

Кодова база організована за функціональними модулями (features), а не за технічними шарами. Кожен із модулів projects, tasks, analytics та settings є самодостатнім і містить власні Cubit та екрани. Такий підхід суттєво спрощує навігацію по коду, оскільки всі компоненти однієї функції зосереджені в одній папці, а додавання або видалення модуля не зачіпає інші частини проєкту. Головним архітектурним принципом є однонаправленість залежностей, згідно з яким шар представлення залежить від шару даних, але не навпаки. Шар даних не містить жодних посилань на Flutter-віджети, що дозволяє тестувати репозиторії та DAO ізольовано, передаючи mock-реалізацію через конструктор.

Потік даних у системі є суворо однонаправленим і відповідає ланцюжку SQLite, DAO, Repository, Cubit, Screen. Drift повертає реактивні потоки Stream<List<T>>, на які підписуються кубіти через StreamSubscription у методі _init(). При будь-якій зміні в базі даних Drift автоматично сповіщає всіх підписників, а кубіти емітують новий стан без ручного перезавантаження. Це забезпечує консистентність даних між усіма екранами застосунку: оновлення завдання на одному екрані миттєво відображається в аналітиці та картці проєкту без будь-яких додаткових викликів.

Потік даних у системі є суворо однонаправленим і проходить через послідовність від SQLite, DAO, Repository, Cubit до Screen. Drift повертає реактивні потоки Stream<List<T>>, на які підписуються кубіти через StreamSubscription у методі _init() [18]. При будь-якій зміні в базі даних Drift автоматично сповіщає всіх підписників, а кубіти емітують новий стан без ручного перезавантаження. Такий реактивний підхід гарантує, що UI завжди відображає актуальний стан бази даних без необхідності явних викликів методів refresh() або reload().

Діаграма пакетів архітектури системи EstiMate наведена у додатку А.

features (шар представлення) – верхній блок, що об'єднує чотири функціональні модулі: settings (SettingsScreen, SettingsCubit), projects (ProjectListScreen, ProjectDetailScreen, ProjectsCubit, ProjectDetailCubit), tasks (TaskFormScreen, TaskFormCubit) та analytics (AnalyticsScreen, AnalyticsCubit).

shared / core – лівий блок зі спільними компонентами: PertCalculator, AppRouter (GoRouter), AppTheme, service_locator (GetIt) та L10n/ARB. Пунктирна стрілка з підписом «використовує» вказує на зв'язок із шаром даних.

data (шар даних) – правий блок, що містить три вкладені пакети: models (TaskModel, ProjectModel), repositories (ProjectRepository, TaskRepository) та database (ProjectsDao/TasksDao та AppDatabase на основі Drift/SQLite). Стрілки від шару представлення до репозиторіїв показують напрямок залежностей, а від service_locator – стрілки з підписом «реєструє».

2.3.2 Побудова UML-діаграм ієрархії класів

Ієрархія класів у системі формується навколо двох базових типів, а саме Equatable (для моделей та станів) та Cubit<S> (для кубітів). Такий підхід забезпечує чітке розмежування відповідальностей, де Equatable відповідає за коректне порівняння об'єктів, а Cubit<S> за типобезпечне управління станом.

Усі моделі даних ProjectModel та TaskModel наслідують Equatable. Це забезпечує порівняння об'єктів за значенням полів, а не за посиланням у пам'яті. Така властивість є критичною для BLoC-патерну, коли кубіт емітує стан то Flutter

порівнює новий стан зі старим і перебудовує UI лише якщо вони відрізняються. Без Equatable будь-яке emit() призводило б до повного перебудування дерева віджетів, навіть якщо дані не змінилися, що негативно впливало б на продуктивність. Equatable реалізує порівняння через перевизначення оператора == та методу hashCode на основі переліку полів, зазначених у геттері props. Для ProjectModel до props входять усі поля: id, name, description, deadline, createdAt, isArchived. Це гарантує коректне виявлення будь-якої зміни в даних проєкту.

Стани кубітів також наслідують Equatable. Для кожного кубіту визначено абстрактний базовий клас стану (ProjectsState, ProjectDetailState тощо) та набір конкретних реалізацій у вигляді Initial, Loading, Loaded та Error. Такий підхід дозволяє обробляти різні стани через exhaustive switch у BlocBuilder, де компілятор Dart гарантує, що всі можливі підтипи стану оброблено, а пропущений стан призводить до помилки компіляції. Це суттєво підвищує надійність UI-коду порівняно з перевірками типу через if-else. Стан Loaded є найбагатшим за структурою і містить не лише дані, а й похідні значення. Наприклад ProjectDetailLoaded включає список відфільтрованих завдань та відсоток прогресу, що обчислюються безпосередньо у геттерах стану без звернення до бази даних.

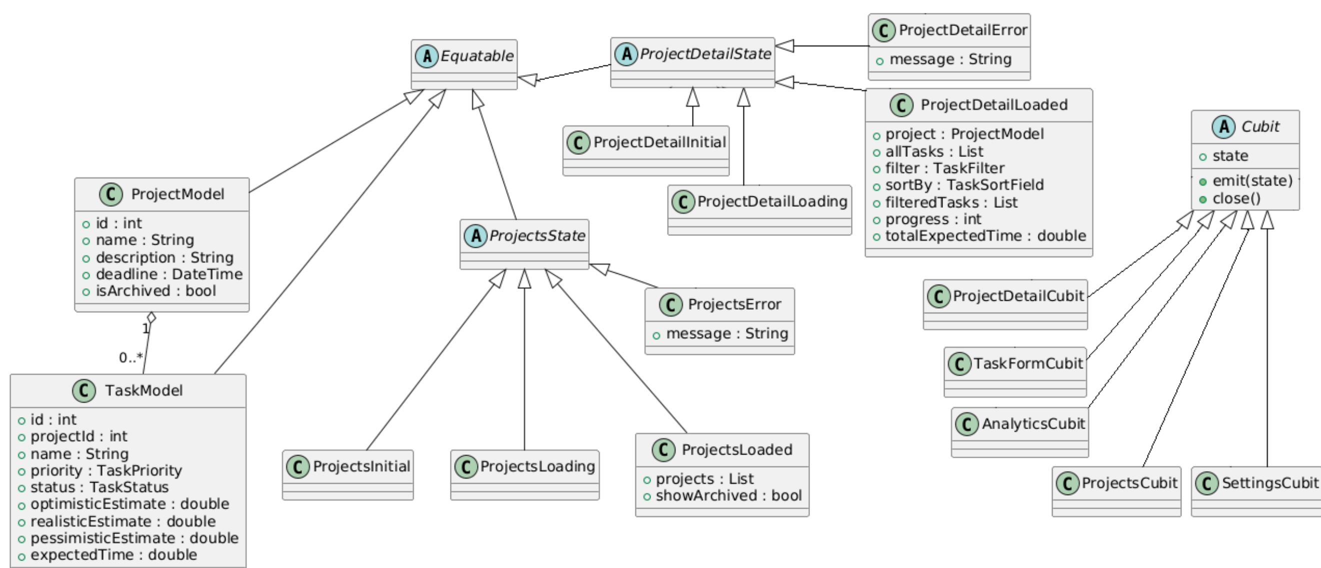


Рисунок 2.3 – Діаграма ієрархії класів системи

Кубіти наслідують Cubit<S>, де S відповідний тип стану. Наприклад, ProjectsCubit наслідує Cubit<ProjectsState> і може емітувати лише підкласи ProjectsState. Це забезпечує типову безпеку на рівні компілятора, тобто неможливо передати стан одного кубіту іншому.

Залежності між кубітами та репозиторіями вводяться через конструктор і реєструються у service_locator.dart засобами GetIt. Впровадження залежностей унеможливлює прямий доступ до бази даних з екранів і спрощує підміну залежностей в тестуванні. Перелік кубітів та їх відповідальностей у таблиці 2.3.

Таблиця 2.3 – Кубіти системи EstiMate та їх відповідальності

Cubit	**Тип стану**	**Відповідальність**
ProjectsCubit	ProjectsState	Список активних і архівованих проєктів, перемикач відображення
ProjectDetailCubit	ProjectDetailState	Деталі проєкту, список завдань, фільтр і сортування
TaskFormCubit	TaskFormState	Форма завдання, живий PERT-прев'ю, збереження
AnalyticsCubit	AnalyticsState	Агрегація статистики: прогрес, розподіл, часові оцінки
SettingsCubit	SettingsState	Тема, мова інтерфейсу, одиниці виміру часу

2.4 Детальне проєктування класів підсистем

Підсистему даних утворюють таблиці бази даних, DAO-класи, репозиторії та моделі. AppDatabase є центральною точкою доступу до SQLite і агрегує ProjectsDao та TasksDao, кожен з яких інкапсулює SQL-запити для своєї таблиці. Клас AppDatabase реєструється у сервіс-локаторі як singleton, що гарантує існування єдиного екземпляра з'єднання з базою даних протягом усього часу роботи застосунку. Це є важливою умовою коректної роботи реактивних потоків Drift, оскільки підписки на Stream повинні отримувати сповіщення від того самого екземпляра бази, до якого вносяться зміни.

Репозиторії (ProjectRepository, TaskRepository) є фасадами над DAO. Вони отримують сирі об'єкти Drift-таблиць (Project, Task) і перетворюють їх на чисті Dart-моделі (ProjectModel, TaskModel) через приватний метод _toModel(). Завдяки цьому шар представлення повністю ізольований від деталей Drift і міг би бути замінений іншою базою даних без змін у кубітах. Такий підхід є практичною реалізацією принципу інверсії залежностей де кубіти залежать від абстракції репозиторію, а не від конкретної реалізації Drift. Репозиторії також відповідають за інкапсуляцію бізнес-правил доступу до даних. Наприклад, метод createTask() автоматично обчислює expectedTime через PertCalculator перед збереженням, а метод archiveProject() змінює лише прапорець isArchived без видалення пов'язаних завдань, що гарантує збереження історії оцінок навіть для завершених проєктів. Крім того, репозиторії є єдиним місцем, де виконується маппінг між nullable-полями Drift (optimisticEstimate, pessimisticEstimate) та відповідними полями моделей, що унеможливлює потрапляння некоректних null-значень у шар представлення.

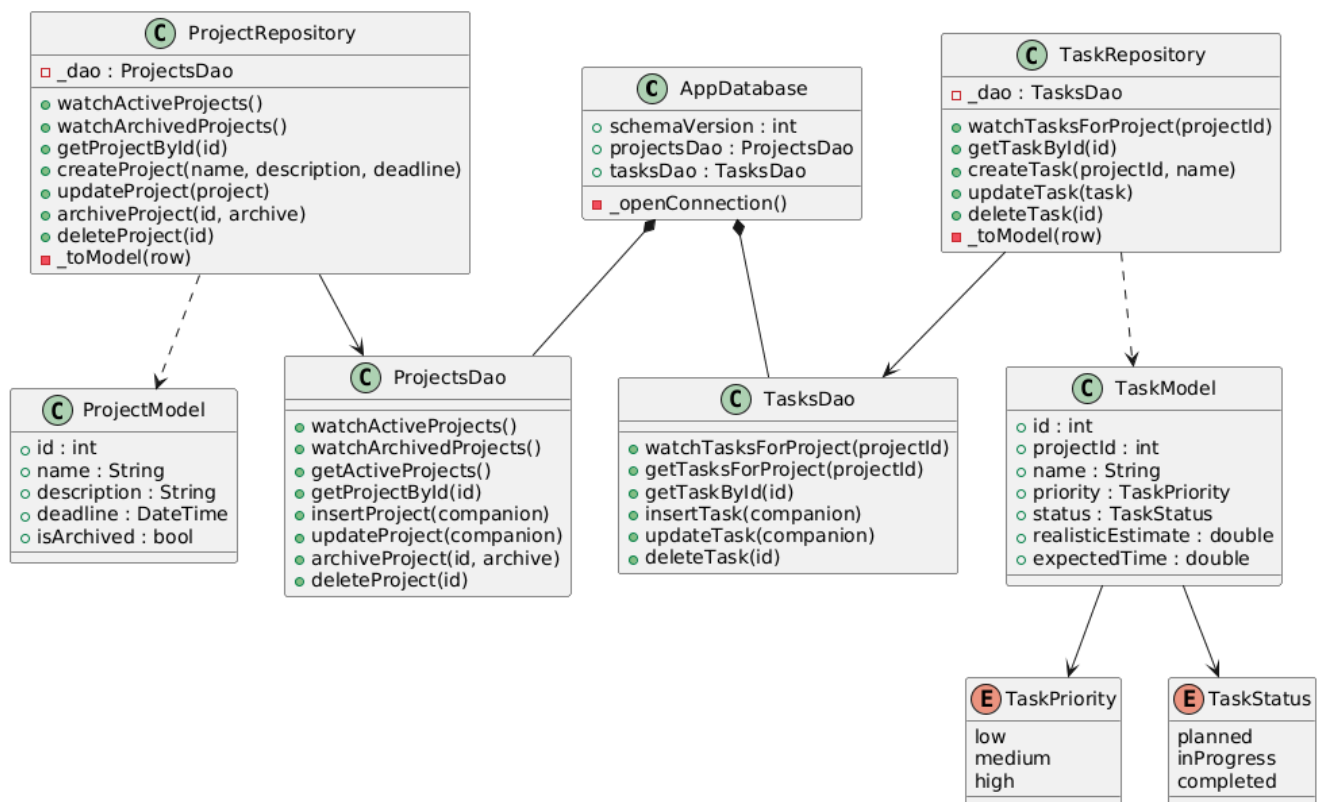


Рисунок 2.4 – Діаграма класів підсистеми даних

Схема бази даних містить дві таблиці з відношенням «один до багатьох». Детальний опис полів обох таблиць наведено у таблиці 2.4.

Таблиця 2.4 – Схема бази даних системи EstiMate

Таблиця	**Поле**	**Тип**	**Опис**
Projects	id	INTEGER PK	Унікальний ідентифікатор проєкту (autoincrement)
Projects	name	TEXT NOT NULL	Назва проєкту
Projects	description	TEXT NULL	Необов'язковий опис проєкту
Projects	deadline	INTEGER NULL	Дедлайн у мілісекундах Unix timestamp (nullable)
Projects	createdAt	INTEGER NOT NULL	Дата створення (Unix timestamp)
Projects	isArchived	BOOLEAN	Ознака архівування (за замовчуванням false)
Tasks	id	INTEGER PK	Унікальний ідентифікатор завдання (autoincrement)
Tasks	projectId	INTEGER FK	Зовнішній ключ → Projects.id (CASCADE DELETE)
Tasks	name	TEXT NOT NULL	Назва завдання
Tasks	priority	INTEGER	Пріоритет: 1 – низький, 2 – середній, 3 – високий
Tasks	status	INTEGER	Статус: 1 – заплановане, 2 – у процесі, 3 – завершене
Tasks	optimistic / pessimistic Estimate	REAL NULL	Оптимістична / песимістична оцінки (nullable)
Tasks	realisticEstimate	REAL NOT NULL	Реалістична оцінка (обов'язкова)
Tasks	expectedTime	REAL NOT NULL	Обчислений очікуваний час за PERT-формулою

Поля optimisticEstimate і pessimisticEstimate є nullable якщо вони не введені, PertCalculator підставляє реалістичну оцінку як значення за замовчуванням, і формула спрощується до T = R. Поле expectedTime обчислюється та зберігається в момент створення або редагування завдання, що дозволяє виконувати агрегацію

часових оцінок на рівні SQL-запитів без необхідності завантажувати всі завдання в пам'ять.

Підсистема управління станом побудована на патерні Cubit. На відміну від повноцінного BLoC, де для кожної дії визначається окремий клас події (Event), Cubit надає прямі методи. Це скорочує кількість файлів і є виправданим, коли відсутня потреба у трансформації потоків подій (debounce, throttle, switchMap). Для даного проєкту такі трансформації не потрібні. Всі операції є синхронними викликами репозиторію або прямими змінами стану.

Кожен кубіт має єдину відповідальність. ProjectsCubit керує списком проєктів і перемиканням між активними та архівованими. ProjectDetailCubit керує деталями одного проєкту та його завданнями з підтримкою фільтрації та сортування. TaskFormCubit обробляє форму завдання й обчислює живий PERT-прев'ю при кожній зміні оцінок. AnalyticsCubit агрегує статистику для екрану аналітики. SettingsCubit зберігає налаштування теми, мови та одиниць часу у SharedPreferences.

2.5 Проєктування сценаріїв ВВ на основі UML-діаграм послідовності

Для двох найважливіших сценаріїв взаємодії побудовано діаграми послідовності. Перший сценарій – створення завдання з PERT-оцінкою – є центральним варіантом використання застосунку і демонструє взаємодію всіх основних компонентів системи. Другий – завантаження деталей проєкту – показує реактивний механізм оновлення UI через потоки Drift.

Сценарій 1: Створення завдання з PERT-оцінкою. Користувач відкриває форму нового завдання через FloatingActionButton на екрані деталей проєкту. GoRouter виконує навігацію на маршрут /projects/:id/tasks/new, передаючи projectId як path parameter. TaskFormScreen ініціалізує TaskFormCubit через BlocProvider, передаючи projectId та залежності через GetIt. Користувач вводить назву та три часові оцінки. Після кожного введення оцінки (O, R або P) onChanged викликає cubit.updateEstimate(), який викликає PertCalculator та емітує новий стан з

оновленим previewPert. BlocBuilder миттєво перебудовує контейнер прев'ю. При натисканні «Зберегти» TaskFormCubit викликає taskRepository.createTask(), Drift записує рядок у таблицю tasks та автоматично сповіщає всіх підписників. Кубіт емітує стан TaskSaved, BlocConsumer закриває екран через context.pop().

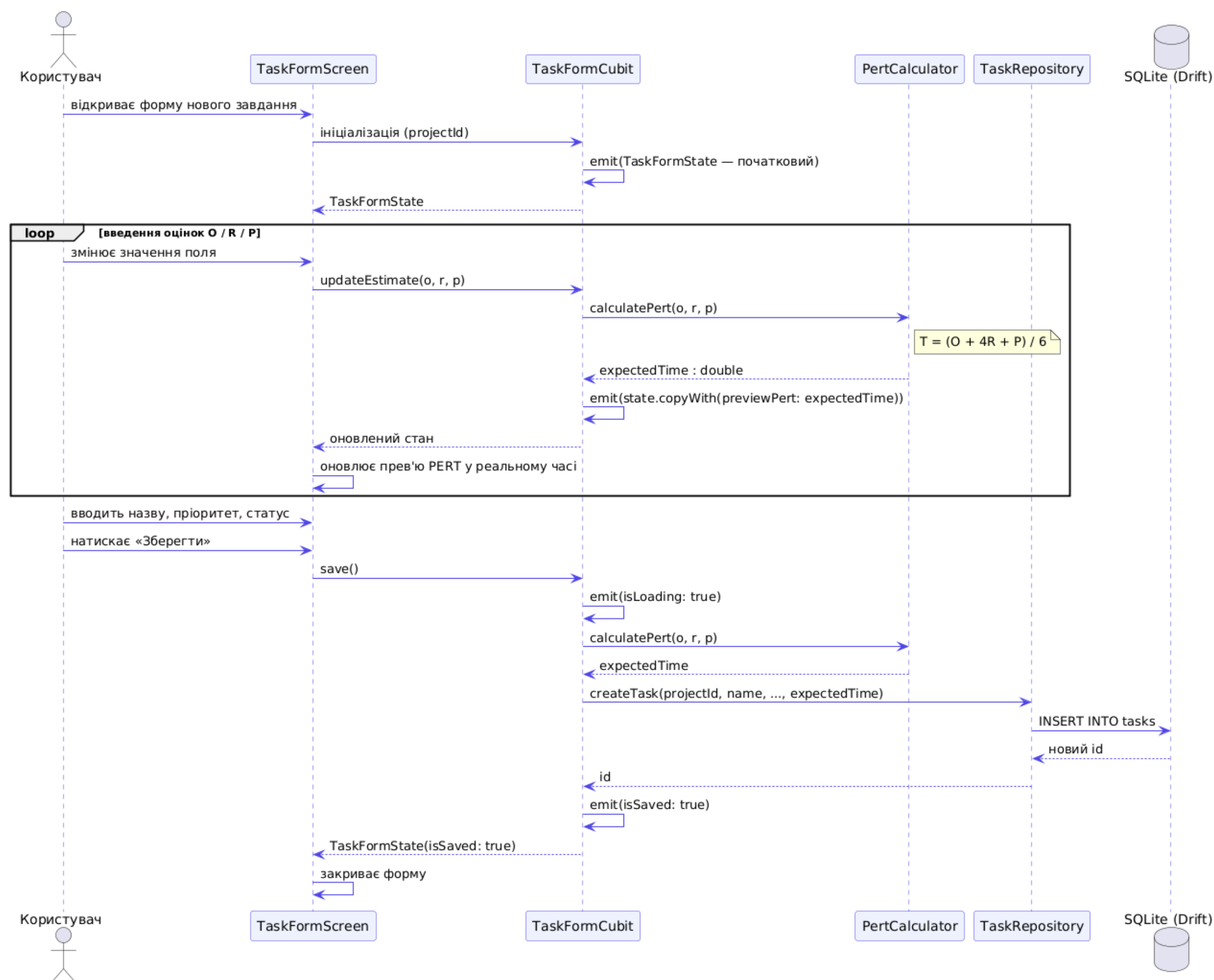


Рисунок 2.5 – Діаграма послідовності: створення завдання з PERT-оцінкою

Сценарій 2: Завантаження деталей проєкту. При відкритті ProjectDetailScreen GoRouter передає projectId з маршруту. BlocProvider ініціалізує ProjectDetailCubit, який у методі _init() підписується на два реактивних потоки: projectRepository.watchProjectById(id) та taskRepository.watchTasksForProject(id). Drift повертає StreamSubscription<Project> та StreamSubscription<List<Task>>. При отриманні першого snapshot кубіт емітує стан ProjectDetailLoaded з даними

проєкту, списком завдань, поточним фільтром та показниками прогресу. BlocBuilder перебудовує екран.

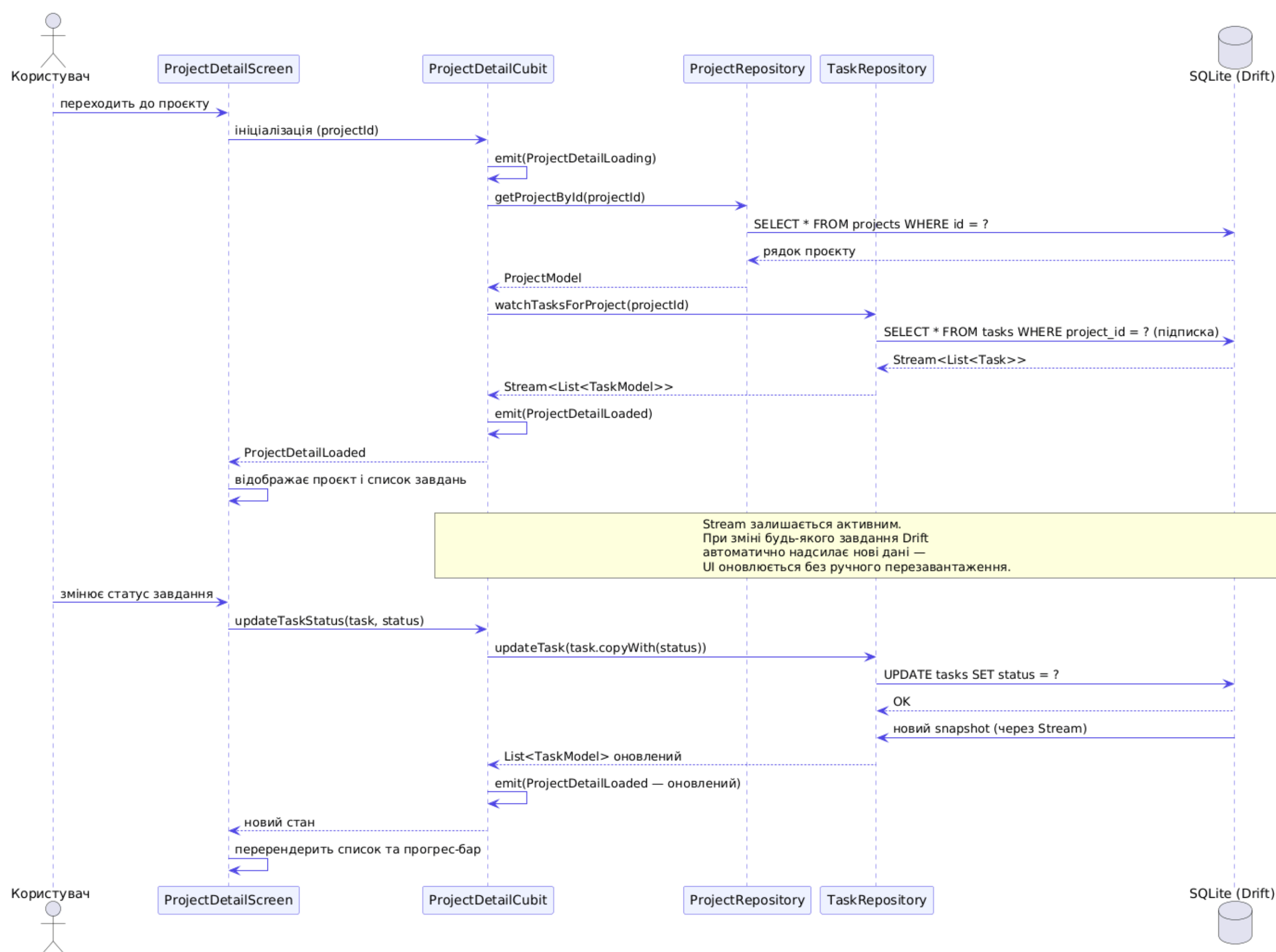


Рисунок 2.6 – Діаграма послідовності: завантаження деталей проєкту

При проєктуванні системи EstiMate застосовано патерн Unidirectional Data Flow (однонаправлений потік даних), згідно з яким дані рухаються строго в одному напрямку: від бази даних через репозиторії та кубіти до екранів. Жоден компонент верхнього рівня не може безпосередньо змінити стан компонента нижнього рівня, що унеможливлює непередбачувані побічні ефекти. Поєднання цього патерну з реактивними потоками Drift забезпечує автоматичне поширення змін по всьому застосунку: оновлення даних у базі миттєво відображається на всіх екранах, які підписані на відповідний Stream, без жодної явної координації між ними.

3 КОНСТРУЮВАННЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

У цьому розділі описано практичну реалізацію застосунку EstiMate. Від ініціалізації та налаштування проєкту до створення користувацького інтерфейсу і проведення тестування на різних рівнях. Реалізація виконана відповідно до архітектурних рішень, прийнятих у розділі 2.

3.1 Базова структура, налаштування проєкту та опис моделей і фасадів системи

Ініціалізація застосунку виконується у файлі main.dart і складається з трьох послідовних кроків. Спочатку викликається WidgetsFlutterBinding.ensureInitialized(), що ініціалізує зв'язок між Dart-кодом і нативним рушієм Flutter саме це обов'язкова умова для виконання будь-яких асинхронних платформних викликів до runApp(). Без цього виклику звернення до SharedPreferences або FlutterLocalNotificationsPlugin до запуску застосунку призвело б до помилки виконання. Далі викликається setupDependencies() з service_locator.dart, що реєструє всі залежності у контейнері GetIt. Після цього запускається застосунок.

Дерево провайдерів стану у точці входу є навмисно мінімальним – лише один кореневий BlocProvider для SettingsCubit. Розміщення SettingsCubit на кореневому рівні дерева є архітектурним рішенням: теми та локаль є глобальними властивостями застосунку, тому вони мають контролюватися єдиним провайдером поза будь-яким конкретним маршрутом. Усі інші кубіти (ProjectsCubit, ProjectDetailCubit тощо) створюються безпосередньо в екранах через BlocProvider і живуть рівно стільки, скільки живе відповідний екран. Це запобігає накопиченню зайвих підписок і витоку пам'яті.

Структуру точки входу застосунку наведено у лістингу 3.1.

Лістинг 3.1 – Точка входу застосунку (main.dart)

```
runApp(
  BlocProvider.value(
    value: sl<SettingsCubit>(),
    child: BlocBuilder<SettingsCubit, SettingsState>(
      builder: (context, settings) => MaterialApp.router(
        theme: AppTheme.light,
        darkTheme: AppTheme.dark,
        themeMode: settings.themeMode,
        locale: settings.locale,
        localizationsDelegates:
AppLocalizations.localizationsDelegates,
        supportedLocales: AppLocalizations.supportedLocales,
        routerConfig: appRouter,
      ),
    ),
  ),
);
```

Використання BlocProvider.value (а не BlocProvider.create) є навмисним бо SettingsCubit вже створений і зареєстрований у GetIt до виклику runApp(), тому BlocProvider лише надає доступ до існуючого екземпляра без передачі права власності. Це забезпечує коректне закриття кубіту при завершенні застосунку через GetIt.

Сервіс-локатор (service_locator.dart) реєструє залежності у суворо визначеному порядку, що відображає граф залежностей знизу вгору. Першим ініціалізується SharedPreferences (асинхронно через await), оскільки від нього залежить SettingsCubit. Потім реєструється AppDatabase як singleton є єдиний екземпляр бази даних на весь час роботи застосунку. З нього отримуються DAO-об'єкти (projectsDao, tasksDao), які передаються у репозиторії. Після репозиторіїв реєструється DeadlineReminderService, що залежить від FlutterLocalNotificationsPlugin та репозиторіїв. Останнім кроком викликається syncAllProjectReminders(), що при кожному запуску застосунку перепланує всі активні нагадування. Це необхідно, оскільки Android може скасовувати заплановані сповіщення після перезавантаження пристрою.

Шар бази даних реалізовано засобами Drift, реактивного ORM поверх SQLite. Клас AppDatabase визначає схему і є центральною точкою доступу до сховища. Реалізацію класу наведено у лістингу 3.2.

Лістинг 3.2 – Клас AppDatabase

```
@DriftDatabase(tables: [Projects, Tasks], daos: [ProjectsDao,
TasksDao])
class AppDatabase extends _$AppDatabase {
  AppDatabase() : super(_openConnection());

  @override
  int get schemaVersion => 2;

  @override
  MigrationStrategy get migration => MigrationStrategy(
    onUpgrade: (m, from, to) async {
      if (from < 2) {
        await m.addColumn(projects, projects.deadline);
      }
    },
  );
}
```

Значення schemaVersion = 2 означає, що схема пройшла одну міграцію. У версії 2 до таблиці projects додано стовпець deadline. MigrationStrategy описує інкрементальну міграцію про підхід, при якому кожна версія описує лише зміни відносно попередньої. Такий підхід гарантує коректне оновлення бази даних для користувачів, що встановили попередню версію застосунку, без втрати їхніх даних. Генерація допоміжного коду (DAO-класи, компаньйони для вставки та оновлення) виконується командою dart run build_runner build --delete-conflicting-outputs.

Таблиця Projects зберігає ідентифікатор, назву, необов'язковий опис, необов'язковий дедлайн, дату створення та прапорець архівування. Таблиця Tasks зберігає зовнішній ключ на Projects з каскадним видаленням (CASCADE DELETE), назву, опис, пріоритет (1–3), статус (1–3), три числових оцінки (оптимістична та песимістична є nullable) та обчислене expectedTime. Каскадне видалення гарантує, що видалення проєкту автоматично видаляє всі пов'язані завдання без додаткових запитів.

Репозиторії є тонким фасадом між DAO та кубітами. Їх єдина відповідальність – перетворення сирих Drift-рядків на чисті Dart-моделі (див. лістинг 3.3). Репозиторії також інкапсулюють бізнес-правило обчислення expectedTime: метод createTask() автоматично викликає PertCalculator та записує результат у поле перед збереженням.

Лістинг 3.3 – Метод _toModel репозиторію проєктів

```
ProjectModel _toModel(Project row) => ProjectModel(
  id: row.id,
  name: row.name,
  description: row.description,
  deadline: row.deadline,
  createdAt: row.createdAt,
  isArchived: row.isArchived,
);
```

Завдяки цьому перетворенню кубіти та екрани повністю ізольовані від типів, що генеруються Drift. Якщо в майбутньому замінити Drift на інший ORM, достатньо буде змінити лише реалізацію репозиторію без будь-яких змін у шарі представлення. Це є практичним проявом принципу інверсії залежностей.

Утиліта PertCalculator є єдиним місцем у коді, де реалізована формула PERT (див. лістинг 3.4). Централізація формули в одній функції гарантує, що будь-яке виправлення розрахунку автоматично поширюється на всі місця використання де форму завдання (живий прев'ю), збереження задачі та аналітику. Функція приймає реалістичну оцінку як обов'язковий параметр, тоді як оптимістична та песимістична є nullable. За їх відсутності підставляється реалістична оцінка, що спрощує формулу до T = R.

Лістинг 3.4 – Функція calculatePert

```
double calculatePert({
  double? optimistic,
  required double realistic,
  double? pessimistic,
}) {
  final o = optimistic ?? realistic;
  final p = pessimistic ?? realistic;
  return (o + 4 * realistic + p) / 6;}
```

3.2 Створення та реалізація користувацького інтерфейсу

Система теми побудована на Material Design 3 з єдиним кольором-насінням #1565C0 (насичений синій) [17, 18]. ColorScheme.fromSeed автоматично генерує всю палітру кольорів від поверхонь до акцентів відповідно до специфікації M3. Визначено два статичних гетери AppTheme.light та AppTheme.dark, які передаються у MaterialApp.router. Перемикання теми виконується через SettingsCubit.toggleTheme(), що зберігає вибір у SharedPreferences і оновлює themeMode у MaterialApp без перезапуску застосунку.

3.2.1 Головний екран та управління проєктами

Навігація реалізована через GoRouter з декларативним визначенням маршрутів. Коренева сторінка / веде на ProjectListScreen, маршрут /settings – на SettingsScreen, а /projects/:projectId – на ProjectDetailScreen з вкладеними маршрутами для завдань та аналітики.

Головний екран (ProjectListScreen) є точкою входу застосунку. При першому запуску відображається порожній стан із підказкою натиснути кнопку "+" для створення першого проєкту, що показано на рисунку 3.1.

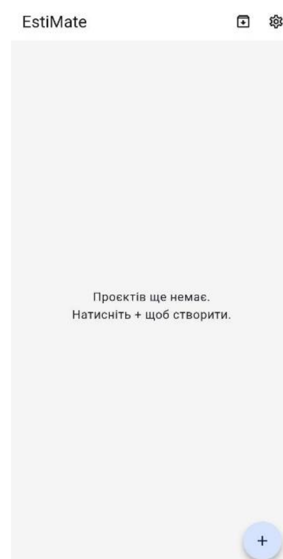


Рисунок 3.1 – Стартовий екран застосунку EstiMate

Для створення нового проєкту відкривається модальне вікно, зображене на рисунку 3.2. Діалог містить три шаблони: без шаблону, веб-застосунок та мобільний застосунок. При виборі шаблону автоматично підставляється набір типових завдань з попередньо заповненими PERT-оцінками. Рисунок 3.3 демонструє вигляд форми після вибору шаблону «Мобільний застосунок», коли система автоматично повідомляє про додавання 8 завдань.

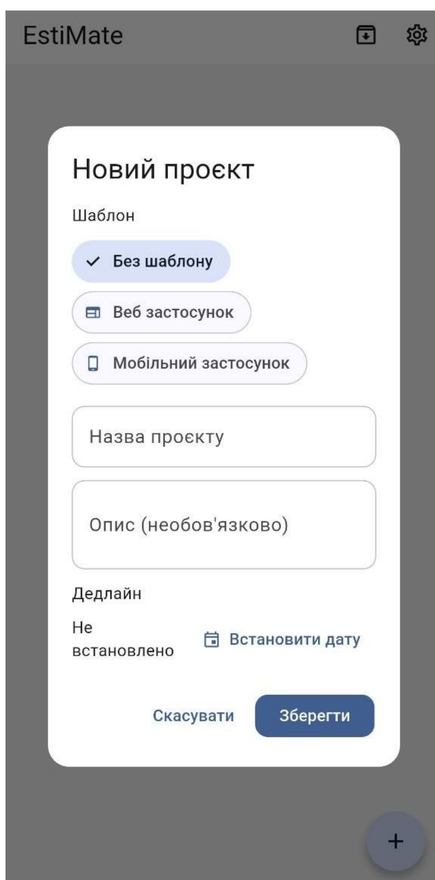


Рисунок 3.2 – Діалог створення проєкту без шаблону

Після вибору шаблону форма автоматично заповнює поле назви та повідомляє про кількість завдань, які будуть додані до проєкту. Шаблон «Мобільний застосунок» містить 8 попередньо налаштованих завдань із реалістичними PERT-оцінками, що дозволяє одразу розпочати роботу без ручного введення типових задач. Шаблон «Веб застосунок» пропонує аналогічний набір завдань, адаптований під специфіку веб-розробки. Поля назви, опису та дедлайну залишаються доступними для редагування незалежно від обраного шаблону.

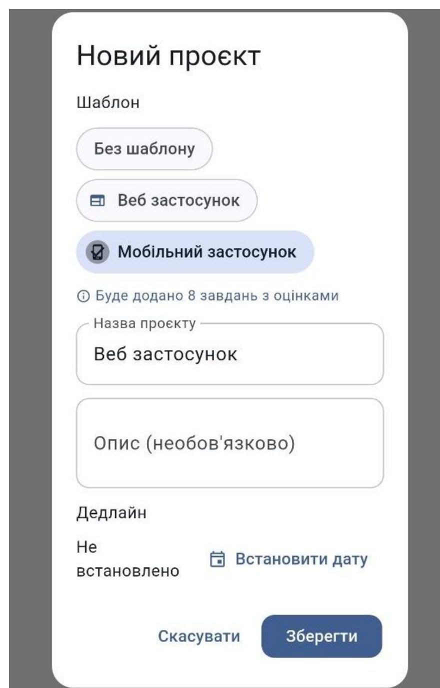


Рисунок 3.3 – Діалог створення проєкту з шаблоном «Мобільний застосунок»

Для встановлення дедлайну відкривається стандартний Material DatePicker, зображений на рисунку 3.4. Після збереження дедлайну застосунок відображає SnackBar-підтвердження, що показано на рисунку 3.5, та автоматично планує push-сповіщення через flutter_local_notifications.

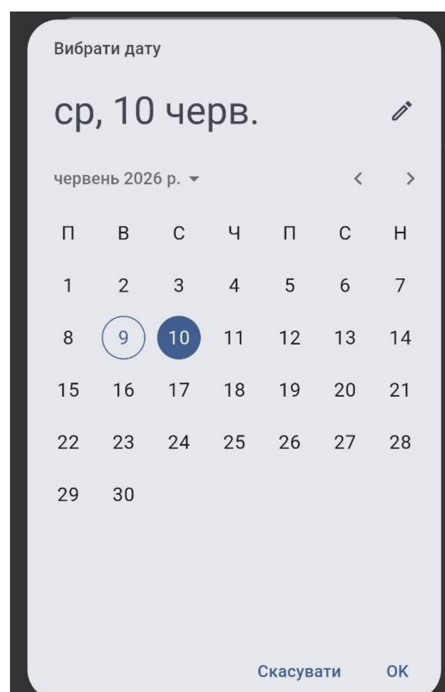


Рисунок 3.4 – Вибір дати дедлайну через DatePicker

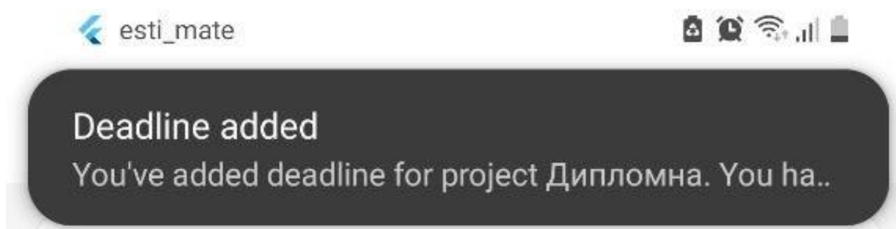


Рисунок 3.5 – SnackBar-підтвердження після встановлення дедлайну

Після створення проєкт відображається на головному екрані у вигляді картки, як показано на рисунку 3.6. Картка містить назву, опис, дату дедлайну, загальний очікуваний час та прогрес-бар. Прогрес-бар оновлюється в реальному часі через StreamBuilder незалежно від стану батьківського кубіту.

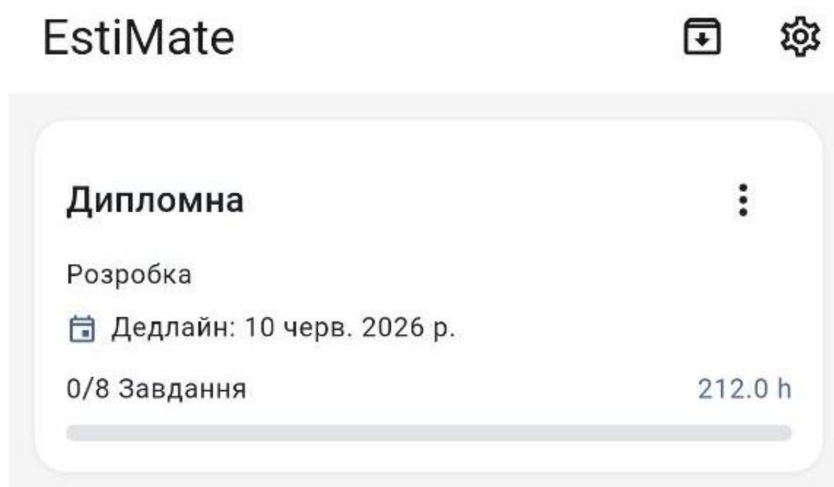


Рисунок 3.6 – Головний екран із створеним проєктом

3.2.2 Екран деталей проєкту та управління завданнями

Екран деталей проєкту (ProjectDetailScreen) відображає три статистичні чіпи у заголовку: загальна кількість завдань, кількість завершених та сумарний очікуваний час. Рисунок 3.7 демонструє повний список завдань із відкритим контекстним меню для зміни статусу. Меню містить опції «Заплановано», «В процесі», «Завершено» та «Видалити».

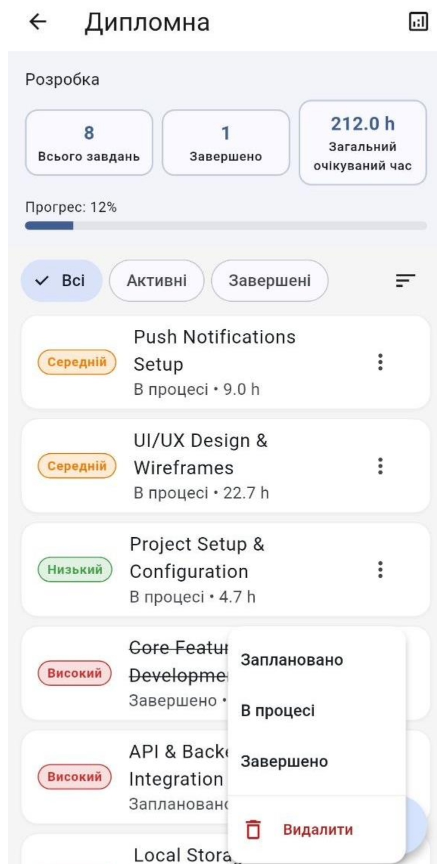


Рисунок 3.7 – Список завдань зі зміною статусу через контекстне меню

На рисунку 3.8 показано результат застосування фільтру «Активні» – у списку залишаються лише завдання зі статусом «В процесі» та «Заплановано». Це дозволяє розробнику зосередитися виключно на поточній роботі, не відволікаючись на вже завершені задачі. Кожне завдання у списку відображає кольоровий чіп пріоритету, назву, поточний статус та розрахований PERT-час, що дає вичерпну інформацію без переходу до детального перегляду.

На рисунку 3.9 показано фільтр «Завершені», де виконані завдання відображаються із закресленим текстом назви. Такий візуальний індикатор одразу відрізняє завершені задачі від активних і є стандартним патерном у застосунках для управління задачами відповідно до специфікації Material Design 3. Завершені завдання продовжують зберігатися у базі даних та враховуються в аналітиці проєкту, що дозволяє відстежувати фактичний прогрес.

Фільтрація виконується через getter filteredTasks у стані кубіту без додаткових запитів до бази даних. Це означає, що при перемиканні між фільтрами

«Всі», «Активні» та «Завершені» застосунок не звертається до SQLite – список перераховується безпосередньо з уже завантажених даних у пам'яті. Завдяки цьому перемикання між фільтрами відбувається миттєво незалежно від кількості завдань у проєкті.

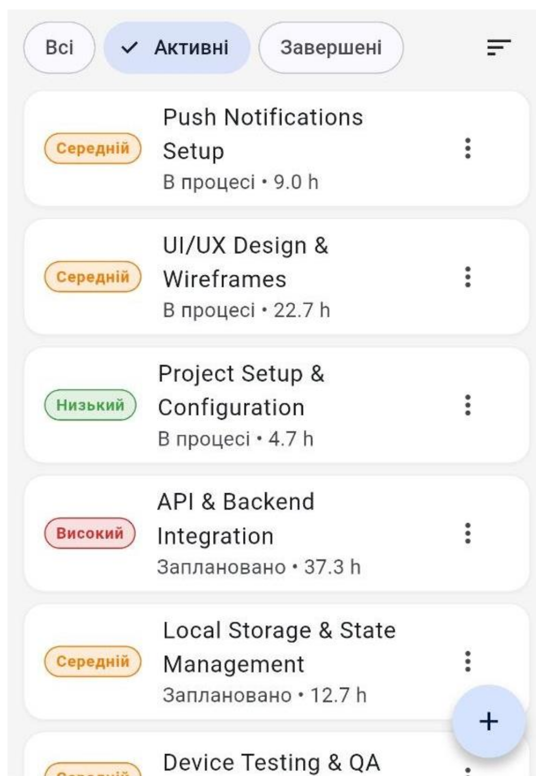


Рисунок 3.8 – Фільтрація активних завдань проєкту

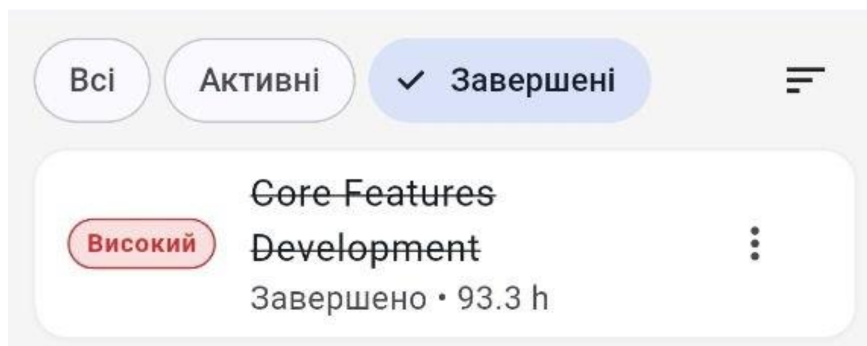


Рисунок 3.9 – Завершені завдання з закресленим текстом

Сортування завдань реалізовано через спливаюче меню, зображене на рисунку 3.10. Доступні чотири опції: за пріоритетом, статусом, датою створення та очікуваним часом.

Рисунок 3.10 – Меню сортування завдань у проєкті

3.2.3 Форма створення та редагування завдання

Форма нового завдання (TaskFormScreen) показана на рисунку 3.11. Вона містить поля для назви, необов'язкового опису, вибору пріоритету та статусу через SegmentedButton, а також три поля для введення оптимістичної, реалістичної та песимістичної оцінок часу.

Рисунок 3.11 – Порожня форма нового завдання

Ключова особливість форми – живе PERT-прев'ю. На рисунку 3.12 показано заповнену форму з введеними оцінками 15, 20 та 40 годин. У нижній частині форми відображається розрахований очікуваний час: T = (15 + 4·20 + 40) / 6 = 22.5 год. Прев'ю оновлюється після кожного натиснення клавіші через цикл onChanged → cubit.updateEstimate() → PertCalculator → emit → BlocBuilder.

Рисунок 3.12 – Заповнена форма завдання з живим PERT-прев'ю 22.5 год

Після збереження завдання воно з'являється у списку проєкту, як показано на рисунку 3.13. Картка завдання містить кольоровий чіп пріоритету, назву, поточний статус та розрахований очікуваний час. Кольорове кодування пріоритетів

відповідає стандарту Material Design 3: червоний для високого, помаранчевий для середнього та зелений для низького пріоритету. Після додавання завдання заголовок екрану деталей проєкту автоматично оновлює лічильник завдань та загальний очікуваний час без перезавантаження сторінки, оскільки ProjectDetailCubit підписаний на реактивний потік watchTasksForProject() і отримує оновлення миттєво через механізм Drift Stream.

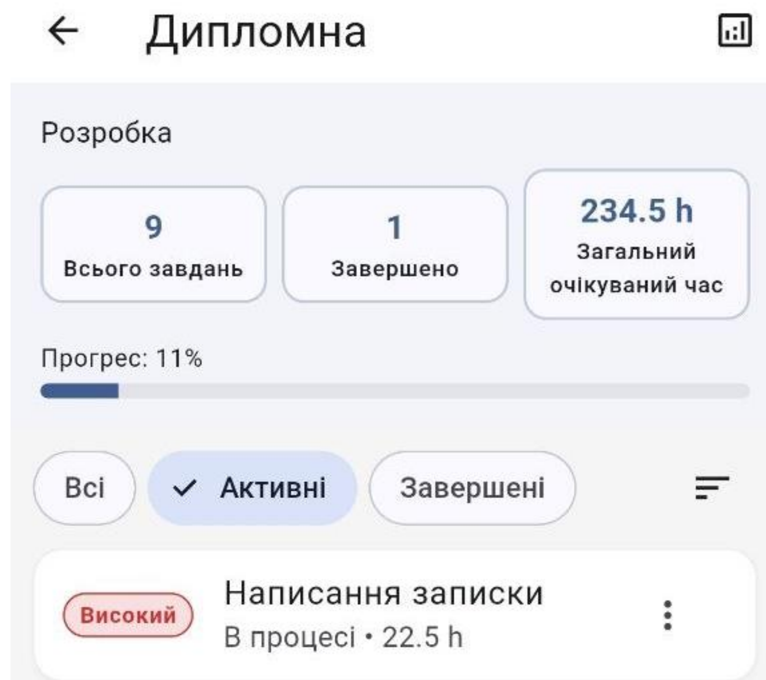


Рисунок 3.13 – Збережене завдання у списку проєкту

3.2.4 Екран аналітики

Екран аналітики (AnalyticsScreen) доступний через іконку у правому верхньому куті екрану деталей проєкту. На рисунку 3.14 показано повний вигляд екрану аналітики. Він відображає чотири картки: загальний прогрес (11%, 1 з 9 завдань), три часові прогнози (мінімальний 97.0 год, очікуваний PERT 234.5 год, максимальний 494.0 год), розподіл за статусами та розподіл за пріоритетами. Усі розрахунки виконуються у геттерах стану AnalyticsState без звернення до бази даних, що дозволяє тестувати аналітику ізольовано від UI.

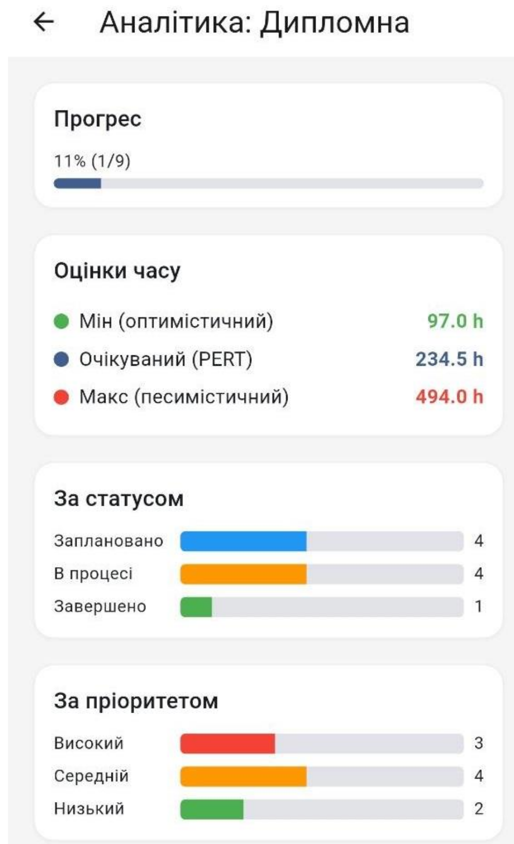
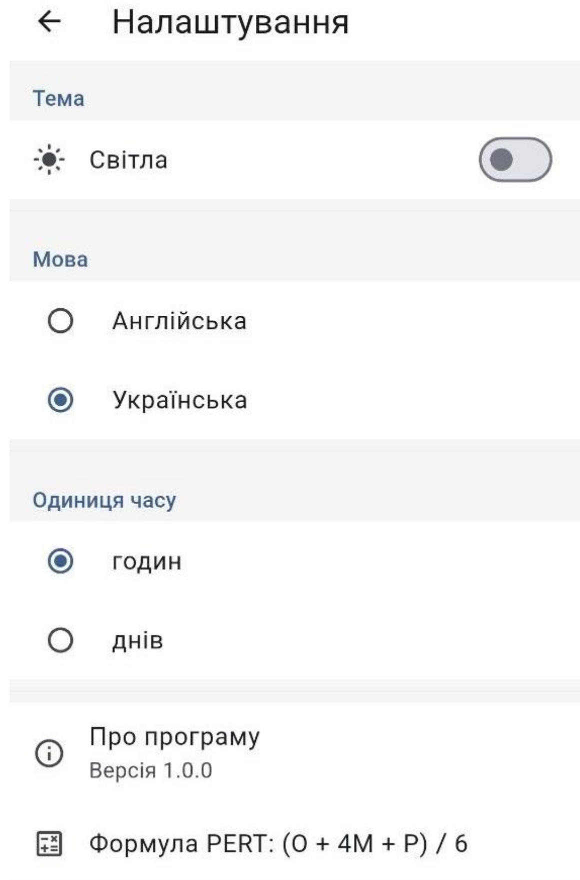


Рисунок 3.14 – Екран аналітики проєкту з розподілом завдань

3.2.5 Екран налаштувань

Екран налаштувань (SettingsScreen) зображено на рисунку 3.15. Він містить три функціональні секції: перемикач теми (Switch між світлою та темною), вибір мови інтерфейсу (Англійська / Українська) та вибір одиниць виміру часу (години / дні). Вибір теми зберігається у SharedPreferences і відновлюється при наступному запуску. Перемикання мови відбувається без перезапуску завдяки динамічній зміні Locale у MaterialApp, що тригерить перебудову всіх залежних віджетів. Вибір одиниць часу впливає на відображення PERT-оцінок у всьому застосунку: якщо обрано «дні», всі значення автоматично конвертуються через функцію форматування formatTime().

На рисунку 3.16 показано діалог «Про EstiMate», що відображає версію застосунку, технологічний стек та формулу PERT для довідки.



Рисунки 3.15 – Екран налаштувань застосунку

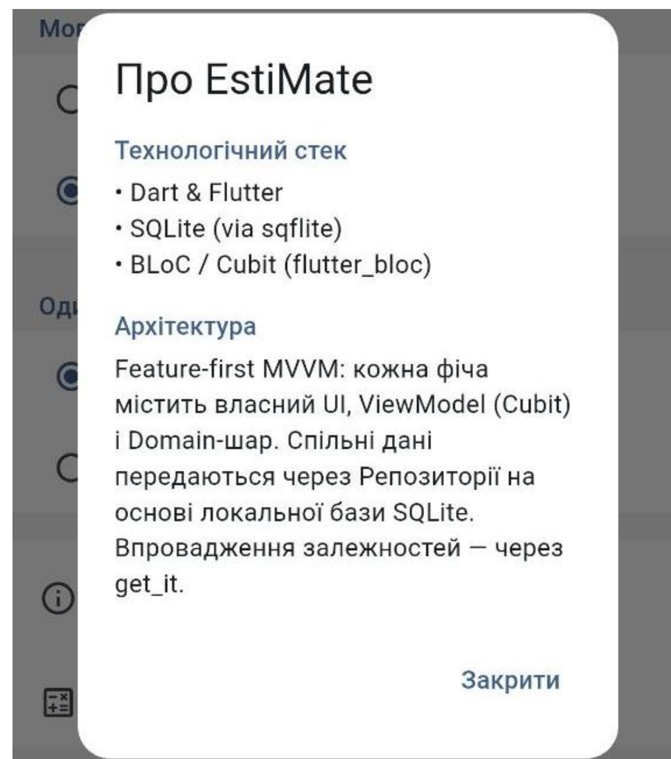


Рисунок 3.16 – Діалог «Про EstiMate» з технологічним стеком

3.2.6 Архівування проєктів та push-сповіщення

Завершені або призупинені проєкти можна перевести до архіву через контекстне меню на головному екрані. На рисунку 3.17 показано список архівованих проєктів, доступний через іконку архіву у правому верхньому куті головного екрану. Архівовані проєкти зберігають усі завдання та оцінки, що дозволяє переглядати історію без засмічення активного списку.

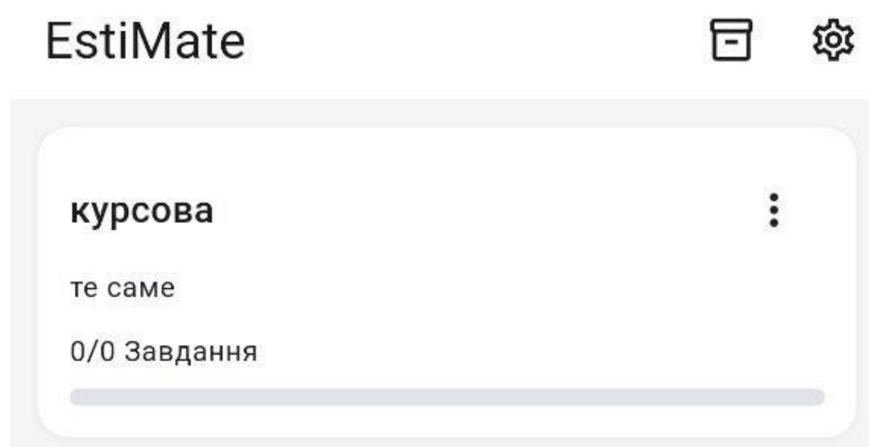


Рисунок 3.17 – Список архівованих проєктів

Push-сповіщення реалізовані через flutter_local_notifications у поєднанні з пакетом timezone. На рисунку 3.18 показано сповіщення від застосунку EstiMate у шторці сповіщень Android: при настанні дедлайну відображається назва проєкту та повідомлення про прострочення. Сповіщення планується на точний час з урахуванням часового поясу пристрою та автоматично відновлюється після перезавантаження через метод syncAllProjectReminders().

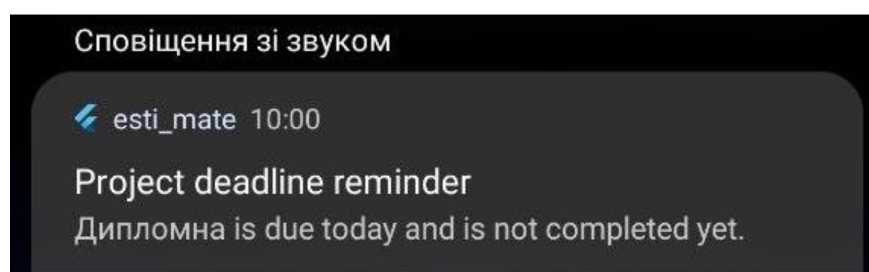


Рисунок 3.18 – Push-сповіщення про настання дедлайну проєкту

Локалізація реалізована через ARB-файли (lib/l10n/app_en.arb та app_uk.arb) [22]. Кожен файл містить пари ключ-значення для всіх рядків інтерфейсу відповідною мовою. Клас AppLocalizations генерується автоматично командою flutter gen-l10n під час збірки проєкту, що унеможливлює розбіжності між ключами у різних мовних файлах. Усі рядки інтерфейсу доступні через цей згенерований клас без жодного хардкодування тексту безпосередньо у віджетах. Перемикання мови відбувається миттєво через SettingsCubit.setLanguage() без перезапуску застосунку – зміна locale у MaterialApp тригерить перебудову всіх залежних віджетів, і інтерфейс повністю перекладається за частки секунди.

3.3 Тестування системи

Тестування застосунку EstiMate охоплює три рівні відповідно до піраміди тестування коли модульне тестування ізольованих одиниць логіки утворює базу, автоматизоване тестування віджетів це середній рівень, ручне тестування наскрізних сценаріїв про вершину. Такий підхід дозволяє перевірити коректність системи як на рівні окремих функцій, так і на рівні повних користувацьких сценаріїв.

3.3.1 Модульне тестування

Модульне тестування спрямоване на перевірку ізольованих одиниць логіки функцій, методів класів та кубітів. Тести написані з використанням стандартного пакету flutter_test, що входить до SDK Flutter [21]. Перевага модульних тестів полягає у швидкості виконання коли весь набір тестів PertCalculator виконується менш ніж за 100 мілісекунд, що дозволяє запускати їх при кожному коміті.

Ключовим об'єктом для модульного тестування є функція calculatePert з PertCalculator, оскільки вона реалізує основну бізнес-логіку застосунку. Перевіряються чотири сценарії, наведені у таблиці 3.2.

Таблиця 3.2 – Тестові сценарії функції calculatePert

Сценарій	**O**	**R**	**P**	**Очікуваний результат**
Стандартний	2	5	10	(2 + 4·5 + 10) / 6 = 5.333...
Лише R (O та P відсутні)	–	4	–	4.0 (O=P=R)
Симетричний розподіл	3	5	7	5.0 (O та P рівновіддалені від R)
Рівні значення	5	5	5	5.0 (нульова невизначеність)

Тести функції PertCalculator наведено у лістингу 3.6.

Лістинг 3.6 – Модульні тести функції calculatePert

```
void main() {
  group('PertCalculator', () {
    test('стандартний розрахунок за формулою (O + 4R + P) / 6', () {
      final result = calculatePert(
        optimistic: 2, realistic: 5, pessimistic: 10,
      );
      // (2 + 4*5 + 10) / 6 = 32 / 6 ≈ 5.33
      expect(result, closeTo(5.333, 0.001));
    });

    test('якщо оптимістична та песимістична відсутні – результат
дорівнює R', () {
      final result = calculatePert(realistic: 4);
      expect(result, equals(4.0));
    });

    test('симетричний розподіл – результат дорівнює R', () {
      final result = calculatePert(
        optimistic: 3, realistic: 5, pessimistic: 7,
      );
      expect(result, equals(5.0));
    });
  });}
```

Окрім тестування утиліти, модульним тестуванням охоплено логіку кубітів [22]. Для тестування кубітів використовується пакет bloc_test, що надає зручний метод blocTest() для перевірки послідовності станів. При тестуванні кубітів залежності (SharedPreferences, репозиторії) замінюються mock-об'єктами за допомогою пакету mocktail [23]. Це дозволяє тестувати логіку кубіту ізольовано,

без реального запису до SharedPreferences або звернення до SQLite (див. лістинг 3.7).

Лістинг 3.7 – Модульний тест SettingsCubit

```
blocTest<SettingsCubit, SettingsState>(
  'toggleTheme перемикає на темну тему',
  build: () => SettingsCubit(prefs),
  act: (cubit) => cubit.toggleTheme(),
  expect: () => [
    isA<SettingsState>().having(
      (s) => s.themeMode, 'themeMode', ThemeMode.dark,
    ),
  ],
);
```

Метод blocTest() приймає чотири основних параметри. Перший з них build є фабрикою кубіту, другий act описує дію що виконується, третій expect містить очікувану послідовність станів, а четвертий verify є опціональною перевіркою побічних ефектів, наприклад того, що mock-метод збереження був викликаний. Такий декларативний стиль тестів є більш читабельним порівняно з ручним управлінням підписками на Stream.

3.3.2 Автоматизоване тестування

Автоматизоване тестування віджетів у Flutter виконується через WidgetTester, який запускає дерево віджетів у тестовому оточенні без реального пристрою або емулятора. Такі тести є значно швидшими за інструментальні тести, оскільки не вимагають запуску Android-емулятора. Водночас вони перевіряють реальний код рендерингу Flutter, на відміну від звичайних mock-об'єктів, що робить їх надійнішим інструментом перевірки поведінки UI.

Для TaskFormScreen перевіряється, чи відображається прев'ю PERT після введення всіх трьох оцінок (див. лістинг 3.8).

Лістинг 3.8 – Автоматизований тест PERT-прев'ю у TaskFormScreen

```
testWidgets('PERT-прев\'ю з\'являється після введення оцінок',
(tester) async {
```

```
  await tester.pumpWidget(
    MaterialApp(
      home: BlocProvider(
        create: (_) => TaskFormCubit(
          taskRepo: MockTaskRepository(),
          deadlineReminderService: MockDeadlineReminderService(),
          projectId: 1,
        ),
        child: const TaskFormBody(),
      ),
    ),
  );
  await tester.enterText(find.byKey(const Key('optimistic_field')),
'2');
  await tester.enterText(find.byKey(const Key('realistic_field')),
'5');
  await tester.enterText(find.byKey(const Key('pessimistic_field')),
'10');
  await tester.pump();
  expect(find.textContaining('5.3'), findsOneWidget)
  ;}
);
```

Виклик tester.pump() після введення тексту необхідний для обробки pendingMicrotasks, оскільки він дозволяє BlocBuilder отримати новий стан від TaskFormCubit та перебудувати відповідний віджет. Без pump() стан ще не встиг поширитися через дерево віджетів, і expect() завершився б невдачею.

Окрім тестування форм, виконується перевірка відображення станів кубітів. ProjectsLoading показує CircularProgressIndicator, ProjectsLoaded відображає список карток або повідомлення про порожній список залежно від наявності даних, а ProjectsError виводить текст помилки з можливістю повторного завантаження.

Тестування аналітичних розрахунків виконується на рівні геттерів стану AnalyticsState без запуску UI. Перевіряється коректність обчислення прогресу, агрегація за статусами та пріоритетами, а також точність значень totalOptimistic, totalExpected і totalPessimistic. Такий підхід є найефективнішим, оскільки тест виконується за мікросекунди без побудови дерева віджетів.

Ручне тестування застосунку проведено на реальному пристрої Android версії 14 (API 34). Результати ручного тестування зведено у таблицю 3.3.

Таблиця 3.3 – Результати ручного тестування на реальному пристрої Android

Сценарій	**Результат**	**Примітка**
Повний цикл CRUD проєктів та завдань	Успішно	Дані зберігаються між сесіями
Архівування і відновлення проєкту	Успішно	Завдання зберігаються при архівуванні
Планування push-сповіщення про дедлайн	Успішно	Сповіщення за 24 год та в момент дедлайну
Перепланування після перезавантаження	Успішно	syncAllProjectReminders відновлює нагадування
Перемикання теми (світла/темна/системна)	Успішно	Тема зберігається між запусками
Перемикання мови (укр/eng)	Успішно	Перемикання без перезапуску застосунку
Аналітика при різному наповненні	Успішно	Коректна агрегація при 0 і 20+ завданнях
Збереженість даних після примусового закриття	Успішно	SQLite транзакції гарантують цілісність

Усі перевірені сценарії пройшли успішно. Особливої уваги заслуговує перепланування сповіщень після перезавантаження пристрою, оскільки Android 12+ скасовує всі заплановані AlarmManager-події. Метод syncAllProjectReminders() викликається до запуску UI, отримує активні проєкти з дедлайнами та переплануе сповіщення з урахуванням часового поясу пристрою.

У третьому розділі описано практичну реалізацію застосунку EstiMate. Реалізовано п'ять екранів на основі реактивної архітектури Drift Stream та Cubit, живе PERT-прев'ю, надійне планування push-сповіщень та локалізацію інтерфейсу. Тестування на трьох рівнях підтвердило коректність бізнес-логіки та надійність наскрізних сценаріїв на реальному пристрої Android.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

Розробка мобільного застосунку EstiMate здійснюється в умовах тривалої роботи за ПК у поєднанні з використанням смартфона для тестування інтерфейсу. Попри відсутність важкої фізичної праці, такий вид діяльності формує специфічне навантаження на зоровий апарат та нервову систему розробника. У цьому розділі розглянуто ергономічні проблеми безпеки життєдіяльності мобільного розробника та вимоги до режимів праці й відпочинку при роботі з відеодисплейними терміналами.

4.1 Безпека життєдіяльності. Ергономічні проблеми безпеки життєдіяльності

Ергономіка вивчає взаємодію людини з технічними системами з метою забезпечення ефективності, безпеки та комфорту в процесі праці [1]. Для розробника мобільного застосунку ця дисципліна є особливо актуальною, оскільки розробка EstiMate поєднує одночасну роботу з монітором ПК при написанні коду в Android Studio та зі смартфоном при тестуванні інтерфейсу. Нормативну базу організації робочого місця оператора ПК в Україні формує ДСанПіН 3.3.2.007-98 та ДСТУ EN ISO 9241 [7]. Ці стандарти регламентують параметри робочого місця, вимоги до обладнання та допустимі рівні навантаження для операторів ВДТ. Дотримання їх вимог є обов'язковою умовою безпечної організації праці розробника.

Специфіка мобільної розробки створює підвищене зорове навантаження через необхідність постійно переакомодовувати зір між двома пристроями з різною відстанню до очей. Монітор ПК розташований на відстані 600–700 мм, тоді як смартфон при триманні в руці знаходиться приблизно на 300–350 мм від очей. Різниця у відстанях змушує циліарний м'яз ока постійно змінювати кривизну кришталика, що прискорює настання зорової втоми порівняно з роботою лише за одним екраном [4]. Окрім зорового навантаження, тривале тримання смартфона в

руках під час тестування спричиняє статичне напруження м'язів передпліч та зап'ясть, що є додатковим чинником ризику. Для зниження цих навантажень рекомендується розміщувати смартфон на підставці поряд з монітором, а не тримати його в руках під час тестування.

Нормативні параметри організації робочого місця розробника мобільного застосунку наведено у таблиці 4.1.

Таблиця 4.1 – Ергономічні параметри робочого місця розробника мобільного застосунку

Параметр	**Нормативне значення**
Відстань від очей до монітора	600–700 мм (не менше 500 мм)
Верхній край монітора	На рівні очей або трохи нижче (10°–20° вниз)
Висота робочого стола	680–800 мм або фіксована 725 мм
Висота сидіння крісла	400–500 мм (регульована)
Кут між стегном та гомілкою	90°–110°
Кут між передпліччям та плечем	90°–100°
Відстань до смартфона при тестуванні	300–400 мм (пристрій на підставці)

Тривала розробка без дотримання ергономічних вимог призводить до зорової втоми та астенопії, синдрому карпального каналу при неправильному положенні зап'ясть і шийного остеохондрозу при нахиленому положенні голови [7]. Для профілактики цих порушень рекомендується виконувати такі заходи:

– налаштувати яскравість монітора та смартфона на однаковий рівень 80–120 кд/м² для зниження контрасту при перемиканні погляду;
– активувати нічний режим на обох пристроях після 20:00 та використовувати темну тему в Android Studio й у застосунку EstiMate;
– виконувати гімнастику для очей за правилом «20-20-20» – кожні 20 хвилин дивитися протягом 20 секунд на об'єкт за 6 метрів;

виконувати вправи для шиї та плечового поясу під час регламентованих перерв для зняття статичного напруження м'язів.

4.2 Основи охорони праці. Вимоги до режимів праці і відпочинку при роботі з ВДТ

Розробка застосунку EstiMate є діяльністю, що відноситься до роботи з відеодисплейними терміналами (ВДТ). Специфіка цієї роботи полягає в одночасному використанні монітора ПК та смартфона як другого ВДТ, що збільшує сумарне зорове навантаження порівняно зі звичайною офісною роботою. Така особливість мобільної розробки є важливою з точки зору охорони праці, оскільки чинні нормативи встановлено для роботи з одним терміналом, тоді як розробник фактично працює з двома. Вимоги до режимів праці та відпочинку регламентуються ДСанПіН 3.3.2.007-98 та відповідними нормативними актами з охорони праці [9].

Розробка програмного забезпечення, включаючи написання коду на Dart, налагодження Flutter-застосунку та проєктування архітектури на основі Clean Architecture, відноситься до I категорії роботи з ВДТ, яка характеризується найвищим інтелектуальним та зоровим навантаженням [10]. На відміну від II та III категорій, де переважають операції введення даних або діалоговий режим, I категорія передбачає тривалу творчу та аналітичну роботу, що суттєво підвищує нервово-емоційне навантаження на розробника. Нормативи регламентованих перерв наведено у таблиці 4.2.

Таблиця 4.2 – Регламентовані перерви при роботі з ВДТ (ДСанПіН 3.3.2.007-98)

Категорія роботи	**Час з ВДТ, год**	**Перерва після кожних, хв**	**Тривалість перерви, хв**
I – розробка ПЗ	до 6	60	15
II – введення даних	до 4	60	10
III – діалоговий режим	до 2	60	10

При 8-годинному робочому дні загальна тривалість регламентованих перерв для розробника EstiMate становить не менше 50–60 хвилин. Тестування інтерфейсу

на смартфоні додатково збільшує фактичне навантаження на зоровий апарат понад нормативне для однієї категорії, тому рекомендується враховувати час роботи зі смартфоном при плануванні загального режиму праці [2].

Ефективна організація режиму праці передбачає раціональний розподіл задач протягом дня. Інтелектуально навантажені задачі, такі як проєктування PERT-алгоритмів та налагодження Cubit-логіки, рекомендується виконувати у першій половині дня, коли розумова працездатність є максимальною. Рутинні задачі – написання шаблонного коду та ручне тестування UI – доцільно переносити на другу половину дня. Під час перерви рекомендується повністю відволікатися від обох екранів, оскільки використання смартфона під час відпочинку від ПК не знімає зорового навантаження, а лише змінює його джерело [9].

Загальна тривалість роботи за ПК не повинна перевищувати 8 годин на добу, а безперервні сеанси написання коду слід обмежувати 45–50 хвилинами з наступною активною перервою. Важливим чинником збереження продуктивності є також психологічне розвантаження: після завершення робочого дня рекомендується уникати перегляду коду або робочих повідомлень протягом щонайменше 1–2 годин. Дотримання режиму сну 7–8 годин є необхідною умовою підтримання когнітивної продуктивності, оскільки недосипання суттєво знижує якість коду та здатність до виявлення логічних помилок у застосунку [10].

ВИСНОВКИ

У кваліфікаційній роботі вирішено науково-технічну задачу розробки мобільного застосунку EstiMate для платформи Android, що реалізує метод PERT (Program Evaluation and Review Technique) для структурованого оцінювання трудовитрат у розрізі проєктів та завдань. Застосунок заповнює виявлену ринкову нішу – відсутність безкоштовного офлайн-інструменту, що поєднує PERT-оцінювання, управління проєктами та аналітику в єдиному мобільному рішенні.

Проведено аналіз п'яти конкурентних рішень (Microsoft Project, Jira, Asana, Todoist, PERT-калькулятори) за дев'ятьма критеріями. Встановлено, що жоден із розглянутих інструментів не поєднує одночасно підтримку PERT, офлайн-режим, мобільну платформу та безкоштовний доступ, що підтвердило актуальність і доцільність розробки застосунку EstiMate.

Сформульовано специфікацію застосунку, що включає 10 функціональних вимог, які охоплюють управління проєктами та завданнями, PERT-оцінювання, відстеження статусів, push-сповіщення про дедлайни та аналітику, а також 6 нефункціональних вимог щодо офлайн-режиму, часу відгуку до 1 секунди, підтримки Android 6.0 і вище, відповідності Material Design 3, локалізації та надійності збереження даних.

Обрано та обґрунтовано технологічний стек. Flutter з мовою Dart обрано як основний фреймворк завдяки вбудованій підтримці Material Design 3 та можливості розширення на iOS. Drift обрано як реактивний ORM для SQLite на підставі порівняльного аналізу альтернатив за критеріями типобезпеки, реактивних потоків та підтримки реляційної моделі. Cubit із пакету flutter_bloc обрано для управління станом як оптимальний баланс між простотою та тестованістю.

Виконано комплексне проєктування системи. Побудовано діаграму варіантів використання, що містить 20 варіантів у п'яти пакетах. Обрано та обґрунтовано архітектуру Feature-based Clean Architecture, що перевершує альтернативи MVC та MVVM за критеріями тестованості та ізоляції модулів. Спроєктовано реляційну

схему бази даних із двома таблицями та каскадним видаленням, а також розроблено UML-діаграми ієрархії класів та послідовності взаємодії компонентів.

Реалізовано мобільний застосунок EstiMate, що включає п'ять функціональних екранів. Ключовою особливістю реалізації є живе PERT-прев'ю у формі завдання, реактивна архітектура на основі Drift Stream та Cubit, що забезпечує автоматичне оновлення UI без ручного перезавантаження, а також надійний механізм перепланування push-сповіщень після перезавантаження пристрою через метод syncAllProjectReminders.

Проведено тестування на трьох рівнях. Модульне тестування верифікувало коректність функції calculatePert на чотирьох сценаріях та логіку SettingsCubit. Автоматизоване тестування віджетів підтвердило коректність відображення PERT-прев'ю та відповідність екранів станам кубітів. Ручне тестування на реальному пристрої Android підтвердило коректність дев'яти наскрізних сценаріїв.

Практична цінність роботи полягає у створенні готового до використання мобільного застосунку, який індивідуальні розробники та проєктні менеджери можуть застосовувати для структурованого ведення проєктів та PERT-оцінювання завдань в умовах офлайн-роботи. Застосунок підтримує українську та англійську мови, що робить його доступним для широкої аудиторії.

Достовірність результатів підтверджується відповідністю реалізованої функціональності визначеній специфікації вимог, успішним проходженням модульних та автоматизованих тестів, а також верифікацією наскрізних сценаріїв на реальному Android-пристрої. Архітектурні рішення обґрунтовані порівняльним аналізом альтернатив за формалізованими критеріями.

Перспективами подальшого розвитку застосунку EstiMate є розширення на платформу iOS засобами Flutter без зміни бізнес-логіки, додавання опціональної хмарної синхронізації даних між пристроями, реалізація командного режиму роботи з розмежуванням прав доступу, а також інтеграція з популярними системами управління проєктами через REST API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chaos Report 2020. The Standish Group International. URL: https://www.standishgroup.com/sample_research_files/CHAOSReport2020.pdf (дата звернення: 02.04.2026).

2. Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK Guide). 7th ed. Newtown Square: PMI, 2021. 370 p.

3. Microsoft Project. Microsoft Corporation. URL: https://www.microsoft.com/uk-ua/microsoft-365/project/project-management-software (дата звернення: 04.04.2026).

4. Jira Software. Atlassian. URL: https://www.atlassian.com/software/jira (дата звернення: 05.04.2026).

5. Asana: Manage your team's work, projects, & tasks online. Asana, Inc. URL: https://asana.com (дата звернення: 06.04.2026).

6. Todoist: The to do list to organize work & life. Doist Inc. URL: https://todoist.com (дата звернення: 07.04.2026).

7. Flutter documentation. Google LLC. URL: https://docs.flutter.dev (дата звернення: 10.04.2026).

8. Drift: Reactive & typesafe persistence library for Dart & Flutter. URL: https://drift.simonbinder.eu (дата звернення: 15.04.2026).

9. flutter_bloc | Flutter package. Dart packages. URL: https://pub.dev/packages/flutter_bloc (дата звернення: 18.04.2026). 10.

10. get_it package. Thomas Burkhart. URL: https://pub.dev/packages/get_it (дата звернення: 19.04.2026).

11. go_router package. Flutter team. URL: https://pub.dev/packages/go_router (дата звернення: 20.04.2026).

12. flutter_local_notifications package. Michael Bui. URL: https://pub.dev/packages/flutter_local_notifications (дата звернення: 25.04.2026).

13. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. New Jersey: Prentice Hall, 2017. 432 p.

14. Arias-Orezano J. F., Reyna-Barreto B. D., Mamani-Apaza G. Impact of Clean Architecture and ISO/IEC 25010 on the Maintainability of Android Applications. TecnoLógicas. 2021. Vol. 24, No. 52. P. 226–241. DOI: https://doi.org/10.22430/22565337.2104.

15. PlantUML: Open-source tool to draw UML diagrams. URL: https://plantuml.com (дата звернення: 26.04.2026).

16. Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5.1. OMG Document Number formal/2017-12-05. Needham: Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (дата звернення: 29.04.2026).

17. Santiago-Salazar J. A., Rico-Bautista D. Clean Architecture: Impact on Performance and Maintainability of Native Android Projects. Advances in Computing. CCC 2023. Springer, Cham, 2024. P. 85–97. DOI: https://doi.org/10.1007/978-3-031-47372-2_8.

18. Reactive programming in Dart. Dart team. URL: https://dart.dev/tutorials/language/streams (дата звернення: 28.04.2026).

19. Material Design 3. Google LLC. URL: https://m3.material.io (дата звернення: 01.05.2026).

20. Internationalizing Flutter apps. Flutter documentation. Google LLC. URL: https://docs.flutter.dev/ui/accessibility-and-internationalization/internationalization (дата звернення: 05.05.2026).

21. Testing Flutter apps. Flutter documentation. Google LLC. URL: https://docs.flutter.dev/testing/overview (дата звернення: 06.05.2026).

22. bloc_test package. Felix Angelov. URL: https://pub.dev/packages/bloc_test (дата звернення: 07.05.2026).

23. mocktail package. Felix Angelov. URL: https://pub.dev/packages/mocktail (дата звернення: 09.05.2026).

24. Андрейчук Н.І. та ін. Охорона праці. Львів: Видавництво Львівська політехніка, 2021. 276 с.

25. Жидецький В.Ц. Охорона праці користувачів комп'ютерів. Львів: Афіша, 2020. 176 с.

26. Безпека життєдіяльності та охорона праці / Сокуренко В.В. та ін. Харків: ХНУВС, 2021. 308 с.

27. Мелех Л.В. Безпека життєдіяльності та охорона праці. Львів: ЛДУ, 2022. 219 с.

28. Пістун І.П., Кочубей В.І. Практикум з безпеки життєдіяльності. 2023. 560 с.

29. Атаманчук П.С. Безпека життєдіяльності. Київ, 2020. 276 с.

30. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі: Михалик Д.М., Цуприк Г.Б., Бревус В.М. – Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 45с.

31. Репозиторій EstiMate. Github. URL: https://github.com/Gr1bochek/esta-mate

32. Постер. Застосування методу pert для оцінки трудомісткості задач у мобільних застосунках управління проєктами. URL: https://docs.google.com/presentation/d/1W9B6NIlxEglA19Du9gyhcZN8XQHhhA1IRs1D5TXh1g4/edit?pli=1&slide=id.p1#slide=id.p1.

ДОДАТКИ

Додаток А – Діаграма пакетів архітектури системи EstiMate