МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка iOS застосунку, орієнтованого на автоаматорів**

Виконав: студент 4 курсу, групи 6.1210-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
Д. В. Климов
(ініціали та прізвище)
Керівник к.т.н., доцент, доцент кафедри ЕІС та ПЗ
О. М. Міхайлуца
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел» П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра Електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший(бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Т. В. Критська
“ 01 ” березня 2024 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Климову Данилу Віталійовичу
(прізвище, ім’я, по батькові)

1.Тема роботи Розробка iOS застосунку, орієнтованого на автоаматорів

керівник роботи Міхайлуца Олена Миколаївна, к.т.н., доцент
(прізвище ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від 26.12.2023 № 2215-с

2. Строк подання студентом кваліфікаційної роботи 14.06.2024
3. Вихідні дані кваліфікаційної роботи бакалавра
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
15 слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	03.01 – 25.01.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	26.01 – 31.01.24	виконано
3	Розробка структури та плану роботи	01.02 – 07.02.24	виконано
4	Аналіз існуючих методів рішення	07.02 – 10.02.24	виконано
5	Огляд та збір літератури стосовно теми кваліфікаційної роботи	10.02 – 14.02.24	виконано
6	Огляд та аналіз існуючих рішень та аналогів	14.02 – 18.02.24	виконано
7	Опис предметної області	18.02 – 29.02.24	виконано
8	Огляд мов програмування та інструментів розробки	18.02 – 29.02.24	виконано
9	Огляд архітектурних паттернів та систем керування залежностями	01.03 – 13.03.24	виконано
10	Розробка прототипу додатку	13.03 – 01.04.24	виконано
11	Реалізація функціоналу додатку	01.04 – 31.04.24	виконано
12	Тестування додатку	01.05 – 11.05.24	виконано
13	Оформлення звіту	11.05 – 29.05.24	виконано

Студент _____ Климов Д.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Міхайлуца О.М.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок — 78

Рисунків — 27

Таблиць — 4

Джерел — 25

Климов Д.В. Розробка iOS застосунку, орієнтованого на автоаматорів: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник О. М. Міхайлуца. Запоріжжя : ЗНУ, 2024. 78 с.

Мета даної роботи полягає у розробці зручного і швидкого застосунку, який надасть користувачам можливості вести облік робіт проведених з власним авто та відстеження пробігу автомобіля.

Було проведено аналіз актуальних технологій розробки застосунків для платформи iOS, а саме: мова програмування Swift, фреймворк UIKit та бібліотека RxSwift. Swift обрано як основну мову програмування завдяки її продуктивності, безпеці та зручному синтаксису, що робить її ідеальною для створення додатків під екосистему Apple. UIKit використовується для розробки користувацьких інтерфейсів завдяки своїй стабільності, широкому набору компонентів та підтримці старих версій iOS, що забезпечує сумісність з широким спектром пристроїв. RxSwift інтегрований для управління асинхронними операціями та потоками даних, що дозволяє створювати більш гнучкі та реактивні додатки.

Був проведений аналіз функціональності додатків аналогів, вже присутніх на ринку застосунків. Проведено тестування працездатності розробленого iOS додатку.

Ключові слова: *Swift, UIKit, RxSwift, iOS, авто, контроль пробігу, контроль витрат.*

ABSTRACT

Pages — 78

Drawings — 27

Tables — 4

Sources — 25

Klymov D.V. Development of an iOS application aimed at car enthusiasts: qualification work of a bachelor of specialty 121 "Software engineering" / Science. manager O. M. Mikhailutsa. Zaporizhzhia: ZNU, 2024. 78 p.

The purpose of this work is to develop a convenient and fast application that will give users the opportunity to keep track of the work done on their own car and track the car's mileage.

An analysis of current application development technologies for the iOS platform was conducted, namely: the Swift programming language, the UIKit framework, and the RxSwift library. Swift has been chosen as the primary programming language due to its performance, security, and user-friendly syntax, making it ideal for building applications for the Apple ecosystem. UIKit is used to develop user interfaces due to its stability, extensive set of components and support for older versions of iOS, which ensures compatibility with a wide range of devices. RxSwift is integrated to manage asynchronous operations and data streams, allowing you to create more flexible and reactive applications.

An analysis of the functionality of analog applications already present on the application market was carried out. Performance testing of the developed iOS application was conducted.

Keywords: *Swift, UIKit, RxSwift, iOS, auto, mileage control, expense control.*

ЗМІСТ

ВСТУП

Актуальність теми

У сучасному світі, де кількість автомобілів на дорогах зростає кожного дня, потреби автоаматорів у якісних інструментах для управління та обслуговування своїх транспортних засобів стають дедалі виразнішими.

При цьому, із загальним технологічним прогресом і розвитком мобільних платформ, виникає ростуча потреба у спеціалізованих застосунках, які б забезпечували зручний та ефективний контроль за автомобілем. Особливо великий попит на такі застосунки спостерігається серед власників автомобілів, які вважають свої авто не просто транспортними засобами, а своєрідними улюбленцями. Такі автовласники бажають зберегти гарний стан свого авто якомога довше, ретельно слідкуючи за витратами і регулярним технічним обслуговуванням свого залізного друга.

При цьому, сучасні вимоги до таких мобільних додатків для автомобілістів стають все більш високими. Власники автомобілів очікують від них не лише базових функцій відслідковування та контролю параметрів транспортного засобу, але й інтегрованих можливостей ведення журналу технічного обслуговування, можливості слідкування за пробігом і витратами на своє авто. Такий додаток має стати своєрідним помічником для власників автомобілів, що дозволяє їм ефективно вирішувати різноманітні завдання, пов'язані з управлінням та обслуговуванням авто, забезпечуючи при цьому максимальний рівень комфорту та безпеки.

Станом на 2024 рік у світі налічується понад 1.4 мільярда активних користувачів платформи iOS на мобільних телефонах iPhone. Тобто майже 18% населення нашої планети є щоденними користувачами платформи iOS [1]. А загальна кількість автомобілів у світі на 2024 рік приблизно дорівнює 1.5 мільярда автомобілів на дорогах світу [2].

У зв'язку з цим, розробка спеціалізованого мобільного застосунку для автомобілістів, зокрема для користувачів платформи iOS, має великий потенціал у сучасному ринковому середовищі.

Такий додаток може стати не тільки зручним інструментом для керування авто, а й важливим елементом інфраструктури для автовласників, що забезпечить їм доступ до інноваційних рішень та сервісів у сфері автомобільного обслуговування.

Мета дослідження

Розробка iOS застосунку, орієнтованого на автоаматорів.

Завдання дослідження

Розглянути та порівняти актуальні рішення для розробки iOS застосунку, використовуючи нативні інструменти. Проаналізувати актуальні інструменти і фреймворки розробки iOS додатків та опираючись на результати дослідження обрати інструменти для розробки iOS застосунку орієнтованого на автоаматорів.

Об’єкт дослідження

Об’єктом дослідження є процес розробки iOS застосунку орієнтованого на автоаматорів.

Предмет дослідження

Предметом дослідження є технології створення власного застосунку для задоволення потреб автовласників з метою покращення їх досвіду обслуговування і піклування за власним транспортним засобом.

Методи дослідження

Опитування — проведення опитувань серед автоаматорів для вивчення і аналізу їх потреб та вподобань у обслуговуванні власних авто.

Аналіз існуючих додатків — порівняльний аналіз існуючих мобільних додатків для автоаматорів для виявлення їх переваг і недоліків.

Аналіз засобів розробки — порівняльний аналіз існуючих засобів розробки мобільних додатків на платформі iOS.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає у тому, що розроблений мобільний додаток для платформи iOS надає автовласникам зручний та доступний вибір інструментів для піклування за своїми авто. Використання платформи iOS дозволяє залучити велику кількість автовласників з усього світу. Результати дослідження можуть бути використані для створення інших додатків на платформі iOS з метою поліпшення користувацького досвіду у різних сферах життя.

Апробація результатів

Результати роботи були представлені на XVII університетській науково-практичної конференції студентів, аспірантів, докторантів і молодих вчених «Молода наука-2024» [3].

Глосарій

iOS – операційна система від компанії Apple для мобільних пристроїв.

Swift – сучасна мова програмування, розроблена компанією Apple для створення додатків під iOS, macOS, watchOS та tvOS.

Objective-C – мова програмування, яка використовувалася для розробки додатків під платформи Apple до появи Swift.

Dart – мова програмування, розроблена компанією Google.

Кроссплатформеність – здатність програмного забезпечення працювати на декількох платформах або операційних системах без значних змін у коді.

XCode – інтегроване середовище розробки (IDE) від Apple, яке забезпечує повний набір інструментів для створення, тестування та налагодження додатків під платформи Apple.

Фреймворк – програмне забезпечення, що забезпечує основу та стандартні функції для розробки додатків, дозволяючи розробникам зосередитися на написанні специфічної для їх проекту логіки.

UIKit – фреймворк для створення користувацьких інтерфейсів в iOS додатках.

SwiftUI – сучасний фреймворк від Apple для створення користувацьких інтерфейсів, який використовує декларативний підхід до програмування.

MVC (Model-View-Controller) – архітектурний паттерн, що розподіляє додаток на три компоненти: модель, представлення та контролер.

MVP (Model-View-Presenter) – архітектурний паттерн, що розподіляє додаток на три компоненти: модель, представлення та презентер.

MVVM (Model-View-ViewModel) – архітектурний паттерн, що розподіляє додаток на три компоненти: модель, представлення та ViewModel.

VIPER (View, Interactor, Presenter, Entity, Router) – архітектурний паттерн, який забезпечує чітке розділення відповідальностей у додатку між компонентами View, Interactor, Presenter, Entity та Router.

CocoaPods – інструмент для управління залежностями в проектах iOS, що забезпечує автоматичну інтеграцію сторонніх бібліотек у проект.

Swift Package Manager (SPM) – офіційний інструмент управління пакетами від Apple, інтегрований безпосередньо у Swift та Xcode.

Carthage – інструмент для управління залежностями, який надає більше контролю над процесом інтеграції бібліотек.

RxSwift – бібліотека для функціонального реактивного програмування на мові Swift.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд літературних джерел

У спільноті iOS розробників велика увага приділяється вибору та використанню сучасних підходів, фреймворків розробки, архітектурних паттернів та систем керування залежностями. Ці аспекти є критично важливими для створення ефективних, продуктивних та масштабованих додатків. Розробка мобільного додатку для управління автомобільними витратами та ведення журналу проведених технічних робіт вимагає глибокого розуміння основних підходів та компонентів розробки сучасних мобільних додатків, а також аналізу існуючих рішень та методів, які використовуються в індустрії. При підготовці дипломної роботи, був проведений огляд літературних джерел, що допомогло розширити розуміння теми створення мобільних застосунків на платформі iOS та виявити підходи, що вже активно використовуються у цій сфері.

Був проведений збір та аналіз актуальних літературних джерел, які стали фундаментальною інформаційною базою під час розробки додатку орієнтованого на автоаматорів на платформі iOS. Методологія відбору літературних джерел включає використання академічних журналів, онлайн ресурсів та онлайн бібліотек. Головними критеріями під час відбору джерел стали: актуальність інформації та авторитетність автора.

Літературні джерела були піддані детальному аналізу, їх зосередженість до основних тем розробки у сфері створення мобільних додатків на платформі iOS. Був врахований вплив окремих підходів у розробці відносно їх впливу на ефективність і актуальність розробки.

Основними темами, під час аналізу літературних джерел, стали мови програмування для створення мобільних застосунків на платформі iOS, дослідження різних архітектурних паттернів для забезпечення максимальної ефективності та зручності під час розробки. Додатково були проаналізовані

системи управління залежностями, для зручного керування підключенням сторонніх бібліотек.

Основою для створення додатку на платформі iOS стала мова програмування Swift, що вказує на важливість аналізу матеріалів пов’язаних з інформацією щодо розробки застосунків на цій мові.

1.2 Аналіз програмних продуктів-аналогів

Оскільки сучасний світ мобільних додатків зростає с кожним днем і знаходиться на особистому піці популярності, вже існує певна кількість додатків аналогів з гарними відгуками та підтримкою iOS. Я обрав декілька з них, які на мою думку мають певні переваги над іншими додатками.

Пропоную порівняти деякі з них, а саме: My car, CarKeep, Drivvo.

Порівняння характеристик цих аналогів можна побачити в таблиці 1.

MyCar

MyCar — це сучасний автомобільний додаток нового покоління для відстеження споживання пального, заправок, пробігу, послуг та витрат вашого автомобіля [4]. Додаток надає можливість вносити останні заправки автомобіля, дописуючи пробіг авто і перегляди графіки з минулими заправками і пробігами. Додаток є досить популярним серед користувачів і немає гарні відгуки в AppStore.

Інтерфейс додатку MyCar продемонстрований на рисунку 1.

Переваги:

- Надає можливість докладно вести журнал записів про пробіг та заправку власного авто, включаючи витрати на пальне.
- Пропонує різноманітну вибірку з місць обслуговування власного авто.
- Надає можливість прикріпляти чеки за покупку до кожного запису з витратами.
- Надає можливість переглядати графіки витрат і пробігу авто.
- Наявна українська локалізація.

Недоліки:

- Наявність реклами.
- Не інтуїтивний дизайн з не працюючими елементами.
- Обмежений функціонал у безкоштовній версії додатку.

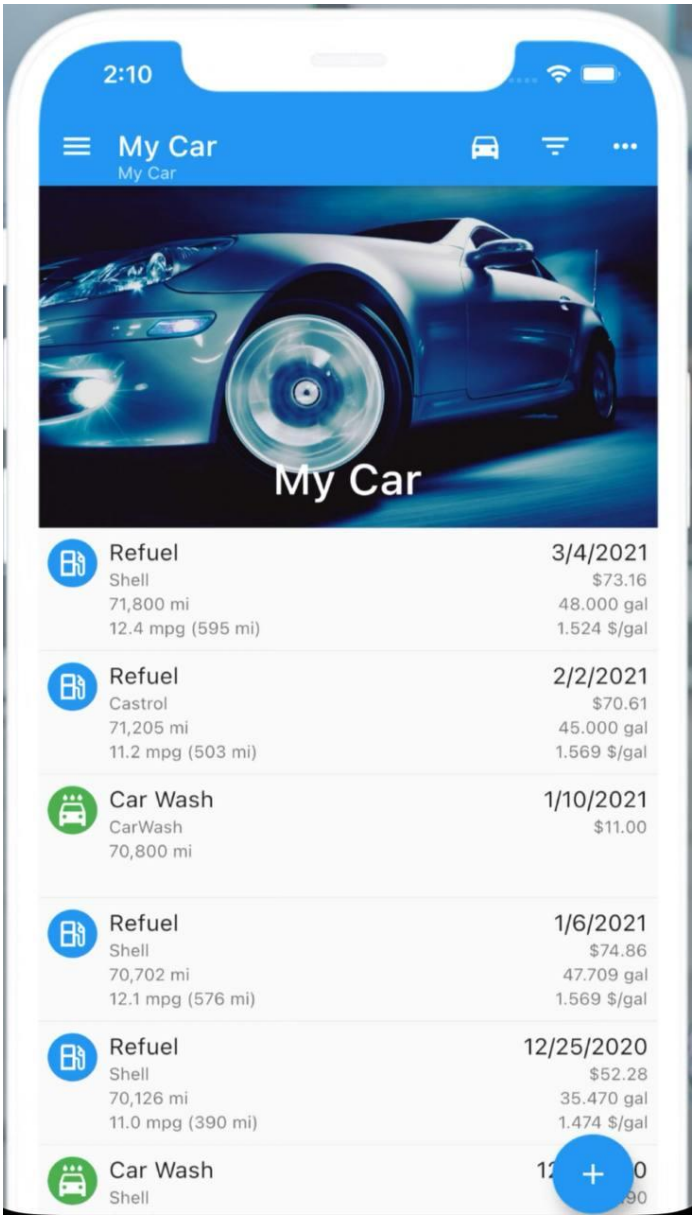

Рисунок *1 — Інтерфейс MyCar*

CarKeep

CarKeep — це надійний інструмент керування функціями автомобіля, розроблений для спрощення технічного обслуговування та відстеження витрат. Завдяки своїм функціям, надає можливість стежити за станом авто [5].

Інтерфейс додатку CarKeep демонструється на рисунку 2.

Рисунок *2 — Інтерфейс CarKeep*

Переваги:

- Зручний та інтуїтивно зрозумілий інтерфейс.
- Можливість ведення журналу записів про стан свого авто.
- Маленькій розмір застосунку 22mb.
- Наявна власна досить широка база с марок та моделей авто.

Недоліки:

- Обмежений функціонал у безкоштовній версії додатку.
- Відсутність додаткових локалізацій. Обмежена лише англійською.

Drivvo

Drivvo — Реєструйте, впорядковуйте та відстежуйте всю інформацію про вашому автомобілі, мотоциклі, вантажівці, автобусі або автопарку в будь-який час і в будь-якому місці і де б ви не знаходилися [6].

Інтерфейс додатку Drivvo продемонстрований на рисунку 3.

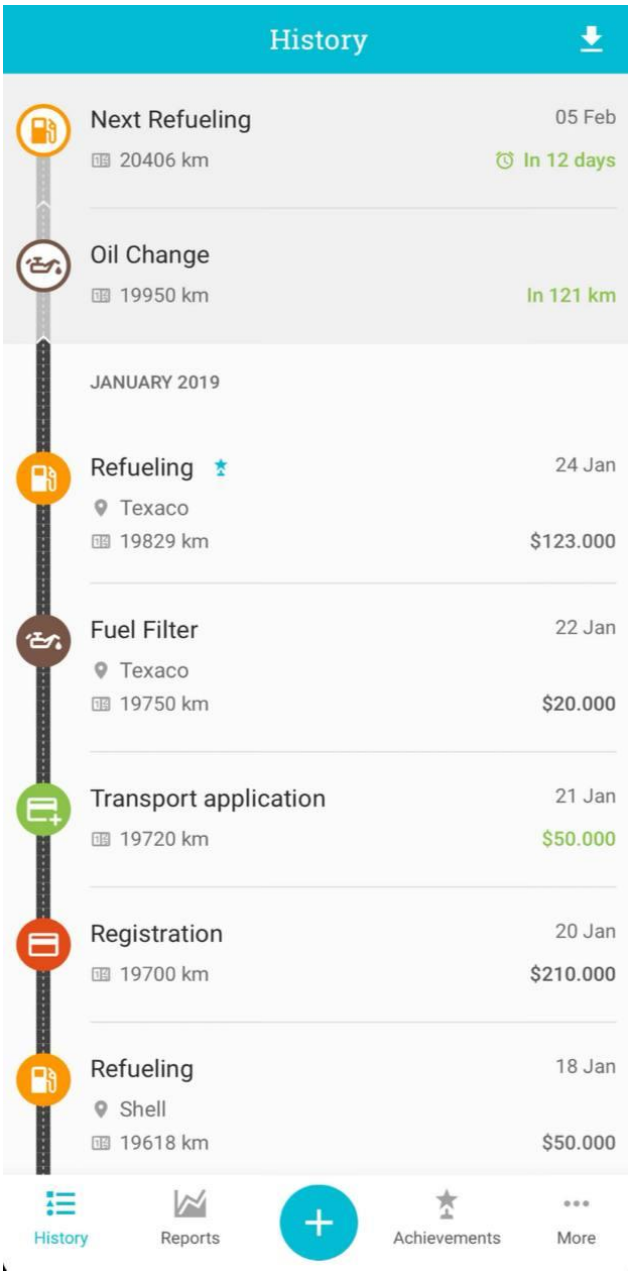

Рисунок *3* — *Інтерфейс Drivvo*

Переваги:

- Можливість ведення журналу записів про стан свого авто.

- Простий та зрозумілий інтерфейс.
- Можливість перегляди графіки пробігу авто.
- Наявна власна система досягнень.

Недоліки:

- Відсутність додаткових локалізацій. Обмежена лише англійською.
- Наявність реклами.

Таблиця *1 — Порівняння характеристик MyCar, Car Keep та Drivvo*

Властивість	**MyCar**	**Car Keep**	**Drivvo**
База даних	Відсутня	Обширна	Відсутня
Базовий функціонал	Журнал пробігу авто, калькулятор розходу палива	Калькулятор розходу палива, журнал пробігу авто, нагадування про заплановані технічні роботи	Журнал пробігу авто, нагадування про заплановані технічні роботи
Персоналізація та аналітика	Відсутня	Високий рівень	Відсутня
Підтримка мов	Наявна	Відсутня	Наявна
Вартість	Безкоштовний	Платний $3.99	Безкоштовний
Платна підписка	Наявна, $2/місяць	Відсутня	Наявна, $1/місяць
Функції платної підписки	Відсутність реклами, додатковий функціонал	Відсутня платна підписка	Відсутність реклами

Отже, опираючись на порівняння наявних додатків на ринку мобільних застосунків для платформи iOS, а саме MyCar, CarKeep, Drivvo, які розроблені з метою покращення досвіду володіння автомобілем і піклуванням за його станом, можна зробити висновок про потребу розробки додатку, який забезпечить більш зручний та надійний функціонал для піклування про власне авто, з відсутніми недоліками існуючих додатків.

1.3 Постановка задачі

У сучасному світі мобільні додатки стали незмінними помічниками в багатьох сферах нашого життя, кожного дня ми користуємось мобільними телефонами і навіть не помічаємо цього. Навіть у піклування про власне авто є потреба у надійному і корисному помічнику, яким і має стате мобільний додаток. Важливість контролю витрат на обслуговування автомобілів, піклування за їх технічним станом зростає, особливо в умовах швидкого розвитку технологій.

Саме з метою покращення досвіду володіння і використання власного авто стоїть задача розробки сучасного, швидкого і зручного додатку, який надасть користувачам можливість стежити за технічним станом авто, відстежувати його пробіг, витрати на обслуговування та перегляди корисні матеріали, які покращать розуміння роботи транспортного засобу.

Основні задачі, які необхідно вирішити підчас розробки такого додатку:

1. Реєстрація та вхід користувачів:

Забезпечити користувачів можливість створювати нові облікові записи, або увійти у вже існуючий обліковий запис, який був створений раніше. Цей процес має бути розділений на окремі екрани, щоб користувачу було більш зрозуміло, що саме він робить у даний момент, а саме: головна сторінка для обрання методу авторизації у обліковий запис, екран авторизації у обліковий запис, екран створення нового облікового запису, екран для відновлення паролю для існуючого облікового запису.

2. Персоналізація облікового запису:

Надання можливості користувачу змінювати особисті дані про себе та про своє авто, а саме: ім’я, електронна пошта, стать, бренд авто, марка авто, рік випуску авто. Крім того необхідно надати користувачу можливість обрати бренд та марку авто з наявного списку.

3. Журнал обліку технічних робіт та пробігу авто:

Додаток повинен мати у собі функціонал створення записів про технічні роботи, проведені з автомобілем з можливість вказати корисну інформацію та суму затрачених коштів. Мати можливість створювати записи про пробіг автомобіля, який неупинно збільшуватиметься з використанням транспортного засобу.

4. Додаткова інформація:

Додаток має надати користувачу можливість ознайомитись з корисною інформацію у форматі відео або веб-статей. Потрібно розробити інтерактивну мапу, на якій користувач зможе переглядати найближчі до себе авто заправні станції або станції технічного обслуговування.

5. Інтерфейс застосунку:

Інтерфейс застосунку є однією з найголовніших частин застосунку, бо саме його користувач буде бачити кожного разу коли запускатиме додаток. Інтерфейс має бути інтуїтивно зрозумілим, мати текстові підказки та відповідати сучасним трендам.

6. Безпека даних:

Забезпечення особистих даних користувачів є не менш важливою задачею. Потрібно використовувати надійні та сучасні методи шифрування, та аутентифікації користувачів. Розробка політики конфіденційності та заходів безпеки.

7. Адаптивність:

Адаптивність мобільного додатку полягає у його здатності ефективно працювати на різних моделях iPhone з різними розмірами екранів та технічними характеристиками. Застосунок повинен забезпечувати оптимальне відображення і функціонування на пристроях з будь-якою роздільною здатністю, забезпечуючи однаковий користувацький досвід для всіх користувачів. Також важливо враховувати можливість подальшого масштабування мобільного додатку та його функціоналу.

2 ДОСЛІДЖЕННЯ ЗАСОБІВ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ МОБІЛЬНОГО ДОДАТКУ

2.1 Огляд мов програмування

Мови програмування є основним інструментом для розробки програмного забезпечення, включаючи й сучасні мобільні додатки. Вони забезпечують засоби для опису алгоритмів, маніпуляції даними та створення користувацьких інтерфейсів. У цьому розділі буде розглянуто основні мови програмування для платформи iOS, що використовуються для розробки мобільних додатків, з акцентом на їх особливості, переваги та недоліки.

Порівняння характеристик цих мов можна побачити в таблиці 2.

Dart

Dart — це мова програмування, розроблена компанією Google, яка використовується для створення веб- та мобільних додатків [7]. Dart призначена для того, щоб бути продуктивною мовою з простим і зрозумілим синтаксисом, яка підтримує як об'єктно-орієнтоване, так і функціональне програмування. Найчастіше Dart асоціюється з фреймворком Flutter [8], який використовується для розробки кросплатформених мобільних додатків.

Переваги:

- Продуктивність: Dart компілюється в нативний код, що забезпечує високу швидкість виконання та оптимізацію продуктивності. Завдяки AOT-компіляції додатки на Dart запускаються швидко і ефективно.
- Кросплатформеність: Використання Dart у поєднанні з Flutter дозволяє створювати додатки для iOS, Android, Web та декстопних платформ з єдиною кодовою базою.
- Зрозумілий синтаксис: Dart має простий і зрозумілий синтаксис, що робить його легким у вивченні та використанні для розробників з різним рівнем досвіду.

- Асинхронне програмування: Вбудована підтримка асинхронного програмування спрощує роботу з операціями вводу/виводу та іншими асинхронними процесами.
- Велика стандартна бібліотека: Dart поставляється з багатою стандартною бібліотекою, яка включає в себе різноманітні функції для роботи з колекціями, потоками, файловою системою та багато іншого.

Недоліки:

- Відносно нова мова: Dart все ще відносно нова мова програмування, тому може бути обмежена підтримка деяких специфічних функцій платформ та наявність бібліотек.
- Менша спільнота: Порівняно з більш популярними мовами програмування, такими як Swift або Java, спільнота розробників Dart є меншою, що може вплинути на доступність ресурсів та прикладів коду.
- Залежність від Flutter: Незважаючи на те, що Dart можна використовувати самостійно, найбільш популярним є його використанням з Flutter. Це може створити враження, що Dart менш універсальний і знаходиться поза контекстом мобільної розробки без Flutter.
- Не нативна мова: Dart не є нативною мовою для iOS або Android, що може викликати деякі обмеження у використанні специфічних функцій платформ та зменшити продуктивність порівняно з нативними рішеннями, написаними на Swift або Kotlin.

Objective-C

Objective-C — це мова програмування, яка була основною для розробки додатків під iOS і macOS до появи Swift [9]. Вона базується на мові C та має об'єктно-орієнтоване розширення. Objective-C була розроблена в 1980-х роках компанією NeXT, яку пізніше придбала Apple [10]. Ця мова була основним інструментом для створення програмного забезпечення для екосистеми Apple до 2014 року, коли Apple представила Swift як нову мову програмування.

Переваги:

- Сумісність зі старими проектами: Objective-C має високу сумісність з існуючими проектами та бібліотеками, що робить її ідеальною для підтримки та розвитку старих додатків.
- Велика кількість готових бібліотек: Існує велика кількість готових бібліотек та фреймворків, які були розроблені для Objective-C за час його існування.
- Глибока інтеграція з C та C++: Objective-C дозволяє легко інтегрувати код, написаний на C та C++, що забезпечує високу продуктивність і можливість використовувати низькорівневі функції.
- Гнучкість програмування: Objective-C підтримує динамічне введення і типізацію, що дозволяє розробникам писати більш гнучкий код.

Недоліки:

- Складний синтаксис: Синтаксис Objective-C є складнішим та менш зрозумілим порівняно зі Swift, що може погіршити процес навчання для нових розробників.
- Небезпека під час розробки: Objective-C менш безпечний у порівнянні зі Swift через менш строгий підхід до типізації та управління пам'яттю.
- Менш активна спільнота: З появою Swift, більшість розробників перейшли на нову мову, тому спільнота Objective-C стала менш активною, що може ускладнити пошук допомоги та ресурсів.
- Не сучасна мова: Objective-C не підтримує багато сучасних можливостей, таких як обробка помилок за допомогою try/catch, опціональні типи та сучасний синтаксис, які присутні в Swift.

Swift

Swift — це мова програмування, розроблена компанією Apple, яка використовується для створення додатків під iOS, macOS, watchOS та tvOS [11]. Мова Swift була представлена у 2014 році як сучасна альтернатива Objective-C, з акцентом на безпеку, продуктивність та зручність використання [12]. Swift поєднує в собі найкращі риси різних мов програмування, таких як

Python, Ruby та C#, і є відкритою мовою, що сприяє її розвитку та поширенню у спільноті програмістів.

Переваги:

- Продуктивність: Swift компілюється в нативний код, що забезпечує високу швидкість виконання та продуктивність. Оптимізації компілятора дозволяють створювати високопродуктивні додатки.
- Безпека: Swift підтримує строгий контроль типів і управління пам'яттю, що допомагає уникнути багатьох типових помилок програмування, таких як нульові посилання.
- Зрозумілий синтаксис: Swift має сучасний, читабельний синтаксис, який робить код більш зрозумілим та легким у підтримці. Це полегшує процес навчання для нових розробників.
- Підтримка сучасних функцій: Swift підтримує багато сучасних можливостей, таких як обробка помилок, опціональні типи, замикання, а також функціональне програмування та інші.
- Активна підтримка і розвиток: Apple активно підтримує та розвиває Swift, регулярно випускаючи нові версії та покращення. Swift також має велику та активну спільноту розробників з усього світу.
- Інтеграція: Swift легко інтегрується з Objective-C, що дозволяє використовувати існуючі бібліотеки та фреймворки, написані на Objective-C, C та C++.

Недоліки:

- Відносно нова мова: Незважаючи на швидке поширення, Swift все ще є відносно новою мовою, тому деякі розробники можуть стикатися з обмеженою підтримкою деяких специфічних функцій або бібліотек.
- Зміни в синтаксисі: У ранніх версіях Swift синтаксис мови зазнавав значних змін, що могло ускладнити підтримку коду. Однак останні версії стали більш стабільними.

Таблиця *2 — Порівняння характеристик мов програмування Dart, Objective-C та Swift*

Властивість	**Dart**	**Objective-C**	**Swift**
Розробник	Google	NeXT, Apple	Apple
Синтаксис	Простий та зрозумілий	Базується на C, складніший та менш інтуїтивний	Сучасний, простий та легко читабельний
Парадигма програмування	Об'єктно-орієнтоване, функціональне	Об'єктно-орієнтоване	Об'єктно-орієнтоване, функціональне, протоколо-орієнтоване
Підтримка платформи iOS	Обмежена підтримка через Flutter	Повна підтримка з перших версій iOS	Повна підтримка з iOS 7
Компіляція	Just-In-Time (JIT) та Ahead-Of-Time (AOT)	Компіляція в рідний код	Компіляція в рідний код
Продуктивність	Висока, але залежить від Flutter	Висока	Висока
Навчання	Легка для вивчення	Складне через синтаксис та концепції	Легка для вивчення
Підтримка асинхронності	Зручна у використанні	Складна у використанні	Зручна у використанні
Спільнота	Спільнота зростає і вже має великий обсяг активних користувачів	Спільнота зменшується, через зріст популярності Swift, спільнота знаходиться на низькому рівні активності	Велика та активна спільнота користувачів.
Стабільність	Відносна стабільна, через активний розвиток	Стабільна, але застаріла	Незважаючи на активний розвиток, стабільна

Документація	Хороша документація	Вичерпна, велика кількість застарілих ресурсів	Вичерпна документація, активне оновлення
Кросплатформеність	Висока	Обмежена	Тільки екосистема Apple

Після порівняння основних мов програмування, що використовуються для розробки мобільних додатків на платформі iOS, можна зробити такі висновоки:

- Dart: Найкраще підходить для кросплатформеної розробки за допомогою Flutter. Має сучасний та зрозумілий синтаксис, але обмежена підтримка iOS робить його менш підходящим для нативної iOS розробки.
- Objective-C: Стабільна та перевірена часом мова, яка все ще використовується для підтримки старих проектів. Складний синтаксис і застарілість роблять її менш привабливою для нових проектів.
- Swift: Найсучасніша та найкраще підтримувана мова для розробки додатків під екосистему Apple, а саме платформу iOS. Пропонує високу продуктивність, простий синтаксис та активну підтримку з боку Apple. Є оптимальним вибором для нових проектів під iOS.

Отже Swift є оптимальним вибором для створення додатків під екосистему Apple. Ця мова програмування поєднує в собі сучасні можливості, високу продуктивність та безпеку, що робить її ідеальним інструментом для розробки високоякісного програмного забезпечення. Саме через ці переваги була обрана мова програмування Swift.

2.2 Огляд інструментів розробки

Інструменти розробки відіграють ключову роль у процесі створення сучасних мобільних додатків, забезпечуючи розробникам необхідні засоби для написання, тестування та налагодження коду. Використання правильних інструментів може значно підвищити ефективність роботи, зменшити

кількість помилок та скоротити час, необхідний для розробки додатків. У цьому розділі буде розглянуто основні інструменти, які використовуються для розробки мобільних додатків для платформи iOS, з акцентом на їх особливості, переваги та недоліки.

XCode

XCode — це інтегроване середовище розробки (IDE), розроблене компанією Apple для створення додатків під iOS, macOS, watchOS та tvOS [13]. Xcode надає розробникам потужні інструменти для написання, тестування та налагодження коду, забезпечуючи повний цикл розробки додатків для всіх платформ Apple. Завдяки своїм багатим можливостям та тісній інтеграції з екосистемою Apple, Xcode є незамінним інструментом для розробників додатків для платформ Apple.

Основні можливості:

1. Редактор коду: Xcode пропонує зручний та інтуїтивно зрозумілий редактор коду з підтримкою підсвічування синтаксису, автодоповнення, навігації по коду та інших функцій, що підвищують продуктивність розробки.
2. Інструменти налагодження: Xcode містить потужні інструменти для налагодження, включаючи інтерактивний відладчик, підтримку точок зупинки, можливість перегляду значень змінних в реальному часі та аналізу стеку викликів.
3. Interface Builder: Interface Builder в Xcode дозволяє розробникам створювати користувацькі інтерфейси за допомогою UI інтерфейсу, забезпечуючи візуальне конструювання інтерфейсів без необхідності писати код вручну.
4. Симулятори: Xcode надає можливість тестувати додатки на віртуальних пристроях, що емулюють різні моделі iPhone, iPad, Apple Watch та Apple TV. Це дозволяє розробникам перевіряти роботу додатків на різних пристроях і версіях операційних систем.

5. Інструменти тестування: Вбудовані інструменти для написання та виконання тестів, включаючи Unit Tests та UI Tests, дозволяють автоматизувати процес тестування і забезпечувати високу якість додатків.

6. Інтрументи продуктивності: Instruments, інтегровані в Xcode, дозволяють аналізувати продуктивність додатків, виявляти витоки пам'яті, аналізувати використання процесора і графічного процесора, що допомагає оптимізувати додатки.

7. Інтеграція з Git: Xcode підтримує інтеграцію з системами контролю версій, такими як Git, що дозволяє ефективно відстежувати зміни в коді, співпрацювати з командою та керувати версіями проектів.

Xcode є потужним і незамінним інструментом для розробки додатків під екосистему Apple. Завдяки своїй глибокій інтеграції з платформами Apple, багатому набору інструментів для написання, налагодження та тестування коду, Xcode дозволяє розробникам створювати високоякісні додатки, які повністю використовують можливості апаратного та програмного забезпечення Apple. Незважаючи на певні недоліки, такі як високі вимоги до апаратного забезпечення, Xcode залишається провідним вибором для розробників, що працюють в екосистемі Apple.

UIKit

UIKit — це сучасний фреймворк, розроблений компанією Apple, який використовується для створення користувацьких інтерфейсів у додатках під iOS, iPadOS, tvOS та watchOS [14]. UIKit надає широкий набір класів та інструментів для створення інтерактивних і візуально привабливих інтерфейсів, забезпечуючи високий рівень інтеграції з апаратними та програмними можливостями пристроїв Apple.

Основні можливості:

1. Компоненти інтерфейсу: UIKit включає в себе великий набір готових компонентів інтерфейсу, таких як кнопки, текстові поля, таблиці, колекції та інші елементи, що дозволяють швидко створювати складні та функціональні інтерфейси.

2. Керування подіями: Фреймворк надає засоби для обробки користувацьких подій, таких як дотики, жести, натискання кнопок та інші дії, що забезпечує високу інтерактивність додатків.

3. Автоматичне розташування (Auto Layout): UIKit підтримує Auto Layout, що дозволяє створювати адаптивні інтерфейси, які автоматично підлаштовуються під різні розміри екранів та орієнтації пристроїв.

4. Анімації: Фреймворк надає потужні інструменти для створення анімацій, що дозволяють оживити інтерфейс і зробити взаємодію з користувачем більш привабливою та інтуїтивною.

5. Підтримка різних типів навігації: UIKit забезпечує різні схеми навігації, включаючи таб-бар, навігаційні контролери та сторінкову навігацію(segue), що допомагає створювати зручні та логічні структури додатків.

SwiftUI

SwiftUI — це сучасний фреймворк для розробки користувацьких інтерфейсів, розроблений компанією Apple [15]. Він був представлений на WWDC 2019 як декларативний підхід до створення інтерфейсів для iOS, macOS, watchOS та tvOS [16]. SwiftUI дозволяє розробникам описувати інтерфейси за допомогою простого та інтуїтивно зрозумілого коду, що значно спрощує процес розробки та знижує обсяг необхідного коду

Основні можливості:

1. Декларативний синтаксис: SwiftUI використовує декларативний підхід до створення інтерфейсів, що дозволяє розробникам описувати, що саме має бути відображено на екрані, без необхідності писати код для оновлення інтерфейсу вручну.

2. Кодовий базис: SwiftUI дозволяє використовувати єдиний код для створення додатків під iOS, macOS, watchOS та tvOS, що значно спрощує процес розробки кросплатформених додатків.

3. Анімовані переходи та ефекти: Фреймворк надає прості засоби для створення анімацій та переходів, що робить інтерфейси більш динамічними та привабливими для користувачів.

4. Прев'ю в реальному часі: Xcode дозволяє переглядати зміни в інтерфейсі в режимі реального часу завдяки SwiftUI Preview, що значно прискорює процес розробки та тестування інтерфейсів.

5. Підтримка сучасних можливостей iOS: SwiftUI підтримує багато сучасних можливостей iOS, таких як Dark Mode, Accessibility, та інші, що забезпечує високий рівень зручності та доступності додатків.

6. Сумісність з UIKit: SwiftUI може бути легко інтегрований з UIKit, що дозволяє розробникам використовувати переваги обох фреймворків в одному додатку.

Обидва фреймворки, UIKit та SwiftUI, мають свої унікальні переваги та недоліки, що впливають на вибір інструменту для розробки додатків під екосистему Apple. Кожен з них пропонує різні підходи до створення користувацьких інтерфейсів, і вибір між ними залежить від конкретних потреб проекту, досвіду розробників та вимог до підтримки різних версій операційних систем.

Враховуючи основні можливості описані вище, UIKit залишається найбільш доцільним вибором для багатьох проектів, що потребують стабільності, гнучкості та широкої сумісності. Незважаючи на перспективність SwiftUI, UIKit продовжує забезпечувати надійну основу для створення високоякісних додатків під екосистему Apple та платформи iOS.

RxSwift

RxSwift — це бібліотека для функціонального реактивного програмування на мові Swift, яка базується на концепціях Reactive Extensions (Rx) [17, 18]. Вона дозволяє створювати асинхронні програми, що реагують на потоки подій, використовуючи декларативний підхід. RxSwift широко використовується для роботи з асинхронними даними та подіями, що дозволяє значно спростити управління станом додатків та обробку складних асинхронних операцій.

Основні можливості:

1. Observables: Основною концепцією в RxSwift є observables, які представляють собою потоки даних або подій. Вони можуть випромінювати події, такі як next, error, та completed, які обробляються підписниками.
2. Оператори: RxSwift надає широкий набір операторів для трансформації, фільтрації, комбінування та обробки observables. Це дозволяє створювати складні асинхронні потоки даних за допомогою декларативного синтаксису.
3. Суб'єкти (Subjects): Суб'єкти в RxSwift є одночасно observable та observer. Вони можуть випромінювати нові події та передавати їх підписникам, що робить їх корисними для управління станом та взаємодії між різними частинами програми.
4. Disposables: RxSwift використовує концепцію disposables для управління життєвим циклом підписок. Це дозволяє автоматично відписуватися від observables, коли вони більше не потрібні, запобігаючи витокам пам'яті.
5. Сумісність з UIKit: RxSwift легко інтегрується з UIKit, що дозволяє використовувати реактивний підхід для управління інтерфейсом користувача, взаємодії з користувачем та обробки подій.

RxSwift є потужною бібліотекою для реактивного програмування на мові Swift, що дозволяє ефективно працювати з асинхронними даними та потоками подій. Його декларативний підхід, широкий набір операторів та можливість інтеграції з UIKit роблять його привабливим вибором для розробників, що прагнуть створювати складні та реактивні додатки. Незважаючи на складну практику навчання та потенційні складнощі з налагодженням, RxSwift забезпечує високу гнучкість та ефективність у роботі з асинхронними операціями, що робить його важливим інструментом для сучасної мобільної розробки.

2.3 Огляд архітектурних паттернів

Важливість обрання архітектурного паттерну не можна переоцінити, оскільки він визначає структуру додатку, розподіл залежностей між його

компонентами та взаємодію між ними. Правильно обраний паттерн сприяє зручності підтримки та розширення додатку, підвищенню його продуктивності та надійності. Якщо на етапі створення проекту обрати не доцільний архітектурний паттерн, це може призвести до труднощів у розробці, зниження якості коду та виникнення труднощів при впровадженні нових функцій, або розширенні додатку**.**

Архітектурні паттерни є скелетом програми при проектуванні програмного забезпечення, оскільки вони забезпечують стандартизований підхід до вирішення типових задач. Вони дозволяють розробникам використовувати перевірені часом методики, що знижують ризик помилок та сприяють створенню більш надійного та ефективного коду.

На даний момент часу існує велика кількість актуальних, перевірених часом архітектурних паттернів, але у цьому розділі буде розглянуто основні з них, які використовуються у розробці мобільних додатків, зокрема для платформи iOS, а саме: Model-View-Controller (MVC), Model-View-Presenter, Model-View-ViewModel (MVVM), VIPER (View, Interactor, Presenter, Entity и Router).

Model View Controller

Model-View-Controller (MVC) є одним з найперших і найпоширеніших архітектурних паттернів, що використовуються в розробці програмного забезпечення. Після анонсу мови програмування Swift, сама компанія Apple рекомендувала використовувати саме цей архітектурний паттерн, при створені власних застосунків [19].Основна ідея MVC полягає у розподілі додатку на три основні компоненти:

- Model (Модель): Відповідає за управління даними. Модель отримує, зберігає та маніпулює даними, а також повідомляє контролеру про зміни в даних.
- View (Представлення): Відповідає за відображення даних користувачу. Представлення отримує дані від моделі і відображає їх у потрібному форматі.

Важливо також пам’ятати, що представлення не містить бізнес-логіки і не змінює дані.

- Controller (Контролер): Відповідає за обробку вводу користувача і взаємодію між моделлю та представленням. Контролер отримує події від представлення, обробляє їх і оновлює представлення відповідно до змін у моделі. Схему архітектури MVC продемонстрована на рисунку 4.

Переваги MVC:

1. Чіткий розподіл відповідальностей та обов’язків

2. Масштабованість, кожен компонент може бути змінений та розширений незалежно від інших.

3. Розділення логіки додатку між моделлю, представленням та контролером спрощує процес тестування.

4. Можливість повторного використання коду.

Недоліки MVC:

1. У великих проектах кількість контролерів може були дуже великою, що може ускладнити управління та навігацію по коду.

2. Іноді контролери можуть стати занадто великими і містити бізнес-логіку, що порушує принцип розділення відповідальностей.

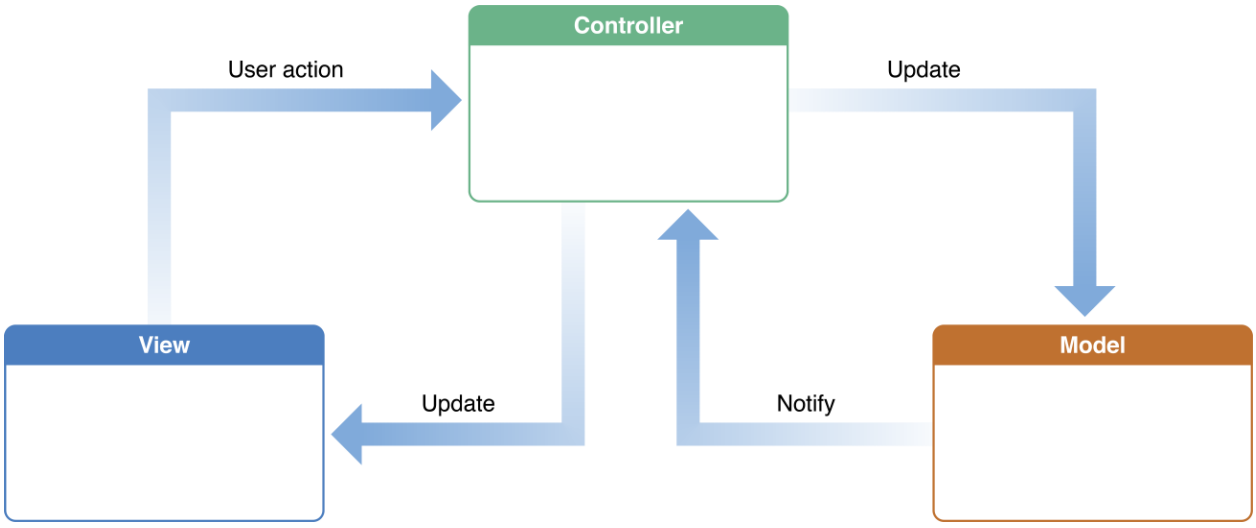

Рисунок *4 — Схема архітектури MVC*

Model View Presenter

Model-View-Presenter (MVP) розподіляє відповідальності в додатку на три основні компоненти: Model, View та Presenter [20]. MVP виник як еволюція паттерну MVC з метою розв'язання деяких його недоліків, зокрема перенавантаження контролерів бізнес-логікою у великих додатках.

Основні компоненти:

- Model (Модель): Відповідає за управління даними додатку. Модель отримує, зберігає та маніпулює даними, а також повідомляє презентер про зміни в даних.
- View (Представлення): Відповідає за відображення даних користувачеві отриманих від презентера у зручному форматі. Представлення не містить бізнес-логіки та взаємодіє з користувачем.
- Presenter (Презентер): Відповідає за взаємодію між моделлю та представленням, отримує дані від моделі та обробляє їх для відображення у представленні. Він також обробляє дії користувача, викликаючи відповідні методи. Схему архітектури MVP продемонстрована на рисунку 5.

Переваги MVP:

1. Чітке розділення відповідальностей.

2. Презентер не залежить від життєвого циклу представлення, це дозволяє легко тестувати кожен елемент окремо.

3. Завдяки чіткому розділенню обов'язків між моделлю, представленням та презентером, код стає більш структурованим і легким у підтримці та розширенні.

4. Модульність: MVP дозволяє створювати модульні компоненти, які легко замінювати або оновлювати без впливу на інші частини додатку.

Недоліки MVP:

1. Розподіл обов'язків між трьома компонентами може викликати збільшення кількості коду.

2. Хоча презентер вирішує проблему перенавантаження контролерів у MVC, він сам може стати занадто великим та складним.

3. Велика кількість залежностей може призвести до проблем з розумінням коду.

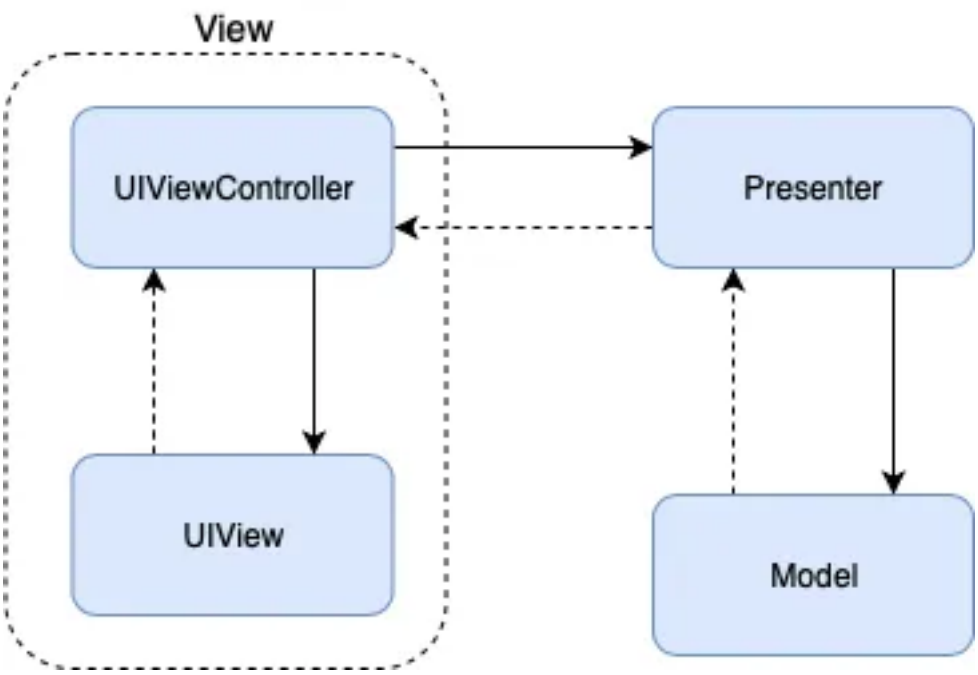

Рисунок 5 — *Схема архітектури MVP*

Model View ViewModel

Model-View-ViewModel (MVVM) розділяє додаток на три основні компоненти: Model, View та ViewModel [21]. MVVM є еволюцією паттерну MVC і надає більш чіткий розподіл обов'язків та полегшує двосторонню прив'язку даних між моделлю та представленням.

Основні компоненти:

- Model (Модель): Відповідає за управління даними додатку, а також повідомляє ViewModel про зміни в даних.
- View (Представлення): Відповідає за відображення даних користувачеві. Представлення отримує дані від ViewModel і відображає їх у зручній форматі для взаємодії з користувачем, не містить бізнес-логіки.
- ViewModel (Модель представлення): Відповідає за взаємодію між моделлю та представленням. ViewModel отримує дані від моделі, обробляє їх та надає представленню, він також обробляє дії користувача, оновлюючи

модель та представлення відповідно. Схему архітектури MVVM продемонстрована на рисунку 6.

Переваги MVVM:

1. Чітке розділення відповідальностей.

2. Двостороннє зв'язування даних між моделлю та представленням, що значно спрощує синхронізацію стану додатку.

3. Полегшене тестування взаємозв’язаних компонентів.

4. MVVM дозволяє створювати модульні компоненти, які легко розширювати та підтримувати.

Недоліки MVVM:

1. MVVM може ускладнити архітектуру додатку, особливо у випадку невеликих проектів, де його використання може бути надлишковим і тільки ускладнити проект.

2. Реалізація двостороннього зв'язування даних та розподілу обов'язків може бути складним для розуміння.

3. ViewModel може стати занадто великим і складним.

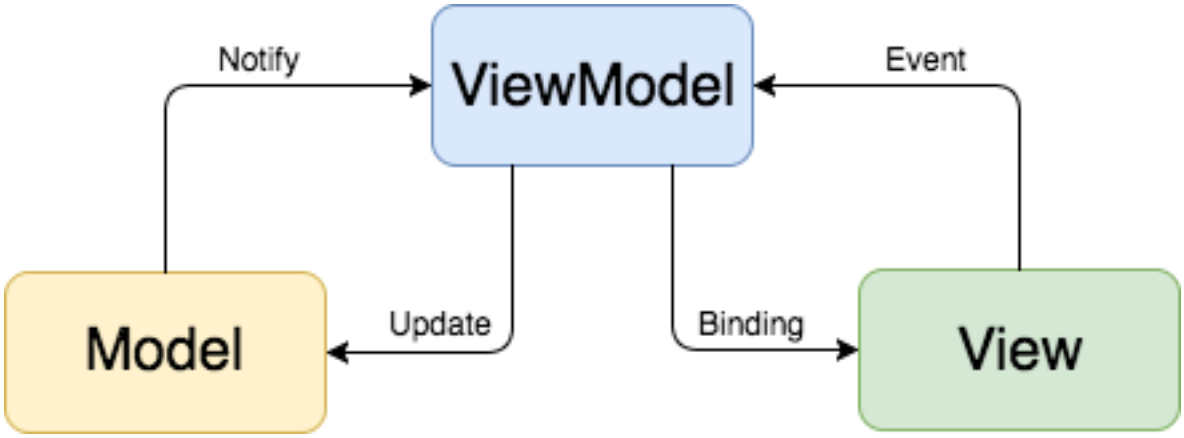

Рисунок 6 — *Схема архітектури MVVM*

VIPER

VIPER є одним з найбільш структурованих та модульних паттернів, що використовуються для розробки мобільних додатків, особливо у великих проектах [22]. Він забезпечує чітке розділення відповідальностей у додатку,

розподіляючи їх між п'ятьма основними компонентами. Схему архітектури VIPER можна побачити на рисунку 7.

- View (Представлення): Відповідає за відображення даних користувачеві та отримання вводу від користувача. View не містить бізнес-логіки і взаємодіє з презентером для отримання даних та обробки дій користувача.
- Interactor (Інтерактор): Містить бізнес-логіку додатку. Інтерактор відповідає за обробку даних та передає результати презентеру.
- Presenter (Презентер): Здійснює взаємодію між View та Interactor. він також обробляє дії користувача, отримані від View, та передає їх в Interactor.
- Entity (Сутність): Містить модель даних, вони використовуються Інтерактором для управління даними.
- Router (Маршрутизатор): Відповідає за навігацію між екранами додатку.

Переваги VIPER:

1. VIPER забезпечує чітке розділення обов'язків між компонентами.
2. Кожен компонент VIPER є незалежним модулем, що полегшує їх тестування, повторне використання та заміну.
3. Завдяки модульній структурі, легко додавати новий функціонал.
4. Поліпшена підтримуваність та масштабованість.
5. Висока гнучкість у налаштуванні навігації.

Недоліки VIPER:

1. Велика кількість коду через використання як мінімум п’яти елементів для кожного модуля.
2. Для невеликих проектів VIPER може бути надто складним та громіздким, що ускладнює його реалізацію та підтримку.
3. Освоєння цього паттерну потребує великої кількості часу.
4. Збільшені затрати часу на початковому етапі проекту.
5. Потенційна складність у розумінні різних модулів, розроблених різними розробниками, що відповідають за різні модулі моделі Viper.

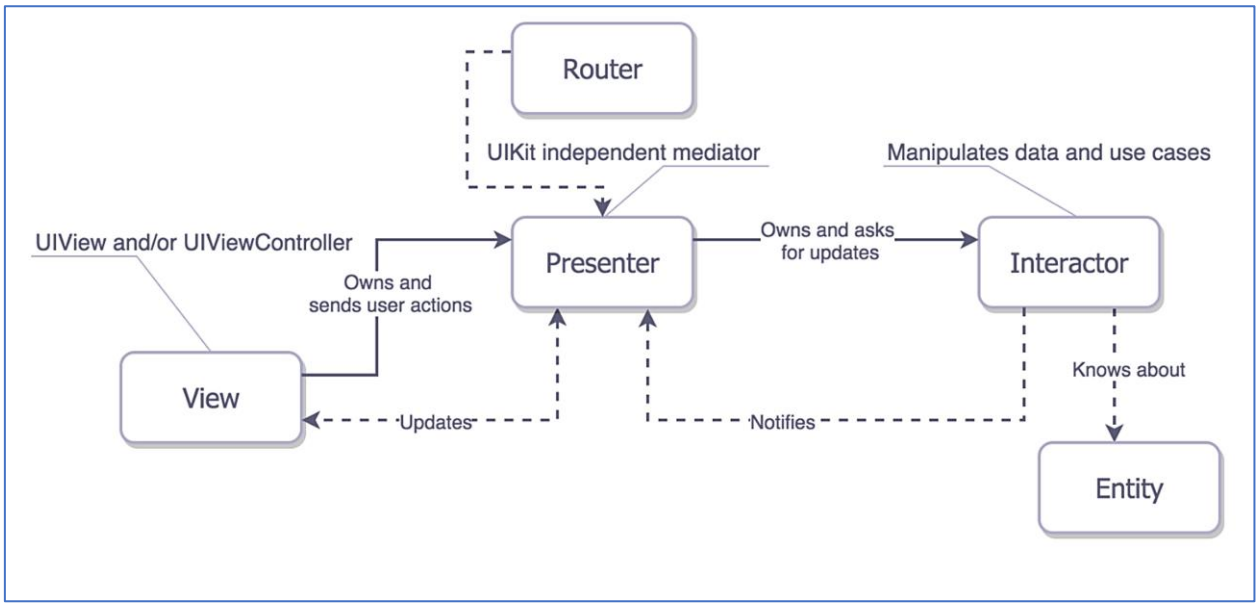

Рисунок *7 — Схема архітектури VIPER*

Архітектурні паттерни є основою для створення структурованих та підтримуваних додатків. Кожен паттерн має свої переваги та недоліки, які впливають на вибір найкращого для розробки власного застосунку. Були розглянуті чотири основні архітектурні паттерни: MVC, MVP, MVVM та VIPER. Зважаючи на переваги та недоліки різних паттернів, MVVM є найбільш збалансованим і ефективним підходом для розробки сучасних додатків. Він забезпечує оптимальне поєднання чіткого розділення обов'язків, зручного зв'язування даних та легкості тестування, що дозволяє створювати якісні та надійні додатки. MVVM є ідеальним вибором для проектів, що вимагають високої продуктивності, зручності підтримки та можливості масштабування.

2.4 Огляд систем керування залежностями

Інструменти управління залежностями — це програмні засоби, які допомагають розробникам додавати, оновлювати, видаляти та керувати сторонніми бібліотеками та фреймворками у своїх проектах. Вони автоматизують процес інтеграції зовнішніх залежностей, спрощують їх

використання та забезпечують сумісність версій, що значно підвищує ефективність розробки.

У розробці iOS додатків найпопулярнішими інструментами для управління залежностями є CocoaPods, Swift Package Manager (SPM) та Carthage. Кожен з цих інструментів має свої унікальні можливості та підходи до інтеграції залежностей у проекти.

Порівняння характеристик цих систем керування залежностями можна побачити в таблиці 3.

CocoaPods

CocoaPods один з найпоширеніших інструментів для управління залежностями в проектах iOS [23]. Він автоматизує процес завантаження, встановлення та оновлення сторонніх бібліотек, інтегруючи їх у проект через спеціальний файл з налаштуваннями Podfile.

Основні можливості:

- Централізоване управління залежностями через файл налаштувань Podfile.
- Великий репозиторій бібліотек.
- Автоматична інтеграція залежностей у проект XCode.

Swift Package Manager

Swift Package Manager (SPM) є офіційним інструментом управління пакетами від Apple, інтегрований у Swift та XCode [24]. SPM дозволяє легко додавати, оновлювати та видаляти залежності безпосередньо з інтерфейсу XCode IDE.

Основні можливості:

- Інтеграція з XCode IDE.
- Простий та зрозумілий формат файлу налаштувань Package.swift.

Carthage

Carthage простий та гнучкий інструмент управління залежностями, який надає максимальний контроль над інтеграцію бібліотек [25].

Основні можливості:

- Мінімальне втручання у проект.
- Підтримка динамічних фреймворків.
- Простий формат файлу налаштувань Cartfile для визначення залежностей.

Таблиця *3 — Порівняння характеристик CocoaPods, Swift Package Manager (SPM) та Carthage*

Критерій	**CocoaPods**	**Swift Package Manager (SPM)**	**Carthage**
Розробник	Незалежний розробник	Apple	Незалежний розробник
Рік випуску	2011	2015	2014
Інтеграція з XCode IDE	Автоматична інтеграція у проект XCode	Повна інтеграція з XCode	Мінімальні втручання у проект, ручне управління
Автоматизація процесів	Висока	Висока	Низька
Документація та підтримка	Велика спільнота, багата документація, регулярні оновлення	Підтримується Apple, інтеграція з офіційною документацією	Велика спільнота, помірна документація

Інструменти для управління залежностями є невід'ємною частиною сучасної розробки додатків для платформи iOS. Вони значно спрощують процес інтеграції та підтримки сторонніх бібліотек, забезпечують сумісність версій та полегшують оновлення залежностей. Вибір конкретного інструменту з розглянутих(CocoaPods, SPM або Carthage) залежить від потреб проекту та особистих уподобань розробників, але кожен з них забезпечує ефективний та зручний спосіб управління залежностями в проектах iOS.

Для застосунку було обрано одразу дві системи керування залежностями, а саме CocoaPods для інтеграції сторонніх бібліотек не великих за розміром та Swift Package Manager для інтеграції Firebase SDK.

2.5 Огляд СКБД

Системи керування базами даних (СКБД) відіграють ключову роль у зберіганні, організації та управлінні даними в сучасних додатках. Вибір відповідної СКБД є важливим етапом розробки, оскільки він впливає на продуктивність, масштабованість та надійність системи. Був проведений аналіз популярних СКБД, які часто використовуються у мобільних додатках для iOS-платформи. Аналіз сучасних систем керування базами даних можно подивитись у таблиці 4.

CoreData

Core Data фреймворк від компанії Apple для управління об'єктно-графічними моделями, який забезпечує збереження та відновлення стану додатка. Core Data часто використовується в iOS-додатках для управління даними та складними об'єктами.

Переваги:

- Глибока інтеграція з iOS-платформою.
- Підтримка складних реляційних моделей та графів об'єктів.
- Вбудовані можливості для управління версіями даних та міграції.

Недоліки:

- Високий рівень складності при налаштуванні та використанні.
- Може вимагати значних ресурсів при роботі з великими обсягами даних.

Realm

Realm сучасна мобільна СКБД, розроблена спеціально для мобільних додатків. Realm забезпечує високу продуктивність, простоту використання та підтримку складних об'єктно-реляційних моделей.

Переваги:

- Висока продуктивність та низька затримка при доступі до даних.
- Простий та інтуїтивний API для роботи з базою даних.

- Підтримка багатопотоковості та автоматичне управління версіями даних.

Недоліки:

- Обмежена підтримка деяких функцій SQL.
- Відносно нова технологія, що може призвести до обмеженої підтримки в деяких випадках.

Firebase Firestore

Firebase Firestore хмарна СКБД від Google, яка забезпечує зберігання та синхронізацію даних у реальному часі. Firestore часто використовується для мобільних додатків завдяки своїй можливості синхронізації даних між клієнтами.

Переваги:

- Реальна синхронізація даних між різними клієнтами.
- Простота масштабування завдяки хмарній архітектурі.
- Підтримка складних запитів та транзакцій.

Недоліки:

- Залежність від інтернет-з'єднання для синхронізації даних.
- Можливі затримки при роботі з великими обсягами даних у реальному часі.

Таблиця *4 — Порівняння характеристик CoreData, Realm та Firebase Firestore*

Критерій	**CoreData**	**Realm**	**Firebase Firestore**
Розробник	Apple	Realm	Google
Інтеграція з iOS	Глибока та нативна	2015	2014
Підтримка реляційних моделей	Так	Так	Ні
Продуктивність	Висока	Висока	Залежить від інтернету

Продовження таблиці 4

Критерій	**CoreData**	**Realm**	**Firebase Firestore**
Складність налаштування та використання	Висока	Низька	Помірна
Міграція даних	Так	Так	Ні
Зберігання даних	Локально	Локально	Хмарно
Масштабованість	Локально обмежена	Локально обмежена	Висока завдяки хмарній архітектурі
Залежність від інтернет-з’єднання	Ні	Ні	Так
Документація та підтримка	Підтримується Apple, інтеграція з офіційною документацією	Гарна документація та достатня кількість матеріалів	Підтримується Google, має гарну документацію і підтримку

Вибір відповідної системи керування базами даних дуже важливий та складний аспект розробки сучасних мобільних додатків. Остаточний вибір повинен базуватись на специфічних потребах проекту, з урахуванням факторів розглянутих у порівняльній таблиці.

2.6 Висновок

Під час дослідження засобів програмної реалізації мобільного iOS додатку був проведений ретельний аналіз сучасних, стабільних і надійних засобів, які зможуть запровадити високу якість і швидку роботу застосунку. Основним елементом розробки стала мова Swift. Ця мова насамперед була обрана через свою сучасність, швидкодію, стабільність і безпеку, зрозумілий синтаксис, який дозволяє легко реалізувати актуальні концепції програмування і втілити будь які ідеї у мобільному додатку.

У роли інтегрованого середовища розробки був обраний XCode, він забезпечує повний набір інструментів для написання, тестування та

налагодження коду, саме ці переваги роблять його основним IDE для розробки iOS додатків.

Для створення привабливого і зручного інтерфейсу користувача був обраний фреймворк UIKit. Цей фреймворк відокремився своєю стабільністю, гнучкими підходами в розробці та можливість використовувати вже готові елементи. Переваги фреймворку UIKit роблять його чудовим вибором для розробки візуально привабливих, зручних, надійних і інтерактивних iOS додатків.

Архітектурний паттерн MVVM був обраний для розробки цього iOS застосунку, бо у порівнянні він виявився найбільш збалансованим і ефективним підходом для розробки сучасних додатків. MVVM є ідеальним вибором для проектів, що вимагають високої продуктивності, зручності підтримки та можливості масштабування.

У якості системи управління залежностями було вирішено використовувати поєднання CocoaPods та Swift Package Manager. Кожен з них значно спрощують процес інтеграції та підтримки сторонніх бібліотек.

Для розробки iOS застосунку, орієнтованого на автоаматорів був обраний стек з таких технологій: Swift, UIKit, MVVM, CocoaPods, Swift Package Manager, XCode. Обрані технології забезпечать стабільну і швидку роботу застосунку, а також дозволять легко підтримувати і масштабувати його у майбутньому.

3 ПРОЄКТУВАННЯ ТА РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ

3.1 Опис предметної області

У сучасному світі автомобілі стали невід'ємною частиною нашого життя, забезпечуючи зручний та швидкий спосіб пересування. Однак з володінням автомобілем приходить і відповідальність за його обслуговування та утримання у належному стані. Потреба в ефективному управлінні витратами на обслуговування автомобіля, відстеженні пробігу та плануванні технічного обслуговування залишається актуальною з самого початку автомобілебудівництва. Власники автомобілів шукають способи оптимізувати ці процеси, знизити витрати та забезпечити довговічність та надійність своїх транспортних засобів.

Розробка мобільного додатку для iOS, орієнтованого на автоаматорів, допоможе вирішити ці завдання. Додаток дозволить користувачам зручно вести облік витрат на обслуговування автомобіля, відстежувати пробіг, отримувати своєчасні нагадування про необхідність проведення технічного обслуговування та отримувати корисну інформацію про технічний стан автомобіля. Такий додаток стане незамінним помічником для всіх, хто прагне ефективно керувати своїм автомобілем і зберігати його у відмінному стані.

У розробці додатку залучені такі сучасні технології як: Swift, UIKit, MVVM. Таке поєднання забезпечує високу швидкість, надійність та можливості масштабування проекту у майбутньому. Додаток спроектований таким чином, щоб користувачі постійно мали можливість створювати нові та переглядати вже створені записи про пробіг або витрати на авто. Використані технології забезпечують високу продуктивність та зручність використання сучасних додатків.

Основна ідея додатку полягає в наданні користувачу зручного та корисного інструменту для контролю за станом власного авто. Додаток дозволяє вводити особисті дані для заповнення профілю користувача, створення картки власного автомобіля, ввести журнал обліку робіт і витрат

проведених з авто. Інтеграція з Firebase Authentication та Firebase Firestore забезпечує безпеку зберігання особистих та чутливих даних, наприклад щодо фінансових витрат. Використання Firebase Firestore також є надійним рішенням, оскільки Firestore є NoSQL базою даних, вона не вимагає суворого визначення схеми даних. Це дає змогу легко змінювати і розширювати структуру даних у міру необхідності.

Застосунок допомагає користувачу ефективно планувати технічні роботи з транспортним засобом, контролювати витрати та підтримувати безпечний стан автомобіля. Завдяки сучасному дизайну і використанню швидких та надійних інструментів, для створення візуальної частини застосунку, додаток є легким у використанні, що лише сприяє кращому користувацькому досвіду, що неодмінно покращить досвід догляду за автомобілем.

3.2 Архітектура системи

Архітектура мобільного додатку для управління автомобільними витратами та ведення журналу технічних робіт побудована на базі архітектурного паттерну MVVM (Model-View-ViewModel) у поєднанні з паттерном Coordinator. Такий підхід забезпечує чітке розділення відповідальностей між компонентами додатку, полегшує його підтримку, масштабованість та тестування.

Coordinator

Coordinator є паттерном, що забезпечує управління навігацією між екранами додатку. Coordinator дозволяє розділити логіку навігації на окремі компоненти, що полегшує підтримку та масштабованість додатку. Основні компоненти паттерну:

- Coordinator: Основний компонент, що відповідає за управління навігацією в додатку. Coordinator координує перехід між різними екранами та забезпечує передачу даних між ними.

- Child Coordinators: Дочірні координатори, які відповідають за управління навігацією у певних розділах додатку. Використання дочірніх координаторів дозволяє розділити навігаційну логіку на окремі модулі, що полегшує їх підтримку.

Використання мови програмування Swift у поєднанні з архітектурним паттерном MVVM з додатковим використанням паттерну Coordinator є надійним фундаментом для побудови надійного, швидкого і безпечного застосунку. Використання Swift дозволяю розробляти ефективний та стійкий до помилок код, як для побудови візуальної частини, з допомогою фреймворку UIKit, так і для роботи з даними. UIKit надає потужний набір інструментів для створення інтуїтивно зрозумілого і логічного додатку. З його допомогою розробник може створювати візуально привабливі, адаптивні і гнучкі інтерфейси, які автоматично будуть підлаштовуватись під різні розміри екранів.

MVVM є одним з найпоширеніших архітектурних паттернів для розробки iOS додатків, який забезпечує чітке розділення обов'язків між трьома основними компонентами:

- Model: Відповідає за управління даними додатку. Модель містить бізнес-логіку та забезпечує збереження, отримання та маніпулювання даними.
- View: Відповідає за відображення даних користувачеві. View отримує дані від ViewModel та відображає їх у зручній для користувача формі. View також відповідає за взаємодію з користувачем (наприклад, натискання кнопок), але не містить бізнес-логіки.
- ViewModel: Здійснює взаємодію між Model та View. ViewModel отримує дані від Model, обробляє їх та передає View у зручному для відображення форматі. ViewModel також обробляє дії користувача, передаючи їх у Model та оновлюючи View відповідно до змін у Model.

MVVM забезпечує двостороннє зв'язування даних між View та ViewModel, що значно спрощує синхронізацію стану інтерфейсу користувача з даними. Це означає, що будь-які зміни в даних автоматично відображаються

в інтерфейсі, а зміни в інтерфейсі безпосередньо впливають на дані в ViewModel. Такий підхід покращує користувацький досвід та зменшує кількість коду, необхідного для оновлення інтерфейсу.

Розділення логіки додатку між Model, View та ViewModel значно спрощує тестування. ViewModel не містить залежностей від інтерфейсу користувача, що дозволяє легко писати юніт-тести для перевірки бізнес-логіки та взаємодії з даними. Це сприяє підвищенню якості коду та зменшенню кількості помилок у додатку.

Використання паттерну MVVM забезпечує високу модульність коду, що дозволяє легко розширювати та масштабувати додаток. Кожен компонент можна розробляти, тестувати та підтримувати незалежно від інших, що спрощує впровадження нових функцій та покращень. Це особливо важливо для великих проектів, де зростання функціональності є критично важливим.

Паттерн Coordinator додає додатковий рівень контролю над навігацією між екранами додатку. Використання Coordinator дозволяє розділити логіку навігації на окремі компоненти, що забезпечує гнучкість та зручність у підтримці. Кожен координатор відповідає за певний набір екранів та управляє переходами між ними, що спрощує управління складною навігацією та підвищує модульність коду. Поєднання паттернів MVVM та Coordinator забезпечує високу підтримуваність коду завдяки чіткому розділенню обов'язків, модульності та легкості тестування. Це дозволяє швидко адаптувати код до змін у вимогах до додатку, впроваджувати нові функції та виправляти помилки без значних витрат часу та зусиль. Схему MVVM + Coordinator архітектури можна переглянути на рисунку 8.

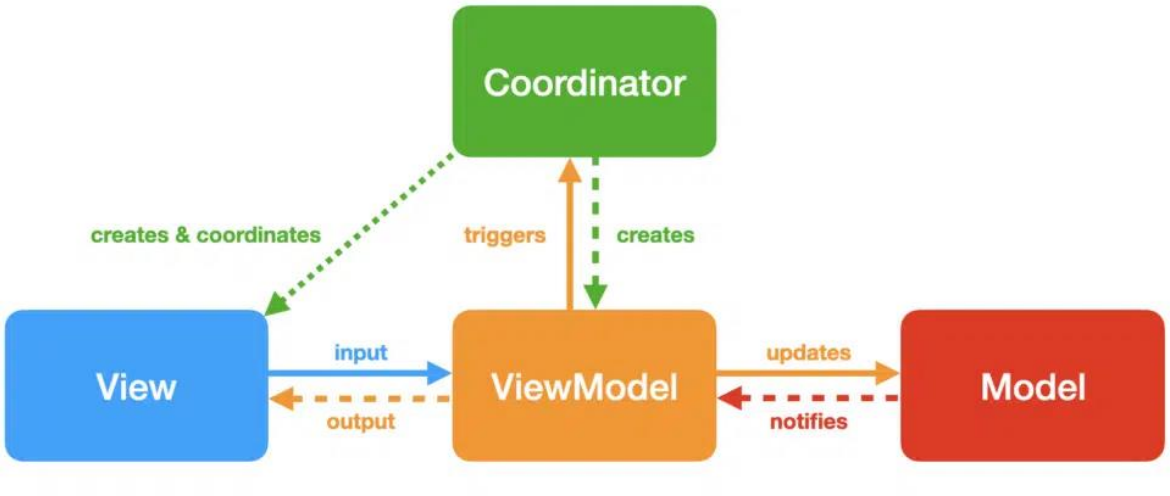

Рисунок *8 — Схема архітектури MVVM + Coordinator*

BaseCoordinator

У Лістингу 1 наведено приклад базового класу координатора BaseCoordinator, який містить основні методи та механізми для реалізації концепції "батько -> дитина" та реалізує протокол Coordinator з переліком обов’язкових властивостей і методів.

Лістинг *1 Код класу BaseCoordinator*

```
public typealias Block<T> = (T) -> Void

protocol Coordinator: AnyObject {
    associatedtype ResultType

    var onComplete: Block<ResultType>? { get set }
    var identifier: UUID { get }
    var childCoordinators: [UUID: any Coordinator] { get }

    func coordinate(to coordinator: any Coordinator)
    func start()
    func freeAllChildCoordinators()
```

```
    func dismiss()
}

extension Coordinator {
    func findCoordinator<T>(ofType type: T.Type) -> T? {
            for child in childCoordinators.values {
                if let coordinator = child as? T {
                    return coordinator
                }      else      if      let      found      =
child.findCoordinator(ofType: type) {
                    return found
                }
            }
            return nil
        }
}

open class BaseCoordinator<ResultType>: Coordinator {

    // MARK: - Public properties

    var onComplete: Block<ResultType>?

    // MARK: - Private properties

    let identifier = UUID()

    var childCoordinators: [UUID: any Coordinator] = [:]

    // MARK: - Init
    public init() {}

    // MARK: - Public methods
```

```
    func coordinate(to coordinator: any Coordinator) {
        store(coordinator: coordinator)
        handleCoordinatorCompletion(coordinator)
        DispatchQueue.main.async {
            coordinator.start()
        }
    }

    func start() {
        fatalError("❌ Method should be overridden")
    }

    func dismiss() { }

    func freeAllChildCoordinators() {
        childCoordinators.values.forEach {
            $0.dismiss()
            self.free(coordinator: $0)
        }
    }

    // MARK: - Private methods

    private func handleCoordinatorCompletion(_ coordinator:
some Coordinator) {
        let completion = coordinator.onComplete
        coordinator.onComplete = { [weak self, weak
coordinator] value in
            completion?(value)
            if let coordinator = coordinator {
                self?.free(coordinator: coordinator)
            }
        }
```

```
    }

    private func store(coordinator: some Coordinator) {
        childCoordinators[coordinator.identifier]          =
coordinator
    }

    private func free(coordinator: some Coordinator) {
        childCoordinators.removeValue(forKey:
coordinator.identifier)
    }
}
```

Опис BaseCoordinator

- BaseCoordinator — це базовий клас координатора, який забезпечує основні механізми для реалізації концепції "батько -> дитина". Він містить методи для управління дочірніми координаторами, запуску та завершення їхніх потоків.
- Coordinator Protocol: протокол, який визначає базові властивості та методи координатора, включаючи start, coordinate(to:), freeAllChildCoordinators та dismiss.

Публічні властивості:

- onComplete: блок, який викликається координатором в кінці свого потоку з заданим зворотнім викликом.
- identifier: унікальний ідентифікатор координатора.
- childCoordinators: словник дочірніх координаторів, необхідний для підтримки життєвого циклу дочірнього координатора.

Публічні методи:

- coordinate(to:): запускає новий координатор, зберігаючи посилання на нього з моменту його запуску і звільняючи його в момент завершення.

- start(): абстрактний метод, який має бути перевизначений для запуску потоку координатора.
- freeAllChildCoordinators(): звільняє всі дочірні координатори.

Приватні методи:

- handleCoordinatorCompletion(_:): обробляє завершення координатора, викликаючи зворотний виклик onComplete.
- store(coordinator:): зберігає дочірній координатор у словнику.
- free(coordinator:): звільняє дочірній координатор зі словника.

AnyViewModel

У Лістингу 2 представлено приклад протоколу AnyViewModel, що забезпечує чітку структуру для створення ViewModel, базовий протокол для ViewModel, який буде використовуватися для взаємодії між Model та View.

Лістинг *2 Код протоколу AnyViewModel*

```
protocol AnyViewModel {
    associatedtype Input
    associatedtype Output

    func transform(_ input: Input) -> Output
}
```

Опис протоколу AnyViewModel

- protocol AnyViewModel: це протокол, який визначає базову структуру для всіх ViewModel у додатку. Він використовує generics для визначення типів вхідних (Input) та вихідних (Output) даних, що дозволяє створювати гнучкі та повторно використовувані ViewModel.
- Input: асоційований тип, що представляє вхідні дані для ViewModel. Це можуть бути будь-які дані, які надходять з View, такі як події або дії користувача.

- Output: асоційований тип, що представляє вихідні дані ViewModel. Це можуть бути оброблені дані, які необхідно відобразити у View, такі як відформатовані дані або результати обробки.
- transform(_:): метод, який трансформує вхідні дані в вихідні. Цей метод приймає вхідні дані типу Input та повертає вихідні дані типу Output. Такий підхід забезпечує чітке розділення обов'язків між View та ViewModel, де ViewModel відповідає за обробку даних, а View за їх відображення.

BaseViewController

Для реалізації базового контролера у паттерні MVVM, який може працювати з будь-яким типом ViewModel, у додатку використовується узагальнений клас BaseViewController. Такий підхід дозволяє створити гнучкий та повторно використовуваний контролер, що забезпечує чітку структуру для відображення даних та взаємодії з ViewModel.

Лістинг 3 *Код класу BaseViewController*

```
class BaseViewController<ViewModel>: UIViewController {

    var viewModel: ViewModel!

    var disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        navigationController?.navigationBar.isTranslucent  =
true
        setupView()
        setupRx()
    }

    func setupView() { }
```

```
    func setupRx() { }
}
```

Опис класу BaseViewController

- BaseViewController: це узагальнений базовий клас для всіх ViewController, що використовують паттерн MVVM. Він містить властивості та методи, які є спільними для всіх контролерів, що працюють з ViewModel.
- <ViewModel>: узагальнений тип, що представляє ViewModel для даного ViewController. Це дозволяє використовувати цей базовий клас з будь-яким типом ViewModel, забезпечуючи гнучкість та повторне використання коду.
- viewModel: властивість, що зберігає екземпляр ViewModel, з яким взаємодіє ViewController.
- disposeBag: властивість, що зберігає підписки RxSwift. Вона використовується для автоматичного звільнення ресурсів при деінтеграції ViewController.
- viewDidLoad(): метод життєвого циклу ViewController, який викликається при завантаженні контролера. Тут відбувається налаштування вигляду інтерфейсу та зв'язків Rx.
- setupView(): метод, який використовується для налаштування вигляду інтерфейсу. Він реалізується у підкласах для налаштування конкретного інтерфейсу ViewController.
- setupRx(): метод, який використовується для налаштування зв'язків Rx. Він реалізується у підкласах для налаштування конкретних зв'язків Rx між View та ViewModel.

3.3 Функціональні вимоги системи

Функціональні вимоги визначають основні можливості та функції, які повинен забезпечувати мобільний додаток. Ці вимоги розроблені на основі аналізу предметної області, потреб користувачів та сучасних підходів до

розробки мобільних додатків. У цьому розділі представлені основні функціональні вимоги, які розроблені для успішної розробки iOS застосунку, орієнтованого на автоаматорів:

1. Реєстрація та аутентифікація користувачів.

- Реєстрація нового користувача: Додаток повинен надавати можливість новим користувачам реєструватися, вводячи особисті дані, такі як ім'я, електронна пошта та пароль.
- Аутентифікація користувача: Додаток повинен забезпечувати безпечну аутентифікацію користувачів за допомогою електронної пошти та пароля.
- Відновлення пароля: Додаток повинен надавати можливість користувачам відновити забутий пароль через електронну пошту.

2. Управління профілем

- Редагування профілю: Користувач повинен мати можливість редагувати свої особисті дані, включаючи ім'я, електронну пошту та пароль.

3. Управління карткою автомобіля

- Додавання нового автомобіля: Користувач повинен мати можливість додавати новий автомобіль, вводячи дані, такі як марка, модель, рік випуску.
- Редагування даних автомобіля: Користувач повинен мати можливість редагувати дані про автомобіль, включаючи марку, модель, рік випуску.
- Видалення автомобіля: Користувач повинен мати можливість видалити автомобіль зі свого профілю.

4. Облік проведених технічних робіт/записів

- Додавання запису: Користувач повинен мати можливість додавати нові записи про обслуговування автомобіля, вводячи дані, такі як дата, сума, тип витрати та коментар.
- Редагування запису: Користувач повинен мати можливість редагувати дані про запис, включаючи дату, суму, тип витрати та коментар.
- Видалення запису: Користувач повинен мати можливість видалити записи зі свого профілю.

5. Використання розділу Медіа

- Користувач повинен мати можливість переглядати актуальні веб-ресурси з цікавою текстовою інформацією про авто світ.
- Користувач повинен мати можливість переглядати актуальні відео, у вбудованому плеєрі, з цікавою інформацією про авто світ.

6. Використання розділу Інструменти
   - Користувач має мати можливість використати інструмент для розрахунку розходу пального.
   - Користувач має мати можливість використати інструмент для перегляду найближчих авто заправних станцій.
   - Користувач має мати можливість використати інструмент для перегляду найближчих станцій технічного обслуговування.

Загальну UseCase діаграму можливостей користувача можна переглянути на рисунку 9.

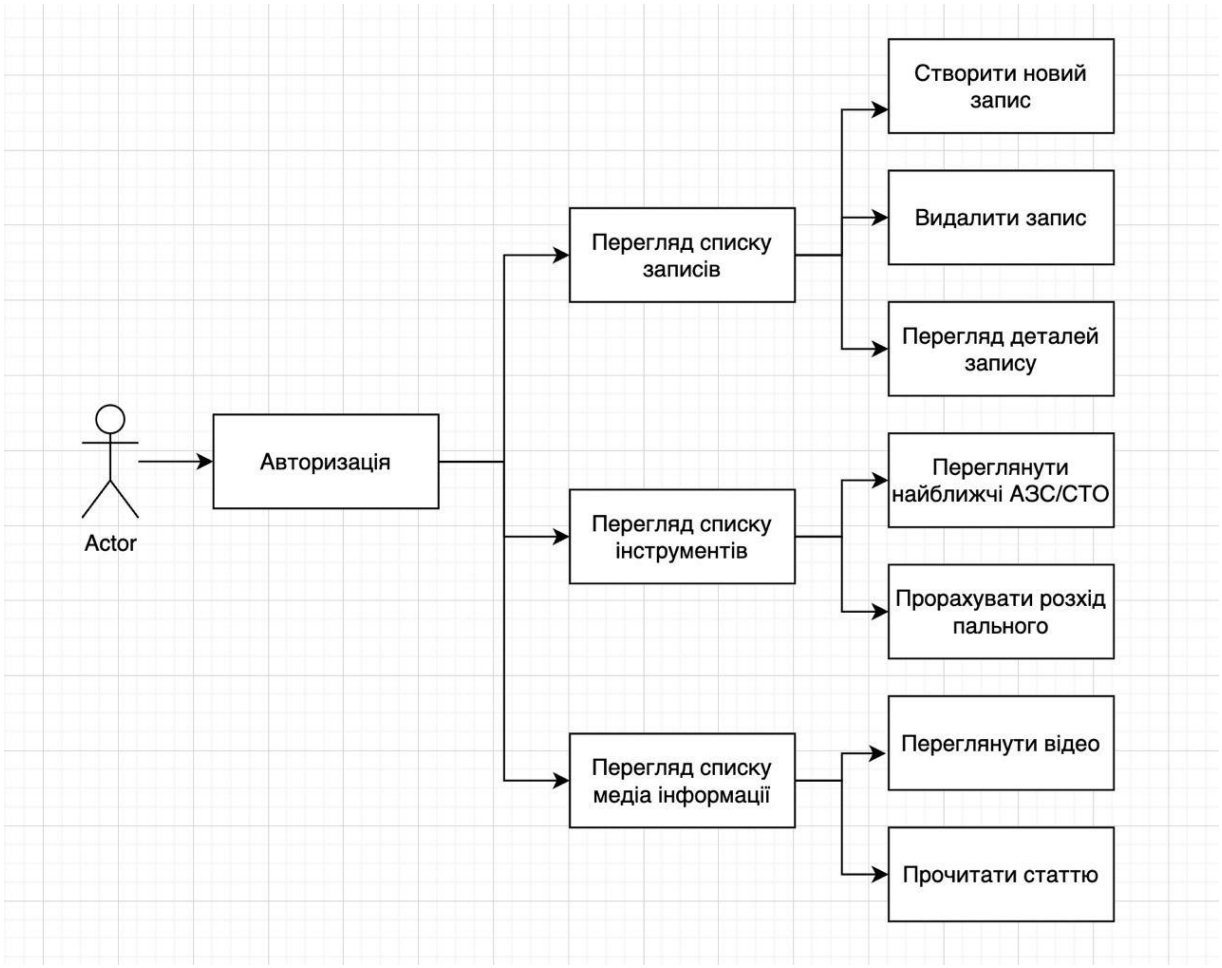

Рисунок 9 — *Загальна UseCase діаграма можливостей користувача*

На діаграмі зображеній на рисунку 10, можна побачити прецедент відносно пунктів 1 та 2. Користувач повинен мати можливість зареєструватись і після реєстрації заповнити свій профіль, маючи можливість змінити свої дані, або створити картку свого персонального авто.

Рисунок *10 — UseCase діаграма до 1 та 2 пункту функціональних вимог.*

Доступ до корисних та актуальних медіа даних є важливим інструментом у застосунку. Користувач повинен мати доступ до цього розділу з можливість більш детально ознайомитись як з веб-статтею, так і з відео матеріалом. Діаграма до пункту 5 зображена на рисунку 11.

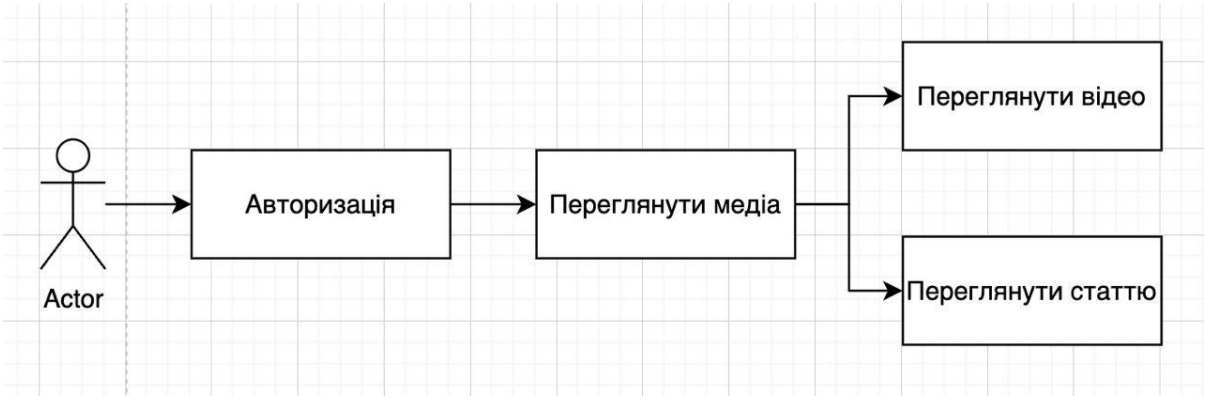

Рисунок *11 — UseCase діаграма до 5 пункту функціональних вимог.*

Можливість використовувати функціональні інструменти для покращення досвіду володіння автомобілем також є важливою функцією

додатку орієнтованого на автоаматорів. Діаграму до 6 пункту фукнціональних вимог можна побачити на рисунку 12.

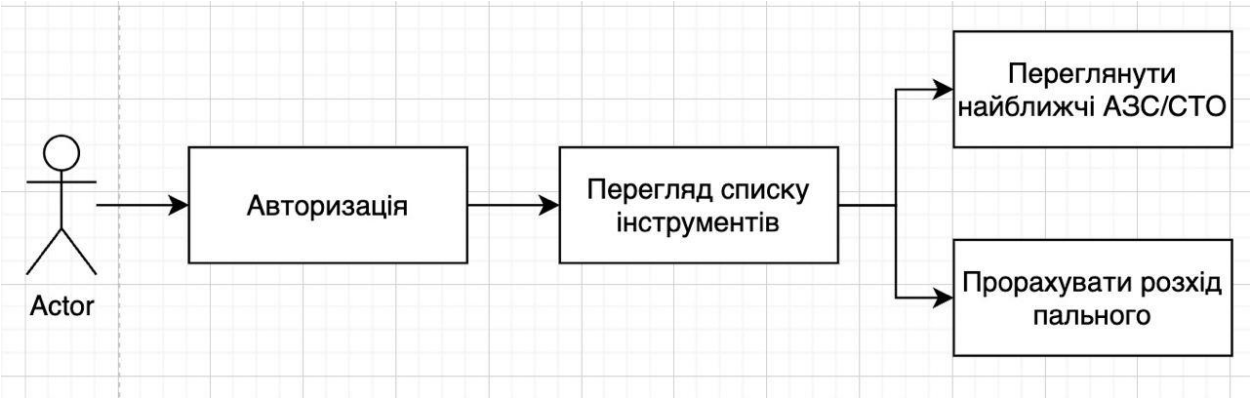

Рисунок *12 — UseCase діаграма до 6 пункту функціональних вимог.*

Створення нових записів про технічне обслуговування, можливість їх перегляду раніше створених записів та видалення цих записів є важливою функцією додатку. Діаграму до пункту 4 можна переглянути на рисунку 13.

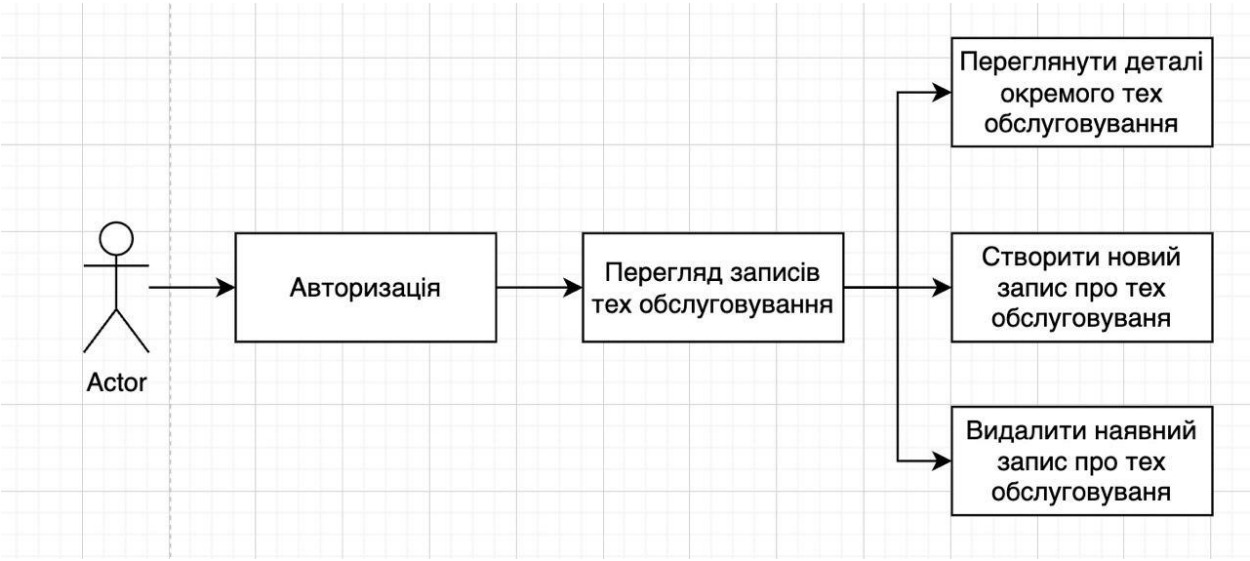

Рисунок *13 — UseCase діаграма до 4 пункту функціональних вимог.*

Функціональні вимоги системи визначають основні можливості, які повинен забезпечувати iOS застосунок призначений для автоаматорів. Виконання цих вимог дозволить створити зручний та функціональний додаток, який задовольняє потреби користувачів, забезпечує зручність

використання та ефективне управління автомобільними витратами та технічним обслуговуванням.

3.4 Вимоги до апаратного та програмного забезпечення

Розробка мобільного додатку для управління автомобільними витратами вимагає відповідності певним апаратним та програмним вимогам. Визначення цих вимог є важливим етапом у процесі розробки, оскільки вони забезпечують сумісність додатку з цільовими пристроями та операційною системою, а також гарантують його ефективність та стабільність роботи.

Мінімальні вимоги:

- Операційна система: Додаток розробляється для платформи iOS та підтримує версію iOS 12.0 і вище. Це забезпечує сумісність додатку з широким спектром пристроїв Apple, включаючи старі моделі, що все ще активно використовуються користувачами.
- Пристрії: Додаток підтримує iPhone 5s і всі новіші за нього моделі.
- Процесор: Для забезпечення продуктивної та стабільної роботи додаток потребує від пристроїв процесор с 64-бітовою системою, такі як Apple A7 або новіші за нього.
- Оперативна пам'ять: Для коректної роботи додатку рекомендується мати не менше як 1 ГБ оперативної пам’яті.
- Місце на диску: Додаток вимагає наявності не менш як 100мб вільного місця.
- Інтернет з’єднання: Для роботи окремих функцій необхідне стабільне інтернет підключення.

Оптимальні вимоги:

- Операційна система: iOS 14 та вище.
- Пристрій: iPhone 11
- Процесор: Apple A13

- Оперативна пам’ять: 2 ГБ
- Місце на диску: Додаток вимагає наявності не менш як 100мб вільного місця.
- Інтернет з’єднання: Для роботи окремих функцій необхідне стабільне інтернет підключення.

Визначення апаратних та програмних вимог є ключовим етапом у процесі розробки мобільного додатку для управління автомобільними витратами. Відповідність цим вимогам забезпечує сумісність додатку з цільовими пристроями та операційними системами, гарантує його ефективність, стабільність та безпеку роботи. Дотримання цих вимог дозволяє створити високоякісний продукт, який задовольнить потреби користувачів та забезпечить їм зручний та надійний інструмент для керування станом свого транспортного засобу.

3.5 Модулі та алгоритми

Важливим аспектом при розробці мобільного застосунку є алгоритми та модулі, які забезпечують швидку та зручну роботу. Кожний модуль додатку, призначений для відображення екранів з користувацьким інтерфейсом, використовує архітектурний паттерн MVVM+R, описаний раніше у розділі 3.2.

Були розроблені такі модулі:

- SplashScreen – модуль екрану завантаження застосунку.
- AuthScreen – вітальний модуль у якому користувач може обрати увійти або створити новий обліковий запис, також є можливість переглянути екран з гідом застосунку.
- Tour – модуль екрану гіда по застосунку.
- Login – модуль де користувач може увійти в додаток, авторизація виконується за допомогою Firebase Auth.
- Register – модуль для створення нового облікового запису.

- UpdatePassword – модуль який надає користувачу можливість відновити пароль до свого облікового запису.
- Home – основний модуль, надає можливість перейти до розділу записів, інструментів, медіа або налаштувань профілю.
- Profile – модуль з можливостями вийти з облікового або видалити обліковий запис, перейти до налаштувань облікового запису та змінити фотографію профілю.
- Records – модуль де користувач може створити, видалити, редагувати та переглядати раніше створені записи.
- Media – модуль з корисною медіа інформацією, де користувач може прочитати веб-статтю, або переглянути відео.
- Tools – модуль з корисними інструментами, які направлені на покращення досвіду володіння та піклування за автомобілем.

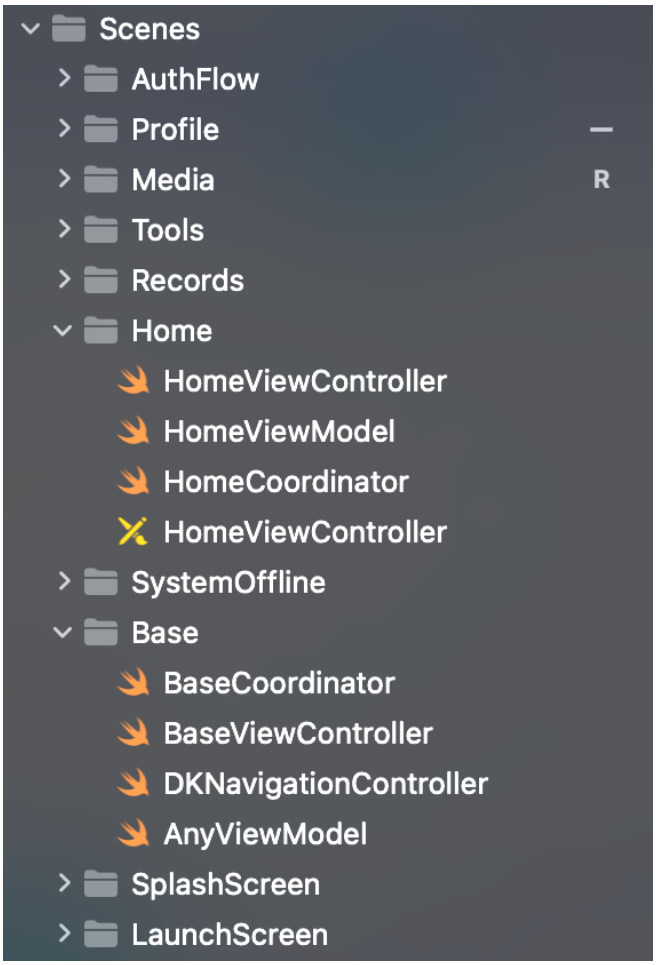

Рисунок *14 — Структура модулів додатку*

AppNetwork

AppNetwork – клас призначений для керування мережевими запитами додатку. Цей клас відповідає за створення, відправлення та обробку запитів, а також за обробку можливих помилок. Клас використовує RxSwift для роботи з асинхронними операціями.

Перелік методів класу AppNetwork можна побачити на лістингу 4.

Лістинг 4 *Методи класу AppNetwork*

```
final class AppNetwork {
    func request<T>(endpoint: T) -> Single<T.Response> where
T: Requestable
    func request<T>(
            endpoint: T,
            additionalHeaders: [String: String?] = [:]
        ) -> Single<T.Response> where T: Requestable
    func makeRequest<T: Decodable>(request: URLRequest,
responseType: T.Type) -> Single<T>
    func prettyPrintedResponse(_ data: Data) -> String
    func getAuthToken() -> String?
    func getAuthToken() -> Single<String?>
}
```

Основні методи:

- request(endpoint:) — основний метод для відправки мережевих запитів. Він отримує endpoint та повертає асинхронний результат запиту у вигляді об'єкта Single<T.Response>, де T — тип відповіді.
- request(endpoint:additionalHeaders:) — метод для відправки запитів з додатковими заголовками. Працює аналогічно до попереднього методу, але дозволяє додавати заголовки специфічні для запиту.

Приватні методи:

- makeRequest(request:responseType:) — виконує фактичний запит до сервера, обробляє відповідь та повертає результат або помилку.
- prettyPrintedResponse(data:) — форматує відповідь сервера у читабельний JSON формат для зручності налагодження.
- getAuthToken() — отримує токен авторизації зі сховища.
- getAuthToken() -> Single<String?> — отримує або оновлює токен авторизації та повертає його асинхронно.

AppUserService

AppUserService — клас, призначений для керування даними користувачів у додатку. Він забезпечує збереження, оновлення та отримання цих даних з захищеного сховища. Клас також відслідковує зміни в особистих даних користувача і повідомляє про ці зміни відповідні класи-спостерігачі за допомогою бібліотеки RxSwift.

Лістинг 5 *Методи класу AppUserService*

```
final class AppUserService {
    var userObservable: Observable<AppUser?>
    var isAuthorized: Bool
    func initiateCurrentUser()
    func setCurrentUser(_ user: AppUser?)
    func setUserId(_ id: Int)
    func updateUserPhoto(_ encodedPhoto: String?)
    func storeToken(_ token: String, for key:
Keys.SecureKey)
    func getToken() -> JWTToken?
    func logout()
    func deleteTokens()
}
```

Основні методи:

- userObservable — повертає об'єкт типу Observable<AppUser?>, що дозволяє спостерігати за змінами в даних користувача.
- isAuthorized — перевіряє наявність та дійсність токена авторизації. Повертає true, якщо користувач авторизований.
- initiateCurrentUser() — ініціалізує поточного користувача, використовуючи дані з Firebase Authentication.
- setCurrentUser(_ user: AppUser?) — встановлює поточного користувача.
- setUserId(_ id: Int) — встановлює ID користувача та повідомляє про зміну.
- updateUserPhoto(_ encodedPhoto: String?) — оновлює фотографію користувача та повідомляє про зміну.
- storeToken(_ token: String, for key: Keys.SecureKey) — зберігає токен авторизації в захищеному сховищі.
- getToken() -> JWTToken? — отримує токен авторизації з захищеного сховища.
- logout() — виконує вихід користувача, видаляючи дані користувача та токени авторизації.

LocalStorage

LocalStorage — це клас, що забезпечує єдиний інтерфейс для взаємодії з локальним сховищем даних. Він включає методи для збереження, оновлення, видалення та отримання даних різних типів.

Лістинг 6 *Методи класу LocalStorage*

```
final class LocalStorage {
      func storeEntity<T: LocalStorageTypeDescription>(_
entity: T)
    func                                    updateEntity<T:
LocalStorageTypeDescription>(entity: T)
    func deleteEntity<T: LocalStorageTypeDescription>(
        _ entity: T,
```

```
        callback: @escaping (Result<Bool, Error>) -> Void
    )
    func fetchEntity<T: LocalStorageTypeDescription>(
        ofType type: T.Type,
        predicate: NSPredicate?,
        callback: @escaping (T.LocalStorageType.Domain?) ->
Void
    )
    func fetchAllEntities<T: LocalStorageTypeDescription>(
        ofType type: T.Type,
        sortDescriptor: NSSortDescriptor?,
        predicate: NSPredicate?,
        callback: @escaping ([T.LocalStorageType.Domain]) ->
Void
    )
    func deleteStorage()
    func deleteAllEntities<T: LocalStorageTypeDescription>(
        ofType type: T.Type,
        callback: @escaping (Result<Void, Error>) -> Void
    )
}
```

Основні методи:

1. storeEntity(_ entity: T) — зберігає сутність у локальному сховищі.
2. updateEntity(entity: T) — оновлює сутність у локальному сховищі.
3. deleteEntity(_ entity: T, callback: @escaping (Result<Bool, Error>) -> Void) — видаляє сутність з локального сховища
4. fetchEntity(ofType type: T.Type, predicate: NSPredicate?, callback: @escaping (T.LocalStorageType.Domain?) -> Void) — отримує одну сутність з локального сховища відповідно до заданих умов.
5. fetchAllEntities(ofType type: T.Type, sortDescriptor: NSSortDescriptor?, predicate: NSPredicate?, callback: @escaping ([T.LocalStorageType.Domain]) ->

Void) — отримує всі сутності з локального сховища відповідно до заданих умов та сортування.

6. deleteStorage() — видаляє все сховище даних.
7. deleteAllEntities(ofType type: T.Type, callback: @escaping (Result<Void, Error>) -> Void) — видаляє всі сутності заданого типу з локального сховища.

3.6 Структури даних

Під час розробки додатку були розроблені такі таблиці: “Record”, “AppUser” та “UserInfo”. Кожна таблиця має свої інформаційно ємкі поля, які забезпечують надійне зберігання та швидкий доступ до даних.

Таблиця “AppUser” зберігає особисті дані про обліковий запис користувача, а “UserInfo” особисті дані користувача, які він надав у застосунку.

Поля таблиці “AppUser”:

- Поле “uuid” (String): унікальний ідентифікатор користувача.
- Поле “email” (String): электрона адреса користувача.
- Поле “isEmailVerified” (Bool): статус верифікації електронної пошти.
- Поле “userInfo” (UserInfo): Персональна інформація користувача.

Поля таблиці “UserInfo”:

- Поле “nickname” (String): псевдонім користувача.
- Поле “birthDate” (Int): дата народження користувача.
- Поле “Gender” (String): стать користувача.

Поля таблиці “Record”:

- Поле “name” (String): назва запису.
- Поле “id” (String): ідентифікатор запису.
- Поле “date” (Int): дата запису.
- Поле “description” (String): інеформаційний допис до запису.
- Поле “spendedMoney” (Int): кількість витрачених коштів.

3.7 Проект інтерфейсу

Сучасні iOS додатки вимагають від розробників все більш зручних, привабливих, інтерактивних і чітких інтерфейсів. Такі інтерфейси не тільки відповідають вимогам користувачів, а ще й впливають на ефективність та швидкодію застосунку.

Створення сучасних інтерфейсів складний та часоємкий процес, він вимагає аналізу потреб користувачів і детального планування. Використання таких сучасних технологій, як Swift та UIKit надає розробникам можливість задовольняти постійно зростаючі вимоги користувачів, а поєднання цих інструментів з архітектурним паттерном MVVM + Coordinator забезпечує простоту розширення та підтримки такого додатку.

Результатом розробки є сучасний інтерфейс, який дозволяє користувачам використовувати такі функції додатку як: створення облікового запису, редагування облікового запису, створення нових та перегляд старих записів про технічні роботи, перегляд медіа інформації та інструментів направлення на покращення досвіду піклування за власним автомобілем.

Детальніше інтерфейс iOS застосунку, орієнтованого на автоаматорів можна переглянути на рисунках 15- 26.

Рисунок *15* — *Вітальний екран застосунку*.

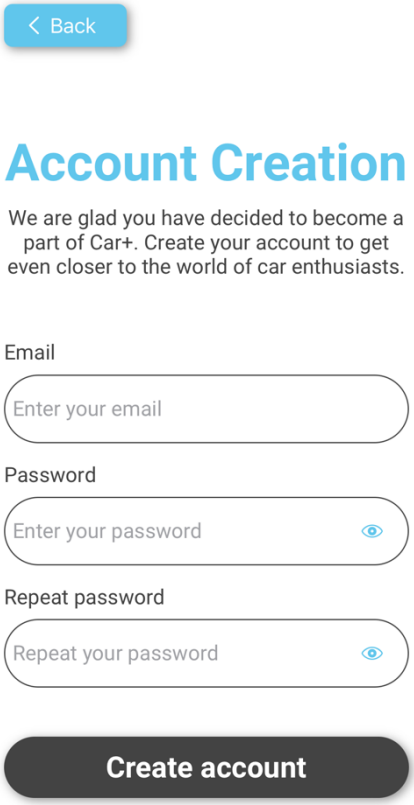

Рисунок *16* — *Екран реєстрації*.

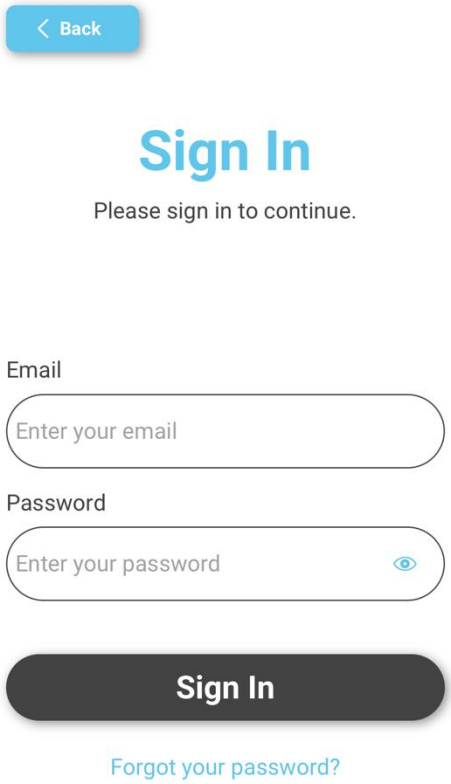

Рисунок *17* — *Екран авторизації.*

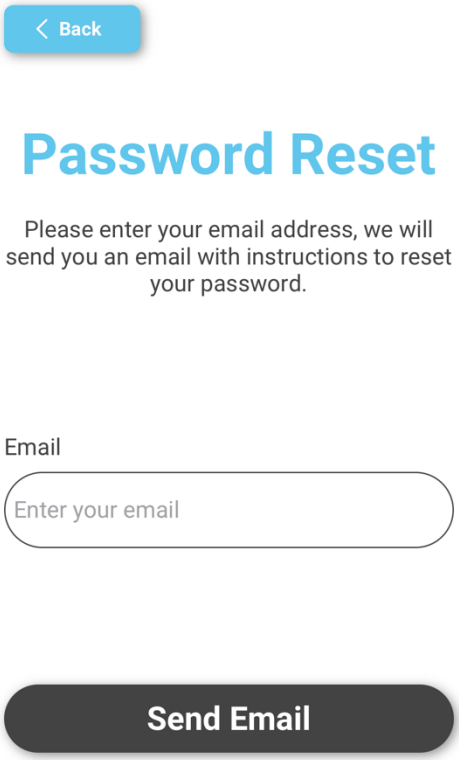

Рисунок *18* — *Екран відновлення паролю.*

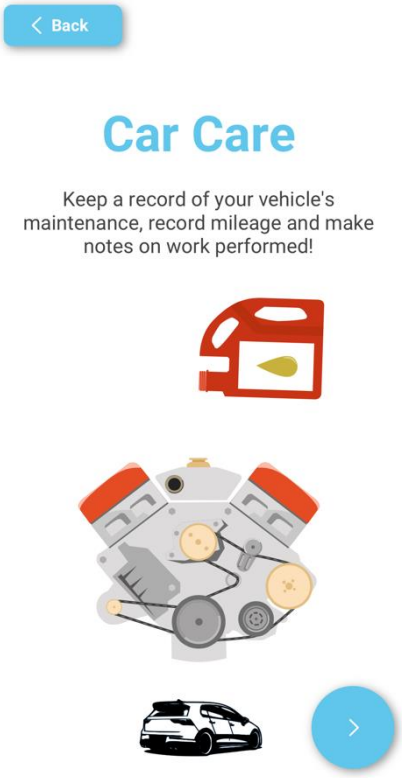

Рисунок *19* — *Екран гіду по застосунку.*

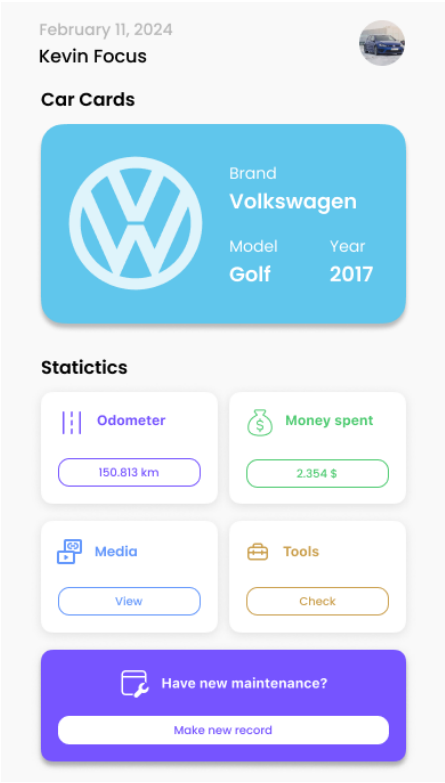

Рисунок *20* — *Головний екран застосунку.*

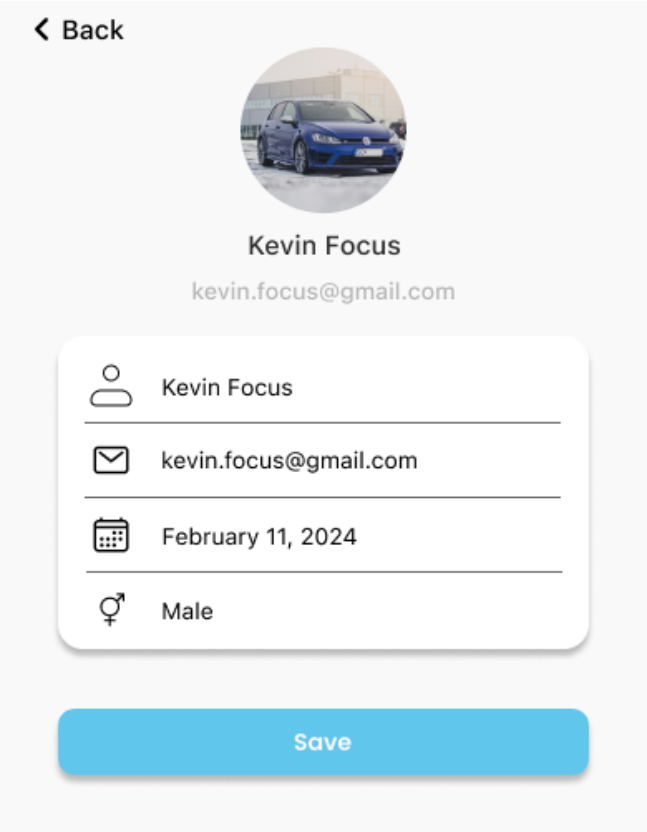

Рисунок *21 — Екран редагування персональних даних користувача.*

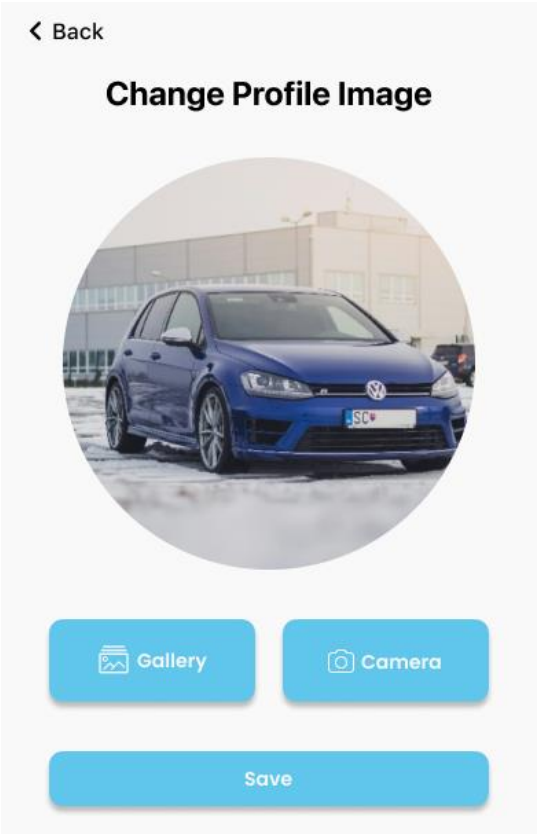

Рисунок *22 — Екран зміни зображення профілю.*

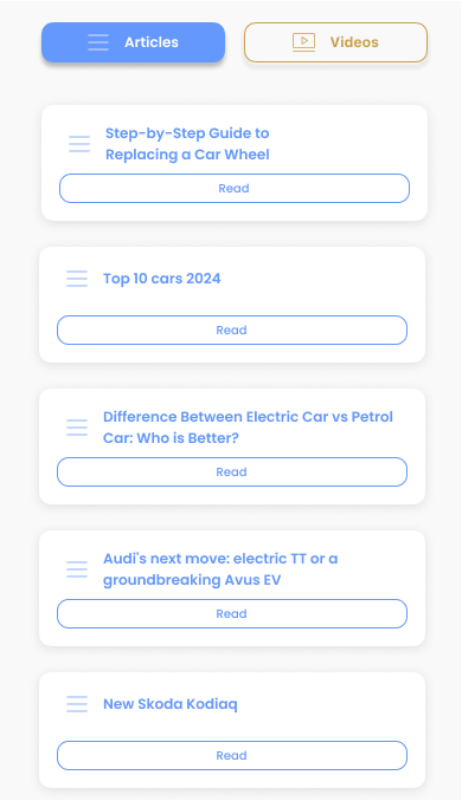

Рисунок *23* — *Екран медіа з веб-статтями.*

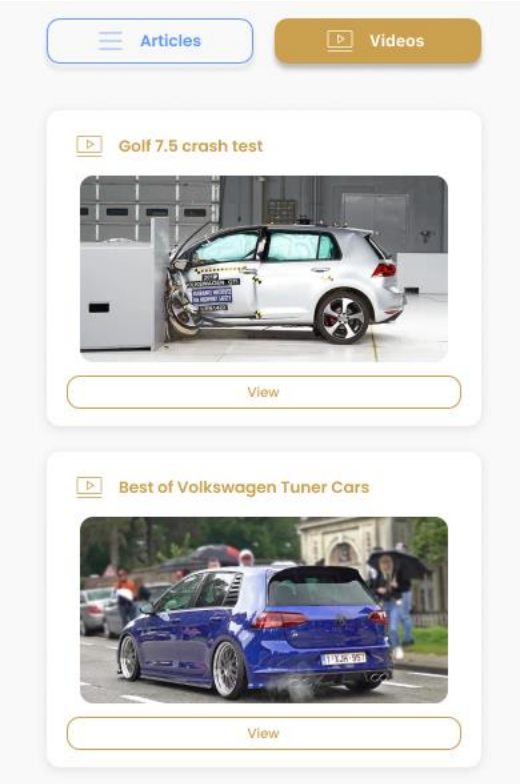

Рисунок *24* — *Екран медіа з відео.*

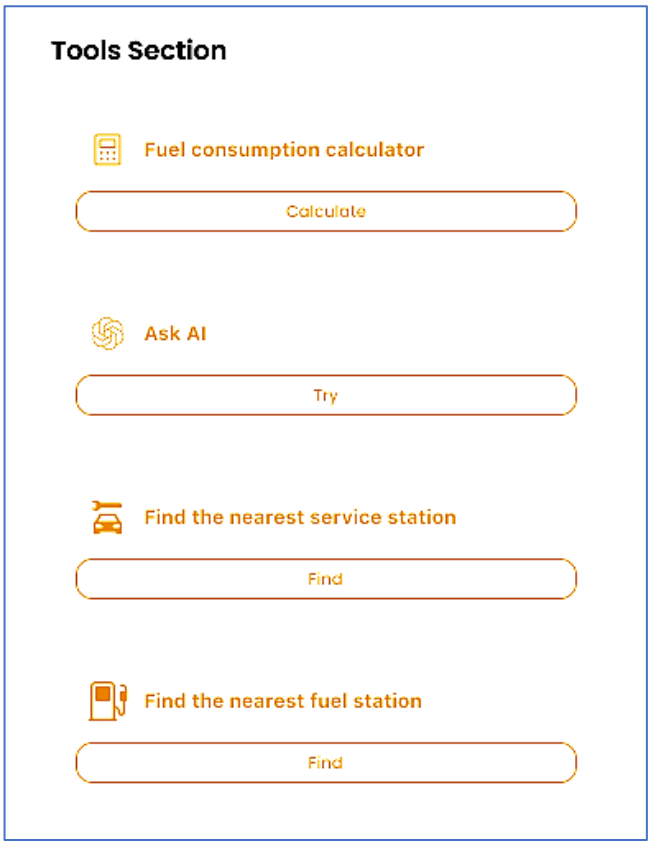

Рисунок *25* — *Екран інструментів.*

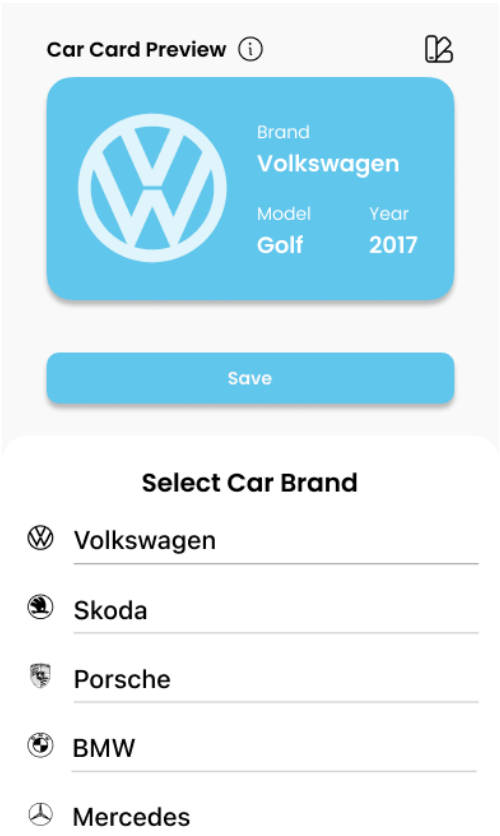

Рисунок *26* — *Створення картки авто.*

3.8 Фізичні характеристики та тестування

Фізичні характеристики

Фізичні характеристики поточної версії системи:

- Об'єм коду: приблизно 20.000 рядків
- Кількість модулів: 11
- Розмір: 104 мегабайти.
- Кількість залежностей: 12
- Платформа: iOS
- Середній час збірки: 56 секунд
- Кількість функцій: 762
- Кількість файлів: 1321

Тестування

Тестування iOS застосунку для автоаматорів охоплює різні види тестування функціональне, мануальне та тестування продуктивності. Використання автоматизованих інструментів дозволило ефективно виявляти та виправляти помилки, забезпечуючи високу якість продукту та його відповідність поставленим вимогам.

Мануальне тестування проводилось на телефоні iPhone 13 з встановленою iOS 17.4, а для тестування інших розмірів екрану та операційних систем використовувався вбудований емулятор XCode.

Для тестування продуктивності, використовувався вбудований у XCode інструмент Activity Monitor, який дозволяє відстежувати використовувані ресурси такі як: завантаженість центрального процесору, дискову активність та мережевий трафік. Результати тестування продуктивності можна переглянути на рисунку 27.

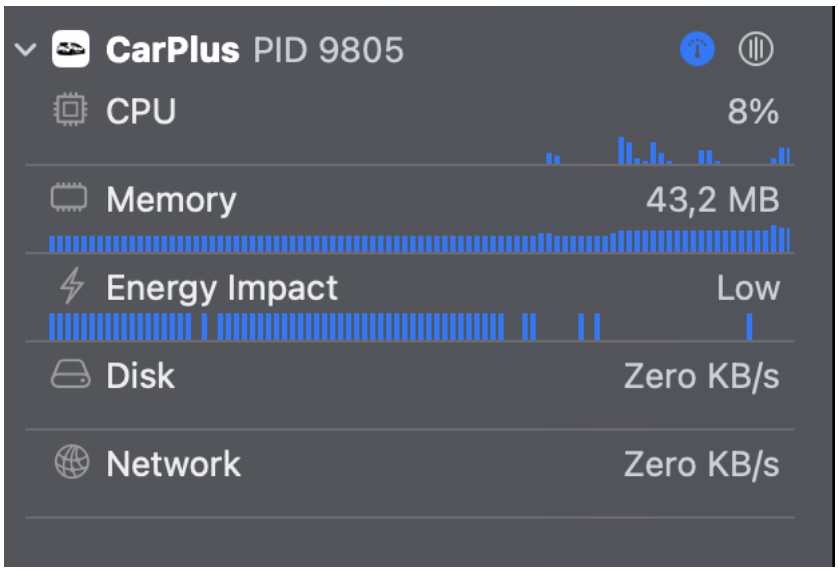

Рисунок *27 — Результати тестування ActivityMonitor*

ВИСНОВКИ

1. Під час виконання кваліфікаційної роботи був проведений аналіз літератури присвяченої дослідженням та розробкам у сфері створення сучасних мобільних застосунків для платформи iOS, також був проведений аналіз існуючих на ринку аналогів.
2. Проведений аналіз існуючих та актуальних інструментів розробки iOS ззастосунків. Були визначені їх переваги та недоліки, вивчено їх особливості.
3. В ході розробки застосунку використовувалась мова програмування Swift, швидкий та надійний інструмент для розробки сучасних застосунків на платформі iOS. Також був використаний фреймворк UIKit для створення візуально привабливого та інтерактивного інтерфейсу користувача.
4. У процесі проектування застосунку був проведений аналіз архітектурних патернів, що дозволило створити систему готову для масштабування та подальшої підтримки.
5. Результатом роботи є успішно створений iOS застосунок, орієнтований на автоаматорів. Реалізація функціональності включала можливості створення та авторизації облікового запису користувача, можливість створювати, переглядати, редагувати та видаляти записи про проведені технічні роботи, або облікові витрати на автомобіль, а також можливість використовувати корисні інструменти які допоможуть покращити досвід володіння та піклування за власним авто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. How Many People Have iPhones in 2024?: веб-ресурс. URL: https://www.bankmycell.com/blog/number-of-iphone-users (дата звернення: 04.05.2024).
2. How many cars are there in the world?: веб-ресурс. URL: https://hedgescompany.com/blog/2021/06/how-many-cars-are-there-in-the-world/ (дата звернення: 04.05.2024).
3. Климов Д.В. Розробка iOS застосунку, орієнтованого на автоаматорів. Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «Молода наука-2024» / Запорізький національний університет. – Запоріжжя : ЗНУ, 2024. Т.5. С. 180-182. Веб ресурс. URL: https://sites.znu.edu.ua/stud-sci-soc//2009/tom_5_2024.pdf (дата звернення: 14.05.2024).
4. MyCar. AppStore page: app store, платформа цифрової дистрибуції. URL: https://apps.apple.com/ua/app/my-car-vehicle-manager/id1165749302 (дата звернення: 04.05.2024).
5. CarKeep. AppStore page: app store, платформа цифрової дистрибуції. URL: https://apps.apple.com/ua/app/carkeep-vehicle-manager/id6479198249 (дата звернення: 04.05.2024).
6. Drivvo: app store, платформа цифрової дистрибуції. URL: https://apps.apple.com/ua/app/drivvo-car-management/id1206041425 (дата звернення: 04.05.2024).
7. Dart: веб-ресурс. URL: https://dart.dev/overview (дата звернення: 04.05.2024).
8. Flutter. Flutter on mobile: електронний ресурс. URL: https://flutter.dev/multi-platform/mobile (дата звернення: 04.05.2024).
9. Objective-C. The Objective-C Programming Language: електронний ресурс. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptua

l/ObjectiveC/Introduction/introObjectiveC.html (дата звернення: 05.05.2024).

10. Apple acquires Next, Jobs: веб-ресурс. URL: https://www.cnet.com/tech/tech-industry/apple-acquires-next-jobs/ (дата звернення: 05.05.2024).
11. Swift the powerful programming language: веб-ресурс. URL: https://developer.apple.com/swift/ (дата звернення: 04.05.2024).
12. Apple - WWDC 2014. Timecode 1:44:00 URL: https://www.youtube.com/watch?v=w87fOAG8fjk (дата звернення: 05.05.2024).
13. XCode overview: веб-ресурс. URL: https://developer.apple.com/documentation/xcode (дата звернення: 05.05.2024).
14. UIKit overview: веб-ресурс. URL: https://developer.apple.com/documentation/uikit (дата звернення: 05.05.2024).
15. SwiftUI overview: веб-ресурс. URL: https://developer.apple.com/xcode/swiftui/ (дата звернення: 05.05.2024).
16. Apple - WWDC 2019. Timecode 2:06:35 URL: https://www.youtube.com/watch?v=psL_5RIBqnY (дата звернення: 05.05.2024).
17. Introduction to RxSwift: веб-ресурс. URL: https://medium.com/@mumensh/introduction-to-rxswift-c01dcda29a30 (дата звернення: 06.05.2024).
18. Reactive Extensions Introduction: веб-ресурс. URL: https://reactivex.io/intro.html (дата звернення: 06.05.2024).
19. Model-View-Controller (MVC) in iOS – A Modern Approach: веб-ресурс. URL: https://www.kodeco.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach (дата звернення: 06.05.2024).

20. MVP Architecture: веб-ресурс. URL: https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3 (дата звернення: 06.05.2024).
21. MVVM in iOS Swift: веб-ресурс. URL: https://medium.com/@abhilash.mathur1891/mvvm-in-ios-swift-aa1448a66fb4 (дата звернення: 06.05.2024).
22. iOS Project Architecture: Using VIPER: веб-ресурс. URL: https://cheesecakelabs.com/blog/ios-project-architecture-using-viper/ (дата звернення: 06.05.2024).
23. CocoaPods: веб-ресурс. URL: https://cocoapods.org/ (дата звернення: 07.05.2024).
24. Swift Package Manager: веб-ресурс. URL: https://www.swift.org/documentation/package-manager/ (дата звернення: 07.05.2024).
25. Carthage Tutorial: веб-ресурс. URL: https://www.kodeco.com/7649117-carthage-tutorial-getting-started (дата звернення: 07.05.2024).

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Климов Данило Віталійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-206@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «Розробка iOS застосунку, орієнтованого на автоаматорів» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 14.06.2024 _____ Климов Данило Віталійович
(підпис) (прізвище та ініціали) (студент)

Дата 15.06.2024 _____ Міхайлуца Олена Миколаївна
(підпис) (прізвище та ініціали) (керівник)