ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка IOS-застосунку для зберігання та обліку карток лояльності "U card" мовою Swift»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро ЖИЛІНСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Дмитро ЖИЛІНСЬКИЙ

Керівник: Вікторія ЖЕБКА
д.т.н., професор

Рецензент: _____

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Жилінському Дмитру Іллічу

1. Тема кваліфікаційної роботи: «Розробка IOS-застосунку для зберігання та обліку карток лояльності "U card" мовою Swift»
керівник кваліфікаційної роботи д.т.н., професор, завідувач кафедри ТЦР Вікторія ЖЕБКА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «27» лютого 2024 р. № 36.

2. Строк подання кваліфікаційної роботи «28» травня 2024 р.
3. Вихідні дані до кваліфікаційної роботи:
3.1.Науково-технічна література стосовно проектування iOS застосунків.
3.2 Науково-технічна література стосовно розробки інтерфейсів користувача для iOS застосунків.
3.3. Технології прототипування інтерфейсів користувача.
3.4. Технології розробки iOS застосунків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
4.1.Аналіз предметної області.
4.2.Аналіз існуючих програмних рішень на ринку.
4.3.Формування вимог до програмного забезпечення з менеджменту персональних карток лояльності.

4.4.Розгляд основних інструментів для створення iOS застосунку.

4.5.Створення iOS додатку відповідно до вимог.

4.6.Опис процесу реалізації iOS застосунку.
4.7.Опис процесу тестування створеної програми.

5. Перелік графічного матеріалу: *презентація*

5.1. Аналіз існуючих рішень-аналогів.

5.2.Вимоги, сформовані до розроблюваного додатку.

5.3.Огляд програмних засобів для реалізації додатку.

5.4.Результати проектування додатку.
5.5. Екранні форми додатку.
5.6.Апробація результатів дослідження.

6.Дата видачі завдання «28» лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	28.02.-06.03.2024	
2	Аналіз та дослідження існуючих аналогів	07.03-13.03.2024	
3	Дослідження програмної сторони додатків-аналогів	14.03.-20.03.2024	
4	Моделювання розроблюваного програмного продукту	20.03.-30.03.2024	
5	Процес розробки iOS застосунку	30.03.-15.04.2024	
6	Створення звітної частини роботи	15.04.-25.04.2024	
7	Оформлення роботи: вступ, висновки, реферат	29.04.-05.05.2024	
8	Розробка демонстраційних матеріалів	06.05.-12.05.2024	
9	Попередній захист роботи	13.05.-31.05.2024	

Здобувач вищої освіти _____ Дмитро ЖИЛІНСЬКИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Вікторія ЖЕБКА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 53 стор., 1 табл., 17 рис., 13 джерел.

Мета роботи – Полегшення процесу організації та використання карток лояльності для клієнтів, підняття рівня залучення клієнтів до програм лояльності, та їх переваг за допомогою програмного забезпечення з менеджменту клієнтських карток лояльності

Об'єкт дослідження – Процес організації, зберігання та використання персональних карток лояльності клієнта.

Предмет дослідження – програмне забезпечення для організації, зберігання та використання особистих даних програм лояльності клієнта

Короткий зміст роботи: У цій роботі був проведений аналіз технологій розробки програми під iOS для оптимізації організації карток лояльності клієнтів. Під час дослідження визначено ключові процеси та вимоги до програмного забезпечення, необхідні для створення швидкої в роботі, та орієнтованої на користувача програми. Для проектування використання такі методології діаграм UML, задля забезпечення чіткого контролю над процесом розробки програмного забезпечення, та досягнення реалізації бажаного функціоналу.

В результаті аналізу для розробки програмного забезпечення були обрані такі технології, як Swift, Swift UI, Core Data, та Xcode для найбільш оптимальної реалізації додатку під платформу iOS

Сферою використання застосунку є організація персональних карток лояльності середньостатистичним користувачем системи iOS, що являється членом великої кількості клієнтських програм лояльності.

КЛЮЧОВІ СЛОВА: iOS, Swift, розробка додатку, картка лояльності, програма лояльності, інтерфейс користувача, Core Data

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

iOS	-	iPhone Operating System
ОС	-	Операційна Система
UI	-	User Interface
NFC	-	Near Field Communication
QR-code	-	Quick Response code
UPC	-	Universal Product Code
EAN	-	European Article Number
UML	-	Unified Modeling Language
ORM	-	Object-Relational Mapping
API	-	Application Programming Interface
IDE	-	Integrated Development Environment

ВСТУП

В умовах сьогодення, коли вподобання споживачів і лояльність до бренду відіграють ключову роль у формуванні успіху бізнесу, управління картками лояльності клієнтів є серйозною проблемою. З поширенням програм лояльності в різних галузях у споживачів накопичується все більше пластикових карток та брелків, що призводить до безладу, незручностей і втрати зацікавленості у їх використанні. Ця фрагментація не тільки ускладнює роботу користувача, але й створює матеріально-технічні та операційні перешкоди для компаній, які, за допомогою програм лояльності, прагнуть контролю за потоком продукту, прибутком, та залучення нових клієнтів.

Додаток iOS для керування персональними картками лояльності пропонує комплексне вирішення проблем, пов’язаних із фрагментованими та громіздкими методами користування програмами лояльності. Надаючи користувачам централізовану платформу для зберігання, упорядкування та доступу до їхніх карток лояльності в цифровому вигляді, програма усуває потребу у фізичних картках, брелках або кількох інших програмах для кожного підприємства. Завдяки інтуїтивно зрозумілим функціям, таким як сканування карток, їх персоналізація, та категоризація - користувачі можуть зі значно меншою кількістю зусиль користуватись перевагами програм лояльності.

Унаслідок підвищеної зручності використання карток лояльності, варто очікувати і підвищення кількості залучених до програм лояльності клієнтів. Збільшення залученості клієнтів покращує відстеження продажів і прибутків компаній, сприяючи глибшому розуміння потреб клієнтів. При більш активної участі клієнтів у програмі лояльності збільшується їх мотивація робити повторні покупки та витрачати більше на бренд, щоб отримати додаткові винагороди та переваги. Такий підвищений рівень взаємодії надає компаніям цінні відомості про поведінку клієнтів, уподобання та моделі покупок, дозволяючи уточнювати маркетингові стратегії, адаптувати рекламні акції та оптимізувати пропозиції продуктів для кращого задоволення потреб клієнтів. Відстежуючи взаємодію з

клієнтами та транзакції в рамках програми лояльності, компанії можуть отримати більш чітке уявлення про життєву цінність своїх клієнтів, рівень утримання та загальну прибутковість, що в кінцевому підсумку сприяє стійкому зростанню та прибутковості в довгостроковій перспективі.

Отже, програмне рішення з управління картками лояльності демонструє високу актуальність, та потенціал оптимізації процесів використання карток лояльності клієнтами, і як наслідок — збільшення їх залученості до клієнтських програм компаній, та вигідних пропозицій, що ті надають.

Об'єкт дослідження — Процес організації, зберігання та використання персональних карток лояльності клієнта.

Предмет дослідження — програмне забезпечення для організації, зберігання та використання особистих даних програм лояльності клієнта.

Мета роботи — полегшення процесу організації та використання карток лояльності для клієнтів, підняття рівня залученості клієнтів до програм лояльності, та їх переваг за допомогою програмного забезпечення з менеджменту клієнтських карток лояльності

Враховуючи мету роботи, до її виконання були встановлені такі задачі:

1. Виконати огляд предметної області
2. Виконати аналіз існуючих програмних рішень в сфері управління особистими картками програм лояльності
3. Провести формулювання функціональних і нефункціональних вимог до розробки програми зберігання карт лояльності
4. Виконати огляд доступних програмно-технічних засобів для створення додатків по зберігання карток лояльності
5. Обрати інструмент розробки, що надає змогу реалізувати всі необхідні функції програмного забезпечення
6. Розробити програмне забезпечення відповідно до встановлених вимог

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність теми, та особливості систематизації персональних карток лояльності

Тема організації персональних карток лояльності є актуальною в сучасному світі через широке поширення програм лояльності в різних галузях і все більшу залежність від цифрових рішень для повсякденних завдань.

Оскільки споживачі накопичують постійно зростаючу кількість карток лояльності від різних продавців і брендів, потреба в зручному та ефективному способі керування цим членством стає першорядною. Крім того, перехід до безконтактних транзакцій за допомогою мобільних додатків ще більше підкреслює важливість рішень для цифрових карток лояльності, надаючи користувачам кращий інтегрований досвід, одночасно дозволяючи підприємствам ефективно стимулювати залучення клієнтів і лояльність.

Серед інструментів та цифрових рішень задля забезпечення ефективного керування особистими картками постійного клієнта варто виділити варіант додатка, що забезпечує максимальну централізацію всіх із наявних у користувача варіантів карток. Користування таким додатком являється важливим елементом сучасного життя через його глибокий вплив на зручність, фінансову економію та залучення споживачів. В епоху, коли споживачі завалені безліччю програм лояльності від різних продавців і постачальників послуг, здатність просто та зручно організувати ці членства та отримати доступ до них безпосередньо впливає на повсякденні справи та купівельну поведінку. Централізувавши картки лояльності на єдиній цифровій платформі, люди можуть безперешкодно використовувати переваги під час транзакцій, покращуючи свій досвід покупок. Крім того, персоналізоване керування картками лояльності сприяє глибшому відчуттю зв’язку та лояльності між споживачами та брендами, посилюючи взаємодію та зміцнюючи довгострокові відносини, які виходять за рамки простих транзакцій, зрештою

збагачуючи життя та створюючи взаємну цінність як для споживачів, так і для компаній.

Отже можна стверджувати, що цифрова платформа для керування особистими картками лояльності є важливою частиною життя людей, що на регулярній основі здійснюють покупки, або користуються послугами певних компаній. За даними дослідження Mastercard 2021 року, 72% українців знайомі з програмами лояльності, проте лише 58% з них ними користуються, а опитування Gradus Research 2022 року показало, що 65% з українців, що користуються програмами лояльності вважають їх корисними, а 42% з них змінюють свою поведінку покупки, щоб отримати переваги від цих програм. За цими даними можна стверджувати, що лише близько четвертої частини із опитаних являються активними користувачами програм лояльності. Звісно зі звітів окремих компанії можна отримати і більш позитивно статистику. Наприклад, "Нова Пошта" повідомила, що у 2023 році 82% її активних користувачів були учасниками програми лояльності "Пошта Плюс". Проте через велику кількість таких компаній та відсутність універсальних методів менеджменту персональних програм лояльності, — активні користувачі однієї з них часто майже не користується перевагами інших компаній із менш зручними умовами.

Розглядаючи тему процесу централізації персональних карток лояльності для клієнтів варто виділити кілька ключових кроків, спрямованих на консолідацію та організацію їх членства в єдиній цифровій платформі. По-перше, користувачам потрібно додати свої наявні картки лояльності до програми, що може включати такі параметри, як введення вручну, сканування фізичних карток за допомогою камери пристрою або імпортування цифрових версій, наприклад, з галереї. Після користувач матиме змогу скористуватися функціями для організації та категоризації цих карток, дозволяючи користувачам групувати їх за роздрібним продавцем, типом програми або іншими відповідними категоріями для полегшення доступу та керування.

До основних, характеристик дій з персональними карт картками лояльності входять:

1. Тип операції:

До типів операцій входять додавання нових карт лояльності, упорядкування їх та вже існуючих карток за категоріями, перегляд та редагування даних збережених карт, їх демонстрація та безперешкодне ідентифікація апаратом зчитування.

2. Тип ідентифікатора:

До основних типів особистого ідентифікатора в системах лояльності входять: QR-код, Штрих-код, NFC-чіп, та персональний номер ідентифікації, що частіше за все закодований одними з попередніх способів.

3. Метод ідентифікації:

- Процес ідентифікації може проходити в декількох формах:
- Скануванням візуальних кодів (штрих-коду чи QR-коду) на касах чи спеціальних терміналах
- Сканування електромагнітного чіпа NFC за допомогою безконтактних цифрових терміналів
- Введення персонального номеру ідентифікації вручну, або оператором терміналу.

4. Особливості доступності:

До параметрів доступності додатку входить адаптований для користувачів із обмеженими руховими можливостями інтерфейс. Принципи розробки такого інтерфейсу також роблять додаток зручнішими у сценаріях використання однією рукою, Для полегшення проведення операцій по ідентифікації.

Ці характеристики безпосередньо впливають на якість роботи програми для користувачів, забезпечуючи безперебійну функціональність та зручний доступ до інформації картки постійного клієнта, що в кінцевому підсумку оптимізує ефективну взаємодію та користувача із додатком.

1.2 Особливості розробки iOS додатку

Специфіка розробки додатків для iOS визначається кількома ключовими факторами до основних входить цільова аудиторія, необхідні функції, технічні вимоги, та особливості дизайну інтерфейсу. Розуміння цих факторів під час розробки та впровадження програми має вирішальне значення для створення успішної програми для iOS, яка відповідає потребам і очікуванням користувачів. У процесі дослідження предметної області, отримується цінна інформація про необхідні функціональні можливості та елементи дизайну додатку, дозволяючи відповідним чином адаптувати вимоги до розроблюваного програмного забезпечення. Також, врахування технічних вимог, таких як особливості платформи для взаємодії, оптимізація продуктивності та заходи безпеки, гарантує безперебійну роботу програми в основних сценаріях використання. Зрештою, коректний вибір інструментів, та технології розробки, значним чином ви впливають на кінцеву якість роботи продукту, та досвід користувача у його використанні.

Одним із найважливіших аспектів розробки додатку iOS є проектування інтуїтивно зрозумілого інтерфейсу користувача, оскільки він являється першочерговим фактором, що впливає на досвід користувачів, їх залучення та утримання. Розробка інтерфейсу простим у навігації, візуально привабливим та інтуїтивно зрозумілим у використанні, дозволяє створити приємний досвід для користувачів, скорочуючи криву навчання та збільшуючи рівень впровадження. Крім того, інтуїтивно зрозумілий інтерфейс користувача сприяє позитивній взаємодії з користувачем, що з часом призводить до збільшення використання та лояльності. Зрештою, надання пріоритету інтуїтивно зрозумілому інтерфейсу користувача не тільки покращує загальну взаємодію з користувачем, але й зміцнює конкурентні переваги програми на ринку, сприяючи залучення більшої кількості користувачів і успіху в бізнесі.

Розробка програми з розрахунком на її подальшу масштабованість пропонує численні потенційні переваги, включаючи здатність пристосуватися до зростання та впоратися зі збільшеним попитом користувачів без шкоди для продуктивності

чи взаємодії з користувачем. Масштабованість гарантує, що програма залишається ефективною, навіть якщо кількість користувацьких даних та їх типів збільшується, що дозволяє ефективно масштабувати актуальність функції програми , Та як наслідок — підтримувати конкурентну спроможність на ринку. Крім того, масштабована програма дозволяє заощадити кошти за рахунок оптимізації використання ресурсів і мінімізації потреби в частих оновленнях або модернізації інфраструктури, що в кінцевому підсумку підвищує довгострокову життєздатність і успіх програми в умовах постійного оновлення, та конкуренції на ринку.

Загалом, розгляд специфіки розробки застосунків для iOS має важливе значення для створення успішних додатків, які відповідають потребам користувачів, дотримуються вказівок щодо платформи та використовують особливості екосистеми iOS. Розуміння таких факторів, як уподобання цільової аудиторії, технічні вимоги та особливості дизайну, дає можливість адаптувати програми для забезпечення оптимальної взаємодії з користувачем і досягнення бізнес-цілей. Крім того, врахування особливостей платформи та найкращих практик гарантує безперебійну роботу програми на всіх пристроях iOS, максимізацію продуктивності та підтримку сумісності з майбутніми оновленнями. Зрештою, розуміння специфіки розробки додатків для iOS підвищує конкурентоспроможність, зручність використання та довгостроковий успіх додатків на ринку додатків, що постійно розвивається.

1.3 Аналіз аналогів

Для визначення особливостей предметної області даної роботи, та реалізації поставлених у ній задач було проведено аналіз існуючих рішень.

Далі проведено розгляд п’яти проаналізованих існуючих програмних рішень:

1. Stocard: Переваги:

- Можливість персоналізації карток: Наявність у користувача можливості змінювати дані та зовнішній вигляд карток під себе
- Наявність категорій карток: Користувач може сортувати картки з

алфавітним списком, або додавати їх до створеної категорії (компанії з програмою лояльності)

- Адаптація під український ринок споживачів: додаток має велику базу підприємств із програмами лояльності, в тому числі і українських.
- Актуальна технічна підтримка застосунку: Додаток активно підтримується на даний момент. Додаються нові шаблони для карток лояльності, та регулярно виправляються незначні помилки в роботі.

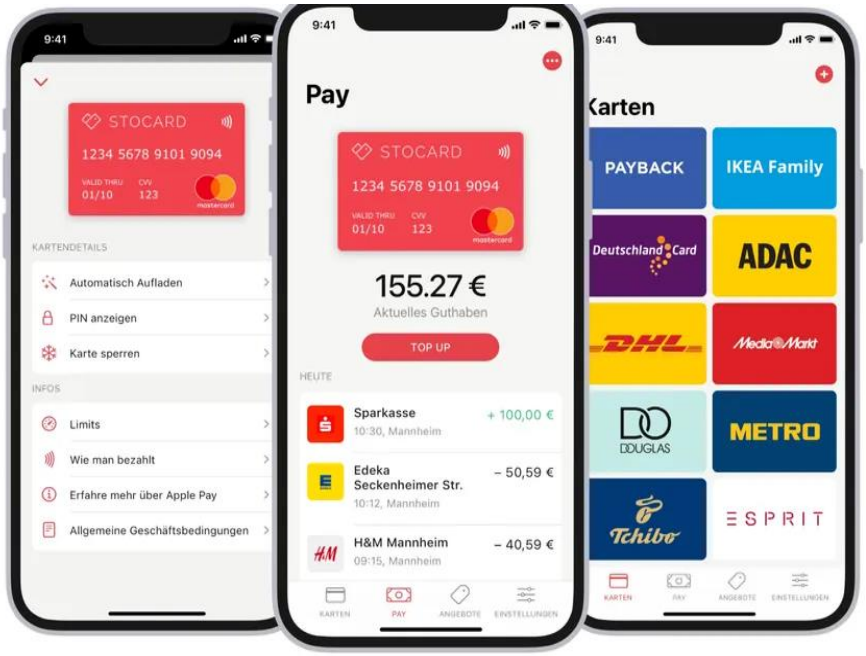


Рис 1.1 Екранні форми інтерфейсу Stocard0

Недоліки:

- Інтерфейс користувача: додаток входить у п’ятірку найпопулярніших ринку, проте не в трійку лідерів через не найкращу оптимізацію інтерфейсу користувача. Більшість користувачів вважають його пере насиченим, та менш зручним за конкурентів

- Оптимізація швидкодії для девайсів Apple: додаток має версію для приладів на системі iOS приладів з операційною системою iOS, проте не був першочергово розроблений під неї, що призводить до недостатку в його оптимізації.

1 Key Ring:

Переваги

- Користувацький інтерфейс: грамотно адаптований для девайсів на операційній системі iOS інтерфейс дуже простий, та інтуїтивний у користуванні навіть для нових користувачів.
- Оптимізація для девайсів Apple: Додаток не розроблено виключно для девайсів на операційній системі iOS, проте має досить високий рівень оптимізації швидкодії для них. Відсутність багів та помилок позитивно впливає на досвід користування.
- Можливість персоналізації карток: користувачі мають можливість редагувати дані та зовнішній вигляд особисто створених карток
- Можливість персоналізації категорій карток: додаток надає можливість сортування карток за категоріями
- Актуальна технічна підтримка застосунку: на момент тестування аналог додаток має активну технічну підтримку, що проводить регулярні оновлення задля оптимізації його роботи.

Недоліки:

- Адаптація під український ринок споживачів: даний аналог немає ні шаблонів, Адаптованих під український ринок, ні користувацького інтерфейсу, візуально знайомого для українського користувача.

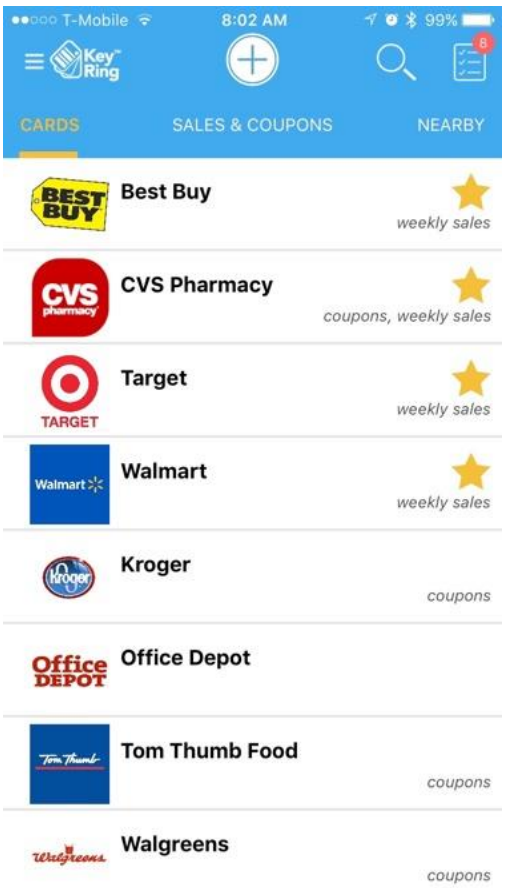


Рис. 1.2 Приклади інтерфейсу Key Ring

2. Google pay:

Переваги:

- Адаптація під український ринок споживачів: додаток здатний автоматично створювати зовнішній вигляд навіть карток локальних компаній виходячи із завантажених даних.
- Актуальна технічна підтримка застосунку: додаток є 1 із лідируючих на ринку, та має постійну активну технічну підтримку.

Недоліки:

- Інтерфейс користувача: багатьом клієнтам із великою кількістю карток лояльності інтерфейс Google Pay здається недостатньо інтуїтивним для повного переходу на нього.
- Оптимізація для девайсів Apple: додаток не є належним чином оптимізований до девайсів на операційній системі iOS, та навіть не підтримується офіційним магазином додатків Apple Store.

- Можливість персоналізації карток: функція користувацької персоналізації зовнішнього вигляду карток доступна не для всіх доданих карток, та на всіх із доступних працює коректно.
- Можливість персоналізації категорій карток: У додатку відсутня можливість розбиття карток по категоріям, створеним користувачем.

Рис. 1.3 Приклад екранних форм Google pay

3. Apple Wallet:

Переваги:

- Оптимізація швидкодії для девайсів Apple: додаток розроблений виключно для девайсів еко системи Apple, Та як наслідок має найвищий рівень оптимізації для девайсів операційна система iOS серед конкурентів.
- Можливість персоналізації карток: додаток має можливість автоматичного створення дизайну карток, як і їх редагування за бажанням користувача.
- Актуальна технічна підтримка застосунку: додаток є лідером на ринку серед рішень для девайсів на операційній системі iOS, та має активну технічну підтримку і регулярні системні оновлення для покращення його роботи, та досвіду використання користувачами.

Недоліки:

- Інтерфейс користувача: як у аналога Google Pay велика кількість користувачів Apple Wallet вважає інтерфейс цього додатку занадто складним, та пере насиченим для повної централізації власних карток за його допомогою.
- Персоналізація категорій карток: додаток має певне сортування карток, проте немає можливості створення користувача власних категорії, що є великим недоліком при наявності у клієнта великої кількості карток лояльності.
- Адаптація під український ринок споживачів: додаток не має адаптованого особливим чином інтерфейсу до українського ринку споживачів.

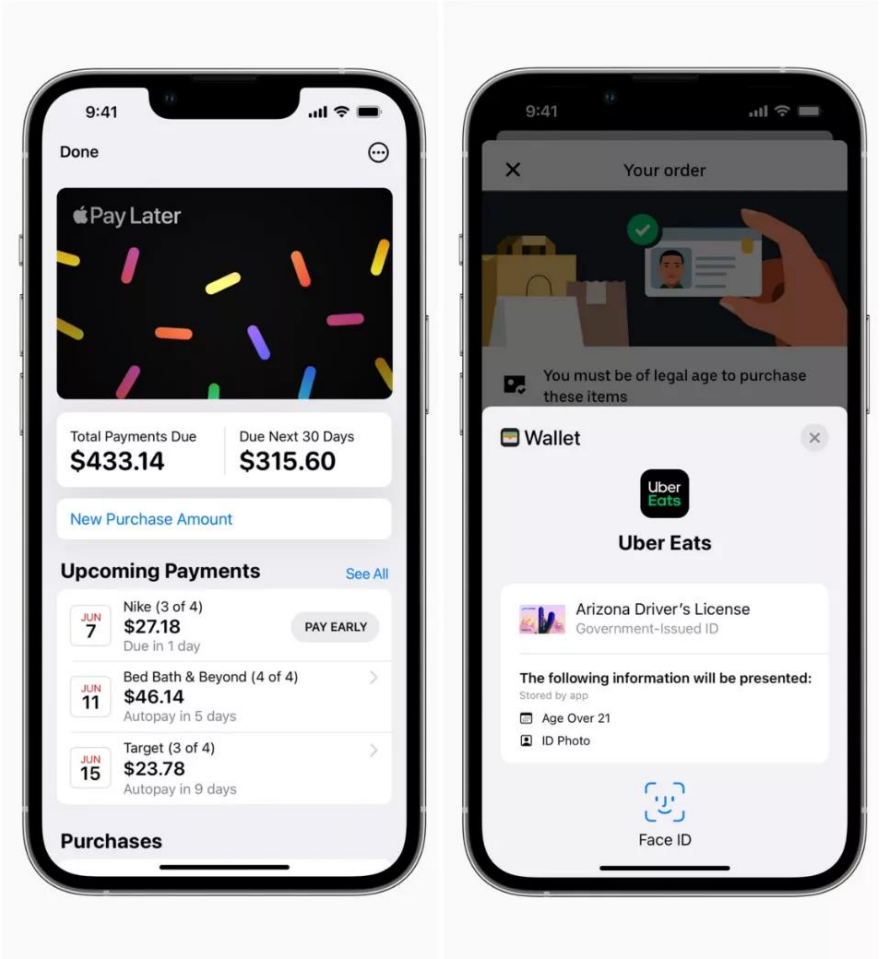


Рис. 1.4 Приклад екранних форм Apple Wallet

4. Samsung Pay:

Переваги:

- Спрощений до сприйняття інтерфейс: додаток має максимально простий до для сприйняття інтерфейс, навіть для нових користувачів.

- Можливість персоналізації карток: додаток має шаблони найпоширеніших мереж із програмами лояльності для автоматичного створення зовнішнього вигляду карток, а також можливість редагування зовнішнього вигляду за бажанням користувача.
- Можливість персоналізації категорій карток: додаток має категорії для карток за замовченням, а також можливість сортувати картки за бажанням клієнтів.

Недоліки:

- Недоступність для девайсів Apple: додаток розроблений компанією Samsung для їх власних давайсів, та деяких девайсів на системі андроїд. Як наслідок — додаток недоступний на платформі iOS.
- Адаптація під український ринок споживачів: додаток не був розроблений з метою адаптації інтуїтивності його інтерфейсу під український ринок.
- Актуальна технічна підтримка застосунку: на момент тестування компанія Samsung повністю припинила технічну підтримку додатку. Його функціонал доступний лише на попередніх версіях за умови відмови від оновлення.

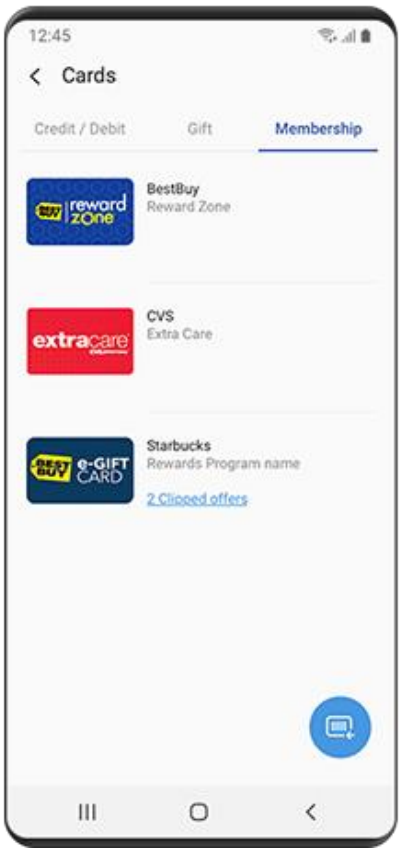


Рис. 1.5 Приклад інтерфейсу Samsung Pay

Відповідно до описаних вище переваг та недоліків програмних аналогів на ринку, була створена таблиця (Таблиця 2.1) з інформацією про їх функціональні можливості для візуалізації порівняння їх із розроблюваним програмним забезпеченням.

Таблиця 2.1

Результуюча таблиця порівняння аналогів

	Stocard	Key Ring	Google pay	Apple Wallet	Samsung Pay	U-Card
Спрощений до сприйняття інтерфейс	—	+	—	—	+	+
Оптимізація швидкодії для девайсів Apple	—	+	—	+	—	+
Можливість персоналізації карток	+	+	—	+	+	+
Можливість персоналізації категорій карток	+	+	—	—	+	+
Адаптація під український ринок споживачів	+	—	+	—	—	+
Актуальна технічна підтримка застосунку	+	+	+	+	—	+

З цієї таблиці можна наглядно побачити, що розроблюваий додаток Має функції, відсутні у додатків аналогів. Наприклад спрощений до сприйняття інтерфейс, оптимізація швидкодії для девайсів Apple та можливість персоналізації категорій карток. Ці функціональні можливості і є найважливішими ключовими елементами у власному додатку, та роблять його розробку актуальною для вирішення розглянутої у роботі проблеми.

1.4 Визначення функціональних та нефункціональних вимог

Встановлення функціональних і нефункціональних вимог до програмного забезпечення для організація персональних карт лояльності має високу важливість, оскільки встановлює чіткі вказівки та очікування щодо процесу розробки. Функціональні вимоги окреслюють конкретні функції та можливості, які повинно виконувати програмне забезпечення для задоволення потреб користувачів, гарантуючи, що програма відповідає всім основним необхідним функціям. Нефункціональні вимоги зосереджені на таких аспектах, як продуктивність, безпека та зручність використання, які мають вирішальне значення для забезпечення високоякісної та надійної взаємодії з користувачем.

Виходячи із отриманих даних після проведення аналізу предметної області були виділені, та встановлені наступні основні вимоги до розроблюваного програмного забезпечення під пристрої на операційній системі iOS, використовуючи мову програмування Swift.

1. Демонстрація інформації збережених карток для їх застосування:

- Можливість демонстрації ідентифікатора картки з метою його застосування
- Безперешкодне зчитування терміналом продемонстрованого ідентифікатора
- Можливість демонстрації детальної інформації про збережену картку
- Можливість редагування користувачем збереженої раніше інформації
- Можливість додавати нову інформацію до поточної картки

2. Можливість організації карток, та їх списків користувачем:

- Можливість створення користувачем власних категорій для сортування карток
- Можливість перейменування користувачем створених категорій окрім категорії за замовчуванням (“всі картки”)
- Можливість зберігати додавати декілька карток до одної категорії
- Можливість додавання користувачем одної картки до декількох

категорій

3. Додавання будь-яких наявних у клієнта карток:

- Можливість додавання картки з QR-кодом
- Можливість додавання картки зі штрих-кодом
- Можливість додавання картки з NFC-чіпом

4. Оптимізація інтерфейсу для забезпечення доступності:

- Використання стандартів для розробка інтерфейсу з урахуванням користувачів з обмеженими можливостями, А також сценаріїв використання додатку однією рукою в умовах обмеженої мобільності.

5. Доступ до збережених карток навіть за умов відсутності підключення до мережі.

6. Оптимізована швидкість роботи додатку:

- Забезпечити максимальну швидкодію додатку за допомогою Swift та оптимізованої для iOS інтегрованої середи розробки Xcode.

7. Сумісність з усією лінійкою портативних девайсів Apple на iOS:

- За допомогою інструментів середи розробки Xcode забезпечити адаптивність додатку до всієї лінійки портативної давай сів iOS.

8. Оптимізація користувацького інтерфейсу:

- Створення користувацького інтерфейсу в мінімалістичнийму стилі, та за правилами максимальної його ефективності.
- Зменшення нагромадженості інтерфейсу задля забезпечення інтуїтивності використання додатку.

2 ДОСЛІДЖЕННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ ТА АРХІТЕКТУРНИХ РІШЕНЬ IOS ДОДАТКУ

2.1 Вибір мови програмування

Вибір мови програмування має першочергове значення у розробці мобільних додатків, оскільки безпосередньо впливає на ефективність, продуктивність і масштабованість кінцевого продукту. Вибір правильної мови програмування відповідно до вимог платформи та екосистеми розробки є важливим для досягнення бажаної функціональності та взаємодії з користувачем. Наприклад, у контексті розробки додатків для iOS, Swift став переважною мовою завдяки своєму сучасному синтаксису, функціям безпеки та оптимізації продуктивності. Використовуючи потужність Swift, розробники можуть писати чистий і стислий код, що сприяє швидшому виконанню додатків, зменшенню витрат пам’яті та покращенню загальної продуктивності. Крім того, сумісність Swift із інструментами та фреймворками Apple, такими як Xcode, оптимізує процес розробки, дозволяючи розробникам ефективно використовувати специфічні для платформи функції та API, що призводить до створення високоякісних, швидких і багатофункціональних програм для iOS.

Вибір же невідповідної мови програмування може призвести до різноманітних проблем і обмежень під час процесу розробки, перешкоджаючи функціональності програми, її зручності обслуговування та продуктивності. Наприклад, вибір мови з низькою продуктивністю або недостатньою підтримкою цільовою мобільної платформи може призвести до повільної та невідповідної роботи програм, які не відповідають очікуванням користувачів. Крім того, проблеми із сумісністю та відсутність інтеграції з інструментами та бібліотеками, що стосуються певної платформи, можуть збільшити час і складність розробки, що ускладнює надання відшліфованої та цілісної взаємодії з користувачем. Тому при плануванні процесу розробки були ретельно оцінені вимоги до програмного забезпечення і враховані такі фактори, як сумісність платформи, характеристики

продуктивності та підтримка екосистеми, щоб гарантувати відповідність цільовій аудиторії, платформі, поставленим функціональним і не функціональним вимогам.

Тож для реалізації поставлених до проекту задач була обрана мова програмування Swift. Створення привабливих та інтуїтивно зрозумілих інтерфейсів користувача для додатків iOS розробка інтерфейсної логіки передбачає використання потужних функцій і інструментів, розроблених спеціально для створення додатків під дану операційну систему. У контексті розробки додатків для операційної системи iOS такими являються Swift і SwiftUI.

Swift — це сучасна, потужна та надзвичайно експресивна мова програмування, розроблена Apple для створення програм для девайсів власної платформи, в особливості під операційну систему iOS. Однією з ключових особливостей Swift є його безпека та надійність, досягнута завдяки поєднанню надійної типізації, опцій та механізмів керування пам’яттю. Надійна система типів Swift забезпечує безпеку типів під час компіляції, зменшуючи ймовірність помилок під час виконання та підвищуючи надійність коду. Крім того, Swift представляє опції, які надають механізм безпечної та контрольованої обробки нульових значень, запобігаючи виняткам нульового покажчика та зменшуючи ризик збоїв. Також, механізм автоматичного підрахунку посилань (ARC) Swift автоматично керує пам’яттю, знімаючи об’єкти, коли в них зникає потреба, тим самим зменшуючи ймовірність витоку пам’яті та покращуючи продуктивність.

Іншим важливим аспектом Swift є його сучасний синтаксис і мовні функції, які підвищують продуктивність розробника та читабельність коду. Swift використовує стислий і виразний синтаксис, натхненний іншими сучасними мовами програмування, такими як Python і Ruby, що полегшує написання чистого коду, та зручність його підтримки. Такі функції, як замикання, узагальнення та зіставлення шаблонів, дозволяють писати потужний і гнучкий код із меншою кількістю рядків, зменшуючи шаблонність і покращуючи чіткість коду. Крім того, підтримка Swift парадигм функціонального програмування, таких як функції відображення, фільтрування та скорочення, дозволяє писати код у більш декларативному та виразному стилі, сприяючи застосуванню принципу DRY

(створення коду без повторень) та підвищуючи зручність обслуговування коду.

Крім того, природа Swift з відкритим кодом і активна спільнота сприяють його швидкому розвитку та прийняттю з регулярними оновленнями, покращеннями та внесками розробників з усього світу. Загалом поєднання безпеки, надійності, сучасного синтаксису та сумісності робить Swift переконливим вибором для розробки додатків iOS, надаючи можливість розробникам створювати високоякісні, продуктивні та придатні для обслуговування програми для платформ Apple.

2.2 Фреймворки та бібліотеки для обраної мови програмування і предметної області

Фреймворки в контексті розробки програмного забезпечення — це багаторазово використовувані колекції попередньо написаного коду, бібліотек і інструментів, які забезпечують основу для створення програм або вирішення конкретних проблем. З технічної точки зору фреймворки зазвичай складаються з набору API (інтерфейсів прикладного програмування) та абстракцій, які визначають інтерфейс між кодом програми та основною системою чи платформою. Ці API абстрагуються від низькорівневих деталей і надають функціональні можливості та послуги вищого рівня, дозволяючи розробникам на реалізації логіки конкретного додатка без необхідності заново винаходити колесо для типових завдань.

Фреймворки бувають різних форм, включаючи бібліотеки, фреймворки інтерфейсу і фреймворки повного стека, кожен з яких служить різним цілям і рівням абстракції. Бібліотеки — це набори функцій, класів або модулів, які забезпечують певну функціональність, наприклад роботу в мережі, доступ до бази даних або компоненти інтерфейсу користувача. З іншого боку, фреймворки з повним стеком надають наскрізні рішення для створення цілих додатків, включаючи як зовнішні, так і внутрішні компоненти, часто дотримуючись певного архітектурного шаблону або філософії дизайну.

Підсумовуючи, фреймворки є важливими інструментами в розробці

програмного забезпечення, які полегшують дотримання принципів DRY, підвищують продуктивність і дозволяють створювати програми більш ефективно. Забезпечуючи готові компоненти, абстракції та API, фреймворки абстрагуються від низькорівневих деталей і шаблонного коду, дозволяючи зосередитися на реалізації логіки для конкретної програми та вирішенні проблем вищого рівня. Фреймворки відіграють вирішальну роль у розробці актуального програмного забезпечення, надаючи будівельні блоки та інструменти, необхідні для створення надійних і масштабованих програм.

2.2.1 SwiftUI

SwiftUI — це фреймворк інтерфейсу користувача, представлений Apple в iOS 13, яка пропонує декларативний підхід до побудови інтерфейсів користувача для додатків платформи Apple, включаючи операційну систему iOS. В основі SwiftUI лежить його декларативний синтаксис, який дозволяє визначати компоненти інтерфейсу користувача разом з їхніми властивостями за допомогою простого та інтуїтивно зрозумілого синтаксису. Замість того, щоб обов'язково вказувати, як має бути створений і оновлений інтерфейс користувача, достатньо оголосити бажаний стан інтерфейсу та дозволити SwiftUI автоматично обробляти рендеринг і оновлення. Цей декларативний підхід спрощує розробку інтерфейсу користувача, зменшуючи кількість коду, необхідного для опису складних макетів і взаємодій, а також покращує читабельність коду та зручність його обслуговування.

Однією з ключових особливостей SwiftUI є його обширна бібліотека вбудованих компонентів, включаючи представлення, елементи керування та модифікатори, які дозволяють створювати багаті інтерактивні інтерфейси користувача з мінімальними затратами часу. SwiftUI надає широкий спектр готових компонентів для типових елементів користувацького інтерфейсу, таких як списки, текстові поля, кнопки і навігаційні вказівки, а також розширені компоненти для анімації, жестів і переходів. Крім того, SwiftUI пропонує потужну систему модифікаторів, які дозволяють розробникам легко налаштовувати зовнішній вигляд і поведінку компонентів інтерфейсу користувача. Поєднуючи модифікатори

разом, розробники можуть застосовувати трансформації, стилі та анімацію до елементів інтерфейсу користувача, що дає їм змогу досягти широкого діапазону ефектів дизайну та взаємодії з користувачем.

Крім того, SwiftUI представляє потужні функції для керування даними, обробки стану та реактивного програмування, які спрощують розробку складних взаємодій інтерфейсу користувача та динамічних інтерфейсів користувача. Механізм зв’язування даних SwiftUI забезпечує двонаправлений зв’язок між елементами інтерфейсу користувача та джерелами даних, дозволяючи змінам інтерфейсу автоматично оновлювати базові дані та навпаки. Це дозволяє створювати користувацький інтерфейи, що керуються даними, які адаптуються до змін у режимі реального часу, забезпечуючи позитивний досвід користувача. Крім того, SwiftUI забезпечує підтримку керування станом програми за допомогою обгорток властивостей, таких як @State, @Binding і @ObservableObject, які дозволяють інкапсулювати та керувати станом у своїх представленнях і забезпечувати узгоджену поведінку користувацького інтерфейсу в різних частинах програми.

```swift
import SwiftUI

// Define a model to represent the application state
class AppState: ObservableObject {
    @Published var count: Int = 0
}

// Define a view that displays the current count and allows the user to increment it
struct ContentView: View {
    // Inject the app state into the view using @ObservedObject property wrapper
    @ObservedObject var appState: AppState

    var body: some View {
        VStack {
            Text("Count: \(appState.count)")
                .font(.largeTitle)
                .padding()

            Button(action: {
                // Increment the count when the button is tapped
                appState.count += 1
            }) {
                Text("Кількість карток:")
                    .font(.headline)
                    .padding()
                    .foregroundColor(.white)
                    .background(Color.blue)
                    .cornerRadius(10)
            }
        }
    }
}

// Define the entry point of the application
@main
struct MyApp: App {
    // Create an instance of the AppState class to manage the application state
    @StateObject var appState = AppState()

    var body: some Scene {
        WindowGroup {
            // Pass the app state to the ContentView
            ContentView(appState: appState)
        }
    }
}

```

Рис. 2.1 Приклад керування станом лічильника за допомогою обгортки властивості @State

Загалом поєднання декларативного синтаксису, вбудованих компонентів і функцій керування даними в SwiftUI робить його інноваційним інструментом для розробки iOS, даючи розробникам змогу створювати елегантні, ефективні та масштабовані інтерфейси користувача для платформ Apple.

2.2.2 Kingfisher

Kingfisher — це бібліотека Swift, призначена для ефективного завантаження та кешування зображень у програмах iOS. Однією з його видатних особливостей є простота використання, що робить його кращим вибором для покращення програм за допомогою функцій, пов'язаних із зображеннями. Kingfisher дозволяє легко інтегрувати можливості завантаження зображень і кешування в додатки для iOS з мінімальними зусиллями. Бібліотека усуває складності обробки зображень, надаючи високорівневий API, який дозволяє зосередитися на створенні основних функцій програм.

Однією з ключових особливостей Kingfisher є механізм кешування зображень, який допомагає покращити продуктивність і взаємодію з додатками iOS. Бібліотека автоматично кешує завантажені зображення в пам'яті, зменшуючи потребу в повторних мережевих запитах і прискорюючи час завантаження зображень. Цей механізм кешування гарантує, що зображення будуть легко доступними для відображення, навіть коли пристрій офлайн або мережеве з'єднання повільне. Локально кешуючи зображення, Kingfisher допомагає оптимізувати використання ресурсів, що забезпечує більш плавну та чутливу роботу додатка.

У контексті програми iOS для керування персональними картками постійного клієнта Kingfisher відіграє важливо роль у покращенні візуального представлення зображень карт постійного клієнта. Використовуючи Kingfisher для завантаження та кешування зображень карток постійного клієнта, можна гарантувати швидкий і надійний доступ до візуальних елементів карток, незалежно від умов мережі чи продуктивності пристрою. Це покращує загальну зручність використання програми та підвищує задоволеність користувачів. Крім того, механізм кешування Kingfisher допомагає зменшити використання даних і звести до мінімуму залежність програми від підключення до мережі, що робить її придатною для використання в автономних сценаріях, коли доступ до інформації картки постійного клієнта може бути критичним.

Крім того, Kingfisher пропонує ряд розширених функцій і параметрів

налаштування для більшого контролю робочих процесів завантаження зображень та кешування. Бібліотека підтримує такі функції, як зображення-заповнювачі, трансформації зображень та індикатори прогресу завантаження зображень, що дозволяє адаптувати процес завантаження зображень відповідно до конкретних вимог додатків. Крім того, Kingfisher легко інтегрується з популярними фреймворками такими як SwiftUI, що дозволяє легко включати розширені можливості обробки зображень у програми iOS, зберігаючи при цьому чисту кодову базу, яку можна підтримувати.

Загалом Kingfisher — це універсальна та багатофункціональна бібліотека, яка значно спрощує завантаження та кешування зображень у додатках iOS, що робить її важливим інструментом розробки програми для керування особистими картками постійного клієнта та інших випадків використання зображень.

```swift
import UIKit
import Kingfisher

class LoyaltyCardViewController: UIViewController {

    // UIImageView to display the loyalty card background image
    @IBOutlet weak var loyaltyCardBackgroundImageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Load the loyalty card background image from local storage
        if let image = UIImage(named: "loyalty_card_background") {
            loyaltyCardBackgroundImageView.image = image
        } else {
            print("Failed to load loyalty card background image from local storage")
        }
    }
}
```

Рис. 2.2 Приклад класу для контролю зовнішнього виду картки, створеного за допомогою Kingfisher

2.2.3 AVFoundation

AVFoundation — це структура, створена Apple для обробки мультимедійних даних у додатках iOS, що пропонує широкий спектр функцій і можливостей для керування аудіовізуальним вмістом. У контексті розробки програми iOS для керування персональними картками постійного клієнта AVFoundation можна використовувати для виконання завдань, пов’язаних зі скануванням штрих-кодів,

розпізнаванням QR-кодів і обробкою зображень. Однією з ключових особливостей AVFoundation є його підтримка захоплення медіа-даних із вхідних даних пристрою, наприклад камери, що важливо для реалізації функції сканування штрих-кодів у програмі.

За допомогою AVFoundation можна отримати доступ до камери пристрою та записувати відеокадри в режимі реального часу, забезпечуючи ефективне сканування штрих-коду та розпізнавання QR-коду. Фреймворк можливості для налаштування сеансів захоплення та керування ними, а також для обробки відеокадрів і вилучення даних штрих-коду або QR-коду. Використання цих можливостей дозволяє реалізувати надійну функцію сканування штрих-кодів у програмі iOS, дозволяючи користувачам легко сканувати та керувати своїми картками лояльності.

Крім того, AVFoundation пропонує вбудовану підтримку виявлення метаданих, які можна використовувати для розпізнавання та декодування штрих-кодів і QR-кодів із кадрів, знятих камерою. Фреймворк надає можливості для виявлення та вилучення об’єктів метаданих, включаючи символіки штрих-кодів, такі як UPC, EAN і QR-коди. Об’єкт AVCaptureMetadataOutput може бути налаштований для отримання об’єктів метаданих із сеансу захоплення та реалізувати спеціальну логіку для обробки різних типів штрих-кодів і QR-кодів, виявлених у відеопотоці.

Крім того, AVFoundation містить можливості обробки зображень, які можна використовувати для підвищення якості та точності сканування штрих-кодів і розпізнавання QR-кодів. Фреймворк надає можливість створення класів для виконання різних завдань обробки зображень, таких як фільтрування, покращення та виправлення зображень. За допомогою AVFoundation можна використовувати фільтри Core Image для попередньої обробки відеокадрів перед подачею їх у конвеєр виявлення штрих-кодів, покращуючи читабельність і надійність розпізнавання штрих-кодів і QR-кодів у складних умовах освітлення або із зображеннями низької якості.

Підводячи підсумок, AVFoundation — це багатофункціональна структура,

яка пропонує повну підтримку сканування штрих-кодів, розпізнавання QR-кодів і обробки зображень у додатках iOS. Використання його API та класів, надає можливість реалізувати надійну та ефективну функцію сканування штрих-кодів у програмі iOS для керування особистими картками постійного клієнта, дозволяючи користувачам безперешкодно сканувати, зберігати та керувати інформацією своїх карток постійного клієнта з легкістю та зручністю.

2.2.4 Core Data

Core Data — це універсальна платформа, надана Apple, яка надає численні функції та переваги для розробки додатків iOS для керування особистими картками постійного клієнта. Однією з ключових особливостей Core Data є його здатність об’єктно-реляційного відображення (ORM), яка дозволяє визначати модель даних програми за допомогою об’єктно-орієнтованих конструкцій високого рівня, таких як класи та зв’язки. Ця абстракція спрощує процес роботи зі складними моделями даних, оскільки дозволяє взаємодіяти з керованими об’єктами безпосередньо в коді, без необхідності писати запити SQL або керувати підключеннями до бази даних вручну.

Ще однією перевагою Core Data є вбудована підтримка збереження даних, яка надає можливість зберігати дані картки постійного клієнта локально на пристрої користувача. Core Data абстрагує деталі взаємодії з основною базою даних (наприклад, SQLite) і надає високорівневий API для збереження, отримання, оновлення та видалення керованих об’єктів. Це спрощує реалізацію таких функцій, як зберігання інформації картки постійного клієнта, відстеження взаємодії користувачів і синхронізаціяї даних між пристроєм і віддаленим сервером.

Core Data також пропонує потужні можливості запитів за допомогою механізму запиту на вибірку, що дозволяє отримувати певні підмножини даних із постійного сховища на основі різних критеріїв. Ця функція особливо корисна для реалізації функцій пошуку, та отримання відсортованих за категоріями карток лояльності. Використовуючи API запиту вибірки Core Data, можна реалізувати розширені функції пошуку та фільтрації, щоб покращити взаємодію з користувачем

і зручність використання.

Крім того, підтримка Core Data перевірки даних і автоматичного вирішення конфліктів допомагає забезпечити цілісність і узгодженість даних навіть у багатокористувацьких середовищах або під час синхронізації даних між кількома пристроями. Це зменшує ризик пошкодження або втрати даних і забезпечує користувачам досвід стабільності під час керування своїми картками постійного клієнта. Крім того, інтеграція Core Data з іншими фреймворками iOS, як SwiftUI, полегшує розробку інтуїтивно зрозумілих і візуально привабливих інтерфейсів користувача для програм керування картками постійного клієнта, дозволяючи створювати захоплюючі враження, які заохочують користувачів продовжувати взаємодію з програмою. Загалом, великий набір функцій Core Data та бездоганна інтеграція з iOS роблять дану платформу найкращим вибором для створення багатофункціональних програм для керування картками постійного клієнта, які відповідають потребам сучасних користувачів.

2.2.5 Підсумок обраних фреймворків

У контексті розробки програми iOS для керування особистими картками постійного клієнта використання Swift і пов’язаних з ним інфраструктур створює комплексний набір інструментів для розробки надійного та зручного рішення. SwiftUI, декларативна структура інтерфейсу користувача Apple, забезпечує сучасний та інтуїтивно зрозумілий підхід до розробки інтерфейсів користувача, що дозволяє створити візуально привабливі макети та інтерактивні компоненти з мінімальною кількістю коду. За допомогою SwiftUI легко впроваджувати такі функції, як відображення карток, елементи керування навігацією та інтерактивні елементи, оптимізуючи процес розробки та забезпечуючи зручну взаємодію з користувачем.

Kingfisher, — бібліотека для завантаження та кешування зображень для Swift, доповнює SwiftUI, спрощуючи керування зображеннями для зовнішнього вигляду карток постійного клієнта. Kingfisher надає можливість ефективно завантажувати та відображати зображення карток із локального сховища, використовуючи

розширені функції, такі як кешування зображень, підтримка заповнювачів і трансформація зображень. Інтеграція Kingfisher у програму керування картками лояльності, дозволяє підвищити продуктивність, зменшити навантаження на пропускну здатність та забезпечити оптимізовану роботу із зображеннями.

AVFoundation — мультимедійна структура Apple, відіграє вирішальну роль у функціях сканування штрих-кодів і QR-кодів у програмі керування картками постійного клієнта. Використання AVFoundation, надає можливість отримати доступ до камери пристрою, записувати відеокадри в реальному часі та виявляти об’єкти метаданих, такі як штрих-коди та QR-коди. Це дозволяє користувачам легко додавати нові картки лояльності до програми шляхом сканування фізичних карток або цифрових кодів, спрощуючи процес керування картками.

Нарешті, Core Data служить основою для керування даними в додатку. Core Data забезпечує можливість визначати модель даних для карток постійного клієнта, зберігати інформацію про картки локально на пристрої та впроваджувати такі функції, як створення, редагування та видалення карток. Підтримка Core Data об’єктно-реляційного відображення, перевірки даних і керування зв’язками спрощує реалізацію складних структур даних і забезпечує цілісність даних у всій програмі. Використання Core Data, забезпечує масштабованість, та надійність зберігання, та обробки даних розроблюваного рішення для керування картками постійного клієнта.

Загалом, Swift і пов’язані з ним інфраструктури, включаючи SwiftUI, Kingfisher, AVFoundation і Core Data, пропонують універсальний набір інструментів для розробки програми iOS для керування особистими картками постійного клієнта. Поєднання цих фреймворків надає можливість створити багатофункціональну та зручну у використанні програму, що дозволить легко організовувати, відстежувати та використовувати свої картки лояльності.

2.2.6 Обраний інструмент для розробки програмного забезпечення

Основним інструментом для розробки програмного забезпечення для централізації процесів використання персональних карток лояльності клієнтів була обрана програма Xcode. Це інтегроване середовище розробки (IDE) від Apple, яке є основним інструментом для розробки додатків iOS. Xcode пропонує безліч переваг і функцій, адаптованих до потреб розробки програми для керування картками постійного клієнта.

Перш за все, Xcode надає комплексний набір інструментів для проектування, кодування, тестування та налагодження додатків iOS, оптимізації процесу розробки та підвищення продуктивності кінцевого продукту. Завдяки нативно зрозумілому інтерфейсу та бездоганній інтеграції з іншими інструментами розробника Apple, Xcode дає змогу максимально оптимізувати процес творення складних та візуально привабливих інтерфейсів для додатків iOS, включаючи застосунки для організації персональних даних.

Іншою з ключових переваг Xcode є його підтримка Interface Builder, редактора графічного інтерфейсу користувача, який дозволяє проектувати та прототипувати інтерфейси користувача за допомогою нативного інтерфейсу. Interface Builder надає багатий набір компонентів інтерфейсу користувача та інструментів макета, що дозволяє створювати адаптивні макети для відображення карток постійного клієнта, елементів керування навігацією, форм введення та інших функціональних елементів. Візуальне проектування інтерфейсу користувача в Interface Builder, надає можливість швидко повторювати ідеї дизайну, переглядати зміни інтерфейсу в реальному часі та забезпечує послідовність і узгодженість у всій програмі.

Крім того, Xcode пропонує надійну підтримку SwiftUI, декларативного інтерфейсу користувача Apple, представленого в iOS 13, який революціонізує спосіб створення інтерфейсів користувача для додатків iOS. Вбудований у Xcode інтерфейс SwiftUI спрощує процес створення динамічних та інтерактивних інтерфейсів, дозволяючи описувати макет і поведінку елементів інтерфейсу за

допомогою стислого та декларативного синтаксису. За допомогою SwiftUI, Xcode надає можливість використовувати такі функції, як прив’язка даних, керування станом і анімація, для створення позитивного досвіду роботи з розроблюваним програмним забезпеченням, зменшуючи при цьому об’єм необхідного до написання коду та досягаючи швидших циклів розробки.

На додаток до Interface Builder і SwiftUI, Xcode надає набір інструментів для редагування коду, налагодження та аналізу продуктивності, що дозволяє забезпечити чистоту, ефективність, та придатність коду до подальшої підтримки. Редактор коду Xcode пропонує такі функції, як підсвічування синтаксису, доповнення коду та вбудована документація, що полегшує написання коду Swift і впевнену навігацію по ньому. Вбудований налагоджувач дозволяє перевіряти змінні, встановлювати контрольні точки та поетапно виконувати код, полегшуючи ідентифікацію та вирішення помилок і проблем. Крім того, інструменти аналізу продуктивності Xcode дозволяють профілювати використання пам’яті, продуктивність процесора та мережеву активність, допомагаючи оптимізувати програму для швидкої реакції та ефективності.

Підводячи підсумок, Xcode служить потужним і незамінним інструментом для розробки додатків iOS для керування особистими картками лояльності клієнтів. Завдяки повному набору функцій, включаючи Interface Builder, підтримку SwiftUI, інструменти редагування коду, можливості налагодження та інструменти аналізу продуктивності, Xcode надає усі необхідні інструменти для розробки, реалізації та вдосконалення інтерфейсів користувача, які відповідають встановленим потребам до розроблюваного програмного забезпечення.

2.2.7 Особливості баз даних

Розробка бази даних для програми керування картками постійного клієнта з локальним зберіганням інформації про користувачів включає кілька ключових функцій і міркувань, щоб забезпечити ефективне керування даними та безперебійну роботу додатка. Перш за все, використання Core Data як основної бази даних забезпечує численні переваги, включаючи можливості об’єктно-реляційного

відображення (ORM), моделювання даних і стійкість. Визначивши модель даних для карток постійного клієнта, профілів користувачів і пов'язаних метаданих, з'являється можливість створити структуровану та ефективну схему бази даних, яка відповідає вимогам програми та робочим процесам користувачів.

Однією з основних особливостей розробки баз даних за допомогою Core Data є підтримка об'єктно-реляційного відображення, яке спрощує взаємодію між об'єктами рівня моделі програми та основною базою даних. Core Data абстрагує складність керування базами даних, дозволяючи працювати з керованими об'єктами у звичній об'єктно-орієнтованій парадигмі. Це забезпечує повну інтеграцію даних карток лояльності в користувацький інтерфейс програми, дозволяючи користувачам легко переглядати, редагувати та впорядковувати свої картки постійного клієнта.

Крім того, Core Data забезпечує надійну підтримку збереження даних, дозволяючи локально зберігати інформацію картки лояльності на пристрої користувача для доступу в автономному режимі та підвищення продуктивності. Використовуючи вбудовану підтримку Core Data для SQLite як основного постійного сховища, відкривається можливість забезпечити цілісність і надійність даних, мінімізуючи накладні витрати на зберігання та оптимізуючи використання ресурсів. Це дозволяє користувачам швидко отримувати доступ до інформації про картку постійного клієнта навіть у режимі офлайн або в середовищах із низьким доступом до Інтернету.

Окрім моделювання та збереження даних, розробка баз даних за допомогою Core Data дозволяє впроваджувати розширені функції, такі як перевірка даних, міграція та керування паралелізмом. Механізми перевірки Core Data гарантують, що дані картки постійного клієнта відповідають попередньо визначеним обмеженням і бізнес-правилам, зменшуючи ризик пошкодження чи невідповідності даних. Крім того, полегшені можливості міграції Core Data спрощують процес розвитку моделі даних з часом, забезпечуючи безперебійне оновлення та оновлення програми без шкоди для наявних даних користувача.

Нарешті, розробка бази даних за допомогою Core Data сприяє бездоганній

інтеграції з іншими фреймворками та технологіями iOS, дозволяючи використовувати додаткові функції та можливості для покращення програми керування картками постійного клієнта. Наприклад, можливість використовувати функції керування взаємовідносинами Core Data для встановлення зв’язків між картками лояльності та профілями користувачів, забезпечуючи персоналізований досвід і цільові рекламні акції. Крім того, інтеграція Core Data із SwiftUI дозволяє створювати динамічні користувацькі інтерфейси, які відображають зміни в базовій моделі даних, надаючи користувачам інтуїтивно зрозумілий досвід під час керування своїми картками лояльності. Загалом розробка баз даних за допомогою Core Data пропонує комплексне та гнучке рішення для керування інформацією про користувачів і даними карток постійного клієнта, що дозволяє створювати надійні, масштабовані та зручні програми, які відповідають встановленим до розроблюваного програмного забезпечення вимогам.

2.2.8 Особливості побудови архітектури iOS додатку

У розробці програми для iOS, архітектурні рішення відіграють вирішальну роль у формуванні структури та поведінки програмного забезпечення відповідно до встановлених вимог і очікувань користувачів. Одним з архітектурних рішень, яке добре відповідає вимогам додатка, є архітектура Model-View-ViewModel (MVVM), яка забезпечує чіткий розподіл завдань і сприяє тестуванню, придатності до обслуговування та масштабованості. В архітектурі MVVM модель представляє рівень даних, що містить інформацію про картку лояльності, що зберігаються за допомогою основних даних. Подання представляє рівень інтерфейсу користувача, реалізований за допомогою SwiftUI для створення динамічних та інтерактивних макетів для відображення та взаємодії з картками лояльності. Нарешті, модель перегляду діє як посередник між моделлю та шарами перегляду, обробляючи введення користувача, маніпулювання даними та бізнес-логіку для забезпечення безперебійного та швидкого реагування користувача.

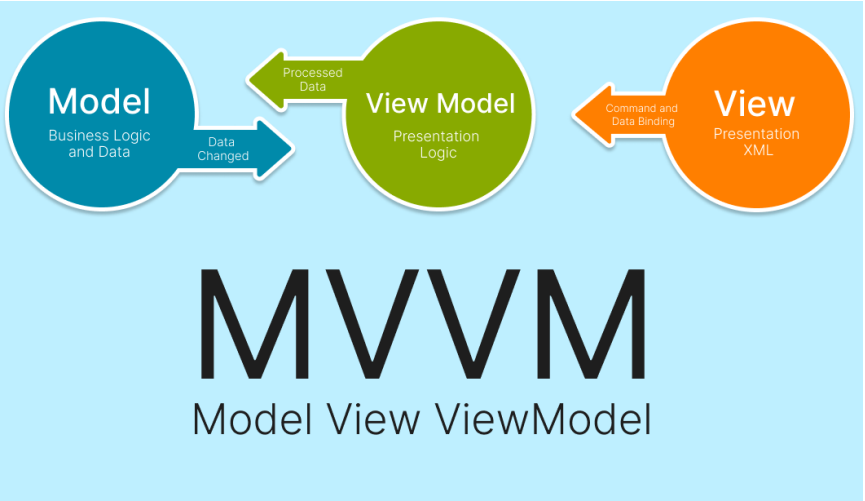


Рис. 2.3 схема архітектури Model-View-ViewModel

Іншим архітектурним аспектом програм для iOS є використання dependency injection для керування залежностями та полегшення модульності та тестування. Відокремлюючи компоненти та вводячи залежності під час виконання, створює можливість легко замінити або імітувати залежності для цілей тестування, забезпечуючи надійність і стабільність програми. Наприклад, ін'єкцію залежностей можна використовувати, щоб вставити екземпляр стеку основних даних у модель перегляду, дозволяючи йому взаємодіяти з фреймворком основних даних без тісного зв'язку з деталями реалізації бази даних.

Нарешті, враховуючи особливість офлайн-доступу та синхронізації даних, прийняття шаблону синхронізації, такого як repository pattern, може полегшити керування даними та забезпечити узгодженість даних на різних пристроях. Інкапсулюючи логіку доступу до даних в об'єктах сховища, створюється можливість абстрагуватися від складнощів синхронізації даних і вирішення конфліктів, дозволяючи програмі легко обробляти зміни в даних картки постійного клієнта як локально, так і віддалено. Цей архітектурний підхід дозволяє користувачам отримувати доступ до інформації про картку постійного клієнта навіть без підключення до інтернету та гарантує, що оновлення, зроблені на одному пристрої, синхронізуються з іншими пристроями після відновлення з'єднання. Загалом, ретельний вибір та впровадження таких архітектурних рішень, як MVVM, впровадження залежностей і шаблону сховища, дозволяє створити надійну, масштабовану та зручну програму iOS для керування особистими картками лояльності клієнтів, яка відповідає встановленим вимогам до розробки.

3 РОЗРОБКА ДОДАТКУ ДЛЯ МЕНЕДЖМЕНТУ КАРТОК ЛОЯЛЬНОСТІ

3.1 Етапи розробки програмного забезпечення

Розробка додатків для iOS зазвичай складається з кількох етапів, кожен з яких має значний вплив на кінцевий стан програмного забезпечення, та вирішальне значення в успішності реалізації розробленого продукту на ринку. Етапи розробки додатків для iOS є ітеративними та взаємопов'язаними, причому кожен етап базується на попередньому для забезпечення якості програми. Дотримуючись структурованого та дисциплінованого підходу до розробки, можна гарантувати, відповідність програми потребам користувачів, її узгодження з вимогами проекту та забезпечення задовільної взаємодії з користувачем.

Перший етап –аналіз вимог, на якому розробники збирають і документують функціональні та нефункціональні вимоги програми. Це передбачає розуміння потреб і очікувань користувачів, визначення ключових функцій і функцій, а також визначення обсягу проекту. Аналіз вимог також передбачає проведення дослідження ринку, аналіз програмних продуктів-аналогів конкурентів та потреб користувачів, щоб отримати уявлення про переваги користувачів і проблемні моменти, які створити актуальність розробки нового програмного рішення.

Другим етапом є проектування системи, на якому розробники створюється високорівневий архітектурний дизайн програми на основі зібраних вимог. Це передбачає визначення загальної структури програми, включаючи модель даних, макет інтерфейсу користувача, процес навігації та інтеграцію із зовнішніми службами та API. Проектування системи також передбачає вибір відповідних технологій, фреймворків та інструментів розробки на основі вимог і обмежень проекту. Крім того, проектування системи може включати створення каркасів,

макетів і прототипів для візуалізації інтерфейсу користувача та досвіду користувача перед початком реалізації.

Третім етапом є етап впровадження. На цьому етапі проводиться написання коду для створення програми відповідно до специфікацій проекту. Це передбачає створення моделі даних за допомогою Core Data, реалізацію інтерфейсу користувача за допомогою SwiftUI, інтеграцію зовнішніх бібліотек і фреймворків, таких як Kingfisher для завантаження зображень, а також написання бізнес-логіки та функціональності за допомогою Swift. На етапі впровадження застосовуються такі практики програмування, як модульність, принципи DRY та контроль версій, щоб забезпечити якість коду, зручність подальшого обслуговування додатку, та його масштабованість.

Четвертим етапом розробки є етап тестування для виявлення та усунення будь-яких помилок або невідповідностей перед завантаженням програми у відкритий доступ. Тестування включає в себе різні методи, такі як модульне тестування, інтеграційне тестування та тестування прийнятності користувача, щоб перевірити функціональність програми, продуктивність і взаємодію з користувачем. Використовуються такі інструменти, як XCTest і вбудовані засоби налагодження Xcode, для автоматизації тестування, імітації взаємодії користувачів і вимірювання показників продуктивності. Крім того, може бути проведено бета-тестування за участю реальних користувачів, щоб зібрати відгуки та визначити області, які потрібно вдосконалити, перш ніж додаток буде випущено у відкритий доступ користування.

3.2 Використання засобів UML для моделювання застосунку

Використання інструментів UML (Unified Modeling Language) для моделювання додатків охоплює широкий спектр переваг, особливо в контексті розробки додатків для iOS. Інструменти UML пропонують стандартизований і комплексний спосіб візуалізації, проектування та передачі технічних аспектів своїх додатків, що забезпечує підвищену ясність і ефективність протягом усього процесу розробки. Основна перевага використання інструментів UML є їх здатність

надавати уніфіковану та стандартизовану нотацію для представлення різних аспектів архітектури програми, включаючи її структуру, поведінку та взаємодію. Інструменти UML пропонують високу універсальність і гнучкість у моделюванні різних аспектів програми, починаючи від високорівневих архітектурних діаграм до детальних діаграм класів і діаграм послідовності. Це дає змогу створити цілісне уявлення про дизайн і функціональність програми, дозволяючи виявляти потенційні недоліки конструкції, залежності та потенційно проблемні місця на ранніх етапах життєвого циклу розробки. Крім того, інструменти UML підтримують широкий спектр методів моделювання та діаграм, таких як діаграми варіантів використання, діаграми діяльності та діаграми станів, надаючи повний набір інструментів для вираження складних концепцій і сценаріїв.

Інструменти UML пропонують ряд аналітичних можливостей і можливостей перевірки, дозволяючи аналізувати та перевіряти правильність, повноту та узгодженість моделей додатків. Сюди входять такі функції, як автоматичні перевірки узгодженості, симуляція моделі та генерація коду, які допомагають гарантувати, що дизайн програми точно відображає заплановану поведінку та вимоги. Використовуючи ці аналітичні можливості та можливості перевірки, розробники можуть виявити потенційні проблеми та розбіжності на ранніх стадіях процесу розробки, зменшуючи ризик дорогих помилок і переробки пізніше.

Загалом, використання інструментів UML для моделювання додатків впроваджує велику кількість переваг для розробки додатків iOS, включаючи стандартизовану нотацію, ефективні методи моделювання, а також аналітичні можливості та можливості перевірки. Використання цих інструментів надає можливість створювати якісно розроблені, задокументовані та зрозумілі програми, які відповідають потребам користувачів і встановленим вимогам, що призводить до більш ефективного та успішного процесу розробки в цілому.

3.3 Діаграма варіантів використання

Діаграми варіантів використання забезпечують графічне представлення функціональних вимог системи, детально описуючи, як користувачі взаємодіють із системою для досягнення своїх цілей. Візуально відображаючи функціональні можливості системи в чіткій і стислій формі, діаграми варіантів використання дозволяють швидко зрозуміти масштаб і призначення програмної системи, сприяючи кращому розумінню вимоги до розроблюваного продукту.

До основних переваг створення діаграм варіантів використання входять:

1. Простота та інтуїтивність: діаграми варіантів використання пропонують простий та інтуїтивно зрозумілий спосіб представлення функціональних вимог системи, роблячи їх загально доступними для розуміння.

2. Ефективне спілкування та співпраця: діаграми варіантів використання служать інструментом візуального спілкування, що дозволяє всім, залученим до проекту сторонам співпрацювати над визначенням і вдосконаленням функціональності системи.

3. Виявлення та документування вимог: Діаграми варіантів використання забезпечують структурований підхід до виявлення та документування вимог до системи.

4. Ідентифікація та аналіз системних вимог: Діаграми варіантів використання полегшують ідентифікацію та аналіз системних вимог шляхом моделювання взаємодії між учасниками та варіантами використання програмного забезпечення.

5. Валідація та перевірка системних вимог: діаграми варіантів використання підтримують перевірку та перевірку системних вимог, надаючи візуальну структуру для оцінки повноти, послідовності та правильності функціональності системи.

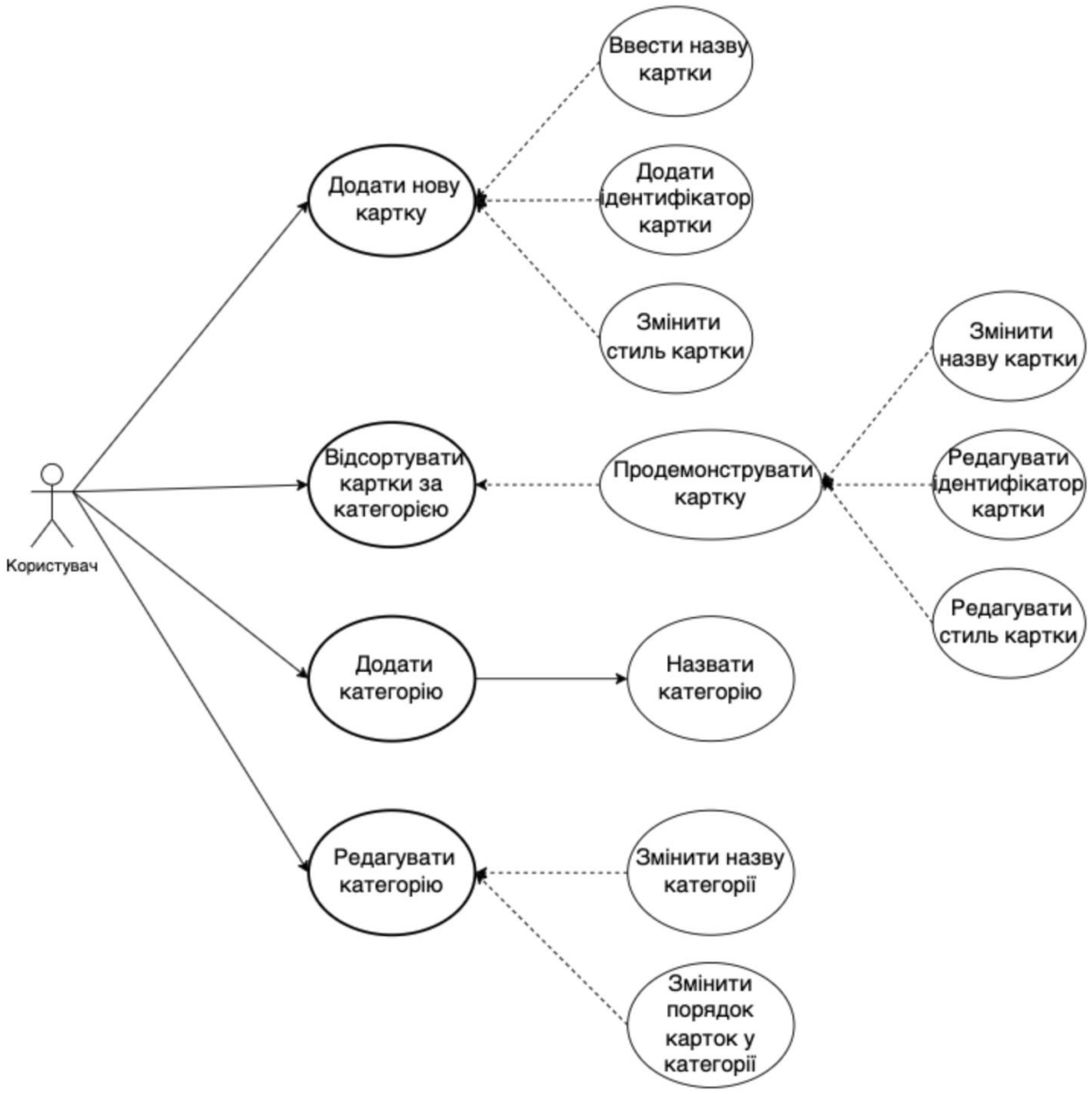


Рис. 3.1 Діаграма варіантів використання додатку U-Card

3.4 Схема бази даних

Моделювання схеми бази даних пропонує структурований підхід до представлення структури бази даних. Забезпечуючи ясність і структуру, схема баз даних дає змогу перевіряти вимоги, оптимізувати обслуговування та планувати масштабованість. Крім того, вона служить цінною документацією, допомагаючи в розумінні системи та майбутніх планів розвитку. Керуючись принципами ефективного моделювання схема бази даних спрощує представлення функціональних можливостей системи, сприяючи адаптації додатку для відповідності встановленим вимогам.

Серед основних переваг, які впроваджує коректно змодельована схема баз даних варто виділити наступні:

1. Чіткість і структура: Моделювання схеми бази даних забезпечує чітке та структуроване представлення структури бази даних, що полегшує розуміння загальної структури та роботу з базою даних.

2. Перевірка вимог: Моделювання дизайну схеми бази даних допомагає перевірити та уточнити вимоги до бази даних, візуалізуючи зв’язки між різними об’єктами даних і гарантуючи, що дизайн бази узгоджується із встановленими вимогами.

3. Технічне обслуговування та масштабованість: Коректно розроблена модель схеми бази даних дозволяє полегшити обслуговування та масштабованість системи бази даних з часом.

4. Документація: Моделі проектування схем бази даних служать цінними артефактами документації структури бази даних, обмеження та зв’язків. Ця документація допомагає розуміти систему, усувати несправності та впроваджувати подальший розвиток програмного продукту.

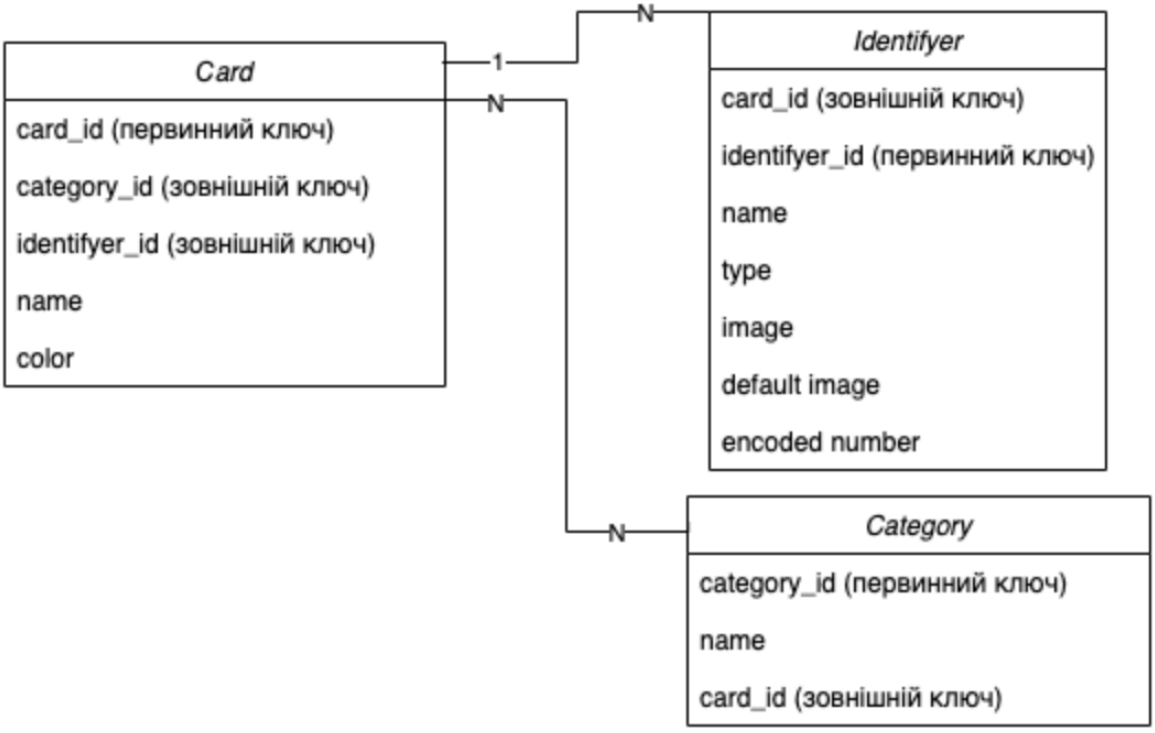


Рис. 3.2 Схема бази даних додатку U-Card

Для додатку було вирішено використати реляційну базу даних, із використанням реалізованих за допомогою “зовнішнього ключа” зв’язків. На схемі продемонстровано, що кожен з елементів має власний первинний, та зовнішні ключі за допомогою яких вони зв’язуються з іншими елементами. Елемент Card має зв’язок з елементом Identifyer один до багатьох, оскільки до одної картки може бути додано декілька ідентифікаторів, але кожен ідентифікатор може відповідати лише одній картці. Між елементами Card та Category встановлений зв’язок багато до багатьох, оскільки до категорії може бути додано багато карток, так само як і кожна картка може входити в декілька категорій.

3.5 Діаграма діяльності

Моделювання діаграми діяльності програми пропонує покращення візуалізації робочого процесу, розуміння поведінки системи, визначення можливостей оптимізації та підтвердження вимог до проекту.

Серед основних переваг коректно спроектованої діаграми діяльності варто виділити наступні:

1. Візуалізація робочого процесу: Діаграма діяльності забезпечує візуальне представлення процесу роботи програми, зображуючи послідовність дій і рішень, необхідних для виконання різних завдань або процесів.

2. Розуміння поведінки системи: моделювання дій та їхніх взаємозв’язків дозволяє отримати глибше розуміння поведінки системи, включно з тим, як користувачі взаємодіють із програмою та потоком даних і керуванням у системі.

3. Виявлення потенційно вразливих місць і можливостей оптимізації: Діаграми діяльності допомагають визначити потенційно вразливі місця та неефективність у робочому процесі програми, дозволяючи оптимізувати процес роботи програми для покращення продуктивності та взаємодії з користувачем.

4. Перевірка вимог: Діаграми діяльності служать інструментом перевірки системних вимог шляхом візуалізації очікуваної поведінки програми та забезпечення її відповідності очікуванням і потребам зацікавлених сторін.

На рисунку 3.3 наведена діаграма діяльності, розроблена відносно встановлених функціональних вимог до проекту.

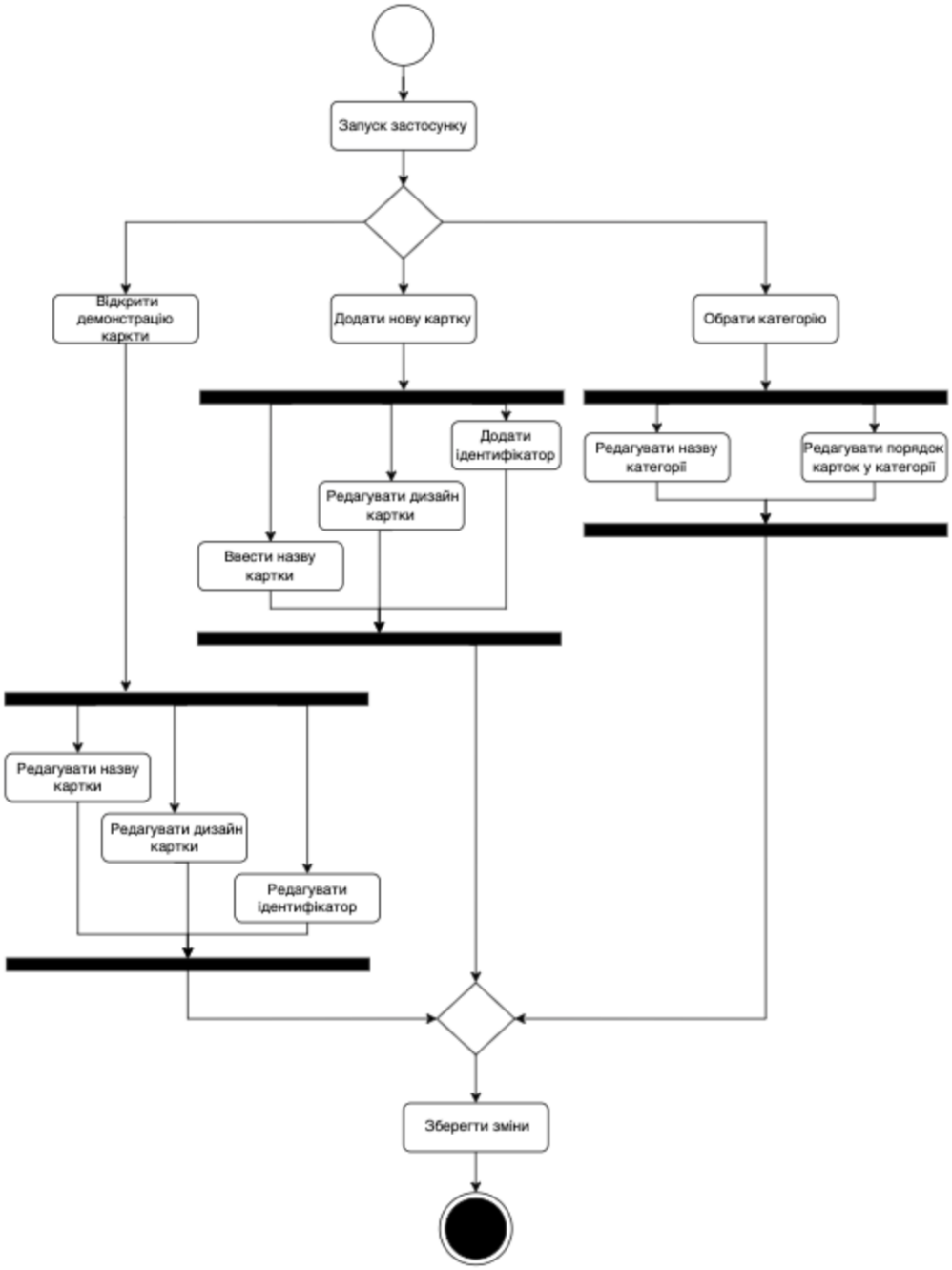


Рис. 3.3 Діаграма діяльності додатку U-Card

3.6 Діаграма класів

Розробка діаграми класів для програми пропонує можливість проводити структурне представлення, модульний дизайн, моделювання зв’язків, перевірку дизайну та генерацію коду/документацію. Використовуючи ці переваги, з’являється можливість створювати конкурентноспроможний, та добре структуровані, підтримувані і масштабовані програми, які відповідають встановленим потребам і очікуванням від функціоналу.

Серед основних переваг, що надає розробка діаграма класів, варто виділити наступні:

1. Структурне представлення: Діаграми класів надають структурне представлення архітектури програми, ілюструючи класи, атрибути, методи та зв’язки між об’єктами в системі. Це пропонує повний огляд структури програми, сприяючи кращому розумінню встановлених вимог.

2. Модульний дизайн: Візуалізуючи класи та їхні зв’язки, діаграми класів допомагають розбити програму на керовані модулі або компоненти. Цей модульний підхід до проектування сприяє дотриманню принципів DRY, зручності обслуговування та масштабованості, оскільки кожен модуль можна розробляти, тестувати та підтримувати незалежно.

3. Моделювання зв’язків: Діаграми класів дозволяють моделювати різні типи зв’язків між класами, наприклад асоціації, агрегації та успадкування. Це допомагає представити залежності та взаємодію між різними частинами системи, забезпечуючи надійність і гнучкість дизайну програми.

4. Перевірка дизайну: Діаграми класів служать інструментом для перевірки дизайну програми на відповідність вимогам і обмеженням. Візуалізувавши структуру класу та зв’язки, зацікавлені сторони можуть виявити потенційні недоліки дизайну, невідповідності або відсутні функції на ранніх етапах процесу розробки, мінімізуючи ризик дорогих помилок і переробки пізніше.

5. Генерація коду та документація: Діаграми класів можна використовувати для створення скелетів коду або документації для програми, прискорюючи процес реалізації та забезпечуючи узгодженість між проектом і реалізацією. Крім того, діаграми класів служать цінними артефактами документації, які документують структуру програми та проектні рішення, допомагаючи розуміти систему та підтримувати її.

На рисунку 3.4 наведена діаграма класів, розроблена відносно встановлених функціональних вимог до проекту.

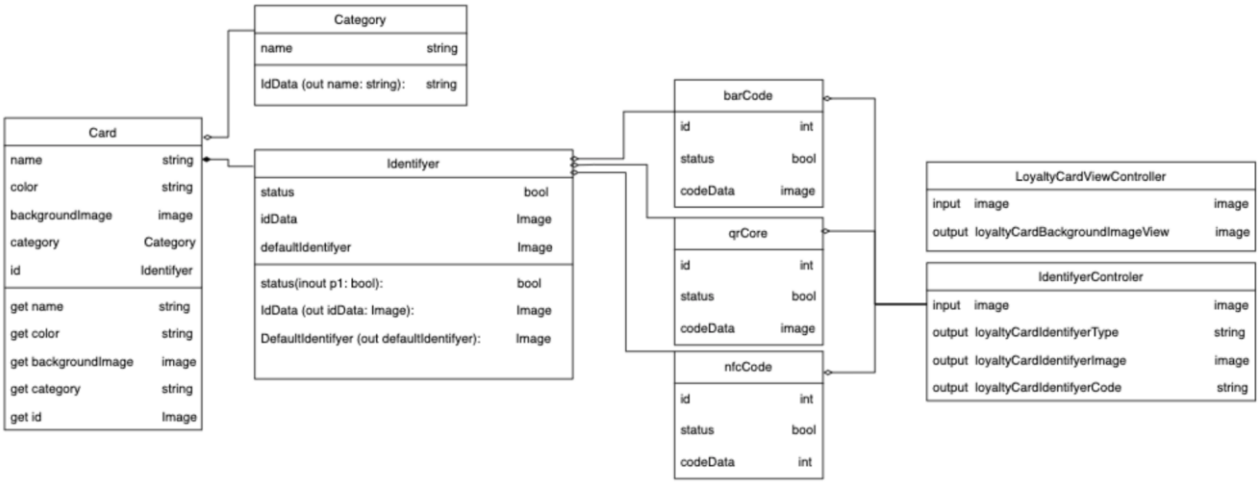


Рис. 3.4 Діаграма класів додатку U-Card

Починаючи з лівої сторони у діаграмі класів продемонстрований клас Card, Який відповідає за збір всіх параметрів картки для їх подальшого відображення, та параметр color, Котрий відображає колір заднього фону картки за замовчуванням, за умови що зображення заднього фону не було додано користувачем.

Правіше та вище, зі встановленим зв’язком агрегації, відображений клас категорії, котрий має параметр name, відповідаючи за назву категорії, до якої віднесено екземпляр класу Card. Зв’язок агрегації встановлений з причини того, що якщо картка не була додана користувачем до жодної із категорій вона все ще може відображатись у головні категорії за замовчуванням.

Нижче, з’єднаний із класом Card зв’язком композиції, відображений клас Identifyer, Що відповідає за тип та дані ідентифікатора, котрі передаються екземпляру класу Card. Параметр статус даного класу відповідає за наявність доданого користувачем ідентифікатора. За відсутності доданого користувачем ідентифікатора клас Identifyer повертає екземпляру класу Card параметр defaultIdentifyer із штрих кодом за замовчуванням для тимчасового заповнення місця для відсутнього користувацького ідентифікатора.

До класу Identifyer зв’язком агрегації приєднані три класи, що відповідають за дані окремих типів ідентифікаторів. Кожного із них присутні параметри id, що відповідає за тип конкретного ідентифікатора, status, що відповідає за наявність даного ідентифікатора, та codeData, Котрий відповідає безпосередньо за дані, що

мають бути передані до екземпляру класу Card.

З краю праворуч відображені класи-контролери LoyaltyCardViewController та IdentifyerController. Перший відповідає за коректне отримання та відображення користувацьких зображень для заднього фону карток, а другий за коректне отримання та обробку користувацьких ідентифікаторів карток лояльності.

3.7 Проектування інтерфейсу користувача для додатку з менеджменту персональних карток лояльності клієнта

Під час розробки інтерфейсу користувача iOS для програми, спрямованої на керування особистими картками постійного клієнта, був проведений аналіз інтерфейсів схожих за спрямуванням iOS додатків. Керуючись загальноприйнятими принципами проектування простих у сприйнятті інтерфейсів були зібрані загальні вимоги до розроблюваного інтерфейсу. До даних принципів належать принцип консистентності, принципи трьох кліків, принцип KISS, та урахування особливостей доступності додатку, для полегшення його використання людьми з обмеженими можливостями, та у сценаріях використання однією рукою.

Детальніше про виконані використані під час розробки додатку принципи: принцип консистентності заключається у дотриманні єдиного візуального стилю, та використання однакових навігаційних інструментів для виконання однакових дій. Принцип застосовується для полегшення розуміння інтерфейсу користувачем. Принцип трьох кліків заключається в розробці застосунку, де користувач зможе досягти будь-якої його функції всього за три натискання. Це сприяє простішому сприйняттю користувачем функціональних можливостей застосунку. Принцип KISS заключається в створенні візуально простих та мінімалістичнийх інтерфейсів, уникаючи черезмірних нагромаджень елементів на екрані. Дотримання даного методу дозволяє сконцентрувати увагу користувача на основних функціях додатку.

Дотримання встановлених вимог дозволило забезпечити розробку інтуїтивно зрозуміло, та привабливого дизайну. Спершу були створені каркас та макети, щоб окреслити загальний вигляд та потік програми. Цей крок включав огляд програми

аналогів, аналіз їх інтерфейсів, та проектування прототипів власних екранів з розрахунком на спрощення сприйняття функціональної частини користувачем, визначення ключових функцій і шляхів навігації. Сеанси тестування придатності до використання були включені, щоб удосконалити початкові проекти та вирішити будь-які проблеми інтуїтивності використання.

Після того, як каркаси були завершені, загальний вигляд базових функціїй був протестований за допомогою інструменту прототипування Figma. Це передбачало створення наближених до бажаного результату макетів кожного екрана, що включало елементи брендування, колірні схеми та типографіку, що узгоджувалися із загальною мовою дизайну програми. Було приділено увагу візуальній ієрархії, забезпечивши, щоб такі важливі елементи, як інформація про картку лояльності та елементи керування навігацією, були помітно відображені та легкодоступні. Протягом цього процесу дотримання Рекомендацій щодо людського інтерфейсу iOS було найважливішим для забезпечення узгодженості зі стандартами дизайну платформи та надання користувачам звичного та інтуїтивно зрозумілого досвіду.

Після етапу проектування елементи інтерфейсу було реалізовано за допомогою Swift і SwiftUI в Xcode, офіційному IDE від Apple для розробки додатків iOS. Декларативний синтаксис SwiftUI дозволив створити динамічні та адаптивні інтерфейси користувача, забезпечуючи інтеграцію складних компонентів інтерфейсу користувача, таких як списки, форми та елементи керування навігацією. Функції доступності також були враховані при розробці дизайну, щоб гарантувати, що програму зможуть використовувати всі користувачі, незалежно від їхніх здібностей або ситуативних обставин використання додатку.

На рисунках 3.5 – 3.9 зображені екранні форми додатку, створені в Xcode IDE та запущені на вбудованому у середу розробки симуляторі iPhone.

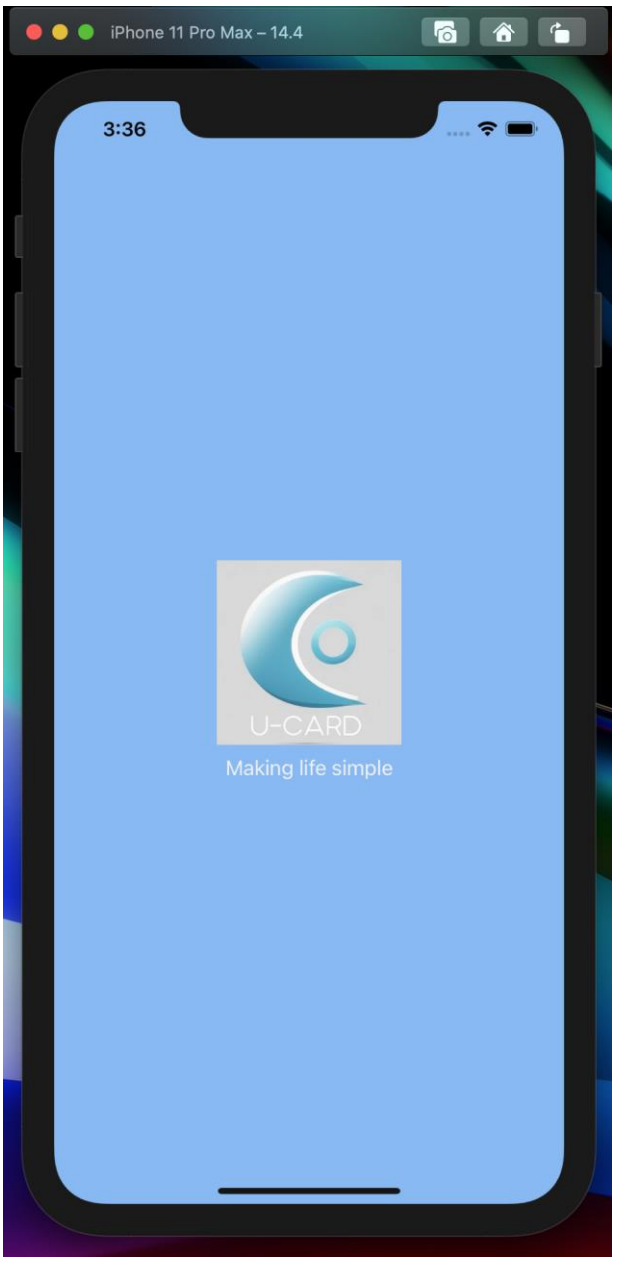


Рис. 3.5 Екран запуску додатку U-Card

На рисунку 3.5 зображено екран запуску, котрий відображається при завантаженні додатку. Даний екран демонструє логотип додатку, для забезпечення користувача комфортним відчуттям під час запуску додатку замість банальних анімацій завантаження.

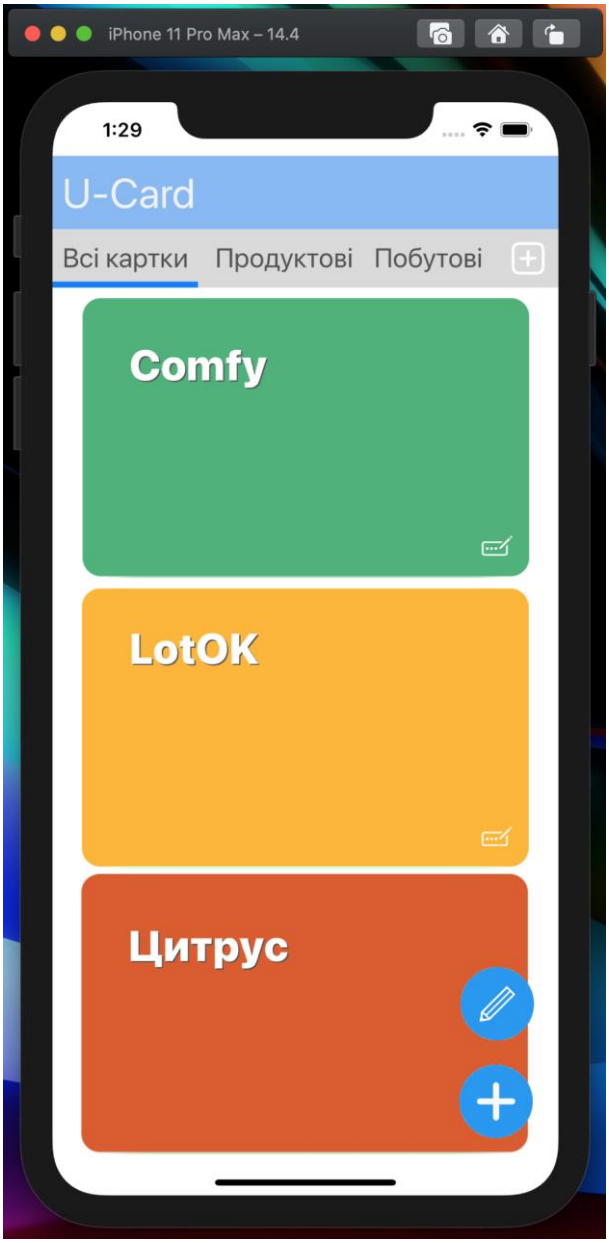


Рис. 3.6 Головний екран додатку U-Card

На рисунку 3.6 зображено головний екран додатку із відображення всіх тестових карток лояльності, та основних функцій додатку. На цьому екрані можна побачити такі функції, як сортування карток по категоріям, додавання нової категорії, редагування конкретної картки, редагування категорії, та додавання нової картки лояльності клієнта.

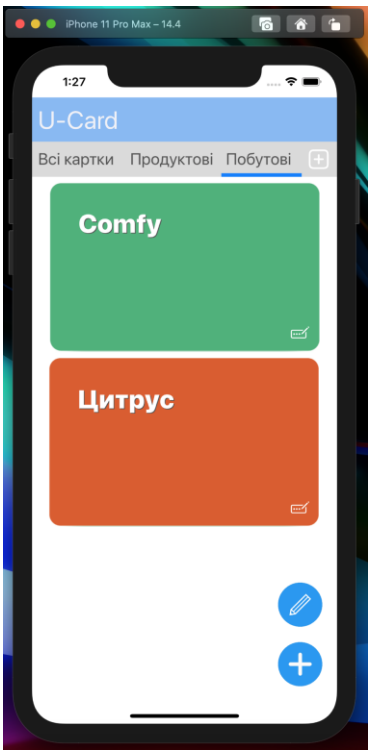


Рис. 3.7 Екран категорії карток

На рисунку 3.7 зображено екран із відсортованими картками за одною із категорій.

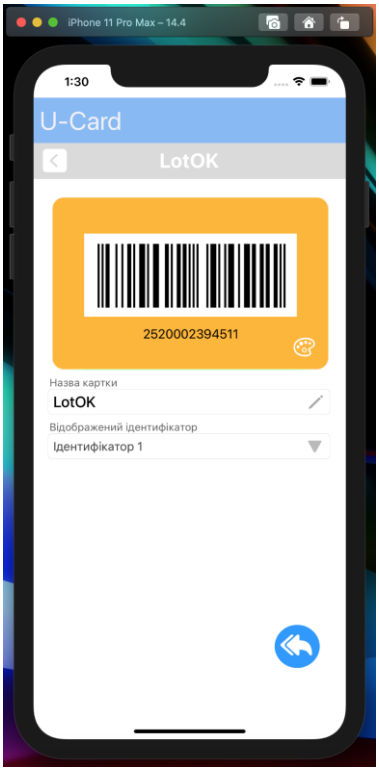


Рис. 3.8 Екран демонстрації картки

На рисунку 3.8 зображено екран демонстрації картки лояльності користувача. Даний кран призначений для демонстрації збереженої користувачем картки з метою її застосування. Окрім основної своєї функції екран також має опції по редагуванню назви картки, редагування зовнішнього вигляду картки, а також редагування, зміни, чи додавання нового ідентифікатора до поточної картки.

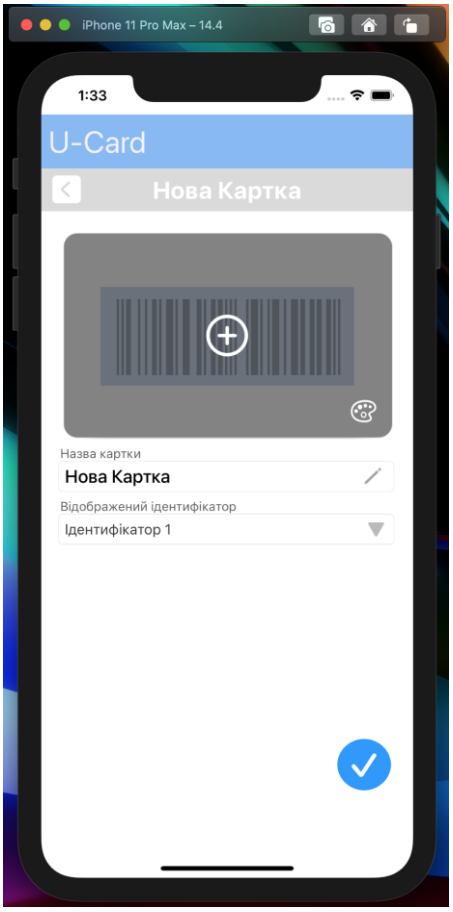


Рис. 3.9 Екран додавання картки

На рисунку 3.9 зображено екран додавання нової картки лояльності користувача. Користувач потрапляє на даний екран після натискання кнопки “+” на головному екрані, чи екрані категорії. Даний кран призначений для створення нової картки лояльності користувача. Екран надає можливість додати ідентифікатор до нової картки, редагувати зовнішній вигляд нової картки, ввести ім’я нової картки (або залишити ім’я за замовчуванням), редагувати назву поточного ідентифікатора, та зберегти введені дані натисканням кнопки “✓”.

Результатом створення інтерфейсу користувача програми для керування особистими картками постійного клієнта є інтуїтивний, зручний, створений із дотриманням встановлених принципів розробки інтерфейс, який ідеально поєднує функціональність з естетичною привабливістю. Завдяки мінімалістичномо дизайну та методичній розробці був створений інтерфейс, який відповідає потребам середньостатистичного користувача. Завдяки інтуїтивно зрозумілій навігації, чіткій візуалізація елементів керування та функціям доступності даний інтерфейс гарантує, що користувачі можуть легко керувати своїми картками лояльності, безперешкодно використовувати, та редагувати їх. Дотримуючись стандартів дизайну iOS і використовуючи найновіші технології для розробки, такі як SwiftUI, інтерфейс вийшов не тільки простим, естетичним, та інтуїтивно зрозумілим, а й швидким у роботі.

3.8 Тестування

Суть тестування полягає у виявленні та мінімізації ризиків, забезпеченні якості, надійності та зручності використання програми. За допомогою різних методів тестування, таких як модульне тестування, інтеграційне тестування та тестування прийнятності користувача, виявляється дефекти, помилки та невідповідності програмного забезпечення. Крім того, тестування допомагає підтвердити вимоги, покращити якість коду та оптимізувати продуктивність, що в кінцевому підсумку сприяє покращенню користувацького досвіду користування. Ретельне тестування програми на кожному етапі розробки, дозволяє впевнитись в коректності її роботи, та внести корективи у елементи, що функціонують не відповідним чином.

Для тестування розробленого програмного забезпечення додатку iOS з менеджменту персональних карток лояльності вирішено було використовувати мануальне тестування за наступними тест сценаріями:

Запуск додатку:

1. Перевірено коректність запуску додатку.
2. Підтверджено коректність відображення екрану запуску під час

завантаження програми.

3. Підтвердження відкриття програмою головної сторінки по завершенню процесу запуску.

Сортування карток за категоріями:

1. Перевірено коректність сортування карток за замовчуванням.
2. Перевірено коректність сортування карток за користувацькою категорією.
3. Перевірено можливість редагування поточної категорії.

Відображення картки:

1. Перевірено коректність відображення екрану демонстрації картки.
2. Перевірено коректність відображення ідентифікатора обраної картки.
3. Перевірена можливість редагування назви поточної картки.
4. Перевірена функція редагування ідентифікатора поточної картки.
5. Проведено тестування функції редагування зовнішнього вигляду картки.

Додавання нової картки:

1. Перевірено коректність відображення екрану додавання нової картки.
2. Проведено тестування функції з додавання нового ідентифікатора.
3. Перевірена функція збереження новоствореної картки.

Мануальне тестування було обрано, оскільки воно пропонує кілька переваг у процесі розробки програмного забезпечення. По-перше, це дозволяє тестувальнику особисто досліджувати функціональні можливості програми в динамічний та інтерактивний спосіб, виявляючи дефекти та невідповідності, які неможливо виявити лише за допомогою автоматизованого тестування. Ручне тестування також дає змогу тестувальнику на власному досвіді оцінити зручність використання, доступність і взаємодію інтерфейсу з користувачем від першого обличчя, формуючи більш конкретний чек-ліст необхідних до виправлення функцій. Крім того, ручне тестування можна легко адаптувати до змін у вимогах або функціях програми, що робить його гнучким і універсальним підходом до тестування.

ВИСНОВКИ

В результаті виконання дипломної роботи був розроблений iOS застосунок для зберігання карток лояльності клієнта мовою Swift. Отже можна стверджувати, що мета дипломної роботи була досягнута.

1. Виконо огляд предметної області та встановлена доцільність розробки власного програмного забезпечення
2. Виконано аналіз існуючих програмних рішень в сфері, таких як Stocard, Google wallet та інших. Виявлені їх особливості та недоліки, для врахування при розробці власного застосунку.
3. Виконано формулювання функціональних і нефункціональних вимог, основними з яких є можливість додавання, зберігання, та редагування даних карток користувача.
4. Виконано огляд доступних програмно-технічних засобів для створення iOS застосунку. Проект розроблено з використанням: Swift, SwiftUI, Core Data, Xcode.
5. Інструментом розробки обраний Xcode, як найбільш оптимізований для розробки iOS застосунків на інтегровані мові Swift.
6. Розроблений, та протестований застосунок для обліку клієнтських карток лояльності.
7. Робота пройшла апробацію на Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в ІКТ», 24 квітня 2024 р., Збірник тез. К.: ДУІКТ, 2024. С.420, та Всеукраїнській науково-практичній конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15 травня 2024 р. Збірник тез. К.: ДУІКТ, 2024. С.441.

ПЕРЕЛІК ПОСИЛАНЬ

1. Мова програмування Swift. Swift. URL: https://book.swift.org.ua/book/mova-programuvannya-swift/.

2. Apple. Wallet. Apple. URL: https://www.apple.com/wallet/.

3. CORP S. L. KEYRING: MY DIGITAL KEYS. App Store. URL: https://apps.apple.com/us/app/keyring-my-digital-keys/id1598834378?l=en.

4. GmbH S. Stocard - Loyalty Cards Wallet. Apple Store. URL: https://apps.apple.com/ua/app/stocard-loyalty-cards-wallet/id444578884?l=ua.

5. Google. Google Wallet: Carry more with Google Wallet. URL: https://wallet.google/intl/uk_ua/.

6. Layering content | Apple Developer Documentation. Apple Developer Documentation. URL: https://developer.apple.com/tutorials/swiftui-concepts/layering-content.

7. Ltd S. E. C. Samsung Wallet/Pay – Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=com.samsung.android.samsungpay.gear&hl=uk.

8. Morefield B., Team K. T., Galata I. SwiftUI by Tutorials: SwiftUI in Motion. Kodeco Inc., 2022.

9. Smyth N. iOS 17 App Development Essentials : навчальний посібник. 2024.

10. Swift Knowledge Base. Hacking with Swift. URL: https://www.hackingwithswift.com/example-code.

11. Swift Overview. W3Schools Tutorials | Learn HTML, CSS, JavaScript, Swift and More. URL: https://www.w3schools.in/swift/overview.

12. SwiftUI Overview - Xcode - Apple Developer. Apple Developer. URL: https://developer.apple.com/xcode/swiftui/.

13. Qualitative research method: How to Choose and Apply Your Marketing Research Technique - FasterCapital. FasterCapital. URL: https://fastercapital.com/content/ Qualitative-research-method--How-to-Choose-and-Apply-Your-Marketing-Research-Technique.html

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА IOS ЗАСТОСУНКУ ДЛЯ ЗБЕРІГАННЯ КАРТОК ЛОЯЛЬНОСТІ "U-CARD" МОВОЮ SWIFT

Виконав студент 4 курсу
групи ПД-44
Жилінський Дмитро Ілліч
Керівник роботи
д.т.н., доц., зав. кафедри ТЦР ДУІКТ, м. Київ, Жебка Вікторія Вікторівна

Київ – 2024

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи**: Полегшення процесу організації та використання персональних карток лояльності для клієнтів, підняття рівня залученості клієнтів до програм лояльності, та їх переваг за допомогою програмного забезпечення з менеджменту клієнтських карток лояльності
- **Об’єкт дослідження:** Процес організації, зберігання та використання персональних карток лояльності клієнта.
- **Предмет дослідження:** програмне забезпечення для організації, зберігання та використання особистих даних програм лояльності клієнта

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. *Виконати огляд предметної області*
2. *Виконати аналіз існуючих програмних рішень в сфері управління особистими картками програм лояльності*
3. *Провести формулювання функціональних і нефункціональних вимог до розробки програми зберігання карт лояльності*
4. *Виконати огляд доступних програмно-технічних засобів для створення додатків по зберігання карток лояльності*
5. *Обрати інструмент розробки, що надає змогу реалізувати всі необхідні функції програмного забезпечення*
6. *Розробити програмне забезпечення відповідно до встановлених вимог*

3

АНАЛІЗ АНАЛОГІВ

Розглянуті додатки	Stocard	Key Ring	Google pay	Apple Wallet	Samsung Pay	U-Card
Оптимізований для простоти сприйняття інтерфейс	—	+	—	—	+	+
Оптимізація швидкодії для девайсів Apple	—	+	—	+	—	+
Можливість персоналізації карток	+	+	—	+	+	+
Можливість персоналізації категорій карток	+	+	—	—	+	+
Адаптація під український ринок споживачів	+	—	+	—	—	+
Актуальна технічна підтримка застосунку	+	+	+	+	—	+
Відсутність нав'язливої реклами	—	—	+	+	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Демонстрація інформації збережених карток для їх застосування.
2. Можливість організації карток, та їх списків користувачем.
3. Додавання карток із будь-яким типом ідентифікатора.
4. Оптимізація інтерфейсу для забезпечення доступності, та можливості використання додатку однією рукою
5. Доступ до збережених карток навіть за умов відсутності підключення до мережі

Нефункціональні вимоги:

1. Оптимізована для девайсів на операційній системі iOS швидкість роботи додатку.
2. Сумісність з усією лінійкою підтримуваних девайсів Apple iPhone на операційній системі iOS.
3. Оптимізація користувацького інтерфейсу задля забезпечення інтуїтивності його використання

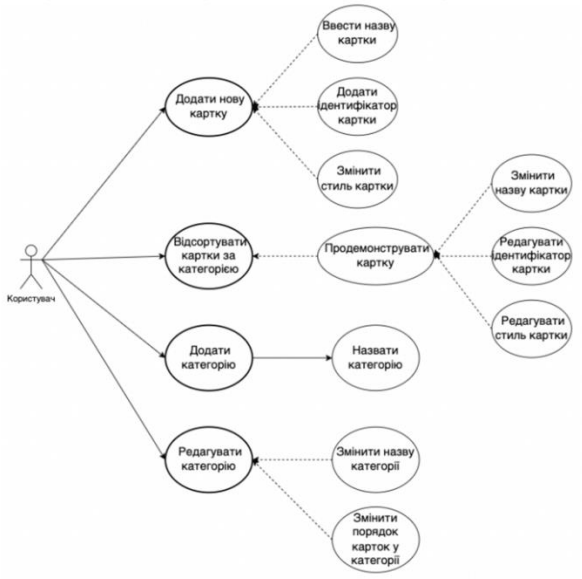
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Swift | Swift UI | XCode | Core Data | AVFoundation

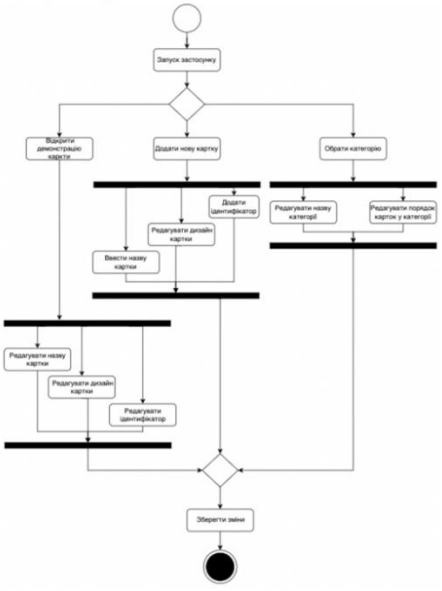
6

Діаграма варіантів використання



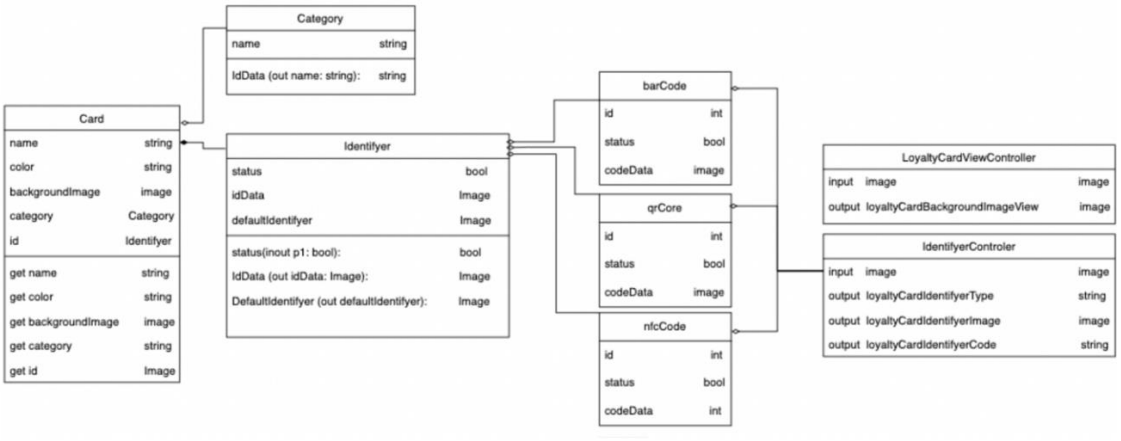
7

Діаграма діяльності



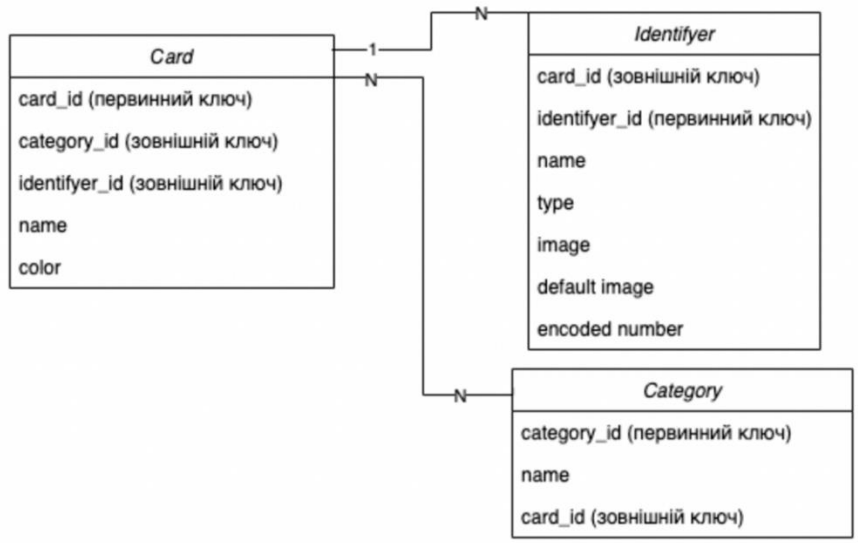
8

Діаграма класів



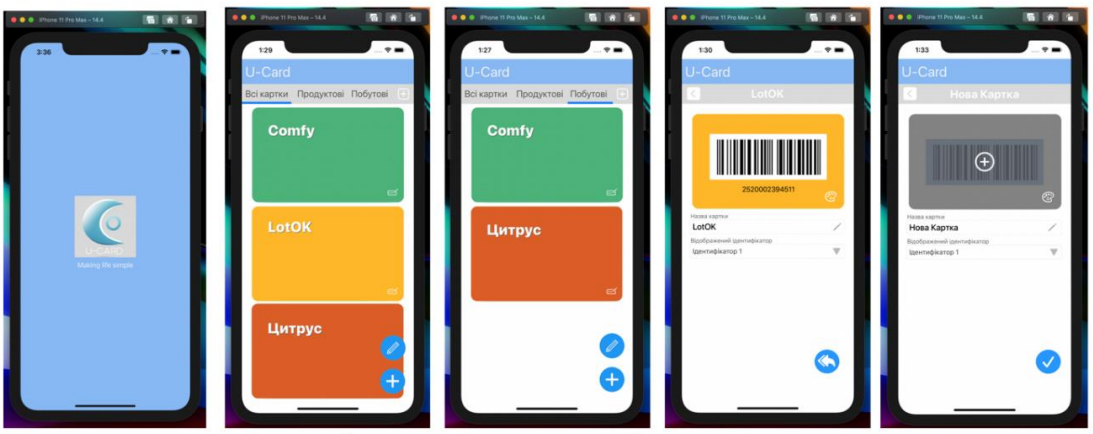
9

Схема бази даних



10

ЕКРАННІ ФОРМИ



Екран запуску | Головний екран (Список карт за замовчуванням) | Екран категорії карток | Екран демонстрації картки | Екран додавання картки

11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Жилінський Д.І., Жебка В.В., Розробка iOS застосунку для зберігання карток лояльності "U card" мовою swift/ Д.І. Жилінський // Застосування програмного забезпечення в ІКТ: Матеріали Всеукраїнської науково-технічної конференції, 24 квітня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024. С.420.
2. Жилінський Д.І., Жебка В.В., Розгляд аналогів програмного забезпечення для зберігання карток лояльності клієнта/ Д.І. Жилінський // Сучасні інтелектуальні інформаційні технології в науці та освіті: Матеріали Всеукраїнської науково-практичної конференції, 15 травня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024. С. 441.

12

ВИСНОВКИ

1. Виконано огляд предметної області та встановлена доцільність розробки власного програмного забезпечення.
2. Проведено аналіз існуючих програмних рішень в сфері, таких як Stocard, Google wallet та інших. Виявлені їх особливості та недоліки, для врахування при розробці власного застосунку.
3. Зформульовані функціональні і нефункціональні вимоги, основними з яких є можливість додавання, зберігання, та редагування даних карток користувача.
4. Виконано огляд доступних програмно-технічних засобів для створення iOS застосунку. Проект розроблено з використанням: Swift, SwiftUI, Core Data, Xcode.
5. Інструментом розробки обраний Xcode, як найбільш оптимізований для розробки iOS застосунків на інтегровані мові Swift.
6. Розроблений та протестований застосунок для обліку клієнтських карток лояльності.

13

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

MainViewController.swift

```
import UIKit
import SnapKit

class MainViewController: UIViewController {

    // MARK: - Properties
    var categories: [Category] = [Category(name: "All")]
    var selectedCategory: Category? {
        didSet {
            cardsCollectionView.reloadData()
        }
    }

    // UI Elements
    let categoriesPad: UIView = {
        let view = UIView()
        view.backgroundColor = .lightGray
        return view
    }()

    let categoriesCollectionView: UICollectionView = {
        let layout = UICollectionViewFlowLayout()
        layout.scrollDirection = .horizontal
        let collectionView = UICollectionView(frame: .zero, collectionViewLayout:
layout)
        collectionView.backgroundColor = .lightGray
        collectionView.showsHorizontalScrollIndicator = false
        return collectionView
    }()

    let cardsCollectionView: UICollectionView = {
        let layout = UICollectionViewFlowLayout()
        layout.scrollDirection = .vertical
        let collectionView = UICollectionView(frame: .zero, collectionViewLayout:
layout)
        collectionView.backgroundColor = .white
        return collectionView
    }()

    let addCategoryButton: UIButton = {
```

```
    let button = UIButton(type: .system)
    button.setImage(UIImage(systemName: "plus.app"), for: .normal)
    button.tintColor = .white
    return button
}()

let addCardButton: UIButton = {
    let button = UIButton(type: .system)
    button.setImage(UIImage(systemName: "plus"), for: .normal)
    button.backgroundColor = .blue
    button.tintColor = .white

    return button
}()

// MARK: - Lifecycle
override func viewDidLoad() {
    super.viewDidLoad()
    setupUI()
    setupCategories()
    selectedCategory = nil
}

// MARK: - Setup UI
private func setupUI() {
    view.backgroundColor = .white

    // Title View
    let titleView = UIView()
    titleView.backgroundColor = .blue
    let titleLabel = UILabel()
    titleLabel.text = "U-Card"
    titleLabel.textColor = .white
    titleLabel.textAlignment = .center
    titleLabel.font = UIFont.boldSystemFont(ofSize: 24)

    view.addSubview(titleView)
    titleView.addSubview(titleLabel)
    view.addSubview(categoriesPad)
    categoriesPad.addSubview(categoriesCollectionView)
    view.addSubview(addCategoryButton)
    view.addSubview(cardsCollectionView)
    view.addSubview(addCardButton)

    titleView.snp.makeConstraints { make in
```

```
    make.top.equalTo(view.safeAreaLayoutGuide.snp.top)
    make.leading.trailing.equalToSuperview()
    make.height.equalTo(60)
}

titleLabel.snp.makeConstraints { make in
    make.edges.equalToSuperview()
}

addCategoryButton.snp.makeConstraints { make in
    make.trailing.equalToSuperview().inset(10)
    make.centerY.equalTo(categoriesCollectionView)
    make.size.equalTo(40)
}

categoriesPad.snp.makeConstraints { make in
    make.trailing.equalTo(addCategoryButton.snp.trailing).offset(20)
    make.leading.equalToSuperview()
    make.top.equalTo(titleView.snp.bottom).offset(10)
    make.height.equalTo(50)
}

categoriesCollectionView.snp.makeConstraints { make in
    make.trailing.equalTo(addCategoryButton.snp.leading)
    make.leading.equalToSuperview().offset(20)
    make.top.equalTo(titleView.snp.bottom).offset(10)
    make.height.equalTo(50)
}

cardsCollectionView.snp.makeConstraints { make in
    make.leading.trailing.equalToSuperview()
    make.top.equalTo(categoriesCollectionView.snp.bottom).offset(10)
    make.bottom.equalToSuperview()
}

addCardButton.snp.makeConstraints { make in
    make.trailing.bottom.equalTo(view.safeAreaLayoutGuide).inset(20)
    make.size.equalTo(60)
}

addCardButton.layer.cornerRadius = 30

// Setup Collection Views

categoriesCollectionView.dataSource = self
```

```
        categoriesCollectionView.delegate = self

        categoriesCollectionView.register(CategoryCell.self, forCellWithReuseIdentifier:
CategoryCell.reuseIdentifier)

        cardsCollectionView.dataSource = self
        cardsCollectionView.delegate = self
        cardsCollectionView.register(CardCell.self, forCellWithReuseIdentifier:
CardCell.reuseIdentifier)

        // Add Targets
        addCategoryButton.addTarget(self, action: #selector(addCategoryButtonTapped),
for: .touchUpInside)
        addCardButton.addTarget(self, action: #selector(addCardButtonTapped), for:
.touchUpInside)
    }

    private func setupCategories() {
        categories = [Category(name: "All")]
        categories.append(Category(name: "Продуктовые"))
        categories.append(Category(name: "Бытовые"))
    }


    // MARK: - Actions
    @objc private func addCategoryButtonTapped() {

        let alert = UIAlertController(title: "New Category", message: "Enter category
name", preferredStyle: .alert)

        alert.addTextField { textField in
            textField.placeholder = "Category Name"
        }

        alert.addAction(UIAlertAction(title: "Add", style: .default, handler: { [weak self] _
in
            if let name = alert.textFields?.first?.text, !name.isEmpty {
                let newCategory = Category(name: name)
                self?.categories.append(newCategory)
                self?.categoriesCollectionView.reloadData()
                if self?.selectedCategory == nil {
                    self?.selectedCategory = newCategory
                }
            }
        }))
```

```
        alert.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
        present(alert, animated: true, completion: nil)
    }

    @objc private func addCardButtonTapped() {
        let editCardVC = EditCardViewController()
        editCardVC.selectedCategory = selectedCategory
        editCardVC.delegate = self
        editCardVC.categories = categories.filter { $0.name != "All" }
        navigationController?.pushViewController(editCardVC, animated: true)
    }

}

// MARK: - UICollectionViewDataSource, UICollectionViewDelegate

extension MainViewController: UICollectionViewDataSource,
UICollectionViewDelegate, UICollectionViewDelegateFlowLayout {

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
section: Int) -> Int {
        if collectionView == categoriesCollectionView {
            return categories.count
        } else {
            if let selectedCategory = selectedCategory {
                  return selectedCategory.cards.count
            }else {
               return categories.flatMap { $0.cards }.count

            }
        }
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
IndexPath) -> UICollectionViewCell {

        if collectionView == categoriesCollectionView {
            let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
CategoryCell.reuseIdentifier, for: indexPath) as! CategoryCell

            let category = categories[indexPath.row]
            let isSelected = category == selectedCategory
            cell.configure(with: category, isSelected: isSelected)
            return cell
```

```
        } else {
            let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
CardCell.reuseIdentifier, for: indexPath) as! CardCell
            if let selectedCategory = selectedCategory {
                let card = selectedCategory.cards[indexPath.row]
                cell.configure(with: card)
            } else {
                let allCards = categories.flatMap { $0.cards }
                let card = allCards[indexPath.row]
                cell.configure(with: card)
            }
            return cell
        }
    }

    func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath)
-> CGSize {

        if collectionView == categoriesCollectionView {
            let category = categories[indexPath.item]
            let labelWidth = category.name.size(withAttributes: [
                NSAttributedString.Key.font: UIFont.systemFont(ofSize: 17)
            ]).width

            return CGSize(width: labelWidth, height: collectionView.bounds.height)
        } else {
            let collectionViewWidth = collectionView.frame.width
            let cellWidth = collectionViewWidth - 20
            return CGSize(width: cellWidth, height: 200)
        }
    }

    func collectionView(_ collectionView: UICollectionView, didSelectItemAt
indexPath: IndexPath) {

        if collectionView == categoriesCollectionView {
            let category = categories[indexPath.row]
            selectedCategory = category.name == "All" ? nil : category
            cardsCollectionView.reloadData()
        } else {
            let editCardVC = EditCardViewController()
            if let selectedCategory = selectedCategory {
                editCardVC.card = selectedCategory.cards[indexPath.row]
            } else {
```

```
            // Если категория не выбрана, берем карту из общего списка
            let allCards = categories.flatMap { $0.cards }
            editCardVC.card = allCards[indexPath.row]
        }
        editCardVC.selectedCategory = selectedCategory
        editCardVC.delegate = self
        editCardVC.categories = categories
        editCardVC.categories.removeFirst()
        navigationController?.pushViewController(editCardVC, animated: true)
    }
  }
}

// MARK: - EditCardViewControllerDelegate
extension MainViewController: EditCardViewControllerDelegate {
  func didSaveCard() {
    cardsCollectionView.reloadData()
  }
}
```

AppDelegate.swift

```
import UIKit
import CoreData

@main
class AppDelegate: UIResponder, UIApplicationDelegate {


  var window: UIWindow?

  func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    window = UIWindow(frame: UIScreen.main.bounds)
    window?.rootViewController = UINavigationController(rootViewController:
MainViewController())
    window?.makeKeyAndVisible()
    return true
  }

  // MARK: UISceneSession Lifecycle

  func application(_ application: UIApplication, configurationForConnecting
connectingSceneSession: UISceneSession, options: UIScene.ConnectionOptions) ->
UISceneConfiguration {
```

```
        // Called when a new scene session is being created.
        // Use this method to select a configuration to create the new scene with.
        return UISceneConfiguration(name: "Default Configuration", sessionRole:
connectingSceneSession.role)
    }

    func application(_ application: UIApplication, didDiscardSceneSessions sceneSessions:
Set<UISceneSession>) {
        // Called when the user discards a scene session.
        // If any sessions were discarded while the application was not running, this will be
called shortly after application:didFinishLaunchingWithOptions.
        // Use this method to release any resources that were specific to the discarded scenes,
as they will not return.
    }

    // MARK: - Core Data stack

    lazy var persistentContainer: NSPersistentContainer = {
        /*
         The persistent container for the application. This implementation
         creates and returns a container, having loaded the store for the
         application to it. This property is optional since there are legitimate
         error conditions that could cause the creation of the store to fail.
        */
        let container = NSPersistentContainer(name: "UCard")
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {
                // Replace this implementation with code to handle the error appropriately.
                // fatalError() causes the application to generate a crash log and terminate. You
should not use this function in a shipping application, although it may be useful during
development.

                /*
                 Typical reasons for an error here include:
                 * The parent directory does not exist, cannot be created, or disallows writing.
                 * The persistent store is not accessible, due to permissions or data protection
when the device is locked.
                 * The device is out of space.
                 * The store could not be migrated to the current model version.
                 Check the error message to determine what the actual problem was.
                 */
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
        return container
```

```
    }()

    // MARK: - Core Data Saving support

    func saveContext () {
        let context = persistentContainer.viewContext
        if context.hasChanges {
            do {
                try context.save()
            } catch {
                // Replace this implementation with code to handle the error appropriately.
                // fatalError() causes the application to generate a crash log and terminate. You
should not use this function in a shipping application, although it may be useful during
development.
                let nserror = error as NSError
                fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
            }
        }
    }

}
```

SceneDelegate.swift

```
import UIKit
import CoreData

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    lazy var persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "UCard")
        container.loadPersistentStores { storeDescription, error in
            if let error = error as NSError? {
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        }
        return container
    }()

    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
        guard let windowScene = (scene as? UIWindowScene) else { return }
        window = UIWindow(windowScene: windowScene)
```

```
        let mainViewController = MainViewController()
        window?.rootViewController = UINavigationController(rootViewController:
mainViewController)
        window?.makeKeyAndVisible()
    }

    func sceneDidDisconnect(_ scene: UIScene) {
        // Called as the scene is being released by the system.
        // This occurs shortly after the scene enters the background, or when its session is
discarded.
        // Release any resources associated with this scene that can be re-created the next time
the scene connects.
        // The scene may re-connect later, as its session was not necessarily discarded (see
`application:didDiscardSceneSessions` instead).
    }

    func sceneDidBecomeActive(_ scene: UIScene) {
        // Called when the scene has moved from an inactive state to an active state.
        // Use this method to restart any tasks that were paused (or not yet started) when the
scene was inactive.
    }

    func sceneWillResignActive(_ scene: UIScene) {
        // Called when the scene will move from an active state to an inactive state.
        // This may occur due to temporary interruptions (ex. an incoming phone call).
    }

    func sceneWillEnterForeground(_ scene: UIScene) {
        // Called as the scene transitions from the background to the foreground.
        // Use this method to undo the changes made on entering the background.
    }

    func sceneDidEnterBackground(_ scene: UIScene) {
        // Called as the scene transitions from the foreground to the background.
        // Use this method to save data, release shared resources, and store enough scene-
specific state information
        // to restore the scene back to its current state.

        // Save changes in the application's managed object context when the application
transitions to the background.
        (UIApplication.shared.delegate as? AppDelegate)?.saveContext()
    }


}   }
```

CardCell.swift

```
import UIKit

class CardCell: UICollectionViewCell {
    static let reuseIdentifier = "CardCell"

    let nameLabel: UILabel = {
        let label = UILabel()
        label.font = UIFont.systemFont(ofSize: 35, weight: .heavy)
        label.textColor = .white
        return label
    }()

    let barcodeLabel: UILabel = {
        let label = UILabel()
        label.font = UIFont.systemFont(ofSize: 16)
        label.textColor = .gray
        label.isHidden = true
        return label
    }()

    let editButton: UIButton = {
        let button = UIButton(type: .system)
        button.setImage(UIImage(systemName: "pencil.and.ellipsis.rectangle"), for:
.normal)
        button.tintColor = .white
        return button
    }()

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupUI()
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func setupUI() {
        contentView.addSubview(nameLabel)
        contentView.addSubview(barcodeLabel)
        contentView.addSubview(editButton)
```

```
        nameLabel.snp.makeConstraints { make in
            make.top.leading.equalToSuperview().inset(30)
        }

        barcodeLabel.snp.makeConstraints { make in
            make.top.equalTo(nameLabel.snp.bottom).offset(5)
            make.leading.equalToSuperview().inset(10)
        }

        editButton.snp.makeConstraints { make in
            make.trailing.bottom.equalToSuperview().inset(10)
            make.size.equalTo(30)
        }
    }

    func configure(with card: Card) {
        nameLabel.text = card.name
        barcodeLabel.text = card.barcode
        contentView.backgroundColor = card.color
        contentView.layer.cornerRadius = 16
    }
}
```

CategoryCell.swift

```
import UIKit

class CategoryCell: UICollectionViewCell {
    static let reuseIdentifier = "CategoryCell"

    private let selectionView: UIView = {
        let view = UIView()
        view.backgroundColor = .blue
        view.isHidden = true
        return view
    }()

    private let nameLabel: UILabel = {
        let label = UILabel()
        label.textAlignment = .center
        return label
    }()

    override init(frame: CGRect) {
        super.init(frame: frame)
```

```
        contentView.addSubview(nameLabel)
        contentView.addSubview(selectionView)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func layoutSubviews() {
        super.layoutSubviews()
        nameLabel.frame = contentView.bounds
        selectionView.frame = CGRect(x: 0, y: contentView.bounds.maxY - 4, width:
contentView.bounds.width, height: 4)
        selectionView.isHidden = !isSelected

    }

    func configure(with category: Category, isSelected: Bool) {
        nameLabel.text = category.name
        selectionView.isHidden = !isSelected
    }
}
```

BarcodeId.swift

```
import Foundation

struct BarcodeId {
    let name: String
    let id: String
}
```

Card.swift

```
import UIKit

class Card {
    var name: String
    var barcode: String
    var color: UIColor
    var category: Category
    var selectedId: BarcodeId
    var ids: [BarcodeId]

    init(name: String, barcode: String, color: UIColor, category: Category, ids:
```

```
[BarcodeId], selectedId: BarcodeId) {
        self.name = name
        self.barcode = barcode
        self.color = color
        self.category = category
        self.ids = ids
        self.selectedId = selectedId
    }
}
```

Category.swift

```
import UIKit

class Category {
    var name: String
    var cards: [Card]

    init(name: String, cards: [Card] = []) {
        self.name = name
        self.cards = cards
    }
}

extension Category: Equatable {
    static func == (lhs: Category, rhs: Category) -> Bool {
        return lhs.name == rhs.name
    }
}
```

EditCardViewController.swift

```
import UIKit
import SnapKit

protocol EditCardViewControllerDelegate: AnyObject {
    func didSaveCard()
}

class EditCardViewController: UIViewController, UIPickerViewDataSource,
UIPickerViewDelegate {

    var card: Card?
    var selectedCategory: Category?
    weak var delegate: EditCardViewControllerDelegate?
    var categories: [Category] = []
```

```
var ids: [BarcodeId] = []

let nameLabel: UILabel = {
    let label = UILabel()
    label.text = "Назва картки"
    label.font = .systemFont(ofSize: 14)
    return label
}()

let idLabel: UILabel = {
    let label = UILabel()
    label.text = "Відображений ідентифікатор"
    label.textAlignment = .center
    label.font = .systemFont(ofSize: 14)
    return label
}()

let categoryLabel: UILabel = {
    let label = UILabel()
    label.text = "Категорія"
    label.textAlignment = .center
    label.font = .systemFont(ofSize: 14)
    return label
}()

let cardView: UIView = {
    let view = UIView()
    view.backgroundColor = .red
    view.layer.cornerRadius = 16
    return view
}()

let barcodeImageView: UIImageView = {
    let imageView = UIImageView()
    imageView.backgroundColor = .clear
    return imageView
}()

let nameTextField: UITextField = {
    let textField = UITextField()
    textField.placeholder = "Card Name"
    textField.borderStyle = .roundedRect
    return textField
}()
```

```
let barcodeTextField: UITextField = {
    let textField = UITextField()
    textField.placeholder = "Barcode"
    textField.borderStyle = .roundedRect
    textField.textAlignment = .center
    textField.backgroundColor = .clear
    return textField
}()

let colorButton: UIButton = {
    let button = UIButton(type: .system)
    let image = UIImage(systemName: "paintpalette")
    button.setBackgroundImage(image, for: .normal)
    button.tintColor = .white
    return button
}()

let saveButton: UIButton = {
    let button = UIButton(type: .system)
    let image = UIImage(systemName: "checkmark")
    button.setImage(image, for: .normal)
    button.tintColor = .white
    button.backgroundColor = .blue
    button.layer.cornerRadius = 30
    return button
}()

let idPickerView: UIPickerView = {
    let pickerView = UIPickerView()
    return pickerView
}()

let newIdButton: UIButton = {
    let button = UIButton()
    let image = UIImage(systemName: "plus.app")
    button.setBackgroundImage(image, for: .normal)
    button.tintColor = .gray
    return button
}()

let categoryPickerView: UIPickerView = {
    let pickerView = UIPickerView()
    return pickerView
}()
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        setupUI()
        loadData()

        categoryPickerView.dataSource = self
        categoryPickerView.delegate = self

        idPickerView.dataSource = self
        idPickerView.delegate = self

        // Автоматически выбрать категорию, если она сохранена
        if let selectedCategory = card?.category,
           let categoryIndex = categories.firstIndex(where: { $0.name ==
selectedCategory.name }) {
            categoryPickerView.selectRow(categoryIndex, inComponent: 0, animated:
false)
        }

        // Автоматически выбрать ID, если он сохранен
        if let card = card,
           !card.ids.isEmpty {
            let selectedId = card.selectedId
            let idIndex = ids.firstIndex(where: { $0.id == selectedId.id })!
            idPickerView.selectRow(idIndex, inComponent: 0, animated: false)
            barcodeTextField.text = selectedId.id
            barcodeImageView.image = generateBarcode(from: selectedId.id)
        }
    }


    private func setupUI() {
        view.backgroundColor = .white

        view.addSubview(cardView)
        cardView.addSubview(barcodeImageView)
        view.addSubview(barcodeTextField)
        view.addSubview(colorButton)
        view.addSubview(nameLabel)
        view.addSubview(nameTextField)
        view.addSubview(idLabel)
        view.addSubview(idPickerView)
        view.addSubview(newIdButton)
        view.addSubview(categoryLabel)
        view.addSubview(categoryPickerView)
```

```
view.addSubview(saveButton)

cardView.snp.makeConstraints { make in
    make.top.equalTo(view.safeAreaLayoutGuide.snp.top).offset(20)
    make.leading.trailing.equalToSuperview().inset(20)
    make.height.equalTo(200)
}

barcodeImageView.snp.makeConstraints { make in
    make.top.leading.trailing.equalToSuperview().inset(40)
    make.bottom.equalToSuperview().inset(65)
}

barcodeTextField.snp.makeConstraints { make in
    make.centerX.equalTo(cardView)
    make.bottom.equalTo(cardView).inset(20)
    make.leading.trailing.equalTo(cardView).inset(70)
}

colorButton.snp.makeConstraints { make in
    make.trailing.bottom.equalTo(cardView).inset(20)
    make.size.equalTo(32)
}

nameLabel.snp.makeConstraints { make in
    make.top.equalTo(cardView.snp.bottom).offset(20)
    make.leading.trailing.equalToSuperview().inset(20)
}

nameTextField.snp.makeConstraints { make in
    make.top.equalTo(nameLabel.snp.bottom).offset(6)
    make.leading.trailing.equalToSuperview().inset(20)
}

idLabel.snp.makeConstraints { make in
    make.top.equalTo(nameTextField.snp.bottom).offset(20)
    make.leading.equalToSuperview().inset(20)
}

newIdButton.snp.makeConstraints { make in
    make.centerY.equalTo(idLabel)
    make.trailing.equalToSuperview().inset(20)
}

idPickerView.snp.makeConstraints { make in
```

```
            make.top.equalTo(idLabel.snp.bottom).offset(6)
            make.leading.trailing.equalToSuperview().inset(20)
            make.height.equalTo(100)
        }

        categoryLabel.snp.makeConstraints { make in
            make.top.equalTo(idPickerView.snp.bottom).offset(6)
            make.leading.trailing.equalToSuperview().inset(20)
        }

        categoryPickerView.snp.makeConstraints { make in
            make.top.equalTo(categoryLabel.snp.bottom).offset(20)
            make.leading.trailing.equalToSuperview().inset(20)
            make.height.equalTo(100)
        }

        saveButton.snp.makeConstraints { make in
            make.top.equalTo(categoryPickerView.snp.bottom).offset(20)
            make.trailing.equalToSuperview().inset(20)
            make.size.equalTo(60)
        }

        colorButton.addTarget(self, action: #selector(colorButtonTapped), for:
.touchUpInside)
        saveButton.addTarget(self, action: #selector(saveButtonTapped), for:
.touchUpInside)
        newIdButton.addTarget(self, action: #selector(newIdButtonTapped), for:
.touchUpInside)
    }

    private func loadData() {
        if let card = card {
            nameTextField.text = card.name
            barcodeTextField.text = card.barcode
            cardView.backgroundColor = card.color

            // Если у карты есть идентификаторы, отобразим их
            if !card.ids.isEmpty {
                let selectedId = card.ids[0] // Возьмем первый идентификатор для примера
                self.ids = card.ids
                barcodeTextField.text = selectedId.id
                barcodeImageView.image = generateBarcode(from: selectedId.id)
            }
        }
    }
```

```
    @objc private func colorButtonTapped() {
        let alert = UIAlertController(title: "Select Color", message: nil, preferredStyle:
.alert)
        alert.addTextField { textField in
            textField.placeholder = "#FFFFFF"
        }
        alert.addAction(UIAlertAction(title: "OK", style: .default, handler: { [unowned
self] _ in
            if let colorHex = alert.textFields?.first?.text {
                self.cardView.backgroundColor = UIColor(hexString: colorHex)
            }
        }))
        alert.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
        present(alert, animated: true, completion: nil)
    }


    @objc private func saveButtonTapped() {
        // Add logging to see the values before the guard statement
        print("Attempting to save card with details:")
        print("Name: \(String(describing: nameTextField.text))")
        print("Barcode: \(String(describing: barcodeTextField.text))")
        print("Color: \(String(describing: colorButton.backgroundColor))")
        print("Selected Category: \(String(describing: selectedCategory?.name))")

        guard let name = nameTextField.text, !name.isEmpty else {
            print("Name is empty")
            showAlert(message: "Name is required")
            return
        }

        guard let barcode = barcodeTextField.text, !barcode.isEmpty else {
            print("Barcode is empty")
            showAlert(message: "Barcode is required")
            return
        }

        guard let color = cardView.backgroundColor else {
            print("Color is nil")
            showAlert(message: "Color is required")
            return
        }
```

```
        guard let selectedCategory = selectedCategory else {
            print("Selected category is nil")
            showAlert(message: "Category is required")
            return
        }

        // Logging for debugging
        print("Saving card with details:")
        print("Name: \(name)")
        print("Barcode: \(barcode)")
        print("Color: \(color.toHex())")
        print("Category: \(selectedCategory.name)")
        print("IDs: \(ids.map { $0.id }.joined(separator: ", "))")

        if let card = card {
            card.name = name
            card.barcode = barcode
            card.color = color
            card.category = selectedCategory
            card.ids = ids
        } else {
            let newCard = Card(name: name, barcode: barcode, color: color, category:
selectedCategory, ids: ids, selectedId: .init(name: "Default", id: barcode))
            selectedCategory.cards.append(newCard)
        }

        delegate?.didSaveCard()
        navigationController?.popViewController(animated: true)
    }

    // Helper method to show alerts
    private func showAlert(message: String) {
        let alert = UIAlertController(title: "Error", message: message, preferredStyle:
.alert)
        alert.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
        present(alert, animated: true, completion: nil)
    }

    @objc private func newIdButtonTapped() {
        let alert = UIAlertController(title: "New Barcode ID", message: "Enter a name and
ID", preferredStyle: .alert)
        alert.addTextField { textField in
            textField.placeholder = "Name"
        }
        alert.addTextField { textField in
```

```
            textField.placeholder = "ID"
        }
        alert.addAction(UIAlertAction(title: "Add", style: .default, handler: { [unowned
self] _ in
            if let name = alert.textFields?[0].text, !name.isEmpty,
               let id = alert.textFields?[1].text, !id.isEmpty {
                let newBarcodeId = BarcodeId(name: name, id: id)
                self.ids.append(newBarcodeId)
                self.idPickerView.reloadAllComponents()
            }
        }))
        alert.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
        present(alert, animated: true, completion: nil)
    }

    // MARK: - Barcode Generation
    private func generateBarcode(from string: String) -> UIImage? {
        let data = string.data(using: String.Encoding.ascii)
        if let filter = CIFilter(name: "CICode128BarcodeGenerator") {
            filter.setValue(data, forKey: "inputMessage")
            if let output = filter.outputImage {
                let scaleX = barcodeImageView.frame.size.width / output.extent.size.width
                let scaleY = barcodeImageView.frame.size.height / output.extent.size.height
                let transformedImage = output.transformed(by: CGAffineTransform(scaleX:
scaleX, y: scaleY))
                return UIImage(ciImage: transformedImage)
            }
        }
        return nil
    }
    // MARK: - UIPickerViewDataSource & UIPickerViewDelegate
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 1
    }
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if pickerView == categoryPickerView {
            return categories.count
        } else if pickerView == idPickerView {
            return ids.count
        }
        return 0
    }

    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent
```

```
component: Int) -> String? {
        if pickerView == categoryPickerView {
            return categories[row].name
        } else if pickerView == idPickerView {
            return ids[row].name
        }
        return nil
    }
    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
component: Int) {
        if pickerView == categoryPickerView {
            selectedCategory = categories[row]
            // Logging for debugging
            print("Selected category: \(categories[row].name)")
        } else if pickerView == idPickerView {
            let selectedId = ids[row]
            barcodeTextField.text = selectedId.id
            DispatchQueue.main.async {
                self.barcodeImageView.image = self.generateBarcode(from: selectedId.id)
            }
        }
    }
}
// MARK: - UIColor Extension for Hex
extension UIColor {
    convenience init(hexString: String) {
        let scanner = Scanner(string: hexString)
        scanner.scanLocation = 1
        var hexNumber: UInt64 = 0
        scanner.scanHexInt64(&hexNumber)
        let r = CGFloat((hexNumber & 0xff0000) >> 16) / 255
        let g = CGFloat((hexNumber & 0x00ff00) >> 8) / 255
        let b = CGFloat(hexNumber & 0x0000ff) / 255
        self.init(red: r, green: g, blue: b, alpha: 1.0)
    }
    func toHex() -> String {
        guard let components = cgColor.components, components.count >= 3 else {
            return "#FFFFFF"
        }
        let red = Float(components[0])
        let green = Float(components[1])
        let blue = Float(components[2])
        return String(format: "#%02lX%02lX%02lX", lroundf(red * 255), lroundf(green *
255), lroundf(blue * 255))
    }  }
```