НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
«__» _____ 2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Інженерія програмного забезпечення комп'ютерних систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри»

Виконав:
студент 4 курсу, групи ІП-04
Пащенко Дмитро Олексійович _____

Керівник:
асистент Ковальчук Олександр Миронович _____

Консультант (нормоконтроль):
доцент Волокита Артем Миколайович _____

Рецензент:
доцент кафедри ІСТ Шимкович Володимир Миколайович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)
Освітньо-професійна програма
«Інженерія програмного забезпечення комп'ютерних систем»
спеціальності 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
«__» _____ 2024 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента
Пащенка Дмитра Олексійовича

1. Тема проєкту: «Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри»,
керівник проєкту: ас. Ковальчук Олександр Миронович,
затверджений наказом по університету від 27 травня 2024 р. №2112-с.

2. Термін здачі студентом закінченого проєкту: 06 червня 2024 р.

3. Вихідні дані до проєкту: мова програмування Swift, інтегроване середовище розробки Xcode, технологія інтерфейсу користувача UIKit, платформа для розробки iOS.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються): огляд наявних рішень, огляд технологій в розробці системи, опис розробки застосунку, методика роботи розробленого застосунку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень): алгоритм взаємодії адміністратора з системою (принципова схема), діаграма класів (функціональна схема), архітектурна структура мобільного застосунку (структурна схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання: 05 січня 2024 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	Затвердження теми проєкту	19.02.2024-10.03.2024	
2.	Вивчення та аналіз завдання	11.03.2024-17.03.2024	
3.	Розробка архітектури та загальної структури системи	18.03.2024-24.03.2024	
4.	Розробка структур окремих підсистем	25.03.2024-31.03.2024	
5.	Програмна реалізація системи	01.04.2024-28.04.2024	
6.	Оформлення пояснювальної записки	29.04.2024-02.06.2024	
7.	Захист програмного продукту	05.06.2024	
8.	Передзахист	06.06.2024	
9.	Захист	20.06.2024	

Студент-дипломник Дмитро ПАЩЕНКО

Керівник проєкту Олександр КОВАЛЬЧУК

Анотація

В бакалаврському дипломному проєкті реалізовано мобільний застосунок для адміністрування системи для діагностики захворювань шкіри. За його допомогою адміністратор може надавати доступ до застосунку новим користувачам, здійснювати менеджмент діагностик, моделей та версій моделей.

Функціонал програми зосереджений у двох напрямах: менеджмент користувачів та менеджмент діагностик/моделей/версій моделей. Менеджмент користувачів здійснюється шляхом додавання нових. Менеджмент діагностик/моделей/версій моделей здійснюється шляхом додавання нових, видалення непотрібних та обрання активної моделі та поточної версії моделі.

Програмний продукт був створений за допомогою мови Swift в інтегрованому середовищі розробки Xcode. Для інтерфейсу користувача використовується технологія UIKit.

Ключові слова: мобільний застосунок, адміністрування, діагностика захворювань шкіри, iOS, Swift, UIKit, Xcode.

Annotation

In this project for a bachelor’s degree, a mobile application for the administration of the system for the diagnosis of skin diseases was implemented. With its help, the administrator can grant access to the application to new users, manage diagnostics, models, and model versions.

The functionality of the program is concentrated in two areas: user management and management of diagnostics/models/model versions. Users are managed by adding new ones. Diagnostics/models/model versions are managed by adding new ones, removing unnecessary ones, and selecting the active model and the current model version.

The software product was created using the Swift language in the Xcode integrated development environment. UIKit technology is used for the user interface.

Key words: mobile application, administration, diagnosis of skin diseases, iOS, Swift, UIKit, Xcode.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	ІАЛЦ.467200.002 ТЗ	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	3		
			Технічне завдання			
	А4	ІАЛЦ.467200.003 ПЗ	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	63		
			Пояснювальна записка			
	А4	ІАЛЦ.467200.004 Д1	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	1		
			Алгоритм взаємодії адміністратора з системою (принципова схема)			
	А4	ІАЛЦ.467200.005 Д2	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	1		
			Діаграма класів (функціональна схема)			
	А4	ІАЛЦ.467200.006 Д3	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	1		
			Архітектурна структура мобільного застосунку (структурна схема)			
	А4	ІАЛЦ.467200.007 Д4	Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри	21		
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА			
Зм	Лист	№ докум.	Підп.	Дата				
Розробив		Пащенко Д. О.			Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри Опис альбому			
Перевірив		Ковальчук О. М.						
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-04		
Н. Контр.		Волокита А. М.						
Затв.								

Технічне завдання
до дипломного проєкту
на тему: «Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри»

Київ – 2024 р.

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

Це технічне завдання поширюється на розробку мобільного застосунку для адміністрування системи для діагностики захворювань шкіри. Область застосування: допомога адміністратору в адмініструванні системи для діагностування шкірних захворювань за допомогою мобільного телефона.

2 ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського проєкту по освітньо-професійній програмі «Інженерія програмного забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розробка мобільного застосунку для адміністрування системи для діагностики захворювань шкіри, що поширить обізнаність про свої шкірні хвороби користувачів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література щодо використаних технологій, їх офіційна документація, а також статті та публікації в інтернеті.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Простий, зрозумілий інтерфейс мобільного застосунку.
- Дотримання Human Interface Guidelines.
- Надати можливість адміністраторам додавати нових користувачів, керувати набором діагностик, моделей, а також їхніми версіями.

- Надати можливість робити певну модель активною для діагностики, а також обирати поточну версію моделі.

5.2. Вимоги до програмного забезпечення

- Операційна система iOS 16 та вище, iPadOS 16 та вище.

5.3. Вимоги до апаратної частини

- Вільний простір жорсткого диска не менше 10 Мбайт.
- Підключення до інтернету.

6 ЕТАПИ РОЗРОБКИ

	Дата
6.1 Затвердження теми проєкту	19.02.2024-10.03.2024
6.2 Вивчення та аналіз літератури	11.03.2024-17.03.2024
6.3 Розробка загальної структури системи	18.03.2024-24.03.2024
6.4 Програмна розробка системи	01.04.2024-28.04.2024
6.5 Тестування системи та виправлення помилок	28.04.2024
6.6 Оформлення документації	29.04.2024-02.06.2024

Пояснювальна записка
до дипломного проєкту
на тему: «Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри»

Київ – 2024 р.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

OS (Operating System) – Операційна система

API (Application Programming Interface) – Інтерфейс програмування застосунків

JSON (JavaScript Object Notation) – Об'єктна нотація JavaScript

REST (Representational State Transfer) – Передача стану представлення

HTTP (HyperText Transfer Protocol) – Протокол передачі гіпертексту

XML (eXtensible Markup Language) – Розширювана мова розмітки

UI (User Interface) – Інтерфейс користувача

SDK (Software Development Kit) – Набір засобів розробки програмного забезпечення

IDE (Integrated Development Environment) – Інтегроване середовище розробки

ВСТУП

У сучасному світі мобільні технології не лише перетворилися на невіддільну складову нашого повсякденного життя, але й проникли в різноманітні сфери, включаючи медицину та охорону здоров'я. Швидкість та зручність, які притаманні мобільним застосункам, роблять їх ідеальними інструментами для інноваційних підходів до діагностики та лікування.

У контексті сучасних викликів, таких як епідемія коронавірусу чи повномасштабне вторгнення, потреба у доступних та ефективних засобах діагностики стає ще більш актуальною. Система охорони здоров'я стикається з необхідністю швидкого та точного виявлення захворювань, в тому числі шкірних, з метою оперативного лікування та запобігання подальшому поширенню.

На сьогодні Україна стикається зі складними викликами в контексті війни. В таких умовах доступ до медичної допомоги може бути обмеженим, що підкреслює значення розвитку інноваційних технологій, які забезпечують можливість дистанційного консультування та діагностики.

Мобільний застосунок для адміністрування системи діагностики захворювань шкіри, який розглядається в цьому дипломному проєкті, відповідає цим потребам, поєднуючи в собі передові технології зручного та швидкого доступу до медичної інформації з можливістю ефективного управління процесом діагностики за допомогою сучасних засобів обробки зображень і взаємодії з моделями штучного інтелекту.

Розробка такого застосунку не лише спростить процес адміністрування системи діагностики шкірних захворювань для медичних працівників, але й може мати значний соціальний вплив, забезпечуючи для користувачів доступну та швидку медичну допомогу.

Таким чином, цей дипломний проєкт націлений на створення інноваційного рішення, яке не лише відповідає сучасним вимогам медичної діагностики, але й може стати важливим кроком у забезпеченні доступної медичної допомоги, особливо в умовах воєнного стану.

1 ОГЛЯД НАЯВНИХ РІШЕНЬ

1.1 Визначення понять

На цей момент в системі охорони здоров'я по всьому світу зберігається тенденція до все частішого використання інноваційних технологій для допомоги в діагностуванні та лікуванні захворювань пацієнтів. Водночас також полегшується доступ до якісної та швидкої медицини шляхом використання нових рішень[1].

Одним з видів таких систем є таке програмне забезпечення, що дозволяє легко взаємодіяти пацієнту з системою охорони здоров'я його країни. Серед функціоналу таких систем можна виділити легку взаємодію з сімейним лікарем, можливість запису до нього та профільних лікарів, перегляд аналізів тощо. В кожній країні такі сервіси відрізняються за функціоналом, що зумовлено відмінностями в системах охорони здоров'я, а також різними культурами та звичками людей. В Україні серед відомих медичних систем, що мають значний вплив у сфері, є Helsi та Health24.

Іншим видом медичних систем є такі, що допомагають пацієнтам визначити, чи слід звертатись до лікаря з приводу певної потенційної хвороби. Такі застосунки можуть використовувати алгоритми та аналіз за допомогою систем штучного інтелекту для визначення ймовірності наявності того чи іншого захворювання. Після використання подібного сервісу користувач може отримати рекомендацію звернутись до лікаря у разі виявлення сервісом високої ймовірності недуги. Однак, варто зауважити, що часто моделі штучного інтелекту, що використовуються в таких системах, можуть приймати на вхід лише зображення. Тому визначення ймовірності присутності певного захворювання можливе лише на основі аналізу зображень, наприклад, шкірних патологій. Прикладом таких систем є Skin Issue Detector[2] та SkinScanner[3].

Для цієї роботи було обрано другий вид медичних систем, оскільки використання провідних технологій штучного інтелекту є більш перспективним напрямком, до того ж такі рішення можливо використовувати у всьому світі, тоді як загальні медичні системи складно адаптовувати під різні країни. Для розробки

обрано операційну систему iOS, оскільки смартфони, що працюють на цій ОС, є одними з найбільш популярними у світі[4].

1.2 Огляд наявних рішень

1.2.1 Skin Issue Detector

Skin Issue Detector – мобільний застосунок з відкритим вихідним кодом, що призначений для виявлення проблем шкіри. Програма використовує модель нейронної мережі для аналізу фотографії шкіри користувача, що він робить в застосунку або обирає з галереї. Після цього користувач може бачити, який недолік було виявлено, та отримує рекомендацію звернутись до дерматолога.

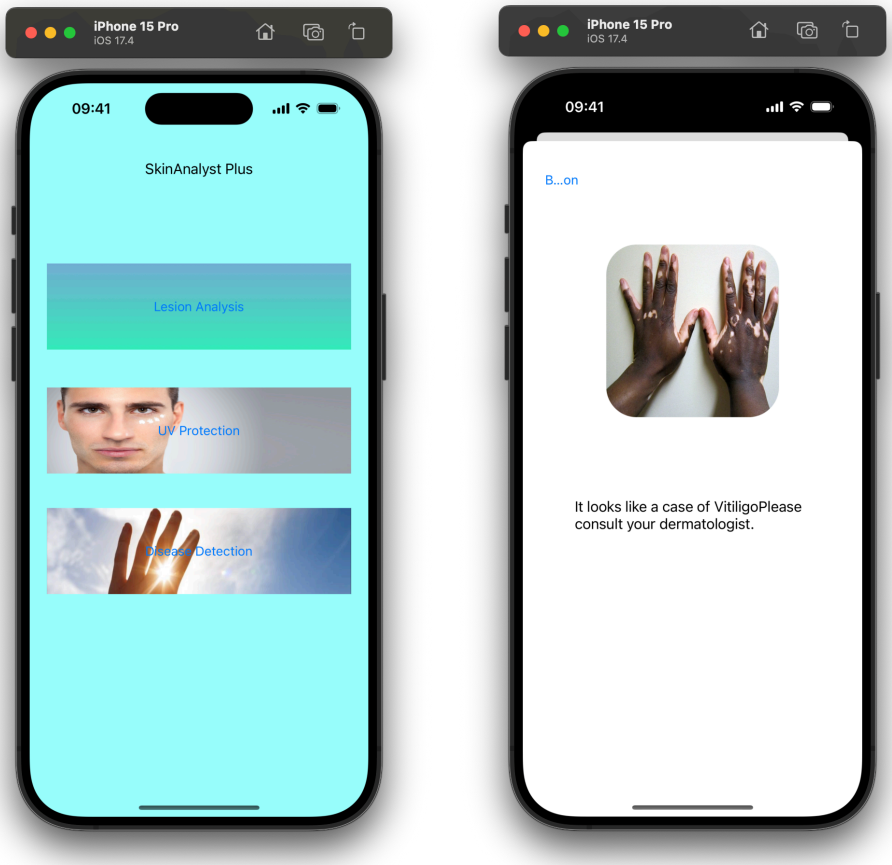


Рисунок 1.1 Застосунок Skin Issue Detector

Skin Issue Detector має ряд виражених переваг, серед яких:

1) Автономність: застосунок працює без інтернету, оскільки використовує локальну модель нейронної мережі.

2) Функціональність: можливість аналізу великої кількості шкірних захворювань, таких як:
    a. Звичайне акне/розацеа
    b. Вітиліго
    c. Псоріаз
    d. Екзема (на руках)
    e. Мозолі
    f. Кандидоз (грибкова інфекція)
3) Відкритість: застосунок має репозиторій на платформі GitHub, що дозволяє всім користувачам впевнитись в тому, як працює система.

Водночас система має деякі недоліки:

1) Неінтуїтивність: програма має складний інтерфейс, в якому складно розібратись новому користувачу.
2) Примітивність: в системі неможливо отримати повне розуміння стосовно наступних кроків, нема зв’язку з лікарем.
3) Застосунок не перекладено українською.

Загалом, застосунок Skin Issue Detector є хорошим рішенням для пацієнтів, що хотіли б перевірити дерматологічний недолік своєї шкіри. Однак, через неінтуїтивність інтерфейсу, примітивність застосунку та відсутність перекладу українською деякі користувачі можуть відмовитись від використання.

1.2.2 SkinScanner

SkinScanner – ще один застосунок з відкритим кодом, що покликаний розпізнавати дерматологічні хвороби на основі аналізу фотографії. Принцип роботи схожий на минулу програму: користувач обирає фотографію або робить новий знімок, завантажує його та отримує результат.

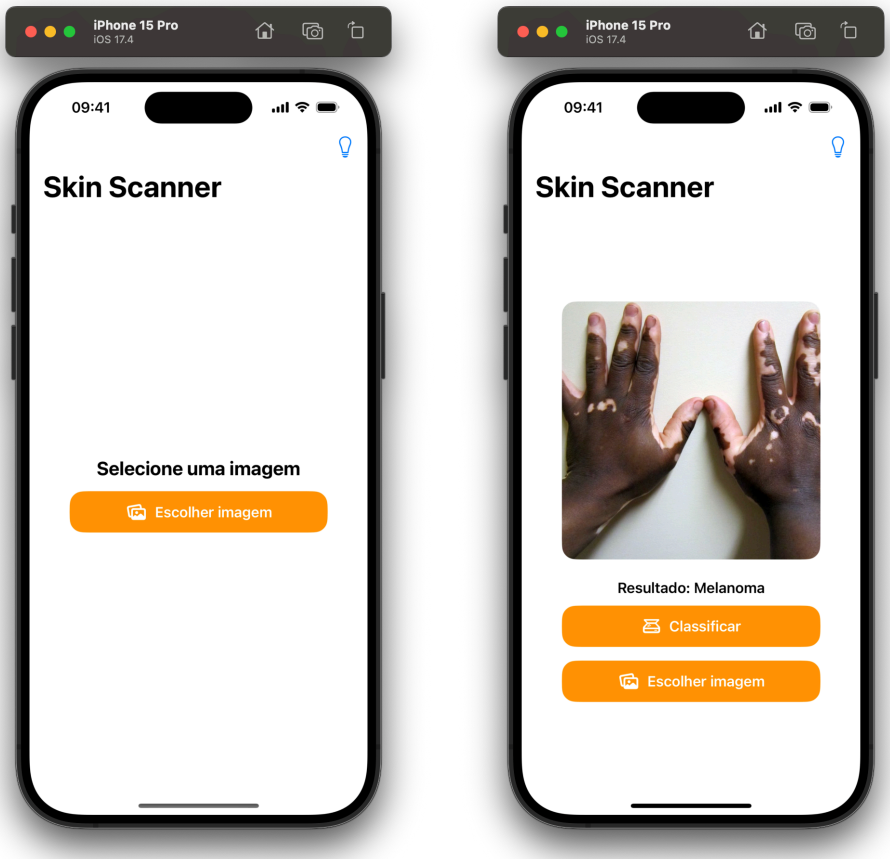


Рисунок 1.2 Застосунок SkinScanner

Серед позитивних моментів можна зазначити такі:

1) Робота офлайн: застосунок працює й без підключення до мережі.
2) Відкритий код: програма має репозиторій на GitHub, що дозволяє перевірити точність модель нейронної мережі.
3) Простота: користувачам легко розібратись з інтерфейсом через відсутність зайвого функціоналу.

Однак варто зауважити, що система також має кілька мінусів:

1) Мова застосунку: інтерфейс не перекладено англійською та іншими мовами.
2) Відсутність застережень: застосунок видає інформацію без вказання того, що можлива помилка. Також нема списку подальших дій, рекомендацій звернутись до лікаря.
3) Неточність: є вірогідність, що результат аналізу в застосунку буде некоректним.

Підсумовуючи, можна сказати, що SkinScanner може бути цікавим інструментом для людей, які хотіли б підтримувати свій стан здоров'я та слідкувати за хворобами шкіри, але мова застосунку та відсутність застережень можуть відштовхнути певну кількість користувачів.

1.2.3 AI Dermatologist

AI Dermatologist – це мобільний застосунок для виявлення проблем зі шкірою, який використовує штучний інтелект для аналізу фотографій. На відміну від Skin Issue Detector та SkinScanner, AI Dermatologist пропонує більш комплексний підхід до діагностики шкірних захворювань.

Переваги:

1) Комплексний підхід: AI Dermatologist пропонує більш комплексний підхід до діагностики шкірних захворювань, ніж інші мобільні застосунки, завдяки поєднанню аналізу фотографій, індивідуальних рекомендацій та можливості зв'язатися з дерматологом.
2) Персоналізовані рекомендації: застосунок надає персоналізовані рекомендації щодо лікування та догляду за шкірою, які ґрунтуються на результатах аналізу фотографій.
3) Зв'язок з дерматологом: користувачі можуть зв'язатися з ліцензованим дерматологом через застосунок для отримання додаткової консультації та діагностики.

Недоліки:

1) Вартість: AI Dermatologist пропонує безплатну пробну версію, але для повного доступу до всіх функцій програми потрібна підписка.
2) Доступність: Застосунок доступний лише англійською мовою, що ускладнює його використання людьми, що погано знають іноземні мови й говорять лише українською.

Отже, AI Dermatologist є потужним інструментом для людей, які хочуть краще зрозуміти стан своєї шкіри та отримати персоналізовані рекомендації щодо лікування. Однак вартість та доступність лише англійською мовою можуть бути обмежувальними факторами для деяких користувачів.

1.3 Порівняння наявних рішень

Створимо порівняльну таблицю.

Таблиця 1.1 Порівняння наявних рішень

Критерій	Skin Issue Detector	SkinScanner	AI Dermatologist
Безплатність	+	+	−
Українська мова	−	−	−
Відкритість коду	+	+	−
Зв’язок з лікарем	−	−	−
Інтуїтивність інтерфейсу	−	+	+

Оглядаючи таблицю, можна побачити декілька основних проблем, які мають наявні аналоги. Серед них – відсутність української мови та зв’язку з лікарем після отримання результатів обробки фотографій. Серед інших недоліків слід зауважити те, що деякі рішення платні та з закритим вихідним кодом. Також не менш важливою деталлю є інтерфейс, який має бути інтуїтивним та зрозумілим для користувача.

ВИСНОВОК ДО РОЗДІЛУ 1

При розробці системи для діагностики захворювань шкіри необхідно врахувати всі недоліки, що мають наявні рішення. Загалом, система має мати переклад українською мовою та відкритий вихідний код. Також, оскільки в цьому проєкті буде розроблено застосунок саме для адміністратора, необхідно надати йому широкі можливості для підтримки працездатності системи та забезпечення користувача необхідним функціоналом. Окремим пунктом слід винести інтуїтивність та зрозумілість інтерфейсу застосунку, оскільки не всі адміністратори мають великий досвід використання комп'ютерних технологій.

Окрім вищезазначеного, варто також зберегти можливість масштабування продукту та можливість розширення роботи на різних ринках. Наприклад, можна передбачити розширення кількості шкірних захворювань та роботу у різних географічних просторах, де людська шкіра може певні відмінності.

2 ОГЛЯД ТЕХНОЛОГІЙ В РОЗРОБЦІ СИСТЕМИ

2.1 Мови програмування

У наступних пунктах буде описано різні мови програмування, що використовуватимуться або можуть бути використаними при розробці мобільного застосунку, їхні позитивні та негативні моменти та обґрунтування щодо використання.

2.1.1 Swift

Swift – це мова програмування загального призначення, що легка для вивчення для новачків та має велику низку переваг та можливостей для експертів. Саме Swift є найпопулярнішою мовою для розробки застосунків під такі платформи компанії Apple, як iOS, watchOS, iPadOS, macOS та tvOS. І хоча в основному цю мову використовують для написання застосунків, вона також підтримує і серверну розробку. Swift є сучасною, безпечною, швидкою та потужною мовою, а також має відкритий вихідний код.

Рисунок 2.1 Swift

У порівнянні зі старішою мовою Objective-C, Swift має більш сучасний та зрозумілий синтаксис та є більш безпечною з погляду обробки помилок. Окрім цього, Swift зазвичай працює швидше, ніж Objective-C, оскільки Swift використовує новітній оптимізований компілятор та структури даних. Також Swift має ширший функціонал, серед якого, наприклад, вбудована підтримка функцій високого рівня, таких як замикання, кортежі, генератори тощо.

Отже, у підсумку, Swift має низку переваг:

1) Новітність: мова є новою та сучасною, що означає наявність більш зрозумілого та простого синтаксису. Це дозволяє більш швидко почати розробку застосунків.
2) Популярність: Swift є найпопулярнішим рішенням для написання програм під платформи Apple, а отже має велику спільноту розробників, які завжди допоможуть вирішити ту чи іншу складність при виконанні задач.
3) Безпечність: Swift має низку вбудованого функціонала, що покликаний збільшити безпеку коду, не дозволяючи виникати помилкам та крашам. Серед такого функціонала можна виокремити необхідність декларації змінних перед їхнім використанням, перевірку масивів та цілих чисел на переповнення та автоматичне керування пам’яттю. Також окремою можливістю Swift, що знижує кількість помилок, є опціональний тип даних.
4) Швидкість: як вже було зазначено вище, у порівнянні з аналогами, Swift є швидкою мовою програмування, випереджуючи своїх конкурентів у швидкості компіляції. Синтаксис та стандартна бібліотека були налаштовані таким чином, аби найочевидніший спосіб написання коду був водночас і найефективнішим, незалежно від того, чи він виконується на годиннику, на телефоні чи на кластері серверів.

Звичайно, Swift має і певні недоліки. Через те, що ця мова програмування є досить новою, кожна нова версія може суттєво змінити синтаксис, що може ускладнити роботу із застарілим кодом. Це також вимагає від розробника активно слідкувати за оновленнями. Окрім цього, нові версії Swift можуть мати обмежену підтримку старших версій операційних систем, під які розробляється певний застосунок.

2.1.2 Objective-C

Objective-C – це об'єктноорієнтована мова програмування загального призначення високого рівня, що була розроблена Apple у 1984 році[5].

Рисунок 2.2 Objective-C

Ця мова програмування є об'єктноорієнтованою, що означає те, що вона має всі властивості об'єктноорієнтованих мов програмування, серед яких підтримка класів, об’єктів, спадкування, поліморфізму, інкапсуляції. Довгий час саме Objective-C була основним рішенням для розробки застосунків під платформи Apple, однак із виходом Swift у 2014 році, популярність Objective-C падала з року в рік. На цей час, все ще залишається низка проєктів, написаних саме цією мовою програмування, однак їхня кількість поступово знижується через переваги Swift.

Недоліки Objective-C включають:

1) Складний синтаксис.
2) Повільність у порівнянні з Swift.
3) Непопулярність.

У підсумку можна сказати, що, хоча Objective-C і використовується в деяких проєктах, починати новий, використовуючи саме цю мову, є недоцільно.

2.1.3 Висновок щодо вибору мови програмування

Оскільки мова програмування Swift має низку переваг, а також не має тих недоліків, що має Objective-C, було вирішено використовувати саме цю мову програмування для написання коду.

Мову Swift було розроблено компанією Apple, та саме ця мова є рекомендованою для розробки нових застосунків під мобільні платформи

компанії. На цей час є безліч бібліотек, що ще більше полегшують розробку з використанням цього інструменту, а отже час, необхідний для написання програми, є значно меншим. Як вже було зазначено, серед переваг Swift – новітність, популярність, безпечність та швидкість.

Отже, хоча розробку мобільних застосунків під iOS можна виконувати на різних мовах програмування, саме мова Swift підходить до цього найкраще. Тому і було обрано безпосередньо цей інструмент.

2.2 Технології

Для написання мобільного застосунку для діагностики захворювань шкіри також необхідно використати певні технології, опис можливостей та розбір суті яких буде виконано у наступних пунктах.

2.2.1 UIKit

UIKit – це фреймворк, розроблений компанією Apple для написання користувацьких інтерфейсів для iOS, iPadOS, watchOS та tvOS. Фреймворк надає низку інструментів, необхідних для створення застосунків з інтуїтивно зрозумілим інтерфейсом.

Рисунок 2.3 UIKit

Фреймворк забезпечує архітектуру вікон і переглядів для реалізації інтерфейсу користувача, інфраструктуру обробки подій для забезпечення Multi-Touch та інших типів введення у програмі, а також основний цикл виконання для керування взаємодією між користувачем, системою та програмою[6].

UIKit також включає підтримку анімацій, документів, малювання та друку, керування текстом і його відображення, пошук, розширення програм, керування ресурсами та отримання інформації про поточний пристрій. Також підтримуються спеціальні можливості для користувачів із додатковими потребами та локалізація інтерфейсу програми для різних мов, країн або культурних регіонів.

Отже, у підсумку, UIKit має такий набір переваг:

1) Стабільність: фреймворк вже достатньо довго залишається актуальним та підтримуваним, що додає стабільності.
2) Широка підтримка: фреймворк працює на багатьох версіях iOS, включно зі старшими.
3) Популярність: як вже було зазначено, UIKit вже достатньо довго використовується розробниками, а тому має велику спільноту.

Однак, попри переваги, також є й недоліки:

1) Верстка в коді: написання інтерфейсу вимагає багато часу та коду, що може бути складно та менш наочно.

2.2.2 SwiftUI

SwiftUI – це сучасний фреймворк від Apple для розробки інтерфейсів користувача, який використовує декларативний підхід до створення UI. Основна відмінність від UIKit полягає в тому, що UIKit використовує імперативний підхід, де розробники явно описують послідовність кроків для побудови інтерфейсу, тоді як SwiftUI дозволяє описувати, як повинен виглядати інтерфейс і його стан, а фреймворк самостійно управляє оновленнями й рендерингом.

Рисунок 2.4 SwiftUI

SwiftUI надає представлення, контролери та структури розташування для декларування користувацького інтерфейсу застосунку. Фреймворк забезпечує обробниками подій для обробки торкань, жестів та інших типів введення у застосунку, а також інструменти для управління потоком даних від моделей застосунку до представлень та контролерів, які бачать та з якими взаємодіють користувачі[7].

Основні переваги включають:

1) Декларативний підхід: код описує, як має виглядати інтерфейс, концентруючись на кінцевий результат, а не на спосіб його досягнення.
2) Менше коду: у порівнянні з UIKit, для написання коду потребується менше часу та кількість рядків завдяки автоматичній обробці стану інтерфейсу.

Хоча також присутні й недоліки:

1) Обмежена сумісність: SwiftUI було представлено лише в iOS 13, тож неможливо використовувати цю технологію для старших версій ОС. Також, хоч і можливості SwiftUI з року в рік збільшуються, але чим старіша мінімальна підтримувана версія, тим менше доступного функціоналу.
2) Новизна: фреймворк, як вже сказано, є достатньо новим, а тому може мати певні недоліки та обмеження. Окрім цього, доступно менше документації та прикладів коду, що може ускладнити навчання та утрудняти розробку.
3) Продуктивність: у певних випадках SwiftUI може бути менш продуктивним, ніж UIKit, особливо у складних інтерфейсах.

4) Потреба у вивченні нового: розробникам, що вже звикли до UIKit, може знадобитись час для засвоєння нових правил та концепцій, що використовуються у SwiftUI.

2.2.3 REST API

REST API (також RESTful API або RESTful web API) – це прикладний програмний інтерфейс (Application Programming Interface, API), який відповідає принципам проєктування архітектурного стилю передачі репрезентативного стану (Representational State Transfer, REST). API REST забезпечують гнучкий, легкий спосіб інтеграції програм і підключення компонентів в архітектурі мікросервісів[8].

Для зв’язку з ресурсами використовуються такі види HTTP-запитів:

- GET
- PUT
- POST
- DELETE

Принципи, на яких працює REST API, включають:

1) Єдиний інтерфейс: усі API-запити до одного й того самого ресурсу мають виглядати однаково, незалежно від джерела запиту.
2) Розділення клієнта та сервера: клієнтські та серверні додатки повинні бути повністю незалежними один від одного.
3) Відсутність стану: кожен запит повинен містити всю необхідну для його обробки інформацію, без зберігання стану на сервері.
4) Кешованість: ресурси повинні бути кешованими на стороні клієнта або сервера, якщо це можливо.
5) Шарова архітектура системи: виклики та відповіді проходять через різні шари, і клієнт та сервер не повинні знати про наявність посередників.
6) Код на вимогу (опціонально): у відповідях можуть міститися виконувані коди, які повинні виконуватися лише на вимогу.

2.2.4 JSON

JSON (JavaScript Object Notation) – простий формат обміну даними, що є зручним як для читання та написання людиною, так і для парсингу та генерації комп'ютером[9].

Рисунок 2.5 JSON

Він використовує зрозумілу ієрархічну структуру, що дозволяє легко організовувати та зберігати дані у вигляді пар «ключ:значення». Однією з основних переваг JSON є його простота та легкість інтеграції з багатьма мовами програмування. JSON також забезпечує ефективний обмін даними між клієнтськими та серверними додатками, завдяки чому часто використовується у веброзробці для передачі даних між браузерами та серверами.

Перевагами формату JSON є:

1) Простота: формат є простим й зручним для читання як комп’ютерами, так і людьми.
2) Поширеність: JSON є досить популярним форматом, а тому має широку підтримку у різних мовах програмуваннях та бібліотеках, серед яких й Swift.

3) Легкість та компактність: формат JSON має менш громіздку структуру у порівнянні з іншими форматами, наприклад XML. Це дозволяє зберігати та передавати більшу кількість даних.
4) Ефективність: для читання даних з файлу формату JSON необхідно менше часу та ресурсів, ніж потребується на парсинг XML.

2.2.5 XML

XML (Extensible Markup Language) – це мова розмітки та формат файлів для зберігання, передачі та реконструкції довільних даних. Він визначає набір правил для кодування документів у форматі, який читається як людиною, так і машиною[10].

Рисунок 2.6 XML

XML забезпечує багату семантику завдяки підтримці атрибутів, що дозволяє описувати складні структури даних. Він широко використовується у багатьох галузях, таких як вебсервіси (SOAP), документообіг (Office Open XML) та конфігураційні файли, завдяки своїй гнучкості та можливості чітко визначати схеми даних через XML Schema Definition (XSD). Однак XML має більший обсяг і складніший синтаксис у порівнянні з JSON, що може призводити до збільшення часу обробки.

Отже, у підсумку варто зазначити такі недоліки цієї мови розмітки:

1) Об’ємність: XML має більш громіздку структуру, аніж JSON, через що розмір файлів цього формату є більшим аніж файлів формату JSON. Це призводить до збільшення часу на їх обробку.
2) Складність обробки: через складнішу структуру, потребується не лише більше часу на зчитування файлу, але й більше ресурсів.
3) Низька популярність: у розробці мобільних застосунків набагато більш часто використовується JSON, що ускладнює розробку з використанням XML через меншу кількість бібліотек для роботи з такими файлами.

2.2.6 Висновок щодо вибору технологій

Щодо фреймворку для написання користувацького інтерфейсу, було обрано використовувати UIKit через ряд його переваг, серед яких стабільність, розповсюдженість та широку підтримку. SwiftUI є непоганим варіантом, але на цю мить цей фреймворк залишається ще дуже молодим, що негативно впливає на його якості, серед яких обмежений функціонал, підтримка старих версій операційних систем та продуктивність.

Для роботи з файлами, обрано використовувати формат JSON через відсутність зайвої громіздкості, легкість обробки та високу популярність, що уможливлює використання бібліотек для роботи з даними та знижує кількість часу та ресурсів, необхідних для зчитування файлів цього формату, як порівняти з файлами формату XML.

2.3 Бібліотеки

Окрім технологій, що були описані, у застосунку також присутні певні бібліотеки та менеджер для керування ними, про які буде розписано у наступних пунктах.

2.3.1 Swift Package Manager

Swift Package Manager (SwiftPM, SPM) – розроблена Apple утиліта для керування бібліотеками. Вона була розроблена для автоматизації завантажування, компіляції та лінкування залежностей[11].

Рисунок 2.7 Swift Package Manager

Вперше Swift Package Manager було представлено разом з релізом Swift 3.0, як альтернатива вже наявним менеджерам пакетів, таких як CocoaPods та Carthage. Починаючи з Xcode 11, SwiftPM було вбудовано в інтегроване середовище розробки, що дозволяє легко завантажувати потрібні бібліотеки для роботи із застосунками під iOS, macOS, watchOS та tvOS.

Основні переваги Swift Package Manager включають:

1) Інтеграція з Swift та Xcode: SPM інтегровано безпосередньо в мову програмування Swift та інтегроване середовище розробки Xcode, що дозволяє керувати бібліотеками, не встановлюючи додаткові утиліти.
2) Автоматизація: SPM дозволяє автоматично завантажувати та встановлювати залежності з віддалених репозиторіїв, що економить час, необхідний для ручного виконання цих дій.

Однак, окрім переваг, Swift Package Manager має і недолік:

1) Обмеженість: деякі застарілі бібліотеки можуть не мати підтримку SPM.

Отже, зважаючи на те, що цей менеджер пакетів був розроблений безпосередньо компанією Apple та є рекомендованим варіантом для організації бібліотек, використання Swift Package Manager в дипломному проєкті є логічним вибором. Інтеграція з мовою програмування Swift та середовищем розробки Xcode, простота використання та автоматизоване завантаження залежностей роблять його потужним інструментом для ефективної роботи над проєктами.

Такий вибір дозволить зосередитися на розробці програмного забезпечення, мінімізуючи витрати часу на керування залежностями.

2.3.2 CocoaPods

CocoaPods – інша популярна утиліта для організації роботи із залежностями. Згідно з офіційними даними, CocoaPods має більш ніж 100 тисяч бібліотек та використовується у більш ніж 3 мільйонах застосунків[12].

Рисунок 2.8 CocoaPods

Історія CocoaPods починається 1 вересня 2011 року, коли Елой Дуран та Фабіо Пелосін після місяця розробки випустили першу релізну версію. Відтоді цей менеджер пакетів став найбільш популярним серед аналогів, надаючи зручний механізм управління залежностями.

Переваги:

1) Простота: CocoaPods є досить простою та зрозумілою утилітою для використання розробниками, яка легко встановлюється за допомогою всього однієї команди.
2) Широкий вибір бібліотек: завдяки своїй популярності та розвиненій спільноті розробників, CocoaPods має багато підтримуваних бібліотек.
3) Автоматичне оновлення залежностей: для оновлення усіх бібліотек достатньо використати одну команду, що дозволяє значно економити час.

Недоліки:

1) Необхідність встановлення: як вже було зазначено вище, у порівнянні з Swift Package Manager, що вже вбудовано в Xcode, CocoaPods необхідно додатково встановлювати.

2) Можливі конфлікти версій: інколи виникають проблеми з сумісністю між різними версіями бібліотек.

А відтак, CocoaPods хоч і є найпопулярнішим менеджером бібліотек серед розробників, однак на цей момент часу є кращі рішення.

2.3.3 Carthage

Carthage – найменш популярна, але все ще використовувана утиліта для роботи з бібліотеками[13].

Рисунок 2.9 Carthage

Carthage був не лише першим менеджером залежностей, який працював із Swift, але й написаний на Swift. Він використовує виключно динамічні фреймворки, а не статичні бібліотеки.

Переваги Carthage:

1) Децентралізований підхід: Carthage не вимагає централізованого сховища пакетів. Натомість він безпосередньо використовує GitHub, Bitbucket або будь-які інші сховища Git. Таким чином розробники можуть легко

розміщувати власні бібліотеки без необхідності публікувати їх у центральному сховищі.

Разом із тим цей менеджер пакетів має досить велику кількість недоліків:

1) Мала кількість контриб'юторів: через те, що досить невелика кількість людей беруть участь в розробці проєкту, він може мати певні помилки, а підтримуваність наступних версій ставиться під питання.
2) Складність: для встановлення бібліотеки через Carthage потребується занадто велика кількість кроків, а відтак розробнику легше та швидше буде використати інший менеджер пакетів.
3) Carthage працює лише з динамічними бібліотеками. Підтримка статичних відсутня.

Отже, не зважаючи на те, що саме цей менеджер пакетів був першим серед своїх конкурентів, наразі він здає позиції, оскільки має низку недоліків, які відсутні у Swift Package Manager та CocoaPods.

2.3.4 Alamofire

Alamofire – це мережева бібліотека HTTP, написана мовою Swift. Alamofire є найпопулярнішою бібліотекою для роботи з мережевими запитами, оскільки завдяки компактному синтаксису і широкому набору функцій дозволяє писати запити всього в кількох рядках коду[14].

Рисунок 2.10 Alamofire

Alamofire було вперше представлено 25 вересня 2014 року, і з того часу бібліотека зберігає знання лідера серед мережевих бібліотек для iOS розробки.

Переваги:

1) Популярність: як було вже зазначено, Alamofire є найбільш звантажуваною бібліотекою для роботи із мережею, а тому має велику спільноту розробників, які готові допомогти у розв’язанні проблем.
2) Простота: Alamofire має легкий синтаксис, що дозволяє економити час та рядки коду в проєкті. Також код, що було написано з використанням саме Alamofire, набагато зручніше читати іншим розробникам.
3) Автоматична валідація відповіді: за допомогою метода .validate() можна переконатись, що відповідь має дійсний статус код та тип вмісту. Це спрощує обробку помилок.

Загалом, Alamofire є чудовим вибором, який спростить роботу з мережевими запитами, оскільки має велику спільноту розробників, простоту та низку функціонала, який є простим та ефективним одночасно.

2.3.5 SwiftyJSON

SwiftyJSON – бібліотека, що полегшує роботу із JSON у Swift. Swift є мовою зі строгою типізацією. І хоча явна типізація рятує від помилок, робота із JSON стає складнішою. Саме цю проблему покликаний вирішити SwiftyJSON[15].

Переваги:

1) Простота і легкість використання: через простий і зрозумілий синтаксис для розбору JSON даних, код, написаний за допомогою цієї бібліотеки, є легким для читання та написання.
2) Безпека типів: як вже було зазначено, робота із JSON у таких мовах зі строгою типізацією, як Swift, є ускладненою. SwiftyJSON розв’язує цю проблему, знижуючи ймовірність помилок, пов’язаних з неправильними типами даних.
3) Обробка помилок: у четвертій версії бібліотеки було представлено перелічуваний тип SwiftyJSONError, який містить низку різних видів помилок. Завдяки цьому можна бути впевненим, що неправильний JSON не покличе за собою краш мобільного застосунку.

Отже, SwiftyJSON є чудовою можливістю полегшити та прискорити розробку мобільного застосунку, зекономивши при цьому кількість рядків та

полегшивши читання та написання коду. Крім того, важливо забезпечити захист застосунку від збоїв, спричинених можливими некоректними JSON файлами.

2.3.6 Kingfisher

Kingfisher – це потужна бібліотека Swift для завантаження та кешування зображень з інтернету. Вона надає можливість легко працювати із віддаленими зображеннями у мобільних застосунках[16].

Рисунок 2.11 Kingfisher

Серед основних переваг бібліотеки:

1) Асинхронність: Kingfisher дозволяє асинхронно завантажувати та кешувати зображення.
2) Гібридний кеш для пам’яті та диска: Kingfisher використовує багатошаровий кеш для збереження зображень, як у пам’яті, так і на диску, що забезпечує швидкий доступ до зображень і зменшує час завантаження.
3) Широка підтримка інтерфейсних компонентів: бібліотека підтримує завантаження зображень для різних елементів інтерфейсу користувача, серед яких UIImageView, NSImageView, NSButton, UIButton та інші.

Відтак, Kingfisher є гарним доповненням до проєкту, оскільки дозволяє розширити вбудований функціонал для віддалених зображень.

2.3.7 Висновок щодо вибору бібліотек

Swift Package Manager – потужна утиліта для роботи з бібліотеками, розроблена Apple. В ході порівняння різних менеджерів пакетів, було прийнято рішення використовувати саме SwiftPM через ряд його переваг, серед яких автоматизація, інтеграція з Xcode та безпосередньо те, що це є рекомендований варіант для роботи із залежностями.

Серед бібліотек, що буде використано при написанні мобільного застосунку, вирішено використовувати Alamofire, SwiftyJSON та Kingfisher. Ряд переваг, що мають ці бібліотеки, перевершують час, необхідний для вивчення основних правил роботи із ними, а також дозволяють виконувати розробку продукту більш ефективно та продуктивно.

Бібліотека Alamofire є популярним рішенням для роботи із мережевими запитами мовою програмування Swift. Вона пропонує простий та зрозумілий синтаксис для написання запитів та обробки відповідей. Серед основних переваг – популярність, простота та автоматична валідація відповіді.

Інструмент SwiftyJSON – інша популярна бібліотека для розробки на мові Swift. Це рішення допомагає розробникам працювати з таким форматом для обміну даних, як JSON. Вона дозволяє розв'язати проблему зі строгою типізацією даних, що присутня у Swift. Окрім цього, плюси використання бібліотеки включають простоту та легкість використання та широкий функціонал для обробки помилок.

Бібліотека Kingfisher – це потужний інструмент для роботи із зображеннями. Він дозволяє асинхронно завантажувати та кешувати зображення з інтернету та зберігати їх як у пам'яті, так і на диску. Також, важливою перевагою використання Kingfisher є широка підтримка різних інтерфейсних компонентів.

В результаті, для завантаження та організації бібліотек було обрано Swift Package Manager. Серед бібліотек обрано Alamofire, SwiftyJSON та Kingfisher для розробки мобільного застосунку. Alamofire надає простий та ефективний спосіб роботи з мережевими запитами, SwiftyJSON допомагає зручно та безпечно працювати з файлами JSON, а Kingfisher забезпечує потужні можливості для роботи із зображеннями, дозволяючи швидко їх завантажувати та кешувати. Використання саме цих інструментів сприятиме швидкій, ефективній розробці продукту, зменшуючи час, необхідний для вивчення та реалізації необхідного функціоналу, який би потребувався без використання таких бібліотек.

ВИСНОВОК ДО РОЗДІЛУ 2

В цьому розділі розглянуто низку технологій, зокрема мови програмування, фреймворки, бібліотеки та інші технології, що використовуватимуться або ж могли б використовуватись в розробці дипломного проєкту. Підбито підсумки щодо їх позитивних та негативних якостей.

Для написання мобільного застосунку під операційні системи Apple, такі як iOS, iPadOS та інші, загалом можна використовувати різні мови програмування. Найпопулярніші з них – Swift та Objective-C. Було розглянуто обидві з цих мов. Однак, хоча розробку мобільних застосунків під iOS можна виконувати на обох з них, саме мова Swift підходить до цього найкраще. Swift було розроблено компанією Apple, і саме ця мова є рекомендованою для розробки нових застосунків під мобільні платформи компанії. Важливо зазначити, що Swift є популярною, безпечною та швидкою мовою програмування, та саме ці фактори вплинули на вибір.

Що стосується інтерфейсу користувача, то при використанні мови Swift є дві найпопулярніші фреймворки: UIKit та SwiftUI. Було обрано використовувати UIKit для написання користувацького інтерфейсу через його стабільність, розповсюдженість та широку підтримку. Хоча SwiftUI є потенційно перспективним варіантом, наразі його молодість відображається в обмеженому функціоналі, відсутності підтримки старих версій операційних систем та питаннях щодо продуктивності.

Для файлів, що застосунок отримуватиме з сервера, буде використовуватись формат JSON. Цей формат має високу популярність, а також не містить занадто складної структури, що робить файли з даними більш читабельними. Також файли формату JSON потребують менше часу та ресурсів на зчитування, що робить вибір в сторону JSON очевидним.

На останок слід зазначити, що на цей час є безліч бібліотек, що значно полегшують розробку мобільних застосунків. Також є декілька утиліт, що дозволяють автоматично завантажувати та встановлювати такі бібліотеки. Ці

утиліти популярні тим, що значно спрощують роботу з залежностями, а відтак розробка стає швидкою та менш ресурсозатратною. Серед таких утиліт було зазначено Swift Package Manager, CocoaPods та Carthage. Попри те, що кожна з цих утиліт має свої переваги та недоліки, для розробки буде достатньо користуватись лиш однією з них. Вибір було зроблено на користь першого варіанту, Swift Package Manager. Swift Package Manager, або SwiftPM, є розробкою компанії Apple, що гарантує довгу підтримку цієї утиліти, нативну інтеграцію в середовищі розробки Xcode та простоту роботи.

Серед популярних та найбільш використовуваних пакетів у спільноті розробників, було обрано використовувати такі інструменти, як Alamofire, SwiftyJSON та Kingfisher. Alamofire дозволяє легко та ефективно працювати з мережевими запитами, SwiftyJSON – зчитувати файли формату JSON, а Kingfisher забезпечує завантаження та збереження фотографій.

3 ОПИС РОЗРОБКИ ЗАСТОСУНКУ

3.1 Опис функціонала системи

Мобільний застосунок для адміністратора має забезпечити йому можливість виконання таких функцій:

- Додавати нових користувачів
- Додавати нові діагностики, моделі, а також їхні версії.
- Можливість робити певну модель активною для діагностики, а також обирати поточну версію моделі.

Отже, на основі цих вимог намалюємо діаграму прецедентів мобільного застосунку на рисунку 3.1. Зокрема на діаграмі виділено усі можливі способи використання системи адміністратором.

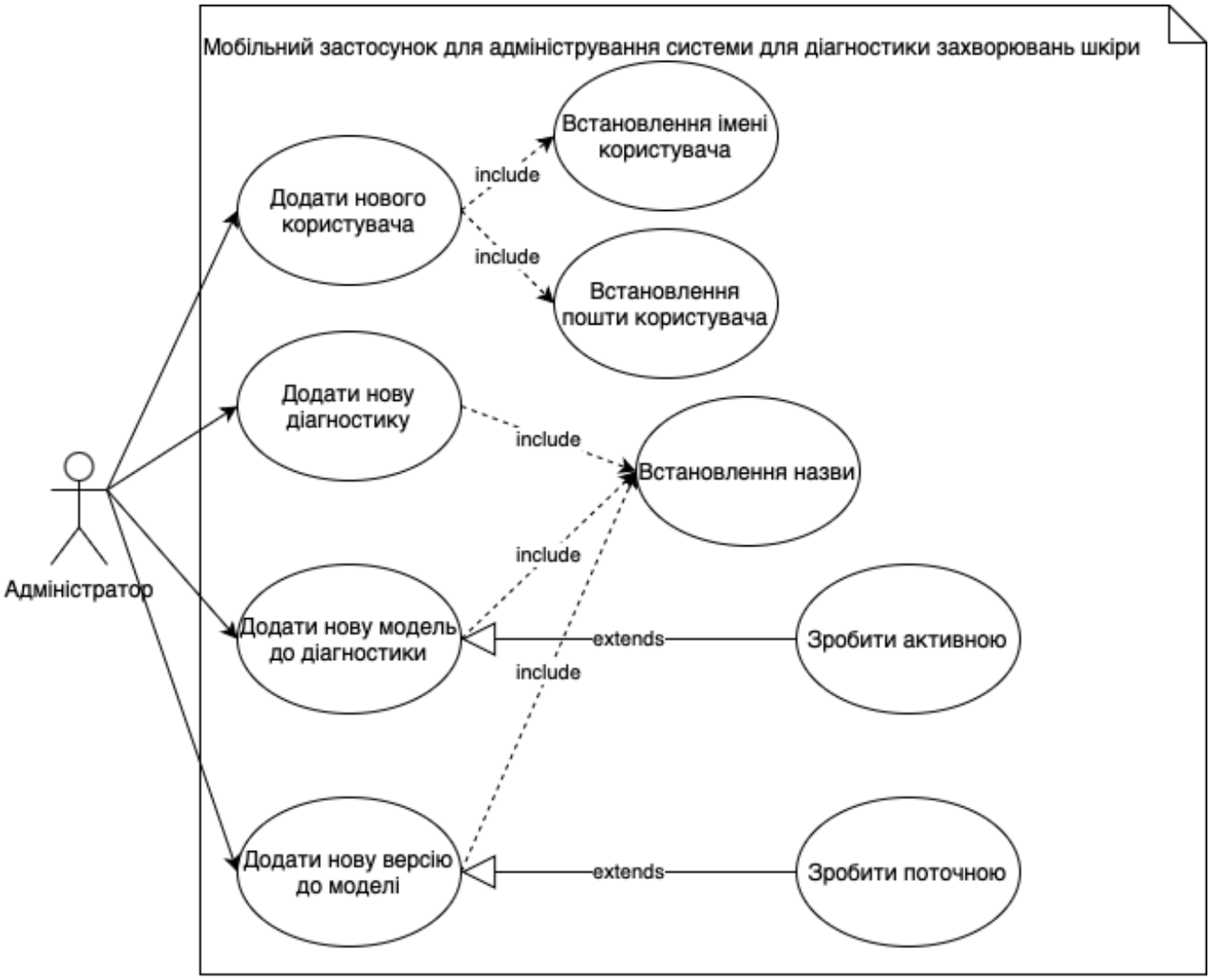


Рисунок 3.1 Діаграма прецедентів системи для адміністратора

3.2 Інтерфейс та екрани мобільного застосунку

Для виконання усіх функціональних вимог було розроблено низку екранів застосунку. Кожний екран виконує той чи інший елемент діаграми прецедентів. Усі елементи розташовано в логічному порядку, що дозволяє адміністратору інтуїтивно розуміти принцип та логіку взаємодії із застосунком. Таким чином, необхідна мінімальна кількість часу для навчання адміністратора для роботи із системою.

В застосунку було додано дев’ять екранів, кожен з яких виконує одну чи кілька функцій. Серед екранів такі:

1) Екран входу
2) Екран зі списком користувачів
3) Екран додавання нового користувача
4) Екран зі списком діагностик
5) Екран додавання нової діагностики
6) Екран зі списком моделей діагностики
7) Екран додавання нової моделі
8) Екран зі списком версій моделі
9) Екран додавання нової версії

Екрани застосунку, їх взаємозв’язки та переходи відображені у вигляді діаграми на рисунку 3.2.

Рисунок 3.2 Діаграма екранів застосунку

Написання такої великої кількості екранів дозволило поєднати інтуїтивно зрозумілу навігацію всередині застосунку та широкий функціонал.

Розробка екранів застосунку відбувається за допомогою написання контролерів перегляду (View Controller). Відповідно до офіційної документації Apple, контролер перегляду керує головним Root View, що може містити будь-яку кількість елементів (Subview). Саме контролер перегляду обробляє всі взаємодії користувача з такими елементами інтерфейсу[17]. В цьому дипломному проєкті було реалізовано такі контролери перегляду:

1) AdminLoginViewController: відповідає за екран входу.
2) AdminTabBarViewController: є ключовим для взаємодії адміністратора із вкладками на нижній частині екрана, дозволяє перемикатись між переліком користувачів та переліком діагностик.
3) UsersTableViewController: перелік користувачів.
4) AddUserViewController: додавання нового користувача.
5) DiagnosticsTableViewController: перелік діагностик.
6) AddDiagnosticViewController: додавання нової діагностики.
7) ModelsTableViewController: перелік моделей діагностики.

8) AddModelViewController: додавання нової моделі діагностики.
9) VersionsTableViewController: перелік версій моделі.
10) AddVersionViewController: додавання версії моделі.

Слід також зауважити, що деякі з перелічених контролерів, зокрема, UsersTableViewController, DiagnosticsTableViewController, ModelsTableViewController та VersionsTableViewController є успадкованими класами TableViewController. Інші ж, такі як AdminLoginViewController, AddUserViewController, AddDiagnosticViewController, AddModelViewController та AddVersionViewController, є успадкованими класами звичайного ViewController.

Кожен із зазначених контролерів перегляду міститься в окремому файлі з ідентичною назвою. Більше про архітектуру проєкту зазначено в розділі 3.3 Архітектура застосунку.

Під час розробки кожного екрану важливо не забувати про вигляд інтерфейсу користувача. Саме дизайн екранів впливатиме на кількість часу, що необхідна адміністратору для того, аби розібратись із доступним функціоналом. Відповідно, чим більш зрозумілим та простим для новачка є інтерфейс, тим менше зусиль та часу необхідно витратити людині, аби почати користуватись застосунком.

Під час розробки мобільного застосунку для адміністрування системи для діагностики шкірних захворювань було виконано підготовчу роботу щодо дизайну, а саме було прочитано основні рекомендації компанії Apple, що складені в форматі документа під назвою «Правила людського інтерфейсу» (Human Interface Guidelines)[18]. Цей документ є важливим для всіх розробників, адже він містить рекомендації для створення легких для сприйняття мобільних застосунків, а також регулює різні дизайнерські елементи, такі як іконка застосунку, розташування пунктів меню, назви екранів тощо. Відтак, було докладено максимальні зусилля для відповідності мобільним застосунком усім рекомендаціям.

Для того, аби системою могли користуватись адміністратори, що говорять різними мовами, було додано локалізацію до застосунку. Локалізація була

виконана за допомогою файлу Localizable.xcstrings, підтримку типу якого було додано у версії Xcode 15. Таким чином, порівнюючи з минулими версіями, перекладати рядки, що використовуються в застосунку стало набагато простіше. Для цього лише необхідно додати рядок, задати йому назву ключа та переклад різними мовами. Наразі було додано підтримку української та англійської мов.

3.3 Архітектура застосунку

У цьому пункті буде розглянуто архітектуру, що було використано для розробки мобільного застосунку в рамках дипломного проєкту. Вибір правильної архітектури є критично важливим для забезпечення підтримуваності, розширюваності та ефективності застосунку в майбутньому. Загалом є велика кількість архітектурних рішень, що є популярними в розробці під iOS. Однак, я зробив вибір на користь архітектурного патерну «Модель-Вигляд-Контролер» (Model-View-Controller, MVC).

Як можна побачити зі сценарію на рисунку 3.3, архітектурний патерн «Модель-Вигляд-Контролер» складається із трьох елементів: Модель (Model), Вигляд (View) та Контролер (Controller), кожен з яких є відповідальним за свою частину в застосунку.

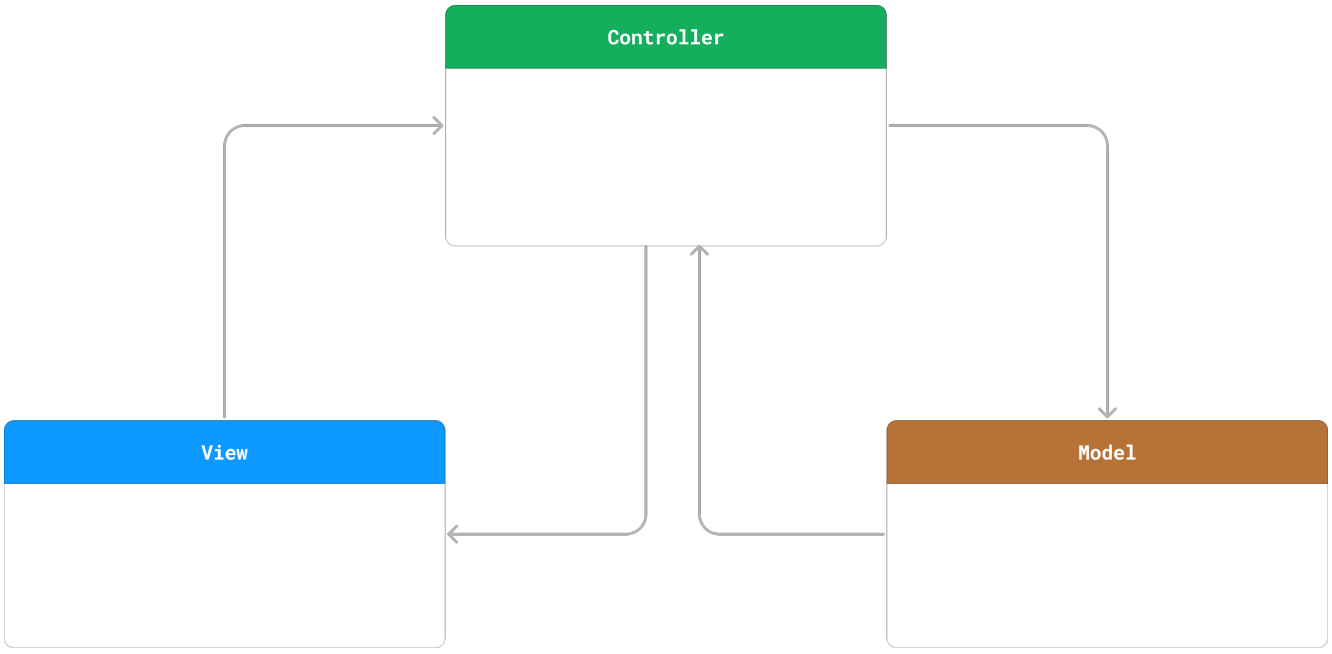


Рисунок 3.3 Архітектура «Модель-Вигляд-Контролер»

Таке розділення архітектурного патерну «Модель-Вигляд-Контролер» на три ключові елементи в iOS-розробці забезпечує чітке розділення обов’язків, де «Модель» відповідає за дані та логіку бізнес-процесів, «Вигляд» відповідає за відображення елементів на екрані застосунку, а «Контролер» виступає посередником між ними, оброблюючи вхідні дані від користувача, а також надсилаючи сигнали до «Моделі» для оновлення чи збереження даних. Також «Контролер» надсилає запити до «Вигляду» для змінення зовнішнього вигляду інтерфейсу. Таким чином це розділення дозволяє полегшити тестування, оскільки набагато легше тестувати кожний компонент застосунку окремо. Окрім цього, це покращує підтримку та масштабування застосунку, дозволяючи мінімізувати вплив змін в одному компоненті на інші компоненти. Така архітектура, як «Модель-Вигляд-Контролер», підвищує гнучкість, дозволяючи змінювати інтерфейс чи бізнес-логіку без необхідності переписувати увесь код. Вона також сприяє повторному використанню коду, покращує організацію проєкту, роблячи його більш зрозумілим для читання та орієнтації новими розробниками.

Відповідно до архітектурного патерну Model-View-Controller файли в застосунку були розподілені необхідним чином. Більш детально структуру файлів можна побачити на рисунку 3.4.

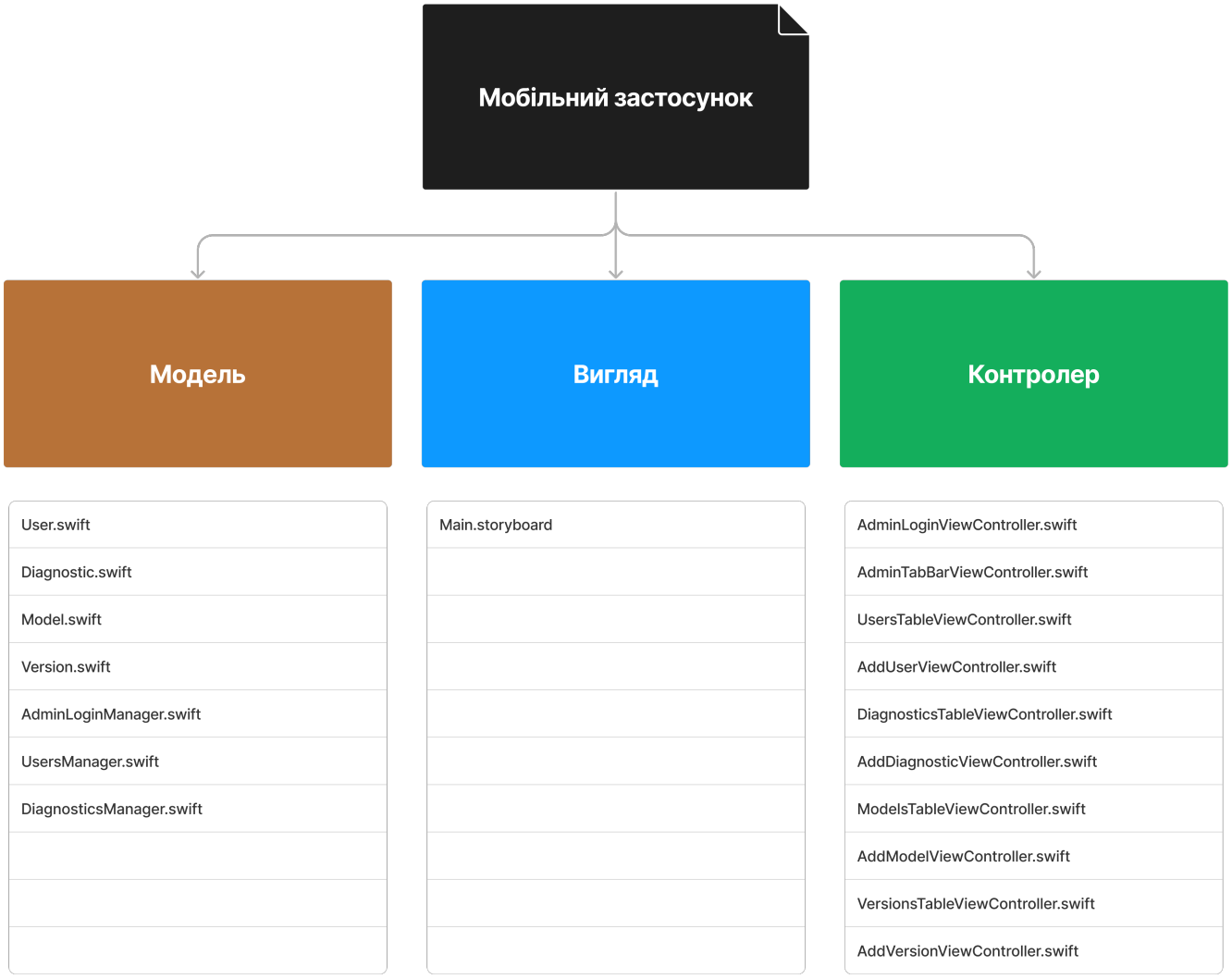


Рисунок 3.4 Структура файлів в проєкті

Отже, існують такі три об’єкти в архітектурній моделі «Модель-Вигляд-Контролер»:

1) Модель: місце, де зберігаються дані, описуються класи, структури, є менеджери та мережевий код.
2) Вигляд: частина застосунку, що відповідає за зовнішній вигляд екранів. Наприклад, до цієї частини відносяться такі класи, як UILabel, UITextField, UIImage.
3) Контролер: посередник між частинами «Модель» та «Вигляд». Саме тут здійснюється зв’язок між елементами інтерфейсу та даними застосунку.

Опишемо більш детально про кожну з цих частин.

3.3.1 Модель

Модельний шар охоплює дані застосунку. Зазвичай, до цього шару входять класи та об’єкти, що відповідають за такі речі:

1) Мережева взаємодія: клас для мережевої комунікації, що спрощує абстрагування загальних концепцій, таких як HTTP заголовки, обробка відповідей та помилок.
2) Збереження даних: цей клас використовується безпосередньо для збереження даних у базі даних, таких як Core Data, SwiftData тощо.
3) Парсинг: об’єкти, що парсять мережеві відповіді, також входять до модельного шару. Наприклад, такі об’єкти можуть парсити JSON файли зі списком користувачів для подальшої обробки.
4) Менеджери й абстракції: класи, що діють як зв’язок між іншими класами чи обгортають більш низькорівневі API, такі як робота з keychain або ж зі сповіщеннями.
5) Джерела даних й делегати: модельні об’єкти можуть бути джерелом даних чи делегатами для інших компонентів, таких як таблиці чи колекції. Однак частіше це все ж імплементовано в шарі «Контролер».
6) Константи: файл з певними рядками чи змінними, що можуть бути повторно використані в коді. Наприклад, рядки, що використовуються в інтерфейсі чи ідентифікатори екранів та переходів.
7) Хелпери та розширення: моделі для розширення властивостей рядків, колекцій тощо також є частиною модельного шару.

Отже, модельний шар є відповідальним за дані та за роботу з даними, зокрема їх отримання, обробку та збереження.

3.3.2 Вигляд

Шар «Вигляд» є відповідальним за взаємодію користувача із застосунком. В цьому шарі заведено описувати інтерфейс та займатись організацією його елементів. Також тут не має міститись жодної бізнес-логіки. Зазвичай до шару входять такі класи:

1) Підкласи UIView: це можуть бути базові UIView та користувацькі складніші класи, що успадковані від них.
2) Класи, що є частиною UIKit. Наприклад, UILabel чи UIImage.
3) Core Animation: Core Animation використовується для створення плавних анімацій та переходів в застосунку.
4) Core Graphics: Core Graphics відповідає за рендеринг графіки, а також за можливість малювання, візуалізації та анімації графіки.

Підсумовуючи, шар вигляду є теж важливим для роботи застосунку, адже він забезпечує безпосередню взаємодію користувача з даними шляхом додавання інтерфейсу. Гарний та ефективний інтерфейс не тільки сприяє покращенню користувацького досвіду, але й дозволяє застосунку залишатись легким для сприйняття та ефективним з погляду часу, що користувачу необхідно витратити, аби отримати необхідний йому функціонал.

3.3.3 Контролер

Шар контролеру є надзвичайно важливим, адже він забезпечує роботу всього застосунку шляхом взаємодії між такими шарами, як «Модель» та «Вигляд». В коді зазвичай кожний екран має свій контролер.

Отже, функціонал шару «Контролер» можна коротко описати такими пунктами:

1) Управління потоком даних
2) Оновлення даних на екрані
3) Навігація між екранами
4) Обробка подій
5) Управління станами
6) Отримка даних з шару «Модель» та передача в шар «Вигляд»
7) Ініціалізація процесів

3.4 Процес входу адміністратора в застосунок

Аби скористатись застосунком, адміністратору необхідно спочатку в нього зайти, використовуючи облікові дані від свого профілю. Зокрема, такими даними в системі є електронна пошта та пароль.

Розпишемо більш детально процес входу адміністратора в систему.

Автентифікація – це процес, під час якого відбувається перевірка особи користувача чи адміністратора, тобто це процес під час якого відбувається порівняння даних, що увів користувач із наявними у базі даних. Такими даними є електронна пошта та пароль. Отже, коли користувач вводить свої облікові дані та натискає на кнопку «Вхід», відбувається перевірка, чи дійсно користувач є тим, за кого себе видає.

Авторизація – це процес надання чи обмеження доступу користувача до певних ресурсів або функцій системи після успішної автентифікації. Це визначення того, які саме дії може виконувати користувач у системі на основі його прав та можливостей.

Отже, автентифікація має такі характеристики:

1) Мета: підтвердження особи користувача.
2) Процес: верифікація даних користувача, таких як електронна пошта та пароль.
3) Результат: користувач підтверджений як той, за кого себе видає.

А авторизація, своєю чергою, має такі характеристики:

1) Мета: визначити права доступу користувача до ресурсів або функцій системи.
2) Процес: перевірка дозволів на виконання певних дій після автентифікації.
3) Результат: надання або обмеження доступу до конкретних ресурсів або дій.

Відтак, автентифікація стосується перевірки особи, тоді як авторизація стосується визначення, що саме така особа може робити в застосунку. Важливо зазначити, що в ході виконання завдання з розробки застосунку в цьому бакалаврському проєкту було виконано лише автентифікацію адміністратора, оскільки саме ця частина є головним завданням до проєкту.

Тільки після автентифікації адміністратор може переглядати список користувачів системи, список діагностик, що доступні, а також списки моделей та версій до них. Лише автентифікована людина може вносити зміни до цих списків шляхом додавання нових користувачів та діагностик, а також й нових моделей до діагностик та версій моделей.

Зобразимо схематично, як саме відбувається послідовність авторизації на рисунку 3.5.

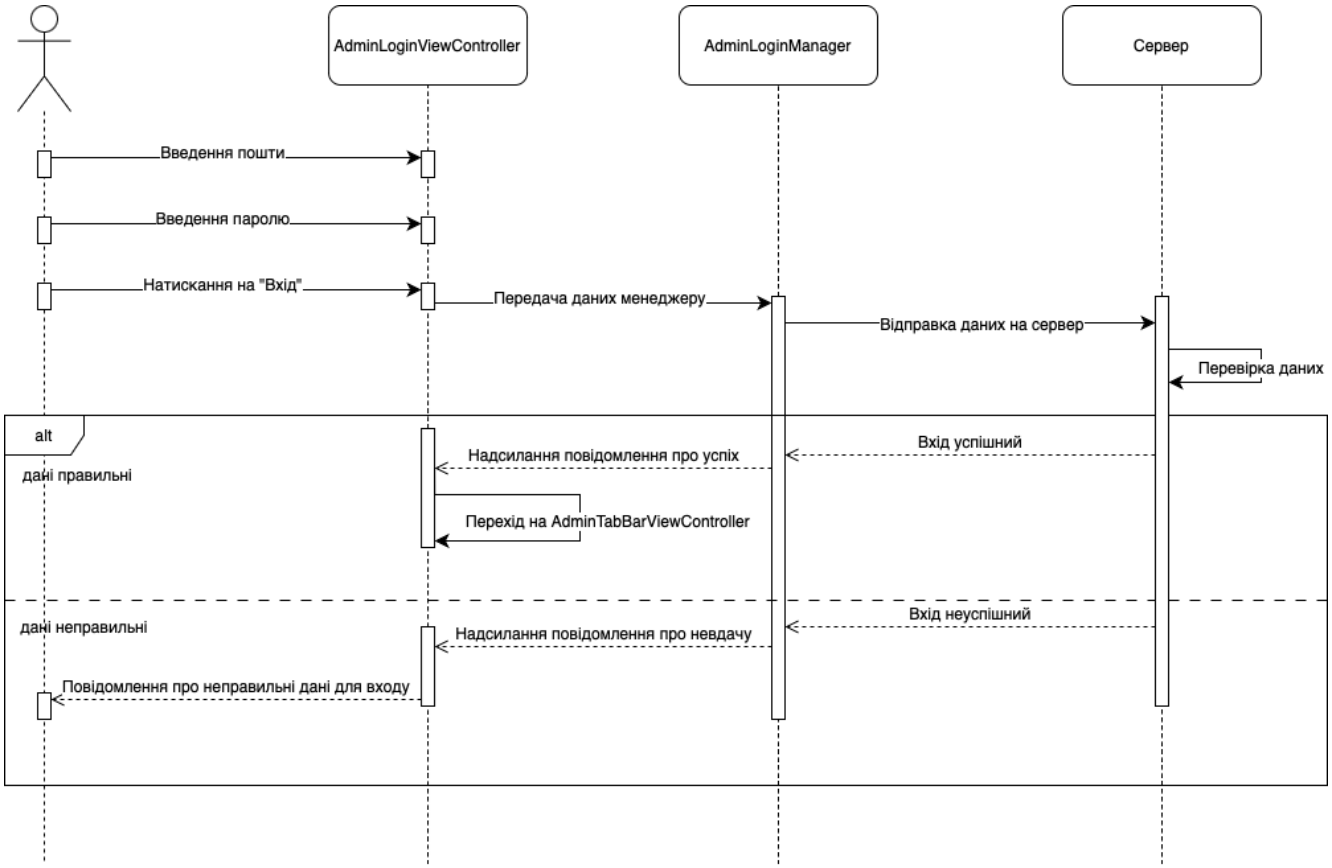


Рисунок 3.5 Діаграма послідовності автентифікації

Як можна побачити з діаграми послідовності автентифікації, процес входу в застосунок є доволі простим. На екрані входу адміністратор вводить свої облікові дані від акаунту, а саме пошту та пароль. Після того, як користувач увів свої дані, йому необхідно натиснути на кнопку «Вхід». Слід зазначити, що ця кнопка стає активною лише після того, як користувач увів хоча б один символ в поля для вводу електронної пошти та паролю. Після натискання на кнопку «Вхід» контролер екрана входу передає ці дані менеджеру, що надсилає дані на сервер. Сервер проводить певну перевірку цих даних, ймовірно звіряючи їх із даними, що

містяться в базі даних, та надсилає повідомлення про успішний вхід чи неуспішну спробу входу. У разі, якщо дані правильні та вхід було дозволено сервером, контролер здійснює перехід на екран з вкладками (перелік користувачів та перелік діагностик). Якщо ж дані було введено неправильні, то користувач отримує повідомлення про помилку, після чого він може спробувати здійснити нову спробу входу.

3.5 Розгортання системи

Мобільний застосунок для адміністрування системи для діагностики шкірних захворювань, що було розроблено під час виконання цього бакалаврського проєкту, було написано для операційної системи iOS, що працює на смартфонах компанії Apple під назвою iPhone. Оскільки ця операційна система є доволі закритою, то встановлення зовнішніх застосунків на такі телефони в Україні можливе лиш одним шляхом – за допомогою магазину застосунків від Apple, що називається App Store. Важливо зазначити, що у країнах Європейського Союзу також можливе завантаження застосунків з інших джерел, наприклад, з інших магазинів застосунків чи безпосередньо з вебсайту розробника застосунку[19]. Для завантаження застосунків з інших магазинів застосунків необхідно мати версію ОС iOS 17.4 та вище, а для завантаження з вебсайту розробника застосунку – ОС iOS 17.5 та вище. Однак, на момент написання дипломного проєкту, така можливість недоступна для користувачів з України, оскільки Україна поки що не є державою-членом Європейського Союзу та на неї не поширюється дія «Закону про цифрові ринки» (Digital Markets Act).

Рисунок 3.6 App Store

Окрім встановлення застосунку з магазину застосунків чи з вебсайту розробника застосунку, також можливий варіант самостійного збирання та запуску застосунку через інтегроване середовище розробки Xcode. Для цього необхідно завантажити репозиторій із вихідним кодом застосунку та іншими необхідними для його роботи файлами, після чого відкрити проєкт в Xcode, встановити необхідні параметри в меню Signing & Capabilities в Target проєкту, а саме встановити команду (Team) та ідентифікатор пакунка (Bundle Identifier). Таким чином можливе встановлення застосунку не лише безпосередньо на мобільний телефон, але й запуск його в симуляторі пристроїв Apple.

Також окремим варіантом розповсюдження застосунку є використання TestFlight. TestFlight – це спеціальна система, розроблена компанією Apple, що дозволяє розробникам легко розповсюджувати свій застосунок для його тестування[20]. Такий метод є гарним варіантом для знаходження недоліків у програмі, оскільки дозволяє збирати відгуки від користувачів, не потребуючи при цьому значних зусиль. Однак недоліком такого методу розповсюдження застосунку є те, що такий спосіб призначений суто для тестування та має обмеження в 10 000 користувачів.

Отже, на момент розробки застосунок встановлювався на пристрій за допомогою інтегрованого середовища розробки Xcode, що дозволяло швидко реагувати на недоліки, виправляти різні елементи інтерфейсу та вирішувати помилки. Наступним етапом розповсюдження системи є публікація застосунку в магазин застосунків App Store, що дозволить користувачам з усього світу мати можливість завантажити його собі на телефон.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі бакалаврського проєкту було описано етапи та деталі розробки мобільного застосунку для адміністрування системи для діагностики шкірних захворювань. Застосунок було написано для операційної системи iOS для використання на смартфонах компанії Apple iPhone.

Функціонал застосунку дозволяє адміністратору виконувати такі функції:

1) Додавати нових користувачів.
2) Додавати нові діагностики, моделі та їхні версії.
3) Робити певну модель активною для діагностики.
4) Обирати поточну версію моделі.

Для реалізації функціонала було використано архітектурний патерн «Модель-Вигляд-Контролер» (Model-View-Controller, MVC). Цей патерн розділяє застосунок на три логічні частини: модель, вигляд і контролер. Кожна з цих частин має свою відповідальність:

1) Модель: зберігає дані застосунку.
2) Вигляд: відповідає за інтерфейс користувача.
3) Контролер: зв’язує модель та вигляд, обробляє події та оновлює інтерфейс.

Вхід в систему для адміністратора здійснюється за допомогою електронної пошти та пароля. Після успішного входу адміністратор отримує доступ до всіх функцій застосунку.

Розгортання застосунку можливе декількома способами:

1) Публікація застосунку в магазин App Store.
2) Самостійне збирання та запуск через Xcode.
3) Використання TestFlight.

На момент написання дипломного проєкту застосунок встановлювався на пристрій за допомогою Xcode. Наступним етапом буде публікація застосунку в магазин застосунків App Store.

Отже, підсумовуючи, можна сказати, що було розроблено мобільний застосунок для адміністрування системи діагностики шкірних захворювань. Окрім

цього було також описано етапи розробки, функціонал, обґрунтовано вибір MVC, описано процес входу адміністратора, і наостанок наведено способи розгортання застосунку. Розроблена програма є зручним та функціональним інструментом, що дозволяє адміністратору виконувати всі необхідні дії зі свого мобільного телефону. Застосунок має простий та зрозумілий інтерфейс, а також високу продуктивність.

4 МЕТОДИКА РОБОТИ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

В цьому розділі буде детально описано увесь результат роботи над застосунком. Зокрема буде продемонстровано роботу за допомогою скриншотів та вичерпних пояснень до них.

4.1 Вхід адміністратора

На екрані входу адміністратора (рисунок 4.1) у верхній частині знаходиться напис, що сповіщає про можливість входу в систему адміністрування. Посередині екрана користувач бачить два поля для вводу його електронної скриньки та пароля, а трохи нижче розташована кнопка «Вхід». Після того, як адміністратор вводить свої облікові дані, йому необхідно натиснути на цю кнопку, аби потрапити до екранів з основним функціоналом застосунку.

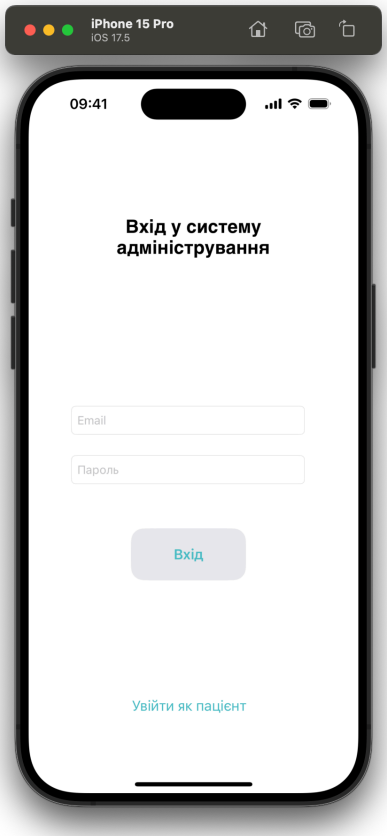


Рисунок 4.1 Вхід адміністратора

Важливо зазначити, що натиснути на кнопку входу можна лише при умові, що поля «Email» та «Пароль» не є пустими. Окрім цього, застосунок перевіряє

електронну пошту на правильність, що також знижує ймовірність неуспішної спроби входу.

Також досить важливою частиною застосунку є його локалізація різними мовами. На цей час є підтримка української та англійської мови. Таким чином навіть адміністратори, що не знають української мови, але знають англійську, можуть використовувати цей застосунок. На рисунку 4.2 продемонстровано переклад англійською.

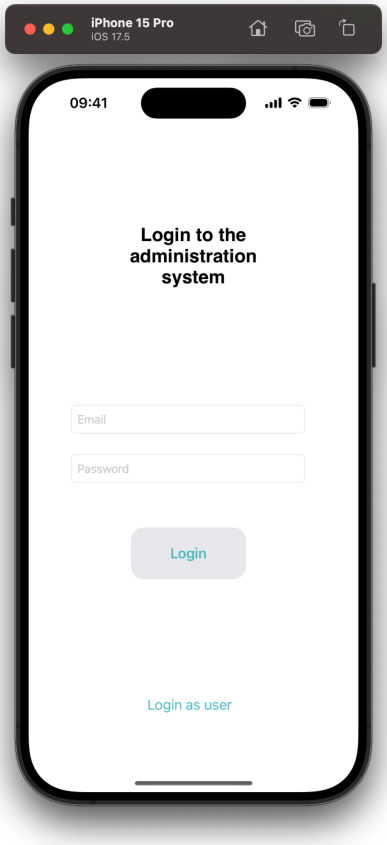


Рисунок 4.2 Переклад англійською

Як можна побачити, усі елементи інтерфейсу, зокрема заголовок, кнопки та поля для вводу мають написи англійською мовою, що дозволяє розуміти значення структурних частин застосунку англомовним користувачам. Локалізація застосунків є важливою складовою під час розробки систем різного рівня, а саме тому вона й була зроблена.

4.2 Список користувачів

Після того, як адміністратор здійснив вхід в систему, він потрапляє на екран із двома вкладками: вкладка зі списком користувачів та вкладка зі списком діагностик. Перемикання між цими вкладками відбувається за допомогою спеціальної панелі внизу екрана.

На екрані зі списком користувачів (рисунок 4.3) адміністратор бачить безпосередньо всіх користувачів, що мають облікові записи в системі. Задля безпеки та протидії витоку цінних даних на цьому екрані адміністратор бачить лише ім’я користувача (без прізвища), а також його електронну пошту.

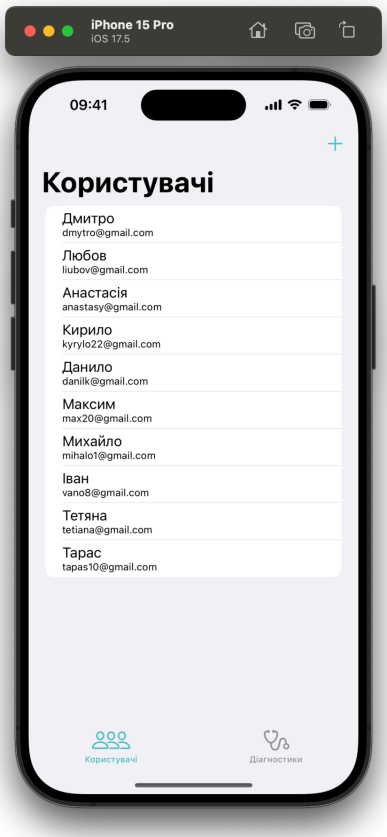


Рисунок 4.3 Список користувачів

Важливими елементами на екрані зі списком користувачів також є заголовок та кнопка у формі плюсу у верхній частині пристрою. Заголовок відображає зміст списку, маючи напис «Користувачі», а кнопка є значущим компонентом інтерфейсу, оскільки дозволяє здійснити перехід на екран додавання нового користувача системи.

4.3 Додавання нового користувача

При натисканні на кнопку «Плюс» на екрані зі списком користувачів системи адміністратор потрапляє на екран додавання нового користувача, де можна здійснити реєстрацію людини в системі. На рисунку 4.4 представлено скриншот, на якому видно розміщення складових екрану, серед яких, зокрема, поля для вводу імені та пошти, а також кнопка «Додати користувача».

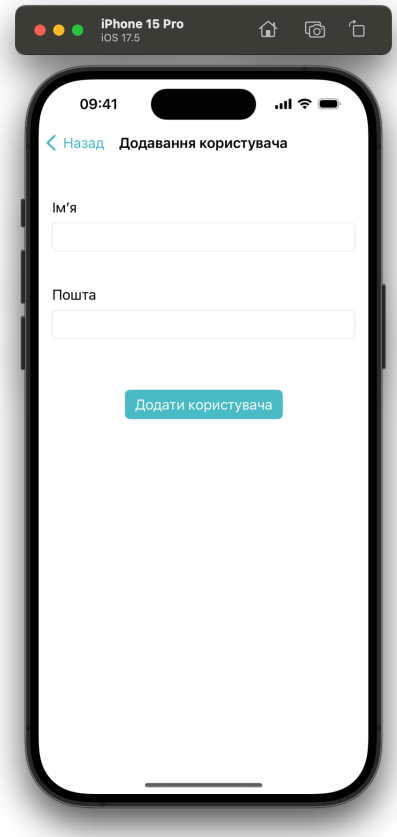


Рисунок 4.4 Додавання нового користувача

Важливо зазначити, що адміністратор при натисканні на кнопку «Додати користувача» не лише додає людину до списку на попередньому екрані, але й реєструє її в системі. З питань безпеки адміністратор не може встановлювати пароль, а тому й нема відповідного поля. Пароль генерується системою на серверній частині та відправляється користувачу на пошту.

На завершення, слід зауважити, що після того, як користувача було зареєстровано, адміністратор повертається на екран зі списком користувачів.

4.4 Список діагностик

Подібно до екрана зі списком користувачів, екран зі списком діагностик є іншою вкладкою у застосунку. На цьому екрані розміщується список усіх діагностик, що доступні для адміністрування в системі. На рисунку 4.5 можна побачити безпосередньо сам список, заголовок екрана, в якому описується зміст списку, а також кнопку у формі плюсу.

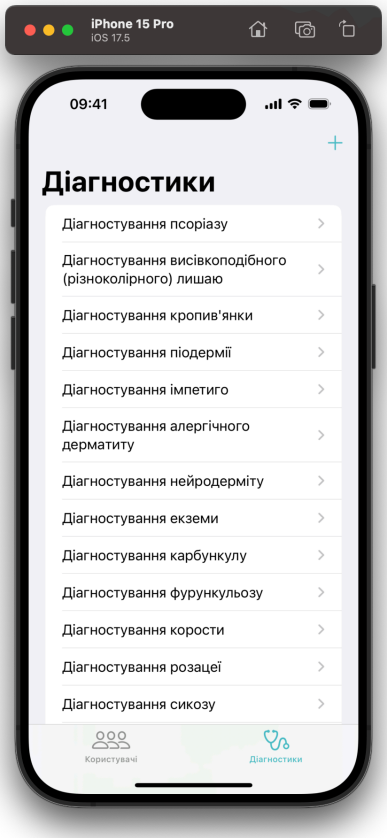


Рисунок 4.5 Список діагностик

Так само як і з екраном зі списком користувачів, при натисканні на кнопку «Плюс» адміністратор потрапляє на новий екран, в якому він може додати нову діагностику в систему.

4.5 Додавання нової діагностики

Після того, як адміністратор натисне на кнопку «Плюс», він потрапляє на екран додавання нової діагностики. На цьому екрані можна увести бажану назву діагностики.

На рисунку 4.6 можна побачити, що екран містить декілька елементів, серед яких заголовок, поле для вводу назви діагностики та безпосередньо кнопка «Додати діагностику». При натисканні на цю кнопку адміністратор реєструє діагностику в системі та потрапляє на минулий екран зі списком усіх зареєстрованих діагностик.

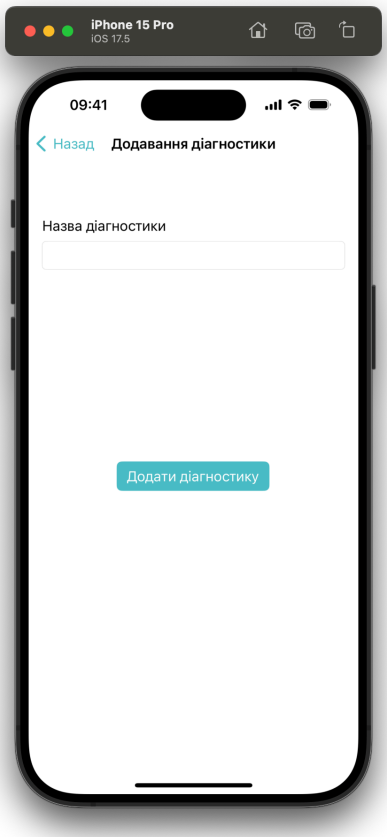


Рисунок 4.6 Додавання нової діагностики

Варто зазначити, що під час створення діагностики, її властивостями є не лише назва та список моделей, але й унікальний ідентифікатор UUID, що створюється автоматично застосунком.

4.6 Список моделей певної діагностики

Список моделей є екраном, на який відбувається перехід при натисканні на певну діагностику. Ці моделі прив’язані до конкретної діагностики, а тому у верхній частині екрана можна побачити назву обраної діагностики.

На рисунку 4.7 зображено скриншот цього екрана, на якому видно, що у верхній частині зображено заголовок, а саме назву тої діагностики, що адміністратор обрав на минулому екрані.

Рисунок 4.7 Список моделей певної діагностики

Також, як можна побачити, на екрані присутні кнопка для додавання нової моделі до обраної діагностики та безпосередньо сам список вже наявних моделей. Окремо варто зазначити, що, оскільки в системі присутній функціонал призначення певної моделі відповідальною за діагностику, або ж активною, то і в мобільному застосунку для адміністрування цієї системи також присутня така можливість. Як можна побачити на рисунку 4.7 «Модель» була обрана активною за діагностику, а тому має припис «активна» та виділена іншим кольором – зеленим. Активна модель, що присутня у списку всіх моделей діагностики, є головною моделлю. Саме така модель використовується для діагностики шкірного захворювання. Відтак, вкрай важливо правильно налаштовувати діагностики, зокрема шляхом вибору активної моделі.

Через те, що у системі присутній функціонал обрання тої чи іншої моделі діагностики активною, то в мобільному застосунку для адміністрування цієї системи існує аналогічна можливість. На рисунку 4.8 видно, що помітка моделі як активної відбувається шляхом натискання на кнопку, що з'являється при зміщенні рядка з моделлю ліворуч.

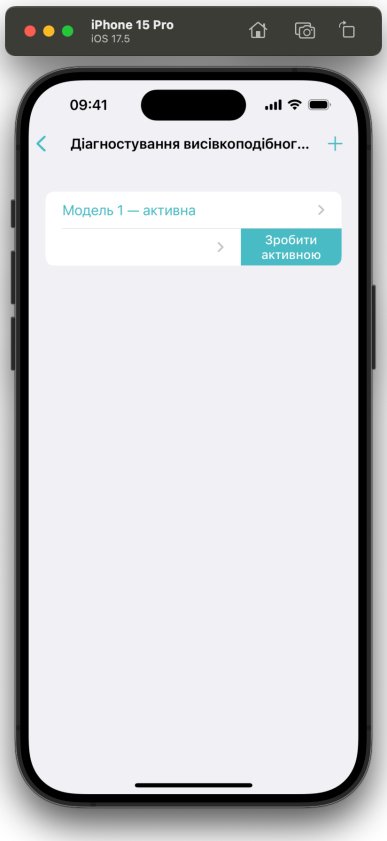


Рисунок 4.8 Можливість зробити модель активною

Після натискання на кнопку «Зробити активною» змінюється не лише представлення цієї моделі в базі даних на сервері, але й візуальне представлення в мобільному застосунку. А саме змінюється колір тексту на зелений та додається примітка «активна», що дозволяє зрозуміти, яка саме модель є активною для тої чи іншої діагностики.

4.7 Додавання нової моделі діагностики

Для того, аби додати нову модель до певної діагностики, адміністратор має натиснути на кнопку «Плюс» на екрані зі списком моделей.

На рисунку 4.9 зображено екран, що уможливлює додавання нової моделі. Зокрема на екрані присутній заголовок із назвою функції, що виконується, а також такі поля, як назва діагностики, що було обрано на минулих екранах, та назва моделі, яку адміністратор задає власноруч.

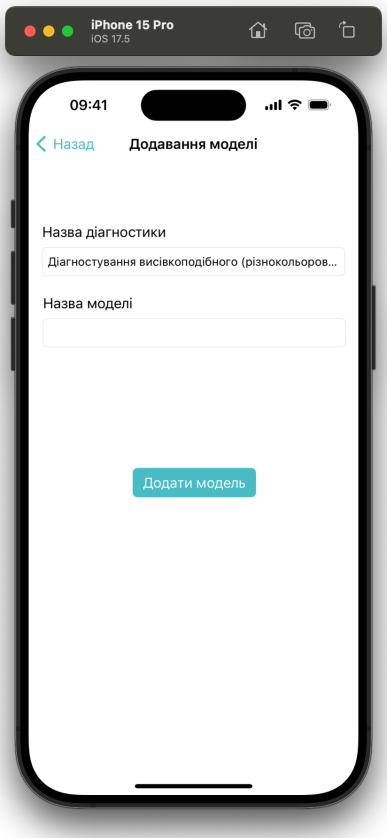


Рисунок 4.9 Додавання нової моделі діагностики

При натисканні на кнопку «Додати модель» застосунок відправляє запит на сервер із новою інформацією, а адміністратор переходить назад на екран зі списком моделей.

4.8 Список версій моделі

Екран зі списком версій моделі є аналогічним до екрана зі списком моделей до діагностики. Перехід на цей екран відбувається після натискання на ту модель, що цікавить адміністратора.

На рисунку 4.10 зображено скриншот екрана зі списком версій моделі «Модель 1» з попереднього пункту. Зокрема можна побачити заголовок із назвою моделі, кнопку для додавання нової версії та сам список.

Рисунок 4.10 Список версій моделі

Також реалізований функціонал обрання певної версії моделі як поточної. Так само як і на екрані зі списком моделей, версія, що позначена, як поточна, має інший колір шрифту в рядку, а також має примітку «поточна», що вказує на те, що саме ця версія використовується при аналізі зображення.

Для того, аби адміністратор міг зробити ту чи іншу версію поточною, було реалізовано спеціальний функціонал для цього. Схожим чином, що й на екрані зі списком моделей, тут можна потягнути за рядок з назвою версії ліворуч, після чого з’явиться кнопка «Зробити поточною», при натисканні на яку бажана версія буде помічена поточною в базі даних та в самому застосунку.

На рисунку 4.11 більш детально показано даний функціонал. Тут, на цю мить, версія 1.2 є поточною, однак рядок з версією 1.1 було посунуто ліворуч, що дозволило з’явитись кнопці для обрання версії поточною.

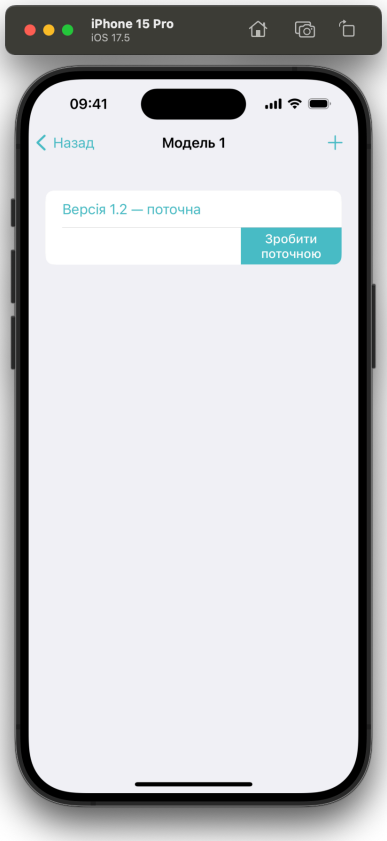


Рисунок 4.11 Можливість зробити версію поточною

4.9 Додавання нової версії моделі

Для того, аби можна було додавати нові версії моделі, було зроблено екран, перехід на який відбувається після натискання на кнопку «Плюс» на списку з усіма версіями. На цьому екрані, аналогічно до пункту 4.7 є такі поля, як «Назва діагностики», «Назва моделі», а також є поле «Назва версії», куди адміністратор вносить дані.

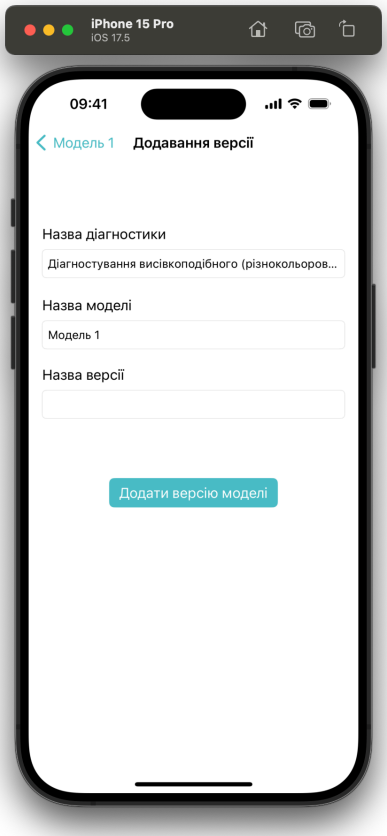


Рисунок 4.12 Додавання нової версії моделі

ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі детально описується методика роботи із розробленим застосунком для адміністрування системи для діагностики шкірних захворювань. За допомогою детальних скриншотів симулятора пристрою iPhone 15 Pro на операційній системі iOS 17.5, на який встановлено застосунок через інтегроване середовище розробки Xcode, та вичерпних пояснень до них продемонстровано функціонал усіх екранів застосунку. Зокрема, було описано екран входу, екрани списків користувачів та діагностик, екрани додавання нового користувача, додавання нової діагностики, а також екрани зі списками моделей діагностики та версіями моделі, екрани додавання нової моделі та додавання нової версії.

Отже, екрани виконують такі функції:

1) Екран входу адміністратора: забезпечує безпечний доступ до системи за допомогою реєстраційних даних.
2) Список користувачів: відображає список зареєстрованих користувачів, показуючи їхні імена та адреси електронних скриньок.
3) Додавання нового користувача: дозволяє адміністратору реєструвати нових користувачів в системі.
4) Список діагностик: відображає список усіх доступних діагностик.
5) Додавання нової діагностики: дозволяє адміністратору додавати нові діагностики в систему.
6) Список моделей певної діагностики: відображає список моделей, пов’язаних з певною діагностикою.
7) Додавання нової моделі діагностики: дозволяє адміністратору додавати нові моделі до діагностики.
8) Список версій моделі: відображає список версій певної моделі.
9) Додавання нової версії моделі: дозволяє адміністратору додавати нові версії моделей до діагностики.

Варто зазначити, що застосунок завдяки чіткій структурі та зручному мобільному інтерфейсу забезпечує ефективне адміністрування системи для діагностики шкірних захворювань.

Окрім того, важливими складовими застосунку є його локалізація різними мовами, зокрема українською та англійською, а також можливість призначення активною якусь модель та поточною певну версію.

В цілому, розроблений застосунок є зручним та функціональним інструментом для адміністрування системи.

ВИСНОВКИ

У сучасному світі мобільні технології стали невіддільною частиною нашого життя. Багато сфер знаходять нові моделі роботи з використанням таких технологій. Сфера охорони здоров’я не є винятком. Швидкість та зручність мобільних застосунків роблять їх ідеальними інструментами для інноваційних підходів до діагностики та лікування.

Для полегшення адміністрування системи для діагностики шкірних захворювань в ході роботи над цим дипломним проєктом було розроблено мобільний застосунок для адміністраторів. За допомогою цього застосунку адміністратори можуть виконувати низку функціонала для налаштування системи. Зокрема, реєструвати та додавати й налаштовувати нові діагностики з використанням нейромережевих моделей.

Розроблений застосунок можна назвати зручним, надійним та функціональним рішенням для адміністрування системи діагностики шкірних захворювань. Він має простий та зрозумілий інтерфейс, що перекладено українською та англійською мовами. В умовах обмежених можливостей роботи з громіздким комп’ютером, саме мобільний застосунок такого типу стає у пригоді.

У майбутньому можливий варіант розширення функціонала адміністрування та публікація застосунку в магазин застосунків App Store, що зробить можливим його завантаження з будь-якої точки планети. Окрім цього, можна також адаптувати застосунок й під інші системи, що мають на меті допомогти у діагностиці інших захворювань.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. 5 Public Health Technologies to Revolutionize Health Care Delivery. URL: https://mphdegree.usc.edu/blog/public-health-technology (дата звернення: 05.05.2024).
2. skin-issue-detector. URL: https://github.com/SecretSeeker/skin-issue-detector (дата звернення: 05.05.2024).
3. SkinScanner. URL: https://github.com/vitoriapinheiro/SkinScanner (дата звернення: 05.05.2024).
4. Global Top 10 Best-Selling Smartphones: All From Apple, Samsung. URL: https://www.forbes.com/sites/johnkoetsier/2024/05/06/global-top-10-best-selling-smartphones-all-from-apple-samsung/ (дата звернення: 05.05.2024).
5. Objective-C. URL: https://en.wikipedia.org/wiki/Objective-C (дата звернення: 05.05.2024).
6. UIKit. URL: https://developer.apple.com/documentation/uikit (дата звернення: 05.05.2024).
7. SwiftU. URL: https://developer.apple.com/documentation/swiftui (дата звернення: 05.05.2024).
8. What is a REST API?. URL: https://www.ibm.com/topics/rest-apis (дата звернення: 05.05.2024).
9. Вступ у JSON. URL: https://www.json.org/json-uk.html (дата звернення: 05.05.2024).
10. XML. URL: https://en.wikipedia.org/wiki/XML (дата звернення: 05.05.2024).
11. Package Manager. URL: https://www.swift.org/documentation/package-manager/ (дата звернення: 05.05.2024).
12. CocoaPods. URL: https://cocoapods.org (дата звернення: 05.05.2024).
13. Carthage. URL: https://github.com/Carthage/Carthage (дата звернення: 05.05.2024).

14. Alamofire. URL: https://github.com/Alamofire/Alamofire (дата звернення: 05.05.2024).
15. SwiftyJSON. URL: https://github.com/SwiftyJSON/SwiftyJSON (дата звернення: 05.05.2024).
16. Kingfisher. URL: https://github.com/onevcat/Kingfisher (дата звернення: 05.05.2024).
17. View controllers. URL: https://developer.apple.com/documentation/uikit/view_controllers (дата звернення: 05.05.2024).
18. Human Interface Guidelines. URL: https://developer.apple.com/design/human-interface-guidelines (дата звернення: 05.05.2024).
19. About alternative app distribution in the European Union. URL: https://support.apple.com/en-us/118110 (дата звернення: 05.05.2024).
20. TestFlight. URL: https://developer.apple.com/testflight/ (дата звернення: 05.05.2024).

ДОДАТОК 1

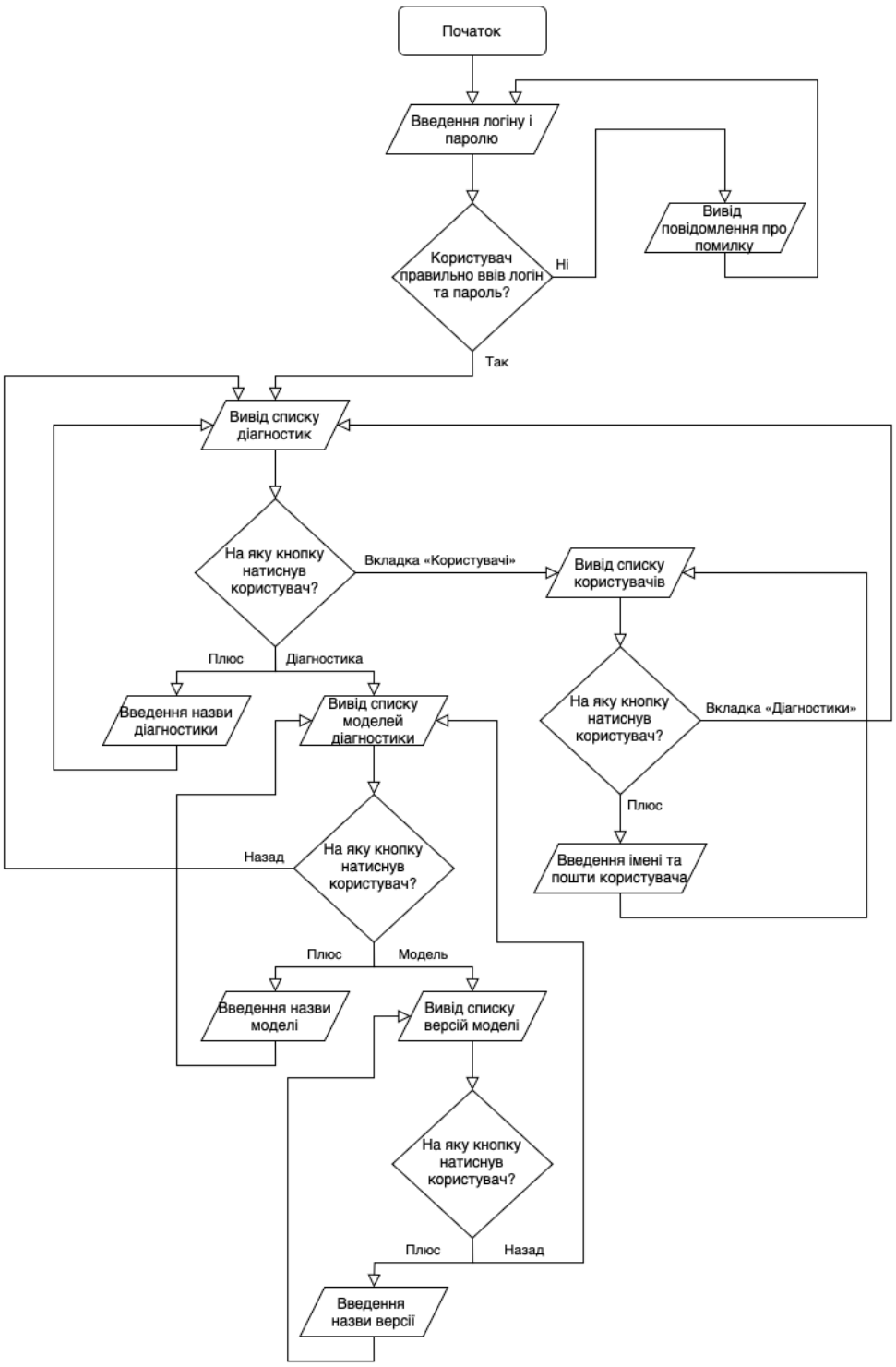
Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри

Алгоритм взаємодії адміністратора з системою (принципова схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ – 2024 р.



					ІАЛЦ.467200.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Пащенко Д. О.			Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри Алгоритм взаємодії адміністратора з системою (принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.					
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-04	
Н. Контр.		Волокита А. М.					
Затв.							

ДОДАТОК 2

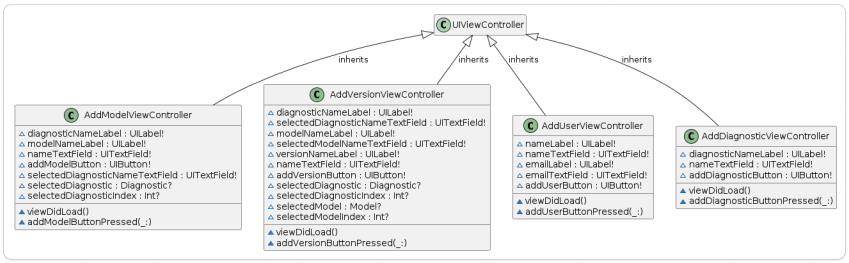
Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри

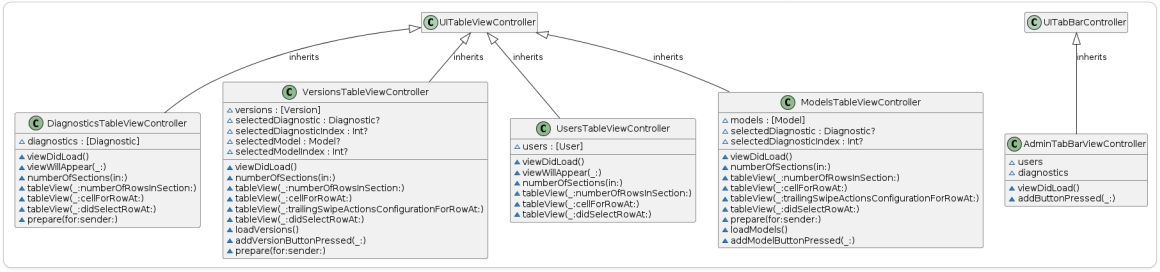
Діаграма класів (функціональна схема)

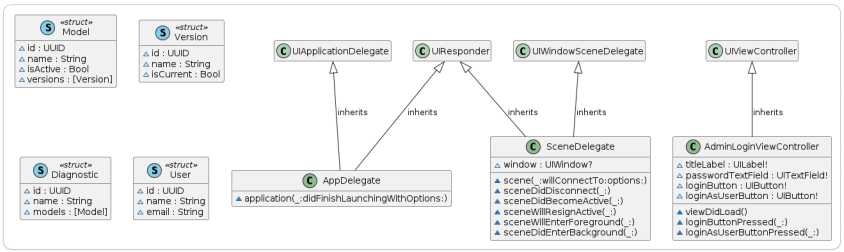
ІАЛЦ.467200.005 Д2

Аркушів 1

Київ – 2024 р.







					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Пащенко Д. О.			Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри Діаграма класів (функціональна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.					
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-04	
Н. Контр.		Волокита А. М.					
Затв.							

ДОДАТОК 3

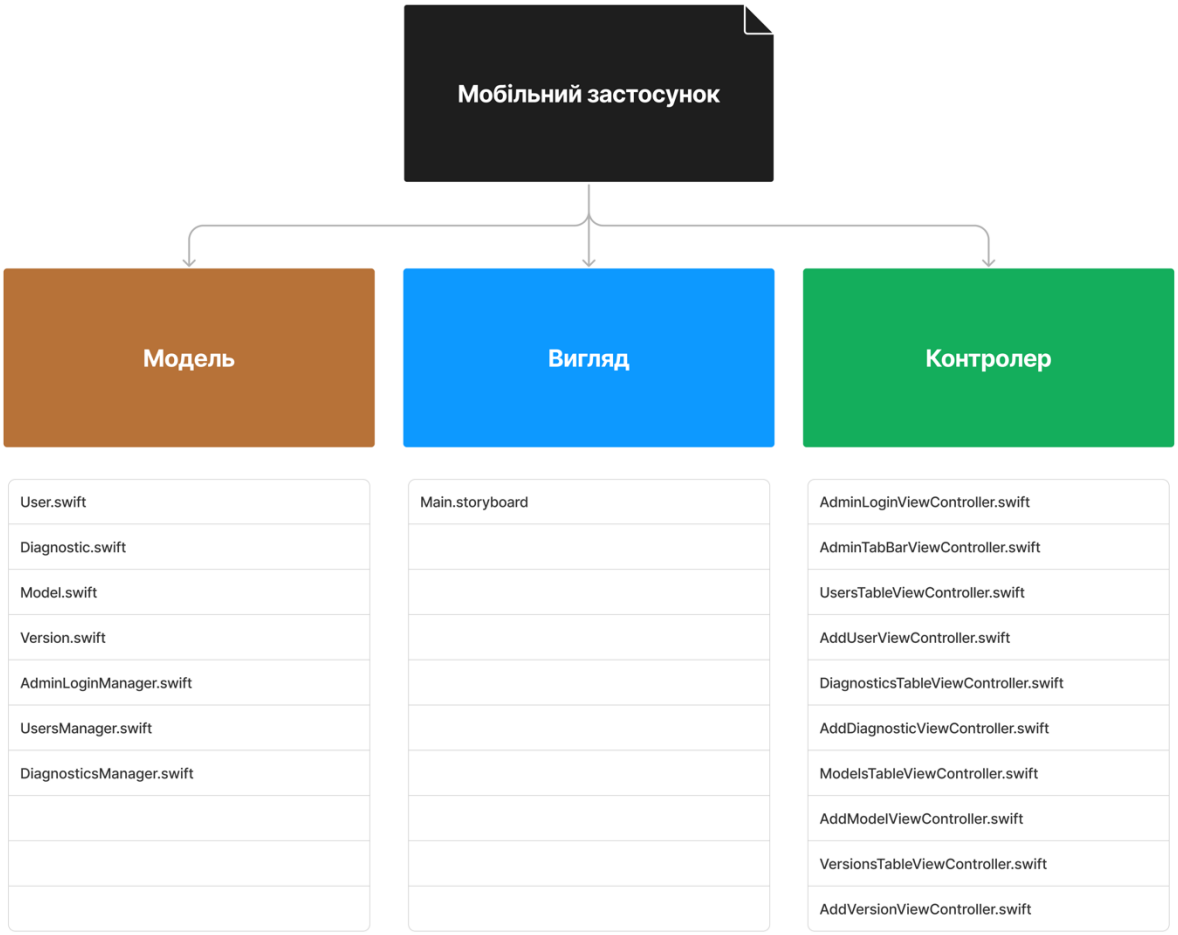
Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри

Архітектурна структура мобільного застосунку (структурна схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ – 2024 р.



					ІАЛЦ.467200.006 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Пащенко Д. О.			Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри Архітектурна структура мобільного застосунку (структурна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.				
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-04
Н. Контр.		Волокита А. М.				
Затв.						

ДОДАТОК 4

Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 21

Київ – 2024 р.

AdminLoginViewController.swift

```swift
import UIKit

class AdminLoginViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var passwordTextField: UITextField!
    @IBOutlet weak var loginButton: UIButton!
    @IBOutlet weak var loginAsUserButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

        navigationItem.hidesBackButton = true

        titleLabel.text = String(localized: "Login to the administration
system")
        passwordTextField.placeholder = String(localized: "Password")
        loginButton.setTitle(String(localized: "Login"), for: .normal)
        loginAsUserButton.setTitle(String(localized: "Login as user"),
for: .normal)
    }

    func authenticateAdmin(username: String, password: String) {
        AdminLoginManager.shared.authenticateAdmin(username: username,
password: password) { result in
            switch result {
            case .success(let token):
                UserDefaults.standard.setValue(token, forKey: "Token")
                print("Auth token: \(token)")
                self.performSegue(withIdentifier: "loginAsAdminToTabBar",
sender: self)

            case .failure(let error):
                print("Authentication failed: \(error)")
                let alert = UIAlertController(title: "Помилка", message:
"Виникла помилка під час аутентифікації. Спробуйте знову.",
preferredStyle: .alert)
                alert.addAction(UIAlertAction(title:
NSLocalizedString("OK", comment: "Default action"), style: .default,
handler: { _ in
                    NSLog("The \"OK\" alert occured.")
                }))
                self.present(alert, animated: true, completion: nil)
            }
        }
    }
```

					ІАЛЦ.467200.007 Д4		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Пащенко Д. О.			Мобільний застосунок для адміністрування системи для діагностики захворювань шкіри Текст програмного коду	Літ.	Аркуш 1 / Аркушів 21
Перевірив		Ковальчук О. М.					
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-04	
Н. Контр.		Волокита А. М.					
Затв.							

```
    @IBAction func loginButtonPressed(_ sender: UIButton) {
        authenticateAdmin(username: "", password: "")
    }

    @IBAction func loginAsUserButtonPressed(_ sender: UIButton) {
        navigationController?.popViewController(animated: true)
    }
}
```

AdminTabBarViewController.swift

```
import UIKit

class AdminTabBarViewController: UITabBarController {
    var users = [User(id: UUID(), name: "Дмитро", email:
"dmytro@gmail.com"),
                 User(id: UUID(), name: "Любов", email:
"liubov@gmail.com"),
                 User(id: UUID(), name: "Анастасія", email:
"anastasy@gmail.com"),
                 User(id: UUID(), name: "Кирило", email:
"kyrylo22@gmail.com"),
                 User(id: UUID(), name: "Данило", email:
"danilk@gmail.com"),
                 User(id: UUID(), name: "Максим", email:
"max20@gmail.com"),
                 User(id: UUID(), name: "Михайло", email:
"mihalo1@gmail.com"),
                 User(id: UUID(), name: "Іван", email: "vano8@gmail.com"),
                 User(id: UUID(), name: "Тетяна", email:
"tetiana@gmail.com"),
                 User(id: UUID(), name: "Тарас", email:
"tapas10@gmail.com"),]

    var diagnostics = [Diagnostic(id: UUID(), name: "Діагностування
псоріазу" , models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
висівкоподібного (різноколірного) лишаю", models:
                                  [Model(id: UUID(), name: "Модель 1",
isActive: true, versions:
                                         [Version(id: UUID(), name:
"Версія 1.2", isCurrent: true),
                                          Version(id: UUID(), name:
"Версія 1.1", isCurrent: false)]),
                                   Model(id: UUID(), name: "Модель 2",
isActive: false, versions: [])]),
                       Diagnostic(id: UUID(), name: "Діагностування
кропив'янки", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
піодермії", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
імпетиго", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
алергічного дерматиту", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
нейродерміту", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
екземи", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
карбункулу", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
фурункульозу", models: []),
                       Diagnostic(id: UUID(), name: "Діагностування
корости", models: []),
```

```
                        Diagnostic(id: UUID(), name: "Діагностування
розацеї", models: []),
                        Diagnostic(id: UUID(), name: "Діагностування
сикозу", models: []),
                        Diagnostic(id: UUID(), name: "Діагностування
хвороби Шамберга", models: []),
                        Diagnostic(id: UUID(), name: "Діагностування
меланоми", models: []),
                        Diagnostic(id: UUID(), name: "Діагностування
хвороби Лайма", models: [])]

    override func viewDidLoad() {
        super.viewDidLoad()

        navigationItem.hidesBackButton = true

        tabBar.items![0].title = String(localized: "Users")
        tabBar.items![1].title = String(localized: "Diagnostics")
    }

    @IBAction func addButtonPressed(_ sender: UIBarButtonItem) {
        let index = selectedIndex

        if index == 0 {
            performSegue(withIdentifier: "tabBarToAddUser", sender: self)
        }

        if index == 1 {
            performSegue(withIdentifier: "tabBarToAddDiagnostic", sender:
self)
        }
    }
}
```

UsersTableViewController.swift

```
import UIKit

class UsersTableViewController: UITableViewController {
    var users: [User] = []

    override func viewDidLoad() {
        super.viewDidLoad()

        if let tabBarController = self.tabBarController as?
AdminTabBarViewController {
            self.users = tabBarController.users
        }
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        self.tabBarController?.title = String(localized: "Users")
    }

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return users.count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"userReuseCell", for: indexPath)

        var configuration = cell.defaultContentConfiguration()

        configuration.text = users[indexPath.row].name
        configuration.secondaryText = users[indexPath.row].email

        cell.contentConfiguration = configuration

        return cell
    }

    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        tableView.deselectRow(at: indexPath, animated: true)
    }
}
```

AddUserViewController.swift

```
import UIKit

class AddUserViewController: UIViewController {
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var emailLabel: UILabel!
    @IBOutlet weak var emailTextField: UITextField!
    @IBOutlet weak var addUserButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.title = String(localized: "Add New User")
        nameLabel.text = String(localized: "Name")
        emailLabel.text = String(localized: "Email")
        addUserButton.setTitle(String(localized: "Add user"), for:
.normal)
    }

    @IBAction func addUserButtonPressed(_ sender: Any) {
        if nameTextField.text == "" || emailTextField.text == "" {

        } else {
            let newUserId = UUID()
            let newUserName = nameTextField.text ?? ""
            let newUserEmail = emailTextField.text ?? ""

            let newUser = User(id: newUserId, name: newUserName, email:
newUserEmail)

            if let navigationController = self.navigationController {
                let viewControllers = navigationController.viewControllers
                let adminTabBarViewController =
viewControllers[viewControllers.count - 2] as! AdminTabBarViewController
                let usersTableViewController =
adminTabBarViewController.viewControllers![0] as! UsersTableViewController

                let previousCountOfUsers =
usersTableViewController.users.count

                // Update data source
                adminTabBarViewController.users.append(newUser)
                usersTableViewController.users.append(newUser)

                // Update table
                let newIndexPath = IndexPath(row: previousCountOfUsers,
section: 0)
                usersTableViewController.tableView.insertRows(at:
[newIndexPath], with: .fade)

                // Return
                navigationController.popViewController(animated: true)
            }
```

```
            }
        }
    }
```

DiagnosticsTableViewController.swift

```
import UIKit

class DiagnosticsTableViewController: UITableViewController {
    var diagnostics: [Diagnostic] = []

    override func viewDidLoad() {
        super.viewDidLoad()

        if let tabBarController = self.tabBarController as?
AdminTabBarViewController {
            self.diagnostics = tabBarController.diagnostics
        }
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        self.tabBarController?.title = String(localized: "Diagnostics")
    }

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return diagnostics.count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"diagnosticReuseCell", for: indexPath)

        var configuration = cell.defaultContentConfiguration()

        configuration.text = diagnostics[indexPath.row].name

        cell.contentConfiguration = configuration

        return cell
    }

    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        performSegue(withIdentifier: "diagnosticsToModels", sender: self)
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "diagnosticsToModels" {
```

```
            let modelsTableViewController = segue.destination as!
ModelsTableViewController

            if let indexPath = tableView.indexPathForSelectedRow {
                modelsTableViewController.selectedDiagnostic =
diagnostics[indexPath.row]
                modelsTableViewController.selectedDiagnosticIndex =
indexPath.row
            }
        }
    }
}
```

AddDiagnosticViewController.swift

```
import UIKit

class AddDiagnosticViewController: UIViewController {
    @IBOutlet weak var diagnosticNameLabel: UILabel!
    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var addDiagnosticButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

        diagnosticNameLabel.text = String(localized: "Diagnostic name")
        addDiagnosticButton.setTitle(String(localized: "Add diagnostic"),
for: .normal)
    }

    @IBAction func addDiagnosticButtonPressed(_ sender: UIButton) {
        let newDiagnosticId = UUID()
        let newDiagnosticName = nameTextField.text ?? ""

        let newDiagnostic = Diagnostic(id: newDiagnosticId, name:
newDiagnosticName, models: [])

        if let navigationController {
            let viewControllers = navigationController.viewControllers
            let adminTabBarViewController =
viewControllers[viewControllers.count - 2] as! AdminTabBarViewController
            let diagnosticsTableViewController =
adminTabBarViewController.viewControllers![1] as!
DiagnosticsTableViewController

            let previousCountOfDiagnostics =
diagnosticsTableViewController.diagnostics.count

            // Update data source
            adminTabBarViewController.diagnostics.append(newDiagnostic)

diagnosticsTableViewController.diagnostics.append(newDiagnostic)

            // Update table
            let newIndexPath = IndexPath(row: previousCountOfDiagnostics,
section: 0)
            diagnosticsTableViewController.tableView.insertRows(at:
[newIndexPath], with: .fade)

            // Return
            navigationController.popViewController(animated: true)
        }
    }
}
```

ModelsTableViewController.swift

```
import UIKit

class ModelsTableViewController: UITableViewController {
    var models: [Model] = []

    var selectedDiagnostic: Diagnostic? {
        didSet {
            loadModels()
            self.title = selectedDiagnostic?.name
        }
    }
    var selectedDiagnosticIndex: Int?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return max(1, models.count)
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"modelReuseCell", for: indexPath)

        var configuration = cell.defaultContentConfiguration()

        if !models.isEmpty {
            let model = models[indexPath.row]

            configuration.text = model.name
            if model.isActive {
                configuration.text! += String(localized: " — active")
                configuration.textProperties.color = .color
            }
            cell.accessoryType = .disclosureIndicator
        } else {
            configuration.text = String(localized: "No models")
            cell.accessoryType = .none
        }

        cell.contentConfiguration = configuration

        return cell
    }
```

```
    override func tableView(_ tableView: UITableView,
trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) ->
UISwipeActionsConfiguration? {
        let makeActive = UIContextualAction(style: .normal, title:
String(localized: "Make active")) { (action, view, completionHandler) in
            if let cell = tableView.cellForRow(at: indexPath) {
                var configuration = cell.defaultContentConfiguration()

                let model = self.models[indexPath.row]

                configuration.text = "\(model.name)\(String(localized: " -
active"))"
                configuration.textProperties.color = .color

                cell.contentConfiguration = configuration
            }

            if let navigationController = self.navigationController {
                let viewControllers = navigationController.viewControllers
                let adminTabBarViewController = viewControllers[2] as!
AdminTabBarViewController
                let diagnosticsTableViewController =
adminTabBarViewController.viewControllers![1] as!
DiagnosticsTableViewController

                // Update data source
                /// 1) Make model active

adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s[indexPath.row].isActive = true

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models[indexPath.row].isActive = true
                self.models[indexPath.row].isActive = true

                /// 2) Make all other models inactive
                for index in self.models.indices {
                    if index != indexPath.row {

adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s[index].isActive = false

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models[index].isActive = false
                        self.models[index].isActive = false
                    }
                }

                // Update table
                var indexPaths: [IndexPath] = []

                for index in self.models.indices {
                    if index != indexPath.row {
                        let indexPath = IndexPath(row: index, section: 0)
                        indexPaths.append(indexPath)
```

```
                    }
                }

                tableView.reloadRows(at: indexPaths, with: .automatic)
            }

            completionHandler(true)
        }

        makeActive.backgroundColor = .color

        var actions: [UIContextualAction] = []
        if !models.isEmpty {
            actions.append(makeActive)
        }

        return UISwipeActionsConfiguration(actions: actions)
    }

    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        if !models.isEmpty {
            performSegue(withIdentifier: "modelsToVersions", sender: self)
        } else {
            tableView.deselectRow(at: indexPath, animated: true)
        }
    }

    // MARK: - Navigation

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "modelsToAddModel" {
            let addModelViewController = segue.destination as!
AddModelViewController

            addModelViewController.selectedDiagnostic = selectedDiagnostic
            addModelViewController.selectedDiagnosticIndex =
selectedDiagnosticIndex
        }

        if segue.identifier == "modelsToVersions" {
            let versionsTableViewController = segue.destination as!
VersionsTableViewController

            versionsTableViewController.selectedDiagnostic =
selectedDiagnostic

            if let indexPath = tableView.indexPathForSelectedRow {
                versionsTableViewController.selectedDiagnostic =
selectedDiagnostic
                versionsTableViewController.selectedDiagnosticIndex =
selectedDiagnosticIndex
                versionsTableViewController.selectedModel =
models[indexPath.row]
                versionsTableViewController.selectedModelIndex =
indexPath.row
```

```
                }
            }
        }

        func loadModels() {
            models = selectedDiagnostic!.models
        }

        @IBAction func addModelButtonPressed(_ sender: UIBarButtonItem) {
            performSegue(withIdentifier: "modelsToAddModel", sender: self)
        }
    }
```

AddModelViewController.swift

```
import UIKit

class AddModelViewController: UIViewController {
    @IBOutlet weak var diagnosticNameLabel: UILabel!
    @IBOutlet weak var modelNameLabel: UILabel!
    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var addModelButton: UIButton!
    @IBOutlet weak var selectedDiagnosticNameTextField: UITextField!

    var selectedDiagnostic: Diagnostic?
    var selectedDiagnosticIndex: Int?

    override func viewDidLoad() {
        super.viewDidLoad()

        selectedDiagnosticNameTextField.text = selectedDiagnostic?.name

        diagnosticNameLabel.text = String(localized: "Diagnostic name")
        modelNameLabel.text = String(localized: "Model name")
        addModelButton.setTitle(String(localized: "Add model"), for:
.normal)
    }

    @IBAction func addModelButtonPressed(_ sender: UIButton) {
        let newModelId = UUID()
        let newModelName = nameTextField.text ?? ""

        let newModel = Model(id: newModelId, name: newModelName, isActive:
false, versions: [])

        if let navigationController {
            let viewControllers = navigationController.viewControllers
            let adminTabBarViewController = viewControllers[2] as!
AdminTabBarViewController
            let diagnosticsTableViewController =
adminTabBarViewController.viewControllers![1] as!
DiagnosticsTableViewController
            let modelsTableViewController = viewControllers[3] as!
ModelsTableViewController

            let previousCountOfModels =
modelsTableViewController.models.count

            // Update data source

adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s.append(newModel)

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models.append(newModel)
            modelsTableViewController.models.append(newModel)

            // Update table
```

```
            if previousCountOfModels == 0 {
                modelsTableViewController.tableView.reloadData()
            } else {
                let newIndexPath = IndexPath(row: previousCountOfModels,
section: 0)
                modelsTableViewController.tableView.insertRows(at:
[newIndexPath], with: .fade)
            }

            // Return
            navigationController.popViewController(animated: true)
        }
    }
}
```

VersionsTableViewController.swift

```
import UIKit

class VersionsTableViewController: UITableViewController {
    var versions: [Version] = []

    var selectedDiagnostic: Diagnostic?
    var selectedDiagnosticIndex: Int?
    var selectedModel: Model? {
        didSet {
            loadVersions()
            self.title = selectedModel?.name
        }
    }
    var selectedModelIndex: Int?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return max(1, versions.count)
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"versionReuseCell", for: indexPath)

        var configuration = cell.defaultContentConfiguration()

        if !versions.isEmpty {
            let version = versions[indexPath.row]

            configuration.text = version.name
            if version.isCurrent {
                configuration.text! += String(localized: " — current")
                configuration.textProperties.color = .color
            }
        } else {
            configuration.text = String(localized: "No versions")
        }

        cell.contentConfiguration = configuration

        return cell
    }
```

```
    override func tableView(_ tableView: UITableView,
trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) ->
UISwipeActionsConfiguration? {
        let makeCurrent = UIContextualAction(style: .normal, title:
String(localized: "Make current")) { (action, view, completionHandler) in
            print(self.versions)

            if let cell = tableView.cellForRow(at: indexPath) {
                var configuration = cell.defaultContentConfiguration()

                let version = self.versions[indexPath.row]

                configuration.text = "\(version.name) \(String(localized: "
— current"))"
                configuration.textProperties.color = .color

                cell.contentConfiguration = configuration
            }

            if let navigationController = self.navigationController {
                let viewControllers = navigationController.viewControllers
                let adminTabBarViewController = viewControllers[2] as!
AdminTabBarViewController
                let diagnosticsTableViewController =
adminTabBarViewController.viewControllers![1] as!
DiagnosticsTableViewController
                let modelsTableViewController = viewControllers[3] as!
ModelsTableViewController

                // Update data source
                /// 1) Make version current

adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s[self.selectedModelIndex!].versions[indexPath.row].isCurrent = true

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models[self.selectedModelIndex!].versions[indexPath.row].isCurrent = true

modelsTableViewController.models[self.selectedModelIndex!].versions[indexP
ath.row].isCurrent = true
                self.versions[indexPath.row].isCurrent = true

                /// 2) Make all other versions not current
                for index in self.versions.indices {
                    if index != indexPath.row {

adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s[self.selectedModelIndex!].versions[index].isCurrent = false

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models[self.selectedModelIndex!].versions[index].isCurrent = false

modelsTableViewController.models[self.selectedModelIndex!].versions[index]
.isCurrent = false
                        self.versions[index].isCurrent = false
```

```
                    }
                }

                // Update table
                var indexPaths: [IndexPath] = []

                for index in self.versions.indices {
                    if index != indexPath.row {
                        let indexPath = IndexPath(row: index, section: 0)
                        indexPaths.append(indexPath)
                    }
                }

                tableView.reloadRows(at: indexPaths, with: .automatic)
            }

            completionHandler(true)
        }

        makeCurrent.backgroundColor = .color

        var actions: [UIContextualAction] = []
        if !versions.isEmpty {
            actions.append(makeCurrent)
        }

        return UISwipeActionsConfiguration(actions: actions)
    }

    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        tableView.deselectRow(at: indexPath, animated: true)
    }

    func loadVersions() {
        versions = selectedModel!.versions
    }

    @IBAction func addVersionButtonPressed(_ sender: UIBarButtonItem) {
        performSegue(withIdentifier: "versionsToAddVersion", sender: self)
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        let addVersionViewController = segue.destination as!
AddVersionViewController

        addVersionViewController.selectedDiagnostic = selectedDiagnostic
        addVersionViewController.selectedDiagnosticIndex =
selectedDiagnosticIndex
        addVersionViewController.selectedModel = selectedModel
        addVersionViewController.selectedModelIndex = selectedModelIndex
    }
}
```

AddVersionViewController.swift

```
import UIKit

class AddVersionViewController: UIViewController {
    @IBOutlet weak var diagnosticNameLabel: UILabel!
    @IBOutlet weak var selectedDiagnosticNameTextField: UITextField!
    @IBOutlet weak var modelNameLabel: UILabel!
    @IBOutlet weak var selectedModelNameTextField: UITextField!
    @IBOutlet weak var versionNameLabel: UILabel!
    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var addVersionButton: UIButton!

    var selectedDiagnostic: Diagnostic?
    var selectedDiagnosticIndex: Int?
    var selectedModel: Model?
    var selectedModelIndex: Int?

    override func viewDidLoad() {
        super.viewDidLoad()
        selectedDiagnosticNameTextField.text = selectedDiagnostic?.name
        selectedModelNameTextField.text = selectedModel?.name

        diagnosticNameLabel.text = String(localized: "Diagnostic name")
        modelNameLabel.text = String(localized: "Model name")
        versionNameLabel.text = String(localized: "Version name")
        addVersionButton.setTitle(String(localized: "Add version"), for:
.normal)
    }

    @IBAction func addVersionButtonPressed(_ sender: UIButton) {
        let newVersionId = UUID()
        let newVersionName = nameTextField.text ?? ""

        let newVersion = Version(id: newVersionId, name: newVersionName,
isCurrent: false)

        if let navigationController {
            let viewControllers = navigationController.viewControllers
            let adminTabBarViewController = viewControllers[2] as!
AdminTabBarViewController
            let diagnosticsTableViewController =
adminTabBarViewController.viewControllers![1] as!
DiagnosticsTableViewController
            let modelsTableViewController = viewControllers[3] as!
ModelsTableViewController
            let versionsTableViewController = viewControllers[4] as!
VersionsTableViewController

            let previousCountOfVersions =
versionsTableViewController.versions.count

            // Update data source
```

```
adminTabBarViewController.diagnostics[self.selectedDiagnosticIndex!].model
s[self.selectedModelIndex!].versions.append(newVersion)

diagnosticsTableViewController.diagnostics[self.selectedDiagnosticIndex!].
models[self.selectedModelIndex!].versions.append(newVersion)

modelsTableViewController.models[self.selectedModelIndex!].versions.append
(newVersion)
            versionsTableViewController.versions.append(newVersion)

            // Update table
            if previousCountOfVersions == 0 {
                versionsTableViewController.tableView.reloadData()
            } else {
                let newIndexPath = IndexPath(row: previousCountOfVersions,
section: 0)
                versionsTableViewController.tableView.insertRows(at:
[newIndexPath], with: .fade)
            }

            // Return
            navigationController.popViewController(animated: true)
        }
    }
}
```