Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

«Інтелектуальний моніторінг фізичної активності користувача пристроями носимої електроніки»
(тема)

Виконав:
студент 2 курсу, групи СШІм-18-3
Суворов Р. С.
(прізвище, ініціали)
Спеціальність 122 – Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва спеціалізації)

Керівник професор, д.т.н. Терзіян В.Я.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ В.О. Філатов
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 – Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Суворов Роману Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Інтелектуальний моніторинг фізичної активності користувача пристроями носимої електроніки
затверджена наказом університету від 30 03 2020 р. № 480Ст
2. Термін подання студентом роботи до екзаменаційної комісії 21 05 2020 р.
3. Вихідні дані до роботи Зібрані дані щодо активності користувача з сенсорів пристрою носимої електроніки
4. Перелік питань, що потрібно опрацювати в роботі Аналіз предметної області та технологій Вибір платформи для вирішення проблеми Доступні технології для вирішення проблеми Постановка задачі Рішення проблеми Багатошарова архітектура Проектування бізнес-логічного шару CLEAN та SOLID архітектури Функціонально-реактивне програмування з Swift UI та Combine On-Device Machine Learning Неоморфічний дизайн та Swift UI Практичне дослідження

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)_____

_____
_____
_____
_____
_____
_____
_____
_____
_____

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Аналіз літератури	проф.Терзіян Я. В.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз літератури	30.03.2020 – 05.04.2020	виконано
2	Вибір методу дослідження	06.04.2020 – 15.04.2020	виконано
3	Вибір відповідної платформи розробки	16.04.2020 – 17.04.2020	виконано
4	Огляд наявного програмного забезпечення	18.04.2020 – 19.04.2020	виконано
5	Проектування архітектури рішення	20.04.2020 – 25.04.2020	виконано
6	Вибір та попередня обробка даних	28.04.2020 – 05.05.2020	виконано
7	Розробка програмного забезпечення	20.04.2020 – 20.05.2020	виконано
8	Виконання, аналіз та документування	10.04.2020 – 21.05.2020	виконано
9	Написання дипломної роботи	15.04.2020 - 21.05.2020	виконано

Дата видачі завдання 30 03 2020 р.

Студент _____
(підпис)

Керівник роботи _____ професор, д.т.н. Терзіян В.Я.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 109 с., 25 рис., 20 джерел.

SWIFT, WATCHOS, COREML, CREATEML, TENSORFLOW, TURIML, PYHON, SWIFTUI

Об'єкт дослідження – Використання машинного навчання на сучасних пристроях носимої електроніки для відстеження фізичної активності користувача

Предмет дослідження – Core ML, Create ML, TensorFlow, TuriML, що дозволять створити модель та оновлювати її в процесі користування додатком користувачем

Мета роботи – знайти рішення, що дозволить замість використання натренованої під певну кількість типів вправ модель, використовувати модель котру можна доповнювати активністю користувача, додаючи тим самим нові, не відомі раніше типи активності.

Методи дослідження – теоретичні (пошук інформації в мережі) та практичні (створення власного додатку для реалізації рішення проблеми).

РЕФЕРАТ

Записка пояснительная: 109 с., 25 рис., 20 источников.

SWIFT, WATCHOS, COREML, CREATEML, TENSORFLOW, TURIML, PYHON, SWIFTUI

Объект исследования – Использование машинного обучения на современных устройствах носимой электроники для отслеживания физической активности пользователя

Предмет исследования – Core ML, Create ML, TensorFlow, TuriML, которые позволят создать и обновлять модель в процессе ииспользования устройства пользователями

Цель работы – найти решение, которое позволит вместо заранее определенного под определённые типы активностей алгоритма или модели, использовать модель, которую можно было бы постоянно обучать и дополнять, добавляя ранее неизвестные виды активностей

Методы исследования – теоретические (поиск информации в сети) и практические (создание собственного приложения для реализации решения проблемы).

ABSTRACT

Explanatory note: 109 p., 25 im., 20 src.

SWIFT, WATCHOS, COREML, CREATEML, TENSORFLOW, TURIML, PYHON, SWIFTUI

Object of study - The use of machine learning on modern wearable electronics devices to track the user's physical activity

Research subject - Core ML, Create ML, TensorFlow, TuriML, which will allow you to create and update a model in the process and use of the device by users

The purpose of the work is to find a solution that, instead of a predetermined type of activity for an algorithm or model, can use a model that could be constantly trained and supplemented by adding previously unknown types of activities

Research methods - theoretical (searching for information on the network) and practical (creating your own application to implement a solution to the problem).

-

ЗМІСТ

Перелік термінів та скорочень …… 6
Вступ …… 7
1 Аналіз предметної області та технологій …… 9
1.1 Вибір платформи для вирішення проблеми …… 9
1.2 Доступні технології для вирішення проблеми …… 12
2 Постановка задачі …… 26
3 Рішення проблеми …… 27
3.1 Багатошарова архітектура …… 27
3.2 Проектування бізнес-логічного шару …….. 31
3.3 CLEAN та SOLID архітектури …….. 35
3.4 Функціонально-реактивне програмування з Swift UI та Combine …….. 41
3.5 On-Device Machine Learning …… 49
4 Неоморфічний дизайн та Swift UI …….. 63
5 Практичне дослідження ……... 102
Висновки …….. 104
Додаток А …… 107
Додаток Б …….. 108

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

ЧСС – Частота серцевих скорочень (пульс);

ARM – Advanced Reduced Instruction Set Computing Machine;

Health KIT – Набір інструментів Apple для доступу до показників здоров`я користувача;

BNNS – Basic Neural Network Subroutines;

Core ML – Інструмент Apple для роботи з моделями;

Metal – Графічний API з низьким рівнем доступа до GPU, високою продуктивністю;

Public API – Інтерфейс системних класів доступний розробнику;

Private API – Інтерфейс системних класів недоступний розробнику;

Swift – Нова сучасна мова програмування від Apple, що забезпечує високу швидкість операцій;

Swift UI – Інструмент Apple для роботи з графічним інтерфейсом в реактивно-функціональному підході;

watchOS – операційна система на базі iOS для Apple Watch;

Watch KIT – Набір інструментів Apple для створення додатку watchOS;

WWDC – World Wide Developers Conference.

ВСТУП

На сьогодні, машинне навчання використовується майже всюди – в телевізорах та моніторах, для покращення якості зображення та звуку, для корегування тексту, що вводить користувач з клавіатури та вивчення нових слів, сленгу, в холодильниках, пральних машинах, в мережевому обладнанні, для покращення якості маршрутизації пакетів, для компресії аудіо та відео потоків, для аналізу фотографій у бібліотеках користувачів, для актуалізації контенту, що надають сервіси такі як Netflix, YouTube, Crunchyroll, навіть у системах клімат контролю, та звичайно ж, у пристроях носимої електроніки – як дешевих і доступних (наприклад у Xiaomi Mi Band, Amazefit Bip), так і в дорогих, як Apple Watch чи Garmin, для покращення точності визначення кількості кроків та частоти серцевого ритму (пульсу) користувача, а також для моніторингу його сну та активності.

Але якщо у випадку з Netflix чи YouTube вподобання та особливості користувача відразу ж коригують результати пошуку, тобто, впливають на результат без необхідності оновлення моделі та додатку в цілому, то у випадку з пристроями носимої електроніки ці до сих пір не так – для покращення роботи алгоритму, що коригує показники, що надаються з сенсорів пристрою (гіроскопу, акселерометру, сенсору частоти серцевих скорочень, барометра, магнітометра, сенсору освітленості та гучності), необхідно проводити оновлення пристрою для оновлення моделі або самого алгоритму кожен раз, як буде зібрано достатню кількість даних та перетренована спільна для всіх модель.

У 2018му році, компанія Apple створила набір інструментів для створення та тренування моделі відразу на пристроях, без необхідності оновлення додатку чи прошивки, а в 2019му році ці інструменти стали доступними на усіх пристроях носимої електроніки що випускає компанія

Apple, починаючи зі старої моделі Apple Watch Series 2, що і дозволить нам вирішити одразу декілька проблем, що досі актуальні для користувачів, а саме те, що будь-які алгоритми, наприклад, для відстежування фізичної активності користувача та додатки для тренувань, використовують моделі загального призначення, тобто ті, котрі натреновані на даних усіх користувачів, а не на даних конкретної людини, тому вони слабо враховують можливі особливості конкретною людини, або не враховують їх взагалі, що значно знижує точність роботи таких алгоритмів, а що ще гірше – не дають можливості користувачу проаналізувати певні нестандартні активності та активності що не були передбачені виробником пристрою або розробником програми.

Так, наприклад, можливості сучасних пристроїв носимої електроніки щодо тренінгу активності користувача значно обмежені стандартними категоріями – бігу, їзді на велосипеді, катанню на сноуборді та лижах, що робить неможливим для таких пристроїв обробку нестандартних активностей, наприклад, гри у футбол, теніс, дайвінгу, дуже часто, навіть тренування у спортивних залах чи вдома, а якщо програма чи пристрій і спроможні якось обробляти таки тренування, вони потребують ручного вводу даних, щодо вправ, кількості виконань та підходів, перерв, і це для кожної вправи, котрих у одному тренування може бути десятки, при тому кожен день ці набори будуть різними. Також у сучасних пристроях та додатках немає можливості постійної актуалізації моделі та можливих наборів вправ / активностей, що також додає незручності користувачеві і загалом робить такі рішення не дуже життєздатними. Було б дуже добре це виправити.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕХНОЛОГІЙ

1.1 Вибір платформи для вирішення проблеми

Для вирішення даної проблеми я використаю засоби watchOS – це операційна система Apple Watch на основі iOS, котра в свою чергу є версією macOS для мобільних пристроїв на базі ARM.

На це є декілька причин:

- watchOS є повноцінною операційною системою зі значним набором інструментів для розробника програмного забезпечення;
- Apple Watch це вже повністю готовий пристрій носимої електроніки;
- Apple Watch є найбільш розповсюдженим пристроям носимої електроніки, що має можливість встановлення та запуску додатків та надає доступ до усіх необхідних сенсорів, має можливість інтеграції додатку із штатною системою Здоров`я (Health) через Health KIT, та з режимом тренування, а також сфокусований на фізичній активності користувача та безпечності тренувань, а також на стані здоров`я користувача.

Також значним плюсом щодо мого рішення є те, що Apple Watch є дуже продуманою системою, котра не лише аналізує та зберігає активності користувача, але й заохочує його робити більше, бути більш активним, рухатись, тренуватись, досягати більших результатів, що є важливою основою для фітнес-трекерів та інших пристроїв носимої електроніки, адже більшість з них більшістю користувачів згодом вже не використовується саме як фітнес-пристрій та для покращення фізичного стану користувача, а лише як модний аксесуар чи годинник, вважаю що цього недостатньо. Apple Watch став справжнім хітом продажів. Аналітики вважають, що Apple вдалося продати 3 мільйони пристроїв і зайняти 75% ринку розумних годин, а ще отримати 1 млрд виручки. За деякими оцінками, ці цифри вище, ніж

при виході iPhone і iPad. Але дещо вражає найбільше - 97% покупців задоволені за недостатністю майна в захваті від пристрою, а 86% носять Apple Watch щодня. Це робить їх безумовним переможцем в категорії носяться гаджетів, які за іронією надягають рідко. Більше половини покупців перестають користуватися аналогічними пристроями протягом шести місяців після покупки. І хоча преса не шанує Apple Watch, коли мова заходить про їх можливості в якості фітнес-трекера, годинник впливають на поведінку тих, хто їх носить. За даними опитування Wristly (дослідницької групи, яка вивчає і лобіює переносну електроніку), 75% відсотків власників Watch стверджують, що пристрій допомагає їм частіше рухатися, 59% роблять більш здоровий вибір (що б це не означало), а 57% відзначають, що вони стали частіше тренуватися.

Жоден інший пристрій не задовольняє усім цим критеріям, а отже не може бути використаний як платформа для вирішення проблеми із максимальною ефективністю. Також, щотижневі підсумкові показники мотивують людей. Решта пристрої зосереджені націлені на одне - кроки і калорії. Така простота привертає і одночасно вводить в оману. Скажи мені, що спалюєш більше калорій днем, і я тут же спершу, як і коли ти їх спалюєш? Watch відстежують відразу три показника: рух, час, проведений стоячи, і витрачений на вправи час. Це більше, ніж просто кількість калорій, але і не панацея. Зазначу, Watch - це не просто трекер активності. Це ще й ком'юніті людей, які підтримують тебе. Бачити таблицю лідерів Nike + або чути в навушниках підбадьорювання друзів так само важливо, як і знати, скільки і як швидко ти пробіг. Це як побачити правильний твіт в потрібний час. Це мотивує і відволікає тебе від думки закінчити пробіжку і піти пити пиво. Велика кількість даних є доказом, що навіть атлети повинні думати про активний спосіб життя. Недавнє дослідження в журналі Annals of Internal Medicine доводить, що сидячий спосіб життя пов'язаний з проблемами зі здоров'ям навіть якщо ви займаєтеся спортом.

На мій подив, дуже мало фітнес-трекерів були створені з розумінням цієї проблеми. Багато з них запропонують вам встановити денну мета і допоможуть досягти її. Це круто. Але уяви, ти вийшов на ранкову пробіжку і вже до обіду досяг потрібної кількості кроків. Твоє пристрій не стане тобі нагадувати, що потрібно продовжувати рухатися. З Apple Watch все інакше. Ти не зможеш досягти мети Stand вже вранці, оскільки додаток нагадує тобі, що потрібно встати один раз на годину. Watch пропонує спортсменам не зациклюватися тільки на заняттях, але також займатися фітнесом протягом дня.

Існує багато видів спорту, дані про яких не можна трактувати однозначно. Це, наприклад, теніс, баскетбол або бейсбол. Якщо ми зможемо виміряти специфічні показники, це допоможе нам говорити не тільки про кількість спалених калорій, але і попереджати травми. Можна отримувати дані з різних пристроїв. Сьогодні багато людей купують ваги, щоб схуднути. Ці ж ваги можуть допомогти велосипедисту набрати форму до гонки. Хтось може придумати додаток для таких ваг або для велосипедних аксесуарів PowerTap. Споживачі буквально одержимі всякими пристроями, які можуть виміряти їх показники активності. Що мені подобається в Apple, так це бажання допомагати людям. Ми створюємо цілі платформи. І Apple Watch - це теж платформа для сторонніх додатків. Вони можуть реалізувати задумки, про які ми навіть не замислювалися. Людський організм - неймовірно складна штука. І немає сенсора або пристрою, який завжди буде точно визначати всі показники. Тобі потрібно копнути глибше і змусити людей бігати або сісти на справжні велосипеди. А всі отримані дані тільки доводять, що нам належить ще багато чого довідатися про фітнесі.

1.2 Доступні технології для вирішення проблеми

В останні кілька років тема штучного інтелекту і машинного навчання перестала бути для людей чимось з області фантастики і міцно увійшла в повсякденне життя. Соціальні мережі пропонують відвідати цікаві нам заходи, автомобілі на дорогах навчилися пересуватися без участі водія, а голосовий помічник в телефоні підказує, коли краще виходити з дому, щоб уникнути пробок, і чи потрібно брати з собою парасольку. Ми розглянемо інструменти для машинного навчання, які пропонує розробникам Apple, розберемо, що нового в цій області компанія показала на WWDC18 [3], і спробуємо зрозуміти, як можна застосувати це все на практиці.

Отже, машинне навчання - це процес, в ході якого система, використовуючи певні алгоритми аналізу даних і обробляючи величезну кількість прикладів, виявляє закономірності і використовує їх, щоб прогнозувати характеристики нових даних. Машинне навчання народилося з теорії про те, що комп'ютери можуть вчитися самостійно, будучи ще не запрограмованими на виконання певних дій. Іншими словами, на відміну від звичайних програм із заздалегідь введеними інструкціями для вирішення конкретних завдань, машинне навчання дозволяє системі навчитися самостійно розпізнавати шаблони і робити прогнози. До watchOS 6 створення такого додатку через Public API було неможливим через обмеження самої системи – ми не мали можливості створити автономний додаток, що працював би на Apple Watch не як extension до основного додатку на iPhone, а повністю незалежно, не мали можливості отримувати данні з сенсорів, лише те, що нам надає система, вже готову кількість кроків та частоту серцевих скорочень, та багато іншого. На WWDC 17 Apple представила новий фреймворк для роботи з технологіями машинного навчання Core ML. На основі нього в iOS реалізовані власні продукти Apple: Siri, Camera і QuickType. Core ML дозволяє спростити інтеграцію

машинного навчання в додатки і створювати різні «розумні» функції за допомогою пари рядків коду.

За допомогою watchOS 6 SDK [5] ви можете створити абсолютно незалежний додаток watchOS або навіть додаток watchOS без аналога на iOS. Ви можете створити більш гнучкі та динамічні користувацькі інтерфейси з SwiftUI та написати сучасний код обробки подій за допомогою Combine. Ви також можете включити нові функції, такі як Увійти за допомогою Apple, розширені сеанси виконання та потокове передавання аудіо у ваші програми. За допомогою watchOS 6 ви можете створювати додатки, доступні лише для Apple Watch, без супутнього додатка на iOS. Користувачі можуть навіть придбати ваші програми watchOS безпосередньо у магазині додатків на Apple Watch.

SwiftUI - це сучасний підхід до створення інтерфейсів користувача для iOS, macOS, watchOS та tvOS. Ви можете створювати динамічні інтерфейси швидше, ніж будь-коли раніше, використовуючи декларативні програмування на основі композиції. Рамка містить структури представлень, елементів керування та компонування для оголошення користувальницького інтерфейсу програми. Він також надає обробники подій для доставки натискань, жестів та інших типів введення у вашу програму та інструменти для управління потоком даних із моделей вашого додатка аж до представлень даних та елементів управління, з якими користувачі побачать та взаємодіють.

Тепер ваші програми можуть отримати тривалий час для доступу до даних сенсорів та завершення сеансів для таких випадків, як розумність, фізична терапія та тривожність. Для отримання додаткової інформації дивіться розділ Розширені сесії виконання в рамковій документації WatchKit.

Combine - це нова рамка, яка забезпечує декларативний інтерфейс Swift для обробки значень у часі. Ці значення можуть представляти події інтерфейсу користувача, відповіді мережі, заплановані події та багато інших видів асинхронних даних. За допомогою комбінації ви заявляєте про видавців, які викривають значення, які можуть змінюватися, та передплатників, які отримують ці значення від видавців. Комбінування полегшує читання та підтримку коду, централізуючи код обробки подій та усуваючи проблемні методи, такі як вкладені закриття та зворотні виклики на основі конвенції.

Apple використовує технології машинного навчання на своїх пристроях вже досить давно: Mail визначає листи, що містять спам, Siri допомагає швидко дізнатися відповіді на питання, що цікавлять, Photos розпізнає обличчя на зображеннях.

На WWDC16 [3] компанія представила два API на базі нейронних мереж - Basic Neural Network Subroutines (BNNS) і Convolutional Neural Network (CNN). BNNS - частина системи Accelerate, яка є основою для виконання швидких обчислень на CPU, а CNN - бібліотеки Metal Performance Shaders, що використовує GPU.

Apple анонсувала фреймворк, значно облягаючий роботу з технологіями машинного навчання - Core ML. В його основі ідея про те, щоб взяти заздалегідь навчену модель даних, і буквально в кілька рядків коду інтегрувати її в свій додаток.

За допомогою Core ML в додатку можна реалізувати наступні функції:

- розпізнавання зображень в реальному часі;
- інтелектуальне введення тексту;
- розпізнавання образів;
- аналіз тональності;
- розпізнавання рукописного тексту;
- ранжування пошуку;
- стилізація зображень;
- розпізнавання осіб;
- ідентифікація голосу;
- визначення музики;
- реферування тексту;
- та багато іншого.

Core ML дозволяє легко імпортувати в вашу програму різні алгоритми машинного навчання, такі як: tree ensembles, SVMs і generalized linear models. Він використовує низько рівневі технології, такі як Metal, Accelerate і BNNS. Результати обчислень відбуваються майже миттєво. Ядро підтримує нейронні мережі, узагальнені лінійні моделі (generalized linear models), генерацію ознак (feature engineering), деревовидні алгоритми прийняття рішень (tree ensembles), метод опорних векторів (support vector machines), пайплайн моделі (pipeline models). Але Apple з самого початку не показала власних технологій для створення і навчання моделей, а лише зробила конвертер для інших популярних фреймворків: Caffe, Keras, scikit-learn, XGBoost, LIBSVM. Використання сторонніх інструментів часто було не найпростішим завданням, навчені моделі мали досить великий обсяг, а саме навчання займало чимало часу. В кінці року компанія представила Turi Create - фреймворк для навчання моделей, основною ідеєю якого була простота у використанні і підтримка великого числа сценаріїв - класифікація зображень, визначення об'єктів, рекомендаційні системи, і безліч інших.

Але Turi Create, незважаючи на свою відносну простоту у використанні, підтримував тільки Python. І ось в 2018 році Apple, крім Core ML 2, нарешті показала власний інструмент для навчання моделей - фреймворк Create ML, який використовує нативні технології Apple - Xcode і Swift. Він працює швидко, а створення типових моделей за допомогою Create ML стає по-справжньому простим. На WWDC були озвучені вражаючі показники Create ML і Core ML 2 на прикладі додатка Memrise. Якщо раніше на навчання однієї моделі з використанням 20 тисяч зображень потрібно 24 години, то Create ML скорочує цей час до 48 хвилин на MacBook Pro і до 18 хвилин - на iMac Pro. Розмір навченої моделі зменшився з 90MB до 3MB. Create ML дозволяє використовувати в якості вихідних даних зображення, тексти і структуровані об'єкти - наприклад, таблиці.

Фреймворк Vision працює на основі Core ML і допомагає з відстеженням і розпізнаванням осіб, тексту, об'єктів, штрих-кодів. Також є визначення горизонту і отримання матриці для вирівнювання зображення. Починаючи з iOS 5 Apple представила NSLinguisticTagger, який дозволяє аналізувати природну мову, підтримує безліч мов і алфавітів. З виходом iOS 11 клас вдосконалили, тепер йому можна згодувати рядок з текстом на різних мовах і він поверне домінуюча мова в цьому рядку і ще багато інших поліпшень. NSLinguisticTagger теж використовує машинне навчання для глибокого розуміння тексту і його аналізу. На промо сторінці Core ML Apple надала 4 моделі. Всі вони аналізують зображення. Моделі Core ML працюють локально і оптимізовані для роботи на мобільних пристроях, зводячи до мінімуму обсяг використовуваної пам'яті і енергоспоживання.
Ви можете згенерувати власні моделі за допомогою Core ML Tools.

Робочий спосіб завантажувати моделі під час виконання:

- покласти файл моделі в таргет додатка;
- скомпілювати нову модель з .mlmodel в .mlmodelc, не змінюючи її інтерфейс;
- покласти ці файли на сервер;
- завантажити їх всередині програми;
- ініціалізувати нову модель, наприклад: CoreMLModelClass.init (contentOf: URL).

Завданням Create ML, у моєму випадку, буде побудова моделі, що у якості входу використовує таблицю з даними із сенсорів. Таке завдання в машинному навчанні називається завданням регресії і є класичним прикладом навчання з учителем/ Create ML підтримує безліч моделей - Linear Regression, Decision Tree Regression, Tree Classifier, Logistic Regression, Random Forest Classifier, Boosted Trees Regression, і ін. [1]

Ми скористаємося об'єктом MLRegressor, який на основі вхідних даних сам підбере оптимальний варіант. Спочатку ініціалізуємо об'єкт MLDataTable вмістом нашого csv-файлу, як у прикладі 1.1, нижче:

```
let trainingFile = URL (fileURLWithPath:
"/Users/CreateMLTest/Apartments.csv")
let apartmentsData = try MLDataTable (contentsOf:
trainingFile)
```

Приклад 1.1 – Ініціалізація MLDataTable

Поділяємо вихідний набір даних на дані для навчання моделі і тестування в процентному співвідношенні 80/20, як це вказано у прикладі 1.2, нижче:

```
    let (trainingData, testData) =
apartmentsData.randomSplit (by: .8, seed: 0)
```

Приклад 1.2 – Поділ даних для навчання моделі у співвідношенні 8:2

Створюємо модель MLRegressor, вказуючи дані для навчання і назва колонки, значення якої хочемо передбачати. Специфічний для завдання тип регресорів (linear, decision tree, boosted tree або random forest) буде обраний автоматично на основі дослідження вхідних даних. Також ми можемо вказати feature columns - конкретні колонки-параметри для аналізу, але в даному прикладі в цьому немає необхідності, будемо використовувати всі параметри. В кінці збережемо навчену модель і додаймо в проект, як це вказано в прикладі 1.3, нижче:

```
let model = try MLRegressor (trainingData:
apartmentsData, targetColumn: "Price")
let modelPath = URL (fileURLWithPath:
"/Users/CreateMLTest/ApartmentsPricer.mlmodel")
try model.write (to: modelPath, metadata: nil)
```

Приклад 1.3 – Збереження навченної моделі

Core ML 3 [4] буф представлений у 2019му році, він забезпечує цікаві функції iPhone, такі як FaceID, Animoji та Augmented Reality. З моменту запуску Core ML у 2017 році він пройшов довгий шлях і тепер підтримує безліч інструментів, які допоможуть нам швидко створити великі програми на основі машинного навчання. Apple зробила велику роботу зі створення інструментів та рамок, які використовують машинне навчання. Існує чимало варіантів створення AI-програм, кожен зі своїми плюсами та мінусами. Ось високоякісний огляд екосистеми AI від Apple, зображено на рисунку. 1.1, нижче:

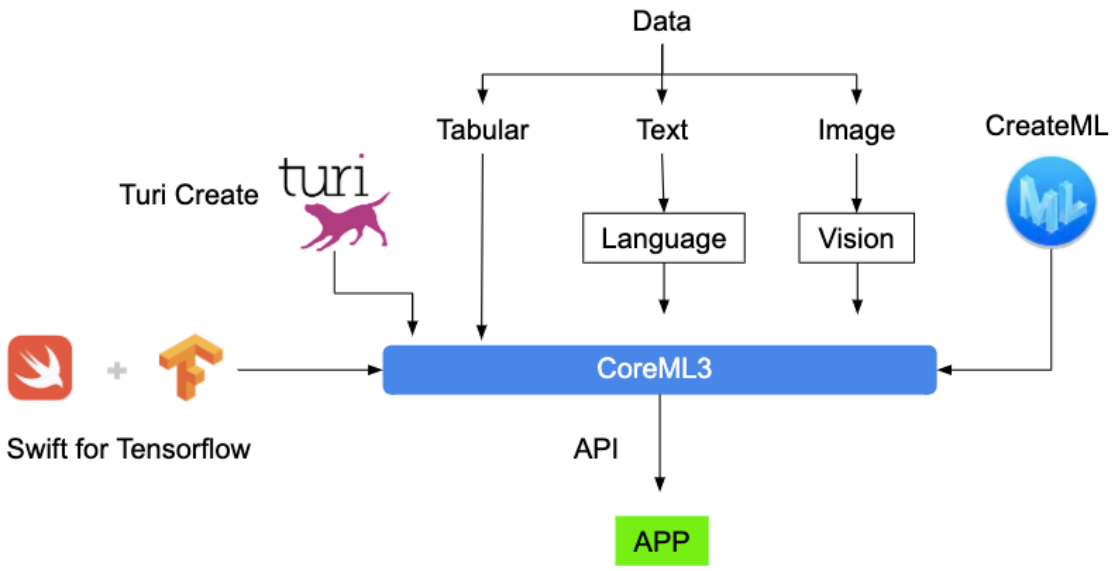

Рисунок 1.1 – Екосистема AI від Apple

Це має бути вашою базовою базою, якщо ви хочете додати в додаток рекомендації, виявлення об'єктів, класифікацію зображень, подібність зображень або класифікацію діяльності. Що мені подобається у програмі Turi Create - це те, що ми можемо працювати з нею в Python так само, як і звичайний робочий процес. І коли ми задоволені нашою моделлю, просто імпортуємо її в Core ML для використання в додатках для iOS, macOS, watchOS та tvOS!

Ось деякі завдання Turi Create підтримує прямо з коробки, зображено на рисунку 1.2:

ML Task	Description
Recommender	Personalize choices for users
Image Classification	Label images
Drawing Classification	Recognize Pencil/Touch Drawings and Gestures
Sound Classification	Classify sounds
Object Detection	Recognize objects within images
One Shot Object Detection	Recognize 2D objects within images using a single example
Style Transfer	Stylize images
Activity Classification	Detect an activity using sensors
Image Similarity	Find similar images
Classifiers	Predict a label
Regression	Predict numeric values
Clustering	Group similar datapoints together
Text Classifier	Analyze sentiment of messages

Рисунок 1.2 – Turi Create ML Tasks

Create ML дозволяє нам будувати моделі машинного навчання без написання великого коду. Що мені подобається в цьому інструменті, це те, що ви можете просто перетягнути свої навчальні дані та вибрати модель, яку ви хочете (розпізнавання мови, виявлення об'єктів тощо), і він автоматично почне тренувати модель, у той час як Turi Create працює в Python, ми можемо використовувати CreateML для створення на Mac. Отжке, він підтримує навчання на GPU.

Swift for TensorFlow має гнучкий, високоефективний API TensorFlow / PyTorch для створення складних архітектур нейронних мереж [1]. Найбільш захоплююче в цьому фреймворці - це те, що її код настільки читабельний, як і в Python. Ось той же код моделі як у Swift, так і в Python (зауважте подібність), рисунку 1.3:

```swift
struct MyModel: Layer {
    var conv = Conv2D<Float>(filterShape: (5, 5, 3, 6))
    var pool = MaxPool2D<Float>(2)
    var flatten = Flatten<Float>()
    var dense = Dense<Float>(16*5*5, 10)
    @differentiable
    func call(_ input: Tensor<Float>) -> Tensor<Float> {
        return dense(flatten(pool(conv(input)))) }}
```

Swift

```python
class MyModel(nn.Model):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3,6,kernel_size=5)
        self.pool = nn.MaxPool2d(2)
        self.flatten = Flatten()
        self.dense = nn.Linear(16*5*5, 10)
    def forward(self, x):
        return self.dense(self.flatten(self.pool(self.conv(input))))
```

Python

Рисунок 1.3 – Схожість кодів на Swift та Python при використанні TensorFlow

Використовуйте Swift for TensorFlow, коли вам потрібні високі показники роботи ваших моделей і хочете їх ефективно розгорнути.

Я люблю Apple’s Core ML 3 [4] framework. Він не тільки дозволяє використовувати інструменти, які ми бачили вище, але також підтримує деякі власні функції. По-перше, CoreML3 дозволяє нам імпортувати навчальні моделі машинного навчання або глибокого навчання з усіх основних каркасів Python, подивіться на Рисунок 1.4, на сторінці 22.

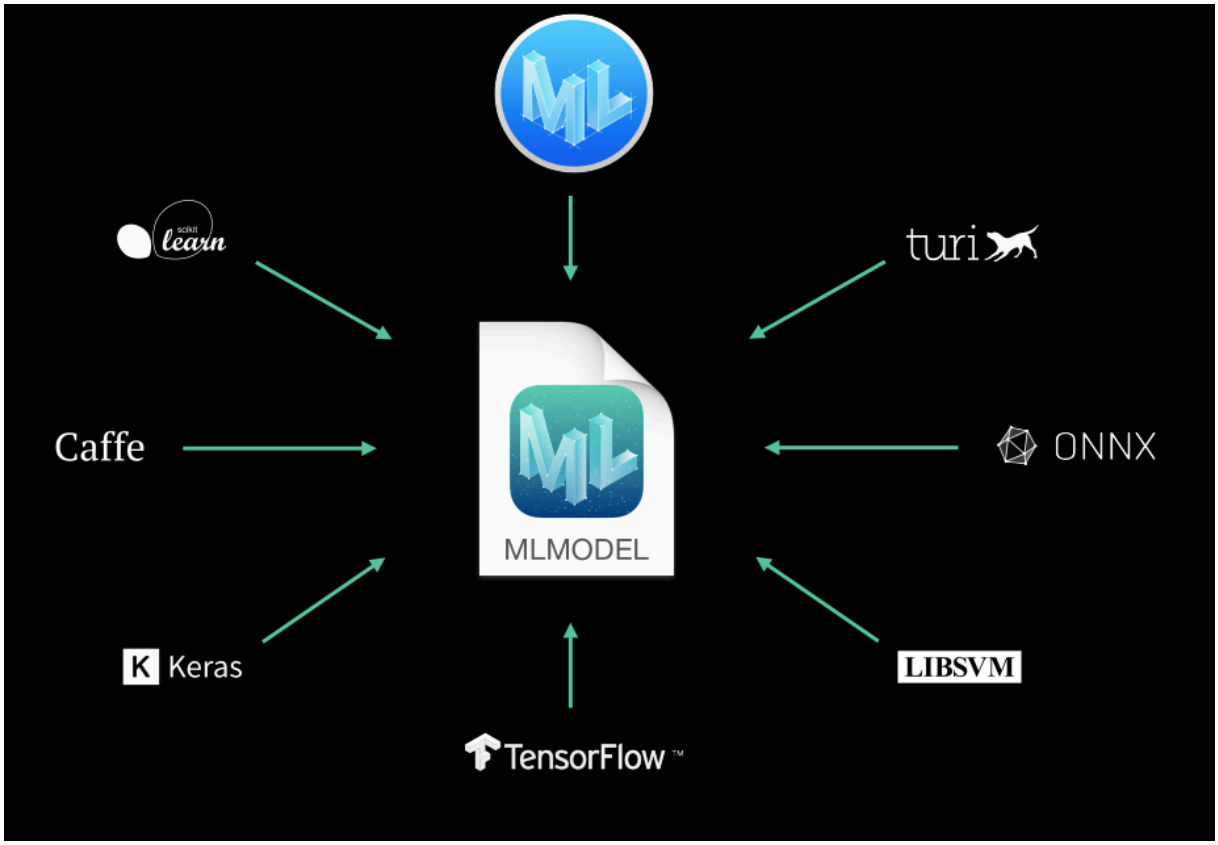

Рисунок 1.4 – Сумісні для імпорту з Core ML моделі

Ми розглядали цю особливість Core ML у попередній моїй роботі. Тут ми побачимо ще одну цікаву особливість Core ML 3 - як ми можемо використовувати безліч заздалегідь підготовлених попередньо тренованих моделей, які тепер підтримує CoreML3. Нижче, на рисунку 1.5, наведено список моделей, які підтримує Core ML 3. Зауважте, що деякі з них (як Squeezenet, DeeplabV3, YOLOv3) настільки недавні, що вийшли б лише місяці тому. Найвідоміші з доступних варіантів перераховані на рисунку 1.5, на сторінці 23

Рисунок1.5 – список моделей що підтримує Core ML

Усі ці моделі фактично оптимізовані для забезпечення найкращої продуктивності на мобільних пристроях, планшетах та комп’ютерах. Це велика річ у Apple, а особливо на пристроях носимої електроніки. Це означає, що хоча багато з них є складними моделями на основі глибокого навчання, нам не потрібно багато турбуватися про ефективність під час розгортання та використання їх у наших додатках. Ви дивилися цьогорічну конференцію WWDC? Було кілька цікавих анонсів про Core ML 3 [4] та підтримку, яку матимуть пристрої Apple для цієї основи. Ось короткий підсумок, якщо ви пропустили його.

Навчання на пристрої - це найбільша ролі в Core ML 3 [4]. Поки що у нас була лише підтримка «висновку на пристрої». Це в основному означає, що ми навчаємо нашу модель на якійсь іншій машині, а потім використовуємо навчену модель для прогнозування в реальному часі на самому пристрої. Це призвело до кращого користувальницького досвіду, оскільки ми не залежали від Інтернету, щоб отримувати прогнози. Core ML 3 тепер також підтримує навчання на пристрої! Ви отримуєте доступ до процесора, графічного процесора та нейронного двигуна iPhone для навчання моделей машинного навчання та глибокого навчання.

Ви можете розглянути тренінг Core ML 3 як форму трансферного навчання або навчання в Інтернеті, де ви лише налаштовуєте існуючу модель. Візьмемо, наприклад, Face ID. Їй потрібно постійно оновлювати свою модель, коли обличчя користувача змінюється з часом (росте борода, носить різний макіяж, старіє тощо). Основна ідея полягає в тому, щоб спочатку створити загальну модель, яка дає середню продуктивність для всіх, а потім зробити копію, налаштовану для кожного користувача. І з часом ця модель стане справді хорошою для конкретного користувача, як це зображено на рисунку 1.6, нижче:

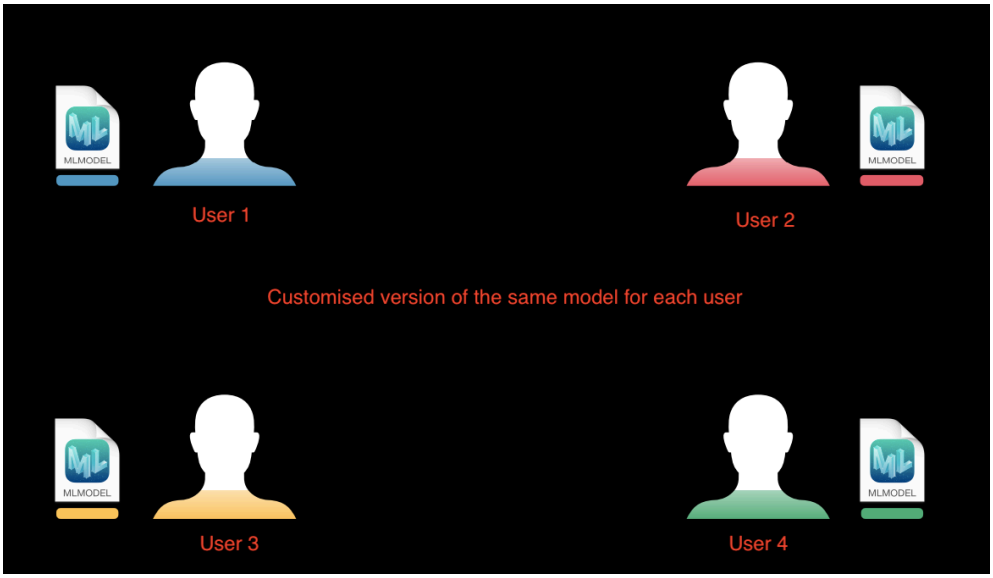

Рисунок1.6 – можливість кастомізації моделі під окремих користувачів

Це має кілька переваг:

- навчання відбуватиметься на особистому пристрої користувача, що означає високу конфіденційність даних для користувача;
- нам не потрібно встановлювати величезні сервери, щоб полегшити навчання моделям мільйонам користувачів додатків;
- оскільки Інтернет не задіяний, моделі завжди будуть доступні для прогнозування.

Також, в Core ML 3 [4] були додані нові типи шарів нейронної мережі, рисунку 1.7:

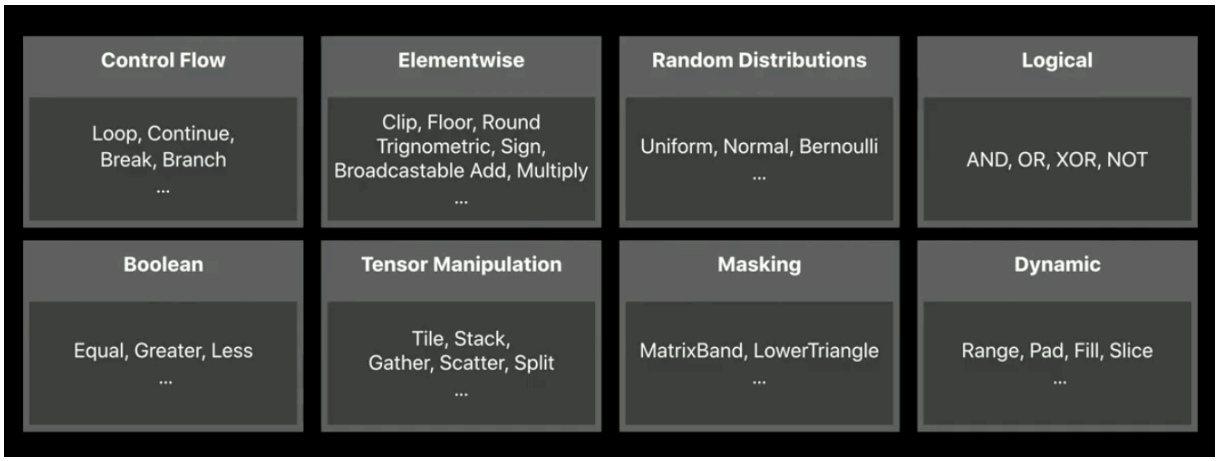

Рисунок1.7 – Нові типи шарів нейронної мережі

Крім шарів для різних типів моделей, Core ML 3 [4] також містить 100+ шарів для проміжних операцій, таких як маскування, маніпуляція тензором, булева логіка, управління потоком тощо. Деякі з цих типів шарів використовуються в найсучасніших архітектурах нейронної мережі, а Core ML 3 вже підтримує їх. Це просто означає, що ми можемо легко створити такі моделі відразу для наших додатків.

2 ПОСТАНОВКА ЗАДАЧІ

Створення програмного продукту для пристроїв носимої електроніки, що дозволить користувачеві вільно відслідковувати та аналізувати будь-які фізичні активності, а також моніторити показники частоти серцевих скорочень та, можливо, інших показників, що можуть відстежуватись за допомогою медицинських пристроїв, підключених до основного пристрою носимої електроніки (смарт-годиннику чи фітнес-трекеру) для покращення результатів і підтриманню здорових режимів тренувань. Як я вже писав раніше, наразі таких додатків не існує, а ті що існують, значно обмежені за функціоналом та не спроможні на більш-менш універсально роботу та жорстко зав`язані на вже натреновані моделі, без можливості врахування особливостей кожної окремої людини та нестандартних (не передбачених моделлю вправ).

Звісно ж що нічого поганого у використанні вже готової моделі немає, вона працює і забезпечує достатньо високу точність результатів, але особисто я, як і багато інших користувачів пристроїв носимої електроніки, помітив, що деякі з вправ котрі я роблю, відслідкувати та проаналізувати навіть за допомогою сторонніх додатків, не кажучи вже про стандартні функції пристрою, не можливо, бо модель від розробника їх не передбачає.

Також необхідно створити додаток-компаньйон для смартфону, що дозволить користувачеві переглядати та корегувати результати зручно, на екрані достатнього розміру, а також, за бажанням, синхронізувати власні дані із системою через Health Kit, щоб вони були доступні на інших пристроях та для інших додатків.

3 РІШЕННЯ ПРОБЛЕМИ

За останні декілька років з`явилися технології та набори інструментів, що дозволяють ефективно проводити оновлення моделі на самому пристрої. Загалом, звісно ж, це було теоретично можливо і раніше, але вартістю такого рішення було б значне енергонавантаженя на пристрої носимої електроніки, що значно зменшило би їх час життя та унеможливлювало використання таких пристроїв достатньо довго, що необхідно для деяких користувачів і що важніше через види активності, і що більш важливо, необхідно всім користувачам, бо інакше пристрій стане неможливо використовувати – нікому не потрібен пристрій що сяде за декілька годин використання.

Найбільш актуальним рішення проблеми вважаю використання Core ML 3 в режимі on-device machine learning, тобто навчання в режимі реального часу, на основі вже існуючої моделі, котра буде спільною точкою старту для усіх нових користувачів, та буде доповнюватись на основі даних, отриманих від користувачів, за умови їх згоди, що дозволить швидко розширювати та покращувати точність визначення активностей.

За необхідне, на даний момент, вважаю використання у і якості платформи watchOS на Apple Watch, як платформи, котра надає усі необхідні інструменти, має сертифікацію FCA як медицинского пристрою, підтримує підключення усіх необхідних додаткових сенсорів та медицинських пристроїв по Bluetooth та є найрозповсюдженішим пристроєм із цієї категорії. Для створення такого додатку буде використано мову програмування Swift, а також Swift UI та Combine як додаткові інструменти для покращення взаємодії користувача з додатком та пристроєм в цілому, а також для забезпечення сучасності рішення для певного ресурсу на майбутнє, для простішого поліпшення функцій додатку у подальшому.

Також задля максимізації ефективності розробки та забезпечення подальшої можливості у доповнені програмного засобу, необхідно дотримуватись сучасних правил архітектурного проектування та шаблонів (паттернів) програмування. Найбільше нас цікавлять CLEAN та SOLID та організація архітектури додатку в цілому. Для створення UI використано Swift UI, а у якості дизайн-гайду – Неоморфізм (Soft UI). У міру того, як додаток зростає у складності функцій, нам потрібен спосіб управління ним. Один із способів поводження з ним - це організація застосування в шари. Зараз я розповім, як багатошарова архітектура застосовується до мого додатку iOS.

3.1 Багатошарова архітектура

Багатошарова архітектура організовує додаток iOS у набір шарів. Шар - це логічний механізм структурування елементів, які складають ваш додаток. Традиційно він складається з трьох частин, як це зображено на рисунку 3.1, нижче

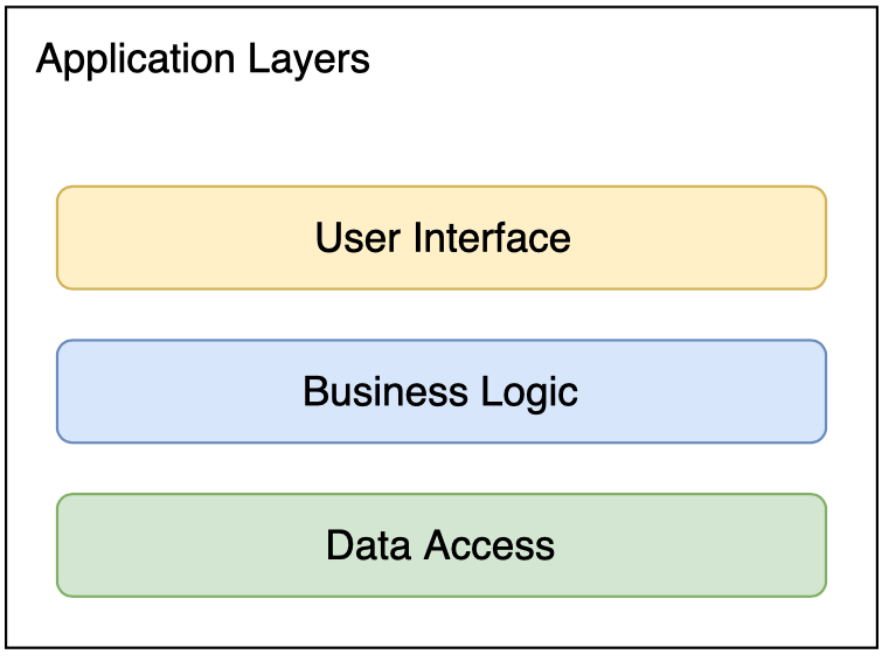

Рисунок 3.1 – Багатошарова архітектура додатку iOS

Головні шари багатошарової архітектури:

- User Interface (Інтерфейс користувача) обробляє взаємодію між користувачем та програмою;
- Data Access (Доступ до даних) відкриває дані, що зберігаються у зовнішній системі, наприклад CoreData, iCloud, REST API до бізнес-рівня;
- Business Logic (Бізнес-логіка) - це програмне представлення бізнес-концепцій. Воно обмежує поведінку програми, щоб відповідати потребам конкретної компанії.

Шари відповідають правилу залежності: вищий шар може отримати доступ до будь-якого шару під ним, але не навпаки. Це означає, що бізнес-логіка ніколи не повинна викликати методи з рівня інтерфейсу користувача. Така архітектура дозволяє робити деталі вищого рівня, такі як бізнес-логіка, незалежними від нижчих, що покращує спроможність до тестування, стабільність та адаптивність, тобто, можливість заміни фізичної реалізації кожного з шаріф на іншу, бо взаємодіють вони через абстракції. Чим ближче компонент до входу та виходу, тим нижчий рівень у нього. Враховуючи додаток iOS, інтерфейс користувача має найнижчий рівень, а сутність - найвищий. Визначивши три основні шари, розглянемо кожен детально.

Шар користувальницького інтерфейсу. Основними обов'язками цього рівня є відображення інформації для користувача та обробка команд користувача. Команди будуть відповідні дії до ділової логіки та даних. Основними одиницями цього шару є перегляди, презентатори, дошки розповідей тощо. Вони, як правило, організовані за допомогою будь-якого з MV *-шаблонів (MVC, MVP, MVVM). Ось як Роберт Мартін описує структуру шару інтерфейсу: “User Interface Layer повністю міститиме архітектуру MVC графічного інтерфейсу. Тут присутні всі представники, перегляди та контролери. Моделі, ймовірно, є лише структурами даних, які передаються ... від випадків використання до презентаторів та переглядів".

Шар доступу до даних. Шар доступу до даних містить зв'язок із зовнішніми системами, такими як бази даних та мережа. Враховуючи, що більшість додатків для iOS - це тонкі клієнти, вони часто зводяться до отримання даних з Інтернету та, можливо, кешування їх у CoreData або UserDefaults. Шлюз, репозиторій та об’єкт передачі домену є основними моделями дизайну, застосованими в цьому шарі. Шлюз інкапсулює доступ до зовнішньої системи або ресурсу. Він використовується для перетворення складного API в зручний для вашої програми. Шлюз орієнтований на клієнта і не призначений для загального використання. Як і шлюз, репозиторій інкапсулює об'єкти, що містяться у джерелі даних (наприклад, Інтернет, база даних) та операції над ними. Відмінна особливість полягає в тому, що репозиторій приймає запити, отже, більш загального призначення порівняно з шлюзом. Репозиторій також надає можливості авторизації, якщо цього вимагає базове джерело даних. Шлюз та сховище дозволяють досягти спільної мети різними способами: вони інкапсулюють джерело даних та те, як дані запитуються з нього.

Об'єкт передачі домену (Data Transfer Object, DTO) - об’єкт, який містить дані без будь-якої поведінки. Мета його - перетнути якусь межу та перенести дані своїм клієнтам. Ця межа може бути фізичною, як iOS-додаток та Інтернет, і логічною, як різні архітектурні шари. Якщо ви перечитаєте цитату Роберта Мартіна, ви побачите, що "M" з шаблону MVC означає модель перегляду. Саме DTO агрегує інформацію, яка повинна відображатися на екрані, а не модель бізнес-логіки. Шаблон DTO широко використовується для зв'язку з мережею. Для цього створюються проміжні структури Codable, які відображають структуру запиту та відповіді.

Шар Бізнес-Логіки:

- Бізнес-логіка - це те, що заробляє або економить гроші та нематеріальні речі для вас або вашого клієнта. Шар бізнес-логіки зосереджений навколо правил бізнесу;
- Бізнес-правила - це операції, які реалізують різні аспекти бізнес-логіки. Вони задовольняють потреби компанії і є основною причиною того, чому заява насправді написана;
- Entities - це найважливіші правила ведення бізнесу разом із даними, якими вони діють. Сутність - це абстрактне поняття, яке зазвичай реалізується у вигляді класу з кількома полями та методами. У чисто функціональних мовах це може бути сукупність процедур.

Правила ведення бізнесу описуються у випадках використання. Випадок використання визначає поведінку програми з точки зору вводу користувачів та відповідних результатів. Наприклад, що додаток повинен робити, коли користувач подає логін та пароль. Через свою значимість рівень бізнес-логіки повинен розроблятися з особливою ретельністю.

3.2 Проектування бізнес-логічного шару

Рівень бізнес-логіки складається з усіх компонентів, необхідних для представлення та реалізації даної бізнес-концепції. Особисто я підтримую рішення «Чиста архітектура», яке визначає три одиниці: об'єкти, послуги та інтерфейси. Давайте детально обговоримо кожен блок.

Суб'єкти - це звичайні об'єкти Swift, які містять критичні бізнес-правила та дані, необхідні для їх виконання. Правила бізнесу - найважливіша частина цього визначення. Структури з кількома полями недостатньо для створення сутності. Якщо будуватися виключно з

сутностей, абстракції, які вона вводить, можуть бути занадто слабкими. Це призводить до витоку правил бізнесу на рівень користувальницького інтерфейсу.

Слабкі абстракції є основною причиною масових контролерів перегляду (massive view controller) та масових делегатів програм (massive view delegate). В іншому випадку, розміщення всіх бізнес-правил в сутностях перетворює їх у універсальні класи, що називаються «щось –» + «менеджер», «контролер», «двигун», «помічник» тощо. Це трапляється, тому що необхідні взаємодії можуть бути складним і впливати на глобальний стан програми, залучати кілька ресурсів, вимагати декількох відповідей на команду. Незалежно від того, як створені практичні організації, вони не можуть дотримуватися всіх правил роботи програми. Деякі бізнес-концепції природно описуються операціями, а не даними, що призводить нас до поняття послуг.

Службовий шар визначає набір операцій, доступних для багатьох видів клієнтів, найбільш значущим з яких є інтерфейс користувача. Ці операції повинні бути простими у використанні та орієнтовані на потреби своїх клієнтів. Основними одиницями цього рівня є сервісні класи.

Клас обслуговування (Service) - це об'єкт без стану, який визначає лише набір операцій з даними. Він має такі відмінні властивості:

- сервіси орієнтовані на клієнтів і повністю визначені з урахуванням потреб їх клієнтів;
- сервіси не мають стану. Однак вони можуть мутувати глобальні дані, що призводить до побічних ефектів;
- сервіси визначають свої входи та результати з урахуванням сутності та мовних примітивів.

Зазвичай послуги виконують такі операції:

- Бізнес-логіка;
- CRUD (створювати, читати, оновлювати, видаляти) для об'єктів;
- Мережі;
- Перевірки.

Окрім entities та services, рівень бізнес-логіки містить інтерфейси. Вони є абстрактними операціями, котрі виконуватимуться рівнем доступу до даних. Це дає нам три основні одиниці рівня бізнес-логіки: сутності, послуги та інтерфейси. Отримаємо наступну структуру додатку, зображену на рисунку 3.2, нижче:

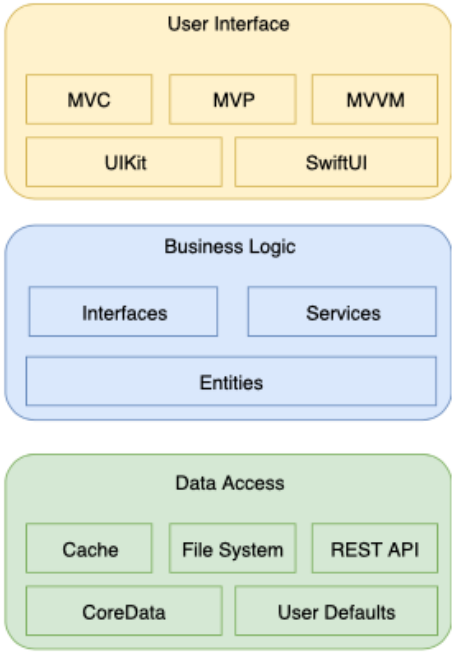

Рисунок 3.2 – Типова архітектура iOS додатку

Далі розглянемо, як дані перетікають із шару в шар. Усе починається користувачем, що взаємодіє з додатком. Потім дані переходять до рівня бізнес-логіки, де це призводить до операцій CRUD над сутностями. Нарешті, об'єкти зберігаються у back-end або в базі даних. Зворотний процес

також має місце. Дані отримуються з back-end або бази даних, трансформуються за правилами бізнесу та нарешті представляються користувачеві, а порядок залежностей різний, як у даних. Шари нижнього рівня, що представляють собою інтерфейс користувача та доступ до даних, повинні залежати від рівня логіки бізнесу, як це зображено на рисунку 3.3, нижче:

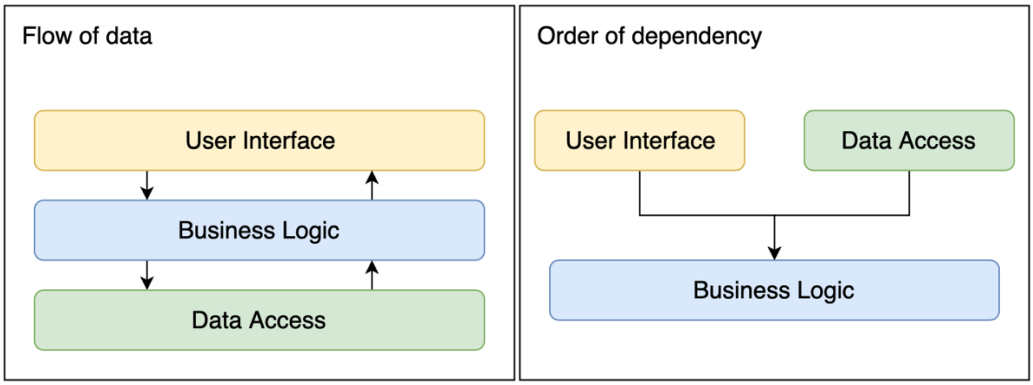

Рисунок 3.3 – Data Flow та Order of Dependency в iOS додатку

Багатошарова архітектура широко застосовується в розробці програмного забезпечення в цілому, і iOS не є виключенням. Він розділяє подібні проблеми на шари і застосовує до них правило залежності. Це дозволяє мінімізувати вплив змін та керувати складністю. Врешті-решт, багатошарова архітектура вирішує стабільність програми для iOS, що є ключовим фактором успіху.

3.3 CLEAN та SOLID архітектури

CLEAN та SOLID необхідні для збільшеня стабільості додатку, а також його спроможності до тестування та змін у майбутньому – усе це необхідно для зменшення вартості розробки додатку, адже в Spagetty Code дуже важно розібратись навіть автору коду, а при роботі в команді, людині що отримає цей код після вас буде взагалі нічого не зрозуміло. Основне правило полягає в тому, що нижній шар нічого не повинен знати про верхньому шарі і залежність повинна залишатися односпрямованою. Зверніть увагу на рисунок 3.4 нижче, що відповідає ідеї Uncle Bob:

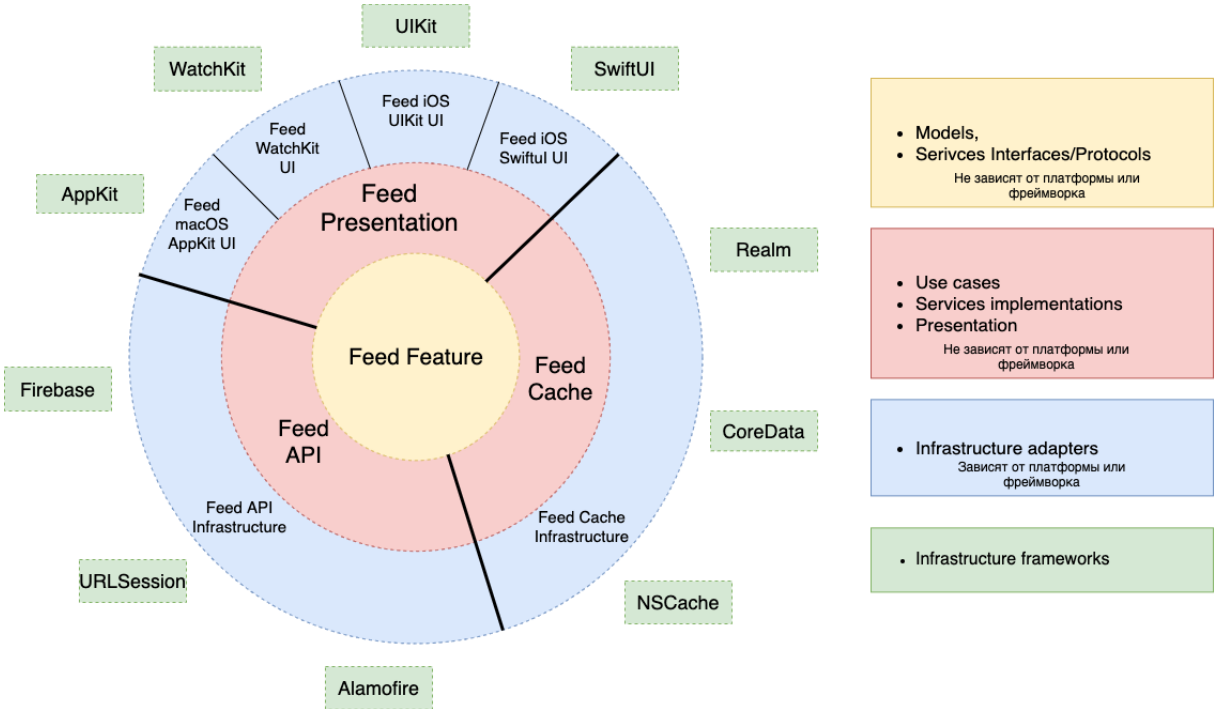

Рисунок 3.4 – CLEAN архітектура на прикладі типової Feed задачі

У центрі всієї архітектури у нас Feed Feature. Це платформонезалежна модль і інтерфейси, які відносяться до бізнес-логіки нашої фічі. Вони не мають ніякої залежності від оточення, будь то шар презентації або UI.

Наступний шар - це use cases нашої фічі: модуль для роботи по мережі, модудь для кешування, модуль презентації. Тут лежить реалізація інтерфейсів сервісів з нашого ядра. Цей шар залежить тільки від ядра архітектури. Піднімаємося на рівень вище. Тут знаходяться інфрастуктурні адаптери, які дозволяють шару презентації не залежати від фреймворків і використовуваних бібліотек. Цей шар вже залежить від платформи і фреймворків.

Дотримуючись даної архітектури, ми отримуємо чіткий поділ відповідальностей і можливість швидко і легко підміняти реалізації / фреймворки / платформи. Можна легко замінити URLSession для роботи з мережею на Alamofire не чіпаючи при цьому бізнес логіку і її презентацію. Можна легко перейти на SwiftUI чи RXSwift, помінявши при цьому тільки UI шар і т.д. У випадку, якщо ви цього не знали, в комп'ютерному програмуванні акронім принципів SOLID був введений Майклом Піршером для п'яти принципів, які були визначені Робертом К. Мартіном на початку 2000-х. Як ви знаєте, для отримання робочого програмного забезпечення ми повинні мати низьку зв'язок, високу когезію та сильну інкапсуляцію (high coheasion, low coupling, strong encapsulation), що допомагає нам отримати принципи SOLID. Ідея полягає в тому, що, застосовуючи ці принципи разом, ви зможете написати більш якісний код, який є надійним. Створена система стає простою в обслуговуванні, повторному використанні та розширенні з часом. В основному принципи SOLID допомагають розробникам програмного забезпечення досягти масштабованості та уникати того, щоб ваш код порушувався щоразу, коли ви стикаєтесь із зміною.

S.O.L.I.D. stands for:

- Single-responsibility principle;
- Open-closed principle;
- Liskov substitution principle;
- Interface segregation principle;
- Dependency Inversion Principle.

При розробці будь-якого програмного забезпечення є дуже важливими дві концепції: згуртованість (коли різні частини системи працюватимуть разом для отримання кращих результатів, ніж якщо кожна частина працюватиме індивідуально) та з'єднання (може розглядатися як ступінь залежності клас, метод або будь-яке інше програмне забезпечення). Зв'язок зазвичай присутній у великій кількості коду, і як я вже згадував раніше, оптимальною ситуацією було б низьке з'єднання та висока згуртованість. Загальна схема SOLID передана на рисунку 3.5, нижче:

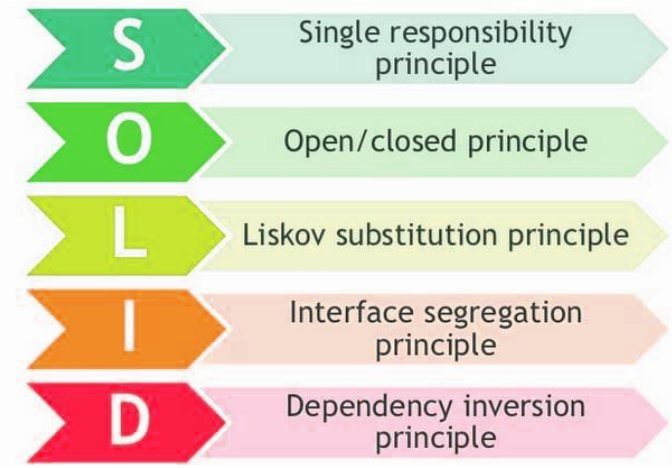

Рисунок 3.5 – SOLID Principle

Розглянемо кожен із принципів окремо:

1. Принцип єдиної відповідальності

«Ніколи не повинно бути більше однієї причини для зміни класу».

Як бачимо, цей принцип стверджує, що об’єкт / клас повинен нести лише одну відповідальність і що він повинен бути повністю інкапсульований класом. Тут, коли ми говоримо про відповідальність, ми маємо на увазі причину для зміни. Цей принцип призведе до посилення згуртованості в класі та більш слабкої зв'язку між класами залежностей, кращої читабельності, та коду з меншою складністю. Набагато складніше зрозуміти та редагувати клас, коли він має різні обов'язки. Отже, якщо у нас є декілька причин для зміни, функціональність має бути розділеним на два класи, і кожен матиме свою відповідальність. Ми дбаємо про розділення функціональних можливостей, оскільки кожна відповідальність - це доступ до змін. Коли у класу є більше ніж одна відповідальність, ці обов'язки поєднуються, і це з'єднання може призвести до крихкої бази коду, яку важко переробити, коли з'являються нові вимоги.

2. Принцип відкритого закриття

«Програмні об'єкти (класи, модулі, функції тощо) повинні бути відкритими для розширення, але закритими для модифікації».

Тут ідея полягає в тому, що суб'єкт господарювання дозволяє розширити свою поведінку, але ніколи не змінюючи свій вихідний код. Будь-який клас (або все, що ви пишете) повинен бути написаний таким чином, щоб його можна було використовувати як є. Його можна продовжити за потреби, але він ніколи не може бути змінений. Використовуйте клас будь-яким необхідним вам способом, але зміна його

поведінки відбувається шляхом додавання нового коду, не зміною старого. Цей же принцип може застосовуватися і для модулів, пакетів, бібліотек.
Застосовуючи принцип відкритого закритого типу, ви отримаєте нещільну зв'язок, поліпшите читабельність і, нарешті, зменшите ризик порушити існуючий функціонал.

3. Принцип заміщення Лісков
«Підтипи повинні бути замінені для їх базових типів»

Як видно з назви, принцип заміни Лісков був визначений Барбарою Лісков. Ідея тут полягає в тому, що об'єкти повинні бути замінені примірниками їх підтипів, і це не впливає на функціонування вашої системи з точки зору клієнта. В основному, замість використання реальної реалізації, ви завжди повинні мати можливість використовувати базовий клас та отримувати результат, якого ви чекали. Часто, коли ми хочемо зобразити об’єкт, ми моделюємо наші класи на основі його властивостей, і замість цього ми насправді повинні приділяти більше уваги поведінці.
Цей принцип в основному підтверджує, що наші абстракції є правильними, і допомагає нам отримати код, який легко використовувати для повторного використання та ієрархії класів, які дуже легко зрозуміти.
Багато хто говорить, що Принцип заміщення Лісков має дуже міцне відношення до попереднього принципу, принципу відкритого закриття. Роберт К. Мартін навіть каже, що «порушення LSP є прихованим порушенням OCP».

4. Принцип поділу інтерфейсу

«Класи, що реалізують інтерфейси, не повинні змушувати реалізовувати методи, які вони не використовують.»

Тут мова йде про те, як писати інтерфейси. Отже, що зазначено? В основному, як тільки інтерфейс стає занадто великим / жирним, нам абсолютно потрібно розділити його на невеликі інтерфейси, які є більш конкретними. І інтерфейс буде визначений клієнтом, який ним буде користуватися, а це означає, що клієнт інтерфейсу буде знати лише про методи, які з ними пов'язані.

Насправді, якщо ви додати методи, яких там не повинно бути, класи, що реалізують інтерфейс, повинні також реалізувати ці методи. Ось чому; клієнта не слід змушувати залежати від інтерфейсів, які він не використовує. ISP призначений для того, щоб підтримувати роз'єднаність системи і, таким чином, легше переробляти, змінювати та розгортати.

5. Принцип інверсії залежності

«Модулі високого рівня не повинні залежати від модулів низького рівня, а обидва повинні залежати від абстракції. Абстракція не повинна залежати від деталей; а деталізація повинна залежати від абстракції.»

Останній із принципів SOLID, але не в останню чергу, цей принцип стосується насамперед зменшення залежностей між модулями коду. Принцип інверсії залежності залежить від розуміння того, як правильно зв'язати свою систему разом - якщо деталізація вашої реалізації залежатиме від абстракцій вищого рівня, це допоможе вам отримати систему, яка правильно поєднана. Також це вплине на інкапсуляцію та згуртованість цієї системи. Тобто, залежність має бути однонаправленою.

3.4 Функціонально-реактивне програмування з Swift UI та Combine

Як iOS Software Engineer, я не можу ігнорувати функціональне програмування. З впровадженням фреймворків SwiftUI та Combine Apple зробила функціональне програмування основною парадигмою в інженерії iOS, watchOS, macOS та tvOS. Якщо ви шукаєте в Google визначення функціонального програмування, ви, швидше за все, знайдете безліч різних і суперечливих думок. Вони зводиться до думки, що функціональне програмування - це програмування з чистими функціями. Ваш додаток - одна велика чиста функція. Це визначення може бути добре для науковців, оскільки функціональні програми є засобом побудови формальних математичних доказів. У комерційних (промислових) проектах у нас є побічні ефекти. Ми надаємо користувальницький інтерфейс, здійснюємо платежі, записуємо у файли, користуємося мережею та надсилаємо електронні листи. Це все побічні ефекти. Ми хочемо, щоб наші користувачі могли робити все це. Побічні ефекти - це причина, чому наші програми корисні. Ми пишемо програми, тому що хочемо цих ефектів, цей функціонал. Те, що ми дійсно хочемо у функціональному програмуванні, це мати якомога більше рядків коду в чистих функціях і якомога менше побічних ефектів.

Суть функціонального програмування полягає в тому, що він робить різницю між чистим кодом і нечистим кодом і мінімізує останній. Як і об'єктно-орієнтоване програмування, функціональне програмування є парадигмою. Він має набір принципів, які формують ваш розумовий каркас під час проведення мистецтва програмування. Ціна функціонального програмування полягає в тому, що почати з цього важко. Нам потрібно вивчити багато понять, перш ніж ми зможемо написати корисні програми.

Ми можемо розглядати кожну функціональну програму як набір із трьох понять. Це дуже важливо знати і розуміти.

У двох словах, функціональну програму можна розділити на три частини: функції, ефекти та дані. Об'єктно-орієнтоване програмування має свої концепції - інкапсуляція, успадкування та поліморфізм. Ви не можете написати програму об'єктно-орієнтованим способом без цих трьох.

Swift - мова з багато парадигми, що означає, що ми можемо виконувати як функціональне, так і об’єктно-орієнтоване програмування. Який з них нам використовувати? Відповідь і те, і інше. Дві парадигми не є взаємовиключними. Якщо ви вибираєте функціональне програмування, ви просто використовуєте функціональні навички частіше, ніж об'єктно-орієнтоване, і навпаки. Це дозволяє будувати рішення з точки зору проблеми замість обмеженого набору інструментів. Ви не хочете використовувати функціональне програмування для рівня зв'язку в веб-сокетах. І навпаки, перетворення Фур'є природно описується функціями замість об'єктів та класів. Тому функції, ефекти та дані дуже важливі для розуміння при виконанні функціонального та об'єктно-орієнтованого програмування. Давайте дізнаємося, що це таке. Я впевнений, що ви вже добре розумієте, що означає функція в об'єктно-орієнтованому світі. Пам'ятайте, що у функціональному програмуванні ми повинні розрізняти, що є чистим, а що ні. Ми щойно назвали три основні поняття: функції, дані та ефекти. Кажучи функцію, ми дійсно маємо на увазі чисту функцію. Що таке чиста функція? Чиста функція не може залежати від нічого, що змінюється. Наприклад: змінюється час, змінюється мережа, змінюються властивості та налаштування пристрою iOS. Чиста функція сама по собі нічого не змінює. Ви можете викликати чисту функцію стільки разів, скільки захочете, і вона даватиме вам однаковий результат кожного разу. Написання функцій таким чином має глибокі наслідки. Це дозволяє легко

міркувати про свій код і передбачати його поведінку. Чисті функції легко перевірити та повторно використовувати, оскільки вони не залежать від того, коли і де вони виконуються.

Виклик чистої функції багато разів дає один і той же результат кожного разу. Виклик ефекту багато разів не робить. Уявіть, як запустити ракету. Це ефект. Запуск трьох ракет відрізняється від запуску однієї ракети. Ефекти залежать від того, скільки разів вони викликаються і коли вони викликаються. Читання з глобальної чи змінної змінної - ще один приклад ефекту, оскільки це залежить від того, коли ми це робимо. Ми можемо отримати різні відповіді в різні моменти часу. Ефекти надзвичайно важливі. Вони є точкою, коли ваш код відповідає реальному світу. У той же час ефекти є найбільш клопіткою частиною програми, оскільки це має значення, коли, в якому порядку і скільки разів ви їх запускаєте. Деякі мови ускладнюють виконання ефекту. Наприклад, Haskell має спеціальний параметризований тип під назвою IO. Будь-яке значення в контексті IO повинно залишатися в цьому контексті. Це не дозволяє змішувати чистий код із нечистим. Що ми маємо у Swift, щоб розділити чисті функції та ефекти? Нічого. Коли ми пишемо функції у Swift, ми маємо довіряти собі, що певні функції є чистими, а певні функції мають побічні ефекти.

Дані - це інформація про зовнішній світ, представлена в пам'яті комп'ютера. Дані інертні, тобто вони не запускаються самі. Дані просто такі, якими вони є. Дані потребують тлумачення. Що таке ім’я? Це назва ресторану чи це назва статті, яку я прочитав сьогодні вранці? Більшість мов програмування, включаючи Swift, вирішують ці проблеми з типами. Типи важливі, оскільки вони визначають види обчислень, дозволених на певних даних, щоб дані знали, як інтерпретувати себе.

Читання з змінної структури даних - це ефект. Тому у функціональному програмуванні ми прагнемо використовувати незмінні структури даних. Незмінна структура даних - це та, яка не може змінюватися.

Розуміння даних, функцій та ефектів є необхідною умовою, якщо ви виконуєте функціональне програмування за допомогою Swift. Але цього недостатньо Нам потрібні інструменти та методи, щоб сформувати наш код відповідно до цих принципів. До них відносяться незмінність, функції вищого порядку, рекурсія та функціональна архітектура. Я вважаю, що майбутнє техніки Swift - гібридне. Ми виберемо найкращі функції та ідеї як з функціональних, так і з об'єктно-орієнтованими парадигмами, щоб вирішити проблему.

Combine - це декларативний фреймворк Swift для обробки значень у часі, тобто, потоком. Він нав'язує функціональну реактивну парадигму програмування, яка відрізняється від об'єктно-орієнтованої, що переважає у спільноті розробки iOS. Реактивне означає програмування з асинхронними потоками значень. Реактивний маніфест розповідає більше про це. Функціональне програмування, як я вже писав раніше, стосується програмування з функціями. У Swift функції можуть передаватися як аргументи іншим функціям, повертаються з функцій, зберігаються у змінних та структурах даних та будуються під час виконання роботи як закриття. В той час як у декларативному стилі програмування ви описуєте, що робить програма, не описуючи потік управління, в імперативному стилі ви пишете, як працює програма, реалізуючи та обробляючи ряд завдань. Імперативні програми розраховують насамперед на стан, який зазвичай модифікується завданнями.

Програмування за допомогою програми Swift Combine є декларативним, реактивним та функціональним. Він включає ланцюжок функцій та передачу значень від однієї до іншої. Це створює потоки значень, що надходять від входу до виходу. Якщо ми скоротимо всі зайві слова, отримаємо схему, зображену на рисунку 3.6, нижче, або Combine = Publishers + Subscribers + Operators.

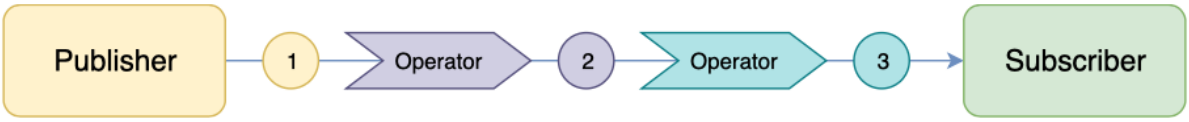

Рисунок 3.6 – Схема роботи Combine [1]

Видавець (Publisher) надсилає послідовності значень протягом одного часу або більше передплатників, зверніть увагу на приклад 3.1, нижче:

```
    protocol Publisher {
        associatedtype Output
        associatedtype Failure : Error
        func receive<S>(subscriber: S) where S :
Subscriber, Self.Failure == S.Failure, Self.Output ==
S.Input
    }
```

Приклад 3.1 - Combine publishers відповідають протоколу

Видавець може надсилати значення або припиняти з успіхом чи помилкою. Вихід визначає, які значення може надсилати видавець. Невдача визначає тип помилки, з якою вона може не виходити. Метод отримання receive(subscriber:) підключає абонента до видавця. Він визначає договір: вихід видавця повинен відповідати вкладу абонента, а також типи відмов.

Абонент отримує значення від видавця. Об'єднані абоненти відповідають наступному протоколу, як це вказано у прикладі 3.2, нижче:

```
    public protocol Subscriber :
CustomCombineIdentifierConvertible {
        associatedtype Input
        associatedtype Failure : Error

        func receive(subscription: Subscription)
        func receive(_ input: Self.Input) ->
Subscribers.Demand
        func receive(completion:
Subscribers.Completion<Self.Failure>)
    }
```

Приклад 3.2 - Combine subscribers відповідають протоколу

Абонент може отримати значення типу Введення або подія припинення з успіхом або Невдачею. Три методи отримання описують різні етапи життєвого циклу абонента.

Combine має двох вбудованих абонентів: Subscribers.Sink і Subscribers.Assign. Ви можете підключити їх, визваши у publisher будь-який із цих методів, як це вказано на прикладі 3.3:

```
    sink(receiveCompletion:receiveValue:) to handle new
element or completion event in a closure.
    assign(to:on:) to write new element to a property
```

Приклад 3.3 – Головні методи классу Subscriber

Subject - це особливий вид Publisher, який може вставляти в потік значення, передані ззовні. Інтерфейс теми пропонує три різні способи надсилання елементів. Приклад 3.4, нижче.

```
public protocol Subject : AnyObject, Publisher {
    func send(_ value: Self.Output)
    func send(completion:
Subscribers.Completion<Self.Failure>)
    func send(subscription: Subscription)
}
```

Приклад 3.4 – Subject Protocol

Combine має два вбудованих subjects: PassthroughSubject та CurrentValueSubject.

Publisher-Subscriber lifecycle:

1. Абонент підключається до видавця, визваши subscribe<S>(S);
2. Видавець створює підписку, через receive<S>(subscriber: S) на собі;
3. Видавець підтверджує запит на підписку. Він приймає receive(subscription:) від абонентів;
4. Абонент запитує кількість елементів, які він хоче отримати. Він викликає request (:) підписки і передає Demand як параметр. Попит визначає, скільки предметів видавець може надіслати передплатнику через підписку. У нашому випадку попит необмежений;
5. Видавець надсилає значення, зателефонувавши на отримання recieve (_ :) на абоненті. Цей метод повертає екземпляр Demand, який показує, скільки ще елементів очікує отримати абонент. Абонент може лише збільшити попит або залишити його таким же, але не може зменшити його;

6. Підписка закінчується одним із таких результатів:

   a. Скасовано. Це може статися автоматично при звільненні абонента, що показано у наведеному вище прикладі. Інший спосіб - скасувати вручну: token.cancel ();

   b. Закінчити успішно;

   c. Помилка з помилкою.

Оператори - це спеціальні методи, які викликаються у видавцях та повертають іншого видавця. Це дозволяє застосовувати їх один за одним, створюючи ланцюжок. Кожен оператор перетворює видавця, повернений попереднім оператором. Ланцюжок повинен бути створений видавцем. Тоді оператори можуть бути застосовані по черзі. Кожен оператор отримує видавця, створеного попереднім оператором у ланцюжку. Ми відносимо їх відносний порядок як вище, так і нижче за потоком, тобто безпосередньо попередній та наступний оператори. Подивимося, як ми можемо ланцюжком операторів під час обробки запиту HTTP-URL із фреймворком Combine Swift. Зверніть увагу на приклад 3.5, нижче:

```
    // 1
    let url = URL(string:
"https://api.github.com/users/kikiwora/repos")!

    // 2
    let token = URLSession.shared.dataTaskPublisher(for:
url)
        .map { $0.data } // 3
        .decode(type: [Repository].self, decoder:
JSONDecoder()) // 4
        .sink(receiveCompletion: { completion in // 5
            print(completion)
        }, receiveValue: { repositories in
            print("kikiwora has \(repositories.count)
repositories")
        })
```

Приклад 3.5 – Ланцюжок викликів Combine при зверенні до GitHub

3.5 On-Device Machine Learning

Машинне навчання на мобільних з кожним роком стає все популярнішим! WWDC 2019 подарував нам багато нових смаколиків для додавання ML в наші додатки. Одним з найбільших оголошень було те, що Core ML 3 зараз підтримує навчання моделей на iPhone та iPad. Хто б подумав кілька років тому, що ми будемо тренувати конвенти на своїх кишенькових пристроях! Мета цих нових API - дозволити точну настройку існуючої моделі на особистих даних користувача.

Це точно не для використання iPhone як заміни для оренди великого сервера з графічними процесорами NVIDIA Tesla для навчання величезних моделей з нуля. GPU вашого iPhone потужний, але не такий потужний. Подумайте, що тренінг Core ML 3 є формою трансферного навчання або навіть навчання в Інтернеті, де ви лише трохи налаштуєте існуючу модель. Наприклад, Face ID використовує ці методи, щоб дізнатись, як виглядає власник телефону, і щоб оновити свою модель, коли їх обличчя з часом змінюється (рости бороду, носити інший макіяж, старіти тощо). Ідея полягає в тому, щоб почати з загальної моделі, яка працює в порядку для всіх, а потім зробити копію, налаштовану для кожного користувача.

Це не те саме, що об'єднане навчання, де оновлюється єдина модель на основі - анонімізованих - даних багатьох користувачів. Федеративне навчання - це спосіб провести розподілене навчання «на межі». Замість того, щоб тренуватися на централізованих серверах, він використовує пристрої тисяч або навіть мільйонів користувачів для поширення робочого навантаження. Федеративне навчання також використовує перевагу навчання власних даних користувача, зберігаючи їх приватними, але це не створює персоналізовану модель для кожного окремого користувача.

Просто, щоб усунути будь-яку потенційну плутанину, перш ніж продовжувати: навчання на пристрої абсолютно не пов'язане з Create ML.

Create ML - це модельний навчальний додаток Apple для macOS. Це чудово для швидкої побудови простих моделей, таких як класифікатори зображення / звуку / тексту. Як і Turi Create, інший навчальний інструмент Apple, він використовує трансферне навчання, щоб прискорити час навчання. Create ML та Core ML були розроблені для спільної роботи: Після навчання ви можете безпосередньо зберегти свою модель у форматі Core ML. Не потрібно спочатку запускати інструмент перетворення. Існує також CreateML.framework, який дозволяє тренувати моделі зі сценарію Swift або Playground. Але звісно ж, ви можете й натренувати модель, скажімо, в PyTorch а потім конвертувати її в формат Core ML.

Однак важливо усвідомити, що тренінг на пристрої взагалі не використовує Create ML. Персоналізація моделі на пристрої завжди виконується за допомогою Core ML, а не Create ML. Це може здатися дублюванням функціональності, але обидва фреймворки тренуються дещо по-різному. Create ML корисно для побудови базової версії моделі, використовуючи якомога більше даних, тоді як персоналізація Core ML на пристрої призначена для незначного налаштування цієї моделі з використанням порівняно мало даних користувача. Для того, щоб налагодити модель, вам вже потрібно мати навчену модель.

Моделі машинного навчання починають життя, нічого не знаючи. Мозок непідготовленої моделі - вивчені параметри або ваги - складається з випадково вибраних чисел. Якщо ви запитаєте таку непідготовлену модель для прогнозування, вона просто зробить випадкові здогадки. Щоб отримати обґрунтовані прогнози, потрібно спочатку навчити модель.

Часто ваші користувачі матимуть дуже схожі дані за широкими штрихами, але вони відрізняються деталями. У такому випадку має сенс використовувати персональну персоналізацію на пристрої, щоб адаптувати модель до конкретного використання кожного користувача, що я і роблю у своєму проекті - для цього потрібно надати навчену модель, яка вже розуміє дані в загальних рисах, навіть якщо вона не надто точна. Це основна модель, яка постачається разом із додатком. Навчання на пристрої потім може бути використане для того, щоб модель дізнавалася нові речі лише про цього користувача та його дані, значно збільшуючи точність прогнозів та висновків.

Візьміть фотографії як приклад: фотоальбоми більшості людей містять зображення, які дуже схожі. Це здебільшого фотографії людей чи домашніх тварин чи повсякденні речі - на відміну, скажімо, від мікроскопічних зображень чи рентгенівських променів. Тож якщо ваша модель розраховує працювати над фотографіями - наприклад, розпізнавати домашніх тварин і позначати їх своїм іменем - ви надасте базову модель, яка навчається на самих різних зображеннях домашніх тварин, так що вона вже розуміє, який вид речей, які можна очікувати на таких фото. Або якщо ваша програма призначена для лікарів, ви надасте модель, яка вже навчена розбирати рентген тощо.

Нерозумно було б очікувати, що і наші телефони дізнаються все про фотографії домашніх тварин з нуля. Мало того, що багато дублюючих зусиль, це також вимагатиме багато навчальних даних. Набагато розумніше використовувати вже відпрацьовану модель як вихідну точку та просто трохи її модифікувати для кожного окремого користувача.

Тренування з нуля має сенс, якщо дані кожного користувача абсолютно унікальні. Це має бути на відміну від даних будь-якого іншого користувача вашого додатка. Крім того, дані повинні бути простими (не зображеннями або звуками) і мати відносно невелику кількість, що загалом підходить мені, та необхідне для функції створення нових вправ користувачем. Але як я пояснював у попередньому розділі, якщо дані багатьох користувачів містять подібні зразки, вам краще скористатися заздалегідь підготовленою базовою моделлю, яка вже навчилася розуміти ці загальні закономірності.

Але скажемо, що дані кожного користувача досить різні, що попередня підготовка моделі не допоможе. У цьому випадку навчити унікальну модель для кожного користувача з нуля - це шлях. Про що я тут говорю - це лінійна чи логістична регресія, чи, можливо, деревова модель. Навчання таких моделей на помірній кількості простих векторних даних може бути досить швидким. Однак Core ML в даний час не дозволяє вам тренувати подібні моделі - якщо ви хочете це зробити, вам доведеться прокатати власні.

Припустимо, ви хочете тренувати класифікатор зображень з нуля. По-перше, вам потрібно зібрати тисячі прикладів тренувань - зазвичай це близько 1000 на кожну категорію об'єкта. Ви зв’яжете ці дані за допомогою програми або дозволите користувачеві збирати свої власні. Можливо, ви можете використовувати активність користувача, але пам’ятайте, що вам також знадобляться етикетки для цих навчальних прикладів. Не багато користувачів матимуть терпіння ретельно збирати та коментувати свою активність.

Щоб тренувати модель, оптимізатор багато разів пройде всі приклади навчання. Один прохід через усі приклади називається епохою. Скажімо, у

вас є 5000 прикладів тренувань і ви тренуєтесь протягом 100 епох, тоді за час навчання модель бачить 100 × 5000 = 500 000 прикладів. Якщо припустити, що для опрацювання кожного прикладу моделі потрібно 0,01 секунди, підготовка цієї моделі займає 5000 секунд або майже півтори години.

Мобільні пристрої не створені для роботи з максимальною швидкістю протягом кількох годин. Телефон швидко нагрівається, внаслідок чого теплове дроселювання починає сповільнювати процесори, щоб пристрій не пошкодився. Плюс до цього навантаження на обчислювальну машину швидко розрядить акумулятори. І користувач не зможе користуватися будь-якими іншими програмами в цей час. Ви можете тренуватися у фоновому режимі, але, наскільки я знаю, у цьому режимі додаток використовує процесор приблизно на 10 × повільніше, ніж GPU або Neural Engine. На тренування знадобиться близько 15 або більше годин, а це взагалі не актуально для мобільних пристроїв, а тим паче носимої електроніки.

Як ви можете сказати, тренування нейронної мережі повністю з нуля на мобільному це дуже погана ідея. Але ви можете зробити модель, яка відповідає індивідуальним побажанням або потребам кожного користувача, як це зробив я. Звичайно, що ви можете зробити в хмарі, але це вимагатиме запуску дорогих серверів та надсилання даних користувача через мережу, а для пристроїв носимої електроніки це не допустимо.

Оскільки навчання відбувається на пристрої, дані користувача ніколи не повинні залишати пристрій, захищаючи конфіденційність користувача та радуючи державних регуляторів.

Дійсно використовувати цю техніку лише в тому випадку, якщо невелика кількість тренувань на пристрої є достатньою для покращення моделі - де «краще» може означати щось інше для кожного користувача.

Щоб вдосконалити свою модель на основі зворотного зв’язку всіх користувачів, те, що ви шукаєте, замість цього - це розподілене навчання, де кожен користувач трохи тренується на власному пристрої та за власними даними, але результати агрегуються та об’єднуються у нове головна модель, яка потім надсилається всім користувачам у рамках оновлення програми, як у федеральному навчанні.

Як я вже згадував, нам потрібна перевірена модель. Зазвичай це передбачає якусь нейронну мережу. Навіть для k-Nearest Neighbors (k-NN) зазвичай вам пропонується pipeline, який складається з нейронної мережі для feature extraction, за яким слід класифікатор k-NN.

Перевірена модель - це лише звичайний файл ML MLmodel, який налаштований так, щоб дозволити оновлення. Це передбачає наступне:

- модель позначена як оновлена;
- модель має спеціальні навчальні матеріали, які надають приклади та основні цілі для навчання.

Для нейронних мереж mlmodel також має:

- будь-які шари, які слід навчити, також позначаються оновленими;
- loss function (перехресна ентропія або середня помилка в квадраті);
- оптимізатор (SGD або Adam);
- значення за замовчуванням для (гіпер) параметрів, таких як кількість епох.

Якщо ваша нейронна мережа мала, ви б не встановили всі шари, які можна відслідковувати. Як правило, слід відслідковувати лише останні шари. Інакше навчання займе занадто довго.

Під час перетворення вашої моделі у формат Core ML за допомогою coremltools ви можете передати конвертору respect_trainable параметр, і він автоматично зробить модель оновлюваємою. Але ви також можете внести ці зміни в існуючий файл mlmodel згодом, це займе трохи більше зусиль.

В даний час Core ML підтримує навчання таких типів моделей:

- k-Nearest Neighbor classifiers;
- Neural Networks (classifier, regressor, or general purpose).

Можна також тренувати ці моделі, якщо вони є частиною трубопроводу, але лише якщо це остання модель в трубопроводі. Інші типи моделей, такі як лінійна регресія або дерева рішень, не можуть тренувати Core ML.

Примітка. Якщо ви хочете тренувати лінійну чи логістичну регресію, ви можете побудувати дуже основну одношарову нейронну мережу для виконання регресії та тренування цього.

Для нейронних мереж можна навчати лише такі типи шарів:

- convolution layers;
- fully-connected layers.

Підтримується зворотне розповсюдження (Backpropagation) багатьох інших типів шарів, але не всіх. Тренуватися можуть лише шари, що слідують за таким непідтримуваним шаром, але не будь-який з шарів, що передували йому. Я впевнений, що майбутні версії Core ML дозволять тренувати і багато інших типів шарів. Типи шарів, які мають вагу, але які ще не піддаються навчанню, включають: batchnorm, вбудовування, зміщення / масштаб та шари RNN, такі як LSTM або GRU.

Core ML 3 пропонує обмежений вибір loss functions:

- categorical cross-entropy;
- mean squared error (MSE).

Модель з декількома виходами також може мати кілька loss function. Наприклад, у моделі, яка прогнозує як ярлик класу, так і обмежувальне поле, ви б використовували перехресну ентропію для виводу мітки класу та MSE для обмежувального поля. Однак наразі немає можливості зважити ці втрати, щоб один налічував більше, ніж інший. Також немає способу визначення власних функцій втрат.

На даний момент доступні такі оптимізатори:

- plain SGD;
- Adam.

Ви також визначаєте гіперпараметри для цих оптимізаторів у файлі mlmodel, такі як learning rate, momentum, mini-batch size, тощо. Існує лише одна швидкість навчання, яка використовується для всіх оновлених шарів. Не велика справа, але більш розширені навчальні пакети дозволяють використовувати різні курси навчання для різних (груп) рівнів. Це дає змогу оновлювати більш загальні шари з дуже невеликою швидкістю навчання, а шари, більш конкретні для вашого завдання, з більшою швидкістю навчання.

Як очевидно, Core ML 3 ще не є заміною для TensorFlow або PyTorch. Але навіть маючи ці обмеження, він пропонує нові цікаві можливості того, що ми можемо зробити з машинним навчанням на наших пристроях!

Велика перевага k-NN полягає в тому, що ви можете додавати нові класи за бажанням, тоді як за допомогою нейронної мережі кількість підтримуваних класів фіксується всередині mlmodel і його потрібно визначати заздалегідь (це залежить від кількості нейронів в останньому шарі).

k-NN також потрібно дуже мало прикладів для навчання, щоб дізнатися про новий клас, що також дуже добре підходе для задач мого додатку – користувачі повинні мати можливість додавання нових класів активностей без значних часових затрат.

Вхід до класифікатора k-NN - це одновимірний вектор чисел, вектор функції. Якщо ваші дані вже подібні, ви можете використовувати k-NN безпосередньо на даних. Якщо ні, то вам потрібно буде запустити якийсь екстрактор функцій, щоб спершу перетворити свої дані в такі вектори. Очевидно, що потік даних з сенсорів може будти перетворений на одномірний. Хоча кожна модель персоналізована для свого користувача, всі вони мають спільний екстрактор функцій, який був попередньо навчений на зображеннях багатьох ескізів, так що модель вже має деякі попередні знання, завдяки попередніх данних користувача.

Якщо це додаток поставляється без кваліфікованого екстрактора функцій, модель не буде працювати дуже добре, оскільки спочатку доведеться навчитися перетворювати вектор даних з усіх сенсорів у векторний вектор - і це виходить за межі можливостей класифікатора k-NN.

З боку Core ML, модель від Turi - це так званий NeuralNetworkClassifier. Це особливий тип нейронної мережі, який завжди повинен передбачати мітку класу. Однак ми хочемо використовувати нейронну мережу в якості екстрактора функцій, тому вона повинна бути типу NeuralNetwork - без частини "класифікатора". Зверніть увагу на рисунок 3.7, на сторінці 58.

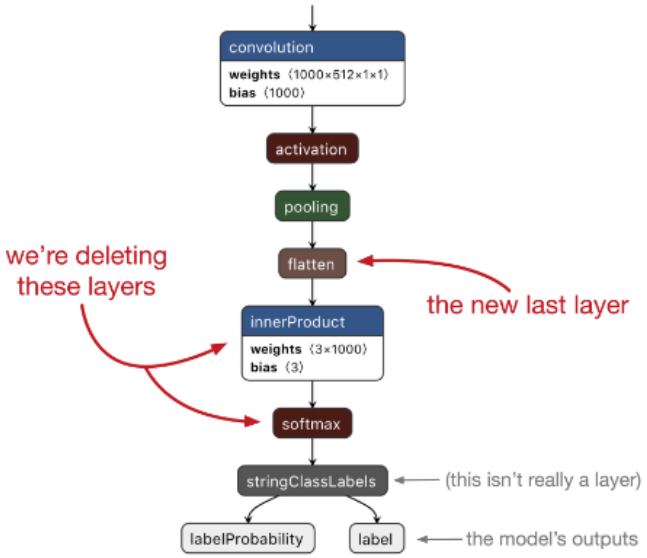

Рисунок 3.7 – Приклад використання конвеєру для перетворення даних з сенсорів у одномірний вектор

Це можна зробити, видаливши шари з base_spec.neuralNetworkClassifier і призначивши їх base_spec.neuralNetwork, як це вказано у прикладі 3.6, нижче:

```
preprocessing =
base_spec.neuralNetworkClassifier.preprocessing

del base_spec.neuralNetworkClassifier.layers[:]
base_spec.neuralNetwork.layers.extend(layers)

base_spec.neuralNetwork.preprocessing.extend(preprocessing
)

base_spec.description.ClearField("metadata")
base_spec.description.ClearField("predictedFeatureName")
base_spec.description.ClearField("predictedProbabilitiesNa
me")
```

Приклад 3.6 – видалення шарів

Модель NeuralNetworkClassifier завжди має два виходи: передбачувана мітка класу та словник з усіма ймовірностями. Оскільки ми видалили шар softmax і перетворили це на звичайну нейронну мережу, ми

повинні сказати моделі, що тепер він має лише один вихід, векторний елемент 1000 елементів, як це вказано у прикладі 3.7, нижче:

```
    del base_spec.description.output[:]

    output = base_spec.description.output.add()
    output.name = "features"
    output.type.multiArrayType.shape.append(1000)

    import coremltools.proto.FeatureTypes_pb2 as ft
    output.type.multiArrayType.dataType =
ft.ArrayFeatureType.FLOAT32

    base_spec.neuralNetwork.layers[-1].output[0] =
"features"
```

Приклад 3.7 – Модель має лише один вхід

Це видаляє старі виходи з моделі та додає новий вихід з назвою «характеристики». Зауважте, що тип даних цього виводу є FLOAT32, а не DOUBLE, оскільки k-NN Core ML очікує, що вектори функцій будуть складені з 32-бітових поплавців. Останній рядок з'єднує існуючий flatten шар з новими висновками «особливостей». І це все для першої моделі в нашому конвеєрі. Ми взяли класифікатор, зняли останні кілька шарів і перетворили його в екстрактор функцій, який виводить вектор 1000 елементів.

Якщо б у нас вас вже була навчена модель k-NN, ми могли б використати для цього інструмент конвертора scikit. Але наша модель k-NN спочатку буде повністю непідготовлена, тому ми будемо використовувати KNearestNeighborsClassifierBuilder, як вказано в прикладі 3.8, на сторінці 60

```
from coremltools.models.nearest_neighbors import \
        KNearestNeighborsClassifierBuilder

knn_builder = KNearestNeighborsClassifierBuilder(
        input_name="features",
        output_name="label",
        number_of_dimensions=1000,
        default_class_label="???",
        number_of_neighbors=3)
```

Приклад 3.8 – Використання KNearestNeighborsClassifierBuilder

Ключовим гіперпараметром k-NN є k, або number_of_neighbors, як його називають coremltools. Тут ми використовуємо трьох сусідів, але на практиці ви експериментуєте з цим налаштуванням, щоб знайти найкраще для ваших даних.

Замість того, щоб завжди використовувати фіксовану величину для k, Core ML також дозволяє вам бути гнучкими, як це вказано у прикладі 3.9, нижче:

```
knn_builder.set_number_of_neighbors_with_bounds(3,
allowed_range=(1, 10))
```

Приклад 3.9 – Власні параметри для k

Це дозволяє розробнику програми змінювати кількість сусідів під час виконання з будь-якого місця між 1 і 10, за замовчуванням 3. Це дійсно все, що потрібно зробити, щоб створити непідготовлену модель k-NN.

Якщо ви вже хотіли додати кілька прикладів до моделі, тепер можете зателефонувати builder.add_samples () зі своїми навчальними даними та мітками. Але ми цього не робимо для цього додатка, k-NN залишимо порожнім.

Додамо кілька описів до входів і виходів моделі k-NN, як це вказано у прикладі 3.10:

```
    knn_spec = knn_builder.spec
    knn_spec.description.input[0].shortDescription       =
"Input vector"
    knn_spec.description.output[0].shortDescription      =
"Predicted label"
    knn_spec.description.output[1].shortDescription = \
            "Probabilities for each possible label"

    knn_spec.description.predictedProbabilitiesName      =
"labelProbability"
    knn_spec.description.output[1].name = \

knn_spec.description.predictedProbabilitiesName
```

Приклад 3.10 – Додавання дескріпторів

Оскільки це модель класифікатора, вона має два виходи: мітка для передбачуваного класу та словник, який має ймовірність для всіх класів. Останнє було автоматично додано KNearestNeighborsClassifierBuilder, але воно має за замовчуванням назву «labelProbs». Ми змінюємо цю назву тут на «labelProbability», оскільки це також ім'я виводу, яке використовуються моделями нейронної мережі з цього додатку. (Це дозволяє нам використовувати один і той же код для оцінки обох типів моделі)

Тепер у нас є дві наші моделі, екстрактор функцій та модель k-NN. Давайте зведемо їх у конвеєр, як у прикладі 3.11:

```
    pipeline_spec = coremltools.proto.Model_pb2.Model()

    pipeline_spec.description.input.extend(base_spec.desc
ription.input[:])
    pipeline_spec.description.output.extend(knn_spec.desc
ription.output[:])

    pipeline_spec.description.predictedFeatureName = \
```

```
          knn_spec.description.predictedFeatureName
     pipeline_spec.description.predictedProbabilitiesName
= \
          knn_spec.description.predictedProbabilitiesName

     pipeline_spec.pipelineClassifier.pipeline.models.add(
).CopyFrom(base_spec)
     pipeline_spec.pipelineClassifier.pipeline.models.add(
).CopyFrom(knn_spec)
```

Приклад 3.11 – Новий Pipeline

Також ми копіюємо властивості predvitedFeatureName та predvictedProbablesName з k-NN моделі. Оскільки ми робимо класифікатор, нам потрібно скопіювати обидві моделі у властивість spec's pipelineClassifier. (Є також конвеєр конвеєра та звичайний конвеєр, але це неправильний вибір для цього додатка.) Також може бути корисно вказати моделі всередині імен pipeline, тому ми можемо з легкістю звернутися до них пізніше. Якщо цього не зробити, моделі просто називаються «model0», «model1» тощо.

То навіщо використовувати нейронну мережу? Ну, це набагато потужніше, ніж k-NN, і може дати кращі результати. k-NN насправді нічого не «вивчає» - це просто запам'ятовує - і тому воно може не узагальнювати, а також тип моделі, який справді здатний до навчання. Моделі k-NN також не дуже надійні щодо галасливих даних, або коли вони існують, або коли дані містять неактуальні функції. Нейронні мережі набагато краще в такому роді. Незважаючи на те, що k-NN в цілому також може бути використаний для регресії, в Core ML це лише класифікація. Якщо ваша модель не є класифікатором, вам потрібно буде використовувати нейронну мережу.

4 НЕОМОРФІЧНИЙ ДИЗАЙН ТА SWIFT UI

Неоморфний дизайн - це найцікавіша тенденція дизайну останніх місяців, хоча компанія Apple, використовувала його як мотив дизайну ще в WWDC18, тобто у 2018 році. Але саме з 2020го року він стане основним трендом у новому дизайні. Як ви можете зрозуміти із назви, він об'єднує старий скеоморфізм (дизайн, що оснований на зовнішньому вигляді реальних речей що ми використовуємо – записників, нотаток, механізмів, кнопок, тощо) та новий модернізм (плоского, здебільш аскетичного та мінімалістичного, котрий зараз можна зустріти повсюди – у Windows 10, iOS 13, Android 10, на багатьох сучасних сайтах).

У цьому розділі ми розглянемо, як саме я створюю неоморфні конструкції за допомогою SwiftUI, чому саме цього я хочу, і - найголовніше - як необхідно модифікувати проект, щоб зробити його більш доступними, адже неоморфізм, іноді його називають неоморфізмом, має серйозні наслідки для доступності, за рахунок зниження контрастності.

Перш ніж ми розберемося з кодом, я просто хочу коротко торкнутися двох основних принципів цієї тенденції дизайну, тому що вона буде актуальною в міру просування:

- він використовує виділення та тіні для визначення форм предметів на екрані;
- контрастність, як правило, знижується;
- повний білий або чорний кольори не використовуються, саме це дозволяє виділити світлі і тіні;
- кінцевим результатом є «екструдований м'який пластик» - дизайн інтерфейсу користувача, який, безумовно, виглядає свіжо та цікаво, не маючи жорстких поглядів.

Я не можу досить повторити, що зменшення контрасту та покладання тіней на фігури серйозно впливає на доступність, і ми повернемося до цього пізніше. Однак я все-таки вважаю, що варто витратити час на вивчення неоморфізму в SwiftUI - навіть якщо ви не використовуєте його у власних додатках, це трохи схоже на ката (паттерни) кодування, що допоможе відточити ваші навички.

Найпростіший вихідний пункт - побудувати неоморфну карту: округлий прямокутник, який буде містити певну інформацію. Звідти ми розглянемо, як ми можемо перенести його в інші частини SwiftUI, але принципи залишаються тими ж, оскільки сам Swift UI, звісно ж, є універсальним інструментом, було б дуже дивно якби ми були б змушені робити усе це кожен раз наново. До речі, в новій версії SwiftUI Xcode з'явилась функція попереднього перегляду, я рекомендую активувати його одразу, оскільки це значно прискорює процес розробки інтерфейсів, бо замість повної зборки додатку кожен раз при зміні параметрів UI ви одразу будете бачити що саме змінилось.

Почнемо з визначення кольору, який буде представляти собою відтінок білого кольору. Це не сірий, а дуже тонкий відтінок, який додає інтерфейсу трохи тепла чи прохолоди. Ми можемо додати це в каталог активів, для більшої гнучкості – можливо в майбутньому ми змінемо параметри кольорів або взагалі усю кольорову палітру, але у прикладі 4.1, нижче, це зроблено одразу в коді

```
extension Color {
    static let offWhite = Color(red: 225 / 255,
green: 225 / 255, blue: 235 / 255)
}
```

Приклад 4.1 – Додаймо Color extension поза структури ContentView

Зверніть увагу – цей колір майже білий, але він достатньо зміщений, щоб користувач зміг побачити повний білий і для нього це виглядало як світіння від об'єму об'єкта, коли нам це потрібно. Тепер ми можемо заповнити тіло для ContentView, надавши йому ZStack, який займає весь екран, і використовуючи наш новий відтінок білого кольору, щоб заповнити весь простір, як зазначено в прикладі 4.2

```
struct ContentView: View {
    var body: some View {
        ZStack {
            Color.offWhite
        }
        .edgesIgnoringSafeArea(.all)
    }
}
```

Приклад 4.2 – Задаймо усьому вікну додатка offWhite Color

Відповідно до нових стандартів дизайну, ми будемо використовувати округлий прямокутник для представлення наших карток, та задаймо розмір картки 300x300, щоб зробити її красивою і зрозумілою на екрані. Отже, додаймо це до ZStack, нижче кольору, як на Прикладі 4.3

```
RoundedRectangle(cornerRadius: 25)
    .fill(Color.offWhite)
    .frame(width: 300, height: 300)
```

Приклад 4.3 – Додаймо округлість та розмірність картки

SwiftUI дозволяє нам застосовувати модифікатори кілька разів, що робить неоморфізм легким для досягнення. Нам необхідно додати трохи тіней, і завдяки цим модифікаторам це зробити набагато простіше ніж раніше, за часів UIKit, як на прикладі 4.4 - це зміщення темної тіні вниз та вправо, відносно картки, а світла тінь зміщена вгору та ліворуч. Світлу тінь видно, тому що ми використовували для нашого фону offWhite Color, і картка загалом стає видимою. Ми написали лише декілька рядків коду, але вже маємо неоморфну карту - SwiftUI робить це напрочуд просто! Магія нових технологій. Результат можна бачини на рисунку 4.1

```
.shadow(color: Color.black.opacity(0.2), radius: 10, x: 10, y: 10)
.shadow(color: Color.white.opacity(0.7), radius: 10, x: -5, y: -5)
```

Приклад 4.4 – Додання модифікаторів для тіней на картках

Рисунок 4.1 – Приклад картки неоморфічного дизайну

Щодо елементи інтерфейсу - картки мають досить низький ризик для неоморфізму - доки інтерфейс всередині ваших карт досить контрастний, сама межа картки може легко не існувати взагалі, і ви не вплинете на доступність, бо користувач все одно буде здатний виділити контури ваших карток, хоча без тіней це буде виглядати не так гарно. А от Кнопки - це інша справа, тому що вони існують, щоб їх натискати, не маючи власного контенту всередині, окрім гліфів, тому зменшення їх контрастності може принести більше шкоди, ніж користі.

Але по-перше, нам необхідно створити власний стиль кнопки, саме так SwiftUI дозволяє нам ділитися конфігураціями кнопок у багатьох місцях. Це набагато зручніше, ніж приєднувати безліч модифікаторів до кожної створеної кнопки - ми можемо просто визначити стиль один раз та поділитися ним у багатьох місцях.

Ми будемо визначати стиль кнопки, який фактично порожній: SwiftUI передасть нам мітку для кнопки, яка може бути текстом, зображенням чи чимось іншим, і ми повернемо його немодифікованим, я в прикладі 4.5, нижче.

Наразі config.label є вмістом кнопки, але у наступному прикладі, незабаром, я додам ще більше вмісту. Зверніть увагу на приклад 4.5, на сторінці 68.

```
struct SimpleButtonStyle: ButtonStyle {
    func makeBody(configuration: Self.Configuration)
-> some View {
        configuration.label
    }
}

Button(action: {
    print("Button tapped")
}) {
    Image(systemName: "heart.fill")
        .foregroundColor(.gray)
}
.buttonStyle(SimpleButtonStyle())
```

Приклад 4.5 – Проста Swift UI кнопка без модифікаторів для Neumorphism

Наразі, ми не побачимо нічого особливого на екрані, бо це лише контейнер, без будь-якого власного дизайну, але ми це виправимо, додавши наш неоморфний ефект до стилю кнопок, так, саме той, з тінями. Цього разу ми не збираємося використовувати закруглений прямокутник, тому що для простих піктограм коло виглядає краще, але нам потрібно додати трохи накладки, щоб область натискання кнопки була красивою і великою, а також трохи анімацій для реакції на дії користувача, пізніше. Змінемо метод makeBody () так, щоб додати трохи відступів (padding), а потім розмістить наш неоморфний ефект як фон кнопки, як на прикладі 4.6, а результат цієї процедури на рисунку 4.2, на сотрінці 69.

```
    configuration.label
        .padding(30)
        .background(
            Circle()
                .fill(Color.offWhite)
                .shadow(color: Color.black.opacity(0.2),
radius: 10, x: 10, y: 10)
                .shadow(color: Color.white.opacity(0.7),
radius: 10, x: -5, y: -5)
        )
```

Приклад 4.6 – Додавання форми та тіней для кнопки

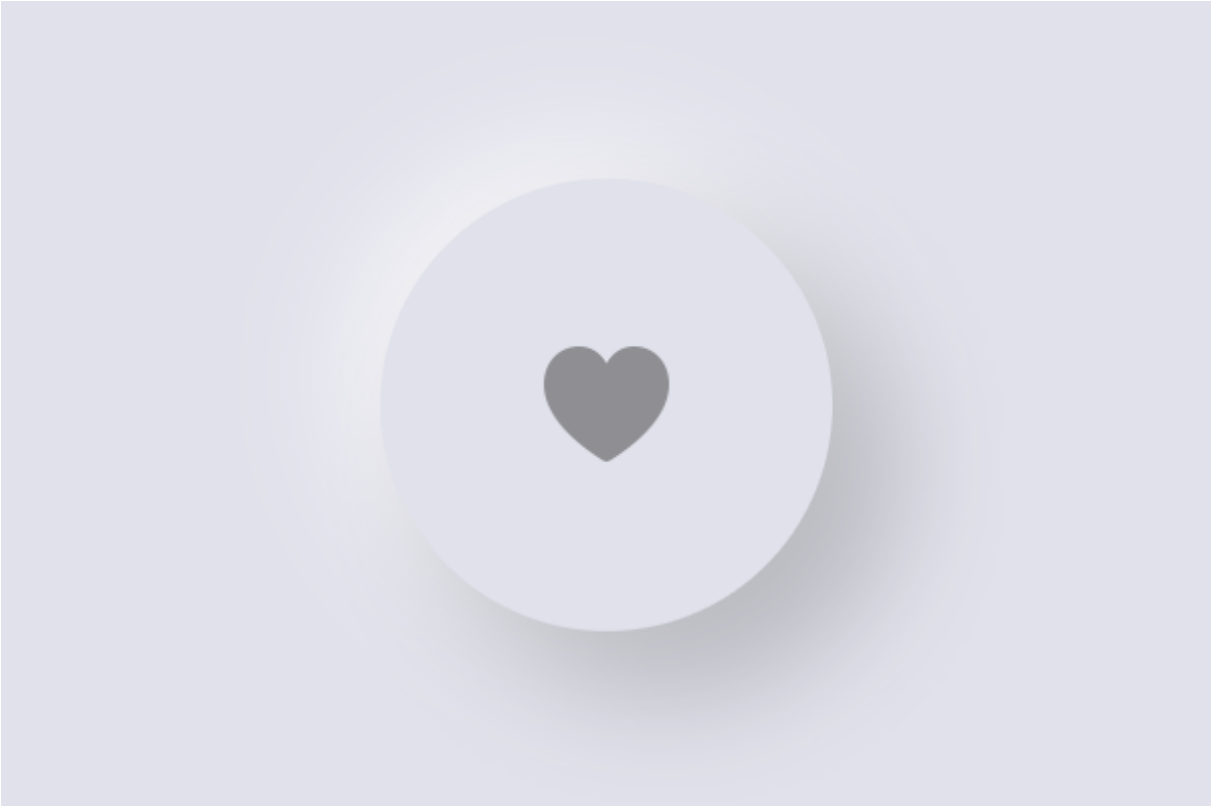

Рисунок 4.2 – Приклад майже готової кнопки неоморфічного дизайну

Це вже значний прогрес до бажаного ефекту, але якщо запустити додаток, ви побачите, що він не веде себе на практиці так, як це очікує користувач, про що я вже писав раніше, - кнопка ніяк не реагує при натисканні візуально, що просто дивно.

Щоб виправити це, нам потрібно прочитати властивість config.isPress всередині нашого власного стилю кнопок, який повідомляє про те, чи натиснута кнопка в даний час чи ні. Ми можемо використовувати це для налаштування стилю, щоб дати певну візуальну індикацію того, чи натиснута кнопка.

Почнемо просто: ми використаємо групу для фону кнопки, потім перевіримо configuration.isPress і повернемо або рівне коло, якщо кнопку натиснуто, або повернемо наше поточне тіньове коло в іншому випадку, що й створить ефект “втискання” кнопки, як у житті, зверніть увагу на приклад 4.7, нижче:

```
    configuration.label
        .padding(30)
        .background(
            Group {
                if configuration.isPressed {
                    Circle()
                        .fill(Color.offWhite)
                } else {
                    Circle()
                        .fill(Color.offWhite)
                        .shadow(color:
Color.black.opacity(0.2), radius: 10, x: 10, y: 10)
                        .shadow(color:
Color.white.opacity(0.7), radius: 10, x: -5, y: -5)
                }
            }
        )
```

Приклад 4.7 – додання ефекту втискання кнопки

Оскільки в стані isPress використовується коло з білим кольором, це робить наш ефект непомітним при натисканні кнопки.

В процесі розробки ПЗ, я з’ясував, що SwiftUI дивно обчислює області, які можна простежувати, випадково зробивши область, яку можна натиснути для нашої кнопки, дійсно невеликою – потрібно було торкнутися самого зображення, а не неоморфного дизайну навколо нього. Щоб виправити це, я додав модифікатор .contentShape (Circle ()) безпосередньо після .padding (30), змусивши SwiftUI використовувати весь доступний простір для крана.

Після цього, має сенс створити підроблений ефект втискання кнопки під поверхню, перевернувши тіні - скопіювавши два модифікатори тіні з регулярного ефекту, а потім обмінявшись білими та чорними значеннями X та Y, наприклад 4.7.

```
if configuration.isPressed {
    Circle()
        .fill(Color.offWhite)
        .shadow(color: Color.black.opacity(0.2), radius:
10, x: -5, y: -5)
        .shadow(color: Color.white.opacity(0.7), radius:
10, x: 10, y: 10)
} else {
    Circle()
        .fill(Color.offWhite)
        .shadow(color: Color.black.opacity(0.2), radius:
10, x: 10, y: 10)
        .shadow(color: Color.white.opacity(0.7), radius:
10, x: -5, y: -5)
}
```

Приклад 4.7 – Створення ефекту втискання кнопки

Але результат не виглядає натурально, дивись на рисунку 4.3, нижче

Рисунок 4.3 – Порівняння звичайного стану та стану натиснутої кнопки

Як ми бачимо, це виглядає не як натиснута кнопка, а як зміна напрямку освітлення (особливості сприйняття мозком світлової інформації щодо форм обэкту на основі освітлення), тому нам необхідно зробити ще деякі зміни. Код працює, але люди трактують ефект по-різному - одні бачать його як увігнуту кнопку, але більшість бачать її як кнопку, яка все ще піднімається, але світло виходить з іншого кута. Краща ідея - створити внутрішню тінь, яка буде набагато більше схожа на натискання кнопки всередину. Це не в комплекті з SwiftUI як стандарт, але ми можемо зробити його досить легко, використовуючи деякі модифікатори. Створення внутрішньої тіні вимагає двох лінійних градієнтів, і це буде перший із багатьох внутрішніх градієнтів, котрі я буду використовувати у цій роботі, тому ми додаймо невелике допоміжне розширення до LinearGradient, щоб спростити створення стандартних градієнтів пізніше. Дивись приклад 4.8:

```
extension LinearGradient {
    init(_ colors: Color...) {
        self.init(gradient: Gradient(colors: colors),
startPoint: .topLeading, endPoint: .bottomTrailing)
    }
}
```

Приклад 4.8 – Extension на створення лінійного градієнту

Маючи це, ми можемо просто надати різноманітний перелік кольорів, щоб повернути лінійний градієнт їх у діагональному напрямку. Тепер важлива частина: замість того, щоб додати до нашого пресованого кола дві перевернуті тіні, ми збираємося накладати поверх неї нове коло, надати йому розмитий хід, а потім замаскувати його іншим колом, який має градієнт.

Це багато, але дозвольте мені пояснити:

- наше базове коло – це неоморфний ефект, який ми маємо зараз, який наповнений нашим білим кольором;
- поміщаємо коло над цим, погладжуємо і трохи розмиваємо, щоб зробити м’який край, що імітує фізичне розповсюдження світла;
- потім маскуємо накладене коло іншим колом, цього разу заповненим лінійним градієнтом.

Коли ви маскуєте одне подання (view) з іншим, SwiftUI використовує альфа-канал маски, щоб визначити, що має бути показано для нижнього перегляду. Отже, якщо ми намалюємо розмитий сірий штрих, а потім замаскуємо його за допомогою лінійного градієнта чорного кольору для очищення, розмитий штрих буде невидимий з одного боку і стає більш видимим з іншого – ми отримаємо плавний внутрішній градієнт. Щоб зробити ефект більш вираженим, ми можемо трохи компенсувати погладжені кола в будь-який бік, і за допомогою невеликого експерименту я виявив, що малювання світлої тіні більш товстою лінією, ніж темна тінь справді допомогло досягти максимального ефекту, що також імітує фізичне розповсюдження світла із життя. Але пам’ятайте, що неоморфізм використовує дві тіні, одну світлу та одну темну, щоб створити відчуття глибини, тому ми додаймо цей ефект внутрішньої тіні двічі різними кольорами. Хоча звісно ж що така кількість об’єктів та накладають збільшені вимоги до пристрою, бо потребує більше ресурсів. Але результат

таких змін того вартий, тим паче що ми не повинні кожен раз прораховувати усе це заново – система має певні оптимізації, зо дозволяють використовувати попередні розрахунки, дивіться приклад 4.9 та рисунку 4.4

```
Circle()
    .fill(Color.offWhite)
    .overlay(
        Circle()
            .stroke(Color.gray, lineWidth: 4)
            .blur(radius: 4)
            .offset(x: 2, y: 2)

.mask(Circle().fill(LinearGradient(Color.black,
Color.clear)))
    )
    .overlay(
        Circle()
            .stroke(Color.white, lineWidth: 8)
            .blur(radius: 4)
            .offset(x: -2, y: -2)

.mask(Circle().fill(LinearGradient(Color.clear,
Color.black)))
    )
```

Приклад 4.9 – Додання більш реалістичного стану втискання кнопки

Рисунок4.4 – Порівняння старого та нового підхідо да стану натиснутої кнопки, через зміну гри тіней та світла для забезпечення більшої реалістичності та покращення сприйняття користувачами

На щастя , компанія Apple має певні норми якості щодо продуктів для своїх платформ, що забезпечує високу стабільність та когерентність системи в цілому, включаю дизайн. І підтримка темної теми є одним із обов'язкових функціоналів що має надавати додаток користувачеві, адже багато користувачів використовують темну тему або постійно, або лише ввечері, для послаблення навантаження на очі. Тому, я просто зобов'язаний імплементувати і цей фунціонал. У Swift UI це зробити просто - додаймо ще два кольори до розширення Color, щоб у нас були деякі постійні значення для роботи, а також використаємо їх як фон для ContentView, замінивши його наявний Color.white, як вказано у прикладі 4.10, нижче

```
static let darkStart = Color(red: 50 / 255, green: 60
/ 255, blue: 65 / 255)
static let darkEnd = Color(red: 25 / 255, green: 25 /
255, blue: 30 / 255)

var body: some View {
    ZStack {
        LinearGradient(Color.darkStart,
Color.darkEnd)
```

Приклад 4.10 – Додавання темних кольорів для підтримки темної теми у додатку

Наш SimpleButtonStyle виглядає неправильно, оскільки він застосовує яскравий стиль на темному тлі. Отже, ми будемо створювати новий темний стиль, який працює краще для наших компонентів, але цього разу розділимо його на два: фоновий вигляд, який ми можемо застосувати будь-де, і стиль кнопки, який охоплює його поряд із формою накладки та модифікатором вмісту. Це дозволить нам отримати більшу гнучкість, як ви побачите пізніше, на рисунку 4.5, на сторінці 76

Новий фоновий вигляд, який ми збираємося додати, дозволить нам задати будь-яку форму для нашого візуального ефекту, тому ми більше не просто прив’язані до кіл. Він також буде відслідковувати, чи намалювати увігнутий наш опуклий ефект (активний чи ні) залежно від властивості isHighlighted, яку ми можемо надсилати ззовні, як це вказано у прикладі 4.11, нижче:

```
struct DarkBackground<S: Shape>: View {
    var isHighlighted: Bool
    var shape: S

    var body: some View {
        ZStack {
            if isHighlighted {
                shape
                    .fill(Color.darkEnd)
                    .shadow(color: Color.darkStart,
radius: 10, x: 5, y: 5)
                    .shadow(color: Color.darkEnd,
radius: 10, x: -5, y: -5)

            } else {
                shape
                    .fill(Color.darkEnd)
                    .shadow(color: Color.darkStart,
radius: 10, x: -10, y: -10)
                    .shadow(color: Color.darkEnd,
radius: 10, x: 10, y: 10)
            }
        }
    }
}
```

Приклад 4.11 – Додання модифікатора тіней

Модифікація там полягає в тому, що розмір тіні зменшується при натисканні кнопки - ми використовуємо відстань у 5 поінтів, а не 10. Потім ми можемо обернути це всередині DarkButtonStyle, який застосовує доповнення та форму вмісту, дивіться приклад 4.1 на сторінці 77.

```
struct DarkButtonStyle: ButtonStyle {
    func makeBody(configuration: Self.Configuration)
-> some View {
        configuration.label
            .padding(30)
            .contentShape(Circle())
            .background(
                DarkBackground(isHighlighted:
configuration.isPressed, shape: Circle())
            )
    }
}
```

Приклад 4.12 – Структура DarkButtonStyle

Нарешті, ми можемо активувати це для нашої кнопки в ContentView, змінивши її модифікатор buttonStyle (): .buttonStyle(DarkButtonStyle())
Отримаємо відповідний результат для темної теми, що зображено на рисунку 4.5, нижче.

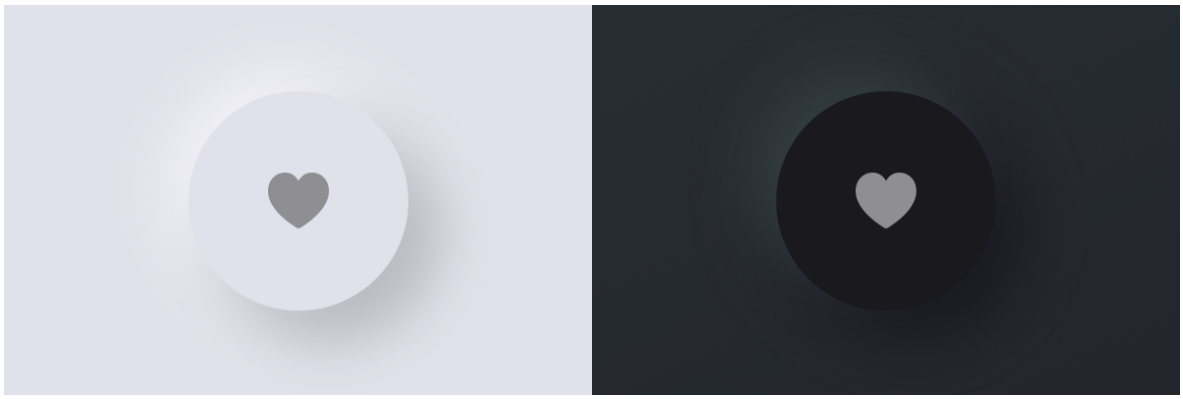

Рисунок 4.5 – Порівняння світлої та темної тем у новій iOS 13

Зараз чудовий час експериментувати з ефектом, адже це допоможе вам зрозуміти, що саме SwiftUI робить для нас. Наприклад, ми могли б створити плавну кнопку, що нагадує пупирку, додавши до кнопки лінійний градієнт і перевернувши його при натисканні, як це вказано на прикладі 4.13 нижче:

```
if isHighlighted {
    shape
        .fill(LinearGradient(Color.darkEnd,
Color.darkStart))
        .shadow(color: Color.darkStart, radius: 10,
x: 5, y: 5)
        .shadow(color: Color.darkEnd, radius: 10, x:
-5, y: -5)

} else {
    shape
        .fill(LinearGradient(Color.darkStart,
Color.darkEnd))
        .shadow(color: Color.darkStart, radius: 10,
x: -10, y: -10)
        .shadow(color: Color.darkEnd, radius: 10, x:
10, y: 10)
}
```

Приклад 4.13 – Створення додаткових ефектів для більшої реалістичності інтерфейсу додатку

Це призведе до того, що під час натискання та відпускання кнопки ви побачите, як зміна статусу кнопки плавно анімується. Нажаль, оскільки ця робота повинна бути роздрукована, я не можу додати анімованого Рисунокдля демонстрації цього ефекту, але можливо зможу продемонструвати деякі зміни на рисунку 4.6, на сторінці 79.

Рисунок 4.6 – додатковий ефект та анімація кнопки у темному режимі

Цей ефект кнопки «подушка» є захоплюючим, але я вважаю що необхідно внести деякі зміни, задля більшої контрастності кнопки. По-перше, незважаючи на те, що це суперечить низько контрастному принципу неоморфного дизайну, я б відкинув сіру піктограму і замість неї пішов з білим кольором, щоб він справді виділявся, оскільки це значно покращить сприйняття темної теми додатку, у прикладі 4.14:

```
Image(systemName: "heart.fill")
    .foregroundColor(.white)
```

Приклад 4.14 – Збільшення контрасту піктограми всередині кнопки

А якщо ми додаємо накладку в натиснутому стані для кнопки, вона не тільки виглядає більше, ніж фізична кнопка, яка натискається плоскою, але також допомагає відрізнити її натиснутий стан від нестисненого. Щоб це сталося, нам потрібно вставити модифікатор overlay () після fill (), коли isHighlight є true, як це приведено у прикладі 4.15 на сторінці 80

```
if isHighlighted {
    shape
        .fill(LinearGradient(Color.darkEnd,
Color.darkStart))

.overlay(shape.stroke(LinearGradient(Color.darkStart,
Color.darkEnd), lineWidth: 4))
        .shadow(color: Color.darkStart, radius: 10,
x: 5, y: 5)
        .shadow(color: Color.darkEnd, radius: 10, x:
-5, y: -5)
```

Приклад 4.15 – Додання модифікатору Overlay

Для ще більш чіткого вигляду ми можемо видалити два модифікатори shadow() для натиснутого стану, які дійсно фокусують накладення. Ми також можемо додати накладення в стиснутому стані, просто щоб помітити, що це кнопка. Поміщаємо його безпосередньо після fill(), наступним чином, а результат видно на рисунку 4.7, на сторінці 81 та прикладі 4.16, нижче:

```
} else {
    shape
        .fill(LinearGradient(Color.darkStart,
Color.darkEnd))
        .overlay(shape.stroke(Color.darkEnd,
lineWidth: 4))
        .shadow(color: Color.darkStart, radius: 10,
x: -10, y: -10)
        .shadow(color: Color.darkEnd, radius: 10, x:
10, y: 10)
}
```

Приклад 4.16 – Оновлення модифікатору fill()

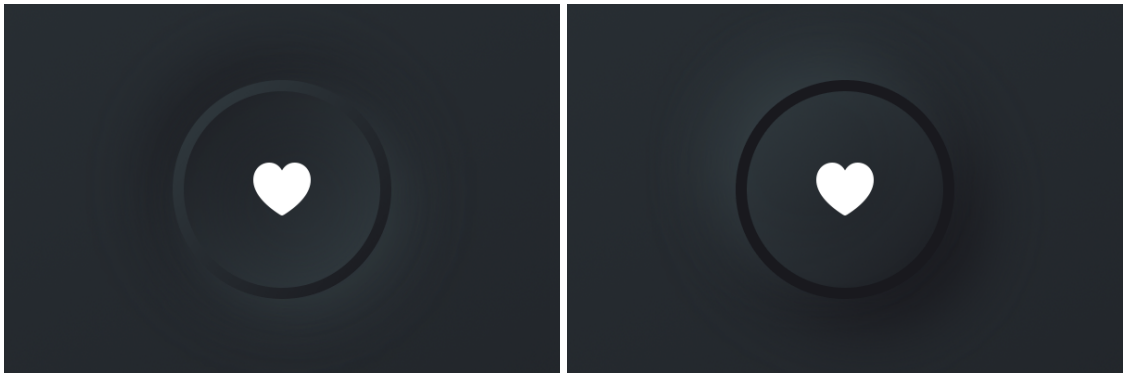

Рисунок 4.7 – Альтернативний, більш контрастний вигляд

Однією з переваг розділення нашого стилю кнопок від стилю фонового неоморфного є те, що тепер ми можемо додати стиль перемикання, використовуючи той самий ефект. Це означає створити нову структуру, що відповідає протоколу ToggleStyle, який схожий на ButtonStyle, за винятком:

- ми повинні прочитати config.isOn, щоб визначити, включений чи ні перемикач;
- нам потрібно надати кнопку для самої дії переключення або хоча б якусь програму OnTapGesture () або подібну.

Я додав наступну структуру до свого проекту, дивись приклад 4.17, нижче:

```
struct DarkToggleStyle: ToggleStyle {
    func makeBody(configuration: Self.Configuration) ->
some View {
        Button(action: {
            configuration.isOn.toggle()
        }) {
            configuration.label
                .padding(30)
                .contentShape(Circle())
        }
        .background(
            DarkBackground(isHighlighted:
configuration.isOn, shape: Circle())
        )
    }
}
```

Приклад 4.17 – Структура DarkToggleStyle

Також, необхідно додати @State private var isToggled = false та загорнути існуючу кнопку в VStack з проміжком 40, і розмістіть її як вказано нижче:

```
Toggle(isOn: $isToggled) {
    Image(systemName: "heart.fill")
        .foregroundColor(.white)
}
.toggleStyle(DarkToggleStyle())

struct ContentView: View {
    @State private var isToggled = false

    var body: some View {
        ZStack {
            LinearGradient(Color.darkStart, Color.darkEnd)

            VStack(spacing: 40) {
                Button(action: {
                    print("Button tapped")
                }) {
                    Image(systemName: "heart.fill")
                        .foregroundColor(.white)
                }
                .buttonStyle(DarkButtonStyle())

                Toggle(isOn: $isToggled) {
                    Image(systemName: "heart.fill")
                        .foregroundColor(.white)
                }
                .toggleStyle(DarkToggleStyle())
            }
        }
        .edgesIgnoringSafeArea(.all)
    }
}
```

Приклад 4.18 – Додання обробника стану Toggle та модифікація ContentView структури

У мене було достатньо часу, щоб пограти з різними неоморфними стилями, але тепер я хочу вирішити його проблеми: надзвичайна відсутність контрасту означає, що кнопки та інші важливі елементи управління недостатньо виділяються з речей навколо них, що ускладнює користування нашими програмами людям із порушеннями зору. А відповідно до вимог Apple щодо якості програмних продуктів що надають розробники, я ще й повинен це зробити – бо в системі задля більшої доступності існує опція підвищення контрастності у додатках, котру всі додатки зобов'язані поважати та використовувати:

- так, я усвідомлюю, що кнопки Apple за замовчуванням просто виглядають як синій текст, тому не спочатку «виглядають» як кнопки, але вони мають високу контрастність;
- мало сказати, що «це працює для мене, але я можу додати спеціальний варіант, щоб зробити його більш доступним» - доступність не є «приємною річчю», це вимога, тому наші програми повинні бути доступними за замовчуванням, а не ніж «доступний доступ»;
- елементу управління що ми зробили - це все ще кнопки та перемикачі SwiftUI, а це означає, що всі наші зміни не вплинули на їх видимість та функціональність для VoiceOver або інших допоміжних технологій Apple.

Ми вже оновили сірі значки до білого кольору, щоб отримати негайне збільшення контрасту, але кнопки та перемикачі все ж потребують набагато більшої контрастності, якщо вони мають бути більш доступними. Отже, я дослідив, які зміни ми могли б внести, щоб допомогти справді виділитися.

По-перше, я хотів би, щоб ви додали два нових кольори до нашого розширення. По-друге, скопіювати існуючий DarkBackground з визовом до копії ColorfulBackground. По-третє, скопіювати стиль темної кнопки та перемикання стилю, те перейменувати їх на ColorfulButtonStyle та ColorfulToggleStyle, та використовувати в них новий ColorfulBackground для мого нового фону. Виглядатиме це наступним чином, дивись приклад 4.19:

```
static let lightStart = Color(red: 60 / 255, green: 160 /
255, blue: 240 / 255)
static let lightEnd = Color(red: 30 / 255, green: 80 /
255, blue: 120 / 255)

struct ColorfulButtonStyle: ButtonStyle {
    func makeBody(configuration: Self.Configuration) ->
some View {
        configuration.label
            .padding(30)
            .contentShape(Circle())
            .background(
                ColorfulBackground(isHighlighted:
configuration.isPressed, shape: Circle())
            )
            .animation(nil)
    }
}

struct ColorfulToggleStyle: ToggleStyle {
    func makeBody(configuration: Self.Configuration) ->
some View {
        Button(action: {
            configuration.isOn.toggle()
        }) {
            configuration.label
                .padding(30)
                .contentShape(Circle())
        }
        .background(
            ColorfulBackground(isHighlighted:
configuration.isOn, shape: Circle())
        )
    }
}
```

Приклад 4.19 – Створення нових стилів для Toggle-Styled Button

Нарешті, я можу використати усе це в ContentView для нового стилю кнопки, як у прикладі 4.20:

```
Button(action: {
    print("Button tapped")
}) {
    Image(systemName: "heart.fill")
        .foregroundColor(.white)
}
.buttonStyle(ColorfulButtonStyle())

Toggle(isOn: $isToggled) {
    Image(systemName: "heart.fill")
        .foregroundColor(.white)
}
.toggleStyle(ColorfulToggleStyle())
```

Приклад 4.20 – Використання нового стилю в ContentView

Щоб реалізувати нашу різнокольорову версію, ми змінимо модифікатори fill() та overlay() для обох, натиснутого та ненатиснутого, станів. Отже, коли isHighlight є true, ми змінюємо darkStart і darkEnd на lightStart і lightEnd, як у прикладі 4.21, нижче

```
if isHighlighted {
    shape
        .fill(LinearGradient(Color.lightEnd,
Color.lightStart))

.overlay(shape.stroke(LinearGradient(Color.lightStart,
Color.lightEnd), lineWidth: 4))
        .shadow(color: Color.darkStart, radius: 10, x: 5,
y: 5)
        .shadow(color: Color.darkEnd, radius: 10, x: -5,
y: -5)
```

Приклад 4.12 – додання перевірки на isHighlighted

Тепер ми побачимо, що це вже велике поліпшення: натиснутий стан тепер має синій колір, тож користувачеві не може бути не зрозуміло, коли кнопка натиснута або затиснута. Але ми все ще можемо зробити краще - ми можемо додати один і той же колір навколо кнопки, коли її не чіпають, допомагаючи привернути до неї увагу, звісно ж, якщо нам це потрібно. Зробити це дуже просто, дивіться приклад 4.22 нижче, а результат на рисунку 4.8 на сторінці 87:

```
ZStack {
    if isHighlighted {
        shape
            .fill(LinearGradient(Color.lightEnd,
Color.lightStart))

.overlay(shape.stroke(LinearGradient(Color.lightStart,
Color.lightEnd), lineWidth: 4))
            .shadow(color: Color.darkStart, radius: 10, x:
5, y: 5)
            .shadow(color: Color.darkEnd, radius: 10, x: -
5, y: -5)
    } else {
        shape
            .fill(LinearGradient(Color.darkStart,
Color.darkEnd))

.overlay(shape.stroke(LinearGradient(Color.lightStart,
Color.lightEnd), lineWidth: 4))
            .shadow(color: Color.darkStart, radius: 10, x:
-10, y: -10)
            .shadow(color: Color.darkEnd, radius: 10, x:
10, y: 10)
    }
}
```

Приклад 4.22 – Остання версія коду Swift UI для Toggle Button

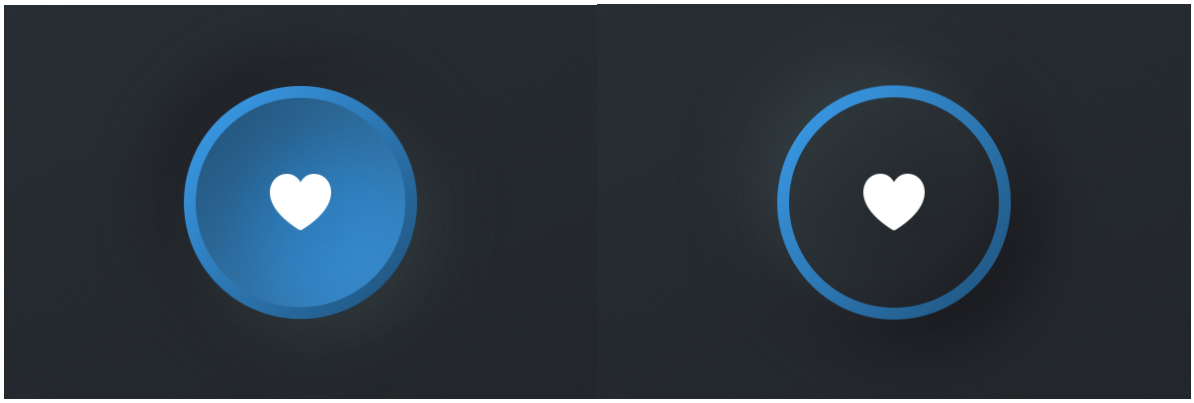

Рисунок 4.8 – Натиснутий та ненатиснутий стани для Toggle Button

Найважливішою плюшкою Swift UI є універсальність та можливість використання вже існуючих SwiftUI Views на усіх платформах Apple, як на iOS, так і на watchOS, tvOS, macOS. Один із слоганів SwiftUI під час WWDC [3] минулого 2019го року був: «Learn one time, use anywhere». SwiftUI - це декларативна база для побудови інтерфейсів користувача на платформах Apple. Але що означає слово декларативний? Це означає, що ви описуєте те, чого хочете досягти, і фреймворки виконують усі необхідні для вас роботи. Залежно від платформи, SwiftUI може виводити вміст по-різному. Результат моєї роботи над додатком дивіться на рисунку 4.9, нижче:

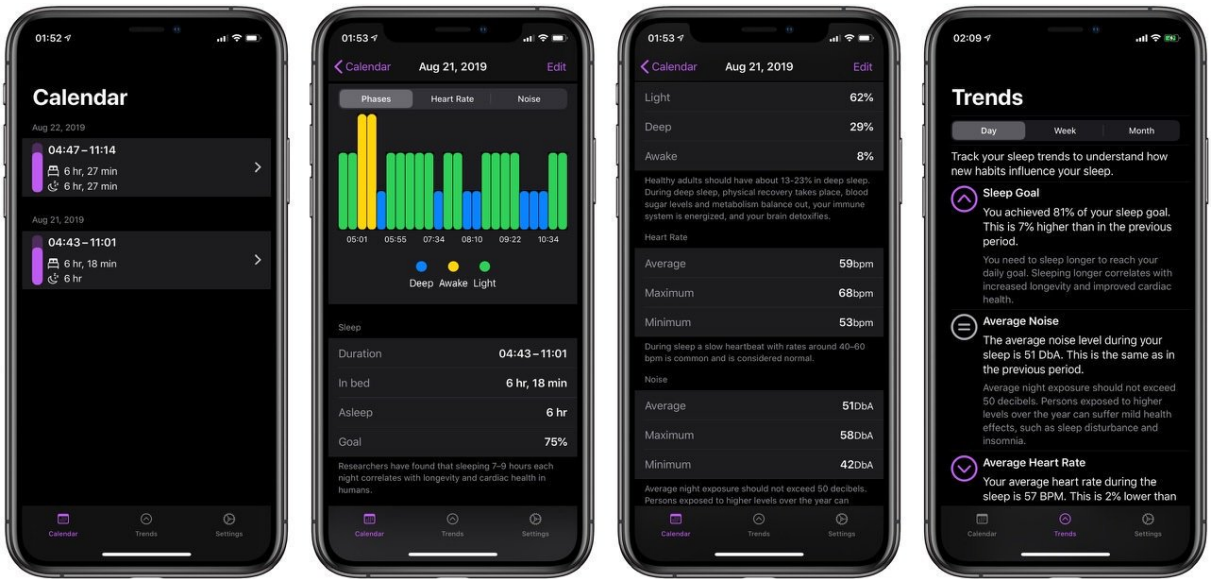

Рисунок 4.9 – Концептуальний дизайн інтерфейсу додатка для iOS

Фактично, цей інтерфейс можливо розділити з watchOS додавши лише декілька простих перевірок на версію системи. Але обмеженням тут є те, що я використовую Swift UI а отже працюватиме це лише на iOS 13 та watchOS 6, але не на попередніх версіях – це обмеження нових технологій. Але і цього мені достатньо, бо iOS 13 підтримується навіть на пристроях котрим вже 5ть років, тобто фактично, ці системи доступні усім. Давайте подивимося на екран деталей календарного дня. У якості кореневого перегляду використовується компонент списку. Всередині списку маємо діаграму та купу розділів з текстовими даними. Щодо того як працює діаграма з даними, я розповім трохи згодом. На цьому екрані також є компонент вибору зі стилем SegmentedPicker, який дозволяє змінювати дані, представлені на графіку. На жаль, watchOS не підтримує стиль SegmentedPicker. Ось чому я вирішив видалити його з програми watchOS, але зберігати його в додатку iOS. Ми можемо умовно включити або виключити код, використовуючи макрос #if у SwiftUI. Давайте подивимось вихідний код мого SleepDetailsView. Дивіться приклад 4.23, нижче

```
    @State private var chart: Chart = .sleepPhases

    private var chartSection: some View {
        VStack {
            #if os(iOS)
            Picker(selection: $chart, label:
Text("chartType")) {
                Text("phases").tag(Chart.sleepPhases)
                Text("heartRate").tag(Chart.heartRate)
            }
            .pickerStyle(SegmentedPickerStyle())
            .labelsHidden()
            #endif

            BarChartView(bars: bars, labelsCount: 5)
                .frame(height: 280)
                .animation(.spring())
        }
    }
```

Приклад 4.23 – Код SleepDetailsView

Як ви бачите в наведеному вище прикладі, ми зберігаємо компонент вибору в ієрархії перегляду лише тоді, коли ми будуємо його для iOS. Компілятор Swift виключить ці рядки під час процесу створення додатків для перегляду. Ще однією проблемою під час адаптації екрана деталей сну для watchOS був NavigationBarItems. watchOS взагалі не підтримує елементи навігаційної панелі. На щастя, я великий шанувальник концепції Container Views, яка дозволяє нам позбутися цієї проблеми.

Container Views - це представлення, яке обробляє потік даних і не надає жодного інтерфейсу користувача. Натомість Container View отримує дані та передає їх у режим перегляду. Дивись приклад 4.24:

```
struct SleepDetailsContainer: View {
    @EnvironmentObject var store: SleepStore

    var body: some View {
        SleepDetailsView(sleep: store.sleep)
              .onAppear(perform: store.fetch)
              .navigationBarItems(trailing:
         EditButton())
    }
}

struct SleepDetailsView: View {
    let sleep: Sleep

    var body: some View {
        Form {
            chartSection
            durationSection
            phasesSection
            heartRateSection
            noiseSection
        }
    }
}
```

Приклад 4.24 – Адаптер у вигляді Container View

У цьому випадку у нас є SleepDetailsView, який повністю сумісний з watchOS. Все, що нам потрібно, - це створити нову версію SleepDetailsContainer для watchOS, в якій не використовуються елементи навігації. Результат використання вже існуючого елемента ми бачимо на рисунку 4.10, нижче.

Рисунок 4.10 – Використання вже існуючого SleepDetailsView через контейнер на watchOS 6

Концепція Container Views - це все ще одна з моїх улюблених моделей, яку я багато використовую під час розробки додатків SwiftUI.

Container Views відповідає лише за data-flow:

- зберігати стан Rendering View;
- отримувати оновлені данні за допомогою ObservableObject;
- обробляти life cycle (onAppear/onDisappear);
- надавати action handlers для Rendering View.

Rendering views відповідають лише за rendering:

- створювати User Interface використовуючи примітиви SwiftUI;
- компонувати User Interface використовуючи інші Rendering views;
- використовувати дані як input для рендеру User Interface та не зберігати будь-якого стану взагалі.

SwiftUI забезпечує протокол Shape, який має функцію single requirement path. Я використовую цю функцію, щоб описати форму всередині наданого прямокутника. Спробуємо скористатися нею для малювання кільця прогресу. Тут ми використовуємо трохи математики для обчислення дуги, яка описує наш прогрес. Протокол форми відповідає протоколу View, тому ми можемо легко використовувати його всередині Stack або Group. Ми також можемо налаштувати малюнок за замовчуванням, встановивши спеціальний StrokeStyle. Дивіться приклади 4.25 нижче та 4.26, на сторінці 92.

```
struct ProgressShape: Shape {
    let progress: Double

    func path(in rect: CGRect) -> Path {
        var path = Path()
        path.addArc(
            center: CGPoint(x: rect.midX, y: 
rect.midY),
            radius: rect.width / 2,
            startAngle: .radians(1.5 * .pi),
            endAngle: .init(radians: 2 * Double.pi * 
progress + 1.5 * Double.pi),
            clockwise: false
        )
        return path
    }
}
```

Приклад 4.25 – Код для створення простого індикатору прогресу

```
struct ProgressView: View {
    let progress: Double

    var body: some View {
        ProgressShape(progress: progress)
            .stroke(Color.blue,
                    style: StrokeStyle(
                        lineWidth: 4,
                        lineCap: .round,
                        lineJoin: .round,
                        miterLimit: 0,
                        dash: [],
                        dashPhase: 0
                    )
        )
    }
}
```

Приклад 4.26 – Додавання StrokeStyle до ProgressView

Тепер ми готові до використання спеціальних фігури, але більшу частину часу достатньо застосувати готові до використання фігури, надані SwiftUI. SwiftUI пропонує такі форми, як «Коло», «Капсула», «Прямокутник» тощо. Ми можемо легко використовувати їх для створення BarChart в SwiftUI. Почнемо з опису моделі даних Bar. Дивись приклад 4.27, нижче:

```
struct Legend: Hashable {
    let color: Color
    let label: String
}

struct Bar: Identifiable {
    let id: UUID
    let value: Double
    let label: String
    let legend: Legend
}
```

Приклад 4.27 – Опис моделі даних для Bar

Як бачите, наша структура барів відповідає протоколу Identifiable. Нам потрібна відповідність (conformance) цьому протоколу Identifiable для використання структури Bar в ForEach. ForEach використовує Identifiable, щоб зрозуміти зміни даних під час процесу візуалізації. Тепер ми можемо намалювати наші bars за допомогою Capsule shape. А також реалізувати BarsView, який використовує HStack для відображення капсул у горизонтальному макеті.

Ще один цікавий факт тут - GeometryReader. Згорнувши будь-який вид всередині GeometryReader, ми можемо отримати доступ до даних геометрії, таких як розмір батьків та відступи з безпечної області (safe area), що надаються його контейнером. Ми використовуємо розмір контейнера, щоб обчислити висоту bars і правильно намалювати їх. Ми також використовуємо тут модифікатор доступності, щоб зробити наші бари більш доступними та зручними для навігації.

Кожна діаграма повинна мати перегляд легенд. Давайте реалізуємо його, використовуючи іншу регулярну форму, надану SwiftUI. Я хочу відобразити легенди як вигляд, який містить невелике кольорове коло ліворуч та заголовок справа. Дивіться приклад 4.28 на сторінці 94

```
struct BarsView: View {
    let bars: [Bar]
    let max: Double

    init(bars: [Bar]) {
        self.bars = bars
        self.max = bars.map { $0.value }.max() ?? 0
    }

    var body: some View {
        GeometryReader { geometry in
            HStack(alignment: .bottom, spacing: 0) {
                ForEach(self.bars) { bar in
                    Capsule()
                        .fill(bar.legend.color)
                        .frame(height: CGFloat(bar.value)
/ CGFloat(self.max) * geometry.size.height)

.overlay(Rectangle().stroke(Color.white))
                        .accessibility(label:
Text(bar.label))
                        .accessibility(value:
Text(bar.legend.label))
                }
            }
        }
    }
}
```

Приклад 4.28 – головна структура BarsView

```
struct LegendView: View {
    private let legends: [Legend]

    init(bars: [Bar]) {
        legends = Array(Set(bars.map { $0.legend }))
    }

    var body: some View {
        HStack(alignment: .center) {
            ForEach(legends, id: \.self) { legend in
                VStack(alignment: .center) {
                    Circle()
                        .fill(legend.color)
                        .frame(width: 16, height: 16)

                    Text(legend.label)
                        .font(.subheadline)
                        .lineLimit(nil)
                }
            }
        }
    }
}
```

Приклад 4.29 – головна структура LegendView

У наведеному вище прикладі 4.29 ми використовуємо форму кола, щоб намалювати кольорові показники кожної легенди. Тепер ми можемо скласти наші BarsView і LegendsView, щоб забезпечити надати користувачеві повний BarChart.

```
struct BarChartView: View {
    let bars: [Bar]

    var body: some View {
        Group {
            if bars.isEmpty {
                Text("There   is   no   data   to   display
chart...")
            } else {
                VStack {
                    BarsView(bars: bars)
                    LegendView(bars: bars)
                        .padding()
                        .accessibility(hidden: true)
                }
            }
        }
    }
}
```

Приклад 4.30 – головна структура BarChartView

У прикладі 4.30 ми маємо BarChartView, який розміщує BarsView і LegendsView всередині VStack. Ми також використовуємо модифікатор доступності, щоб приховати LegendsView у дереві доступності, задля забезпечення Open-Closed Principle в SOLID

```
    struct LabelsView: View {
        let bars: [Bar]
        let labelsCount: Int

        private var threshold: Int {
            let threshold = bars.count / labelsCount
            return threshold == 0 ? 1 : threshold
        }

        var body: some View {
            HStack {
                ForEach(0..<bars.count,  id:  \.self)  {
index in
                    Group {
                        if index % self.threshold == 0 {
                            Spacer()
                            Text(self.bars[index].label)
                                .font(.caption)
                            Spacer()
                        }
                    }
                }
            }
        }
```

Приклад 4.31 – Використання BarChartView на реальних даних

Приклад 4.31 - це лише одна із Views що використовуються в моєму додатку, та навіть вона на разі хоча і є повністю функціональную, але ще не відповідає новим стандартам дизайну. Результати такого поступового вкладання простих view ви можете бачити на рисунку 4.11 на сторінці 98, а приклад коду – у прикладі 4.32 на сторінці 99

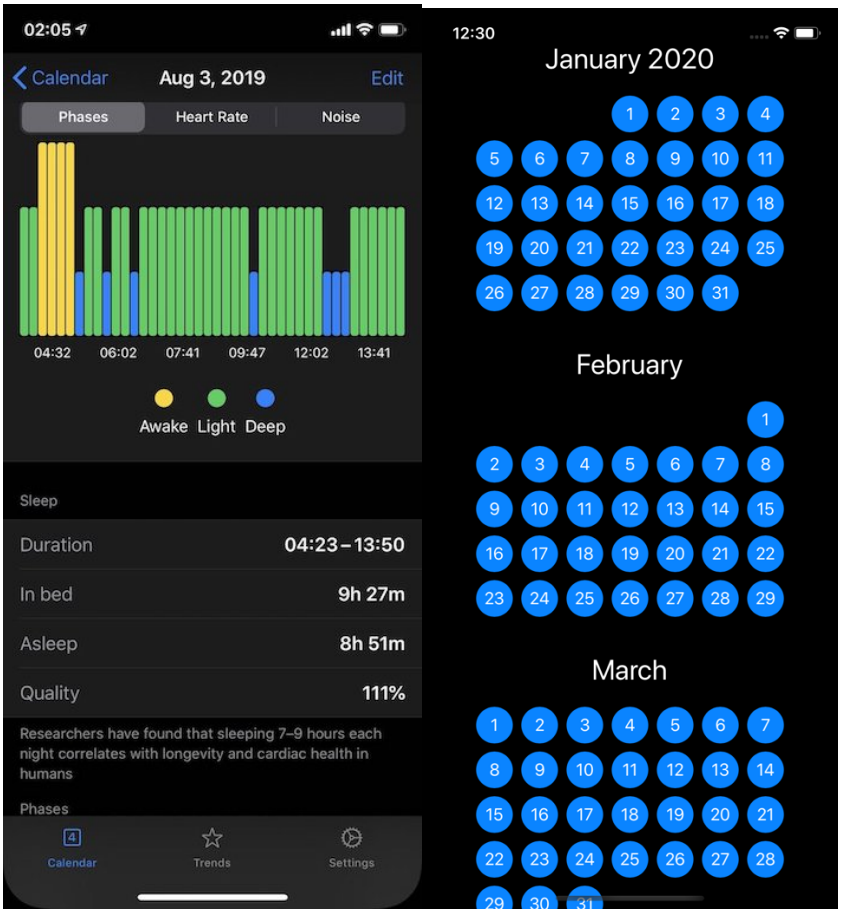

Рисунок 4.11 – Використання графічних примітивів Swift UI для створення власного інтерфейсу

```
struct CalendarView<DateView>: View where DateView: View {
    @Environment(\.calendar) var calendar

    let interval: DateInterval
    let content: (Date) -> DateView

    init(
        interval: DateInterval,
        @ViewBuilder content: @escaping (Date) -> DateView
    ) {
        self.interval = interval
        self.content = content
    }

    private var months: [Date] {
        calendar.generateDates(
            inside: interval,
            matching: DateComponents(day: 1, hour: 0,
minute: 0, second: 0)
        )
    }

    var body: some View {
        ScrollView(.vertical, showsIndicators: false) {
            VStack {
                ForEach(months, id: \.self) { month in
                    MonthView(month: month, content:
self.content)
                }
            }
        }
    }
}
```

Приклад 4.32 – Базова структура CalendarView

Swift UI чудово дозволяє компонувати вже існуючі обєкти для створення нових, тим самим породжуючи ієрархію View, що і дозволяє будувати складні інтерфейси із дуже простих компонентів. Це забезпечує високу спроможність до тестування, надійність та легкість модифікації та створення нових, але схожих на попередників об’єктів, як ми бачимо це на прикладі 4.33, на сторінці 100

```
struct WeekView<DateView>: View where DateView: View {
    @Environment(\.calendar) var calendar

    let week: Date
    let content: (Date) -> DateView

    init(
        week: Date,
        @ViewBuilder content: @escaping (Date) -> DateView
    ) {
        self.week = week
        self.content = content
    }

    private var days: [Date] {
        guard
            let weekInterval = calendar.dateInterval(of:
.weekOfYear, for: week)
            else { return [] }
        return calendar.generateDates(
            inside: weekInterval,
            matching: DateComponents(hour: 0, minute: 0,
second: 0)
        )
    }

    var body: some View {
        HStack {
            ForEach(days, id: \.self) { date in
                HStack {
                    if self.calendar.isDate(self.week,
equalTo: date, toGranularity: .month) {
                        self.content(date)
                    } else {
                        self.content(date).hidden()
                    }
                }
            }
        }
    }
}
```

Приклад 4.33 – Функціональна структура WeekView на основі CalendarView

Я неодноразово казав, що дуже важливо слідкувати за доступністю свого додатку, і це означає більше, ніж просто переконатися, що VoiceOver працює з вашим інтерфейсом користувача. Необхідно переконатися, що елементи керування виглядають інтерактивно, що ваші текстові мітки та значки мають достатнє співвідношення контрасту до їх фонів (принаймні 4,5:1, але націлені на 7:1), що відступи є красивими та великими (принаймні 44x44 поінтів), усе це необхідно щоб підказати користувачу котрий елемент за що саме відповідає та що він робить – у цьому полягає суть цього стилю дизайну – усе повинно бути зрозумілим навіть дитині. Розробник додатку повинен ніколи не забувати, що якщо ви пожертвуєте зручністю для фантазії нового дизайнерського тренду, ніхто не виграє, додаток повинен бути доступний всім та бути зручним, приємним до користування.

5 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ

Нажаль, оскільки додаток незавершений, я не можу викласти його до App Store, а тому не маю доступу до достатньо зачної кількості людей аби дослідити його роботу на практиці, на багатьох різних людях, але я певний що модель та алгоритми ще буде необхідно доналаштовувати деякий час, бо завжди буде якийсь невідомий раніше, неврахований фактор, котрий зможе зіпсувати поведінку моделі для певних людей. Наразі, в умовах карантину, я мав доступа лише до своїх показників, до особливостей свого тіла та виконання багатьох різних вправ. Первинний датасет саме так і було здобуто – в умовах домашнього тренування, але, по-перше, ці тренування відносяться до дуже вузької області, а по-друге, на їх основні не можливо зробити будь-яких валідних висновків, бо кожна людина може рухатись по-своєму та допустимі вікна та маркери для її рухів можуть бути зовсім іншими. Що саме виділила модель при тренуванні мені не відомо, хоча я й сприяв її роботі фільтруючи первинний потік данних, тобто ті данні, що не можуть відноситись до тренування через відсутність аткивності.

На практиці, програма здана розпінавати лише декілька вправ та лише у стандартних позиціях, через відсутність нормалізації данних з гіроскопа та акселерометра, а також через відсутність саме в моєму пристрої магнітометра. Але при цьому, висновки щодо наявності активності, пульсу, хоч з оптичного сенсору, хоч з контактного Bluetooth пульсометра на груди, щодо загальної активності та сну досить точні, бо можливо визначити певні стани та фази. Хоча й точність роботи взагалому можливо поставити під сумнів, адже перевіряв роботу додатка я також лише на собі. При цьому у данній роботі я не враховував багато інших датчиків, як от барометр та оксіметр, що дозволяють визначити зміну висоти користувача та зміну

кількості кисню в крові, що також важливо при визначенні стану активності, навантаження, вправ, та стану здоров’я в цілому.
А щодо електрокардіограмми – на разі watchOS API не надає доступу до цього сенсору, та й використовувати його, фактично, можливо лише в статичному стані, тобто, ані під час сну, ані під час вправ чи іншого навантаження, як от біг чи ходьба, ані навіть під час активного відпочинку користувач всеодно не зміг би ним скористуватись.
Хоча доступа до данних з цього сенсору через HealthKIT API як розробник я маю, тому якби я знав більше про кардіологічні параметри людини, я можливо і міг би розширити функціонал додатку, при статичному аналізі даних користувача.

Після завершення аттестаційної роботи та карантину я планую доробити додаток, як за функціоналом, так і інтерфейсом та локалізацію та випустити його у продаж через App Store, що дозволить мені, за дозволом користувачів, зібрати їх данні та відгуки та натренувати модель краще, а також, загальний для всіх список вправ, та створити функціонал що дозволить розповсюджувати такі вправи до інших користувачів, тобто, мережеву складову, що значно підвищить як точність висновків програмного комплексу, так і значно розширить функціонал, а також, додати підтримку інших, не відомих мені медичних датчиків, як додаткову складову вектора, бо я певен що існує багато людей котрим це необхідно.

ВИСНОВКИ

У 2020му році нарешті стало можливим створення додатку для пристроїв носимої електроніки, котрий зміг би не лише використовувати раніше натреновану модель або завантажувати оновлені моделі з серверів, але й тренувати власну модель одразу на пристрої, з, або без відправки даних для оновлення основної моделі на сервер, при низьких (не помітних для користувача) витратах енергії, пам`яті та часу процесора, використовуючи новий фреймворк від Apple – Core ML 3 та сучасні пристрої Apple Watch на watchOS 6, що дозволить користувачу оперувати не лише раніше заданими категоріями, а й створення власних категорій, автоматичного навчання пристрою, класифікації та подальшої кластерізації типів тренувань та вправ, а також правильне їх відношення до показників життєдіяльності, котрі можливо отримати не лише із вбудованих сенсорів, наприклад, частоти серцевих скорочень, а й з підключених до годинника по Bluetooth медицинських пристроїв, наприклад, оксіметра (датчик кисню у крові), а також тонометрів, барометрів, термометрів, та інших.

ДЖЕРЕЛА

1. Why TensorFlow. URL: https://www.tensorflow.org (дата звернення 16.03.2020).
2. Apple Developer. URL: https://developer.apple.com (дата звернення 17.03.2020).
3. World Wide Developers Conference 2019. URL: https://developer.apple.com/wwdc19/ (дата звернення 20.03.2020).
4. Introduction to Core ML 3. URL: https://www.analyticsvidhya.com/blog/2019/11/introduction-apple-core-ml-3-deep-learning-models-iphone/ (дата звернення 24.03.2020).
5. What's New in watchOS 6 SDK. URL: https://developer.apple.com/watchos/whats-new/ (дата звернення 14.03.2020).
6. Blasco, J., Chen, T. M., Tapiador, J., & Peris-Lopez, P. (2016). A survey of wearable biometric recognition systems. ACM Computing Surveys (CSUR), 49(3), 1-35.
   Miotto, R., Wang, F., Wang, S., Jiang, X., & Dudley, J. T. (2018). Deep learning for healthcare: review, opportunities and challenges. Briefings in bioinformatics, 19(6), 1236-1246.
8. Ravì, D., Wong, C., Deligianni, F., Berthelot, M., Andreu-Perez, J., Lo, B., & Yang, G. Z. (2016). Deep learning for health informatics. IEEE journal of biomedical and health informatics, 21(1), 4-21.
9. Patel, M. S., Asch, D. A., & Volpp, K. G. (2015). Wearable devices as facilitators, not drivers, of health behavior change. Jama, 313(5), 459-460.
10. Buke, A., Gaoli, F., Yongcai, W., Lei, S., & Zhiqi, Y. (2015). Healthcare algorithms by wearable inertial sensors: a survey. China Communications, 12(4), 1-12.
11. Haghi, M., Thurow, K., & Stoll, R. (2017). Wearable devices in medical internet of things: scientific research and commercially available devices. Healthcare informatics research, 23(1), 4-15.
12. Bonfiglio, A., & De Rossi, D. (Eds.). (2010). Wearable monitoring systems. Springer Science & Business Media.
13. Mukhopadhyay, S. C. (2014). Wearable sensors for human activity monitoring: A review. IEEE sensors journal, *15*(3), 1321-1330.

14. Pandian, P. S., Mohanavelu, K., Safeer, K. P., Kotresh, T. M., Shakunthala, D. T., Gopal, P., & Padaki, V. C. (2008). Smart Vest: Wearable multi-parameter remote physiological monitoring system. Medical engineering & physics, 30(4), 466-477.

15. Zhang, Q., Zeng, X., Hu, W., & Zhou, D. (2017). A machine learning-empowered system for long-term motion-tolerant wearable monitoring of blood pressure and heart rate with ear-ECG/PPG. IEEE Access, 5, 10547-10561.

16. Özdemir, A. T., & Barshan, B. (2014). Detecting falls with wearable sensors using machine learning techniques. Sensors, 14(6), 10691-10708.

17. Banaee, H., Ahmed, M. U., & Loutfi, A. (2013). Data mining for wearable sensors in health monitoring systems: a review of recent trends and challenges. Sensors, 13(12), 17472-17500.

18. Jeong, E., Oh, S., Kim, H., Park, J., Bennis, M., & Kim, S. L. (2018). Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data. arXiv preprint arXiv:1811.11479.

19. Xie, C., Koyejo, O., & Gupta, I. (2019). SLSGD: Secure and Efficient Distributed On-device Machine Learning. In: Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (pp. 213-228). Springer, Cham.

20. Varma, J. (2019). Integrating UIKit. In SwiftUI for Absolute Beginners (pp. 141-153). Apress, Berkeley, CA.

21. Dobkin, B. H., & Dorsch, A. (2011). The promise of mHealth: daily activity monitoring and outcome assessments by wearable sensors. Neurorehabilitation and neural repair, *25*(9), 788-798.