Міністерство освіти і науки України
Львівський національний університет ім. Івана Франка
Факультет електроніки та компютерних технологій
Кафедра системного проектування

КУРСОВА РОБОТА

на тему:”Дослідження параметрів проектування та реалізації за допомогою UI фреймворків для операційної системи IOS на прикладі програми погоди”

Виконав:
студент групи ФеІм-12с
Витрикуш М.Ю.
Науковий керівник:
доцент Ненчук Т. М.

Львів -2023

АНОТАЦІЯ

Дана робота виконується за темою «Порівняння двох UI фреймворків для операційної системи iOS на прикладі програми погоди». По-перше, це продиктовано проблемою вибору нативного UI-фреймворку для створення мобільного додатку для операційної системи iOS. Багатьом розробникам важко визначитися з вибором того чи іншого інструменту через відсутність чітких прикладів і пояснень про відмінності кожного з них.

Мета роботи - на прикладі мобільного погодного додатку показати всі тонкощі та нюанси кожного з фреймворків, виявивши їх сильні та слабкі сторони. Сама Apple на сьогодні пропонує два фреймворки інтерфейсу користувача - UIKit і SwiftUI.

У цій роботі пояснюються основні відмінності між кожним із фреймворків, порівнюється розмір блоків коду однієї функціональності та описується складність реалізації в кожному з них.

В результаті на основі теоретичних даних було розроблено додаток, код якого наочно показує різницю між кожним із фреймворків.

Ключові слова: мобільна розробка, фреймворк для операційної системи iOS, погодний додаток, порівняння блоків коду, написаного для фреймворку.

ABSTRACT

This work is done on the topic “Comparison of two UI frameworks for the iOS operating system using the example of a weather application”. First, it is dictated by the problem of choosing a native UI framework for creating a mobile application for the iOS operating system. Many developers find it difficult to decide on the choice of a particular tool due to the lack of clear examples and explanations about the differences between each of them.

The purpose of the work is to show all the subtleties and nuances of each of the frameworks using the example of a mobile weather application, identifying their strengths and weak- nesses. Apple itself provides two UI frameworks today - UIKit and SwiftUI.

In this work explain the main differences between each of the frameworks, compares the size of the code blocks of the same functionality, and describes the complexity of imple- mentation in each of them.

As a result, based on theoretical data, an application was developed, the code of which clearly shows the difference between each of the frameworks.

Keywords: mobile development, framework for the iOS operating system, weather applica- tion, comparison of blocks of code written for the framework.

Зміст

Анотація 2

Abstract…….....3

Вступ…….....6

1.Доступні UI фреймворки від компанії Apple.....11

1.1. UIKit.....13

1.2. SwiftUI.....13

1.3. Парадигми програмування.....14

1.3.1. Імперативний стиль програмування.....14

1.3.2. Декларативний стиль програмування…….....15

1.3.3. Відмінність стилів програмування з прикладу …….....15

1.4. Налаштування проекту в різних UI фреймворків…….....16

1.4.1. Налаштування проекту в UIKit….....17

1.4.1.1. Розробка програми у UIKitюю.....19

1.4.2. Налаштування проекту у SwiftUI…….....20

1.4.2.1. Розробка програми на SwiftUI….....21

2.Розробка і порівняння блоків коду в кожному з фрейм-ворків……23

2.1. Написання однакових екранів програми….....24

2.1.1. Створення екрана в UIKit……24

2.1.2. Створення екрана у SwiftUI……..24

3. Розробка додатку…….....30

3.1 Прототип інтерфейсу програми…….....30

3.2 Реалізація програми на UIKit……..31

3.3. Реалізація на SwiftU…….....37

3.4 Підсумкове порівняння технологій SwiftUI та UIKit……...39

Висновок…….....….....….....44

Використані джерела….....….....45

Додатки….....47

Приклад коду програми …….….....48

Вступ

У сучасному світі мобільні пристрої невід’ємна частина життя, яка виконує значно більше функцій, ніж просто забезпечує зв’язок. Вони стали мобільними помічниками, що забезпечують оплату рахунків, зберігають фотографії, музику та особисті дані, і допомагають вирішувати різні завдання. Ринок мобільних пристроїв є величезним і дуже перспективним, тому все більше розробників програмного забезпечення звертають на нього увагу.

Пріоритетом стають повноцінні мобільні додатки, які покращують зручність використання та збільшують рівень взаємодії з користувачем. Однак, не всі мобільні додатки мають цінність для користувачів, і багато з них є лише платформою для реклами. Мобільний додаток повинен не тільки бездоганно виконувати свої функції, але і зацікавлювати користувача своїм зручним інтерфейсом, розумним функціоналом та приємними нововведеннями, щоб забезпечити комфортне користування та відрізнятися від конкурентів.

У кінці червня 2021 року аналітична компанія Strategy Analytics випустила новий звіт, згідно з яким половина всього населення планети вже володіє смартфонами. Таким чином, до третього кварталу 2021 року смартфонами користувалось близько 4 млрд людей. Для досягнення такого високого показника знадобилося 27 років. За прогнозами Strategy Analytics, до 2030 року смартфонами у всьому світі будуть користуватись понад 5 млрд людей.

За даними аналітичної компанії IDC, частка компанії Apple на кінець четвертого кварталу становить більше 22%.

Багато початківців розробників хочуть бути частинами розвитку мобільних технологій, а також написання своїх власних додатків для мобільного пристрою Apple.

У цій роботі мною будуть використовуватися такі терміни, як мова програмування, фреймворк, інтегроване середовище розробки (IDE).

Мова програмування - це комп'ютерна мова, яку використовують програмісти для написання програм, скриптів або інших наборів команд для виконання комп'ютером.

Фреймворк - це програмні продукти, які полегшують створення технічно складних або великих проектів. Фреймворк, як правило, містить лише базові програмні модулі, а всі специфічні для проекту компоненти реалізуються розробником на їх основі.

IDE - Інтегроване середовище розробки (IDE) дозволяє програмістам об'єднувати різні аспекти написання комп'ютерної програми. IDE підвищують продуктивність програмістів за рахунок об'єднання загальних дій по написанню програмного забезпечення в одному додатку: редагування вихідного коду, створення проектів, налагодження та інше. Найпопулярнішими IDE для розробки мобільних додатків для iOS є Xcode та AppCode.

Xcode - це інтегроване середовище розробки (IDE) від компанії Apple для розробки програмного забезпечення для платформи macOS, iOS, iPadOS, watchOS та tvOS. Воно містить в собі все необхідне для створення програм: текстовий редактор, засоби налагодження, дизайнер інтерфейсу, засоби для розгортання програм тощо.

Як вибрати фреймворк для розробки додатка під операційну систему iOS?

Компанія Apple пропонує два UI фреймворки - UIKit та SwiftUI. Цим питанням задаються початківці та досвідчені розробники перед початком розробки додатка.

Перед тим, як починати розробку мобільного додатка, розробник повинен визначитися, для яких платформ буде написано додаток, з якої версії операційної системи він планує випустити цей додаток. Також необхідно зрозуміти, скільки розробник готовий виділити зусиль та витрат на додаток при виборі конкретного фреймворку, адже у кожного з них є свої плюси та мінуси.

Проблема даного дослідження - визначення оптимального доступного UI фреймворку розробки від компанії Apple та вибір його для написання сучасного додатка.

При розробці ми задаємо собі такі питання:

1. Які відмінності мають нативні UI фреймворки від Apple.
2. На що перш за все варто звертати увагу при виборі фреймворку початкуючому розробнику при розробці першого додатка.
3. Які труднощі можуть виникнути при розробці додатку.
4. Наскільки відрізняються блоки написаного одного і того ж додатка на двох фреймворках на прикладі додатка з таблицею.

Опис методу та досліджуваного об'єкта

На всі вищезгадані питання будуть дані відповіді в теоретичній частині даної роботи. Для цього були проаналізовані наукові статті, книги, присвячені розробці та оцінці мобільного додатка на даних фреймворках. Також була вивчена документація самостійним самих інструментів для розробки та фреймворків.

У першому розділі розповідається про появу цих фреймворків від Apple, формуються основні відмінності кожного з них. Наприкінці розділу на основі проаналізованих даних наводиться таблиця з основних відмінностей самих фреймворків.

У другому розділі аналізується написаний блок коду на основі маленького додатку в кожному з фреймворків, порівнюючи довжину та складність реалізації самого блоку коду.

У третій главі представлено практичне втілення мобільного додатку "Погода" на кожному з доступних фреймворків. У рамках втілення спочатку буде створено прототип додатку, потім його перенесуть у цифровий вигляд, де буде переведений на код кожного з фреймворків, що в подальшому порівняють кожний аспект та складність втілення додатку.

Отже, робота складається з теоретичної та практичної частини, в якій розроблено прототип та сам працюючий додаток "Погода". На основі коду, який допоможе початківцям розробникам визначити для себе, який з фреймворків буде обраний для написання їх нового додатку, а також подальшого більш глибокого вивчення його.

Метою роботи є порівняння двох доступних нативних UI фреймворків від компанії Apple, що допоможе розробникам вибрати для написання свого власного додатку і подальшої його публікації в App Store.

Після цього ми, аналізуючи теоретичну частину та розробляючи власний додаток, зробить висновок про те, наскільки кожен з UI фреймворків відрізняється в підході до реалізації та з якого починати вивчення сьогодні початкуючому розробнику.

Для досягнення мети ми поставили наступні завдання:

- Порівняти існуючі UI фреймворки від компанії Apple.

- Створити таблицю, в якій порівнюється кожен з фреймворків.

- Порівняти однакові блоки коду між собою на кожному з фреймворків.

- Розробка та реалізація додатку погоди для фінального порівняння та визначення вибору фреймворку для вивчення початкуючому розробнику

1. ДОСТУПНІ UI ФРЕЙМВОРКИ ВІД КОМПАНІЇ APPLE

Спочатку розглянемо що таке нативний додаток.

Нативний додаток - у перекладі з англ. native означає "рідний". Нативним називається додаток, розроблений «рідною» для платформи мовою програмування. Для Android це Java, для Apple iOS – Objective-C та Swift.

Мову програмування Objective-C створив Бред Кокс (ориг. Brad J. Cox) у своїй компанії Stepstone у 1980 році. Objective-C - компільована об'єктно-орієнтована мова програмування, заснована на C, яка пізніше стала основною мовою розробки для пристроїв Apple, що використовується на платформі NeXT для операційної системи NeXTSTEP. У 1996 році Apple Computer придбала NeXT і почала використовувати OpenStep у MAC OS X. (Medium, 2017, 1 абзац)

У лютому 2008 року корпорація Apple випустила iPhone SDK 2.0 - набір засобів, призначений для розробки програмного забезпечення, що допомагає розробникам створювати власні нативні програми для пристроїв Apple.

На той момент це справило революцію в розробці мобільних додатків і створило нове покоління розробників – на iOS.

Що стосується Swift, у 2010 році почалася робота над створенням нової мови програмування Swift, яку презентували через чотири роки у 2014 році на Всесвітній конференції Apple. Swift продуктивніший і

функціональніший, ніж Objective-C, оскільки розроблений спеціально під iOS.

На відміну від Objective-C, Swift - статично типізована мова, що зменшує кількість помилок на етапі програмування. Тим не менш, Swift підтримує взаємодію як з C, так і з Objective-C, включаючи нові ефективні функції: замикання, дженерики та інші додатки для розробників iOS. Згідно з опитуванням компанії JetBrains (виробник програмного забезпечення), понад 75% розробників використовують Swift для створення своїх додатків. З часом Objective-C пішов на другий план, і сьогодні лише одиниці розробників пишуть на ньому свої програми. (JetBrains, 2020, 1 абзац)

Кількість користувачів Swift та Objective-C

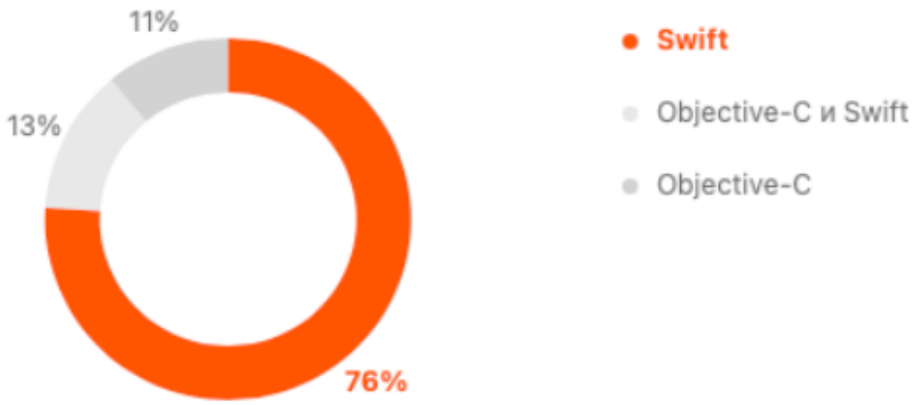


Рис. 1. Кількість користувачів Swift та Objective-C (Джерело: jetbrains.com)

У зв'язку з цим, далі йтиметься лише про фреймворк Swift.

Компанія Apple пропонує два актуальні фреймворки на сьогоднішній день: UIKit та SwiftUI.

1.1. UIKit

UIKit - це імперативний керований подіями фреймворк для створення інтерфейсів для платформи iOS. Це означає, що розробники повинні обробляти запити на зміну стану під час завантаження подання, натискання кнопки і т. д. Великим мінусом цього підходу є складність синхронізації інтерфейсу користувача з його станом. (CoderLessons, 2018, 1 абзац)

UIKit доступний для Apple, випущених не раніше 2008 року, версії 2.0 і вище.

UIKit був фреймворком, доступним для створення уявлень та розробки користувальницького інтерфейсу в iPhone. За більш ніж 13 років UIKit перетворився на потужний і універсальний фреймворк для розробки інтерфейсу користувача. У результаті спільноти UIKit був час, щоб створити безліч різноманітних і неповторних програм. А це означає, що розробники змогли створювати елегантні та високопродуктивні програми, що користуються успіхом на ринку.

Фреймворк і досі підтримується компанією Apple і постійно доопрацьовується.

1.2. SwiftUI

SwiftUI — це декларативний фреймворк для створення інтерфейсів для всіх платформ компанії Apple. Ключове слово в цьому випадку є

декларативним. Говорячи простими словами, декларативно означає, дати команду SwiftUI, про подальшу роботу та вид інтерфейсу, і фреймворк задіює меншу кількість коду для виконання роботи. SwiftUI знає найкращий спосіб візуалізації інтерфейсу користувача, на відміну від UIKit, який зазвичай використовується в поєднанні з StoryBoards. (Cocoacasts, 2020, 1 абзац)

SwiftUI заснований на програмному коді та доступний на всіх платформах, включаючи macOS, iOS, iPadOS, watchOS та tvOS/

Сам фреймворк був представлений разом з iOS 13 у 2019 році, і встиг зібрати багато фанатів навколо себе.

1.3. Парадигми програмування

Вище зазначено, що один імперативний фреймворк, а інший декларативний. Розглянемо кожен із них докладніше і порівняємо їх один з одним.

1.3.1. Імперативний стиль програмування

Під імперативним фреймворком мається на увазі імперативний стиль програмування. Це такий стиль, при якому ви описуєте, як досягти бажаного результату. Наприклад, у ньому пишеться:

• Поставте сковорідку на вогонь

• Візьми яєць (курячих)

• Нанеси удар ножем по кожному

• Вилий вміст на сковорідку

• Викинь шкаралупу

Це декларативний стиль, але при цьому з домішкою імперативного

1.3.2. Декларативний стиль програмування

Декларативний стиль, це такий стиль, в якому ви описуєте, який саме результат вам потрібен.

Тут достатньо написати:

• Приготуй яєчню

І одержувач такого повідомлення вже сам розбирається, які кроки для цього необхідно зробити.

1.3.3. Відмінність стилів програмування з прикладу

Пропоную розглянути різницю стилів на реальному прикладі.

Завдання полягає в написанні функції, яка приймає масив чисел і повертає новий масив, кожен елемент якого вдвічі більший за вхідний: [1,2,3] -> [2,4,6]].

Реалізація використовуючи імперативний стиль:

```
1 func returnArrayWithLargeValues(_ array: [Int]) -> [Int] {
2     var result: [Int] = []
3     for i in 0..<array.count {
4         result.append(array[i] * 2)
5     }
6     return result
7 }
```

Рис. 2. Реалізація задачі використовуючи імперативний стиль програмування

Очевидно, що в цій функції описується, як вирішити проблему: тут явно вказуються всі кроки.

У цьому прикладі відбувається зміна будь-якого стану. Як приклад, відбувалася зміна змінної result.

Реалізація декларативного стилю:

```
1 func returnArrayWithLargeValues(_ array: [Int]) -> [Int] {
2     return array.map { $0 * 2 }
3 }
```

Рис. 3. Реалізація задачі використовуючи декларативний стиль програмування

Можна зауважити, що код став більш читаним та компактним. У ньому використовується вбудована можливість мови, вказали, що хочемо отримати на виході, помістивши все це в один рядок.

Обидва приклади дають однакові результат, і ніхто не є ні кращим, ні гіршим за інше. Перший приклад вимагає більше коду, але код для тестування, і імперативний підхід дає повний контроль над деталями реалізації. У другому прикладі код є, ймовірно, більш читаним, однак .map, яка не дає вам контроль над тим, що відбувається за лаштунками. Розробник повинен бути впевнений, що .map буде забезпечувати необхідний результат. (Swift Apprentice, 2021, Swift Basic)

1.4. Налаштування проекту в різних UI фреймворків

Грунтуючись на знаннях про фундаментальні відмінності при написанні коду в кожному з фреймворків, я демонструю як налаштовуюти ті самі фреймворки і як відбувається робота.

1.4.1. Налаштування проекту в UIKit

При першому запуску користувача зустрічає вікно налаштування програми. Із зазначенням назви додатка, його версії, під які операційні платформи вона розробляється. Розробника-початківця може злякати такий неочевидний вступ у розробку своєї програми, адже не відразу зрозуміло, що йому робити і куди натискати.

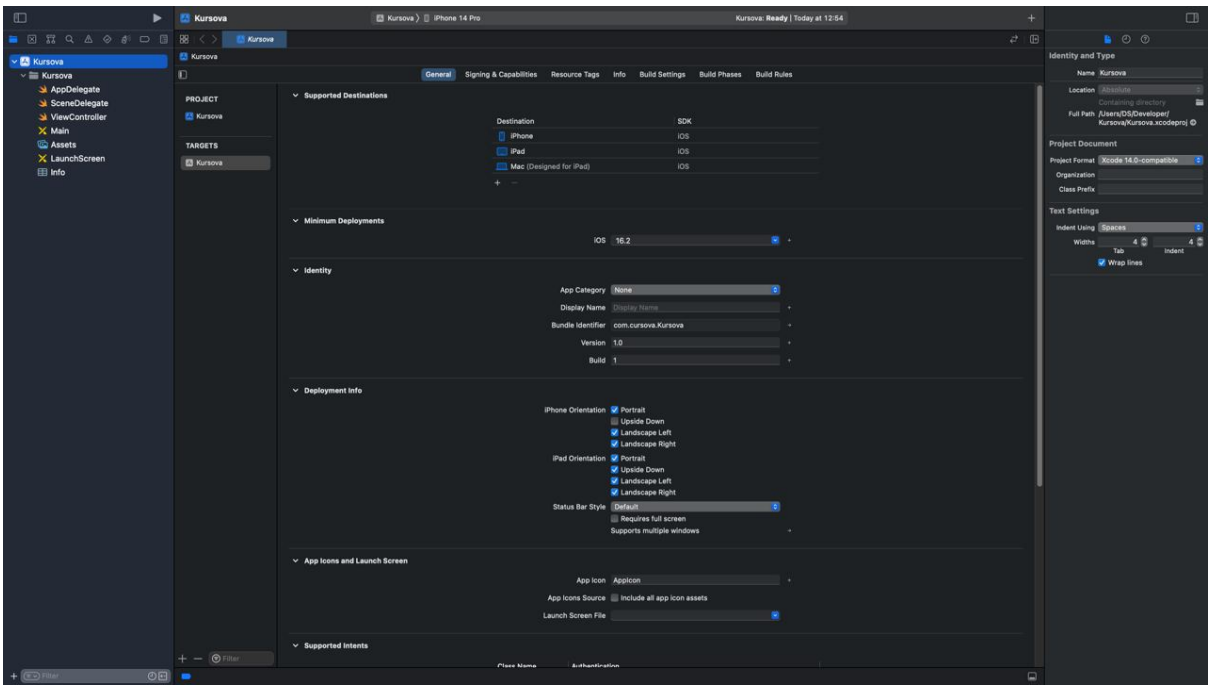


Рисунок 4. Приклад першого запуску розробки програми у UIKit

Насамперед ми повинні розібрати з яких компонентів складається програма написана на UIKit. Зліва видно список компонентів, які потрібні для розробника у його розробці.

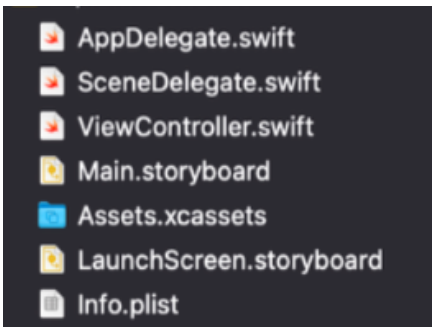


Рисунок 5. Список компонентів для розробки програми на UIKit

Перший файл містить клас з ім'ям AppDelegate, який дозволяє стежити за станами вашої програми. Він також містить головне вікно програми - тобто UIWindow. У цьому класі налаштовуються всі основні потреби вашого додатка, будь то Push Notifications або Voice Calls тощо. Можна сказати це ядро нашої програми.

Наступним йде файл із класом SceneDelegate. Цей клас у нас відповідає за стан програми. Наприклад:

• Що робити з програмою, якщо користувач її згорнув

• Що робити програми, якщо користувач надійшов дзвінок

• Що робити, якщо користувач заблокував пристрій

• І так далі…

У цьому класі описується будь-який стан і дії у разі їх виконання.

Наступні два файли, як Main.Storyboard і ViewController, є основними в розробці програми. Саме в них і вибудовується інтерфейс та логіка всієї програми. Опис їхньої роботи буде пояснено пізніше.

У папці Assets містяться всі зображення, музичні файли, іконки для застосування.

Файл LaunchScreen відповідає за вступний екран всієї програми

Файл Info відповідає за всю тонку та поглиблену інформацію програми, такої як мінімальна модель процесора для програми, з якого екрана нас зустрічатиме програма і т.д.

1.4.1.1. Розробка програми у UIKit

Як і було описано вище, все написання програми відбувається в Main.Storyboard і прив'язаний до нього ViewController. Саме в них додаються елементи на екран і задаються функції, які користувач хоче в них побачити.

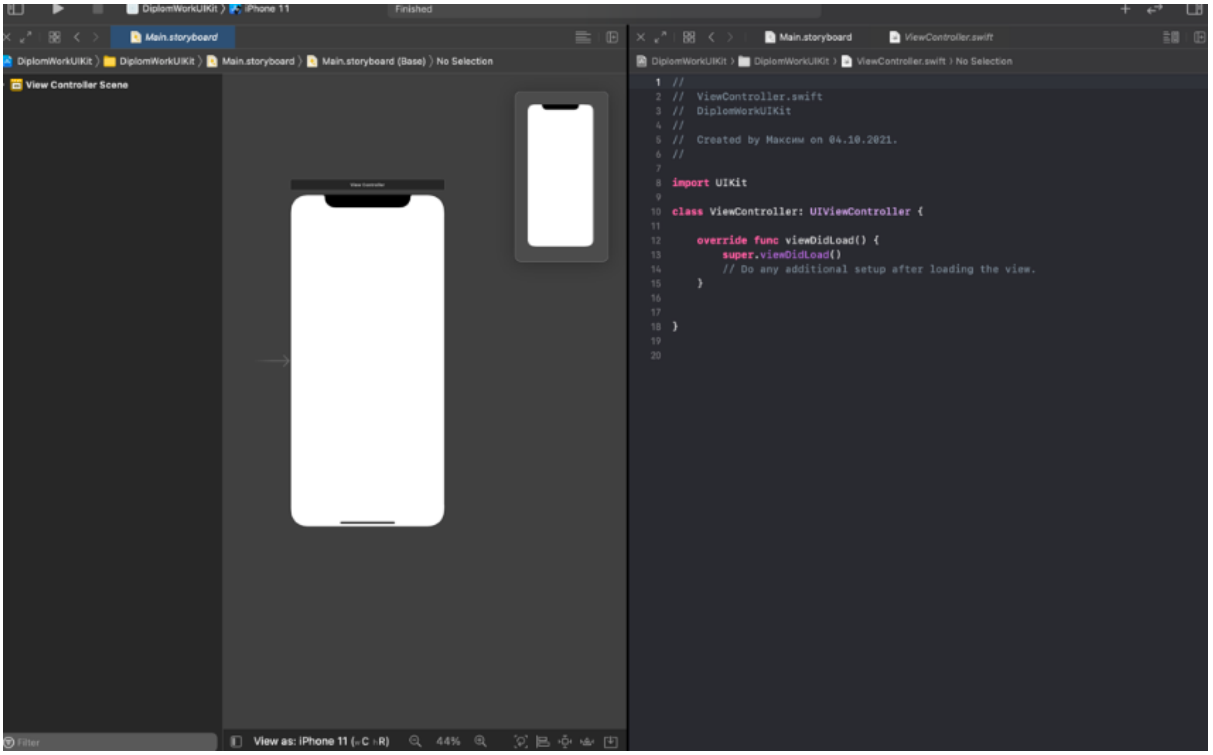


Рисунок 6. Приклад інтерфейсу розробки програми на UIKit

UIKit має імперативний стиль написання програми. А це означає, що при додаванні якої-небудь кнопки, повзунка, смужки прогресу, користувач повинен для кожного з об'єктів вказати, де він знаходиться, наскільки він далеко від інших об'єктів, розписати його дії та логіку виконання цих дій. і т.д.

1.4.2. Налаштування проекту у SwiftUI

Тепер розглянемо розробку програми під час використання SwiftUI. Користувача зустрічає одразу екран з невеликим написаним кодом і телефоном, на якому цей код зображений у дії. Такий вступ швидше більш доброзичливий до розробника-початківця, адже він відразу бачить перед собою область роботи і може відразу подивитися результат свого виконаного коду.

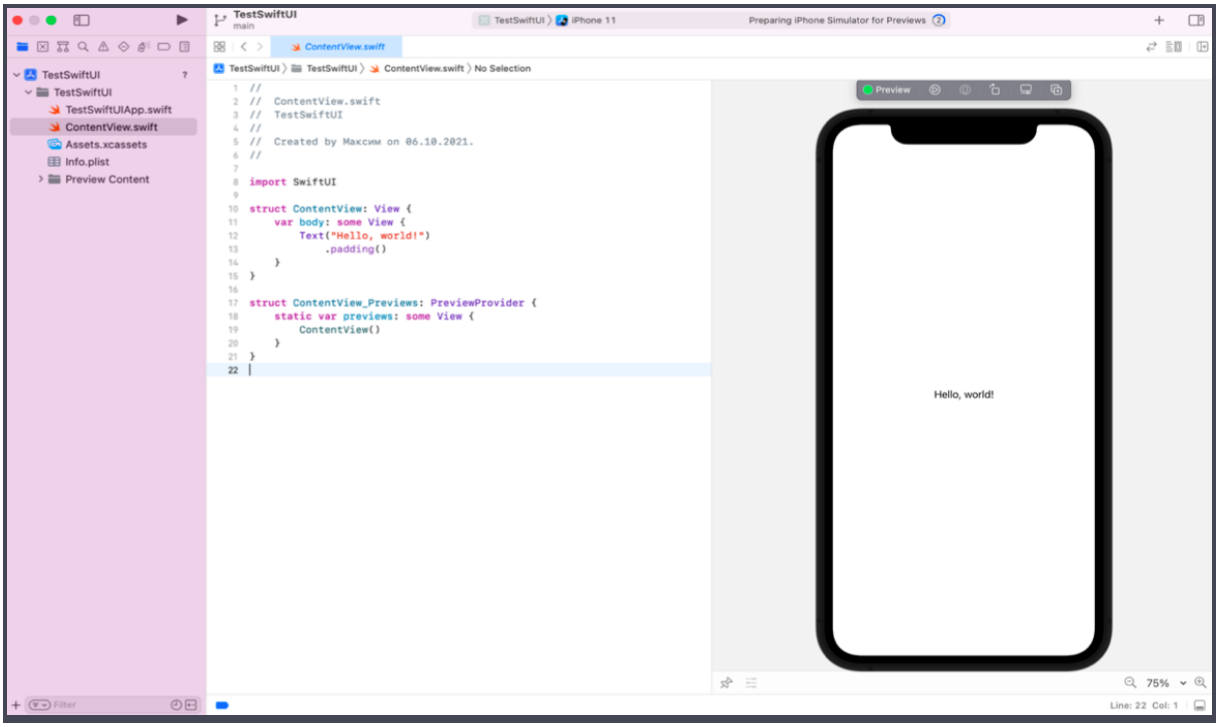


Рисунок 7. Приклад першого запуску розробки програми Swift

Але перш за все, я хочу розібрати за що якийсь файл відповідає вже в новому фреймворку від Apple.

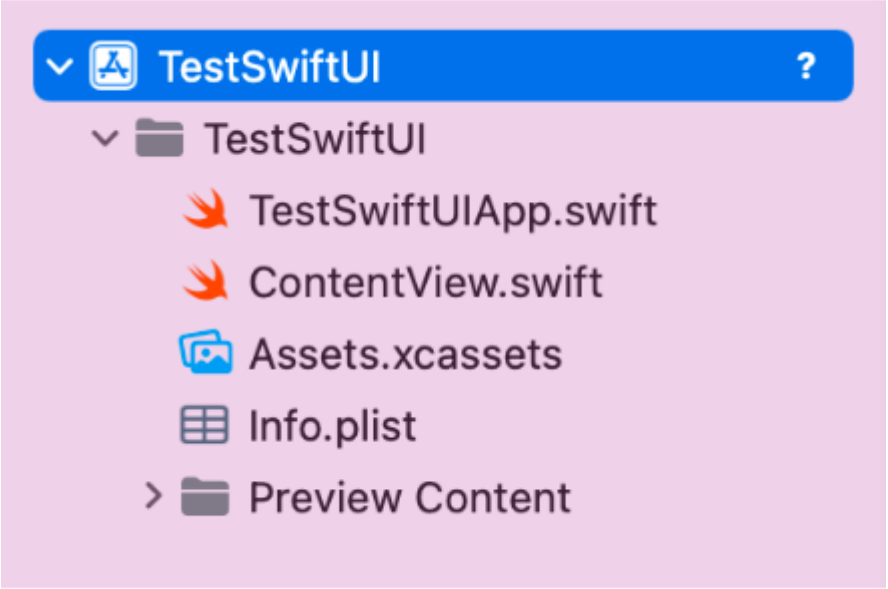


Рисунок 8. Список компонентів для розробки програми у SwiftUI

У прикладі перший файл називається - TestSwiftUIApp. Цей файл бере назву, ґрунтуючись на назві проекту. Як і AppDelegate у UIKit, цей файл є відправною точкою всієї програми. У ньому вказується з якого екрану буде включатиметься наша програма, в якій ієрархії вона буде показана і як між собою об'єднана.

Файли Assets та Info залишилися такими ж, як і в UIKit.

ContentView файл є основним для розробки програми. Саме в ньому влаштовується логіка всієї програми та її інтерфейс.

1.4.2.1. Розробка програми на SwiftUI

Як говорилося раніше, SwiftUI має декларативний стиль написання коду, тим самим користувачеві достатньо написати, що він хоче бачити на екрані і що воно має робити, а фреймворк вже за нього вибудує як той чи

інший об'єкт має виглядати і де знаходиться. Так само дуже важливою частиною є те, що в SwiftUI використовуються блоки коду, що описують, в якому положенні повинен шикуватися екран.

Усього таких 3 блоки, а саме:

• VStack(Vertical Stack) - відповідальний за вертикальне розташування об'єктів.

• HStack(Horizontal Stack) – відповідальний за горизонтальне розташування

• ZStack – відповідальний за об'єкти в Z-осі, що описує накладання об'єктів один на одного

Всі ці блоки дуже важливі у розробці програми в SwiftUI і є основоположними у розробці на даному фреймворку

Маючи весь вивчений матеріал за типами додатків, я ґрунтуючись на головних критеріях оцінки фреймворку даних Vikram Yadav(Hackernoon, 2019) складено порівняльну таблицю основних відмінностей кожного з фреймворку за перерахованими даними. Грунтуючись на них, розробник-початківець зможе вибрати для себе підходящий йому фреймворк для розробки програми.

2.РОЗРОБКА І ПОРІВНЯННЯ БЛОКІВ КОДУ В КОЖНОМУ З ФРЕЙМ-ВОРІКІВ

Вивчивши теоретичну інформацію про кожного з фреймворків, я написав і порівняв написання та реалізацію одного й того ж функціоналу у кожному з фреймворків

2.1. Написання однакових екранів програми

За основу було взято простий додаток, що містить невелику таблицю та переходом на наступний екран з докладним описом космічного корабля. Я порівняю окремі блоки реалізації в кожному з UI фреймворків, їх обсяг та складність написання.

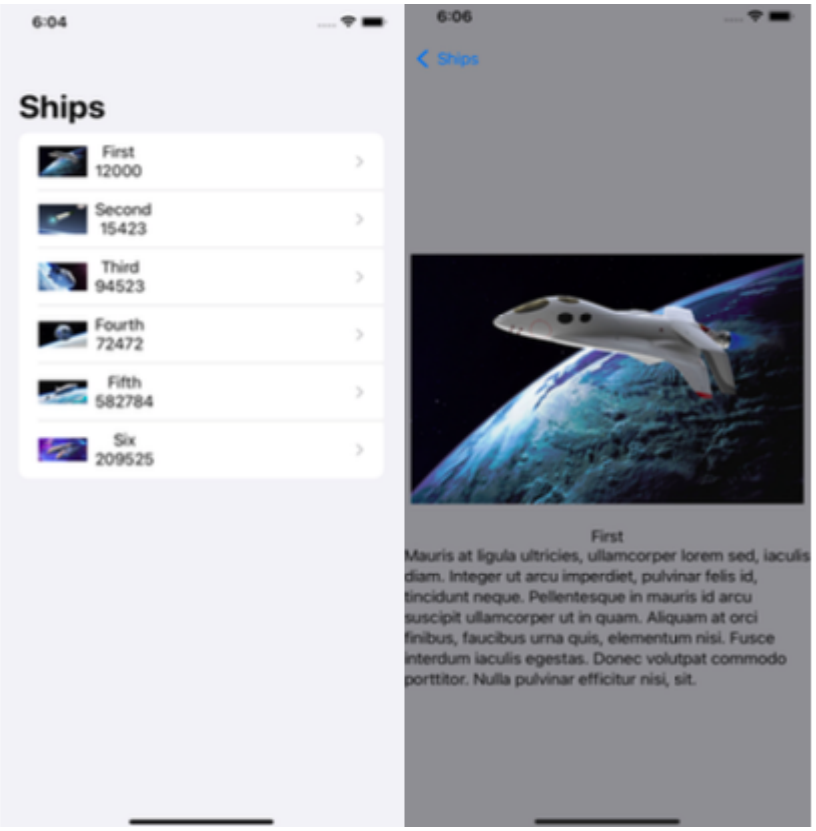


Рисунок 9. Зразок програми для демонстрації написання у двох UI фреймворках

2.1.1 Створення екрана в UIKit

Як я зауважив, на першому екрані нам представлена таблиця. Насамперед потрібно позначити масив з даними космічних кораблів і занести їх у таблицю.

Так як UIKit не розум іє, як він повинен ставитися до кожного з об'єктів, що поряд стоять до нього, користувач повинні розставити точки опори, які відносяться до всього екрану нашого смартфона. Також варто враховувати, що кожен осередок направляє на наступний екран, ми це повинні так само врахувати, і прописати до кожного з осередків її логіку виконання. Для комірки створюється окремий клас, у якому і прописується, що в ній знаходиться, а також описується її розташування один до одного й розміри.

2.1.2 Створення екрана у SwiftUI

У SwiftUI також позначається масив даних космічних кораблів, який був і в UIKit. Для реалізації користувач повинен прописати, що бажає бачити аркуш із даними масиву, а також вказати, що кожен осередок є кнопкою для переходу на наступний екран. Усередині комірки вказується Stack, в якому тільки прописується знаходження картинки та тексту всередині.

Порівняння за обсягом написаного коду

Вивчивши теоретичну частину написання екранів до кожного з фреймворків, той хто буде створював додаток напише обидва екрани і порівняє окремі частини друг з одним.

Перше - це екран із таблицею:

Так виглядає таблиця в UIKit:

```
//MARK: - Configure Table View

var tableView = UITableView()

func configureTableView() {
    view.addSubview(tableView)
    //set delegates
    setTableViewDelegates()
    //set constraint
    setTableViewConstraint()
    //set cell
    tableView.register(self, forCellReuseIdentifier: "cell")
    //set cell height
    tableView.rowHeight = 60
}

func setTableViewDelegates() {
    tableView.delegate = self
    tableView.dataSource = self
}


func setTableViewConstraint() {
    tableView.translatesAutoresizingMaskIntoConstraints = false
    tableView.topAnchor.constraint(equalTo: segmentControll.bottomAnchor, constant: 10).isActi
    tableView.leadingAnchor.constraint(equalTo: view!.leadingAnchor).isActive = true
    tableView.trailingAnchor.constraint(equalTo: view!.trailingAnchor).isActive = true
    tableView.bottomAnchor.constraint(equalTo: view!.bottomAnchor).isActive = true
}
```

Рис. 10. Реалізація таблиці в UIKit

Тут я задаю всі обов'язки таблиці, її співвідношення до інших сторін екрану, а також визначаю модель комірки та їх розмір.

Разом для створення таблиці нам потрібно 25 рядків коду

Так само, варто визначити, який буде осередок усередині таблиці, її так само варто винести в окремий клас і описати.

```
var starShipImage = UILabel()
var starShipName = UILabel()
let starShipPrice = UIStackView()

//MARK: - All Configurations

func configureStackView(){
    addSubview(starShipPrice)
    starShipPrice.axis = .vertical
    starShipPrice.distribution = .fillEqually
    starShipPrice.spacing = 0
    starShipPrice.addArrangedSubview(starShipImage)
    starShipPrice.addArrangedSubview(starShipName)
    setStackViewConstraint()
}
func setStackViewConstraint () {
    starShipPrice.translatesAutoresizingMaskIntoConstraints = false
    starShipPrice.topAnchor.constraint(equalTo: topAnchor, constant: 10).isActive = true
    starShipPrice.leadingAnchor.constraint(equalTo: safeAreaLayoutGuide.leadingAnchor, constant: 10).isActive = true
}

func configureStarShipImage() {
    starShipImage.numberOfLines = 0
    starShipImage.adjustsFontSizeToFitWidth = true
}

func configureStarShipName() {
    starShipName.numberOfLines = 0
    starShipName.adjustsFontSizeToFitWidth = true
    starShipName.font = textLabel?.font.withSize(13)
}
```

Рис. 11. Реалізація осередку в UIKit

Як ми помічаємо, щоб у осередку описати картинку, і два тексти поруч із нею, потрібно 30 рядків коду, у яких здебільшого описуємо, де будуть тексту та картини щодо друг до друга.

Тепер ми створимо таку ж таблицю, але на SwiftUI

```
List(example) { ship in
    NavigationLink {
        StarshipView(name: ship.name, imageName: ship.image, description: ship.description, cost: ship.cost)
    } label: {
        HStack {
            Image(uiImage: UIImage(named: ship.image)!)
                .resizable()
                .scaledToFit()
                .frame(width: 50, height: 50)
            VStack {
                Text(ship.name)
                Text(ship.cost)
            }
        }
    }
```

Рис. 12. Реалізація таблиці у SwiftUI

Як ми зауважимо, для написання аркуша з осередками потрібно вказати, що хочемо бачити Аркуш з масивів “example”, і розписати як виглядатиме осередок у аркуші і як поводитиметься.

Осередок у нас знаходиться в блоці label, де маємо вказати, що хочемо на горизонтальній осі вказати картинку, а біля неї вертикальний блок із двома текстами, у вигляді імені та ціни за корабель.

Разом на весь блок з листом і осередками нам знадобилося 13 рядків коду.

Тепер варто звернути увагу на другий екран, що складається з картинки, а під ним назва та опис корабля.

Також варто почати з UIKit:

Я створюю клас, який буде приймати дані корабля, з його картинкою і текстом. Об'єднавши картинку та тексту в один блок, так само варто виставити точки опори. (Outltes)

```
var starShipImage = UILabel()
var starShipDescription = UILabel()
var starShipName = UILabel()
let starShipStackView = UIStackView()

func configureStackView(){
    addSubview(starShipImage)
    starShipStackView.axis = .vertical
    starShipStackView.distribution = .fillEqually
    starShipStackView.alignment = .center
    starShipStackView.spacing = 16
    starShipStackView.addArrangedSubview(starShipImage)
    starShipStackView.addArrangedSubview(starShipName)
    starShipStackView.addSubview(starShipDescription)
    setStackViewConstraint()
}
func setStackViewConstraint () {
    starShipStackView.translatesAutoresizingMaskIntoConstraints = false
    starShipStackView.centerXAnchor.constraint(equalTo: centerXAnchor).isActive = true
    starShipStackView.centerYAnchor.constraint(equalTo: centerYAnchor).isActive = true
}

func configureStarShipImage() {
    starShipImage.adjustsFontSizeToFitWidth = true
}

func configureStarShipName() {
    starShipName.adjustsFontSizeToFitWidth = true
}
```

Рис. 13. Реалізації осередку в UIKit

Цей блок коду дуже схожий на той, що ми описали вище, єдине тепер всі об'єкти об'єднали в один блок і вирівняли контент у ньому по центру. А також виставили точки опори по центру відносно всього екрану.

Разом нам на написання нового екрана знадобилося 68 рядків коду Тепер код на SwiftUI:

Так само створюється структура, яка прийматиме модель корабля з його картинкою, назвою і описом.

```
let name: String
let imageName: String
let description: String
var body: some View {
    ZStack {
        Color.gray.edgesIgnoringSafeArea(.all)
        VStack{
            Image(uiImage: UIImage(named: imageName)!)
                .resizable()
                .scaledToFit()
                .frame(width: 400, height: 300)
            Text(name)
            Text(description)
        }

    }
}
```

Рис. 14. Реалізація осередку у SwiftUI

Так як все це знаходиться один під одним, потрібно використовувати вертикальний блок, який об'єднує всі три елементи. А також, так як у заднього фону у екрану, варто додати блок в Z-блок, на першому шарі якого буде фон

Разом на написання екрану нам знадобилося 16 рядків коду.

Маючи загальне уявлення про те, як пишеться код в одному та іншому UI фреймворку, Ми приступаємо до написання цілої програми погоди.

3. РОЗРОБКА ДОДАТКУ

Розробка програми погоди за допомогою кожного фреймворку була розділена на кілька етапів.

Перший етап – створення прототипу майбутнього прототипу у програмі Sketch.

Другий етап – реалізація програми використовую обидва фреймворки.

Третій етап – порівняння кожного додатка один з одним, його швидкість завантаження, час на написання коду, а також складності в реалізації кожного з них.

3.1 Прототип інтерфейсу програми

Як додаток для реалізації, було вибрано додаток для перегляду пороки. У цьому додатку реалізована робота з мережею, запити та отриманням JSON файлом, згодом привівши його в UI елементи.

Так як обидві програми написані за допомогою мови Swift і за допомогою однієї архітектури, вся бізнес-логіка, а саме отримання даних із сервера, обробка в потрібну структуру та передача в потрібну модель, у них загальна. А реалізація UI елементів у кожного своя так само, як і підхід до реалізації. Що я й планую продемонструвати.

Як джерело погоди, був використаний відкритий API код із сайту OpenWeath-erMap.Org (OpenWeatherMar, 2021, 1 абзац). З нього були отримані дані про погоду в конкретному регіоні на даний момент, а також погода по годинах та на 7 днів наперед.

Основне завдання програми було мати пошуковий рядок, де після введення, користувач натискає кнопку пошуку і видає погоду в потрібному йому регіоні.

Також у користувача є можливість подивитися погоду на сьогоднішній і на завтрашній день. Є можливість переглянути погоду на 7 днів наперед.

На основі цієї логіки нами складено прототип додатка (додаток 2).

Дизайн програми було намальовано у програмі Sketch для MacOS. Також нами було використано рекомендації щодо налаштування інтерфейсу Human interface guidelines від компанії Apple. Саме в ньому описані основні рекомендації щодо розташування та налаштування зрозумілі користувачеві.

Насамперед ми спираємось на зручність користувача, щоб зайшовши у додаток, міг відразу побачити корисну йому інформацію.

Так само, як було сказано вище, нами було оброблено JSON файл. Це такий текстовий формат обміну даними, що базується на JavaScript. Як і багато інших текстових форматів, JSON легко читається людьми. Він допомагає розробникам швидко передавати потрібну інформацію та її структурувати

3.2 Реалізація програми на UIKit

Насамперед було перенесено весь інтерфейс з дизайну на робочу область програми. UIKit дозволяє легко вибрати зі списку доступних об'єктів потрібні кнопки, картинки, тексту та перенести їх на робочу область.

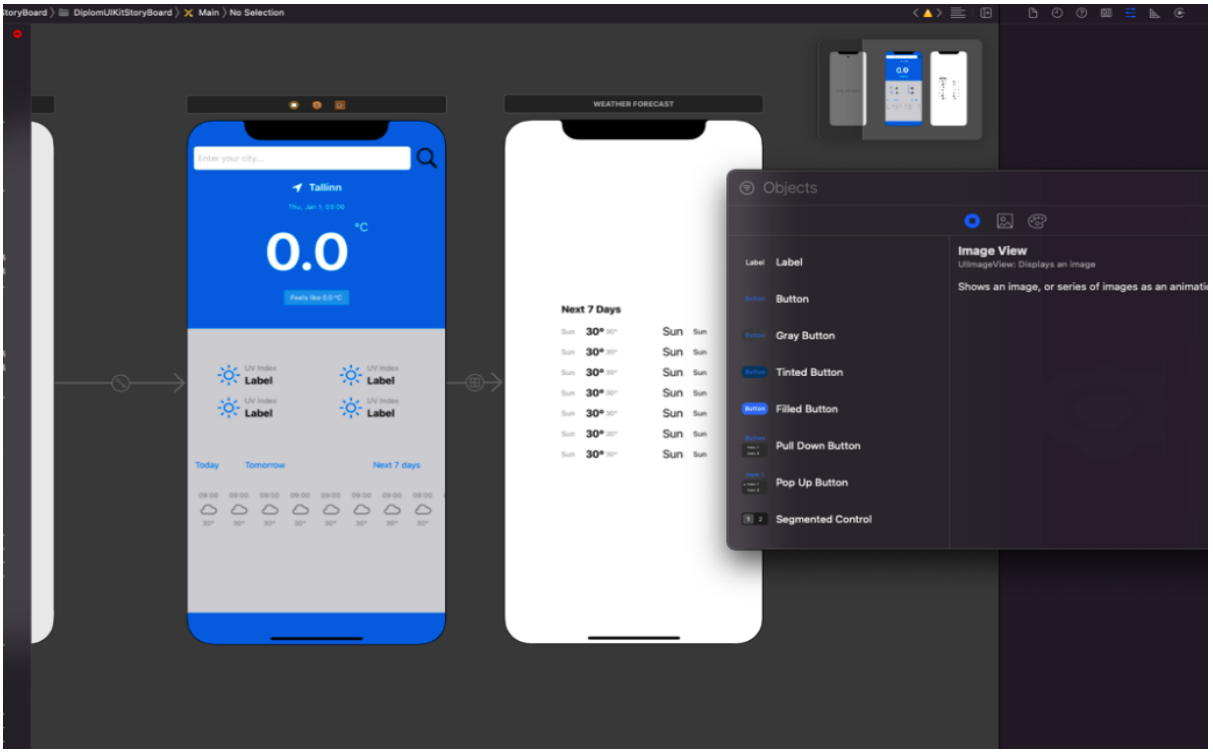


Рисунок 15. Приклад додавання компонентів на екран у UIKit

Потім після перенесення кожного елемента, його варто перенести в робочу область коду, де буде задана логіка і опис кожного елемента інтерфейсу.

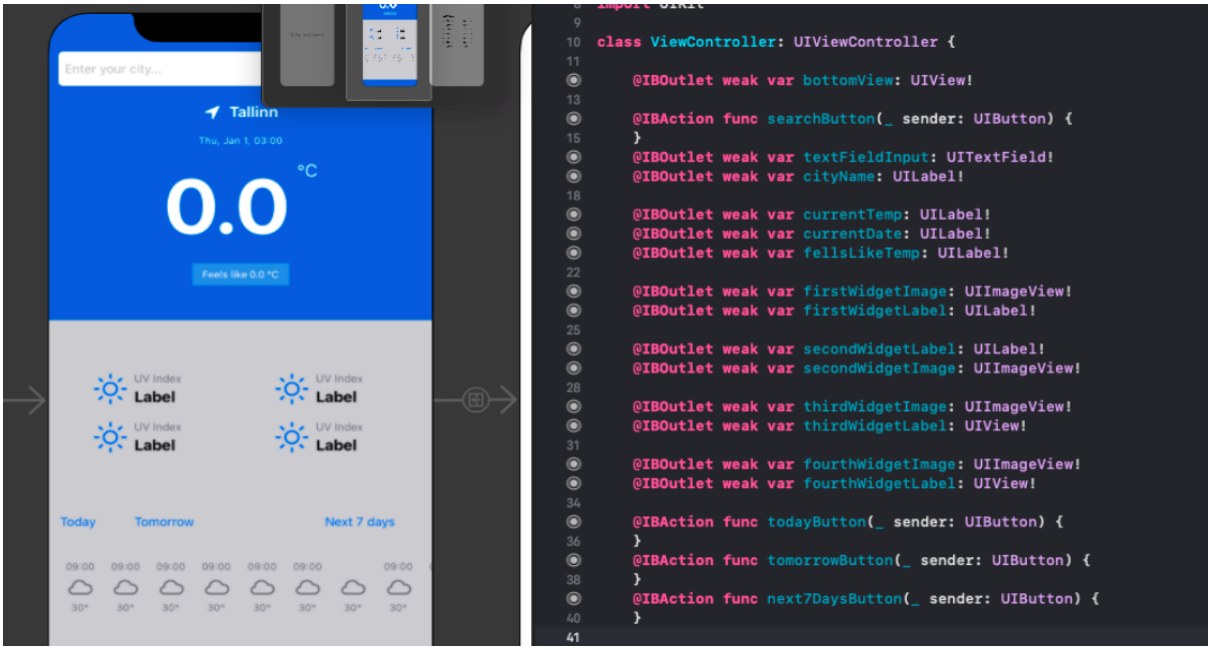


Рис. 16. Перенесення об'єктів задіяних на екрані

Так як UIKit не знає, як розташовувати наші елементи, розставляються всі точки опори (Outlets) для кожного з них

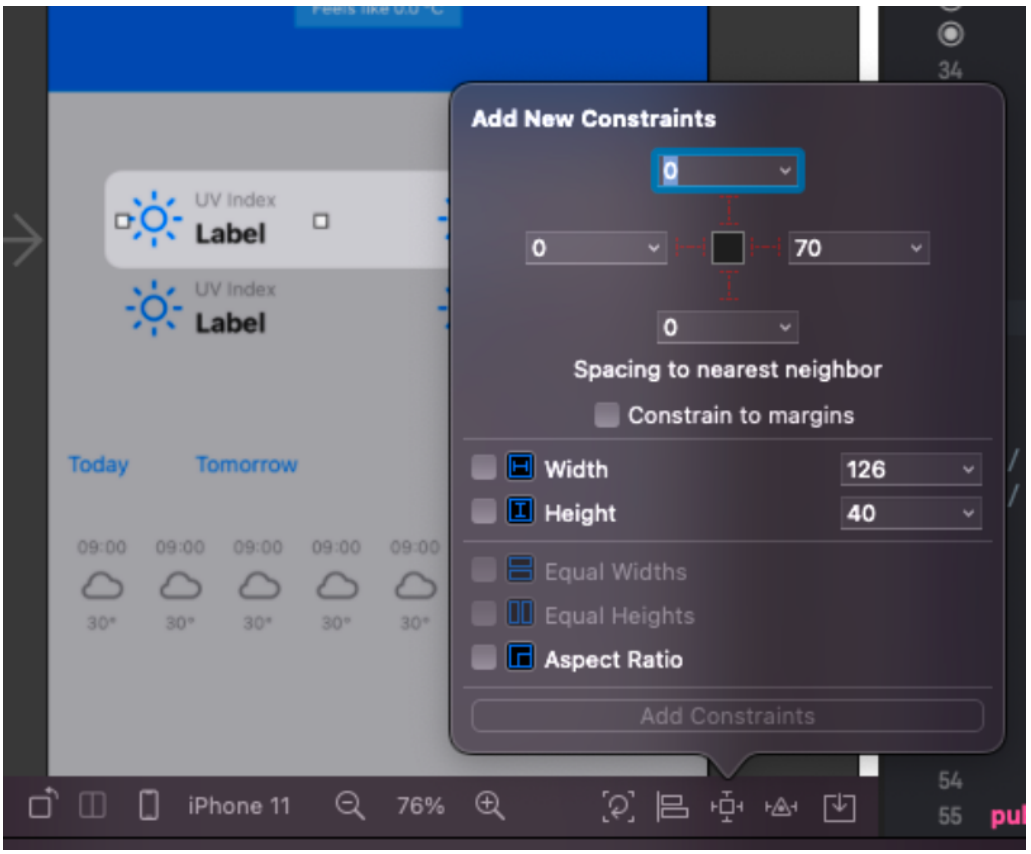


Рис. 17. Розставлення точок опори

Потім ми підключаємо вже заготовлену модель даних, в якій описана логіка роботи кожного елемента.

Насамперед до кожного елемента окремо заносяться з моделі його параметри.

```
func setWeather() {
    cityName.text = cityVM.city
    currentDate.text = cityVM.temperature
    fellsLikeTemp.text = cityVM.feelsLike
    firstWidgetImage.image = UIImage(systemName: "sun.max")
    firstWidgetLabel.text = cityVM.uVIIndex
    secondWidgetLabel.text = cityVM.sunrise
    secondWidgetImage.image = UIImage(systemName: "sunrise")
    thirdWidgetLabel.text = "\(cityVM.windDeg) \(cityVM.windSpeed)m/s"
    thirdWidgetImage.image = UIImage(systemName: "wind")
    fourthWidgetLabel.text = cityVM.sunset
    fourthWidgetImage.image = UIImage(systemName: "sunset")
}
```

Рис. 18. Додавання опису до кожного елемента

Потім, оскільки є список погоди по годинах, будується колекція, що складається з осередків, де знаходиться погода за конкретну годину. Так як UIKit має імперативний стиль написання, для комірки так само потрібно створити окремий клас, що описує її поведінку.

```
class HourlyWeatherCollectionCell: UICollectionViewCell {

    @IBOutlet weak var temp: UILabel!
    @IBOutlet weak var weatherIcon: UIImageView!
    @IBOutlet weak var time: UILabel!

    var cityVM: CityViewViewModel?
    var hourlyWeather: Weather? {
        didSet {
            temp.text = cityVM!.getTempFor(temp: hourlyWeather!.temp)
            weatherIcon.image = cityVM!.getWeatherIcon(by: hourlyWeather!.weather)
            time.text = cityVM!.getTimeFor(timestamp: hourlyWeather!.dt!)
        }
    }

}
```

Рис. 19. Налаштування логіки комірки у таблиці

І після створення кожного класу, ми описуємо вже всередині екрану модель комірки та її поведінку.

```
func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
    return hourlyWeather.count
}

func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
    if let itemCell = collectionView.dequeueReusableCell(withReuseIdentifier: "hourlyCell", for: indexPath) as?
        HourlyWeatherCollectionCell {

        itemCell.cityVM = cityVM
        itemCell.hourlyWeather = hourlyWeather[indexPath.row]

        return itemCell
    }

    return UICollectionViewCell()
}
```

Рис. 20. Реалізація масивів комірок

Після того як весь екран налаштований, залишається тільки написати логіку поведінки кнопок, а саме пошук погоди по місту та перемикання погоди по днях.

```
@IBAction func searchButton(_ sender: UIButton) {
    textFieldInput.endEditing(true)
}

@IBAction func todayButton(_ sender: UIButton) {
    isToday = true
}
@IBAction func tomorrowButton(_ sender: UIButton) {
    isToday = false
}

@objc func textFieldDidEndEditing(_ textField: UITextField) {
    cityVM.city = textField.text ?? ""

    textField.text = ""
}
```

Рисунок 21. Налаштування дії під час натискання кнопок

Для реалізації другого екрана ми робимо такі ж операції, як і на першому екрані, а саме перенесення необхідних об'єктів на екран, розставляння точки опори один до одного і перенесення в код.

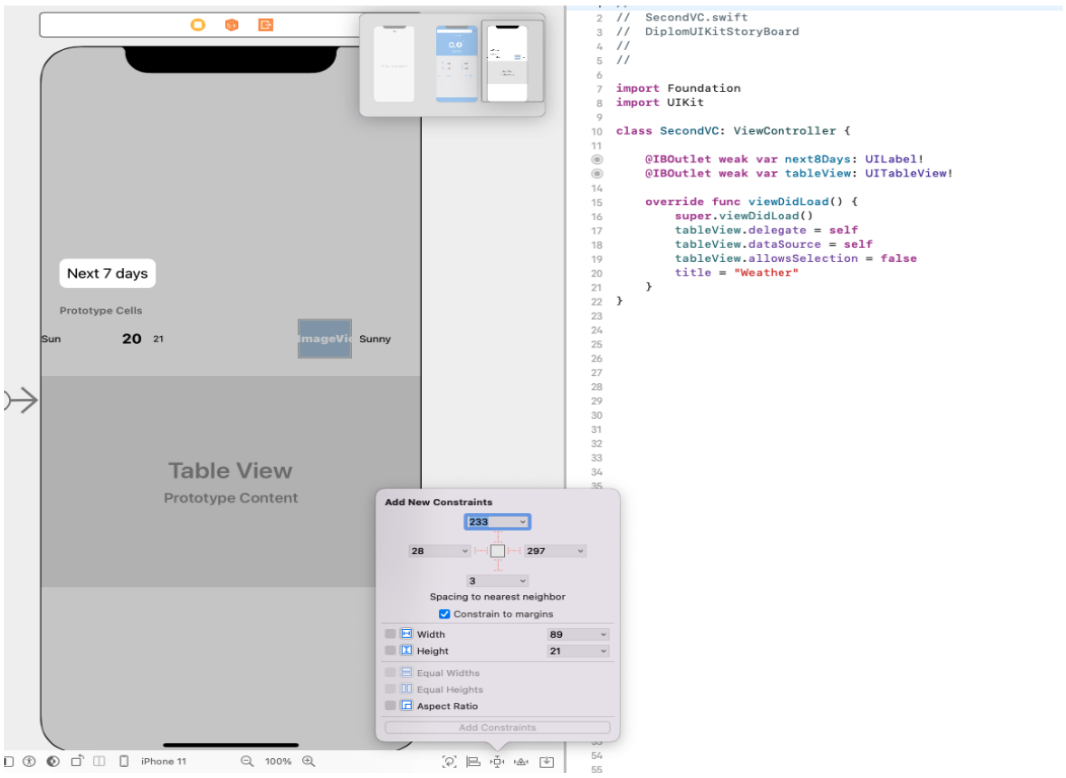


Рисунок 22. Налаштування другого екрану

Для реалізації погоди по дням ми будемо використовувати таблицю з 7 комірок. ЇЇ реалізація схожа на ту же, яку ми описували вище для погоди по часу. Так же потрібно створити окремий клас для ячейки, де описано кожен блок у кожній комірці, а саме день неділі, середня погода, погода, яка буде відчуватися, картинка зі знаком погоди та опис цієї погоди.

Мною був розділений головний екран на 3 блоки:

• Шапка: пошук, назва міста, великий блок погоди і як відчувається температура

• 4 віджети, в яких дається коротке уявлення про погоду в цьому регіоні

• Перегортаючи блок з погодою на сьогодні, а також кнопками на перегляд на завтра та на найближчі 7 днів

Завдяки цьому поділу нам вдалося окремо розписати кожен блок і в кінці їх з'єднати спільним екраном.

```
class SevenDaysCell: UITableViewCell {

    @IBOutlet weak var dayOfTheWeek: UILabel!
    @IBOutlet weak var temp: UILabel!
    @IBOutlet weak var disc: UILabel!
    @IBOutlet weak var weatherIcon: UIImageView!
    @IBOutlet weak var feelsLike: UILabel!

    var cityVM: CityViewViewModel?

    var dailyWeather: DailyWeather? {
        didSet {
            dayOfTheWeek.text = cityVM!.getDayOfTheWeek(day: dailyWeather!)
            temp.text = "\(cityVM!.getDailyTemperature(for: dailyWeather!))°"
            feelsLike.text = " \(cityVM!.getFeelsLikeTemperature(for: dailyWeather!))°"
            weatherIcon.image = cityVM!.getWeatherIcon(by: dailyWeather!.weather)
            disc.text = dailyWeather!.weather[0].description
        }
    }
```

Рисунок 23. Налаштування комірки для таблиці другого екрану

Потім після опису самої комірки, в самому класі з екраном, описується, як виглядає комірка, яку модель вона приймає і скільки хоче комірок побачити

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    cityVM.weather.daily.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    if let cell = tableView.dequeueReusableCell(withIdentifier: "tableCell", for:
        indexPath) as? SevenDaysCell {
        cell.cityVM = cityVM
        cell.dailyWeather = cityVM.weather.daily[indexPath.row]
        return cell
    }

    return UITableViewCell()
}
```

Рисунок 24. Реалізація масиву комірок в таблиці

Об'єднавши все це, ми отримуємо готовий додаток, можливе описати погоду на найближчі 48 годин, а також на наступні 7 днів у будь-якому місті.

3.3. Реалізація на SwiftUI

У SwiftUI у відмінності від UIKit необхідно вручну прописати кожен елемент, який хочеться відобразити.

Нами було розділений головний екран на 2 блоки:

• **Шапка:** пошук, назва міста, великий блок погоди і як відчувається температура

• **Перегортаючий блок з погодою** на сьогодні, а також кнопками на перегляд на завтра та на найближчі 7 днів.

Завдяки цьому поділу нам вдалося окремо розписати кожен блок і в кінці їх з'єднати спільним екраном.

Шапка:

Так як SwiftUI використовує декоративний стиль програмування, він вимагає лише вказати, що хочемо бачити і в якому вигляді на екрані.

Оскільки всі об'єкти розташовані один під одним, ми використовуємо Vstack — вертикальний блок, у ньому кожен об'єкт йтиме один під одним.

Від нас тільки залишається вказувати, які об'єкти це будуть.

```
var body: some View {
    VStack(spacing: 0) {
        HStack {
            TextField("Enter your city...", text: $cityName)
                .padding(.vertical, 6)
                .padding(.horizontal, 6)
                .background(RoundedRectangle(cornerRadius: 10).fill(Color.white))
                .padding()
            Button(
                action: {
                    cityVM.city = cityName
                },
                label: {
                    Image(systemName: "magnifyingglass")
                }
            )
                .foregroundColor(.white)
                .font(.system(size: 34))
                .padding(.trailing, 16)
        }
        .padding(.bottom, 20)
        VStack(spacing: 0) {
            HStack {
                Image(systemName: "location.fill")
                Text(cityVM.city)
            }
            .font(.body.bold())

            Text(cityVM.date)
                .font(.caption)
        }
        .foregroundColor(.white)
        .padding(.bottom, 16)

        HStack(alignment: .top, spacing: 0) {
            Text(cityVM.temperature)
                .font(Font.system(size: 90).bold())
            Text("°C")
                .font(Font.system(size: 20))
        }
        .padding(.leading, 20)
        .foregroundColor(.white)
        .padding(.bottom, 12)

        ZStack {
            Rectangle()
                .fill(Color.white)
                .opacity(0.3)
                .cornerRadius(6)
                .frame(width: 105, height: 26)
            HStack(spacing: 4) {
                Text("Feels like")
                HStack(spacing: 2) {
                    Text(cityVM.feelsLike)
                    Text("°C")
```

Рис. 25. Структура шапки головного екрана

Як видно, кожен блок відокремлений своїм Stack, який і вказує як розташовувати об'єкти по відношенню один до одного, а відстань між об'єктами визначається Padding'ами по горизонталі, або вертикалі

3.4 Підсумкове порівняння технологій SwiftUI та UIKit

В результаті ми маємо два однакові програми, з однаковою бізнес-логікою та моделями, але при цьому з різним підходом до їх реалізації UI.

У UIKit було достатньо перенести потрібні компоненти на робочу область і прописати їх логіку, тоді як у SwiftUI вручну прописується які елементи хочемо бачити і приблизне їх співвідношення один до одного

Разом для реалізації на UIKit потрібно два класи з описом кожного з екранів, що в сумі вийшло близько 170 рядків коду. І два класи під кожну з осередків у двох таблицях, що у сумі вийшло близько 70 рядків коду. Разом виходить 240 рядків коду для всієї програми.

У SwiftUI ми постаралися розділити багато блоків за окремими структурами, щоб надалі можна було їх перевикористовувати
Разом було створено чотири додаткові структури, дві під кожну з осередків у таблицях. На всі чотири структури знадобилося 145 символів. Після їхнього об'єднання у двох структурах під кожен екран ми задіяли сумарно 165 рядків коду. Разом для всієї програми на SwiftUI використовувалося 310 символів.

Я зауважую, що хоч і SwiftUI досить молодий фреймворк, на відміну від UIKit, має великий потенціал у розвитку, адже щороку компанія Apple випускає оновлені версії свого фреймворку, що в найближчому майбутньому поставить його врівень з UIKit. Але так чи інакше UIKit має фундаментальне значення для створення програм на платформах Apple зараз і в майбутньому.

SwiftUI – хороший вибір за умови, що найменша версія проекту для iOS встановлена на iOS 13. Однак варто мати на увазі, що при створенні програми ви можете зіткнутися з проблемами та
обмеженнями. Знання UIKit допоможе вам розібратися у цих проблемах.

Отже загальна порівняльна таблиця: UIKit та SwiftUI - це дві технології, які використовуються для розробки iOS-додатків:

1. Синтаксис: UIKit використовує імперативний підхід до розробки, де програміст вказує, які дії повинні виконуватися крок за кроком. У той же час SwiftUI базується на декларативному підході, де програміст описує, як повинен виглядати і взаємодіяти інтерфейс користувача, а фреймворк самостійно керує відображенням.
2. Код: UIKit написаний мовою Objective-C або Swift, тоді як SwiftUI побудований на основі Swift. SwiftUI надає більш компактний та зрозумілий код завдяки використанню декларативного підходу.
3. Інтерфейс: UIKit надає більшу гнучкість при налаштуванні інтерфейсу, оскільки дозволяє працювати зі стандартними контролами і створювати власні вигляди. SwiftUI надає велику кількість вбудованих елементів і контролів, які можна складати один на одного для швидкої розробки інтерфейсу.
4. Розподіленість: SwiftUI працює на платформах iOS 13 та новіших версіях, тоді як UIKit підтримує всі версії iOS. Це означає, що розробка за допомогою SwiftUI може обмежуватися новішими пристроями, але дозволяє використовувати більш сучасні можливості.
5. Процес розробки: SwiftUI дозволяє використовувати "Live Preview", що дозволяє бачити зміни в реальному часі без запуску додатка. Це значно прискорює процес розробки та тестування. UIKit не має вбудованої функції Live Preview, тому зміни потрібно перевіряти за допомогою запуску.

Порівняння UI фреймворків компанії Apple

	UIKit	SwiftUI
Мова розробки	Objective-C, Swift	Swift
Мінімальна версія підтримуваної ОС	iOS 2.0+, MacOs 10.2+, WatchOS 1.0	iOS 13+, MacOs 12.0+, WatchOS 1.0+
Доступний функціонал	Повний	Не повний (є можливість підключати UIKit)
Парадигма програмування	ООП	Функціональне програмування
Кількість відкритих бібліотек	більше як 5000	більше 25000
Рік анонсування	2008	2019
Кількість матеріалу для вивчення	більше 1000	більше 500

Стиль програмування	**Імперативний**	**Декоративний**
Рекативне програмування	**Немає**	**Combine**

ВИСНОВОК

І до сьогодні компанія Apple продовжує розвивати свій новий UI фреймворк. Співтовариство розробників постійно поповнюється, додає нові бібліотеки, варіанти вирішення різних завдань, а також розвиває вже наявні рішення від компанії Apple.

В результаті цього дослідження ми відповіли на всі вище поставлені питання, а також розробили два мобільні програми на кожному з фреймворків, показавши основні відмінності при написанні та реалізації додатків на кожному з них.

У ході теоретичної частини ми розглянули кожен фреймворк окремо, дізнавшись про нюанси розробки в них. На основі цієї теоретичної частини ми написали два невеликі додатки, де пліч-о-пліч порівняли реалізацію кожного елемента, його обсяг і труднощі в ході їх побудови.

У практичній частині роботи, на основі теоретичної частини, було реалізовано два додатки про перегляд погоди. Спочатку було розроблено макет програми на основі Human Interface guidlanes від Apple, а потім у кожному з фреймворку переведено в код.

В результаті розробки додатків ми змогли глибше вивчити доступні UI фрейм-ворки від компанії Apple, розібратися в основних їх відмінностях у побудові додатків, а також продемонструвати ці відмінності розробникам-початківцям.

ВИКОРИСТАНІ ДЖЕРЕЛА:

1.Neil Smyth (2019). *SwiftUI Essentials - iOS Edition: Learn to Develop iOS Apps using SwiftUI, Swift 5 and Xcode 11*

2.Matt Neuberg (2019). *iOS 13 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics*

3.Jon Hoffman (2019). *Mastering Swift 5: Deep dive into the latest edition of the Swift programming language*, 5th Edition

4.Matt Neuburg (2019). *Programming iOS 13*. Strategy analiticts (2021) *Smartphones*,
https://www.strategyanalytics.com/strategy-analytics/blogs/devices/smartphones
Web-creator(2019)*Фреймворки*.
https://web-creator.en/articles/about_frameworks

5.IDC(2021)*Smartphone Market Share.*
https://www.idc.com/promo/smartphone-market- share

6.Code-Academy (2019) *IDE* -
https://www.codecademy.com/articles/what-is-an-ide Wezom (2021) *Нативний додаток*. - https://wezom.com.ua/blog/chem-ot-

lichajutsja-nativnoe-i-gibridnoe-mobilnye-prilozhenija

7.Apple (2021) *Human Interface Guidelines* -
https://developer.apple.com/design/human- interface-guidelines/

8.Fahim Farook (2021) *UIKit Apprentice*
9.Caroline Begbie & Audrey Tam (2020) *SwiftUI Apprentice*
10.Jay Strawn & Joshua Greene (2019) *Design Patterns by Tutorials*

11.Hackernoon (2019) *React Native vs.Flutter vs. Ionic Which is Challenging Platform for Your App*

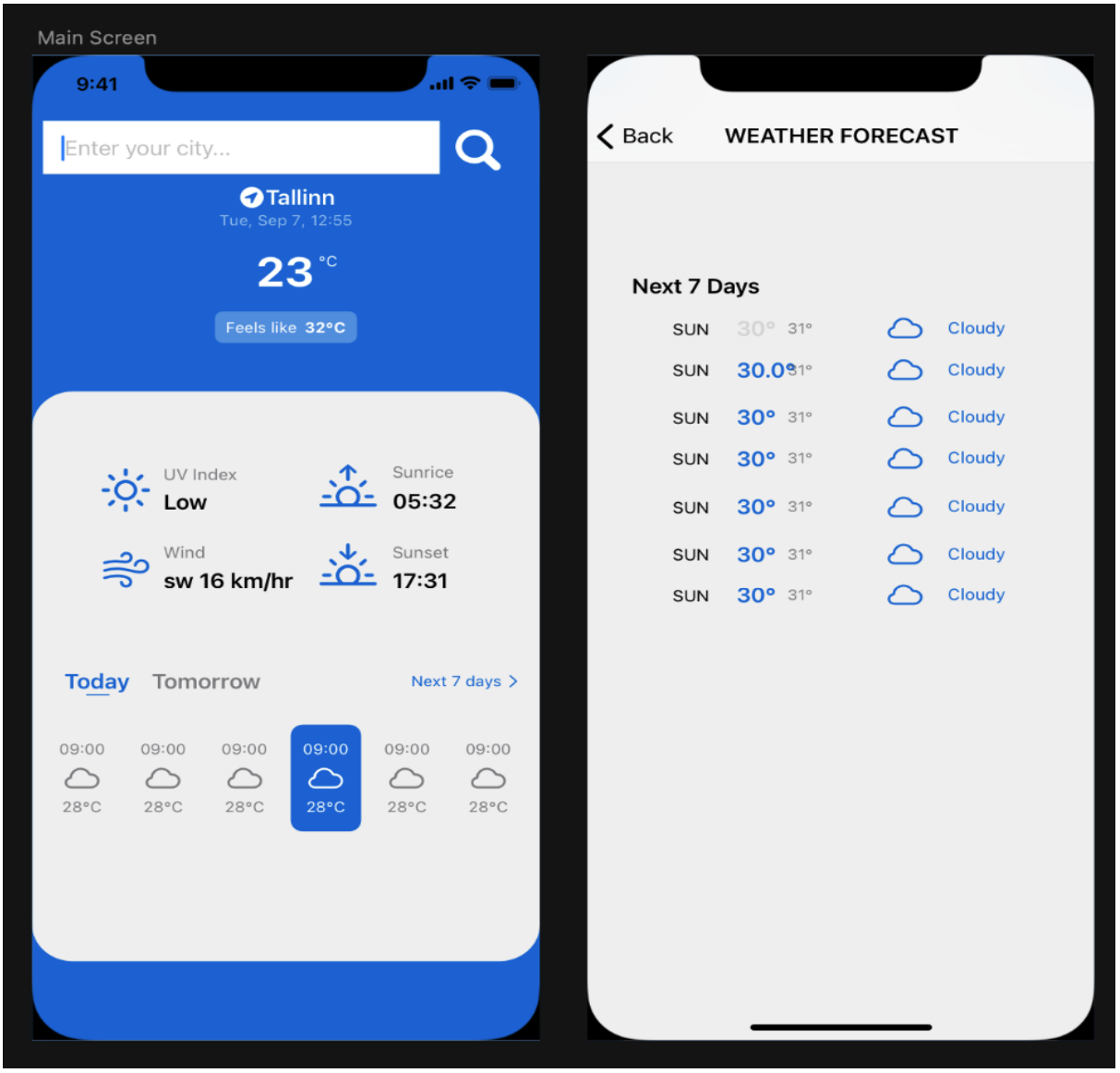
12.*Jon Hoffman (2020) Mastering Swift 5.3*
JetBrains (2019) *Swift u Objective-C*

13.Medium (2017) *A Short History of Objective-C*
https://medium.com/chmcore/a-short-his- tory-of-objective-c-aff9d2bde8dd

14.CoderLessons(2018) *UIKit*
https://coderlessons.com/articles/mobilnaia-razrabotka-arti-
cles/ios-s-nulia-s-swift-pervye-shagi-s-uikit

15.Cocoacasts(2020) *What Is SwiftUI*
https://cocoacasts.com/swiftui-fundamentals-what-is- swiftui

Прототип додатку з погодою для порівняння технологій SwiftUI та UIKit:



Додаток 1

Код програми

```
//
//  ContentView.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import SwiftUI
// MARK: - ContentView
struct ContentView: View {
    @StateObject var locationManager = LocationManager()
    var weatherManager = WeatherManager()
    @State var weather: ResponseBody?

    var body: some View {
        VStack{
            if let location = locationManager.location{
                if let weather = weather{
                    WeatherView(weather: weather)
                } else{
                    LoadingView()
                        .task {
                            do{
                                weather = try await weatherManager.getCurrentWeather(latitude:
location.latitude, longitude: location.longitude)

                            }catch{
                                print("error getting weather: \(error)")
                            }
                        }

                }
            }
            else{
                if locationManager.isLoading{
                    LoadingView()
                }else{
                    WelcomeView()
```

```
                .environmentObject(locationManager)
            }
        }
    }
    .background(Color(hue: 0.679, saturation: 0.956, brightness: 0.441))
    .preferredColorScheme(.dark)
  }
}
struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    ContentView()
  }
}


//
//  WelcomeView.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import SwiftUI
import CoreLocationUI
// MARK: - WelcomeView
struct WelcomeView: View {
  @EnvironmentObject var locationManager:
  LocationManager


  var body: some View {
    VStack{
      VStack(spacing: 20){
        Text("Weather App")
          .bold().font(.title)

        Text("Share Location")
          .padding()
      }
      .multilineTextAlignment(.center)
      .padding()

      LocationButton(.shareCurrentLocation) {
        locationManager.requestLocation()
      }
      .cornerRadius(30)
      .symbolVariant(.fill)
      .foregroundColor(.black)
```

```
        }
        .frame(maxWidth: .infinity, maxHeight: .infinity)
    }
}
struct WelcomeView_Previews: PreviewProvider {
    static var previews: some View {
        WelcomeView()
    }
}


//
//  LoadingView.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import SwiftUI
// MARK: - LoadingView
struct LoadingView: View {
    var body: some View {
        ProgressView()
            .progressViewStyle(CircularProgressViewStyle(tint: .white))
            .frame(maxWidth: .infinity, maxHeight: .infinity)
    }
}
struct LoadingView_Previews: PreviewProvider {
    static var previews: some View {
        LoadingView()
    }
}



//
//  WeatherView.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import SwiftUI
// MARK: - WeatherView
struct WeatherView: View {
    var weather: ResponseBody


    var body: some View {
        ZStack(alignment: .leading){
```

```
        VStack{
            VStack(alignment: .leading, spacing: 5){
                Text(weather.name)
                    .bold() .font(.title)
                Text("Today,\(Date().formatted(.dateTime.month().day().hour().minute()))")
                    .fontWeight(.light)
            }

            .frame(maxWidth: .infinity, alignment: .leading)
            Spacer()

            VStack{
                HStack{
                    VStack(spacing: 20){
                        Image(systemName: "cloud")
                            .font(.system(size: 40))
                        Text(weather.weather[0].main)
                    }
                    .frame(width: 150, alignment: .leading)

                    Spacer()


                    Text(weather.main.feelsLike.roundDouble() + "°")
                        .font(.system(size: 100))
                        .fontWeight(.bold)
                        .padding()
                }
                Spacer()
                    .frame(height: 80)

                AsyncImage(url: URL(string:
"https://cdn.pixabay.com/photo/2020/01/24/21/33/city-4791269_960_720.png")){
                    image in image
                        .resizable()
                        .aspectRatio(contentMode: .fit)
                        .frame(width: 350)
                }placeholder: {
                    ProgressView()
                }

                Spacer()
            }
            .frame(maxWidth: .infinity)
        }

        .padding()
        .frame(maxWidth: .infinity, alignment: .leading)
```

```
        VStack{
            Spacer()

            VStack(alignment: .leading, spacing: 20){
                Text("Today's Weather")
                    .bold()
                    .padding(.bottom)

                HStack{
                    WeatherRow(logo: "thermometer.low", name: "Min Temp", value:
(weather.main.tempMin.roundDouble() + "°"))

                    Spacer()

                    WeatherRow(logo: "thermometer.high", name: "Max Temp", value:
(weather.main.tempMax.roundDouble() + "°"))

                }
                HStack{
                    WeatherRow(logo: "wind", name: "Wind Speed", value:
(weather.wind.speed.roundDouble() + "m/s"))

                    Spacer()

                    WeatherRow(logo: "humidity", name: "Humidity", value:
(weather.main.humidity.roundDouble() + "%"))


                }
                HStack{
                    WeatherRow(logo: "aqi.medium", name: "Pressure", value:
(weather.main.pressure.roundDouble() + "hpa"))

                    Spacer()
                }

            }
            .frame(maxWidth: .infinity, alignment: .leading)
            .padding()
            .padding(.bottom, 20)
            .foregroundColor(Color(hue: 0.679, saturation: 0.956, brightness: 0.441))
            .background(.white)
            .cornerRadius(20, corners: [.topLeft, .topRight])

        }
    }
```

```
        .edgesIgnoringSafeArea(.bottom)
        .background(Color(hue: 0.679, saturation: 0.956, brightness: 0.441))
        .preferredColorScheme(.dark)
    }
}
struct WeatherView_Previews: PreviewProvider {
    static var previews: some View {
        WeatherView(weather: previewWeather)
    }
}

//
//  LocationManager.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import Foundation
import CoreLocation
class LocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {
    let manager = CLLocationManager()

    @Published var location: CLLocationCoordinate2D?
    @Published var isLoading = false

    override init() {
        super.init()
        manager.delegate = self
    }
    func requestLocation(){
        isLoading = true
        manager.requestLocation()
    }
    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations:
[CLLocation]) {
        location = locations.first?.coordinate
        isLoading = false
    }
    func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
        print("Error getting your Location", error)
        isLoading = false
    }
}


//
//  WeatherManager.swift
//  WeatherApp
```

```
//
//  Created by Markiyan Vytrykush
//
import Foundation
import CoreLocation
class WeatherManager{
    func getCurrentWeather(latitude: CLLocationDegrees, longitude: CLLocationDegrees)
async throws -> ResponseBody{
        guard let url = URL(string:
"https://api.openweathermap.org/data/2.5/weather?lat=\(latitude)&lon=\(longitude)&appid=\("
YourAPIKEY")&units=metric") else {fatalError("Missing URL")}

        let urlRequest = URLRequest(url: url)

       let(data, response) = try await URLSession.shared.data(for: urlRequest)
        guard (response as? HTTPURLResponse)?.statusCode == 200 else{ fatalError("Error
fetching data")}
        let decodeData = try JSONDecoder().decode(ResponseBody.self, from: data)

        return decodeData
    }
}
// Model of the response body we get from calling the OpenWeather API
struct ResponseBody: Decodable {
    var coord: CoordinatesResponse
    var weather: [WeatherResponse]
    var main: MainResponse
    var name: String
    var wind: WindResponse
    struct CoordinatesResponse: Decodable {
        var lon: Double
        var lat: Double
    }
    struct WeatherResponse: Decodable {
        var id: Double
        var main: String
        var description: String
        var icon: String
    }
    struct MainResponse: Decodable {
        var temp: Double
        var feels_like: Double
        var temp_min: Double
        var temp_max: Double
        var pressure: Double
        var humidity: Double
    }
```

```
    struct WindResponse: Decodable {
        var speed: Double
        var deg: Double
    }
}
extension ResponseBody.MainResponse {
    var feelsLike: Double { return feels_like }
    var tempMin: Double { return temp_min }
    var tempMax: Double { return temp_max }
}


//
//  WeatherAppApp.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush
//
import SwiftUI
@main
struct WeatherAppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}


//
//  Extensions.swift
//  WeatherApp
//
//  Created by Markiyan Vytrykush on 30/09/2022.
//
import Foundation
import SwiftUI
extension Double{
    func roundDouble()-> String{
        return String(format: "%.0f", self)
    }
}
extension View{
    func cornerRadius(_ radius: CGFloat, corners: UIRectCorner) -> some View{
        clipShape(RoundedCorner(radius: radius, corners: corners))
    }
}
// Custom RoundedCorner shape used for cornerRadius extension above
```

```
struct RoundedCorner: Shape {
    var radius: CGFloat = .infinity
    var corners: UIRectCorner = .allCorners
    func path(in rect: CGRect) -> Path {
        let path = UIBezierPath(roundedRect: rect, byRoundingCorners: corners, cornerRadii:
CGSize(width: radius, height: radius))
        return Path(path.cgPath)
    }
}
```