УДК 004.9
DOI https://doi.org/10.32782/IT/2024-2-20

Сергій ЧОРНОНОГ
спеціаліст напряму електронні системи, Київський національний університет технологій та дизайну, вул. Мала Шияновська, 2, м. Київ, Україна, 01011
***ORCID:** 0009-0006-9364-8827*

Бібліографічний опис статті: Чорноног, С. (2024). Дослідження проблеми міграції MVVM архітектури на сучасні iOS-додатки побудовані на SwiftUI. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 2, 153–159, doi*:* https://doi.org/10.32782/IT/2024-2-20

ДОСЛІДЖЕННЯ ПРОБЛЕМИ МІГРАЦІЇ MVVM АРХІТЕКТУРИ НА СУЧАСНІ IOS-ДОДАТКИ ПОБУДОВАНІ НА SWIFTUI

Досвідчені розробники програмного забезпечення з багатьма роками досвіду при переході на новий технічний стек можуть, не помічаючи цього, створювати собі додаткові складності в новому оточенні, шляхом перевикористання свого минулого досвіду в новому контексті. Це працює в багатьох випадках, але іноді, це може додавати не потрібної комплексності проєкту, що збільшує кількість коду та ускладнює його підтримку та розширення. В проєктах побудованих для екосистеми Apple все будується навколо бібліотеки, на основі якої буде побудовано графічний інтерфейс додатку. Абсолютна більшість розробників в цій екосистемі починала свій шлях з фреймворків створених компанією Apple UIKit або AppKit.

***Мета роботи.** В рамках статті розглянута проблема використання архітектури MVVM для додатків, інтерфейс яких, побудований на SwiftUI. Дослідження спрямоване на висвітлення використання готових, вбудованих у SwiftUI рішень, для отримання переваг концептуально закладених в MVVM архітектуру не використовуючи її.*

Також, була розглянута проблема часто не правильного розуміння зони відповідальності Model шару.

***Методологія.** Проведено теоретичний аналіз декількох архітектур, які широко використовуються у побудові додатків в екосистемі Apple. Для наочності, створені числені приклади для охоплення як простих, так і більш складних випадків.*

***Наукова новизна.** Компанія Apple у своїх навчальних матеріалах WWDC по SwiftUI ніколи не робила акцент на архітектурі додатків, але наводила численні приклади використання нових фреймворків. Для написання статті були розглянуті та проаналізовані наявні навчальні матеріали які стосуються обраної теми. Було відстежено причинно-наслідковий зв'язок непотрібного ускладнення коду при переході з UIKit/AppKit на SwiftUI через вплив минуло досвіду розробника. Було введено термін MV архітектури, яка є нативною для SwiftUI додатків.*

***Висновки.** У результаті проведеного дослідження було визначено ряд аргументів згідно яких, використання MVVM архітектури в додатку, інтерфейс якого побудовано на SwiftUI, є недоречним та зайвим. Також, було окреслено ряд проблем які виникають при використанні цього шаблону та як їх уникнути. Було сфокусовано увагу на конкретних важливих деталях реалізації всіх шарів додатку використовуючи MV архітектуру.*

***Ключові слова:** архітектура SwiftUI, MVC, MVVM, MV, iOS розробка.*

Serhii CHORNONOH
Master of Electronic Systems, Kyiv National University of Technologies and Design, 2, Mala Shyianovska Str., Kyiv, Ukraine, 01011, chornonoh.serhii@gmail.com
***ORCID:** 0009-0006-9364-8827*

To cite this article: Chornonoh, S. (2024). Doslidzhennya problemy mihratsiyi MVVM arkhitektury na suchasni iOS-dodatky pobudovani na SwiftUI [Research on the problem of migrating MVVM architecture to modern iOS applications built with SwiftUI]. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 2, 153–159, doi: https://doi.org/10.32782/IT/2024-2-20

RESEARCH ON THE PROBLEM OF MIGRATING MVVM ARCHITECTURE TO MODERN IOS APPLICATIONS BUILT WITH SWIFTUI

Experienced software developers with many years of experience may unknowingly create additional complexity for themselves in a new environment by reusing their past experience in a new context when switching to a new technical stack. This works in many cases, but sometimes it can add unnecessary complexity to the project, which

increases the amount of code and makes it difficult to maintain and extend. In projects built for the Apple ecosystem, everything is built around a library on which the application's graphical interface will be built. The vast majority of developers in this ecosystem started their journey with frameworks created by Apple, such as UIKit or AppKit.

Objective of the study. *The article deals with the problem of using the MVVM architecture for applications whose interface is built on SwiftUI. The research is aimed at highlighting the use of ready-made solutions built into SwiftUI to take advantage of the benefits conceptually embedded in the MVVM architecture without using it.*

Also, the problem of often misunderstanding the area of responsibility of the Model layer was considered.

Methodology. *A theoretical analysis of several architectures that are widely used in building applications in the Apple ecosystem is carried out. For clarity, numerous examples are created to cover both simple and more complex cases.*

Scientific novelty. *Apple has never focused on the application architecture in its WWDC educational materials on SwiftUI, but provided numerous examples of the use of new frameworks. To write this article, we have reviewed and analysed the available educational materials related to the chosen topic. The authors traced the cause-and-effect relationship of unnecessary code complexity when switching from UIKit/AppKit to SwiftUI due to the influence of the developer's past experience. The term MV architecture was introduced, which is native to SwiftUI applications.*

Conclusions. *As a result of the work, a number of arguments have been identified according to which the use of MVVM architecture in an application whose interface is built using SwiftUI is inappropriate and unnecessary. Also, a number of problems that arise when using this pattern and how to avoid them were outlined. Attention was focused on specific important details of implementing all application layers using MV architecture.*

Key words: *SwiftUI architecture, MVC, MVVM, MV, iOS development.*

Актуальність проблеми. На момент написання статті, коли SwiftUI існує вже біля 5 років, спільнота iOS розробників продовжує називати його не готовим до використання в реальних проєктах. На мою думку, це відбувається тому, що розробники намагаються використовувати свій минулий досвід написання коду не усвідомлюючи, що зараз вони працюють з концептуально іншим User Interface (UI) фреймворком.

UIKit – це імперативний фреймворк, що передбачає опис програми як послідовність інструкцій зміни стану. SwiftUI, в свою чергу, використовує декларативний підхід, відповідно до якого, програма описує, який результат необхідно отримати (Mejia, 2019).

Не усвідомлення того, що ці фреймворки концептуально різні, призводить до декількох проблем. В рамках цієї статті буде розглянута одна з них, а саме: використання архітектури MVVM (Model-View-ViewModel)(і похідних від неї), яка є найпопулярнішою для iOS додатків, інтерфейс яких, побудований на UIKit, для додатків створених на SwiftUI.

Аналіз останніх досліджень і публікацій. Як показує досвід та наукові праці багатьох інженерів та розробників програмного забезпечення, впровадження та слідування архітектурам в сучасних проєктах є обов'язковою умовою. Це полегшує підтримку такої програми, надає чітку і зрозумілу структуру написаному коду та спрощує процес розширення та додавання нових функцій.

Тема найбільш поширених архітектур для побудови iOS додатків та їх порівняння в найважливіших аспектах добре розкрита в науковій праці «Review of iOS Architectural pattern for testability, modifiability, and performance quality» (Fauzi Sholichin, Monh Adham Bin Isa, Shahliza Abd Halim, Muhammad Firdaus Bin Harun, 2019). Більш детально найпопулярніші шаблони розглянуті в книжці «App Architecture: iOS Application Design Patterns» (Chris Eidhof, Matt Gallagher, Florian Kugler, 2018). В межах цієї статті вважаю за потрібне сфокусувати увагу на двох з них: MVC та MVVM.

Метою дослідження є пояснити недоцільність використання MVVM архітектури для додатків, інтерфейс яких базується на SwiftUI, та показати на прикладах вбудовані у SwiftUI інструменти, за допомогою яких реалізується зв'язок між різними шарами програми, що виступає одним із аргументів недоречності використання цього шаблону. Вторинною метою роботи є звернення уваги на часто не правильну реалізації Model шару, що в подальшому додає складності проєкту.

Виклад основного матеріалу дослідження.

Архітектура MVC

Компанія Apple, яка є творцем обох фреймворків, активно використовує у своїх навчальних матеріалах архітектуру MVC (Model-View-Controller) пояснюючи це тим, що ця архітектура чудово підходить для Cocoa проєктів. MVC надає об'єктам одну з трьох ролей: Model, View або Controller. Шаблон визначає не тільки роль кожного об'єкту, а й спосіб і правила їх взаємодії один з одним. Об'єкти розділені абстрактними рамками та комунікують на рівні цих абстракцій (рис. 1).

Model об'єкти інкапсулюють дані, специфічні для програми, і визначають логіку та обчислення, які обробляють та змінюють ці дані. Наприклад, Model може представляти

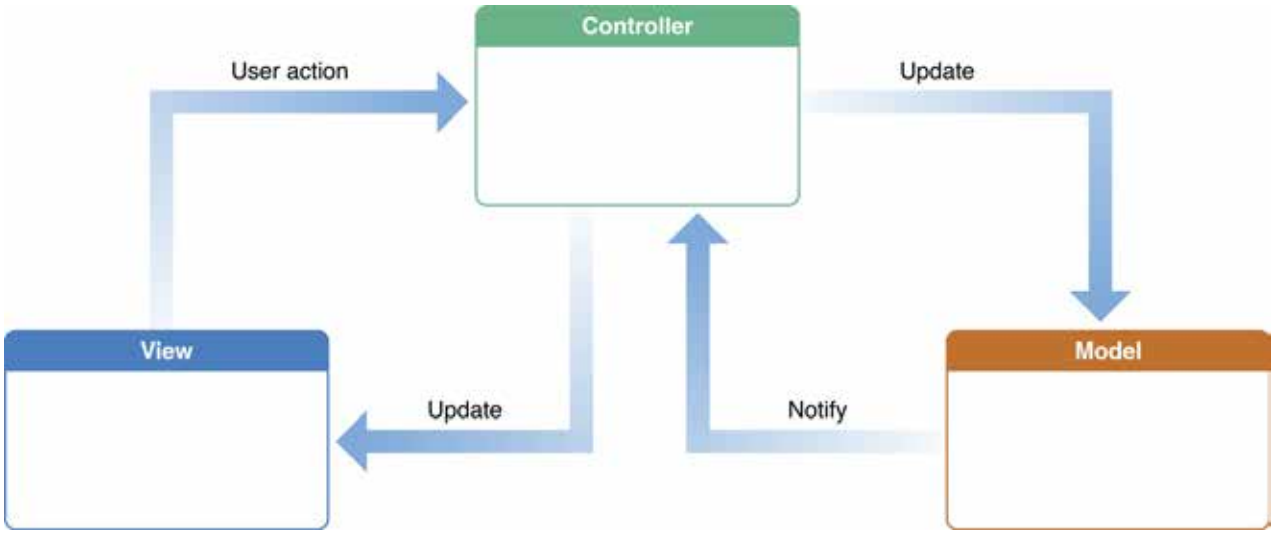

Рис. 1. Схема архітектури MVC

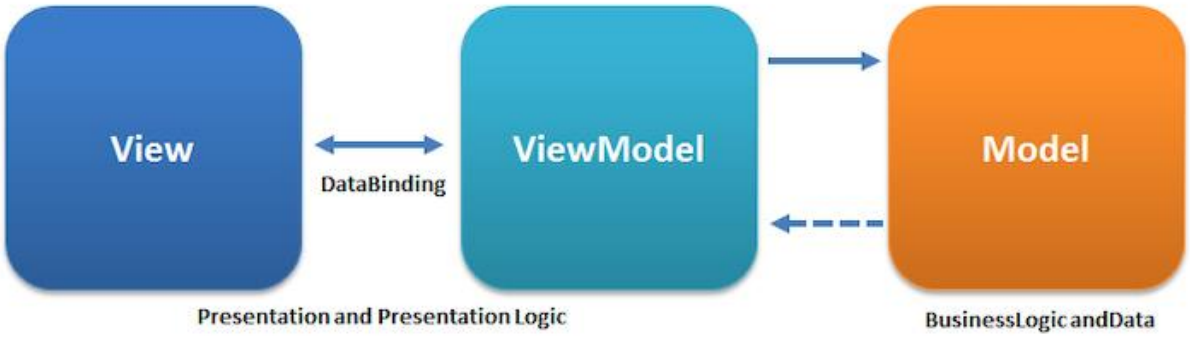

Рис. 2. Схема архітектури MVVM

собою книгу, яка має автора, зміст, сторінки, масив поміток або закладок та вміє змінювати свій стан.

View об'єкти представляють UI елементи, які відтворюються на екрані пристрою. View знає як малювати себе та реагувати на дії користувача. Основне його завдання, відобразити дані з Model та вміти ініціювати їх зміни при необхідності.

Controller об'єкти виступають як посередник між одним або декількома Model та одним або декількома View. Таким чином, Controller повідомляє View, про зміни Model, для ініціації процесу оновлення View, та навпаки, змінює Model, при ініціації цих змін користувачем через View.

На практиці, в контексті додатків, які базуються на UIKit, Controller-а в чистому вигляді не існує. Кожен екран iOS додатку повинен мати ViewController, як ядро цього екрану. Відповідно, вже на цьому етапі в нас відбувається з'єднання двох абстракцій в одну.

В свою чергу, часто, Model представляють собою структури даних без логіки зміни чи оброблення цих даних всередині(в спільноті відомі як POJO (Plain Old Java Object). Але вимоги до програми від цього не змінюються, відповідно хтось має вміти змінювати ці данні. Як правило, це робить ViewController. Як результат, ми отримуємо головну проблему MVC підходу, яка відома як MassiveViewController.

Будь-який iOS/macOS/tvOS/watchOS розробник знайомий з MVC архітектурою і знає всі переваги та недоліки цього шаблону. Через намагання вирішити деякі очевидні проблеми MVC, спільнота почала використовувати запозичену з інших платформ MVVM архітектуру, і це, на мою думку, дійсно вирішує головну проблему MVC яку ми визначили раніше.

Архітектура MVVM

MVVM – це архітектура розроблена співробітниками компанії Microsoft, а саме Кеном Купером та Тедом Пітерсом. Шаблон явно відокремлює бізнес логіку додатку від інтерфейсу. Головним завданням MVVM є зробити View повністю незалежним компонентом від логіки додатку (Gallardo, 2023).

```swift
struct Book: Identifiable {
    let id: Int
    let title: String
    let author: String
    let text: String
}

class BookListViewModel: ObservableObject {
    @Published var books: [Book] = []

    func fetchBooks() async throws {•••}
}

struct BookListView: View {
    @StateObject var viewModel = BookListViewModel()

    var body: some View {
        VStack {
            ForEach(viewModel.books) { book in
                BookView(book: book)
            }
        }
        .task {
            try? await viewModel.fetchBooks()
        }
    }
}
```

```swift
struct BookView: View {
    let book: Book

    var body: some View {
        HStack {
            Text(book.title)

            Text(book.author)
        }
    }
}
```

Рис. 3

Погодьтесь, дуже схоже на схему MVC. Фактично, замість Controller ми тут маємо ViewModel, яка має виконувати ті самі функції що були покладені на Controller. Чудовий приклад того, що те, як ми називаємо сутності або абстракції в коді, є дуже важливою деталлю, тому що це спрацювало.

ViewController тепер представляє View, Model залишається без змін і додається ще один новий об'єкт – ViewModel, який виступає посередником між View та Model. Повідомляє першого, що стан другого змінився, і навпаки. Навіть у випадку, коли ми не зовсім правильно використовуємо Model об'єкти (коли це просто POJO), наша бізнес логіка дуже чітко відокремлена від View, що позитивно впливає на читабельність коду та можливість розширення функціоналу.

Вважаю, MVVM дійсно чудовий варіант при виборі архітектури для UIKit додатку і цілком зрозуміло розробників, які починають використовувати цей шаблон в своїх перших SwiftUI проєктах.

MVVM у SwiftUI

Нижче розглянемо як виглядає найпростіший варіант одного екрану додатку побудованого на SwiftUI за архітектурою MVVM:

Згідно шаблону наша ViewModel виступає посередником між Model та View, тримає в собі бізнес логіку та при потребі спілкується з іншими сервісами додатку.

В середині SwiftUI View треба намагатись тримати тільки код пов'язаний з описом інтерфейсу і нічого більше. Це надасть швидкості її відтворення на екрані.

Структура Book відноситься до шару Model. В даному випадку вона представляє специфічні для додатку дані.

Архітектура MV

Але що використовує Apple у своїх навчальних матеріалах в контексті SwiftUI? Це щось схоже на MVC але вже без C, тобто MV (Model-View). Якщо в нас тепер не має Controller-а, хто виступає посередником між Model та View? Як реалізувати двонаправлене прив'язування (bidirectional binding)? Хто тепер відповідає за бізнес логіку? Справа в тому, що SwiftUI має вбудовану можливість спостереження за змінами (та їх зміненням) value type об'єктів (таких як struct та enum) за допомогою обгорток (propertyWrapper) таких як @State та @Binding та спостереженням за змінами (та їх зміненням) reference type об'єктів (class або actor) за допомогою @StateObject або @ObservedObject.

```swift
struct Book: Identifiable {
    let id: Int
    let title: String
    let author: String
    let text: String

    // Фабрика
    static func all() async -> [Book] {•••}
}
struct BookListView: View {
    @State var books: [Book] = []

    var body: some View {
        VStack {
            ForEach(books) { book in
                BookView(book: book)
            }
        }
        .task {
            books = await Book.all()
        }
    }
}
```

```swift
struct BookView: View {
    let book: Book

    var body: some View {
        HStack {
            Text(book.title)

            Text(book.author)
        }
    }
}
```

Рис. 4

Адаптуємо наш попередній приклад до MV архітектури:

В попередньому випадку сутність Book, яка належить до шару Model, була так званою POJO без будь-якої логіки. Вона може бути такою, але пропоную повернутись до класичного розуміння шару Model з MVC архітектури, згідно якого саме тут має бути логіка та змінення цих даних. В конкретно нашому випадку, ми можемо використати шаблон Фабрика, для реалізації аналогу функції ViewModel *fetchBooks*. Так як наша ViewModel тепер не має бізнес логіки, а двонаправлене прив'язування ми можемо реалізувати вбудованими обгортками SwiftUI-ю, в ній не має жодного сенсу. Видаляємо.

В реальних проєктах така реалізація може бути використана на екранах з досить простою внутрішньою логікою, наприклад, простого відтворення даних.

Що стосовно більш комплексних випадків? Додаткова логіка залишається у Model шарі у трошки зміненому вигляді. Для цих сутностей назва не так важлива, але Apple у своїх навчальних матеріалах використовує суфікс Store. Зазвичай це class, який слідує протоколу ObservableObject і побудований по принципу єдиного обов'язку (single responsibility):

Може здатися, що наш BookStore перетворився у ViewModel, але різниця полягає в тому, що за MVVM архітектурою кожен екран повинен мати окрему ViewModel, в якій реалізована логіка окремо взятого екрану. Також, у випадку з ViewModel, використати її знову в інших частинах додатку стає досить складно і не раціонально, якщо вона потребує доробок. У випадку ж із Store, ми можемо дуже просто використати таку сутність знову будь де. Ми наче створюємо цеглини, які відносяться до шару Model, і можуть бути як завгодно скомбіновані та використані знову у будь-якій View. Також, логіка більш рівномірно розподіляється в середині Model шару між різними об'єктами.

Зверніть увагу, що залежність Store доставляється у View за допомогою @ EnvironmentObject. Це спеціальний механізм SwiftUI який вводить поняття оточення і дозволяє отримати доступ до об'єкту з оточення будь якій View нижче по ієрархії. Зазвичай такі глобально необхідні об'єкти ініціюються десь біля кореня додатку або можуть бути ініційовані по шаблону Singleton.

Починаючи з iOS 17.0 інженери Apple ще більше спростили механізм нагляду за змінами reference type об'єктів, тому як альтернатива слідування протоколу ObservableObject, може

```swift
class BookStore: ObservableObject {
    @Published var books: [Book] = []
    @Published var cart: [Book] = []

    func load() async {...}
    func cartAddBook(_ book: Book) {...}
    func cartRemoveBook(_ book: Book) {...}
    func isInCart(_ book: Book) -> Bool {...}
    func clearCart() {...}

    func purchase() async throws {...}
}
struct BookListView: View {
    @EnvironmentObject var store: BookStore

    var body: some View {
        VStack {
            ForEach(store.books) { book in
                let inCart = store.isInCart(book)
                BookView(book: book, inCart: inCart)
                    .onTapGesture {
                        inCart ?
                            store.cartRemoveBook(book) :
                            store.cartAddBook(book)
                    }
            }
        }
        .task {
            await store.load()
        }
    }
}
```

```swift
struct Book: Identifiable {
    let id: Int
    let title: String
    let author: String
    let text: String

    // Фабрика
    static func all() async -> [Book] {...}
}

struct BookView: View {
    let book: Book
    let inCart: Bool

    var body: some View {
        HStack {
            Text(book.title)

            Text(book.author)

            Image(systemName: cartIcon)
        }
    }

    var cartIcon: String {
        inCart ? "cart.fill" : "cart"
    }
}
```

Рис. 5

```swift
// iOS 13...16
class Library: ObservableObject {
    @Published var books: [Book] = []
}
```

```swift
// iOS 17+
@Observable class Library {
    var books: [Book] = []
}
```

Рис. 6

бути використання макросу @Observable, а для оголошення та зберігання такого екземпляру класу, нам вже відому @State обгортку.

SwiftUI автоматично буде наглядати за змінами усіх властивостей такого об'єкту, а для відключення цього для однієї конкретної властивості можна використати @ObservationIgnored макрос.

Висновки і перспективи подальших досліджень. У результаті проведеної роботи було визначено ряд аргументів згідно яких, використання MVVM архітектури в додатку, інтерфейс якого побудовано на SwiftUI, є недоречним та зайвим. Використання цього шаблону у UIKit проєктах, часто призводить до не правильного розуміння і реалізації шару Model та, в подальшому, міграції цих проблем у SwiftUI додатки. Виправлення наших архітектурних помилок в комбінації з використанням чистого SwiftUI дає розуміння, що ViewModel шар вже вбудований в реалізацію фреймворку SwiftUI, тобто, використання такого додає тільки складності проєкту не даючи переваг.

Було розглянуто використання MV шаблону як для простих, так і для більш складних частин додатку.

Найбільшою перевагою використання MV шаблону, є слідування принципу DRY (Don't Repeat Yourself), згідно якого ми не дублюємо однакову бізнес логіку для кожної ViewModel, в якій вона потрібна. Ми використовуємо знову наш шар Model, в необхідній комбінації сутностей для конкретного екрану. Менша кількість коду, в свою чергу, полегшує його підтримку, розширення та зменшує кількість помилок в ньому.

Важливим моментом, також, є слідування принципу KISS (Keep It Simple Stupid).

Безумовно, приємно писати інженерно складний та ефективний код, але це досить відчутно підвищує кількість часу та зусиль, яких вимагає робота з таким кодом в майбутньому.

Починаючи використовувати SwiftUI необхідно розуміти, що концептуально це інший фреймворк. Використання підходів, які зарекомендували себе дійсно добре в UIKit, можуть створити тут, в новому оточені, додаткові проблеми. На жаль, минулий досвід розробника може мати як позитивну, так і негативну сторону в цьому випадку.

В перспективі подальших досліджень слід розглянути варіанти тестування iOS додатків побудованих на SwiftUI за архітектурою MV.

ЛІТЕРАТУРА:

1. Fauzi Sholichin, Monh Adham Bin Isa, Shahliza Abd Halim, Muhammad Firdaus Bin Harun, Review of iOS Architectural pattern for testability, modifiability, and performance quality, 2019.

2. Robert Mejia, Declarative and Imperative Programming using SwiftUI and UIKit, 2019. URL: https://medium.com/@rmeji1/declarative-and-imperative-programming-using-swiftui-and-uikit-c91f1f104252

3. Apple Developer Documentation, SwiftUI. URL: https://developer.apple.com/documentation/swiftui/

4. Baeldung, What is Pojo Class?, 2024. URL: https://www.baeldung.com/java-pojo-class

5. Apple Developer Documentation, Cocoa Core Competencies. URL: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

6. Estefanía García Gallardo, What is MVVM Architecture?, 2023. URL: https://builtin.com/software-engineering-perspectives/mvvm-architecture

7. Apple Developer Documentation, Model data. URL: https://developer.apple.com/documentation/swiftui/model-data

8. Apple Developer Documentation, Managing model data in your app. URL: https://developer.apple.com/documentation/swiftui/managing-model-data-in-your-app

9. Chris Eidhof, Matt Gallagher, Florian Kugler, App Architecture: iOS Application Design Patterns in Swift, 2018.

10. Apple Developer Documentation, Migrating from the Observable Object protocol to the Observable macro. URL: https://developer.apple.com/documentation/swiftui/migrating-from-the-observable-object-protocol-to-the-observable-macro

11. Apple Developer Documentation, ObservationIgnored. URL: https://developer.apple.com/documentation/Observation/ObservationIgnored()

REFERENCES:

1. Fauzi Sholichin, Monh Adham Bin Isa, Shahliza Abd Halim, Muhammad Firdaus Bin Harun. (2019). Review of iOS Architectural pattern for testability, modifiability, and performance quality,

2. Robert, Mejia. (2019). Declarative and Imperative Programming using SwiftUI and UIKit, Retrieved from https://medium.com/@rmeji1/declarative-and-imperative-programming-using-swiftui-and-uikit-c91f1f104252

3. Apple Developer Documentation, SwiftUI. Retrieved from https://developer.apple.com/documentation/swiftui/

4. Baeldung, What is Pojo Class?, 2024. Retrieved from https://www.baeldung.com/java-pojo-class

5. Apple Developer Documentation, Cocoa Core Competencies. Retrieved from https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

6. Estefanía García Gallardo, What is MVVM Architecture?, 2023. Retrieved from https://builtin.com/software-engineering-perspectives/mvvm-architecture

7. Apple Developer Documentation, Model data. Retrieved from https://developer.apple.com/documentation/swiftui/model-data

8. Apple Developer Documentation, Managing model data in your app. Retrieved from https://developer.apple.com/documentation/swiftui/managing-model-data-in-your-app

9. Chris, Eidhof, Matt, Gallagher, Florian, Kugler. (2018). App Architecture: iOS Application Design Patterns in Swift,

10. Apple Developer Documentation, Migrating from the Observable Object protocol to the Observable macro. Retrieved from https://developer.apple.com/documentation/swiftui/migrating-from-the-observable-object-protocol-to-the-observable-macro

11. Apple Developer Documentation, ObservationIgnored. Retrieved from https://developer.apple.com/documentation/Observation/ObservationIgnored()