แนวทางในการจัดกิจกรรมภาคปฏิบัติ

หัวข้อการเรียนรู้ :
สถาปัตยกรรมแบบ Model–View–ViewModel (MVVM)ในการพัฒนา iOS Application ด้วย SwiftUI

ระยะเวลาในการทำกิจกรรม: 180 นาที

วัตถุประสงค์ในการทำกิจกรรม :

1. การอธิบายองค์ประกอบของสถาปัตยกรรม MVVM (Model, View, ViewModel) และเชื่อมโยงกับแนวคิด Separation of Concerns (SoC) ได้
2. สร้างแอปด้วย SwiftUI โดยใช้ MVVM เพื่อจัดการ State และตรรกะการทำงานของแอปได้อย่างเป็นระบบ
3. ใช้ ObservableObject, @Published และ @StateObject ร่วมกับ SwiftUI เพื่ออัปเดต UI แบบ Reactive ผ่าน Combine Framework ได้

ความคิดรวบยอด:
กิจกรรมการเรียนรู้ชุดนี้มุ่งพัฒนาความรู้และความเข้าใจของผู้เรียนเกี่ยวกับการนำ**สถาปัตยกรรม MVVM (Model, View, ViewModel)** มาใช้ในการออกแบบโครงสร้างการทำงานของแอป โดยแยกความรับผิดชอบของแต่ละส่วนออกจากกันอย่างเป็นระบบตาม**หลักการ Separation of Concerns (SoC)** ซึ่งเป็นหลักสำคัญของการออกแบบซอฟต์แวร์คุณภาพสูง นอกจากนี้ ผู้เรียนจะได้เรียนรู้การใช้**เฟรมเวิร์ก Combine** เพื่อจัดการสถานะ (State) และเชื่อมโยงการเปลี่ยนแปลงของข้อมูลเข้ากับ**การอัปเดต UI แบบ Reactive** ซึ่งทำให้แอปสามารถตอบสนองต่อเหตุการณ์และข้อมูลที่เปลี่ยนแปลงแบบอะซิงโครนัสได้อย่างมีประสิทธิภาพ แนวคิดเหล่านี้จะช่วยส่งเสริมให้ผู้เรียนสามารถพัฒนาแอปที่มีโครงสร้างชัดเจน ดูแลรักษาง่าย และสามารถรองรับการขยายตัวของฟีเจอร์ใหม่ในอนาคตได้อย่างยั่งยืน

บทความสำหรับอ่านประกอบการทำกิจกรรม
https://ajthiti.gitbook.io/develop-in-swift/getting-started/mvvm-in-swiftui

แนวทางในการจัดกิจกรรมการเรียนรู้

ขั้นนำเข้าสู่บทเรียน (Warm-up)

ในช่วงเริ่มต้นของบทเรียน ครูอาจเริ่มจากการชวนผู้เรียนมองสิ่งใกล้ตัวผ่านคำถามง่าย ๆ เช่น ถ้าเราเป็นประธานในการจัดงานกีฬาสีของโรงเรียน และต้องทำทุกอย่างด้วยตัวเองตั้งแต่จัดกิจกรรม ดูแลอุปกรณ์ ถือธงเชียร์ และรายงานคะแนน—จะเกิดอะไรขึ้นบ้าง? (ชวนให้ผู้เรียนคิดเรื่องความวุ่นวาย ความซ้ำซ้อน และความยากในการประสานงาน) และถ้าเราแบ่งหน้าที่ในการทำงานเป็นฝ่าย โดยแต่ละฝ่ายมีหน้าที่ชัดเจนและทำงานของตัวเองได้ดี งานกีฬาสีจะมีผลลัพธ์ต่างจากกรณีแรก อย่างไร ? (สะท้อน SoC: เมื่อแยก Concerns งานจะราบรื่นขึ้น)

หรือลองคิดดูต่อว่า ถ้าอยากเพิ่มกิจกรรมใหม่ เช่น การเต้นประกอบเพลงหรือซุ้มอาหาร ต้องแก้ไขฝ่ายไหนบ้าง? ทุกฝ่ายต้องแก้หมดหรือแก้เฉพาะฝ่ายที่รับผิดชอบ? (เชื่อมโยงกับการขยายฟีเจอร์ในแอป—แก้เฉพาะชั้นที่เกี่ยวข้อง)

นอกจากนี้ อาจเปรียบเทียบเพิ่มเติมด้วยตัวอย่างเช่น ร้านอาหาร: พ่อครัว–พนักงานเสิร์ฟ–แคชเชียร์ ถ้าพนักงานเสิร์ฟต้องไปทำทุกอย่างเอง เช่น รับออเดอร์ + ไปเก็บเงิน + เข้าครัว → เกิดความวุ่นวาย ทำงานผิดพลาดง่าย และร้านเติบโตยาก

ขั้นการเรียนรู้ (Learn)

เมื่อผู้เรียนมีความสนใจและเข้าใจบริบทของการแบ่งหน้าที่การทำงานตามหลัก SoC จากสถานการณ์จริงแล้ว ครูสามารถชี้ให้เห็นความเชื่อมโยงกับการออกแบบแอปพลิเคชัน SwiftUI ได้อย่างชัดเจน ดังนี้:

ใน SwiftUI หากเราเขียนโค้ดแบบรวมทุกอย่างไว้ใน View เดียว—แสดงผล เก็บข้อมูล ควบคุมสถานะ ประมวลผล และตรวจเงื่อนไขต่าง ๆ — โค้ดจะมีพฤติกรรมเหมือนประธานกีฬาสีที่ต้องทำงานทั้งหมดเอง นั่นคือหนักเกินไป ซับซ้อน และทำให้ดูแลรักษาหรือพัฒนาต่อได้ยากมาก นี่คือสิ่งที่เรียกว่า **Massive View** และเป็นปัญหาที่เกิดขึ้นบ่อยในโค้ดของผู้เริ่มต้น SwiftUI โดยครูยกตัวอย่างโค้ด ดังต่อไปนี้

```swift
import SwiftUI

struct ContactView: View {

    // เก็บข้อมูล Contact ของสมาชิกไว้ใน View (ไม่แยก Model)
    @State private var name: String = "Jessica Anderson"
    @State private var dateOfBirth: String = "28 Aug 1979"
    @State private var gender: String = "Female"
    @State private var occupation: String = "Artist"
    @State private var salary: Int = 32000
    @State private var statusOfMembership: Bool = true

    var body: some View {

        VStack(alignment: .leading, spacing: 8) {
            // การแสดงผล UI
            Text("Name: \(name)")
            Text("Date of Birth: \(dateOfBirth)")
            Text("Gender: \(gender)")
            Text("Occupation: \(occupation)")
            Text("Salary: \(salary) THB")
            Text("Status: \(statusOfMembership ? "Active" : "Inactive")")
                .bold()
                .foregroundColor(statusOfMembership ? .green : .red)
                .padding(.top, 10)

            // ปุ่มที่มี Logic อยู่ใน View โดยตรง
            Button("Change Status") {
                changeStatus()          // UI ควบคุม Logic เอง
            }
            .buttonStyle(.borderedProminent)
            .padding(.top, 12)
        }
        .padding()
    }

    // Logic: การเปลี่ยนสถานะสมาชิก ถูกสร้างอยู่ใน View
    func changeStatus() {
        statusOfMembership.toggle()
    }
}
```

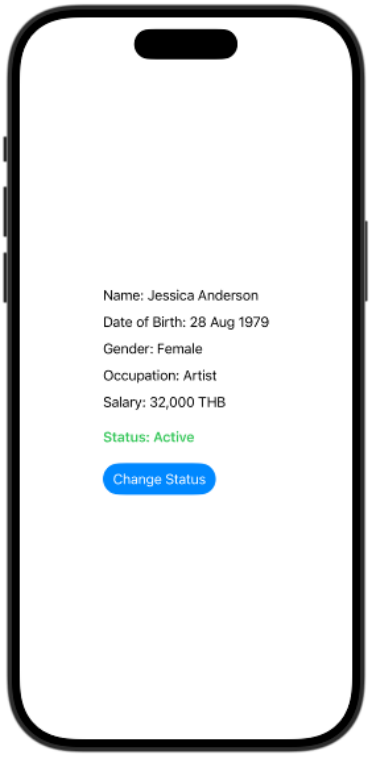
ชวนผู้เรียนสังเกตและอภิปรายร่วมกันในประเด็นต่างๆ ดังต่อไปนี้

- ใน View นี้ มี “หน้าที่” อะไรบ้าง ?
- เปรียบเทียบกับแนวคิด **Separation of Concerns**: ส่วนแสดงผลควรทำหน้าที่อะไรบ้าง และ “ไม่ต้องทำอะไร” บ้าง ?

ผู้สอนสรุปปัญหาแบบสั้น ๆ ว่า การรวมทุกอย่างใน View ทำให้เกิด Massive View ซึ่งยากต่อการทดสอบ(เนื่องจาก Logic ผูกติดกับ UI) และยากต่อการขยายฟีเจอร์ในอนาคต จากนั้นครูสามารถเชื่อมเข้าประเด็นสำคัญว่า:

เพื่อแก้ปัญหานี้ เราจึงต้องนำสถาปัตยกรรมแบบ Model–View–ViewModel (MVVM) มาใช้ เพราะเป็นแนวคิดที่ช่วยแยกความรับผิดชอบของแต่ละส่วนออกจากกันอย่างชัดเจนตามหลัก SoC ทำให้โค้ดอ่านง่าย ดูแลง่าย และขยายต่อได้โดยไม่กระทบส่วนอื่นของระบบ

กิจกรรมที่ 1 ทำความเข้าใจเกี่ยวกับแนวคิด Separation of Concerns และสถาปัตยกรรม MVVM



ขั้นที่ 1: ทำความเข้าใจโค้ด

1. ให้ผู้เรียนอ่านโค้ด ContactView อย่างละเอียด
2. ครูใช้ Xcode เพื่อทดลองรัน เพื่อดูหน้าตา UI และการทำงานของปุ่ม Change Status โดยมีจุดประสงค์เพื่อให้ผู้เรียนเห็นว่า “โค้ดทำงานได้” แต่เราจะสนใจ “โครงสร้าง” มากกว่า “ฟังก์ชันการทำงาน” เพียงอย่างเดียว

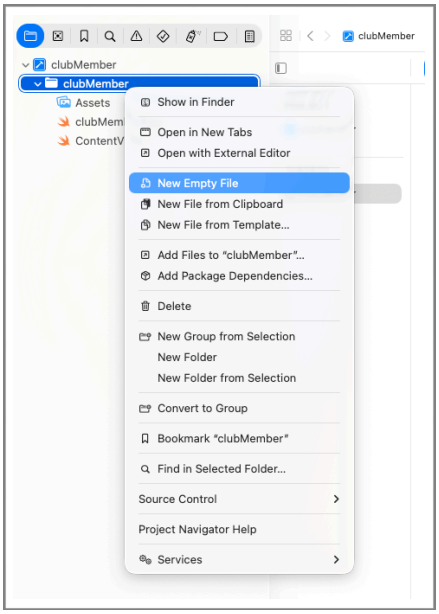
ขั้นที่ 2: ชี้ให้เห็นจุดที่ “View ทำงานมากเกินไป”

1. ระบุส่วนของโค้ดที่เป็น “ข้อมูลของสมาชิก” (Contact Data)
2. ระบุส่วนของโค้ดที่เป็น “การแสดงผล UI” (Presentation/UI)
3. ระบุส่วนของโค้ดที่เป็น “ตรรกะการทำงาน” (Business Logic)

และสรุปว่า ตอนนี้ ContactView กำลังทำหน้าที่ 3 อย่างผสมกันคือ Data + UI + Logic ซึ่งขัดกับหลัก Separation of Concerns (SoC) อย่างชัดเจน

ขั้นที่ 3 ให้ผู้เรียนลองระบุเพื่อแบ่งแยกความรับผิดชอบในโค้ดแต่ละส่วน เพื่อให้ผู้เรียนได้ฝึกคิดเชิงสถาปัตยกรรม โดยศึกษาบทความ เรื่อง MVVM in SwiftUI ประกอบการทำกิจกรรม

กิจกรรมที่ 2 การปรับปรุง (Refactor) จาก ContactView ไปเป็นโครงสร้างแบบ MVVM



1. ให้ผู้เรียนเปิด Xcode และสร้าง Project ใหม่ โดยเลือก Template เป็น **iOS > App**
2. ตั้งชื่อ Project เป็น **clubMember**
   - กำหนด Organization Identifier เพื่อสร้าง Bundle ID
   - กำหนดค่า Interface เป็น **SwiftUI**
   - กำหนดค่า Language เป็น **Swift**
   - กำหนดค่า Testing System และ Storage เป็น **None**
3. ลบไฟล์ ContentView.swift ออกจาก Project
4. เลือกคำสั่ง **New Empty File** เพื่อสร้างไฟล์ใหม่ จำนวน 3 ไฟล์ ดังนี้
   - Contact.swift —> Model
   - ContactViewModel.swift —> ViewModel
   - ContactView.swift —> View

5. แก้ไขโค้ดในไฟล์ clubMemberApp.swift ดังนี้

```
struct clubMemberApp: App {
    var body: some Scene {
        WindowGroup {
            ContactView()
        }
```

6. ให้ผู้เรียนลอง “ย้ายตำแหน่ง” ความรับผิดชอบในโค้ด ใส่ลงในไฟล์แต่ละไฟล์ให้เหมาะสม โดยศึกษาบทความเรื่อง MVVM in SwiftUI ประกอบการทำกิจกรรม

สร้างส่วนของ Model ในไฟล์ Contact.swift

```
import Foundation

struct Contact {
    let name: String
    let dateOfBirth: String
    let gender: String
    var weight: Double
    var height: Double
    var occupation: String
    var salary: Int
    var statusOfMembership: Bool
}
```

สร้างส่วนของ ViewModel ในไฟล์ ContactViewModel.swift

```
import SwiftUI
import Combine

class ContactViewModel: ObservableObject {

    @Published var clubMember = Contact(
        name: "Jessica Anderson",
        dateOfBirth: "28 Aug 1979",
        gender: "Female",
        weight: 65.7,
        height: 162.5,
        occupation: "Artist",
        salary: 32000,
        statusOfMembership: true
    )

    func changeStatus() {
        clubMember.statusOfMembership.toggle()
    }
}
```

และเขียนคำสั่งดังต่อไปนี้ในไฟล์ ContactView.swift

```
@StateObject private var viewModel = ContactViewModel()
```

7. ครูอธิบายเกี่ยวกับ **แนวคิด Reactive Programming** และ **การใช้ Combine เพื่อการอัปเดต UI แบบ Reactive**
    - SwiftUI ไม่ได้ทำงาน reactive ด้วยตัวเองทั้งหมด
    - ส่วนที่ทำให้ข้อมูลเปลี่ยนแล้ว UI เปลี่ยนคือ Combine Framework
    - Combine ทำงานผ่าน “Publisher–Subscriber Pattern”
    - ViewModel → คือ **Publisher** (ผู้กระจายสัญญาณเมื่อข้อมูลเปลี่ยน)
    - View → คือ **Subscriber** (ผู้ติดตามว่าต้องเปลี่ยน UI เมื่อข้อมูลเปลี่ยน)

    โดยองค์ประกอบอย่างเช่น ObservableObject, @Published และ @StateObject ที่อยู่ใน Combine Framework จะทำหน้าที่ต่างๆ ดังนี้
    - ObservableObject เป็นโปรโตคอลที่ทำให้ ViewModel สามารถส่งสัญญาณแจ้งการเปลี่ยนแปลงข้อมูล
    - @Published ทำหน้าที่ประกาศให้ระบบทราบเมื่อสถานะของข้อมูล (State) เกิดการเปลี่ยนแปลง
    - @StateObject ทำให้ View สามารถติดตามการเปลี่ยนแปลงของ ViewModel ได้อย่างต่อเนื่อง

    เมื่อ Publisher เปลี่ยน → Subscriber ได้รับข้อมูล → UI บนหน้าจอก็จะเปลี่ยนตามทันที

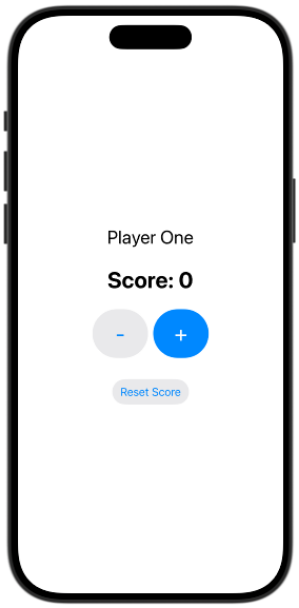
8. เขียนคำสั่งในการสร้างส่วนติดต่อกับผู้ใช้ ในไฟล์ ContactView.swift

```
import SwiftUI

struct ContactView: View {

    @StateObject private var viewModel = ContactViewModel()

    var body: some View {

        VStack(alignment: .leading, spacing: 8) {

            Text("Name: \(viewModel.clubMember.name)")
            Text("Date of Birth: \(viewModel.clubMember.dateOfBirth)")
            Text("Gender: \(viewModel.clubMember.gender)")
            Text("Occupation: \(viewModel.clubMember.occupation)")
            Text("Salary: \(viewModel.clubMember.salary) THB")
            Text("Status: \(viewModel.clubMember.statusOfMembership ? "Active" :
                 "Inactive")")
                .bold()
                .foregroundColor(viewModel.clubMember.statusOfMembership ? .green : .red)
                .padding(.top, 10)

            Button("Change Status") {
                viewModel.changeStatus()
            }
            .buttonStyle(.borderedProminent)
            .padding(.top, 12)

        }
        .padding()

    }
}


#Preview {
    ContactView()
}
```

ทดสอบการทำงานของแอปผ่าน Simulator และลองกดปุ่ม Change Status เพื่อสังเกตการเปลี่ยนแปลงของข้อความบนจอภาพ

9. ครูถามผู้เรียนด้วยคำถามต่อไปนี้เพื่อการสะท้อนความเข้าใจ
   - หลัง refactor แล้ว ส่วนไหนคือ **Model – View – ViewModel** ?
   - ตอนนี้ **View ทำหน้าที่อะไรบ้าง** ที่ต่างจากก่อน refactor?
   - ถ้าต้องเพิ่มฟังก์ชันใหม่ เช่น `updateSalary(by amount: Int)` ควรเพิ่มที่ไหน? View หรือ ViewModel? เพราะอะไร?
   - จากมุมมองของหลัก **Separation of Concerns** โครงสร้างใหม่ดีกว่าเดิมอย่างไร ?

กิจกรรมที่ 3 การสร้างแอป Player Score Tracker ด้วย MVVM



1. ให้ผู้เรียนเปิด Xcode และสร้าง Project ใหม่ โดยเลือก Template เป็น **iOS > App**
2. ตั้งชื่อ Project เป็น **PlayerScoreTracker**
   - กำหนด Organization Identifier เพื่อสร้าง Bundle ID
   - กำหนดค่า Interface เป็น **SwiftUI**
   - กำหนดค่า Language เป็น **Swift**
   - กำหนดค่า Testing System และ Storage เป็น **None**
3. ลบไฟล์ ContentView.swift ออกจาก Project
4. เลือกคำสั่ง **New Empty File** เพื่อสร้างไฟล์ใหม่ จำนวน 3 ไฟล์ ดังนี้
   - Player.swift —> Model
   - PlayerViewModel.swift —> ViewModel
   - PlayView.swift —> View

5. แก้ไขโค้ดในไฟล์ PlayerScoreTrackerApp.swift ดังนี้

```
struct PlayerScoreTrackerApp: App {
    var body: some Scene {
        WindowGroup {
            PlayerView()
        }
```

6. ให้ผู้เรียนออกแบบ "ระบบติดตามคะแนนผู้เล่นอย่างง่าย" ประกอบด้วย Player 1 คน ซึ่งมีชื่อผู้เล่น และ คะแนนเริ่มต้นเป็น 0 จากนั้นสร้างปุ่มเพื่อเพิ่มคะแนน และ ปุ่มเพื่อลดคะแนน โดยมีเงื่อนไขว่า คะแนนต้องไม่ติดลบ
   - ให้ผู้เรียนสร้าง Model ของผู้เล่นในไฟล์ Player.swift
   - ให้ผู้เรียนสร้าง ViewModel ในไฟล์ PlayerViewModel.swift
   - ให้ผู้เรียนสร้าง View ในไฟล์ PlayView.swift

7. สร้างความท้าทายโดยการขยายฟีเจอร์ของแอป เพื่อสะท้อนความเข้าใจเกี่ยวกับแนวคิด SoC
   - เพิ่มกฎว่า "ห้ามให้คะแนนเกิน 10"
   - เพิ่มปุ่ม Reset Score

ขั้นสรุปบทเรียน (Conclusion)

ภายหลังการทำกิจกรรม ครูควรช่วยผู้เรียนทบทวนและเชื่อมโยงความรู้ทั้งหมดในบทเรียนเข้าหากันอย่างเป็นระบบ เพื่อให้ผู้เรียนเห็น “ภาพใหญ่ว่าเหตุใดเราต้องใช้ MVVM” และ “บทบาทของ Combine ใน SwiftUI คืออะไร” โดยเสนอประเด็นสรุปดังต่อไปนี้:

- การรวม Logic และ UI ไว้ใน View เดียว ทำให้เกิด Massive View
- สถาปัตยกรรม MVVM ช่วยแบ่งหน้าที่ของโปรแกรมเป็นส่วนๆ อย่างชัดเจน
- Combine ทำหน้าที่เชื่อม State ระหว่าง ViewModel และ UI ของ SwiftUI

จากตัวอย่างในการสร้างแอป Player Score Tracker โดยออกแบบให้ View เรียกใช้ Method ของ ViewModel เช่น viewModel.changeStatus() หรือ viewModel.increaseScore() ช่วยให้ขยายระบบได้ง่าย เพราะ View ไม่จำเป็นต้องรู้รายละเอียดการทำงานภายในของ ViewModel และหากวันหนึ่งเราต้องเปลี่ยนเงื่อนไข เช่น คะแนนสูงสุด การ validate ค่า หรือการบันทึกลงฐานข้อมูล เราก็จะสามารถแก้ไขคำสั่งเฉพาะใน ViewModel โดยไม่ต้องแตะต้องโค้ดใน View เลย ซึ่งแสดงให้เห็นถึงประโยชน์ของ MVVM ที่รองรับการเปลี่ยนแปลงในอนาคตได้ดีมาก

คำถามเพื่อการสะท้อนผลการเรียนรู้จากการทำกิจกรรม

1. นักเรียนสามารถอธิบายได้หรือไม่ว่า เหตุใดการรวมทุกอย่างไว้ใน View เดียวจึงทำให้โค้ดกลายเป็น Massive View และขัดกับหลัก Separation of Concerns (SoC)
2. การแยกโครงสร้างเป็น Model–View–ViewModel ช่วยให้ระบบซอฟต์แวร์จัดการง่ายขึ้นอย่างไรบ้าง? ยกตัวอย่างจากโค้ด Contact หรือ Player Score Tracker ที่สร้างขึ้น
3. เมื่อเพิ่มฟีเจอร์ใหม่ เช่น ปุ่ม Reset Score หรือ การตรวจสอบกฎ “ห้ามให้คะแนนเกิน 10” คุณจัดการฟีเจอร์เหล่านี้ในส่วนใดของ MVVM และ เพราะเหตุใดจึงเลือกปรับปรุงคำสั่งในส่วนนั้น ?
4. Reactive Programming คืออะไร และใน Combine Framework บทบาทของ `ObservableObject`, `@Published`, และ `@StateObject` ช่วยทำให้ SwiftUI ทำงานแบบ Reactive ได้อย่างไร ?
5. นักเรียนคิดว่าแนวคิด MVVM และ SoC เกี่ยวข้องกับแนวคิดทางวิศวกรรมซอฟต์แวร์อื่น ๆ เช่น Agile หรือ Clean Architecture อย่างไรบ้าง?

คำศัพท์สำคัญ

สถาปัตยกรรม MVVM (Model–View–ViewModel)
สถาปัตยกรรม MVVM เป็นรูปแบบการออกแบบซอฟต์แวร์ที่แบ่งความรับผิดชอบของแอปออกเป็น 3 ส่วนอย่างชัดเจน ได้แก่:

- Model — จัดเก็บข้อมูลและกฎพื้นฐานของข้อมูล
- View — แสดงผล UI และรับการโต้ตอบจากผู้ใช้
- ViewModel — จัดการ State ตรรกะ และสื่อสารระหว่าง Model กับ View

จุดเด่นของ MVVM คือทำให้โค้ดเป็นระบบ ง่ายต่อการดูแลรักษา สามารถทดสอบได้ง่าย และลดความซับซ้อนของ UI โดยเฉพาะเมื่อใช้ร่วมกับ SwiftUI และ Combine ซึ่งรองรับการอัปเดต UI แบบ Reactive ได้ดี

Separation of Concerns (SoC)
แนวคิดพื้นฐานในการออกแบบสถาปัตยกรรมซอฟต์แวร์ โดยกำหนดให้ แต่ละส่วนของโปรแกรมทำหน้าที่เฉพาะอย่างหนึ่งเท่านั้น และไม่ปะปนกับงานอื่น ๆ เช่น:

- UI จัดการแค่การแสดงผล
- Logic จัดการแค่การประมวลผล
- Model จัดการแค่ข้อมูล

ผลลัพธ์ของ SoC คือ โค้ดอ่านง่าย แก้ไขเฉพาะจุดได้ง่าย ลดผลกระทบต่อส่วนอื่น ๆ และช่วยให้ระบบสามารถขยายตัวได้โดยไม่ต้องแก้ทั้งโปรเจกต์

Massive View

คำที่ใช้เรียก View ที่ทำงานมากเกินไป ใน SwiftUI ซึ่งจะทำให้มีโค้ดที่ยาว ซับซ้อน แก้ยาก และบำรุงรักษายากขึ้นอย่างมาก Massive View เป็น “สัญญาณเตือน” ว่าโปรแกรมขาดการแบ่งหน้าที่ตามหลัก SoC และควรถูกปรับโครงสร้าง (Refactor) ไปใช้ MVVM

Reactive Programming

แนวคิดการเขียนโปรแกรมที่ UI ตอบสนองต่อการเปลี่ยนแปลงของข้อมูลโดยอัตโนมัติ โดยไม่ต้องเขียนคำสั่งสั่งให้ UI อัปเดตด้วยตนเอง เช่น:

- เมื่อ State เปลี่ยน → UI เปลี่ยนเอง
- เมื่อข้อมูลกลายเป็น Publisher → Subscribers (เช่น View) จะปรับตัวเองให้ตรงกับข้อมูลใหม่

SwiftUI นำแนวคิดนี้มาใช้ร่วมกับ Combine ผ่านกลไก ObservableObject, @Published, และ @StateObject ทำให้การอัปเดต UI เป็นเรื่องอัตโนมัติและมีประสิทธิภาพสูง

Refactor Code

การปรับโครงสร้างโค้ดเดิมให้ “ดีขึ้น” โดย:

- แยกหน้าที่ส่วนต่าง ๆ ออก
- เพิ่มความเป็นระเบียบ
- ลดความซ้ำซ้อน
- เพิ่มความเข้าใจง่าย
- ทำให้ง่ายต่อการบำรุงรักษา

Refactor เป็นกระบวนการสำคัญในงานวิศวกรรมซอฟต์แวร์ และเป็นทักษะที่นักพัฒนาต้องใช้บ่อยมากเมื่อต้องดูแลโค้ดระยะยาว

รายละเอียดสำหรับการอ้างอิงเอกสารชุดนี้

- **ผู้เขียน** ธิติ ธีระเธียร
- **วันที่เผยแพร่** วันที่ 1 ธันวาคม 2568
- **เข้าถึงได้จาก** https://ajthiti.gitbook.io/develop-in-swift/getting-started/mvvm-in-swiftui