Seminář 5: Architektura aplikace

Tvorba mobilních aplikací

Roman Vyjídáček

Obsah semináře

- Architektura aplikace

Architektura aplikace

Softwarová architektura

- Softwarová architektura je struktura systému, která definuje způsob organizace kódu, komunikaci mezi částmi aplikace a rozdělení odpovědností

Proč je architektura klíčová?

- Udržovatelnost – Snadnější oprava chyb a přidávání funkcí.
- Testovatelnost – Možnost snadného testování jednotlivých částí aplikace.
- Škálovatelnost – Možnost rozšiřování aplikace bez velkých zásahů do kódu.
- Znovupoužitelnost – Kód lze snadno využít v jiných částech projektu.
- Minimalizace závislostí – Snížení propojení mezi částmi kódu, což usnadňuje vývoj.

Architektonické vzory v iOS

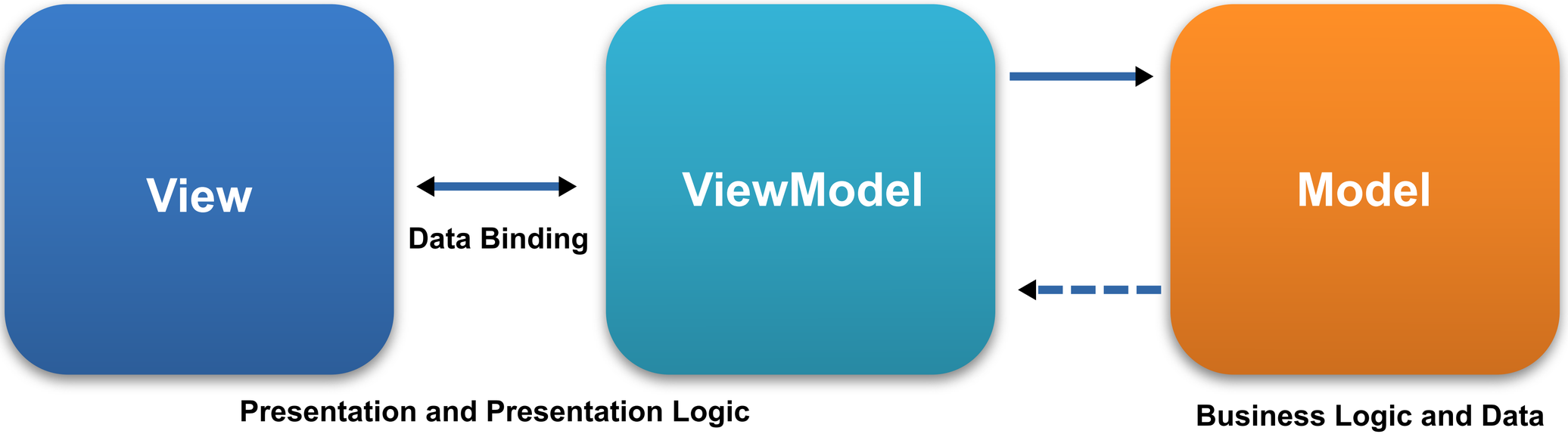
V iOS vývoji existuje několik oblíbených architektonických přístupů:

Architektura	Popis
MVC (Model-View-Controller)	Tradiční Apple přístup, ale často vede k Massive View Controller
MVVM (Model-View-ViewModel)	Odstraňuje problémy MVC – odděluje UI od logiky pomocí ViewModel
VIPER	Striktně odděluje vrstvy, ideální pro velké aplikace, ale složitější na implementaci
Clean Architecture	Rozděluje kód na Presentation, Domain, Data, což minimalizuje závislosti

V moderním vývoji iOS aplikací se nejčastěji používá MVVM + Clean Architecture

Architektury iOS aplikace (MVVM + Clean Architecture)

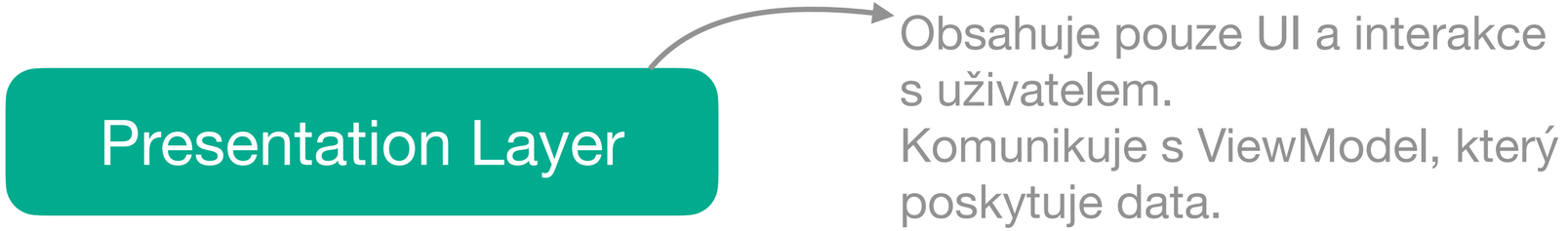
Tento přístup odděluje aplikační vrstvy, čímž zajišťuje modularitu a lepší správu kódu.



https://en.wikipedia.org/wiki/Model–view–viewmodel

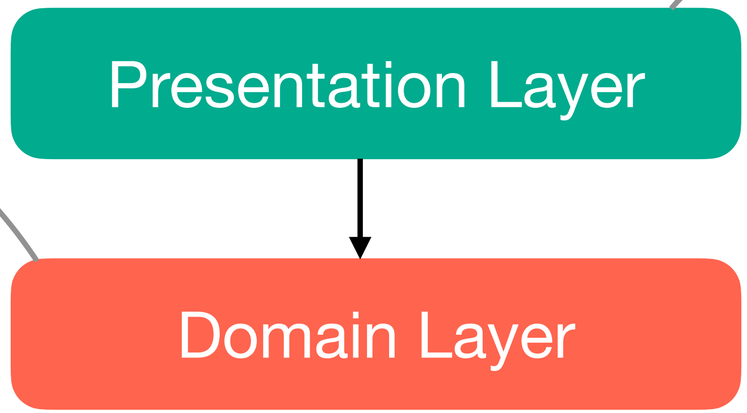
Architektury iOS aplikace (MVVM + Clean Architecture)

Architektury iOS aplikace (MVVM + Clean Architecture)



Architektury iOS aplikace (MVVM + Clean Architecture)

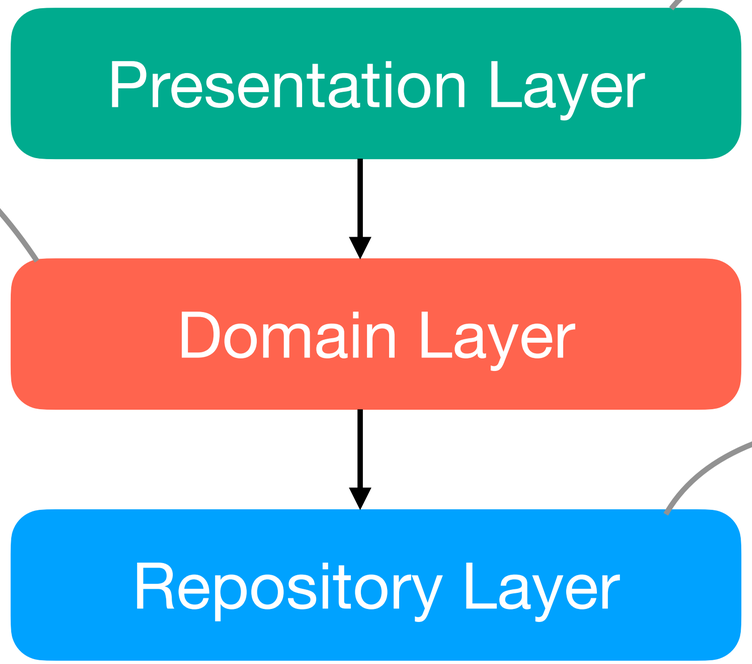
Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.



Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Architektury iOS aplikace (MVVM + Clean Architecture)

Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.

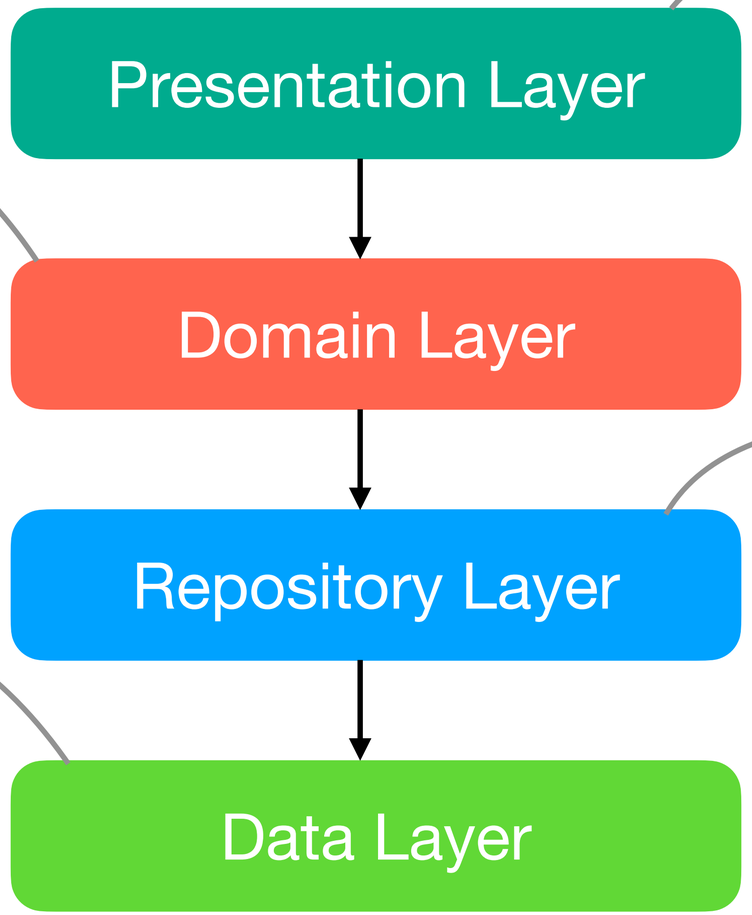


Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Poskytuje Domain vrstvě abstrakci nad daty.
Může získávat data z různých zdrojů (API, databáze).

Architektury iOS aplikace (MVVM + Clean Architecture)

Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.



Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Poskytuje Domain vrstvě abstrakci nad daty.
Může získávat data z různých zdrojů (API, databáze).

Obsahuje API volání, databázi (CoreData, Realm), práci se soubory.
Díky této vrstvě lze snadno změnit zdroj dat bez ovlivnění aplikace

State

- ContentViewState umožňuje sledování změn ve SwiftUI pomocí @Published.
- Vlastnost isLoading označuje, zda probíhá načítání, a při její změně se automaticky aktualizuje navázané view.

```
final class ContentViewState: ObservableObject {

    @Published var isLoading = false
}
```

View

- Podobně jako v architekturách MVC a MVP, „view“ představuje strukturu, rozvržení a vzhled toho, co uživatel vidí na obrazovce.
- Zobrazuje reprezentaci modelu (dat) a reaguje na interakce uživatele (kliknutí, psaní, gesta atd.).
- Interakce nepředává přímo modelu, ale posílá je „view modelu“ přes datové vazby.
- Datová vazba (properties, callbacky) propojuje „view“ a „view model“ a umožňuje přenos dat i událostí.

```
struct ContentView: View {

    @ObservedObject
    private var state: ContentViewState
    private let viewModel: ContentViewModel

    var body: some View {
        if state.isLoading {
            ProgressView()
                .progressViewStyle(.circular)
        } else {
            Button("Load Data") {
                viewModel.loadData()
            }
        }
    }

    static func build() -> ContentView {
        let viewModel = ContentViewModel()
        return ContentView(state: viewModel.state,
                           viewModel: viewModel)
    }
}
```

ViewModel

- ViewModel je prostředník mezi modelem a SwiftUI pohledem – vystavuje veřejné vlastnosti a metody, které může view používat.
- Na rozdíl od MVC nebo MVP neobsahuje ViewModel odkaz na view – ve SwiftUI se view přímo váže na data pomocí @StateObject, @ObservedObject apod.
- SwiftUI používá datové vazby (binding), které automaticky zajišťují přenos dat mezi ViewModelem a View.
- ViewModel často reprezentuje stav dat a může být chápán jako datový objekt (Data Transfer Object).

```
final class ContentViewModel {

    let state = ContentViewState()

    func loadData() {
        state.isLoading = true
    }
}
```

Doménový model

- Reprezentace klíčových entit a pravidel specifických pro oblast (doménu), kterou aplikace řeší.
- Oddělení obchodní logiky od uživatelského rozhraní a datové vrstvy.

Klíčové vlastnosti Doménového modelu

- Nezávislost: Nemá závislosti na UI (SwiftUI) ani na konkrétní databázové implementaci.
- Čistota: Zaměřuje se na obchodní logiku, validaci a pravidla domény.
- Testovatelnost: Snadno testovatelný díky absenci závislostí na dalších vrstvách aplikace.

Integrace Doménového modelu ve SwiftUI

- ViewModel: Spojuje Doménový model s uživatelským rozhraním, transformuje data pro zobrazení.
- SwiftUI View: Pozoruje ViewModel a reaguje na změny dat pomocí datových vazeb.

Usecase

- Zaměřuje se na jednu konkrétní akci nebo cíl uživatele
- Kombinací menších Use Case lze vytvářet složitější funkce systému
- V naší architektuře je Usecase umístěn mezi ViewModel a Repository
- Má pouze jednu veřejnou metodu use(…)

```
final class GetCurrentLocationUsecase {

    func use() -> CLLocation {
        // Zde ziskame aktualni pozici
        // napriklad z GPSManager
    }
}

class ViewModel {

    private let getCurrentLocationUsecase:
GetCurrentLocationUsecase

    func loadData() {
        let location =
getCurrentLocationUsecase.use()
    }
}
```

Repository

- Repository Pattern poskytuje abstraktní vrstvu mezi obchodní logikou aplikace a datovými zdroji, jako jsou databáze či API
- Oddělení obav (Separation of Concerns): Umožňuje, aby doménová vrstva nebyla závislá na konkrétní implementaci datového zdroje
- Flexibilita: Usnadňuje změnu nebo kombinaci různých datových zdrojů bez nutnosti úprav v obchodní logice
- Testovatelnost: Umožňuje snadné vytváření mock implementací pro jednotkové testování.

```
protocol UserRepository {

    func fetchUsers() async throws -> [User]
    func addUser(_ user: User) async throws
}

class UserRepositoryImpl: UserRepository {

    func fetchUsers() async throws -> [User] {
        // Načtení uživatelů z UserDefaults
    }
    func addUser(_ user: User) async throws {
        // Uložení uživatele do UserDefaults
    }
}

class UserRepositoryMock: UserRepository {
    func fetchUsers() async throws -> [User] {
        return User.mockArray
    }

    func addUser(_ user: User) async throws {
        // zde potřebujeme Mock
        // implementaci DBManager
    }
}
```

Manager

- Třída, která se stará o konkrétní funkčnost, např. síťovou komunikaci, ukládání do souborů, notifikace, GPS apod.
- Zapouzdřuje logiku, která nesouvisí přímo s UI ani doménovou logikou.

Role Managera

- Oddělení odpovědností:
  - Zajišťuje, že ViewModel není zahlcen technickými detaily (např. voláním URLSession, prací s diskem).
- Opakovatelnost:
  - Lze ho znovu použít napříč různými ViewModely nebo i aplikacemi.
- Snadnější testování:
  - Může být nahrazen mockem při testování ViewModelu.

Úkoly

https://vyjidacek.cz/tmai/seminar5.php