MVVM-C EnterPrise CleanArchitecture

🛠️ Technologies & Tools

Proje geliştirilirken endüstri standardı kütüphaneler ve modern Swift yetenekleri kullanılmıştır.

📱 Platform & Environment

- **Language:** Swift 5.9
- **Minimum Target:** iOS 15.0+
- **IDE:** Xcode 15+
- **UI Framework:** UIKit (Programmatic UI)

🏗️ Architecture & Design Patterns

- **MVVM-C:** Navigation logic'i View'dan ayırmak için **Coordinator** pattern entegreli MVVM.
- **Clean Architecture:** Data, Domain ve Presentation katmanlarının tam izolasyonu.
- **Builder Pattern:** Modüllerin bağımlılıklarının (Dependency Injection) yönetilmesi.
- **Repository Pattern:** Veri kaynağı (Remote/Local) soyutlaması.
- **Strategy Pattern:** Dinamik iş kuralları (Business Rules) yönetimi.
- **Factory Pattern:** Strategy nesnelerinin üretimi.
- **Delegation Pattern:** View ve ViewModel ile Coordinator arası iletişim.
- **Singleton Pattern:** (Sadece `NetworkManager` gibi servislerde kontrollü kullanım).

📦 Third-Party Libraries

- **Alamofire:** Network istekleri için güvenilir ve test edilmiş HTTP networking kütüphanesi.
- **SnapKit:** Programmatic UI yazarken okunabilirliği artıran ve Auto Layout kodunu sadeleştiren DSL.
- **Kingfisher:** Asenkron resim indirme ve önbellekleme (Caching) yönetimi.

⚡ Core Frameworks & Capabilities

- **Swift Concurrency (Async/Await):** Modern, thread-safe ve callback-hell'den uzak asenkron veri akışı.
- **Codable:** JSON veri parse işlemleri.
- **XCTest:** Unit Test senaryoları.
- **SPM (Swift Package Manager):** Bağımlılık yönetimi.

Architectural Layers

Proje, sorumlulukların net bir şekilde ayrıldığı (Separation of Concerns) modüler bir katman yapısı üzerine inşa edilmiştir:

- 📂 **Application Layer**
  Uygulamanın başlatılması ve yaşam döngüsü yönetiminden sorumludur; `SceneDelegate` ve `AppCoordinator` başlangıç noktaları burada yer alır.
- 🧠 **Domain Layer**
  Saf iş mantığının (Business Logic) döndüğü, dış dünyadan izole merkezdir; `UseCase` (Interactor), `Entity`, `Repository Protocol` ve `Strategy` desenleri burada bulunur.
- 💾 **Data Layer**
  Veri akışının yönetildiği ve ham verinin işlendiği katmandır; `Repository` implementasyonları, API tanımları (`Endpoint`) ve `DTO` modelleri burada yaşar.
- ⚙️ **Infrastructure Layer**
  Sistemin motor kısmıdır; ağ isteklerini yöneten Generic `NetworkManager` ve alt yapı araçları burada konumlanır.
- 🎨 **Presentation Layer**
  Kullanıcı arayüzü ve navigasyonun yönetildiği katmandır; MVVM bileşenleri

(`ViewModel`, `View`), bağımlılıkları çözen `Builder` ve akışı sağlayan `Coordinator` buradadır.

- 🛠️ **Common Layer**
  Proje genelinde tekrar kullanılan yardımcı kodlardır; `Extensions` ve `Utilities` (örn. `ViewState`) gibi ortak araçları kapsar.

```
MVVM-C-Strategy-Builder-Repository-CleanArch/
├── 📂 Application
│   ├── AppDelegate.swift
│   └── SceneDelegate.swift
│
├── 📂 Common
│   ├── 📂 Extensions
│   └── 📂 Utils
│
├── 📂 Data                    ← Networkings details
│   └── 📂 Scenes
│       ├── 📂 Detail
│       │   └── DetailEndpoint.swift
│       └── 📂 Home
│           ├── HomeDTO.swift
│           └── HomeEndpoint.swift
│
├── 📂 Domain                  ← Business Logic - Uygulamanın Beyni
│   └── 📂 Scenes
│       ├── 📂 Detail
│       │   ├── DetailRepository.swift
│       │   ├── DetailRepositoryProtocol.swift
│       │   └── DetailUseCase.swift
│       └── 📂 Home
│           ├── CharacterEntity.swift
│           ├── CharacterStatusStrategy.swift
│           ├── CharacterStrategyFactory.swift
│           ├── HomeRepository.swift
│           ├── HomeRepositoryProtocol.swift
```

```
│           └── HomeUseCase.swift
│
├── 📂 Infrastructure          ← "Altyapı" Katmanı (Dış Dünya ile İletişim)
│   └── 📂 Network
│       ├── Endpoint.swift
│       ├── NetworkError.swift
│       ├── NetworkLogger.swift
│       ├── NetworkManager.swift
│       └── NetworkTypes.swift
│
├── 📂 Presentation            ← "Sunum" Katmanı (UI & Navigasyon)
│   ├── 📂 Coordinators
│   │   ├── AppCoordinator.swift
│   │   └── Coordinator.swift
│   └── 📂 Scenes
│       ├── 📂 Detail
│       │   ├── DetailBuilder.swift
│       │   ├── DetailViewController.swift
│       │   └── DetailViewModel.swift
│       └── 📂 Home
│           ├── HomeBuilder.swift
│           ├── HomeViewController.swift
│           └── HomeViewModel.swift
│
└── 📂 Resources               ← Kaynak Dosyaları
    ├── Assets.xcassets
    ├── Info.plist
    └── LaunchScreen.storyboard
```

Engineering Principles & Disciplines

Projenin mimarisi, sürdürülebilirlik (**Maintainability**) ve ölçeklenebilirlik (**Scalability**) için aşağıdaki temel yazılım prensipleri üzerine inşa edilmiştir:

💎 SOLID Prensipleri

- **Single Responsibility (SRP):** Her sınıfın (`Builder`, `UseCase`, `ViewModel`) değişmek için tek bir nedeni vardır. UI mantığı ile İş mantığı kesin çizgilerle ayrılmıştır.
- **Open/Closed (OCP):** Proje gelişime açık, değişime kapalıdır. Örneğin; `CharacterStatusStrategy` sayesinde, mevcut kod değiştirilmeden yeni durumlar (Status) eklenebilir.
- **Dependency Inversion (DIP):** Üst seviye modüller (`Domain`), alt seviye modüllere (`Data`) bağımlı değildir. Her ikisi de soyutlamalara (`Protocols`) bağımlıdır.

🏗️ Architectural Concepts & Patterns

- **Clean Architecture:** Bağımlılık kuralına (**Dependency Rule**) sadık kalınmıştır. Dış katmanlar (UI, DB) iç katmanları (Domain) bilir, ancak iç katmanlar dış dünyadan tamamen habersizdir.
- **Protocol-Oriented Programming (POP):** Somut sınıflar yerine soyut protokoller üzerinden iletişim kurulur. Bu sayede modüller arası sıkı bağ (**Tight Coupling**) engellenmiş ve **Loose Coupling** sağlanmıştır.
- **Dependency Injection (DI):** Bağımlılıklar sınıfların içinde oluşturulmaz, dışarıdan enjekte edilir. Bu proje özelinde 3. parti kütüphane yerine, **Builder Pattern** kullanılarak "Pure Dependency Injection" uygulanmıştır.
- **Separation of Concerns (SoC):** Ağ istekleri (`Infrastructure`), iş kuralları (`Domain`) ve ekran çizimi (`Presentation`) farklı katmanlarda izole edilmiştir.
- **Testability (Test Edilebilirlik):** Tüm dış bağımlılıklar (Network, Repository) protokoller arkasına gizlendiği için, **Unit Test** senaryolarında kolayca **Mock** objelerle değiştirilebilir yapıdadır.

Application Execution Flow (Adım Adım Çalışma Mantığı)

Uygulamanın `SceneDelegate` ile ayağa kalkmasından, kullanıcının bir karaktere tıklayıp detay sayfasına gitmesine kadar geçen sürecin teknik akışı:

1. 🚀 Initialization (Başlatma)

- **SceneDelegate:** Uygulama açılır. `UIWindow` oluşturulur.
- **AppCoordinator:** `SceneDelegate`, ana navigasyonu yönetmesi için `AppCoordinator`'ı başlatır (`start()`).
- **Routing:** `AppCoordinator`, ilk gösterilecek ekranın `Home` olduğuna karar verir.

2. 🏗️ Module Assembly (Builder & DI)

- **HomeBuilder:** Coordinator, `HomeBuilder.make()` metodunu çağırır.
- **Network:** `NetworkManager` (Generic Engine) oluşturulur.
- **Data:** `HomeRepository` oluşturulur (`NetworkServiceProtocol` enjekte edilir).
- **Domain:** `HomeUseCase` oluşturulur (`HomeRepositoryProtocol` enjekte edilir).
- **Presentation:** `HomeViewModel` oluşturulur (`HomeUseCaseProtocol` enjekte edilir).
- **View:** `HomeViewController` oluşturulur ve `viewModel` içine set edilir.
- **Display:** Hazırlanan `HomeViewController`, Coordinator tarafından ekrana push edilir.

3. 📡 Data Flow & Business Logic (Veri Akışı)

- **View Cycle:** `HomeVC` açılır (`viewDidLoad`) ve ViewModel'e "Verileri getir" emrini verir.
- **UseCase Execution:** `HomeViewModel` → `HomeUseCase.execute()` metodunu çağırır.
- **Repository Fetch:** `UseCase` → `HomeRepository.fetchCharacters()` metodunu çağırır.
- **Network Request:** Repository, `NetworkManager` üzerinden `HomeEndpoint.getCharacters` isteğini atar.
- **Mapping (DTO → Entity):**
  - API'den gelen ham JSON, `HomeDTO` (`Decodable`) modeline dönüştürülür.
  - Repository, bu DTO'yu `toDomain()` metoduyla saf `CharacterEntity`'ye çevirir ve UseCase'e döner.

4. 🧠 Strategy Pattern Implementation

- **Business Rule:** `HomeUseCase`, gelen veriyi ham haliyle ViewModel'e vermez.

- **Strategy Logic:** Karakterin statüsüne (Alive/Dead) bakar ve `CharacterStrategyFactory`'yi çağırır.
- **Formatting:** İlgili stratejiyi (`AliveStrategy` vb.) seçer ve ismin yanına ikonu (✅ veya 💀) işler.
- **State Update:** İşlenmiş veri `HomeViewModel`'e gelir. ViewModel, `ViewState.success` durumunu View'a bildirir.
- **UI Update:** `HomeVC`, değişikliği algılar ve TableView'ı yeniler (`reloadData`).

5. 👉 Navigation (Coordinator Pattern)

- **User Action:** Kullanıcı bir karaktere tıklar (`didSelectRow`).
- **Delegation:** `HomeVC`, bu tıklamayı kendisi yönetmez; `HomeViewControllerDelegate` üzerinden (ki bu `AppCoordinator`'dır) haber verir.
- **Detail Module:** `AppCoordinator`, tıklanan karakterin ID'sini alır ve `DetailBuilder.make(id: ...)` çağırır.
- **Transition:** Tüm bağımlılıkları (Repo, UseCase, VM) hazırlanmış `DetailViewController` ekrana sürülür.

📝 Özet Diyagram (Flowchart)

```
[SceneDelegate]
    ⬇
[AppCoordinator]
    ⬇
[HomeBuilder] --(Injects)--> [Network] -> [Repo] -> [UseCase] -> [ViewModel]
    ⬇
[HomeViewController] --(User Taps)--> [AppCoordinator]
                                          ⬇
                                    [DetailBuilder]
                                          ⬇
                                    [DetailViewController]
```