ChatGPT

VIPER Mimarisi ile Profesyonel iOS Uygulama Geliştirme: Sözlük Uygulaması Örneği

VIPER Mimarisinin Temel Prensipleri

VIPER, **View-Interactor-Presenter-Entity-Router** kelimelerinin kısaltması olan bir mimari desendir. Aslında VIPER, "Temiz Mimari" (Clean Architecture) prensiplerinin iOS uygulamalarına uyarlanmış halidir [1]. Bu mimaride her katman, tek bir sorumluluğa sahip olacak şekilde tasarlanır (Single Responsibility Principle) [2] [3]. VIPER'ın beş ana bileşeni ve bunların rollerini şöyle özetleyebiliriz:

- **View (Görünüm):** Kullanıcı arayüzüdür ve **pasif** olmalıdır. Görevi, Presenter'dan aldığı verileri ekranda göstermek ve kullanıcı etkileşimlerini Presenter'a iletmektir [4] [5]. Örneğin bir `UIViewController`, Presenter'ın ilettiği metinleri uygun `UILabel` ve `UIButton` gibi bileşenlerde görüntüler; buton tıklamalarını veya metin girişlerini Presenter'a haber verir.

- **Interactor (İşleyici):** Uygulamanın **iş mantığını** barındıran katmandır. Belirli bir **use case**'i (kullanım senaryosunu) gerçekleştirmek için gerekli veri işlemlerini yapar [6]. Interactor, veri modeliyle çalışır, ağ istekleri yapar veya veritabanından okuma/yazma işlemlerini yürütür [7]. **Kullanıcı arayüzünden tamamen bağımsızdır**; bu sayede aynı interactor kodu iOS veya başka bir platformda da kullanılabilir [8].

- **Presenter (Sunucu):** Arayüz ile iş mantığı arasında köprü görevi görür. **View logic** de denir; ekranda gösterilecek verileri formatlar, kullanıcı etkileşimlerine tepki verir ve gerekli olduğunda Interactor'dan yeni veri talep eder [9]. Presenter, View'den gelen olaylara yanıt olarak Interactor'ı çağırır, Interactor'dan gelen ham verileri alır ve **ViewModel** veya gösterime uygun formata çevirip View'a iletir [10]. Ayrıca, ekranlar arası geçiş gerektiğinde Router'ı tetikleyen de Presenter'dır [11] [12].

- **Entity (Varlık):** Uygulamanın temel **model** nesneleridir. Interactor tarafından işlenen saf veri yapılarıdır [13] [14]. Örneğin sözlük uygulamasında "Kelime" nesnesi, kelimenin metni, tanımı, dil kodu gibi alanları içeren bir Entity olabilir. Entity'ler **sadece Interactor içinde kullanılır**; Presenter veya View'a direkt Entity göndermekten kaçınılır [15] [16]. Bu sayede, UI katmanında modelin işlenmesi engellenir ve katmanlar arası bağımlılık azalır.

- **Router (Yönlendirici):** Uygulamadaki ekranlar arası **navigasyon** mantığını içerir [17]. Bir VIPER modülünün Router'ı, o modülün View'ını oluşturmaktan ve gerekli olduğunda yeni ekranlara geçiş yapmaktan sorumludur. Örneğin bir Router, mevcut görünümden "Kelime Detay" modülüne geçiş yaparken yeni modülü oluşturup ilgili `UIViewController`'ı `UINavigationController` aracılığıyla ekrana sürebilir [18]. Router ayrıca ekranlar arası veri transferini de üstlenebilir (örneğin arama modülünden detay modülüne seçilen kelimenin bilgisini taşımak).

Bu rollerin her biri kendi dosya/sınıfında yer alır ve birbirleriyle **protokoller aracılığıyla** iletişim kurarlar. VIPER'da her modül (ekran veya özelliğe karşılık gelen bölüm) için bu beş bileşenin birer implementasyonu vardır. Bu mimari yaklaşım, uygulamanın mantığını küçük ve iyi tanımlanmış

katmanlara bölerek **bağımlılıkları izole etmeyi** amaçlar [1]. Sonuç olarak, her parçanın sorumluluğu net bir şekilde ayrılır ve sistem şu şekilde işler:

- View kullanıcıdan gelen etkileşimleri Presenter'a iletir.
- Presenter bu olaya uygun iş mantığını Interactor'a sorar veya Router ile navigasyonu başlatır.
- Interactor gerekli veriyi Entity'ler üzerinde işler veya servis katmanından (ağ, veritabanı vb.) alır ve sonucunu Presenter'a döner.
- Presenter bu sonucu gerekiyorsa düzenler/formatlar ve View'a sunulmak üzere iletir.
- Router, Presenter'ın talebiyle gerekli yeni modülü oluşturur ve View'ını gösterir.

Bu şekilde her katman, sadece kendi göreviyle ilgilenir ve diğer katmanlarla arayüz (protokoller) üzerinden iletişim kurar [3]. Böylece **gevşek bağlı** (loosely coupled) bir yapı elde edilir: Örneğin Interactor, veriyi nasıl gösterileceğini bilmez; View da verinin nasıl elde edildiğini bilmez. Bu prensipler, VIPER'ı karmaşık uygulamalarda dahi düzenli ve yönetilebilir bir mimari haline getirir.

VIPER Mimarisi'nin Avantajları

Profesyonel iOS projelerinde VIPER mimarisinin popüler hale gelmesinin sebepleri arasında, sağladığı güçlü yapısal avantajlar vardır:

- **Tek Sorumluluk ve Temiz Kod:** VIPER, katı bir sorumluluk dağılımı uygular. Her bileşen tek bir iş yapar, bu da kodu daha temiz ve anlaşılır kılar. Örneğin "Massive View Controller" (şişkin UIViewController) problemini ortadan kaldırır [19]. Kodunuz SOLID prensiplerine uygun hale gelir ve her yeni özelliği doğru yere ekleyerek mimari tutarlılığı korursunuz [20].

- **Modülerlik ve Ölçeklenebilirlik:** VIPER, uygulamayı modüllere ayırır; her ekran veya özellik kendi VIPER modülüne sahip olur [21]. Bu modüller birbirinden bağımsız geliştirilebilir ve gerektiğinde projeden çıkarılabilir. Bu, özellikle **büyük ekipler** için ölçeklenebilirlik sağlar: Birden fazla geliştirici farklı modüllerde rahatça çalışabilir, çakışmalar azalır [22]. Kod tabanı yapısal olarak tutarlı olduğundan, yeni katılan geliştiriciler de projeyi daha hızlı kavrayabilir ve başkalarının kodunu okumak kolaylaşır [23].

- **Test Edilebilirlik:** VIPER'ın en büyük faydalarından biri test yazmayı kolaylaştırmasıdır. Katmanlar arası gevşek bağlı yapı sayesinde birimi ilgilendiren kod izole edilebilir. Örneğin Interactor tamamen arayüzden bağımsız olduğu için **ünite testleri** rahatça yazılabilir [24]. Presenter ve Interactor genellikle sade `Swift` sınıflarıdır, bu da %80-90'a varan test kapsamına ulaşmayı mümkün kılar; oysa devasa bir ViewController'ın test edilmesi neredeyse imkânsızdır [24]. Ayrıca iş mantığı UI'dan ayrıldığı için, arayüz değişse dahi iş kurallarına dair testler etkilenmez.

- **Bakım ve Sürdürülebilirlik:** Kodu modüler ve sorumlulukları ayrılmış yazmak, uzun vadede bakım kolaylığı sağlar. Hataları izlemek ve çözmek daha kolay olur; çünkü her hata, genellikle belirli bir modüldeki belirli bir katmanda izole olur. Tek sorumluluk ilkesi sayesinde bir kaza raporundaki hatanın kaynağını ilgili modülde hızla bulabilirsiniz [25]. Yeni bir özellik eklemek veya değişiklik yapmak da VIPER ile daha az risklidir; zira mevcut kodun diğer bölümlerine dokunmadan, yeni bir modül ekleyebilir veya var olan modüle yeni davranışlar katabilirsiniz [26] [27]. Bu da **teknik borcun** daha yönetilebilir olmasını sağlar.

- **Paralel Geliştirme:** Bağımsız modüller sayesinde birden fazla ekip üyesi aynı anda farklı özellikler üzerinde çalışabilir. Örneğin biri arama özelliği üzerinde çalışırken diğeri favoriler modülünü geliştirebilir. Modüller arası bağımlılık asgari düzeyde olduğu için entegrasyon

sırasında minimum çakışma yaşanır. Bu da versiyon kontrolü kullanılırken **daha az merge konflikti** anlamına gelir [28] [22].

- **Yüksek Kalite ve Tutarlılık:** VIPER kod tabanının geneli için bir standart yapısı vardır. Her modül benzer şekilde düzenlendiğinden, uygulamanın her bölümünde tanıdık bir yapı görürsünüz. Dosyalar daha küçüktür (ör. binlerce satırlık tek bir sınıf yerine, her biri birkaç yüz satırlık beş ayrı sınıf) ve bu da anlamayı kolaylaştırır [23]. Kodun tutarlı olması, **kod incelemelerini** de kolaylaştırır; hatta sadece iOS ekibi içinde değil, Android ekibiyle bile iş mantığı kodlarını karşılıklı gözden geçirmek mümkün olabilir, çünkü benzer soyutlama seviyeleri kullanılmıştır [29].

- **SOLID Prensiplerine Uygunluk:** VIPER, adeta SOLID prensiplerinin pratik bir uygulamasıdır. Bağımlılıkların tek yönlü akması (Dependency Inversion) ve arayüz ayrımı (Interface Segregation) gibi prensipler, VIPER modüllerinde doğal olarak sağlanır. Bu da kod kalitesini artırır. VIPER aynı zamanda **TDD dostu** bir yapıdır; tasarımından ötürü test-yönelimli geliştirme kültürünü benimsemeyi kolaylaştırır [30].

Özetle, VIPER mimarisi özellikle **orta ve büyük ölçekli**, uzun ömürlü projelerde yapıyı korumak, karmaşıklığı yönetmek ve yüksek kaliteli kod sağlamak için önemli avantajlar sunar [31]. Doğru uygulandığında, VIPER ile geliştirilen bir iOS uygulaması daha öngörülebilir davranır, değişikliklere karşı daha dirençli olur ve ekip çapında anlaşılması daha kolay bir kod tabanına sahip olur.

VIPER Mimarisiyle Karşılaşılabilecek Zorluklar

Her ne kadar VIPER pek çok avantaj sunsa da, uygulama ve öğrenme aşamasında bazı zorluklar ve dikkat edilmesi gereken noktalar vardır:

- **Öğrenme Eğrisi ve Ekip Uyumu:** VIPER, MVC veya MVVM'ye kıyasla daha fazla bileşen içerdiği için başlangıçta kavraması zor gelebilir. Bu mimariye yeni bir geliştirici, temel kavramları öğrenmek için birkaç güne, tam anlamıyla üretken olabilmek için belki birkaç haftaya ihtiyaç duyabilir [32]. Takım içinde VIPER kullanımına dair **standartlar ve en iyi uygulamalar** belirlenmemişse, her geliştirici farklı yorumlayabilir. Bu nedenle ekip içinde ortak bir anlayış oluşturmak için zaman ve eğitim yatırımı gerekebilir [32]. Ayrıca VIPER iOS dünyasında MVC kadar yaygın olmadığından, topluluk ve dokümantasyon sınırlı olabilir; kaynak bulmak eskiye nazaran daha kolay olsa da halen MVC/MVVM kadar popüler değildir [33].

- **Artan Dosya ve Kod Miktarı:** VIPER'ın belki de en çok eleştirilen yönü, basit bir ekran için bile çok sayıda dosya ve protokol oluşturma gerekliliğidir. Örneğin, ufak bir özelliği bile VIPER ile yazdığınızda bir View, Presenter, Interactor, Entity ve Router sınıfı, ayrıca bunların iletişim protokolleri ortaya çıkar. Bu, **küçük projeler için aşırı** gelebilir [34]. Eğer projeniz çok basitse veya prototip aşamasındaysa, VIPER gereksiz karmaşıklık katabilir. Hatta her yeni ekran için 5-6 yeni dosya oluşturmak zaman alıcıdır; bu yüzden VIPER kullananlar genellikle **kod üretici araçlar** kullanmayı önerir (ör. Xcode eklentileri veya Generamba gibi araçlar) [35]. Kod üretimi otomatikleştirilmezse, elle bu şablonları oluşturmak başlangıçta verimliliği düşürebilir.

- **Apple Mimarisiyle Uyuşmazlık Noktaları:** VIPER, UIKit'in standart yaklaşımından farklı bir disiplin getirir. UIKit'de `UIViewController` genellikle hem UI yönetimi hem de bir miktar iş mantığını içerir, ancak VIPER bu sorumlulukları böler. Bu geçişte bazı konular ortaya çıkar: Örneğin **UITableView datasource/delegate** yöntemlerini nereye koyacağınız tartışmalı olabilir. En pratik yaklaşım, bu tür UI kontrolüyle ilgili sorumlulukların hala View/ViewController'da kalmasıdır (VIPER bile kullanılsa, ViewController tablonun dataSource'u olabilir) [36]. Yani *"UIKit'i*

aşırı zorlamayın" prensibi geçerlidir: Interactor'a UIKit'e özgü işler vermekten kaçının, aksi takdirde mimariyi zorlamış olursunuz [36]. Bazı üçüncü parti kütüphaneler de VIPER'ın katı ayrımlarına uymayabilir; bu durumda ekibin uygun entegrasyon yolu bulması gerekebilir [37].

- **Aşırı Mühendislik ve Karmaşıklık:** Küçük veya orta ölçekli basit uygulamalarda VIPER kullanmak, gereğinden fazla soyutlama yaratabilir. Bazen Presenter sadece View'dan gelen çağrıyı Interactor'a iletip sonucunu geri View'a iletiyor ve fazladan bir katman gibi görünebiliyor [38] [39]. Bu durum, hatalı kullanımda **"boş sunucu"** (redundant presenter) gibi kokulara yol açabilir. Benzer şekilde, her sınıf arası bir protokol tanımlamak teoride esneklik sağlasa da, pratikte çoğu modülde tek bir Presenter-View veya Presenter-Interactor çifti olduğu için bu protokoller bazen hiç farklı implementasyon kullanılmadan kalır [40]. Bu da kod içinde **gereksiz soyutlama** hissi verebilir ve kod takibini IDE üzerinde zorlaştırabilir (çünkü Cmd+Click ile bir metodun tanımına giderken protokole gider, gerçek implementasyonu bulmak için ekstra adım gerekir) [41].

- **Performans ve Başlangıç Gecikmesi:** Çok sayıda sınıf ve katman olması, **uygulamanın başlangıç süresine** ufak da olsa etki yapabilir [34]. Örneğin uygulama açılırken her modülün Router'ının başlangıçta kurulum yapması gerekiyorsa (genelde lazy init ile yapılır, ama bazı yapılar ilk başta kurabilir), bu bir miktar overhead oluşturabilir. Ancak bu etki genellikle kritik değildir ve dikkatli yönetilerek minimize edilebilir.

- **Modüller Arası İletişim Zorlukları:** VIPER, her modülü bağımsız tutmayı hedefler. Fakat modüllerin birbiriyle konuşması gerektiğinde en iyi yaklaşımın ne olduğu tartışmalıdır. Klasik VIPER yaklaşımında bir modül başka bir modülü çağırırken, genellikle Router üzerinden yeni modül yaratılıp veriler aktarılır; geri dönüş için ise delegasyon veya callback kullanılır. Bazı kaynaklar modüllerin Presenter'ları arasında doğrudan iletişimi önermiş ancak bu SRP'ye aykırı olabilir [42]. En yaygın ve temiz yöntem, **Router üzerinden veri geçmek** ve gerektiğinde bir sonraki modülün bir delegasyon arayüzünü önceki modüle vermektir (örneğin B modülü kapatılırken A modülüne sonuç döndürmek için) [43]. Bu konfigürasyonu düzgün tasarlamazsanız, modüller arası veri akışı karmaşıklaşabilir. Bu nedenle, modüllerin olabildiğince bağımsız kalmasına dikkat edilmeli, ortak bir **servis katmanı** veya merkezi bir olay aracısı (event bus) gibi çözümler düşünülmelidir.

- **Küçük Değişikliklerin Çok Yere Dokunması:** Basit bir etkileşim bile VIPER'da birden fazla dosyayı etkileyebilir. Örneğin kullanıcı arayüzünde bir düğmeye basıldığında, bu olayı View'dan Presenter'a, oradan Interactor'a taşıyıp, geri Presenter ve View'a dönmek gerekebilir. Bu akış net olsa da, geliştirme sırasında bir değişiklik yaparken ilgili tüm protokol ve metot imzalarını güncellemek gerekebilir. Bu da başlangıçta daha zahmetli görünür. Ancak iyi tarafı, değişikliğin etkisi tek bir modülle sınırlı kalır; diğer bölümlere zincirleme yan etki pek olmaz.

Yukarıdaki zorluklar, VIPER'ın her proje için mükemmel olmadığına işaret eder. Eğer uygulamanız küçük, ekibiniz sınırlı ve hızlı prototipleme ön plandaysa, MVVM gibi daha hafif bir mimari daha uygun olabilir [44] [31]. VIPER en çok **uzun vadeli, kapsamlı ve karmaşık projelerde** parlıyor; çünkü bu gibi projelerde ilk baştaki karmaşıklık maliyeti, zaman içinde daha düzenli bir kod tabanıyla fazlasıyla geri ödenir. Nitekim bazı uzmanlar, VIPER'ın *"ilk başta yavaşlatsa da uzun vadede hız kazandırdığı"* görüşünde [45]. Özetle, VIPER'ı tercih ederken projenizin ihtiyaçlarını, takımınızın tecrübesini ve mimarinin getirilerini dikkatle değerlendirmelisiniz.

Modülerlik, Test Edilebilirlik ve Sürdürülebilirlik Açısından VIPER

Modülerlik: VIPER, uygulamanızı modüller halinde organize eder. Her bir **ekran veya özelliği**, kendi VIPER katmanlarına sahip bağımsız bir modül olarak düşünebilirsiniz [21]. Örneğin bir sözlük uygulamasında "Arama", "Arama Sonuçları", "Kelime Detayı" ve "Favoriler" ayrı modüller olabilir. Bu modüler yaklaşım, uygulamayı parçalara bölerek geliştirmeyi kolaylaştırır. Bir modülde yapılan değişiklik, diğer modülleri etkilemez (bağımlılıklar yalnızca belirlenmiş arayüzler üzerinden aktığı için) [46]. Bu, **bakım** sırasında da avantajlıdır: Bir hatanın hangi modülde olduğunu saptamak ve o modülün içinde düzeltmek mümkündür. Ayrıca modüler yapı sayesinde uygulamanın belirli bir parçasını tamamen yeniden yazmak gerekirse, diğer kısımlara dokunmadan bu yapılabilir.

Test Edilebilirlik: VIPER mimarisi, test edilebilirliği en üst düzeye çıkarmak için tasarlanmıştır. İş mantığı (Interactor) ve prezentasyon mantığı (Presenter) arayüzden ayrıldığı için, bu bileşenler üzerinde arayüz bağımsız birim testleri yazılabilir. Örneğin bir Interactor metodunu test etmek için bir sahte (mock) veri yöneticisi vererek, ağ ya da veritabanı çağrılarını kontrol edebilirsiniz. Presenter'ı test etmek için sahte bir View ve sahte bir Interactor kullanarak, belirli kullanıcı eylemlerinde Presenter'ın doğru tepkiyi verip vermediğini doğrulayabilirsiniz. VIPER'da her sınıf küçük ve odaklı olduğu için, test senaryoları da o oranda basit olur. Gerçek dünyada %80 üzeri test kapsama oranlarına ulaşan VIPER tabanlı projeler mevcuttur [47]. Özellikle **TDD** (Test Driven Development) yapıyorsanız, VIPER'ın belirgin arayüz ayrımları bu süreci kolaylaştırır: Önce arayüz protokollerini ve beklentileri tanımlayıp sonra implementasyonu yazabilirsiniz. Böylece VIPER, karmaşık uygulamalarda dahi regresyon riskini azaltarak güvenli gelişim ortamı sunar [24].

Sürdürülebilirlik (Bakım Kolaylığı): Kodun uzun vadede sürdürülebilir olması, değişen gereksinimlere uyum sağlayabilmesiyle ölçülür. VIPER, katmanlar arası gevşek bağ ve tek yönlü bağımlılık sayesinde **değişime dirençli** bir yapı kurar. Örneğin uygulamaya yeni bir özellik eklerken, var olan modüllere minimal dokunuş yaparsınız veya yeni bir modül eklersiniz. Bu da mevcut kodun stabilitesini korur. Kodun sorumluluklara ayrılmış olması, bir geliştiricinin belirli bir davranışı değiştirmek istediğinde, tam olarak ilgili sınıfa odaklanmasını sağlar. Örneğin sözlük uygulamasında arama algoritmasını değiştirmek istiyorsanız, doğrudan AramaInteractor içine bakmanız yeterli olur. Diğer katmanlar (View veya Presenter) bu değişimden etkilenmez. Bu izolasyon, **refactoring** (yeniden düzenleme) işlemlerini de güvenli kılar; bir modül elden geçirilirken diğerleri çalışmaya devam eder. Ayrıca VIPER, loglama ve hata raporlama açısından da netlik sağlar: Bir crash raporunda hatanın hangi modül ve katmanda olduğu daha kolay anlaşılabilir. Tek sorumluluk ilkesi sayesinde, örneğin bir çökme mesajı belirli bir sınıfa işaret ediyorsa, o sınıfın yaptığı iş bellidir ve hata ayıklama süreci hızlanır [25].

Özetle, VIPER ile geliştirilen bir uygulama **modüler** yapısı sayesinde büyümeye ve değişime hazırdır, **test edilebilir** yapısı sayesinde güvenilir ve hatasızdır, **sürdürülebilir** yapısı sayesinde de yıllar içinde bile kolaylıkla evrilip ayakta kalabilir. Bu nitelikler, özellikle uzun ömürlü ve kurumsal uygulamalarda VIPER'ı cazip kılmaktadır [31].

Sözlük Uygulamasına VIPER Entegrasyonu

VIPER mimarisini, örnek olarak ele aldığımız bir **iOS sözlük uygulamasına** nasıl entegre edebileceğimizi inceleyelim. Bu sözlük uygulaması kullanıcıların kelime araması yapabildiği, arama sonuçlarından kelime seçerek tanımını görebildiği, kelimeleri favorilere kaydedebildiği ve çoklu dil desteği sunan bir yapıda olsun. Bu senaryoda uygulamayı VIPER modüllerine bölmek, geliştirme ve bakım açısından büyük fayda sağlayacaktır. Olası modül ve bileşen tasarımı şöyle olabilir:

- **Arama Modülü:** Kullanıcının kelime araması yapabildiği ana ekran.

- *View:* Bir arama çubuğu (`UISearchBar/UISearchController`) ve sonuçları listeleyen bir tablo (`UITableView`) içeren `AramaViewController` yer alır. Kullanıcı arama çubuğuna metin girdiğinde veya arama tuşuna bastığında, View bu olayı Presenter'a iletir (ör. `presenter.aramaYap(kelimeler: "apple")`).
- *Presenter:* `AramaPresenter` , arama metni alındığında Interactor'a sorgu yapması için talepte bulunur. Ayrıca Interactor'dan dönen arama sonuçlarını alıp, görüntülenebilir bir formata (ör. dizi halinde sonuçlar) çevirerek View'a iletir (ör. `view.sonuclariGoster(...)`). Kullanıcı bir sonuç seçtiğinde, Presenter Router'ı kullanarak Kelime Detay modülüne geçiş yapar (ör. `router.kelimeDetayiniGoster(secilenKelime)`).
- *Interactor:* `AramaInteractor` , arama metnini alarak ilgili **iş mantığını** yürütür. Örneğin bir **VeriServisi** (ağ isteği yapan bir API sınıfı veya yerel veritabanı sorgusu) üzerinden metne uygun kelime sonuçlarını getirir. Bu işlem asenkron olabilir; tamamlandığında sonuç listesini Presenter'a döner (tercihen protokol ile, ör. `AramaInteractorOutput` protokolünü implemente eden Presenter'a). Interactor ayrıca arama geçmişini tutma gibi ek mantıklar da içerebilir.
- *Entity:* Arama modülünde belirgin bir Entity olmayabilir; ancak kelime sonuçları muhtemelen bir "Kelime" modelinin basit örnekleridir. Bu model, kelimenin kendisi, anlamı, dili gibi alanları içerir. Bu Entity, Interactor içinde kullanılır. Interactor, gerekirse API'den aldığı JSON'u bu model nesnelerine dönüştürür.

- *Router:* `AramaRouter` , genellikle `AramaModuluOluşturucu` gibi statik bir fonksiyon aracılığıyla modülün başlangıç montajını (assembly) yapar. Arama ekranından detay ekranına geçiş gerektiğinde de `AramaRouter` , Kelime Detay modülünü yaratıp navigasyonu gerçekleştirir. Örneğin:

```
func kelimeDetayiniGoster(navigationController: UINavigationController,
kelime: Kelime) {
    let detayModuluVC = KelimeDetayRouter.modulOlustur(kelime: kelime)
    navigationController.pushViewController(detayModuluVC, animated:
true)
}
```

Bu sayede, seçilen *kelime* nesnesi yeni modüle aktarılır ve ekrana sunulur.

- **Kelime Detay Modülü:** Seçilen bir kelimenin tanımının, örnek cümlelerinin vs. gösterildiği ekran.

- *View:* `KelimeDetayViewController` , arayüzde kelimenin başlığını, anlamını, belki telaffuzunu veya görselini gösterir. Favorilere ekle/çıkar butonu gibi etkileşimler olabilir. View yüklenince (`viewDidLoad`) Presenter'dan veriyi göstermesini ister veya Presenter'ın talimatını bekler. Kullanıcı favori butonuna dokunduğunda, View bunu Presenter'a iletir (ör. `presenter.favoriToggled()`).
- *Presenter:* `KelimeDetayPresenter` , ilgili kelime Entity'sini ya başlangıçta almıştır (Router üzerinden) ya da Interactor'dan talep eder. View yüklendiğinde Presenter, Interactor'a "bu kelimenin detaylarını getir" diyebilir eğer ek veri gerekiyorsa. Interactor yanıt verdiğinde Presenter, veriyi uygun şekilde View'a iletir (`view.tanimGoster(tanim)` gibi). Favori butonuna basılınca Presenter, Interactor'a bu kelimenin favori durumunu değiştir komutu verebilir. Ayrıca, geri dönüş veya paylaşma gibi kullanıcı aksiyonlarında Router'ı kullanarak navigasyonu yönetir (örn. paylaşma için iOS share sheet açmak Router'ın işi olabilir).
- *Interactor:* `KelimeDetayInteractor` , belirli bir kelime için daha kapsamlı veriye ihtiyaç duyulursa bunu sağlar. Örneğin sözlük API'sinden o kelimenin detaylı tanımını, eş anlamlılarını vb. çekebilir. Veya yerel veritabanından bu kelimenin daha önce favori olarak işaretlenip

işaretlenmediğini sorgulayabilir. Interactor, bir **VeriDeposu** veya **Servis** aracılığıyla (protokol ile bağlı) çalışarak ağ veya veritabanı çağrıları yapar. Asenkron sonuçlandığında Presenter'a döner (`KelimeDetayInteractorOutput` protokolü ile). Ayrıca "Favorilere ekle/çıkar" işlemi de Interactor'da halledilir: Interactor, bir FavoriServisi (ör. Core Data yöneticisi) çağırarak kelimeyi favorilere kaydeder veya siler ve sonucunu Presenter'a bildirir.

- *Entity:* Bu modülün temel verisi "Kelime" modelidir. Kelimenin detayları (tanım, örnekler vs.) da bir Entity veya basit veri yapısı olarak modellenebilir (ör. `KelimeDetay` adında bir struct, kelimeyle ilgili tüm bilgileri içeren). Interactor bu Entity'leri kullanır ve gerektiğinde Presenter'a iletir. Presenter mümkün olduğunca Entity yerine gösterime hazır basit yapılar göndermeye çalışır (örneğin tarih formatlama, telaffuz sesini URL'den indirme gibi işler Presenter/Interactor'da halledilip View'a salt string veya hazır data verilir).

- *Router:* `KelimeDetayRouter`, bu modülün oluşumunu sağlar ve modül içinde başka bir ekrana geçiş olursa (bizim senaryoda belki yok, ama örneğin "daha fazla bilgi için web sayfası aç" gibi bir navigasyon olursa) bunu yönetir. Genelde Kelime Detay modülü, Arama veya Favoriler modülünden push edilerek açılır ve içinde farklı bir alt navigasyon yoktur, bu nedenle Router'ı basit kalacaktır. Router yine assembly fonksiyonu içerir:

```
static func modulOlustur(kelime: Kelime) -> UIViewController {
    let view = KelimeDetayViewController()
    let presenter = KelimeDetayPresenter(kelime: kelime)
    let interactor = KelimeDetayInteractor(veriDeposu:
CoreDataFavoriDeposu.shared, sozlukServisi: SozlukAPI.shared)
    let router = KelimeDetayRouter()
    // VIPER bileşenlerini birbirine bağlama
    view.presenter = presenter
    presenter.view = view
    presenter.interactor = interactor
    presenter.router = router
    interactor.output = presenter
    return view
}
```

Yukarıdaki kod parçası, Kelime Detay modülünün oluşturulup bileşenlerin birbirine **dependency injection** ile bağlandığını gösterir. Bu sayede modül kendi içinde çalışmaya hazır hale gelir.

- **Favoriler Modülü:** Kullanıcının favori olarak işaretlediği kelimeleri listeler.

- *View:* Favori kelimelerin listesi (UITableView) ve yönetim (silme vb.) arayüzü.
- *Presenter:* Favoriler listesini görüntülemek için Interactor'dan favori kayıtları ister, sonucu View'da listeletir. Kullanıcı bir favoriyi silmek isterse Presenter Interactor'a iletir, başarıyla silinirse View'ı günceller. Bir favori kelimeye tıklandığında, Presenter Router ile Kelime Detay modülüne geçiş yapar (ilgili kelimeyi parametre olarak aktararak).
- *Interactor:* Favori kelimeleri **kalıcı depodan** (Core Data, SQLite veya Realm) okur. Örneğin bir `FavoriVeriYöneticisi` (FavoriDataManager) protokolü aracılığıyla Core Data'dan verileri çeker. Favori silme/ekleme işlemlerini de aynı veri katmanı aracılığıyla yapar. Tüm bu işlemler tamamlanınca Presenter'a güncellenmiş listeyi veya başarılı/başarısız durumunu bildirir.
- *Entity:* Favori listesi muhtemelen "Kelime" Entity nesnelerinin bir listesi şeklindedir. Bunlar Interactor seviyesinde işlenir.

- *Router:* Favoriler ekranından bir kelime seçildiğinde Kelime Detay modülüne navigasyonu yönetir. Ayrıca bu modülün assembly'sini içerir (benzer bir modulOlustur fonksiyonu).

- **Dil Seçimi/ Ayarlar Modülü (opsiyonel):** Çoklu dil desteği varsa, kullanıcıya dili değiştirme imkanı sunulabilir. Bu modül, mevcut dili gösterip başka bir dil seçmeye yarar. Dil değiştiğinde uygulamanın diğer modülleri (Arama, Detay) bunu bilmeli. Bu, yaygın olarak **uygulama genelinde bir ayar** olduğu için, VIPER modülleri arasındaki paylaşım bir *Servis* aracılığıyla yapılabilir (ör. `DilYönetici` singleton'ı veya app-wide dependency). Interactor'lar bu servis aracılığıyla hangi dilin etkin olduğunu bilebilir ve arama yaparken ya da tanım gösterirken o dile özgü verileri kullanır. Örneğin AramaInteractor, arama yaparken `DilYonetici.currentLanguage` bilgisini de kullanarak doğru dildeki indeks/veri kaynağını sorgular.

Çoklu Dil Desteği Entegrasyonu: Sözlük uygulaması birden fazla dilde sözlük sağlıyorsa, mimari olarak bunu desteklemek için birkaç yaklaşım düşünülebilir:

- Uygulama genelinde aktif dili tutan bir **ayar** (ör. seçili dil kodu). Bu ayar, VIPER modüllerindeki Interactor veya Servis katmanı tarafından okunur. Mesela `SozlukServisi` sınıfı veya API istemcisi, istek yaparken bu dil parametresini kullanır. VIPER'da bu servis, Interactor'a protokolle enjekte edilir, böylece Interactor sadece `veriServisi.getDefinitions(for: word)` gibi bir çağrı yapar; servis içeride doğru dildeki uç noktayı kullanır.

- Eğer diller tamamen ayrı veri setleri ise (örneğin İngilizce-Türkçe sözlük ve Türkçe-İngilizce sözlük gibi), her dil için ayrı bir modül yapısı da kurulabilir. Fakat genellikle daha pratik olan, **aynı modülün** dil parametresiyle çalışmasıdır. Örneğin AramaInteractor içinde:

```
let dil = DilYonetici.shared.aktifDil // "EN" veya "TR"
veriServisi.kelimeAra("apple", dil: dil) { sonuçlar in ... }
```

  şeklinde bir kullanım olabilir. Presenter da gerekirse dil bilgisini View'a (UI metinleri için) iletebilir ancak çoğu durumda iOS yerelleştirme mekanizmasıyla halledilecektir.

- Çoklu dil, sadece arayüzün dilini değil, aranan kelimelerin dilini de etkiliyorsa (iki ayrı sözlük verisi gibi), bunu da servis katmanında halletmek akıllıcadır. Örneğin `SozlukVeriDeposu` protokolü, `func ara(kelime: String, dil: DilKod) -> [Kelime]` gibi bir imzaya sahip olabilir. Interactor da bu protokolü kullanarak dil parametresini iletir. Bu sayede Interactor'un kendisi dil detaylarına takılmadan, sadece geçerli dili ilgili servise aktarır.

Özetle, VIPER mimarisinde çoklu dil desteği, **bağımlılık enjeksiyonu** ve **servis katmanı** ile kolaylıkla entegre edilebilir. Modüller, dil gibi global bir durumu doğrudan bilmez; bu bilgiyi onlara sağlayan bir servis veya yönetici sınıf vardır. Bu da yine tek yönlü bağımlılık ilkesine uygundur: UI veya Presenter katmanı, aktif dil değiştiğinde kendini güncellemesi gerektiğinde, bunu bir bildirim mekanizmasıyla yapabilir (ör. NotificationCenter veya bir Observer pattern ile DilYonetici değişikliği bildirir, Presenter o bildirimi alıp View'ı yeniler). Ancak bu tarz detaylar uygulamanın kapsamına göre planlanır.

VIPER Entegrasyonu Özet: Yukarıda tasarladığımız şekilde, sözlük uygulamasının her özelliği net bir VIPER modülüne ayrılmış olur. Örneğin Arama modülü sadece arama işine odaklanırken, Kelime Detay modülü yalnızca seçilen kelimenin sunumuna odaklanır. Bu modüller Router'lar aracılığıyla birbirine bağlanır ve veri transferi yapar. Bu sayede, örneğin arama sonucunda bulunan bir kelimenin detayını

göstermek istediğimizde, AramaRouter yeni bir KelimeDetay modülü yaratıp kelime bilgisini ona verir; KelimeDetayInteractor ihtiyaç duyarsa daha fazla veriyi yükler ve Presenter-View aracılığıyla kullanıcıya sunar. Kullanıcı "favorilere ekle" dediğinde, KelimeDetayPresenter, Favori servisini çağırması için Interactor'ı kullanır; Interactor favoriyi kaydeder ve Presenter'a başarı mesajını döner; Presenter da View'a bir onay mesajı gösterebilir.

Bu entegrasyon sayesinde uygulama akışı, VIPER'ın öngördüğü gibi kontrol altında olur. Hangi işlemin nerede yapıldığı bellidir: Arama sorgusu Interactor'da, navigasyon Router'da, veri formatlama Presenter'da, görüntüleme View'da. Sonuç olarak sözlük uygulaması, **düzenli, genişlemeye açık ve test edilebilir** bir yapıda geliştirilmiş olur.

Clean Architecture ve MVVM ile Kıyaslama

VIPER mimarisini daha iyi anlamak için onu diğer yaygın mimarilerle de kıyaslayabiliriz:

Clean Architecture (Temiz Mimari): VIPER aslında Clean Architecture'ın iOS dünyasındaki somut bir uygulamasıdır [1]. Robert C. Martin'in ortaya attığı Clean Architecture prensipleri, bağımlılıkların soyutlamalar etrafında tek yönlü olmasını ve katmanlı bir yapı kurulmasını önerir. VIPER da benzer şekilde, Entity-Interactor-Presenter gibi katmanlarla iş mantığı ile arayüzü ayrıştırır. Clean Architecture'da genellikle "Use Case" veya "Interactor" katmanı, "Entities" (iş modelleri) ve "Interface Adapters / Presenters" gibi bileşenler tanımlıdır – VIPER bunları isimlendirip iOS'a uygular. Yani VIPER, Clean Architecture'ın pratik bir şablonudur denebilir. Bu yüzden VIPER kullandığınızda, aslında Clean Architecture prensiplerini (SOLID, bağımlılıkların merkezden dışa akması vs.) uygulamış olursunuz. Clean Architecture kavramında esneklik biraz daha yüksektir (katman sayısı ve isimleri projeye göre değişebilir), VIPER ise bunları 5 bileşende somutlaştırmış ve çerçevelemiştir. Dolayısıyla Clean Architecture tartışmalarında bahsedilen çoğu fayda (bağımlılık izolasyonu, test edilebilirlik, sürdürülebilirlik) VIPER için de geçerlidir.

MVVM (Model-View-ViewModel): MVVM, iOS (özellikle **SwiftUI** ile) oldukça popüler bir mimari kalıptır. MVVM'de temel olarak View (UI katmanı) ve ViewModel (durum ve sunum mantığı) ayrımı vardır, Model ise veri katmanını temsil eder. MVVM, VIPER'a göre daha **basit ve hafif** bir yapıdır. Birçok senaryoda MVVM yeterli modülerliği sağlayabilir ve fazla boilerplate gerektirmez. Örneğin MVVM'de `ViewController` ile iş mantığını ayırmak için bir ViewModel sınıfı yaratır, bu ViewModel genelde `Observable` (Combine Publisher, KVO, Delegation vs.) olarak View ile data bağlar. VIPER ise bu ayrımı daha ileri götürerek sadece ViewModel değil, ek olarak Interactor ve Router katmanlarını da tanıtır.

VIPER ile MVVM karşılaştırıldığında:
- MVVM öğrenmesi ve kodlaması nispeten kolay, daha **az seremonili** bir yapıdır. Küçük ve orta ölçekli projelerde, veya arayüzün çok karmaşık olmadığı durumlarda, MVVM geliştirme hızını yüksek tutabilir [48]. Özellikle **SwiftUI** MVVM pattern'ını teşvik ettiğinden, yeni projelerde MVVM sıkça tercih edilir.
- VIPER ise **daha büyük ve karmaşık uygulamalarda** yapıyı korumak için avantajlıdır. Katı ayrımlar sayesinde, MVVM'de ViewModel'lerin şişerek yaşayabileceği sorunlar VIPER'da önlenir (ama dikkat: Presenter'ın şişmemesi için de özen göstermek gerekir) [49]. VIPER, MVVM'ye göre **daha fazla kod** anlamına gelir, ancak bu kod disiplin altındadır ve büyük ekiplerin çalışmasını kolaylaştırır [31]. Reusable component (yeniden kullanılabilir bileşen) oranı VIPER'da yüksek olabilir; çünkü modüller iyi soyutlandığı için başka projelere bile taşınabilir.

    - MVVM genelde veri bağlama (data binding) tekniğini kullanır. VIPER'da veri bağlama doğrudan yoktur; bunun yerine geleneksel delegate/protocol veya closure mekanizmaları kullanılır. Bu açıdan MVVM, özellikle UI değişikliklerine reaktif olarak yanıt vermek için (ör. Combine veya

RxSwift kullanarak) daha modern bir yaklaşım sunar. VIPER'da isterseniz Presenter ile View arasında ReactiveCocoa/Combine gibi araçlar yine kullanabilirsiniz [50], ancak MVP tarzı protokol iletişimi da genellikle yeterlidir.

Özetle, **hangi mimarinin seçileceği projeye göre değişir**. Küçük ve sık değişen projelerde MVVM'nin sadeliği zaman kazandırabilir. Büyük, uzun vadeli ve birden çok kişinin çalıştığı projelerde VIPER'ın getirdiği düzen uzun vadede fayda sağlayabilir [44] [31]. Bazı durumlarda, VIPER'ın katmanlarını ihtiyaca göre adapte etmek de mümkün (örneğin Router'ı basitleştirmek veya Presenter-Interactor yerine sadece ViewModel kullanmak, ki bu tür varyasyonlar Clean Swift gibi yaklaşımlarda görülür). Mühim olan, projenin karmaşıklığı ile mimarinin getirileri arasındaki dengeyi bulmaktır. Bir söz vardır: *"En iyi mimari, ekibin ve projenin kaldırabileceği en basit mimaridir."* Bu bağlamda VIPER, belli bir ölçekteki projelerde ideal olabilirken, gereksiz olduğu durumlar da olabilir. Ancak VIPER'ı öğrenmek, bir geliştiriciye mimari bakış açısı kazandırdığı için değerlidir; gerektiğinde proje içinde sadece en kritik ekranlarda bile uygulanabilir.

VIPER ile Dependency Injection ve Servis Katmanı

Dependency Injection (DI) - Bağımlılık Enjeksiyonu: VIPER mimarisi, bağımlılık enjeksiyonunu doğal olarak destekler ve teşvik eder. Aslında VIPER'ın kendisi, bağımlılıkları yönetmenin bir yoludur: Örneğin Presenter, ihtiyacı olan Interactor'ı kendisi yaratmaz, ona dışarıdan (genellikle Router/Assembler tarafından) verilir. Benzer şekilde Interactor, bir veri servisine ihtiyaç duyuyorsa, onu ya initializer yoluyla alır ya da dışarıdan enjekte edilir. Bu yaklaşım, birimi test ederken gerçek bağımlılık yerine **mock** (sahte) nesne vermeyi kolaylaştırır.

VIPER modüllerinde genellikle modülün Router'ı veya özel bir *Assembly/Configurator* kodu, ilgili tüm parçaları oluşturup birbirine bağlar (daha önceki `modulOlustur` örneğinde olduğu gibi). Bu noktada Interactor'a ihtiyaç duyduğu servisler enjekte edilebilir. Örneğin:

```
let interactor = AramaInteractor(veriServisi: SozlukServisi(), favoriDeposu:
FavoriDepo())
```

şeklinde Interactor yaratılırken, uygulamanın servis katmanındaki **SozlukServisi** ve **FavoriDepo** nesneleri verilir. Interactor, bu nesnelere genellikle protokol aracılığıyla erişir (yani `SozlukServisi` bir protokol, somut sınıfı sonradan belirleniyor olabilir). Bu sayede, gerçek servis yerine testlerde farklı bir implementasyon da takılabilir.

Swift ekosisteminde DI yapmanın farklı yolları vardır. En basiti, yukarıdaki gibi initializer ile geçmek veya property olarak atamaktır. Bazı projelerde **Swinject** gibi kütüphaneler kullanılarak konteyner tabanlı DI yapılır; modüller ve servisler bu konteynıra kayıt edilir ve çözülür. VIPER ile Swinject kullanımı da meşhurdur – özellikle VIPER'ın çok sayıda nesne yaratma ihtiyacı düşünüldüğünde, Swinject bu işi dinamik olarak yönetebilir. Ancak, Swinject olmadan da VIPER modüllerini manuel olarak oluşturmak gayet mümkündür ve kontrol geliştiricide olur.

Servis Katmanı Entegrasyonu: VIPER, iş mantığı ile veri kaynağı arasında genellikle bir **Servis veya Data Manager** katmanı kullanılmasını önerir. Interactor içinde doğrudan network çağrısı yapmak veya veritabanı işlemi yazmak yerine, bu işleri yapan servis sınıfları kullanılır. Örneğin, sözlük uygulamamızda: - `SozlukAPIServisi` adında bir sınıf, belirli bir dil ve kelime için sunucudan tanım çeken bir metot sağlayabilir. - `FavoriVeriDeposu` adında bir sınıf, favori kelimeleri saklamak/okumak için Core Data işlemlerini kapsülleyebilir.

Interactor bu servislerin protokollerini bilir, ama onların nasıl çalıştığını bilmez. Sadece “favorite kaydet” der, servis içerde Core Data işlemini yapar; veya “kelime ara” der, API servisi HTTP isteğini yapar. Bu servislerin protokollerini tanımlayıp, gerçek implementasyonları uygulamanın **Utils** veya **Data** katmanında tutmak en iyi uygulamalardandır [51] [52]. Cheesecake Labs’in VIPER kılavuzunda da belirtildiği gibi, *Data Manager* denen bu servisler API ve Local (yerel veritabanı) olarak ayrılabilir ve Interactor içerisinde uygun yerde çağrılabilir [53] [54]. Böylece Interactor kodu daha okunaklı olur ve tek bir sınıfın çok fazla sorumluluk almasının önüne geçilir.

Servis katmanı entegrasyonu, VIPER’da modülerliği bozmaz, aksine güçlendirir: Bir Interactor sadece protokole dayalı bir servis kullanıyorsa, yarın o servisin implementasyonunu değiştirmek tüm modülü etkilemez. Örneğin bugün `SozlukServisi` internetten veri çekiyordur, yarın değiştirip cihaz içindeki bir JSON dosyasından okumaya karar verdiniz. Tek yapmanız gereken, `SozlukServisi` protokolünü diskten okuyan bir sınıfla implement etmek ve DI konteynırından bunu sağlamaktır; Interactor aynı kalacaktır. Bu esneklik, Clean Architecture’ın da vurguladığı üzere, **bağımlılık tersine çevrimi** (Dependency Inversion) ilkesinin bir sonucudur [55] [56].

Örnek: Sözlük uygulamamızda AramaInteractor şöyle tanımlanabilir:

```
protocol SozlukVeriServisi {
    func ara(terim: String, dil: DilKod, tamamlandi: @escaping ([Kelime]?) -
> Void)
}
protocol FavoriDeposu {
    func favorileriGetir() -> [Kelime]
    func favoriEkle(_ kelime: Kelime)
    func favoriSil(_ kelime: Kelime)
}
class AramaInteractor {
    var sozlukServisi: SozlukVeriServisi
    var favoriDeposu: FavoriDeposu
    weak var output: AramaInteractorOutput? // Presenter
    ...
    init(sozlukServisi: SozlukVeriServisi, favoriDeposu: FavoriDeposu) {
        self.sozlukServisi = sozlukServisi
        self.favoriDeposu = favoriDeposu
    }
    func kelimeAra(_ sorgu: String) {
        let dil = DilYonetici.shared.aktifDil
        sozlukServisi.ara(terim: sorgu, dil: dil) { [weak self] sonucListesi
in
            if let liste = sonucListesi {
                self?.output?.aramaSonucuGetirildi(liste)
            } else {
                self?.output?.aramaBasarisiz()
            }
        }
    }
}
```

Yukarıdaki pseudo-kodda Interactor, herhangi bir networking detayıyla uğraşmıyor; sadece servis üzerinden arama yapıyor. Sonuç gelince de Presenter'a iletiyor. Bu yaklaşım **asenkron veri akışı** ile de uyumludur: VIPER, Swift'in `async/await` özelliği ile de kullanılabilir. İster callback, ister Combine publisher, ister async fonksiyon kullanın – mimarinin akışını değiştirmeden uygulayabilirsiniz. Mesela Swift 5.5 ile `kelimeAra` metodunu:

```
func kelimeAra(_ sorgu: String) async {
    let dil = DilYonetici.shared.aktifDil
    do {
        let liste = try await sozlukServisi.aramaYap(terim: sorgu, dil: dil)
        await output?.aramaSonucuGetirildi(liste)
    } catch {
        await output?.aramaBasarisiz()
    }
}
```

şeklinde async hale getirmek mümkün. Presenter da bu async metodu çağırırken `Task` ile çağırabilir. Önemli olan, Interactor içinde bekleme yaptığımızda, bitince Presenter'a haber vermeyi unutmamak ve ana thread'e dönerek View'ın güncellenmesini sağlamaktır (UIView bileşenleri ana thread'de güncellenmelidir). Bu sorumluluk genellikle Presenter'da olur; Presenter, Interactor'dan aldığı callback'i veya geleceği (`Future`) yakalayıp, main thread üzerinde View'a iletir.

Kalıcı Depolama (Core Data/Realm) Kullanımı: VIPER'da verinin uzun süreli saklanması için Core Data veya Realm gibi teknolojiler kullanılabilir. Burada kritik nokta, bu çerçevelerin nesnelerinin **Interactor seviyesinin altındaki katmanda kalması** gerektiğidir. Objc.io'daki VIPER makalesinde belirtildiği üzere, eğer Core Data kullanıyorsanız `NSManagedObject`'lerinizi olabildiğince **veri katmanının arkasında** tutun, Interactor içinde doğrudan `NSManagedObjectContext` işlemleri yapmayın [57] [58]. Bunun yerine, bir **Data Manager** sınıfı bu işleri üstlensin: - Data Manager, Core Data yığınını (Persistent Container vs.) yönetir, - İhtiyaç duyulan sorguları içerir (ör. favorileri çekmek için bir `NSFetchRequest` çalıştırır), - Dönen `NSManagedObject`'leri olabildiğince çabuk **Plain Old Swift Object** (PONSO) denen saf modellere çevirir [58].

Böylece Interactor, Core Data bağımlılığı taşımaz; yalnızca Data Manager'dan gelen Swift model nesneleriyle çalışır. Örneğin `FavoriDeposu` protokolünün Core Data implementasyonu, `Kelime` nesnelerini Core Data'daki `KelimeEntity` varlıklarından dönüştürerek verir. Interactor bu ayrıntıyı bilmez, sadece protokolü kullanır. Bu yaklaşım, ileride depo teknolojisini değiştirmeyi (Realm'e geçmek vs.) kolaylaştırır ve testlerde bellek içi sahte depolar kullanmaya imkan tanır.

Realm için de benzer prensipler geçerli. Realm'in `Object` sınıfları doğrudan Entity olarak kullanılabilir ancak bunları da Interactor içinde değil, bir Depo katmanında kullanmak ve Interactor'a normal Swift modelleri sunmak temizlik sağlar. Örneğin `FavoriDepoRealm` sınıfı, Realm'den okuduğu `FavoriObject` listelerini `Kelime` model listesine çevirip dönebilir.

Özetle, VIPER mimarisinde DI ve servis katmanı kullanımı, kodunuzun **daha esnek, test edilebilir ve düzenli** olmasını sağlar. Interactor ve Presenter'ın odakları dağılmaz; networking, veritabanı gibi konular ayrı servislerde soyutlanır. Uygulamanın ilerleyen safhalarında bu servisler değişse bile (örneğin farklı bir API sürümüne geçiş, veya yerel veri formatının değişmesi), VIPER katmanlarınız etkilenmeden uyarlamalar yapılabilir [58] [59].

VIPER Katmanları Arasında İletişim ve Protokoller

VIPER'da katmanlar arasındaki iletişim **belirlenmiş arayüzler (protokoller)** üzerinden yapılır. Bu, mimarinin bel kemiğidir: Katmanlar birbirlerinin implementasyon detaylarını bilmez, sadece sözleşmelerini bilir. Bu yaklaşıma genelde **"Loose Coupling" (gevşek bağlılık)** denir ve esnekliği, test edilebilirliği artırır.

İletişim yönlerine göre protokol kullanımı:

- **Presenter → View:** Presenter, View'ı kontrol ederken bir protokol üzerinden çağrı yapar. Örneğin `KelimeDetayViewProtocol` gibi bir protokol tanımlanır ve ViewController bu protokolü uygular. Presenter, `view?.tanimGoster("Anlamı şudur...")` dediğinde aslında `KelimeDetayViewProtocol`de tanımlı `tanimGoster` metodunu çağırır. Bu sayede Presenter, View'ın tam olarak hangi sınıf olduğunu bilmez (Mock bir view ile test edilebilir). Protokol, View'ın göstermesi gereken içerik düzeyinde metodlar içerir (örn. `func tanimGoster(_ metin: String)` veya `func gorselGuncelle(imageData: Data)`). Bu arayüz, Presenter'ın View'a **ne** göstereceğini belirtir, **nasıl** göstereceğini View'a bırakır [60]. Bu sayede, Presenter'ın testinde sahte bir View implementasyonu kullanarak çağrıların doğru yapıldığını kontrol edebilirsiniz.

- **View → Presenter:** ViewController, kullanıcı etkileşimlerini Presenter'a iletirken yine bir protokol kullanır. Genellikle Presenter protokolü (ör. `KelimeDetayPresenterProtocol` veya `KelimeDetayViewOutput`) tanımlanır ve Presenter bunu uygular. ViewController, bu protokol üzerinden Presenter'a erişime sahip olur. Örneğin View'daki bir buton aksiyonunda `presenter.favoriButonunaBasildi()` çağrılır. Bu protokol, View'ın tetikleyebileceği eylemleri tanımlar. Bu sayede View, Presenter'ın iç işlerini bilmez; sadece protokol üzerinden bildirir. Bu da birim testlerde, Presenter yerine geçen sahte bir nesneyle View'ın davranışlarını test etmeyi mümkün kılar.

- **Presenter → Interactor:** Presenter, veri veya iş mantığı gerektiğinde Interactor'a protokol aracılığıyla çağrı yapar. Örneğin `KelimeDetayInteractorProtocol` tanımlı olsun, Presenter içinde `interactor?.favoriDegistir(kelime)` gibi bir çağrı yapılır. Presenter, Interactor'un implementasyonunu (ağ çağrısı yapıyor mu, yerel mi) bilmez, sadece protokole güvenir. Bu, Interactor'u kolay test edilebilir kıldığı gibi, farklı Interactor implementasyonlarıyla çalışmayı da mümkün kılar (örneğin aynı arayüze sahip farklı stratejiler). Genellikle bu protokolü Interactor uygular.

- **Interactor → Presenter:** Interactor işini bitirdiğinde (örneğin arama sonuçlarını aldığında veya bir işlem başarısız olduğunda), Presenter'a geri bilgi verir. Bunu genellikle bir **output protokolü** ile yapar. Örneğin `KelimeDetayInteractorOutput` protokolü Presenter tarafından uygulanır, Interactor ise bu protokol üzerinden Presenter'ı haberdar eder (ör. `output?.favoriIslemeSonucu(.basarili)` gibi). Bu, **Inversion of Control** prensibini de yansıtır: Interactor, doğrudan Presenter'ı çağırmaz; onun arayüzünü tanımlayan output protokolü üzerinden çağırır [61] [62]. Bu sayede Interactor, kimin output olduğunu bilmez; Presenter değişse bile Interactor kodu değişmez.

- **Presenter → Router:** Presenter, navigasyon gerektiğinde Router protokolünü kullanır. Örneğin `KelimeDetayRouterProtocol` de olabilir veya Presenter direkt Router sınıfına erişebilir. İyi pratik, Router'ın da bir arayüzünün olması ve Presenter'ın onu kullanmasıdır. Mesela `router?.navigateToSettings()` gibi bir çağrı, Router implementasyonunda yeni modül

oluşturup ekrana getirme işini yapar. Bu da test edilebilir: Eğer navigasyonu test etmek isterseniz, sahte bir Router ile Presenter'ın doğru fonksiyonu çağırıp çağırmadığını kontrol edebilirsiniz.

Protokollerin Tanımlanması: Genellikle VIPER modülünün başında, tüm bu iletişim arayüzlerini tanımlayan bir "Contract" Swift dosyası oluşturulur [63] [64]. Bu contract dosyasında örneğin:

```
protocol KelimeDetayViewProtocol: AnyObject { ... }
protocol KelimeDetayPresenterProtocol: AnyObject { ... }
protocol KelimeDetayInteractorProtocol: AnyObject { ... }
protocol KelimeDetayInteractorOutput: AnyObject { ... }
protocol KelimeDetayRouterProtocol: AnyObject { ... }
```

gibi tüm protokoller yer alır. Bu, modülün iç bileşenlerinin nasıl konuştuğunu bir bakışta anlamayı sağlar. Ayrıca, geliştirmeye başlamadan önce bu protokolleri düşünmek, **veri akışını planlamak** açısından yararlıdır: Hangi veriler View'dan Presenter'a gidecek, Presenter Interactor'dan ne isteyecek, Interactor'un döndüreceği şey ne olacak gibi sorular cevaplanmış olur.

Loose Coupling'in Faydası: Tüm bu protokol bazlı iletişim, VIPER modüllerinin birbirinden ve detay implementasyonlardan bağımsız olmasını sağlar [3]. Örneğin UI tamamen değişse bile (Storyboard yerine programatik UI, ya da UIKit yerine SwiftUI), Presenter-Interactor mantığınız ve Entity'leriniz bozulmaz; sadece View katmanını değiştirip yeni View'ın Presenter protokolünü çağırmasını sağlarsınız. Aynı şekilde bir servis katmanı değişse bile Interactor'un protokol arayüzünü koruduğunuz sürece Presenter ve diğer kısımlar etkilenmez.

VIPER'ın bu yapısı başlangıçta biraz fazladan iş gibi görünse de, özellikle **büyük projelerde hata yakalamayı** kolaylaştırır. Eğer modüller arası bir veri akışı protokol tanımıyla yapılmışsa, derleyici size eksik implementasyonları veya yanlış tipte verileri hemen gösterecektir. Bu, runtime'da ortaya çıkabilecek hataları en aza indirir.

Ek olarak, VIPER yapısı iOS'un MVC paradigmalarına tam uymadığı için, protokoller yardımıyla adaptasyon yapar. Apple'ın kendi MVC modelinde çoğu şey ViewController etrafında dönerken, VIPER bunu parçalar. Bu parçalar arasındaki yapıştırıcı da protokollerdir. Bu sayede, **UIKit ile barışık bir ayrım** elde edilir: ViewController halen View'dir (ve delege/datasource gibi UIKit görevlerini yapar), Presenter halen kullanıcı aksiyonlarını dinler ve yönlendirir, Interactor da arka planda çalışır. Protokoller, bu rolleri netleştirir ve istek/yanıt ilişkisini tanımlar.

Sonuç olarak, VIPER'da katmanlar arası iletişim, **protokoller ve delegasyon** mekanizmasıyla gerçekleşir. Bu yapı, kodu esnek, anlaşılır ve bakımını kolay kılar. Protokoller sayesinde bağımlılıklar *soyut* hale gelir – bir bileşeni yenisiyle değiştirmek (örn. farklı bir Interactor ya da farklı bir View) asgari eforla olur, çünkü diğerleri sadece arayüzü görür. Bu da VIPER'ın en güçlü yanlarından biridir: Değişime karşı koyabilen ama aynı zamanda değişime adapte olabilen bir mimari.

"Kelime Detayı" VIPER Modülü: Örnek Yapı

Bir VIPER modülünün somut olarak nasıl yapılandığını görmek için, sözlük uygulamamızdaki **"Kelime Detayı"** ekranını ele alalım. Kullanıcı bir kelimeye tıkladığında açılan bu ekranda, kelimenin tanımını,

belki telaffuzunu ve ilgili birkaç bilgiyi gösteriyoruz. Ayrıca kullanıcı bu ekrandan kelimeyi favorilere ekleyebiliyor. Şimdi bu ekranın VIPER modülünü bileşen bileşen inceleyelim:

1. **View (KelimeDetayViewController):**
   UIKit tarafındaki görünüm bu sınıf tarafından kontrol edilir. `KelimeDetayViewController`, muhtemelen bir XIB veya Storyboard ile tasarlanmış arayüzü yönetir. Ekranda kelime ve tanım için etiketler, favori ekleme için bir buton gibi UI elemanları vardır. ViewController, `KelimeDetayViewProtocol` arayüzünü uygular. Bu protokol, Presenter'ın View'a verebileceği komutları içerir. Örneğin:

   ```
   protocol KelimeDetayViewProtocol: AnyObject {
       func tanimGoster(_ tanimMetni: String)
       func favoriDurumunuGuncelle(_ favoriMi: Bool)
       func hataMesajiGoster(_ mesaj: String)
   }
   ```

   ViewController, bu metodları implemente ederek Presenter'dan gelecek verilere hazır olur. Örneğin `tanimGoster` çağrıldığında ilgili UILabel'ın text'ini ayarlar. Diğer yandan, ViewController kullanıcı etkileşimlerini Presenter'a iletmek için bir referansa sahiptir:

   ```
   var presenter: KelimeDetayPresenterProtocol!
   ```

   Buton aksiyonlarında veya `viewDidLoad` içinde Presenter'ı haberdar eder:

   ```
   @IBAction func favoriButonuTapped() {
       presenter.favoriButonunaBasildi()
   }
   override func viewDidLoad() {
       super.viewDidLoad()
       presenter.viewYuklendi()
   }
   ```

   Bu sayede ViewController mümkün olduğunca "pasif" kalır ve karar verme işini Presenter'a bırakır (yalnızca UIKit'in gerektirdiği, örneğin tablo datasource, gibi konuları yapar).

2. **Presenter (KelimeDetayPresenter):**
   `KelimeDetayPresenter`, `KelimeDetayPresenterProtocol` arayüzünü uygular (bu protokol, View'ın çağırabileceği metotları belirler). Örnek:

   ```
   protocol KelimeDetayPresenterProtocol: AnyObject {
       func viewYuklendi()
       func favoriButonunaBasildi()
   }
   ```

   Presenter, ayrıca `KelimeDetayInteractorOutput` protokolünü de uygulayarak Interactor'dan gelen geri bildirimleri alır. Bu Presenter sınıfı, modülün beynidir:

3. View yüklendiğinde (`viewYuklendi`) Interactor'a "ilgili kelimenin detayını getir" komutunu verebilir. Ya da eğer Router üzerinden halihazırda kelime bilgisini aldıysa (diyelim ki kelime ve bir ön tanım Router'dan init ile gelmiş), o zaman direkt View'a gösterebilir. Bu karar, mimari detaylara göre değişebilir. Profesyonel pratikte, Presenter genellikle Router'dan kelime nesnesini alır ve Interactor'a daha fazlası gerekirse sorar.
4. Kullanıcı favori butonuna tıkladığında, Presenter Interactor'a favori durumunu değiştir komutu verir. Ardından, Interactor'dan başarı yanıtı gelince View'a favori ikonunu güncellemesini söyler.
5. Interactor bir hata bildirirse (örneğin tanım yüklenemedi), Presenter uygun bir hata mesajı hazırlar ve `view.hataMesajiGoster("Tanım yüklenemedi")` şeklinde View'a iletir.
6. Presenter aynı zamanda Router'ı kullanarak navigasyonu tetikler. Bu modülde belki navigasyon sadece geri dönmek olabilir (o da iOS'un geri navigasyonu ile doğal), ama örneğin "Paylaş" butonu olsa, Presenter `router.paylas(kelime)` diyerek Router'ın iOS share sheet açmasını sağlayabilir.

Presenter'ın önemli bir görevi de **veri formatlamaktır**. Interactor'dan gelen Entity'ler ham olabilir; Presenter bunları kullanıcıya sunulacak hale getirir. Örneğin Interactor, "Kitap" kelimesinin bir `KelimeDetay` modeli döndürdü diyelim (içinde tanım metni, benzer kelimeler listesi vs. var). Presenter belki tanım metnini olduğu gibi View'a iletir ama benzer kelimeler listesini bir cümle haline getirip öyle gönderir. Veya tarih/ saat gibi veriler varsa uygun formatlama yapar. Böylece View sadece aldığı string veya hazır data'yı ekrana basar, bu mantıkla uğraşmaz [65] [66]. Bu, MVP/MVVM mimarilerinde de benzer bir görevdir ve **Presenter'i ViewModel gibi** konumlandırır.

1. **Interactor (KelimeDetayInteractor):**
   `KelimeDetayInteractor`, `KelimeDetayInteractorProtocol` arayüzünü uygular. Bu protokol, Presenter'ın isteyebileceği işlemleri tanımlar:

   ```
   protocol KelimeDetayInteractorProtocol: AnyObject {
       func detayYukle(kelime: Kelime)
       func favoriDegistir(kelime: Kelime)
   }
   ```

   Interactor'ın ayrıca Presenter'a bilgi vermek için bir output'u vardır:

   ```
   var output: KelimeDetayInteractorOutput? // Presenter
   ```

   Interactor'ın asıl rolü, **iş kurallarını** uygulamak ve gerekli veri erişimlerini yapmaktır. Kelime Detay özelinde:
2. `detayYukle` çağrıldığında, ilgili kelimenin detayını veritabanından veya API'den getirir. Örneğin `sozlukServisi.getDefinition(for: kelime)` metodunu çağırır. Asenkron olarak yanıt geldiğinde, eğer başarıysa output'a `detayYuklendi(veri)` şeklinde döner; hata ise `detayYuklenemedi(hata)` döner.
3. `favoriDegistir` çağrıldığında, favori depo servisi aracılığıyla bu kelime favorilerdeyse çıkarır, değilse ekler. Sonra sonucunu output'a iletir (`favoriDurumGuncellendi(yeniDurum)` gibi).
4. Interactor kendi içinde karmaşık iş mantıkları da barındırabilir. Örneğin bir kelimenin günlük kullanım istatistiğini tutmak veya her görüntülenmede sayaç arttırmak gibi bir kural varsa, Presenter'dan bağımsız olarak bu kuralı uygulayıp sonucu arka planda kaydedebilir. Bu tip iş odaklı şeyler Interactor'ın alanıdır.
5. Interactor UI hakkında hiçbir şey bilmez. Örneğin bir kelimenin tanımı çok uzunsa kısaltalım mı gibi bir karar Interactor'da verilmez; bu Presenter/View kararına bırakılır. Interactor sadece veriyi olduğu gibi (ya da iş mantığında işleyip) sunar.

Interactor'un veri erişimi, yukarıda bahsettiğimiz gibi servisler aracılığıyla olur. Kelime Detay için muhtemelen iki servis kullanılıyor: Sözlük veri servisi (tanım çekmek için) ve favori depo (favori durumunu okumak/yazmak için). Bu servisler, Interactor'a DI ile sağlanır. Interactor bu sayede testlerde de kolayca taklit edilebilir hale gelir (sahte servis ile denenebilir).

1. **Entity (Kelime/KelimeDetay Modeli):**
   Bu modüldeki Entity, büyük ihtimalle uygulama genelinde de kullanılan bir modeldir. "Kelime"yi temsil eden bir veri yapısı düşünelim:

   ```
   struct Kelime {
       let id: String
       let baslik: String
       let dil: DilKod
   }
   struct KelimeDetay {
       let kelime: Kelime
       let tanim: String
       let ornekCumleler: [String]
       let benzerKelimeler: [Kelime]
   }
   ```

   Burada `KelimeDetay`, bir kelimeye dair tüm bilgileri taşıyan, Entity konumunda bir model. Interactor belki API'den JSON çekip bu modele çeviriyor. Bu Entity, Presenter'a kadar gidebilir veya Presenter daha küçük parçalara ayırıp View'a iletebilir. VIPER prensibine göre, Interactor Entity'yi doğrudan Presenter'a vermemeli, bunun yerine daha basit bir yapı geçirmelidir der [67] [68]. Ancak pratikte, eğer Entity zaten bir PONSO ise ve üzerinde iş kuralı yoksa, Presenter'a iletilebilir de. Yine de, Presenter'ın bu Entity'yi işleyip View'a sunması tercih edilir.

Entity'ler genelde *"anemik"* (davranışı olmayan, sadece veri taşıyan) yapılardır [69]. Tüm mantık Interactor'da olmalıdır. Bu da test edilebilirliği artırır, çünkü Entity'leri dilediğiniz gibi oluşturup Interactor veya Presenter testlerinde kullanabilirsiniz. Bir de VIPER felsefesi, Entity'lerin Presenter'a direkt geçmemesini söyleyerek, Presenter'da "gerçek iş" yapılmasının önüne geçmek ister [16]. Yani Presenter sadece formatlama yapsın, asıl işlem Interactor'da bitsin. Bizim senaryomuzda da bu ayrımı koruyabiliriz: Interactor, `KelimeDetay` nesnesini oluşturup Presenter'a verebilir; Presenter da bunu kullanarak string'ler oluşturur vs. Veya Interactor, tanım string'i ve favori bool'u ayrı ayrı da verebilir. Tasarım tercihlerine göre şekillenebilir.

1. **Router (KelimeDetayRouter):**
   Router, bu modülün oluşturulması ve diğer modüllerle bağlantısını yönetir. `KelimeDetayRouterProtocol` tanımlanabilir ama çoğu zaman Router'ın kendisi dış dünya tarafından pek çağrılmaz, aksine Presenter onu kullanır. Bu modül özelinde Router şunları yapar:
2. **Montaj (Assembly):** Az önceki kod örneğinde görüldüğü gibi, tüm bileşenleri oluşturup birbirine bağlayarak bir `KelimeDetayViewController` örneği döndürür. Bu fonksiyon, modulün public giriş noktasıdır. Dışarıdan bu fonksiyon çağrılarak yeni bir modül örneği alınır. Router burada DI görevini de üstlenmiş olur (servisleri yaratıp Interactor'a vermek vs).
3. **Navigasyon İşlemleri:** Presenter'dan gelebilecek navigasyon isteklerini karşılar. Mesela Presenter, "bu kelimeyi Safari'de aç" dese, Router `UIApplication.shared.open(url)` diyerek uygular. Veya bu modülde olmasa da, başka modüle geçiş Router'ın işidir (bizim örneğimizde Kelime Detay Router belki "Daha fazla bilgi" modülü açabilir farz edelim).

4. **Veri Aktarımı:** Router, yeni modül oluştururken veri aktarımını yapar. Örneğimizde AramaRouter, KelimeDetayRouter.modulOlustur(kelime: Kelime) metoduna parametre olarak Kelime'yi geçiriyor. KelimeDetayRouter da bu parametreyi alıp Presenter veya Interactor'a iletecek şekilde kurulumu yapıyor (biz Presenter'a init ile verdik). Bu veri aktarma sorumluluğu net bir şekilde Router'da tanımlıdır [70] .
5. **Storyboard/Segue Yönetimi (Varsa):** Bazı VIPER yaklaşımlarında Router, Storyboard'dan sahneyi instantiate edip içindeki ViewController'ı çıkarma işini de yapar. Örneğin modül .xib ile tasarlandıysa Router o xib'den yükleme yapıp View'i oluşturur ve diğer bileşenleri bağlar. Bu sayede ViewController içinde Storyboard referanslarına dair kod yazılmaz. Bizim örneğimiz kodla yarattığı için buna gerek duymadık.

Bu modülün bileşenleri birbirleriyle protokoller sayesinde iletişim kurar: View-Presenter, Presenter-Interactor, Presenter-Router arasında yukarıda bahsettiğimiz arayüzler kullanılır. Örneğin, KelimeDetayPresenter hem `KelimeDetayInteractorOutput` hem de `KelimeDetayViewProtocol` referanslarına sahiptir:

```
class KelimeDetayPresenter: KelimeDetayPresenterProtocol,
KelimeDetayInteractorOutput {
    weak var view: KelimeDetayViewProtocol?
    var interactor: KelimeDetayInteractorProtocol!
    var router: KelimeDetayRouterProtocol!
    // ...
    func viewYuklendi() {
        interactor.detayYukle(kelime)
    }
    func favoriButonunaBasildi() {
        interactor.favoriDegistir(kelime)
    }
    // Interactor Output
    func detayYuklendi(_ veri: KelimeDetay) {
        view?.tanimGoster(veri.tanim)
        // belki favoriDurumunuGuncelle vs.
    }
    func detayYuklenemedi(_ hata: Error) {
        view?.hataMesajiGoster("Tanım getirilemedi.")
    }
    func favoriDurumGuncellendi(yeniDurum: Bool) {
        view?.favoriDurumunuGuncelle(yeniDurum)
    }
}
```

Bu pseudo-kod, modül içi etkileşimi özetler. Görüldüğü gibi Presenter, View ve Interactor'u protokol aracılığıyla kullanıyor, Interactor da sonuçları Presenter'a protokolle döndürüyor.

Modülün Profesyonel Yapısı: Bu yapıyı Xcode'da düzenlerken, genellikle her modül için bir klasör oluşturulur ve içinde `View, Presenter, Interactor, Entity, Router, Protocols` (veya Contract) Swift dosyaları yer alır. Bu klasör modül adını taşır (ör. `KelimeDetay/`). Bu proje yapısı, modüllerin bağımsızlığını görsel olarak da ortaya koyar [71] . Ortak kullanılan Entity tanımları modül dışına konabilir (örn. `Kelime` modelini tüm modüller kullanıyorsa `Entities/` altında tanımlanır)

[72]. Aynı şekilde ortak servisler veya yardımcılar da modül dışında, örneğin `Services/` ya da `Core/` gibi bir yapıda tutulur.

Bu örnekte "Kelime Detay" modülünü ayrıntılı inceledik. VIPER'ın 5 parçasının somut bir ekranda nasıl etkileştiğini gördük. Bu yaklaşımı uygulamanın diğer kısımlarına da aynen yansıtabilirsiniz: Arama modülü benzer şekilde kendi Presenter, Interactor vb. sınıflarına sahip olacak, Favoriler modülü de öyle. Her modül kendi kullanım senaryosuna odaklandığı için, modüller arası bağımlılık minimuma iner. Sonuç: Sözlük uygulamamız VIPER ile yapılandırıldığında, her bir özelliği net sınırlarla ayrılmış, iç implementasyon detayları saklanmış ve gerektiğinde birbirine mesajlaşarak çalışan bileşenlerden oluşan profesyonel bir mimariye kavuşur.

VIPER için Profesyonel En İyi Uygulamalar ve İpuçları

VIPER mimarisini verimli ve sürdürülebilir kullanmak için, deneyimli iOS geliştiricilerinin ve yazılım mimarlarının önerdiği bazı **en iyi uygulamalar (best practices)** şunlardır:

- **Projeyi Modüler Yapıda Organize Edin:** VIPER'ın gücünden tam yararlanmak için dosya yapınızı da modüler hale getirin. Tüm VIPER modüllerini **özellik bazlı** klasörlerde tutun; her modülün kendi View, Presenter, Interactor, Router, Protocol dosyaları o klasörde olsun [73]. Örneğin `Arama`, `KelimeDetay`, `Favoriler` gibi klasörler altında ilgili MVPIR dosyaları bulunmalı. Bu sayede bir modülü komple silmek veya taşımak gerekirse, sadece o klasörü ele almanız yeterli olur [72]. Ayrıca tüm **Entity (model) sınıflarını** ortak bir yere koymak, onların modüllerden bağımsız olmasını sağlar [74]. Örneğin Kelime modeli bir klasörde ve hiçbir modüle ait değil, böylece her modül onu kullanabilir. Bu düzen, projede büyüdükçe sizi "dosya çorbası"ndan korur.

- **Katı Protokol Sözleşmeleri (Contract'lar) Kullanın:** VIPER modülünü kodlamaya başlamadan önce, o modüldeki protokolleri tanımlayın (View <-> Presenter, Presenter <-> Interactor, Presenter <-> Router, Interactor -> Presenter arayüzleri gibi) [63]. Bu **Contract.swift** dosyasında modülün bileşenlerinin hak ve sorumluluklarını belirlemek, hem geliştiricilere rehber olur hem de kod yazımını hızlandırır. Bu, aynı zamanda tasarımı düşünmeye zorladığı için, daha temiz bir etkileşim yaratır. Protokoller sayesinde, modülün iç iletişimi dışa kapalı ve net olur.

- **VIPER Bileşenlerini Otomatik Üretmeyi Düşünün:** VIPER, elle yazıldığında çok fazla tekrar şablon kod (boilerplate) içerir. Büyük projelerde bu zahmeti azaltmak için araçlar kullanmak çok yararlıdır. Örneğin **Generamba** gibi araçlar, tanımlı bir şablona göre size tek komutla bir VIPER modülünün tüm dosyalarını oluşturabilir [75]. Xcode için VIPER template'leri de mevcuttur. Bu araçlar, her seferinde aynı kod iskeletini yazma işini otomatikleştirir, hem zaman kazandırır hem de standart sağlar. Özellikle 20'den fazla ekranı olan uygulamalarda, code generator kullanmak neredeyse bir gerekliliktir [75] [35].

- **Her Katmanın Sorumluluk Sınırlarına Sadık Kalın:** Uygulama geliştirme ilerlerken bazen hızlı çözüm için kuralları esnetmek cazip gelebilir. Örneğin "Interactor'da hazır model var, Presenter'ı bypass edip direkt View'a vereyim" gibi istekler olabilir. Veya "tableView datasourcelarını Presenter halletsin" denebilir. Bu tip kısayollar, VIPER'ın getirdiği ayrım faydasını azaltır. **UIKit ile entegrasyonda** pratik olmak gerekir: ViewController hala UIKit'in gerektirdiği işleri (delegation/datasource) yapmalı, Presenter'a yüklememeli [36]. Aynı zamanda Presenter'da da UI'ye dair kod (renk, font, frame vs.) olmamalı; bunlar View'a ait. Bu sınırları korursanız, ileride kod karışmaz. Aksi takdirde VIPER modülleri de şişebilir veya sorumlulukları bulanıklaşabilir.

- **Interactor'ları Olabildiğince Bağımsız Tutun:** Interactor'larınızı yazarken, onları **UI'den tamamen soyutlayın** ve mümkünse işletim sistemi bağımlılıklarını dahi azaltın. İyi bir Interactor, teorik olarak iOS yerine komut satırında da çalıştırılabilecek iş mantığı içerir. Bu, test yazarken de size kolaylık sağlar: Interactor metodunu çalıştırıp sonucunu kontrol etmek basit bir Swift kodu olur. Interactor içinde UIKit objeleri (`UIImage`, `UIColor` gibi) kullanmaktan kaçının. Veriyi gerektiğinde Presenter'a taşıyın, orada görsele dönüştürün. Örneğin bir API'den resim URL'i geldiyse, Interactor sadece URL'i versin, UIImage'e Presenter veya View dönüştürsün. Bu tarz ayrımlar, bağımlılıkları en aza indirir.

- **Presenter'ları "Orta Seviye"de Tutun:** Presenter, fazla mantık yüklenirse yine "Massive Presenter" problemi doğabilir. Bu nedenle, veri formatlama ve basit akış kontrolü dışında ağır işlemleri Presenter'a koymayın. Eğer Presenter karmaşık hale geliyorsa, bazı parçaları Interactor'a geri itmek mümkün mü diye düşünün. Ya da Presenter'ı alt parçalara bölebilirsiniz (örn. bir Presenter ve yardımcı bir **EventHandler** veya **ViewModel** gibi alt sınıf). Gerekirse mantığı **Worker/Service** denilen ayrı sınıflara çıkarın ve Interactor'dan çağırın [76] [77]. Clean Swift mimarisinde "Worker" kavramı mesela, Interactor'un altına ekstradan bir servis görevi görür. Amaç, Presenter'ın çok şişmesini önlemek ve testlerinin karmaşıklaşmasının önüne geçmek.

- **Modüller Arası Veri Aktarımı için Delegasyon Kullanımı:** Bir modülün diğerine veri göndermesi gerekiyorsa (özellikle geri dönüş verisi için), en temiz yöntem delegasyon pattern'idir [43] [78]. Örneğin B modülü, A modülüne geri bilgi verecekse, A modülü bir protokol tanımlar (`BModuluDelegate`) ve B modülünü açarken Router aracılığıyla bu delegasyonu atar. B modülü işi bitince delegede metodu çağırır, A modülü alır. Bu sayede B, A hakkında bir şey bilmez sadece protokolü bilir. Alternatif olarak NotificationCenter veya closure kullanımları da görülebilir ancak delegasyon, tip güvenliği ve yapı açısından tercih edilir. Bununla beraber, modüller arası iletişimi mümkün mertebe minimumda tutmak en iyisidir; modüller bağımsız olmalıdır. Sıkı bağlı modüller oluşuyorsa, belki tek modül olmalılar veya aradaki ilişkileri basitleştirmek lazım.

- **Kod İnceleme ve Tutarlılık:** VIPER projesinde, belli bir modülü ilk kez yazdıktan sonra, diğer modülleri yazarken aynı kalıba sadık kalın. Tüm modüllerin benzer yapıda olması, takım içinde tutarlılık sağlar. Örneğin protokol isimlendirmeleri (`SomethingView`, `SomethingPresentation`, `SomethingInteractorInput/Output` vs.) proje genelinde uyumlu olmalı. Bazı projeler `PresenterToView` gibi isimlendirmeler kullanır, bazıları sadece `ViewProtocol`. Hangisi seçilirse seçilsin, aynı terminoloji her yerde kullanılmalı. Bu, **okunabilirliği** ciddi oranda artırır [23]. Kod incelemelerinde mimari ihlalleri veya tutarsızlıkları yakalamaya çalışın; örneğin birisi Presenter'dan direkt bir view kontrol özelliğine erişmiş mi, Interactor protokol yerine Presenter'a referans mı tutmuş gibi. Bu tür hatalar erken fark edilirse mimari bütünlük korunur.

- **Ardışık Geçişler ve Router Yönetimi:** Karmaşık navigasyon yapılarında (örn. çok adımlı kayıt sihirbazı gibi), VIPER'ın Router'larını dikkatli tasarlamak gerekir. Bazen koordinatör gibi davranmaları istenir. Eğer Presenter'da aşırı sayıda `router.showX()` çağrısı birikiyorsa, belki akışı yöneten ayrı bir Coordinator yapısı düşünebilirsiniz. Fakat VIPER ile de bu yapılabilir: Router, bir flow'un başlatıcısı olabilir ve içinde birkaç ekranın oluşturulmasını yönetebilir. Gerekirse **Router'lar arası delegasyon** ile ekranlar arası veri akışını sağlayın. Örneğin bir Router, diğer modülün Router'ını veya Presenter'ını initialize ederken bir closure/delegate verisi geçebilir.

- **Dokümantasyon ve Paylaşım:** VIPER kullanan ekipler, genellikle kendi iç best-practice dokümanlarını oluştururlar. Çünkü VIPER, belirli kurallar bütünüdür ama her ekip bunu biraz farklı yorumlayabilir. Örneğin "Configurator sınıfı kullanalım mı, yoksa Router içinde mi assembly olsun?", "Protokolleri ayrı dosyada mı tutalım modül dosyalarında mı?" gibi kararlar verilir. Bu kararları proje başında netleştirip yazılı hale getirirseniz, ekibe yeni katılanlar veya dış katkı sunanlar bu standartlara uyabilir. Zamanla bu doküman gelişir ve takımın VIPER rehberi olur.

- **Araçlar ve Ekstra Öneriler:** Xcode eklentileri (eski Xcode templates, vs), generamba dışında **Sourcery** gibi metaprogramlama araçları da boilerplate azaltmak için kullanılabilir (örneğin protokol implementasyonlarını veya Equatable vs. otomatik oluşturmak için). Hata ayıklarken VIPER modüllerini debug etmek bazen zor olabilir (çünkü call stack'te sürekli protokoller görürsünüz), bunu kolaylaştırmak için loglama stratejileri kurun (her Presenter'ın önemli olaylarda log basması gibi). Performans takibi yaparken de modüler yapı işinize yarar: Belirli bir ekran yavaşsa hangi Interactor çağrılarının uzun sürdüğünü ayrı ayrı ölçebilirsiniz.

Sonuç olarak, VIPER mimarisi güçlü fakat disiplin gerektiren bir yaklaşımdır. **En iyi uygulamaların özü**, VIPER'ın prensiplerine sadık kalmak ancak gereksiz karmaşıklıktan kaçınmaktır. Gerçek dünya uygulamalarında bazen VIPER'ı ekibin ihtiyaçlarına göre uyarlamak gerekebilir. Örneğin Uber, klasik VIPER'dan edindiği tecrübeyle RIBs adında bir türev mimari geliştirmiştir [79]. Siz de projenizde VIPER kullanırken, sürekli retrospektif yapıp "Acaba bu katmanı kullanmasak olur muydu, ya da daha iyi nasıl ayırırız?" diye düşünebilirsiniz. Bu şekilde mimarinizi evrimleştirirsiniz. Ancak temel ilkelere (modülerlik, tek sorumluluk, protokollerle iletişim) uyduğunuz sürece, VIPER size profesyonel bir iOS uygulaması için sağlam bir iskelet sunacaktır.

Kaynakça:

- Brent Simmons, **"Architecting iOS Apps with VIPER"**, *objc.io (Issue #13)* – VIPER'ın ortaya çıkışını ve temel kavramlarını anlatan klasik makale [1] [4].
- Apiumhub Tech Blog, **"Viper architecture advantages for iOS apps"** – VIPER'ın tek sorumluluk ilkesi, test edilebilirlik ve ölçeklenebilirlik konularındaki avantajlarını madde madde özetliyor [80] [81].
- Andrej Malyhin, **"The Good, The Bad and the Ugly of VIPER architecture for iOS apps."** – VIPER'ın pratikteki artılarını ve eksilerini gerçek proje deneyimiyle ele alan bir yazı [22] [82]. Özellikle büyük takımlarda modülerliğin faydalarını ve VIPER'ın UIKit paradigmalarıyla uyuşmazlıklarını vurguluyor.
- Cheesecake Labs, **"VIPER Architecture: Our Best Practices to build an app like a boss"** – VIPER kullanarak proje yapısı oluşturma, protokollerle sözleşme tanımlama, modüller arası veri iletimi gibi konularda en iyi uygulamaları paylaşan rehber niteliğinde bir makale [83] [84].
- Pedro Alvarez, **"VIPER Architecture and SOLID Principles in iOS"** – VIPER'ı Clean Architecture bağlamında açıklarken, senaryo kavramı ve modüllerin bağımsızlığı üzerine faydalı bilgiler sunuyor [21] [71]. Ayrıca VIPER'da model dönüştürme (mapper) konusuna değiniyor.
- Lemon.io Q&A, **"How does VIPER compare to MVVM for iOS projects?"** – VIPER ve MVVM'nin hangi durumlarda uygun olduğunu özetleyen karşılaştırma yazısı [31]. Özellikle projenin büyüklüğü ve karmaşıklığına göre mimari seçimine dair pratik bir bakış içeriyor.
- Radu Dan, **"VIPER vs VIP (Clean Swift)"** – VIPER'ın bir türevi olan Clean Swift mimarisiyle farklarını ve VIPER'ın artı/eksilerini sıralayan bir seri yazı [49] [85]. VIPER'ın küçük projelerde uygun olmadığını ve kod üretim araçlarının önemini not ediyor.

Bu kaynaklar ve örnek senaryolar ışığında, VIPER mimarisinin profesyonel iOS geliştiriminde nasıl verimli kullanılacağını kapsamlı şekilde ele aldık. VIPER, disiplinli uygulanırsa modülerlik, test edilebilirlik ve sürdürülebilirlik konularında önemli kazanımlar sağlayan bir mimaridir. Sözlük uygulamamız

örneğinde olduğu gibi, özelliği bol bir projede VIPER kullanmak başlangıçta biraz efor gerektirse de, uzun vadede temiz kod yapısıyla bunu fazlasıyla geri öder. **VIPER, doğru durumlarda kullanıldığında, iOS projelerinde güçlü bir mimari çerçeve sunar.** [3] [86]

---

[1] [4] [6] [8] [9] [13] [15] [16] [17] [50] [57] [58] [59] [60] [67] [68] Architecting iOS Apps with VIPER · objc.io
https://www.objc.io/issues/13-architecture/viper/

[2] [3] [5] [7] [10] [25] [26] [27] [28] [30] [80] [81] Viper architecture advantages for iOS apps | Apiumhub
https://apiumhub.com/tech-blog-barcelona/viper-architecture/

[11] [12] [18] [21] [46] [61] [62] [65] [66] [70] [71] VIPER Architecture and Solid Principles in iOS | by Pedro Alvarez | The Startup | Medium
https://medium.com/swlh/viper-architecture-and-solid-principles-in-ios-96480cfe88b9

[14] [20] [22] [23] [24] [29] [32] [33] [35] [36] [37] [42] [45] [47] [79] [82] [86] The Good, The Bad and the Ugly of VIPER architecture for iOS apps. | by Andrej Malyhin | Medium
https://medium.com/@ottobat/the-good-the-bad-and-the-ugly-of-viper-architecture-for-ios-apps-7272001b5347

[19] [34] [49] [76] [77] [85] Radu Dan - Doing magic things in Swift
https://www.radude89.com/blog/vip.html

[31] [48] How does Viper compare to MVVM for iOS projects? - Lemon.io
https://lemon.io/answers/viper/how-does-viper-compare-to-mvvm-for-ios-projects/

[38] [39] [40] [41] [55] [56] Why VIPER is a bad choice for your next application | by Sergey Petrov | Medium
https://medium.com/@Pr0Ger/why-viper-is-a-bad-choice-for-your-next-application-725f4e16fbee

[43] [51] [52] [53] [54] [63] [64] [72] [73] [74] [75] [78] [83] [84] VIPER architecture: Our best practices to build an app like a boss
https://cheesecakelabs.com/blog/best-practices-viper-architecture/

[44] Comparing MVVM and Viper architectures: When to use one or the ...
https://auth0.com/blog/compare-mvvm-and-viper-architectures/

[69] Architecture Design Patterns: VIPER | by Ashley Ng | Medium
https://medium.com/@shley_ng/architecture-design-patterns-viper-d8ade20795de