

Yazılım Tasarımı ve Mimarisi

Ders Kapsamında

- Yazılım Mimarileri
- Yazılım karmaşıklığı ve değişim (software complexity and change)
- Birliktelik ve bağımlılık (cohesion & coupling) ve anti-patternları
- SOLID, GRASP, Demeter prensipleri
- Nesne rolleri
- Yaratımsal, Yapısal ve Davranışsal kalıplar
- Null Object vb.diğer kalıplar

Amaç

- Tasarım kalıpları üzerinden yazılım tasarımı disiplini kazandırmaktır. Dolayısıyla;
 - Yazılım tasarımında soyut düşünme/modelleme,
 - Yazılım tasarımının temel problemleri, kavramları ve teknikleri,
 - Sık rastlanan yazılım problemleri üzerine düşünme,
 - Kalıpların iş alanlarındaki problemlere uygulanması,

Yazılım Krizi Sebepleri?

- Projenin zamanında tamamlanamaması
- Bütçe aşımı
- Bakım maliyetleri
- Projede hata miktarının fazla olması ve bu hataların çözülmemesi
- Yazılımın yeni gereksinimler neticesinde güncellenememesi
- Eski projelerdeki modüllerin yeni projelere entegre edilememesi

Yazılım Geliřtirmenin Özellikleri

- Yazılım ürünleri karmaşıktır.
- Takım içinde iyi bir iletişim gerektirir.
- Modüller arası uyum gerektirir.
- Yazılım sürekli deęiřeceęi için esnek olmalıdır.

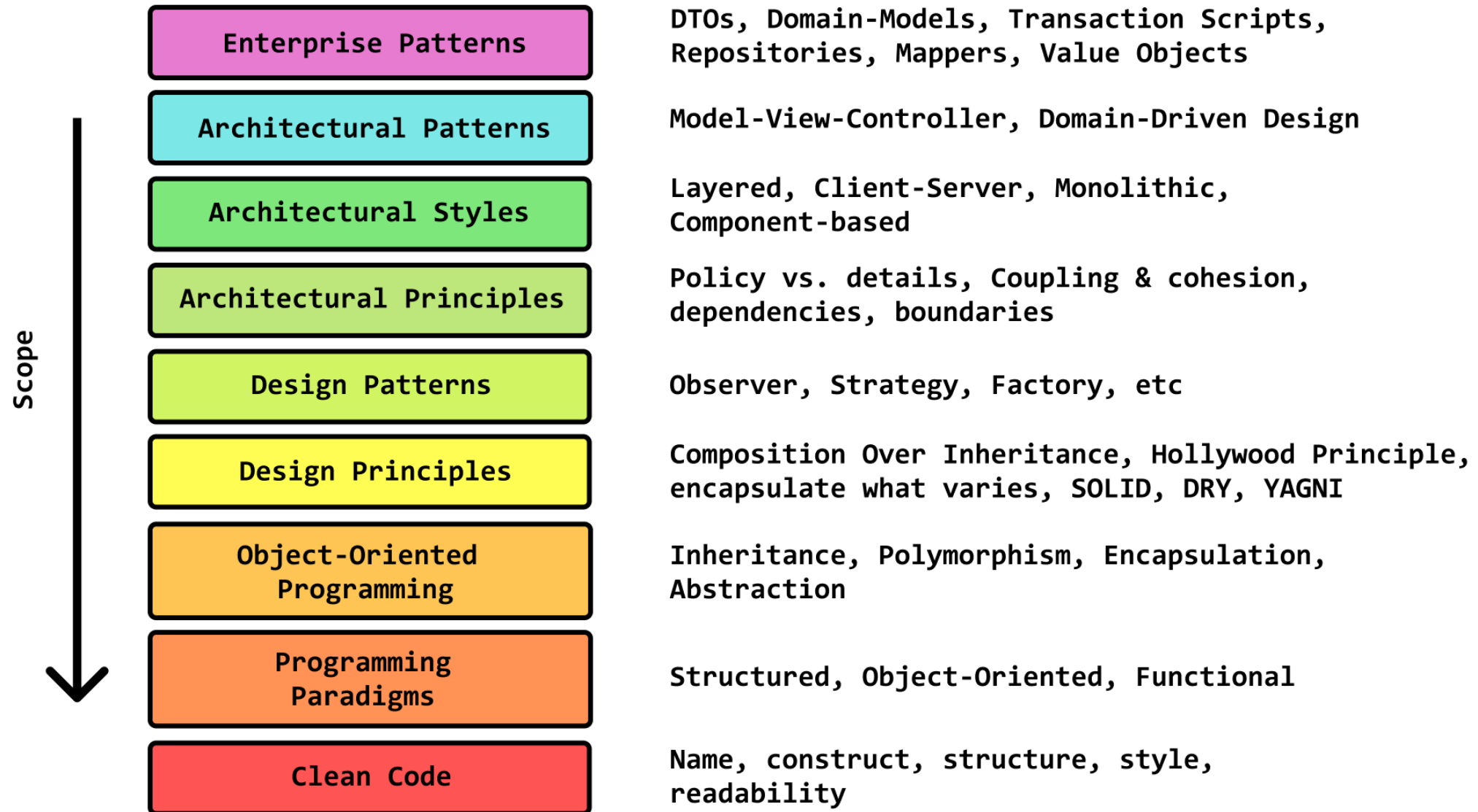
- Yazılım Geliştirme Aşamalarının Maliyetleri:
 - İsteklerin Çözümlemesi (Requirements): %3
 - Tasarım: %8
 - Kodlama: %7
 - Sınama: %15
 - Bakım: %67
- Hataların %85'i isteklerin çözümlenmesi ve tasarım aşamalarında gerçekleşmektedir.

- Hataların Giderilme Maliyeti (Belirlendikleri aşamaya göre)
 - İsteklerin Çözömlenmesi (Requirement): 1 Birim
 - Tasarım: 1.5-2 Birim
 - Kodlama: 5-10 Birim
 - Sinama: 10-15 Birim
 - Bakım: 15-100 Birim



The Software Design & Architecture Stack

Khalil Stemmler
@stemmlerjs



Yığının Akış Haritası İçin;

- <https://khalilstemmler.com/articles/software-design-architecture/full-stack-software-design/>

Yazılım Tasarımı

- Eğer bir yazılım projesinde başarıyı elde etmek istiyorsanız doğru olan yazılım mimarisini kurmak zorundasınız.
- Tasarım, Sistem Analizi çalışması sonucunda üretilen mantıksal modelin fiziksel modele dönüştürülmesidir.
- Yazılım mimarı, yazılım projesini baştan sona tasarlar, gerekli yerlerde devreye girer ve yazılım süreci boyunca tasarladığı planın dışına çıkılmamasını sağlar.
- Mimari tasarım sürecinin çıktısı bir sistemin haberleşen bileşenler olarak nasıl yapılandığını anlatan bir mimari modeldir.
 - Soyutlama
 - İyileştirme
 - Modülerlik
- Yapı, arayüz ve süreç tasarımı önce veri tasarımı yapılmalıdır. En son süreç tasarımı yapılır.

Yazılım Mimarisi

- Yazılım tasarım mimarileri, bir yazılım sisteminin bileşenlerinin nasıl organize edileceğini, nasıl iletişim kuracağını ve nasıl çalışacağını tanımlayan yapılardır. Farklı ihtiyaçlara, ölçeklenebilirliğe, performansa ve bakım kolaylığına göre çeşitli yazılım mimarileri geliştirilmiştir.
- Yazılım Mimarisi, bir bilgisayar sistemini meydana getiren yazılım bileşenleri, görünür dış özellikleri ve bunlar arasındaki ilişkilerdir. Yazılımdaki karmaşıklığın giderilebilmesi için önceleri veri yapıları kullanılırken, 1980'lerden sonra yazılım mimarisi kullanılmaya başlanmıştır.
- Mimari tasarım yazılım sisteminin nasıl yapılandırılması gerektiğini anlamak ve sistemin genel yapısını tasarlamaktır.
- İyi bir yazılım için doğru seçilmiş yazılım mimarisi gerekmektedir. Ayrıca kalite üzerinde büyük etkisi vardır.
- Bir mimari tasarım dökümanı ürüne genel bir bakış, mimari spesifikasyonlar ve tasarım gerekçesi gibi bölümlerden oluşur.

- Yazılım mimarisi sistemin güvenliğini, performansını, evrimleşebilirliğini, kullanılacak donanım gibi sonradan değiştirilmesi maliyetli olacak durumları belirler.
- Mimari tasarım genellikle ürün tasarımı sırasında başlamalıdır. Yazılım sürecindeki Tasarım bölümünün ilk aşamasıdır.

4+1 Görünüm Modeli

- 1995 yılında Philippe Kruchten tarafından geliştirilen ve karmaşık yazılım sistemlerinin mimarisini farklı paydaşların (müşteri, yazılımcı, sistem yöneticisi vb.) bakış açılarına göre organize eden bir modeldir.
- Yazılım mimarisi tek bir diyagramla anlatılamayacak kadar katmanlı olduğu için, Kruchten bu yapıyı 4 temel görünüm ve bunları birbirine bağlayan "+1" (Senaryo) görünümüyle tanımlamıştır.

1. Mantıksal boyut: Sistemin ana hatları, nesneler ve sınıflar arası ilişki. Sistem hangi nesnelerden oluşur ve bu nesneler birbirleriyle nasıl etkileşime girer?
2. İşleyişsel boyut: Sistemin runtime'daki davranışı. Performans, ölçeklenebilirlik, eşzamanlılık (concurrency) ve hata toleransı gibi "fonksiyonel olmayan" özelliklerle ilgilenir.
3. Geliştirme boyutu: Sistem hangi parçalardan oluşacağı. Yazılımın kod bazındaki organizasyonu. Modüller, paketler, kütüphaneler ve bunların arasındaki bağımlılıklar burada tanımlanır.
4. Fiziksel boyut: Hangi donanımdan oluşacağı ve hangi yazılımın hangi işlemcide çalışacağı.

+1. Senaryolar (Scenarios / Use Case View): Aslında bu 4 boyutu birbirine bağlayan "tutkal" budur. Sistemin en kritik işlevlerini (Use Case) adım adım açıklar. Diğer 4 görünümün tutarlı olup olmadığını kontrol eder.

- Gereksinim mühendisliği ile mimari tasarım süreçleri arasında ciddi örtüşme vardır.
- Gereksinim mühendisliği sürecinin bir parçası olarak, sistem işlev ve yeteneklerinin büyük çaplı bileşenler ve alt sistemlerle bağlandığı soyut bir sistem mimarisi önerilebilir.
- Sistem mimarisi üzerindeki en büyük etki fonksiyonel olmayan gereksinimlerdir.

Yazılım Mimarisinin Avantajları

- Üst seviye soyutlama ile daha anlaşılabilir olur. (Ayrıntılı tasarıma doğru soyutlanma derecesi azalır)
- Karmaşık sistemler daha iyi analiz edilir.
- Paydaşlar için iletişim ortamı sağlar.
- Gerçekleştirilecek güncellemelerin ne olacağı, nasıl yapılacağı ve etkinliği sağlanır.
- Yazılım alt bileşenlere ayrıldığı için hem işin büyüklüğü öngörülmüş olur, hem de takım içinde görev paylaşımı kolaylaşır.
- Yazılımın geliştirilebilirliği öngörülmüş olur.
- Performans, sağlamlık, dağıtılabilirlik ve bakım kolaylığı sağlar
- Yazılımın testi kolaylaşır.
- Yazılımın aşamalı teslimi kolaylaşır.
- Benzer bileşenler başka yazılımlar için de tekrar kullanılabilir.

- Programın mimari modelinin iki farklı kullanımı vardır;
- Sistem tasarımını tartışmak
- Tasarlanmış bir mimariyi belgelemek

- Çoğu büyük sistem, sistem yazılımının çok sayıda bilgisayar üzerine dağılmış olduğu dağıtık sistemlerdir.
- Yazılım sisteminin mimarisi mimari bir desene dayanıyor olabilir.
- İşlevsel olmayan gereksinimler ile yazılım mimarisi arasındaki yakın ilişki nedeniyle seçilecek stil seçimi sistemin işlevsel olmayan gereksinimlerine bağlıdır.
 - Eğer performans kritik bir gereksinim ise,
 - Eğer güvenlik kritik bir gereksinim ise,
 - Eğer emniyet kritik bir gereksinim ise,
 - Eğer erişilebilirlik ve bakım kolaylığı kritik bir gereksinim ise,

Mimari İin

- Mimari taktikler
- Mimari desenler
- Referans mimarileri
- Bařkalarının geliřtirdiđi bileřenler

Mimari Tasarım Kalıpları

- Desen Mühendisliği (Pattern Engineering) sistematik ve disiplinli bir şekilde verilen kısıt ve sağlanan kaynaklarla kalıpları tespit etme, kullanma, bakım yapma yaklaşımıdır.
- Hangi kalıbın ne zaman kullanılacağını proje kısıtları ve sağlanan kaynaklar belirler. Kalıp mühendisliği de yazılım mühendisliğinde hem tasarım aşamasında tasarım kalıpları olarak hem de mimari tasarımda mimari kalıplar olarak kendini göstermektedir.

Sık Kullanılan Mimari Modeller

- Monolitik Mimari
- Katmanlı Mimari
- Boru (Kanal) & Filtre Mimarisi
- İstemci - Sunucu Mimarisi
- Yinelemeli İçerme Mimarisi
- Bileşen Tabanlı (Ambar/Depo) Mimarisi
- Model - View - Controller
- Servis Odaklı Mimari
- Microservis Mimarisi
- Olay Tabanlı Mimari (Event-Driven Architecture)
- Hexagonal Mimari

Monolitik Mimari

- Monolitik bir uygulama; kullanıcı arayüzü (UI), iş mantığı (Business Logic) ve veri erişim katmanlarının (Data Access) tek bir platformda, tek bir kod tabanında (codebase) birleştiği ve tek bir birim (single deployable unit) olarak yayınlandığı yapıdır.
- Tüm bileşenler aynı bellek alanını ve genellikle aynı veritabanı şemasını kullanır.
- Sistem zorlandığında tek çözüm sunucunun RAM veya CPU kapasitesini artırmaktır.
- Küçük bir CSS değişikliği bile yapsanız, tüm uygulamayı yeniden derleyip sunucuya yüklemeniz gerekir.

- Avantajları;
 - **Geliştirme Kolaylığı:** Başlangıçta çok daha hızlıdır. Tek bir proje dosyası, tek bir IDE penceresi.
 - **Test Edilebilirlik:** Uçtan uca (End-to-End) testler çok basittir çünkü tüm bağımlılıklar elinizin altındadır.
 - **Performans:** Bileşenler arası iletişim bellek üzerinden yapıldığı için, mikroservislerdeki ağ gecikmesi burada yoktur.
 - Tek bir dosya, tek bir sunucu vardır.

- Dezavantajları;
 - **Bakım Zorluğu:** Kod tabanı milyonlarca satıra ulaştığında, bir yeri değiştirmek beklenmedik yan etkilere neden olur.
 - **Teknolojiye Bağımlılık:** Uygulamanın bir kısmını Java'dan Python'a geçirmek isterseniz, tüm sistemi yeniden yazmanız gerekir.
 - **Hata Toleransı Düşüklüğü:** Bellek sızıntısı yapan küçük bir modül, tüm sistemi (örneğin koca bir bankacılık uygulamasını) çökertebilir.
 - **Yavaş Dağıtım:** Derleme ve test süreçleri saatler sürmeye başlar, bu da Çevik yöntemleri baltalar.

Katmanlı Mimari

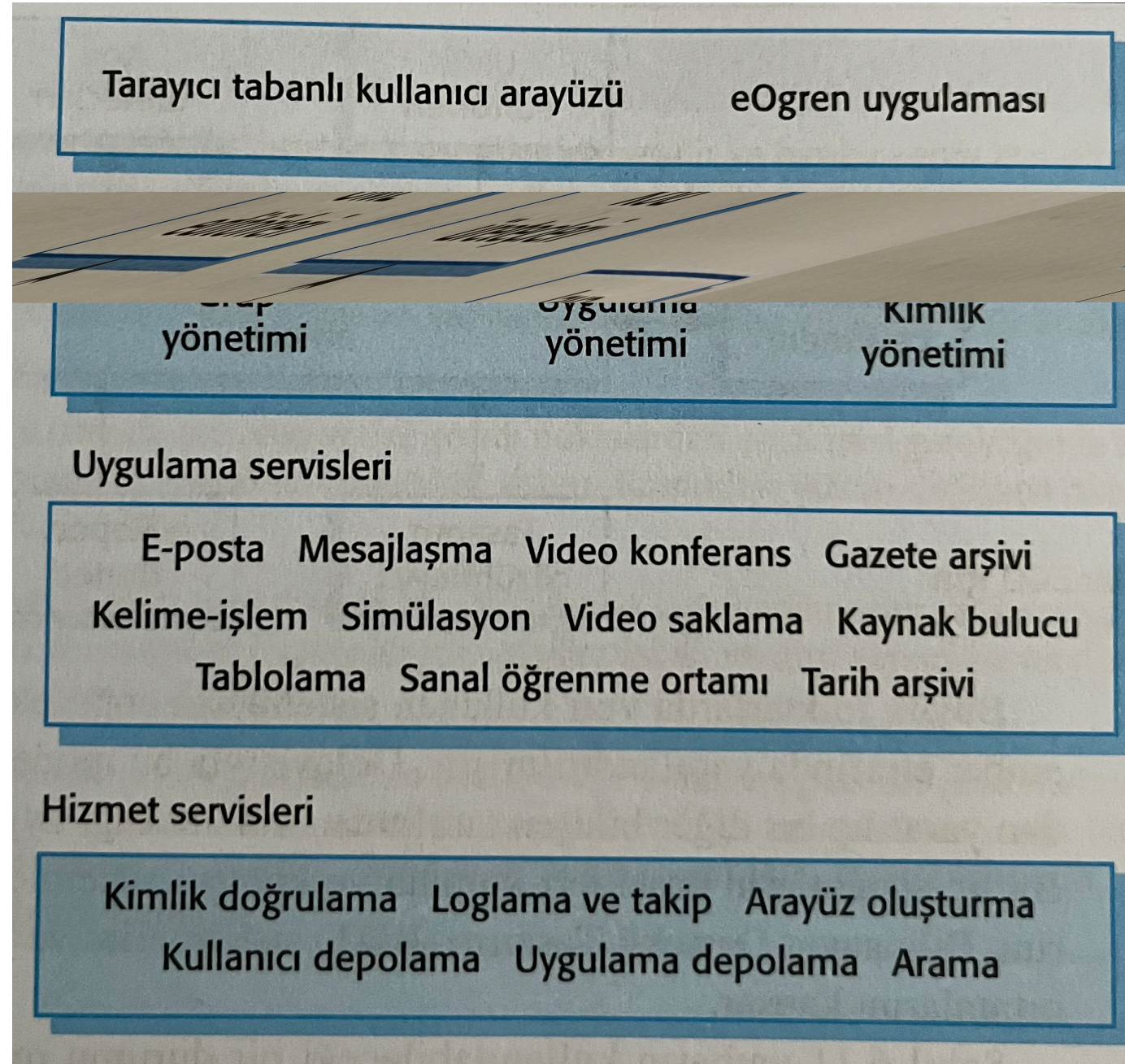
- N-katmanlı (n-tier) mimari olarak da bilinir. “Sorumlulukların Ayrılması” ilkesine dayanır. Ayırıştırma ve bağımsızlaştırmanın bir başka yoludur.
- Her katman farklı bir fonksiyonu yerine getirir ve genel olarak kendisinden bir alt ve bir üst katman ile iletişim kurar. Her katman bir üst katmana hizmet eder.
- Katmanlı mimari kalıbında eğer bir katman sadece bir alt katmandan servis alabiliyor ise bu tür sistemlere kapalı katmanlı sistem denir. Bu ideal tasarımdır. Performans gibi kaygılardan dolayı bir katman kendinden iki ve daha fazla altta olan katmanlara da erişiyor ise buna açık katmanlı sistem denmektedir.
- Kolay geliştirilip test edilebilir fakat performans, çeviklik ve genişleyebilme açısından kısıtlıdır.

- Sistemlerin artırımı geliştirilmesini destekler.
- Mimari aynı zamanda değiştirilebilir ve taşınabilir yapıdadır.
- Bir mimari katman değiştiğinde sadece komşu katmanlar bundan etkilenir.

Katmanlar

- **Sunum Katmanı (Presentation Layer):** Kullanıcı arayüzünü içerir. Web arayüzü, mobil ekran veya bir API endpoint'i bu katmandadır.
- **Uygulama/Servis Katmanı (Application Layer):** İş mantığının koordinasyonunu sağlar. "Hangi iş mantığı ne zaman çalışacak?" sorusuna yanıt verir.
- **İş Mantığı Katmanı (Business Logic Layer):** İş kurallarını ve veri işleme mantığını barındırır. Hesaplamalar, kurallar ve iş süreçleri burada tanımlanır.
- **Veri Erişim Katmanı (Data Access Layer):** Veritabanı işlemleri gerçekleştirilir.
- **Veri Katmanı (Database Layer):** Verilerin depolandığı ve yönetildiği katmandır.

Eogren Sistem Mimarisi



- Avantajları;
 - Arayüz korunduğu sürece katmanların bütün olarak değiştirilebilmesini sağlar.
 - Sistemin güvenilirliğini artırmak için her bir katmanda yedeklenmiş imkanlar sunulabilir.
- Dezavantajları;
 - Pratikte katmanlar arasında temiz bir ayrışma genellikle mümkün olmaz.
 - Üst düzeyli bir katman alt katmanın aracılığı olmaksızın daha alt katmanla etkileşmek zorunda kalabilir.
 - Bir servis talebinin her bir katmanda tekrar yorumlanması performansı düşürebilir.

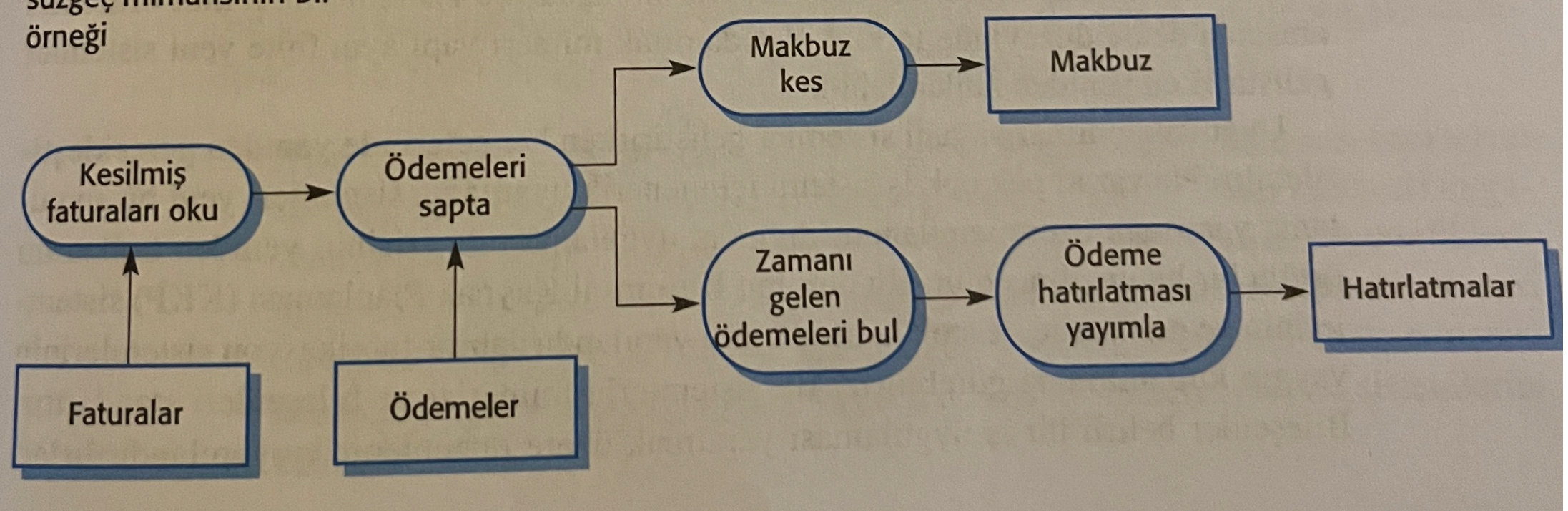
Kanal ve Filtre (Boru ve Süzgeç) Mimarisi

- Veri işleme için yaygın kullanılan bir mimaridir. Sisteme giren veri, Filtre dediğimiz veri üzerinde işlem yapan bileşenler ve bu bileşenler arasında veriyi aktaran kanallardan oluşur.
- Güçlü bir mimaridir.
- Bu mimaride ayrı ayrı filtreleme bileşenleri bulunur. Filtreler arasında da yerel bağlantı bulunur. Bu bağlantılara pipe denir.
- Her filtre kendisine gelen veriyi işler ve sıradaki filtreye aktarır.
- Filtreler verileri zenginleştirebilir, detayları gizleyebilir, başka bir gösterime dönüştürebilir.
- Farklı işlemler farklı filtreler halinde geliştirilerek sistem daha modüler hale gelebilir. Ayrıca filtrelerin farklı amaçlarla kullanımı ile yeniden kullanılabilirliği artar.

- Sistem içindeki veri işleme düzeni öyle düzenlenmiştir ki her bir işlemci bileşeni (süzgeç) ayrıktır ve tek bir veri dönüşümü gerçekleştirir.
- Veriler, bitişik filtreler arasındaki bağlayıcılardan geçirilir. Tüm filtreler aynı anda işler. Bu sayede farklı iş parçacıkları (threads) veya eşyordamlar (coroutines) aynı anda çalıştırılabilir veya filtreler farklı makinelerde yer alabilir. İşlemin sonunda veriler, sink adı verilen bir veri hedefine aktarılmış olur.
- Sınırlı kullanıcı etkileşimine sahip ve ya gömülü sistemler için uygun bir mimaridir.
- Görüntü işleme ve derleyiciler de bu mimari modelinin iyi bilinen ve yaygın olarak kullanılan alanlarıdır.

- Avantajları;
 - Anlaşılması kolaydır.
 - Dönüşümlerin yeniden kullanımını destekler.
 - Bir çok iş sürecine uyar.
 - Sıralı ya da eş zamanlı çalışabilir böylece hızlıdır.
 - Aynı anda çalışan yeni filtre ve bağlayıcıların kolayca eklenebilir ve böylece hızlı bir şekilde büyütülebilir.
- Dezavantajları;
 - Haberleşen sistemlerde veri transferi karara bağlanmalıdır.
 - Her bir dönüşüm iyi planlanmalıdır.
 - Sistem maliyeti artar.
 - Uyumsuz veri yapılarını kullanan mimari bileşenleri kullanmak imkansızdır.

Süzgeç mimarisinin bir
örneği



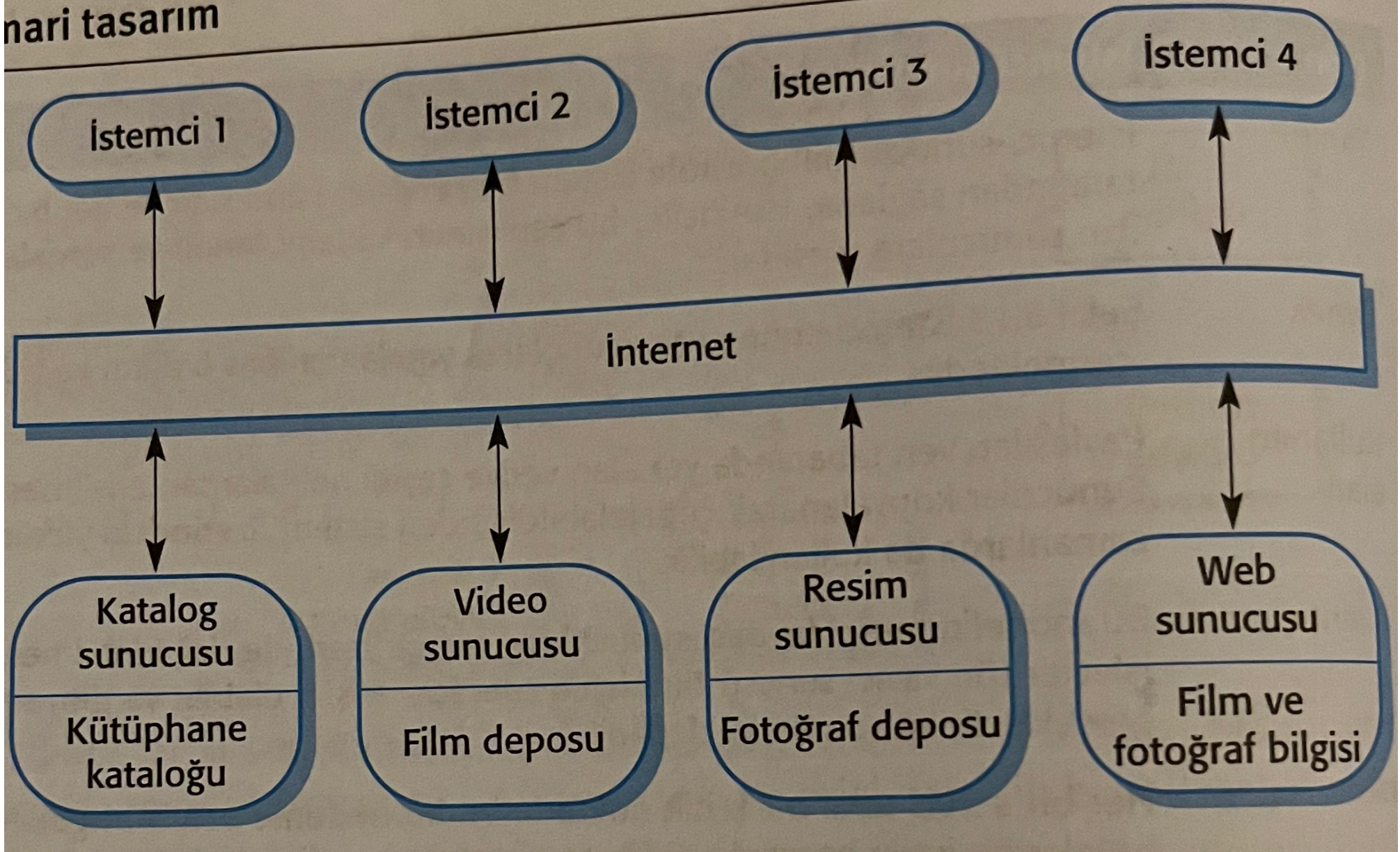
- Lamda mimarisi kanal ve filtre mimarisinin özelleşmiş şeklidir. Bu mimaride kaynaktaki veri iki farklı hattan ilerlemektedir. Gerçek zamanlı ihtiyacı karşılamak üzere hız katmanı üzerinde veri işlenir iken daha detaylı analize olanak verecek şekilde yığın katmanı üzerinden işlenmektedir. Her iki kanalın çıktıları "Serving Layer"da birleşerek kullanıcıya sunulur.

İstemci Sunucu Mimari

- İstemci isteği yapan, sunucu ise isteği karşılayan kısımdır.
- Sunucu, istemciden gelen isteği kabul ettiğinde, istemciyle belirli bir protokol üzerinden bağlantı kurar.
- Sunucu istemciden gelen isteklerin kaydını tutuyorsa buna oturum denir.
- Sistem bir servisler kümesidir. Her bir servis ayrı bir sunucu tarafından sağlanır. İstemciler bu servislerin kullanıcılarıdır ve servislerden yararlanabilmek için sunuculara ulaşırlar.
- Paylaşılan veritabanı farklı lokasyonlardaysa, farklı istemciler tarafından ulaşılacak isteniyorsa ve sistem üzerindeki yük değişkense tercih edilir.
- Esas paylaşılan servis sunucuda yer alır ve bu sunucudan servis alacak istemciler istekte bulunarak sunucudan servis alırlar. İstemci ayrı çalışan bir uygulama olabileceği gibi WEB üzerinde hizmet alan bir web uygulaması da olabilir.
- İnce istemci ve Kalın istemci yaklaşımları vardır.

- Avantajları;
 - Dağıtık bir mimaridir. Sunucunun sunduğu servis tüm istemciler tarafından kullanılabilir.
 - Servisler bağımsız ve birbirinden ayrıdır. Sistemin geri kalanını etkilemeden sunucular güncellenebilir.
 - Genel işlevler (yazıcı servisi) tüm istemciler için erişilebilir olabilir ve tüm sunucular tarafından gerçekleştirilmesi gerekli değildir.
- Dezavantajları;
 - Her bir servis tekil bir kritik noktadır, bu nedenle servis engelleme ve sunucu hatalarına karşı hassastır.
 - Ağa bağlı oldu için performansı kestirmek zordur.
 - Sunucular farklı kurumlara ait olduğunda yönetilmeleri problemli olabilir

nari tasarım



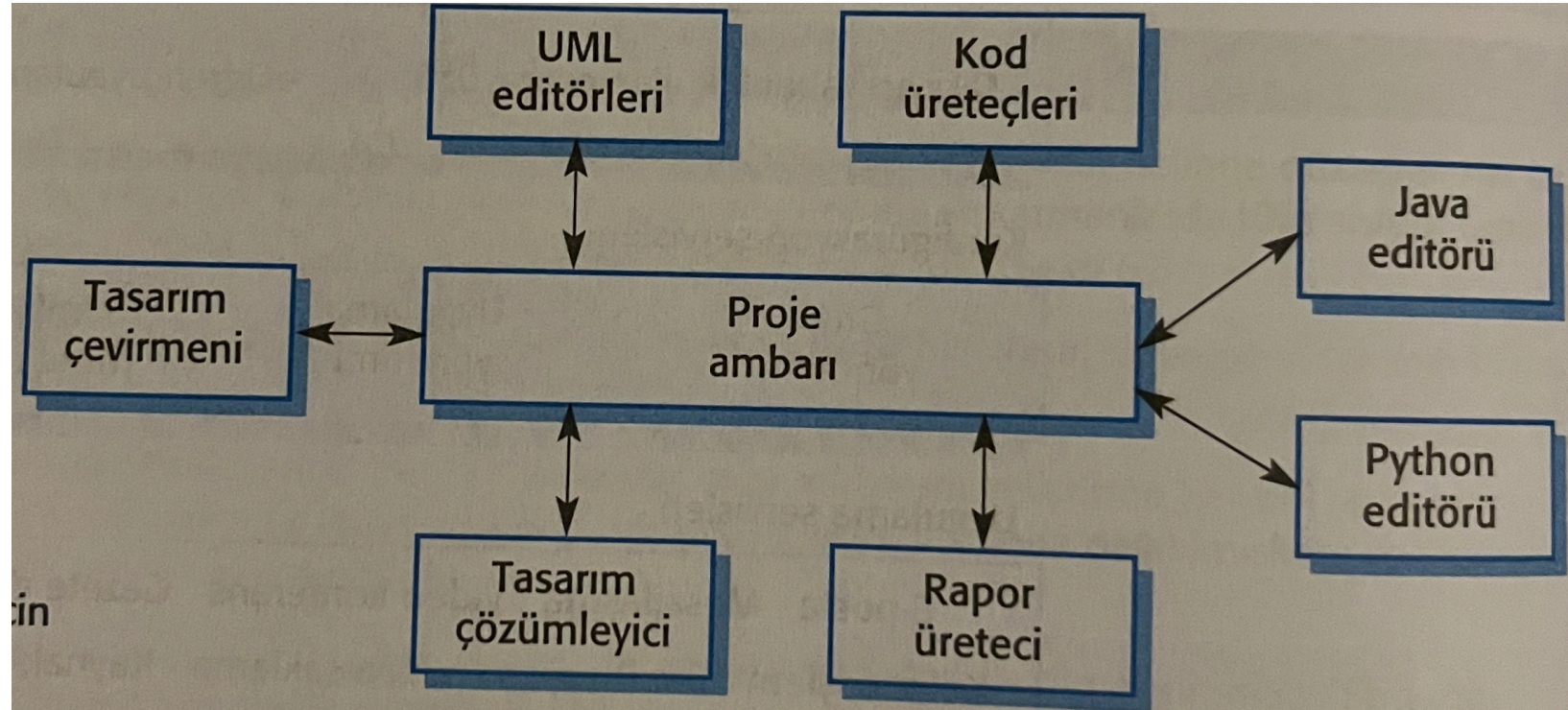
Yinelemeli İçerme Tasarım Kalıbı

- Composite (Bileşik) Tasarım Kalıbı olarak adlandırılan yapının temel mantığını oluşturur.
- Büyük bir sistemi alt sistemlere oradan da daha alt bileşenlere bölecek şekilde tasarlar. Burada her bir sistem kendi alt parçalarını içerir. Bu kalıp aynı zamanda Bütün Parça tasarım kalıbı ya da Birleşik tasarım kalıbı olarak bilinmektedir. Örneğin bir çizim şekillerden oluşur.

Bileşen Tabanlı (Ambar) Mimari

- Sistem küçük bileşenler ve bu bileşenleri bir arada tutan bileşen yöneticisinden meydana gelir.
- Tüm veriler, tüm sistem bileşenlerinin erişebildiği merkezi bir havuzda yönetilir.
- Uzun süre saklanması gereken büyük miktarda veri olduğunda kullanılır.
- Yazılım sistemi yeniden kullanılabilir, birleştirilebilir ve paketlenmiş bileşenlerden oluşur. JavaBean, .NET, WEB Servisleri vb. gibi. Bağlantılar bileşenlerin diğer bileşenlerle nasıl iletişim kuracağını belirler.
- Bileşen etkileşimi metot çağırma, uzaktan metot çağırma, yayımlama, mesaj tabanlı iletişim ve benzeri protokoller aracılığı ile yapılır.
- Yeniden kullanılabilirlik, test edilebilirlik ve genişleyebilirlik açısından önemli avantajlar sağlamaktadır.

- Avantajları;
 - Bileşenler bağımsızdır.
 - Bir bileşen tarafından yapılan değişiklik diğer bileşenlere yansıtılabilir.
 - Tüm veri aynı yerdeymişçesine tutarlı biçimde yönetilir. (Yedeklemeler aynı anda yapılır)
- Dezavantajları;
 - Ambardaki tek bir hata tüm sistemi etkiler.
 - İletişimin ambar üzerinden yapılması performansı düşürür.
 - Ambarı çok sayıda bilgisayara dağıtmak zordur.



- Hexagonal mimarideki her bir "Adaptör" aslında bir bileşen olarak düşünülebilir. Her ikisi de iş mantığını dış detaylardan arayüzler (portlar) kullanarak ayırır.
- Mikroservisler, bileşen tabanlı mimarinin dağıtık sistemlere uygulanmış bir evrimidir. Her mikroservis, kendi içinde bağımsız bir bileşendir.

Model Sunum Kontrolcü (Model Görünüm Kumanda - Model View Controller-MVC) Mimarisi

- En çok bilinen ve sık sık kullanılan mimaridir.
- İş mantığını ve verileri sunum detaylarından ayırarak bağımsız olarak değişebilmelerine olanak tanır.
- Kullanıcı arayüzü tabanlı yazılımlarda çok yaygın olarak kullanılır.
- Sunum ve etkileşimi sistem verisinden alır ve ayrı katman olarak değerlendirir.
- Pek çok Web tabanlı sistemdeki etkileşim yönetiminin tabanını oluşturur.
- Temel amaç veri ve kullanıcı arayüzünü birbirinden ayırmaktır. Veri ile kullanıcı ara yüzü arasındaki aracılığı kontrolcü sağlar. ASP .Net MVC, Rubby on Rails, Django gibi kütüphaneler yapılarını bu kalıp üzerine kurmuşlardır.

- Model bileşeni sistem verilerini ve o verilerle bağlantılı işleri yönetir. En düşük seviye bileşendir. Temel fonksiyonlar yer alır.
- Görünüm bileşeni verinin kullanıcıya nasıl sunulacağını tanımlar ve yönetir.
- Kumanda bileşeni kullanıcı etkileşimini yönetir. Model ve görünüm arasındaki bağlantıyı sağlar. Kullanıcı eylemlerini (butonclick) alır, işler, görünüm ve modele iletir ve bunlara yanıt verir. Kullanıcının isteklerini gerçekleştirir.

- Avantajları;
 - Verinin ve gösteriminin birbirinden bağımsız olarak değişebilmesine imkan sağlar.
 - Üç bileşen ayrı olarak tasarlanıp kullanıldığı için kullanımı kolaydır.
 - Aynı verinin farklı sunuş şekillerini birlikte destekleyerek birinde yapılan değişiklik aynı anda hepsinde gözlenir.
- Dezavantajları;
 - Veri modeli ve etkileşimler basit olsa bile ek koda ihtiyaç duyulur.
 - Mimarideki karmaşıklık yüksektir.

ServisTabanlı Mimari

- Değiştirilmesi zor olan eski monolitik bağımsız uygulamalardan, iş hedeflerine ulaşmak için birlikte çalışmak üzere birleştirilen yeniden kullanılabilir kurumsal bir yapıya geçiş için mimari bir yaklaşımdır.
- Burada farklı kabiliyetler servis olarak geliştirilmekte ve istemcilerin istekleri bu servislere iletildiğinde ilgili servis diğer servislerden tanımlı arayüzler üzerinden hizmet olarak istemcinin ihtiyacını karşılamaktadır.

- Amaç, büyük monolitik uygulamaları yeniden kullanılabilir ve birlikte çalışabilir bağımsız servislere ayrıştırmak ve standart protokoller üzerinden iletişim kurmalarını sağlamaktır.

Servis tabanlı mimari için servisin temel yapısı;

- Service Provider (Servis Sağlayıcı)
- Service Requestor (Servis Kullanıcısı)
- Service Registry (Servis Kaydı)
- Publish (Yayınla)
- Find (Bul)
- Bind (Bağla)

Temel Bileşenleri

- Application Frontend (Uygulama Önyüzü)
- Service (Servis)
- Contract (Anlaşma)
- Interface (Arayüz)
- Implementation (Uygulama)
- Service Repository (Servis Deposu)
- Service Bus(Servis Veri Yolu)

Mikro Servis Mimarisi

- Mikro servisler, SOA'nın daha küçük, daha bağımsız ve modern teknolojilerle (Docker, Cloud) harmanlanmış bir evrimidir.
- Birden fazla mimari tasarım kalıbını bir arada kullanan bir referans mimaridir.
- Sistem birbiri ile iletişim halinde olan servislerden oluşur. Bu servisler HTTP/Rest çağrıları ile birbirleri ile doğrudan konuşabilirler ya da AMQP gibi yayımla abone ol tabanlı ara katmanlar üzerinden iletişim sağlarlar.
- Servislerin mikro olmasından çok anlamlı ve tek bir iş mantığı sağlıyor olmasıdır. Aynı zamanda yönetilebilir küçük bir takım tarafından geliştirilebilir olması önem verilen bir noktadır.
- Temelde bileşenlere ayırma kalıbı olarak iki yaklaşım önerilmektedir (Kabiliyete göre ya da alt domainlere göre).
- İletişim tasarım kalıpları olarak mesajlaşma, uzaktan metot çağırma ve alana özel haberleşme kalıpları bulunmaktadır.

- Dağıtık mimarinin temel faydalarından faydalanmak üzere gelişmiştir.
- Servisler birbirinden bağımsız şekilde yüklenebilir.
- Microservisler stateless'dir. Herhangi bir nesnenin state'ini tutmaz.
- Bir servis istenmeden kapandığında diğer servisleri etkilemez.
- Kritik servisler birden fazla kopya çalıştırılarak genişleyebilen yapı sağlanır.
- Projenin ileri ki aşamalarında geliştirilecek yeni servisler tamamen farklı bir programlama dili ve kütüphane seti ile geliştirilebilir.
- Çevik yazılım süreçleri, kurulum yöneticisi, sürekli teslimat yaklaşımlarını birleştiren bir akımdır.

- Mimari en baştan kurulmak zorunda değildir, ürün geliştikçe mimari şekillenir.
- Projeye sonradan dahil olan kişi bile kendi bildiği dil ve DB ile projeye destek olabilir çünkü üzerine düşen projenin ufak iş parçacığıdır.
- Bu ufak iş parçacıklarının sırasını ve hiyerarşisini değiştirmek kolaydır.
- Uygulama esnek ve tekrar kullanılabilir.
- Ölçeklenebilir.

- Dikkat edilmesi gereken husus olarak;
 - Kubernetes, Service Mesh veya karmaşık dağıtık sistem izleme araçlarına ihtiyaç duyar.
 - Sistemi bileşenlere bölündüğünde bileşenlerin yönetilmesi ve bileşenler arası veri paylaşımının sağlanması gerekir.
 - Bileşenler arası bağımlılıklar ve bileşenlerden birisinin kapanması durumunda sistemin davranışının tutarlı olması sağlanmalıdır.
 - Tüm servislerin bir bütün olarak uyumlu çalışabilirliğinin test edilmesi gereklidir.

- Servis Tabanlı Mimari ile Mikroservis Mimarisi çok benzer yaklaşımlardır. İkisi de aynı veya benzer tasarım prensiplerine sahiptir ve client/server yaklaşımından geliştirilmiştir.
- İkisi de geleneksel, monolitik mimariden farklıdır çünkü her servisin kendi sorumluluğu vardır. Ayrıca büyük ve karmaşık uygulamaları, birlikte çalışması daha kolay olan küçük, esnek bileşenlere böler.
- Mikro servisin servis tabanlı mimariden farkı, servis tabanlı mimaride servislerin bir arada yer alması sebebiyle büyük ve hantal sistemler oluşturur. Sadece servis odaklı mimariden ayırtırmak için adı mikro servis olarak verilmiştir. Servis odaklı mimaride veri ortak bir veri tabanında tutulur iken, mikro serviste her servisin kendi veri tabanını kullanabilmesi öngörülmektedir.

Olay Tabanlı Mimari (Event Driven Architecture)

- Olay tabanlı mimari (EDA), sistem bileşenlerinin olaylar yoluyla birbirleriyle haberleştiği ve senkron ya da asenkron olarak tepki verdiği bir yazılım tasarım modelidir. Geleneksel **istek-yanıt (request-response)** modelinden farklı olarak, **olaylar** sistemin temel itici gücü olarak kullanılır.
- Mediator (Küçük veya monolitik projelerde) ve Broker (büyük ve dağıtık sistemlerde) olarak tasarlanabilir. Broker; uygulamalar arası, farklı sunuculardaki sistemleri bağlar.
- Olaylar birbirinden bağımsız bileşenleri (mikro servisler veya modüller) tetikleyerek sistemin belirli parçalarının çalışmasını sağlar.

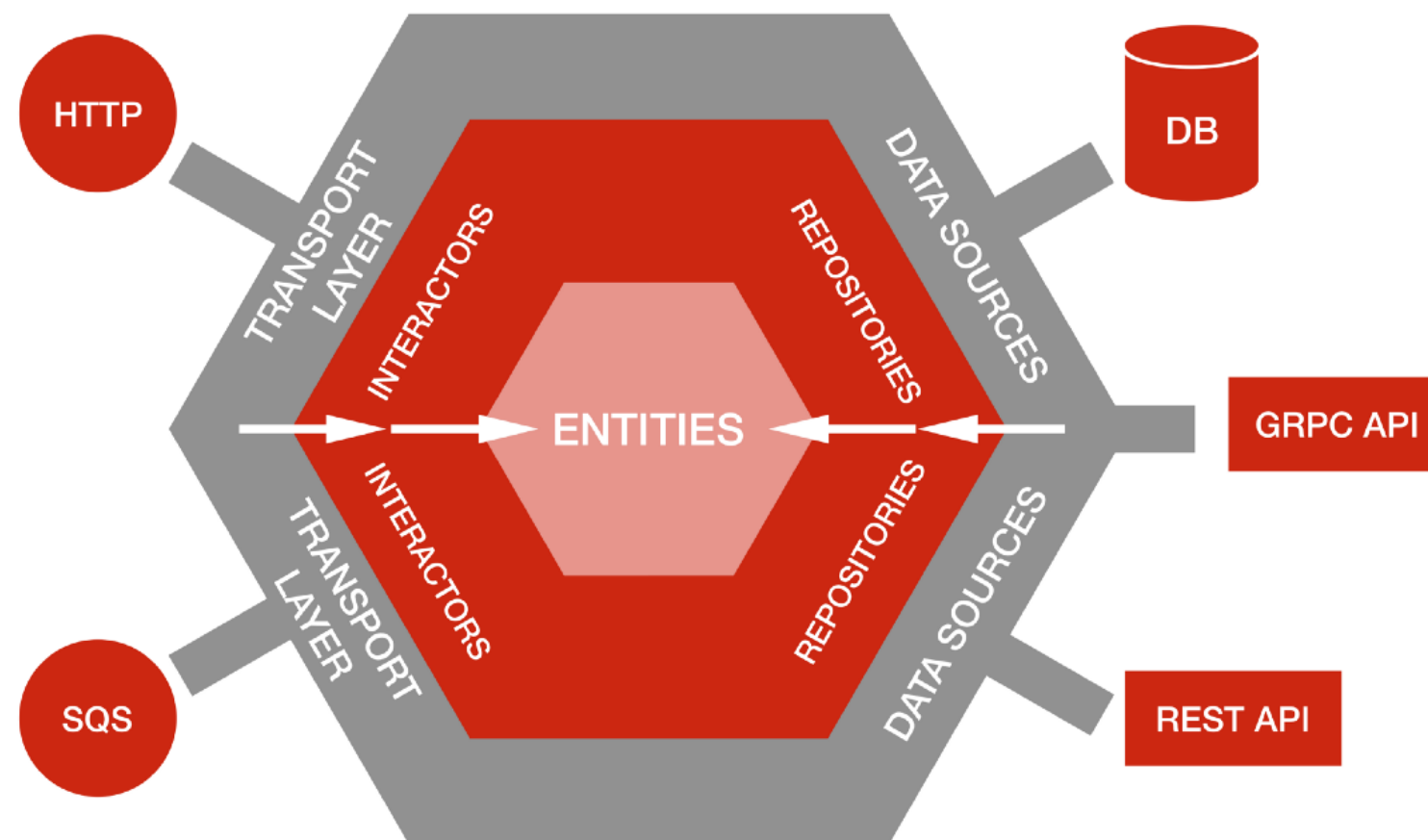
- Olay tabanlı mimaride iki temel bileşen vardır:
- **Olay Üreticileri (Event Producers):** Olayları üreten ve yayınlayan bileşenlerdir.
- **Olay Tüketicileri (Event Consumers):** Yayınlanan olayları dinleyen ve onlara tepki veren bileşenlerdir.
- Bu bileşenler doğrudan birbirleriyle iletişim kurmaz, mesaj kuyrukları (message queues) veya olay aracısı (Event broker-Event Bus) üzerinden haberleşirler.

- Avantajları:
 - Loose Coupling: Bileşenler birbirine doğrudan bağımlı değildir, sadece olaylar aracılığıyla haberleşir. Bu, sistemi daha modüler ve esnek hale getirir.
 - Ölçeklenebilirlik: Mikro hizmetlerin bağımsız çalışabilmesi sayesinde yük dengeleme (load balancing) ve yatay ölçeklenme (horizontal scaling) daha kolaydır.
 - Gerçek Zamanlı İşleme: Gerçek zamanlı veri işleme gerektiren uygulamalar (örneğin finans, IoT, telemetri sistemleri) için idealdir.
 - Yüksek Uygunluk (High Availability): Bir bileşenin çökmesi durumunda, olay kuyruğunda bulunan mesajlar sistem iyileştikten sonra tekrar işlenebilir.
 - Esnek Genişleme: Yeni tüketici servisleri sisteme kolayca entegre edilebilir, böylece yeni özellikler eklemek kolaylaşır.

- Dezavantajları:
 - Artan Sistem Karmaşıklığı: Olaylar aracılığıyla haberleşen bileşenler nedeniyle, hata ayıklama ve sistem izleme zorlaşabilir.
 - İletişim Yönetimi: Senaryo büyüdükçe, olaylar arasındaki ilişkiyi yönetmek zorlaşır. Olay sırasının korunması ve olayların kaybolmaması için ek önlemler alınmalıdır.
 - Veri Tutarlılığı (Eventual Consistency): Tüm servisler birbirinden bağımsız çalıştığı için, gerçek zamanlı veri tutarlılığı sağlamak zor olabilir. Örneğin, bir siparişin onaylanması ile ödeme işleminin tamamlanması arasında geçen sürede farklı durumlar oluşabilir.

Hexagonal Mimari

- Yazılımın merkezine "İş Mantığını" (Domain) koyan ve onu dış dünyadan tamamen izole eden bir mimari stildir.
- Hexagonal mimaride temel amaç, uygulamanın iş mantığının; kullanıcı arayüzünden, veritabanından, harici servislerden veya test araçlarından bağımsız olmasını sağlamaktır.



- Entitiy'ler kurumsal kuralları, use case'ler uygulama kurallarını belirtir.
- Interface Adapter, en dıştaki yapı ile use case'ler arasında bağlantı kurar ve gerekli dönüşümleri yapar.

- Hexagonal mimari, bir uygulamanın kolay geliştirilebilmesine ve kolay deploy (yazılımı yaymak, dağıtmak, sahaya yerleştirmek) edilebilmesine, bakım maliyeti az olan sistemlerin tasarlanmasına, eğer ekip olarak bir ürün üzerinde çalışıyorsanız aynı perspektif ile ürüne yaklaşmanıza olanak sağlayan bir mimari modeldir.
- Bu mimari daha çok gerçek hayatta karşılığı olan kompleks iş gereksinimlerinin ve iş kurallarının olduğu domain 'ler için tercih edilmektedir.

- Klasik katmanlı mimaride iş mantığı katmanı, fiziksel olarak veritabanı katmanına bağımlıdır. Hexagonal mimaride ise bu durum Dependency Inversion prensibi ile tersine çevrilir:
- Bağımlılıklar her zaman dışarıdan içeriye (çekirdeğe) doğrudur.

- Avantajları;
 - **Yüksek Test Edilebilirlik:** Veritabanına veya arayüze ihtiyaç duymadan, Portları "Mock"layarak (sahteleyerek) tüm iş mantığını saniyeler içinde test edebilirsiniz.
 - **Gevşek Bağımlılık (Loose Coupling):** Sistem parçaları birbirine "Boru (Kanal) & Filtre" mantığına benzer bir esneklikle eklemlenir, ancak veri akışı merkezi bir iş mantığı üzerinden yönetilir.
 - **Teknoloji Eskimesine Karşı Direnç:** UI veya DB teknolojileri eskidiğinde, uygulamanın kalbine dokunmadan sadece dış çeperdeki adaptörleri yenilersiniz.

Kaynak

- <https://khalilstemmler.com/articles/software-design-architecture/full-stack-software-design/>
- www.akademi.itu.edu.tr/buzluca
- <https://www.horato.com/tr/post>
- www.wikipedia.org
- www.medium.com
- www.arakatman.com
- Ian Sommerville, Yazılım Mühendisliği
- <https://blog.finartz.com/hexagonal-mimari-eb09b5feb245>