İSTANBUL MEDENİYET ÜNİVERSİTESİ

2025-2026 AKADEMİK YILI GÜZ DÖNEMİ

BİL403 YAZILIM MÜHENDİSLİĞİ

6. DERS

Dr. Öğr. Üyesi Ertürk ERDAĞI
erturk.erdagi@medeniyet.edu.tr

Yazılım Tasarımı ve Mimarisi

- Yazılım mühendisliğinde **gereksinimler belirlendikten sonra**, bu gereksinimlerin **nasıl gerçekleştirileceği** sorusu ortaya çıkar.
- Bu aşamada iki temel kavram devreye girer:
  - **Yazılım Tasarımı (Software Design)**
  - **Yazılım Mimarisi (Software Architecture)**
- Bu ikisi birlikte, yazılımın **iskeletini ve detaylarını** oluşturur. Yani mimari “binanın planı”, tasarım ise “odaların iç düzeni” gibidir.

Yazılım Tasarımı (Software Design)

- Yazılım tasarımı, bir yazılım sisteminin **nasıl çalışacağını planlama sürecidir.** Gereksinim analizinde "ne yapılacağı" tanımlanır; yazılım tasarımında ise **"nasıl yapılacağı"** belirlenir. Yani;
  - Yazılım tasarımı = fikirlerin koda dönüşmeden önce mühendislik planı haline getirilmesidir.
- Amaçları
  - Sistemi parçalara ayırmak (modülerlik sağlamak)
  - Her parçanın sorumluluğunu netleştirmek
  - Veri yapısı, algoritma, arayüz ve bileşen etkileşimlerini belirlemek
  - Kodlamadan önce sistemi daha anlaşılır hale getirmek

Yazılım Tasarımı (Software Design)

- Yazılım tasarımı, bir yazılım sisteminin **nasıl çalışacağını planlama sürecidir.** Gereksinim analizinde "ne yapılacağı" tanımlanır; yazılım tasarımında ise **"nasıl yapılacağı"** belirlenir. Yani;
  - Yazılım tasarımı = fikirlerin koda dönüşmeden önce mühendislik planı haline getirilmesidir.
- Amaçları
  - Sistemi parçalara ayırmak (modülerlik sağlamak)
  - Her parçanın sorumluluğunu netleştirmek
  - Veri yapısı, algoritma, arayüz ve bileşen etkileşimlerini belirlemek
  - Kodlamadan önce sistemi daha anlaşılır hale getirmek
- Bir yazılım projesine hemen kod yazarak başlamak mı yoksa önce tasarlamak mı daha doğrudur? Neden?

Yazılım Tasarımı (Software Design)

- Yazılım tasarımı, bir yazılım sisteminin **nasıl çalışacağını planlama sürecidir.** Gereksinim analizinde "ne yapılacağı" tanımlanır; yazılım tasarımında ise **"nasıl yapılacağı"** belirlenir. Yani;
  - Yazılım tasarımı = fikirlerin koda dönüşmeden önce mühendislik planı haline getirilmesidir.
- Amaçları
  - Sistemi parçalara ayırmak (modülerlik sağlamak)
  - Her parçanın sorumluluğunu netleştirmek
  - Veri yapısı, algoritma, arayüz ve bileşen etkileşimlerini belirlemek
  - Kodlamadan önce sistemi daha anlaşılır hale getirmek
- Bir yazılım projesine hemen kod yazarak başlamak mı yoksa önce tasarlamak mı daha doğrudur? Neden?
  - Muhtemel cevap 1 : Kod yazmak hızlıdır ama ileride karmaşa yaratır.
  - Muhtemel cevap 2 : Tasarım zaman alır ama hataları erken fark etmemizi sağlar.

Temel Tasarım İlkeleri (SOLID Prensipleri)

- Bir yazılımın sadece "çalışması" yeterli değildir; **anlaşılır, sürdürülebilir ve genişletilebilir** olması da gerekir. *İyi bir yazılım tasarımı;* **hataları azaltır, değişiklikleri kolaylaştırır ve kodun ömrünü uzatır.** Bu amaçla yazılım mühendisliğinde **SOLID prensipleri** adı verilen beş temel ilke tanımlanmıştır. Bu ilkeler, **nesne yönelimli programlamanın (OOP)** yapı taşlarındandır.

İlke	Açıklama	Kısa Örnek
S – Single Responsibility	Bir sınıfın veya modülün **yalnızca tek bir sorumluluğu olmalı** ve bu sorumluluk da yalnızca **bir nedenle değişmelidir.**	"UserManager" sadece kullanıcıyı yönetir, login işlemini değil
O – Open/Closed	Bir yazılım bileşeni **genişletilmeye açık**, ama **değiştirilmeye kapalı** olmalıdır.	Yeni özellik alt sınıfla eklenir
L – Liskov Substitution	Bir alt sınıf (subclass), üst sınıfın (superclass) yerine **sorunsuzca** kullanılabilmelidir. Yani, miras alan sınıf **ebeveynin davranışını bozmaz.**	"Square" yerine "Rectangle" geçebilir
I – Interface Segregation	Bir sınıf, **kullanmadığı metodları içeren büyük bir arayüzü** uygulamak zorunda kalmamalıdır. Arayüzler **küçük ve odaklı** olmalıdır.	"IPrinter" ve "IScanner" ayrı olmalı
D – Dependency Inversion	Yüksek seviyeli modüller, düşük seviyeli modüllere **doğrudan bağımlı olmamalıdır.** Her iki katman da **soyutlamalara (interfaces)** bağımlı olmalıdır.	Arayüzler üzerinden iletişim

Yazılım Tasarımı Süreci

- Yazılım tasarımı genellikle iki seviyede ele alınır:
  - **Yüksek Seviyeli Tasarım (Architectural Design)**
    - Sistemi bileşenlere ve alt sistemlere ayırır.
  - **Düşük Seviyeli Tasarım (Detailed Design)**
    - Her bileşenin iç yapısını, algoritmasını ve veri akışını belirler.
- Örnek
  - Kütüphane Yönetim Sistemi
    - **Yüksek Seviyeli Tasarım:**
      - Modüller:
        - Kullanıcı Yönetimi
        - Kitap Kataloğu
        - Ödünç Verme İşlemleri
        - Raporlama
    - Düşük Seviyeli Tasarım:
      - "Kullanıcı Yönetimi" modülünde:
        - Sınıflar: User, Librarian, Student
        - Fonksiyonlar: addUser(), deleteUser(), login()

Yazılım Tasarımı Süreci

- Bir sistemi önce küçük modüllere bölmek bize hangi avantajları sağlar?

Yazılım Tasarımı Süreci

- Bir sistemi önce küçük modüllere bölmek bize hangi avantajları sağlar?
  - Kodun okunabilirliği artar.
  - Farklı ekipler paralel çalışabilir.
  - Hataları bulmak kolaylaşır.
  - Kod tekrarını önler.

İyi Yazılım Tasarımının Özellikleri

- İyi bir tasarım **güzel kod üretir**, kötü tasarım ise **karmaşa doğurur**. Bir yazılımın kalitesi, büyük oranda tasarım kalitesiyle belirlenir.

Özellik	Açıklama	Örnek
Modülerlik	Yazılım, bağımsız modüllere bölünmeli	"Kullanıcı yönetimi" modülü, "ödünç verme"den bağımsız
Soyutlama (Abstraction)	Gereksiz detaylar gizlenmeli	Bir "veritabanı sınıfı" sadece save() ve load() metodu gösterir
Bağımsızlık (Low Coupling)	Modüller birbirine az bağımlı olmalı	Kullanıcı arayüzü, veri tabanını doğrudan çağırmaz
Uyum (High Cohesion)	Her modül tek bir amaç taşımalı	"Raporlama" modülü sadece rapor oluşturmalı
Yeniden kullanılabilirlik	Kod farklı projelerde kullanılabilmeli	"Login" sınıfı tüm web projelerinde kullanılabilir

Yazılım Tasarımı Süreci

- Bir yazılımda modüller birbirine çok bağımlı hale gelirse ne olur?

Yazılım Tasarımı Süreci

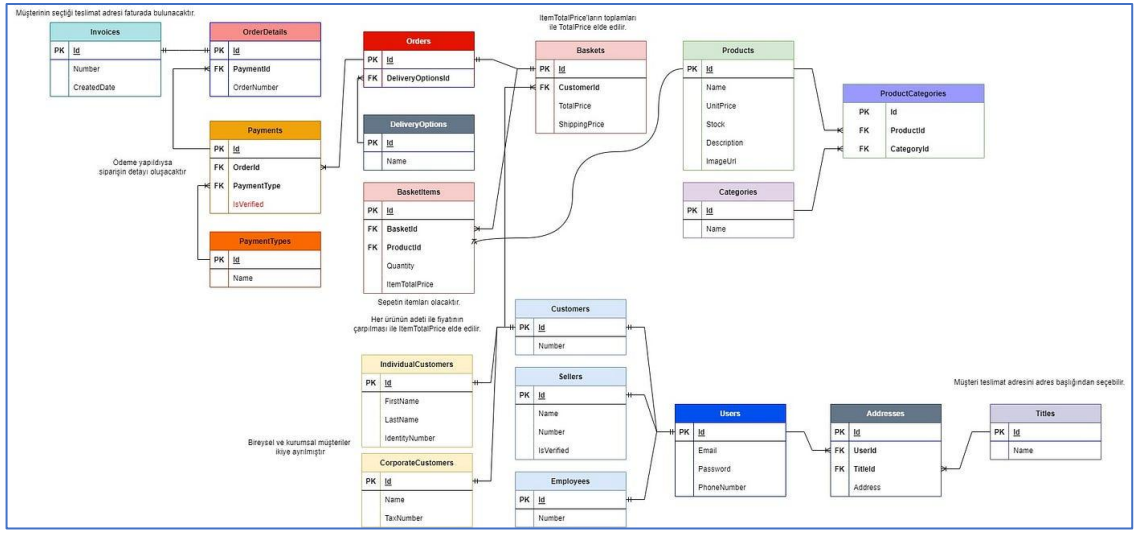
- Bir yazılımda modüller birbirine çok bağımlı hale gelirse ne olur?
  - Birini değiştirince diğeri bozulur.
  - Bakımı zorlaşır.
  - Yeni özellik eklemek riskli olur.

Yazılım Tasarım Yaklaşımları

- **Top-Down Design (Yukarıdan Aşağıya Tasarım)**
  - Önce sistemin genel yapısı belirlenir, sonra detaylara inilir.
  - Büyük resimden küçük parçalara doğru ilerlenir.
  - Örnek
    - Önce “Öğrenci Yönetim Sistemi” → sonra “Not İşlemleri” → sonra “Not Hesaplama Algoritması”.
- **Bottom-Up Design (Aşağıdan Yukarıya Tasarım)**
  - Önce küçük parçalar tasarlanır, sonra sistem bütünleştirilir.
  - Genellikle mevcut bileşenlerin yeniden kullanımına dayanır.
  - Örnek
    - Önce “Dosya Okuma Modülü” tasarlanır, sonra bu modül “Raporlama Sistemi”ne entegre edilir.
- **Object-Oriented Design (Nesne Tabanlı Tasarım)**
  - Her şey “nesne” olarak düşünülür.
  - Gerçek dünyadaki varlıklar (ör. öğrenci, kitap, ders) sınıflara dönüştürülür.
  - UML diyagramları ile görselleştirilir.
  - Örnek
    - Class Ogrenci:

Yazılım Tasarım Sürecinde Kullanılan Araçlar

- UML (Unified Modeling Language) : Diyagramlarla sistemi görselleştirme
- ER Diagramı : Veritabanı ilişkilerini tanımlama
- Flowchart / Akış Diyagramı : Algoritmaları anlatma
- Pseudocode : Gerçek koda geçmeden mantığı gösterme

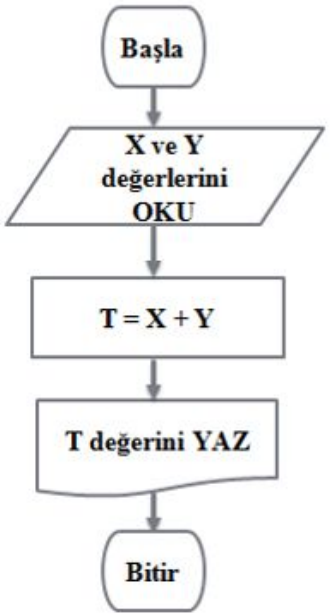


ER Diagramı

Toplam için T, birinci sayı için X, ikinci sayı için Y seç

1. BAŞLA
2. X değerini OKU
3. Y değerini OKU
4. T = X + Y
5. T değerini YAZ
6. BİTİR

Pseudocode



Flowchart

Yazılım Mimarisi (Software Architecture)

- Yazılım mimarisi, bir yazılım sisteminin **yüksek seviyeli yapısını**, **bileşenlerini**, ve bu bileşenler arasındaki **ilişkileri** tanımlayan yapıdır. Basit tanımıyla yazılım mimarisi, **sistemin iskeletidir.**
- Yazılım mimarisi, bir sistemin yapısını, bu yapıyı oluşturan bileşenleri, bileşenlerin dışa dönük davranışlarını ve aralarındaki ilişkileri tanımlar. (IEEE Standard 1471)
- **Amaçları**
  - Sistemin genel yapısını planlamak
  - Teknik kararları erken safhada almak
  - Kalite özelliklerini (performans, güvenlik, ölçeklenebilirlik) sağlamak
  - Takım üyeleri arasında ortak bir anlayış oluşturmak

Yazılım Mimarisi (Software Architecture)

- Yazılım Mimarisinin Önemi

Neden	Açıklama
Yapısal kontrol sağlar	Sistemin büyümesini düzenler.
Teknik riskleri azaltır	Erken dönemde hatalı kararlar fark edilir.
Bakımı kolaylaştırır	Bağımsız bileşenler kolay güncellenir.
Ekip koordinasyonunu sağlar	Her ekip hangi parçadan sorumlu olduğunu bilir.
Teknoloji seçimlerine yön verir	Hangi platform, dil veya altyapı kullanılacağı belirlenir.

- Yazılım Mimarisinin Bileşenleri

Bileşen	Açıklama	Örnek
Bileşenler (Components)	Sistemi oluşturan modüller veya servisler	Kullanıcı modülü, ürün modülü
Bağlantılar (Connectors)	Bileşenler arası iletişim mekanizması	API çağrısı, mesaj kuyruğu
Arayüzler (Interfaces)	Bileşenlerin dış dünyayla etkileşim noktası	REST API, sınıf metodu
Veri akışı (Data Flow)	Bilginin sistem içinde hareketi	Kullanıcı → API → Veritabanı
Yapısal Kurallar (Constraints)	Sistem içi standartlar	“Tüm servisler JWT ile doğrulanacak.”

Yazılım Mimarisi Türleri (Architectural Styles)

- Yazılım mimarisi için farklı ihtiyaçlara göre çeşitli patterns uygulanır. Her tür belirli problem türleri için avantajlar sunar.
- **1. Katmanlı Mimari (Layered Architecture)**
  - Bakım kolaylığı, katmanlar arası bağımsızlık.
  - Katmanlar arası geçiş sayısı fazla ise performans düşebilir.
  - Sistem, katmanlara bölünür (ör. 3 katmanlı yapı)
    - Sunum Katmanı (Presentation Layer)
    - İş Mantığı Katmanı (Business Logic Layer)
    - Veri Katmanı (Data Layer)
- **2. İstemci–Sunucu Mimarisi (Client–Server Architecture)**
  - Sistem iki ana parçadan oluşur
    - İstemci (Client): İstekleri başlatır
    - Sunucu (Server): İstekleri işler ve sonuç döner.
    - Web tarayıcısı (client) → Web sunucusu (server)
  - Avantaj: Basit, anlaşılır yapı
  - Dezavantaj: Sunucuya bağımlılık yüksektir.
- **3. Mikroservis Mimarisi (Microservices Architecture)**
  - Uygulama birçok küçük bağımsız servisten oluşur. Örnek : Bir e-ticaret sistemi: Kullanıcı Servisi, Ürün Servisi …
  - Her servis kendi veritabanına ve iş mantığına sahiptir.
  - Servisler API'lerle haberleşir.
  - Avantaj: Bağımsız geliştirme, kolay ölçeklenebilirlik.
  - Dezavantaj:Servis yönetimi karmaşıktır, izleme zorlaşır.

Yazılım Mimarisi Türleri (Architectural Styles)

- **4. Olay Tabanlı Mimari (Event-Driven Architecture)**
  - Sistem, “olay” (event) kavramına göre çalışır.
  - Olaylar kuyruklara gönderilir, başka bileşenler bunları dinler.
  - Örnek
    - Kullanıcı bir ürün satın aldığında:
      - “Sipariş Oluşturuldu” olayı tetiklenir → Fatura Servisi → Bildirim Servisi → Stok Servisi zinciri çalışır.
  - Avantaj: Esneklik ve gevşek bağ.
  - Dezavantaj: Hataları takip etmek zor olabilir.
- **5. MVC (Model–View–Controller)**
  - Model: Verileri ve iş mantığını temsil eder.
  - View: Kullanıcı arayüzü.
  - Controller: Aradaki koordinasyonu sağlar.
  - Avantaj:Kodun sorumlulukları net ayrılır.
  - Kullanım: Web uygulamaları (Django, ASP.NET, Spring).

Monolitik Yapı

- Monolitik sistemlerde, uygulamanın tüm bileşenleri tek bir kod tabanı içinde bulunur:
  - Kullanıcı arayüzü
  - İş mantığı
  - Veri erişim katmanı
  - Hepsi tek bir derleme ve tek bir deploy dosyası (örneğin .war, .exe) halindedir.
  - Örnek:Bir “Online Mağaza” sisteminde ürün yönetimi, kullanıcı işlemleri, sipariş yönetimi ve ödeme sistemi tek bir uygulama olarak çalışır.
  - Sonuç: Kod büyüdükçe yapı karmaşıklaşır, her küçük değişiklik tüm sistemi yeniden derlemeyi gerektirir.

Özellik	Monolitik Mimari	Mikroservis Mimari
Yapı	Tüm modüller tek uygulama içinde	Uygulama, küçük bağımsız servislerden oluşur
Dağıtım	Tek paket halinde sunulur	Her servis bağımsız olarak dağıtılır
Ölçekleme	Tüm sistemi ölçeklendirmek gerekir	Sadece yoğun servis ölçeklenir
Hata etkisi	Bir hata tüm sistemi etkiler	Hata sadece o servisi etkiler
Ekip çalışması	Tek ekip çalışır	Her servis için küçük ekipler çalışabilir
Teknoloji	Tek dil / framework	Her servis farklı teknolojiyle geliştirilebilir

CI/CD (Continuous Integration & Continuous Delivery/Deployment)

- Geleneksel yazılım geliştirmede, kodlama bittikten sonra:
  - Günlerce test yapılır,
  - Manuel olarak sistemlere yüklenir,
  - Küçük bir hata tüm projeyi geri çekerdi.
  - CI/CD, bu süreci otomatik, hızlı ve güvenilir hale getiren bir yaklaşımdır.
- Amaç: Yazılım değişikliklerinin sürekli olarak test edilip dağıtılmasını sağlamak.

Kısaltma	Tam Açılım	Türkçesi	Ana Amaç
CI	Continuous Integration	Sürekli Entegrasyon	Kod birleştirme ve otomatik test
CD (1)	Continuous Delivery	Sürekli Teslim	Ürünü her an dağıtıma hazır hale getirme
CD (2)	Continuous Deployment	Sürekli Dağıtım	Otomatik olarak canlı sisteme aktarma

CI/CD (Continuous Integration & Continuous Delivery/Deployment)

- **Continuous Integration (Sürekli Entegrasyon) :** Geliştiricilerin kodlarını sık sık (günde birkaç kez) ortak depoya (ör. GitHub, GitLab) entegre etmesi ve bu entegre işleminin **otomatik olarak test edilmesi** sürecidir.
- Amaç:
  - "Kod birleştirme kabusu"nu ortadan kaldırmak
  - Hataları erken fark etmek
  - Takım içinde entegrasyon çatışmalarını azaltmak
  - CI Sürecinin Adımları:
  - Geliştirici kodu Git deposuna gönderir (push).
  - CI aracı (ör. Jenkins, GitHub Actions) otomatik olarak:
    - Kodun derlenmesini sağlar
    - Otomatik testleri çalıştırır
    - Kod kalitesi analizini yapar
    - Eğer testler geçerse kod ana dala (main branch) birleştirilir.

CI/CD (Continuous Integration & Continuous Delivery/Deployment)

- ## Continuous Delivery (Sürekli Teslim)
  - CI sürecinden geçen kod, her zaman **canlı ortama dağıtılmaya hazır hale getirilir.** Ancak dağıtım kararı genellikle **manuel onay** gerektirir.
  - Ana fikir: “Her an release yapılabilecek durumda olmak.”
  - **Amaç:**
    - Kodun her zaman kararlı durumda kalması
    - Yeni sürümlerin hızlı ve güvenilir yayınlanması
    - İnsan hatasını azaltmak
- ## Continuous Deployment (Sürekli Dağıtım)
  - Sürekli teslim sürecinin bir adım ötesidir. Başarılı olan her kod değişikliği, **manuel onay olmadan otomatik olarak canlı ortama dağıtılır.** Kısaca: Her commit → test → deploy.
  - Amaç
    - Tam otomasyon
    - Minimum insan müdahalesi
    - Anında geri bildirim

CI/CD (Continuous Integration & Continuous Delivery/Deployment)

CI/CD Süreci Nasıl Çalışır?

1- Geliştirici kodu gönderir
2- CI süreci başlar: testler, derleme, kod analizi
3- Başarılı olursa “build artifact” üretilir
4- CD süreci başlar: staging → production dağıtımı
5- Sistem testleri ve izleme yapılır

CI/CD Araçları

Araç	Türü	Özellikleri
Jenkins	CI/CD	Açık kaynak, geniş eklenti desteği
GitHub Actions	CI/CD	GitHub içinden entegre akış
GitLab CI/CD	CI/CD	Git tabanlı otomatik pipeline
CircleCI	CI/CD	Bulut tabanlı, kolay konfigürasyon
Travis CI	CI/CD	Açık kaynak projelerde yaygın
Azure DevOps / AWS CodePipeline	Kurumsal	Bulut tabanlı entegre sistemler

CI/CD Kullanmanın Faydaları

Faydalar	Açıklama
Erken hata tespiti	Hatalar küçük parçalarda fark edilir.
Hızlı teslimat	Yeni özellikler hızlıca son kullanıcıya ulaşır.
Otomasyon	İnsan hatası azalır, süreç tekrarlanabilir hale gelir.
Sürekli kalite kontrol	Her commit otomatik test edilir.
Ekip verimliliği	Geliştiriciler kodlama yerine sorun çözmeye odaklanır.

Soru / Cevap

- Bir sistemde neden tek bir dev uygulama (monolitik) yerine küçük servisler tercih edilir?

Soru / Cevap

- Bir sistemde neden tek bir dev uygulama (monolitik) yerine küçük servisler tercih edilir?
  - Bağımsız Geliştirme ve Dağıtım:
    - Her servis ayrı geliştirilebilir, test edilebilir ve güncellenebilir.
    - Bir modülde değişiklik yapıldığında tüm sistemi yeniden dağıtmak gerekmez.
  - Kolay Ölçeklenebilirlik:
    - Yük altında olan yalnızca ilgili servis (ör. “arama servisi”) büyütülür; tüm sistem değil.
    - Bu, kaynak kullanımını optimize eder.
  - Küçük, Uzman Ekiplerle Çalışma:
    - Her mikroservis belirli bir ekibe ait olabilir.
    - Takımlar birbirinden bağımsız ve paralel çalışabilir.
  - Teknoloji Esnekliği:
    - Her servis farklı programlama dili veya veritabanı teknolojisi kullanabilir.Örneğin: Ödeme servisi Java, öneri sistemi Python olabilir.
  - Hata İzolasyonu:
    - Bir serviste hata olduğunda tüm sistem çökmez.Hatalar sadece ilgili servisi etkiler, sistemin geri kalanı çalışmaya devam eder.
  - Sürekli Entegrasyon ve Dağıtım (CI/CD) Kolaylığı:
    - Servis bazında otomatik test, build ve deploy yapılabilir.Agile ve DevOps süreçleriyle uyumludur.
  - Bakım ve Güncelleme Kolaylığı:
    - Servisler küçük, bağımsız ve anlaşılır olduğu için bakım süreci kolaylaşır.Yeni özellik eklemek veya hata düzeltmek diğer modülleri etkilemez.

Yazılım Mimarisi Belgeleme (Architecture Documentation)

- Bir mimari sadece kafada değil, **belgelenmiş** olmalıdır.
- Bu belgeler ekipte ortak anlayışı sağlar.

Belge	İçerik
Bileşen Diyagramı (Component Diagram)	Sistem bileşenlerini ve etkileşimlerini gösterir
Dağıtım Diyagramı (Deployment Diagram)	Bileşenlerin hangi donanımda çalıştığını gösterir
Veri Akışı Diyagramı (DFD)	Veri hareketini görselleştirir
Sistem Blok Diyagramı	Genel sistem mimarisinin kuşbakışı görünümü

Mimarinin Kalite Üzerindeki Etkisi

- İyi bir mimari:
  - Performansı artırır (örneğin cache kullanımı)
  - Bakımı kolaylaştırır (modüler yapı)
  - Güvenliği güçlendirir (katmanlı erişim kontrolü)
  - Ölçeklenebilirliği sağlar (mikroservis veya bulut tabanlı yapı)
- Kötü mimari:
  - Projenin erken değil ama kesinlikle geç bir aşamada “çökmesine neden olur.”

Uygulama

- Senaryo:
  - Bir “Online Yemekhane Sistemi” geliştiriyorsunuz.
    - Öğrenciler mobil uygulamadan menüyü görecek,
    - QR kodla ödeme yapabilecek,
    - Aşçılar günlük menüyü güncelleyecek.
- Görev:
  - Sistemde hangi bileşenler olacak?
  - Bu bileşenler arasında veri akışı nasıl olacak?
  - Hangi mimari tarz daha uygun olur (ör. Katmanlı, Mikroservis, MVC)?

Yazılım Tasarım Kalıpları (Design Patterns)

- Yazılım geliştirmede birçok problem tekrar tekrar karşımıza çıkar. Bu problemleri çözmek için her seferinde sıfırdan çözüm üretmek zaman kaybıdır.
- İşte burada **tasarım kalıpları (design patterns)** devreye girer.
- Tanım : Tasarım kalıpları, yazılım geliştirmede sıkça karşılaşılan problemler için **tekrarlanabilir, test edilmiş ve kanıtlanmış çözüm şablonlarıdır.**
- Amaç:
  - Yeniden kullanılabilir çözümler sunmak
  - Takım içi ortak dil oluşturmak
  - Kodun bakımını ve genişletilmesini kolaylaştırmak
  - Yazılımın mimari kalitesini artırmak
- Örnek:
  - Aynı problem için 5 geliştirici 5 farklı çözüm yerine, tasarım kalıbı ile ortak bir standart yaklaşım kullanabilir.

Soru / Cevap

- Sizce iyi yazılmış bir yazılımda neden belirli tasarım kalıplarını kullanmak tercih edilir? Özgün çözüm üretmek daha yaratıcı değil midir?

Soru / Cevap

- Sizce iyi yazılmış bir yazılımda neden belirli tasarım kalıplarını kullanmak tercih edilir? Özgün çözüm üretmek daha yaratıcı değil midir?
- Tasarım kalıpları tekerleği yeniden icat etmememizi sağlar.
  - Yazılım dünyasında birçok problem zaten çözülmüştür.
  - Kalıplar, bu çözümleri tekrarlanabilir ve güvenilir hale getirir.
- Kalıplar kodu standardize eder.
  - Bir ekipte herkes aynı problemi aynı biçimde çözer; bu da okunabilirliği ve sürdürülebilirliği artırır.
- Kalıplar deneyimin birikimidir.
  - Onlar “hazır reçete” değil, yıllar içinde yüzlerce projede sınanmış mühendislik prensipleridir.
- Özgünlük kalıbın içinde de mümkündür.
  - Kalıp, yalnızca çerçeveyi verir; içindeki uygulama, algoritma, veri modeli yine geliştiricinin yaratıcılığına kalır.
- Kalıplar karmaşık sistemleri sadeleştirir.
  - Özellikle büyüyen projelerde, kodun tutarlılığını ve genişletilebilirliğini korur.
- Ekip iletişimini kolaylaştırır.
  - “Observer kullanalım” veya “Factory ekleyelim” dendiğinde herkes aynı yapıyı anlar — bu ortak bir mühendislik dili oluşturur.
- Kalıplar, yeniden kullanılabilirliği ve bakımı kolaylaştırır.
  - Kodun parçalarını değiştirmeden yeni işlevler eklenebilir.(Örneğin Strategy Pattern, yeni algoritmaları kolayca sisteme entegre eder.)

Tasarım Kalıplarının Sınıflandırılması

- Gamma'ya göre tasarım kalıpları **üç ana kategoriye** ayrılır:

Kategori	Amaç	Örnek Kalıplar
Yaratımsal (Creational)	Nesne oluşturma sürecini esnek hale getirir	Singleton, Factory, Builder
Yapısal (Structural)	Nesneler arası ilişkileri düzenler	Adapter, Facade, Decorator
Davranışsal (Behavioral)	Nesneler arasındaki iletişimi düzenler	Observer, Strategy, Command

- **Yaratımsal (Creational) Tasarım Kalıpları**
- Bu kalıplar, nesne oluşturma sürecini soyutlar. Amaç, nesnelerin doğrudan oluşturulmasından kaynaklanan karmaşıklığı azaltmaktır.
  - **Singleton Pattern (Tek Nesne Kalıbı) :** Bir sınıfın yalnızca **tek bir örneğe (instance)** sahip olmasını ve bu örneğe global erişim sağlanmasını garanti eder. Veritabanı bağlantısı, Logger (kayıt sistemi), Konfigürasyon yöneticisi kısımlarında kullanılır.
  - **Factory Pattern (Fabrika Kalıbı) :** Nesne oluşturma sürecini alt sınıflara devrederek, **nesne türünü dinamik olarak belirlemeye** olanak tanır. Farklı veri kaynağı tipleri (JSON, XML, SQL) ve UI bileşen üretimi kısılmarında kullanılır.
  - **Builder Pattern:** Karmaşık nesnelerin adım adım oluşturulmasını sağlar. Örnek: Bir "pizza siparişi" sisteminde, farklı malzemelerin esnek kombinasyonlarla eklenmesi.

Tasarım Kalıplarının Sınıflandırılması

- **Yapısal (Structural) Tasarım Kalıpları**
- Bu kalıplar, sınıfların veya nesnelerin **birlikte çalışma biçimini** tanımlar. Amaç, mevcut sınıfların yapısını değiştirmeden aralarında **uyum** sağlamaktır.
  - **Adapter Pattern (Uyarlayıcı Kalıp) :** Bir sınıfın arayüzünü başka bir sınıfın beklediği arayüze dönüştürür.
    - Yani, **uyumsuz sistemlerin birlikte çalışmasını sağlar. Gerçek Dünya Örneği:** Elektrik priz dönüştürücüsü (220V → 110V).
    - Kullanım Alanı: Eski sistemlerin yeni kodla entegre edilmesi.
  - **Facade Pattern (Cephe Kalıbı) :** Karmaşık bir sistemi basitleştirilmiş bir arayüzle sunar. Kullanıcı, detaylarla uğraşmadan ana işlevi çağırır.
    - Örnek: Bir "Ev Otomasyon Sistemi"nde ışık, perde, müzik kontrolü tek bir "SmartHomeController" üzerinden yapılır.
  - **Decorator Pattern (Süsleyici Kalıp) :** Bir nesnenin davranışını **kalıtım kullanmadan** dinamik olarak genişletir.
    - **Örnek:** Bir kahve sipariş sisteminde, "süt", "şeker" gibi eklemeleri sonradan yapabilmek.

Tasarım Kalıplarının Sınıflandırılması

- **Davranışsal (Behavioral) Tasarım Kalıpları**
  - Bu kalıplar, nesneler arasındaki **iletişimi ve sorumluluk paylaşımını** düzenler. Odak noktası: "**Kim, ne zaman ve nasıl tepki verir?**"
  - **Observer Pattern (Gözlemci Kalıbı) :** Bir nesnede değişiklik olduğunda, o nesneyi izleyen diğer nesnelerin **otomatik bilgilendirilmesini** sağlar.
    - **Gerçek Dünya Örneği:** Bir YouTube kanalına abone olduğunuzda, yeni video yüklendiğinde bildirim almanız.
  - **Strategy Pattern (Strateji Kalıbı) :** Bir algoritmanın farklı varyasyonlarını birbirinin yerine geçebilir hale getirir.
    - **Örnek:** Bir "ödeme sistemi" için "kredi kartı", "havale", "PayPal" stratejileri.
  - **Command Pattern (Komut Kalıbı) :** Bir isteği nesne olarak kapsüller. Geri alma (undo) işlemleri için uygundur.
    - **Örnek:** Metin editöründe "Ctrl+Z" özelliği.

Tasarım Kalıplarının Faydaları

Fayda	Açıklama
Yeniden Kullanılabilirlik	Aynı çözüm farklı projelerde uygulanabilir.
Bakım Kolaylığı	Kod daha modüler ve anlaşılır olur.
Standartlaşma	Ekipler arasında ortak dil oluşturur.
Esneklik	Davranışlar, yeni sınıf eklemeden genişletilebilir.
Sürdürülebilirlik	Yazılımın yaşam döngüsünü uzatır.

İSTANBUL MEDENİYET ÜNİVERSİTESİ

KATILIMINIZ İÇİN TEŞEKKÜR EDERİM.

YOKLAMA İÇİN İMZANIZI ATMAYI UNUTMAYINIZ.
CLASSROOM ÜZERİNDEN SINIFA DAHİL OLMAYI UNUTMAYINIZ

Sınıf Kodu : pua2tnoe

Dr. Öğr. Üyesi Ertürk ERDAĞI
erturk.erdagi@medeniyet.edu.tr