

Tasarım Kalıpları

Tasarım Kalıbı Nedir?

- Kalıpların amacı nesne merkezli prensipleri kullanarak;
 - Doğru sorumlulukları bulmak,
 - Bu sorumlulukları nesnelere dağıtmak,
 - Nesnelere aralarında az bağımlılık olacak şekilde kurgulamak
- Kalıplar nesne seviyesindeki problemler içindir, iş alanından bağımsızdır
- Yazılımcının ne yaptığını açıklamak istediğinde kullanılır

Amaç

- Formal ve yaygın bir dil oluşturmak.
- Nesne oluşturulurken mümkün olduğunca soyutlanmalı (clean code için)
- Herkezin gelişigüzel nesne üretmesini engellenmeli
- Nesneye erişim kontrol edilmeli
- Nesneler arasındaki bütün parça ilişkisi iyi kurgulanmalı
- İşlemi yapmanın farklı şekilleri varsa bunları düzgün ayırmalıyız ve client'dan gizlemeliyiz
- Nesneleri düzgün haberleştirmeliyiz
- Nesneye çalışma zamanında yetkinlik kazandırılmalıdır
- Nesneyi yeri geldiğinde saklamalı yeri geldiğinde ulaşım açmalıyız

Neden Kalıplara İhtiyaç Duyuyoruz

- F. Brooks'a göre;
- Yazılım diğer mühendislik alanlarından daha karmaşıktır (fiziksel kısıtları yoktur) (kastedilen algoritma karmaşıklığı değildir)
- Yazılım sürekli değişir
- Yazılım görünmezdir
- Yazılım uyumludur

Yazılım Karmaşıktır

- Yapısal olarak karmaşıktır.
- Nesnelerin birbiri ile olan ilişkileri.
- Önemli olan binlerce sınıfın karmaşıklığını yönetmek.
- **Bileşen karmaşıklığı + ilişkisel karmaşıklık.**

Birliktelik (Cohesion)

- Bileşen karmaşıklığı, bileşen alt parçalarının ne kadar “birlikte” olduğunun ölçüsüdür.
- Birliktelik tek bir amaca, tek bir işe odaklıdır.
- Birlikteliğe sahip bileşenler;
 - Düşük karmaşıklığa sahiptir
 - Bakım maliyeti düşüktür
 - Tekrar kullanıma sahiptir
 - Sınıflarda yüksek birliktelik beklenir

Birliktelik Gelişigüzel ise

- Birbiriyle alakalı olmayan yapıların aynı sınıf içinde toplanmasıdır. Her şeyi aynı anda yapmaya çalışan "God Object" sınıflardır.
- Nesne soyutlaması yapılmadığında ya da eksik yapıldığında karşılaşılr.
- Özellikle çok fazla tanımlanan staticler.

```
public class RandomOperations {  
  
    public void printHello() {  
  
        System.out.println("Hello");  
  
    }  
  
  
    public int addNumbers(int a, int b) {  
  
        return a + b;  
  
    }  
  
  
    public void saveToFile(String data) {  
  
        System.out.println("Saving data: " + data);  
  
    }  
  
}
```

Birliktelik Mantıksal ise

- Gerçekte farklı işlevleri olmalarına rağmen tek bir işlev ile ilgili olduğu düşünülen yapılardır. (hem input hem de şifrelemelerin yapıldığı sınıf)

- `public class Alisveris {`
`public void marketAlisveris();`
`public void isiAlisveris();`
`public void isAlisveris();`
`public void emlakAlisveris();`
`}`

Birliktelik Zamansal ise;

- Belirli bir zaman diliminde peş peşe yapılması gereken, ancak işlevsel olarak farklı olan işlerin bir araya getirilmesidir.
- Bir dosya açmak istendiğinde hata meydana gelince bunun loglanması, gerekli hata bildirimleri ve başka işleme devam etme. Bu karmaşıklığa sebep olur.

```
public class FileOperations {  
  
    public void openFile() {  
  
        System.out.println("Dosya açıldı.");  
  
    }  
  
  
    public void logError() {  
  
        System.out.println("Hata kaydedildi.");  
  
    }  
  
  
    public void notifyUser() {  
  
        System.out.println("Kullanıcıya hata bildirildi.");  
  
    }  
  
}
```

Birliktelik Prosedürel ise;

- Bir sürecin parçası olan işlemlerin, belirli bir sırayla yapılmak üzere tek bir sınıfta toplanmasıdır.
- Bir konu ile ilgili işlerin tek bir sınıfta sırayla metotsal olarak ayrılması. Örneğin dosya ile ilgili tüm işlemlerin tek bir sınıfta olması.

```
public class OrderProcessor {  
  
    private String orderId;  
  
    void createOrder() {  
  
        System.out.println("Sipariş oluşturuldu.");  
  
    }  
  
    void processPayment() {  
  
        System.out.println("Ödeme işleniyor.");  
  
    }  
  
    void printInvoice() {  
  
        System.out.println("Fatura yazdırıldı."); }  
  
}
```

Birliktelik İletişimsel ise;

- Aynı veri üzerinde farklı işlemler yapan fonksiyonların bir araya getirilmesidir. Genellikle veri işleme ve veritabanı işlemlerinde görülür.

```
public class FileProcessor {  
  
    void read() {  
  
        System.out.println("Dosya okunuyor.");  
  
    }  
  
    void process() {  
  
        System.out.println("Dosya işleniyor.");  
  
    }  
  
    void save() {  
  
        System.out.println("Dosya kaydedildi."); }  
  
}
```

Birliktelik Ardışıl ise;

- Bir işlemin çıktısının bir diğer işlemin girdisi olduğu, "pipe" (boru hattı) şeklindeki yapıların bir arada bulunmasıdır.

```
public class SequentialOrderManager {  
  
    boolean checkStock(int productId) {  
  
        return true;  
  
    }  
  
    double calculatePrice(boolean isAvailable) {  
  
        return isAvailable ? 150.0 : 0.0;  
  
    }  
  
    void generateInvoice(double price) {  
  
        System.out.println("Fatura Tutarı: " + price);  
  
    }  
  
}
```

Birliktelik Fonksiyonel ise;

- En iyi birliktelik durumudur. Sınıfın tüm parçaları tek bir spesifik ve iyi tanımlanmış işe odaklanmıştır.
- Diğer birliktelik tiplerinden farklı olarak iş büyüdükçe böl parçala işlemleri ile odak korunabilir.

```
public class MathCalculator {
```

```
    public int sum(int a, int b) {
```

```
        return a + b;
```

```
    }
```

```
    public double sqrt(double value) {
```

```
        return Math.sqrt(value);
```

```
    }
```

```
}
```

- Bu birliktelikler belirlenirken iyi bir iş analizi yapılmalı.
(Müşteri için belirlenen bir giriş çıkış işlemi daha sonra farklı kategoriler için de gerekebilir)

Örnek

- Bir öğrenci sınıfının içinde tüm işlemler varsa hangi birlikteliğe uyar? Bölersek hangi birlikteliğe uyar?

Örnek

- İki farklı sınıf kurgularsak hangisi daha birliktedir?

- Her şeyi tek sınıfa toplamak "Birliktelik-Karşıtı Kalıp" (Anti-pattern) olarak değerlendirilirken, görevleri ayırmak yüksek birliktelik ve düşük bağımlılık (Low Coupling) hedefine ulaştırır.

Nesne Sınıflandırması

- Nesneleri bazı rollere göre sınıflandırırız.
- Rollere uygun olarak sorumluluk ve veriyi artırmak birlikteliği güçlendirir.

- Ivar Jacobson'a göre;
 - Boundary: Sistemin aktörleri ile olan iletişimini yöneten (interface) nesneler. (Pencereler veya düğmeler gibi GUI bileşenleri, web servisler vb.) (OgrenciArayuzu, OgrenciWebServis.)
 - Control: İş süreçlerini yöneten nesnelerdir (service). (OgrenciIslemYoneticisi)
 - Entity: İş alanı (business domain) nesneler. İş ile ilgili bilgiyi bilen ve en temel davranışları modelleyen nesnedir. (Öğrenci Sınıfı)

- Hexagonal Mimari buna en güzel örnek mimaridir.

- Veri ile veriyi işleyen davranış entity, veri sunumu boundary, koordinasyon use case ve uygulama davranışı controller nesneleri tarafından yapılır.
 - Model → Entity
 - View → Boundary
 - Controller → Controller

- Aynı roldeki nesneler birbirini kullanabilir.
- B C nesnelerini, C de E nesnelerini kullanabilir.
- Bunun tersi ya da B'nin doğrudan E'yi kullanması tavsiye edilmez.

- Neden Bu Ayrımı Yapıyoruz?
- **Mixed-instance Cohesion'dan Kaçınmak:** Bir sınıfın hem veriyi tutup hem de o veriyi yönetmesi (yönetici-nesne problemi) mimari bir sorundur.
- **Değişim Kolaylığı:** Arayüz (Boundary) değiştiğinde iş mantığı (Control) veya veriler (Entity) bundan etkilenmez.
- **Tek Sorumluluk (Single Responsibility):** Her sınıfın değişmek için tek bir sebebi olur, bu da bakım maliyetini düşürür.

Birliktelik-Karşı Kalıpları

- Nesnelerin mimarideki ve fonksiyonel yapıdaki rollerini karıştırıp, aralarındaki farkı görmezden gelmektir.
- Nesnelerin mimarideki ve fonksiyonel yapıdaki rollerinin birbirine karıştırılması veya bu rollerin tamamen görmezden gelinmesi durumu olarak tanımlanır. Bu kalıplar, yazılımın esnekliğini azaltan ve bakımını zorlaştıran mimari hatalardır.

- Mixed-instance cohesion: Bir sınıfın, yarattığı nesnelerin sadece bir kısmına hitap eden özellikler veya metotlar barındırmasıdır. Bir sınıfın birtakım özellikleri bazı nesneler için geçerli, bazı nesneler için geçerli değildir. (çalışan sınıfı için yönetici olunması problemi)
- Mixed-domain cohesion: Yazılımın farklı katmanlarına veya iş alanlarına ait mantıkların tek bir yapıda toplanmasıdır. Sınıf farklı yazılım alanlarının (domain) özelliklerini bir araya getiri. (Bir e-mail uygulamasının her şeyi yapması)
- Mixed-role cohesion: Mimari tasarlanırken nesnelerin üstlenmesi gereken rollerin (Boundary, Control, Entity) ayırt edilememesidir. Aynı alanda olsa bile farklı rollere sahip nesnenin özelliklerinin tek bir nesnede toplanması.

- Mixed-instance cohesion
 - Kötü Örnek (Anti-Pattern): Öğrenci sınıfının hem öğrenci bilgilerini (isim, numara) tutması hem de veritabanına kendini kaydetmesi veya not ortalaması hesaplaması.
 - **Sorun:** Bir öğrencinin adını değiştirmek istediğinizde de, veritabanı bağlantı ayarını değiştirmek istediğinizde de aynı sınıfı modifiye etmeniz gerekir.
 - **Çözüm:** Veriyi Entity (Öğrenci), iş mantığını Control (ÖğrenciServis) nesnelere ayırmak.

- Mixed-domain cohesion
 - Kötü Örnek (Anti-Pattern): Bir nesnenin hem e-mail gönderme, hem spam kontrolü yapma, hem de kullanıcı rehberini yönetme gibi farklı alanlardaki tüm rolleri üstlenmesidir.
 - **Sorun:** Bu durum "God Object" yaratır; kodun bir yerini düzeltmek sistemin alakasız bir yerinde hataya sebep olur.
 - **Çözüm:** İşlevleri birbirinden bağımsız modüllere veya mikro servis benzeri küçük sınıflara ayırmaktır.

- Mixed-role cohesion
 - Kötü Örnek (Anti-Pattern): Bir HTML sayfasından (Arayüz) gelen veriyi doğrudan bir SQL sorgusuna (Veri) gönderen ve aradaki hesaplama mantığını (Kontrol) atlayan bir yapı.
 - **Sorun:** Arayüz değiştiğinde (örneğin webden mobil uygulamaya geçildiğinde), tüm veritabanı mantığının da çökmesi veya yeniden yazılması gerekir.
 - **Çözüm:** Sistemin aktörle iletişimini Boundary, işleyişini Control, verisini Entity sınıflarına bölmektir.

Kaynakça

- F. Brooks, No silver Bullet
- <https://netflixtechblog.com/>
- Akın Kaldıroğlu ders notları