

Yazılım Tasarım ve Mimarisi

Bağımlılık (Coupling)

- Bir işin kendi başına olabilmesi ya da diğerleriyle ne kadar ilgili olduğu (Bağımlılık) koordinasyon karmaşıklığıdır.
- Bağımlılık düşük ise karmaşıklık da düşüktür.
- Birliktelik ve Bağımlılık tasarım prensibi 1960'lı yıllardan beridir (Larry Constantine, Structured Design) bilinen ve kullanılan prensiplerdir. Mimari temelleri sağlam atmak için bu prensipleri iyi bilmeyi gerektirir.

- Nesneler arasındaki bağımlılık arttıkça aralarında daha fazla veri akışı meydana gelir, koordine etmek güçleşir. Değişiklik gerektiğinde tümü etkilenir.
- Birlikteliği düşük olan her şeyin bağımlılığı yüksektir.

Bağımlılıkta İçerik

- Yapıların birbirlerinin iç yapılarına bağımlı olduğu durumdur.
- Yanlış soyutlama yapılmıştır.
- Doğrudan alan erişimi en sık görülen bağımlılıktır.
- İç yapılara bağımlılıktan dolayı birçok şeyi birlikte değiştirmek gerekir.
- Bir bileşen diğer bileşenin verisini ya da kontrol akışını değiştirir.
- Bu en kötü bağımlılıktır.

- İerik bağımlılığını azaltmak için kapsülleme yaklaşımı kullanmalıdır.

```
class Car {  
  
    public int oilLevel = 0; // Aracın yağ seviyesi (iç veri alanı)  
  
}
```

```
class Mechanic {  
  
    void service(Car car) {  
  
        // Kötü: Arabanın iç durumunu doğrudan değiştiriyor (içerik bağımlılığı)  
  
        car.oilLevel = 100;  
  
        System.out.println("Yağ seviyesi ayarlandı: " + car.oilLevel);  
  
        // Mechanic, Car'ın iç yapısına bağımlı -> Car değişirse burası da bozulur.  
  
    }  
  
}
```

```
class Car {  
  
    private int oilLevel = 0;    // Kapsülleme: yağ seviyesini değiştirmek için public arayüz (metot)  
  
    public void refillOil(int amount) {  
  
        this.oilLevel += amount;  
  
        System.out.println("Yağ seviyesi güncellendi: " + this.oilLevel);  
  
    }  
  
}  
  
class Mechanic {  
  
    void service(Car car) { // İyi: Car'ın iç yapısını değiştirmiyor, onun sunduğu arayüzü kullanıyor  
  
        car.refillOil(100);    // Mechanic, Car'ın sadece public metotlarını biliyor, iç detaylarına bağımlı değil.  
  
    }  
  
}
```

Bağımlılıkta Ortaklık

- Global değişkenleri kullanan yapılardır. İki bileşen ortak bir veri yapısına bağımlıdır.
- OOP'nin genel mantığına da aykırıdır. Bir sınıfta her şeyi public tanımlamak yerine hizmet almak gerekir.
- Sabit değişkenler, static tanımlanan ögeler ve Singleton gibi kalıplarda görülür.
- Ortak veri yapısındaki değişiklik ilgili bütün bileşenlerde değişikliğe sebep olur.

- Ortak bağımlılığı ortadan kaldırmak için soyutlama prensibini uygulanmalıdır.

```
class CommonData {  
    public static int counter = 0; // Global paylaşılan veri  
}
```

```
class A {  
    void doSomething() {  
        CommonData.counter = 5;    // Kötü: Global (ortak) veriyi değiştiriyor  
    }  
}
```

```
class B {  
    void doSomethingElse() {  
        System.out.println("Sayaç: " + CommonData.counter);    // Kötü: Global veriye bağımlı (dışarıda bir yerde değişebiliyor)  
    }  
}
```

```
class Counter {  
  
    private int count = 0;  
  
    public void setCount(int c) { this.count = c; }  
  
    public int getCount() { return this.count; }  
  
}
```

```
class A {  
  
    void doSomething(Counter counter) {  
  
        counter.setCount(5);    // İyi: Gerekli veri, parametre olarak sağlanan nesne aracılığıyla ayarlanıyor    }  
  
}
```

```
class B {  
  
    void doSomethingElse(Counter counter) {  
  
        System.out.println("Sayaç: " + counter.getCount());    // İyi: Gerekli veri ihtiyaç halinde parametreyle alınıyor    }  
  
}
```

Bağımlılıkta Dışsallık

- Dışsallık bağımlılığı, birden fazla modülün ortak bir dış sistem, kaynak veya veri formatına bağımlı olması durumudur.
- Örneğin iletişim kurulan cihazın arayüz tanımına bağımlılık bulunmaktadır.
- Örneğin, iki farklı sınıfın aynı dosyaya yazma/okuma yapması, aynı donanım cihazının arayüzüne doğrudan erişmesi veya aynı sabit yapılandırma değerine (dosya yolu, ortam değişkeni vb.) bağımlı olması bu kategoridedir. Bu durumda, dış sistemdeki bir değişiklik (örn. dosya formatının değişmesi) tüm bağlı modülleri etkiler. İyi tasarım, bu dışsal bağımlılıkları soyutlama katmanı arkasına alarak yönetir. Yani, dış sisteme erişim tek bir yerde toplanır ve diğer kısımlar bu soyut arayüzü kullanır.

- Veri biçiminden bağımsız şekilde iş alanı nesnelerini tanımlayarak yok edilebilir.

```
class DataExporter {  
  
    void exportData(String data) {  
  
        String filePath = "data.txt";    // Kötü: Doğrudan sabit bir dosya yoluna (dış kaynağa) bağımlı. data.txt'den  
        "dataset.txt"ye değişirse iki yerin de değişmesi lazım.  
  
    }  
  
}
```

```
class DataImporter {  
  
    String importData() {  
  
        String filePath = "data.txt"; // Kötü: Aynı sabit dosya yoluna bağımlı  
  
        return "veri"; // ... filePath'ten veri okuma işlemi ...  
  
    }  
  
}
```

```
class DataStorage {  
    private String filePath;  
  
    DataStorage(String path) {  
        this.filePath = path;  
    }  
  
    public void save(String data) {  
        System.out.println(filePath + " dosyasına yazılıyor: " + data);    // Veriyi filePath ile belirtilen dış kaynağa yazar  
  
        // ... (gerçek dosya yazma işlemleri) ...  
    }  
  
    public String load() {  
        System.out.println(filePath + " dosyasından okunuyor...");    // filePath konumundan veri okur  
  
        return "veri";  
    }  
}
```

```
class App {  
  
    public static void main(String[] args) {  
  
        DataStorage storage = new DataStorage("data.txt");    // İyi: Dış kaynak  
erişimi sadece DataStorage üzerinden yapıyor  
  
        storage.save("örnek veri");  
  
        String veri = storage.load();  
  
        System.out.println("Okunan Veri: " + veri);  
  
    }  
  
}
```

Bağımlılıkta Kontrol

- Yapıların flag geçerek birbirlerinin akışlarını kontrol etmesidir. Yani, bir metodun parametre olarak başka bir modüle ne yapması gerektiğini söylemesi durumudur.
- İçerik bağımlılığının özel bir halidir.
- Başka bir bileşenin nasıl işleyeceği ile ilgili kontrol algoritmasını belirler. Örneğin sıralama algoritmasına gönderilecek karşılaştırma metodu böyle bir bağımlılığı ifade eder.

- Durum makinesi (state machine) ve strateji patterni ile önlenebilir.

```
class Controller {    //Controller, dışarıdan gelen parametreye bağlı olarak işlemi yönetiyor.  
  
    void process(int type) {  
  
        if (type == 1) {  
  
            System.out.println("İşlem Tipi 1");  
  
        } else if (type == 2) {  
  
            System.out.println("İşlem Tipi 2");  
  
        }  
  
    }  
  
}
```

```
interface Strategy {    //Strategy Pattern ile akış bağımsız hale getirildi.
```

```
    void execute();
```

```
}
```

```
class Type1Strategy implements Strategy {
```

```
    public void execute() {
```

```
        System.out.println("İşlem Tipi 1"); }
```

```
}
```

```
class Type2Strategy implements Strategy {
```

```
    public void execute() {
```

```
        System.out.println("İşlem Tipi 2"); }
```

```
}
```

```
class Controller {
```

```
    void process(Strategy strategy) {
```

```
        strategy.execute(); }
```

```
}
```

Bağımlılıkta Veri Yapısı

- Yapıların birbirlerine karmaşık veri yapıları ile oluşturdukları bağımlılıktır.
- Bir bileşen diğer bir bileşen ile bir nesne ya da veri yapısının tamamını paylaşır. Burada esas sorun veriyi alan bileşenin bütün veri yapısına değil sadece belirli bilgilere bağımlı olmasıdır.

```
class Person {  
    String name;  
  
    int age;  
  
    String address;  
  
}
```

```
class Report {  
    void print(Person person) {    //Tüm Person nesnesi alındı fakat hepsi kullanılmadı.  
  
        System.out.println(person.name);  
  
    }  
  
}
```

```
class Person {    //Yalnızca gerekli veriler paylaşılır.
```

```
    private String name;
```

```
    private int age;
```

```
    private String address;
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
}
```

```
class Report {
```

```
    void print(String personName) {
```

```
        System.out.println(personName);
```

```
    }
```

```
}
```

```
class Main {  
  
    public static void main(String[] args) {  
  
        Person person = new Person();  
  
        Report report = new Report();  
  
  
        String isim = person.getName();  
  
        report.print(isim);  
  
    }  
  
}
```

Bağımlılıkta Veri Bağımlılığı

- Veri bağımlılığı, bir modülün diğerine gereken verileri parametre olarak geçirerek etkileşime girmesidir. Bu aslında en gevşek (düşük) bağımlılık türlerinden biridir çünkü bileşenler sadece ihtiyaç duydukları veriyi alır ve başka hiçbir detayı bilmez. Ancak bazen aşırı sayıda parametre kullanımı veya verinin düzenlenme biçimi, tasarımı karmaşıklıştırabilir.
- Veri yapısı bağımlılığının daha basit halidir.

- Bu bağımlılık türünü yönetmek için Anemic Domain Model (Fonksiyonsuz sadece veri içeren sınıflar) yapısından kaçınmak ve veri ile davranışı bir arada tutmak (kapsülleme) kritik öneme sahiptir.

- Mesaj tabanlı ara katman kullanılarak yok edilir.

```
class Order {
```

```
    void process(String product, int quantity, double price,  
String customer) {
```

```
        System.out.println(product + " sipariş edildi.");
```

```
    }
```

```
}
```

```
class OrderDetails {    //Parametreler tek bir nesnede gruplandı.  
  
    String product;  
  
    int quantity;  
  
    double price;  
  
    String customer;  
  
}
```

```
class Order {  
  
    void process(OrderDetails details) {  
  
        System.out.println(details.product + " sipariş edildi.");  
  
    }  
  
}
```

Bağımlılıkta Mesaj

- Nesneler birbirleri üzerinden metotlarla haberleşirler. Mesajlar bir araya gelerek arayüzleri oluşturur.
- Arayüz bilgisi dışında başka hiçbir bilgi gerektirmeyen bağımlılıktır.
- Gerekirse API dokümantasyonu oluşturulabilir.

```
class Button {  
  
    void onClick() {  
  
        Screen screen = new Screen();        // Kötü: Button doğrudan Screen sınıfına bağımlı ve onu çağırıyor  
  
        screen.showMessage("Butona tıklandı!");  
  
    }  
  
}
```

```
class Screen {  
  
    void showMessage(String msg) {  
  
        System.out.println("Ekran mesajı: " + msg);  
  
    }  
  
}
```

```
interface ClickListener {  
    void onClick();  
}  
  
class Button {  
    private ClickListener listener;  
  
    public void setClickListener(ClickListener l) {  
        this.listener = l;  
    }  
  
    public void onClick() {  
        // İyi: Button, dinleyici arayüzünü çağırıyor, somut sınıfı bilmiyor  
        if (listener != null) {  
            listener.onClick(); // bir mesaj gönderiyor diyebiliriz  
        }  
    }  
}
```

```
class Screen implements ClickListener {  
  
    public void onClick() {  
  
        // Butona tıklanınca yapılacak işlemler  
  
        showMessage("Butona tıklandı!"); }  
  
    void showMessage(String msg) {  
  
        System.out.println("Ekran mesajı: " + msg); }  
  
}  
  
class Main {  
  
    public static void main(String[] args) {  
  
        Button button = new Button();  
  
        button.setOnClickListener(new Screen()); // Button'a bir ClickListener veriliyor  
  
        // Artık Button, Screen'in detaylarını bilmeden iletişim kurabilir  
  
        button.onClick(); // Buton tıklandığında Screen.onClick() çağrılacak  
  
    }  
  
}
```

Bağımlılıkta Sıfır Bağımlılık

- Sistem olabilmek için mümkün olmayan bağımlılıktır.

- Bir bileşen diğer bileşenler ile ilgili en az şey bilmeli
- Bir bileşen sadece yakın bileşenler ile haberleşmeli yani komşusu değilse doğrudan haberleşmemeli

NTP'de Bağımlılık

- Miras bağımlılığı: Bir sınıfta bir şey değişince diğer sınıf bozulabilir.
- Soyut bağımlılık
 - Implementation inheritance (Gerçekleştirme mirası)
 - Interface inheritance (arayüz mirası)

Soyut Bağımlılık

- Mesaj bağımlılığından daha iyidir.
- Nesneler arayüzleri belirleyen üst tipi bilir, gerçek tipi bilmez.
- Polymorphism en iyi örnektir.
- Dependency inversion ile elde edilir. (Nesne yönelimli tasarımda, bağımlılığı tersine çevirme ilkesi, yazılım modüllerini gevşek şekilde bağlamak için özel bir metodolojidir.)

Birliktelik ve Bağımlılık

- Bileşenlerin yüksek birliktelikte ve düşük bağımlılıkta olmasını bekleriz.
 - Daha küçük parçalara bölünememeli.
 - Tek bir amacı olmalı.
 - Birbiriyle ilişkisi en az seviyede olmalı.

Süreçsel İşler (Koordinasyon İşleri)

- İşlerin koordineli olarak bir süreç oluşturacak şekilde bir araya getirilmesidir.
- Genellikle sırayla işler. Birinin çıktısı diğerinin girdisi olabilir.

Değişim

- Yeni ihtiyaçlar yazılım bileşen ve koordinasyon karmaşıklığını artırır.
- Değişim çok maliyetlidir.
- En sık değişiklik yapılanlar en kolay değiştirmelidir.
- Birlikteliği düşük, bağımlılığı yüksek bileşenleri değiştirmek zordur.

Veri - Davranış

- Veri temelli bir yaklaşım hem birliktelik hem de bağımlılık açısından sıkıntılıdır.
- Davranış temelli bir yaklaşım hem birliktelik hem de bağımlılık açısından idealdir.
- Kapsülleme sorumlulukları yerine getirmek için davranış ve veriyi bir araya getirir.
- Veri davranış için vardır.

Anemic Domain Model (Hastalıklı Alan Modeli)

- Hiç bir fonksiyonelliği olmayan içerisinde hiç bir fonksiyon barındırmayan sadece propertylerden oluşmuş bir modeldir ve düşünüldüğünde OOP'e aykırı bir yapı teşkil eder.
- Nesneler arasındaki bağımlılık doğrudan erişilerek yada get/set metotlarıyla veri üzerinden sağlanır.

Kaynaklar

- en.wikipedia.org
- <http://cagataykiziltan.net/>
- Akın Kaldıroğlu Ders Notları
- medium.com