TASARIM KALIPLARI

metin, ekran görüntüsü, diyagram, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir."Singleton Pattern" (Tekil Tasarım Deseni) kavramını açıklayan bir diyagramdır. Bu desen, bir sınıfın yalnızca bir örneğinin (instance) oluşturulmasını ve bu örneğe global bir erişim noktası sağlanmasını amaçlar.

**Private Constructor (Özel Kurucu):** Sınıfın dışından yeni örnekler oluşturulmasını engeller.

**Static Instance (Statik Örnek):** Sınıfın tek örneğini tutar.

**Public Static Method (Genellikle getInstance()):** Sınıfın tek örneğine erişim sağlar; eğer örnek henüz oluşturulmamışsa, oluşturur.

Kaç tip tasarım kalıbı var?"

1. **Creational - Oluşturma**  
   ➤ Yeni **nesnelerin nasıl oluşturulacağını** tanımlar.  
   ➤ Örnek: Singleton, Factory Method, Builder, Abstract Factory, Prototype
2. **Structural - Yapısal**  
   ➤ Farklı nesnelerin **birbirleriyle nasıl ilişkilendirileceğini** ve **daha büyük yapılar oluşturacağını** açıklar.  
   ➤ Örnek: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
3. **Behavioral - Davranışsal**  
   ➤ Nesneler **birbiriyle nasıl iletişim kurar**, **davranışlarını nasıl organize eder**, bunu tanımlar.  
   ➤ Örnek: Observer, Strategy, Command, Iterator, State, Mediator, Visitor, Chain of Responsibility

**1. Creational (Oluşturucu) Tasarım Kalıpları**

Nesne oluşturmayı soyutlayan kalıplardır.

* **Tanım**: Nesne oluşturma süreciyle ilgilidir. Amaç, bir nesne yaratılırken uygulamanın o nesnenin hangi sınıftan geldiğine bağımlı olmasını **engellemektir**.
* **Faydası**: İstemcinin (client) doğrudan nesne oluşturmak yerine, bu işin kalıplar aracılığıyla yapılmasını sağlar; böylece **bağımlılık azalır**, **esneklik artar**.
* **Örnekler**:
  + Singleton
  + Builder
  + Prototype
  + Abstract Factory
  + Factory Method

**🔹 2. Behavioral (Davranışsal) Tasarım Kalıpları**

Nesneler ve sınıflar arasındaki **iletişimi** düzenler.

* **Tanım**: Sınıf ve nesnelerin nasıl **etkileşime girdiğini** ve **sorumlulukların nasıl dağıtıldığını** belirler.
* **Faydası**: Karmaşık davranışları merkezi olmayan şekilde yönetir, **esnek ve yeniden kullanılabilir çözümler** sunar.
* **Örnekler**:
  + Template Method
  + Interpreter
  + Visitor
  + Mediator
  + Iterator
  + Command
  + Memento
  + Observer
  + Chain of Responsibility
  + State
  + Strategy *(aslında bu desen bazı kaynaklarda davranışsal olarak da kabul edilir, ancak yukarıdaki listede Structural kısmına yazılmış)*

**🔹 3. Structural (Yapısal) Tasarım Kalıpları**

Nesnelerin bir araya gelerek **daha büyük yapılar** oluşturmasını sağlar.

* **Tanım**: Sınıf ve nesneleri birleştirerek daha **karmaşık ama yönetilebilir yapılar** oluşturur.
* **Faydası**: Alt sistemlerin veya bileşenlerin **birbirine nasıl bağlanacağını** belirleyerek esneklik sağlar.
* **Örnekler**:
  + Decorator
  + Flyweight
  + Composite
  + Facade
  + Adapter
  + Proxy
  + Bridge

notların kapsamı dışında kalan ek desenler olabilir.

**Head First Design Patterns**

Değişen ne varsa onu al ve kapsülle, böylece kodunun geri kalan kısmı etkilenmesin

• Kod değişikliği az, esneklik fazla

HAS-A can be better than IS-A

metin, ekran görüntüsü, sayı, numara, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**🔷 Abstract Factory Nedir?**

**Abstract Factory**, **Creational (Oluşturucu)** tasarım kalıplarından biridir.

**Abstract Factory**, **ilgili nesne ailelerini** oluşturan bir **arayüz (interface)** sunar, ama bu nesnelerin **tam sınıflarını belirtmeden** oluşturulmasını sağlar.

Yani, **birbirine bağımlı nesneleri birlikte oluşturmak** istediğinde kullanılır.

**🔧 Ne İşe Yarar?**

* Farklı ürün aileleri (örneğin: Windows UI vs MacOS UI) arasında **bağımlılık olmadan geçiş yapmanı sağlar.**
* Kodun, oluşturduğu nesnelerin **somut türlerine bağlı kalmadan çalışmasını sağlar.**
* Yeni ürün ailelerini sisteme **minimum kod değişikliğiyle** ekleyebilirsin.

**📌 Ne Zaman Kullanılır?**

* Birbirine **bağımlı** nesne setleri birlikte kullanılacaksa.
* Uygulamanın çalıştığı ortam/platform değişebiliyorsa (örnek: Windows vs Mac vs Linux).
* Ürünlerin **türünü belirtmeden** üretmek istiyorsan.

**🧠 Özet:**

| **Özellik** | **Açıklama** |
| --- | --- |
| Kalıp Adı | Abstract Factory |
| Tür | Creational (Oluşturma) |
| Amaç | İlgili nesneleri, türlerini belirtmeden üretmek |
| Faydası | Bağımlılığı azaltır, kolay genişleme sağlar |
| Kullandığı Yapılar | Arayüzler, soyut sınıflar, alt sınıflar |
| Kullanım Durumu | GUI sistemleri, veritabanı bağlantıları, platforma özel üretimler |

Örnek :

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir. metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**✅ PROS – Avantajlar**

| **Avantaj** | **Açıklama** |
| --- | --- |
| **Isolation of concrete classes** | Somut sınıflar kullanıcıdan saklanır. Kullanıcı sadece interface kullanır. Bu da bağımlılığı azaltır. |
| **Exchanging Product Families easily** | Örneğin, birden fazla platforma (Windows/Mac/Linux) destek vermek kolaydır. Sadece uygun factory kullanılır. |
| **Promoting consistency among products** | Aynı “aileden” ürünler birlikte kullanılır. (örneğin hepsi Windows bileşeni) Uyumsuz bileşenler bir araya gelmez. |

**❌ CONS – Dezavantaj**

| **Dezavantaj** | **Açıklama** |
| --- | --- |
| **Difficult to support new kinds of products** | Yeni bir ürün (örneğin yeni bir buton türü) eklemek zordur çünkü tüm factory sınıflarına yeni metot eklemen gerekir. Bu da mevcut kodun değişmesine neden olur (Open/Closed prensibine aykırı olabilir). |

**🏭 Factory Method Nedir?**

**Factory Method**, bir **Creational (Oluşturucu)** tasarım desenidir ve amacı, nesne yaratma işlemini alt sınıflara bırakarak **kodda gevşek bağlılık (loose coupling)** sağlamaktır.

**Factory Method**, nesne oluşturma işini doğrudan new anahtar kelimesiyle yapmaz. Bunun yerine, bir **factory method** aracılığıyla hangi sınıfın örnekleneceğine **alt sınıf karar verir**.

**🎯 Amaç**

* **Nesne oluşturmayı soyutlamak**.
* Kodun, **hangi sınıfın** örneklendiğini **bilmeden** çalışmasını sağlamak.
* Yeni sınıflar eklenmesini kolaylaştırmak (esnek ve genişletilebilir yapı).

Bir üst sınıfın birden fazla alt sınıfı vardır ve girdiye bağlı olarak alt sınıflardan birini döndürmek için.

**✅ Avantajları**

| **Avantaj** | **Açıklama** |
| --- | --- |
| Kod, hangi nesnenin oluşturulduğunu bilmez | Yani Truck mı Ship mi bilmiyor |
| Yeni ürün eklemek kolaydır | Yeni sınıf + yeni alt Factory yazarsın |
| Bağımlılığı azaltır | Kod soyutlama üzerinden çalışır |

**❌ Dezavantajları**

| **Dezavantaj** | **Açıklama** |
| --- | --- |
| Sınıf sayısını artırır | Her ürün için ayrı bir sınıf ve factory gerekir |
| Basit uygulamalarda karmaşık kaçabilir | Gereksiz soyutlama olabilir |

**📌 Factory Method vs Abstract Factory**

| **Özellik** | **Factory Method** | **Abstract Factory** |
| --- | --- | --- |
| Amaç | Tek bir ürün nesnesi oluşturmak | Ürün ailesi oluşturmak |
| Ürün Sayısı | Genellikle 1 | Genellikle birçok ürün birlikte |
| Örnek | Truck, Ship | WindowsButton, WindowsCheckbox |

metin, diyagram, yazı tipi, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**👷 "The Creator is a class..."**

"The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method."

**Açıklama:**

* “Creator” sınıfı, ürünle ilgili işlemleri tanımlar (örneğin: sipariş, taşıma, fatura yazma gibi).
* Ancak **ürünü oluşturma kısmı** soyuttur (yani factoryMethod() tanımlanmaz, sadece bildirilir).
* Bu yöntem (factory method), alt sınıflarda tanımlanır.

**🧩 "All products must implement the same interface..."**

Tüm ürün sınıfları aynı arabirimi (interface) ya da soyut sınıfı kullanır. Böylece sistem, ürünlerin **tam olarak hangi sınıf olduğunu bilmeden** onlarla çalışabilir.

**Örnek:**

csharp

Kodu kopyala

interface ITransport

{

void Deliver();

}

**🧭 Şematik Yapı (Açıklamaları)**

| **Parça** | **Açıklama** |
| --- | --- |
| **Product** | Ortak arabirim / soyut sınıf (örn. ITransport) |
| **Creator** | Soyut sınıf – factoryMethod() burada tanımlanır (abstract olarak) |
| **ConcreteProduct** | Ürünün somut hali (örn. Truck, Ship) |
| **ConcreteCreator** | factoryMethod()'u override eden sınıf (örn. RoadLogistics) |

**📌 "The abstract factoryMethod()..."**

factoryMethod() soyut bir metottur. Bunu, **alt sınıflar (Concrete Creators)** implement eder.

**🧱 "The Concrete Creator is responsible..."**

Concrete Creator:

* **Hangi ürünün üretileceğini bilir.**
* factoryMethod() metodunu override ederek, bir **ConcreteProduct** üretir.

**Yani:** Bu sınıf ürünün nasıl üretileceğini bilen **tek yerdir**. Ana sistem (Creator) bu bilgiyi bilmez.

**🔁 Özetle**

Bu notta anlatılanlar:

| **Kavram** | **Açıklama** |
| --- | --- |
| **Amaç** | Nesne oluşturmayı soyutlayarak sistemin bağımlılığını azaltmak |
| **Factory Method()** | Creator sınıfında soyut, alt sınıflarda somut olarak tanımlanır |
| **Ürünler** | Ortak interface’e sahiptir (polimorfizm sağlanır) |
| **Concrete Creator** | Hangi ürünün nasıl oluşturulacağını bilen tek yerdir |
| **Avantaj** | Yeni ürün türleri eklemek kolay, kod esnek ve genişletilebilir hale gelir |

* **Factory Method**, bazen "**Sanal Yapıcı (Virtual Constructor)**" olarak adlandırılır.
* Çünkü new kullanmadan nesne yaratmayı sağlar ama yine de **dinamik olarak** doğru türde nesne üretir.

Factory Method deseninin, **kütüphanelerin çalışma mantığı** ile ilişkili olduğu söyleniyor.

**Açıklama:**

* Bir kütüphane (örneğin bir GUI ya da veritabanı kütüphanesi), **soyut sınıflar** tanımlar.
* Ama **hangi somut sınıfın** örnekleneceğini **uygulama** (client) belirler.
* Kütüphane ne zaman nesne yaratılması gerektiğini bilir ama neyin yaratılacağını bilmez. Bu ayrım önemlidir.

**talep üzerine** doğru "nesneyi" sağlar

public abstract class Creator

{

public abstract IProduct CreateProduct(); // Tek bir factory method

}

* **tek bir fabrika metodu** vardır: CreateProduct() gibi.
* Bu metod **alt sınıflar tarafından override edilir**,

public interface IGUIFactory

{

IButton CreateButton();

ICheckbox CreateCheckbox();

}

* Bu desen bir **factory nesnesi** üretir.
* Ve bu nesne, **birbirine ait birden fazla ürünü yaratmak için** birden fazla factory metoduna sahiptir.

**🔁 Fark Özet Tablosu**

| **Özellik** | **Factory Method** | **Abstract Factory** |
| --- | --- | --- |
| Yapı | Tek bir metod | Birden fazla metod |
| Ürün Sayısı | Genelde 1 | Birden fazla, ürün ailesi |
| Kullanım | Alt sınıf tek ürün yaratır | Nesne birden fazla ürün yaratır |
| Amaç | Yaratmayı alt sınıfa bırakmak | Ürün ailesini soyutlamak ve bir arada üretmek |

**Builder Design Pattern (Yapılandırıcı Tasarım Deseni)**, yazılım geliştirmede **karmaşık nesnelerin adım adım oluşturulmasını** sağlayan bir **creational (oluşturucu)** tasarım desenidir.

**🎯 Amaç:**

Karmaşık bir nesne oluştururken, oluşturma işlemini nesnenin temsilinden ayırmak.

**🔍 Ne zaman kullanılır?**

* Oluşturulacak nesne birçok parçadan oluşuyorsa.
* Aynı inşa süreciyle farklı türde nesneler oluşturmak istiyorsan.
* Nesne oluşturma süreci karmaşıksa (örneğin: birçok ayarı, bağımlılığı vs. varsa).

**📦 Builder Pattern Bileşenleri:**

1. **Builder (Soyut Arayüz):**
   * Nesnenin parçalarını oluşturacak adımları tanımlar.
2. **ConcreteBuilder (Somut Yapıcı):**
   * Bu adımları uygular ve sonucu verir.
3. **Product (Ürün/Nesne):**
   * Ortaya çıkan karmaşık nesnedir.
4. **Director (Yönetici):**
   * Builder’ı kullanarak nesneyi oluşturur.
   * Nesnenin nasıl oluşturulacağını bilir ama detayları bilmez.

**🧱 Örnek (Basit Pizza Builder):**

java

KopyalaDüzenle

// Product

class Pizza {

String dough;

String sauce;

String topping;

}

// Builder

interface PizzaBuilder {

void buildDough();

void buildSauce();

void buildTopping();

Pizza getPizza();

}

// ConcreteBuilder

class MargheritaPizzaBuilder implements PizzaBuilder {

private Pizza pizza = new Pizza();

public void buildDough() { pizza.dough = "Thin crust"; }

public void buildSauce() { pizza.sauce = "Tomato"; }

public void buildTopping() { pizza.topping = "Mozzarella"; }

public Pizza getPizza() { return pizza; }

}

// Director

class PizzaDirector {

private PizzaBuilder builder;

public PizzaDirector(PizzaBuilder builder) {

this.builder = builder;

}

public Pizza constructPizza() {

builder.buildDough();

builder.buildSauce();

builder.buildTopping();

return builder.getPizza();

}

}

**✅ Avantajları:**

* Nesne oluşturma sürecini soyutlar.
* Kod tekrarını azaltır.
* Farklı varyasyonlarda nesne üretimi kolaylaşır.
* Okunabilirliği artırır.

**⛔ Dezavantajları:**

* Küçük ve basit nesneler için gereksiz karmaşıklık yaratabilir.
* Fazladan sınıf (Builder, Director) ihtiyacı olabilir.

**🧩 Prototype nedir?**

Prototype, **var olan bir nesneden yeni bir nesne oluşturma** yöntemidir.  
Bu yöntem, **"yeni nesneyi sıfırdan oluşturmak yerine"**, mevcut bir nesnenin **kopyasını (clone)** alarak oluşturmayı sağlar.

**Prototype (Prototip) Tasarım Deseni**, bir nesnenin kopyasını oluşturmak için kullanılan bir **creational (oluşturucu)** tasarım desenidir.

**🎯 Ne zaman kullanılır?**

* Nesneler karmaşıksa (örneğin iç içe nesneler varsa).
* Nesne oluşturmak maliyetliyse (örneğin yapılandırması zaman alıyorsa).
* Aynı nesneden birçok benzer kopya gerekiyorsa.
* Her nesneyi new ile oluşturmak yerine, var olan bir prototipten çoğaltmak istiyorsan.

**🧠 Avantajları:**

* Nesne oluşturma işlemini hızlandırır.
* Karmaşık nesnelerin yeniden yapılandırılmasına gerek kalmaz.
* Nesneler arasında bağımlılığı azaltır.

C# dilinde ICloneable bir arayüzdür. Bu arayüzü uygulayan sınıflar, kendilerinden bir **kopya (clone)** oluşturabilir.

csharp

Kodu kopyala

public interface ICloneable

{

object Clone();

}

Yani Clone() metodu, bir nesnenin **kendisinin birebir kopyasını** döner.

**⚔️ Factory vs Prototype Karşılaştırma**

| **Özellik** | **Factory Pattern** | **Prototype Pattern** |
| --- | --- | --- |
| Oluşturma Yöntemi | Inheritance (Miras) | Delegation (Yetkilendirme/Kopyalama) |
| Alt Sınıf Gerekir mi? | Evet (subclassing) | Hayır |
| Başlangıç Nesnesi Gerekli mi? | Hayır | Evet (ilk prototip oluşturulmalı) |
| Kullanım | Türüne göre farklı sınıflardan üretim | Mevcut nesneden birebir ya da özelleştirilmiş klon |
| Esneklik | Yeni türler için yeni sınıflar gerekir | Mevcut nesne üzerinden esnek üretim sağlanır |

**🔚 Özetle:**

* **Factory:** Yeni türler yaratmak için alt sınıflar tanımlarsın.
* **Prototype:** Var olan bir nesneyi kopyalayarak yeni nesneler oluşturursun; mirasa gerek yoktur ama bir **örnek nesne (prototip)** olması gerekir.

**Singleton (Tek Nesne)**, bir **creational (yaratımsal)** tasarım desenidir ve amacı **uygulama boyunca yalnızca bir tane** nesne oluşturulmasını sağlamaktır.

**🧩 Singleton Nedir?**

Singleton deseni, bir sınıfın yalnızca **bir örneğe (instance)** sahip olmasını **garanti eder**  
ve bu örneğe global (her yerden) erişim sağlar.

**🎯 Ne zaman kullanılır?**

* Tüm uygulama boyunca **tek bir veri tabanı bağlantısı**, **tek bir yapılandırma yöneticisi**, **tek bir oturum**, **tek bir logger** gibi şeyler gerektiğinde.
* **Kaynakların paylaşılması** gereken durumlarda.
* Aynı nesneden birden fazla oluşturulmasının sistemde sorun çıkaracağı yerlerde.

**🔒 Temel Amaç:**

* Tek bir örnek yaratmak
* Global erişim sağlamak
* Yeni nesneler yaratılmasını **engellemek**

**“Bir nesne tek bir instance’a sahip olur. (LOCK)”**

* **Amaç:** Uygulama boyunca bir sınıftan yalnızca **tek bir nesne** (instance) oluşturulsun.
* **LOCK:** Çoklu iş parçacıkları (multi-threading) varsa, aynı anda birden fazla nesne **oluşmasını engellemek** için lock (kilitleme) yapılır.

**🔧 “Oluşturmayı kilitle”**

Bu, **aynı anda birden fazla iş parçacığı çalışırken** sadece **bir tanesinin nesne oluşturmasına izin ver** demektir.

**❓ NASIL?**

**✅ 1. Private Constructor**

* Dışarıdan new Singleton() çağrısını engellemek için constructor **private yapılır.**

private Singleton() { }

**✅ 2. Static Private Instance**

* Nesne saklanacak yer: sadece bir tane olacak ve **sınıfa ait olacak (static)**.

private static Singleton instance;

**✅ 3. Public Instance Property / Method ({get;} veya GetInstance())**

* Nesneye erişmek için dışarıdan çağrılabilen bir yapı gerekir.

public static Singleton GetInstance()

{

if (instance == null)

{

instance = new Singleton();

}

return instance;

}

Ya da:

public static Singleton Instance => instance;

**🔄 Özet Şema**

| **Adım** | **Açıklama** |
| --- | --- |
| private constructor | Dışarıdan new ile yeni nesne oluşturulamaz |
| static private instance | Tek bir örnek saklanır |
| public static GetInstance() | Tek örneği dışarı verir |
| lock | Aynı anda iki iş parçacığının oluşturmasını engeller |

**✅ Kısaca:**

“Bir sınıfın sadece bir tane örneği (instance) olmasını garanti altına almak için, **oluşturmayı kilitlersin**, dışarıdan erişimi **kontrollü şekilde** sağlarsın.”

**🧩 Adapter Nedir? (Yapısal / Structural Tasarım Deseni)**

**Adapter**, farklı iki sistemin **birlikte çalışmasını sağlamak için araya giren bir dönüştürücü**dür.

Basitçe: **“Ben bu eski sistemi kullanamam, ama sen onu benim anlayacağım hale getir.”**

**🎯 Ne İşe Yarar?**

* Var olan bir sınıfın arayüzünü, başka bir sınıfın beklediği arayüze çevirir.
* **Uyumsuz yapıları uyumlu hale getirir.**
* Eski sistemleri yeni sistemle entegre etmek için kullanılır.

**🏢 Örnek Problem: Firma Birleşmesi (Migros + Kipa)**

**🎬 Senaryo:**

* Migros, Kipa'yı satın aldı.
* Kipa'nın:
  + Müşteri kayıt sistemi
  + Personel veri tabanı
  + Web API'si
  + Mağaza kod yapısı... **Migros sistemiyle uyumsuz.**

**❓ Asıl Problem Ne?**

“Ben Migros’un sistemiyim, ama Kipa’nın verilerini direkt kullanamam. **Uyumsuzlar.**”

**✅ Çözüm: Adapter**

“Kipa sisteminden gelen verileri al, Migros sistemine **uygun formata dönüştür**.”

**Bu noktada:**

* Bir **interface** tanımlarsın (örneğin IMusteriKayit)
* Migros bu interface'i kullanır.
* Kipa'nın sistemine özel bir **Adapter sınıfı** yazarsın → KipaAdapter : IMusteriKayit
* Bu sınıf, Kipa'nın verisini alır, **Migros'un formatına dönüştürür.**

**🧱 Adapter Deseninde Yapılar:**

| **Yapı** | **Görevi** |
| --- | --- |
| **Client** | Yeni sistem – örneğin Migros |
| **Adaptee** | Uyumsuz/Eski sistem – örneğin Kipa sistemi |
| **Target (Interface)** | Yeni sistemin beklediği arayüz |
| **Adapter** | Arayüz ile eski sistemi bağlayan köprü sınıfı |

**🧩 Bridge (Köprü) Tasarım Deseni Nedir?**

**Bridge**, bir sınıf ile onun davranışlarını (özelliklerini) **ayrı ayrı geliştirilebilecek şekilde ayıran** bir **yapısal (structural)** tasarım desenidir.

Amaç: **Uygulama (kullanan taraf) ile uygulamanın detaylarını (nasıl çalıştığı) birbirinden ayırmak.**  
Böylece biri değişirse diğerini etkilemez. (low coupling)

**🎯 Ne Zaman Kullanılır?**

* Uygulamanın **kullanım şekli (arayüz)** ile **uygulamanın davranışı (altyapı)** birbirinden **bağımsız olarak geliştirilsin** isteniyorsa.
* Birden çok **platform, ortam, cihaz** gibi değişkenler varsa.
* **Sınıf sayısı patlamasın**, esneklik sağlansın diye.

**📦 Örnek Problem:**

**Müşteri bildirimleri + Firma duyuruları yayınlamak istiyoruz.**

**📌 Sorunlar:**

* Bildirim **nereden yayınlanacak** belli değil: Eposta, SMS, Mobil, Web, WhatsApp...
* Bildirim **neden yayınlanıyor** belli değil: Kampanya, Acil Durum, Genel Bilgi, İndirim...
* Eğer ortam veya içerik değişirse **tüm yapıyı baştan yazmak mı gerekecek?** ❌

**✅ Çözüm: Bridge Deseni**

Kullanım ve özellikleri **iki ayrı hiyerarşide geliştir.**  
Mesajın *ne olduğunu* bir sınıf temsil eder.  
Mesajın *nasıl gönderildiğini* başka bir sınıf.

**🧱 Yapı**

| **Rol** | **Açıklama** |
| --- | --- |
| **Abstraction** | Kullanıcıya görünen temel sınıf (örneğin Mesaj) |
| **RefinedAbstraction** | Özelleştirilmiş kullanım (KampanyaMesaji, GenelDuyuru) |
| **Implementor** | Soyut davranış – örneğin gönderim şekli (IGonderim) |
| **ConcreteImplementor** | Gerçek uygulama – SmsGonder, EpostaGonder, PushNotification |

**📌 Avantaj:**

Yeni bir **mesaj türü** (örneğin AcilDurumMesaji) veya yeni bir **gönderim yöntemi** (örneğin PushBildirim) eklemek için diğerlerini değiştirmene gerek yok.

**✅ Sonuç Olarak:**

| **Durum** | **Bridge Ne Sağlar** |
| --- | --- |
| Ortam Değişebilir | Yeni Implementor yaz, diğerlerini değiştirme |
| Davranış Değişebilir | Yeni RefinedAbstraction yaz, diğerlerini etkilemez |
| Kod tekrarını azaltır | Sınıf patlamasını önler, yapı modüler kalır |

**📌 Cümle 1:**

**Adapter: how to run designed things.**  
**Açıklama:**

* Adapter zaten **önceden tasarlanmış ve var olan sistemleri** çalıştırmak (uyarlamak) için kullanılır.
* Yani: “**Zaten var olan bir şeyi, benim sistemimde nasıl kullanabilirim?**”
* Bu yüzden **geriye dönük uyumluluk sağlar.**

**📌 Cümle 2:**

**Bridge: Run before they were designed.**  
**Açıklama:**

* Bridge deseni, sistemi **en baştan esnek olacak şekilde tasarlamak** içindir.
* Yani: “**İleride ne geleceğini bilmiyorum, esnek yapayım**” gibi düşün.
* Uygulama **önceden düşünülerek** iki farklı sorumluluğa (arayüz & uygulama) ayrılır.

**📌 Cümle 3:**

**Bridge: run time-binding of the implementation**  
**Açıklama:**

* Bridge sayesinde hangi gerçek uygulamanın kullanılacağı **çalışma zamanında (runtime)** belirlenebilir.
* Örneğin: SmsGonderici mi kullanılacak, yoksa EmailGonderici mi — bu kararı **kod çalışırken** verebilirsin.

**🤝 Kıyas Tablosu:**

| **Özellik** | **Adapter** | **Bridge** |
| --- | --- | --- |
| Amaç | Uyumsuz yapıları birleştirmek | Soyutlama ve gerçekliği ayırmak |
| Kullanım zamanı | Mevcut sistemi değiştirmeden | Yeni sistemleri esnek kurmak için |
| Zamanlama | **Tasarım sonrası** | **Tasarım sırasında** |
| Bağlama (binding) | Derleme zamanı (compile-time) | **Çalışma zamanı (run-time)** |
| Örnek | Eski API'yi yeni sistemde kullanmak | Farklı bildirim yollarını desteklemek |

**🔑 Sonuç:**

* **Adapter**, eskiyi yenide kullanmak için çözüm sağlar (geç uyum).
* **Bridge**, yapıyı baştan esnek kurgular (önceden uyum).
* **Bridge daha soyut** ve genişletilebilirken, **Adapter daha pratik ve dar kapsamlıdır.**

🌳 COMPOSITE (Bileşik) Tasarım Deseni Nedir?

Composite, bir ağaç yapısını temsil eden sınıflar arasında, birey (yaprak) ile grup (dallanma) nesneleri aynı şekilde işlemek için kullanılan bir yapısal (structural) tasarım desenidir.

**📌 Örnek Problemler Ne Anlatıyor?**

* “Tüm yorumları listelemek istiyorum.”  
  ➤ Yorumun altına yorum yapılabilir. Yani **yorum içinde yorumlar olabilir**.
* “Dosya sistemini taramak istiyorum.”  
  ➤ Klasörler içinde **dosyalar da olabilir**, başka klasörler de olabilir.
* “Takipçimin takipçisinin takipçileri…”  
  ➤ Her kullanıcı, başka kullanıcıları takip eder; **kendi içlerinde yapı tekrar eder**.

Bunların hepsi **ağaç yapıları**dır.

**🧠 Asıl Problem Ne?**

* Birbirine **bağlı veri yapıları (treeview)**.
* **Parent-child ilişkileri** var.
* Ve bu yapı **tekrar eden yapıların kendini çağırarak çalışmasını** gerektiriyor.

**✅ Çözüm: COMPOSITE Deseni**

**Amaç:**  
Yaprak (son düğüm) ve dal (alt düğüm içeren) nesnelerini **aynı arayüz üzerinden işleyebilmek.**

**🧱 Yapısı:**

| **Rol** | **Açıklama** |
| --- | --- |
| **Component** | Ortak arayüz veya soyut sınıf |
| **Leaf** | Alt elemanı olmayan nesne (örnek: Dosya) |
| **Composite** | Alt eleman içeren nesne (örnek: Klasör) |

**📌 Gerçek Hayatta Composite Kullanımı**

| **Senaryo** | **Composite Yapısı** |
| --- | --- |
| Yorum sistemi | Yorum > Alt yorumlar |
| Menü – Alt menüler | Menü > Alt menü öğeleri |
| Klasör ve dosya sistemi | Klasör > Dosya/Klasör |
| Organizasyon şeması | Müdür > Alt personeller |
| Sosyal ağ takip ilişkileri | Kullanıcı > Takipçiler |

**✅ Özet**

* **Tree-like veri yapıları** varsa,
* Yaprak ve grup nesnelerini **ayrı ayrı değil, aynı şekilde** işlemek istiyorsan,
* Yapı **kendini tekrar ediyorsa** (recursive),

➡ **Composite pattern** en doğru çözümdür.

**🎨 Decorator Nedir?**

* **Decorator**, bir nesnenin işlevselliğini (özelliklerini) **dinamik olarak**, yani **çalışma zamanında** artırmak için kullanılan yapısal (structural) tasarım desenidir.
* Nesneye yeni sorumluluklar eklemek için, **nesneyi sarmalar (wrap)** ve ek davranışlar katarsın.
* Bu yöntem, **kalıtım (inheritance)** ile yeni sınıflar oluşturmaya alternatif, daha esnek ve modüler bir yaklaşımdır.

**📌 Örnek Problemler ve Anlatılanlar:**

**1. Fiat 500L Otomobil Satış Uygulaması**

* Temel bir model var (Fiat 500L).
* Ancak; opsiyonel paketler (örneğin klima, navigasyon, güvenlik paketi) fiyatı etkiler.
* İndirim varsa fiyatı tekrar güncellemek gerekir.
* Bu opsiyonel özellikleri **dinamik olarak eklemek istiyoruz.**

➡ Dekoratörler sayesinde:

* Temel araca "paket" gibi davranan küçük "süsleyici"ler (decorators) eklenebilir.
* Her biri fiyatı günceller, toplam fiyat ortaya çıkar.
* Yeni özellik eklemek için temel sınıfı değiştirmeye gerek kalmaz.

**2. Blog / Sosyal Medya Yorumları**

* Temel yorum var (text).
* Yorumlara emoji, resim, video gibi farklı içerikler dinamik olarak eklenebilir.
* Tüm bu özellikler tek bir sınıfta kalabalık yapmadan, **birebir** birleştirilebilir.

**🧩 Temel Özellikler**

* **Decorator**, orijinal nesneye yeni davranışlar ekler.
* Yeni davranışlar, orijinal nesne çalışmadan önce veya sonra olabilir (before/after).
* Nesneleri sarmalayarak (wrap) **zorunlu olmayan ek özellikler** dinamik eklenir.
* **Kalıtım yerine** kullanılır, çünkü kalıtım statik ve serttir, decorator esnek ve dinamik.

**💡 Örnek Analojisi**

Bir kahve siparişi veriyorsun.

* Temel kahve var.
* Üzerine süt ekleyebilirsin.
* Üzerine şurup ekleyebilirsin.
* Üzerine krem şanti ekleyebilirsin.

Her ek özellik, kahvenin fiyatını ve tadını değiştirir. Ama kahveyi yeniden tanımlamazsın, sadece **dekorasyon (süsleme)** yaparsın.

**Özetle:**

| **Konu** | **Açıklama** |
| --- | --- |
| **Amaç** | Nesneye dinamik olarak yeni sorumluluklar eklemek. |
| **Nasıl?** | Nesneyi bir decorator sınıfıyla sarmalayarak. |
| **Neden?** | Kalıtımın esnek olmaması yerine daha esnek çözüm. |
| **Örnek** | Otomobil opsiyonları, sosyal medya yorumları, kahve siparişi. |

**Adapter vs Decorator Karşılaştırması**

| **Özellik** | **Adapter** | **Decorator** |
| --- | --- | --- |
| **Amaç** | Farklı **arayüzleri uyumlu hale getirmek** (interface dönüşümü) | Mevcut arayüzü **geliştirmek/ek özellikler eklemek** (enhance interface) |
| **Kullanım durumu** | İki uyumsuz arayüzü birbirine bağlamak için | Nesnelere dinamik olarak yeni sorumluluklar eklemek için |
| **Alt sınıf ilişkisi** | Subclass (alt sınıf) üzerinden çalışır, farklı arayüzü adapte eder | Alt sınıflandırma yapmadan, nesneye yeni özellikler ekler |
| **Nesne sorumlulukları** | **Sorumluluğu değiştirmek veya uyarlamak** | **Yeni sorumluluklar veya davranışlar eklemek** |
| **Yapısal odak** | Arayüz uyumluluğu ve dönüşümü | Nesnenin işlevselliğini sarmalayıp genişletmek |
| **Örnek** | Eski API’yi yeni API’ye uyarlamak | Bir nesneye ekstra özellikler veya davranışlar katmak (örneğin, loglama, filtreleme) |

**Özet**

* **Adapter**, farklı arayüzlere sahip nesneleri bir araya getirmek için kullanılır.
* **Decorator**, nesnenin arayüzünü değiştirmeden, **aynı arayüzü kullanarak** yeni işlevsellikler ekler.

**Adapter:** “Bu nesne benim beklediğim arayüze uymuyor, nasıl kullanırım?”  
**Decorator:** “Bu nesne gayet güzel, ama ben ona biraz daha özellik eklemek istiyorum.”

**Facade Nedir?**

**Facade**, yapısal (structural) tasarım desenlerinden biridir ve karmaşık sistemlerin kullanımını **basitleştirmek** için kullanılır.

* Karmaşık bir sistemin birçok alt sistemi olabilir.
* Bu alt sistemlerin her biri farklı arayüzlere ve işlemlere sahiptir.
* Facade, tüm bu karmaşık alt sistemlerin işlevlerini kapsayan **tek bir yüksek seviyeli arayüz (interface)** sağlar.
* Böylece, kullanıcı ya da başka bir sistem, karmaşık iç detaylarla uğraşmak zorunda kalmadan işlemleri kolayca yapabilir.

**Senin Verdiğin Örnek:**

**Problem:**

* Webden satış yapmak istiyorsun.
* Sipariş almak, fatura düzenlemek, kargoyu yönetmek gibi farklı aşamalar var.
* Bunların hepsini ayrı ayrı mı yapacaksın?
* Tek bir arayüzden (interface) bu işlemleri kolayca yapabilir miyiz?

**Facade Çözümü:**

* Sipariş alma, fatura düzenleme, kargo gönderme gibi farklı alt sistemleri içeren karmaşık işlemler var.
* Facade bu alt sistemleri **tek bir arayüzde toplar**.
* Sen sadece bu Facade arayüzünü kullanırsın.
* Böylece alt sistemlerin karmaşıklığı saklanır, kullanım basitleşir.

**Kısaca Facade**

| **Özellik** | **Açıklama** |
| --- | --- |
| Amaç | Karmaşık alt sistemleri basitleştirmek |
| Sağladığı | Tek, basit ve yüksek seviyeli arayüz |
| Kullanımı | Kullanıcı sadece Facade’yi çağırır, alt sistemlerle doğrudan uğraşmaz |
| Faydası | Kod okunabilirliği ve bakım kolaylığı |

**Adapter vs Facade**

| **Özellik** | **Adapter** | **Facade** |
| --- | --- | --- |
| **Amaç** | Var olan farklı arayüzleri **birbirine uyumlu hale getirmek** (interface dönüşümü) | Karmaşık sistemleri **tek, basit bir arayüzle** kullanıma sunmak |
| **Kullanım Durumu** | İki farklı arayüz veya API’yi birleştirmek | Birden fazla alt sistemi tek bir arayüzde toplamak |
| **Arayüz Tasarımı** | Tasarım, mevcut bir arayüze uyum sağlamak için yapılır | Basitleştirilmiş, genellikle kendi başına bir arayüz sağlar |
| **Singleton Olma Durumu** | Genellikle değildir | Genellikle **Singleton** olur (tek Facade objesi) |
| **İşlev** | Arayüz dönüşümü | Kullanımı kolaylaştırma ve karmaşıklığı gizleme |

**Abstract Factory Nedir?**

* **Abstract Factory**, **benzer veya ilişkili nesneler ailesi** yaratmak için kullanılan creational (yaratıcı) tasarım desenidir.
* Factory Method’un daha genel ve kapsamlı halidir.
* Birbirine bağlı nesneleri gruplar halinde üretir, böylece nesnelerin uyumlu olmasını sağlar.

Flyweight (Hafif Nesne) tasarım deseni, **çok sayıda benzer nesne oluşturulması gereken durumlarda hafıza kullanımını optimize etmek için** kullanılır.

**Flyweight Nedir?**

* Çok sayıda nesne aynı veya benzer içeriğe sahipse, bu nesnelerin ortak özellikleri **paylaşılır**.
* Böylece her nesne için ayrı ayrı veri tutmak yerine, ortak veriler tek bir yerde saklanır.
* Nesnelerin yalnızca değişen özellikleri ayrı tutulur.
* Bu sayede **bellek kullanımı azalır**, performans artar.

**Senin Verdiğin Örnek:**

**Problem:**

* Oyun nesneleri, kullanıcılar, emojiler, etiketler, gönderiler gibi çok fazla nesne var.
* Her nesne için ayrı ayrı veri tutarsak hafıza çok kullanılır.
* Sürekli yeni nesne oluşturmak pahalıdır.

**Çözüm:**

* Ortak özellikleri (örneğin, emoji resmi, oyun karakterinin tipi) bir kere oluştur.
* Bu ortak özellikleri **paylaş**.
* Farklı nesneler sadece kendine özgü küçük bilgileri tutar.

**Özet:**

| **Özellik** | **Açıklama** |
| --- | --- |
| Amaç | Belleği ve kaynakları verimli kullanmak |
| Nasıl | Ortak verileri paylaşarak |
| Kullanım Alanı | Çok sayıda benzer nesnenin gerektiği durumlar (oyunlar, grafik, metin düzenleme, sosyal medya emojileri vb.) |
| Avantajı | Hafıza tasarrufu, performans artışı |

**Basit Analogi:**

Bir oyun sahnesinde 1000 tane ağaç var diyelim.  
Her ağaç için ayrı yaprak, gövde gibi veri tutmak yerine, aynı türdeki ağaçların ortak verilerini paylaşırız. Böylece hafıza azalır.

* **Hafıza** kullanımı çok fazla oluyor.
* **Sürekli yeni nesne oluşturmak** pahalı ve verimsiz (zaman, bellek açısından).  
  Yani, çok sayıda benzer nesneyi ayrı ayrı oluşturmak kaynakları tüketiyor.

**Çözüm Ne?**

* Bir nesneyi **bir kez oluştur**, sonra tekrar ihtiyaç duyulduğunda aynı nesneyi kullan (paylaş).
* Eğer nesne daha önce oluşturulmamışsa, oluştur ve kaydet.
* Bu sayede aynı nesnenin birden çok kopyası yerine, **referansları/gölge kopyaları** kullanılır.

**Flyweight Özeti**

* **Flyweight** deseni, çok sayıda benzer nesneyi **paylaşarak** oluşturur, böylece bellek kullanımı azalır.
* Nesnelerin ortak özellikleri paylaşılır, farklı özellikler ise dışarıdan verilir.

**İlişkili Tasarım Desenleri**

* **Prototype Pattern:**  
  Nesnelerin **klonlanması**dır (shallow copy, yani sığ kopya). Nesneyi kopyalayarak yeni nesne üretir. Burada her seferinde yeni bir nesne oluşturulur.
* **Factory Pattern:**  
  Nesne yaratma işini merkezi olarak yapar, yeni nesne yaratmak için sanal (virtual) yöntem sağlar. Nesne yaratma mantığını soyutlar.

**Burada Ne Anlatılmış?**

* Flyweight, **yeni nesne oluşturmayı minimize etmek için aynı nesneyi paylaşır** (referans kullanır).
* Prototype, nesneyi klonlayarak yeni nesne yaratır (dolayısıyla nesneler ayrı olur).
* Factory, nesne yaratmayı yönetir ama nesne paylaşımı ya da klonlama yapmaz.

**Kısaca:**

| **Desen** | **Nesne Oluşturma Yöntemi** | **Nesne Paylaşımı** |
| --- | --- | --- |
| Flyweight | Nesneyi **bir kez oluştur**, tekrar kullan (paylaş) | Evet (referans paylaşımı) |
| Prototype | Nesneyi **klonla** (kopya oluştur) | Hayır (her seferinde yeni nesne) |
| Factory | Nesne yaratma işlemini merkezi olarak yap | Hayır (paylaşım değil, yaratım) |

**🔧 Proxy Nedir?**

Proxy, bir nesneye doğrudan erişmek yerine onun yerine çalışan **aracı (vekil) bir nesne** sağlar.

**🧩 Örnek Problem:**

* Bir oyun sahnesinde farklı seviyeler arası geçiş yapmak istiyorsun. Ama her seviye büyük ve yüklemesi zaman alıyor.
* Banka uygulamasında ödeme bilgisi doğrulama gibi işlemler yavaş olabilir.
* Büyük veri işlemleri, grafik hesaplamaları zaman alabilir.

**🎯 Asıl Problem Ne?**

* Bu işlemler:
  + **Yavaş** çalışıyor.
  + **Uzak sunuculara** bağlantı gerektiriyor.
  + Uygulamanın **donmasına**, kullanıcıya **gecikmeli yanıt verilmesine** yol açabilir.

**✅ Çözüm Ne?**

* Bu yavaş ve ağır işi **başka bir yapı (Proxy)** üstlensin.
* Gerçek nesneye erişmeden önce **Proxy** devreye girer.
* Asıl nesne hazır olana kadar uygulama **çalışmaya devam eder**.

Yani:

❗ "Gerçek işi yapan nesneye doğrudan değil, onun vekiline eriş."

**🧠 Teknik Açıklama:**

* Proxy genellikle **asenkron (asynchronous)** çalışır.
* Gerçek nesneye erişim **kontrollü**, **güvenli** veya **ertelenmiş** olur.

**📦 Proxy Türleri:**

| **Tür** | **Açıklama** |
| --- | --- |
| **Virtual Proxy** | Büyük nesneleri ihtiyaç duyulana kadar yüklemez. (örnek: resim yükleme) |
| **Remote Proxy** | Uzak bir sunucudaki nesneye yerel gibi erişim sağlar. |
| **Protection Proxy** | Erişim kontrolleri yapar. (kim girebilir, kim giremez) |
| **Smart Proxy** | Ek işlemler yapar: önbellek, sayaç, loglama gibi. |

**🎮 Oyun Örneği:**

Her seviye büyük grafik içeriyor. Hepsini baştan yüklemek sistemi kasar.  
**Proxy**, sadece geçilen seviyeyi yükler. Diğerleri gerektiğinde (istek gelince) yüklenir.

**💳 Banka Örneği:**

Kullanıcı ödeme yaparken, ödeme işlemi sunucuda olur.  
**Proxy**, bu işlemi alır, sunucuya gönderir, yanıt gelene kadar kullanıcıya arayüzü kilitlemez.

**Kısaca:**

| **Özellik** | **Açıklama** |
| --- | --- |
| Amaç | Gerçek nesneye erişimi kontrol etmek veya geciktirmek |
| Avantaj | Performans, güvenlik, yönetilebilirlik |
| Tür | Yapısal (Structural) Tasarım Deseni |
| Kullanım Alanı | Uzak erişim, ağır işlemler, asenkron yapılar |

**Chain of Responsibility (Sorumluluk Zinciri) Tasarım Deseni**dir. İşte detaylı açıklaması:

**🔗 Chain of Responsibility (Sorumluluk Zinciri) Nedir?**

Bir isteği bir nesne zinciri boyunca iletme imkanı tanır. Her nesne, isteği işleyebilir ya da zincirdeki bir sonraki nesneye iletebilir.

**🎓 Örnek Problem: Mezuniyet İşlemleri**

**Kullanıcı Ne Yapmak İstiyor?**

* Mezun olmak için işlemleri tamamlamak istiyor.

**Aşamalar:**

1. SKS Onayı (Sosyal Kültürel İşler)
2. Kütüphane borç kontrolü
3. Öğrenci işleri kontrolü
4. Bölüm başkanı onayı

**Asıl Problem Ne?**

* Her aşama sırasıyla ve **birbirinden bağımsız** kontrol edilmeli.
* Her biri sadece kendi işinden sorumlu.
* Önceki adım onaylanmadan sonraki başlatılmamalı.
* Karmaşık if-else yerine daha esnek bir yapı gerekli.

**✅ Çözüm: Chain of Responsibility**

Her sorumlu birim, kendisinden sonraki birime **bağlanır**. Gelen istek:

* Her birime sırayla iletilir,
* İlgili birim uygunsa işlem yapar,
* Gerekirse sıradakine devreder.

**🔄 Nasıl Çalışır?**

1. **Bir "Handler" sınıfı** tanımlanır. Bu sınıf bir isteği işler ve bir sonraki "Handler"a geçiş yapabilir.
2. Her onay bir "Handler" nesnesidir.
3. İşlem sırası zincir şeklinde kurulur: SKS → Kütüphane → Öğrenci İşleri → Bölüm

Eğer biri isteği işleyemezse, "Sıradaki işlesin" der.

**🧠 Yapısal Avantajları**

| **Avantaj** | **Açıklama** |
| --- | --- |
| ✨ Esneklik | Yeni birim eklemek kolay |
| 🔁 Sıra kontrolü | İşlem sırası belirlenebilir |
| 👤 Ayrık sorumluluk | Her adım yalnızca kendi işinden sorumlu |
| 🧼 Temiz kod | Büyük if-else bloklarına gerek kalmaz |

**🎯 Gerçek Hayatta Nerede Görülür?**

* Form doğrulama adımları
* İş akışı yönetimi
* Yetkilendirme (authorization) zincirleri
* Teknik destek sistemleri (destek taleplerini sırayla yönlendirme)

**🛠 Kısaca**

| **Özellik** | **Açıklama** |
| --- | --- |
| Desen Türü | Behavioral (Davranışsal) |
| Amaç | İşlemleri bir zincir şeklinde sırayla ve sorumluluk ayrımıyla yürütmek |
| Örnek | Mezuniyet işlemleri, teknik destek, iş akış adımları |

**Command (Komut) Design Pattern**, bir işlemi (eylemi) nesneye dönüştürmeyi sağlayan **davranışsal (behavioral)** bir tasarım desenidir. Bu sayede komutlar:

* **Kuyruğa alınabilir**,
* **Geri alınabilir (undo)**,
* **Kaydedilebilir**,
* Ve **çağırıcı (invoker)** ile **eylem yapan (receiver)** birbirinden bağımsız hale gelir.

🧱 Yapı (Bileşenler):

Rol Açıklama

Command Bir arayüz veya soyut sınıf. Örn: ICommand { void Execute(); }

ConcreteCommand Gerçek komutu temsil eder. Örn: LightOnCommand

Receiver Asıl işi yapan sınıf. Örn: Lamba

Invoker Komutu çağıran sınıf. Örn: Kumanda

Client Komutları oluşturur ve bağlar.

**✅ Avantajları:**

* İstekleri nesne olarak ele alma
* Komutları sıraya koyma, kaydetme, geri alma
* Çağıran ve uygulayanı birbirinden ayırır (loose coupling)

**🧮 Örnek Problem:**

Hesap makinesi yapmak istiyorum  
5 + 4 - 7 × 2 ÷ (5 + 9)

Bu ifadeler birçok farklı işlemi içeriyor: toplama, çıkarma, çarpma, bölme. Ayrıca:

* Her biri farklı şekilde çalışıyor.
* Bir kuyruğa alınıp sırayla çalıştırılabilir.
* Her işlem bağımsız olarak tanımlanabilir mi?

**❓ Asıl Problem Ne?**

* Tüm işlemler birbirinden farklı ve **bağımsız şekilde çalışmalı**.
* İşlem sırası önemli olabilir, dolayısıyla bu işlemleri sırayla **kuyrukta** saklamak da gerekebilir.
* Ancak istemci (örneğin "hesap makinesi arayüzü") bu işlemlerin **nasıl çalıştığını bilmek zorunda kalmamalı**.
* Yani, “Bu + işlemi, şu da / işlemi” diye if-else’lerle kontrol edilmemeli. **Kötü tasarım olur.**

**✅ Çözüm: Command Pattern (Komut Deseni)**

**Command deseni**, bir isteği (örneğin topla(), çarp(), böl()) bir nesne olarak sarmalar. Bu sayede:

* İstekler (komutlar) nesneye dönüşür: ToplaKomutu, BölKomutu gibi.
* Bu komutlar bir liste veya kuyrukta saklanabilir.
* Uygulamak için sadece komut.çalıştır() çağrılır.
* Arka planda nasıl çalıştığını bilmeye gerek kalmaz.

**Interpreter (Yorumlayıcı) Tasarım Deseni**, bir dili (basit bir söz dizimini veya komutları) **yorumlamak (anlamlandırmak)** için kullanılan **behavioral (davranışsal)** bir tasarım desenidir.

diyagram, çizgi, tasarım içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**Iterator (Yineleyici) Tasarım Deseni**, bir koleksiyonun (liste, ağaç, dizi, vb.) elemanlarına **sıralı şekilde ve iç yapısını bilmeden** erişmeyi sağlayan **behavioral (davranışsal)** bir tasarım desenidir.

**🧠 Tanım:**

Iterator deseni, bir koleksiyondaki elemanlara **tek tek erişmek** için **standart bir yol sağlar**, böylece koleksiyonun iç yapısı (array mi, linked list mi, ağaç mı...) kullanıcı tarafından bilinmek zorunda kalmaz.

**🎯 Neden Kullanılır?**

* Bir koleksiyonda gezinmek için for ya da foreach dışında **kontrollü, esnek bir yöntem** gerekir.
* **Aynı koleksiyon üzerinde birden fazla gezinme şekli** uygulanmak istenebilir (örneğin, ileri-geri).
* Koleksiyonun yapısı değişse bile gezinme şekli değişmemeli.

**🧠 Mediator (Arabulucu) Deseni Nedir?**

**Mediator (Davranışsal / Behavioral)** tasarım deseni, **nesneler arasındaki karmaşık iletişimi merkezi bir yapı üzerinden yönetmeyi** sağlar. Yani nesnelerin **birbirleriyle doğrudan iletişim kurmasını engelleyerek** bu iletişimi bir arabulucuya (mediator) devreder.

**✈️ Örnek Senaryo: Havaalanı Kontrol Merkezi**

* Uçaklar doğrudan birbirleriyle konuşmaz.
* Hepsi sadece **kontrol kulesiyle iletişim kurar**.
* Kontrol kulesi de **ne zaman hangi uçak inecek, hangisi kalkacak** bunu koordine eder.

🔁 Böylece:

* Her uçak diğerlerini **bilmek zorunda kalmaz**.
* **İletişim merkezi ve düzenlidir.**

**🎯 Neden Kullanılır?**

* Nesneler arası **aşırı bağlantıyı azaltmak** için.
* Sistemdeki iletişimi merkezi bir yerden kontrol etmek.
* Her bileşenin **yalnızca bir aracıyla konuşması**, sistemi daha esnek hale getirir.

🧠 Memento Pattern (Hatıra Deseni) Nedir?

Davranışsal (Behavioral) bir desendir.

Bir nesnenin iç durumunu (state) dışarıdan erişilemeyecek şekilde (encapsulation bozulmadan) kaydetmek ve gerektiğinde geri yüklemek (restore) için kullanılır.

* Nesnenin iç durumunu kaydet.
* Geri yüklenebilme imkanı sunar.
* **Nesnenin gizliliğini (encapsulation) bozmadan** bu işlemi yap.

**💡 Kullanım Alanları:**

* Geri al (Undo) / ileri al (Redo) işlemleri
* Oyunlarda kayıt sistemi
* Kod editörlerinde geçmiş yönetimi
* Form doldurulurken geçici veri saklama

**✅ Avantajlar:**

* **Encapsulation** bozulmaz (iç veri gizliliği korunur).
* Eski durumlara kolay dönüş yapılabilir.

**❌ Dezavantajlar:**

* Çok fazla durum saklanırsa **hafıza tüketimi artabilir**.
* Geri alma sistemleri karmaşıklaşabilir.

**💡 Kullanım Alanları:**

* Geri al (Undo) / ileri al (Redo) işlemleri
* Oyunlarda kayıt sistemi
* Kod editörlerinde geçmiş yönetimi
* Form doldurulurken geçici veri saklama

**✅ Avantajlar:**

* **Encapsulation** bozulmaz (iç veri gizliliği korunur).
* Eski durumlara kolay dönüş yapılabilir.

**❌ Dezavantajlar:**

* Çok fazla durum saklanırsa **hafıza tüketimi artabilir**.
* Geri alma sistemleri karmaşıklaşabilir.

**Observer** (Gözlemci) Tasarım Deseni, davranışsal (behavioral) bir desendir ve şu şekilde çalışır:

📌 Tanım:

Bir nesnede bir değişiklik olduğunda, bu değişikliği ilgilenen diğer nesnelere otomatik olarak bildiren tasarım desenidir.

“Bir nesne değiştiğinde, onu izleyenleri haberdar et.”

**🧠 Genel Tanım:**

* **Observer Pattern**, bir nesnenin (Subject) durumu değiştiğinde, bu nesneye bağlı diğer nesnelerin (Observers) otomatik olarak bilgilendirilmesini sağlar.
* **“Bir değişiklik varsa, bunu izleyen herkese haber ver.”**
* **Davranışsal (Behavioral)** bir tasarım desenidir.
* Bu desen **birden-çoğa (one-to-many)** bağımlılık ilişkisi tanımlar

🔹 Subject Arayüzü:

Subject yani gözlemlenen nesne şunları sağlar:

 **registerObserver**: Yeni bir gözlemci abone olur.

 **removeObserver**: Abonelikten çıkar.

 **notifyObservers**: Tüm gözlemcilere bildirim gönderir.

**🔹 Observer Arayüzü:**

Observer yani **gözlemci nesneler** şunu yapar:

Subject bir değişiklik olduğunda bu update() metodunu çağırarak gözlemciyi bilgilendirir.

🔹 DisplayElement Arayüzü (Ekstra):

Genellikle Observer olan nesnelerin, bilgiyi kullanıcıya nasıl gösterdiğini belirtir.

**🧱 ConcreteSubject ve ConcreteObserver**

**🔹 WeatherData (ConcreteSubject):**

* Hava durumu bilgilerini tutan sınıf.
* measurementsChanged() metodu, veri güncellendiğinde tüm gözlemcilere bildirim gönderiyor.

**Mediator** ve **Observer** desenlerinin karşılaştırması yapılmış ve aralarındaki temel fark açıklanmış. Aşağıda sade bir şekilde ne anlatıldığını özetliyorum:

**🔄 Genel Fark**

**🔹 Mediator:**

* **Amaç**: Nesneler arasındaki **iletişimi merkezi bir yapıya (aracıya)** devrederek karmaşayı azaltmak.
* **Tüm nesneler**, doğrudan birbirleriyle konuşmaz → Bunun yerine, **tek bir Mediator (aracı)** ile konuşurlar.
* **İletişim tek bir merkezde toplanır.**
* **Nesneler birbirinden habersizdir**, sadece aracıdan bilgi alır/verir.

**🔹 Observer:**

* **Amaç**: Bir nesnedeki değişimi, ona bağlı olan tüm diğer nesnelere **otomatik dağıtmak (bildirim)**.
* Bu dağıtımı yapmak için **Subject ve Observer** nesneleri tanımlanır.
* **Nesneler birbirini tanır** (Subject → Observer’ları bilir).
* Bildirim dağılımı (event propagation) gibi çalışır.

**🎯 MVC Bağlantısı:**

“The ‘View’ part of Model-View-Controller.”

* MVC deseninde:
  + **Model**, verileri temsil eder.
  + **View**, bu verilerin kullanıcıya gösterilmesini sağlar.
  + **Controller**, kullanıcı etkileşimlerini yorumlar ve modeli değiştirir.

➡️ **Observer deseni**, genellikle **Model → View** arasındaki ilişkiyi kurmak için kullanılır.  
Model değiştiğinde, bağlı tüm View’lar otomatik olarak güncellenir. Bu yüzden "Observer, MVC'nin View kısmıdır" deniyor.

**📌 Özet:**

| **Özellik** | **Mediator** | **Observer** |
| --- | --- | --- |
| Amaç | İletişimi merkezi bir yere toplamak | Değişiklikleri bağlı nesnelere yaymak |
| Yapı | Nesneler → Mediator → Nesneler | Subject → Observer(lar) |
| Bağlantı | Zayıf bağlı nesneler, Mediator’a bağımlı | Observer’lar Subject'e kayıtlıdır |
| Kullanım Yeri | UI bileşenleri arası iletişim | Veri değişimi – bildirim sistemi |
| MVC Rolü | Controller veya koordinasyon | View (güncellenen kısım) |

**🎯 Strategy Pattern Nedir?**

Strategy Pattern, bir işlemi (algoritmayı) gerçekleştirmek için birden fazla yol (strateji) varsa:

* Bu algoritmaları **ayrı sınıflarda tanımlar** (encapsulation),
* Bu sınıfları **birbirlerinin yerine geçebilecek şekilde tasarlar** (interchangeable),
* Böylece algoritma seçimini **çalışma zamanında (runtime)** yapabilmeni sağlar.

**💡 Özet Tanım:**

**"Bir algoritmalar ailesi tanımlar, her birini kapsüller ve bunları birbirinin yerine kullanılabilir hale getirir. Algoritma, onu kullanan sınıflardan bağımsız olarak değişebilir."**

**🧠 Neden Kullanılır?**

* Kodun **esnekliğini artırmak** için.
* Algoritmaların **kodun geri kalanından bağımsız** değiştirilebilmesi için.
* if-else veya switch-case yerine daha temiz ve genişletilebilir bir yapı kurmak için.

**🧩 Temel Yapı:**

* **Context**: Stratejiyi kullanan sınıf.
* **Strategy (interface/abstract)**: Algoritma için ortak arayüz.
* **ConcreteStrategyA/B**: Gerçek algoritmalar.

**✅ Avantajları:**

* Algoritmalar kolayca değiştirilebilir.
* Yeni strateji eklemek için mevcut kodu değiştirmene gerek kalmaz (Open/Closed Principle).
* if-else yapılarının yerine daha bakımı kolay nesne tabanlı yapı sağlar

**🧠 State Pattern Nedir?**

Bir nesnenin **davranışı**, **iç durumuna (state)** bağlıysa ve bu davranış **durum değiştikçe değişiyorsa**, State Pattern kullanılır.

**🎯 Amaç:**

**Bir nesnenin davranışlarını, onun içsel durumuna göre değiştirmek.** Sanki nesne, sınıfını değiştirmiş gibi davranır.

**📦 Ne Zaman Kullanılır?**

* Nesne **birden fazla duruma** sahipse.
* Davranışı, **bu durumlara göre değişiyorsa**.
* if-else veya switch yerine daha **bakımı kolay bir yapı** isteniyorsa.

**🔄 Strategy vs State**

| **Strategy Pattern** | **State Pattern** |
| --- | --- |
| Algoritmalar arasında geçiş | Davranışlar arasında geçiş |
| Kullanıcı tarafından seçilir | İçsel duruma göre otomatik değişir |
| Genellikle dışsal kontrol vardır | Davranış nesne içinde kontrol edilir |

**🧩 Temel Yapı:**

* **Context**: Duruma göre davranışı değişen sınıf.
* **State (interface)**: Ortak davranışları tanımlar.
* **ConcreteStateA/B**: Her durumun davranışını tanımlar.

**✅ Avantajları:**

* if-else karmaşasından kurtulursun.
* Yeni durumlar eklemek kolaydır.
* Her durumun davranışı kendi sınıfında kapsüllenir.

**🔨 Creational Patterns (Yaratımsal Desenler)**

Amaç: **Nesne oluşturma sürecini kontrol etmek**, böylece sistemi nesnelerden bağımsız ve daha esnek hâle getirmek.

| **Pattern** | **Açıklama** |
| --- | --- |
| **Factory Method** | Alt sınıflar üzerinden nesne oluşturur. Arayüz bilgisine göre hangi sınıfın örneği üretileceğine karar verir. |
| **Abstract Factory** | Farklı sınıf ailelerinden nesneler üretir. Gerçek sınıfları belirtmeden. |
| **Builder** | Karmaşık nesneleri adım adım oluşturur. Aynı inşa süreciyle farklı temsiller oluşturulabilir. |
| **Prototype** | Kopyalanabilir (cloneable) nesneler yaratır. Mevcut bir nesneden yeni bir tane üretir. |
| **Singleton** | Sistem boyunca **tek bir örnek** barındıran sınıftır. Küresel erişim sağlar. |

**🧱 Structural Patterns (Yapısal Desenler)**

Amaç: **Sınıflar ve nesneler arası ilişkileri kurarak** daha büyük yapılar oluşturmaktır.

| **Pattern** | **Açıklama** |
| --- | --- |
| **Adapter** | Uyuşmayan arayüzleri bir araya getirir. İki farklı sınıfın birlikte çalışmasını sağlar. |
| **Bridge** | Arayüz ve uygulamayı ayırır. Böylece bağımsız değişebilirler. |
| **Composite** | Bileşenlerden ve alt bileşenlerden oluşan ağaç yapısı kurar (örneğin menüler). |
| **Decorator** | Nesnelere **çalışma anında (runtime)** ek özellikler ekler. Kalıtım yerine tercih edilir. |
| **Facade** | Karmaşık sistemleri tek bir sınıfla sadeleştirir. Kullanımı kolaylaştırır. |
| **Flyweight** | Hafıza tasarrufu için **paylaşılan küçük nesneler** kullanır. |
| **Proxy** | Gerçek nesne yerine kullanılan vekil nesnedir. Erişim kontrolü, gecikmeli yükleme gibi amaçlarla kullanılır. |

**🤝 Behavioral Patterns (Davranışsal Desenler)**

Amaç: Nesneler arasındaki iletişimi, sorumlulukları ve algoritmaları düzenlemektir.

| **Pattern** | **Açıklama** |
| --- | --- |
| **Interpreter** | Mini bir dil tanımı varsa, gramer yapısını çözümlemek için kullanılır. |
| **Template Method** | Ana algoritma yapısı üst sınıfta tanımlanır, alt sınıflar adımları doldurur. |
| **Chain of Responsibility** | İstek, uygun olan nesneye kadar zincir halinde dolaşır. |
| **Command** | Eylemler, nesne hâlinde temsil edilir. Kuyruk, geri alma (undo), tekrar (redo) gibi işlemler yapılabilir. |
| **Iterator** | Koleksiyonlar içinde sırayla gezinmeyi sağlar, iç yapıyı bilmeye gerek yoktur. |
| **Mediator** | Nesneler doğrudan değil, merkez bir nesne (arabulucu) aracılığıyla haberleşir. |
| **Memento** | Bir nesnenin iç durumunu kaydeder ve sonra geri almayı (restore) sağlar. |
| **Observer** | Bir nesnedeki değişiklik, ona bağlı diğer nesnelere otomatik olarak bildirilir. |
| **State** | Nesne duruma göre davranış değiştirir. Yeni duruma geçtiğinde, yeni davranışlar gösterir. |
| **Strategy** | Algoritmaları kapsülleyip birbirinin yerine kullanılabilir hale getirir. |
| **Visitor** | Sınıflara dokunmadan, dışarıdan yeni işlevler eklenmesini sağlar. |

**🧠 Genel Karşılaştırma**

| **Kategori** | **Amaç** |
| --- | --- |
| **Creational** | Nesne üretimini kontrol et. |
| **Structural** | Nesneleri/bileşenleri birleştir ve düzenle. |
| **Behavioral** | Nesnelerin iş birliğini ve davranışlarını düzenle. |