Tasarım Desenleri

NESNEYE YÖNELİK PROGRAMLAMA KAVRAMLARI

DR. ŞAFAK KAYIKÇI

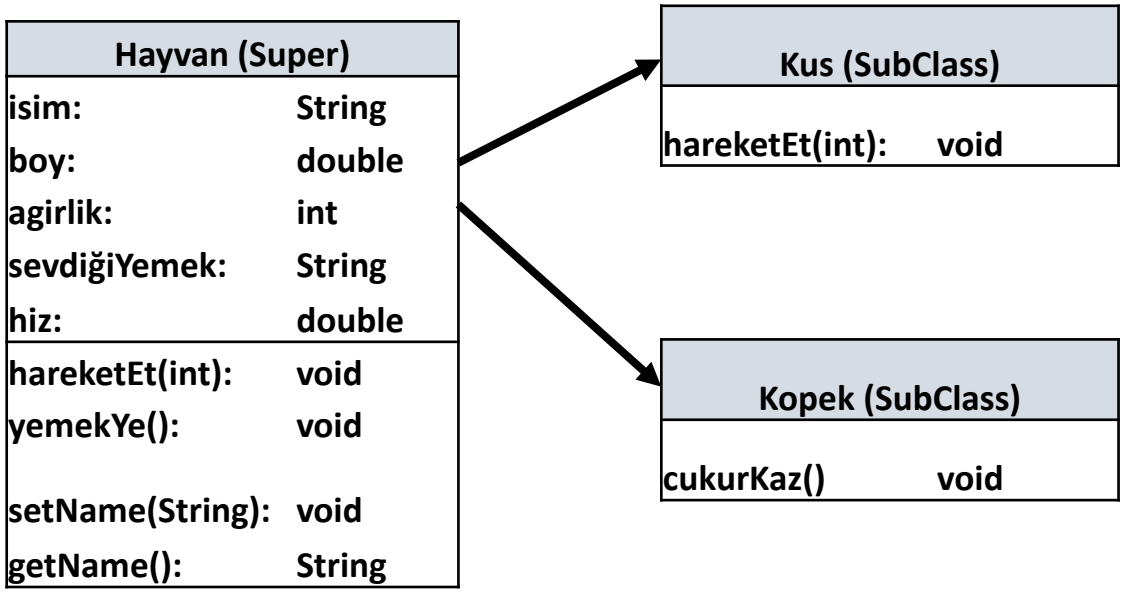
Sınıf (Class)

- Taslak yada tasarı planı olarak tanımalanabilir
- Özellikler
  - Nelere sahip olduğu
- Metotlar (Fonksiyonlar)
  - Neler yapabildiği (eylemler)

Hayvan	
İsim:	String
boy:	double
agirlik:	int
sevdiğiYemek:	String
hiz:	double
hareketEt(int):	void
yemekYe():	void
setName(String):	void
getName():	String

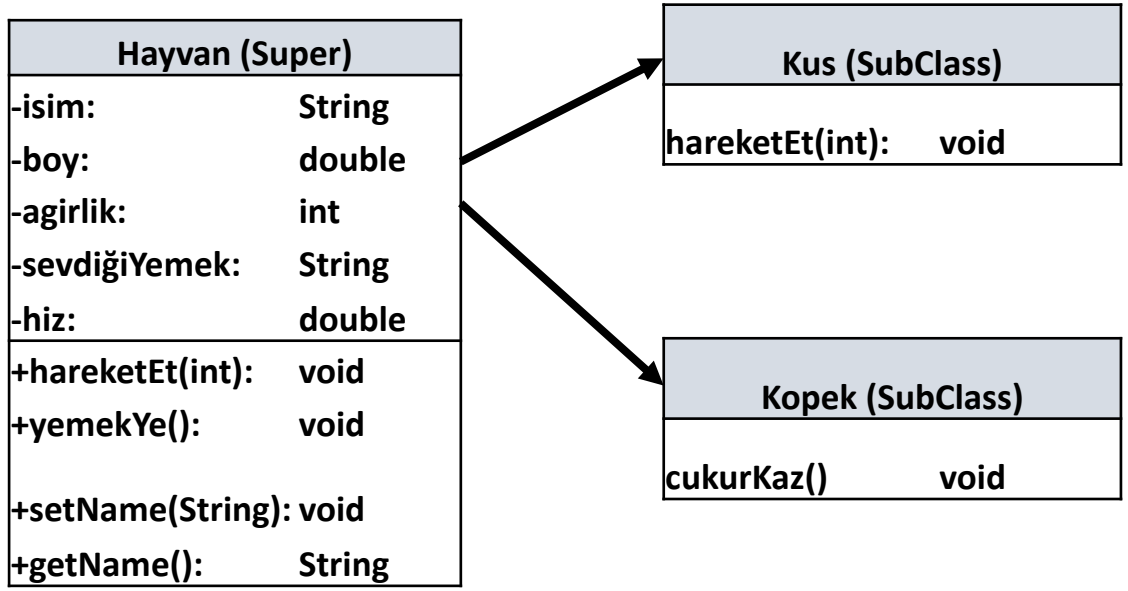
Kalıtım (Inheritance) (is a özelliği)

- Sınıfların ortak özellikleri nelerdir ?
- Bu özellikler soyutlanır ve duruma göre ezilir (override)
- Kod tekrarını engeller ve ana sınıfta yapılan değişiklikler alt sınıflara yansır.



Kapsülleme (Encapsulation)

```
class Kopek{

String isim;

int agirlik;

setAgirlik (int yeniAgirlik){
   if(yeniAgirlik>0){
                agirlik = yeniAgirlik;
      }
  else{
         //throw error
      }

 }

}
```



- private ve public özellikleri ayırmak gereklidir.
- get ve set kullanmak gerekir.

Polymorphism (Çok Şekilcilik)

- Çokbiçimlilik (Polymorphism) bir nesnenin farklı amaçlar için de kullanılabileceği anlamına gelir

```
Hayvan kopek = new Kopek();
Hayvan kedi = new Kedi();
```

- Farklı alt sınıfları bir dizi altında toplayabilirsiniz.

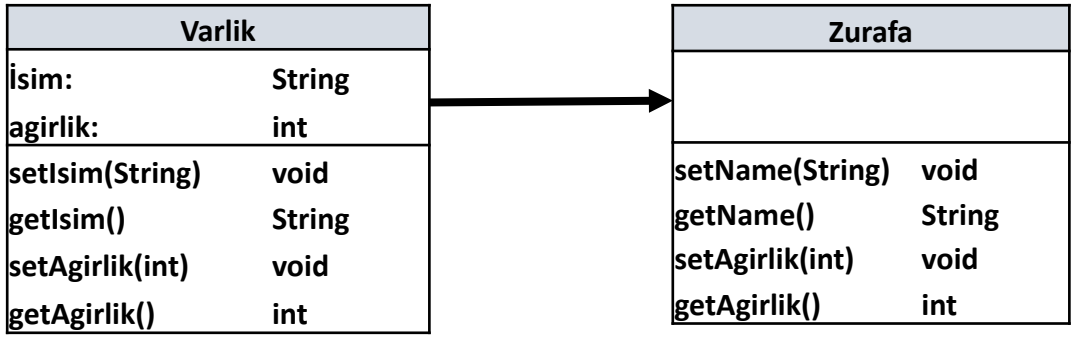
Hayvanlar Dizisi
Kedi Objesi
Kopek Objesi

- Ana sınıfta olmayan bir özelliği kullanamazsınız. Çevirim (cast) işlemi yapmalsınız.

```
((Kopek) kopek).cukurKaz();
```

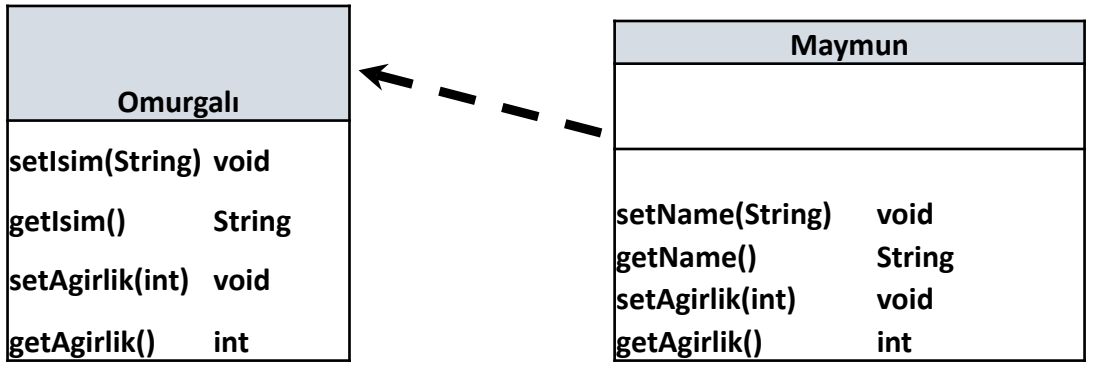
Soyut Sınıflar (Abstract Class)

- Soyut sınıflar diğer sınıflara taban olmak için kullanılırlar
- Nesne türetemezler
- Bütün metotların soyut olmasına gerek yoktur.
- static metotlar içerebilir.



Arayüzler (interface)

- Arayüz ,interface anahtar sözcüğü ile tanımlanır. Arayüz abstract metotlar içerir, gövdeleri yoktur.
- Yalnızca public ve ön-tanımlı (default) erişim belirtkesi alabilir.
- Bir sınıf birden çok arayüze çağırabilir. Böylece çoklu kalıtım sağlanır (multiple inheritance)



Tasarım Desenleri

GİRİŞ

DR. ŞAFAK KAYIKÇI

Tasarım Desenleri

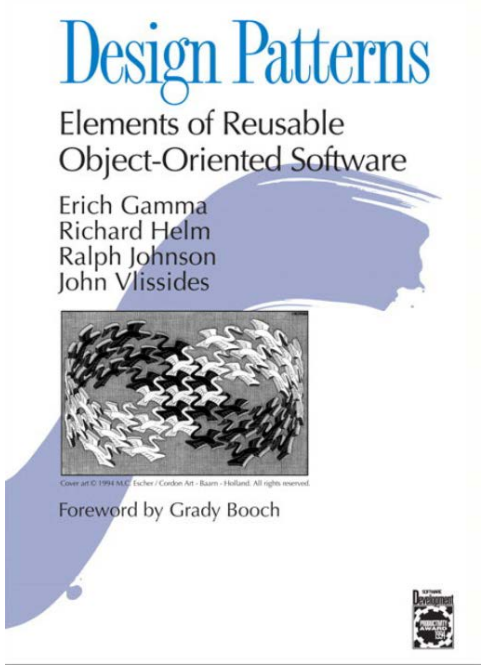
- Bir tasarım desenleri, belirli bir problemi çözmek için daha önceden başarıyla kullanılmış bir çözümlerdir. Bu çözümler, oldukça uzun bir süre boyunca sayısız yazılım geliştiricisi tarafından deneme yanılma yoluyla elde edilmiştir.
- Nesne yönelimli tasarım problemlerini çözmek için programlama dilinden bağımsız stratejilerdir.
- Tasarım desenlerini kullanarak kodunuzu daha esnek, yeniden kullanılabilir ve bakımı yapılabilir hale getirebilirsiniz.
- Java'nın kendisi de tasarım desenleri kullanılırak yapılmıştır.
- Bir uygulamanın tasarımında şeffaflık sağlarlar.
- Yazılım Geliştirme Yaşam Döngüsünün gereksinim analizi ve tasarım aşamasında kullanılır.

Gang of Four (GOF) - Dörtlü Çete

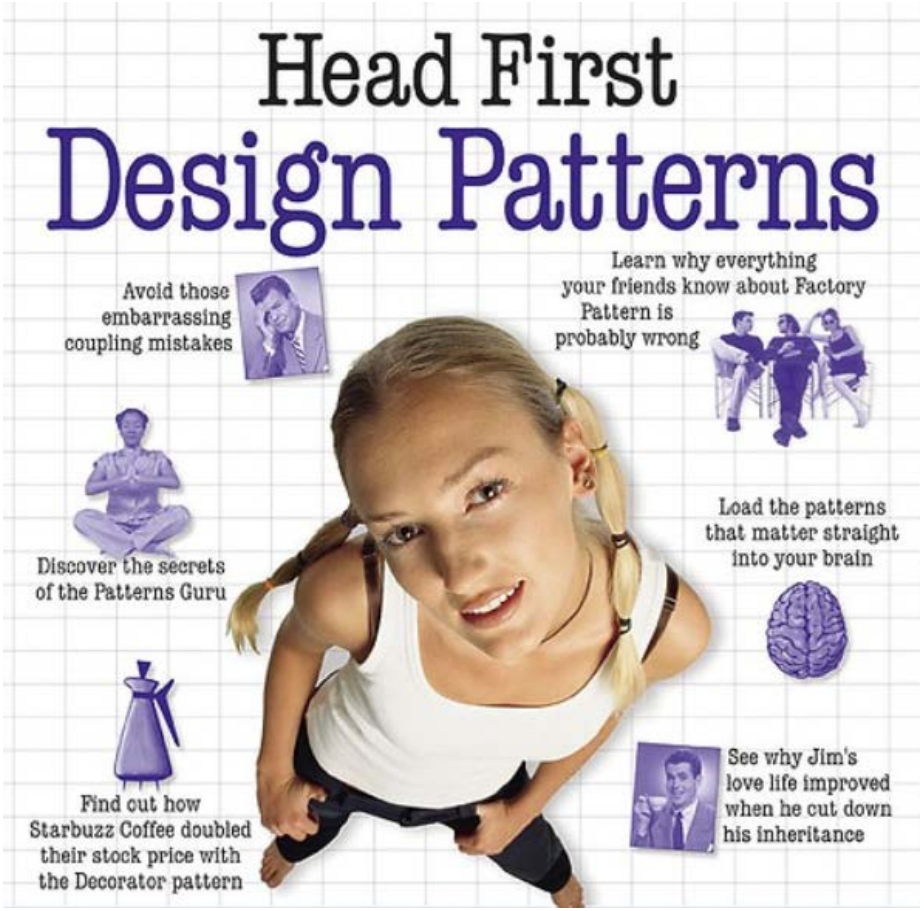
1994 yılında dört yazar Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides, yazılım geliştirmede tasarım deseni kavramını başlatan "Design Patterns - Elements of Reusable Object-Oriented Software" başlıklı bir kitap yayınladı. Bu yazarlara göre tasarım desenleri öncelikle aşağıdaki nesne yönelimli tasarım ilkelerine dayanmaktadır :

1. Uygulama yerine daha çok arayüz kullanımına yönelik program yazma
2. Kalıtım yerine nesne kompozisyonunu tercih etme

(L-R) Ralph, Erich, Richard and John



Tavsiye Kitap



1.Creational Design Pattern (Yaratıcı Tasarım Desenleri)

Factory Pattern

Abstract Factory Pattern

Singleton Pattern

Prototype Pattern

Builder Pattern

2. Structural Design Pattern (Yapısal Tasarım Desenleri)

Adapter Pattern

Bridge Pattern

Composite Pattern

Decorator Pattern

Facade Pattern

Flyweight Pattern

Proxy Pattern

3. Behavioral Design Pattern (Davranışsal Tasarım Desenleri)

Chain Of Responsibility Pattern

Command Pattern

Interpreter Pattern

Iterator Pattern

Mediator Pattern

Memento Pattern

Observer Pattern

State Pattern

Strategy Pattern

Template Pattern

Visitor Pattern

Tasarım Desenleri

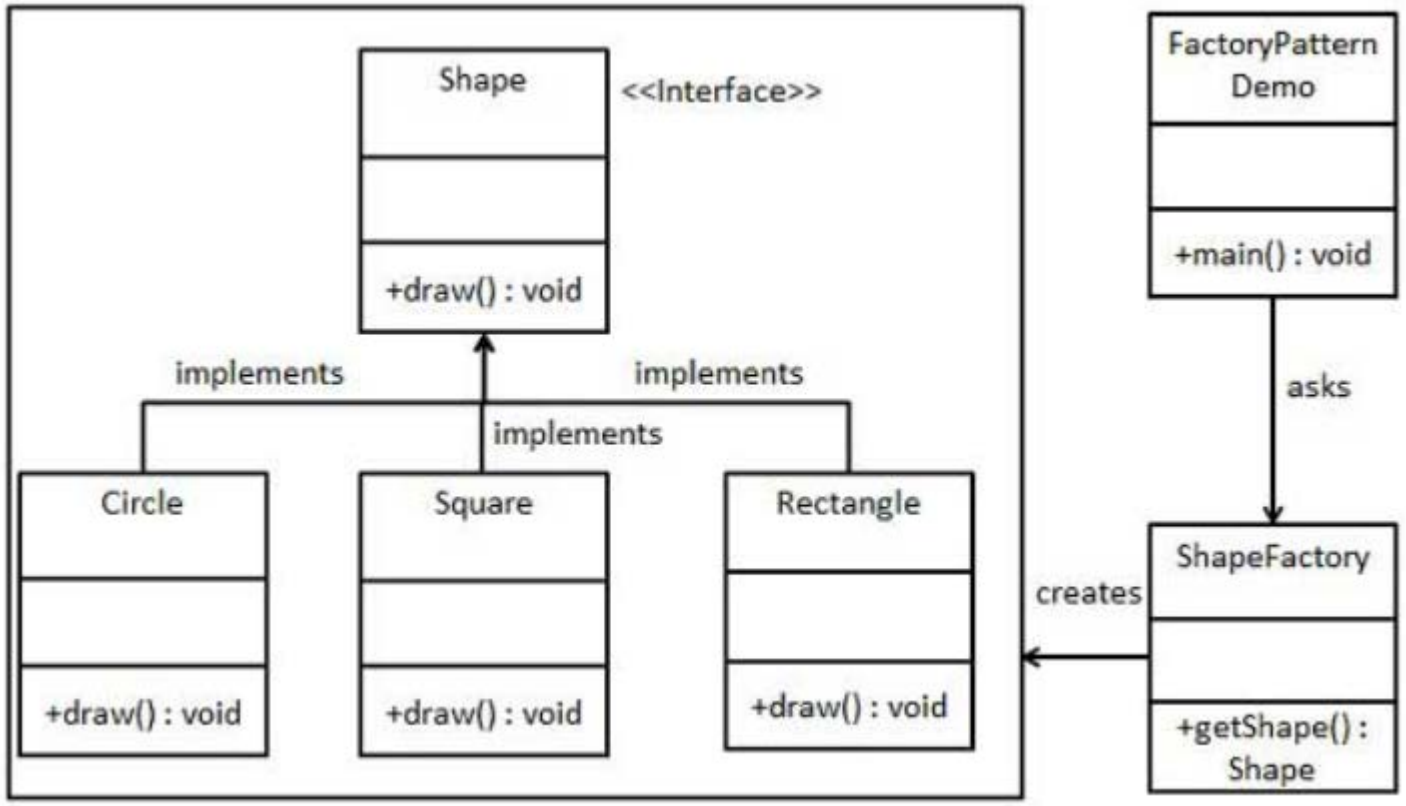
FACTORY PATTERN – ABSTRACT FACTORY PATTERN

DR. ŞAFAK KAYIKÇI

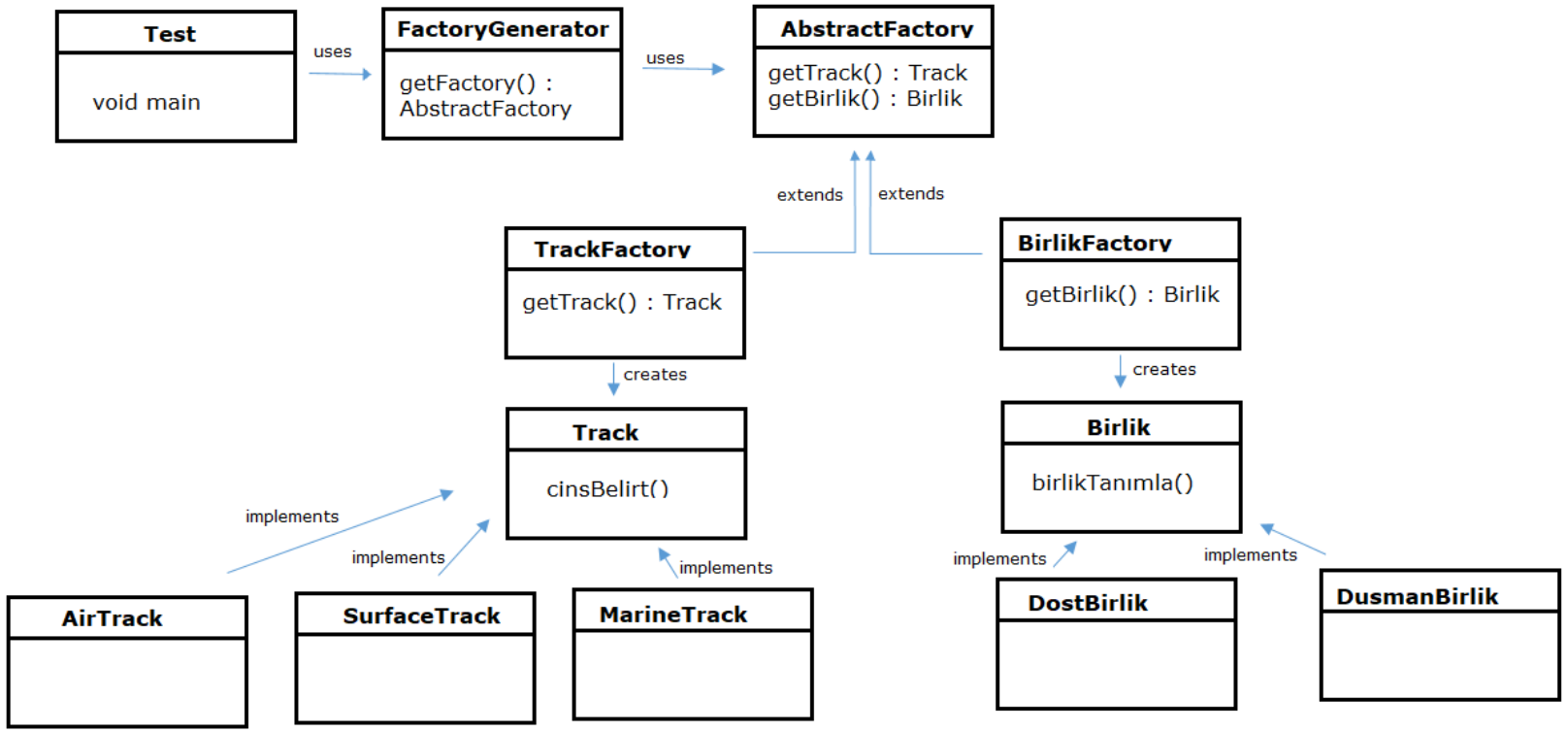
Factory Pattern – AbstractFactory Pattern

- Yaratıcı Tasarım Desenleri içerisinde yer almaktadır.
- Factory Pattern Alt sınıfların oluşturulacak nesne türünü seçmesini sağlar.
- Factory design pattern'de tek bir ürün ailesine ait tek bir arayüz mevcutken,abstract factory'de farklı ürün aileleri için farklı arayüzler mevcuttur.
- Birden fazla ürün ailesi ile çalışmak durumunda kaldığımızda , ürün ailesi ile istemci tarafını soyutlamak için kullanılır.

Tutorialspoint



Abstract Factory - Example



Tasarım Desenleri

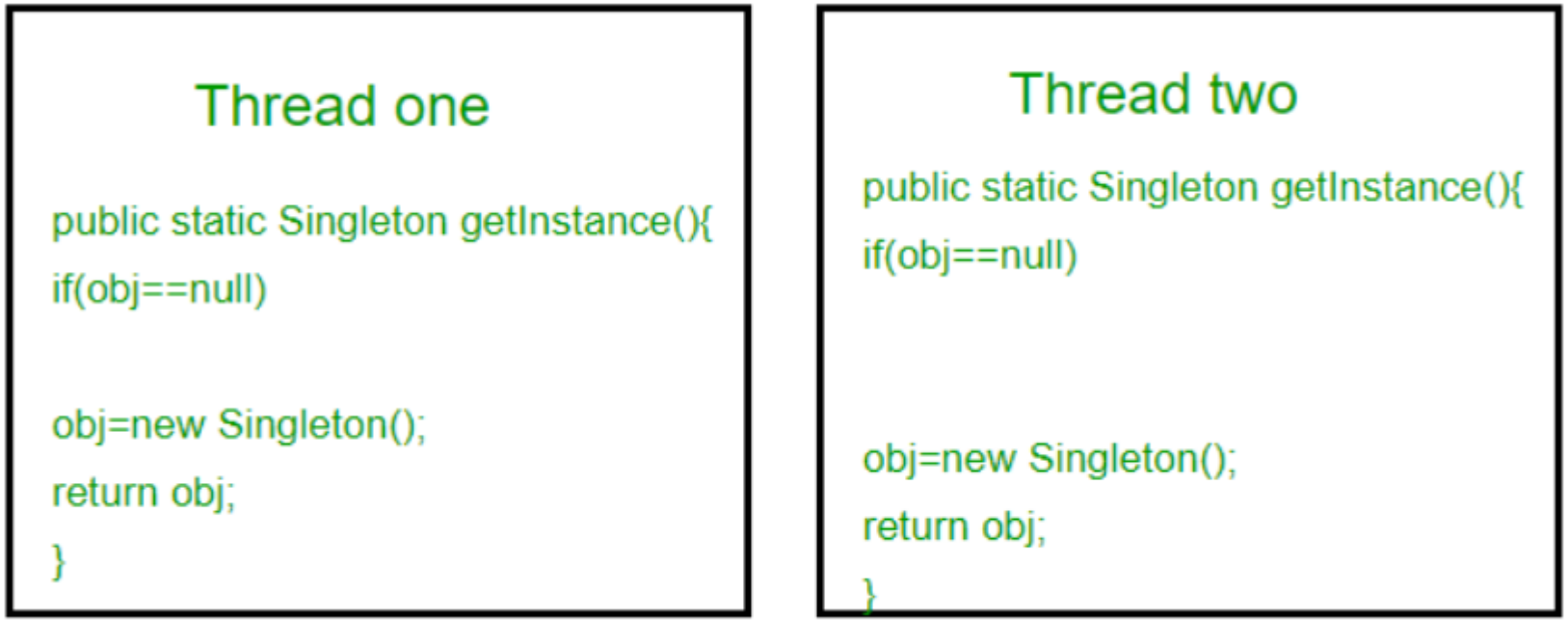
SINGLETON PATTERN

DR. ŞAFAK KAYIKÇI

Singleton Pattern

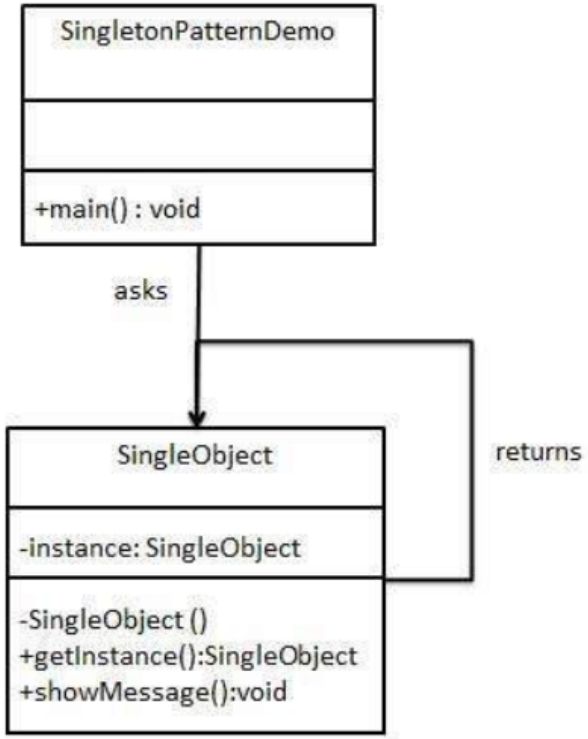
- Yaratıcı Tasarım Desenleri içerisinde yer almaktadır.
- En basit tasarım desenlerinden biridir.
- Nesne oluşturmayı sağlayan tek bir sınıf içerir ve bu sınıf sadece tek bir nesnenin yaratılmasını sağlar.
- Bu sınıf aynı zamanda, sınıf nesnesinin somutlaştırılmasına gerek kalmadan doğrudan erişilebilen tek nesnesine erişmenin bir yolunu sağlar.
- Örnek olarak **veritabanı bağlantılarında** sık kullanılır.

Klasik metod



public static synchronized Singleton getInstance()

Singleton - TutorialsPoint



Tasarım Desenleri

BUILDER PATTERN

DR. ŞAFAK KAYIKÇI

Builder Pattern

- Yaratıcı Tasarım Desenleri içerisinde yer almaktadır.
- Basit nesneleri kullanarak adım adım karmaşık bir nesne oluşturur.
- Bu üretici diğer nesnelerden bağımsızdır.
- Örnek kullanımlar
  - Çoklu sınıf üretimleri
  - Deserialization işlemi
  - Constructor işlemleri

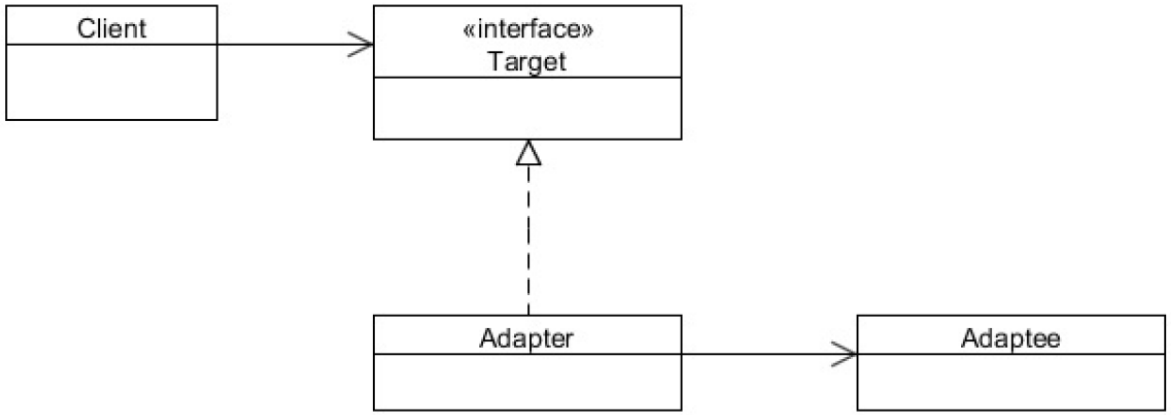
Tasarım Desenleri

ADAPTER PATTERN

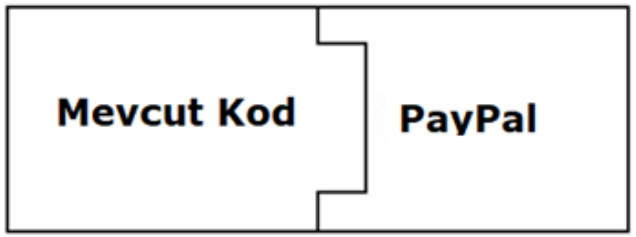
DR. ŞAFAK KAYIKÇI

Adapter Pattern

- Yapısal Tasarım Desenleri içerisinde yer almaktadır.
- Bir sistemi başka bir sistemi etkileşimine yönelik köprü gibi kullanılır.
- Wrapper (sarmalayıcı) olarakta tanımlanır.
- Bu şekilde iki farklı sistemin varolan özellikleri kullanılabilir.

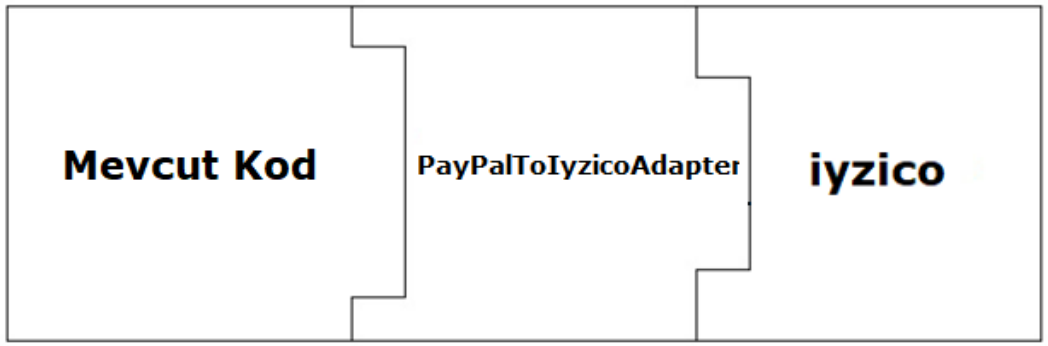


örnek


Mevcut Kod
PayPal


Mevcut Kod


iyzico


Mevcut Kod
PayPalToIyzicoAdapter
iyzico

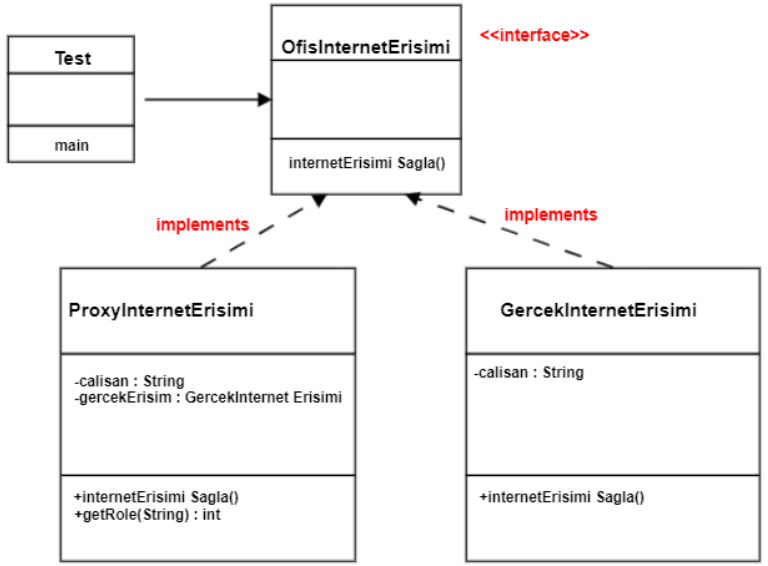
Tasarım Desenleri

PROXY PATTERN

DR. ŞAFAK KAYIKÇI

Proxy Pattern

- Yapısal Tasarım Desenleri içerisinde yer almaktadır.
- Basitçe, proxy, başka bir nesneyi temsil eden bir nesne anlamına gelir.
- Orijinal nesnenin bilgisini gizleme, talep üzerine yükleme vb. gibi birçok işlemi gerçekleştirilebilir.

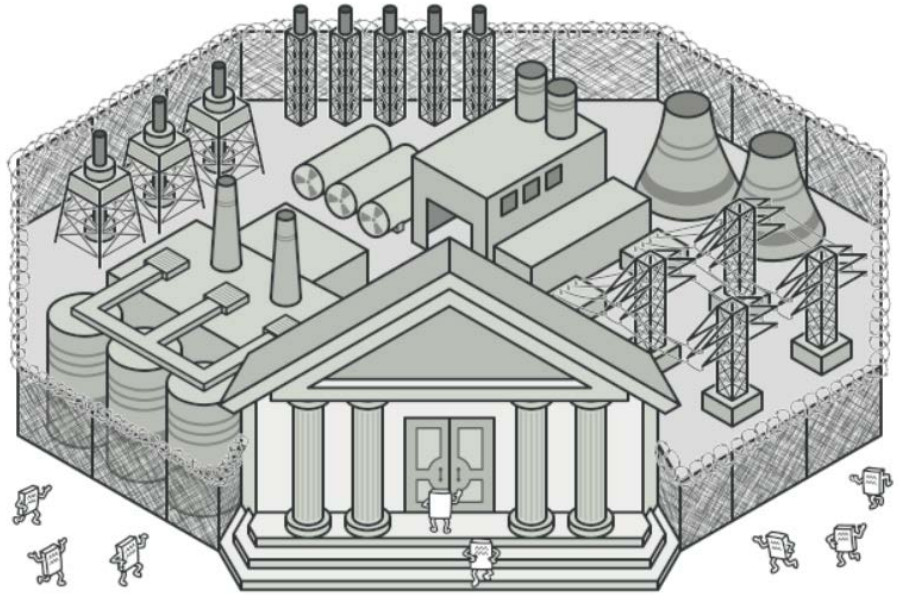


Tasarım Desenleri

FACADE PATTERN

DR. ŞAFAK KAYIKÇI

Facade (Cephe) Pattern



Facade , bir kütüphane, framework veya diğer karmaşık sınıf kümelerine basitleştirilmiş bir arayüz sağlayan yapısal bir tasarım modelidir.

Durum

PROBLEM

Kodunuzun karmaşık bir kitaplığa veya çerçeveye ait geniş bir nesne kümesiyle çalıştırmanız gerektiğini hayal edin. Normalde, tüm bu nesneleri initialize etmeniz, bağımlılıkları takip etmeniz, yöntemleri doğru sırada çalıştırmanız vs. gerekir.

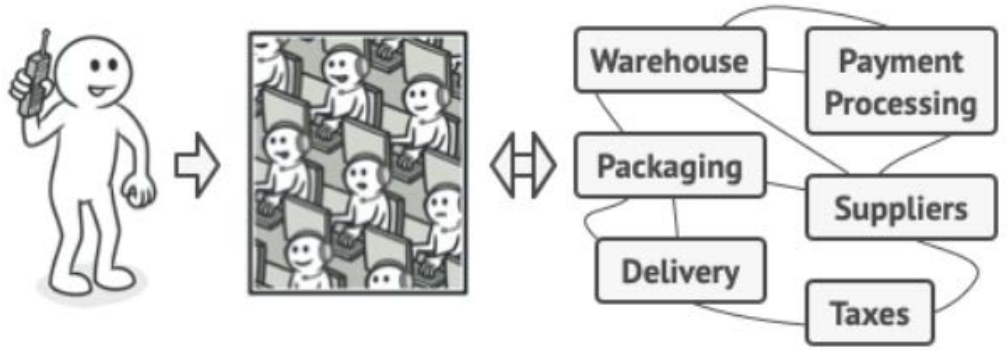
Sonuç olarak, sınıflarınızın iş mantığı, 3rd party sınıfların uygulama ayrıntılarına sıkı bir şekilde bağlanır ve bu da anlaşılması ve sürdürülmesi zorlaşır.

ÇÖZÜM

Facade, çok sayıda hareketli parça içeren karmaşık bir alt sisteme basit bir arayüz sağlayan bir sınıftır. Facade, alt sistemle doğrudan çalışmaya kıyasla sınırlı işlevsellik sağlayabilir. Ancak, yalnızca müşterilerin gerçekten önemsediği özellikleri içerir.

Bir facade sahip olmak, uygulamanızı düzinelerce özelliğe sahip bir kitaplıkla entegre etmeniz gerektiğinde kullanışlıdır, ancak yalnızca işlevselliğinin küçük bir kısmına ihtiyacınız vardır.

Gerçek Dünya Örneği



Telefonla sipariş vermek.

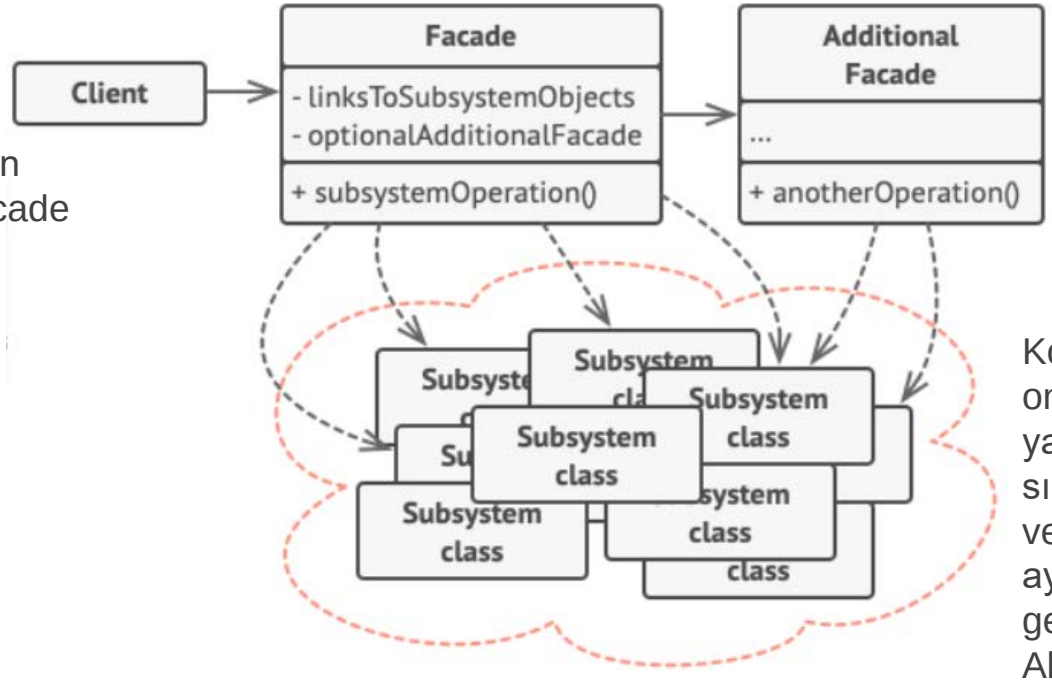
Telefonla sipariş vermek için bir mağazayı aradığınızda, bir operatör mağazanın tüm hizmet ve departmanlarına karşı cephenizdir. Operatör, sipariş sistemine, ödeme ağ geçitlerine ve çeşitli teslimat hizmetlerine basit bir ses arabirimi sağlar.

Yapısı

Facade alt-sistemin işlevselliği belirli bir bölümüne kolay erişim sağlar. İstemcinin talebini nereye yönlendireceğini ve tüm hareketli parçaları nasıl çalıştıracağını bilir.

Bir **Ek Facade** sınıfı, alakasız özellikler kullanılarak yapılabilecek olan Facade sınıfının bozulmasını önlemek için oluşturulabilir. Ek cepheler hem istemciler hem de diğer cepheler tarafından kullanılabilir.

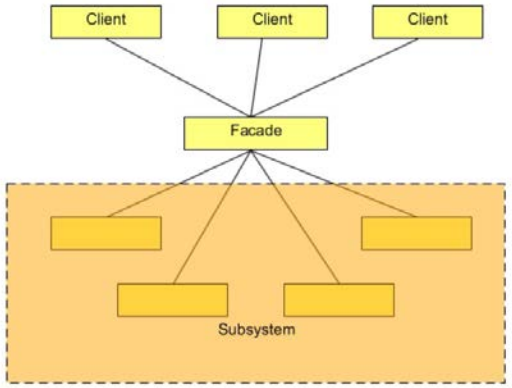
İstemci, alt sistem nesnelerini doğrudan çağırmak yerine facade kullanır.



Kompleks Altsistemi çeşitli nesneler onlarca oluşur. Hepsinin anlamlı bir şey yapmasını sağlamak için, nesneleri doğru sırada başlatmak ve onlara uygun formatta veri sağlamak gibi alt sistemin uygulama ayrıntılarına derinlemesine dalmanız gerekir.
Alt sistem sınıfları cephenin varlığından haberdar değildir. Sistem içinde çalışırlar ve birbirleriyle doğrudan ilişkililerdir.

Facade (Cephe) Pattern

- Yapısal Tasarım Desenleri içerisinde yer almaktadır.
- Facade Deseni, alt sistemin kullanımını kolaylaştıran daha yüksek bir arayüz tanımlar. İstemcileri, alt sistem bileşenlerinin karmaşıklığından korur.
- Aslında her AbsractFactory bir çeşit Facade Desenidir.
- Adapter Deseni gibi birden fazla sınıfı sarmayalabilir ancak Facade Deseni karmaşık olan bir arayüzü sadeleştirmeye yönelik bir arayüz yaparken, Adapter Deseni varolan arayüzü istemcinin gerek duyduğu arayüze çevirmekte kullanılmaktadır.

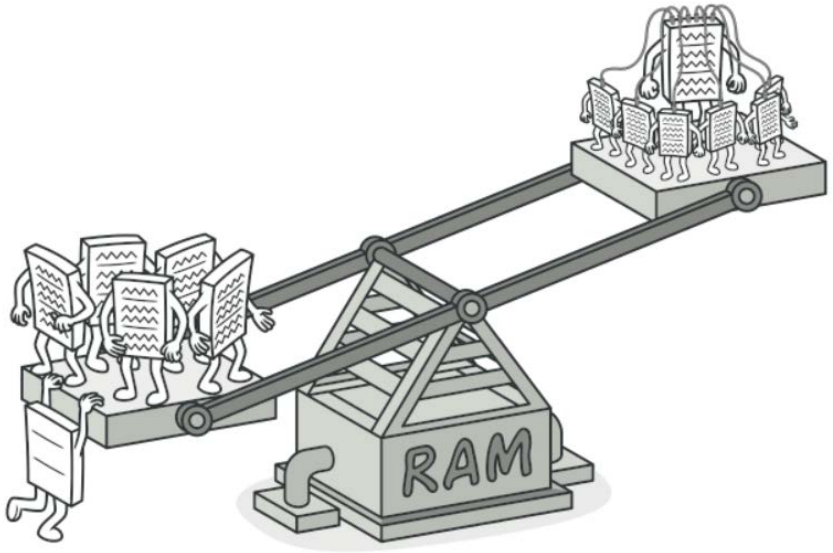


Tasarım Desenleri

FLYWEIGHT PATTERN

DR. ŞAFAK KAYIKÇI

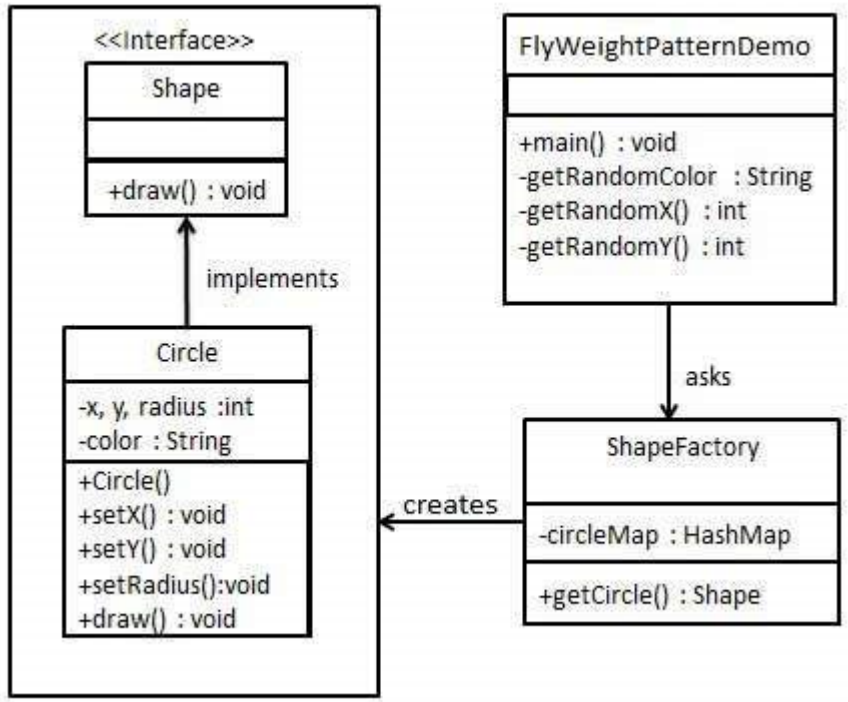
Flyweight Pattern (cache)



Flyweight , her bir nesnedeki tüm verileri tutmak yerine birden çok nesne arasında ortak durum parçalarını paylaşarak mevcut RAM miktarına daha fazla nesne sığdırmanıza olanak tanıyan yapısal bir tasarım modelidir.

Flyweight Pattern

- Yapısal Tasarım Desenleri içerisinde yer almaktadır.
- Oluşturulan nesne sayısını azaltmak ve bellek alanını azaltmak ve performansı artırmak için kullanılır.
- Flyweight modeli zaten var olan benzer türdeki nesneleri saklayarak yeniden kullanmaya çalışır ve eşleşen bir nesne bulunmadığında yeni bir nesne oluşturur.


<<Interface>>
Shape
+draw() : void
implements
Circle
-x, y, radius :int
-color : String
+Circle()
+setX() : void
+setY() : void
+setRadius():void
+draw() : void
FlyWeightPatternDemo
+main() : void
-getRandomColor : String
-getRandomX() : int
-getRandomY() : int
asks
ShapeFactory
-circleMap : HashMap
+getCircle() : Shape
creates

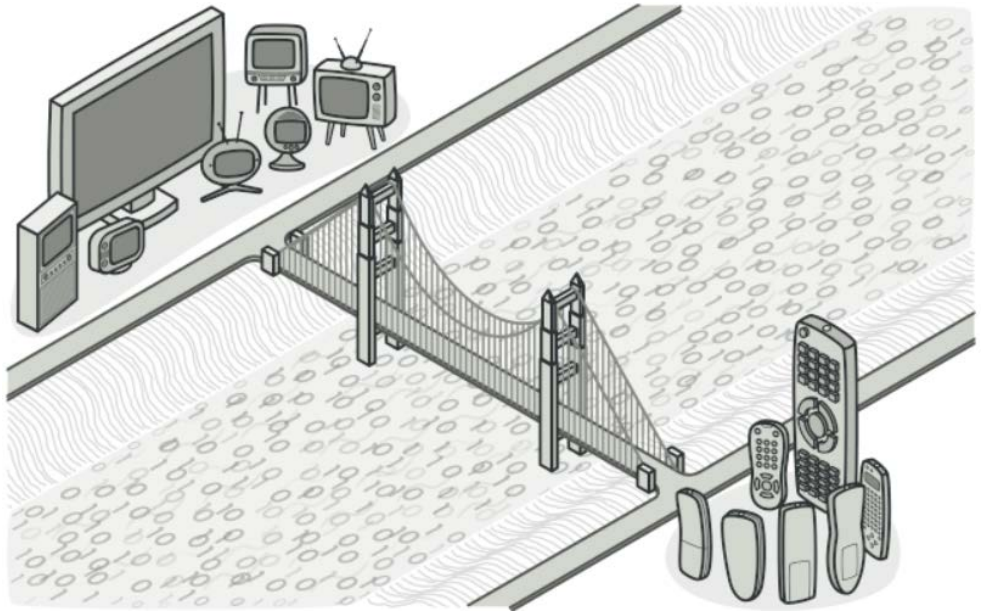
Tasarım Desenleri

BRIDGE PATTERN (KÖPRÜ)

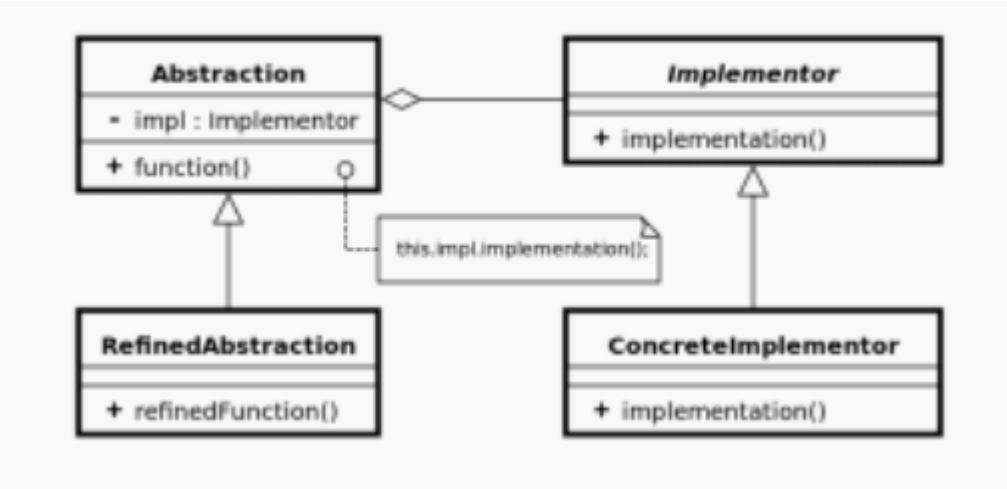
DR. ŞAFAK KAYIKÇI

Bridge (Köprü) Pattern

Bridge (Köprü) modeli, büyük bir sınıfı veya yakından ilişkili bir sınıf kümesini birbirinden bağımsız olarak geliştirilebilen iki ayrı hiyerarşiye (soyutlama (abstraction) ve uygulama (implementation)) ayırmanıza olanak tanıyan yapısal bir tasarım modelidir



UML



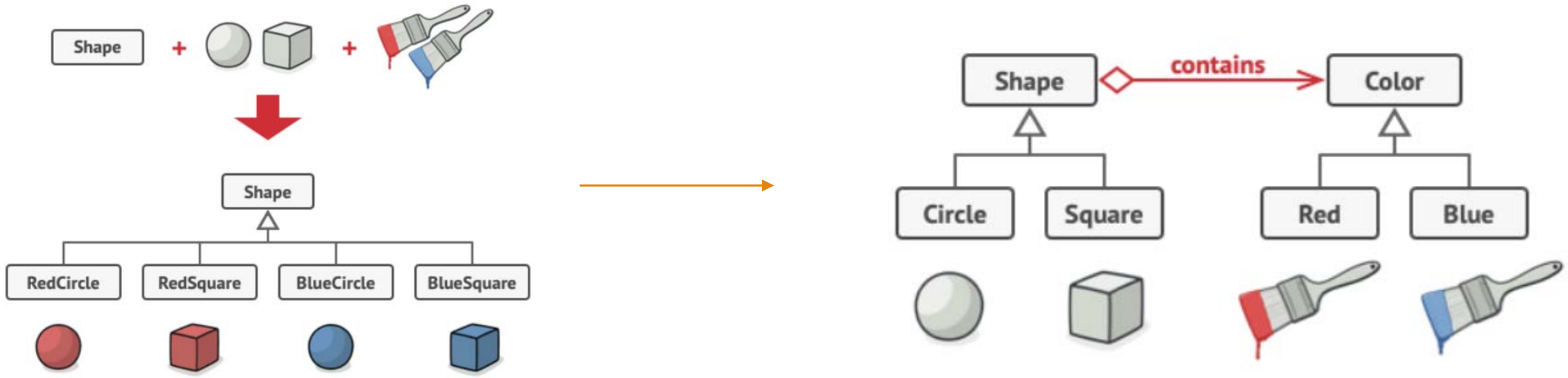
1.Abstraction (abstract class) : Soyut arayüzü, yani davranış bölümünü tanımlar. Ayrıca uygulayıcı (Implementer) referansını da ele alır.

2.RefinedAbstraction (normal class): Abstraction tarafından tanımlanan arayüzü uygular.

3.Implementer (interface) : Uygulama sınıfları için arayüzü tanımlar. Bu arayüzün doğrudan soyutlama arayüzüne karşılık gelmesi gerekmez ve çok farklı olabilir.

4.ConcreteImplementor (normal class) : Implementer arayüzünü uygular.

Sorun - Çözüm

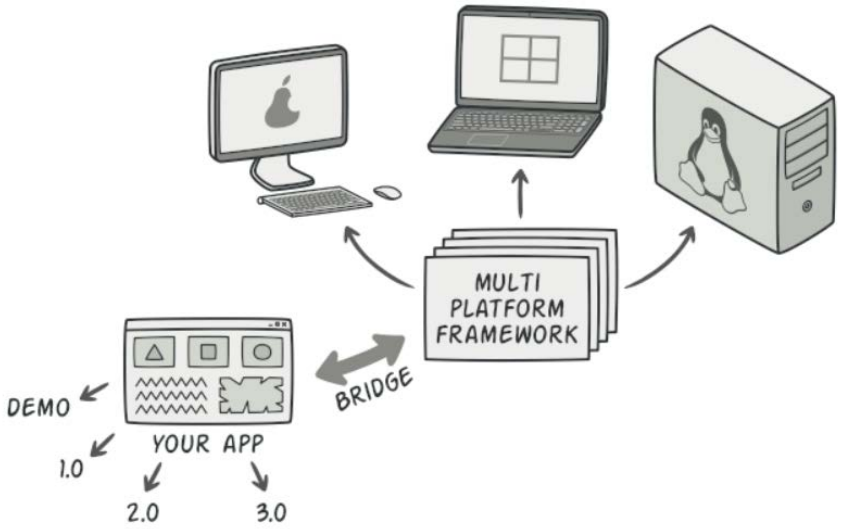


Köprü modeli, kalıtımdan (inheritance) nesne kompozisyonuna (composition) geçerek bu sorunu çözmeye çalışır. Boyutlardan birini ayrı bir sınıf hiyerarşisine çıkararak orijinal sınıflar, tüm durum ve davranışlarına tek bir sınıf içinde sahip olmak yerine yeni hiyerarşinin bir nesnesine başvurur.

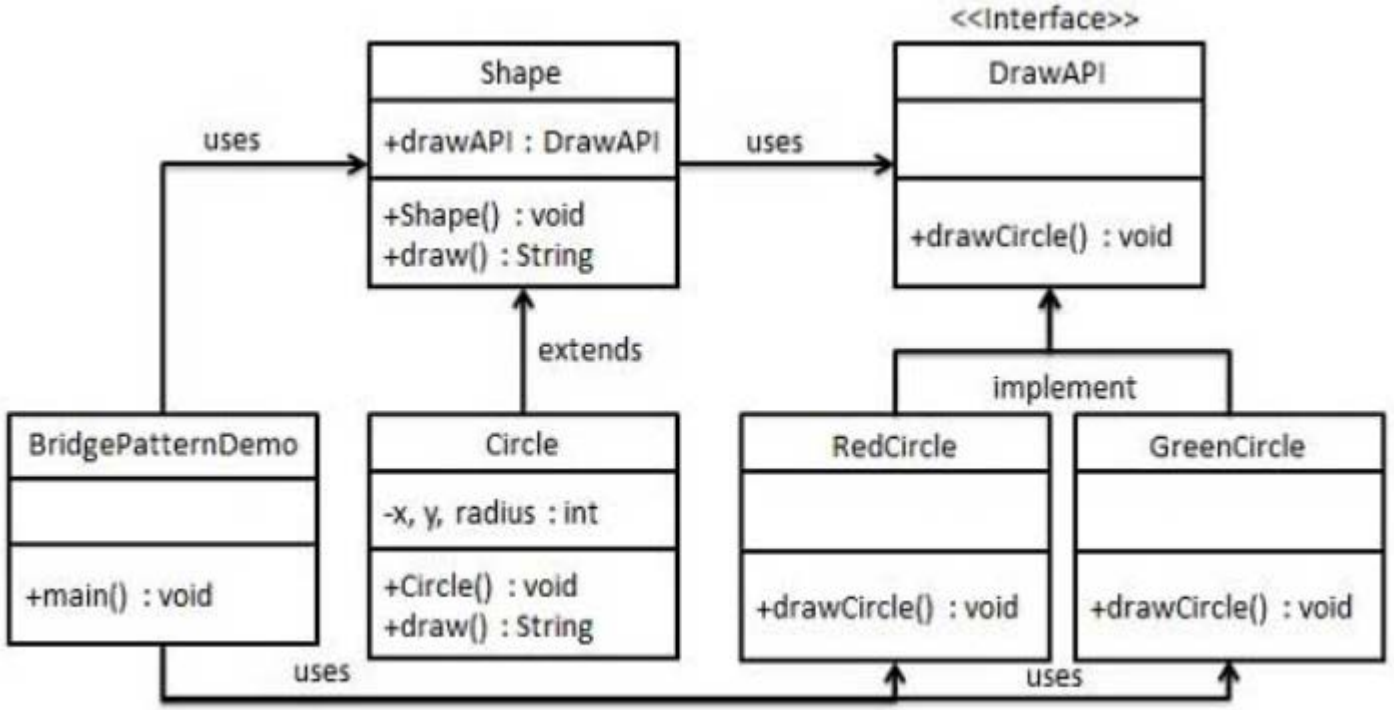
Bridge (Köprü) Pattern

Bir sınıfı birkaç ortogonal (bağımsız) boyutta genişletmeniz gerektiğinde kullanılır.

Platformdan bağımsız sınıflar ve uygulamalar ouşturulabilir. İstemci kodu, üst düzey soyutlamalarla çalışır. Platform ayrıntılarına maruz kalmaz.



Örnek


<<Interface>>
Shape
DrawAPI
uses
+drawAPI : DrawAPI
uses
+Shape() : void
+draw() : String
+drawCircle() : void
extends
implement
BridgePatternDemo
Circle
RedCircle
GreenCircle
-x, y, radius : int
+main() : void
+Circle() : void
+draw() : String
+drawCircle() : void
+drawCircle() : void
uses
uses

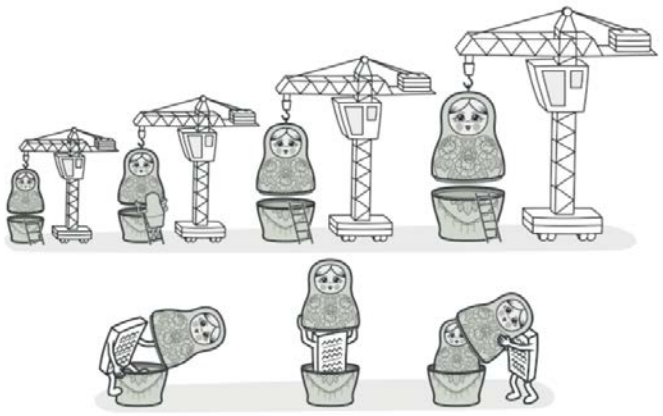
Tasarım Desenleri

DECARATOR PATTERN

DR. ŞAFAK KAYIKÇI

Decarator Pattern

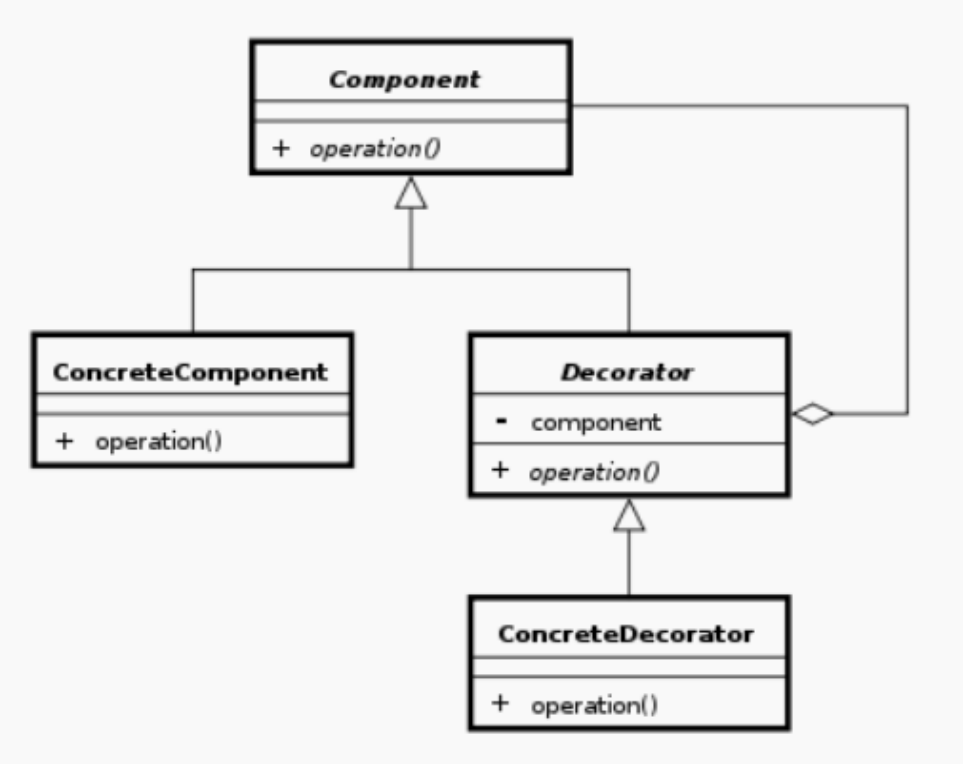
Dekoratör deseni, bir kullanıcının yapısını değiştirmeden mevcut bir nesneye yeni işlevsellik eklemesine izin verir. Bu desen türü, bu desen mevcut sınıf için bir sarmalayıcı görevi gördüğünden yapısal modele girer.



Bir nesneniz var ve içine başka bir nesne koyuyorsunuz. Her ikisi de aynı (veya benzer) arayüzleri destekler. Dışarıdaki nesne "decorator " ve o kullanılır. İçerisindeki nesnenin yöntemlerini ya maskeler, değiştirir ya da aynen kullanılır.

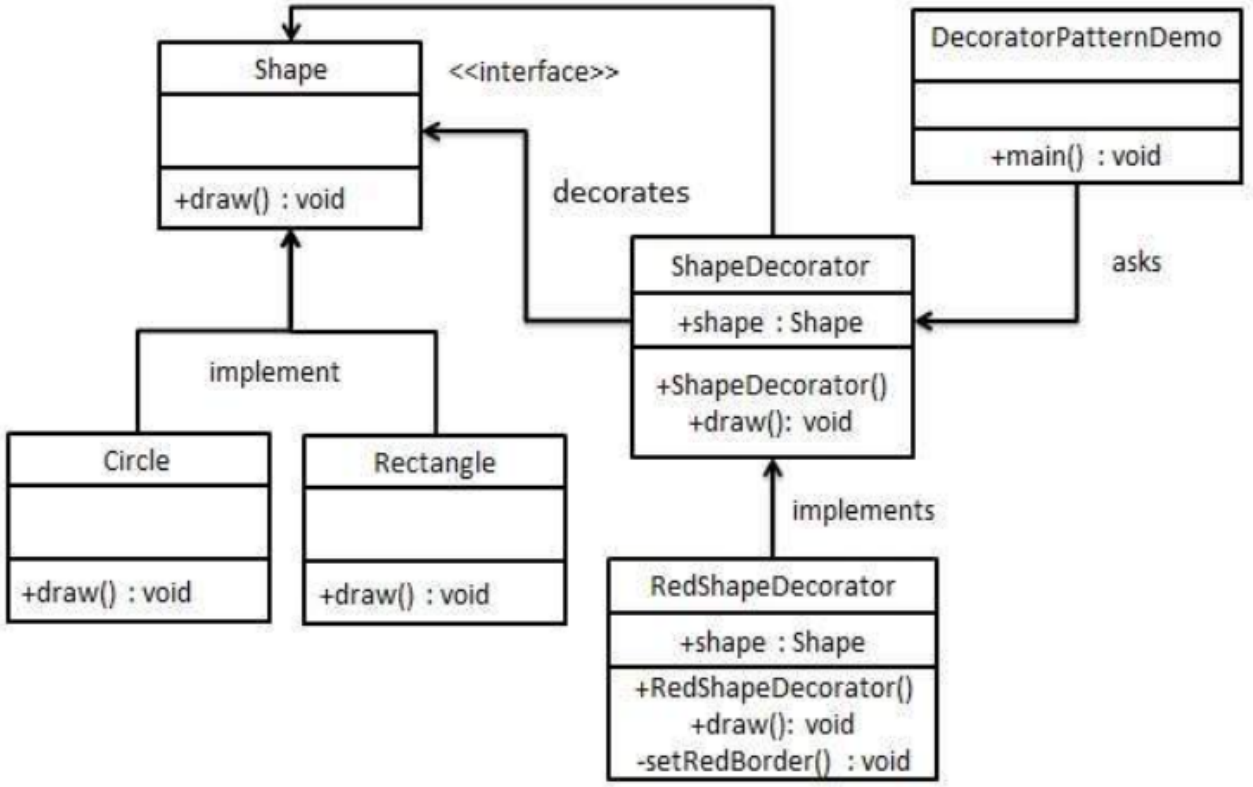
Decarator Pattern

Dekoratörler, işlevselliği genişletmek için alt sınıflamaya esnek bir alternatif sağlar.



- **Component** – çalışma zamanında kendisiyle ilişkili ek sorumluluklara sahip olabilen sarmalayıcıdır.
- **Concrete component**– programda ek sorumlulukların eklendiği orijinal nesnedir.
- **Decorator**- bu, component nesnesine bir referans içeren ve aynı zamanda component arayüzünü uygulayan soyut bir sınıftır.
- **Concrete decorator**- dekoratörü genişletir ve Component sınıfının üzerine ek işlevsellik oluşturur.

Örnek


Shape
<<interface>>
+draw() : void
decorates
DecoratorPatternDemo
+main() : void
asks
ShapeDecorator
+shape : Shape
+ShapeDecorator()
+draw(): void
implement
Circle
+draw() : void
Rectangle
+draw() : void
implements
RedShapeDecorator
+shape : Shape
+RedShapeDecorator()
+draw(): void
-setRedBorder() : void

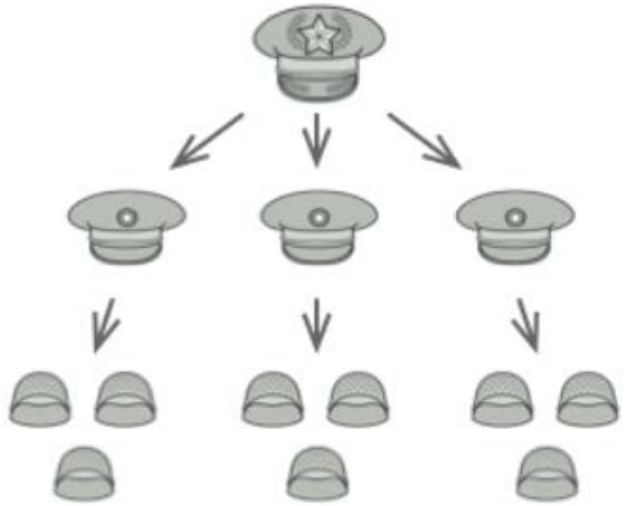
Tasarım Desenleri

COMPOSITE PATTERN (BİLEŞİK)

DR. ŞAFAK KAYIKÇI

Composite Pattern (Bileşik)

Bileşik desen, bir grup nesneyi benzer şekilde tek bir nesne olarak ele almamız gerektiğinde kullanılır. Bileşik desen, tüm hiyerarşinin yanı sıra parçayı da temsil etmek için bir ağaç yapısı bağlamında nesneleri oluşturur. Bu desen türü, bu desen bir nesne grubu ağaç yapısı oluşturduğundan yapısal modelin altına girer.



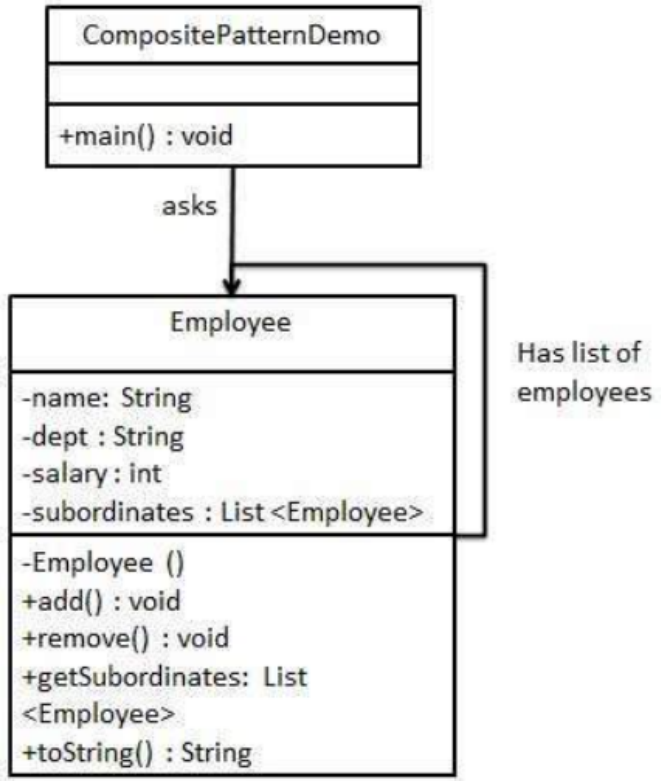
Composite Pattern (Bileşik)

Bu desen, kendi nesnelerinin grubunu içeren bir sınıf oluşturur. Bu sınıf, aynı nesnelerden oluşan grubunu değiştirmenin yollarını sağlar.

Karmaşık ağaç yapılarıyla daha rahat çalıştırılabilir. Bu sayede gerektiği yerlerde polimorfizm ve özyineleme(recursion) kullanılabilir.

Oluşturulan ağaçta, her düğüm / nesne (kök düğüm hariç) ya bileşik ya da yaprak düğümdür. Bileşik desen uygulamak, istemcilerin tek tek nesneleri ve kompozisyonları aynı şekilde işlemesine olanak tanır.

Örnek


CompositePatternDemo
+main() : void
asks
Employee
-name: String
-dept : String
-salary : int
-subordinates : List <Employee>
-Employee ()
+add() : void
+remove() : void
+getSubordinates: List
<Employee>
+toString() : String
Has list of
employees

Tasarım Desenleri

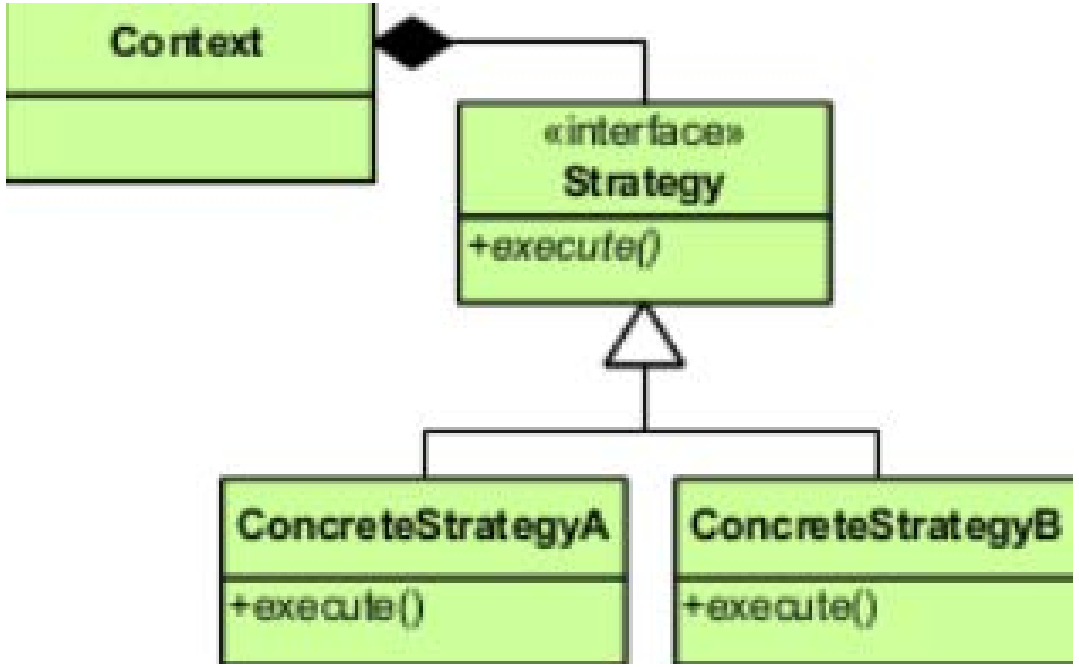
STRATEGY PATTERN

DR. ŞAFAK KAYIKÇI

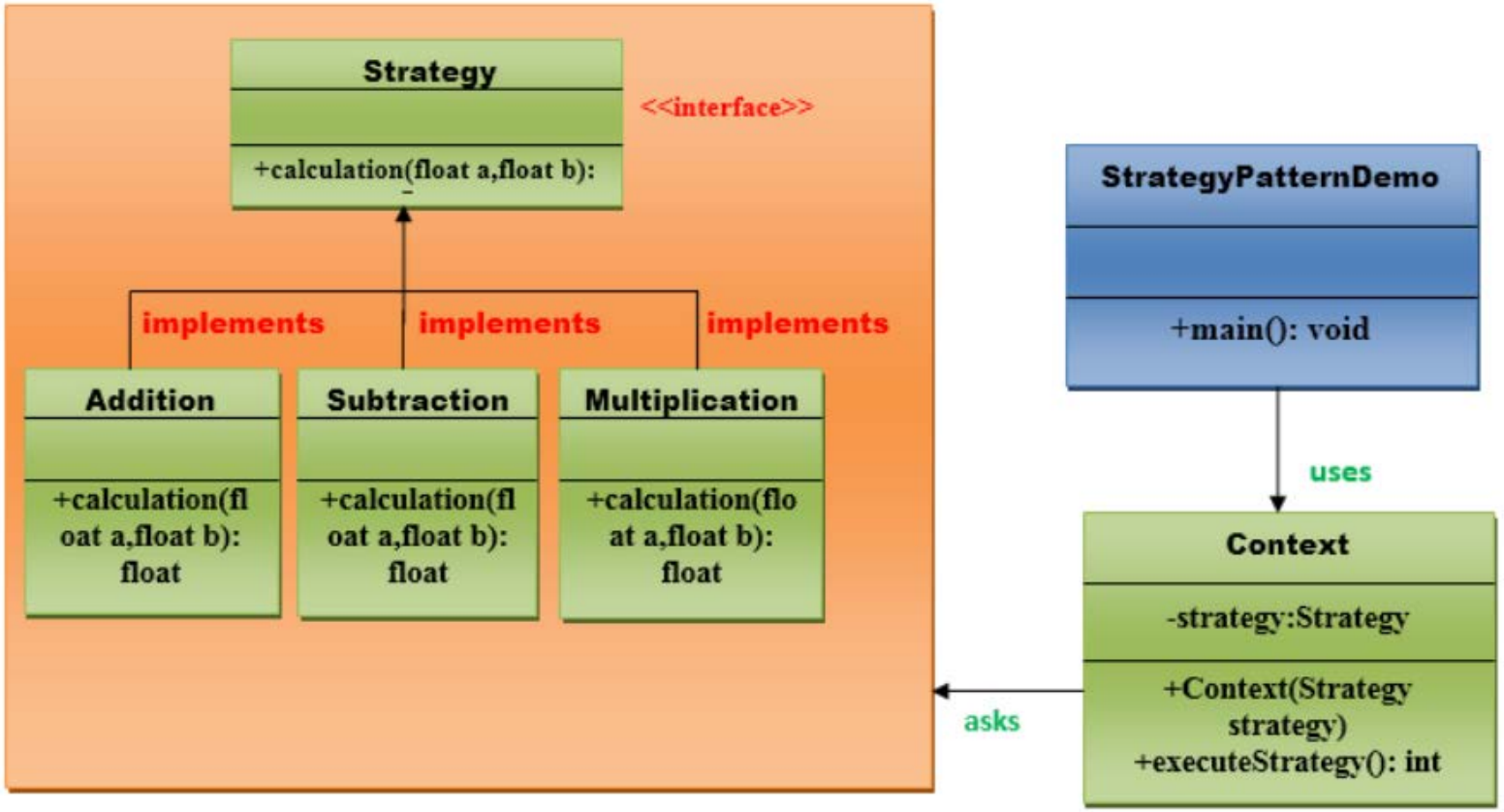
Strategy Pattern

- Strategy tasarım deseni behavioral (davranışsal) grubuna aittir.
- Nesne tarafından kullanılan bir algoritmayı çalışma zamanında dinamik olarak değiştirmeyi sağlar.
- Kod tekrarını ortadan kaldırmanıza izin verir.

Strategy Pattern UML



Örnek – Hesap Makinesi



Adımlar

1. Strategy arayüzü (interface) oluştur
2. Arayüzü kullanacak sınıfları oluştur (Addition, Subtraction, Multiplication)
3. Strateji arayüzünden strateji tipini yürütmesini isteyecek içerik (Context) sınıfı oluştur.
4. Programı çalıştıracak (main) içeren *StartegyPatternDemo* sınıfını oluştur.

Tasarım Desenleri

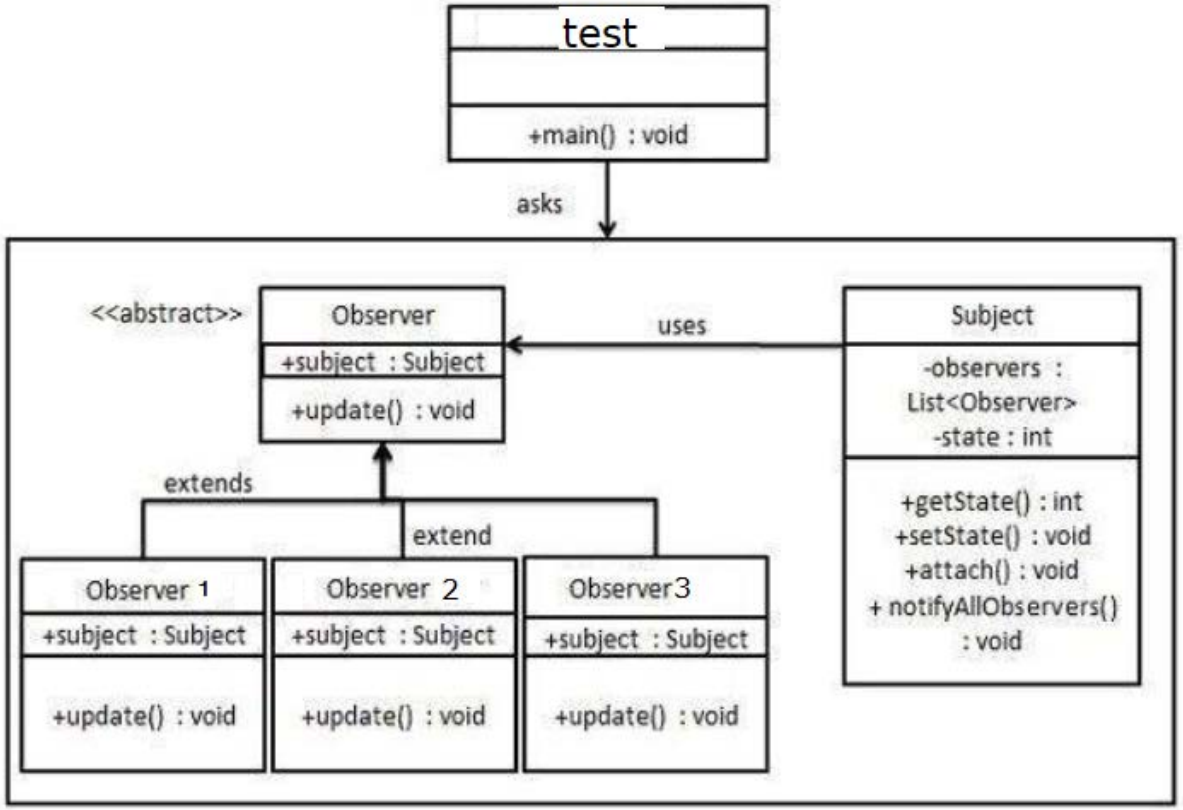
OBSERVER (GÖZLEMCİ) PATTERN

DR. ŞAFAK KAYIKÇI

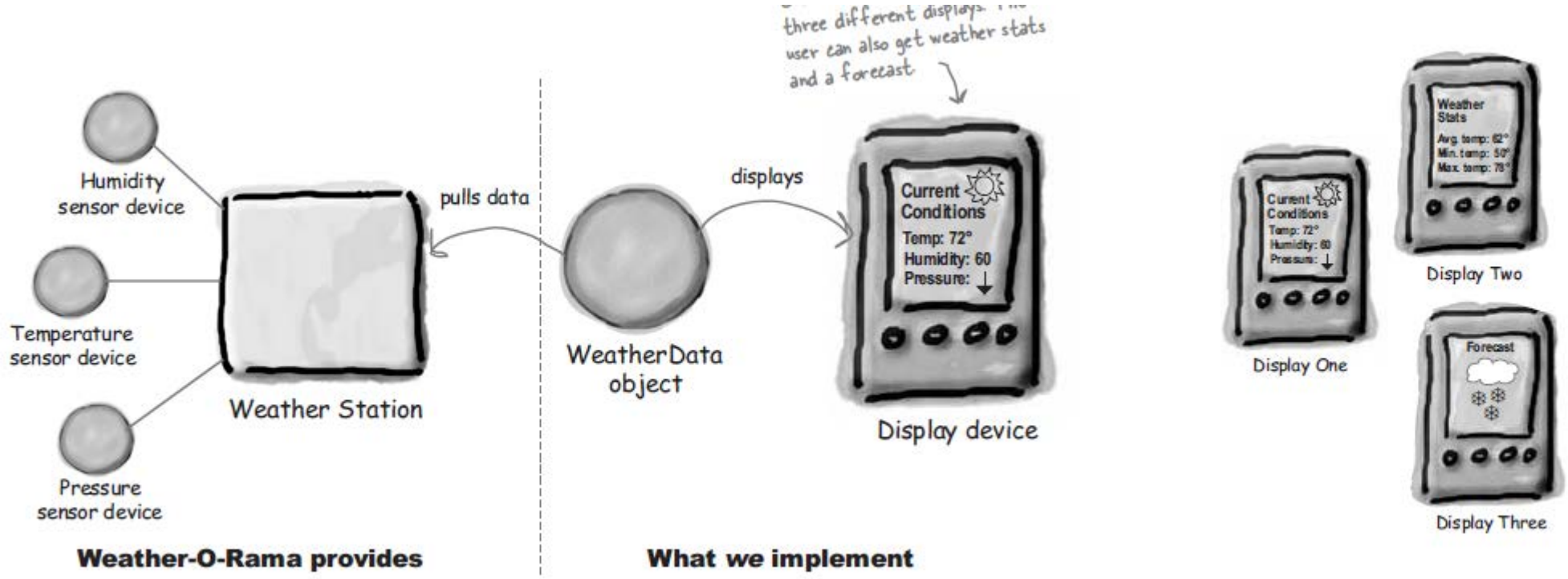
Observer Pattern (Gözlemci Deseni)

- Observer tasarım deseni behavioral (davranışsal) grubuna aittir.
- Geşek bağlantı (loosely coupling) sağlar. Gevşek bağlantı, etkileşime giren nesnelerin birbirleri hakkında daha az bilgiye sahip olması gerektiği anlamına gelir.
- Bir objenin durum değişikliği ile birçok objenin değişikliğine ihtiyaç duyduğumuz zamanlar için kullanılır. Nesneler arasında bire çok ilişki vardır ve gözlemci nesnelerine otomatik olarak bildirim yapılır.
- Göndericinin (subject-publisher), gözlemciler hakkında herhangi birşey bilmesine gerek yoktur.
- Gönderici, alakasız gözlemcilere (observer-subscriber) de güncelleme göndermiş olacaktır.

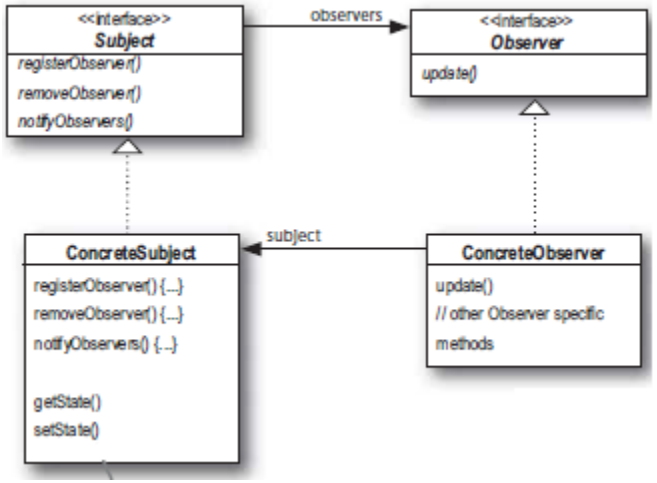
Örnek 1- tutorialspoint

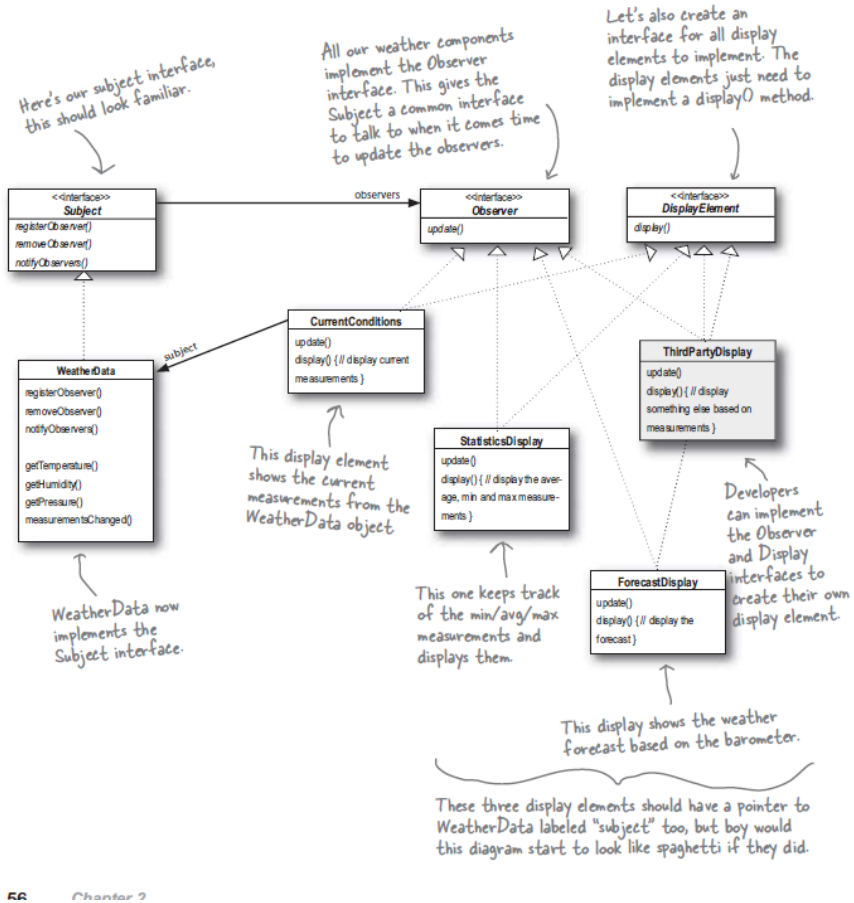


Örnek 2- Head First Design Patterns



Örnek 2- Head First Design Patterns





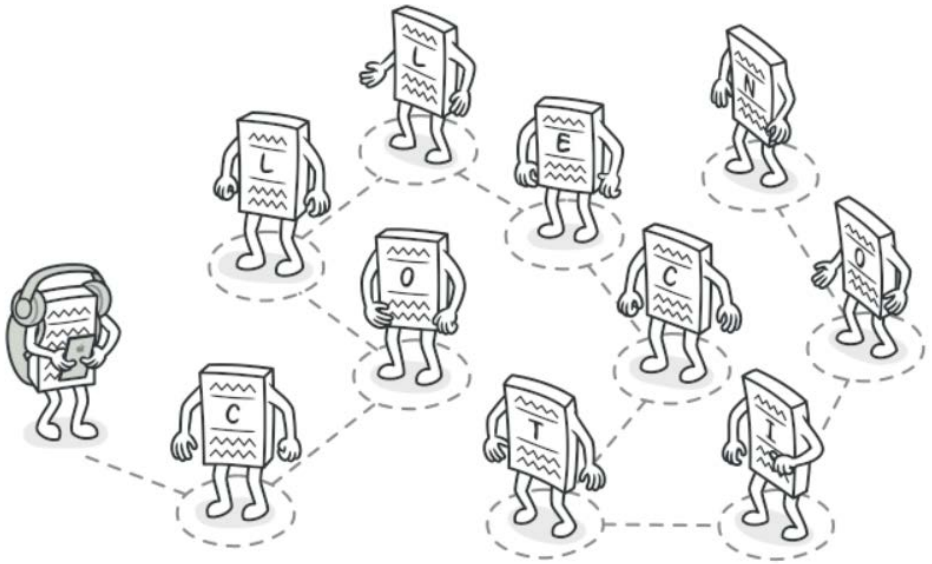
Tasarım Desenleri

ITERATOR PATTERN

DR. ŞAFAK KAYIKÇI

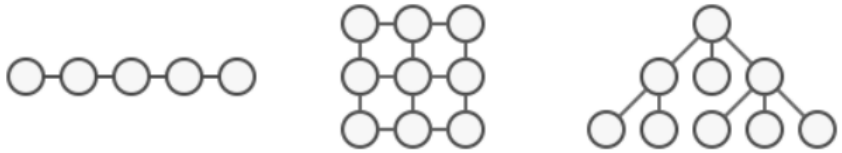
Iterator Pattern

- Davranışsal Tasarım Desenleri içerisinde yer almaktadır.
- Nesne koleksyonlarının (list,array,queue) elemanlarını belirlenen kurallara göre elde edilmesini düzenleyen tasarım desenidir.
- Listenin iç yapısı bilinmeden gezinmeye izin verir.



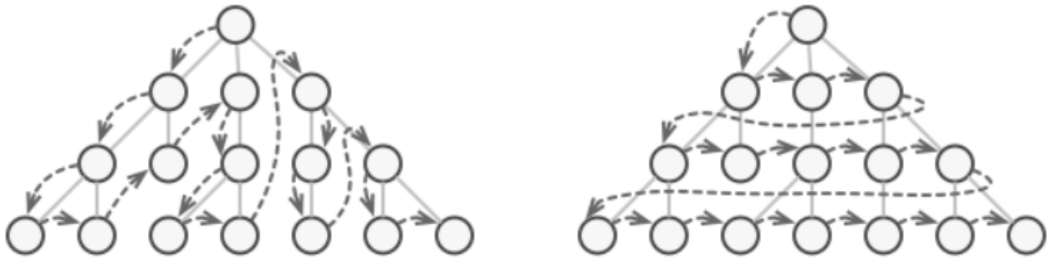
Koleksiyonlar (Collections)

Koleksiyonlar, programlamada en çok kullanılan veri türlerinden olup nesne grupları için torba görevi yapmaktadır.



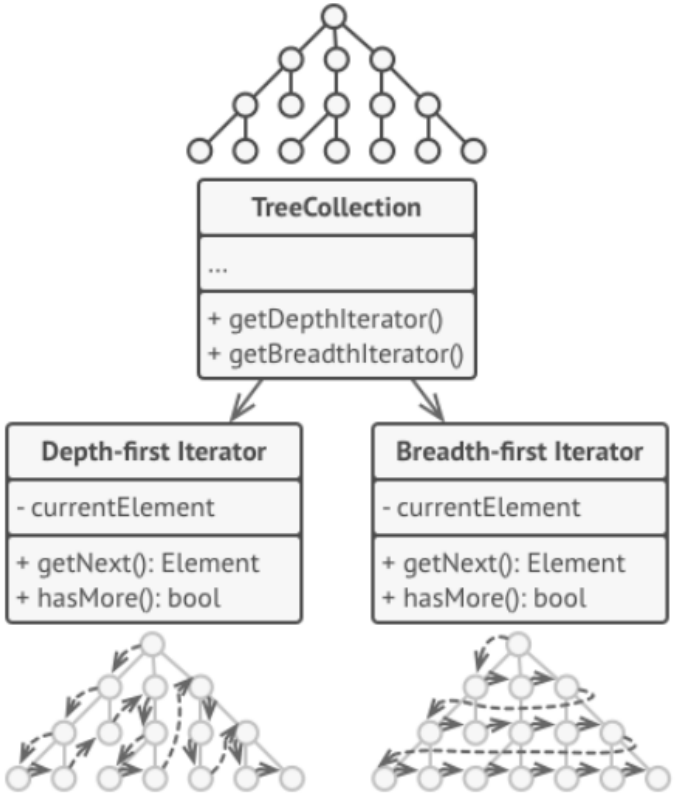
Çeşitli koleksiyon türleri.

Koleksiyonlar basit listeler olabileceği gibi yığınlara (stack), ağaçlara (tree), grafiklere (graph) ve diğer karmaşık veri yapılarına dayanmaktadır. Ancak bir koleksiyon nasıl yapılandırılırsa tasarlansın, aynı öğelere tekrar tekrar erişmeden koleksiyonun her bir öğesi üzerinden geçmenin bir yolu olmalıdır.



Aynı koleksiyon birkaç farklı yoldan geçilebilir.

Iterator Deseni



Iteratörler, çeşitli geçiş(traverse) algoritmalarını uygularlar ve geçiş ayrıntılarını (örn: geçerli konum veya sonuna kadar kaç öğenin kaldığı gibi) kapsüller. Bu sebeple, birkaç iteratatör aynı anda aynı koleksiyonda gezinebilir.

Genellikle iteratörler, koleksiyonun öğelerini getirmek için tek bir ana bir yöntem sağlar. İstemci hiçbir şey döndürmeyinceye kadar bu yöntemi çalıştırmaya devam edebilir; bu, iteratörün tüm öğeleri geçtiği anlamına gelir.

Tüm iteratörler aynı arayüzü (interface) uygulamalıdır. Bu, istemcinin uygun bir iteratör bulduğu sürece kodunu herhangi bir koleksiyon türü veya geçiş algoritmasıyla uyumlu hale getirmesini sağlar.

Bir koleksiyonda gezinmek için özel bir yol ihtiyacı doğarsa, koleksiyonu veya istemciyi değiştirmek zorunda kalmadan sadece yeni bir iteratör sınıfı oluşturulur.

Avantaj - Dezavantaj

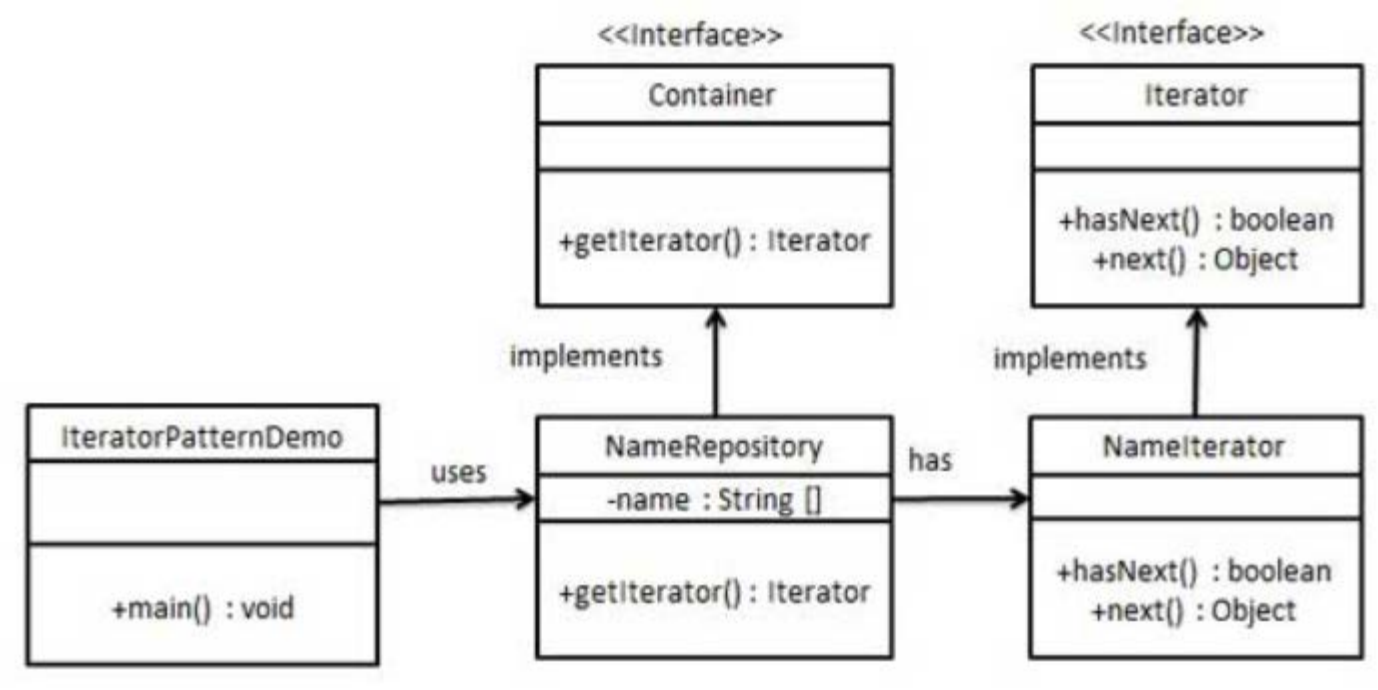
Avantajlar

- *Single Responsibility Principle (Tek Sorumluluk İlkesi) :* Çok yer kaplayan geçiş (traverse) algoritmalarını ayrı sınıflara çıkararak istemci kodunu ve koleksiyonları temizleyebilirsiniz.
- *Open/Closed Principle (Açık / Kapalı Prensibi) :* Yeni koleksiyon ve iteratör türleri uygulayabilir ve bunları hiçbir şeyi bozmadan mevcut koda geçirebilirsiniz.
- Her iteratör kendi iterasyon durumunu içerdiğinden, aynı koleksiyon üzerinde paralel olarak gezinme yapabilirsiniz. Aynı nedenle, bir yinelemeyi erteleyebilir ve gerektiğinde devam edebilirsiniz.

Dezavantajlar

- Uygulamanız sadece basit koleksiyonlarla çalışıyorsa, kalıbı uygulamak abartılı olabilir.
- Bir iteratör kullanmak, bazı özel koleksiyonların öğelerinden doğrudan geçmekten daha az verimli olabilir.

Iterator Deseni



Tasarım Desenleri

MEMENTO PATTERN

DR. ŞAFAK KAYIKÇI

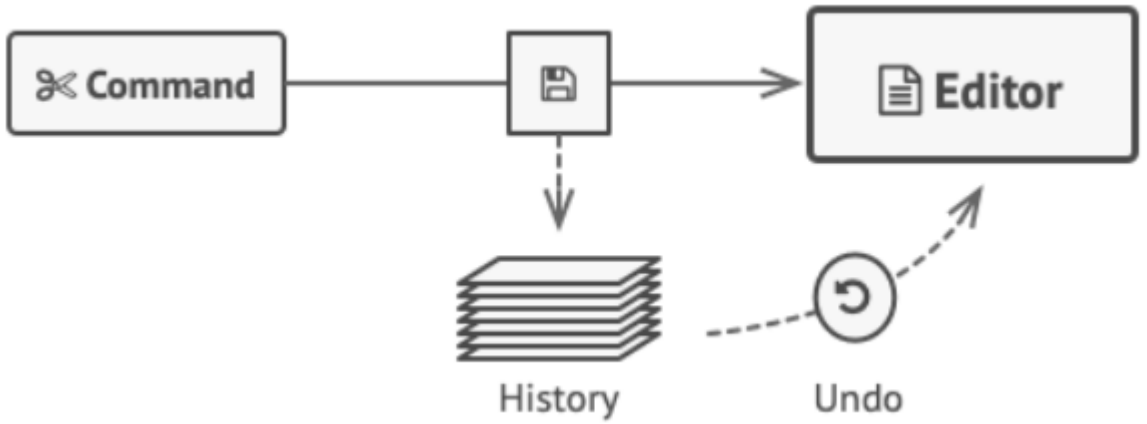
Memento (Snapshot) Pattern

- Davranışsal Tasarım Desenleri içerisinde yer almaktadır.
- Bir nesnenin durumunu kaydetmenize ve önceki bir duruma geri yüklemek için kullanılır.

Akıl Defteri

Memento Deseni

Undo veya ctrl + z bir düzenleyicide en çok kullanılan işlemlerden biridir.



Avantaj - Dezavantaj

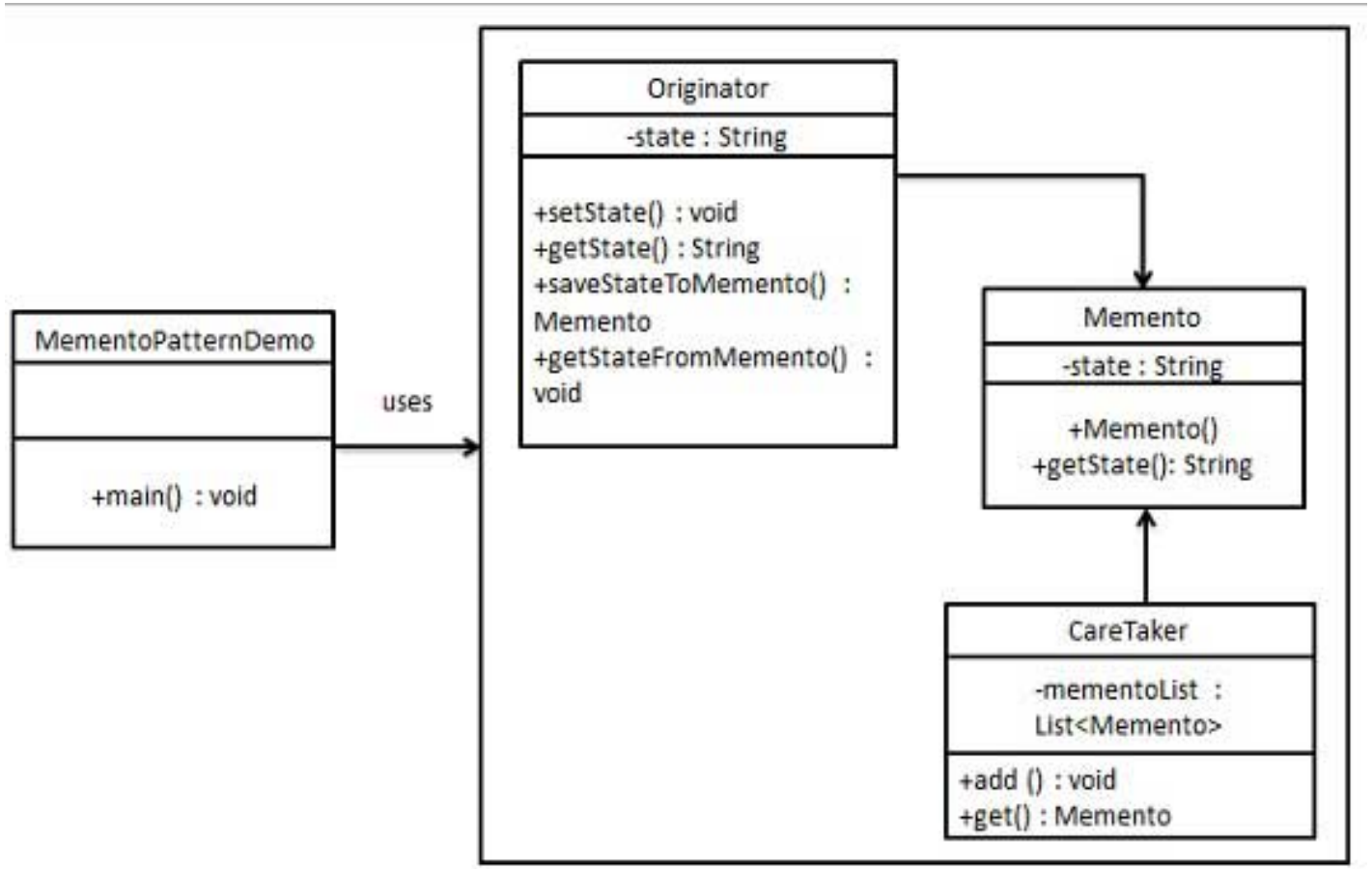
Avantajlar

- Encapsulation ihlal etmeden nesnenin durumunun anlık görüntülerini üretebilirsiniz.
- Bakıcının (caretaker), oluşturanın (Originator) durum geçmişini korumasına izin vererek, oluşturanın (Originator) kodunu basitleştirebilirsiniz.

Dezavantajlar

- İstemciler çok sık snaphot oluşturuyorsa, uygulama çok fazla RAM tüketebilir.
- Bakıcılar (caretaker), eskimiş snapshotları yok edebilmek için oluşturanın (Originator) yaşam döngüsünü izlemelidir.
- PHP, Python ve JavaScript gibi çoğu dinamik programlama dili, mementodaki duruma dokunulmadan kalacağını garanti edemez.

Memento Deseni



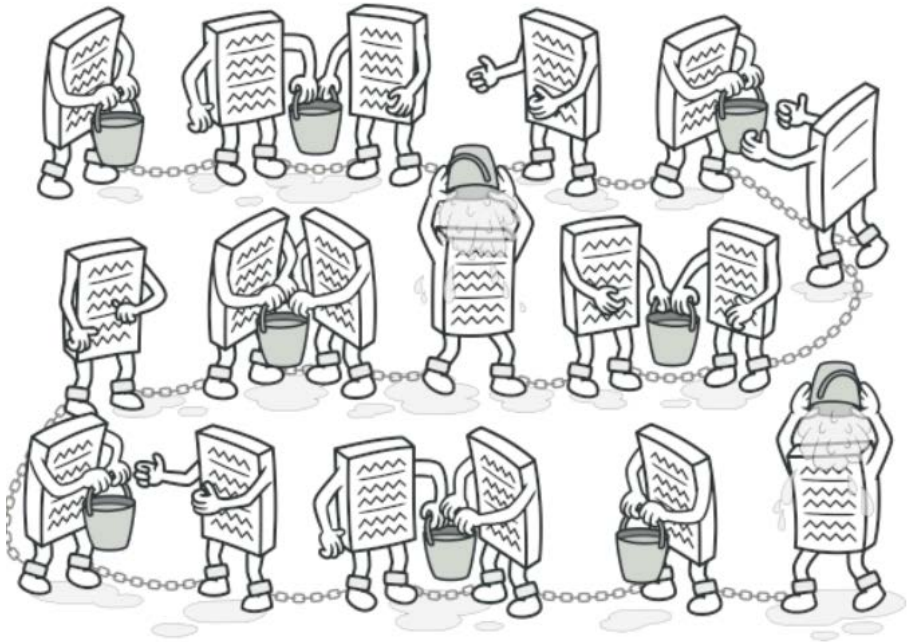
Tasarım Desenleri

CHAIN OF RESPONSIBILITY

DR. ŞAFAK KAYIKÇI

CoR – Chain of Responsibility

CoR (sorumluluk zinciri) istekleri bir işleyici (handler) zinciri boyunca iletmenize izin veren davranışsal bir tasarım modelidir. Bir istek geldikten sonra, her işleyici isteği işlemeye veya zincirdeki bir sonraki işleyiciye iletmeye karar verir.



CoR – Chain of Responsibility

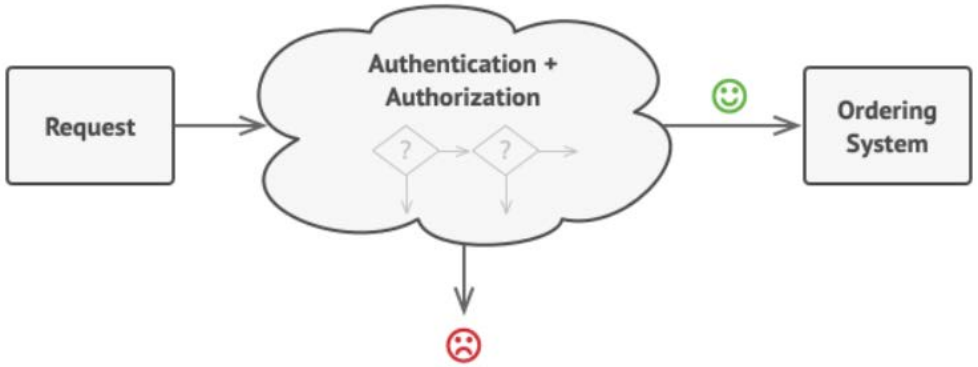
Programın farklı türden istekleri çeşitli şekillerde işlemesi beklendiğinde ancak isteklerin kesin türleri ve sıraları önceden bilinmediğinde CoR kullanılır.

CoR, birkaç işleyiciyi tek bir zincire bağlamanıza ve bir istek alındığında, her işleyiciye onu işleyip işleyemeyeceğini "sormanıza" olanak tanır. Bu şekilde, tüm işleyiciler isteği işleme şansı elde eder.

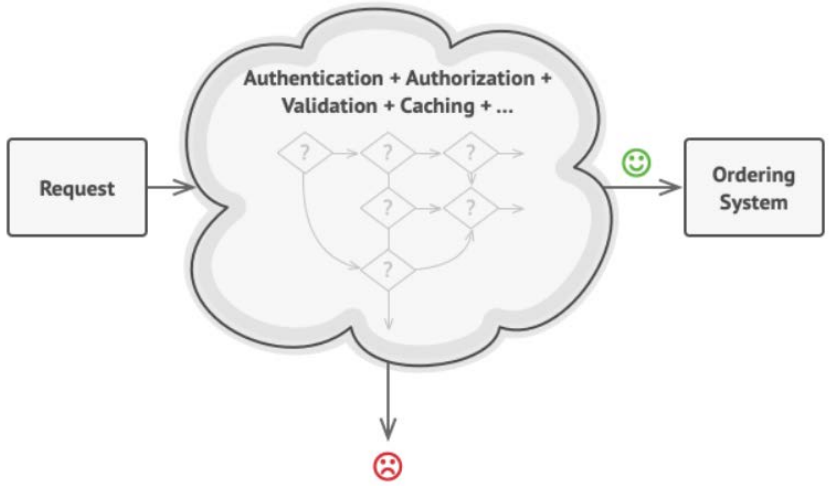
Zincirdeki işleyicileri herhangi bir sırayla bağlayabileceğiniz için tüm istekler tam olarak planladığınız gibi zincirden geçecektir.

İşleyici sınıflarının içindeki bir referans alanı için setter metodları sağlarsanız, işleyicileri dinamik olarak ekleyebilir, kaldırabilir veya yeniden sıralayabilirsiniz.

Örnek



1- Sipariş sisteminde talebin bir dizi kontrolden geçmesi gerekir.



2- Kod büyüdükçe daha karmaşık hale gelir

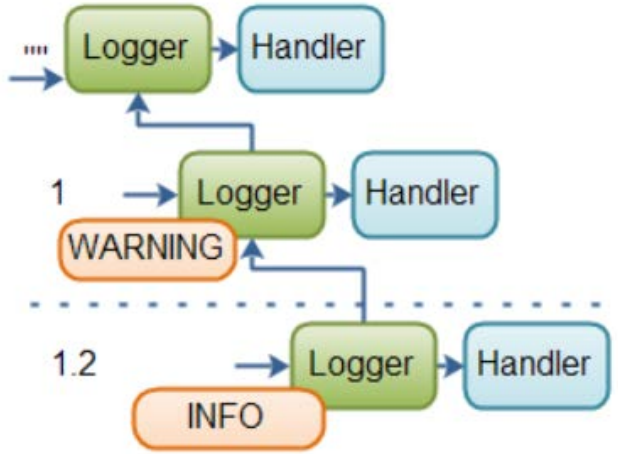
3- İşleyiciler bir zincir oluşturacak şekilde sıraya dizilir.

Örnek 2 – JDK

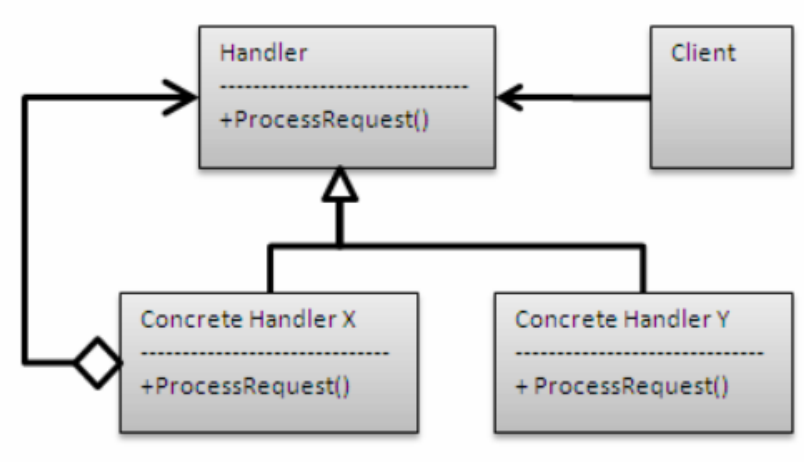
try-catch

```
try
{
      istisna yaratabilecek program kodu
}
catch ( exception-1 ex-1 )
{
      istisna-1 durumunu ele alacak kod
}
catch ( exception-2 ex-2 )
{
      istisna-2 durumunu ele alacak kod

}
finally
{
      her durumda çalışacak kod
}
```

logger



UML



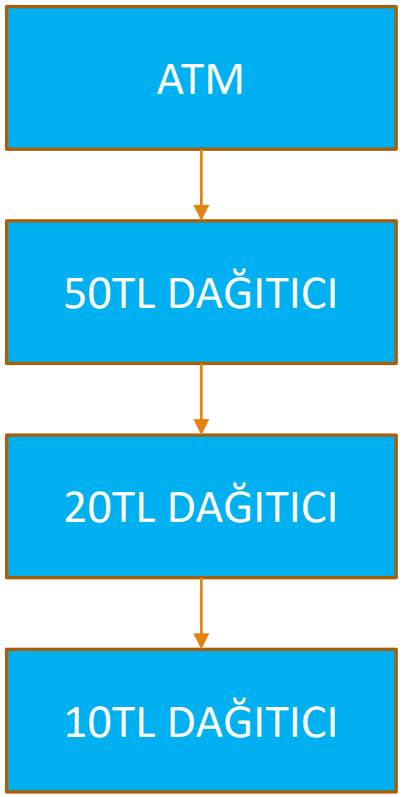
Handler: Kendisinden türeyen ConcreteHandler' ların, talebi ele alması için gerekli arayüzü tanımlar. Abstract class veya Interface olarak tasarlanır.

ConcreteHandler: Sorumlu olduğu talebi değerlendirir ve işler. Gerekirse talebi zincir içerisinde arkasından gelen nesneye iletir. Sonraki nesnenin ne olacağı genellikle istemci tarafında belirlenir.

Client : Talebi veya mesajı gönderir.

Örnek

10TL katlarında miktar giriniz :



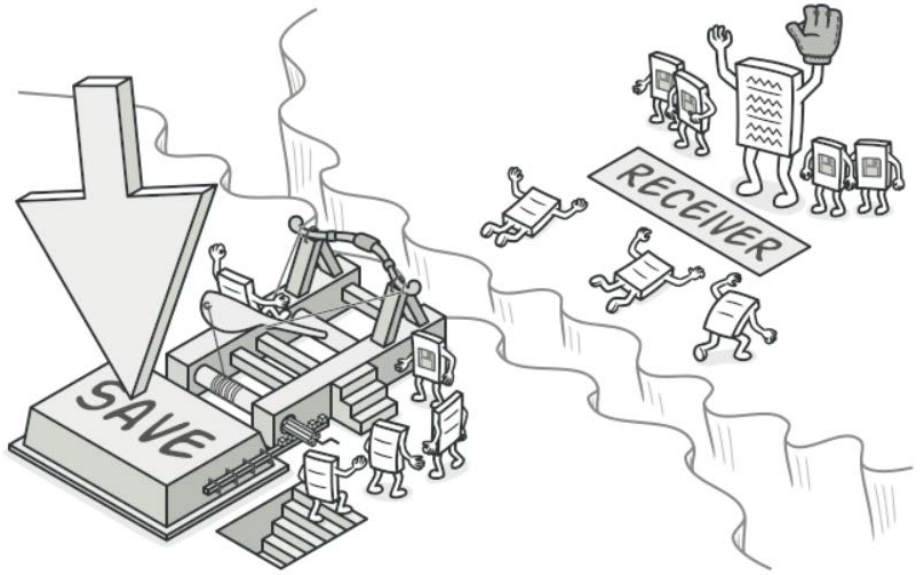
Tasarım Desenleri

COMMAND PATTERN

DR. ŞAFAK KAYIKÇI

Command Pattern (Komut Deseni)

Command deseni bir isteği, istekle ilgili tüm bilgileri içeren bağımsız bir nesneye dönüştüren davranışsal bir tasarım modelidir. Bu dönüşüm, farklı isteklere sahip yöntemleri parametrelendirmenize, bir isteğin yürütülmesini geciktirmenize veya sıraya koymanıza ve geri alınamaz işlemleri desteklemenize olanak tanır.



Command Pattern (Komut Deseni)

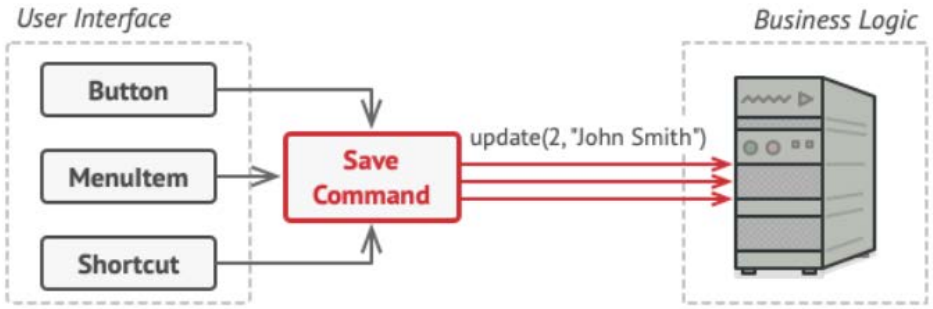
Command deseni, belirli bir yöntem çağrısını bağımsız bir nesneye dönüştürebilir. Bu değişiklik sayesinde komutlar yöntem argümanları olarak iletebilir, bunları diğer nesnelerin içinde saklanabilir veya çalışma zamanında bağlantılı komutları değiştirilebilir.

Diğer nesnelerde olduğu gibi, komutta serileştirilebilir. Bu da onu bir dosyaya veya veritabanına kolayca yazılabilen bir stringe dönüştürmek anlamına gelir. Daha sonra, string ilk komut nesnesi olarak geri yüklenebilir. Böylece, komutun yürütülmesini geciktirebilir, yarlanabilir ve ağ üzerinden iletilebilir.

Örnek

Birkaç sınıf aynı işlevselliği uygular.

GUI nesneleri, iş mantığı nesnelerine doğrudan erişebilir.

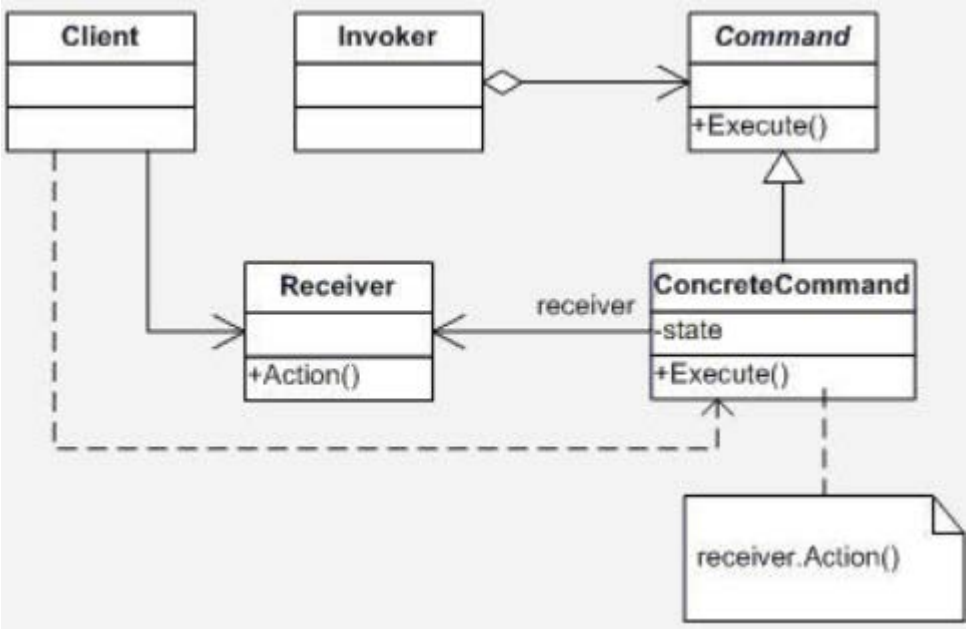


İş mantığı katmanına bir komut aracılığıyla erişim.

Örnek 2- JDK

- Runnable interface (java.lang.Runnable)
- Swing Action (javax.swing.Action)
- Invocation of Struts Action class by Action Servlet (dahili olarak kullanmaktadır)

UML



Komut (Command) : Gerçekleştirilecek işlem için bir ara yüz tanımlar.

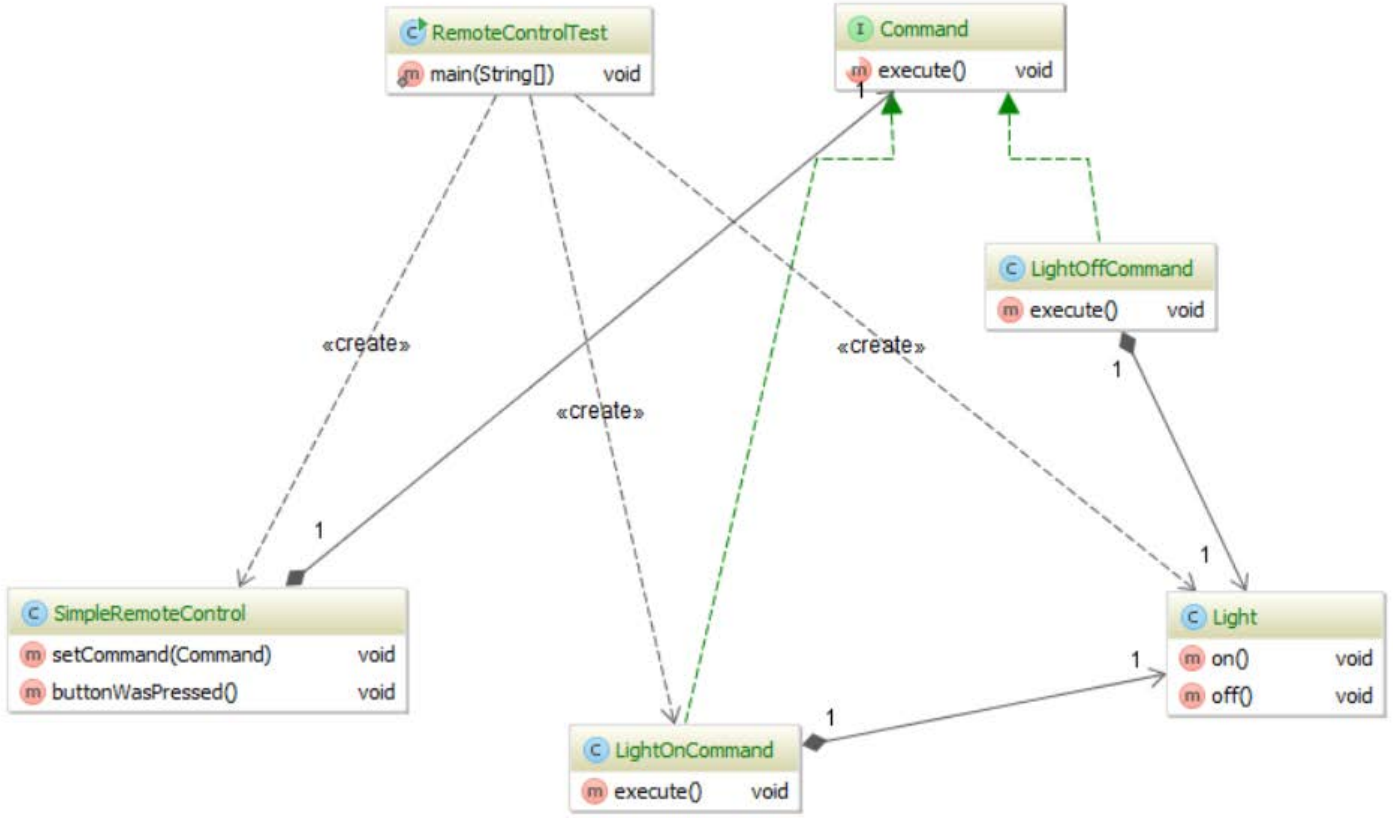
Somut Komut (Concrete Command): Alıcı ve gerçekleştirilecek işlemler arasında bir bağ kurar, alıcıda karşılık düşen işlemleri çağırarak çalıştırma eylemini gerçekleştirir.

İstemci (Client): Komut nesnesini oluşturur ve metodun sonraki zamanlarda çağrılabilmesi için gerekli bilgiyi sağlar.

Çağırıcı (Invoker): Metodun ne zaman çağrılacağını belirtir.

Alıcı (Receiver): Kullanıcı isteklerini gerçekleştirecek asıl metod kodlarını içerir.

Akıllı Ev


RemoteControlTest
main(String[]) void
Command
execute() void
LightOffCommand
execute() void
SimpleRemoteControl
setCommand(Command) void
buttonWasPressed() void
Light
on() void
off() void
LightOnCommand
execute() void
«create»
«create»
«create»
1
1
1
1
1
1

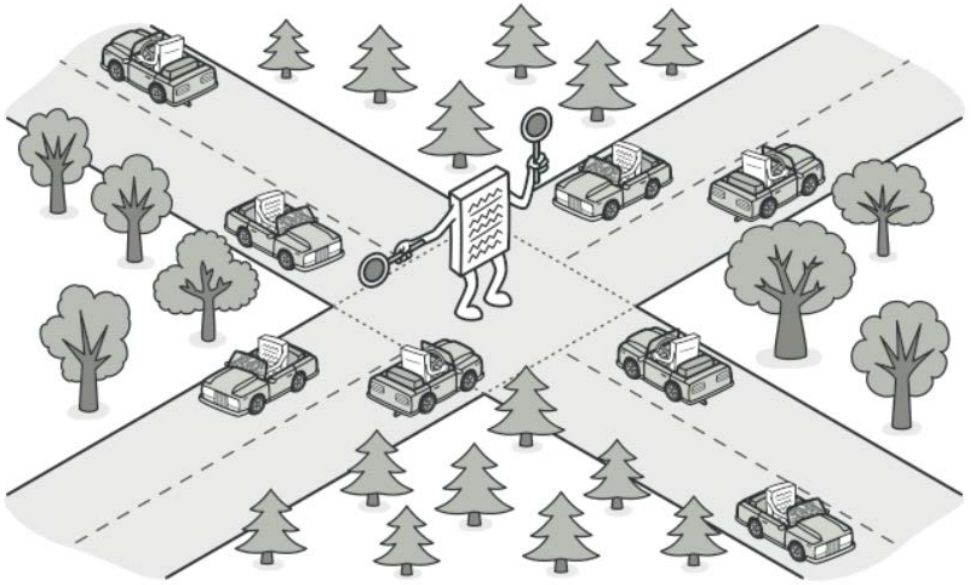
Tasarım Desenleri

MEDIATOR (ARACI)

DR. ŞAFAK KAYIKÇI

Mediator Pattern

Arabulucu (Meditor) tasarım kalıbı, nesneler arasındaki bağımlılıkları azaltmanıza izin veren davranışsal bir tasarım modelidir. Model, nesneler arasındaki doğrudan iletişimi kısıtlar ve onları yalnızca bir aracı nesnesi aracılığıyla işbirliği yapmaya zorlar.

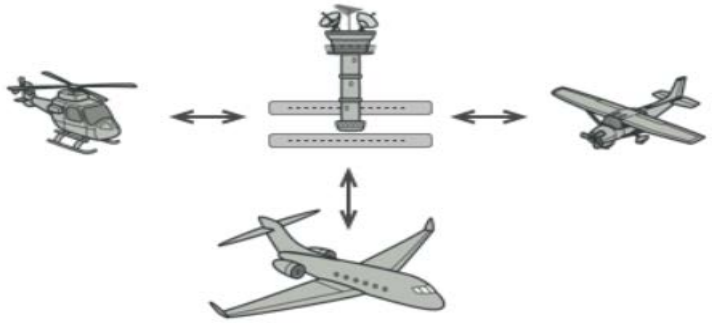


Mediator Pattern

Mediator, nesneler arasında gevşek bağlı (**loosely coupled)** iletişim kurulmasına yardımcı olur ve birbirine doğrudan referansları azaltmaya yardımcı olur. Bu, bağımlılık yönetiminin karmaşıklığını ve katılan nesneler arasındaki iletişimin en aza indirilmesine yardımcı olur. Nesneler, düzinelerce başka nesneye bağlanmak yerine yalnızca tek bir aracı sınıfına bağlıdır.

Birden çok nesnenin isteği işlemek için birbiriyle etkileşime girmesi gereken ve doğrudan iletişim karmaşık bir sistem oluşturabileceği durumlar için Mediator modeli kullanılabilir.

Örnekler

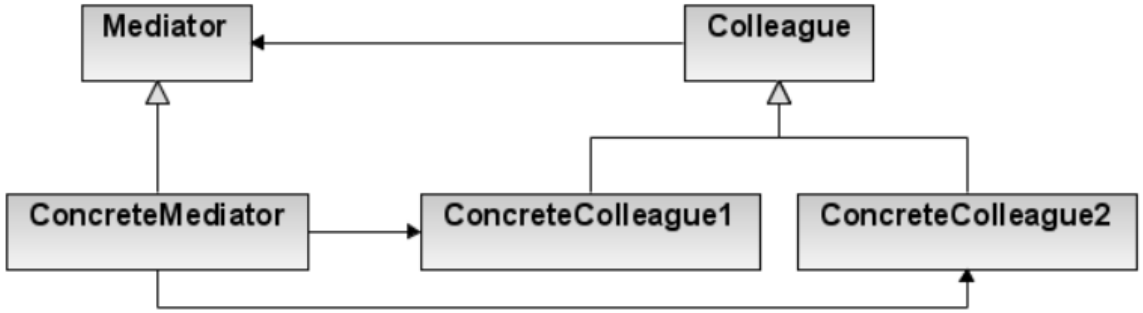


- **Havalimanları :** Uçak pilotları, bir sonraki uçağa kimin ineceğine karar verirken birbirleriyle doğrudan konuşmazlar. Tüm iletişim kontrol kulesinden geçer. Bu kulelerin tüm uçuşu kontrol etmediğini unutmamalıyız. Yalnızca terminal bölgelerinde kısıtlamalar uygularlar.
- **Chat Odaları:** Bir sohbet uygulamasında birkaç katılımcımız olabilir. Her katılımcıyı diğerlerine bağlamak iyi bir fikir değildir çünkü bağlantıların sayısı gerçekten çok yüksek olacaktır. En iyi çözüm, tüm katılımcıların bağlanacağı bir merkeze sahip olmaktır; bu hub mediator sınıfıdır.

JDK

- java.util.concurrent.Executor arayüzündeki execute () yöntemi bu kalıbı kullanır.
- java.util.Timer sınıfına ait schedule() yöntemlerinin farklı aşırı yüklenmiş (overloaded) sürümlerinin de bu kalıbı izlediği düşünülebilir.

UML



Mediator - Colleague nesneleri arasındaki iletişim için arayüzü tanımlar
ConcreteMediator - Mediator arayüzünü uygular ve Colleague nesneleri arasındaki iletişimi koordine eder.
Colleague - Mediator aracılığıyla diğer Colleague nesneleriyle iletişim için arayüzü tanımlar
ConcreteColleague - Colleague arayüzünü uygular ve Mediator aracılığıyla diğer Colleague nesneleri ile iletişim kurar

Tasarım Desenleri

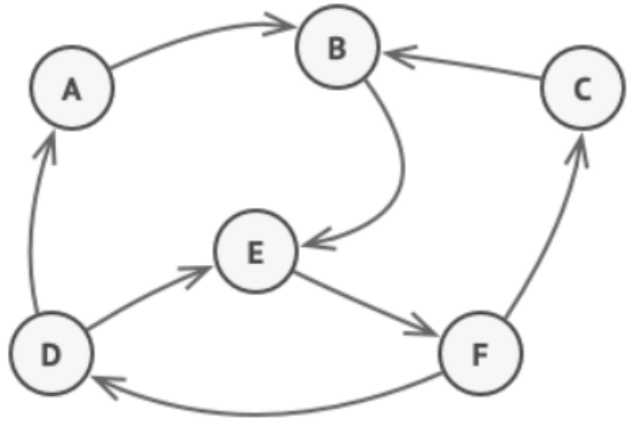
STATE PATTERN (DURUM)

DR. ŞAFAK KAYIKÇI

State Pattern

State (durum), bir nesnenin iç durumu değiştiğinde davranışını değiştirmesine izin veren davranışsal bir tasarım modelidir. Nesne, sınıfını değiştirmiş gibi görünür.

Finite State Machine (FSM -Sonlu Durum Makinesi) kavramı ile yakından ilgilidir.



Finite-State Machine.

State Pattern

Herhangi bir uygulamada, yaşam döngüsü boyunca farklı durumlarda olabilecek bir nesneyle ve mevcut durumuna göre gelen istekleri nasıl işlediğini (veya durum geçişlerini yaptığını) ele alındığında State Pattern kullanılabilir. Böyle bir durumda State Pattern kullanılmazsa, kod tabanında çirkin, gereksiz yere karmaşık ve bakımı zor hale getiren çok sayıda if-else ifadesi yer alacaktır.

Nesnenin olası her durumu için ayrı bir somut sınıf olacaktır. Her somut durum nesnesi, kendisine iletilen parametrelere bağlı olarak durumunu değiştirecek veya değiştirmeyecektir. Ayrıca, yeni durumların eklenmesi, mevcut durumların davranışını etkilememelidir.

Örnekler

TV : Uzaktan kumandadaki düğmelere basarak TV'nin durumunu değiştirilebilir. TV'nin mevcut durumuna bağlı olarak, AÇIK ise onu KAPATABİLİR, sessize alabilir veya özellikleri değiştirilebilir. Ancak TV KAPALI ise, uzaktan kumanda düğmelerine basıldığında bir şey olmayacaktır. KAPALI TV için yalnızca bir sonraki olası durum AÇIK konuma gelmesidir.

Fast Food Otomatları : Para ile alışveriş yapılan kahve yada fastfood makineleri

JDK

Threads : Java iş parçacıkları yaşam döngüsü boyunca beş durumundan biri olabilir. Bir sonraki durumu ancak mevcut durumu alındıktan sonra belirlenebilir. Örneğin. Durdurulmuş bir iş parçacığını başlatamayız veya bir iş parçacığı çalışmaya başlayana kadar bekleyemeyiz

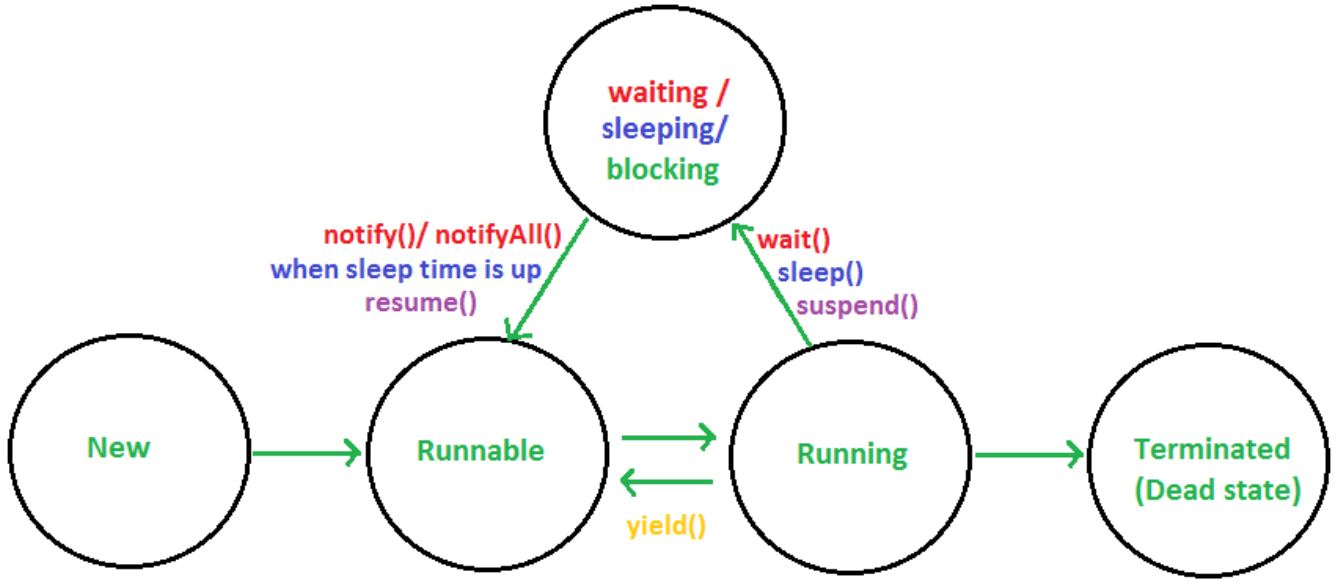
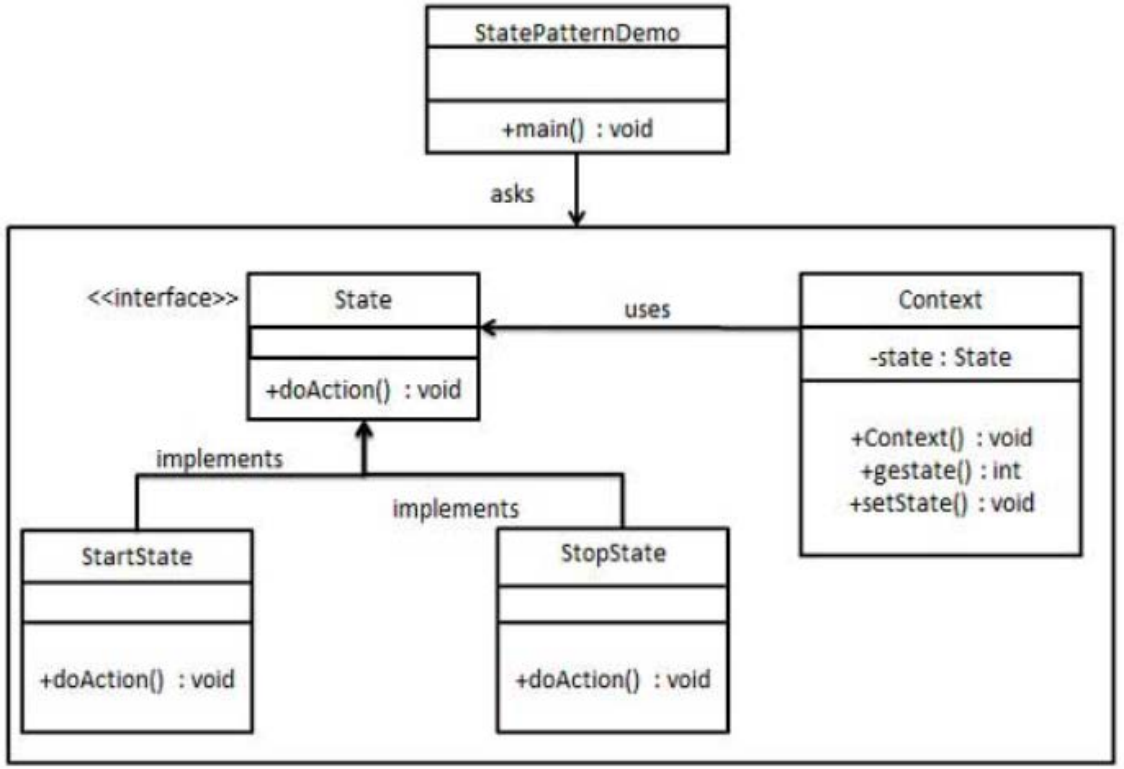


Fig. THREAD STATES

UML


StatePatternDemo
+main() : void
asks
<<interface>>
State
+doAction() : void
uses
Context
-state : State
+Context() : void
+gestate() : int
+setState() : void
implements
implements
StartState
+doAction() : void
StopState
+doAction() : void

Tasarım Desenleri

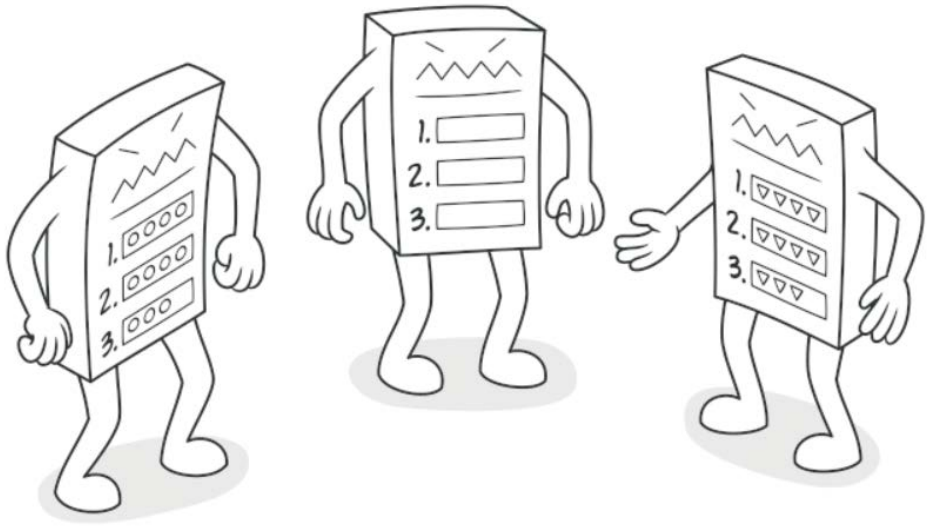
TEMPLATE PATTERN (KALIP-ŞABLON)

DR. ŞAFAK KAYIKÇI

Template Pattern

Template tasarım deseni, programlama bağlamında algoritmaların adımlarını uygulamak için yaygın olarak kabul edilen davranışsal tasarım modelidir .

Çok adımlı bir algoritmayı yürütmek için sıralı adımları tanımlar ve isteğe bağlı olarak varsayılan bir uygulama da sağlanabilir.



Template Pattern

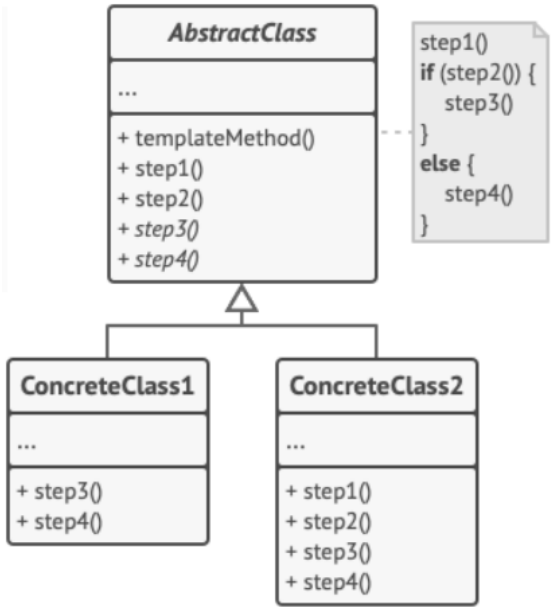
Şablon desen , programlama dünyasında belirli sıralı algoritma adımlarını tanımlamak ve uygulamak için kullanılan çok kolay bir tasarım modelidir. Alt sınıflarda ezilecek veya ezilmeyecek olan bir algoritmanın iskeletini tanımlamaya yardımcı olur.

İstemcilerin bir algoritmanın yalnızca belirli adımlarını genişletmesine izin vermek, ancak tüm algoritmayı veya yapısını genişletmek istemediğimizde kullanılır.

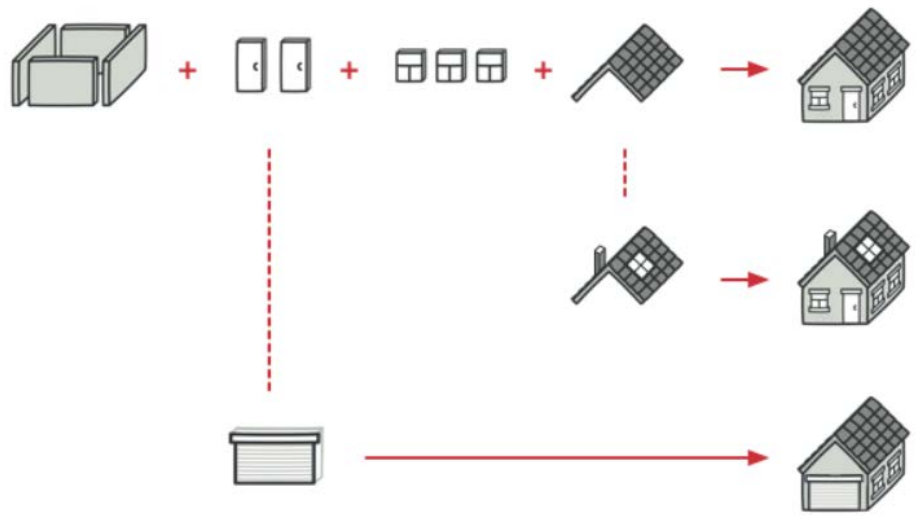
Bazı küçük farklarla neredeyse aynı algoritmaları içeren birkaç sınıf olduğunda modeli kullanın. Sonuç olarak, algoritma değiştiğinde tüm sınıfları değiştirmeniz gerekebilir.

Önemli Noktalar

- Template methodu final yapın : Böylece uygulayan sınıflar ezemez (override) ve adımların sırasını değiştiremez.
- Ana sınıfta, soyut (abstract) olmayan tüm yöntemlerin içini default kodu ile doldurun. Böylece alt sınıfların bunları tanımlaması gerekmez.
- Alt sınıfların uygulaması gereken tüm metotları soyut (abstract) yapın.



Örnek



Tipik bir mimari plan, müşterinin ihtiyaçlarına daha iyi uyması için biraz değiştirilebilir.

öntanımlı (default) yapılışı olan adımlar

- Temel inşaatı
- Çatı yapımı

öntanımlı (default) yapılışı olmayan adımlar

- Duvar yapımı
- Pencere ve kapıların yapımı
- Boyama
- İç dekorasyon

Amaç : Uygulamadaki tüm sınıfların bu **adımları sırasına uygun** olarak uygulaması

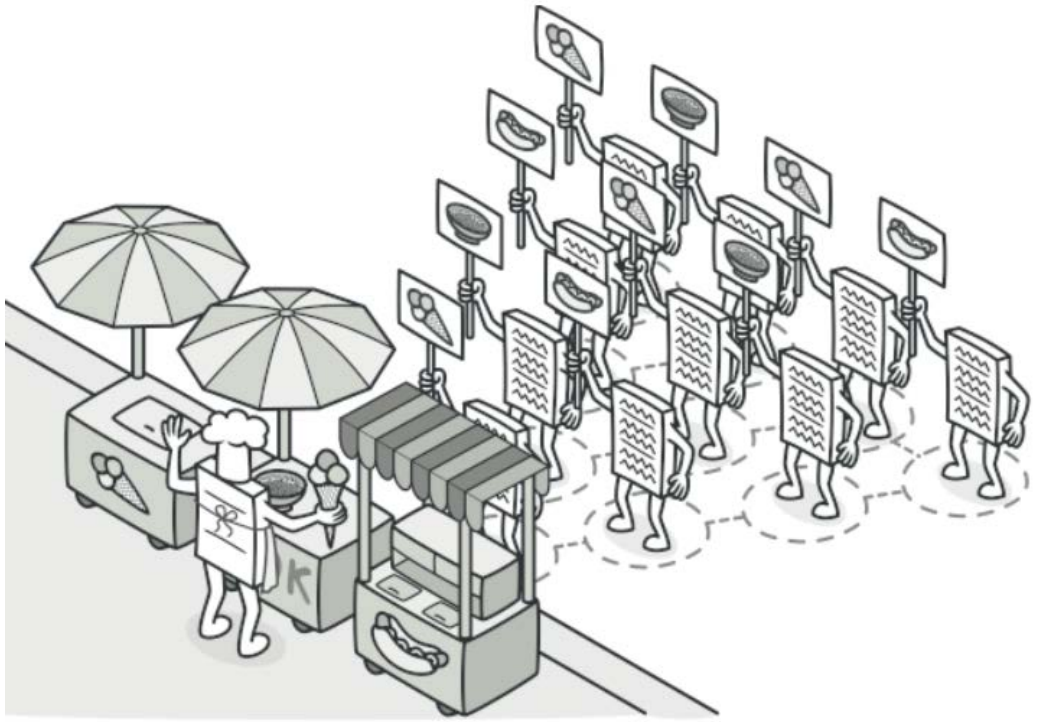
Tasarım Desenleri

VISITOR PATTERN (ZIYARETCI)

DR. ŞAFAK KAYIKÇI

Visitor Pattern

Ziyaretçi (visitor) çok sayıda ve farklı tipteki nesneler üzerinde işlem yapabilmek amacıyla kullanılır. İşlem yapılacak nesnelerde herhangi bir değişiklik yapılmaz.

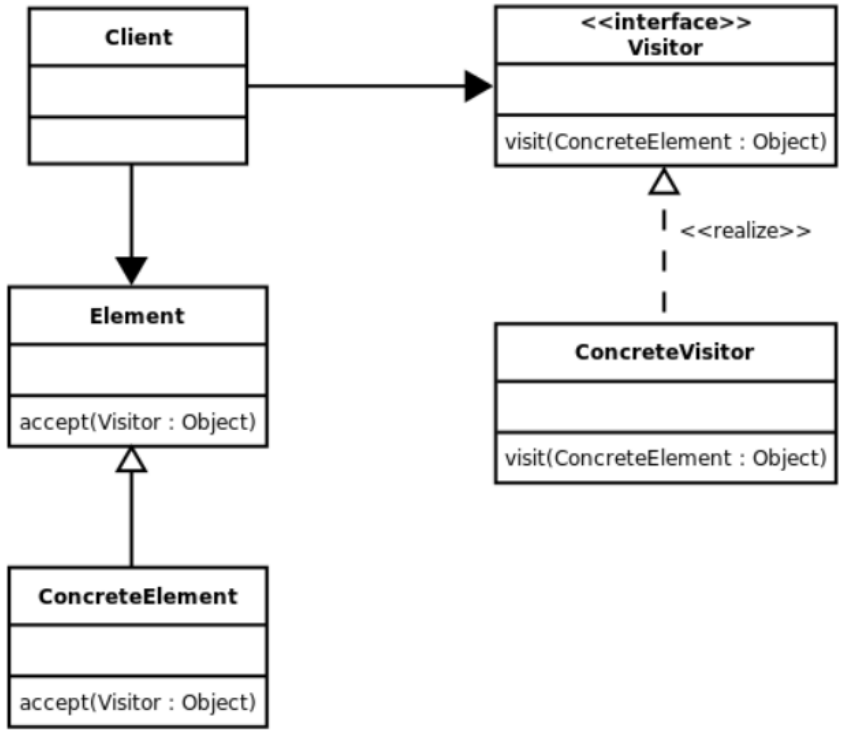


Visitor Pattern

Visitor (Ziyaretçi) Tasarım kalıbı popüler bir tasarım kalıbı değildir ancak yazdığımız sınıflara sonradan modüler olarak metodlar eklememizi sağlar. Sınıfın olmayan özelliğini sonradan eklemeyi kolaylaştırır

İşlemi ziyaretçi nesneleri yapar. Eğer sisteme yeni nesneler eklenmiyor, fakat sık sık yeni işlemlerin eklenmesi gerekiyorsa bu tasarım deseni kullanılabilir. Bu tasarım deseninin kullanılmasıyla, yapılacak işlemle ilgili kodların merkezi bir nesnede toplanır.

UML


Client
<<interface>>
Visitor
visit(ConcreteElement : Object)
<<realize>>
Element
accept(Visitor : Object)
ConcreteVisitor
visit(ConcreteElement : Object)
ConcreteElement
accept(Visitor : Object)

Visitor Pattern

Ziyaretçi Modeli Faydaları :

- İşlem mantığı değişirse, tüm öğe sınıflarında yapmak yerine sadece ziyaretçi uygulamasında değişiklik yapılır
- Sisteme yeni bir öğe eklemenin kolaydır. Yalnızca ziyaretçi arayüzünde ve uygulamasında değişiklik gerektirir ve mevcut öğe sınıfları etkilenmez.

Ziyaretçi Modeli Kısıtlamaları:

- Tasarım sırasında visit() yöntemlerinin dönüş türünün bilinmesi gerekir, aksi takdirde arayüzü ve tüm uygulamalar değiştirilir.
- Ziyaretçi arayüzünün çok fazla uygulaması varsa, genişletmeyi zorlaştırmasıdır.

Örnek – Alışveriş Sepeti

Farklı türde öğeler ekleyebileceğimiz bir alışveriş sepeti düşünün. Ödeme butonuna tıkladığımızda ödenecek toplam tutarı hesaplanır.

Bu hesaplama işlemini her öğe sınıfının içerisinde yapabiliriz veya bu mantığı ziyaretçi kalıbını kullanarak başka bir sınıfa taşıyabiliriz.