TASARIM KALIPLARI

Tasarım Kalıbı Nedir

- Sık karşılaşılan tasarım problemlerinin kalıplaşmış çözümleridir
- Kalıp, belirli bir bağlamda karşılaşılan genel bir soruna yeniden kullanılabilir bir çözümün taslağıdır.
- Birçoğu, tüm yazılım geliştiricilerin kullanması için sistematik olarak kanıtlanmıştır
- İyi bir kalıp için:
  - Mümkün olduğunca geneldir
  - Belirtilen bağlamda sorunu etkin bir şekilde çözdüğü kanıtlanmış bir çözüm içerir.
  - Kalıpları incelemek, başkalarının deneyimlerinden öğrenmenin etkili bir yoludur

GoF Kalıpları

- Bu grupta 23 adet kalıp var

Temel Türleri

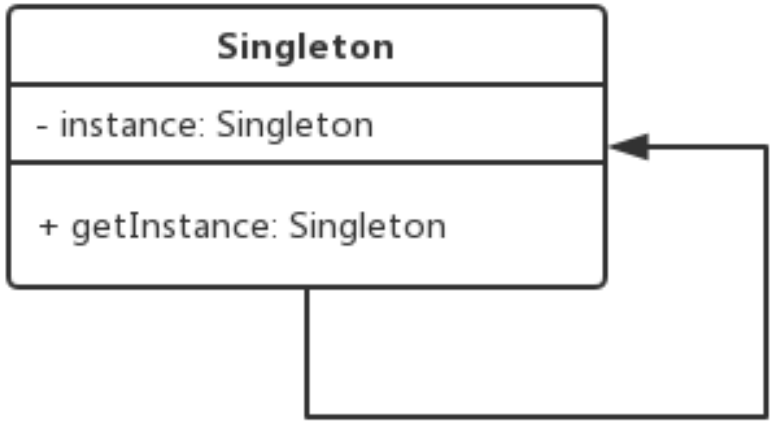
1. Nesne oluşturulmasına ilişkin kalıplar (Creational pattern)
2. Yapısal kalıplar (Structured pattern)
3. Davranışsal kalıplar (Behavioral pattern)

Nesne oluşturulmasına ilişkin kalıplar (Creational pattern)

1. Singleton : Bir nesnenin yalnızca bir örneğinin oluşturulmasını sağlar.
2. Factory : Oluşturulacak tam sınıfı belirtmeden nesneler oluşturur
3. Prototype : Mevcut bir nesneden yeni bir nesne oluşturur
4. Builder : Karmaşık nesneler oluşturmak için kullanır
5. Abstract factory : Somut türünü belirtmeden nesnelerin oluşturulmasına izin verir

Singleton

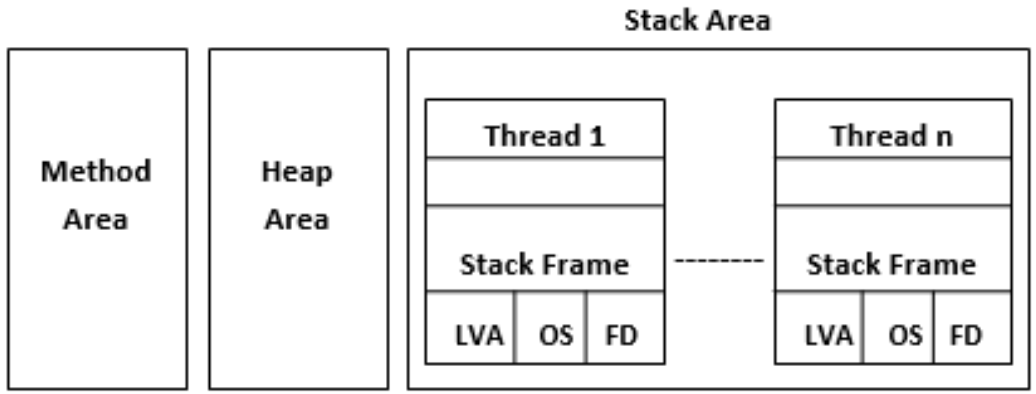
- Bir sınıftan kaç nesne üretildiğini kontrol etmek istediğimiz zaman kullanılır.



- ADIMLARI:
  1. Kendi tipinde değişken tut
  2. Private constructor tanımla
  3. getInstance metodunda tuttuğu değişken null mu kontrol et, değilse nesne oluştur

Thread safe singleton

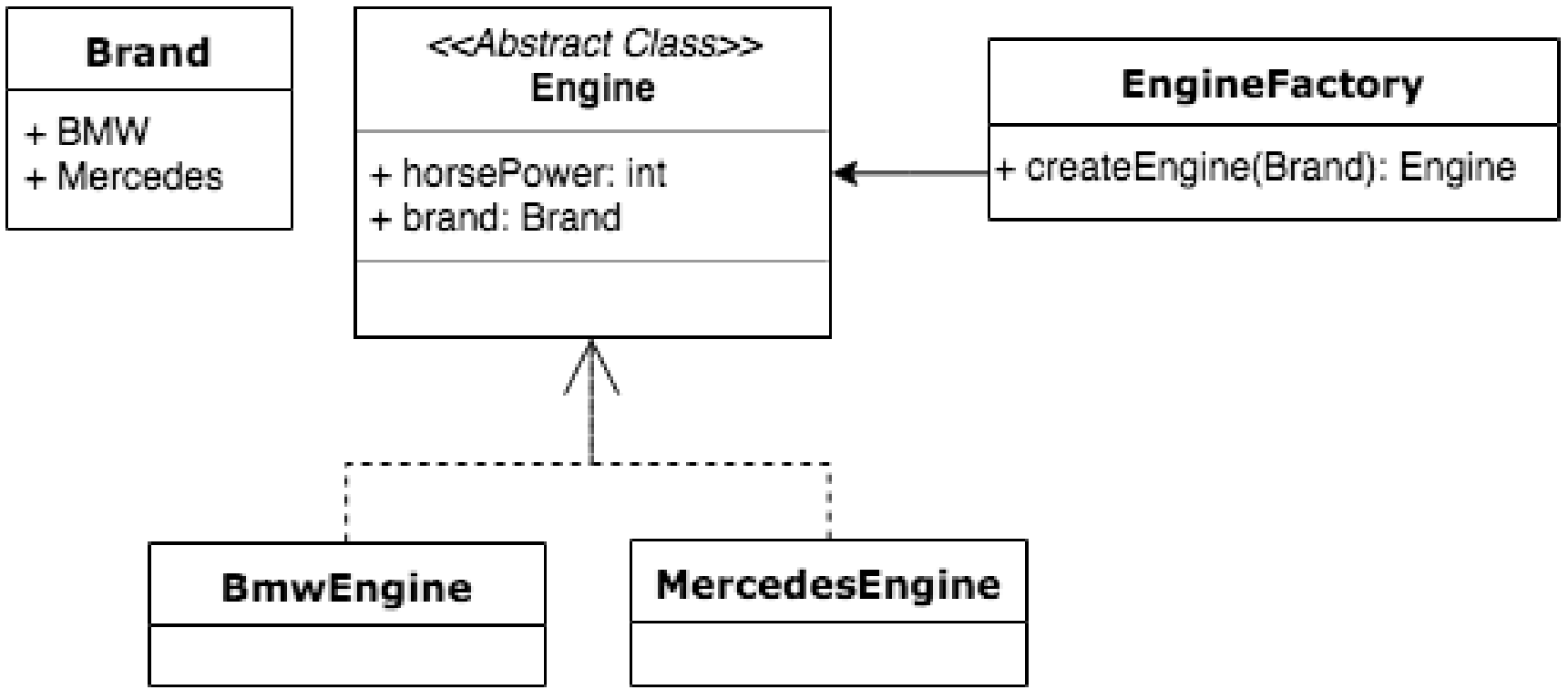
- Programda bir den fazla kanal varsa singleton sınıftan birden fazla nesne türetilmesi söz konusu olabilir.



ÇÖZÜM : **`synchronized`**

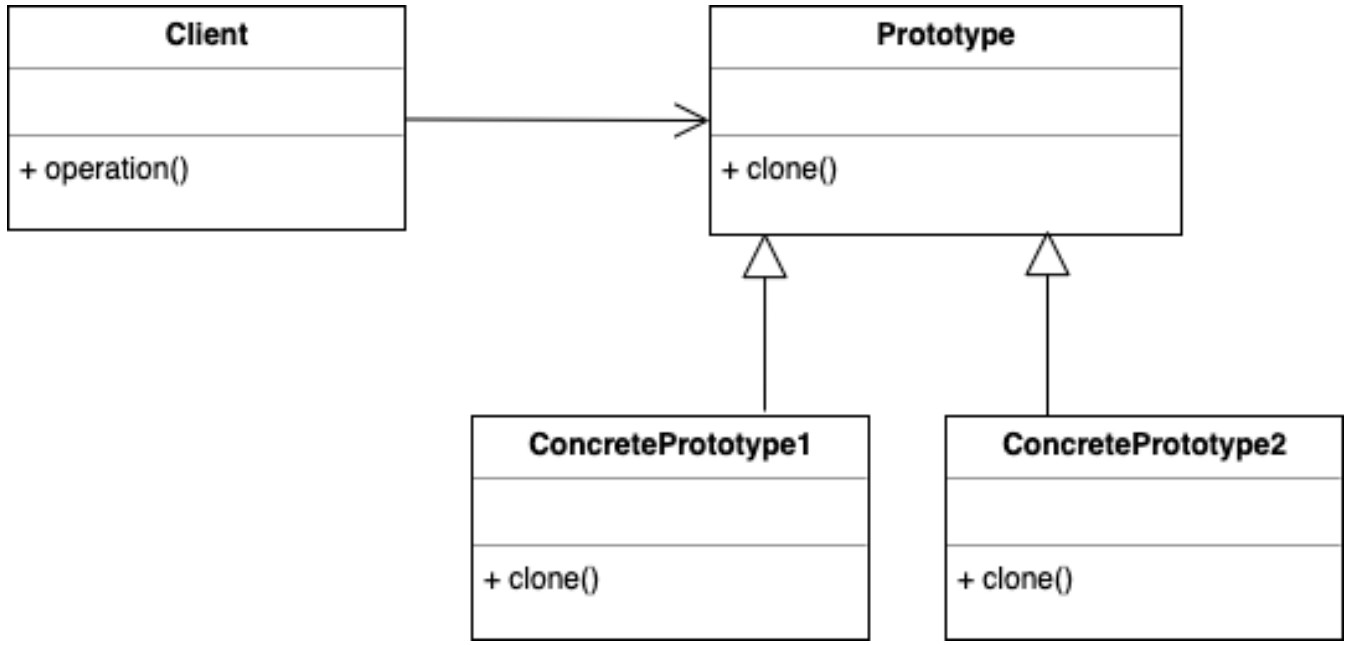
Factory

- Bir nesneyi oluşturmanın en iyi yolunu sağlar.
- Oluşturma mantığını istemciye göstermeden nesne oluşturuyoruz ve ortak bir arayüz kullanarak yeni oluşturulan nesneye atıfta bulunuyoruz.


Brand
+ BMW
+ Mercedes
<<Abstract Class>>
Engine
+ horsePower: int
+ brand: Brand
EngineFactory
+ createEngine(Brand): Engine
BmwEngine
MercedesEngine

Prototype

- Bazen bir nesneyi üretmek çok maliyetli olabilir.
  - Veritabanı işlemleri
  - Büyük işlemci gücü gerektiren hesaplara girilmesi
  - Ağ haberleşme işlemleri.
  - Vs.
- Bu durunda nesne program çalıştırılırken bir kez prototip olarak oluşturulur
- Yeni nesne gerektiğinde bu nesne kopyalanır.



Bir nesnenin kopyalanması? (clone)

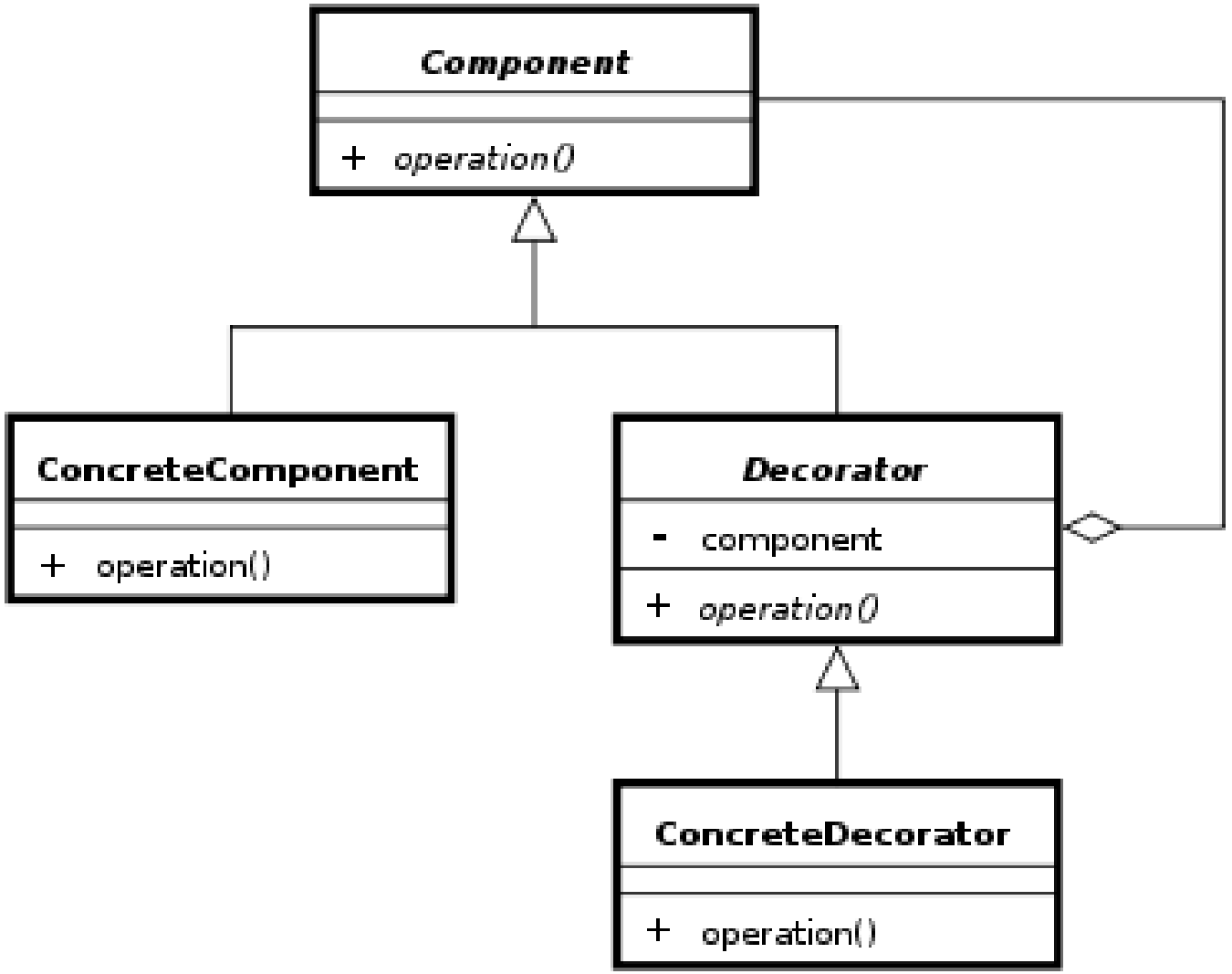
- Shallow copy
- Deep copy

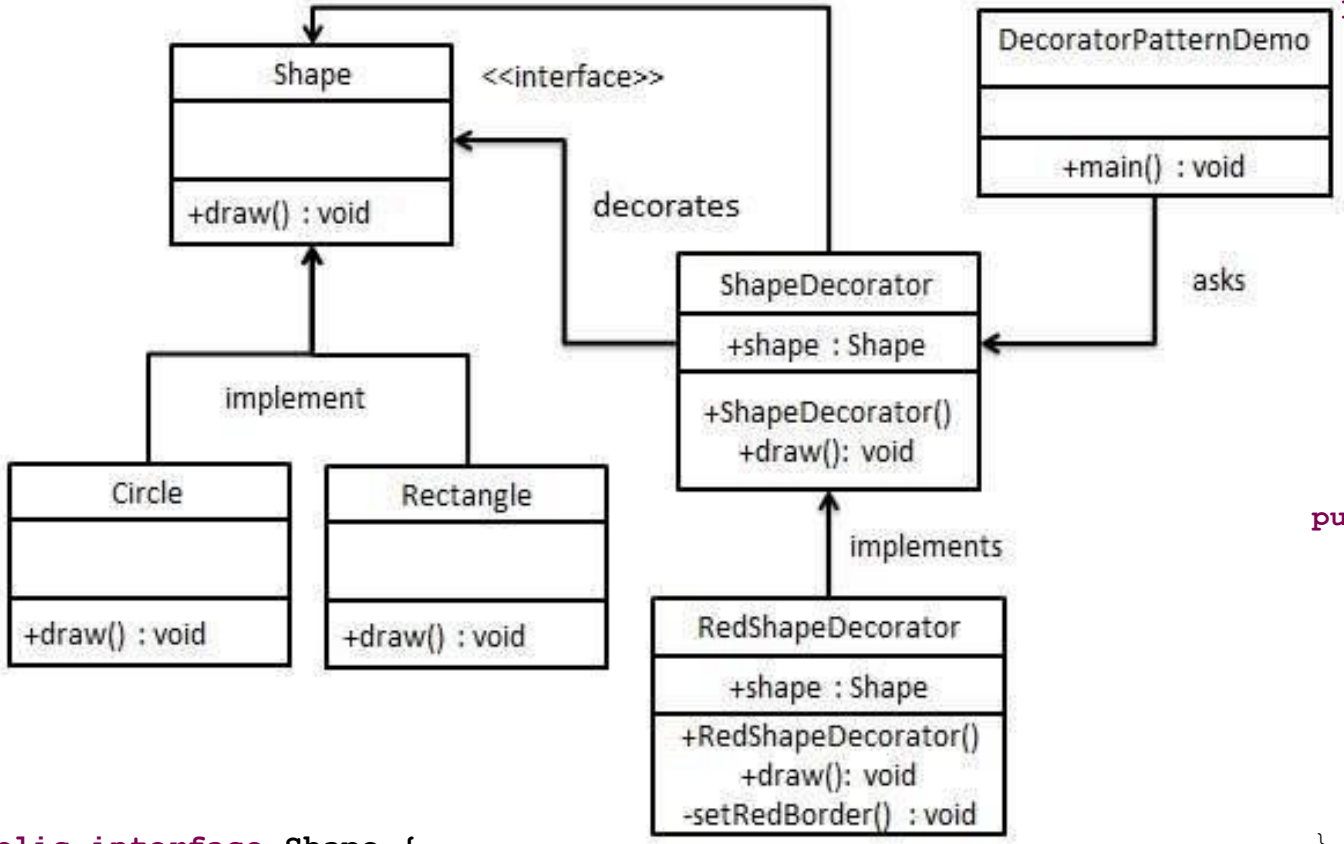
Yapısal kalıplar (structured patterns)

1. Decorator : Bir nesnenin davranışının çalışma zamanında dinamik olarak genişletilmesine izin verir.
2. Adapter: Mevcut sınıflardan birini bir arabirim ile sarmalayarak uyumsuz iki sınıfın birlikte çalışmasına izin verir.
3. Facade : Daha karmaşık bir temel nesneye basit bir arabirim sağlar.
4. Proxy : Erişimi kontrol etmek, maliyeti azaltmak veya karmaşıklığı azaltmak için temel alınan bir nesneye yer tutucu arabirimi sağlar.
5. Bridge: Bir soyutlamayı ayırır, böylece iki sınıf bağımsız olarak değişebilir.
6. Compozite : Bir grup nesneyi tek bir nesneye alır.
7. Flyweight : Karmaşık nesne modellerinin maliyetini azaltır.

Decorator

- Var olan bir nesneye yeni özellik eklemek için kullanılır.
- Decorator fonksiyonelliği artırmak için mirasa alternatiftir.






```java
public interface Shape {
   void draw();
}

public class Circle implements Shape {
   public void draw() {
      System.out.println("Shape: Circle");
   }
}

public class Rectangle implements Shape {
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

```java
public static void main(String[] args) {
   Shape circle = new Circle();
   Shape redCircle = new RedShapeDecorator(new Circle());
   Shape redRectangle = new RedShapeDecorator(new Rectangle());
   System.out.println("Circle with normal border");
   circle.draw();

   System.out.println("\nCircle of red border");
   redCircle.draw();

   System.out.println("\nRectangle of red border");
   redRectangle.draw();
}
```

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }
   public void draw(){
      decoratedShape.draw();
   }
}
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

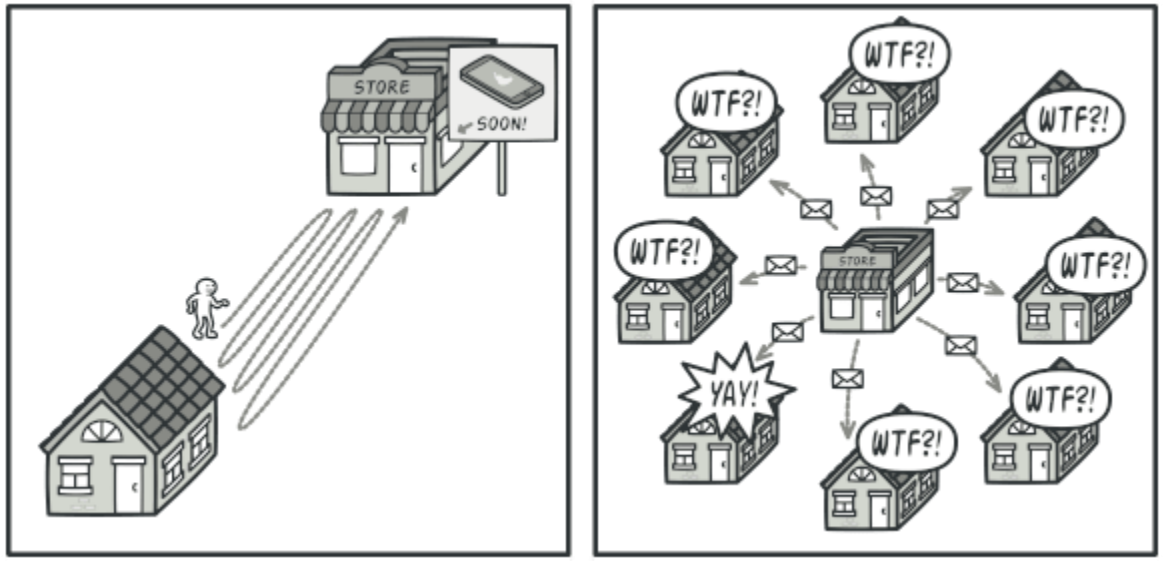
Davranışsal kalıplar (Behavioral pattern)

1. Chain of Responsibility: Komutları işleme nesneleri zincirine devreder.
2. Command: Eylemleri ve parametreleri içine alan nesneler oluşturur.
3. Interpreter: Özel bir dil uygular.
4. Iterator: Bir nesnenin öğelerine, temeldeki temsilini göstermeden sırayla erişir.
5. Mediator: Yöntemleri hakkında ayrıntılı bilgiye sahip tek sınıf olarak sınıflar arasında gevşek bağlantıya izin verir.
6. Momento: Bir nesneyi önceki durumuna geri yükleme yeteneği sağlar.
7. Observer: Bir dizi gözlemci nesnenin bir olayı görmesine izin veren bir yayınlama/abone olma modelidir.
8. State: Bir nesnenin, dahili durumu değiştiğinde davranışını değiştirmesine izin verir.
9. Strategy Algoritma ailesinden birinin çalışma zamanında anında seçilmesine izin verir.
10. Template Method: Bir algoritmanın iskeletini soyut bir sınıf olarak tanımlar ve alt sınıflarının somut davranış sağlamasına izin verir.
11. Visitor: Yöntemlerin hiyerarşisini tek bir nesneye taşıyarak bir algoritmayı nesne yapısından ayırır.

Observer

- Müşterinin ilgilendiği telefon fiyatlarını öğrenmek için yapabilecekleri
  - Müşteri her mağazayı periyodik gezebilir
  - Mağaza müşteriye ilgilenebileceği ürünler için mail atabilir

! Ya müşteri, ürünün kullanılabilirliğini kontrol etmek için zaman harcıyor ya da mağaza, yanlış müşterilere bildirimde bulunarak kaynakları boşa harcıyor.

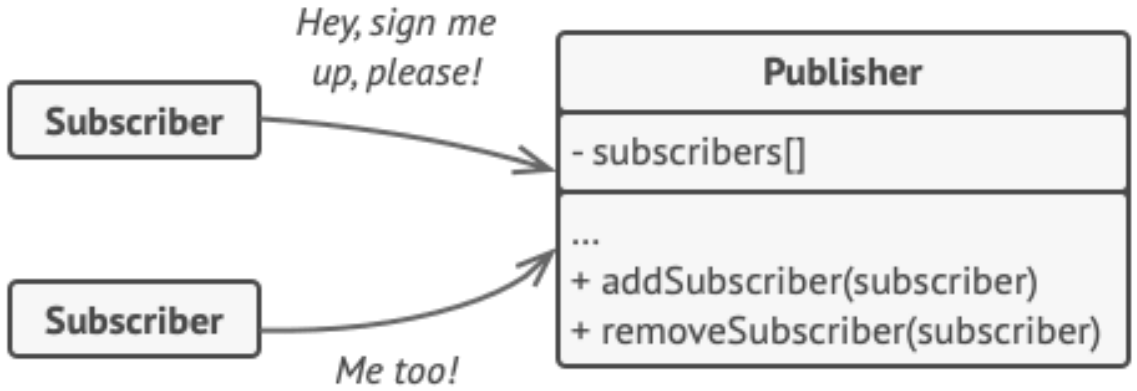


Kaynak : https://refactoring.guru/design-patterns/observer

Observer

- Durumundaki değişiklikleri diğer nesnelere de bildiren *yayıncı (publisher)* nesne
- Yayıncının durumundaki değişiklikleri izlemek isteyen diğer tüm nesnelere *aboneler (observer)* denir
- Observer modeli, yayıncı sınıfına bir abonelik mekanizması eklemenizi sağlar,
- Nesneler o yayıncıdan gelen bir olay akışına :
  - abone olabilir.
  - abonelikten çıkabilir.
- Bu mekanizma
  1) abone nesnelerine yapılan referansların bir listesini depolamak için bir dizi alanından
  2) bu listeye abone eklemeye ve onları listeden çıkarmaya izin veren birkaç genel yöntemden

  oluşur

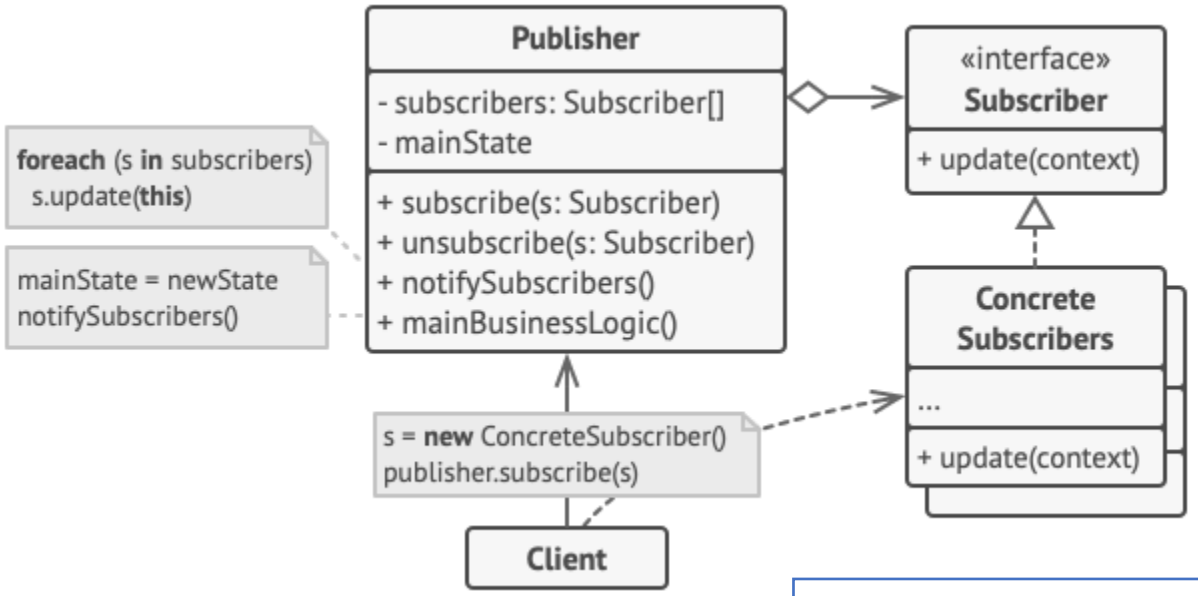


Kaynak : https://refactoring.guru/design-patterns/observer

Yayıncı , diğer nesneleri ilgilendiren olayları yayınlar. Bu olaylar, yayıncı durumunu değiştirdiğinde veya bazı davranışları yürüttüğünde gerçekleşir. Yayıncılar, yeni abonelerin listeye katılmasını ve mevcut abonelerin listeden çıkmasını sağlayan bir abonelik altyapısı içerir.

Yeni bir olay olduğunda, yayıncı abonelik listesinin üzerinden geçer ve her abone nesnesinde abone arayüzünde belirtilen bildirim yöntemini çağırır.

Abone arabirimi, **bildirim** arabirimini bildirir. Çoğu durumda, tek bir update yöntemden oluşur. Yöntem, yayıncının güncellemeyle birlikte bazı olay ayrıntılarını iletmesine izin veren birkaç parametreye sahip olabilir.

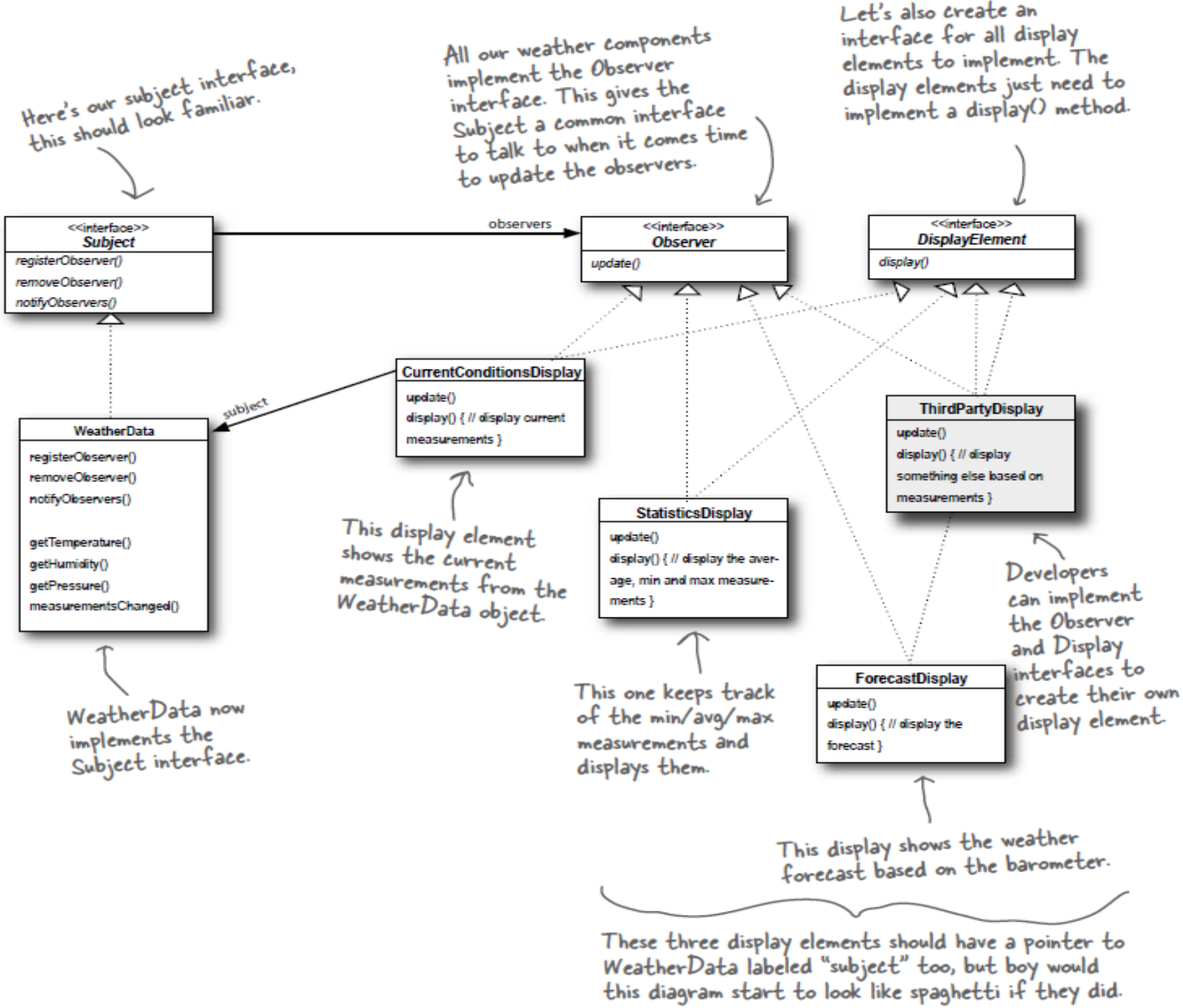


Somut Aboneler , yayıncı tarafından yapılan bildirimlere karşılık olarak bazı işlemler gerçekleştirir. Yayıncının somut sınıflara bağlanmaması için bu sınıfların tümü aynı arabirimi uygulamalıdır.

Müşteri , yayıncı ve abone nesnelerini ayrı **ayrı** oluşturur ve ardından yayıncı güncellemeleri için aboneleri kaydeder.

Genellikle, aboneler güncellemeyi doğru bir şekilde işlemek için bazı bağlamsal bilgilere ihtiyaç duyar. Bu nedenle, yayıncılar genellikle bazı bağlam verilerini bildirim yönteminin bağımsız değişkenleri olarak iletir. Yayıncı, abonenin gerekli verileri doğrudan almasına izin vererek kendisini bir bağımsız değişken olarak iletebilir.

Kaynak : https://refactoring.guru/design-patterns/observer


Here's our subject interface, this should look familiar.
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.
Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
subject
CurrentConditionsDisplay
update()
display() { // display current measurements }
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
ForecastDisplay
update()
display() { // display the forecast }
WeatherData now implements the Subject interface.
This display element shows the current measurements from the WeatherData object.
This one keeps track of the min/avg/max measurements and displays them.
Developers can implement the Observer and Display interfaces to create their own display element.
This display shows the weather forecast based on the barometer.
These three display elements should have a pointer to WeatherData labeled "subject" too, but boy would this diagram start to look like spaghetti if they did.

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Here we implement the Subject Interface.

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.