Nesneye Yönelik Yazılım Mühendisliği (376)

Dr. Öğr. Üyesi Ahmet Arif AYDIN

Prototype (Creational) Design Pattern

- Nesne tabanlı programlamada nesneyi oluşturmak , davranışlarını tanımlamak ve ***oluşturulacak nesne sayısını da kontrol altından tutmak çok önemlidir***.

- Bir nesnenin tekrar oluşturulması sistem açısından *maliyet* (zaman, kaynaklar, RAM) oluşturacağı durumlarda var olan bir nesnenin ihtiyaclar doğrultusunda **kopyalanarak (clone)** ve yeni özellikler ekleyerek(polymorphism) oluşturmayı sağlayan tasarım kalıbı Prototype'dır

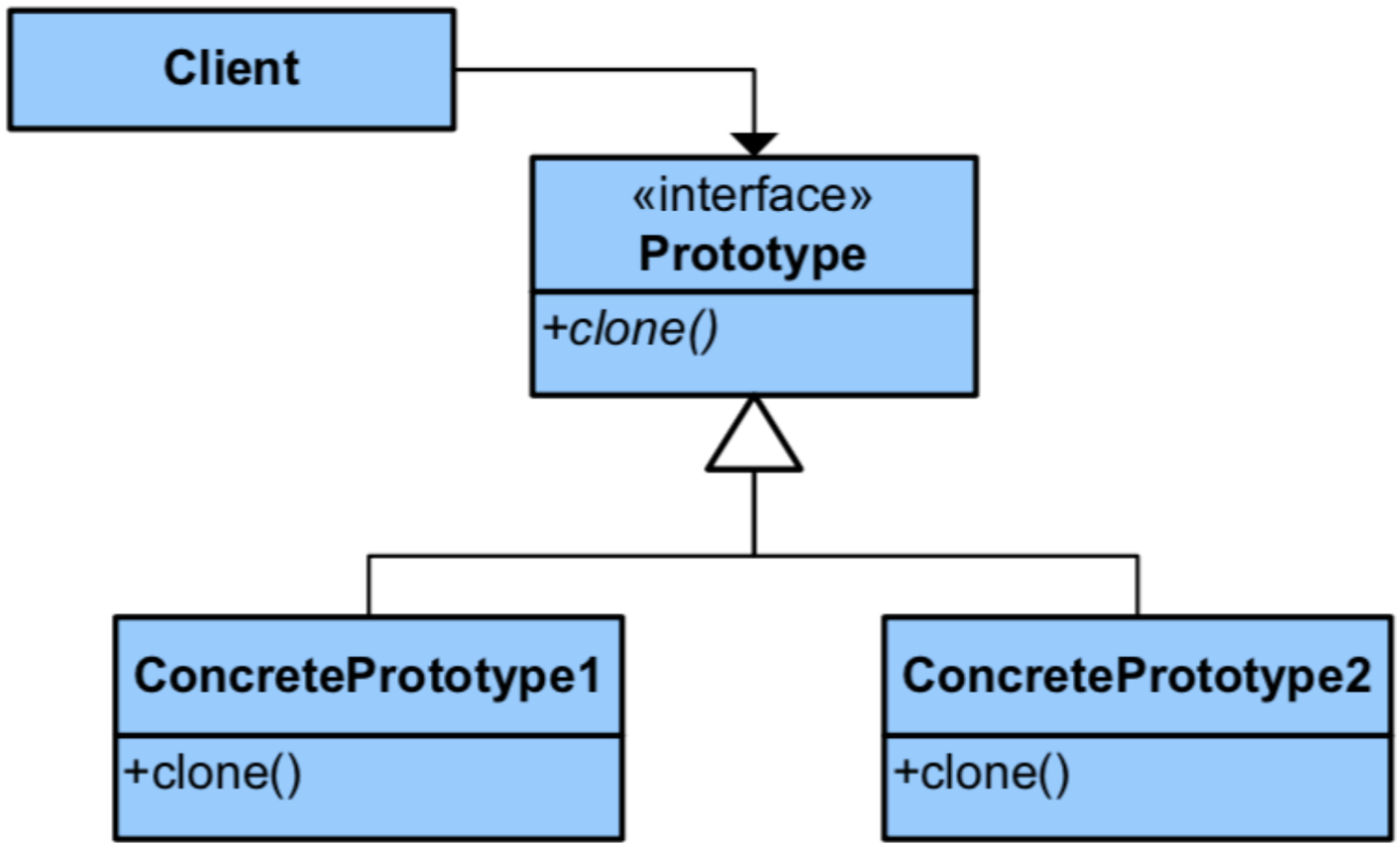
- Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing
- Prototype provides a mechanism ***to copy the original object to a new object*** and t**hen modify it according to our needs**.
- Prototype design pattern uses **java cloning** to copy the object

Prototype (Creational) Design Pattern

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Prototype (Creational) Design Pattern

```
import java.util.ArrayList;
import java.util.List;

public class Employees implements Cloneable{

    private List<String> empList;

        public Employees(){
            empList = new ArrayList<String>();
        }
        public Employees(List<String> list){
             this.empList=list;
        }
        public void loadData(){
                //read all employees from database and put into the list
                empList.add("A");
                empList.add("B");
                empList.add("C");
                empList.add("D");
        }

        public List<String> getEmpList() {
                return empList;
        }

        @Override
        public Object clone() throws CloneNotSupportedException{
                        List<String> temp = new ArrayList<String>();
                        for(String s : this.getEmpList()){
                                temp.add(s);
                        }
                        return new Employees(temp);
        }

}
```

Prototype (Creational) Design Pattern

```
import java.util.List;


public class Test {

        public static void main(String[] args) throws
                CloneNotSupportedException {

            Employees emps = new Employees();
                emps.loadData();

                //Use the clone method to get the Employee object
                Employees empsNew = (Employees) emps.clone();

                Employees empsNew1 = (Employees) emps.clone();

                List<String> list = empsNew.getEmpList();

                list.add("K");
                List<String> list1 = empsNew1.getEmpList();
                list1.remove("C");

                System.out.println("emps List: "+emps.getEmpList());
                System.out.println("empsNew List: "+list);
                System.out.println("empsNew1 List: "+list1);
        }

}
```

Prototype (Creational) Design Pattern

- Bir nesnenin oluşturulması ıle alakalı gelen zaman ortadan kaldırır (eliminates the (potentially expensive) overhead of initializing an object)

- Aynı verı üzerinde calışan nesnelerın kullanımını kolaylaştırır (It simplifies and can optimize the use case where **multiple objects of the same type** will have mostly the same data)

Bridge (Structural) Design Pattern

- Bridge pattern oluşturulan sınıf yapısı içerisinde soyut interface ve implementation(kod) kısımlarını birbirimden bağımsız ve hiyerarşik olarak tanımlamaya imkan sağlar.

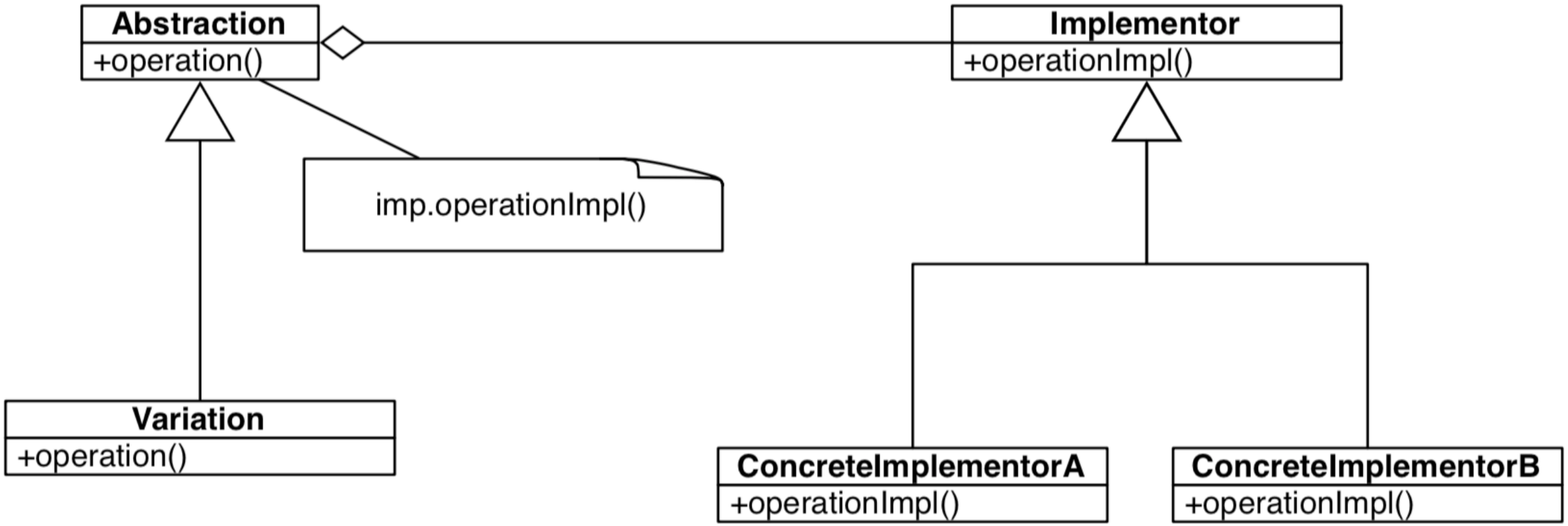
- interface hierarchies in both interfaces as well as implementations, then **bridge design pattern** is used to *decouple the interfaces from implementation and hiding the implementation details from the client programs*.

Bridge (Structural) Design Pattern

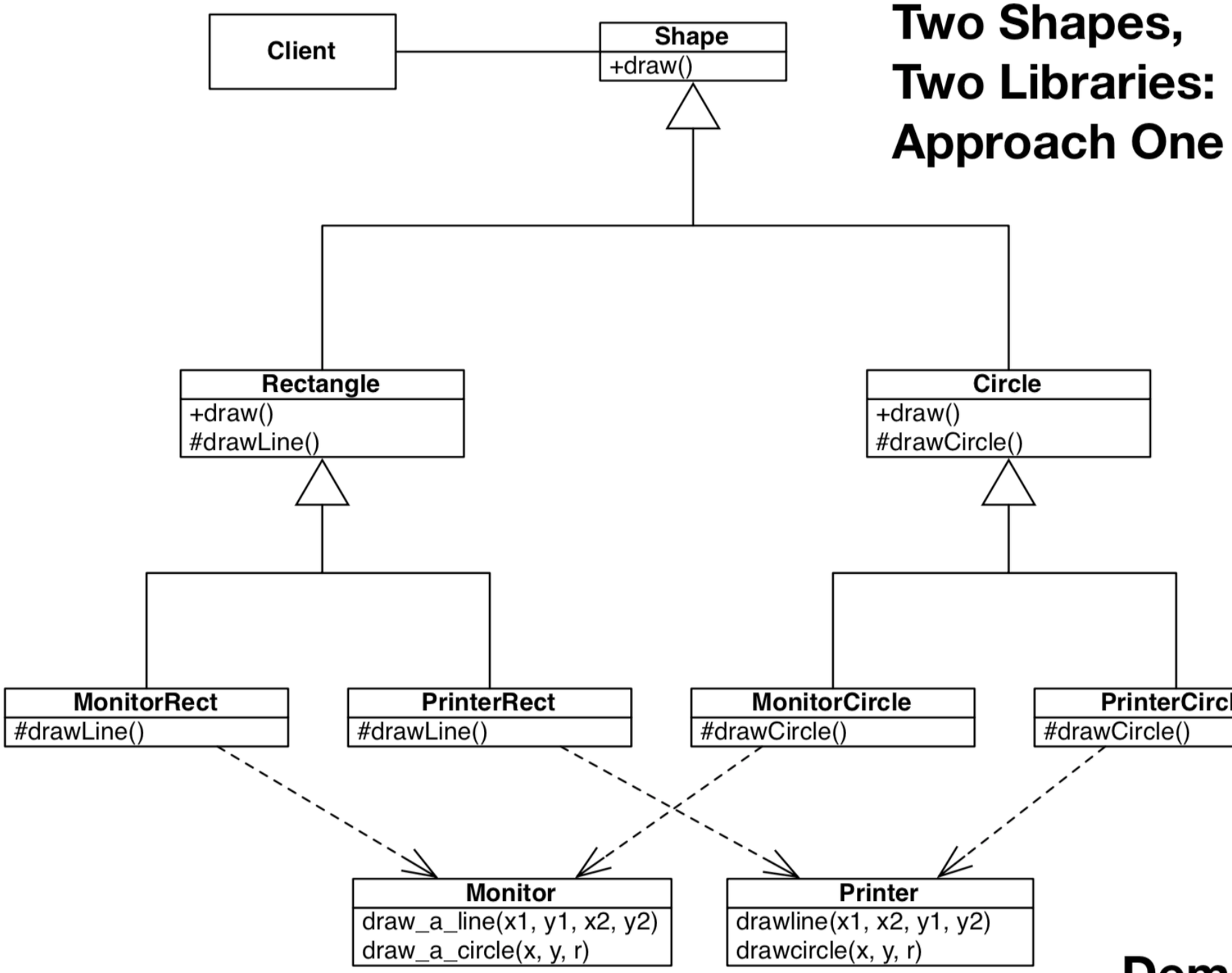
- The Gang of Four says the intent of bridge pattern is to *"decouple an abstraction from its implementation so that the two can vary independently"*

- Oluştrulacak nesnelerin ölçeklenebilir bir biçimde artmasını saglar (Allows a set of abstract objects to implement their operations in a number of ways in a scalable fashion)

Bridge (Structural) Design Pattern



Bridge (Structural) Design Pattern



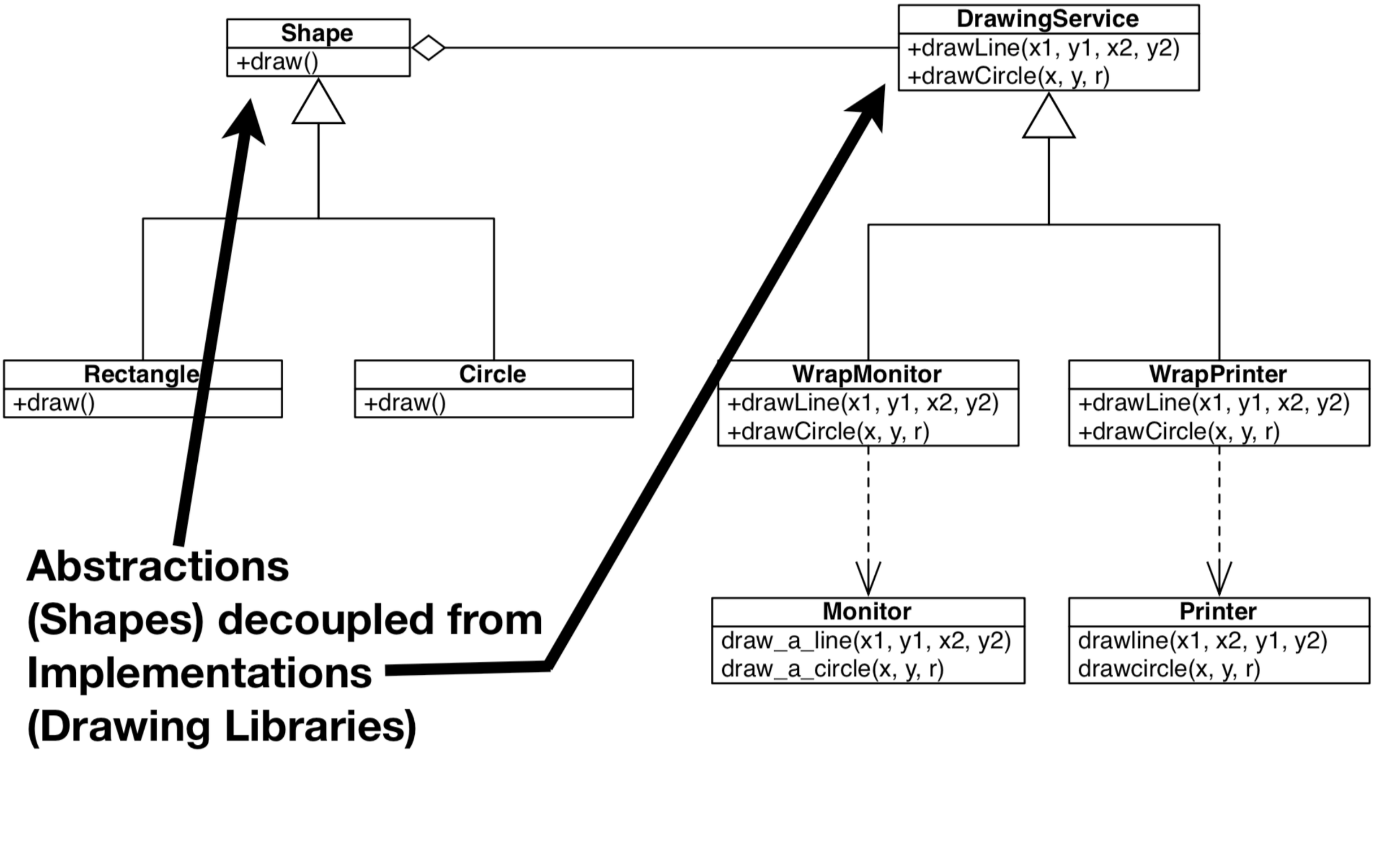
Two Shapes, Two Libraries: Approach One

Bridge Olmadan Gerçekleştirilmiş!!!!

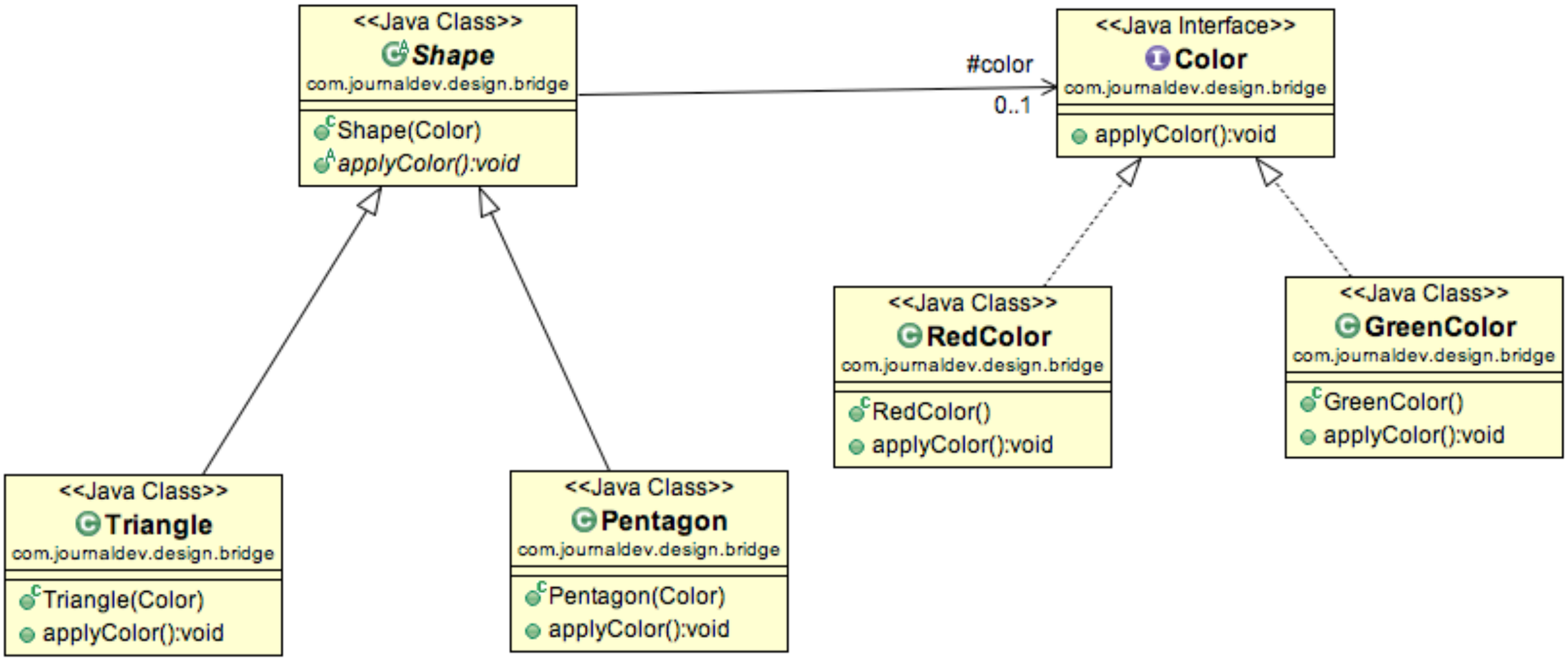
Demo

Geometrik sekilleri Monitor ve Printer'a yazdırılmasını sağlayan bir uygulama

Bridge (Structural) Design Pattern



Bridge (Structural) Design Pattern



Bridge (Structural) Design Pattern

```
public abstract class Shape {
        //Composition - implementor
        protected Color color;

        //constructor with implementor as input argument
        public Shape(Color c){
                this.color=c;
        }

        abstract public void applyColor();
}
```

```
public interface Color {

        public void applyColor();
}
```

```
public class Pentagon extends Shape{

        public Pentagon(Color c) {
                super(c);
        }

        @Override
        public void applyColor() {
                System.out.print("Pentagon filled with color ");
                color.applyColor();
        }

}
```

```
public class GreenColor implements Color{

        public void applyColor(){
                System.out.println("green.");
        }
}
```

https://www.journaldev.com/1491/bridge-design-pattern-java

Observer (Behavioral) Design Pattern

- Bir nesnenin durumunda olan bir değişikliği nesneyi takip eden kullanıcıların otomatik olarak nesnede olna değişiklikten haberdar edilmesini sağlar. (*Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change*)

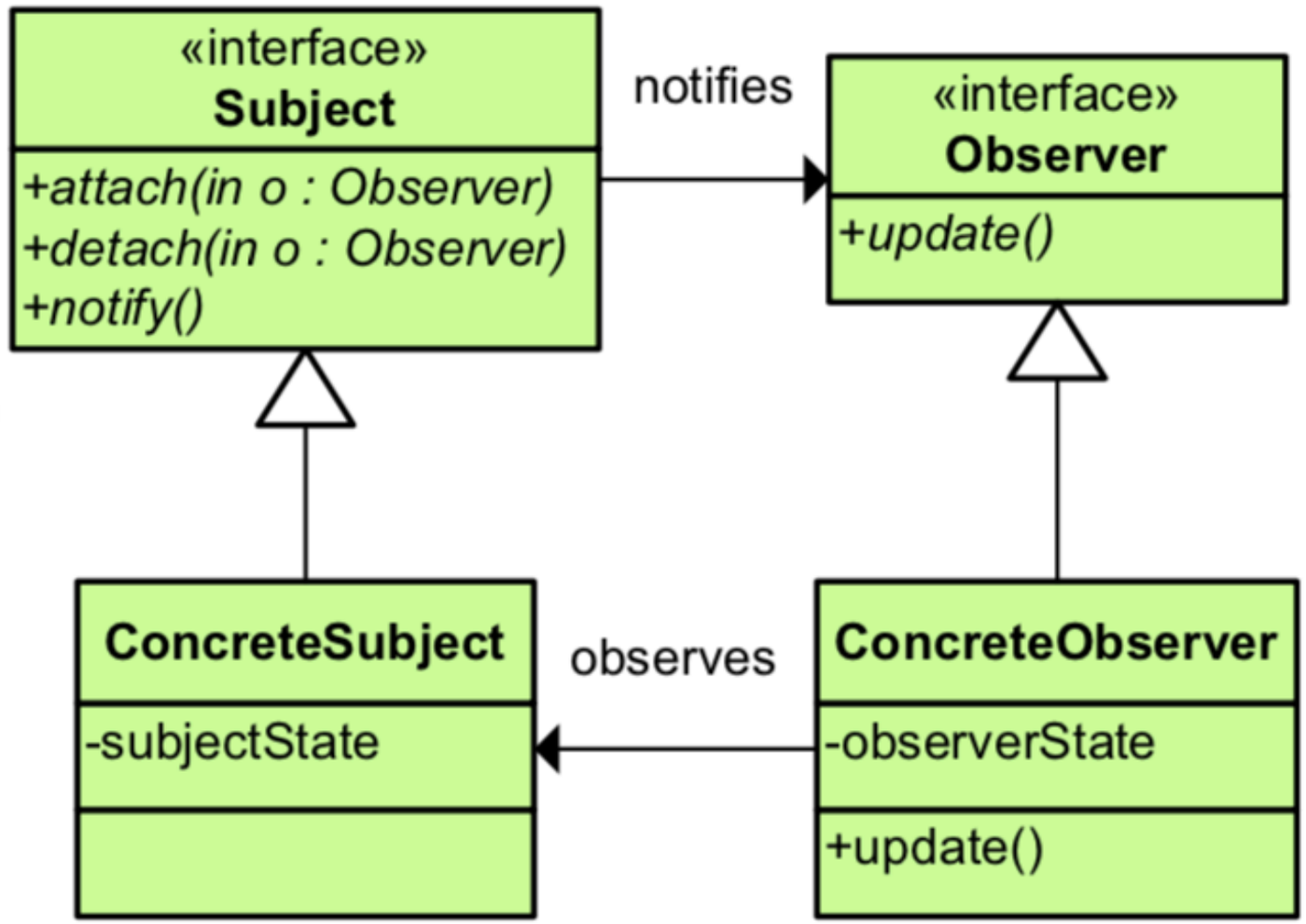
    - Define a **one-to-many dependency between objects** so that when one object changes state, all its dependents are notified and updated automatically
    - the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

Observer (Behavioral) Design Pattern

Observer

Type: Behavioral

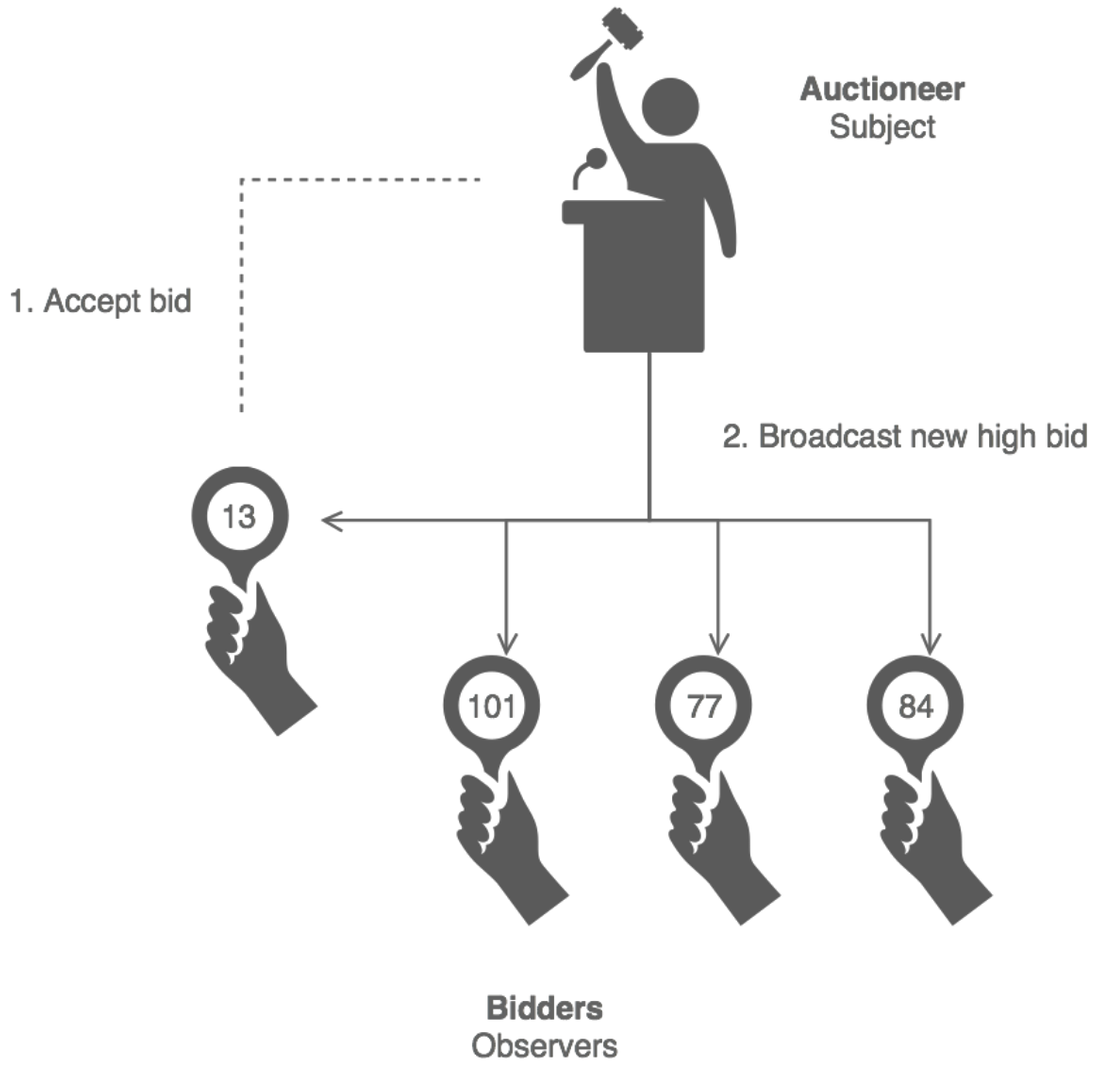
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

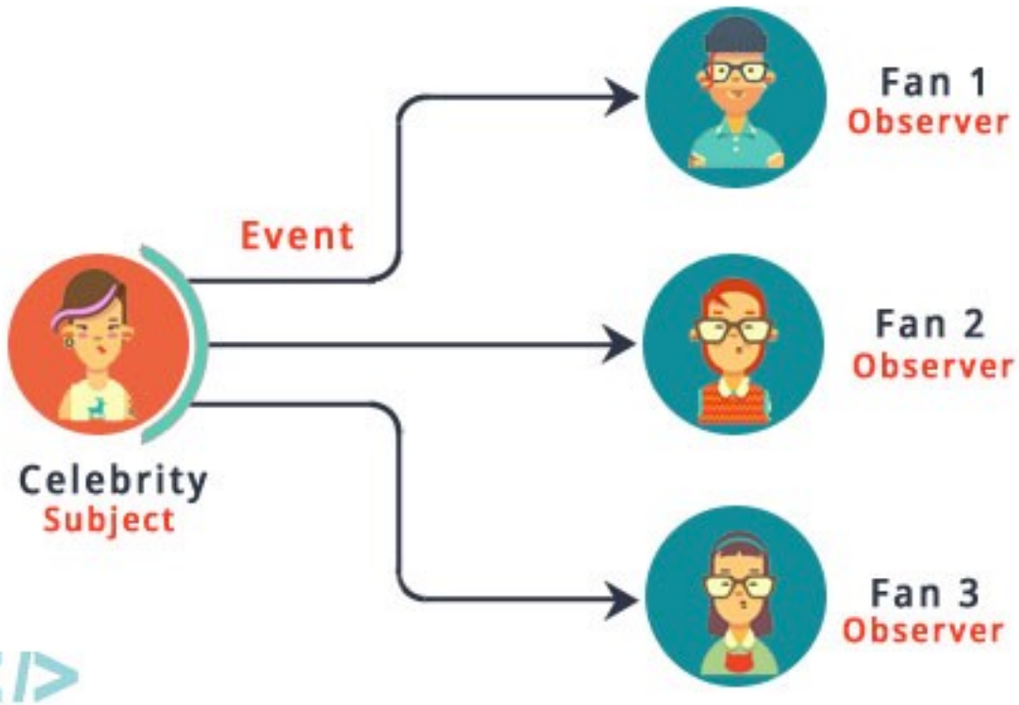


Observer (Behavioral) Design Pattern

- Bir futbol sitesinde bulunan üyeler
  - maçlarda gelişen anlık skor değişikliklerden haberdar edilebilirler
  - kullanıcıların takip ettikleri ligler olabilir
  - her bir kullanıcı takip ettiği takımları seçebilir
  - kullanıcıların bilmesi gereken ve algılı olan butun kullanıcıların haber edilmesi gereken durumları bildirmek için Observer pattern kullanılır.

Observer (Behavioral) Design Pattern





https://sourcemaking.com/design_patterns/observer

Observer (Behavioral) Design Pattern

```java
public interface Subject {

    //methods to register and unregister observers
    public void register(Observer obj);
    public void unregister(Observer obj);

    //method to notify observers of change
    public void notifyObservers();

    //method to get updates from subject
    public Object getUpdate(Observer obj);
}
```

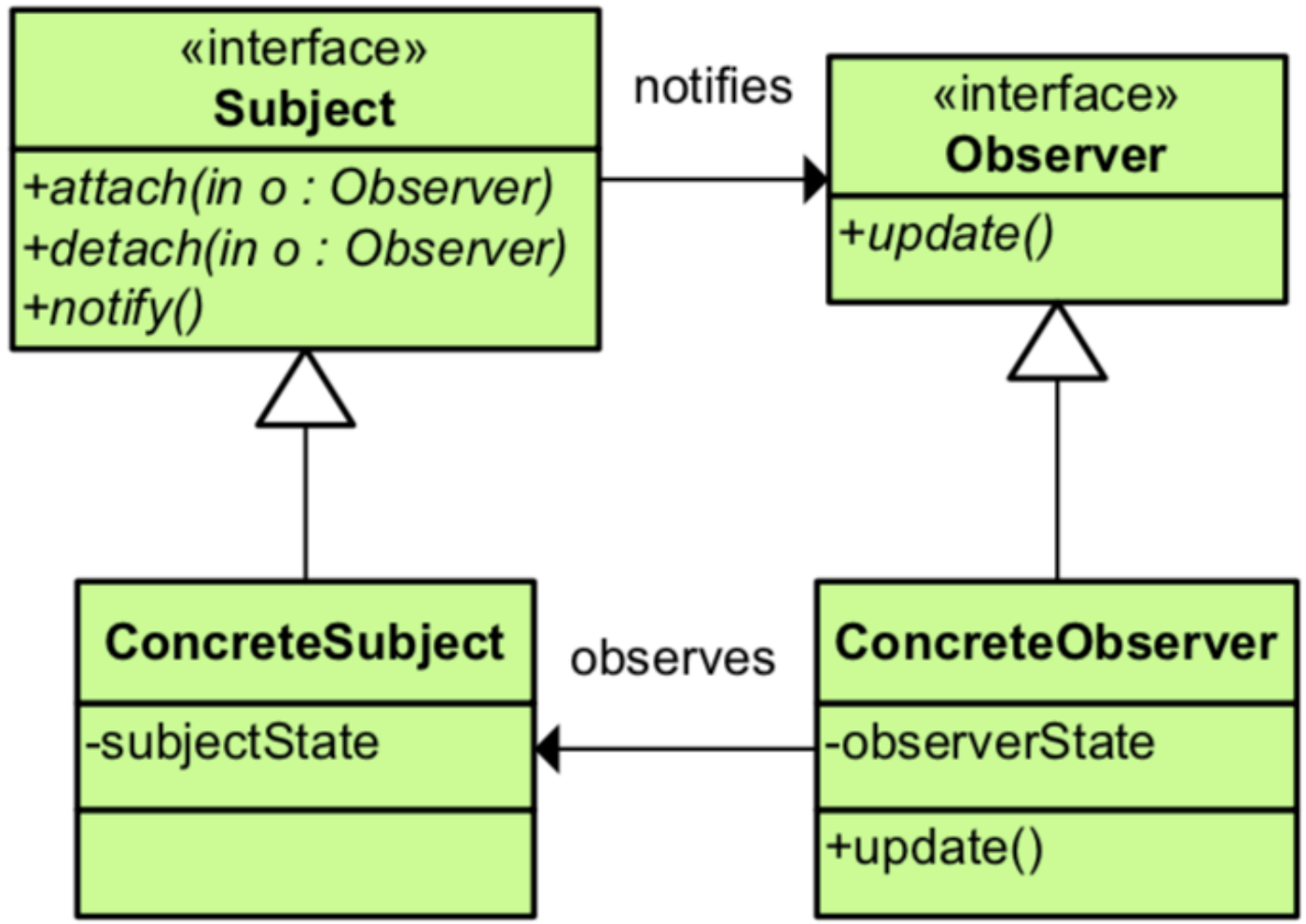
```java
public class MyTopic implements Subject {
        private List<Observer> observers;
        private String message;
        private boolean changed;
        private final Object MUTEX= new Object();

        public MyTopic(){
                this.observers=new ArrayList<>();
        }
        @Override
        public void register(Observer obj) {
                if(obj == null) throw new NullPointerException("Null Observer");
                synchronized (MUTEX) {
                if(!observers.contains(obj)) observers.add(obj);
                }
        }

        @Override
        public void unregister(Observer obj) {
                synchronized (MUTEX) {
                observers.remove(obj);
                }
        }
        @Override
        public void notifyObservers() {
                List<Observer> observersLocal = null;
                //synchronization is used to make sure any observer registered a
                synchronized (MUTEX) {
                        if (!changed)
                                return;
                        observersLocal = new ArrayList<>(this.observers);
                        this.changed=false;
                }
                for (Observer obj : observersLocal) {
                        obj.update();
                }
        }
        @Override
        public Object getUpdate(Observer obj) {...3 lines }
        //method to post message to the topic
        public void postMessage(String msg){...6 lines }
}
```

Observer (Behavioral) Design Pattern

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer (Behavioral) Design Pattern

```java
public interface Observer {
    //method to update the observer, used
    public void update();

    //attach with subject to observe
    public void setSubject(Subject sub);
}
```

```java
public class MyTopicSubscriber implements Observer {

    private String name;
    private Subject topic;

    public MyTopicSubscriber(String nm){
            this.name=nm;
    }
    @Override
    public void update() {
            String msg = (String) topic.getUpdate(this);
            if(msg == null){
                    System.out.println(name+":: No new message");
            }else
            System.out.println(name+":: Consuming message::"+msg);
    }

    @Override
    public void setSubject(Subject sub) {
            this.topic=sub;
    }

}
```