CREATIONAL DESIGN PATTERNS
Factory   Design Pattern
&
BEHAVIORAL DESIGN PATTERNS
Observer Design Pattern

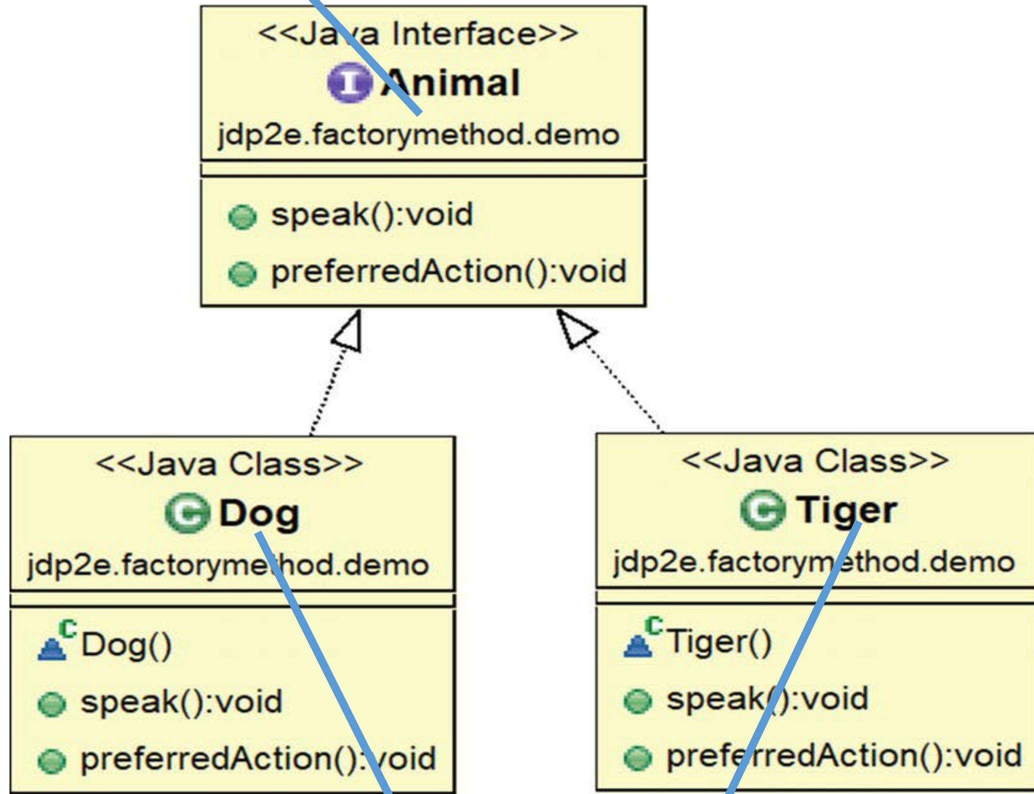
17.03.2025

Factory Method Pattern

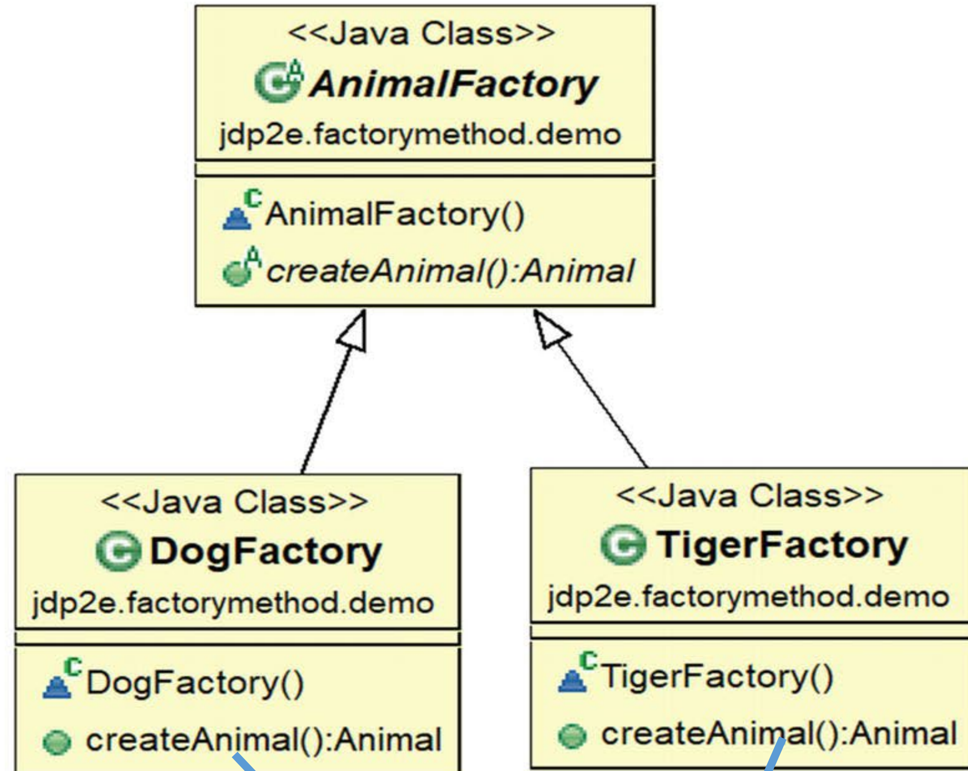
- Bir **interface** tanımlanır. Böylece bir nesne yaratmak mümkün olacaktır.
  - Alt sınıflar hangi sınıfın örnekleneceğine karar verir?
- Hangi alt sınıf örneklemeyi (instantiation) gerçekleştirecektir?
  - **Cevap**: Eğer Factory şablon kullanılıyorsa, bu şablon bir sınıfın alt sınıflara örneklemesini ertelemesine izin verir.
- Arayüz /interface) ve alt sınıfları örneğinde alt sınıflara ait bir yöntem bir **Tiger** veya **Dog** oluşturur, yani bir örneğini yaratır.
- Ama **Factory** şablonu ile: Bunun kararını **Dog** ya da **Tiger** sınıfı almayacak, bunun yerine hangi hayvanın yaratılacağı **DogFactory** veya **TigerFactory** sınıfı tarafından belirlenecektir.
  - Böyle bir örnekleme yapıldığı

  public abstract Animal createAnimal(); metodu ile bir factory (yaratma) rolü gerçekleştiği ile görülebilir.

Hierarchy-1



Hierarchy-2



Factory Method Pattern
Real-World Example

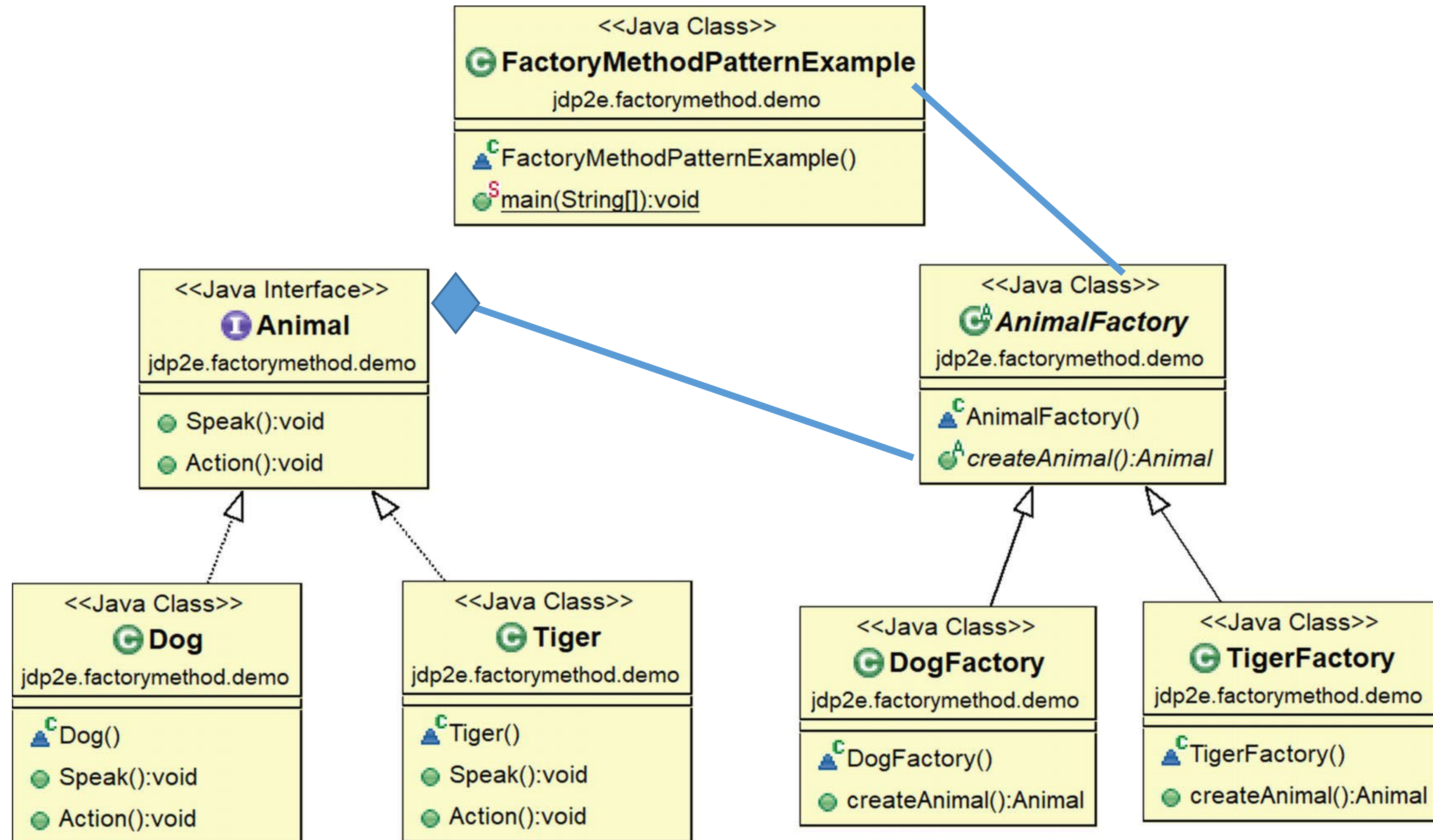
- A car manufacturing company produces different models of a car and runs its business well.
  - Based on the model of the car, different parts are manufactured and assembled.
- The company should be prepared for changes where customers can choose for better models in the near future.
- If the company needs to do a whole new setup for a new model, it can hugely impact its profit margin.
  - The new model demands only a few new features,
- The company should set up the factory in such a way that it can produce parts for the upcoming models also.

Factory Method Pattern
Computer-World Example

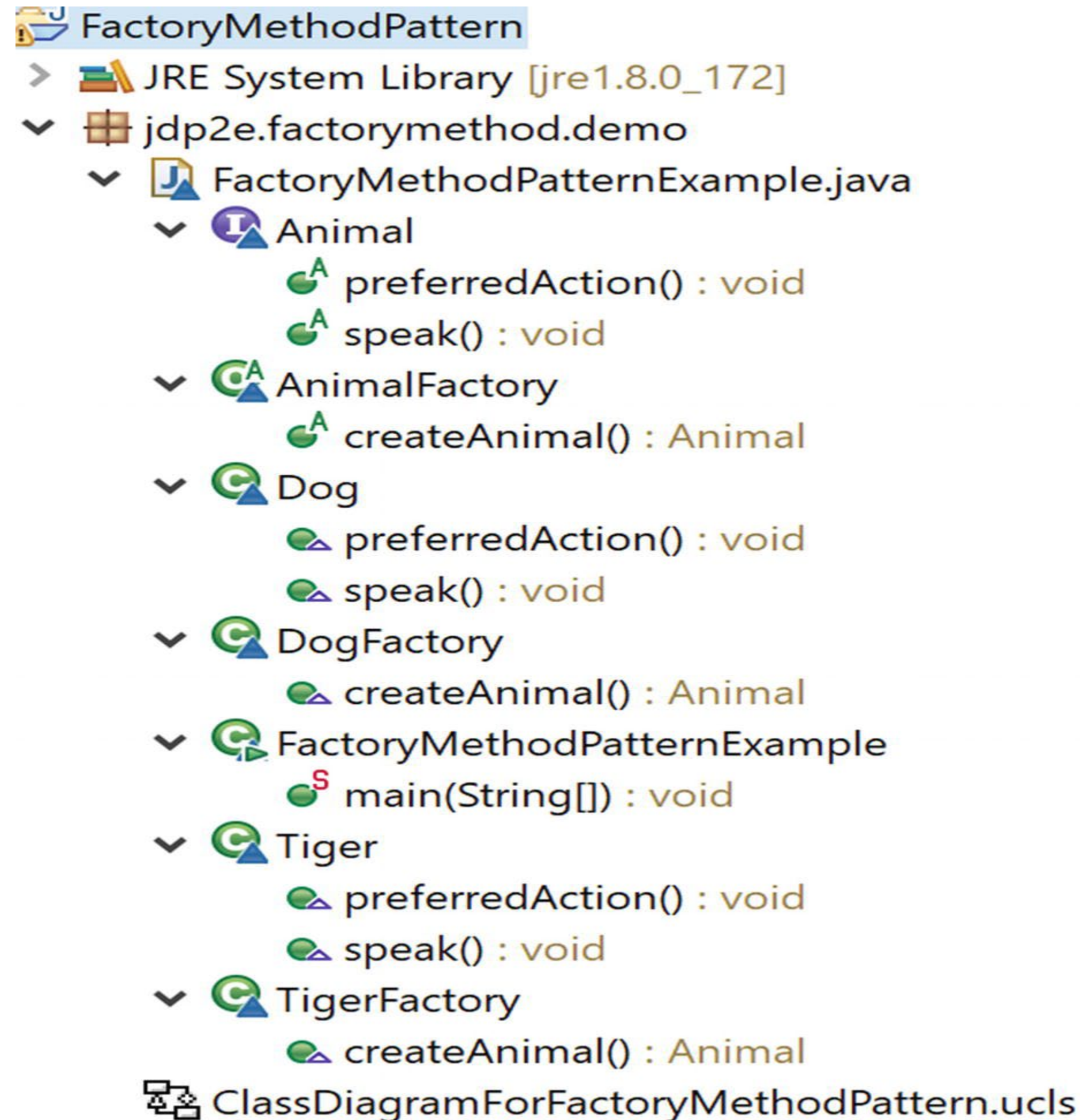
- We are building an application that needs to support two different databases, Oracle and SQL Server.
- Whenever we insert a data into a database, we create a SQL Server–specific connection (SqlServerConnection) or an Oracle server–specific connection (OracleConnection) and then we can proceed.
- If we put these codes into if-else (or switch) statements, we may need to repeat a lot of code.
- This kind of code is not easily maintainable
  - whenever we need to support a new type of connection, we need to reopen code and place the modifications.

SOLID principles !!!!

Factory Method Class Diagram



Factory Method Package Explorer View

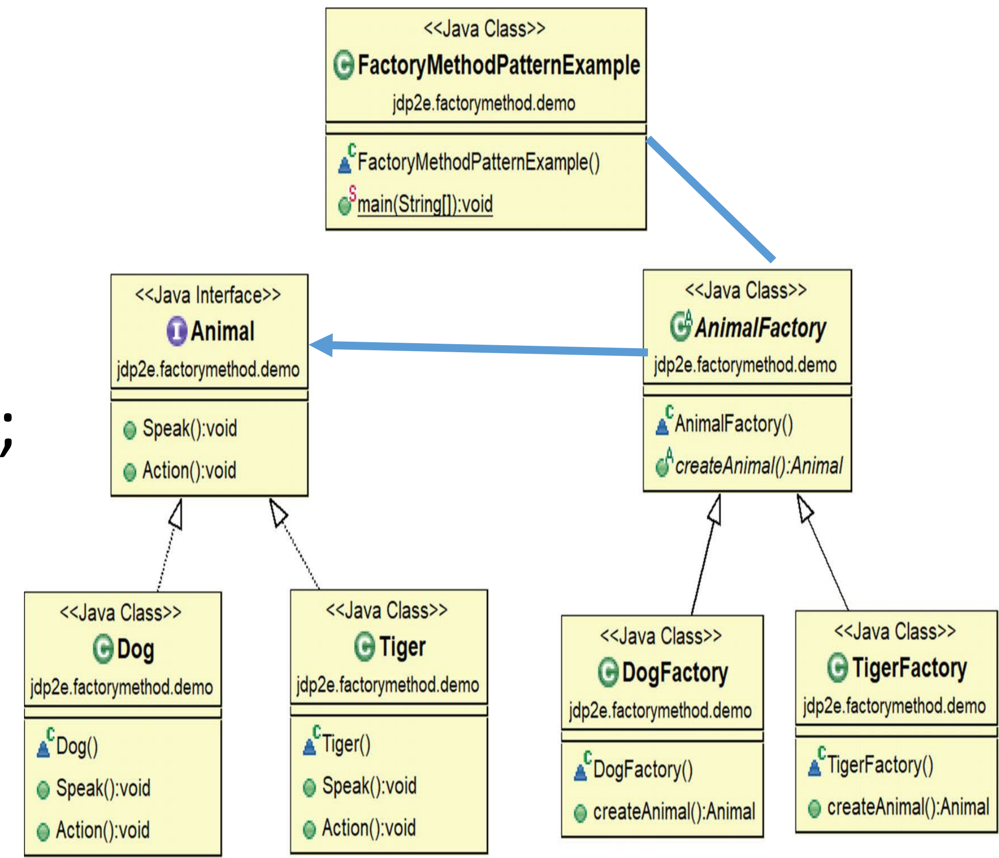


```
interface Animal   {
 void speak();
void preferredAction();
                                   }
class Dog implements Animal                              {
    public void speak()
  {
 System.out.println("Dog says: Bow-Wow.");
    }
public void preferredAction()
  {
      System.out.println("Dogs prefer barking...\n");      }
}
```

```java
class Tiger implements Animal   {
public void speak()
  {            System.out.println("Tiger says: Halum.");
  }
public void preferredAction()
  {       System.out.println("Tigers prefer hunting...\n");
  }
                                                         }
```

```
abstract class AnimalFactory {
    public abstract Animal    createAnimal();
                                    }


class DogFactory   extends   AnimalFactory       {
            public Animal createAnimal()   {
                 return new Dog();         }          }



class TigerFactory   extends  AnimalFactory  {
  public Animal createAnimal()      {
                return new Tiger();    }           }
```

```
class  FactoryMethodPatternExample {
    public static void main(String[] args) {
System.out.println("***Factory Pattern  Demo***\n");
       AnimalFactory   tigerFactory =new TigerFactory();
           Animal   aTiger =  tigerFactory.createAnimal();
                      aTiger.speak();
                      aTiger.preferredAction();

     AnimalFactory dogFactory = new DogFactory();
            Animal aDog = dogFactory.createAnimal();
                  aDog.speak();
                  aDog.preferredAction();
                                              }      }
```



Output

Factory Pattern Demo

Tiger says: Halum.

Tigers prefer hunting...

Dog says: Bow-Wow.

Dogs prefer barking...

Factory Method Pattern
Modified Implementation

- The AnimalFactory class is an abstract class.
  - It is advantage of using an abstract class.
- If we want a subclass to follow a rule that can be imposed from its parent (or base) class.
- Only AnimalFactory class is modified
  - we can introduce a new makeAnimal() method.

Modifying the AnimalFactory class

```
abstract class AnimalFactory{
        public Animal makeAnimal()   {
    System.out.println ("Here is inside makeAnimal() of AnimalFactory.");
              Animal animal =  createAnimal();
                    animal.speak();
                    animal.preferredAction();
        return animal;   }
```

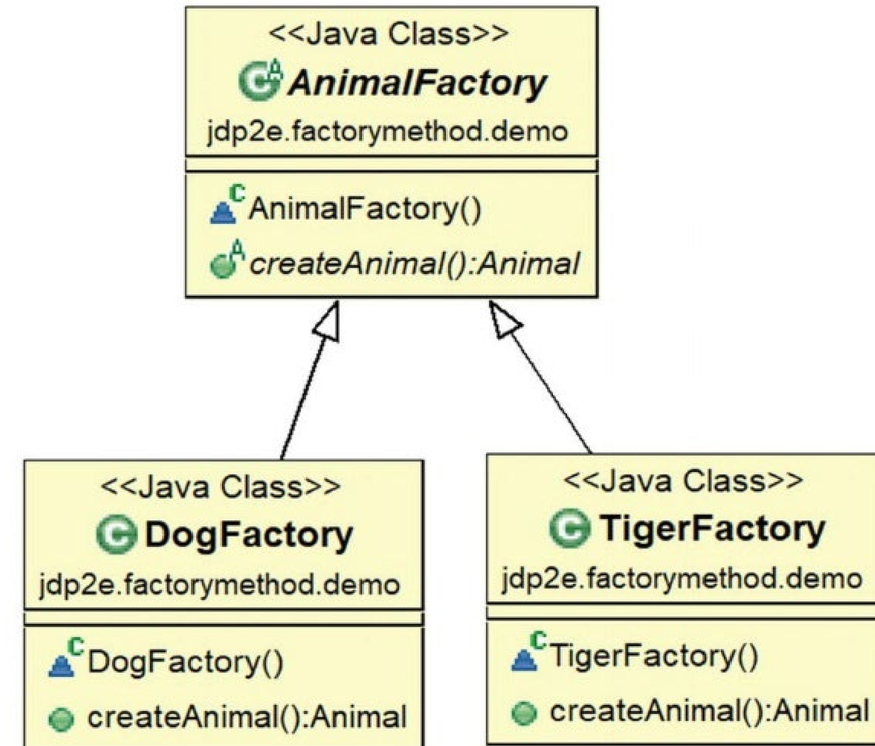
Modifying the AnimalFactory class (cont'd)

- Factory metodu alt sınıfların örneklerini oluşturmasını bekletebilir.

```
public abstract Animal createAnimal();
```

metodu **Tiger** ya da **Dog** oluşturacaktır(create).

- Fakat bu metodun implementasyonu ile hangisinin örnekleneceği bilinmemektedir.
- Bunun kararını **DogFactory** alt sınıfı ya da **TigerFactory** altsınıfı verecektir.



```
class ModifiedFactoryMethodPatternExample {  //Client code has adapted the changes
    public static void main(String[] args) {
     System.out.println("***Modified Factory Pattern Demo***\n");
AnimalFactory   tigerFactory =new TigerFactory();
   Animal aTiger = tigerFactory.makeAnimal();

AnimalFactory dogFactory = new DogFactory();
   Animal aDog = dogFactory.makeAnimal();
                                                      }    }
```

```
abstract class AnimalFactory{
  public Animal makeAnimal()   {
        System.out.println ("");
        Animal animal =  createAnimal();
                    animal.speak();
                     animal.preferredAction();
 return  animal;    }
```

```
***Modified Factory Pattern Demo***
      Here is  inside makeAnimal() of AnimalFactory.
Tiger says: Halum.
Tigers prefer hunting...
     Here is inside makeAnimal() of AnimalFactory.
Dog says: Bow-Wow.
Dogs prefer barking...
```

What are the advantages of using a factory?

- We separate code that can vary from the code that does not vary
- This technique helps you easily maintain code.
- The code is not tightly coupled;
  - We can add new classes like Lion, Beer, and so forth, at any time in the system *without modifying the existing architecture.*
- We have followed the “closed for modification but open for extension” principle.

What are the challenges of using a factory?

- ❑If we need to deal with a large number of classes, then we may encounter maintenance overhead.

Behavioral Design Patterns

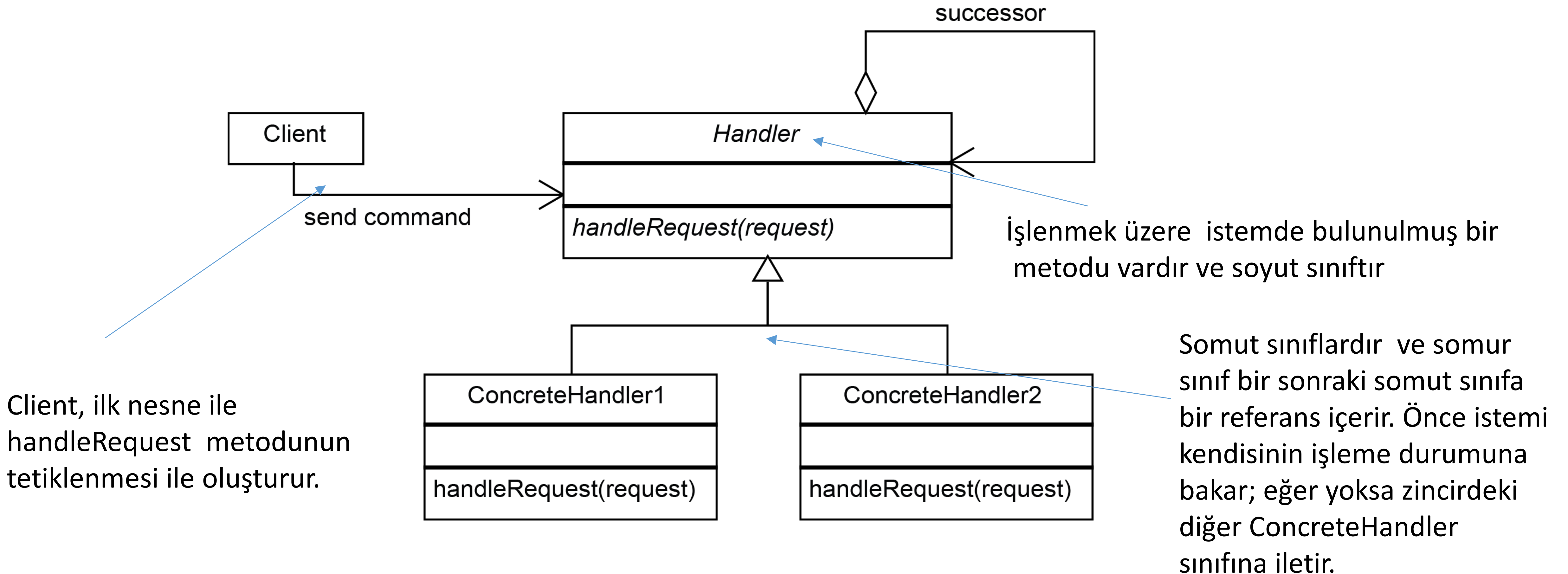
Observer Pattern

Chain of Responsibility - Sorumluluk Zinciri

- Client istemi ile bir veri seti üzerinde gerçekleşen farklı tipteki işlemler gerçekleşsin.
- Özetle işlemler sadece tek bir sınıftan değil de, farklı tipteki operasyonlar için farklı sınıflardan yapılsın.
- Sonuç «clean» ve «loosely coupled» koddur ve sınıflar işleyiciler (handlers) olarak adlandırılır.
- İlk işleyici (handler) bir eylemi gerçekleştireceği zaman bir «request» alacaktır veya onu ikinci işleyiciye (handler) iletecektir.
- Benzer şekilde bu da istemi bir sonraki işleyiciye zincir şeklinde iletecektir.

Chain of Responsibility - Sorumluluk Zinciri

Sorumluluk zinciri modeliyle işleyiciler bir istemi (request) işler ya da bunu gerçekleştiremezse diğerine iletebilecek şekilde zincirler.



```
protected Handler successor;
public void setSuccessor(Handler successor)
{
   this.successor = successor;
}
```

Her işleyici (handler) client tarafından kullanılan bir yöntemi gerçekleştirecektir.

Metot, ilgili istemin (request) işlenmesi mümkün değilse bir sonrakine geçileceğini belirtir.

Bu metot temel Handler sınıfında tanımlanmalıdır.

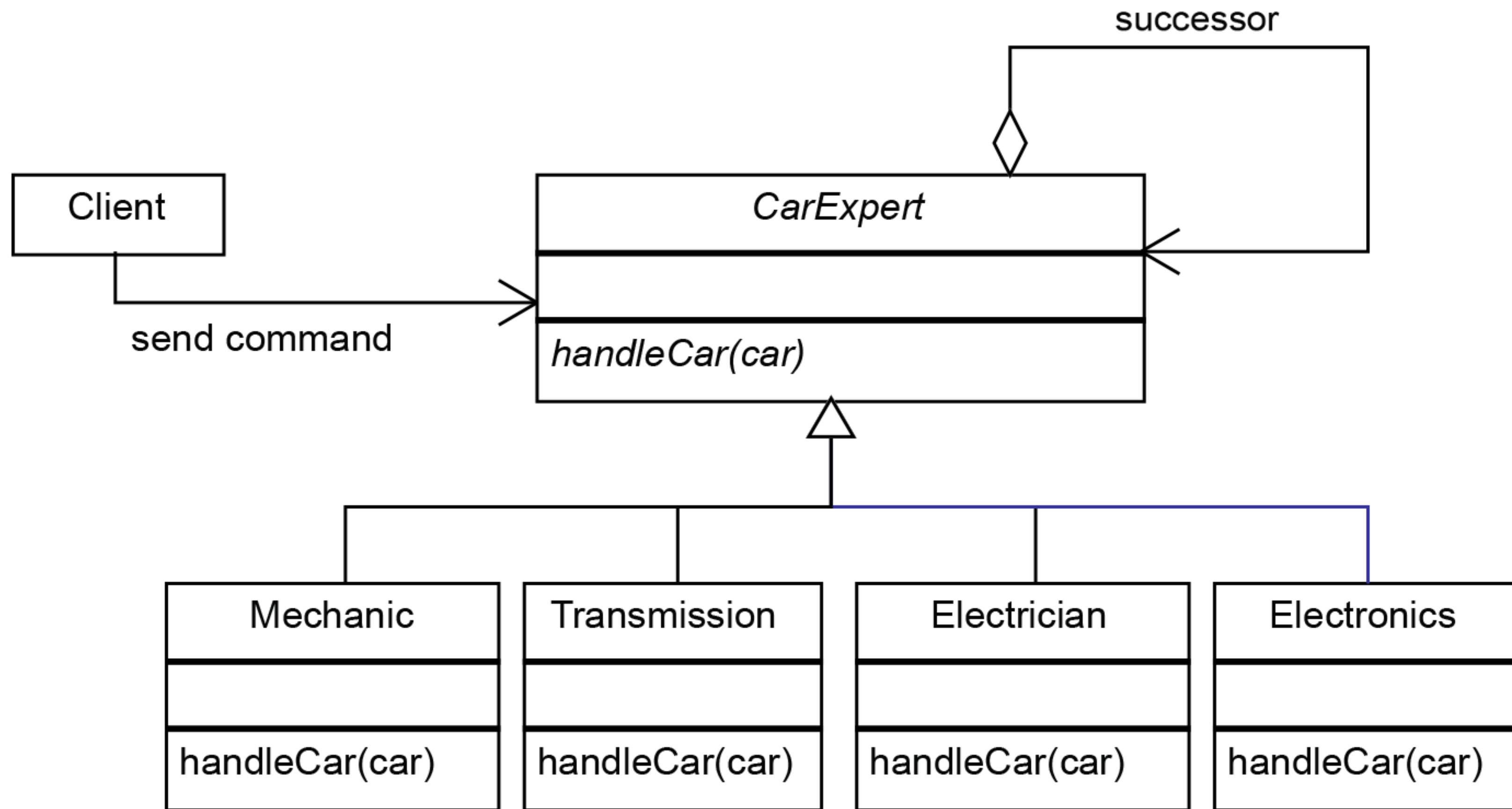
Her *ConcreteHandler* sınıfı aşağıdaki kodu içerir.

Bu kod istemin (request) işlenme durumunu kontrol eder, gerçekleşmezse diğerine geçer.

```
public void handleRequest(Request request)
{
    if (canHandle(request))
{    //code to handle the request   }
 else
    {
      successor.handleRequest();
     } }
```

Client, ilk zincirin tetiklenmesinden önce işleyicilerin (handlers) oluşturulmasından sorumludur.

Çağrı ilgili çağrıyı (request) işleyebilecek doğru işleyiciyi bulana kadar ilerleyecektir



Observer Şablon Örneği : Araba bakım servisinin işleyişi

«Chain of Responsibility» Tipleri

- Event Handlers (Olay işleyicileri): Örneğin, çoğu GUI çerçevesi olayları işlemek için sorumluluk zinciri modelini kullanır.

Örneğin bir pencere bazı düğmeleri içeren bir panelden oluşsun. Düğmenin olay işleyicisi (event handler) yazılır. İlerlemeye (isteği iletmeye) karar verilirse, zincirdeki bir sonraki panele geçilir. Eğer Panel bunu atlarsa, bir sonraki pencereye gidilecektir.

- Log Handlers(Günlük işleyicileri): Olay işleyicilerine benzer şekilde, her günlük işleyicisi durumuna göre belirli bir isteği kaydeder veya bunu bir sonraki işleyiciye iletir.

- Servlets: Java'da, istekleri veya yanıtları filtrelemek için javax.servlet.Filter kullanılır.

  https://docs.oracle.com/javaee/7/api/javax/servlet/FilterChain.html

  doFilter yöntemi ayrıca filtre zincirini bir parametre olarak alır ve isteği iletebilir.

Observer Design Pattern

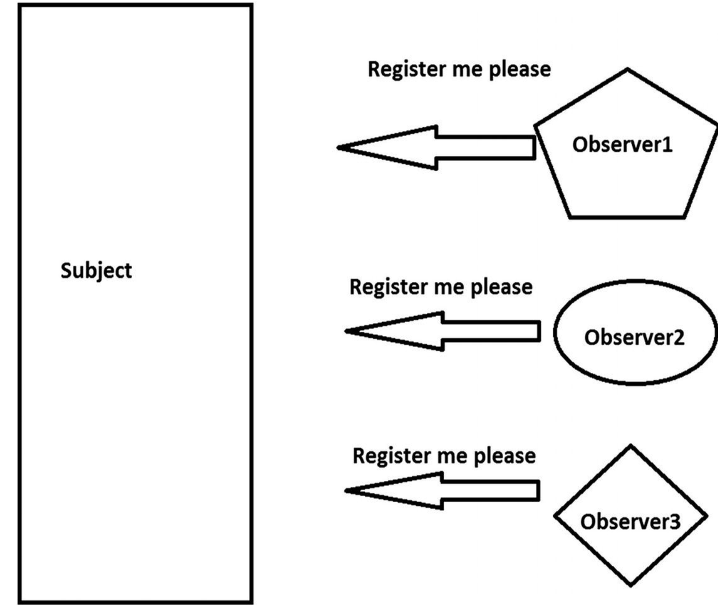
- ❑**Nesneler** arasında bire bir bağımlılık tanımlanır.
- ❑Nesnelerden birinin durumu değiştiğinde, buna bağlı olan tümü bilgilendirilir ve otomatik olarak güncellenir.

- ❑Herhangi bir subject (aynı zamanda object) pek çok gözlemci (observer object) tarafından gözlenir.
- ❑«Observers» bir bildirim almak üzere bir «subject» ile kayıtlıdır.
  - ❖«Subject» içerisinde bir değişiklik olduğunda,
  - ❖«Subject» ile ilgisini bitirdiğinde «subject» üzerinden kaydını siler.
  - ❖publish-subscribe pattern olarak ta adlandırılır.

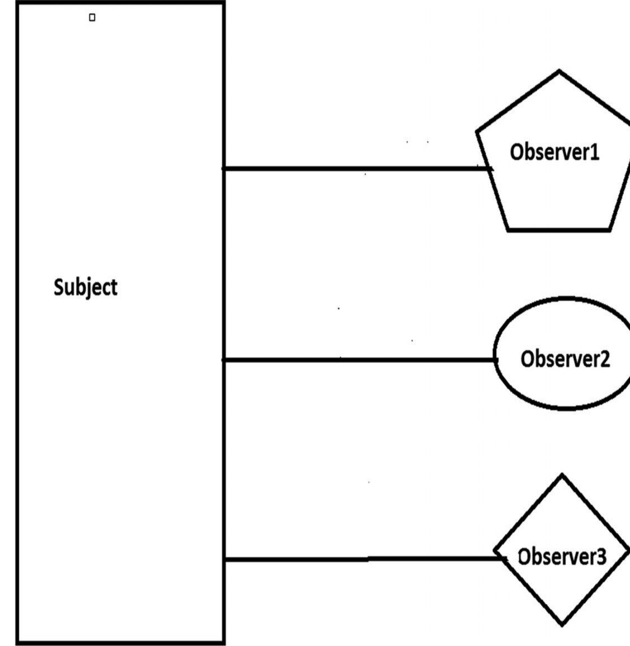
Özetle:

- ❑ Bir nesne (subject) aynı zamanda pek çok nesneye (set of objects) bildirim gönderir.

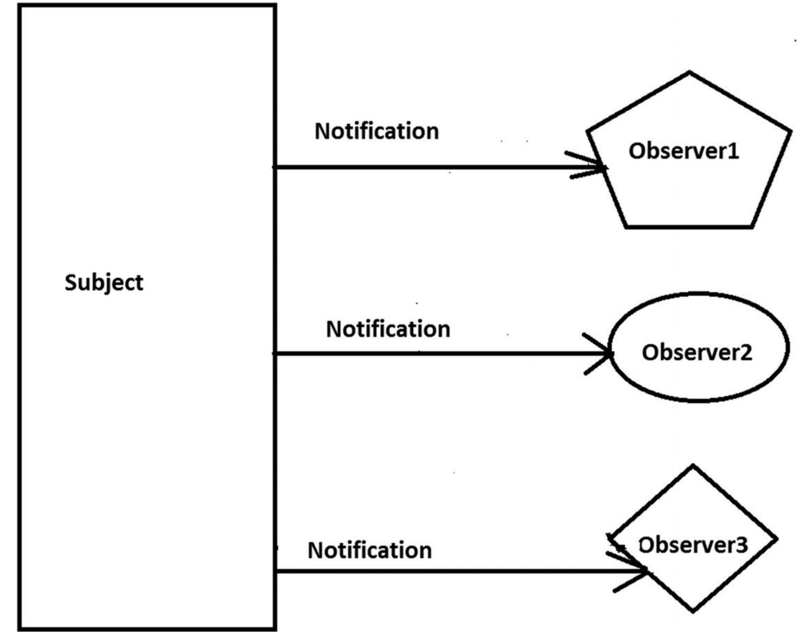
Senaryoların Görselleştirilmesi



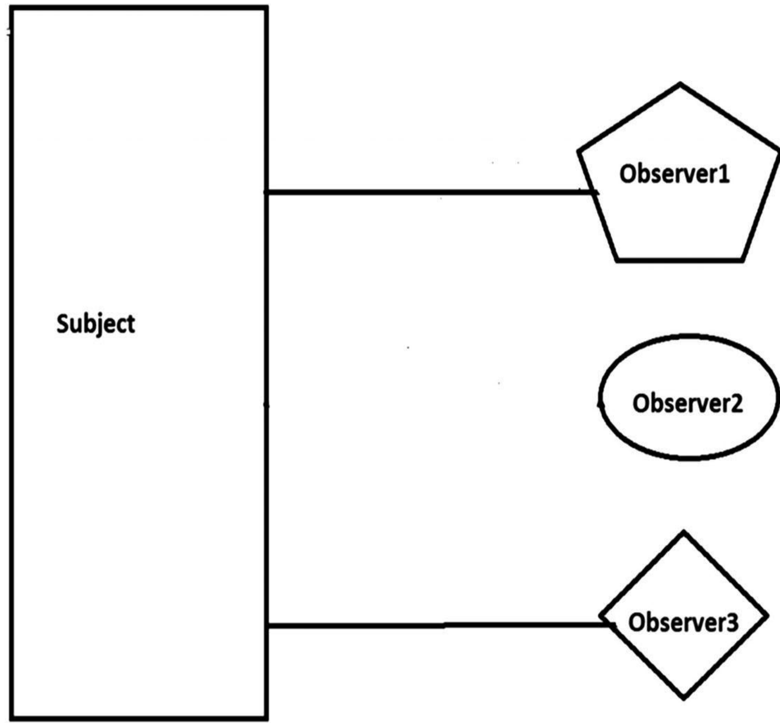
Adım 1. Bildirim gönderilmek üzere gözlemciler (Observers) talepleri izler

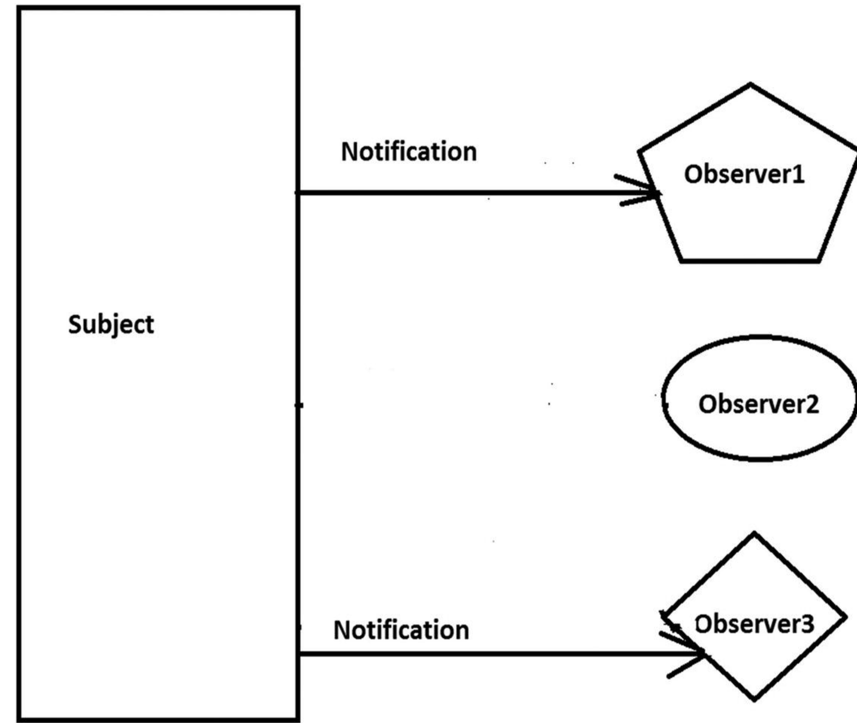


Adım 2. «Subject» olan nesne istemleri alarak bağlantıyı sağlar.



Adım 3. «subject» kayıtlı kullanıcılara bildirim gönderir.





Adım 4 (optional). Observer2 bildirim almaktan vazgeçer ve kaydını kaldırır.

Adım 5. Sadece Observer1 ve Observer3 «subject» ten bildirim almaya devam eder.

Real-World Example

- Think about a celebrity who has many followers on social media.
- Each of these followers wants all the latest updates from their favorite celebrity.
- So, they follow the celebrity until their interest wanes.
- When they lose interest, they simply do not follow that celebrity any longer.
- You can think each of these **followers** as an observer and the **celebrity** as a subject.

Computer-World Example

- UI-based
- UI is connected to a database.
- A user can execute a query through that UI
  - After searching the database, the result is returned in the UI.
- We segregate the UI from the database in such a way that:
  - if a change occurs in the database, the UI is notified, and it updates its display according to the change.

To simplify this scenario:

- Assume that we are the person responsible for maintaining a particular database in the organization.
- Whenever there is a change made inside the database, we want a notification so that we can take action if necessary.

Event –Driven Software

https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html

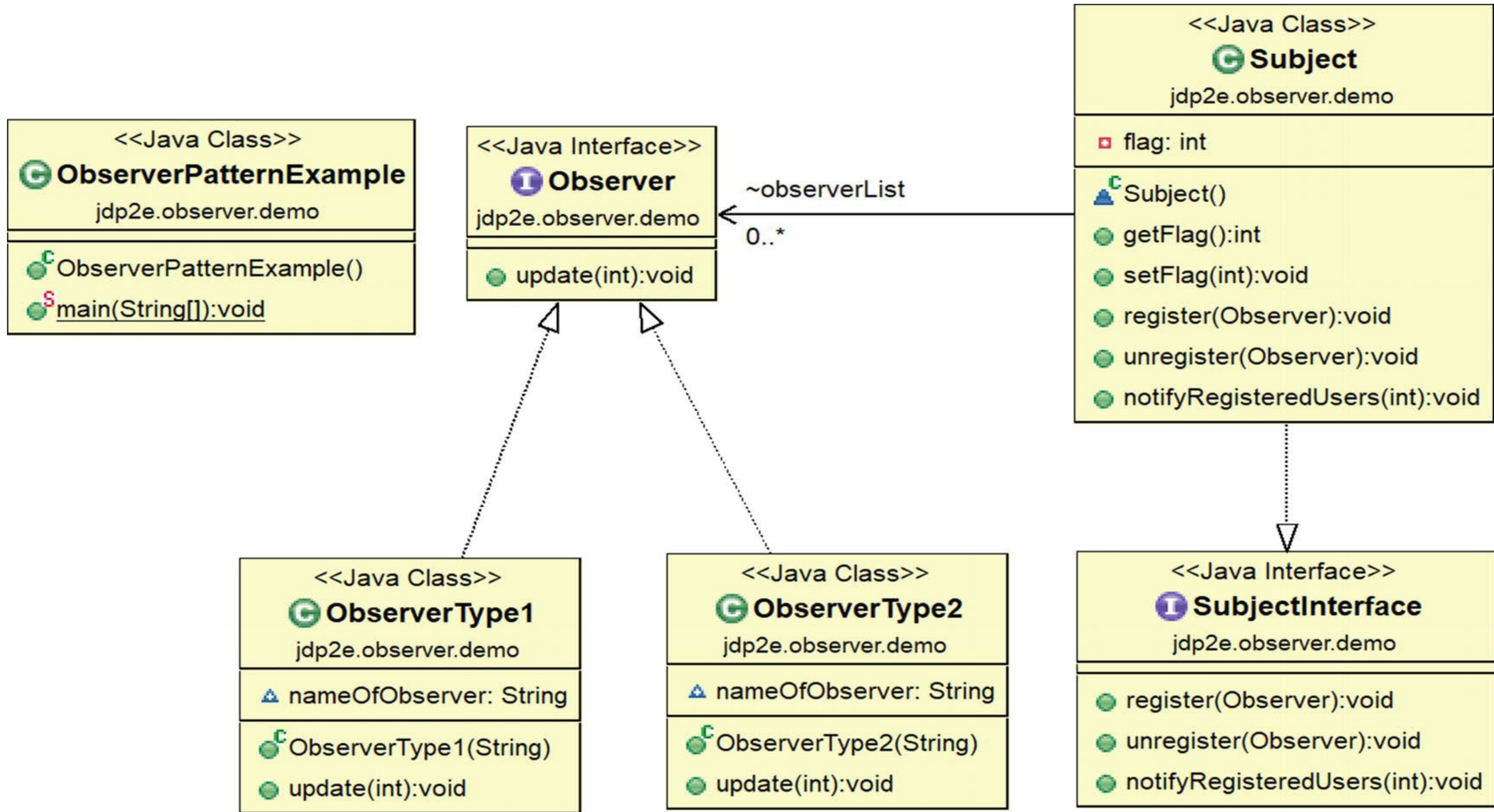
- This pattern is generally seen in **event-driven software.**
- C#, Java, and so forth have **built-in support** for handling events following this pattern.
- In Java, we can see the use of event listeners.
  - These listeners are **observers only.**
- In Java, we have a **ready-made class** called Observable
  - Observable class can have multiple observers.
  - These observers need to implement the Observer interface.
  - The Observer interface has an "update" method:

    void update(Observable o, Object arg).

  This method is invoked whenever a change occurs in the observed object.

  The application needs to call the Observable object's **notifyObservers** method to notify about the change to the observers.

  addObserver(Observer o) and deleteObserver(Observer o) methods add and delete an observer, similar to the register and unregister methods



<<Java Class>>
ObserverPatternExample
jdp2e.observer.demo
ObserverPatternExample()
main(String[]):void
<<Java Interface>>
Observer
jdp2e.observer.demo
update(int):void
~observerList
0..*
<<Java Class>>
Subject
jdp2e.observer.demo
flag: int
Subject()
getFlag():int
setFlag(int):void
register(Observer):void
unregister(Observer):void
notifyRegisteredUsers(int):void
<<Java Class>>
ObserverType1
jdp2e.observer.demo
nameOfObserver: String
ObserverType1(String)
update(int):void
<<Java Class>>
ObserverType2
jdp2e.observer.demo
nameOfObserver: String
ObserverType2(String)
update(int):void
<<Java Interface>>
SubjectInterface
jdp2e.observer.demo
register(Observer):void
unregister(Observer):void
notifyRegisteredUsers(int):void

```java
import java.util.*;

interface Observer  {

        void update (int  updatedValue);        }


class ObserverType1 implements Observer  {

     String   nameOfObserver;

   public ObserverType1(String name)

           {        this.nameOfObserver = name;         }

     @Override

    public void update(int  updatedValue)    {

   System.out.println( nameOfObserver+ "has received an alert : Updated myValue   in Subject  is:  "   +

                                                                                                          updatedValue);

                                                     }    }
```

```
class ObserverType2 implements Observer{

      String nameOfObserver;

        public ObserverType2(String name)

            {             this.nameOfObserver = name      }

          @Override

           public void update(int updatedValue)    {

           System.out.println( nameOfObserver+  "has received an alert : The current
value of   myValue in Subject is: "+ updatedValue);

                                                                    }    }
```

```java
interface SubjectInterface {
void register(Observer  anObserver);
void unregister(Observer anObserver);
void notifyRegisteredUsers (int
        notifiedValue);    }
```

```java
class Subject implements  SubjectInterface  {
    private int flag;
    public int getFlag()   {    return flag;     }
   public void setFlag(int flag)              {
            this.flag = flag;
//Flag value changed.Notify registered users/observers.
        notifyRegisteredUsers  (flag);        }
  List<Observer> observerList = new  ArrayList <Observer>();
@Override       public void register(Observer anObserver) {
                  observerList.add(anObserver);       }
 @Override      public void unregister(Observer anObserver) {
                  observerList.remove(anObserver);          }
@Override      public void notifyRegisteredUsers  (int updatedValue)   {
                          for (Observer observer :observerList)
                             observer.update(updatedValue);           } }
```

```java
public class ObserverPatternExample {
    public static void main(String[] args) {
        System.out.println (" ***Observer PatternDemo***\n");
        Observer myObserver1 = new ObserverType1("Roy");
        Observer myObserver2 = new ObserverType1("Kevin");
        Observer myObserver3 = new ObserverType2("Bose");
        Subject subject = new Subject();
        subject.register(myObserver1);
        subject.register(myObserver2);
        subject.register(myObserver3);
        System.out.println (" Setting Flag = 5 ");
        subject.setFlag(5);

        subject.unregister(myObserver1);
        //No notification this time Roy. Since it is unregistered.
        System.out.println ("\n Setting Flag = 50 ");
        subject.setFlag(50);
        subject.register(myObserver1);
        //Roy is registering himself again
        System.out.println( "\n Setting Flag = 100 ");
        subject.setFlag(100); }
}
```

Observer Pattern Demo

Setting Flag = 5

Roy has received an alert: Updated myValue in Subject is: 5

Kevin has received an alert: Updated myValue in Subject is: 5

Bose has received an alert: The current value of myValue in Subject is: 5

Setting Flag = 50

Kevin has received an alert: Updated myValue in Subject is: 50

Bose has received an alert: The current value of myValue in Subject is: 50

Setting Flag = 100

Kevin has received an alert: Updated myValue in Subject is: 100

Bose has received an alert: The current value of myValue in Subject is: 100

Roy has received an alert: Updated myValue in Subject is: 100

Örneğin Analizi

- Üç gözlemci (observer) , Roy, Kevin ve Bose subject nesnesinden bildirim almak üzere kayıt olsun (registered)
- Başlangıç aşamasında ikisi de bildirimleri alır.
- Aynı zamanda Roy bildirimlerle artık ilgili olmadığı için kaydını siler.
- Bundan sonra sadece iki nesne bildirim almaya devam eder.
- Bir süre sonra Roy fikrini değiştirerek yine «subject «nesnesinden bildirim almaya karar verir.
- Sonunda, üç kişi de «subject» ten bildirim almaktadır.

Soru – Cevap 1

Soru Sadece tek bir «observer» varsa, bir interface oluşturmaya gerek var mıdır?

Cevap Evet

Niçin?

Clean Kod bağlamında i arayüz/ soyut sınıf   pratiği önemsenir.

O nedenle de arayüzler (soyut sınıflar) somut sınıflardan daha fazla tercih edilir.

Aslında çoğunlukla çoklu gözlemcilerle işlemler yapılır; bu süreci de sistematik bir şekilde gerçekleştirmek, ortak bir dizilimin takibi önemlidir.

Soru - Cevap 2

Soru: Aynı uygulama için farklı tipteki gözlemciler mümkün müdür?

Cevap: Evet

Örneğin, bir firma bir yazılımın sürümünü ya da güncellemesini yaptığında, yazılımı satın müşterilerine ve şirket ortaklarına bildirimler gönderir.

Bu durumda şirket ortakları ve müşteriler farklı tipteki gözlemcilerdir.

Soru - Cevap 3

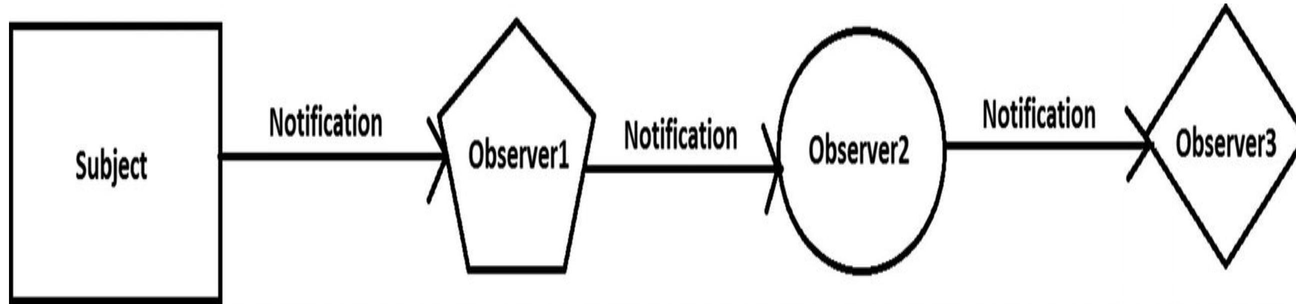
Soru: Gözlemciler «run time» sırasında eklenebilir mi ? Çıkarılabilir mi?

Cevap: Evet

Soru - Cevap 4

Soru «Observer pattern» ile «chain of responsibility pattern» arasında bir benzerlik var mıdır?

Cevap «observer pattern» örneklerinde tüm bildirimler aynı anda alınır. Fakat, chain of responsibility pattern örneklerinde , zincirdeki nesneler birer birer bildirim alırlar ve bu proses nesnenin bildirimin tümünü işleyene kadar devam eder.



Soru ve Cevap 5

Soru Bu model, «one-to- many» ilişkileri destekler. Doğru mudur?

Cevap Evet

Bir «subject» in çoklu gözlemcilere (observers) bildirim göndermesi, «one-to-many» ilişkisidir.

Soru - Cevap 6

Soru: Bu hazır şablon yapıları zaten mevcut. O zaman kod yazmaya gerek var mı?

Doğrudan bu hazır yapılar kullanılarak ta her zaman çözüm elde edilebilir mi?

Cevap: Hazır yapıları tercihe göre değiştirmek her zaman kolay değildir.

Çoğu durumda, (built-in) hazır fonksiyonellikleri değiştirmek mümkün olmaz.

Herhangi bir yapının implementasyonu sağlandığında, bu hazır yapıların (easy-made constructs) nasıl kullanılacağı daha iyi anlaşılır. .

Soru 6 ile ilgili bazı açıklamalar

1. Java da , Observable somut bir sınıftır. Bu sınıf bir arayüz implemente etmez.

Bu nedenle de Java nın yerleşik (built in) Observer API si ile çalışacak bir implementasyon oluşturmak mümkün değildir.

2. Java does not doğal olarak çoklu mirasa izin vermez.

«Observable» sınıfını genişletmek gerektiğinde, yeniden kullanıp potansiyeline limit getirebilir.

3. setChanged metodunun imzası (Observable sınıfında)

protected void setChanged() olarak verilir.

- Observable sınıfı alt sınıf olarak mevcut olmalıdır.
- Ama miras yerine «composition» ilişkisinin olması tasarım ilkelerinden biridir.

java.util.Observable altsınıflar oluşturmak üzere kullanılır ve programın diğer kısımşarı tarafından gözlemlenebilir.

Böyle bir altsınıfın değişimi gerektiğinde observable sınıflar bilgilendirilir (notified) .

java.util

Class Observable

java.lang.Object

java.util.Observable

public class Observable extends Object

Soru - Cevap 7

Soru. Bu şablonun yararları nelerdir?

Cevap .

1. Subject ve kayıtlı kullanıcılar (objects) loosely copule bir sistem oluşturur. Yani birbirlerini açık olarak bilmek zorunda değillerdir.

2. Bildirim listesinden bir «observer» çıkartıldığında ya da eklendiğinde «subject» üzerinde bir değişiklik gerekmez.

3.Bağımsız olarak herhangi bir zamanda «observer» ekleme ve çıkarma yapabilmek mümkündür.

Soru - Cevap 8

Soru: Observer Pattern ile ilgili karşılaşılabilecek sorunlar nelerdir?

1. Event-based bir sistemde «memory leak» en çok karşılaşılan sorundur.

«automatic garbage collector» kullanımı burada yardımcı olabilir. Çünkü yeniden kayıt ve kayıt silme işlemlerinin düzgün şekilde gerçekleşmediği düşünülebilir.

2.Bildirimlerin (notification) sırası bağlı değildir.

3.Java's built-in support bu şablon için bazı kısıtlar içerir.

Bunlardan biri «composition» yerine «inheritance» özelliğini kullanmaya zorlamasıdır.