**Template design mutlaka scada için kullanılmalı vtslere mesaj atmak için.State kullanılmalı**

**Behavioral Patterns - Mediator Design Pattern**

* Genelde GUI’ler için kullanılır.Örnek bir chat uygulaması, yada bir havalimanı kulesi. Uçakların hiçbiri birbirleri ile konuşmazlar kule ile konuşurlar. Kule karar verir
* Mediator Design Pattern nesnelerin aralarındaki iletişimin tek bir noktadan sağlanması ve koordine edilmesi gerektiği durumlarda kullanılır.Nesneler birbirleri ile doğrudan konuşmak yerine merkezi bir yapı aracılığı ile haberleşirler bu sayede nesneler arasında bağımlılık azalır. Nesneler birbirlerinin kim olduklarını bilmeden merkez aracılığı ile haberleşebilirler.

**interface** Mediator{  
 *//Mediator interface'i* **void** notify(Component sender,String message);  
 **void** register(Component component);  
}

Tüm component’ler için ortak olacak abstract Component class’ı

**abstract class** Component{  
 **private** String **name**;  
 **protected** Mediator **mediator**;  
 **public** Component(String name,Mediator mediator) {  
 **this**.**name** = name;  
 **this**.**mediator** = mediator;  
 }  
 **public abstract void** send();  
 **public abstract void** receive(String message);  
 **public** String getName(){  
 **return this**.**name**;  
 }  
}

Yukarıda gördüğünüz üzere Component’in ismi ve mediator bilgisi yer alacak

Component Class’ları

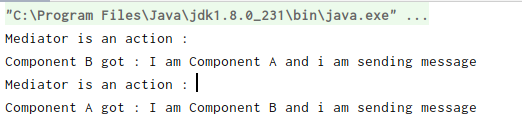
**class** ComponentA **extends** Component{  
 **public** ComponentA(Mediator mediator){  
 **super**(**"Component-A"**, mediator);  
 }  
 @Override  
 **public void** send() {  
 *//Component B'ye mesaj* String message = **"I am Component A and i am sending message"**;  
 *//notify'i yapacak olan mediator yani arabulucu* **this**.**mediator**.notify(**this**,message);  
 }  
 @Override  
 **public void** receive(String message) {  
 *//Component A mesaj alıyor* System.***out***.println(**"Component A got : "** + message);  
 }  
}  
**class** ComponentB **extends** Component{  
 **public** ComponentB(Mediator mediator){  
 **super**(**"Component-B"**,mediator);  
 }  
 @Override  
 **public void** send() {  
 *//Component A'ya mesaj* String message = **"I am Component B and i am sending message"**;  
 *//notify'i yapacak olan mediator yani arabulucu* **this**.**mediator**.notify(**this**,message);  
 }  
 @Override  
 **public void** receive(String message) {  
 *//Component B mesaj alıyor* System.***out***.println(**"Component B got : "** + message);  
 }  
}

Mediator Interface’inin concrete nesnesi

**class** ConcreteMediator **implements** Mediator{  
 **private final** String **COMPONENT\_A** = **"Component-A"**;  
 **private final** String **COMPONENT\_B** = **"Component-B"**;  
 **private** Map<String,Component> **registerComponents** = **new** HashMap<>();  
 **public void** notify(Component sender, String message) {  
 String senderName = sender.getName();  
 **if** (**COMPONENT\_A**.equals(senderName)){  
 reactOnA(message);  
 }**else if** (**COMPONENT\_B**.equals(senderName)){  
 reactOnB(message);  
 }  
 }  
 *//Map içerisine eğer yoksa component'leri ekliyorum* **public void** register(Component component) {  
 **this**.**registerComponents**.put(component.getName(),component);  
 }  
 **private void** reactOnA(String message){  
 System.***out***.println(**"Mediator is an action : "**);  
 **registerComponents**.get(**COMPONENT\_B**).receive(message);  
 }  
 **private void** reactOnB(String message){  
 System.***out***.println(**"Mediator is an action : "**);  
 **registerComponents**.get(**COMPONENT\_A**).receive(message);  
 }  
}

Main

**public static void** main(String[] args) {  
 Mediator mediator = **new** ConcreteMediator();  
 Component compA = **new** ComponentA(mediator);  
 Component compB = **new** ComponentB(mediator);  
 mediator.register(compA);  
 mediator.register(compB);  
 compA.send();  
 compB.send();  
}

****

**Behavioral - Strategy Design Pattern -** Bir işlemi gerçekleştirmek için birden fazla yöntem(algoritma) mevcut olabilir bunları if else bloklarıyla çözmek yerine strategy design pattern kullanılır. Duruma göre bir yöntem seçip, uygulamak için strategy tasarım şablonu kullanılır.**String tipinde bir değişkenimiz olduğunu düşünelim. String tipindeki değişkenimizi XML yada JSON şeklinde serialize ve deserialize yapmamız gereken bir senaryomuz olsun.Oluşturulması gerekenler;**

1. BaseSerialize adlı bir interface tanımlanır ve içerisine void tipinde geri dönüşü olan ve içerisine String tipinde değer alan Serialize ve Deserialize adlı 2 method imzasını koyalım

interface SerializeStrategy{

void Serialize(string value);

void Deserialize (string value);

}

1. Context adlı bir class create edelim ve BaseSerialize interface’ini private olarak enjekte edelim. ExecuteSerialize-ExecuteDeserialize adlı Context sınıfına özel 2 adet method tanımlıyorum. Bu methodlar kullanılarak SerializeStrategy interface’i encapsule edilmiş olacak. Serializer işlemini yapacak olan asıl methodlarım Context sınıfı ile çalışacaklar

class Context{

private SerializeStrategy \_serializeStrategy;

public Context(SerializeStrategy serializeStrategy)

{

\_serializeStrategy = serializeStrategy;

}

public void executeSerialize(string value){

\_serializeStrategy.Serialize(value);

}

public void executeDeserialize(string value){

\_serializeStrategy.Deserialize(value);

}

}

1. **Bu algoritmada XmlSerializer yapması için ilgili class’ı ekliyorum**

class XmlSerializer : SerializeStrategy

{

public void Deserialize(string value)

{

Console.WriteLine("Value deserialized : {0}",value);

}

public void Serialize(string value)

{

Console.WriteLine("Value serialized : {0}",value);

}

}

1. **Artık main method içerisinden Context’i call edip String value’muzu Serialize - Deserialize edebiliriz**

void Main()

{

Context myContext = new Context(new XmlSerializer());

myContext.executeSerialize("Hello from Serialize");

myContext.executeDeserialize("Hello from Deserialize");

}

Not : Dikkat edildiği üzere tek interface üzerinden istediğimiz kadar algoritmik işlem yaptırabiliriz.JSON Serializer ihtiyacımız olduğunda sınıfı yazıp algoritmasını yazmamız yeterli olacaktır.Strategy pattern buralar da kullanılır.Bir diğer örnek ise bir text’in tamamen büyütülmesi tamamen küçültülmesi gibi iki farklı işlemimiz olduğunu düşünelim.Burada da strategy design pattern kullanılabilir

**Creational - Prototype Design Pattern - Tüm Örnekleri incele** Ağır bir database nesnesini çağırıp bir kopyasını deep copy olarak çıkardıktan sonra üzerinde çalışmak maliyeti düşürecek ve performans sağlayacaktır. Prototype design pattern bu problemi çözmek için vardır,initialize masraflarını minimize eder.Bir diğer örnekte excelde aylık bir rapor hazırlandığını düşünelim, her ay raporu baştanmı yaratıyorsunuz yoksa diğer ay kullandığınız excel i copy paste edip verilerimi değiştiriyorsunuz.

**Örnek 1;**

Cloneable interface’inden türetilen bir abstract class create ediyoruz,içerisinde colorName diye bir property ve fillColor adında bir method barındırıyor.Clone methodu ise gelen objenin clone’unu oluşturuyor;

**abstract class** ColorPrototype **implements** Cloneable{  
 **protected** String **colorName**;  
 **abstract void** fillColor();  
  
 @Override  
 **protected** Object clone() {  
 Object clone = **null**;  
 **try**{  
 clone = **super**.clone();  
 }**catch** (CloneNotSupportedException e){  
 e.printStackTrace();  
 }  
 **return** clone;  
 }  
}

YellowColor adında bir class ColorPrototype’ımızı inherit ediyor,aynı şekilde değişik renklerde eklenebilir;

**class** YellowColor **extends** ColorPrototype{  
 **public** YellowColor() {  
 **this**.**colorName** = **"Yellow"**;  
 }  
 @Override  
 **void** fillColor() {  
 System.***out***.println(**"filling yellow color"**);  
 }  
}

Prototype class’ımızı yaratacak bir factory create ediyoruz.Bu class maliyeti yüksek olan class’larımızı newleme görevini üstlenen class’ımız;

**class** ColorPrototypeFactory{  
 **private static** HashMap<String,ColorPrototype> *colorMap* = **new** HashMap<>();  
 **static**{  
 *colorMap*.put(**"Yellow"**,**new** YellowColor());  
 *colorMap*.put(**"Red"**,**new** RedColor());  
 }  
 **public static** ColorPrototype getColor(String colorName){  
 **return** *colorMap*.get(colorName);  
 }  
}

Main içerisinde yellow’u create ettiğimiz anda Red clone olarak hiç uğraşılmadan direk create edilmiş olacak

**class** main{  
 **public static void** main(String[] args) {  
 ColorPrototypeFactory.*getColor*(**"Yellow"**).fillColor();  
 ColorPrototypeFactory.*getColor*(**"Red"**).fillColor();  
 }  
}

**Örnek 2;**

**abstract class** Prototype **implements** Cloneable{  
 @Override  
 **protected** Object clone() **throws** CloneNotSupportedException {  
 Object clone = **null**;  
 **try**{  
 clone = **super**.clone();  
 }**catch** (CloneNotSupportedException exception){  
 exception.printStackTrace();  
 }  
 **return** clone;  
 }  
}

**Class ;**

**class** Movie **extends** Prototype{  
 **private** String **name** = **null**;  
 @Override  
 **protected** Object clone() **throws** CloneNotSupportedException {  
 System.***out***.println(**"Cloning Movie ..."**);  
 **return super**.clone();  
 }  
}

**class** Album **extends** Prototype{  
 **private** String **name**;  
 @Override  
 **protected** Object clone() **throws** CloneNotSupportedException {  
 System.***out***.println(**"Cloning Album ..."**);  
 **return super**.clone();  
 }  
}

**Factory;**

**class** PrototypeFactory{  
 **public static class** ModelType{  
 **public static final** String ***MOVIE*** = **"Movie"**;  
 **public static final** String ***ALBUM*** = **"Album"**;  
 }  
 **private static** HashMap<String,Prototype> *type* = **new** HashMap<>();  
 **static**{  
 *type*.put(ModelType.***MOVIE***,**new** Movie());  
 *type*.put(ModelType.***ALBUM***,**new** Album());  
 }  
  
 **public static** Prototype getInstance(**final** String s) **throws** CloneNotSupportedException {  
 **return** (Prototype) *type*.get(s).clone();  
 }  
}

**Main;**

**class** Test{  
 **public static void** main(String[] args) **throws** CloneNotSupportedException {  
 String test = PrototypeFactory.*getInstance*(**"Movie"**).toString();  
 String test2 = PrototypeFactory.*getInstance*(**"Album"**).toString();  
 System.***out***.println(test);  
 System.***out***.println(test2);  
 }  
}

**Örnek 3;**

**interface** Animal **extends** Cloneable{  
 **public** Animal clone();  
}

**Concrete Class**

**class** Sheep **implements** Animal{  
 **public** Sheep(){  
 System.***out***.println(**"Sheep is made"**);  
 }  
 @Override  
 **public** Animal clone() {  
 Sheep object = **null**;  
 **try**{  
 object = (Sheep) **super**.clone();  
 } **catch** (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 **return** object;  
 }  
}

**Factory**

**class** CloneFactory{  
 **private static** HashMap<String,Animal> *animalHashMap* = **new** HashMap<>();  
 **static**{  
 *animalHashMap*.put(**"Sheep"**,**new** Sheep());  
 }  
 **public static** Animal getInstance(**final** String s){  
 **return** *animalHashMap*.get(s);  
 }  
}

**Main**

**class** main{  
 **public static void** main(String[] args) {  
 Sheep test = (Sheep) CloneFactory.*getInstance*(**"Sheep"**);  
 System.***out***.println(test.getClass());  
 }  
}

**Behavioral - Memento Design Pattern**

Memento Design Pattern, elimizdeki mevcut nesnenin herhangi bir T anındaki durumunu kayda alarak, sonradan oluşabilecek değişiklikler üzerine tekrardan o kaydı elde etmemizi sağlayan bir desendir. Burada mevcut nesnenin özel bir halinden bahsetmemiz mümkündür. O hal ilgili tasarım kalıbı sayesinde sonradan da elde edilebilecektir. Memento Class - Sadece POJO class’ıdır ve içerisinde sadece state’i getter ile tutar.Originator Class - İçerisinde private şeklinde String state tutar ve state’i set(içerisinde setter bulunur) eder.save methodu statei memento olarak kaydeder.restore methodu da undo işlemi için kullanılır CareTaker - Birden fazla memento noktasını takip etmek için kullanılır.Save pointleri saklar.

*//Memento***class** Memento{  
 **private** String **state**;  
 **public** Memento(String state) {  
 **this**.**state** = state;  
 }  
 **public** String getState() {  
 **return state**;  
 }  
}

*//CareTaker***class** CareTaker{  
 **private** ArrayList<Memento> **mementoList** = **new** ArrayList<>();  
 **public void** addMementoToMementoList(Memento memento){  
 **mementoList**.add(memento);  
 }  
 **public** Memento getMemento(**int** index){  
 **return mementoList**.get(index);  
 }  
 **public int** getCount(){  
 **return mementoList**.size();  
 }  
}

*//Origin***class** Origin{  
 **private** String **state**;  
 **public void** setState(String state) {  
 System.***out***.println(**"Originator set to state : "** + state);  
 **this**.**state** = state;  
 }  
 **public** Memento save(){  
 System.***out***.println(**"Originator saving to memento"**);  
 **return new** Memento(**state**);  
 }  
 **public void** restore(Memento memento){  
 **this**.**state** = memento.getState();  
 System.***out***.println(**"Originator restoring to memento : "**+ **state**);  
 }  
}

**class** Test{  
 **public static void** main(String[] args) {  
 CareTaker careTaker = **new** CareTaker();  
 Origin origin = **new** Origin();  
 origin.setState(**"State 1"**);  
 origin.setState(**"State 2"**);  
 careTaker.addMementoToMementoList(origin.save());  
 System.***out***.println(careTaker.getCount());  
 origin.setState(**"State 3"**);  
 careTaker.addMementoToMementoList(origin.save());  
 System.***out***.println(careTaker.getCount());  
 origin.restore(careTaker.getMemento(1));  
 }  
}

**Structural - Flyweight Design Pattern**

Aynı datanın değişik objeler tarafından kullanılması için bir havuz yaratır ve RAM’i çok efektif kullanılması için tasarlanmıştır.Bellek kullanımı ile ilgili bir sıkıntı yoksa bu pattern den uzak durulması tavsiye edilir.Bir örnekte bir web sitesinde image’leri bir havuzda tutmak için kullanılabilir. 3 State’den oluşur (Extensic state - Instrinsic state - Factory);

* Extensic state yani her instance’da farklılık gösterebilecek alanlar ayrılmalıdır.
* Ayrılmış olan alanlar ilgili metoda parametre olarak gönderilmelidir.
* Nesne oluşturmak için bir factory class oluşturulmalıdır.

Ilk olarak interface create ediliyor;

**interface** Vehicle{  
 **void** assingColor(String color);  
 **void** startEngine();  
}

Interface’imizi inherit edecek olan class.

* Extensic State her instance’de farklılık gösterecek alanlar. (not shareable)
* Intrinsic State her instance’da aynı olacak alanlar (shareable object)

Birinci Aracımız;

**class** Truck **implements** Vehicle{  
 **private final** String **MAXSPEED**; *//intrinsic* **private** String **color**; *//Extensic* **public** Truck(){  
 **MAXSPEED** = **"120km/s"**;  
 }  
 @Override  
 **public void** assingColor(String color) {  
 **this**.**color** = color;  
 }  
 @Override  
 **public void** startEngine() {  
 System.***out***.println(**color**+ **" colored Truck with max speed : "** + **MAXSPEED**);  
 }  
}

Ikinci Aracımız;

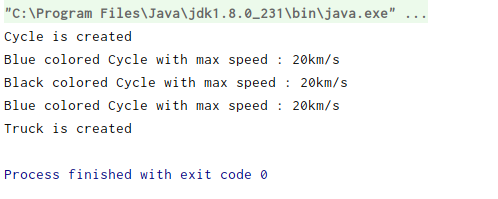
**class** Cycle **implements** Vehicle{  
 **private final** String **MAXSPEED**;  
 **private** String **color**;  
 **public** Cycle() {  
 **MAXSPEED** = **"20km/s"**;  
 }  
 @Override  
 **public void** assingColor(String color) {  
 **this**.**color**=color;  
 }  
 @Override  
 **public void** startEngine() {  
 System.***out***.println(**color**+ **" colored Cycle with max speed : "** + **MAXSPEED**);  
 }  
}

Factory ;

**class** VehicleFactory{  
 *//type of vehicle (truck,cycle or taxi)* **private static** HashMap<String,Vehicle> *typeOfVehicleCollection* = **new** HashMap<>();  
 *//vehicle already exist?* **public static** Vehicle getVehicle(String vehicleType){  
 Vehicle vehicle = **null**;  
 **if** (VehicleFactory.*typeOfVehicleCollection*.containsKey(vehicleType)){  
 vehicle = VehicleFactory.*typeOfVehicleCollection*.get(vehicleType);  
 }**else**{  
 **switch**(vehicleType){  
 **case "Cycle"** :  
 System.***out***.println(**"Cycle is created"**);  
 vehicle = **new** Cycle();  
 **break**;  
 **case "Truck"** :  
 System.***out***.println(**"Truck is created"**);  
 vehicle = **new** Truck();  
 **break**;  
 **default**:**throw new** IllegalArgumentException(**"Vehicle type not exist"**);  
 }  
 VehicleFactory.*typeOfVehicleCollection*.put(vehicleType,vehicle);  
 }  
 **return** vehicle;  
 }  
}

HashMap içerisinde instance’i create edilmiş olan Vehicle interface’inden inherit edilmiş concrete class’ların listesini tutuyoruz. getVehilce methodumuz vehicle’ın concrete class’larının instance’larının create edilip edilmediğini kontrol ederek eğer instance’ları create edilmemişse create ediyor ve HashMap’imize ekliyor

**class** Test{  
 **public static void** main(String[] args) {  
 Vehicle cycle = VehicleFactory.*getVehicle*(**"Cycle"**);  
 cycle.assingColor(**"Blue"**);  
 cycle.startEngine();  
 cycle.assingColor(**"Black"**);  
 cycle.startEngine();  
 Vehicle cycleTwo = VehicleFactory.*getVehicle*(**"Cycle"**);  
 cycleTwo.assingColor(**"Blue"**);  
 cycleTwo.startEngine();  
 Vehicle truckOne = VehicleFactory.*getVehicle*(**"Truck"**); *//Truck instance'i initialize edildi* Vehicle truckTwo = VehicleFactory.*getVehicle*(**"Truck"**); *//Truck instance'i zaten yukarıda create edildiği için bir kez daha initialize edilmedi* }  
}



**Creational Pattern - Builder Design Pattern (2. Örneği mutlaka incele)**

Çok basit bir örnek ile açıklayalım.Person isimli bir class’ımız var ve bu sınıfın içinde field’larımız var

**class** Person{  
 **private** String **firstName**;  
 **private** String **lastName**;  
 **private** String **address**;  
}

Bizim bu class’ın field’lerinden sadece firstName ve lastName’i kullanacağımızı düşünelim. Bunun için bir constructor inşa ettik ve şimdilik problemi çözdüm. Başka bir senaryo da sadece firstName’e ihtiyacımız oldu, tekrardan yeni bir constructor oluşturduk. Bu senaryolar böyle uzayıp gittiğinde işin içinden çıkılmaz bir hal oluşmaya başlar. Bunun önüne geçmek içinde Builder Design Pattern kullanılır

**interface** HousePlan {  
 **void** setBasement(String basement); *//temel* **void** setStructure(String structure); *//yapı,bina* **void** setRoof(String roof); *//çatı* **void** setInterior(String interior); *//iç mekan*}

Setter’ları görüldüğü gibi HousePlan içerisinde tanımlıyoruz.House class’ı HousePlan’ı implemente ediyor

**class** House **implements** HousePlan {  
 **private** String **basement**;  
 **private** String **structure**;  
 **private** String **roof**;  
 **private** String **interior**;  
 @Override  
 **public void** setBasement(String basement) {  
 **this**.**basement** = basement;  
 }  
 @Override  
 **public void** setStructure(String structure) {  
 **this**.**structure** = structure;  
 }  
 @Override  
 **public void** setRoof(String roof) {  
 **this**.**roof** = roof;  
 }  
 @Override  
 **public void** setInterior(String interior) {  
 **this**.**interior** = interior;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "House{"** +  
 **"basement='"** + **basement** + **'\''** +  
 **", structure='"** + **structure** + **'\''** +  
 **", roof='"** + **roof** + **'\''** +  
 **", interior='"** + **interior** + **'\''** +  
 **'}'**;  
 }  
}

*//Builder***interface** HouseBuilder{  
 **void** buildBasement();  
 **void** buildStructure();  
 **void** buildRoof();  
 **void** buildInterior();  
 House getHouse();  
}

Her bir HouseBuilder nesnesi HouseBuilder nesnesini implemente edecek

**class** IglooHouseBuilder **implements** HouseBuilder{  
 **private** House **house**;  
 **public** IglooHouseBuilder(){  
 **this**.**house** = **new** House();  
 }  
 @Override  
 **public void** buildBasement() {  
 **house**.setBasement(**"Ice Bars"**);  
 }  
 @Override  
 **public void** buildStructure() {  
 **house**.setStructure(**"Ice Blocks"**);  
 }  
 @Override  
 **public void** buildRoof() {  
 **house**.setRoof(**"Ice Dome"**);  
 }  
 @Override  
 **public void** buildInterior() {  
 **house**.setInterior(**"Ice carvings"**);  
 }  
 @Override  
 **public** House getHouse() {  
 **return this**.**house**;  
 }  
}

----------------------------------------------------------------------------------------------------------------------------------------------------

**class** TipiHouseBuilder **implements** HouseBuilder{  
 **private** House **house**;  
 **public** TipiHouseBuilder(){  
 **this**.**house** = **new** House();  
 }  
 @Override  
 **public void** buildBasement() {  
 **house**.setBasement(**"Wooden Poles"**);  
 }  
 @Override  
 **public void** buildStructure() {  
 **house**.setStructure(**"Wood and ice"**);  
 }  
 @Override  
 **public void** buildRoof() {  
 **house**.setRoof(**"Wood,caribou and seal skins"**);  
 }  
 @Override  
 **public void** buildInterior() {  
 **house**.setInterior(**"Fire wood"**);  
 }  
 @Override  
 **public** House getHouse() {  
 **return house**;  
 }  
}

Şimdi bu işleri yönetecek olan director interface’i

**interface** Director{  
 House getHouse();  
 **void** ConstructHouse();  
}

*//Concrete Director***class** CivilEngineer **implements** Director{  
 **private** HouseBuilder **houseBuilder**;  
 **public** CivilEngineer(HouseBuilder houseBuilder){  
 **this**.**houseBuilder** = houseBuilder;  
 }  
 @Override  
 **public** House getHouse() {  
 **return this**.**houseBuilder**.getHouse();  
 }  
 @Override  
 **public void** ConstructHouse() {  
 **this**.**houseBuilder**.buildBasement();  
 **this**.**houseBuilder**.buildStructure();  
 **this**.**houseBuilder**.buildInterior();  
 **this**.**houseBuilder**.buildRoof();  
 }  
}

Main Class;

**class** Test{  
 **public static void** main(String[] args) {  
 HouseBuilder iglooHouseBuilder = **new** IglooHouseBuilder();  
 Director director = **new** CivilEngineer(iglooHouseBuilder);  
 director.ConstructHouse();  
 House house = director.getHouse();  
 System.***out***.println(house);  
 }  
}

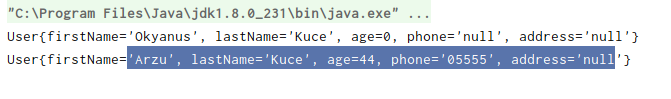


**Creational Pattern - Builder Design Pattern (Static class ile) -** User class’ı içerisinde static bir class olarak builder tanımlıyoruz ve istediğimiz gibi nesneyi tanımlayabiliyoruz.

**class** User {  
 **private final** String **firstName**; *//required* **private final** String **lastName**; *//required* **private final int age**; *//optional* **private final** String **phone**; *//optional* **private final** String **address**; *//optional  
 //private constructor* **private** User(UserBuilder userBuilder){  
 **this**.**firstName** = userBuilder.**firstName**;  
 **this**.**lastName** = userBuilder.**lastName**;  
 **this**.**age** = userBuilder.**age**;  
 **this**.**phone** = userBuilder.**phone**;  
 **this**.**address** = userBuilder.**address**;  
 }  
 *//All getter, and NO setter to provde immutability* **public** String getFirstName() {  
 **return firstName**;  
 }  
 **public** String getLastName() {  
 **return lastName**;  
 }  
 **public int** getAge() {  
 **return age**;  
 }  
 **public** String getPhone() {  
 **return phone**;  
 }  
 **public** String getAddress() {  
 **return address**;  
 }  
 @Override  
 **public** String toString() {  
 **return "User{"** +  
 **"firstName='"** + **firstName** + **'\''** +  
 **", lastName='"** + **lastName** + **'\''** +  
 **", age="** + **age** +  
 **", phone='"** + **phone** + **'\''** +  
 **", address='"** + **address** + **'\''** +  
 **'}'**;  
 }  
 **public static class** UserBuilder{  
 **private final** String **firstName**;*//required* **private final** String **lastName**;*//required* **private int age**;*//optional* **private** String **phone**;*//optional* **private** String **address**;*//optional  
 //Required alanlar* **public** UserBuilder(String firstName, String lastName) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 }  
 *//Optional* **public** UserBuilder age(**int** age) {  
 **this**.**age** = age;  
 **return this**;  
 }  
 *//Optional* **public** UserBuilder phone(String phone) {  
 **this**.**phone** = phone;  
 **return this**;  
 }  
 *//Optional* **public** UserBuilder address(String address) {  
 **this**.**address** = address;  
 **return this**;  
 }  
 **public** User build(){  
 User user = **new** User(**this**);  
 validateUserObject(user);  
 **return** user;  
 }  
 **private void** validateUserObject(User user) {  
 *//Do some basic validations to check  
 //if user object does not break any assumption of system* }  
 }  
}

Main Class;

**class** Test{  
 **public static void** main(String[] args) {  
 User okyanus = **new** User.UserBuilder(**"Okyanus"**,**"Kuce"**).build();  
 User arzu = **new** User.UserBuilder(**"Arzu"**,**"Kuce"**).age(44).phone(**"05555"**).build();  
 System.***out***.println(okyanus);  
 System.***out***.println(arzu);  
 }  
}

****

**Behavioral - Observer Design Pattern (İkinci örneğe mutlaka bakılmalı)**

One To Many ilişkilerde sıklıkla kullanılır.Bir nesnenin durumu değiştiğinde subscribe olan herkese bu bilgi gönderilir.Bir alışveriş sitesinde indirim olduğunu düşünelim, siteye tüm üye olan herkese bu bilgi gönderilir.Yada şöyle düşünelim yeni bir ürün çıkacak ve siz bu ürünü sürekli gidip mağazaya sormak mı istersiniz yoksa mağazanın size evet ürün geldi şeklinde bir mail atmasını mı istersiniz?

İlk olarak state class’ımızı create ediyorum;

*//state object  
//Bu değişmez bir class olmalıdır yanlışlıkla içeriği değiştirilememelidir***class** Message {  
 **final** String **messageContent**;  
 **public** Message(String messageContent) {  
 **this**.**messageContent** = messageContent;  
 }  
 **public** String getMessageContent() {  
 **return messageContent**;  
 }  
}

Observer Interface;

*//Observer***interface** Observer {  
 **void** update(Message content);  
}

* Observer interface’i içerisinde update methodu yer alır.Bu interface’i inherit eden her subscriber update methodunu override etmek zorundadır.Update methodu Subject üzerinde bir değişiklik olduğu anda tüm inherit etmiş objelere bunu bildirmek için kullanılır
* Observer’ı inherit eden class’lar.Yani updateleri alacak olan class’lar

*//Concrete Observer***class** MessageSubscriberOne **implements** Observer {  
 @Override  
 **public void** update(Message content) {  
 System.***out***.println(**"Message subscriber one : "** + content.getMessageContent());  
 }  
}  
**class** MessageSubscriberTwo **implements** Observer {  
 @Override  
 **public void** update(Message content) {  
 System.***out***.println(**"Message subscriber two : "** + content.getMessageContent());  
 }  
}  
**class** MessageSubscriberThree **implements** Observer{  
 @Override  
 **public void** update(Message content) {  
 System.***out***.println(**"Message subscriber three : "** + content.getMessageContent());  
 }  
}

Subject

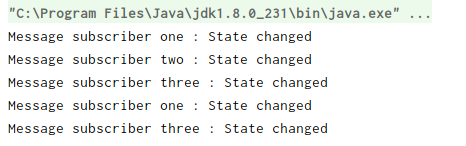
*//Subject***interface** Subject {  
 **void** attach(Observer observer);  
 **void** deAttach(Observer observer);  
 **void** notifyUpdate(Message message);  
}

Concrete Subject

*//concrete subject***class** MessagePublisher **implements** Subject {  
 List<Observer> **observerList** = **new** ArrayList<>();  
 @Override  
 **public void** attach(Observer observer) {  
 **observerList**.add(observer);  
 }  
 @Override  
 **public void** deAttach(Observer observer) {  
 **observerList**.remove(observer);  
 }  
 @Override  
 **public void** notifyUpdate(Message message) {  
 **observerList**.forEach(observer -> observer.update(message));  
 }  
}

Main class;

**class** Test{  
 **public static void** main(String[] args) {  
 MessageSubscriberOne msOne = **new** MessageSubscriberOne();  
 MessageSubscriberTwo msTwo = **new** MessageSubscriberTwo();  
 MessageSubscriberThree msThree = **new** MessageSubscriberThree();  
  
 MessagePublisher publisher = **new** MessagePublisher();  
 publisher.attach(msOne);  
 publisher.attach(msTwo);  
 publisher.attach(msThree);  
 publisher.notifyUpdate(**new** Message(**"State changed"**));  
 publisher.deAttach(msTwo);  
 publisher.notifyUpdate(**new** Message(**"State changed"**));  
 }  
}



**Behavioral - Observer Design Pattern (İkinci örnek)**

Business Logic

**class** ProductManager {  
 ArrayList<Observer> **observerList** = **new** ArrayList<>();  
 **public void** UpdatePrice(){  
 System.***out***.println(**"Product price changed"**);  
 Notify();  
 }  
 **void** attach(Observer observer){  
 **observerList**.add(observer);  
 }  
 **void** detach(Observer observer){  
 **observerList**.remove(observer);  
 }  
 **private void** Notify(){  
 **for** (Observer observer : **observerList**) {  
 observer.Update();  
 }  
 }  
}

Observer

**abstract class** Observer {  
 **abstract void** Update();  
}  
**class** CustomerObserver **extends** Observer {  
 @Override  
 **void** Update() {  
 System.***out***.println(**"Message to Customer : Product price changed"**);  
 }  
}

Main

**class** Test{  
 **public static void** main(String[] args) {  
 ProductManager pm = **new** ProductManager();  
 pm.attach(**new** CustomerObserver());  
 pm.UpdatePrice();  
 }  
}

**Behavioral - Iterator Design Pattern**

*foreach döngüsü*, iterasyon mantığıyla çalışan bir mekanizmadır. Haliyle bu mekanizmanın kullandığı kaynak niteliğindeki yapıları genel olarak koleksiyonlar ve diziler olarak düşünebiliriz. Iterator design pattern için temel amaç elimizdeki veri yığını üzerinde ki bu array, list, stack vs olabilir, döngüsel işlemleri sağlayabilmektir

* Iterator  
  Veri kümesi içerisinde dolaşmanın tüm şart ve imzasını bu arayüz belirlemektedir. Yani bir enumerator(sayıcı) görevi üstlenmektedir. Uzun lafın kısası, elimizdeki veri kümesi üzerinde döngü esnasında verileri/nesneleri elde edebilmemiz için gerekli işlemleri/kontrolleri/şartları/hususları tanımlar.

**interface** Iterator{  
 **public boolean** hasNext();  
 **public** Object next();  
}

* Aggregate  
  Veri kümesi içerisinde dolaşmak için bir Iterator interface’i tipinden Iterator yaratılmasını zorunlu tutan arayüzdür.

**interface** Container{  
 Iterator getIterator();  
}

* Class

**class** Person **implements** Container{  
 **public** String[] **names** = {**"Okyanus"**,**"Arzu"**};  
 @Override  
 **public** Iterator getIterator() {  
 **return new** PersonIterator();  
 }  
 **private class** PersonIterator **implements** Iterator{  
 **int index**=0;  
 @Override  
 **public boolean** hasNext() {  
 **if** (**names**.**length**>**index**){  
 **return true**;  
 }  
 **return false**;  
 }  
 @Override  
 **public** Object next() {  
 **if** (**this**.hasNext()){  
 **return names**[**index**++];  
 }  
 **return false**;  
 }  
 }  
}

* Main

**public static void** main(String[] args) {  
 Person person = **new** Person();  
 Iterator iter = person.getIterator();  
 **while**(iter.hasNext()){  
 String name = (String) iter.next();  
 System.***out***.println(name);  
 }  
}

PersonIterator inner class’ı içerisinde kendimize has algoritmalar oluşturup kullanabiliriz

**Structural - Proxy Design Pattern (İkinci örneğe mutlaka bakılmalı)**

Açıklama : Gerçek nesne ile isteyen yani client arasına ara bir katman olarak proxy konulur ve client’a proxy sunulur.Client real subject ile erişim kurmamış olur.Mesela online izlenen bir filmde Film real subject’dir araya bir Proxy eklenerek kısım kısım çekilen kısmını bize izletmeyi hedefler.Dolayısıyla oluşturulması zaman alan bir nesne yaratılması gerektiğinde, uzaktaki bir sunucuya erişilerek bir nesne yaratılması gerektiğinde, real subject’e ulaşmadan önce bazı kontrollerin yapılması gerektiğinde yararlı olacak bir desendir.Aslında basit bir cache’leme mekanizmasıdır

Açıklama : Proxy design pattern Client tarafından erişilecek nesneye vekalet eden bir tasarım desenidir. Burada vekaletten kasıt ilgili nesneyi kontrol edecek bir Proxy nesnesinin kullanılmasıdır.

Üç farklı durumda Proxy Design Pattern kullanılır.

Remote(Uzak) Proxy  
Remote(uzak) bir nesne kullanılacağı durumlarda kullanılabilir. Uzaktaki nesneye local bir temsilci sağlar ve gerekli kontrolleri yapmamıza olanak tanır.

Virtual Proxy  
Üretimi yahut kullanımı maliyetli nesnelerin oluşturulması veya kullanılması için tercih edilir. Buna örnek olarak genelde herkesin dillendirdiği resim yükleme işlevini verebiliriz. Yüksek boyutlu bir resmin boyutundan dolayı geç yüklenmesi durumunda verilen -yükleniyor- mesajı ve ardından yükleme işlemi bittiği anda resmin gösterilmesinde kullanılabilir.

Protection Proxy  
Yetkilendirme yahut login durumlarında kullanılabilir.

*//Subject***interface** IBank {  
 *//ödeme yap* **boolean** pay(**double** price);  
}

*//Real Subject***class** Bank **implements** IBank {  
 @Override  
 **public boolean** pay(**double** price) {  
 **if** (price < 100)  
 System.***out***.println(**"tutar 100 den az olamaz"**);  
 **else if** (price > 100)  
 System.***out***.println(**"tutar 100 den fazla olamaz"**);  
 **else** {  
 System.***out***.println(**"ödeme gerçekleşti"**);  
 **return true**;  
 }  
 **return false**;  
 }  
}

*//Proxy nesnesi***class** ProxyBank **implements** IBank {  
 **private** Bank **bank**;  
 **private boolean login**;  
 **private** String **userName**;  
 **private** String **password**;  
 **public** ProxyBank(String userName, String password) {  
 **this**.**userName** = userName;  
 **this**.**password** = password;  
 }  
 **boolean** enterLogin(){  
 **if** (**userName**.equals(**"oky"**) && **password**.equals(**"oky"**)){  
 **bank** = **new** Bank();  
 **login** = **true**;  
 System.***out***.println(**"Hesaba login yapıldı"**);  
 **return true**;  
 }  
 **else** System.***out***.println(**"Kullanıcı adı şifre hatalı"**);  
 **login** = **false**;  
 **return false**;  
 }  
  
 @Override  
 **public boolean** pay(**double** price) {  
 enterLogin();  
 **if** (!**login**){  
 System.***out***.println(**"hesaba giriş yapılmadığından dolayı ödeme yapılamıyor"**);  
 **return false**;  
 }  
 **bank**.pay(price);  
 **return true**;  
 }  
}

*//Main*

**public static void** main(String[] args) {  
 IBank bank = **new** ProxyBank(**"oky"**,**"2"**);  
 bank.pay(0);  
}

**Structural - Proxy Design Pattern (Örnek 2)**

Genelde verilen örneğe değinelim. Disk’ten bir image’in yüklendiği bir senaryo düşünelim;

**interface** Image{  
 **void** display();  
}

Image interface’ini implemente edecek RealSubject class’ımız;

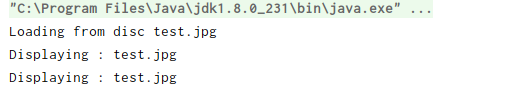
**class** RealImage **implements** Image{  
 **private** String **fileName**;  
 **public** RealImage(String fileName){  
 **this**.**fileName** = fileName;  
 loadFromDisc(fileName);  
 }  
 @Override  
 **public void** display() {  
 System.***out***.println(**"Displaying : "** + **fileName**);  
 }  
  
 **private void** loadFromDisc(String fileName){  
 System.***out***.println(**"Loading from disc "** + fileName);  
 }  
}

Proxy class’ımıza geldi sıra;

**class** ProxyImage **implements** Image{  
 **private** RealImage **realImage**;  
 **private** String **fileName**;  
 **public** ProxyImage(String fileName){  
 **this**.**fileName** = fileName;  
 }  
 @Override  
 **public void** display() {  
 **if** (**realImage** == **null**){  
 **realImage** = **new** RealImage(**fileName**);  
 }  
 **realImage**.display();  
 }  
}

Proxy’imizi kullanalım;

**public static void** main(String[] args) {  
 Image image = **new** ProxyImage(**"test.jpg"**);  
 image.display();  
 image.display();  
}



Görüldüğü gibi image.display’in ikinci çağırımın da disk’e gitmedi

**Behavioral - Command Design Pattern**

Komutların wrap edilip yani sarmalanıp tek bir nesne haline getirilmesini amaçlar. Herhangi bir methodu direk çağırıyor olmak bağımlılığı artıracaktır, bu bağımlılığı decoupling hale getirmek için kullanılır.Uzaktaki (RPC) bir methodu çalıştırmak içinde tavsiye edilmektedir.Client methodu çağırmak istediğin araya koyduğumuz aracıya gider ve aracı da methodu çalıştırır.Böylece client’i değiştirirsek method bundan etkilenmez,methodu değiştirirsek client bundan etkilenmez.Dolayısıyla decoupling sağlanmış olur.Aracı işini görecek olan nesneye **Invoker** adı veriliyor. “İşlemleri geri almak” konusuna nihayet gelebildik. COMMAND deseninin en yaygın kullanıldığı kısım bu başlıktır. Undo/Redo yapabilmek aslında işlemden etkilenen nesnelerin durumunu ileri/geri hareket ettirebilmek olarak düşünülebilir. Yani bir nesnenin durumunda (state) ileri geri gitmek istersek bize en uygun tasarım deseni; MEMENTO olacaktır. Ancak durum (state) temelli değil de süreci etkileyen işlemler penceresinden baktığımızda ise COMMAND deseni işimize çok yarayacaktır.

* 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.

Tüm electronic cihazların base class’ı

**interface** ElectronicDevice{  
 **void** on();  
 **void** off();  
 **void** volumeUp();  
 **void** volumeDown();  
}

Command interface

*//Command***interface** Command{  
 **void** execute();  
 *//void undo(); //kullanılabilir*}

Invoker

*//Invoker***class** RemoteControl{  
 Command **command**;  
 **public** RemoteControl(Command command){  
 **this**.**command** = command;  
 }  
  
 **public void** pressButton(){  
 **command**.execute();  
 }  
}

*//Örnek kullanım*

**class** TurnItAllOff **implements** Command{  
 List<ElectronicDevice> **electronicDevices**;  
 **public** TurnItAllOff(List<ElectronicDevice> electronicDevices) {  
 **this**.**electronicDevices** = electronicDevices;  
 }  
  
 @Override  
 **public void** execute() {  
 **for** (ElectronicDevice device : **electronicDevices**){  
 device.off();  
 }  
 }  
}

Receiver class’lar

*//Receiver***class** Television **implements** ElectronicDevice{  
 **private int volume** = 0;  
 @Override  
 **public void** on() {  
 System.***out***.println(**"TV is ON"**);  
 }  
  
 @Override  
 **public void** off() {  
 System.***out***.println(**"TV is OFF"**);  
 }  
  
 @Override  
 **public void** volumeUp() {  
 **volume**++;  
 System.***out***.println(**"TV volume is at : "** +**volume**);  
 }  
  
 @Override  
 **public void** volumeDown() {  
 **volume**--;  
 System.***out***.println(**"TV volume is at : "** +**volume**);  
 }  
}  
  
*//Receiver***class** Radio **implements** ElectronicDevice{  
 **private int volume** = 0;  
 @Override  
 **public void** on() {  
 System.***out***.println(**"Radio is ON"**);  
 }  
  
 @Override  
 **public void** off() {  
 System.***out***.println(**"Radio is OFF"**);  
 }  
  
 @Override  
 **public void** volumeUp() {  
 **volume**++;  
 System.***out***.println(**"Radio volume is at : "** +**volume**);  
 }  
  
 @Override  
 **public void** volumeDown() {  
 **volume**--;  
 System.***out***.println(**"Radio volume is at : "** +**volume**);  
 }  
}

Concrete Command

*//Concrete Command***class** TurnTvOn **implements** Command{  
 **private** ElectronicDevice **device**;  
 **public** TurnTvOn(ElectronicDevice device){  
 **this**.**device** = device;  
 }  
 @Override  
 **public void** execute() {  
 **device**.on();  
 }  
}  
**class** TurnTvOff **implements** Command{  
 **private** ElectronicDevice **device**;  
 **public** TurnTvOff(ElectronicDevice device){  
 **this**.**device** = device;  
 }  
 @Override  
 **public void** execute() {  
 **device**.off();  
 }  
}  
**class** TvVolumeUp **implements** Command{  
 **private** ElectronicDevice **device**;  
 **public** TvVolumeUp(ElectronicDevice device){  
 **this**.**device** = device;  
 }  
 @Override  
 **public void** execute() {  
 **device**.volumeUp();  
 }  
}  
**class** TvVolumeDown **implements** Command{  
 **private** ElectronicDevice **device**;  
 **public** TvVolumeDown(ElectronicDevice device){  
 **this**.**device** = device;  
 }  
 @Override  
 **public void** execute() {  
 **device**.volumeDown();  
 }  
}

Radio içinde Concrete Command class’ı yaratılmalı. Yer kaplamasın diye yazmadım

**public static void** main(String[] args) {  
 ElectronicDevice device = TVRemote.*getDevice*();  
 TurnTvOn tvOn = **new** TurnTvOn(device);  
 RemoteControl remoteControl = **new** RemoteControl(tvOn);  
 remoteControl.pressButton();  
  
 Television television = **new** Television();  
 Radio radio = **new** Radio();  
 List<ElectronicDevice> electronicDevices = **new** ArrayList<>();  
 electronicDevices.add(television);  
 electronicDevices.add(radio);  
 TurnItAllOff turnOffAllDevices = **new** TurnItAllOff(electronicDevices);  
 RemoteControl allDeviceOff = **new** RemoteControl(turnOffAllDevices);  
 allDeviceOff.pressButton();  
}

**Structural - Composite Design Pattern**

Bir kurumda çalışanların hiyerarşik yapısını tasarlayabilmek için yani ağaç yapılarında sıklıkla kullanılan bir patterndir. Compose design pattern’in görevi, nesneleri bir ağaç yapısında birleştirip uygulamanın genelindeki parça bütün ilişkisini yeniden düzenleyip şekillendirmektir

* Component
  + Bileşikler için temel soyut tanımlamalardır.
  + Bileşik işlemi için nesnelerin arayüzünü oluşturur.
  + Tüm sınıfların arayüzündeki varsayılan davranışı gerçekleştirir.
  + Yavru bileşenlere ulaşmamızı ve onları kontrol etmemizi sağlamak için bir arayüz tanımlar.
* Leaf
  + Bileşik işleminde yavru nesneleri temsil eder.
  + Tüm bileşen metodları yapraklar tarafından tamamlanır.
* Composite
  + Yaprakları olan bileşenleri temsil eder.
  + Çocuklarını yönlendiren metodları gerçekler.
  + Genelde çocuklarını görevlendirerek bileşik metodlarını gerçekler.

Composite design pattern için bilmemiz gereken iki temel tanım vardır. Bunlar;

Component: Alt bileşenleri olmayan bileşen. (Çalışanlar örnek olarak)

Composite: Alt bileşenleri mevcut olan bileşendir. (Manager’lar örnek olarak)

* Component içerisinde tüm nesneler için ortak olan method(lar) bulunur.Abstract bir class ya da Interface olarak tasarlanabilir

**interface** Department{  
 **void** printDepartmentName();  
}

* Leaf diğer nesnelere bir referans içermez ve Component’i implemente eder.

*//Leaf***class** FinancialDepartment **implements** Department{  
 **private** Integer **id**;  
 **private** String **name**;  
  
 *//default constructor, getter and setter* @Override  
 **public void** printDepartmentName() {  
 System.***out***.println(**"Department name : "** + getClass().getSimpleName());  
 }  
}

*//Leaf***class** SalesDepartment **implements** Department{  
 **private** Integer **id**;  
 **private** String **name**;  
  
 *//default constructor, getter and setter* @Override  
 **public void** printDepartmentName() {  
 System.***out***.println(**"Department name : "** + getClass().getSimpleName());  
 }

* Composite Leaf elemanlarına sahiptir.İçerisinde Component’lere ait listeyi tutar

*//Composite***class** HeadDepartment **implements** Department{  
 **private** Integer **id**;  
 **private** String **name**;  
 List<Department> **childDepartments**;  
 **public** HeadDepartment(Integer id, String name) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 **this**.**childDepartments** = **new** ArrayList<>();  
 }  
 @Override  
 **public void** printDepartmentName() {  
 **childDepartments**.forEach(Department::printDepartmentName);  
 }  
 **public void** addDepartment(Department department){  
 **childDepartments**.add(department);  
 }  
 **public void** removeDepartment(Department department){  
 **childDepartments**.remove(department);  
 }  
}

* Main

**public static void** main(String[] args) {  
 Department salesDepartment = **new** SalesDepartment(1,**"Sales"**);  
 salesDepartment.printDepartmentName();  
 Department financialDepartment = **new** FinancialDepartment(1, **"Financial"**);  
 financialDepartment.printDepartmentName();  
  
 HeadDepartment headDepartment = **new** HeadDepartment(1,**"Head Department"**);  
 headDepartment.addDepartment(salesDepartment);  
 headDepartment.addDepartment(financialDepartment);  
 headDepartment.printDepartmentName();  
}

Head Department iki departmanı da altında tutmaktadır.Ast üst ilişkisi,daha doğru tabir ile tree yapısı

**Structural - Decorator Design Pattern**

1. Şöyle bir durum düşünelim. Bir rapor formatı hazırladınız ve sundunuz. Ardından her bir satır başına çizgi istendi, başka bir gün hem çizgili hem imza yeri olan bir rapor istendi, veya bu tür istekler devam etti. Dolayısıyla basit rapor formatımızı dekore etmek mantığı için kullanılan bir pattern’dir. Yani bir nesneye yeni özellikler eklemek için kullanılır

Faydaları Nedir?

1. loosely-coupled uygulamalar yapmayı sağlar.  
2. Runtime zamanında(dinamik olarak) bir nesneye yeni özellikler eklenmesini sağlar.  
3. Özellikleri kalıtım yolu dışında [composition ve delegation](http://www.codesenior.com/en/tutorial/Distinguishing-Between-Delegation-Composition-And-Aggregation) ile de alınabilmesini sağlar.  
4. open-closed prensibinin uygulandığı tasarım desenidir.

**interface** Report{  
 String getText();*//raporun metni*}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**class** BasicReport **implements** Report{  
 *//Düz rapor hiç bir özelliği yok* **private** String **text**;  
 **public** BasicReport(String newText){  
 **text** = newText;  
 }  
 @Override  
 **public** String getText() {  
 **return text**;  
 }  
}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**class** LittleReport **extends** ReportDecorator{  
 *//küçük rapor* **public** LittleReport(Report report) {  
 **super**(report);  
 }  
  
 @Override  
 **public** String getText() {  
 **return super**.getText() + **"Kucuk rapor oluşturuldu"**;  
 }  
}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

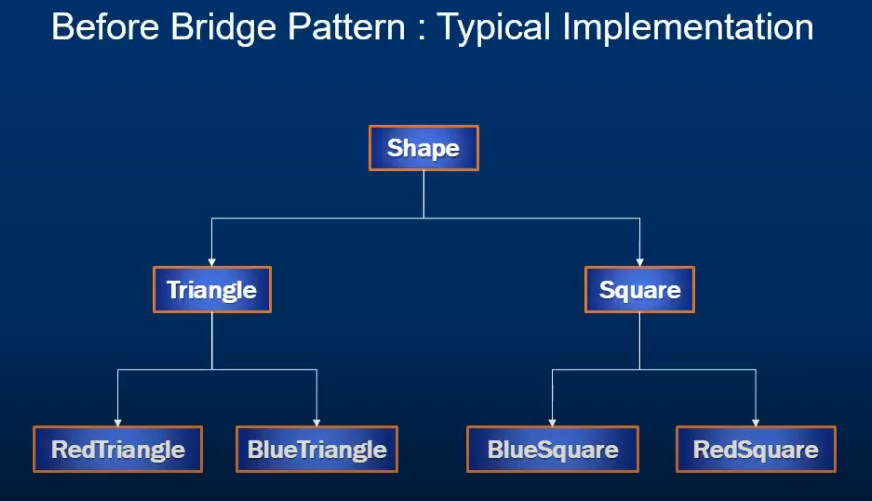
**class** LinedReport **extends** ReportDecorator{  
 *//çizgili rapor* **public** LinedReport(Report report) {  
 **super**(report);  
 }  
  
 @Override  
 **public** String getText() {  
 String text = **super**.getText();  
 **return** ReportUtil.*getLinedReport*(text);  
 }  
}

*//Report'ları oluşturmak için yardımcı bir class***class** ReportUtil{  
 **public static** String getLinedReport(String text){  
 *//logic yazmak istemediğim mock bir string yazdım  
 //gelen rapor text'inin her bir satırına çizgi eklemek aslında amacımız* String description = **"Her bir satıra çizgi ekle"** + text;  
 **return** description;  
 }  
}

*//Decorator***abstract class** ReportDecorator **implements** Report{  
 **private** Report **report**;  
 **public** ReportDecorator(Report report) {  
 **this**.**report** = report;  
 }  
 @Override  
 **public** String getText() {  
 **return report**.getText();  
 }  
}

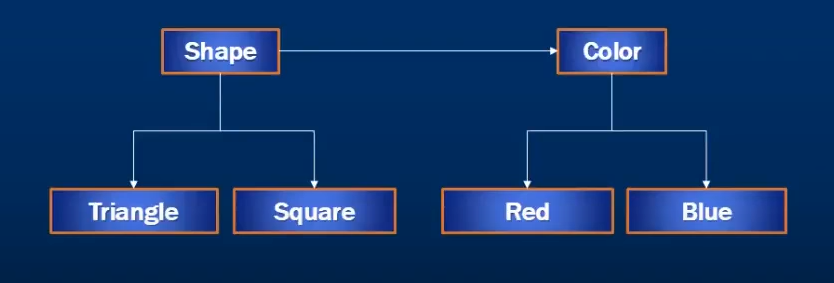
*//Main***public static void** main(String[] args) {  
 BasicReport basicReport = **new** BasicReport(**"asdadsasdads"**);  
 String text = basicReport.getText();  
 System.***out***.println(text);  
 LinedReport report = **new** LinedReport(basicReport);  
 String linedReportText= report.getText();  
 System.***out***.println(linedReportText);  
}

**Structural - Bridge Design Pattern**



Bridge Pattern kullanılmadan önce Shape base class’ından Triange ve Square adlı iki concrete class türüyor, bunlarda kendi aralarında BlueSquare ve RedSquare örneğinde olduğu gibi ayrışıyorlar. Bridge Design Pattern diyor ki; soyutlaşmış (abstract) bir yapıyı, implementasyondan ayır. Böylece bağımsız olarak geliştirilebilir iki yapı elde edersin.

Bridge pattern kullanıldığında;



Şöyle bir örnek üzerinden devam edelim;

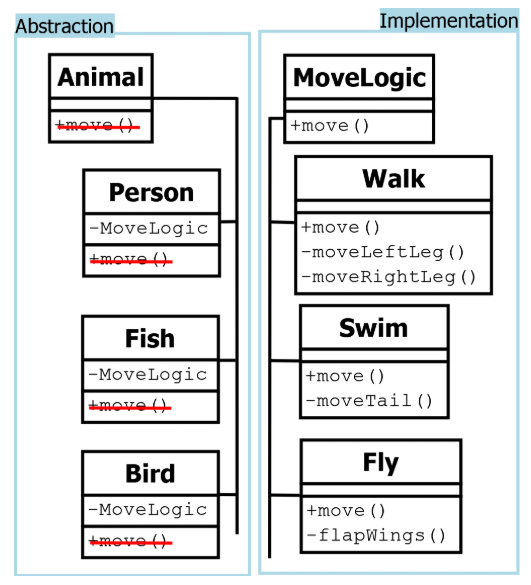
**(Abstraction)**

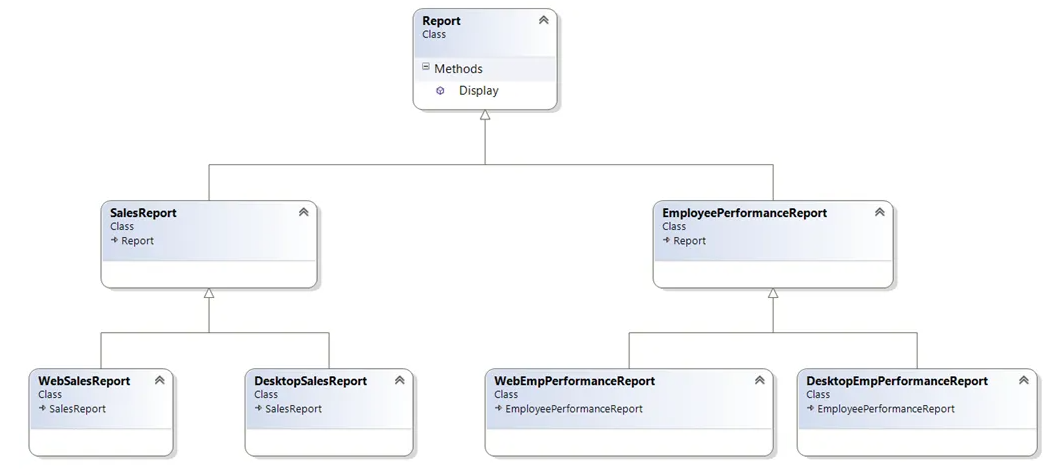
* İnsan yürüyerek hareket eder
* Bir balık yüzerek hareket eder
* Bir kuş uçarak hareket eder

Farklı yollarla hareket eden 3 nesne. Burada bridge pattern’in amacı move mantığını abstraction’dan ayırmaktır.Her bir nesne için move methodu başka şekilde implemente edilmektedir

**(Implementation)**

Move mantığı abstraction’dan ayrıştırılır





Bu örnek üzerinden devam edecek olursak. Bridge Design Pattern diyor ki; soyutlaşmış (abstract) bir yapıyı, implementasyondan ayır. Böylece bağımsız olarak geliştirilebilir iki yapı elde edersin.Implementasyon’dan kastettiği şey en son da elde edeceğimiz sınıftır yani bu örnekte DesktopSalesReport. WebEmpPerformanceReport ile DesktopSalesReport sınıfları ortak bir atadan türediklerine göre birbirleriyle akrabalar öyle değil mi? Belirttiğim iki sınıf geliştirilebilir bir modelde olabilmesi için akraba **olmamalılar.**  Nitekim Web formatında oluşturulmuş çalışan performans raporu başka bir şey, masaüstü formatında oluşturulmuş satış raporu ise bambaşka. Bu iki sınıf da **farklı iki formatta**bir rapordur. O halde bakın ne açığa çıktı! **Rapor Formatı, bu modelde soyutlaşmış bir yapıdır.**Yani, satış ya da çalışan performans raporunu oluştururken, hangi formatta (Masaüstü veya web) kaydetmesi gerektiğini söylemem yeterli olacak. Bu durumu ayarlamak için **ReportFormat** isminde bir interface oluşturuyorum ve ilgili formatları bu interface’den implemente ediyorum

**interface** ReportFormat{  
 **void** generate();  
}

Interface’i implemente eden Concrete class’lar

**class** WebFormat **implements** ReportFormat{  
 **public void** generate(){  
 System.***out***.println(**"Generate to the Web format"**);  
 }  
}  
  
**class** DesktopFormat **implements** ReportFormat{  
 **public void** generate(){  
 System.***out***.println(**"Generate to the Desktop format"**);  
 }  
}

Abstraction :

**abstract class** Report{  
 **protected** ReportFormat **reportFormat**;  
 **public** Report(ReportFormat newReportFormat){  
 **reportFormat** = newReportFormat;  
 }  
  
 **void** display(){  
 **reportFormat**.generate();  
 }  
}

Abstraction’ı extends eden class’lar:

**abstract class** Report{  
 **protected** ReportFormat **reportFormat**;  
 **public** Report(ReportFormat newReportFormat){  
 **reportFormat** = newReportFormat;  
 }  
  
 **void** display(){  
 **reportFormat**.generate();  
 }  
}  
  
**class** SalesReport **extends** Report{  
 **public** SalesReport(ReportFormat newReportFormat) {  
 **super**(newReportFormat);  
 }  
  
 @Override  
 **void** display() {  
 System.***out***.println(**"---Sales report---"**);  
 **super**.display();  
 }  
}  
  
**class** EmployeePerformaceReport **extends** Report{  
 **public** EmployeePerformaceReport(ReportFormat newReportFormat) {  
 **super**(newReportFormat);  
 }  
  
 @Override  
 **void** display() {  
 System.***out***.println(**"---Employee performance report---"**);  
 **super**.display();  
 }  
}

Main:

**public static void** main(String[] args) {  
 Report report = **new** SalesReport(**new** DesktopFormat());  
 report.display();  
 Report reportTwo = **new** EmployeePerformaceReport(**new** DesktopFormat());  
 reportTwo.display();  
}

**Structural - Facade Design Pattern**

**Facade design pattern’i aynı zamanda Singleton olarak da ayarlanmalıdır.**

Subsystem’ler içerisinde ki class’ları wrap ederek client tarafından kullanılmasını sağlar ve kolaylaştırır.Subsystem’in içerisinde ki tüm class’lar Facade nesnesi içerisinde yer almak zorundalardır.İstemciler subsystem’lere direk ulaşamıyorlar.Facade ile subsystem’ler arasında loosed coupling bir ilişki vardır.Bankacılık uygulamaları için çok uygundur.Crosscuttingconcern objelerinde kullanılabilir.Service Oriented içinde kullanılabiliyor

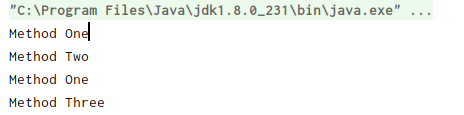
**class** SubSystemOne{  
 **void** methodOne(){  
 System.***out***.println(**"Method One"**);  
 }  
}  
**class** SubSystemTwo{  
 **void** methodTwo(){  
 System.***out***.println(**"Method Two"**);  
 }  
}  
**class** SubSystemThree{  
 **void** methodThree(){  
 System.***out***.println(**"Method Three"**);  
 }  
}

SubSystem’leri wrap edecek facade class’ı

**class** Facade{  
 **private** SubSystemOne **subSystemOne**;  
 **private** SubSystemTwo **subSystemTwo**;  
 **private** SubSystemThree **subSystemThree**;  
 **public** Facade(){  
 **subSystemOne** = **new** SubSystemOne();  
 **subSystemTwo** = **new** SubSystemTwo();  
 **subSystemThree** = **new** SubSystemThree();  
 }  
 **void** MethodA() {  
 **subSystemOne**.methodOne();  
 **subSystemTwo**.methodTwo();  
 }  
 **void** MethodB(){  
 **subSystemOne**.methodOne();  
 **subSystemThree**.methodThree();  
 }  
}

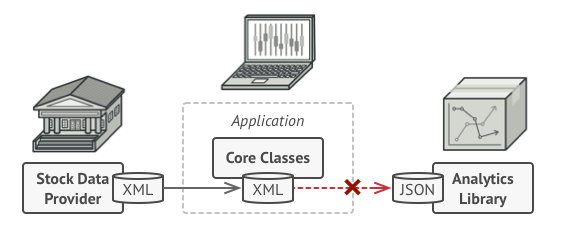
Main class;

**public static void** main(String[] args) {  
 Facade facade = **new** Facade();  
 facade.MethodA();  
 facade.MethodB();  
}



**Structural - Adapter Design Pattern**

Şöyle bir örnek üzerinden pattern’i açıklamaya çalışalım. Örnek olarak Tc Kimlik No Sorgulama sitesinden bize bir servis gelecek. Biz bu servis hakkında hiçbir bilgiye sahip olamayacağız dolayısıyla araya bir adapter nesnesi ekleyip kendi kodumuza dahil etmeye çalışacağız. Bu örnekte Log4Net dll’inden örneklendirme yapacağız.



Yukarıda ki örnekten yola çıkarak şöyle bir açıklama getirelim.Stock Data Priveder veriyi XML olarak gönderiyor, bizim Core Class’larımızın olduğu application JSON okuyor. Dolayısıyla bir adaptöre ihtiyacımız var ki XML’i bizim kullanabileceğimiz JSON haline çevirsin

**interface** Logger{  
 **void** log();  
}

Business nesnemiz;

**class** ProductManager{  
 **private** Logger **logger**;  
 **public** ProductManager(Logger logger){  
 **this**.**logger**=logger;  
 }  
 **void** save(){  
 System.***out***.println(**"Product saved"**);  
 **logger**.log();  
 }  
}

Interface’i implemente eden ve edemeyen class’larımız

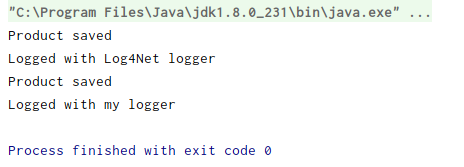
**class** MyLogger **implements** Logger{  
 @Override  
 **public void** log() {  
 System.***out***.println(**"Logged with my logger"**);  
 }  
}  
**class** Log4NetLogger{  
 *//dll şekilde download edildiğini ve Logger interface'inden implemente edemeyeceğimizi düşünelim  
 //kendi başına çalışması gerekiyor* **public void** log() {  
 System.***out***.println(**"Logged with Log4Net logger"**);  
 }  
}

Log4NetLogger sistemimize entegre olabilmesi için araya bir adapter eklememiz gerekiyor.İşte adapter design pattern bu iş ile ilgileniyor

**class** Log4NetLoggerAdapter **implements** Logger{  
 @Override  
 **public void** log() {  
 Log4NetLogger logger = **new** Log4NetLogger();  
 logger.log();  
 }  
}

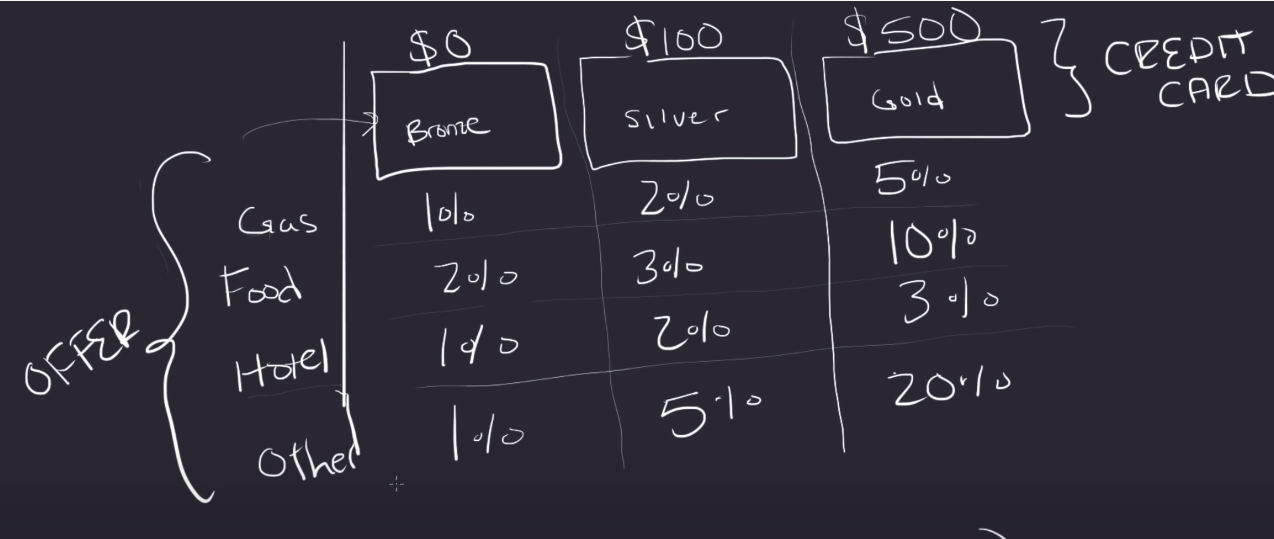
Main

**public static void** main(String[] args) {  
 ProductManager pm = **new** ProductManager(**new** Log4NetLoggerAdapter());  
 pm.save();  
 ProductManager pmTwo = **new** ProductManager(**new** MyLogger());  
 pmTwo.save();  
}

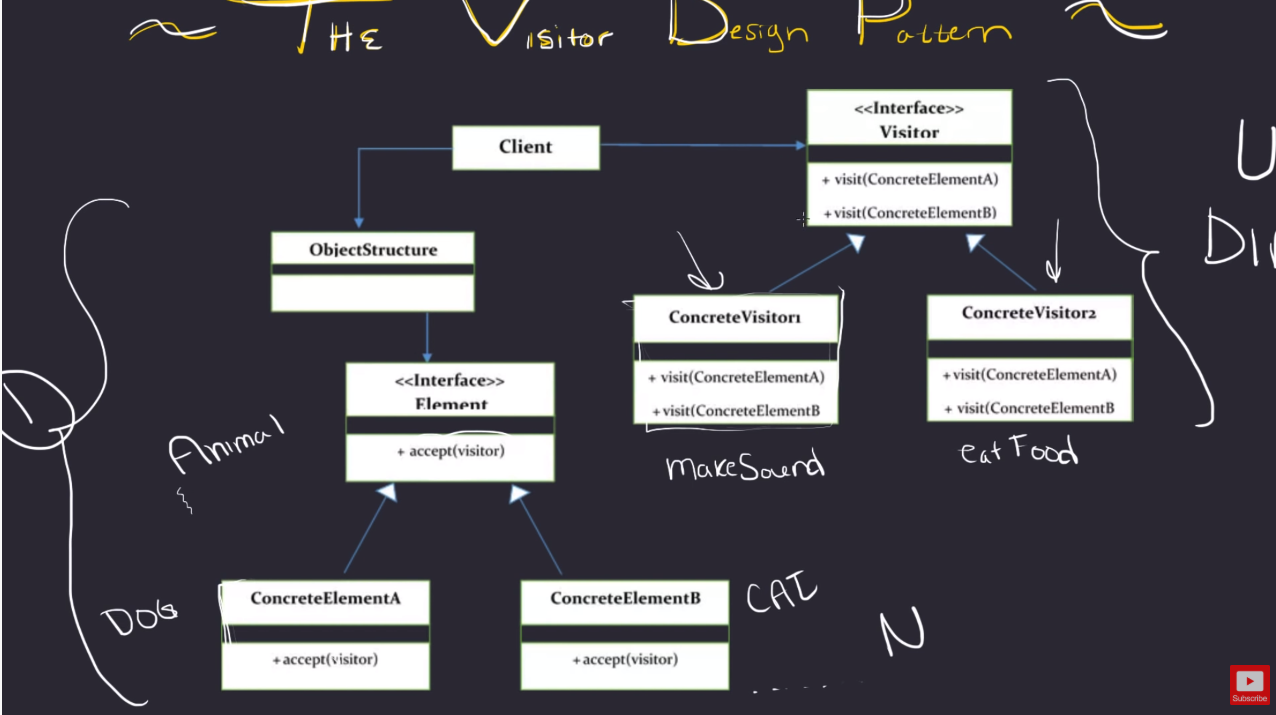


**Behavioral - Visitor Design Pattern**

Aslında basit bir amacı var, o da şu; var olan sınıfların hiyerarşik yapılarını ve mevcut yapılarını değiştirmeden yeni metotlar eklemek. Yeni metotlarımız visitor sınıfı üzerinde tanımlanır ve mevcut sınıflar kendilerini bu visitor sınıfa parametre olarak aktarıp gerekli işlemleri yaparlar



Offer bir nesne, Credit card ayrı bir nesnedir. Bronze,Silver ve Gold card’lar için alışverişlerde geri alınacak değerler farklıdır



Visitor’lar burada offerType’lar olacak.Sağ tarafta yer alan memeli parantezin orası

ConcreteVisitor1 (Gas Offer)

ConcreteVisitor2(Food Offer)

ConcreteVisitor3(Hotel Offer)

ConcreteVisitor4(Other Offer)

Element

*//Interface element***interface** CreditCard{  
 String getName();  
 **void** accept(OfferVisitor v);  
}

*//Interface visitor*

**interface** OfferVisitor{  
 **void** visitBronzeCreditCard(BronzeCard bronzeCard);  
 **void** visitSilverCreditCard(SilverCard silverCard);  
 **void** visitGoldCreditCard(GoldCard goldCard);  
}

Element’in concrete class’ları

**class** BronzeCard **implements** CreditCard{  
 @Override  
 **public** String getName() {  
 **return "Bronze Card"**;  
 }  
 @Override  
 **public void** accept(OfferVisitor v) {  
 v.visitBronzeCreditCard(**this**);  
 }  
}  
**class** SilverCard **implements** CreditCard{  
 **public** String getName(){  
 **return "Silver Card"**;  
 }  
 **public void** accept(OfferVisitor v){  
 v.visitSilverCreditCard(**this**);  
 }  
}  
**class** GoldCard **implements** CreditCard{  
 **public** String getName(){  
 **return "Gold Card"**;  
 }  
 @Override  
 **public void** accept(OfferVisitor v) {  
 v.visitGoldCreditCard(**this**);  
 }  
}

Visitor’ın Concrete Class’ları

**class** GasOfferVisitor **implements** OfferVisitor{  
 @Override  
 **public void** visitBronzeCreditCard(BronzeCard bronzeCard) {  
 System.***out***.println(**"We are computing the cashback for a bronze card buying gas"**);  
 }  
 @Override  
 **public void** visitSilverCreditCard(SilverCard silverCard) {  
 System.***out***.println(**"We are computing the cashback for a silver card buying gas"**);  
 }  
 @Override  
 **public void** visitGoldCreditCard(GoldCard goldCard) {  
 System.***out***.println(**"We are computing the cashback for a gold card buying gas"**);  
 }  
}  
  
**class** HotelOfferVisitor **implements** OfferVisitor{  
 @Override  
 **public void** visitBronzeCreditCard(BronzeCard bronzeCard) {  
 System.***out***.println(**"We are computing the cashback for a bronze card buying hotel reservation"**);  
 }  
  
 @Override  
 **public void** visitSilverCreditCard(SilverCard silverCard) {  
 System.***out***.println(**"We are computing the cashback for a silver card buying hotel reservation"**);  
 }  
  
 @Override  
 **public void** visitGoldCreditCard(GoldCard goldCard) {  
 System.***out***.println(**"We are computing the cashback for a gold card buying hotel reservation"**);  
 }  
}

Main

**public static void** main(String[] args) {  
 OfferVisitor hotelOfferVisitor = **new** HotelOfferVisitor();  
 OfferVisitor gasOfferVisitor = **new** GasOfferVisitor();  
  
 CreditCard bronzeCard = **new** BronzeCard();  
 CreditCard silverCard = **new** SilverCard();  
 CreditCard goldCard = **new** GoldCard();  
 bronzeCard.accept(hotelOfferVisitor);  
 silverCard.accept(hotelOfferVisitor);  
 goldCard.accept(hotelOfferVisitor);  
 bronzeCard.accept(gasOfferVisitor);  
 silverCard.accept(gasOfferVisitor);  
 goldCard.accept(gasOfferVisitor);  
}

**Behavioral - State Design Pattern (2. Örneğe mutlaka bak)**

**State** tasarım kalıbının kullanılması, nesnelerin durumlarına bağlı değişen davranışlarının karmaşık koşul ve kontrol **(if/else, switch)** ifadeleriyle yönetilmesini önler. Bir nesnenin internal state’inde meydana gelecek değişimler sonrası runtime’da dynamic bir şekilde davranışlar göstermesi için kullanılır.Nesnenin internal state’ini taşıyan nesneye context diyeceğiz. Birden fazla davranış ve doğal olarak durum olabileceğinden, Context’in farklı durumlarına ulaşılabilmeli ve state’ler arasında ki Transitions’ları sağlayabilmesi gerekmektedir.

**interface** State {  
 **void** doAction(Context context);  
}

Context içerisinde State’i aggregate etmektedir;

**class** Context {  
 **private** State **state**;  
  
 **public** State getState() {  
 **return state**;  
 }  
  
 **public void** setState(State state) {  
 **this**.**state** = state;  
 }  
}

Concrete State’ler;

**class** ModifiedState **implements** State {  
 @Override  
 **public void** doAction(Context context) {  
 System.***out***.println(**"State modified at the moment"**);  
 context.setState(**this**);  
 }  
}  
**class** DeletedState **implements** State{  
 @Override  
 **public void** doAction(Context context) {  
 System.***out***.println(**"State deleted at the moment"**);  
 context.setState(**this**);  
 }  
}

Main methodu;

**public static void** main(String[] args) {  
 Context context = **new** Context();  
 ModifiedState modifiedState = **new** ModifiedState();  
 modifiedState.doAction(context);  
 DeletedState deletedState = **new** DeletedState();  
 deletedState.doAction(context);  
}

**Behavioral - State Design Pattern**

**interface** PackageState{  
 **void** next(Context pkg);  
 **void** prev(Context pkg);  
 **void** printStatus();  
}

Context

**class** Context{  
 **private** PackageState **state** = **new** OrderedState(); *//henüz yeni sipariş edilmiş order* **public** PackageState getState() {  
 **return state**;  
 }  
 **public void** setState(PackageState state) {  
 **this**.**state** = state;  
 }  
 **void** next(){  
 **state**.next(**this**);  
 }  
 **void** prev(){  
 **state**.prev(**this**);  
 }  
 **void** printStatus(){  
 **state**.printStatus();  
 }  
}

Concrete Class’lar

**class** OrderedState **implements** PackageState{  
 @Override  
 **public void** next(Context pkg) {  
 pkg.setState(**new** DeliveredState());  
 }  
 @Override  
 **public void** prev(Context pkg) {  
 System.***out***.println(**"The package is in its root state."**);  
 }  
 @Override  
 **public void** printStatus() {  
 System.***out***.println(**"Package ordered, not delivered to the office yet."**);  
 }  
}  
**class** DeliveredState **implements** PackageState {  
 @Override  
 **public void** next(Context pkg) {  
 pkg.setState(**new** ReceivedState());  
 }  
 @Override  
 **public void** prev(Context pkg) {  
 pkg.setState(**new** OrderedState());  
 }  
 @Override  
 **public void** printStatus() {  
 System.***out***.println(**"Package delivered to post office, not received yet."**);  
 }  
}  
**class** ReceivedState **implements** PackageState {  
 @Override  
 **public void** next(Context pkg) {  
 System.***out***.println(**"This package is already received by a client."**);  
 }  
 @Override  
 **public void** prev(Context pkg) {  
 pkg.setState(**new** DeliveredState());  
 }  
 @Override  
 **public void** printStatus() {  
 System.***out***.println(**"Package was received by client."**);  
 }  
}

Main

**public static void** main(String[] args) {  
 Context context = **new** Context();  
 context.printStatus();  
 context.next();  
 context.printStatus();  
 context.next();  
 context.printStatus();  
 context.next();  
 context.printStatus();  
}

**Behavioral - Template Method Design Pattern (2. Örneğe mutlaka bakılmalı)**

**Strategy design patterni davranışın tamamen değiştiği durumlarda, Template design pattern ise bir kısmı değiştiği durumlarda tercih etmekteyiz.**

Örnekte **HotDrink** parent class’ı içerisinde bir **doIt**() adlı template metodu hazırlanmış, **Tea** ve **nescafe** içecekleri için gerekli yerlerde değişiklik yaparak kullanılmıştır. Bu template içinde tanımlanmış **heatWater** ve **fillTheGlass** fonksiyonlarının tüm içecekler için aynı olması nedeniyle bu fonksiyonlar geçersiz kılınması isteğe bağlı olacak şekilde şekilde tanımlanmıştır. **Contents** methodu ise **Tea** ve **nescafe** için farklılık göstereceğinden bu fonksiyon tanımlanması zorunlu olacak şekilde **abstract**olarak tanımlanmıştır. **Tea** ve **Nescafe** alt sınıflarında **Content** farklı olarak tanımlanmış ve daha sonra **doIt**() metodu çağırılarak program çıktıları elde edilmiştir. Bu durumda programa yeni bir içecek eklenmesi gerektiğinde bütün kodları kopyalamak yerine yeni içecek sınıfını **HotDrink** parent classından türetip **Contents** metodunu tanımlamak yeterli olur. Böylece hem kod tekrarının önüne geçilmiş hem de sade ve anlaşılır bir kod yazılmış  olur.

**abstract class** HotDrink{  
 **void** heatWater(){  
 System.***out***.println(**"Su isitildi"**);  
 }  
 *//Primitive operation* **abstract void** contents(); *//içerik* **void** fillTheGlass(){  
 System.***out***.println(**"Icecek bardaga dolduruldu"**);  
 }  
 *//Template method* final**void** doIt(){  
 heatWater();  
 contents();  
 fillTheGlass();  
 System.***out***.println(**"Icecek hazırlandı"**);  
 }  
}

Concrete class’lar

**class** Tea **extends** HotDrink{  
 @Override  
 **void** contents() {  
 System.***out***.println(**"Cay hazirlandi"**);  
 }  
}  
  
**class** Nescafe **extends** HotDrink{  
 @Override  
 **void** contents() {  
 System.***out***.println(**"Nescafe hazirlandi"**);  
 }  
}

Main

**public static void** main(String[] args) {  
 HotDrink tea = **new** Tea();  
 tea.doIt();  
 HotDrink nescafe = **new** Nescafe();  
 nescafe.doIt();  
}

**Behavioral - Template Method Design Pattern (2.Örnek)**

Template method design pattern’i bir algoritmayı yürütmek için adımları tanımlar ve alt sınıfların tümü veya bazıları için ortak olabilecek varsayılan implementasyonları sağlayabilir. Bu deseni bir örnekle anlayalım, bir ev inşa etmek için bir algoritma sağlamak istediğimizi varsayalım. Bir ev inşa etmek için adımlar atılmalıdır - bina temeli, bina direkleri, bina duvarları ve pencereler. Önemli olan nokta, yürütme sırasını değiştirememiz çünkü temeli inşa etmeden önce pencereleri inşa edemiyoruz. Yani bu durumda biz ev inşa etmek için farklı yöntemler kullanacak bir template method yöntemi oluşturabiliriz.Alt class’ların template methodunu geçersiz kılmamaları için mutlaka method imzasını **final** olarak işaretlemeliyiz

**abstract class** HouseTemplate{  
 *//template method* **public final void** buildHouse(){  
 buildFoundation(); *//temel* buildPillars(); *//sütunlar* buildWalls(); *//duvarlar* buildWindows(); *//pencereler* System.***out***.println(**"House is built"**);  
 }  
 *//default implementation* **private void** buildWindows(){  
 System.***out***.println(**"Building glass windows"**);  
 }  
 **private void** buildFoundation(){  
 *//Çimento,demir çubuklar ve kum ile temel oluşturma* System.***out***.println(**"Building foundation with cement,iron rods and sand"**);  
 }  
 *//methods to be implemented by subclasses* **public abstract void** buildWalls();  
 **public abstract void** buildPillars();  
}

Concrete Classes;

**class** WoodenHouse **extends** HouseTemplate{  
 @Override  
 **public void** buildWalls() {  
 System.***out***.println(**"Building wooden walls"**);  
 }  
 @Override  
 **public void** buildPillars() {  
 System.***out***.println(**"Building Pillars with Wood coating"**);  
 }  
}  
**class** GlassHouse **extends** HouseTemplate{  
 @Override  
 **public void** buildWalls() {  
 System.***out***.println(**"Building glass walls"**);  
 }  
 @Override  
 **public void** buildPillars() {  
 System.***out***.println(**"Building Pillars with glass coating"**);  
 }  
}

Main;

**public static void** main(String[] args) {  
 HouseTemplate woodenHouse = **new** WoodenHouse();  
 woodenHouse.buildHouse();  
}

**Behavioral - Chain of Responsibility Design Pattern (2. Örnek mutlaka bakılmalı)**

Birbirini takip eden iş dizisine ait process'leri redirect ve handle etmek, yada istekte bulunan-confirm eden süreçleri için çözüm olarak ortaya çıkmış bir tasarım desendir. Yani iş client tarafından receiver’lara request edilir, receiver çözemeyeceği bir şey ise zincirin 2. Elemanına request’i gönderir bu böyle çözüm buluncaya kadar devam eder. Ancak pratikte CoR deseni, zincirdeki eleman sayısının 10’u aşmadığı durumlar için uygundur

DoFactory açıklaması : **Chain of Responsibility** deseni, ortak bir **mesaj**veya **talebin(Request),**birbirlerine **zayıf bir şekilde bağlanmış(Loosly Coupled)** nesneler arasında gezdirilmesi ve bu zincir içerisinde asıl sorumlu olanı tarafından ele alınması gerektiği vakalarda kullanılmaktadır.

Bir örnek ile anlatmak gerekirse;

1. Müşteri 480 bin TL lik para çekme isteğini veznedar'a iletir.
2. Veznedar bu isteği alır ve kontrol eder eğer onaylayabileceği bir tutar ise onaylar, onaylayabileceği bir tutar değilse yöneticisine gönderir,
3. Yönetici isteği alır  onaylayabileceği bir tutar değilse müdüre iletir,
4. Müdür kontrol eder eğer onaylayabileceği bir tutar değilse bölge sorumlusunun onayına gönderir,
5. Bölge sorumlusu onaylar ve para müşteriye verilir.

Bir başka örnek;

Otomatik ürün makinelerine ait jeton slotları verilmektedir. Her tip jeton için bir slot oluşturmak aslında arka arkaya if blokları yazarak, gelen talebin anlaşılmaya çalışılmasına benzetilebilir. Bunun yerine makine üzerinde, her bir jetonu ele alan tek bir slot tasarlanır**(Handler).**Ürünü satın almak isteyen kişinin attığı jeton, verdiği komuta(Cola, çikolata vs istemek gibi) ve jetonun tipine göre, içeride uygun olan saklama alanına**(ConcreteHandler)** düşecektir. Sonrasında ise süreç, jetonun uygun olan saklama alanında değerlendirilerek istenilen ürünün teslim edilmesiyle tamamlanacaktır

Örnek ; Bir service’in hangi lokasyonda çalıştığı bilgisi zincir içerisinden bulunacak

**enum** ServiceLocation{  
 ***LocalMachine***,  
 ***Intranet***,  
 ***Internet***,  
 ***SecureZone***}

*//Chain içerisinde ki nesnelerde dolaşacak olan class***class** ServiceInfo{  
 **private** String **name**;  
 **private** ServiceLocation **serviceLocation**;  
 **public** String getName() {  
 **return name**;  
 }  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
 **public** ServiceLocation getServiceLocation() {  
 **return serviceLocation**;  
 }  
 **public void** setServiceLocation(ServiceLocation serviceLocation) {  
 **this**.**serviceLocation** = serviceLocation;  
 }  
}

*//Handler*

**abstract class** ServiceHandler{  
 **protected** ServiceHandler **successor**;  
 **public void** setSuccessor(ServiceHandler successor) {  
 **this**.**successor** = successor;  
 }  
 **public abstract void** processRequest(ServiceInfo serviceInfo);  
}

*//Concrete Handler  
// Servisin yerel makineye ait olma durumunu ele alır.***class** LocalMachineHandler **extends** ServiceHandler{  
 @Override  
 **public void** processRequest(ServiceInfo serviceInfo) {  
 **if** (serviceInfo.getServiceLocation() == ServiceLocation.***LocalMachine***){  
 System.***out***.println(**"Yerel makinada yer alan bir servis"**);  
 } **else if**(**successor**!=**null**){  
 **successor**.processRequest(serviceInfo);  
 }  
 }  
}  
*//Concrete Handler  
// Servisin Intranet üzerinde olma durumunu ele alır.***class** IntranetHandler **extends** ServiceHandler{  
 @Override  
 **public void** processRequest(ServiceInfo serviceInfo) {  
 **if** (serviceInfo.getServiceLocation()==ServiceLocation.***Intranet***){  
 System.***out***.println(**"Intranet üzerinde çalışan bir servis"**);  
 } **else if**(**successor**!=**null**){  
 **successor**.processRequest(serviceInfo);  
 }  
 }  
}  
*//Concrete Handler  
// Servisin Internet üzerinde olma durumunu ele alır.***class** InternetHandler **extends** ServiceHandler{  
 @Override  
 **public void** processRequest(ServiceInfo serviceInfo) {  
 **if** (serviceInfo.getServiceLocation()==ServiceLocation.***Internet***){  
 System.***out***.println(**"Internet üzerinde çalışan bir servis"**);  
 } **else if**(**successor**!=**null**){  
 **successor**.processRequest(serviceInfo);  
 }  
 }  
}  
*//Concrete Handler  
// Servisin Secure Zone üzerinde olma durumunu ele alır.  
//next successor son servis olduğu için yoktur***class** SecureZoneHandler **extends** ServiceHandler{  
 @Override  
 **public void** processRequest(ServiceInfo serviceInfo) {  
 **if** (serviceInfo.getServiceLocation()==ServiceLocation.***SecureZone***){  
 System.***out***.println(**"Secure zone üzerinde çalışan bir servis"**);  
 } **else** {  
 System.***out***.println(**"servis bulunamadı"**);  
 }  
 }  
}

Main

**public static void** main(String[] args) {  
 ServiceHandler localMachineHandler = **new** LocalMachineHandler();  
 ServiceHandler intranetHandler = **new** IntranetHandler();  
 ServiceHandler internetHandler = **new** InternetHandler();  
 ServiceHandler secureZoneHandler = **new** SecureZoneHandler();  
 localMachineHandler.setSuccessor(intranetHandler);  
 intranetHandler.setSuccessor(internetHandler);  
 internetHandler.setSuccessor(secureZoneHandler);  
 ServiceInfo info = **new** ServiceInfo();  
 info.setName(**"Order Process Service"**);  
 info.setServiceLocation(ServiceLocation.***Internet***);  
 localMachineHandler.processRequest(info);  
}

**Behavioral - Chain of Responsibility Design Pattern (2. Örnek)**

Özellikle loglama,hata yönetimi vb. işlerde çeşitli kurallara bağlı kod yönlendirme işlemlerini gerçekleştirmek amacıyla kullanılan tasarım desenlerinden biridir

*//harcama***class** Expense{  
 **protected** String **detail**;  
 **protected** Double **amount**;  
 **public** Expense(String newDetail,Double newAmount){  
 **detail** = newDetail;  
 **amount** = newAmount;  
 }  
}

Bu harcamayı yönetecek abstract class;

**abstract class** ExpenseHandlerBase{  
 **protected** ExpenseHandlerBase **successor**;  
 **public abstract void** HandleExpense(Expense expense);*//harcamayı yönetme süreci* **public void** setSuccessor(ExpenseHandlerBase newSuccessor){  
 **successor** = newSuccessor;  
 }  
}

Expense base class’ının concrete’leri

*//yönetici***class** Manager **extends** ExpenseHandlerBase{  
 @Override  
 **public void** HandleExpense(Expense expense) {  
 **if** (expense.**amount**<=100){  
 System.***out***.println(**"Manager handled the expense"**); *//manager bu ödemeyi yetkisi dahilinde yapabilir* } **else if**(**successor**!=**null**){  
 *//manager'in yetkisini aşan bir miktar gelirse bir üstüne yani successor'une gönderdik  
 //manager'in bir üstü yani successor'ü varsa ödemeyi ona yönlendir demiş olduk* **successor**.HandleExpense(expense);  
 }  
 }  
}  
*//başkan yardımcısı***class** VicePresident **extends** ExpenseHandlerBase{  
 @Override  
 **public void** HandleExpense(Expense expense) {  
 **if** (expense.**amount**>100 && expense.**amount**<=1000){  
 System.***out***.println(**"Vice president handled the expense"**);  
 } **else if**(**successor**!=**null**){  
 **successor**.HandleExpense(expense);  
 }  
 }  
}  
*//başkan***class** President **extends** ExpenseHandlerBase{  
 @Override  
 **public void** HandleExpense(Expense expense) {  
 **if** (expense.**amount**>1000){  
 System.***out***.println(**"President handled the expense"**); *//manager bu ödemeyi yetkisi dahilinde yapabilir* }  
 }  
}

Main

**public static void** main(String[] args) {  
 Manager manager = **new** Manager();  
 VicePresident vicePresident = **new** VicePresident();  
 President president = **new** President();  
 manager.setSuccessor(vicePresident);  
 vicePresident.setSuccessor(president);  
 Expense expense = **new** Expense(**"Training"**,2000.5);  
 manager.HandleExpense(expense);  
}

**Creational - Singleton Design Pattern**

* Bu tasarım örüntüsündeki amaç, bir class’tan sadece bir instance yaratılmasını sağlar. Yani herhangi bir class’tan bir instance yaratılmak istendiğinde, eğer daha önce yaratılmış bir instance yoksa yeni yaratılır. Daha önce yaratılmış ise var olan instance kullanılır.
* **Singleton pattern logging, caching ve thread havuzları içinde kullanılırSingleton design pattern, Abstract Factory,Builder,Prototype ve Facade desenlerinin içerisinde de kullanılır**
* Diyelim ki bir üyelik sistemi projeniz var ve bu üyelik sistemi içerisinde kullanıcı bir hareket gerçekleştirdi. Bu gerçekleşen hareketin tarihini, saatini vs. kaydetmeniz gerekiyor. İşte bu durumda sistem kullanıcıya özel bir veri değil standart bir veri üretmesi gerekir. Bu durumda da Singleton Design Pattern uygulanarak bu ihtiyaç karşılanabilir. Singleton Design Pattern uygulanacağı zaman kullanıcıya ya da parametreye bağlı olarak çıktı üretilmeyeceği yani herkes için standart bir akış uygulanacak demektir.

Desing pattern neden kullanılmalıdır?

* Herkes bu nesneyi kullanıyormu?
* Yada aynı nesneyi herkes kullanacak mı?
* Çok nadir kullanılan bir nesne ise singleton olarak tasarlanmamalıdır

**class** Singleton{  
 **private volatile static** Singleton *singleton*;  
 **private** Singleton(){} *//private constructor* **public static** Singleton getInstance(){  
 *//synchronized her defasında çalışmasın diye objenin create edilip edilmediğini kontrol ediyoruz  
 //çünkü synchronized pahalı bir işlem* **if** (*singleton* == **null**){  
 **synchronized** (Singleton.**class**){  
 **if** (*singleton*==**null**){  
 *singleton* = **new** Singleton();  
 }  
 }  
 }  
 **return** *singleton*;  
 }  
 **void** printTestMessage(){  
 System.***out***.println(**"Singleton olarak çalıştı"**);  
 }  
}

Main

**public static void** main(String[] args) {  
 Singleton s1 = Singleton.*getInstance*();  
 s1.printTestMessage();  
}

**Behavioral - Null Object Design Pattern**

Genellikle yazığımız kodların içinde bir nesnenin null olup olmadığının kontrolünü hepimiz olcukça fazla yapıyoruz .Kendim için söyleyeyim null görünce kodun o kısmında ne zaman hata çıkacak diye beklerim genelde. Örneğin veritabanından herhangi bir nesne çekmeye çalıştığımızda önce gelen nesnenin boş olup olmadığını kontrol ederiz ardından boş değilse ona ait çeşitli işlemler gerçekleştiririz. Kodumuzun içinde bu tür null değerlerini fazlaca kullandığımızda hem gereksiz kod tekrarı ve hem hata eğilimi artar hemde kodumuzun okunulabilirliği oldukça azalır. Bu gibi durumlarda uygulayabileceğimiz çözüm için nesneye yönelik programlama dünyasında oldukça fazla kullanılan Null Object tasarım kalıbını kullanmaktayız

Customer class’ımız için base class;

**abstract class** CustomerBase{  
 **protected** String **name**;  
 **public abstract boolean** isNull();  
 **public abstract** String getName();  
}

Concrete class’lar

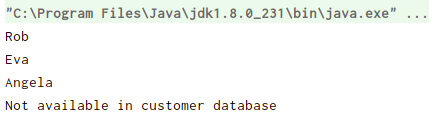
**class** RealCustomer **extends** CustomerBase{  
 **public** RealCustomer(String newName) {  
 **this**.**name** = newName;  
 }  
 @Override  
 **public boolean** isNull() {  
 **return false**;  
 }  
 @Override  
 **public** String getName() {  
 **return name**;  
 }  
}  
**class** NullCustomer **extends** CustomerBase{  
 @Override  
 **public boolean** isNull() {  
 **return true**;  
 }  
 @Override  
 **public** String getName() {  
 **return "Not available in customer database"**;  
 }  
}

Burada null gelebilecek bir customer için NullCustomer adında bir class create ettim ve CustomerBase’den extends ettim.Sürekli null kontrolü yapmaktan kaçınmak için.Boilerplate kod ile 3 tane isim create ettim.

**class** CustomerFactory {  
 **public static final** String[] ***names*** = {**"Rob"**, **"Eva"**,**"Angela"**};  
 **public static** CustomerBase getCustomer(String name){  
 **for** (**int** i = 0; i < ***names***.**length**; i++) {  
 **if** (***names***[i].equalsIgnoreCase(name)){  
 **return new** RealCustomer(name);  
 }  
 }  
 **return new** NullCustomer();  
 }  
}

Main

**public static void** main(String[] args) {  
 CustomerBase customer1 = CustomerFactory.*getCustomer*(**"Rob"**);  
 CustomerBase customer2 = CustomerFactory.*getCustomer*(**"Eva"**);  
 CustomerBase customer3 = CustomerFactory.*getCustomer*(**"Angela"**);  
 CustomerBase customer4 = CustomerFactory.*getCustomer*(**"Luna"**);  
 System.***out***.println(customer1.getName());  
 System.***out***.println(customer2.getName());  
 System.***out***.println(customer3.getName());  
 System.***out***.println(customer4.getName());  
}



Görüldüğü üzere Luna ismi array içerisinde yer almadığı için null nesne olduğu için bizim yazdığımız kodu döndürdü

**Creational - Factory Design Pattern**

Oluşturduğumuz bir interface ya da abstract sınıftan türeterek başka bir sınıf oluşturma işlemine verilen addır Factory Pattern. Araba’nın özelliklerini bildirecek bir yazılım yapacağız. Kullanıcı araba seçecek ancak seçilen arabanın hatchback mi yoksa sedan özellikli olacağını bilmiyoruz ve bu değişkenlik gösterebilir. Kullanıcının seçimine bırakılan bu işlemde kullanıcının her seçiminde kod değişikliği yapmamak için Factory Pattern’e başvurulur. Yada şöyle bir şey düşünelim. Loglama yapan bir interface’imiz ve altında concrete class’ları var. Hangi loglama mekanizmasını seçersek fabrika onu üretecek.

//Logic’e Sahip class’ların inherit edileceği interface

**interface** IServer{  
 *//belli bir sunucu için ağ ile ilgili sorunları giderir* **public void** resolve();  
}

Concrete Class’lar

**class** MailServer **implements** IServer{  
 @Override  
 **public void** resolve() {  
 System.***out***.println(**"Mail server üzerinde ki problem giderildi"**);  
 }  
}  
  
**class** FtpServer **implements** IServer{  
 @Override  
 **public void** resolve() {  
 System.***out***.println(**"Ftp Server üzerinde ki problem giderildi"**);  
 }  
}

Gelen string değere göre dynamicly üretim yapan fabrikamız

**class** ServerFactory{  
 **public static** IServer getServer(String serverType) **throws** Exception {  
 **switch**(serverType){  
 **case "mail"** : **return new** MailServer();  
 **case "ftp"** : **return new** FtpServer();  
 **default**: **throw new** Exception(**"Invalid server type"**);  
 }  
 }  
}

Main

**public static void** main(String[] args) **throws** Exception {  
 Scanner input = **new** Scanner(System.***in***);  
 System.***out***.println(**"Which server do you want to resolve"**);  
 String serverResult = input.nextLine();  
 IServer server = ServerFactory.*getServer*(serverResult);  
 server.resolve();  
}

Problemli olan server’ı kullanıcığın gireceği string değere göre gidip resetleyecek basit bir kod örneği

**Creational - Abstract Factory Design Pattern**

Birden fazla ürün ailesi ile çalışmak durumunda kaldığımızda , ürün ailesi ile istemci tarafını soyutlamak için kullanılır. Factory design pattern’den farkı budur. Soyut fabrika tasarım kalıbının en belirgin özelliği, üretilecek nesnelerin birbirleriyle ilişkili olmasıdır. Oluşturulan bu yapıda üretilen nesnelerin kendisiyle ilgilenmeye gerek yoktur. Diğer bir deyişle üretim sınıfında, üretimin yapılacağı fabrikanın hangi fabrika olduğu veya üretilen nesnelerin hangi tür olduğu ile ilgilenilmez. Bu sayede aynı arayüzü veya soyut sınıfı kullanarak yeni nesneleri kalıba eklemek kolaylaştırılmıştır.  Birçok platformda çalışacak biçimde geliştirilen uygulamalar var. Yani, hem web üzerinden erişebildiğiniz; hem de Android, IOS ya da Windows 10 üzerinde native olarak kullanabildiğiniz uygulamalardan bahsediyorum. Böyle bir uygulamada basit bir ekran varsayalım. Basitçe, iki metin giriş kontrolü ve bir butondan oluşan bir kullanıcı giriş ekranı olsun mesela. Buradaki kontrollerin arayüzü ve özellikleri çalışacağı işletim sistemine bağlıdır değil mi? Android’de başka, IOS’da başka, Windows’da başka, Web’de başka bir buton ama hepsinde bir buton sonuç olarak.O zaman şöyle bir uygulama oluşturucusu yapsak ne iyi olurdu… Biz ekran tasarımını yapsak sonra da desek ki, al bunu şu işletim sistemine göre oluştur. Uygulama da tasarımı alıp, belirttiğimiz işletim sistemine göre sıfırdan yaratsa.

Loglama için abstract class’ımız

**abstract class** LoggingBase{  
 **public abstract void** log(String message);  
}

Loglama için concrete classlarımız

**class** NLogger **extends** LoggingBase{  
 @Override  
 **public void** log(String message) {  
 System.***out***.println(**"Logged with NLogger"**);  
 }  
}  
**class** Log4Net **extends** LoggingBase{  
 @Override  
 **public void** log(String message) {  
 System.***out***.println(**"Logged with Log4Net"**);  
 }  
}

Cacheleme için abstract class’ımız

**abstract class** CachingBase{  
 **public abstract void** Cache(String data);  
}

Cacheleme için concrete classlarımız

**class** RedisCache **extends** CachingBase{  
 @Override  
 **public void** Cache(String data) {  
 System.***out***.println(**"Cached with Redis Cache"**);  
 }  
}  
**class** MemCache **extends** CachingBase{  
 @Override  
 **public void** Cache(String data) {  
 System.***out***.println(**"Cached with MemCache"**);  
 }  
}

Factory’lere hizmet edecek interface’imiz

**interface** FactoryBase {  
 **public** LoggingBase createLogger();  
 **public** CachingBase createCacher();  
}

Factory’lerimiz

**class** FactoryOne **implements** FactoryBase{  
 @Override  
 **public** LoggingBase createLogger() {  
 **return new** Log4Net();  
 }  
 @Override  
 **public** CachingBase createCacher() {  
 **return new** RedisCache();  
 }  
}  
**class** FactoryTwo **implements** FactoryBase{  
 @Override  
 **public** LoggingBase createLogger() {  
 **return new** NLogger();  
 }  
 @Override  
 **public** CachingBase createCacher() {  
 **return new** MemCache();  
 }  
}

Business

**class** ProductManager{  
 **private** FactoryBase **factoryBase**;  
 **private** LoggingBase **loggingBase**;  
 **private** CachingBase **cachingBase**;  
 **public** ProductManager(FactoryBase factoryBase){  
 **this**.**factoryBase** = factoryBase;  
 **this**.**loggingBase** = factoryBase.createLogger();  
 **this**.**cachingBase** = factoryBase.createCacher();  
 }  
 **public void** getAll(){  
 System.***out***.println(**"All product returned"**);  
 **loggingBase**.log(**"Logged"**);  
 **cachingBase**.Cache(**"Cached"**);  
 }  
}

Main

**public static void** main(String[] args) {  
 ProductManager pm = **new** ProductManager(**new** FactoryOne());  
 pm.getAll();  
}

**Creational - Multiton Design Pattern**

Singleton Design pattern’in sadece içerisinde map halinde instance’ları tuttuğu halidir.

**class** Multiton{  
 **private static final** Map<String,Multiton> ***cameras*** = **new** HashMap<>();  
 **private** Multiton(){}  
 **public static** Multiton getInstance(String brand){  
 Multiton result = ***cameras***.get(brand);  
 **if** (result==**null**){  
 **synchronized** (Multiton.**class**){  
 **if** (result==**null**){  
 ***cameras***.put(brand,**new** Multiton());  
 }  
 }  
 }  
 **return *cameras***.get(brand);  
 }  
}

Main

**public static void** main(String[] args) {  
 Multiton m1 = Multiton.*getInstance*(**"Canon"**);  
 Multiton m2 = Multiton.*getInstance*(**"Canon"**);  
 Multiton m3 = Multiton.*getInstance*(**"Nikon"**);  
 Multiton m4 = Multiton.*getInstance*(**"Nikon"**);  
}