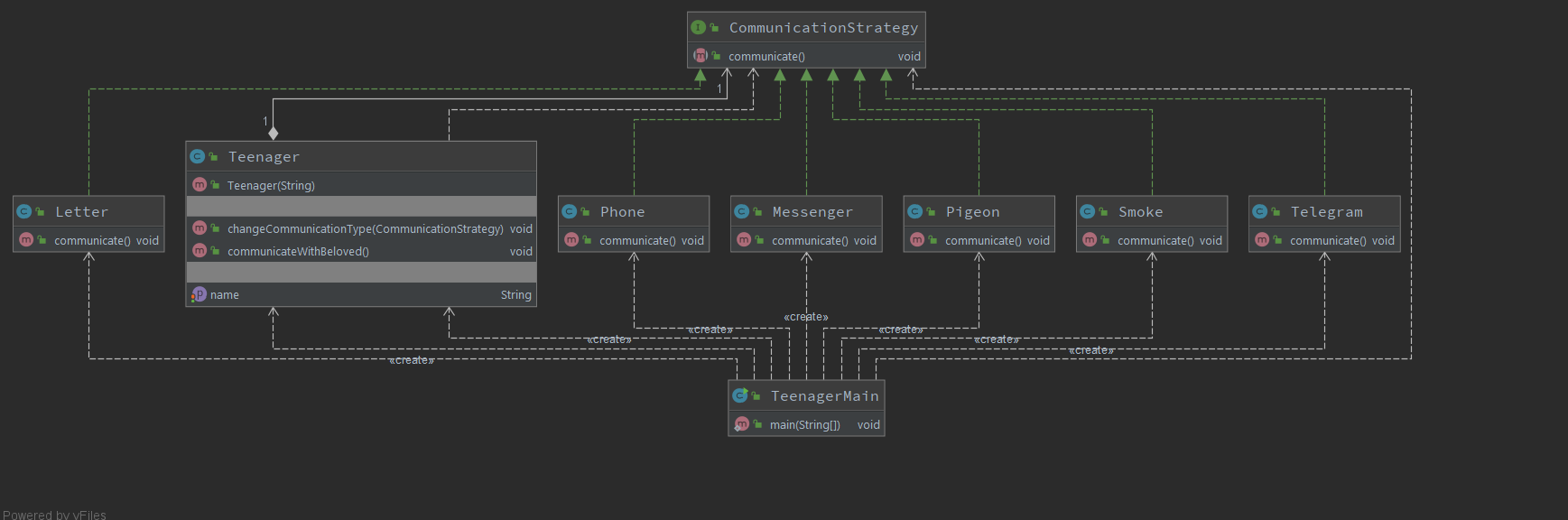
[**https://github.com/ker3mk/Final-DesignPatternsCourse.git**](https://github.com/ker3mk/Final-DesignPatternsCourse.git)

**1-) Strategy Pattern**

**Tanım:**

Aynı arayüz altında, aynı sorunu çözebilecek birçok çözüm yöntemi sınıfını saklayarak, kullanıcı nesnelerin hangi yöntemin kullanıldığından haberdar olmaksızın isteklerininin sağlanmasını olanaklı kılar. Kullanıcı nesneler aynı türden nesnelerle çalıştıklarını varsayarken, farklı davranış biçimleri ile karşılanırlar.

Bir işlem için farklı yöntemlerin uygulanabilir olduğu durumlarda, bu yöntemi kullanacak olan nesne, hangi yöntemin uygulanacağını seçer. Çünkü bu içerik nesnesi, yöntemleri belirleyen üst sınıfı içerir. Farklı yöntem veya strateji alt sınıfları da, bu üst sınıftan türerler. Bu tasarım deseniyle, yöntemin nasıl uygulanması gerektiği ile ilgili detaylar, bu yöntemi kullanacak nesneden ayrılmış olur. Ayrıca bu tasarım deseninin kullanılmasıyla, kod uzun "if/else" veya "switch" ifadelerinden ayıklanır.



package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public interface CommunicationStrategy {  
 void communicate();  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Letter implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with LETTER");  
 }  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Messenger implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with MESSENGER");  
 }  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Phone implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with PHONE");  
 }  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Pigeon implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with PIGEON");  
 }  
}

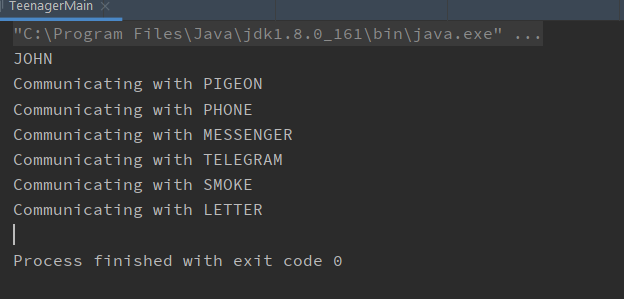
package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Smoke implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with SMOKE");  
 }  
}

package teenager;  
public class Telegram implements CommunicationStrategy {  
 public void communicate() {  
 System.*out*.println("Communicating with TELEGRAM");  
 }  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Teenager {  
 private String name;  
 private CommunicationStrategy communicationType;  
  
 public Teenager(String name) {  
 System.*out*.println(name);  
 this.name = name;  
 }  
  
 public void changeCommunicationType(CommunicationStrategy communicationType) {  
 this.communicationType = communicationType;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
  
 public void communicateWithBeloved() {  
 communicationType.communicate();  
 }  
  
}

package teenager;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class TeenagerMain {  
  
 public static void main(String[] args) {  
 Teenager teenager=new Teenager("JOHN");  
 CommunicationStrategy communicationStrategy=new Pigeon();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
  
 communicationStrategy=new Phone();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
  
 communicationStrategy=new Messenger();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
  
 communicationStrategy=new Telegram();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
  
 communicationStrategy=new Smoke();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
  
  
 communicationStrategy=new Letter();  
 teenager.changeCommunicationType(communicationStrategy);  
 teenager.communicateWithBeloved();  
 }  
}

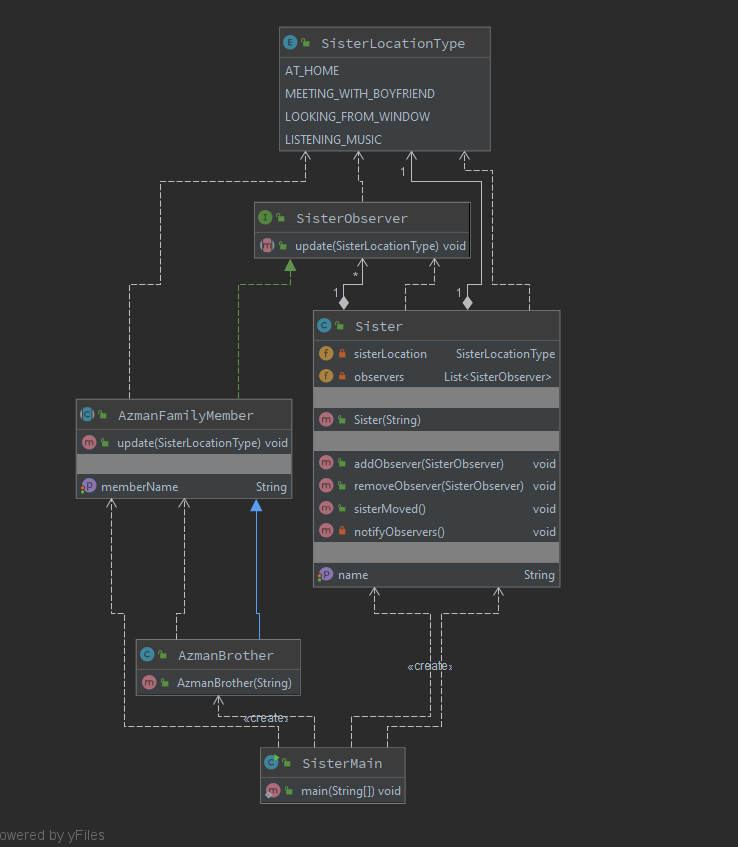
**ÇIKTI**



**2-Observer Pattern**

**Tanım:**

Observer pattern,objeler arasında one-to-many ilişkiye sahip,bir objenin durumu değiştiğinde diğer objelerinde haberdar olmasını sağlayan bir yapıdır.Publisher/Subscriber olarak tanımlanmaktadır.Bir publisher(yayınlayıcı) vardır ve diğer üyeler bu publisher e subscriber olmaktadır. İsmindende anlaşılacağı gibi nesneyi gözlemleme işlemi yapar herhangi bir durum değişikliğinde yazılan işlemi tetikler. Örneğin sosyal medyada herhangi bir arkadaşınız bir durum veya fotoğraf paylaştığında kullanıcılara bildirim gelmesi veya fiyatı indirime girdiği anda kullanıcılara haber verilmesi gibi durumları bu tasarım deseni ile modelleyebiliriz.



package sister;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class AzmanBrother extends AzmanFamilyMember {  
  
 public AzmanBrother(String memberName) {  
 setMemberName(memberName);  
 }  
}

package sister;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public abstract class AzmanFamilyMember implements SisterObserver {  
 private String memberName;  
 public void update(SisterLocationType sisterLocationType) {  
 System.*out*.println( "Azman Family Member :"+memberName+" is know Sister Location whic is "+sisterLocationType);  
 }  
  
 public String getMemberName() {  
 return memberName;  
 }  
  
 public void setMemberName(String memberName) {  
 this.memberName = memberName;  
 }  
}

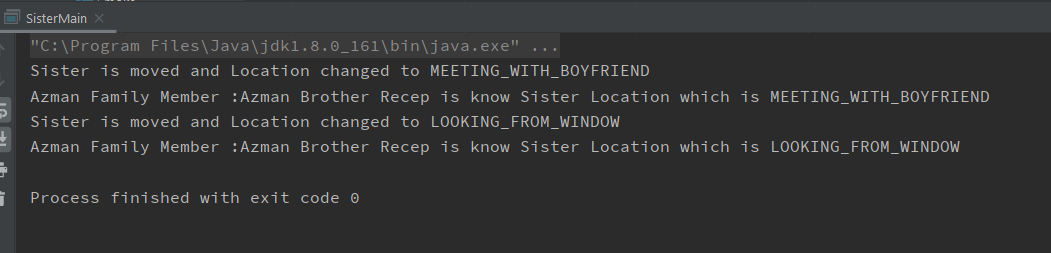
package sister;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public enum SisterLocationType {  
 *AT\_HOME*,*MEETING\_WITH\_BOYFRIEND*,*LOOKING\_FROM\_WINDOW*,*LISTENING\_MUSIC*}

package sister;  
  
import teenager.CommunicationStrategy;  
  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Sister {  
 private String name;  
 private SisterLocationType sisterLocation;  
 private List<SisterObserver> observers;  
  
 public Sister(String name) {  
 this.name=name;  
 observers = new ArrayList<SisterObserver>();  
 sisterLocation = SisterLocationType.*AT\_HOME*;  
 }  
 public void addObserver(SisterObserver obs) {  
 observers.add(obs);  
 }  
  
 public void removeObserver(SisterObserver obs) {  
 observers.remove(obs);  
 }  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void sisterMoved() {  
 SisterLocationType[] enumValues = SisterLocationType.*values*();  
 sisterLocation = enumValues[(sisterLocation.ordinal() + 1) % enumValues.length];  
 System.*out*.println("Sister is moved and Location changed to "+sisterLocation );  
 notifyObservers();  
 }  
  
 private void notifyObservers() {  
 for (SisterObserver obs : observers) {  
 obs.update(sisterLocation);  
 }  
 }  
  
}

package sister;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public interface SisterObserver {  
  
 void update(SisterLocationType sisterLocation);  
  
}

package sister;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class SisterMain {  
  
 public static void main(String[] args) {  
 AzmanFamilyMember azmanFamilyMember=new AzmanBrother("Azman Brother Recep");  
 Sister sister=new Sister("Ayşe");  
 sister.addObserver(azmanFamilyMember);  
 sister.sisterMoved();  
 sister.sisterMoved();  
 }  
  
  
}

**ÇIKTI**



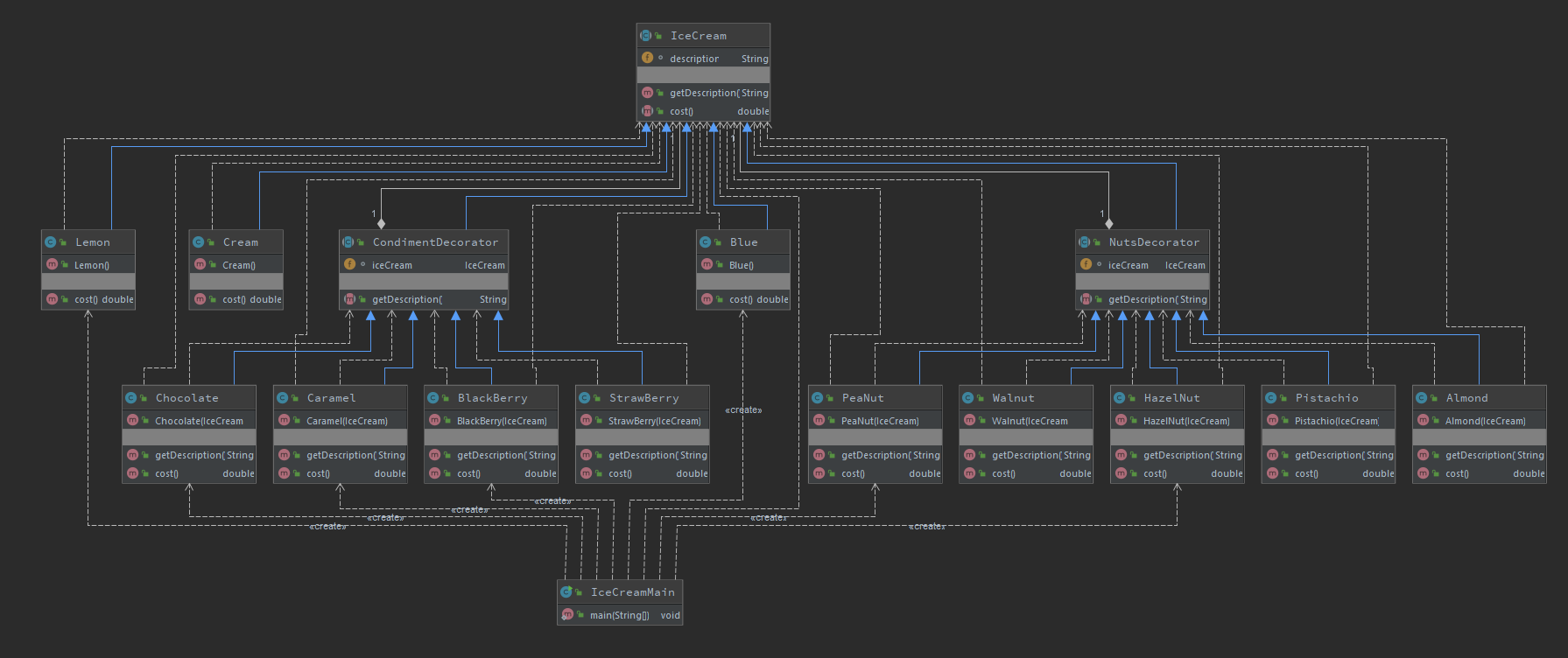
**3.Decorator Pattern**

Mevcut bir sınıf hiyerarşisini ya da sınıfın yapısını değiştirmeden, oluşturulan nesnelere yeni özelliklerin eklenme işlemini gerçekleştirmek için Decorator tasarım şablonu kullanılır.

Alt sınıfların oluşturulması yöntemiyle, sınıflara yeni özelliklerin eklenmesi, daha sonra sisteme eklenecek alt sınıflar için değiştirilmesi zor kalıpların oluşmasını beraberinde getirir. Bu durumda, üst sınıflarda tanımlanmış olan bazı özellikler statik ve alt sınıflar için değistirilemez ya da kullanımı engellenemez bir hal alabilir. Kullanıcı sınıflar içinde bu sorun teşkil edebilir, çünkü kendi istekleri doğrultusunda bir nesnenin ne zaman ve nasıl oluşturulması gerektiğini yönlendiremeyebilirler.

Nesnelere, sahip oldukları sınıfların yapılarının değistirilmeden yeni özelliklerin eklenmesini sağlayan Decorator tasarım şablonu ile, istenilen özelliklerin ekleneceği nesne başka bir nesne içine gömülür. Yeni özellik eklenen nesneyi içine alan nesneye dekoratör ismi verilir. Decorator nesnesi ile yeni özellik eklenen nesne aynı üst sınıfa dahil olduklarından, birbirleriyle değiştirilebilir haldedirler. Bu özellikten dolayı kullanıcı sınıf, dekoratör sınıf ile dekoratör nesne bünyesinde bulunan diğer nesne arasında ayrım yapmaz

Decorator tasarım deseni, structural tasarım desenlerinden biridir. Bir nesneye dinamik olarak yeni özellikler eklemek için kullanılır. Kalıtım kullanmadan da bir nesnenin görevlerini artırabileceğimizi gösterir.



package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public abstract class IceCream {  
 String description = "Unknown IceCream";  
  
 public String getDescription() {  
 return description;  
 }  
  
 public abstract double cost();  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public abstract class CondimentDecorator extends IceCream {  
 IceCream iceCream;  
 public abstract String getDescription();  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public abstract class NutsDecorator extends IceCream {  
 IceCream iceCream;  
 public abstract String getDescription();  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Blue extends IceCream {  
 public Blue() {  
 description = "Blue";  
 }  
  
 public double cost() {  
 return 1.75;  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Lemon extends IceCream {  
 public Lemon() {  
 description = "Lemon";  
 }  
  
 public double cost() {  
 return 2.0;  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Cream extends IceCream {  
 public Cream() {  
 description = "Cream";  
 }  
  
 public double cost() {  
 return 1.50;  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Caramel extends CondimentDecorator {  
 public Caramel(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", Caramel";  
 }  
  
 public double cost() {  
 return .25 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Chocolate extends CondimentDecorator {  
 public Chocolate(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", Chocolate";  
 }  
  
 public double cost() {  
 return .35 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class StrawBerry extends CondimentDecorator {  
 public StrawBerry(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", BlackBerry";  
 }  
  
 public double cost() {  
 return .30 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class BlackBerry extends CondimentDecorator {  
 public BlackBerry(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", StrawBerry";  
 }  
  
 public double cost() {  
 return .30 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class HazelNut extends NutsDecorator {  
 public HazelNut(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", +HazelNut";  
 }  
  
 public double cost() {  
 return .10 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class PeaNut extends NutsDecorator {  
 public PeaNut(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", +PeaNut";  
 }  
  
 public double cost() {  
 return .15 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Pistachio extends NutsDecorator {  
 public Pistachio(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", +Walnut";  
 }  
  
 public double cost() {  
 return .11 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Walnut extends NutsDecorator {  
 public Walnut(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", +Walnut";  
 }  
  
 public double cost() {  
 return .09 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class Almond extends NutsDecorator {  
 public Almond(IceCream iceCream) {  
 this.iceCream = iceCream;  
 }  
  
 public String getDescription() {  
 return iceCream.getDescription() + ", +Almond";  
 }  
  
 public double cost() {  
 return .16 + iceCream.cost();  
 }  
}

package icecream;  
  
*/\*\*  
 \** ***@author*** *Kerem  
 \* 6/15/2020  
 \*/*public class IceCreamMain {  
 public static void main(String[] args) {  
 IceCream iceCream=new Lemon();  
 System.*out*.println(iceCream.getDescription()  
 + " $" + iceCream.cost());  
  
 IceCream iceCream2 = new Blue();  
 iceCream2 = new Chocolate(iceCream2);  
 iceCream2 = new Caramel(iceCream2);  
 iceCream2 = new BlackBerry(iceCream2);  
 iceCream2 = new HazelNut(iceCream2);  
 iceCream2 = new PeaNut(iceCream2);  
  
 System.*out*.println(iceCream2.getDescription()  
 + " $" + iceCream2.cost());  
  
 }  
}

**ÇIKTI**

