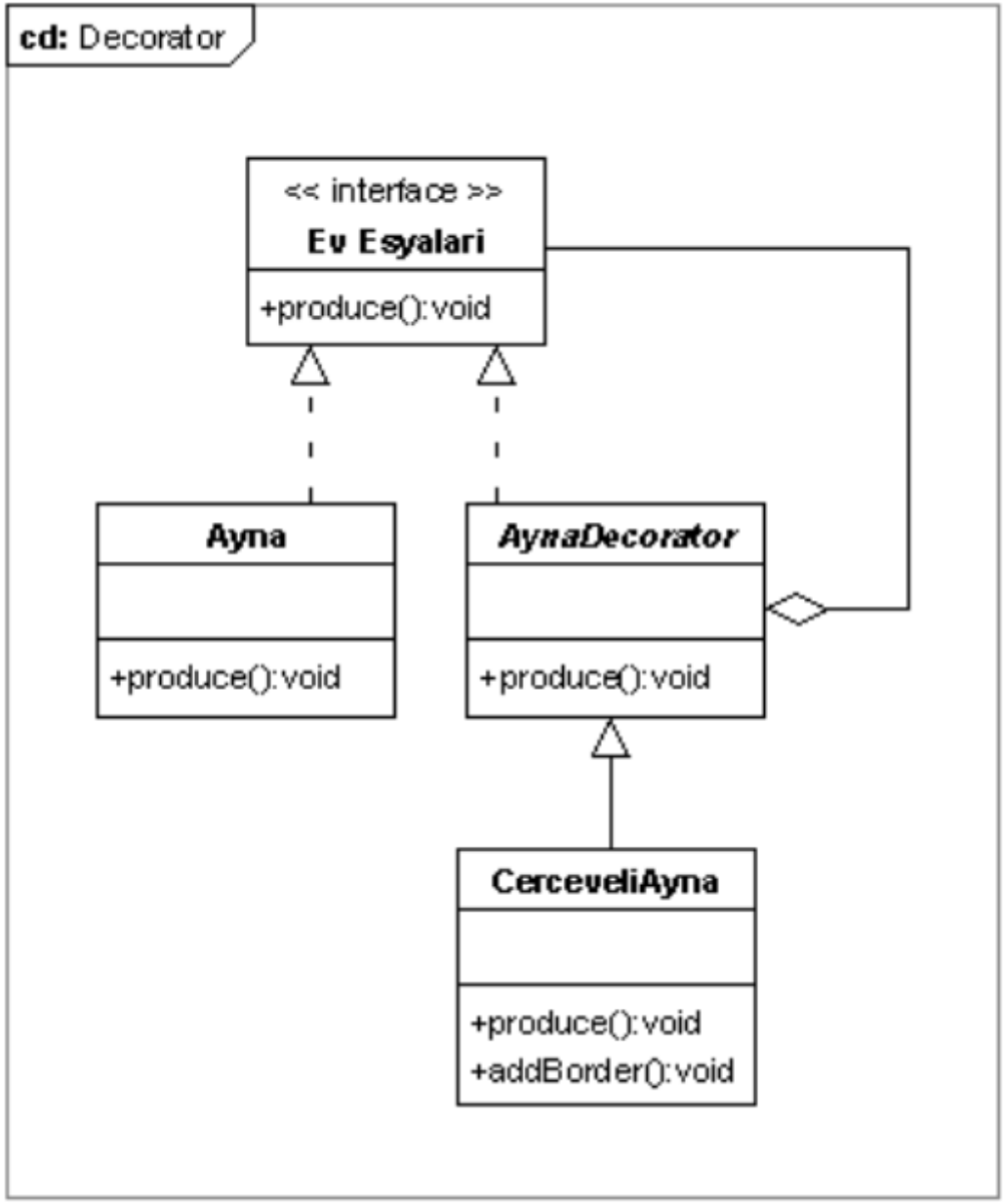
Dekoratör

Yapısal Tasarım Şablonları -
Structural Patterns

- Mevcut bir sınıf hiyerarşisini ya da sınıfın yapısını değiştirmeden, oluşturulan nesnelere yeni özelliklerin eklenme işlemini gerçekleştirmek için dekoratör tasarım şablonu kullanılmaktadır.
- Altsınıfların oluşturulması yöntemiyle sınıflara yeni özelliklerin eklenmesi, daha sonra sisteme eklenecek altsınıflar için değiştirilmesi zor kalıpların oluşmasını beraberinde getirmektedir. Bu durumda üstsınıflarda tanımlanmış olan bazı özellikler statik ve altsınıflar için değistirilemez ya da kullanımı engellenemez bir hal alabilir. Kullanıcı sınıflar için de bu durum sorun teşkil edebilmektedir, çünkü kendi istekleri doğrultusunda bir nesnenin ne zaman ve nasıl oluşturulması gerektiğini yönlendiremeyebilirler

Nesnelere sahip oldukları sınıfların yapılarının değistirilmeden yeni özelliklerin eklenmesini sağlayan dekoratör tasarım şablonu ile istenilen özelliklerin ekleneceği nesne başka bir nesne içine gömülür.

Yeni özellik eklenen nesneyi içine alan nesneye dekoratör ismi verilir. Dekoratör nesnesi ile yeni özellik eklenen nesne aynı üstsınıfa dahil olduklarından, birbirleriyle değiştirilebilir haldedirler. Bu özellikten dolayı kullanıcı sınıf, dekoratör sınıf ile dekoratör nesne bünyesinde bulunan diğer nesne arasında ayrım yapmaz. Nesneler arası ilişkiye aşagıda yer alan UML diyagramında görmekteyiz.


cd: Decorator
<< interface >>
Ev Esyalari
+produce():void
Ayna
+produce():void
AynaDecorator
+produce():void
CerceveliAyna
+produce():void
+addBorder():void

- public interface EvEsyalari {
  //Üretimi gerçekleştirmek için kullanılan metot.
  public void produce();
  }

- public class Ayna implements EvEsyalari {

```
        @Override
        public void produce() {
        System.out.println("Ayna imal edildi.");
        }
}
```

```
• public abstract class AynaDecorator implements EvEsyalari {
  /*
  * Bünyesinde mevcut bir ayna nesnesi bulundurur ve değişik metotlar
  * kullanarak bu ayna nesnesini dekore eder.
  */
     private EvEsyalari ayna = new Ayna();
     public EvEsyalari getAyna()
          {
          return ayna;
          }
  public void setAyna(final EvEsyalari ayna)
          {
          this.ayna = ayna;
          }
}
```

AynaDecorator sınıfı nesne katma (aggregation) yöntemiyle bünyesinde bir Ayna nesnesi tutmaktadır. Dekorasyonu, yani aynaya bir çerçeve ekleme işlemini gerçeklestirmebilmek için
Ayna nesnesi ile böyle bir bağ oluşturulması gerekmektedir. Ayna nın üretilme işlemi Ayna.produce() metoduna delege edildikten sonra, dekorasyon için gerekli metot, yani aynaya çerçeve takma işlemi AynaDecorator ya da bir altsınıfı tarafından kullanılacaktır.

- AynaDecorator sınıfını soyut olarak tanımlıyoruz. Amacımız değişik türdeki aynaları üretebilmek için esnek bir model oluşturabilmektir. Bugün çerçeveli aynalar üretimi için kullanılan programımız, yarın yeni bir altsınıf oluşturularak, başka türde bir aynanın üretimine izin verebilecek yapıda olmalıdır. Bu yüzden AynaDecorator isminde bir soyut sınıf tanımlıyarak, gelecekte üretimi yapılacak tüm aynalı ürünler için bir baz oluşturmuş oluyoruz. Çerçeveli aynaların üretimi için CerceveliAyna sınıfını oluşturuyoruz.

```
public class CerceveliAyna extends AynaDecorator {

/**
* Üretim için kullanılan sınıf. addBorder metodu ile aynaya cerceve ekler.
*/
        @Override
        public void produce() {
                getAyna().produce();
                addBorder();
        }
/
**
* Cerceve ekleme işlemini gerçekleştirmek için kullanılan metot.
* *
/
        public void addBorder()
        {
        System.out.println("Aynaya cerceve eklendi.");

        }
}
```

- CerceveliAyna sınıfı, ayna üretim işlemini üstsınıfı AynaDecorator üzerinden gerçekleştirmektedir.
  AynaDecorator sınıfında ayna isminde bir sınıf değişkeni olduğu için CerceveliAyna sınıfı getAyna() metodu ile üretilen aynaya ulaşabilmektedir.
  Üretimi yapılmış bir aynaya çerçeve takmak için addBorder() metodu tanımlanmıştır.
  CerveveliAyna.produce() metodu içinde önce ayna üretilmekte, daha sonra addBorder() metodu ile bu aynaya bir çerçeve takılmaktadır yani ayna dekore edilmiş olmaktadır.
  Sonuç itibariyle aynanın ve aynalı ürünlerin üretimini birbirinden ayırmış olduk ve mevcut bir nesneye (Ayna), sahip olduğu sınıfın yapısını değiştirmeden yeni bir özellik (çerçeve) ekleyebildik.
  Aşağıda yer alan Test sınıfı ile çerçeveli ayna üretimine başlayabiliriz:

```
public class Test {
public static void main(final String[] args) {
final EvEsyalari ayna = new CerceveliAyna();
ayna.produce();
}
}
```

https://www.onlinegdb.com/edit/BkD-U4gI_

https://www.onlinegdb.com/edit/HyAE5seIu

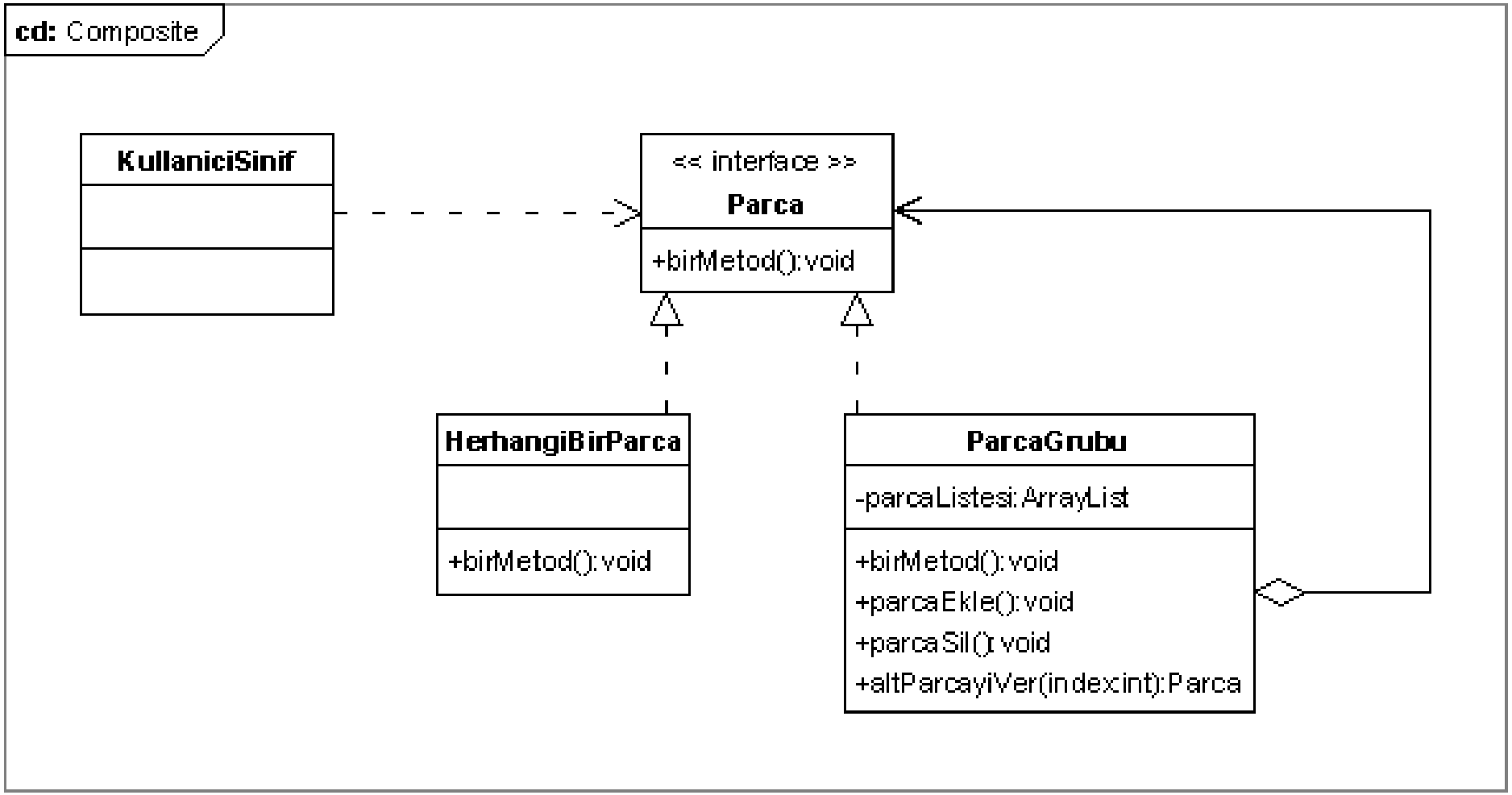
Bileşik (Composite)
Design Pattern

Yapısal Tasarım Şablonları -
Structural Patterns

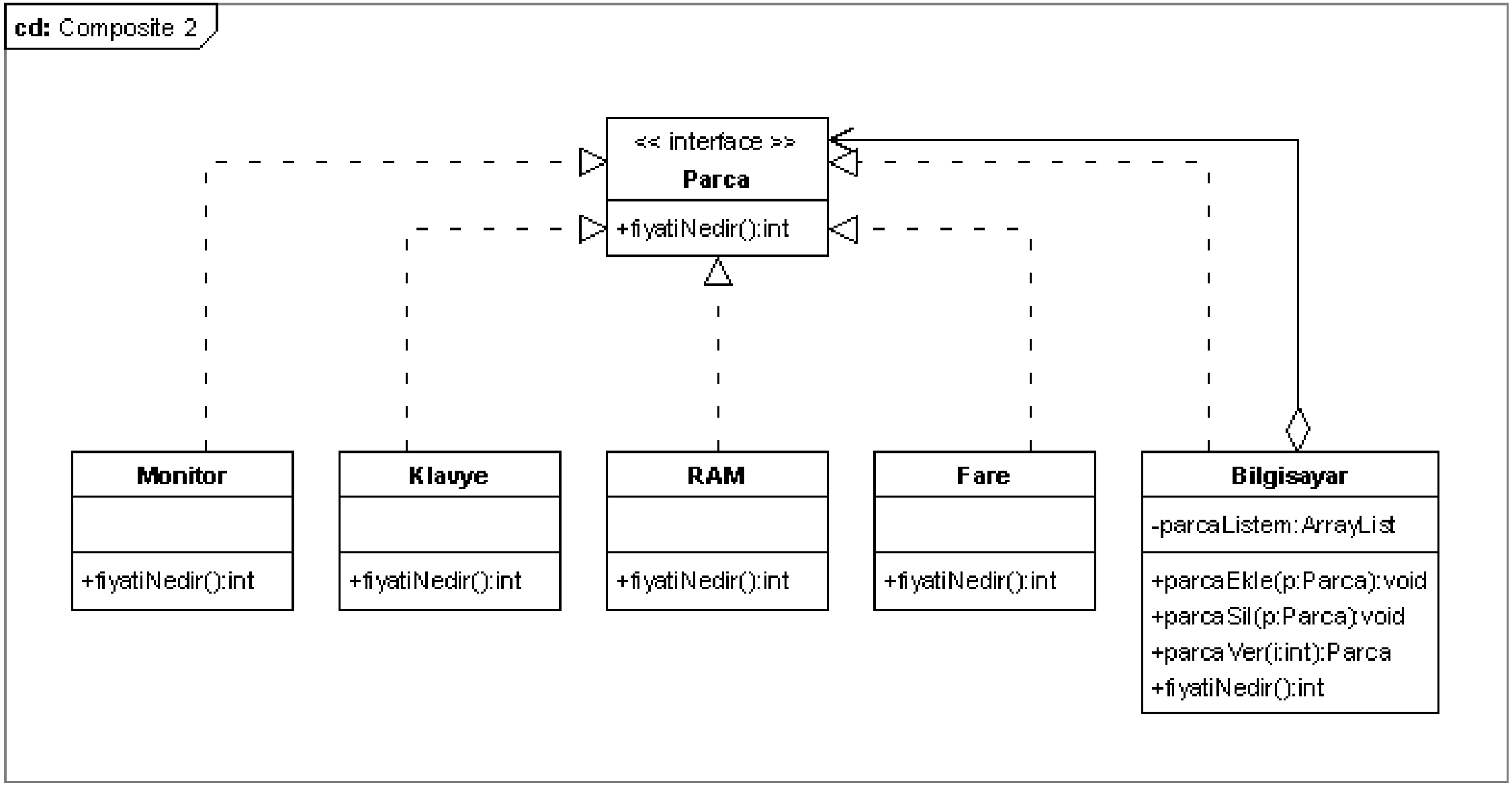
Bileşik tasarım şablonu bir sistemin bütünü ve parçaları arasındaki ilişkileri modellemek için kullanılmaktadır.

Sistemin bütününü oluşturan parçalar, kendi içlerinde alt parçalardan oluşabilir.
Bileşik tasarım şablonu kullanıcı sınıfın, sistem, sistemin parçaları ve alt parçalar arasında ayrım yapmadan nesneleri kullanmasına izin vermektedir.

Bu şekilde sistem yazılımı ve kullanımı daha sadeleştirilmektedir.


cd: Composite
KullaniciSinif
<< interface >>
Parca
+birMetod():void
HerhangiBirParca
+birMetod():void
ParcaGrubu
-parcaListesi:ArrayList
+birMetod():void
+parcaEkle():void
+parcaSil():void
+altParcayiVer(indexint):Parca

Sistemin bütününü oluşturan nesneler Parca interface sınıfını implemente ederler. Kendi bünyesinde alt parçalardan oluşan parçalar (ParcaGrubu) Parca interface sınıfını implemente etmekle beraber, bünyelerinde barındırdıkları alt parçalar için bir liste (ArrayList) tanımlarlar.
Bunun yanı sıra bu liste üzerinde ekleme, silme ve parça edinme işlemleri için metotlara sahiptirler. Kullanıcı sınıf bir parça grubu ve sistemin herhangi bir parçası arasında ayrım yapmaz, çünkü sistemin tüm parçaları Parca interface sınıfını implemente ederler.
Bilgisayar ve bilgisayar parçaları satımı yapan bir mağaza için bileşik tasarım şablonunu kullanarak, aşağıda yer alan modeli oluşturuyoruz:


cd: Composite 2
<< interface >>
Parca
+fiyatiNedir():int
Monitor
+fiyatiNedir():int
Klavye
+fiyatiNedir():int
RAM
+fiyatiNedir():int
Fare
+fiyatiNedir():int
Bilgisayar
-parcaListem:ArrayList
+parcaEkle(p:Parca):void
+parcaSil(p:Parca):void
+parcaVer(i:int):Parca
+fiyatiNedir():int

Verilen bir siparişi sistemin bütünü olarak düşünürsek, sipariş içinde bulunan bir monitör ve
bilgisayar siparişin parçalarıdır. Monitörün tek bir parça olduğunu farz ediyoruz.

Bir bilgisayarRAM, anakart (mainboard), fare, klavye ve grafik kartı gibi değişik alt parçalardan oluşmaktadır.
Bilgisayarın kendisi ve parçaları, Parca interface sınıfını implemente ederek birer parça haline gelmektedirler. Bilgayar ile sahip olduğu parçaların arasındaki tek fark, bilgisayar nesnesinin sahip olduğu parçaları tutabileceği liste ve metotlara sahip olmasıdır. Bunun haricinde kullanıcı sınıf açısından bakıldığında bir bilgisayar ve grafik kartı arasında bir farklılık yoktur, çünkü ikisi de Parca interface sınıfını implemente etmektedirler. Böyle bir farkın olmaması, kullanıcı sınıf için
yazılan kodun daha sade bir hal almasını sağlamaktadır, çünkü kullanıcı sınıf if ve else yapıları ile önce önündeki parçanın hangi sınıftan olduğunu tespit etmek zorunda kalmamaktadır.

```java
// Kod 25

package com.pratikprogramci.designpatterns.bolum6.composite;

public interface Parca {

    /**
     * Her altsınıf bu metodu implemente ederek, parçanın
     * fiyatını belirler.
     */
    public int fiyatiNedir();

}
```

```java
public class Monitor implements Parca {

    @Override
    public int fiyatiNedir() {
        return 250;
    }
}

package com.pratikprogramci.designpatterns.bolum6.composite;

public class Klavye implements Parca {

    @Override
    public int fiyatiNedir() {
        return 50;
    }
}
```

Bilgisayar.java ve main sınıfı

https://onlinegdb.com/BJ9AFi0wu

Bilgisayar sınıfı Parca interface sınıfını implemente ettiği için öncelikle bir parçadır.

Bünyesinde barındırdığı parçaların yer aldığı bir listeye sahip olduğu için bir bileşik (composite) nesne halini alır. Bu liste içinde Parca interface sınıfını implemente eden diğer parçalar yer almaktadır .

Bilgisayar sınıfında da fiyatiNedir() metodunu implemente etmemiz gerekiyor. Bu metot içinde bilgisayar nesnesinin parcaListem (ArrayList) değişkeninde yer alan tüm parçaların fiyatiNedir() metodu kullanılarak, bilgisayarın toplam fiyatı tespit edilmiş olur. Kullanıcı sınıf açısından bakıldığında, Parca interface sınıfında yer alan fiyatiNedir() metodu hem normal bir parça nesnesinde hem de bilgisayar gibi bir bileşik nesne üzerinde kullanılabilmektedir. Kullanıcı sınıf normal bir parça nesne ile bileşik nesne arasında ayrım yapmak zorunda kalmaz.

Kaynak ve örnekler

Design patterns – Özcan ACAR

http://www.tasarimdesenleri.com/

https://www.javatpoint.com/decorator-pattern

https://www.youtube.com/watch?v=Op97BRovYtA

https://www.youtube.com/watch?v=21NCVsu41I4