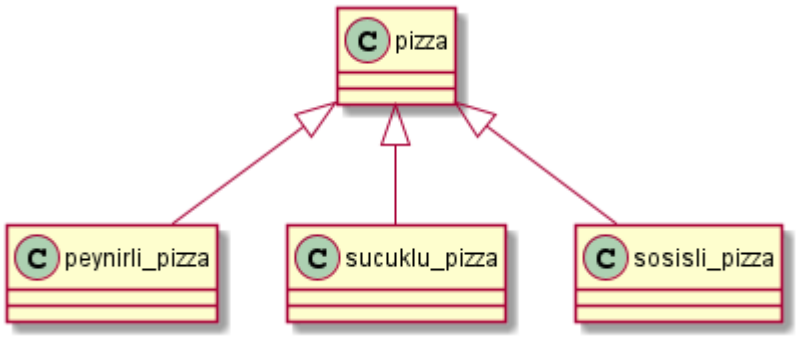
Decorator Pattern

Tayyip Mert Denizgez 160201036 Örgün Eğitim

Problem

Bir pizza oluşturma kütüphanesi yazdığımızı varsayalım. Bu kütüphane birkaç adet özellik ve bir adet oluştur() metoduna sahip olacaktır. Kullanıcı pizza istediğinde bu oluştur metoduyla kullanıcıya bir adet pizza oluşturabilirsiniz.

Daha sonrasında kullanıcıların farklı tipte pizzalar istediğini gördünüz ve bu problemi çözmek için ise değişik pizza türlerini bir pizza sınıfından türetmeyi düşündünüz ve şöyle bir yapı kurdunuz.



Peki ya kullanıcı aynı anda hem sucuklu hem de sosisli pizza isterse ne yapacağız ?

Bu tarz alternatif senaryoların her biri için sonradan alt sınıf yazmak her ne kadar programın çalışmasına engel olmasa da efektif bir çözüm oluşturmamaktadır.

Çözüm

Kalıtım oluşturmanın şunlar gibi bazı promlemleri vardır:

- Kalıtım statiktir. Uygulama çalışırken üstünde değişiklik yapamazsınız.
- Bir çok programlama dilinde çoklu kalıtım bulunmamaktadır. Bu sebeple alt sınıflar birden fazla davranışı kalıtım yoluyla elde edememektedir.

Decorator pattern sayesinde bu sorunların üstesinden gelebilmekteyiz. Wrapper olarak kabul edebileceğimiz ana bir nesne ile hedef nesneyi sarabilliriz. Wrapper nesnesi hedef olan nesne ile aynı interface'i implement ettiği için aynı metotları içermektedir. Bu sayede hedef objenin tüm isteklerini ilgili yerlere iletebilmektedir. Aynı interface'i implement ettikleri için Client tarafında iki nesne aynı olarak görülmektedir. Bu sayede wrapper referansı o interface'i implement eden tüm sınıflardan nesne alabilecektir.

```
public interface Pizza {
    public String getMalzemeler();
    public void setMalzelemeler(String malzeme);
    public double getFiyat();
    public void setFiyat(double fiyat);
}
```

Örneğimize bakacak olursak her pizza türü için ayrı alt sınıf oluşturmak yerine Pizza interface'ini implement eden abstract bir base decorator ve bu base decoratorı extend eden diğer decoratorlar oluşturulabilir.

Base Decorator

```
public abstract class MalzemeDecorator implements Pizza{

    private Pizza obj;
    protected double fiyat;

    public  MalzemeDecorator(Pizza obj){   ①
        this.obj = obj;
    }
    public String getMalzemeler(){}
    public double getFiyat(){}
    public void setMalzelemeler(String malzeme){}
    public void setFiyat(double fiyat) {}

}
```

① Pizza interface'in tipinde değişken alarak polymorphisimden de yararlanarak o interface'i implement eden her nesneyi kabul etmektedir.

Misir Decoratorı

```
public class Misir extends MalzemeDecorator{ ①
    public Misir(Pizza obj){ ②
        super(obj); ③
        System.out.println("Misir Eklendi.");
        setMalzemeler();
        setFiyat();
        this.fiyat = 5;
    }
    public void setMalzemeler(){}
    public void setFiyat(){}
}
```

① Base Decorator sınıf olan Abstract MalzemeDecorator Sıfını extend ediyor.

② Base Decoratorda olduğu gibi pizza interface'i tipinde nesne kabul ediyor.

③ Öncelikle gelen nesneyi ana sınıfın constructorını gönderiyor.

Pizza Oluştur

Temel pizza sınıfıdır. Decorator sınıflar bu sınıftan üretilen nesnenin üzerinde işlem yapmaktadır.

```
public class PizzaOlustur implements Pizza{
    private List<String> malzelemeler = new ArrayList<>(); ①
    private double fiyat = 0; ②

    public void setMalzelemeler(String malzeme){}
    public String getMalzemeler(){}
    public double getFiyat(){}
    public void setFiyat(double fiyat) {}
}
```

① Eklenen malzemeleri tutan liste

② Her malzeme eklendiğinde o malzemenin fiyatıyla artan toplam pizza fiyatını veren değişken

Uygulama

```
public class Uygulama {

    public static void main(String[] args) {
        Pizza p1 = new Misir(new Peynir(new HamurTipi(new PizzaOlustur(),"Normal"))); ①

        System.out.println(p1.getMalzemeler().toString());
        System.out.println(p1.getFiyat());

        Pizza p2 = new Misir(new Peynir(new PizzaOlustur())); ②

        System.out.println(p2.getMalzemeler().toString());
        System.out.println(p2.getFiyat());
    }
}
```

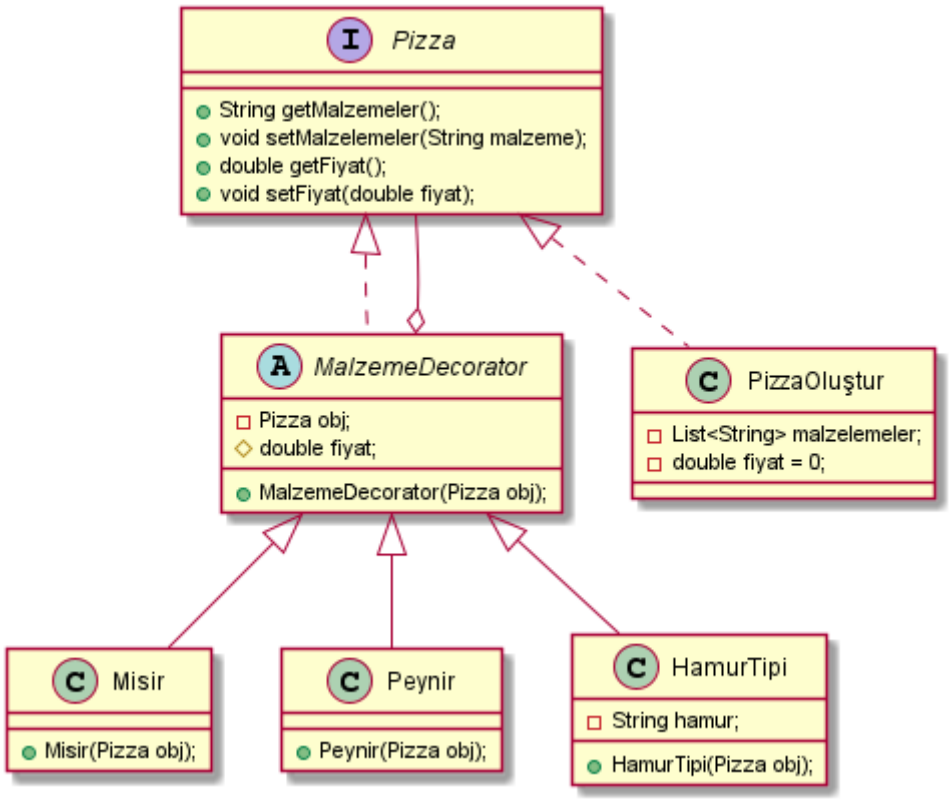
① Temel pizza nesnesine hamur tipi peynir ve mısır decoratorları kullanılarak ekstra özellikler eklendi.

② Bu satırda ise p2 nesnesine sadece peynir ve mısır özellikleri eklendi.

1 ve 2 ile numaralandırılmış satırlar uygulama çalışırken o özelliklere sahip nesneleri oluşturacaktır. Böylece dinamik olarak bir nesneye özellik eklenmiş olacaktır.

Ayrıca burada Mısır peynir gibi decoratorların öncelik sırası bulunmamaktadır.

Bu işlemler sonucunda yapının uml diyagramı şu şekilde olacaktır:



Bu yapı sayesinde yeni pizza türleri dinamik olarak runtime'da oluşturulabilecektir.

Uygulanabilirlik

- Çalışma zamanında bir objeye davranış eklenmesi gerektiğinde kullanılabilir.
- Kalıtım yoluyla davranış aktarılması uygun ya da efektif olmadığında kullanılabilir.

Avantajları

- Alt sınıf oluşturmadan davranış aktarımı yapılabilir.
- Çalışma zamanında davranış eklenip çıkartılabilir.
- Birden fazla decorator kullanarak bir nesneye birden fazla davranış eklenebilir.
- Single Responsibility Principle ı gerçekleştirir. Her özellik için farklı küçük sınıflar açılmaktadır.

Dezavantajları

- Davranışlar bir yığın olarak eklendiğinde belli bir davranışı çıkarmak maliyetli olabilir.(Üst üste giyilen ceketlerden ortadaki bir ceketi çıkartmak gibi.)
- Kodun okunabilirliği azalabilir.