Tarih:08.01.2022

NESNEYE DAYALI PROGRAMLAMA

PROJE

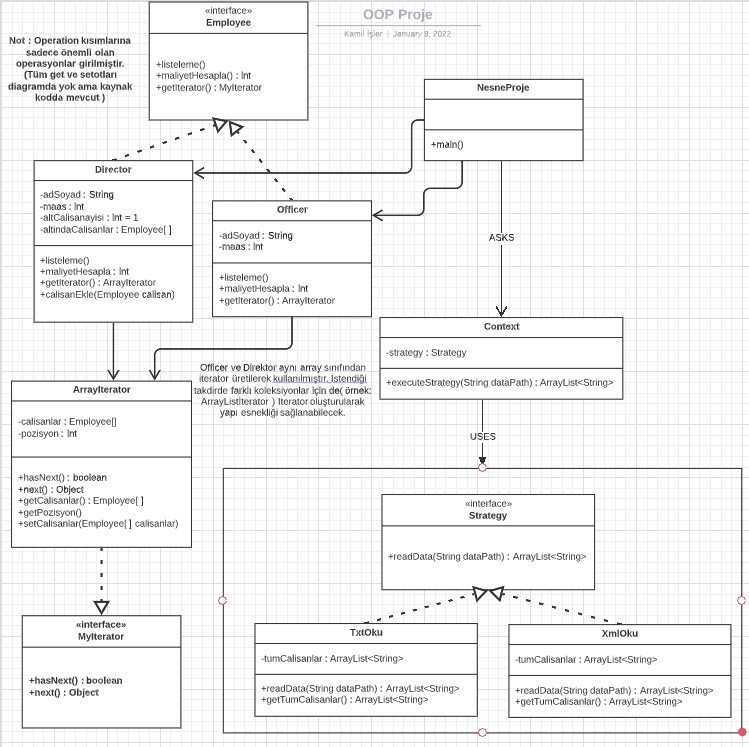
**Nesneye Dayalı Programlamanın Temel Yapıları, Tasarım Desenleri ve JUnit Kullanımı**

Samet ÇEREZCİ 05180000045

Kamil İŞLER 05180000050



**UML Sınıf Diagramı**



**Gereksinimler:**

Bir firmanın çalışanlarına ilişkin hiyerarşik yapısını, composite design pattern kullanarak oluşturduk.

Firmada, “Direktör” ve “Memur” çalışanların olduğu varsayılmaktadır. Bir direktörün emrinde sıfır veya N tane çalışan (memur veya direktör) olabilmektedir. Çalışanlar için String tipinde adSoyad ve tamsayı (int) tipinde maaş verileri bulunmaktadır.

Bizden oluşturulması istenen iki tane işlem bulunmaktadır.Bunlar birincisi,bir direktörün altında çalışanları bulup hiyerarşik yapıda yazdırılması,ikincisi ise bir çalışanın maliyetini yani direktör ise kendi ve altında çalışanların maaş toplamı,memur ise kendi maaşını hesaplamasıdır.

**Sınıf Açıklamaları**

Employee interface’imiz listeleme,maliyet hesaplama fonksiyonlarını ve iterator design pattern için alt sınıflarının gerçekleştireceği getIterator fonksiyonun barındırıyor.(Employee interface=Component)

Director(Director class=Composite) ve Officer(Officer class=Leaf) classları Employee’yi interface ediyor ve hiyerarşik yapıyla istenen işlemleri doğru bir şekilde gerçekleştirecek gerekli interface fonksiyonlarını dolduruyor. Kaynak kod yorum satırlarında açıklandığı gibi Director classı altında çalışanları Employee array’inde tutuyor ve ilk eleman olarak kendisini saydığı için default value olarak altCalisanSayisi = 1 olarak oluşturuluyor. Maliyet hesaplama ve altındaki çalışanları listeleme işlemleri ise recursive olarak yapılıyor ( Kaynak kodda comment olarak açıklamalar mevcut.)

Aynı zamanda Director ve Officer yapımız MyIterator arayüzünü implement eden ArrayIterator sınıfını kullanarak koleksiyonlarında dolaşma işlemlerini gerçekleştiriyor. ArrayIterator sınıfı ihtiyaç böyle olduğu için içindeki Employee’leri “Array” yapısı içinde tutuyor ( verileri farklı yapılarda tutma ihtiyacı oluşursa bu yapı MyIterator interface’imizden başka koleksiyon döndüren bir Iterator sınıfı eklememizi sağlar ve bu sorun çözülür. Yani Iterator Design Pattern’ımız esnekliğini sağlamış olur. )

Strategy Design Pattern Açıklama:

İlk sayfadaki UML diagrama bakınca kare içine aldığımızı görebileceğiniz Strategy isimli bir interface’imiz ve onu implement eden TxtOku, XmlOku classları var . Bu alt classlar kullanıcıdan gelen verilerin okuma işlemini gerçekleştirmek için farklı algoritmalar kullanıyorlar. (TxtOku txt dosyası için splitle ayırma vs işlemlerini yapıyor , Xml okuma ise Xml okuma işlemlerini yaptığı varsayılıyor, dat ,json veri tipi gibi veri tipleri için de ayrı classlar ve algoritmalar oluşturulabilir..) Bu Strategy Design Pattern’ı bize farklı veritabanı veya farklı ortamlardan, farklı dosya çeşidiyle veri geldiğinde o sınıfın davranışının yani algoritmasının gelen girdilere göre değişebilmesini sağlamaktadır . Yani main’imizdeki “readData” yani dosya okuma işlemi verilen strateji tipine göre değişecektir.

Main’den çağırılan Context sınıfımız ise kendisine gelen strateji türüne göre “Strategy” interface’i üzerinden ilgili metodu çağırmamıza ve o metod üzerinden Strateji tipine göre veriyi okuyarak bize tüm çalışan nesnelerini ve özelliklerini(maaş,çalışan türü,isim,direktör) tutan bir ArrayList döndürülmesini sağlıyor.

Junit Test İçin Açıklama:

Bir sınıfın veya bir methodun çalışıp çalışmadığını kolayca kontrol edebiliriz.

Biz projemizde director ve officerlar için test classları oluşturduk.DirectorTest classında director oluşturup içinde listeleme,maliyet hesaplama ve iteratorın çalışıp çalışmadığını kontrol eden hasNext fonksiyonlarını oluşturduk.OfficerTest kısmında ise officer oluşturup listeleme ve maliyet hesaplama fonksiyonları oluşturduk.Son olarak da bunları test ettik.