GEBZE TEKNİK ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

CSE443 - Object Oriented Analysis and Design

HW2 Raporu

Ad Soyad	**Lütfullah TÜRKER**
Numara	**141044050**

Uygulamaları çalıştırmak için jar dosyasını çalıştırmak yeterlidir.
Terminalde : "java -jar Executable.jar" yazılabilir veya
Jar dosyasını çalıştırmak için terminal komutlarını yazmak yerine "Quick Run Command" dosyasını çalıştırarak direkt olarak uygulamayı çalıştırabilirsiniz.

Question 1

1. Normal durumda eğer Object'den gelen clone() metodunu override etmediysek Singleton objesinin klon'unu oluşturabiliriz ve iki farklı obje oluşur. Bunu clone() dan gelen obje ile clone() fonksiyonunu çağırdığımız objeyi hashCode fonksiyonu ile hashCode larını karşılaştırarak anlayabiliriz. İki farklı hash kodu verecektir.
2. Clone() metodu ile clone yapılmasına izin vermemenin yolu clone() metodunu override etmektir. Şu şekilde Singleton class ının kopyasının oluşturulmasını önleyebiliriz. Singleton class ında aşağıdaki şekilde override yaparak :

```
@Override
  protected Object clone() throws CloneNotSupportedException
  {
    throw new CloneNotSupportedException();
  }
```

   Bu sayede, klon yapılmaya çalışıldığı zaman Exception fırlatılacak ve klonlanma önlenmiş olacaktır.
3. Parent class ından dolayı Singleton class ı da klonlanabilir olacaktır. 1. Sorudaki gibi bunu önlemessek klonlanabilir fakat Singleton Class'ında Parent'tan gelen clone() metodu kullanılırsa klonlanabilir olacağı için bu metodu 2. Soruda yanıtladığım gibi override edersek yine klonlanamaz hale gelecektir ve sorun olmayacaktır. Kod örneği aşağıdadır.

```
class Parent implements Cloneable {
//…
  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();  // klonlanabilir halde
  }
}
```

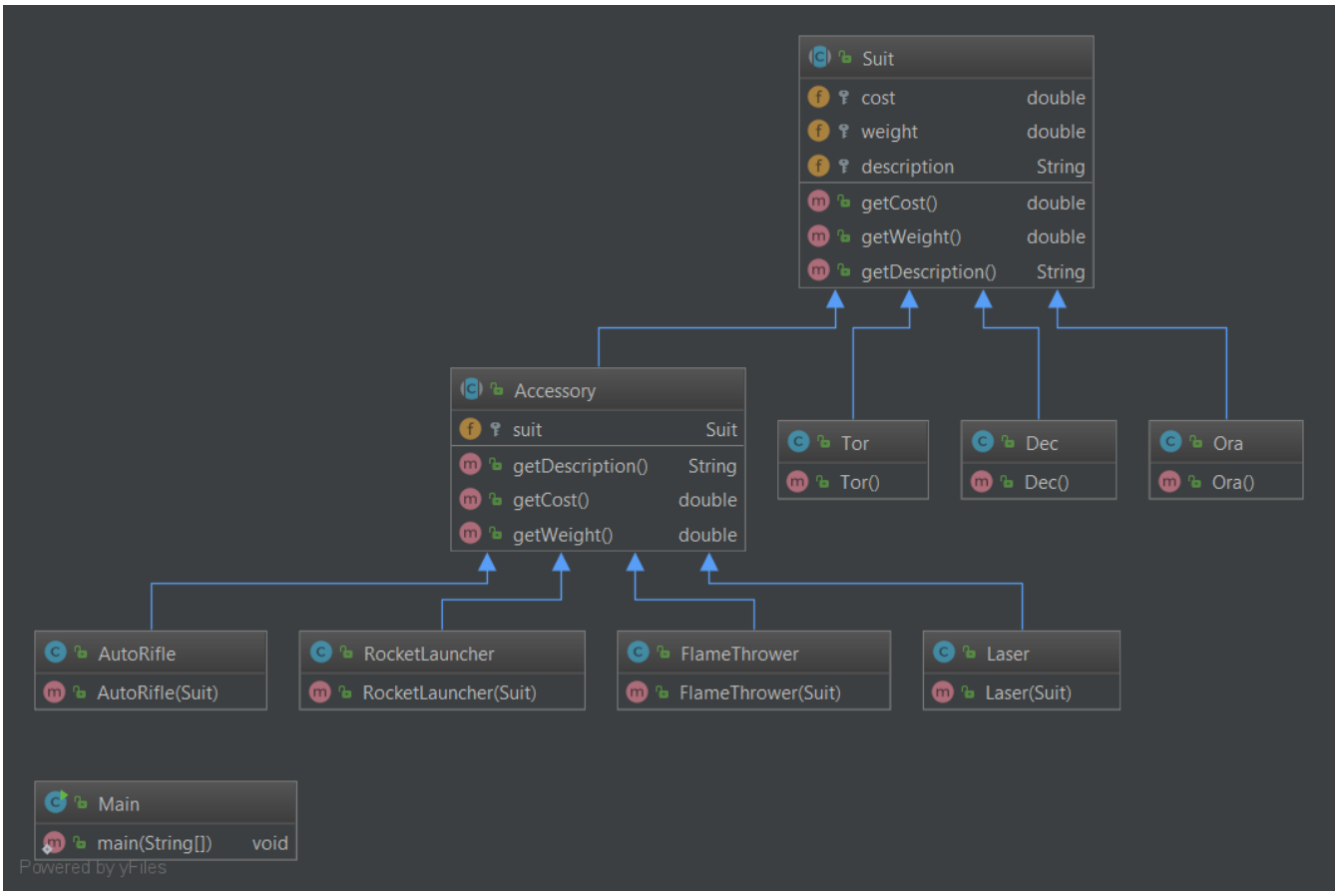
```
class Singleton extends Parent {
  public static Singleton instance = new Singleton();

  private Singleton()
  {}

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
  throw new CloneNotSupportedException();//Singleton'un klonlanmasını önledik
  }
}
```

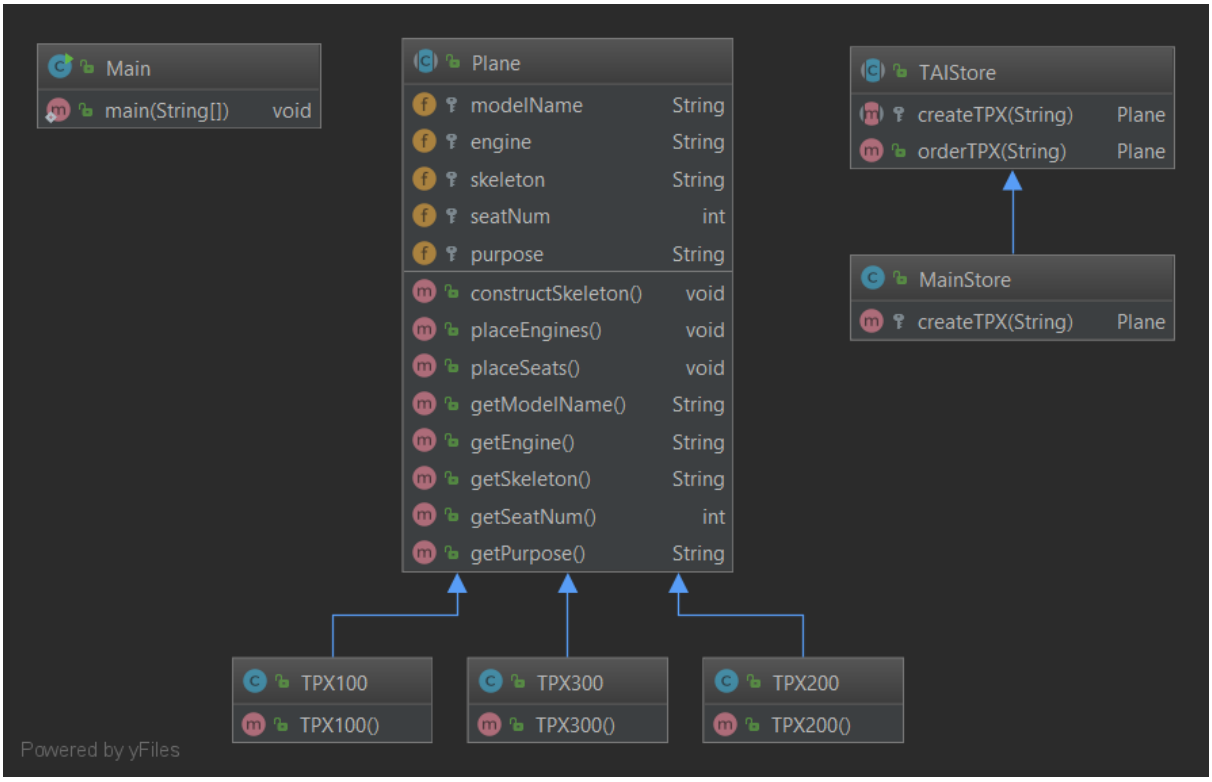
Question 2

2. part için Decorator design pattern kullanmayı uygun gördüm. Çünkü ana Suit seçimi yapıldıktan sonra sınırsız sayıda eklenti ekleme işlemi dinamik şekilde yapılabiliyor ve bu da decorator patternini bu sistem için uygun kılıyor.
Class Diagram aşağıdadır ve Klasörde de fotoğraf olarak mevcuttur.



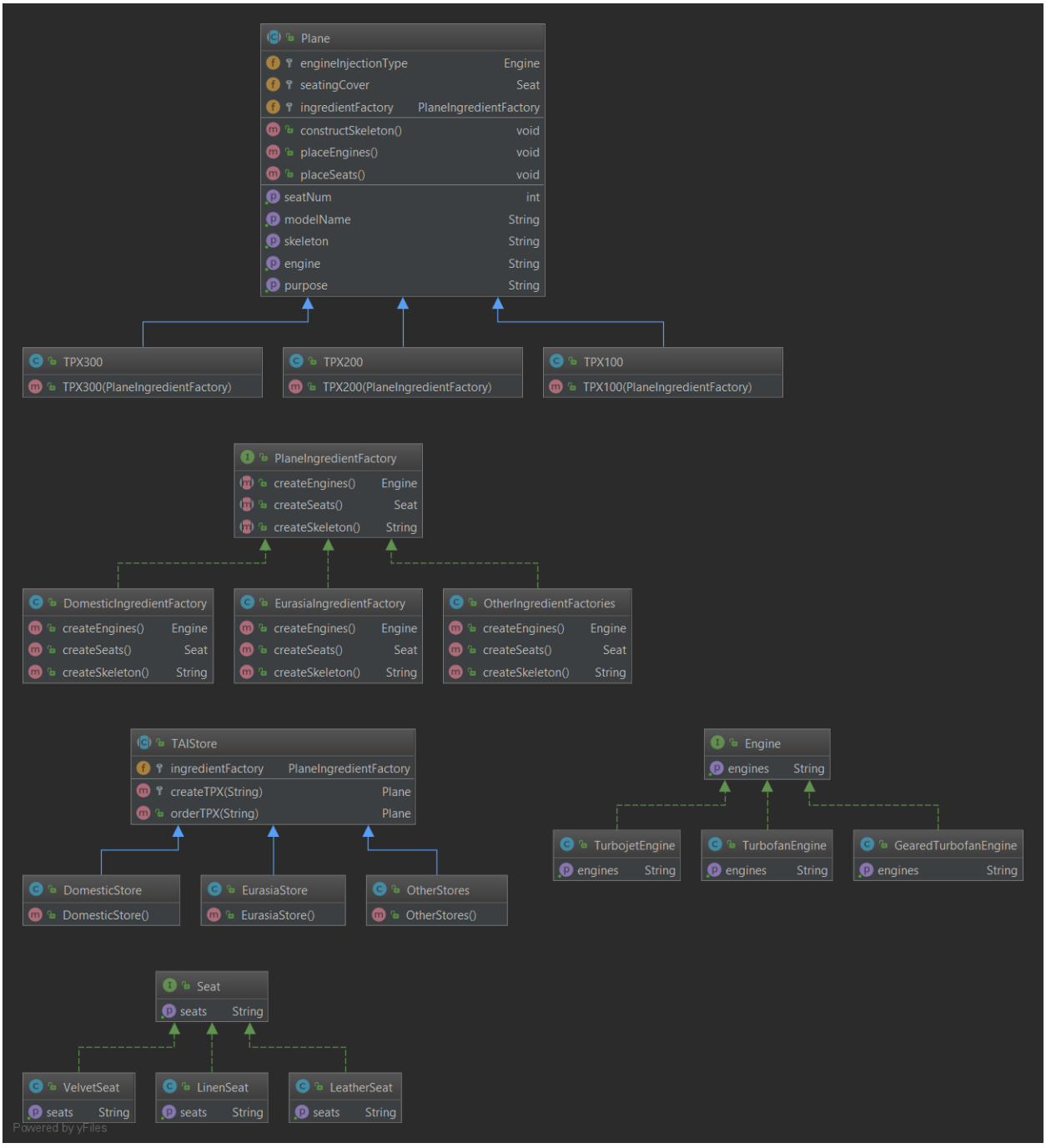
Question 3

1. Factory Method Pattern ini uyguladım. Yalnızca bir simülasyon olacağı için malzemeler yerine Stringler tutarak doğru malzemelerin kullanılıp kullanılmadığını anladım. Prepare fonksiyonu yerine gerekli atamaları constructor'da yaptım.
   TAIStore Abstract class'ında orderTPX fonksiyonunu implement ettim. Bu fonksiyon gelen String parametresini createTPX e göndererek doğru Plane objesini alıyor ve uçağı üretme işlemleri için plane içindeki constructSkeleton() ,placeEngines(), ve placeSeats() metodlarını çağırıyor ve uçağı üretiyor.(Üretim aşamalarını ve malzemeleri ekrana basıyor.)
   CreateTPX metodu ise alt sınıfta implement ediliyor.(kitaptaki Factory Method bu şekilde yapıldı diye öyle yaptım. Yoksa abstract Class'da da yapılabilirdi bu durumda. Farketmiyor.)
   Plane'ler için abstract Plane yazıldı ve gerekli tüm işlemler abstract class da yapıldı. Alt sınıf olan TPX 100,200 ve 300 sınıfları sadece constructor'larında gerekli variable leri kendi sınıfına uygun şekilde atama yapıyor ve üst sınıftaki fonksiyonları kullanıyor.
   Class Diagram aşağıdaki gibidir.

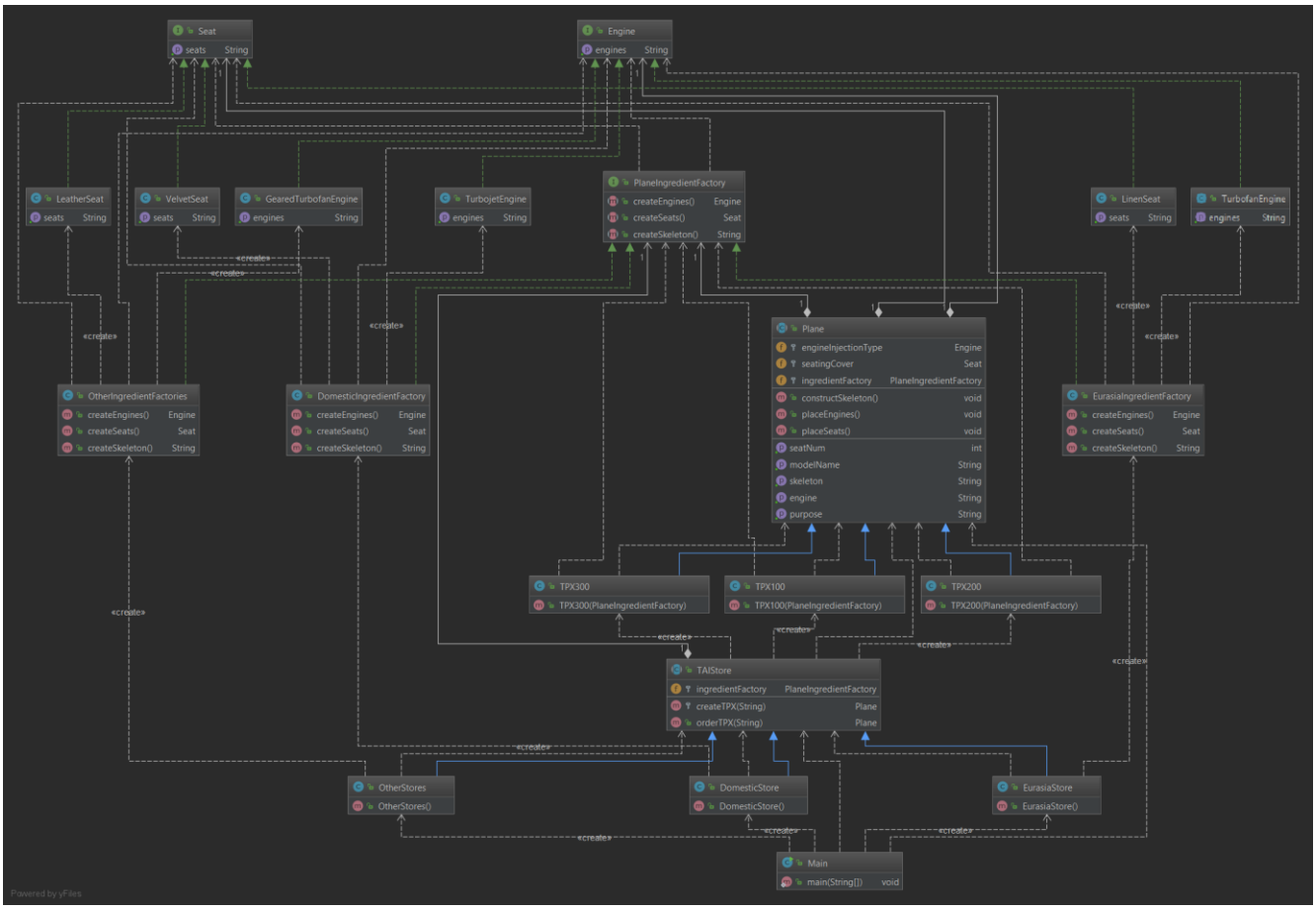


2. Abstract Factory Design Pattern i uygulandı. Önceki parttaki Plane ve alt sınıflarında Ingredient Factory'ye göre malzeme değişikliği olacağı için constructorlarda Ingredient Factory parametresi eklendi ve bu parametreye göre gerekli local malzemeler değişmekte.
   Local'e göre malzeme fabrikası değişeceği için PlaneIngredientFactory Interface'i yazıldı ve Domestic, Eurasia ve Other IngredientFactory class ları yazılıp bu Interface implement edildi. Bu classlar createEngine,createSeat ve createSkeleton fonksiyonları ile Local Markete göre farklı malzemeler ile üretim yapacak ve buraların Store'larına malzemeleri gönderecektir. Dolayısıyla bunun için de TAIStore abstract sınıfı yazıldı ve altında yine 3 Local Store için alt sınıflar oluşturuldu.
   Alt sınıflar sadece constructor'larında gerekli ingredientFactory objesini oluşturarak üst sınıftan gelen objeye atama yapıyor. createTPX ve orderTPX metodları abstract sınıfta yazıldı. (Kitapta her alt sınıf createPizza'yı implement etmiş ve sadece 1 satırında ingredientFactory atamasını farklı yapmış ve geri kalanı aynı. Bu yüzden ben constructor'da atamayı yapıp diğer işlemleri her class'da aynı şeyi yazmamak amacıyla abstract class'da yapmayı daha uygun gördüm.)
   Engine ve Seat tipleri için birer interface ve alt sınıflar yazıldı.
   Class diagramda Dependency'leri gösterdiğim zaman asıl bağlantıların anlaşılması zorlaştığı için iki halini de ayrı olarak hazırladım. Diyagramlar aşağıda mevcuttur.

Sade Class Diyagram:



Dependency’ler ile Class Diyagram:



Diyagramların tam boyutlu halleri fotoğraf olarak klasörlerde mevcuttur.