NESNE YÖNELİMLİ ANALİZ VE TASARIM DERSİ

AKILLI CİHAZ PROJE ÖDEVİ

Öğrenci Adı- Soyadı :Mertcan Demiralan

Öğrenci Numarası:G181210033

Öğrenci Ders Grubu:2A

1)Kullanıcı Doğrulama Ekranı

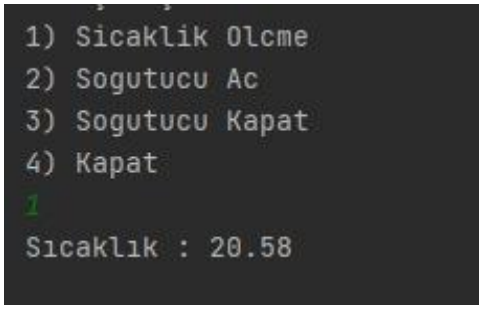
Username: user

Password: 123456

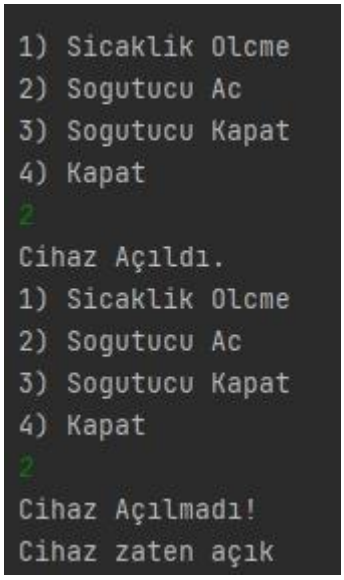
Service classında bulunan girisYap() fonksiyonu ile veritabanına bağlanarak veritabanında kayıtlı olan username ve passwordu kontrol ederiz. Eğer username ve password doğru ise giriş yapar değil ise giriş yapılmaz.

2)Sıcaklığın Görüntülenmesi, Soğutucunun açılıp kapatılması

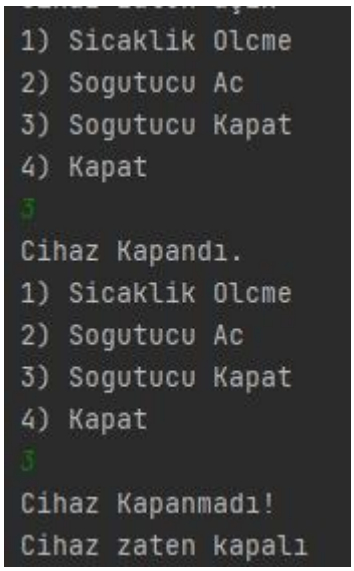
“1” secimi yapılırsa random bir sıcaklık oluşturulur. Ekrana yazdırılır. Service classındaki sicaklikGuncelle() fonksyonu ile ölçülen değer veritabanına gönderilir.



“2” secimi yapılırsa Api package içerisinde bulunan Service classındaki sogutucuGuncelle() fonksyonu sayesinde veritabanından soğutucunun durumunu güncelleriz. Eyleyici classında sogutucuAc() fonksyonu içinde çağırılan service.sogutucuGuncelle(true) fonksyonu true döner ve sogutucu açılır. Eğer halihazırda açıksa “zaten açık” mesajı ekrana yazdırılır.

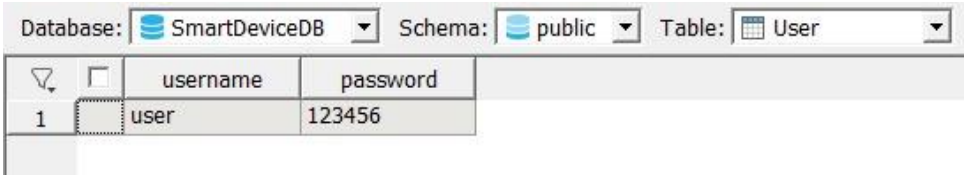


Aynı şekilde “3” secimi yapılırsa Api package içerisinde bulunan Service classındaki sogutucuGuncelle() fonksyonu sayesinde veritabanından soğutucunun durumunu güncelleriz. Eyleyici classında sogutucuKapat() fonksyonu içinde çağırılan service.sogutucuGuncelle(false) fonksyonu false döner ve sogutucu kapatılır. Eğer halihazırda kapalıysa “zaten kapalı” mesajı ekrana yazdırılır.

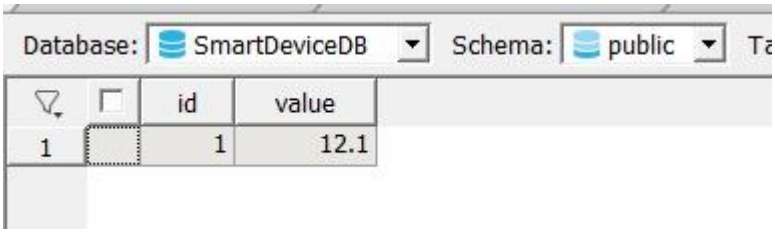


3)Veritabanı

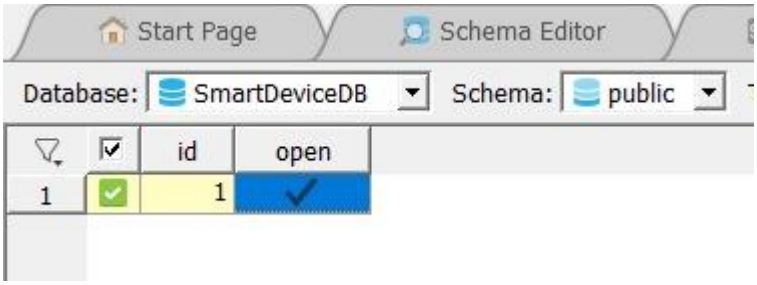
Kullanıcı tablosu:



Sicaklik tablosu:



Sogutucu tablosu:

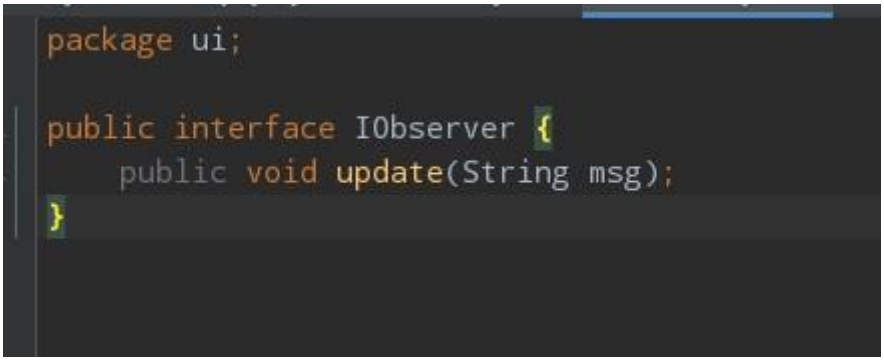


4)Dependency Inversion

Dependency inversion alt ve üst sınıfların birbirine bağımlılığının interface ile giderilmesine dayanan bir SOLID ilkesidir. Bu ilkenin amacı projelerdeki zaman maliyetini düşürmek ve koda sonradan eklenecek yapıların projeye girişini, okunabilirliği kolaylaştırmak.

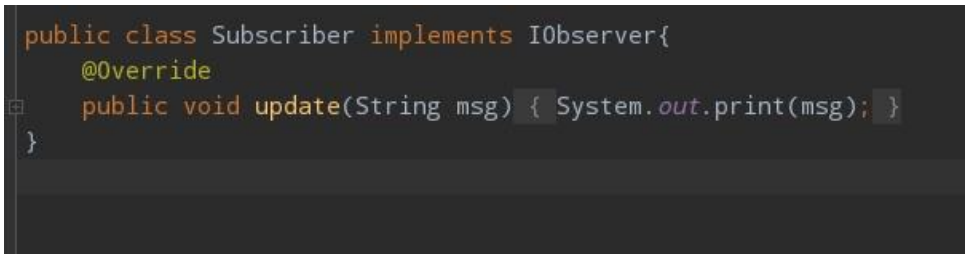
Kodumda bu ilkeyi mesaj gönderimi yapmak için kullandım. Yazici classı olusturdum ve araya soyut bir IObserver interface yazdım. IObserver interfaceini Subscriber sınıfana implement ettim. Böylelikle ikinci bir Subscribe sınıfına ihtiyac duyulursa

koda kolayca eklenebilecek. Ve yüksek seviyeli bir sınıfın alt seviyeli sınıflara bağlılığını azaltmış oluyoruz. Artık işlemler soyut bir sınıf üzerinden ilerleyecek.

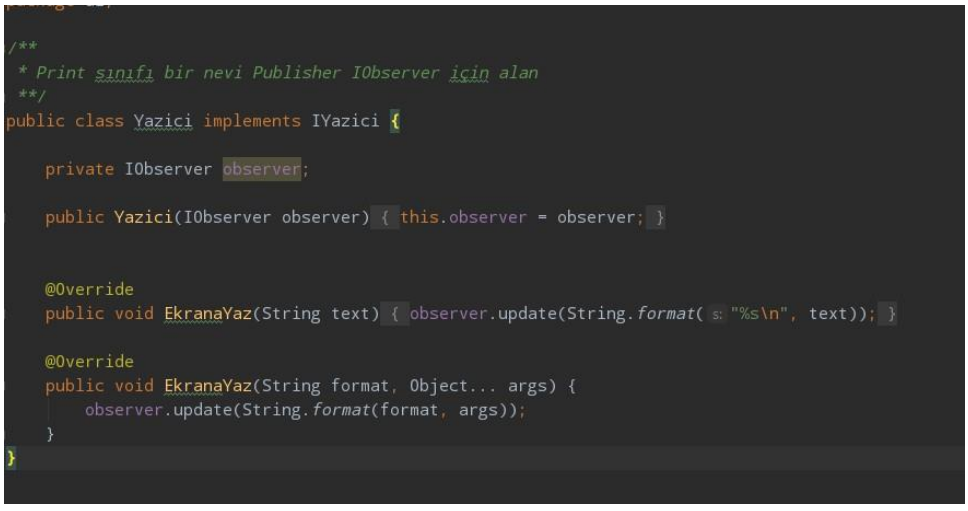


```
package ui;

public interface IObserver {
    public void update(String msg);
}
```



```
public class Subscriber implements IObserver{
    @Override
    public void update(String msg) { System.out.print(msg); }
}
```



```
/**
 * Print sınıfı bir nevi Publisher IObserver için alan
 **/
public class Yazici implements IYazici {

    private IObserver observer;

    public Yazici(IObserver observer) { this.observer = observer; }


    @Override
    public void EkranaYaz(String text) { observer.update(String.format( s: "%s\n", text)); }

    @Override
    public void EkranaYaz(String format, Object... args) {
        observer.update(String.format(format, args));
    }
}
```

```
    IObserver sub1 = new Subscriber();
    Yazici p = new Yazici(sub1); // dependency inversion
```

5)Builder & Observer Design Pattern

Builder Design Pattern

Bir nesne oluşturduğumuzda sınıfımızın içerisindeki değişkenlerden ürettiğimiz constructor metod çok fazla veya gereksiz parametreden oluşabilir. Nesne yaratılırken bu constructor metod içerisindeki kadar parametre alması gerekir. Farklı parametreler alan ya da istediğimiz kadar parametre alabilen metod oluşturumunda Builder Pattern kullanabiliriz.

```
Service service = new Service.ServiceBuilder( name: "SmartDeviceDB")
        .username("<username>")
        .password("<password>")
        .build();
```

Kodumda builderı Service classı içinde kullandım. Builder sayesinde database bilgilerimi ServiceBuildera parametre olarak verdim.

```
    public static class ServiceBuilder {
        private String name; // database adı
        private String username; // database usernamei
        private String password; // database sifresi

        public ServiceBuilder(String name) { this.name = name; }

        public ServiceBuilder username(String username) {
            this.username = username;
            return this;
        }

        public ServiceBuilder password(String pass) {
            this.password = pass;
            return this;
        }

        public Service build() { return new Service( builder: this); }
    }
}
```

Observer Design Pattern

Observer pattern, bir sistem içerisindeki farklı bölümlerin birbirine engel olmadan çalışmasını sağlar.

Çok sayıda nesneye gözlemledikleri olayı bildirmek için kullanırız.

Kodumda ekrana mesaj gönderiminde observer kullandım.

```java
package ui;

public interface IObserver {
    public void update(String msg);
}
```

```java
public class Subscriber implements IObserver{
    @Override
    public void update(String msg) { System.out.print(msg); }
}
```

6)Kaynak Kodlar & Video Adresi

Youtube Linki:

https://www.youtube.com/watch?v=85R8WgHcHfQ

Github Linki:

https://github.com/MertcanDemiralan/Nesne-Yonelimli-Analiz-Ve-Tasarim-Projesi