eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

SOLID LABORATEGIA

Software Ingeniaritza

Joritz Arocena eta Mikel Carrasco

25/26 ikasturtea

Proiektuaren esteka:

Hemen aurkitu daiteke proiektu osoaren kodea, honekin batera, garatutako uml-a ere agertzen da:
https://github.com/Carrascomikel/Solid/tree/master/SOLID_printzipioak

Solid printzipioak

1. Ireki-itxia printzipioa (OCP)

OCP printzipioa ez da betetzen emandako kode zatian, izan ere, autentifikazio zerbitzu bat gehitu beharko bagenu AuthService klasea aldatu beharko genuke.

➔ Arazoa konpontzeko, Service izeneko interfaze bat sortu dugu signIn metodoarekin eta gainontzeko zerbitzu bakoitzarentzat klase bat. Klase horiek, Service interfazea inplementatuko dute, OCP printzipioa betez.

Hona hemen Service interfazea:

Service.java
```
1  package ocp;
2
3  public interface Service {
4      public boolean signIn(String username,String password);
5  }
6
```

Aurretik genituen zerbitzuak, bakoitza klase batean egongo dira. Adibide gisa ServiceFB jarri dugu, bakoitzak bere kodea izango zukeen baina garrantzitsua klasearen goiburukoa da.

ServiceFB.java
```
1  package ocp;
2
3  public class ServiceFB implements Service {
4      public boolean signIn(String username,String pass){
5          //use the FB api
6          return true;
7      }
8  }
```

Azkenik egindako aldaketak AuthService-n izandako eragina:

AuthService.java

```java
package ocp;

public class AuthService {
    private Service service;
    public AuthService(Service service) {
        this.service=service;
    }
    public boolean signIn(String username,String pass) {
        return service.signIn(username, pass);
    }
}
```

➔ Gure ustez, errefaktorizazio bat egin dugu. Izan ere, errefaktorizazioa programaren portaera aldatu gabe kodea mantentzeko errazagoa bihurtzea, garbiagoa egitea eta hedagarriagoa egitea da. Beraz, gure kasua ikusita, errefaktorizazio bat egin dugula esan genezake.

2. Erresponsabilitate bakarreko printzipioa (SRP)

Emandako kode zatian, SRP hausten da. Bill klasean dagoen totalCalc() metodoak fakturaren totala kalkulatzea du helburu, baina horren baitan, dedukzioa eta VAT-a kalkulatzen ditu.

➔ Dedukzioa eta VAT-a kalkulatzeko klase bana sortu dugu. Honi esker, erresponsabilitate bakoitzak klase bana izango du eta SRP beteko da.
Deduction klasean, instantzia bat sortzen dugunean billAmount parametroa ezarriko zaio, ondoren dedukzioaren kalkulua egiteko erabilgarri egon dadin.

Dedukzio eta VAT-ari dagozkien klaseak:

```java
public class Deduction {
    private int deductionPercentage;
    public Deduction(int deductionPercentage) {
        this.deductionPercentage=deductionPercentage;
    }
    public float deductionCalculator(float billAmount) {
        float billDeduction;
        if (billAmount >50000)
             billDeduction = (billAmount * deductionPercentage +5) / 100;
        else
            billDeduction = (billAmount * deductionPercentage)/100;

        return billDeduction;

    }

}
```

```java
public class Vat {
    private double percentage=0.16;
    public float vatCalculator(float billamount,String code) {
        if(code.equals("0"))
            return 0;
        else
            return(float) (billamount*percentage);
    }
}
```

Bill klasea jasandako aldaketak ikus ditzakegu hurrengo irudian:

```java
package srp;

import java.util.Date;

public class Bill {
    public String code;
    public Date billDate;
    public float billAmount;
    public float VAT;
    public float billDeduction;
    public float billTotal;
    public int deductionPercentage;

    // Fakturaren totala kalkulatzen duen metodoa.
    public void totalCalc() {
    Deduction d=new Deduction(deductionPercentage);
    billDeduction=d.deductionCalculator(billAmount);

    Vat vat=new Vat();
    VAT=vat.vatCalculator(billAmount, code);

    billTotal = (billAmount - billDeduction) + VAT;
    }

}
```

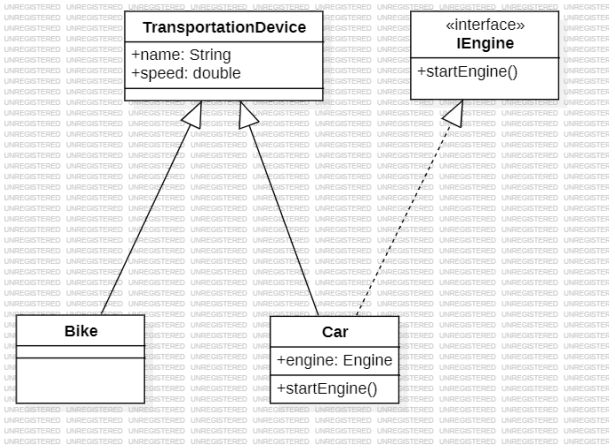
- ➔ Vat klasean percentage balioa aldatu beharko genuke. Dagokion setterra ezarriz posible izango litzateke.
- ➔ Vat klaseko vatCalculator metodoan sarrerako parametro gisa kodea jaso beharko litzateke. Horrekin batera, baldintza bat ezarri kodea zero denean zero itzultzeko.

3. Liskov printzipioa (LSK)

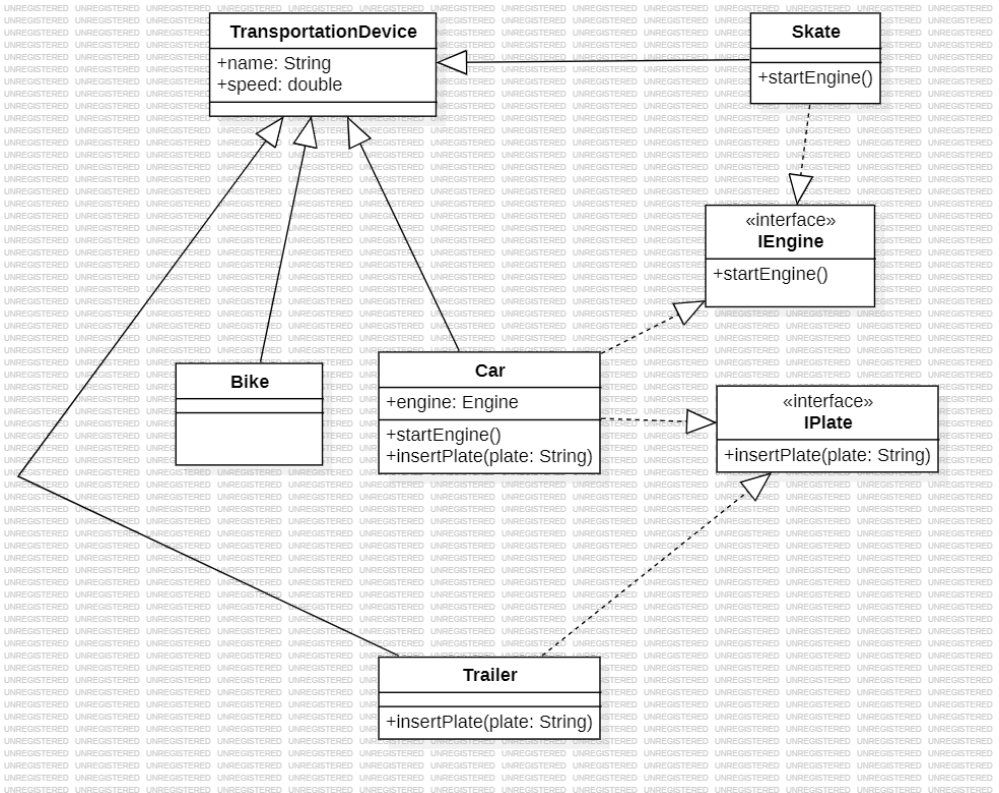
Jasotako kodeak ez du LSK betetzen motorra ez duten mugitzeko obkejtuak existitzen direlako. Bizikleta batek ez du motorrik baina

TransportationDevice bat da. Liskov-ek zioen ezin duela hedatutako metodo hutsik egon, hau da, bere barnean implementaziorik ez duena edo salbuespen bat altxatzen duena.

➔ Arazoa konpontzeko, interfaze bat sortuko dugu motorra duen klase oro identifikatu ahal izateko. Adibidez, Car klaseak TransportationDevice hedatzeaz gain, IEngine izeneko interfazea ere implementatuko du.



➔ Beste interfaze bat sortuko dugu matrikula duten ibilgailuentzako. Honela geratuko zaigu gure UML diagrama:



4. Dependentzi inbertsioaren printzipioa (DIP)

- ➔ Ez du dependentzin inbertsioaren printzipioa betetzen. Izan ere, vat edo dedukzioa kalkulatzeko era aldatuko bagenu, Bill klasea ere aldatu beharko genuke. Hau da, mendekotasuna du bi klase horiekin.
- ➔ Arazoa konpontzeko, portaera orokorra errepresentatzen duten interfazeak sortuko ditugu. IDeduction eta IVat interfazeak sortuz abstrakzio maila bat sortuko dugu eta Bill klasea ez da aldatu beharko dedukzioa edo vat-a kalkulatzeko modua aldatzen denean.
  Ikus dezagun kodea nola geratu den:

  Aipatutako bi interfazeak:

```
public interface IDeduction {
    public float calcDeduction(float billAmount,float deductionPercentage);

}
```

```
public interface IVat {
    public float calcVAT(float billAmount);
}
```

  Interfaze horiek inplementatzen dituzten klaseak:

```
public class Deduction implements IDeduction {

   @Override
   public float calcDeduction(float billAmount, float deductionPercentage) {
       float billDeduction;
       if (billAmount >50000)
            billDeduction = (billAmount * deductionPercentage +5) / 100;
       else
           billDeduction = (billAmount * deductionPercentage)/100;

       return billDeduction;
   }

}
```

```
public class Vat implements IVat {

    private double percentage = 0.16;
    @Override
    public float calcVAT(float billAmount) {
        return (float) (billAmount*percentage);
    }

}
```

Amaitzeko, Bill klasea:

```
public class Bill {
    public String code;
    public Date billDate;
    public float billAmount;
    public float VAT;
    public float billDeduction;
    public float billTotal;
    public float VATAmount;
    public int deductionPercentage;

    private IDeduction d;
    private IVat v;
    public Bill(IDeduction d,IVat v) {
        this.d=d;
        this.v=v;
    }
    // Fakturaren totala kalkulatzen duen metodoa.
    public void totalCalc() {
    // Dedukzioa kalkulatu

    billDeduction = d.calcDeduction(billAmount ,deductionPercentage);
    // VAT kalkulatzen dugu

    VATAmount = v.calcVAT(billAmount);
    // Totala kalkulatzen dugu
    billTotal = (billAmount - billDeduction) + VATAmount;
    }

}
```

5. Interfaze bananduaren printzipioa (ISP)

➔ EmailSender-ek Person klaseko korreora bidaltzen du mezua, hau da, nahikoa du pertsonaren korreoarekin.
Bestalde, SMSender-ek Person klaseko telefonora bidaltzen du mezua, hau da, nahikoa du telefonoarekin.
Nahiz eta klase bakar bateko bi metodo behar dituzten, Person instantzia oso bat pasatzen zaie parametro gisa eta honek ISP bortxatzen du.

➔ Bi interfaze sortu ditugu, bata EmailSender-ek beharrezkoak dituen metodoak definitzeko eta bestea SMSSender-ek beharrezkoa dituen metodoak definitzeko:

```java
public interface IEmail {
    public String getAddress();
}
```

```java
public interface ISMS {
    public String getTelephone();
}
```

Person klaseak bi interfazeak inplementatuko ditu:

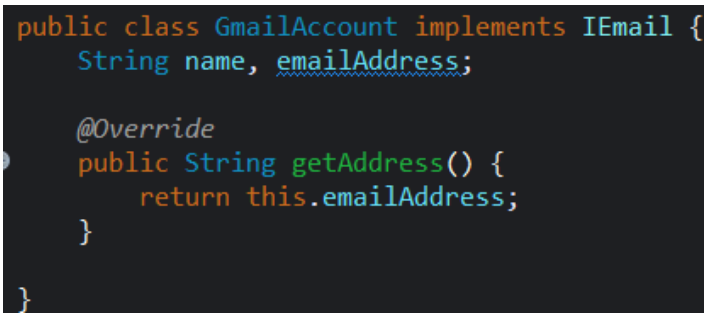
```java
public class Person implements IEmail,ISMS {
    String name, address, email, telephone;
     public void setName(String n) { name=n; }
     public String getName() { return name; }

     public void setAddress(String a) { address=a; }
     public String getAddress() { return address; }

     public void setEmail (String e) { email=e; }
     public String getEmail () { return email; }

     public void setTelephone(String t) { telephone=t; }
     public String getTelephone() { return telephone;}
}
```

Interfazeak sortu ondoren honela ditugu mezua bidaltzen duten klaseak:

```java
public class SMSSender {
    public static void sendSMS(ISMS c, String message){
        //SMS bat bidaltzen du Person klaseko telefono zenbakira. }
    }
}
```

```java
public class EmailSender {
    public static void sendEmail(IEmail e, String message){
        // Mezu bat bidaltzen du Person klaseko korreo helbidera. }
        }
}
```

➔ Azkenik honela geratu zaigu GmailAccount klasea:



```
public class GmailAccount implements IEmail {
    String name, emailAddress;

    @Override
    public String getAddress() {
        return this.emailAddress;
    }

}
```