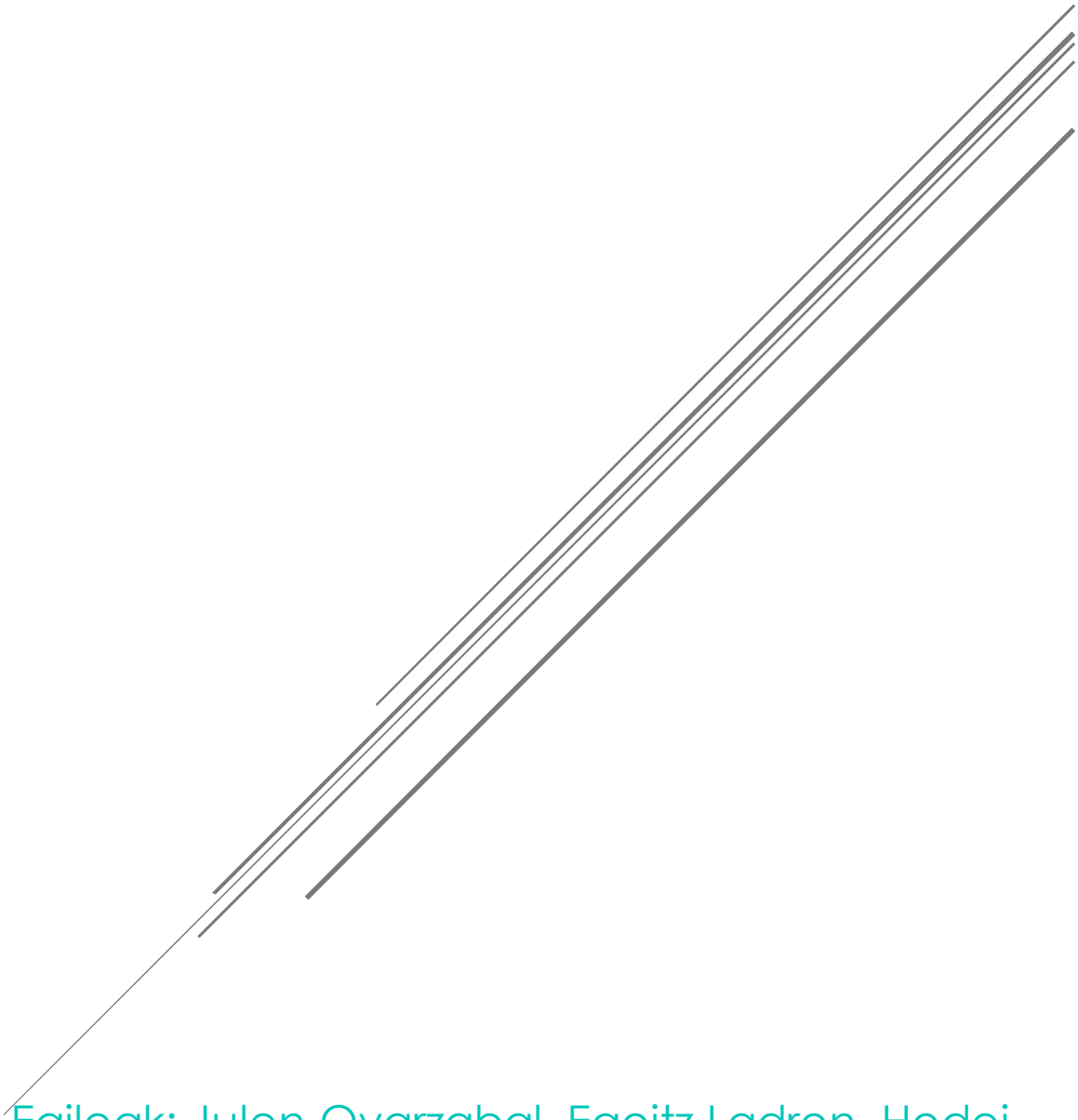
SOLID LABORATEGIA

SOLID printzipioen inguruko laborategiaren soluzioa



Egileak: Julen Oyarzabal, Egoitz Ladron, Hodei Jauregi

1.SARRERA....2
2.IREKI-ITXIA PRINTZIPIOA (OCP)....2
1....2
2....3
2.ERRESPONSABILITATE BAKARREKO PRINTZIPIOA (SRP)....3
1....3
2 eta 3. 4
3.LISKOV PRINTZIPIOA (LSK)....4
1....4
2....5
4.DEPENDENTZI INBERTSIOAREN PRINTZIPIOA (DIP) 6
1....6
2....6
5.INTERFAZE BANANDUAREN PRINTZIPIOA (ISP)....7
1....7
2....8
3....9

1.SARRERA

Dokumentu honetan SOLID printzipioen inguruko ariketen emaitzak daude. Ariketen zehar garatutako kodea ikusteko ondorengo link-a erabili: https://github.com/egoitz-ehu/SOLID_printzipioak

2.IREKI-ITXIA PRINTZIPIOA (OCP)

1.

Hasierako kodeak ez du betetzen OCP printzipioa. Arazo hori konpontzeko ondorengoa egingo dugu.

- IService izeneko interfaze bat sortuko dugu. Bertan signIn izeneko metodo baten signatura definituko dugu (benetan saioa irekitzeko kodea egongo den metodoaren signatura). Honakoa da kodea:

```
public interface IService {
    public boolean signIn(String log, String pass);
}
```

- Ondoren autentifikazio zerbitzu bakoitzeko klase bat sortuko dugu, non aurreko interfazea inplementatuko duen, bertan dagokion kodea idatziko dugun. Hemen GoogleService adibidea:

```
public class GoogleService implements IService {

    @Override
    public boolean signIn(String log, String pass) {
        // use Google API
        return true;
    }

}
```

- Azkenik AuthService klasea aldatuko dugu. Bertan definituta zeuden metodoak dagokion klasera mugitu dira. Ondorioz signIn metodoa geratzen da. Bertan parametro moduan edo klaseko atributu moduan jasotako IService interfazearen signIn metodoa exekutatuko da. Horrela exekuzioaren garaian aldatu daiteke autentifikazio metodoa. Hemen kodea:

```java
public class AuthService {
    private IService service;

    public AuthService(IService service) {
        this.service=service;
    }

    public void setService(IService service) {
        this.service=service;
    }

    public boolean signIn(String log, String pass) {
        return service.signIn(log, pass);
    }
}
```

2.

Errefaktorizazioaren definizioarekin zorrotza izango bagina ez litzateke errefaktorizazio bat izango, metodoaren signatura aldatu dugulako. Baina errefaktorizazio bat dela esan daiteke, kodearen egiturak aldaketa bat eduki duelako.

2.ERRESPONSABILITATE BAKARREKO PRINTZIPIOA (SRP)

1.

Dedukzioaren eta VAT kalkulatzea beste klase batzuetara mugitu ditugu, zehazki, DeductionService eta VATService. Dedukzioaren kalkulua beste metodo batera eraman dugunez parametro moduan igaro beharko ditugu fakturaren buruzko datuak:

```java
public float calculateDeduction(float billAmount) {
    if (billAmount >50000)
        return (billAmount * deductionPercentage +5) / 100;
    else return (billAmount * deductionPercentage) / 100;
}
```

2 eta 3.

Beste klase batean dugunez VAT kalkulua bertan if galdera bat jarri jakiteko eta aplikatu behar zaion ehunekoa:

```
public class VATService {
    private double percentage = 0.16;
    public float calculateVAT(float amount, String code) {
        if(!code.equals("0")) {
            return (float) (amount*percentage);
        } else {
            return 0;
        }
    }
}
```

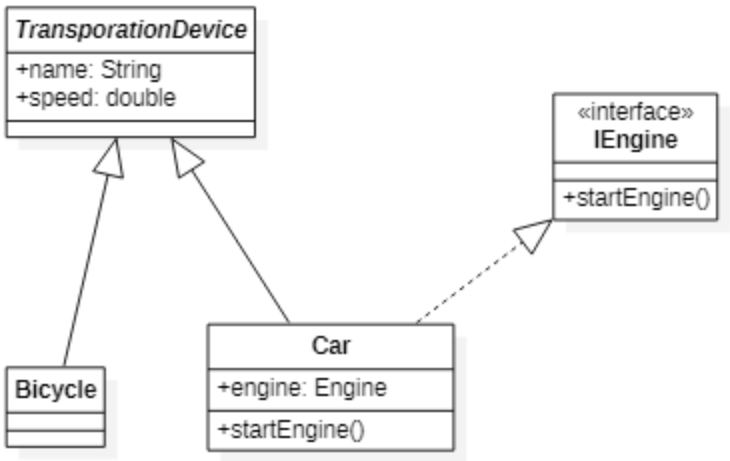
OCP printzipioa bete nahiko bagenuke IVATizeneko interfaze bat sortu genezake, eta interfaze hori inplementatzen duten klaseak sortu (bakoitza bere arauekin). Berdina dedukzioarekin.

Beste aukera zen parametro moduan jasotzea ehunekoa, horrela ehunekoa dei bakoitzean aldatu dezakegu.

3.LISKOV PRINTZIPIOA (LSK)

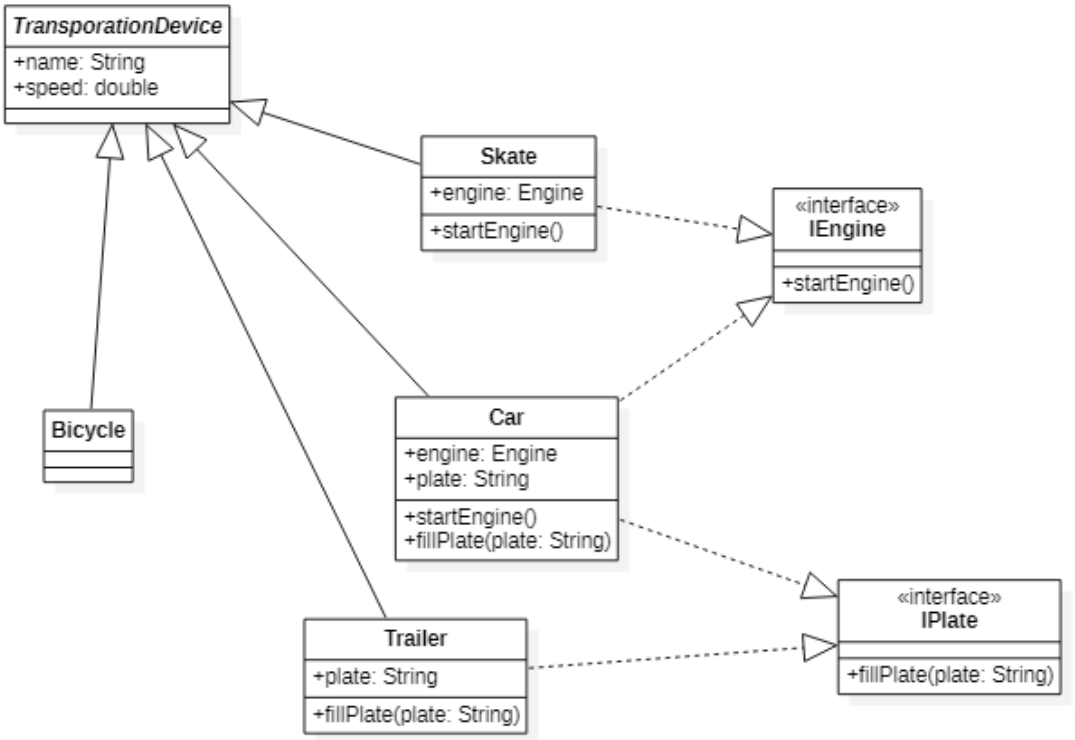
1.

Interfaze bat sortuko dugu motor bat daukaten klaseak identifikatzeko. Ondoren motorra duten gailuek inplementatuko dute. Honako da dagokion bere UML diagrama:



2.

Beste interfaze bat sortuko dugu, hau matrikula duten kotxeentzako. Horrela geratuko da:



Adibidez Car klasea honela geratuko da:

```
public class Car extends TransportationDevice implements IEngine, IPlate {
    Engine engine;
    String plate;

    @Override
    public void fillPlate(String plate) {
        // Dagokion kodea

    }

    @Override
    public void startEngine() {
        // Dagokion kodea

    }
}
```

Aldaketa guzti horiek dituen kodea hasierako link-ean dago.

4.DEPENDENTZI INBERTSIOAREN PRINTZIPIOA (DIP)

1.

Ez du DIP printzipioa betetzen. Bill klaseko totalCalc metodoak Deduction eta VAT klaseekiko dependentzia dauka, hau da, kalkuluak egiteko modua aldatu nahi dugunean gure metodoa aldatu beharko dugu. Ondorioz ez da DIP printzipioa betetzen. Printzipio honek esaten du gure klaseak ez duela dependentzia eduki behar inplementazioarekin, abstrakzioarekin baizik.

2.

Printzipioa betetzeko abstrakzio maila bat ezarri beharko dugu. Horretarako DeductionCalculator eta VATCalculator izeneko interfaze batzuk definituko ditugu. Hasieran genuen klaseek klase horiek inplementatuko dituzte, Deduction klaseak DeductionCalculator eta VAT klaseak VATCalculator hain zuzen. Ondoren gure Bill klaseari eraikitzailearen bidez sar diezaiokegu interfaze horiek. Horrela gure klaseak ez du jakin behar zein inplementazio konkretu erabili behar duen, eraikitzailea sortzean DeductioCalculator eta VATCalculator interfazea inplementatzen dituen instantziak jasoko ditu.

Honakoa izango da Bill klasea(gainerako kodea hasierako link-Ean):

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

    private DeductionCalculator d;
    private VATCalculator vat;

    public Bill(DeductionCalculator d, VATCalculator vat) {
        this.d=d;
        this.vat=vat;
    }

    public float getVATAmount() {
        return VATAmount;
    }

    public int getDeductionPercentage() {
        return deductionPercentage;
    }

    // Fakturaren totala kalkulatzen duen metodoa.
    public void totalCalc() {
        // Dedukzioa kalkulatu
        billDeduction = d.calcDeduction(billAmount ,deductionPercentage);
        // VAT kalkulatzen dugu
        VATAmount = vat.calcVAT(billAmount);
        // Totala kalkulatzen dugu
        billTotal = (billAmount - billDeduction) + VATAmount;
    }
}
```

5.INTERFAZE BANANDUAREN PRINTZIPIOA (ISP)

1.

sendEmail metodoaren kasuan emaila eta mezuarekin nahiko da bere lana burutzeko, baina Person klaseko instantzia osoa jasotzen du. Bere lana burutzeko behar ez dituen balio asko lortzen ditu, ondorioz ISP printzipioa apurtu egiten du.

sendSMS metodoaren kasuan berdina gertatzen da. Kasu honetan Person klaseko telefono zenbakia bakarrik behar dugu, baina instantzia osoa jasotzen du parametro moduan. Berriz ere ISP printzipioa apurtzen da.

2.

Bi interfaze sortuko ditugu. Bat Emailable izenekoa, email bat bidaltzeko beharrezko metodoak definitzen dituena, eta beste interfazea SMSable SMS mezuak bidaltzeko. Honakoa da interfazeen kodea:

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

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

Person klaseak bi ekintza egiteko prest egon behar denez bi interfazeak inplementatuko ditu:

```java
public class Person implements Emailable, SMSable{
    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; }
}
```

Ondoren EmailSender eta SMSSender klaseek ez dute jasko Person klase osoa, beharrezkoa interfazea baizik. Horrela bakarrik beharrezko informazioa lortuko dute. Honakoa da kodea:

```java
public class EmailSender {
    public static void sendEmail(Emailable c, String message) {
        // Mezua bidali
    }
}
```

```java
public class SMSSender {
    public static void sendSMS(SMSable c, String message) {
        // SMS bidali
    }
}
```

3.

GmailAccount klaseak mezu elektronikoak bidaltzeko ahalmena eduki behar duenez Emailable inplementatuko du, baina ez duenez SMS mezurik bidali behar, hortaz ez du inplementatuko SMSable. Honakoa izango da GmailAccount klasea:

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

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

}
```

Emailable interfazea inplementatzen duenez EmailSender erabili dezake, baina ez duenez SMSable inplementatzen ezingo du SMSSender erabili.

Honakoa da eskatutako programaren adibidea:

```
public class Programa {

    public static void main(String[] args) {
        GmailAccount gA = new GmailAccount();
        EmailSender.sendEmail(gA, "Mezua");

        // Errorea ematen du:
        // The method sendSMS(SMSable, String) in the type SMSSender is not applicable for the arguments (GmailAccount, String)
        // SMSSender.sendSMS(gA, "mezua");
    }

}
```