

Principios SOLID:

Laboratorio

Autores:

- Rubén Gallego
- Mikel Berasategui

Asignatura: Ingeniería de Software II

Enlaces del proyecto:

- Github: <https://github.com/RubenGGBC/SolidEjercicios>

1. Principio Abierto-Cerrado (OCP).....	3
2. Principio de Responsabilidad Única (SRP)	6
3. Principio de Sustitución de Liskov (LSK).....	14
4. Principio de Segregación de Interfaces (ISP)	19
5. Principio de Inversión de dependencia (DIP)	24

1. Principio Abierto-Cerrado (OCP)

La siguiente clase implementa un servicio de autenticación a través de varios servicios externos (facebook, google y twitter)

```
public class AuthService {  
    public boolean signIn(String service, String log, String pass) {  
        if (service.compareTo("facebook")==0)  
            return signInWithFB(log, pass);  
        if (service.compareTo("google")==0)  
            return signInWithGoogle(log, pass);  
        if (service.compareTo("twitter")==0)  
            return signInWithTwitter(log, pass);  
        return false;  
    }  
    public boolean signInWithFB(String log, String pass) {  
        //use the FB api  
        return true;  
    }  
    public boolean signInWithGoogle(String log, String pass) {  
        //use the google api  
        return true;  
    }  
    public boolean signInWithTwitter(String log, String pass) {  
        //use the Twitter api  
        return true;  
    }  
}
```

Tareas a realizar:

1. Te solicitan que extiendas tu servicio de autenticación, para que tambien te puedas autenticar con el servicio de Apple. ¿Qué cambios deberias realizar en la clase `AuthService` para que cumpliese el Principio OCP?. Justifica tu respuesta.

-La clase `AuthService` contiene varios ifs para determinar que servicio usar, como indica el enunciado, si queremos añadir otro servicio, como por ejemplo el de registrarse con Apple, tendríamos que modificar la clase, esto viola la condición de OCP, que indica que el código tiene que estar abierto para extensión, pero no para modificación.**

Para realizar este cambio lo que tenemos que hacer es hacer que `AuthService` sea una interfaz que contenga el método

*singIn, esto nos permite que todos los servicios que implementen esta interfaz contengan su propia definición del método y para cada servicio de autenticación hacer que sea una nueva clase que hereden dicha interfaz. ***

2. ¿Consideras que el cambio que has realizado es una refactorización? Justifica la respuesta.

*-**Sí, ya que ahora mismo si queremos extender nuestros servicios de autenticación simplemente tenemos que crear una clase nueva que implemente AuthService y definiríamos su propio método de login, si quisiéramos extender las funciones de cada clase, tampoco tendríamos problemas ya que cada clase es independiente de la otra**

-Clase AuthServiceI:

```
AuthServiceI.java
package ejer1;

public interface AuthServiceI { 3 usages 3 implementati
    public boolean singIN( String log, String pass);

}
```

-Clase AuthServiceFB:

```
AuthServiceFB.java
package ejer1;

public class AuthServiceFB implements AuthServiceI {
    public boolean singIN( String log, String pass) {
        //use the FB api
        return true;
    }
}
```

-Clase AuthServiceGoogle:

```
AuthServiceGoogle.java
package ejer1;

public class AuthServiceGoogle implements AuthServiceI{
    public boolean singIN(String user, String pass){ no
        //use the google api
        return true;
    }
}
```

-Clase AuthServiceTwitter:

```
AuthServiceTwitter.java

package ejer1;

public class AuthServiceTwitter implements AuthServiceI {
    public boolean singIN(String log, String pass) { no u:
        //use the Twitter api
        return true;
    }
}
```

2. Principio de Responsabilidad Única (SRP)

Dada la siguiente clase Bill:

```
public class Bill {
    public String code; // Representa un número de 5 dígitos
    public Date date;
    public float InitialAmount;
    public float totalVAT;
    public float totalDeduction;
    public float billTotal;
    public int deductionPercentage;

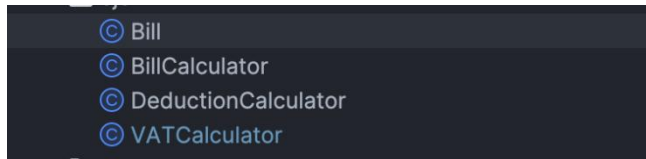
    // Método que calcula el total de la factura
    public void billTotalCalc() {
        // Calculamos la deducción
        totalDeduction = (InitialAmount * deductionPercentage) / 100;
        // Calculamos el IVA
        totalVAT = (float) (InitialAmount * 0.16);
        // Calculamos el total
        billTotal = (InitialAmount - totalDeduction) + totalVAT;
    }
}
```

Puede parecer que la responsabilidad de esta clase es única, ya que calcula el total de la factura y que, efectivamente, la clase cumple con su cometido. Sin embargo, no es cierto que la clase contenga una única responsabilidad. En la implementación del método `billTotalCalc` se observan además el cálculo del importe base de la factura, la aplicación sobre el importe a facturar de un descuento o deducción, y la aplicación de un 16% de IVA. El problema está en que, si en el futuro tuviéramos que tuviéramos que aplicar una deducción diferente en base al importe de la factura o bien, modificar la tasa de IVA, tendríamos que modificar la clase `Bill`. Por lo tanto, con el diseño actual las responsabilidades quedan acopladas entre sí, y la clase violaría el principio SRP.

Tareas a realizar:

1. Refactoriza la aplicación para que cada responsabilidad quede aislada en una clase.

-Objetivo:



En la clase `Bill`

debemos cambiar el método `billTotalCalc()` y hacer que cree una instancia de la clase `BillCalculator`, que será una nueva clase, la cual se encargará de realizar el cálculo:

```
Bill.java
package ejer2;

import java.util.Date;

public class Bill { 1 usage  @RubenGGBC *
    public String code; // Representa un número de 5 dígitos
    public Date date; no usages
    public float InitialAmount; 3 usages
    public float totalVAT; 2 usages
    public float totalDeduction; 2 usages
    public float billTotal; 1 usage
    public int deductionPercentage; 1 usage

    public void billTotalCalc() { no usages  @RubenGGBC
        BillCalculator calculator = new BillCalculator();
        calculator.calculateTotals( bill: this);
    }
}
```

La nueva clase *BillCalculator* tendrá un método que se encargará de realizar el cálculo de Bill. Para realizar dicho calculo se crea una instancia de cada una de las 2 nuevas clases: *DeductionCalculator* (la cual se encarga de calcular la deducción) y *VATCalculator* (la cual se encarga de calcular el VAT).

```
BillCalculator.java
public class BillCalculator { 2 usages  @ RubenGGBC

    private DeductionCalculator deductionCalculator; 2 usages
    private VATCalculator vatCalculator; 2 usages

    public BillCalculator() { 1 usage  @ RubenGGBC
        this.deductionCalculator = new DeductionCalculator();
        this.vatCalculator = new VATCalculator();
    }

    public void calculateTotals(Bill bill) { 1 usage  @ RubenGGBC
        // Calcular deducción usando la clase especializada
        bill.totalDeduction = deductionCalculator.calculate(bill.InitialAmount, bill.deductionPercentage);

        // Calcular IVA usando la clase especializada
        bill.totalVAT = vatCalculator.calculate(bill.InitialAmount);

        // Calcular total final
        bill.billTotal = (bill.InitialAmount - bill.totalDeduction) + bill.totalVAT;
    }
}
```

En el caso de la clase *DeductionCalculator* definimos el método *calculate*, que calcula la deducción según el criterio utilizado (en nuestro caso, $(\text{cantidad inicial} * \text{porcentaje de deducción}) / 100$).

```
DeductionCalculator.java
public class DeductionCalculator { 2 usages  @ RubenGGBC

    public float calculate(float initialAmount, int deductionPercentage) {
        return (initialAmount * deductionPercentage) / 100;
    }
}
```


Por otro lado, el VATCalculator tendrá el método calculate, que calculará el VAT según el criterio utilizado.

```
VATCalculator.java
public class VATCalculator { 2 usages  RubenGGBC

    public float calculate(float initialAmount) { return (float) (initialAmount * 0.16); }
}
```

- Indica qué cambios tendrías que realizar (en cada una de las versiones antes y después de la refactorización) si el totalDeduction se calculase en base al montante de la factura:

-Antes de la refactorización habría que cambiar toda la clase Bill, en cambio después de la refactorización, solo tenemos que aplicárselo a una de las 4****

-Antes de la refactorización:

```
Bill.java
public void billTotalCalc() { no usages
    if (InitialAmount > 35600) {
        totalDeduction = (InitialAmount * deductionPercentage + 4.5f) / 100;
    } else {
        totalDeduction = (InitialAmount * deductionPercentage) / 100;
    }

    totalVAT = (float) (InitialAmount * 0.16);
    billTotal = (InitialAmount - totalDeduction) + totalVAT;
}
```

-Después de la refactorización:

```
DeductionCalculator.java
package ejer2;

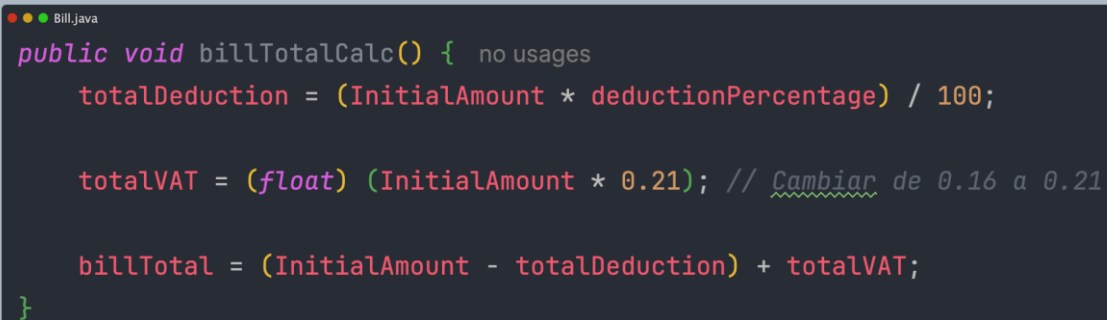
public class DeductionCalculator { 2 usages  RubenGGBC *
    public float calculate(float initialAmount, int deductionPercentage) {
        if(initialAmount>3600) {
            return (initialAmount * deductionPercentage+4.5f) / 100;
        }
        return (initialAmount * deductionPercentage) / 100;
    }
}
```

3. Indica los cambios que tendrías que realizar (en cada una de las versiones antes y después de la refactorización) si el IVA cambiase del 16 al 21%.

```
si (importeFactura>35600)
    totalDeduction =
        (InitialAmount * deductionPercentage +4.5) / 100;
sino totalDeduction =
        (InitialAmount * deductionPercentage) / 100;
```

-**Si quisiéramos realizar el cambio de IVA tendríamos que cambiar otra vez toda la clase de Bill antes de la refactorización, después de la refactorización solo tenemos que el valor de la variable**

-Antes de la refactorización:



```
Bill.java
public void billTotalCalc() { no usages
    totalDeduction = (InitialAmount * deductionPercentage) / 100;

    totalVAT = (float) (InitialAmount * 0.21); // Cambiar de 0.16 a 0.21

    billTotal = (InitialAmount - totalDeduction) + totalVAT;
}
```

-Después de la refactorización:

```
VATCalculator.java
package ejer2;
public class VATCalculator { 2 usages  ⤵ RubenGGBC *
    private float vatRate = 0.21f; // SOLO CAMBIAR AQUÍ: de 0.16 a 0.21

    public float calculate(float amount) { 1 usage  new *
        return amount * vatRate;
    }
}
```

- Indica los cambios que tendrías que realizar (en cada una de las versiones antes y después de la refactorización) si a las facturas de código menores de 10, no se le aplicase el IVA.

-**Tendríamos que volver a cambiar toda la clase Bill, añadir un if de si el código es menor que 10 y el IVA se tendría que eliminar, en cambio, al hacer la refactorización solo tenemos que agregar el if en la clase correspondiente**

-Antes de la refactorización:

```
Bill.java
public void billTotalCalc() { no usages
    totalDeduction = (InitialAmount * deductionPercentage) / 100;

    // Cambiar IVA en funcion del code
    if (Integer.parseInt(code) < 10) {
        totalVAT = 0;
    } else {
        totalVAT = (float) (InitialAmount * 0.16);
    }

    billTotal = (InitialAmount - totalDeduction) + totalVAT;
}
```

-Después de la refactorización:

```
BillCalculator.java
package ejer2;

public class BillCalculator { 2 usages  @ RubenGGBC *

    private DeductionCalculator deductionCalculator; 2 usages
    private VATCalculator vatCalculator; 2 usages

    public BillCalculator() { 1 usage  @ RubenGGBC
        this.deductionCalculator = new DeductionCalculator();
        this.vatCalculator = new VATCalculator();
    }

    public void calculateTotals(Bill bill) { 1 usage  @ RubenGGBC *
        // Calcular deducción usando la clase especializada
        bill.totalDeduction = deductionCalculator.calculate(bill.InitialAmount, bill.deductionPercentage);

        // Calcular IVA usando la clase especializada
        if(Integer.parseInt(bill.code)<10) {
            bill.totalVAT=0;
        }
        else {
            bill.totalVAT = vatCalculator.calculate(bill.InitialAmount);
        }

        // Calcular total final
        bill.billTotal = (bill.InitialAmount - bill.totalDeduction) + bill.totalVAT;
    }
}
```

5. Asegúrate de que quede claro ¿en qué clases se hacen los cambios en cada caso? ¿Dónde están las responsabilidades?

-**Para cada cambio previo realizado se ha modificado única y exclusivamente una clase debido a que cada clase tiene una responsabilidad.

Si queremos cambiar algo de la estructura tenemos que cambiar la clase Bill ya que es la clase que se encarga de los objetos Bill.

Si queremos cambiar algo de las deducciones tenemos que cambiar DeductionCalculator, ya que es la única responsable de calcular las responsabilidades.

Si queremos cambiar algo en el IVA tenemos que cambiar VATCalculator, ya que es la encargada de calcular el IVA

Si queremos cambiar algo del cálculo de la Bill (por ejemplo añadir otra clase que calcule otra cosa) tenemos que cambiar la clase BillCalculator, ya que es la responsable de utilizar las otras 2 clases y calcular el valor total.**

3. Principio de Sustitución de Liskov (LSK)

Tenemos una clase `TransportationDevice` que cuya implementación es la siguiente:

```
abstract class TransportationDevice {  
    String name;  
    double speed;  
    Engine engine;  
  
    void startEngine() { ... }  
}
```

A continuación tenemos clases que heredan de esta `TransportationDevice`, como por ejemplo `Car`:

```
class Car extends TransportationDevice {  
    @Override  
    void startEngine() { ... }  
}
```

Aquí no hay ningún problema, la clase `Car` hereda de la clase abstracta `TransportationDevice` e implementa el método `startEngine`.

Vamos a crear otra clase `Bicycle` que también hereda de `TransportationDevice`:

```
class Bicycle extends TransportationDevice {  
    @Override  
    void startEngine() /*problem!*/  
}
```

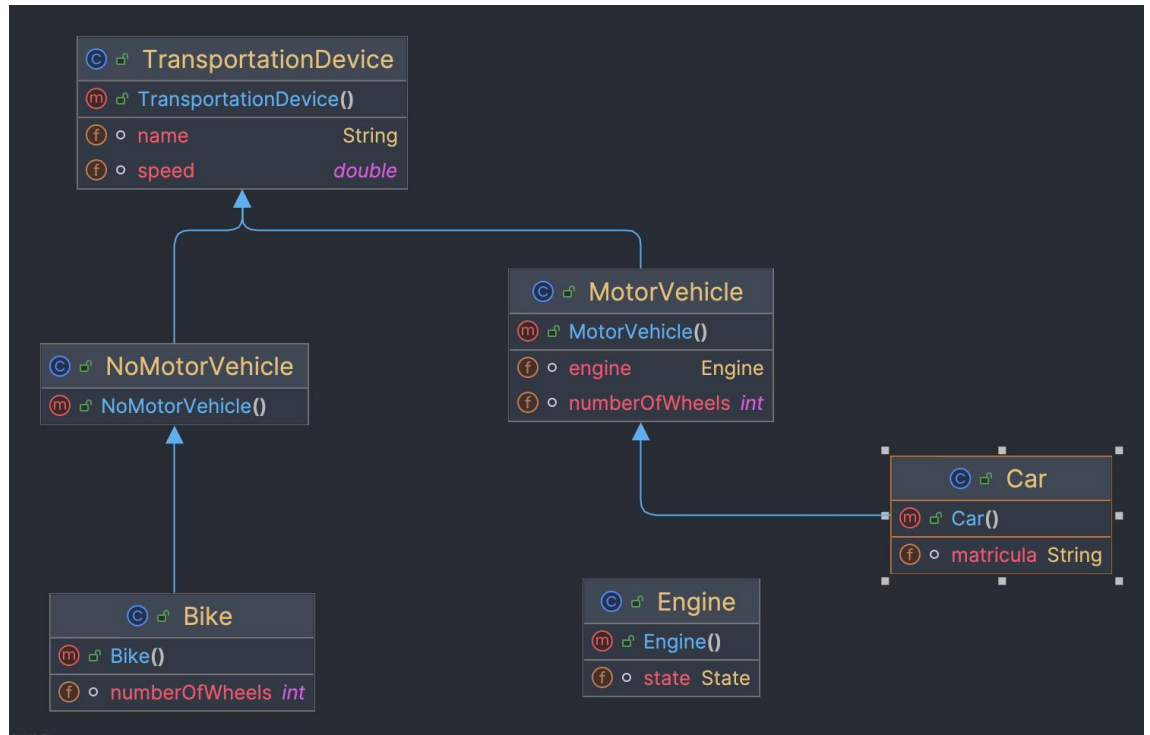
Como se puede observar, una bicicleta no tiene motor (al menos las tradicionales), y por lo tanto, no tiene sentido que implemente el método `startEngine`.

Tareas a realizar:

1. ¿Cómo rediseñarías la aplicación, para que ese cumpliese el principio de Liskov?. Diseña el modelo UML y reimplementa las clases.

-Haríamos que `transportationDevice` se quedara como está, pero que no tuviera el `startEngine()`. Luego crearíamos 2 clases más que hereden de `transportationDevice`: `VehicleMotor` y `NoMotorVehicle`, donde cada clase nos sirve para identificar el arranque de cada vehículo, los que tienen motor heredaran el `startEngine()` de `VehicleMotor` y los de `NoMotorVehicle` heredarían otro método que iniciaría su marcha.****

-Diagrama generado en IntelliJIdea:



-Class transportationDevice:

```
TransportationDevice.java

package ejer3;

public class TransportationDevice {
    String name; no usages
    double speed; no usages
}
```

-Class NoMotorVehicle:

```
NoMotorVehicle.java

package ejer3;

public class NoMotorVehicle extends TransportationDevice {
    public void empezar(){ no usages 2 RubenGGBC
    |
    |   System.out.println("No hay motor");
    |
    }
}
```

-Class MotorVechicle:

```
MotorVehicle.java

package ejer3;

public class MotorVehicle extends TransportationDevice{ 2 usages :
    Engine engine; 1 usage
    int numberOfWheels; no usages

    public void startEngine() { engine.state = Engine.State.ON; }
}
```


-Clase Bike:

```
package ejer3;

public class Bike extends NoMotorVehicle{
    int numberOfWheels; no usages
}
```

-Clase Car:

```
package ejer3;

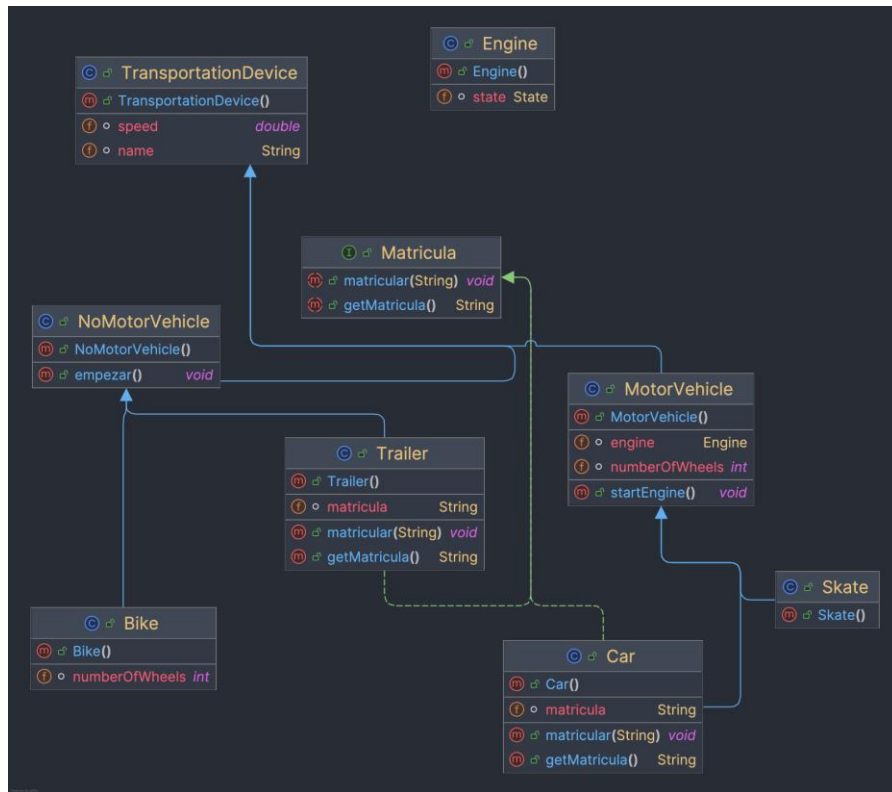
public class Car extends MotorVehicle implements Matricula{
    String matricula; 2 usages

    @Override no usages 2 RubenGGBC
    public void matricular(String matricula) {
        this.matricula = matricula;
    }

    @Override no usages 2 RubenGGBC
    public String getMatricula() {
        return this.matricula;
    }
}
```

2. Los Coches(Car) tienen motor y además tienen matricula y el método matricular, sin embargo, los patinetes (Skate) tienen motor pero no matricula. Algo similar pasa con los vehiculos sin motor: las bicicletas(Bike) no tienen matricula pero los remolques(Trailer) sí. ¿Cómo modelarías esta situación?

-Diagrama generado por IntelliJIdea:



4. Principio de Segregación de Interfaces (ISP)

Disponemos de la siguiente clase de contacto telefónico:

```
public class Contact {
    String name, address, emailAddress, 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 setEmailAddress(String ea) { emailAddress=ea; }
    public String getEmailAddress() { return emailAddress; }

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

}
```

y dos clases adicionales que envían correos electrónicos y SMS's tal y como se muestra a continuación (a través de métodos de la clase, static):

```
public class EmailSender {
    public static void sendEmail(Contact c, String message){
        //Envía un mensaje a la dirección de correo electrónico del
        // Contacto c.
    }
}

public class SMSSender {
    public static void sendSMS(Contact c, String message){
        //Envía un mensaje SMS al teléfono del Contacto c.
    }
}
```

Tareas a realizar:

1. Indica qué información necesitan las clases EmailSender y SMSSender de la clase Contact para realizar su tarea, y qué información recogen.

-**Tanto EmailSender como SMSSender reciben un objeto Contact, por lo cual obtienen toda la información de ese objeto.**

2. Justifica porque incumplen el principio ISP.

-**El problema es que tanto EmailSender como SMSSender reciben un objeto Contact, el cual contiene toda la información del contacto y ninguno de las 2 clases necesitan toda la información puesto que a EmailSender no le interesa el teléfono de contacto y a SMSSender no le interesa el correo electrónico**

3. Refactoriza las clases anteriores, sustituyendo el parámetro Contact, por una interfaz. Esta interfaz tendrá los métodos necesarios para acceder a la información que necesita. Modifica también la clase Contact.

-Clase Contact:

```
package ejer4;

public class Contact implements EmailAble, TelephoneAble{
    String name, address, emailAddress, 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 setEmailAddress(String ea) {
        emailAddress = ea;
    }

    @Override
    public String getEmailAddress() {
        return emailAddress;
    }

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

    @Override
    public String getTelephone() {
        return telephone;
    }
}
```

-Interfaces nuevas:

```
EmailAble.java  
  
package ejer4;  
  
public interface EmailAble {  
    String getEmailAddress();  
}
```

```
TelephoneAble.java  
  
package ejer4;  
  
public interface TelephoneAble {  
    String getTelephone(); no us  
}
```

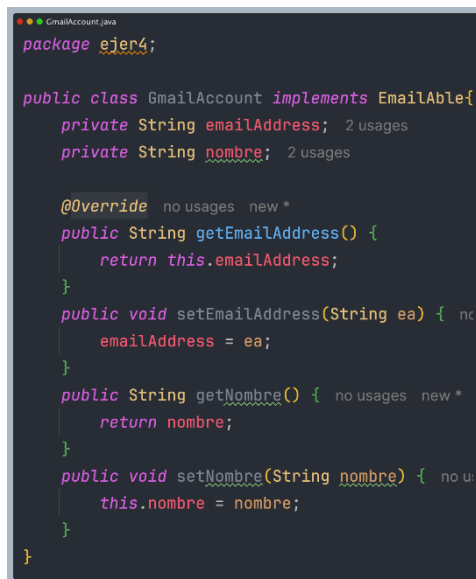
```
EmailSender.java  
  
package ejer4;  
  
public class EmailSender { no usages new *  
    public static void sendEmail(EmailAble e, String message) {  
  
    }  
}
```

```
SMSSender.java  
  
package ejer4;  
  
public class SMSSender { no usages new *  
    public static void sendSMS(TelephoneAble e, String message) {  
  
    }  
}
```

4. Completa la clase GmailAccount que podrá enviar mensajes por correo electrónico (clase EmailSender), pero no SMS por teléfono (clase SMSSender)

```
public class GmailAccount {  
    String name, emailAddress;  
}
```

-Clase GmailAccount:



```
GmailAccount.java  
package ejer4;  
  
public class GmailAccount implements EmailAble{  
    private String emailAddress; 2 usages  
    private String nombre; 2 usages  
  
    @Override no usages new *  
    public String getEmailAddress() {  
        return this.emailAddress;  
    }  
    public void setEmailAddress(String ea) { no  
        emailAddress = ea;  
    }  
    public String getNombre() { no usages new *  
        return nombre;  
    }  
    public void setNombre(String nombre) { no u  
        this.nombre = nombre;  
    }  
}
```

5. Crea un programa principal que permita invocar al método sendEmail de la clase EmailSender con un objeto de la clase GmailAccount.

-main:

```
package ejer4;

public class Main { new *
    public static void main(String[] args) { new *

        Contact contact = new Contact(
            nombre: "María García",
            direccion: "Avenida Libertad 456",
            mail: "maria.garcia@email.com",
            tel: "+34 600 987 654"
        );

        System.out.println("1. Usando Contact (implementa Emailable y SMSable):");
        EmailSender.sendEmail(contact, message: "Hola María!");
        SMSSender.sendSMS(contact, message: "SMS para María!");

        GmailAccount gmailAccount = new GmailAccount(
            nombre: "Carlos López",
            mail: "carlos.lopez@gmail.com"
        );

        System.out.println("2. Usando GmailAccount (solo implementa Emailable):");
        EmailSender.sendEmail(gmailAccount, message: "Email a cuenta de Gmail!");

        System.out.println("3. Usando polimorfismo:");
        Emailable[] emailables = {contact, gmailAccount};

        for (Emailable e : emailables) {
            EmailSender.sendEmail(e, message: "Email masivo!");
        }

    }
}
```

-Resultado de ejecutar el main:

```
1. Usando Contact (implementa Emailable y SMSable):
Email: maria.garcia@email.com
Mensaje: Hola María!
Email enviado exitosamente!

Teléfono: +34 600 987 654
Mensaje: SMS para María!
SMS enviado exitosamente!

2. Usando GmailAccount (solo implementa Emailable):
Email: carlos.lopez@gmail.com
Mensaje: Email a cuenta de Gmail!
Email enviado exitosamente!

3. Usando polimorfismo:
Email: maria.garcia@email.com
Mensaje: Email masivo!
Email enviado exitosamente!

Email: carlos.lopez@gmail.com
Mensaje: Email masivo!
Email enviado exitosamente!

Process finished with exit code 0
```

5. Principio de Inversión de dependencia (DIP)

Imaginemos que la clase Factura del ejercicio anterior la hubiésemos implementado de la siguiente forma:

```
public class Bill {
    public String code; // Representa un número de 5 dígitos
    public Date date;
    public float initialAmount;
    public float totalVAT;
    public float totalDeduction;
    public float billTotal;
    public int deductionPercentage;

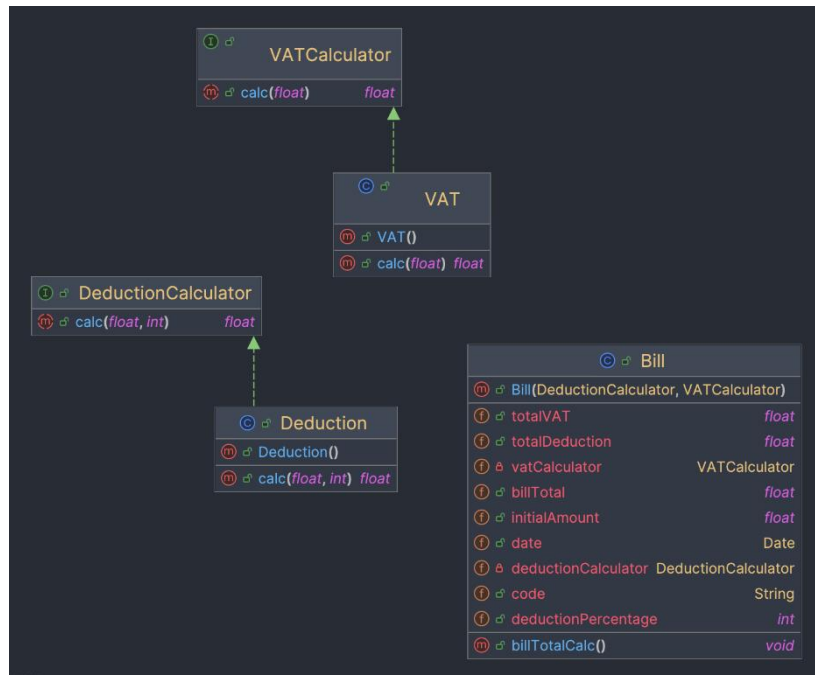
    // Método que calcula el total de la factura
    public void billTotalCalc() {
        // Calculamos la deducción
        Deduction d=new Deduction();
        totalDeduction = d.calc(initialAmount, deductionPercentage);
        // Calculamos el IVA
        VAT v=new VAT();
        totalVAT = v.cal(initialAmount);
        // Calculamos el total
        billTotal = (initialAmount - totalDeduction) + totalVAT;
    }
}
```

Tareas a realizar:

1. Justifica porqué no cumple el principio de Inversión de dependencia.

-**Bill depende de clases de más bajo nivel que ella como lo son Deduction y VAT, además dentro de la clase se usa new, por lo que hace que esten fuertemente acopladas ya que sin la clase Bill no se crea Deduction ni VAT**
2. Refactoriza el código para que cumpla el principio. Indica mediante un grafo de dependencias cómo están relacionadas ahora las clases/interfaces

-Grafo de dependencias de IntelliJ:



-Clase Bill:

```
package ejer5;

import ejer2.BillCalculator;

import java.util.Date;

public class Bill {
    public String code;
    public Date date;
    public float initialAmount;
    public float totalVAT;
    public float totalDeduction;
    public float billTotal;
    public int deductionPercentage;

    private DeductionCalculator deductionCalculator;
    private VATCalculator vatCalculator;

    public Bill(DeductionCalculator deductionCalculator, VATCalculator vatCalculator) {
        this.deductionCalculator = deductionCalculator;
        this.vatCalculator = vatCalculator;
    }

    public void billTotalCalc() {
        totalDeduction = deductionCalculator.calc(initialAmount, deductionPercentage);
        totalVAT = vatCalculator.calc(initialAmount);
        billTotal = (initialAmount - totalDeduction) + totalVAT;
    }
}
```

-Interfaz DeductionCalculator:

```
DeductionCalculator.java
package ejer5;

public interface DeductionCalculator { 3 usages 1 implementation
    float calc(float initialAmount, int deductionPercentage);
}
```

-Clase Deduction:

```
Deduction.java
package ejer5;

public class Deduction implements DeductionCalculator { no usages 2 Ru
    @Override 1 usage 2 RubenGGBC
    public float calc(float initialAmount, int deductionPercentage) {
        return (initialAmount * deductionPercentage) / 100;
    }
}
```

-Interfaz VATCalculator:

```
VATCalculator.java
package ejer5;

public interface VATCalculator { 3 u
    float calc(float initialAmount);
}
```

-Clase VAT:

```
VAT.java
package ejer5;

public class VAT implements VATCalculator { no usages 2 RubenGGBC
    @Override 1 usage 2 RubenGGBC
    public float calc(float initialAmount) { return (float) (initialAmount * 0.16); }
}
```