LABORATORIO 4
DISEÑO AVANZADO- PRINCIPIOS SOLID

Introducción

En esta actividad se proponen varios ejercicios para su resolución basándose en los principios SOLID.

Objetivos

El objetivo de este conjunto de ejercicios es revisar, estudiar y trabajar cada uno de los principios SOLID.

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 singInWithFB(log, pass);
		if (service.compareTo("google")==0)
			return singInWithGoogle(log, pass);
		if (service.compareTo("twitter")==0)
			return singInWithTwitter(log, pass);
		return false;
	}
	public boolean singInWithFB(String log, String pass) {
		//use the FB api
		return true;
	}
	public boolean singInWithGoogle(String log, String pass) {
		//use the google api
		return true;
	}
	public boolean singInWithTwitter(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.
2. ¿Consideras que el cambio que has realizado es una refactorización? Justifica la respuesta.

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 calculo 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.
2. 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:

```
si (importeFactura>35600)
   totalDeduction =
                    (InitialAmount * deductionPercentage +4.5) / 100;
  sino totalDeduction =
                         (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%.
4. 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.
5. Asegúrate de que quede claro ¿en qué clases se hacen los cambios en cada caso? ¿Dónde está las responsabilidades?

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 hererdan de eta `TransportationDevice`, como por ejemplo `Car`:

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

Aqui no hay ningún problema, la clase `Car` *hereda de la clase abstracta* `TransportationDevice` *e implementa en método* `startEngine`.

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

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

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

Cuestiones:

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

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.
2. Refactoriza el código para que cumpla el principio. Indica mediante un grafo de dependencias cómo están relacionadas ahora las clases/interfaces.

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.
2. Justifica porque incumplen el principio ISP.
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`.
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;
}
```

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