LABORATORIO SOLID

EUKEN SÁEZ JIMÉNEZ y MIKEL FAJARDO BONASTRE

INGENIERIA DEL SOFTWARE 2

LABORATORIO SOLID

INTRODUCCIÓN

Durante este laboratorio se van a analizar diferentes situaciones partiendo de unas líneas de código previamente proporcionado. Este código presenta diferentes violaciones de los principios SOLID.

Se han proporcionado diferentes problemas divididos en los cinco principios SOLID:

1. Principio Abierto-Cerrado (OCD)
2. Principio de Responsabilidad Única (SRP)
3. Principio de Sustitución de Liskov (LSK)
4. Principio de Segregación de Interfaces (ISP)

Para cada apartado se adjuntarán las soluciones propuestas acordes con los principios mencionados, justificando dichas resoluciones.

Código en GitHub : https://github.com/lekim3fb/LabSolid.git

1.PRINCIPIO ABIERTO-CERRADO (OCD)

Definición: El objetivo principal del OCP es evitar modificaciones directas al código existente cuando se requiere agregar nuevas funcionalidades, ya que las modificaciones pueden afectar el comportamiento previamente probado del sistema. En lugar de modificar el código base, se extiende mediante herencia, interfaces o composición.

Código proporcionado:

```
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;
	}

}
```

Resumen: Se presenta una clase de nombre ‘AuthService’, la cual proporciona una serie de métodos que permites a un usuario acceder a su cuenta personal proporcionando el servicio (en este caso la red social) al que quiere acceder, su nombre de usuario y contraseña.

Problema: Si observamos el principio abierto-cerrado del grupo de principios SOLID, este nos dice las entidades deber estar abiertas para extensión y cerradas para modificación, es decir, la capacidad de añadir nuevas funcionalidades, sin tener que modificar el código ya creado. En el caso propuesto, observamos como, si queremos añadir un nuevo servicio, como nos propone el ejercicio, como Apple, el código tiene que ser modificado para poder hacerlo posible.

Solución: La forma de corregir esto, es mediante el uso de interfaces y/o herencia. Con ello seremos capaces de añadir, como en este caso, un nuevo servicio, creando una nueva clase que herede los métodos o funcionalidades existentes. En este caso se ha creado una clase interfaz de nombre ‘IInitializable’ que funcionara como intermediario al cual las clases, que representan el servicio al que se quiere iniciar sesión, accederán para hacer uso del método de la clase principal, De esta forma será sencillo modificar nuestro código.

Cambios realizados:

```
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;
	}

}
```

```
package ocp;

public interface IInitializable {
    public Boolean getInit(String us, String pass);
}
```

2. PRINCIPIO DE RESPONSABILIDAD ÚNICA (SRP)

Definición: El Principio de Responsabilidad Única (SRP) es el primero de los cinco principios SOLID de la programación orientada a objetos. Este principio establece que una clase debe tener una, y solo una, razón para cambiar.

En otras palabras, una clase debe tener una única responsabilidad o propósito. Si una clase asume más de una responsabilidad, será más difícil de mantener, ya que un cambio en una de sus responsabilidades podría afectar a otras partes del código.

Problema:

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?

Solución:

1. Basándonos en la definición vamos a crear 2 nuevas clases que gestione de manera independiente los cálculos.

2. Cambiamos la responsabilidad del calculo a la nueva clase creada quitándole la responsabilidad a la clase Bill.

3. Se encapsula en la clase nueva la variable referente al IVA.

4. En la versión original habría que utilizar un condicional para que cumpla, mientras que en la refactorizada estaría ya aplicado.

5. Para que no viole el principio hay que evitar que, como en el original, haga cambios en la clase Bill y lo haga en la nueva clase que gestiona los cálculos.

3. PRINCIPIO DE SUSTITUCIÓN DE LISKOV (LSK)

Definición:

Código proporcionado:

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`:

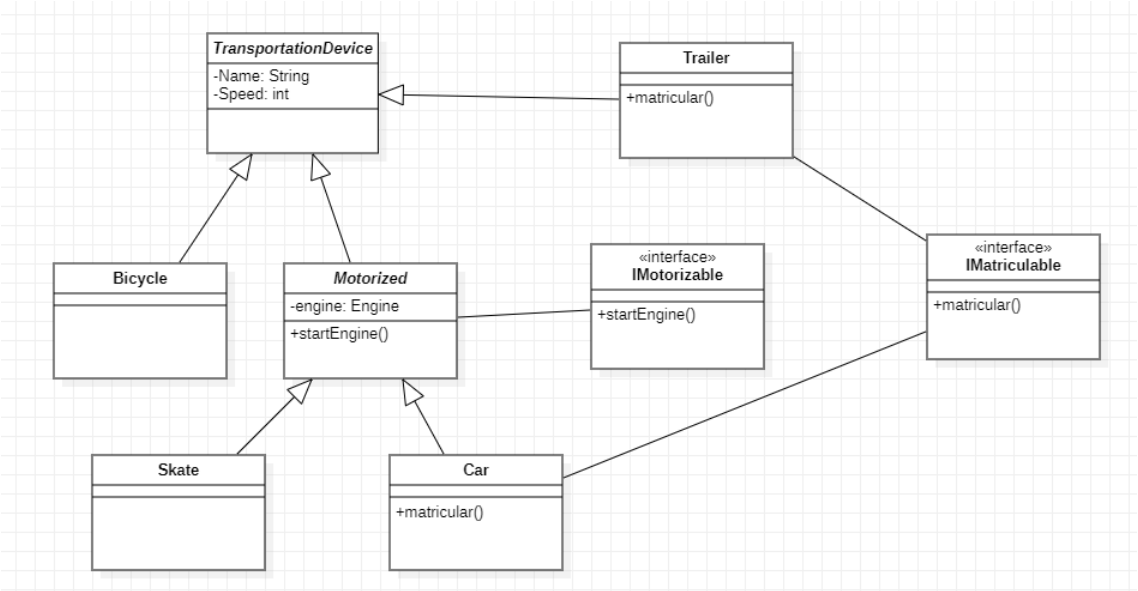
```
class Bicycle extends TransportationDevice {
   @Override
   void startEngine() /*problem!*/
}
```

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 que 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 vehículos sin motor: las bicicletas (Bike) no tienen matricula pero los remolques (Trailer) sí. ¿Cómo modelarías esta situación?



La implementación esta en el package LSK.

4- PRINCIPIO DE SEGREGACIÓN DE INTERFACES (ISP)

Definición:

Código proporcionado:

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 para 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.

Solución:

1) EmailSender necesita el correo electrónico (emailAddress) del objeto Contact que es accesible con el método getEmailAddress(). SMSSender necesita el número de teléfono (telephone) del objeto Contact que es accesible con el método getTelephone().
2) El Principio de Segregación de Interfaces establece que una clase no puede depender de interfaces que no usa. En este caso tenemos que EmailSender y SMSSender dependen de la clase Contact cuando solo necesitan el correo electrónico en el caso de EmailSender y el número de teléfono en el caso de SMSSender. Esto viola el principio ISP porque están obligadas a depender de una clase que contiene métodos y propiedades que no utilizan.

3,4,5 están en el paquete ISP.