Principios SOLID

Principio abierto-cerrado (OCP)

1. ¿Cumple la clase figuras con el principio OCP?

No, el principio de OCP nos dice que el código tiene que ser abierto para extensiones y cerrado para modificaciones, y en este caso para poder extender la clase Figuras con nuevos tipos de figuras, sería necesario modificar el código de la clase Figuras.

```
public class Figuras {
	Vector<Cuadrado> cuadrados=new Vector<Cuadrado>();
	Vector<Circulo> circulos= new Vector<Circulo>();
...
}
```

Podemos ver que en la clase figuras se utilizan dos vectores de figuras uno de tipo *Cuadrado* y otro de tipo Circulo. Si quisiéramos añadir un nuevo tipo de figura (por ejemplo Triangulo), tendríamos que modificar esta clase añadiéndole un nuevo vector con elementos de la nueva figura, añadiendo los métodos correspondientes a ese nuevo vector (addTriangulo() y extendiendo el método dibujarFiguras() para que pueda dibujar triángulos también).

```
public class Figuras {
	Vector<Cuadrado> cuadrados=new Vector<Cuadrado>();
	Vector<Circulo> circulos= new Vector<Circulo>();
	Vector<Triangulo> triangulos = new Vector<Triangulo>();
	public void addTriangulo(Triangulo c){
		triangulos.add(c);
	}
	public void dibujarFiguras(){
		...
		Enumeration<Triangulo> tris=triangulos.elements();
		Triangulo ct;
		while (tris.hasMoreElements()){
			ct=tris.nextElement();
			ct.dibujar();
		}
	}
...
}
```

Al extender la funcionalidad hemos tenido que modificar la clase Figuras, y esto rompe con el principio de OCP, ya que el diseño es abierto para extensiones y **cerrado para modificaciones.**

2. Modificar la clase para que cumpla con el principio OCP

Si abstraemos el tipo elementos que contiene la clase Figuras, esto nos permitira que los elementos puedan ser círculos, cuadrados u otros tipos de Figura que posteriormente queramos añadir. La clase Figura, puede pasar a ser una interfaz, y las clases Circulo, Cuadrado, Triangulo... implementarán esta interfaz. La clase Figuras quedaría así:

```
public class Figuras {
	Vector<Figura> figuras = new Vector<Figura> ();
	public void addFigura(Figura f){
		figuras.add(f);
	}
	public void dibujarFiguras(){
		Enumeration<Figura> figs = figuras.elements();
		Figura f;
		while(figs.hasMoreElements()){
			f=figs.nextElement();
			f.dibujar();
		}
	}
}
```

Ahora solo tenemos un único método para añadir figuras, y las figuras se tratan igual para dibujarlas, independientemente de su tipo.

La clase Figura pasará a ser una interfaz:

```
public interface Figura {
	public void dibujar();
}
```

Y las clases que la implementan, implementarán sus métodos como correspondan:

```
public class Circulo implements Figura{
	@Override
	public void dibujar() {
	}
}
public class Cuadrado implements Figura{
	@Override
	public void dibujar() {
	}
}
```

De esta manera si quisiéramos añadir un tipo de figura no tendríamos que realizar ninguna modificación en la clase Figuras, simplemente crearíamos la nueva clase e implementaríamos esa interfaz.

```
public class Triangulo implements Figura{
	@Override
	public void dibujar() {
	}
}
```

¿Consideras que la tarea realizada es una refactorización?

No es una refactorización, ya que hemos cambiado el comportamiento de la clase Figuras, es decir, hemos cambiado la signatura de los métodos.

Principio Liskov (LSK)

Programa principal que ejecuta los métodos de la clase Configuración

```
public class Configuracion {
...
  public void main(String args[]){
        Configuracion config = new Configuracion();
        config.cargarConfiguracion();
        config.salvarConfiguracion();
  }
}
```

¿Cumple la clase Configuración el principio OCP?

Aunque en el método *cargarConfiguración*, si queremos cargar todas las configuraciones, sí que tendríamos que añadir una nueva línea de código añadiendo al vector esa nueva configuración. Si cumple con el principio OCP, porque la clase está abierta para añadir nuevas funcionalidades sin tener que modificar el programa.

```
public class Configuracion {
      Vector<RecursoPersistente> conf=new Vector<RecursoPersistente>();
      ...}
```

Como tenemos un vector de *RecursosPersistentes* podemos añadir nuevos *Recursos*, ya que simplemente tendríamos que crear la nueva clase de configuración e implementar la interfaz RecursosPersistenes. Los métodos se llamarán mediante la interfaz así que simplemente los implementamos y luego llamamos a la interfaz para que los utilice, sin tener que aplicar cambios.

¿Cumple la clase Configuración con el principio Liskov?

No, ya que este principio dice que la herencia debe garantizar que cualquier propiedad probada para cualquier objeto de la superclase, se cumplir para cualquier objeto de las subclases.

```
public interface RecursoPersistente {
      public void load();
      public void save();}
public class ConfiguracionUsuario implements RecursoPersistente{
      public void load(){
            System.out.println("Configuracion usuario cargado");}
      public void save(){
            System.out.println("Configuracion usuario almacenado");}
}
public class ConfiguracionHoraria implements RecursoPersistente{
      public void load(){
            System.out.println("Configuracion horaria cargada");
      }
      public void save()  {
            System.out.println("ERROR, la hora no se puede almacenar,
es solo de lectura");
    }}
```

La clase `ConfiguracionUsuario` si cumple con este principio, ya que la configuración de usuario si se puede cargar y también se puede guardar, pero en el caso de la `ConfiguracionHoraria`, está si se puede cargar, pero **NO se puede guardar** luego la propiedad que se pueda guardar que nos exige el método save() que nos obliga a implementar la interfaz no se cumple en esta clase y **estaríamos incumpliendo el principio de Liskov.**

Uno de los heurísticos del principio LSP nos dice que **no puede haber una subclase** que sobrescriba una superclase **con un método en el que no hace nada** o lanza una **excepción** diciendo que no puede hacer nada (como es el caso del método save de la configuración horaria).

Refactorizar la aplicación para que cumpla el principio de Liskov.

Lo primero que haremos será extraer esas propiedades de cargar y guardar en dos interfaces diferentes.

Una para la propiedad de las configuraciones que **se pueden cargar**

```
public interface ILoadRecursoPersistente {
	public void load();
}
```

Y otra para la propiedad de las configuraciones que **se pueden guardar**

```
public interface ISaveRecursoPersistente {
	public void save();
}
```

Luego hacemos que cada clase implemente las interfaces de las funcionalidades que le corresponden.

```
public class ConfiguracionSistema implements
ILoadRecursoPersistente, ISaveRecursoPersistente{
	public void load(){
		System.out.println("Configuracion sistema cargada");
	}
	public void save(){
		System.out.println("Configuracion sistema almacenada");
	}
}
public class ConfiguracionUsuario implements
ILoadRecursoPersistente, ISaveRecursoPersistente{
	public void load(){
		System.out.println("Configuracion usuario cargado");
	}
	public void save(){
		System.out.println("Configuracion usuario almacenado");
	}
}
```

De esta forma las clases `ConfiguracionUsuario` y `ConfiguracionSistema` tendrán implementada la propiedad de que se pueden guardar y se pueden cargar ya que implementan ambas funcionalidades, en cambio la clase `ConfiguracionHoraria` solo tendrá la propiedad de que se puede cargar porque solo implementa dicha interfaz

```
public class ConfiguracionHoraria implements ILoadRecursoPersistente{
      public void load(){
            System.out.println("Configuracion horaria cargada");
      }
}
```

Por último en la clase configuración tendremos dos vectores para las diferentes propiedades. De esta manera, solo guardaremos aquellas configuraciones que se puedan guardar y solo cargaremos las que se puedan cargar.

```
public class Configuracion {
  Vector<ILoadRecursoPersistente> lrp = new Vector<ILoadRecursoPersistente>();
  Vector<ISaveRecursoPersistente> srp = new Vector<ISaveRecursoPersistente>();
  public void cargarConfiguracion(){
        lrp.add(new ConfiguracionHoraria());
        lrp.add(new ConfiguracionUsuario());
        lrp.add(new ConfiguracionSistema());

        for (ILoadRecursoPersistente irp:lrp)
              irp.load();
  }
  public void salvarConfiguracion(){
        for (ISaveRecursoPersistente isrp:srp)
              i.save();
  }
}
```

En el método para cargar las configuraciones mirará en el vector de las configuraciones que se pueden cargar y las irá cargando y el de guardar las configuraciones hará lo mismo.

Principio de Responsabilidad Única (SRP)

Refactorizar la aplicación

```
public class Factura {
	public String _codigo;
	public Date fechaEmision;
	public float importeFactura;
	public float importeIVA;
	public float importeDeduccion;
	public float importeTotal;
	public int porcentajeDeduccion;

	// Método que calcula el total de la factura
	public void calcularTotal() {
		// Calculamos la deducción
		importeDeduccion = (importeFactura * porcentajeDeduccion) / 100;
		// Calculamos el IVA
		importeIVA = (float) (importeFactura * 0.16);
		// Calculamos el total
		importeTotal = (importeFactura - importeDeduccion) + importeIVA;
	}
}
```

El principio de responsabilidad única, nos dice que cada clase debe de tener una única responsabilidad y un único motivo por el que pueda ser modificada. En nuestra clase podemos ver que hay tres responsabilidades (**calcularTotal**, **calcularIVA** **y** **calcularDeduccion**) y por tanto también tendremos tres motivos por los que la clase podrá ser modificada (por ejemplo que cambien la forma de calcular el IVA o la forma de calcular la deducción, o que al total le queramos añadir un descuento o sumarle una cantidad).

Para refactorizar este problema, lo que haremos será extraer dos de las responsabilidades y encapsularlas en una nueva clase cuya única responsabilidad sea esa. Para ello primero vamos a crear una nueva clase CalcularDeduccion y extraeremos ahí la funcionalidad correspondiente a calcular la deducción.

```
public class CalcularDeduccion {
	private float porcentajeDeduccion;

	public CalcularDeduccion(float pPorcentajeDeduccion){
		porcentajeDeduccion = pPorcentajeDeduccion;
	}

	public float calculaDeduccion(float importeFactura){
		return   (importeFactura * porcentajeDeduccion) / 100;
	}
}
```

Y repetimos el mismo proceso para la funcionalidad de calcular el IVA.

```
public class CalcularIVA {
	private static final double IVA = (16/100);

	public CalcularIVA(){
	}

	public float calculaIVA(float importeFactura){
		return (float) (importeFactura * IVA);
	}
}
```

Una vez que hemos extraído esas responsabilidades y las hemos encapsulado en clases diferentes, lo que tenemos que hacer en la clase original es llamar a esas clases para usar sus métodos.

```
public class Factura {
	public String _codigo;
	public Date fechaEmision;
	public float importeFactura;
	public float importeIVA;
	public float importeDeduccion;
	public float importeTotal;
	public int porcentajeDeduccion;

	public void calcularTotal(){
	  //Calculamos la deduccion
	  CalcularDeduccion cd = new CalcularDeduccion(porcentajeDeduccion);
	  importeDeduccion = cd.calculaDeduccion(importeFactura);
	  //Calculamos el IVA
	  CalcularIVA ci = new CalcularIVA();
	  importeIVA = ci.calculaIVA(importeFactura);
	  //Calculamos el total
	  importeTotal = (importeFactura - importeDeduccion) + importeIVA;

	}
}
```

Cambios si el importeDeduccion se calculase en base al importe de la factura

Al aplicar la refactorización, este cambio solo afecta a la clase que tiene que cambiar cuando se realicen cambios relacionados con el cálculo de la deducción.

```
public class CalcularDeduccion {
	private float porcentajeDeduccion;
	...
	public float calculaDeduccion(float importeFactura){
		if(importeFactura > 1000){
			return	(importeFactura * porcentajeDeduccion+3) / 100;
		}
		else{
			return	(importeFactura * porcentajeDeduccion) / 100;
		}
	}}
```

Si no hubiésemos aplicado la refactorización...

```
public void calcularTotal() {
       // Calculamos la deducción
       if(importeFactura > 1000)
           importeDeduccion = (importeFactura * porcentajeDeduccion+3) / 100;
         else
              importeDeduccion = (importeFactura * porcentajeDeduccion) / 100;
              ...
}
```

Si no hubiésemos realizado la refactorización, este cambio se haría en la clase Factura complicado aún más el programa y añadiéndole más funcionalidades y responsabilidades.

Cambios si cambia el IVA del 16% al 18%

Al haber extraído la funcionalidad del IVA en una nueva clase, el único cambio que tendremos que realizar será en esta nueva clase

```
public class CalcularIVA {
       private static final double IVA = (18/100);
       ...
}
```

En cambio, si no hubiésemos aplicado la refactorización, este cambio afectaría a la clase Factura.

```
public class Factura {
       ...
       public void calcularTotal() {
              ...
              // Calculamos el IVA
              importeIVA = (float) (importeFactura * 0.18);
              ...
       }
}
```

Cambios a realizar si a las facturas de código 0 no se les aplicase el IVA

En este caso el cambio a realizar si es un cambio de una responsabilidad de la clase Factura, por lo que tanto en el código original como en el código refactorizado este cambio se realiza en la clase Factura.

```
public class Factura {
      // Método que calcula el total de la factura
      public void calcularTotal() {
        ...
        // Calculamos el total
        if(_codigo.equalsIgnoreCase("0"))
            importeTotal = (importeFactura - importeDeduccion);
        else
           importeTotal = (importeFactura - importeDeduccion) + importeIVA;
      }
}
```

En este caso, tanto si hemos aplicado la refactorización como si no, el cambio que hay que hacer es el mismo, pero en los otros dos anteriores, si aplicamos la refactorización, no es necesario cambiar la clase Factura si cambia el IVA o la deducción, por lo que la clase Factura solo tendrá un único motivo de cambio (siguiendo el principio SRC), al igual que las clases de CalcularIVA y CalcularDeduccion. En cambio si no aplicamos esta refactorización, habrá que hacer cambios en la clase Factura por tres motivos diferentes, yendo en contra del principio de responsabilidad única.

Principio de Inversión de dependencia (DIP).

```
public class Factura {
   public String _codigo;
   public Date fechaEmision;
   public float importeFactura;
   public float importeIVA;
   public float importeDeduccion;
   public float importeTotal;
   public int porcentajeDeduccion;

   // Método que calcula el total de la factura
   public void calcularTotal() {
      // Calculamos la deducción
      importeDeduccion = (importeFactura * porcentajeDeduccion) / 100;
      // Calculamos el IVA
      importeIVA = (float) (importeFactura * 0.16);
      // Calculamos el total
      importeTotal = (importeFactura - importeDeduccion) + importeIVA;
   }
}
```

¿Cumple el principio de inversión de dependencia?

No, la clase Factura tiene responsabilidades de otras clases de bajo nivel y **una superclase no debe conocer ninguna de sus subclases.** En este caso las funcionalidades de las clases CalcularIVA y CalcularDeduccion aparecen en la clase Factura.

```
public class Factura {
   public String _codigo;
        ...
   public void calcularTotal(){
      CalcularDeduccion cd = new CalcularDeduccion(porcentajeDeduccion);
      importeDeduccion = cd.calculaDeduccion(importeFactura);
      CalcularIVA ci = new CalcularIVA();
      importeIVA = ci.calculaIVA(importeFactura);
      importeTotal = (importeFactura - importeDeduccion) + importeIVA;
   }
}
```

En este caso una clase de más alto nivel como Factura está dependiendo de las clases de CalcularIVA y CalcularDeduccion. Como la clase de más alto nivel depende de otras de más bajo nivel, si quisiéramos, no podríamos extraer esta clase y reutilizarla en otras aplicaciones, porque tiene dependencias con las otras clases.

Refactorización del código para que cumpla el principio DIP

(Para esta refactorización partimos del ejemplo anterior en el que ya se habían extraído las funcionalidades que no le correspondían a la clase Factura en otras dos clases CalcularIVA y CalcularDeduccion).

Lo que haremos para solucionar este problema, es aplicar un heurístico que nos dice que hay que diseñar para la interfaz y no para la implementación, por lo que utilizaremos la herencia y las interfaces para evitar los enlaces directos entre las clases, ya que las abstracciones no deben de depender de los detalles.

Para ello lo primero que haremos será **extraer dos interfaces** para esas clases de más bajo nivel de las que depende la clase Factura, de forma que ahora dependerá de esas Interfaces y no de las clases. De esta forma cuando apliquemos cambios en las clases CalcularIVA y CalcularDeduccion no habrá que realizar cambios en la clase Factura

```
public interface IDeduccion {
      public float calculaDeduccion( float importeFactura);
}

public interface IIVA {
      public float calculaIVA(float importeFactura);
}
```

Finalmente para elimiar la dependencia con las clases concretas, cuando creamos un objeto de la clase Factura, en el constructor le indicamos cuáles con los objetos Ideduccion y IIVA que debe utilizar. Si estos objetos pueden variar, se podrían mover los parámetros del método `calcularTotal()`.

```
public class Factura {
      ...
      private IDeduccion cd;
      private IIVA ci;

      public Factura(IDeduccion pCd, IIVA pCi){
            cd = pCd;
            ci = pCi;
      }
      public void calcularTotal(){
      //Calculamos la deduccion
      importeDeduccion = cd.calculaDeduccion(importeFactura);
      //Calculamos el IVA
      importeIVA = ci.calculaIVA(importeFactura);
      //Calculamos el total
      importeTotal = (importeFactura - importeDeduccion) + importeIVA;
      }
```

De esta forma hemos conseguido hacer que los módulos de alto nivel no dependan de los de bajo nivel de manera que las abstracciones no dependen de los detalles, por lo que si realizamos algún cambio en esos módulos de bajo nivel no se verá afectado el de mayor nivel.

Principio de Segregación de Interfaces (ISP)

¿Qué información necesitan las clases EmailSender y SMSSender de la clase Contacto para realizar su tarea, y qué información recogen?

A la clase EmailSender únicamente le interesará la información del email del contacto, y a la clase SMSSender solo le interesará el número de teléfono del contacto.

¿Incumplen el principio (ISP)?

Sí, incumple el principio, porque estamos obligando a las clases EmailSender y SMSSender a depender de la interfaz de Contacto, cuando realmente solo utilizan una propiedad de esta clase (email y telephone respectivamente).

Refactorización de las clases

El principio (ISP) nos dice que varias interfaces específicas son mejores que una de propósito general, por lo que lo primero que haremos, será extraer de contacto dos interfaces más específicas para el teléfono y el email

```
public interface ITelephone {
      public void setTelephone(String telephone);
      public String getTelephone ();
}
public interface IEmail {
      public void setEmailAddress(String email);
      public String getEmailAddress();
}
```

Luego hacemos que la clase de propósito general (en este caso Contacto) implemente dichas interfaces, para que así escriba su código.

```
public class Contacto implements IEmail, ITelephone{
      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; }
}
```

Finalmente modificamos las clases de SMSSender y EmailSender para que estas no dependan de la clase Contacto si no de las interfaces específicas que hemos definido.

```
public class SMSSender {
       public static void sendSMS(ITelephone s, String message){
            //Envia un mensaje SMS al telefono ITelephone s.
       }
}

public class EmailSender {
      public static void sendEmail(IEmail e, String message){
            //Envia un mensaje la direccion IEmail e.
      }
}
```

La clase GmailAcount

Para que la clase GmailAccount pueda enviar mensajes, esta tendrá que implementar la interfaz de IEmail, de manera que cuando le pasemos un objeto instanciado como GmailAcount también pueda enviar emails.

```
public class GmailAccount implements IEmail{
      String name, emailAddress;
      @Override
      public void setEmailAddress(String email) {
            // TODO Auto-generated method stub
      }
      @Override
      public String getEmailAddress() {
            // TODO Auto-generated method stub
            return null;
      }
}
```

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

Como hemos hecho que la clase GmailAccount implemente la interfaz IEmail, podemos pasarle al método sendEmail de la clase EmailSender un objeto de la clase GmailAccount para que envié un email.

```
public void main(String args[]){
            GmailAccount gmail = new GmailAccount();
            gmail.setEmailAddress(cuentaGmail@gmail.com);
            EmailSender.sendEmail(gmail, "hola");
      }
```