Taller principios SOLID:

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

1) Principio Abierto-Cerrado (OCP).

Dadas las siguientes clases (algunas clases están incompletas):

```
public class Figura {
public void dibujar(){}; }
public class Cuadrado extends Figura{}
public class Circulo extends Figura{}

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

        public void addCirculo(Circulo c){
                circulos.add(c);
        }
        public void addCuadrado(Cuadrado c){
                cuadrados.add(c);
        }
        public void dibujarFiguras(){
                Enumeration<Cuadrado> cuads=cuadrados.elements();
                Cuadrado c;
                while (cuads.hasMoreElements()){
                        c=cuads.nextElement();
                        c.dibujar();
                }
                Enumeration<Circulo> circs=circulos.elements();
                Circulo ci;
                while (cuads.hasMoreElements()){
                        ci=circs.nextElement();
                        ci.dibujar();
                }
        }
}
```

SOLUCIÓN PUNTO 1

1. ¿Cumple la clase Figuras el Principio OCP. Justifica tu respuesta.

1. La clase Figuras no cumple el principio OCP, ya que no está abierta a la extensión y cerrada a la modificación. La razón es que cada vez que se agregue una nueva figura (por ejemplo, un Triángulo), se tendría que modificar la clase Figuras añadiendo un nuevo método addTriangulo.

2. En caso de que no lo cumpla, modifica las clase para cumpla este criterio.

2. Una posible solución para cumplir el principio OCP sería crear una nueva interfaz llamada "Figura", la cual tendría un método "dibujar". Luego, la clase Figuras tendría una lista de objetos de esta interfaz en lugar de dos vectores de objetos de las clases Cuadrado y Círculo. De esta manera, se podrían añadir nuevas figuras sin necesidad de modificar la clase Figuras.

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

public class Cuadrado implements Figura {
public void dibujar() {
// Implementación del dibujo del Cuadrado
}
}

public class Circulo implements Figura {
public void dibujar() {
// Implementación del dibujo del Círculo
}
}

public class Figuras {
List<Figura> figuras = new ArrayList<Figura>();
```

3. ¿Consideras que la tarea realizada es una refactorización? Justifica tu respuesta.

3. Sí, se podría considerar como una refactorización, ya que se ha reestructurado el código para mejorar su calidad sin alterar su funcionalidad. En este caso, se ha modificado la clase Figuras para cumplir el principio OCP, lo que permite una mayor flexibilidad en el futuro a la hora de añadir nuevas figuras sin tener que modificar la clase existente.

2) Principio Liskov (LSK).

Tenemos una interfaz que recoge el comportamiento de los objetos que pueden cargarse en memoria y posteriormente guardarse de forma persistente:

```
public interface RecursoPersistente {
 public void load();
 public void save();
}
```

y 3 clases que implementan dicha interfaz:

```
public class ConfiguracionSistema implements RecursoPersistente{
        public void load(){
                System.out.println("Configuracion sistema cargada");
        }
        public void save(){
                System.out.println("Configuracion sistema almacenada");
        }
}

public class ConfiguracionUsuario implements RecursoPersistente{
        public void load(){
                System.out.println("Configuracion usuario cargada");
        }
        public void save(){
                System.out.println("Configuracion usuario almacenada");
        }
}

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");
  }
}
```

De manera que tenemos una clase Configuracion, que es responsable de cargar todas las configuraciones disponibles y posteriormente almacenarlas, tal y como se muestra en la siguiente clase:

```
public class Configuracion {
        Vector<RecursoPersistente> conf=new Vector<RecursoPersistente>();
 public void cargarConfiguracion(){
         conf.add(new ConfiguracionSistema());
         conf.add(new ConfiguracionUsuario());
```

```
            conf.add(new ConfiguracionHoraria());

            for (Iterator<RecursoPersistente> i = conf.iterator(); i.hasNext(); )
                  i.next().load();
}
public void salvarConfiguracion(){
            for (Iterator<RecursoPersistente> i = conf.iterator(); i.hasNext(); )
                  i.next().save();
}
}
```

Consultas:

1. Crea un programa principal que ejecute los métodos de la clase Configuración.

```
public class Ejecuta{
    public static void main(String args[]){
        Configuracion c = new Configuracion();
        c.cargaConfiguracion();
        c.salvarConfiguracion();
    }
}
```

2. Cumple la clase Configuracion en Principio OCP. Justifica la respuesta.
   No cumple, ya que no existen clases abstractas configuracionSistema y configuracionusuario
3. Cumple la clase Configuración el Principio de Liskov. Justifica la respuesta.
   Si cumple, ya que todos las clases que implementan ConfiguracionPersistente pueden usar los métodos de las clases heredadas
4. Refactoriza la aplicación para que cumpla el principio de Liskov. La solución a este ejercicio lo puedes encontrar en:./
5. Explica de forma general (independientemente del ejemplo) cual es el problema y la solución propuesta.

3) Principio de Responsabilidad Única (SRP).

Dada la siguiente clase Factura:

```
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;
            }
```

}

podríamos decir que la responsabilidad de esta clase es la de calcular 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. Si nos fijamos detenidamente en la implementación del método calcularTotal, podremos ver que, además de calcular el importe base de la factura, se está aplicando sobre el importe a facturar un descuento o deducción y un 16% de IVA. El problema está en que si en el futuro tuviéramos que modificar la tasa de IVA, o bien tuviéramos que aplicar una deducción en base al importe de la factura, tendríamos que modificar la clase Factura por cada una de dichas razones; por lo tanto, con el diseño actual las responsabilidades quedan acopladas entre sí, y la clase violaría el principio SRP.

Ejercicio propuesto:

1. Refactoriza la aplicación para que cada responsabilidad quede aislada en una clase. Indica qué cambios tendrías que realizar si el importeDeduccion se calculase en base al importe de la factura:

Si (importeFactura>10000)
importeDeduccion = (importeFactura * porcentajeDeduccion+3) / 100; sino
importeDeduccion = (importeFactura * porcentajeDeduccion) / 100;

RTA:

```
public float calcular(float importeFactura, int porcentajeDeduccion) {
if (importeFactura > 10000) {
return (importeFactura * porcentajeDeduccion + 3 + importeFactura * 0.02f) / 100;
} else {
return (importeFactura * porcentajeDeduccion + importeFactura * 0.02f) / 100;
}
}
}
```

2. Indica los cambios que tendrías que realizar si el IVA cambiase del 16 al 18%.

RTA:

```
IVA iva = new IVA();
iva.setTasaIVA(0.18f);
```

3. Indica los cambios que tendrías que realizar si a las facturas de código 0, no se le aplicase el IVA.

RTA:

Si a las facturas de código 0 no se les aplica el IVA, podríamos añadir un atributo booleano a la clase Factura que indique si se debe aplicar el IVA o no. Luego, en la función calcularTotal, sólo se aplicaría el IVA si el atributo está en true:

```
public class Factura {
public String codigo;
public Date fechaEmision;
public float importeFactura;
public float importeIVA;
public float importeDeduccion;
public float
```

4) Principio de Inversión de dependencia (DIP).

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

```
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
         Deduccion d=new Deduccion();
         importeDeduccion = d.calculaDeduccion(importeFactura, porcentajeDeduccion);
        Iva iva=new Iva();
         // Calculamos el IVA
         importeIVA = iva.calculaIva(importeFactura);
         // Calculamos el total
          importeTotal = (importeFactura - importeDeduccion) + importeIVA;
        }
}
```

Consultas:

SOLUCIÓN PUNTO 4

1.Cumple el principio de Inversión de dependencia.
Justifica la respuesta.

No cumple con el principio de Inversión de dependencia ya que la clase Factura depende directamente de las clases Deducción e IVA, lo que significa que cualquier cambio en estas clases podría afectar directamente a la clase Factura.

2. En caso negativo, refactoriza el código para que cumpla el principio.

```
public class Factura {

    public String codigo;
    public Date fechaEmision;
    public float importeFactura;
    public float importeIVA;
    public float importeDeduccion;
    public float importeTotal;
    public int porcentajeDeduccion;

    private CalculadorDeduccion calculadorDeduccion;
    private CalculadorIVA calculadorIVA;

    public Factura(CalculadorDeduccion calculadorDeduccion, CalculadorIVA
calculadorIVA) {
        this.calculadorDeduccion = calculadorDeduccion;
        this.calculadorIVA = calculadorIVA;
    }

// Métodos
// calcula el total de la factura
    public void calcularTotal() {
// Calcula la deducción
        importeDeduccion = calculadorDeduccion.calculaDeduccion(importeFactura,
porcentajeDeduccion);
// Calcula el IVA
        importeIVA = calculadorIVA.calculaIVA(importeFactura);
// Calcula el total
```

```
            importeTotal = (importeFactura - importeDeduccion) + importeIVA;
        }
    }
```

5) Principio de Segregación de interfaces (ISP).

Disponemos de la siguiente clase Contacto:

```
public class Contacto {
 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:

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

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

SOLUCIÓN PUNTO 5

1. **¿Qué información necesitan las clases EmailSender y SMSSender de la clase Contacto para realizar su tarea, y qué información recogen? Consideras que incumplen en principio ISP.**

- Las clases EmailSender y SMSSender necesitan la información de la dirección de correo electrónico y el número de teléfono respectivamente para realizar su tarea, y la información que recogen es el mensaje que se va a enviar. Sí, estas clases incumplen el principio ISP ya que dependen de métodos que no utilizan en su tarea.

2. **Refactoriza las clases anteriores, sustituyendo el parámetro Contacto, por una interfaz. Esta interfaz tendrá los métodos necesarios para acceder a la información**

que necesita en método. Modifica también la clase Contacto.

```
public interface CorreoElectronico {
   public String getEmailAddress();
}

public interface SMS {
   public String getTelephone();
}

public class Contacto implements CorreoElectronico, SMS {
   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; }
}

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

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

3. Piensa que después de refactorización, la clase GmailAccount (con alguna modificación) podrá ser enviada a la clase EmailSender pero no a la clase SMSSender.

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

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