Ejercicios Principios SOLID

Ejercicios principios SOLID

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

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();
        }
    }
}
```

Consultas:

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

   El Principio de Abierto/Cerrado (OCP) establece que las clases deben estar abiertas para la extensión pero cerradas para la modificación. Esto significa que debería poder extender el comportamiento de una clase sin modificar su código fuente.

   En la clase `Figuras`, se viola el Principio OCP porque si queremos agregar una nueva figura (por ejemplo, un `Triangulo`), tendríamos que modificar la clase `Figuras` para agregar un nuevo vector y métodos para manejar esta nueva figura. Esto implica modificar el código existente, lo cual va en contra del OCP.

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

   Para cumplir con el Principio de Abierto/Cerrado (OCP), se necesita diseñar las clases de manera que podamos extender su funcionalidad sin modificar el código existente. En este caso, se puede utilizar una lista genérica de `Figura` en lugar de listas específicas para cada tipo de figura. Esto nos permite agregar nuevas figuras sin modificar la clase `Figuras`.

   ```
   import java.util.Vector;
   import java.util.Enumeration;

   public class Figura {
       public void dibujar() {
           // Implementación por defecto
       }
   ```

```
}

public class Cuadrado extends Figura {
    @Override
    public void dibujar() {
        System.out.println("Dibujando un cuadrado");
    }
}

public class Circulo extends Figura {
    @Override
    public void dibujar() {
        System.out.println("Dibujando un círculo");
    }
}

public class Figuras {
    private Vector<Figura> figuras = new Vector<Figura>
();

    public void addFigura(Figura f) {
        figuras.add(f);
    }

    public void dibujarFiguras() {
        Enumeration<Figura> figs = figuras.elements();
        while (figs.hasMoreElements()) {
            Figura f = figs.nextElement();
            f.dibujar();
        }
    }
}
```

Para agregar una nueva figura, como `Triangulo`, simplemente creamos una nueva clase que extiende `Figura` y la agregamos a la lista de `figuras`:

```
// Clase Triangulo que extiende Figura
public class Triangulo extends Figura {
```

```
    @Override
    public void dibujar() {
        System.out.println("Dibujando un triángulo");
    }
}

// En el código principal
public class Main {
    public static void main(String[] args) {
        Figuras figuras = new Figuras();
        figuras.addFigura(new Cuadrado());
        figuras.addFigura(new Circulo());
        figuras.addFigura(new Triangulo());
        figuras.dibujarFiguras();
    }
}
```

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

   Sí, la tarea realizada es una refactorización. La refactorización es el proceso de reestructurar el código existente sin cambiar su comportamiento externo. En este caso, he reestructurado la clase `Figuras` para que utilice una lista genérica de `Figura` en lugar de listas específicas para cada tipo de figura. Esto mejora la flexibilidad y la extensibilidad del código sin cambiar su comportamiento externo, lo cual es una característica clave de la refactorización.

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 RecursoPersist
ente {
    public void load() {
        System.out.println("Configuracion sistema cargad
a");
    }

    public void save() {
        System.out.println("Configuracion sistema almacenad
a");
    }
}

public class ConfiguracionUsuario implements RecursoPersist
ente {
    public void load() {
        System.out.println("Configuracion usuario cargad
a");
    }

    public void save() {
        System.out.println("Configuracion usuario almacenad
a");
    }
}

public class ConfiguracionHoraria implements RecursoPersist
ente {
    public void load() {
        System.out.println("Configuracion horaria cargad
a");
    }

    public void save() {
        System.out.println("ERROR, la hora no se puede alma
cenar, 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<RecursoPer
sistente>();

    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 Main {
    public static void main(String[] args) {
        Configuracion configuracion = new Configuracion
();
```

```
        // Cargar configuraciones
        System.out.println("Cargando configuraciones:");
        configuracion.cargarConfiguracion();

        // Salvar configuraciones
        System.out.println("\nAlmacenando configuracione
s:");
        configuracion.salvarConfiguracion();
    }
}
```

Codigo Completo

```
import java.util.Vector;
import java.util.Iterator;

// Excepción personalizada
public class ConfiguracionNoAlmacenableException extends
Exception {
    public ConfiguracionNoAlmacenableException(String me
ssage) {
        super(message);
    }
}

// Interfaz RecursoPersistente
public interface RecursoPersistente {
    void load();
    void save() throws ConfiguracionNoAlmacenableExcepti
on;
}

// Clase ConfiguracionSistema
public class ConfiguracionSistema implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion sistema cargad
a");
```

```
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        System.out.println("Configuracion sistema almace
nada");
    }
}

// Clase ConfiguracionUsuario
public class ConfiguracionUsuario implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion usuario cargad
a");
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        System.out.println("Configuracion usuario almace
nada");
    }
}

// Clase ConfiguracionHoraria
public class ConfiguracionHoraria implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion horaria cargad
a");
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        throw new ConfiguracionNoAlmacenableException("E
RROR, la hora no se puede almacenar, es solo de lectur
a");
    }
}

// Clase Configuracion
```

```
public class Configuracion {
    Vector<RecursoPersistente> conf = new Vector<Recurso
Persistente>();

    public void cargarConfiguracion() {
        conf.add(new ConfiguracionSistema());
        conf.add(new ConfiguracionUsuario());
        conf.add(new ConfiguracionHoraria());
        for (Iterator<RecursoPersistente> i = conf.itera
tor(); i.hasNext(); ) {
            i.next().load();
        }
    }

    public void salvarConfiguracion() {
        for (Iterator<RecursoPersistente> i = conf.itera
tor(); i.hasNext(); ) {
            try {
                i.next().save();
            } catch (ConfiguracionNoAlmacenableException
e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

// Clase Main
public class Main {
    public static void main(String[] args) {
        Configuracion configuracion = new Configuracion
();

        // Cargar configuraciones
        System.out.println("Cargando configuraciones:");
        configuracion.cargarConfiguracion();

        // Salvar configuraciones
```

```
        System.out.println("\nAlmacenando configuracione
s:");
        configuracion.salvarConfiguracion();
    }
}
```

2. **Cumple la clase Configuracion en Principio OCP. Justifica la respuesta.**

   No, la clase `Configuracion` no cumple con el Principio de Abierto/Cerrado (OCP). La razón es que si se necesita agregar una nueva configuración que implemente la interfaz `RecursoPersistente`, se debe modificar el método `cargarConfiguracion` de la clase `Configuracion` para incluir la nueva configuración. Esto viola el OCP porque la clase no está cerrada para la modificación; se debe cambiar su código fuente para extender su funcionalidad.

3. **Cumple la clase Configuracion el Principio de Liskov. Justifica la respuesta.**

   No, la clase `Configuracion` no cumple con el Principio de Sustitución de Liskov (LSP). La razón es que la clase `ConfiguracionHoraria` viola el contrato definido por la interfaz `RecursoPersistente`. En lugar de almacenar la configuración, el método `save()` de `ConfiguracionHoraria` lanza un error. Esto significa que no se puede reemplazar un objeto de tipo `RecursoPersistente` con un objeto de tipo `ConfiguracionHoraria` sin alterar la corrección del programa.

   Refactoriza la aplicación para que cumpla el principio de Liskov. La solución a este ejercicio lo puedes encontrar en: https://lassala.net/2010/11/04/a-good-example-of-liskovsubstitution-principle/

   Para cumplir con el Principio de Liskov, necesitamos asegurarnos de que todas las clases que implementan la interfaz `RecursoPersistente` cumplan con el contrato definido por la interfaz. En este caso, podemos modificar la interfaz y las clases para manejar correctamente la situación en la que una configuración no puede ser almacenada.

   Excepción Personalizada

```
public class ConfiguracionNoAlmacenableException extends
Exception {
    public ConfiguracionNoAlmacenableException(String me
ssage) {
        super(message);
    }
}
```

Interfaz `RecursoPersistente`

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

Clase `ConfiguracionSistema`

```
public class ConfiguracionSistema implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion sistema cargad
a");
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        System.out.println("Configuracion sistema almace
nada");
    }
}
```

Clase `ConfiguracionUsuario`

```
public class ConfiguracionUsuario implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion usuario cargad
a");
```

```
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        System.out.println("Configuracion usuario almace
nada");
    }
}
```

Clase `ConfiguracionHoraria`

```
public class ConfiguracionHoraria implements RecursoPers
istente {
    public void load() {
        System.out.println("Configuracion horaria cargad
a");
    }
    public void save() throws ConfiguracionNoAlmacenable
Exception {
        throw new ConfiguracionNoAlmacenableException("E
RROR, la hora no se puede almacenar, es solo de lectur
a");
    }
}
```

Clase `Configuracion`

```
import java.util.Vector;
import java.util.Iterator;

public class Configuracion {
    Vector<RecursoPersistente> conf = new Vector<Recurso
Persistente>();

    public void cargarConfiguracion() {
        conf.add(new ConfiguracionSistema());
        conf.add(new ConfiguracionUsuario());
        conf.add(new ConfiguracionHoraria());
        for (Iterator<RecursoPersistente> i = conf.itera
```

```
tor(); i.hasNext(); ) {
            i.next().load();
        }
    }

    public void salvarConfiguracion() {
        for (Iterator<RecursoPersistente> i = conf.itera
tor(); i.hasNext(); ) {
            try {
                i.next().save();
            } catch (ConfiguracionNoAlmacenableException
e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Clase `Main`

```
public class Main {
    public static void main(String[] args) {
        Configuracion configuracion = new Configuracion
();

        // Cargar configuraciones
        System.out.println("Cargando configuraciones:");
        configuracion.cargarConfiguracion();

        // Salvar configuraciones
        System.out.println("\nAlmacenando configuracione
s:");
        configuracion.salvarConfiguracion();
    }
}
```

4. **Explica de forma general (independientemente del ejemplo) cual es el problema y la solución propuesta.**

El problema es que la clase `ConfiguracionHoraria` viola el Principio de Sustitución de Liskov (LSP) porque su método `save()` no cumple con el contrato definido por la interfaz `RecursoPersistente`. En lugar de almacenar la configuración, lanza un error. Esto significa que no se puede reemplazar un objeto de tipo `RecursoPersistente` con un objeto de tipo `ConfiguracionHoraria` sin alterar la corrección del programa.

Para cumplir con el Principio de Liskov, se propone introducir una excepción personalizada (`ConfiguracionNoAlmacenableException`) que se lance cuando una configuración no puede ser almacenada. La interfaz `RecursoPersistente` se modifica para incluir esta excepción en el método `save()`. Las clases que implementan la interfaz lanzan esta excepción cuando no pueden almacenar la configuración. La clase `Configuracion` maneja adecuadamente esta excepción, asegurando que el programa siga funcionando correctamente incluso cuando se reemplazan objetos de tipo `RecursoPersistente` con objetos de tipo `ConfiguracionHoraria`.

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 * porcentajeDedu
ccion) / 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 * porcentajeD
educcion+3) / 100;
sino importeDeduccion = (importeFactura * porcentajeDedu
ccion) / 100;
```

   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(CalculadoraIVA calculadora
IVA, CalculadoraDeduccion calculadoraDeduccion) {
        // Calculamos la deducción
        importeDeduccion = calculadoraDeduccion.calcular
Deduccion(importeFactura, porcentajeDeduccion);
        // Calculamos el IVA
        importeIVA = calculadoraIVA.calcularIVA(importeF
actura);
        // Calculamos el total
        importeTotal = (importeFactura - importeDeduccio
n) + importeIVA;
    }
}
```

Clase `CalculadoraIVA`

```
public class CalculadoraIVA {
    private float tasaIVA;

    public CalculadoraIVA(float tasaIVA) {
        this.tasaIVA = tasaIVA;
    }

    public float calcularIVA(float importeFactura) {
        return importeFactura * tasaIVA;
    }
}
```

Clase `CalculadoraDeduccion`

```
public class CalculadoraDeduccion {
    public float calcularDeduccion(float importeFactura,
int porcentajeDeduccion) {
        if (importeFactura > 10000) {
            return (importeFactura * (porcentajeDeduccio
n + 3)) / 100;
        } else {
            return (importeFactura * porcentajeDeduccio
```

```
n) / 100;
        }
    }
}
```

Main

```
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Factura factura = new Factura();
        factura.codigo = "F001";
        factura.fechaEmision = new Date();
        factura.importeFactura = 15000;
        factura.porcentajeDeduccion = 5;

        CalculadoraIVA calculadoraIVA = new CalculadoraI
VA(0.16f);
        CalculadoraDeduccion calculadoraDeduccion = new
CalculadoraDeduccion();

        factura.calcularTotal(calculadoraIVA, calculador
aDeduccion);

        System.out.println("Código de la factura: " + fa
ctura.codigo);
        System.out.println("Fecha de emisión: " + factur
a.fechaEmision);
        System.out.println("Importe de la factura: " + f
actura.importeFactura);
        System.out.println("Importe del IVA: " + factur
a.importeIVA);
        System.out.println("Importe de la deducción: " +
factura.importeDeduccion);
        System.out.println("Importe total: " + factura.i
mporteTotal);
```

```
    }
}
```

Los cambios necesarios ya están implementados en la clase `CalculadoraDeduccion`. No se requieren modificaciones adicionales en la clase `Factura` o en el programa principal. La lógica de cálculo de la deducción está aislada en la clase `CalculadoraDeduccion`, cumpliendo con el Principio de Responsabilidad Única (SRP).

Con esta refactorización, cada responsabilidad queda aislada en una clase separada. La clase `Factura` se encarga de calcular el total de la factura, mientras que las clases `CalculadoraIVA` y `CalculadoraDeduccion` se encargan de calcular el IVA y la deducción, respectivamente. Esto facilita la modificación de la tasa de IVA o la lógica de cálculo de la deducción sin afectar a la clase `Factura`.

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

   Para cambiar la tasa de IVA del 16% al 18%, solo necesitaríamos modificar la clase `CalculadoraIVA` para que utilice la nueva tasa de IVA. Esto es posible gracias a la refactorización que se ha realizado, donde cada responsabilidad está aislada en su propia clase.

   En el Main, al crear una instancia de `CalculadoraIVA`, pasamos la nueva tasa de IVA (18%) como parámetro al constructor.

   ```
   CalculadoraIVA calculadoraIVA = new CalculadoraIVA(0.18
   f);
   ```

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

   En el método `calcularTotal` de la clase `Factura` se debe añadir una condición para verificar si el código de la factura es `0`. Si es `0`, el IVA se establece en `0`. Si no es `0`, se calcula el IVA utilizando la clase `CalculadoraIVA`.

   ```
   import java.util.Date;
   ```

```
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(CalculadoraIVA calculadora
IVA, CalculadoraDeduccion calculadoraDeduccion) {
        // Calculamos la deducción
        importeDeduccion = calculadoraDeduccion.calcular
Deduccion(importeFactura, porcentajeDeduccion);

        // Calculamos el IVA solo si el código de la fac
tura no es 0
        if (!codigo.equals("0")) {
            importeIVA = calculadoraIVA.calcularIVA(impo
rteFactura);
        } else {
            importeIVA = 0;
        }

        // Calculamos el total
        importeTotal = (importeFactura - importeDeduccio
n) + importeIVA;
    }
}
```

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, porce
ntajeDeduccion);
Iva iva=new Iva();
// Calculamos el IVA
importeIVA = iva.calculaIva(importeFactura);
// Calculamos el total
importeTotal = (importeFactura - importeDeduccion) + import
eIVA;
}
}
```

Consultas:

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

   No, la clase `Factura` no cumple con el Principio de Inversión de Dependencia (DIP). El Principio de Inversión de Dependencia establece que los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Esto significa que las clases de alto nivel deben depender de interfaces o clases abstractas, no de implementaciones concretas.

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

   Para cumplir con el Principio de Inversión de Dependencia, debemos introducir interfaces o clases abstractas para `Deduccion` e `Iva`, y hacer que la clase `Factura` dependa de estas abstracciones en lugar de las implementaciones concretas.

   Se definen interfaces para `Deduccion` e `Iva`

   ```
   public interface IDeduccion {
       float calculaDeduccion(float importeFactura, int por
   ```

```
centajeDeduccion);
}

public interface IIva {
    float calculaIva(float importeFactura);
}
```

Se implementan las interfaces en clases concretas

```
public class Deduccion implements IDeduccion {
    public float calculaDeduccion(float importeFactura,
int porcentajeDeduccion) {
        if (importeFactura > 10000) {
            return (importeFactura * (porcentajeDeduccio
n + 3)) / 100;
        } else {
            return (importeFactura * porcentajeDeduccio
n) / 100;
        }
    }
}

public class Iva implements IIva {
    private float tasaIVA;

    public Iva(float tasaIVA) {
        this.tasaIVA = tasaIVA;
    }

    public float calculaIva(float importeFactura) {
        return importeFactura * tasaIVA;
    }
}
```

Se modifica la clase `Factura` para depender de las Interfaces

```
import java.util.Date;
```

```
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 IDeduccion deduccion;
    private IIva iva;

    public Factura(IDeduccion deduccion, IIva iva) {
        this.deduccion = deduccion;
        this.iva = iva;
    }

    // Método que calcula el total de la factura
    public void calcularTotal() {
        // Calculamos la deducción
        importeDeduccion = deduccion.calculaDeduccion(im
porteFactura, porcentajeDeduccion);

        // Calculamos el IVA
        importeIVA = iva.calculaIva(importeFactura);

        // Calculamos el total
        importeTotal = (importeFactura - importeDeduccio
n) + importeIVA;
    }
}
```

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 messag
e) {
        // Envía un mensaje a la dirección de correo del Co
ntacto c.
    }
}
public class SMSSender {
    public static void sendSMS(Contacto c, String message)
{
```

```
        // Envía un mensaje SMS al teléfono del Contacto c.
    }
}
```

Consultas:

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.**

   Información Necesaria:

   - **EmailSender**: Necesita la dirección de correo electrónico del contacto para enviar un mensaje de correo electrónico.
   - **SMSSender**: Necesita el número de teléfono del contacto para enviar un mensaje SMS.

   Información Recogida:

   - **EmailSender**: Recoge la dirección de correo electrónico del contacto.
   - **SMSSender**: Recoge el número de teléfono del contacto.

   Sí, incumplen el Principio de Segregación de Interfaz (ISP). Las clases `EmailSender` y `SMSSender` dependen de la clase `Contacto`, que contiene más métodos de los que realmente necesitan. Esto viola el ISP porque estas clases están forzadas a depender de métodos que no utilizan.

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.**

   Interfaz para la dirección de correo electrónico:

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

   Interfaz para el número de teléfono:

```
public interface ITelephoneContacto {
    String getTelephone();
}
```

Modificación de la Clase `Contacto`

```
public class Contacto implements IEmailContacto, ITeleph
oneContacto {
    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) { emailAddres
s = ea; }
    public String getEmailAddress() { return emailAddres
s; }

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

Clase `EmailSender`:

```
public class EmailSender {
    public static void sendEmail(IEmailContacto c, Strin
g message) {
        // Envía un mensaje a la dirección de correo del
Contacto c.
        System.out.println("Enviando correo a: " + c.get
EmailAddress());
        System.out.println("Mensaje: " + message);
```

```
    }
}
```

Clase `SMSSender`:

```
public class SMSSender {
    public static void sendSMS(ITelephoneContacto c, Str
ing message) {
        // Envía un mensaje SMS al teléfono del Contacto
c.
        System.out.println("Enviando SMS a: " + c.getTel
ephone());
        System.out.println("Mensaje: " + message);
    }
}
```

Main

```
public class Main {
    public static void main(String[] args) {
        Contacto contacto = new Contacto();
        contacto.setName("Juan Pérez");
        contacto.setAddress("Calle Falsa 123");
        contacto.setEmailAddress("juan.perez@example.co
m");
        contacto.setTelephone("123456789");

        EmailSender.sendEmail(contacto, "Hola, este es u
n correo de prueba.");
        SMSSender.sendSMS(contacto, "Hola, este es un SM
S de prueba.");
    }
}
```

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

Clase `GmailAccount`

```
public class GmailAccount implements IEmailContacto {
    String name, emailAddress;

    public void setName(String n) { name = n; }
    public String getName() { return name; }

    public void setEmailAddress(String ea) { emailAddres
s = ea; }
    public String getEmailAddress() { return emailAddres
s; }
}
```

La clase `GmailAccount` implementa la interfaz `IEmailContacto`, lo que permite que `EmailSender` pueda trabajar con `GmailAccount`. El programa principal crea un objeto de `GmailAccount` y lo pasa al método `sendEmail` de `EmailSender`.