# Principio Abierto-Cerrado (OCP).

Dadas las siguientes clases (algunas clases están incompletas)

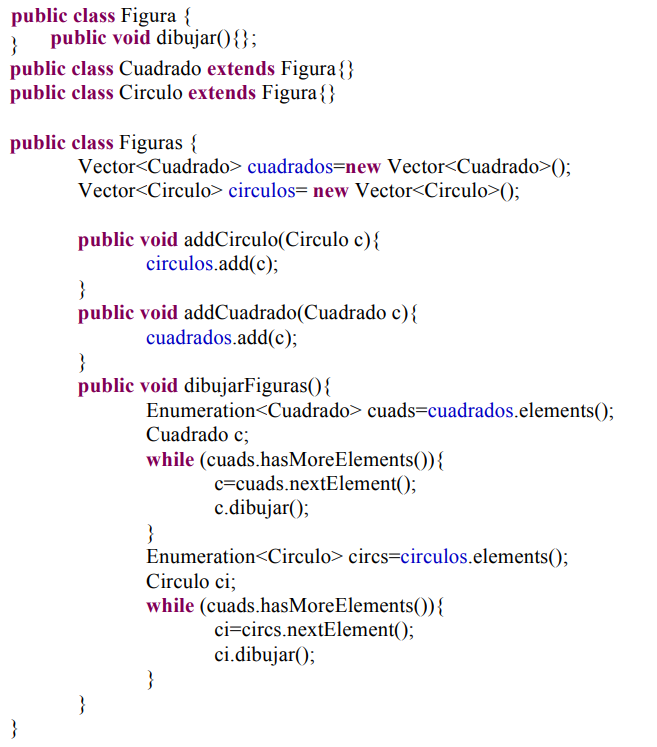


Ilustración 1. Ejercicio N° 1

Consultas: 1:

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

* Rta: No cumple ya que no es extensible, al momento de crear otra figura, si queremos adicionar otro tipo de figura tendremos que modificar el código actual, perdiendo la propiedad de extensibilidad.
* El Principio de Abierto-Cerrado (OCP) establece que las clases deben estar abiertas para la extensión, pero cerradas para la modificación. Es decir, deberíamos poder añadir nuevas funcionalidades a una clase sin necesidad de modificar su código existente.
* En la clase Figuras, tenemos dos listas: una para Cuadrado y otra para Circulo. El método dibujarFiguras() está diseñado para iterar sobre cada una de estas listas y llamar al método dibujar() de cada figura. Este diseño no cumple con el Principio OCP porque si queremos añadir una nueva forma, como un Triangulo, tendríamos que modificar la clase Figuras para añadir una nueva lista y cambiar el método dibujarFiguras() para manejar esta nueva figura.

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

* Rta: Consultar código en NetBeans.

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

* Rta: Sí, la tarea realizada es una refactorización. Refactorizar significa mejorar el diseño del código existente sin cambiar su comportamiento observable. En este caso, hemos mejorado el diseño de la clase Figuras para que cumpla con el Principio OCP sin cambiar la funcionalidad que ofrece. El código ahora es más flexible y adherente a los principios SOLID, lo que facilita su mantenimiento y extensión en el futuro.

# Principio Liskov (LSK).

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

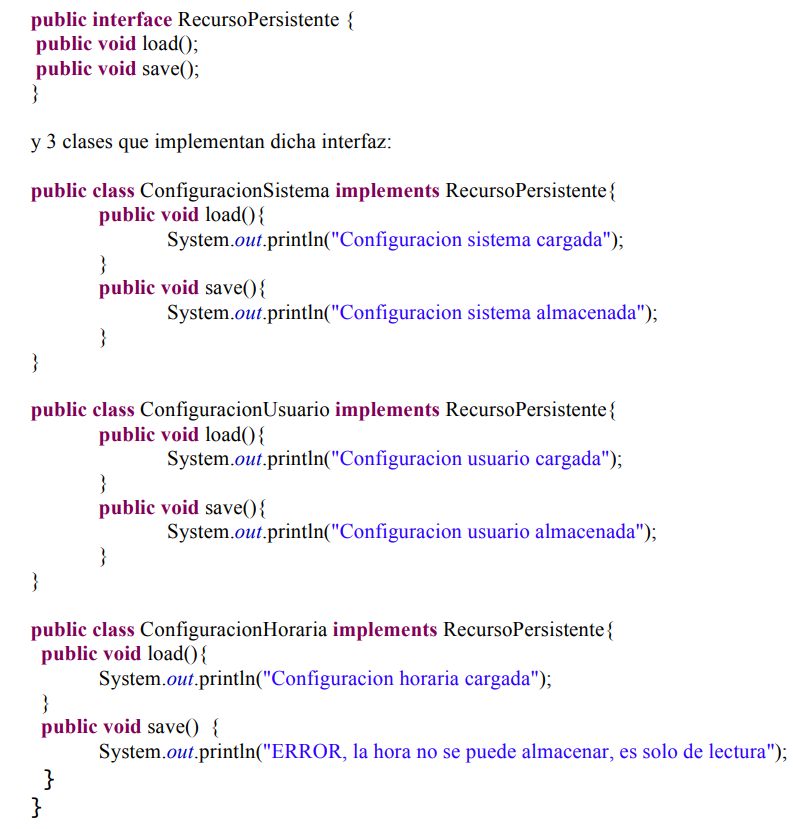


Ilustración 2.Ejercicio N° 2.

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:

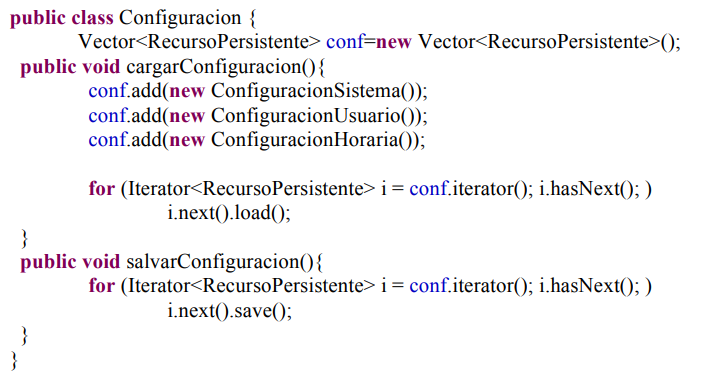


Ilustración 3.Ejercicio N° 2.1.

Consultas:

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

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

* Rta: El Principio de Abierto-Cerrado (OCP) establece que las clases deben estar abiertas para la extensión, pero cerradas para la modificación.
* La clase Configuracion no cumple completamente con el OCP porque si queremos añadir una nueva clase que implemente RecursoPersistente, tendríamos que modificar el método cargarConfiguracion() para añadir una nueva instancia de esta clase en la lista conf. Esto viola el OCP, ya que la clase Configuracion debería ser extensible sin necesidad de ser modificada.

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

* El **Principio de Sustitución de Liskov (LSK)** indica que los objetos de una subclase deben poder sustituir a los objetos de la clase base sin afectar la funcionalidad del programa.
* La clase ConfiguracionHoraria **no cumple con el Principio de Liskov** porque el método save() no realiza la acción esperada, sino que lanza un error indicando que la hora no se puede almacenar. Esto significa que si se utiliza un objeto de tipo ConfiguracionHoraria donde se espera un RecursoPersistente, el programa podría fallar o comportarse de manera inesperada, lo cual viola el LSK.

4. 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-goodexample-of-liskov-substitution-principle/

* Rta: Consultar código en NetBeans.

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

* **Problema**: En el código original, una subclase (ConfiguracionHoraria) no cumplía completamente con el contrato de la interfaz que implementaba (RecursoPersistente), ya que su método save() no realizaba la operación esperada. Esto podría causar errores en tiempo de ejecución si un objeto de esta subclase se usaba en un contexto que esperaba que el método save() funcionara correctamente.
* **Solución**: La solución fue refactorizar el código para que cada clase implementara solo las interfaces que reflejan correctamente sus capacidades. Esto se logra mediante la segregación de interfaces, permitiendo que las clases implementen solo las interfaces necesarias, y así respetar el contrato definido por esas interfaces. De esta manera, se garantiza que cualquier clase que implemente una interfaz puede sustituir a otra clase que también la implemente sin provocar errores inesperados.

# Principio de Responsabilidad Única (SRP)

Dada la siguiente clase Factura:

Imagen que contiene Texto

Descripción generada automáticamente

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;

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

**Rta: Explicación de la Refactorización**

1. **Clase Factura: Se encargará solo de almacenar los datos básicos de la factura.**
2. **Clase CalculadorDeduccion: Será responsable de calcular las deducciones.**
3. **Clase CalculadorIVA: Será responsable de calcular el IVA.**
4. **Clase CalculadorTotal: Será responsable de calcular el total de la factura.**
5. Indica los cambios que tendrías que realizar si a las facturas de código 0, no se le aplicase el IVA

Rta:

**Refactoriza la aplicación para que cada responsabilidad quede aislada en una clase.**

* **Hecho**: Como se muestra arriba, cada cálculo ha sido movido a su propia clase para seguir el SRP.

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

* **Solución**: Solo sería necesario cambiar el valor de la constante IVA en la clase CalculadorIVA:

|  |
| --- |
| private static final float IVA = 0.18f; |

Este cambio no afecta a ninguna otra parte del código, lo cual es una ventaja de seguir el SRP.

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

* **Solución**: Podemos modificar el método calcularIVA en la clase CalculadorIVA para incluir una condición que verifique el código de la factura:

|  |
| --- |
| public float calcularIVA(Factura factura) { if (factura.getCodigo().equals("0")) { return 0; } return factura.getImporteFactura() \* IVA; } |

Este cambio también es local y no afecta a las otras responsabilidades.

Esta refactorización asegura que cada clase tiene una única responsabilidad, cumpliendo así con el SRP, y facilita la modificación del comportamiento del sistema sin afectar otras partes del código.

# 4. Principio de Inversión de dependencia (DIP).

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Consultas:

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

**No, el código no cumple con el principio de Inversión de Dependencia.**

La clase Factura depende directamente de las clases concretas Deduccion e Iva, lo que significa que está acoplada a estas implementaciones específicas. Esto puede llevar a un código menos flexible y más difícil de mantener, ya que cualquier cambio en la forma en que se calculan las deducciones o el IVA requeriría modificar la clase Factura.

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

Para cumplir con el DIP, deberíamos hacer que Factura dependa de abstracciones en lugar de implementaciones concretas. Esto se puede lograr mediante el uso de interfaces o clases abstractas para Deduccion e Iva, y luego inyectando estas dependencias en Factura, ya sea a través del constructor o mediante inyección de dependencias.

* **Abstracciones**: Se crearon las interfaces CalculadorDeduccion y CalculadorIVA para definir las operaciones necesarias.
* **Dependencia de abstracciones**: La clase Factura ahora depende de las interfaces CalculadorDeduccion y CalculadorIVA en lugar de depender directamente de las implementaciones concretas.
* **Inyección de dependencias**: Las implementaciones concretas (Deduccion e Iva) se pasan a la clase Factura a través de su constructor. Esto permite que Factura sea más flexible, ya que diferentes implementaciones de deducción o IVA pueden ser inyectadas sin necesidad de modificar la clase.

# Principio de Segregación de interfaces (ISP).

Disponemos de la siguiente clase Contacto:



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.

Rta: Las clases EmailSender y SMSSender necesitan la dirección de correo electrónico (emailAddress) y el número de teléfono (telephone), respectivamente, de la clase Contacto. Actualmente, ambas clases reciben un objeto Contacto, lo que les da acceso a más información de la necesaria (por ejemplo, name y address). Esto incumple el **Principio de Segregación de Interfaces (ISP)**, ya que estas clases deberían depender solo de la información que realmente necesitan, no de más.

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.

Rta: Podemos crear dos interfaces: una que proporcione la dirección de correo electrónico (EmailRecipient) y otra que proporcione el número de teléfono (SMSRecipient). Luego, refactorizamos las clases EmailSender y SMSSender para que dependan de estas interfaces en lugar de la clase Contacto completa.

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