Laboratorio de Software - 2025
Práctica n° 1

Temas
- Interfaces
- Polimorfismo

(1) Declaración e implementación de **Interfaces**.
a) ¿Son correctas las siguientes declaraciones?

```
interface ColPrimarios {
 int ROJO=1, VERDE=2, AZUL=4;
}

interface ColArcoIris extends ColPrimarios {
 int AMARILLO=3, NARANJA=5, INDIGO=6, VIOLETA=7;
}

interface ColImpresion extends ColPrimarios {
 int AMARILLO=8, CYAN=16, MAGENTA=32;
}

interface TodosLosColores extends ColImpresion, ColArcoIris {
 int FUCSIA=17, BORDO=ROJO+90;
}

class MisColores implements ColImpresion, ColArcoIris {
   public MisColores() {
      int unColor=AMARILLO;
   }
}
```

No es correcta. En la clase MisColores intenta utilizar a la variable AMARILLO, pero como extiende ColImpresion (que tiene una variable AMARILLO=8) y ColArcoIris (que tiene una variable AMARILLO=3), tiene dos veces la variable AMARILLO con distintos valores y el compilador no sabrá a cuál hace referencia al intentar asignarla a la variable unColor.
Puede solucionarse usando el nombre completo de la variable.

b) Analice el código de la interface y las clases que la implementan. Determine si son legales o no. En caso de ser necesario, realice las correcciones que correspondan. ¿Cómo podría modificar el método afinar() para evitar realizar cambios en las clases que implementan InstrumentoMusical?

```
public interface InstrumentoMusical {
  void hacerSonar();
  String queEs();
  void afinar(){}
 }

class abstract InstrumentoDeViento implements InstrumentoMusical {
  void hacerSonar(){
     System.out.println("Sonar Vientos");
  }
  public String queEs() {
     return "Instrumento de Viento";
  }
}

class InstrumentoDeCuerda implements InstrumentoMusical {
  void hacerSonar(){
     System.out.println("Sonar Cuerdas");
  }
  public String queEs() {
     return "Instrumento de Cuerda";
  }
}
```

En Java, todos los métodos declarados en una interface son públicos y abstractos por defecto. Cuando una clase (sea concreta o abstracta) implementa esa interfaz, el método debe tener como mínimo la misma visibilidad. Tanto la clase abstracta InstrumentoDeViento como la clase InstrumentoDeCuerda no declaran el método hacerSonar() como public, haciéndolo package-private, lo cual es un acceso más restringido.
Para evitar realizar cambios en las clases que implementan InstrumentoMusical puede declararse el método afinar() como default y darle determinado comportamiento. Así las otras clases no se romperán y pueden sobreescribirlo si lo desean.

(2) Redefina la clase PaintTest del ejercicio 6 de la práctica 1 de manera de imprimir las figuras geométricas ordenadas de acuerdo al valor de su área. Defina la comparación entre figuras geométricas usando la siguiente regla: una figura A es menor que una figura B si el área de A es menor que el área de B. Use para ordenar el arreglo de figuras los métodos de ordenación disponibles en la clase java.util.Arrays.

(3) Se desea implementar un tipo especial de HashSet con la característica de poder consultar la cantidad total de elementos que se agregaron al mismo. Analice y pruebe el siguiente código de manera de corroborar si realiza lo pedido.

```
public class HashSetAgregados<E> extends HashSet<E> {

    private int cantidadAgregados = 0;

    public HashSetAgregados() {
    }

    public HashSetAgregados(int initCap, float loadFactor) {
       super(initCap, loadFactor);
    }

    @Override public boolean add(E e) {
      cantidadAgregados++;
      return super.add(e);
    }

    @Override public boolean addAll(Collection<? extends E> c) {
       cantidadAgregados += c.size();
       return super.addAll(c);
    }
    public int getCantidadAgregados() {
       return cantidadAgregados;
    }
}
```

El código no realiza lo pedido, en cantidadAgregados se guarda la cantidad de elementos que se intentaron agregar, no los efectivamente agregados. Como HashSet no permite elementos duplicados, al hacer add() dos veces con el mismo elemento, el contador se incrementará pero no reflejará la cantidad de elementos agregados.

a) Agregue a una instancia de HashSetAgregados los elementos de otra colección (mediante el método addAll). Invoque luego al método getCantidadAgregados. ¿La clase tiene el funcionamiento esperado? ¿Por qué? ¿Tiene relación con la herencia?
Nuevamente, el contador reflejará la cantidad de elementos que se intentaron agregar, no los realmente agregados.
Tiene relación con la herencia. La clase extiende HashSet, sobreescribiendo los métodos pero a su vez utilizando los métodos de la clase padre para agregar los elementos.

b) Diseñe e implemente una alternativa para HashSetAgregados. ¿Qué interface usaría? ¿Qué ventajas proporcionaría esta nueva implementación respecto de la original?
Podría usarse la interfaz Set, la cual implementa también HashSet. Con este cambio puede utilizarse la clase ForwardingSet para trabajar con cualquier implementación de Set y que funcione con cualquier constructor preexistente.

Básicamente, es la implementación del patrón Decorator.

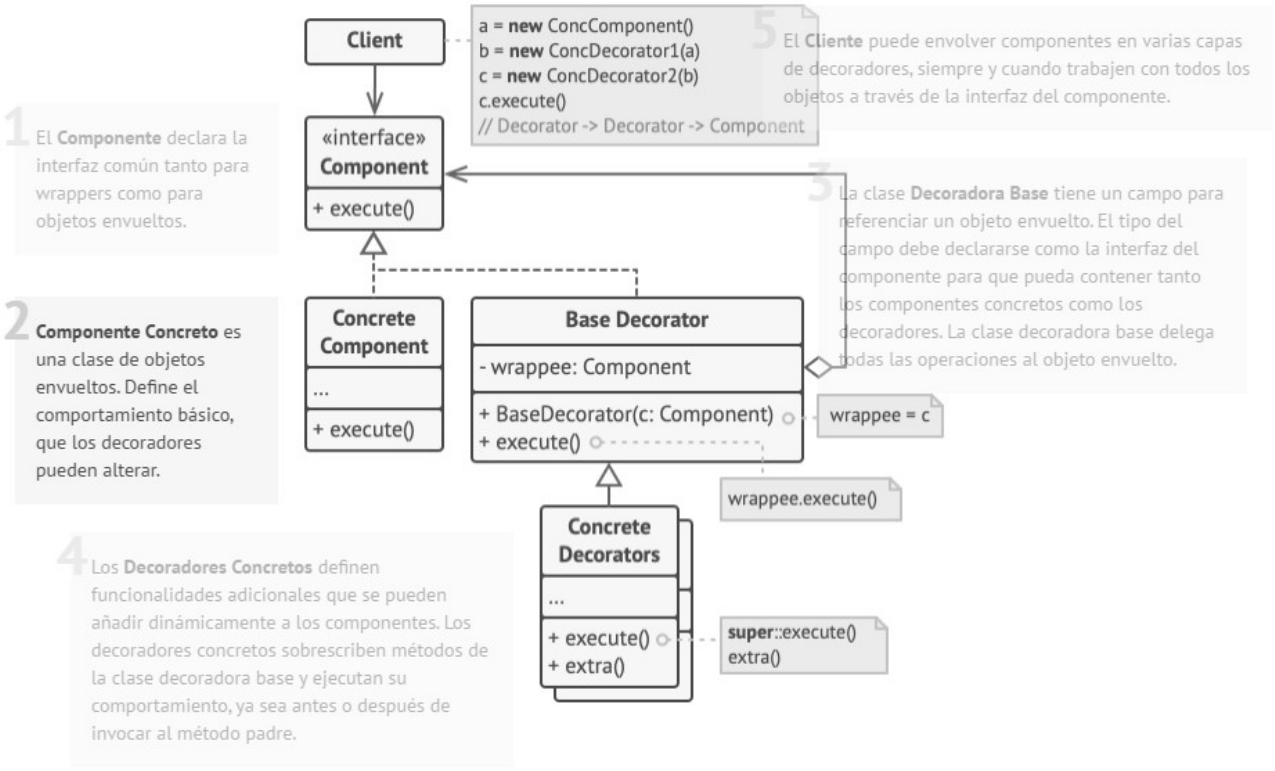
Decorator es un patrón de diseño estructural que permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase. No obstante, la herencia tiene varias limitaciones:

- La herencia es estática. No se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente.
- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo.

Una de las formas de superar estas limitaciones es empleando la Agregación o la Composición en lugar de la Herencia. Ambas alternativas funcionan prácticamente del mismo modo: un objeto tiene una referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto puede realizar ese trabajo, heredando el comportamiento de su superclase.

"Wrapper" (envoltorio) es el sobrenombre alternativo del patrón Decorator. Un wrapper es un objeto que puede vincularse con un objeto objetivo. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.



c) Se desea implementar otro tipo especial de Set con la característica de poder consultar la cantidad total de elementos que se removieron del mismo. Diseñe e implemente una solución que permita fácilmente definir nuevos tipos de Set con distintas características.

(4) Redefina las clases del ejercicio 6 de la práctica 1 de manera que las figuras se puedan serializar.

a) ¿Cómo se serializa un objeto? ¿Con qué fin?

La serialización en Java es el proceso de guardar el estado de un objeto como una secuencia de bytes. La deserialización en Java es el proceso de restaurar un objeto a partir de estos bytes.
La serialización permite trasferir un objeto por alguna red o guardarlo en un archivo para luego trabajar con una copia reconstruida de la instancia del objeto.
A veces podemos necesitar que se haga algo especial en el momento de serializarlo, bien al construirlo a bytes, bien al recibirlo.
Los objetos se serializan implementando la interfaz Serializable o Externalizable.
Serializable es una interfaz marker sin métodos, no es necesario implementar nada. Las variables de clase de la clase a serializar deben ser serializables, si no se quiere puede usarse la palabra clave transient para excluirla.
La documentación indica que se pueden declarar métodos para controlar el proceso de serialización y deserialización, por ejemplo, usando los métodos

writeObject y *readObject* que escriben datos en el byte stream o escriben valores en los campos de los objetos serializados al leer el byte stream.
Por su parte, la interfaz Externalizable, sí tiene métodos a implementar:

- *writeExternal(ObjectOutput out)* → escribe los valores del objeto serializable
- *readExternal(ObjectInput input)* → restaura un objeto, asignando valores leídos desde el input

b) ¿Qué relación tiene con el serialVersionUID? Analice su impacto al modificar la implementación de las clases.

Un serialVersionUID es un identificador único de versión. Su objetivo principal es proporcionar un mecanismo de control de versiones para objetos serializados. Cuando un objeto se serializa, el serialVersionUID también se serializa.
Esto permite que el proceso de deserialización verifique que la versión del objeto serializado coincida con la versión de la definición de clase. Si las versiones no coinciden, se lanza una excepción InvalidClassException, lo que evita problemas de compatibilidad que pueden surgir al intentar deserializar un objeto con una versión de clase diferente.
El serialVersionUID es una variable estática de tipo final long que normalmente se declara en la definición de clase. Si no se declara manualmente, el compilador lo genera automáticamente basándose en la estructura de la clase, lo que hace que pequeños cambios (ej: cambiar el nombre de un método privado) alteren el serialVersionUID y rompan la compatibilidad.