Orientación a Objetos II (Parte II)

Abstracción II

Competencias

- Comprender las clases abstractas para generar herencia.
- Implementar polimorfismo para los principios de POO mediante herencia.

Introducción

En este capítulo seguiremos viendo aquellos conceptos de abstracción que son utilizados por desarrolladores/as de todo el mundo. Por ejemplo, veremos qué son las clases abstractas, polimorfismo aplicado y algunos principios que son útiles en la Programación Orientada a Objetos.

Las clases abstractas

Las clases abstractas no son más que clases que no pueden instanciarse. De ahí el concepto de abstracto. Según el filósofo José Ortega y Gasset podemos entender por "sustantivo abstracto" a aquella palabra que nombra un objeto que no es independiente, es decir, que siempre necesita de otro elemento en el que apoyarse para "ser". Esto significa que dichos sustantivos, al no referirse a un elemento concreto, hacen referencia a objetos que no se pueden percibir con los sentidos, sino solo imaginar.

El significado de que no puedan instanciarse radica en que son clases que permiten categorizar a otras. Por ejemplo, la clase `Animal` que posee una herencia, debiera ser abstracta, ya que la palabra Animal la asociamos al concepto de animal como tal, pero no es algo tangible. En la imagen 1 se puede apreciar un ejemplo que será la guía para el ejercicio guiado y entender estos conceptos.

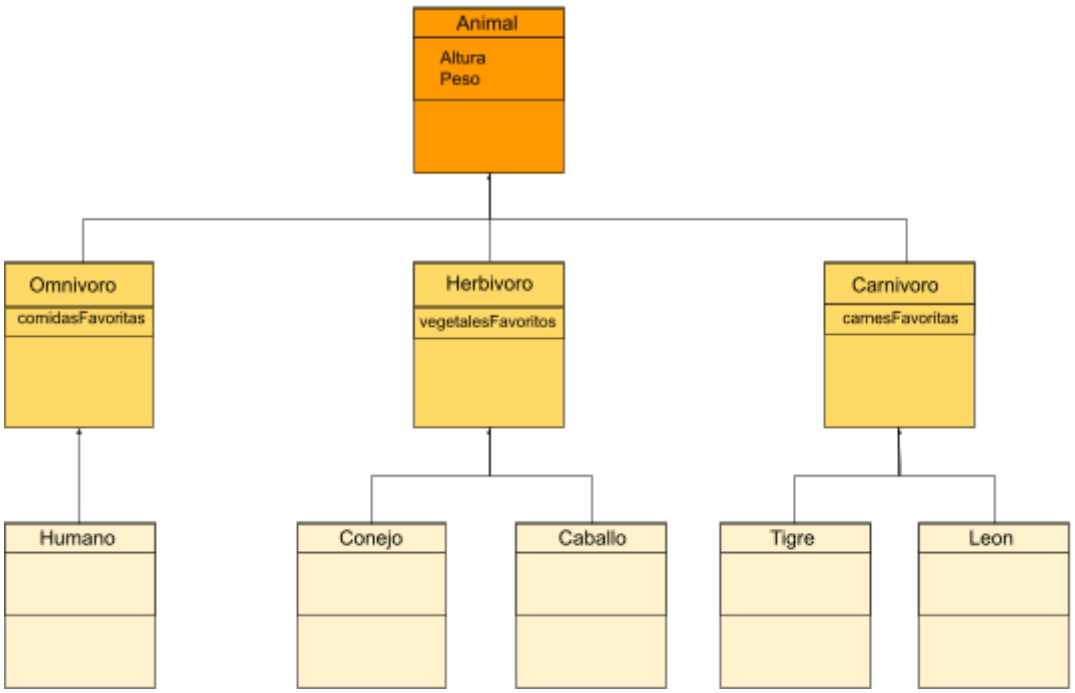

Imagen 1. Ejemplo de clase abstracta clase Animal.
Fuente: Desafío Latam

En la imagen se puede apreciar que la herencia se cumple mediante la superclase `Animal` y las subclases `Omnivoro`, `Herbivoro` y `Carnivoro`. A su vez existe otra herencia de otras subclases de cada categoría de alimentación del animal.

Ejercicio Guiado: Animales

Paso 1: Crear una clase llamada Animal que contiene la palabra reservada `abstract` antes de la palabra `class`. Esto significa que la clase **no puede ser instanciada,** generamos los getter and setter correspondientes de las variables altura y peso**.**

```
public abstract class Animal {
 private int altura;
 private int peso;

 public int getAltura() {
 return altura;
 }
 public void setAltura(int altura) {
 this.altura = altura;
 }
 public int getPeso() {
 return peso;
 }
 public void setPeso(int peso) {
 this.peso = peso;
 }
```

Esta clase puede ser la superclase de otras, como se ve en el diagrama, las cuales pueden o no ser abstractas al igual que `Animal`.

Paso 2: Terminar el árbol de carnívoros, creando la clase **abstracta** `Carnivoro`.

```
import java.util.List;

public abstract class Carnivoro extends Animal{
    List<String> carnesFavoritas;

public List<String> getCarnesFavoritas() {
      return carnesFavoritas;
   }
public void setCarnesFavoritas(List<String> carnesFavoritas) {
this.carnesFavoritas = carnesFavoritas;
  }
}
```

Paso 3: Crear la clase `Tigre` que extiende de `Carnivoro`.

```
//Tigre
public class Tigre extends Carnivoro{
}
```

Paso 4: Crear la clase `Leon` que extiende de `Carnivoro`.

```
//Leon
public class Leon extends Carnivoro{
}
```

`Tigre` y `Leon` podrán tener todos los atributos de `Carnivoro` y `Animal` .

Paso 5: Agregar un método main para crear esta lista polimórfica con clases abstractas.

```
public class Main {
  public static void main(String[] args) {
    Leon leon = new Leon();
    Tigre tigre = new Tigre();
     ArrayList<Animal> listaAnimales = new ArrayList<>();
     ArrayList<Carnivoro> listaCarnivoros = new ArrayList<>();
     listaAnimales.add(leon);
     listaAnimales.add(tigre);
     listaCarnivoros.add(leon);
     listaCarnivoros.add(tigre);
     System.out.println(listaAnimales);
     System.out.println(listaCarnivoros);
   }
}

    ---------------------------
    Impresión en pantalla:
    [desafio1.Leon@58372a00, desafio1.Tigre@4dd8dc3]
    [desafio1.Leon@58372a00, desafio1.Tigre@4dd8dc3]
```

Podemos agregar instancias de las subclases sin problemas a la lista, es decir, se puede utilizar polimorfismo y a la vez ayudamos a mantener un código ordenado, impidiendo que otros desarrolladores creen instancias de `Animal` o `Carnivoro`.

Otra ventaja de esto, es que podemos escribir el código en la superclase de un método o un atributo y utilizarlos en todas las subclases.

El concepto de abstracción en POO es el conjunto de todo lo aprendido previamente, ya que la abstracción es la realización de polimorfismo en busca de características comunes en objetos, todo esto con la finalidad de trabajarlos dentro de un mismo contexto.

Imagen 2. Principio de Abstracción.
Fuente: Desafío Latam

Ejercicio Propuesto (1)

En base al ejercicio guiado “Animales”, debemos hacer los siguientes requerimientos:

- Crear una clase nueva llamada `Herbivoro`, que contendrá una List<String> llamada `vegetalesFavoritos` con su respectivo getter and setter.
- Crear 2 clases nuevas llamadas `Conejo` y `Caballo` que extiendan de `Animal`.
- En la clase main instanciamos la clase `Conejo` y `Caballo`.
- Agregamos en la `listaAnimales` al conejo y al caballo.
- Agregamos una nueva ArrayList llamada `listaHerbivoros`.

Programación con principios

Competencias

- Comprender los principios que se aplican en la POO.
- Aplicar los diversos principios para entender la POO.

Introducción

A continuación, veremos algunos de los lineamientos más importantes en la Orientación a Objetos: los famosos **principios de la programación** o "técnicas para el desarrollo más acertado". Veremos ejemplos básicos para comprender lo que significa cada uno de ellos, con el fin de lograr que nuestros desarrollos en el futuro sean bien calificados.

Principio de modularización

El principio de modularización dicta que se deben separar las funcionalidades de un software en módulos y cada uno de ellos debe estar encargado de una parte del sistema. Esto ayuda a que cada módulo sea independiente y, por consecuencia, si se desea modificar uno de los módulos, el impacto en los otros no sea importante.

Para lograr la modularidad se debe atomizar un problema y obtener sub-problemas donde cada módulo tienda a dar solución. Estos módulos pueden estar separados en métodos, clases, paquetes, colecciones de paquetes e incluso proyectos. La escala de modularidad va a depender del tamaño del software en cuestión.

Imagen 3. Ejemplo modularización con piezas de puzzle.
Fuente: Desafío Latam

Principio DRY (Don't Repeat Yourself)

El principio DRY, como su nombre lo indica, se refiere a no repetir el código en ninguna instancia. Por ejemplo, si vamos a usar un `if` idéntico más de una vez, estamos violando el principio y, como solución, deberíamos guardar ese `if` dentro de un método y re-utilizarlo donde sea necesario.

Hay otros casos en que dos porciones de código hacen casi lo mismo, por ejemplo:

```
int indiceNombre;
int indiceApellido;
 for(int i = 0; i <= listaNombres; i++){
   if(listaNombres.get(i).equals("Juan")){
    indiceNombre = i;
}
  for(int i = 0; i <= listaApellidos; i++){
   if(listaApellidos.get(i).equals("Perez")){
    indiceApellido = i;
}
```

En este caso, tenemos dos ciclos que hacen casi lo mismo y podríamos reemplazarlos creando el siguiente método:

```
public int retornarIndice(String elementoBuscado, List<String> lista){
for(int i = 0; i <= lista; i++){
 if(lista.get(i).equals(elementoBuscado)){
 return i;
 }
}
```

Entonces el primer código quedaría así:

```
int indiceNombre = retornarIndice("Juan", listaNombres);
int indiceApellido = retornarIndice("Perez", listaApellidos);
```

De esta forma, el código queda más ordenado y se cumple el principio DRY.

Principio KISS (Keep It Simple Stupid)

Este principio se refiere a crear un software sin necesidad de hacerlo más complejo. De esta forma es más fácil de entender y utilizar. La simplicidad es bien aceptada en el diseño de todo ámbito y qué mejor ejemplo que el de Apple que creó un teléfono con un solo botón para manejarlo.

Imagen 4. Principio KISS.
Fuente: Desafío Latam

Principio YAGNI (You Aren't Gonna Need It)

Este principio indica que no se debe agregar piezas que no se utilizarán. Por ejemplo, al pensar en lo que se hará en el futuro, se agregan piezas de software que aún no van a ser utilizadas, pero que podrían ser necesarias más adelante cuando la aplicación crezca. En este caso, se está violando el principio porque estaríamos agregando código que no necesitamos en el momento, es mejor enfocarse en lo realmente necesario y no perder tiempo en funcionalidades extras que no se han pedido… **tu tiempo como desarrollador/a vale oro**.

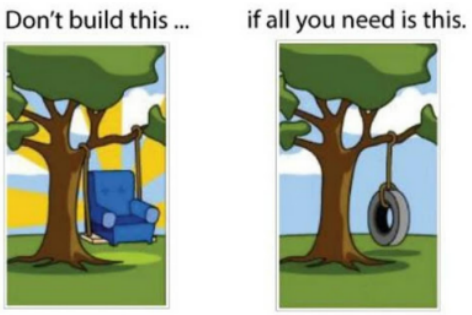

Imagen 5. Principio YAGNI.
Fuente: Desafío Latam

Estos últimos tres principios se refieren a la programación en general, hay algunos principios que se refieren a la programación modularizada y a la cohesión del software y diseño.

Principio de Acoplamiento y Cohesión (Tight & Loose)

La cohesión es el concepto que mide la "fuerza" con que las partes o piezas de un software están conectadas dentro de un módulo. La cohesión puede medirse como alta (fuerte) o baja (débil). Se prefiere una cohesión alta debido a que esto significa que el software es más robusto, escalable y fiable. Además el código facilita el entendimiento de los desarrolladores debido a su grado de reutilización del mismo.

Este concepto y sus métricas fueron primero diseñadas por Larry Constantine en el diseño estructural (o diseño para programación estructurada). El concepto de acoplamiento, se podría decir que es lo contrario de la cohesión del software, ya que un código acoplado es difícil de entender y mejorar debido a que muchas cosas dependen de otras muchas cosas dentro del código y si algo se modifica podrían dejar de funcionar correctamente, ya sea durante su compilación o durante la ejecución del software.

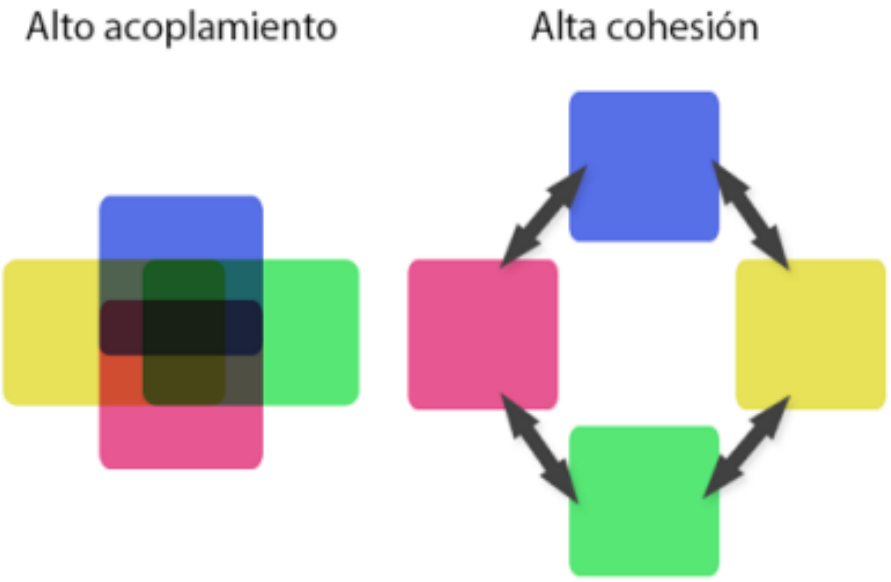

Imagen 6. Acoplamiento vs Cohesión.
Fuente: Desafío Latam

A continuación, veremos la segunda parte de los principios de desarrollo. Es el turno del conjunto de principios SOLID, una mezcla de principios un tanto complejos de entender, pero que si se llevan a la práctica el código creado será de una calidad excelente.

Principios SOLID

Los principios SOLID le deben su nombre a dos cosas, la primera y la más evidente es su traducción de la palabra en inglés: "ROBUSTO". La segunda razón es debido a los cinco principios de Programación Orientada a Objetos que lo conforman, que no son más que buenas prácticas para realizar un software de buena calidad. Los cinco principios que conforman SOLID son:

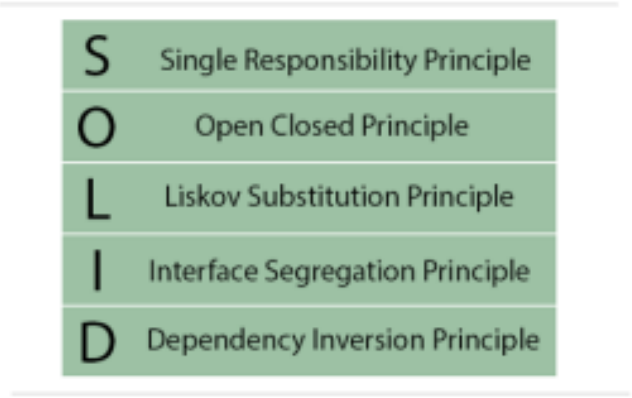

Imagen 7. Principios SOLID.
Fuente: Desafío Latam

A continuación, conoceremos la importancia y el significado de cada uno:

Single Responsibility Principle (Principio de responsabilidad única)

Este principio, aunque es fácil de explicar, es difícil de implementar y no es más que lo que su mismo nombre indica: cada objeto debe tener una única responsabilidad dentro del software.

Para este principio, la responsabilidad es la razón por la cual cambia el estado de una clase, es decir, darle a una clase una responsabilidad es darle una razón para cambiar su estado. Robert C. Martin es el creador de este principio y lo dio a conocer en su obra "Agile Principles, Patterns, and Practices in C#".

Si, por ejemplo, tenemos una clase PDF representando un archivo .pdf:

```
public class PDF{
 int paginas;
 String titulo;
}
```

Y quisiéramos hacer que el archivo se imprima, deberíamos crear otra clase que lo imprima y no hacer un método `imprimir()` dentro de la misma clase.

Ahora, teniendo una clase que imprima PDF, podríamos tener una clase Docx y no tendríamos que copiar y pegar el método imprimir, sino que podríamos usar la clase que imprime PDF para imprimir Docx.

Este principio ayudará a que las clases puedan ser reutilizadas fácilmente, gracias a que se transforman en algo más genérico al tener una sola responsabilidad dentro del software. Al regirnos por este principio ayuda a la cohesión del software, ya que cada clase está encargada de una función en específico y, por ende, puede ser modificada fácilmente.

Open Closed Principle (Principio Abierto - Cerrado)

Este principio dice que un objeto dentro del software, sea este una clase, módulo, función, etc., debe estar disponible para ser extendido (Abierto), pero no estarlo para modificaciones directas de su código actual (Cerrado).

Esto quiere decir que si, por ejemplo, tenemos un objeto con los atributos:

```
int valor;
String nombre;
```

Deberíamos agregar nuevos atributos sin modificar valor ni nombre, que son los que ya existen. La razón de esto es simple, puesto que el atributo podría estar referenciado en otras partes del software y, al cambiar su nombre, la referencia se perdería.

Por ejemplo, si en el main llamamos al método getter de valor:

```
public int getValor(){
 return valor;
}
```

Y cambiamos la variable de la siguiente forma:

```
int valorActual;
```

El getter dejaría de funcionar, debido a que no podrá encontrar int valor dentro de la clase, ya que esta ahora es int `valorActual`. Por ende, tendríamos que cambiar el getter:

```
public int getValorActual(){
 return valorActual;
}
```

Y ahora deberíamos cambiar todas las referencias que hay hacia `getValor()` para que sean `getValorActual()`. En el fondo, es un problema cambiar la variable de nombre y el ejemplo anterior es el menor de los problemas que podría ocasionar el cambio del nombre de una variable. Es por eso que se debe hacer un análisis y al menos un diagrama de clases de lo que será el proyecto que se vaya a realizar en cualquier caso.

Aplicando el principio "Abierto-Cerrado" conseguirás una mayor cohesión, mejorarás la lectura y reducirás el riesgo de romper alguna funcionalidad ya existente.

Liskov Substitution Principle (Principio de sustitución de Liskov)

Este principio nos indica que cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. Este principio fue creado por Barbara Liskov y dicta la forma correcta de utilizar la herencia.

Imagen 8. Liskov Substitution Principle.
Fuente: Desafío Latam

Por ejemplo, si tenemos una clase `Gato` que extiende de `Animal`, y la clase `Animal` tiene el método `volar()`, la herencia deja de ser válida. Es bastante evidente que los gatos no pueden volar, por ende, siempre que se haga una herencia se debe pensar que todas las subclases de una clase realmente utilicen los métodos y atributos que están heredando.

Interface Segregation Principle (Principio de segregación de interfaces)

Este principio dicta que las clases que implementen una interface deberían utilizar todos y cada uno de los métodos que tiene la interface. Si no es así, la mejor opción es dividir la interface en varias, hasta lograr que las clases solo implementen métodos que utilizan. Los clientes no deberían verse forzados a depender de interfaces que no usan.

Imagina que haces una interface llamada `VehiculoMotorizado` y tres clases que le hereden: `Auto`, `Lancha` y `Avión`. Le agregamos el método `despegar()` y `encenderMotor()`. Sin embargo, al hacer esto estaríamos violando el principio de segregación de clases, ya que un auto no debería ocupar el método despegar. La solución a esto sería crear más interfaces hasta lograr que todos los que heredan de estas necesiten hacerlo, tal como se muestra en la imagen 10.

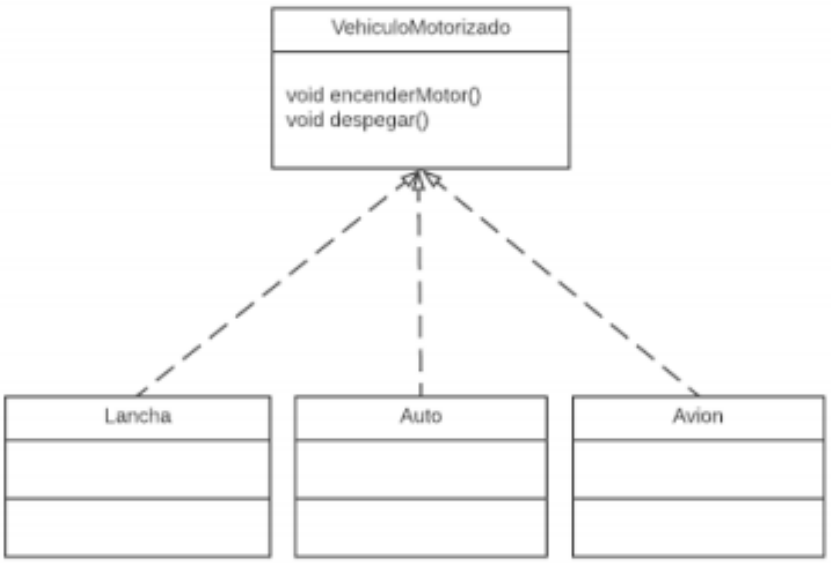

Imagen 9. Interface Segregation Principle.
Fuente: Desafío Latam

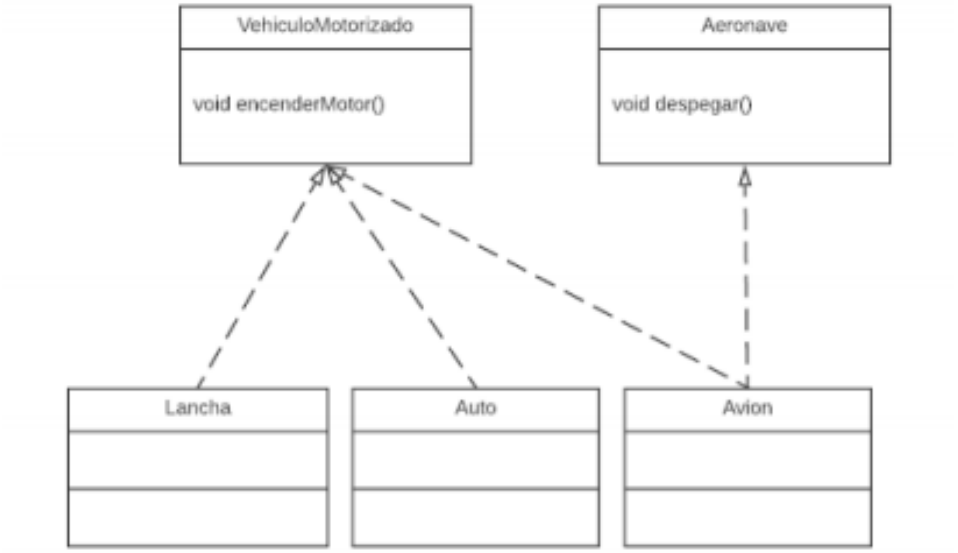

Imagen 10. Aplicando el principio de segregación de clases.
Fuente: Desafío Latam

Dependency Inversion Principle (Principio de inversión de dependencias)

Este principio indica que los módulos de alto nivel no deben depender de módulos de bajo nivel. Las abstracciones no deberían depender de los detalles, sino que los detalles deberían depender de las abstracciones.

Cuando hay una fuente externa de datos, como por ejemplo, un usuario ingresando datos, se debe hacer que el módulo del software donde se reciben los datos ingresados por el usuario (módulo de bajo nivel) no sea parte del módulo donde los datos se procesan (módulo de alto nivel). Esto permite que el módulo donde se reciben los datos pueda ser reemplazado fácilmente por uno diferente, manteniendo el módulo que los procesa intacto y haciéndolo reutilizable.

Esta reutilización se hace incluso sin necesidad de reemplazar el módulo de ingreso de datos, recibiendo estos datos paralelamente de diferentes fuentes sin estar ligado a ninguna de ellas. Por ejemplo, si tenemos un botón que enciende y apaga una lámpara, y lo hiciéramos sin aplicar el principio de inversión de dependencias, se vería algo así:

```
class Boton{
   Lampara lamp;
   void presionarBoton(Boolean presionado){
   this.lamp.encenderApagar(presionado);
  }
}

class Lampara{
   boolean encendido;
   void encenderApagar(Boolean presionado){
   this.encendido = presionado;
  }
}
```

Analicemos lo anterior. Si tenemos una "interface de usuario" llamada `Boton` que es un código de bajo nivel (ya que recibe datos del usuario) y que se comunica con una lámpara en concreto, encendiéndose o apagándose gracias a su método `encenderApagar()`; la lógica para apagar cuando está encendida y encender cuando está apagada, se considera de alto nivel, ya que es la que procesa la información.

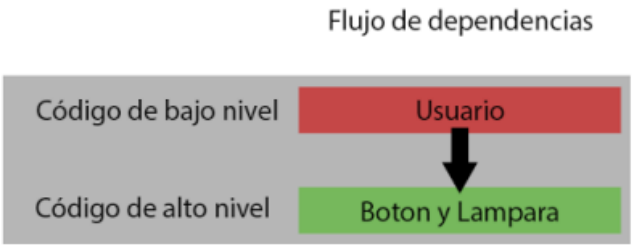

Imagen 11. Flujo de Dependencias.
Fuente: Desafío Latam

En este caso, estamos haciendo qué `Lampara`, que es el código de alto nivel, dependa del `Boton`, por ende estamos violando el principio Inversión de Dependencias. Para solucionar esto, debemos crear una interfaz de `Boton`... te preguntarás, ¿para qué necesitas una interfaz de `Boton` si puedes dejarlo como una clase y usarlo cuando sea necesario instanciarlo?

La respuesta es que haciendo una interface podrás usar el mismo botón para encender y apagar cualquier otro aparato, y esto se rige por la segunda parte que dicta el principio de inversión de dependencias: las abstracciones no deberían depender de los detalles, sino que los detalles deberían depender de las abstracciones.

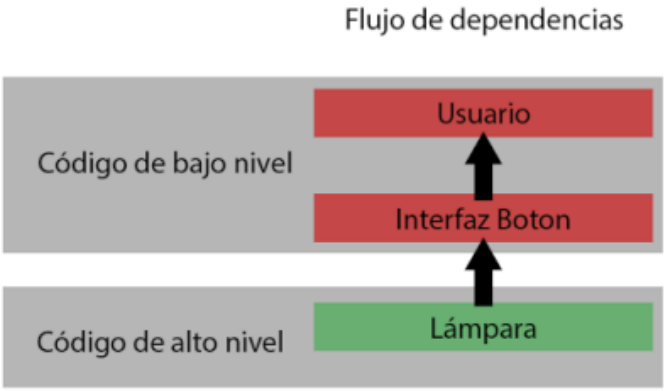

Imagen 12. Flujo de Dependencias para Lampara.
Fuente: Desafío Latam

Haremos que la lámpara dependa de la abstracción de botón, abstracción que servirá para cualquier otro aparato que pueda necesitar un botón que lo encienda o apague.

```
interface Boton{
   void presionarBoton(Boolean presionado);
}
class Lampara implements Boton{
   public boolean encendido;
 @Override
   void presionarBoton(Boolean presionado){
   this.encendido = presionado;
  }
}
```

Aunque es un ejemplo simple debido a su reducido nivel de complejidad, este principio es muy importante, al igual que los otros cuatro, debido a que aumenta muchísimo la cohesión del código, haciendo que solo dependa de abstracciones y no existan clases acopladas como el `Boton` y la `Lampara` de la primera versión del código.

Ejercicio propuesto (2)

- Dividir en parejas según la cantidad de principios (10) y elegir uno.

- En 5 minutos buscar por internet información sobre el principio elegido para responder las siguientes preguntas: ¿para qué sirve el principio? y ¿qué ejemplo de problema resolverían con ese principio?

- Exponer las respuestas en un máximo de 3 minutos, junto con una imagen o "meme" que represente al principio resaltando sus principales características.

Instancias únicas

Competencias

- Comprender el Patrón de diseño Singleton para obtener un código limpio al ejecutar.
- Comprender el concepto Synchronized para evitar ejecutar hilos en paralelo.

Introducción

Conoceremos el patrón Singleton de POO, un patrón básico de la programación que logra reducir el uso de memoria si es bien implementado en el software. Es una instancia que se utiliza día a día por múltiples desarrolladores alrededor del mundo, varios/as utilizan este patrón para hacer solicitudes o "request" a través de la web, por ejemplo. Es esencial aprender este concepto para realizar instancias únicas e irrepetibles.

A continuación, veremos qué es Singleton y cómo se utiliza. ¡Comencemos!

¿Qué es Singleton?

Singleton es un patrón de diseño de software que se caracteriza porque los objetos del software se rigen por el patrón, solo se instancian una vez y esa instancia es la que se utilizará en toda la aplicación. Te preguntarás de qué sirve aplicar este patrón... pues reduce la cantidad de instancias durante la ejecución de la aplicación y ayuda a tener un código más limpio porque también reduce la cantidad de variables. La ventaja más importante es que si hay varias partes de la aplicación que comparten un mismo recurso, podrán acceder a él desde cualquier parte.

La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto, el mismo objeto es siempre compartido por distintas partes del código. Es visto como una solución elegante para una variable global, porque los datos son abstraídos por detrás de la interfaz que publica la clase singleton.

¿Cómo se aplica el patrón Singleton?

Para aplicarlo se debe crear una clase que se instancie a sí misma en un contexto `static` y que tenga un constructor privado para que no se pueda acceder a él. Por último, se debe crear un método `static` para que las otras clases puedan acceder a la instancia existente.

Realicemos un ejercicio guiado para ponerlo en práctica:

Ejercicio Guiado: Instituto Educativo

Paso 1: Crear la clase `InstitutoEducativo` y colocar una variable del mismo tipo que la clase pero llamada “instance”. Aquí reside el secreto de este patrón, ya que dicha variable es la que se instancia por única vez y se devuelve al cliente.

```
public class InstitutoEducativo {
   private static InstitutoEducativo instance;
   }
```

Paso 2: Privatizar el constructor para que no se pueda hacer new `InstitutoEducativo()` desde otro lugar que no sea dentro de la misma clase.

```
 private InstitutoEducativo() {}
```

Paso 3: Para utilizar la única instancia de la clase, los clientes deberán convocar al método `getInstance()`. Crear la condición `if` que solo será `true` la primera vez.

```
public static InstitutoEducativo getInstance() {
      if (instance == null) {
            instance = new InstitutoEducativo();
      }
      return instance;
   }
```

Realizando los pasos 1, 2 y 3 quedaría de la siguiente manera:

```
public class InstitutoEducativo {
   private static InstitutoEducativo instance;
   private InstitutoEducativo() {}
   public static InstitutoEducativo getInstance() {
      if (instance == null) {
            instance = new InstitutoEducativo();
      }
      return instance;
   }
}
```

Paso 4: Para llamar al instituto, obtener la instancia del instituto en el Main.

```
public class Main{

public static void main(String[] args){
InstitutoEducativo instituto = InstitutoEducativo.getInstance();
 }
}
```

Otro ejemplo de Singleton

Ahora veamos otro ejemplo donde podemos ocupar Singleton. Para ello crearemos la clase `Configurador()` en un proyecto aparte y definiremos los siguientes métodos e instancias:

```
public class Configurador{

 //Variable encapsulada y estática donde se almacenará la instancia.

  private static Configurador config;

 //Constructor privado para que no se pueda hacer un new Configurador()
desde otro lugar que no sea dentro de la misma clase

 private Configurador() {}

 //Método estático encapsulador para acceder a la instancia única

public static Configurador getConfig() {
 if (config== null) {
 config= new Configurador();
 }
 return config;
 }
}
```

A pesar de que este código valida si existe una instancia antes de crearla, puede provocar un error si hay dos usuarios ejecutando el método al mismo tiempo, ya que ambos usuarios podrían entrar a `if(config==null)`, creando una instancia del Configurador para cada uno.

Cuando trabajamos con Java, cada vez que se ejecuta un método o algoritmo se crea un "hilo" y este se usará hasta que termine la ejecución. Hay algunos algoritmos que tardan milésimas de segundo y otros que duran meses ejecutándose, todo depende de su complejidad.

Por defecto, en aplicaciones online cada usuario genera su propio hilo al ejecutar un método, esto permite que la aplicación no se bloquee con un cuello de botella si dos usuarios quieren ejecutar el mismo método.

Synchronized

Cuando hablamos de Singleton, trabajaremos de una manera diferente a la que tienen por defecto los hilos. En el ejemplo anterior, necesitamos que la ejecución del método `getConfig()` se haga de manera sincronizada y así evitar tener hilos en paralelo al crear dos instancias de la misma clase. Para esto, utilizaremos la palabra reservada "synchronized".

```
public class Configurador{
 private static Configurador config;
 private Configurador() {}

public static Configurador getConfig() {
  if (config == null) {

 synchronized(Configurador.class) {

  if (config == null) {
   config = new Configurador();
   System.out.println("Instancia creada");
    }
  }
}
  System.out.println("Llamada al Configurador");
  return config
}
```

En el código anterior, podemos ver que si no existe una instancia `config == null`, se procede a iniciar una porción de código que está sincronizada mediante una sentencia **if**. Esto con la finalidad de que si hay dos o más usuarios tratando de acceder al mismo método, el primer usuario bloqueará el acceso al segundo. Para demostrar este ejemplo, podemos llamar al método `getConfig()` 2 veces en la clase `Main` y corremos nuestro programa.

```
public class Main{
public static void main(String[] args){
 Configurador.getConfig();
 Configurador.getConfig();
}
```

La consola nos arrojará como resultado que la instancia se crea con la primera persona y esta no se vuelve a crear cuando la segunda persona acceda a la validación de la instancia.

```
   ---------------------------
   Impresión en pantalla:

   Instancia Creada
   Llamada al configurador
   Llamada al configurador
```

Ejercicio Propuesto (3)

Dado el ejercicio anterior, ahora debemos agregar lo siguiente:

- Un atributo del tipo String llamado `nombreInstituto`, generar los getter and setter correspondientes dentro de la clase InstitutoEducativo.
- En la clase Main generar la llamada de la clase mediante `getInstance()`, luego llamamos a esa variable declarada y le pasamos como variable “Educación S.A.”.
- Crear la instancia para imprimir cada instituto con un número identificatorio. Es decir, debería imprimir de la siguiente manera:

  `System.out.println("1: " + instituto3.getNombreInstituto());`.
- Cambiamos el tercero mediante un setNombreInstituto a “Capacitación S.A.”
- Imprimimos el valor con la obtención del nuevo instituto agregado.

Palabras de cierre

Hemos visto varios conceptos asociados a la Programación Orientada a Objetos, estos son la base que debe tener todo programador/a a la hora de realizar código. Sin embargo, existen otros principios que también pertenecen a las buenas prácticas del mundo del desarrollo.

Soluciones ejercicios propuestos

Solución Ejercicio Propuesto (1)

Paso 1: Crear la clase **abstracta** `Herbivoro`.

```
import java.util.List;

public abstract class Herbivoro extends Animal{
    List<String> vegetalesFavoritos;

public List<String> getVegetalesFavoritos() {
      return vegetalesFavoritos;
   }
public void setVegetalesFavoritos(List<String> vegetalesFavoritos) {
this.vegetalesFavoritos = vegetalesFavoritos;
  }
}
```

Paso 2: Crear la clase `Conejo` que extiende de `Herbivoro`.

```
//Conejo
public class Conejo extends Herbivoro{
}
```

Paso 3: Crear la clase `Caballo` que extiende de `Herbivoro`.

```
//Caballo
public class Caballo extends Herbivoro{
}
```

Paso 4: Agregar un método `main` para crear esta lista polimórfica con clases abstractas.

```
public class Main {
  public static void main(String[] args) {
     Leon leon = new Leon();
     Tigre tigre = new Tigre();
     Caballo caballo = new Caballo();
     Conejo conejo = new Conejo();
     ArrayList<Animal> listaAnimales = new ArrayList<>();
     ArrayList<Carnivoro> listaCarnivoros = new ArrayList<>();
     ArrayList<Herbivoro> listaHerbivoros = new ArrayList<>();
     listaAnimales.add(leon);
     listaAnimales.add(tigre);
     listaAnimales.add(conejo);
     listaAnimales.add(caballo);
     listaCarnivoros.add(leon);
     listaCarnivoros.add(tigre);
     listaHerbivoros.add(conejo);
     listaHerbivoros.add(caballo);
     System.out.println(listaAnimales);
     System.out.println(listaCarnivoros);
     System.out.println(listaHerbivoros);
   }
}
    ---------------------------
    Impresión en pantalla:
    //Lista completa animales
    [desafio1.Leon@58372a00, desafio1.Tigre@4dd8dc3,
desafio1.Conejo@5fd0d5ae, desafio1.Caballo@2d98a335]

    //Lista Carnivoros
    [desafio1.Leon@58372a00, desafio1.Tigre@4dd8dc3]

    //Lista Herbivoros
    [desafio1.Conejo@5fd0d5ae, desafio1.Caballo@2d98a335]
```

Solución Ejercicio Propuesto (3)

Paso 1: Crear el atributo correspondiente en la clase `InstitutoEducativo`.

```
public class InstitutoEducativo {
   private static InstitutoEducativo instance;
   private String nombreInstituto;

   public String getNombreInstituto() {
    return nombreInstituto;
}

public void setNombreInstituto(String nombreInstituto) {
    this.nombreInstituto = nombreInstituto;
}

private InstitutoEducativo() {}

   public static InstitutoEducativo getInstance() {
      if (instance == null) {
            instance = new InstitutoEducativo();
      }
      return instance;
   }
}
```

Paso 2: En la clase `Main` crear lo siguiente por cada instituto.

```
public class Main {

  public static void main(String[] args) {

    InstitutoEducativo instituto = InstitutoEducativo.getInstance();
    instituto.setNombreInstituto("Educación S.A.");
    System.out.println("1: " + instituto.getNombreInstituto());

    InstitutoEducativo instituto2 = InstitutoEducativo.getInstance();
    System.out.println("2: " + instituto2.getNombreInstituto());

    InstitutoEducativo instituto3 = InstitutoEducativo.getInstance();
    System.out.println("3: " + instituto3.getNombreInstituto());
    instituto3.setNombreInstituto("Capacitación S.A.");
    System.out.println("1 bis: " +instituto.getNombreInstituto());
  }
}

    ---------------------------
    Impresión en pantalla:
    1: Educación S.A.
    2: Educación S.A.
    3: Educación S.A.
    1 bis: Capacitación S.A
```