Programación orientada a objetos

Programación con principios

{desafío} latam_

Utilizar principios básicos de diseño orientado a objetos para la implementación de una pieza de software acorde al lenguaje Java para resolver un problema de baja complejidad

- Unidad 1: Flujo, ciclos y métodos
- Unidad 2: Arreglos y archivos
- Unidad 3: Programación orientada a objetos
- Unidad 4: Pruebas unitarias y TDD

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

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

{desafío} latam_

¿Qué entendemos por principios de la programación?

{desafío} latam_

/* Principios */

Principio de modularización

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
- 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.

Principio DRY

(Don't Repeat Yourself)

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 reutilizarlo donde sea necesario.

Principio DRY

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;
          }
```

Principio DRY

Ejemplo

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;
          }
     }
```

Principio DRY

Ejemplo

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

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

Principio KISS

Keep It Simple Stupid

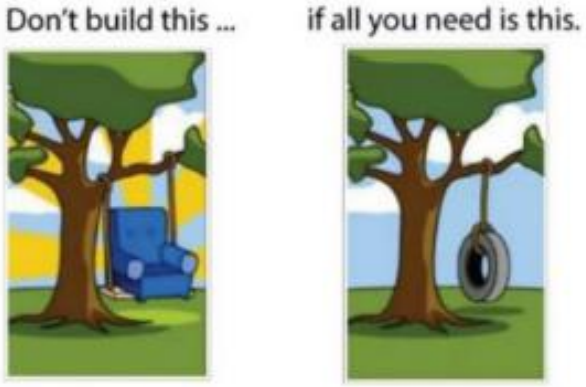
Crear un software sin necesidad de hacerlo más complejo. De esta forma es más fácil de entender y utilizar.

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.

Principio YAGNI

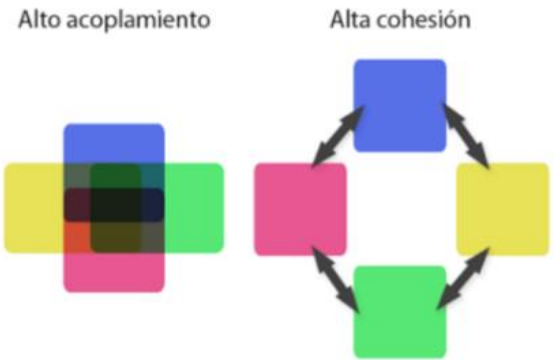
You Aren't Gonna Need It

No se debe agregar piezas que no se utilizarán.



Principio de Acoplamiento y Cohesión

Tight & Loose



Acoplamiento

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.

Cohesión

Mide la "fuerza" con que las partes o piezas de un software están conectadas dentro de un módulo. Además, el código facilita el entendimiento de los desarrolladores debido a su grado de reutilización del mismo.

Principios SOLID

ROBUSTO

5 principios de Programación Orientada a Objetos que lo conforman, que no son más que buenas prácticas para realizar un software de buena calidad.

Principios SOLID

Single Responsibility Principle (Principio de responsabilidad única)

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.

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.

Principios SOLID

Open Closed Principle (Principio Abierto - Cerrado)

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).

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

Principios SOLID

Liskov Substitution Principle (Principio de sustitución de Liskov)

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

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.

Principios SOLID

Interface Segregation Principle (Principio de segregación de interfaces)

- Las clases que implementen una interface deberían utilizar todos y cada uno de los métodos que tiene la interface.

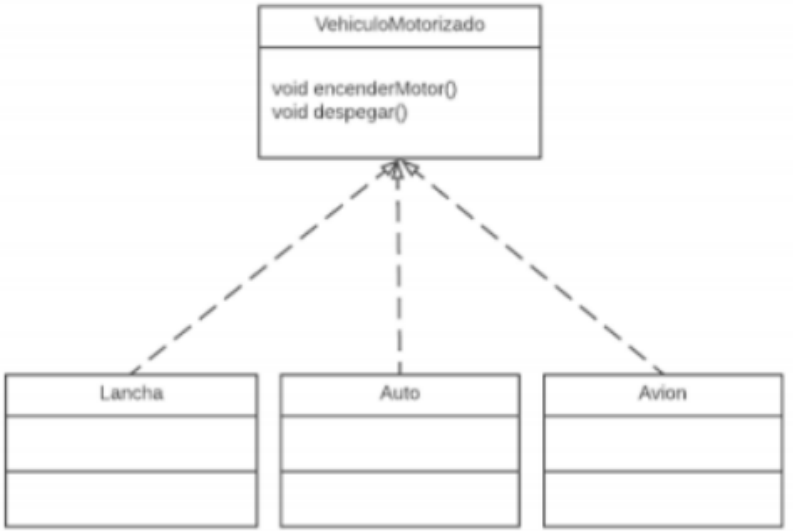
- Si no es así, la mejor opción es dividir la interface en varias, hasta lograr que las clases solo implementen métodos que utilizan.

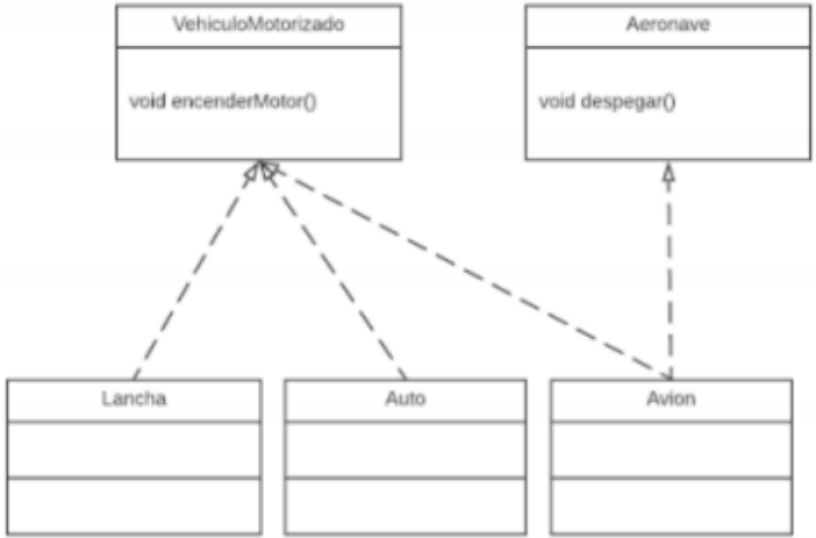
- Los clientes no deberían verse forzados a depender de interfaces que no usan.

Principios SOLID

Interface Segregation Principle (Principio de segregación de interfaces)

Ejemplo:





Principios SOLID

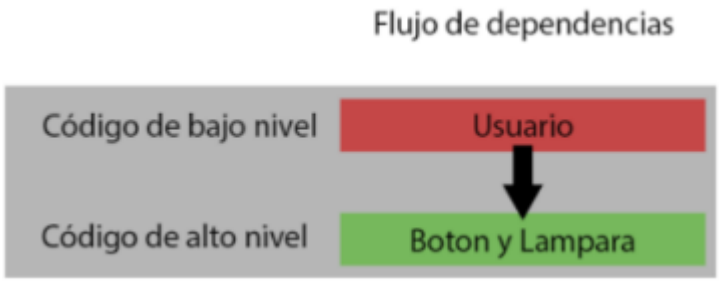
Dependency Inversion Principle (Principio de inversión de dependencias)

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.

Principios SOLID

Dependency Inversion Principle (Principio de inversión de dependencias)

Ejemplo:

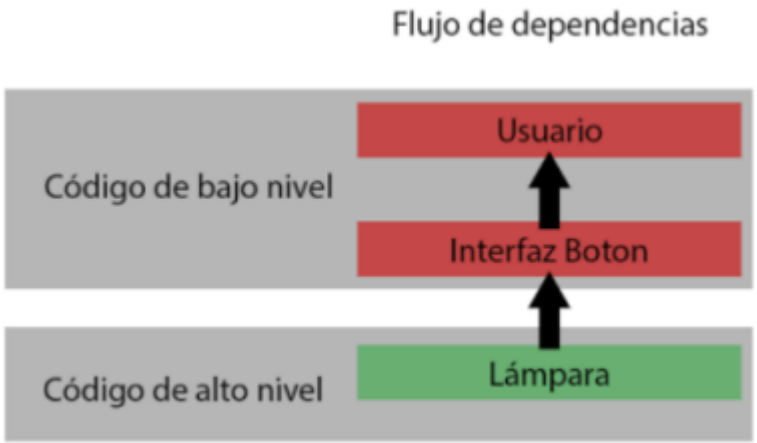



```
class Boton{
     Lampara lamp;
     void presionarBoton(Boolean presionado){
          this.lamp.encenderApagar(presionado);
     }
}
class Lampara{
     boolean encendido;
     void encenderApagar(Boolean presionado){
          this.encendido = presionado;
     }
}
```

Principios SOLID

Dependency Inversion Principle (Principio de inversión de dependencias)

Ejemplo:




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

Propongamos una actividad para trabajar con los principios de programación

¿Qué es SOLID?

Próxima sesión...

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

{desafío} latam_

{desafío}
latam_

Academia de
talentos digitales