{desafío} latam_

Orientación a Objetos II - Parte II

Abstracción II

Competencias

- Implementar polimorfismo mediante herencia
- Conocer el principio de abstracción

Introducción

Continuaremos viendo conceptos de abstracción, como lo son las clases abstractas y el polimorfismo con ellas. Vamos a conocer también el principio de abstracción, para tener claridad de gran parte de lo que es la abstracción en POO.

Las clases abstractas

Las clases abstractas no son más que clases que no pueden instanciarse. De ahi 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**, o sea, que siempre necesita otro elemento en el que apoyarse para poder 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 imaginarse.

El que no puedan instanciarse radica en que son algo asi como clases que permiten categorizar a otras, como lo hemos estado viendo en la unidad anterior, una herencia donde se pueden ver animales contiene la clase Animal, la cual debería ser abstracta, ya que no existen ejemplares de Animal en el mundo, la palabra Animal la asociamos al concepto de animal, pero no es algo tangible.

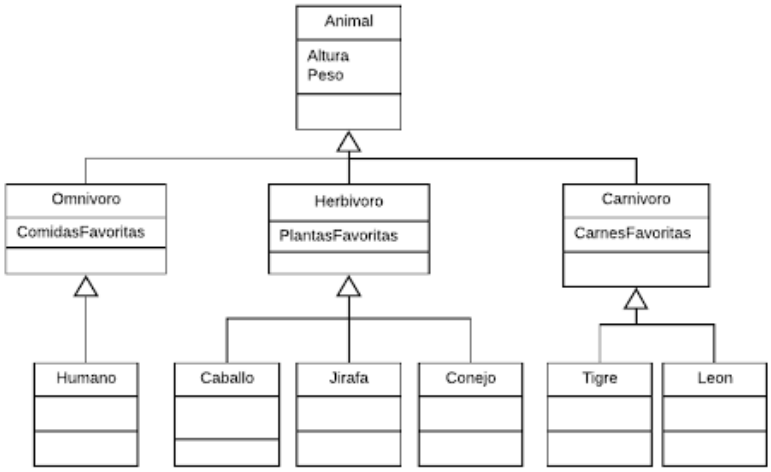

Imagen 1. Las clases abstractas en ejemplo Animal.

Creando clases abstractas.

Para poder crear una clase abstracta, basta con declararla como tal:

```
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;
  }
}
```

En la declaración de la clase, contiene la palabra reservada `abstract` antes de `class`, esto significa que la clase **no puede ser instanciada**.

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, vamos a terminar el árbol de carnívoros, creando la clase **abstracta** `Carnivoro` y luego las clases Tigre y Conejo:

```
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;
  }
}
```

Tigre:

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

León:

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

Tigre y León podrán tener todos los atributos de `Carnivoro` y `Animal` y podrá hacerse una lista con carnívoros aplicando polimorfismo, vamos a agregar un método main como lo habíamos estado haciendo antes en otra clase Main, para poder crear esta lista polimórfica con clases abstractas:

```
package main;

import java.awt.List;
import java.util.ArrayList;

import Modelo.Animal;
import Modelo.Carnivoro;
import Modelo.Leon;
import Modelo.Tigre;

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);
  }
}
```

Podemos agregar instancias de las subclases sin problemas a la lista, osea, se puede utilizar polimorfismo y 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 reutilizarlos en todas las subclases.

El principio de abstracción en POO

Vamos ahora a ver el concepto de abstracción en POO, que es algo asi como el conjunto de todo lo que hemos estado aprendiendo previamente, ya que la abstracción es la realización del polimorfismo, trata de buscar características en común de objetos para poder trabajarlos dentro de un mismo contexto.

Imagen 2. Principio de abstracción.

Por ejemplo, estuvimos haciendo una herencia con animales y vimos que todos ellos tienen un peso y altura determinados, por ende, si quisiéramos mejorar la aplicación para hacer una para gestionar el transporte de animales, debemos medirlos y pesarlos antes de subirlos a su jaula de transporte, ya que, no podemos subir a cinco leones de 100kg a un camión que soporta 300kg.

Tendríamos las mismas clases que antes, pero necesitamos además agregar una interface que nos sirva para realizar las operaciones de carga del Camion, a la cual, le daremos el nombre Cargable:

```
public interface Cargable {
  void almacenarCarga(Object carga);
}
```

y ahora implementaremos esta interfaz en un Camión

```
public class Camion implements Cargable{

  public List<Object> carga;
  public double cargaMaxima;

  @Override
  public void almacenarCarga(Object carga) {

  }
  //Getters y Setters...
}
```

Podemos guardar la instancia que representa a ambos animales en una misma lista, que sería la lista del camión al que se subirían para ser transportados y validar que la suma del peso de todos los animales que ya están en el camión, más el animal que está a punto de subirse, no supere el peso máximo del camión. Esto lo haremos en un nuevo paquete llamado servicios, puesto que el camión no puede calcular por si solo esta operación, vamos a crear un servicio para que haga este cálculo, el cual será una clase abstracta:

```
public abstract class ServicioCalculo {

  public static boolean esPosibleSubirNuevoAnimal(Camion camion, Animal
nuevoAnimal) {
    double peso = 0;
    //Recorremos la lista de animales del camion (carga)
    for(Object objeto : camion.getCarga()) {
      //Casteamos el objeto actual a la clase Animal
      Animal animal = (Animal) objeto;
      //Se suman todos los pesos para obtener la carga actual
      peso = peso + animal.getPeso();
    }

    //Se compara el peso de la carga actual más el peso del nuevoAnimal, versus
la carga máxima
    // Si la carga maxima es inferior a la suma, entonces
    // es posible agregar el nuevo animal y se devuelve true.
    // En el caso contrario, se devolverá false
    return (camion.getCargaMaxima() < (nuevoAnimal.getPeso()+peso));
  }
}
```

Recuerda que el método debe ser estático, sino, no podrá ser utilizado en el main. Ahora usaremos este servicio en el main, para ver si es posible subir al animal antes de hacerlo:

```
public class Main {

  public static void main(String[] args) {
    Camion camion = new Camion();
    camion.setCargaMaxima(1000);

    Leon leon = new Leon();
    leon.setPeso(200);

    if(ServicioCalculo.esPosibleSubirNuevoAnimal(camion, leon) == true) {
      camion.almacenarCarga(leon);
    }
  }
}
```

En este caso específico sólo nos interesa conocer el peso del animal en cuestión, siendo que hay muchos otros atributos que podría tener el animal. La abstracción dicta que, en base a un contexto específico, podamos utilizar una propiedad en común de dos o más objetos, osea que no necesitamos conocer los detalles de porqué ni cómo funcionan las clases; simplemente solicitamos determinadas acciones o atributos, en espera de una respuesta. En este caso, utilizamos el peso de los animales y un servicio de cálculo para validar que un animal pueda ser cargado antes de hacerlo, utilizando polimorfismo en varios puntos del sistema.

Programación con principios

Competencias

- Principios de diseño y modularización
- Principios de diseño orientado a objetos

Introducción

A continuación vamos a introducir una de las partes más importantes de la programación transversalmente y que no juega un papel menor en la programación orientada a objetos, estos son, los principios de programación. Pueden llamarse técnicas para desarrollar de una forma más acertada. Vamos a ver ejemplos básicos, pero dejando en claro qué es lo que significa cada uno de los principios y cuándo se están violando, para lograr de esta forma que cada uno de nuestros desarrollos en el futuro sean bien calificados.

Principio de modularización

Imagen 3. Ejemplo modularización con piezas de puzzle.

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 pueda ser más independiente, y por consecuencia, si se desea modificar uno, el impacto en los otros no será tan grande. Para lograr la modularidad, se debe atomizar un problema para obtener sub-problemas y que cada módulo atienda a dar solución a uno de los sub-problemas. 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

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 reutilizarlo 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, 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;
  }
}
```

Y entonces el primer código quedaría asi:

```
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 es bastante fácil de entender, se refiere a que crees el software sin hacerlo más complejo innecesariamente. De esta forma, es más fácil de entender y utilizar. La simplicidad es bastante bien aceptada en el diseño en todo ámbito y que mejor ejemplo que el de Apple, que creó un teléfono con un sólo botón para manejarlo.

Imagen 4. Principio KISS.

Principio YAGNI

- You Aren't Gonna Need It

Este principio indica que no se deberían agregar piezas que no se van a utilizar, 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, estaríamos agregando código que no necesitamos, es mejor enfocarse en lo realmente necesario y no perder tiempo en funcionalidades extra que no se han pedido (tu tiempo como desarrollador vale oro).

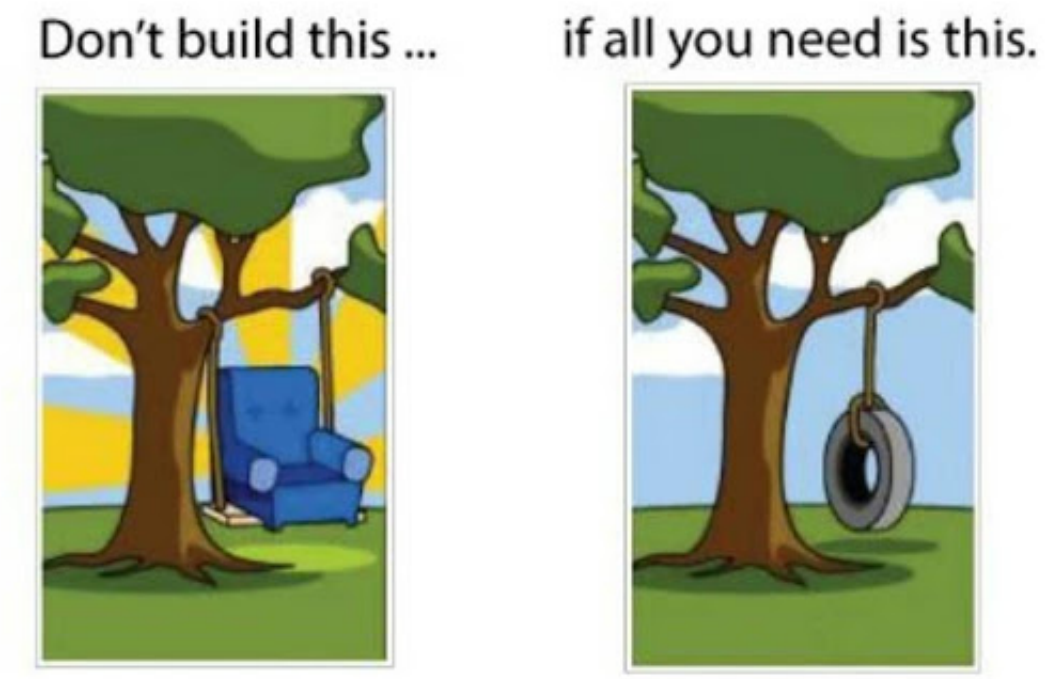

Imagen 5. Principio Yagni.

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 de su diseño.

Cohesión y Acoplamiento

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 estructurado (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 muchas cosas dentro del código y si algo se modifica, esas cosas que dependen del algo, podrían dejar de funcionar correctamente, ya sea durante su compilación o durante la ejecución del software.

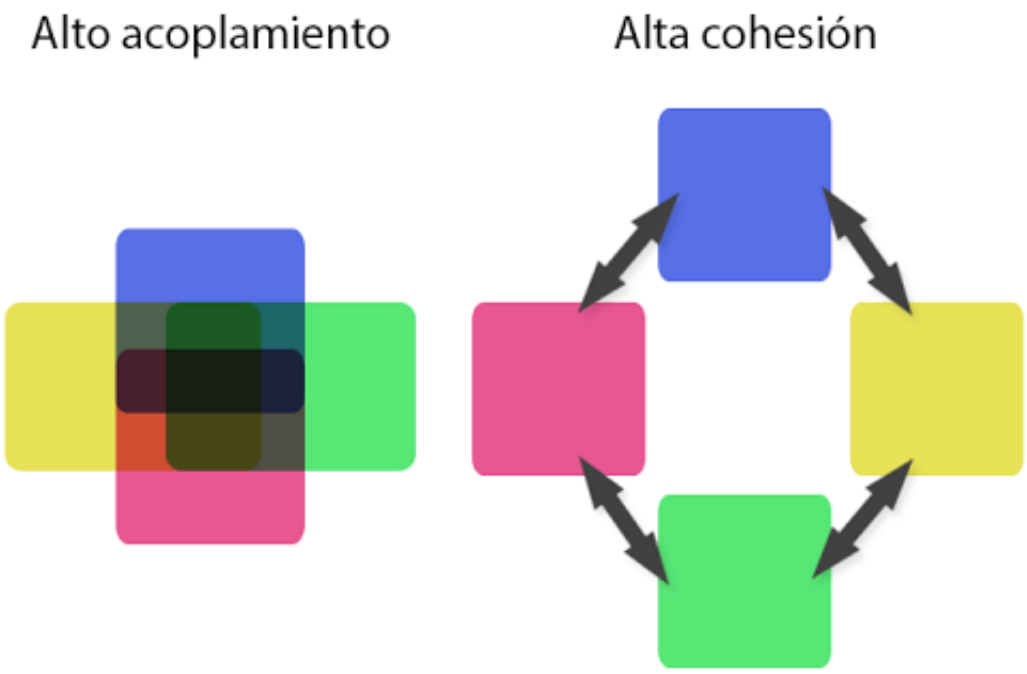

Imagen 6. Aclopamiento vs cohesión.

Programación con principios II

Competencias

- Conocer los principios SOLID y aplicarlos a un proyecto.

Introducción

A continuación veremos la segunda parte de los principios de desarrollo, ahora es el turno del conjunto de principios SOLID, una mezcla exquisita 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 de la palabra ROBUSTO del ingles SOLID; Y la segunda razón, son los cinco principios orientados a la programación orientada a objetos que lo conforman, estos, no son más que buenas prácticas que ayudan a hacer un software de buena calidad.

Los cinco principios que conforman SOLID, son:

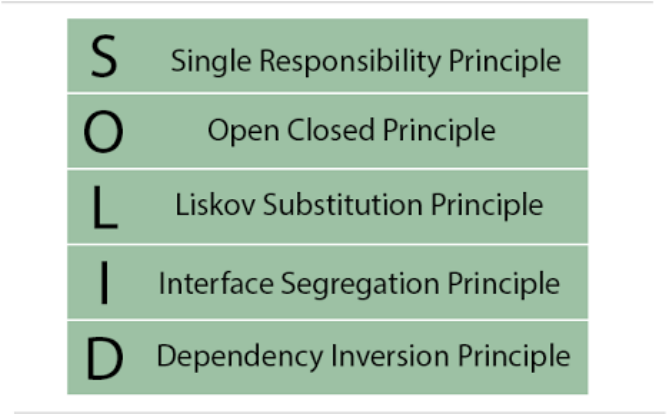

Imagen 7. Principios SOLID.

Vamos a conocer la importancia y que significa cada uno de ellos:

SRP – Single Responsibility Principle

- Principio de responsabilidad única

Éste 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. El principio dice que **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 quisieramos hacer que el archivo se imprima, deberíamos crear otra clase que lo imprima y no hacer un método `imprimir()` dentro de la clase misma.

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 deberían transformar en algo un poco más genérico al tener una sola responsabilidad dentro del software.

El regirse por este principio ayuda a la cohesión de un software, ya que cada clase está encargada de una función en específico del software, y por ende, puede ser modificada más fácil que si tuviese más de una responsabilidad.

OCP – 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 poder agregar nuevos atributos, pero no modificar valor ni nombre, que son los que ya existen.

La razón de esto es simple, y es 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 para que ahora sea asi:

```
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 esto 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 de abierto-cerrado conseguirás una mayor cohesión, mejorarás la lectura y reducirás el riesgo de romper alguna funcionalidad ya existente.

LSP - 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.

Este principio fue creado por Bárbara Liskov y dicta la forma correcta de utilizar la herencia:

Imagen 8. Liskov Substitution Principle.

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.

ISP - Interface Segregation Principle

- Principio de segregación de interfaces

> “Los clientes no deberían verse forzados a depender de interfaces que no usan”

El principio dicta que las clases que implementen una interfaz deben utilizar todos y cada uno de los métodos que tiene la interfaz y si no es asi, la mejor opción es dividir la interfaz en varias, hasta lograr que las clases sólo implementen métodos que utilizaran.

Imagina si hicieras una interfaz VehiculoMotorizado y además tres clases que le hereden, Auto, Lancha y Avión y a esta interfaz, le agregaras el método `despegar()`. De esta forma estarías violando el principio de segregación de clases.

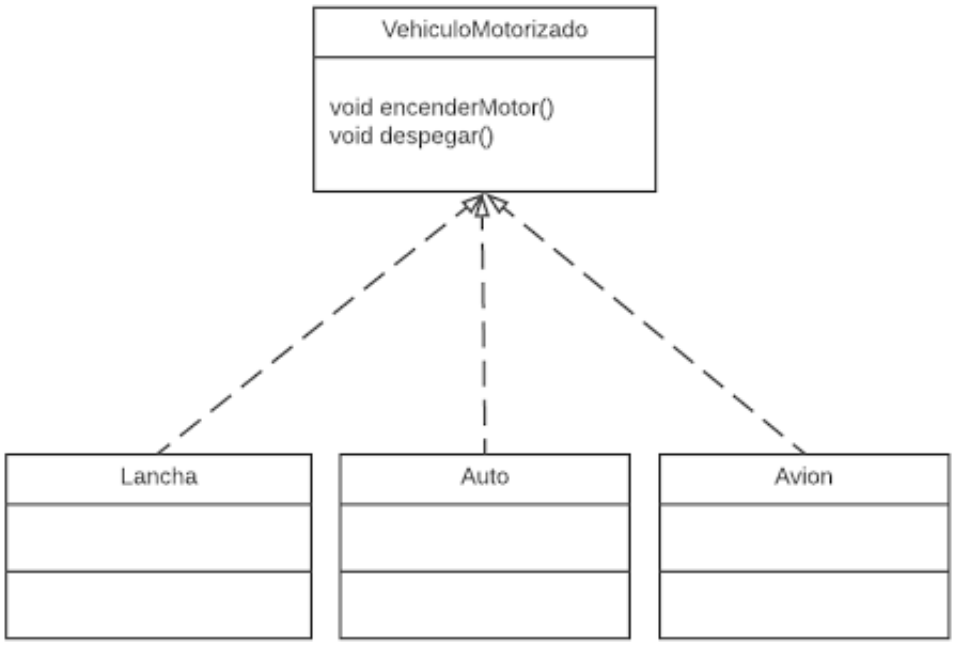

Imagen 9. Interface Segregation Principle.

La solución a esto, sería crear más interfaces hasta lograr que todos los que heredan de estas, necesiten hacerlo:

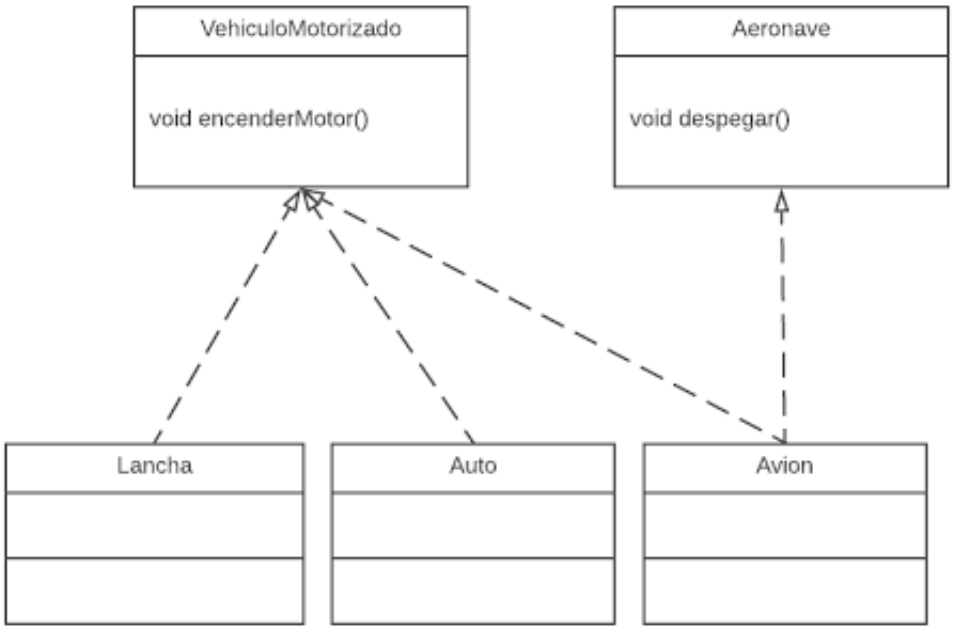

Imagen 10. Aplicando el principio de segregación por interfaces.

DIP – 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. 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 puede ser incluso sin necesidad de reemplazar el módulo de ingreso de datos, sino que puede recibir 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 hicieramos sin aplicar el principio de inversión de dependencias, ser vería algo asi:

```
class Boton{
  Lampara lamp;

  void presionarBoton(Boolean presionado){
    this.lamp.encenderApagar(presionado);
  }
}

class Lampara{
  boolean encendido;

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

Vamos a analizar esto, tenemos una "interfaz de usuario" llamada Botón, que es un código de bajo nivel, (ya que recibe datos del usuario) y que se comunica con una lámpara en concreto, encendiéndola o apagándola gracias a su método `encenderApagar()`, que contiene la lógica para apagar cuando está encendida y encender cuando está apagada, lógica que se considera de alto nivel, ya que es la que procesa la información.

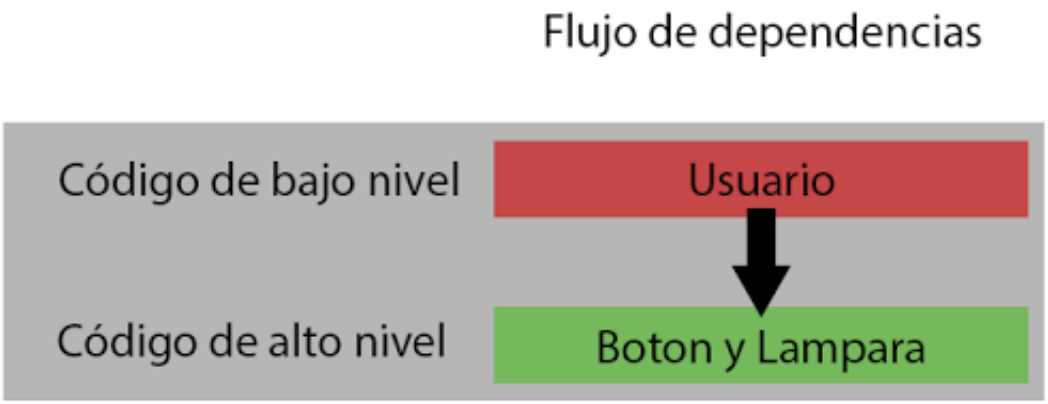

Imagen 11. Flujo de dependecias .

En este caso, estamos haciendo que la lámpara, que es el código de alto nivel, dependa del botón, por ende, estamos violando el principio DIP.

Para solucionar esto, debemos crear una interfaz de botón, te preguntarás: ¿para qué necesitas una interfaz de botón, si puedes dejarlo como una clase y usarlo cuando sea necesario instanciándolo?.

Es muy sencillo de responder y es que haciéndolo una interfaz, podrás usar el mismo botón para encender y apagar cualquier aparato, esto se rige por la segunda parte que dicta el principio de inversión de dependencias:

> Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

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

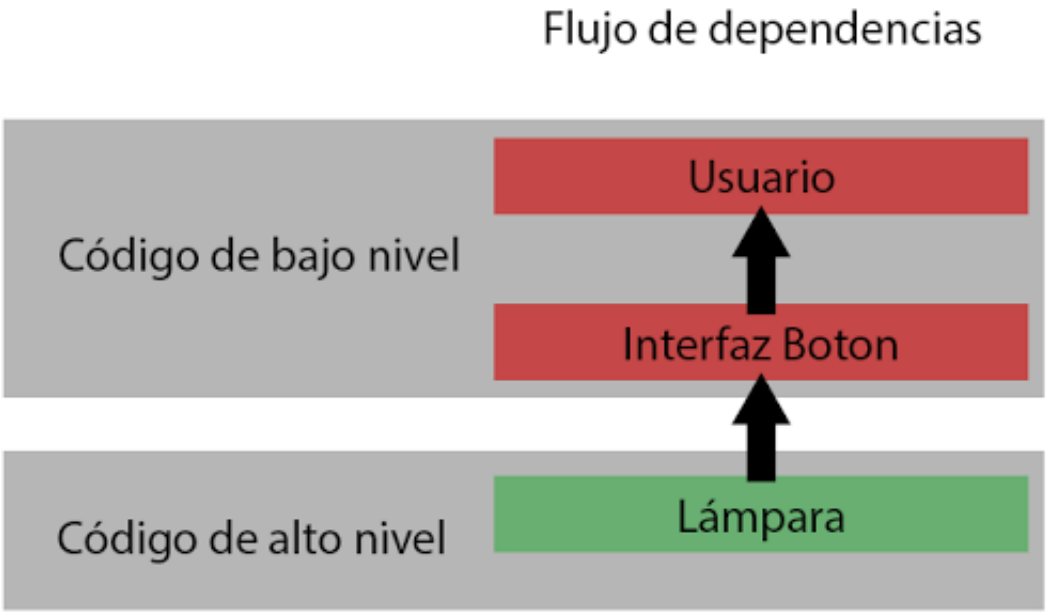

Imagen 12. Flujo de dependencias para lámpara.

```
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 absurdo 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 sólo se dependa de abstracciones y no existan clases acopladas como el Botón y la Lampara de la primera versión del código.

Practiquemos polimorfismo

Competencias

- Utilizar herencia para resolver un problema de baja complejidad
- Utilizar interfaces para resolver un problema de baja complejidad

Introducción

Crearemos una pequeña aplicación para el pago de remuneraciones de trabajadores utilizando polimorfismo con herencia y otra aplicación donde implementaremos polimorfismo con interfaces. Podremos practicar de esta forma, las dos formas que hemos conocido durante la unidad, con las que es posible implementar el polimorfismo en Java.

Aplicación para el pago de remuneración (Polimorfismo con herencia)

Ahora aplicaremos métodos abstractos y polimorfismo para realizar cálculos de pago de sueldos a empleados.

Una compañía paga a sus empleados en forma semanal y existen tres tipos de empleados:

- Empleados asalariados que reciben un sueldo semanal fijo, sin importar el número de horas trabajadas;
- Empleados por hora, que reciben un sueldo por hora y pago por tiempo extra;
- Empleados por comisión, que reciben un porcentaje de sus ventas

Para esto, vamos a crear un servicio estático que permita obtener los datos, los cuales serán:

- Andrés, 16313986-3;
    - Andrés es un empleado asalariado, con 180 dólares semanales.
- Camila, 8438298-1;
    - Camila es una empleada por hora, que recibe 3,5 dólares por hora y 5,3 por hora extra, que ha trabajado 47 horas esta semana y 7 de esas horas son hora extra.
- Rodrigo, 11800835-9;
    - Rodrigo es un empleado por comisión, que recibe un 19% de comisión por venta y esta semana ha realizado ventas por un total de 450 dólares.

Crearemos una clase abstracta que servirá como servicio, el cual tendrá un método que calculará el salario de los empleados.

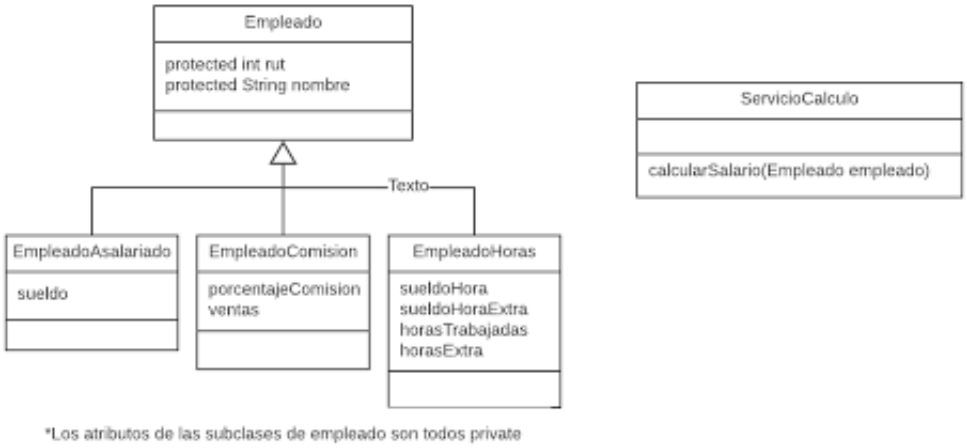

Imagen 13. Diagrama de clases.

Estructura del proyecto:

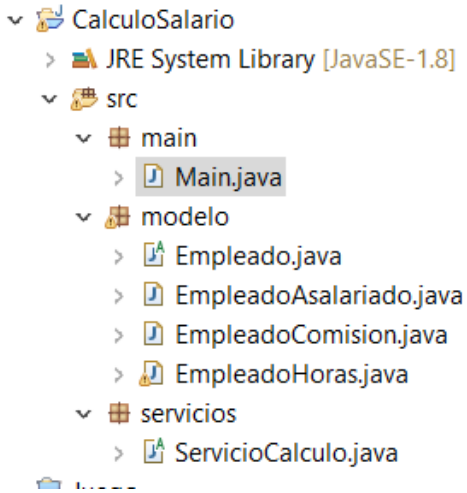

Imagen 14. Estructura del proyecto.

Clase abstracta Empleado:

```
public abstract class Empleado {
  protected int rut;
  protected String nombre;
  public Empleado(int rut, String nombre) {
    super();
    this.rut = rut;
    this.nombre = nombre;
  }
```

```
  public Empleado() {
    super();
  }
  public int getRut() {
    return rut;
  }
  public void setRut(int rut) {
    this.rut = rut;
  }
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
}
```

Subclase de Empleado, EmpleadoAsalariado:

```
public class EmpleadoAsalariado extends Empleado {

  double sueldo;

  public EmpleadoAsalariado(int rut, String nombre, double sueldo) {
    super(rut, nombre);
    this.sueldo = sueldo;
  }

  public EmpleadoAsalariado(int rut, String nombre) {
    super(rut, nombre);
  }

  public double getSueldo() {
    return sueldo;
  }

  public void setSueldo(double sueldo) {
    this.sueldo = sueldo;
  }
}
```

Subclase de Empleado, EmpleadoComision:

```
public class EmpleadoComision extends Empleado {
  private double porcentajeComision;
  private double ventas;

  public EmpleadoComision(int rut, String nombre, double porcentajeComision,
double ventas) {
    super(rut, nombre);
    this.porcentajeComision = porcentajeComision;
    this.ventas = ventas;
  }
  public EmpleadoComision(int rut, String nombre) {
    super(rut, nombre);
  }
  public double getPorcentajeComision() {
    return porcentajeComision;
  }
  public void setPorcentajeComision(double porcentajeComision) {
    this.porcentajeComision = porcentajeComision;
  }
  public double getVentas() {
    return ventas;
  }
  public void setVentas(double ventas) {
    this.ventas = ventas;
  }
}
```

Subclase de Empleado, EmpleadoHoras:

```
public class EmpleadoHoras extends Empleado {

  private double sueldoHora;
  private double sueldoHoraExtra;
  private int horasTrabajadas;
  private int horasExtra;

  public EmpleadoHoras(int rut, String nombre, double sueldoHora, double
sueldoHoraExtra, int horasTrabajadas, int horasExtra) {
    super(rut, nombre);
    this.sueldoHoraExtra = sueldoHoraExtra;
    this.sueldoHora = sueldoHora;
    this.horasTrabajadas = horasTrabajadas;
    this.horasExtra = horasExtra;
```

```java
  }

  public EmpleadoHoras(int rut, String nombre) {
    super(rut, nombre);
  }
  public double getSueldoHora() {
    return sueldoHora;
  }
  public void setSueldoHora(double sueldoHora) {
    this.sueldoHora = sueldoHora;
  }
  public int getHorasTrabajadas() {
    return horasTrabajadas;
  }
  public void setHorasTrabajadas(int horasTrabajadas) {
    this.horasTrabajadas = horasTrabajadas;
  }
  public int getHorasExtra() {
    return horasExtra;
  }
  public void setHorasExtra(int horasExtra) {
    this.horasExtra = horasExtra;
  }

  public double getSueldoHoraExtra() {
    return sueldoHoraExtra;
  }

  public void setSueldoHoraExtra(double sueldoHoraExtra) {
    this.sueldoHoraExtra = sueldoHoraExtra;
  }
}
```

El servicio de cálculo:

```
public abstract class ServicioCalculo {

  public static double calcularSalario(Empleado empleado) throws Exception {
    if(empleado.getClass().equals(EmpleadoComision.class)) {
      //Casteamos la instancia a EmpleadoComision
      EmpleadoComision empleadoComision = (EmpleadoComision) empleado;
      //Obtenemos el porcentaje en formato 0,19 para multiplicarlo por el total
de ventas y obtener el 19%
      double porcentajeMultiplicable = empleadoComision.getPorcentajeComision()
/ 100;
      //Devolvemos la siguiente multiplicacion
      return porcentajeMultiplicable * empleadoComision.getVentas();
    }
    else if(empleado.getClass().equals(EmpleadoHoras.class)) {
      //Casteamos la instancia a EmpleadoHoras
      EmpleadoHoras empleadoHoras = (EmpleadoHoras) empleado;
      double gananciaHoraNormal = empleadoHoras.getSueldoHora() *
empleadoHoras.getHorasTrabajadas();
      double gananciaHoraExtra = empleadoHoras.getSueldoHoraExtra() *
empleadoHoras.getHorasExtra();
      return gananciaHoraNormal+gananciaHoraExtra;
    }else if(empleado.getClass().equals(EmpleadoAsalariado.class)) {
      //Devolvemos el sueldo casteando la instancia para poder llegar al
atributo sueldo.
      return ((EmpleadoAsalariado) empleado).getSueldo();
    }
    else {
      //Si la instancia no es de ningun tipo de Empleado, devolvemos una
excepcion.
      throw new Exception();
    }
  }
}
```

En el Main, crearemos las instancias a los a los tres empleados:

```
public class Main {

  public static void main(String[] args) {
    //Creamos las instancias solicitadas
    //EmpleadoAsalariado(int rut, String nombre, double sueldo)
    EmpleadoAsalariado empAsalariado = new EmpleadoAsalariado(16313986,
"Andrés", 180);
    //EmpleadoComision(int rut, String nombre, double porcentajeComision,
double ventas)
    EmpleadoComision empComision = new EmpleadoComision(11800835, "Rodrigo",
19, 450);
    //EmpleadoHoras(int rut, String nombre, double sueldoHora, double
sueldoHoraExtra, int horasTrabajadas, int horasExtra)
    EmpleadoHoras empHoras = new EmpleadoHoras(8438298, "Camila", 3.5, 5.3, 40,
7);
  }
}
```

Luego agregar las tres instancias dentro de un arreglo, el cual vamos a recorrer calculando el salario correspondiente e imprimiendolo en el output:

```
//Creamos la lista y agregamos los empleados
ArrayList<Empleado> empleados = new ArrayList<>();
empleados.add(empAsalariado);
empleados.add(empComision);
empleados.add(empHoras);

//Recorremos el arreglo para imprimir el salario correspondiente
for(Empleado empleado :  empleados) {
  //Utilizamos un try catch por si se arroja la excepción del método
calcularSalario.
  //Si algúna excepción se arrojara, el ciclo continuará ejecutandose.
  try {
    System.out.println("El empleado "+empleado.getNombre()+" debe recibir:
"+ServicioCalculo.calcularSalario(empleado));
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

> El empleado Andrés debe recibir: 180.0 El empleado Rodrigo debe recibir: 85.5 El empleado Camila debe recibir: 177.1

Módulo para exportar archivos (Polimorfismo con interfaces)

Una empresa te ha pedido que crees un módulo en java para exportar datos en excel y en archivos txt para una aplicación de la cual no desean exponer el código, por ende, el módulo debe ser una aplicación que sólo reciba datos y los exporte, para que pueda ser agregado a la aplicación padre.

Los datos siempre vendrán en el mismo formato, una sola linea, con valores separados por coma. El módulo debe ser configurable desde un archivo externo, ya que el cliente utilizará el módulo en dos partes diferentes, una que debe exportar en txt y otra que debe exportar en excel, además del formato del archivo, se debe poder especificar la ruta. El nombre del archivo siempre debe ser Valores_Sobrantes*(fecha del dia en formato dd-MM-yyyy)*.(xls | txt)

El archivo txt recibe una lista de palabras y debe separarlas en filas.

El archivo excel recibe una lista de números y deben estar todos en una misma columna.

Para realizar este módulo, vamos a necesitar descargar la librería poi de apache, que permitirá exportar archivos excel y txt más fácilmente.

> Enlace librería: http://central.maven.org/maven2/org/apache/poi/poi/4.1.0/poi-4.1.0.jar

La aplicación a desarrollar se realizará en base al siguiente diagrama:

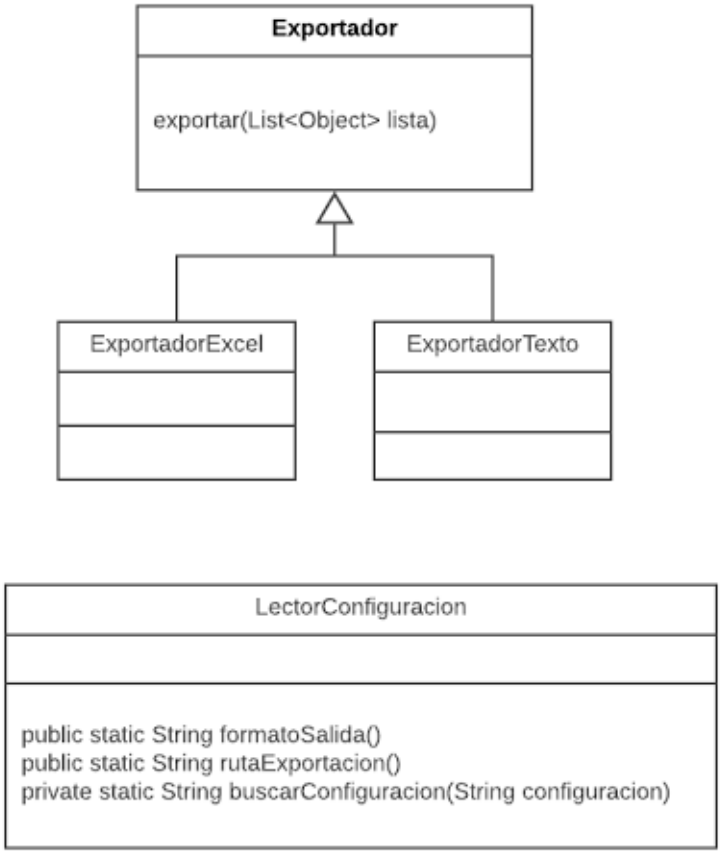

Imagen 15. Diagrama para módulo de exportación.

Tendrá la siguiente estructura:

Imagen 16. Estructura del proyecto.

Utilizaremos la carpeta lib para guardar la libreria apache poi (http://central.maven.org/maven2/org/apache/poi/poi/4.1.0/poi-4.1.0.jar) Vamos a crear la interfaz y sus implementaciones:

```
public interface Exportador {
  void exportar(List<Object> lista);
}
```

```
public class ExportadorExcel implements Exportador{

  @Override
  public void exportar(List<Object> lista) {
    String fechaHoyString = new SimpleDateFormat("dd-MM-yyyy").format(new
Date());
    HSSFWorkbook libro = new HSSFWorkbook();     // Creamos el libro de trabajo
de Excel
    HSSFSheet hoja = libro.createSheet();        // Creamos la hoja de Excel
        // Creamos las filas de Excel
    for(int i = 0; i < lista.size() ; i++) {
      //Casteamos el indice de la lista a String y luego parseamos el resultado
a Integer
      Integer numero = Integer.parseInt((String)lista.get(i));
      //creamos la fila
      HSSFRow fila = hoja.createRow(i);
      //creamos la celda en la primera columna
      HSSFCell celda = fila.createCell((short)0);
      //Insertamos el numero en la celda
      celda.setCellValue(numero);
    }
    //Exportamos el archivo
    try {
       FileOutputStream elFichero = new
FileOutputStream("Valores_Sobrantes_"+fechaHoyString+"_.xls");
       libro.write(elFichero);
       elFichero.close();
       libro.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
    try {
      libro.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
```

```
  }
}
```

```
public class ExportadorTexto implements Exportador{

  @Override
  public void exportar(List<Object> lista) {
    try {
      String fechaHoyString = new SimpleDateFormat("dd-MM-yyyy").format(new
Date());
            PrintWriter writer = new
PrintWriter("Valores_Sobrantes_"+fechaHoyString+"_.txt", "UTF-8");
            for(Object textObject : lista) {
              String text = (String) textObject;
                writer.println(text);
            }
            writer.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
  }
}
```

Ahora crearemos la clase abstracta que servirá para leer las configuraciones:

```
public abstract class LectorConfiguracion {

  public static String formatoSalida() throws IOException {
    return buscarConfiguracion("formatoSalida");
  }

  public static String rutaExportacion() throws IOException {
    return buscarConfiguracion("rutaSalida");
  }

  private static String buscarConfiguracion(String configuracion) throws
IOException{
      BufferedReader reader =
Files.newBufferedReader(Paths.get("configuracion.txt"));

        String linea;
        while ((linea = reader.readLine()) != null) {
          if(linea.contains(configuracion)) {
            return (linea.substring(linea.indexOf("=")+1, linea.indexOf(";")));
          }
```

```
        }
      throw new IOException("No se pudo encontrar el archivo configuracion.txt
en la carpeta del módulo", new Throwable("Archivo de configuraciones no
encontrado"));
  }
}
```

Esta clase, se encargará de leer el archivo configuracion.txt, el cual vamos a crear más adelante. Vamos a crear una clase main que exportará los valores que se le pasen como argumento al jar una vez finalizado el desarrollo.

```
public class Main {

  private static Exportador exportador;

  public static void main(String[] args) {
    if(null == args || args.length == 0) {
      System.out.println("No existen datos para exportar");
    }else {
      try {
        ArrayList<Object> lista = new ArrayList<Object>();
        for(String arg : args[0].split(",")) {
          lista.add(arg);
        }
        if(LectorConfiguracion.formatoSalida().equals("xls")) {
          exportador = new ExportadorExcel();
        }else if(LectorConfiguracion.formatoSalida().equals("txt")) {
          exportador = new ExportadorTexto();
        }
        exportador.exportar(lista);
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Si probamos la aplicación obtendremos el siguiente resultado:

> No existen datos para exportar

Vamos entonces a pasarle como argumento algunos datos al programa

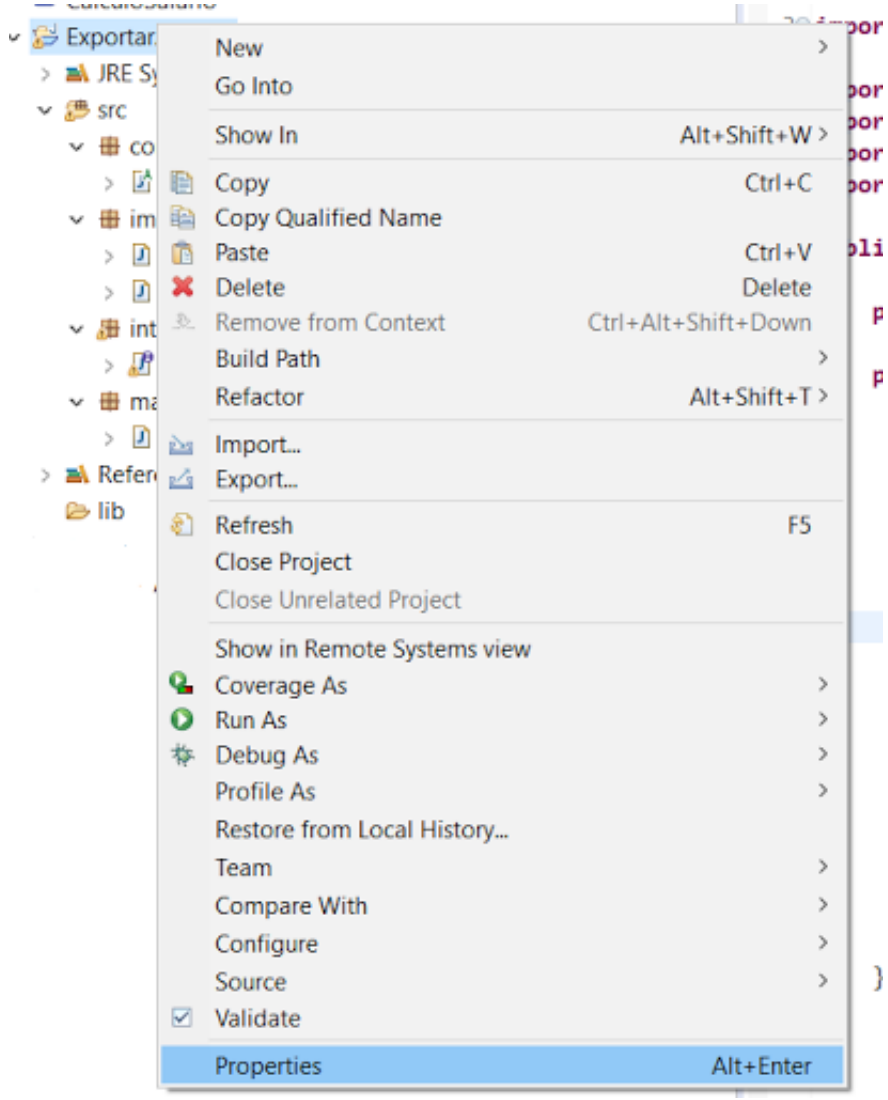

Imagen 17. Agregando argumentos al proyecto.

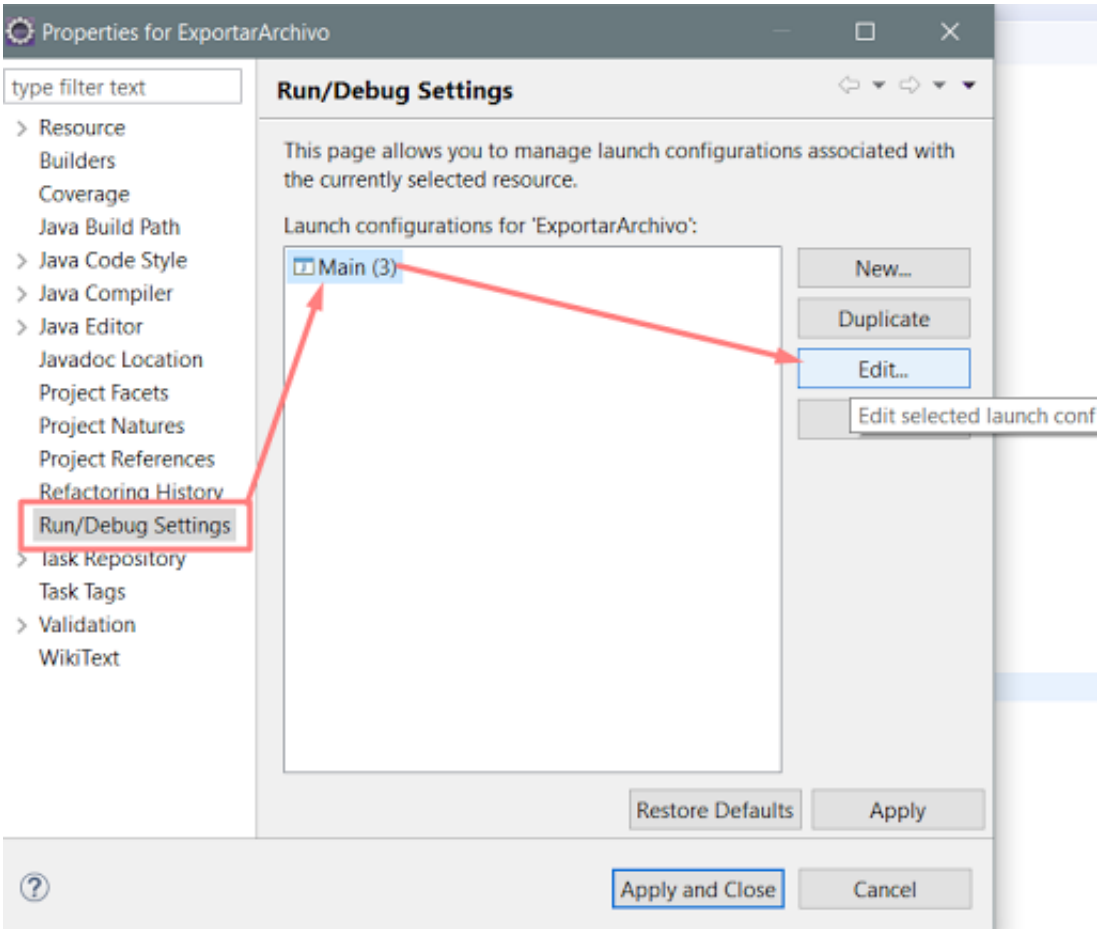

Imagen 18. Buscar Run/Debug Settings, seleccionar Main.

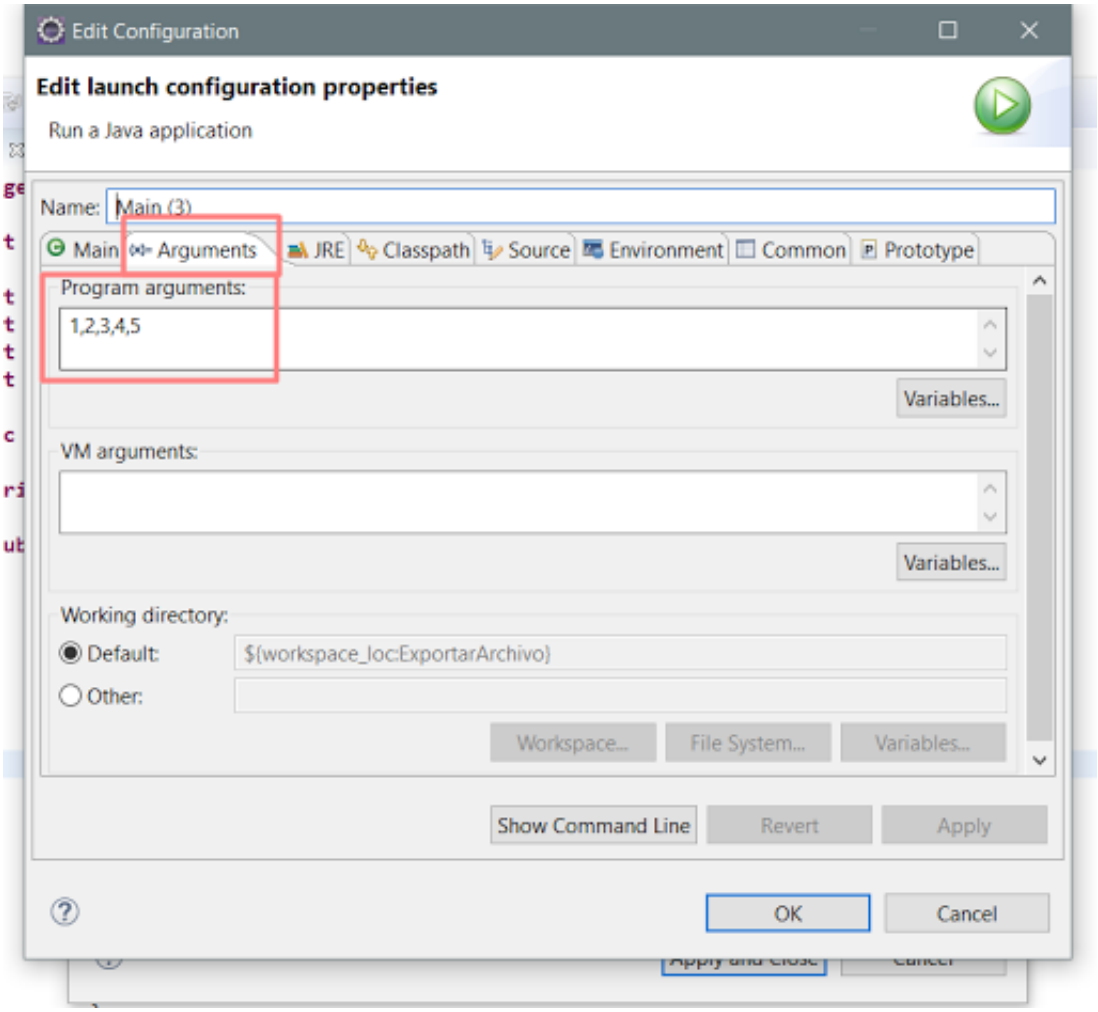

Imagen 19. Seleccionar Arguments.

Ahora ejecutamos y obtenemos el siguiente resultado:

```
java.nio.file.NoSuchFileException: configuracion.txt at
sun.nio.fs.WindowsException.translateToIOException(WindowsException.java:79) at
sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:97) at
sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:102) at
sun.nio.fs.WindowsFileSystemProvider.newByteChannel(WindowsFileSystemProvider.java:230) at
java.nio.file.Files.newByteChannel(Files.java:361) at
java.nio.file.Files.newByteChannel(Files.java:407) at
java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:384) at
java.nio.file.Files.newInputStream(Files.java:152) at
java.nio.file.Files.newBufferedReader(Files.java:2784) at
java.nio.file.Files.newBufferedReader(Files.java:2816) at
configuracion.LectorConfiguracion.buscarConfiguracion(LectorConfiguracion.java:19) at
configuracion.LectorConfiguracion.formatoSalida(LectorConfiguracion.java:11) at
main.Main.main(Main.java:23)
```

Esto es porque nos falta el archivo configuracion.txt, debemos crearlo en la carpeta raíz del proyecto:

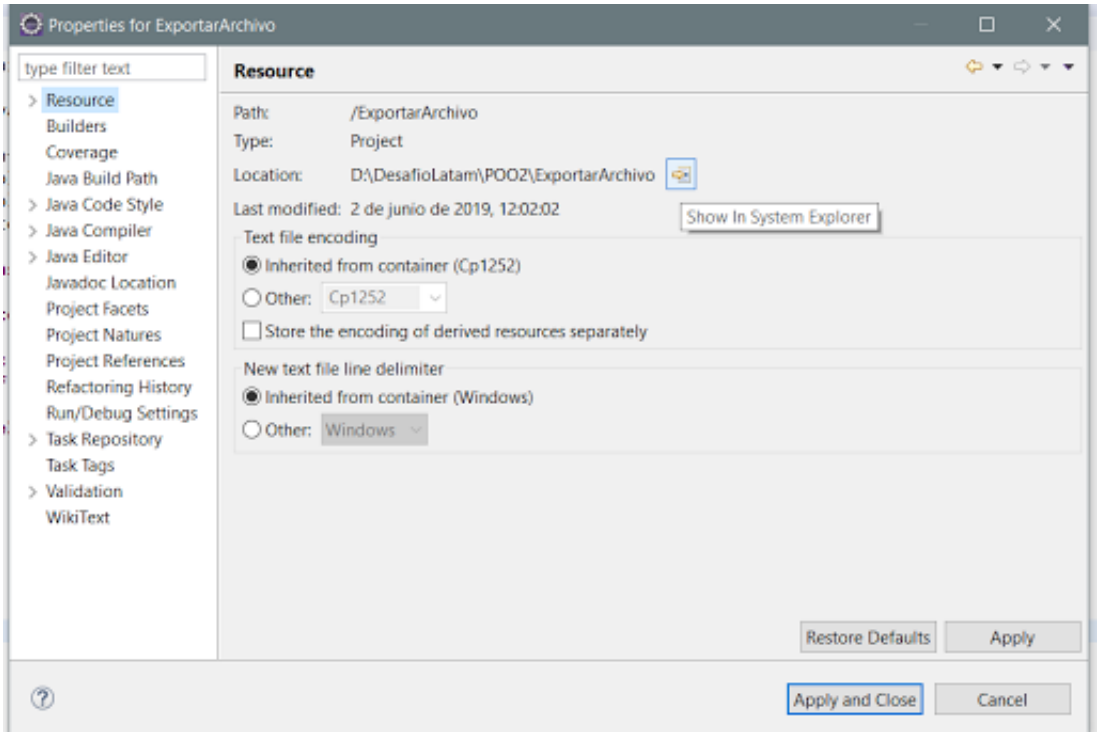

Imagen 20. Ir a Recursos.

Si no recuerdas qué carpeta es esa, simplemente abre las propiedades del proyecto y clickea el botón "Show in system explorer". Ahora, crearemos un nuevo archivo de texto llamado configuracion.txt y le vamos a agregar el siguiente contenido:

```
formatoSalida=txt;
rutaSalida=D:\DesafioLatam\POO2\ExportarArchivo;
```

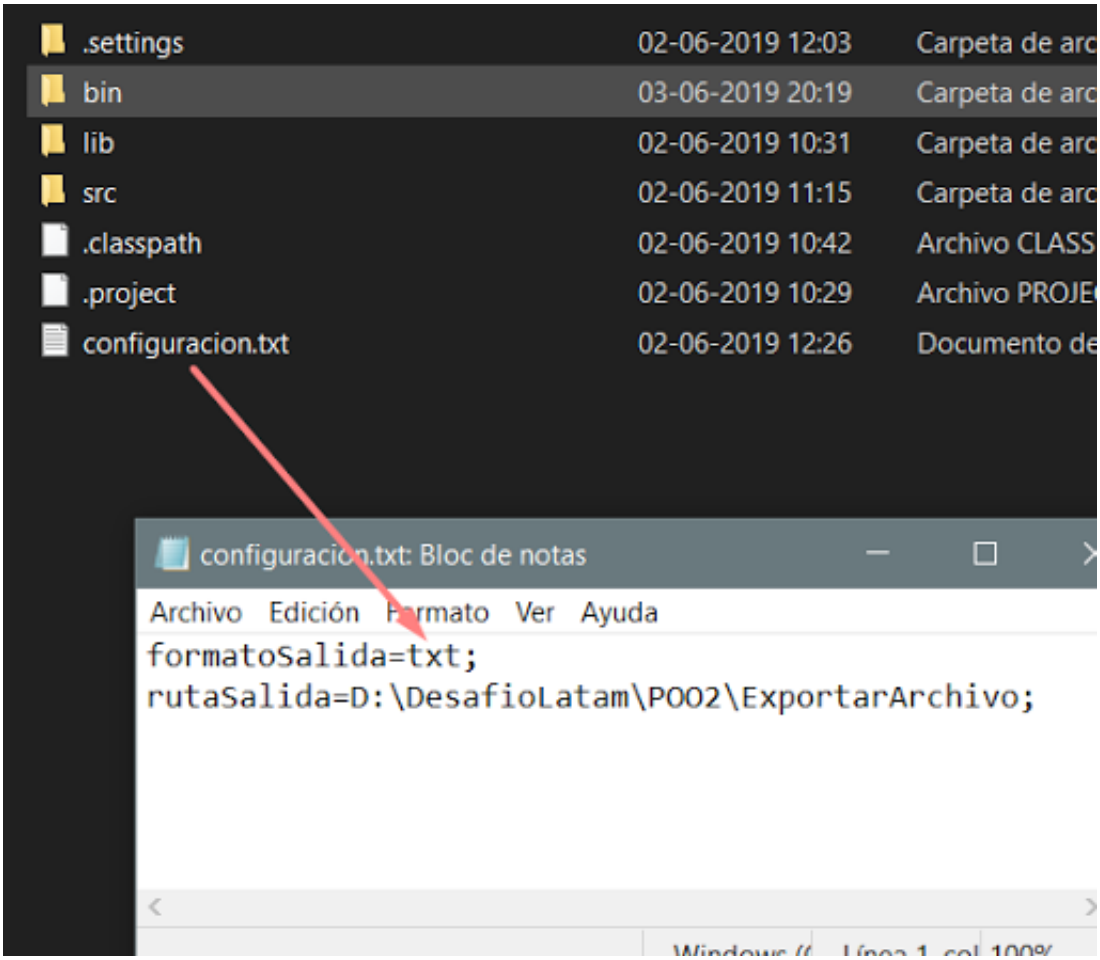

Imagen 21. Buscando la ruta.

La variable rutaSalida es la ruta de tu equipo donde se guardará el archivo creado, en este caso, lo crearemos en la carpeta raíz del proyecto.

Guardamos el archivo y ejecutamos nuevamente la aplicación.

Debería haberse creado el archivo en la ruta especificada:

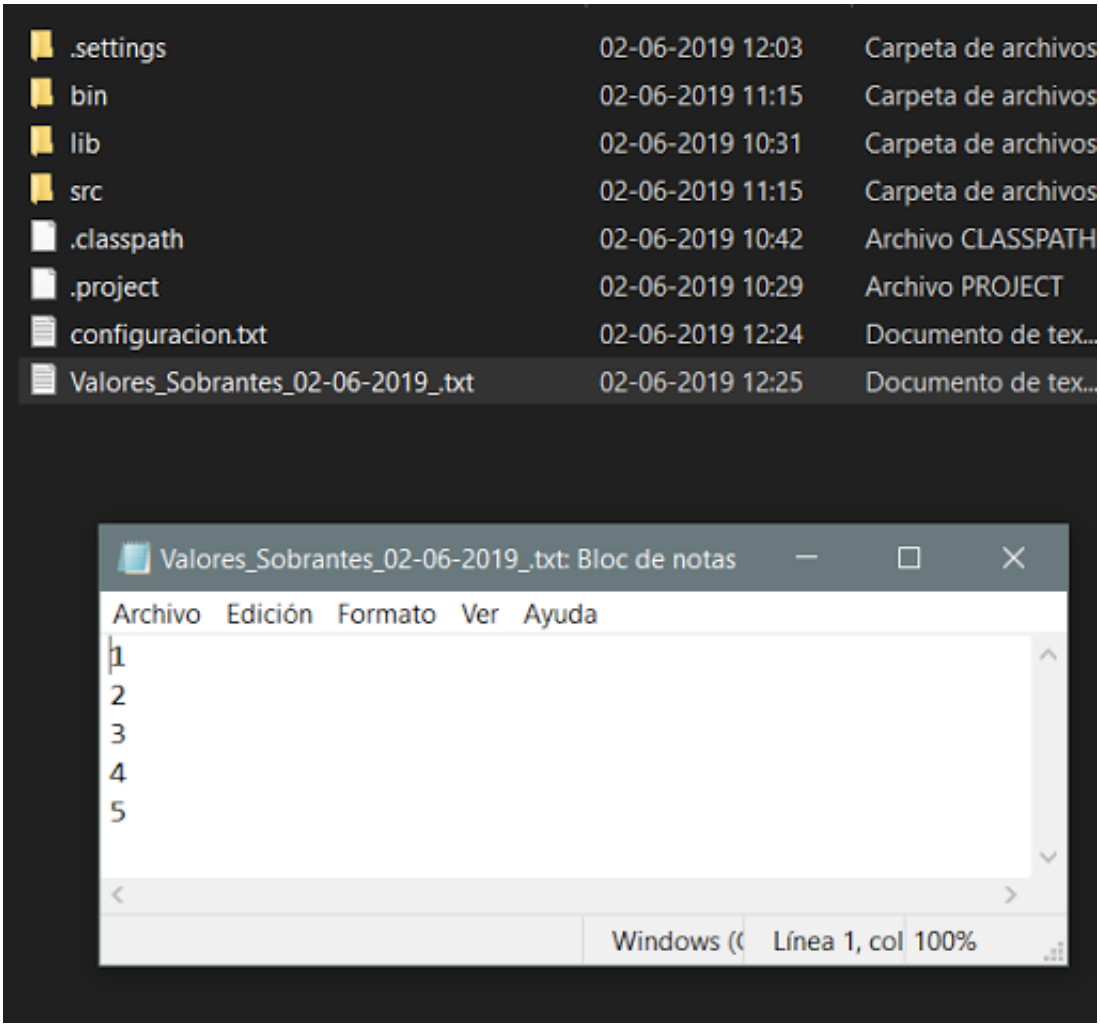

Imagen 22. Validando la creación del archivo .

Si ahora cambiamos el archivo de configuracion, indicando un formatoSalida=xls;

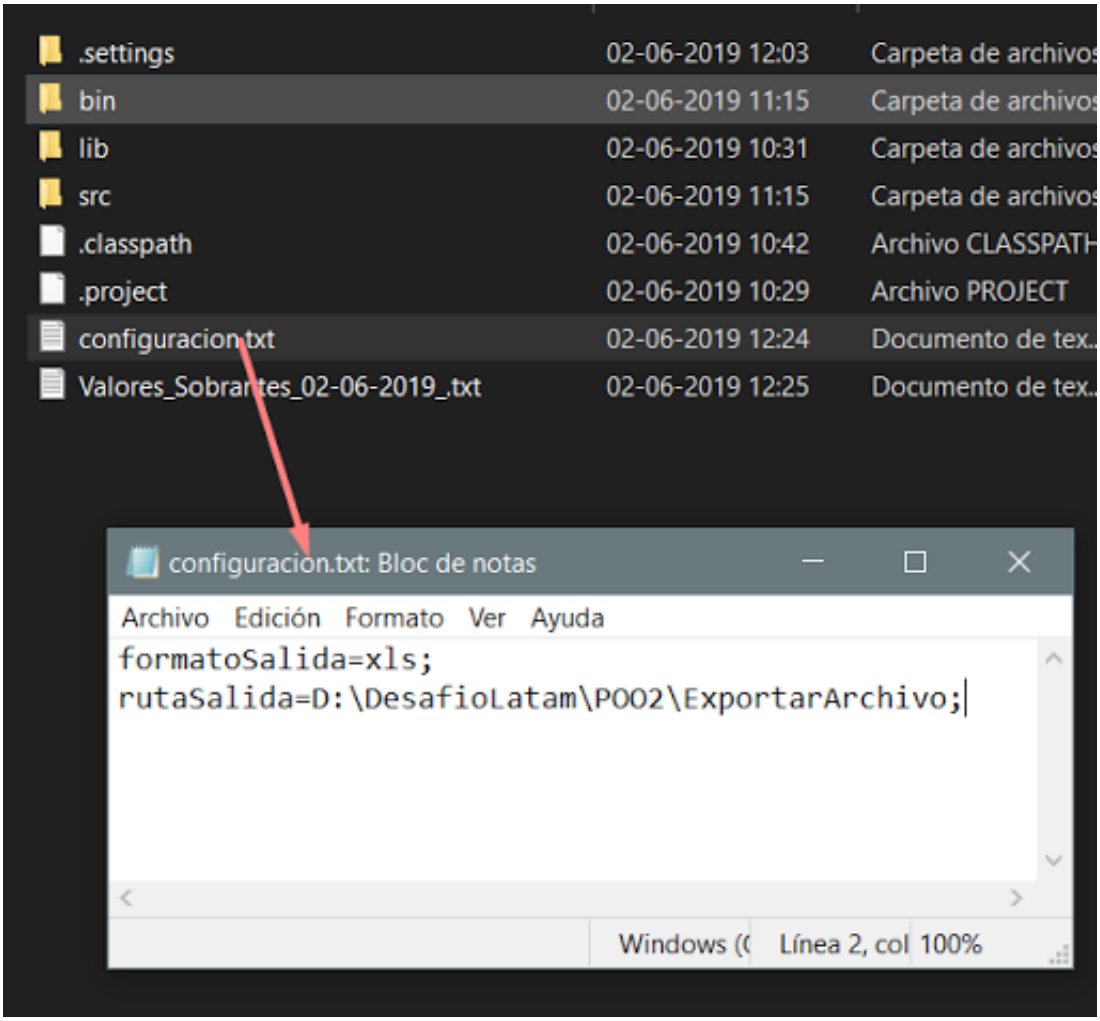

Imagen 23. Cambiando el formato de salida.

Obtenemos el siguiente resultado:

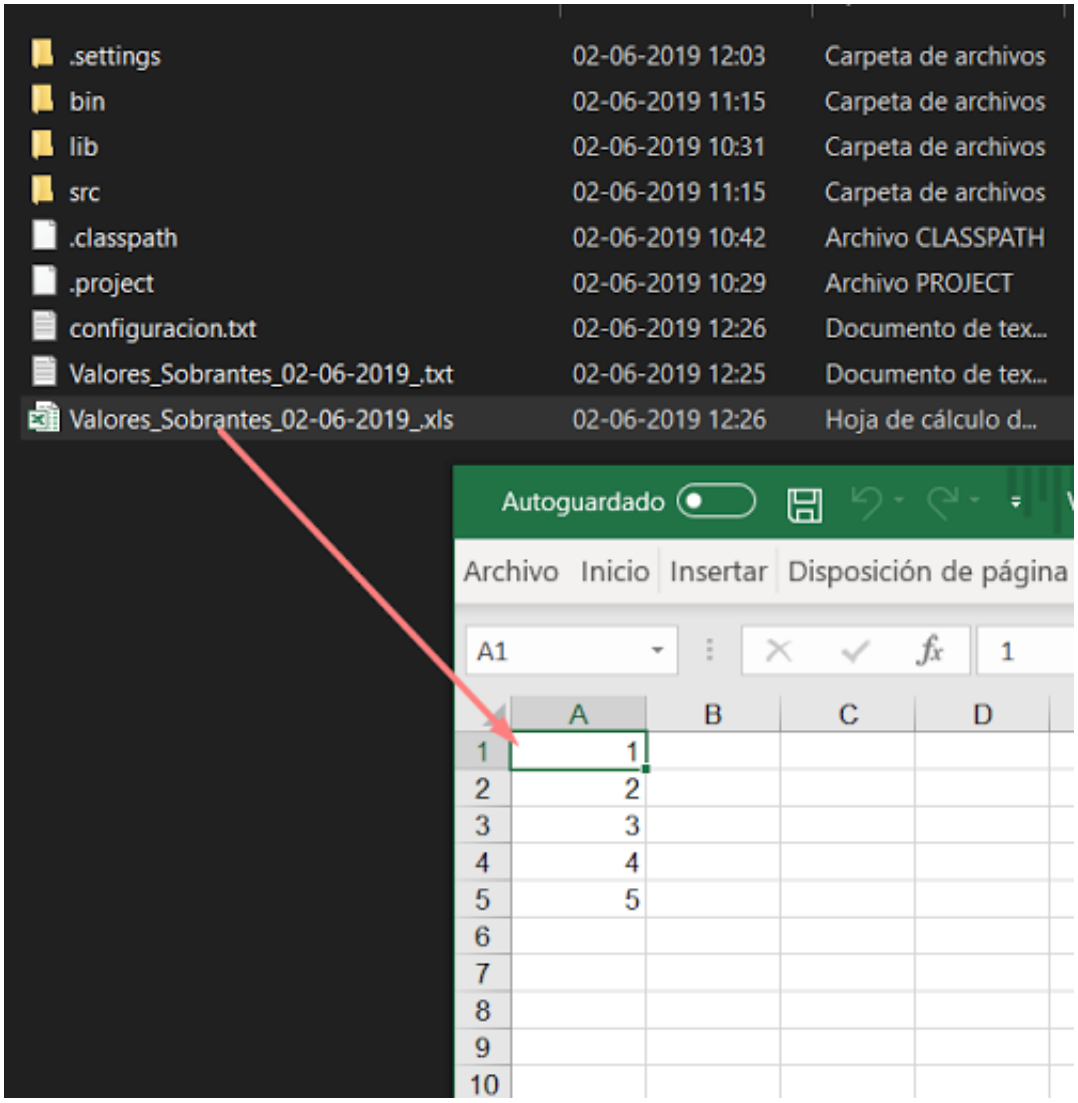

Imagen 24. Abriendo el archivo excel.

Y listo, hemos completado el módulo configurable de exportación a excel y txt.

Instancias únicas

Competencias

- Patrón de diseño Singleton

Introducción

Conoceremos el patrón Singleton de POO, un patrón bastante básico de la programación, que logra reducir el uso de memoria si es bien implementado en el software.

¿Qué es singleton?

Singleton, es un patrón de diseño de software muy fácil de comprender, se caracteriza porque los objetos del software que se rigen por el patrón, sólo se instanciarán 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, ya que reduce también la cantidad de variables. Su ventaja más importante es que si hay varias partes de la aplicación que comparten un mismo recurso, podrán acceder a él desde cualquier parte.

¿Cómo se aplica el patrón Singleton?

Para aplicarlo, se debe crear una clase que se instancie a si 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.

```
class Configurador{
  //Variable encapsulada y estática donde se almacenará la instancia.
  private static Configurador config;

  //Constructor privado
  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 podrían entrar a `if(config==null)` en el mismo instante y lograrían entrar al if, creando una instancia cada uno del Configurador.

En el software java, cada vez que se ejecuta un método o algoritmo, se crea un hilo, hasta que termina la ejecución de éste. Hay algunos algoritmos que tardan milésimas de segundo y otros que pueden llegar a durar meses ejecutándose, dependiendo 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.

Si esto no fuese asi, ocurriría algo como esto:

- El usuario 1 está exportando un archivo PDF desde una web, que demora unos 30 segundos en terminar de exportarse
- El usuario 2 está a punto de exportar el mismo PDF, pero esta vez, debe esperar 60 segundos para que termine de exportarse, los 30 del primer usuario y los 30 de él mismo.
- El usuario 3 está a apunto de exportar el mismo PDF, pero tendrá que esperar 90 segundos para que termine, que son los 60 segundos de los hilos de los otros dos usuarios y los 30 del hilo del usuario 3.

Imagen 25. Ejmplo para exportar pdf en un mismo archivo.

Pero por defecto el comportamiento de los hilos es paralelo al ejecutar un método.

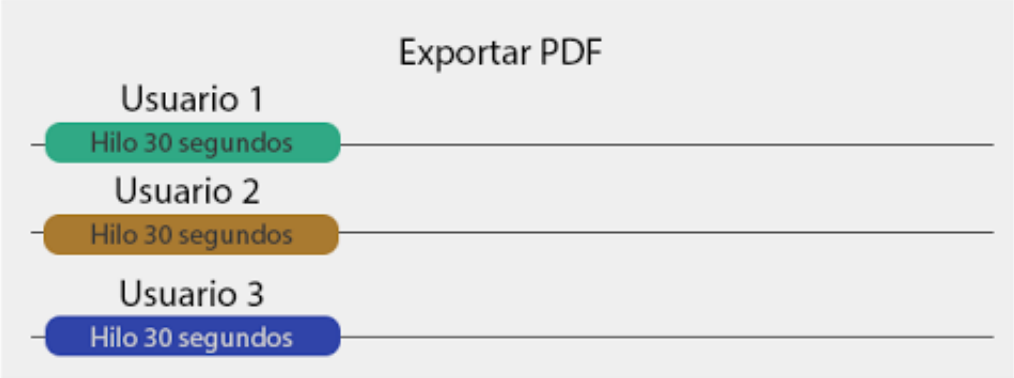

Imagen 26. Ejemplo hilos en paralelo.

Es por esta razón que cuando se tiene un Singleton, se debe cambiar este comportamiento por defecto del lenguaje, para prevenir que dos hilos en paralelo ejecuten el método que crea la instancia, para llevar a cabo esto, se debe sincronizar el método.

Synchronized

Cuando hablamos de un Singleton, necesitamos que se trabaje de una manera diferente a la que tienen por defecto los hilos, necesitamos que cada ejecución del método `getConfig()` del ejemplo anterior de la clase `Configurador` se ejecute de manera sincronizada, para que asi no se ejecuten hilos en paralelo y no se puedan crear dos instancias de la clase, lo que sería el patrón de Singleton,

Para esto, se utilizará la palabra reservada `synchronized`:

```
private 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
}
```

Como puedes ver, ahora, luego de comprobar la primera vez que no existe una instancia (`config == null`), se procede a crear una, pero en una porción de código que está sincronizada para todos los hilos, es decir, si hay dos o más usuarios tratando de acceder al método, el primero en hacerlo bloqueará el método para que los demás tengan que esperar a que termine de ejecutarlo. Permitiendo que, en la segunda validación de `config == null` si es que un segundo hilo accede, el primero ya haya creado la instancia y no se vuelva a crear otra.

Vamos a utilizarlo en un main para ver qué es lo que se muestra en el output:

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

> Instancia creada Llamada al Configurador Llamada al Configurador Llamada al Configurador