**Ejercicio 1.1:**

* Nombres poco descriptivos, son abreviaciones que a primera vista no se entiende para que sirve ese método.
* Los nombres no siguen la convención camelCase
* Permisos de acceso a los métodos inconsistentes, en los comentarios se indica que son datos que solicitan los clientes y los métodos son privados y protegidos

Mejora:

/\*\*

\* Retorna el límite de crédito del cliente.

\*/

public double getLimiteCredito() {

}

/\*\*

\* Calcula el monto facturado al cliente en un período de tiempo específico.

\*/

public double getMontoFacturado(LocalDate fechaInicio, LocalDate fechaFin) {

}

/\*\*

\* Calcula el monto cobrado al cliente en un período de tiempo específico

\*/

public double getMontoCobrado(LocalDate fechaInicio, LocalDate fechaFin) {

}

**Ejercicio 1.2:**

Realizo un “Move Method” de el método “participaEnProyecto” de la clase persona al la clase Proyecto.

Esto es necesario ya que habia dos malos (Envidia de atributos, Clase de datos) olores si el modelo no se cambiaba, al mover el método a la clase que es responsable de los personas en el proyecto se solucionan varios malos olores.

**Ejercicio 1.3:**

Lista de malos olores:

* El método “imprimirValores()” tiene multiples responsabilidades cuando podria dividirse en varios métodos mas pequeños para no tener un solo método mas largo
* Posible división por cero cuando se realiza el promedio de edades
* Uso de varias variables locales cuando muchos operaciones pueden solucionarse sin necesitarlas

Cambios:

public void imprimirValores() {

String message;

if (personal.isEmpty()) { //se puede evitar el uso de este if? O simplemente se puede ignorar?

message = ”No hay empleados registrados.";

}else{

Mensaje = String.format("El promedio de las edades es %.2f y el total de salarios es %.2f ", calcularPromedioEdades(), calcularTotalSalarios());

}

System.out.print(mensaje);

}

private double calcularPromedioEdades() {

return (double) personal.stream().mapToInt(personal -> personal.getEdad()).sum() / personal.size();

}

private double calcularTotalSalarios() {

return personal.stream().mapToDouble(personal -> personal.getSalario()).sum();

}

**Ejercicio 2.1:**

1. Atributos publicos, se cambian por atributos privados
2. Varias clases comparten tanto nombre, atributos y métodos, se necesita una clase abstracta para evitar repetir codigo
3. El método sueldo() se repite en todas las clases, se necesita un método abstracto ya que todas las clases lo utilizan de forma distinta, incluso se podría estandarizar una parte del codigo, ya que en todos los métodos sueldo() comparten la característica de que al sueldo basico se le resta el trece porciento

public class EmpleadoTemporario extends Empleado {

private String nombre;

private String apellido;

private double sueldoBasico = 0;

private double horasTrabajadas = 0;

private int cantidadHijos = 0;

// ......

@Override

protected double calcularAdicionales() {

return (horasTrabajadas \* 500) + (cantidadHijos \* 1000);

}

}

public class EmpleadoPlanta extends Empleado {

private String nombre;

private String apellido;

private double sueldoBasico = 0;

private int cantidadHijos = 0;

// ......

@Override

protected double calcularAdicionales() {

return (cantidadHijos \* 2000);

}

}

public class EmpleadoPasante extends Empleado {

private String nombre;

private String apellido;

private double sueldoBasico = 0;

// ......

}

public abstract class Empleado {

private String nombre;

private String apellido;

private double sueldoBasico;

// Método abstracto para calcular los adicionales de cada tipo de empleado

protected abstract double calcularAdicionales();

// Método final que estandariza el descuento

public final double sueldo() {

return this.sueldoBasico + calcularAdicionales() - (this.sueldoBasico \* 0.13);

}

}

**Ejercicio 2.2:**

1. El juego tiene Envidia de Atributos, incrementa y decrementa una variable del jugador, el método incrementar() y decrementar() tiene que moverse a la clase jugador
2. La clase jugador es una clase de datos, unicamente posee variables las cuales son utilizadas por la clase Juego, los métodos de la clase juego que utilizan variables de la clase Jugador deberían ser transferidos a la clase jugador
3. La clase jugador tiene todas sus variables publicas, probablemente porque la clase Juego estaba modificando sus valores con sus métodos, tiene que volverse privados para mantener el encapsulamiento

public class Juego {

// ......

public void incrementar(Jugador j) {

j.incrementar(100);

}

public void decrementar(Jugador j) {

j.decrementar(50);

}

}

public class Jugador {

private String nombre;

private String apellido;

private int puntuacion = 0;

public void incrementar(int valor){

this.puntuacion += valor;

}

public void decrementar(int valor){

this.puntuacion -= valor;

}

}

**Ejercicio 2.3:**

1. La clase PostApp tiene Envidia de Atributos, utiliza atributos de la clase Post en lugar de hacer una consulta
2. La clase PostApp tiene un método masivo en lugar de muchos métodos cortos para realizar distintas acciones, se necesita hacer Extract Method

public class PostApp{

private List<Post> posts;

/\*\*

\* Retorna los últimos N posts que no pertenecen al usuario user

\*/

public List<Post> ultimosPosts(Usuario user, int cantidad) {

List<Post> postsFiltrados = this.posts.stream

.filter(post -> !post.conoceUsuario(user)) //elimina a todos los posts que conocen al usuario

.sorted((post1, post2) -> post2.getFecha().copareTo(post1.getFecha())) //Ordena los posts por fecha en orden descendente (de más reciente a más antiguo). //Como post2 está primero, se invierte el orden normal (ascendente) y se obtiene orden descendente.

.limit(cantidad) //trunca la lista a la cantidad establecida

.collect(Collectors.toList()); //Convierte el Stream<Post> nuevamente en una List<Post>.

}

}

public class Post{

private String Texto;

private LocalDateTime fecha;

//getUsuario()

//getFecha()

public boolean conoceUsuario(Usuario u){

return this.getUsuario().equals(u);

}

}

public class Usuario{

private String userName;

}