OBJETOS 2 - PRACTICA REFACTORING

Ejercicio 1.1

Bad Smell: Nombre del método poco explicativo/descriptivo —> uso de muchos comentarios
Los métodos tienen nombres poco explicativos/descriptivos, lo que a su vez provoca uso de muchos comentarios para explicar lo que hacen.

Solución: Rename Methods
Se cambian los nombres de los métodos a nombres más descriptivos, lo que logra no necesitar comentarios para explicar lo que hacen.

Método lmtCrdt

Original:
// Retorna el límite de crédito del cliente
public double lmtCrdt() {…}

Solución:
public double getLimiteCreditoCliente() {…}

Método mtFcE

Original:
// Retorna el monto facturado al cliente desde la fecha f1 a la fecha f2
protected double mtFcE(LocalDate f1, LocalDate f2) {…}

Solución:
protected double getMontoFacturadoEntreFechas(LocalDate f1, LocalDate f2) {…}

Método mtCbE

Original:
// Retorna el monto cobrado al cliente desde la fecha f1 a la fecha f2
private double mtCbE(LocalDate f1, LocalDate f2) {…}

Solución:
private double getMontoCobradoEntreFechas(LocalDate f1, LocalDate f2) {…}

Bad Smell: Nombre de parámetros poco explicativo/descriptivo
Los parámetros de los métodos tienen nombres poco explicativos/ descriptivos.

Solución: Rename Parameters
Se cambian los nombres de los parámetros a nombres más descriptivos.

Método getMontoFacturadoEntreFechas

Original:
protected double getMontoFacturadoEntreFechas(LocalDate f1, LocalDate f2) {…}

Solución:
protected double getMontoFacturadoEntreFechas(LocalDate fechaInicio, LocalDate fechaFin) {…}

Método getMontoCobradoEntreFechas

Original:
private double getMontoCobradoEntreFechas(LocalDate f1, LocalDate f2) {…}

Solución:
private double getMontoCobradoEntreFechas(LocalDate fechaInicio, LocalDate fechaFin) {…

Ejercicio 1.2

Bad Smell: Feature Envy y mala asignación de responsabilidad
El método participaEnProyecto(..) no debería estar en la clase Persona, sino que debería estar en la clase Proyecto, ya que es esta última la encargada de evaluar la participación.

Solución: Move Method
Se mueve el método a la clase Proyecto.

Bad Smell: Rompe el encapsulamiento
En la clase Persona la variable id es pública, lo que viola el encapsulamiento.

Solución: Encapsulate Field
Se cambia la visibilidad de la variable id de pública a privada.
private String id;

Ejercicio 1.3

Código original

```
public void imprimirValores() {
     int totalEdades = 0;
     double promedioEdades = 0;
     double totalSalarios = 0;

     for (Empleado empleado : personal) {
          totalEdades = totalEdades + empleado.getEdad();
          totalSalarios = totalSalarios +
empleado.getSalario();
     }
     promedioEdades = totalEdades / personal.size();

     String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);

     System.out.println(message);
}
```

Bad Smell: Nombre del método poco explicativo/descriptivo.
El nombre del método imprimir valores es poco explicativo/descriptivo.
Solución: Rename Method
Se cambia el nombre del método de imprimirValores() a imprimirPromedioEdadYSalarios().

```
public void imprimirPromedioEdadYSalarios() { ... }
```

Bad Smell: Reinventa La Rueda
El bucle utilizado para calcular el promedio de la edad de los empleados no óptimo. Puede resolverse de manera más eficiente.
Solución: Replace Loop with Pipeline
Se reemplaza la linea dentro del for que calculaba el promedio de la edad de los empleados por un stream fuera del for (se asigna el resultado a la variable promedioEdades).

```
public void imprimirPromedioEdadYSalarios() {
     int totalEdades = 0;
     double promedioEdades =
this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
     double totalSalarios = 0;

     for (Empleado empleado : personal) {
```

```
            totalSalarios = totalSalarios +
empleado.getSalario();
        }
        promedioEdades = totalEdades / personal.size();

        String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);

        System.out.println(message);
}
```

Bad Smell: Reinventa La Rueda
El bucle utilizado para calcular el total de los salarios tampoco es óptimo ni eficiente.
Solución: Replace Loop with Pipeline
Se reemplaza la linea dentro del for que calculaba el total de los salarios de los empleados por un stream fuera del for (se asigna el resultado a la variable totalSalarios). Consecuentemente se elimina el loop for.

```
public void imprimirPromedioEdadYSalarios() {
        int totalEdades = 0;
        double promedioEdades =
this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
        double totalSalarios =
this.personal.stream().mapToDouble(e ->
e.getSalario()).sum();

        promedioEdades = totalEdades / personal.size();

        String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);

        System.out.println(message);
}
```

Bad Smell: Temporary Field
Las variables totalEdades y promedioEdades (la segunda vez) se calculan de forma innecesaria.
Solución: Eliminar variables redundantes (Replace temp with query)
Se eliminan las variables totalEdades y promedioEdades (la segunda vez).

```
public void imprimirPromedioEdadYSalarios() {
```

```
    double promedioEdades =
this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
    double totalSalarios =
this.personal.stream().mapToDouble(e ->
e.getSalario()).sum();

    String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);

    System.out.println(message);
}
```

Bad Smell: **Long Method**
El método imprimirPromedioEdadYSalarios hace muchas cosas (muchas responsabilidades).
Solución: **Extract Method para el cálculo de edades**

```
public void imprimirPromedioEdadYSalarios() {
    double promedioEdades = this.calcularPromedioEdades();
    double totalSalarios =
this.personal.stream().mapToDouble(e ->
e.getSalario()).sum();

    String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);

    System.out.println(message);
}

public double calcularPromedioEdades() {
    return this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
}
```

Solución: **Extract Method para el cálculo de salarios**

```
public void imprimirPromedioEdadYSalarios() {
    double promedioEdades = this.calcularPromedioEdades();
    double totalSalarios = this.calcularTotalSalarios();

    String message = String.format("El promedio de las
edades es %s y el total de salarios es %s", promedioEdades,
totalSalarios);
```

```
        System.out.println(message);
}

public double calcularPromedioEdades() {
        return this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
}

public double calcularTotalSalarios() {
        return this.personal.stream().mapToDouble(e ->
e.getSalario()).sum();
}
```

Bad Smell: Long Method y Temporary Field
Método sigue siendo largo con uso de variables temporales innecesarias.
Solución: Eliminar variables redundantes (Replace temp with query)
Se simplifica el método eliminando las variables temporales.

```
public void imprimirPromedioEdadYSalarios() {
        String message = String.format("El promedio de las
edades es %s y el total de salarios es %s",
this.calcularPromedioEdades(),
this.calcularTotalSalarios());

        System.out.println(message);
}

public double calcularPromedioEdades() {
        return this.personal.stream().mapToDouble(e ->
e.getEdad()).average().orElse(0);
}

public double calcularTotalSalarios() {
        return this.personal.stream().mapToDouble(e ->
e.getSalario()).sum();
}
```

Ejercicio 2.1

Código original

```
public class EmpleadoTemporario {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    public double horasTrabajadas = 0;
    public int cantidadHijos = 0;
    // ......
```

```
public double sueldo() {
return this.sueldoBasico
       +(this.horasTrabajadas * 500)
       +(this.cantidadHijos * 1000)
       -(this.sueldoBasico * 0.13);
}
}

public class EmpleadoPlanta {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    public int cantidadHijos = 0;
    // ......

    public double sueldo() {
        return this.sueldoBasico
            + (this.cantidadHijos * 2000)
            - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    // ......

    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }
}
```

Bad Smell: Duplicated Code
Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante cuentan con campos y métodos comunes.
Solución: Extract Superclass
Se crea SuperClase Empleado y las clases existentes pasan a ser subclases de la misma.

```
public abstract class Empleado { … }

public class EmpleadoTemporario extends Empleado {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
```

```
    public double horasTrabajadas = 0;
    public int cantidadHijos = 0;
    // ......

public double sueldo() {
return this.sueldoBasico
       +(this.horasTrabajadas * 500)
       +(this.cantidadHijos * 1000)
       -(this.sueldoBasico * 0.13);
}
}

public class EmpleadoPlanta extends Empleado {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    public int cantidadHijos = 0;
    // ......

    public double sueldo() {
        return this.sueldoBasico
            + (this.cantidadHijos * 2000)
            - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante extends Empleado {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    // ......

    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }
}
```

Bad Smell: Duplicated Code
Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante siguen contando con los mismos campos.
Solución: Pull up field
Se extraen campos comunes de las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante a la SuperClase Empleado.

```
public abstract class Empleado {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;}

public class EmpleadoTemporario extends Empleado {
    public double horasTrabajadas = 0;
    public int cantidadHijos = 0;
    // ......

public double sueldo() {
return this.sueldoBasico
       +(this.horasTrabajadas * 500)
       +(this.cantidadHijos * 1000)
       -(this.sueldoBasico * 0.13);
}
}

public class EmpleadoPlanta extends Empleado {
    public int cantidadHijos = 0;
    // ......

    public double sueldo() {
        return this.sueldoBasico
            + (this.cantidadHijos * 2000)
            - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante extends Empleado {
    // ......

    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }
}
```

Bad Smell: **Rompe el encapsulamiento**
Las clases Empleado, EmpleadoTemporario, EmpleadoPlanta tienen variables de instancia públicas, lo que viola el encapsulamiento.
Solución: **Encapsulate field**
Se cambia visibilidad de variables de instancia a privadas y se agregan setters.

necesito getters también? si no son necesarios no los agrego

```
public abstract class Empleado {
    private String nombre;
    private String apellido;
    private double sueldoBasico = 0;

    public void setNombre(String nombre) {
     this.nombre = nombre;
    }

    public void setApellido(String apellido) {
     this.apellido = apellido;
    }

    public void setSueldoBasico(double sueldoBasico) {
     this.sueldoBasico = sueldoBasico;
    }

}

public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;

    public void setHorasTrabajadas(double horasTrabajadas) {
     this.horasTrabajadas = horasTrabajadas;
    }

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
    }

    // ......

public double sueldo() {
return this.sueldoBasico
       +(this.horasTrabajadas * 500)
       +(this.cantidadHijos * 1000)
       -(this.sueldoBasico * 0.13);
}
}

public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos = 0;

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
    }
```

```
    // ......

    public double sueldo() {
        return this.sueldoBasico
            + (this.cantidadHijos * 2000)
            - (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante extends Empleado {
    // ......

    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }
}
```

Bad Smell: **Long Method**
Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante tienen métodos largos (método sueldo).
Solución: **Extract Method**
Se extraen responsabilidades de los métodos a otros métodos más simples.

```
public abstract class Empleado {
    private String nombre;
    private String apellido;
    private double sueldoBasico = 0;

    public void setNombre(String nombre) {
     this.nombre = nombre;
    }

    public void setApellido(String apellido) {
     this.apellido = apellido;
    }

    public void setSueldoBasico(double sueldoBasico) {
     this.sueldoBasico = sueldoBasico;
    }

}

public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;
```

```
    public void setHorasTrabajadas(double horasTrabajadas) {
     this.horasTrabajadas = horasTrabajadas;
    }

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
    }

    // ......

     public double sueldo() {
        return this.sueldoConDescuento() +
        this.calculoHorasTrabajadas() +
        this.asignacionFamiliar();
}

     public double sueldoConDescuento() {
         return this.sueldoBasico - (this.sueldoBasico *
     0.13);
    }

     public double asignacionFamiliar() {
         return this.cantidadHijos * 1000);
    }

     public double calculoHorasTrabajadas() {
         return this.horasTrabajadas * 500);
    }
}

public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos = 0;

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
    }

    // ......

    public double sueldo() {
         return this.sueldoConDescuento() +
                this.asignacionFamiliar();
    }

    public double asignacionFamiliar() {
         return this.cantidadHijos * 2000);
    }
```

```
      public double sueldoConDescuento() {
         return this.sueldoBasico - (this.sueldoBasico *
      0.13);
      }
}

public class EmpleadoPasante extends Empleado {
     // ......

      public double sueldo() {
         return this.sueldoConDescuento();
      }

     public double sueldoConDescuento() {
         return this.sueldoBasico - (this.sueldoBasico *
0.13);
     }
}
```

Bad Smell: **Duplicated Code**
Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante tienen métodos comunes.
Solución: **Pull Up Method**
Se extraen métodos comunes a SuperClase Empleado.

```
public abstract class Empleado {
     private String nombre;
     private String apellido;
     private double sueldoBasico = 0;

     public void setNombre(String nombre) {
      this.nombre = nombre;
     }

     public void setApellido(String apellido) {
      this.apellido = apellido;
     }

     public void setSueldoBasico(double sueldoBasico) {
      this.sueldoBasico = sueldoBasico;
     }

      public double sueldo() {
         return this.sueldoConDescuento();
      }
```

```
    public double sueldoConDescuento() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }

}

public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;

    public void setHorasTrabajadas(double horasTrabajadas) {
     this.horasTrabajadas = horasTrabajadas;
    }

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
    }

    // ......

     public double sueldo() {
       return super.sueldo() + this.calculoHorasTrabajadas()
       + this.asignacionFamiliar();
}

     public double asignacionFamiliar() {
         return this.cantidadHijos * 1000);
     }

     public double calculoHorasTrabajadas() {
         return this.horasTrabajadas * 500);
     }
}

public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos = 0;

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
     }

     // ......

     public double sueldo() {
         return super.sueldo() + this.asignacionFamiliar();
     }
```

```
    public double asignacionFamiliar() {
        return this.cantidadHijos * 2000);
    }
}

public class EmpleadoPasante extends Empleado {
    // ……
}
```

Bad Smell: **Duplicated Code**
Las clases EmpleadoPlanta y EmpleadoTemporario comparten variable de instancia cantidadHijos y método setCantidadHijos, sin embargo considero que no vale la pena generar otra SuperClase solamente por una sola variable y método.
Posible Solución: **Extract SuperClass, Pull Up Field, Pull Up Method**
Se podría crear SuperClase EmpleadoConHijos que sea subclase de Empleado, y superClase de EmpleadoPlanta y EmpeladoTeporario. Se llevaría la variable de instancia cantidadHijos y un método abstracto asignacionFamiliar() que heredarían y tendrían que redefinir las clases EmpeladoPlanta y EmpleadoTemporario.

Bad Smell: **Lazy Class**
La clase EmpeladoPasante es pequeña, pero decide mantenerse por un tema de expresividad y diseño.

Bad Smell: **Nombre de método poco explicativo**
El método sueldo() no tiene un nombre muy explicativo.
Solución: **Rename Method**
Se renombra el método sueldo() a calcularSueldo().

```
public abstract class Empleado {
    private String nombre;
    private String apellido;
    private double sueldoBasico = 0;

    public void setNombre(String nombre) {
     this.nombre = nombre;
    }

    public void setApellido(String apellido) {
```

```
        this.apellido = apellido;
    }

    public void setSueldoBasico(double sueldoBasico) {
        this.sueldoBasico = sueldoBasico;
    }

    public double calcularSueldo() {
        return this.sueldoConDescuento();
    }

    public double sueldoConDescuento() {
        return this.sueldoBasico - (this.sueldoBasico *
0.13);
    }

}

public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;

    public void setHorasTrabajadas(double horasTrabajadas) {
        this.horasTrabajadas = horasTrabajadas;
    }

    public void setCantidadHijos(int cantidadHijos) {
        this.cantidadHijos = cantidadHijos;
    }

    // ......

    public double calcularSueldo() {
       return super.calcularSueldo() +
       this.calculoHorasTrabajadas() +
       this.asignacionFamiliar();
}

    public double asignacionFamiliar() {
        return this.cantidadHijos * 1000);
    }

    public double calculoHorasTrabajadas() {
        return this.horasTrabajadas * 500);
    }
}

public class EmpleadoPlanta extends Empleado {
```

```
    private int cantidadHijos = 0;

     public void setCantidadHijos(int cantidadHijos) {
     this.cantidadHijos = cantidadHijos;
     }

     // ......

     public double calcularSueldo() {
          return super.calcularSueldo() +
this.asignacionFamiliar();
     }

     public double asignacionFamiliar() {
          return this.cantidadHijos * 2000);
     }
}

public class EmpleadoPasante extends Empleado {
     // ……
}
```

Ejercicio 2.2

```
public class Juego {
     // ......
     public void incrementar(Jugador j) {
          j.puntuacion = j.puntuacion + 100;}
     public void decrementar(Jugador j) {
          j.puntuacion = j.puntuacion - 50;}
}

public class Jugador {
     public String nombre;
     public String apellido;
     public int puntuacion = 0;}
```

PREGUNTAR: AL HACER PRIVADAS LAS VARIABLES DE INSTANCIA DEBO CREAR GETTERS Y SETTERS? SOLO GETTERS? Y UN CONSTRUCTOR? SOLO SI SON NECESARIAS EN ESE CONTEXTO. (podria estar rompiendo el encapsulamiento)

Bad Smell: Rompe el encapsulamiento
La clase Jugador tiene variables de instancia públicas, lo que viola el encapsulamiento.
Solución: Encapsulate field

Se cambia la visibilidad de las variables de instancia de Jugador a privadas. Se crean también los métodos getters y setters de cada una.

```
public class Juego {
    // ......
    public void incrementar(Jugador j) {
        j.puntuacion = j.puntuacion + 100;}
    public void decrementar(Jugador j) {
        j.puntuacion = j.puntuacion - 50;}
}

public class Jugador {
    private String nombre;
    private String apellido;
    private int puntuacion = 0;

     public void setNombre(String nombre) {
        this.nombre = nombre;}
     public void setApellido(String apellido) {
        this.apellido = apellido;}
     public void setPuntuacion(int puntuacion) {
        this.puntuacion = puntuacion;}

     public String getNombre() {
        return this.nombre;}
     public String getApellido() {
        return this.apellido;}
     public String getPuntuacion() {
        return this.puntuacion;}
}
```

Bad Smell: Nombres de métodos poco explicativos
Los métodos incrementar y decrementar en la clase Juego son poco explicativos.
Solución: Rename Method
Se renombran los métodos incrementar y decrementar a incrementarPuntuacion y decrementarPuntuacion.

```
public class Juego {
    // ......
    public void incrementarPuntuacion(Jugador j) {
        j.puntuacion = j.puntuacion + 100;}
    public void decrementarPuntuacion(Jugador j) {
        j.puntuacion = j.puntuacion - 50;}
}

public class Jugador {
    private String nombre;
```

```
    private String apellido;
    private int puntuacion = 0;

     public void setNombre(String nombre) {
        this.nombre = nombre;}
     public void setApellido(String apellido) {
        this.apellido = apellido;}
     public void setPuntuacion(int puntuacion) {
        this.puntuacion = puntuacion;}

     public String getNombre() {
        return this.nombre;}
     public String getApellido() {
        return this.apellido;}
     public String getPuntuacion() {
        return this.puntuacion;}
}
```

Bad Smell: Feature Envy

Los métodos incrementarPuntuacion y decrementarPuntuacion en la clase Juego acceden a variables de la clase Jugador haciendo una mala asignación de responsabilidades.

Solución: Move Method

Se mueven los métodos incrementarPuntuacion y decrementarPuntuacion a la clase Jugador. Los métodos ya existentes en la clase Juego se modifican para que llamen a los métodos correspondientes en la clase Jugador (los cuales hacen uso de variables de instancia de su propia clase).

```
public class Juego {
    // ......
    public void incrementarPuntuacion(Jugador j) {
        j.incrementarPuntuacion;}
    public void decrementarPuntuacion(Jugador j) {
        j.decrementarPuntuacion;}
}

public class Jugador {
    private String nombre;
    private String apellido;
    private int puntuacion = 0;

     public void setNombre(String nombre) {
        this.nombre = nombre;}
     public void setApellido(String apellido) {
        this.apellido = apellido;}
     public void setPuntuacion(int puntuacion) {
        this.puntuacion = puntuacion;}
```

```
    public String getNombre() {
        return this.nombre;}
    public String getApellido() {
        return this.apellido;}
    public String getPuntuacion() {
        return this.puntuacion;}

    public void incrementarPuntuacion() {
        this.puntuacion = this.puntuacion + 100;}
    public void decrementarPuntuacion() {
        this.puntuacion = this.puntuacion - 50;}
}
```

PREGUNTAR: LA FALTA DE CONSTRUCTOR SERIA UN BAD SMELL? HABRIA QUE AGREGAR UNO? no lo agrego salvo q en ese contexto sea estrictamente necesario

Ejercicio 2.3


```
/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {
        if (!post.getUsuario().equals(user)) {
            postsOtrosUsuarios.add(post);
        }
    }

   // ordena los posts por fecha
   for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
       int masNuevo = i;
       for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
           if (postsOtrosUsuarios.get(j).getFecha().isAfter(
     postsOtrosUsuarios.get(masNuevo).getFecha())) {
                masNuevo = j;
```

```
            }
        }
        Post unPost =
postsOtrosUsuarios.set(i ,postsOtrosUsuarios.get(masNuevo));
        postsOtrosUsuarios.set(masNuevo, unPost);
    }

    List<Post> ultimosPosts = new ArrayList<Post>();
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }
    return ultimosPosts;
}
```

Bad Smell: Long Method
El método ultimosPosts hace demasiadas cosas.
Solución: Extract Method
Se extraen responsabilidades del método ultimosPosts en diferentes métodos: getPostsOtrosUsuarios, ordenarPostsPorFecha, filtarPrimerosNPosts.

```
// Retorna los últimos N posts que no pertenecen al usuario user
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    getPostsOtrosUsuarios(user, postOtrosUsuarios);

    ordenarPostsPorFecha(postsOtrosUsuarios);

    List<Post> ultimosPosts = new ArrayList<Post>();
    filtrarPrimerosNPosts(cantidad, postsOtrosUsuarios,
     ultimosPosts);

    return ultimosPosts;
}

private void getPostsOtrosUsuarios(Usuario user, List<Post>
postsOtrosUsuarios){
     for (Post post : this.posts) {
        if (!post.getUsuario().equals(user)) {
            postsOtrosUsuarios.add(post);}
     }
}

private void ordenarPostsPorFecha(List<Post> postsOtrosUsuarios){
      for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
       int masNuevo = i;
       for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
           if (postsOtrosUsuarios.get(j).getFecha().isAfter(
```

```
      postsOtrosUsuarios.get(masNuevo).getFecha())) {
                masNuevo = j;
            }
        }
       Post unPost =
postsOtrosUsuarios.set(i ,postsOtrosUsuarios.get(masNuevo));
       postsOtrosUsuarios.set(masNuevo, unPost);
    }
}

private void filtrarPrimerosNPosts(int cantidad, List<Post>
postsOtrosUsuarios, List<Post> ultimosPosts){
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }
}
```

Bad Smell: Long Method y Temporary Field
El método ultimosPosts puede simplificarse más eliminando variables temporales innecesarias.
Solución: Replace temp with query
Se eliminan variables temporales innecesarias.

```
// Retorna los últimos N posts que no pertenecen al usuario user
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = getPostsOtrosUsuarios(user);

    ordenarPostsPorFecha(postsOtrosUsuarios);

    return filtrarPrimerosNPosts(cantidad, postsOtrosUsuarios);
}

private List<Post> getPostsOtrosUsuarios(Usuario user){
     List<Post> postsOtrosUsuarios = new ArrayList<Post>();
     for (Post post : this.posts) {
        if (!post.getUsuario().equals(user)) {
            postsOtrosUsuarios.add(post);}
     }
     return postsOtrosUsuarios;
}

private void ordenarPostsPorFecha(List<Post> postsOtrosUsuarios){
      for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
       int masNuevo = i;
       for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
           if (postsOtrosUsuarios.get(j).getFecha().isAfter(
     postsOtrosUsuarios.get(masNuevo).getFecha())) {
                masNuevo = j;
```

```
            }
        }
        Post unPost =
postsOtrosUsuarios.set(i ,postsOtrosUsuarios.get(masNuevo));
        postsOtrosUsuarios.set(masNuevo, unPost);
    }
}

private List<Post> filtrarPrimerosNPosts(int cantidad, List<Post>
postsOtrosUsuarios){
    List<Post> ultimosPosts = new ArrayList<Post>();
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }

    return ultimosPosts;
}
```

Bad Smell: Reinventa La Rueda
Los métodos getPostsOtrosUsuarios, ordenarPostsPorFecha, filtarPrimerosNPosts reinventan la rueda en sus lazos.
Solución: Replace Loop with Pipeline
Se modifican los métodos con soluciones más modernas y eficientes usando streams.

```
// Retorna los últimos N posts que no pertenecen al usuario user
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = getPostsOtrosUsuarios(user);

    postsOtrosUsuarios = ordenarPostsPorFecha(postsOtrosUsuarios);

    return filtrarPrimerosNPosts(cantidad, postsOtrosUsuarios);
}

private List<Post> getPostsOtrosUsuarios(Usuario user){
     return posts.stream().filter(post -> !
post.getUsuario().equals(user)).collect(Collectors.toList());
}

private List<Post> ordenarPostsPorFecha(List<Post>
postsOtrosUsuarios){
      return postsOtrosUsuarios.stream().sorted((post1, post2) ->
post1.getFecha().compareTo(post2.getFecha())).collect(Collectors.t
oList());
}
```

```
private List<Post> filtrarPrimerosNPosts(int cantidad, List<Post>
postsOtrosUsuarios){
     return postsOtrosUsuarios.stream()
     .limit(cantidad).collect(Collectors.toList());
}
```

Bad Smell: Long Method y Temporary Field
El método ultimosPosts puede simplificarse más eliminando variables temporales innecesarias.
Solución: Replace temp with query
Se eliminan variables temporales innecesarias.

```
// Retorna los últimos N posts que no pertenecen al usuario user
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    return filtrarPrimerosNPosts(cantidad,
ordenarPostsPorFecha(getPostsOtrosUsuarios(user)));
}

private List<Post> getPostsOtrosUsuarios(Usuario user){
     return posts.stream().filter(post -> !
post.getUsuario().equals(user)).collect(Collectors.toList());
}

private List<Post> ordenarPostsPorFecha(List<Post>
postsOtrosUsuarios){
      return postsOtrosUsuarios.stream().sorted((post1, post2) ->
post1.getFecha().compareTo(post2.getFecha())).collect(Collectors.t
oList());
}

private List<Post> filtrarPrimerosNPosts(int cantidad, List<Post>
postsOtrosUsuarios){
     return postsOtrosUsuarios.stream()
     .limit(cantidad).collect(Collectors.toList());
}
```

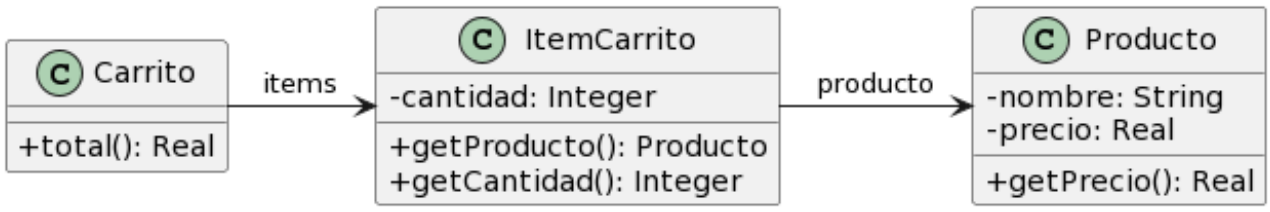
Bad Smell: Nombre poco explicativo
El método ultimosPosts tiene un nombre poco explicativo.
Solución: Rename Method
Se cambia el nombre del método ultimosPosts a obtenerUltimosNPostsMenosUsuario. Consecuentemente se puede eliminar el comentario que indica qué hace ese método.

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

    return filtrarPrimerosNPosts(cantidad,
ordenarPostsPorFecha(getPostsOtrosUsuarios(user)));
```

```
}

private List<Post> getPostsOtrosUsuarios(Usuario user){
     return posts.stream().filter(post -> !
post.getUsuario().equals(user)).collect(Collectors.toList());
}

private List<Post> ordenarPostsPorFecha(List<Post>
postsOtrosUsuarios){
      return postsOtrosUsuarios.stream().sorted((post1, post2) ->
post1.getFecha().compareTo(post2.getFecha())).collect(Collectors.t
oList());
}

private List<Post> filtrarPrimerosNPosts(int cantidad, List<Post>
postsOtrosUsuarios){
     return postsOtrosUsuarios.stream()
     .limit(cantidad).collect(Collectors.toList());
}
```

consultar: a veces al realizar refactorings en diferentes pasos el código intermedio queda erróneo o incluso sin sentido hasta que llega al código corregido final. eso está bien? o debe siempre quedar bien en cada paso? puede pasar. tratar de que vaya teniendo sentido, pero a veces varios refactorings van "de la mano" pero me piden q haga uno a la vez. el codigo final si debería tener sentido.

Ejercicio 2.4




```
public class Producto {
    private String nombre;
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}

public class ItemCarrito {
    private Producto producto;
    private int cantidad;

    public Producto getProducto() {
        return this.producto;
```

```
    }

    public int getCantidad() {
        return this.cantidad;
    }

}

public class Carrito {
    private List<ItemCarrito> items;

    public double total() {
      return this.items.stream()
            .mapToDouble(item ->
                  item.getProducto().getPrecio() *
               item.getCantidad())
            .sum();
    }
}
```

Bad Smell: **Feature Envy**
La clase Carrito ejecuta lógica que debería estar asignada a la clase ItemCarrito, por lo que hay responsabilidades mal asignadas.
Solución: **Move Method**
Se mueve la funcionalidad de obtener el precio total de un producto a un método getTotal en la clase ItemCarrito.

```
public class Producto {
    private String nombre;
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}

public class ItemCarrito {
    private Producto producto;
    private int cantidad;

    public Producto getProducto() {
        return this.producto;
    }

    public int getCantidad() {
        return this.cantidad;
    }
     public int getTotal() {
```

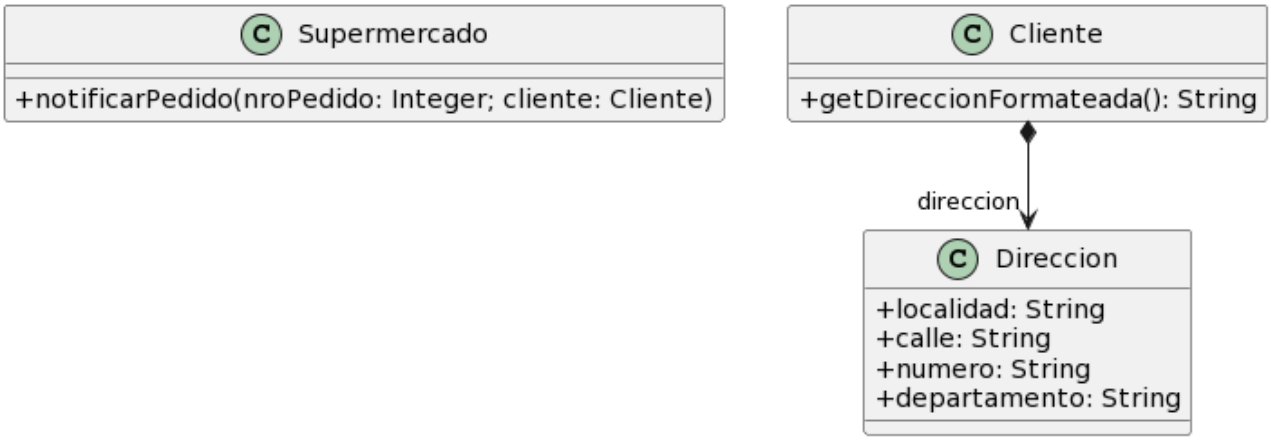
```
        return producto.getPrecio() * this.cantidad;
    }


}

public class Carrito {
    private List<ItemCarrito> items;

    public double total() {
      return this.items.stream()
            .mapToDouble(item ->
                  item.getTotal())
            .sum();
    }
}
```

Bad Smell: Nombre poco explicativo
El método total en Carrito y el método getTotal en ItemCarrito tienen nombres poco explicativos.
Solución: Rename Method
Se renombran ambos métodos.

```
public class Producto {
    private String nombre;
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}

public class ItemCarrito {
    private Producto producto;
    private int cantidad;

    public Producto getProducto() {
        return this.producto;
    }

    public int getCantidad() {
        return this.cantidad;
    }
     public int getPrecioTotalProducto() {
        return producto.getPrecio() * this.cantidad;
    }
```

```
}

public class Carrito {
    private List<ItemCarrito> items;

    public double getPrecioTotalCarrito() {
      return this.items.stream()
            .mapToDouble(item ->
                  item.getPrecioTotalProducto())
            .sum();
    }
}
```

Ejercicio 2.5




```
public class Supermercado {
   public void notificarPedido(long nroPedido, Cliente
cliente) {
     String notificacion = MessageFormat.format("Estimado
cliente, se le informa que hemos recibido su pedido con
número {0}, el cual será enviado a la dirección {1}", new
Object[] { nroPedido, cliente.getDireccionFormateada() });

     // lo imprimimos en pantalla, podría ser un mail, SMS,
etc..
    System.out.println(notificacion);
  }
}

public class Cliente {
   public String getDireccionFormateada() {
     return
          this.direccion.getLocalidad() + ", " +
          this.direccion.getCalle() + ", " +
          this.direccion.getNumero() + ", " +
          this.direccion.getDepartamento()
```

```
        ;
}
```

Bad Smell: **Feature Envy**
La clase Cliente está accediendo a varios atributos de Direccion, lo que indica que Direccion debería encargarse de eso.
Solución: **Move Method**
Se aplica Move Method y la clase Direccion pasa a encargarse de eso.

```
public class Supermercado {
   public void notificarPedido(long nroPedido, Cliente
cliente) {
     String notificacion = MessageFormat.format("Estimado
cliente, se le informa que hemos recibido su pedido con
número {0}, el cual será enviado a la dirección {1}", new
Object[] { nroPedido, cliente.getDireccionFormateada() });

     // lo imprimimos en pantalla, podría ser un mail, SMS,
etc..
    System.out.println(notificacion);
  }
}

public class Cliente {
   private Direccion direccion;
   public String getDireccionFormateada() {
     return this.direccion.toString();}
}

public class Direccion {
   public String localidad;
   public String calle;
   public String numero;
   public String departamento;

   public String toString() {
     return this.localidad + ", " +this.calle + ", " +
          this.numero + ", " +this.departamento;}
     }
```

Bad Smell: **Rompe el encapsulamiento**
En la clase Direccion, las variables publicas violan el encapsulamiento.
Solución: **Encapsulate Field**
Se modifica la visibilidad de las variables y pasan a ser privadas. Se agregan setters y getters.

```
public class Supermercado {
```

```
   public void notificarPedido(long nroPedido, Cliente
cliente) {
     String notificacion = MessageFormat.format("Estimado
cliente, se le informa que hemos recibido su pedido con
número {0}, el cual será enviado a la dirección {1}", new
Object[] { nroPedido, cliente.getDireccionFormateada() });

     // lo imprimimos en pantalla, podría ser un mail, SMS,
etc..
    System.out.println(notificacion);
  }
}

public class Cliente {
   private Direccion direccion;
   public String getDireccionFormateada() {
     return this.direccion.toString();}
}

public class Direccion {
   private String localidad;
   private String calle;
   private String numero;
   private String departamento;

   public String getLocalidad() {
     return this.localidad;}
   public String getCalle() {
     return this.calle;}
   public String getNumero() {
     return this.numero;}
   public String getDepartamento() {
     return this.departamento;}

   public void setLocalidad(String localidad) {
     this.localidad = localidad;}
   public void setCalle(String calle) {
     this.calle = calle;}
   public void setNumero(String numero) {
     this.numero = numero;}
   public void setDepartamento(String departamento) {
     this.departamento = departamento;}

   public String toString() {
     return this.localidad + ", " +this.calle + ", " +
          this.numero + ", " +this.departamento;}
     }
```

CONSULTAR ESTE BAD SMELL: ES NECESARIO? ESTA BIEN ELIMINAR CLIENTE? no está tan bueno eliminar la clase Cliente, el supermercado debería conocer al cliente, no a la direccion. aunque depende del contexto, pero no lo tengo.
mejor dejar la clase cliente. (si tuviese una clase persona que es un cliente y tiene direccion tal vez persona no fuera necesario pero igual habría q evaluar el caso especifico) es ideal no eliminar las clases q ya existen modificando tanto el diseño

Bad Smell: **Middle Man**

La clase Cliente solo actúa de intermediario entre Supermercado y Direccion.

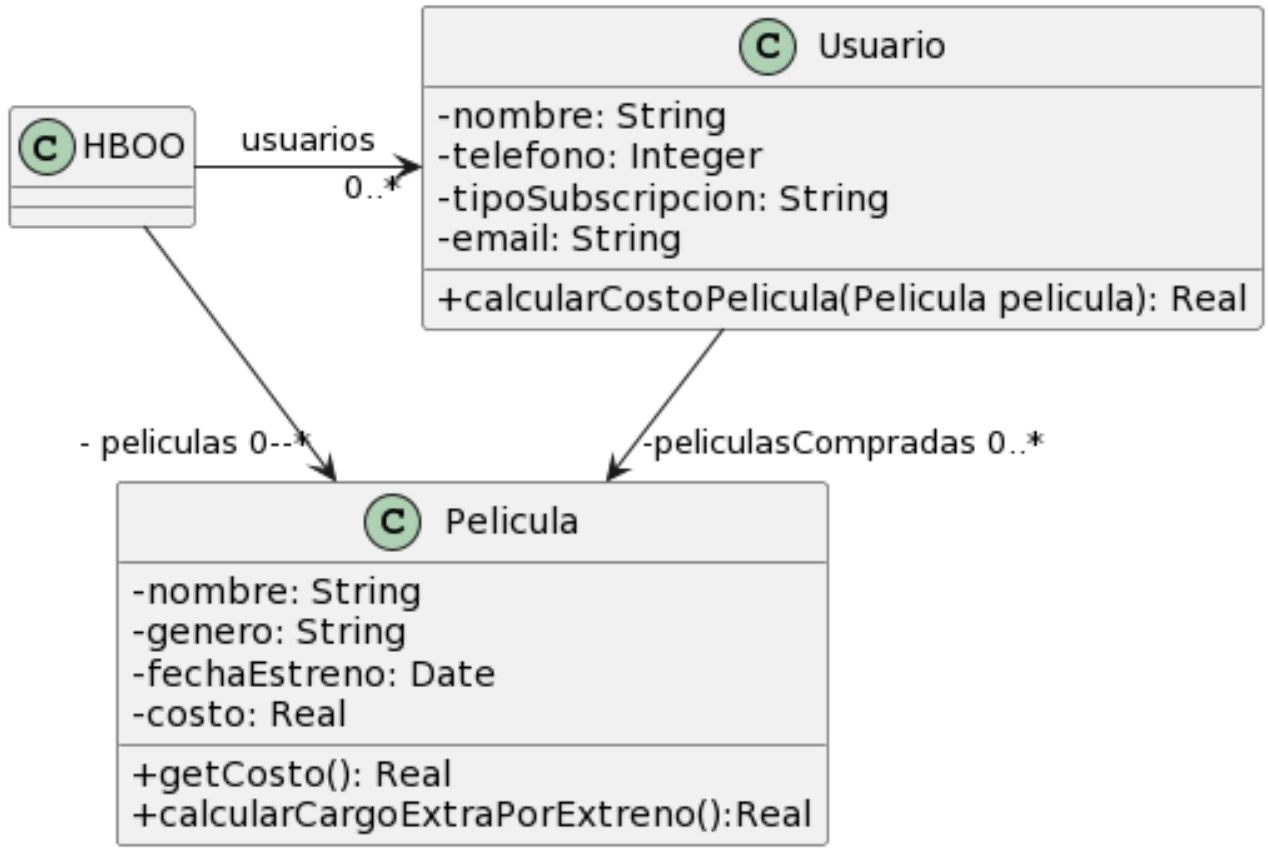
Solución: **Remove Middle Man**

Se elimina la clase Cliente.

```
public class Supermercado {
   public void notificarPedido(long nroPedido, Direccion
direccion) {
     String notificacion = MessageFormat.format("Estimado
cliente, se le informa que hemos recibido su pedido con
número {0}, el cual será enviado a la dirección {1}", new
Object[] { nroPedido, direccion.toString() });

     // lo imprimimos en pantalla, podría ser un mail, SMS,
etc..
    System.out.println(notificacion);
  }
}

public class Direccion {
   private String localidad;
   private String calle;
   private String numero;
   private String departamento;

   public String getLocalidad() {
     return this.localidad;}
   public String getCalle() {
     return this.calle;}
   public String getNumero() {
     return this.numero;}
   public String getDepartamento() {
     return this.departamento;}

   public void setLocalidad(String localidad) {
     this.localidad = localidad;}
   public void setCalle(String calle) {
     this.calle = calle;}
   public void setNumero(String numero) {
     this.numero = numero;}
   public void setDepartamento(String departamento) {
     this.departamento = departamento;}
```

```
    public String toString() {
      return this.localidad + “, ” + this.calle + “, ” +
           this.numero + “, ” + this.departamento;}
      }
```

Ejercicio 2.6




```
public class Usuario {
    private String tipoSubscripcion;
    // ...

    public void setTipoSubscripcion(String unTipo) {
      this.tipoSubscripcion = unTipo;
    }

    public double calcularCostoPelicula(Pelicula pelicula) {
      double costo = 0;
      if (tipoSubscripcion=="Basico") {
           costo = pelicula.getCosto() +
pelicula.calcularCargoExtraPorEstreno();
      }
      else if (tipoSubscripcion== "Familia") {
           costo = (pelicula.getCosto() +
pelicula.calcularCargoExtraPorEstreno()) * 0.90;
```

```
      }
      else if (tipoSubscripcion=="Plus") {
           costo = pelicula.getCosto();
      }
      else if (tipoSubscripcion=="Premium") {
           costo = pelicula.getCosto() * 0.75;
      }
      return costo;
    }
}

public class Pelicula {
    LocalDate fechaEstreno;
    // ...

    public double getCosto() {
      return this.costo;
    }

    public double calcularCargoExtraPorEstreno(){
     // Si la Película se estrenó 30 días antes de la fecha
actual, retorna un cargo de 0$, caso contrario, retorna un
cargo extra de 300$
     return (ChronoUnit.DAYS.between(this.fechaEstreno,
LocalDate.now()) ) > 30 ? 0 : 300;
    }
}
```

CONSULTAR: está bien resuelto? siento q puedo estar haciendo varios refactorings a la vez y no me doy cuenta : en este caso estaría bien xq en si el replace conditional with polymorphism viene con todo lo que hice. sin embargo, podria pensarse que se esta haciendo move method extract method, etc.

Bad Smell: Switch Statements - Obsesión por los primitivos
El método calcularCostoPelicula de la clase Usuario esta lleno de if-else (switch).
Solución: Replace conditional with Polymorphism
Se reemplazan las sentencias if-else (switch) y se aplica polimorfismo creando la interfaz Subscripcion.

```
public class Usuario {
    private Subscripcion tipoSubscripcion;
    // ...

    public void setTipoSubscripcion(Subscripcion
tipoSubscripcion) {
      this.tipoSubscripcion = tipoSubscripcion;
    }
```

```
    public double calcularCostoPelicula(Pelicula pelicula)
     return this.tipoSubscripcion
          .calcularCostoPelicula(película);}
}


public interface Subscripcion {
    public double calcularCostoPelicula(Pelicula pelicula;
}

public class SubscripcionBasica implements Subscripcion{
    public double calcularCostoPelicula(Pelicula pelicula{
     return pelicula.getCosto()+
          pelicula.calcularCargoExtraPorEstreno();
}

public class SubscripcionFamilia implements Subscripcion{
    public double calcularCostoPelicula(Pelicula pelicula{
     return pelicula.getCosto() +
          pelicula.calcularCargoExtraPorEstreno()) * 0.90;
}

public class SubscripcionPlus implements Subscripcion{
    public double calcularCostoPelicula(Pelicula pelicula{
     return pelicula.getCosto();
}

public class SubscripcionPremium implements Subscripcion{
    public double calcularCostoPelicula(Pelicula pelicula{
     return pelicula.getCosto() * 0.75;
}

public class Pelicula {
    LocalDate fechaEstreno;
    // ...

    public double getCosto() {
      return this.costo;
    }

    public double calcularCargoExtraPorEstreno(){
     // Si la Película se estrenó 30 días antes de la fecha
actual, retorna un cargo de 0$, caso contrario, retorna un
cargo extra de 300$
     return (ChronoUnit.DAYS.between(this.fechaEstreno,
LocalDate.now()) ) > 30 ? 0 : 300;
    }
```

```
}
```

Ejercicio 3

Dado el siguiente código implementado en la clase Document y que calcula algunas estadísticas del mismo:

```
public class Document {
    List<String> words;

    public long characterCount() {
        long count = this.words
                    .stream()
                    .mapToLong(w -> w.length())
                    .sum();
        return count;
        }
    public long calculateAvg() {
        long avgLength = this.words
                    .stream()
                    .mapToLong(w -> w.length())
                    .sum() / this.words.size();
        return avgLength;
        }
// Resto del código que no importa
}
```

Bad Smell: Rompe el encapsulamiento
La variable words en la clase Document es pública, lo que viola el encapsulamiento.
Solución: Encapsulate Field
Se cambia la visibilidad de la variable a privada.

```
public class Document {
    private List<String> words;

    public void setWords(…) { … }
    public List<String> getWords() { … }

    public long characterCount() {
        long count = this.words
                    .stream()
```

```
                .mapToLong(w -> w.length())
                .sum();
        return count;
        }
    public long calculateAvg() {
        long avgLength = this.words
                .stream()
                .mapToLong(w -> w.length())
                .sum() / this.words.size();
        return avgLength;
        }
// Resto del código que no importa
}
```

Bad Smell: **Duplicated code**
Los métodos characterCount y calculateAvg repiten el mismo código contando la cantidad de caracteres de una palabra.
Solución: **Extract Method**

```
public class Document {
    private List<String> words;

    public void setWords(…) { … }
    public List<String> getWords() { … }

    public long characterCount() {
        long count = this.words
                .stream()
                .mapToLong(w -> w.length())
                .sum();
        return count;
        }
    public long calculateAvg() {
        long avgLength = this.characterCount() / this.words.size();
        return avgLength;
        }
// Resto del código que no importa
}
```

Bad Smell: **Temporary field**
Los métodos characterCount y calculateAvg hacen uso de variables temporales innecesarias que pueden obviarse.
Solución: **Replace temp with query**
Se eliminan las variables temporales y se devuelve directamente el cálculo que realiza cada método.

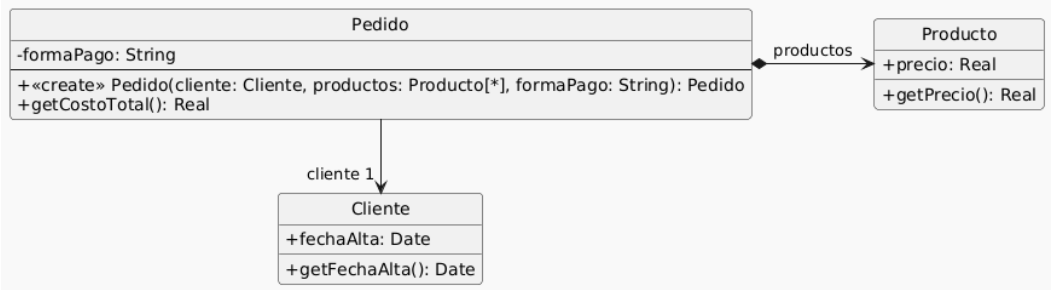
```
public class Document {
    private List<String> words;

    public void setWords(…) { … }
    public List<String> getWords() { … }

    public long characterCount() {
        return this.words
                    .stream()
                    .mapToLong(w -> w.length())
                    .sum();

        }
    public long calculateAvg() {
        return this.characterCount() / this.words.size();
        }
// Resto del código que no importa
}
```

Errores:

- **Tipo del método calculateAvg:** el método calculateAvg devuelve un objeto de tipo long pero en realidad debería devolver un double ya que el promedio puede tener decimales. No puedo modificarlo porque al hacer refactoring no puedo modificar comportamiento del código.

- **Posible división por cero:** el método calculateAvg podría querer dividir por cero si this.words.size() devuelve cero, es decir, si la lista de palabras está vacía. No puedo modificarlo porque al hacer refactoring no puedo modificar comportamiento del código.

Ejercicio 4




```
01: public class Pedido {
02:  private Cliente cliente;
03:  private List<Producto> productos;
04:  private String formaPago;
05:  public Pedido(Cliente cliente, List<Producto> productos, String
formaPago) {
06:    if (!"efectivo".equals(formaPago)
07:       && !"6 cuotas".equals(formaPago)
08:       && !"12 cuotas".equals(formaPago)) {
09:         throw new Error("Forma de pago incorrecta");
10:   }
11:   this.cliente = cliente;
12:   this.productos = productos;
13:   this.formaPago = formaPago;
14:  }
15:  public double getCostoTotal() {
16:    double costoProductos = 0;
17:    for (Producto producto : this.productos) {
18:      costoProductos += producto.getPrecio();
19:    }
20:    double extraFormaPago = 0;
21:    if ("efectivo".equals(this.formaPago)) {
22:      extraFormaPago = 0;
23:    } else if ("6 cuotas".equals(this.formaPago)) {
24:      extraFormaPago = costoProductos * 0.2;
25:    } else if ("12 cuotas".equals(this.formaPago)) {
26:      extraFormaPago = costoProductos * 0.5;
27:    }
28:    int añosDesdeFechaAlta =
Period.between(this.cliente.getFechaAlta(), LocalDate.now()).getYears();
29:    // Aplicar descuento del 10% si el cliente tiene más de 5 años de
antiguedad
30:    if (añosDesdeFechaAlta > 5) {
31:      return (costoProductos + extraFormaPago) * 0.9;
32:    }
33:    return costoProductos + extraFormaPago;
34:  }
35: }
36: public class Cliente {
37:  private LocalDate fechaAlta;
38:  public LocalDate getFechaAlta() {
39:    return this.fechaAlta;
40:  }
41: }
42: public class Producto {
43:  private double precio;
44:  public double getPrecio() {
45:    return this.precio;
46:  }
47: }
```

Tareas:

1. Dado el código anterior, aplique **únicamente** los siguientes refactoring:
   - Replace Loop with Pipeline (líneas 16 a 19)
   - Replace Conditional with Polymorphism (líneas 21 a 27)
   - Extract method y move method (línea 28)
   - Extract method y replace temp with query (líneas 28 a 33)

2. Realice el diagrama de clases del código refactorizado.

Bad Smell: Switch statements (lineas 16 a 19)

Entre las lineas 16 y 19 se utiliza una sentencia for que no se considera eficiente.

Solución: Replace Loop with Pipeline

Se reemplaza la estructura for por stream, el cual es más eficiente y moderno.

código original	código modificado
16: double costoProductos = 0; 17: for (Producto producto : this.productos) { 18: costoProductos += producto.getPrecio(); 19: }	16: double costoProductos = this.productos.stream(); 17: .mapToDouble(p -> p.getPrecio()).sum();}

Bad Smell: Switch statements (linea 21 a 27)

Entre las lineas 21 y 27 se utiliza una estructura de if-else (switch) que no se considera eficiente.

Solución: Replace Conditional with Polymorphism

Se reemplaza la estructura if-else (switch) haciendo uso de polimorfismo creando una interfaz FormaDePago con 3 clases que la implementan (Efectivo, SeisCuotas, DoceCuotas).

código original	código modificado

20: double extraFormaPago = 0;
21: if ("efectivo".equals(this.formaPago)) {
22: extraFormaPago = 0;
23: } else if ("6 cuotas".equals(this.formaPago)) {
24: extraFormaPago = costoProductos * 0.2;
25: } else if ("12 cuotas".equals(this.formaPago)) {
26: extraFormaPago = costoProductos * 0.5;
27: }
	01: public class Pedido {
	.....
	private FormaDePago formaPago;
	.....
	double extraFormaPago =
	this.formaPago.calcularPrecio(costoProductos); }
	.....
	public interface FormaDePago {
	public double calcularPrecio(double costoProductos); }
	.....
	public class Efectivo implements FormaDePago {
	public double calcularPrecio(double costoProductos){
	return 0; }}
	public class SeisCuotas implements FormaDePago {
	public double calcularPrecio(double costoProductos){
	return costoProductos * 0.2; }}
	public class DoceCuotas implements FormaDePago {
	public double calcularPrecio(double costoProductos){
	return costoProductos * 0.5; }}

Bad Smell: Long method y feature envy (linea 28)
En la linea 28 se resuelve algo que debería ser un método aparte, además debería ser responsabilidad de la clase Cliente.
Solución: Extract method y move method (línea 28)
Se extrae la linea 28 a un método, y además se mueve ese método a la clase Cliente.

código original	código modificado
.....	.....
28: int añosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), LocalDate.now()).getYears();	28: int añosDesdeFechaAlta = this.cliente.calcularAntiguedad();
.....	.....
	36: public class Cliente {
	37: private LocalDate fechaAlta;
	38: public LocalDate getFechaAlta() {
	39: return this.fechaAlta;
	40: }
	41: public int calcularAntiguedad() {
	42: return Period.between(this.fechaAlta, LocalDate.now()).getYears();
	43: }
	44: }
	.....

Bad Smell: Temporary field y long method
Método largo que tiene responsabilidades que deberían ser extraídas a un método aparte. Además, se usa variable temporal innecesaria.
Solución: Extract method y replace temp with query (líneas 28 a 33)
Se extrae el método y se elimina variable temporal innecesaria.

código original	código modificado
28: int añosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), LocalDate.now()).getYears();	28: return calcularDescuento(costoProductos,
29: // Aplicar descuento del 10% si el cliente tiene más de 5 años de antiguedad	29: this.cliente.calcularAntiguedad(), extraFormaPago);}
30: if (añosDesdeFechaAlta > 5) {	30: public double calcularDescuento(double costoProductos,
31: return (costoProductos + extraFormaPago) * 0.9;	31: int anosDesdeFechaAlta, double extraFormaPago){
32: }	32: if (añosDesdeFechaAlta > 5) {
33: return costoProductos + extraFormaPago;	33: return (costoProductos + extraFormaPago) * 0.9;
34: }	34: }
35: }	35: return costoProductos + extraFormaPago;
	36: }

Código final

```
01: public class Pedido {
02:  private Cliente cliente;
03:  private List<Producto> productos;
04:  private FormaDePago formaPago;
05:  public Pedido(Cliente cliente, List<Producto> productos, FormaDePago
formaPago) {

11:    this.cliente = cliente;
12:    this.productos = productos;
13:    this.formaPago = formaPago;
14:   }
15:   public double getCostoTotal() {
16:   double costoProductos = this.productos.stream();
17:     .mapToDouble(p -> p.getPrecio()).sum();}
18:   double extraFormaPago =
19:   this.formaPago.calcularPrecio(costoProductos);

20:     return calcularDescuento(costoProductos,
21:           this.cliente.calcularAntiguedad(), extraFormaPago);}

22:     public double calcularDescuento(double costoProductos,
23:           int anosDesdeFechaAlta, double extraFormaPago){
24:           if (añosDesdeFechaAlta > 5) {
25:             return (costoProductos + extraFormaPago) * 0.9;
26:     }
27:     return costoProductos + extraFormaPago;
28:   }
29: public class Cliente {
30:   private LocalDate fechaAlta;
31:   public LocalDate getFechaAlta() {
32:     return this.fechaAlta;
33:   }
34:   public int calcularAntiguedad() {
35:     return Period.between(this.fechaAlta, LocalDate.now()).getYears();
36:   }
37: }
38: public class Producto {
39:   private double precio;
40:   public double getPrecio() {
41:     return this.precio;
42:   }
43: }

public interface FormaDePago {
public double calcularPrecio(double costoProductos); }
……..
public class Efectivo implements FormaDePago {
public double calcularPrecio(double costoProductos){
 return 0;  }}
public class SeisCuotas implements FormaDePago {
public double calcularPrecio(double costoProductos){
 return costoProductos * 0.2;   }}
public class DoceCuotas implements FormaDePago {
public double calcularPrecio(double costoProductos){
return costoProductos * 0.5;   }}
```

Linea 18-21: la variable temporal extraFormaPago podría obviarse enviando directamente como parámetro en la linea 21:
this.formaPago.calcularPrecio(costoProductos).
Sería Temporary field el bad smell y Replace temp with query el refactoring. No lo hice para que fuese más sencillo leerlo.

(bien justificado lo previo. si en un ejercicio o en un parcial me piden refactorings de lineas especificas, mejor enfocarme en eso nada mas y no buscar bad smells en otros lugares ya que a veces le puedo errar y no es lo que pide el ejercicio)