Ejercicios

Ejercicio 1 Red Social

Se quiere programar en objetos una versión simplificada de una red social parecida a Twitter. Este servicio debe permitir a los usuarios registrados postear y leer mensajes de hasta 280 caracteres. Ud. debe modelar e implementar parte del sistema donde nos interesa que quede claro lo siguiente:

- Cada usuario conoce todos los Tweets que hizo.
- Un tweet puede ser re-tweet de otro, y este tweet debe conocer a su tweet de origen.
- Twitter debe conocer a todos los usuarios del sistema.
- Los tweets de un usuario se deben eliminar cuando el usuario es eliminado. No existen tweets no referenciados por un usuario.
- Los usuarios se identifican por su screenName.
- No se pueden agregar dos usuarios con el mismo screenName.
- Los tweets deben tener un texto de 1 carácter como mínimo y 280 caracteres como máximo.
- Un re-tweet no tiene texto adicional.

Tareas:

Su tarea es diseñar y programar en Java lo que sea necesario para ofrecer la funcionalidad antes descrita. Se espera que entregue los siguientes productos.

1. Diagrama de clases UML.
2. Implementación en Java de la funcionalidad requerida.
3. Implementar los tests (JUnit) que considere necesarios.

Nota: para crear el proyecto Java, lea el material llamado “Trabajando en OO2 con proyectos Maven”. Código en plantUML:

```
Class Twitter {
 + <<create>> Twitter(): Twitter

 + addUser(screenName: String): Boolean;

 + deleteUser(user: User): boolean

}
Twitter -> User : 0..* users;

class User{
  - screenName: String

 + <<create>> User(screenName: String): User

 + postTweet(message: String): Post

 + postReTweet(tweet: Tweet): Post

 + deleteTweets(): boolean
```

```
 + isEqualName (text: String): boolean
}

User -> Post : 0..* tweets

interface Post {
 + eliminar(): boolean
}

class ReTweet implements Post {

 + <<create>> ReTweet(origin: Tweet): ReTweet
 + eliminar(): boolean;

}
ReTweet -> Tweet : origin
class Tweet implements Post{
 - message: String

 + <<create>> Tweet(message: String): Tweet

 + eliminar(): boolean

 + {static} isValidLength( text: String): boolean;
}
```

Ejercicio 2 Piedra Papel o Tijera

Se quiere programar en objetos una versión del juego Piedra Papel o Tijera. En este juego dos jugadores eligen entre tres opciones: piedra, papel o tijera. La piedra aplasta la tijera, la tijera corta el papel, y el papel envuelve la piedra. Los jugadores eligen una opción y se determina un ganador según las reglas:

	Piedra	**Papel**	**Tijera**
Piedra	Empate	Papel	Piedra
Papel	Papel	Empate	Tijera
Tijera	Piedra	Tijera	Empate

Tareas:

1. Diseñe e implemente una solución a este problema, de forma tal que dadas dos opciones, determine cuál fue la ganadora, o si hubo empate
2. Se desea extender al juego a una versión más equitativa que integre a lagarto y Spock, con las siguientes reglas:
    1. Piedra aplasta tijera y aplasta lagarto.
    2. Papel cubre piedra y desaprueba Spock.
    3. Tijera corta papel y decapita lagarto.
    4. Lagarto come papel y envenena Spock.
    5. Spock rompe tijera y vaporiza piedra.

¿Qué cambios se necesitan agregar?

3. Agregue los cambios a la solución anterior.

Refactoring

Ejercicio 1: Algo huele mal

Indique qué malos olores se presentan en los siguientes ejemplos.

1.1 Protocolo de Cliente

La clase Cliente tiene el siguiente protocolo. ¿Cómo puede mejorarlo?

```
**

/**

* Retorna el límite de crédito del cliente

*/

public double lmtCrdt() {...



/**

* Retorna el monto facturado al cliente desde la fecha f1 a la fecha f2

*/

protected double mtFcE(LocalDate f1, LocalDate f2) {...



/**

* Retorna el monto cobrado al cliente desde la fecha f1 a la fecha f2

*/

private double mtCbE(LocalDate f1, LocalDate f2) {...

**
```

Primero que nada utilizaría el método Change Function Declaration ya que los nombres de las funciones me parecen muy poco claros, los nombres de las funciones deberían decir que hacen así es mas sencillo de entender. Quizás tambien renombraría los parámetros como startTime y endTime para saber cual debe ser mayor sin necesidad de entrar a la declaración

1.2 Participación en proyectos

Al revisar el siguiente diseño inicial (Figura 1), se decidió realizar un cambio para evitar lo que se consideraba un mal olor. El diseño modificado se muestra en la Figura 2. Indique qué tipo de cambio se

realizó y si lo considera apropiado. Justifique su respuesta.

Diseño inicial:

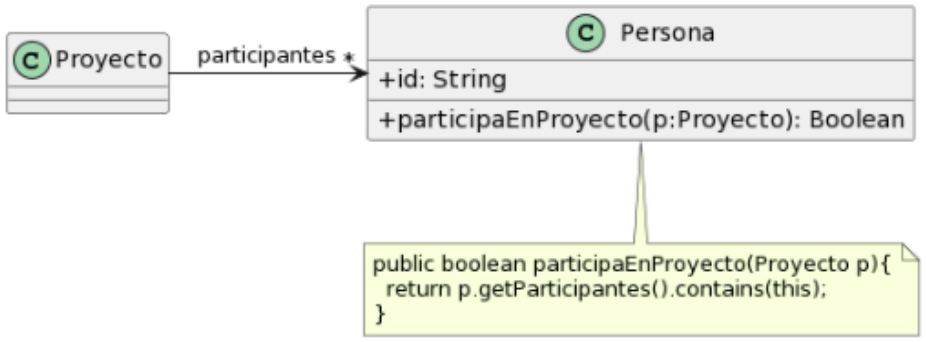
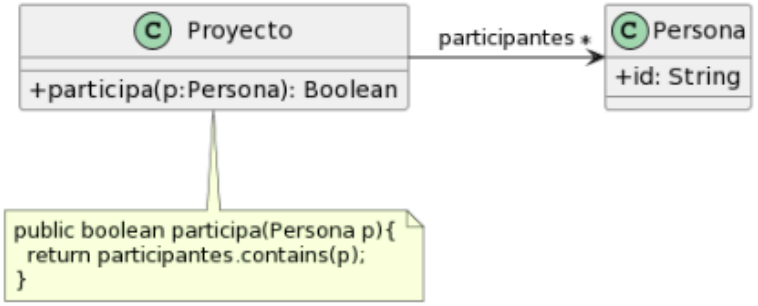


Figura 1: Diagrama de clases del diseño inicial.

Diseño revisado:



Se utilizó Move method, y lo considero apropiado ya que me parece más que un proyecto sepa quiénes participaron en el, y a partir de ahí saber si esa persona que recibe participa o no. Además la solución anterior poseía envidia de atributos, rompía el encapsulamiento y poseía una responsabilidad mal asignada

1.3 Cálculos

Analice el código que se muestra a continuación. Indique qué code smells encuentra y cómo pueden corregirse.

```
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);

}
```

1. Método con demasiadas responsabilidades, debería separarse en métodos diferentes(getTotalSalarios, getPromedioEdades)
2. Nombre poco explícito de lo que hace
3. Utilización de bucles de tipo for/while cuando se pueden utilizar librerías/frameworks más modernos los cuales facilitan el trabajo
4. Junto con el uso de bucles viene el uso de variables temporales que deberían no usarse
5. Podría utilizarse el método average de los streams para las edades del personal

Ejercicio 2

Para cada una de las siguientes situaciones, realice en forma iterativa los siguientes pasos:

(i) indique el mal olor,

(ii) indique el refactoring que lo corrige,

(iii) aplique el refactoring, mostrando el resultado final (código y/o diseño según corresponda).

Si vuelve a encontrar un mal olor, retorne al paso (i).

2.1 Empleados

```
**

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);

    }

}

**

**
```

Malos olores y soluciones :
Código dupicado, se podría utilizar extract superclass para generalizar los datos comunes a cada empleado

```
public abstract class Empleado {
    public String nombre;

    public String apellido;

    public double sueldoBasico = 0;


}
**

public class EmpleadoTemporario extends Empleado {



    private double horasTrabajadas = 0;

    private int cantidadHijos = 0;

    // ......

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 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);

    }

}

**

**
```

Rompe el encapsulamiento, variables de instancia de clases públicas. Utilizaría Encapsulate Field para que no puedan ser accedidas

```
public abstract class Empleado {
    private String nombre;

    private String apellido;

    private double sueldoBasico = 0;

    protected double getSueldoBasico(){
    return this.sueldoBasico
    }
}
**

public class EmpleadoTemporario extends Empleado {



    private double horasTrabajadas = 0;

    private int cantidadHijos = 0;

    // ......

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 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);

    }

}

**

**
```

Se puede generalizar aún más ya que sigue habiendo código duplicado entre el empleado temporario y de planta. Se utiliza método Extract SuperClass

```
public abstract class Empleado {
    private String nombre;

    private String apellido;
```

```
    private double sueldoBasico = 0;

    protected double getSueldoBasico(){
    return this.sueldoBasico
    }
}
**
public abstract class EmpleadoConHijos extends Empleado {
    protected int cantidadHijos = 0;

}
public class EmpleadoTemporario extends EmpleadoConHijos {



    private double horasTrabajadas = 0;



    // ......

public double sueldo() {

return this.sueldoBasico

+ (this.horasTrabajadas * 500)

- (this.cantidadHijos * 1000)


- (this.sueldoBasico * 0.13);


}

}
**

public class EmpleadoPlanta extends EmpleadoConHijos {



    // ......

    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);

    }

}

**

**
```

El siguiente mal olor detectado es que hay código duplicado en todas las subclases de empleado con el método sueldo, debería generalizarse en la clase Empleado. Utilizar método Pull up Method para refactorizar

```
public abstract class Empleado {
    private String nombre;

    private String apellido;

    private double sueldoBasico = 0;

    protected double getSueldoBasico(){
    return this.sueldoBasico
    }
    public double sueldo(){
        return this.sueldoBasico + this.extra() - (this.sueldoBasico * 0.13);
    }
    public abstract double extra();
}
**
public abstract class EmpleadoConHijos extends Empleado {
    protected int cantidadHijos = 0;


}
public class EmpleadoTemporario extends EmpleadoConHijos {



    private double horasTrabajadas = 0;



    // ......

public double extra(){
```

```
    return (this.horasTrabajadas * 500) + (this.cantidadHijos * 1000);

}

}
**

public class EmpleadoPlanta extends EmpleadoConHijos {



    // ......

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

}

}



public class EmpleadoPasante extends Empleado {

public double extra(){
return 0
}

}

**

**
```

2.2 Juego

```
    **
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;

}



}

**
```

El primer mal olor encontrado es que se rompe el encapsulamiento de la clase Jugador, ya que sus variables de instancia se encuentran como públicas. Se debería utilizar el método de refactoring es Encapsulate Field

```
    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;

}
```

Junto con ello, además de que el error persiste en la clase Juego ya que las variables no son públicas, el siguiente mal olor es envidia de atributos. Que se soluciona con el método Move Method ya que no le corresponde al juego modificar la puntuación del jugador directamente

```
    public class Juego {

    // ......

    public void incrementar(Jugador j) {

        j.incrementar();

    }

    public void decrementar(Jugador j) {

        j.decrementar();
    }


public class Jugador {

    private String nombre;

    private String apellido;

    private int puntuacion = 0;

    public void incrementar(){
        this.puntuacion = puntuacion + 100;
    }

    public void decrementar(){
        this.puntuacion = puntuacion - 50;
    }

}
```

2.3 Publicaciones

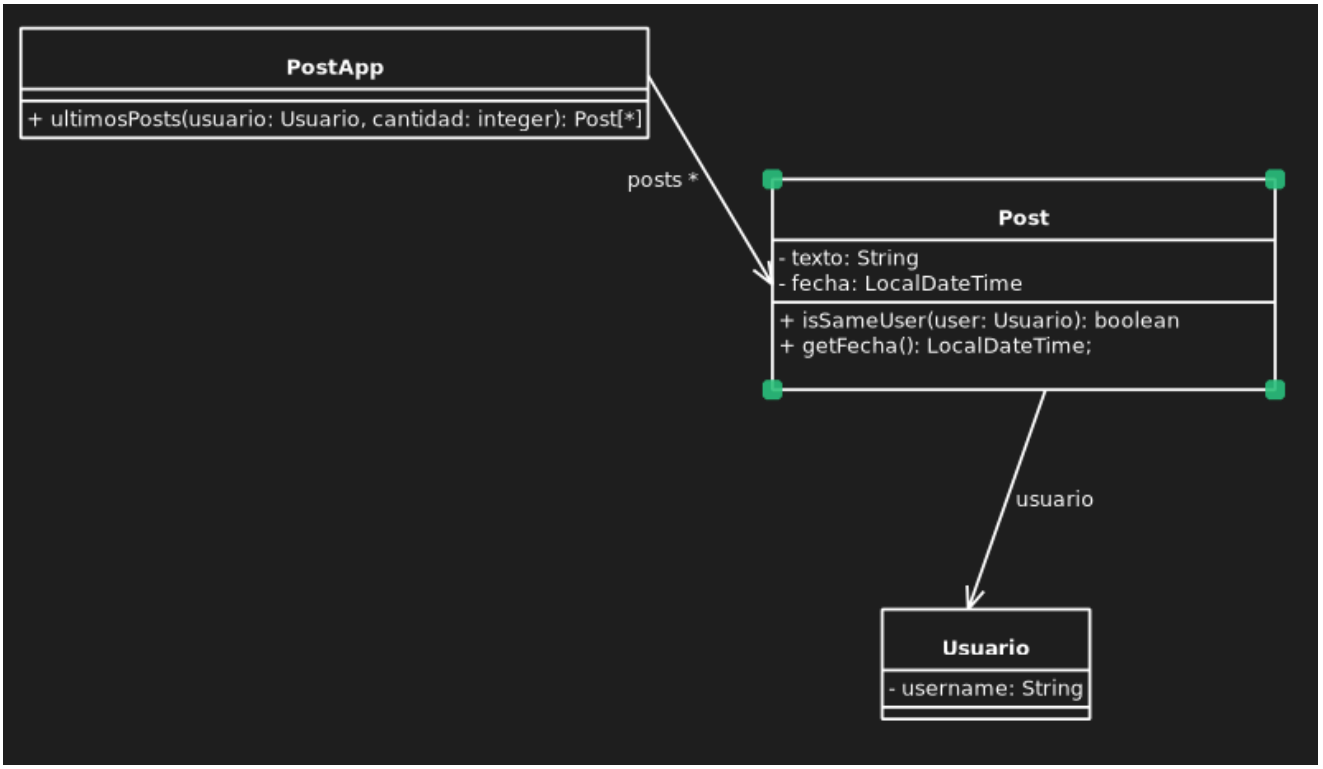
1. Malos olores:
    1. Envidia de atributos, en el caso de la fecha del post y el usuario en el post.
    2. Reinventando la rueda en caso de los for para quedarse con las publicaciones más recientes y para obtener una lista de los post mas recientes según la cantidad recibida por parámetro
    3. Clase post es solo una clase de datos, no tiene ningun tipo de comportamiento. Solo getters y setters
    4. Rompe el encapsulamiento ya que accede directamente al usuario de el Post
    5. Método largo, incluye comentarios en el medio los cuales a su vez indican que el código no es demasiado legible
    6. El nombre del método no me parece el adecuado ya que no indica lo que retorna si no que pareciera que retorna todos los posts

```
**

/**

* 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;
}

**
```

Envidia de atributos en el usuario del post. Ya que en lugar de que la clase post compare su usuario con el recibido, la clase que tiene los post lo hace. Utilizaría Hide Delegate/Move Method, creando un método

en la clase Post el cual sea, isSameUser

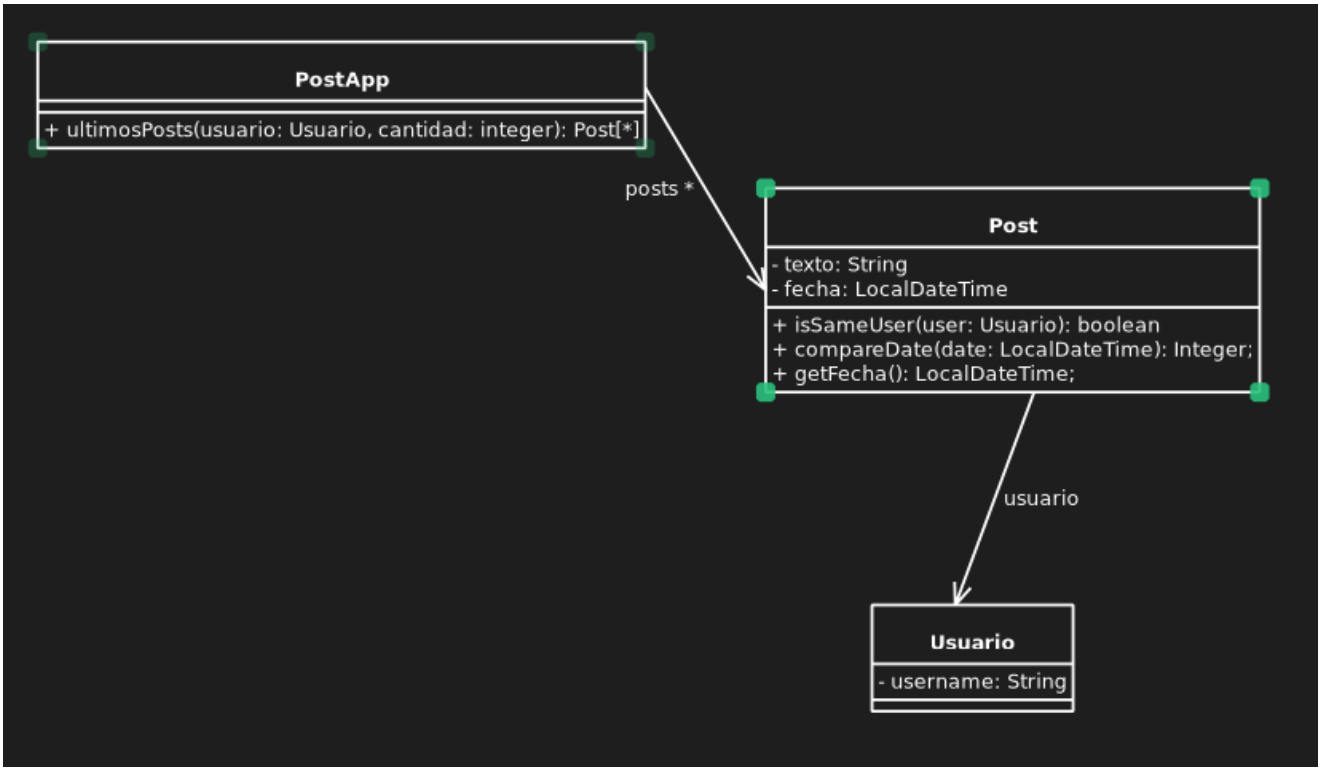



```
**

/**

* 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.isSameUser(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;

}


**
```

Envidia de atributos, en el caso de la fecha del post. Ya que
en lugar de que la clase post su fecha con la recibida, la clase que tiene los post lo hace. Utilizaría Hide

Delegate, creando un método en la clase Post el cual utilizaría el compareTo, que sería compareDate




```
**

/**

* 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.isSameUser(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).compareDate(

     postsOtrosUsuarios.get(masNuevo).getFecha()) > 0 ) {
```

```
                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;

}


**
```

Lo siguiente que vemos, es que hay un método largo, el cual posee mas reesponsabilidades de las que debería. Comenzaré utilizando extract method para el primer for

```
public List<Post> postSinUsuario(Usuario user){
List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {

        if (!post.isSameUser(user)) {

            postsOtrosUsuarios.add(post);

        }

    }
    return postOtrosUsuarios;
}

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

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    postOtrosUsuarios = this.postsSinUsuario(user);
```

```
    // 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).compareDate(
      postsOtrosUsuarios.get(masNuevo).getFecha()) > 0 ) {
               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;
}
```

Sigue habiendo un método largo, el cual posee mas reesponsabilidades de las que debería. Aplico de nuevo extract method para el 2do for

```
public List<Post> postSinUsuario(Usuario user){
List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {
        if (!post.isSameUser(user)) {
            postsOtrosUsuarios.add(post);
```

```
        }

    }
    return postOtrosUsuarios;
}

public List<Post> postOrdenadosPorFecha(List<Post> users){
    // ordena los posts por fecha

   for (int i = 0; i < users.size(); i++) {

       int masNuevo = i;

       for(int j= i +1; j < userss.size(); j++) {

           if (users.get(j).compareDate(

     users.get(masNuevo).getFecha()) > 0 ) {

              masNuevo = j;

           }

       }

       }

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

      users.set(masNuevo, unPost);

   }
}

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

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    postOtrosUsuarios = this.postsSinUsuario(user);


    postOtrosUsuarios = this.postOrdenadosPorFecha(postOtrosUsuarios);

    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;

}
```

Ahora que ya delegamos las responsabilidades a otros métodos veamos el método ultimos post. Se está "reinventando la rueda" ya que existen métodos para limitar lo que se quiere retornar de una lista. Vamos a utilizar Replace loop with pipeline

```
public List<Post> postSinUsuario(Usuario user){
List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {

        if (!post.isSameUser(user)) {

            postsOtrosUsuarios.add(post);

        }

    }
    return postOtrosUsuarios;
}

public List<Post> postOrdenadosPorFecha(List<Post> users){
    // ordena los posts por fecha

   for (int i = 0; i < users.size(); i++) {

       int masNuevo = i;

       for(int j= i +1; j < userss.size(); j++) {

           if (users.get(j).compareDate(

     users.get(masNuevo).getFecha()) > 0 ) {

              masNuevo = j;

           }

       }

       }

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

      users.set(masNuevo, unPost);

   }
}

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

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    postOtrosUsuarios = this.postsSinUsuario(user);


    postOtrosUsuarios = this.postOrdenadosPorFecha(postOtrosUsuarios);
```

```
    List<Post> ultimosPosts = new ArrayList<Post>();

    ultimosPosts =
postOtrosUsuarios.stream().limit(cantidad).collect(Collectors.toList());

return ultimosPosts;

}
```

Otro de los malos olores que sigue estando el método postOrdenadosPorFecha es "Reinventando la rueda". Para solucionarlo utilizo Replace loop with pipeline

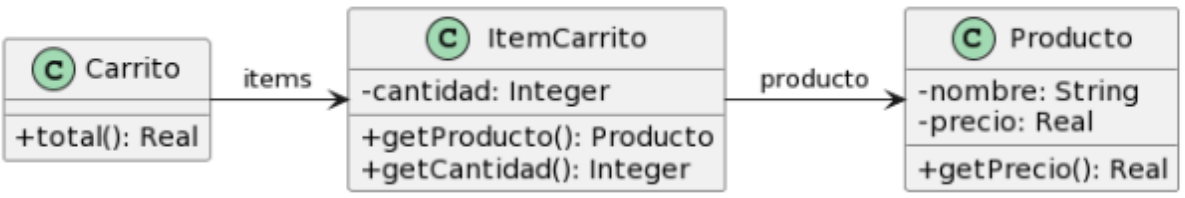
```
public List<Post> postSinUsuario(Usuario user){
List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    for (Post post : this.posts) {

        if (!post.isSameUser(user)) {

            postsOtrosUsuarios.add(post);

        }

    }
    return postOtrosUsuarios;
}

public List<Post> postOrdenadosPorFecha(List<Post> users){
    // ordena los posts por fecha
    **

return users.stream()

     .sorted((p1, p2) ->
p2.compareDate(p1.getFecha())).collect(Collectors.toList());

**

}

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

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    postOtrosUsuarios = this.postsSinUsuario(user);


    postOtrosUsuarios = this.postOrdenadosPorFecha(postOtrosUsuarios);

    List<Post> ultimosPosts = new ArrayList<Post>();

    ultimosPosts =
postOtrosUsuarios.stream().limit(cantidad).collect(Collectors.toList());

return ultimosPosts;
```

```
}
```

Por último, en el método postSinUsuario sigue el mal olor de "Reinventando la rueda". Lo soluciono con Replace Loop With pipeline.

```
public List<Post> postSinUsuario(Usuario user){
 return this.posts.stream().filter(p ->
!p.isSameUser(user)).collect(Collectors.toList());
}

public List<Post> postOrdenadosPorFecha(List<Post> users){
    // ordena los posts por fecha
    return users.stream()

     .sorted((p1, p2) ->
p2.compareDate(p1.getFecha())).collect(Collectors.toList());


}

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

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();
    postOtrosUsuarios = this.postsSinUsuario(user);


    postOtrosUsuarios = this.postOrdenadosPorFecha(postOtrosUsuarios);

    List<Post> ultimosPosts = new ArrayList<Post>();

    ultimosPosts =
postOtrosUsuarios.stream().limit(cantidad).collect(Collectors.toList());

return ultimosPosts;

}
```

2.4 Carrito de Compras




```
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();
    }
}
```

El primer problema que veo es la envidia de atributos en la clase Carrito, en el método total. A su vez, es una Especie de "Clase dios" debido a que la clase Carrito hace todo, y las demás clases no hacen nada. Utilizaría el método de refactor Move method, moviendo la lógica de calcular el precio del Item, al item

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


}


public class Carrito {

    private List<ItemCarrito> items;
```

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

2.5 Envío de Pedidos


```
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.getLocalidad() + “, ” +

this.direccion.getCalle() + “, ” +

this.direccion.getNumero() + “, ” +

this.direccion.getDepartamento();

}

**
```

Lo primero que se puede ver es la envidia de atributos de parte de la clase Cliente a la clase Dirección. Para solucionarlo utilizo Move method, ya que necesito que la dirección se devuelva formateada por si sola

```
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.getDireccionFormateada();

}
```

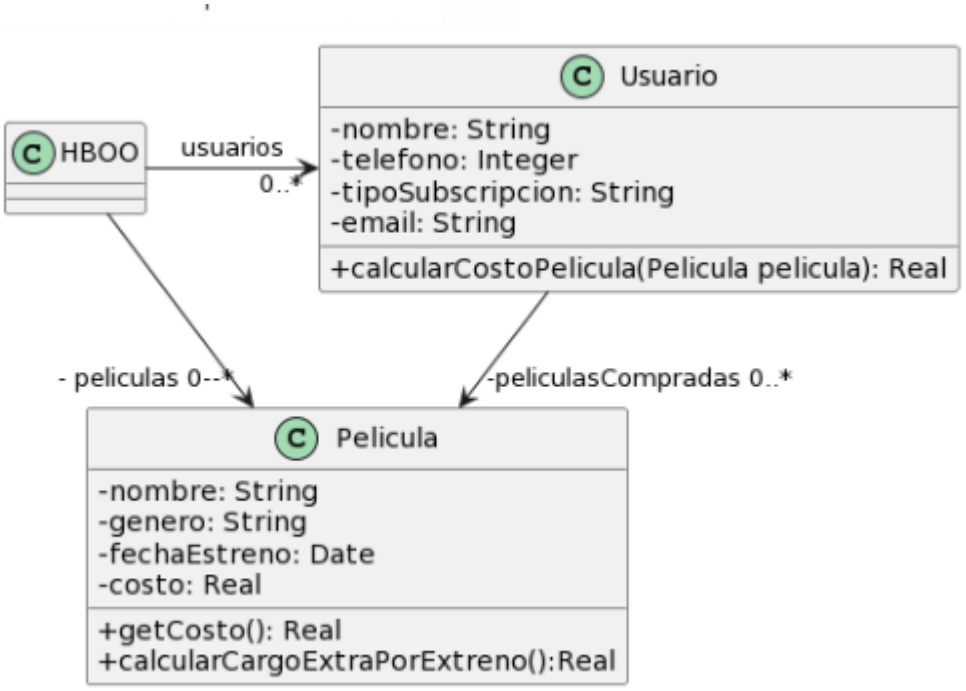
```
public class Direccion {
    private Direccion direccion;
    public String localidad;
    public String calle;
    public int numero;
    public String departamento;

public String getDireccionFormateada(){

    return this.localidad() + “, ” +

    this.calle() + “, ” +

    this.numero() + “, ” +

    this.departamento();

}
```

Segundo error a corregir: Se rompe el encapsulamiento de las variables de Direccion ya que están decladradas como públicas. Utilizaría Encapsulate FIeld

```
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.getDireccionFormateada();

}

public class Direccion {
```

```
    private String localidad;
    private String calle;
    private int numero;
    private String departamento;

public String getDireccionFormateada(){

    return this.localidad() + “, ” +

    this.calle() + “, ” +

    this.numero() + “, ” +

    this.departamento();

}
```

2.6 Películas




```
public class Usuario {

    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;
```

```
    }
}
```

El primer mal olor detectado en este diseño es la obsesión por los primitivos. En el caso de el atributo tipoSubscripcion en la clase Usuario. Se debe utilizar el método de refactor Extract class y rename variable.

```
public class Usuario {
    Subscripcion subscripcion;
    // ...

    public void setTipoSubscripcion(String unTipo) {
    this.tipoSubscripcion = unTipo;
    }
    public double calcularCostoPelicula(Pelicula pelicula) {
    double costo = 0;
    if (subscripcion.getTipo() =="Basico") {
    costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
    }
    else if (subscripcion.getTipo() == "Familia") {
    costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) * 0.90;
    }
    else if (subscripcion.getTipo() =="Plus") {
    costo = pelicula.getCosto();
    }
    else if (subscripcion.getTipo()=="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;

    }

}

public class Subscripcion {

private String tipo;

public String getTipo(){
    return this.tipo;
}
}
```

El mal olor a solucionar es: Envidia de atributos. Utilizaré Move method para mover la lógica del cálculo de la subscripción

```
public class Usuario {

    Subscripcion subscripcion;

    // ...
```

```
    public void setTipoSubscripcion(Subscripcion unTipo) {

    this.subscripcion = unTipo;

    }

    public double calcularCostoPelicula(Pelicula pelicula) {

    double costo = 0;

    costo = this.subscripcion.getCosto(pelicula)

    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;

    }

}

public class Subscripcion {

private String tipo;

public String getTipo(){
    return this.tipo;
}

public double getCosto(Pelicula pelicula){
    double costo = 0;
    if (this.tipo == "Basico") {
```

```
        costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();

        }

        else if (this.tipo == "Familia") {

        costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) *
0.90;

        }

        else if (this.tipo =="Plus") {

        costo = pelicula.getCosto();

        }

        else if (subscripcion.getTipo()=="Premium") {

        costo = pelicula.getCosto() * 0.75;

        }
    return costo;
    }
```

Los siguientes malos olores a solucionar es: switch statement. Se soluciona aplicando los siguientes métodos: Extract superclass y Replace conditional with polymorphism , para aprovechar el polimorfismo y que cada tipo de subscripción retorne lo que necesite.

```
public class Usuario {

    Subscripcion subscripcion;

    // ...


    public void setTipoSubscripcion(Subscripcion unTipo) {

    this.subscripcion = unTipo;

    }

    public double calcularCostoPelicula(Pelicula pelicula) {

    double costo = 0;

    this.subscripcion.getCosto(pelicula)

    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;

    }

}

public interface class Subscripcion {

    public double getCosto(Pelicula pelicula);
}

public class Basico implements Subscripcion {

    public double getCosto(Pelicula pelicula){
        return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
    }

}

public class Familia implements Subscripcion {

    public double getCosto(Pelicula pelicula){
        return (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno())*
0.90;
    }

}

public class Plus implements Subscripcion {

    public double getCosto(Pelicula pelicula){
        return pelicula.getCosto() ;
    }
```

```
}

public class Premium implements Subscripcion {

    public double getCosto(Pelicula pelicula){
        return pelicula.getCosto() * 0.75 ;
    }

}
```

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

    }
```

Tareas:

1. Enumere los code smell y que refactorings utilizará para solucionarlos.
2. Aplique los refactorings encontrados, mostrando el código refactorizado luego de aplicar cada uno.
3. Analice el código original y detecte si existe un problema al calcular las estadísticas. Explique cuál es el error y en qué casos se da ¿El error identificado sigue presente luego de realizar los refactorings? En caso de que no esté presente, ¿en qué momento se resolvió? De acuerdo a lo visto en la teoría, ¿podemos considerar esto un refactoring?

1er code smell, que no se rompe el encapsulamiento de la variable words, ya que si no se especifica si es pública o privada, cualquier clase dentro del mismo "paquete" podría acceder a ella sin problema. Utilizaría encapsulate Field
Opcional a consultar:
Se podrían remover las 2 variables temporales. Utilizando Replace Temp with query

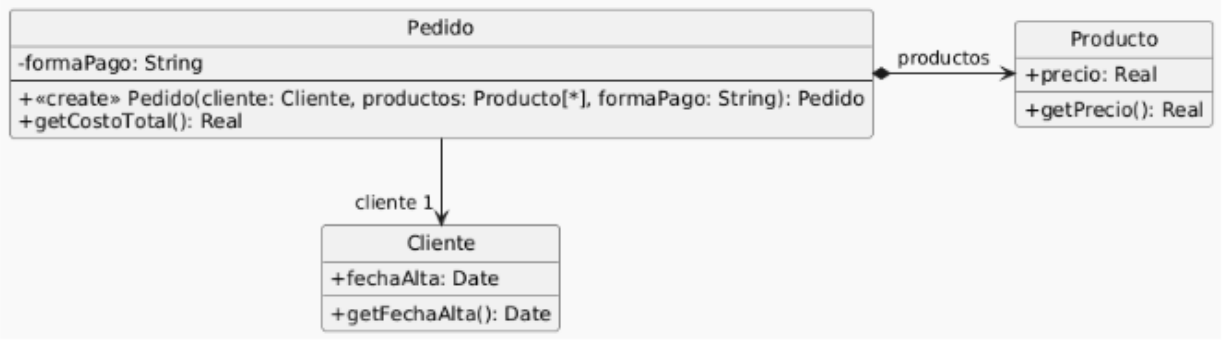
```
**

public class Document {

        private 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

     }
```

El error encontrado es el siguiente. La función calculate average no contempla que la lista no esté vacía, entonces si la lista está vacía, será dividir por 0, entonces el código levantará una excepción. Este problema sólo ocurrirá cuando la lista esté vacía. Con o sin los refactorings ocurrirá, ya que el refactoring,

ya que el refactoring no afecta funcionalidad. Lo que se debería hacer es utilizar el método average de los streams con un orElse(0) para en aquellos casos que la lista esté vacía, devuelva 0.

Ejercicio 4

Se tiene el siguiente modelo de un sistema de pedidos y la correspondiente implementación.




```
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.

1- Paso 1:

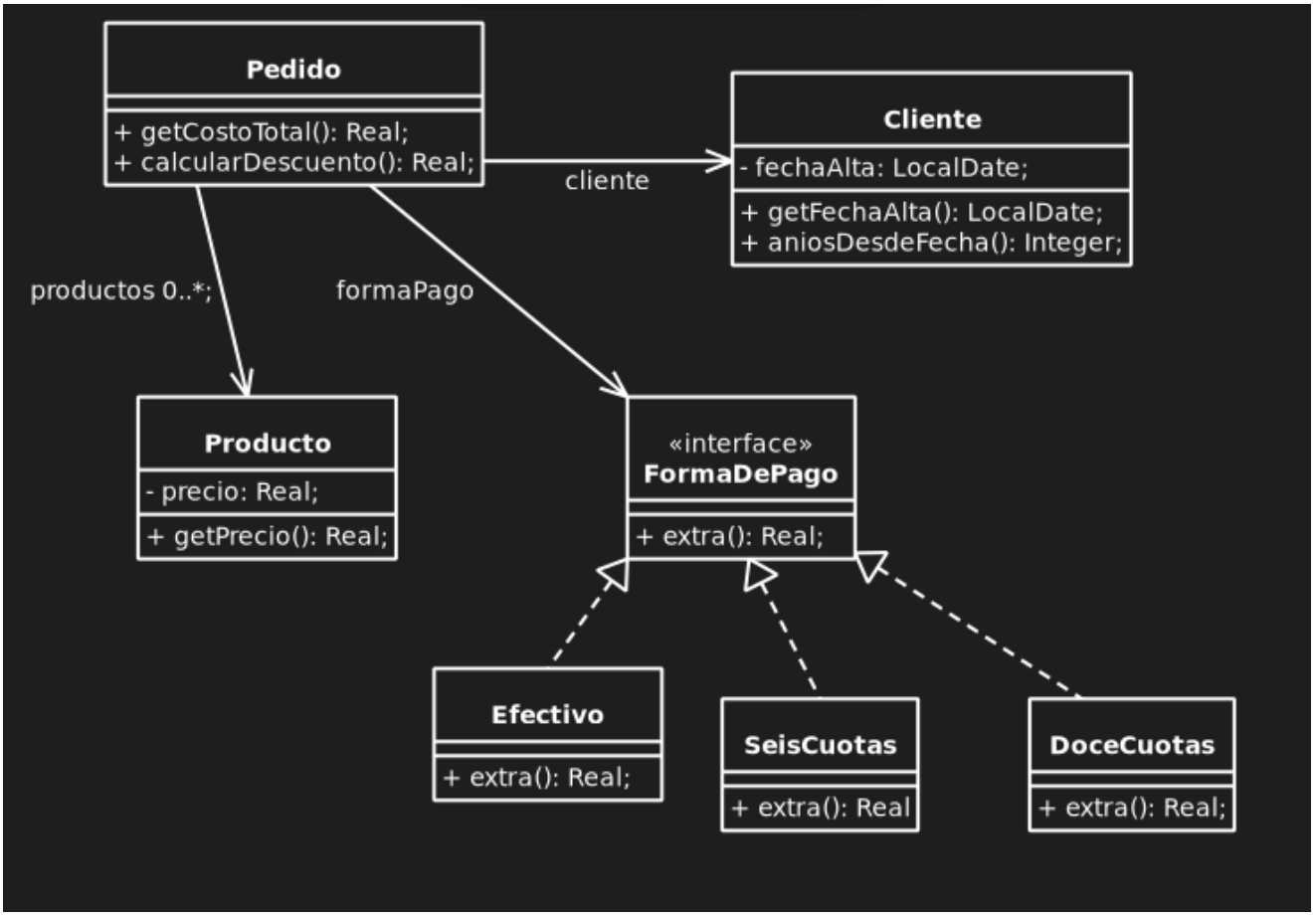
```
01: public class Pedido {
02:  private Cliente cliente;
03:  private List<Producto> productos;
04:  private String formaPago;
05:  public Pedido(Cliente cliente, List<Producto> productos, String formaPago) {
06:     if (!"efectivo".equals(formaPago)
07:        && !"6 cuotas".equals(formaPago)
08:        && !"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 = this.productos.stream()mapToDouble(p ->
17:p.getPrecio()).sum();
18:
19:     double extraFormaPago = 0;
20:     if ("efectivo".equals(this.formaPago)) {
21:       extraFormaPago = 0;
22:     } else if ("6 cuotas".equals(this.formaPago)) {
23:       extraFormaPago = costoProductos * 0.2;
24:     } else if ("12 cuotas".equals(this.formaPago)) {
25:       extraFormaPago = costoProductos * 0.5;
26:     }
27:     int añosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(),
LocalDate.now()).getYears();
30:     // Aplicar descuento del 10% si el cliente tiene más de 5 años de antiguedad
31:     if (añosDesdeFechaAlta > 5) {
32:       return (costoProductos + extraFormaPago) * 0.9;
33:     }
```

```
34:     return costoProductos + extraFormaPago;
35:   }
36: }
37: public class Cliente {
38:   private LocalDate fechaAlta;
39:   public LocalDate getFechaAlta() {
40:     return this.fechaAlta;
41:   }
42: }
43: public class Producto {
44:   private double precio;
45:   public double getPrecio() {
46:     return this.precio;
47:   }
48: }
```

Pasos siguientes:

```
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)
{
06:
07:
08:
09:
10:
11:    this.cliente = cliente;
12:    this.productos = productos;
13:    this.formaPago = formaPago;
14:   }
15:   public double getCostoTotal() {
16:     double costoProductos = this.productos.stream()mapToDouble(p ->
17:p.getPrecio()).sum();
18:
19:     double extraFormaPago = this.formaDePago.extra(costoProductos);
27:
30:     // Aplicar descuento del 10% si el cliente tiene más de 5 años de antiguedad
31:     return calcularDescuento(costoProductos, extraFormaPago)
35:   }
    public calcularDescuento(Double costoProductos, Double extraFormaPago){

        if (this.cliente.añosDesdeFecha() > 5) {
           return (costoProductos + extraFormaPago) * 0.9;
    }
     return costoProductos + extraFormaPago;
     }
36: }
37: public class Cliente {
38:   private LocalDate fechaAlta;
39:   public LocalDate getFechaAlta() {
40:     return this.fechaAlta;
41:   }
```

```
    public int aniosDesdeFecha(){
    return Period.between(this.fechaAlta, LocalDate.now()).getYears();
     }
42: }
43: public class Producto {
44:   private double precio;
45:   public double getPrecio() {
46:     return this.precio;
47:   }
48: }
49:public Interface FormaDePago {
50:   public double extra(Double costoProducto);
51: }
52: public class Efectivo implements FormaDePago{
53:    public double extra(Double costoProducto){
54:    return 0;
55:    }
56:   }
57: public class SeisCuotas implements FormaDePago{
58:
59:     public double extra(Double costoProducto){
60:     return costoProducto * 0.2;
61:    }
62:   }
63: public class DoceCuotas implements FormaDePago{
64:
65:     public double extra(Double costoProducto){
66:     return costoProducto * 0.5;
67:    }
68:   }
```



Patrones de diseño

Ejercicio 16

1. Evaluación del patrón que mejor describe el diseño de los filtros:

El patrón que más se ajusta al diseño implementado para los filtros es **Strategy**.

¿El objetivo del patrón se distingue en el diseño?

Sí, el objetivo del patrón *Strategy* se evidencia claramente en el diseño del sistema. El objetivo de este patrón es definir una familia de algoritmos (en este caso, filtros de imágenes), encapsular cada uno de ellos y hacerlos intercambiables. Esto permite que el algoritmo varíe independientemente de quienes lo utilizan. En la implementación provista, la clase `PNGFilterLauncher` aplica una secuencia de estrategias (filtros), todas con una interfaz común: el método `filter(BufferedImage)` definido en la clase abstracta `Filter`.

¿La estructura del proyecto coincide con la estructura y los participantes del patrón?

Sí, la estructura coincide. La clase `Filter` actúa como la interfaz común para todas las estrategias (filtros), y las clases concretas como `Rainbow`, `Artifacter`, `RGBShifter`, etc., son implementaciones específicas de dicha estrategia. Por otro lado, la clase `PNGFilterLauncher` actúa como el *contexto*, ya que es quien conoce y utiliza las distintas estrategias aplicándolas en secuencia sobre una imagen de entrada. Además, la selección de las estrategias activas se realiza en tiempo de ejecución a partir de los argumentos recibidos, lo que refuerza aún más la adecuación del patrón Strategy.

¿Se puede distinguir un "code smell" o algo que se aleje del patrón según el libro?

Sí, aunque el diseño se ajusta en gran medida al patrón *Strategy*, hay un aspecto mejorable. El uso de una colección `Map<String, Filter>` dentro de `PNGFilterLauncher` para registrar los filtros disponibles es válido, pero se realiza de forma muy rígida y centralizada. Cada nuevo filtro requiere modificar directamente la función `initializeFilters`, lo que rompe el principio de *abierto/cerrado* (Open/Closed Principle de los SOLID). Una mejora sería utilizar un mecanismo de registro dinámico o carga por reflexión que permita añadir nuevos filtros sin modificar esa clase. Además, `PNGFilterLauncher` conoce demasiados detalles sobre los filtros concretos, lo que acopla fuertemente las clases.

Si bien también hay un indicio superficial de un comportamiento tipo *Decorator* —ya que los filtros se encadenan y modifican la imagen original paso a paso—, no se trata de una composición recursiva de objetos del mismo tipo, como define el patrón *Decorator*. Es decir, los filtros no decoran a otros filtros, sino que todos reciben y devuelven una imagen, sin conocerse entre sí. Tampoco encaja con *Template Method*, ya que no hay una clase base con una estructura de algoritmo fija que delegue pasos variables a subclases.

Ejercicio 20

Una empresa de videojuegos ofrece personajes para sus juegos de rol. Cada personaje tiene un nombre y viene equipado con armaduras, armas, y habilidades únicas que les permiten desempeñarse mejor.

Las armaduras pueden ser de cuero, hierro o acero. Las armas pueden ser espadas, arcos o bastones de mago. Por último, las habilidades pueden ser de combate cuerpo a cuerpo, de combate a distancia, de curación o de magia.

Actualmente ofrece tres configuraciones estándar de personajes que pueden seleccionarse al inicio del juego: guerreros, arqueros y magos, cada uno con una combinación característica de armadura, arma y habilidades.

Los magos son expertos en el uso de la magia. Están equipados con una armadura de cuero para permitir la máxima movilidad. Su arma es un bastón mágico y sus habilidades son la magia y el combate a distancia. Los guerreros son los expertos en combate cuerpo a cuerpo, por lo tanto requieren una armadura de acero y una espada. Finalmente, los arqueros son especialistas en disparos de flechas. Cómo deben moverse rápidamente, tienen una armadura de cuero y están equipados con arcos.

En el juego, los personajes tienen la posibilidad de enfrentarse entre sí. Durante un enfrentamiento, el resultado dependerá del arma que utilice el atacante y de la armadura que lleve el defensor. A continuación, se muestra una tabla con el daño que cada arma causa. Ese valor afecta el puntaje del jugador atacado.

	Armadura de cuero	**ArmaduraDeHierro**	**ArmaduraDeAcero**
Espada	8	5	3
Arco	5	3	2
Bastón	2	1	1

Al inicio del juego, cada personaje comienza con 100 puntos de vida, los cuales se reducirán a medida que se enfrenten a otros jugadores. Un personaje podrá participar de un combate siempre que tenga vida.

Refactoring To Patterns

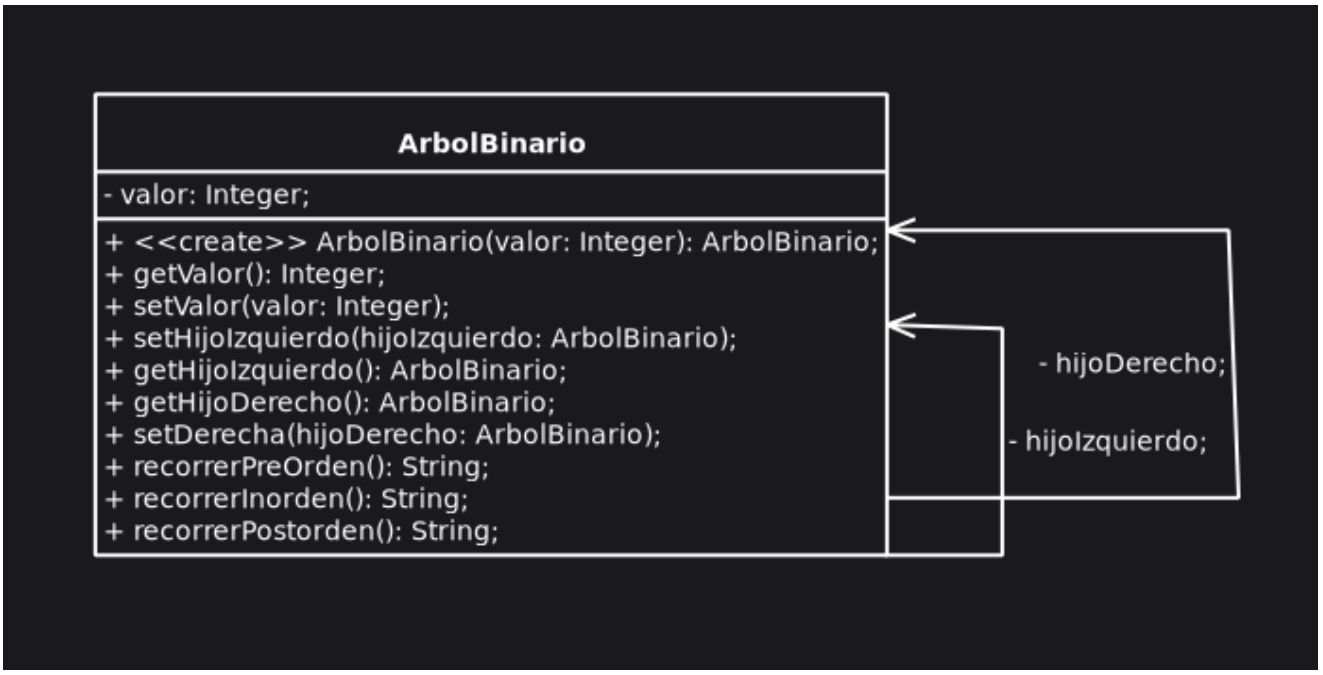
Ejercicio 6

Un árbol binario es una estructura de datos en la que cada nodo puede tener como máximo dos hijos: uno izquierdo y uno derecho. Es común utilizar esta estructura para representar jerarquías o realizar operaciones de búsqueda, recorrido y ordenamiento.

El código correspondiente a la implementación de un árbol binario se encuentra en el material adicional. En varias partes del código se realizan verificaciones repetitivas para comprobar si el árbol tiene hijo izquierdo o derecho antes de realizar los recorridos.

Tareas:

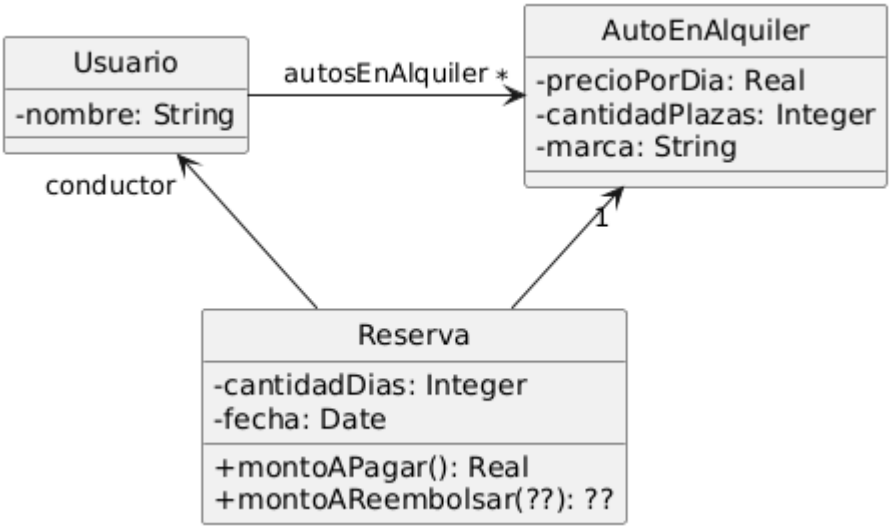
1. Describa la solución inicial con un diagrama de clases UML.
2. Refactorice la implementación para evitar estas verificaciones repetidas, explicando los pasos realizados.
3. Describa la solución final con un diagrama de clases UML.



Repaso

Ejercicio 9 Patrones

**En un sistema de alquiler de automóviles se quiere introducir funcionalidad para calcular el monto que será reembolsado (devuelto) si se cancela una reserva. Dicho reembolso podrá variar con respecto al monto total pagado, de acuerdo a la política de cancelación que sea determinada para el vehículo.



Se parte del siguiente diseño al que se necesita agregar la funcionalidad antes mencionada. El monto a pagar por una reserva se calcula como el precio por día del auto del cual se hizo la reserva, multiplicado por la cantidad de días.

Cada automóvil debe tener una política de cancelación que puede ser una de tres: flexible, moderada o estricta. Dichas políticas pueden cambiar con el tiempo en cualquier momento.

Se quiere calcular el monto a reembolsar de una reserva si se hiciera una cancelación. Dada una fecha tentativa de cancelación, se debe devolver el monto que sería reembolsado. El cálculo se hace de la siguiente manera.

1. Si el automóvil tiene política de cancelación flexible, se reembolsará el monto total sin importar la fecha de cancelación (que de todas maneras debe ser anterior a la fecha de inicio de la reserva).
2. Si el automóvil tiene política de cancelación moderada, se reembolsará el monto total si la cancelación se hace hasta una semana antes y 50% si se hace hasta 2 días antes.
3. Si el automóvil tiene política de cancelación estricta, no se reembolsará nada (0, cero) sin importar la fecha tentativa de cancelación.

Tareas:

1. Modifique el diagrama de clases UML para considerar los cambios necesarios. Indique el patrón de diseño utilizado y las ventajas de su uso en este diseño en particular.
2. Muestre los roles del patrón utilizado en el diseño realizado.
3. Implemente en Java
4. Muestre en un snippet de código Java cómo crear un automóvil con una política de cancelación flexible y luego imprima en pantalla el valor de reembolso. Luego, cambie la política a cancelación moderada y vuelva a imprimir en pantalla el valor de reembolso.**

Ejercicio 18: File Manager

En un File Manager se muestran los archivos. De los archivos se conoce:

- Nombre
- Extensión
- Tamaño
- Fecha de creación
- Fecha de modificación
- Permisos

Implemente la clase FileOO2, con las correspondientes variables de instancia y accessors.

En el File Manager el usuario debe poder elegir cómo se muestra un archivo (instancia de la clase FileOO2), es decir, cuáles de los aspectos mencionados anteriormente se muestran, y en qué orden. Esto quiere decir que un usuario podría querer ver los archivos de muchas maneras. Algunas de ellas son:

- nombre - extensión
- nombre - fecha de creación - extensión
- nombre - tamaño - permisos - extensión

Para esto, el objeto o los objetos que representen a los archivos en el FileManager debe(n) entender el mensaje prettyPrint(), retornando su nombre.

Es decir, un objeto cliente (digamos el FileManager) le enviará al objeto que Ud. determine, el mensaje prettyPrint(). De acuerdo a cómo el usuario lo haya configurado se deberá retornar un String con los aspectos seleccionados por el usuario en el orden especificado por éste. Considere que un mismo archivo podría verse de formas diferentes desde distintos puntos del sistema, y que el usuario podría cambiar la configuración del sistema (qué y en qué orden quiere ver) en runtime.

Tareas:

1. Discuta los requerimientos y diseñe una solución. Si aplica un patrón de diseño, indique cuál es y justifique su aplicabilidad.

2. Implemente en Java.
3. Instancie un objeto para cada uno de los ejemplos citados anteriormente y verifique escribiendo tests de unidad.

**

Ejercicio 18: File Manager

En un File Manager se muestran los archivos. De los archivos se conoce:

- Nombre
- Extensión
- Tamaño
- Fecha de creación
- Fecha de modificación
- Permisos

Implemente la clase FileOO2, con las correspondientes variables de instancia y accessors.

En el File Manager el usuario debe poder elegir cómo se muestra un archivo (instancia de la clase FileOO2), es decir, cuáles de los aspectos mencionados anteriormente se muestran, y en qué orden. Esto quiere decir que un usuario podría querer ver los archivos de muchas maneras. Algunas de ellas son:

- nombre - extensión
- nombre - fecha de creación - extensión
- nombre - tamaño - permisos - extensión

Para esto, el objeto o los objetos que representen a los archivos en el FileManager debe(n) entender el mensaje prettyPrint(), retornando su nombre.

Es decir, un objeto cliente (digamos el FileManager) le enviará al objeto que Ud. determine, el mensaje prettyPrint(). De acuerdo a cómo el usuario lo haya configurado se deberá retornar un String con los aspectos seleccionados por el usuario en el orden especificado por éste. Considere que un mismo archivo podría verse de formas diferentes desde distintos puntos del sistema, y que el usuario podría cambiar la configuración del sistema (qué y en qué orden quiere ver) en runtime.

Tareas:

1. Discuta los requerimientos y diseñe una solución. Si aplica un patrón de diseño, indique cuál es y justifique su aplicabilidad.
2. Implemente en Java.
3. Instancie un objeto para cada uno de los ejemplos citados anteriormente y verifique escribiendo tests de unidad.

**

Frameworks

Ejercicio 4

A partir del código compartido en teoría de tcp.server.reply.

1. Reimplementar el servidor PasswordServer empleando un enfoque basado en composición de objetos utilizando los componentes del framework tcp.server.reply vistos en teoría
2. Modifique el framework para que la condición de cierre de una conexión sea configurable con strings provistos por las instanciaciones.
3. Respecto a las dos formas vistas para implementar los servidores (PasswordServer ejercicio 1) iii) y 4) i) :

- ¿Qué debe hacer un desarrollador para extender el framework en cada una de las formas? Especifique qué clases debe subclasificar o implementar, qué métodos debe definir.
- ¿Cuánto conocimiento necesita tener el desarrollador sobre la estructura interna del framework para instanciarlo? ¿Y para extenderlo?
- ¿Qué técnica usarías si tuviera que ofrecer muchas configuraciones posibles para el servidor? ¿Por qué?
- Identifique los hotspots y frozen spots en cada una de las implementaciones.
- Considerando las dos formas de implementación del servidor PasswordServer, los programadores pueden asegurar que hay inversión de control? Justifique su respuesta identificando en qué parte se produce la inversión de control en cada uno de los casos.

Frameworks de caja blanca	**Frameworks De caja negra**
Para extender un framework de caja blanca, el desarrollador debe conocer la implementación interna del framework, incluyendo las clases base y los métodos que se pueden o deben sobrescribir. Generalmente se subclasifica una clase abstracta o base del framework, como por ejemplo `SingleThreadTCPServer` o una clase `Handler`. En el caso de el ejercicio 1, se debe subclasificar la clase base y definir el método `handleMessage` para procesar los mensajes recibidos. Además, si se quiere personalizar el comportamiento más a fondo, se pueden redefinir otros métodos protegidos o públicos que controle el flujo de la aplicación. Esto implica entender el ciclo de vida del servidor y cómo interactúan sus componentes internos.	Para extender un framework de caja negra, el desarrollador no necesita conocer la implementación interna, sino únicamente la interfaz pública y la funcionalidad que ofrece el framework. Se trabaja implementando un handler o componente externo que el framework utiliza para delegar ciertas responsabilidades (por ejemplo, el procesamiento de mensajes). El desarrollador debe conocer qué métodos debe implementar para cumplir con el contrato que el framework espera, pero sin modificar ni subclasificar la lógica interna del framework. Por ejemplo, implementar un `MessageHandler` que procese los mensajes, sin alterar cómo el servidor acepta conexiones o gestiona hilos internamente.
El FrozenSpot sería handleClient ya que es un método que no se puede modificar ni redefinir por subclases Los hotspots identificados son: - handleMessage(String message, PrintWriter out) - Métodos protegidos que pueden ser redefinidos para personalizar comportamiento: - `beforeComunication(Socket clientSocket, String inputLine)` (hook para acciones antes de procesar cada	Los hotsPot del framework serían los messages handlers unicamente, que es lo unico que se permite modificar Los FrozenSpot serían Condición de corte de la conección, un tipo de conexión (runtime), un tipo de MessageHandler (runtime)

Frameworks de caja blanca	Frameworks De caja negra
mensaje, como logging) - `onConnectionEnd(Socket clientSocket)` (hook para acciones cuando se cierra la conexión con el cliente) - `shouldFinish(String input)` (permite cambiar la condición para terminar la conexión) - `getTerminationWord()` (palabra que indica cierre de comunicación, se puede redefinir)	