Refactoring

1

1.1

```
/**
* Retorna el li´mite de cre´dito del cliente
*/
protected 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
*/
protected double mtCbE(LocalDate f1, LocalDate f2) {...
```

Se podría mejorar el protocolo de la clase Cliente, modificando los nombres de sus métodos, haciéndolos más expresivos, ya que no comunican lo que hace el método:

lmCrdt() => getLimiteDeCredito()

mtFcE() => getMontoFacturadoEntre()

mtCbE() => getMontoCobradoEntre()

1.2

Diseño inicial:

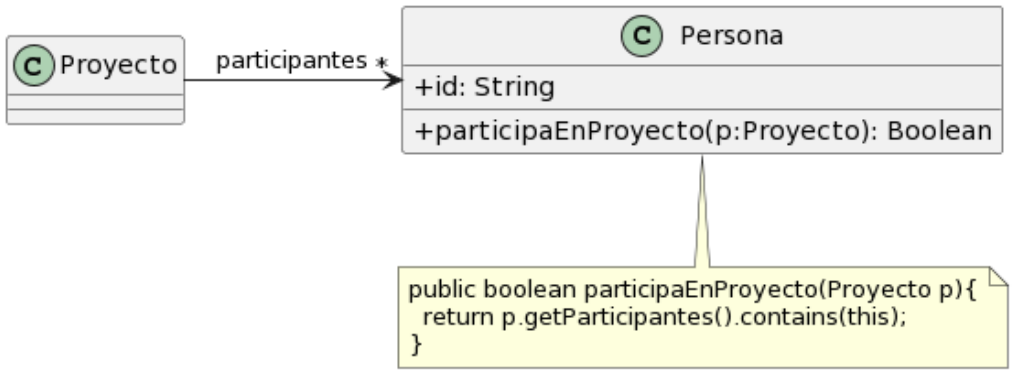


Figura 1: Diagrama de clases del diseño inicial.

Diseño revisado:

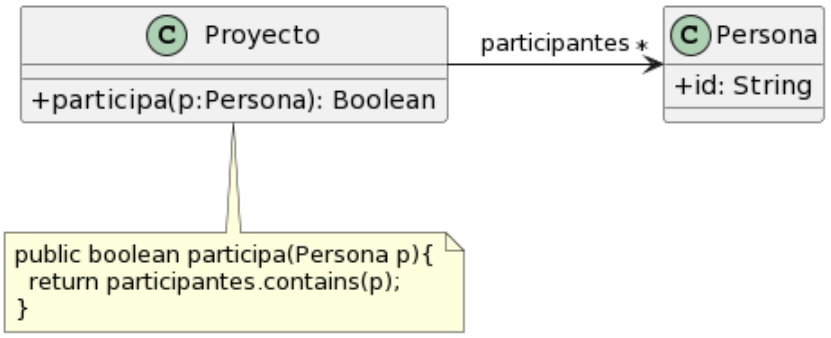


Figura 2: Diagrama de clases modificado.

Se aplicó el refactoring move method. Se movió el método Persona>>participaEnProyecto(p: Proyecto) a Proyecto>>participa(p: Persona)

Es apropiado ya que el método Persona>>participaEnProyecto(p: Proyecto) presentaba envidia de atributos, al acceder a los participantes del proyecto.

Además, la clase Proyecto era una clase de datos.

Al realizar el refactoring, se elimina la envidia de atributos y la clase Proyecto deja de ser una clase de datos.

Desventajas: la clase Persona pasa a ser una clase de datos.

1.3

```
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);

}
```

El método imprimirValores() no tiene un nombre descriptivo. Podría renombrarse a imprimirPromedioEdadesYTotalSalarios().

Además, el método imprimirValores() es un método largo. Para solucionar este code smell, se podría extraer varios métodos:

```
public void imprimirPromedioEdadesYTotalSalarios() {
```

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

   System.out.println(message);
}

public double getPromedioEdades(){
   int totalEdades = 0;

   for (Empleado empleado : personal) {
         totalEdades = totalEdades + empleado.getEdad();
   }
   return totalEdades / (double) personal.size();
}

public double getTotalSalarios(){
   double totalSalarios = 0;

   for (Empleado empleado : personal) {
         totalSalarios = totalSalarios + empleado.getSalario();
   }

   return totalSalarios;
}
```

A su vez, se podría querer separar la impresión del promedio de edades de la impresión del total de salarios, o aplicar el refactoring replace temp with query al método getPromedioEdades() para separar la lógica necesaria para obtener el la suma de las edades, de la necesaria para calcular el promedio de las edades.

Pero no lo voy a hacer je.

2

2.1

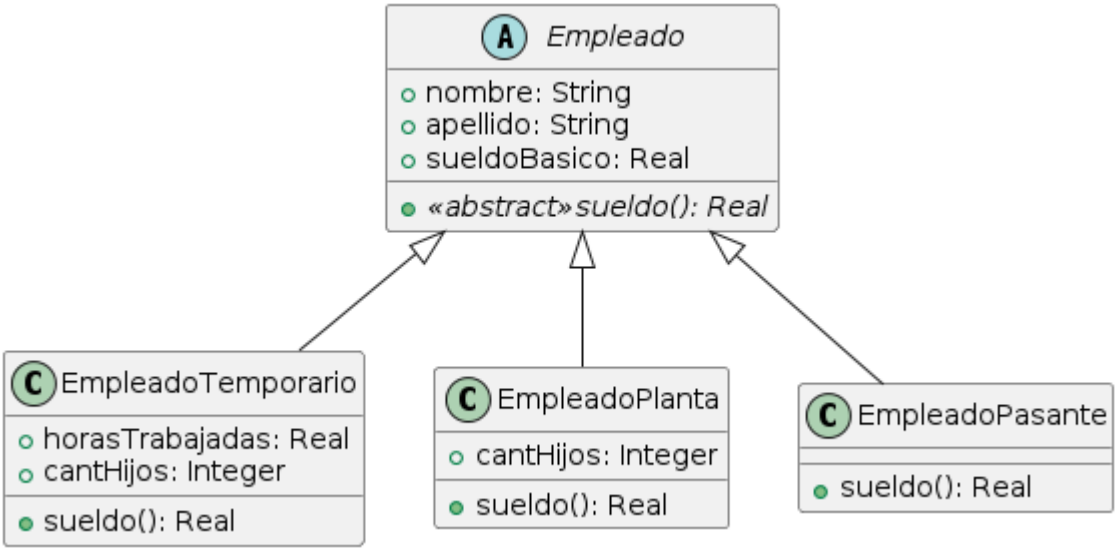
```
public class EmpleadoTemporario {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    public double horasTrabajadas = 0;
    public int cantidadHijos = 0;
    // ......

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);
    }
}
```

I. Mal olor: código duplicado (variables de instancia nombre, apellido y sueldoBasico y método sueldo())

II. Refactoring para corregirlo: Extract Superclass

III.

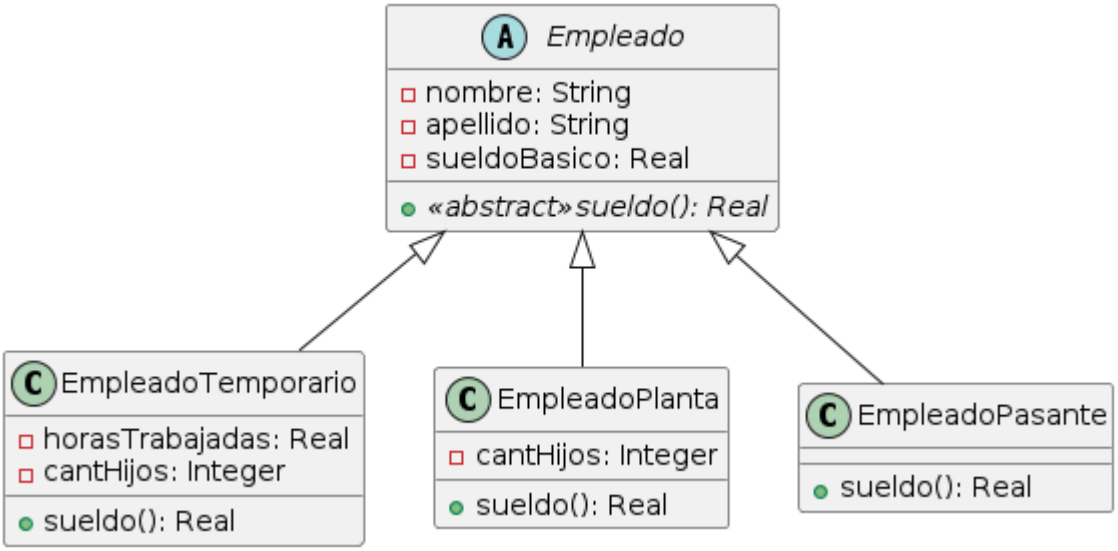



```
public abstract class Empleado {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    // ......

    public abstract double sueldo();
}
```

```
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);
    }
}
```

I. Mal olor: variables de instancia públicas
II. Refactoring: Encapsulate Field
III.




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

    public abstract double sueldo();
    public abstract double sueldo();
    public abstract double sueldo();

    public double nombre() {
        return this.nombre;
    }

    public double apellido() {
        return this.apellido;
    }

    public double sueldoBasico() {
        return this.sueldoBasico;
    }

    // ......

    public abstract double sueldo();
}
```

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

    public double horasTrabajadas() {
         return this.horasTrabajadas;
    }

    public double cantHijos() {
         return this.cantHijos;
    }

    // ......

      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 cantHijos() {
         return this.cantHijos;
    }

    // ......

    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);
    }
}
```

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;
}
```

I. Mal olor: envidia de atributos
II. Refactoring: Move Method
III.

```
public class Juego {
    // ......
}
```

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

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

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

I. Mal olor: variables de instancia públicas
II. Refactoring: Encapsulate Field
III.

```
public class Juego {
    // ......
}

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

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

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

I. Mal olor: nombre de método poco descriptivo (incrementar() y decrementar() hago 2x1)
II. Refactoring: Rename Method
III.

```
public class Juego {
    // ......
}

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

    public void sumar100Puntos() {
        this.puntuacion = this.puntuacion + 100;
    }

    public void restar50Puntos() {
```

```
        this.puntuacion = this.puntuacion - 50;
    }
}
```

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;
}
```

Muy bueno el refactoring y lo que quieras, pero es más fácil:

```
return this.posts.stream()
    .filter(post -> post.isFromUser(user))
    .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
    .limit(cantidad)
    .collect(Collectors.toList())
```

y chau

I. Mal olor: nombre poco descriptivo
II. Refactoring: Rename Method
III. ultimosPosts => ultimosPostsDeOtrosUsuarios

I. Mal olor: método largo
II. Refactoring: Extract Method
III.

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


/**
```

```
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(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).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;
}
```

I. Mal olor: método largo
II. Refactoring: Extract Method
III.

```
private List<Post> todosLosPostsDeOtrosUsuarios(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 ordenarPostsPorFechaDesc(List<Post> posts) {
   for (int i = 0; i < posts.size(); i++) {
       int masNuevo = i;
       for(int j= i +1; j < posts.size(); j++) {
           if (posts.get(j).getFecha().isAfter(
              posts.get(masNuevo).getFecha())) {
              masNuevo = j;
           }
       }
      Post unPost = posts.set(i,posts.get(masNuevo));
      posts.set(masNuevo, unPost);
   }
}


/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(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;
}
```

I. Mal olor: método largo
II. Refactoring: Extract Method
III.

```
private List<Post> todosLosPostsDeOtrosUsuarios(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 ordenarPostsPorFechaDesc(List<Post> posts) {
    for (int i = 0; i < posts.size(); i++) {
        int masNuevo = i;
        for(int j= i +1; j < posts.size(); j++) {
            if (posts.get(j).getFecha().isAfter(
                posts.get(masNuevo).getFecha())) {
                masNuevo = j;
            }
        }
       Post unPost = posts.set(i,posts.get(masNuevo));
       posts.set(masNuevo, unPost);
    }
}

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



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

    return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
}
```

I. Mal olor: ya existe la funcionalidad de ordenar por fecha
II. Refactoring: Replace Loop with Pipeline
III.

```
private List<Post> todosLosPostsDeOtrosUsuarios(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 ordenarPostsPorFechaDesc(List<Post> posts) {
   return posts.stream()
         .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
         .collect(Collectors.toList())
}

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



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

    return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
```

```
}
```

I. Mal olor: ya existe la funcionalidad de filtrar colecciones
II. Refactoring: Replace Loop with Pipeline
III.

```
private List<Post> todosLosPostsDeOtrosUsuarios(Usuario user) {
   return this.posts
         .filter(post->!post.getUsuario().equals(user))
         .collect(Collectors.toList())
}

private void ordenarPostsPorFechaDesc(List<Post> posts) {
   return posts.stream()
         .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
         .collect(Collectors.toList())
}

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



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

    return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
}
```

I. Mal olor: ya existe la funcionalidad de limitar el tamaño de colecciones
II. Refactoring: Replace Loop with Pipeline
III.

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

private void ordenarPostsPorFechaDesc(List<Post> posts) {
   return posts.stream()
         .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
         .collect(Collectors.toList())
}

private List<Post> limitarCantidadDePosts(List<Post> posts, int cantidad) {
   return posts.stream().limit(cantidad)
         .collect(Collectors.toList())
}



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

    return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
}
```

I. Mal olor: el chequeo del usuario de un post lo tiene que hacer la clase Post (envidia de atributos)
II. Refactoring: Extract Method y después Move Method (2x1)
III.

Extract Method:

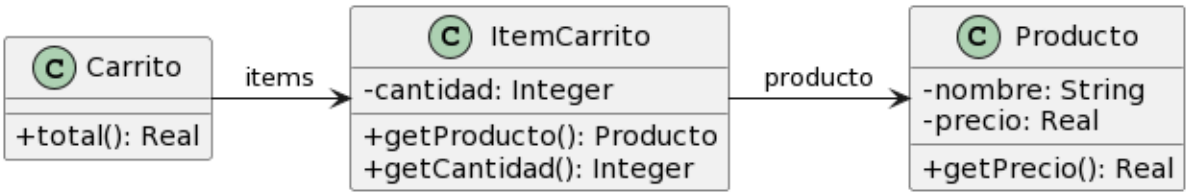
```
private List<Post> esPostDeOtroUsuario(Post post, Usuario user) {
          return !post.getUsuario().equals(user)
}

private List<Post> todosLosPostsDeOtrosUsuarios(Usuario user) {
    return this.posts.stream()
          .filter(post->esPostDeOtroUsuario(user, post))
          .collect(Collectors.toList())
}

private void ordenarPostsPorFechaDesc(List<Post> posts) {
    return posts.stream()
          .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
          .collect(Collectors.toList())
}

private List<Post> limitarCantidadDePosts(List<Post> posts, int cantidad) {
    return posts.stream().limit(cantidad)
          .collect(Collectors.toList())
}



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

     this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

     return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
}
```

Move Method:

```
class Post {
    // ..........

       private List<Post> esDeUsuario(Usuario user) {
                 return this.getUsuario().equals(user)
```

```
      }
}

private List<Post> todosLosPostsDeOtrosUsuarios(Usuario user) {
   return this.posts.stream()
         .filter(post->!post.esDeUsuario(user))
         .collect(Collectors.toList())
}

private void ordenarPostsPorFechaDesc(List<Post> posts) {
   return posts.stream()
         .sorted((emp1, emp2)->emp2.getFecha().compareTo(emp1.getFecha()))
         .collect(Collectors.toList())
}

private List<Post> limitarCantidadDePosts(List<Post> posts, int cantidad) {
   return posts.stream().limit(cantidad)
         .collect(Collectors.toList())
}



/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPostsDeOtrosUsuarios(Usuario user, int cantidad) {
   List<Post> postsOtrosUsuarios = this.todosLosPostsDeOtrosUsuarios(user);

    this.ordenarPostsPorFechaDesc(postsOtrosUsuarios);

    return this.limitarCantidadDePosts(postsOtrosUsuarios, cantidad);
}
```

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();
    }
}
```

I. Mal olor: el precio total de un ItemCarrito lo tiene que calcular el ItemCarrito (envidia de atributos)

II. Refactoring: Extract Method y después Move Method (2x1)

III.

Extract Method:

```
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;

   private double getPrecioTotalItemCarrito(ItemCarrito item) {
         return item.getProducto().getPrecio() * item.getCantidad();
   }


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

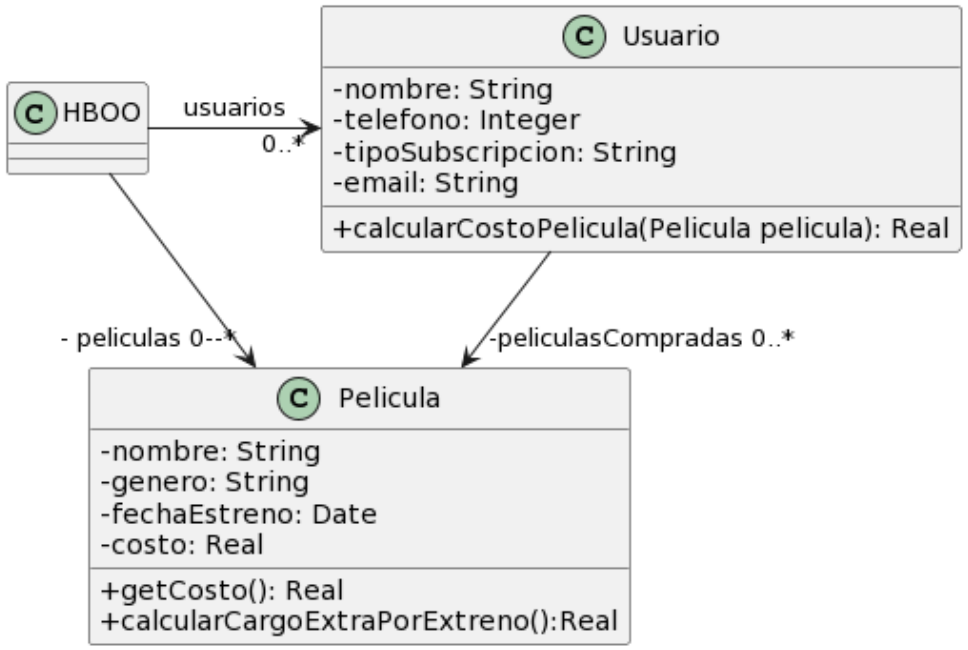
Move Method:

```
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;
    }

   private double getPrecioTotal() {
         return this.getProducto().getPrecio() * this.getCantidad();
   }

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

public class Carrito {
    private List<ItemCarrito> items;

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

I. Mal olor: nombre poco descriptivo de método (Carrito>>total())
II. Refactoring: Rename Method
III.

```
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;
    }

   private double getPrecioTotal() {
         return this.getProducto().getPrecio() * this.getCantidad();
   }

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

public class Carrito {
    private List<ItemCarrito> items;

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

2.6




```
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;
    }
}
```

I. Mal olor: existe lógica condicional en un método (calcularCostoPelicula) que controla qué variante ejecutar entre distintas posibles

II. Refactoring: Replace Conditional Logic with Strategy

   A. Crear una clase Strategy

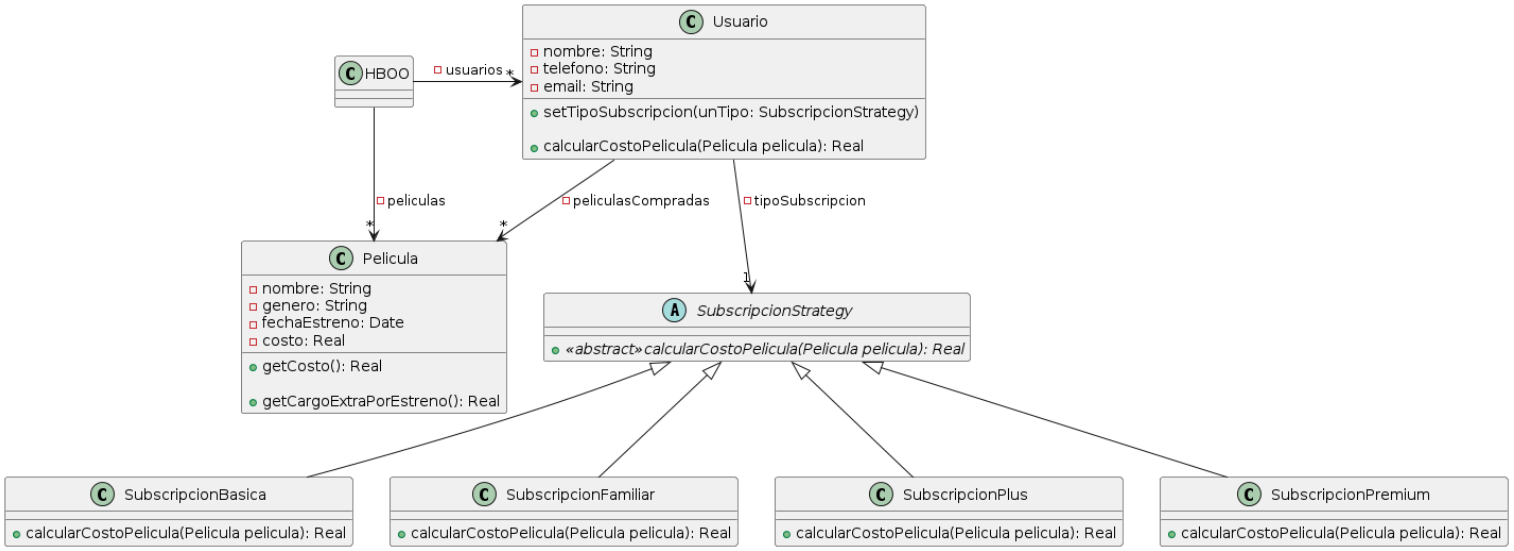
   B. Aplicar “Move Method” para mover el cálculo con los condicionales del contexto al strategy

      1. Definir una v.i. en el contexto para referenciar al strategy y un setter (generalmente el constructor del contexto)
      2. Dejar un método en el contexto que delegue
      3. Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables? Y en qué momento?)
      4. Compilar y testear

   C. Aplicar “Extract Parameter” en el código del contexto que inicializa un strategy concreto, para permitir a los clientes setear el strategy. - Compilar y testear

   D. Aplicar “Replace Conditional with Polymorphism” en el método del Strategy.

   E. Compilar y testear con distintas combinaciones de estrategias y contextos.



III.

```
public class SubscripcionStrategy {
	public abstract double calcularCostoPelicula(Pelicula pelicula);
}

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

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

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

public class SubscripcionPremium {
```

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

public class Usuario {
    SubscripcionStrategy tipoSubscripcion;
    // ...

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

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



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;
    }
}
```