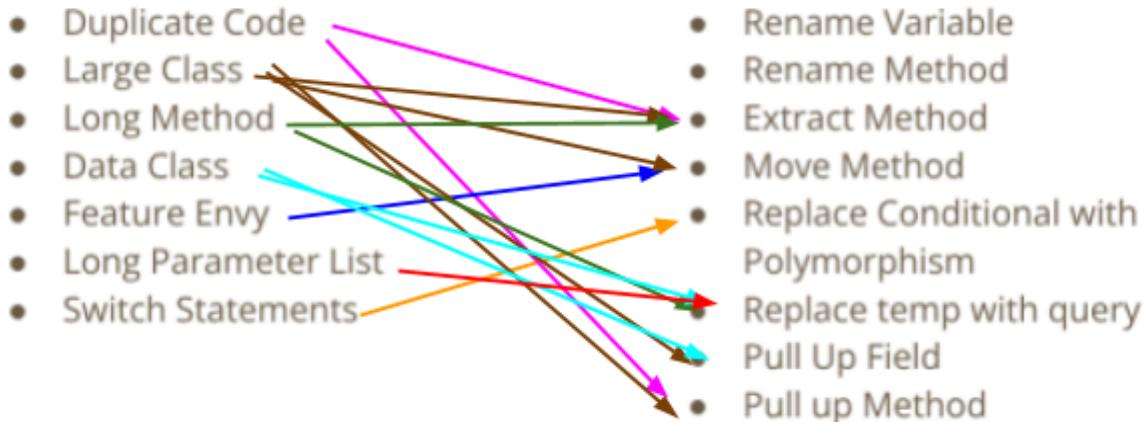


Resolución de Práctica de Refactoring

Ejercicio -1

Estudie en detalle los CodeSmells y los Refactoring del libro “Refactoring - Improving The Design of Existing Code” de Martin Fowler y una con flechas los code smells y los refactoring a los que se relacionan



Duplicate Code: a este code smell se le aplica el refactoring

- Extract Method
- Pull up Method

Large Class: a este code smell se le aplica el refactoring

- Extract Method
- Move Method
- Replace temp with query
- Pull Up Field (atributo)
- Pull up Method

Long Method: a este code smell se le aplica el refactoring

- Extract Method
- Replace Conditional with Polymorphism

Data Class: a este code smell se le aplica el refactoring

- Move Method

Feature Envy: a este code smell se le aplica el refactoring

- Move Method

Long Parameter List: a este code smell se le aplica el refactoring

- Replace temp with query

Switch Statements: a este code smell se le aplica el refactoring

- Replace Conditional with Polymorphism

Ejercicio 0

Estudie los refactorings listados del libro “Refactoring - Improving The Design of Existing Code” de Martin Fowler y liste el conjunto de precondiciones para cada uno:

• Rename Variable/field (pág. 14)	• Rename Method	• Extract Method	• Move Method
• Replace Conditional with Polymorphism	• Replace temp with query	• Pull Up Field	• Pull up Method

Refactoring	Precondiciones
Rename Variable/field	<ul style="list-style-type: none"> - Alcance (Scope): El nuevo nombre no debe estar ya en uso dentro del mismo ámbito (scope) o en un ámbito que confunda al compilador/intérprete (shadowing). - Referencias: Debes ser capaz de identificar y actualizar todas las referencias a la variable. Si es una variable pública o publicada en una librería, el cambio podría romper código cliente (en cuyo caso se debe dejar el viejo como <i>deprecated</i>).
Rename Method	<ul style="list-style-type: none"> - Polimorfismo: El nuevo nombre no debe existir ya en la clase, ni en ninguna de sus superclases o subclases. Si colisiona, podrías estar sobreescribiendo (overriding) accidentalmente un método o rompiendo una sobreescritura existente. - Interfaces: El método no debe estar implementando una interfaz de terceros que obligue a mantener ese nombre específico
Extract Method	<ul style="list-style-type: none"> - Variables Locales Modificadas: Esta es la precondición más crítica. Si el fragmento de código que quieras extraer modifica más de una variable local que se usa después del fragmento, el refactoring es complejo (o imposible limpiamente sin crear un objeto de retorno). - Flujo de Control: El código a extraer no debe contener saltos de control inusuales que entren o salgan del fragmento (como un break o return que afecte al método original de forma que no se pueda replicar en el sub-método).
Move Method	<ul style="list-style-type: none"> - Uso de características: El método debe usar (o planear usar) más características (datos o funciones) de la clase destino que de la clase origen. - Polimorfismo: El método no debe ser una sobreescritura (override) de un método en la superclase, ni estar siendo sobreescrito por una subclase (a menos que muevas toda la jerarquía). - Dependencias: Mover el método no debe crear una dependencia circular entre las dos clases.
Replace Conditional with Polymorphism	<ul style="list-style-type: none"> - Dependencia del Tipo: La lógica condicional debe basarse en el tipo de objeto (o un código de tipo/enum que represente el estado del objeto). - Jerarquía de Clases: Debe existir (o ser posible crear) una estructura de herencia o implementación de interfaz donde cada rama del condicional se corresponda con una subclase.
Replace temp with query	<ul style="list-style-type: none"> - Efectos Secundarios (Side Effects): La expresión que calcula el valor no debe tener efectos secundarios. Ejecutar la consulta varias veces debe ser seguro e inocuo.

	<ul style="list-style-type: none"> - Inmutabilidad: La variable temporal debe asignarse una sola vez (o ser fácil de refactorizar para que sea <code>final/const</code>). Si la variable cambia de valor varias veces durante el método, no puedes reemplazarla simplemente por una consulta.
Pull Up Field	<ul style="list-style-type: none"> - El campo debe estar duplicado en varias subclases con el mismo propósito. - La superclase debe ser el lugar lógico para contenerlo. - Debe llamarse igual en todos lados - si representa lo mismo pero no se llama igual, antes debo poner Rename Variable/field y después de ese refactoring aplicar pull up field, SON 2 PASOS DISTINTOS.
Pull up Method	<ul style="list-style-type: none"> - El método debe estar duplicado en varias subclases con lógica idéntica o muy similar. - No debe depender de miembros exclusivos de las subclases. - Si el método no se llama igual primero hago un Rename method, y otro paso es hacer un pull up method.

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?

Código con Bad Smell	Código con refactoring
<pre>/** * Retorna el límite de crédito 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) {...}</pre>	<p>Bad Smell/ Code Smell:</p> <ul style="list-style-type: none"> - Nombres poco descriptivos tanto en nombre de métodos como en parámetros, no reflejan con precisión la intención. <p>Refactoring:</p> <ul style="list-style-type: none"> - Rename method - Rename variable/field <pre>// Esto esta demas el enunciado solo pide Indicar no hacer protected double limiteCreditoCliente () {...} protected double montoFacturadoAlCliente(LocalDate fechalinicial, LocalDate fechaFinal){..} protected double montoCobradoAlCliente (LocalDate fechalinicial, LocalDate fechaFinal) {}</pre>

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:

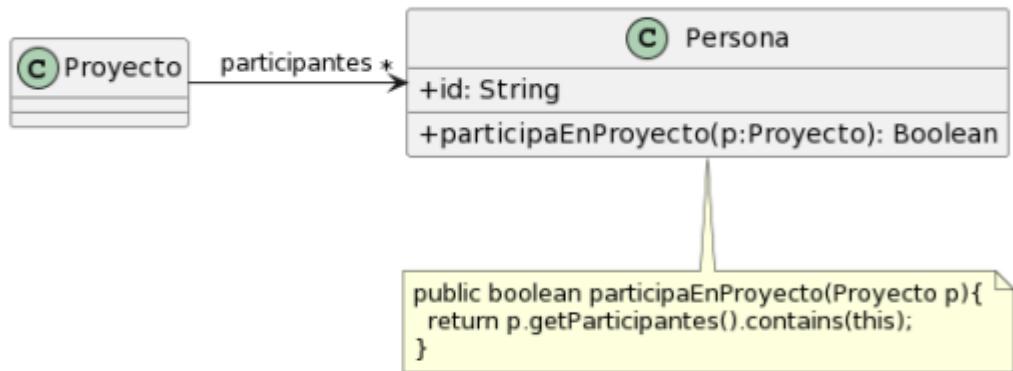


Figura 1: Diagrama de clases del diseño inicial.

Diseño Revisado:

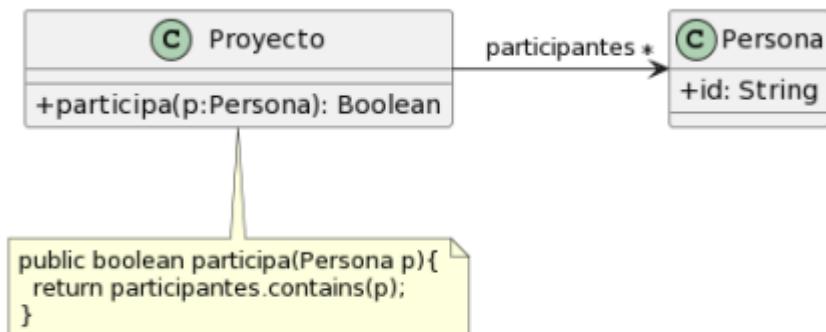


Figura 2: Diagrama de clases modificado.

El tipo de cambio que se hizo de la figura 1 a la figura 2 es el que le corresponde a el code smell envidia de atributos (Feature Envy). Se le aplicó el refactoring move method, que mueve el método “participaEnProyecto(..)” de la clase Persona a la clase Proyecto. Tiene sentido ya que el método trabaja con los datos que tiene la lista de participantes que contiene la clase Proyecto.

También se aplicó un rename method para el método que se movió, el cual mantiene mejor la claridad pero en mi opinión el nombre anterior era mejor porque era más descriptivo. Se cambio de “participaEnProyecto(..) → participa(..)”

Otro cambio que se hizo fue el parámetro. En la figura 1 se pasa como parámetro un proyecto y en la figura 2 se pasa a una persona.

Conclusión: se cambió el nombre del método, el parámetro y la clase donde implementarlo.

1.3 Cálculos

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

Código Code Smell	Código Refactorizado
<pre>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); }</pre>	<p>Code smell:</p> <ul style="list-style-type: none"> - Long method <p>Este método tiene demasiadas tareas. Empezaría con extraer cada tarea en un método.</p> <p>Refactoring:</p> <ul style="list-style-type: none"> - Extract method - Replace temp with query: llamó a los métodos y ese valor se lo asignó a la variable temporal que corresponda. <p>Resultado:</p> <pre>public integer totalEdades (){ int totalEdades = 0; for (Empleado empleado : personal) { totalEdades = totalEdades + empleado.getEdad(); } return totalEdades; } public double totalSalarios(){ double totalSalarios = 0; for (Empleado empleado : personal) { totalSalarios = totalSalarios + empleado.getSalario(); } return totalSalarios; } public double promedioEdades(int totalEdades){ double promedioEdades = totalEdades / personal.size(); return promedioEdades; } public void imprimirValores() { int totalEdades = totalEdades(); double totalSalarios = totalSalarios(); double promedioEdades = promedioEdades(totalEdades) String message = String.format("El promedio de las edades es %s y el total de salarios es %s", promedioEdades, totalSalarios); System.out.println(message); }</pre> <hr/> <p>Una vez testeados los cambios realizados en el paso anterior, sigo refactorizando.</p> <p>Code smell:</p> <ul style="list-style-type: none"> - Se reinventa la rueda usando un for para recorrer una

colección y no un stream().

Refactoring:

- replace loop with pipeline

Resultado:

```
public integer totalEdades (){
    int totalEdades = personal.stream()
        .mapToInt(Empleado::getEdad) .sum();

    return totalEdades;
}

public double totalSalarios(){
    double totalSalarios = personal.stream()
        .mapToDouble(Empleado::getSalario) .sum();

    return totalSalarios;
}

public double promedioEdades(int totalEdades){
    double promedioEdades = totalEdades / personal.size();
    return promedioEdades;
}

public void imprimirValores() {

    int totalEdades = totalEdades();
    double totalSalarios = totalSalarios();
    double promedioEdade = promedioEdades(totalEdades)

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

Una vez testeados los cambios realizados en el paso anterior, sigo refactorizando.

Code smell:

- Eliminar variables temporales de los métodos

Refactoring:

- *Replace Temp with Query*

Resultado:

```
public integer totalEdades (){
    return personal.stream()
        .mapToInt(Empleado::getEdad) .sum();
}

public double totalSalarios(){
    return personal.stream()
        .mapToDouble(Empleado::getSalario) .sum();
}

public double promedioEdades(int totalEdades){
```

```

        return totalEdades / personal.size();
    }

    public void imprimirValores() {

        int totalEdades = totalEdades();
        double totalSalarios = totalSalarios();
        double promedioEdade = promedioEdades(totalEdades)

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

```

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".

Code Smell	Refactorizado:
<p>Empleado</p> <pre> public class EmpleadoTemporal { 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; } </pre>	<p>Code smell:</p> <ul style="list-style-type: none"> - Tiene las v.i públicas en todas las clases. Se rompe el encapsulamiento. Las cambiaría a privadas. <p>Refactoring:</p> <ul style="list-style-type: none"> - Encapsulate field <p>Resultado:</p> <pre> public class EmpleadoTemporal { private String nombre; private String apellido; private double sueldoBasico = 0; 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 { private String nombre; private String apellido; private double sueldoBasico = 0; private int cantidadHijos = 0; } </pre>

```

// .....

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);
}

```

```

private String nombre;
private String apellido;
private double sueldoBasico = 0;
private int cantidadHijos = 0;
// .....

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

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

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

```

Testeo, y sigo refactorizando

Code Smell:

- Hay v.i repetidas en todas las clases. Haría una jerarquía y subiría las v.i repetidas y crearía el método calculoSueldo(). Todas las subclases extienden de esta nueva clase. Creo una clase abstracta Empleado.
- Cambio v.i de private a protected ya que voy a crear la clase abstracta con esas v.i. O es mejor hacer getters?

//NO ES UN PROBLEMA ESTE ULTIMO o sea no es un code smell.

Refactoring:

- Create Abstract Class, no es con este nombre → Extract Superclass en este paso poner que pusimos las variables protected.
- pull up field
- pull up method

// consulta ger: si hubiera comportamiento que justifique hacer una clase intermedia para int cantHijos se debe hacer. En este caso no lo vale porque no hay métodos o código repetido.

Resuelto:

```

public abstract class Empleado() {
    private String nombre;
    private String apellido;

```

private double sueldoBasico = 0; //ACA SON
PROTECTED NO?SI NO TENGO QUE AGREGAR
GETTER , la consulta esta en si EN ESTE CASO, elegir
la opcion de protected es correcta.

Indiferente elegir cualquier opción. No es un mal olor
poner de private a protected. Es uno de los pasos de pull
up field. Si tenía getter ya definido no hace falta este
paso.

```
protected String nombre;  
protected String apellido;  
protected double sueldoBasico = 0;  
  
public double calcularSueldo(){  
    return (this.sueldoBasico * 0.13);  
}  
  
}  
  
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.calcularSueldo());  
    }  
}  
  
public class EmpleadoPlanta extends Empleado {  
    public int cantidadHijos = 0;  
    // .....  
  
    public double sueldo() {  
        return this.sueldoBasico +  
            (this.cantidadHijos * 2000) - this.calcularSueldo();  
    }  
}  
  
public class EmpleadoPasante extends Empleado {  
    private double sueldoBasico = 0;  
    // .....  
  
    public double sueldo() {  
        return this.sueldoBasico -  
            (this.calcularSueldo());  
    }  
}
```

Code Smell	Refactoring
<pre> 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; } </pre>	<p>Code Smell:</p> <ul style="list-style-type: none"> - En la clase Jugador tiene las variables de instancias públicas. Se rompe el encapsulamiento. Ponerlas privadas. <p>Refactoring:</p> <ul style="list-style-type: none"> - Encapsulate field <p>Resuelto:</p> <pre> 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; } </pre> <hr/> <p>Testeo y sigo refactorizando</p> <p>Code smell:</p> <ul style="list-style-type: none"> - Envidia de atributos, la clase Juego tiene los métodos de incrementar y decrementar con lógica que debería estar en Jugador. Lo que se hace es crear 2 métodos en Jugador donde se va a mover esa lógica y en los métodos de Juego llaman a esos métodos. <p>Refactoring:</p> <ul style="list-style-type: none"> - Extract method, muevo lógica de método de Juego a Jugador - Rename method <pre> public class Juego { // public void incrementar(Jugador j) { j.incrementarPuntaje() } public void decrementar(Jugador j) { j.decrementarPuntaje() } </pre>

```

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

    public void incrementarPuntaje(){
        this.puntuacion +=100;
    }
    public void decrementarPuntaje(){
        this.puntuacion -=50;
    }
}

```

2.3 Publicaciones



Code Smell	Refactorizado
<pre> /** * Retorna los últimos N posts que no pertenezcan al usuario user */ public List ultimosPosts(Usuario user, int cantidad) { List postsOtrosUsuarios = new ArrayList(); 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(</pre>	<p>Code Smell: El método realiza muchas tareas, debería extraerse cada tarea a un método. Long method</p> <p>Refactoring:</p> <ul style="list-style-type: none"> - Extract method - Replace temp with query <p>Resolución:</p> <pre> public List<Post> postsOtrosUsuarios (Usuario user){ List<Post> postsOtrosUsuarios = new ArrayList<Post>(); for (Post post : this.posts) { if (!post.getUsuario().equals(user)) { postsOtrosUsuarios.add(post); } } // ordena los posts por fecha public void postOrdenadosPorFecha(){ for (int i = 0; i < postsOtrosUsuarios.size(); i++) { </pre>

```

postsOtrosUsuarios.get(masNuevo).getFecha()))
{ masNuevo = j; } }
Post unPost =
postsOtrosUsuarios.set(i,postsOtrosUsuarios.get(masNuevo));
postsOtrosUsuarios.set(masNuevo, unPost);
}
List ultimosPosts = new ArrayList();
int index = 0;
Iterator postIterator =
postsOtrosUsuarios.iterator();
while (postIterator.hasNext() && index < cantidad)
{ ultimosPosts.add(postIterator.next());
} return ultimosPosts
}

```

```

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);
}

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

postOrdenadosPorFecha(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;
}

```

Testeo, compilo y sigo refactorizando
Code Smell:

- se reinventa la rueda en el método “postsOtrosUsuarios()” y en “postOrdenadosPorFecha()” se debería usar un stream() para recorrer una colección no un for

Refactoring:

- Replace loop with pipeline

Resolución:

```
public List<Post> postsOtrosUsuarios(Usuario user) {
```

```

        return this.posts.stream()
            .filter(post -> !post.getUsuario().equals(user))
            .collect(Collectors.toList());
    }

    // ordena los posts por fecha
    public void postOrdenadosPorFecha() {
        postsOtrosUsuarios = postsOtrosUsuarios.stream()

            .sorted(Comparator.comparing(Post::getFecha).reversed()
        )
            .collect(Collectors.toList());
    }

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

        List<Post> postsOtrosUsuarios =
        postsOtrosUsuarios(user);

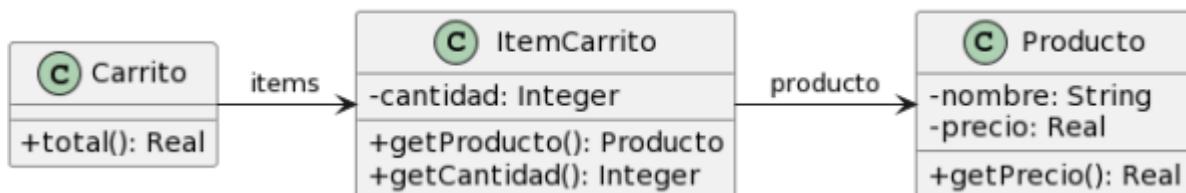
        postOrdenadosPorFecha(postsOtrosUsuarios)

        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;
    }

```

Testeo. Fin.

2.4 Carrito de compras



Code Smell	Refactorizado
------------	---------------

```

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 items;

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

```

Code Smell:

- El precio por Item deberá sacarse en la clase ItemCarrito. Debería tener un método para calcularlo y en carrito llamar a ese método.

Refactoring:

- Envidia de atributos

Resolución:

```

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;
    }

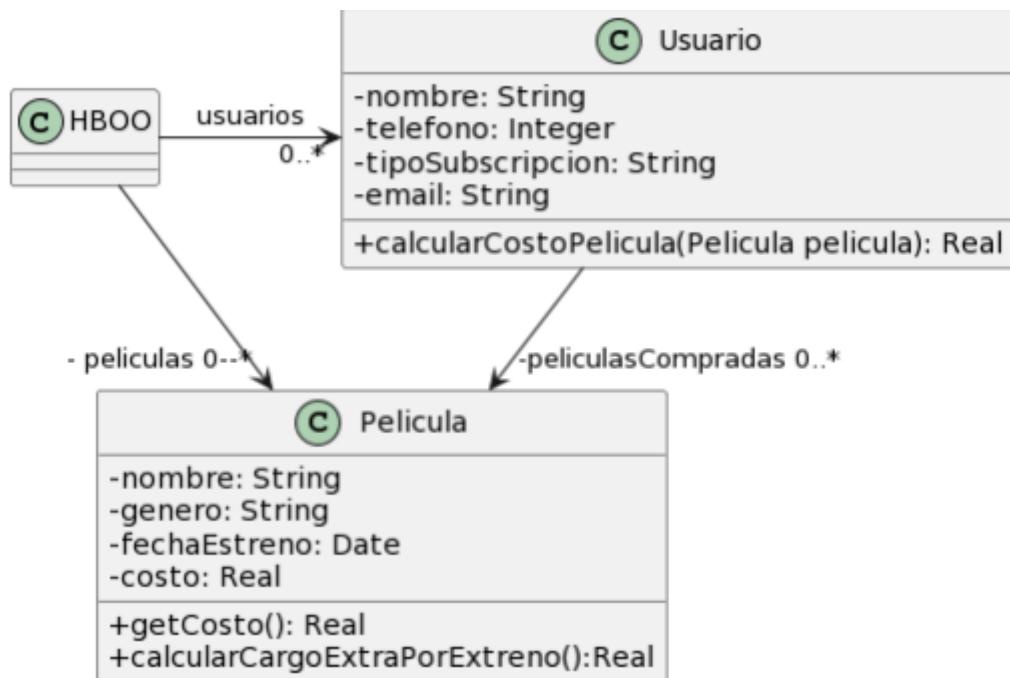
    public int precioItem(){
        return this.getProducto().getPrecio * cantidad;
    }
}

public class Carrito {
    private List items;

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

//Consultar ejercicio, queda asi? No veo otra cosa "mal
//la clase producto es una Data class ya que no aporta
//lógica, refactorizar."

```



Code Smell	Refactorizado
<pre> 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") { </pre>	<p>Code smell:</p> <ul style="list-style-type: none"> - Las v.i deberían ser privadas, para que no se rompa el encapsulamiento. <p>Refactoring</p> <ul style="list-style-type: none"> - Encapsulate field <p>Resolución:</p> <pre> 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") { </pre>

```

        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;
    }
}

```

```

}return costo;
}

public class Pelicula {
    private 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;
    }
}

```

Testeo y sigo refactorizando

Code Smell:

- Switch Statement

Refactoring:

- Replace conditional with Strategy

(elegí strategy y no polimorfismo porque hay un setTipoDeSuscripcion por lo tanto puede cambiar dinámicamente el tipo de suscripción) / Esta bien, fue consultado a Alejandra

Mecánica:

1. Crear una clase Strategy (Suscripción)
2. Aplicar “Move Method” para mover el cálculo con los condicionales del contexto al Strategy.
 - a. Definir una v.i en el contexto para referenciar al strategy (en este caso cambiar el tipo de la v.i **String tipoSuscripcion** por **Suscripcion tipoSuscripcion**) y un setter (constructor del contexto) (**setTipoSuscripcion (Suscripcion s)**)
 - b. Dejar un metodo en el contexto que delegue (**calcularCostoPelicula(..)**)
 - c. Elegir los parametros necesarios para pasar al Strategy (**calcularCostoPelicula(pelicula.calcularCargoExtraPorEstreno()), pelicula.getCosto()**)
 - d. Compilar y testear
3. Aplicar “Extract Parameter” en el código del contexto que inicializa un strategy concreto, para permitir a los clientes setear el strategy. Compilar y testear
4. Aplicar “Replace Condicional with Polymorphism”

en el método del Strategy. En este paso creo a las subclases no? Basico, Familia, Plus, Premium

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

Resultado

```
public class Usuario {  
    private Suscripcion tipoSubscripcion;  
  
    // ...  
    public void setTipoSubscripcion(String unTipo) {  
        this.tipoSubscripcion = unTipo;  
    }  
    public double calcularCostoPelicula(Pelicula pelicula) {  
  
        return  
this.tipoSubscripcion.calcularCostoPelicula(pelicula.calcul  
arCargoExtraPorEstreno(), pelicula.getCosto);  
  
    }  
}  
  
public class Pelicula {  
    private 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 abstract class Suscripcion{  
    protected int costo = 0;  
    public double calcularCostoPelicula(int cargoExtra, int  
costoPelicula) {  
        this.costo = this.costo(cargoExtra, costoPelicula);  
        return costo;  
    }  
    public abstract int costo(cargoExtra, costoPelicula);  
  
    public class Basico extends Suscripcion{  
  
        public double costo(int cargoExtra, int costoPelicula){  
            return cargoExtra + costoPelicula;  
        }  
    public class Familia extends Suscripcion{
```

```

public double costo(int cargoExtra, int costoPelicula){
    return (cargoExtra + costoPelicula ) * 0.90;
}

}
public class Plus extends Suscripcion{

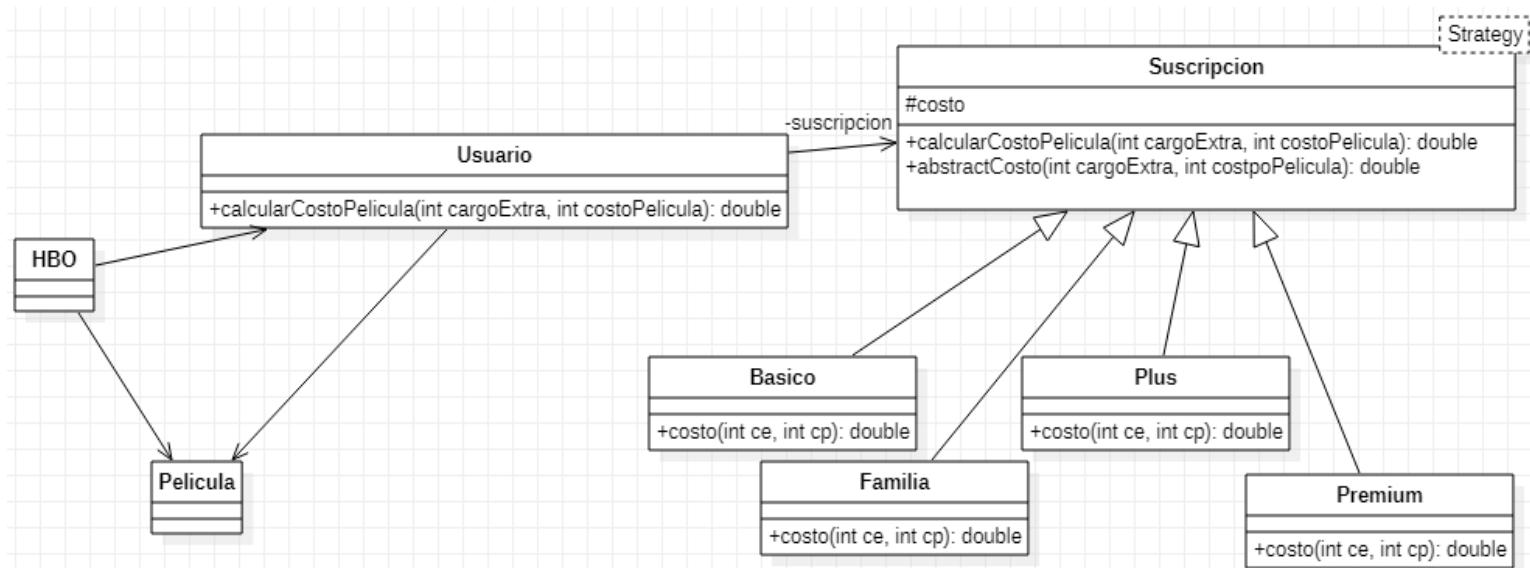
public double costo(int cargoExtra, int costoPelicula){
    return costoPelicula;
}

}
public class Premium extends Suscripcion{

public double costo(int cargoExtra, int costoPelicula){
    return costoPelicula * 0.75;
}

```

UML: // Suscripción es una clase abstracta.



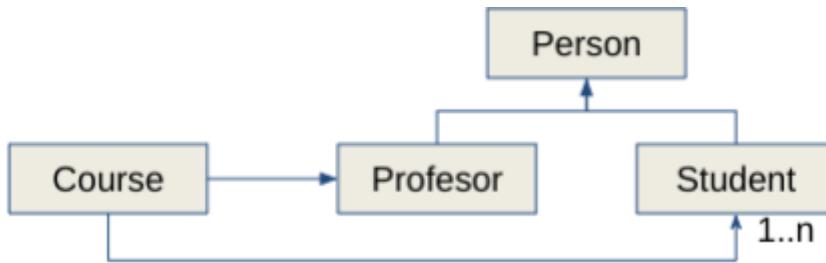
Ejercicio 3

Una plataforma de cursos on-line está preparando su sistema para hacer una actualización en su sistema de administración de cursos, alumnos y docentes. Los requerimientos iniciales del sistema fueron:

- El sistema permite registrar los cursos, los docentes y a los alumnos.
- Un Curso tiene un docente.
- Un Curso debe tener al menos un alumno.

La plataforma busca poder implementar otro tipo de cursos como talleres en donde los alumnos forman grupos de trabajo y se agrega la figura de mentor. Al mismo tiempo se

requiere que una persona pueda ser profesor y mentor. El diagrama de clases simplificado del diseño original se presenta a continuación.

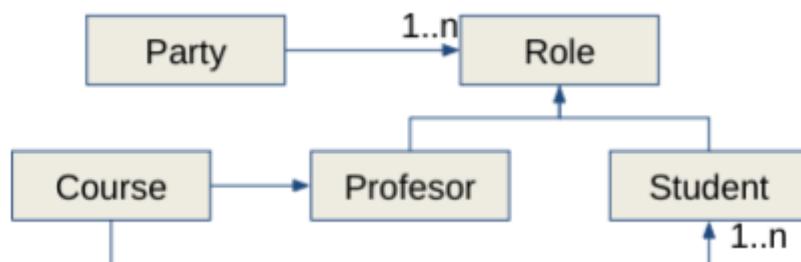


Antes de avanzar con la actualización, la empresa que desarrolló la plataforma propone hacer un refactoring que prepare el diseño para ser extendido en un siguiente paso mientras que implementa los requerimientos originales. El diseño resulta de aplicar el patrón Roles en donde una persona (Party) puede tener diferentes roles dentro de un modelo. El diagrama de clases simplificado del diseño resultante del refactoring se presenta a continuación.

VOLVER A CONSULTAR tengo 2 rtas que se contradicen.

Rta profesora Alejandra G:

No se agrega funcionalidad, lo de persona se pasó a party. Si es un Refactoring ya que no se agregó nada, solo se cambió la estructura.



La plataforma le solicita que audite el diseño propuesto demostrando si la transformación propuesta es o no un refactoring.

Rta mía: No es un refactoring porque se agrega funcionalidad. El refactoring no agrega funcionalidad, es una transformación del código que preserva el comportamiento. Si agrego un rol agregó comportamiento. El refactoring no corrige errores, no agrega funcionalidad, modifica la estructura del código existente.

Justificación: El diseño original NO permite que una misma persona sea Profesor Y Alumno simultáneamente. El nuevo diseño al permitir que una persona tenga una lista de Roles introduce una nueva capacidad, una persona ahora SI puede ser profesor y alumno al mismo tiempo. Al cambiar lo que el sistema puede representar se está alterando el comportamiento observable. Esto es un rediseño para extender funcionalidad no un refactoring.

//CONSULTAR si esta bien

Rta ayudante Martín:

Cambio funcionalidad. Por lo tanto no es refactoring. Ahora una persona puede tener un rol y ese rol puede ser profesor y estudiante. En el caso original solo es uno o el otro. El refactoring no debe alterar el comportamiento actual, ya sea por quitar o agregar cosas.

HAY QUE MANTENER COMPORTAMIENTO, ENTRADAS Y SALIDAS. Podes cambiar los métodos internamente pero debe tener misma salida y misma entrada. Para que sea refactoring

Ejercicio 4

Consideré el código fuente de la clase Student que se presenta a continuación.

```
import java.util.Vector;
import java.util.Iterator;

class Student {
    class ClassTaken {
        String name;
        Boolean done;
        int grade;

        public ClassTaken(String aName) {
            name = aName;
            done = false;
            grade = 0;
        }
    }

    String name;
    private Vector<ClassTaken> classes; // Plan de estudio
    String id;

    public Student(String newName, String ident) {
        name = newName;
        id = ident;
        classes = new Vector<ClassTaken>();
    }

    void addClass(String name) {
        classes.add(new ClassTaken(name));
    }

    void addDoneClass(String name) {
        ClassTaken aClass = new ClassTaken(name);
        aClass.done = true;
        classes.add(aClass);
    }

    void addGradedClass(String name, int grade) {
```

```

        ClassTaken aClass = new ClassTaken(name);
        aClass.done = true;
        aClass.grade = grade;
        classes.add(aClass);
    }

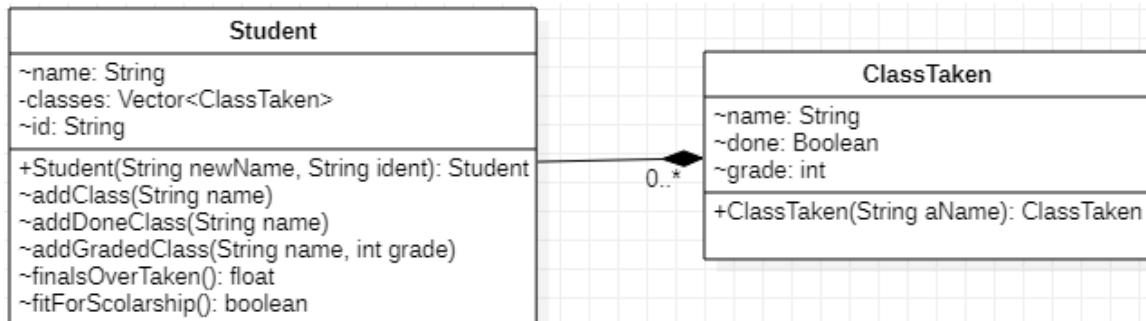
float finalsOverTaken() {
    int finals = 0;
    int done = 0;
    ClassTaken taken;
    Iterator it = classes.iterator();
    while(it.hasNext()){
        taken = (ClassTaken) it.next();
        if (taken.done)
            done++;
        if (taken.grade >= 6)
            finals++;
    }
    return (float) finals / done ;
}

boolean fitForScolarship() {
    int tally = 0;
    Iterator it = classes.iterator();
    ClassTaken taken;
    while (it.hasNext()) {
        taken = (ClassTaken) it.next();
        if (taken.grade >= 4)
            tally++;
    }
    return ((float) tally / classes.size() > 0.5) &&
        (this finalsOverTaken() > 0.66);
}
}

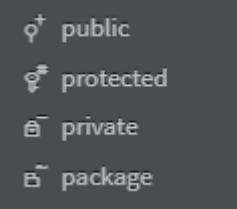
```

Responda las siguientes consignas:

1. Documente el código con un diagrama de clases UML
2. Escriba las reglas que hacen que un alumno esté en condiciones de recibir una beca, implementadas por el método *fitForScolarship()*.
3. Desde el código original se implementaron algunas transformaciones:
 1. UML diamante al revez -// CONSULTE A MARTIN Y A GUILIANA Y AMBOS DIJERON QUE IGNORE ESTE EJERCICIO.



Dudas: Cuando los métodos y las v.i no dice que tipo de alcance tiene (privado/public/protected) significa package?? en StartUML la 4ta opción la tiene como “package”



//mostrar opción 2 que está en hoja física y consultar cual prefiere la cátedra.

Class taken es una clase privada

2. Escriba las reglas que hacen que un alumno esté en condiciones de recibir una beca, implementadas por el método fitForScolarship().

// hacer

3. Desde el código original se implementaron algunas transformaciones:

a. Evalúe (analizando el código) si estos cambios pueden ser refactorings o no

Si, es refactoring, ya que no se corrigieron errores, no se agregó funcionalidad y solo se modificó la estructura del código existente. //CONSULTAR

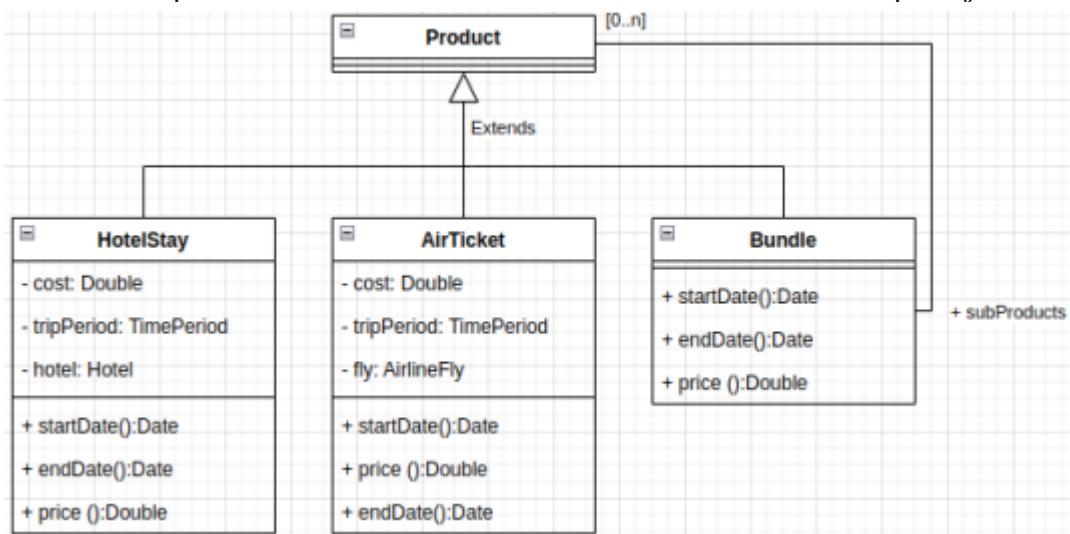
b. Compile el código del proyecto original y del refactorizado que se encuentra en el Moodle de la OO2_2025 y compare los resultados de correr el programa StudentCmd (original y refactorizado). //hacer

Se eliminó constante (hardcoded)	<pre>this finalsOverTaken()>0.66 se cambió a this finalsOverTaken()>fitRatio() y se implementó el método float fitRatio() { return 2/3; }</pre>
Se eliminó constante (hardcoded)	<pre>(float) tally / classes.size() >= 0.5 se cambió por (float) tally / classes.size() >= finalsRatio() y se implementó el método float finalsRatio() { return 1/2; }</pre>
Cambio de nombre	Se renombró el método <code>finalsOverTaken()</code> a <code>goodGradesOverTaken()</code>

Ejercicio 5 Estudie el siguiente diagrama de clases:

Refactorizar de manera que cumpla con las siguientes condiciones:

- A. Se requiere eliminar la definición de variables duplicadas.
- B. Se requiere eliminar código repetido.
- C. Se requiere minimizar la cantidad de definiciones del método `price()`



HotelStay	<pre> startDate() { return tripPeriod.start; } endDate() { return tripPeriod.end; } </pre>	
	<pre> price() { return tripPeriod.duration() * hotel.nightPrice() * hotel.discountRate(); } </pre>	
AirTicket	<pre> startDate() { return tripPeriod.start; } endDate() { return tripPeriod.end; } </pre>	
	<pre> price() { return fly.price() * fly.promotionRate(); } </pre>	
Bundle	<pre> startDate() { return min(subProducts.startDate()); } endDate() { return max(subProducts.startDate()); } </pre>	
	<pre> price() { return sum(subProducts.price()); } </pre>	

A. Se requiere eliminar la definición de variables duplicadas.

Code smell:

Hacer clase intermedia entre HotelStay y AirTicket para poner las v.i repetidas como cost, tripPeriod. La llamaría ProductoPorNoche, sería una clase abstracta

Refactoring

create new class intermedia
pull up field //consultar

B. Se requiere eliminar código repetido.

Code smell:

Subo los métodos duplicados entre HotelStay y AirTicket a la clase abstracta Intermedia.
Esos métodos se llaman starDate() y endDate()

Refactoring

pull up method

C. Se requiere minimizar la cantidad de definiciones del método price()

Se define como método abstracto en la clase Producto, convirtiendo en Product en abstracta

no se puede porque no hacen lo mismo MAL //consultado

Se puede simplificar, solo hay que entender que hace. → hacer

Ejercicio 7 - Notas En el sistema de cálculo de notas de alumnos de una facultad se contaba con el siguiente método para calcular promedios de notas de los alumnos: **//CONSULTADO**

```
public double promedio(List<Integer> notas) {
    int suma = 0;
    for (Integer n : notas) {
        suma += n;
    }
    return suma / notas.size();
}
```

Se evaluó que dicho método presentaba varios *bad smells* y se decidió aplicar *refactorings* para solucionarlos, produciendo el siguiente código:

```
public class Nota {
    private final int valor;

    public Nota(int valor) {
        if (valor < 0 || valor > 10) {
            throw new IllegalArgumentException("Nota inválida");
        }
        this.valor = valor;
    }

    public int getValor() {
        return valor;
    }
}

public class EstadisticaNotas {
    public static double promedio(List<Nota> notas) {
        return notas.stream()
                    .mapToInt(Nota::getValor)
                    .average()
                    .orElse(0.0);
    }
}
```

Tareas:

1. Analice ambas versiones del código y determine si se realizó un correcto refactoring. Justifique.
 2. Determine los *bad smells* que fueron detectados y qué *refactorings* se aplicaron para solucionarlos. ¿Ve algún otro *bad smell* que no se haya detectado?
 3. ¿El nuevo código presenta *bad smells*? De ser así indique cuáles y qué refactoring aplicaría para solucionarlos.
 4. Identifique claramente las precondiciones y postcondiciones de cada versión del código.
-
1. No es un refactoring por que se agregó funcionalidad. La variable *notas* dejó de ser tipo *List<Integer> notas* y paso a ser *List<Nota> notas*. Esta clase *Notas* tiene un método que valida si la nota es válida o no, antes los aceptaba, eso es cambio de funcionalidad. Si solo estaba la clase pero mantenía la misma funcionalidad eso si es considerado refactoring, crear una clase es parte de reestructurar diseño.

Y otro cambio también en la funcionalidad es que ahora no hace falta crear la clase donde esta el metodo promedio porque ahora es un método **static**

Antes:

```
EstadisticaNotas estadistica = new EstadisticaNotas();  
double resultado = estadistica.promedio(listaDeNotas);
```

Ahora con el método static:

```
double resultado = EstadisticaNotas.promedio(listaDeNotas);
```

Es decir se cambió parte de la funcionalidad.

2. Los bad smell que se fueron detectando son:

- a. se reinventará la rueda recorriendo una lista con un for y no con un stream().
- b. y se eliminaron variables temporales

Refactoring aplicados :

- a. Replace loop with pipeline
- b. replace temp variables for temp query

No veo ningun bad smell que no se haya detectado

ESTAN MAL IMPLEMENTADOS, NO CUMPLEN CON LAS PRE CONDICIONES EN EL CASO 1 SI LISTA DE NOTAS ESTÁ VACIA SE HARIA UNA DIVISIÓN POR CERO Y EL PROGRAMA FALLA EN EL CASO 2 NO SE CUMPLE ESA PRECONDICIÓN.

SE APLICO MAL EL REFACTRING. SE DA UN CAMBIO DE FUNCIONALIDAD. EN EL 2DO CASO DA 0 (cero) NO ERORR. SE APLICO MAL -//consultado

3. No veo bad smell en el nuevo código

4. Versión del código 1: //este punto no lo consulte

definición de precondicion en refactoring: es un conjunto de afirmaciones que deben ser verdaderas en el codigo fuente original antes que se pueda aplicar un refactoring de manera segura. Son restricciones estructurales del codigo fuente original que deben cumplirse antes de aplicar el refactoring

postcondiciones: son garantías estructurales que deben cumplirse despues del refactoring ej: todos los tester siguen pasando. El metodo fue movido con exito a la nueva clase y mantiene su comportamiento. remarcó (SE MANTUVO

COMPORTAMIENTO sin esto no es refactoring)

precondiciones: el parámetro notas es una lista de números enteros o sea el rango de números que puede tener el valor de nota es el de Integer. La lista de notas no puede estar vacia (ya que se sino se divide por cero)

postcondiciones :si la lista notas no esta cargada se genera una división por cero devolviendo un error. Y si esta cargada devuelve un entero.

Versión del codigo 2:

precondiciones: el valor de la nota debe ser un número entre 0 a 10. No puede ser ni 0 ni 10.

postcondiciones: Si el valor de las notas es un número entre 0 a 10 sin contar los extremos nos da un promedio, en caso contrario un mensaje de “Nota invalida”

Si la lista notas no esta cargada devuelve 0.0 no un error.

precondiciones para pasar del código 1 al código 2: (rtas javi)
que no haya una clase llamada EstadosicasNota, y Nota, el método promedio() se movió
de clase así que debió ser una precondición que no haya un método en esa clase con la
firma del método promedio.

postcondiciones: no cargar números fuera de rango, en caso de hacerlo lanza excepción. El
método promedio devuelve el promedio de notas o el valor 0. Se crearon 2 clases nuevas .
Las referencias de promedio se cambiaron, ahora son mediante nota.promedio()

Gemini

Ejercicio 7 - Notas

Este ejercicio pide analizar dos versiones de un método que calcula el promedio de
notas.

1. ¿Se realizó un correcto refactoring?

No, estrictamente hablando no es un refactoring correcto, aunque sí es una mejora de
diseño.

Justificación:

Un refactoring es una modificación de la estructura interna que no altera el
comportamiento observable externo. En este caso, el comportamiento sí cambió en
dos aspectos clave:

1. **Manejo de errores:** La versión original aceptaba cualquier número entero
(List<Integer>), incluyendo valores sin sentido como -5 o 15. La nueva versión
introduce la clase Nota, que lanza una IllegalStateException si el valor no
está entre 0 y 10. Esto cambia cómo el sistema reacciona ante datos inválidos.
2. Comportamiento con listas vacías: La versión original provocaría un error de
división por cero (ArithmetricException) si recibe una lista vacía. La nueva
versión, al usar .orElse(0.0), devuelve 0.0 en ese caso, cambiando el resultado
en este escenario límite.

Aunque el cálculo para datos válidos se preserva, estos cambios en el manejo de
casos excepcionales constituyen una alteración del comportamiento.

2. Bad smells detectados y refactorings aplicados

- **Bad Smell 1: Primitive Obsession** (Obsesión por Primitivos).
 - Descripción: El código original usaba un tipo primitivo (Integer) para
representar un concepto del dominio (Nota), que tiene reglas propias
(debe estar entre 0 y 10).
 - Refactoring aplicado: Replace Primitive with Object. Se creó la clase
Nota para encapsular el valor y sus reglas de validación.
- **Bad Smell 2: Loops.**
 - Descripción: El for loop para calcular la suma es una forma imperativa
y menos expresiva de procesar una colección.

- Refactoring aplicado: Replace Loop with Pipeline. Se reemplazó el bucle por un Stream de Java, que es más declarativo y conciso.

3. ¿El nuevo código presenta bad smells?

Sí, presenta los siguientes:

- Bad Smell: Data Class (Clase de Datos).
 - Descripción: La nueva clase Nota es esencialmente un contenedor de datos con un constructor y un getter, pero sin comportamiento propio.
 - Refactoring sugerido: En este caso simple, podría ser aceptable. Sin embargo, si el concepto de "Nota" creciera (ej. necesitara métodos como estaAprobada()), se le podría añadir más comportamiento.
- Bad Smell: Feature Envy (Envidia de Características) / Uso de Métodos Estáticos.
 - Descripción: El método promedio es estático y reside en una clase de utilidad (EstadisticaNotas), operando sobre una lista de notas que no le pertenece. Esto es un enfoque más procedural que orientado a objetos.
 - Refactoring sugerido: Move Method. Se podría crear una nueva clase, por ejemplo LibretaDeNotas, que contenga la List<Nota> y tenga su propio método promedio(). Esto aumenta la cohesión, ya que los datos y las operaciones sobre esos datos vivirían en el mismo objeto.

4. Precondiciones y Postcondiciones

- Versión 1 (Original):
 - Precondiciones (implícitas): La lista notas no debe ser nula ni vacía (para evitar NullPointerException y ArithmeticException). Los valores enteros de la lista deben ser semánticamente válidos como notas.
 - Postcondiciones: Devuelve un double con el promedio aritmético de los valores de la lista.
- Versión 2 (Refactorizar):
 - Precondiciones (explícitas): La lista notas no debe ser nula. Todos sus elementos deben ser instancias válidas de Nota (lo que garantiza que los valores están entre 0 y 10).
 - Postcondiciones: Devuelve un double con el promedio aritmético de los valores. Si la lista está vacía, devuelve 0.0.

Ejercicio 8 //CONSULTADO a Martin

Precio final Dado el siguiente código de cálculo de precio para un sistema de ventas:

```
public double calcularPrecioFinal(double precio, double descuento, double impuestos) {
    return (precio - (precio * descuento)) + impuestos;
}
```

- Analice el código e indique qué precondiciones se deben cumplir para con los datos de entrada y qué postcondiciones se garantizan tras la finalización del método.
- Identifique claramente los malos olores en el código.
- Aplique un refactoring que mejore la legibilidad y que permita expresar más claramente las precondiciones y postcondiciones del código.

1.

precondiciones: descuento tiene que ser un valor entre 0 y 1(es un porcentaje de 0% a 100% si es mayor a 1 el precio final podria ser negativo, si es menor a cero seria un aumento) , precio e impuesto no debe ser negativo

postcondiciones: devuelve el precio de la venta con impuestos y descuento en caso de que haya.

2.

Bad smell: se puede resolver cada cuenta en un método aparte y hacer la llamada a ese método.

No veo un bad smell.

3.

Sin refactoring

```
public double calcularPrecioFinal(double precio, double descuento, double
impuestos) {
    return (precio - (precio * descuento)) + impuestos;
}
```

con refactoring: (extract method)

```
private double precioConDescuento(double precio, double descuento){
    return precio - (precio * descuento)
}
```

```
private double precioConImpuestoAplicado( double impuesto, double precio, double
descuento) {
    return impuesto + precioConDescuento(precio, descuento);
}
public double calcularPrecioFinal(double precio, double descuento, double
impuestos) {
    return precioConImpuestoAplicado( impuestos, precio, descuento)
}
```

Esta mal corregido segun Gemini

1. Análisis de Precondiciones y Postcondiciones

El método `calcularPrecioFinal` funciona bajo ciertos supuestos que no están explícitos en el código.

Precondiciones (Lo que se debe cumplir para que funcione correctamente):

- **precio debe ser un número no negativo (≥ 0)**. Un precio negativo no tiene sentido en este contexto.
- **descuento debe ser un valor entre 0.0 y 1.0**. Representa un porcentaje (de 0% a 100%). Si es mayor a 1, el precio final podría ser negativo. Si es menor a 0, sería un aumento en lugar de un descuento.
- **impuestos debe ser un número no negativo (≥ 0)**. Los impuestos no suelen ser negativos.

Postcondiciones (Lo que el método garantiza si las precondiciones se cumplen):

- El método devuelve un `double` que representa el precio final tras aplicar un descuento porcentual y sumar un monto fijo de impuestos.
- El método no modifica el estado de ningún objeto; es una operación pura.

2. Identificación de Malos Olores (Bad Smells)

El código presenta principalmente un mal olor:

- **Primitive Obsession (Obsesión por Primitivos)**: Este es el "bad smell" más evidente. El código utiliza tipos de datos primitivos (`double`) para representar conceptos complejos del dominio que tienen sus propias reglas y comportamientos.
 - Un **precio** no es cualquier `double`; no puede ser negativo.
 - Un **descuento** no es cualquier `double`; es un porcentaje que debe estar entre 0 y 1.
 - Los impuestos tampoco deberían ser negativos.
 Al usar `double`, perdemos la capacidad de hacer explícitas estas reglas y el código se vuelve menos expresivo y seguro.
- **Long Parameter List (Lista Larga de Parámetros)**: Aunque tres parámetros no es un caso extremo, es un síntoma de la obsesión por primitivos. Estos tres datos están fuertemente relacionados y probablemente deberían estar encapsulados en un objeto.

3. Aplicación de Refactoring

Para solucionar estos malos olores, el refactoring principal a aplicar es **Replace Primitive with Object**. Crearemos "Value Objects" para cada concepto, haciendo que las precondiciones sean explícitas y el código mucho más legible.

Paso 1: Crear los Value Objects

Creamos clases para `Precio`, `Descuento` e `Impuestos`. La validación (las precondiciones) se mueve dentro de los constructores de estas clases.

Java

```
// Objeto para manejar el concepto de Precio (y cualquier valor monetario)
public class Monto {
    private final double valor;
```

```

public Monto(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("Un monto no puede ser negativo.");
    }
    this.valor = valor;
}

public Monto aplicarDescuento(Descuento descuento) {
    double valorFinal = this.valor * (1 - descuento.getTasa());
    return new Monto(valorFinal);
}

public Monto sumar(Monto otroMonto) {
    return new Monto(this.valor + otroMonto.getValor());
}

public double getValor() {
    return valor;
}
}

// Objeto para manejar el concepto de Descuento
public class Descuento {
    private final double tasa; // Valor entre 0.0 y 1.0

    public Descuento(double tasa) {
        if (tasa < 0 || tasa > 1) {
            throw new IllegalArgumentException("La tasa de descuento debe estar entre 0 y 1.");
        }
        this.tasa = tasa;
    }

    public double getTasa() {
        return tasa;
    }
}

```

Paso 2: Refactorizar el método original

Ahora, el método `calcularPrecioFinal` se vuelve mucho más expresivo. En lugar de manejar `doubles`, opera con los conceptos del dominio. La lógica de cálculo se delega a los propios objetos, mejorando la cohesión.

Java

```

public class CalculadoraDeVentas {

    public Monto calcularPrecioFinal(Monto precio, Descuento descuento, Monto impuestos) {

```

```
        Monto precioConDescuento = precio.aplicarDescuento(descuento);
        return precioConDescuento.sumar(impuestos);
    }
}
```

Ventajas del Código Refactorizado ✨

- **Legibilidad:** El nuevo método se lee casi como lenguaje natural: "calcula el precio final tomando un precio, aplicándole un descuento y luego sumándole los impuestos".
- **Seguridad:** Las precondiciones ahora son explícitas. Es imposible crear un `Monto` negativo o un `Descuento` inválido. Si se intenta, el programa fallará de inmediato con una excepción clara, en lugar de producir resultados incorrectos silenciosamente.
- **Cohesión:** La responsabilidad de aplicar un descuento ahora reside en la clase `Monto`, que es donde lógicamente pertenece.
- **Reutilización:** Las clases `Monto` y `Descuento` pueden ser reutilizadas en cualquier otra parte del sistema que maneje estos conceptos.

No estás agregando nueva funcionalidad, y esta es una distinción crucial en el concepto de refactoring.

Agregar esas condiciones **hace explícitas las precondiciones que ya existían implícitamente**. No estás cambiando lo que el método *debería* hacer, sino que lo estás haciendo más robusto y seguro al validar los supuestos bajo los cuales ya operaba.

Funcionalidad vs. Supuestos del Dominio

- **Nueva Funcionalidad:** Sería agregar algo que el sistema no podía hacer antes. Por ejemplo, si al cálculo de precio le añadieras un "descuento por lealtad del cliente" que no existía. Eso cambia el resultado esperado para ciertos casos y añade una nueva regla de negocio.
- **Refactorización (con validación):** El método original **ya asume** que nadie le pasará un precio negativo o un descuento del 200%. Si lo hicieran, el resultado sería un valor sin sentido (-150, por ejemplo), lo cual es un bug. El "comportamiento observable" que se preserva es el cálculo correcto para datos válidos. Al agregar las validaciones (`precio >= 0`), estás:
 1. **Formalizando los supuestos** del dominio.
 2. **Previniendo bugs** que ocurren cuando esos supuestos se rompen.
 3. **Haciendo que el código falle rápido** y de manera clara (con una excepción) en lugar de producir resultados incorrectos silenciosamente.

La Perspectiva de Fowler

Martin Fowler considera que este tipo de cambios son refactorizaciones válidas. Al aplicar **Replace Primitive with Object**, el objetivo es precisamente encapsular no solo el valor, sino también el **comportamiento y las reglas** asociadas a ese valor.

Piensa en esto: el comportamiento observable del método calcularPrecioFinal para un precio de 100, un descuento de 0.10 y un impuesto de 21 era y sigue siendo 111. El refactoring no cambia eso. Lo que cambia es cómo reacciona el sistema ante datos *inválidos* que nunca debieron ser aceptados. Pasar de "dar un resultado absurdo" a "lanzar un error claro" es una mejora de la estructura interna y la robustez, no un cambio de funcionalidad.

Ejercicio 9 - Reserva de habitaciones En el sistema de gestión de un hotel se cuenta con el siguiente método: //consultado

```
public Reserva reservarHabitacion(Hotel hotel, Cliente cliente, LocalDate desde,
                                    LocalDate hasta) {
    Habitacion habitacion = hotel.buscarDisponible(desde, hasta);
    Reserva reserva = new Reserva(habitacion, cliente, desde, hasta);
    hotel.registrarReserva(reserva);
    return reserva;
}
```

Responda:

- 1. ¿Qué supuestos deben cumplirse para que el método funcione correctamente?

Tiene que haber un hotel, un cliente y 2 fechas.

Ningun parametro sea null

Fecha inicio tiene que ser menor que fecha hasta

El hotel tiene que tener habitaciones registradas.

El hotel tiene disponibilidad en ese periodo. No se hace nada en caso de que no la haya porque hay disponibilidad infinita?

- 2. ¿Qué se puede asumir tras la ejecución correcta del método?

Que se realizó una reserva con un hotel, cliente y fecha. Y por lo tanto que hubo disponibilidad.

- 3. Existe un problema cuando se intenta reservar habitaciones para un día sin vacantes disponibles, ¿qué refactoring se podría aplicar para corregir esto?

No, no hay chequeo de disponibilidad. ¿Sería un refactoring? estoy agregando

funcionalidad. //consultado . NO HAY REFACTORYING PORQUE

ESTARIAS AGREGANDO FUNCIONALIDAD. esta el problema

pero eso no es refactoring,

SE PUEDE AGREGAR UN NULL OBJECT como refactoring para que se haga algo si no hay habitación

En caso de que se agregue la funcionalidad lo haría con un método en Hotel que consulte si hay disponibilidad es decir si hay habitación para esa fecha, me devolvería un true o false, si no se puede realizar la reserva escribiría un mensaje de ""No hay disponibilidad para ese periodo".

```
public class Hotel {
    private List habitaciones;
```

```
public boolean hayDisponibilidad(LocalDate desde, LocalDate hasta) {  
    return habitaciones.stream().anyMatch(h -> h.estaDisponible(desde, hasta));  
}
```

Gemini

Ejercicio 9 - Reserva de habitaciones

Este ejercicio pide analizar un método `reservarHabitacion` y proponer una mejora.

1. ¿Qué supuestos deben cumplirse? (Precondiciones)

Para que el método funcione correctamente, se asume lo siguiente:

- Los parámetros `hotel`, `cliente`, `desde` y `hasta` no son nulos.
- La fecha `hasta` es posterior a la fecha `desde`.
- El supuesto más importante: El hotel debe tener al menos una habitación disponible en el rango de fechas especificado, de modo que la llamada `hotel.buscarDisponible(desde, hasta)` devuelva una instancia válida de ¹⁰ `Habitacion`.

2. ¿Qué se puede asumir tras la ejecución? (Postcondiciones)

Si el método se ejecuta sin errores, se puede garantizar que:

- Se ha creado y devuelto un objeto `Reserva`. ¹¹
- La reserva ha sido registrada exitosamente en el sistema del hotel. ¹²
- El estado interno del hotel ha cambiado (una habitación que antes estaba libre ahora está ocupada para el período de la reserva).

3. Refactoring para el problema de falta de vacantes

El problema ocurre cuando `hotel.buscarDisponible` no encuentra una habitación y probablemente devuelve `null`. La línea siguiente (`new Reserva(habitacion, ...)`) intentaría usar esa referencia nula, provocando un `NullPointerException`.

- Bad Smell: Returning `null`. Devolver `null` para señalar un evento excepcional (como la falta de disponibilidad) obliga al cliente a realizar comprobaciones y puede llevar a errores.
- Refactoring Aplicado: La mejor solución es usar excepciones para manejar flujos excepcionales.

Solución Propuesta: Replace Error Code with Exception.

1. Modificar `hotel.buscarDisponible`: En lugar de devolver `null` cuando no hay habitaciones, este método debería lanzar una excepción personalizada y explícita.
2. Java

```

// Dentro de la clase Hotel
public Habitacion buscarDisponible(LocalDate desde, LocalDate hasta) {
    // ... lógica para buscar habitación ...
    if (habitacionEncontrada == null) {
        throw new NoHabitacionesDisponiblesException("No hay habitaciones disponibles
para el período seleccionado.");
    }
    return habitacionEncontrada;
}

```

3.

4.

5. Modificar el método reservarHabitacion: El cliente (en este caso, el método que llama) ahora debe manejar esta condición excepcional de forma explícita con un bloque try-catch.

6. Java

```

public Reserva reservarHabitacion(Hotel hotel, Cliente cliente, LocalDate desde,
LocalDate hasta) {
    try {
        Habitacion habitacion = hotel.buscarDisponible(desde, hasta);
        Reserva reserva = new Reserva(habitacion, cliente, desde, hasta);
        hotel.registrarReserva(reserva);
        return reserva;
    } catch (NoHabitacionesDisponiblesException e) {
        // Manejar el error: se puede loguear, notificar al usuario,
        // o devolver null de forma controlada.
        System.err.println(e.getMessage());
        return null; // O relanzar una excepción de más alto nivel.
    }
}

```

7.

8.

- Ventaja: Este cambio hace que el código sea mucho más robusto. El "camino feliz" (cuando hay habitaciones) queda limpio y legible dentro del try, mientras que el caso excepcional se gestiona de forma explícita y segura en el catch.

Observacion: no entendia porque hacer un chequeo en el ejercicio 7 era agregar funcionalidad y en el ejer 8 era hacer refactoring: "No entiendo porque en el ejercicio 7 no es un refactoring si agrega una comprobación de nota para que cumpla un rango y cuando hago ese refactoring de comprobación de un rango en el ejercicio 8 si es un refactoring "aca la respuesta

Excelente pregunta. Es una de las distinciones más importantes y sutiles del refactoring.

La diferencia clave está en si el cambio altera un comportamiento definido (incluso si es un error) o si formaliza un supuesto implícito (previniendo un bug).

Ejercicio 7: Cambio de Comportamiento Definido

En el código original, el método tenía un comportamiento observable específico para un caso límite:

- Comportamiento Original: Si le pasabas una lista vacía, el código intentaba dividir por cero (`suma / notas.size()`)¹, lo que provocaba un error en tiempo de ejecución (`ArithmeticException`). Ese "crash" era el comportamiento original del sistema.
- Comportamiento Nuevo: El código refactorizado, al usar `.orElse(0.0)`², ahora devuelve `0.0` si la lista está vacía.

Conclusión: Se cambió el resultado observable para un caso de entrada específico (lista vacía). Pasar de "lanzar una excepción" a "devolver 0.0" es una adición de funcionalidad para manejar un caso que antes no se contemplaba correctamente. Por lo tanto, no es un refactoring puro.

Ejercicio 8: Formalización de un Supuesto Implícito

En este caso, el método original no tenía un comportamiento definido o útil para datos inválidos; simplemente producía resultados sin sentido.

- Comportamiento Original: Si le pasabas un precio negativo o un descuento del 200%, el método no fallaba, sino que calculaba un resultado matemáticamente correcto pero lógicamente absurdo (un precio final negativo). El código ya asumía implícitamente que los datos serían válidos. El mal resultado era un bug causado por romper ese supuesto.
- Comportamiento Nuevo: Al agregar las validaciones, no estás cambiando el resultado para los casos válidos (el comportamiento que se quiere preservar). Lo que haces es hacer explícita la precondition que antes estaba implícita. En lugar de devolver un resultado basura (un bug), ahora el código falla de manera rápida y clara (con una excepción).

Conclusión: Estás reemplazando un bug (resultado incorrecto) con un manejo de errores robusto. No estás cambiando la funcionalidad principal, sino reforzando las reglas del dominio para que el método no pueda ser usado incorrectamente. Esto se considera una mejora de la estructura interna y la seguridad, lo cual sí es un refactoring válido.

Analogía Simple analogy

Imagina una función que calcula el área de un cuadrado (`lado * lado`).

- Caso similar al Ejercicio 7: Si la función originalmente se rompía si le pasabas `lado = 0`, y la modificas para que devuelva un mensaje como "El área es cero", estás cambiando su comportamiento.

- Caso similar al Ejercicio 8: La función ya asume que lado no puede ser negativo. Si un usuario le pasa -5 y la función devuelve 25 (un resultado absurdo), eso es un bug. Si la refactorizas para que lance un error si lado < 0, no estás cambiando la funcionalidad (calcular el área de cuadrados válidos), sino que estás previniendo un mal uso.

Ejercicio 10

```

1  public class PrimeGeneratorStream {
2
3      public int currentNumber;
4
5      public PrimeGeneratorStream() {
6          this.currentNumber = 1;
7      }
8      public int next() {
9          while (true) {
10             if (isPrime(this.currentNumber)) {
11                 int nextPrime = this.currentNumber;
12                 this.currentNumber++;
13                 return nextPrime;
14             }
15             this.currentNumber++;
16         }
17     }
18     private boolean isPrime(int number) {
19         if (number < 1)
20             return false;
21         for (int i = 1; i <= Math.sqrt(number); i++) {
22             if (number % i == 0) {
23                 return false;
24             }
25         }
26         return true;
27     }
28 }
```

a) Un colega que revisó el código se dio cuenta que este código es poco eficiente dado que las instancias de esta clase repiten mucho de sus cálculos y para lograr mayor eficiencia se propone "memorizar" los números primos ya calculados para no tener que re calcularlos. Presente el código resultante.

Para mejorar la eficiencia, se puede modificar la clase para que "recuerde" los números primos que ya ha calculado. Esto evita tener que recalcular la primalidad de un número cada vez. Este tipo de optimización, que no altera el comportamiento observable del objeto, es considerada un refactoring.

La estrategia es guardar los primos encontrados en una lista. El método isPrime puede usar esta lista para verificar si un número es divisible solo por los primos ya conocidos, lo cual es mucho más rápido.

Código Resultante con Memorización

```
public class PrimeGeneratorStream {

    private int currentNumber;
    private List<Integer> primes; // Lista para memorizar los primos

    public PrimeGeneratorStream() {
        this.currentNumber = 1;
        this.primes = new ArrayList<>();
    }

    public int next() {
        while (true) {
            this.currentNumber++;
            if (isPrime(this.currentNumber)) {
                primes.add(this.currentNumber); // Agrega el nuevo primo a
la lista
                return this.currentNumber;
            }
        }
    }

    private boolean isPrime(int number) {
        if (number <= 1) {
            return false;
        }
        // Un número es primo si no es divisible por ningún primo anterior a
su raíz cuadrada
        for (Integer prime : primes) {
            if (prime > Math.sqrt(number)) {
                break;
            }
            if (number % prime == 0) {
                return false;
            }
        }
        return true;
    }
}
```

b) Un colega que revisó el código se dio cuenta que hay un error, pero no le ha dicho cual es. Puede detectar el error y corregirlo? Considera que ese cambio es un refactoring?
El código original tiene un error crítico en el método isPrime.

Error Detectado: El bucle for (int i = 1; i <= Math.sqrt(number); i++) comienza la comprobación desde i = 1. Como cualquier número entero es divisible por 1, la condición number % i == 0 siempre será verdadera en la primera iteración, haciendo que el método retorne false incorrectamente para cualquier número mayor que 1.

Código Corregido

Para solucionarlo, el bucle debe comenzar en 2.

```
private boolean isPrime(int number) {  
    if (number < 2) { // 1 y los negativos no son primos  
        return false;  
    }  
    for (int i = 2; i <= Math.sqrt(number); i++) { // El bucle debe empezar  
en 2  
        if (number % i == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

¿Es un Refactoring?

No, este cambio no es un refactoring. Según la definición de Fowler, un refactoring es una modificación al código que no altera su comportamiento observable externo. Un bug fix, por definición, cambia el comportamiento del programa (de uno incorrecto a uno correcto). Por lo tanto, es una corrección, no un refactoring.

c) Dado el cambio que Ud. presentó para que la clase "recuerde" los números primos ya calculados. ¿Considera que es un refactoring?

Sí, la modificación presentada en el punto (a) para "memorizar" los números primos es un refactoring.

Justificación:

- Comportamiento Externo: El comportamiento observable de la clase no cambia. Si llamas al método next(), seguirás obteniendo la misma secuencia de números primos (2, 3, 5, 7, ...).
- **Estructura Interna: La estructura interna se ha modificado para mejorar una característica no funcional:** el rendimiento. El código es más eficiente, pero su resultado es idéntico.

Fowler considera que las optimizaciones de rendimiento que no cambian la interfaz pública ni el resultado de un componente son un tipo de refactoring.

d) Un colega que revisó el código se dio cuenta que la variable currentNumber no debería ser publica sino privada. ¿Qué condicion o condiciones se deberían dar para que el cambio a privada se pueda considerar un refactoring?

Cambio de Visibilidad de currentNumber

La variable currentNumber actualmente es public. Cambiarla a private corresponde al refactoring conocido como "Encapsulate Field" (Encapsular Campo).

Para que este cambio sea considerado un refactoring, debe cumplirse una condición fundamental:

Condición: Ningún código externo a la clase PrimeGeneratorStream debe estar accediendo directamente a la variable currentNumber.

Si otra parte del sistema estuviera utilizando miGenerador.currentNumber para leer o modificar su valor, cambiar la visibilidad a private rompería esa funcionalidad, alterando así el comportamiento observable del sistema. Si ninguna clase externa depende de este campo público, el cambio es seguro y se considera un refactoring puro que mejora la encapsulación y la mantenibilidad del código.