ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

DISEÑO DE SOFTWARE

TAREA DE PATRONES DE DISEÑO

INTEGRANTES:

ICAZA BORBOR ARIHUZ ZAMIR

PACHECO CANTOS JOSUE JORDAN

VILLON PALMA MARIA GABRIELA

ROMAN TORRES AXEL SAMUEL

II PAO 2025

Sección A: Plan de Pruebas

1. Sección A: Plan de Pruebas

1.1Clase Hotel.java

Clase	ID	Método a probar	Precondicio-nes	Datos de entrada	Salida esperada	Propósito de la prueba
Hotel	HT01	agregar Habitacion ()	Hotel creado correctamente con nombre y dirección válida.	habitación = null	Se lanza IllegalArgumentException con el mensaje “La habitación no puede ser null”	Comprobar que no se puede agregar null como habitació n y que se lanza la excepción correcta
Hotel	HT02	Constructor	Ninguna	nombre= null, dirección= “Calle x”	Se espera que se lance IllegalArgumentException con el mensaje “Los parámetros no pueden ser nulos”	Verificar que no se pueden crear hoteles con nombres nulos.
Hotel	HT03	Constructor	Ninguna	nombre= “Hotel x”, direccion= null	Se espera que se lance IllegalArgumentException con el mensaje “Los parámetros	Verificar que no se pueden crear direcciones nulas.

					no pueden ser nulos”	

Al aplicar Extract Method, se crearon las clases Validador y ConstructorMensaje.

1.2 Clase Validador.java

Clase	ID	Método a probar	Precondiciones	Datos de entrada	Salida esperada	Propósito de la prueba
Validador	VR01	validarReserva()	Ninguna	usuario=null, mensaje= “mensaje ejemplo”	Lanza IllegalArgumentException con mensaje "El usuario y su nombre no pueden ser nulos".	Verificar que se valida usuario nulo

Validador	VO02	validarObserver()	Ninguna	observer!= null	No lanza excepción	Confirm ar que observer válido no genera error

Validador	VR03	validarReserva()	usuario != null, usuario.nombre != null	mensaje= null	Lanza IllegalArgumentException con mensaje "El mensaje no puede ser nulo".	Verificar que se valida mensaje nulo

Validador	VR04	validar Reserva ()	usuario != null, usuario. nombre != null	mensaje =“mensaje valido”	No lanza excepción	Confirmar que datos válidos no generan error
Validador	VO01	validar Observer ()	ninguna	observer = null	Lanza IllegalArgumentException con mensaje "Observ er no puede ser null"	Verificar que se detecta observer nulo

Validador	VO02	validar Observer ()	ninguna	observer!= null	No lanza excepción	Confirmar que observer válido no genera error

1.3 Clase ConstructorMensaje.java

Clase	ID	Método a probar	Precondicio-nes	Datos de entrada	Salida esperada	Propósito de la prueba

Construc-ctor Mensaje	CM01	construir Reserva Creada	usuario con nombre “Miriam ”	usuario = (id = 1, nombre = "Miriam ", email = "miriam@ example. com") mensaje = "Reserva confirmada para el 05/10"	"Hola Miriam,\ n\Reserva confirmada para el 05/10\n\ n Gracias por elegir TravelStay."	Verificar construcción normal del mensaje de creación de reserva.
Construc tor Mensaje	CM02	construir Reserva Creada	usuario valido	usuario = (id=2, nombre = "Miriam ", email = "miriam@ example. com") mensaje = ""	"Hola Miriam,\ n\n\n\nG racias por elegir TravelStay."	Verificar que funciona con mensaje vacío.
Construc tor torMens aje	CM03	construir Reserva Creada	usuario con nombre vacío	usuario = (id = 3, nombre = "", email = "usuario3 @example. com") mensaje = "Reserva para el 05/10"	"Hola, \n\Reserva para el 05/10\n\ nGracias por elegir TravelStay."	Verificar que el formato se mantiene aunque el nombre esté vacío.

Constructor torMensaje	CM04	construir Reserva Cancelada	usuario con nombre "Samuel"	usuario = (id=4, nombre = "Samuel ", email = "samuel@ example. com") mensaje = "Reserva cancelada para el 10/05"	"Hola Samuel,\n\nReserva cancelada para el 10/05\n\ Lamentamos que hayas cancelado. Esperamos verte pronto."	Verificar construcción normal del mensaje de cancela-ción.
Constructor torMensaje	CM05	construir Reserva Cancelada	usuario válido	usuario = (id=5, nombre = "Samuel ", email = "samuel@ example.com") mensaje = ""	"Hola Samuel,\n\n\n\nLamentamos que hayas cancelado. Esperamos verte pronto."	Verificar que funciona con mensaje vacío.

Constructor torMensaje	CM06	construirReservaCancelada	usuario con nombre vacío	usuario = (id=6, nombre= "", email=" usuario6@example.com") mensaje= "Reserva cancelada para el 05/10"	"Hola, \n\nReserva cancelada para el 05/10\nm Lamentamos que hayas cancelado. Esperamos verte pronto."	Verificar que el formato se mantiene, aunque el nombre esté vacío.

1.4 Clase Usuario.java

Al cambiar el atributo correo de tipo String a una clase Email, se hicieron los siguientes test:

Clase	ID	Método a probar	Precondiciones	Datos de entrada	Salida esperada	Propósito de la prueba
Usuario	UT-USUARIO-01	getId()	Usuario válido creado	id=1, nombre= "Maria", email=" maria@mail.com"	1	Verificar que el id se almacena y retorna correctamente

Usuario	UT-USUARIO-02	getNombre()	Usuario válido creado	id=2, nombre="Juan", email="juan@mail.com"	"Juan"	Verificar que el nombre se almacena y retorna correcto
Usuario	UT-USUARIO-03	getCorreo()	Usuario válido creado	id=3, nombre="Ana", email="ana@mail.com"	Email("ana@mail.com")	Verificar que el correo se almacena y retorna correcto
Usuario	UT-USUARIO-04	Constructor	Ninguna	id=4, nombre="Bad", email="bademail "	IllegalArgumentException	Validar que no se permite crear usuario con email inválido
Usuario	UT-USUARIO-05	Constructor	Ninguna	id=5, nombre="Null", email=null	NullPointerException	Validar que no se permite crear usuario con email nulo

1.5 Clase Email.java

Para la refactorización de la clase Usuario, se creó la clase Email.

Clase	ID	Método a probar	Precondiciones	Datos de entrada	Salida esperada	Propósito de la prueba

Email	UT-EMAIL-01	Constructor/getValor	Ninguna	"test@mail.com"	"test@mail.com"	Verificar creación y obtención de email válido
Email	UT-EMAIL-02	Constructor	Ninguna	"noemail "	IllegalArgumentException	Validar que no se permite email inválido
Email	UT-EMAIL-03	Constructor	Ninguna	null	IllegalArgumentException	Validar que no se permite email nulo
Email	UT-EMAIL-04	equals/hashCode	Ninguna	"a@mail.com", "a@mail.com", "b@mail.com"	equals true/false, hashCode igual/distinto	Verificar igualdad y hashCode

Email	UT-EMAIL-05	toString	Ninguna	"string@mail.co m"	"string@mail.com"	Verificar representación en texto

1.6 Clase PaseoTuristico.java

Clase	ID	Método a probar	Precondiciones	Datos de entrada	Salida esperada	Propósito de la prueba
PaseoTuristico	pt1	verificarDisponibilidad	Estado de la instancia: CANCELADO	Ninguna (solo configuración del estado)	false	Verificar que un paseo cancelado no se considera disponible
PaseoTuristico	pt2	calcularPrecio	Precio de la instancia: 0	Ninguna	0.0	Comprobar que calcularPrecio() retorna correctamente 0
PaseoTuristico	pt3	calcularPrecio	Precio de la instancia: -50.0	Ninguna	-50.0	Verificar manejo de precios negativos

PaseoTuristico	pt4	mostrarDetalles	Instancia con nombre, precio y estado válidos	Ninguna	Imprime nombre, precio y estado sin excepciones	Comprobar que mostrarDetalles() imprime información básica correctamente

1.7 Clase Hospedaje.java

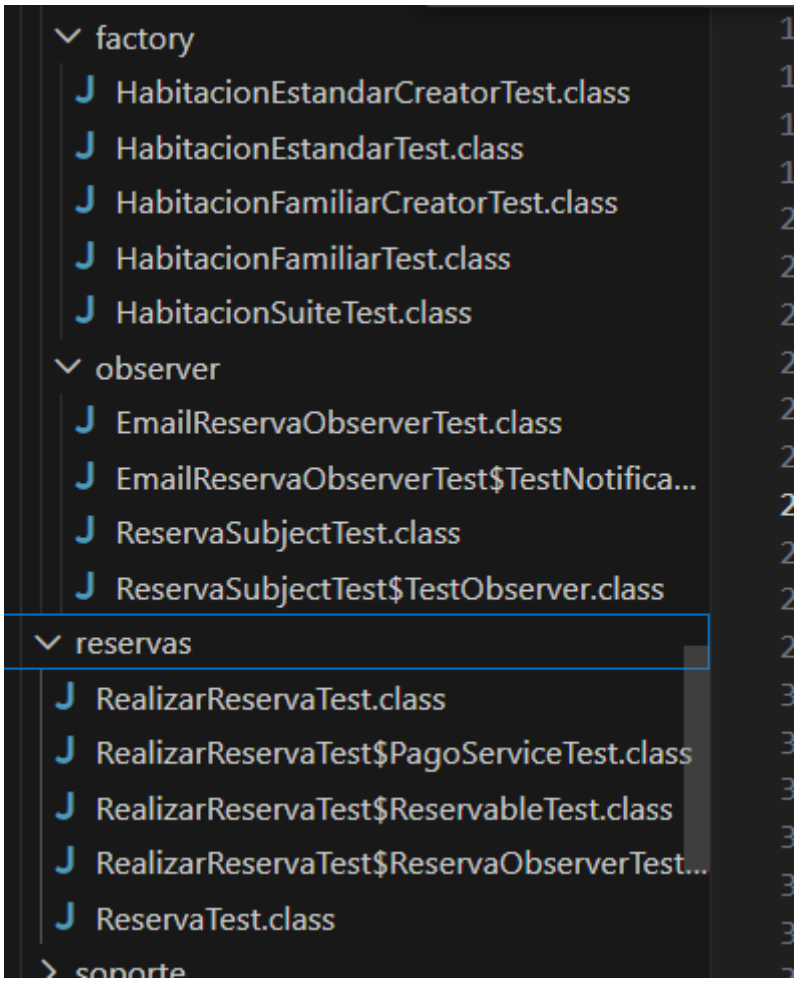
Clase	ID	Método a probar	Precondiciones	Datos de entrada	Salida esperada	Propósito de la prueba

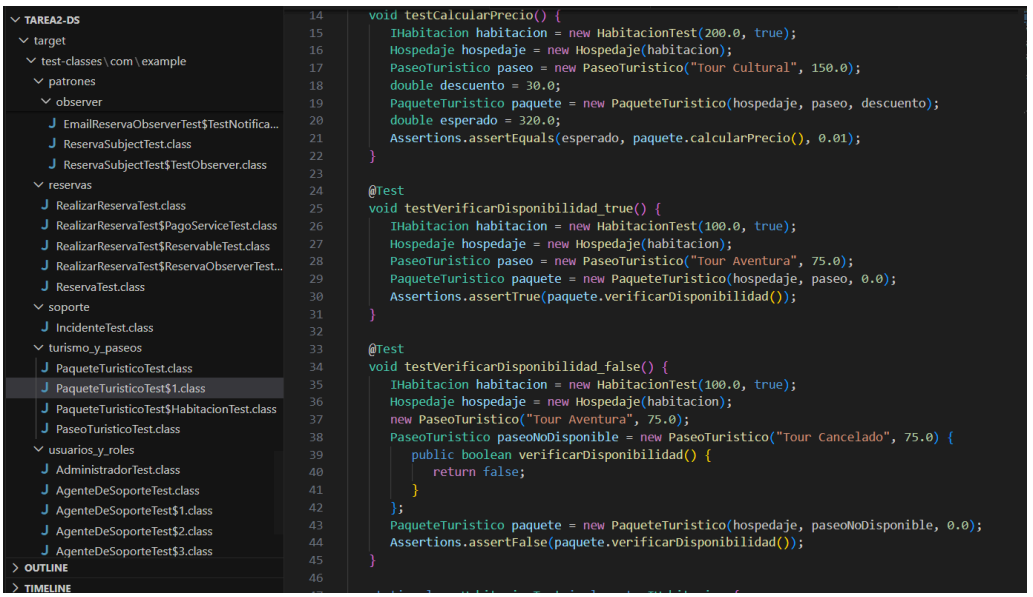
Hospedaje	Hp1	calcularPrecio	Instancia con precio=-50.0	Ninguno	Retorna -50.0	Verificar robustez del método ante valores negativos
Hospedaje	hp2	verificarDisponibilidad	Instancia con habitacion=null	Ninguno	Lanza NullPointerException o manejo controlado	Comprobar que la clase maneja correctamente ausencia de habitación
Hospedaje	hp3	calcularPrecio	Instancia con habitacion=null	Ninguno	Lanza NullPointerException o manejo controlado	Verificar robustez al calcular precio sin habitación

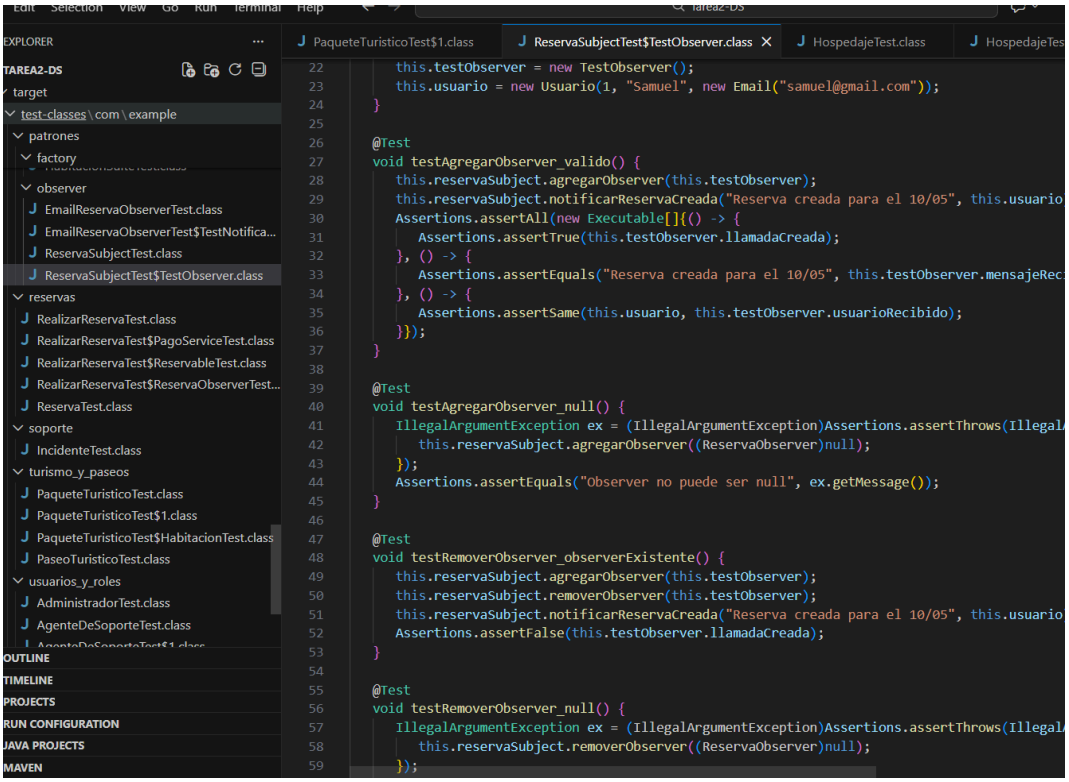
Hospedaje	hp4	mostrarDetalles	Instancia con habitación válida	Ninguno	Imprime detalles de la habitación sin excepción	Confirmar que mostrarDetalles() funciona correctamente y no lanza errores

Sección B: Implementación

Dentro del código se encuentra toda la implementación de las pruebas







Sección C: Detección de Code Smells

1) Duplicate Code

Descripción: Contienen lógica duplicada: Validaciones de párametros y construcción del asunto y cuerpo del mensaje. En la clase Reserva Subject los métodos de notificarReservaCreada y notificarReservaCancelada repiten la validaciones de párametros usuario y mensaje y la validación del observer. 1.1.2. Impacto en la calidad del software. **Impacto**: La duplicación afecta en la mantenibilidad del código, cada vez que se deba modificar la lógica habrá que actualizar múltiples métodos

Ubicación exacta:

- Archivo: src\main\java\com\example\patrones\observer\EmailReservaObserver.java
- Clase: EmailReservaObserver
- Método: onReservaCreada y onReservaCancelada

Evidencia:

```java
@Override
public void onReservaCreada(String mensaje, Usuario usuario) {
    if (usuario == null || usuario.getNombre() == null) {
        throw new IllegalArgumentException(s: "El usuario y su nombre no pueden ser nulos.
    }
    if (mensaje == null) {
        throw new IllegalArgumentException(s: "El mensaje no puede ser nulo.");
    }

    String asunto = "Confirmación de reserva TravelStay";
    String cuerpo = String.format(format: "Hola %s,\n\n%s\n\nGracias por elegir TravelStay
    notificador.enviar(asunto + "\n" + cuerpo, usuario);
}

@Override
public void onReservaCancelada(String mensaje, Usuario usuario) {

    if (usuario == null || usuario.getNombre() == null) {
        throw new IllegalArgumentException(s: "El usuario y su nombre no pueden ser nulos.
    }
    if (mensaje == null) {
        throw new IllegalArgumentException(s: "El mensaje no puede ser nulo.");
    }

    String asunto = "Cancelación de reserva TravelStay";
    String cuerpo = String.format(
            format: "Hola %s,\n\n%s\n\nLamentamos que hayas cancelado. Esperamos verte pro
            usuario.getNombre(),
            mensaje);
    notificador.enviar(asunto + "\n" + cuerpo, usuario);
}
```

Refactor recomendado:
Extract Method: mover la lógica repetida a métodos reutilizables.

2) Temporary Variable

Descripción: Declara una variable temporal precioHabitacion que solo sirve para almacenar un valor intermedio y luego se usa una vez en la expresión de retorno.
Impacto: El código es más largo y difícil de seguir. Al existir variables innecesarias aumenta la probabilidad de errores.
Ubicación exacta :

- Archivo: src\main\java\com\example\turismo_y_paseos\PaqueteTuristico.java
- Clase: PaqueteTuristico
- Método: calcularPrecio (...)

Evidencia:

```java
@Override
public double calcularPrecio() {
    // Usar el precio decorado de la habitación
    double precioHabitacion = 0.0;
    if (hospedaje.getHabitacion() != null) {
        precioHabitacion = hospedaje.getHabitacion().calcularPrecio();
    }
    return precioHabitacion + paseo.calcularPrecio() - descuento;
}
```

Refactor recomendado: Se recomienda aplicar Inline Temp y devolver el valor de la variable directamente en el return.

3) Feature envy

Descripción: Se accede directamente a los datos y métodos de otros objetos (hospedaje y paseo) en lugar de delegar la responsabilidad de mostrar sus detalles a las propias clases correspondientes.
Impacto: Disminuye la mantenibilidad, ya que los cambios en la forma de mostrar detalles de hospedaje o paseo requieren modificar PaqueteTuristico.
Ubicación exacta:

- Archivo: src\main\java\com\example\turismo_y_paseos\PaqueteTuristico.java
- Clase: PaqueteTuristico
- Método: mostrarDetalles(...)

Evidencia:

```java
@Override
public void mostrarDetalles() {
    System.out.println(x: "Paquete turístico:");
    if (hospedaje.getHabitacion() != null) {
        hospedaje.getHabitacion().mostrarDetalles();
    }
    System.out.println("Incluye paseo: " + paseo);
    System.out.println("Descuento aplicado: $" + descuento);
}
```

Refactor recomendado: Usar Move Method para trasladar la lógica de mostrar detalles a las clases Hospedaje y PaseoTuristico.

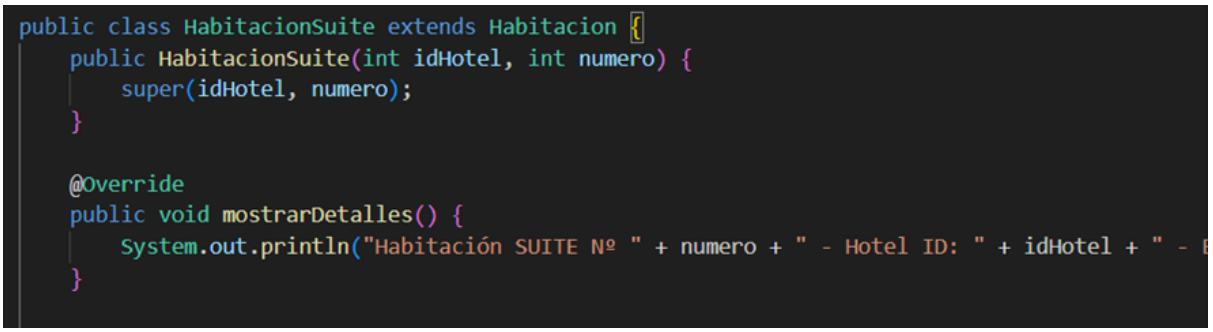
4) Shotgun Surgery

Descripción: Si mañana se quisiera cambiar la forma de mostrar detalles se tendría que modificar todas las subclases

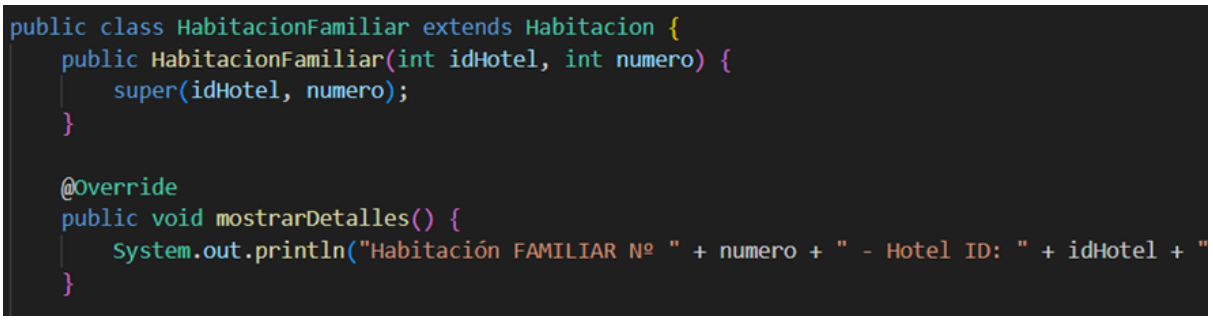
Impacto: Afecta a la mantenibilidad del código. Cada vez que se cambia el formato, hay que recordar actualizar todas las subclases. Así como la legibilidad, tener código duplicado en varias clases hace que sea más difícil de entender la lógica.
Ubicación exacta:

- Archivo: src\main\java\com\example\patrones\factory\Habitacion.java
- Clase: HabitacionSuite, HabitacionFamiliar, HabitacionEstandar (subclases de Habitacion)
- Método: mostrarDetaller()

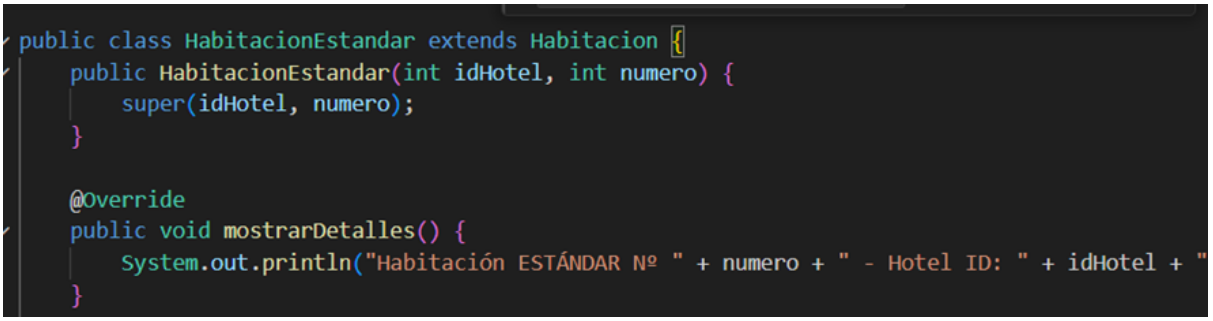
Evidencia:



```
public class HabitacionSuite extends Habitacion {
    public HabitacionSuite(int idHotel, int numero) {
        super(idHotel, numero);
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Habitación SUITE Nº " + numero + " - Hotel ID: " + idHotel + " - E
    }
```



```
public class HabitacionFamiliar extends Habitacion {
    public HabitacionFamiliar(int idHotel, int numero) {
        super(idHotel, numero);
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Habitación FAMILIAR Nº " + numero + " - Hotel ID: " + idHotel + "
    }
```



```
public class HabitacionEstandar extends Habitacion {
    public HabitacionEstandar(int idHotel, int numero) {
        super(idHotel, numero);
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Habitación ESTÁNDAR Nº " + numero + " - Hotel ID: " + idHotel + "
    }
```

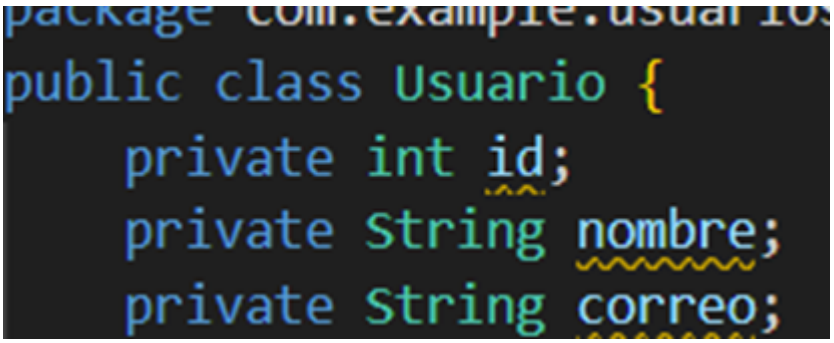
Refactor recomendado: Form Template Method para definir la estructura del método mostrarDetalles() en la superclase.

5) Primitive Obsession

Descripción: El atributo correo en la clase Usuario es de tipo String, lo que permite asignar cualquier valor sin validación ni restricciones. Esto puede llevar a errores, datos inconsistentes y lógica de validación dispersa en el código.
Impacto: Mayor probabilidad de errores por valores inválidos y dificultad para centralizar y reutilizar la lógica de validación de correos.
Ubicación exacta:

- Archivo: src\main\java\com\example\usuarios_y_roles\Usuario.java
- Clase: Usuario
- Atributo: correo

Evidencia:



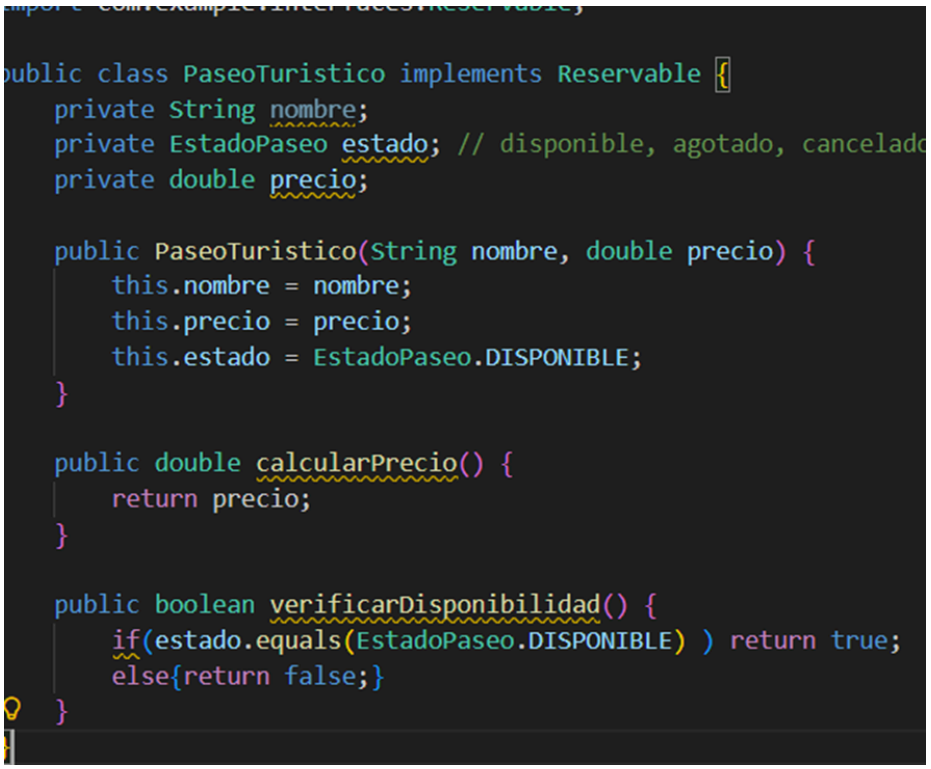
Refactor recomendado: Usar Replace Data Value with Object (Reemplazar valor de dato con objeto). Esto implica crear una clase específica Email para encapsular el campo correo, junto con su lógica de validación y comportamiento asociado, en lugar de usar un String simple.

6) Data Class

Descripción: La principal función de esta clase es almacenar datos (nombre, estado, precio). Sus métodos (calcularPrecio, verificarDisponibilidad) solo retornan valores de sus campos, sin lógica compleja ni comportamiento propio relevante.
Impacto: Dificulta la evolución del sistema, ya que la lógica relacionada con el paseo turístico puede dispersarse en otras clases. Aumenta el acoplamiento y reduce la cohesión, pues otras clases deben manipular directamente los datos.
Ubicación exacta:

- Archivo: src\main\java\com\example\turismo_y_paseos\PaseoTuristico.java
- Clase: PaseoTuristico
- Método:

Evidencia:



```
ublic class PaseoTuristico implements Reservable {
    private String nombre;
    private EstadoPaseo estado; // disponible, agotado, cancelado
    private double precio;

    public PaseoTuristico(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
        this.estado = EstadoPaseo.DISPONIBLE;
    }

    public double calcularPrecio() {
        return precio;
    }

    public boolean verificarDisponibilidad() {
        if(estado.equals(EstadoPaseo.DISPONIBLE) ) return true;
        else{return false;}
    }
}
```

Refactor recomendado: Simplify Conditional Expression Se reemplaza el if redundante en verificarDisponibilidad() por una sola expresión booleana, también se aplica Rename field en precio se renombra a preciobase para aclarar que es el valor inicial antes de aplicar cualquier regla de negocio. Además, es importante el uso de Encapsulate Field puesto que se agregan getters para acceder a los atributos de forma segura, evitando exposición directa.

7) Refused Bequest

Descripción: Las clases HabitacionFamiliar y HabitacionSuite heredan de la clase Habitacion, pero sobrescriben el método calcularPrecio() únicamente para lanzar una excepción (UnsupportedOperationException). Esto indica que estas subclases no pueden implementar correctamente toda la funcionalidad heredada.

Impacto:

- Rompe el principio de sustitución de Liskov: las subclases no pueden ser usadas en lugar de la superclase sin errores.
- Puede causar errores en el tiempo de ejecución si se invoca el método no implementado.
- Indica una jerarquía de herencia inapropiada, lo que dificulta el mantenimiento y la extensión del sistema.

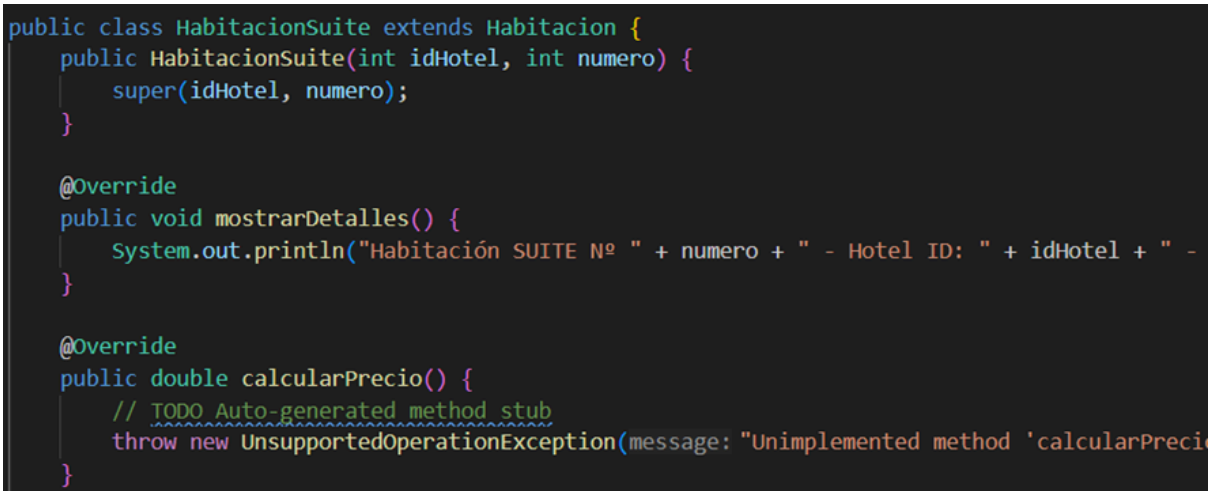
Ubicación exacta:

- Archivo:

src\main\java\com\example\patrones\factory\HabitacionFamiliar.java
src\main\java\com\example\patrones\factory\HabitacionSuite.java

- Clase: HabitacionFamiliar y HabitacionSuite
- Método: la herencia

Evidencia:



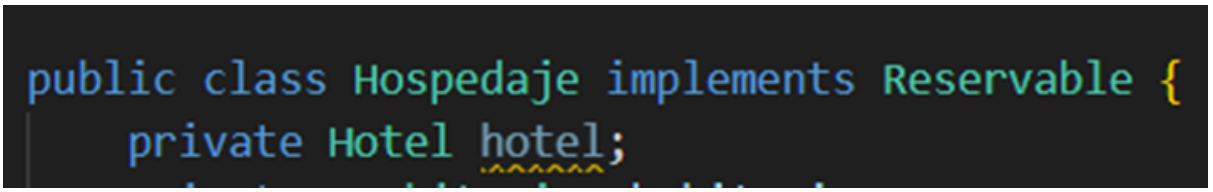
Refactor recomendado: Extract Interface de modo que sólo las clases que necesitan tal método la implementen, o por el contrario, implementar el método basado en la lógica de las clases de ser necesario.

8) Speculative Generality

Descripción: Se declara un atributo que no se. En Hospedaje existe hotel pero en el código actual no participa.
Impacto: Añade complejidad innecesaria al código y puede confundir a otros desarrolladores sobre su propósito o uso.
Ubicación exacta:

- Archivo: src/main/java/com/example/hospedaje/Hospedaje.java
- Clase: Hospedaje
- Campo: Hotel

Evidencia:



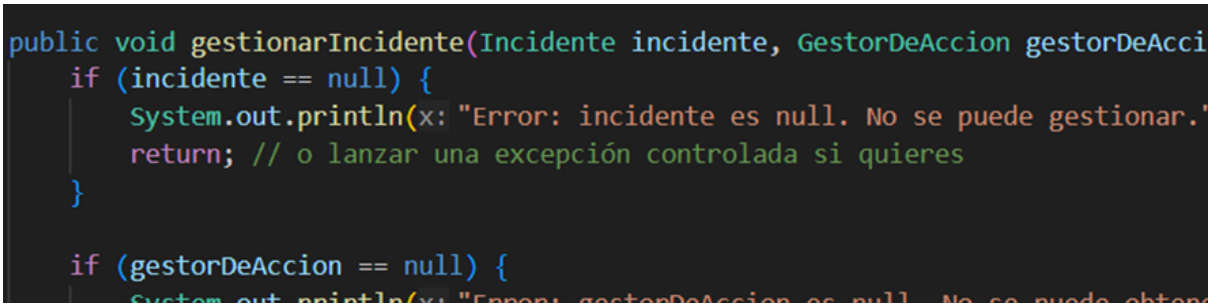
Refactor recomendado: Utilizará Remove Dead Field para eliminar el campo hotel de la clase para simplificar y Simplify Class si el campo no se usa.

9) Comments

Descripción: En la clase AgenteDeSoporte se han agregado varios comentarios para explicar líneas obvias de código las cuales fácilmente se puede reconocer su funcionalidad.
Impacto: Añade líneas innecesarias de código y aumenta el ruido visual, dificultando leer el código relevante.
Ubicación exacta:

- Archivo: src\main\java\com\example\usuarios_y_roles\AgenteDeSoporte.java
- Clase: AgenteDeSoporte
- Métodos:

Evidencia:



Refactor recomendado: Podemos eliminar las líneas de comentarios y además usar la técnica de refactorización Rename Method de modo que sea mucho más entendible a simple vista

10) Dead Code

Descripción: El método void reservar() en la clase no está implementado (lanza una excepción UnsupportedOperationException) y no es llamado ni utilizado en ningún otro método de la clase ni desde fuera de ella.
Impacto: Añade ruido y confusión al código, dificultando su comprensión y mantenimiento.
Ubicación exacta:

- Archivo: src/main/java/com/example/hospedaje/Hospedaje.java
- Clase: Hospedaje
- Método: reservar()

Evidencia:

```
void reservar() {
    throw new UnsupportedOperationException(message: "Not supported yet.");
}
```

Refactor recomendado: Eliminar el código no utilizado y los archivos innecesarios.

Sección D: Refactorización del Código

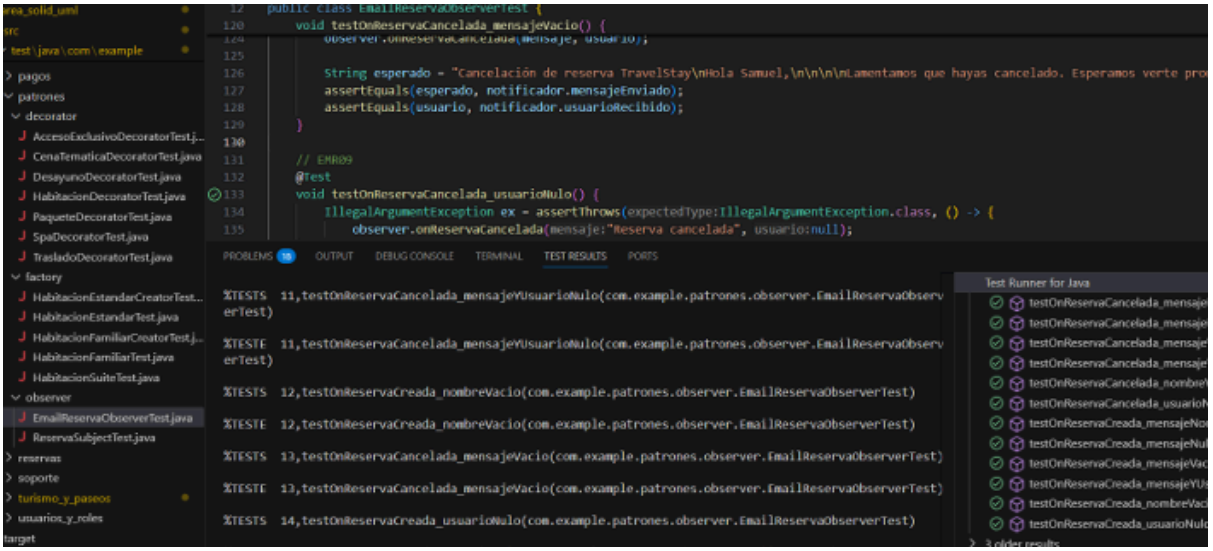
Refactoring:

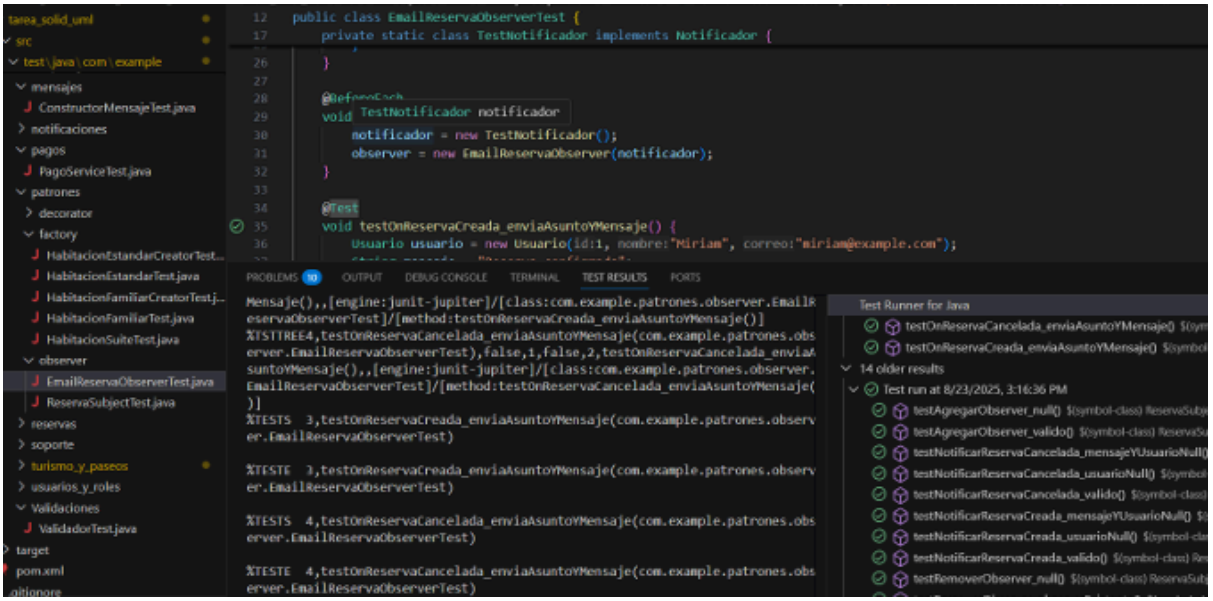
Extract Method

Code Smell Corregido: Duplicate Code
Antes/Después:

```java
package com.example.patrones.observer;

import com.example.interfaces.Notificador;
import com.example.mensajes.ConstructorMensaje;
import com.example.usuarios_y_roles.Usuario;
import com.example.validaciones.Validador;

public class EmailReservaObserver implements ReservaObserver {

    private Notificador notificador;

    public EmailReservaObserver(Notificador notificador) {
        if (notificador == null) {
            throw new IllegalArgumentException(s: "El notificador no puede ser nulo.");
        }
        this.notificador = notificador;
    }

    @Override
    public void onReservaCreada(String mensaje, Usuario usuario) {

        Validador.validarReserva(usuario, mensaje);
        String asunto = "Confirmación de reserva TravelStay";
        String cuerpo = ConstructorMensaje.construirReservaCreada(usuario, mensaje);
        notificador.enviar(asunto + "\n" + cuerpo, usuario);
    }

    @Override
    public void onReservaCancelada(String mensaje, Usuario usuario) {

        Validador.validarReserva(usuario, mensaje);

        String asunto = "Cancelación de reserva TravelStay";
        String cuerpo = ConstructorMensaje.construirReservaCancelada(usuario, mensaje);
        notificador.enviar(asunto + "\n" + cuerpo, usuario);
    }
}
```

Justificación:
En la clase EmailReservaObserver, los métodos onReservaCreada y onReservaCancelada contenían código duplicado. Al aplicar Extract Method: las validaciones se movieron a validarReserva(), ahora la validación se hace en un solo lugar, la construcción del cuerpo del mensaje se movió a contruirCuerpoReservaCreada() y construirCuerpoReservaCancelada() cada método tiene una responsabilidad
Pruebas unitarias:





Refactoring:

Form Template Method

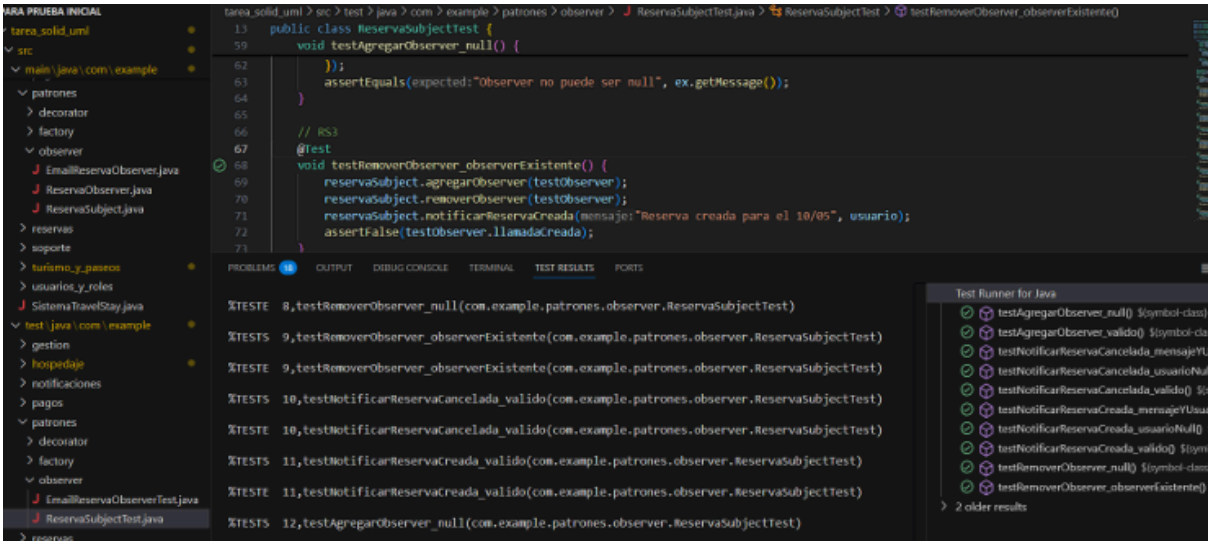
Code Smell Corregido: Shotgun Surgery
Antes:

```
public class ReservaSubject {

    private List<ReservaObserver> observers = new ArrayList<>();

    public void agregarObserver(ReservaObserver observer) {

        if (observer == null) {
            throw new IllegalArgumentException(s: "Observer no puede ser null");
        }

        observers.add(observer);
    }

    public void removerObserver(ReservaObserver observer) {

        if (observer == null) {
            throw new IllegalArgumentException(s: "Observer no puede ser null");
        }

        observers.remove(observer);

    }

    public void notificarReservaCreada(String mensaje, Usuario usuario) {

        if (mensaje == null || usuario == null) {
            throw new IllegalArgumentException(s: "Mensaje y Usuario no pueden ser null");
        }

        for (ReservaObserver observer : observers) {
            observer.onReservaCreada(mensaje, usuario);
        }
    }

    public void notificarReservaCancelada(String mensaje, Usuario usuario) {
```

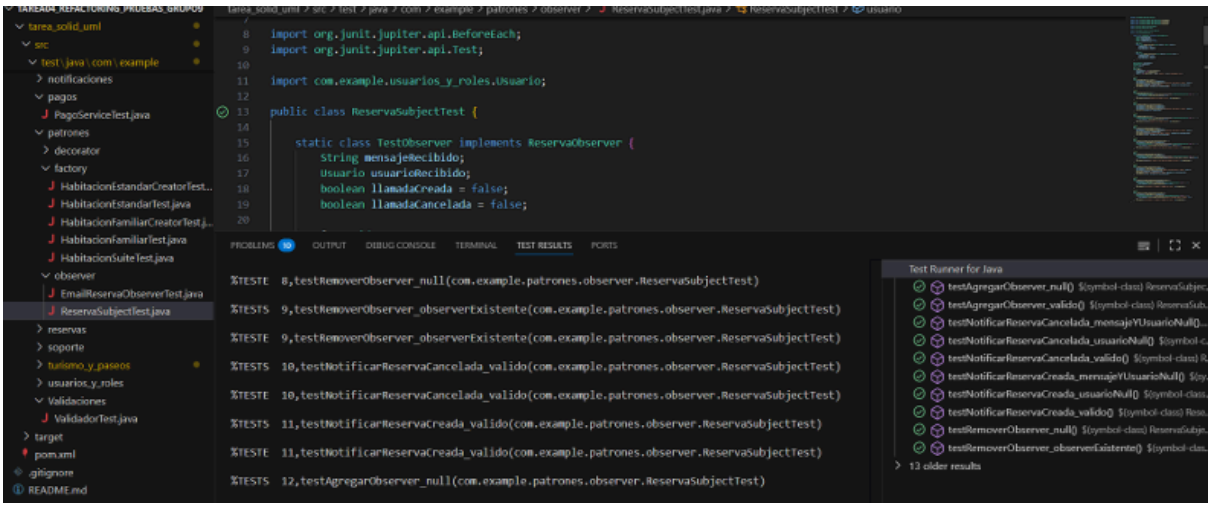
Despues

```java
package com.example.patrones.observer;

import java.util.ArrayList;
import java.util.List;

import com.example.usuarios_y_roles.Usuario;
import com.example.validaciones.Validador;

public class ReservaSubject {

    private List<ReservaObserver> observers = new ArrayList<>();

    public void agregarObserver(ReservaObserver observer) {

        Validador.validarObsever(observer);
        observers.add(observer);
    }

    public void removerObserver(ReservaObserver observer) {

        Validador.validarObsever(observer);
        observers.remove(observer);

    }
    public void notificarReservaCreada(String mensaje, Usuario usuario) {

       Validador.validarReserva(usuario, mensaje);

        for (ReservaObserver observer : observers) {
            observer.onReservaCreada(mensaje, usuario);
        }
    }

    public void notificarReservaCancelada(String mensaje, Usuario usuario) {

        Validador.validarReserva(usuario, mensaje);
```

Justificación:
El problema era que cada subclase implementa su propia versión de mostrarDetalles(). La solución consiste en aplicar Form Template Method y definir en mostrarDetalles toda la lógica común.
Pruebas unitarias:
Antes de refactoring



Despues de refactoring



Refactoring:

Move Method

Code Smell Corregido: Feature envy
Antes:

```java
public PaqueteTuristico(Hospedaje hospedaje, PaseoTuristico paseo, double descuento) {
    this.hospedaje = hospedaje;
    this.paseo = paseo;
    this.descuento = descuento;
}

@Override
public double calcularPrecio() {
    // Usar el precio decorado de la habitación
    double precioHabitacion = 0.0;
    if (hospedaje.getHabitacion() != null) {
        precioHabitacion = hospedaje.getHabitacion().calcularPrecio();
    }
    return precioHabitacion + paseo.calcularPrecio() - descuento;
}

@Override
public boolean verificarDisponibilidad() {
    return hospedaje.verificarDisponibilidad() && paseo.verificarDisponibilidad();
}
@Override
public void mostrarDetalles() {
    System.out.println(x: "Paquete turístico:");
    if (hospedaje.getHabitacion() != null) {
        hospedaje.getHabitacion().mostrarDetalles();
    }
    System.out.println("Incluye paseo: " + paseo);
    System.out.println("Descuento aplicado: $" + descuento);
}
```

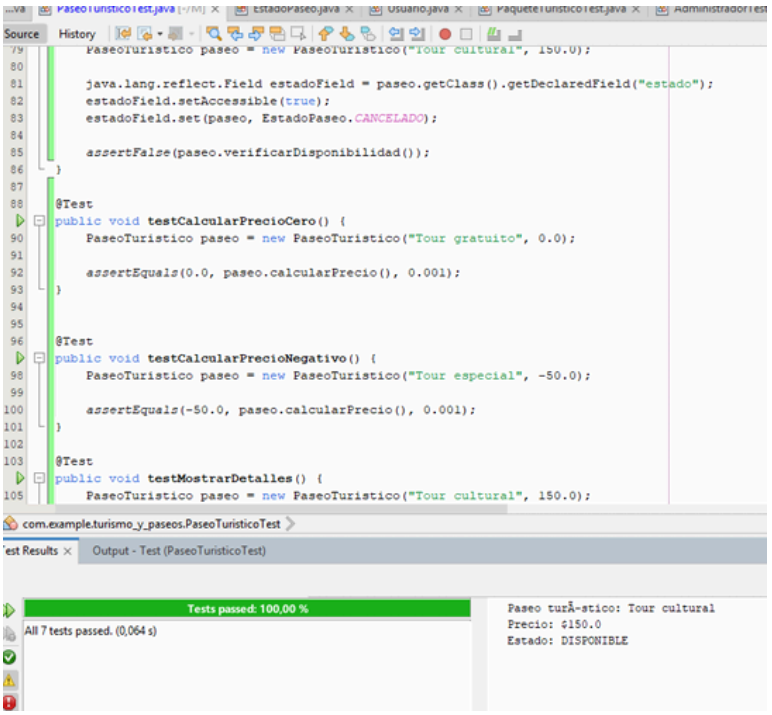
Después:

```java
public class PaqueteTuristico implements IPaqueteTuristico {
    private Hospedaje hospedaje;
    private PaseoTuristico paseo;
    private double descuento;

    public PaqueteTuristico(Hospedaje hospedaje, PaseoTuristico paseo, double descuento) {
        this.hospedaje = hospedaje;
        this.paseo = paseo;
        this.descuento = descuento;
    }

    @Override
    public double calcularPrecio() {
        if (hospedaje.getHabitacion() != null) {
            return hospedaje.getHabitacion().calcularPrecio() + paseo.calcularPrecio() - descu
        }
        else{ return  paseo.calcularPrecio() - descuento; }
    }

    @Override
    public boolean verificarDisponibilidad() {
        return hospedaje.verificarDisponibilidad() && paseo.verificarDisponibilidad();
    }
    @Override
    public void mostrarDetalles() {
        logger.info(msg: "Paquete turístico:");
        if (hospedaje != null) {
            hospedaje.mostrarDetalles();
        }
        if (paseo != null) {
            paseo.mostrarDetalles();
        }
        logger.info("Descuento aplicado: $" + descuento);
    }
```

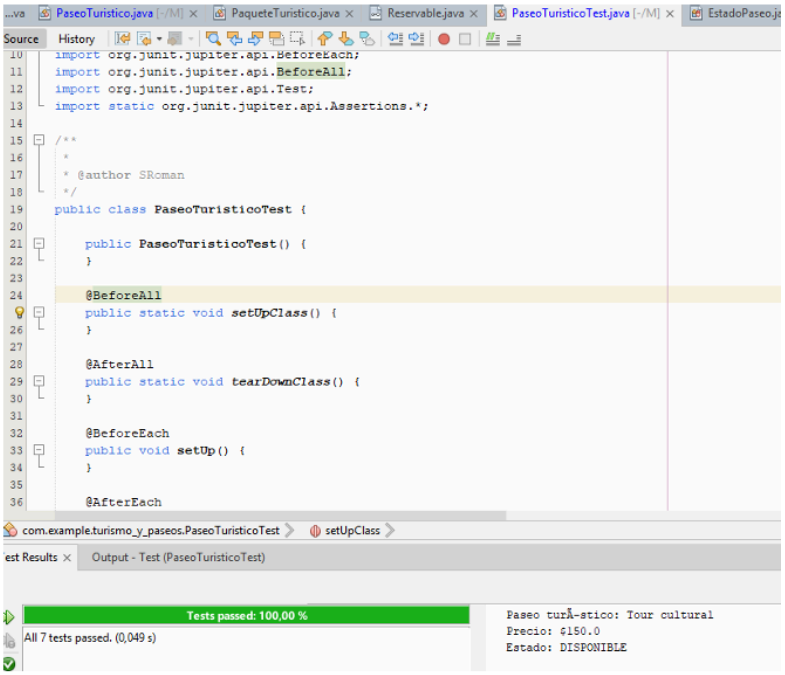
Justificación: Se aplicó la técnica Move Method trasladando la lógica de presentación de detalles desde la clase PaqueteTuristico hacia las clases responsables de los datos: Hospedaje y PaseoTuristico. Ahora, cada clase muestra sus propios detalles, y PaqueteTuristico solo coordina la llamada a estos métodos.

Pruebas unitarias:

Antes de refactoring



Despues de refactoring:



Refactoring:

Replace Data Value with Object

Code Smell Corregido: Primitive Obsesion
Antes:

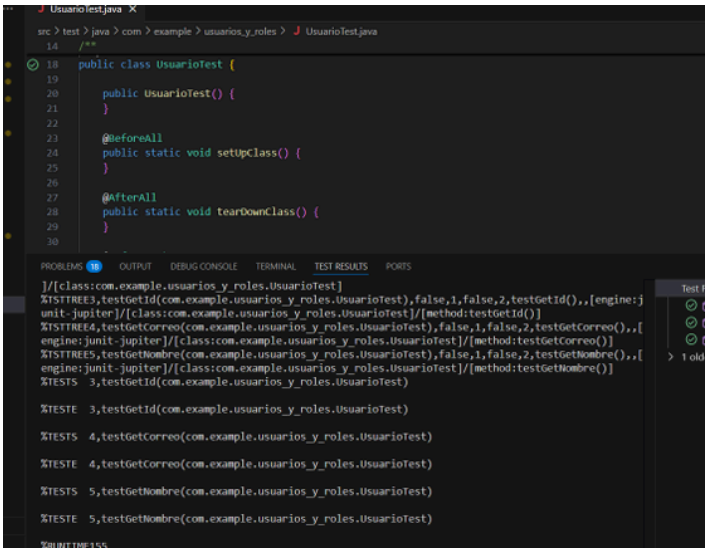
```
ackage com.example.usuarios_y_roles;
ublic class Usuario {
    private int id;
    private String nombre;
    private String correo;

    public Usuario(int id, String nombre, String correo) {
        this.id = id;
        this.nombre = nombre;
        this.correo = correo;
    }
    public String getCorreo() {
        return correo;
    }

    public String getNombre() {
        return nombre;
    }

    public int getId(){
        return id;
    }
```

Después:

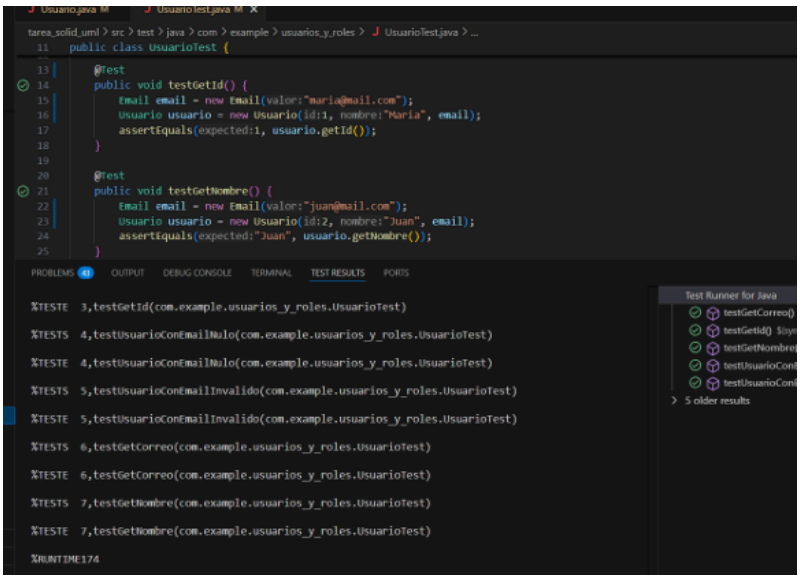
```java
package com.example.usuarios_y_roles;

public class Usuario {
    private int id;
    private String nombre;
    private Email correo;

    public Usuario(int id, String nombre, Email correo) {
        if (correo == null) {
            throw new NullPointerException(s: "El correo no puede ser null");
        }
        this.id = id;
        this.nombre = nombre;
        this.correo = correo;
    }


    public Email getCorreo() {
        return correo;
    }

    public String getNombre() {
        return nombre;
    }

    public int getId() {
        return id;
    }
}
```

Justificación:
Se reemplazó el campo correo de tipo String en la clase Usuario por un objeto de valor Email, el cual encapsula la validación y manejo de direcciones de correo electrónico. Esto centraliza la lógica y evita el uso de valores primitivos sin control.

Pruebas unitarias:
Antes de refactoring:



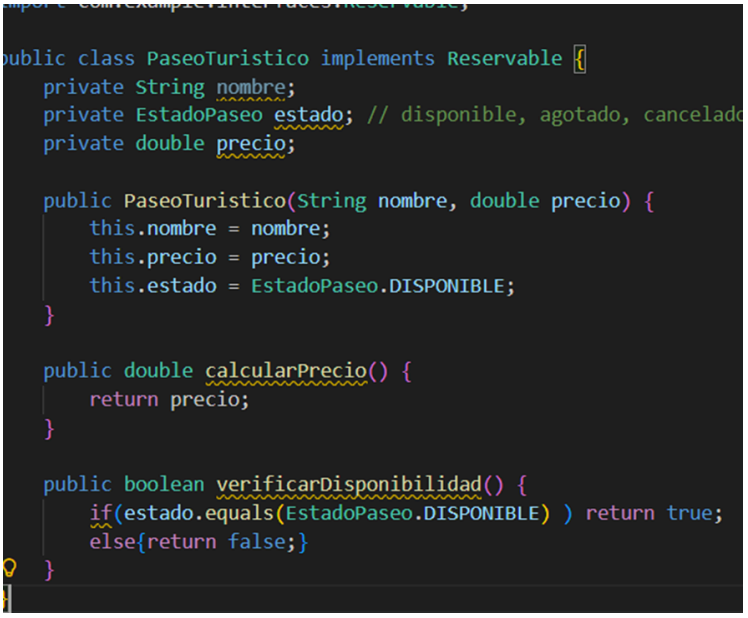
Despues de refactoring:



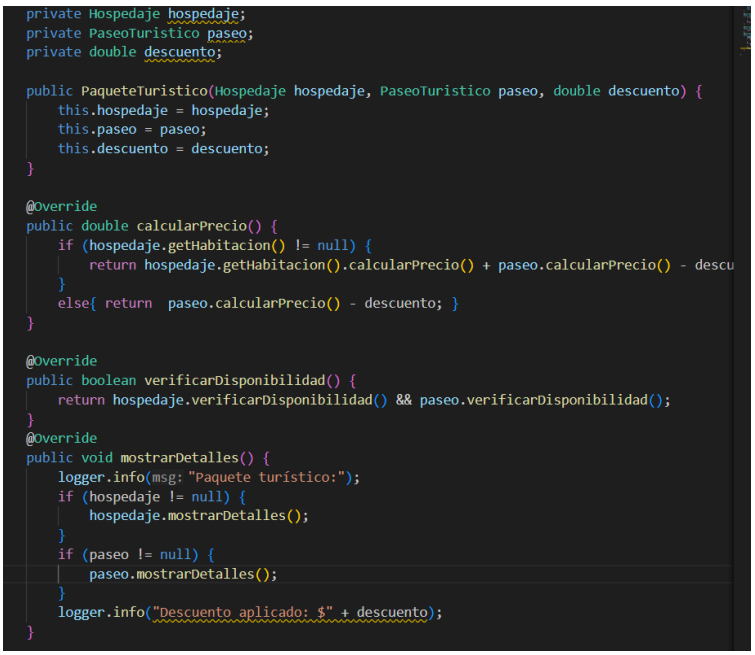
Refactoring:

Encapsulate Field

Code Smell Corregido: Data Class
Antes:

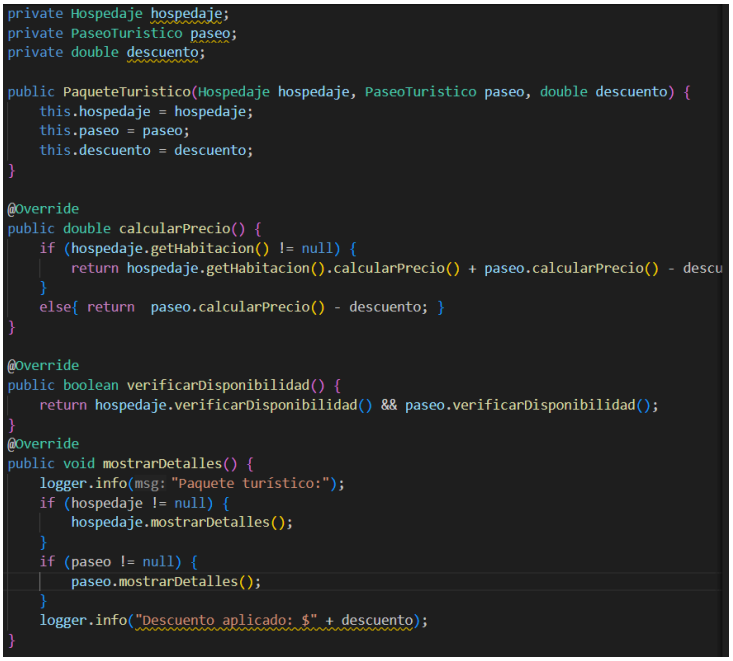


Después:



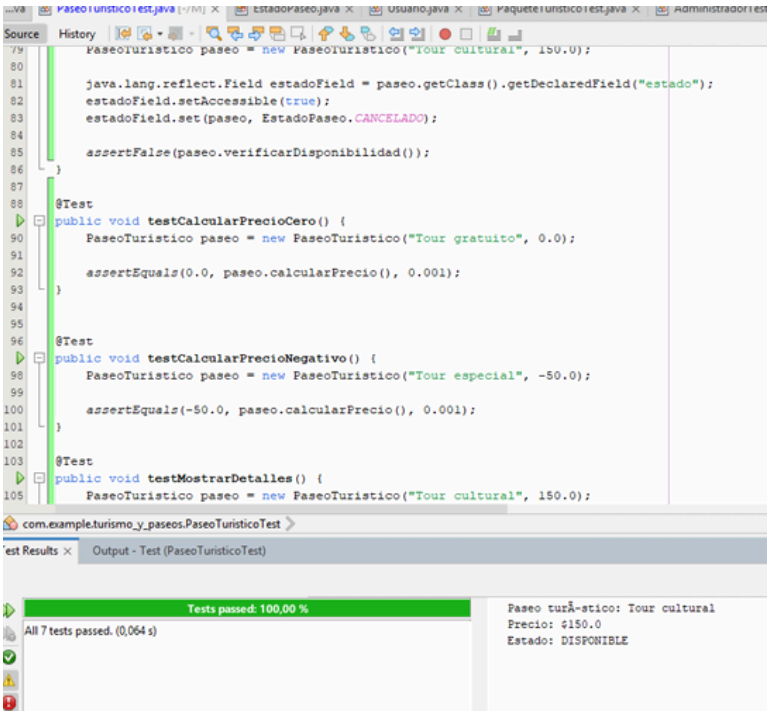
```java
private Hospedaje hospedaje;
private PaseoTuristico paseo;
private double descuento;

public PaqueteTuristico(Hospedaje hospedaje, PaseoTuristico paseo, double descuento) {
    this.hospedaje = hospedaje;
    this.paseo = paseo;
    this.descuento = descuento;
}

@Override
public double calcularPrecio() {
    if (hospedaje.getHabitacion() != null) {
        return hospedaje.getHabitacion().calcularPrecio() + paseo.calcularPrecio() - descu
    }
    else{ return  paseo.calcularPrecio() - descuento; }
}

@Override
public boolean verificarDisponibilidad() {
    return hospedaje.verificarDisponibilidad() && paseo.verificarDisponibilidad();
}
@Override
public void mostrarDetalles() {
    logger.info(msg: "Paquete turístico:");
    if (hospedaje != null) {
        hospedaje.mostrarDetalles();
    }
    if (paseo != null) {
        paseo.mostrarDetalles();
    }
    logger.info("Descuento aplicado: $" + descuento);
}
```

Justificación: Se encapsularon los campos nombre, precio y estado dentro de la clase PaseoTuristico, eliminando la lógica trivial que solo devolvía los valores.

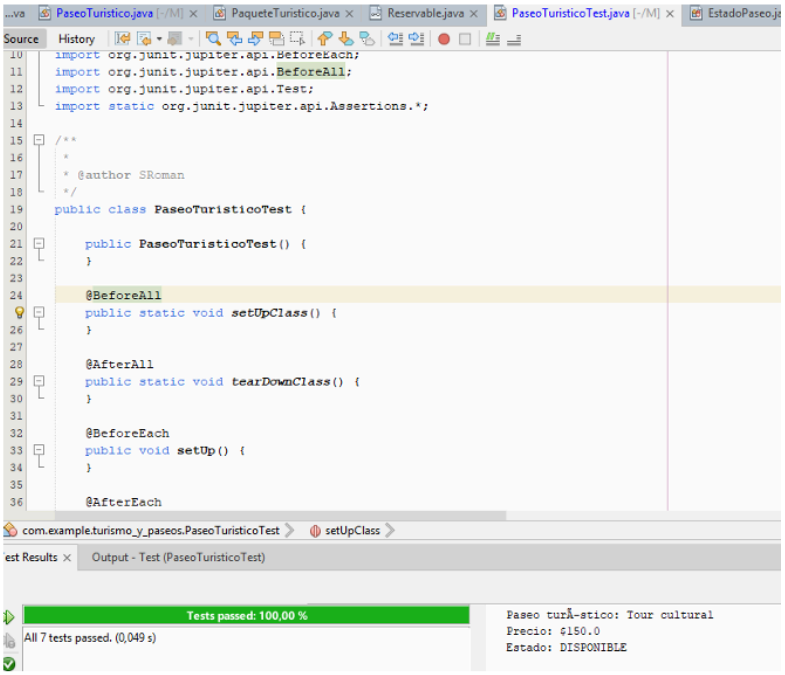


```java
private Hospedaje hospedaje;
private PaseoTuristico paseo;
private double descuento;

public PaqueteTuristico(Hospedaje hospedaje, PaseoTuristico paseo, double descuento) {
    this.hospedaje = hospedaje;
    this.paseo = paseo;
    this.descuento = descuento;
}

@Override
public double calcularPrecio() {
    if (hospedaje.getHabitacion() != null) {
        return hospedaje.getHabitacion().calcularPrecio() + paseo.calcularPrecio() - descu
    }
    else{ return  paseo.calcularPrecio() - descuento; }
}

@Override
public boolean verificarDisponibilidad() {
    return hospedaje.verificarDisponibilidad() && paseo.verificarDisponibilidad();
}
@Override
public void mostrarDetalles() {
    logger.info(msg: "Paquete turístico:");
    if (hospedaje != null) {
        hospedaje.mostrarDetalles();
    }
    if (paseo != null) {
        paseo.mostrarDetalles();
    }
    logger.info("Descuento aplicado: $" + descuento);
}
```

Pruebas unitarias:
Antes de refactoring



Despues de refactoring:



Refactoring:

Speculative Generality

Code Smell Corregido: Remove Dead Field
Antes:

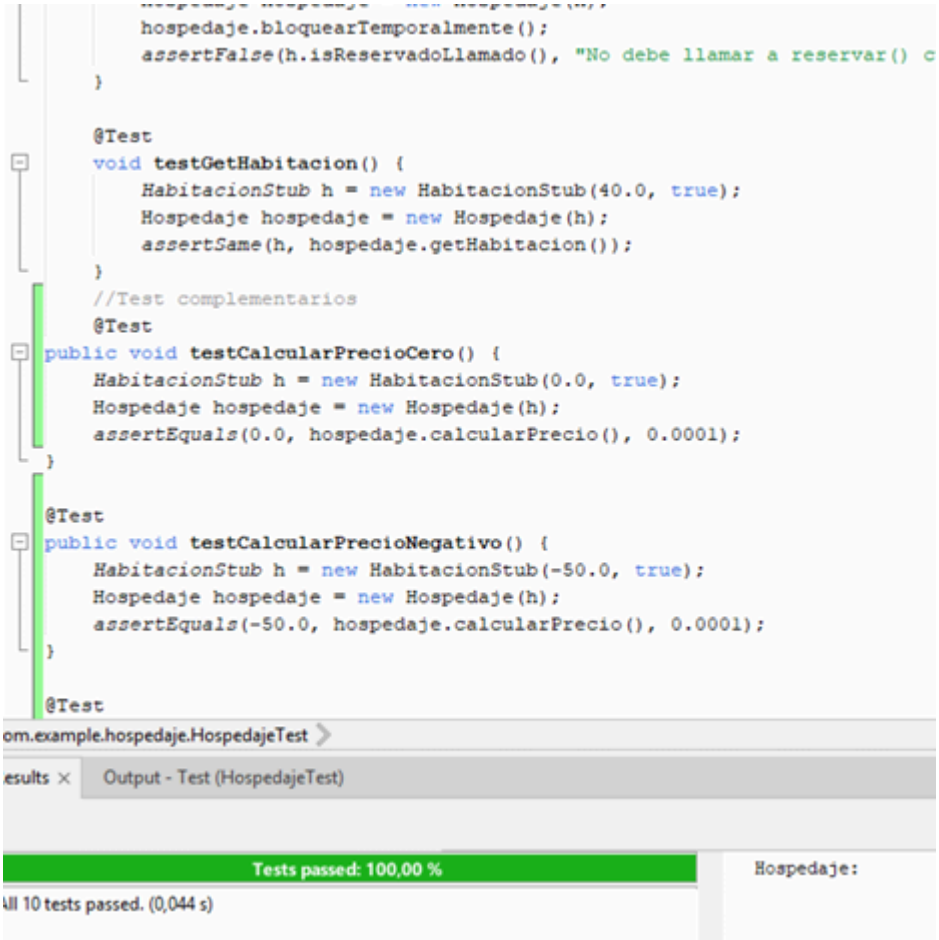
```
public class Hospedaje implements Reservable {
    private Hotel hotel;
```

Después:

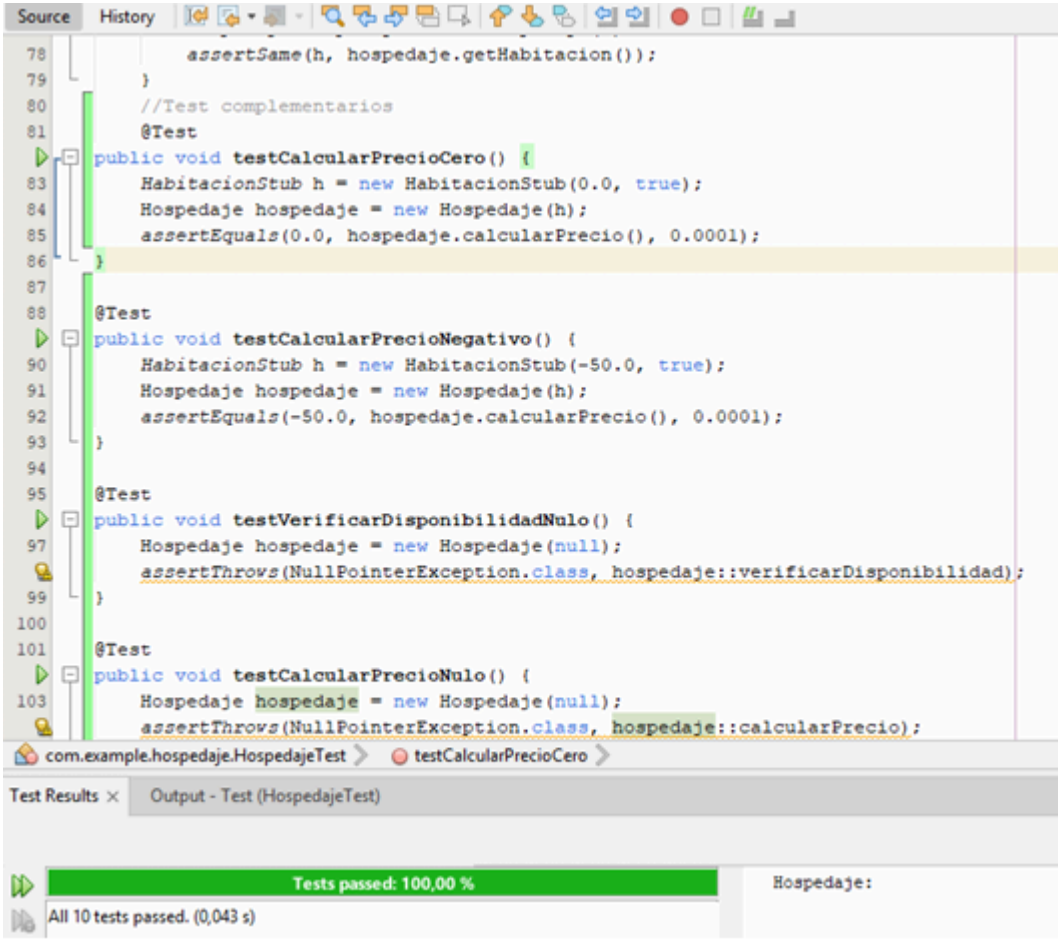
```
public class Hospedaje implements Reservable {
   private IHabitacion habitacion;

   public Hospedaje(IHabitacion habitacion) {
```

Justificación: Se eliminó el campo hotel de la clase Hospedaje, ya que no se inicializaba ni se utilizaba en ningún método. La clase ahora solo mantiene la referencia a IHabitacion, simplificando su estructura y evitando complejidad innecesaria.

Pruebas unitarias:
Antes de refactoring:



Despues de refactoring:



Refactoring:

Implementación de Método

Code Smell Corregido: Refused Bequest
Antes:

```java
public class HabitacionSuite extends Habitacion {
    public HabitacionSuite(int idHotel, int numero) {
        super(idHotel, numero);
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Habitación SUITE Nº " + numero + " - Hotel ID: " + idHotel + " -
    }

    @Override
    public double calcularPrecio() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException(message: "Unimplemented method 'calcularPreci
    }
```

Después:

```java
package com.example.patrones.factory;

import com.example.enums.TipoHabitacion;

public class HabitacionFamiliar extends Habitacion {
    public HabitacionFamiliar(int idHotel, int numero) {
        super(idHotel, numero, TipoHabitacion.FAMILIAR);
    }

    @Override
    public double calcularPrecio() {
        return 120.00;
    }
}
```

```java
package com.example.patrones.factory;

import com.example.enums.TipoHabitacion;

public class HabitacionSuite extends Habitacion {
    public HabitacionSuite(int idHotel, int numero) {
        super(idHotel, numero, TipoHabitacion.SUITE);
    }
```

Documentos\GitHub\Tarea2-
ple\patrones • Contains emphasized items

```java
    public double calcularPrecio() {
        return 180.00;
    }
}
```

Justificación:Se implementó el método calcularPrecio de las clases HabitaciónFamiliar y HabitaciónSuite según la lógica que manejan estas clases. El método ahora retorna un valor específico para cada clase, de modo que no es innecesaria su sobreescritura.

Pruebas unitarias:

```
    }

    @Test
    void reservarTestCambiaReservado() {
        HabitacionSuite suite = new HabitacionSuite(2, 201);
        suite.estado = EstadoHabitacion.OCUPADA;
        suite.reservar();
        Assertions.assertEquals(EstadoHabitacion.RESERVADA, suite.estado);
    }

    @Test
    void mostrarDetallesTestSinErrores() {
        HabitacionSuite suite = new HabitacionSuite(3, 303);
        suite.reservar();
        Assertions.assertDoesNotThrow(suite::mostrarDetalles);
        Assertions.assertEquals(EstadoHabitacion.RESERVADA, suite.estado);
    }

    @Test
    void estaDisponibleRetornaTrueSiDisponible() {
        HabitacionSuite suite = new HabitacionSuite(5, 501);
        Assertions.assertTrue(suite.estaDisponible());
    }

    @Test
    void calcularPrecioLanzaUnsupportedOperation() {
        HabitacionSuite suite = new HabitacionSuite(5, 501);
```

Refactoring:

Rename Method

Code Smell Corregido: Comments
Antes:

```
public void gestionarIncidente(Incidente incidente, GestorDeAccion gestorDeAcci
    if (incidente == null) {
        System.out.println(x: "Error: incidente es null. No se puede gestionar."
        return; // o lanzar una excepción controlada si quieres
    }

    if (gestorDeAccion == null) {
```

Después:

```java
public AgenteDeSoporte(int id, String nombre, Email correo) {
    super(id, nombre, correo);
}

public void gestionarIncidente(Incidente incidente, GestorDeAccion gestorDeAccion) {
    if (incidente == null) {
        logger.warning(msg: "Error: incidente es null. No se puede gestionar.");
        return;
    }

    if (gestorDeAccion == null) {
        logger.warning(msg: "Error: gestorDeAccion es null. No se puede obtener acción.");
        return;
    }
```

Justificación:Se renombró el método escalar() de la clase Incidente de modo que la interpretación de su funcionalidad sea mucho más sencilla. Además se eliminaron comentarios innecesarios de modo que el código es mucho más comprensible y sin tanto ruido visual.

Pruebas unitarias:

```java
@Test
void testGestionarIncidente_resolver_sinMockito() {
    Email correo = new Email("agente@mail.com");
    Usuario usuario = new Usuario(1, "Juan", correo);
    Incidente incidente = new Incidente(1, usuario, "Problema X");
    AgenteDeSoporte agente = new AgenteDeSoporte(2, "Agente1", correo);
    GestorDeAccion gestor = new GestorDeAccion() {
        public String obtenerAccion(Incidente inc) {
            return "resolver";
        }
    };
    agente.gestionarIncidente(incidente, gestor);
    Assertions.assertEquals(EstadoIncidente.RESUELTO, incidente.getEstado());
}

@Test
void testGestionarIncidente_escalar() {
    Email correo = new Email("agente2@mail.com");
    Usuario usuario = new Usuario(1, "Juan", correo);
    Incidente incidente = new Incidente(2, usuario, "Problema Y");
    AgenteDeSoporte agente = new AgenteDeSoporte(2, "Agente2", correo);
    GestorDeAccion gestor = new GestorDeAccion() {
        public String obtenerAccion(Incidente inc) {
            return "escalar";
        }
    };
    agente.gestionarIncidente(incidente, gestor);
```

Refactoring:

Inline Temp

Code Smell Corregido: Temporary Variable
Antes:

```java
@Override
public double calcularPrecio() {
    // Usar el precio decorado de la habitación
    double precioHabitacion = 0.0;
    if (hospedaje.getHabitacion() != null) {
        precioHabitacion = hospedaje.getHabitacion().calcularPrecio();
    }
    return precioHabitacion + paseo.calcularPrecio() - descuento;
}
```

Después:

```java
@Override
public double calcularPrecio() {
    if (hospedaje.getHabitacion() != null) {
        return hospedaje.getHabitacion().calcularPrecio() + paseo.calcularPrecio() - descu
    }
    else{ return  paseo.calcularPrecio() - descuento; }
}

@Override
```

Justificación: Se refactoriza el código eliminando la variable temporal precioHabitacion que sólo era utilizada para un retorno posterior sin alguna utilidad importante.

Pruebas unitarias:

```java
public class PaqueteTuristicoTest {
    public PaqueteTuristicoTest() {
    }

    @Test
    void testCalcularPrecio() {
        IHabitacion habitacion = new HabitacionTest(200.0, true);
        Hospedaje hospedaje = new Hospedaje(habitacion);
        PaseoTuristico paseo = new PaseoTuristico("Tour Cultural", 150.0);
        double descuento = 30.0;
        PaqueteTuristico paquete = new PaqueteTuristico(hospedaje, paseo, descuento);
        double esperado = 320.0;
        Assertions.assertEquals(esperado, paquete.calcularPrecio(), 0.01);
    }

    @Test
    void testVerificarDisponibilidad_true() {
        IHabitacion habitacion = new HabitacionTest(100.0, true);
        Hospedaje hospedaje = new Hospedaje(habitacion);
        PaseoTuristico paseo = new PaseoTuristico("Tour Aventura", 75.0);
        PaqueteTuristico paquete = new PaqueteTuristico(hospedaje, paseo, 0.0);
        Assertions.assertTrue(paquete.verificarDisponibilidad());
    }

    @Test
    void testVerificarDisponibilidad_false() {
        IHabitacion habitacion = new HabitacionTest(100.0, true);
        Hospedaje hospedaje = new Hospedaje(habitacion);
        new PaseoTuristico("Tour Aventura", 75.0);
        PaseoTuristico paseoNoDisponible = new PaseoTuristico("Tour Cancelado", 75.0) {
            public boolean verificarDisponibilidad() {
                return false;
```

Refactoring:

Remove Dead Code

Code Smell Corregido: Dead code
Antes:

```java
void reservar() {
    throw new UnsupportedOperationException(message: "Not supported yet.");
}
```

Después:

```java
    public void bloquearTemporalmente() {
        if (habitacion.estaDisponible()) {
            habitacion.reservar();
        }
    }

    public IHabitacion getHabitacion() {
        return habitacion;
    }

}
```

Justificación: Se eliminó el método reservar de la clase Hospedaje puesto que no tenía objetivo definido ni funcionalidad alguna dentro del código formando parte del code smell Dead Code. Al no tener lógica implementada, su eliminación elimina líneas innecesarias de código que no permitan una correcta interpretación del código.

Pruebas unitarias:

```java
import org.junit.jupiter.api.Test;

public class HospedajeTest {
    public HospedajeTest() {
    }

    @Test
    void testCalcularPrecio() {
        HabitacionStub h = new HabitacionStub(100.0, true);
        Hospedaje hospedaje = new Hospedaje(h);
        Assertions.assertEquals(100.0, hospedaje.calcularPrecio(), 1.0E-4);
    }

    @Test
    void testVerificarDisponibilidad() {
        HabitacionStub h = new HabitacionStub(50.0, true);
        Hospedaje hospedaje = new Hospedaje(h);
        Assertions.assertTrue(hospedaje.verificarDisponibilidad());
    }

    @Test
    void testBloquearTemporalmente_ReservarSiDisponible() {
        HabitacionStub h = new HabitacionStub(80.0, true);
        Hospedaje hospedaje = new Hospedaje(h);
        hospedaje.bloquearTemporalmente();
        Assertions.assertTrue(h.isReservadoLlamado(), "Debe llamar a reservar() cuando está disp
    }

    @Test
    void testBloquearTemporalmente_NoReservarSiNoDisponible() {
        HabitacionStub h = new HabitacionStub(80.0, false);
        Hospedaje hospedaje = new Hospedaje(h);
        hospedaje.bloquearTemporalmente();
        Assertions.assertFalse(h.isReservadoLlamado(), "No debe llamar a reservar() cuando no es
    }
```

Link del repositorio

URL: https://github.com/JosuePach3co/Tarea2-DS.git

Colaboracion: