Facultad de Ingeniería en Electricidad y Computación

Sistema de Alquiler de Hospedaje HomeStay

Tarea Grupal #3

Pruebas Refactoring

Estrada Lara Windsor Alexander

Defranc Rojas Francisco Etienne

Arenas Tituaña Juan Pablo

Gonzalez Prieto Lenier

Escuela Superior Politécnica del Litoral

Diseño de Software - Paralelo 1

Jurado Mosquera David Alonso

15 de enero de 2026

Índice

Sección A: Plan de Pruebas **3**
Chain of Responsibility 3
Factory Method 4
Facade 5
Strategy 7
Sección B: Implementación y Pruebas Unitarias **8**
Chain of Responsibility 8
Factory Method 9
Facade 9
Strategy 10
Sección C: Detección de Code Smells **11**
Speculative Generality 11
Data Class 12
Long Parameter List 13
Long Method 14
Message Chains 15
Lazy Class 16
Dead Parameter 17
Primitive Obsession 18
Duplicated Code 19
Middle Man 20
Sección D: Refactorización del Código **21**
Encapsulamiento de lógica de negocio en Data Class (Move Method) 21
Simplificación de creación de objetos con Long Parameter List (Introduce Parameter Object) 22
Descomposición de método monolítico (Extract Method) 23
Eliminación de dependencia indirecta (Hide Delegate) 24
Eliminación de clase innecesaria (Inline Class) 25
Limpieza de parámetros no utilizados (Remove Parameter) 26
Eliminación de código duplicado en jerarquía (Pull Up Method / Template Method) 27
Eliminación de intermediario innecesario (Remove Middle Man) 29
Reemplazo de Números Mágicos por Constantes (Replace Magic Number with Symbolic Constant) 30
Reemplazo de Primitivo por Objeto/Enum (Replace Primitive with Object) 31

Sección A: Plan de Pruebas

Chain of Responsibility

- Anfitrión: Atiende solicitudes con una prioridad de hasta nivel 2.
- Moderador: Gestiona casos con un rango de prioridad entre 3 y 5.
- Soporte Legal: Se encarga de todas las solicitudes de nivel 6 en adelante."

ID	Método	Datos de Entrada	Salida Esperada	Propósito
A01	canHandle (Anfitrión)	severity = 2	true	Verificar que Anfitrión maneja incidentes con severidad <= 2
A02	canHandle (Anfitrión)	severity = 3	false	Verificar que Anfitrión no maneja incidentes con severidad > 2
H01	canHandle (Moderador)	severity = 4	true	Verificar que Moderador maneja incidentes con severidad entre 3 y 5
H02	canHandle (Moderador)	severity = 6	false	Verificar que Moderador no maneja incidentes con severidad entre > 5
S01	canHandle (Soporte)	severity = 7	true	Verificar que Soporte Legal maneja incidentes con severidad > 5
S02	canHandle (Soporte)	severity = 5	false	Verificar que Soporte Legal no maneja incidentes con severidad <= 5
C01	handleIncident (Chain completa)	severity = 1	isResolved = true	Verificar que Anfitrión maneja incidente leve en cadena completa
C02	handleIncident (Chain completa)	severity = 4	isResolved = true	Verificar que Moderador maneja incidente medio en cadena completa
C03	handleIncident (Chain completa)	severity = 7	isResolved = true	Verificar que Soporte Legal maneja incidente grave en cadena completa
C04	handleIncident (Chain parcial)	severity = 9	isResolved = true	Demostrar que pasa al siguiente handler cuando el actual no puede manejar
C05	handleIncident (Chain incompleta)	severity = 5	isResolved = false	Verificar caso límite cuando ningún handler en cadena puede manejar

Factory Method

ID	Método	Datos de Entrada	Salida Esperada	Propósito
FM-01	crearUnidad()	ninguno	Instancia not null, tipo = “Casa”, precio base = 200, capacidad 6	CasaFactory crea la instancia correcta.
FM-02	crearUnidad()	ninguno	Instancia not null, tipo = “Habitación Privada”, precio base = 50, capacidad 2	HabitacionFactory crea la instancia correcta.
FM-03	crearUnidad()	ninguno	Instancia not null, tipo = “Departamento”, precio base = 120, capacidad 4	DepartamentoFactory crea instancia correcta.
FM-04	getDescripcion()	casa,departamento,habitación	todas not null y no vacías	Todas las unidades tienen descripción válida.
FM-05	crearUnidad()	CasaFactory invocado 2 veces	instancias diferentes	Verificar que cada llamada al factory crea una nueva instancia independiente.
FM-06	getPrecioBase(), getCapacidad()	casa,departamento, habitación	todos los valores>0	Validar que precios y capacidades son positivos.

Facade

ID	Método	Datos de Entrada	Salida Esperada	Propósito
DS-01	verificarDisponibilidad(Unidad)	unidad con estado “DISPONIBLE”	True	Verificar que el método retorna true cuando la unidad está disponible.
DS-01a	verificarDisponibilidad(Unidad)	unidad con estado “RESERVADA”	False	Verificar que el método retorna false cuando la unidad no está disponible.
DS-02	bloquearUnidad(Unidad)	unidad con estado “DISPONIBLE”	Estado = RESERVADA	Verificar que el método cambia el estado de la unidad a RESERVADA
POL-01a	verificarPoliticas(Unidad, Huesped)	huésped con calificación 4.5	True	Verificar que un huésped con calificación >=3.0 cumple las políticas.
POL-01b	verificarPoliticas(Unidad, Huesped)	huésped con calificación 2.0	False	Verificar que un huésped con calificación 3.0 cumple las políticas.
POL-02a	calcularPenalizacion(Reserva,Date)	reserva con cancelación 5 días antes	50% del total	Verificar que penalización del 50% cuando se cancela con menos de días de anticipación
POL-02b	calcularPenalización(Reserva,Date)	reserva con cancelación 20 días antes	25%	Verificar penalización del 25% cuando se cancela entre 7 y 0 días de anticipación.
POL-02C	calcularPenalización(Reserva,Date)	reserva con 35 días antes	0%	Verificar que no hay penalización cuando se cancela con más de 30 días de anticipación.
PR-01a	calcularPrecioTotal(Unidad, Date, Date)	5 días, precio base 120	600	Verificar que el precio total se calcula correctamente.
PR-01b	calcularDeposito(Unidad)	unidad con precio base 120	24	Verificar que el depósito se calcula como 20% del precio base.
PR-01c	calcularTarifasAdicionales(Unidad)	unidad con precio base 120	12	Verificar que las tarifas adicionales se calculan como 10% del precio base.
RF-01a	realizarReserva(Unidad, Huesped,	unidad disponible, huésped(cal.	Reserva creada (not	Verificar que se crea una reserva exitosamente cuando se cumple todos los requisitos.

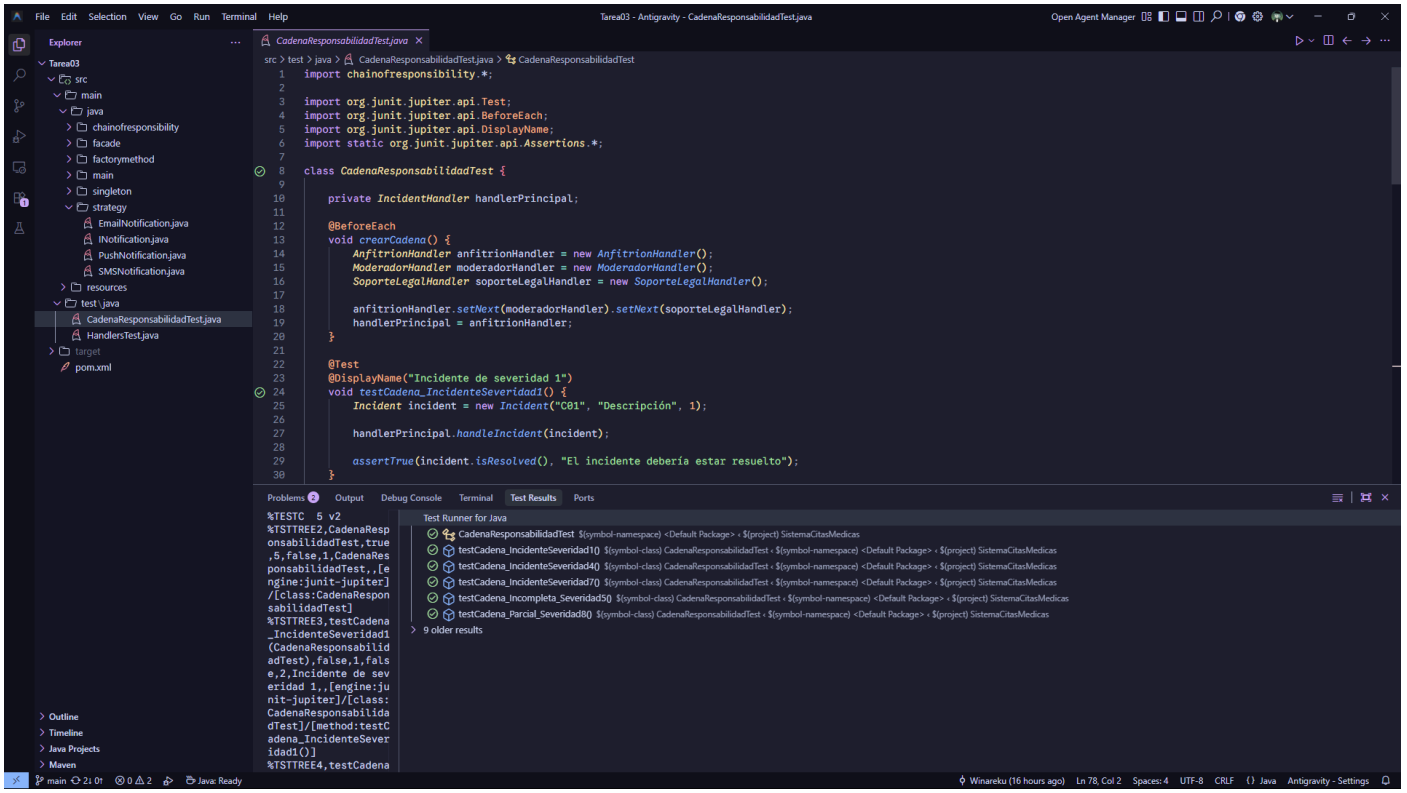
	Date, Date)	4.5) válido, fechas válidas	null), Estado unidad = RESERVADA, Precio total = 600. Reserva en lista del huésped	
RF-02a	realizarReserva(Unidad, Huesped, Date, Date)	unidad disponible, huésped inválido, fechas válidas	null	Verificar que se rechaza la reserva cuando el huésped no cumple políticas.
RF-02b	realizarReserva(Unidad, Huesped, Date, Date)	unidad RESERVADA, huésped válido, fechas válidas	null	Verificar que se rechaza la reserva cuando la unidad no está disponible.
RF-03a	cancelarReserva(Reserva)	reserva confirmada	Estado reserva = “CANCELADA”. Estado unidad = DISPONIBLE	Verificar que el estado de la reserva cambia a CANCELADA y estado vuelve a DISPONIBLE.
RF-04a	modificarReserva(Reserva, Date)	reserva existente, nueva fecha, unidad RESERVADA	False	Verificar que no se puede modificar cuando la unidad no está disponible.
RF-04b	modificarReserva(Reserva, Date)	reserva existente, nueva fecha, unidad DISPONIBLE	True, fechaInicio actualizada	Verificar que sí se puede modificar cuando la unidad está disponible y la fecha de inicio se actualiza correctamente.

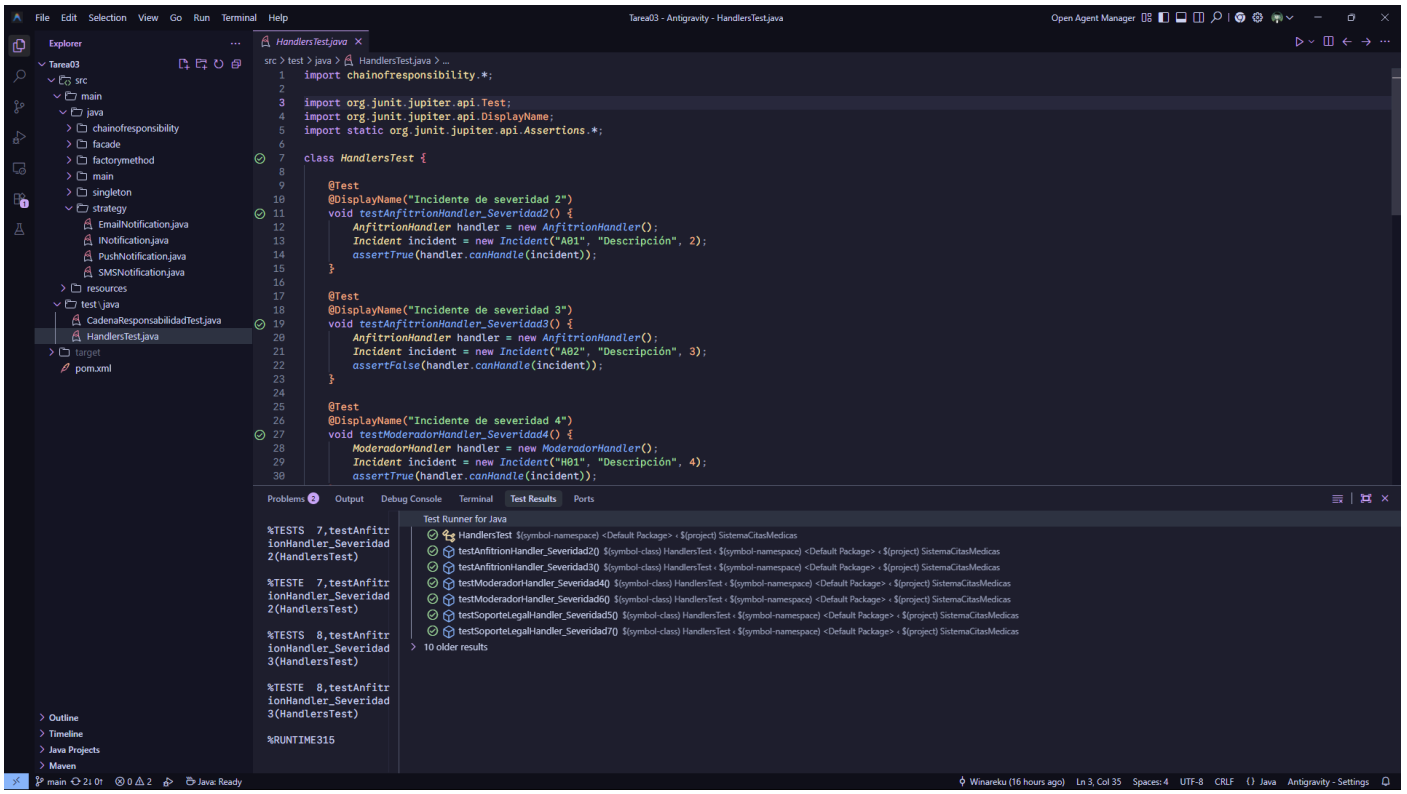
Strategy

ID	Método	Datos de Entrada	Salida Esperada	Propósito
N01	send (EmailNotification)	“Test Email” “test@test.com”	No hay excepción	Verificar que Email Notification envía mensajes correctamente
S01	send (SMSNotification)	“Test SMS” “+593987654321”	No hay excepción	Verificar que SMS Notification envía mensajes correctamente
P01	send (PushNotification)	“Test Push” “Dispositivo 01”	No hay excepción	Verificar que Push Notification envía notificaciones correctamente
I01	send (INotification)	Polimorfismo con Email Notification	Comportamiento específico de la clase	Verificar que la interfaz se implementa correctamente y que hay polimorfismo
I02	send (INotification)	Polimorfismo con SMS Notification	Comportamiento específico de la clase	Verificar que la interfaz se implementa correctamente y que hay polimorfismo
I03	send (INotification)	Polimorfismo con Push Notification	Comportamiento específico de la clase	Verificar que la interfaz se implementa correctamente y que hay polimorfismo

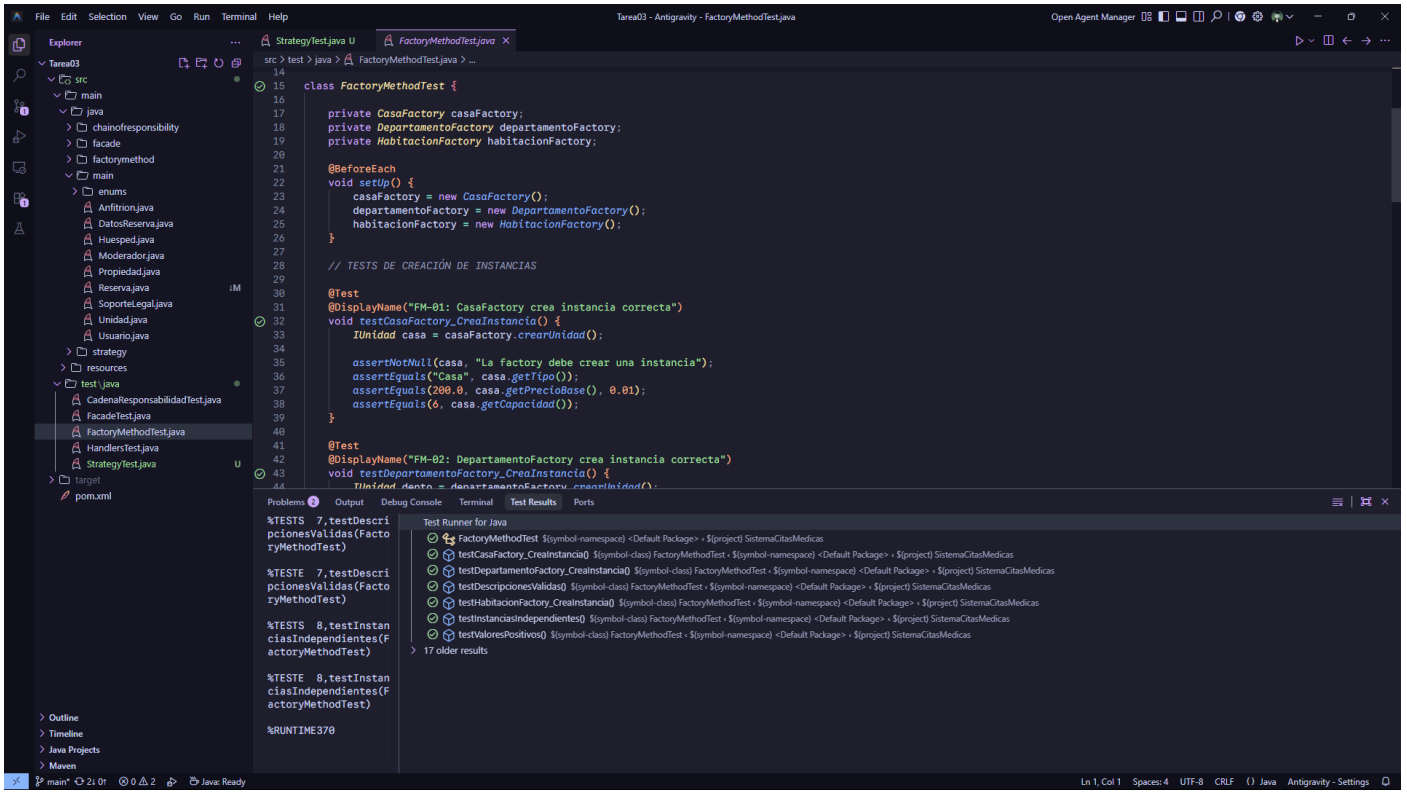
Sección B: Implementación y Pruebas Unitarias

Chain of Responsibility

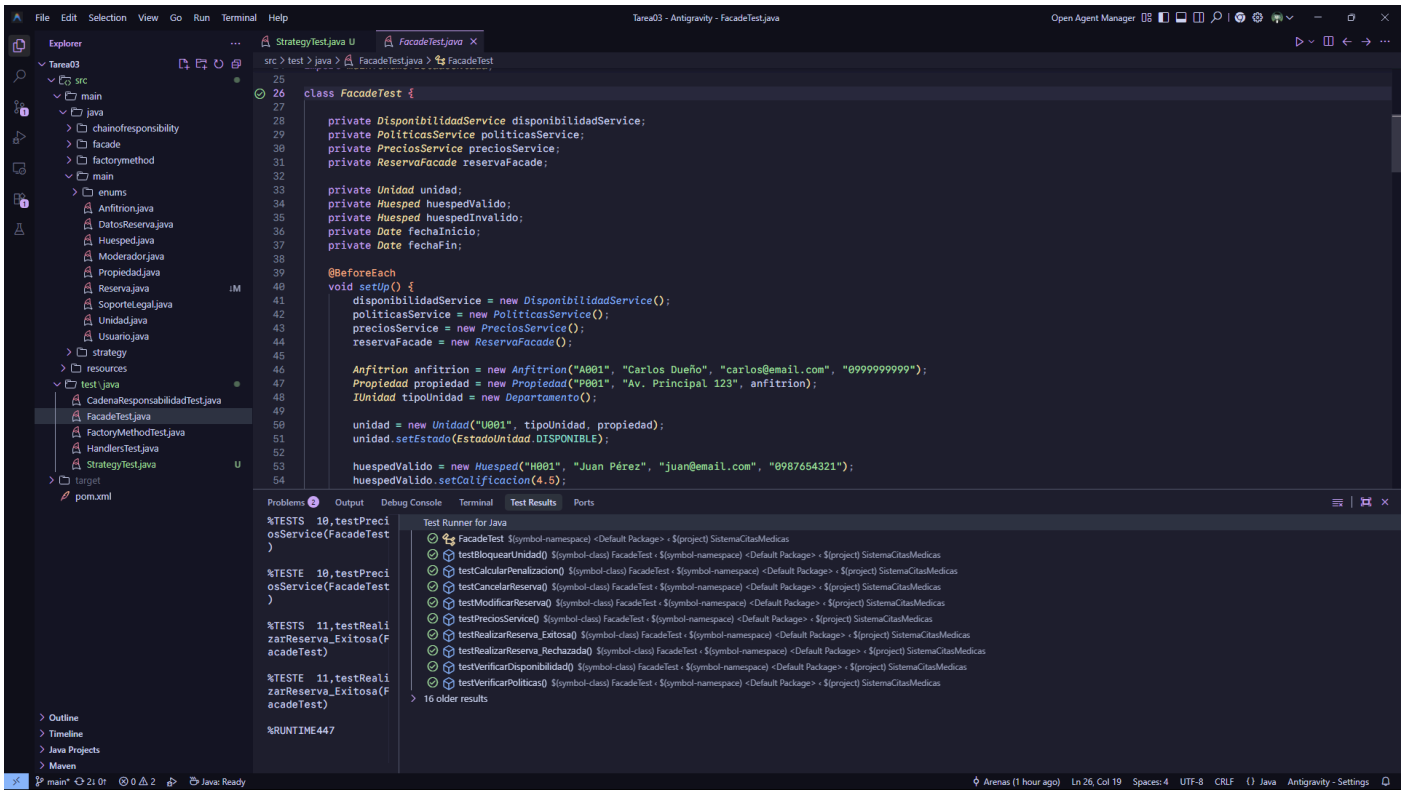




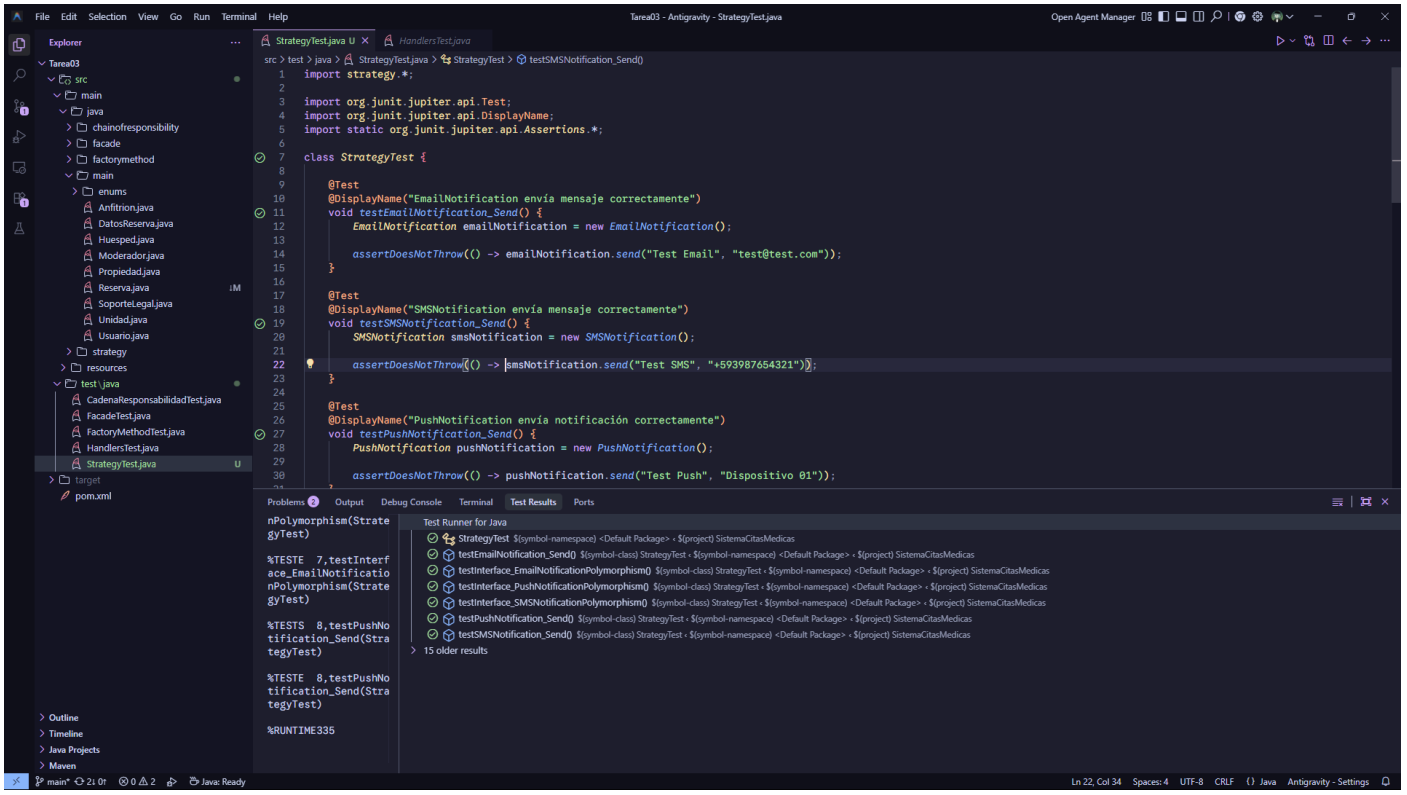
Factory Method



Facade



Strategy


File Edit Selection View Go Run Terminal Help
Tarea03 - Antigravity - StrategyTest.java
Open Agent Manager
Explorer
Tarea03
src
main
java
chainofresponsibility
facade
factorymethod
main
enums
Anfitrion.java
DatosReserva.java
Huesped.java
Moderador.java
Propiedad.java
Reserva.java
SoporteLegal.java
Unidad.java
Usuario.java
strategy
resources
test\java
CadenaResponsabilidadTest.java
FacadeTest.java
FactoryMethodTest.java
HandlersTest.java
StrategyTest.java
target
pom.xml
StrategyTest.java U
HandlersTest.java
src > test > java > StrategyTest.java > StrategyTest > testSMSNotification_Send()
import strategy.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
import static org.junit.jupiter.api.Assertions.*;
class StrategyTest {
@Test
@DisplayName("EmailNotification envía mensaje correctamente")
void testEmailNotification_Send() {
EmailNotification emailNotification = new EmailNotification();
assertDoesNotThrow(() -> emailNotification.send("Test Email", "test@test.com"));
}
@Test
@DisplayName("SMSNotification envía mensaje correctamente")
void testSMSNotification_Send() {
SMSNotification smsNotification = new SMSNotification();
assertDoesNotThrow(() -> smsNotification.send("Test SMS", "+593987654321"));
}
@Test
@DisplayName("PushNotification envía notificación correctamente")
void testPushNotification_Send() {
PushNotification pushNotification = new PushNotification();
assertDoesNotThrow(() -> pushNotification.send("Test Push", "Dispositivo 01"));
Problems 2
Output
Debug Console
Terminal
Test Results
Ports
nPolymorphism(StrategyTest)
%TESTE 7,testInterface_EmailNotificationPolymorphism(StrategyTest)
%TESTS 8,testPushNotification_Send(StrategyTest)
%TESTE 8,testPushNotification_Send(StrategyTest)
%RUNTIME335
Test Runner for Java
StrategyTest ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testEmailNotification_Send() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testInterface_EmailNotificationPolymorphism() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testInterface_PushNotificationPolymorphism() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testInterface_SMSNotificationPolymorphism() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testPushNotification_Send() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
testSMSNotification_Send() ${symbol-class} StrategyTest ‹ ${symbol-namespace} <Default Package> ‹ ${project} SistemaCitasMedicas
15 older results
Outline
Timeline
Java Projects
Maven
main*
Java: Ready
Ln 22, Col 34
Spaces: 4
UTF-8
CRLF
Java
Antigravity - Settings

Sección C: Detección de Code Smells

Speculative Generality

Usuario está declarado como abstract, pero no define comportamiento abstracto ni variaciones reales. Es una generalización innecesaria que complica el diseño.

Impactos:

Crear usuarios con datos inválidos fácilmente.

Validaciones repetidas en el código.

Mayor dificultad para mantener y extender por falta de encapsulación.

Ubicación:

espol/main/Usuario.java

Evidencia:

```java
public abstract class Usuario {
    protected String id;
    protected String nombre;
    protected String email;
    protected String telefono;

    public Usuario(String id, String nombre, String email, String telefono) {
        this.id = id;
        this.nombre = nombre;
        this.email = email;
        this.telefono = telefono;
    }

    public String getId() {
        return id;
    }

    public String getNombre() {
        return nombre;
    }

    public String getEmail() {
        return email;
    }

    public String getTelefono() {
        return telefono;
    }
}
```

Técnica de Refactorización:

Replace Inheritance with Delegation: usar composición o delegación en lugar de herencia.

Data Class

Incident actúa como bolsa de datos, con solo getters y poca lógica propia. Las reglas de negocio terminan dispersas fuera de la clase.

Impactos:

Reglas dispersas aumentan errores.

Baja cohesión y mayor acoplamiento.

Otros objetos manipulan su data directamente.

Ubicación:

chainofresponsibility/Incident.java

Evidencia:

```java
public class Incident {
    private String id;
    private String description;
    private int severity;
    private boolean resolved;

    public Incident(String id, String description, int severity) {
        this.id = id;
        this.description = description;
        this.severity = severity;
        this.resolved = false;
    }

    public String getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }

    public int getSeverity() {
        return severity;
    }

    public boolean isResolved() {
        return resolved;
    }

    public void markResolved() {
        this.resolved = true;
    }
}
```

Técnica de Refactorización:

Move Method: mover lógica relacionada al manejo del incidente dentro de la clase Incident.

Long Parameter List

El constructor de Reserva recibe demasiados parámetros, dificultando su uso y comprensión.

Impactos:

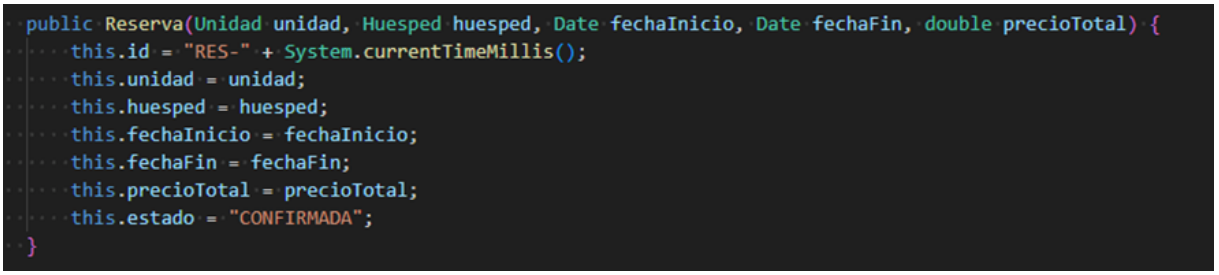
Mayor probabilidad de errores al llamarlo.

Menos legibilidad y difícil de extender.

Ubicación:

main/Reserva.java

Evidencia:



```
public Reserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin, double precioTotal) {
    this.id = "RES-" + System.currentTimeMillis();
    this.unidad = unidad;
    this.huesped = huesped;
    this.fechaInicio = fechaInicio;
    this.fechaFin = fechaFin;
    this.precioTotal = precioTotal;
    this.estado = "CONFIRMADA";
}
```

Técnica de Refactorización:

Introduce Parameter Object: agrupar parámetros relacionados en un objeto único.

Long Method

realizarReserva mezcla muchas responsabilidades en un solo método, haciéndolo difícil de leer y probar.

Impactos:|

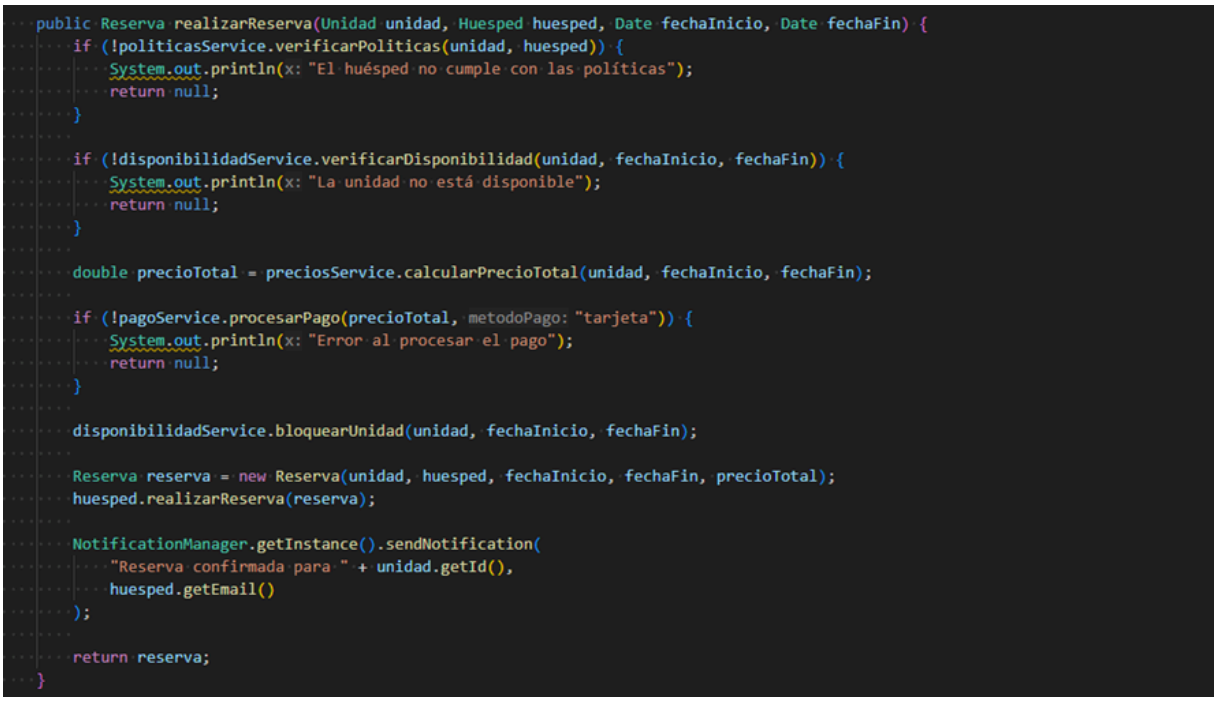
Difícil de leer y testear.

Cambios en una regla afectan todo el método.

Ubicación:

espol/facade/ReservaFacade.java

Evidencia:



```
public Reserva realizarReserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin) {
    if (!politicasService.verificarPoliticas(unidad, huesped)) {
        System.out.println(x: "El huésped no cumple con las políticas");
        return null;
    }

    if (!disponibilidadService.verificarDisponibilidad(unidad, fechaInicio, fechaFin)) {
        System.out.println(x: "La unidad no está disponible");
        return null;
    }

    double precioTotal = preciosService.calcularPrecioTotal(unidad, fechaInicio, fechaFin);

    if (!pagoService.procesarPago(precioTotal, metodoPago: "tarjeta")) {
        System.out.println(x: "Error al procesar el pago");
        return null;
    }

    disponibilidadService.bloquearUnidad(unidad, fechaInicio, fechaFin);

    Reserva reserva = new Reserva(unidad, huesped, fechaInicio, fechaFin, precioTotal);
    huesped.realizarReserva(reserva);

    NotificationManager.getInstance().sendNotification(
        "Reserva confirmada para " + unidad.getId(),
        huesped.getEmail()
    );

    return reserva;
}
```

Técnica de Refactorización:

Extract Method: separar pasos como validar, calcular, cobrar y notificar.

Message Chains

PreciosService usa cadenas de llamadas largas que aumentan el acoplamiento y fragilidad.

Impactos:

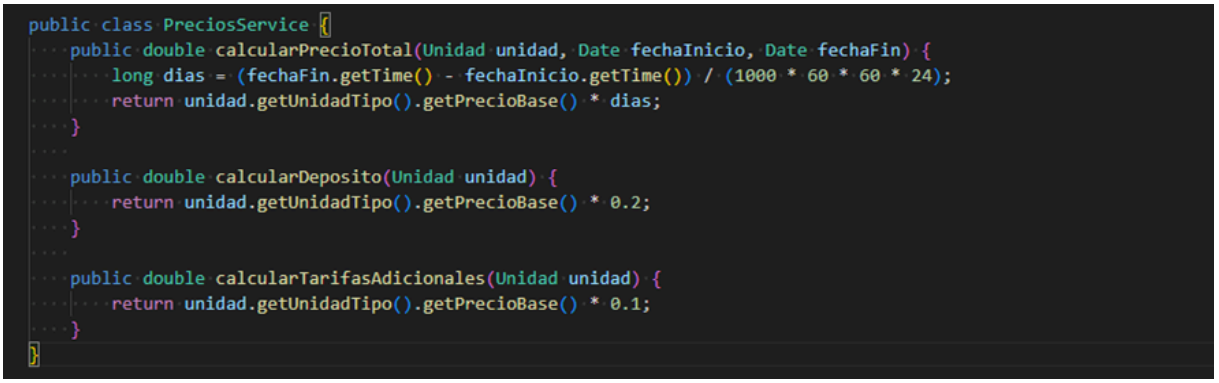
Mayor acoplamiento entre clases.

Cambios internos en Unidad afectan directamente al servicio.

Ubicación:

facade/PreciosService.java

Evidencia:



```java
public class PreciosService {
    public double calcularPrecioTotal(Unidad unidad, Date fechaInicio, Date fechaFin) {
        long dias = (fechaFin.getTime() - fechaInicio.getTime()) / (1000 * 60 * 60 * 24);
        return unidad.getUnidadTipo().getPrecioBase() * dias;
    }

    public double calcularDeposito(Unidad unidad) {
        return unidad.getUnidadTipo().getPrecioBase() * 0.2;
    }

    public double calcularTarifasAdicionales(Unidad unidad) {
        return unidad.getUnidadTipo().getPrecioBase() * 0.1;
    }
}
```

Técnica de Refactorización:

Hide Delegate: crear un método directo en Unidad para ocultar la cadena.

Lazy Class

PagoService es una clase con comportamiento mínimo, agregando capas innecesarias.

Impactos:

Clases sin valor real aumentan el mantenimiento.

Falsa sensación de lógica de pagos.

Ubicación:

facade/PagoService.java

Evidencia:

```
public boolean procesarPago(double monto, String metodoPago) {
    System.out.println("Procesando pago de $" + monto + " con " + metodoPago);
    return true;
}

public boolean procesarReembolso(double monto) {
    System.out.println("Procesando reembolso de $" + monto);
    return true;
}

public boolean cobrarDeposito(double monto) {
    System.out.println("Cobrando depósito de $" + monto);
    return true;
}
```

Técnica de Refactorización:

Inline Class: eliminar la clase y mover su lógica a donde se usa.

Dead Parameter

El método verificarDisponibilidad recibe parámetros que no usa, engañando al lector.

Impactos:

Engaña al lector sobre su funcionalidad.

Puede permitir reservas cruzadas por no validar fechas.

Ubicación:

facade/DisponibilidadService.java

Evidencia:

```
public class PoliticasService {
    public boolean verificarPoliticas(Unidad unidad, Huesped huesped) {
        return huesped.getCalificacion() >= 3.0;
    }
```

Técnica de Refactorización:

Remove Parameter: eliminar parámetros no usados.

Primitive Obsession

El estado de Reserva se maneja como String, permitiendo valores inválidos.

Impactos:

Errores por typos y validaciones repetidas.

Menos seguridad en el modelo.

Ubicación:

main/Reserva.java

Evidencia:

```
private String estado;

public Reserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin, double precioTotal) {
    this.id = "RES-" + System.currentTimeMillis();
    this.unidad = unidad;
    this.huesped = huesped;
    this.fechaInicio = fechaInicio;
    this.fechaFin = fechaFin;
    this.precioTotal = precioTotal;
    this.estado = "CONFIRMADA";
}
```

Técnica de Refactorización:

Replace Type Code with State Strategy: usar un objeto para representar el estado.

Duplicated Code

La lógica de procesar incidente se repite en varios handlers sin diferencias significativas.

Impactos:

Cambios requieren actualizar múltiples handlers.

Riesgo de inconsistencias.

Ubicación:

chainofresponsibility/ModeradorHandler.java

Evidencia:

```java
@Override
protected void process(Incident incident) {
    System.out.println("Moderador manejando incidente: " + incident.getDescription());
    incident.markResolved();
}
```

Técnica de Refactorización:

Pull Up Method: mover lógica repetida a la clase base.

Middle Man

NotificationManager solo delega llamadas sin añadir lógica, siendo una capa innecesaria.

Impactos:

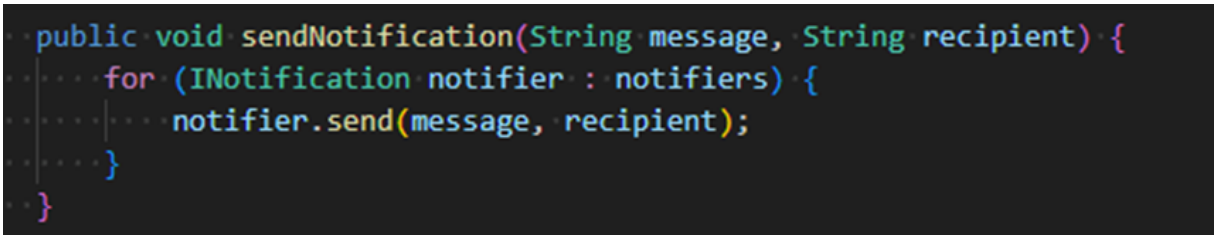
Capa extra sin valor añadido.

Aumenta el acoplamiento y complejidad.

Ubicación:

singleton/NotificationManager.java

Evidencia:



Técnica de Refactorización:

Remove Middle Man: eliminar el intermediario y llamar directamente a los notificadores.

Sección D: Refactorización del Código

Encapsulamiento de lógica de negocio en Data Class (Move Method)

Motivo:

La clase Incident actuaba como una Data Class (solo getters/setters), exponiendo sus datos internos (severity) para que otras clases (Handlers) tomaran decisiones. Esto violaba el principio de encapsulamiento.

Cambios realizados:

- **Incident.java:** Se agregaron métodos de negocio isLowSeverity(), isMediumSeverity() e isHighSeverity() para encapsular las reglas de clasificación de severidad.
- **AnfitrionHandler.java:** El método canHandle() ahora invoca incident.isLowSeverity() en lugar de comparar getSeverity() <= 2.
- **ModeradorHandler.java:** El método canHandle() ahora invoca incident.isMediumSeverity() en lugar de comparar rangos manuales.
- **SoporteLegalHandler.java:** El método canHandle() ahora invoca incident.isHighSeverity() en lugar de comparar getSeverity() > 5.

Resultado:

Lógica de clasificación centralizada en el dominio (Incident) y reducción del acoplamiento en los Handlers.

```
public boolean isLowSeverity() {
    return severity <= 2;
}

public boolean isMediumSeverity() {
    return severity > 2 && severity <= 5;
}

public boolean isHighSeverity() {
    return severity > 5;
}
```

Simplificación de creación de objetos con Long Parameter List (Introduce Parameter Object)

Motivo:

El constructor de Reserva recibía demasiados parámetros (5) que viajaban juntos (Data Clumps), afectando la legibilidad y el mantenimiento del código.

Cambios realizados:

- **DatosReserva.java:** Se creó una nueva clase Parameter Object que encapsula unidad, huésped, fechas y precio. Actúa como un DTO auxiliar, justificando su estructura de Data Class en este contexto.
- **Reserva.java:** El constructor ahora recibe un único objeto DatosReserva en lugar de cinco parámetros independientes.
- **ReservaFacade.java:** Se actualizó la creación de Reserva, instanciando previamente DatosReserva.

Resultado:

Código más limpio y manejo coherente de parámetros relacionados.

```java
public class DatosReserva {
    private Unidad unidad;
    private Huesped huesped;
    private Date fechaInicio;
    private Date fechaFin;
    private double precioTotal;

    public DatosReserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin, double precioTotal) {
        this.unidad = unidad;
        this.huesped = huesped;
        this.fechaInicio = fechaInicio;
        this.fechaFin = fechaFin;
        this.precioTotal = precioTotal;
    }
```

```java
public Unidad getUnidad() {
    return unidad;
}

public Huesped getHuesped() {
    return huesped;
}

public Date getFechaInicio() {
    return fechaInicio;
}

public Date getFechaFin() {
    return fechaFin;
}

public double getPrecioTotal() {
    return precioTotal;
}
```

Descomposición de método monolítico (Extract Method)

Motivo:

El método realizarReserva() presentaba el smell “Long Method”, concentrando múltiples responsabilidades en un solo flujo.

Cambios realizados:

- **ReservaFacade.java:** Se extrajeron métodos privados para encapsular responsabilidades específicas:
  - **cumpleRequisitos():** Validaciones de políticas y disponibilidad.
  - **procesarPago():** Lógica de cobro.
  - **finalizarReserva():** Bloqueo de la unidad, creación de la reserva y notificación.

Resultado:

realizarReserva() ahora funciona como un orquestador de alto nivel, mejorando la legibilidad y mantenibilidad del flujo principal.

```
private boolean cumpleRequisitos(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin) {
    if (!politicasService.verificarPoliticas(unidad, huesped)) {
        System.out.println("El huésped no cumple con las políticas");
        return false;
    }

    if (!disponibilidadService.verificarDisponibilidad(unidad)) {
        System.out.println("La unidad no está disponible");
        return false;
    }
    return true;
}

private boolean procesarPago(double precioTotal) {
    // Logica inlined de PagoService.procesarPago
    System.out.println("Procesando pago de $" + precioTotal + " con tarjeta");
    boolean pagoExitoso = true; // Simulación

    if (!pagoExitoso) {
        System.out.println("Error al procesar el pago");
        return false;
    }
    return true;
}

private Reserva finalizarReserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin,
        double precioTotal) {
    disponibilidadService.bloquearUnidad(unidad);

    DatosReserva datosReserva = new DatosReserva(unidad, huesped, fechaInicio, fechaFin, precioTotal);
    Reserva reserva = new Reserva(datosReserva);
    huesped.realizarReserva(reserva);

    notificationService.send("Reserva confirmada para " + unidad.getId(), huesped.getEmail());

    return reserva;
}
```

Eliminación de dependencia indirecta (Hide Delegate)

Motivo:

PreciosService presentaba el smell “Message Chain” al acceder a unidad.getUnidadTipo().getPrecioBase(), generando acoplamiento innecesario.

Cambios realizados:

- **Unidad.java:** Se agregó el método delegado getPrecioBase(), que redirige internamente a unidadTipo.getPrecioBase().
- **PreciosService.java:** Se modificaron las llamadas para utilizar directamente unidad.getPrecioBase().

Resultado:

Reducción del acoplamiento y ocultamiento de la estructura interna de Unidad.

```
public double getPrecioBase() {
    return unidadTipo.getPrecioBase();
}
```

Eliminación de clase innecesaria (Inline Class)

Motivo:

PagoService era una Lazy Class, con lógica mínima que no justificaba su existencia como componente independiente.

Cambios realizados:

- **ReservaFacade.java:** Se integró la lógica simple de pagos directamente en la fachada.
- **PagoService.java:** Se eliminó la clase.

Resultado:

Reducción de complejidad accidental y simplificación de la arquitectura.

```java
private boolean procesarPago(double precioTotal) {
    // Logica inlined de PagoService.procesarPago
    System.out.println("Procesando pago de $" + precioTotal + " con tarjeta");
    boolean pagoExitoso = true; // Simulación

    if (!pagoExitoso) {
        System.out.println("Error al procesar el pago");
        return false;
    }
    return true;
}
```

```java
public boolean cancelarReserva(Reserva reserva) {
    Date ahora = new Date();
    double penalizacion = politicasService.calcularPenalizacion(reserva, ahora);
    double reembolso = reserva.getPrecioTotal() - penalizacion;

    // Logica inlined de PagoService.procesarReembolso
    System.out.println("Procesando reembolso de $" + reembolso);
    boolean reembolsoExitoso = true; // Simulación

    if (reembolsoExitoso) {
        reserva.setEstado(EstadoReserva.CANCELADA);
        reserva.getUnidad().setEstado(EstadoUnidad.DISPONIBLE);
        return true;
    }
    return false;
}
```

Limpieza de parámetros no utilizados (Remove Parameter)

Motivo:

DisponibilidadService recibía parámetros de fechas (fechaInicio, fechaFin) que no eran utilizados, constituyendo Dead Code engañoso.

Cambios realizados:

- **DisponibilidadService.java:** Se eliminaron los parámetros no utilizados de los métodos verificarDisponibilidad() y bloquearUnidad().
- **ReservaFacade.java:** Se actualizaron las llamadas para coincidir con la nueva firma.

Resultado:

Firmas de métodos más honestas y alineadas con su comportamiento real.

```
public class DisponibilidadService {
    public boolean verificarDisponibilidad(Unidad unidad) {
        return unidad.getEstado() == EstadoUnidad.DISPONIBLE;
    }

    public void bloquearUnidad(Unidad unidad) {
        unidad.setEstado(EstadoUnidad.RESERVADA);
    }
}
```

Eliminación de código duplicado en jerarquía (Pull Up Method / Template Method)

Motivo:

Los handlers (AnfitrionHandler, ModeradorHandler, SoporteLegalHandler) duplicaban la lógica del método process().

Cambios realizados:

- **IncidentHandler.java:** Se implementó el método común process() (Pull Up Method) y se definió el método abstracto getHandlerName() (Template Method).
- **Subclases de Handler:** Se eliminó la lógica duplicada y se implementó únicamente getHandlerName().

Resultado:

Eliminación de redundancia y centralización del algoritmo de procesamiento en la clase base.

```java
public class AnfitrionHandler extends IncidentHandler {
    @Override
    public boolean canHandle(Incident incident) {
        return incident.isLowSeverity();
    }

    @Override
    protected String getHandlerName() {
        return "Anfitrión";
    }
}
```

```java
public class ModeradorHandler extends IncidentHandler {
    @Override
    public boolean canHandle(Incident incident) {
        return incident.isMediumSeverity();
    }

    @Override
    protected String getHandlerName() {
        return "Moderador";
    }
}
```

```java
public class SoporteLegalHandler extends IncidentHandler {
    @Override
    public boolean canHandle(Incident incident) {
        return incident.isHighSeverity();
    }

    @Override
    protected String getHandlerName() {
        return "Soporte Legal";
    }
}
```

Eliminación de intermediario innecesario (Remove Middle Man)

Motivo:

NotificationManager actuaba como un Middle Man que solo delegaba llamadas, además de ocultar dependencias al ser un Singleton.

Cambios realizados:

- **ReservaFacade.java:** Se modificó para interactuar directamente con la interfaz INotification, inicializada con EmailNotification.
- **NotificationManager.java:** Se eliminó la clase.

Resultado:

Arquitectura más simple y dependencias explícitas.

```
public class ReservaFacade {
    private DisponibilidadService disponibilidadService;
    private PreciosService preciosService;
    private PoliticasService politicasService;
    private INotification notificationService;
```

```
public class ReservaFacade {
    private DisponibilidadService disponibilidadService;
    private PreciosService preciosService;
    private PoliticasService politicasService;
    private INotification notificationService;
```

```
private Reserva finalizarReserva(Unidad unidad, Huesped huesped, Date fechaInicio, Date fechaFin,
        double precioTotal) {
    disponibilidadService.bloquearUnidad(unidad);

    DatosReserva datosReserva = new DatosReserva(unidad, huesped, fechaInicio, fechaFin, precioTotal);
    Reserva reserva = new Reserva(datosReserva);
    huesped.realizarReserva(reserva);

    notificationService.send("Reserva confirmada para " + unidad.getId(), huesped.getEmail());

    return reserva;
}
```

Reemplazo de Números Mágicos por Constantes (Replace Magic Number with Symbolic Constant)

Motivo:

El uso de literales numéricos sin significado explícito dificulta la comprensión del código y su mantenimiento.

Cambios realizados:

- **PreciosService.java:** Se definieron constantes como PORCENTAJE_DEPOSITO, PORCENTAJE_TARIFAS_ADICIONALES y MILISEGUNDOS_POR_DIA.
- **PoliticasService.java:** Se agregaron constantes como CALIFICACION_MINIMA, DIAS_PLAZO_CORTO y PENALIZACION_ALTA.

Resultado:

Código más legible, expresivo y con reglas de negocio claramente centralizadas.

```java
package facade;

import main.Unidad;
import java.util.Date;

public class PreciosService {
    private static final double PORCENTAJE_DEPOSITO = 0.2;
    private static final double PORCENTAJE_TARIFAS_ADICIONALES = 0.1;
    private static final int MILISEGUNDOS_POR_DIA = 1000 * 60 * 60 * 24;

    public double calcularPrecioTotal(Unidad unidad, Date fechaInicio, Date fechaFin) {
        long dias = (fechaFin.getTime() - fechaInicio.getTime()) / MILISEGUNDOS_POR_DIA;
        return unidad.getPrecioBase() * dias;
    }

    public double calcularDeposito(Unidad unidad) {
        return unidad.getPrecioBase() * PORCENTAJE_DEPOSITO;
    }

    public double calcularTarifasAdicionales(Unidad unidad) {
        return unidad.getPrecioBase() * PORCENTAJE_TARIFAS_ADICIONALES;
    }
}
```

Reemplazo de Primitivo por Objeto/Enum (Replace Primitive with Object)

Motivo:

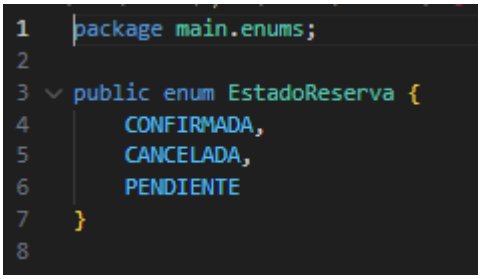
El estado de la reserva se representaba con cadenas de texto, lo cual era propenso a errores y carecía de seguridad de tipos (Primitive Obsession).

Cambios realizados:

- **EstadoReserva.java:** Se creó el enum EstadoReserva con los valores válidos.
- **Reserva.java:** El atributo estado cambió de String a EstadoReserva.
- **ReservaFacade.java:** Se actualizaron las asignaciones para utilizar el enum.

Resultado:

Mayor seguridad de tipos y eliminación de errores asociados a literales de texto incorrectos.



```
1  package main.enums;
2
3  public enum EstadoReserva {
4      CONFIRMADA,
5      CANCELADA,
6      PENDIENTE
7  }
8
```

Enlace del Repositorio

https://github.com/Winareku/Tarea03-PruebasRefactoring