Facultad de Ingeniería en Electricidad y Computación

Sistema de Alquiler de Hospedaje HomeStay

Tarea Grupal #2

Patrones de Diseño

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

19 de diciembre de 2025

Índice

Sección A: Justificación de Patrones

Singleton

Problema identificado:

En el sistema HomeStay, múltiples componentes necesitan enviar notificaciones a los usuarios, como confirmaciones de reserva, cancelaciones, incidentes y actualizaciones. Si cada componente crea su propia instancia del gestor de notificaciones, se generarían inconsistencias en la configuración de los canales de notificación, duplicación de recursos y dificultad para controlar qué notificaciones se envían y por qué canales. Además, sería imposible garantizar una gestión centralizada del estado de las notificaciones enviadas.

Justificación:

El patrón Singleton es ideal porque garantiza que exista una única instancia del NotificationManager en toda la aplicación. Proporciona un punto de acceso global para todos los componentes que necesiten enviar notificaciones, evita la creación múltiple de conexiones a servicios de notificación como email, SMS o push, facilita la configuración centralizada de los canales de notificación y permite mantener un registro coherente de todas las notificaciones enviadas.

Solución aportada:

Se implementó NotificationManager como Singleton. La clase mantiene una lista de notificadores, que son implementaciones de INotification, y proporciona métodos para agregar o remover canales dinámicamente. Esta solución aporta consistencia al garantizar que todas las notificaciones pasan por el mismo gestor, con formato y registro uniformes. También ofrece eficiencia de recursos al evitar la creación de múltiples instancias que consumen memoria y conexiones, y flexibilidad al permitir agregar o quitar canales de notificación en tiempo de ejecución sin afectar el código cliente. Además, mejora la mantenibilidad al concentrar la lógica de notificaciones en un único punto.

Strategy

Problema identificado:

El sistema necesita enviar notificaciones a través de diferentes canales, como email, SMS o notificaciones push, según las preferencias del usuario o la urgencia del mensaje. Si se implementara con condicionales como if-else o switch, el código se volvería rígido y difícil de mantener. Agregar nuevos canales requeriría modificar el código existente, violando el principio Open/Closed. Además, cada canal tiene su propia lógica de conexión, formato y envío.

Justificación:

El patrón Strategy es apropiado porque encapsula cada algoritmo de notificación, como email o SMS, en clases separadas. Permite cambiar el método de notificación en tiempo de ejecución, facilita agregar nuevos canales sin modificar código existente, elimina condicionales complejos y mejora la legibilidad. Cada estrategia es independiente y puede tener su propia implementación compleja.

Solución aportada:

Se definió la interfaz INotification con el método send() y se crearon implementaciones concretas como EmailNotification, SMSNotification y PushNotification. El NotificationManager mantiene una colección de estas estrategias y las ejecuta cuando es necesario. Esta solución aporta extensibilidad, ya que agregar nuevos canales como WhatsApp o Telegram solo requiere crear una clase que implemente INotification. Mejora la testabilidad, pues cada estrategia puede probarse independientemente, y ofrece flexibilidad para que los usuarios elijan sus canales preferidos o el sistema utilice diferentes canales según el tipo de mensaje. El mantenimiento se simplifica porque los cambios en la lógica de un canal no afectan a los demás.

Chain of Responsibility

Problema identificado:

Los incidentes reportados por huéspedes tienen diferentes niveles de severidad y requieren ser atendidos por diferentes niveles de personal, como anfitrión, moderador o soporte legal. Un incidente menor puede resolverse por el anfitrión, pero problemas graves requieren escalamiento. Sin un sistema estructurado, habría un acoplamiento fuerte entre el código que genera incidentes y el que los resuelve, y sería difícil modificar la cadena de escalamiento o agregar nuevos niveles.

Justificación:

El patrón Chain of Responsibility es ideal porque desacopla el emisor del incidente de los receptores que lo manejan. Permite que múltiples objetos tengan la oportunidad de manejar la solicitud, y la cadena puede modificarse dinámicamente agregando o removiendo handlers. Cada handler decide si puede manejar el incidente o debe pasarlo al siguiente, lo que implementa el escalamiento automático de forma natural.

Solución aportada:

Se creó la clase abstracta IncidentHandler con el método handleIncident() que implementa la lógica de cadena. Las clases concretas AnfitrionHandler, ModeradorHandler y SoporteLegalHandler implementan canHandle() y process() según la severidad, asignando niveles 1-3 al anfitrión, 4-7 al moderador y 8-10 al soporte legal. Esta solución permite un escalamiento automático, donde los incidentes van automáticamente al nivel apropiado sin lógica condicional compleja. Ofrece flexibilidad organizacional al facilitar la adición de nuevos niveles, como un supervisor, insertándolos en la cadena. Genera desacoplamiento, ya que el código que reporta incidentes no necesita conocer la estructura organizacional, y establece una responsabilidad clara con criterios definidos para cada handler.

Factory Method

Problema identificado:

El sistema maneja diferentes tipos de unidades de alojamiento, como habitaciones privadas, apartamentos y casas, cada una con características, precios y atributos distintos. La creación directa de estas unidades con new esparcirá la lógica de instanciación por todo el código, haría difícil cambiar cómo se crean las unidades y violaría el principio de responsabilidad única. Además, agregar nuevos tipos de alojamiento requeriría modificar múltiples puntos del código.

Justificación:

El patrón Factory Method es apropiado porque encapsula la lógica de creación de objetos complejos. Permite que las subclases decidan qué clase instanciar, proporciona una interfaz común, IUnidad, para trabajar con diferentes tipos, facilita agregar nuevos tipos de alojamiento sin modificar código cliente y centraliza la configuración inicial de cada tipo de unidad.

Solución aportada:

Se definió la interfaz IUnidad con métodos comunes y las clases concretas HabitacionPrivada, Departamento y Casa. Se creó la clase abstracta UnidadFactory con el método crearUnidad(), implementada por HabitacionFactory, DepartamentoFactory y CasaFactory, cada una configurando los valores apropiados. Esta solución aporta extensibilidad, ya que agregar nuevos tipos como estudios, villas y hostales solo requiere crear una nueva factory e implementación. Asegura consistencia, pues cada fábrica garantiza que las unidades se creen con valores válidos y consistentes. Utiliza polimorfismo para que el código cliente trabaje con IUnidad sin importar el tipo concreto y centraliza la configuración de precios base, capacidades y características en un solo lugar.

Facade Pattern

Problema identificado:

El proceso de realizar una reserva es complejo e involucra múltiples subsistemas: verificar disponibilidad, calcular precios y depósitos, validar políticas del anfitrión, procesar pagos, bloquear la unidad, crear la reserva y enviar notificaciones. Si el código cliente tuviera que interactuar directamente con todos estos servicios, habría alto acoplamiento, duplicación de lógica, y cualquier cambio en el flujo de reserva requeriría modificar múltiples partes del sistema. El cliente necesitaría conocer el orden correcto de llamadas y manejar errores de cada subsistema.

Justificación:

El patrón Facade es ideal porque proporciona una interfaz simplificada a un subsistema complejo. Reduce el acoplamiento entre el cliente y los múltiples servicios, encapsula el flujo de negocio completo de una reserva, facilita cambios en la implementación interna sin afectar clientes y centraliza la orquestación de múltiples operaciones relacionadas.

Solución aportada:

Se implementó ReservaFacade que coordina cuatro servicios: DisponibilidadService, PreciosService, PagoService y PoliticasService. La facade expone métodos simples como realizarReserva(), cancelarReserva() y modificarReserva() que internamente ejecutan el flujo completo en el orden correcto. Esta solución aporta simplicidad para el cliente, como el Huesped, que solo llama a facade.realizarReserva() sin conocer la complejidad interna. Facilita el mantenimiento del flujo, ya que cambiar el orden de validaciones o agregar nuevos pasos solo requiere modificar la facade. Asegura consistencia de negocio al garantizar que todas las reservas sigan exactamente el mismo proceso.

Enlace del Repositorio

https://github.com/Winareku/Tarea02-Patrones