Gestión del Hotel

Autores : Levi Barros García & Raúl Meijide Couto

Logins : levi.barrosg@udc.es & r.meijide@udc.es

Introducción

El documento pretende dar una visión detallada y comprehensiva del sistema de gestión de habitaciones implementado en el proyecto, enfocándose en el uso del Patrón de Diseño de Estado. Este sistema, diseñado para un entorno hotelero, permite la administración eficiente de habitaciones, desde su disponibilidad hasta su reserva, limpieza y liberación, a través de un enfoque modular y altamente adaptable.

Nuestros principios SOLID

- Principio de Responsabilidad Única (SRP):
  - Este principio establece que una clase debe tener solo una razón para cambiar. En el código, las clases Hotel, Habitacion, EstadoDisponible, EstadoReserva, y PendienteAprobacion tienen responsabilidades claramente definidas relacionadas con la gestión de hoteles y habitaciones. Cada clase tiene un propósito específico y no está sobrecargada con múltiples responsabilidades.
- Principio de Abierto/Cerrado (OCP):
  - El principio OCP establece que una clase debe estar abierta para la extensión pero cerrada para la modificación. En nuestro código, las clases de estado (EstadoDisponible, EstadoReserva, EstadoLimpieza, y PendienteAprobacion) pueden extenderse para agregar nuevos estados sin modificar el código existente. Esto se logra a través de la interfaz EstadosHabitacion y la implementación de cada estado en una clase separada.
- Principio de Sustitución de Liskov (LSP):
  - Este principio se refiere a que las instancias de una clase base deben poder ser reemplazadas por instancias de sus clases derivadas sin afectar la correctitud del programa. Las clases de estado (EstadoDisponible, EstadoReserva, EstadoLimpieza, y PendienteAprobacion) cumplen con este principio, ya que pueden ser utilizadas en lugar de la interfaz EstadosHabitacion sin introducir errores.
- Principio de Segregación de Interfaces (ISP):
  - Este principio establece que una clase no debe ser forzada a implementar interfaces que no utiliza. Las interfaces (EstadosHabitacion) están diseñadas de manera que cada estado implementa solo los métodos relevantes para

ese estado. Esto evita que las clases de estado tengan que proporcionar implementaciones vacías para métodos que no son aplicables.

- Principio de Inversión de Dependencias (DIP):
    - Este principio sugiere que los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. La dependencia entre las clases de estado (EstadoDisponible, EstadoReserva, EstadoLimpieza, y PendienteAprobacion) y la interfaz EstadosHabitacion sigue este principio. La Hotel depende de la interfaz, y no de implementaciones concretas.

Uso del Patrón Estado

Hemos usado este patrón estado debido a su eficacia en el uso de este problema. Con su uso nos permite cambiar el comportamiento del hotel según su estado sobre las habitaciones sin la necesidad de saber su estado. Este patrón tiene diversos beneficios como:

- Organización y Claridad:
    - El patrón Estado permite organizar el comportamiento de un objeto en diferentes estados de una manera clara y modular. Cada estado se representa mediante una clase separada, lo que facilita la comprensión y el mantenimiento del código.
- Extensibilidad:
    - El patrón Estado facilita la adición de nuevos estados al sistema sin modificar el código existente. Puedes introducir nuevas clases de estado sin afectar las clases que representan el contexto (por ejemplo, la clase Habitacion).
- Flexibilidad y Cambios Dinámicos:
    - El cambio dinámico de estados es una característica fundamental del patrón Estado. Permite que un objeto altere su comportamiento cuando cambia su estado interno. En tu caso, las transiciones entre los estados de una habitación (reserva, limpieza, etc.) pueden ocurrir en tiempo de ejecución.
- Cumplimiento de Principios SOLID:
    - La implementación del patrón Estado en tus clases (EstadoDisponible, EstadoReserva, EstadoLimpieza, PendienteAprobacion) cumple con los principios SOLID, como el Principio de Responsabilidad Única y el Principio de Abierto/Cerrado. Cada clase de estado tiene una responsabilidad clara y puede extenderse sin modificar otras partes del código.
- Facilita la Mantenibilidad:
    - La separación de los diferentes estados en clases independientes facilita la localización y corrección de errores. Además, facilita la introducción de nuevas funcionalidades o cambios en el comportamiento sin afectar el resto del sistema.
- Mejora de la Legibilidad:

    - La implementación del patrón Estado puede hacer que el código sea más legible y expresivo. Al utilizar clases para representar cada estado y métodos específicos para las transiciones entre estados, el código refleja de manera más clara el comportamiento del sistema en diferentes situaciones.

Se han implementado estos cambios de estados de manera coherente y estructurada, siguiendo el Patrón de Estado. Aquí se proporciona un comentario sobre cómo se manejan los cambios de estados en las diferentes situaciones:

- Reserva de Habitación (reservaHabitacion):
    - Cuando se solicita una reserva de habitación, se verifica primero si la habitación está disponible para reservar. Si la habitación está en el estado EstadoDisponible, se invoca el método reserveRoom del estado actual, que asigna un cliente a la habitación y cambia su estado a EstadoReserva.
- Finalización de Reserva (terminarReserva):
    - Al finalizar una reserva, se verifica si la habitación está en el estado EstadoReserva. Si es así, se invoca el método finishReservation del estado actual, que elimina la referencia al cliente y vuelve a establecer el estado a EstadoDisponible.
- Liberación de Habitación (liberarHabitacion):
    - Cuando se solicita la liberación de una habitación, se verifica si la habitación está en el estado EstadoReserva. Si es así, se invoca el método releaseRoom del estado actual, que realiza las acciones necesarias y cambia el estado a EstadoLimpieza.
- Limpieza de Habitación (limpiarHabitacion):
    - Para la limpieza de una habitación, se verifica si la habitación está en el estado EstadoLimpieza. Si es así, se invoca el método cleanRoom del estado actual, que puede cambiar el estado a PendienteAprobacion y asignar un trabajador de limpieza.
- Aprobación de Limpieza (aprobarLimpieza):
    - Al aprobar la limpieza, se verifica si la habitación está en el estado PendienteAprobacion. Si es así, se invoca el método approveCleaning del estado actual, que cambia el estado a EstadoDisponible después de la aprobación.

Estos cambios de estados están diseñados para reflejar de manera precisa el ciclo de vida de una habitación en un hotel. Se logra un flujo claro y lógico al delegar las responsabilidades específicas de cada estado a sus respectivas implementaciones. Este enfoque permite una fácil adaptación a futuros cambios en los requisitos del sistema y facilita la comprensión del comportamiento de las habitaciones en diferentes situaciones.