Informe Práctica de Diseño
1º Oportunidad 2023/2024

Alejandro Rodríguez Franco **alejandro.rodriguezf@udc.es**
Pablo Manzanares López **pablo.manzanares.lopez@udc.es**

1. **GESTIÓN DE HOTEL**

APLICACIÓN PRINCIPIOS SOLID

- **SINGLE RESPONSABILITY PRINCIPLE**

El Principio de Responsabilidad Única establece que una clase debe tener un único objetivo a la hora de desarrollar el código.

En nuestro programa, la clase Hotel se encarga únicamente de gestionar las listas de habitaciones, supervisores y miembros del equipo de limpieza, relacionándolas entre sí en funciones que llaman a la clase Room, la cual a su vez accederá a su status dependiendo de cada operación.

Cada clase tiene una responsabilidad, y se complementan entre ellas sin estorbar en el funcionamiento de las demás.

- **OPEN CLOSED PRINCIPLE**

Un código que cumple el Principio Abierto-Cerrado está abierto para una fácil extensión y cerrado para la modificación.

Esto se ve aplicado en la interfaz Status y las clases que la implementan (CleanApproved, BookedNotOccupied, Occupied, FreeWaitingClean, CleanWaitingApproval). Gracias a esta implementación en un futuro sería fácil añadir nuevos estados a las habitaciones, como Unavailable, InMaintenance... sin tener que cambiar el código ya existente, solo haría falta crear las clases nuevas implementando el interfaz Status y añadir las funciones pertinentes a la misma.

- ## LISKOV SUBSTITUTION PRINCIPLE

Según el Principio de Sustitución de Liskov implica que las subclases deben poder intercambiarse con sus clases base sin causar comportamientos indebidos.

Como se puede ver en la clase Room, las funciones dependen en la mayoría de los casos del Status de la clase, el cual puede cambiar con el tiempo. Para evitar que esto pueda dar problemas, las funciones utilizan la propia interfaz para realizar otras llamadas, aunque el comportamiento final dependerá de la subclase concreta que la implemente.

- ## INTERFACE SEGREGATION PRINCIPLE

Según el principio de Segregación de Interfaces, es mejor crear varias interfaces sencillas antes que una general de mayor complejidad, y que una clase implemente más de una interfaz en caso necesario.

Aunque en nuestro programa no es necesario usar más de una interfaz en ningún caso, la interfaz Status tiene un único propósito que cumple de forma sencilla, siendo que las clases que la implementen estarán encargadas de cambiar el estado de una habitación o no dependiendo de la acción a realizar y del estado actual de la misma.

- ## DEPENDENCY INVERSION PRINCIPLE

El Principio de Inversión de la Dependencia se basa en la arquitectura orientada a objetos en la que la mayoría de las clases dependen de abstracciones (interfaces o clases abstractas)

Podemos ver el uso de estas abstracciones sólo en la interfaz Status y las clases que la implementan, ya que en el resto de clases no hace falta su uso y podría complicar la legibilidad del código, aunque tampoco se descarta su posible modificación en un futuro para añadir nuevos casos, como distintos tipos de habitaciones, de hoteles, etc.

APLICACIÓN PRINCIPIOS DE DISEÑO

- ### PATRÓN INSTANCIA ÚNICA

Este patrón asegura que la clase en la que se aplique tendrá una única instancia y proporciona un acceso global a la misma.

En nuestro programa, este patrón lo aplicamos en 2 casos. El más importante es en la clase Hotel, ya que la intención del ejercicio es gestionar un único Hotel y tener más de una instancia del mismo podría dar lugar a situaciones inesperadas. Por otro lado, también se usan en las clases concretas que implementan la interfaz Status, como parte del Patrón Estado que se explica a continuación.

- ### PATRÓN ESTADO

Este es el patrón principal del ejercicio, y alrededor del cual se basa la gestión de las habitaciones del Hotel.

Funciona creando una interfaz Status y unas clases que la implementan. Esta interfaz establece unas funciones que establecen las diferencias entre cada posible estado, y las posibles acciones a realizar dependiendo del mismo. Cada subclase implementará estas funciones de una forma distinta, y pueden cambiar el propio Status de las habitaciones, así como los clientes que la reservan, ocupan, etc, además de poder limitar algunas acciones dependiendo del estado en el que se encuentre actualmente.

Al haber separado los posibles estados en una interfaz y unas clases, se facilita en un futuro añadir nuevos casos, tanto funciones nuevas para gestionar de distintas formas las habitaciones como nuevos estados que limiten o amplíen su uso.

2. INCURSIONES NAVALES

APLICACIÓN PRINCIPIOS SOLID

- **SINGLE RESPONSABILITY PRINCIPLE**

En este ejercicio se cumple que cada clase tiene su propia función y la cumple sin interponerse en el trabajo de otras. Por ejemplo, Sortie es la encargada de recorrer los conjuntos de nodos moviendo la flota a través de ellos, mientras que Node y sus subclases definen hacia donde irá la flota.

- **OPEN CLOSED PRINCIPLE**

Este principio se puede ver cumplido en la clase abstracta Node y sus subclases. Se pueden recorrer estos nodos independientemente de cuáles sean, y se pueden añadir nuevas clases como nuevos nodos con nuevas posibilidades sin necesidad de modificar el resto de nodos ya existentes.

- **LISKOV SUBSTITUTION PRINCIPLE**

Como se puede ver en Sortie, los recorridos dependen de nodos, pero no se especifica un tipo concreto de Node, ya estos se pueden intercambiar sin interrumpir la correcta ejecución del código

- **INTERFACE SEGREGATION PRINCIPLE**

Igual que en anterior caso, este programa no necesita de varias interfaces o clases abstractas, sino que con la clase Node llega para resolver la necesidad de establecer distintos comportamientos en base a unas funciones comunes

- **DEPENDENCY INVERSION PRINCIPLE**

De nuevo, gran parte del código depende de abstracción por la clase Node. Aunque ahora no hace falta, en un futuro podrían añadirse clases abstractas para añadir distintos tipos de barcos a una flota, crear clases distintas de flotas, etc.

APLICACIÓN PRINCIPIOS DE DISEÑO

- ## PATRÓN COMPOSICIÓN

La funcionalidad de este ejercicio está hecha a partir de este patrón, el cual permite intercalar distintos tipos de nodos de forma continua, las veces que queramos y sin que el programa cause ningún fallo.

Funciona creando la clase abstracta Node y otras clases concretas que extienden de esta. Node establece una serie de datos y de funciones que se especifican en las subclases, dando lugar a una serie de nodos con distintos comportamientos que pueden funcionar juntos. A la hora de recorrerlos en una Sortie, la flota avanzará al siguiente nodo por la función executeNode(), la cuál tiene distintos efectos dependiendo de la clase a la pertenezca, como reducir la vida de la flota o mandarla por un camino u otro dependiendo de ciertas características.

Además, este patrón facilita añadir en un futuro nuevas subclases de Node con nuevas funcionalidades, como nodos finales con jefes, batallas nocturnas, etc.