Alberto Paz Pérez
Luis Alfonso Rodríguez Rodríguez
12/12/2024

Informe DS - PD

EJERCICIO 1

En este apartado, hemos creado un sistema para controlar las distintas situaciones en las que pueden estar los buques de una flota. Estos buques pueden ser: ultraligeros, ligeros, pesados o ultrapesados, y a su vez pueden estar en diferentes estados: en base, en batalla, pendientes de reparación, en reparación o inactivos (hundidos u obsoletos). Este sistema permite al usuario cambiar entre los distintos estados en los que puede estar un buque, además de poder mostrar varios listados de los buques activos, inactivos o un seguimiento económico de la flota.

A continuación mostraremos los principios SOLID empleados:

- **Principio de Responsabilidad Única:** una responsabilidad es un factor de cambio, si se altera es necesario modificar el código que lo implementa. De esta manera, hemos implementado este principio para evitar la clase “Dios”, la cual estaría cargada y conllevaría un diseño frágil. En nuestro código, esa clase podría ser Buque perfectamente, pero lo solventamos de esta manera: los tipos de buque son clases hijas que heredan de Buque, los estados son implementaciones aparte de una interfaz Estados y la creación de los barcos se realiza en una clase aparte FactoriaBuque.

- **Principio Abierto-Cerrado:** en nuestra implementación, las clases y módulos son abiertos para permitir una posible extensión, pero cerrados a la modificación. Por ejemplo, tenemos clases hijas que heredan propiedades de sus progenitoras, en la clase FactoriaBuque dejamos la posibilidad de que el usuario pueda crear un nuevo tipo de buque y en Estados es posible añadir un nuevo estado sin modificar los existentes.

Por otro lado, hemos aplicado los siguientes patrones de diseño:

- **Patrón de Estados:** en este sistema estamos cambiando constantemente el estado interno de los buques: EnBatalla, EnBase, Inactivo (Obsoleto, Hundido), EnReparacion, PendienteReparacion. Así, cada buque puede actuar de una manera concreta dependiendo de su estado ante las distintas funciones del programa (hundir, empezarEjercicio, pedirReparacion…). Además, hemos definido los estados como *singletons,* para poderlos compartir entre distintos buques.

- **Patrón Método Factoría:** para abstraer el proceso de creación de buques, hemos añadido la clase FactoriaBuque, centralizando la lógica para instanciar los objetos. Según el enumerado TipoBuque, se creará un buque diferente, donde existe un caso default en nuestro switch para abarcar el caso de un nuevo tipo que se añada posteriormente.

EJERCICIO 2

En esta segunda parte, vamos a trabajar con un sistema de acciones del mercado bursátil, de las que nos interesa saber datos como su símbolo, volumen o la cotización máxima, mínima y de cierre de ese día. Los datos de estas acciones estarán a disposición de unos clientes, pero dependiendo de su tipo, sólo le interesará determinados datos según su perfil, para así facilitarle la visualización de la información. Por ejemplo, pueden haber clientes “sencillos” que sólo quieran ver la cotización de cierre de ese día u otros “detallados” que necesitan todos los datos

En esta ocasión, seguimos los siguientes principios SOLID de diseño:

- **Principio de Responsabilidad Única:** al igual que en el primer apartado, separamos las distintas clases del programa para repartir responsabilidades: los distintos tipos de clientes son implementaciones de la interfaz Cliente, Accion (que hereda de Sujeto) gestiona una lista de clientes que notifica a los clientes interesados cuando la información de esta cambia, InfoAccion separa la información de una acción bursátil y la creación de los Clientes se realiza en una clase aparte FactoriaCliente.

- **Principio Abierto-Cerrado:** el código está diseñado para facilitar la extensión sin modificar en exceso las clases existentes. Por ejemplo, si se quisiera agregar un nuevo tipo de cliente, se podría crear una nueva clase que implemente la interfaz Cliente sin tener que modificar la clase Accion ni las implementaciones existentes. También, si se quiere agregar un nuevo dato a una acción, tan sólo habría que modificar la clase infoAccion.

A mayores, aplicamos los siguientes patrones de diseño:

- **Patrón Método Factoría:** según un tipo (string), la clase FactoriaCliente crea objetos concretos de tipo Cliente. Hemos establecido un caso default donde, si no encaja el tipo con los que existen, creará un ClienteSencillo para no romper el programa.

- **Patrón Observador:** con este patrón, el objeto Accion actúa como un “sujeto”, que mantiene una lista de “observadores”, que son los clientes que implementan la interfaz Cliente. Cuando la información de una acción (InfoAccion) se actualiza, el sujeto notifica a todos los observadores interesados en ella. Esto permite separar la lógica de notificación (en el sujeto) de la lógica de actualización (en los observadores).