Informe Práctica 3

Tiago da Costa Teixeira Veloso e Volta

Pablo Herrero Diaz

Ejercicio 1 - Seguimiento de la Flota Naval

En lo que concierne a este ejercicio hemos tomado la decisión de utilizar **el patrón estado.** En principal medida, debido a que podemos encapsular un comportamiento específico del buque dentro de cada estado, es decir, dependiendo del estado, el buque tiene un comportamiento u otro. Además, este patrón permite eliminar la necesidad de usar múltiples estructuras condicionales (if-else o switch-case), fomentando así la claridad y simplicidad.

Debido al uso de este patrón, hemos decidido que el diagrama dinámico más adecuado para este ejercicio es el **diagrama de estados**, ya que nos permite entender en una mirada cómo funciona la comunicación entre estados en nuestro programa, junto con las condiciones, si las hay, que se han tenido en cuenta.

Por otra parte, como el problema requiere un diseño extensible para aceptar nuevos estados hemos tenido en cuenta el **principio abierto/cerrado**, así, por ejemplo, para añadir un nuevo estado, simplemente se crearía una nueva clase que implemente la interfaz EstadoBuque, sin modificar el código existente.

En cuanto al **principio de responsabilidad** única podemos ver que se cumple satisfactoriamente, ya que, cada estado maneja su propia lógica (encapsulamiento), lo que permite delegar el comportamiento de la clase buque al estado actual, mejorando así la cohesión.

Otro principio tenido en cuenta es el de **inversión de la dependencia**. Este principio lo podemos ver muy claramente en la dependencia entre Buque y EstadoBuque, ya que EstadoBuque se trata de una abstracción y no de una clase concreta, lo cual permite la flexibilidad al hacer un cambio de estado. Es decir, que dependemos de una interfaz en vez de depender de una clase o función lo que afirma la utilización de este principio.

Por último, pero no por ello menos importante, se cumple también el **principio de Sustitución de Liskov** en la implementación del patrón estado, ya que todas las implementaciones de EstadoBuque son intercambiables sin llegar a romper el comportamiento del programa, esto es debido a que las subclases estado respetan el contrato de la interfaz, garantizando así la consistencia y previsibilidad en las operaciones.

Cabe destacar que aparte del ya mencionado diagrama de estados, también se ha realizado el **Diagrama de Clases**, el cual juega un papel fundamental en este contexto, ya que actúa como un mapa visual que describe claramente las relaciones entre las clases, sus atributos, y los métodos asociados. En conclusión, el uso adecuado de patrones de diseño, principios de diseño orientado a objetos y herramientas como los diagramas de clases o dinámicos no solo mejora la calidad del software, sino que también asegura que el sistema sea adaptable y sostenible a largo plazo. Esto ha sido especialmente importante en este ejercicio, ya que la flota naval es un sistema complejo de gestionar, donde los estados y comportamientos están en constante cambio.

Ejercicio 2 - Cotización de acciones en el mercado bursátil

En este ejercicio hemos decidido usar 2 patrones, el primero patrón es **patrón observador**. Este patrón permite gestionar eficientemente la actualización automática de múltiples clientes, porque separamos las clases que generan los datos de las clases que los consumen (mercado bursátil de clientes), garantiza un bajo acoplamiento entre las clases y permite una fácil incorporación de nuevos clientes sin modificar el código existente. La clase *Accion* actúa como sujeto y la clase *Client* actúa como observador.

El segundo patrón utilizado es el **patrón estrategia**. Lo usamos para manejar la variabilidad en la forma en que los clientes procesan los datos del mercado, encapsula los diferentes tipos de cliente en clases independientes, promueve la reutilización y también facilita a extensión a nuevos tipos de clientes sin modificar los existentes. La clase *Client* crea los clientes y delega la presentación de los datos del mercado a una estrategia que implementa la interfaz *DisplayStrategy.*

Se han seguido varios principios de diseño para garantizar un sistema robusto, extensible y fácil de mantener. Se cumple el **Principio de Responsabilidad Única** porque cada clase tiene una única responsabilidad: *Accion* se encarga de gestionar la notificación de los clientes, mientras que *Client* crea los clientes y delega el comportamiento de presentación a las estrategias definidas. También se respeta el **Principio de Abierto/Cerrado***,* ya que el sistema permite la incorporación de nuevos tipos de clientes y estrategias sin modificar el código existente a través de *DisplayStrategy* y del switch en *Client*. Además, el diseño sigue **el *Principio de Sustitución de Liskov*** al permitir que cualquier clase que implemente *Observador* pueda interactuar con *Acciones* sin causar inconsistencias. Finalmente, se aplica el **Principio de Inversión de Dependencias** al depender de abstracciones como *Observador* y *DisplayStrategy* en lugar de implementaciones concretas, promoviendo flexibilidad y desacoplamiento.

En cuanto al diagrama dinámico para este ejercicio, decidimos elegir **diagrama de secuencia**. El primero, para mostrar la interacción entre los objetos principales del sistema en respuesta a los datos, destacando cómo fluyen los mensajes y eventos en respuesta a la actualización de datos y la notificación a los observadores. Se incluyen ciclos como la notificación iterativa de los observadores, el uso de excepciones en caso de datos inválidos y la delegación de la estrategia de presentación por parte del cliente, mostrando un flujo claro y detallado del comportamiento dinámico del sistema. En el segundo diagrama, decidimos representar específicamente el proceso de creación de un cliente. Este diagrama ilustra cómo el usuario invoca el método *createClient,* como se registra, cómo se genera una estrategia de visualización adecuada y si hay algún dato invalido retorna un *IllegalArgumentException*, antes de mostrar los datos específicos del tipo de cliente.