Informe

Principios Solid Usados

PRU (Principio de Responsabilidad Única)

Cada clase que usamos, debe tener una única responsabilidad.

> Por ejemplo, *DetailedClient, SimpleClient y SpecificStockClient* son clases que se encargan únicamente de procesas y mostrar datos de acciones en formas específicas. *Stock* se encarga de gestionar los estados de las acciones y notificar a los observadores de los cambios.

Este principio nos ayuda a, en un futuro, simplificar modificaciones y evitar errores.

AC (Abierto/Cerrado)

El sistema es extensible sin modificar el código actual, por lo cual podemos crear nuevos observadores heredando la interfaz *Observer* sin alterar las clases ya existentes.

> Esto se puede ver fácilmente pues todas las clases *Client* son añadidos a partir de *Stock*, la cual no se altera.

SDL (Sustitución de Liskov)

Las clases *Client* pueden alterar la implementación de *Observer* sin alterar el comportamiento de *Stock* gracias a que los observadores siguen la misma interfaz *Observer*.

SIP (Segregación de Interfaces)

La interfaz *Observer* define únicamente un método necesario (*update*), lo cual evita que los observadores deban implementar métodos innecesarios.

> Por ejemplo, todas las clases *Client* implementan sólo *update*, lo que les permite enfocarse en sus propias responsabilidades.

IDD (Inversión de Dependencias)

Las dependencias están invertidas porque *Stock* depende de la abstracción *Observer* en lugar de implementaciones concretas.

> Esto se ve claramente en que *Stock* no conoce detalles de *DetailedClient*, *SimpleClient* o *SpecificStockClient*, sólo trabaja con objetos que implementan la interfaz *Observer*.

Patrón de Diseño Observador

¿Qué hace?

El patrón Observador permite que un objeto (*Stock*) notifique automáticamente a múltiples dependientes (*Observers*) cuando su estado cambia.

> Elegimos este patrón porque nuestro sistema necesita transmitir cambios de estado en tiempo real a diferentes tipos de observadores. Esto se logra sin acoplar directamente *Stock* con las clases observadoras, manteniendo el sistema flexible y extensible.

Por ejemplo, cuando el estado de una acción cambia (a través de *updateStock*), *Stock* automáticamente llama a *notifyObservers*, que a su vez ejecuta *update* en cada observador.

Comparación con otros patrones:

• **Mediador**: Aunque también organiza interacciones, el mediador introduce un intermediario adicional entre *Stock* y los observadores, lo cual sería innecesariamente complejo para nuestro caso.
• **Comando**: Este patrón encapsula operaciones como objetos, pero no encaja con la necesidad de notificar eventos de forma automática y en tiempo real.