PRACTICA DS, CURSO 24-25

Manuel Taibo González, manuel.taibo2@udc.es

Martín Lopez Lodeiro, martin.lopez.lodeiro@udc.es

Grupo 1.2

PRINCIPIOS Y PATRONES DE DISEÑO UTILIZADOS EN EL EJERCICIO 1

1. **Responsabilidad Única (SRP):**

   - Cada clase tiene una responsabilidad clara:

   - `Buque` se encarga de almacenar y gestionar información del buque, como nombre, peso, estado y misiones completadas.

   - `BaseNaval` maneja los fondos disponibles de la base y las interacciones generales con los buques.

   - `EstadoBuque` y sus implementaciones (`EstadoActivo`, `EstadoEnEjercicio`, `EstadoEnReparacion`, etc.) se encargan del comportamiento específico de los buques en diferentes situaciones.

   - *Ejemplo*: La clase `Buque` no se preocupa por cómo se calculan los fondos o recompensas, solo gestiona los atributos relacionados con el buque.

2. Abierto/Cerrado (OCP):

   - El sistema está diseñado para permitir agregar nuevos estados de los buques sin modificar el código existente.

   - *Ejemplo*: Si se desea agregar un nuevo estado, como `EstadoCapturado`, basta con implementar la clase correspondiente sin modificar las clases existentes como `Buque` o `BaseNaval`.

3. Sustitución de Liskov (LSP):

   - Las implementaciones de la interfaz `EstadoBuque` pueden sustituirse entre sí sin afectar el comportamiento del sistema.

   - *Ejemplo:* Cuando un buque termina un ejercicio, su estado cambia de `EstadoEnEjercicio` a `EstadoActivo`. El código que interactúa con el buque no necesita saber qué tipo de estado tiene, solo interactúa con la interfaz `EstadoBuque`.

4. Segregación de Interfaces (ISP):

- La interfaz `EstadoBuque` contiene solo los métodos necesarios para los estados, evitando que las clases implementen métodos innecesarios.

- *Ejemplo:* La clase `EstadoHundido` no necesita implementar el método `finalizarReparacion`, ya que este no es relevante para un buque que ya ha sido hundido. Cada clase de estado implementa solo lo necesario para su comportamiento.

5. Inversión de Dependencia (DIP):

- Las clases de buques dependen de la abstracción `EstadoBuque` en lugar de depender de implementaciones concretas de estado.

- *Ejemplo:* La clase `Buque` no sabe qué tipo de estado tiene, solo sabe que implementa la interfaz `EstadoBuque`. Esto permite que el estado del buque se cambie fácilmente sin afectar el comportamiento de la clase `Buque`.

Patrón de Diseño: ESTADO

- Caso aplicado:

1. Cambio dinámico de comportamiento:

- Un buque cambia su comportamiento según su estado. Por ejemplo, cuando un buque en `EstadoActivo` inicia un ejercicio, su estado cambia a `EstadoEnEjercicio`. Este cambio se realiza sin condicionales en la clase `Buque`, ya que la lógica específica de cada estado está encapsulada en las clases de estado.

2. Delegación de lógica específica:

- La lógica de finalizar un ejercicio está en `EstadoEnEjercicio`. Al finalizar, este estado actualiza el número de misiones del buque, suma la recompensa a los fondos de la base y cambia el estado del buque a `EstadoActivo`. Esto evita que la clase `Buque` tenga que manejar estas reglas directamente.

3. Extensibilidad:

- Si se quiere agregar un nuevo estado, como `EstadoCapturado`, basta con crear una nueva clase que implemente la interfaz `EstadoBuque` y defina el comportamiento para ese estado. No se necesitan cambios en las clases existentes como `Buque` ni en las clases de los estados actuales.

El patrón Estado es útil en este contexto porque permite manejar el comportamiento cambiante de los buques de manera más organizada. Sin este patrón, se tendrían que usar condicionales complejos para verificar el estado de un buque y ejecutar la lógica correspondiente, lo que haría el código más difícil de mantener y entender.

PRINCIPIOS Y PATRONES DE DISEÑO UTILIZADOS PARA EL EJERCICIO 2

1. Responsabilidad Única (SRP):

- Cada clase tiene una responsabilidad clara:

- `ShareData` administra los datos y notifica cambios.

- `Subject` maneja los observadores.

- Los clientes (`SimpleClient`, `DetailedClient`, `CustomClient`) procesan y muestran datos específicos.

- *Ejemplo:* `ShareData` no se preocupa por cómo los datos son mostrados, solo por notificarlos.

2. Abierto/Cerrado (OCP):

- Nuevos clientes pueden agregarse sin modificar el sistema existente.

- Ejemplo: Si se crea un `GraphClient` para graficar datos, solo necesita implementar `Observer`.

3. Sustitución de Liskov (LSP):

- Los clientes concretos pueden sustituir a `Observer` sin afectar el sistema.

- Ejemplo: `notifyObservers()` trata a todos los observadores por igual, llamando su método `update()`

4. Segregación de Interfaces (ISP):

- `Observer` define solo un método (`update`), lo que simplifica la implementación para los clientes.

- Ejemplo: Cada cliente implementa únicamente lo necesario para procesar notificaciones.

5. Inversión de Dependencia (DIP):

- `ShareData` depende de la abstracción `Observer` y no de implementaciones concretas.

- Ejemplo: Cambiar `SimpleClient` por otro cliente no requiere cambios en `ShareData`.

Patrón de Diseño: OBSERVADOR

- Caso aplicado:

- `ShareData` notifica cambios automáticamente a los clientes registrados.

- Asegura que los datos mostrados en todos los clientes estén siempre actualizados sin necesidad de verificaciones manuales.

- *Ejemplo:* Si se actualiza el precio o volumen, `notifyObservers()` llama a `update()` en cada cliente.

- Modelo push:

- Cuando ocurre un cambio en el estado de `ShareData`, éste notifica automáticamente a los observadores (`SimpleClient`, `DetailedClient`, `CustomClient`) enviándoles directamente el objeto `ShareData` como argumento en el método `update()`.

- Los observadores no solicitan información activamente (como en el modelo pull), sino que reciben la notificación y los datos de manera inmediata.

- Los observadores reciben datos actualizados directamente sin necesidad de consultarlos manualmente.

- Reduce la carga en los observadores, ya que `Subject` se encarga de proporcionar los datos necesarios.