e1: Seguimiento de la Flota Naval

Índice

Domino del problema

El problema para el que debemos modelar una solución es representar la situación de una flota de la Marina, sus fondos y cada uno de sus barcos.

Hemos asumido que un barco solo se puede reparar a la vez, entendemos que solo tenemos un dique seco y que un barco que llega dañado ingresa igualmente la recompensa.

Clase y subclases Ship:

Representa al barco abstracto del que luego parten todos los barcos.

En la clase abstracta `ship` hemos decidido usar un enumerado, en lugar de crear subclases específicas. Simplificando así el diseño.

Este enfoque tiene varias ventajas:

- Simplificación de la jerarquía de clases.
- Fácil extensión.
- Centralización de la lógica común en la clase abstracta `Ship`.

Principio de Responsabilidad Única:

- La clase Ship maneja la lógica común para todos los barcos.
- El enumerado encapsula los tipos de barcos.
- UltraLightShip extiende la funcionalidad de Ship con la reparación exprés.

El barco ultraligero (UltraLightShip) aplica una herencia de extensión para añadir la funcionalidad de reparación express.

Esto facilita el mantenimiento del código.

Principio de Sustitución de Liskov:

`UltraLightShip`, se comporta como un `Ship` sin romper la funcionalidad de su clase padre. Esto significa que `cualquier instancia de UltraLightShip` debe poder `sustituir` una instancia de `Ship`.

Hemos decidido usar 2 boolean, uno para representar si en el actual estado del barco se considera activo (entendiendo activo como "listo para"/realizando una actividad, o no activo) y otro para esperando reparación.

La decisión de este enfoque fue por no tener que comprobar el estado y contrastar con una lista de estado si es activo.

Así un nuevo estado podría manejar si debe poner el barco en activo o si aparece una nueva cola de reparación, por ejemplo podemos querer añadir el estado dañado por torpedo sin tener que modificar el código del barco o base (`Principio de Abierto/Cerrado`).

NavalBase

Es el controlador. Tiene una flota, es decir una lista de barcos, utilizando el `Principio de inversión de la dependencia`. Dependiendo de abstracciones y no de concreciones. Implementando un ArrayList, se devuelve y declara como List, si el día de mañana se quiere usar otro tipo de lista sería un cambio mínimo.

Para las estadísticas financieras. Hemos añadido un campo para el presupuesto actual, además otros 2 con los gastos y los beneficios previstos.

Los datos previstos los modifican solo los estados (visibilidad de paquete). Nadie más debería poder modificarlo por seguridad.

Estados

La representación de los estados de los barcos, mediante la interfaz `state`. Utilizando el `patrón estado` e `instancia única`.

Para evitar, por ejemplo, que fuese posible cambiar el estado de un barco, sin el conocimiento de la base que lo administra, el estado recibe el barco a modificar y la base administradora.

Gracias al `patrón estado`, sería fácil añadir nuevos estados, por ejemplo el estado capturado por el enemigo.

Justificación del patrón Estado

A primera vista, podría parecer un problema adecuado para aplicar el Patrón Estrategia, ya que este facilita la adición de nuevas funcionalidades al permitir intercambiar algoritmos o comportamientos. Sin embargo, al analizar más a fondo, observamos que, por ejemplo la forma de reparar el barco no cambia. El `comportamiento permanece constante`. Lo que realmente `varía` son las `situaciones` por las que atraviesa el barco en su ciclo de vida.

Es aquí donde el Patrón `Estado` resulta más adecuado. Este nos permite modelar las transiciones entre los diferentes estados del barco, encapsulando la lógica de cada estado en clases independientes y delegando completamente las transiciones al propio sistema de estados. De este modo, el código se vuelve más modular, extensible y fácil de mantener, ya que el objeto principal no necesita gestionar la lógica interna de cada estado ni preocuparse por las transiciones.

Nuestra clase barco hace una llamada y simplemente cambia el estado. No tiene ni idea de como esto sucede.

Este patrón nos permite delegar toda la lógica de transición en los estados.

Principio de Responsabilidad Única:

Cada estado se encapsula en su propia clase, lo que significa que cada clase tiene una única responsabilidad. Gestionar el comportamiento asociado a un estado específico.

Principio de Abierto/Cerrado:

El barco (`ship`) no necesita ser modificado cuando se añaden nuevos estados o se cambia la lógica de transición. Puedes agregar nuevos estados o alterar el comportamiento existente extendiendo el sistema, sin tocar el código base del objeto o de los estados ya implementados.

Principio de Inversión de Dependencia

El objeto principal no depende de implementaciones concretas de los estados, sino de la abstracción que define el comportamiento de un estado. Esto facilita el intercambio de estados y promueve un `diseño desacoplado`.

Justificación patrón instancia única

Dado que cada estado representa un comportamiento bien definido y compartido, no es necesario crear múltiples instancias de los mismos. En lugar de ello, podemos aplicar el Patrón `Singleton` o `instancia única` para garantizar que cada estado exista como `una única instancia` a lo largo del ciclo de vida del programa.

Al no crear instancias repetitivas para los estados `reducimos el uso de recursos`.