Informe sobre Principios de Diseño y Patrones Aplicados

1. Principios de Diseño Usados (Principios SOLID)

1.1. Principio de Responsabilidad Única (SRP - Single Responsibility Principle)

Descripción: Este principio establece que una clase debe tener una única razón para cambiar, es decir, debe encargarse de una sola tarea o responsabilidad.

Aplicación en el proyecto:

Clase BaseNaval: Esta clase gestiona la flota (añadir, listar, cambiar estado de buques), controla los fondos (ingresos, gastos) y realiza la notificación de eventos. Aunque idealmente se podría dividir en clases más pequeñas, en la implementación actual, todas estas responsabilidades están contenidas en la clase BaseNaval.

Impacto: La clase BaseNaval tiene múltiples razones para cambiar (gestión de la flota, gestión de fondos y lógica de notificación), lo que puede afectar su mantenibilidad y escalabilidad. Dividirla en varias clases especializadas mejoraría la cohesión y la mantenibilidad.

1.2. Principio de Abierto/Cerrado (OCP - Open/Closed Principle)

Descripción: Las clases deben estar abiertas a la extensión pero cerradas a la modificación.

Aplicación en el proyecto:

Clase BaseNaval: La lógica de cálculo de recompensas de buques se basa en un switch-case, lo que viola el principio OCP. Cada vez que se añade un nuevo tipo de buque, se debe modificar la lógica del cálculo de recompensas.

Impacto: Actualmente, cada vez que se introduce un nuevo tipo de buque, hay que modificar la lógica de la clase BaseNaval, lo que afecta la extensibilidad. Usar una estrategia permitiría añadir nuevos tipos de buques sin modificar la clase existente.

1.3. Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle)

Descripción: Las subclases deben ser sustituibles por sus clases base sin alterar el comportamiento del programa.

Aplicación en el proyecto:

Clase Buque y sus subclases (DE, DD, CL, etc.): Todas las subclases de Buque pueden usarse donde se espera una instancia de la clase base Buque. Esto permite la sustitución sin errores ni alteraciones de comportamiento.

Impacto: Se cumple el principio LSP, ya que se pueden utilizar objetos de las subclases de **Buque** (por ejemplo, DE, DD, etc.) en cualquier parte donde se espere un **Buque**.

1.4. Principio de Segregación de Interfaces (ISP - Interface Segregation Principle)

Descripción: Las interfaces deben ser pequeñas y específicas, evitando que las clases tengan que implementar métodos que no necesitan.

Aplicación en el proyecto:

Interfaz Observador: La interfaz tiene un único método actualizar(String mensaje), lo que se ajusta perfectamente al principio ISP. Los observadores, como la clase Almirante, solo deben implementar este único método.

Impacto: La interfaz **Observador** es ligera y específica, cumpliendo con el principio ISP. Esta estructura facilita la inclusión de nuevos observadores sin afectar la clase principal **Buque**.

1.5. Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle)

Descripción: Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones.

Aplicación en el proyecto:

Clase BaseNaval: La clase BaseNaval crea instancias de **Reparacion** directamente usando new Reparacion(), lo que viola el DIP. La clase de alto nivel (BaseNaval) depende de la clase de bajo nivel (Reparacion).

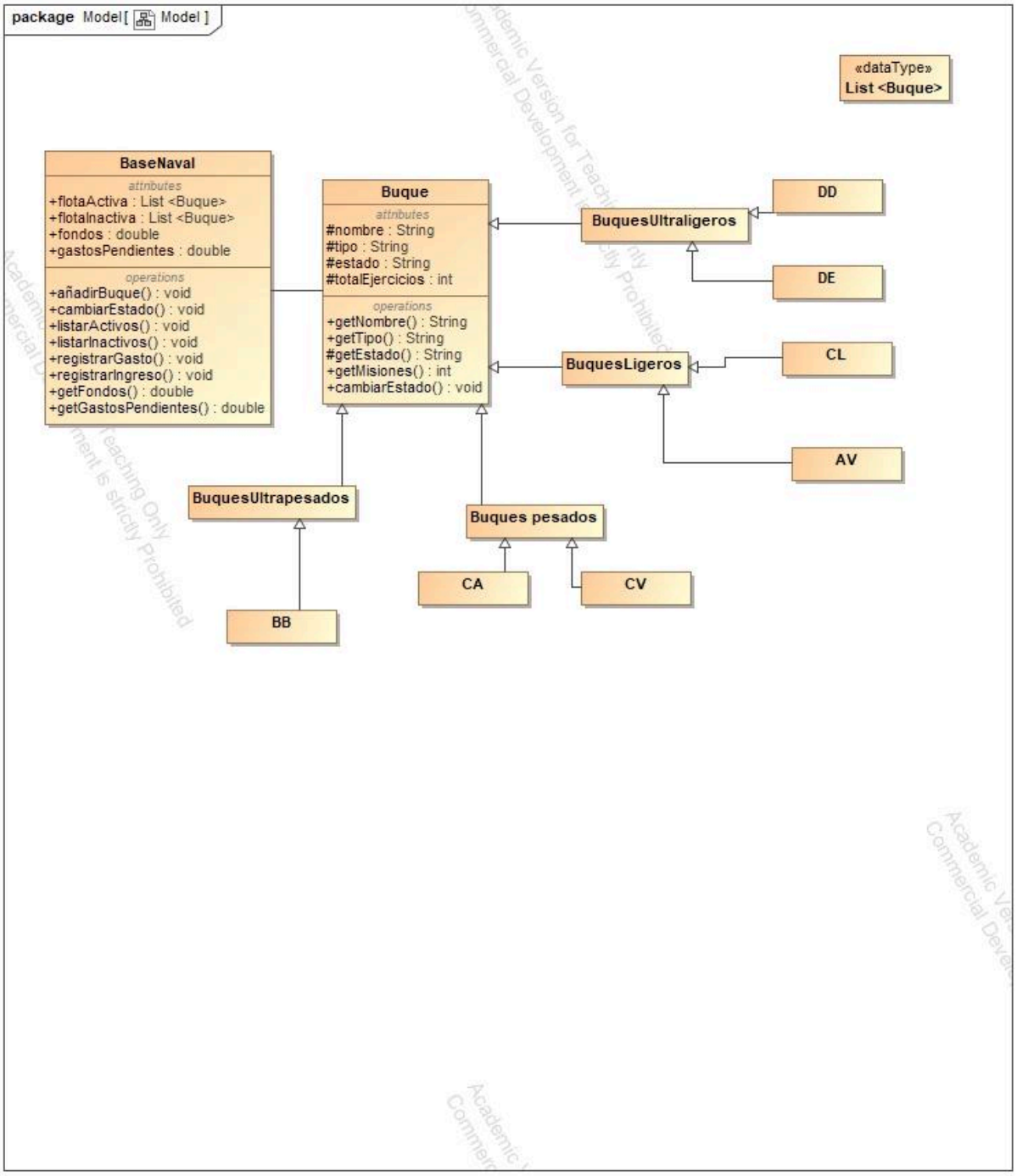
Impacto: El acoplamiento directo de **BaseNaval** con **Reparacion** hace que la clase sea difícil de modificar o sustituir. La inyección de dependencias permitiría cambiar la lógica de reparación sin modificar **BaseNaval**.

2. Patrones de Diseño Usados

2.1. Patrón Singleton

Clases: BaseNaval

Justificación de uso: La base naval debe ser única para toda la aplicación, por lo que se aplica Singleton para garantizar que solo exista una instancia de la clase **BaseNaval**.

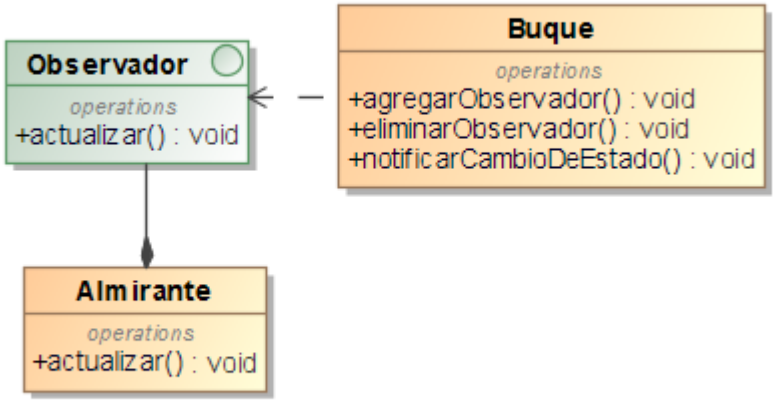


2.2. Patrón Observer

Clases: Buque, Almirante, Observador

Justificación de uso: Los observadores (por ejemplo, **Almirante**) deben ser notificados cuando cambia el estado de un buque.

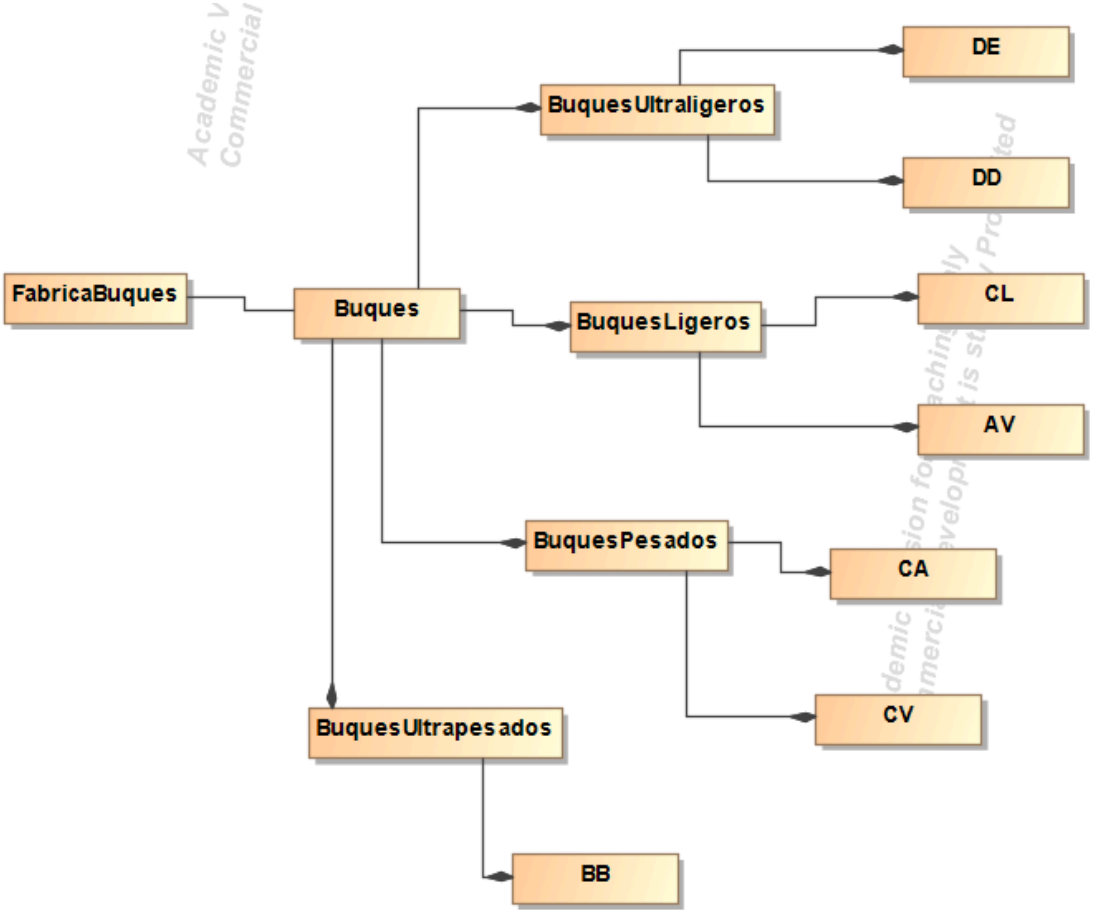
Impacto positivo: Permite la incorporación de nuevos observadores (por ejemplo, **Almirante**) para recibir notificaciones de cambios de estado de los buques, sin modificar la clase **Buque**.



2.3. Patrón Factory

Clases: FabricaBuques, Buque, DE, DD, etc.

Justificación de uso: Facilita la creación de buques de distintos tipos sin acoplar la lógica de construcción a la clase cliente. Aunque actualmente no se aplica, se recomienda usar este patrón para reducir el uso de switch-case.


DE
BuquesUltraligeros
DD
FabricaBuques
Buques
BuquesLigeros
CL
AV
BuquesPesados
CA
CV
BuquesUltrapesados
BB