Práctica 4

Equipo: Los Peaky Blinders

- Bonilla Reyes Dafne - 319089660
- Castañón Maldonado Carlos Emilio - 319053315
- García Ponce José Camilo - 319210536

1. **Menciona los principios de diseño esenciales de los patrones Factory, Abstract Factory y Builder. Menciona una desventaja de cada patrón.**

   ➔ **Principios de diseño esenciales:**

   - **Factory**

     Factory es un patrón de diseño creacional, es decir, está relacionado con la creación de objetos. Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán. En el patrón Factory, creamos objetos sin exponer la lógica de creación al cliente y el cliente usa la misma interfaz común para crear un nuevo tipo de objeto.

     En el patrón de diseño Factory se define una interfaz o una clase abstracta para crear objetos, dejando que las subclases decidan qué clase instanciar. Factory permite en la interfaz que una clase difiera la instanciación a una o más subclases concretas y promueve el acoplamiento flexible al eliminar la necesidad de vincular clases específicas de la aplicación en el código.

   - **Abstract Factory**

     El patrón de diseño Abstract Factory es uno de los patrones de creación que permite producir familias de objetos relacionados sin especificar sus clases concretas. Abstract Factory funciona en torno a una superfábrica que crea otras fábricas.

     La implementación de Abstract Factory nos da un marco que nos permite crear objetos que siguen un patrón general. Entonces, en tiempo de ejecución, la fábrica abstracta se combina con cualquier fábrica concreta deseada que puede crear objetos del tipo deseado.

     Como se mencionó anteriormente, Abstract Factory proporciona interfaces para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas, por lo que el software del cliente crea una implementación concreta de la fábrica abstracta y luego usa las interfaces genéricas para crear los objetos concretos que forman parte de la familia de objetos. El cliente no sabe ni está interesado en qué objetos concretos obtiene de cada una de estas fábricas concretas, ya que utiliza únicamente las interfaces genéricas de sus productos.

   - **Builder**

     Builder es un patrón de diseño creativo que permite construir objetos complejos paso a paso de tal manera que podamos producir diferentes tipos y representaciones de un objeto usando el mismo código de construcción, es decir, Builder permite separar la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones. De esta manera, se utiliza para construir un objeto complejo paso a paso y en paso final, devolver el objeto.

     El proceso de construcción de un objeto debe ser genérico para que pueda usarse para crear diferentes representaciones del mismo objeto.

   ➔ **Desventajas:**

   - **Factory**

     Una de las desventajas de Factory es que el código puede volverse complejo, ya que requiere de muchas clases y subclases nuevas para implementar el patrón, es decir, los clientes pueden tener que sub clasificar la clase en donde se implementa Factory solo para crear un objeto en particular, haciendo que extender su aplicación llegue a ser muy elaborado.

- **Abstract Factory**

  La principal desventaja de Abstract Factory es la admisión de nuevos tipos de "productos". Extender fábricas abstractas para producir nuevos tipos de productos no es fácil. Eso es porque la interfaz de Abstract Factory fija el conjunto de productos que se pueden crear. La compatibilidad con nuevos tipos de productos requiere ampliar la interfaz de fábrica, lo que implica cambiar esta clase y todas sus subclases, sin mencionar que la implementación del patrón puede hacer complejo al código, reduciendo la legibilidad del mismo debido al nivel de abstracción que introduce.

- **Builder**

  Una gran desventaja del patrón de diseño Builder es la complejidad general del código. El número de líneas de código aumenta al menos al doble en el patrón de construcción, ya que el patrón requiere la creación de varias clases nuevas, aunque el esfuerzo vale la pena en términos de flexibilidad de diseño y código mucho más legible.

2. **Uso del programa**

   – Compilar desde `src/`
   javac *.java Ships/*.java ShipComponents/*.java ShipComponents/ShipComponentsFactory/*.java
   – Ejecutar desde `src/`
   java Practica4
   – Generar la documentación desde `src/`
   javadoc -d docs *.java Ships/*.java ShipComponents/*.java ShipComponents/ShipComponentsFactory/*.java

   ➥ **Explicación:**

   Para iniciar el programa, primero es necesario compilar y ejecutar el programa. Después, los pasos para ordenar se mostrarán en terminal.

   Si se quiere generar la documentación, esto sería con el comando dado arriba, y luego, los archivos se generarán en el directorio llamado *docs*.

3. **Implementación**

   Para este proyecto decidimos usar los tres patrones de diseño, para (en nuestra opinión) facilitar como funcionan las cosas.
   Empecemos con Abstract Factory, usamos este patrón para poder crear fabricas para cada tipo de componente que tienen las naves, como hay varios tipos de componentes, y cada uno con diferentes versiones, pensamos que usar este patrón seria la mejor opción para así poder solo crear fabricas de cada tipo de componente y faciliar crearlos componentes que ya están definidos como se crean.
   Luego usamos Builder para poder construir las naves dependiendo de lo que los clientes quieran, elegimos este patrón ya que nos da la posibilidad de crear las naves por pasos, y así podemos preguntarle al cliente como quiere se cree su nave paso por paso. Además nuestros builders para las naves tienen un atributo de fabrica de componentes para poder construir los componentes que quiera el cliente.
   Y por ultimo usamos Factory para poder facilitarnos la creación de las naves del catalogo. Esto ya que, las naves del catalogo siempre serán las mismas, es decir, ya tenemos unos pasos para crearlas igual, por lo tanto decidimos que estas fabricas tuvieran un atributo de un constructor de naves, con lo cual podemos fabricar toda la nave con un solo método y sabemos que siempre saldrán naves iguales o ya predefinidas.