PRÁCTICA 03 - Modelado y Programación

Equipo "Better Code Saul"

Integrantes

Nombre	No. de cuenta
Álcantara Estrada Kevin Isaac	319073799
Cureño Sánchez Misael	418002485
Hernández Páramo Elizabeth	319143209

Instrucciones de compilacion

Forma 1

1. Dirigirse al directorio `src` de la práctica
2. Compilar usando

```
javac -sourcepath . -d ../target/ -cp .:../lib/* ./main/java/p04/Main.java
```

3. Copiar los recursos usando `cp -r ./main/resources ../target/main/`
4. Dirigirse al directorio generado `cd ../target`
5. Ejecutar usando `java -cp .:../lib/* main.java.p04.Main`

Forma 2 (Linux)

1. Ejecuta el script haciendo `./run.sh` o bien `bash run.sh` desde la terminal, dentro de la carpeta raiz de la práctica (en caso de no tener permiso de ejecución ejecutar `chmod +x run.sh`).

Instrucciones de formato estandarizado

1. En la raíz de la práctica ejecutar `make`.
2. En caso de que solo se quiera saber los errores de formato ejecutar `make check-format`.
3. En caso de que se quieran corregir los errores de formato ejecutar `make reformat`.

SECCIÓN TEÓRICA

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

FACTORY

Como se nos explicó en clase, el método Factory define una interfaz para crear un objeto permitiendo que las subclases decidan qué clase instanciar. El método factory permite que una clase difiera a la instalación a subclases.

ESTRUCTURA

- **Producto:** Declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.
- **Productos concretos:** Son distintas implementaciones de la interfaz de producto.
- **Clase creadora:** Declara el método fábrica que devuelve nuevos objetos de producto.Es importante que el tipo de retorno de este método coincida con la interfaz de producto.
- **Los creadores concretos:** Sobrescriben el método factory base, de modo que devuelva un tipo diferente de producto.

IMPLEMENTACIÓN

1. Hacemos que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.
2. Añadimos un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.
3. Encontramos todas las referencias a constructores de producto en el código de la clase creadora.
4. Creamos un grupo de subclases creadoras para cada tipo de producto enumerado en el método Factory.
5. Si hay demasiados tipos de producto y no tiene sentido crear subclases para todos ellos, podemos reutilizar el parámetro de control de clase base en las subclases.
6. Si, tras todas las extracciones el método factory base queda vacío, podemos hacerlo abstracto. Si queda algo dentro, podemos convertirlo en un comportamiento por defecto del método.

DESVENTAJA

- Puede que el código se complique, ya que debemos incorporar una multitud de nuevas subclases para implementar el patrón.

ABSTRACT FACTORY

El método Abstract factory proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

ESTRUCTURA

- **Productos abstractos:** Declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.

- **Productos concretos:** Son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto debe implementarse en todas las variantes dadas.
- **Interfaz Abstract factory:** Declara un grupo de métodos para crear cada uno de los productos abstractos.
- **Fábricas concretas:** Implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

IMPLEMENTACIÓN

1. Mapeamos una matriz de distintos tipos de productos frente a variantes de dichos productos.
2. Declaramos interfaces abstractas de producto para todos los tipos de productos. Después hacemos que todas las clases concretas de productos implementen esas interfaces.
3. Declaramos la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
4. Implementamos un grupo de clases concretas de fábrica, una por cada variante de producto.
5. Creamos un código de inicialización de la fábrica en algún punto de la aplicación.
6. Exploramos el código y encontramos todas las llamadas directas a constructores de producto. Las sustituimos por llamadas al método de creación adecuado dentro del objeto de fábrica.

DESVENTAJA

- El código puede que se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

BUILDER

Como vimos en clase, el patrón builder se utiliza para encapsular la construcción de un producto y permitir que se construya en pasos.

ESTRUCTURA

- **Interfaz constructora:** Declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común.
- **Constructores concretos:** Ofrecen distintas implementaciones de los pasos de construcción.
- **Productos:** Son los objetos resultantes. Los productos construidos por distintos objetos constructores no tienen que pertencer a la misma jerarquía de clases o interfaz.
- **Clase Directora:** Define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.
- **Cliente:** Debe asociar uno de los objetos constructores con la clase directora.

IMPLEMENTACIÓN

1. Nos aseguramos de poder definir claramente los pasos comunes de construcción para todas las representaciones disponibles del producto.
2. Declaramos estos pasos en la interfaz constructora base.

3. Creamos una clase constructora concreta para cada una de las representaciones de producto e implementa sus pasos de construcción.
4. Pensamos en crear una clase directora. Podemos encapsular varias formas de construir un producto utilizando el mismo objeto constructor.
5. El código cliente crea tanto el objeto constructor como el director. Antes de que empiece la construcción, el cliente debe pasar un objeto constructor al director. Normalmente, el cliente hace esto sólo una vez, mediante los parámetros del constructor del director. El director utiliza el objeto constructor para el resto de la construcción. Existe una manera alternativa, en la que el objeto constructor se pasa directamente al método de construcción del director.
6. El resultado de la construcción tan solo se puede obtener directamente del director si todos los productos siguen la misma interfaz. De lo contrario, el cliente deberá extraer el resultado del constructor.

DESVENTAJA

- La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.

¿Por qué escogimos Builder?

- Como el usuario iba a tener la libertad de "armar" la nave a su propio gusto mediante ciertas etapas, consideramos que había una compatibilidad directa con el patrón Builder, pues nos evita problemas en casos generales como cuando un usuario no puede pagar la nave y decide armar otra; además, vimos en él una forma de ahorrarnos la elaboración de clases con todas las posibles combinaciones de naves que se podían formar con lo componentes descritos.

Diagramas

modelos

util

Main

+static catalogoNaves: List<Nave>
+static builder: Builder

+static leerPresupuesto(): BigDecimal
+static elegirNave(): Nave
+static elegirArma(): Componente
+static elegirCabina(): Componente
+static elegirBlindaje(): Componente
+static elegirSistema(): Componente
+static mostrarCatalogo(presupuesto:BigDecimal): void

NaveBuilder

-nave: Nave

+getNave(): Nave

Builder

#arma: Componente
#blindaje: Componente
#cabina: Componente
#sistemaDePropulsion: Componente

+build(): Nave
+arma(a:Arma): Builder
+blindaje(b:Blindaje): Builder
+cabina(c:Cabina): Builder
+sistemaDePropulsion(s:SistemaDePropulsion): Builder

Nave

-ataque: int
-defensa: int
-velocidad: int
-peso: int
-arma: Componente
-blindaje: Componente
-cabina: Componente
-sistemaDePropulsion: Componente

+obtenerCosto(): BigDecimal
+calcularEstadisticas(): void

<<Interfaz>>
Componente

+nombre(): String
+descripcion(): String
+precio(): Double
+peso(): int
+velocidad(): int
+ataque(): int
+defensa(): int

Printer

-static error: AnsiFormat
-static warning: AnsiFormat
+static success: AnsiFormat
-static info: AnsiFormat
-static bold: AnsiFormat

+static println(s:String): void
+static print(s:String): void
+static println(String:s,style:AnsiFormat): void
+static error(s:String): void
+static warning(s:String): void
+static success(s:String): void
+static bold(s:String): void
+static info(s:String): void

Constantes

+static loadProperties(): Properties
+static property(propertyName:String): String
+redondear(cantidad:double): double

<<Abstract>>
Blindaje

BlindajeSimple
BlindajeReforzado
BlindajeFortaleza

<<Abstract>>
Cabina

CabinaTripulacionPequena
Cabina1Piloto
CabinaEjercito

<<Abstract>>
SistemaDePropulsion

ViajeIntercontinental
ViajeInterplanetario
ViajeIntergalactico

<<Abstract>>
Arma

ArmaLaserSimple
ArmaMisilesDePlasma
ArmaLaserDestructorDePlanetas