Taller 5 DPOO Samuel Peña

Proyecto: Se trata de un pequeño proyecto que ejemplifica de forma sencilla el uso del patrón *Factory Design* o *Factory Method Design* en una app que es capaz de dirigir la creación de equipos celulares de 3 diferentes marcas (Samsung, apple y Sony). En tal sentido, el corazón de la app puede considerarse como analogía a una fábrica real que en vez de crear diferentes clases de celulares físicos crea instancias de diferentes clases de celulares previamente definidas. A continuación se presenta el link del repositorio.

https://github.com/java9s/factory-pattern-example.git

EXPLICACIÓN Y CONTEXTO:

Su objetivo principal es proporcionar una interfaz para crear instancias de una clase, pero permite a las subclases alterar el tipo de objetos que se crearán. Este patrón cae bajo la categoría de patrones creacionales, que se centran en la forma en que las instancias de clases son creadas.

Elementos del patrón Factory:

1. Interfaz o Clase Base (Product): Define la interfaz para los objetos que el patrón de fábrica va a crear. Es la clase base que tiene un conjunto común de métodos que todas las subclases deben implementar.

2. Clases Concretas (Concrete Products): Son las implementaciones concretas de la interfaz o clase base. Estas clases son las que se instanciarán utilizando el patrón de fábrica.

3. Interfaz de Fábrica (Creator): Declara el método de fábrica abstracto que debe ser implementado por las subclases. Este método es responsable de crear un objeto de la clase base.

4. Fábricas Concretas (Concrete Creators): Son las implementaciones concretas de la interfaz de fábrica. Cada fábrica concreta produce un tipo específico de producto.

Motivación para su Uso:

- Desacoplamiento: El patrón Factory promueve el desacoplamiento entre la creación de objetos y su utilización. En lugar de que una clase cree directamente una instancia de otra clase, delega esta responsabilidad a una clase de fábrica. Esto hace que el código sea más flexible y fácil de mantener, ya que los cambios en la creación de objetos no afectarán directamente al código que los utiliza.

- Abstracción de la Creación: Permite a una clase delegar la responsabilidad de instanciar sus objetos a sus subclases. Esto es útil cuando una clase no puede anticipar la clase de objetos que debe crear.

- Configuración Centralizada: Facilita la gestión centralizada de la creación de objetos. Si necesitas cambiar la implementación concreta de un objeto en toda tu aplicación, solo necesitas hacerlo en un lugar: la fábrica correspondiente.

Aplicación del patrón en el proyecto y aparición del mismo:

El patrón se aplica en el proyecto alrededor de la clase *Mobile Factory*, que permite crear instancias de diferentes clases que las subclases pueden alterar en cuanto a tipo. De esa forma, sirve como puente entre el objeto solicitante (Exec) y las fabricas específicas (Sony, IPhone y Samsung).

La interfaz *Mobile*, si bien no tiene métodos creacionales definidos (estos están en la clase *Mobile factory*) contiene los tres tipos de celulares que pueden crearse, y de ella heredan las 3 fábricas estos tipos de donde elegir. De esta forma, esta interfaz constituiría el primer elemento *producto* descrito anteriormente.

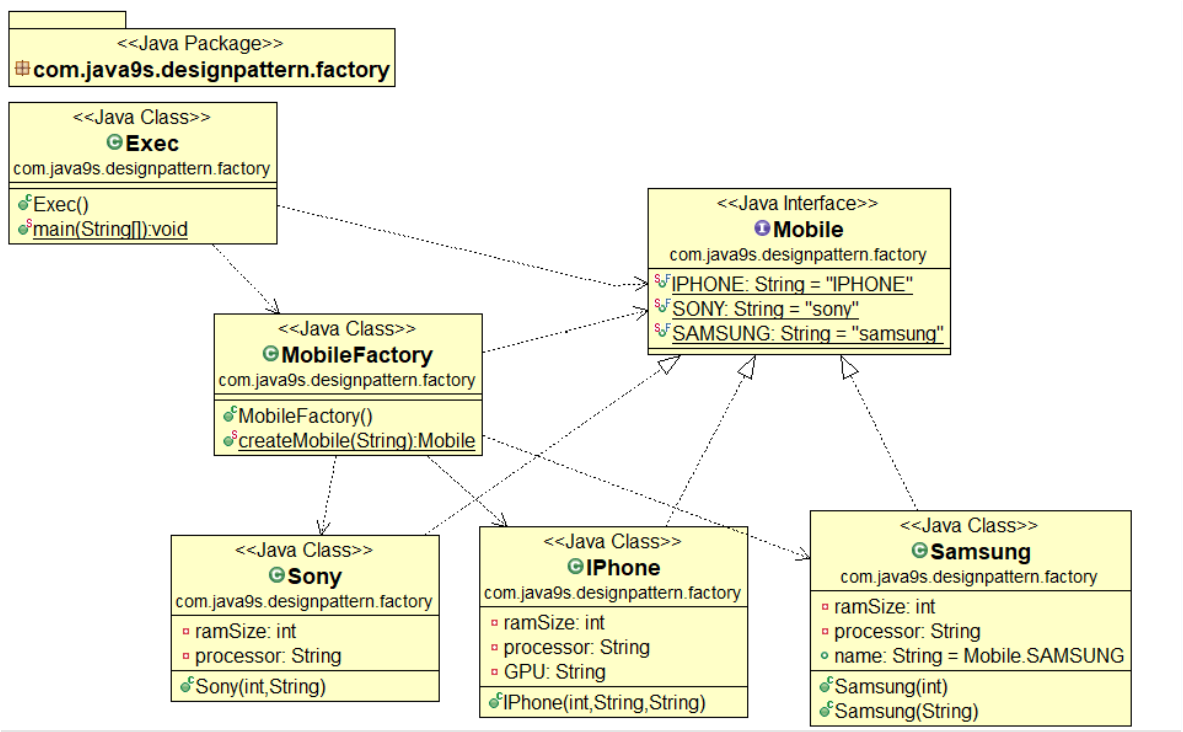
El segundo elemento *(concrete products*) lo conforman las 3 clases concretas de Sony, Iphone y Samnsung las cuales son instanciadas cada vez que se crea un celular nuevo. Estas clases son implementaciones concretas del tipo de celular definido en la interfaz mediante un Constructor particular que es invocado por el *Mobile factory* según sea el caso.

El tercer elemento (Creator) se encuentra definido en la clase Mobile Factory. Este método es capaz de crear instancias de cualquier tipo de celular (*Mobile)* implementando las fábricas concretas de cada una de las clases concretas según sea el caso.

De este modo, el cuarto elemento (Concrete creators) lo componen los métodos constructores de las 3 clases concretas. Estos son los encargados de crear las instancias particulares del tipo de celular solicitado.

En resumen, el patrón en el proyecto se ve alrededor de la clase Mobile Creator porque por un lado en torno a esta se centraliza la tarea de crear las instancias particulares al servir como puente y por otro delega la tarea específica de crear las dichas interfaces a subclases. En otras palabras, contribuye a la centralización y al desacoplamiento.

En el siguiente diagrama UML aparece la totalidad de la aplicación, donde puede evidenciarse todo lo dicho.



Ventajas:

Como se mencionó anteriormente, el uso del patrón en esta parte del proyecto contribuye principalmente a la centralización y al desacoplamiento. Esto porque, al delegar la tarea de crear un celular de un tipo específico en las subclases Samsung, iphone y Sony, la implementación concreta del objeto depende solo de estas subclases, por lo que, si se altera, solo debe preocuparnos la subclase. Esto fomenta la flexibilidad y mantenibilidad del código.

Desventajas:

Las desventajas de utilizar este patrón en el proyecto giran en torno a que incrementan su complejidad. Este es un proyecto muy simple que solo requiere crear instancias de tres clases distintas, por lo que no era del todo necesaria la interfaz *Mobile* que de hecho es muy simple y la aplicación podría servir con normalidad implementando únicamente la clase mobile factory sin que la extensión de esta resulte excesiva. En tal sentido, la implementación de la interfaz y las subclases pueden ser casos de sobreingeniería donde no necesariamente la utilización del patrón ayudaba a reducir la complejidad del proyecto.