Factory Method

El patrón Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos, pero permite a las subclases decidir qué clase concreta instanciar. El propósito principal del patrón Factory Method es permitir la creación de objetos de una manera más flexible y extensible.

En un proyecto en el que se utiliza el patrón Factory Method, es común encontrar una jerarquía de clases donde una clase base define una interfaz para crear objetos y las subclases concretas implementan el método de creación según sus necesidades específicas. La estructura general del diseño puede incluir una clase Creator abstracta que define el Factory Method y las subclases ConcreteCreator que implementan el método de creación para instanciar objetos concretos.

El fragmento del proyecto donde se utiliza el patrón Factory Method puede ser identificado al buscar la implementación de una clase Creator abstracta y sus subclases concretas que proporcionan diferentes implementaciones del Factory Method para crear objetos específicos.

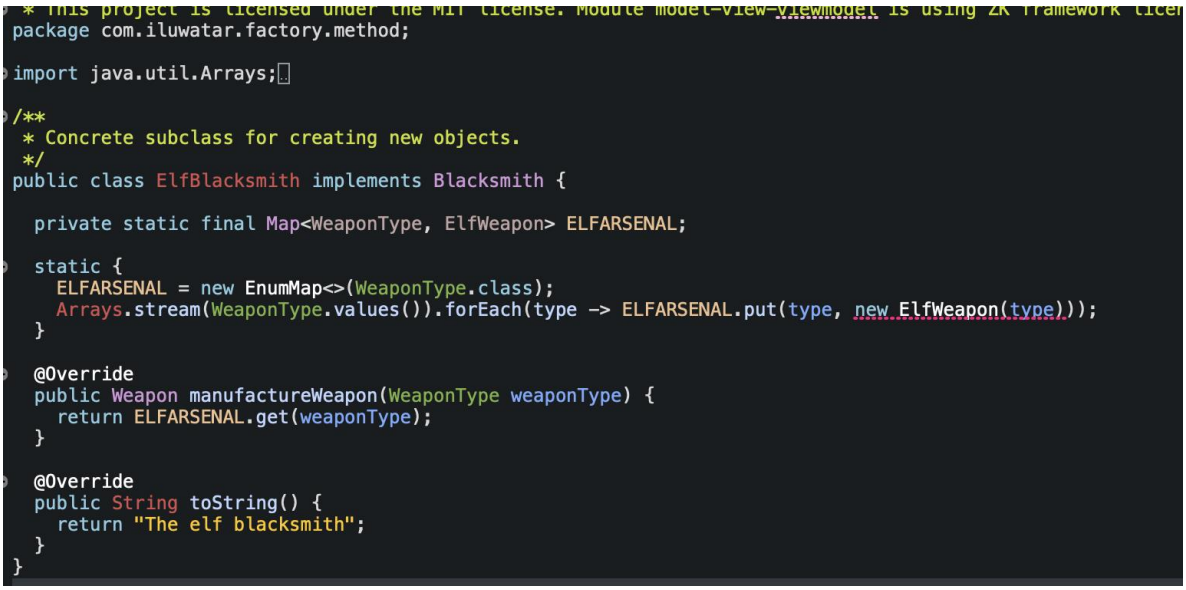
El uso del patrón Factory Method en el proyecto tiene sentido cuando se necesita encapsular la lógica de creación de objetos y proporcionar una forma flexible de extender y personalizar la creación de objetos sin modificar el código existente. Al utilizar el patrón, se pueden agregar nuevas subclases concretas que implementen el Factory Method para crear objetos adicionales sin afectar las clases existentes.

Las ventajas de utilizar el patrón Factory Method en el proyecto incluyen una mayor flexibilidad y extensibilidad al agregar nuevos tipos de objetos sin modificar el código existente. También permite desacoplar el código cliente de las clases concretas, ya que se comunica a través de la interfaz de la clase Creator abstracta.

Sin embargo, algunas posibles desventajas del patrón Factory Method pueden incluir la complejidad adicional debido a la necesidad de crear y mantener múltiples clases y jerarquías de clases. Además, puede haber una sobrecarga en el rendimiento si se utilizan demasiadas instancias de Factory Method en un proyecto grande.

En cuanto a posibles soluciones alternativas, en este caso particular podrían haberse utilizado otros patrones de diseño creacionales, como el patrón Abstract Factory o el patrón Builder, dependiendo de los requisitos específicos del proyecto. Cada patrón tiene sus propias ventajas y consideraciones de diseño, por lo que la elección del patrón más adecuado depende de las necesidades y restricciones del proyecto en cuestión.

A continuación un ejemplo donde se usa el patrón explicado:



```java
* This project is licensed under the MIT license. Module model-view-viewmodel is using ZK framework licen
package com.iluwatar.factory.method;

import java.util.Arrays;

/**
 * Concrete subclass for creating new objects.
 */
public class ElfBlacksmith implements Blacksmith {

  private static final Map<WeaponType, ElfWeapon> ELFARSENAL;

  static {
    ELFARSENAL = new EnumMap<>(WeaponType.class);
    Arrays.stream(WeaponType.values()).forEach(type -> ELFARSENAL.put(type, new ElfWeapon(type)));
  }

  @Override
  public Weapon manufactureWeapon(WeaponType weaponType) {
    return ELFARSENAL.get(weaponType);
  }

  @Override
  public String toString() {
    return "The elf blacksmith";
  }
}
```

En este código se utiliza el patrón Factory Method. En este caso, la clase **ElfBlacksmith** actúa como la implementación concreta del Factory Method. Observamos que la clase **ElfBlacksmith** implementa la interfaz **Blacksmith**, que define el método **manufactureWeapon()** para crear objetos **Weapon**. Dentro de la implementación del método **manufactureWeapon()**, se utiliza un objeto **EnumMap** llamado **ELFARSENAL** para almacenar instancias de **ElfWeapon** asociadas a cada tipo de arma (**WeaponType**).
El Factory Method se utiliza en este proyecto para encapsular la lógica de creación de objetos **Weapon** específicos para el **ElfBlacksmith**. Cada vez que se llama al método **manufactureWeapon()** con un **WeaponType** específico, se recupera la instancia correspondiente de **ElfWeapon** del **ELFARSENAL** y se devuelve como resultado.
El uso del patrón Factory Method en este punto del proyecto tiene sentido porque permite que el **ElfBlacksmith** maneje la creación de objetos **Weapon** de manera flexible y extensible. Si en el futuro se agregan nuevos tipos de armas, solo se necesitaría modificar el **ELFARSENAL** para incluir las nuevas instancias correspondientes. Esto evita que el

código cliente dependa directamente de las implementaciones concretas de **Weapon** y permite una fácil expansión del arsenal de armas del elfo.

Las ventajas de utilizar el patrón Factory Method en este caso incluyen la capacidad de agregar y gestionar fácilmente nuevos tipos de armas, el desacoplamiento entre el código cliente y las clases concretas y la posibilidad de extender y personalizar la lógica de creación de objetos sin afectar el resto del código.
Como posible desventaja, se puede mencionar que el patrón Factory Method puede aumentar la complejidad del código al introducir una jerarquía de clases adicional y requerir el mantenimiento de múltiples implementaciones de Factory Method.
En este caso particular, no se identifican otras formas obvias de solucionar el problema de la creación de objetos **Weapon** sin el uso del patrón Factory Method. La elección de este patrón parece adecuada dadas las necesidades del proyecto y las ventajas que aporta en términos de flexibilidad y extensibilidad en la creación de armas para el elfo.