**Factory method**

Es un patrón de diseño creacional que permite contar con una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearan.

Suponiendo que diseñamos una pequeña aplicación, en la que la base del código se encuentra acoplado a una clase, y en el futuro deseamos crear nuevos tipos de objetos, deberemos modificar la base del código para poder agregar esta nueva clase, teniendo al final un código sucio, lleno de condicionales que cambian el comportamiento del programa dependiendo el tipo de objetos a crear.

Entonces, la solución que propone este patrón, es que en vez de que la creación de objetos se realice en la clase en cuestión con el operador new (o cualquiera que cumpla esta función), lo que se haga sea que el operador en cuestión invoque un método fábrica especial, el cual devolverá unos objetos denominados productos. Al cambiar de ubicación el constructor, el método fábrica puede sobreescribirse en una subclase y cambiar la clase de productos creados por el método. Sin embargo, hay una pequeña limitación: las subclases devuelven productos de distintos tipos si estos tienen una clase base o interfaz común; además de que la superclase debe tener su tipo de retorno declarado como dicha interfaz.

**Estructura:**

* Producto: declara la interfaz que es común a todos los objetos que puede producir la clase creadora y sus subclases.
* Productos concretos: son las 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. El patrón Factory method puede ser declarado como abstracto y obligar a las subclases a implementar sus propias versiones del método. Como alternativa, el método devuelve un tipo de producto por defecto.
* Creadores concretos: sobreescriben el método fábrica base, de modo que devuelva un tipo diferente de producto. El método fábrica no tiene que crear nuevas instancias todo el tiempo, también puede devolver objetos existentes de una memoria cache, una agrupación de objetos o alguna otra fuente.

**Aplicabilidad:**

* Cuando no se conoce de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar el código: esto debido a que este patrón separa el código de construcción del producto del código de uso del producto, permitiendo extender el código de construcción de forma independiente al resto del código.
* Cuando se desea ofrecer a los usuarios de una biblioteca o framework una forma de extender sus componentes internos: la herencia es la forma más sencilla de extender el comportamiento por defecto de una biblioteca o framework, y para que un framework reconozca si debe utilizar una subclase en lugar de un componente estándar, se debe reducir el código de que construye componentes en el framework a un único Factory method y permitir que cualquiera sobrescriba este método aparte de extender el propio componente.
* Cuando se desea ahorrar recursos del sistema mediante la reutilización de objetos existentes en vez de reconstruirlos cada vez: Suele experimentarse cuando se trabaja con objetos grandes y que consumen muchos recursos, como conexiones de bases de datos, sistemas de archivos y recursos de red. Hay que considerar que para reutilizar un objeto existente: primero hay que crear un registro de todos los objetos creados, el programa deberá buscar el objeto que necesite alguien de todo ese agrupamiento, devolver el código al cliente, y si no hay objetos disponibles, el programa deberá crear ese objeto y agregarlo a ese agrupamiento. Esto significa que se necesita mucho código y habrá que ponerlo en un solo lugar para no tener código duplicado. Si bien pareciera que el constructor de la clase es el lugar ideal, pero por definición este siempre debe devolver un nuevo objeto y no instancias existentes, por lo que se requerirá de un método regular capaz de crear nuevos objetos y reutilizar los ya existentes.

**Implementación:**

* Hacer que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.
* Añadir 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.
* Encontrar todas las referencias a constructores del producto en el código de las clase creadora y sustituirlas por invocaciones al Factory method, mientras extraer el código de creación de productos y colocarlo en el Factory method. También puede ser requerido añadir un parámetro temporal al Factory method para controlar el tipo de producto devuelto.
* Crear un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory method. Sobreescribir el Factory method en las subclases y extraer las partes adecuadas del código constructor del método base.
* En caso de existir demasiados tipos de productos al punto de que no tenga sentido crear subclases para todos ellos, reutilizar el control de la clase base en las subclases.
* En caso de que posterior a esta extracción, el Factory method quede vacío, se lo puede hacer abstracto; y en caso de que quede algo de código, convertirlo en un comportamiento por defecto del método.

**Ventajas:**

* Se evita un acoplamiento fuerte entre el creador y los productos concretos.
* Principio de responsabilidad única: es posible ubicar el código de creación de producto en un lugar del programa, haciendo que sea más fácil de mantener.
* Principio de abierto/cerrado: es posible incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.

**Desventaja:**

* El código puede ser complicado, ya que se deben incorporar una multitud de nuevas subclases para implementar el patrón. La situación ideal sería implementar el patrón en una jerarquía existente de clases creadoras.

**Relaciones con otros patrones:**

* Muchos diseños empiezan con Factory method, que es menos complicado y más personalizable con las subclases, y evolucionan a Abstract factory, Prototype o Builder, que son más flexibles pero más complicados.
* Las clases del Abstract factory suelen basarse en un grupo de métodos de fábrica, pero también se puede utilizar Prototype para escribir los métodos de estas clases.
* Es posible utilizar el Factory method junto con el Iterator para permitir que las subclases de la colección devuelvan distintos tipos de iteradores que sean compatibles con las colecciones.
* Prototype no se basa en la herencia, por lo que no presenta sus inconvenientes, pero requiere una inicialización complicada del objeto clonado, mientras que Factory method si se basa en herencia, pero no requiere un paso de inicialización.
* Factory method es una especialización del Template method. Al mismo tiempo, un Factory method puede servir como paso de un gran Template method.