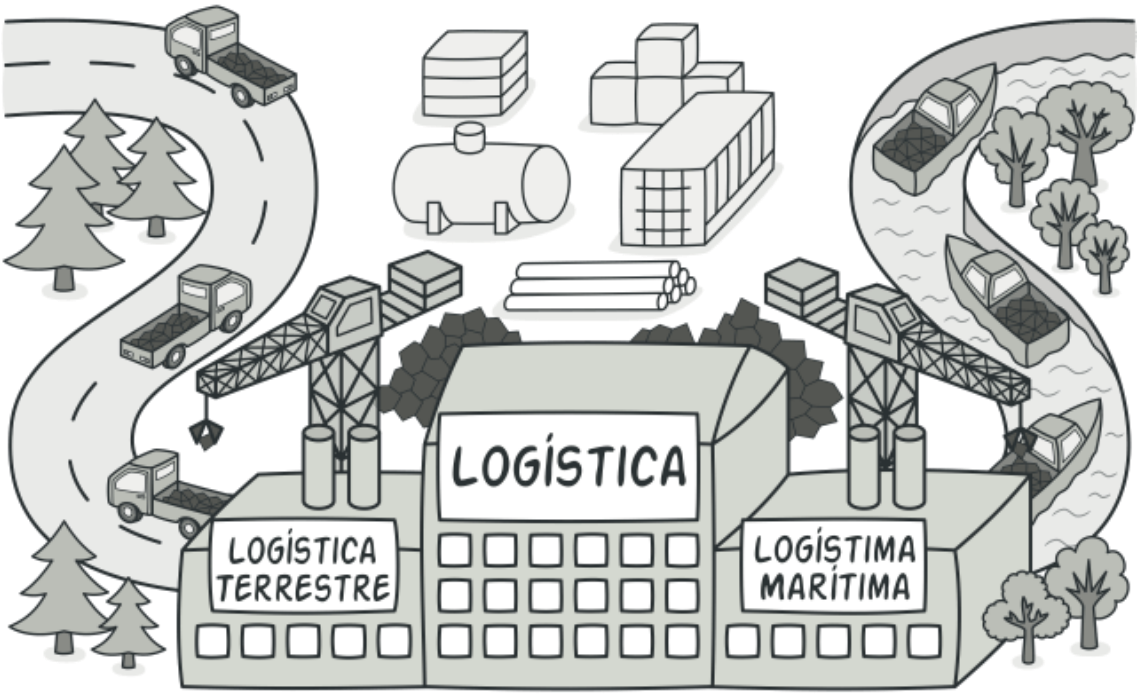
FACTORY PATTERNS

- SIMPLE FACTORY
- FACTORY METHOD

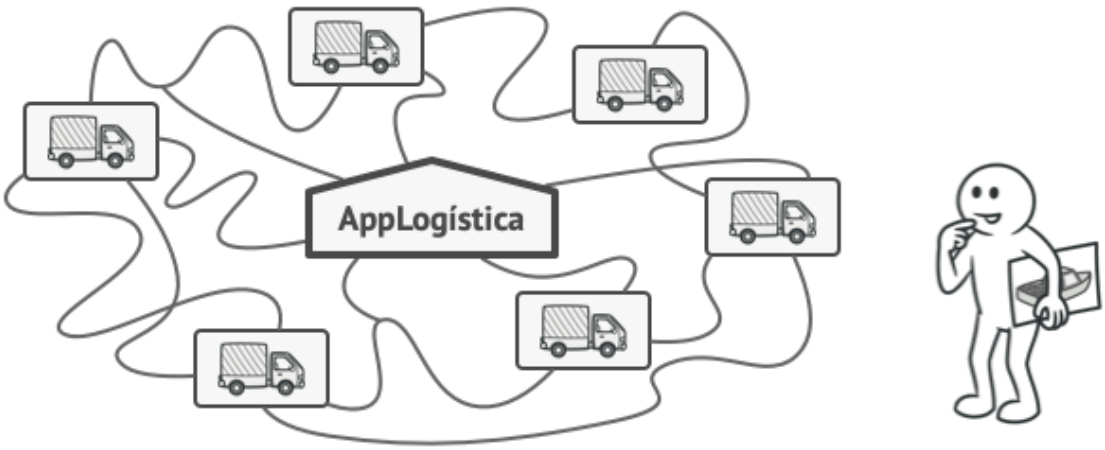
PROPÓSITO DE PATRONES FACTORY

En forma general, los patrones Factory son creacionales y proporcionan una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase Camión.

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.



Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.

Estupendo, ¿verdad? Pero, ¿qué pasa con el código? En este momento, la mayor parte de tu código está acoplado a la clase Camión. Para añadir barcos a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decides añadir otro tipo de transporte a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Al final acabarás con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.

https://refactoring.guru/es/design-patterns/factory-method

SIMPLE FACTORY

Los patrones creacionales o de creación eran aquellos en los que se **delegaba la instanciación de un objeto en otro** en lugar de recurrir a un simple *new()*. La pregunta que nos hacemos es: ¿por qué hacer esto? ¿Qué interés práctico puede existir en crear una clase cuya función sea instanciar otras clases pudiendo dejarle el trabajo a la clase original?

Bien, esta forma de trabajar puede ser útil en algunos escenarios, pero el principal suele involucrar el **no saber qué objeto vamos a instanciar hasta el momento de la ejecución**. Valiéndonos del polimorfismo podremos utilizar una interfaz para alojar una referencia a un objeto que será instanciado por un tercero en lugar de dejar que sea el propio constructor del objeto el que proporcione la instancia.

Por ejemplo y pensando en distintos tipos de motores que puede tener un vehículo y **"sabiendo de antemano que el tipo de motor que tiene el vehículo es Diesel"**, podemos crear la instancia así:

```
MotorDiesel motor = new MotorDiesel();
```

Pero si necesitamos algo más genérico, "**pues no sabemos el tipo de motor que tendrá el vehículo hasta el momento de la ejecución del código**", creamos la instancia así:

```
IMotor iMotor = MotorFactory.CreateInstance(tipoMotor);
```

Por tanto, nuestro objetivo principal será la **encapsulación de la creación de objetos**. Veamos un ejemplo donde tenemos: vehículos con un motor que está representado mediante una interfaz que será implementada por las clases "**MotorDiesel**" y "**MotorGasolina**". La interfaz expondrá las siguientes propiedades y métodos:

Ver código de interfaz IMotor.cs

A continuación, codificamos las dos clases que implementan la interfaz. En primer lugar implementaremos el motor Diesel, cuyos métodos devolverán cadenas de texto con sus respectivos mensajes:

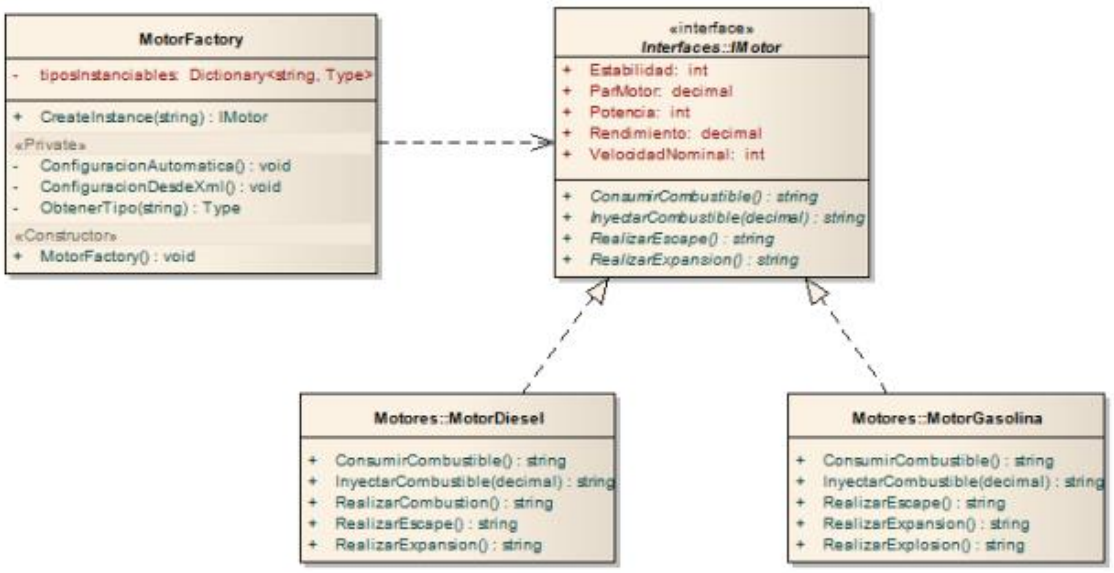
- **Ver código de clase MotorDiesel.cs**
  El motor Diesel realiza una combustión para simbolizar el tratamiento del combustible (ver el método privado `private string RealizarCombustion())`

- **Ver código de clase MotorGasolina.cs**
  El motor a Gasolina en vez de realizar una combustión, realiza una explosión (ver el método privado `private string RealizarExplosion())`

Finalmente, creamos una clase *MotorFactory* que tenga un método público *CreateInstance(nombreClase)* para instanciar ambos tipos de motores.

Ver código de clase MotorFactory.cs

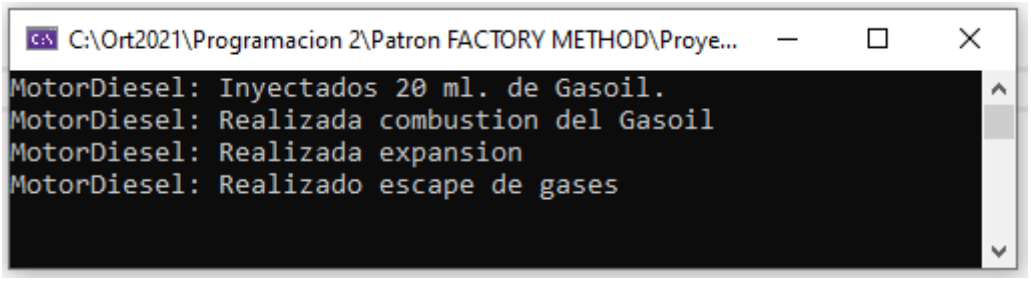
El diagrama de clases de esta primera factoría sería el siguiente:



Si quisiéramos obtener un motor diesel, por lo tanto, bastaría con el siguiente código:

Ver código del Main

La ejecución dará al siguiente resultado:



FACTORY METHOD

Hasta ahora hemos explicado el concepto de factoría: una clase especializada en crear objetos. A partir de aquí, podemos especializar aún más estas factorías para que generen tipos concretos. El patrón *Factory Method* es similar a lo que hemos visto hasta ahora, con una pequeña variación: el método *CreateInstance()* ya no generará una instancia, sino que o bien se convertirá en **abstracto** o bien pertenecerá a una interfaz y dejará a las clases que la implementen la tarea de codificar su comportamiento.

Por lo tanto, nuestra clase *MotorFactory* ya no podrá utilizarse **directamente** para generar objetos, sino que habrá que crear una nueva clase que herede

de *MotorFactory* que implemente el método *CreateInstance()*, o bien transformar *MotorFactory* en una interfaz *IMotorFactory* y dejar a las clases que la implementen el trabajo de instanciar las clases.

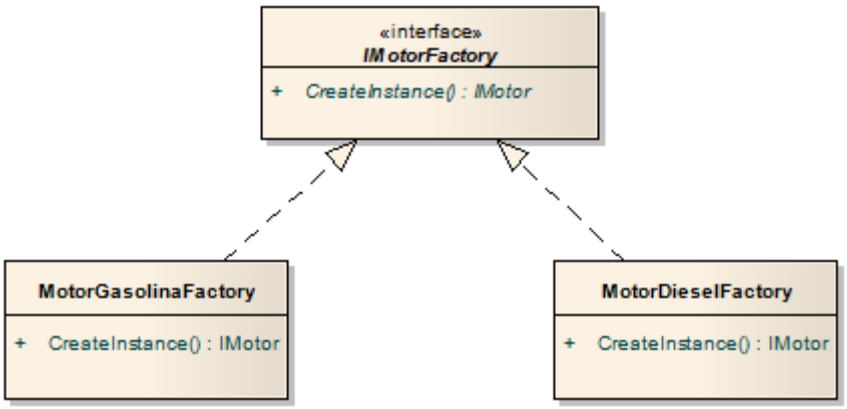
¿Qué conseguimos con esto? Básicamente, especializar las factorías. En lugar de tener una única factoría que centralice todo el proceso de creación de objetos, tendremos varias clases factoría que heredan de *MotorFactory* (o implementan *IMotorFactory*) que estarán especializadas en crear variantes concretas.

En el ejemplo sólo exponemos el caso de la instanciación de las clases *MotorDiesel* y *MotorGasolina*. Además, las diferencias entre ambas clases son mínimas, por lo que una factoría simple nos bastaría para nuestro diseño. Pero, ¿qué ocurriría si estas clases se especializaran cada vez más y más? ¿Y si se agregan muchos más tipos de motores? Quizás la carga funcional sobre nuestra factoría sería demasiada, por lo que sería aconsejable pasar al siguiente paso: utilizar *Factory Method* para especializar las factorías.

Vamos a realizar, por lo tanto, el siguiente proceso:

1) Eliminar la clase *MotorFactory* y sustituirla por la interfaz *IMotorFactory*. Esta nueva interfaz expondrá el método *CreateInstance()*, que deberá ser implementado por otras clases.

2) Crear dos nuevas clases, *MotorDieselFactory* y *MotorGasolinaFactory*, que implementen la interfaz *IMotorFactory* y su método *CreateInstance()*, especializando el proceso de instanciado. El nombre del patrón *Factory Method* viene precisamente de aquí.



El método *CreateInstance* sigue devolviendo una interfaz *IMotor*. Podríamos pensar en que sería mejor que *MotorGasolinaFactory.CreateInstance()* devolviera una instancia de *MotorGasolina*, ¿verdad? Bien, en realidad lo hará, pero recordemos que *MotorGasolina* implementa la interfaz *IMotor*, por lo que puede usar esta referencia para devolver una instancia de *MotorGasolina* tal y como lo ha venido haciendo hasta ahora. Dejemos el nivel de abstracción lo más alto posible retornando algo de la interfaz *IMotor*.

Por tanto, el código de la interfaz *IMotorFactory* se simplificará hasta el punto de exponer únicamente la firma del método *CreateInstance()*:

```
public interface IMotorFactory
{
    IMotor CreateInstance();

}
```

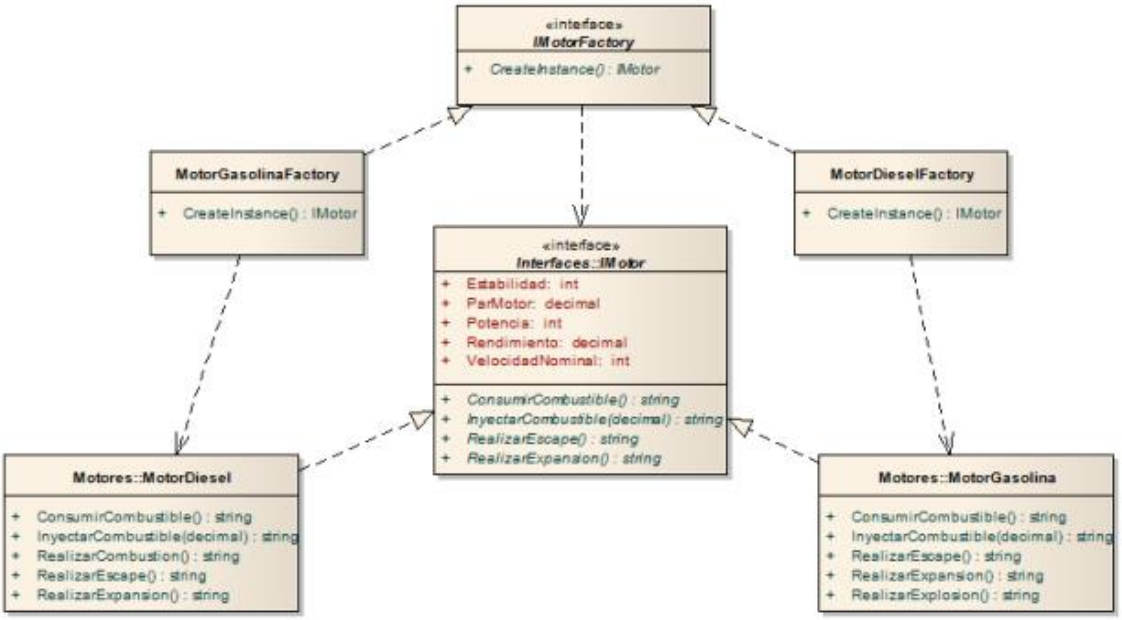
Ver código de interfaz IMotorFactory.cs

Codificando las factorías concretas

Lo siguiente será configurar el código de nuestras factorías individuales: *MotorDieselFactory* y *MotorGasolinaFactory*. En estas clases se codificará el comportamiento del método *CreateInstance()*, realizando las operaciones que consideremos oportunas para cada tipo de motor, como asignarle valores a sus propiedades o realizar tareas fijas de inicialización. Por ejemplo:

Ver código de clases MotorDieselFactory.cs y MotorGasolinaFactory.cs

El diagrama de clases asociado a nuestro modelo tendría, por tanto, el siguiente aspecto:



Ver código de clases MotorDiesel.cs y MotorGasolina.cs y la interfaz IMotor.
Todas son iguales a la implementación del patrón Simple Factory.

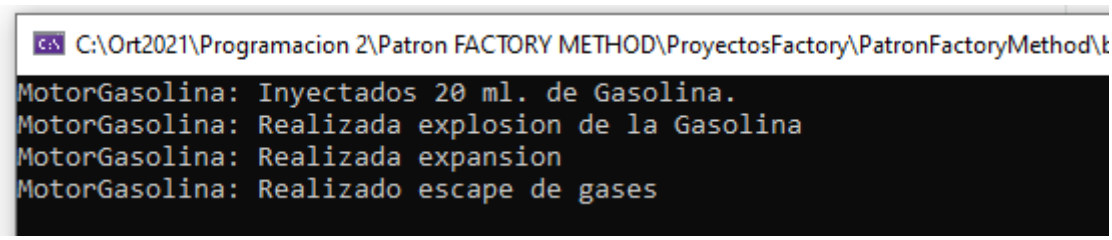
Utilizando Factory Method

Finalmente, haremos uso de *Factory Method* mediante los siguientes pasos:

- Usamos una interfaz de la factoría (genérica) para alojar una instancia de la factoría concreta, que será generada mediante el método *ObtenerFactoria()* accediendo a la configuración para decidir qué factoría debe instanciar.
- Usamos una interfaz del motor (genérica) para alojar una instancia de un motor concreto, que será generado mediante el método *CreateInstance()* de la factoría generada previamente.
- Hacemos uso de los métodos del objeto a través de su interfaz.

Ver código del Main

La ejecución dará al siguiente resultado:



¿Cuándo utilizar este patron? Ejemplos reales

- **Utiliza el Factory Method cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.**

  El patrón Factory Method separa el código de construcción de producto del código que hace uso del producto. Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código.

  Por ejemplo, para añadir un nuevo tipo de producto a la aplicación, sólo tendrás que crear una nueva subclase creadora y sobrescribir el Factory Method que contiene.

- **Utiliza el Factory Method cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.**

  La herencia es probablemente la forma más sencilla de extender el comportamiento por defecto de una biblioteca o un framework. Pero, ¿cómo reconoce el framework si debe utilizar tu subclase en lugar de un componente estándar?

  La solución es reducir el código que construye componentes en todo el framework a un único patrón Factory Method y permitir que cualquiera sobrescriba este método además de extender el propio componente.

  Veamos cómo funcionaría. Imagina que escribes una aplicación utilizando un framework de UI de código abierto. Tu aplicación debe tener botones redondos, pero el framework sólo proporciona botones cuadrados. Extiendes la clase estándar `Botón` con una maravillosa subclase `BotónRedondo`, pero ahora tienes que decirle a la clase principal `FrameworkUI` que utilice la nueva subclase de botón en lugar de la clase por defecto.

  Para conseguirlo, creamos una subclase `UIConBotonesRedondos` a partir de una clase base del framework y sobrescribimos su método `crearBotón`. Si bien este método devuelve objetos `Botón` en la clase base, haces que tu subclase devuelva objetos `BotónRedondo`. Ahora, utiliza la clase `UIConBotonesRedondos` en lugar de `FrameworkUI`. ¡Eso es todo!

- **Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.**

A menudo experimentas esta necesidad cuando trabajas con objetos grandes y que consumen muchos recursos, como conexiones de bases de datos, sistemas de archivos y recursos de red.

Pensemos en lo que hay que hacer para reutilizar un objeto existente:

1. Primero, debemos crear un almacenamiento para llevar un registro de todos los objetos creados.
2. Cuando alguien necesite un objeto, el programa deberá buscar un objeto disponible dentro de ese agrupamiento.
3. … y devolverlo al código cliente.
4. Si no hay objetos disponibles, el programa deberá crear uno nuevo (y añadirlo al agrupamiento).

¡Eso es mucho código! Y hay que ponerlo todo en un mismo sitio para no contaminar el programa con código duplicado.

Es probable que el lugar más evidente y cómodo para colocar este código sea el constructor de la clase cuyos objetos intentamos reutilizar. Sin embargo, un constructor siempre debe devolver **nuevos objetos** por definición. No puede devolver instancias existentes.

Por lo tanto, necesitas un método regular capaz de crear nuevos objetos, además de reutilizar los existentes. Eso suena bastante a lo que hace un patrón Factory Method.