**PATRONES DE DISEÑO**

**FACTORY METHOD:**

**Crear objetos sin exponer su Inicialización**

El proceso de inicialización puede ser a través de una función -> **FACTORY METHOD**

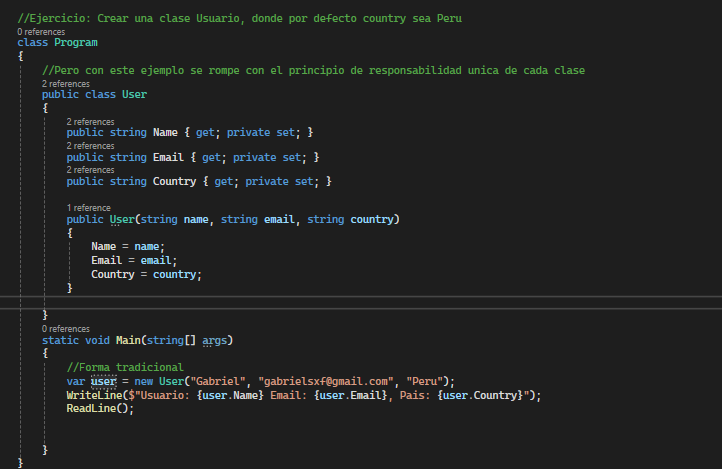
El proceso de inicialización puede ser a través de una clase -> **ABSTRACT FACTORY**

A diferencia del patrón de diseño **Builder** que se hacía pieza por pieza aquí se realiza mediante una única invocación

**Ejercicio1**

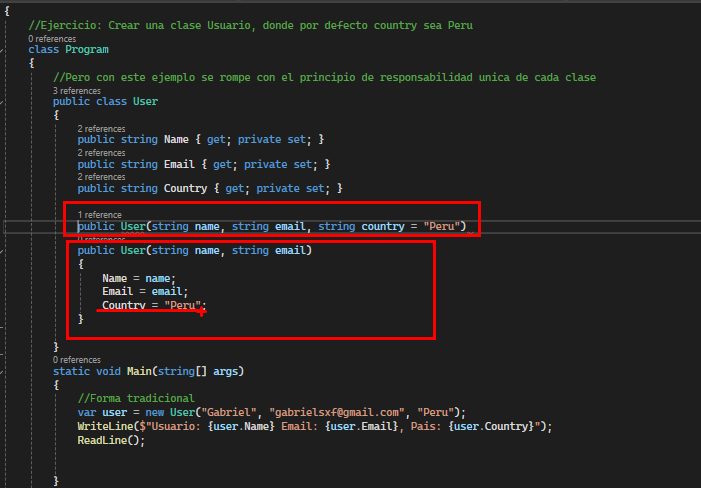
En el siguiente ejercicio tenemos una clase Usuario que solo recibe un **Name** un **email** y un **country,** lo normal que se conoce hasta ahora es que un objeto se cree a partir de su constructor:

Forma tradicional:



Ahora si nos piden que se agine un país por defecto por ejemplo **Peru**

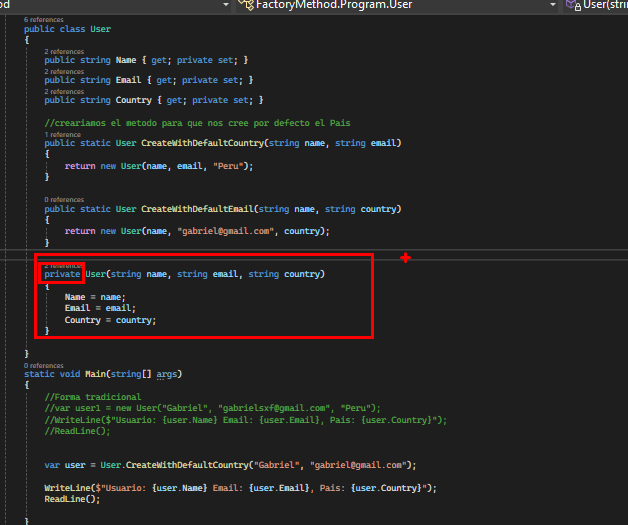
Podríamos hacerlo asi:



Aquí es donde entraría **Factory Method:**

ya que crearíamos un método que realice la parte de inicialización:

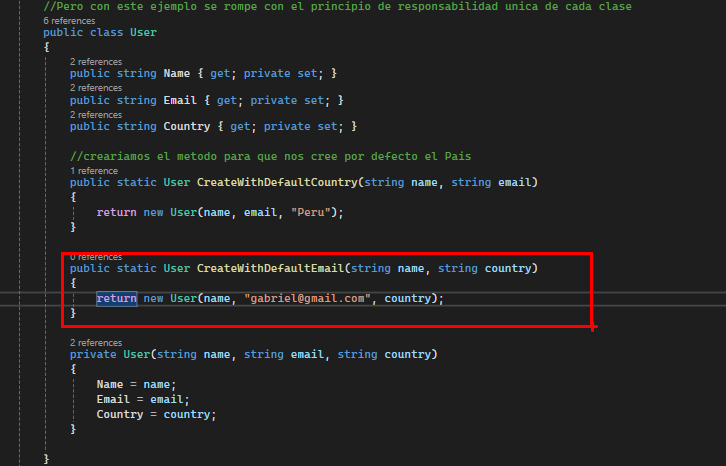
***abrimos el siguiente proyecto -> FactoryMethodBeforeB***



Notamos que ya no usamos el constructor porque lo estamos poniendo como **private**

Pero con este ejemplo se rompe con el principio de responsabilidad unica de cada clase ya que la clase Usuario no debería ser responsable de todos estos métodos de creacion

Ahora si queremos crear por defecto un correo tendríamos que hacer lo mismo crear un método que utilice ese correo por defecto:



Refactorizacion

Abrimos el archivo **AFTER-A**

Entonces para solventar la rotura de ese principio tendríamos que llevar estos dos métodos a otra clase, y marcamos el constructor como internal para que pueda ser accesible desde el dll o nameSpace, lo cual no es del todo correcto ya que no se quiere exponer el constructor solo debería estar a nivel de clase:



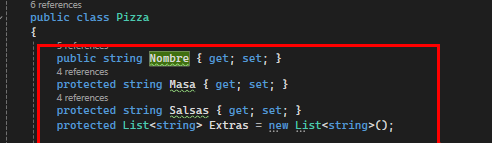
Por lo que tendríamos que aplicar el concepto de **clases anidadas** es decir tener una clase dentro de otra clase:



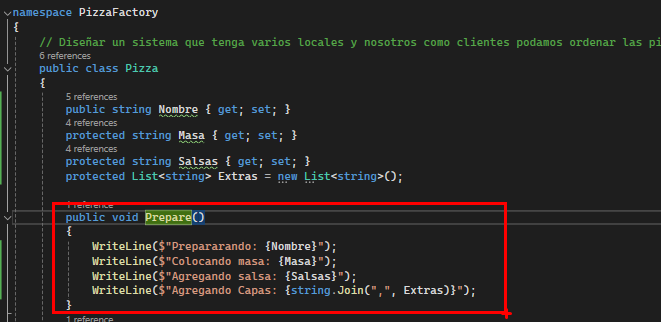
De esta forma evitamos exponer el constructor fuera de la clase

**Ejercicio2**

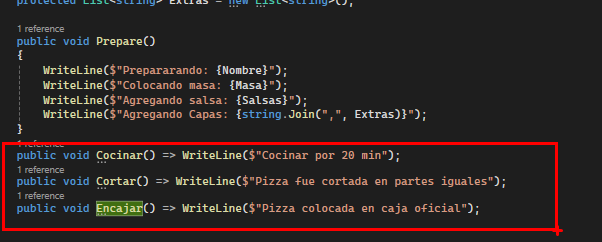
Realizar un ejercicio de que en una cadena de pizzerías pueden existir diferentes tipos de pizzas por locales, donde por ejemplo si existe un local que pertenece a la región Arequipa va a tener unos tipos de pizzas, mientras que si existe otro local en otra región va a tener otros tipos de pizzas, vamos a definir entonces una clase **Pizza**, con las siguientes propiedades:



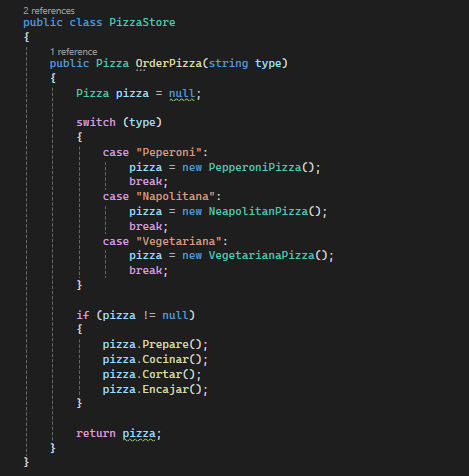
Luego va a tener una función para preparar la pizza:



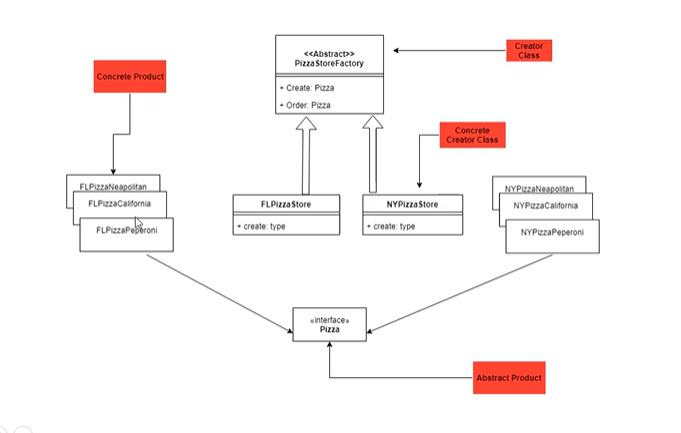
Y otras funciones mas:

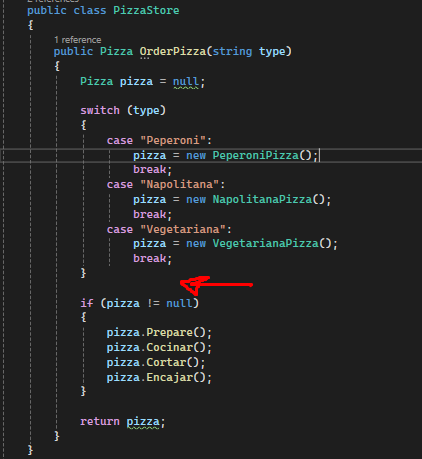


Luego vamos a crear una clase para ordenar una pizza llama **PizzaStore:**

****

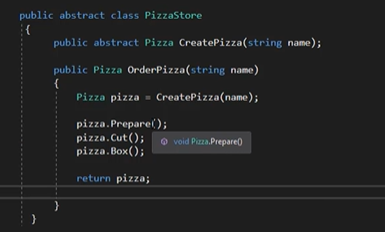
Analizamos la estructura:



Por lo que si quisiéramos agregar otro tipo de pizza, 

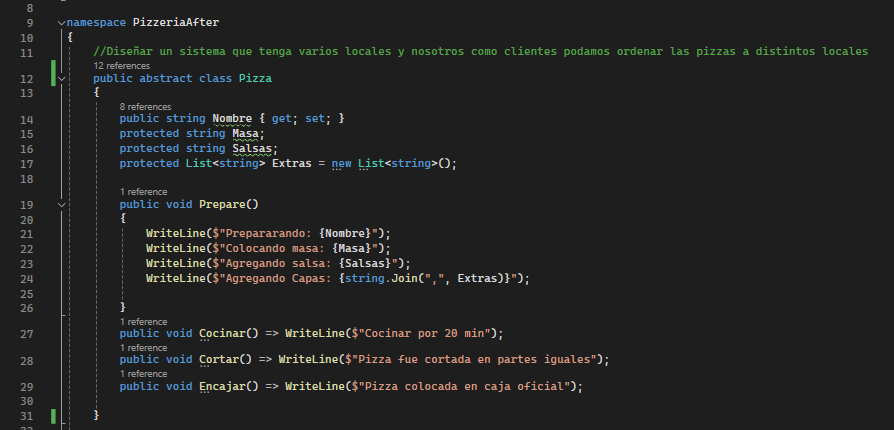
Entonces se rompería con el principio de **AbiertoCerrado** cerrado para la modificación y abierto para la extensión

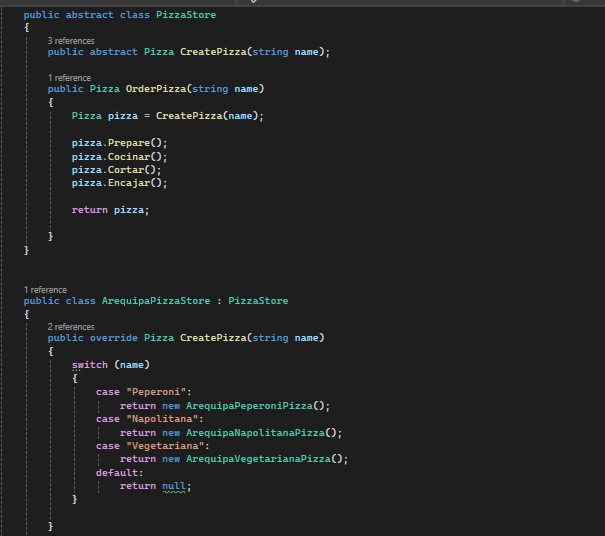
Por ello en vez del **switchCase** modifcariamos a una clase **Abstracta**:



**REFACTORIZACION**

Abrimos el **PizzeriaAfterA**





Ahora refactorizamos utilizando **Reflection**

**Abrimos nuestro proyecto PizzeriaAfterB**

Primero creamos nuestro **Enum**

****

Modificamos esto:

