PRÁCTICA PATRONES 2: FACTORY METHOD & ABSTRACT FACTORY

PATRONES FACTORIA

Cada vez que utilizamos **new** estamos instanciando un objeto de una clase concreta (no de un interface ni de una clase abstracta). Acoplar nuestro código a una clase concreta lo convierte en más frágil, menos flexible y por supuesto menos mantenible y extensible.

El problema surge cuando no sabemos qué clase tendremos que instanciar ya que depende de alguna condición, que no se decide hasta el momento de ejecución.

Si escribimos el código en base a una interface, funcionará con cualquier clase que implemente ese interface gracias al polimorfismo. Sin embargo si el código está basado en clases concretas, si se añaden nuevas clases concretas habrá que cambiar el código (algo que no ocurriría si utilizamos interfaces: solo necesitaríamos que las clases concretas implementasen dicho interface y no habría que cambiar nada de lo ya existente). Es decir nuestro código no será "cerrado a la modificación"

NOTA: uno de los **PRINCIPIOS SOLID**, el **principio abierto-cerrado**, que deberíamos intentar cumplir siempre, dice que las clases deberían ser abiertas a la extensión, pero cerradas a la modificación. Es decir, deberíamos diseñar nuestro código para que, si hay una petición de cambio, podamos extender las clases pero no modificar lo que ya está implementado.

La idea es coger las partes de la aplicación que creen instancias concretas y encapsularlas o aislarlas del resto de la aplicación, para que si se produce un cambio, esté localizado.

Ejemplo, si tenemos una pizzería con varios tipos de pizza:

```
Pizza orderPizza(String type) {
 Pizza pizza;
 if (type.equals("cheese")) {
  pizza = new CheesePizza();
 } else if (type.equals("greek") {
         pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
                  pizza = new PepperoniPizza();
               }
   pizza.prepare();
 pizza.bake();
 pizza.cut();
 pizza.box();
return pizza;
}
```

Para añadir más tipos de pizza habría que modificar el código del método orderPizza añadiendo más ifs anidados.

La parte del método que puede variar es la creación de la pizza, por tanto, esa parte habrá que encapsularla, sacando la parte de la creación de la pizza a un nuevo objeto (Factory) que se encargue de la creación de las pizzas.

1. FACTORIA SIMPLE (no es un patrón de diseño)

Crearemos una clase factoría que se encargue de la creación de las pizzas.

```
public class SimplePizzaFactory {
        public Pizza createPizza(String type) {
          Pizza pizza = null;
          if (type.equals("cheese")) {
             pizza = new CheesePizza();
          } else if (type.equals("pepperoni")) {
             pizza = new PepperoniPizza();
```

```
          } else if (type.equals("clam")) {
             pizza = new ClamPizza();
          } else if (type.equals("veggie")) {
             pizza = new VeggiePizza();
          }
         return pizza;
         }
}
```

El método createPizza es el que usarán todos los clientes para crear pizzas.

Habrá que darle al cliente una referencia a la factoría.

```
public class PizzaStore {
  SimplePizzaFactory factory;

  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
}

public Pizza orderPizza(String type) {
 Pizza pizza;
 pizza = factory.createPizza(type);
 pizza.prepare();
 pizza.bake();
 pizza.cut();
 pizza.box();
 return pizza;
}
```

El diagrama de clases de esta primera solución sería :

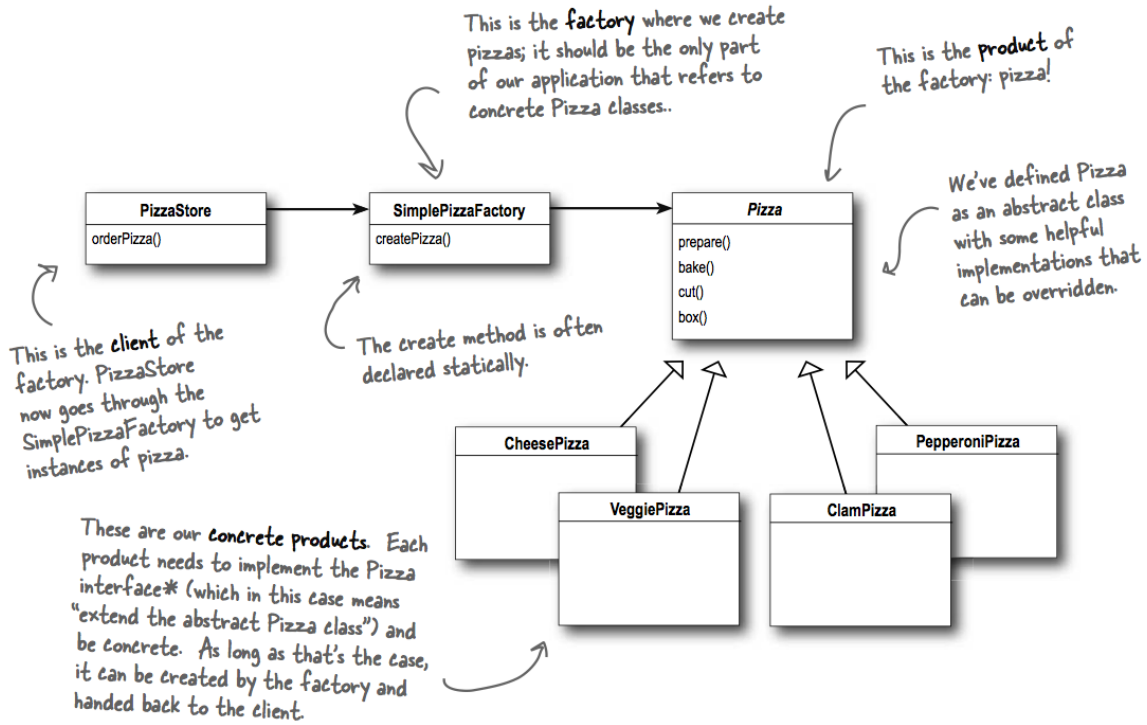
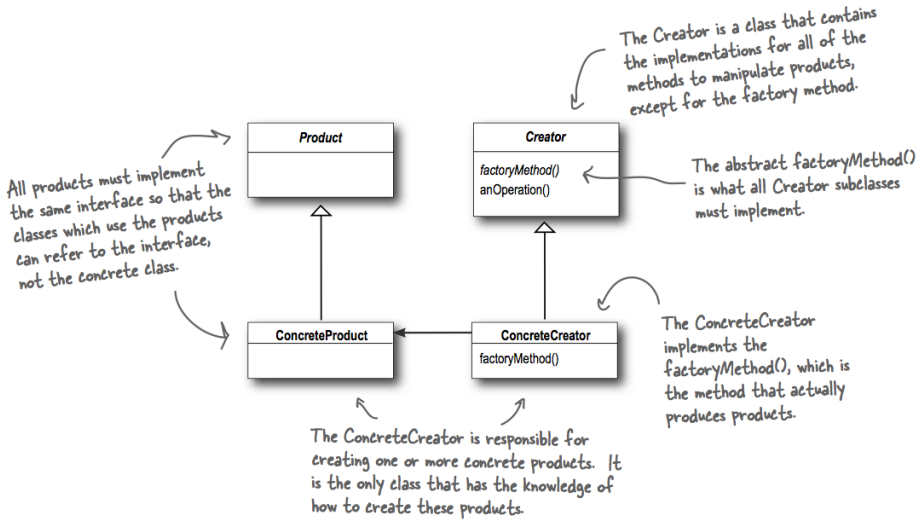


Ilustración 1. Diagrama de clases (Fuente: Head first design patterns)

2. PATRÓN FACTORY METHOD



Solución: El patrón Factory Method propone que, en lugar de llamar al operador new para construir objetos, **se utilice un método *fábrica* especial**. Los objetos se siguen creando a través del operador new, pero se invocan desde el método fábrica.

Patrón Factory method :define un interface para crear objetos, pero permite que las subclases decidan qué clase instanciar. Permite que una clase difiera la instanciación a subclases. Proporciona una forma de encapsular la instanciación de objetos concretos.

Queremos permitir a la factoría de las pizzas, que personalice las pizzas con un estilo diferente. Para ello creamos una clase abstracta con un método abstracto (el Factory method -> createPizza), que sustituye a los ifs anidados de la factoría simple:

```
public abstract class PizzaStore {
  public Pizza orderPizza(String type) {
     Pizza pizza;
     pizza = createPizza(type);
     pizza.prepare();
     pizza.bake();
     pizza.cut();
     pizza.box();
     return pizza;
  }
  protected abstract Pizza createPizza(String type);
}
```

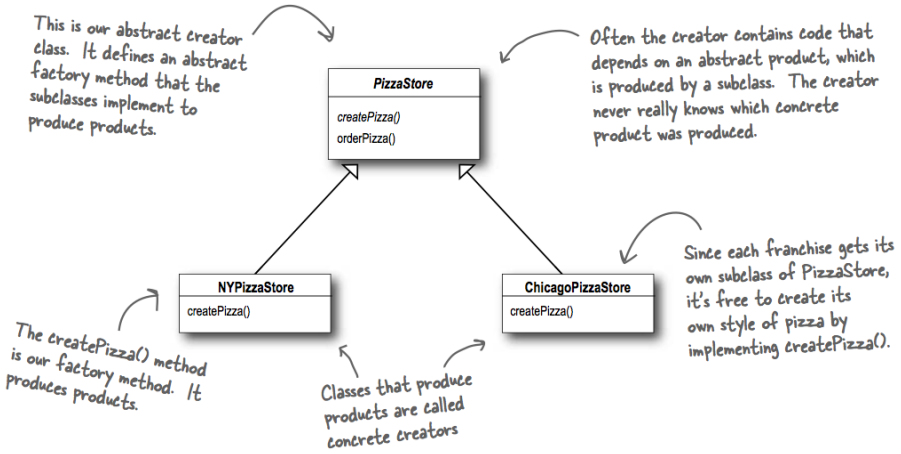
Toda la responsabilidad para crear instancias concretas de pizzas se ha movido al método **createPizza** que actúa como factoría. Puede tener otros métodos (no abstractos) que compartirán todas las factorías concretas.

El método factoría será abstracto en la superclase y se redefinirá en cada subclase. Además puede parametrizarse para indicar qué tipo de producto (en este ejemplo qué tipo de pizza) se creará.

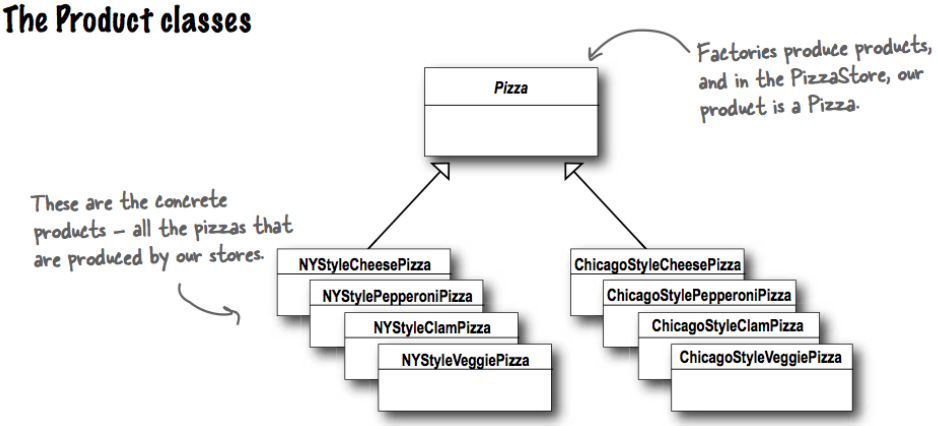
El patrón Factory Method encapsula la creación de objetos permitiendo a las subclases que decidan qué objeto crear. Intervienen clases creadoras y clases producto. Encapsula el conocimiento del producto en la clase creadora.

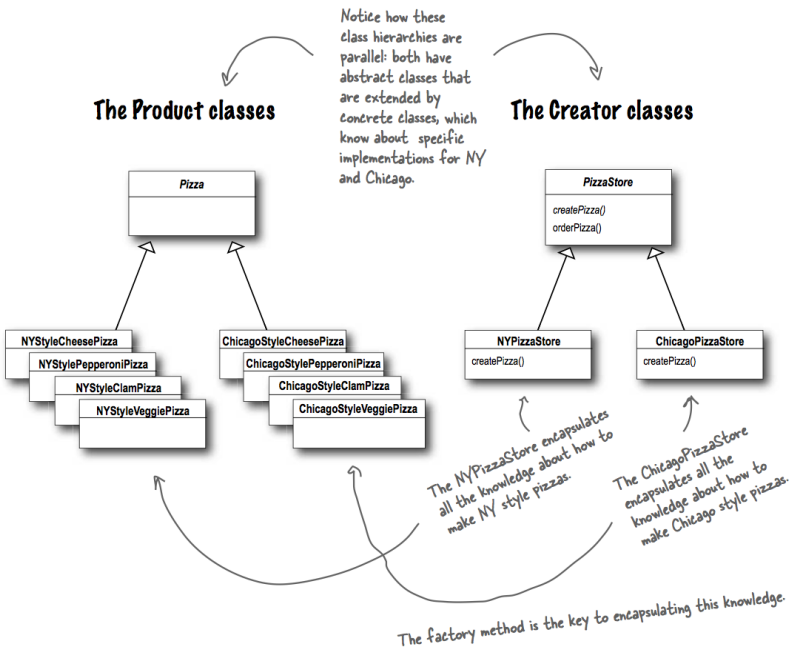
```
public class NYPizzaStore extends PizzaStore {

        Pizza createPizza(String item) {
                if (item.equals("cheese")) {
                        return new NYStyleCheesePizza();
                } else if (item.equals("veggie")) {
                        return new NYStyleVeggiePizza();
                } else if (item.equals("clam")) {
                        return new NYStyleClamPizza();
                } else if (item.equals("pepperoni")) {
                        return new NYStylePepperoniPizza();
                } else return null;
        }
}
```

The Creator classes

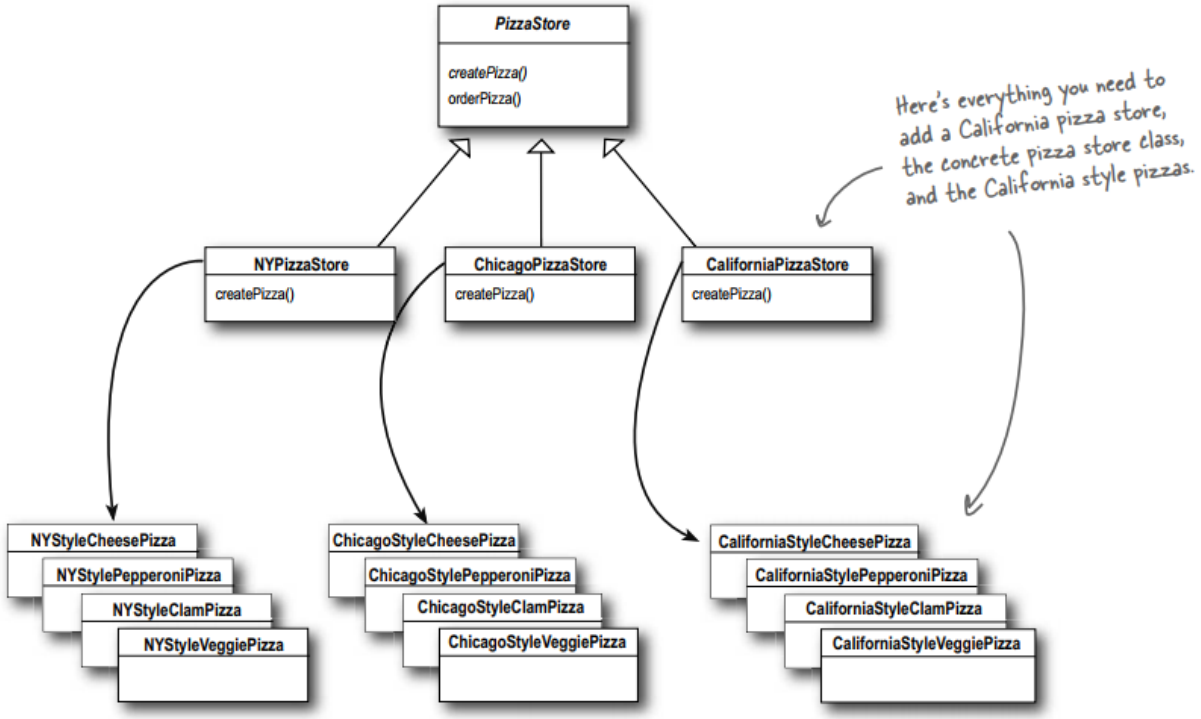


The Product classes

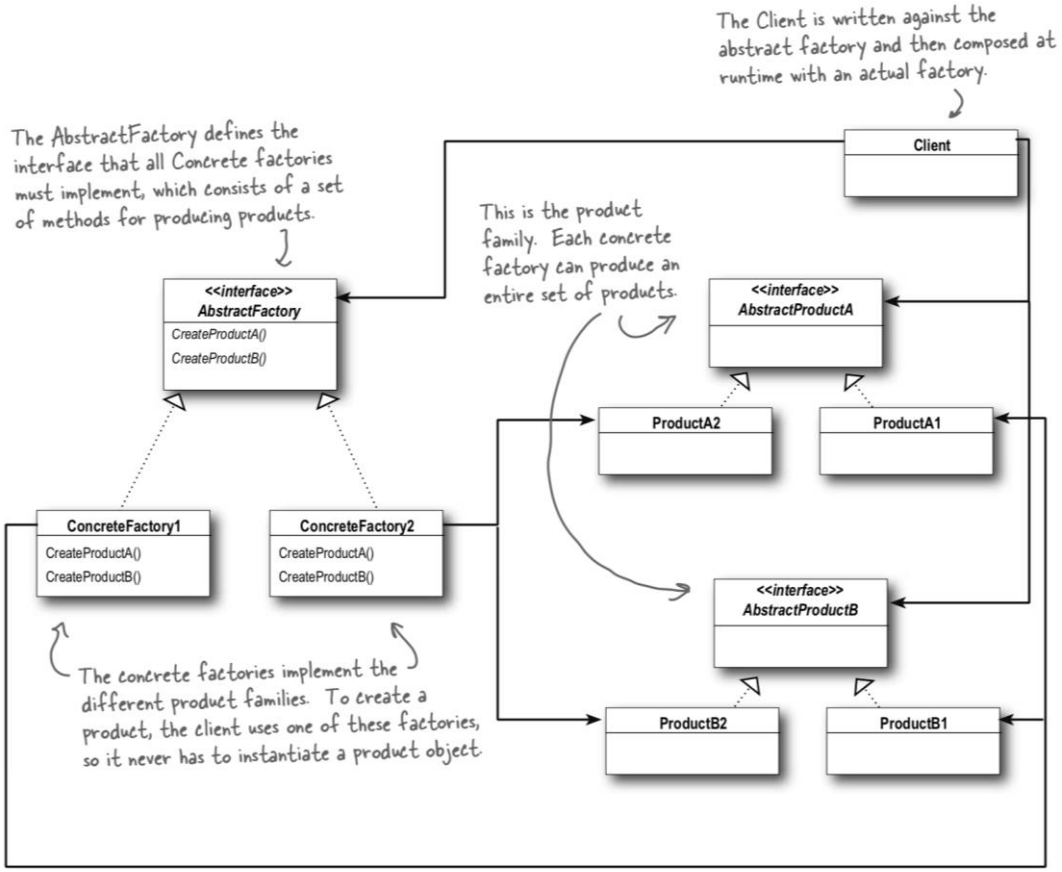




Si necesitásemos extender a otra ciudad, por ejemplo: California (con sus propios estilos de pizza), habría que añadir otra clase que implementase PizzaStore y tantas clases concretas de pizza como variedades tuvieran en California.



3. PATRÓN ABSTRACT FACTORY



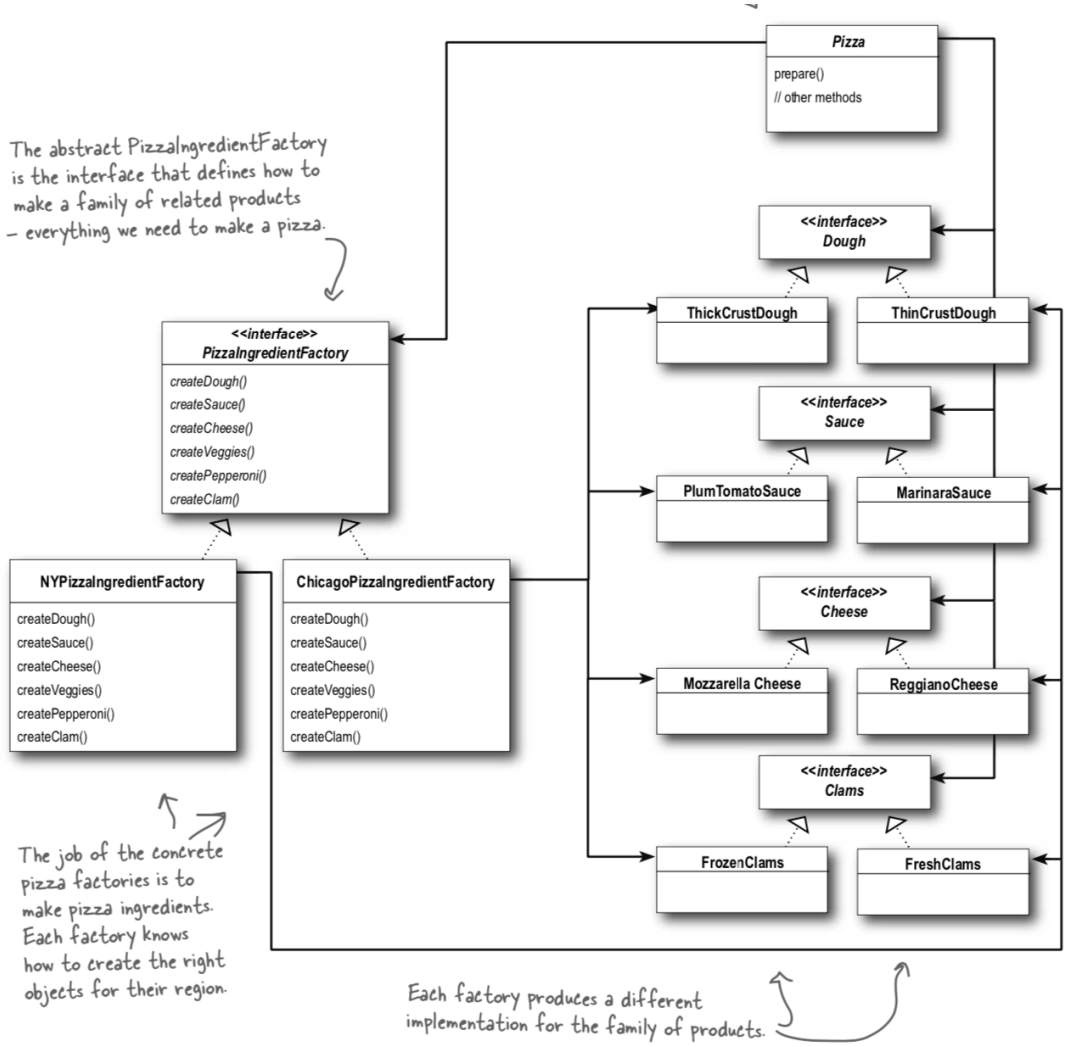
INVERSIÓN DE DEPENDENCIAS

Debemos escribir nuestro código para que dependa de abstracciones (clases abstractas o interfaces), no de clases concretas.

Abstract factory proporciona un interface para crear una familia de productos relacionados o dependientes sin especificar sus clases concretas. De esta forma el cliente se desacopla de los productos concretos (de las clases de los productos concretos que se crearán).

Implementando ese interface vamos creando factorías que desacoplan nuestro código. Esto nos permite implementar factorías que producen productos destinados a diferentes contextos (diferentes regiones, diferentes sistemas operativos, diferentes estilos….)

Cada método en la factoría abstracta es responsable de crear un producto concreto, por lo que se puede implementar siguiendo el patrón método factoría.



Pizza Ingrediente Factory Interface:

```
public interface PizzaIngredientFactory {

        public Dough createDough();
        public Sauce createSauce();
        public Cheese createCheese();
        public Veggies[] createVeggies();
        public Pepperoni createPepperoni();
        public Clams createClam();

}
```

New York Pizza Ingredient Factory:

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

        public Dough createDough() {
                return new ThinCrustDough();
        }

        public Sauce createSauce() {
                return new MarinaraSauce();
        }

        public Cheese createCheese() {
                return new ReggianoCheese();
        }

        public Veggies[] createVeggies() {
                Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
                return veggies;
        }

        public Pepperoni createPepperoni() {
                return new SlicedPepperoni();
        }

        public Clams createClam() {
                return new FreshClams();
```

COMPARACIÓN DE LOS PATRONES: FACTORY METHOD Y ABSTRACT FACTORY

Abstract Factory generalmente se implementa utilizando Factory Method y por tanto provee al menos toda la flexibilidad de éste. La diferencia principal entre ambos es que Abstract Factory trata con familias de productos, mientras que Factory Method se preocupa por un único producto.

Abstract Factory se encuentra a un nivel de abstracción mayor que Factory Method.

¿Que usos tiene el patrón Abstract Factory?

Este patrón se puede aplicar cuando:

- Un sistema debe ser independiente de cómo se crean sus objetos.
- Un sistema debe ser 'configurado' con una cierta familia de productos.
- Se necesita reforzar la noción de dependencia mutua entre ciertos objetos (objetos que se usan juntos, no se mezclan los objetos de diferentes familias).
- Muchos frameworks y bibliotecas lo utilizan para proporcionar una forma de extender y personalizar sus componentes estándar.

Ventajas del Patrón Abstract Factory:

- Proporciona flexibilidad al aislar a las clases concretas.
- Facilita cambiar las familias de productos.
- Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí.
- Evitas un acoplamiento fuerte entre productos concretos y el código cliente.
- Principio de responsabilidad única. Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener.
- Principio de abierto/cerrado. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.

Desventajas del Patrón Abstract Factory

- Para agregar nuevos productos se deben modificar tanto las fabricas abstractas como las concretas.

RESUMEN

- **Simple Factory**, aunque no es un patrón de diseño, es una forma sencilla de desvincular a sus clientes de las clases concretas.
- **Factory Method** se basa en la herencia: la creación de objetos se delega a subclases que implementan el método de fábrica para crear objetos.
- **Abstract Factory** se basa en la composición de objetos: la creación de objetos se implementa en los métodos expuestos en la interfaz de fábrica.
- Todos los patrones de fábrica promueven un acoplamiento flexible al reducir la dependencia de su aplicación en las clases concretas.

- La intención del Factory Method es permitir que una clase aplace la instanciación a sus subclases.
- La intención de Abstract Factory es crear familias de objetos relacionados sin tener que depender de sus clases concretas.
- El **principio de inversión de dependencias** nos guía para evitar dependencias de tipos concretos y luchar por abstracciones.
- Las fábricas son una técnica poderosa para codificar abstracciones, no clases concretas

EJERCICIO:

Crear un proyecto que cumpla este diagrama de clases: