

# 9

(III)

# Factorías, Factory Method y Abstract Factory

*(Patrones de diseño)*

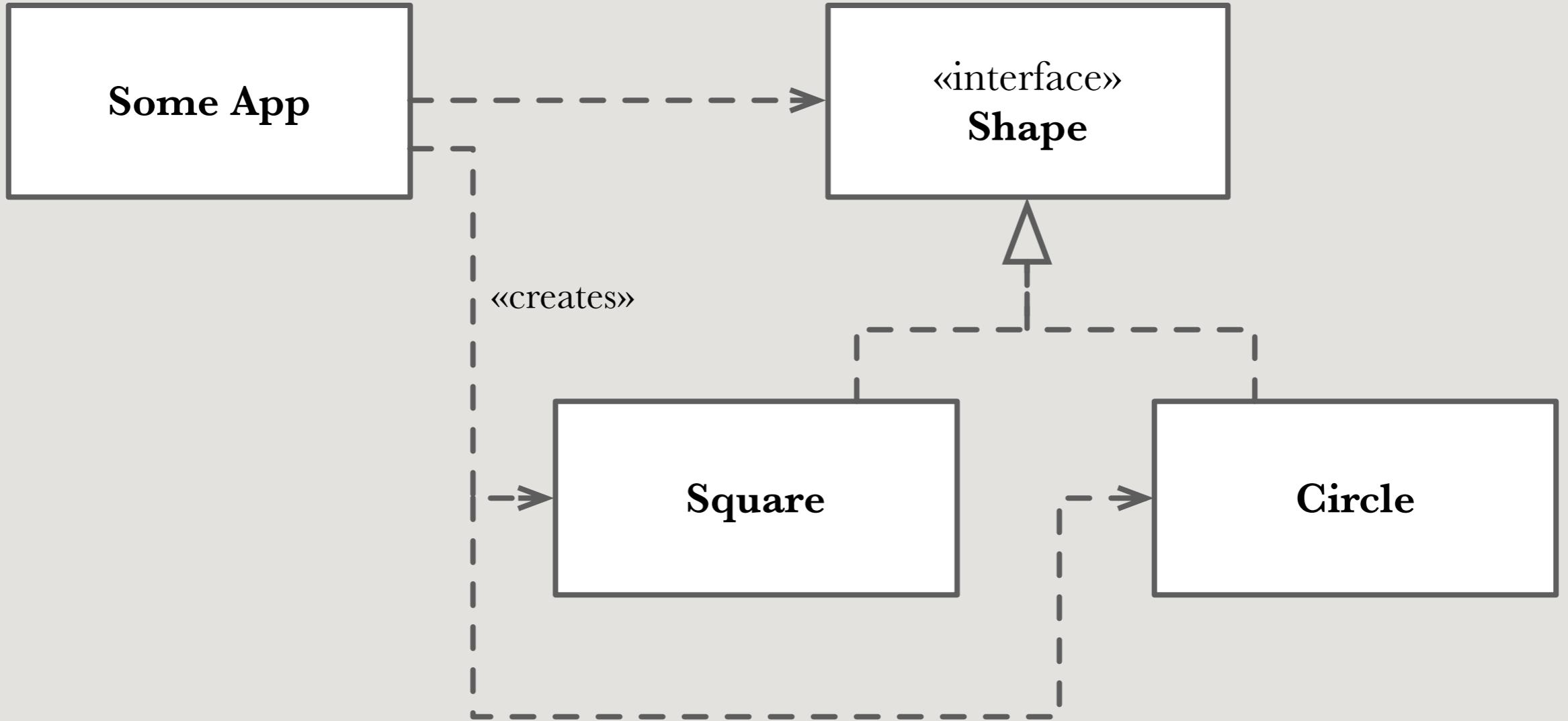
## Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

## *Principio de inversión de dependencias (DIP)*

*Deberíamos preferir depender  
de abstracciones y evitar las  
dependencias de clases concretas  
(especialmente cuando éstas  
pueden cambiar).*



## Viola el principio de dependencia de inversión (DIP)

La clase «SomeApp» se relaciona con la interfaz de las figuras («Shape») para llevar a cabo su labor (la que sea). Hasta ahí, bien. Es justo lo que buscamos siempre: no depender de las implementaciones concretas, sino tratar de manera uniforme a los distintos objetos a través del polimorfismo. Sin embargo, y pese a que no utiliza ningún método específico que pudieran tener «Square» o «Circle», al estar creando instancias de ellas depende también de esas implementaciones concretas.

Shape shape = new Circle(origin, 150);

# ¿Qué pasa con «new»?

- ¿No rompe el principio de programar para una interfaz, en vez de para una implementación?

```
Pato pato = new AnadeReal();
```



Queremos usar interfaces para mantener el código flexible.

# ¿Qué pasa con «new»?

- ¿No rompe el principio de programar para una interfaz, en vez de para una implementación?

Pato pato = new AnadeReal();

Queremos usar interfaces para mantener el código flexible.

¡Pero tenemos que crear un objeto de una clase concreta!

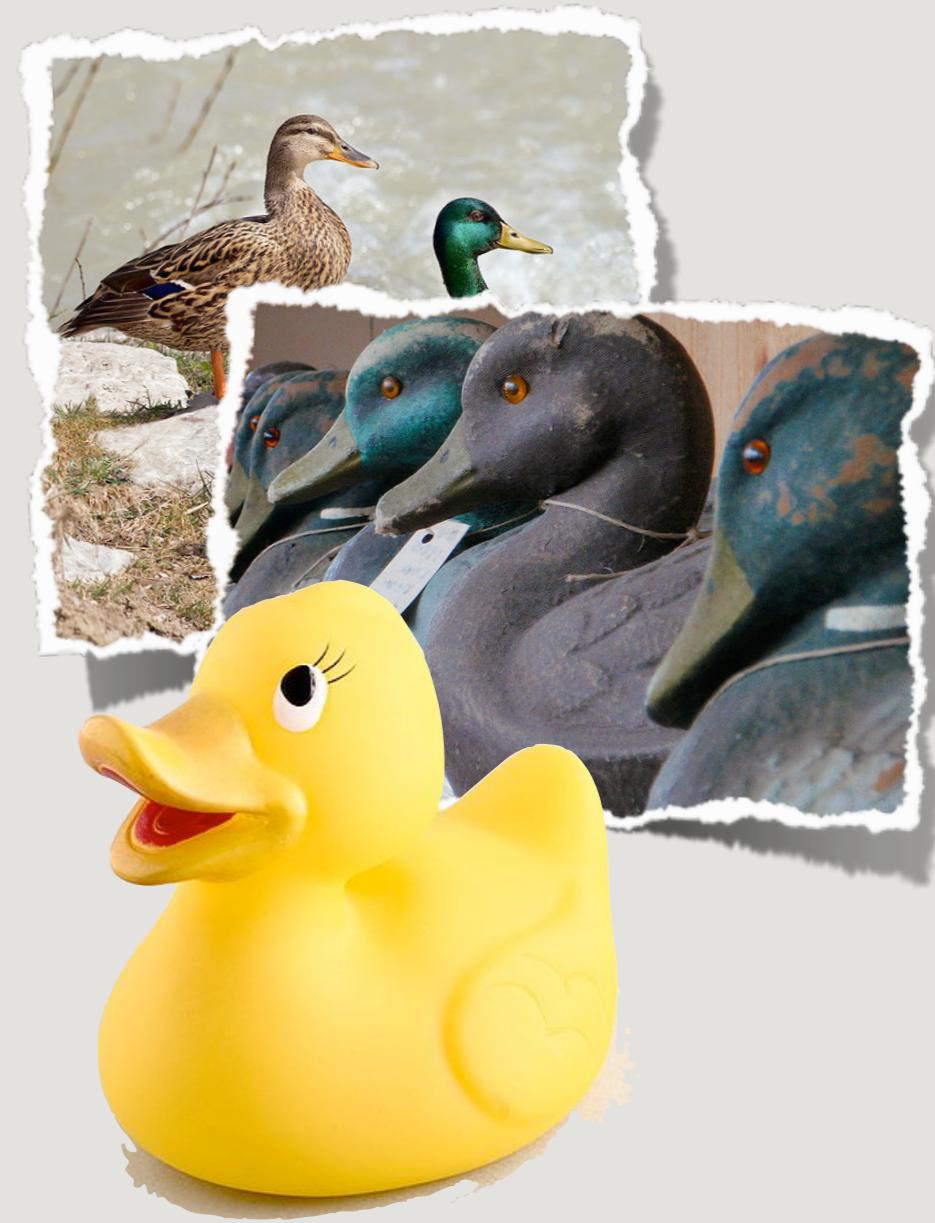
(Acoplamiento)



# ¿Qué pasa con «new»?

- Además, es frecuente código como éste, cuando hay que crear objetos de clases relacionadas:

```
Pato pato;  
if (picnic) {  
    pato = new AnadeReal();  
} else if (caza) {  
    pato = new PatoDeReclamo();  
} else if (baño) {  
    pato = new PatoDeGoma();  
}
```



# ¿Qué pasa con «new»?

- Es decir, tenemos una serie de clases concretas que «instanciar», y la decisión de cuál debe ser sólo se puede tomar en tiempo de ejecución
- Cada vez que aparecen nuevas clases (o se eliminan) hay que modificar ese código
- Lo malo no es este código (inevitablemente tendrá que aparecer en algún sitio)
  - Sino que muy frecuentemente aparece en varios sitios de la aplicación
  - Violando así también el principio de «abierto-cerrado»

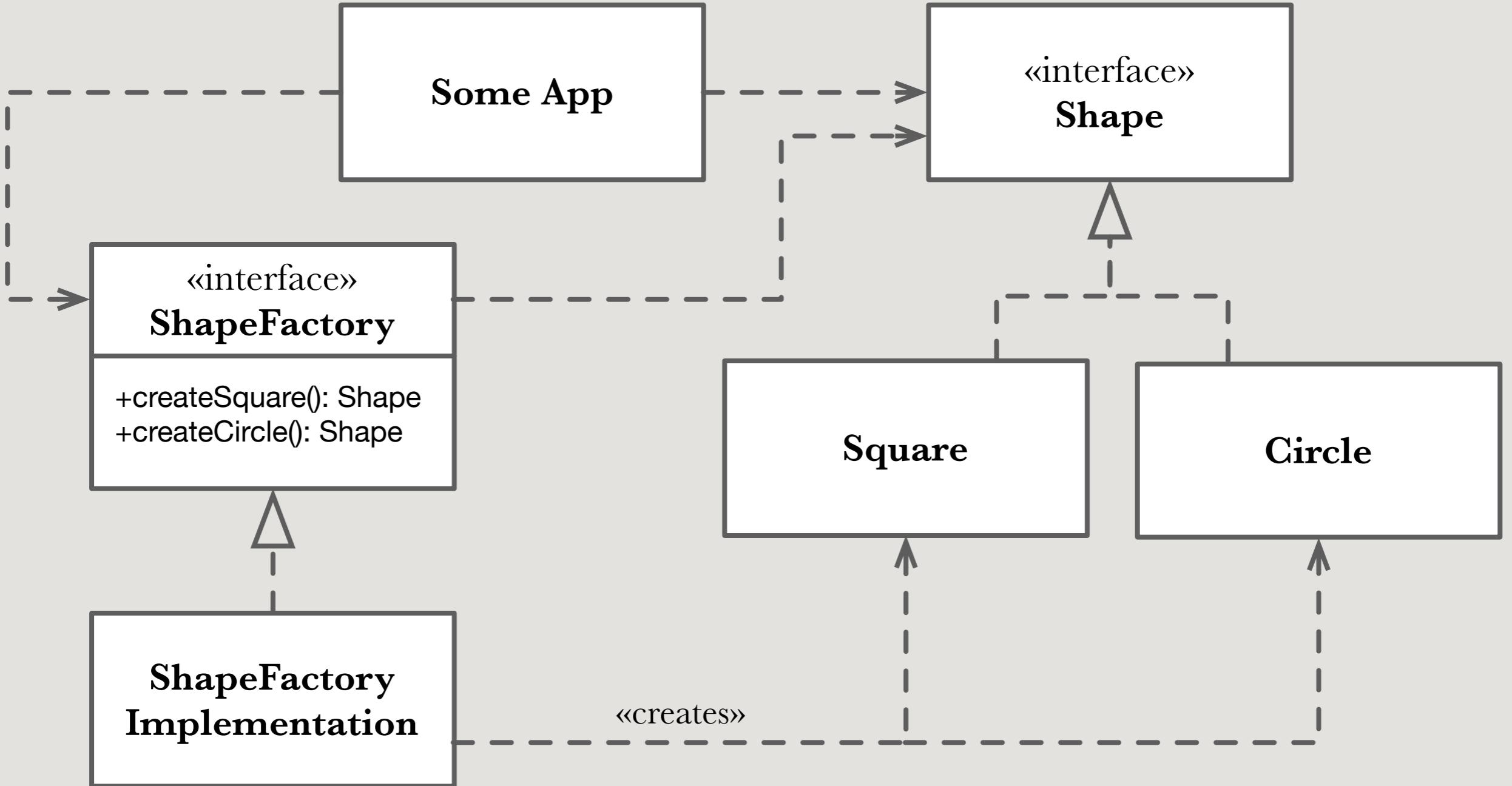
cuando programamos para una interfaz sabemos que nos estamos aislando de un montón de cambios que pueden suceder durante la implementación, ya que funcionará con cualesquiera nuevas clases que implementen la interfaz, a través del polimorfismo. Sin embargo, aquí estamos haciendo uso de clases concretas. Cuando aparezcan nuevas clases (o desaparezcan), habrá que modificar dicho código (deja de estar «abierto para la extensión, pero cerrado para la modificación»).

PERO EN ALGÚN MOMENTO HAY QUE  
CREAR REALMENTE EL OBJETO Y JAVA  
SÓLO PROPORCIONA UN MODO DE  
HACERLO, ¿NO? ¿ENTONCES?



*Otro de los principios básicos del diseño OO*

*Identificar aquellos aspectos  
que varían y separarlos de lo  
que tiende a permanecer igual.*



Ahora, el código de la aplicación ya no depende de las clases concretas círculo o cuadrado, aunque sigue creando objetos de ellas (a través de la factoría). La aplicación manipula dichos objetos únicamente a través de la interfaz común (figura) y nunca invoca métodos específicos de círculo o cuadrado.

# Otra variante

```
public Shape createShape(String shapeName)
    throws IllegalArgumentException
{
    if (shapeName.equals("circle"))
        return new Circle();
    else if (shapeName.equals("square"))
        return new Square();
    else
        throw new IllegalArgumentException(
            "ShapeFactory no puede crear " + shapeName);
}
```

Eliminamos la necesidad de introducir un método nuevo cada vez que aparezca una figura, aunque a cambio hay que añadir una nueva rama condicional y perdemos la comprobación estática de tipos. Además, si los constructores reciben distintos parámetros (como sería lo más lógico en este caso) o no se podría hacer o se oscurecería la intención del código. Pero en ocasiones el método parametrizado puede ser una alternativa perfectamente válida.

# Ejemplo: Pizzería



A través de un ejemplo (un poco demasiado artificial, es cierto) extraído del libro «Head First Design Patterns» (2004) empezaremos creando una factoría y terminaremos viendo el patrón de diseño Factory Method.

# Ejemplo: pizzería

```
Pizza orderPizza() {  
    Pizza pizza;  
  
    pizza = new Pizza();  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

No nos sirve, porque queremos crear varios tipos de pizza (cuatro quesos, Pepperoni, vegetariana...)

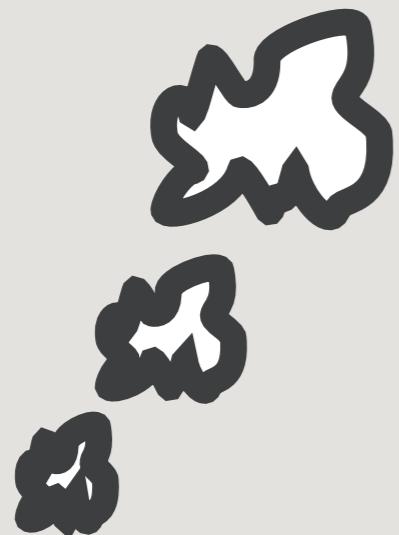
# Varios tipos de pizza

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

cuando se pide una pizza, le pasamos el tipo de pizza a crear

```
if (type.equals("cheese")) {  
    pizza = new CheesePizza();  
} else if (type.equals("greek")) {  
    pizza = new GreekPizza();  
} else if (type.equals("pepperoni")) {  
    pizza = new PepperoniPizza();  
}
```

¿QUÉ OCURRE SI AÑADIMOS  
NUEVOS TIPOS DE PIZZA (O SI  
ELIMINAMOS ALGUNO)?



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

En este caso, parece razonable pensar que ésta es la parte que más propensa es a cambiar con el tiempo, porque se cambie la selección de pizzas.

```
...
```

```
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;
```

```
}
```



Mientras que esta otra parte (preparar la pizza, es decir, hacer la masa, poner la salsa, y añadir los ingredientes, meterla al horno, cortarla y empaquetarla) es de esperar que no cambie, pues es lo que se ha venido haciendo años y años.

---

Nota: en este diseño, cada tipo de pizza sabe cómo prepararse a sí misma. En muchos casos, en problemas reales, esto no tiene por qué ser necesariamente así.

# Factoría

- Si aplicamos el principio anterior (encapsular el concepto que varía), podemos sacar la lógica de la creación del objeto fuera del método `orderPizza`
- ¿A dónde?
  - A un objeto aparte que llamaremos `PizzaFactory`

# Fábrica de 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;  
    }  
}
```

# Cuestiones

## ○ ¿Ganamos algo?

- Después de todo, el código sigue estando parametrizado por el tipo de pizza, y lo único que hemos hecho ha sido trasladar el problema a otro objeto

Efectivamente, así es (y en un ejemplo tan tonto no se aprecia ninguna ventaja aparente —aparte de la legibilidad del código y de que cada método y, a ser posible, cada clase, se dediquen a una sola cosa, que por sí sólo posiblemente ya hiciera que mereciese la pena separarlo—). Pero hemos de pensar que muchas veces dicha lógica se repite en distintos puntos del programa. La clave está en eliminar la creación de los objetos concretos de la lógica del cliente, de manera que sólo haya que ir a un sitio cuando las clases concretas cambien.

Otra ventaja es que los clientes no conocen las pizzas concretas, sino únicamente el tipo base Pizza, como veíamos en el primer ejemplo de las figuras.

# Cuestiones

- ¿Se podría haber hecho con un método estático?

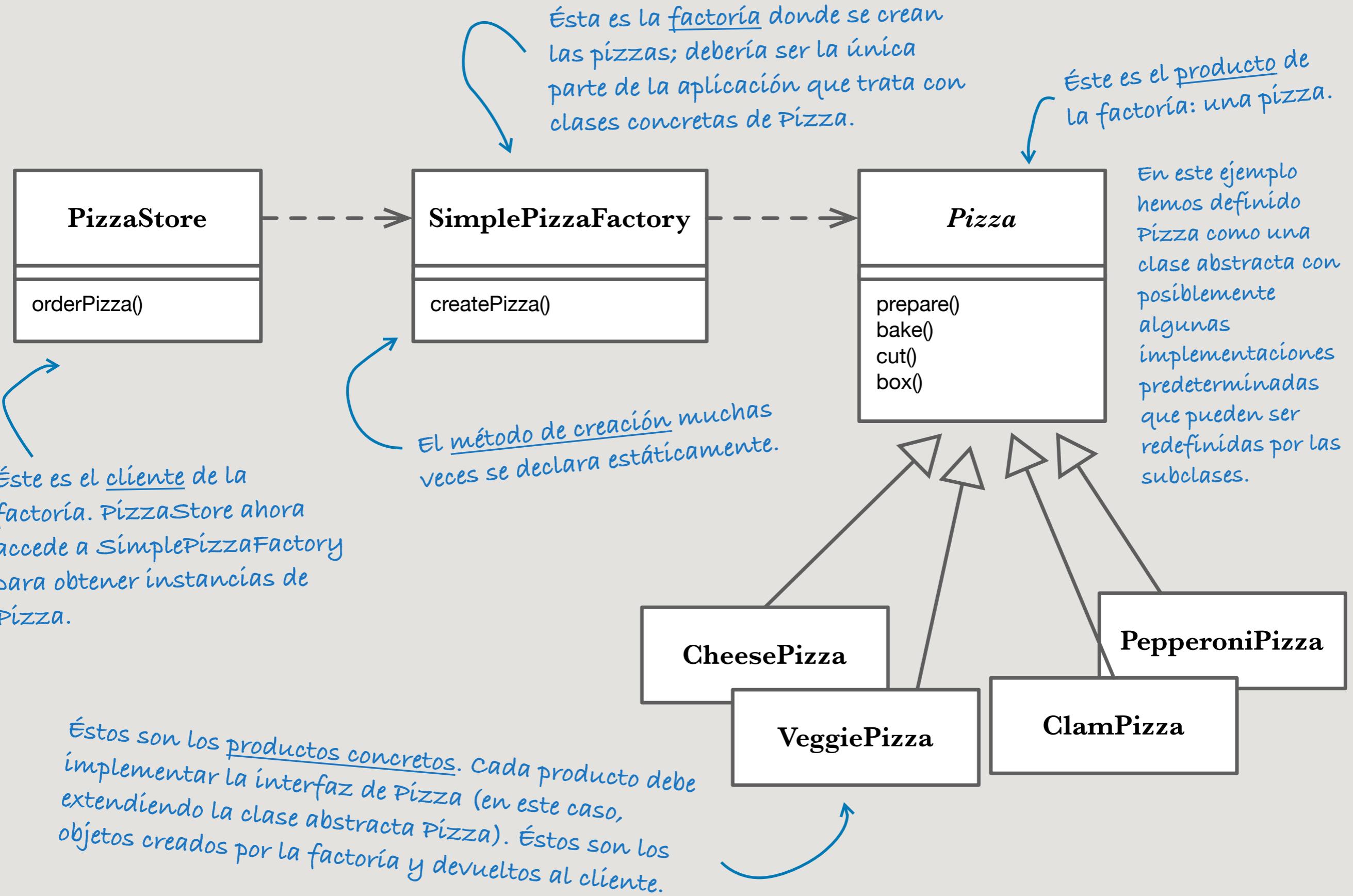
Sí, y de hecho es bastante frecuente. Tiene la ventaja de que no hay que crear el objeto factoría en sí. Pero tampoco permite crear una subclase de la factoría que redefina el comportamiento del método de creación, si fuese necesario.

# ¿Y cómo queda la clase PizzaStore?

```
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;  
    }  
}
```

Como siempre que tenemos esta estructura (una clase que delega en un objeto de otra clase que guarda como referencia)... ¿se os ocurre qué ganancia tendríamos en flexibilidad?

**¿Qué tenemos hasta ahora?**



# Una aclaración

- El ejemplo anterior no se corresponde con ningún patrón de diseño como tal
  - Es más bien una mera aplicación de los principios de diseño orientado a objetos y buenas prácticas de programación que hemos visto hasta ahora
- Se abusa mucho del término «factoría» y prácticamente cada persona se lo llama a cosas distintas:
  - Factory Method
  - Abstract Factory
  - etcétera

# Una aclaración

- **Nosotros lo usaremos en sentido amplio:**
  - Un método de creación es cualquier método (estático o no) que devuelve una instancia de un objeto
  - Una clase es una factoría (*Factory*) si implementa uno o más métodos de creación

# **Encapsulate Classes with Factory**

*A continuación se describe esta factorización extraída  
del catálogo Refactoring to Patterns*



*The Addison-Wesley Signature Series*

MARTIN FOWLER  
Book A SIGNATURE

# REFACTORING TO PATTERNS

JOSHUA KERIEVSKY



*Forewords by Ralph Johnson and Martin Fowler  
Afterword by John Brant and Don Roberts*

Software Development  
15th Annual Productivity Award  
Refactoring to Patterns  
by Joshua Kerievsky

# Refactoring to Patterns

- Libro de Joshua Kerievsky (Addison Wesley, 2005)
- Catálogo de factorizaciones dirigidas por patrones
- Se llega a los patrones a partir del código, no de un diseño previo

# Encapsulate Classes with Factory

*Los clientes instancian directamente clases que residen en un paquete e implementan una interfaz común.*



Hacer los constructores de las clases no públicos y dejar que los clientes creen objetos de ellas usando una factoría.

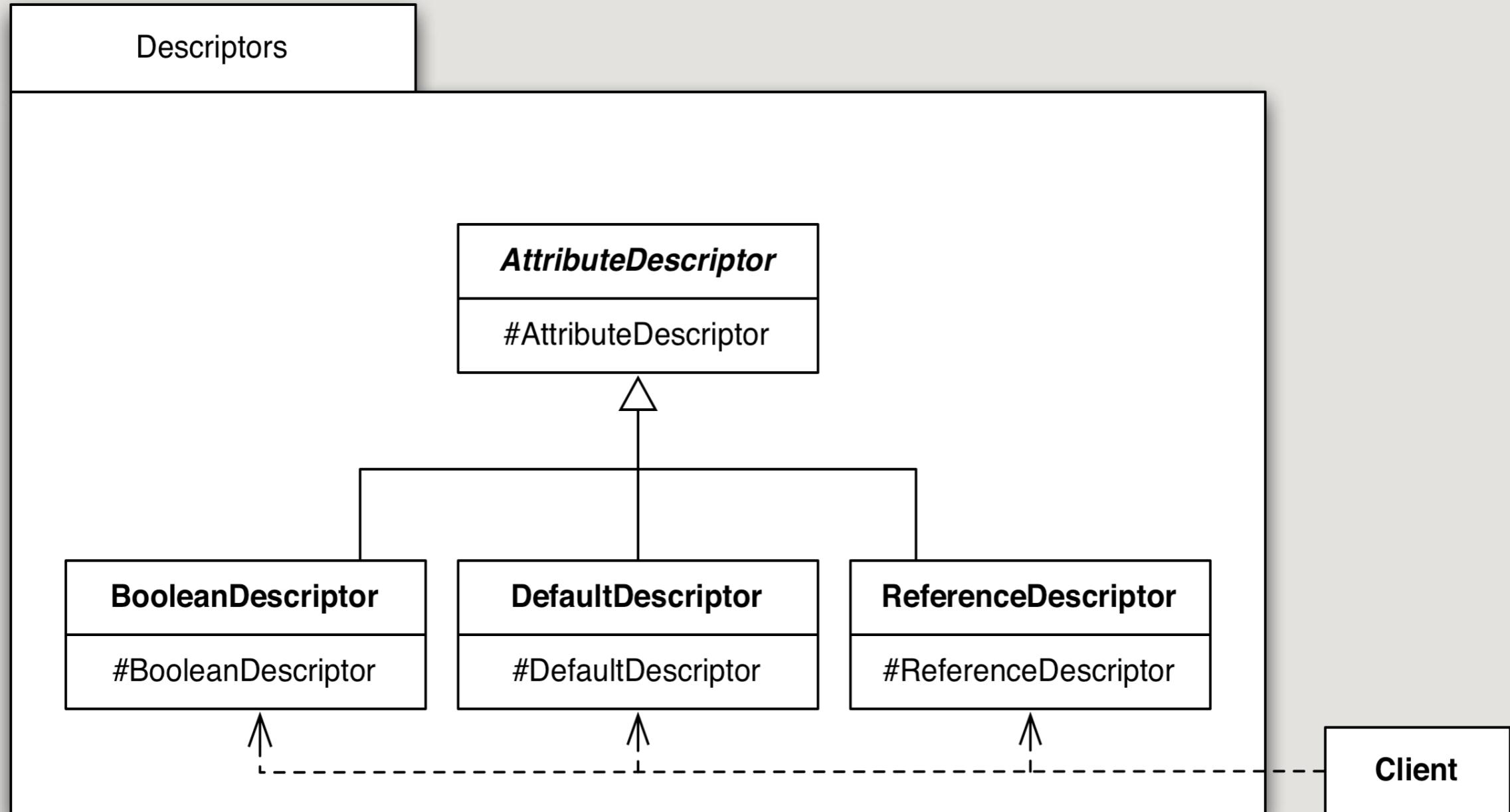
# Motivación

- ¿Qué ocurre si el cliente no necesita conocer la existencia de las clases concretas, sino sólo la interfaz común que implementan?
- Las clases en el paquete pueden ocultarse a los clientes y dar a una clase de fabricación la responsabilidad de crear y devolver dichos objetos

# Ejemplo

- Supongamos un código encargado de pasar objetos a una base de datos relacional
- Tenemos una serie de clases que hacen la correspondencia entre los atributos de la base de datos y las variables de los objetos:
  - BooleanDescriptor,
  - DefaultDescriptor,
  - ReferenceDescriptor...

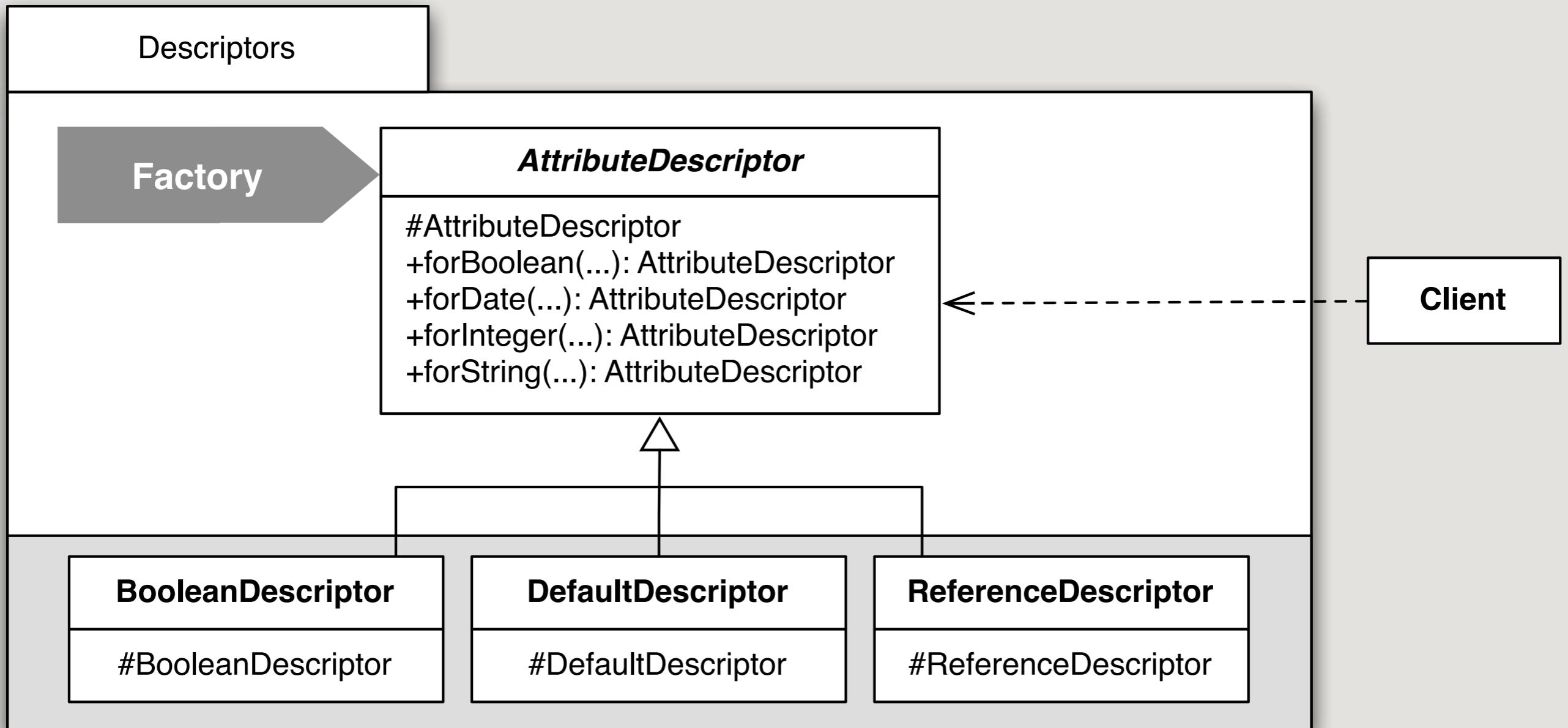
# Antes de usar una factoría



# Antes de usar una factoría

```
protected List createAttributeDescriptors()
{
    List result = new ArrayList();
    result.add(new DefaultDescriptor("remoteId",
        getClass(), Integer.TYPE));
    result.add(new DefaultDescriptor("createdDate",
        getClass(), Date.class));
    result.add(new DefaultDescriptor("createdBy",
        getClass(), User.class, RemoteUser.class));
    ...
    return result;
}
```

# Con métodos de creación



# Con métodos de creación

```
protected List createAttributeDescriptors()
{
    List result = new ArrayList();
    result.add(
        AttributeDescriptor.forInteger("remoteId",
            getClass()));
    result.add(
        AttributeDescriptor.forDate("createdDate",
            getClass()));
    ...
    return result;
}
```

# Resultado

- El acceso a las subclases de `AttributeDescriptor` se hace a través de esa clase base común
- Los clientes obtienen instancias concretas a través de la interfaz de `AttributeDescriptor`
- Previene que los clientes creen objetos de las subclases directamente
- Comunica a otros programadores que las subclases de `AttributeDescriptor` no están pensadas para ser públicas

# Ventajas

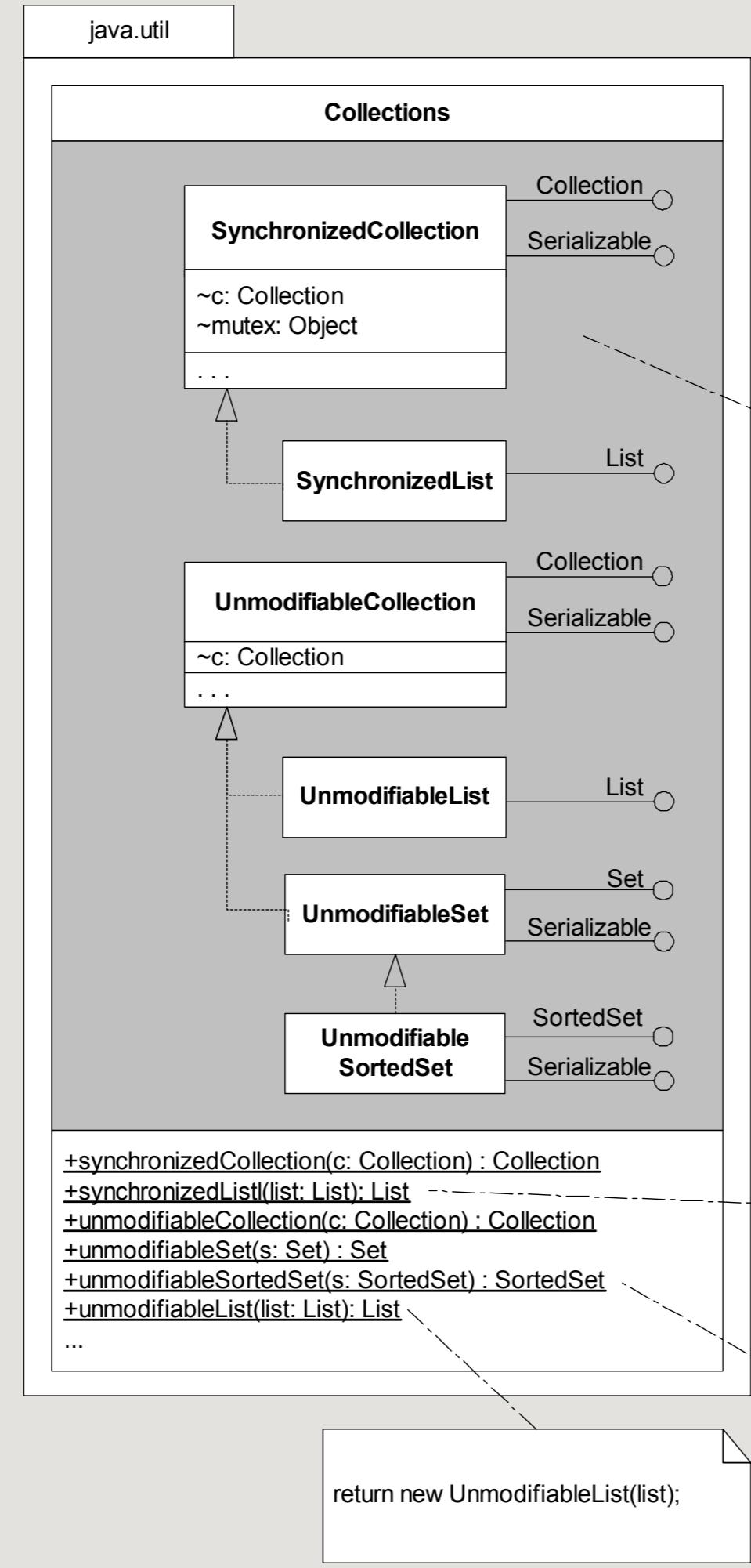
- Aplicación del principio «programar para una interfaz, no para una implementación»
- Reduce el «peso conceptual» del paquete ocultando clases innecesarias
- Simplifica la creación de objetos mediante una serie de métodos de creación que revelan su intención

# Inconveniente

- Cada vez que se añada una nueva subclase (o se modifique el constructor de una existente) hay que añadir un nuevo método de creación a la factoría

# Variación: clases internas

- En Java otro modo de ocultar las clases concretas es mediante «inner classes»
- Ejemplo: `java.util.Collections`
  - Esta clase proporciona un modo de hacer listas, conjuntos y diccionarios (*maps*) inmodificables y sincronizados
  - Johua Bloch, su autor, emplea el patrón Proxy en forma de clases internas no públicas



# Otro ejemplo en la API de Java

- **EnumSet (en `java.util`)**
- **No tiene constructores públicos, sólo métodos de fabricación estáticos**
- **Devuelven dos posibles implementaciones:**
  - **RegularEnumSet (hasta 64 elementos)**
    - ▶ Usa un `long`
  - **JumboEnumSet (64 o más elementos)**
    - ▶ Un array

La existencia de ambas clases es invisible para los clientes. Se podría eliminar alguna de ellas sin que se percatasen, si ya no hubiese mejoras de rendimiento. Igualmente, futuras versiones de la API podrían proporcionar una tercera o cuarta implementación si implicasen mejoras en el rendimiento, y sin que los clientes ni siquiera supiesen de la existencia de esas nuevas implementaciones concretas.

# Otras factorizaciones

*Otra factorizaciones extraídas del mismo libro, relacionadas con la creación de objetos mediante factorías.*

# Otras factorizaciones

- *Replace Constructors with Creation Methods*
  - *Parameterized Creation Methods*
  - *Extract Factory*
- *Move Creation Knowledge to Factory*
- *Introduce Polymorphic Creation with Factory Method*

# Replace Constructors with Creation Methods

*Una clase tiene muchos constructores y es difícil saber a cuál hay que llamar.*



Reemplazar los constructores con métodos de creación que revelen su intención y devuelvan instancias de objetos.

## Loan

```
+Loan(notional, customerRating, maturity)
+Loan(notional, customerRating, maturity, expiry)
+Loan(notional, outstanding, customerRating, maturity, expiry)
+Loan(capitalStrategy, notional, customerRating, maturity, expiry)
+Loan(capitalStrategy, notional, outstanding, customerRating, maturity, expiry)
```



## Loan

```
-Loan(capitalStrategy, notional, outstanding, customerRating, expiry, maturity)
+createTermLoan(notional, customerRating, maturity) : Loan
+createTermLoan(capitalStrategy, notional, outstanding, customerRating, maturity) : Loan
+createRevolver(notional, outstanding, customerRating, expiry) : Loan
+createRevolver(capitalStrategy, notional, outstanding, customerRating, expiry) : Loan
+createRCTL(notional, outstanding, customerRating, maturity, expiry) : Loan
+createRCTL(capitalStrategy, notional, outstanding, customerRating, maturity, expiry) : Loan
```

# Variaciones

## ● *Parameterized Creation Methods*

- Si, por ejemplo, tenemos 50 métodos de creación
- Otra opción es dar métodos de creación sólo para los más usuales

## ● *Extract Factory*

- Cuando los métodos de creación comienzan a ser más prominentes en la interfaz de la clase que las responsabilidades propias de ésta
- En ese caso, se puede sacar fuera la lógica de creación, a una clase factoría
- No confundir con el patrón Abstract Factory [GoF]

# Ejemplo en Java

- ¿Qué creéis que crea este constructor?

`BigInteger(int, int, Random)`

- ¿Y este método estático (añadido en Java 1.4)?

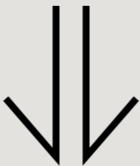
`probablePrime(int, Random)`

# Ventajas de los métodos estáticos de creación

- Tienen nombres
- No necesitan crear un nuevo objeto cada vez que son llamados
  - Boolean.valueOf(boolean)
- Pueden devolver un objeto de cualquiera de las subclases de su tipo de retorno

# Move Creation Knowledge to Factory

*Los datos y el código necesarios para instanciar un objeto están repartidos en numerosas clases.*

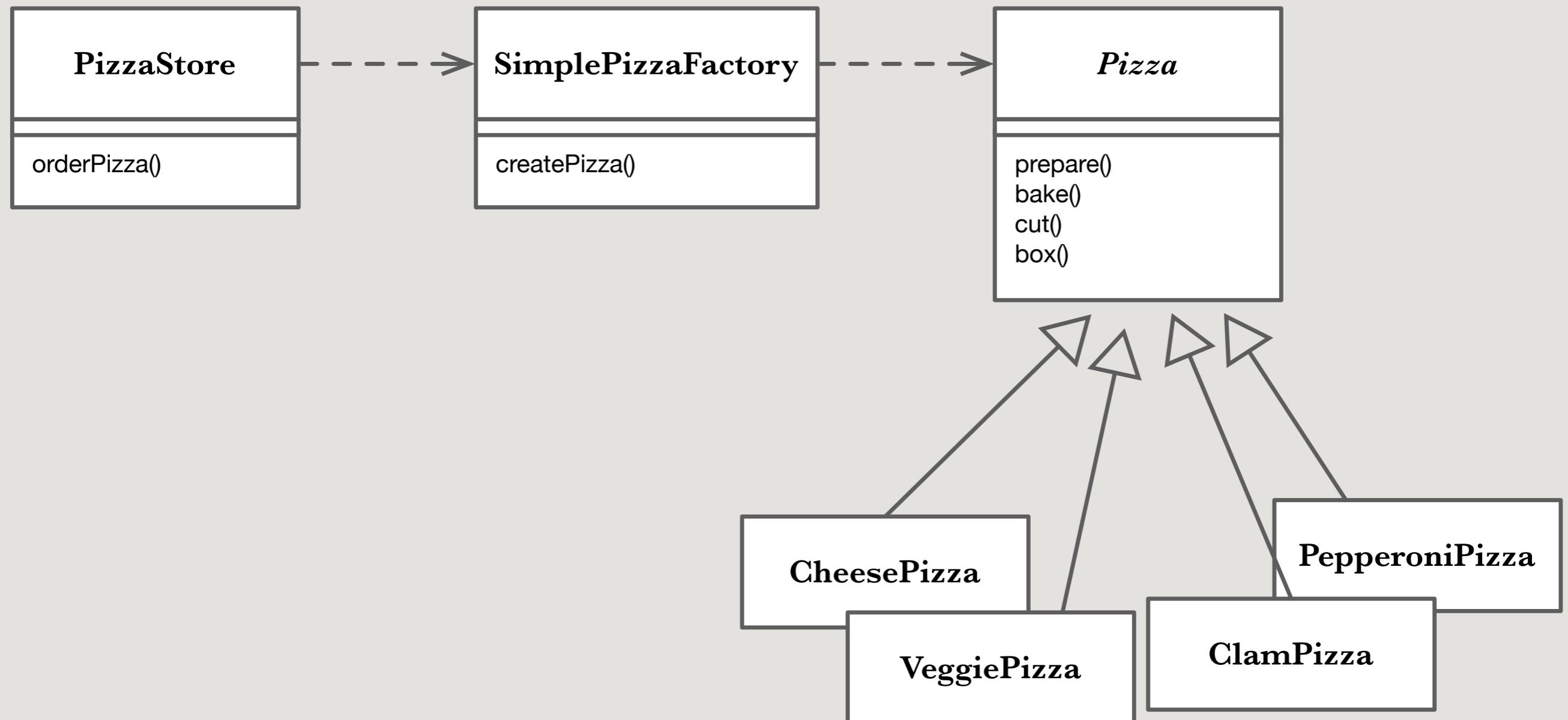


Mover la lógica de creación a una única clase factoría.

# Volvamos a nuestro ejemplo

supondremos que hacemos de nuestra pizzería una franquicia, con pizzerías en distintos lugares, cada una con sus particularidades. Veremos cómo, ahora sí, esos cambios nos llevarán a aplicar un patrón de diseño del GOF.

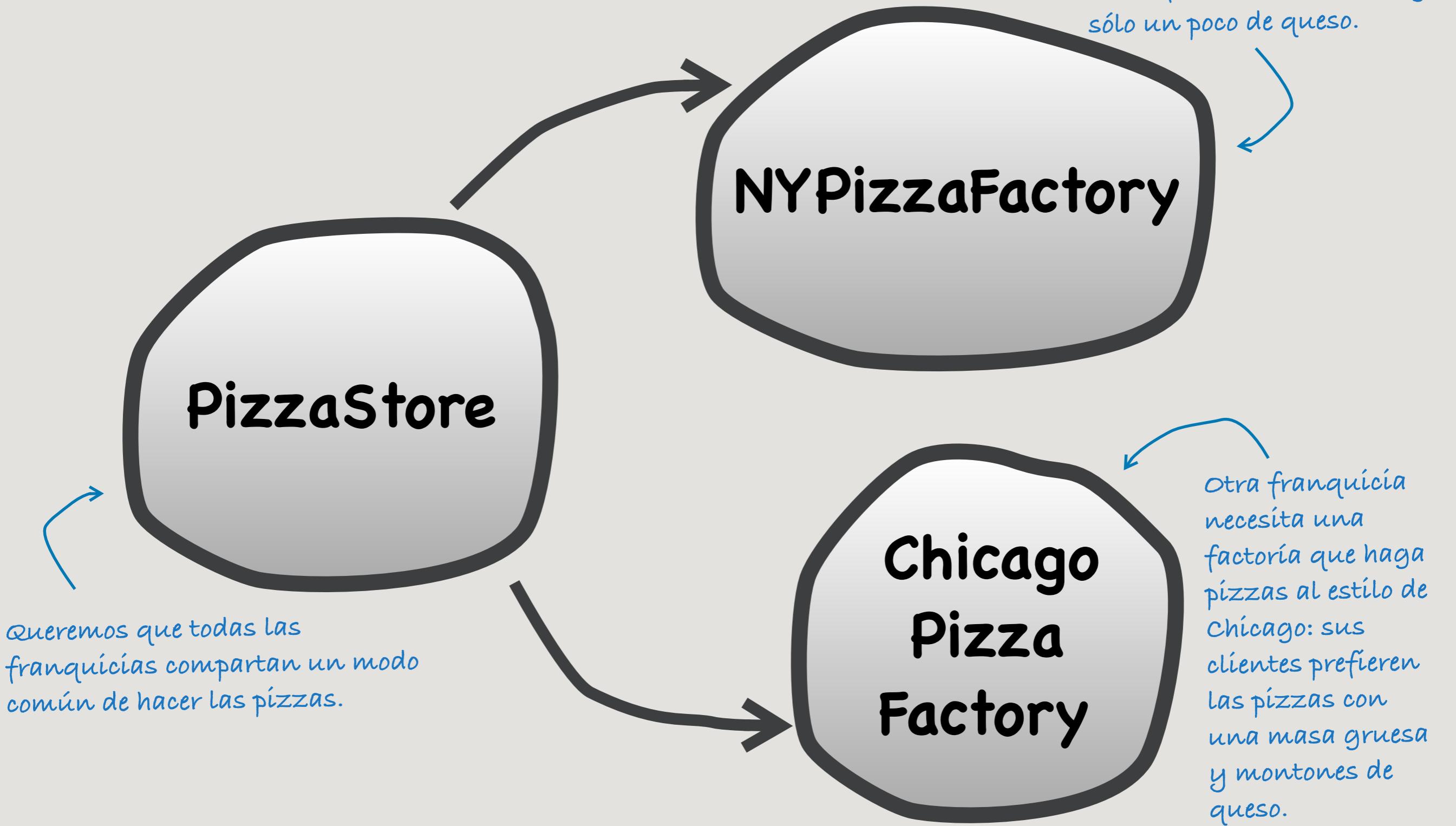
# Situación actual



# La franquicia

- Queremos mantener la calidad y el modo de operar de la franquicia que nos ha hecho famosos
  - Es decir, reutilizar la parte del código de elaboración de las pizzas que tiene que ver con el procedimiento general, con ese conocimiento, y que está bien probado
- Al mismo tiempo, cada franquicia quiere ofrecer distintos estilos de pizzas acorde a las diferencias de cada región

# Un enfoque



# Un enfoque

```
NYPizzaFactory nyFactory = new NYPizzaFactory();  
PizzaStore nyStore = new PizzaStore(nyFactory);  
nyStore.order("veggie");
```

```
ChicagoPizzaFactory chicagoFactory =  
    new ChicagoPizzaFactory();  
PizzaStore chicagoStore =  
    new PizzaStore(chicagoFactory);  
chicagoStore.order("veggie");
```

# Factoría simple

- Lo anterior sería lo que podríamos denominar una «factoría simple»
  - No se corresponde con ningún patrón de diseño como tal
  - Es simplemente una clase a la que hemos extraído la lógica de creación de objetos de un tipo determinado (pizzas)

# Factoría simple

- Se caracteriza porque configuramos cada tienda con un objeto de fabricación determinado
  - Composición de objetos
- Problema
  - Permite asociar a la tienda de pizzas de Nueva York una fábrica de pizzas al estilo de Chicago

Tal vez esa flexibilidad sea precisamente lo que busquemos en algunas ocasiones, y en ese caso este enfoque sería perfectamente válido. Pero... ¿no habría forma de lograr que las diferentes tiendas sean las responsables de saber qué tipo de pizza crear?

# Factoría simple

- Es decir, necesitamos una manera de volver a tener el código de creación de la pizza en la propia PizzaStore, pero siendo a la vez flexible

# Solución

## *Un framework para la pizzería*

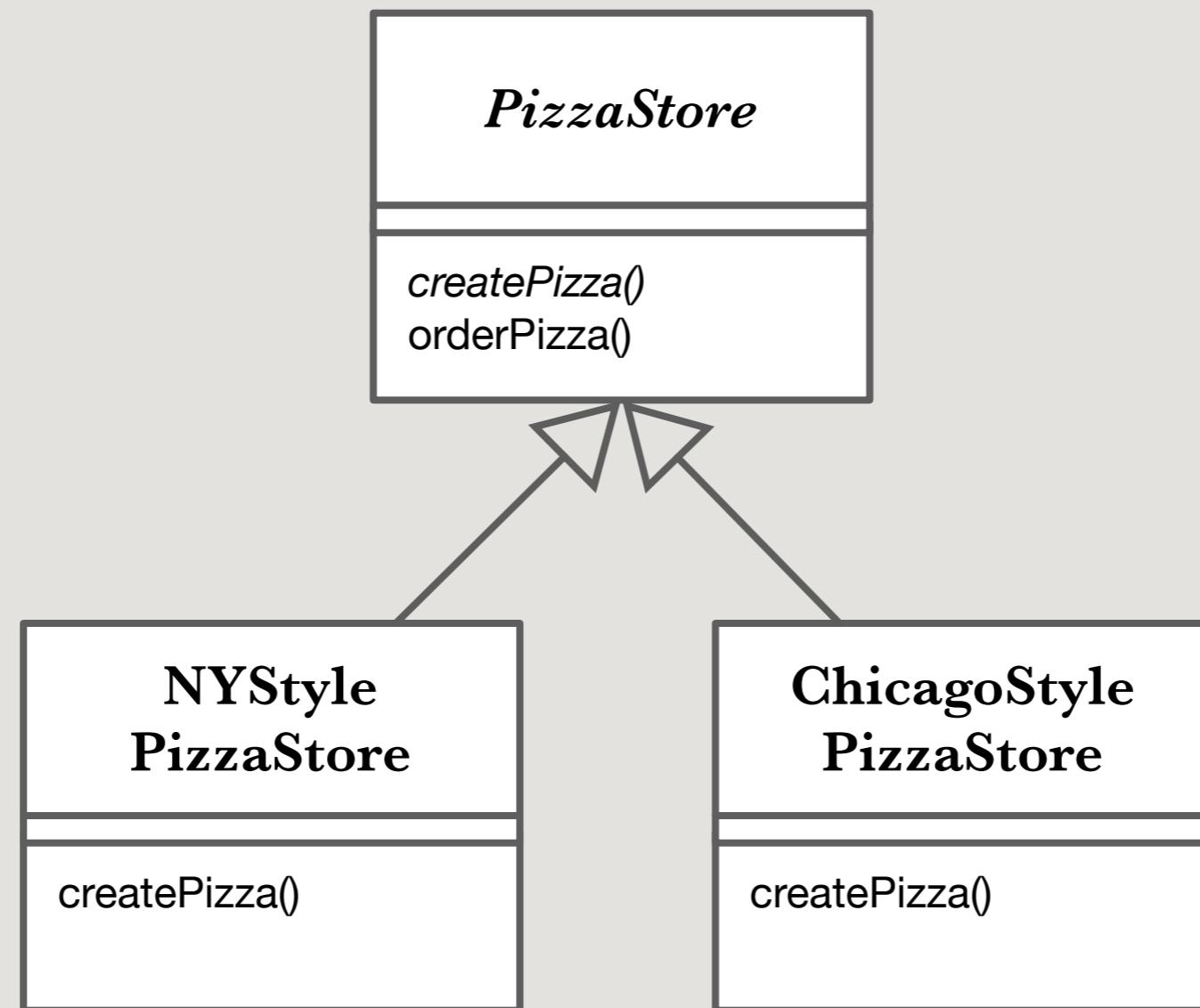
```
public abstract class PizzaStore {  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza = createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    abstract Pizza createPizza(String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

Ahora la creación del objeto Pizza vuelve a la clase PizzaStore, como una llamada a un método propio, en vez de al de la clase factoría.

Pero dicho método es abstracto.

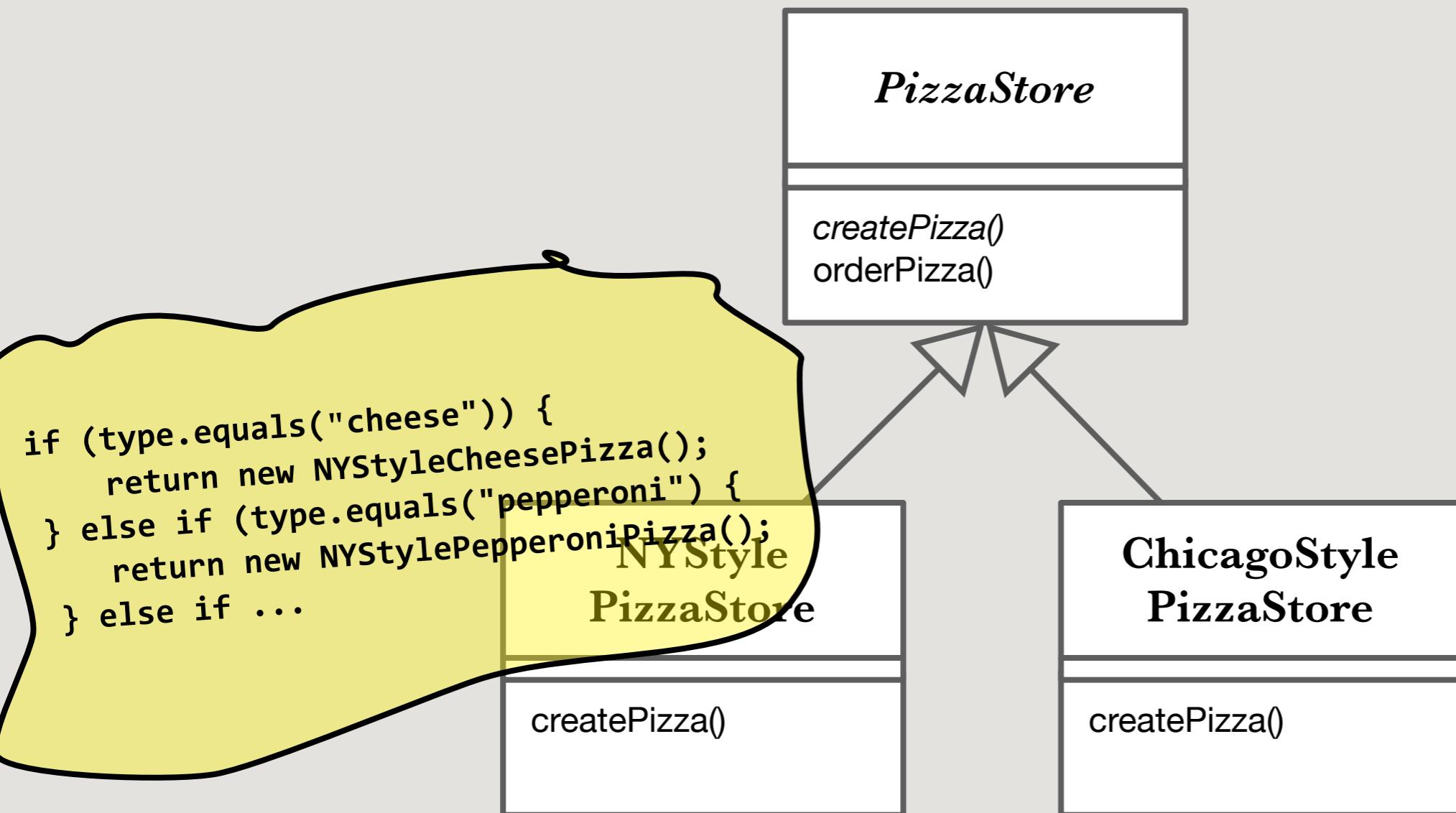
# Solución

## *Un framework para la pizzería*



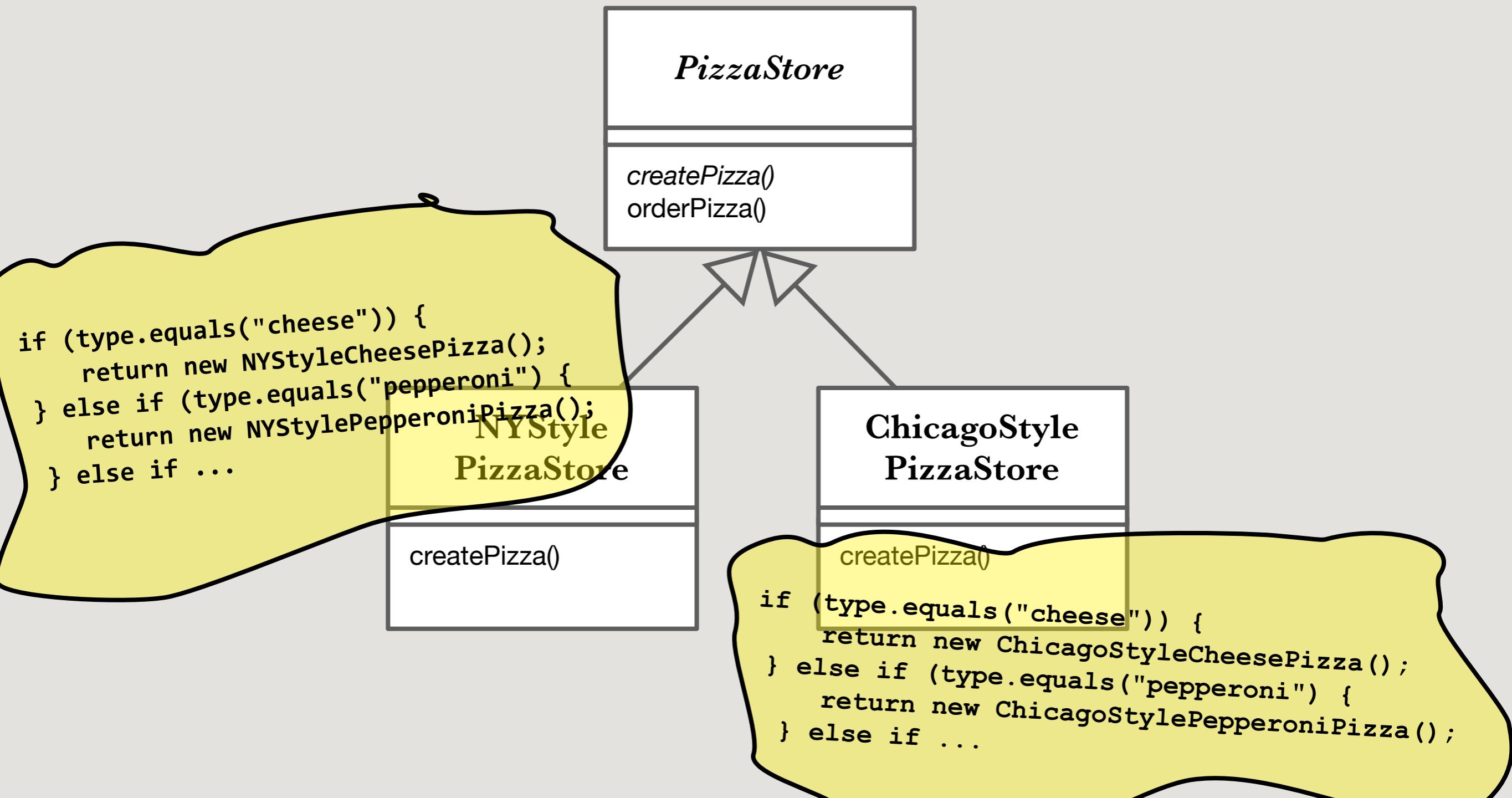
# Solución

## *Un framework para la pizzería*



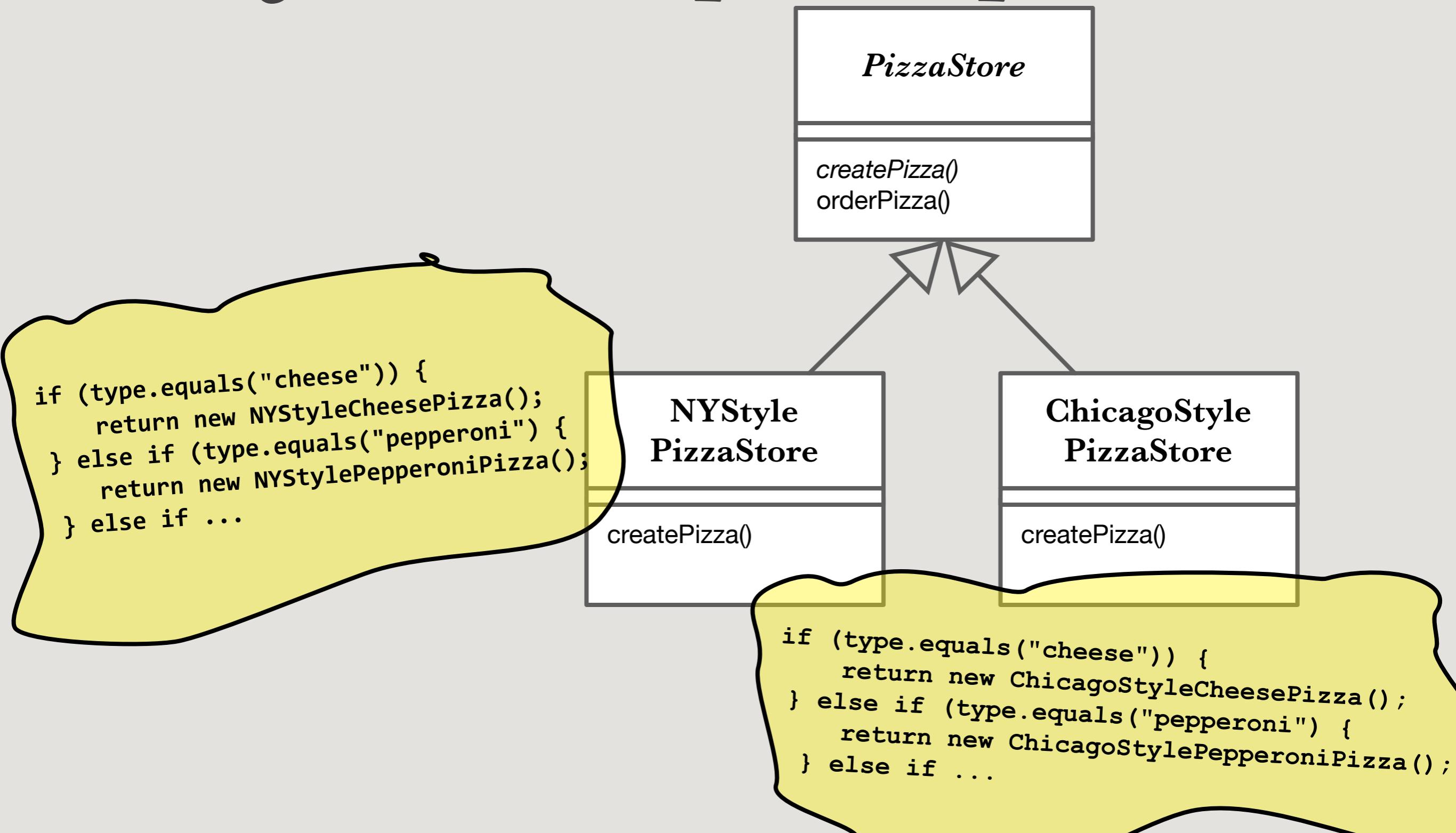
# Solución

## *Un framework para la pizzería*

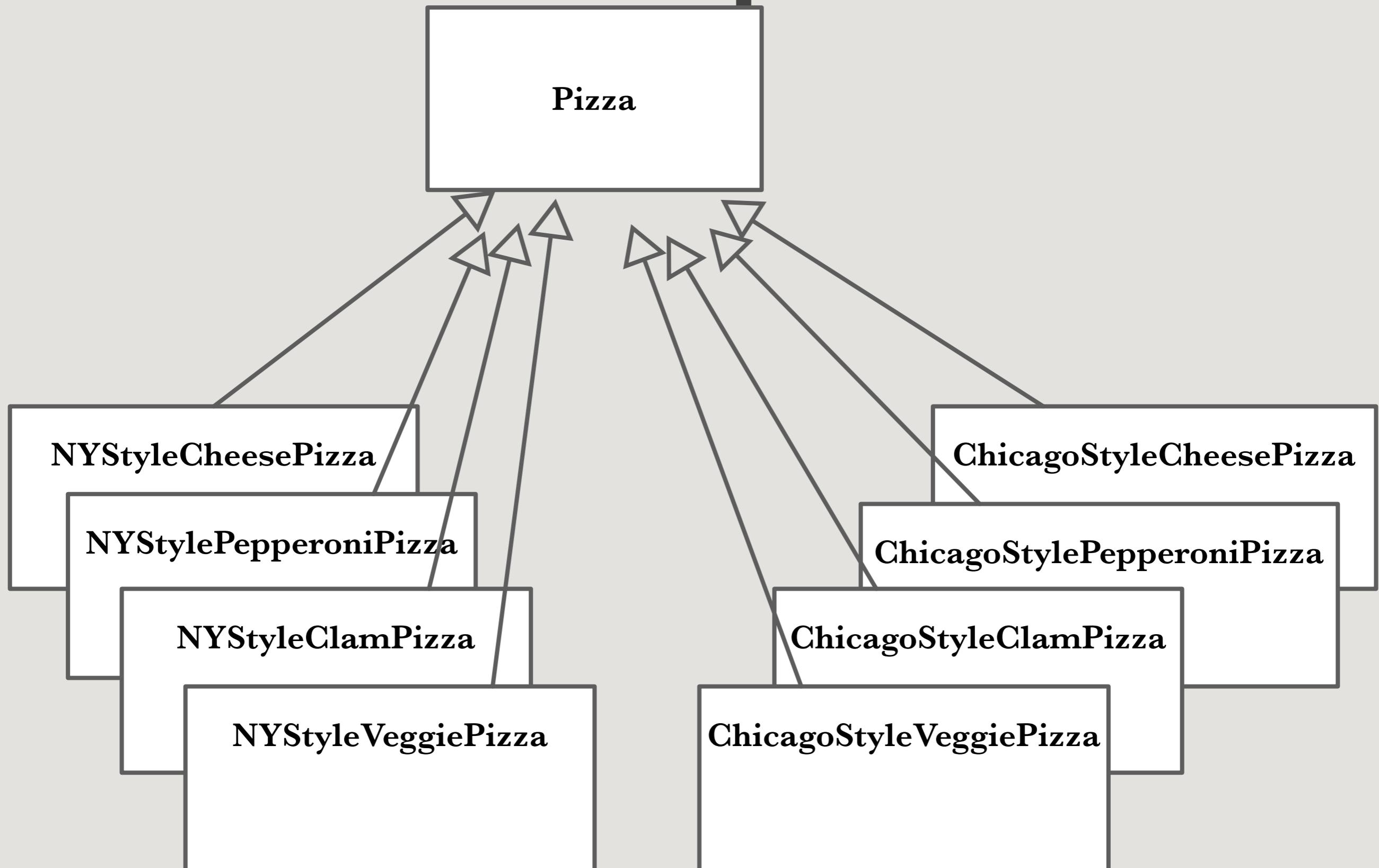


# Solución

## *Un framework para la pizzería*



# Las clases producto



*El método orderPizza() llama a createPizza() y recibe un objeto Pizza, pero...*

**¿Quién es el responsable de decidir qué objeto Pizza se crea?**

**LAS  
SUBCLASES**

# El patrón Factory Method

- Hemos llegado así a una implementación del patrón *Factory Method*

**abstract Product factoryMethod(type)**

Un «factory method» es abstracto: las subclases concretas deben redefinirlo por fuerza.

Devuelve un producto que normalmente es usado en los propios métodos de la superclase, sin falta de saber qué tipo concreto de producto es.

Opcionalmente, puede ser parametrizado (si hay que decidir entre distintas variaciones de un producto).

# Factory Method

# Factory Method (Método de fabricación)

- Patrón de creación (ámbito de clase)
- Propósito:

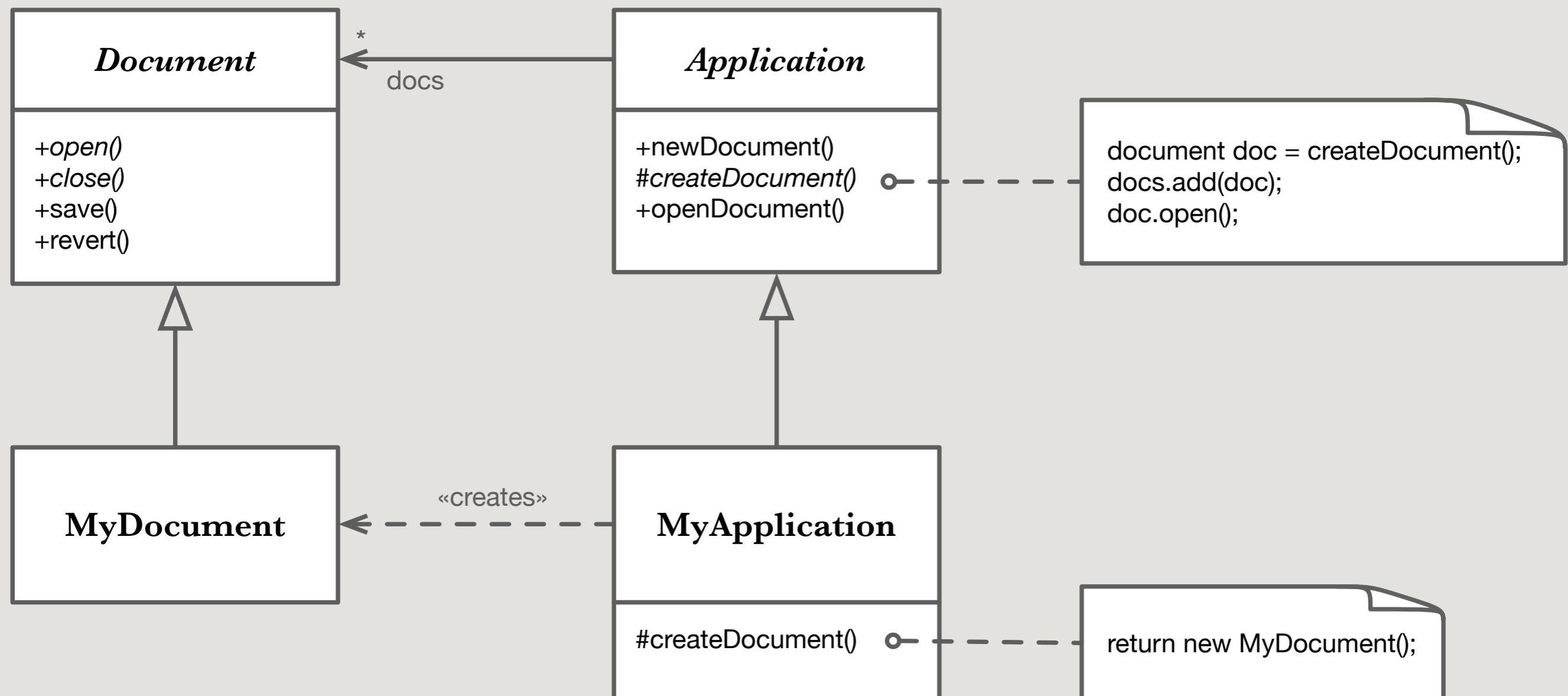
*Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan la clase del objeto a crear.*

- También conocido como:
  - Constructor virtual (Virtual Constructor)

# Motivación

- Sea un framework para la construcción de editores de documentos de distintos tipos
  - Clases abstractas `Application` y `Document`
    - Los clientes deberán heredar de ellas para implementar los detalles de cada aplicación concreta
    - P. ej., `DrawingApplication` y `DrawingDocument`
  - ¿Cómo se implementa la opción `New` del menú?
  - ¿Cómo sabe la clase `Application` qué tipo concreto de documento debe crear?

# Motivación

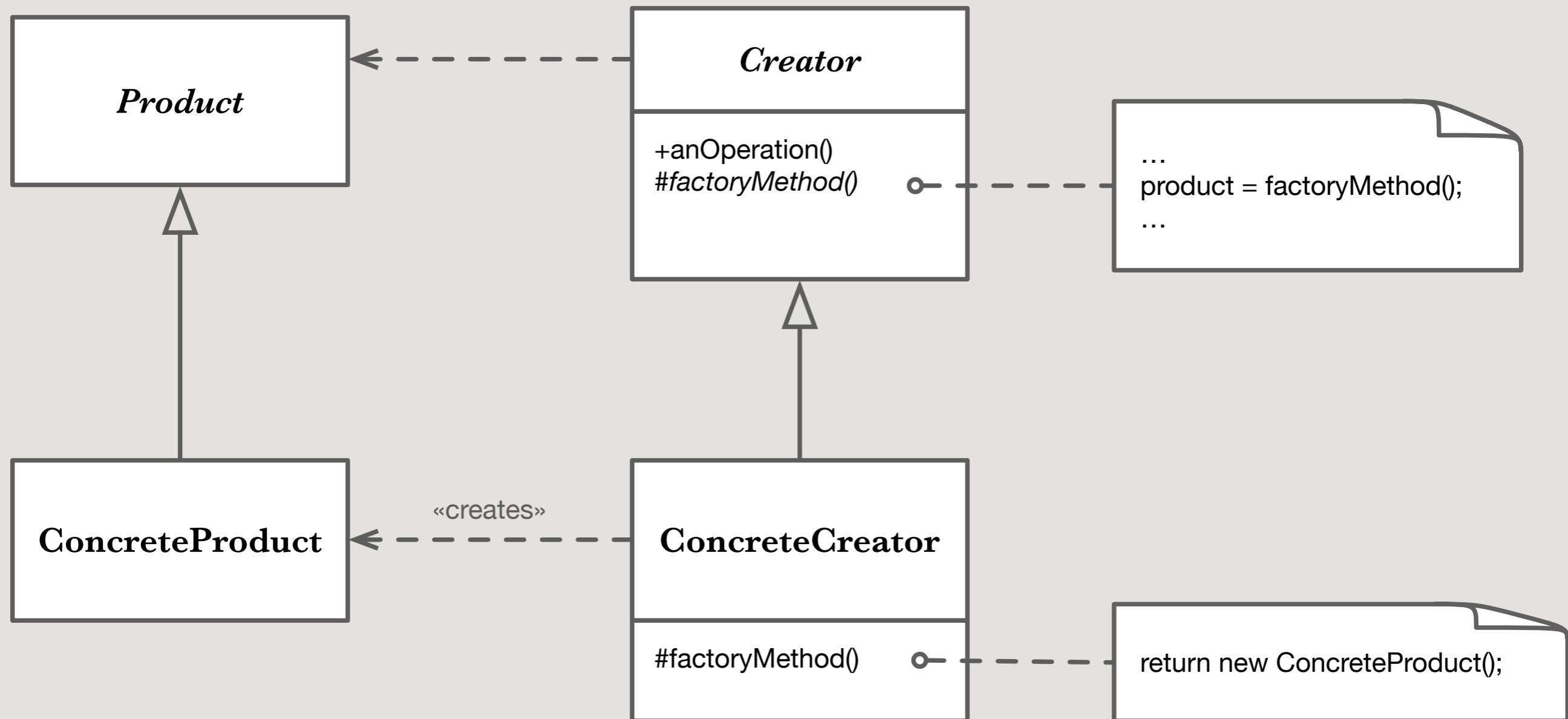


# Aplicabilidad

- Úsese cuando:

- Una clase no puede anticipar la clase de objetos que debe crear
- Una clase quiere que sus subclases especifiquen los objetos a crear
- Hay clases que delegan responsabilidades en una o varias subclases, y queremos localizar el conocimiento de qué subclase es el delegado

# Estructura



La estructura del patrón *Factory Method*

# Participantes

- **Product (Document)**

- Define la interfaz de los objetos creados por el método de fabricación

- **Concrete Product (MyDocument)**

- Implementa la interfaz Product

- **Creator (Application)**

- Declara el método de fabricación, que devuelve un objeto de tipo Product
  - Puede definir una implementación que devuelva el producto concreto predeterminado
  - Puede llamar a dicho método para crear un objeto producto

- **ConcreteCreator (MyApplication)**

- Redefine el método de fabricación para devolver un objeto ConcreteProduct

# Colaboraciones

- El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado

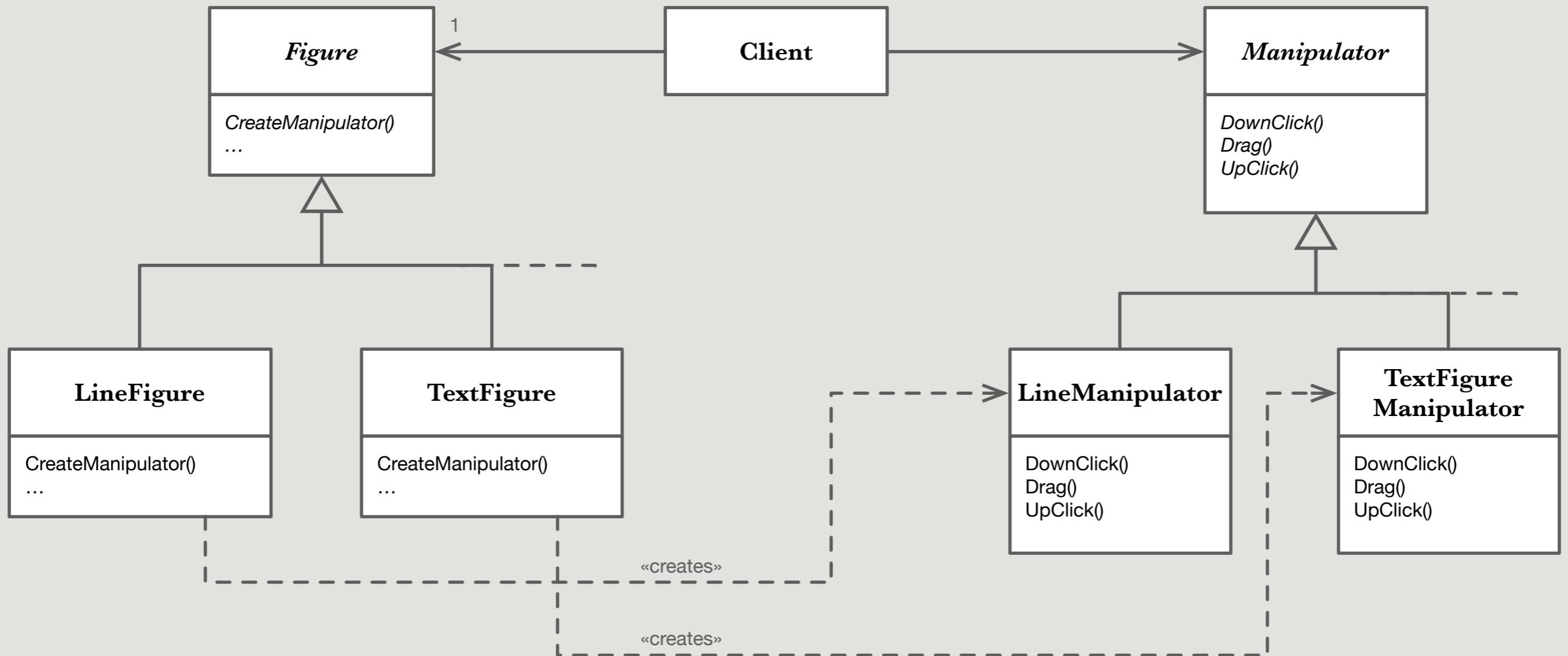
# Consecuencias

- Elimina la necesidad de enlazar clases específicas de la aplicación en el código
  - Sólo maneja la interfaz Product
    - ▶ Por lo que permite añadir cualquier clase ConcreteProduct definida por el usuario
- Inconveniente:
  - Tener que crear una subclase de Creator en los casos en los que ésta no fuera necesaria de no aplicar el patrón

# Jerarquías de clases paralelas

- En los ejemplos anteriores, el método de fabricación era llamado únicamente por el creador y sus subclases
- No tiene por qué ser siempre así: hay veces en que puede ser el cliente quien se encargue de ello
  - Por ejemplo, en los casos de jerarquías de clases paralelas

# Jerarquías de clases paralelas



# **Implementación**

# Métodos de fabricación parametrizados

- Una variante del patrón permite al método de fabricación crear varios tipos de productos
  - Mediante un parámetro que indica el tipo de objeto a crear

```
Encryption createEncryption(Key key)
throws NoSuchAlgorithmException
{
    String algorithm = key.getAlgorithm();
    if ("DES".equals(algorithm))
        return new DESEncryption(key);
    if ("RSA".equals(algorithm))
        return new RSAEncryption(key);
    throw new NoSuchAlgorithmException(algorithm);
}
```

# Aprovechar características del lenguaje de implementación

## ○ Smalltalk

- Un método que devuelve la clase del objeto, y al que luego puede aplicársele el operador new:

```
clientMethod
    document := self documentClass new.
```

```
documentClass
    self subclassResponsibility
```

- Luego, en MyApplication, tendríamos:

```
documentClass
    ^ MyDocument
```

# Aprovechar características del lenguaje de implementación

## ○ Java

- Podemos pasarle el nombre de la clase al método de fabricación y crear ésta mediante reflectividad
  - ▶ (Similar a lo que acabamos de hacer en Smalltalk)
- Más cómodo aún:
  - ▶ Guardar la clase en una variable de clase (estática) de Application
  - ▶ De esa manera no hay que heredar de ella sólo para cambiar el tipo de producto

# Aprovechar características del lenguaje de implementación

## ○ Inicialización perezosa

```
class Creator {  
public:  
    Product* GetProduct();  
protected:  
    virtual Product* CreateProduct();  
private:  
    Product* _product;  
};  
  
Product* Creator::GetProduct ()  
{  
    if (_product == 0)  
        _product = CreateProduct();  
  
    return _product;  
}
```

# Uso de plantillas

- Podemos usar plantillas (genericidad) para evitar tener que heredar del creador

# Convenios de nombrado

- Se debería usar un estilo de nombres que haga evidente dónde se están utilizando métodos de fabricación:
  - Por ejemplo: Create-

# Ejemplos de nombrado en la API de Java

- **valueOf**
- **of**
  - Ejemplo: `EnumSet.of(...)`
- **getInstance**
- **newInstance**
- **get*Type***
- **new*Type***

(De métodos de creación en general, como los que vimos en la primera parte de estas diapositivas; no necesariamente implementaciones del patrón Factory Method)

# Código de ejemplo

Veamos un ejemplo de aplicación del patrón Factory Method en el proyecto fin de carrera Patrones de diseño aplicados a la construcción de procesadores de lenguaje (César Acebal, E.P.S. de Ingenieros de Gijón, 2002).

# ¿Cómo crear objetos de clases predefinidas en un procesador de lenguaje?

- Lo normal es que para esas clases no generemos código

Instrucción de creación de un objeto  
en el lenguaje fuente

```
new <nombre de la clase>
```

# ¿Cómo crear objetos de clases predefinidas en un procesador de lenguaje?

- Sin el patrón, tendríamos algo como lo siguiente:

```
theClass = classTable.get(className);
if (theClass.isPredefined())
{
    if (theClass instanceof ConsoleClass)
        instance = new ConsoleInstance(theClass);
    if (theClass instanceof StringClass)
        instance = new StringInstance(theClass);
    ...
}
else
{
    instance = new UserClass(theClass);
}
```

## 6.2 Métodos de clases predefinidas

Uno de los requisitos de SmallScript es, como ya se ha dicho, que no se genere código para las clases predefinidas, sino que éstas estén implementadas de manera nativa. Es una restricción lógica, con la que habrá de lidiar cualquier procesador de lenguaje (no tiene ningún sentido perder el tiempo en generar siempre el mismo código objeto, amén de la importancia que tiene la eficiencia para el éxito de una biblioteca de clases).

Ahora bien, el requisito anterior se une a otro principio lógico de diseño, según el cual, en el código, debería tratarse de forma uniforme a los métodos de las clases predefinidas y a los definidos por el usuario. Es decir, habría que huir del típico código como el siguiente:

```
void call(String metodo)
{
    a ← pop           // en la cima de la pila está la referencia
                      // si a es null, dar un error
    clase ← a.obj    // todo objeto apunta a su tipo
    si la clase es predefinida
    {
        si es Console.print
            System.out.println(...)
        ...
    }
    else
    {
        m ← buscar(clase, metodo) // si no existe, error
        guardar(CP)
        CP ← {clase, m, 0}
    }
}
```

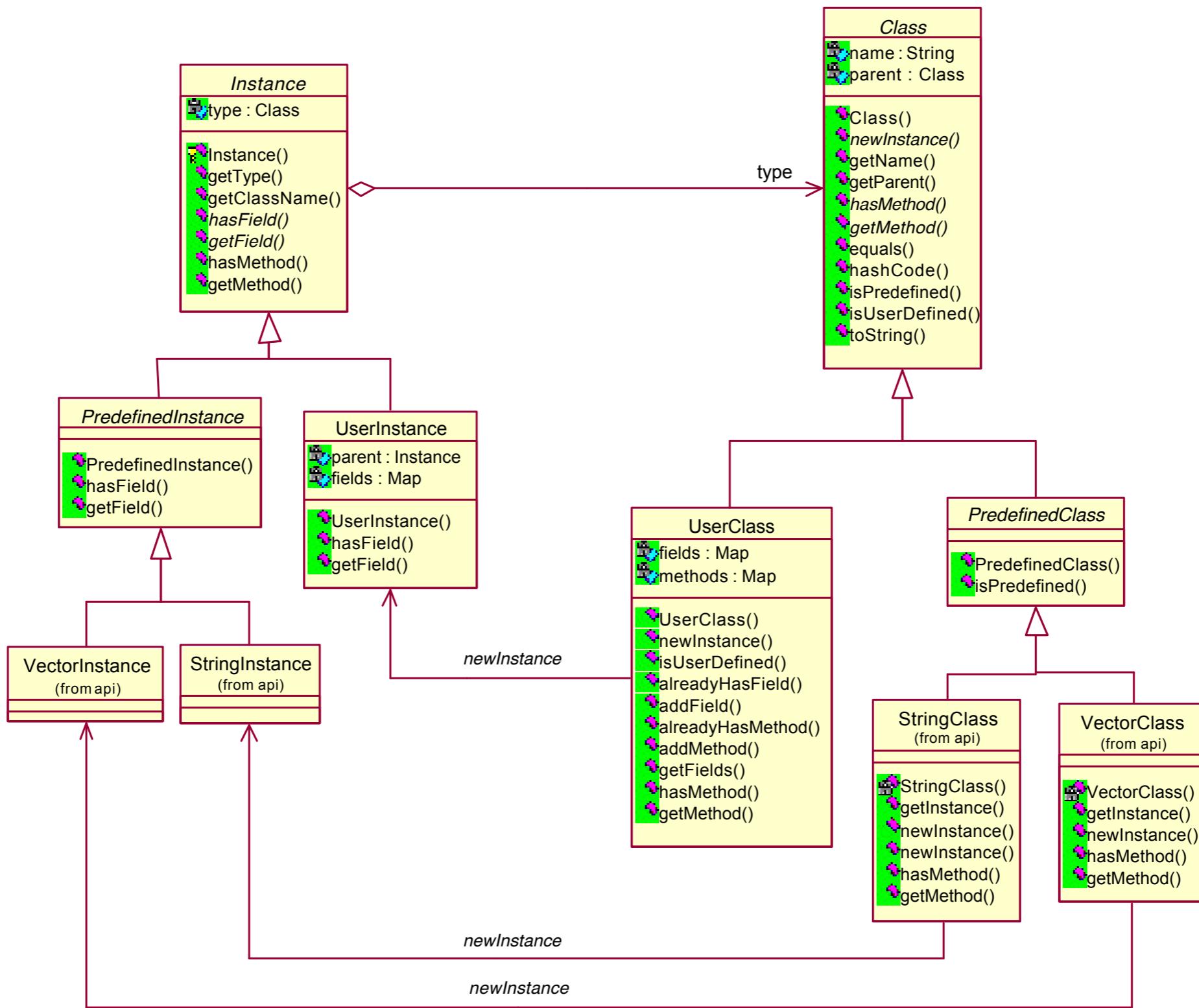
**Figura 6-6: Pseudocódigo de invocación de métodos**

El listado de la figura anterior pretende mostrar el pseudocódigo de cómo podría ser una primera aproximación a la invocación de métodos. En él se ha resaltado el código donde se diferencia entre si se trata de un método predefinido, en cuyo caso se implementa de forma nativa, o bien si es un método definido por el usuario, y entonces se busca el método en la tabla de símbolos y se modifica el contador de programa para que apunte a la primera instrucción en código objeto de dicho método.

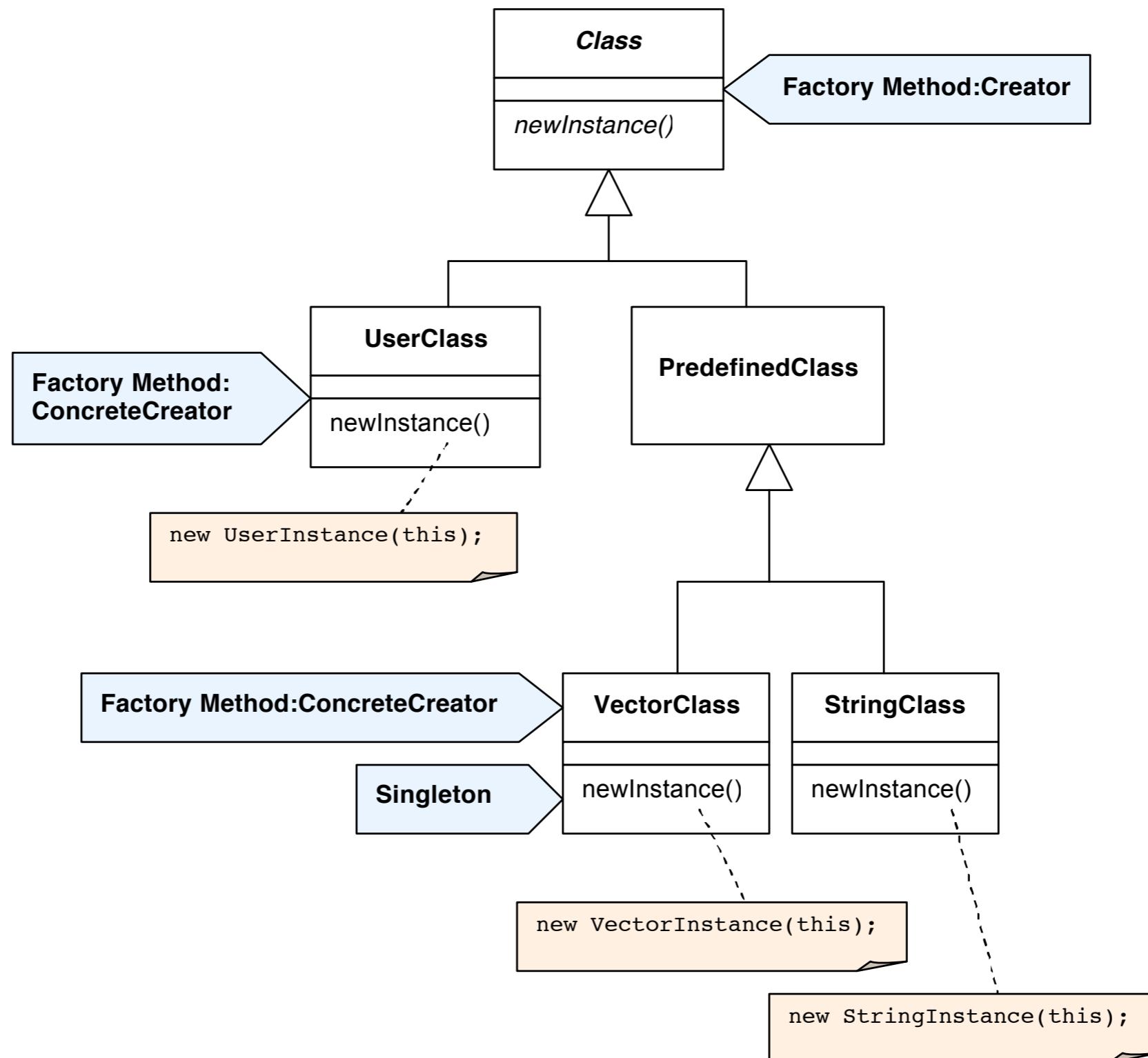
Naturalmente, ese tipo de código es del que debemos huir en un buen diseño orientado a

Problema...

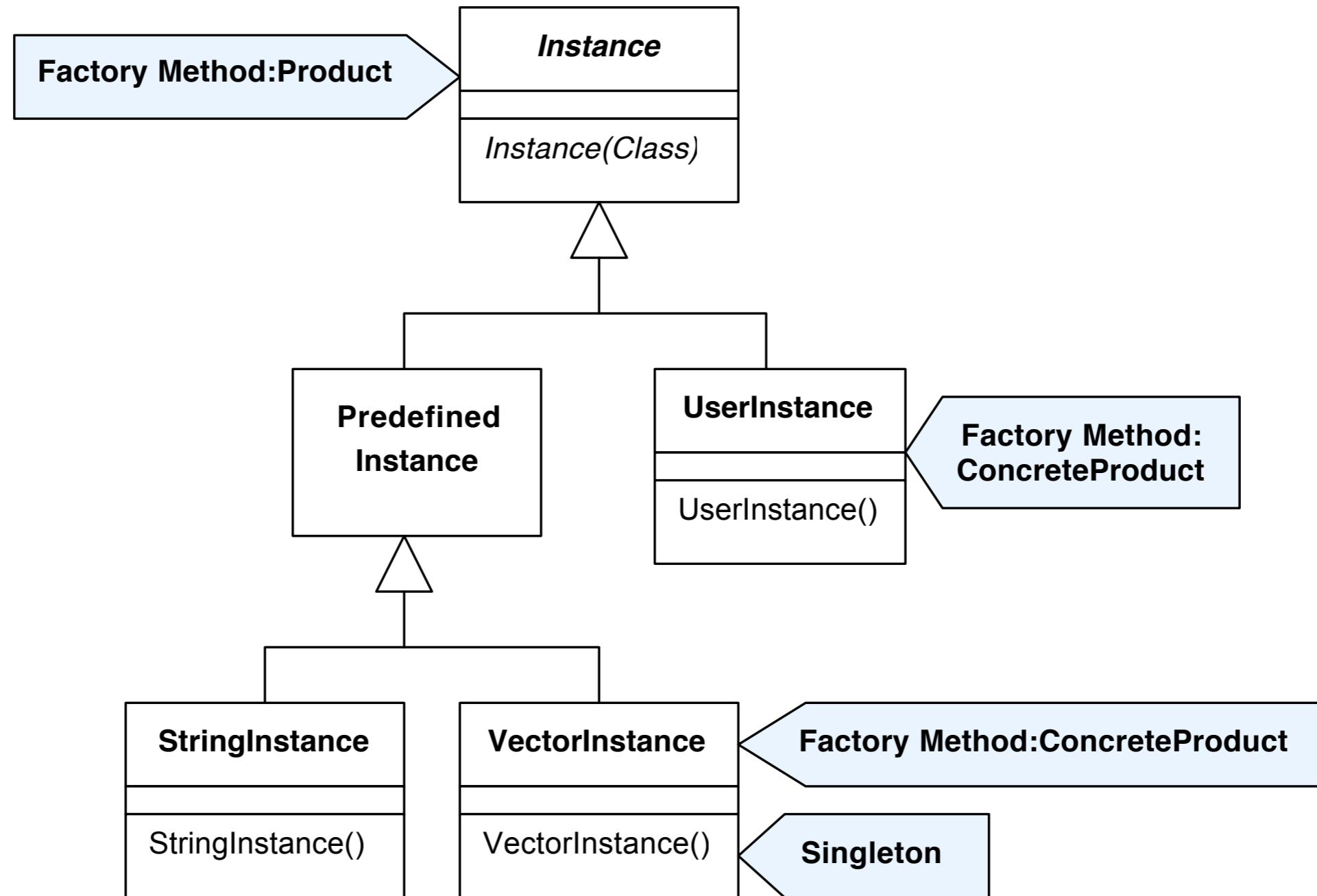
*¿Qué ocurre ante cualquier cambio en la biblioteca de clases del lenguaje?*



**Figura 4-2:** Creación de instancias (patrón Factory Method)



**Figura 4-3:** Patrón Factory Method aplicado a la creación de instancias (Creadores)



**Figura 4-4:** Patrón Factory Method aplicado a la creación de instancias (Productos)

# Resultado de aplicar el patrón

- Tras aplicar el patrón, el código anterior queda reducido a:

```
theClass.newInstance();
```

- (Sea cual sea la clase del objeto a crear)

# **Abstract Factory**

# Abstract Factory (Fábrica abstracta)

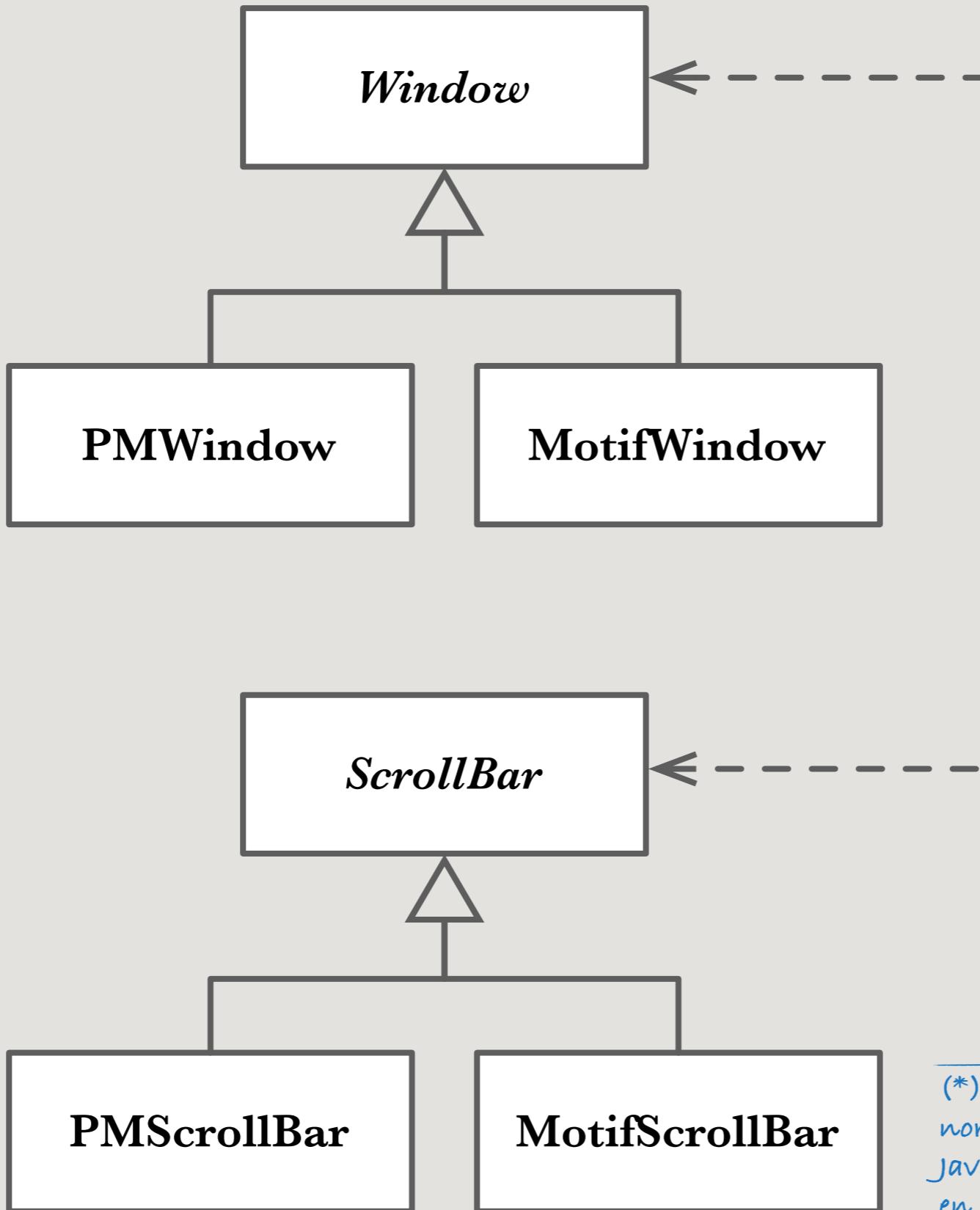
- Patrón de creación (ámbito de objetos)
- Propósito:

*Define una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.*

# Motivación

- Sea una biblioteca gráfica que permita generar interfaces para diferentes entornos de ventanas (p. ej., Motif y Presentation Manager)
  - Cada uno de ellos tendrá una clase distinta para representar una ventana, una barra de desplazamiento, un botón...
    - ▶ `PMWindow`, `MotifWindow`, `PMScrollBar`, `MotifScrollBar`...
  - Si queremos que una aplicación se aproveche de ello y sea portable, no podrá crear directamente objetos de esas clases específicas

# Queremos esto:



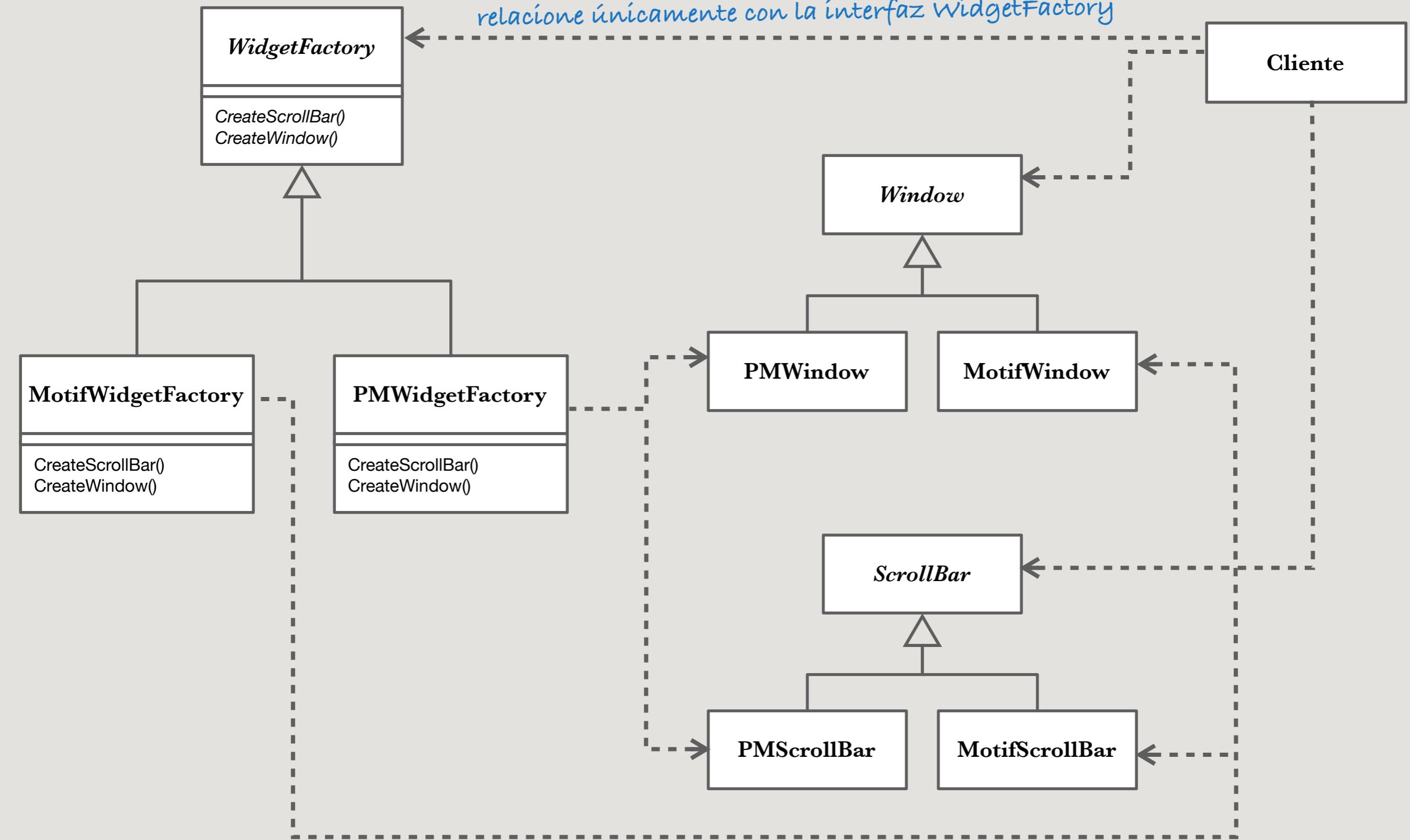
Es decir, que el cliente se relacione únicamente con las interfaces<sup>(\*)</sup> Window, ScrollBar, etcétera, y no con sus implementaciones concretas.

Pero en algún momento habrá que crear esos objetos concretos (hacer new <LoQueSea>...)

## ¿Entonces?

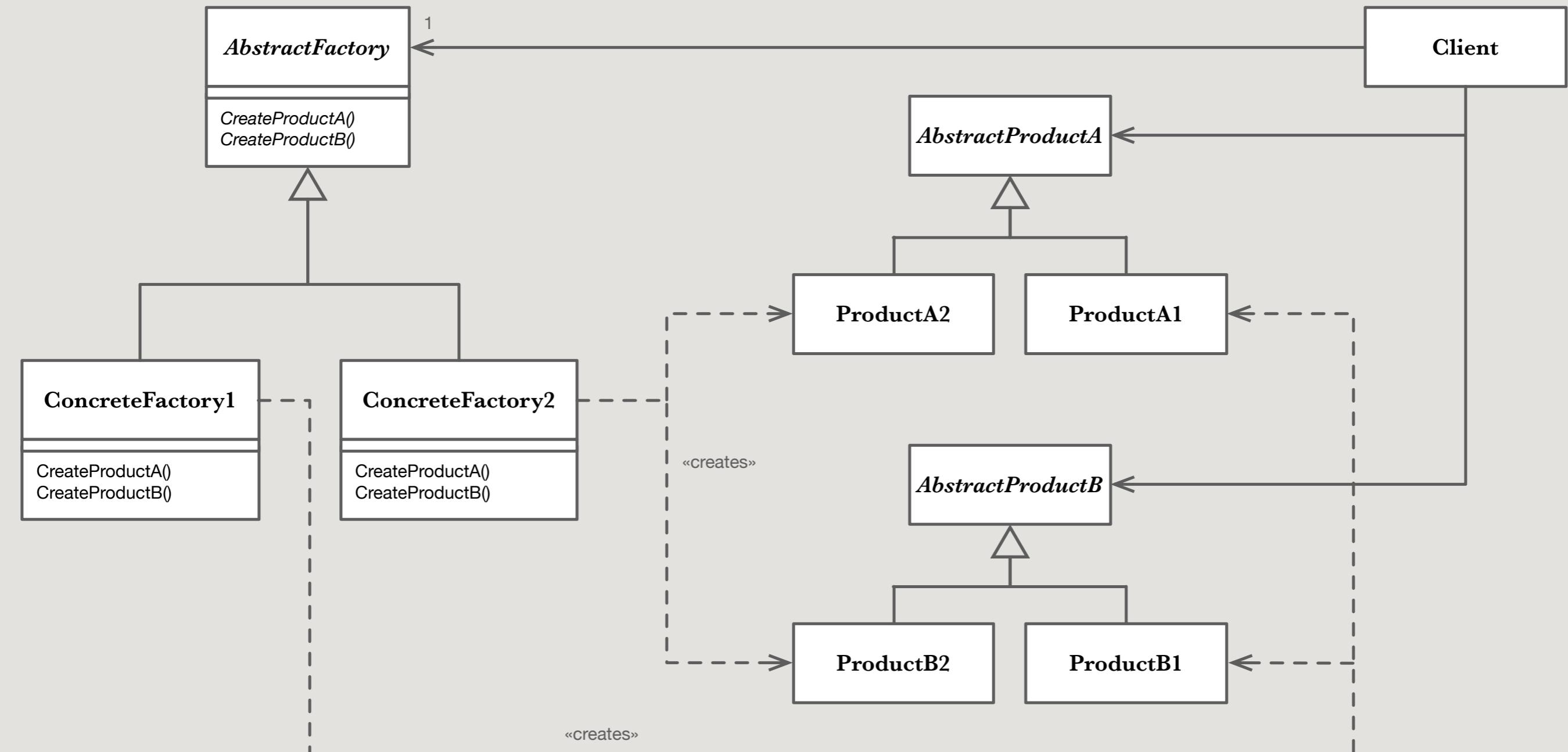
(\*) Recordad: interfaces, clases abstractas, clases normales... No lo asociéis con las interfaces de Java, sino con el concepto más amplio de interfaz en OO

El cliente ni siquiera tiene por qué conocer a las fábricas concretas, sino que lo normal es que se relacione únicamente con la interfaz WidgetFactory



- Nótese cómo, por el mero hecho de utilizar el patrón, se cumple la restricción de que una barra de desplazamiento de Motif sólo pueda usarse con una ventana de Motif, y así sucesivamente
  - Recordemos: familias de objetos relacionados

# Estructura



La estructura del patrón *Abstract Factory*

# Consecuencias

- Aísla las clases concretas
  - Los clientes manipulan los productos únicamente a través de sus interfaces abstractas, gracias a que las clases de productos concretos están encapsuladas en cada fábrica concreta, no aparecen en el código
- Permite intercambiar fácilmente familias de productos
  - Basta con cambiar una única clase, en un único sitio: la fábrica concreta

# Consecuencias

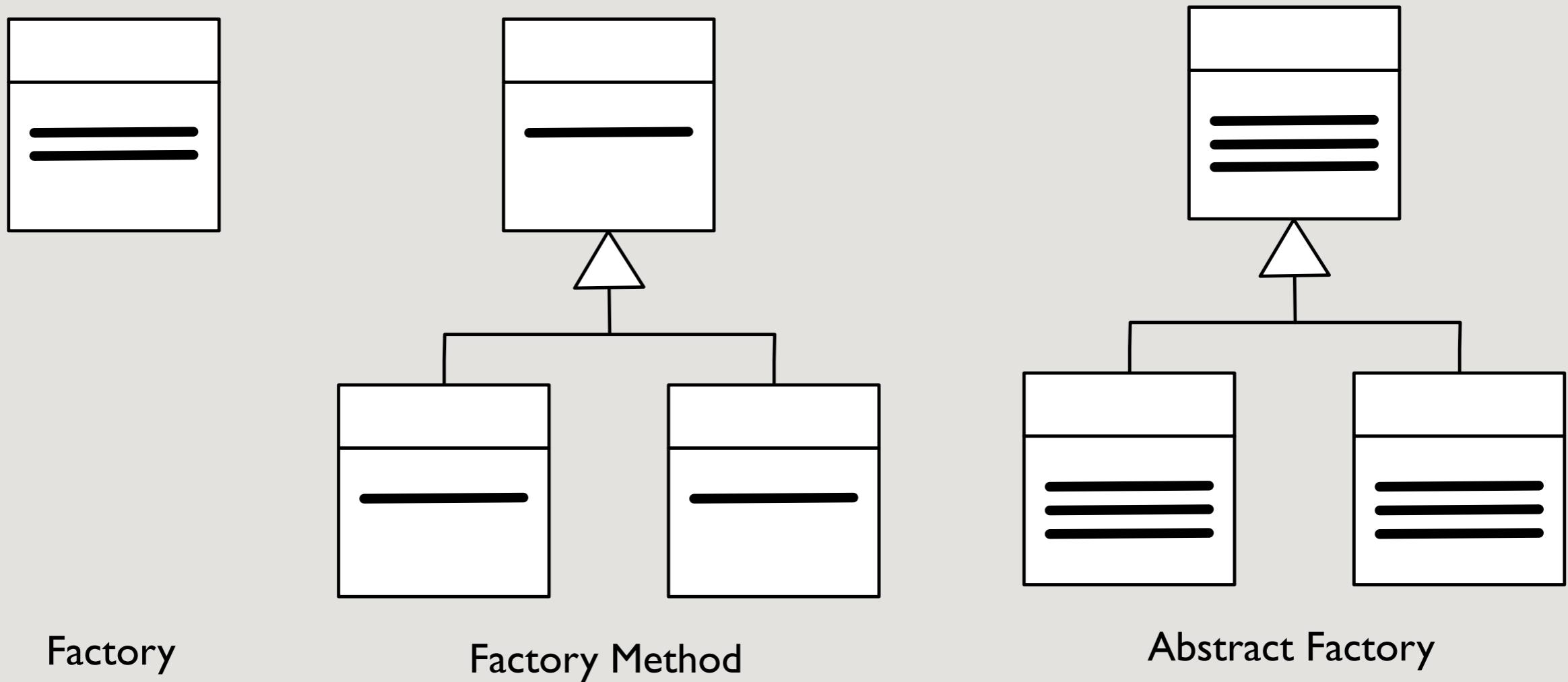
- Promueve la consistencia entre los productos
  - Sólo se pueden usar conjuntamente los objetos de cada familia
- Dificulta añadir nuevos tipos de productos
  - Hay que cambiar la interfaz de la fábrica abstracta y por tanto implementar el nuevo método en todas sus subclases

# Implementación

- Las fábricas suelen ser *Singlets*
- Crear los productos
  - Normalmente, el *Abstract Factory* emplea a su vez un *Factory Method* para cada producto
    - ▶ Es el enfoque más sencillo, con el inconveniente de que requiere crear una subclase por cada familia

# Diferencias

# Diferencias



Factory

Factory Method

Abstract Factory

Las líneas en negrita representan métodos que crean objetos. Esta figura, debida a Kerievsky (2005), ilustra de ese modo, muy esquemáticamente, las diferencias más significativas entre una simple clase de creación y los patrones de diseño *Factory Method* y *Abstract Factory*.

<http://c2.com/cgi/wiki?AbstractFactoryVsFactoryMethod>