

7

Factorías, Factory Method y Abstract Factory

Diseño del Software

Grado en Ingeniería Informática del Software

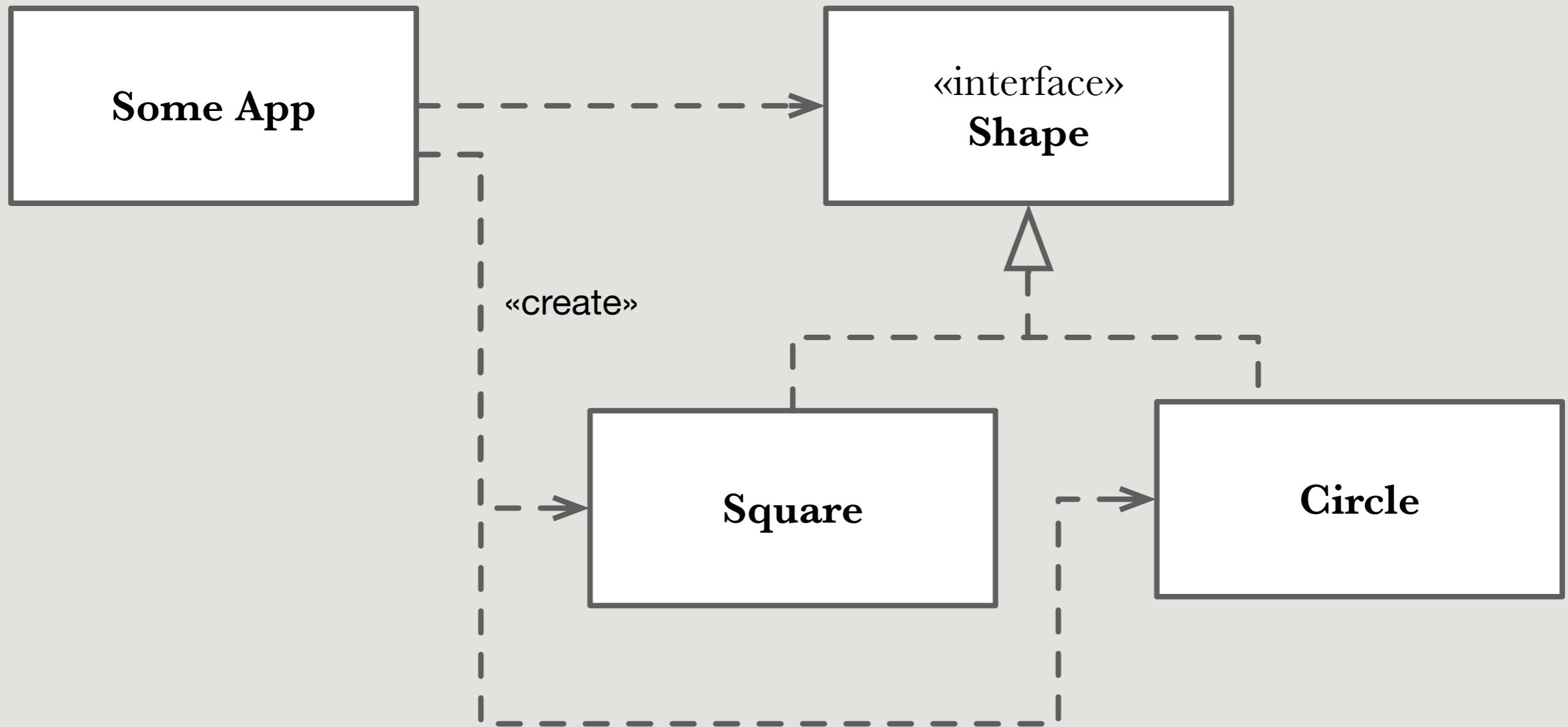
Curso 2024-2025

Introducción

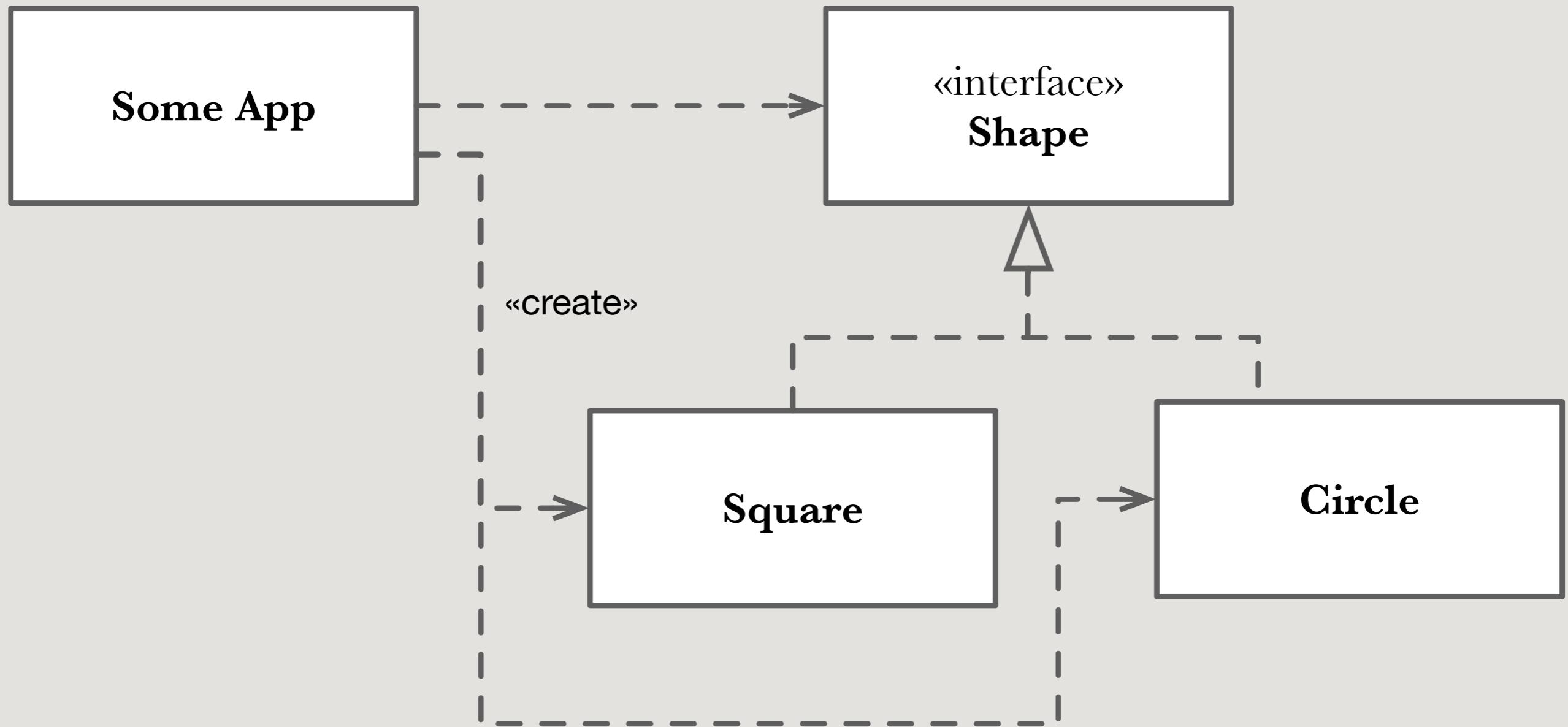
Factorías

*Principio de inversión
de dependencias (DIP)*

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



¿Qué observamos?



Viola el principio de inversión de dependencias

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.

El problema con
“new”

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

```
Shape shape = new Circle(origin, 150);
```

Queremos usar interfaces
para mantener el código
flexible.

pero tenemos que crear
un objeto de una clase
concreta!

¡Acoplamiento!

```
Shape shape = new Circle(origin, 150);
```

Queremos usar interfaces
para mantener el código
flexible.

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

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



```
Shape shape;  
if (type.equals("rectangle")) {  
    shape = new Rectangle(...);  
} else if (type.equals("circle")) {  
    shape = new Circle(...);  
} else if (type.equals("triangle")) {  
    shape = new Triangle(...);  
} else {  
    throw new IllegalArgumentException(  
        "Invalid shape type: " + type);  
}
```

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

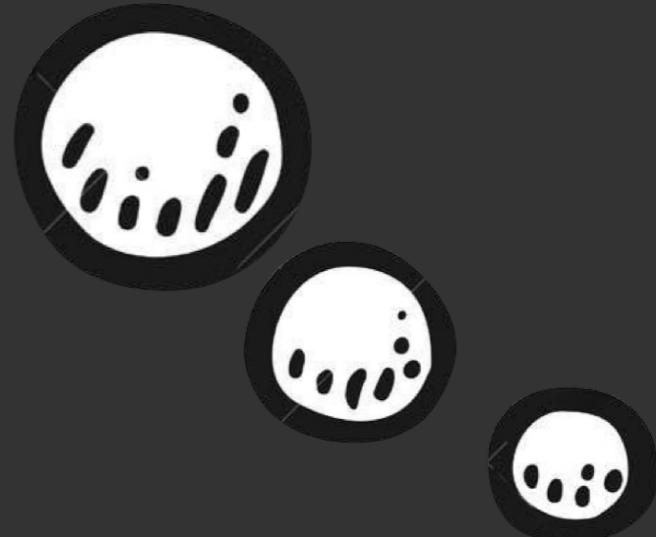
Cada vez que aparecen nuevas clases (o se eliminan) hay que modificar ese código.

El problema no es ya tanto dicho código en sí (que necesariamente deberá aparecer en algún sitio) como el hecho de que muy frecuentemente se repite en varios sitios de la aplicación (violando así el principio de abierto-cerrado).

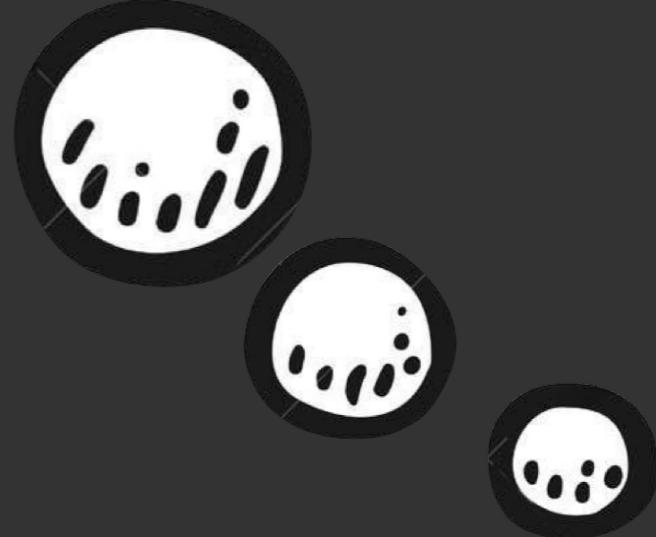


cuando programamos para una interfaz sabemos que nos estamos aislando de un montón de cambios que pueden suceder durante la implementación, ya que funcionará con cualesquiera nuevas clases que implementen la interfaz, a través del polimorfismo. Sin embargo, aquí estamos haciendo uso de clases concretas. 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 HABRÁ QUE CREAR
REALMENTE EL OBJETO Y
JAVA SOLO PROPORCIONA UN
“modo de hacerlo, ¿no?”*

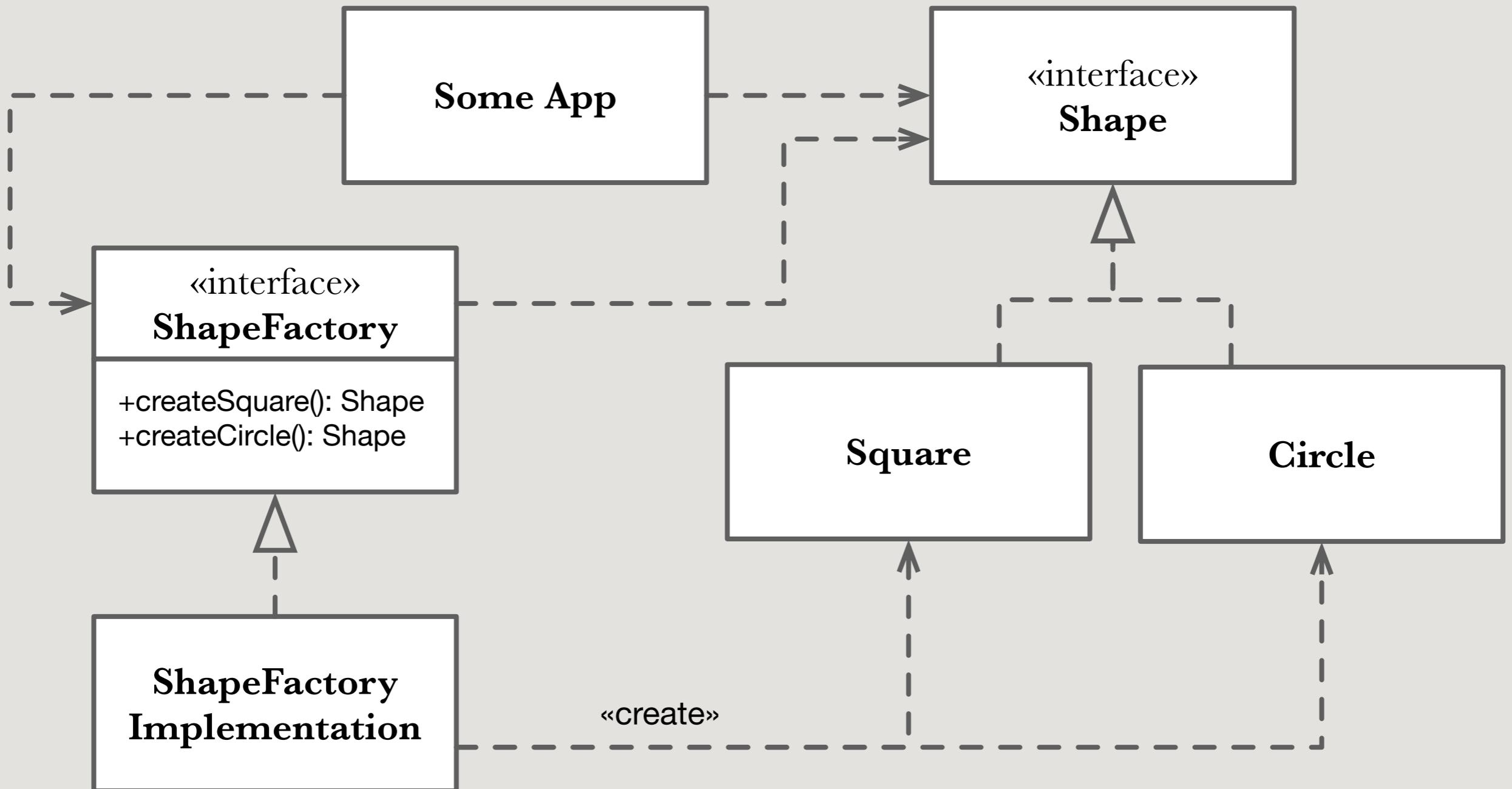


¿ENTONCES?



*Otro de nuestros principios
de cabecera:*

*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.

¿Quiere esto decir que siempre tenemos que crear los objetos de esta manera? ¡Por supuesto que no!

Pero si prevemos que estas clases pueden cambiar y, lo que es más importante, si este código se repite en diferentes partes de la aplicación, entonces las «factorías simples» (no son ningún patrón de diseño como tal) pueden ser a veces la solución. No obstante, hemos de estar seguros antes de introducirlas sin más.

Como siempre, ¡no sobrediseñéis!

Otra variante



```
public Shape createShape(String type) {  
    if (type.equals("rectangle")) {  
        return new Rectangle();  
    } else if (type.equals("circle")) {  
        return new Circle();  
    } else if (type.equals("triangle")) {  
        return new Triangle();  
    } else {  
        throw new IllegalArgumentException(  
            "Invalid shape type: " + type);  
    }  
}
```

Con un método parametrizado

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.

POWELL STREET

PIZZA & PASTA

O'REILLY®

Second
Edition

Head First Design Patterns

Building Extensible
& Maintainable
Object-Oriented
Software

Eric Freeman &
Elisabeth Robson
with Kathy Sierra & Bert Bates



A Brain-Friendly Guide

A través de un ejemplo (es verdad que demasiado artificial) tomado del libro **Head First Design Patterns**, empezaremos creando una factoría simple y terminaremos llegando al patrón de diseño Factory Method.



PizzaStore.java

```
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...)



PizzaStore.java

```
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 O ELI-
MINAMOS TIPOS DE
PIZZA?**



PizzaStore.java

```
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();  
    }  
    // ...  
}
```

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.



PizzaStore.java

```
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.



PizzaStore.java

```
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("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.



PizzaStore.java

```
// ...
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.

Con una factoría simple

Podemos sacar la lógica de la creación del objeto fuera del método `orderPizza`.



SimplePizzaFactory.java

```
public class SimplePizzaFactory {  
    // ...  
    public Pizza createPizza(String type) {  
        Pizza pizza;  
        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();  
        } else // ...  
        return pizza;  
    }  
}
```



¡¿Hemos ganado 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 solo haya que cambiar un sitio cuando las clases concretas cambien.

Otra ventaja (la principal, siempre y cuando preveamos que las implementaciones concretas pueden variar) es que los clientes no conocen las pizzas concretas, sino únicamente el tipo base Pizza, como veíamos en el primer ejemplo de las figuras.

¿Y no se podría haber hecho con un método estático?

En principio... sí. Tendría 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 que suele ser la principal, y yo diría que casi única motivación, para introducir una factoría «a secas».

¿Y cómo queda la clase PizzaStore?

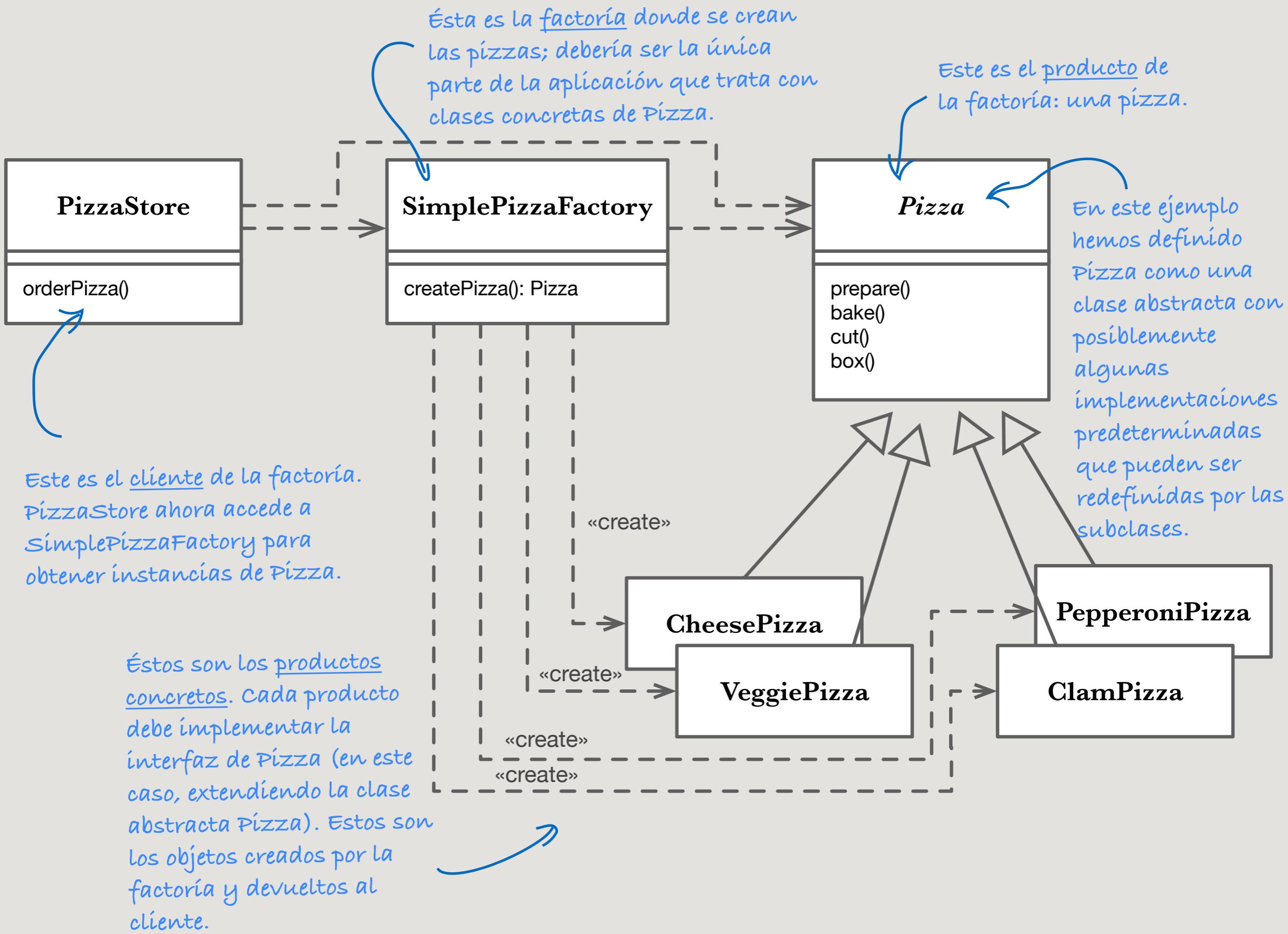


PizzaStore.java

```
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?



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

A este tipo de clases las llamaremos **factorías simples** (o factorías «a secas»).

Se abusa mucho del término «factoría» y cada uno se lo llama a cosas distintas.

Creamos una
franquicia

Nuestro negocio ha tenido tanto éxito que hemos decidido expandirnos y abrir franquicias en muchos lugares.

Queremos preservar la calidad y el conocimiento que nos han hecho famosos, mientras adaptamos nuestras pizzas a los gustos e ingredientes de cada región.

Queremos que todas las franquicias comparten un modo común de hacer las pizzas.



La franquicia de Nueva York quiere una factoría que cree pizzas al estilo neoyorquino: masa fina, salsa sabrosa y solo un poco de queso.



Otra franquicia necesita una factoría que haga pizzas al estilo de Chicago: sus clientes prefieren las pizzas con una masa gruesa y montones de queso.

Una posibilidad sería simplemente sacar fuera **SimplePizzaFactory** y creamos tres factorías concretas, **NYPizzaFactory**, **ChicagoPizzaFactory** y **CaliforniaPizzaFactory**, podemos configurar (mediante composición) **PizzaStore** con la fábrica concreta.

Veamos cómo podría ser...



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

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

Una vez más, como ocurrió con el ejemplo de los patos en el patrón Strategy, tened cuidado con la explicación dada en el libro Head First Design Patterns sobre la solución anterior

Según sus autores, el problema con el enfoque de factorías simple es que las franquicias podrían empezar a emplear sus propios procedimientos caseros para el resto del proceso.

Por ejemplo, hornear las pizzas de forma ligeramente distinta, olvidarse de cortarlas o incluso usar cajas de terceros.

Los autores sugieren que se necesita un mayor control de calidad.

Según sus autores, el problema con el enfoque de factorías simple es que las franquicias podrían empezar a emplear sus propios procedimientos caseros para el resto del proceso.

Por ejemplo, hornear las pizzas de forma ligeramente distinta, olvidarse de cortarlas o incluso usar cajas de terceros.

Los autores sugieren que se necesita un mayor control de calidad.

Eso no es cierto.

Según sus autores, el problema con el enfoque de factorías simple es que las franquicias podrían empezar a emplear sus propios procedimientos caseros para el resto del proceso.

Por ejemplo, hornear las pizzas de forma ligeramente distinta, olvidarse de cortarlas o incluso usar cajas de terceros.

Los autores sugieren que se necesita un mayor control de calidad.

Eso no es cierto.

De hecho, no tiene ningún sentido.

En la solución anterior, la clase **PizzaStore** permanece consistente en todas las franquicias.

Simplemente crean su propia implementación de **SimplePizzaFactory** y configuran la **PizzaStore** con ella en el momento de la creación.

No hay forma de que puedan cambiar cómo se preparan, hornean, cortan o empaquetan las pizzas.

El único problema potencial de este enfoque podría ser, como mucho, la flexibilidad que permite.

Así, sería posible, por ejemplo, pasarle a la pizzería de Nueva York la factoría de Chicago.

En este caso, no solo no necesitamos dicha flexibilidad, sino que podría ser contraproducente.

**Necesitamos una manera de volver
a tener el código de creación de la
pizza en la propia PizzaStore, pero
siendo a la vez flexible.**

Un framework de pizzerías



PizzaStore.java

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

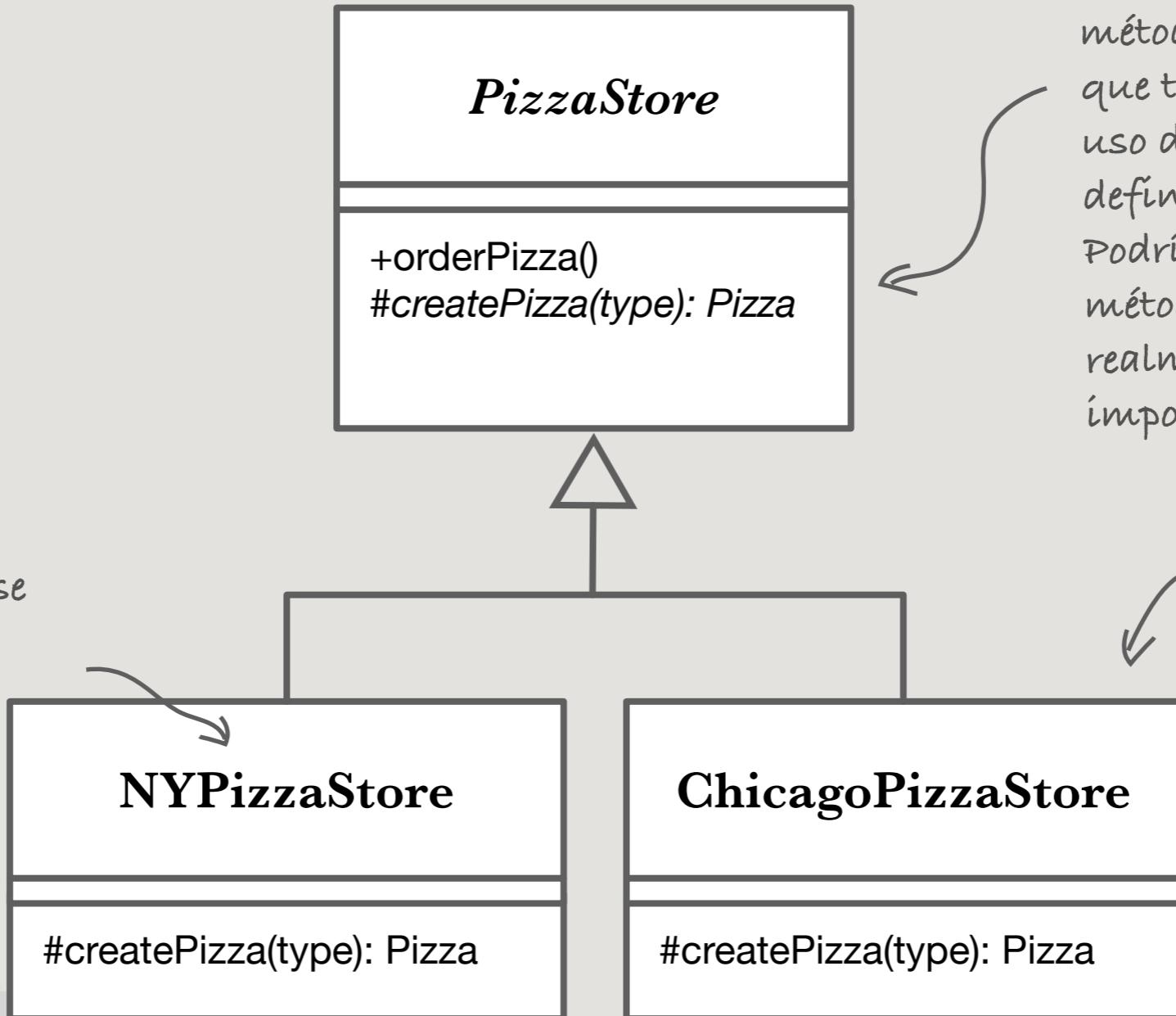
Todo esto no cambia...

La factoría simple se ha cambiado por este método.

El 'factory method' es abstracto y protegido.

cada clase sobrescribe el método `createPizza()`, mientras que todas las subclases hacen uso del método `orderPizza()` definido en `PizzaStore`. Podríamos hacer final el método `orderPizza()` si realmente quisieramos imponerlo.

Si una franquicia quiere pizzas estilo NY para sus clientes, utiliza la subclase `NY`, que tiene su propio método `createPizza()`, creando pizzas estilo NY.



```
if (type.equals("cheese")) {
    return new NYStyleCheesePizza();
} else if (type.equals("pepperoni")) {
    return new NYStylePepperoniPizza();
} else if ...
```

```
if (type.equals("cheese")) {
    return new ChicagoStyleCheesePizza();
} else if (type.equals("pepperoni")) {
    return new ChicagoStylePepperoniPizza();
} else if ...
```

Del mismo modo, utilizando la subclase `Chicago`, obtenemos una implementación de `createPizza()` con ingredientes Chicago.

Pizza

NYStyleCheesePizza

NYStylePepperoniPizza

NYStyleClamPizza

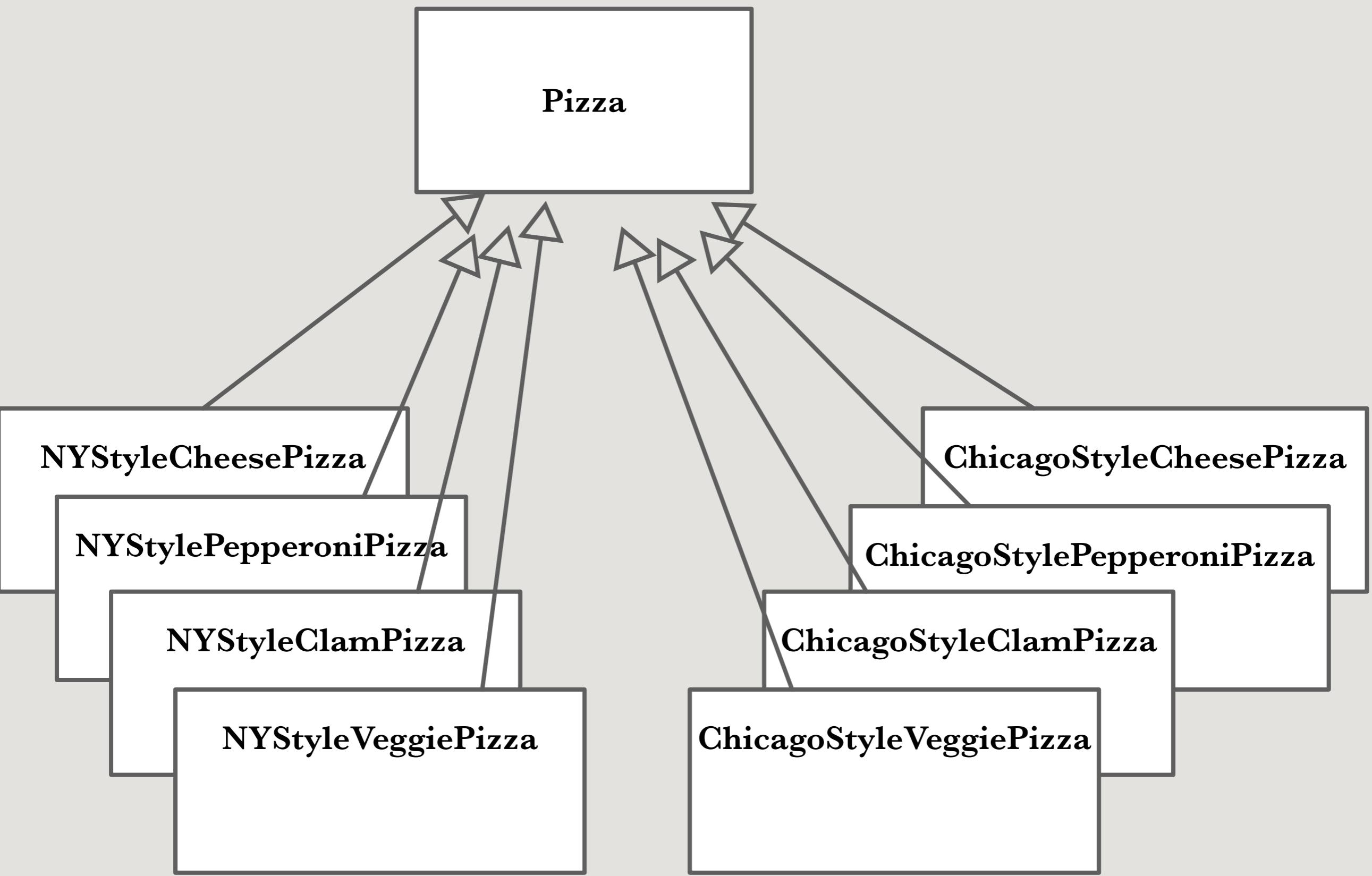
NYStyleVeggiePizza

ChicagoStyleCheesePizza

ChicagoStylePepperoniPizza

ChicagoStyleClamPizza

ChicagoStyleVeggiePizza



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!

Hemos llegado así a una implementación del patrón
Factory Method.

protected abstract Product factoryMethod(type)

Un «factory method» es abstracto: las subclases concretas deben redefinirlo por fuerza..
Declararlo como protegido refuerza esta intención y permite que las subclases residan en paquetes diferentes.

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). Pero no es ésta la esencia del patrón, sino que se delegue la creación a las subclases (de hecho, es más frecuente que haya un solo tipo de producto por cada fábrica).



Factory Method

Patrón de creación, de clases

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

También conocido

como

Virtual Constructor

Constructor «polimórfico»

Motivación

**Sea un framework para la creación
de editores de distintos tipos de
documentos.**

Las dos abstracciones fundamentales son **Application** y **Document**.

Ambas son abstracciones, y los usuarios deben proporcionar implementaciones específicas para cada aplicación.

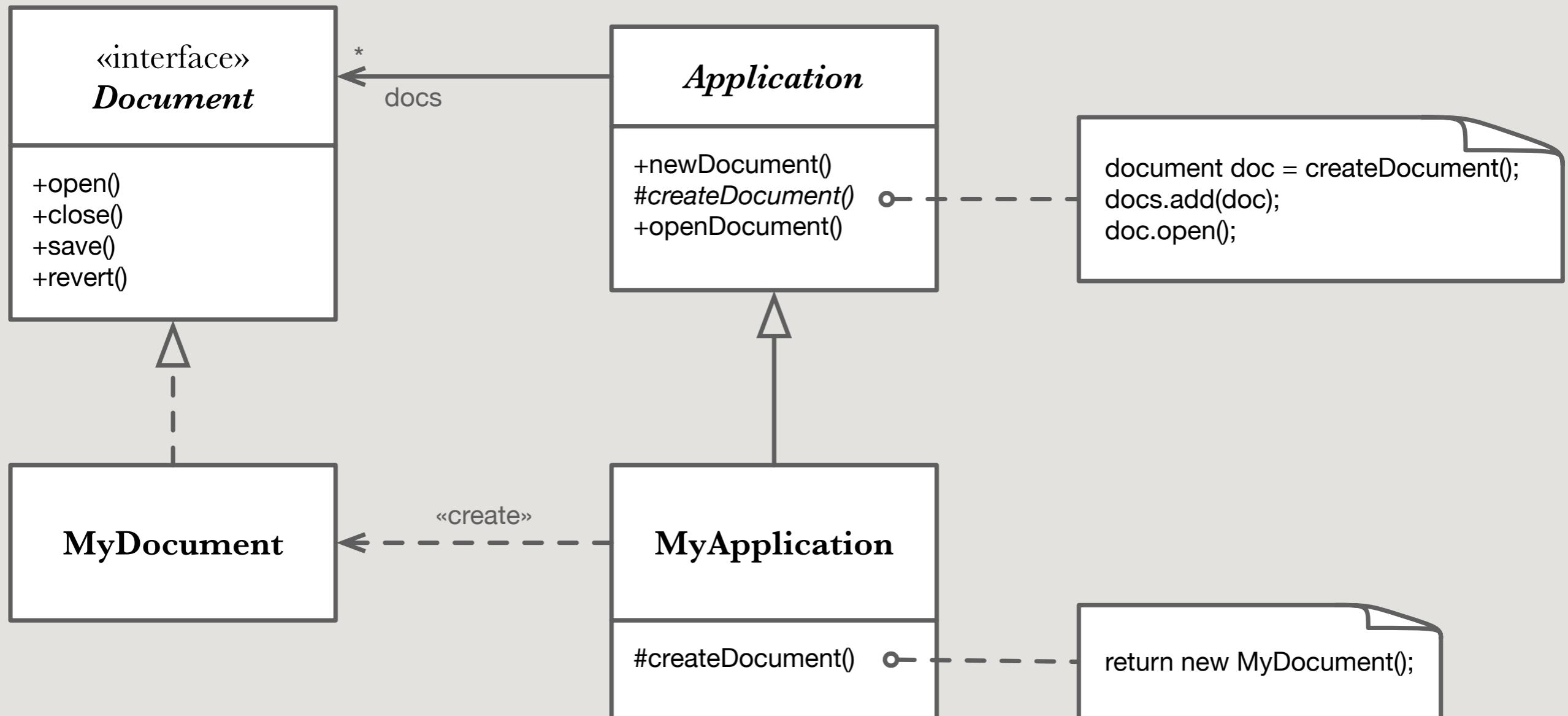
Para crear una aplicación de dibujo, por ejemplo, definimos las clases **DrawingApplication** y **DrawingDocument**. La aplicación es responsable de gestionar los documentos y los creará cuando sea necesario (por ejemplo, cuando el usuario seleccione abrir o nuevo en un menú).

Dado que la clase concreta de documento depende de la aplicación, esta no puede predecir cuál instanciar.

Solo sabe cuándo debe crearse un nuevo documento, no de qué tipo.

Esto supone un dilema:

El framework debe instanciar
clases, pero solo conoce las
abstracciones.



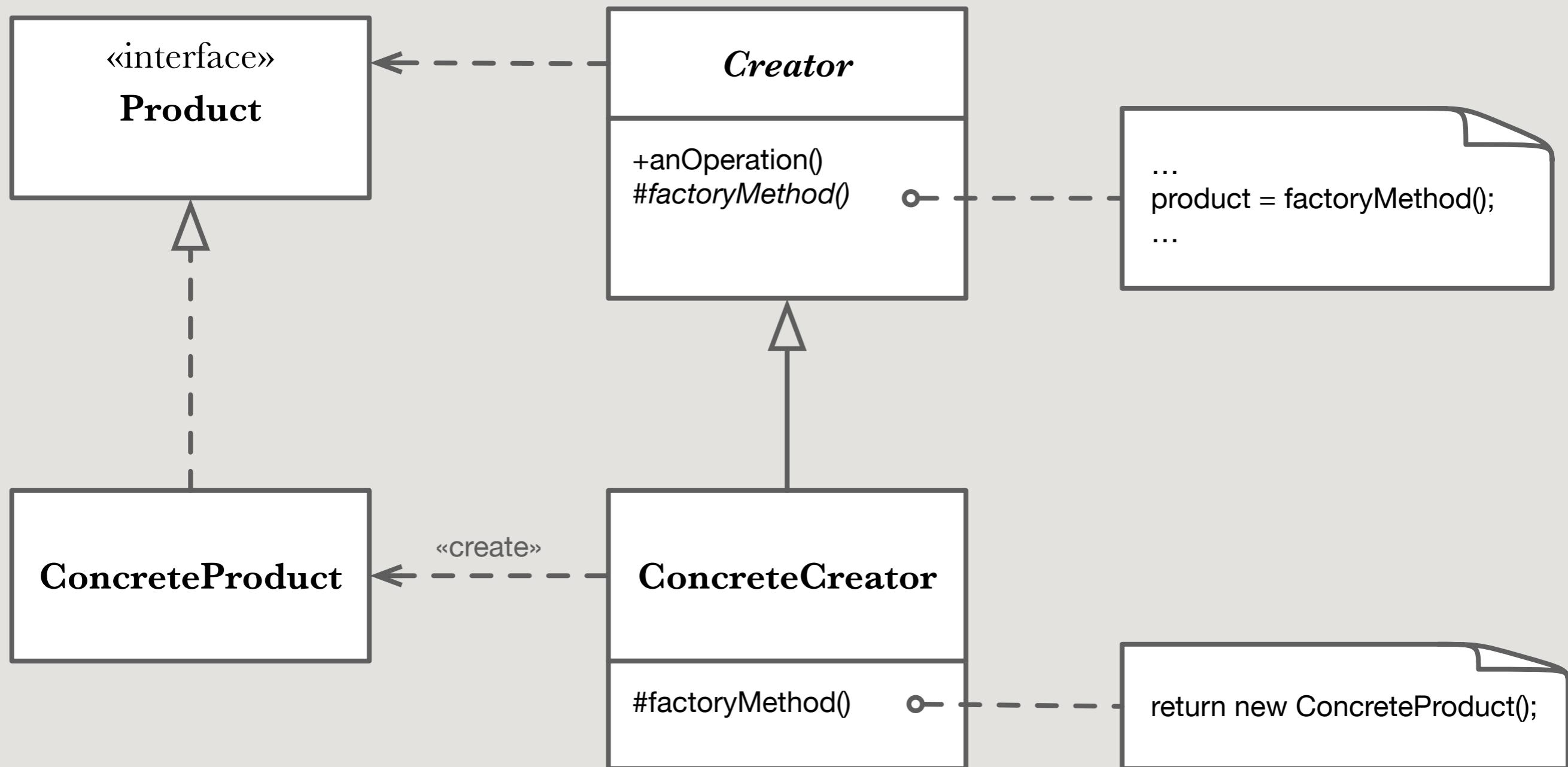
Aplicabilidad

Úsese el patrón Factory
Method cuando

**Una clase no puede anticipar la
clase de objetos que debe crear.**

**Una clase quiere que sean sus
subclases quienes especifiquen
los objetos a crear.**

Estructura



Participantes

Product

(Document)

**Define la interfaz de los objetos creados por el
método de fabricación.**

ConcreteProduct

(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.

Un posible inconveniente

Tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

Funciona mejor cuando dicha herencia de creadores ya existe, con independencia del patrón.

Proporciona «ganchos» para las subclases.

De manera que puedan proporcionar otra versión de un objeto.

En el ejemplo Document, la clase Document podría definir un método de fábrica llamado CreateFileDialog que crea un objeto de diálogo de archivo por defecto para abrir un documento existente.

Una subclase Document puede definir un diálogo de archivo específico de la aplicación sobrescribiendo este método de fábrica.

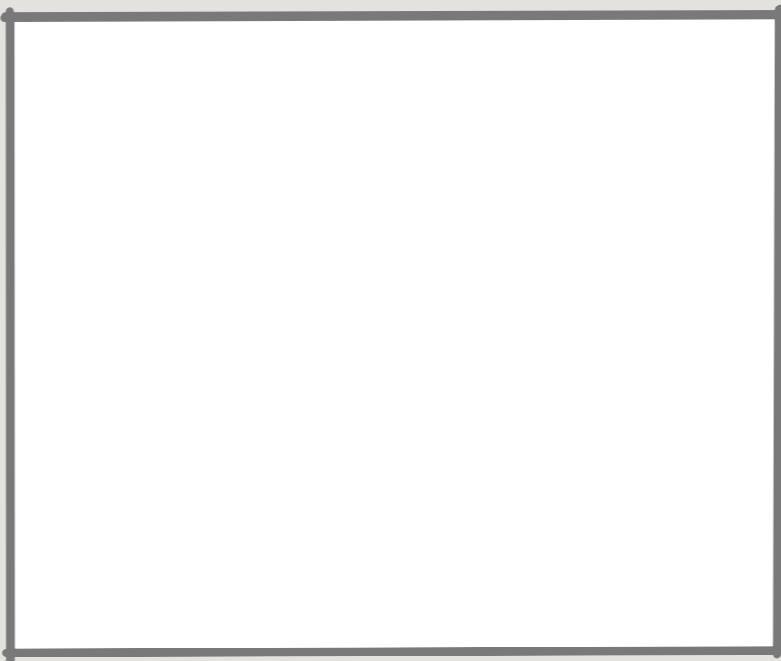
En este caso el método de fábrica no es abstracto, sino que proporciona una implementación predeterminada razonable.

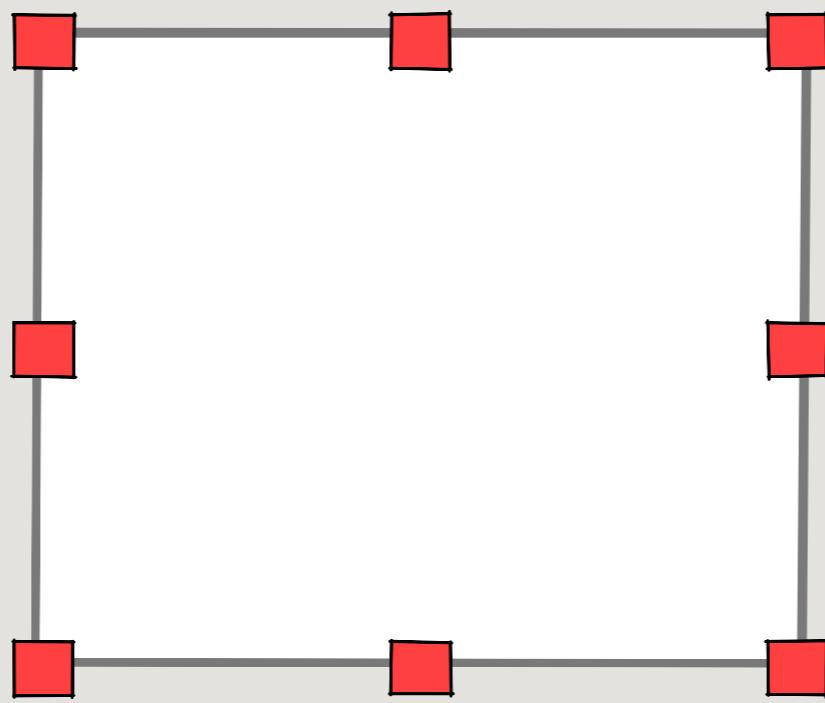
Jerarquías de clases paralelas

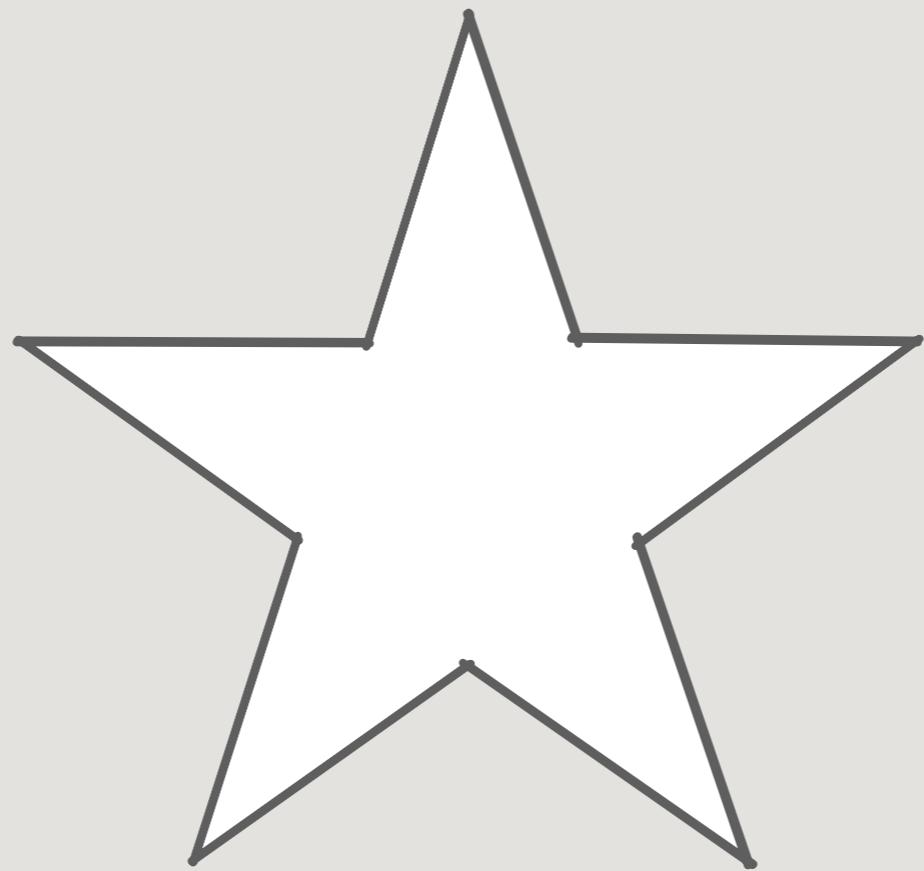
En los ejemplos que hemos considerado hasta ahora, el método de creación solo es llamado por los creadores, pero esto no tiene por qué ser siempre así.

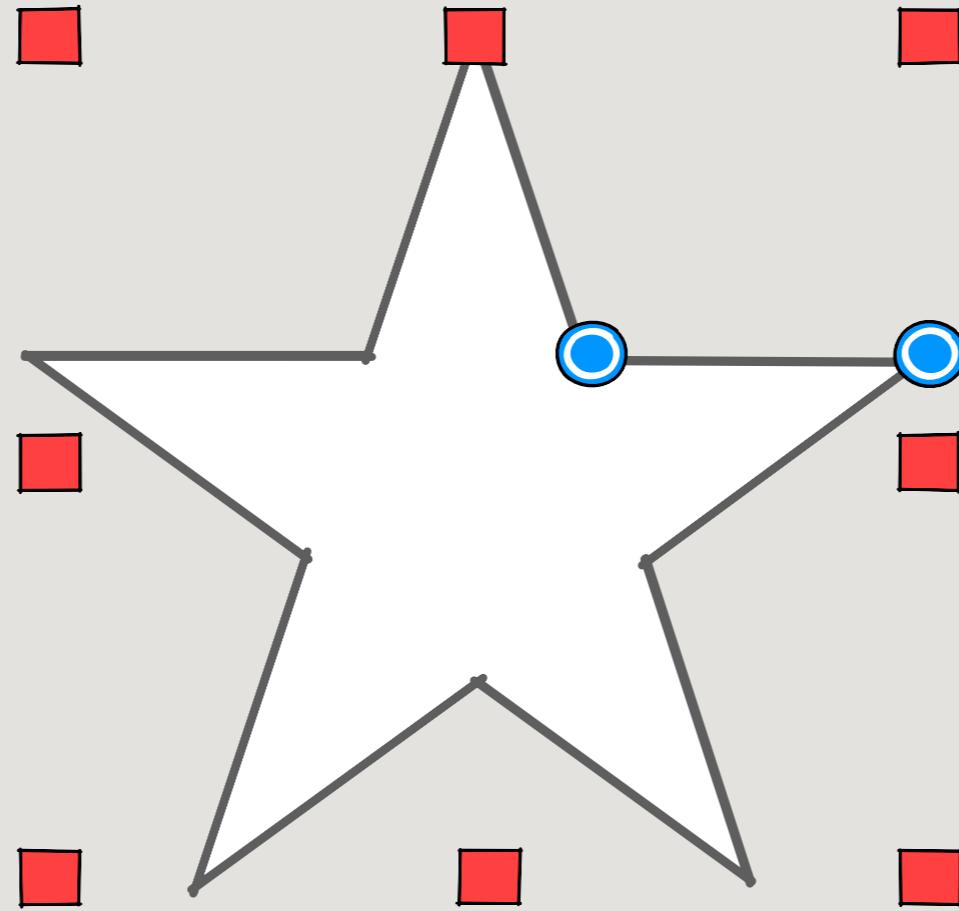
Las jerarquías de clases paralelas resultan cuando una clase delega algunas de sus responsabilidades en otra clase.

Ejemplo: manipuladores de figuras.

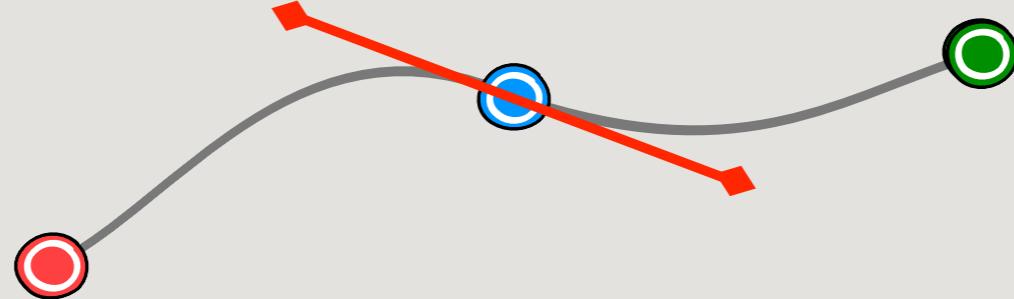


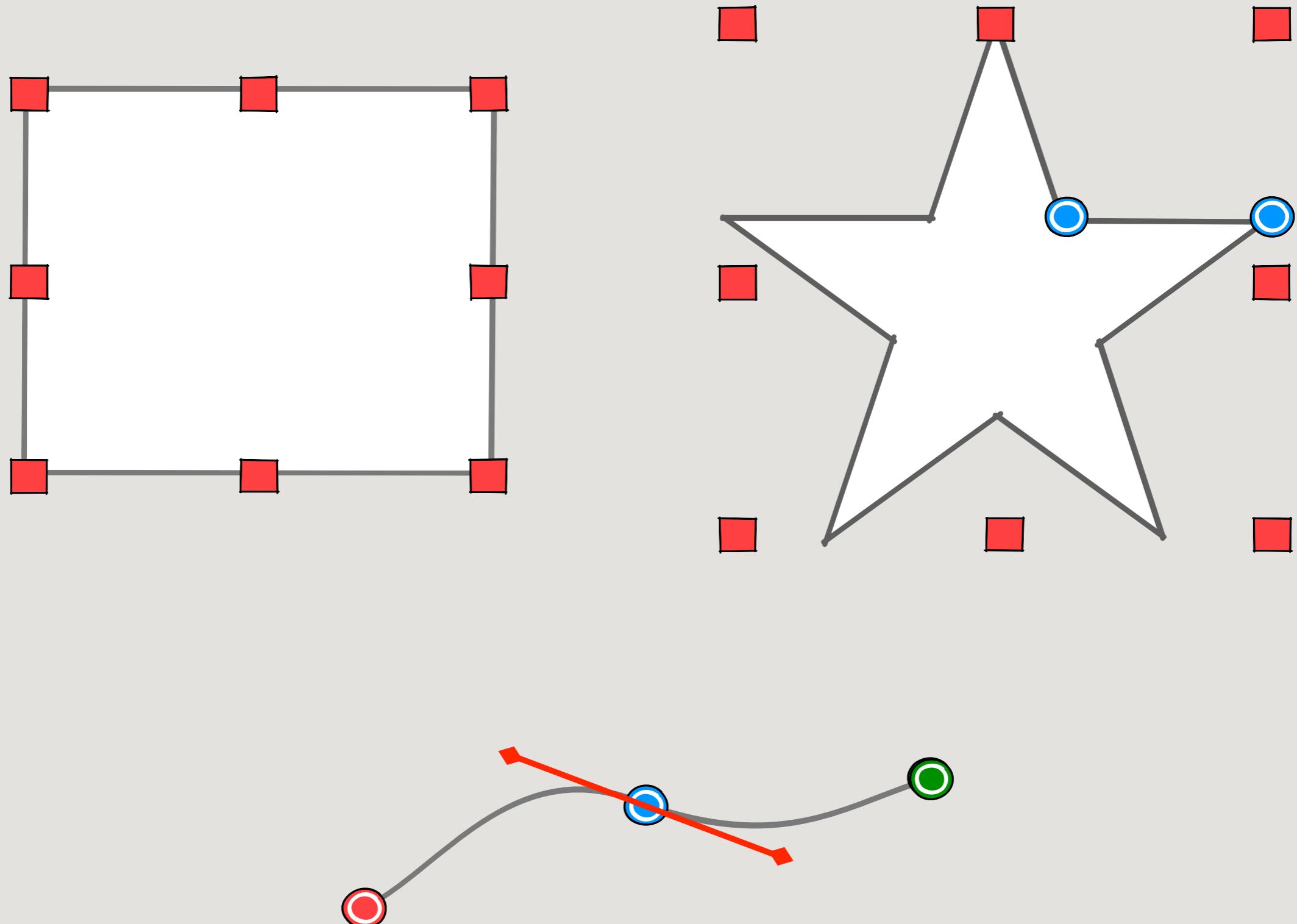


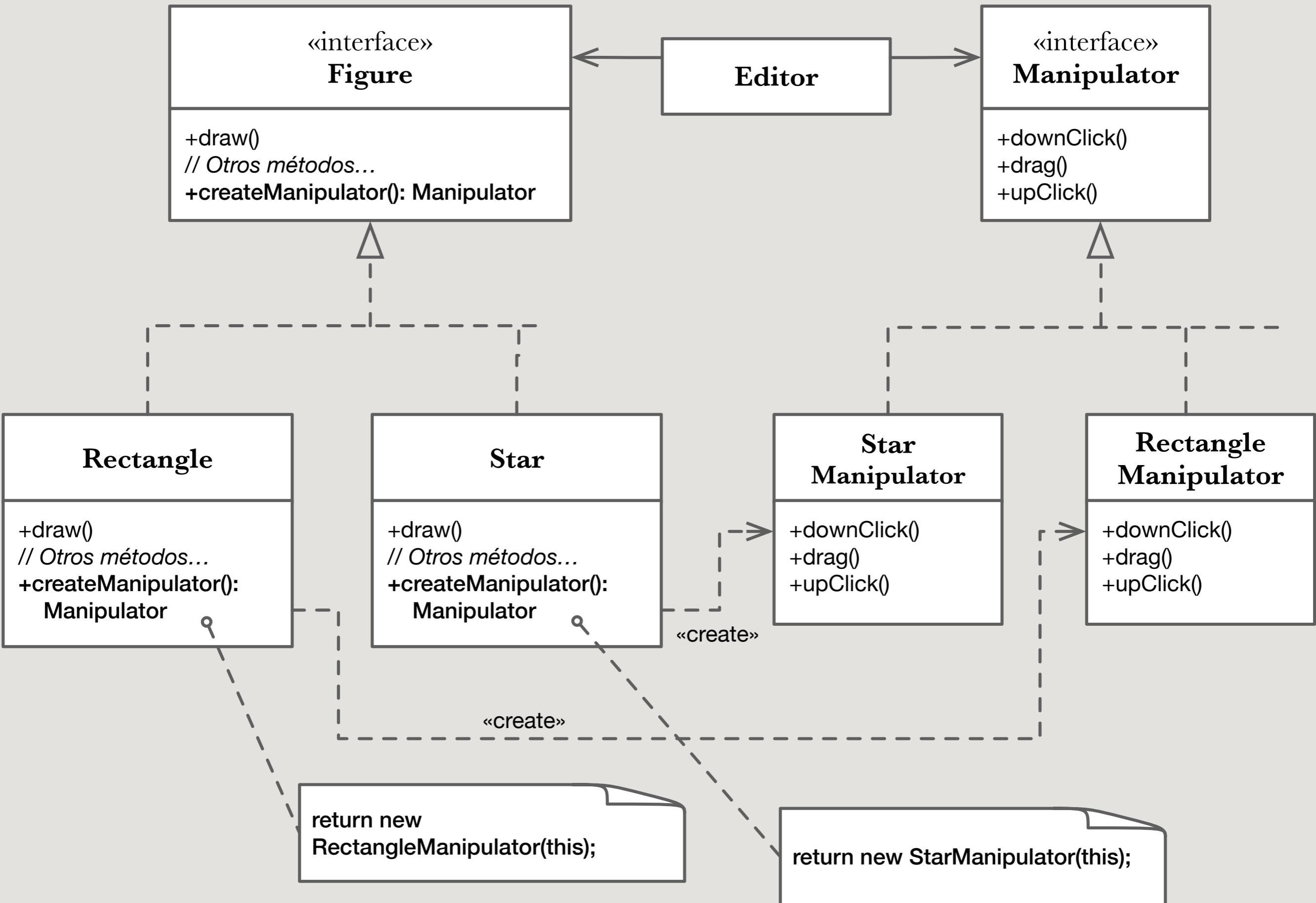












Diferentes figuras utilizarán diferentes subclases de manipuladores para manejar interacciones particulares.

La jerarquía de clases Manipulador resultante es paralela (al menos parcialmente) a la jerarquía de clases Figura.

La clase Figura proporciona un método de creación `createManipulator` que permite a los clientes crear el manipulador correspondiente para dicha figura.

Nótese cómo el método de creación define la conexión entre las dos jerarquías de clases.

Posee el conocimiento de qué clases pertenecen juntas.

Alternativamente, la clase Figura puede implementar CreateManipulator para devolver un manipulador predeterminado, y las subclases Figura pueden simplemente heredar dicha implementación. Las clases Figura que lo hagan no necesitarán la correspondiente subclase Manipulador, por lo que las jerarquías sólo serán parcialmente paralelas.

Así pues...

¿Qué diferencia a las jerarquías paralelas de la versión normal del patrón?

Que es el cliente quien
llama al método de
creación.

Que pasa a ser público, en vez de protegido.

Implementación

Dos variantes principales

- (1) El creador es una clase abstracta y no proporciona una implementación para el método de fábrica que declara.
- (2) Es una clase concreta y proporciona una implementación predeterminada para el método de creación.

(También es posible tener una clase abstracta que defina una implementación por defecto, aunque esta opción es probablemente menos común).

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.



EncryptionFactory.java

```
import java.security.Key;
import java.security.NoSuchAlgorithmException;

public class EncryptionFactory {
    /**
     * This method returns an instance of the appropriate subclass of
     * Encryption as determined from information provided by the given
     * Key object's getAlgorithm method.
     * @param key The key that will be used to perform the encryption.
     */
    public
    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 las
características
del lenguaje**

En Smalltalk un método podría devolver la clase del objeto a instanciar.

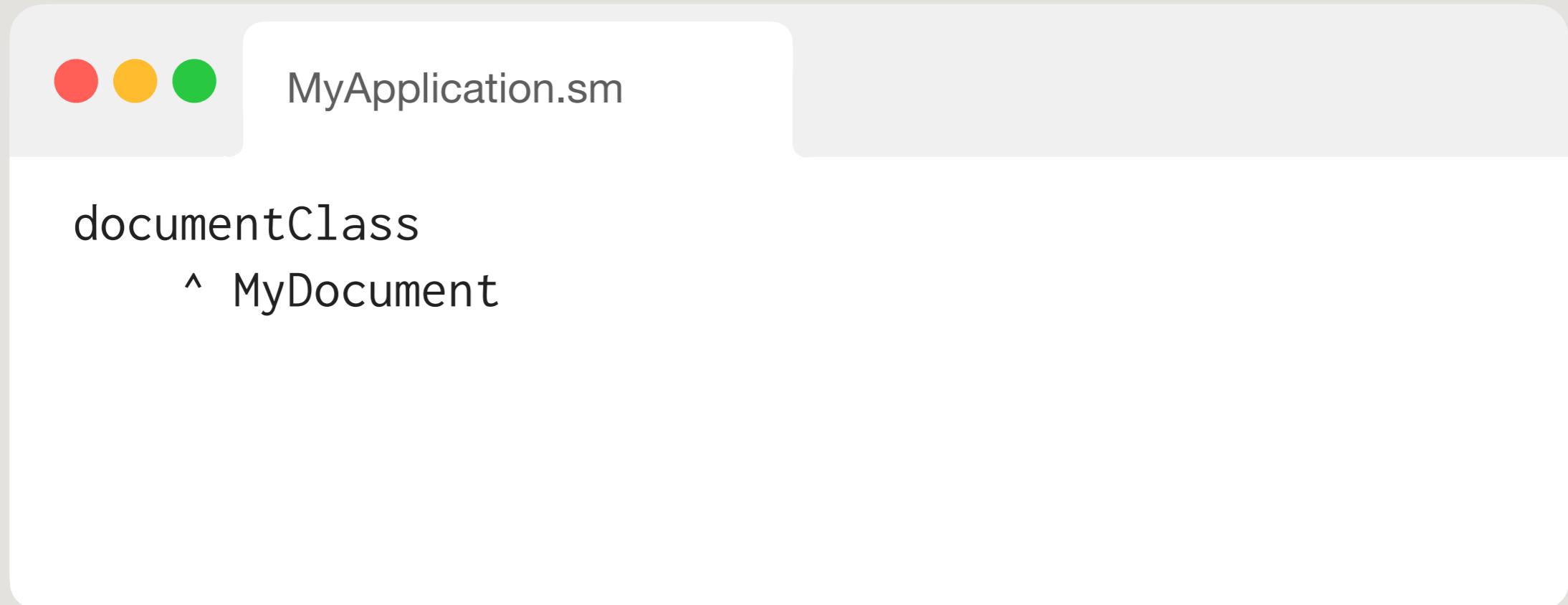


Application.sm

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

En Smalltalk un método podría devolver la clase del objeto a instanciar.



The image shows a screenshot of a Smalltalk application window. The title bar on the left contains three colored circles (red, yellow, green) and the text "MyApplication.sm". The main pane displays the following Smalltalk code:

```
documentClass
^ MyDocument
```



Application.sm

clientMethod

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

documentClass

self subclassResponsibility



MyApplication.sm

documentClass

```
^ MyDocument
```

Un enfoque aún más flexible, similar a los métodos de fábrica parametrizados, es almacenar la clase a crear como una variable de clase de Aplicación.

De esta forma no tenemos que heredar de Application para variar el producto.

(A menos que haya que heredar de todas formas por otros motivos, distintos de la mera aplicación del patrón).

En Java se podría lograr algo parecido por medio de la reflectividad.

Siempre y cuando los distintos productos definan un constructor con la misma signatura.

Convenios de nombrado

Es una buena práctica utilizar convenios de nombrado que dejen claro que estamos utilizando métodos de creación.

Por ejemplo, el marco de aplicación MacApp Macintosh siempre declara la operación abstracta que define el método de fábrica como Class^{*} DoMakeClass(), donde Class es la clase del producto.

protected abstract Figure createFigure(...);

Ejemplos de nombrado en la API de Java

(De métodos de creación en general, no necesariamente implementaciones del patrón Factory Method).

`valueOf` `getType`

`of` `newType`

`EnumSet.of(...)`

`getInstance`

`newInstance`

Código de ejemplo

Usos conocidos

Iteradores en las colecciones de Java

Patrones relacionados

Abstract Factory

A menudo se implementa con factory methods.

Template Method

Los factory method normalmente se llaman desde template methods.

Prototype

No necesitan la jerarquía de creadores.

A veces pueden ser alternativas.

Abstract Factory

Patrón de creación, de objetos

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

**También conocido
como
Kit**

Motivación

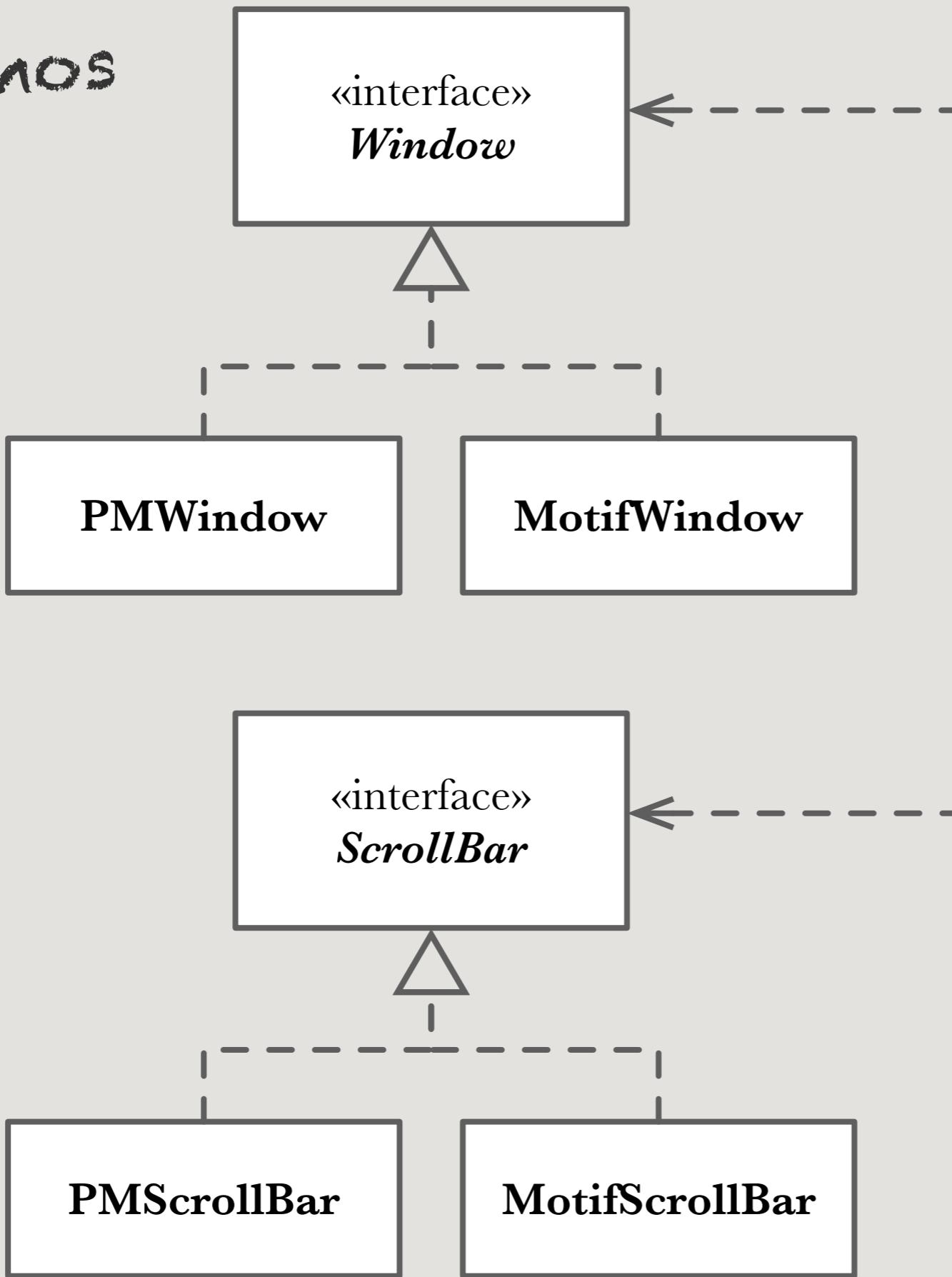
Sea una biblioteca gráfica que
permite generar interfaces para
diferentes entornos de ventanas

Motif, Presentation Manager...

Cada uno de ellos tendrá una clase distinta para representar una ventana, una barra de desplazamiento, un botón...

Si queremos que una aplicación sea portable, no podrá crear directamente objetos de esas clases específicas.

**Queremos
esto**

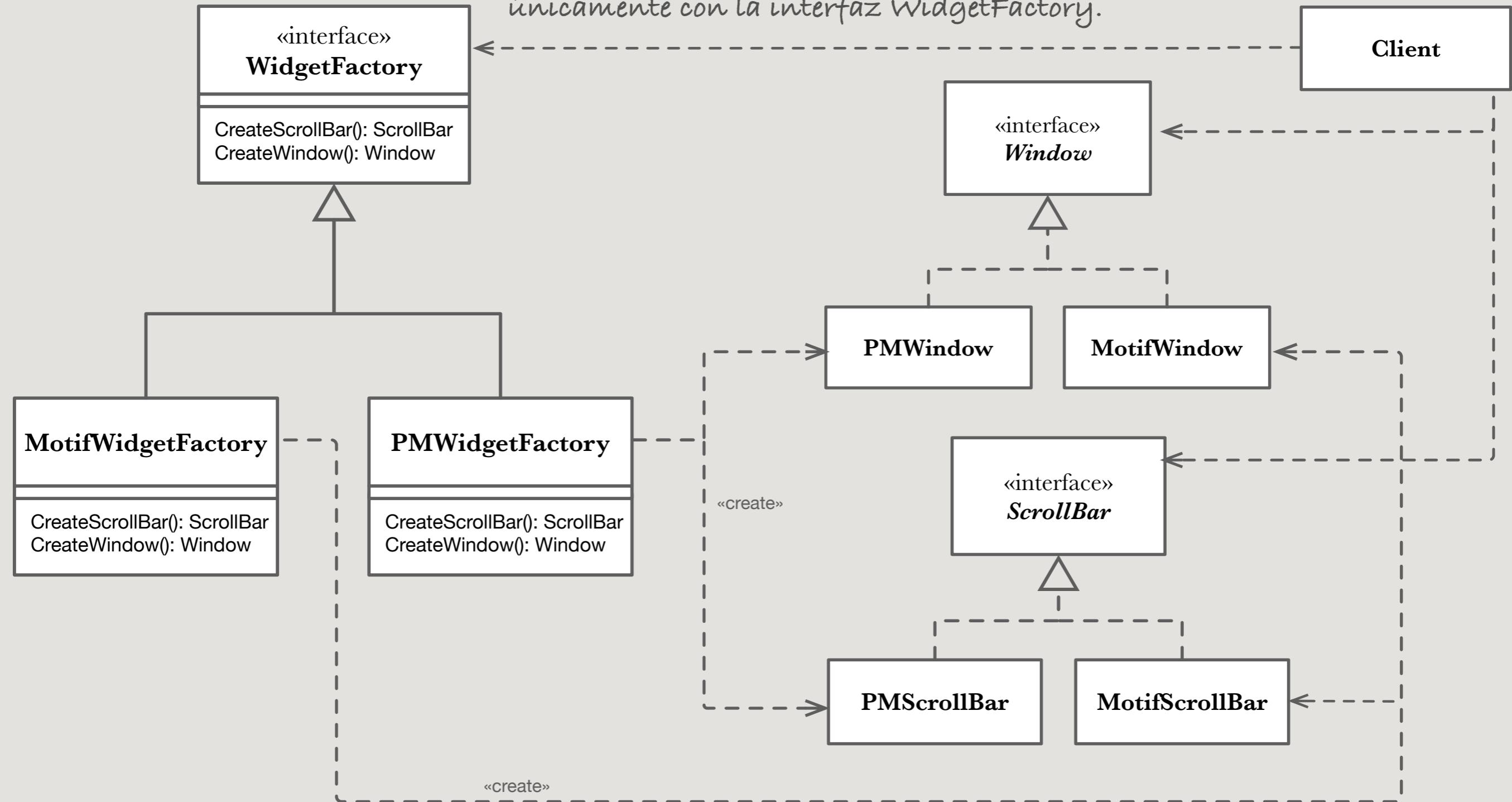


Es decir, que el cliente se relacione únicamente con las interfaces *Window*, *ScrollBar*, etcétera, y no con sus implementaciones concretas.

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

¿Entonces?

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.

Aplicabilidad

Úsese el patrón Abstract Factory cuando

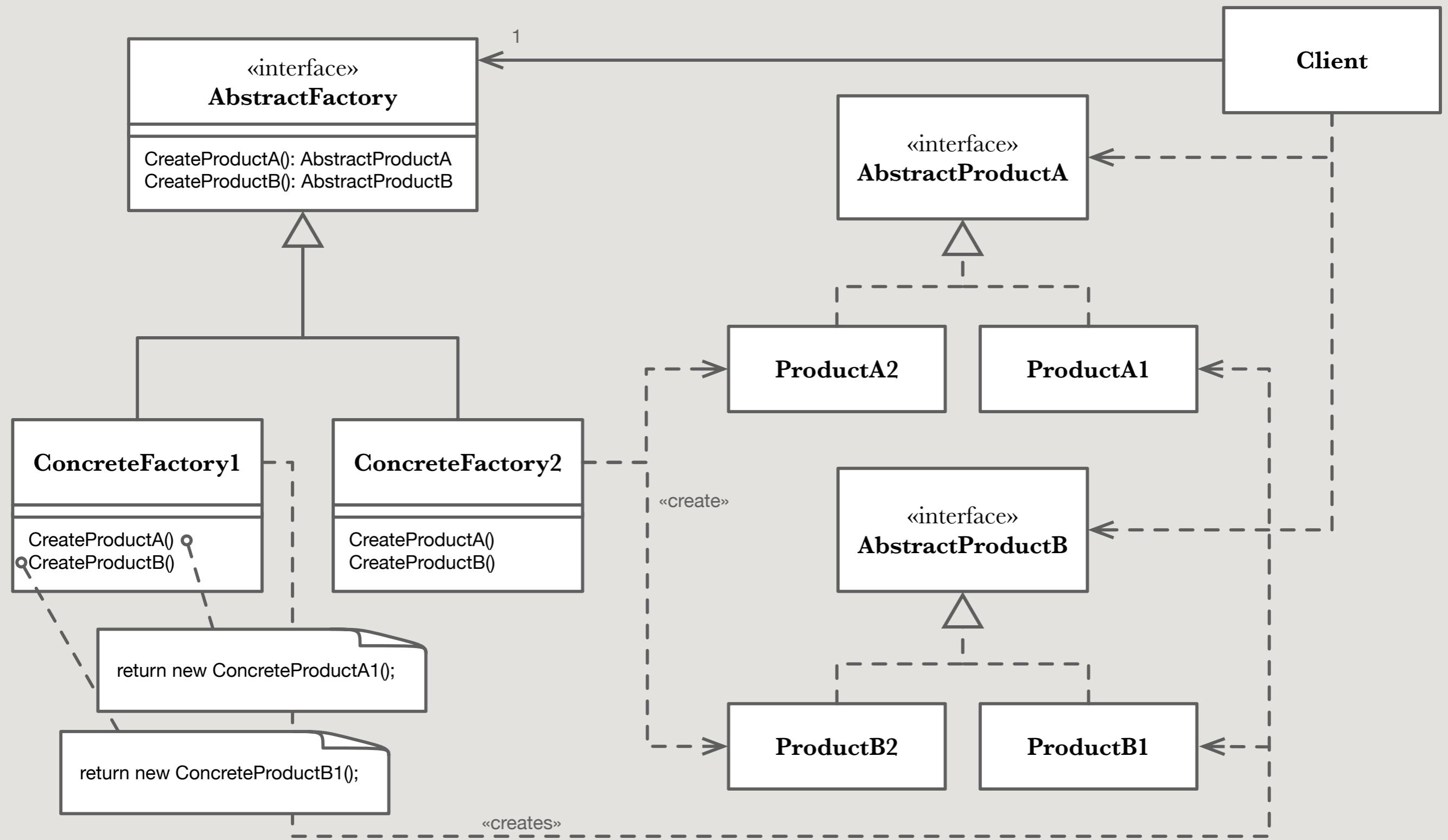
**Un sistema debe ser
independiente de cómo se crean,
componen y representan sus
productos.**

**Un sistema debe configurarse
con una de varias familias de
productos.**

Una familia de objetos de producto relacionados está diseñada para ser utilizada de forma conjunta, por lo que es necesario imponer esta restricción.

**Queremos proporcionar una
biblioteca de clases de
productos, y queremos revelar
sólo sus interfaces, no sus
implementaciones.**

Estructura



Participantes

AbstractFactory

(WidgetFactory)

Declara una interfaz para operaciones que crean objetos producto abstractos.

ConcreteFactory

(MotifWidgetFactory, PMWidgetFactory)

Implementa las operaciones para crear objetos de producto concretos.

AbstractProduct

(Window, ScrollBar)

Declara una interfaz para un tipo de objeto producto.

ConcreteProduct

(MotifWindow, MotifScrollBar)

Define un objeto producto que será creado por la fábrica concreta correspondiente.

Implementa la interfaz AbstractProduct.

Client

(MotifWindow, MotifScrollBar)

Utiliza únicamente interfaces declaradas por las clases AbstractFactory y AbstractProduct.

Colaboraciones

Normalmente se crea una única instancia de una clase `ConcreteFactory` en tiempo de ejecución.

Esta fábrica concreta crea objetos producto que tienen una implementación particular.

Para crear objetos producto diferentes, los clientes deben utilizar una fábrica concreta distinta.

AbstractFactory difiere la creación de objetos producto a su subclase **ConcreteFactory**.

Consecuencias

Aísla a las clases concretas.

Los clientes manipulan los productos únicamente a través de sus interfaces abstractas.

Los nombres de las clases de los productos están encapsuladas en cada fábrica concreta, no aparecen en el código del cliente.

Permite intercambiar fácilmente familias de productos.

Las clase de una fábrica concreta aparece una sola vez en una aplicación: cuando se crea.

Esto hace que sea muy fácil sustituir la fábrica concreta con otra, y pasar a usar así diferentes categorías de productos.

Promueve la consistencia entre los productos.

Solo se pueden usar conjuntamente los objetos de cada familia.

El patrón garantiza lo anterior, hace que no se puedan mezclar productos de familias distintas.

Dificulta añadir nuevos tipos de productos.

Los productos que se pueden crear son los que define la interfaz de la fábrica abstracta.

Añadir nuevos productos implicaría cambiar dicha interfaz y añadir un nuevo método.

Implementación

Singletons

Normalmente una aplicación solo necesita una fábrica concreta por cada familia.

Así que a veces es posible implementarlas como singletons.

Creación de los productos

En su implementación habitual, una fábrica concreta crea sus productos redefiniendo un **factory method** para cada uno.

Necesita una nueva subclase para cada familia.

Creación de los productos

Otra posibilidad es usar el patrón Prototype.

La fábrica concreta se inicializa con una instancia prototípica de cada producto de la familia, y crea un nuevo producto clonando su prototipo.

La implementación con prototipos elimina la necesidad de una nueva clase de fábrica concreta para cada nueva familia de productos.

Fábricas extensibles

El patrón normalmente define una nueva operación por cada producto.

Añadir un nuevo tipo de producto requiere cambiar la interfaz AbstractFactory y todas las clases que dependen de ella.

Un diseño más flexible pero menos seguro es añadir un parámetro a las operaciones que crean objetos.

Este parámetro especifica el tipo de objeto que se va a crear. Puede ser un identificador de clase, un entero, una cadena o cualquier otra cosa que identifique el tipo de producto.

Con este enfoque, `AbstractFactory` solo necesita una única operación `make` con un parámetro que indique el tipo de objeto a crear.

Esta es la técnica utilizada en las fábricas abstractas basadas en prototipos comentadas anteriormente.

Pero tiene el problema de que todos los productos se devuelven al cliente con la misma interfaz dada por el tipo de retorno.

El cliente no podrá diferenciar o hacer suposiciones seguras sobre la clase de un producto.

A cambio de una interfaz flexible y extensible perdemos la comprobación estática de tipos.

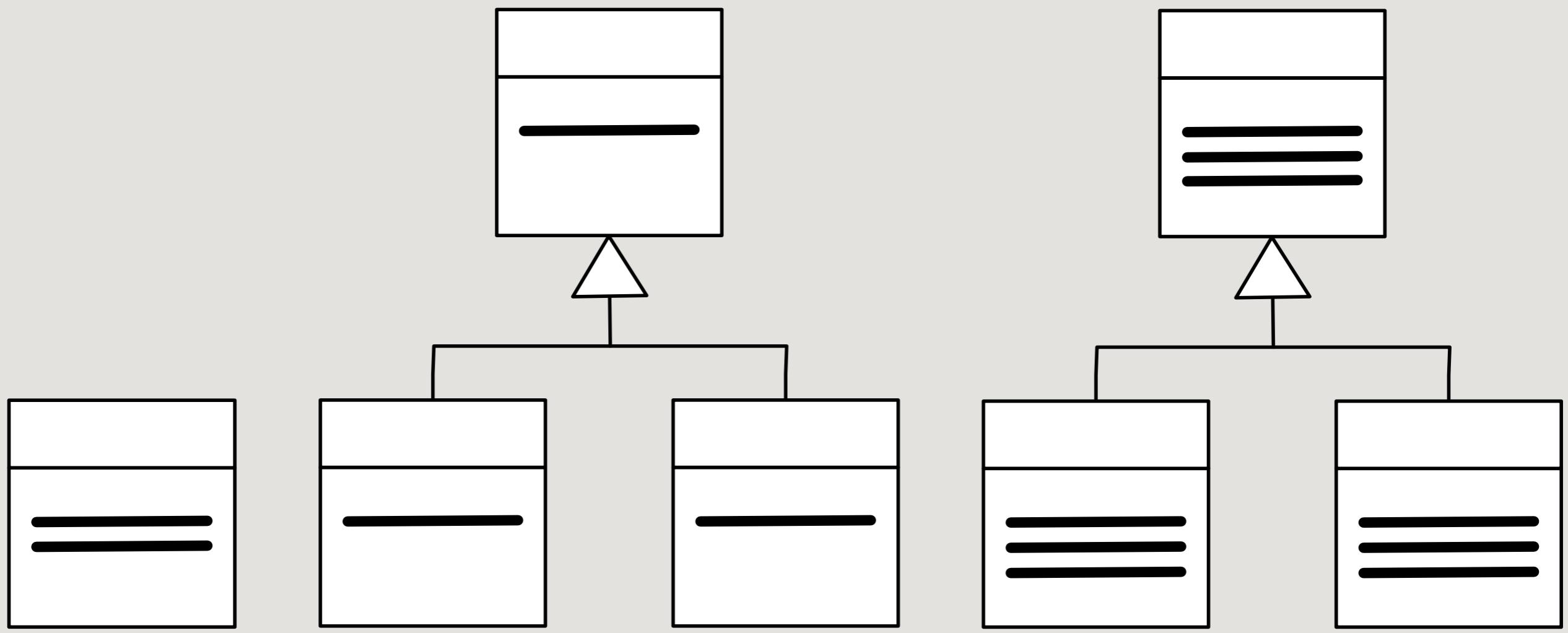
[...] But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the same abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass-specific operations, they won't be accessible through the abstract interface. Although the client could perform a downcast (e.g., with `dynamic_cast` in C++), that's not always feasible or safe, because the downcast can fail. This is the classic trade-off for a highly flexible and extensible interface.

Patrones relacionados

Factory Method y Prototype

El patrón Abstract Factory normalmente se implementa con una serie de métodos de fabricación, pero también existe la variante con prototipos.

Comparativa

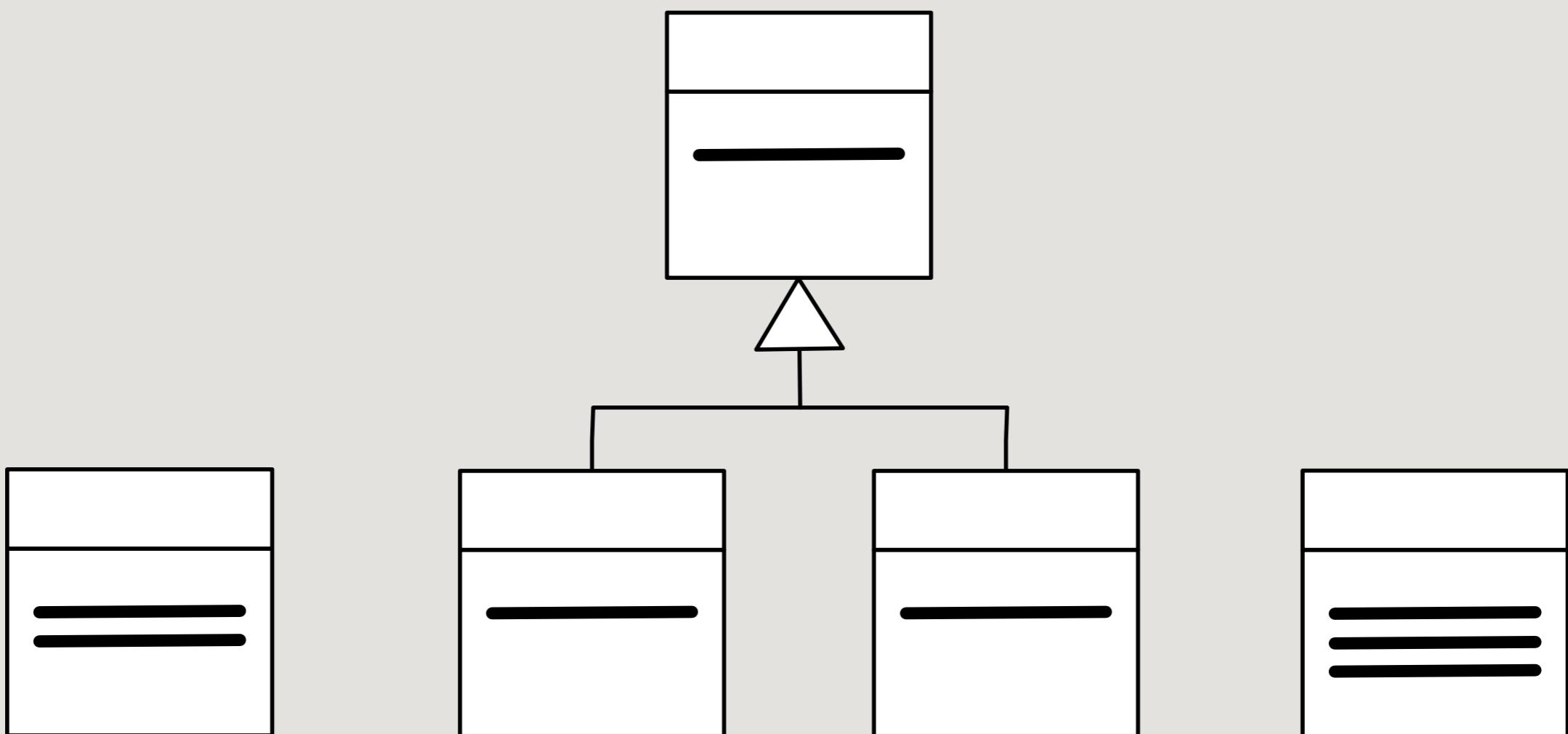


**Simple
Factory**

Factory Method

**Abstract Factory
(with factory methods)**

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.



**Simple
Factory**

Factory Method

**Abstract Factory
(with prototypes)**



Abstract Factory Vs Factory Method

I'm finding myself getting a bit muddled by these two patterns while reading [DesignPatternsBook](#).

Both classes have abstract classes (Abstract Factory, Abstract Product in Abstract Factory & Abstract Creator and Abstract Product in Factory Method). Both have subclasses that customize the classes for specific use - (Concrete Factory, Concrete Product, Concrete Creator, Concrete Product).

So what's the difference between Factory and Creator? Both

- have specific subclasses for specific applications/versions of products
- have specific product classes created by the subclasses

In the Create Maze Examples -

The Factory knows WHAT to create and gets its instructions on WHEN from another object (Instruction through ObjectComposition)

The Creator knows WHAT to create and has a shared method that tells it WHEN to create another object. (Instruction through Inheritance)

But the book also says *Abstract Factory is often implemented with factory methods*. So perhaps WHEN is not important.. and I'm missing something obvious here.

So:

Is [FactoryMethodPattern](#) simply: Each subclass knows what other classes to use/objects to create.

[AbstractFactoryPattern](#) is: Swapping [FactoryMethod](#) Classes with Object Composition

With [FactoryMethodPattern](#), the method is in the object that uses it. With [AbstractFactoryPattern](#), it's moved to a separate factory object. It's the difference between "Foo foo = this.createFoo();" and "Foo foo = factory.createFoo();" -- [JasonArhart](#)

Factorizaciones



The Addison-Wesley Signature Series

Book A MARTIN FOWLER
Signature Edition

REFACTORING TO PATTERNS

JOSHUA KERIEVSKY

software
development
15th annual
productivity
award



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

Catálogo de refactorizaciones basado en patrones

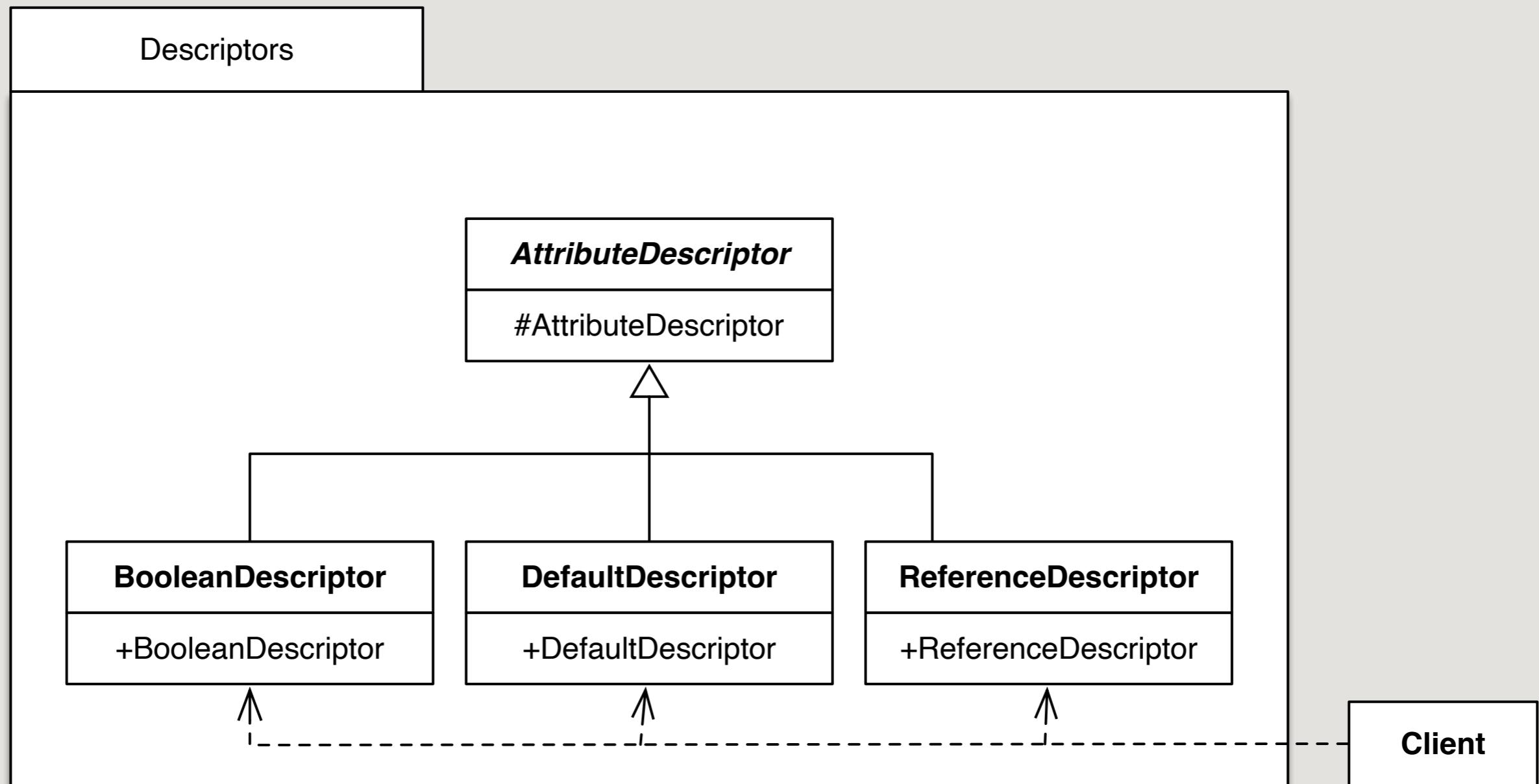
Se llega a los patrones a partir del código existente, no de un diseño previo.

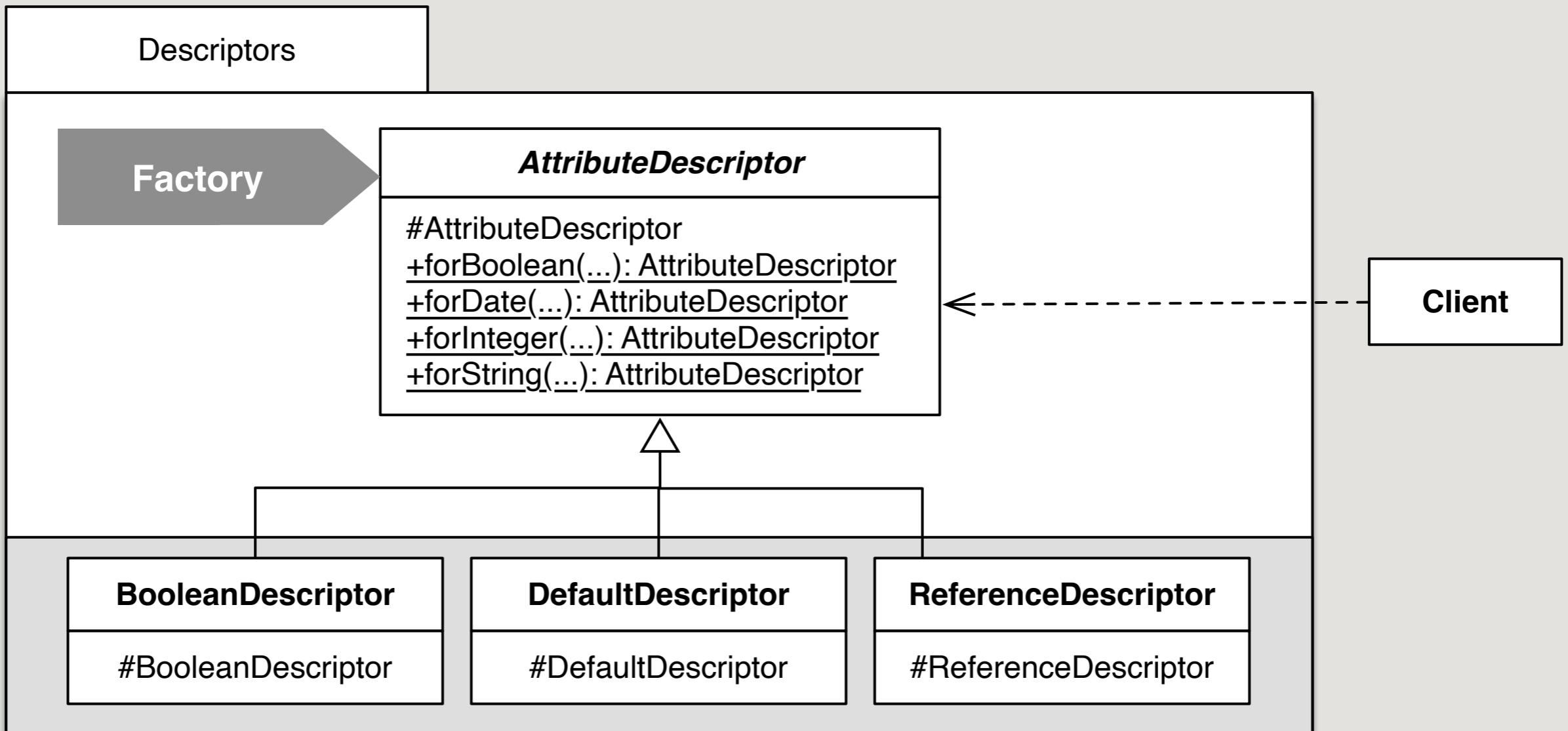
Encapsulate
Classes with
Factory

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



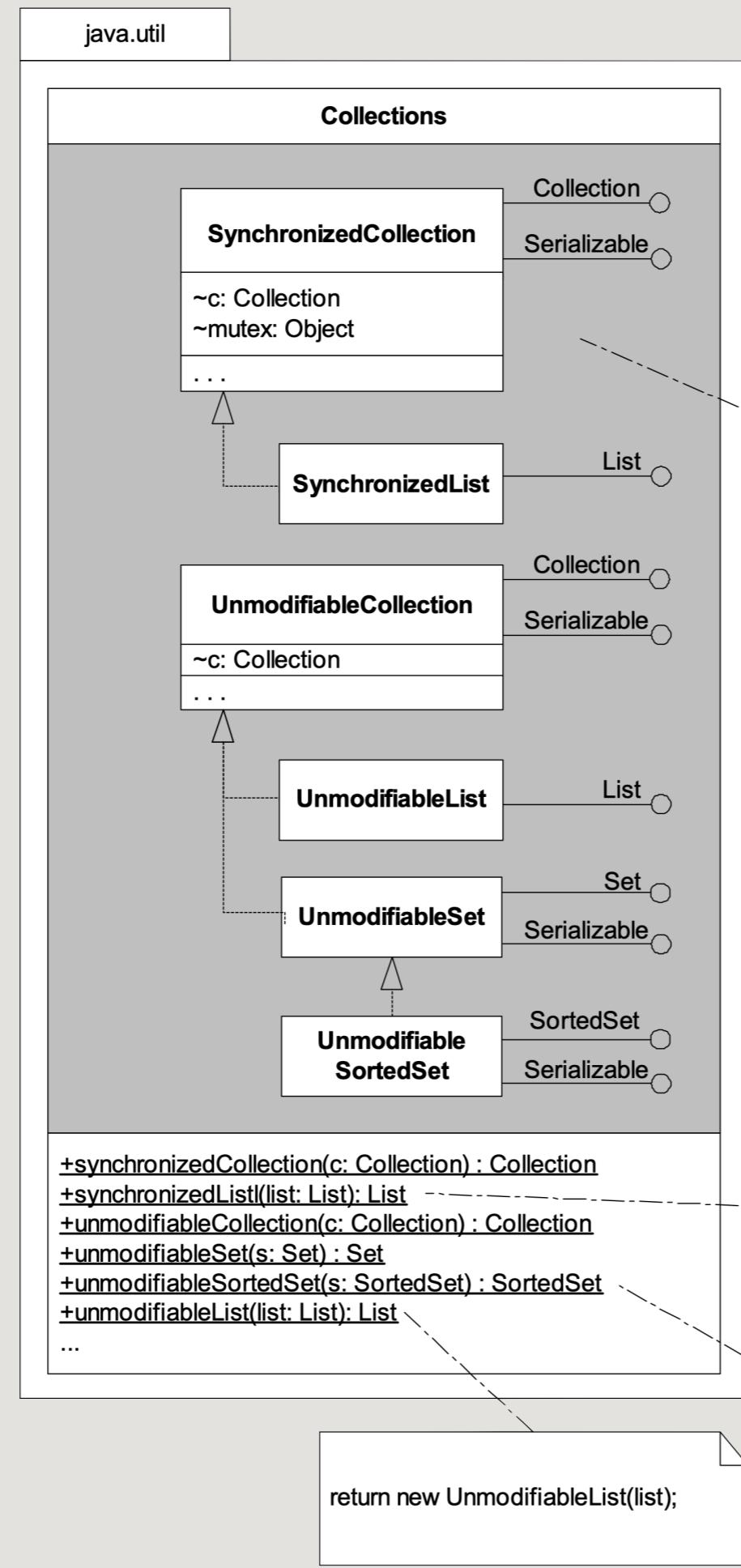
*Hacer los constructores de las clases no
públicos y dejar que los clientes creen
objetos de ellas usando una factoría.*





- + Simplifica la creación de los distintos tipos de objetos mediante una serie de métodos que revelan su intención.
- + Reduce el «peso conceptual» de un paquete al ocultar las clases que no necesitan ser públicas.
- + Aplicación del principio «programar para una interfaz, no para una implementación».
- 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
- Limita la personalización cuando los clientes solo pueden acceder al código binario de una fábrica, no a su código fuente.

Variante: clases internas



some of the many
non-public, inner classes
inside the Collections class

return new SynchronizedList(list);

return new SortedSet(s);

return new UnmodifiableList(list);

Otro ejemplo en la
API de Java

EnumSet

No tiene constructores públicos, solo métodos de fabricación estático. Devuelve dos posibles implementaciones:

- **RegularEnumSet** (hasta 64 elementos)
Usa un long.
- **JumboEnumSet** (64 o más elementos)
Usa 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.

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

Variantes

Parameterised 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 esta.

En ese caso, se puede sacar fuera la lógica de creación, a una clase factoría.

Ejemplo en la API
de Java

¿Qué creéis que crea este constructor?

BigInteger(int, int, Random)

¿Y este método estático?

Añadido en Java 1.4

`probablePrime(int, Random)`

- + Al poder tener otros nombres, comunican mejor que los constructores qué tipo de objetos están creando.
- + No tienen las limitaciones de los constructores, como no poder tener dos con el mismo nombre y tipo de parámetros.
- + Facilita localizar código de creación que no se está usando.
- Hace a la creación no estándar: en unos sitios con new y en otros empleando métodos de creación.

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.

