Patrones de creación

¿Para qué se utilizan?

Este tipo de patrones resuelve problemas relacionados con la creación de instancias de objetos.

Los patrones de creación que se explicarán a continuación son:

- Abstract factory
- Buider
- Factory method
- Prototype
- Singleton

Abstract factory

- Sinopsis: Proporciona una clase que delega la creación de una o más clases concretas con el fin de entregar objetos específicos
- Contexto: Resuelve el problema de crear familias de objetos.

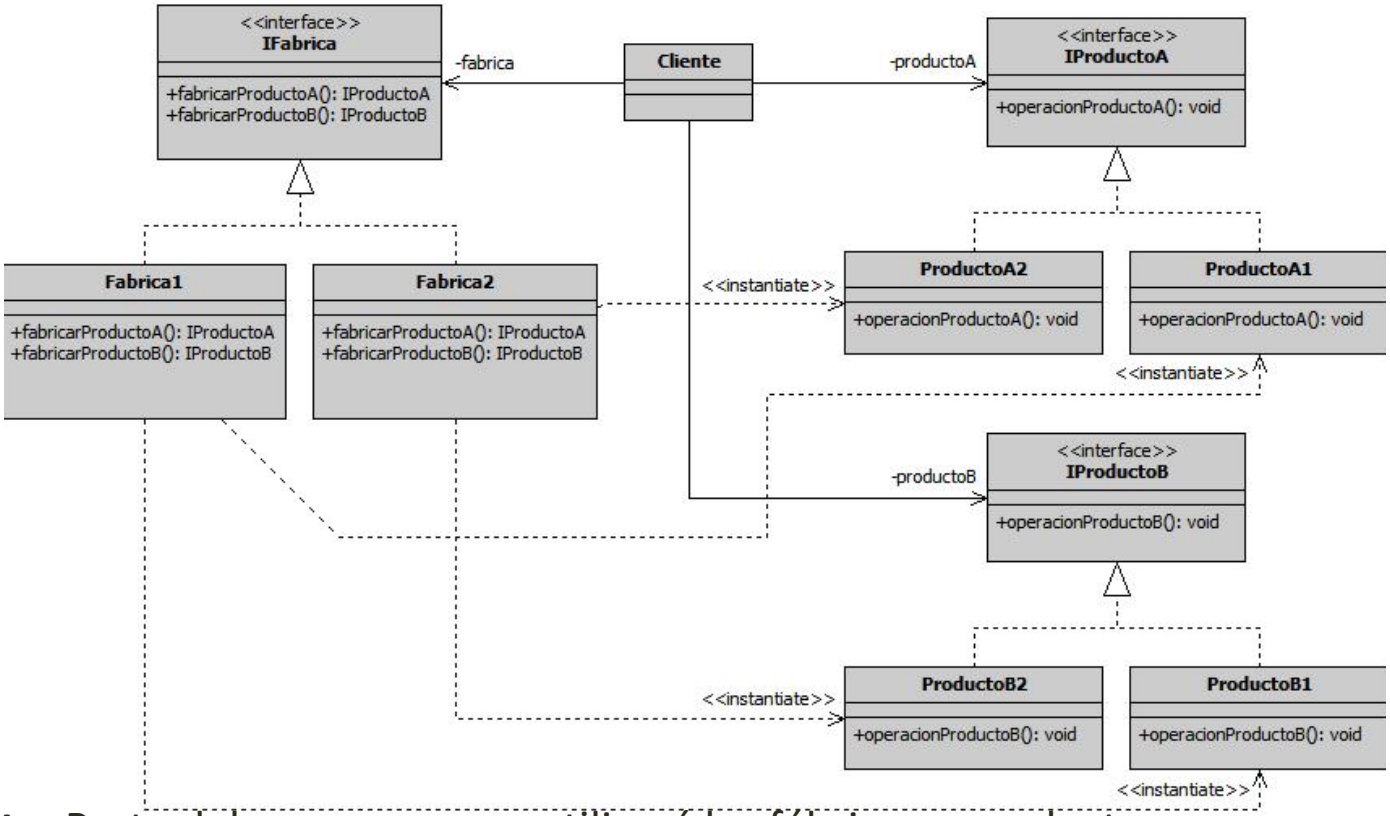
Abstract factory

- Causas:
  - La creación de objetos debe ser independiente del sistema que los utilice.
  - Los sistemas deben ser capaces de utilizar múltiples familias de objetos.
  - Se usan bibliotecas sin exponer detalles de la implementación.

Abstract factory

- Solución: En este patrón se crean ciertas clases adicionales llamadas *fábricas*. Estas clases son las encargadas de crear los diferentes tipos de ventanas y botones.

Abstract factory



Cliente: Parte del programa que utilizará las fábricas y productos.
IProductoA: Interfaz que define un ejemplo de producto.
ProductoA1 y **ProductoA2**: Los diferentes tipos de ese producto.
IProductoB: Interfaz que define otro ejemplo de producto.
ProductoB1 y **ProductoB2**: Los diferentes tipos de ese producto.
IFabrica: Interfaz que define las funciones de creación de productos.
Fabrica1 y **Fabrica2**: Clases encargadas de crear los productos.

Abstract factory

- Pensemos que, en nuestro programa, tenemos las clases **Ventana** y **Boton**. Pongamos, por ejemplo, que tenemos 2 interfaces diferentes: una con colores claros y otra con colores oscuros.
- La forma más básica de hacerlo sería de esta manera:

```
// A la hora de seleccionar la interfaz
var GUI:String = "clara"; // u "oscura";

// A la hora de crear un botón
if(GUI == "clara"){
new BotonClaro();
}else if(GUI == "oscura"){
new BotonOscuro();
}

// A la hora de crear una ventana
if(GUI == "clara"){
new VentanaClara();
}else if(GUI == "oscura"){
new VentanaOscura();
}
```

Esto implicaría realizar una comprobación de la interfaz seleccionada cada vez que se quiera crear una **Ventana** o un **Boton**.

La mejor opción en este caso sería utilizar el patrón **Abstract Factory**.

Abstract factory

- Después de utilizar el patrón, nuestro código quedaría así:

```
// A la hora de seleccionar la interfaz
var GUI:InterfazGrafica = new InterfazClara(); // o new InterfazOscura();

// A la hora de crear un botón
GUI.crearBoton();

// A la hora de crear una ventana
GUI.crearVentana();
```

Según el tipo de **InterfazGrafica** instanciada, se crearán ventanas/botones de un tipo u otro dinámicamente, sin necesidad de comprobar a mano qué interfaz gráfica se está utilizando.

Builder

- Sinopsis: Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
- Contexto: Se utiliza cuando queremos crear un producto que tiene diferentes partes.

Builder

- Causas:
  - Los algoritmos de creación de objetos deben ser desacoplados del sistema.
  - Son obligatorias múltiples representaciones de algoritmos de creación.
  - Se requiere control sobre el proceso de creación en tiempo de ejecución.

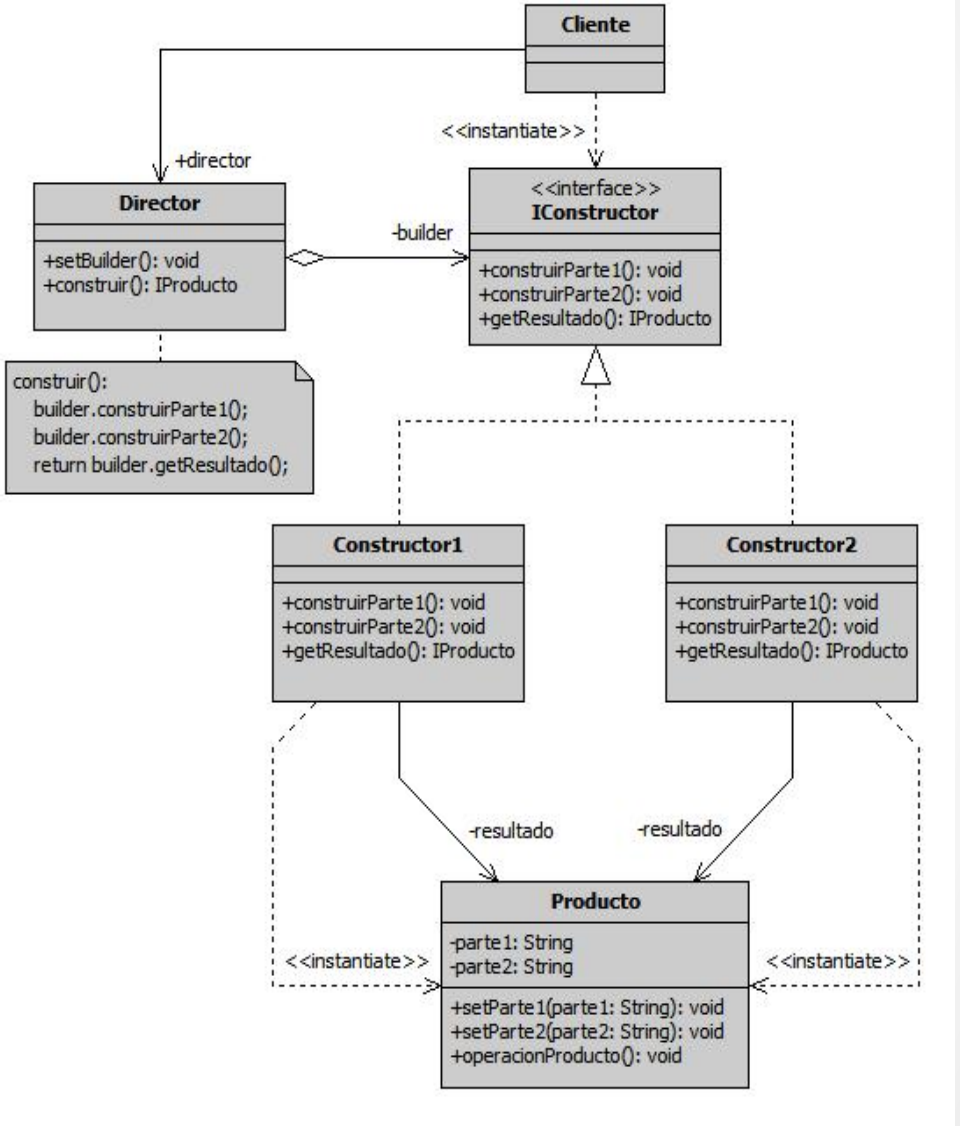
Builder

- Ejemplo: Imaginemos la cocina de una pizzería donde se hacen pizzas. Las pizzas constan de varias partes (masa, salsa y relleno), y podríamos tener 2 cocineros, cada uno especialista en un tipo de pizza. Esto nos llevaría a tener 5 clases:

- **Cocina**
- **Pizza**
- **Cocinero**
- **CocineroHawai**
- **CocineroPicante**

Builder

- **Cliente**: Parte del programa que utilizará el resto de clases.
- **Director**: Clase que decide qué constructor se utiliza y cuando se debe construir el producto.
- **IConstructor**: Interfaz que define las funciones de creación de cada parte del producto y la función de obtención del producto resultante.
- **Constructor1** y **Constructor2**: Clases encargadas de crear las partes del producto.
- **Producto**: Clase del producto en sí.



Builder

- Así el código quedaría

```
var cocina:Cocina = new Cocina();

// Decidimos que se crearán pizzas hawaianas
cocina.elegirCocinero(new CocineroHawai());

// Creamos la pizza
var pizzaHawaiana:Pizza = cocina.nuevaPizza();
```

El código de la clase **Cocina** podría ser algo así:

```
package {
public class Cocina {

private var cocinero:Cocinero;

public function elegirCocinero(cocinero:Cocinero):void {
this.cocinero = cocinero;
}

public function nuevaPizza():Pizza {
cocinero.hacerMasa();
cocinero.utilizarSalsa();
cocinero.hacerRelleno();
}

}
}
```

Factory method

- Sinopsis: Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
- Contexto: Pretende que solo exista un producto y no una familia de ellos.

Factory method

- Causas:
  - Una clase no puede anticipar el tipo de objeto que debe crear
  - Subclases pueden especificar qué objetos deben ser creados.

Factory method

- Ejemplo:

Imaginemos que deseamos crear un juego estilo Tetris. En este juego tendríamos diferentes tipos de piezas. Esto nos llevaría a tener una clase por cada tipo de pieza:

- **PiezaL**
- **PiezaT**
- **PiezaI**
- **...**

Factory method

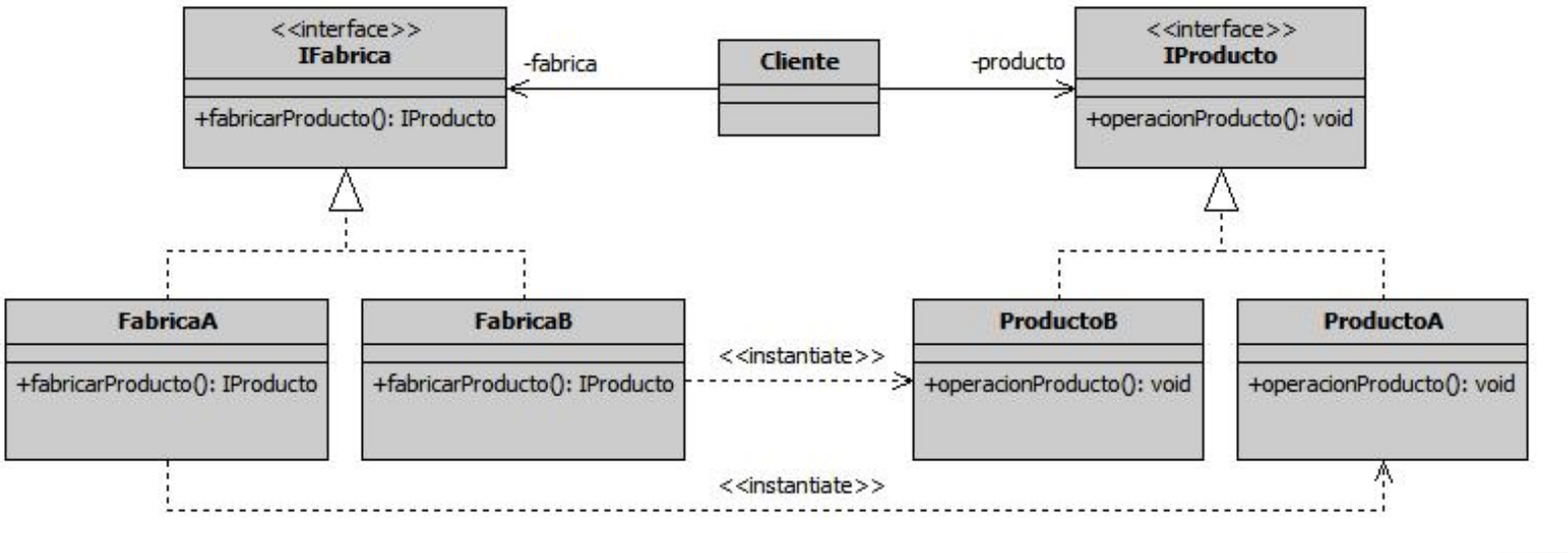
- Cada vez que se crea una pieza nueva, desearíamos seleccionar el tipo de pieza de forma aleatoria. La forma más básica de hacerlo sería la siguiente:

```
// Seleccionaríamos el tipo de pieza aleatoriamente
var tipo:uint = Math.random()*7; // 7 es el número de piezas diferentes en el Tetris

// Creamos dicha pieza
switch(tipo){
case 1:
new PiezaL();
break;
case 2:
new PiezaT();
break;
case 3:
new PiezaI();
break;
// ...
}
```

Factory method

- Solución:



En la siguiente página se identifica cada clase del diagrama con las clases de nuestro ejemplo:

Factory method

- **Cliente**: Parte del programa que utilizará las fábricas y productos. Podría ser el archivo **.fla** principal, por ejemplo.

- **IProducto**: Interfaz que define el producto. Se correspondería con una clase **Pieza** en nuestro ejemplo.
- **ProductoA** y **ProductoB**: Los diferentes tipos del producto. Se corresponderían con la clases **PiezaL**, **PiezaT** y **PiezaI**.

- **IFabrica**: Interfaz que define las función de creación del producto. En nuestro ejemplo podría llamarse **ICreador** y definiría la función *crearPieza():Pieza*.

- **FabricaA** y **FabricaB**: Clases encargadas de crear los productos. En nuestro ejemplo, serían **CreadorL** (que crearía instancias de **PiezaI**), **CreadorT** (que crearía instancias de **PiezaT**), **CreadorI** (que crearía instancias de **PiezaI**), etc.

Factory method

- Una vez utilizado el patrón, el código anterior quedaría así:

```
// Crearíamos una lista con todas las fábricas
var creadores:Vector.<ICreador> = new Vector.<ICreador>();
creadores.push(new CreadorL(), new CreadorT(), new CreadorI());

// Seleccionaríamos el tipo de pieza aleatoriamente
var tipo:uint = Math.random()*7; // 7 es el número de piezas diferentes en el Tetris

// Creamos dicha pieza
creadores[i].crearPieza();
```

De esta manera no necesitaríamos un switch, sino que se crearía la pieza a través del *creador* seleccionado.

Prototype

- Sinopsis: Crear objetos mediante clonación basados en una plantilla de objetos existentes
- Contexto: Resuelve el problema de duplicar objetos ya creados con anterioridad

Prototype

- Causas:
  - Las clases que se creen se especifican en tiempo de ejecución.
  - Para un objeto existen un número limitado de combinaciones de estado.
  - La creación inicial de cada objeto es una operación costosa.

Prototype

- Ejemplo:
- Imaginemos que tenemos un programa de dibujo por ordenador en el cual podemos crear círculos y cuadrados. Cuando se crea un círculo, éste tiene un radio de 50 píxeles y es de color rojo. Sin embargo, podemos redimensionar el círculo y cambiar su color. Cuando se crea un cuadrado, tiene 50 píxeles de lado y es de color azul.

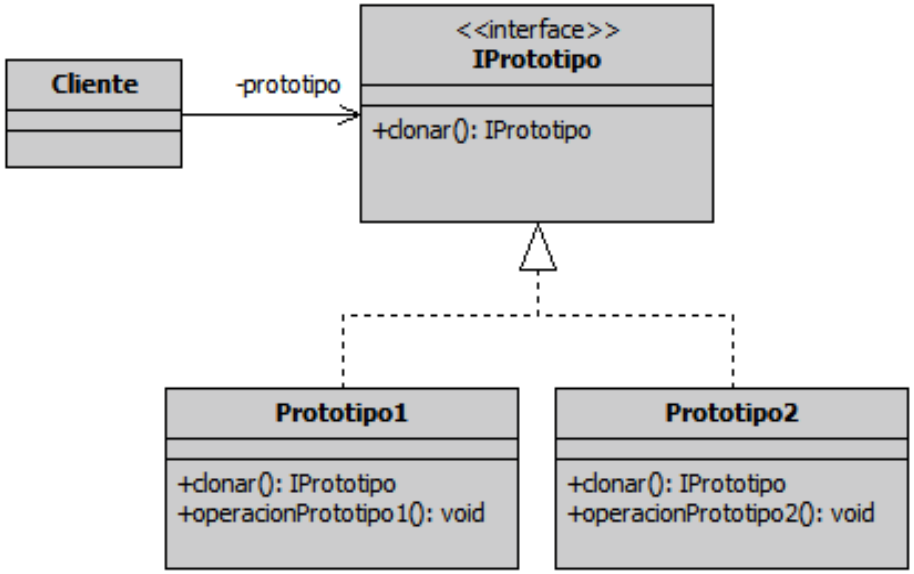
Prototype

- Imaginemos ahora que el usuario decide crear un círculo y modifica su color y tamaño. Acto seguido, el usuario decide hacer una copia de dicho círculo. El código sería el siguiente:

```
var circuloNuevo:Circulo = new Circulo();
circuloNuevo.color = circuloExistente.getColor();
circuloNuevo.radio = circuloExistente.getRadio();
```

Uno de los problemas más inmediatos de hacerlo de esta manera es que, si se añaden nuevos atributos a la clase **Circulo**, habría que modificar el código en cada lugar donde se haya hecho una copia de un **Circulo**.

Prototype



- **Cliente**: Parte del programa que utilizará las fábricas y productos. Podría ser el archivo **.fla** principal, por ejemplo.
- **IPrototipo**: Interfaz que define el método *clonar():IPrototipo*. En nuestro ejemplo podría ser una clase llamada **IObjetoGrafico**.
- **Prototipo1** y **Prototipo2**: Las diferentes clases que implementarán el método de clonación. Se corresponderían con la clases **Circulo** y **Cuadrado** de nuestro ejemplo.

Prototype

- Una vez utilizado el patrón, el código anterior quedaría así:

```
var circuloNuevo:Circulo = circuloExistente.clonar();
```

Si ahora quisiéramos añadir nuevos atributos a la clase **Circulo**, sólo habría que modificar el método *clonar():IPrototipo* de la clase **Circulo**.

Prototype

- Consecuencias:

Este patrón permite la especificación de nuevos objetos generando un objeto con valores por defecto sobre el que posteriormente se podrán aplicar cambios. La especificación de nuevos objetos también puede realizarse mediante la variación de su estructura. Reducción del número de subclases.

Singleton

- Sinopsis: Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella.
- Contexto: Restringir la instanciación de una clase o valor de un tipo a un solo objeto.

Singleton

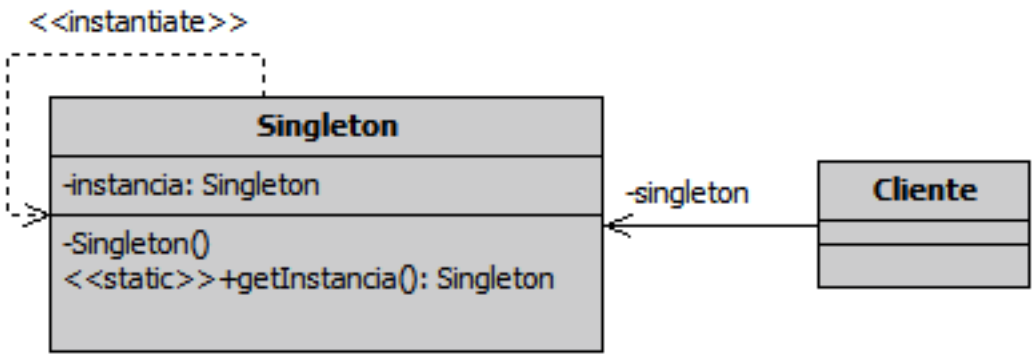
- Causas:
  - Se requiere exactamente una instancia de una clase
  - Es necesario acceso controlado a un solo objeto

Singleton

- Ejemplo: Imaginemos un programa que, al hacer click en un icono de ayuda, cree una ventana nueva con los documentos de ayuda del programa. Normalmente, si el usuario hiciese click en el botón nuevamente, se abriría una nueva ventana, y así sucesivamente.

- Sin embargo, podríamos desear que, si la ventana de ayuda ya ha sido abierta, no se abra de nuevo. Para ello recurriríamos a un patrón Singleton, que aseguraría la existencia de una única ventana de ayuda en todo momento.

Singleton



- **Cliente**: Parte del programa que utilizará las fábricas y productos. Podría ser el archivo **.fla** principal, por ejemplo.
- **Singleton**: Clase que se quiere instanciar una sola vez. Se corresponde con la clase **VentanaAyuda** de nuestro ejemplo.

Singleton

- La forma de implementar el patrón Singleton en este ejemplo sería dotando a la clase VentanaAyuda de un método estático *getInstancia():VentanaAyuda* que comprobase si ya existe una instancia VentanaAyuda o si, por el contrario, se debe crear.

```
new VentanaAyuda(); // Se devolvería un error y no se instanciaría la ventana
VentanaAyuda.getInstancia(); // Se devolvería una nueva instancia de VentanaAyuda
VentanaAyuda.getInstancia(); // Se devolvería la instancia ya existente que se creó en la linea anterior
```

Referencias

- http://es.wikipedia.org/wiki/Patrón_de_diseño
- http://es.slideshare.net/faustol/patrones-creacionales
- http://harrybarrera.wordpress.com/2010/05/01/patrones-de-creacion-–-fabrica-abstracta-y-singleton/
- http://es.wikipedia.org/wiki/Abstract_Factory_%28patrón_de_diseño%29
- http://es.slideshare.net/ikercanarias/patrones-de-diseo-de-software-14836338
- http://www.cristalab.com/tutoriales/patrones-de-diseno-creacionales-c99932l/