**Patrón de diseño**

Los ***patrones de diseño*** son **la base para la búsqueda de soluciones a problemas comunes en el desarrollo de**[**software**](https://es.wikipedia.org/wiki/Software) y otros ámbitos referentes al diseño de interacción o interfaces.

**características**.

* Comprobar su **efectividad** resolviendo problemas similares en ocasiones anteriores.
* **Reutilizable**, aplicable a diferentes problemas de diseño en distintas circunstancias

## Breve reseña histórica

Más tarde, en 1987, [Ward Cunningham](https://es.wikipedia.org/wiki/Ward_Cunningham) y [Kent Beck](https://es.wikipedia.org/wiki/Kent_Beck), sobrepasados por el pobre entrenamiento que recibían los nuevos programadores en orientación a objetos, se preguntaban cómo se podían capturar las buenas ideas para, luego de alguna manera, traspasarlas a los nuevos programadores recién instruidos en herencia y polimorfismo.

se dieron cuenta del paralelo que existía entre la buena arquitectura propuesta por el [arquitecto](https://es.wikipedia.org/wiki/Arquitecto) [Christopher Alexander](https://es.wikipedia.org/wiki/Christopher_Alexander) (aportó al mundo de la arquitectura el libro *The Timeless Way of Building)* y la buena arquitectura OO, de modo que usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI) y publicaron un artículo en OOPSLA-87 titulado **Using Pattern Languages for OO Programs**.

En 1990 cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro *[Design Patterns](https://es.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&redlink=1" \o "Design Patterns (aún no redactado))* escrito por el grupo [Gang of Four](https://es.wikipedia.org/w/index.php?title=Gang_of_Four_(dise%C3%B1o)&action=edit&redlink=1" \o "Gang of Four (diseño) (aún no redactado)) (**GoF**) compuesto por [Erich Gamma](https://es.wikipedia.org/wiki/Erich_Gamma), [Richard Helm](https://es.wikipedia.org/w/index.php?title=Richard_Helm&action=edit&redlink=1), [Ralph Johnson](https://es.wikipedia.org/w/index.php?title=Ralph_Johnson_(inform%C3%A1tica)&action=edit&redlink=1) y [John Vlisides](https://es.wikipedia.org/w/index.php?title=John_Vlisides&action=edit&redlink=1), en el que se recogían 23 patrones de diseño comunes.

## Objetivos de los patrones

* **Proporcionar** catálogos de **elementos reusables** en el diseño de sistemas software.
* **Evitar la reiteración** en la búsqueda **de soluciones a problemas** ya conocidos y solucionados anteriormente.
* **Formalizar un vocabulario común** entre diseñadores.
* **Estandarizar el modo en que se realiza el diseño.**
* **Facilitar el aprendizaje** de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

no pretenden:

* **Imponer ciertas alternativas** de diseño frente a otras.
* **Eliminar la creatividad inherente** al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. "Abusar o forzar el uso de los patrones puede ser un error".

## Categorías de patrones

Según la escala o nivel de abstracción:

* [**Patrones de arquitectura**](https://es.wikipedia.org/wiki/Patrones_de_arquitectura): Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
* **Patrones de diseño**: Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
* **Dialectos**: Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

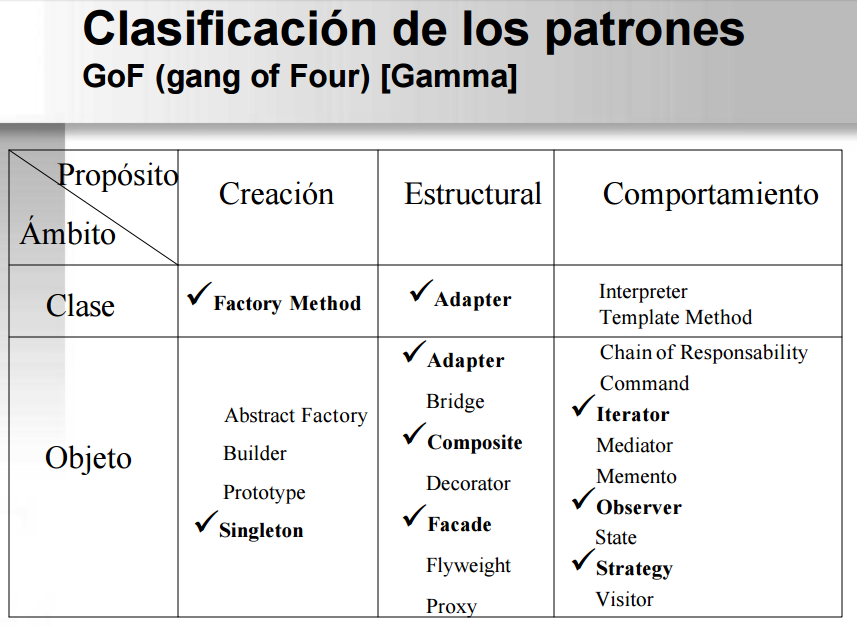
El "[antipatrón de diseño](https://es.wikipedia.org/wiki/Antipatr%C3%B3n_de_dise%C3%B1o" \o "Antipatrón de diseño)", que con forma semejante a la de un patrón, intenta prevenir contra errores comunes de diseño en el software. La idea de los antipatrones es dar a conocer los problemas que acarrean ciertos diseños muy frecuentes, para intentar evitar que diferentes sistemas acaben una y otra vez en el mismo callejón sin salida por haber cometido los mismos errores.

## Estructuras o plantillas de patrones

Para describir un patrón se usan plantillas más o menos estandarizadas, La plantilla más común es la utilizada precisamente por el [GoF](https://es.wikipedia.org/w/index.php?title=Gang_of_Four_(dise%C3%B1o)&action=edit&redlink=1" \o "Gang of Four (diseño) (aún no redactado))

* **Nombre del patrón**: nombre estándar del patrón por el cual será reconocido en la comunidad (normalmente se expresan en inglés).
* **Clasificación del patrón**: creacional, estructural o de comportamiento.
* **Intención**: ¿Qué problema pretende resolver el patrón?
* **También conocido como**: Otros nombres de uso común para el patrón.
* **Motivación**: Escenario de ejemplo para la aplicación del patrón.
* **Aplicabilidad**: Usos comunes y criterios de aplicabilidad del patrón.
* **Estructura**: Diagramas de clases oportunos para describir las clases que intervienen en el patrón.
* **Participantes**: Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón.
* **Colaboraciones**: Explicación de las interrelaciones que se dan entre los participantes.
* **Consecuencias**: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del patrón.
* **Implementación**: Técnicas o comentarios oportunos de cara a la implementación del patrón.
* **Código de ejemplo**: Código fuente ejemplo de implementación del patrón.
* **Usos conocidos**: Ejemplos de sistemas reales que usan el patrón.
* **Patrones relacionados**: Referencias cruzadas con otros patrones.

## Relación de principales patrones *GoF* (*Gang Of Four*)



* [**Abstract Factory**](https://es.wikipedia.org/wiki/Abstract_Factory_(patr%C3%B3n_de_dise%C3%B1o)) (fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo, la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).
* [**Builder**](https://es.wikipedia.org/wiki/Builder_(patr%C3%B3n_de_dise%C3%B1o)) (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
* [**Factory Method**](https://es.wikipedia.org/wiki/Factory_Method_(patr%C3%B3n_de_dise%C3%B1o)) (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística, es decir, la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear. Parte del principio de que las subclases determinan la clase a implementar. A continuación se muestra un ejemplo de este patrón:

[**Abstract Factory**](https://es.wikipedia.org/wiki/Abstract_Factory_(patr%C3%B3n_de_dise%C3%B1o))**(fábrica abstracta):**

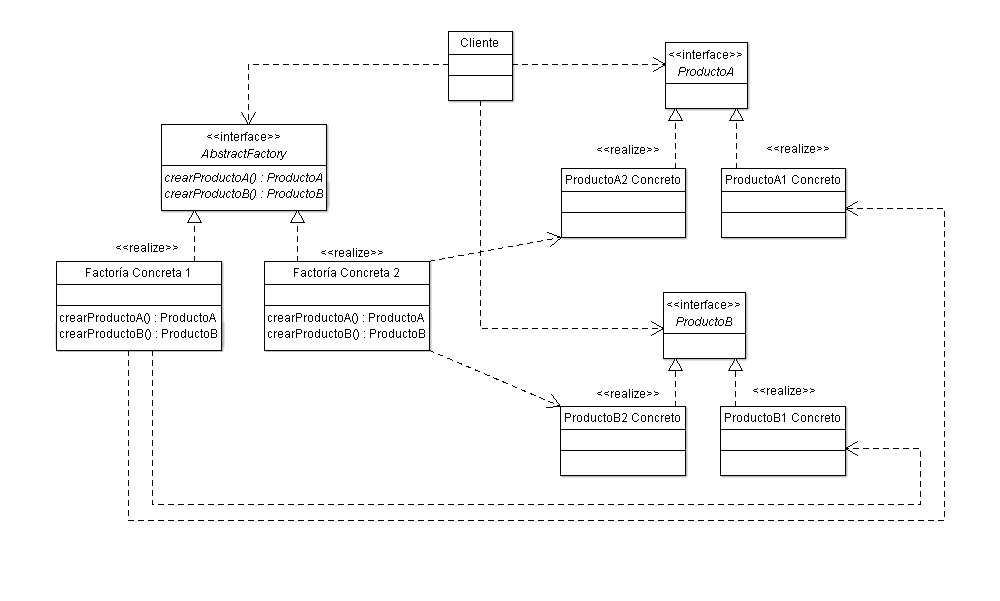
## Contexto y problema

Contexto: Debemos **crear diferentes objetos, todos pertenecientes a la misma familia**. Por ejemplo: las bibliotecas **para crear interfaces gráficas** suelen utilizar este patrón y cada familia sería un sistema operativo distinto. Así pues, el usuario declara un Botón, pero de forma más interna lo que está creando es un BotónWindows o un BotónLinux.

**El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos.**

**El patrón Abstract Factory está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas.**

## Aspecto estático

[](https://commons.wikimedia.org/wiki/File:Diagrama_Abstract_Factory.png)

**Estructura**

* **Cliente**: La clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría, es decir, Cliente lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB).
* **AbstractFactory**: Es la definición de la interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear. ("crearProductoA()" y "crearProductoB()")
* **Factorías Concretas**: Estas son las diferentes familias de productos. **Provee de la instancia concreta de la que se encarga de crear**. De esta forma podemos tener una factoría que cree los elementos gráficos para Windows y otra que los cree para Linux, pudiendo poner fácilmente (creando una nueva) otra que los cree para MacOS, por ejemplo.
* **Producto abstracto**: Definición de las interfaces para la familia de productos *genéricos*. En el diagrama son "ProductoA" y "ProductoB". En un ejemplo de interfaces gráficas podrían ser todos los elementos: Botón, Ventana, Cuadro de Texto, Combo... El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
* **Producto concreto**: Implementación de los diferentes productos. Podría ser por ejemplo "BotónWindows" y "BotónLinux". Como ambos implementan "Botón" el cliente no sabrá si está en Windows o Linux, puesto que trabajará directamente sobre la superclase o interfaz.

## Un ejemplo

Supongamos que disponemos de una cadena de pizzerías. Para crear pizzas disponemos de un método abstracto en la clase Pizzería que será implementada por cada subclase de Pizzería.

**abstract** Pizza crearPizza()

Concretamente se creará una clase PizzeríaZona por cada zona, por ejemplo la Pizzería de New York sería PizzeriaNewYork y la de Californía PizzeríaCalifornia que implementarán el método con los ingredientes de sus zonas.

Las pizzas son diferentes según las zonas. No es igual la pizza de New York que la pizza de California. Igualmente, aunque usarán los mismos ingredientes (tomate, mozzarella...) no los obtendrán del mismo lugar, cada zona los comprará donde lo tenga más cerca. Así pues podemos crear un método creador de Pizza que sea

Pizza(FactoriaIngredientes fi);

Como vemos utilizamos la factoría abstracta (no las concretas de cada zona, como podría ser IngredientesNewYork o IngredientesCalifornia). Pizza podrá obtener los ingredientes de la factoría independientemente de donde sea. Sería fácil crear nuevas factorías y añadirlas al sistema para crear pizzas con estos nuevos ingredientes. Efectivamente, en este ejemplo *cliente* es Pizza y es independiente de la Factoría usada.

El creador de la Pizza será el encargado de instanciar la factoría concreta, así pues los encargados de instanciar las factorías concretas serán las pizzerías locales. En PizzeríaNewYork podemos tener el método crearPizza() que realice el siguiente trabajo:

Pizza crearPizza() {

FactoríaIngredientes fi = **new** IngredientesNewYork();

Pizza pizza = **new** Pizza(fi); *// Uso de la factoría*

pizza.cortar();

pizza.empaquetar();

**return** pizza;

}

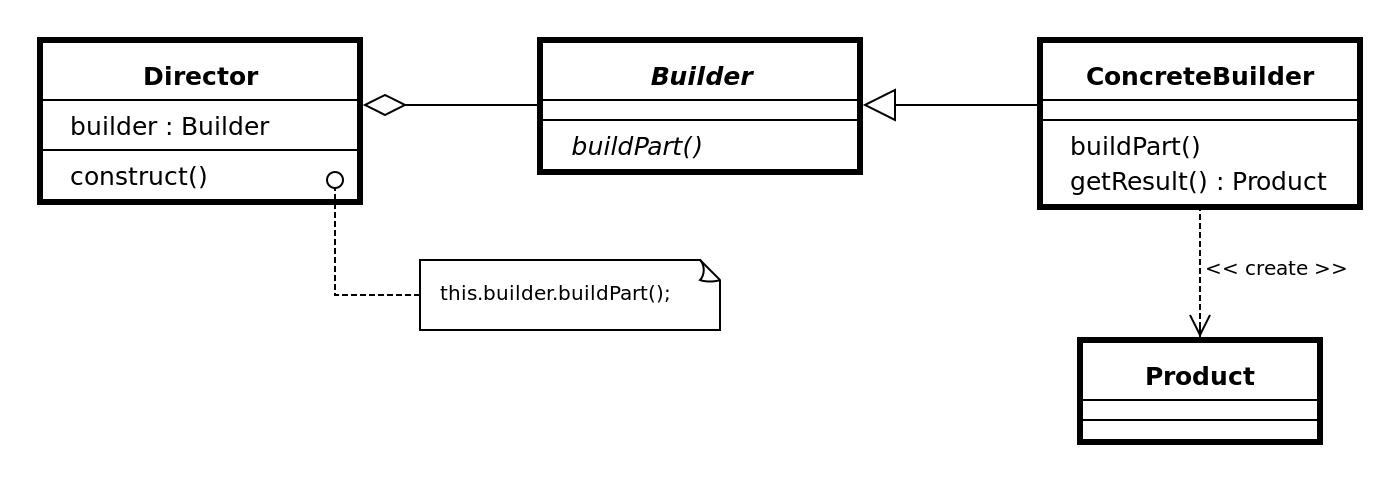
Como conclusión podemos observar que gracias a la factoría de ingredientes crear una nueva zona, por ejemplo una pizzería en Barcelona, no nos implicaría estar modificando el código existente, **solo deberemos extenderlo** (uno de los pilares de la [Ingeniería del software](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_del_software)) ya crearíamos la subclase de Pizzería: PizzeríaBarcelona que al instanciar la factoría solo debería escoger la factoría de Barcelona. Obviamente se debería crear la factoría de Barcelona que se encargaría de crear los productos obtenidos de Barcelona. Así que en ningún momento modificamos las pizzerías existentes, la superclase pizzería o las otras factorías o productos, solo creamos nuevas clases.

# Builder (patrón de diseño)

el patrón builder (Constructor) es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.

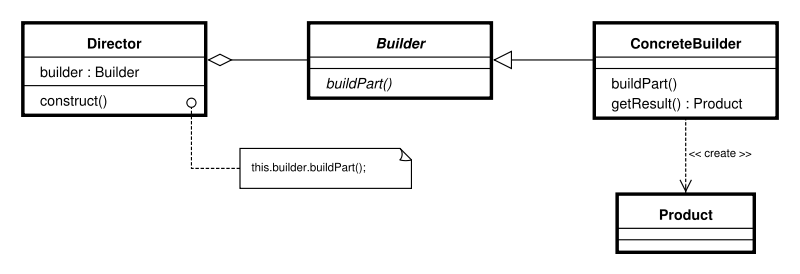
A menudo, el **patrón builder** construye el [patrón Composite](https://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%B1o)), un patrón estructural.

**Intención**: Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.



## Diagrama de Clases

* Builder
  + interfaz abstracta para crear productos.
* Concrete Builder
  + implementación del Builder
  + construye y reúne las partes necesarias para construir los productos
* Director
  + construye un objeto usando el patrón Builder
* Producto
  + El objeto complejo bajo construcción

[](https://commons.wikimedia.org/wiki/File:Builder_UML_class_diagram.svg)

## Ventajas

• Reduce el acoplamiento.

• Permite variar la representación interna de estructuras compleja, respetando la interfaz común de la clase Builder.

• Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del Builder.  
• Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.

• Distintos Director con distintas utilidades (visores, parsers, etc) pueden utilizar el mismo ConcreteBuilder.  
• Permite un mayor control en el proceso de creación del objeto. El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.

## Ejemplos

### Java

*/\*\* "Producto" \*/*

**class** **Pizza** {

**private** String masa = "";

**private** String salsa = "";

**private** String relleno = "";

**public** void setMasa(String masa) { **this**.masa = masa; }

**public** void setSalsa(String salsa) { **this**.salsa = salsa; }

**public** void setRelleno(String relleno) { **this**.relleno = relleno; }

}

*/\*\* "Abstract Builder" \*/*

**abstract** **class** **PizzaBuilder** {

**protected** Pizza pizza;

**public** Pizza getPizza() { **return** pizza; }

**public** void crearNuevaPizza() { pizza = **new** Pizza(); }

**public** **abstract** void buildMasa();

**public** **abstract** void buildSalsa();

**public** **abstract** void buildRelleno();

}

*/\*\* "ConcreteBuilder" \*/*

**class** **HawaiPizzaBuilder** **extends** PizzaBuilder {

**public** void buildMasa() { pizza.setMasa("suave"); }

**public** void buildSalsa() { pizza.setSalsa("dulce"); }

**public** void buildRelleno() { pizza.setRelleno("chorizo+alcachofas"); }

}

*/\*\* "ConcreteBuilder" \*/*

**class** **PicantePizzaBuilder** **extends** PizzaBuilder {

**public** void buildMasa() { pizza.setMasa("cocida"); }

**public** void buildSalsa() { pizza.setSalsa("picante"); }

**public** void buildRelleno() { pizza.setRelleno("pimienta+salchichón"); }

}

*/\*\* "Director" \*/*

**class** **Cocina** {

**private** PizzaBuilder pizzaBuilder;

**public** void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }

**public** Pizza getPizza() { **return** pizzaBuilder.getPizza(); }

**public** void construirPizza() {

pizzaBuilder.crearNuevaPizza();

pizzaBuilder.buildMasa();

pizzaBuilder.buildSalsa();

pizzaBuilder.buildRelleno();

}

}

*/\*\* Un cliente pidiendo una pizza. \*/*

**class** **BuilderExample** {

**public** **static** void main(String[] args) {

Cocina cocina = **new** Cocina();

PizzaBuilder hawai\_pizzabuilder = **new** HawaiPizzaBuilder();

PizzaBuilder picante\_pizzabuilder = **new** PicantePizzaBuilder();

cocina.setPizzaBuilder( hawai\_pizzabuilder );

cocina.construirPizza();

Pizza pizza = cocina.getPizza();

}

}

*/\*\**

*\* 2da opción para el abstract builder quizá más transparente para su uso.*

*\* Dentro del crear se llaman los métodos build.*

*\* Es válido siempre y cuando no se necesite alterar*

*\* el orden del llamado a los "build's".*

*\*/*

**abstract** **class** **OtroPizzaBuilder** {

**protected** Pizza pizza;

**public** Pizza getPizza() { **return** pizza; }

**public** void crearNuevaPizza() {

pizza = **new** Pizza();

buildMasa();

buildSalsa();

buildRelleno();

}

**public** **abstract** void buildMasa();

**public** **abstract** void buildSalsa();

**public** **abstract** void buildRelleno();

}

*/\*\* "Director" \*/*

**class** **OtraCocina** {

**private** OtroPizzaBuilder pizzaBuilder;

**public** void setPizzaBuilder(OtroPizzaBuilder pb) { pizzaBuilder = pb; }

**public** Pizza getPizza() { **return** pizzaBuilder.getPizza(); }

**public** void construirPizza() {

pizzaBuilder.crearNuevaPizza();

*//notar que no se necesita llamar a cada build.*

}

}

[Otro ejemplo sobre este patrón]

**public** **class** **BuildPattern** {

**public** void main(String args[]){

MakeAMovie makeAMovie = **new** MakeAMovie(**new** Rambo());

MakeAMovie.filmAMovie();

Movie movie = makeAMovie.seeMovie();

System.out.println(movie.genere);

System.out.println(movie.name);

System.out.println(movie.duration);

}

*/\*\**

*\* first step we need the movie*

*\**

*\*/*

**public** **class** **Movie**{

**public** String name = "";

**public** String genere = "";

**public** int duration;

**public** void setName(String name){

**this**.name=name;

}

**public** void setGenere(String genere){

**this**.genere=genere;

}

**public** void setDuration(int duration){

**this**.duration=duration;

}

}

*/\*\**

*\* Second step we need a studio to make the movie*

*\**

*\*/*

**abstract** **class** **BuilderMovieStudioAbstract**{

Movie movie;

**abstract** void buildName();

**abstract** void buildGenere();

**abstract** void buildDuration();

Movie getMovie(){**return** movie;}

}

*/\*\**

*\* third step we need an idea to film that movie*

*\**

*\*/*

**public** **class** **Rambo** **extends** BuilderMovieStudioAbstract{

void buildDuration() {

movie.setDuration(120);

}

void buildGenere() {

movie.setGenere("Action");

}

void buildName() {

movie.setName("Rambo");

}

}

*/\*\**

*\* Last Step we make the movie.*

*\**

*\*/*

**public** **class** **MakeAMovie**{

BuilderMovieStudioAbstract abstractMovie;

**public** MakeAMovie(BuilderMovieStudioAbstract abstractMovie) {

**this**.abstractMovie=abstractMovie;

}

**public** void filmAMovie(){

abstractMovie.buildDuration();

abstractMovie.buildName();

abstractMovie.buildGenere();

}

**public** Movie seeMovie(){

**return** abstractMovie.getMovie();

}

}

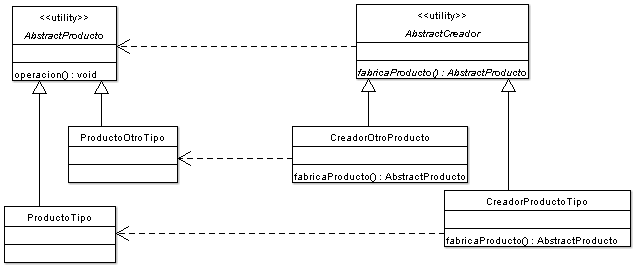
}

# Factory Method (patrón de diseño)

Consiste en utilizar una clase constructora (al estilo del Abstract Factory) abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado. Es una simplificación del [Abstract Factory](https://es.wikipedia.org/wiki/Abstract_Factory_(patr%C3%B3n_de_dise%C3%B1o)" \o "Abstract Factory (patrón de diseño)), en la que la [clase abstracta](https://es.wikipedia.org/wiki/Clase_abstracta) tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

## Estructura

Las clases principales en este patrón son el *creador* y el *producto*. El creador necesita crear instancias de productos, pero el tipo concreto de producto no debe ser forzado en las subclases del creador, porque las posibles subclases del creador deben poder especificar subclases del producto para utilizar.

[](https://commons.wikimedia.org/wiki/File:Factory_Method.png)

La solución para esto es hacer un método abstracto (el método de la fábrica) que se define en el creador. Este método abstracto se define para que devuelva un producto. Las subclases del creador pueden sobrescribir este método para devolver subclases apropiadas del producto...

## Ejemplo de código

**abstract** **class** **Creator**{

*// Definimos método abstracto*

**public** **abstract** Product factoryMethod();

}

Ahora definimos el creador concreto:

**public** **class** **ConcreteCreator** **extends** Creator{

**public** Product factoryMethod() {

**return** **new** ConcreteProduct();

}

}

Definimos el producto y su implementación concreta:

**public** **interface** **Product**{

**public** void operacion();

}

**public** **class** **ConcreteProduct** **implements** Product{

**public** void operacion(){

System.out.println("Una operación de este producto");

}

}

Ejemplo de uso:

**public** **static** void main(String args[]){

Creator aCreator;

aCreator = **new** ConcreteCreator();

Product producto = aCreator.factoryMethod();

producto.operacion();

}