**Factory Method**

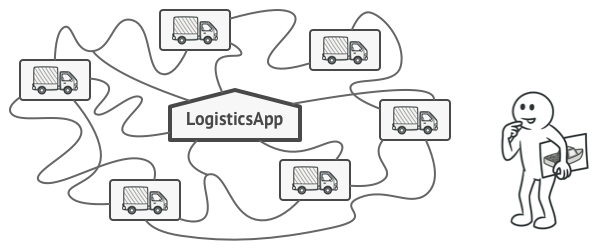
**Intención**

Es un patrón de diseño creativo que proporciona una interfaz para crear objetos en la superclase, pero permite que las subclases alteren el tipo de objetos que se crearán.

**Problema**

Imagina que estás creando una aplicación de gestión logística. La primera versión de su aplicación puede manejar solo el transporte en camiones, por lo que la mayor parte de su código vive en una clase Camión.

Después de un tiempo, su aplicación se vuelve tan popular que recibe toneladas de solicitudes para incluir también el transporte marítimo.



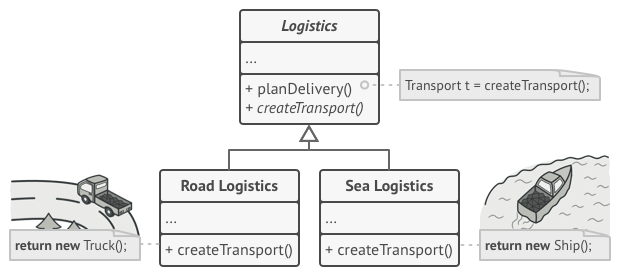
Buenas noticias, ¿verdad? Pero ¿qué hay del código? Parece que la mayoría de su código está acoplado a la clase Camión. Para agregar Barcos sería necesario realizar cambios en la base de código completa. Además, si decide agregar otro tipo de transporte a la aplicación, probablemente necesitará hacer todos esos cambios nuevamente.

Terminará con un código desagradable lleno de condicionales, que seleccionan comportamientos según las clases de objetos de transporte.

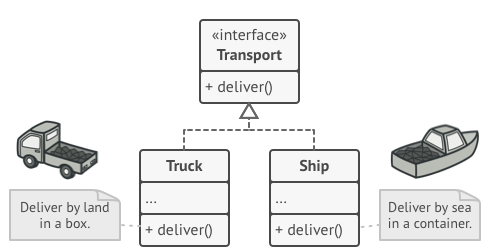
**Solución**

El patrón de Método de fábrica sugiere reemplazar la creación directa de objetos (mediante un new operador) con una llamada a un método especial de "fábrica". La llamada al constructor debe moverse dentro de ese método. Los objetos devueltos por métodos de fábrica a menudo se denominan "productos".

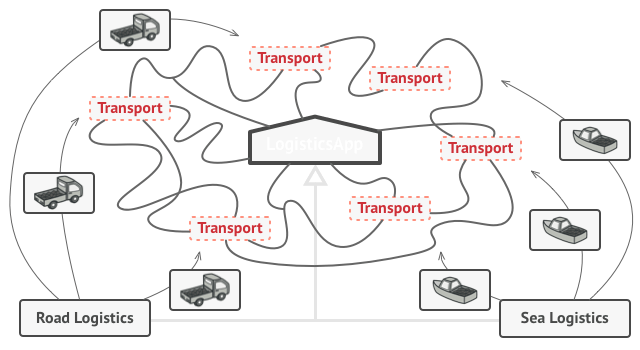
A primera vista, tal movimiento puede parecer inútil. Pero ahora puede anular el método de fábrica en una subclase y cambiar la clase de un objeto que se creará. Vamos a ver cómo funciona:



Por supuesto, hay una ligera limitación: todos los productos deben tener una interfaz común (por ejemplo, Transport). El método de fábrica en una clase base debería devolver esta interfaz común.

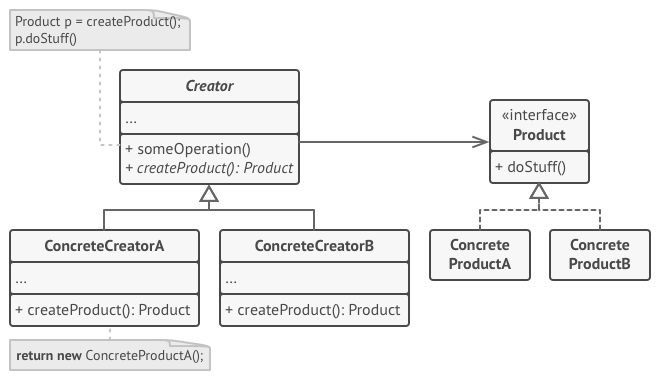


Las subclases pueden devolver productos concretos diferentes siempre que estos productos tengan una clase base o interfaz común (por ejemplo, ambos Camión y Barco implementen la Transport interfaz).



Los clientes de un método de fábrica no se preocupan por el tipo particular de un producto que reciben. Trabajan con todos los productos utilizando una interfaz de producto común.

**Estructura**



1. **El producto** declara la interfaz única para todos los objetos que pueden ser producidos por el creador y sus subclases.
2. **Los productos concretos** son las diferentes implementaciones de la interfaz del producto.

Los creadores concretos crearán y devolverán instancias de estas clases.

1. **El creador** declara un método de fábrica que devuelve el tipo de producto. Este método puede ser abstracto o tener alguna implementación predeterminada. En el primer caso, todos los creadores concretos **deben** implementar sus métodos de fábrica.

A pesar del nombre, en el mundo real, la creación del producto no es la responsabilidad principal de una clase Creator . Por lo general, tiene alguna lógica de negocios central que funciona con Productos.

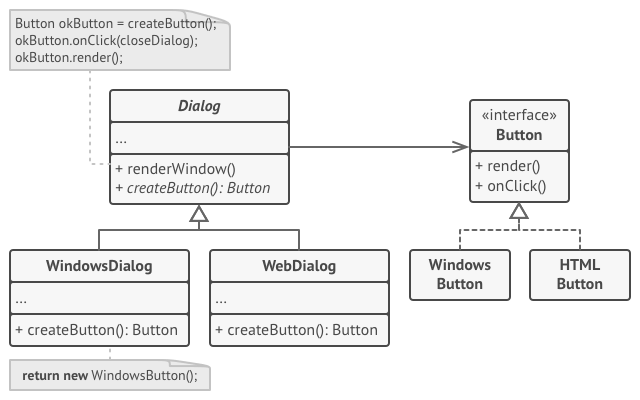
Aquí está la analogía: una gran compañía de desarrollo de software puede tener un departamento de capacitación para programadores. Pero la función principal de la empresa todavía es escribir código.

1. **Los creadores concretos** implementan o anulan el método de fábrica base, al crear y devolver uno de los productos concretos.

Tenga en cuenta que un método de fábrica no tiene que **crear** nuevas instancias todo el tiempo. También puede devolver objetos existentes desde algún caché, etc.

## **# Pseudocódigo**

Este ejemplo ilustra cómo se puede utilizar Factory Method para crear elementos de UI multiplataforma. El método de fábrica se declara en la clase de diálogo UI base. Devuelve botones abstractos. Las subclases de diálogo anulan el método de fábrica al devolver variaciones particulares de botones.



El resultado se utiliza en el código del diálogo base para componer una ventana de IU. El diálogo funciona con botones a través de su interfaz común. Por lo tanto, independientemente del tipo de botones que devuelva el método de fábrica, el código de diálogo permanece funcional.

Por lo tanto, el patrón de Método de fábrica hace que el código primario de una clase sea independiente de las clases concretas de productos que utiliza. Factory Method hace a las subclases responsables de elegir clases concretas de productos que se crearán.



💡 Aplicabilidad

🐞 Cuando no conoce los tipos y dependencias exactos de los objetos, su código debería funcionar.

Por ejemplo, leer y escribir datos de múltiples fuentes: sistema de archivos, base de datos o red. Todas estas fuentes tendrían diferentes tipos, dependencias y código de inicialización.

**⚡** El método de fábrica puede ocultar los detalles de implementación de un producto del otro código. Para admitir un nuevo tipo de producto, solo tendrá que crear una nueva subclase y anular el método de fábrica.

🐞 **Cuando desee proporcionar a los usuarios de su biblioteca o marco una forma de ampliar sus componentes internos.**

**⚡** Un usuario puede subclasificar fácilmente algún componente en particular. Pero, ¿cómo reconocería el marco esa subclase y trabajaría con él en lugar de un componente estándar? Un usuario tendrá que anular todos los métodos que crean instancias de componentes estándar y cambiarlos para crear objetos de una subclase personalizada. Eso es bastante incómodo, ¿no?

La mejor solución sería no solo dar a un usuario medios para extender clases particulares, sino también reducir el código que produce componentes a un solo método creativo. En otras palabras, para proporcionar los métodos de fábrica.

Vamos a ver cómo funcionaría. Imagina que escribes una aplicación utilizando un marco de UI de código abierto. Su aplicación debe tener botones redondos, pero el marco solo proporciona cuadrados.

Lo primero que haces es implementar la RoundButtonclase. Pero ahora necesita decirle a la UIFrameworkclase principal que use la nueva clase de botón en lugar de una predeterminada.

Para lograr esto, crea una subclase a UIWithRoundButtonspartir de una clase de marco base y reemplaza su método createButton. Este método aún devuelve Buttonobjetos en una clase base, pero su nueva subclase produce RoundButtonsobjetos. Ahora, en tu aplicación, debes inicializar el framework usando UIWithRoundButtonsclass en lugar de UIFramework. ¡Y eso es todo!

🐞 **Cuando desee guardar recursos del sistema y reutilizar objetos existentes, en lugar de reconstruir otros nuevos.**

**Por ejemplo, cuando se trata de objetos grandes o que requieren un uso intensivo de recursos, como conexiones de base de datos, etc.**

**⚡** Imagine cuánto se debe hacer para reutilizar los objetos existentes:

1. Primero, deberá crear una agrupación para mantener los objetos existentes.
2. Cuando alguien solicita un objeto, buscará un objeto libre dentro de ese grupo.
3. ... y devolverlo al código del cliente.
4. Solo si no hay objetos libres, creará uno nuevo (y lo agregará al grupo).

Este código debe ser colocado en algún lugar. El lugar más conveniente es un constructor. De esta manera, todas esas comprobaciones se ejecutarán siempre que alguien intente crear un objeto. Pero, por desgracia, los constructores **deben** devolver nuevos objetos por definición, por lo que no pueden devolver instancias existentes.

Por otro lado, el código de cliente, que utiliza sus objetos, tampoco puede contenerlo. De lo contrario, sabría demasiados detalles de implementación de su clase. Por lo tanto, necesita tener un método único para encapsular esta lógica. Eso sería un método de fábrica.

**☑ Como Implementar**

1. Extraer la interfaz común para todos los productos. Esta interfaz debe declarar métodos que tengan sentido para cada producto.
2. Añade un método de fábrica vacío dentro de la clase creadora. Su firma debe devolver el tipo de interfaz del producto.
3. Revise el código del creador y encuentre todas las referencias a los constructores de productos. Una por una, reemplácelas con llamadas al método de fábrica, pero extraiga el código de creación del producto al método de fábrica.

Es posible que deba agregar un parámetro temporal al método de fábrica que se usará para controlar qué producto se creará.

En este punto, el código del método de fábrica puede parecer bastante feo. Puede tener un operador de conmutador grande que elija qué clase de producto se instanciará. Pero no te preocupes, lo arreglaremos de inmediato.

1. Ahora, anule el método de fábrica en las subclases y mueva allí el caso apropiado del operador del interruptor en el método base.
2. El parámetro de control usado en la clase del creador base también se puede usar en subclases.

Por ejemplo, es posible que tenga la jerarquía de un creador con una clase base Maily las clases Airy Ground, además de las clases de productos: Plane, Trucky Train. Aircoincide Planebien, pero Groundcoincide con ambos Trucky Trainal mismo tiempo. Puede crear una nueva subclase para manejar ambos casos, pero hay otra opción. El código del cliente puede pasar un argumento al método de fábrica de la Groundclase para controlar qué producto recibe.

1. Si el método de fábrica base se ha quedado vacío después de todos los movimientos, puede hacerlo abstracto.

⚖ Pros y contras

**Pros**

* Sigue el *Principio Abierto / Cerrado* .
* Evita el acoplamiento apretado entre los productos concretos y el código que los utiliza.
* Simplifica el código debido a mover todo el código creativo a un solo lugar.
* Simplifica la adición de nuevos productos al programa.

**Contras**

Requiere subclases adicionales.

#### **buttons/Button.java:** Common product interface

**package** refactoring\_guru.factory\_method.example.buttons;

*/\*\**

*\* Common interface for all buttons.*

*\*/*

**public** **interface** **Button** {

**public** **void** render();

**public** **void** onClick();

}

#### **buttons/HtmlButton.java:** Concrete product

**package** refactoring\_guru.factory\_method.example.buttons;

*/\*\**

*\* HTML button implementation.*

*\*/*

**public** **class** **HtmlButton** **implements** Button {

**public** **void** render() {

System.out.println("<button>Test Button</button>");

onClick();

}

**public** **void** onClick() {

System.out.println("Click! Button says - 'Hello World!'");

}

}

#### **buttons/WindowsButton.java:** One more concrete product

**package** refactoring\_guru.factory\_method.example.buttons;

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

*/\*\**

*\* Windows button implementation.*

*\*/*

**public** **class** **WindowsButton** **implements** Button {

JPanel panel = **new** JPanel();

JFrame frame = **new** JFrame();

JButton button;

**public** **void** render() {

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JLabel label = **new** JLabel("Hello World!");

label.setOpaque(**true**);

label.setBackground(**new** Color(235, 233, 126));

label.setFont(**new** Font("Dialog", Font.BOLD, 44));

label.setHorizontalAlignment(SwingConstants.CENTER);

panel.setLayout(**new** FlowLayout(FlowLayout.CENTER));

frame.getContentPane().add(panel);

panel.add(label);

onClick();

panel.add(button);

frame.setSize(320, 200);

frame.setVisible(**true**);

onClick();

}

**public** **void** onClick() {

button = **new** JButton("Exit");

button.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

frame.setVisible(**false**);

System.exit(0);

}

});

}

}

## **factory**

#### **factory/Dialog.java:** Base creator

**package** refactoring\_guru.factory\_method.example.factory;

**import** refactoring\_guru.factory\_method.example.buttons.Button;

*/\*\**

*\* Base factory class. Note that "factory" is merely a role for the class. It*

*\* should have some core business logic which needs different products to be*

*\* created.*

*\*/*

**public** **abstract** **class** **Dialog** {

**public** **void** renderWindow() {

*// ... other code ...*

Button okButton = createButton();

okButton.render();

}

*/\*\**

*\* Subclasses will override this method in order to create specific button*

*\* objects.*

*\*/*

**public** **abstract** Button createButton();

}

#### **factory/HtmlDialog.java:** Concrete creator

**package** refactoring\_guru.factory\_method.example.factory;

**import** refactoring\_guru.factory\_method.example.buttons.Button;

**import** refactoring\_guru.factory\_method.example.buttons.HtmlButton;

*/\*\**

*\* HTML Dialog will produce HTML buttons.*

*\*/*

**public** **class** **HtmlDialog** **extends** Dialog {

**@Override**

**public** Button createButton() {

**return** **new** HtmlButton();

}

}

#### **factory/WindowsDialog.java:** One more concrete creator

**package** refactoring\_guru.factory\_method.example.factory;

**import** refactoring\_guru.factory\_method.example.buttons.Button;

**import** refactoring\_guru.factory\_method.example.buttons.WindowsButton;

*/\*\**

*\* Windows Dialog will produce Windows buttons.*

*\*/*

**public** **class** **WindowsDialog** **extends** Dialog {

**@Override**

**public** Button createButton() {

**return** **new** WindowsButton();

}

}

#### **Demo.java:** Client code

**package** refactoring\_guru.factory\_method.example;

**import** refactoring\_guru.factory\_method.example.factory.Dialog;

**import** refactoring\_guru.factory\_method.example.factory.HtmlDialog;

**import** refactoring\_guru.factory\_method.example.factory.WindowsDialog;

*/\*\**

*\* Demo class. Everything comes together here.*

*\*/*

**public** **class** **Demo** {

**private** **static** Dialog dialog;

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

configure();

runBusinessLogic();

}

*/\*\**

*\* The concrete factory is usually chosen depending on configuration or*

*\* environment options.*

*\*/*

**static** **void** configure() {

**if** (System.getProperty("os.name").equals("Windows 10")) {

dialog = **new** WindowsDialog();

} **else** {

dialog = **new** HtmlDialog();

}

}

*/\*\**

*\* All of the client code should work with factories and products through*

*\* abstract interfaces. This way it does not care which factory it works*

*\* with and what kind of product it returns.*

*\*/*

**static** **void** runBusinessLogic() {

dialog.renderWindow();

}

}

**Referencia**

https://refactoring.guru/design-patterns/factory-method