**Abstract Factory**

## **Intent**

**Abstract Factory** es un patrón de diseño creativo que le permite producir familias de objetos relacionados sin especificar sus clases concretas.

## **Problema**

Imagina que estás creando un simulador de una tienda de muebles. Su código consiste en:

1. Familia de productos relacionados, digamos: Chair+ Sofa+ CoffeeTable.
2. Varias variantes de esta familia. Por ejemplo, los productos Chair+ Sofa+ CoffeeTabledisponibles en las siguientes variantes: IKEA, VictorianStyle, ArtDeco.

Necesita una forma de crear objetos de muebles individuales para que coincidan con otros objetos de la misma familia. Los clientes se sienten bastante frustrados cuando reciben muebles que no coinciden.

Además, no desea cambiar el código existente al agregar nuevos productos o familias de productos al programa. Los proveedores de muebles actualizan sus catálogos muy a menudo, y no desea cambiar el código central cada vez que sucede.

## **Solución**

Lo primero que sugiere el patrón de Abstract Factory es revisar todos los distintos productos y forzar a sus variantes a seguir interfaces comunes. Por ejemplo, todas las variantes de silla deben seguir la Chairinterfaz; Todas las mesas de café deben implementar la CoffeeTableinterfaz, etc.

El segundo paso es crear la AbstractFactoryinterfaz base que declara los métodos para crear todos los productos que conforman una familia de productos (es decir createChair, createSofay createCoffeeTable). Lo importante aquí es hacer que estos métodos para devolver los tipos de productos abstractos representados por las interfaces hemos extraído previamente: Chair, Sofa, CoffeeTable, etc.

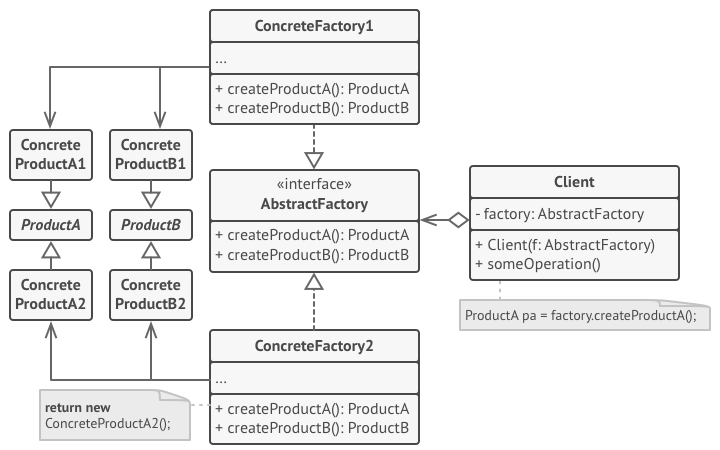
El tercer paso es implementar fábricas concretas. Las fábricas son clases que devuelven productos de un tipo particular. Por ejemplo IKEAFactory,, solo se devolverán IKEAChair, IKEASofay los IKEACoffeeTableobjetos. Todas las fábricas deben seguir la AbstractFactoryinterfaz mientras crean la misma variedad de productos.

El código del cliente tiene que trabajar con fábricas y productos solo a través de sus interfaces abstractas. De esta manera, puede modificar el tipo de productos utilizados en el código del cliente pasándole un objeto de fábrica diferente.

Entonces, cuando el código del cliente le pide a una fábrica que produzca una silla, no debe estar al tanto de la clase concreta de la fábrica. Tampoco debe ser consciente de la clase concreta de silla que obtendrá. Ya sea un modelo moderno de IKEA o una silla de estilo victoriano, debe funcionar con todas las sillas de la misma manera, utilizando la Chair interfaz abstracta. Lo que sabrá el código del cliente es que la silla resultante implementa el sit método, declarado en la interfaz. También sabe que cualquier silla que se devuelva, coincidirá con el tipo de sofá y mesa de centro que fabrica la misma fábrica.

Está bien, pero ¿quién crea los objetos reales de fábrica? Generalmente, el programa crea un objeto de fábrica concreto en la etapa de inicialización, y el tipo de fábrica se selecciona según la configuración o el entorno.

## **Estructura**



1. **Los productos abstractos** declaran interfaces para todos los productos distintos que conforman una familia de productos. Normalmente, debería haber varias interfaces de productos diferentes.
2. **Los productos concretos** implementan diferentes interfaces abstractas de productos. Los conjuntos de productos concretos implementados de manera similar representan diferentes variantes de una familia de productos.
3. **Abstract factory** declara la interfaz para crear todos los productos de una familia.
4. **Fábricas Concretas** implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta corresponde a la variante específica de una familia de productos.
5. Si bien las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben declarar los productos abstractos correspondientes como un tipo de devolución. De esta manera, el código de cliente que utiliza una fábrica no se acoplará a la variante específica de los productos. Podrá trabajar con cualquier variante de fábrica / producto utilizando las interfaces abstractas.

## **Pseudocódigo**

Este ejemplo ilustra cómo se puede usar el patrón de Abstract Factory para crear elementos de UI multiplataforma sin unir el código del cliente a clases de UI concretas.

El código del cliente solicita varios elementos de la interfaz de usuario de una fábrica. El tipo concreto de elementos devueltos depende del tipo de fábrica, que se pasa al código del cliente. El código del cliente trabaja con elementos utilizando interfaces abstractas. Mientras use el mismo objeto de fábrica, todos sus productos deben ser compatibles.

El patrón de diseño de Abstract Factory hace que el código del cliente sea independiente de las clases concretas de elementos de UI. Además, no tiene que tocar el código existente, cuando se solicita una nueva variación de la interfaz de usuario (por ejemplo, la implementación de elementos de la interfaz de usuario de Linux). Solo necesita crear una nueva subclase de fábrica que devuelva elementos de IU de un nuevo tipo.

// This pattern assumes that you have several families of

// products, structured into separate class hierarchies

// (Button/Checkbox). All products of the same family must

// follow the common interface.

**interface** **Button** **is**

**method** paint()

// All products families have the same varieties

// (macOS/Windows).

**class** **WinButton** **implements** Button **is**

**method** paint() **is**

// Render a button in Windows style

**class** **MacButton** **implements** Button **is**

**method** paint() **is**

// Render a button in macOS style

**interface** **Checkbox** **is**

**method** paint()

**class** **WinCheckbox** **implements** Checkbox **is**

**method** paint() **is**

// Render a checkbox in Windows style

**class** **MacCheckbox** **implements** Checkbox **is**

**method** paint() **is**

// Render a checkbox in macOS style

// Abstract factory knows about all (abstract) product types.

**interface** **GUIFactory** **is**

**method** createButton():Button

**method** createCheckbox():Checkbox

// Each concrete factory extends basic factory and responsible

// for creating products of a single variety.

**class** **WinFactory** **implements** GUIFactory **is**

**method** createButton():Button **is**

**return** **new** WinButton()

**method** createCheckbox():Checkbox **is**

**return** **new** WinCheckbox()

// Although concrete factories create the concrete products,

// they still return them with the abstract type. This fact

// makes factories interchangeable.

**class** **MacFactory** **implements** GUIFactory **is**

**method** createButton():Button **is**

**return** **new** MacButton()

**method** createCheckbox():Checkbox **is**

**return** **new** MacCheckbox()

// Factory users do not care which concrete factory they use

// since they work with// factories and products through

// abstract interfaces.

**class** **Application** **is**

**private** **field** button: Button

**constructor** Application(factory: GUIFactory) **is**

**this**.factory = factory

**method** createUI() **is**

**this**.button = factory.createButton()

**method** paint() **is**

button.paint()

// Application picks the factory type and creates it in run time

// (usually at initialization stage), depending on the

// configuration or environment variables.

**class** **ApplicationConfigurator** **is**

**method** main() **is**

config = readApplicationConfigFile()

**if** (config.OS == "Windows") **then**

factory = **new** WinFactory()

**else** **if** (config.OS == "Web") **then**

factory = **new** MacFactory()

**else**

throw **new** Exception("Error! Unknown operating system.")

Application app = **new** Application(factory)

## **Aplicabilidad**

**Cuando una lógica empresarial debe funcionar con diferentes variantes de productos de alguna familia de productos, pero no desea que dependa de clases de productos concretos (o si se desconocen de antemano).**

Abstract Factory oculta de la información del código del cliente sobre las clases de productos que crea. El código del cliente puede funcionar con cualquier producto creado por cualquier fábrica siempre que trabaje con ellos utilizando las interfaces abstractas.

**Cuando una clase tiene múltiples**[**Métodos de Fábrica**](https://refactoring.guru/design-patterns/factory-method)**que borran su principal responsabilidad.**

En un programa bien diseñado, *cada clase debe ser responsable de una sola cosa* . Cuando una clase trata con múltiples tipos de productos, puede valer la pena reemplazar varios métodos de fábrica con una fábrica abstracta independiente.

## **Cómo implementar**

1. Haga un mapa de una matriz de productos distintos frente a las variantes del mismo producto.
2. Cree interfaces abstractas para todos los tipos de productos distintos y haga que todos los productos concretos sigan estas interfaces.
3. Declara la interfaz de fábrica abstracta. Esta interfaz debe enumerar los métodos de creación para todos los tipos distintos de productos.
4. Implementar clases de fábrica separadas para cada variante de la familia de productos.
5. Cree un código de inicialización de fábrica en algún lugar del código del cliente. Debería elegir el tipo y crear una fábrica de concreto según la configuración o el entorno actual.
6. En el código del cliente, reemplace todas las llamadas del constructor del producto con llamadas al método de creación en el objeto de fábrica.

## **Pros y contras**

**Pros**

* Sigue el *Principio Abierto / Cerrado*.
* Permite construir familias de objetos de producto y garantiza su compatibilidad.
* Evita el acoplamiento apretado entre los productos concretos y el código que los utiliza.
* Divide las responsabilidades entre múltiples clases.

**Contras**

* Aumenta la complejidad general del código creando múltiples clases adicionales.

**Ejemplo en JAVA**

## **buttons:** First product hierarchy

#### **buttons/Button.java**

**package** refactoring\_guru.abstract\_factory.example.buttons;

*/\*\**

*\* Abstract Factory assumes that you have several families of products,*

*\* structured into separate class hierarchies (Button/Checkbox). All products of*

*\* the same family have the common interface.*

*\**

*\* This is the common interface for buttons family.*

*\*/*

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

**public** **abstract** **void** paint();

}

#### **buttons/MacOSButton.java**

**package** refactoring\_guru.abstract\_factory.example.buttons;

*/\*\**

*\* All products families have the same varieties (MacOS/Windows).*

*\**

*\* This is a MacOS variant of a button.*

*\*/*

**public** **class** **MacOSButton** **implements** Button {

**@Override**

**public** **void** paint() {

System.out.println("You have created MacOSButton.");

}

}

#### **buttons/WindowsButton.java**

**package** refactoring\_guru.abstract\_factory.example.buttons;

*/\*\**

*\* All products families have the same varieties (MacOS/Windows).*

*\**

*\* This is another variant of a button.*

*\*/*

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

**@Override**

**public** **void** paint() {

System.out.println("You have created WindowsButton.");

}

}

## **checkboxes:** Second product hierarchy

#### **checkboxes/Checkbox.java**

**package** refactoring\_guru.abstract\_factory.example.checkboxes;

*/\*\**

*\* Checkboxes is the second product family. It has the same variants as buttons.*

*\*/*

**public** **interface** **Checkbox** {

**public** **abstract** **void** paint();

}

#### **checkboxes/MacOSCheckbox.java**

**package** refactoring\_guru.abstract\_factory.example.checkboxes;

*/\*\**

*\* All products families have the same varieties (MacOS/Windows).*

*\**

*\* This is a variant of a checkbox.*

*\*/*

**public** **class** **MacOSCheckbox** **implements** Checkbox {

**@Override**

**public** **void** paint() {

System.out.println("You have created MacOSCheckbox.");

}

}

#### **checkboxes/WindowsCheckbox.java**

**package** refactoring\_guru.abstract\_factory.example.checkboxes;

*/\*\**

*\* All products families have the same varieties (MacOS/Windows).*

*\**

*\* This is another variant of a checkbox.*

*\*/*

**public** **class** **WindowsCheckbox** **implements** Checkbox {

**@Override**

**public** **void** paint() {

System.out.println("You have created WindowsCheckbox.");

}

}

## **factories**

#### **factories/GUIFactory.java:** Abstract factory

**package** refactoring\_guru.abstract\_factory.example.factories;

**import** refactoring\_guru.abstract\_factory.example.buttons.Button;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.Checkbox;

*/\*\**

*\* Abstract factory knows about all (abstract) product types.*

*\*/*

**public** **interface** **GUIFactory** {

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

**public** **abstract** Checkbox createCheckbox();

}

#### **factories/MacOSFactory.java:** Concrete factory (MacOS)

**package** refactoring\_guru.abstract\_factory.example.factories;

**import** refactoring\_guru.abstract\_factory.example.buttons.Button;

**import** refactoring\_guru.abstract\_factory.example.buttons.MacOSButton;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.Checkbox;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.MacOSCheckbox;

*/\*\**

*\* Each concrete factory extends basic factory and responsible for creating*

*\* products of a single variety.*

*\*/*

**public** **class** **MacOSFactory** **implements** GUIFactory {

**@Override**

**public** Button createButton() {

**return** **new** MacOSButton();

}

**@Override**

**public** Checkbox createCheckbox() {

**return** **new** MacOSCheckbox();

}

}

#### **factories/WindowsFactory.java:** Concrete factory (Windows)

**package** refactoring\_guru.abstract\_factory.example.factories;

**import** refactoring\_guru.abstract\_factory.example.buttons.Button;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.WindowsCheckbox;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.Checkbox;

**import** refactoring\_guru.abstract\_factory.example.buttons.WindowsButton;

*/\*\**

*\* Each concrete factory extends basic factory and responsible for creating*

*\* products of a single variety.*

*\*/*

**public** **class** **WindowsFactory** **implements** GUIFactory {

**@Override**

**public** Button createButton() {

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

}

**@Override**

**public** Checkbox createCheckbox() {

**return** **new** WindowsCheckbox();

}

}

## **app**

#### **app/Application.java:** Client code

**package** refactoring\_guru.abstract\_factory.example.app;

**import** refactoring\_guru.abstract\_factory.example.buttons.Button;

**import** refactoring\_guru.abstract\_factory.example.checkboxes.Checkbox;

**import** refactoring\_guru.abstract\_factory.example.factories.GUIFactory;

*/\*\**

*\* Factory users don't care which concrete factory they use since they work with*

*\* factories and products through abstract interfaces.*

*\*/*

**public** **class** **Application** {

**private** Button button;

**private** Checkbox checkbox;

**public** Application(GUIFactory factory) {

button = factory.createButton();

checkbox = factory.createCheckbox();

}

**public** **void** paint() {

button.paint();

checkbox.paint();

}

}

#### **Demo.java:** App configuration

**package** refactoring\_guru.abstract\_factory.example;

**import** refactoring\_guru.abstract\_factory.example.app.Application;

**import** refactoring\_guru.abstract\_factory.example.factories.GUIFactory;

**import** refactoring\_guru.abstract\_factory.example.factories.MacOSFactory;

**import** refactoring\_guru.abstract\_factory.example.factories.WindowsFactory;

*/\*\**

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

*\*/*

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

*/\*\**

*\* Application picks the factory type and creates it in run time (usually at*

*\* initialization stage), depending on the configuration or environment*

*\* variables.*

*\*/*

**private** **static** Application configureApplication() {

Application app;

GUIFactory factory;

**String** osName = System.getProperty("os.name").toLowerCase();

**if** (osName.contains("mac")) {

factory = **new** MacOSFactory();

app = **new** Application(factory);

} **else** {

factory = **new** WindowsFactory();

app = **new** Application(factory);

}

**return** app;

}

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

Application app = configureApplication();

app.paint();

}

}

**Referencia**

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