PATRONES DE DISEÑO

ABSTRACT FACTORY

- Copia Juan Manuel
- Fischer Sebastián
- Fraschetti Nicolás

- Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software.

Clasificación:

- **Patrones creacionales**
- Patrones estructurales
- Patrones de comportamiento
- Patrones de interacción

Patrones Creacionales

- Corresponden a patrones de diseño de software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación. Es decir, ayudan a hacer un sistema independiente de como se crean, se componen, y se representan sus objetos.
- ABSTRACT FACTORY se encuentra dentro de este tipo de patrones.

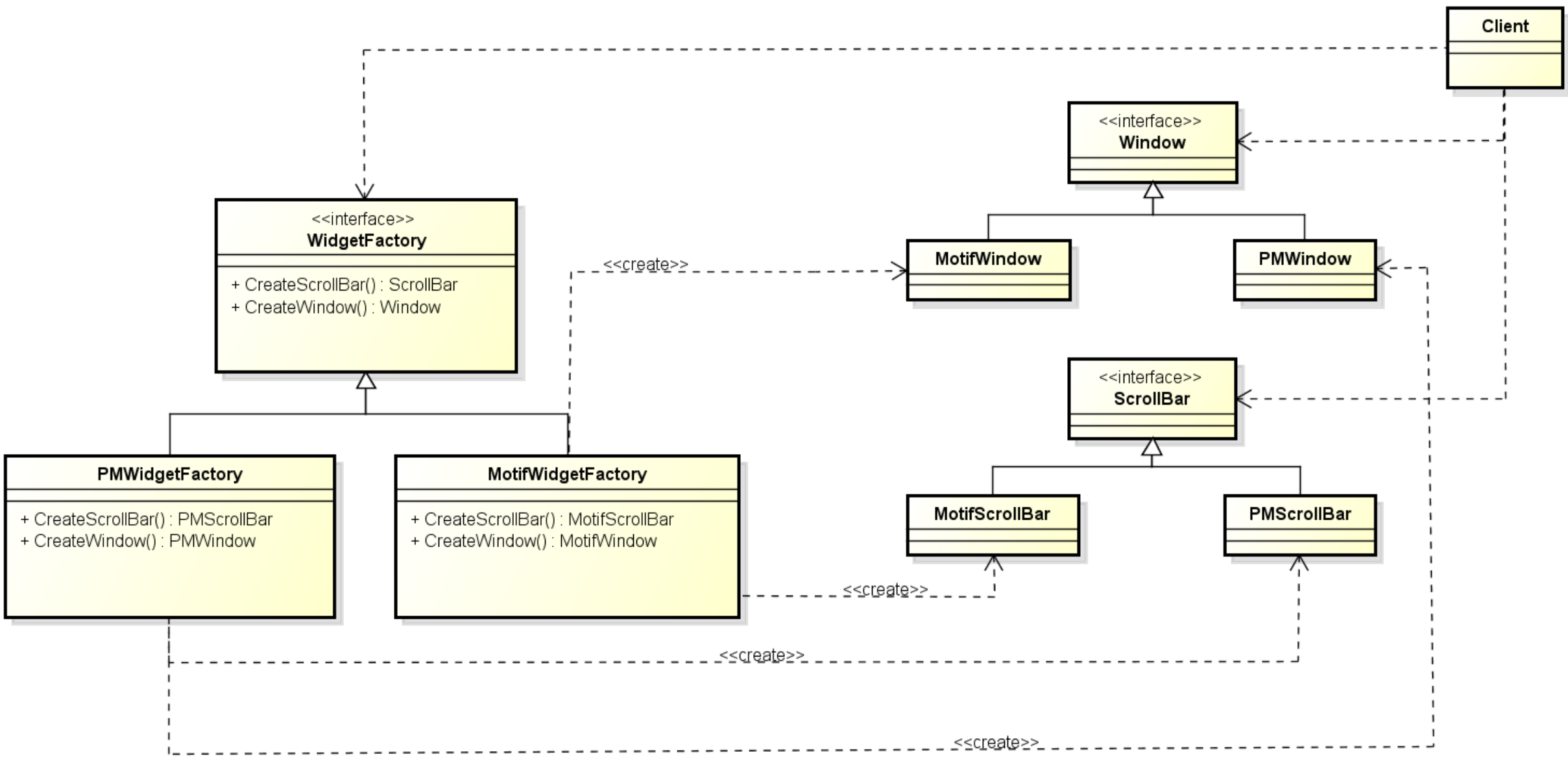
ABSTRACT FACTORY

PROPOSITO

- Proporciona una interfaz para crear familias (clases) de objetos relacionados sin especificar sus clases concretas.

MOTIVACION

- Pensemos en una aplicación que soporta varios estándares de interfaz grafica para el usuario. Los distintos estándares definen distintos aspectos y formas de comportamiento de los “Widget”, tales como barras de desplazamiento, ventanas, botones, etc. Para que una aplicación pueda portarse a distintos estándares, no debería implementar sus Widgets para una interfaz de usuario en particular.
- Podemos solucionar este problema definiendo una clase abstracta “widgetFactory” con los métodos de creación de cada widget, y una clase abstracta para cada tipo de widget.
- Cada estandarización seria una implementación de “widgetFactory” con los respectivos métodos de creación de widgets concretos para cada clase abstracta de widget.

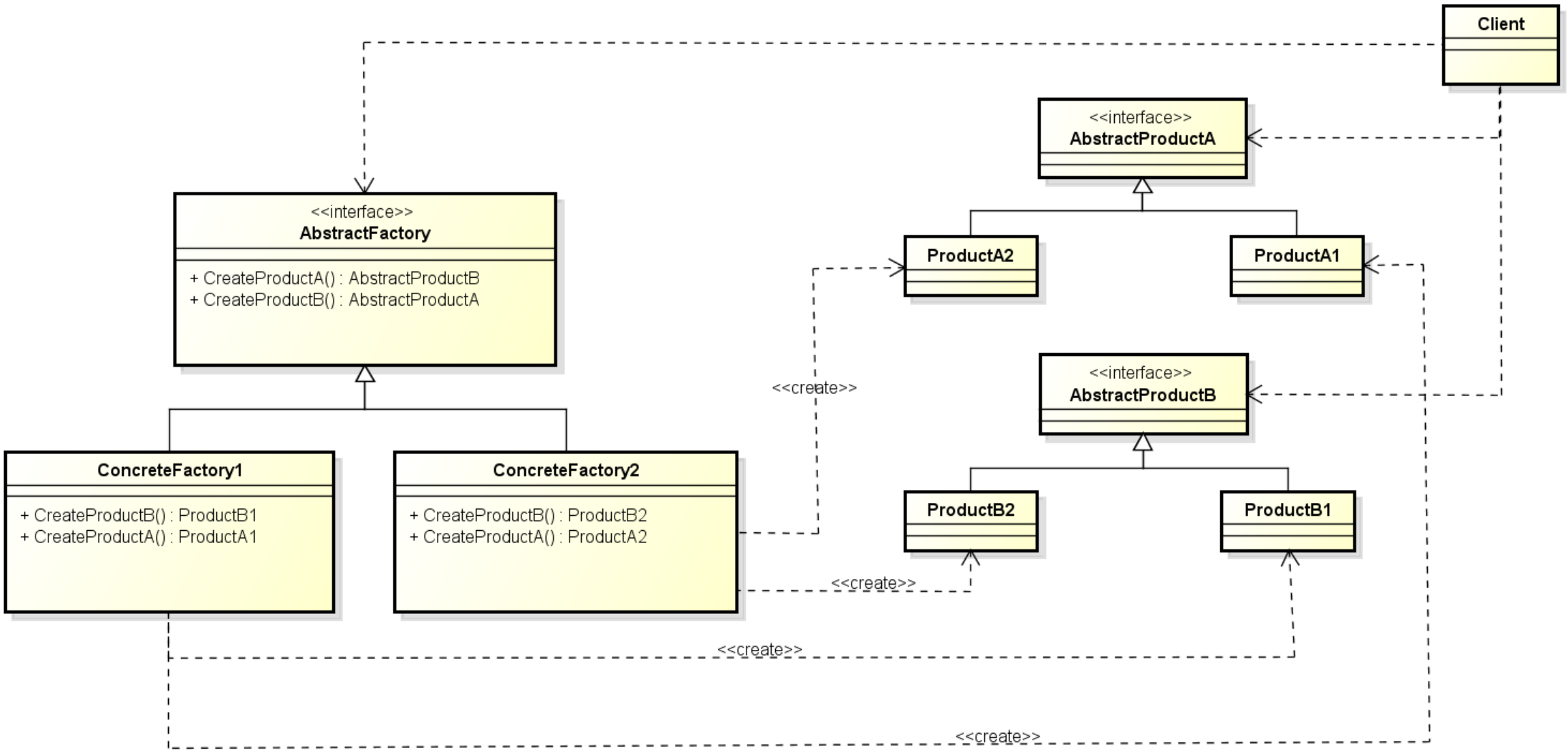

Client
<<interface>>
Window
<<interface>>
WidgetFactory
+ CreateScrollBar() : ScrollBar
+ CreateWindow() : Window
<<create>>
MotifWindow
PMWindow
<<interface>>
ScrollBar
PMWidgetFactory
+ CreateScrollBar() : PMScrollBar
+ CreateWindow() : PMWindow
MotifWidgetFactory
+ CreateScrollBar() : MotifScrollBar
+ CreateWindow() : MotifWindow
MotifScrollBar
PMScrollBar
<<create>>
<<create>>
<<create>>

- Los clientes crean widgets únicamente a través de la interfaz “WidgetFactory” y no tienen conocimiento de las clases que los implementan. Es decir, no tienen que atarse a una clase concreta, sino a una interfaz definida por una clase abstracta.
- A través de estas “Widget factories” concretas también se fuerza a que se cumplan las dependencias entre las clases concretas de Widgets, es decir que los botones y ventanas creados pertenecen a la misma implementación de la clase abstracta.

APLICABILIDAD

- El patron abstact factory se debe usar cuando:
- Un sistema debe ser independiente de como se crean, componen y representan sus productos.
- Un sistema debe estar configurado con una de muchas familias de productos.
- Una familia de productos relacionados esta diseñada para ser usada conjuntamente.
- Se quiere proporcionar una biblioteca de clases de productos, pero revelando solo sus interfaces, no su implementacion.

ESTRUCTURA


Client
<<interface>>
AbstractProductA
<<interface>>
AbstractFactory
+ CreateProductA() : AbstractProductB
+ CreateProductB() : AbstractProductA
ProductA2
ProductA1
<<create>>
<<interface>>
AbstractProductB
ConcreteFactory1
+ CreateProductB() : ProductB1
+ CreateProductA() : ProductA1
ConcreteFactory2
+ CreateProductB() : ProductB2
+ CreateProductA() : ProductA2
ProductB2
ProductB1
<<create>>
<<create>>
<<create>>

PARTICIPANTES

- ABSTRACT FACTORY:Declara una interfaz para operaciones que crean productos abstractos.
- CONCRETE FACTORY:Implementa las operaciones para crear productos concretos.
- ABSTRACT PRODUCT:Declara una interfaz para un tipo de objecto.
- CONCRETE PRODUCT:*Define un producto objeto a ser creado por la correspondiente fabrica concreta.
- *Implementa la interfaz abstract product.
- CLIENTE:Usa solamente interfaces declaradas por las clases abstract factory y abstract product.

COLABORACIONES

- Normalmente una sola instancia de una clase concreteFactory es creada en tiempo de ejecucion. Esta fabrica concreta crea objetos con una implementacion particular. Para crear objetos diferentes, el cliente debe usar una clase concreta diferente.
- AbstractFactory posterga la creacion de productos objetos a su subclase ConcreteFactory.

CONSECUENCIAS

- Aísla las clases concretas .
- Facilita el intercambio de familias de productos.
- Promueve la consistencia entre productos.
- Es difícil dar cabida a nuevos tipos de productos.

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;
}
```

PATRONES RELACIONADOS

Las clases AbstractFactory son usualmente implementadas usando factory method, aunque tambien pueden ser implementadas usando prototype.
Para las fabricas concretas se suele usar singleton, dado que generalmente se necesita una única instancia de ConcreteFactory para cada familia de productos.

BIBLIOGRAFIA

- https://es.wikipedia.org/wiki/Patron_de_diseño
- Design patterns: elements of reusable object-oriented software (1995) , de Erich Gamma , Richard Helm , Ralph Johnson , John Vlissides.
- http://es.wikipedia.org/wiki/Abstract_Factory
- Ejemplo sacado de Head First Design Patterns, de Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Si erra

.