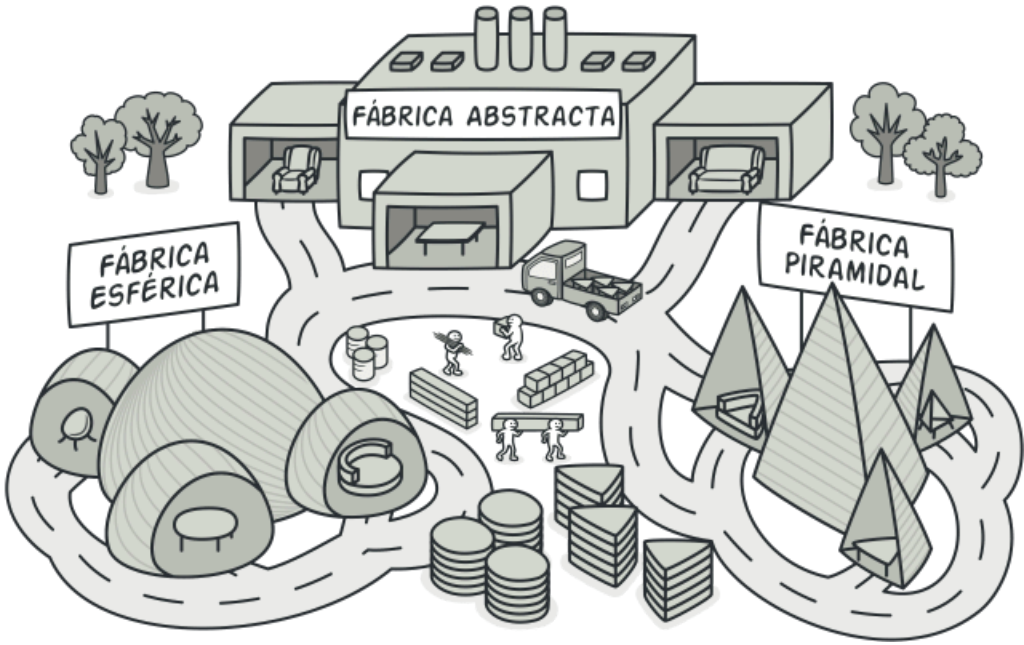
Patrón Abstract Factory

Patrón creacional (POO)

Fuente principal:
https://refactoring.guru/es/design-patterns/abstract-factory

Propósito

Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



Imágenes extraídas de Refactoring.Guru

Problema

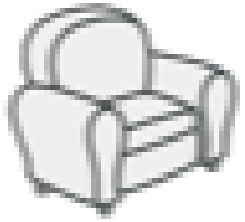
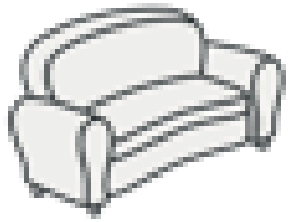
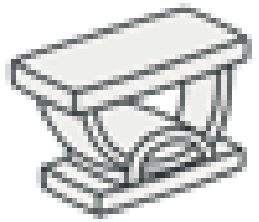
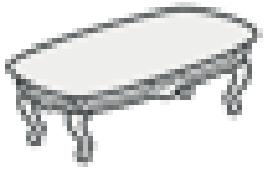
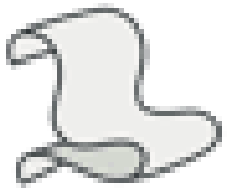
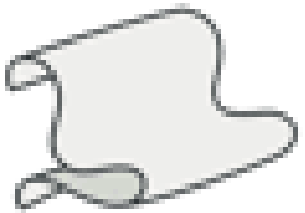
Imagina que estás creando un simulador de tienda de muebles. Tu código necesita manejar:

1. **Familias de productos relacionados**: `Silla` + `Sofá` + `Mesilla`.
2. **Variantes de esta familia**: Estilos como `Moderna`, `Victoriana`, `ArtDecó`.

El desafío:

Necesitamos una forma de crear objetos para que **combinen** con otros del mismo estilo. Un sofá moderno no encaja con una silla victoriana.

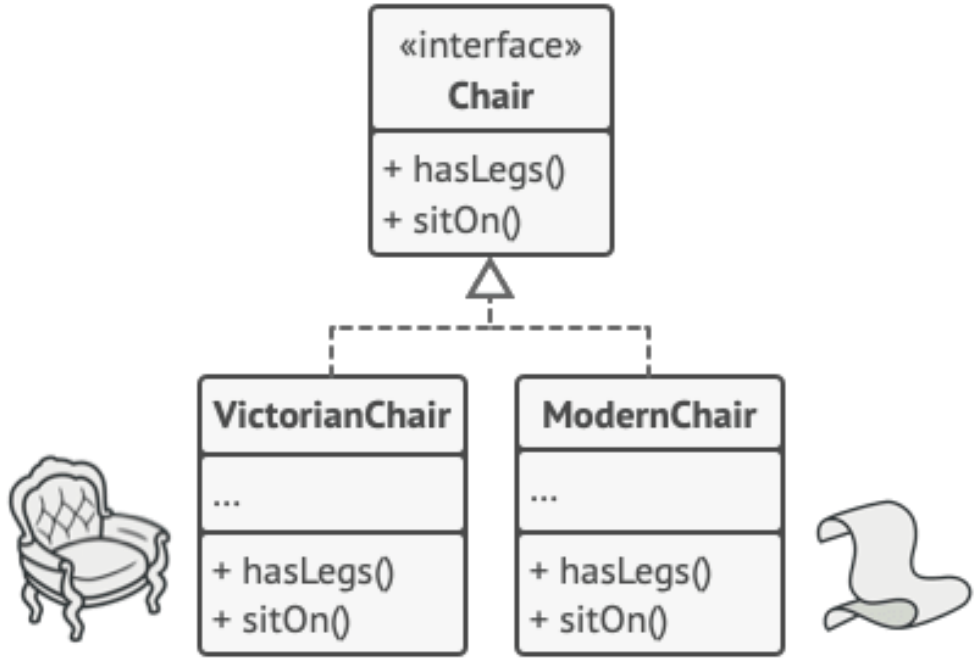
Imágenes extraídas de Refactoring.Guru

	Silla	Sofá	Mesilla
Art Decó			
Victoriana			
Moderna			

Imágenes extraídas de Refactoring.Guru

Solución

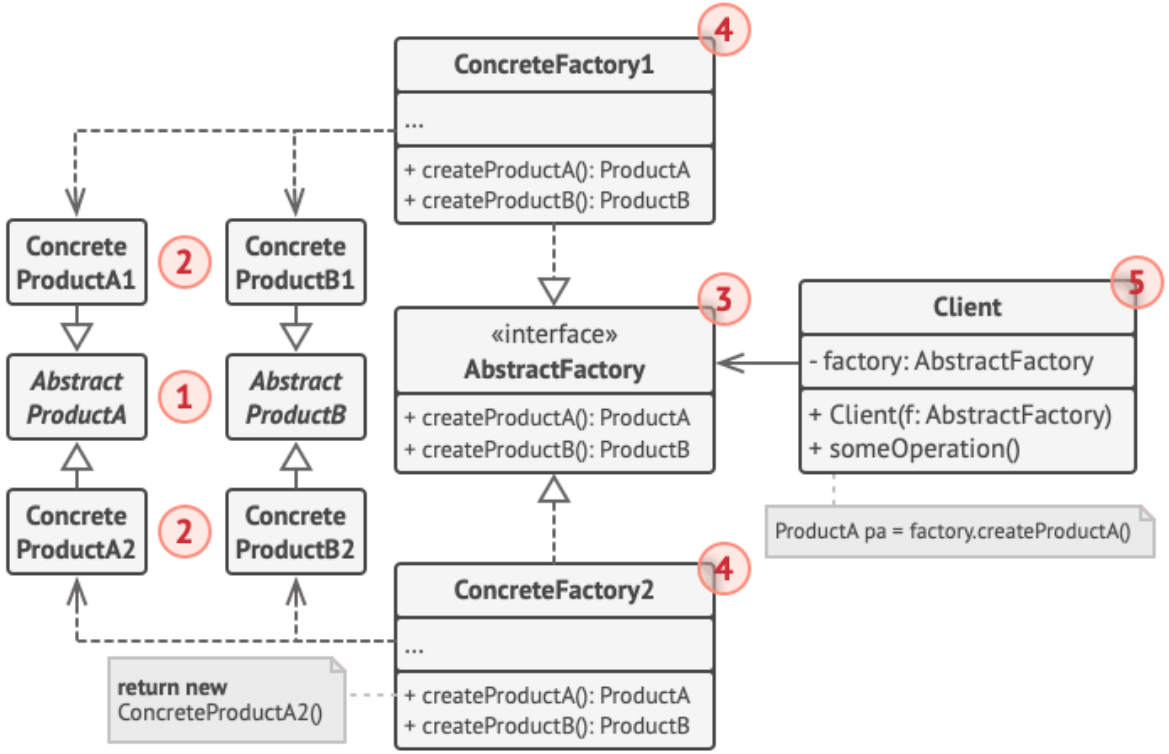
1. **Interfaces de Producto**: Declarar interfaces para cada producto de la familia (`Silla`, `Sofa`, `Mesilla`).
2. **Fábrica Abstracta**: Una interfaz que declara métodos de creación para todos los productos.
3. **Fábricas Concretas**: Clases que implementan la fábrica para una variante específica (ej: `FabricaMueblesModernos`).
4. **Acoplamiento Abstracto**: El cliente interactúa con las fábricas y productos solo a través de interfaces.



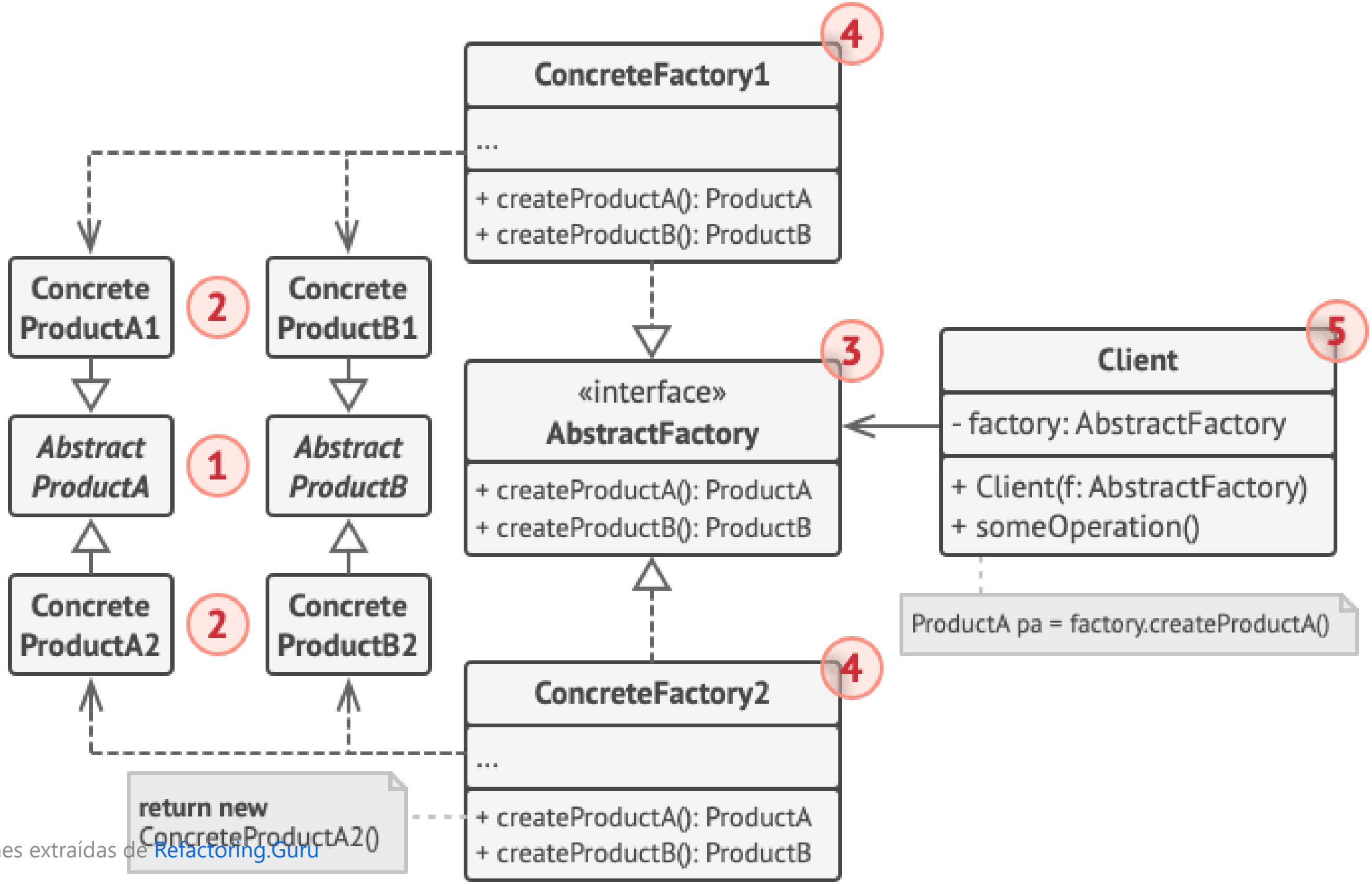
Imágenes extraídas de Refactoring.Guru

Estructura

1. **Productos Abstractos**: Declaran interfaces para una familia de productos.
2. **Productos Concretos**: Implementaciones específicas por variante.
3. **Fábrica Abstracta**: Interfaz con métodos de creación.
4. **Fábricas Concretas**: Crean productos de una variante específica.
5. **Cliente**: Utiliza solo las interfaces abstractas.



Imágenes extraídas de Refactoring.Guru



Imágenes extraídas de Refactoring.Guru

Aplicabilidad

- **Familias de productos**: Cuando tu código deba funcionar con varias familias de productos relacionados.
- **Independencia de clases concretas**: Cuando no quieras que tu código dependa de las clases finales de los productos.
- **Extensibilidad**: Para permitir añadir nuevas variantes de familias fácilmente.
- **Sustitución de Factory Method**: Cuando una clase tiene demasiados métodos de fábrica que ensucian su responsabilidad principal.

Pros y contras

Pros

- ✅ **Compatibilidad**: Garantiza que los productos obtenidos de una fábrica sean compatibles.
- ✅ **Bajo acoplamiento**: Evita la dependencia fuerte entre productos concretos y el cliente.
- ✅ **S.R.P.**: Centraliza la creación de productos.
- ✅ **O.C.P.**: Permite introducir nuevas variantes sin romper el código existente.

Contras

- ❌ **Complejidad**: El código puede volverse más complejo debido a la introducción de múltiples interfaces y clases.

Ejemplo en Java: Fábrica y Productos

```java
// Interfaz de la Fábrica Abstracta
public interface FabricaMuebles {
    Silla crearSilla();
    Sofa crearSofa();
    Mesilla crearMesilla();
}

// Fábrica Concreta para estilo Moderno
public class FabricaMueblesModernos implements FabricaMuebles {
    @Override
    public Silla crearSilla() { return new SillaModerna(); }
    @Override
    public Sofa crearSofa() { return new SofaModerno(); }
    @Override
    public Mesilla crearMesilla() { return new MesillaModerna(); }
}
```

Ejemplo en Java: Cliente

```
public class Cliente {
    private Silla silla;
    private Sofa sofa;
    private Mesilla mesilla;

    // Recibe la fábrica abstracta, no la concreta
    public Cliente(FabricaMuebles fabrica) {
        this.silla = fabrica.crearSilla();
        this.sofa = fabrica.crearSofa();
        this.mesilla = fabrica.crearMesilla();
    }

    public void usarMuebles() {
        silla.sentarse();
        sofa.tumbarse();
        mesilla.ponerCosasEncima();
    }
}
```