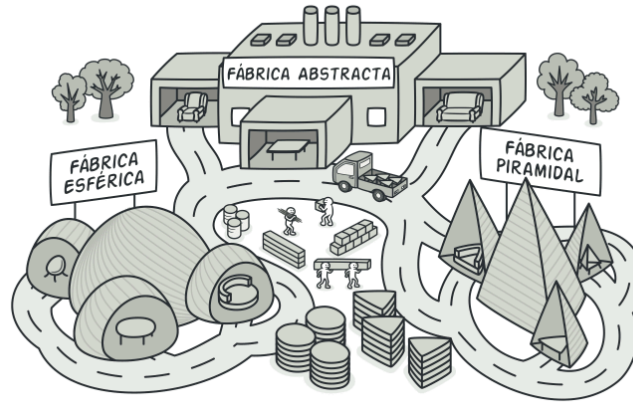


PATRÓN DE DISEÑO: FABRICA ABSTRACTA

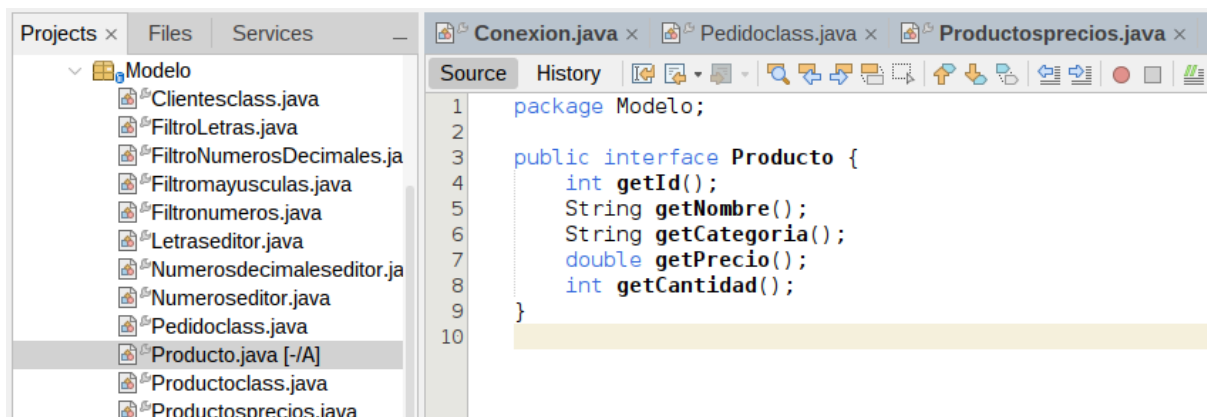
😊 Propósito:

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



Tenemos la clase **Productoclass.java** es una clase en el paquete Modelo, que nos permite contruir un Producto tiene sus variables, getters y setters, y nos ayuda igual a contactar la base de datosy nos ayuda a obtener los productos.

Para poder aplicar el patrón de fábrica abstracta necesitamos una Interfaz para nuestros productos.

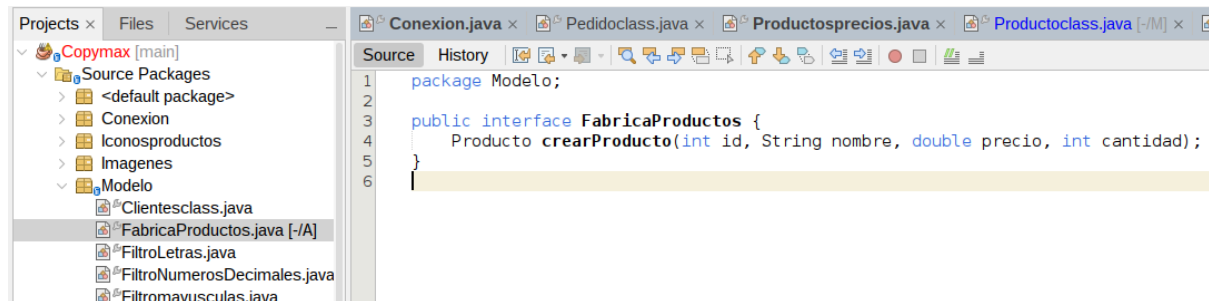


Le agregamos sus parámetros a nuestra interfaz, ahora implementamos la Interfaz en nuestra clase de **productos**.

```
public class Productoclass implements Producto {  
    setters  
    getters  
}
```

Esto es para asegurarse de que un producto que se cree debe de mantener la estructura que establecimos en nuestra interfaz.

Ahora lo que tenemos que implementar son nuestras fábricas en específico pondremos “3 fábricas” en esta sección de productos.



Creamos nuestra interfaz primero que sería como la base de nuestras fábricas, tendremos igual nuestra fábrica de útiles Escolares y otra de Impresiones.

Fabrica de **útiles escolares**:

```
package Modelo;
```

```
public class FabricaUtilEscolar implements FabricaProductos {
```

```
    @Override
```

```
    public Producto crearProducto(int id, String nombre, double precio, int cantidad) {
```

```
        Productoclass producto = new Productoclass();
```

```
        producto.setId(id);
```

```
        producto.setNombre(nombre);
```

```
        producto.setPrecio(precio);
```

```
        producto.setCantidad(cantidad);
```

```
        producto.setCategoria("Útil Escolar");
```

```
        return producto;
```

```
    }
```

```
}
```

Fabrica de Impresion:

```
package Modelo;
```

```
public class FabricaImpresion implements FabricaProductos {
```

```
    @Override
```

```
    public Producto crearProducto(int id, String nombre, double precio, int cantidad) {
```

```
        Productoclass producto = new Productoclass();
```

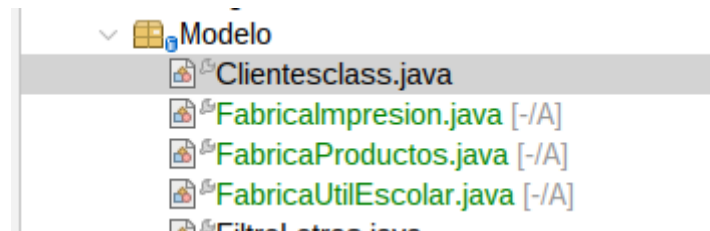
```
        producto.setId(id);
```

```

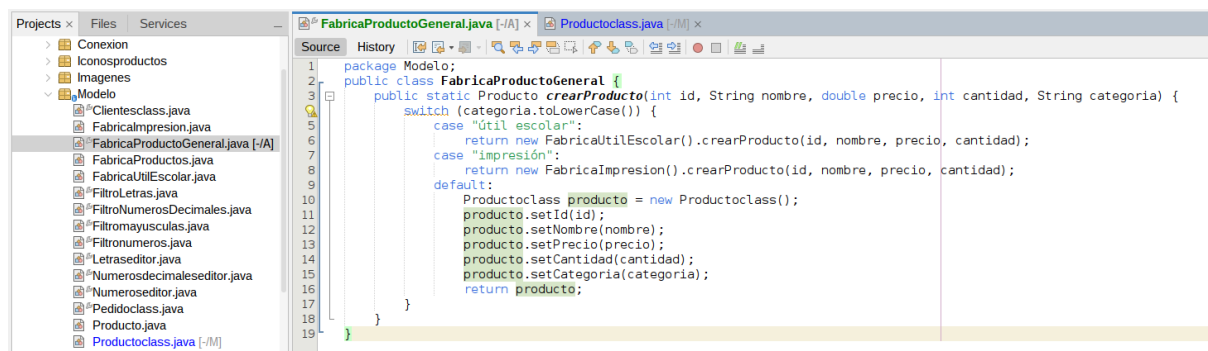
    producto.setNombre(nombre);
    producto.setPrecio(precio);
    producto.setCantidad(cantidad);
    producto.setCategoria("Impresión");
    return producto;
}
}

```

Ya tenemos nuestras fábricas.



Creamos una fábrica general que nos ayudara a utilizar las demás fabricas.



En dado caso de que no tengamos una fábrica establecida para la categoría del producto, se crea una con Default tenemos dos casos de nuestras diferentes fábricas.

Caso numero 1:

```

case "útil escolar":
    return new FabricaUtilEscolar().crearProducto(id, nombre, precio, cantidad);

```

Ya que tengamos nuestras clases, modifícalas las consultas de nuestra clase Modelo de productos, vamos a cambiar la forma en la que se crean los productos, usaremos nuestras fábricas que generemos dependiendo la categoría.

```

public List<Producto> obtenerProductos() {
    List<Producto> productos = new ArrayList<>();
    Conexion conex = new Conexion();
    String sql = "SELECT idProductos, Nombre_producto, Precio, Cantidad, Categoria FROM Productos";
    try (Connection con = conex.getConnection());

```

```

        PreparedStatement pst = con.prepareStatement(sql);
        ResultSet rs = pst.executeQuery() {

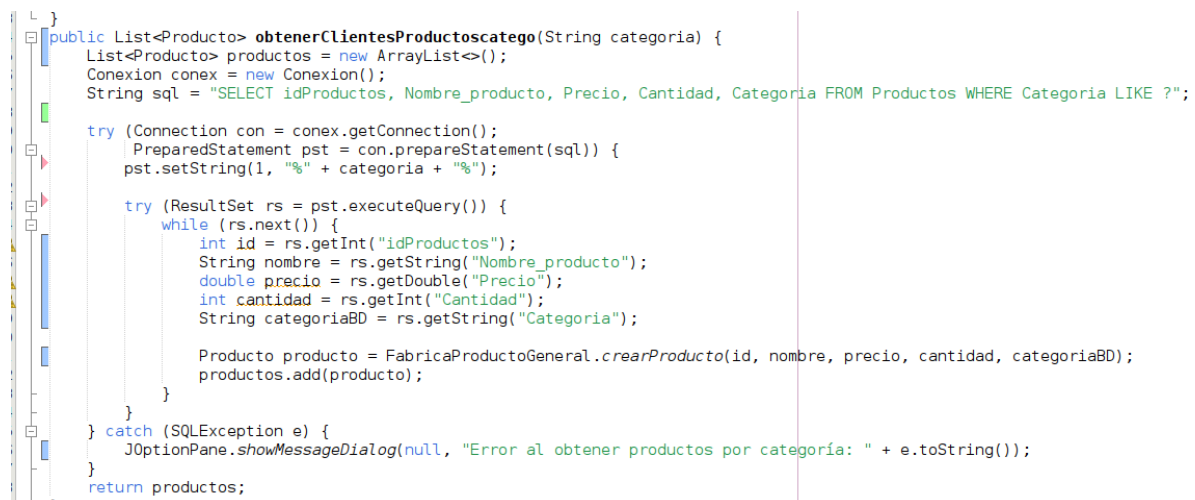
            while (rs.next()) {
                int id = rs.getInt("idProductos");
                String nombre = rs.getString("Nombre_producto");
                double precio = rs.getDouble("Precio");
                int cantidad = rs.getInt("Cantidad");
                String categoria = rs.getString("Categoria");

                Producto producto = FabricaProductoGeneral.crearProducto(id, nombre, precio,
cantidad, categoria);
                productos.add(producto);
            }
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Error al obtener productos: " + e.toString());
        }

        return productos;
    }
}

```

Así mismo cambiamos nuestro otro método existente en nuestra clase de Productoclass.java



```

    }
    public List<Producto> obtenerClientesProductoscatego(String categoria) {
        List<Producto> productos = new ArrayList<>();
        Conexion conex = new Conexion();
        String sql = "SELECT idProductos, Nombre_producto, Precio, Cantidad, Categoria FROM Productos WHERE Categoria LIKE ?";

        try (Connection con = conex.getConnection();
            PreparedStatement pst = con.prepareStatement(sql)) {
            pst.setString(1, "%" + categoria + "%");

            try (ResultSet rs = pst.executeQuery()) {
                while (rs.next()) {
                    int id = rs.getInt("idProductos");
                    String nombre = rs.getString("Nombre_producto");
                    double precio = rs.getDouble("Precio");
                    int cantidad = rs.getInt("Cantidad");
                    String categoriaBD = rs.getString("Categoria");

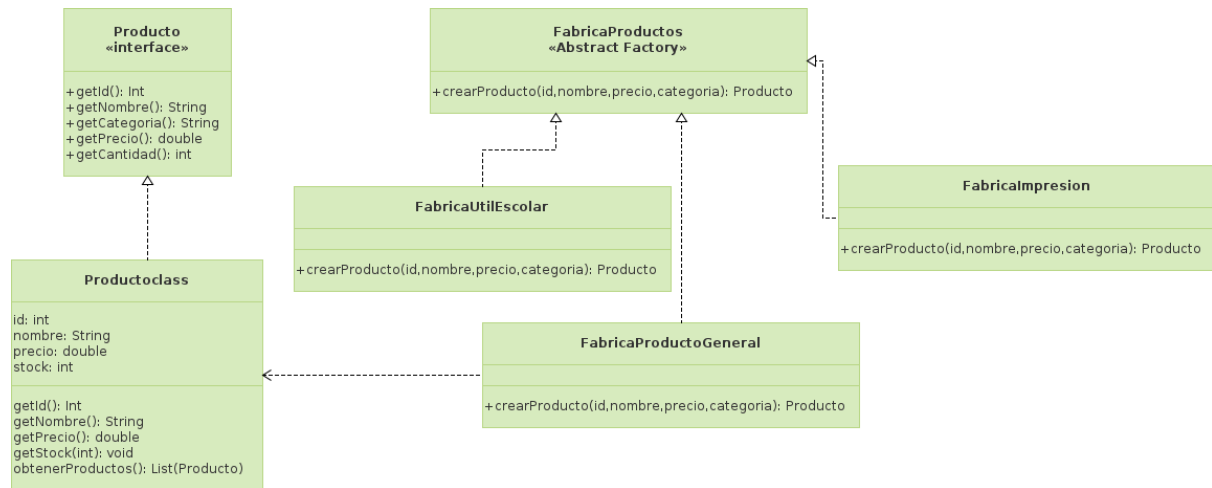
                    Producto producto = FabricaProductoGeneral.crearProducto(id, nombre, precio, cantidad, categoriaBD);
                    productos.add(producto);
                }
            }
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Error al obtener productos por categoría: " + e.toString());
        }

        return productos;
    }
}

```

Esta aplicación del patrón fábrica abstracta no debería modificar el uso del programa, modifica la creación de los productos de nuestro programa, en dado caso de querer añadir más fábricas sería cuestión de construir más clases de “fábricas” por ejemplo: “Productos de oficina”, “Accesorios de teléfonos” etc.

Estructura del Patrón



Pros y contras

Ventajas	Desventajas
Facilita cambios futuros en la creación de productos sin modificar otras partes del código.	Requiere mayor clases e interfaces.
Sistema modular.	Para proyectos pequeños esta complejidad no es tan necesaria.
Cada fábrica tiene la única responsabilidad de un tipo de producto.	

La patrón de fábrica abstracta es especialmente útil cuando se trata de un sistema que puede necesitar modificaciones en la creación de productos o si se espera un crecimiento en el número de categorías de productos.

PATRÓN DE DISEÑO: PROTOTIPO

Propósito: El patrón de diseño Prototype es un patrón creacional que permite crear nuevos objetos copiando instancias existentes, en lugar de crear nuevas instancias desde cero. Esto es útil cuando la creación de un objeto es costosa o compleja.

Implementación en el Proyecto:

Tenemos la clase Metododepago.java dentro del paquete Vista. Esta clase representa la lógica del proceso de pago en el sistema, incluyendo atributos como el total de la venta, los métodos de pago seleccionados y los montos de pago.

Para aplicar el patrón Prototype, primero necesitamos una interfaz que establezca el contrato para la clonación de objetos.

Definición de la interfaz Prototype:

```
public interface PagoPrototype {  
    Metododepago clonar();  
}
```

Esta interfaz define el método `clonar()`, que cada clase que implemente el patrón debe definir para devolver una copia de sí misma.

Implementación en la Clase Metododepago:

```
public class Metododepago extends javax.swing.JFrame implements PagoPrototype {  
    private double totalVenta;  
    private double pago1;  
    private double pago2;  
    private String metodoPago1;  
    private String metodoPago2;  
  
    public Metododepago(double totalVenta, double pago1, double pago2, String  
metodoPago1, String metodoPago2) {  
        this.totalVenta = totalVenta;  
        this.pago1 = pago1;  
        this.pago2 = pago2;  
        this.metodoPago1 = metodoPago1;  
        this.metodoPago2 = metodoPago2;  
    }  
}
```

```

@Override
public Metododepago clonar() {
    return new Metododepago(this.totalVenta, this.pago1, this.pago2, this.metodoPago1,
this.metodoPago2);
}
}

```

Esto garantiza que cada instancia de **Metododepago** pueda clonarse sin necesidad de crear una nueva desde cero.

Uso del Patrón Prototype en la Lógica del Programa:

```

// Crear el objeto original del pago
Metododepago pagoOriginal = new Metododepago(totalVenta, pago1, pago2, metodopago1,
metodopago2);

// Clonar el pago original
Metododepago pagoClonado = pagoOriginal.clonar();

// Modificar el clon sin afectar el original
pagoClonado.setPago1(50.0);

// Verificar que el clon es independiente
System.out.println("Pago Original:");
pagoOriginal.imprimirDatos();
System.out.println("\nPago Clonado:");
pagoClonado.imprimirDatos();

```

Estructura del Patrón:

1. **Interfaz Prototype:** Define el método **clonar()**.
2. **Clase Concreta:** Implementa la interfaz y define la lógica de clonación.
3. **Uso en el Programa:** Se crea un objeto, se clona y se modifican los atributos del clon sin afectar el original.

Pros y contras:

Ventajas:

- Permite crear nuevos objetos de forma eficiente.
- Reduce la necesidad de crear nuevas instancias desde cero.
- Facilita la creación de copias sin afectar el objeto original.

Desventajas:

- Puede aumentar la complejidad si el objeto tiene referencias a otros objetos que también deben clonarse.
- En algunos casos, una clonación profunda puede ser más compleja de implementar.

El patrón Prototype es útil en sistemas donde la creación de objetos es costosa y se necesita una forma eficiente de generar copias sin modificar el objeto original.