Tecnológico Nacional De México

Instituto Tecnológico de Oaxaca

Ingeniería En Sistemas Computacionales

Diseño e Implementación de Software con Patrones.

7 am – 8 am

Unidad 3

“Patrón de Diseño Fliwith”

Presenta:

Implementación de los Patrones en el Sistema Copy Max

Nombres	Numero de Control
Bautista Fabian Max	**C19160532**
Celis Delgado Jorge Eduardo	**21160599**
Flores Guzmán Alan Ismael	**20161193**
García Osorio Bolívar	**20161819**
Pérez Barrios Diego	**21160750**
Pérez Martínez Edith Esmeralda	**21160752**
Sixto Morales Ángel	**21160797**

Periodo Escolar:

Febrero – Julio

Grupo:

7SB

Maestro:

Espinoza Pérez Jacob

Oaxaca de Juárez, Oaxaca **Abril 2025**

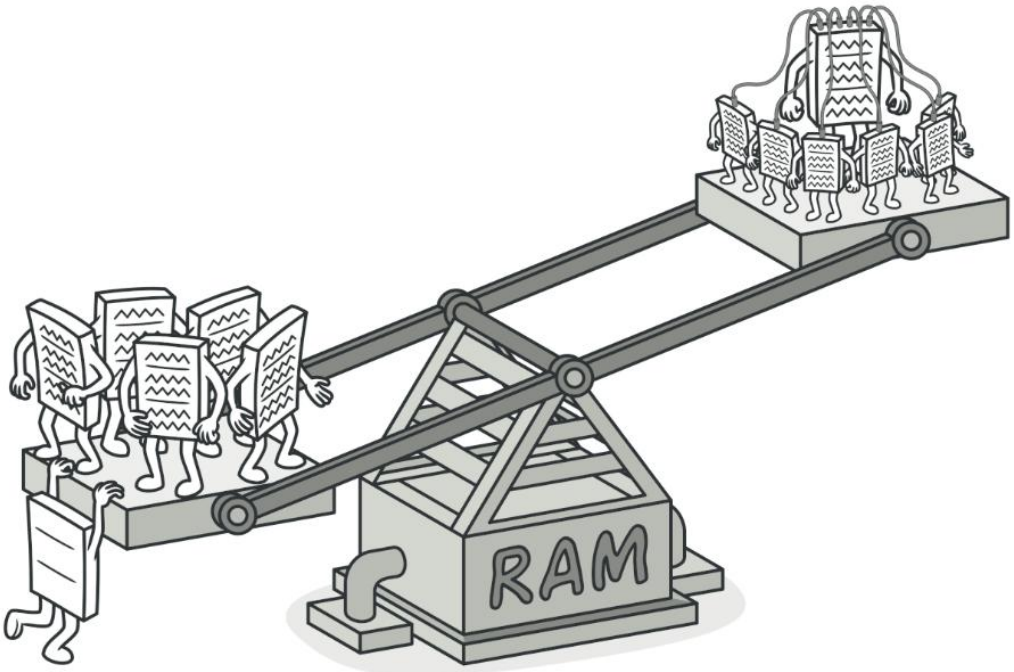
INDICE

INTRODUCCIÓN

EL patrón permite compartir objetos entre múltiples instancias para reducir el uso de memoria. Separa los datos en dos tipos: intrínsecos (que se pueden compartir entre objetos, como el nombre del proveedor o la categoría del producto) y extrínsecos (que cambian según el contexto, como la cantidad en inventario o el precio). Al aplicar Flyweight, el sistema evita crear múltiples copias de los mismos datos, lo que mejora el rendimiento y eficiencia.

PATRÓN DE DISEÑO FLIWIHT: IMPLEMENTADO EN COPY MAX

Flyweight es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



La clase ProductosFactory aplica el patron fliwith con la siguiente líneas de código.

```
public class ProductosFactory {

    private static final Map<String, Productosprecios> pool = new HashMap<>();
```

CÓDIGO IMPLEMENTADO

En public class ProductosFactory {

```
    private static final Map<String, Productosprecios> pool = new HashMap<>();
```

Se define un mapa (pool) que almacenara los objetos Productoprecios.

La clave es el string que se genera con nombre, precio e icono generando como una almacén de objetos compartidos.

El siguiente línea de código

```
public static Productosprecios getProducto(String nombre, double precio, String icono) {

    String key = nombre + "!" + precio + "!" + icono;
```

Se genera una clave única (key) combinando los atributos que identifican al producto asi es como se vrifica si ya existe un objeto con estos datos exactos.

```
if (!pool.containsKey(key)) {

    pool.put(key, new Productosprecios(nombre, precio, icono));

}
```

Si este no existe en el mapa se crea un nuevo objeto y se agrega al pool

```
return pool.get(key);
```

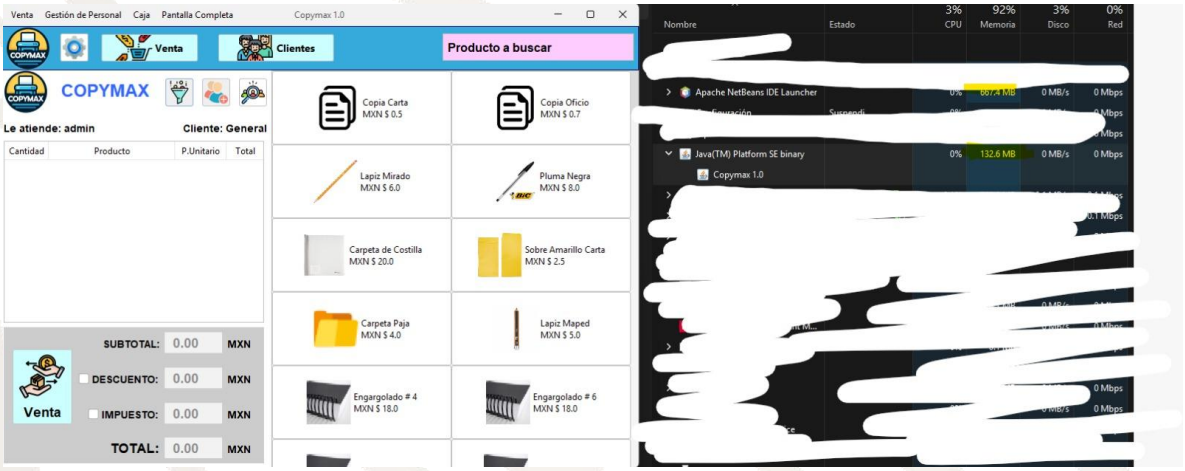
En dado caso que este ya exista, simplemente lo reutiliza y lo retorna.

```java
package Modelo;

import java.util.HashMap;
import java.util.Map;

public class ProductosFactory {
    private static final Map<String, Productosprecios> pool = new HashMap<>();

    public static Productosprecios getProducto(String nombre, double precio, String icono) {
        String key = nombre + "|" + precio + "|" + icono;
        if (!pool.containsKey(key)) {
            pool.put(key, new Productosprecios(nombre, precio, icono));
        }
        return pool.get(key);
    }
}
```

Cada producto se representa con una imagen que en este caso de pruebas se puso una imagen de un gato.

En lugar de crear un nuevo objeto de producto cada vez se carga, se reutiliza si este ya existe.

Esto reduce el consumo de memoria y mejora el rendimiento especialmente si hay productos repetidos o se consultan muchas veces.



Al momento de agregar dos veces un producto con misma dicha información este solo se crea un objeto en memoria y ambas referencias apuntan al mismo.

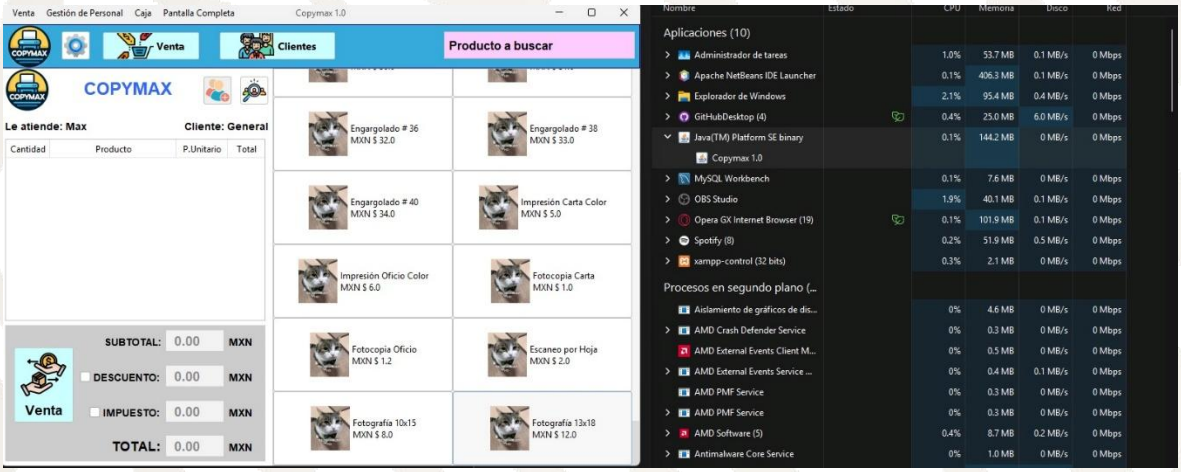
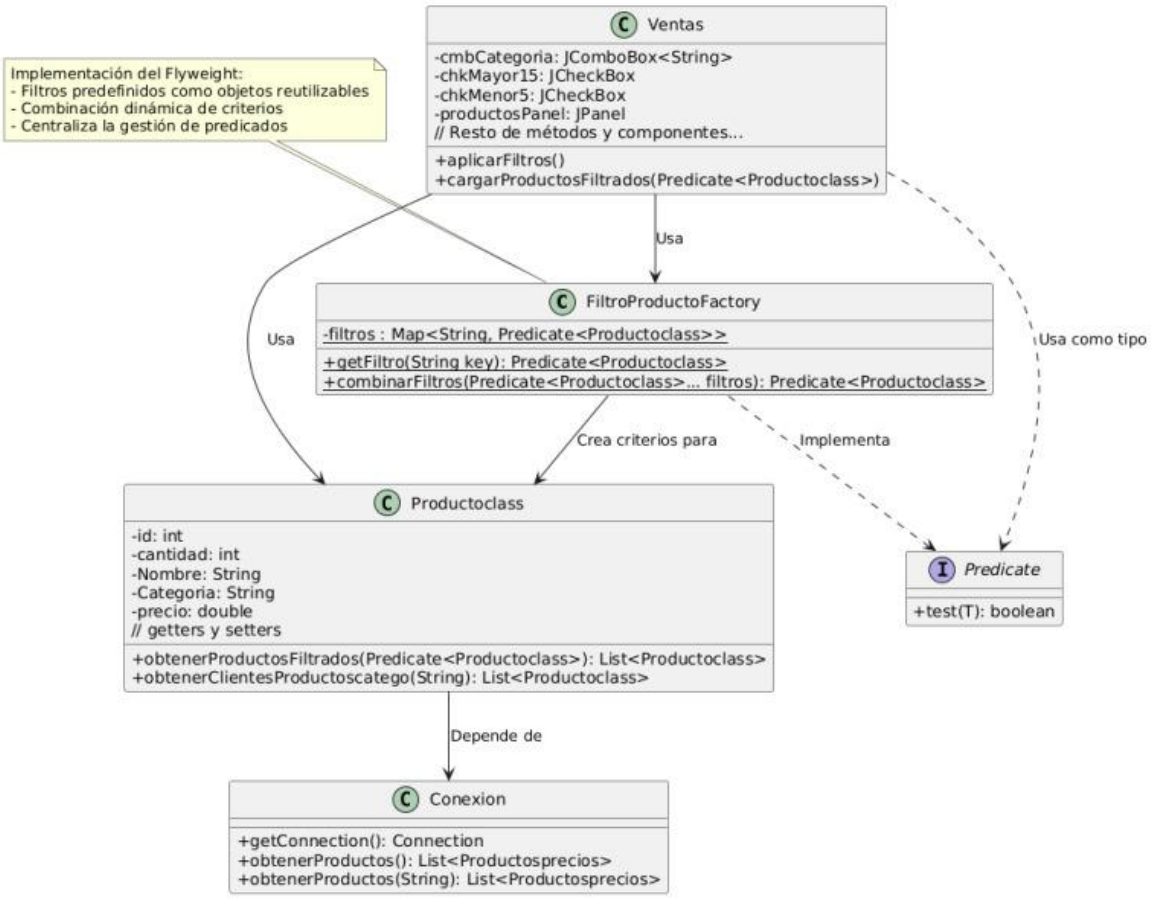


DIAGRAMA UML



VENTAJAS Y DESVENTAJAS

Ventajas	Desventajas
Reducción en el uso de memoria: Disminuye la cantidad de objetos duplicados.	Mayor complejidad estructural: Es necesario separar correctamente los datos compartibles de los contextuales.
Mejor rendimiento: Ideal para listas extensas como catálogos de productos.	Dificultad para implementar si no hay una clara repetición de datos.
Fácil mantenimiento: Permite cambiar datos comunes en un solo lugar.	Riesgo de compartir datos sensibles sin intención si no se tiene cuidado.

CONCLUSIÓN

El patrón Flyweight es especialmente útil cuando el sistema maneja muchos objetos similares, como productos que comparten características comunes. Su implementación ayuda a optimizar el rendimiento del sistema y reduce el consumo innecesario de memoria, lo que es clave para mantener la eficiencia a medida que crece la base de datos de productos.