



“Impacto de los Patrones de Diseño en Nuestro Proyecto”

Transformando código desde el
inicio



Introducción

Esta exposición aborda cómo los patrones de diseño han transformado positivamente nuestro código desde el inicio del proyecto. No presentaremos herramientas específicas, sino un conjunto de técnicas aplicadas en nuestro proyecto basadas en principios de diseño.



**“Hablemos de los
patrones de diseño”**

Factory Method

Propósito

Su principal objetivo es crear objetos sin especificar la clase exacta de los mismos, proporcionando una interfaz para crear instancias de una clase.

Aportaciones



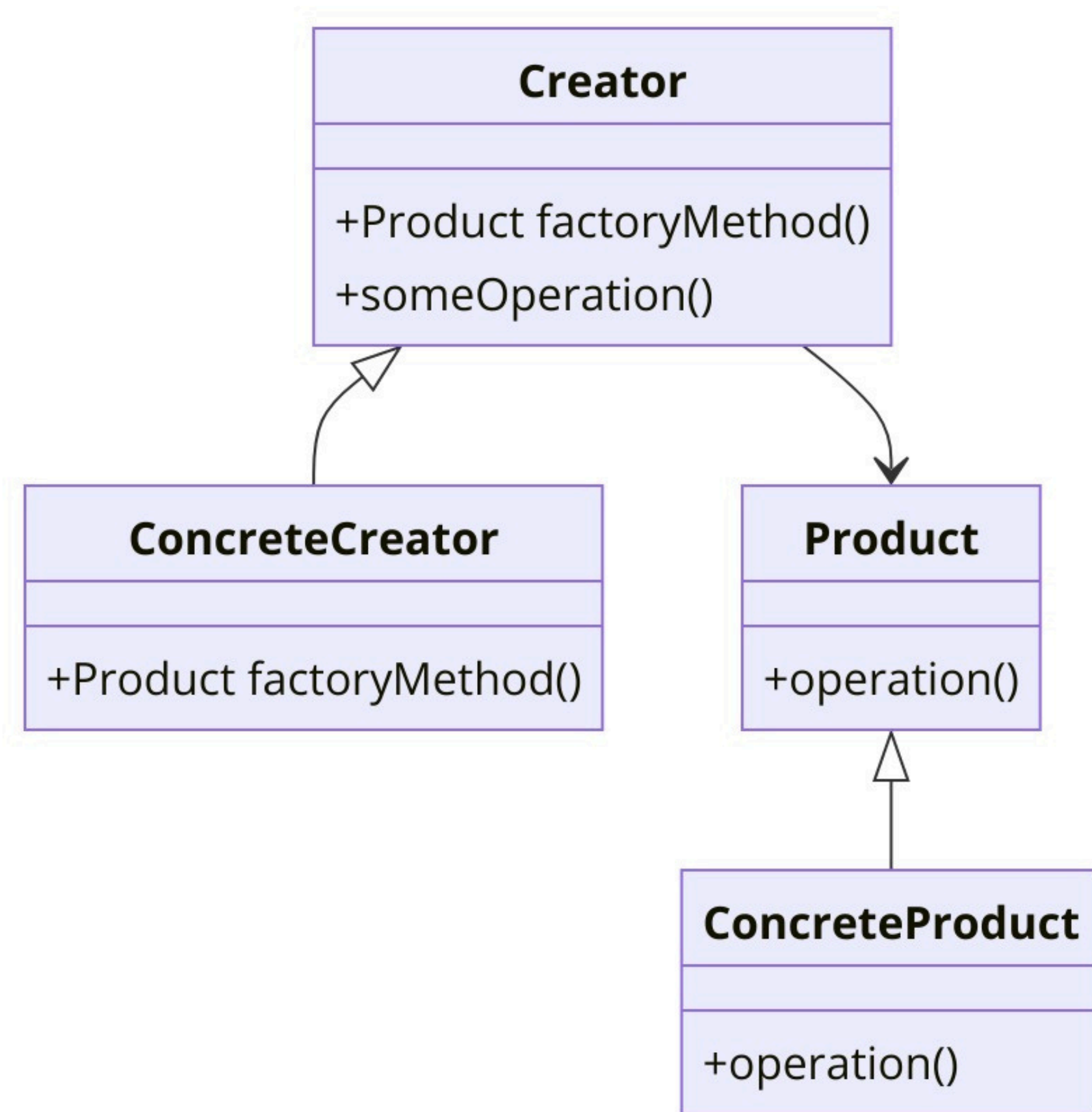
Flexibilidad: Puedes agregar nuevos tipos de membresías fácilmente sin cambiar el código existente.



Desacoplamiento: La lógica de creación de membresías está centralizada en las fábricas, lo que facilita el mantenimiento y la comprensión del código.



Claridad: El código es más claro y organizado, ya que cada fábrica se encarga de un tipo específico de membresía.



¿Por qué se utilizó?



- Flexibilidad en la Creación de Objetos: Permite crear objetos sin especificar la clase exacta del objeto que se va a crear. Esto es particularmente útil en nuestro proyecto donde tenemos diferentes tipos de membresías (Día, Semana, Mes y Personalizada) que necesitan ser creadas dinámicamente.
- Desacoplamiento: Separa el código que crea los objetos del código que los utiliza, promoviendo un diseño más limpio y mantenible.
- Extensibilidad: Facilita la adición de nuevos tipos de membresías en el futuro sin necesidad de modificar el código existente.





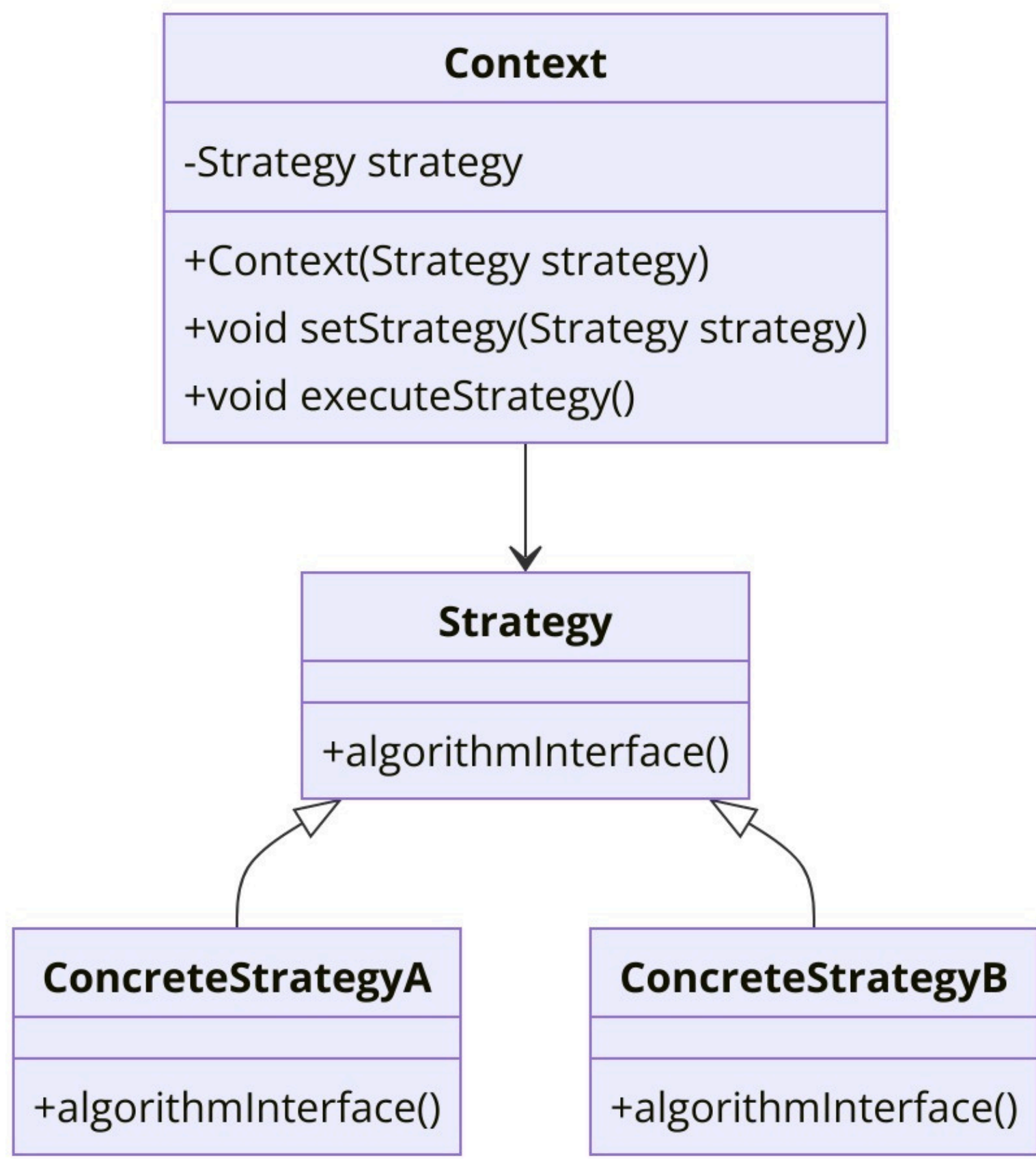
Strategy

Propósito

El propósito de éste patrón es permitir que el algoritmo varíe independientemente del cliente que lo usa. Strategy permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables.

Ventajas

1. Facilita la extensión del código al agregar nuevos algoritmos sin modificar el código existente.
2. Mejora la mantenibilidad y legibilidad del código.
3. Permite cambiar los algoritmos en tiempo de ejecución.



¿Por qué se utilizó?



- Algoritmos Intercambiables: Permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Esto es útil para nuestro proyecto ya que podemos tener diferentes estrategias de cálculo de precios para las membresías.
- Cambio en Tiempo de Ejecución: Los algoritmos pueden cambiarse en tiempo de ejecución según las necesidades del usuario o el contexto.
- Simplicidad y Mantenimiento: Mejora la mantenibilidad del código al encapsular los algoritmos en clases separadas.





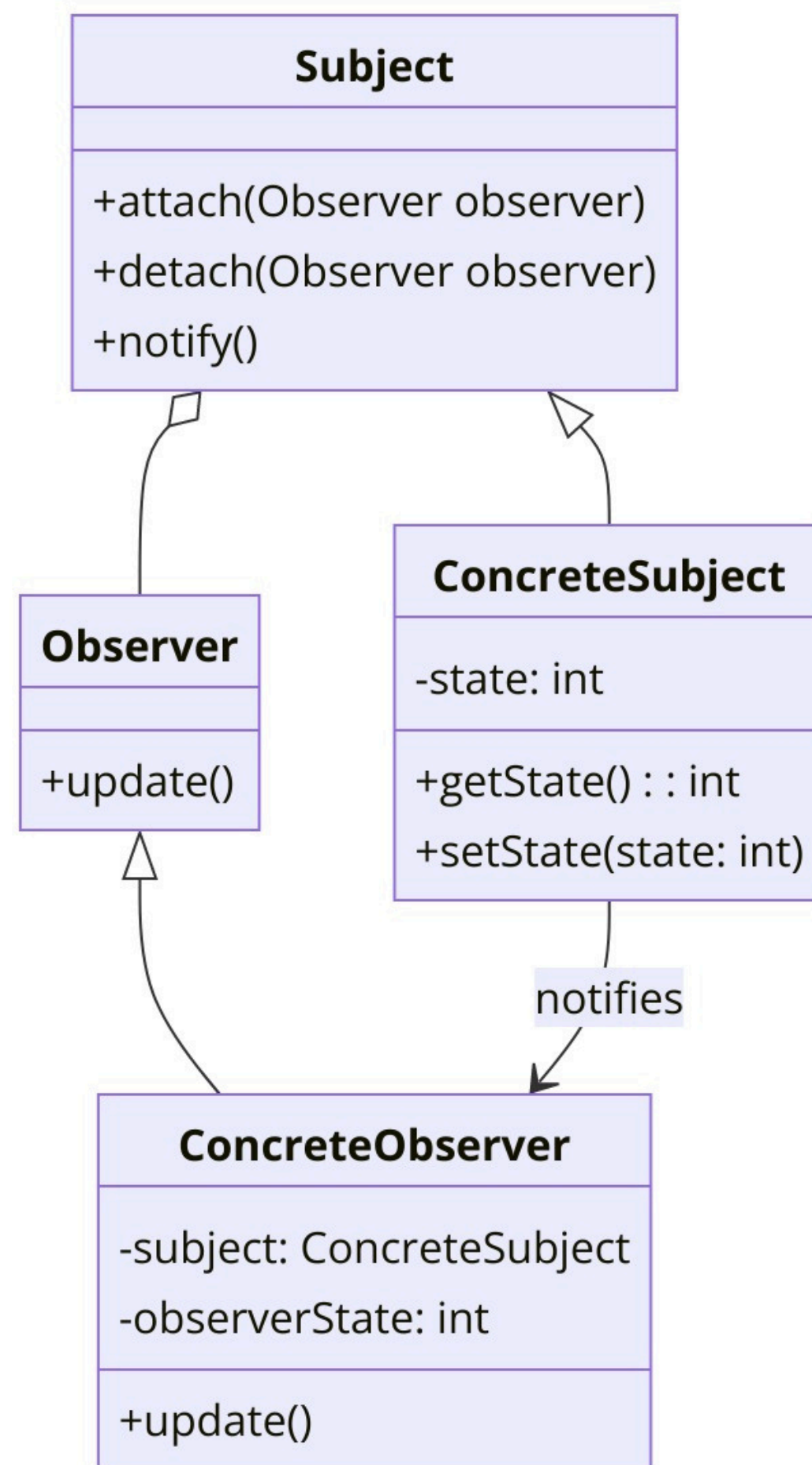
Observer

Definición

El patrón de diseño Observer define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Propósito

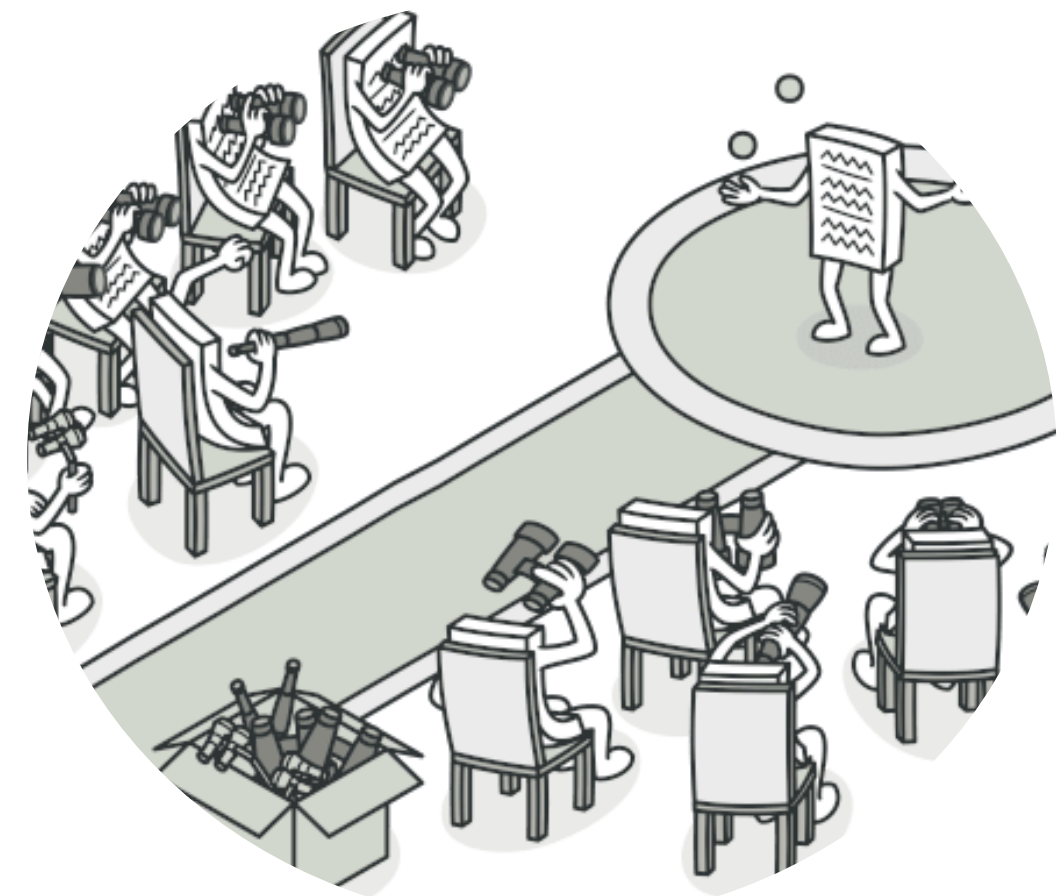
El propósito del patrón Observer es asegurar que los cambios en el estado de un objeto se reflejen automáticamente en todos los objetos que dependen de él.



¿Por qué se utilizó?



- Actualización Automática: Permite que un objeto (sujeto) notifique a otros objetos (observadores) sobre cambios en su estado, promoviendo una actualización automática de las vistas cuando cambian los datos.
- Desacoplamiento: Desacopla el sujeto de sus observadores, permitiendo que se añadan o eliminen observadores sin necesidad de modificar el código del sujeto.
- Escalabilidad: Facilita la escalabilidad del sistema al permitir que múltiples vistas u otros componentes respondan a cambios en los datos de forma coordinada.





**“Hablemos de nuestro
proyecto”**



Contexto de la solución

Monster APP GYM es una aplicación de escritorio desarrollado en Java. El POC permite generar y personalizar membresías flexibles. El proyecto fue diseñado para ser, principalmente, escalable; por lo que, en un futuro, se le agregarán funciones relacionadas con la gestión de un gimnasio.




Diagrama de clases (carece de patrones)



El siguiente diagrama a mostrar no tiene ningún patrón de diseño, además de que no es flexible, ya que solamente se pueden elegir 3 membresías diferentes.



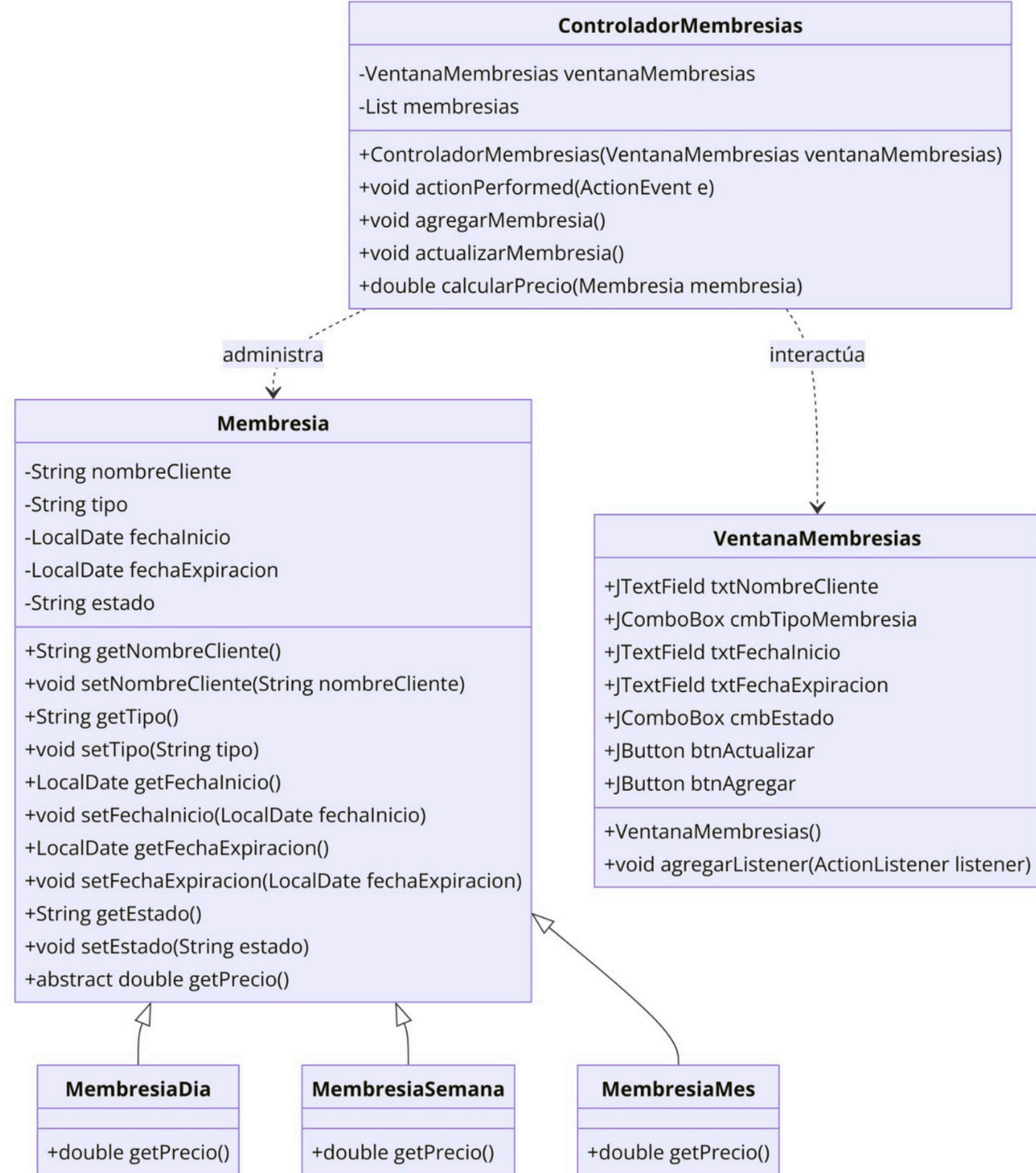
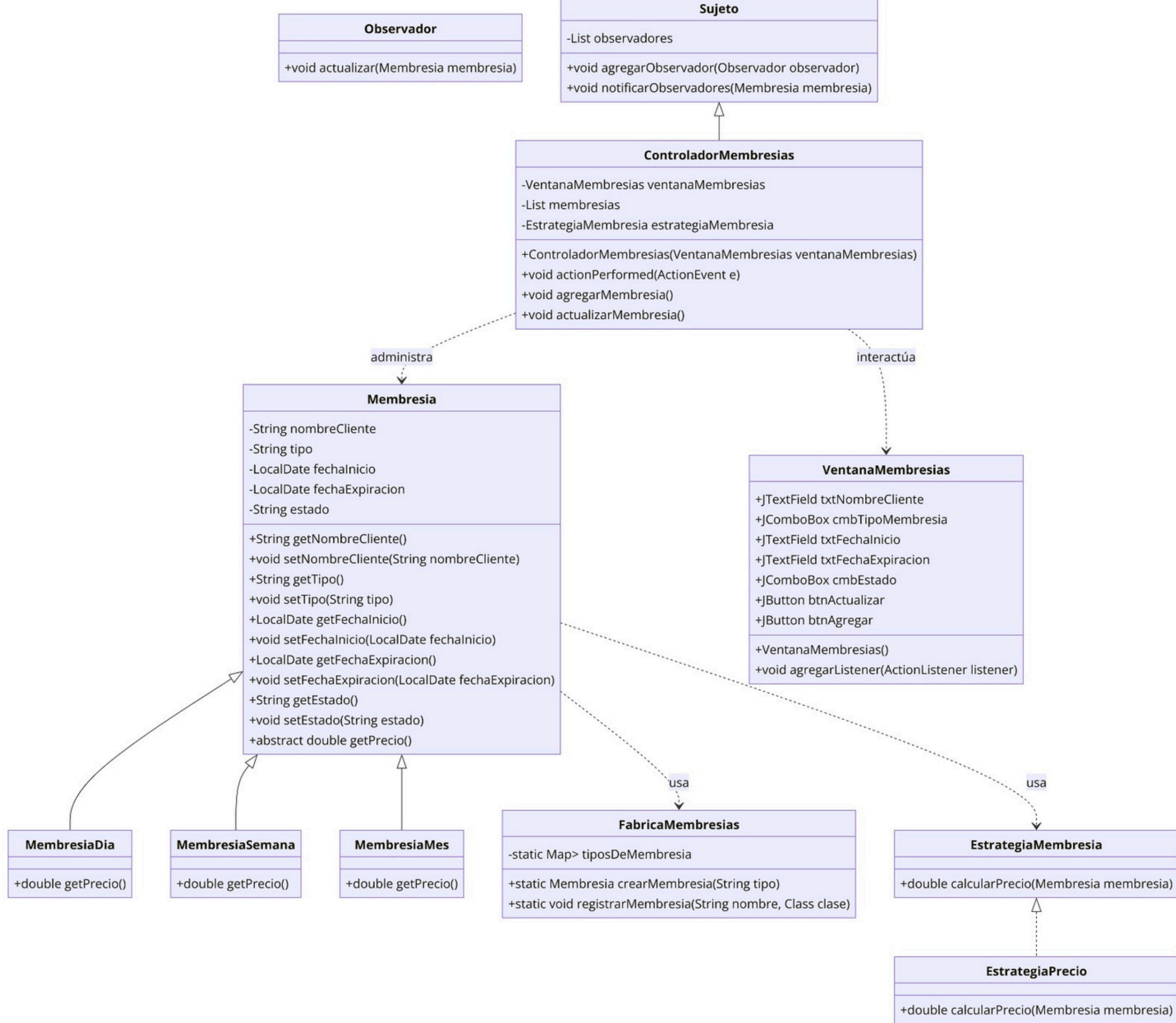


Diagrama de clases actual (con patrones)



Al contrario del diagrama anterior, este diagrama contiene 3 patrones de diseño, los cuales son Observer, así como Factory Method, combinados con Strategy; es flexible, ya que permite crear membresías nuevas, por poner un ejemplo.





Secciones de código



Code: Factory Method

```
package modelo;

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

public class FabricaMembresias {
    private static final Map<String, Class<? extends Membresia>> tiposDeMembresia = new
HashMap<>();

    static {
        tiposDeMembresia.put("Día", MembresiaDia.class);
        tiposDeMembresia.put("Semana", MembresiaSemana.class);
        tiposDeMembresia.put("Mes", MembresiaMes.class);
        tiposDeMembresia.put("Personalizada", MembresiaPersonalizada.class);
    }

    public static Membresia crearMembresia(String tipo, double... parametros) {
        Class<? extends Membresia> clase = tiposDeMembresia.get(tipo);
        if (clase != null) {
            try {
                if ("Personalizada".equals(tipo) && parametros.length > 0) {
                    return new MembresiaPersonalizada(parametros[0]);
                }
                return clase.getDeclaredConstructor().newInstance();
            } catch (InstantiationException | IllegalAccessException | NoSuchMethodException |
java.lang.reflect.InvocationTargetException e) {
                e.printStackTrace();
            }
        }
        throw new IllegalArgumentException("Tipo de membresía no soportado");
    }

    public static void registrarMembresia(String nombre, Class<? extends Membresia> clase) {
        tiposDeMembresia.put(nombre, clase);
    }
}
```



Code : Strategy

```
1 package modelo;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class FabricaMembresias {
7     private static final Map<String, Class<? extends Membresia>> tiposDeMembresia = new
HashMap<>();
8
9     static {
10         tiposDeMembresia.put("Día", MembresiaDia.class);
11         tiposDeMembresia.put("Semana", MembresiaSemana.class);
12         tiposDeMembresia.put("Mes", MembresiaMes.class);
13         tiposDeMembresia.put("Personalizada", MembresiaPersonalizada.class);
14     }
15
16     public static Membresia crearMembresia(String tipo, double... parametros) {
17         Class<? extends Membresia> clase = tiposDeMembresia.get(tipo);
18         if (clase != null) {
19             try {
20                 if ("Personalizada".equals(tipo) && parametros.length > 0) {
21                     return new MembresiaPersonalizada(parametros[0]);
22                 }
23                 return clase.getDeclaredConstructor().newInstance();
24             } catch (InstantiationException | IllegalAccessException | NoSuchMethodException |
java.lang.reflect.InvocationTargetException e) {
25                 e.printStackTrace();
26             }
27         }
28         throw new IllegalArgumentException("Tipo de membresía no soportado");
29     }
30
31     public static void registrarMembresia(String nombre, Class<? extends Membresia> clase) {
32         tiposDeMembresia.put(nombre, clase);
33     }
34 }
```



Code : Strategy

```
1  package modelo;
2
3  public interface EstrategiaMembresia {
4      double calcularPrecio(Membresia membresia);
5  }
6
```

```
1  package modelo;
2
3  public class EstrategiaPrecio implements EstrategiaMembresia {
4      @Override
5      public double calcularPrecio(Membresia membresia) {
6          return membresia.getPrecio();
7      }
8  }
```




Code : Observer

```
1  package modelo;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Sujeto {
7      private List<Observador> observadores = new ArrayList<>();
8
9      public void agregarObservador(Observador observador) {
10         observadores.add(observador);
11     }
12
13     public void notificarObservadores(Membresia membresia) {
14         for (Observador observador : observadores) {
15             observador.actualizar(membresia);
16         }
17     }
18 }
```



Code : Observer

```
1 package modelo;  
2  
3 public interface Observador {  
4     void actualizar(Membresia membresia);  
5 }
```



Gracias!!!

