PATRÓN FLYWEIGHT

Nombre: Solis Ruiz Adrián

No. de Control: 20210641

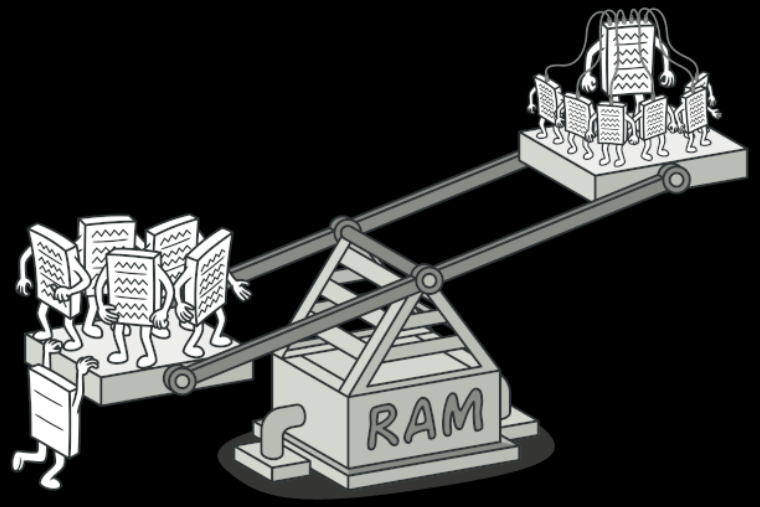
Materia: Patrones de Diseño de Software

INTRODUCCIÓN

- Los patrones de diseño son soluciones probadas y probadas para problemas comunes en el desarrollo de software. Estas soluciones encapsulan las mejores prácticas y experiencias acumuladas de desarrolladores a lo largo del tiempo. A continuación se hablara de un patrón de diseño de la categoría estructural que es el patrón flyweight, un patrón estructural que se utiliza para optimizar el rendimiento y la eficiencia en sistemas donde se manejan grandes cantidades de objetos similares.

¿QUÉ ES EL PATRÓN FLYWEIGHT?

- El patrón de diseño Flyweight es un patrón estructural utilizada en ingeniería de software para optimizar el rendimiento y la eficiencia en sistemas que manejan grandes cantidades de objetos similares. Este patrón se centra en la conservación de memoria al compartir datos comunes entre múltiples objetos, en lugar de replicar esos datos para cada objeto individualmente.
- La idea clave detrás del patrón Flyweight es dividir los objetos en dos partes: el estado intrínseco y el estado extrínseco. El estado intrínseco representa los datos comunes y compartidos entre varios objetos, mientras que el estado extrínseco contiene información única para cada objeto.



IMPORTANCIA DEL PATRÓN FLYWEIGHT

- Este patrón tiene varias razones del porque es importante:
- Optimización del uso de memoria: Este patrón ayuda a reducir la cantidad de memoria utilizada en la aplicación al compartir datos comunes entre múltiples objetos en lugar de replicarlos.
- Mejora del rendimiento: Al reducir la cantidad de datos duplicados y compartir información común, el patrón Flyweight puede mejorar significativamente el rendimiento de la aplicación. Esto se debe a que se minimiza el tiempo de creación y manipulación de objetos, lo que lleva a una ejecución más rápida de la aplicación en general.
- Facilita la escalabilidad: Al reducir la carga en la memoria y mejorar el rendimiento, el patrón Flyweight facilita la escalabilidad de la aplicación. Esto significa que la aplicación puede manejar un mayor volumen de datos y usuarios sin comprometer su rendimiento o consumir recursos excesivos.

APLICABILIDAD

Se utiliza este patrón cuando se cumplan estas características:

- Se utiliza un gran numero de objetos.
- El coste de almacenamiento es alto debido a la cantidad de objetos.
- La mayoría de los estados de los objetos pueden ser creados como comunes.
- Muchos objetos pueden ser reemplazados por unos pocos una vez que han sido borrados los estados no comunes.
- La mayor parte del estado del objeto puede ser extrínseco.

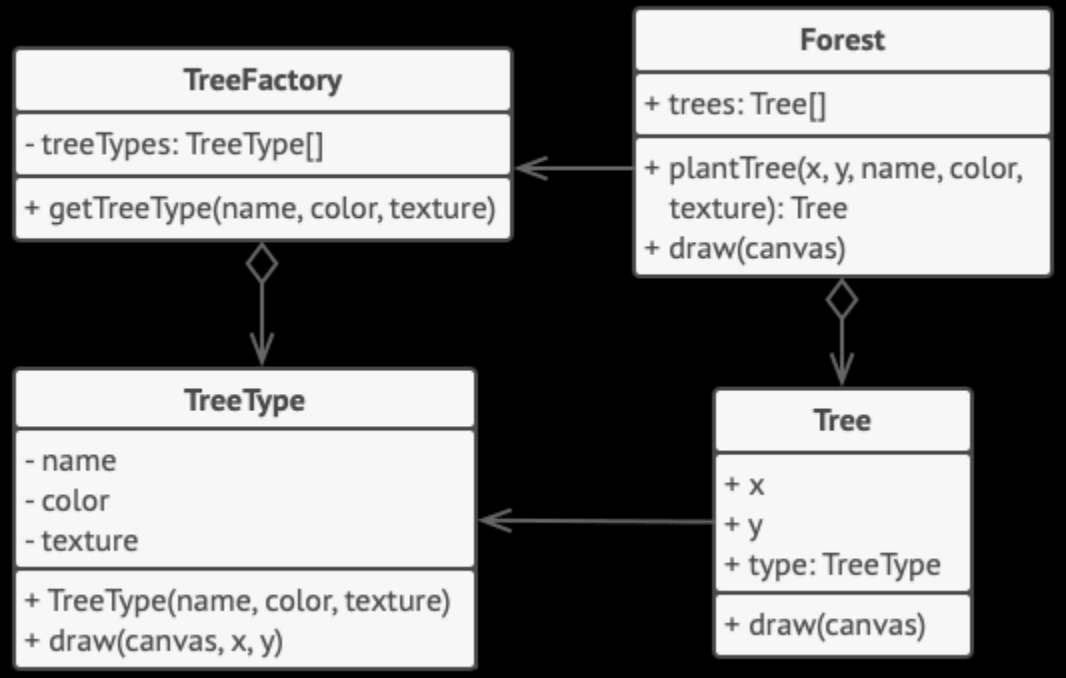
CARACTERÍSTICAS

- **Optimización de memoria:** Una de las características más destacadas del patrón Flyweight es su capacidad para reducir el consumo de memoria al compartir datos comunes entre múltiples objetos en lugar de replicarlos. Esto es útil en situaciones donde se manejan grandes cantidades de objetos similares.
- **Compartición de datos:** El patrón Flyweight permite compartir datos comunes (estado intrínseco) entre múltiples objetos, lo que lleva a una reducción significativa en la cantidad de datos duplicados en memoria. Esto ayuda a mejorar la eficiencia del sistema y a reducir el uso de recursos.
- **Separación de estado intrínseco y extrínseco:** Los objetos Flyweight se dividen en dos partes: estado intrínseco y estado extrínseco. El estado intrínseco, que es compartido entre varios objetos, contiene datos comunes y no cambia entre instancias, mientras que el estado extrínseco se mantiene fuera del objeto y varía para cada instancia.
- **Creación eficiente de objetos:** El patrón Flyweight se combina frecuentemente con un patrón de fábrica para gestionar la creación y recuperación de objetos Flyweight. Esto garantiza que los objetos se creen de manera eficiente y se reutilicen cuando sea posible.
- **Mejora del rendimiento:** Al reducir la cantidad de datos duplicados y compartir información común entre objetos, el patrón Flyweight puede mejorar significativamente el rendimiento de una aplicación al minimizar la carga en la memoria y reducir el tiempo de acceso a los objetos.

VENTAJAS Y DESVENTAJAS

- Ventajas:
- Produce ahorro de la capacidad almacenamiento
- Reduce el número total de objetos
- Reduce en gran cantidad el peso de los datos en un servidor.

- Desventajas:
- Consume un poco mas de tiempo para realizar las búsquedas.



En este ejemplo, el patrón **Flyweight** ayuda a reducir el uso de memoria a la hora de representar millones de objetos de árbol en un lienzo.

¿COMO FUNCIONA?

1.- Divide los campos de una clase que se convertirá en flyweight en dos partes:

- el estado intrínseco: los campos que contienen información invariable duplicada a través de varios objetos.
- el estado extrínseco: los campos que contienen información contextual única de cada objeto.

2.- Deja los campos que representan el estado intrínseco en la clase, pero asegúrate de que sean inmutables. Deben llevar sus valores iniciales únicamente dentro del constructor.

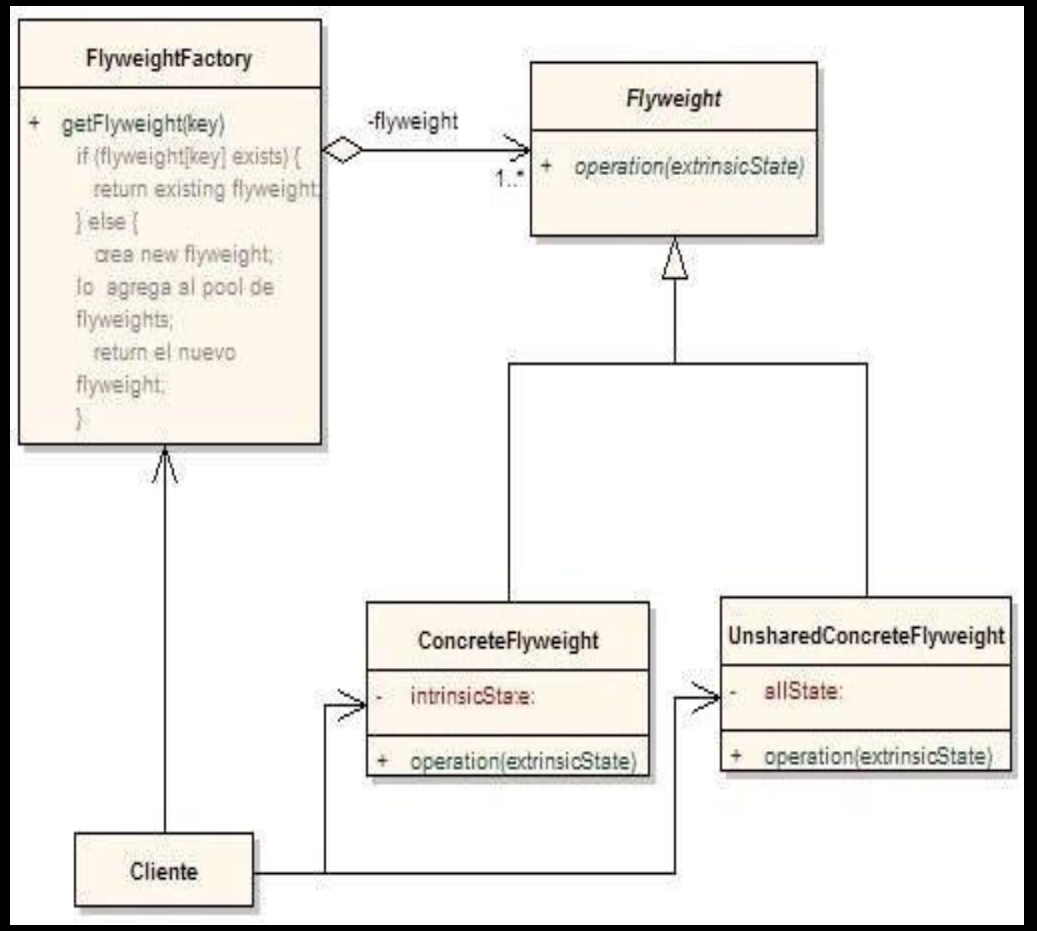
3.- Repasa los métodos que utilizan campos del estado extrínseco. Para cada campo utilizado en el método, introduce un nuevo parámetro y utilízalo en lugar del campo.

4.- Opcionalmente, crea una clase fábrica para gestionar el grupo de objetos flyweight, buscando uno existente antes de crear uno nuevo. Una vez que la fábrica esté en su sitio, los clientes sólo deberán solicitar objetos flyweight a través de ella. Deberán describir el flyweight deseado pasando su estado intrínseco a la fábrica.

5.- El cliente deberá almacenar o calcular valores del estado extrínseco (contexto) para poder invocar métodos de objetos flyweight. Por comodidad, el estado extrínseco puede moverse a una clase contexto separada junto con el campo referenciador del flyweight.

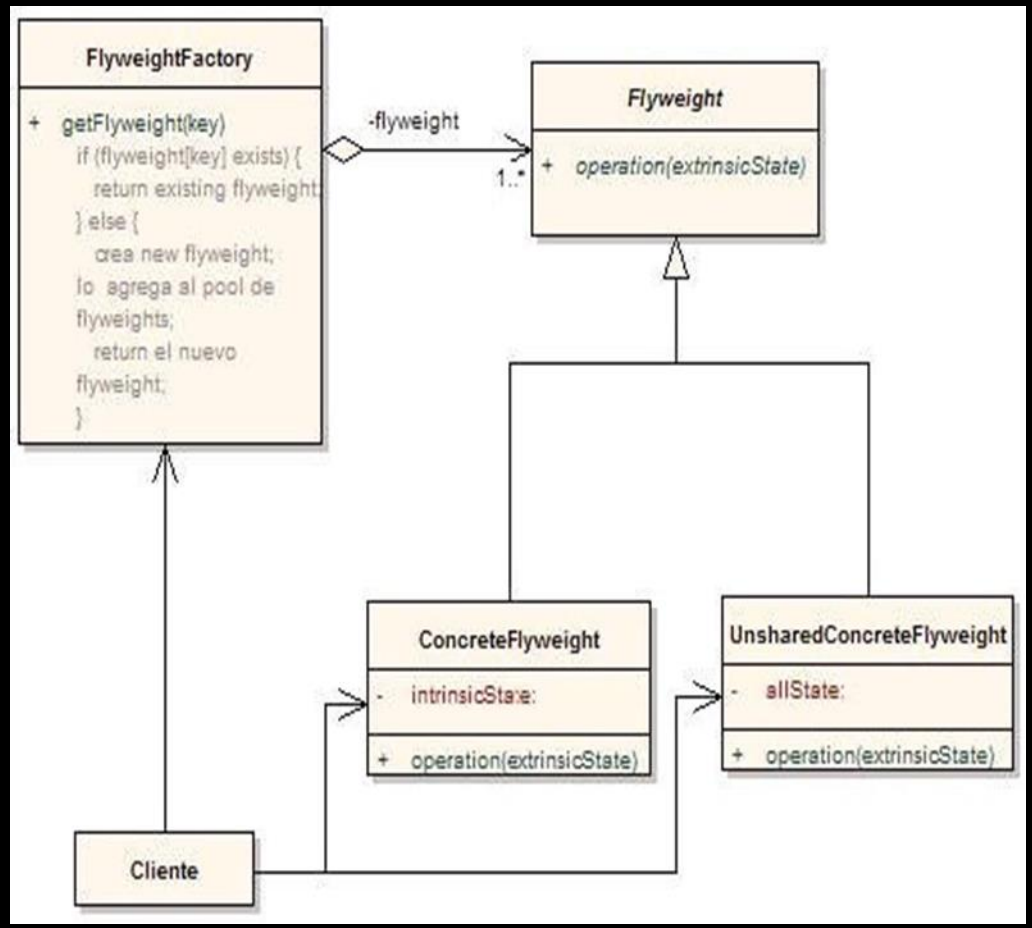
ESTRUCTURA

- **Flyweight:** Declaran una interface a través de la que flyweights pueden recibir y actuar sobre estados no compartidos.
- **ConcreteFlyweight:** Implementa la interfaz Flyweight y almacena los estados compartidos, si los hay. Un objeto ConcreteFlyweight debe ser compartible. Cualquier estado que almacene debe ser intrínseco; es decir, debe ser independiente de su contexto.
- **UnsharedConcreteFlyweight:** No todas las subclases de Flyweight tienen por qué ser compartidas. La interfaz Flyweight permite que se comparta; no lo fuerza. Es común que los objetos de esta clase tengan hijos de la clase ConcreteFlyweight en algún nivel de su estructura.



ESTRUCTURA

- **FlyweightFactory:** Crea y gestiona los objetos flyweight; garantiza que los objetos flyweight se comparten de forma apropiada. Cuando un cliente solicita un flyweight, el objeto de la clase FlyweightFactory proporciona una instancia existente, o crea una.
- **Client:**Contiene referencias a los flyweights, calculando o almacenando los estados no compartidos de los flyweights.



IMPLEMENTACIÓN EN CÓDIGO

Implementación en Python

```
class Flyweight:
    def __init__(self, shared_state):
        self._shared_state = shared_state
    def operation(self, unique_state):
        # Realizar operaciones con el estado compartido y el estado único
        pass
class FlyweightFactory:
    _flyweights = {}
    def get_flyweight(self, shared_state):
        if shared_state not in self._flyweights:
            self._flyweights[shared_state] = Flyweight(shared_state)
        return self._flyweights[shared_state]
# Uso del patrón
factory = FlyweightFactory()
fw = factory.get_flyweight("Shared")
fw.operation("Unique")
```

IMPLEMENTACIÓN EN JAVA

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

class Flyweight {
    private String sharedState;

    public Flyweight(String sharedState) {
        this.sharedState = sharedState;
    }
    public void operation(String uniqueState) {
        // Realizar operaciones con el estado compartido y el estado único
    }
}
class FlyweightFactory {
    private Map<String, Flyweight> flyweights = new HashMap<>();

    public Flyweight getFlyweight(String sharedState) {
        flyweights.putIfAbsent(sharedState, new Flyweight(sharedState));
        return flyweights.get(sharedState);
    }
}
// Uso del patrón
FlyweightFactory factory = new FlyweightFactory();
Flyweight fw = factory.getFlyweight("Shared");
fw.operation("Unique");
```

IMPLEMENTACIÓN EN C#

```
using System;
using System.Collections.Generic;

class Flyweight {
    private string _sharedState;

    public Flyweight(string sharedState) {
        _sharedState = sharedState;
    }

    public void Operation(string uniqueState) {
        // Realizar operaciones con el estado compartido y el estado único
    }
}

class FlyweightFactory {
    private Dictionary<string, Flyweight> _flyweights = new Dictionary<string, Flyweight>();

    public Flyweight GetFlyweight(string sharedState) {
        if (!_flyweights.ContainsKey(sharedState)) {
            _flyweights[sharedState] = new Flyweight(sharedState);
        }
        return _flyweights[sharedState];
    }
}
// Uso del patrón
FlyweightFactory factory = new FlyweightFactory();
Flyweight fw = factory.GetFlyweight("Shared");
fw.Operation("Unique");
```