I. INFORMACION GENERAL			
TEMA:	Patrón Composite y Flyweight		
ASIGNATURA	Patrones de Software	**NIVEL**	Quinto “A”
UOC	Básica	**CICLO ACADEMICO**	Abril 2023 – Septiembre 2023
DOCENTE	Ing. Marco Guachimboza, Mg.		**FECHA:** 15/06/2023
INTEGRANTES	Caiza Ángel, Pincha Diego, Vichicela Kevin		

II. DESARROLLO

Patrón Composite

El patrón compuesto es uno de los 23 patrones de diseño "GoF" para el desarrollo de software que fueron publicados en 199
por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, la llamada "pandilla de los cuatro". “O grupos de cuatro. Al igual que el patrón Facade and Ornament, es un patrón de diseño que agrupa objetos y capas complejos en estructuras más grandes.
El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que, al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

Aplicabilidad

Utilice el patrón compuesto cuando necesite implementar una estructura de árbol de objetos. El patrón Composite le brinda dos tipos básicos de elementos que comparten el mismo aspecto: hojas simples y contenedores complejos. Un contenedor puede incluir platos y otros contenedores. Esto le permite construir una estructura de árbol de objetos recursivos anidados. Utilice patrones cuando desee que el código del cliente trate elementos simples y complejos de la misma manera. Todos los elementos definidos por el patrón compuesto comparten la misma interfaz. Al usar esta interfaz, el cliente no tiene que preocuparse por la clase concreta de los objetos con los que está trabajando.

Pros

Estructura jerárquica: Simplifica el diseño y permite que los clientes trabajen con estructuras complejas de manera transparente.
Flexibilidad: Permite agregar o quitar objetos individuales o compuestos en tiempo de ejecución sin afectar el código existente. Esto proporciona una mayor flexibilidad para modificar la estructura de los objetos sin impactar el resto del sistema.
Operaciones uniformes: Esto simplifica el código del cliente, ya que no es necesario diferenciar entre los dos tipos de objetos al realizar operaciones.
Extensibilidad: permite agregar nuevas clases de objetos individuales o compuestos sin cambiar el código existente. Esto facilita la extensibilidad del sistema y permite un mayor modularidad.

Contras

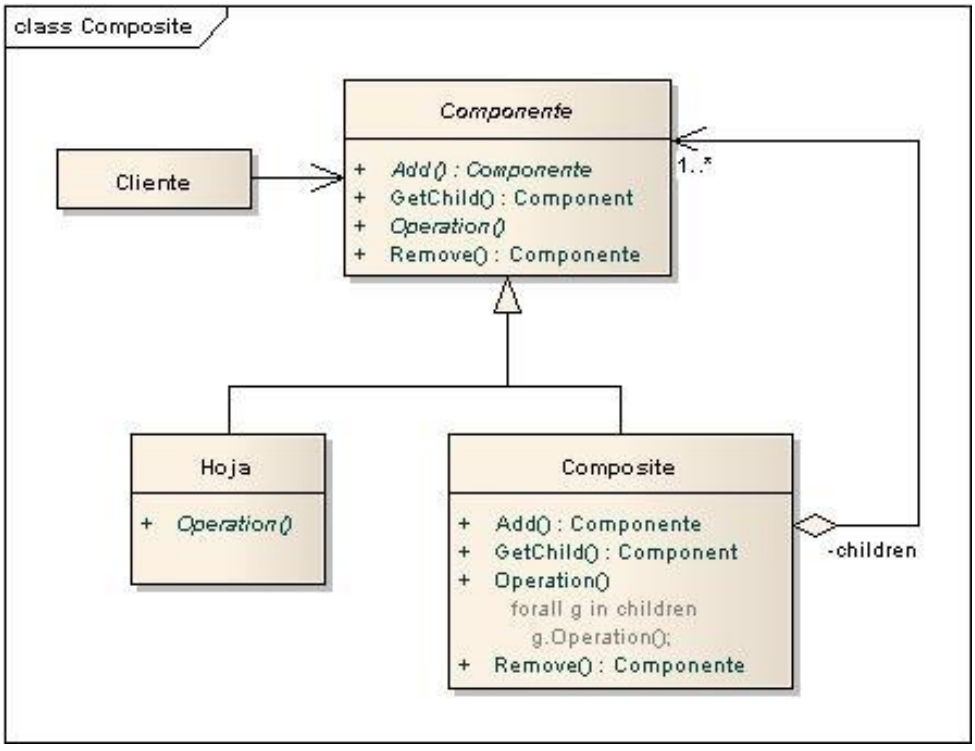
Complejidad potencial: A medida que aumenta la complejidad de la jerarquía de objetos, puede volverse más difícil de entender y mantener el código del patrón Composite.
Dificultad en la eliminación de objetos: Si se necesita eliminar un objeto de la estructura compuesta, puede ser complicado, ya que esto puede requerir la reorganización de los objetos restantes.
Rendimiento: En algunas implementaciones, el patrón Composite puede tener un impacto en el rendimiento debido a la necesidad de recorrer toda la estructura jerárquica para realizar operaciones en los objetos.

Relaciones con otros patrones

Puedes utilizar Builder al crear árboles Composite complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.

Patrón Decorator: El patrón Decorator permite añadir funcionalidad adicional a un objeto de manera dinámica. En el contexto del patrón Composite, se puede utilizar el patrón Decorator para agregar comportamientos específicos a los elementos de la estructura compuesta.
Patrón Iterator: El patrón Iterator permite recorrer los elementos de una colección de objetos de manera secuencial sin exponer su estructura interna. En el contexto del patrón Composite, se puede utilizar el patrón Iterator para recorrer de forma recursiva los elementos de una estructura compuesta.
Patrón Visitor: El patrón Visitor permite definir operaciones o comportamientos que se pueden aplicar a una estructura de objetos sin modificar las clases de dichos objetos. En el contexto del patrón Composite, se puede utilizar el patrón Visitor para realizar operaciones específicas en los elementos de la estructura compuesta.

Estructura



Ejemplo

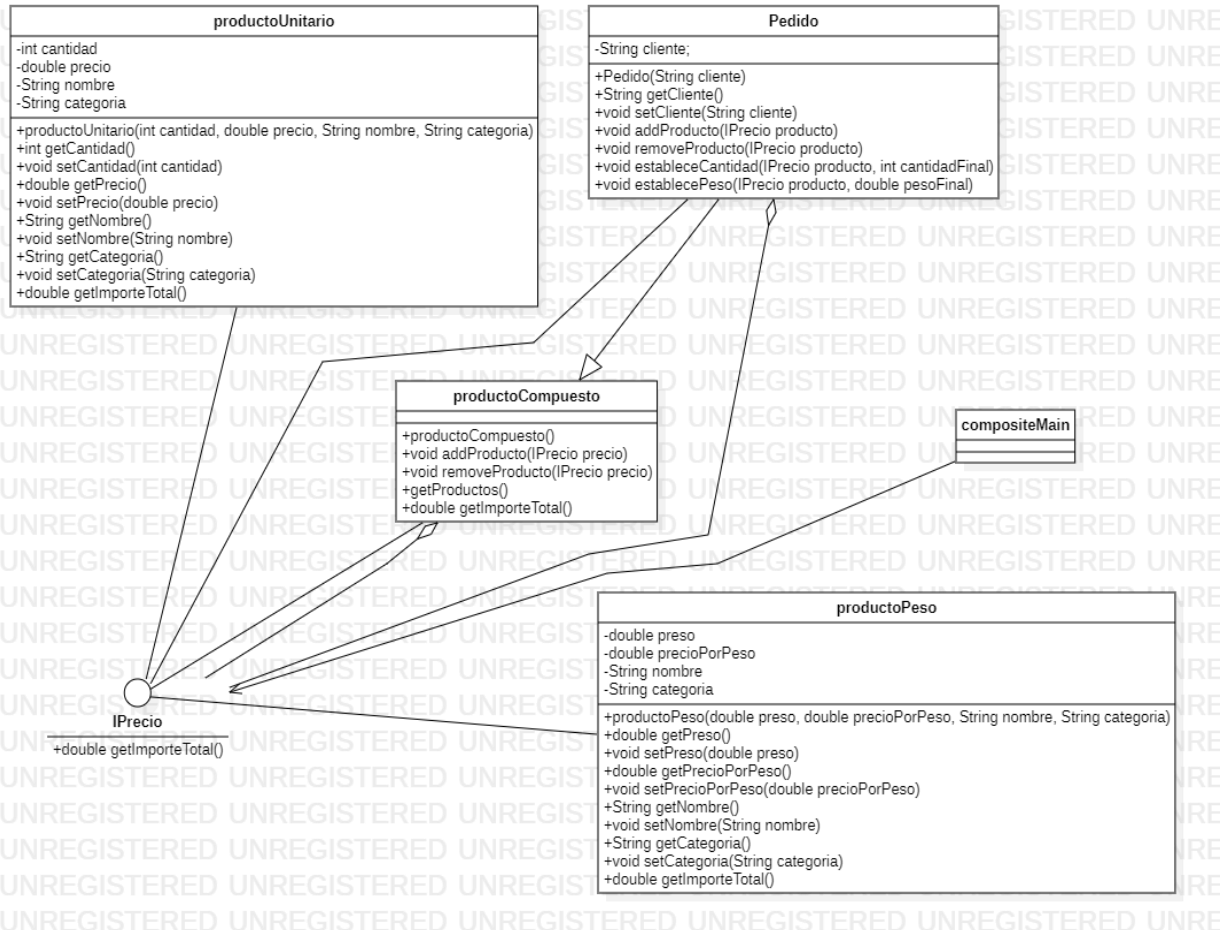
Se desea diseñar un sistema de clases que representan distintos productos y pedidos implementando el patrón Composite.

En la interfaz, se debe definir el comportamiento como se indicaba con anterioridad, estableciendo un método getImporteTotal() que como aparece en la documentación devuelve el precio del elemento/s.

Cada una de las clases que implementa esta interfaz son hojas del patrón. En este caso hemos distinguido 3 clases:

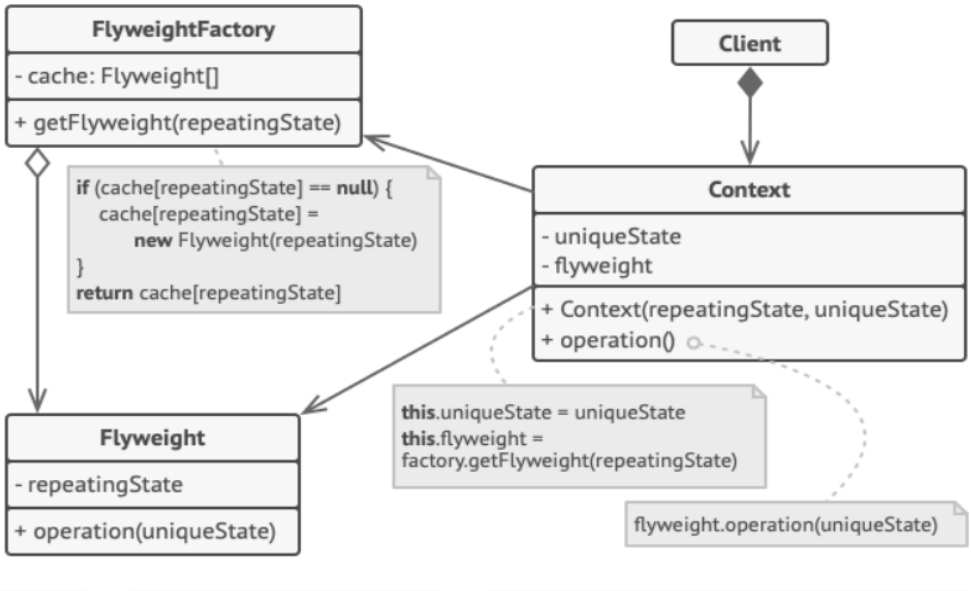
- **ProductoUnitario**: representará productos cuya venta se realiza por unidades indivisibles. Estos productos tendrán una serie de características y una forma específica de calcular el precio.
- **ProductoPeso**: representará productos cuya venta depende del peso de este (e.g.: un jamón). Como el anterior, tendrá una serie de propiedades y otra manera distinta de calcular el precio.
- **ProductoCompuesto**: representará un grupo de productos cuya venta se realiza de forma conjunta.

Como último componente de este patrón vemos la clase Pedido, que es una especialización de la clase ProductoCompuesto que añadirá una serie de propiedades y unas funcionalidades añadidas.



Patrón FlyWeight

El patrón Flyweight o conocido como el de peso ligero tiene como objetivo minimizar el uso de recurso de la computadora para sacar un aprovecho máximo de estos.
Además, ayuda reducir la necesidad de almacenamiento, evitando la duplicidad de los objetos.
Este patrón maneja dos estados:
Intrínseco: esta se almacena en el peso ligero, información independiente del contexto, lo que hace compartible.
Extrínseco: este depende y varia del contexto, esto no puede ser compartida
Estructura:



Consecuencias:

- Consume un poco más de tiempo para realizar las búsquedas.
- Aumenta la complejidad de los objetos.
- Aumenta el número de clases del sistema.

Flyweight:

- Guarda el estado intrínseco del objeto original que se comparte a todos los objetos.

Context

- Esta contiene el estado extrínseco, único en todos los objetos.

FlywieghtFactory

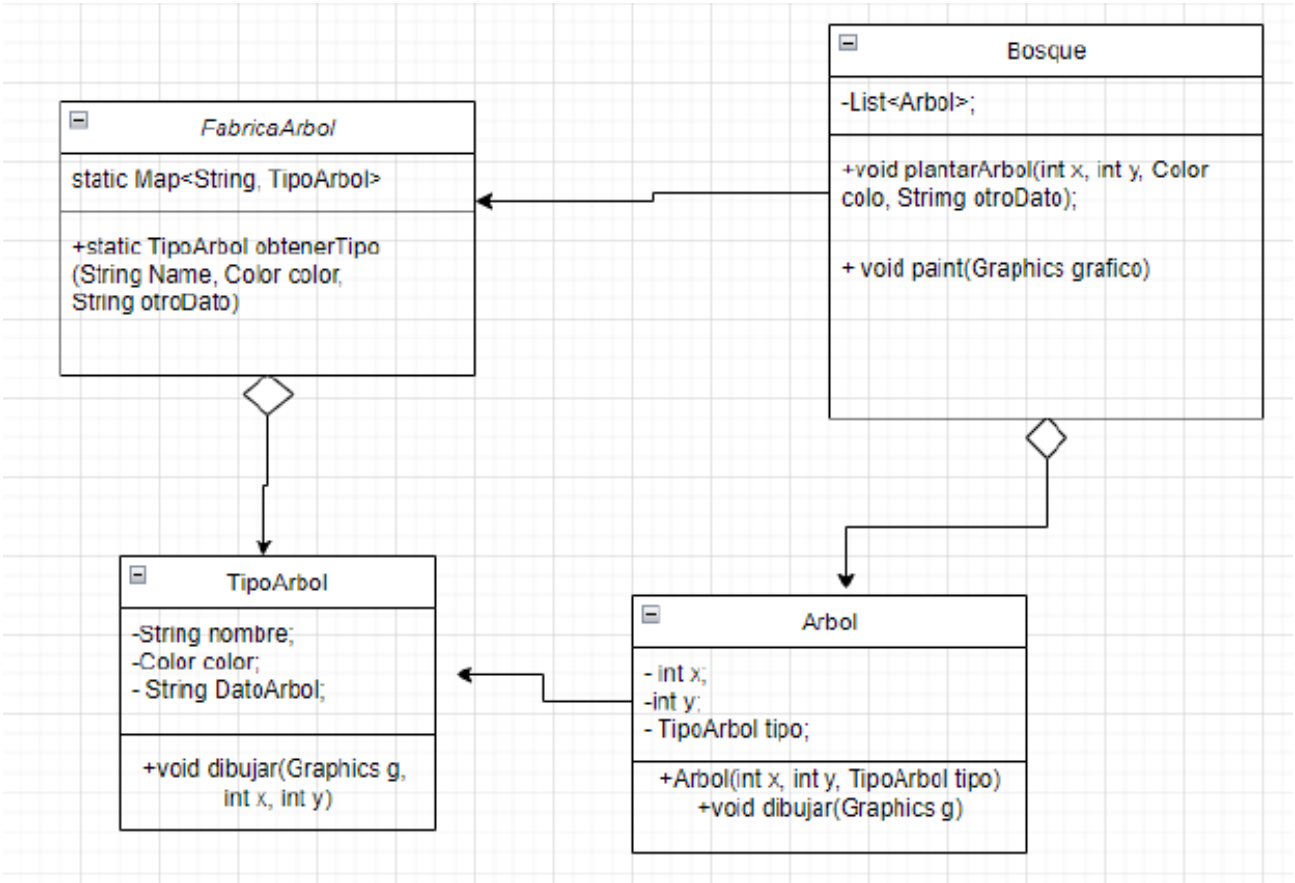
- Control y gestión de un grupo de objetos flyweight existentes.
- Controla que no se creen objetos que ya existen.

Client

- Calcula o almacena el estado extrínseco de los objetos.
- Mantiene un estado no compartido y colección de flyweight.

Ejemplo:

Se desea disminuir el consumo de memoria al momento de crear miles de árboles que son objetos en una interfaz de java.



Primero creamos la Clase Árbol:
Contendrá como atributos tendrá las coordenadas y el Tipo de árbol que se creará.
Esta contiene un estado único para cada árbol como son las coordenadas (Estado extrínseco).
Además de un constructor y el método Dibujar que colocara a cada tipo de árbol en la coordenada indicada.

```
14    public class Arbol {
15        private int x;
16        private int y;
17         private TipoArbol tipo;
18
19        public Arbol(int x, int y, TipoArbol tipo) {
20            this.x = x;
21            this.y = y;
22            this.tipo = tipo;
23        }
24
25         public void dibujar(Graphics g){
26             tipo.dibujar(g, x, y);
27         }
28
29    }
```

Tipo Árbol

Esta contendrá los atributos que podrán ser compartidos por varios árboles (estado intrínseco)
Tendrá los atributos como:
El nombre, el color y un dato adicional del árbol que pueden ser compartidas por varios árboles.
Tiene un constructor.

```
15    public class TipoArbol {
16        private String nombre;
17        private Color color;
18        private String otrodatoArbol;
19
20        public TipoArbol(String nombre, Color color, String otrodatoArbol) {
21            this.nombre = nombre;
22            this.color = color;
23            this.otrodatoArbol = otrodatoArbol;
24        }
25
26        public void  dibujar( Graphics g, int x, int y){
27            g.setColor((Color.BLUE));
28            g.fillRect(x-1, y, 3, 5);
29            g.setColor(color);
30            g.fillOval(x-5, y-10, 10, 10);
31        }
```

El método dibujar, permitirá dibujar en el lienzo la colección de arboles en las diferentes coordenadas.

FabricaArboles

Tiene un atributo de lista de tipo Map en el cual guardara la colección de Tipo de árbol y el nombre del mismo.
Un método estático que controlara y verificara si aun no se encuentra instanciado el objeto para no realizar la duplicidad.

```java
public class FabricaArboles {
    static  Map<String, TipoArbol> tipodArbol= new HashMap<>();


    public static TipoArbol obtenerelTipo( String nombre, Color color, String otrodatoArbol){
        TipoArbol resultado = tipodArbol.get(nombre);
        if (resultado == null) {
            resultado= new TipoArbol(nombre, color, otrodatoArbol);
            tipodArbol.put(nombre, resultado);

        }
        return resultado;
    }
}
```

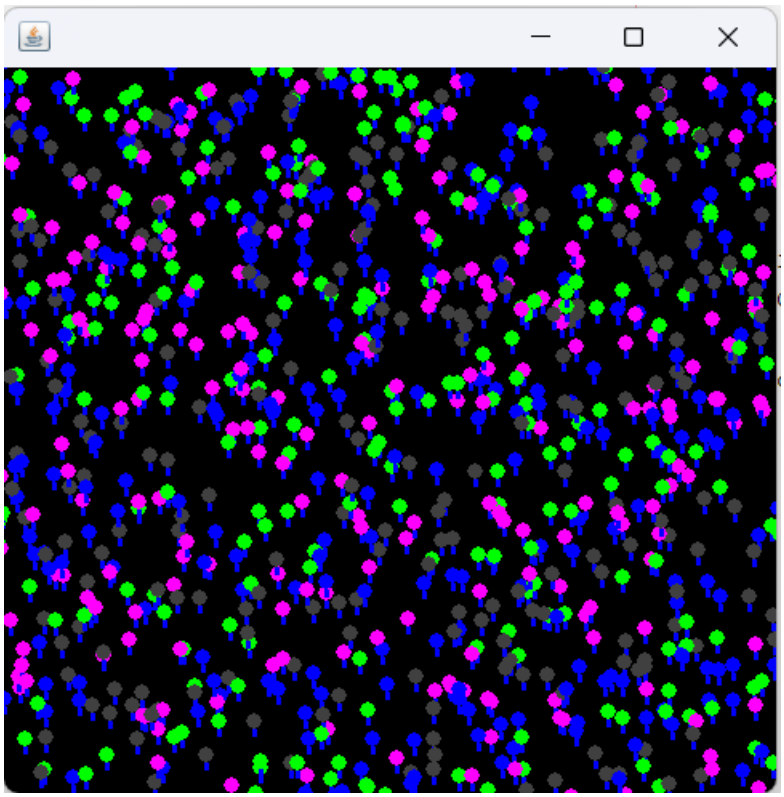
Bosque que se extiende de la clase JFrame:
Tiene como atributo la una lista de Árboles, en el cual se almacena los árboles.
Además, un método que no devuelve nada, pero que almacena cada árbol, primero verifica si existe o no el árbol y devuelve que tipo de árbol es, luego se crea un nuevo árbol con las nuevas coordenadas y el tipo de árbol.
Y el método Paint que permite graficar en el lienzo del JFrame cada árbol.

```java
public class Bosque extends JFrame{

    private List<Arbol> arboles= new ArrayList<Arbol>();

    public void plantarArbol(int x, int y, String nombre, Color color, String OtrodatoArbol){
        TipoArbol tipo = FabricaArboles.obtenerelTipo(nombre, color, OtrodatoArbol);
        Arbol arbol= new Arbol(x, y, tipo);
        arboles.add(arbol);

    }
    @Override
    public void paint( Graphics grafico){
        for (Arbol arbol: arboles) {
            arbol.dibujar(grafico);

        }
    }
```

La clase Cliente o Prueba:
Definimos variables estáticas
La primera será el tamo de la ventana
Y luego el número de árboles que queramos
Declaramos una instancia del Objeto de Bosque
Dentro de un for hacemos un ciclo en donde se instancia 4 tipo de árboles las mismas que serán copiadas y dibujadas en diferentes coordenadas del lienzo del JFrame.

```java
public class Prueba {

    static int tamañoVenta = 500;
    static int arboles_a_dibujar = 1000;
    static int tipoArboles = 4;

    public static void main(String[] args) {
        Bosque bosque= new Bosque();
        for (int i = 0; i < Math.floor(arboles_a_dibujar/tipoArboles); i++) {
            bosque.plantarArbol(random(0,tamañoVenta),random(0,tamañoVenta),"Roble",Color.GREEN,"La textura es de roble ");
            bosque.plantarArbol(random(0,tamañoVenta),random(0,tamañoVenta),"Eucalipto",Color.blue,"La textura es de Eucalipto ");
            bosque.plantarArbol(random(0,tamañoVenta),random(0,tamañoVenta),"Pino",Color.DARK_GRAY,"eXPORTADO DESDE EEUU");
            bosque.plantarArbol(random(0,tamañoVenta),random(0,tamañoVenta),"Ceprez",Color.MAGENTA,"Nuevo arbol ");

        }
        bosque.setSize(tamañoVenta, tamañoVenta);
        bosque.setVisible(true);


    }
        private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));
        }
```

Ejecución:



III. CONCLUSIONES DEL TRABAJO.

- El patrón composite nos sirve para representar jerarquías, unificar las operaciones entre los objetos y los objetos individuales con el fin de usarlos a todos de una sola manera para la simplificación de su uso, pero también está la desventaja de que al definir muchas veces un método o métodos generales que al implementar va a estar vacíos en algunos casos.

- El patrón Flyweight puede ser de gran utilidad ya que nos ayuda a bajar la cantidad de memoria ram que utiliza nuestro programa y eso es muy beneficioso ya que muchos usuarios pueden no tener un computador con mucha memoria ram y eso les puede impedir usar nuestro programa, pero al utilizar el patron flyweigth podemos hacer que nuestro programa llegue a muchas mas personas que tengan dispositivos con bajos recursos, además ayuda a que nuestro programa este mucho mejor optimizado.

IV. BIBLIOGRAFÍA:

(web, 2020) https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/patron-composite/
[1] A. Shvets, Sumérgete en los PATRONES DE DISEÑO, Refactorin.Guru, 2019.
[2] O. Blancarte, Introducción a Los Patrones de Diseño: Un Enfoque Práctico, CreateSpace Independent Publishing Platform, 2016, 2016.