PATRÓN COMPOSITE

Cuando necesitamos objetos organizados en forma de árbol, este patrón nos da una estructura básica que nos servirá tanto para crear como para manipular el árbol. Este patrón supone que los nodos son de naturaleza homogénea, del mismo tipo, lo que permite su visión recursiva y pone un poco de orden para no crear el árbol a lo loco insistiendo en que debe existir un *interface* o *clase abstracta* que define el comportamiento genérico de un nodo.

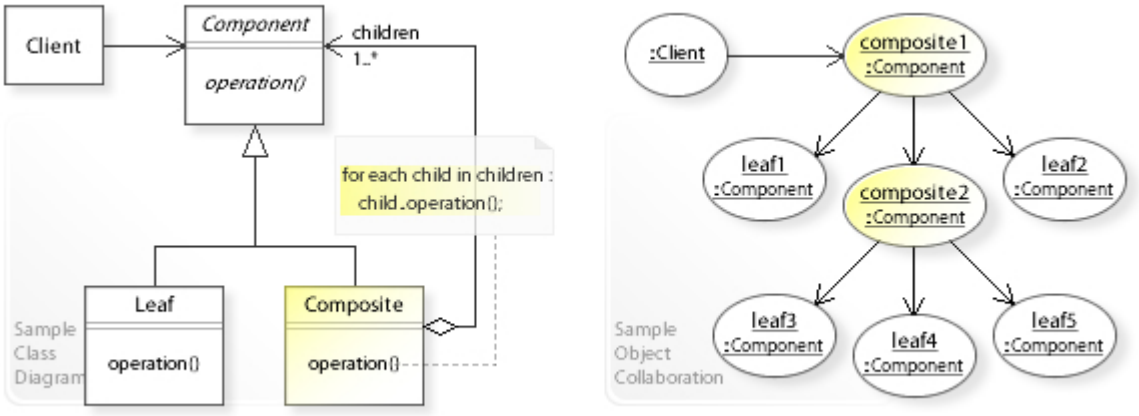
Los árboles de objetos se usan mucho por ejemplo en videojuegos para generar capas y en general en programación gráfica en la que unos objetos contienen a otros y se enganchan formando un árbol de objetos.

La clave de este patrón es la unión de dos conceptos: composición + recursividad

Hay tres entidades implicadas

- **Componente (Component):** es **una interfaz o una clase *abstracta*** que es un supertipo para todo el conjunto de clases
- **Clase compuesta(Composite):** define una implementación concreta de *Componente* y contiene referencias a otros componentes.
- **Clase de hoja(leaf):** una hoja representa un tipo no compuesto, es decir, no está compuesta por otros componentes, no usa composición

Observa el UML de diagrama de clases con las tres entidades anteriores y a su derecha un posible árbol de objetos generado con las clases del UML.



Además, aunque no se refleja en el UML de arriba, será muy habitual, pero no obligatorio, que la clase compuesta tenga un atributo *List* o similar para referenciar a otros nodos (composición) y un método add() y remove() para actualizarlo. Las hojas no tendrán este atributo.

El patrón Composite genera un árbol n-ario.
Recuerda las estructuras dinámicas de árbol. Comenzamos con árboles binarios (un padre y dos hijos) y finalmente vimos un árbol n-ario(*1* padre y *n* hijos). El patrón composite no es más que una forma ordenada y standard de crear un árbol n-ario de objetos. Aunque en principio parece que usa complicaciones innecesarias esta estructura hace que el código sea más fácil de mantener.

La principal diferencia entre los árboles que vimos anteriormente y un árbol composite es que antes teníamos un único tipo de nodo, en composite hay dos tipos de nodos un tipo es para las hojas(nodos simples) y el otro para nodos compuestos. ¿Cómo sabíamos trabajando con un único tipo de nodo que un nodo es una hoja?, si todos sus enlaces a hijos valen null, entonces se trataba de un nodo hoja.

Composite y recursividad.
Observa que la clase Composite tiene una lista de referencias de tipo Componet y que Composite es un Component, por lo tanto, estamos a un caso de clases que contienen referencias a si mismo, y como vimos con los nodos de las listas y lo árboles y en este escenario, la recursividad suele jugar un papel importante para escribir métodos.

EJEMPLO:
Supongamos que queremos hacer un presupuesto. El presupuesto no es más que el despiece de una casa con sus precios. Es un ejemplo imaginario no un presupuesto real.
Casa:

- finca
    - cierre
    - jardín
- estructura
    - tejado
    - alturas
    - sótano
- interior.
    - habitaciones
        - mobiliario
        - pintura
    - electricidad
        - cables
        - operadores
    - fontanería
        - tuberías
        - calefacción
            - caldera
            - radiadores

Fíjate bien en lo natural que resulta escribir un método *getPrecio()* recursivo:

- El precio de la casa es la suma de los precios de la finca de la estructura y del interior.
- A su vez el precio de la finca es el precio del cierre y el jardín
- etc.

¡Por lo tanto recursivamente!:

- El precio de una parte compuesta de la casa es la suma de los precios de las subpartes de dicha parte. CASO RECURSIVO

quedaría

- Si una parte compuesta no tiene subpartes, es decir, por el momento está vacía no sumamos nada.. CASO BASE.
- Si una parte es simple (sin subpartes) devolvemos su precio. CASO BASE.

Observa la estructura en el árbol del despiece. Queremos tener la capacidad de calcular el precio de las diferentes partes. Las hojas tienen un precio, Y los precios de los objetos compuestos los vamos a calcular recursivamente. Observa que:

- El caso base *Si una parte compuesta no tiene subpartes,* simplemente ocurre cuando el for hace 0 iteraciones
- El caso base *Si una parte es simple,* se consigue sin usar el típico if ya que entra en acción el polimorfismo que provoca la invocación del getPrecio() de la parte simple.

```
import java.util.ArrayList;
import java.util.List;

abstract class ParteAbstracta {
    protected String nombre;

    protected ParteAbstracta(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    abstract double getPrecio();

}

class ParteCompuesta extends ParteAbstracta {
    private List<ParteAbstracta> partes = new ArrayList<>();

    public ParteCompuesta(String nombre) {
        super(nombre);
    }

    @Override
    public double getPrecio() {
        double precio = 0;
        for (ParteAbstracta parte : partes) {
            precio += parte.getPrecio();
```

```
        }
        return precio;
    }

    public void addParte(ParteAbstracta parte) {
        this.partes.add(parte);
    }

}

// las hojas del árbol, no hay composición
class ParteSimple extends ParteAbstracta {
    double precio;

    public ParteSimple(String nombre, double precio) {
        super(nombre);
        this.precio = precio;
    }

    @Override
    double getPrecio() {
        return precio;
    }
}

public class App {
    public static void main(String[] args) throws Exception {
        // finca
        ParteSimple cierre = new ParteSimple("Cierre finca", 4000);
        ParteSimple jardin = new ParteSimple("jardín", 1000);
        ParteCompuesta finca = new ParteCompuesta("finca");
        finca.addParte(cierre);
        finca.addParte(jardin);

        // estructura
        ParteSimple tejado = new ParteSimple("tejado", 10000);
        ParteSimple alturas = new ParteSimple("alturas", 10000);
        ParteSimple sotano = new ParteSimple("sótano", 10000);
        ParteCompuesta estructura = new ParteCompuesta("estructura");
        estructura.addParte(tejado);
        estructura.addParte(alturas);
        estructura.addParte(sotano);

        // interior
        // interior-habitaciones
        ParteSimple mobiliario = new ParteSimple("mobiliario", 20000);
        ParteSimple pintura = new ParteSimple("pintura", 10000);
        ParteCompuesta habitaciones = new ParteCompuesta("habitaciones");
        habitaciones.addParte(mobiliario);
        habitaciones.addParte(pintura);

        // interior-electricidad
        ParteSimple cables = new ParteSimple("cables", 500);
        ParteSimple operadores = new ParteSimple("operadores", 500);
        ParteCompuesta electricidad = new ParteCompuesta("electricidad");
        electricidad.addParte(cables);
        electricidad.addParte(operadores);

        // interior-fontanería
        ParteSimple caldera = new ParteSimple("caldera", 4000);
        ParteSimple radiadores = new ParteSimple("radiadores", 2000);
        ParteCompuesta calefacción = new ParteCompuesta("calefacción");
```

```
        calefacción.addParte(caldera);
        calefacción.addParte(radiadores);
        ParteSimple tuberias = new ParteSimple("tuberías", 3000);
        ParteCompuesta fontaneria = new ParteCompuesta("fontanería");
        fontaneria.addParte(tuberias);
        fontaneria.addParte(calefacción);

        // todo interior
        ParteCompuesta interior = new ParteCompuesta("interior");
        interior.addParte(habitaciones);
        interior.addParte(electricidad);
        interior.addParte(fontaneria);

        // la casa completa
        ParteCompuesta casa = new ParteCompuesta("Casa");
        casa.addParte(finca);
        casa.addParte(estructura);
        casa.addParte(interior);

        // probar cálculo de costes
        System.out.println("Precio casa " + casa.getPrecio());
        System.out.println("Precio finca " + finca.getPrecio());

    }
}
```

Ejercicio U7_B4B_E1:

Añadir la capacidad de imprimir el presupuesto de toda la casa desglosado, que no es más que imprimir el árbol de nodos en preorden.

```
Casa 75000.0
        finca 5000.0
                Cierre finca 4000.0
                jardín 1000.0
        estructura 30000.0
                tejado 10000.0
                alturas 10000.0
                sótano 10000.0
        interior 40000.0
                habitaciones 30000.0
                        mobiliario 20000.0
                        pintura 10000.0
                electricidad 1000.0
                        cables 500.0
                        operadores 500.0
                fontanería 9000.0
                        tuberías 3000.0
                        calefacción 6000.0
                                caldera 4000.0
                                radiadores 2000.0
```

MENÚS DE CONSOLA Y PATRÓN COMPOSITE

Los típicos menús de consola tienen una estructura arborescente y se les puede aplicar el patrón composite para ampliar/quitar opciones fácilmente.

Ejercicio U7_B4B_E2: utilizando composite, crea una aplicación con menú de consola donde
la clase abstracta se llame -> ComponenteMenu
la clase compuesta se llame ->Menu
la clase simple se llame->MenuItem

Un menú puede contener sub menús pero finalmente llegaremos a los hojas que son de tipo MenuItem

Todos los objetos ComponenteMenu deben:

- tener un nombre
- un método abstracto ejecutar()

Sobre ejecutar(): su comportamiento será diferente dependiendo al tipo de objeto que pertenezca

- en un objeto Menu consistirá:
  - en imprimir las opciones
  - que el usuario escoja una opción
  - y por último lanzar la ejecución de la opción escogida. (Si se escogió otro menú, recursivamente ejecutamos el nuevo menú)
- en un objeto MenuItem, son hojas y lo único que hacen es simular una acción imprimiendo un mensaje "Ejecutando cosas de ..."

En el siguiente ejemplo se entra en la aplicación que simula un editor, se escoge el menú archivo, luego el item menu archivo y luego se va saliendo hacia atrás hasta salir de la aplicación y nos despedimos con "chao"

```
Menú mi editor
-------------------
0. archivo
1. Editar
2. Salir
Teclea número opcion:  0

Menú archivo
-------------------
0. Nuevo archivo
1. Abrir archivo
2. Guardar archivo
3. Salir
Teclea número opcion:  0
ejecutando cosas de Nuevo archivo
pulsa tecla para regresar a menú anterior

--------------------------------
```

```
Menú archivo
-------------------
0. Nuevo archivo
1. Abrir archivo
2. Guardar archivo
3. Salir
Teclea número opcion:  3

Menú mi editor
-------------------
0. archivo
1. Editar
2. Salir
Teclea número opcion:  2
Chao
```

Observa que tras ejecutarse un menuItem o un menú tenemos que volver al menú padré que lo invocó. Nos hará falta entonces tener un atributo Padre en los Componentes para almacenar la referencia al objeto Padre que nos permite "subir" por el árbol

Utiliza la siguiente clase abstracta para resolver el ejercicio:

```
abstract class ComponenteMenu {
   protected ComponenteMenu padre;
   protected String nombre;
   Scanner sc;

   protected ComponenteMenu(String nombre, Scanner sc) {
      padre = null;
      this.nombre = nombre;
      this.sc = sc;
   }

   abstract void ejecutar();

}
```

Tu solución debe funcionar al menos para los tests del siguiente main() que lanza una salida como la del ejemplo de salida de arriba.

```
public class App {
   public static void main(String[] args) throws Exception {
      Scanner sc = new Scanner(System.in);

      MenuItem nuevoarchivo = new MenuItem("Nuevo archivo", sc);
      MenuItem abrirarchivo = new MenuItem("Abrir archivo", sc);
      MenuItem guardararchivo = new MenuItem("Guardar archivo", sc);
      Menu archivo = new Menu("archivo",sc);
      archivo.addMenu(nuevoarchivo);
      archivo.addMenu(abrirarchivo);
      archivo.addMenu(guardararchivo);


      MenuItem copiar= new MenuItem("copiar", sc);
      MenuItem pegar= new MenuItem("pegar", sc);
      Menu editar= new Menu("Editar", sc);
      editar.addMenu(copiar);
      editar.addMenu(pegar);

      Menu MiMenu= new Menu("mi editor", sc);
```

```
        MiMenu.addMenu(archivo);
        MiMenu.addMenu(editar);

        MiMenu.ejecutar();

    }
}
```

En el siguiente ejemplo se puede intuir que la librería gráfica swing del JDK utiliza el patrón Composite internamente para dar soporte a la confección de menús. Lógicamente hay muchas cosas que no entenderás en el ejemplo, fíjate simplemente como se usan objetos JMenu(objeto compuesto) y JMenuItem(hoja) y como hay métodos add para añadir a los arrayList internos de los objetos JMenu

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;


class MiFrame extends JFrame{
 private JTextField txt=new JTextField("sin clic en menú por el momento");
 private JMenuBar mb= new JMenuBar();
 private JMenu menuOperaciones= new JMenu("operaciones");
 private JMenu menuSalir= new JMenu("salir");
 private JMenuItem itemSumar = new JMenuItem("sumar");
 private JMenuItem itemRestar = new JMenuItem("restar");
 private JMenuItem itemSalir = new JMenuItem("salir");
 private  ActionListener lSumar = new ActionListener(){
   public void actionPerformed(ActionEvent e){
      txt.setText("clic en suma");
   }
 };
 private  ActionListener lRestar = new ActionListener(){
   public void actionPerformed(ActionEvent e){
      txt.setText("clic en restar");
   }
 };
  private  ActionListener lSalir = new ActionListener(){
   public void actionPerformed(ActionEvent e){
      System.exit(0);
   }
 };
  public MiFrame(){
   super("probando menu");
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setSize(300,100);

   menuOperaciones.add(itemSumar);
   menuOperaciones.add(itemRestar);
   menuSalir.add(itemSalir);
   mb.add(menuOperaciones);
   mb.add(menuSalir);
   itemSumar.addActionListener(lSumar);
   itemRestar.addActionListener(lRestar);
   itemSalir.addActionListener(lSalir);
   setJMenuBar(mb);
```

```
   add(txt);

   setLayout(new FlowLayout());
   setVisible(true);
 }
}
public class App{
 public static void main(String[] args) {
   new MiFrame();
 }
}
```

Ejercicio U7_B4B_E3: Evaluar una expresión con composite(Coderunner)