Composite

Nombre:
Composite

Clasificación del patrón:
Estructural

Intención:
Componer objetos en estructura de árbol para representar jerarquías parte-todo. Composite permite al cliente tratar objetos individuales y composiciones de objetos uniformemente.

Otros nombres:
Compuesto.

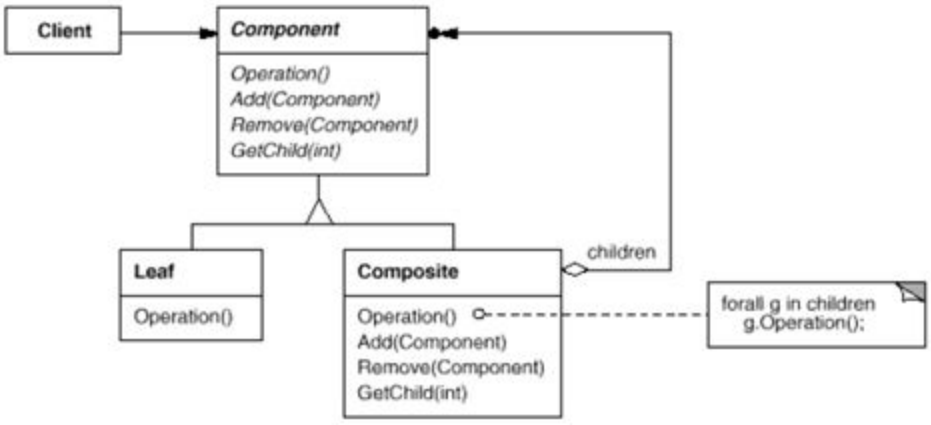
Motivación:
Uno de los ejemplos para dar una justificación al patrón es una aplicación gráfica que tenga componentes sencillos (líneas, texto, etc.), ahora es necesario crear clases que representan figuras geométricas, todas aquellas clases van a heredar los atributos de Figura, y van a tener un método que permita pintarlas. Ahora el problema radica en que debemos tratar grupos de imágenes para moverlos en la pantalla, se podría crear otra clase con un método pintar que gestione el grupo de figuras. Ocurriría un problema en cuanto a la complejidad del sistema ya que incrementa con respecto a la variedad de implementaciones del método pintar.

El patrón tratado en éste artículo proporciona una solución elegante al problema, brindando una clase abstracta que representa componentes y contenedores de la cual todas heredan y definen sus operaciones.

Aplicabilidad:
Se debe usar cuando se quiere:

- Representar jerarquías de objetos parte-todo como un árbol.
- Que los Composites y las composiciones sean indistinguibles para el cliente.

Estructura:



Participantes:

- Componente: Define la interfaz común para los objetos de la composición. También, define la interfaz para gestionar y acceder a los hijos.
- Hoja: Representa objetos sin hijos en la composición y define su comportamiento.
- Composite: Define el comportamiento para los componentes con hijos. Almacena los componentes con hijos e implementa las operaciones para la gestión de hijos.
- Cliente: Manipula los objetos de la composición a través de la interfaz.

Colaboraciones:

Cliente usa la interfaz de Componente para interactuar con objetos dentro de Composite. Si el receptor es Hoja se maneja la petición de forma directa, de ser Composite, lanza la petición a los hijos.

Ventajas:

- Simplifica la representación de jerarquías parte-todo. Los clientes no necesitan distinguir entre los componentes y las composiciones.
- Es más fácil añadir nuevos tipos de componentes. El composite puede trabajar fácilmente con nuevos componentes.

Desventajas:

- Se vuelve difícil añadir la restricción al tipo de componentes que pueden ser añadidos al Composite.

Implementación:

- Recomendaciones explícitas a los padres: Se simplifican operaciones en la estructura compuesta. Lo recomendable es hacer la definición en la clase Componente. Se gestionan al añadir o eliminar elementos del Composite.
- Compartir Componentes: Es muy útil por el ahorro de memoria. La gestión puede tornarse complicada.
- Maximizar la interfaz del componente: Dar un comportamiento que sobreescribirán las interfaces.
- Orden de los hijos: Hay que tener en cuenta el orden de los hijos en la implementación.
- Declaración de las operaciones de manejo de hijos: Se definen las operaciones en Componente y darle una implementación por defecto permite aumentar la transparencia pero ese genera un costo alto en cuanto a seguridad.

Código de ejemplo:

- Principal:

```
1  package estructurales.composite.composite01;
2  public class Main
3  {
4      public static void main(String[] args)
5      {
6          // Crear la carpeta principal e insertar archivos
7          Carpeta c1 = new Carpeta("CARPETA_1");
8          c1.insertarNodo( new Archivo("Archivo1.txt") );
9          c1.insertarNodo( new Archivo("Archivo2.txt") );
10         c1.insertarNodo( new Archivo("Archivo3.txt") );
11         // Crear una subcarpeta e insertar archivos
12         Carpeta c2 = new Carpeta("CARPETA_2");
13         c2.insertarNodo( new Archivo("Archivo4.txt") );
14         c2.insertarNodo( new Archivo("Archivo5.txt") );
15         // Insertar la subcarpeta dentro de la principal
16         c1.insertarNodo( c2 );
17         // Insertar otro archivo dentro de la carpeta principal
18         c1.insertarNodo( new Archivo("Archivo6.txt") );
19         c1.mostrar();
20         System.out.println("--------------------");
21         // Eliminamos la subcarpeta (junto con su contenido)
22         c1.eliminarNodo( c2 );
23         c1.mostrar();
24     }
25 }
```

- Nodo (Componente):

```java
package estructurales.composite.composite01;
public abstract class Nodo
{
    public static final int ARCHIVO = 1;
    public static final int CARPETA = 2;
    protected String nombre = "";
    protected int tipoNodo;
    // ----------------------------
    public String getNombre()
    {
        return this.nombre;
    }
    // ----------------------------
    public void setNombre( String nombre )
    {
        this.nombre = nombre;
    }
    // ----------------------------
    public int getTipoNodo()
    {
        return this.tipoNodo;
    }
    // ----------------------------
    public void setTipoNodo( int tipoNodo )
    {
        this.tipoNodo = tipoNodo;
    }
    // ----------------------------
    // Método a implementar por las clases que hereden
     public abstract void mostrar();
}
```

- Archivo:

```
1  package estructurales.composite.composite01;
2  public class Archivo extends Nodo
3  {
4      public Archivo( String nombre )
5      {
6          this.setTipoNodo( Nodo.ARCHIVO );
7          this.setNombre( nombre );
8      }
9      // ----------------------------
10     @Override
11      public void mostrar()
12     {
13         System.out.println( "Archivo: [" + this.getNombre() + "]" );
14     }
15 }
```

- Carpeta:

```java
package estructurales.composite.composite01;
import java.util.ArrayList;
import java.util.List;
public class Carpeta extends Nodo
{
    List<Nodo> nodos = new ArrayList<Nodo>();
    // ----------------------------
    public Carpeta( String nombre )
    {
        this.setTipoNodo( Nodo.CARPETA );
        this.setNombre( nombre );
    }
    // ----------------------------
    public void insertarNodo( Nodo nodo )
    {
        nodos.add( nodo );
    }
    // ----------------------------
    public void eliminarNodo( Nodo nodo )
    {
        nodos.remove( nodo );
    }
    // ----------------------------
    public List<Nodo> getNodos()
    {
        return nodos;
    }
    // ----------------------------
    public Nodo getNodo( int posicion )
    {
        return nodos.get( posicion );
    }
    // ----------------------------
    @Override
     public void mostrar()
    {
        System.out.println( "Listando carpeta [" + this.getNombre() + "]" );
         for (Nodo nodo : nodos)
        {
            nodo.mostrar();
        }
    }
}
```

Usos conocidos:

- Librerías de Java como Component, Container, Panel, Botón.

Bibliografía:

No específico. (No específico). GoF Design Patterns (Versión 2.1.0) [Aplicación móvil]. Descargado de: https://drive.google.com/file/d/0BywiVyFlIabXcVhGZlJBcnhWTkU/view.

Patrones de Diseño Software [Página Web]. (s.f.). Ubicación: https://informaticapc.com/patrones-de-diseno/composite.php.

Junta de Andalucía. (s.f). Composite. Marco de Desarrollo de la Junta de Andalucía. http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/184.