Composite Pattern

Introducción y nombre

Composite. Estructural. Permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura de árbol.

Intención

El patrón Composite sirve para construir algoritmos u 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.

También conocido como

Compuesto.

Motivación

Este patrón propone una solución basada en composición recursiva. Además, describe como usar correctamente esta recursividad. Esto permite tratar objetos individuales y composiciones de objetos de manera uniforme. Este patrón busca representar una jerarquía de objetos conocida como “parte-todo”, donde se sigue la teoría de que las "partes" forman el "todo", siempre teniendo en cuenta que cada "parte" puede tener otras "parte" dentro.

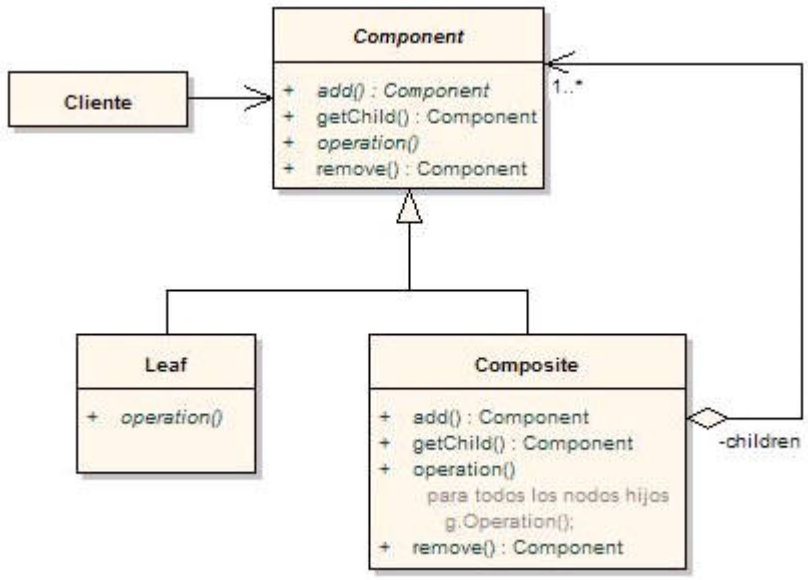
Solución

Se debe utilizar este patrón cuando:

Se busca representar una jerarquía de objetos como “parte-todo”.

Se busca que el cliente puede ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

Diagrama UML



Participantes

Component

Declara la interface para los objetos de la composición.

Implementa un comportamiento común entre las clases.

Declara la interface para acceso y manipulación de hijos.

Declara una interface de manipulación a los padres en la estructura recursiva.

Leaf

Representa los objetos “hoja” (no poseen hijos).

Define comportamientos para objetos primitivos.

Composite

Define un comportamiento para objetos con hijos.

Almacena componentes hijos.

Implementa operaciones de relación con los hijos.

Cliente

Manipula objetos de la composición a través de Component.

Colaboraciones

Los clientes usan la interfaz de Component para interactuar con objetos en la estructura Composite. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

Consecuencias

Define jerarquías entre las clases.

Simplifica la interacción de los clientes.

Hace más fácil la inserción de nuevos hijos.

Hace el diseño más general.

Implementación

Hay muchas formas de llevar a cabo este patrón. Muchas implementaciones implican referencias explicitas al padre para simplificar la gestión de la estructura.

Si el orden de los hijos provoca problemas utilizar Iterator.

Compartir componentes reduce los requerimientos de almacenamiento.

Para borrar elementos hacer un compuesto responsable de suprimir los hijos.

La mejor estructura para almacenar elementos depende de la eficiencia: si se atraviesa la estructura con frecuencia se puede dejar información en los hijos. Por otro lado, el componente no siempre debería tener una lista de componentes, esto depende de la cantidad de componentes que pueda tener.

Código de muestra

Imaginemos una escuela, que esta compuesta de diferentes sectores (Dirección, Aulas, etc) y personas (profesores, alumnos, portero, etc). Se busca que la escuela pueda identificar las personas que posee y la edad de cada una. Todos deben identificarse con la misma interface:

[code]

```
public interface IPersonal {
    public void getDatosPersonal();
}

public class Composite implements IPersonal {
    private List<IPersonal> values = new ArrayList<IPersonal>();

    public void agrega(IPersonal personal) {
        values.add(personal);
    }

    @Override
    public void getDatosPersonal() {
        for (IPersonal personal : values) {
            personal.getDatosPersonal();
        }
    }
}

public class Escuela extends Composite {
}

public class Direccion extends Composite {
}

public class Aula extends Composite {
}

public class Persona implements IPersonal {

    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.edad = edad;
        this.nombre = nombre;
    }

    @Override
    public void getDatosPersonal() {
        String msg = "Me llamo " + nombre;
        msg = msg + ", tengo " + edad + " años";
        System.out.println(msg);
```

```
    }

    public String getNombre() {
        return nombre;
    }

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

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}

public class Main {
    public static void main(String[] args) {
        Persona alumno1 = new Persona("Juan Perez", 21);
        Persona alumno2 = new Persona("Vanesa Gonzalez", 23);
        Persona alumno3 = new Persona("Martin Palermo", 26);
        Persona alumno4 = new Persona("Sebastian Paz", 30);
        Persona alumno5 = new Persona("Pepe Pillo", 22);

        Persona profesor1 = new Persona("Jacinto Dalo", 54);
        Persona profesor2 = new Persona("Guillermo Tei", 43);

        Persona director = new Persona("Dr Cito  Maximo", 65);
        Persona portero = new Persona("Don Manolo", 55);

        Escuela escuela = new Escuela();
        escuela.agrega(portero);

        Direccion direccion = new Direccion();
        direccion.agrega(director);

        Aula aula1 = new Aula();
        aula1.agrega(profesor1);
        aula1.agrega(alumno1);
        aula1.agrega(alumno2);
        aula1.agrega(alumno3);

        Aula aula2 = new Aula();
        aula2.agrega(profesor2);
        aula2.agrega(alumno4);
        aula2.agrega(alumno5);

        escuela.agrega(direccion);
        escuela.agrega(aula1);
        escuela.agrega(aula2);

        escuela.getDatosPersonal();
    }
}
```

Y la salida por consola es:

```
Me llamo Don Manolo, tengo 55 años
Me llamo Dr Cito  Maximo, tengo 65 años
Me llamo Jacinto Dalo, tengo 54 años
Me llamo Juan Perez, tengo 21 años
Me llamo Vanesa Gonzalez, tengo 23 años
Me llamo Martin Palermo, tengo 26 años
Me llamo Guillermo Tei, tengo 43 años
Me llamo Sebastian Paz, tengo 30 años
```

Me llamo Pepe Pillo, tengo 22 años
[/code]

Cuándo utilizarlo

Este patrón busca formar un "todo" a partir de las composiciones de sus "partes". A su vez, estas "partes" pueden estar formadas de otras "partes" o de "hojas". Esta es la idea básica del Composite.

Java utiliza este patrón en su paquete de AWT (interfaces gráficas). En el paquete java.awt.swing el Componente es la clase Component, el Compuesto es la clase Container (sus Compuestos Concretos son Panel, Frame, Dialog y las hojas Label, TextField y Button.

Imaginemos que tenemos un software donde si pinta un gráfico. Algunas partes se pintan, mientras que otras partes de nuestro gráfico, de hecho, son otros gráficos y ciertas partes de estos últimos gráficos son, a su vez, otros gráficos. Este ejemplo es un caso típico para el patrón Composite.

Patrones relacionados

- Decorator: si se usan juntos normalmente tienen una clase común padre.
- Flyweight: permite compartir componentes.
- Iterator: sirve para recorrer las estructuras de los componentes.
- Visitor: localiza comportamientos en componentes y hojas.