NOMBRE	Programación Avanzada		
AREA	Software	REGIMEN	Semestral

Composite

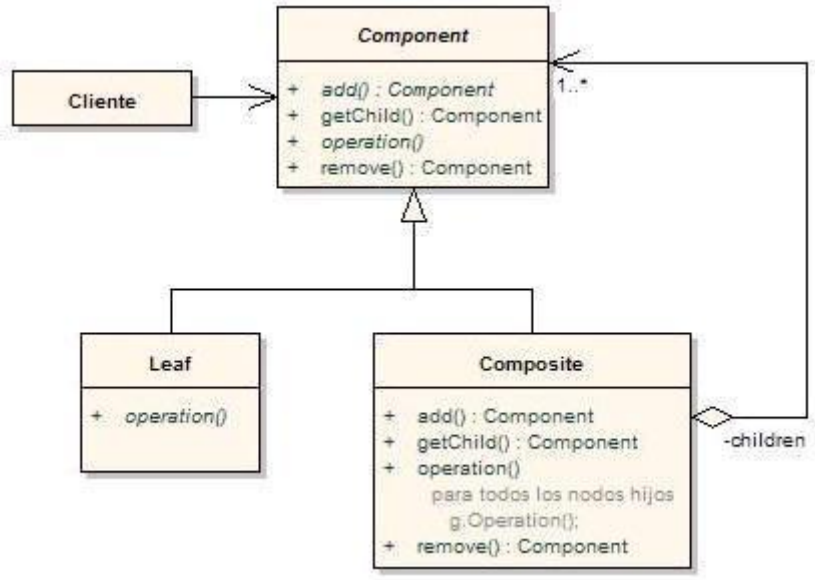
El patrón Composite sirve para construir algoritmos u objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, porque todos ellos una interfaz común, se tratan todos de la misma manera.

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.

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 de Clases



Component: implementa un comportamiento común entre las clases y 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 e implementa operaciones de relación con los hijos.

Cliente: manipula objetos de la composición a través de Component. 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.

Ejemplo

Vamos a realizar un ejemplo de un Banco. Un banco puede tener muchos sectores: Gerencia, Administrativo, RRHH, Cajas, etc. Cada uno de estos sectores tendrá empleados que cobran un sueldo. En nuestro caso utilizaremos el Composite para calcular la sumatoria de sueldos de la empresa. Para ello definimos la Interface y el Composite.

```
public interface ISueldo {
    public double getSueldo();

}
```

```
public class Composite implements ISueldo {
    private ArrayList<ISueldo> empleados = new ArrayList<ISueldo>();

    @Override
    public double getSueldo() {
        double sumador = 0;
        for (int i = 0; i < empleados.size(); i++) {
            sumador = sumador + empleados.get(i).getSueldo();
        }

        return sumador;
    }

    public void agrega(ISueldo p) {
        empleados.add(p);
    }

}
```

En la clase Composite está todo el secreto del patrón: contiene una colección de hijos del tipo ISueldo que los va agregando con el método agrega(ISueldo) y

cuando al composite le ejecutan el getSueldo() simplemente recorre sus hijos y les ejecutas el método. Sus hijos podrán ser de 2 tipos: Composite (que a su vez harán el mismo recorrido con sus propios hijos) u Hojas que devolverán un valor.

En nuestro ejemplo hay varios sectores, solo pongo uno para no llenar de pics el ejemplo. Pero todos son muy parecidos: la única relación que tienen con el composite es que heredan de él.

```
public class SectorCajas extends Composite {
    private int cantidadCajeros;
    //Aca iriran los atributos y metodos propios del sector

    public int getCantidadCajeros() {
        return cantidadCajeros;
    }

    public void setCantidadCajeros(int cantidadCajeros) {
        this.cantidadCajeros = cantidadCajeros;
    }

}
```

Otra hoja es el empleado que trabaja en el Banco.

```
public class Empleado implements ISueldo {
    private String nombreCompleto, cargo;
    private double sueldo;
    // y todos los atributos propios del empleado...

    public Empleado(String nombreCompleto, String cargo, double sueldo){
        setCargo(cargo);
        setNombreCompleto(nombreCompleto);
        setSueldo(sueldo);
    }

    public double getSueldo() {
        return sueldo;
    }
```

El armado se hace en el Main , que es el Cliente pero no siempre es así. El banco se compone de varios sectores, los cuales pueden tener personas o más sectores adentro.

NOMBRE	Programación Avanzada		
AREA	Software	REGIMEN	Semestral

```
    public static void main(String[] args) {
        Banco banco = new Banco();
        SectorAdministrativo administracion = new SectorAdministrativo();
        SectorCajas cajas = new SectorCajas();
        SectorContaduria contaduria = new SectorContaduria();
        SectorGerencia gerencia = new SectorGerencia();
        SectorRRHH rrhh = new SectorRRHH();

        banco.agrega(gerencia); banco.agrega(contaduria);banco.agrega(administracion)
        administracion.agrega(cajas);administracion.agrega(rrhh);

        Empleado cajero1 = new Empleado("Juan Perez", "Cajero", 2000);
        Empleado cajero2 = new Empleado("Perico Perez", "Cajero", 2000);
        cajas.agrega(cajero1); cajas.agrega(cajero2);

        Empleado gerente =  new Empleado("Soy Groso", "Gerente", 5000);
        gerencia.agrega(gerente);

        Empleado selectora1 = new Empleado("Marisa Gomez", "Selectora", 1500);
        rrhh.agrega(selectora1);

        Empleado contador = new Empleado("Don Contador", "Contador", 3000);
        contaduria.agrega(contador);

        System.out.println(banco.getSueldo());
```

Problems | Javadoc | Declaration | Console

```
<terminated> Main [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (01/06/2011 17:21:25)
13500.0
```

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.

Si la operación es compleja, puede ensuciar mucho el código y hacerlo ilegible.