Composite

Patrón de diseño

*Bea, David

*Gonzalez, Cristian

*Saez, Sebastian

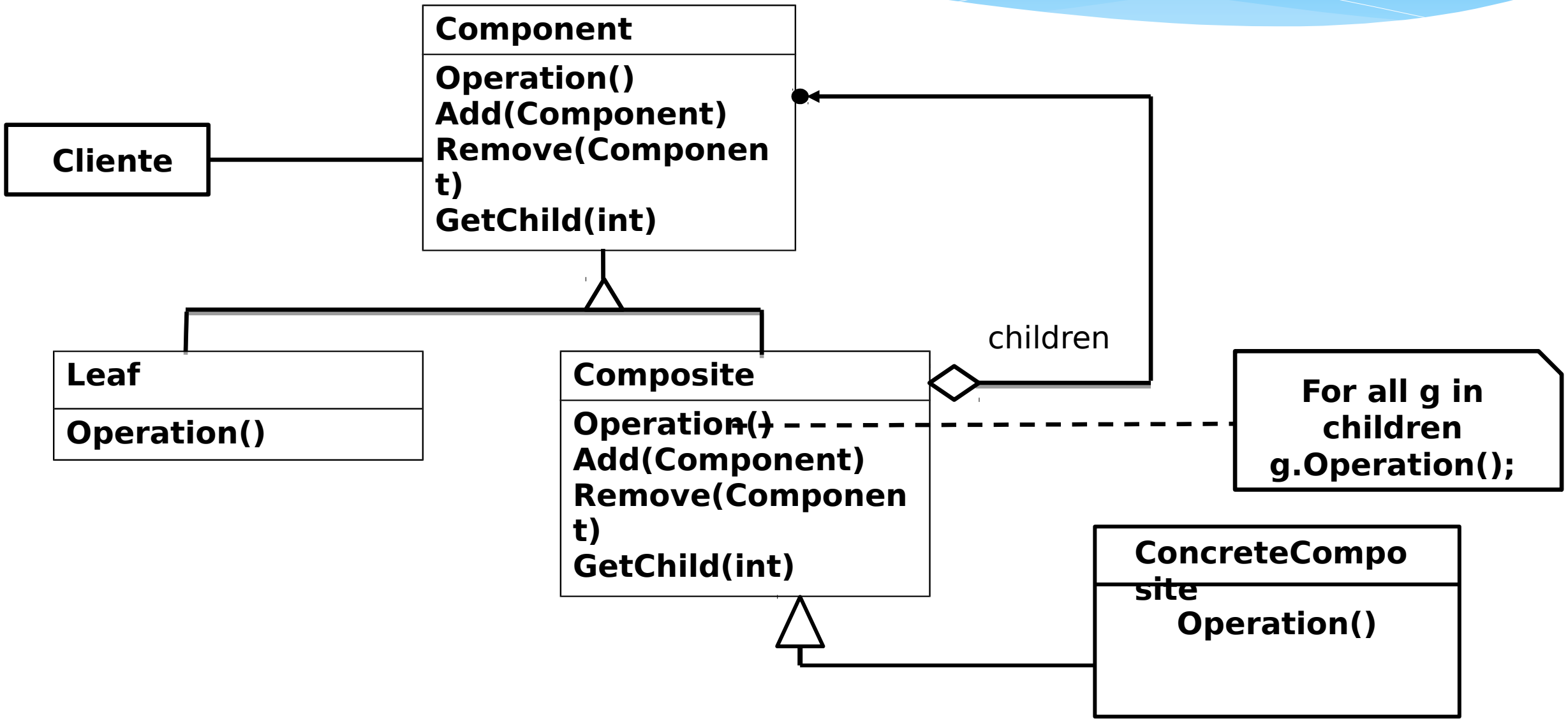
Intención

- Componer objetos en estructura de árbol para representar jerarquías “parte-todo”.
- Se busca que el cliente puede ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

Motivación

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. Dicho de otra forma, permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura 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.

Estructura


Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Cliente
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
For all g in children
g.Operation();
ConcreteComposite
Operation()

Participantes

- **Client:**

*Manipula objetos en la composición a través de la interface Component.

- **Component:**

*Declara la interface para objetos en la composición.
*Implementa el comportamiento de la interface común a todas las clases.
*Declara una interface para el acceso y manejo de sus componentes hijos.

- **Leaf:**

*Representa objetos hoja en la composición. No tiene hijos.
*Define el comportamiento de los objetos primitivos.

- **Composite:**

*Define el comportamiento de los componentes que tienen hijos.
*Almacena componentes hijos.
*Implementa las operaciones relacionadas a los hijos de la interface component.

Árbol generado

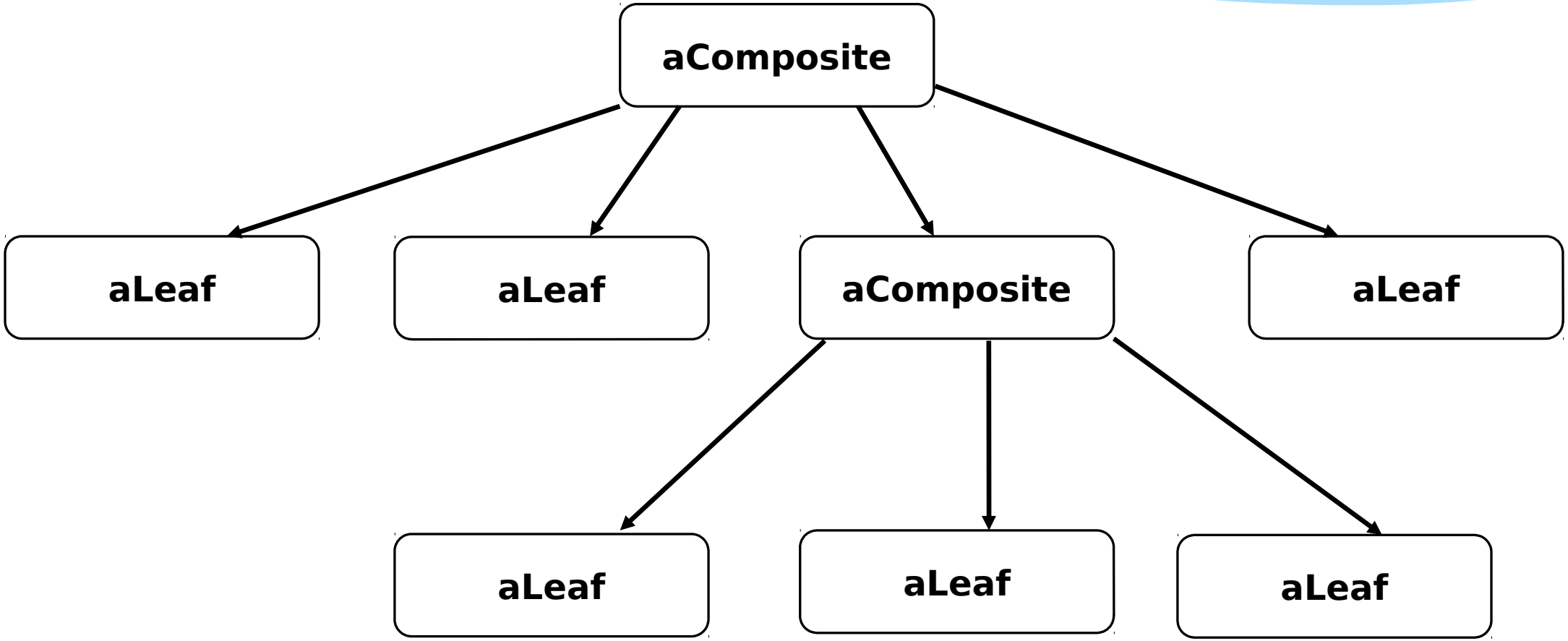
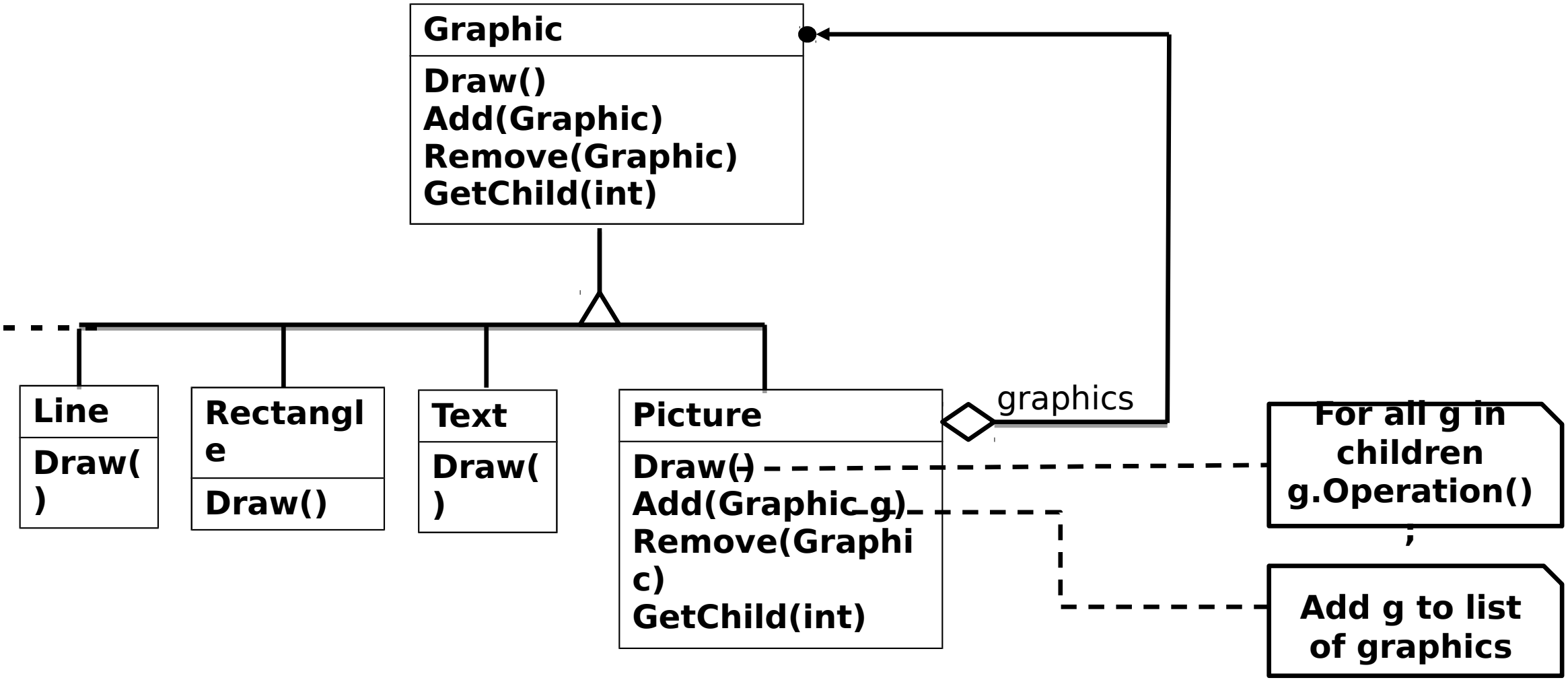


Diagrama de clases ejemplo



Aplicabilidad

El patrón Composite se utiliza cuando:

***Se desea representar jerarquías parte-todo de objetos.**

***Se desea que los clientes ignoren la diferencia entre composiciones de objetos y objetos individuales.**

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.

Codigo

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 l Para ello definimos la Interface y el C

```
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 pro rán un valor.

```
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 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;
    }

}
```

Ahora veamos el caso del empleado que trabaja en el banco (sería una hoja):

```
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;
    }
```

Listo, ahora veamos la forma de concatenar todo. En este caso se realizo en el Main (que vendría a representar al cliente), pero en realidad el armado del banco no siempre lo debe hacer el cliente. De hecho, podríamos utilizar un patrón creacional para que nos ayude en el armado.

El banco se co cuales pueden tener p tro.

```
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
```

Referencias

* Gamma Erich, Helm Richard, Johnson Ralph. Design Patterns: Elements of Reusable Object-Oriented Software . Add. Wesley. 1995.
* http://migranitodejava.blogspot.com.ar/2011/06/composite.html
* https://rootear.com/desarrollo/patron-composite