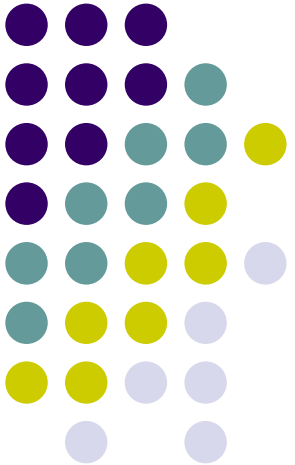
Patrones de Diseño

Patrón estructural *Composite*



Composite

Propósito

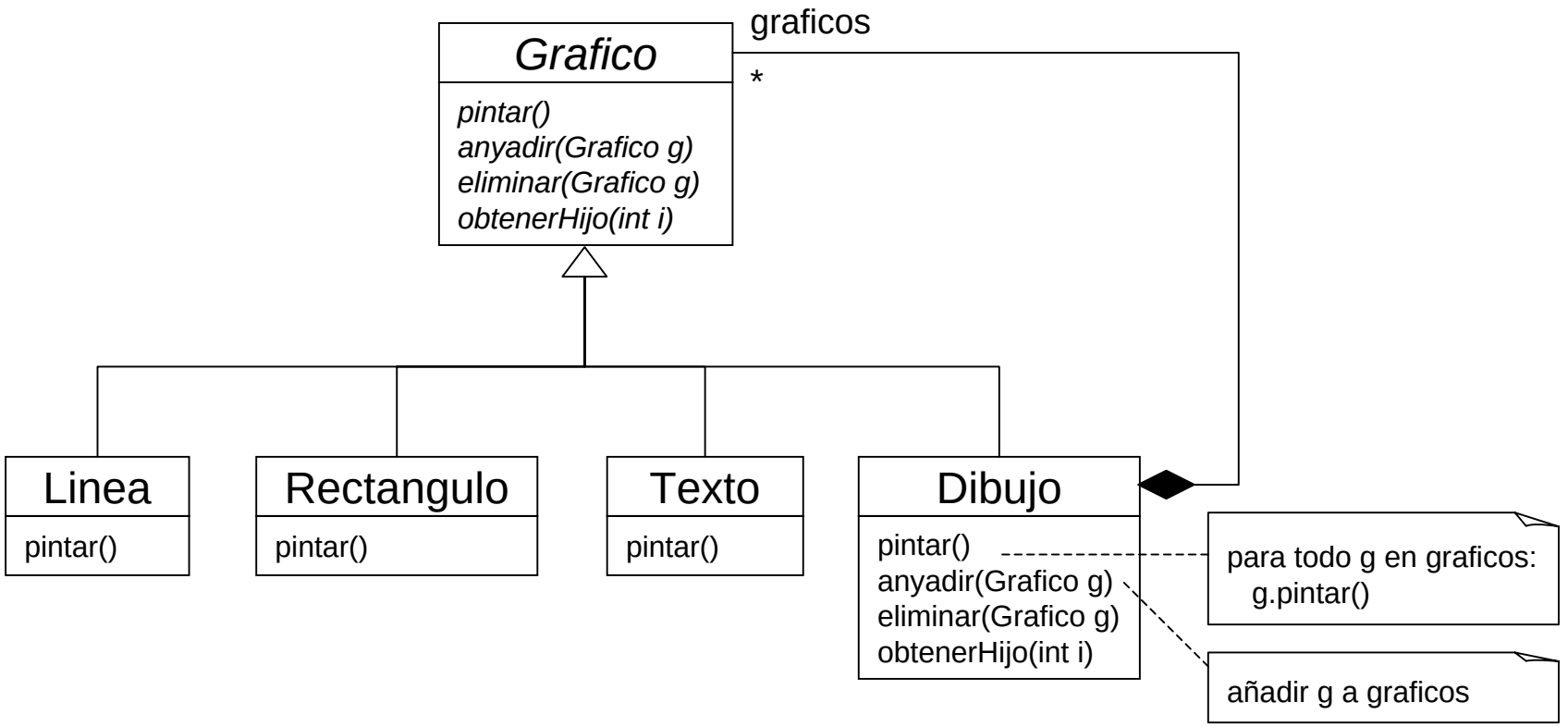
- Componer objetos en estructuras arbóreas para representar jerarquías todo-parte. Manipular todos los objetos del árbol de manera uniforme

Motivación

- Ej.: las aplicaciones gráficas manejan grupos de figuras hechas de componentes sencillos (líneas, texto...)

- Solución:
  - Primitivas para los componentes sencillos, y otras para los contenedores? No porque no se tratan de manera uniforme
  - Definir una clase abstracta que represente componentes y contenedores, de la cual todas heredan, y que define sus operaciones

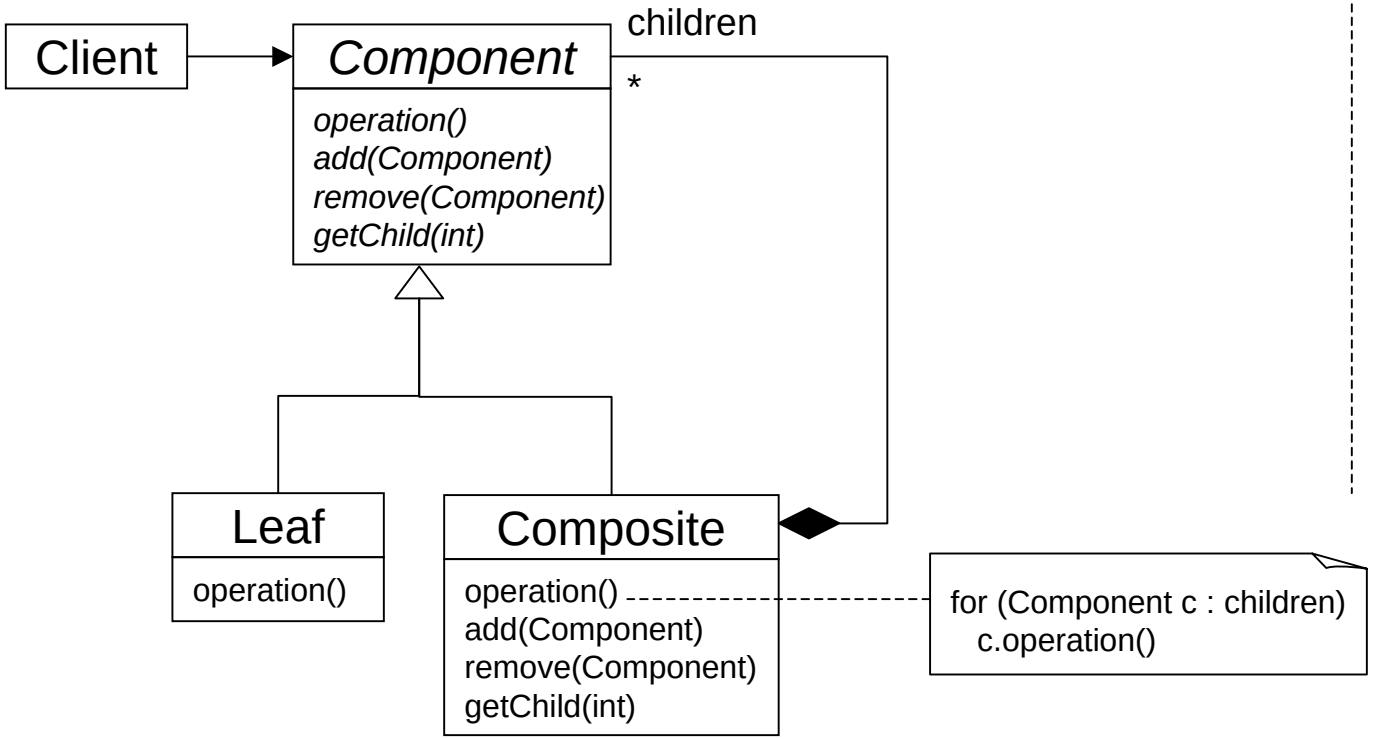
Composite
Motivación



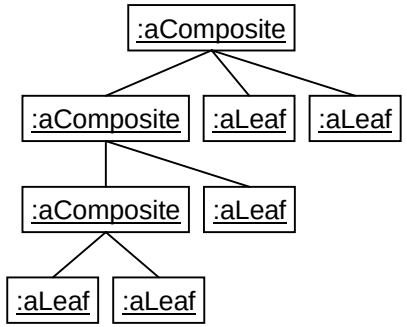
Composite
Aplicabilidad

- Usa el patrón *Composite* cuando:
  - Quieres representar jerarquías de objetos todo-parte
  - Quieres ser capaz de ignorar la diferencia entre objetos individuales y composiciones de objetos. Los clientes tratarán a todos los objetos de la estructura compuesta uniformemente.

Composite
Estructura



Ejemplo de estructura-objeto:



Composite
Participantes

- **Component** (*Grafico*):
  - define la interfaz común para los objetos de la composición
  - define la interfaz para acceder y gestionar los hijos
  - implementa un comportamiento por defecto común a las subclases
  - (opcional) define la interfaz para acceder al padre de un componente en la estructura recursiva, y la implementa si es apropiado
- **Leaf** (*Linea*, *Rectangulo*, *Texto*):
  - representa los objetos hoja (sin hijos) de la composición
  - define comportamiento para los objetos primitivos
- **Composite** (*Dibujo*)*:*
  - define comportamiento para los componentes que tienen hijos
  - almacena componentes hijo
  - implementa las operaciones de *Component* para la gestión de hijos
- **Client***:*
  - manipula los objetos de la composición a través de la interfaz *Component*

Composite
Consecuencias

- Define jerarquías de clases hechas de objetos primitivos y compuestos. Si el código cliente espera un objeto simple, puede recibir también uno compuesto
- Simplifica el cliente. Objetos simples y compuestos se tratan homogéneamente
- Facilita la incorporación de nuevos tipos de componentes
- Puede hacer el diseño demasiado general. Es complicado restringir el tipo de componentes de un *composite*.

Composite
Implementación

- Referencias explícitas a los padres
  - Simplifica algunas operaciones de la estructura compuesta
  - Definirlas en la clase *Component*
  - Gestionarlas al añadir/eliminar elementos de un *Composite*
- Compartir componentes
  - Útil para ahorrar memoria
  - La gestión de un componente con varios padres se complica
- Maximizar la interfaz del componente
  - Dar comportamiento por defecto que sobrescribirán las subclase
  - Ej: por defecto *getChildren* no devuelve hijos, lo cual es válido para las hojas, pero los compuestos deben sobrescribir la operación

Composite
Implementación

- Declaración de las operaciones de gestión de hijos
  - Definirlas en la raíz *Component* y dar implementación por defecto
    - Se obtiene transparencia, se pierde seguridad (¿cómo evitar que un cliente añada/elimine objetos a una hoja?)
  - Definirlas en *Composite*
    - Se obtiene seguridad, se pierde transparencia (interfaz no uniforme)
    - Si se pierde el tipo hay que hacer *downcasting*, lo cual es inseguro
- ¿Debe declarar *Component* una lista de componentes?
  - Penalización de espacio por cada hoja, incluso si no tiene hijos
- A veces los hijos tienen un orden

Composite
Implementación

- ¿Quién debe borrar los componentes?
  - Si no hay recolector de basura, el *Composite*
  - Excepción si las hojas son inmutables y pueden estar compartidas
- ¿Cuál es la mejor estructura de datos para almacenar los componentes?

Composite

Código de ejemplo (1)


```
public interface Component {
   public void add (Component c);
   public void remove (Component c);
   public Component getChild (int i);
}

public class Leaf implements Component {
   public void add (Component c)     {}  // también puede lanzar una excepción
   public void remove (Component c)  {}  // también puede lanzar una excepción
   public Component getChild (int i) { return null; }
}

public class Composite implements Component {
   private Vector _children = new Vector();
   public void add (Component c)     { _children.addElement(c); }
   public void remove (Component c)  { _children.removeElement(c); }
   public Component getChild (int i) { return _children.elementAt(i); }
}
```

Composite

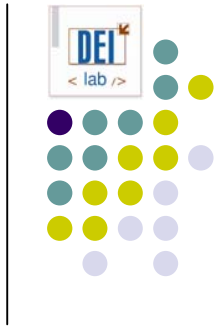
Código de ejemplo (2)


```
public abstract class Component {
   public void add (Component c)     {}  // también puede lanzar una excepción
   public void remove (Component c)  {}  // también puede lanzar una excepción
   public Component getChild (int i) { return null; }
}

public class Leaf extends Component {
}

public class Composite extends Component {
   private Vector _children = new Vector();
   public void add (Component c)     { _children.addElement(c); }
   public void remove (Component c)  { _children.removeElement(c); }
   public Component getChild (int i) { return _children.elementAt(i); }
}
```

Composite
Código de ejemplo (3)


```
public interface Component {
   public Composite getComposite ();
}
public class Leaf implements Component {
   public Composite getComposite () { return null; }
}
public class Composite implements Component {
   private Vector _children = new Vector();
   public void add (Component c)     { _children.addElement(c); }
   public void remove (Component c)  { _children.removeElement(c); }
   public Component getChild (int i) { return _children.elementAt(i); }
   public Composite getComposite ()  { return this; }
}

// código cliente
Composite aComposite = new Composite();
Leaf aLeaf = new Leaf();
Component aComponent;
aComponent = aComposite;
if (aComponent == aComponent.getComposite()) test.add(new Leaf());  // añadirá la hoja
aComponent = aLeaf;
if (aComponent == aComponent.getComposite()) test.add(new Leaf());  // no añadirá la hoja
```

Composite
En java…



- En el paquete java.awt.swing
  - **Component**
    - `Component`
  - **Composite**
    - `Container` (abstracta)
    - `Panel` (concreta)
    - `Frame` (concreta)
    - `Dialog` (concreta)
  - **Leaf**:
    - `Label`
    - `TextField`
    - `Button`

Composite
Ejercicio

- **Se quiere construir un editor de expresiones matemáticas. Especificar el diagrama de clases que permita representar expresiones válidas.**

- **Una expresión válida estará formada o bien por un número, o bien por la suma/resta/división/multiplicación de dos expresiones**

- **Ejemplos de expresiones válidas:**
  - **4**
  - **3+8**
  - **14 * (3+5)**
  - **...**