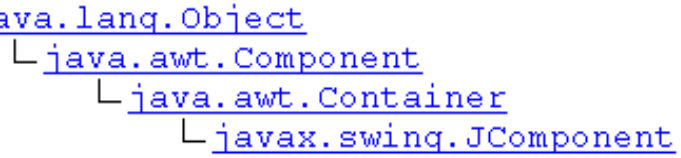
Patrones de Diseño: Patrones Estructurales. Tema 4-4: Composite

Descripción del patrón

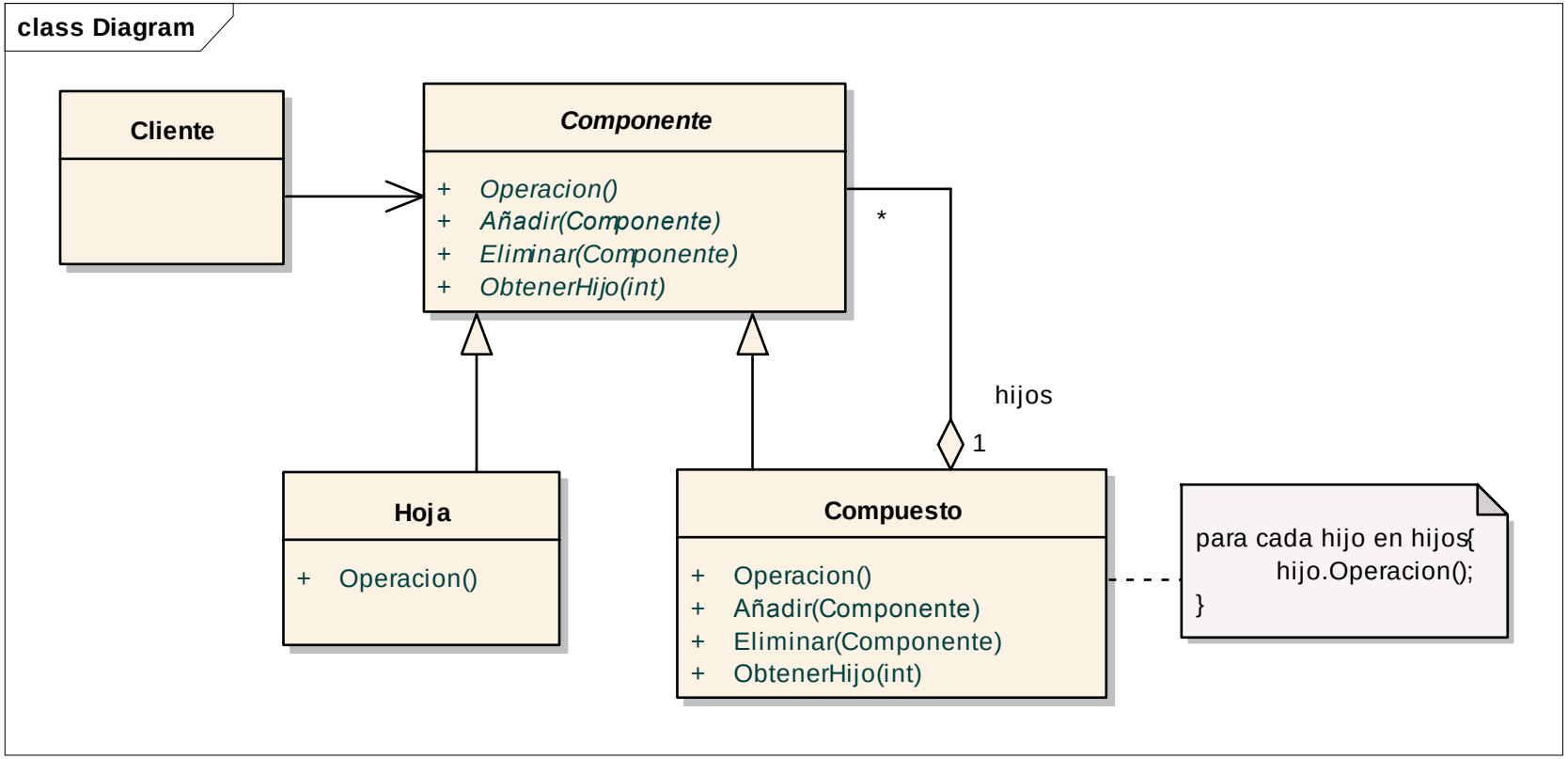
- **Nombre:**
  - Compuesto
  - También conocido como Handle/Body
- **Propiedades:**
  - Tipo: estructural
  - Nivel: objeto, componente
- **Objetivo o Propósito:**
  - Construir objetos de complejidad mayor mediante otros más sencillos de forma recursiva, formando una jerarquía en estructura de árbol. Permite que todos los elementos tanto individuales como compuestos se traten de una manera uniforme por los clientes.

Aplicabilidad

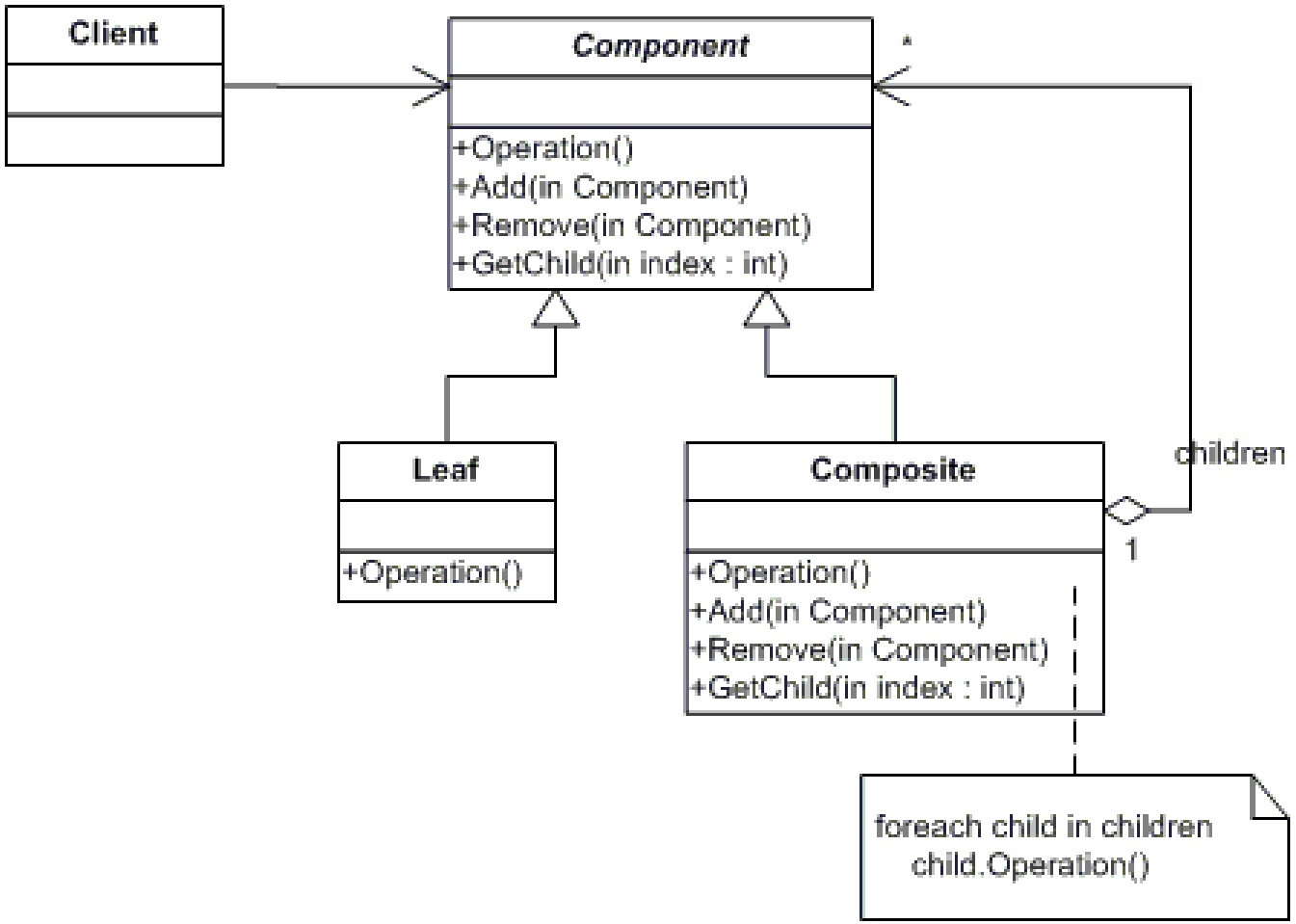
- Use el patrón Composite cuando:
  - Tenga que representar un componente modelado como una estructura rama-hoja (o parte-todo o contenedor-contenido).
  - La estructura pueda tener cualquier nivel de complejidad y sea dinámica.
  - Quiera tratar de forma uniforme toda la estructura del componente, utilizando operaciones comunes para toda la jerarquía.
  - Desee que los clientes sean capaces de obviar las diferencias entre objetos compuestos e individuales.
- Ejemplo: En el API de AWT y Swing de Java podemos encontrar un ejemplo de la aplicación del patrón en los componentes y contenedores.



Estructura


class Diagram
Cliente
Componente
+ Operacion()
+ Añadir(Componente)
+ Eliminar(Componente)
+ ObtenerHijo(int)
*
hijos
1
Hoja
+ Operacion()
Compuesto
+ Operacion()
+ Añadir(Componente)
+ Eliminar(Componente)
+ ObtenerHijo(int)
para cada hijo en hijos{
hijo.Operacion();
}

Estructura


Client
Component
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
*
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Estructura. Participantes

- **Componente:** Clase abstracta común a todos los objetos, declara los métodos u operaciones que pueden realizar tanto los objetos individuales como los compuestos.
- **Compuesto:** Define los métodos propios de los objetos compuestos, normalmente un método para añadir componentes tanto individuales como compuestos, un método para eliminarlos, y otro para poder recuperar los distintos objetos que lo componen.
- **Hoja:** Representa las clases individuales, no compuestas (objetos primitivos de la composición).
- **Cliente:** Clase que crea y manipula los objetos de la composición a través de Componente.

Estructura. Variaciones

- **Variaciones del patrón:**

- **Nodo raíz.** Para mejorar la manejabilidad del sistema, en algunas ocasiones se define un objeto distinto que actúa como base para la jerarquía completa de objetos Composite. Si el objeto raíz se representa como una clase separada se puede implementar como un Singleton para garantizar su acceso.
- **Ramificación basada en reglas.** En estructuras complejas, con múltiples tipos de nodos y ramas se pueden imponer una serie de reglas para determinar como se pueden unir ciertos tipos de nodos a ciertos tipos de ramas.

Consecuencias

- Se crea una jerarquía de objetos básicos y de composiciones de estos de forma recursiva.
- Simplifica el cliente al tratar a todos los objetos de igual forma.
- Facilita agregar nuevas clases de componentes insertándolas en la jerarquía de clases como hijas de Compuesto u Hoja Los clientes siguen siendo compatibles con la nueva estructura.
- Desventaja: Puede generalizar el diseño, haciendo difícil restringir los componentes de un objeto compuesto. No podemos confiar en el sistema de tipos y deberemos hacer comprobaciones en tiempo de ejecución.

Implementación

- Gestionar los enlaces al padre.
- Estudiar si se desea que una agrupación admita agrupaciones o solo objetos simples.
- Operaciones de gestión de nodos hijos (añadir, eliminar, obtenerHijo).
- Estudiar si interesa que tenga significado el orden de los hijos.
- Estudiar la destrucción de los objetos.
- Se pueden utilizar Iteradores para recorrer los integrantes.
- Visitor puede realizar operaciones para evitar distribuirlas por los hijos.

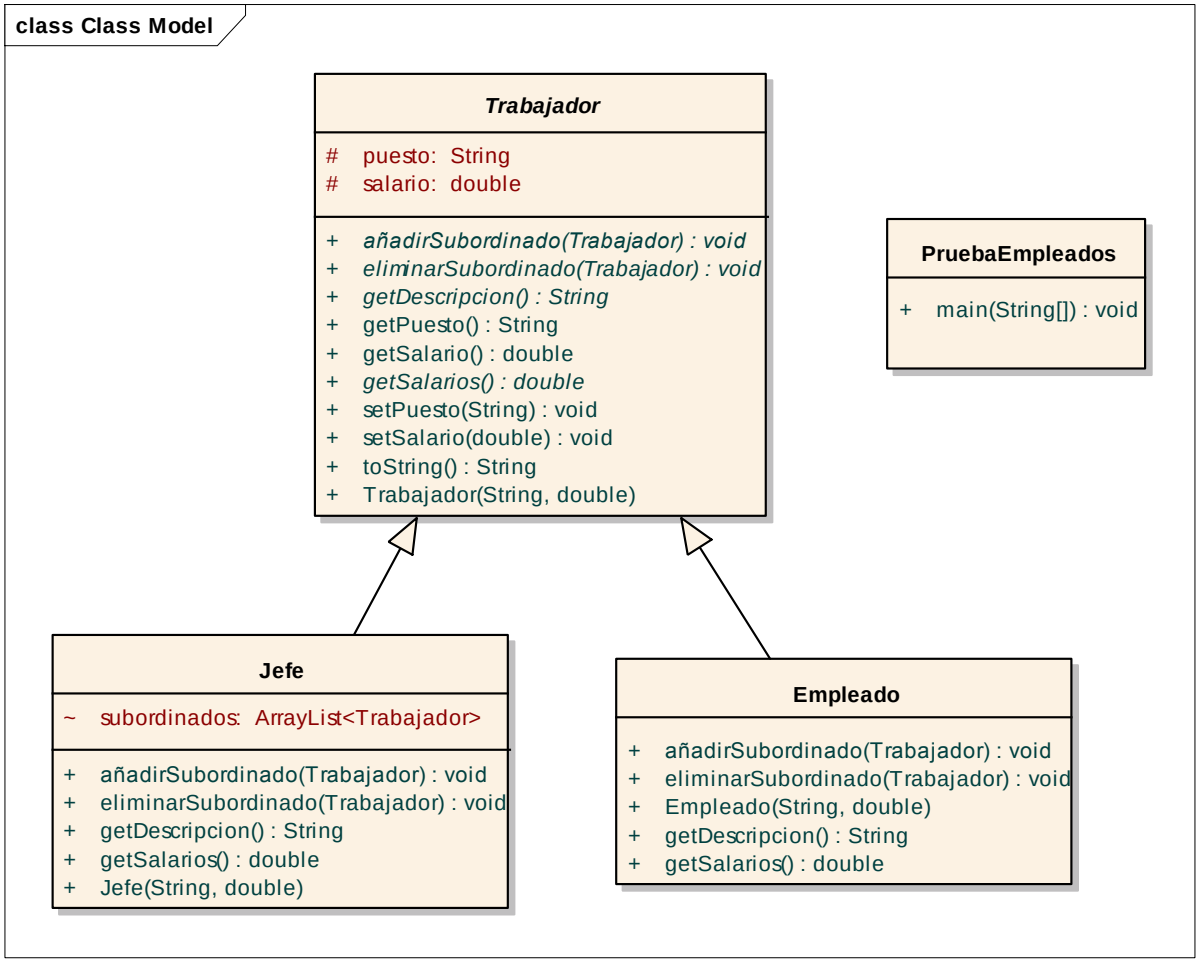
Patrones relacionados

- **Chain of Responsability**: Este patrón puede ser combinado con el patrón Composite para añadir enlaces del hijo al padre, de tal forma que los hijos puedan conseguir información sobre el padre sin tener que conocer quien proporciona esa información.
- **Decorator**: Generalmente se utilizan juntos.
- **Flyweight**: Si la estructura en árbol se hace muy grande, Flyweight puede reducir el número de objetos gestionados por el árbol al compartir componentes.
- **Iterator**: Para recorrer los hijos en un Composite.
- **Visitor**: Se puede utilizar este patrón para encapsular operaciones en una clase simple y de esta forma centralizar el comportamiento, que de lo contrario, podría dividirse a través de muchas clases hojas y compuestos.

Código de ejemplo

EMPLEADOS

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - Componente: Trabajador.
  - Compuesto: Jefe.
  - Hoja: Empleado.
  - Cliente: PruebaEmpleados.