Composite Pattern

Salih ŞEN

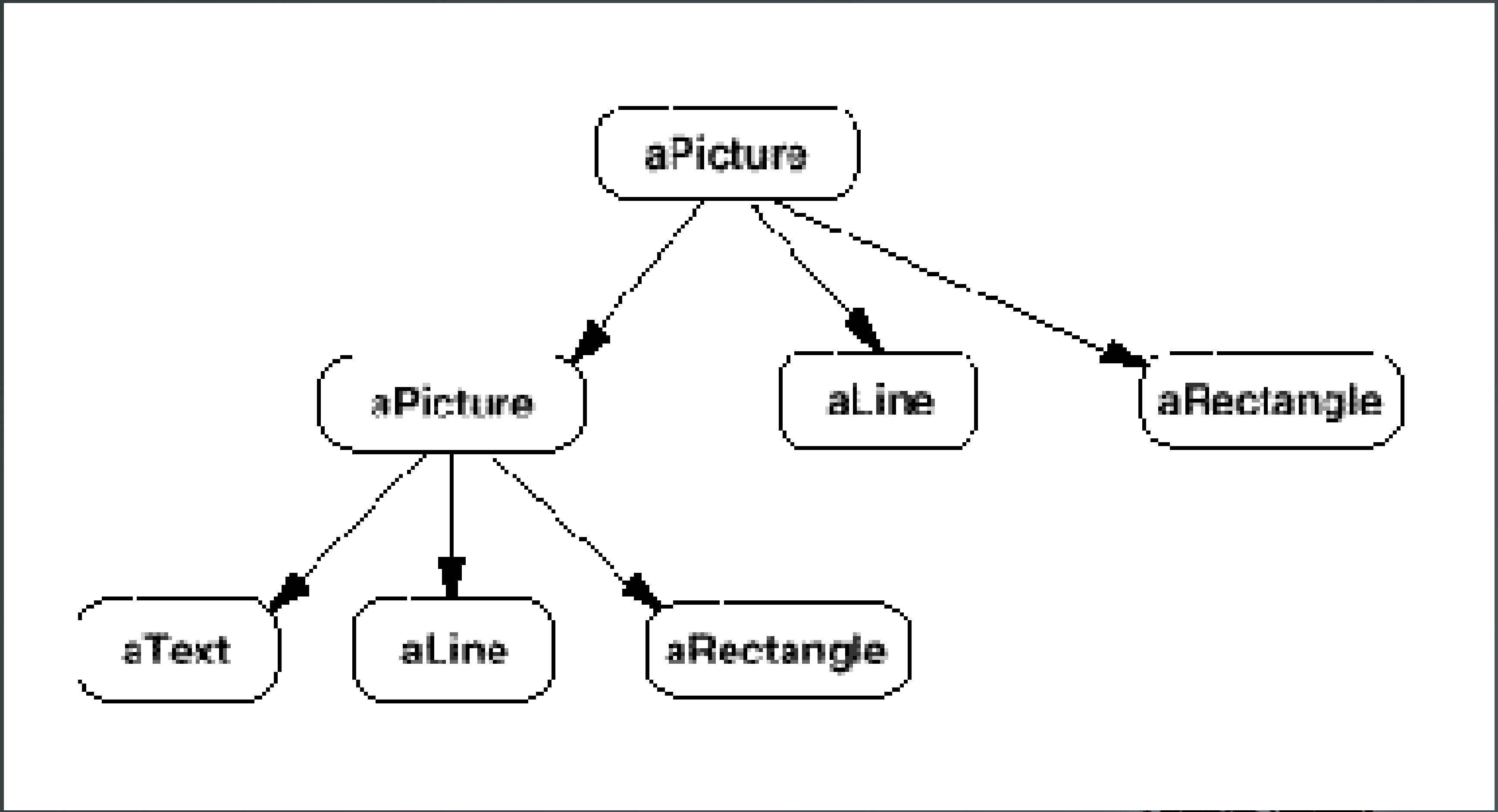
Intent

- Compose objects into tree structures to represent part-whole hierarchies.
- Lets clients treat individual objects and compositions of objects uniformly.

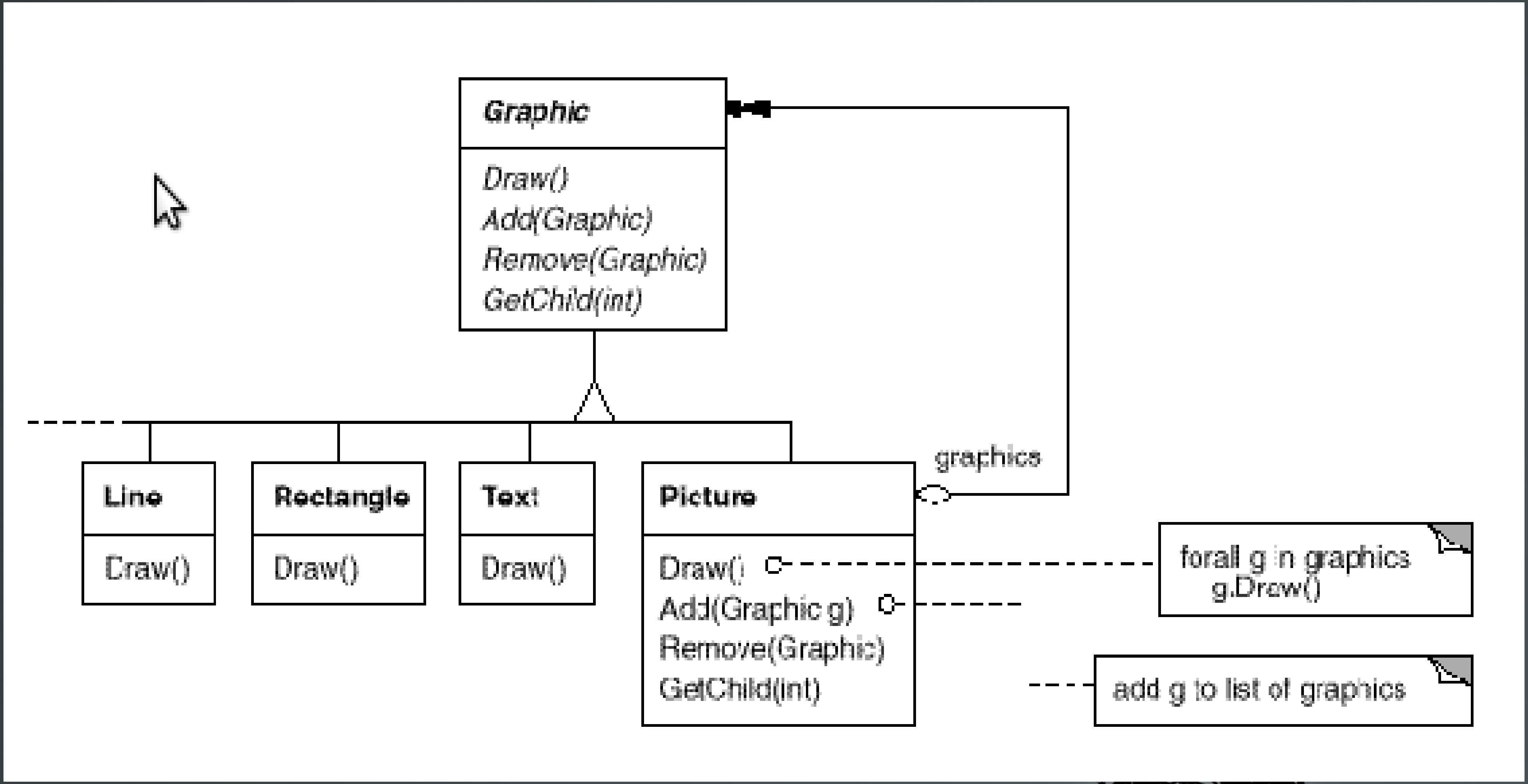
Motivation

- When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone.
- The solution is an interface that allows treating complex and primitive objects uniformly.
- The operations you can perform on all the composite objects often have a least common denominator relationship.

Composite Example



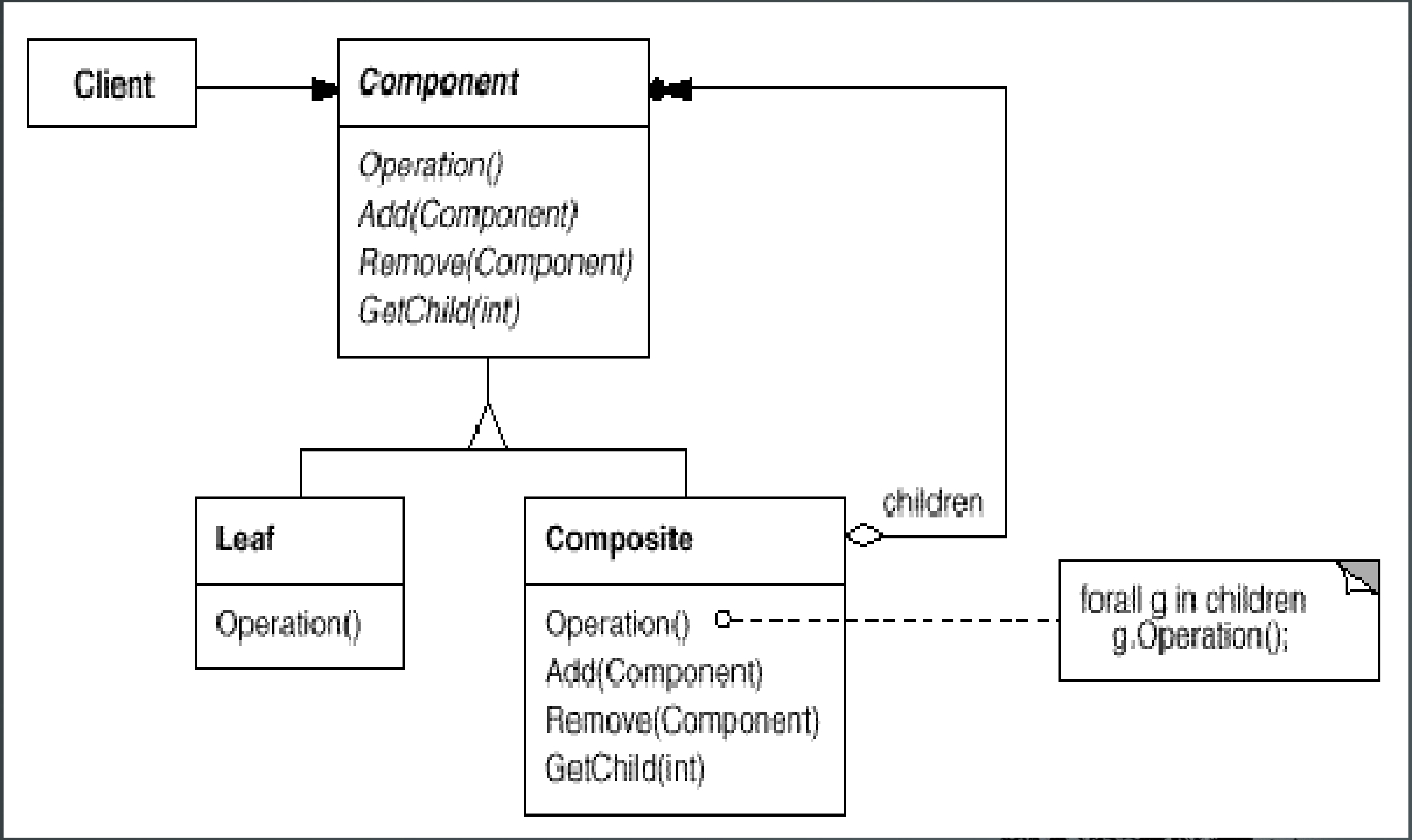
Composite Example



Applicability

- Use the Composite pattern when:
  - You want to represent part-whole hierarchies of objects in a tree like structure.
  - You want to able to handle multiple objects in the same way.
  - You want to add new components to the collection easyly.

Structure


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Participants

- Component:
    - Declares the interface for objects in the composition.
    - Implements default behavior for the interface common to all classes, as appropriate.
    - Declares an interface for accessing and managing its child components.
    - (Optional) Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Participants

- Leaf:
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition.
- Composite:
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child-related operations in the Component interface.

Participants

- Client
  - manipulates objects in the composition through the Componentinterface.