



**SHRI RAMDEOBABA COLLEGE OF
ENGINEERING AND MANAGEMENT,
NAGPUR - 440013**

DESIGN PATTERNS

*V SEMESTER
COURSE COORDINATOR: RINA DAMDOO*

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

STRUCTURAL DESIGN PATTERNS

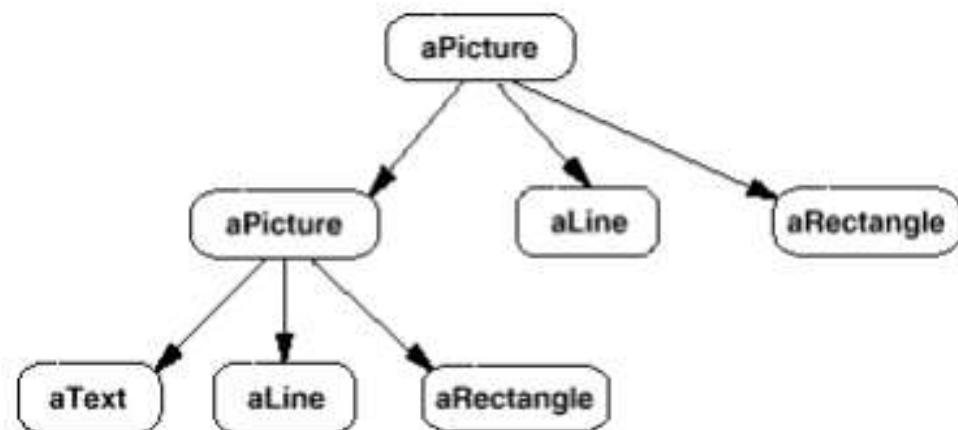
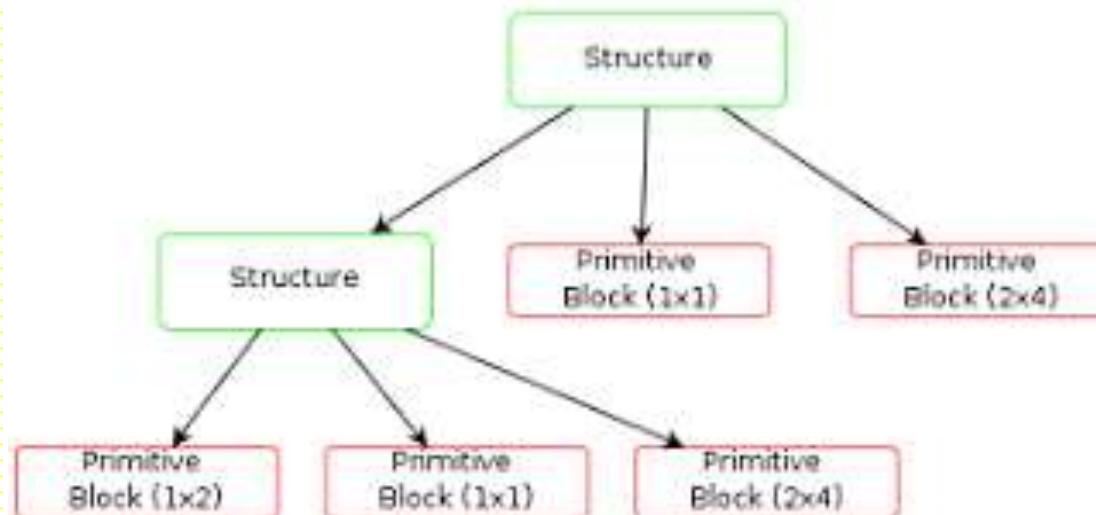
- **Structural patterns** are concerned with how classes and objects are composed to form larger structures.
- **Structural class patterns** use inheritance to compose interfaces or implementations.
- Rather than composing interfaces or implementations, **structural object patterns** describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

COMPOSITE DESIGN PATTERNS

➤ Intent

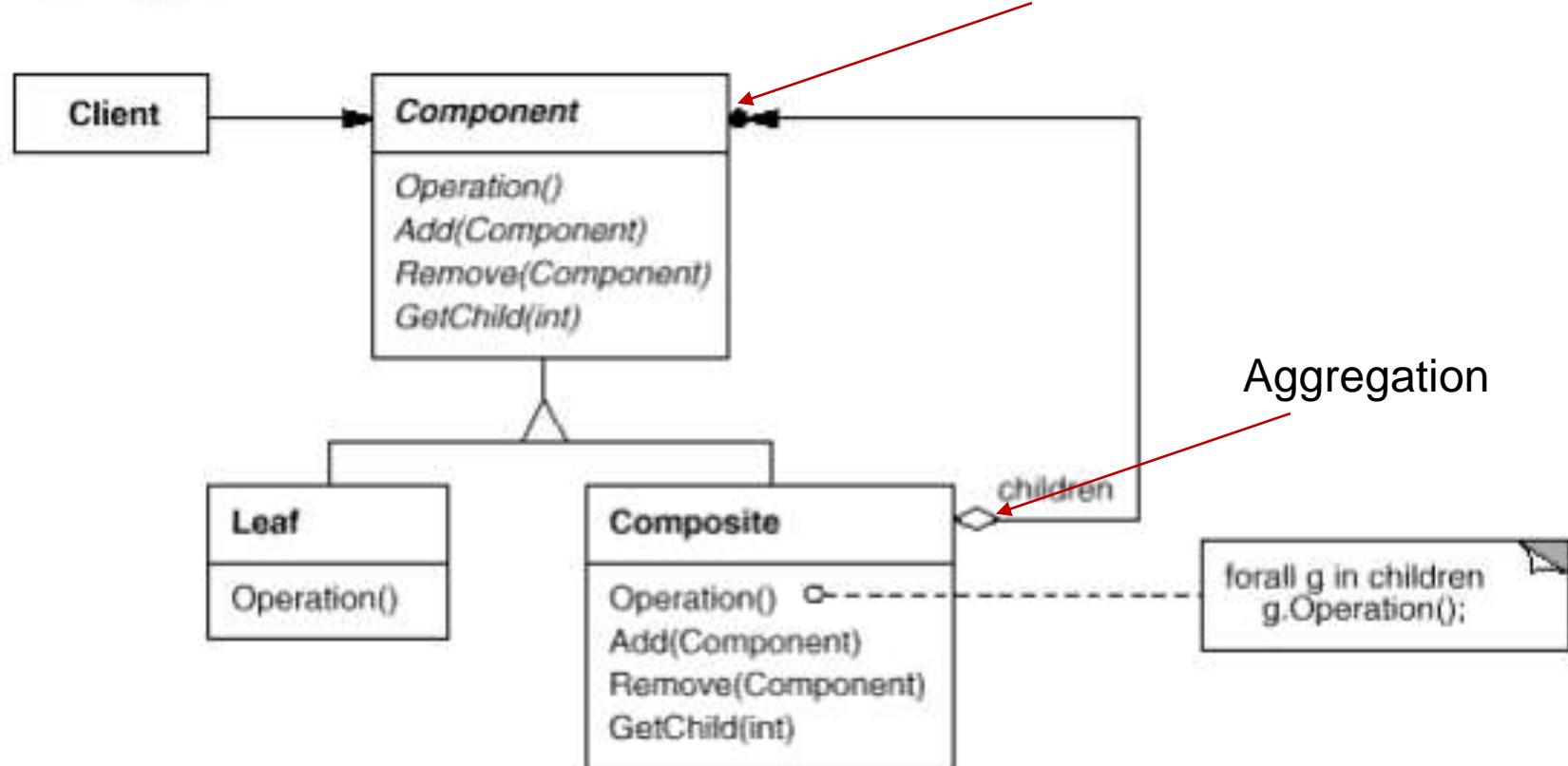
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

COMPOSITE EXAMPLE



COMPOSITE DESIGN PATTERNS

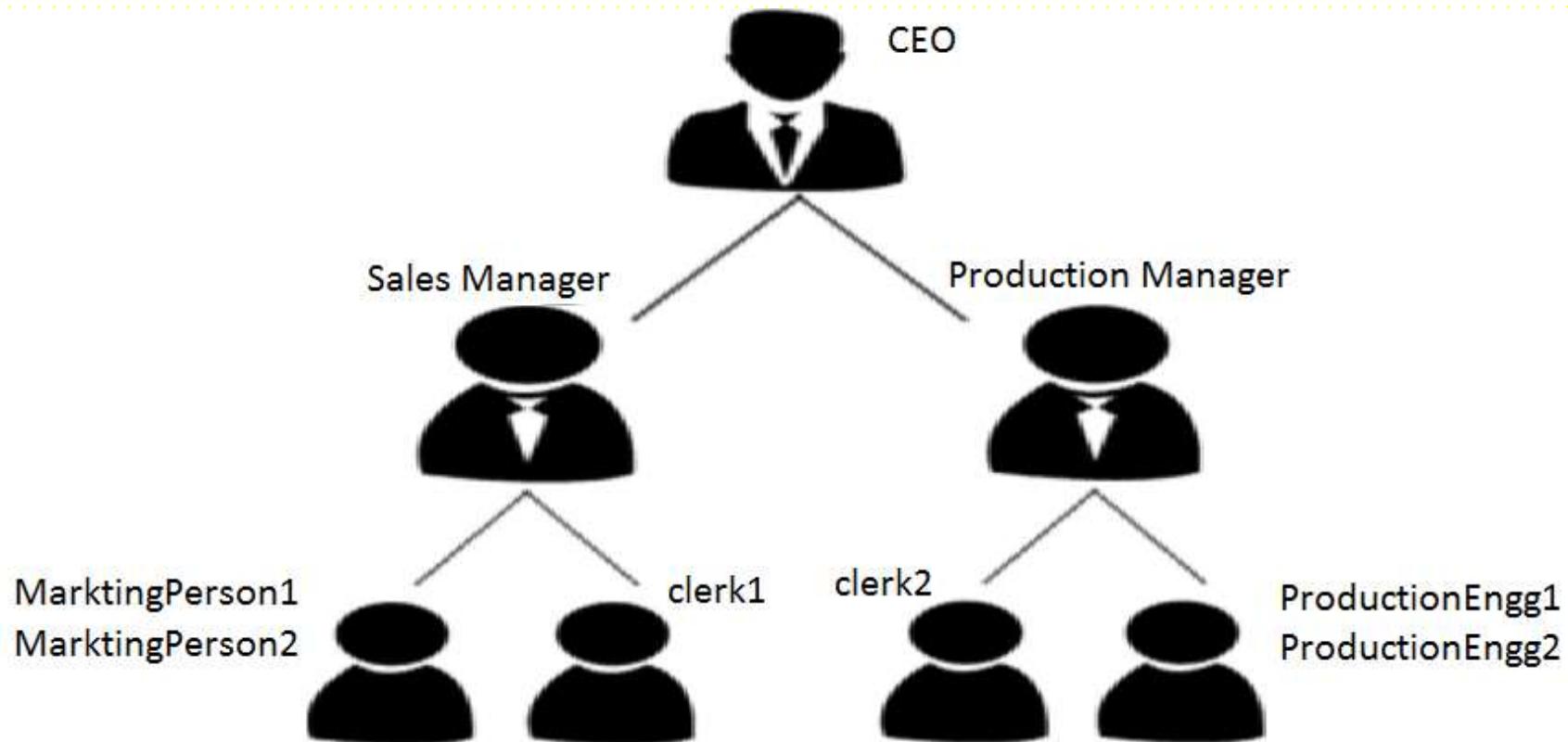
Structure



COMPOSITE EXAMPLE 1

Employee hierarchy in an organization

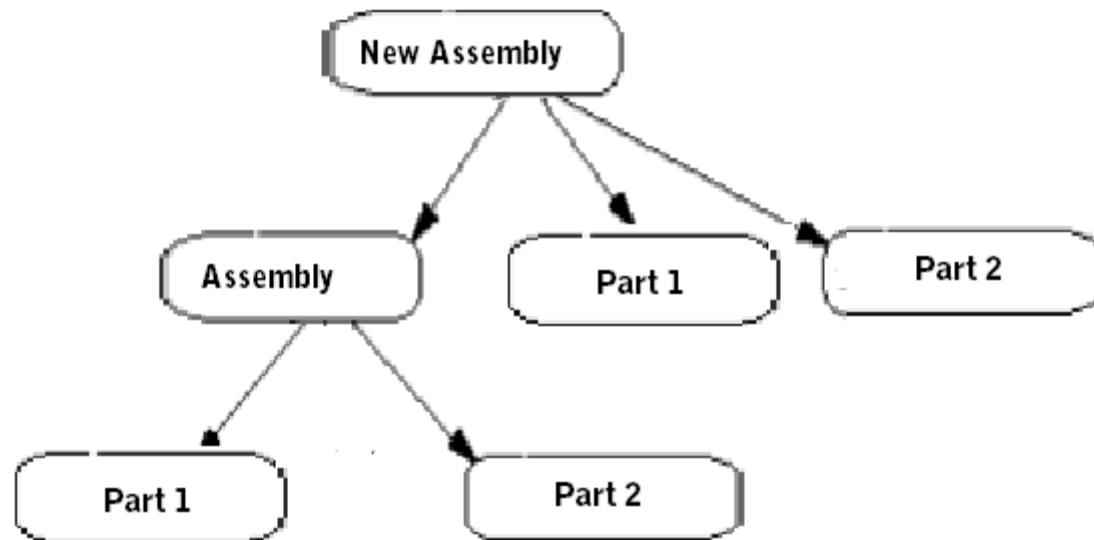
- In the image we have CEO as top node, then managers and then other employees.
- CEO manages the managers, and has group of employees.
- All employees, managers and the CEO are instances of a Employee.



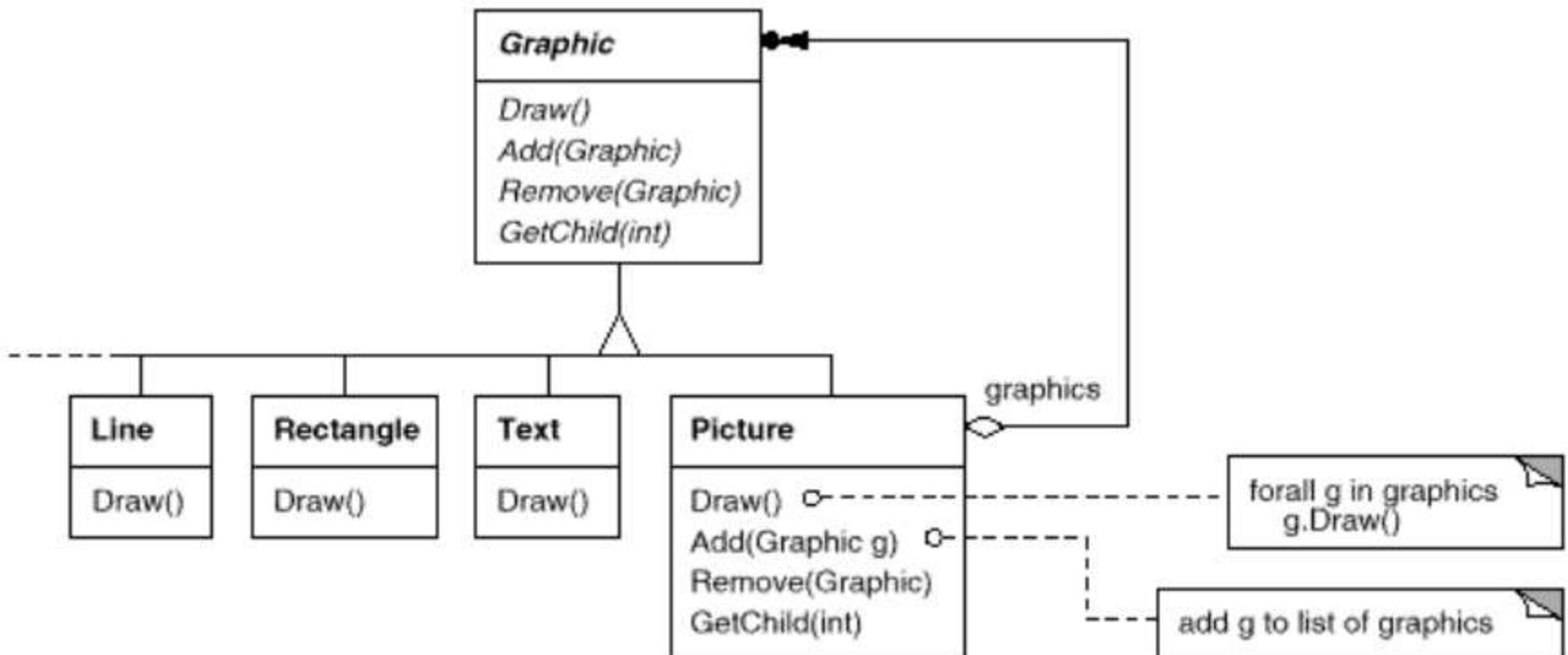
COMPOSITE EXAMPLE 2

Problem Statement:

Create product hierarchy as shown and print costs of a Part or an Assembly as and when demanded by client.



MOTIVATION



APPLICABILITY

Use the Composite pattern when:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

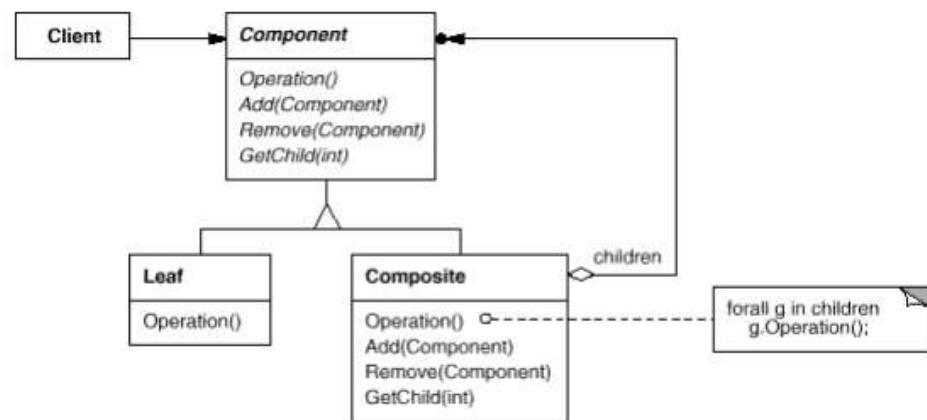
PARTICIPANTS

Participant	Responsibility
Component	<ul style="list-style-type: none">• 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.
Leaf	<ul style="list-style-type: none">• represents leaf objects in the composition. A leaf has no children.• defines behavior for primitive objects in the composition.
Composite	<ul style="list-style-type: none">• defines behavior for components having children.• stores child components.• implements child-related operations in the Component interface.
Client	<ul style="list-style-type: none">• manipulates objects in the composition through the Component interface.

COLLABORATIONS

- Clients use the **Component** class interface to interact with objects in the composite structure.
- If the recipient is a **Leaf**, then the request is handled directly.
- If the recipient is a **Composite**, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

▼ Structure



CONSEQUENCES

- defines class hierarchies consisting of primitive objects and composite objects. Wherever client code expects a primitive object, it can also take a composite object.
- Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code
- Clients don't have to be changed for new Component classes.
- The **disadvantage** of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

IMPLEMENTATION

Implementation Consider the following issues when applying the Composite pattern:

- **Explicit parent references**
- **Sharing components**
- **Maximizing the Component interface**
- **Declaring the child management operations**
- **Should Component implement a list of Components**
- **Child ordering**
- **Caching to improve performance**
- **Who should delete components?**
- **What's the best data structure for storing components?**

KNOWN USES & RELATED PATTERNS

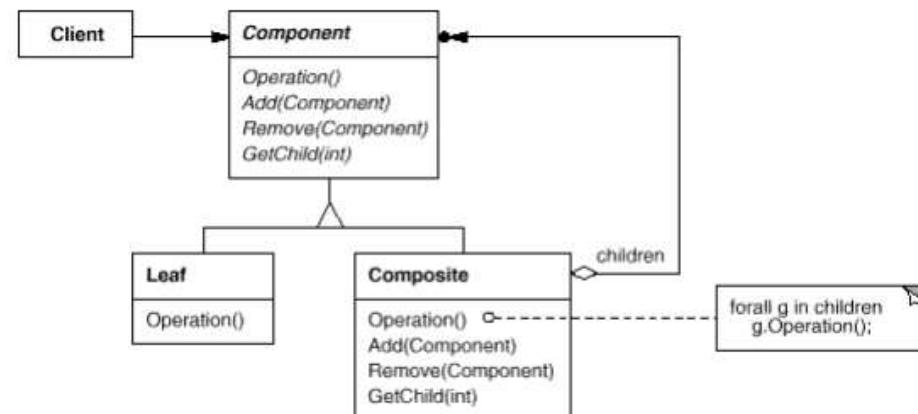
- Examples of the Composite pattern can be found in almost all object-oriented systems
- View class of Model/View/Controller(MVC) DP

- **Chain of Responsibility**
- **Decorator**
- **Iterator**

Does the Composite pattern violate the Interface Segregation Principle?



▼ Structure



- ✓ One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using (to provide transparency).
- ✓ To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible

