CptS 487
Software Design and Architecture

Lesson 12

Design Patterns 3:

Composite

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Instructor: Bolong Zeng

Overview

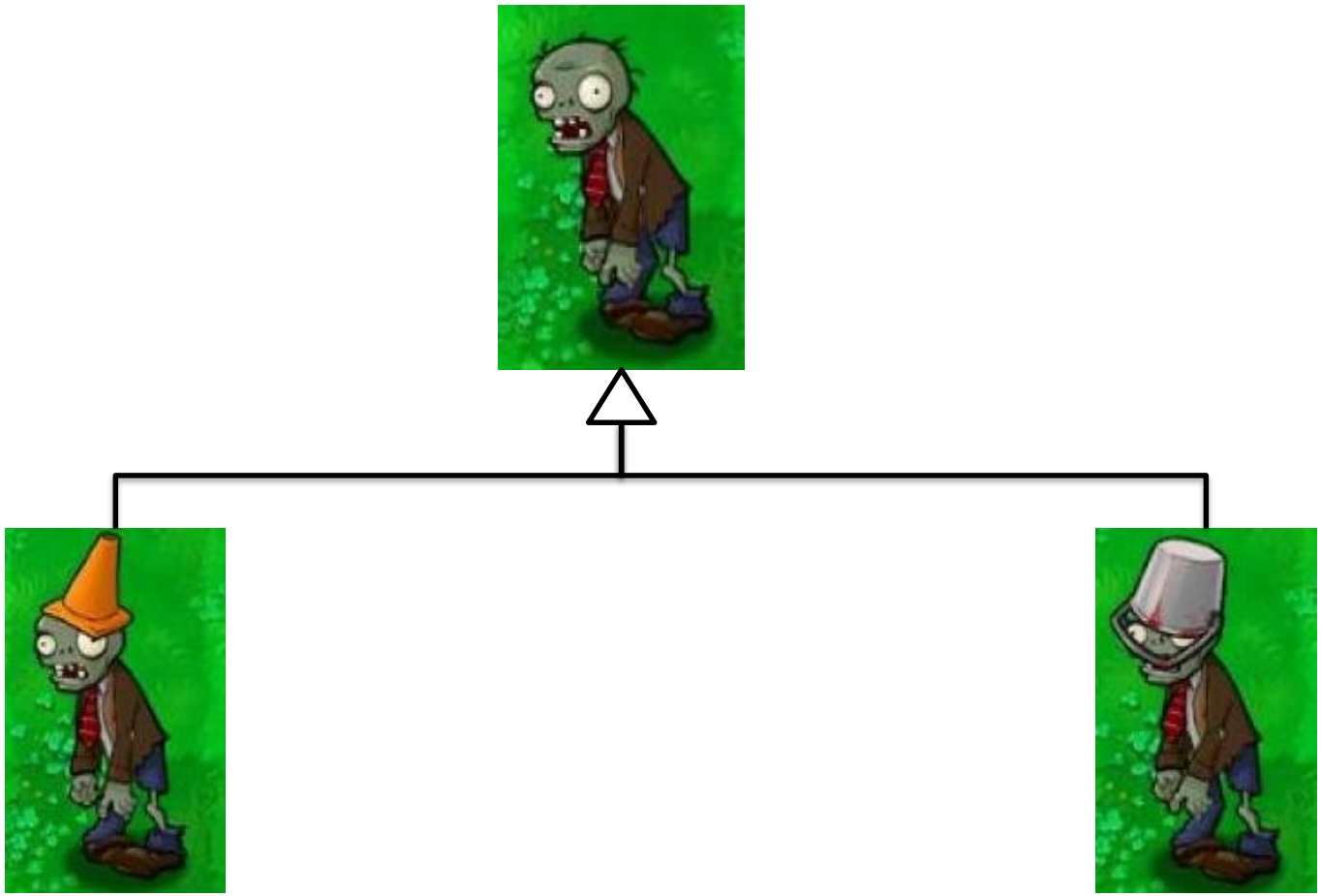
- Introducing a structural design pattern
  - Composite Pattern

Inspiration

- Consider how they take damage from this peashooter, and how to address for changes.
  - takeDamage(int d)

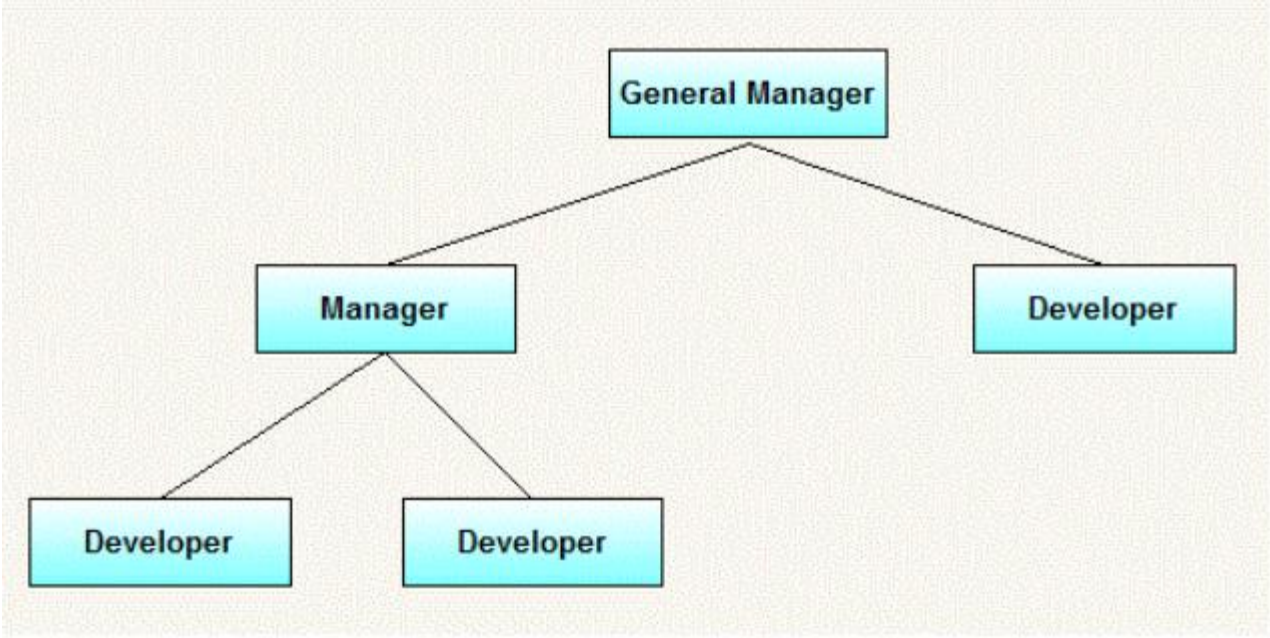
Class Structure

- First look: Direct Inheritance
- Consider an alternative way.

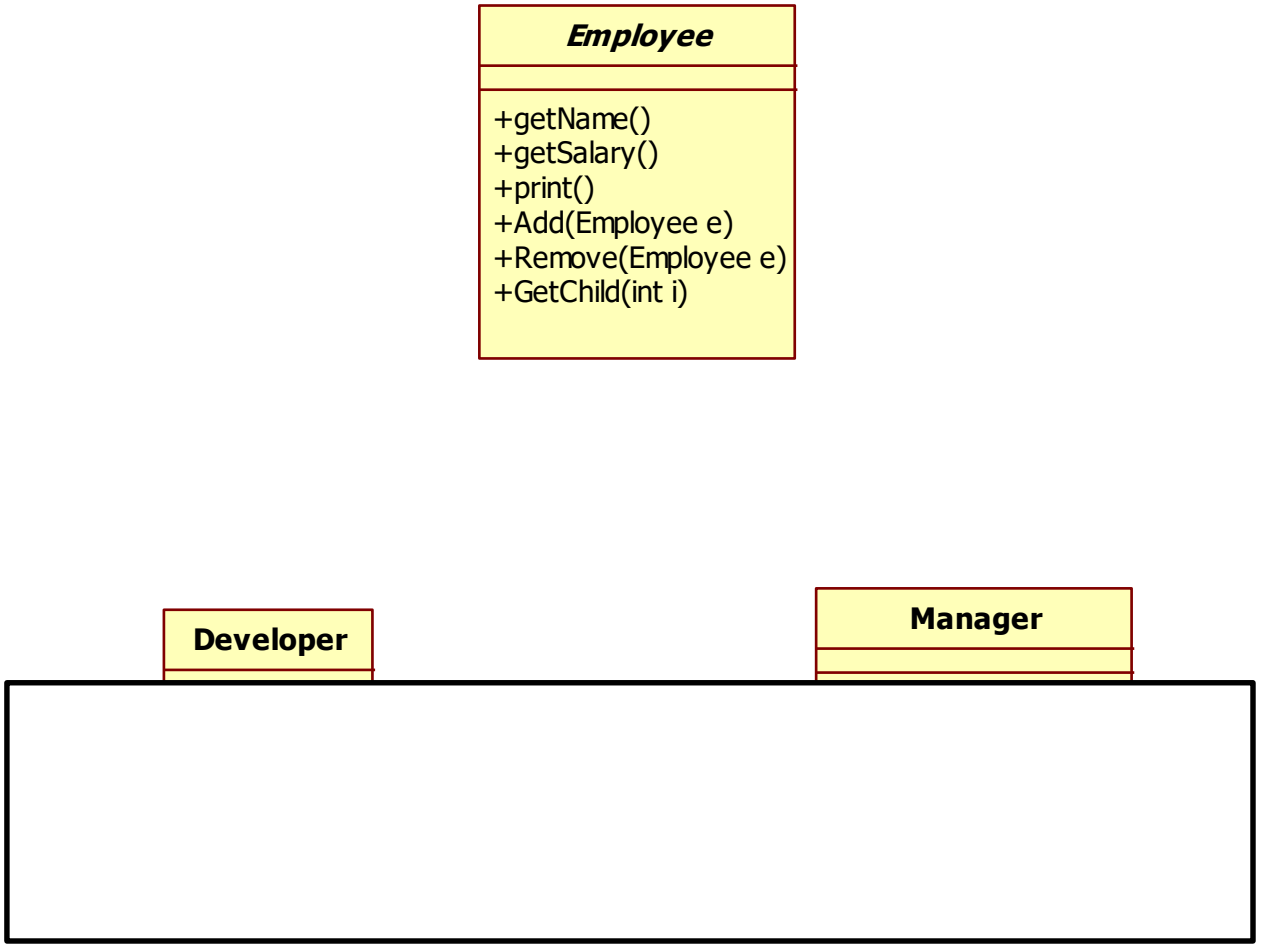


Composite Example

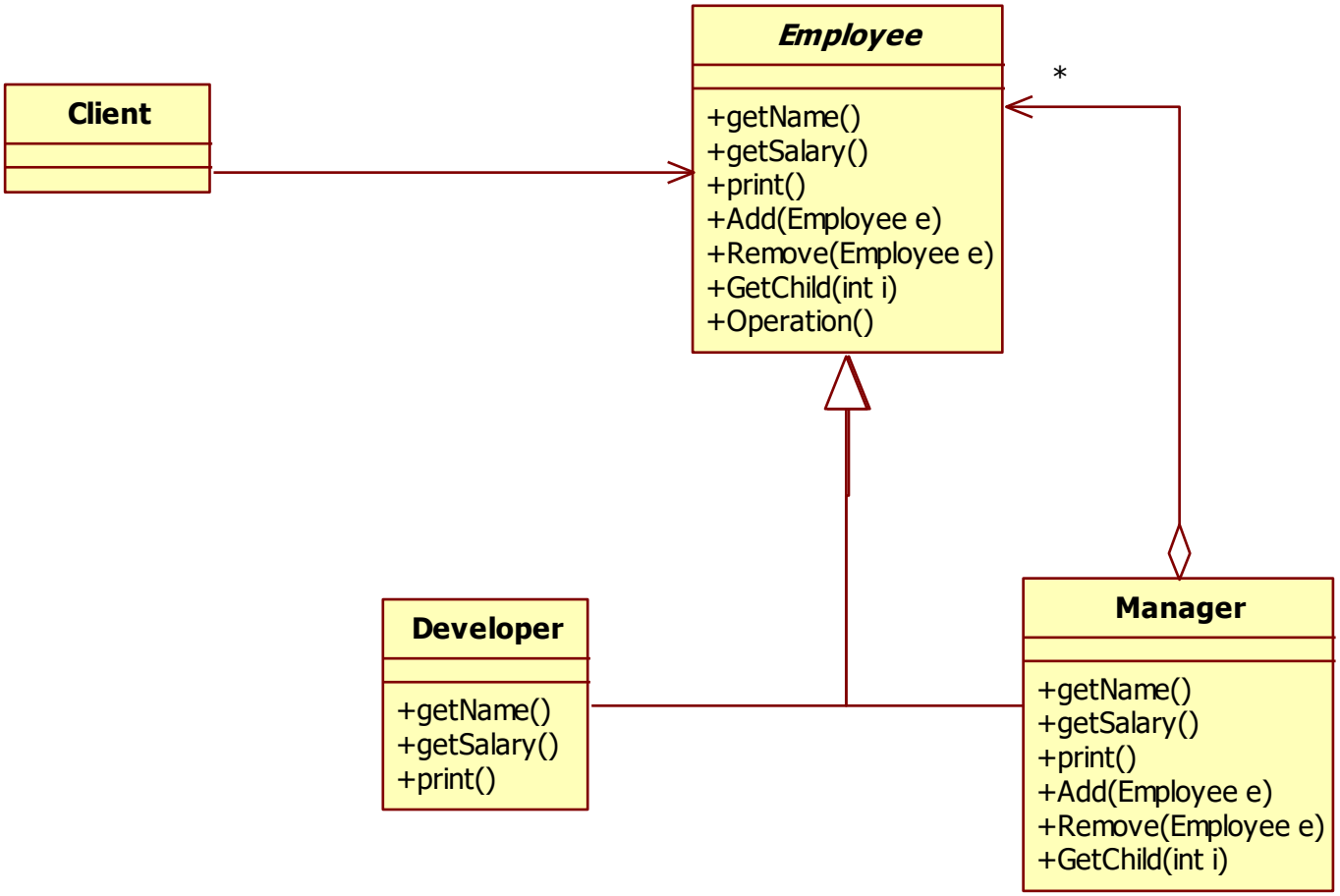
- Problem (cited from java.dzone.com)
  - In a small organization, there are 5 employees. At top position, there is 1 general manager. Under general manager, there are two employees, one is manager and the other is developer and further manager has two developers working under him. We want to print name and salary of all employees from top to bottom.



Composite Example

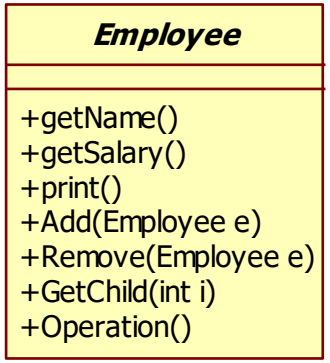


Composite Example

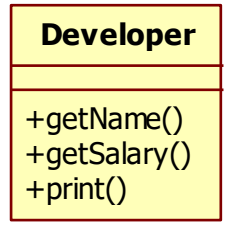


Composite Example

```
03. public interface Employee {
04.
05.   public void add(Employee employee);
06.   public void remove(Employee employee);
07.   public Employee getChild(int i);
08.   public String getName();
09.   public double getSalary();
10.   public void print();
11. }
```




```
07. public class Developer implements Employee{
08.
09.   private String name;
10.   private double salary;
11.
12.   public Developer(String name,double salary){
13.     this.name = name;
14.     this.salary = salary;
15.   }
25.   public String getName() {
26.     return name;
27.   }
28.
29.   public double getSalary() {
30.     return salary;
31.   }
32.
33.   public void print() {
34.     System.out.println("-------------");
35.     System.out.println("Name ="+getName());
36.     System.out.println("Salary ="+getSalary());
37.     System.out.println("-------------");
38.   }
```




```
16.   public void add(Employee employee) {
17.     //this is leaf node so this method is not applicable to this class.
18.   }
19.
20.   public Employee getChild(int i) {
21.     //this is leaf node so this method is not applicable to this class.
22.     return null;
23.   }
40.   public void remove(Employee employee) {
41.     //this is leaf node so this method is not applicable to this class.
42.   }
```

Composite Example

```
07. public class Manager implements Employee{
08.
09.  private String name;
10.  private double salary;
11.
12.  public Manager(String name,double salary){
13.   this.name = name;
14.   this.salary = salary;
15.  }
26.  public String getName() {
27.   return name;
28.  }
29.
30.  public double getSalary() {
31.   return salary;
32.  }
```

Manager

+getName()
+getSalary()
+print()
+Add(Employee e)
+Remove(Employee e)
+GetChild(int i)

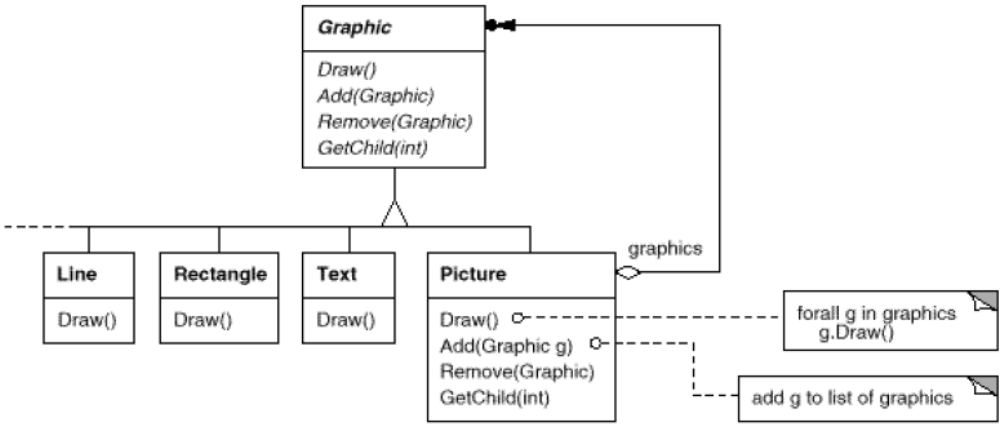
```
17.  List<Employee> employees = new ArrayList<Employee>();
18.  public void add(Employee employee) {
19.    employees.add(employee);
20.  }
47.  public void remove(Employee employee) {
48.   employees.remove(employee);
49.  }
22.  public Employee getChild(int i) {
23.   return employees.get(i);
24.  }
```

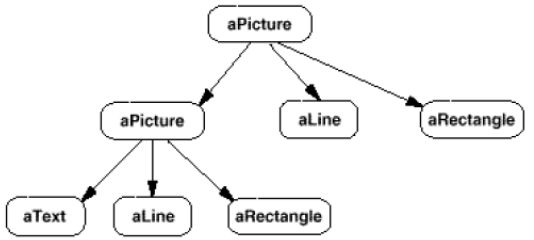
```
34.  public void print() {
35.   System.out.println("-------------");
36.   System.out.println("Name ="+getName());
37.   System.out.println("Salary ="+getSalary());
38.   System.out.println("-------------");
39.
40.
41.
42.
43.
44.    }
45.  }
```

5. Composite (Object structural pattern)

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

- Motivation
  - Describes how to use recursive composition so that clients don't have to make the distinction between primitive objects and container objects.

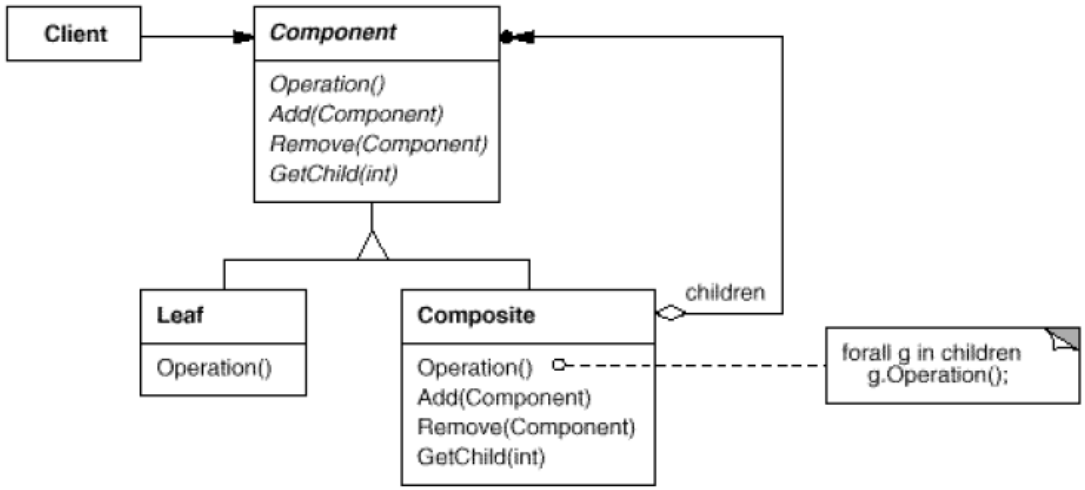




5. Composite (Object structural pattern)

- Applicability
  - Represent part-whole hierarchies of objects.
  - Let 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.

- Structure

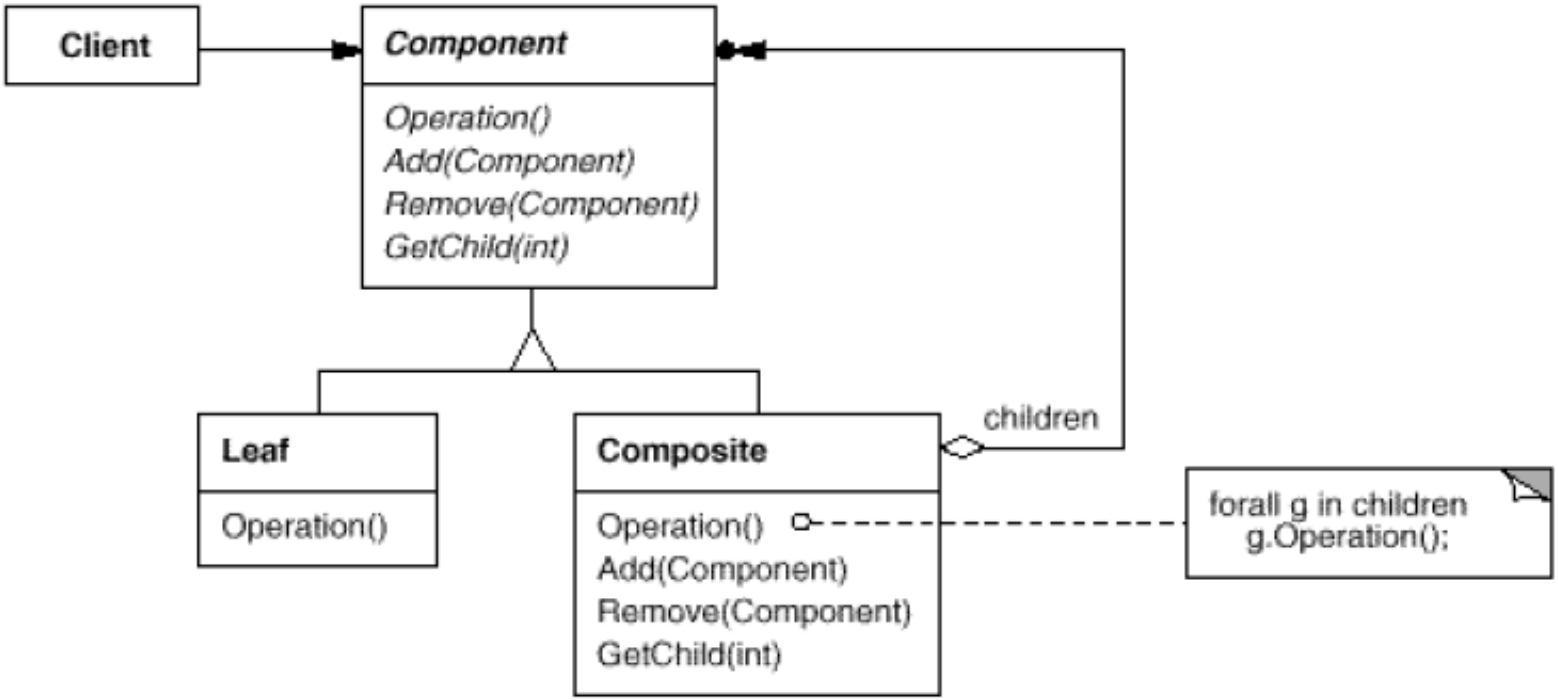


5. Composite (Object structural pattern)

- 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.
  - 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.
  - Client
    - Manipulates objects in the composition through the Component interface.

5. Composite (Object structural pattern)

- Structure



5. Composite (Object structural pattern)

- Consequences
  - Benefits
    - Class hierarchies consisting of primitive objects and composite objects.
    - Make the client simple.
    - Easier to add new kinds of components. No need to change the clients.
  - Liabilities
    - Make your design overly general: harder to restrict the components of a composite.

5. Composite (Object structural pattern)

- Implementation Issues
  - 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?