# Composite

# Composite

Composite is a structural design pattern that lets you compose objects into tree structures and allow clients to work with these structures as if they were individual objects.

The composite pattern makes sense only when your business model can be represented as a tree.

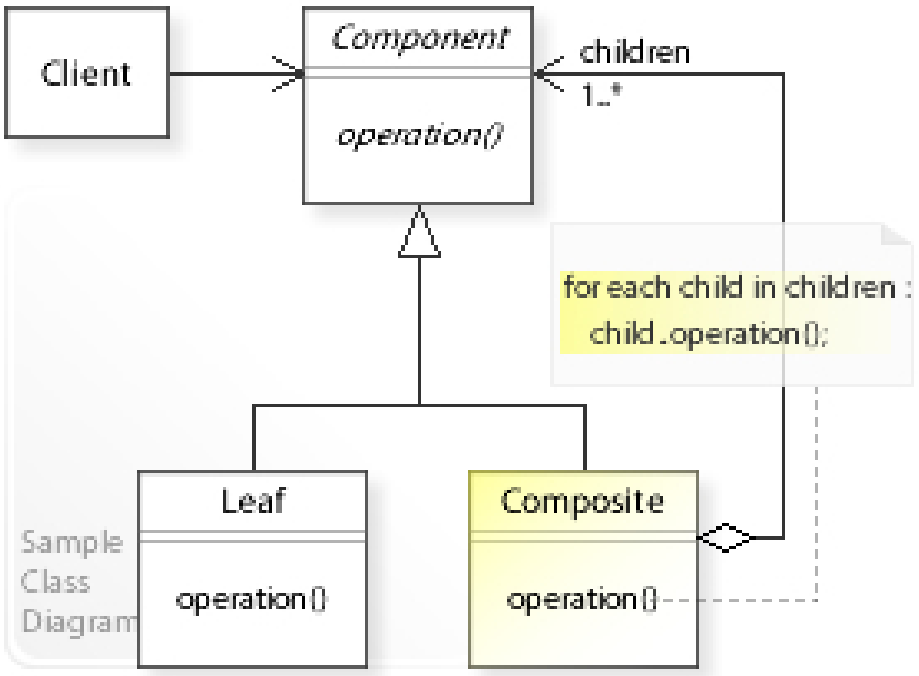
# Real-World Analogy

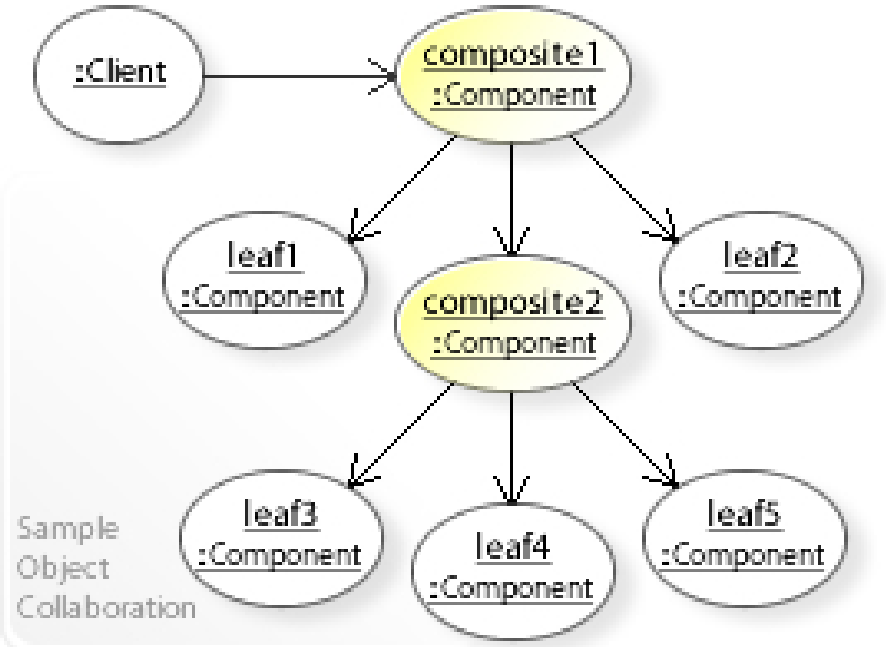
Military structure

Armies of most countries look like composite trees. On the lowest level, there are soldiers. They are grouped into squads. Several squads make a platoon. Platoons make a division. And finally, several divisions make an army.

Orders are given at the top of the hierarchy and passed down at each level until every soldier knows what needs to be done.

# UML diagrams



# Composite overview

**What problems can the Composite design pattern solve?**

- A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
- A part-whole hierarchy should be represented as tree structure.

# Example

```
class Component
{
  public:
    virtual void traverse() = 0;
};

class Leaf: public Component
{
    // 1. Scalar class   3. "isa" relationship
    int value;
  public:
    Leaf(int val)
    {
        value = val;
    }
    void traverse()
    {
        cout << value << ' ';
    }
};
```

```
class Composite: public Component
{
    // 1. Vector class   3. "isa" relationship
    vector < Component * > children; // 4. "container" coupled to the interface
  public:
    // 4. "container" class coupled to the interface
    void add(Component *ele)
    {
        children.push_back(ele);
    }
    void traverse()
    {
        for (int i = 0; i < children.size(); i++)
        // 5. Use polymorphism to delegate to children
          children[i]->traverse();
    }
};

int main()
{
  Composite containers[4];

  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
      containers[i].add(new Leaf(i *3+j));

  for (i = 1; i < 4; i++)
    containers[0].add(&(containers[i]));

  for (i = 0; i < 4; i++)
  {
    containers[i].traverse();
    cout << endl;
  }
}
```

**Output**

```
0 1 2 3 4 5 6 7 8 9 10 11
3 4 5
6 7 8
9 10 11
```

# Pros and Cons

✔ Simplifies the client code that has to interact with a complex tree structure.

✔ Makes easier adding new component types.

✗Creates a too general class design.

# Sources

- https://sourcemaking.com/design_patterns/composite
- https://en.wikipedia.org/wiki/Composite_pattern
- https://refactoring.guru/design-patterns/composite