COMPOSITE PATTERN

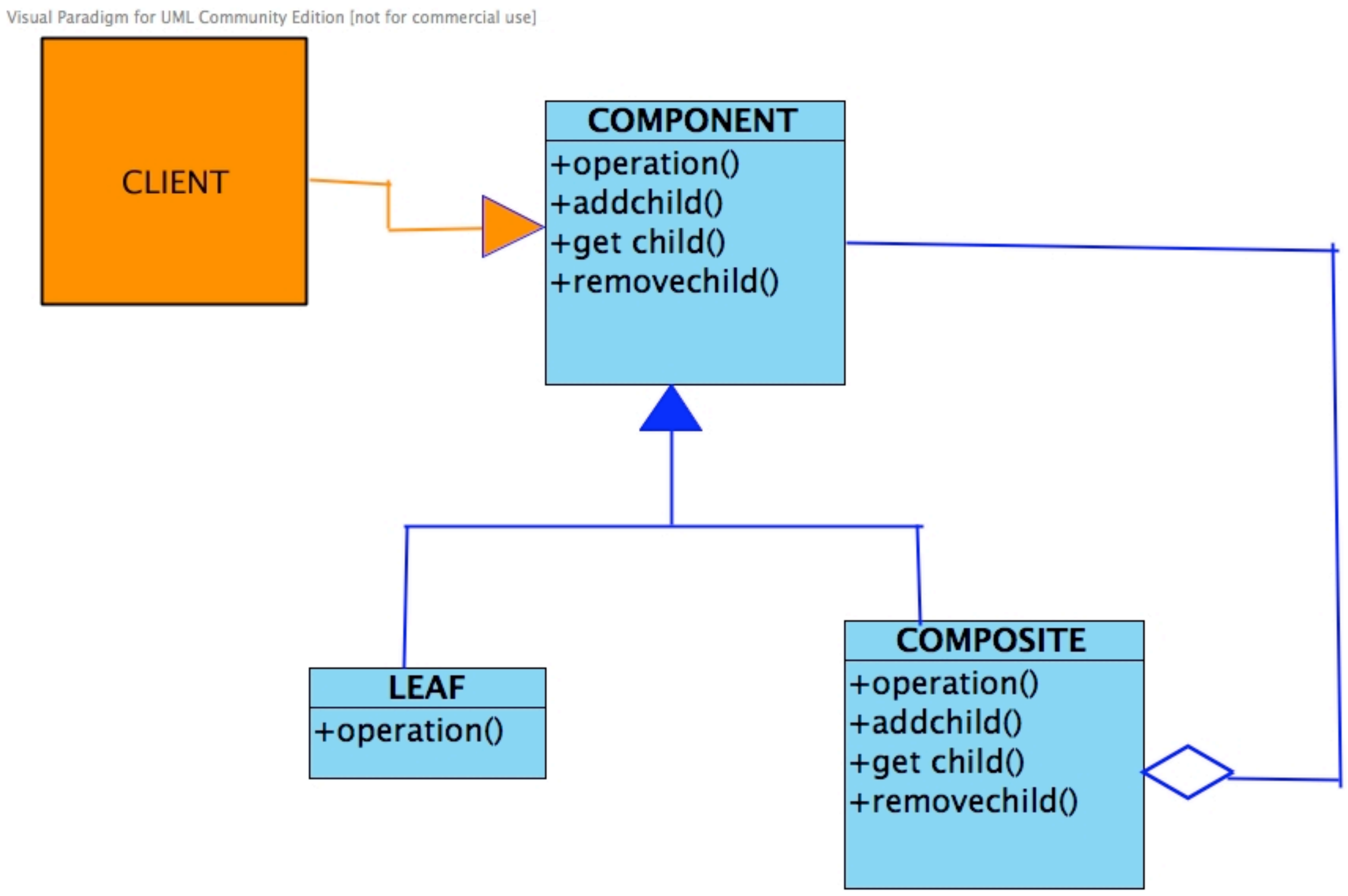
A Structural Pattern

The Basics

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies
  - Treat individual objects and compositions of objects uniformly

- Why Use?
  - Can interact with all the “participants” in the same way
  - Client can ignore the difference between compositions of objects and individual objects
  - Easier to add new components

STRUCTURE



Visual Paradigm for UML Community Edition [not for commercial use]
CLIENT
COMPONENT
+operation()
+addchild()
+get child()
+removechild()
LEAF
+operation()
COMPOSITE
+operation()
+addchild()
+get child()
+removechild()

Key

Create an abstract class that represents both primitives and their container (i.e. the individual objects and a specific collection of objects)

Abstract class declares operations that are:

1. Specific to the composite
2. But that are also shared by all the composite objects (as a composite can be made up of several composite objects)

Who Does What

Component – Base class or interface

1. Implements default behavior for the interface common to all classes
2. Declares interface for accessing and managing child components

Composite

1. Extends the component class
2. Ability to pass a composite to a method that might be expecting a component

Leaf: Defines behavior for primitive objects in composition

Consequences

Benefits

- It makes clients simpler, since they do not have to know if they are dealing with a leaf or a composite component.
- It makes it easy to add new kinds of components.
- It is helpful when dealing with recursive data structures.

Disadvantages

- The design can be overly general - It makes it harder to restrict the type of components of a composite.

The Tangible World

- Equipment is usually made of various parts. Client may want to know the cost of the entire composite or just a sub-component.
  - Recursive in that a sub-part of a computer may be made up of several other parts

- A drawing consists of multiple objects. An object – such as a square – requires lines. A line requires points. Depending on what type of drawing you need, you combine the objects in various different ways,

  (But the disadvantage means a pencil could be mistakenly added to a drawing,)

In the Code World

interface Component { }

Class Composite implements **Component** { }

Class **LeafAA** implements **Component**

-
-
-
-

Class **LeafJZ** extends **Component**

Very easy to add additional leafs

```
interface Equip {
     float calculate ();
     float getprice();
     float calmargin();
}
```

Component

Composite

```
class Module implements Equip {

    float price;
    List<Equip> mylist= new ArrayList<Equip>();

    public void add (Equip e){
        mylist.add(e);
    }

    public void remove (Equip e){
        mylist.remove(e);
    }

public float calculate(){
        Equip part=null;
        float total=0;
    Iterator<Equip>iterator=mylist.iterator();
        while(iterator.hasNext()) {
            part=iterator.next();
            float gmp=part.calculate();
            total=gmp+total;
        }
        return total;
}
```

Ability to add and remove items from the Composite

Create a list and an iterator

```
public float getprice(){
        Equip part=null;
        float totalp=0;

    Iterator<Equip>iterator=mylist.iterator();
            while(iterator.hasNext()) {
                part=iterator.next();
                float pricc=part.getprice();
                totalp=pricc+totalp;
        }
            return totalp;
    }

    public float calmargin() {

        float margin=this.calculate()/this.getprice();
        return margin;
    }
}
```

```
class Hugepart implements Equip {
    private float price;
    private float cost;

    public Hugepart(float price, float cost){
        this.price=price;
        this.cost=cost;
    }

    public float calculate() {
        float profit=this.price-this.cost;
        return profit;
    }

    public float getprice() {
        float A =this.price;
        return A;
}

    public float calmargin() {
        float margin= (this.price-
this.cost)/this.price;
        return margin;
    }
}
```

Leaf

If there was another Leaf (such as Tinypart), the formula used to calculate gross margin could be different.

For example, could add a discount to the price

When Implementing, Consider….

- Where should the child methods be declared?
  - Transparency versus safety
  - Safer in the composite - At run time, clients cannot change components' methods.
  - Transparency is greater in the Component – All components have same interface.

- Maximize Component Interface
  - But putting them in the component violates class hierarchy rules that says a class should only define operations that are meaningful to subclasses. (Some operations in Composite not relevant to leaf.)

Related patterns

- **Iterator:** Traverse composites

- **Visitor:** Localizes operations and behavior that would otherwise be distributed to composite and leaf classes

- **Wrapper/Decorator:** Support the Component interface with operations like Add, Remove, GetChild