Composite

Linda Marshall and Vreda Pieterse

Department of Computer Science
University of Pretoria

27 August 2014

Overview

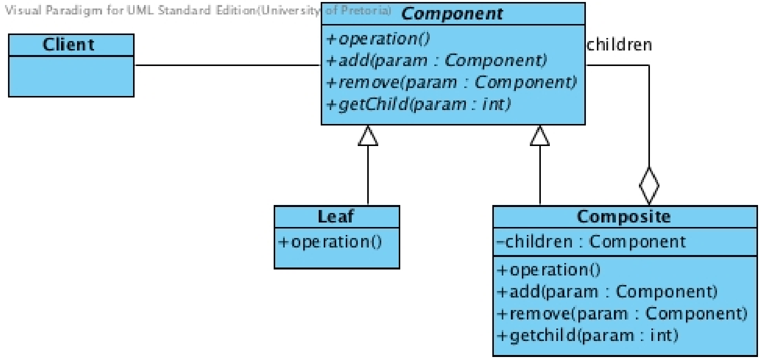
Name and Classification:

Composite (Object Structural)

Intent:

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

GoF(163)


Visual Paradigm for UML Standard Edition(University of Pretoria)
Component
+operation()
+add(param : Component)
+remove(param : Component)
+getChild(param : int)
Client
children
Leaf
+operation()
Composite
-children : Component
+operation()
+add(param : Component)
+remove(param : Component)
+getchild(param : int)

Component

- provides the interface with which the client interacts

Leaf

- do not have children, define the primitive objects of the composition

Composite

- contain children that are either composites or leaves

Client

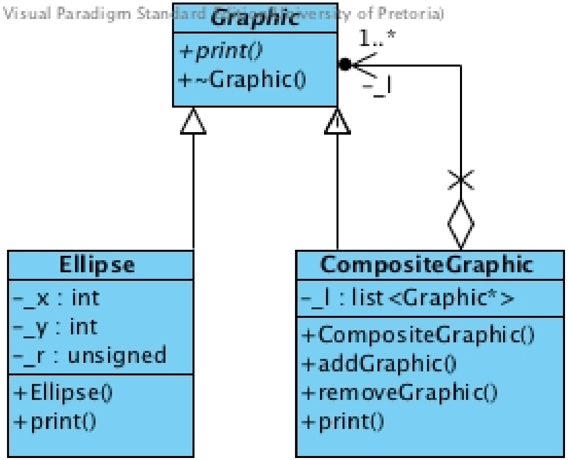
- manipulates the objects that comprise the composite

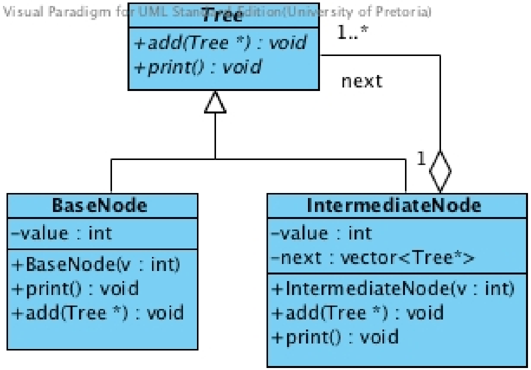
- Used in hierarchies where some objects are composites of others
- Makes use of a store for the children defined by Composite

Related Patterns

- **Chain of Responsibility** (223) : component-parent link.
- **Decorator** (175): Used in conjunction with components. Usually share the same parent class.

- **Flyweight** (195): Allows sharing of objects, particularly the leaf nodes.
- **Iterator** (257) and **Visitor** (331): Used to traverse the composite structure.


Visual Paradigm Standard Edition(University of Pretoria)
Graphic
+print()
+~Graphic()
1..*
-_l
Ellipse
-_x : int
-_y : int
-_r : unsigned
+Ellipse()
+print()
CompositeGraphic
-_l : list<Graphic*>
+CompositeGraphic()
+addGraphic()
+removeGraphic()
+print()


Visual Paradigm for UML Standard Edition(University of Pretoria)
Tree
+add(Tree *) : void
+print() : void
1..*
next
1
BaseNode
-value : int
+BaseNode(v : int)
+print() : void
+add(Tree *) : void
IntermediateNode
-value : int
-next : vector<Tree*>
+IntermediateNode(v : int)
+add(Tree *) : void
+print() : void

```
class Tree {
  public:
    virtual void add(Tree*) = 0;
    virtual void print() = 0;
    virtual ~Tree() {}; // Added
};
```

```
class BaseNode : public Tree {
  public:
    BaseNode(int v) : value(v) {};
    virtual void print() {...};
    virtual void add(Tree*) {};
    virtual ~BaseNode() {}; // Added
private:
  int value; };
```

```
class IntermediateNode: public Tree {
  public:
    ...
    // Add
    virtual ~IntermediateNode();
  private:
    ...
};
```

```
IntermediateNode::~IntermediateNode(){
  vector<Tree*>:: iterator it;

  for (it = next.begin(); it != next.e
    delete *it;
}
```

```
delete b;
        // Not linked into Tree t
        // and therefore needs to
        // be deleted separately
delete t;
```