THE COMPOSITE PATTERN

Frequently programmers develop systems in which a component may be an individual object or it may represent a collection of objects. The Composite pattern is designed to accommodate both cases. You can use the Composite to build part-whole hierarchies or to construct data representations of trees. In summary, a composite is a collection of objects, any one of which may be either a composite, or just a primitive object. In tree nomenclature, some objects may be nodes with additional branches and some may be leaves.

The problem that develops is the dichotomy between having a single, simple interface to access all the objects in a composite, and the ability to distinguish between nodes and leaves. Nodes have children and can have children added to them, while leaves do not at the moment have children, and in some implementations may be prevented from having children added to them.

Some authors have suggested creating a separate interface for nodes and leaves, where a leaf could have the methods

```
public String getName();
public String getValue();
```

and a node could have the additional methods:

```
public Enumeration elements();
public Node getChild(String nodeName);
public void add(Object obj);
public void remove(Object obj);
```

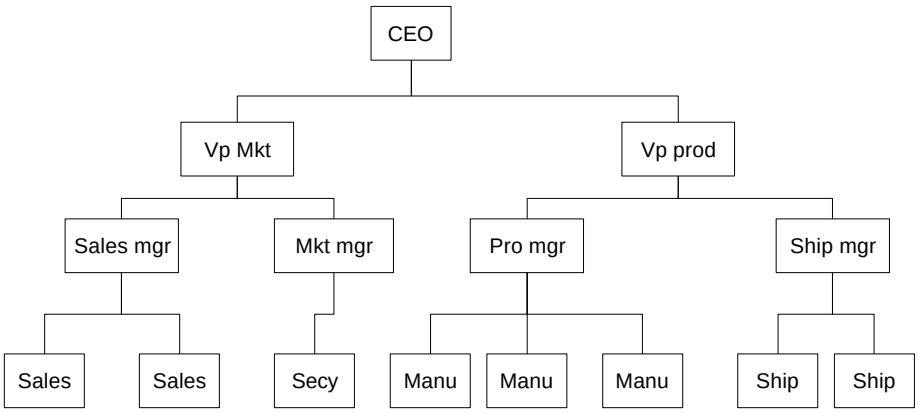
This then leaves us with the programming problem of deciding which elements will be which when we construct the composite. However, *Design Patterns* suggests that each element should have the *same* interface, whether it is a composite or a primitive element. This is easier to accomplish, but we are left with the question of what the *getChild()* operation should accomplish when the object is actually a leaf.

Java makes this quite easy for us, since every node or leaf can return an Enumeration of the contents of the Vector where the children are stored. If there are no children, the *hasMoreElements()* method returns false at once. Thus, if we simply obtain the Enumeration from each element, we can

quickly determine whether it has any children by checking the *hasMoreElements()* method.

An Implementation of a Composite

Let's consider a small company. It may have started with a single person who got the business going. He was, of course, the CEO, although he may have been too busy to think about it at first. Then he hired a couple of people to handle the marketing and manufacturing. Soon each of them hired some additional assistants to help with advertising, shipping and so forth, and they became the company's first two vice-presidents. As the company's success continued, the firm continued to grow until it has the organizational chart we see below:



Now, if the company is successful, each of these company members receives a salary, and we could at any time ask for the cost of any employee to the company. We define the cost as the salary of that person and those of all his subordinates. Here is an ideal example for a composite:

- The cost of an individual employee is simply his salary (and benefits).
- The cost of an employee who heads a department is his salary plus those of all his subordinates.

We would like a single interface that will produce the salary totals correctly whether the employee has subordinates or not.

```
public float getSalaries();
```

At this point, we realize that the idea of all Composites having the same standard interface is probably naïve. We'd prefer that the public methods be related to the kind of class we are actually developing. So rather than have generic methods like *getValue()*, we'll use *getSalaries()*.

The Employee Class

Our Employee class will store the name and salary of each employee, and allow us to fetch them as needed.

```
public class Employee
{
   String name;
   float salary;
   Vector subordinates;
//---------------------------------------
   public Employee(String _name, float _salary)    {
      name = _name;
      salary = _salary;
      subordinates = new Vector();
   }
   //---------------------------------------
   public float getSalary()    {
      return salary;
   }
   //---------------------------------------
   public String getName()    {
      return name;
   }
```

Note that we created a Vector called *subordinates* at the time the class was instantiated. Then, if that employee has subordinates, we can automatically add them to the Vector with the *add* method and remove them with the *remove* method.

```
   public void add(Employee e)    {
         subordinates.addElement(e);
   }
   //---------------------------------------
   public void remove(Employee e)    {
       subordinates.removeElement(e);
   }
```

If you want to get a list of employees of a given supervisor, you can obtain an Enumeration of them directly from the subordinates Vector:

```
public Enumeration elements()    {
      return subordinates.elements();
}
```

The important part of the class is how it returns a sum of salaries for the employee and his subordinates:

```
public float getSalaries()    {
   float sum = salary;                  //this one's salary
 //add in subordinates salaries
 for(int i = 0; i < subordinates.size(); i++) {
   sum +=
    ((Employee)subordinates.elementAt(i)).getSalaries();
return sum;
}
```

Note that this method starts with the salary of the current Employee, and then calls the *getSalaries()* method on each subordinate. This is, of course, recursive and any employees which themselves have subordinates will be included.

Building the Employee Tree

We start by creating a CEO Employee and then add his subordinates and their subordinates as follows:

```
boss = new Employee("CEO", 200000);
boss.add(marketVP =
       new Employee("Marketing VP", 100000));
boss.add(prodVP =
       new Employee("Production VP", 100000));
marketVP.add(salesMgr =
       new Employee("Sales Mgr", 50000));
marketVP.add(advMgr =
       new Employee("Advt Mgr", 50000));
 //add salesmen reporting to Sales Manager
for (int i=0; i<5; i++)
   salesMgr .add(new Employee("Sales "+
       new Integer(i).toString(), 30000.0F
              +(float)(Math.random()-0.5)*10000));
advMgr.add(new Employee("Secy", 20000));

prodVP.add(prodMgr =
       new Employee("Prod Mgr", 40000));
prodVP.add(shipMgr =
       new Employee("Ship Mgr", 35000));
       //add manufacturing staff
for (int i = 0; i < 4; i++)
  prodMgr.add( new Employee("Manuf "+
       new Integer(i).toString(), 25000.0F
              +(float)(Math.random()-0.5)*5000));
 //add shipping clerks
```

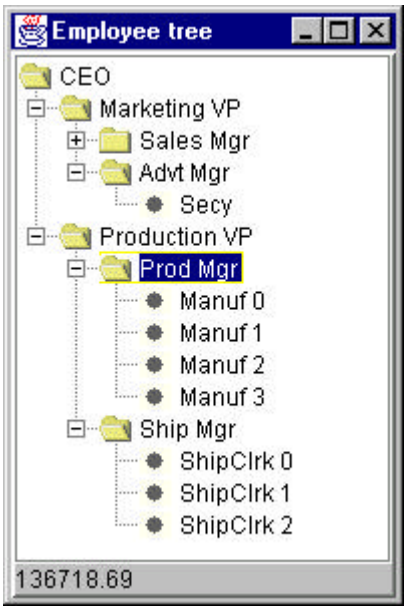
```
      for (int i = 0; i < 3; i++)
        shipMgr.add( new Employee("ShipClrk "+
              new Integer(i).toString(), 20000.0F
                    +(float)(Math.random()-0.5)*5000));
```

Once we have constructed this Composite structure, we can load a visual JTree list by starting at the top node and calling the *addNode()* method recursively until all the leaves in each node are accessed:

```
private void addNodes(DefaultMutableTreeNode pnode,
      Employee emp) {
 DefaultMutableTreeNode node;

 Enumeration e = emp.elements();
 while(e.hasMoreElements())
  {
  Employee newEmp = (Employee)e.nextElement();
  node = new DefaultMutableTreeNode(newEmp.getName());
  pnode.add(node);
  addNodes(node, newEmp);
  }
 }
```

The final program display is shown below:



In this implementation, the cost (sum of salaries) is shown in the bottom bar for any employee you click on. This simple computation calls the *getChild()* method recursively to obtain all the subordinates of that employee.

```
public void valueChanged(TreeSelectionEvent evt)
{
//called when employee is selected in tree llist
TreePath path = evt.getPath();
String selectedTerm =
             path.getLastPathComponent().toString();
//find that employee in the composite
Employee emp = boss.getChild(selectedTerm);
//display sum of salaries of subordinates(if any)
if(emp != null)
   cost.setText(new Float(emp.getSalaries()).toString());
}
```

Restrictions on Employee Classes

It could be that certain employees or job positions are designed so that they never should have subordinates. Assembly workers or salesmen may advance in the company by being named to a new position, but those holding these leaf positions will never have subordinates. In such a case, you may wish to design your Employee class so that you can specify that this is a permanent leaf position. One way to do this is to set a variable which is checked before it allows subordinates to be added. If the position is leaf position, the method returns *false* or throws an exception.

```
public void setLeaf(boolean b)    {
      isLeaf = b;    //if true, do not allow children
   }
//--------------------------------------
public boolean add(Employee e)    {
      if (! isLeaf)
         subordinates.addElement(e);
      return isLeaf;    //false if unsuccessful
   }
```

Consequences of the Composite Pattern

The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.

The Composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface. On the other hand, this has the disadvantage of making your system overly general. You might find it harder to restrict certain classes, where this would normally be desirable.

The composite is essentially a singly-linked tree, in which any of the objects may themselves be additional composites. Normally, these objects do not remember their parents and only know their children as an array, hash table or vector. However, it is perfectly possible for any composite element to remember its parent by including it as part of the constructor:

```
public Employee(Employee _parent, String _name,
                    float _salary)    {
      name = _name;              //save name
      salary = _salary;                //and salary
      parent = _parent;                //and parent
      subordinates = new Vector();
      isLeaf = false;            //allow children
   }
```

This simplifies searching for particular members and moving up the tree when needed.

Other Implementation Issues

Implementing the list in the parent. If there are a very large number of leaves in a composite but only a few nodes, then keeping an empty Vector object in each leaf has some space implications. An alternative approach is to declare all of the objects of type Member, which implements only *getName*() and *getValue*() methods. Then you derive a Node class from Member which implements the *add*, *remove* and *elements* methods. Now only objects that are Node classes can have an enumeration of members. You can check for this in the recursive loop instead of returning empty Vector enumerators.

```
   if(emp instanceof Node)     {
    Enumeration e = emp.elements();
    while(e.hasMoreElements())  {
       Employee newEmp = (Employee)e.nextElement();
             // etc.
   }
```

In most cases it is not clear that the space saving justifies this additional complexity.

Ordering components. In some programs, the order of the components may be important. If that order is somehow different from the order in which they were added to the parent, then the parent must do additional work to return them in the correct order. For example, you might sort the original Vector alphabetically and return the Enumerator to a new sorted vector.

Caching results. If you frequently ask for data which must be computed from a series of child components as we did here with salaries, it may be advantageous to cache these computed results in the parent. However, unless the computation is relatively intensive and you are quite certain that the underlying data have not changed, this may not be worth the effort.