Composite Pattern

What is it?

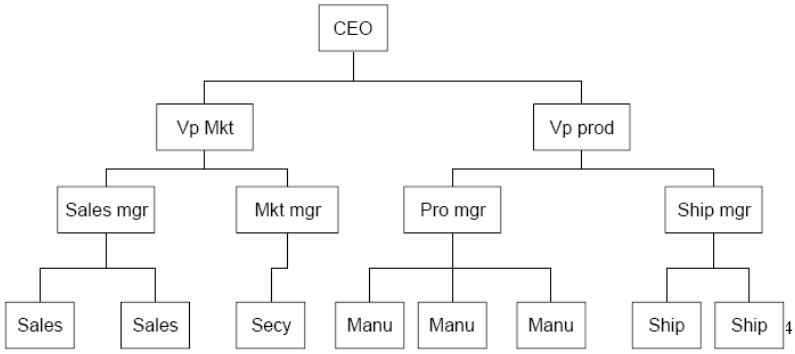
- The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program.
- Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.
- The Composite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

An Example Application

- 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 a (somewhat) complex organizational chart.

Salary

- (If the company is successful) each of these company members receives a salary.
- 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 subordinates. That is,
  - the cost of an individual employee is simply the salary.
  - the cost of an employee who heads a department is the salary plus those of all subordinates.



getSalaries()

- We would like a single interface that will produce the salary totals correctly whether the employee has subordinates or not.
  - public float getSalaries();

Employee Class

- Our Employee class stores the name and salary of each employee, and allows 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;
   }
```

add() and remove() Employees

- 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);
}
```


Enumeration

- 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();
}
```

- NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. You should consider using Iterator instead of Enumeration.

Finally, the getSalaries() method

```
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 Tree

```
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));
```

Building the Tree (2)

```
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));
```