Session will begin shortly

OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH
DUBLIN
TECHNOLOGICAL
UNIVERSITY DUBLIN

Your presenter is now broadcasting audio.

Please indicate if you can hear this audio by typing "**audio ok**" or "**audio not ok**" in the chat window.

In the event that you cannot hear audio, please check your cables and audio levels.

You may need to run the audio wizard. Select **Meeting**> **Audio Setup**

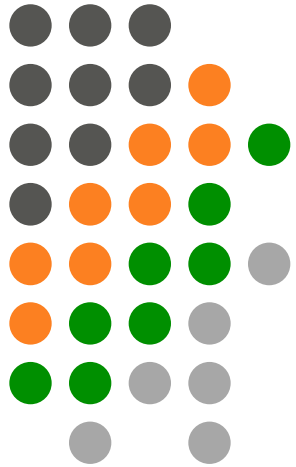
Recording in Progress

This session is now being recorded.

Object Orientation with Design Patterns

Lecture : Reflection and Composite

Dr Irene Murtagh



Ref. notes and code: Orla McMahon and Dr. Stephen Sheridan, TU Dublin.

Two Way Adapters

- The two-way adapter is a clever concept that allows an object to be viewed by different classes in different ways.

- For example last week we looked at using inheritance to adapt a class interface to conform to another interface. This is most easily carried out by the class adapter pattern since all the methods of the base class are automatically available to the derived class.

- However, this can only work if you do not override any of the base class methods with methods that behave differently.

Pluggable Adapters &Reflection

- A pluggable adapter is an adapter that adapts dynamically to one of several classes. Of course, the adapter can adapt only classes that it can recognise, and usually the adapter decides which class it is adapting to based on differing constructors or parameters.

- Java has yet another way for adapters to recognise which of several classes it must adapt to: reflection.

- You can use reflection to discover the names of public methods and their parameters for any class. For example, for any arbitrary object you can use the getClass method to obtain its class and the getMethods method to obtain an array of the method names.

Pluggable Adapters & Reflection

```
public class Person{
    private String name;
    private String address;

    Person(String initName, String initAddress) {
        name = initName;
        address = initAddress;
    }
    public String getName() {return name;}
    public String getAddress(){return address;}
    public void setName(String newName){
        name = newName;
    }
    public void setAddress(String newAddress){
        address = newAddress;
    }
    public void display(){
        System.out.println(name + " " + address);
    }
}
```

Pluggable Adapters & Reflection

```
import java.lang.reflect.*;
import com.sun.java.swing.*;

public class ShowMethods{
    public ShowMethods(){
        Person p = new Person("Alex","Dublin");
        Method[] methods = p.getClass().getMethods();
        for (int i = 0; i < methods.length; i++){
            System.out.println(methods[i].getName());
            Class cl[] = methods[i].getParameterTypes();
            for(int j=0; j < cl.length; j++)
                System.out.println(cl[j].toString());
        }
    }
    static public void main(String argv[]){
        new ShowMethods();
    }
}
```

Pluggable Adapters & Reflection

```
main
class [Ljava.lang.String;
getAddress
getName
setName
class java.lang.String
display
setAddress
class java.lang.String
showMethods
wait
long
int
wait
long
wait
equals
class java.lang.Object
toString
hashCode
getClass
notify
notifyAll
```

Pluggable Adapters &Reflection

- A “method dump” like the one shown on the previous slide can generate a very large list of methods. It is easier to use the name(if you know it) of the method you are looking for.

- From the method signature, you can then deduce the adaptations that you need to carryout.

Pluggable Adapters & Reflection

```
import java.lang.reflect.*;
import com.sun.java.swing.*;

public class ShowMethods{
   public ShowMethods(){
      Person p = new Person("Alex","Dublin");
      p.display();

      Method[] methods = p.getClass().getMethods();
      for (int i = 0; i < methods.length; i++){
         if (methods[i].getName() == "setName"){
            try{
               Object o = methods[i].invoke(p, "John");
            }
            catch (InvocationTargetException | IllegalAccessException e){
               System.out.println(e.getMessage());
            }
         }
      }
      p.display();
   }
   static public void main(String argv[]){
      new ShowMethods();
   }
}
```

Pluggable Adapters & Reflection

Adapters in Java

- In a broad sense, a number of adapters are already built into the Java language. In this case, the Java adapters serve to simplify an unnecessarily complicated event interface.

- One of the most commonly used of these Java adapters is the WindowAdapterclass.

- One inconvenience of Java is that windows do not close automatically when you click on the Close button or window Exit menu item. The general solution to this problem is to have your main window implement the **WindowListener interfaceand leave all of the Window** events empty except for windowClosing.

Adapters in Java

```
public class MainFrame extends Frame implements WindowListener{
    public MainFrame(){
        addWindowListener(this);
        setSize(300,300);
        show();
    }

    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
    // Ignore the rest of the events
    public void windowClosed(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

    public static void main(String[] args){
        MainFrame mf = new MainFrame();
    }
}
```

Adapters in Java

- As you can see this code is inefficient.
- Luckily Java provides a solution. TheWindowAdapter class is provided to simplify this procedure.
- This class contains empty implementations of all seven of the previous WindowEvents.
- All you need to do is override the windowClosing event and insert the appropriate exit code as can be seen in the following code listing.

Adapters in Java

```
public class WinApp extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
}
```

```
public class Closer extends Frame{
    private WinApp wapp;
    public Closer(){
        wapp = new WinApp();
        addWindowListener(wapp);
        setSize(200,200);
        show();
    }

    public static void main(String[] args){
        Closer cl = new Closer();
    }
}
```

Adapters in Java

- Adapters like these are common in Java when a simple class can be used to encapsulate a number of events.

- They include ComponentAdapter, **ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, and MouseMotionAdapter.**

Break

We will now take a short break.

Session will resume shortly

Composite Pattern

- Programmers often develop systems in which a component may be an individual object or may represent a collection of objects. The Composite pattern is designed to accommodate both cases.
- It is possible to build part-whole hierarchies or to construct data representations of trees using the Composite pattern.
- A composite is a collection of objects, any one of which may be either a composite or a primitive object. In tree terminology, some objects may be nodes with additional branches, and some may be leaves.
- A problem that develops using the Composite pattern is the ability to **distinguish between nodes and leaves**. Nodes have children and can have children added to them, while leaves do not have children.

Composite Pattern

- It is possible to create a separate interface for nodes and leaves, where a leaf node could have the following methods:

```
// Leaf interface
public String getName();
public String getValues();
```

- And a branch node could have the following methods:

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

Composite Pattern

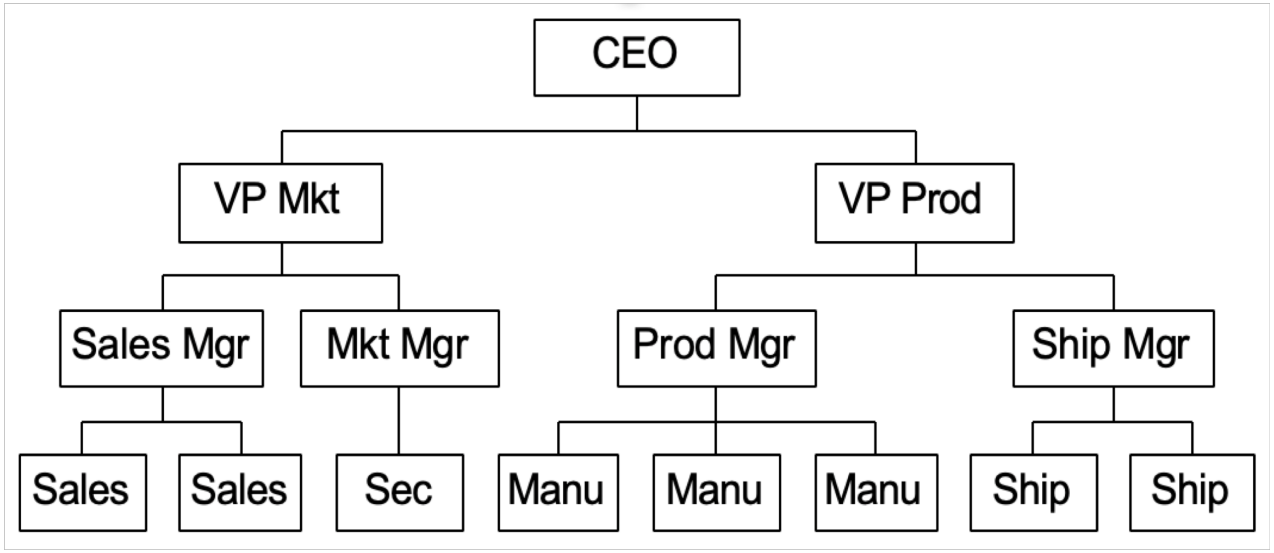
- This leaves us with the problem of deciding which elements will be which when we construct the **composite.**

- 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. Thus is we simply obtain the Enumeration from each element, we can quickly determine whether it has any children by checking the return values of hasMoreElements.

Composite Example

- Lets consider a small company that was started by one person. Then they hired a couple of people to handle the marketing and manufacturing. Soon each of those managers hired some assistants and the company expanded into the organisation shown below:



Composite Example

- If the company is successful, each of the employees receives a salary. At any point in time we could be asked for the salary span of any employee.

- We define the salary span cost as the salary of that person as well as the combined salaries of all of his or her subordinates.
  - The cost of an individual employee is that employee's salary
  - The cost of an employee that manages a department is that employee's salary plus the salaries of all subordinate employees.

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

Composite Example

- We could now imagine representing the company as a **Composite made up of node: managers and employees. We** could use a single class to represent all employees, but since each level might have different properties, defining at least two classes might be more useful: Employee and Boss.
  - **Employees are leaf nodes**
  - **Bosses are nodes that may have subordinate Employees**
- We'll start with an AbstractEmployeeclass and derive our **Concrete employee classes from it.**

Composite Example -AbstractEmployee

```
public abstract class AbstractEmployee {
    protected String name;
    protected float salary;
    protected Employee parent = null;
    protected boolean leaf = true;

    public abstract float getSalary();
    public abstract String getName();
    public abstract boolean add(Employee e)
        throws NoSuchElementException;
    public abstract void remove(Employee e)
        throws NoSuchElementException;
    public abstract Enumeration subordinates();
    public abstract Employee getChild(String s);
    public abstract float getSalaries();
    public boolean isLeaf() {
        return leaf;
    }
}
```

Composite Example -Employee

```
public class Employee extends AbstractEmployee {
    public Employee(String initName, float initSalary) {
        name = initName; salary = initSalary; leaf = true;
    }
    public Employee(Employee initParent, String initName, float initSalary) {
        name = initName; salary = initSalary;
        parent = initParent; leaf = true;
    }
    public float getSalary() {return salary;}
    public String getName(){return name;}
    public boolean add(Employee e) throws NoSuchElementException {
        throw new NoSuchElementException("No subordinates");
    }
    public void remove(Employee e) throws NoSuchElementException {
        throw new NoSuchElementException("No subordinates");
    }
    public Enumeration subordinates () {
        Vector v = new Vector();
        return v.elements ();
    }
    public Employee getChild(String s) throws NoSuchElementException {
        throw new NoSuchElementException("No children");
    }
    public float getSalaries() {return salary;}
    public Employee getParent() {return parent;}
}
```

Composite Example -Employee

- The Employee class must provide concrete implementations of **add, remove**, and **getChild**. Since the Employee is a leaf, all of these will return some sort of error indication. For example, the subordinates method could return null, but it would be better if it returned an empty Enumeration.

```
public Enumeration subordinates () {
    Vector v = new Vector();
    return v.elements ();
}
```

- This ensures that the interface behaves in the same way for an **Employee and a Boss.**

Composite Example -Employee

- The add and remove methods must generate errors, since members of the basic Employee class cannot have subordinates.

```
public boolean add(Employee e) throws NoSuchElementException {
    throw new NoSuchElementException("No subordinates");
}
public void remove(Employee e) throws NoSuchElementException {
    throw new NoSuchElementException("No subordinates");
}
```

Composite Example -Boss

- The Boss class is a subclass of Employee and allows us to store subordinate employees. We'll store them in a Vector called subordinates, which we return through an Enumeration.
- Thus, if a particular Boss has temporarily run out of Employees, the Enumeration will be empty.

Composite Example -Boss

```
public class Boss extends Employee {
    Vector employees;
    public Boss(String initName, long initSalary) {
        super(initName, initSalary);
        leaf = false;
        employees = new Vector();
    }
    public Boss(Employee initParent, String initName, long initSalary) {
        super(initParent, initName, initSalary);
        leaf = false;
        employees = new Vector();
    }
    public Boss(Employee emp) {
        //promotes an employee position to a Boss
        //and thus allows it to have employees
        super(emp.getName (), emp.getSalary());
        employees = new Vector();
        leaf = false;
    }
    ...
```

Composite Example -Boss

```
public boolean add(Employee e) throws NoSuchElementException {
    employees.add(e);
    return true;
}
public void remove(Employee e) throws NoSuchElementException {
    employees.removeElement(e);
}
public Enumeration subordinates () {
    return employees.elements ();
}
public float getSalaries() {
    float sum = salary;
    for (int i = 0; i < employees.size(); i++) {
        sum += ((Employee)employees.elementAt(i)).getSalaries();
    }
    return sum;
}
...
```

Composite Example -Boss

- If you want to get a list of employees of a given manager, you obtain an Enumeration of them directly from the subordinates Vector. Similarly, you can use this same Vector to return a sum of salaries for any employee and his or her subordinates.

```
public float getSalaries() {
    float sum = salary;
    for (int i = 0; i < employees.size(); i++) {
        sum += ((Employee)employees.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 who themselves have subordinates will be included.

Creating the Composite -Client

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

```
private void makeEmployees() {
    prez = new Boss("CEO", 200000);
    prez.add(marketVP = new Boss("Marketing VP", 100000));
    prez.add(prodVP = new Boss("Production VP", 100000));

    marketVP.add(salesMgr = new Boss("Sales Mgr", 50000));
    marketVP.add(advMgr = new Boss("Advt Mgr", 50000));
    //add salesmen reporting to sales manager
    for (int i=0; i<5; i++)
        salesMgr .add(new Employee("Sales "+ i, rand_sal(30000)));
    advMgr.add(new Employee("Secy", 20000));

    prodVP.add(prodMgr = new Boss("Prod Mgr", 40000));
    prodVP.add(shipMgr = new Boss("Ship Mgr", 35000));
    //add manufacturing staff
    for (int i = 0; i < 4; i++)
        prodMgr.add( new Employee("Manuf "+i, rand_sal(25000)));
    //add shipping clerks
    for (int i = 0; i < 3; i++)
        shipMgr.add( new Employee("ShipClrk "+i, rand_sal(20000)));
}
```

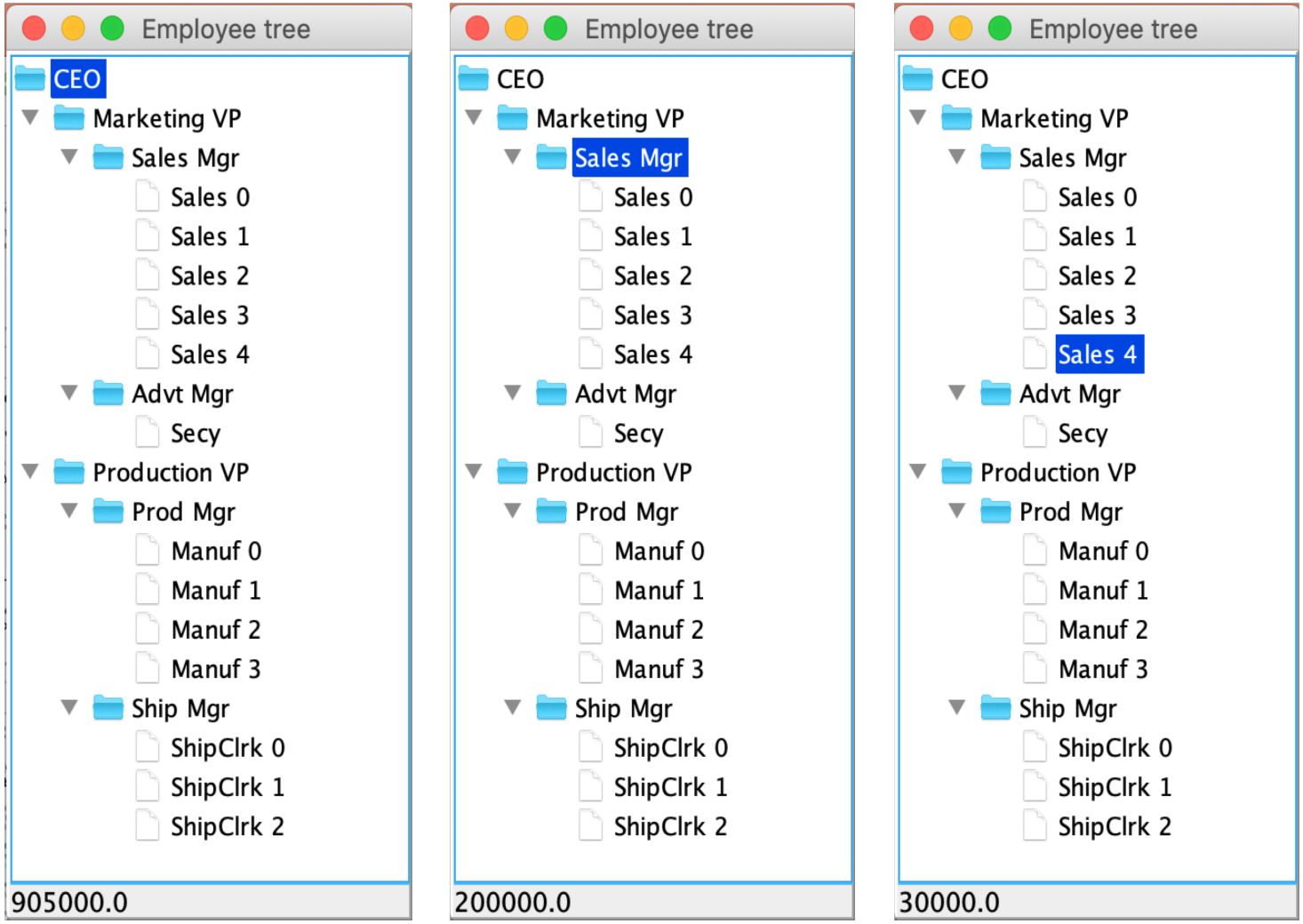
Creating the Composite -Client

- 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 of the leaves in each*** node are processed.

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

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

Creating the Composite -Client



Composite Pattern Consequences

- 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 sameway.