

Week 7: Design Pattern Examples

Patterns to be discussed today:

- Structural:
 - Adapter
 - Composite
- Behavioral:
 - Visitor

The slides are adapted from the book titled: *The Design Patterns (Java Companion)* by JAMES W. COOPER

1

The *Adapter* Pattern

- **Usage:** Is used to convert the programming interface of one class into that of another.
- There are two ways to do this: by inheritance, and by object composition:
 - Derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface (*class adapter*)
 - Include the original class inside the new one and create the methods to translate calls within the new class (*object adapter*)

2

Object Adapter Example

• Moving Data between Lists:

- Consider a Java program that allows you to enter names into a list, and then select some of those names to be transferred to another list. Our initial list consists of a class roster and the second list, those that will be doing advanced work



3

The *AWT Program*

- The program consists of a GUI creation constructor and an `actionListener` routine for the three buttons:

```
public void actionPerformed(ActionEvent e) {
    Button b = (Button)e.getSource();
    if(b == Add) addName();
    if(b == MoveRight) moveNameRight();
    if(b == MoveLeft) moveNameLeft();
}
```

- The button action routines are:

```
private void addName(){
    if (txt.getText().length() > 0) {
        leftList.add(txt.getText());
        txt.setText("");
    }
}
```

Continued on the next page ...

4

The AWT Program (continued)

```
private void moveNameRight() {
    String sel[] = leftList.getSelectedItems();
    if (sel != null) {
        rightList.add(sel[0]);
        leftList.remove(sel[0]);
    }
}

private void moveNameLeft() {
    String sel[] = rightList.getSelectedItems();
    if (sel != null){
        leftList.add(sel[0]);
        rightList.remove(sel[0]);
    }
}
```

5

Migrating to JFC

- **Problem:** The JFC JList class is significantly different from the AWT List class, because the former was designed to represent far more complex kinds of lists. Therefore, there are virtually no methods in common between the two classes.

awt List class	JFC JList class
add(String);	---
remove(String)	---
String[] getSelectedItems()	Object[] getSelectedValues()

- Since we have already written some code based on AWT's List interface, writing an adapter to make the JList class look like the List class seems a sensible solution to our problem.

6

Migrating to JFC (continued)

- We define the needed methods as an *interface* and then make sure that the class we create implements those methods:

```
public interface awtList {
    public void add(String s);
    public void remove(String s);
    public String[] getSelectedItems()
}
```

- In the object adapter approach, we create a class that contains a Jlist class but which implements the methods of the awtList interface above. Here we can use the outer container of our Jlist, i.e. the JScrollPane that encloses it.

7

Migrating to JFC (continued)

- So, the JawsList class will look like this:

```
public class JawsList extends JScrollPane implements awtList {
    private JList listWindow;
    private JListData listContents;

    public JawsList(int rows) {
        listContents = new JListData();
        listWindow = new JList(listContents);
        getViewport().add(listWindow);
    }

    public void add(String s) { listContents.addElement(s); }

    public void remove(String s){ listContents.removeElement(s); }

    public String[] getSelectedItems() {
        Object[] obj = listWindow.getSelectedValues();
        String[] s = new String[obj.length];
        for (int i =0; i<obj.length; i++)
            s[i] = obj[i].toString();
        return s;
    }
}
```

8

Migrating to JFC (continued)

- **Note:** the actual data handling takes place in the JListData class which is derived from the AbstractListModel

```
public class JListData extends AbstractListModel {
    private Vector data;

    public JListData() {
        data = new Vector();
    }

    public void addElement(String s) {
        data.addElement(s);
        fireIntervalAdded(this, data.size()-1, data.size());
    }

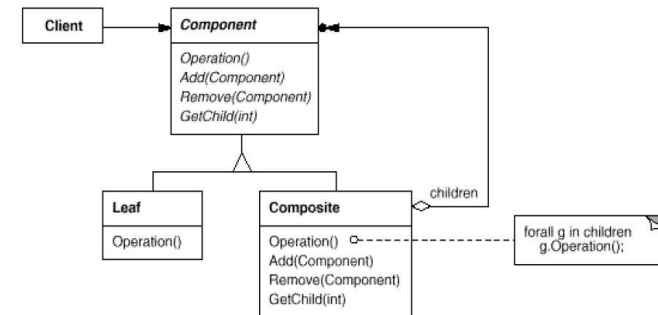
    public void removeElement(String s) {
        data.removeElement(s);
        fireIntervalRemoved(this, 0, data.size());
    }
}
```

- Applying the class adapter approach is very similar.

9

The *Composite* Pattern

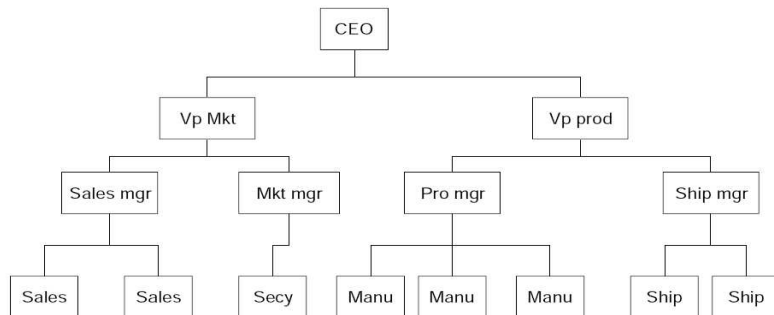
- **Usage:** Is used when a component may either be an individual object or a representative of a collection of objects
- **Structure:**



10

Example of the Composite Pattern

- **Company organizational charts:**



- Need a single interface for calculating costs whether an employee has subordinates or not. The cost of an individual employee is his salary; and the cost of an employee who heads a department is his salary plus those of all his/her subordinates.

11

Composite Example (continued)

- **The Employee class:**

```
public class Employee {
    String name;
    float salary;
    Vector subordinates;

    public Employee(String name, float salary) {
        this.name = name;
        this.salary = salary;
        this.subordinates = new Vector();
    }

    public float getSalary() { return salary; }
    public String getName() {return name; }
}
```

12

Composite Example (continued)

- Add / remove / fetch (immediate) subordinates:

```
public void add(Employee e) { subordinates.addElement(e); }
public void remove(Employee e) { subordinates.removeElement(e); }
public Enumeration elements() { return subordinates.elements(); }
```

- Computing the costs:

```
public float getSalaries() {
    float sum = salary;

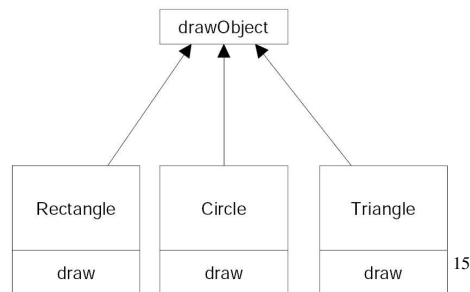
    for(int i = 0; i < subordinates.size(); i++)
        sum += ((Employee)subordinates.elementAt(i)).
            getSalaries();

    return sum;
}
```

13

Why do we need visitors after all?

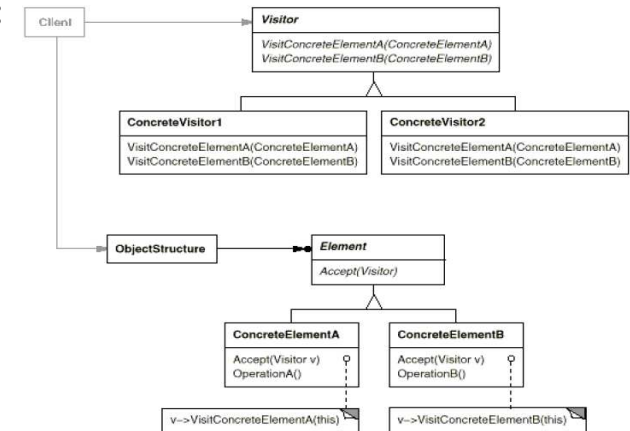
- **Question:** Is it not “unclean” to put operations that should be inside a class in another class?
- **Answer:** There are good reasons for doing.
- **Example:** Suppose the following classes have similar codes for drawing themselves. The drawing methods may be different, but they all use some utility functions that we might have to duplicate in each class. Also, several related functions are scattered among different classes as shown below:



15

The Visitor Pattern

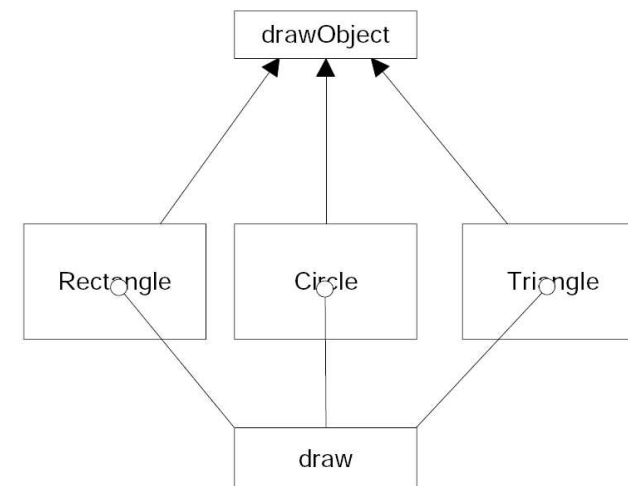
- **Usage:** Is used when we want an external class to act on data in other classes. This is useful if we have a small set of classes and we want to perform some operation that involves all or most of them.
- **Structure:**



14

How does a visitor help?

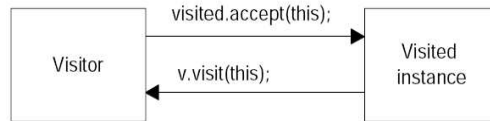
- We write a visitor class which contains all the related draw methods and have it visit each of the objects in succession:



16

What does visiting really mean?

- There is only one way that an outside class can gain access to another class, and that is by calling its public methods. In the visitor case, visiting each class means that you are calling a method already installed for this purpose, called accept. The accept method has one argument: the instance of the visitor, and in return, it calls the visit method of the visitor, passing itself as an argument.



```
public void accept(Visitor v) { v.visit(this); }
```

- In this way, the visitor object receives a reference to each of the instances, one by one, and can then call its public methods to obtain data, perform calculations, generate reports, or just draw the object on the screen.

17

Visitor Example

- Calculating the number of vacation days:
- The new Employee class:

```
public class Employee
{
    int sickDays, vacDays;
    float Salary;
    String Name;

    public Employee(String name, float salary,
                    int vacdays, int sickdays)
    {
        vacDays = vacdays;        sickDays = sickdays;
        Salary = salary;          Name = name;
    }
    public String getName() { return Name; }
    public int getSickdays() { return sickDays; }
    public int getVacDays() { return vacDays; }
    public float getSalary() { return Salary; }
    public void accept(Visitor v) { v.visit(this); }
}
```

18

Visitor Example (continued)

- The abstract visitor class:
- There is no indication as to what the Visitor does with each class
- The VacationVisitor class:

```
public abstract class Visitor {
    public abstract void visit(Employee emp);
}

public class VacationVisitor extends Visitor
{
    protected int total_days;
    public VacationVisitor() { total_days = 0; }
    //-----
    public void visit(Employee emp)
    {
        total_days += emp.getVacDays();
    }
    //-----
    public int getTotalDays()
    {
        return total_days;
    }
}
```

19

Visitor Example (continued)

- Visiting the employees:

```
VacationVisitor vac = new VacationVisitor();
for (int i = 0; i < employees.length; i++)
{
    employees[i].accept(vac);
}
System.out.println(vac.getTotalDays());
```

- **Note:** Here, we are only dealing with instances of the Employee class. The visitor becomes much more useful when there are a number of different classes with different interfaces and we want to encapsulate how we get data from them.

20