

# CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

## Visitor Pattern Apr. 24, 2007

1

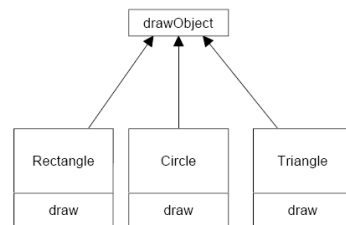
## Fine Print...

- The Visitor pattern turns the tables on our object-oriented model and creates an external class to act on data in other classes.
- This is useful if there are a fair number of instances of a small number of classes and you want to perform some operation that involves all or most of them.
- While at first it may seem “unclean” to put operations that should be inside a class in another class instead, there are (usually) good reasons for doing it.

2

## An Example

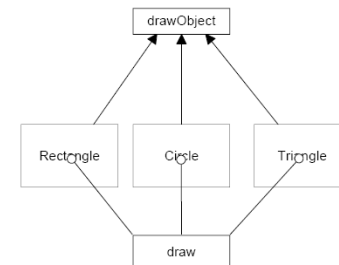
- Suppose each of a number of drawing object classes has similar code for drawing itself.
- The drawing methods may be different, but they probably all use underlying utility functions that we might have to duplicate in each class.
- Further, a set of closely related functions is scattered throughout a number of different classes.



3

## The Visitor Solution

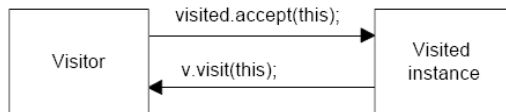
- Instead, we write a Visitor class which contains all the related *draw* methods and have it visit each of the objects in succession.



4

## But What Does “Visiting” Mean?

- There is only one way that an outside class can gain access to another class
  - 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.



5

## In Terms of Code...

- Every object that you want to visit must have the following method:

```
public void accept(Visitor v)
{
    v.visit(this);    //call visitor method
}
```

- 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.

6

## Use a Visitor when...

- You should consider using a Visitor pattern when you want to perform an operation on the data contained in a number of objects that have different interfaces.
- Visitors are also valuable if you have to perform a number of unrelated operations on these classes.

7

## An Example

- Remember the Employee problem we discussed in the Composite pattern.
  - Lets extend it.
- We have a simple Employee object which maintains a record of the employee’s name, salary, vacation taken and number of sick days taken.

8

## The 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); },
}
```

## Generating a Report

- Note that we have included the *accept* method in this class.
- Now let's suppose that we want to prepare a report of the number of vacation days that all employees have taken so far this year.
- We could just write some code in the client to sum the results of calls to each Employee's *getVacDays* function, or we could put this function into a Visitor.

10

## The Visitor Abstract Class

- Since Java is a strongly typed language, your base Visitor class needs to have a suitable abstract *visit* method for each kind of class in your program.
- In this simple example, we only have Employees, so our basic abstract Visitor class is just:

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

11

## A Concrete Visitor

- Notice that there is no indication what the Visitor does with each class in either the client classes or the abstract Visitor class.
- We can in fact write a whole lot of visitors that do different things to the classes in our program.
- The Visitor we are going to write first just sums the vacation data for all our employees.

12

## The Concrete Visitor Code

```
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;
    }
}
```

13

## The Main Program

- Now, all we have to do to compute the total vacation taken is to go through a list of the employees and visit each of them, and then ask the Visitor for the total.

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

14

## The Steps

1. Move through a loop of all the Employees.
2. The Visitor calls each Employee's *accept* method.
3. That instance of Employee calls the Visitor's *visit* method.
4. The Visitor fetches the vacation days and adds them into the total.
5. The main program prints out the total when the loop is complete.

15