THE VISIT

James W. Cooper

In “The Visit,” by Friedrich Durrenmatt, the billionaire heiress Clair Zachanasian revisits her home town with two old neutered boyfriends in tow, and offers the town millions if they will kill her ex-husband, who lives in the town. This is what I call a destructive visit. Rather than just observing interactions, she destroys private data and overruns existing classes with a steamroller.

By contrast, the Visitor pattern visits other classes and adds function to them nondestructively. It turns the tables on our object-oriented model and creates an external class to act on data in other classes. This is useful when you have a polymorphic operation that cannot reside in the class hierarchy for some reason.. For example, the operation wasn’t considered when the hierarchy was designed, or because it would clutter the interface of the classes unnecessarily.

Why are we Doing this?

While at first it may seem “unclean” to put operations that should be inside a class in another class instead, there are good reasons for doing it. 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. Instead, we write a Visitor class which contains all the related *draw* methods and have it visit each of the objects in succession

The question that most people who first read about this pattern ask is “what does visiting 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, as shown in Figure 1.

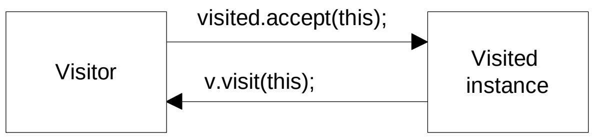


Figure 1- How the visit and accept methods interact.

Putting it in simple code terms, every object that you want to visit must have the following method:

```
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. Of course, if the class does not have an *accept* method, you can subclass it and add one.

When a Visit is called for

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. Visitors are a useful way to add function to class libraries or frameworks for which you either do not have the course or cannot change the source for other technical (or political) reasons. In these latter cases, you simply subclass the classes of the framework and add the *accept* method to each subclass.

On the other hand, as we will see below, Visitors are a good choice only when you do not expect many new classes to be added to your program.

Sample Code

Let's consider a simple Employee program that gathers data for accounting and payroll. We have a simple Employee object which maintains a record of the employee's name, salary, vacation taken and number of sick days taken. A simple version of this class is:

```
public class Employee {
    private int    sickDays, vacDays;
    private float  Salary;
    private 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);
    }
}
```

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.

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

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

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:

```
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;
    }
}
```

Visiting the Classes

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);
        }
        total.setText(new Integer(vac.getTotalDays()).toString());
```

Let’s reiterate what happens for each visit:

1. We 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.

Visiting Several Classes

The Visitor becomes more useful when there are a number of different classes with different interfaces and we want to encapsulate how we get data from these classes. Let’s extend our vacation days model by introducing a new Employee type called Boss. Let’s further suppose that at this company, Bosses are rewarded with bonus vacation days (instead of money). So the Boss class as a couple of extra methods to set and obtain the bonus vacation day information:

```
public class Boss extends Employee {
    private int bonusDays;

    public Boss(String name, float salary, int vacdays,
             int sickdays) {
        super(name, salary, vacdays, sickdays);
    }
    public void setBonusDays(int bonus) {
        bonusDays = bonus;
    }
    public int getBonusDays() {
        return bonusDays;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

When we add a class to our program, we have to add it to our Visitor as well, so that the abstract template for the Visitor is now:

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

This says that any concrete Visitor classes we write must provide polymorphic *visit* methods for both the Employee and the Boss class. In the case of our vacation day counter, we need to ask the Bosses for both regular and bonus days taken, so the visits are now different. We'll write a new bVacationVisitor class that takes account of this difference:

```
public class bVacationVisitor extends Visitor {
    int total_days;

    public bVacationVisitor() {
        total_days = 0;
    }
    //-----------------------------
    public int getTotalDays() {
        return total_days;
    }
//--------------------------------
    public void visit(Boss boss) {
        total_days += boss.getVacDays();
        total_days += boss.getBonusDays();
    }
    //-----------------------------
    public void visit(Employee emp) {
        total_days += emp.getVacDays();
    }
}
```

Note that while in this case Boss is derived from Employee, it need not be related at all as long as it has an *accept* method for the Visitor class. It is quite important, however, that you implement a *visit* method in the Visitor for *every class* you will be visiting and not count on inheriting this behavior, since the *visit* method from the parent class is an Employee rather than a Boss visit method. Likewise, each of your derived classes (Boss, Employee, etc. must have its own *accept* method rather than calling one in its parent class. This is illustrated in the class diagram in Figure 2.

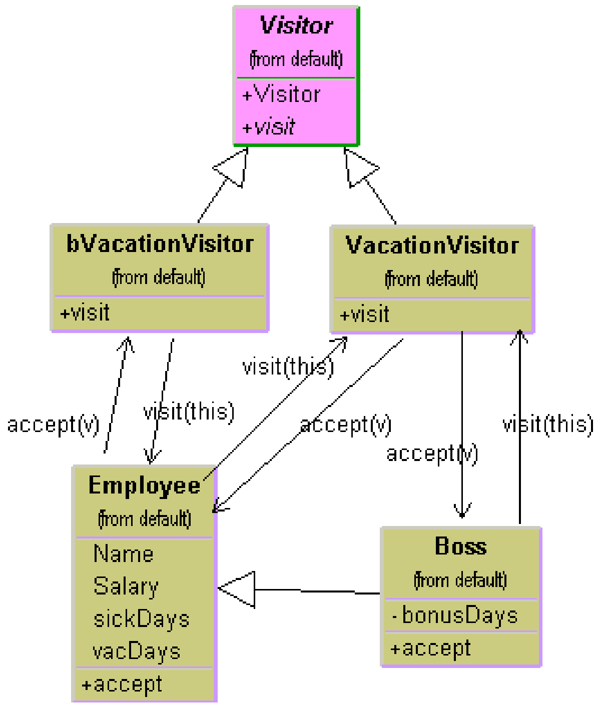


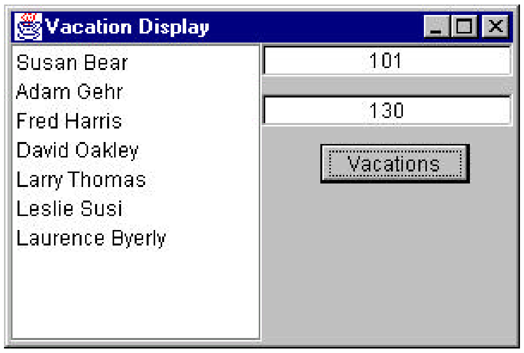
Figure 2 – The two visitor classes visiting the Boss and Employee classes.

Bosses are Employees, too

We show in Figure 3 a simple application that carries out both Employee visits and Boss visits on the collection of Employees and Bosses. The original VacationVisitor will just treat Bosses as Employees and get only their ordinary vacation data. The bVacationVisitor will get both.

```
    VacationVisitor vac = new VacationVisitor();
    bVacationVisitor bvac = new bVacationVisitor();
     for (int i = 0; i < employees.length; i++)      {
      employees[i].accept(vac);
      employees[i].accept(bvac);
     }
total.setText(new Integer(vac.getTotalDays()).toString());
btotal.setText(new Integer(bvac.getTotalDays()).toString());
```

The two lines of displayed data represent the two sums that are computed when the user clicks on the Vacations button.


Vacation Display

Susan Bear
Adam Gehr
Fred Harris
David Oakley
Larry Thomas
Leslie Susi
Laurence Byerly

101

130

Figure 3 – A simple application that performs the vacation visits described above.

Double Dispatching

No discussion on the Visitor pattern is complete without mentioning that you are really dispatching a method twice for the Visitor to work. The Visitor calls the polymorphic *accept* method of a given object, and the *accept* method calls the polymorphic *visit* method of the Visitor. It this bidirectional calling that allows you to add more operations on any class that has an *accept* method, since each new Visitor class we write can carry out whatever operations we might think of using the data available in these classes.

Consequences of the Visitor Pattern

The Visitor pattern is useful when you want to encapsulate fetching data from a number of instances of several classes. *Design Patterns* suggests that the Visitor can provide additional functionality to a class without changing it. We prefer to say that a Visitor can add functionality to a collection of classes and encapsulate the methods it uses.

The Visitor is not magic, however, and cannot obtain private data from classes: it is limited to the data available from public methods. This might force you to provide public methods that you would otherwise not have provided. However, it can obtain data from a disparate collection of unrelated classes and utilize it to present the results of a global calculation to the user program.

It is easy to add new operations to a program using Visitors, since the Visitor contains the code instead of each of the individual classes. Further, Visitors can gather related operations into a single class rather than forcing you to change or derive classes to add these operations. This can make the program simpler to write and maintain.

Visitors are less helpful during a program's growth stage, since each time you add new classes which must be visited, you have to add an abstract *visit* operation to the abstract Visitor class, and you must add an implementation for that class to each concrete Visitor you have written. Visitors can be powerful additions when the program reaches the point where many new classes are unlikely.

Visitors can be used very effectively in Composite systems and the boss-employee system we just illustrated could well be such a Composite.

James W. Cooper is a computer science researcher, and the author of 13 books, most recently Java Design Patterns: a Tutorial (Addison-Wesley, 2000) from which this article is adapted.