

Visitor

*“T is some visitor,” I muttered, “tapping at my chamber door;
Only this and nothing more.”*

-- Edgar Allen Poe, The Raven

Problem: *You need to add a new method to a hierarchy of classes, but the act of adding it will be painful or damaging to the design.*

This is a common problem. For example, suppose you have a hierarchy of `Modem` objects. The base class has the generic methods common to all modems. The derivatives represent the drivers for many different modem manufacturers and types. Suppose also that you have a requirement to add a new method, named `configureForUnix`, to the hierarchy. This method will configure the modem to work with the UNIX operating system. It will do something different in each modem derivative, because each different modem has its own particular idiosyncrasies for setting its configuration, and dealing with UNIX.

Unfortunately adding `configureForUnix` begs a terrible set of questions. What about Windows, what about MacOs, what about Linux? Must we really add a new method to the `Modem` hierarchy for every new operating system that we use? Clearly this is ugly. We’ll never be able to close the `Modem` interface. Every time a new operating system comes along we’ll have to change that interface and redeploy all the modem software.

The VISITOR family of design patterns.

The Visitor family allows new methods to be added to existing hierarchies without modifying the hierarchies.

The patterns in this family are:

- VISITOR

- ACYCLIC VISITOR
- DECORATOR
- EXTENSION OBJECT

VISITOR¹

Consider the Modem hierarchy in Figure 29-1. The Modem interface contains the generic methods that all modems can implement. There are three derivatives shown, one that drives a Hayes modem, another that drives a Zoom modem, and a third that drives the modem card produced by Ernie, one of our hardware engineers.

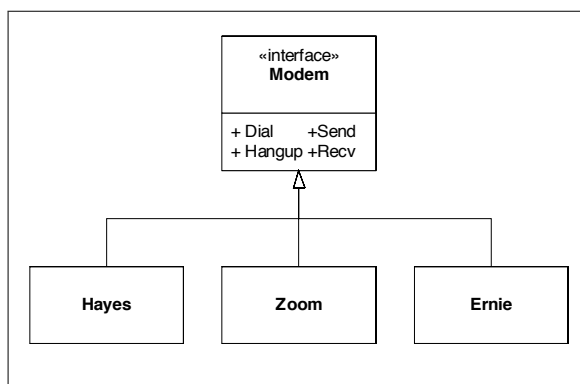


Figure 29-1
Modem Hierarchy

How can we configure these modems for unix without putting the `ConfigureForUnix` method in the `Modem` interface? We can use a technique called *dual dispatch*, which is the mechanism at the heart of the VISITOR pattern.

Figure 29-2 shows the VISITOR structure and Listing 29-1 through Listing 29-6 show the corresponding java code. Listing 29-7 shows the test code that both verifies that the VISITOR works and demonstrates how another programmer should use it..

Notice that there is a method in the visitor hierarchy for every derivative of the visited (`Modem`) hierarchy. This is a kind of 90° rotation — from derivatives to methods.

The test code shows that to configure a modem for unix a programmer creates an instance of the `UnixModemConfigurator` class and passes it to the `accept` function of the `Modem`. The appropriate `Modem` derivative will then call `visit(this)` on `ModemVisitor`, the base class of `UnixModemConfigurator`. If that derivative is a `Hayes`, then `visit(this)` will call `public void visit(Hayes)`. This will deploy to the `public`

1. [GOF95] p 331

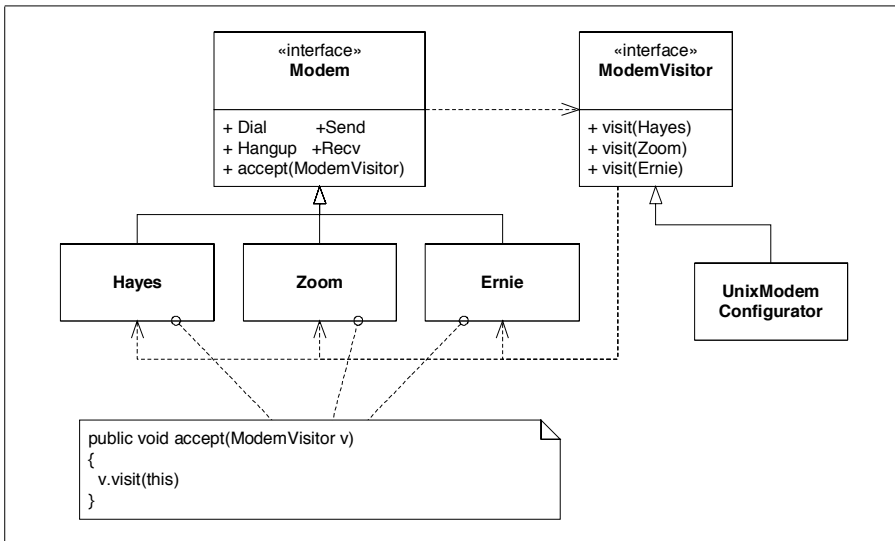


Figure 29-2
Visitor

Listing 29-1

Modem.java

```
public interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
    public void accept(ModemVisitor v);
}
```

Listing 29-2

ModemVisitor.java

```
public interface ModemVisitor
{
    public void visit(HayesModem modem);
    public void visit(ZoomModem modem);
    public void visit(ErnieModem modem);
}
```

Listing 29-3

HayesModem.java

```
public class HayesModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){ }
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}
}
```

Listing 29-3 (Continued)

HayesModem.java

```
String configurationString = null;
}
```

Listing 29-4

ZoomModem.java

```
public class ZoomModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){}
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}

    int configurationValue = 0;
}
```

Listing 29-5

ErnieModem.java

```
public class ErnieModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){}
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}

    String internalPattern = null;
}
```

Listing 29-6

UnixModemConfigurator.java

```
public class UnixModemConfigurator implements ModemVisitor
{
    public void visit(HayesModem m)
    {
        m.configurationString = "&s1=4&D=3";
    }

    public void visit(ZoomModem m)
    {
        m.configurationValue = 42;
    }

    public void visit(ErnieModem m)
    {
        m.internalPattern = "C is too slow";
    }
}
```

Listing 29-7

TestModemVisitor.java

```
import junit.framework.*;
public class TestModemVisitor extends TestCase
```

Listing 29-7 (Continued)

TestModemVisitor.java

```

{
    public TestModemVisitor(String name)
    {
        super(name);
    }

    private UnixModemConfigurator v;
    private HayesModem h;
    private ZoomModem z;
    private ErnieModem e;

    public void setUp()
    {
        v = new UnixModemConfigurator();
        h = new HayesModem();
        z = new ZoomModem();
        e = new ErnieModem();
    }

    public void testHayesForUnix()
    {
        h.accept(v);
        assertEquals("&s1=4&D=3", h.configurationString);
    }

    public void testZoomForUnix()
    {
        z.accept(v);
        assertEquals(42, z.configurationValue);
    }

    public void testErnieForUnix()
    {
        e.accept(v);
        assertEquals("C is too slow", e.internalPattern);
    }
}

```

void visit(Hayes) function in UnixModemConfigurator which then configures the Hayes modem for Unix.

Having built this structure, new operating system configuration functions can be added by adding new derivatives of ModemVisitor without altering the Modem hierarchy in any way. So the VISITOR pattern substitutes derivatives of ModemVisitor for methods in the Modem hierarchy.

This is called dual dispatch because it involves two polymorphic dispatches. The first is the accept function. This dispatch resolves the type of the object that accept is called upon. The second dispatch is the visit method called from the resolved accept method. The second dispatch resolves to the particular function to be executed.

Visitor is like a Matrix. The two dispatches of VISITOR form a matrix of functions. In our modem example, one axis of the matrix is the different types of modems. The other

axis is the different types of operating systems. Every cell in this matrix is filled in with a function that describes how to initialize the particular modem for the particular operating system.

Visitor is fast. It requires only two polymorphic dispatches; regardless of the breadth or depth of the visited hierarchy.

Acyclic Visitor.

Notice that the base class of the visited (`Modem`) hierarchy depends upon the base class of the visitor hierarchy (`ModemVisitor`). Notice also that the base class of the visitor hierarchy has a function for each derivative of the visited hierarchy. Thus there is a cycle of dependencies that ties all the visited derivatives (all the `Modems`) together. This makes it very difficult to compile the visitor structure incrementally, or to add new derivatives to the visited hierarchy.

The VISITOR works very well in programs where the hierarchy to be modified does not need new derivatives very often. If `Hayes`, `Zoom`, and `Ernie` were the only `Modem` derivatives that were likely to be needed; or if the incidence of new `Modem` derivatives was expected to be infrequent, then the VISITOR would be very appropriate.

On the other hand, if the visited hierarchy is highly volatile such that many new derivatives will need to be created, then the visitor base class (e.g. `ModemVisitor`) will have to be modified and recompiled along with all its derivatives every time a new derivatives is added to the visited hierarchy. In C++ the situation is even worse. The entire visited hierarchy must be recompiled and redeployed whenever any new derivative is added.

To solve these problems, a variation known as ACYCLIC VISITOR² can be used. (See Figure 29-3.) This variation breaks the dependency cycle by making the visitor base class (`ModemVisitor`) degenerate³. The lack of any methods in this class means that it does not depend upon the derivatives of the visited hierarchy.

The visitor derivatives also derive from visitor interfaces. There is one visitor interface for each derivative of the visited hierarchy. This is a 180° rotation from derivatives to interfaces. The `accept` functions in the visited derivatives cast the visitor base class⁴ to the appropriate visitor interface. If the cast succeeds, the method invokes the appropriate visit function. Listing 29-8 through Listing 29-16 show the code.

This breaks the dependency cycle and makes it easier to add visited derivatives and do incremental compilations. Unfortunately it also makes the solution much more complex. Worse still, the timing of the cast can depend upon the width and breadth of the visited hierarchy, and is therefore hard to characterize.

2. [PLOPD3], p93

3. A degenerate class is one that has no methods at all. In C++ it would have a pure virtual destructor. In java such classes are called "Marker Interfaces".

4. In C++ we use `dynamic_cast`

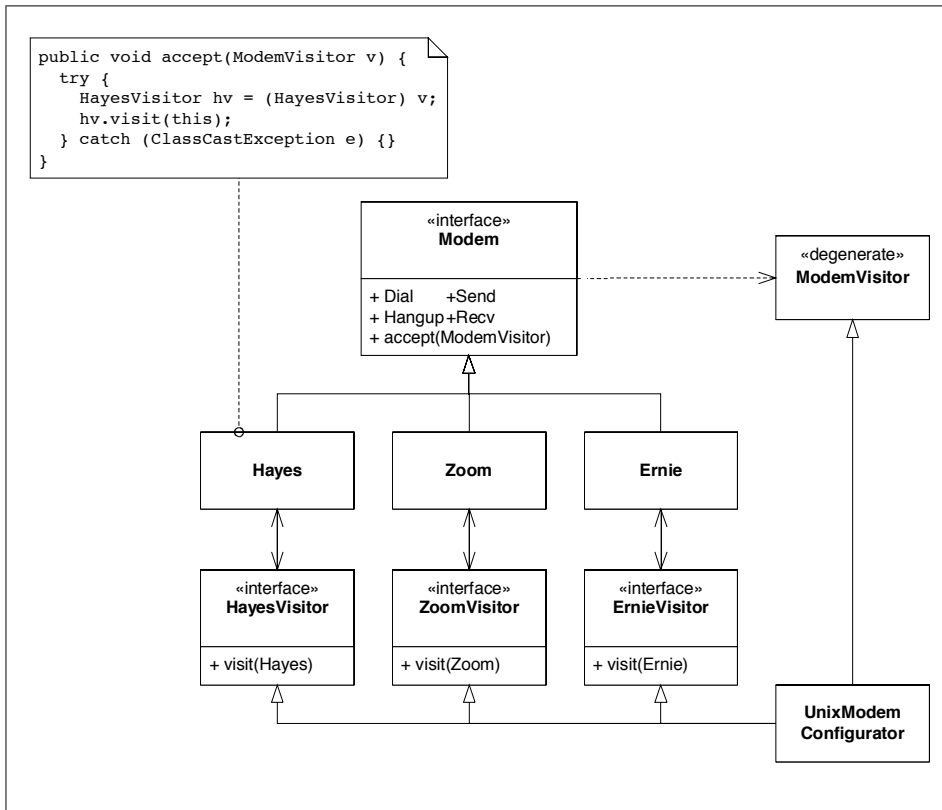


Figure 29-3
Acyclic Visitor

Listing 29-8

Modem.java

```

public interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
    public void accept(ModemVisitor v);
}

```

Listing 29-9

ModemVisitor.java

```

public interface ModemVisitor
{
}

```

Listing 29-10

```
ErnieModemVisitor.java
public interface ErnieModemVisitor
{
    public void visit(ErnieModem m);
}
```

Listing 29-11

```
HayesModemVisitor.java
public interface HayesModemVisitor
{
    public void visit(HayesModem m);
}
```

Listing 29-12

```
ZoomModemVisitor.java
public interface ZoomModemVisitor
{
    public void visit(ZoomModem m);
}
```

Listing 29-13

```
ErnieModem.java
public class ErnieModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){ }
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v)
    {
        try
        {
            ErnieModemVisitor ev = (ErnieModemVisitor)v;
            ev.visit(this);
        }
        catch (ClassCastException e)
        {
        }
    }

    String internalPattern = null;
}
```

Listing 29-14

```
HayesModem.java
public class HayesModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){ }
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v)
    {
        try
```


Listing 29-14 (Continued)

HayesModem.java

```

        {
            HayesModemVisitor hv = (HayesModemVisitor)v;
            hv.visit(this);
        }
        catch (ClassCastException e)
        {
        }
    }

    String configurationString = null;
}

```

Listing 29-15

ZoomModem.java

```

public class ZoomModem implements Modem
{
    public void dial(String pno){}
    public void hangup(){}
    public void send(char c){}
    public char recv() {return 0;}
    public void accept(ModemVisitor v)
    {
        try
        {
            ZoomModemVisitor zv = (ZoomModemVisitor)v;
            zv.visit(this);
        }
        catch(ClassCastException e)
        {
        }
    }

    int configurationValue = 0;
}

```

Listing 29-16

TestModemVisitor.java

```

import junit.framework.*;
public class TestModemVisitor extends TestCase
{
    public TestModemVisitor(String name)
    {
        super(name);
    }

    private UnixModemConfigurator v;
    private HayesModem h;
    private ZoomModem z;
    private ErnieModem e;

    public void setUp()
    {
        v = new UnixModemConfigurator();
        h = new HayesModem();
    }
}

```

Listing 29-16 (Continued)

TestModemVisitor.java

```

    z = new ZoomModem();
    e = new ErnieModem();
}

public void testHayesForUnix()
{
    h.accept(v);
    assertEquals("&s1=4&D=3", h.configurationString);
}

public void testZoomForUnix()
{
    z.accept(v);
    assertEquals(42, z.configurationValue);
}

public void testErnieForUnix()
{
    e.accept(v);
    assertEquals("C is too slow", e.internalPattern);
}
}

```

For hard real time systems the large and unpredictable execution time of the cast may make the ACYCLIC VISITOR inappropriate. For other systems, the complexity of the pattern may disqualify it. But for those systems in which the visited hierarchy is volatile, and incremental compilation is important, then this pattern can be a good option.

Acyclic Visitor is like a Sparse Matrix. Back on page 529 I described how the VISITOR pattern created a matrix of functions, with the visited type on one axis, and the function to be performed on the other. Acyclic Visitor creates the a *sparse* matrix. The visitor classes do not have to implement visit functions for each visited derivative. For example, if Ernie modems cannot be configured for Unix, then the UnixModemConfigurator will not implement the ErnieVisitor interface.

Using VISITOR in Report Generators

A very common use of the VISITOR pattern is to walk large data structures and generate reports. The value of the VISITOR in this case is that the data structure objects do not have to have any report generation code. New reports can be added by adding new VISITORS, rather than by changing the code in the data structures. This means that reports can be placed in separate components, and individually deployed only to those customers that need them.

Consider a simple data structure that represents a bill of materials (See Figure 29-4). There is an unlimited number of reports that we could generate from this data structure. We could generate a report of the total cost of an assembly. Or we could generate a report that listed all the piece-parts in an assembly.

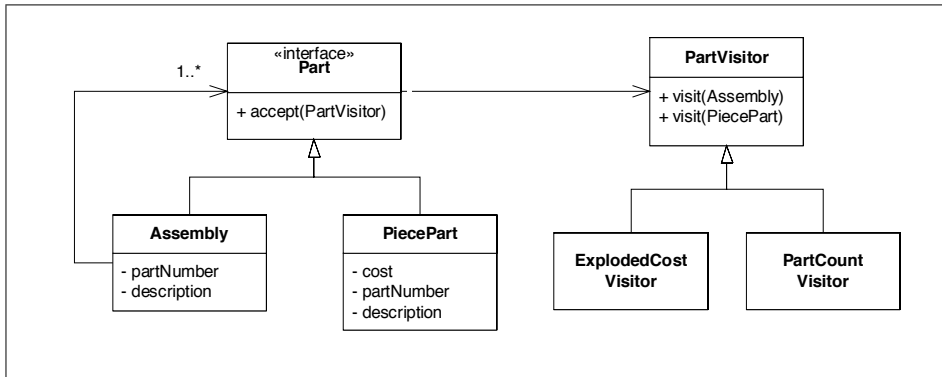


Figure 29-4
Bill Of Materials Report Generator Structure

Each of these reports could be generated by methods in the `Part` class. For example, `getExplodedCost`, and `getPieceCount` could be added to the `Part` class. These methods would be implemented in each derivative of `Part` such that the appropriate reporting was accomplished. Unfortunately that would mean that every new report that the customer wanted would force us to change the `Part` hierarchy.

The Single Responsibility Principle (SRP) told us that we want to separate code that changes for different reasons. The `Part` hierarchy may change as new kinds of parts are needed. However, it should not change because new kinds of reports are needed. Thus we'd like to separate the reports from the `Part` hierarchy. The VISITOR structure shown in Figure 29-4 shows how this can be accomplished.

Each new report can be written as a new visitor. We write the `accept` function of `Assembly` to visit the visitor and also call `accept` on all the contained `Part` instances. Thus the entire tree is traversed. For each node in the tree the appropriate `visit` function is called on the report. The report accumulates the necessary statistics. The report can then be queried for the interesting data and presented to the user.

This structure allows us to create an unlimited number of reports without affecting the part hierarchy at all. Moreover, each report can be compiled and distributed independently of all the others. This is nice. Listing 29-17 through Listing 29-23 show how this looks in java.

Listing 29-17

`Part.java`

```

public interface Part
{
    public String getPartNumber();
    public String getDescription();
    public void accept(PartVisitor v);
}
  
```

Listing 29-18

Assembly.java

```

import java.util.*;

public class Assembly implements Part
{
    public Assembly(String partNumber, String description)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
    }

    public void accept(PartVisitor v)
    {
        v.visit(this);
        Iterator i = getParts();
        while (i.hasNext())
        {
            Part p = (Part)i.next();
            p.accept(v);
        }
    }

    public void add(Part part)
    {
        itsParts.add(part);
    }

    public Iterator getParts()
    {
        return itsParts.iterator();
    }

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescription;
    }

    private List itsParts = new LinkedList();
    private String itsPartNumber;
    private String itsDescription;
}

```

Listing 29-19

PiecePart.java

```

public class PiecePart implements Part
{
    public PiecePart(String partNumber,
                    String description,
                    double cost)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
    }
}

```

Listing 29-19 (Continued)

PiecePart.java

```

        itsCost = cost;
    }

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

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescription;
    }

    public double getCost()
    {
        return itsCost;
    }

    private String itsPartNumber;
    private String itsDescription;
    private double itsCost;
}

```

Listing 29-20

PartVisitor.java

```

public interface PartVisitor
{
    public void visit(PiecePart pp);
    public void visit(Assembly a);
}

```

Listing 29-21

ExplodedCostVisitor.java

```

public class ExplodedCostVisitor implements PartVisitor
{
    private double cost = 0;
    public double cost() {return cost;}

    public void visit(PiecePart p)
    {cost += p.getCost();}

    public void visit(Assembly a) {}
}

```

Listing 29-22

PartCountVisitor.java

```

import java.util.*;

```

Listing 29-22 (Continued)

PartCountVisitor.java

```

public class PartCountVisitor implements PartVisitor
{
    public void visit(PiecePart p)
    {
        itsPieceCount++;
        String partNumber = p.getPartNumber();
        int partNumberCount = 0;
        if (itsPieceMap.containsKey(partNumber))
        {
            Integer carrier = (Integer)itsPieceMap.get(partNumber);
            partNumberCount = carrier.intValue();
        }
        partNumberCount++;
        itsPieceMap.put(partNumber, new Integer(partNumberCount));
    }

    public void visit(Assembly a)
    {
    }

    public int getPieceCount() {return itsPieceCount;}
    public int getPartNumberCount() {return itsPieceMap.size();}
    public int getCountForPart(String partNumber)
    {
        int partNumberCount = 0;
        if (itsPieceMap.containsKey(partNumber))
        {
            Integer carrier = (Integer)itsPieceMap.get(partNumber);
            partNumberCount = carrier.intValue();
        }
        return partNumberCount;
    }

    private int itsPieceCount = 0;
    private HashMap itsPieceMap = new HashMap();
}

```

Listing 29-23

TestBOMReport.java

```

import junit.framework.*;
import java.util.*;

public class TestBOMReport extends TestCase
{
    public TestBOMReport(String name)
    {
        super(name);
    }

    private PiecePart p1;
    private PiecePart p2;
    private Assembly a;

    public void setUp()

```

Listing 29-23 (Continued)

TestBOMReport.java

```
{
    p1 = new PiecePart("997624", "MyPart", 3.20);
    p2 = new PiecePart("7734", "Hell", 666);
    a = new Assembly("5879", "MyAssembly");
}

public void testCreatePart()
{
    assertEquals("997624", p1.getPartNumber());
    assertEquals("MyPart", p1.getDescription());
    assertEquals(3.20, p1.getCost(), .01);
}

public void testCreateAssembly()
{
    assertEquals("5879", a.getPartNumber());
    assertEquals("MyAssembly", a.getDescription());
}

public void testAssembly()
{
    a.add(p1);
    a.add(p2);
    Iterator i = a.getParts();
    PiecePart p = (PiecePart)i.next();
    assertEquals(p, p1);
    p = (PiecePart)i.next();
    assertEquals(p, p2);
    assert(i.hasNext() == false);
}

public void testAssemblyOfAssemblies()
{
    Assembly subAssembly = new Assembly("1324", "SubAssembly");
    subAssembly.add(p1);
    a.add(subAssembly);

    Iterator i = a.getParts();
    assertEquals(subAssembly, i.next());
}

private boolean p1Found = false;
private boolean p2Found = false;
private boolean aFound = false;

public void testVisitorCoverage()
{
    a.add(p1);
    a.add(p2);
    a.accept(new PartVisitor(){
        public void visit(PiecePart p)
        {
            if (p == p1)
                p1Found = true;
            else if (p == p2)
```

Listing 29-23 (Continued)

TestBOMReport.java

```

        p2Found = true;
    }

    public void visit(Assembly assy)
    {
        if (assy == a)
            aFound = true;
    }
    });
    assert(p1Found);
    assert(p2Found);
    assert(aFound);
}

private Assembly cellphone;

void setUpReportDatabase()
{
    cellphone = new Assembly("CP-7734", "Cell Phone");
    PiecePart display = new PiecePart("DS-1428",
        "LCD Display",
        14.37);
    PiecePart speaker = new PiecePart("SP-92",
        "Speaker",
        3.50);
    PiecePart microphone = new PiecePart("MC-28",
        "Microphone",
        5.30);
    PiecePart cellRadio = new PiecePart("CR-56",
        "Cell Radio",
        30);
    PiecePart frontCover = new PiecePart("FC-77",
        "Front Cover",
        1.4);
    PiecePart backCover = new PiecePart("RC-77",
        "RearCover",
        1.2);
    Assembly keypad = new Assembly("KP-62", "Keypad");
    Assembly button = new Assembly("B52", "Button");
    PiecePart buttonCover = new PiecePart("CV-15",
        "Cover",
        .5);
    PiecePart buttonContact = new PiecePart("CN-2",
        "Contact",
        1.2);

    button.add(buttonCover);
    button.add(buttonContact);
    for (int i=0; i<15; i++)
        keypad.add(button);
    cellphone.add(display);
    cellphone.add(speaker);
    cellphone.add(microphone);
    cellphone.add(cellRadio);
    cellphone.add(frontCover);
    cellphone.add(backCover);
    cellphone.add(keypad);
}

```


Listing 29-23 (Continued)

TestBOMReport.java

```
public void testExplodedCost()
{
    setUpReportDatabase();
    ExplodedCostVisitor v = new ExplodedCostVisitor();
    cellphone.accept(v);
    assertEquals(81.27, v.cost(), .001);
}

public void testPartCount()
{
    setUpReportDatabase();
    PartCountVisitor v = new PartCountVisitor();
    cellphone.accept(v);
    assertEquals(36, v.getPieceCount());
    assertEquals(8, v.getPartNumberCount());
    assertEquals("DS-1428", 1, v.getCountForPart("DS-1428"));
    assertEquals("SP-92", 1, v.getCountForPart("SP-92"));
    assertEquals("MC-28", 1, v.getCountForPart("MC-28"));
    assertEquals("CR-56", 1, v.getCountForPart("CR-56"));
    assertEquals("RC-77", 1, v.getCountForPart("RC-77"));
    assertEquals("CV-15", 15, v.getCountForPart("CV-15"));
    assertEquals("CN-2", 15, v.getCountForPart("CN-2"));
    assertEquals("Bob", 0, v.getCountForPart("Bob"));
}
}
```

Other uses of Visitor

In general, the `Visitor` pattern can be used in any application where there is a data structure that needs to be interpreted many different ways. Compilers often create intermediate data structures that represent syntactically correct source code. These data structures are then used to generate compiled code. One could imagine visitors for each different processor and/or optimization scheme. One could also imagine a visitor that converted the intermediate data structure into a cross reference listing, or even a UML diagram.

Many applications make use of configuration data structures. One could imagine the different subsystems of the application initializing themselves from the configuration data by walking it with their own particular visitors.

In every case where visitors are used, the data structure being used is independent of the uses to which it is being put. New visitors can be created, existing visitors can be changed, and all can be redeployed to installed sites without the recompilation, or redeployment of the existing data structures. This is the power of the `VISITOR`.

Bibliography

[GOF95]: : *Design Patterns*, Gamma, et. al., Addison Wesley, 1995

[PLOPD3]: *Pattern Languages of Program Design 3*, Robert C. Martin, et. al., Addison Wesley, 1998.