The Visitor Pattern

consider a conventional class hierarchy:

- an abstract base class Animal
- various subclasses (Dog, Cat, ...)
  - each with attributes:
    - String name;
    - int weight;
    - Diet preferredDiet;
  - each with methods:
    - int getWeight()
    - String getName();
    - Diet getDiet();
  - and more methods to be added (graphics, sound, behaviors, growth rates, ...)
consider how to provide a given capability (method) to all the subclasses within this hierarchy

let’s revisit how to model the animal’s weight

instead of:

- providing an instance variable within the class hierarchy (perhaps at the base class):
  - private int weight;
- and setters and getters within the hierarchy:
  - public int getWeight() ...
  - protected void setWeight(int w) ...

suppose we do not put any of the weight code within the Animal hierarchy

and instead create a ‘weight expert’ class that is responsible for providing the animal’s weight.
The Visitor Pattern

conventionally:

- instance setters/getters delegated to the subclasses
- that makes sense, because:
  - each kind of Animal has a specific weight
  - as you add subclasses of Animal, straightforward to provide weight methods appropriate to that animal
  - getWeight could be abstract in Animal, concrete in each subclass of Animal (Dog, Cat, etc.)

but let this illustrate a case where some functionality can be added outside the hierarchy...

- adding graphics, behavior, sound, debugging
- without modifying the hierarchy.
the mechanism

class C {
    public void acceptVisitor(Visitor v) {
        v.visit(this);
    }
}

class Visitor {
    public void visit(C c) {
        // method appropriate to C
    }
}

usage:
C c = new C();
Visitor v = new Visitor();
c.acceptVisitor(v);  // evokes method for C within v
instead of:

class Dog {
    public int getWeight() { return 25; }
}

Dog d = new Dog();
System.out.println("weight= " + d.getWeight());

do this:
create a "visitor" class that provides the information about the dog’s weight:

first, create a base class for all such Visitors

abstract class Visitor {
    abstract public void visit(Dog d);
}

class WeightV extends Visitor {
    private int weight;

    public void visit(Dog d) { weight = 25; }
    public int getWeight() { return weight; }
}
and tell the Dog to accept any such "visitor":

class Dog {
    void acceptVisitor(Visitor v) { v.visit(this); }
}

usage:

Dog d = new Dog();
WeightV v = new WeightV();
d.acceptVisitor(v);
System.err.println("dog's weight = " + v.getWeight());
class WeightV {
    private int weight;
    public void visit(Dog d) { weight = 25; }
    public int getWeight() { return weight; }
}

class Dog {
    void accept(Visitor v) { v.visit(this); }
}

usage:

Dog     d = new Dog();
WeightV v = new WeightV();

d.acceptVisitor(v);
System.err.println("dog's weight = " + v.getWeight());
public HelloVisitorV1() {
    Dog d = new Dog();
    WeightV v = new WeightV();

    d.acceptVisitor(v);
    System.err.println("dog's weight = " + v.getWeight());
}

public static void main(String[] args) { new HelloVisitorV1(); }

abstract private class Visitor {
    abstract public void visit(Dog d);
}

private class WeightV extends Visitor {
    private int weight;

    public void visit(Dog d) { weight = 25; }
    public int getWeight() { return weight; }
}

private class Dog { void acceptVisitor(Visitor v) { v.visit(this); } }
private class Dog {
    void acceptVisitor(Visitor v) {
        v.visit(this);
    }
}

private class Cat {
    void acceptVisitor(Visitor v) {
        v.visit(this);
    }
}

abstract private class Visitor {
    abstract public void visit(Dog d);
    abstract public void visit(Cat c);
}

private class WeightV extends Visitor {
    private int weight;

    public void visit(Dog d) {
        weight = 25;
    }

    public void visit(Cat c) {
        weight = 10;
    }

    public int getWeight() {
        return weight;
    }
}

Dog     d = new Dog();
Cat     c = new Cat();
WeightV v = new WeightV();

d.acceptVisitor(v);
System.err.println("dog’s weight = " + v.getWeight());
c.acceptVisitor(v);
System.err.println("cat’s weight = " + v.getWeight());
abstract private class Visitor {
    abstract public void visit(Dog d);
    abstract public void visit(Cat c);
}

private class WeightV extends Visitor {
    private int weight;

    public void visit(Dog d) {
        int volume = d.getLength()*d.getWidth()*d.getHeight();
        weight = DOG_DENSITY*volume;
    }

    public void visit(Cat c) { weight = 10; }

    public int getWeight() { return weight; }
}
abstract private class Visitor {
    abstract public void visit(Dog d);
    abstract public void visit(Cat c);
}

private class SpeedV extends Visitor {
    private int speed;

    public void visit(Dog d) {
        speed = DOG_SPEED_COEFFICIENT*d.getHipHeight();
    }

    public void visit(Cat c) { ... code specific to cat speed computation }

    public int getSpeed() { return speed;}
}
The Basic Visitor Pattern

Hierarchy of Visitors
- each extends an abstract base class Visitor
- each subclass provides a polymorphic visit method, one for each possible ‘visitee’ (argument)

Hierarchy of ‘Visitees’
- each has acceptVisitor(Visitor v) { v.visit(this); } 
- (later we will improve the mechanism for declaring a class a potential recipient of a given Visitor)

Usage
- have ‘visitee’ accept Visitor
- Visitor visits with the ‘visitee’ (calls methods in) 
- either visitor causes side effects or is queried
possible benefits

- providing temporary debugging visitors
  - which toString their visitees
  - exercise, interrogate, etc. their visitees
- add entirely new functionality to a class hierarchy
  - adding sound, graphics, etc.
- mutating visitees
  - hunger visitor
  - tooth fairy
- maintenance of a data structure
  - passing a visitor into a list or array, etc.
    - culling out elements (‘bring out the dead’)
  - delegation of traversal to the data structure
    - giving the visitor a guided tour