VISITOR DESIGN PATTERN

1) Visitor lets us define a new operation without changing the classes of the elements on which it operates.

2) For recovering lost type information, Visitor Design pattern is the classical approach.

3) We want to do the right thing based on the type of two objects.

4) Double Dispatch.

PROBLEM

1) A node object have many distinct and unrelated operations that works on them.

2) We want to avoid "polluting" the node classes with these operations.

3) Before performing the desired operation, we don't want to have to query the type of each node and cast the pointers to the correct type.

Detailed Discussion

1) Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects.

2) Using Visitor Design Pattern approach, encourages designing lightweight Element classes -because processing functionality is removed from their list of responsibilities.

3) New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

About Double Dispatch

Visitor implements "double dispatch", in which the operation executed depends on: the name of the request, and the type of two receiver(the type of the visitor and the type of the element it visits).

Implementation of Visitor Design Pattern

Implementation proceeds as follows:

1) Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy.

2) Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.

3) Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy.

4) The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

5) Add a single pure virtual accept() method to the base class of the Element hierarchy.

6)accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

7) Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.

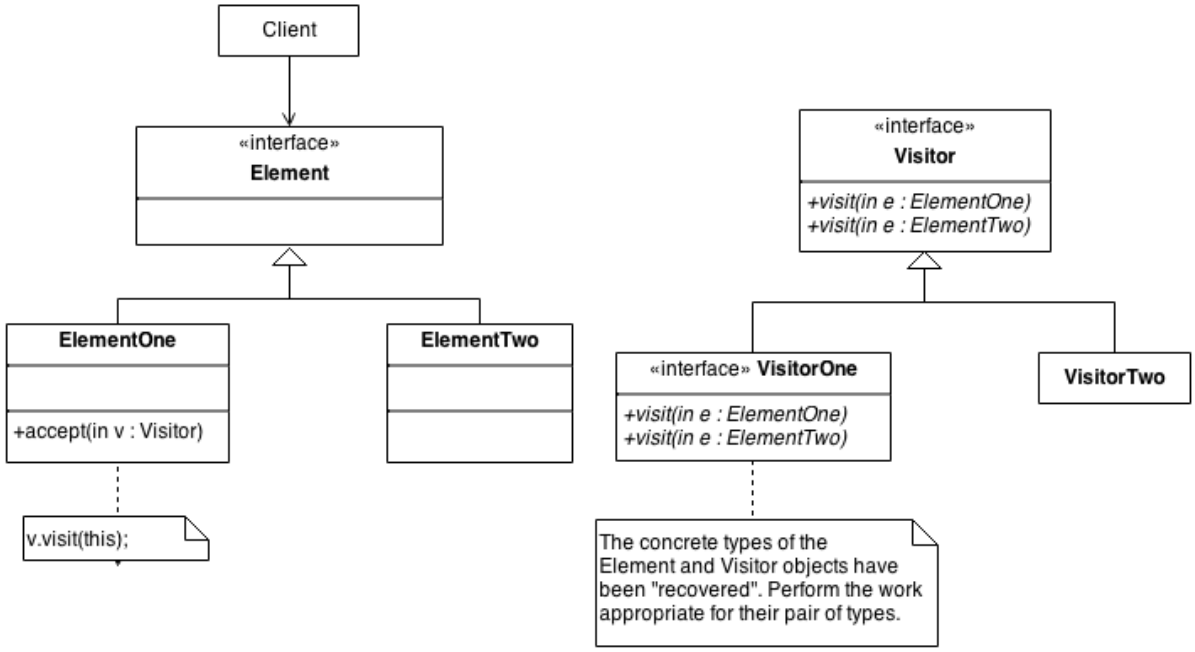
8) Everything for "elements" and "visitors" is now set-up.

i) When the client needs an operation to be performed, s(he) creates an instance of the Visitor object, calls the accept() method on each Element object, and passes the Visitor object.

9) The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass.

10) The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class.

11) But, if the subclasses(derived classes) in the aggregate node hierarchy are not stable, keeping this Visitor subclasses in sync requires a prohibitive amount of effort.



EXAMPLE

The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is.

Cab company dispatcher

Object structure is list of Customers

Customer

Concrete element of Customers list

Taxi

Visitor