Visitor Design Pattern

Intent

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Discussion

Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

The implementation proceeds as follows. 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. Each `visit()` method accepts a single argument - a pointer or reference to an original Element derived class.

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

Add a single pure virtual `accept()` method to the base class of the Element hierarchy. `accept()` is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

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.

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

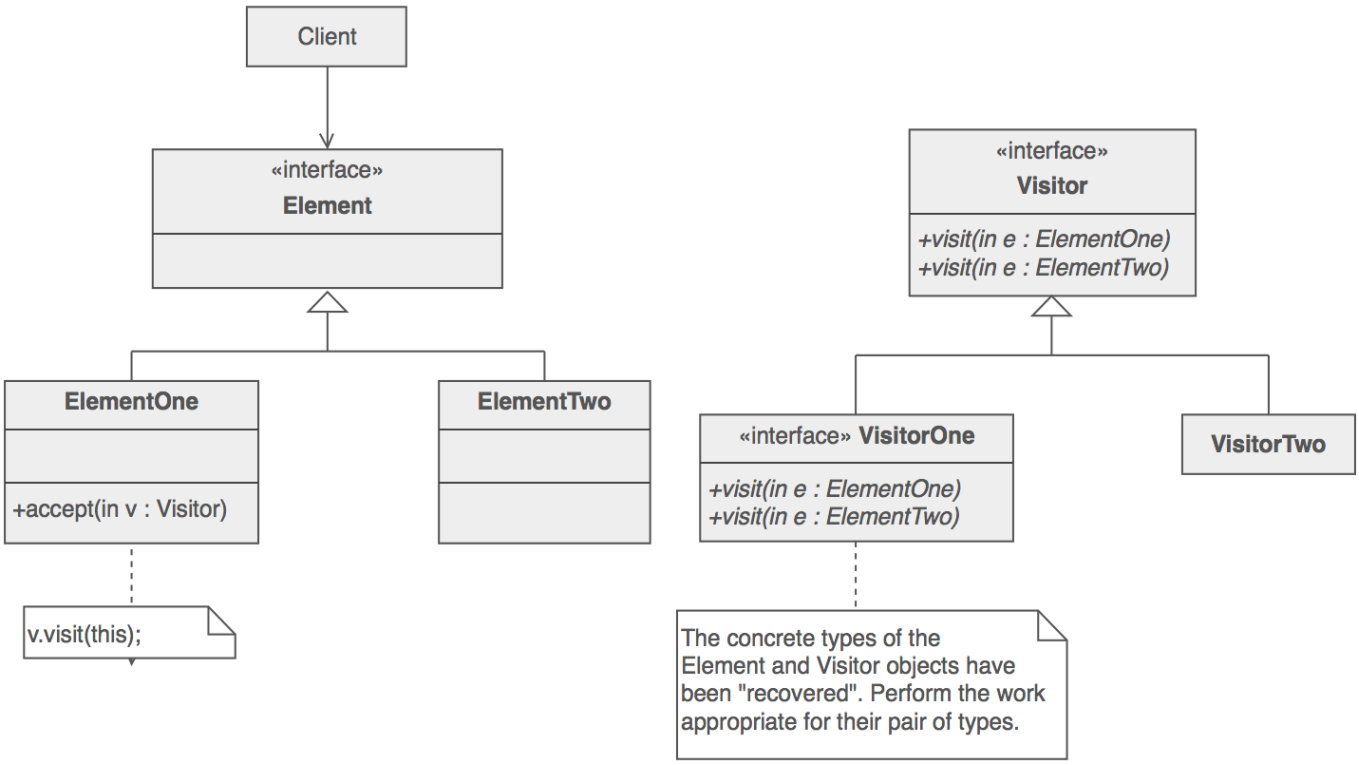
The `accept()` method causes flow of control to find the correct Element subclass. Then when the `visit()` method is invoked, flow of control is vectored to the correct Visitor subclass. `accept()` dispatch plus `visit()` dispatch equals double dispatch.

The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.

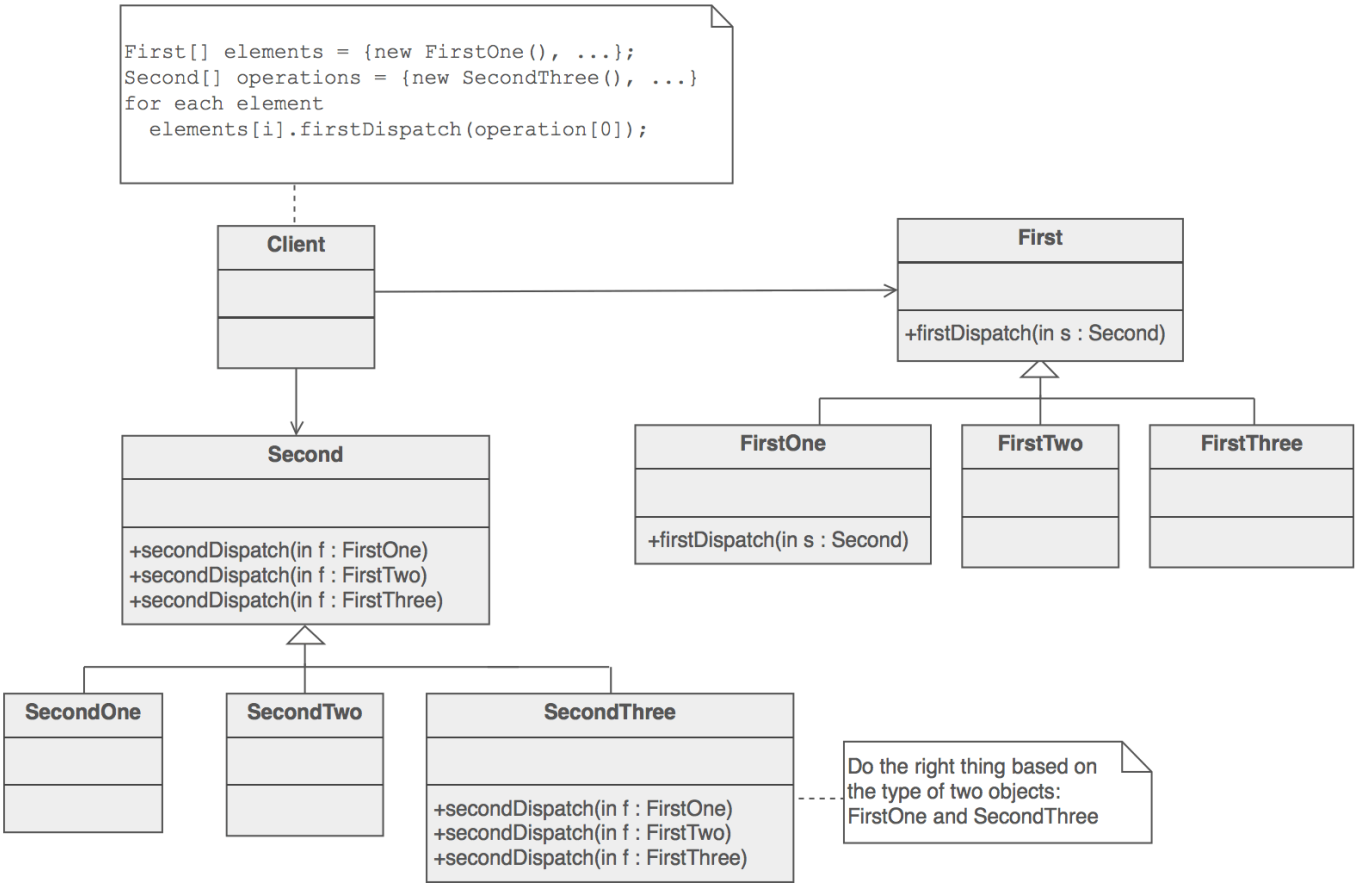
An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/rationale is the goal of promoting non-traditional behavior to full object status.

Structure

The Element hierarchy is instrumented with a "universal method adapter". The implementation of `accept()` in each Element derived class is always the same. But – it cannot be moved to the Element base class and inherited by all derived classes because a reference to `this` in the Element class always maps to the base type Element.

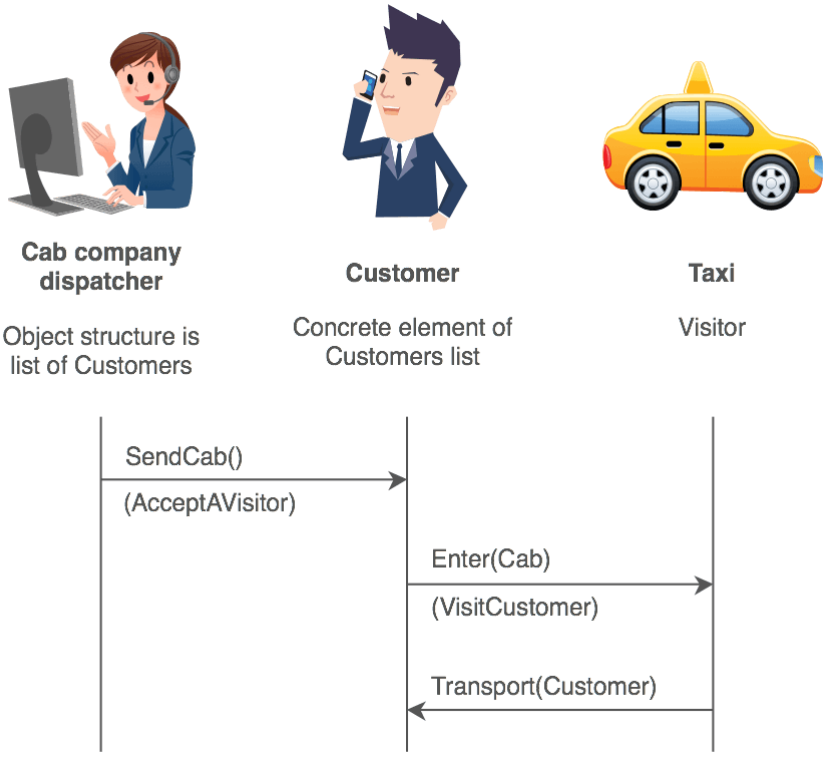


When the polymorphic `firstDispatch()` method is called on an abstract `First` object, the concrete type of that object is "recovered". When the polymorphic `secondDispatch()` method is called on an abstract `Second` object, its concrete type is "recovered". The application functionality appropriate for this pair of types can now be exercised.



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.



Check list

1. Confirm that the current hierarchy (known as the Element hierarchy) will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor classes will require. If these conditions are not met, then the Visitor pattern is not a good match.
2. Create a Visitor base class with a `visit(ElementXxx)` method for each Element derived type.
3. Add an `accept(Visitor)` method to the Element hierarchy. The implementation in each Element derived class is always the same – `accept( Visitor v ) { v.visit( this ); }`. Because of cyclic dependencies, the declaration of the Element and Visitor classes will need to be interleaved.
4. The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class. If the stability of the Element hierarchy is low, and the stability of the Visitor hierarchy is high; consider swapping the 'roles' of the two hierarchies.
5. Create a Visitor derived class for each "operation" to be performed on Element objects. `visit()` implementations will rely on the Element's public interface.
6. The client creates Visitor objects and passes each to Element objects by calling `accept()`.

Rules of thumb

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.
- The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

Notes

The November 2000 issue of JavaPro has an article by James Cooper (author of a Java companion to the GoF) on the Visitor design pattern. He suggests it "turns the tables on our object-oriented model and creates an external class to act on data in other classes ... while this may seem unclean ... there are good reasons for doing it."

His primary example. Suppose you have a hierarchy of Employee-Engineer-Boss. They all enjoy a normal vacation day accrual policy, but, Bosses also participate in a "bonus" vacation day program. As a result, the interface of class Boss is different than that of class Engineer. We cannot polymorphically traverse a Composite-like organization and compute a total of the organization's remaining vacation days. "The Visitor becomes more useful when there are several classes with different interfaces and we want to encapsulate how we get data from these classes."

His benefits for Visitor include:

- Add functions to class libraries for which you either do not have the source or cannot change the source
- Obtain data from a disparate collection of unrelated classes and use it to present the results of a global calculation to the user program
- Gather related operations into a single class rather than force you to change or derive classes to add these operations
- Collaborate with the Composite pattern

Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended.

Read next

Code examples

Java	**Visitor in Java**	**Visitor in Java: Double dispatch (within a single hierarchy)**	**Visitor in Java**	**Visitor in Java**
C++	**Visitor in C++: Before and after**	**Visitor in C++**	**Visitor in C++: Recovering lost type information**	
PHP	**Visitor in PHP**			
Delphi	**Visitor in Delphi**			
Python	**Visitor in Python**			

RETURN

Design Patterns
AntiPatterns
My account
Forum

Refactoring
UML

Contact us
About us