18

VISITOR

This pattern was previously described in GoF95.

DESCRIPTION

The Visitor pattern is useful in designing an operation across a heterogeneous collection of objects of a class hierarchy. The Visitor pattern allows the operation to be defined without changing the class of any of the objects in the collection.

To accomplish this, the Visitor pattern suggests defining the operation in a separate class referred to as a *visitor* class. This separates the operation from the object collection that it operates on. For every new operation to be defined, a new `visitor` class is created. Since the operation is to be performed across a set of objects, the visitor needs a way of accessing the public members of these objects. This requirement can be addressed by implementing the following two design ideas.

Design Idea 1

Every `visitor` class that operates on objects of the same set of classes can be designed to implement a corresponding `VisitorInterface` interface. A typical `VisitorInterface` declares a set of `visit(ObjectType)` methods, one for each object type from the object collection. Each of these methods is meant for processing instances of a specific class. For example, if the object collection consists of objects of `ClassA` and `ClassB`, then the `VisitorInterface` interface would declare the following two methods:

```
visit(ClassA objClassA)
```

for processing `ClassA` objects.

```
visit(ClassB objClassB)
```

for processing `ClassB` objects.

Every object from the object collection makes a call to the respective `visit(ObjectType)` method, passing itself as an argument. A typical

implementer (visitor) of the `VisitorInterface` can access the information required for the operation it is designed for by accessing the public members (methods and attributes) of the object instance passed to it through the `visit` method call.

Design Idea 2

Classes of objects from the object collection need to define a method:

```
accept(visitor)
```

A client interested in executing the visitor operation needs to:

- Create an instance of the implementer (visitor) of the `VisitorInterface` interface that is designed to carry out the required operation.
- Create the object collection and invoke the `accept(visitor)` method on every member of the object collection by passing the visitor instance created above.

As part of the `accept(visitor)` method implementation, every object in the object collection invokes the `visit(ObjectType)` method on the visitor instance. Inside the `visit(ObjectType)` method, the visitor gathers the required data from the object collection to perform the operation it is designed for.

DEFINING NEW OPERATIONS ON THE OBJECT COLLECTION

Whenever a new operation is to be defined across the object collection, a new `visitor` class needs to be created with implementation for the new operation. The `visitor` class needs to implement all of the `visit(ObjectType)` methods declared in the `VisitorInterface` interface to process different types of objects.

With this design, defining a new operation does not require any changes to the classes of the object collection.

ADDING OBJECTS OF A NEW TYPE TO THE COLLECTION

Whenever a new type of object is to be added to the object collection and is to be referred within the scope of an already existing visitor operation:

- The class of the object must provide a method similar to `accept(visitor)` and as part of this method implementation it should invoke the `visit(ObjectType)` method on the `visitor` object passing itself as an argument to it.
- A corresponding `visit(ObjectType)` method needs to be added to the `VisitorInterface` interface and needs to be implemented by all the concrete `visitor` classes.

This means that for objects of an existing class to be added to the object collection and to be considered within the scope of an existing visitor operation, the class needs to be altered to implement a method similar to `accept(visitor)`, if one does not already exist. This implies that when the Visitor pattern is applied for the first time to a class, one should have access to its source code or the class needs to be subclassed to add the `accept(visitor)` method implementation.

EXAMPLE

Let us design an application to define operations over a collection of different `Order` objects. Orders can be of different types. Let us consider three different types of orders as follows:

- *Overseas order* — Order from countries other than the United States. Additional shipping and handling is charged for this type of order.
- *California order* — U.S. order with shipping address in California. Additional sales tax is charged on this type of order.
- *Non-California order* — U.S. order with shipping address not in California. Additional sales tax is not applicable.

DESIGN APPROACH I

Let us assume that we would like to define an operation `getMaxCAOrderAmount` to find the top dollar amount on a California order. We can define a generic `Order` class as in the following Figure 18.1.

The `Order` class maintains a static member variable `orderTotalCA` to keep track of the order amount. Whenever a new order is created, the `orderTotalCA` member variable is updated with the order amount if the new order amount is greater than the old order total already available in the `orderTotalCA` static member variable.

Let us say that we would like to add more methods, such as `getMinCA-OrderAmount`, `getCAOrderTotal`, `getMinNonCAOrderAmount`, `getMax-OverseasOrderAmount`, etc., to find out different types of order amounts.

The `Order` class code needs to be altered for each such new operation. As a result, the class can quickly become cluttered (Figure 18.2).

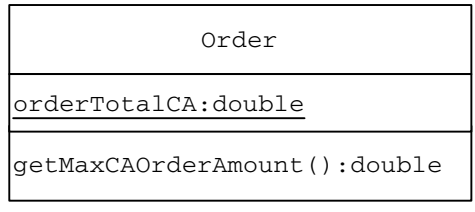

Figure 18.1 `Order` Class

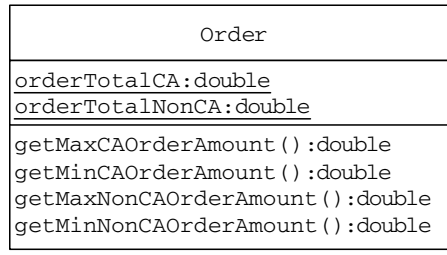

Figure 18.2 `Order` Class with New Methods

DESIGN APPROACH II

Since orders are of three different types, we can design three subclasses of the `Order` class, each representing a specific order type (Figure 18.3).

With this new design, related operations can be kept within an appropriate `Order` subclass. Although this class structure is more manageable and less cluttered, it suffers from the following two limitations:

- There is no appropriate place to define operations involving different order types.
- Whenever a new operation is to be defined in any `Order` class, it requires a change to the class code.

DESIGN APPROACH III (COMPOSITE PATTERN)

During the discussion of the Composite pattern, we defined operations on an object collection containing objects from a class hierarchy. A `Composite` object is designed to maintain this object collection and operate on this collection.

Applying the Composite pattern, we can define a composite `OrderComposite` class as in Figure 18.4 to define operations on a collection of different `order` objects.

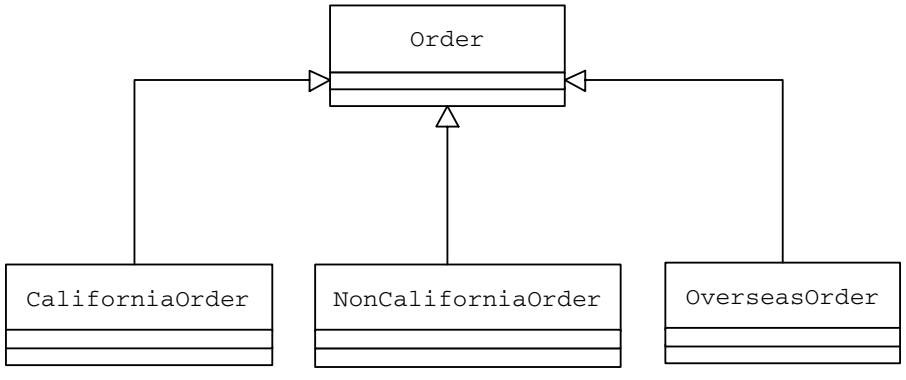

Figure 18.3 `Order` Class Hierarchy

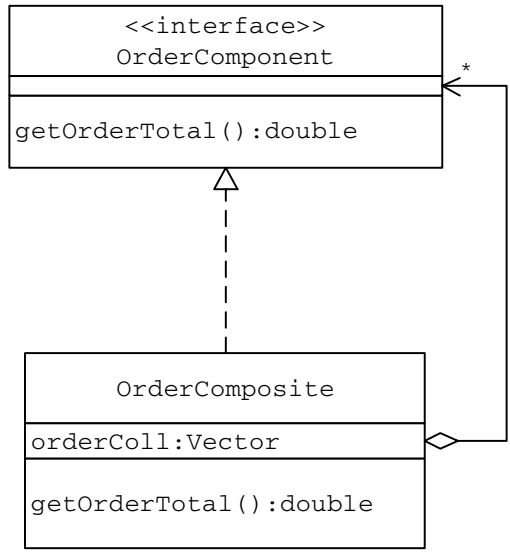

Figure 18.4 `OrderComposite`: Class Structure

The attribute `orderColl` can be used to store different types of `Order` objects and the `getOrderTotal` iterates over this collection to retrieve and sum up different order amounts.

This design does not fully address the limitations of the Design Approach II. It allows the definition of an operation over a heterogeneous collection of objects, but it requires changes to the `OrderComposite` hierarchy classes whenever a new operation is to be added.

DESIGN APPROACH IV (THE VISITOR PATTERN)

Let us define an order class hierarchy as in Figure 18.5 with an interface `Order` at the top of the hierarchy and three of its implementers — `CaliforniaOrder`, `NonCaliforniaOrder` and `OverseasOrder` — each representing a specific order type. The `Order` interface declares a method `accept(OrderVisitor)`. Each of the `Order` implementers provides implementation for the `accept(OrderVisitor)` method (Listings 18.1 through 18.4).

The `VisitorInterface` can be designed as a Java interface that declares a set of `visit(OrderType)` methods, one for each class in the order class hierarchy. In other words, these methods are meant to process different types of orders.

```
public interface VisitorInterface {
  public void visit(NonCaliforniaOrder nco);
  public void visit(CaliforniaOrder co);
  public void visit(OverseasOrder oo);
}
```

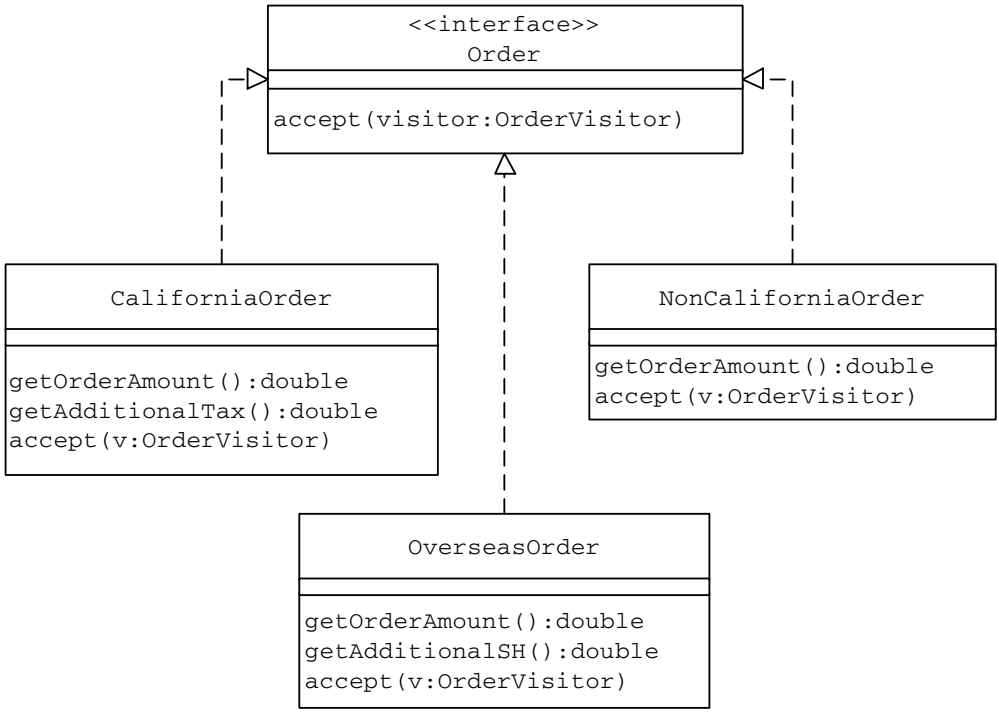

Figure 18.5 Order Class Hierarchy

Listing 18.1 Order Interface

```
public interface Order {
  public void accept(OrderVisitor v);
}
```

Let us define a visitor `OrderVisitor` as an implementer of the `VisitorInterface` (Figure 18.6 and Listing 18.5), to calculate the sum of all order totals.

As part of its implementation of the `VisitorInterface` methods, the `OrderVisitor` retrieves the order amount, any additional tax or shipping and handling amounts from different `Order` objects and maintains a cumulative sum of these amounts in a private instance variable `orderTotal`.

Application Flow

When run, the client `OrderManager` creates an instance of the `OrderVisitor` and displays the necessary user interface to allow a user to create different types of orders (Figure 18.7).

Listing 18.2 `CaliforniaOrder` Class

```
public class CaliforniaOrder implements Order {
  private double orderAmount;
  private double additionalTax;
  public CaliforniaOrder() {
  }
  public CaliforniaOrder(double inp_orderAmount,
      double inp_additionalTax) {
    orderAmount = inp_orderAmount;
    additionalTax = inp_additionalTax;
  }
  public double getOrderAmount() {
    return orderAmount;
  }
  public double getAdditionalTax() {
    return additionalTax;
  }
  public void accept(OrderVisitor v) {
    v.visit(this);
  }
}
```

Listing 18.3 `NonCaliforniaOrder` Class

```
public class NonCaliforniaOrder implements Order {
  private double orderAmount;
  public NonCaliforniaOrder() {
  }
  public NonCaliforniaOrder(double inp_orderAmount) {
    orderAmount = inp_orderAmount;
  }
  public double getOrderAmount() {
    return orderAmount;
  }
  public void accept(OrderVisitor v) {
    v.visit(this);
  }
}
```

Listing 18.4 `OverseasOrder` Class

```
public class OverseasOrder implements Order {
  private double orderAmount;
  private double additionalSH;
  public OverseasOrder() {
  }
  public OverseasOrder(double inp_orderAmount,
      double inp_additionalSH) {
    orderAmount = inp_orderAmount;
    additionalSH = inp_additionalSH;
  }
  public double getOrderAmount() {
    return orderAmount;
  }
  public double getAdditionalSH() {
    return additionalSH;
  }
  public void accept(OrderVisitor v) {
    v.visit(this);
  }
}
```

Every time a user enters the order data and clicks on the `CreateOrder` button, the client `OrderManager` (Listing 18.6):

- Creates an `Order` object with the input data.
- Invokes the `accept(OrderVisitor)` method on the `Order` object by passing the `OrderVisitor` object. The `Order` object internally calls the `OrderVisitor visit` method by passing itself as an argument. The `OrderVisitor` retrieves the required order amount, tax and shipping amounts using the public methods defined by different `Order` classes and adds these amounts, in a cumulative manner, to the order total kept inside the private instance variable `orderTotal`.

When the `GetTotal` button is clicked, the client `OrderManager` invokes the `getOrderTotal` method of the `OrderVisitor`. The `OrderVisitor` simply returns the value stored in the `orderTotal` instance variable, which is the total value of all orders created.

The sequence diagram in Figure 18.8 depicts the message flow, when the client `OrderManager` makes use of the visitor `OrderVisitor` to calculate the grand total of a set of different order amounts. In order to keep the diagram simple, only one type of order is included.

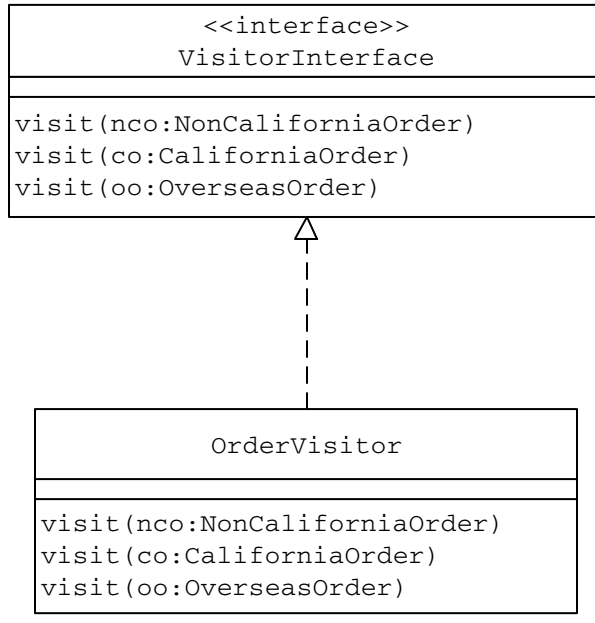

Figure 18.6 `OrderVisitor` Class Structure

DEFINING A NEW OPERATION ON THE ORDER OBJECT COLLECTION

Defining a new operation on the order object collection requires the creation of a new visitor. The new visitor needs to implement the `VisitorInterface` interface providing implementation for different `visit(OrderType)` methods to process different types of `Order` objects.

ADDING A NEW ORDER TYPE TO THE COLLECTION

If a new type of object (a new class) is to be added to the object structure such as a `DiscountOrder` that implements the `Order` interface, then a corresponding `visit(DiscountOrder)` method needs to be added to the `VisitorInterface` and needs to be implemented by the `OrderVisitor` class.

PRACTICE QUESTIONS

1. As part of our discussion of the Composite pattern, we designed a composite `DirComponent` class. Redesign the `DirComponent` class operations applying the Visitor pattern.
2. The Practice Questions section of the Composite pattern discussion lists three applications involving operations on a heterogeneous object collection. Apply the Visitor pattern in designing these operations.

Listing 18.5 `OrderVisitor` Class

```
class OrderVisitor implements VisitorInterface {
  private Vector orderObjList;
  private double orderTotal;
  public OrderVisitor() {
    orderObjList = new Vector();
  }
  public void visit(NonCaliforniaOrder inp_order) {
    orderTotal = orderTotal + inp_order.getOrderAmount();
  }
  public void visit(CaliforniaOrder inp_order) {
    orderTotal = orderTotal + inp_order.getOrderAmount() +
                 inp_order.getAdditionalTax();
  }
  public void visit(OverseasOrder inp_order) {
    orderTotal = orderTotal + inp_order.getOrderAmount() +
                 inp_order.getAdditionalSH();
  }
  public double getOrderTotal() {
    return orderTotal;
  }
}
```

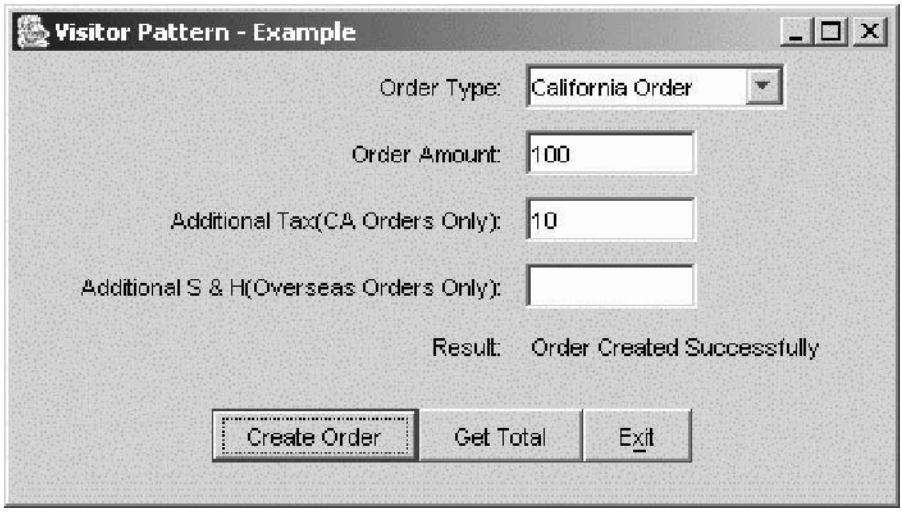

Figure 18.7 Order Manager: User Interface to Create Orders

Listing 18.6 `OrderManager` Class

```
        …
        …
public void actionPerformed(ActionEvent e) {
  String totalResult = null;
  if (e.getActionCommand().equals(OrderManager.EXIT)) {
    System.exit(1);
  }
  if (e.getActionCommand().equals(OrderManager.CREATE_ORDER)
      ) {
    //get input values
    String orderType = objOrderManager.getOrderType();
    String strOrderAmount =
      objOrderManager.getOrderAmount();
    String strTax = objOrderManager.getTax();
    String strSH = objOrderManager.getSH();
    double dblOrderAmount = 0.0;
    double dblTax = 0.0;
    double dblSH = 0.0;
    if (strOrderAmount.trim().length() == 0) {
      strOrderAmount = "0.0";
    }
    if (strTax.trim().length() == 0) {
      strTax = "0.0";
    }
    if (strSH.trim().length() == 0) {
      strSH = "0.0";
    }
    dblOrderAmount =
      new Double(strOrderAmount).doubleValue();
    dblTax = new Double(strTax).doubleValue();
    dblSH = new Double(strSH).doubleValue();
    //Create the order
    Order order = createOrder(orderType, dblOrderAmount,
                  dblTax, dblSH);
```

(continued)

Listing 18.6 `OrderManager` Class (Continued)

```
        //Get the Visitor
        OrderVisitor visitor =
          objOrderManager.getOrderVisitor();
        //accept the visitor instance
        order.accept(visitor);
        objOrderManager.setTotalValue(
          " Order Created Successfully");
      }
      if (e.getActionCommand().equals(OrderManager.GET_TOTAL)) {
        //Get the Visitor
        OrderVisitor visitor =
          objOrderManager.getOrderVisitor();
        totalResult = new Double(
                      visitor.getOrderTotal()).toString();
        totalResult = " Orders Total = " + totalResult;
        objOrderManager.setTotalValue(totalResult);
      }
    }
    public Order createOrder(String orderType,
        double orderAmount, double tax, double SH) {
      if (orderType.equalsIgnoreCase(OrderManager.CA_ORDER))
      {
        return new CaliforniaOrder(orderAmount, tax);
      }
      if (orderType.equalsIgnoreCase(
        OrderManager.NON_CA_ORDER)) {
        return new NonCaliforniaOrder(orderAmount);
      }
      if (orderType.equalsIgnoreCase(
            OrderManager.OVERSEAS_ORDER)) {
        return new OverseasOrder(orderAmount, SH);
      }
      return null;
    }
            …
            …
```

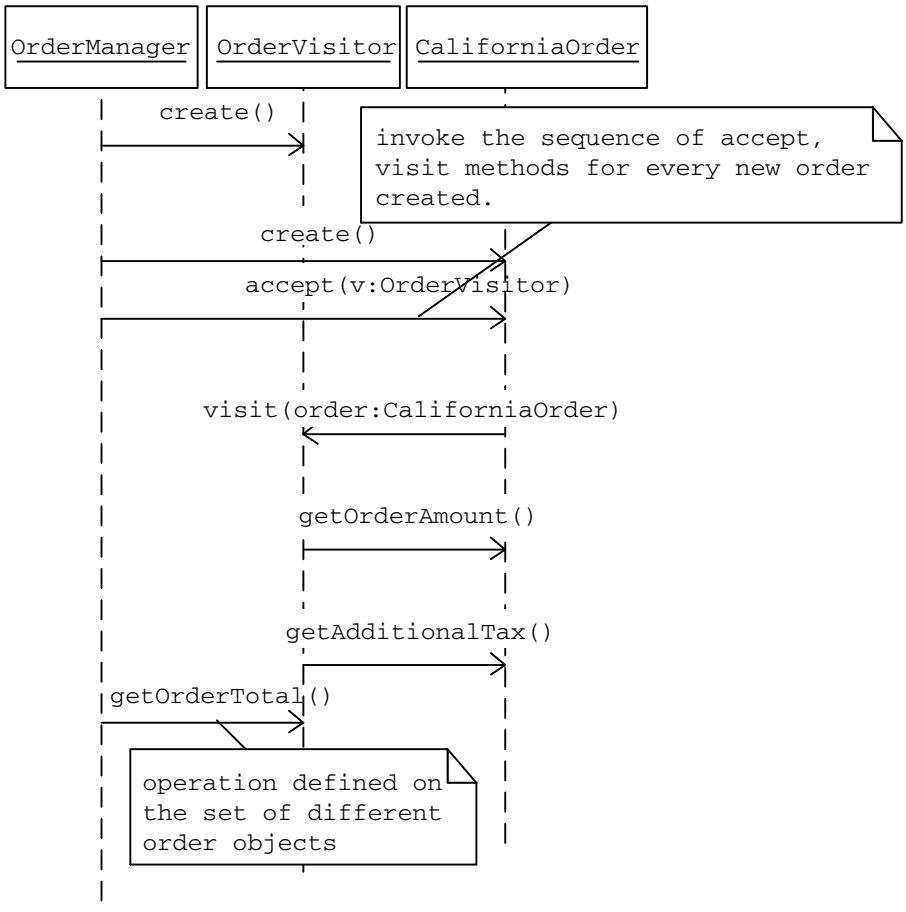

Figure 18.8 Application Message Flow