

# Visitor Pattern

## Visitor Pattern Motivation

Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type.

A possible approach to apply a specific operation on objects of different types in a collection would be the use of blocks in conjunction with 'instanceof' for each element. This approach is not a nice one, not flexible and not object oriented at all. At this point we should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns Visitor Pattern

## Motivation

Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type.

A possible approach to apply a specific operation on objects of different types in a collection would be the use of blocks in conjunction with 'instanceof' for each element. This approach is not a nice one, not flexible and not object oriented at all. At this point we should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

## Intent

- Represents 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.

## Implementation

The participants classes in this pattern are:

- Visitor - This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes. Usually the name of the operation is the same and the operations are differentiated by the method signature: The input object type decides which of the method is called.
- ConcreteVisitor - For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.
- Visitable - is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object. Each object from a collection should implement this abstraction in order to be able to be visited.
- ConcreteVisitable - Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.
- ObjectStructure - This is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. It can be a complex structure, such as a composite object.

Applicability & ExamplesThe visitor pattern is used when:

- Similar operations have to be performed on objects of different types grouped in a structure (a collection or a more complex structure).
- There are many distinct and unrelated operations needed to be performed. Visitor pattern allows us to create a separate visitor concrete class for each type of operation and to separate this operation implementation from the objects structure.
- The object structure is not likely to be changed but is very probable to have new operations which have to be added. Since the pattern separates the visitor (representing operations, algorithms, behaviors) from the object structure it's very easy to add new visitors as long as the structure remains unchanged.

Example 1 - Customers Application.

We want to create a reporting module in our application to make statistics about a group of customers. The statistics should made very detailed so all the data related to the customer must be parsed. All the entities involved in this

hierarchy must accept a visitor so the CustomerGroup, Customer, Order and Item are visitable objects.

In the example we can see the following actors:

- IVisitor and IVisitable interfaces
- CustomerGroup, Customer, Order and Item are all visitable classes. A CustomerGroup represents a group of customers, each Customer can have one or more orders and each order can have one or more Items.
- GeneralReport is a visitor class and implements the IVisitor interface. [Source: Click here to see java source code](#)

Specific problems and implementation

Tight Coupled Visitable objects

The classic implementation of the Visitor pattern have a major drawback because the type of visitor methods has to be known in advance. The Visitor interface can be defined using polymorphic methods or methods with different

```
names:public interface IVisitor {
    public void visit(Customer customer);
    public void visit(Order order);
    public void visit(Item item);
}
```

```
public interface IVisitor {
    public void visitCustomer(Customer customer);
    public void visitOrder(Order order);
    public void visitItem(Item item);
}
```

However this type should be known in advance. When a new type is added to the structure a new method should be added to this interface and all existing visitors have to be changed accordingly. A pair method is written in the concrete

```
Visitable objects:public class Customer implements IVisitable{
    public void accept(IVisitor visitor)
    {
        visitor.visit(this);
    }
}
```

It doesn't really matters if the polymorphic methods with the same name but different signatures are used or not, because in either way the type is known at compile time so for each new visitable object this method must be implemented accordingly. The main advantage of the fact that new visitors can be easily added is compensated by the fact that the addition of new visitable objects is really hard.

Visitor Pattern using Reflection

Reflection can be used to overcome the main drawback of the visitor pattern. When the standard implementation of visitor pattern is used the method to invoke is determined at runtime. Reflection is the mechanism used to determine the method to be called at compile-time. This way the visitable object will use the same code in the accept method. This code can be moved in an abstraction so the IVisitable interface will be transformed to an abstract class.

Let's take our example. We need to add a new visitable class in our structure, called Product. We should modify the IVisitor interface to add a visitProduct method. But changing an interface is one of the worst things that can be done. Usually, interfaces are extended by lots of classes changing the interface means changing the classes. Maybe we have lots of visitors but we don't want to change all of them, we need only another report.

In this case we start from the idea that we should keep the interface unchanged. The solution is to replace the interface with an abstract class and to add an abstract method called defaultVisit. The defaultVisit will be implemented by each new concrete visitor, but the interface and old concrete visitors will remain unchanged.

The code is very simple: the visit(Object object) method check if there is visit method for the specific object. If there is not an available visit the call is delegated to the defaultVisit method:public abstract class Visitor {

```
public abstract void visit(Customer customer);
public abstract void visit(Order order);
public abstract void visit(Item item);
public abstract void defaultVisit(Object object);
```

```
public void visit(Object object) {
```

```

try
{
    Method downPolymorphic = object.getClass().getMethod("visit",
        new Class[] { object.getClass() });

    if (downPolymorphic == null) {
        defaultVisit(object);
    } else {
        downPolymorphic.invoke(this, new Object[] {object});
    }
}
catch (NoSuchMethodException e)
{
    this.defaultVisit(object);
}
catch (InvocationTargetException e)
{
    this.defaultVisit(object);
}
catch (IllegalAccessException e)
{
    this.defaultVisit(object);
}
}
}

```

Another point that should be marked is the defaultVisit method: We should visit only classes we know: public void defaultVisit(Object object)

```

{
    // if we don't know the class we do nothing
    if (object.getClass().equals(Product.class))
    {
        System.out.println("default visit: "
            + object.getClass().getSimpleName());
        itemsNo++;
    }
}
}

```

### Statefull Visitors

The visitors objects can be complex objects and can maintain a context during a traversal.  
Encapsulation of visitable objects

The behavior is defined in the visitor itself and the objects structure is represented by visitable objects. The Visitor needs to access data kept by visitable objects so practically the pattern forces to expose from visitable objects the data required in the visitor, using public methods.

### Visitors and Iterators

The iterator pattern and visitor pattern has the same benefit, they are used to traverse object structures. The main difference is that the iterator is intended to be used on collections. Usually collections contain objects of the same type. The visitor pattern can be used on complex structure such as hierarchical structures or composite structures. In this case the accept method of a complex object should call the accept method of all the child objects.

Another difference is operation performed on the objects: In one case the visitor defines the operations that should be performed, while the iterator is used by the client to iterate through the objects form a collection and the operations are defined by the client itself.

### Visitors and Composites

The visitor pattern can be used in addition with the composite pattern. The object structure can be a composite structure. In this case in the implementation of the accept method of the composite object the accept methods of the component object has to be invoked.

### Hot Points:

- The visitor pattern is a great way to provide a flexible design for adding new visitors to extend existing functionality without changing existing code
- The Visitor pattern comes with a drawback: If a new visitable object is added to the framework structure all the implemented visitors need to be modified. The separation of visitors and visitable is only in one sense: visitors depend of

visitable objects while visitable are not dependent of visitors.

- Part of the dependency problems can be solved by using reflection with a performance cost.