eeksforGeeks
A computer science portal for geeks

es

Visitor design pattern

GeeksforGeeks
A computer science portal for geeks

Visitor design pattern is one of the behavioral design patterns. It is used when we have to perform an operation on a group of simi… …or pattern, we can move the operational logic from the objects to another class.

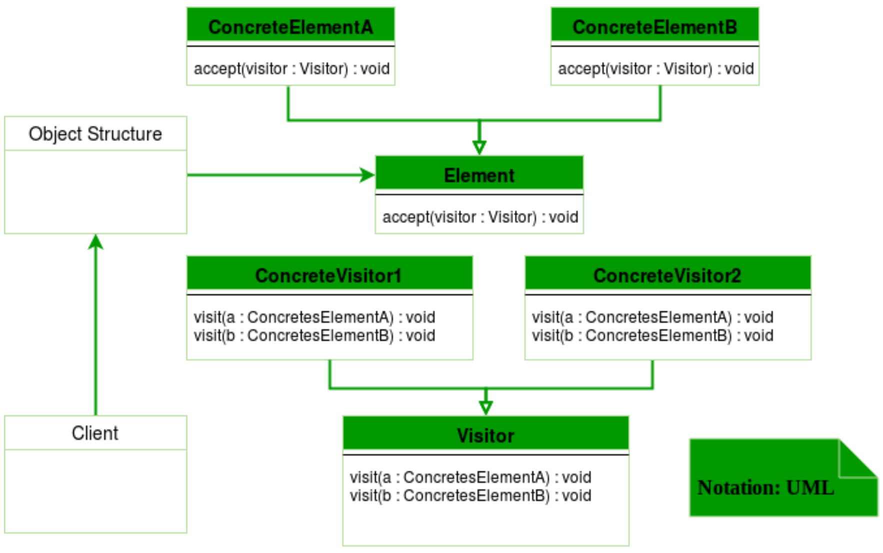
in

Hire with us!

The visitor pattern consists of two parts:

- a method called **Visit()** which is implemented by the visitor and is called for every element in the data structure
- visitable classes providing **Accept()** methods that accept a visitor

UML Diagram Visitor design pattern


ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void
Object Structure
Element
accept(visitor : Visitor) : void
ConcreteVisitor1
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor2
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Client
Visitor
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Notation: UML

Design components

▲

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Got It !

processing.

- **Visitor :** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- **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.
- **Visitable :** This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
- **ConcreteVisitable :** These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Let’s see an example of Visitor design pattern in Java.

```
interface ItemElement
{
    public int accept(ShoppingCartVisitor visitor);
}

class Book implements ItemElement
{
    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice()
    {
        return price;
    }

    public String getIsbnNumber()
    {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}

class Fruit implements ItemElement
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Got It !

```java
    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}

interface ShoppingCartVisitor
{

    int visit(Book book);
    int visit(Fruit fruit);
}

class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{

    @Override
    public int visit(Book book)
    {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50)
        {
            cost = book.getPrice()-5;
        }
        else
            cost = book.getPrice();

        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit)
    {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }

}

class ShoppingCartClient
{
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Got It !

```
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"),
                              new Book(100, "5678"), new Fruit(10, 2, "Banana"),
                              new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items)
    {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
        {
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

Output:

```
Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

Here, in the implementation if accept() method in all the items are same but it can be different. For example there can be logic to check if item is free then don't call the visit() method at all.

Advantages :

- If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Disadvantages :

- We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
- If there are too many implementations of visitor interface, it makes it hard to extend.

Another example of visitor pattern in C++

```
//Write CPP code here

#include <iostream>
using namespace std;

class Stock
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Got It !