Lesson 11 Visitor and Interpreter

Visitor

1. 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.

2. Motivation

   a. Polymorphism works well for objects of the same hierarchy when you iterate through a collection of them and call a common method, also known as ‘Single Dispatch’.
   b. But sometimes ‘Single dispatch’ does not support certain operations well (for example, to accumulate state information in the object structure, or apply business rules depending on the amount of the accumulated information).
   c. Visitor also allows us to externalize operations of an object structure into a separate class yet still supports polymorphism by “Double Dispatch”.
   d. Double Dispatch is a mechanism that dispatches two function calls depending on the real (runtime) types of the references on which the methods are called, as we see it in the demo: `emp.print(manager);`

```
public class Employee {
      public void print(Employee emp){
            emp.print();
      }

      public void print(Manager manager){
```

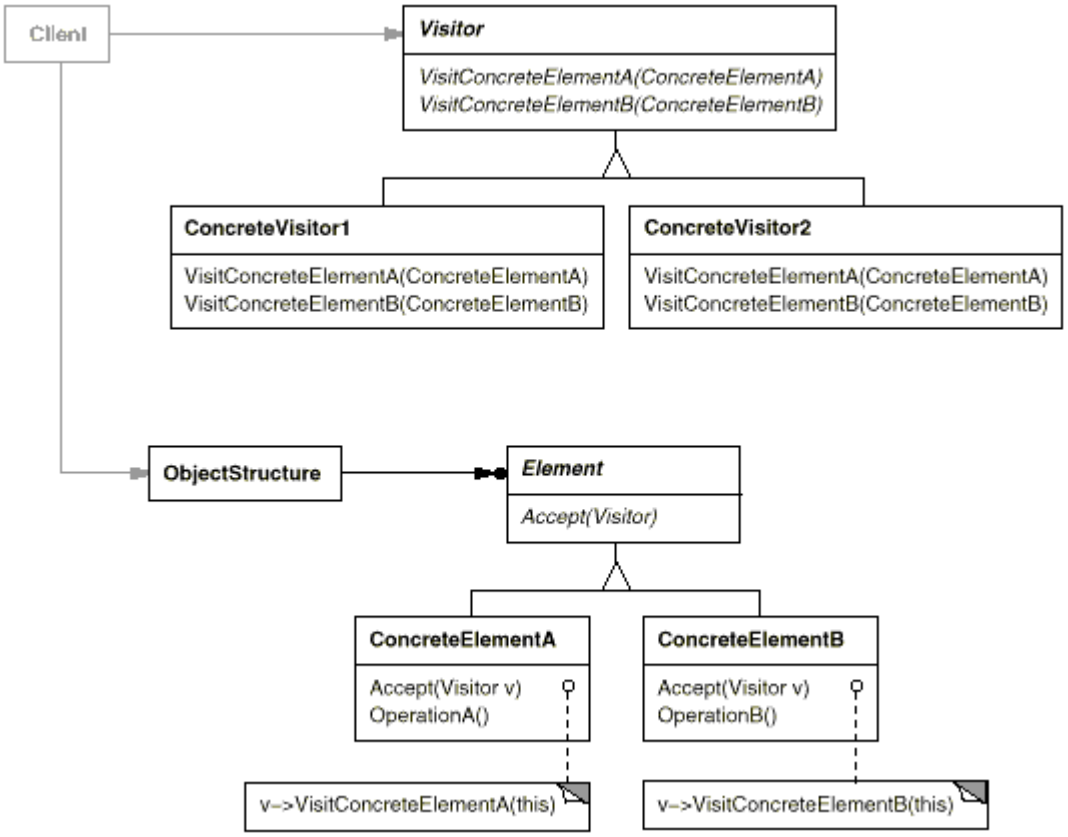
```
            manager.print();
    }
```

3. Applicability

Use the Visitor pattern when

a. An object structure contains objects from different class hierarchies, and you want to perform operations on these objects that depend on their concrete classes.
b. Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting” the classes with these operations. Visitor lets you keep related operations together by defining them in one class. For example, the shoppingCart’s calculateTotal() and calculateShippingFee() methods are considered “unrelated” operations.
c. The class defining the object structure rarely changes, but you often want to define new operations over the structure. Changing the object structure class requires redefining the interface to all visitors, which is potentially costly. If the object structure class changes often, then it's probably better to define the operations in those classes.

4. Structure



5. Participants

a. Visitor
   - declares a Visit operation for each class of ConcreteElement in the object structure.

b. ConcreteVisitor
   - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

c. Element
   - defines an ‘accept’ operation that takes a visitor as an argument.

d. ConcreteElement
   - implements the ‘accept’ operation that takes a visitor as an argument.

e. ObjectStructure
   - can enumerate its elements, as a composite structure or a collection such as a list or a set.

6. Consequences

Some of the benefits and liabilities of the Visitor pattern are as follows:

a. Visitor makes adding new operations easy.
b. A visitor gathers related operations and separates unrelated ones. (for example, separate getShippingFee from OrderItem)
c. Adding a new Concrete Element class is hard.
d. Can accumulate state across different objects in an object structure.
e. It may compromise encapsulation.

7. How to implement the Visitor pattern?

a. Element interface

```
public interface OrderItem {
      public void accept(Visitor visitor);
}
```

b. Concrete element classes

```
public class Book implements OrderItem {

         private double price;
         private double weight;
```

```
        //accept the visitor
        public void accept(Visitor visitor){
            visitor.visit(this);
        }

    …

}

public class DVD implements OrderItem {

        private double price;
        private double weight;

        //accept the visitor
        public void accept(Visitor visitor){
            visitor.visit(this);
            …
}
```

c. Visitor interface

```
public interface Visitor {
    public void visit(Book book);
    public void visit(DVD dvd);
    public void visit(Toy toy);
}
```

d. Concrete visitor classes

```
//Or call it OrderPriceCalculator
public class OrderPriceVisitor implements Visitor {
    private double totalPrice = 0.0;

    @Override
    public void visit(Book book) {
        totalPrice += book.getPrice();
    }
    @Override
    public void visit(Toy toy) {
        totalPrice += toy.getPrice();
    }
    @Override
    public void visit(DVD dvd) {
        totalPrice += dvd.getPrice();
    }

    public double getOrderTotal(){
        return totalPrice;
    }

}
```

```
public class ShippingFeeVisitor implements Visitor {
	private double totalShippingFee;

	// collect data about the book
	public void visit(Book book) {
		// assume shipping fee is related to weight and price
		// free shipping for all books over $10
		if (book.getPrice() < 10.0) {
			totalShippingFee += book.getWeight() * 2;
		}
	}

	public void visit(DVD dvd) {
		// free shipping for all DVDs
	}

	public void visit(Toy toy) {
		// free shipping for all DVDs
		totalShippingFee += 1.50;

	}

	public double getTotalShippingFee() {
		return totalShippingFee;
	}
}
```

e. Object structure

```
public class ShoppingCart {
	// normal shopping cart stuff
	private List<OrderItem> items;

	public double calculateShippingFee() {
		// create a visitor
		ShippingFeeVisitor visitor = new ShippingFeeVisitor();
		// iterate through all items
		for (OrderItem item : items) {
			item.accept(visitor);
		}
		double postage = visitor.getTotalShippingFee();
		return postage;
	}
}
```

Lab 11-1

In this lab, you should use the visitor pattern in your design for displaying a mind map. Fig. 1 is a class diagram for your reference. (We are not implementing the SaveNodeVisitor).

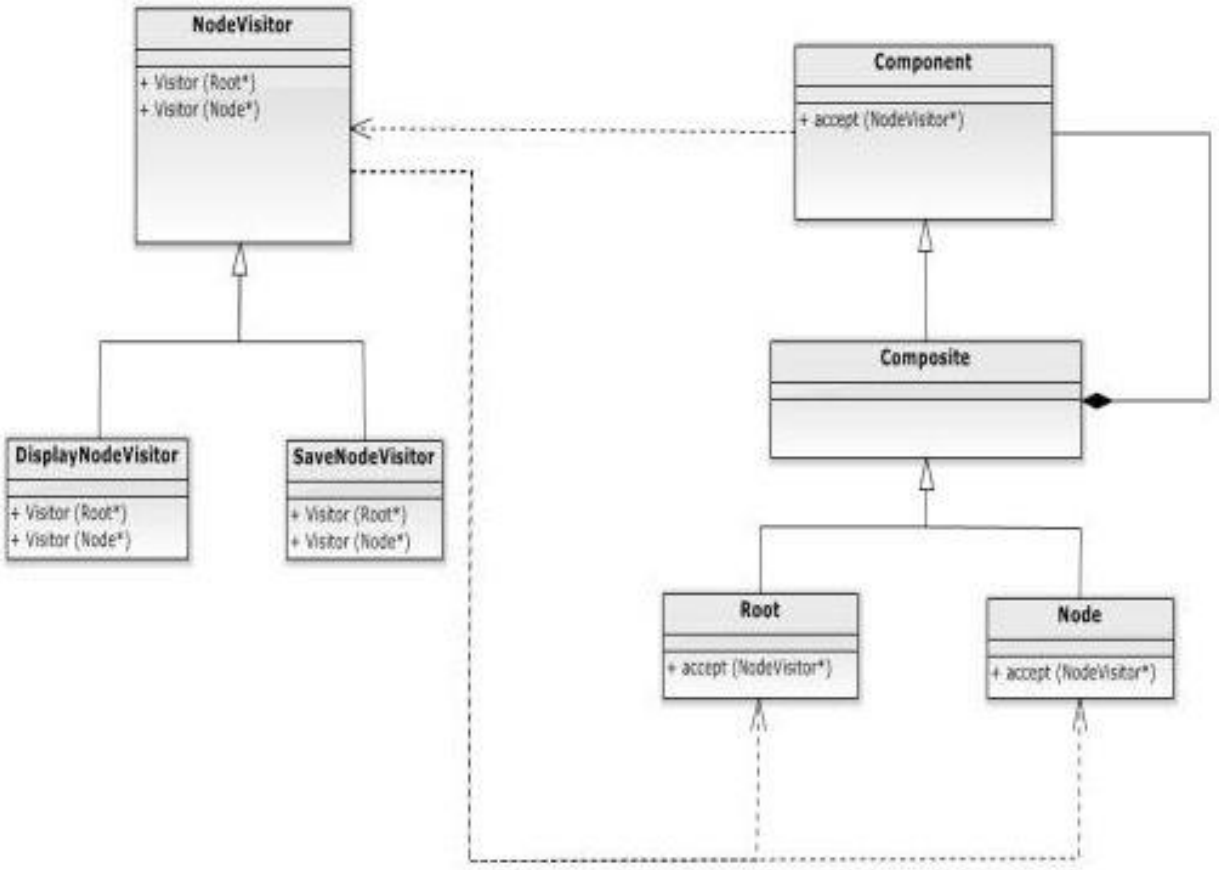


Fig 1: Applying Visitor pattern in *MindMap*

A two-sided Mind Map

The root node of a mind map has two sides for arranging its children nodes. As the example shown in Fig 2, the root node (Topic) is placed at the center.

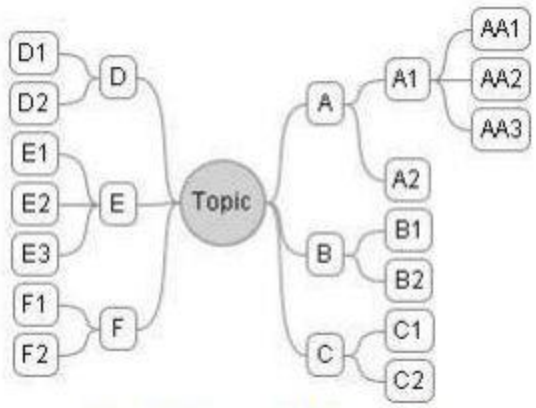


Fig 2 A two-side mind map example

The Component class has a getSide() method to indicate the side of the node. The return value can be RIGHT, LEFT, or NONE. (The side-value of the root node is always NONE). The Component class has another method, getName(), that returns “A”, “A1”, or “AA1” for the nodes shown on the top right of Fig. 2. For now, we suppose a mind map has been created accordingly as we see in Fig 2.

If a node is inserted as a child of another node, its side value is always the same as its parent node, except for the root node. (We do not care how to decide on the side value of direct children of the root for this lab.)

Let’s implement a DisplayNodeVisitor to print the following information for the mind map you see in Fig 2. –

Topic
Right nodes:
6 A nodes
3 B nodes
3 C nodes
Left nodes:
3 D nodes
4 E nodes
3 F nodes

Interpreter

1. Intent

   Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

2. Motivation

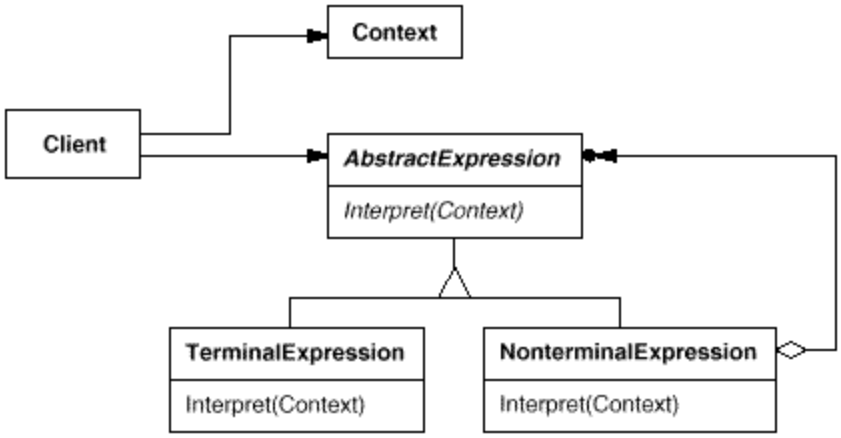
   We need to create a rule-based language to represent expressions or statements or sentences, and we want to describe how to define the grammar (the rules) and how to interpret the language.

3. Applicability

   Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when:

   a. the grammar is simple - the rules can be represented with a limited number of expressions.
   b. efficiency is not a critical concern - all you want to achieve is to interpret the expressions. How efficiently it is done does not matter too much.

4. Structure



5. Participants

a. AbstractExpression
   - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

b. TerminalExpression
   - implements an Interpret operation associated with terminal symbols in the grammar.
   - an instance is required for every terminal symbol in a sentence.

c. NonterminalExpression
   - one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
   - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
   - implements an ‘interpret’ operation for nonterminal symbols in the grammar. ‘interpret’ often calls itself recursively on the variables representing R1 through Rn.

d. Context
   - contains information that's global to the interpreter.

e. Client
   - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
   - invokes the ‘interpret’ operation.

6. Consequences

The Interpreter pattern has the following benefits and liabilities:

a. It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar.
b. Implementing the grammar is easy (if it is not complex by itself!).
c. Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar. But when the grammar is very complex, other techniques such as parser or compiler generators are more appropriate. (compiler vs interpreter).

7. How to implement the Interpreter pattern?

Let’s use a Roman numerals converter as an example.

Symbol	Latin name	Value
I	*unum*	1
V	*quinque*	5
X	*decem*	10
L	*quīnquāgintā*	50
C	*centum*	100
D	*quingenti*	500
M	*mille*	1,000

I = 1	V = 5	X = 10	L =50	C=100	M=1000

1 I	2 II	3 III	4 IV	5 V	6 VI	7 VII	8 VIII	9 IX	10 X
11 XI	12 XII	13 XIII	14 XIV	15 XV	16 XVI	17 XVII	18 XVIII	19 XIX	20 XX
21 XXI	22 XXII	23 XXIII	24 XXIV	25 XXV	26 XXVI	27 XXVII	28 XXVIII	29 XXIX	30 XXX
31 XXXI	32 XXXII	33 XXXIII	34 XXXIV	35 XXXV	36 XXXVI	37 XXXVII	38 XXXVIII	39 XXXIX	40 XL
41 XLI	42 XLII	43 XLIII	44 XLIV	45 XLIV	46 XLVI	47 XLVII	48 XLVIII	49 XLIX	50 L
51 LI	52 LII	53 LIII	54 LIV	55 LV	56 LVI	57 LVII	58 LVIII	58 LIX	60 LX
61 LXI	62 LXII	63 LXIII	64 LXIV	65 LXV	66 LXVI	67 LXVII	68 LXVIII	69 LXIX	70 LXX
71 LXXI	72 LXXII	73 LXXIII	74 LXXIV	75 LXXV	76 LXXVI	77 LXXVII	78 LXXVIII	79 LXXIX	80 LXXX
81 LXXXI	82 LXXXII	83 LXXXIII	84 LXXXIV	85 LXXXV	86 LXXXVI	87 LXXXVII	88 LXXXVIII	89 LXXXIX	90 XC
91 XCI	92 XCII	93 XCIII	94 XCIV	95 XCV	96 XCVI	97 XCVII	98 XCVIII	99 XCIX	100 C

ADRIANBRUCE.COM

a. Context class

```
public class Context {

	List<Expression> tree = new ArrayList<Expression>();
	String input;
	int output;

	public Context(){
		// Build the 'parse tree'
		tree.add(new ThousandExpression());
		tree.add(new HundredExpression());
		tree.add(new TenExpression());
		tree.add(new OneExpression());
	}

	public Context(String input) {
		this();
		this.input = input;
	}

	public List<Expression> getParseTree() {
		return tree;
	}
}
```

b. Expression and its sub-classes

```
public abstract class Expression {

	public void interpret(Context context) {
		if (context.input.length() == 0)
			return;

		if (context.input.startsWith(nine())) {
			context.output = context.output + (9 * multiplier());
			context.input = context.input.substring(2);
```

```
            } else if (context.input.startsWith(four())) {
                  context.output = context.output + (4 * multiplier());
                  context.input = context.input.substring(2);
            } else if (context.input.startsWith(five())) {
                  context.output = context.output + (5 * multiplier());
                  context.input = context.input.substring(1);
            }

            while (context.input.startsWith(one())) {
                  context.output = context.output + (1 * multiplier());
                  context.input = context.input.substring(1);
            }
      }

      public String toString(){
            return this.getClass().getSimpleName();
      }

      public abstract String one();
      public abstract String four();
      public abstract String five();
      public abstract String nine();
      public abstract int multiplier();

}

public class ThousandExpression  extends Expression{

    public String one() { return "M"; }
    public String four(){ return " "; }
    public String five(){ return " "; }
    public String nine(){ return " "; }
    public int multiplier() { return 1000; }
}

public class HundredExpression extends Expression{
    public  String one() { return "C"; }
    public  String four(){ return "CD"; }
    public  String five(){ return "D"; }
    public  String nine(){ return "CM"; }
    public  int multiplier() { return 100; }
}

public class TenExpression  extends Expression{
    public String one() { return "X"; }
    public String four(){ return "XL"; }
    public String five(){ return "L"; }
    public String nine(){ return "XC"; }
    public int multiplier() { return 10; }
}

public class OneExpression  extends Expression{
    public String one() { return "I"; }
    public String four(){ return "IV"; }
    public String five(){ return "V"; }
    public String nine(){ return "IX"; }
    public int multiplier() { return 1; }
```

```
}
```

c. Test it with a client program

```
public class Client {

	public static void main(String[] args) {

		String roman = "MDCCCXIII";
		//Context holds a reference to the input Roman numeral and
intermediate results
		Context context = new Context(roman);

		//Get the parse tree from Context
		List tree = context.getParseTree();

		for ( Iterator it = tree.iterator(); it.hasNext(); ) {
			Expression exp = (Expression) it.next();
			System.out.println("Parsing with exp: "+exp);
			exp.interpret(context);
		}
		System.out.println("---------------------------------");
		System.out.println(roman + " = " + context.output);
	}
}
```

Lab 11-2

1. Use the Interpreter Pattern to interpret/evaluate prefix expressions like what we see below:

```
public static void main(String[] args) {
	String tokenString = "- + 10 5 - 8 2";
	Expression expression = new Expression(tokenString);
	System.out.println(expression.interpret()); // (10 + 5) - (8 - 2)
	= 9
}
```

2. Use a different approach to solve the Roman Numerals problem with Terminal Expression and Non-terminal Expression.

(Question 2 is optional)