CS 4321 Project 1: Visitor Pattern

This document tries to simplify the visitor pattern which will be extensively used in CS 4321 projects.

Why Visitor Pattern?

Visitor Pattern has two important advantages:

1) Directs code to proper function based on object type
2) Makes your code easily extendable for adding new functionality.

Let us not go into the design pattern definition which you find everywhere. Let's try to look at visitor pattern with examples from our project here.

1) Directing code to proper operation

Visitor pattern also does a good job of directing the code to proper function based on object type. One would usually do this "directing to a proper function" using **instanceof** operation in a series of if statements. This is not a good object oriented programming practice. This is how it would look like:

```
ListNode node=getResult();

if(node instanceof AdditionListNode){
        processAdditionNode(node);
}

if(node instanceof SubtractionListNode){
        processSubtractionNode(node);
}
if(node instanceof MultiplicationListNode){
        processMultiplicationNode(node);
}

if(node instanceof DivisionListNode){
        processDivisionNode(node);
}

if(node instanceof NumberListNode){
        processNumberNode(node);
}

if(node instanceof UnaryMinusListNode){
        processMinusNode(node);
}
```

It doesn't end here.

Suppose we have decided it's an AdditionListNode, and you want to further direct it to perform specific operation:

```
String operation=getOperation();

void processAddtionNode(ListNode node){
        if(operation.equals("PRINT")){
                printNode(node);
        }
        if(operation.equals("EVALUATEPOSTFIX"){
                evaluatePostfix(node);
        }
        if(operation.equals("EVALUATEPREFIX"){
                evaluatePrefix(node);
        {
}
```

You can already see how messy and inflexible it gets.

2) Separating logic from objects (Extensibility)

Another important reason to use visitor pattern is separating the operation logic from the object definitions.

Say we have a ListNode. A ListNode can be AdditionListNode, SubtractionListNode, MultiplicationListNode and so on. Say we want to add functionality to print a list node. We would have to add some printNode() implementation to all the ListNode types classes. printNode() will have different implementation for different types of ListNodes. For example, we would print "+ " for a AdditionListNode, "* " for a MultiplicationListNode and so on.

Now, suppose we plan to add another functionality to Evaluate a ListNode in prefix form, say EvaluatePrefixListNode, you would have to add evaluatePrefix() to all your object classes. This gets highly complex and inflexible. Whenever you plan to add some new functionality you have to change large number of object files. Here you have just 6 object types for ListNode. What if you had more than 25? It gets difficult to handle.

Visitor Pattern Basics

This gets somewhat technical, but we will relate it to our project as we go. There are four types of classes to keep in mind for a visitor pattern:

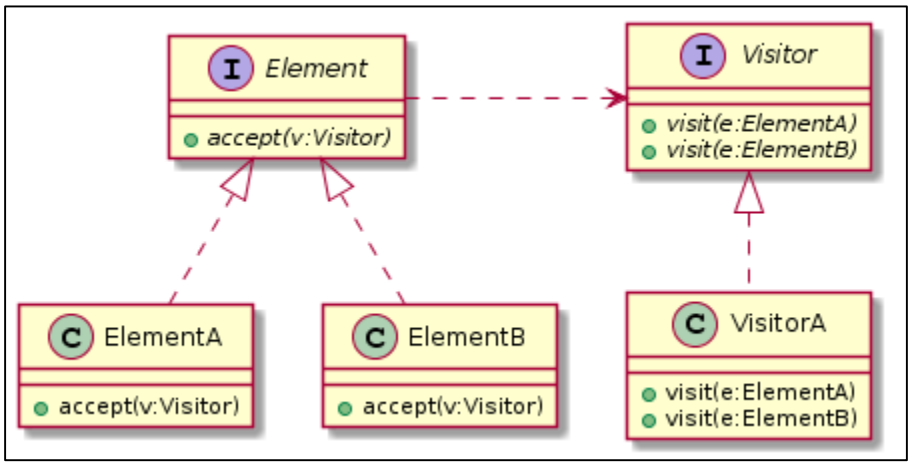
Element: An interface/abstract class which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object. **In our case, this is ListNode.**

ConcreteElement: Those classes which implement the Element interface/abstract class and define the accept operation. The visitor object is passed to this object using the accept operation. **In our case, these are the different types of ListNodes, like AdditionListNode, SubtractionListNode, and so on.**

Visitor: This is an interface/abstract class used to declare the visit operations for all the types of element classes. **In our case, this is ListVisitor. And all the types of "element" classes are the different types of ListNodes.**

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. **In our case, these are different operations like PrintListVisitor, EvaluatePrefixListVisitor, EvaluatePostfixListVisitor.**

This is the simplest image we could find for a visitor pattern:



ElementA, ElementB are ConcreteElements, VisitorA is a ConcreteVisitor.

Note that some websites also call Element as Visitable.

Element and Visitor can be interfaces or abstract classes.

How does visitor pattern solve the problems?

Directing to proper operation:

Everything lies in the implementation of the accept() function.

Here is a sample implementation of accept function we have:

```
class AdditionListNode{
        void accept(ListVisitor visitor){
                visitor.visit(this);
        }
}
```

This function would be present in all types of ListNodes. Consider the way of using this:

```
ListNode node=getResult();
//If you want to print this node:
PrintListVisitor visitor=new PrintListVisitor();
node.accept(visitor);
```

Here, node.accept() will direct to the concrete element (this solves the directing to proper type problem, you don't have to use those instanceof any more). And since we know we would want to print this node, we pass that type of visitor accordingly.

Further, doing visitor.visit(this), now since we passed visitor as a PrintListVisitor, this would call some visit() from the PrintListVisitor. Since we are writing this in AdditionListNode class, "this" keyword would call visit(AdditionListNode node) in PrintListVisitor.

This way, we gracefully direct our code to print an addition node.

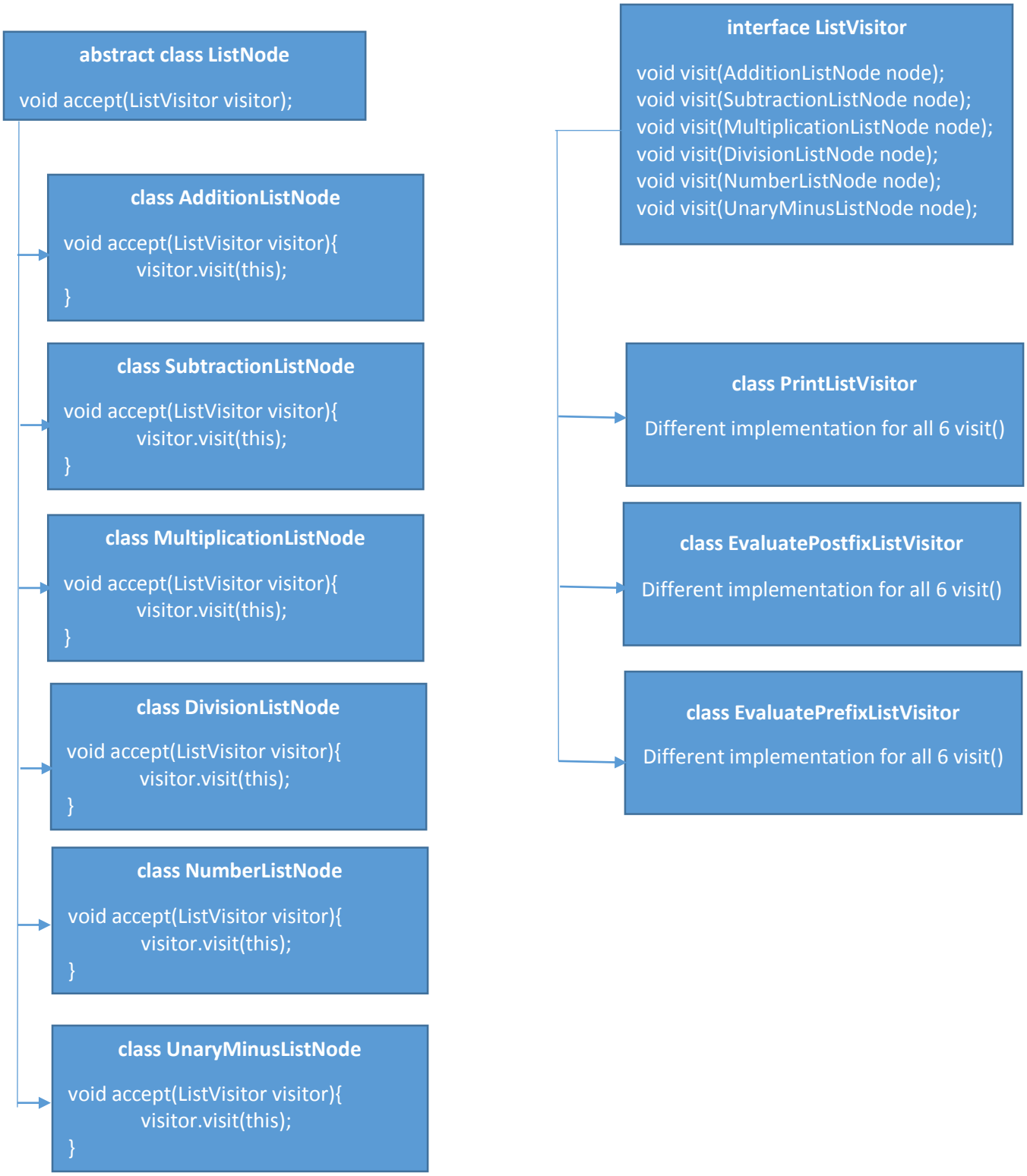
Adding new operations:

Problem of adding new operations, without changing the object definitions is solved by the visitor pattern too. Now, whenever you want to add a new operation, you just write a new class implementing your Visitor interface. Say you want to add EvaluateListVisitor, you would do that writing a class EvaluateListVisitor implementing each of the visit functions in the ListVisitor. Nothing else changes. That is extensibility at its best.

How we use it in our projects?

Here is the class hierarchy for the visitor patterns used for list and tree nodes. Understand the similarity between the generic visitor pattern figure above and the way we use it. These diagrams are just for representation, they are not UML diagrams.

For list nodes:



For tree nodes:

References:

1) Visitor Pattern class types definition from: http://howtodoinjava.com/2013/09/08/visitor-design-pattern-example-tutorial/
2) Visitor Pattern generic figure from: http://blog.frankel.ch/wp-content/resources/the-visitor-design-pattern/visitor.png