Visitor

Linda Marshall

Department of Computer Science
University of Pretoria

2 November 2021

So far, the patterns using delegation have used the dynamic (run-time) type of a single object when calling functions. This is known as *single dispatch*.

Single dispatch allows for the function to be called based on the type of the argument.

Consider a function defined in `ClassA` with the signature:

```
void ClassA::theFunction();
```

This could be written as

```
void theFunction(ClassA);
```

`theFunction` dispatches on the type associated with the type of the class.

In the Visitor, the types of two objects involved in the function call dictates which function is called. This mechanism is referred to as *double dispatch*.

ClassA also has a function

`void ClassA::anotherFunction(ClassB);`

The rewrite results in

`void anotherFunction(ClassA,ClassB);`

`anotherFunction` therefore dispatches on the type of `ClassA` and the type of `ClassB`

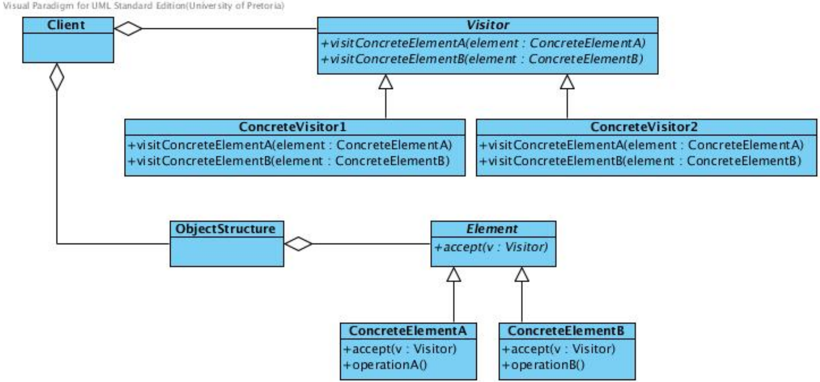
Name and Classification: Visitor, Behavioural delegation

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." (GoF:331)

"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." (GoF:331)


Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
Visitor
+visitConcreteElementA(element : ConcreteElementA)
+visitConcreteElementB(element : ConcreteElementB)
ConcreteVisitor1
+visitConcreteElementA(element : ConcreteElementA)
+visitConcreteElementB(element : ConcreteElementB)
ConcreteVisitor2
+visitConcreteElementA(element : ConcreteElementA)
+visitConcreteElementB(element : ConcreteElementB)
ObjectStructure
Element
+accept(v : Visitor)
ConcreteElementA
+accept(v : Visitor)
+operationA()
ConcreteElementB
+accept(v : Visitor)
+operationB()

This pattern separates the behaviour of the elements in an aggregate (object structure) from the state of these elements to simplify the maintenance when the behaviour of these elements has to be changed or extended. The application of this pattern complicates the maintenance of the aggregate itself. It is difficult to add classes to the aggregate. Thus, this pattern is more applicable in a system with changing processing needs and a stable internal structure of elements. That is, a system where you will seldom add new classes but have the need to often add new functions to some derived classes in an aggregate and consequently new virtual functions to existing interfaces to the aggregates.

- **Visitor**:
  - Each class of ConcreteElement has a `visit()` operation declared for it.
  - The operation's signature identifies the class that sends the `visit()` request to the visitor.
  - The particular class is then accessed through the interface defined for it.

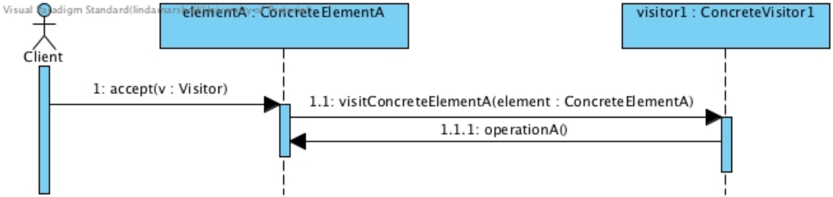
- **ConcreteVisitor**:
  - Implements the operations defined by visitor.
  - May store information about objects that are visited.

```
return_type ConcreteVisitor1::visit(ConcreteElementA element) {
  element.operationA();
}
```

- **Element**: Defines an `accept()` operation that takes an object of Visitor as a parameter.
- **ConcreteElement**: Implements the `accept()` operation that takes an object of Visitor as a parameter.

```
return_type ConcreteElementA::accept(Visitor v) {
  v.visit(this);
}
```

- **ObjectStructure**:
  - Has a highlevel interface that allows the Visitor to access and traverse its elements.
  - This structure may be a Composite or a collection such as an array, list or a set.

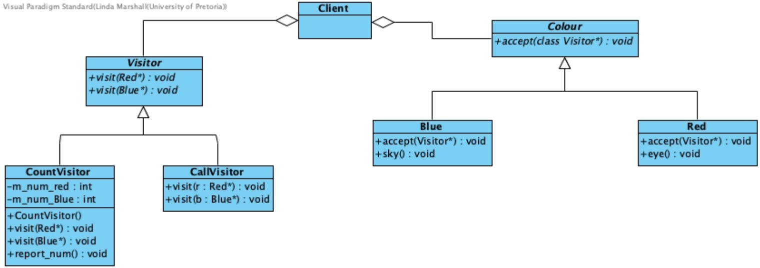

Client
elementA : ConcreteElementA
visitor1 : ConcreteVisitor1
1: accept(v : Visitor)
1.1: visitConcreteElementA(element : ConcreteElementA)
1.1.1: operationA()

- **Iterator** and Visitor have similar intents. Visitor, however, is more general than Iterator. An Iterator is restricted to operations on elements of the same kind while Visitor can operate on elements of different types.

- **Composite** is supportive to the Visitor. Visitors can apply an operation over an object structure defined by the Composite pattern.
- **Interpreter**, the Visitor pattern may be applied to do the interpretation.

- **Abstract Factory** and Visitor has similar structure. Abstract factory applies the structure to create families of objects while Vistor applies this structure to perform a group of related operations.

- **Bridge** and Visitor separates state and behaviour of objects. Bridge applies single dispatch while Visitor applies double dispatch.



Taken from `https://sourcemaking.com/design_patterns/visitor/cpp/1`

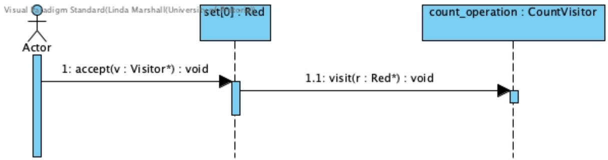
```
int main() {
    Colour* set[] = {
        new Red, new Blue, new Blue, new Red, new Red, 0
    };
    CountVisitor count_operation;
    CallVisitor call_operation;
    for (int i = 0; set[i]; i++) {
        set[i]->accept(&count_operation);
        set[i]->accept(&call_operation);
    }
    count_operation.report_num();
}
```

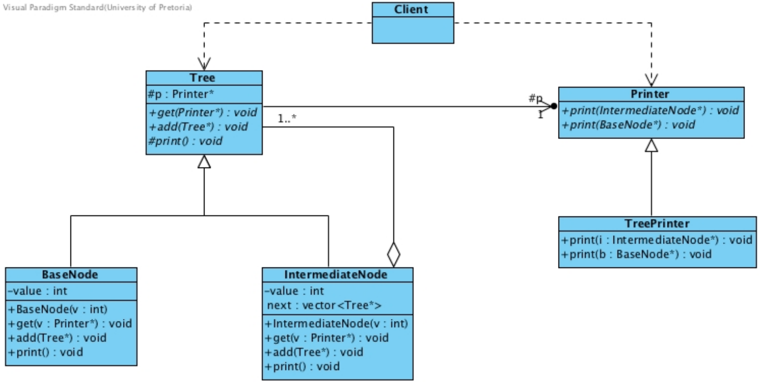
```
// First functions called...

// Call with set[i] as Red and
// count_operation an instance of CountVisitor
set[i]->accept(&count_operation);

// First called with v an instance of CountVisitor
void Red::accept(Visitor *v) {
    v->visit(this); // Dispatch on v and this
                    // that is, CountVisitor and Red
}

void CountVisitor::visit(Red*) {
    ++m_num_red;
}
```




Visual Paradigm Standard(University of Pretoria)
Client
Tree
#p : Printer*
+get(Printer*) : void
+add(Tree*) : void
#print() : void
1..*
#p
1
Printer
+print(IntermediateNode*) : void
+print(BaseNode*) : void
TreePrinter
+print(i : IntermediateNode*) : void
+print(b : BaseNode*) : void
BaseNode
-value : int
+BaseNode(v : int)
+get(v : Printer*) : void
+add(Tree*) : void
+print() : void
IntermediateNode
-value : int
next : vector<Tree*>
+IntermediateNode(v : int)
+get(v : Printer*) : void
+add(Tree*) : void
+print() : void

The Aggregate hierarchy

```
class Tree {
    public:
      virtual void get(Printer*) = 0;
      virtual void add(Tree*) = 0;
    protected:
      virtual void print() = 0;
      Printer* p;
};
```

```
class BaseNode : public Tree {
    public:
        BaseNode(int v) : value(v) {};
        virtual void get(Printer* v) {
          p = v;
          p->print(this);  // call to visitor visit
        };
        virtual void add(Tree*) {};
        virtual void print() {
          cout << " " << value << " ";
        };
    private:
        int value;
};
```

```
class IntermediateNode : public Tree {
    public:
        IntermediateNode(int v) : value(v) {};
        virtual void get(Printer* v) {
           p = v; p->print(this); // call to visitor visit
        };
        virtual void add(Tree*);
        virtual void print();
    private:
        int value;
        vector<Tree*> next;
};
```

```
void IntermediateNode::add(Tree* t){
    next.push_back(t);
}

void IntermediateNode::print(){
    cout << "-" << value << "[";
    vector<Tree*>:: iterator it;
    for (it = next.begin(); it != next.end(); ++it)
      (*it)->get(p);
    cout << "]";
}
```

The Visitor hierarchy

```
class Printer {
    public:
        virtual void print(IntermediateNode*) = 0;
        virtual void print(BaseNode*) = 0;
};
```

```
class TreePrinter : public Printer {
    public:
        void print(IntermediateNode* i) {
          cout<<"i* "; i->print();    // call to aggregate accept
        };
        void print(BaseNode* b) {
          cout<<"b* "; b->print();  // call to aggregate accept
        };
};
```

```
int main(){

    Tree* t = new IntermediateNode(10);
    Tree* b = new BaseNode(5);
    t->add(new BaseNode(5));
    Tree* l1 = new IntermediateNode(20);
    l1->add(new BaseNode(67));
    l1->add(new BaseNode(20));
    t->add( l1 );

    Printer* p = new TreePrinter();
    t->get(p);
    cout<<endl;
    l1->get(p);
    cout<<endl;
    // Deallocate the memory using a similar
    // technique as you used for the Tree example
    return 0;
}
```

Output from the program:

```
i*  -10[b*   5  i*  -20[b*   67  b*   20  ]]
i*  -20[b*   67  b*   20  ]
```

To define another visitor for the aegrogate is easy. The class representing this visitor must inherit from Printer.

One drawback of the pattern is the tight coupling between the visitor and the elements.