A Pattern Language To Visitors*

Yun Mai and Michel de Champlain
Department of Electrical and Computer Engineering
Concordia University
{*y_mai, michel*}*@ece.concordia.ca*

Abstract

Since Gamma et al. first published the Visitor design pattern in 1995, there have been proposed several variations in the design pattern literature. This paper attempts to classify and organize these variations. It presents a pattern language to Visitors to assist the application developer to choose the right Visitor pattern that best suites the intended purpose by enumerating all important forces and consequences for each variation.

Introduction

There are a number of approaches that have been proposed to handle a class of complex and related objects and their related operations. The baseline approach is the VISITOR design pattern proposed by Gamma et al [1]. But this approach gains flexibility in some aspects and loses it in others. Variations are proposed aiming for some improvement to the baseline approach.

This paper classifies the major variations of the VISITOR pattern and organizes them into a pattern language. It first provides general descriptions about the basic concepts of VISITORS, a road map to VISITORS, how to use these patterns and a comparison on these VISITORS. The following sections describe each VISITOR in details.

The target audience is the application developer who is the implementor and direct beneficiary of the VISITORS.

Terms and Concepts

Visitor

A visitor implements behaviors for traversing a set of element objects and assigning responsibilities to these elements. It encapsulates operation to be performed over these elements and wraps them in a class separating from the elements themselves. In general, there are two hierarchies to be defined when a visitor is implemented. One is the element hierarchy representing the objects that will be visited. The other is the visitor hierarchy representing operations to be performed on the elements. The use of visitor lets the developer easily change the behaviors that would otherwise be distributed across classes without modifying these classes.

Element Adder

An Element Adder is a kind of solution that can easily add new elements to an element hierarchy without modifying the existing programming.

Operation Adder

An Operation Adder is a kind of solution that can easily add new operations over an element hierarchy without modifying the existing programming.

Cyclic Dependency

A component of a system is said to depend on another component if the correctness of the first component's behavior requires the correct operation of the second component. A dependency relationship is said to be acyclic if it forms part of a tree. That is the set of possible dependencies in a system are considered to form an acyclic graph. It is possible, however, for a dependency relationship to cycle back upon itself. A cyclic dependency relationship is one that cannot be described as part of a tree, but rather must be described as part of a directed cyclic graph.

*Copyright © 2001, Yun Mai and Michel de Champlain. Permission is granted to copy for the PLoP 2001 conference. All other rights reserved.

A Road Map to Visitors

This section provides a road map for the VISITORS. It also gives a short description on how to use these patterns and a comparison on these VISITORS and a guidance that assists the application developer to make the best choice.

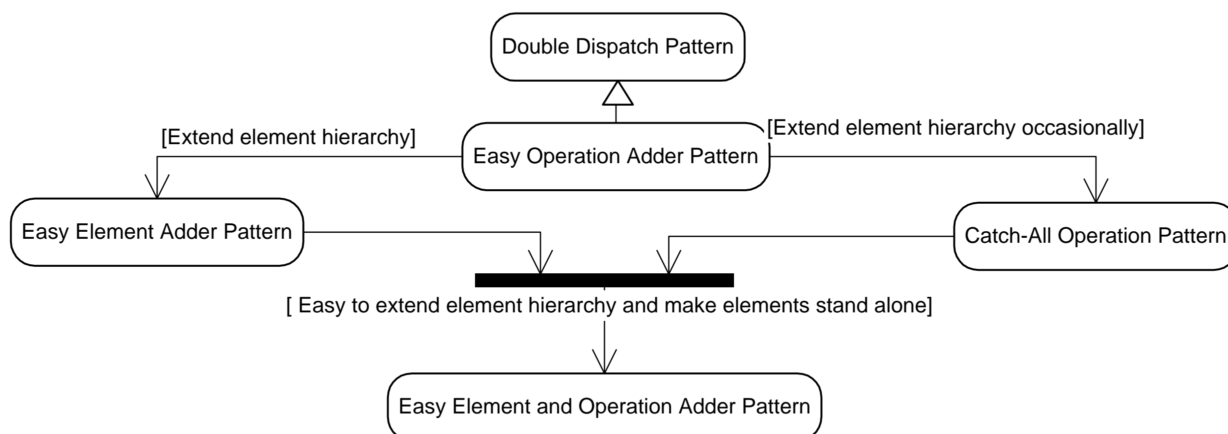

Figure 1: Road Map for the Visitor Patterns

Figure 1 shows the road map for major variations of the VISITOR patterns. The topmost pattern in Figure 1 is the DOUBLE DISPATCH pattern [1, 11, 3] and the following is the EASY OPERATION ADDER pattern [1] introduced by the GoF book. The DOUBLE DISPATCH pattern is a kind of generalization of the EASY OPERATION ADDER pattern and the EASY OPERATION ADDER pattern is the baseline approach of all VISITORS. The followings are two VISITOR patterns: the CATCH-ALL OPERATION pattern [7] introduced by John Vlissides and the EASY ELEMENT ADDER pattern [2] developed by Etienne Gagnon in his Sablecc Compiler framework. Both the CATCH-ALL OPERATION pattern and the EASY ELEMENT ADDER pattern use some kinds of run-time type checking and down-casting to enable the addition of the new elements to the element hierarchy easier. But they do not break the cyclic dependency between the visitor hierarchy and the element hierarchy, so the two hierarchies have to know each other. The last approach named EASY ELEMENT AND OPERATION ADDER pattern [12], presented by the authors, is characterized by the use of the reflection technique. It not only makes the addition of both new operations and new element classes easy, but also breaks the cyclic dependency between the visitors and the elements so that the elements have no knowledge about the visitors. Furthermore, it hides the implementation details of the operations from the clients and thus simplifies the usage of the visitors. But this VISITOR pattern achieves its simplicity in the expense of performance.

There is not a right approach for the VISITORS. The vitality of the VISITORS is that it provides a choice that can robustly apply in certain circumstances.

Pattern Language Summary

This pattern language includes five patterns. They are described in the following order:

1. The DOUBLE DISPATCH pattern
   This pattern [1, 11, 3] is not a kind of VISITOR but a generalization of the most VISITORS. It exists in the context that the execution of an operation depends on the kind of request and the types of two receivers, the dispatcher and the element. The DOUBLE DISPATCH pattern lets the dispatcher request different operations on each class of element without modifying the existing classes.

2. The EASY OPERATION ADDER Pattern
   This pattern was first introduced in the Design Pattern book [1]. It separates the unrelated operations from the element hierarchy and wraps these operations into another class hierarchy. By using the EASY OPERATION ADDER pattern, you can define new operations over the elements by simply adding a new visitor. So the existing elements can remain unchanged. But adding new element classes is hard because all related visitor classes have to be redefined.

3. The CATCH-ALL OPERATION Pattern
   This pattern [7] is an improvement to the EASY OPERATION ADDER pattern for occasional extension of the element hierarchy. It defines a catch-all operation in the visitors and allows new element classes to be

occasionally added without modifying the existing visitor interfaces. The cyclic dependencies still exist between the visitors and the elements.

4. The Easy Element Adder Pattern
   This pattern [2] is also an improvement to the Easy Operation Adder pattern in the situation where the element hierarchy is changed often. It allows new element classes to be added without any limit. But this approach introduces a deeper binding between the element hierarchy and the visitor hierarchy. It can be used where reusability of the element hierarchy is not a major concern to the designer.

5. The Easy Element and Operation Adder Pattern
   This pattern is proposed by the authors [12]. It takes advantages of the reflection technique to simplify its structure and implementation. The cyclic dependency between the visitors and the elements is broken and the implementation details of the operations are encapsulated. Both addition of new operations and new element classes become easy.

How to Use These Patterns

The reader who searches for a solution to a Visitor problem may resort to Table 1 and Table 2. If a pattern is of particular interest, Context, Forces, Rationale and Resulting Context sections can be examined to determine the applicability of this pattern in the target circumstance. Once a pattern is chosen, the Solution and Code Samples sections can help the reader to implement the chosen pattern in the target system. Table 1 summarizes the pairs of problem and solution for the patterns.

Table 1: Problem/Solution Summaries

Problem	Solution	Pattern Name
How to accept additional types of arguments in a method without modifying the existing code of the method?	Shift responsibility from the class that performs the operation to a class hierarchy, where any element may appear as an argument to the operation.	Double Dispatch
How to define new operations on classes over time without changing these class interfaces?	Package the operations in a separate hierarchy and define the *accept* method in the elements to perform dispatch.	Easy Operation Adder
How to prevent the modification of the existing visitor classes while allowing new element classes to be added occasionally?	Define a catch-all operation in the **Visitor** class and override it in the concrete visitors to perform the run-time type checking.	Catch-All Operation
How to easily add new element or new operations without modifying the existing interfaces?	Redefine a complete interface for all visitors and perform a down-casting in the elements.	Easy Element Adder
How to enable both addition of new operations and new element classes easy, while at the same time, make the element hierarchy stand alone?	Remove the *accept* from the elements and let the visitor to elegantly handle the dispatch action based on reflection.	Easy Element and Operation Adder

A Comparison on Visitors

Table 2 compares the Visitor patterns based on easy addition of new operations, easy addition of new elements, coupling between the visitor hierarchy and the element hierarchy, and the run-time efficiency:

Table 2: Comparison on Visitor Patterns

Pattern Name	Add New Operation	Add New Element	Coupling	Efficiency
Easy Operation Adder	Easy	Hard	Tight	High
Catch-All Operation	Easy	Easy for small extension	Tight	High
Easy Element Adder	Easy	Easy	Tight	High
Easy Element and Operation Adder	Easy	Easy	Loose	Low

- All VISITOR patterns can easily add new operations because they all define two class hierarchies.
- The EASY ELEMENT ADDER pattern and the EASY ELEMENT AND OPERATION ADDER pattern allow new element classes to be easily added. The latter is superior to the previous due to the simplicity of its implementation. The CATCH-ALL OPERATION pattern allow a small number of element classes to be added, otherwise, the programming style will degrade into tag-and-case statements. Use the EASY OPERATION ADDER patterns to extend the element classes is hard because all related visitors need to be modified to incorporate new element types.
- The EASY ELEMENT AND OPERATION ADDER patterns breaks the cyclic dependency between the visitors and the elements so that they have loose coupling. But other patterns do not.
- Due to the use of reflection technique, the run-time efficiency of the EASY ELEMENT AND OPERATION ADDER pattern is low.

There is not a right VISITOR pattern all the time. All patterns presented in this paper are selectively applied in certain circumstances. From Table 2, we can also see that a system's extensibility is always traded with its efficiency. For example, the EASY ELEMENT AND OPERATION ADDER pattern has a very superior structure and good extensibility, but its efficiency is very slow.

A Simple Example

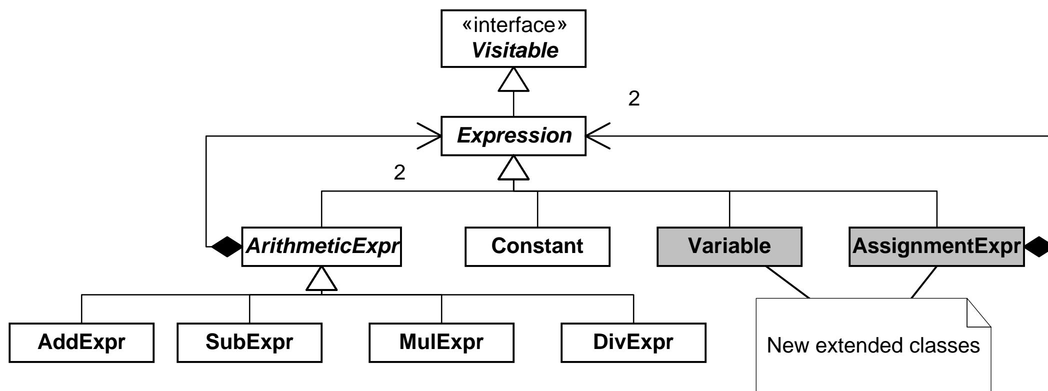

Figure 2: An Expression Hierarchy

Because all patterns presented in this paper are closely related, we'll put all of them together to highlight their similarities and differences. We'll use a common example, a simple expression example, to illustrate their implementations. This simple expression supports arithmetic expressions such as addition, subtraction, multiplication, and division for constants. Figure 2 shows the language structure hierarchy for this expression example. The expression structure hierarchy is organized as a composite structure and can be implemented by a Composite design pattern [1]. The abstract syntax tree therefore is represented as a composite object, which is recursively constructed with the instances of the node classes in the expression structure during the parsing. The code generation or calculation process then performs the code generation operation or calculation over this abstract syntax tree.

Basically, there are two kinds of potential extensions to the above example: one is the changing of the expression structure, the other is generating different code formats or switching it to calculation. For example, the expression structure can be extended with supports of variable and assignment expression that will be used to assign the value or expression to the variable. As shown in figure 2, they are represented as two extended classes in gray. On the other hand, the code generation may require target code to be generated in different code format according to the design requirements or calculate the expression result directly. It may also require easy switching from one format to another and easy addition of new kind of output code format. For instance, the code generation for above example may support the two different virtual machines, VM1 and VM2.

Double Dispatch

Alias

Visitor Essence.

Context

The behavior of a method depends not only on the class that implements the method but also on the classes of the method's arguments as well.

Problem

How to accept additional types of arguments in a method without modifying the existing code of the method?

Forces

- Using a case-like statement to perform type-checking on the types of arguments makes the system difficult to extend.
- Determining the argument types at run time rather than at compiler time makes the system more flexible.

Solution

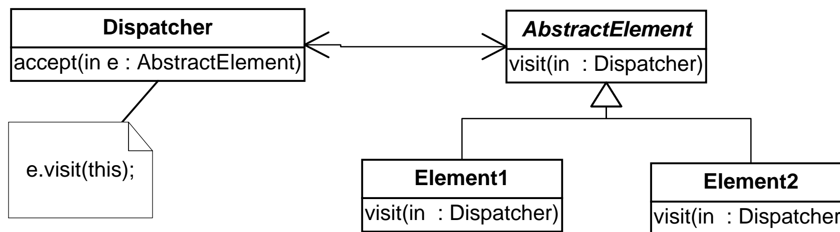

Figure 3: Structure for the Double Dispatch Pattern

Shift responsibility from the class that performs the operation to a class hierarchy, where any element may appear as an argument to the operation. All these elements share an identical operation interface.

Figure 3 shows the structure of the Double Dispatch pattern. There are two groups defined in the design. One is the class **Dispatcher**. The other is the element hierarchy. The **Dispatcher** defines a method *accept* that can accept a concrete element instance. The implementation of the *accept* method calls the related method *visit* in the element hierarchy. The method *accept* is a double-dispatch operation. It depends on both the types of the **Dispatch** and the concrete element.

Rationale

Treating different types of receivers the same and letting the program determine the concrete types at run time help to avoid degrading the program into a long tag-and-case statement in order to distinguish different types of receivers. For instance, in Figure 3, we assume that the class **Element2** is a newly added concrete element class. It defines a *visit* method to implement the corresponding operation that should be executed during the *accept* method's execution. When the *accept* method is called on an instance of **Dispatcher** with a given instance of **Element2** as argument, it invokes the polymorphic *visit* method on the instance of class **Element2**(the argument of the method *accept*) and supplies itself as a parameter. The *visit* method of class **Element2** can call back to any method defined in the **Dispatcher**. As a consequence, any concrete element type can be accepted by the **Dispatcher** without modifying the invoking method defined in the **Dispatcher**.

Resulting Context

- Accept additional types of arguments in a method without modifying the existing code of the method. New type argument can be added in the element hierarchy by inheritance and the method can accept the instance of this new type and dispatch the operation to this instance by polymorphism.

- There is no need to write a case-like statement in the invoking method (eg. *accept*) to perform type-checking on the types of arguments and add additional type-checking statements once a new parameter type is added. Instead, the argument types can be determined at run time dynamically. With Double Dispatch, the parameter type is determined at run-time, the behavior of the invoking method is hidden at compile time.
- The code is distributed in several classes so that locating and understanding the intending behavior becomes hard.

Code Samples

We focus on the **Constant** class in the simple expression example and show how to implement the Double Dispatch pattern, The following shows the definitions of the related classed and the implementation in Java.

```
class Constant{
    public void accept(CodeGen gen) {
        gen.visit(this);
    }
}

abstract class CodeGen {
    abstract public void visit(Constant expr);
}

class VM1CodeGen extends CodeGen {
    public void visit(Constant expr) {
        // code generation for virtual machine VM1 ;
    }
}

class VM2CodeGen extends CodeGen {
    public void visit(Constant expr) {
        // code generation for virtual machine VM2 ;
    }
}
```

Related Patterns

The Double Dispatch pattern is a kind of generalization of the Easy Operation Adder pattern.

Easy Operation Adder

Aliases

GoF Visitor, Visitor Pattern, Standard Visitor.

Context

There are a fair number of instances of a small number of classes that are rarely changed, and you are expecting to perform new operation that involves all or most of them [3].

Problem

How to define a new operation on classes over time without changing the classes of the elements on which it operates?

Forces

- Codes that change often will introduce new bugs that are hard to locate and fix.
- If the number of classes is large, adding new operations to these classes needs a significant overhead of recompile.
- Grouping distinct and unrelated operations in a class may lead to a solution that is hard to understand and maintain.
- A volatile interface is hard to use and maintain because the client code needs to change often.

Solution

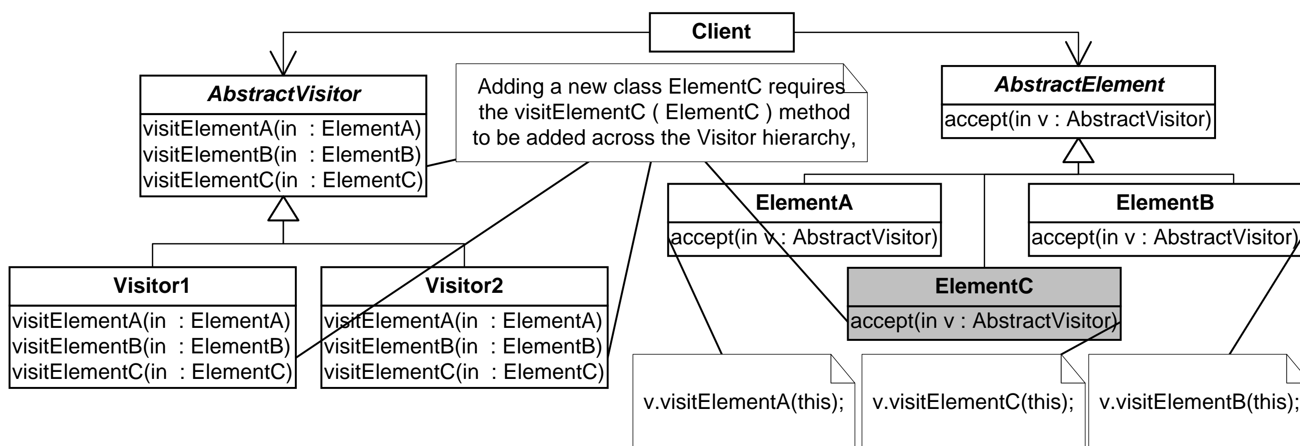

Figure 4: Structure for the Easy Operation Adder Pattern

Define two hierarchies. Related operations are grouped into a hierarchy called the visitor hierarchy, and the other hierarchy includes all elements and is called the element hierarchy. A method *accept* is defined across the element hierarchy. An operation is performed on an element object by a call of the *accept* on the element object and a supply of the corresponding visitor as argument that represents the desired operation. Thus, in the Easy Operation Adder pattern, the dispatch action is performed by the element object. A concrete element object knows which operation associated with it, so it dispatch a call to the corresponding visitor object by supplying itself as the parameter. Figure 4 shows the structure of the Easy Operation Adder pattern.

Rationale

If an object is too complex to understand, it's better to separate it into smaller objects that are less complex. Isolating the changeable parts in an object helps to leading to a system that is easy to maintain and extend. On the other hand, removal of unrelated operations will make the elements more cohesive. It is also desirable to separate code that changes from the code that does not. Adding new operations will only happen within the scope of the visitor hierarchy, while the interface of the element hierarchy keeps unchanged. If any new bugs are introduced due

to the adding of new operations, the bugs are easily located by examining the newly added operations in the visitor hierarchy. This solution structure also supports incremental programming because new operations to an object structure can be added incrementally.

Resulting Context

- A new operation can be easily added by simply adding a new visitor. Any code in the elements needs not to be changed.
- adding a new concrete element class is hard. Any addition of a new concrete element class **Xxx** requires a *visitXxx(Xxx)* method to be defined as abstract in the abstract visitor root class and implemented in all concrete element classes.

Code Samples

We'll use the simple expression example to illustrate this Visitor pattern. Instead of generating code, we implement the example as a calculator that calculates the arithmetic expression for integers. The variables and assignment expressions are added as extensions.

Expression Hierarchy

Figure 2 is the class diagram for the Expression hierarchy. The **Expression** is an interface:

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

The classes **ArithmeticExpr**, **AddExpr**, **SubExpr**, **MulExpr**, **DivExpr**, and **Constant** are defined as:

```
abstract class ArithmeticExpr implements Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    abstract public void accept(Visitor visitor);
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }

    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitAddExpr(this);
    }
}

class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitSubExpr(this);
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
```

```
    }
    public void accept(Visitor visitor) {
        visitor.visitMulExpr(this);
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitDivExpr(this);
    }
}

class Constant implements Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }
    public void accept(Visitor visitor) {
        visitor.visitConstant(this);
    }

    private int value;
}
```

Then we add two extended expressions to the Expression hierarchy. They are classes **Variable** and **Assignment** and can be declared like:

```
class Variable implements Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
    public void accept(Visitor visitor) {
        visitor.visitVariable(this);
    }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class AssignmentExpr implements Expression {
    protected AssignmentExpr(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
    public void accept(Visitor visitor) {
        visitor.visitAssignmentExpr(this);
    }
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }

    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy

The visitor hierarchy encapsulates calculation operations performed over the expression.

The following implementation also shows that a concrete element can also be a composite object.

```
interface Visitor{
    public void visitAddExpr(AddExpr expr);
    public void visitSubExpr(SubExpr expr);
    public void visitMulExpr(MulExpr expr);
    public void visitDivExpr(DivExpr expr);
    public void visitConstant(Constant expr);

    // newly added methods due to the extension of the expression
    public void visitAssignmentExpr(AssignmentExpr expr);
    public void visitVariable(Variable expr);
}

class CalculationVisitor implements Visitor {
    public void visitAddExpr(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visitSubExpr(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult - result;
    }
    public void visitMulExpr(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visitDivExpr(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult / result;
    }
    public void visitConstant(Constant expr) {
        result = expr.getValue();
     }

    // newly added methods due to the extension of the expression
    public void visitAssignmentExpr(AssignmentExpr expr) {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        rvalue.accept(this);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
    public void visitVariable(Variable expr) {
        result = expr.getValue();
    }
```

```
    public int getResult() { return result; }

    protected int result;
}
```

Related Patterns

Visitor pattern can be used to perform operations on a composite object defined in the Composite pattern [1]. It can also be applied to perform the interpretation in the Interpreter pattern [1].

Catch-All Operation

Aliases

Vlissides Visitor, Restricted Element Adder Visitor.

Context

You want to use visitors and you need to occasionally add new element classes. But you do not want to change the interfaces of the visitors once they have been defined.

Problem

How to prevent the modification of the existing visitor classes while allowing new element classes to be added occasionally?

Forces

- It is hard to modify an interface once it has been built in a framework.
- The structure proposed in the EASY OPERATION ADDER pattern separates operations from elements. Adding new operations are easy because it needs not modify the existing interfaces of elements. But adding a new element class is hard. All interfaces of related visitor classes must be modified to incorporate a method to visit the newly added element class.
- A structure with changing interface is fragile.
- An interface can be easily extended by inheritance without modifying the existing interface.
- If a method could not be added to the existing classes, it can be added to the extended classes. But since the definition of this method is missing in the ancestors, a type casting is mandatory if the message receiver has a declared type of the ancestors.
- If the implementation of a method in a class could not be changed, it can be overridden in the extended classes. An overridden method is only necessary if it has a distinct behavior that the original method could not handle.

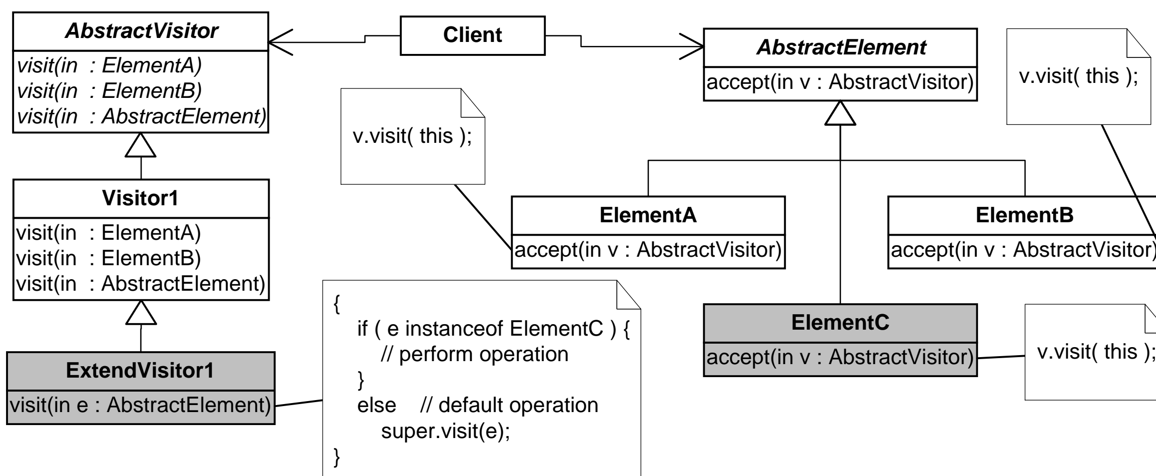

Figure 5: Structure of the Catch-All Operation Pattern

Solution

Similar with the EASY OPERATION ADDER pattern, the structure of the pattern defines two class hierarchies, the element hierarchy and the visitor hierarchy. A catch-all operation is defined in the base class **AbstractVisitor** and it is overridden in its descendants. If a new element class is added in the element hierarchy, new visitor classes

are defined to extend the existing concrete visitor classes and the catch-all operation is re-written to perform the run-time type test on the newly added element class.

Figure 5 shows the structure for the CATCH-ALL OPERATION pattern. The newly added classes are adorned in gray. The **ElementC** is a newly added element class. A **ExtendVisitor1** class is defined to subclass the **Visitor1** and the catch-all operation *visit* is overridden to perform the run-time type checking on the **ElementC**. If a element to be visited is a newly added element, specific operation related to the new class is performed. Otherwise, the method demonstrates its previous behavior.

Rationale

Two simple class hierarchies as that defined in the EASY OPERATION ADDER pattern have many advantages including one that allows the new operation to be added easily without recompiling the element hierarchy. A catch-all operation is so blurring that it leaves margin to allow the new elements to be easily added and handled. Occasional addition of the element classes will not degrade the programming style of the catch-all operation because the tag-and-case statement is very short.

Resulting Context

- New element classes can be added occasionally without any modification of the existing visitor classes.
- If new element classes are constantly added, CATCH-ALL OPERATION will degrade into a tag-and-case-statement style of programming.

Code Samples

We'll still use the expression example to calculate the arithmetic expression for integers. The variables and assignment expressions are added as extensions.

Element Hierarchy

The classes defined in the element hierarchy are the same as the class declarations in the EASY OPERATION ADDER.

Visitor Hierarchy

```
interface Visitor{
    public void visit(AddExpr expr);
    public void visit(SubExpr expr);
    public void visit(MulExpr expr);
    public void visit(DivExpr expr);
    public void visit(Constant expr);

    // catch-all operation
    public void visit(Expression expr);
}

class CalculationVisitor implements Visitor {
    public void visit(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visit(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult - result;
```

```
    }
    public void visit(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visit(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult / result;
    }
    public void visit(Constant expr) {
        result = expr.getValue();
     }
    // catch-all operation
    public void visit(Expression expr) { }
    public int getResult() { return result; }

    protected int result;
}
```

Operations for newly added elements are encapsulated in the extended concrete visitor class.

```
class ExtendCalculationVisitor extends CalculationVisitor {
    // catch-all operation
    public void visit(Expression expr) {
        if ( expr instanceof AssignmentExpr )
            visit((AssignmentExpr)expr);
        else if ( expr instanceof Variable )
            visit((Variable)expr)
        else
            super.visit(expr);
    }
    public void visit(AssignmentExpr expr) {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        rvalue.accept(this);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
    public void visit(Variable expr) {
        result = expr.getValue();
    }
}
```

Related Patterns

This pattern is an improvement to the Easy Operation Adder pattern [1].

Visser's Visitor [8] is a variation on the Vlissides Visitor framework [7]. It defines generic counterparts **AnyVisitor** and **AnyVisitable** for visitor and element hierarchies respectively.

Easy Element Adder

Aliases

Sablecc Visitor.

Context

You have visitor that works well. You are expecting to add element classes in the future, but you are unable to change the existing class interfaces.

Problem

How to easily add new element or new operations without modifying the existing interfaces?

Forces

- The Easy Operation Adder pattern makes the addition of new operations easy, but adding new element classes is hard because it needs to modify the existing interfaces of the visitors.
- The Catch-All Operation pattern is not suitable because constantly adding new element classes will degrade the implementation of the catch-all operation to be a tag-and-case style of programming.
- Inheritance provides a good means to extend the existing interface without actually modifying it.
- If the inheritance tree is high, a structure is hard to understand because the underneath classes could not be understood without resorting to its ancestors.
- A type casting can satisfy the compiler because it can precisely refer to a method defined in some classes but not in others. But the type casting is unsafe. It always depends on the good wills of the programmer.

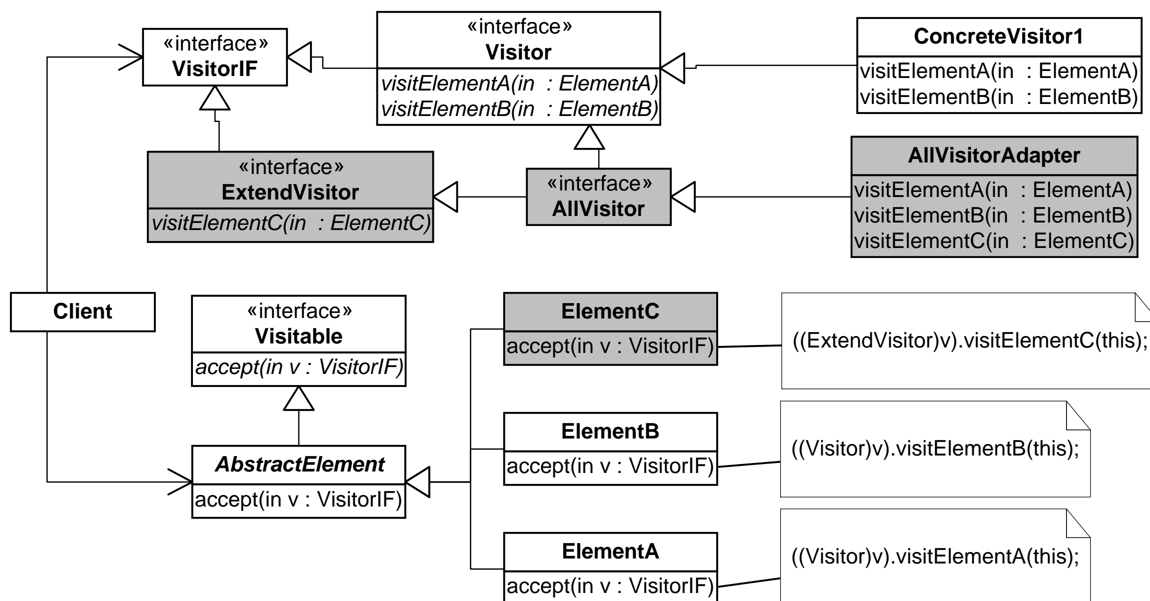

Figure 6: Structure for the Easy Element Adder Pattern

Solution

Separate the elements and their operations in two class hierarchies, one is the Visitor, and the other is the Element. An interface **VisitorIF** is defined on the top of the visitor hierarchy. New operations can be easily added by defining concrete visitor classes in the visitor hierarchy. New element classes can be added by extending existing interfaces in the Visitor hierarchy and defining concrete visitor to implement the newly added interface.

Figure 6 shows the structure for the EASY ELEMENT ADDER pattern. It is a snapshot after adding a new element class **ElementC**. The new added classes are adorned in gray. The interfaces of the visitors are extended by introducing two new interface classes: **ExtendVisitor** and **AllVisitor**. The previous extends the root interface **VisitorIF** by defining a new visit operation for the newly added element class. The latter represents the complete interface for the visitors due to the changing of the subject elements. Class **AllVisitorAdapter** is a new visitor class that implements the complete interface **AllVisitor**. In order to adapt to the changing interfaces in the visitor hierarchy, the *accept* methods that are defined in the element classes must indicate which visitor class it dispatches to by performing down-casting.

Rationale

An existing interface is hard to change, but it is easy to extend. A redefinition of the concrete visitors due to any addition of new element class is easy to understand without resorting to the ancestors. In general, both hierarchies are developed or maintained by the same person. So a type casting in the *accept* can be considered to be safe.

Resulting Context

- The elements and their operations can be easily extended without any restriction and without modifying the existing class interfaces.
- Frequently adding new classes makes the class hierarchy too complicated to understand and implement.
- The type-casting in the *accept* method makes the programming unsafe and tightly coupled with the concrete element types.

Code Samples

The following shows the Java implementation on how to applying EASY ELEMENT ADDER pattern to the calculation for the simple expression example.

Expression Hierarchy

```
interface Visitable { }

interface Expression extends Visitable {
    public void accept(Visitor visitor);
}
```

The classes **ArithmeticExpr**, **AddExpr**, **SubExpr**, **MulExpr**, **DivExpr**, and **Constant** are defined as:

```
abstract class ArithmeticExpr implements Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    abstract public void accept(Visitor visitor);
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }

    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitAddExpr(this);
    }
}
```

```
class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
       ((Visitor)visitor).visitSubExpr(this);
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitMulExpr(this);
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitDivExpr(this);
    }
}

class Constant implements Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitConstant(this);
    }

    private int value;
}
```

Two extended expressions, classes **Variable** and **AssignmentExpr** can be added to the Expression hierarchy as following:

```
class Variable implements Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
    public void accept(Visitor visitor) {
        ((ExtendVisitor)visitor).visitVariable(this);
    }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class AssignmentExpr implements Expression {
    protected AssignmentExpr(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
    public void accept(Visitor visitor) {
```

```
        ((ExtendVisitor)visitor).visitAssignmentExpr(this);
    }
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }

    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy

For extending the new elements without modifying the existing visitors, the new visitor interfaces are defined to support the extension.

```
interface VisitorIF { }

interface Visitor extends VisitorIF {
    public void visitAddExpr(AddExpr expr);
    public void visitSubExpr(SubExpr expr);
    public void visitMulExpr(MulExpr expr);
    public void visitDivExpr(DivExpr expr);
    public void visitConstant(Constant expr);
}

interface ExtendVisitor extends VisitorIF {
    public void visitAssignmentExpr(AssignmentExpr expr);
    public void visitVariable(Variable expr);
}

interface AllVisitor extends Visitor, ExtendVisitor { }

class CalculationVisitor implements Visitor {
    public void visitAddExpr(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visitSubExpr(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult - result;
    }
    public void visitMulExpr(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visitDivExpr(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
```

```
        right.accept(this);
        result = leftResult / result;
    }
    public void visitConstant(Constant expr) {
        result = expr.getValue();
     }
    public int getResult() { return result; }

    protected int result;
}

 class ExtendCalculatorVisitor extends CalculatorVisitor implements AllVisitor {
   public void visitAssignmentExpr(AssignmentExpr expr) {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        rvalue.accept(this);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
    public void visitVariable(Variable expr) {
        result = expr.getValue();
    }
}
```

Related Patterns

This pattern is an improvement to the EASY OPERATION ADDER pattern [1].

Acyclic Visitor [10] allows new elements to be added without changing the existing classes. It defines an individual visitor interface for each element to provide the operation interface. A dynamic cast is needed in the *accept* method to cast the visitor parameter to its corresponding visitor interface.

Easy Element and Operation Adder

Aliases

Reflective Visitor.

Context

You want to use visitors and you want to make both addition of new operations and new elements easy. You are also expecting to reuse the element hierarchy.

Problem

How to enable both addition of new operations and new element classes easy, while at the same time, make the element hierarchy stand alone?

Forces

- A structure that proposed in the EASY OPERATION ADDER pattern makes adding new operations easy, but adding new element classes is hard and reusing these elements is also hard.
- Structures proposed in the the CATCH-ALL OPERATION pattern and the EASY ELEMENT ADDER pattern support the extension on both the element hierarchy and visitor hierarchy, but reuse elements is hard because these elements depend on the visitors.
- Breaking the cyclic dependency and letting the element hierarchy stand alone, the system can reuse the elements hierarchy easily.
- If the dependency is removed on the side of elements, the Visitors must carry out the dispatch action that requires the type information about the related elements.
- The reflection technique provides an easy way to locate a method if its naming convention is known in advance. But the use of reflection gains simplicity in the expense of performance.
- A unified simple interface for operations is easy to use and maintain.
- To modify an interface is hard, but extend it is easy.

Solution

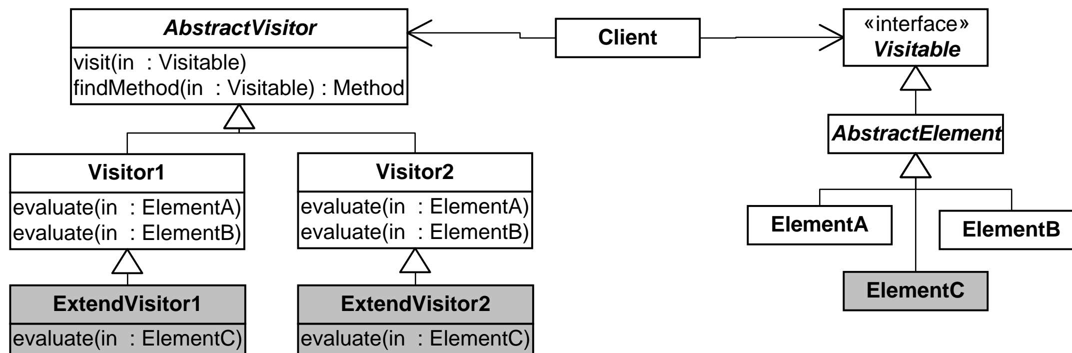

Figure 7: Structure for the Easy Element and Operation Adder Pattern

Separate operations from the elements. Objects of the elements to be visited are specified as **Visitable**. All *accept* methods are removed from the element hierarchy. Method *visit* in the root class **AbstractVisitor** is the only visible method in the visitor hierarchy and invokes the *findMethod* method to perform the dispatch operation. The method *findMethod* uses reflection technique to locate the desirable methods for the supplied parameter. Various *evaluate* methods are defined in the concrete visitors to perform specific operations on the related elements. Figure 7 shows the structure for the EASY ELEMENT AND OPERATION ADDER pattern.

Rationale

A common interface **Visitable** enables distinct elements to be built in different element hierarchies to share a common ancestor. The method *visit* implemented in the **AbstractVisitor** class is the only public method in the visitor hierarchy. It takes a role of a dynamic dispatcher by invoking the corresponding *evaluate* method found by *findMethod* method. The reflection is used by the method *visit* and *findMethod* to support the method finding and method invocation dynamically. Because the dispatch operation is performed by the **Visitor** class, the *accept* methods can be removed from the element hierarchy and thus the developer can reuse these elements without the visitors' supports. The method *evaluate* can visit a composite object recursively because an *evaluate* method is invoked by the *visit* method and it can also make a call to the method *visit* if needed. The use of reflection will lead to a severe performance penalty, but it can still be accepted if performance is not a major concern in the system design. Any addition of new operations only needs to define a new concrete visitor class in the visitor hierarchy. Any addition of new element class only needs to extend the related concrete visitor classes and define new *evaluate* methods in the new visitor classes. Existing classes are thus kept from any potential modification.

Resulting Context

1. As that of the Easy Operation Adder pattern, adding a new operation is easy. The existing code can be avoided from modifying by simply subclassing the visitor hierarchy if a new operation over the element hierarchy is to be added.

2. Adding a new element class is easy. Since the **AbstractVisitor** is responsible for the dynamic dispatch, any operation operating on this new element can be defined within a new visitor subclass without modifying the existing codes. The system's extensibility is then improved.

3. The cyclic dependencies are broken and the coupling between the element hierarchy and the visitor hierarchy is reduced. As the key of the traditional Visitor pattern, the double-dispatch technique is used to associate the operation with the concrete element at run time. But this technique reduces the system's reusability. With the reflection technique, the Visitor pattern can avoid the cyclic dependencies by performing the dynamic dispatch within the **AbstractVisitor** class. Since the visitor is responsible for the dynamic dispatch, the element hierarchy has no knowledge about the visitor. Hence the system's reusability is improved. On the other hand, the visitor can visit any object that has a corresponding *evaluate* operation in the visitor hierarchy only if this object has a **Visitable** interface.

4. The *visit* method is the only visible interface of the visitor hierarchy. The client only needs to invoke this method to perform any desired operation on the visitable elements. Since the interface and the implementation of the operations on the elements are separated, the client is shielded from any potential changes of the implementation details.

5. The name of the operation needs to be fixed. The system designer should follow the name convention and keeps all the operations named *evaluate*. Since the *evaluate* is only visible within the visitor hierarchy, there is no direct influence to other parts of the system.

6. The programming languages that used to implement this Visitor pattern need to support reflection. This limitation lets some languages, like C++, can not be used as the implementation language for this pattern.

7. The use of reflection imposes a significant performance penalty and reduces the system efficiency [5]. This pattern can be considered to be used only in time non-critical systems.

Code Samples

The Java implementation of the Easy Element and Operation Adder for the simple expression example as following:

Expression Hierarchy

The general interface **Visitable** for all visitable expressions may be declared like:

```
interface Visitable { }
```

The **Expression** is an abstract class implementing the **Visitable** interface:

```
abstract class Expression implements Visitable { }
```

The classes **ArithmeticExpr**, **AddExpr**, **SubExpr**, **MulExpr**, **DivExpr**, and **Constant** are defined as:

```
abstract class ArithmeticExpr extends Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }
    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class Constant extends Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }

    private int value;
}
```

Then two extended expressions, classes **Variable** and **AssignmentExpr** are added to the Expression hierarchy.

```
class Variable extends Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class AssignmentExpr extends Expression {
    protected AssignmentExpr(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
```

```
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }

    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy

The abstract class **Visitor** would be declared in Java like:

```
abstract class Visitor {
    public void visit(Visitable v) throws NoSuchMethodException {
        Method m = findMethod(v);
        try {
            m.invoke(this, new Object[] { v });
        }
        catch ( IllegalAccessException    e1 ) { /* code handling */ }
        catch ( InvocationTargetException e2 ) { /* code handling */ }
    }

    private Method findMethod(Visitable v) throws NoSuchMethodException {
        String methodName = "evaluate";
        Class visitable = v.getClass();
        while ( isAncestorOf("Visitable", visitable) {
            Class visitor = getClass();
            while ( isAncestorOf("Visitor", visitor) {
                try {
                    Method m = visitor.getDeclaredMethod(methodName,new Class[]{visitable});
                    return m;
                } catch ( NoSuchMethodException e ) {
                    visitor = visitor.getSuperclass();
                }
            }
            visitable = visitable.getSuperclass();
        }
        String errMsg = "put error message here";
        throw new NoSuchMethodException(errMsg);
    }

    private boolean isAncestorOf(String ancestorName, Class descendant) {
        try {
            return Class.forName(ancestorName).isAssignableFrom(descendant);
        }
        catch ( ClassNotFoundException e ) { /* code handling */ }
        return false;
    }
}
```

The **CalculationVisitor** derives from the Visitor class and is defined to perform a calculation operation on the expressions. Its declaration may like:

```
class CalculationVisitor extends Visitor {
    protected void evaluate(AddExpr expr) throws NoSuchMethodException {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        visit(left);
        int leftResult = result;
        visit(right);
        result = leftResult + result;
    }
```

```
    protected void evaluate(SubExpr expr) throws NoSuchMethodException {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        visit(left);
        int leftResult = result;
        visit(right);
        result = leftResult - result;
    }
    protected void evaluate(MulExpr expr) throws NoSuchMethodException {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        visit(left);
        int leftResult = result;
        visit(right);
        result = leftResult * result;
    }
    protected void evaluate(DivExpr expr) throws NoSuchMethodException {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        visit(left);
        int leftResult = result;
        visit(right);
        result = leftResult / result;
    }
    protected void evaluate(Constant c) { result = c.getValue(); }
    public int getResult() { return result; }
    protected int result;
}
```

In order to adapt to the changing of the Expression hierarchy, a concrete Visitor class **ExtendCalculationVisitor** is defined to perform calculation operation on the newly added Expression classes. The class **ExtendCalculation-Visitor** is an immediate subclass of the **CalculationVisitor** and can be declared like:

```
class ExtendCalculationVisitor extends CalculationVisitor {
    protected void evaluate(Variable var) {
        result = var.getValue();
    }
    protected void evaluate(AssignmentExpr expr) throws NoSuchMethodException {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        visit(rvalue);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
}
```

Client Code

To calculate the expression x= 2*y+3, a client method *calculate* can be written as:

```
void calculate() {
        Expression expr = new Assignment( new Variable("x"),
                                          new AddExpr( new MulExpr( new Constant(2),
                                                                    new Variable("y") ),
                                                       new Constant(3) ) );
        ExtendCalculationVisitor calculator = new ExtendCalculationVisitor();
        try {
            calculator.visit(expr);
            System.out.println( calculator.getResult() );
        }
        catch ( NoSuchMethodException e ) { /* code handling */ }
}
```

Related Patterns

This pattern is an improvement to all other Visitor patterns presented in this paper and is applied when the programming environment supports reflection and efficiency is not a major concern.

Walkabout Visitor pattern [5] removes the cyclic dependency between the elements and the visitor hierarchy by using the Java reflection technique to perform the dispatch action. Its drawback is that it can not visit a complex multi-level composite hierarchy. The Reflective Visitor pattern can replace Walkabout Visitor pattern wherever it is used.

Blosser Visitor pattern [6] and Jeanne Sebring [4] Visitor pattern also implements the dispatch action with Java reflection. It supports re-dispatch actions so that it can visit a complex multi-level composite hierarchy. The `accept` method is still used to implement the recursive traversal in this pattern. The Blosser Visitor pattern can be replaced by the Reflective Visitor pattern when the designer wants to remove the cyclic dependencies and to define a unified operation interface and encapsulate the implementation details.

Extrinsic Visitor pattern [9] removes the cyclic dependencies between the element hierarchy and the visitor hierarchy by defining a *dispatch* method in the visitor to perform the dispatch action dynamically. Although the Extrinsic Visitor Pattern reduces the coupling between the elements and visitors, adding new element classes is hard because all related visitor classes have to redefined. The Extrinsic Visitor Pattern is limited to be implemented under a C++ development environment.

Conclusion

This paper presents a pattern language to VISITORs that have been proposed since 1995. The authors hope the pattern language can assist the application developer to better understand the circumstance a VISITOR pattern can be applied and the pros and cons of them so that a right decision can be made. However, this pattern language does not come to the end. As long as new VISITOR patterns continue to emerge, this pattern language will evolve with them.

Acknowledgements

We would like to thank Dr. Stephane Ducasse, our PLoP '2001 shepherd, for the valuable comments for this paper.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] Etienne Gagnon. Sablecc, An Object-Oriented Compiler Framework. Master's thesis, McGill University, 1998.

[3] James W. Cooper. *The Design Patterns: Java Companion.* Addison-Wesley, 1998.

[4] Jeanne Sebring. Reflecting on the Visitor Design Pattern. *Java Report*, March 2001.

[5] Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. *Technical Report 05, University of Technology, Sydney*, 1997.

[6] Jeremy Blosser. Reflect on the Visitor Design Pattern. *http://www.javaworld.com/javatips/jw-javatip98.html*, January 2001.

[7] John Vlissides. Visitor in Frameworks. *C++ Report*, November 1999.

[8] Joost Visser. Visitor Combination and Traversal Control. *http://www.jforester.org*, 2001.

[9] Martin E. Nordberg III. The Variations on the Visitor Pattern. *PLoP '96 Writer's Workshop*, September 1996.

[10] Robert C. Martin. Acyclic Visitor. *PLoP '96*, September 1996.

[11] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns: Smalltalk Companion.* Addison-Wesley, 1998.

[12] Yun Mai and Michel de Champlain. Reflective Visitor Pattern. *Submitted and accepted for EuroPLoP '2001 Writer's workshop, Irsee, Germany*, July 2001.