Reflective Visitor Pattern

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

Abstract

The Visitor pattern wraps associated operations that are performed on the elements of an object structure into a separate object. It allows the software designer to define new kinds of operations over the object structure without changing the classes of this structure. But a well-known drawback of this standard Visitor structure is that extending the object structure is hard. This paper presents the design and implementation of a flexible Visitor pattern based on the reflection technique, we call it the Reflective Visitor pattern. The reflection technique enables the Visitor to perform the run-time dispatch action on itself. The separation of the run-time dispatch action from the object structure makes any extension to the object structure become easy. It also removes the cyclic dependencies between the Visitor and the object structure, so the reusability and extensibility of the system are improved.

Intent

Define a new operation over the object structure without changing the classes of the elements on which it operates, while in the meantime, allow the element classes in the object structure to be extended constantly without changing the existing system.

Motivation

Consider the code generation design in a compiler framework. The responsibility of the code generation is to generate a target code list as the output of the compiler. The format of the target code list is specified by the system requirements, which may require the target code list to be compatible with different operation platforms. In order to support the cross-platform features, different code generation operations need to co-exist and allow easily switching from one to another. On the other hand, the code generation process depend on the parser result. The parser result, normally in terms of an abstract syntax tree, can be represented as a compound data structure, each of whose elements is constructed from the language structure. The language structure is a composite hierarchy consisting of a set of terminals and non-terminals, which can be extracted from the language grammar specification. Since the operation of code generation is actually performed on the abstract syntax tree, its design should accommodate any potential changes on the language structure. For a compiler framework, the design of the code generation needs to:

1. Prepare for changes of the generated code format.
2. Prepare for modifications in the language grammar.
3. Reduce the coupling between the language structure and the code generator in order to promote the system reusability.

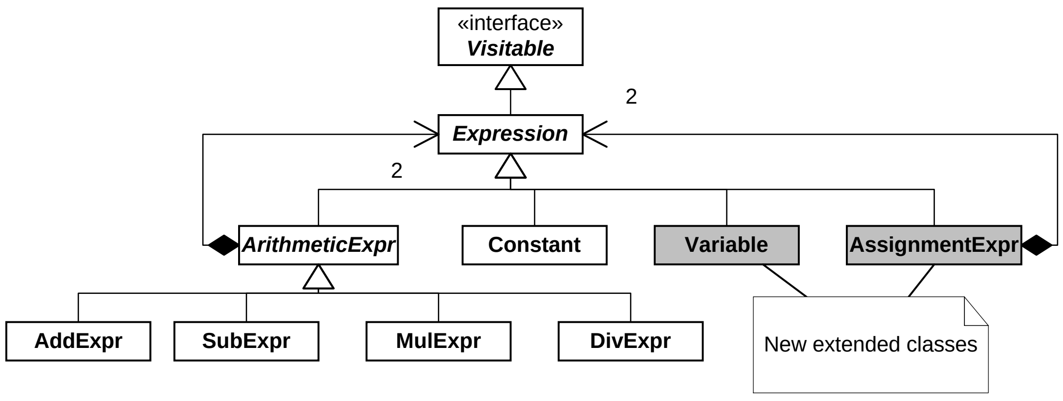

Figure 1: An Expression Hierarchy

Given a simple expression example, suppose it supports arithmetic expression such as addition, subtraction, multiplication, and division for constants. Figure 1 shows the language structure hierarchy for this expression example. The language (expression) structure hierarchy is organized as a composite structure and can be implemented by a Composite design pattern [3]. 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 process then performs the code generation operations 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. 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 1, 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. It may also require easy switching from one format to another and easy addition of new kind of output code format. For simplicity, we suppose the code generation for above example should support the two different virtual machines, VM1 and VM2.

The basic design issues in the design of code generation for this expression example are:

1. Both the code generations for VM1 and VM2 should be represented as different operations that are performed on the abstract syntax tree.
2. The code generation should support easy switching between the VM1 output format and the VM2 output format. Any future extension of the output format can be easily added without modifying and re-compiling the existing system program.
3. The addition of the variable and assignment expression needs not affect the rest of the system.
4. The language (expression) structure can stand alone and has no knowledge about the code generator.

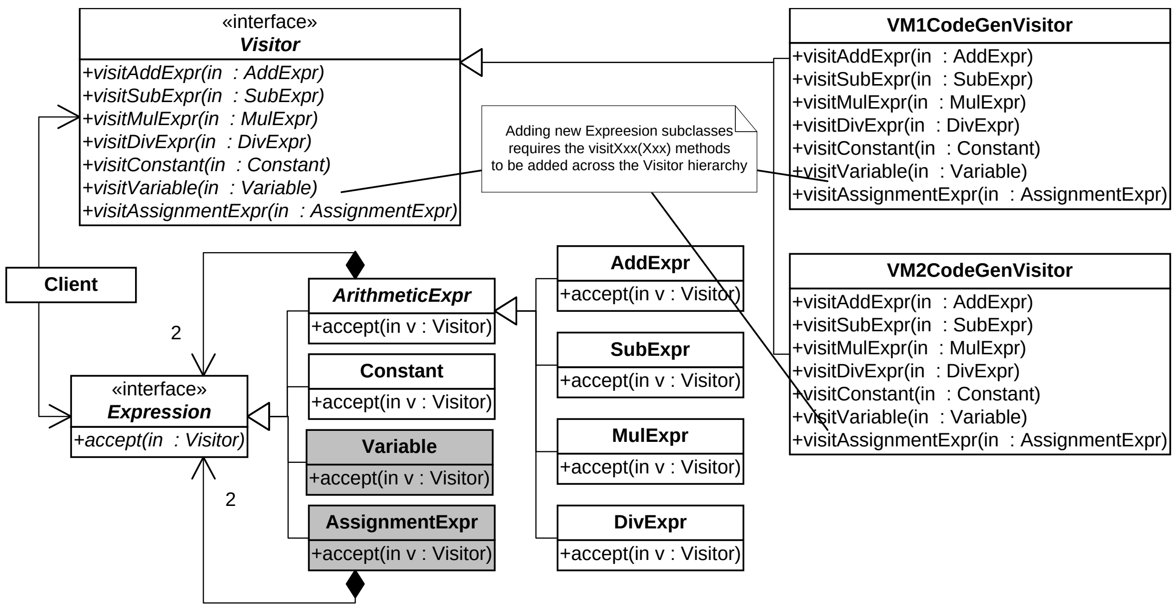
«interface»
Visitor
+visitAddExpr(in : AddExpr)
+visitSubExpr(in : SubExpr)
+visitMulExpr(in : MulExpr)
+visitDivExpr(in : DivExpr)
+visitConstant(in : Constant)
+visitVariable(in : Variable)
+visitAssignmentExpr(in : AssignmentExpr)
VM1CodeGenVisitor
+visitAddExpr(in : AddExpr)
+visitSubExpr(in : SubExpr)
+visitMulExpr(in : MulExpr)
+visitDivExpr(in : DivExpr)
+visitConstant(in : Constant)
+visitVariable(in : Variable)
+visitAssignmentExpr(in : AssignmentExpr)
Adding new Expreesion subclasses requires the visitXxx(Xxx) methods to be added across the Visitor hierarchy
Client
ArithmeticExpr
+accept(in v : Visitor)
AddExpr
+accept(in v : Visitor)
VM2CodeGenVisitor
+visitAddExpr(in : AddExpr)
+visitSubExpr(in : SubExpr)
+visitMulExpr(in : MulExpr)
+visitDivExpr(in : DivExpr)
+visitConstant(in : Constant)
+visitVariable(in : Variable)
+visitAssignmentExpr(in : AssignmentExpr)
2
«interface»
Expression
+accept(in : Visitor)
Constant
+accept(in v : Visitor)
SubExpr
+accept(in v : Visitor)
Variable
+accept(in v : Visitor)
MulExpr
+accept(in v : Visitor)
2
AssignmentExpr
+accept(in v : Visitor)
DivExpr
+accept(in v : Visitor)

Figure 2: Apply GoF Visitor Pattern to the Expression Example

As Gamma et al. pointed out in their Design Patterns book [3], the Visitor pattern is suitable to represent an operation to be performed on the elements of an object structure. We refer this Visitor pattern as GoF Visitor pattern in this paper. The GoF Visitor pattern lets the designer define a new operation over the object structure without changing the elements of that structure. Figure 2 shows the design of the above expression example with the GoF Visitor pattern. By applying GoF Visitor pattern, the code generation can be easily changed or extended to support different kinds of output code formats, for example, switching between the VM1 output format and the VM2 output format. But we can also see that the extension of the expression structure becomes difficult. In our example, to support the addition of the variable and assignment expression, the GoF Visitor pattern requires the new code generation methods, such as *visitVariable(Variable)* and *visitAssignmentExpr(AssignmentExpr)*, to be added across the Visitor hierarchy (the `Visitor` interface the `VM1CodeGenVisitor` class and the `VM2CodeGenVisitor` class in figure 2). Therefore, all classes in the Visitor hierarchy need to be modified and re-compiled due to changings of the expression structure. Obviously, the GoF Visitor pattern could not fit our system design requirement and it is not suitable for the code generation design in a compiler framework.

There are several variations of the Visitor pattern [6] intended to overcome this shortcoming so that the Visitor pattern can also be used in an environment that the object structure changes often. A brief summary of them is mentioned in the Related Patterns section of this paper.

The Reflective Visitor pattern introduced in this paper supports both the changes of the generated code format and the changes of the language grammar without changing the existing classes. It achieves this goal by performing the dynamic operation dispatch in the Visitor class through reflection. The Visitor class declares a *visit* method to be responsible for the dynamic dispatch. The corresponding operation can be invoked automatically at run-time. Therefore the *accept* methods are no longer needed and the cyclic dependencies are removed. This *visit* method is defined as the only interface visible to the outside of the system so that detailed implementation of code generation is hidden from the outside of the system.

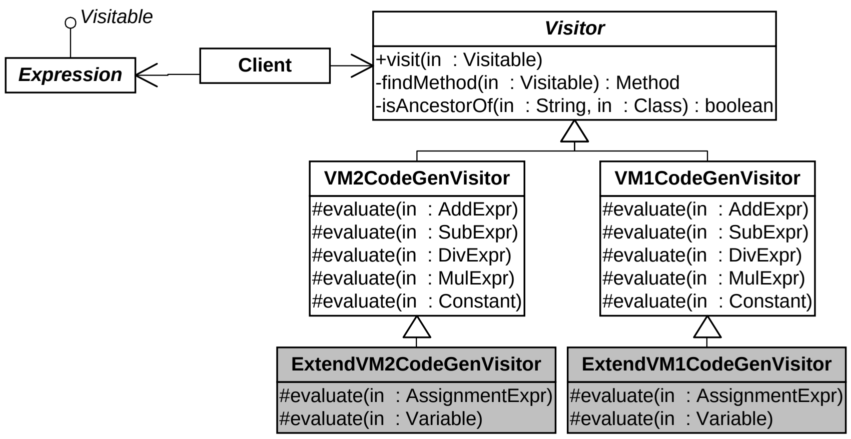

Figure 3: Apply Reflective Visitor Pattern to the Expression Example

Figure 3 shows the solution for the above expression example by applying the Reflective Visitor pattern to the code generation. The *visit* method declared in the `Visitor` class takes the concrete `Expression` object as argument. It queries the concrete `Expression` class information through reflection to find the *evaluate* method based on the concrete `Expression` object and then invokes the *evaluate* method to perform the operation. In our example, to support two code generation formats for the virtual machine VM1 and VM2, we define two concrete visitors `VM1CodeGenVisitor` and `VM2CodeGenVistor` respectively. The addition of the variable and assignment expression in the expression structure only requires two new visitor classes (`ExtendVM1CodeGenVistor` and `ExtendVM2CodeGenVisitor`) to be added in the Visitor hierarchy and their *evaluate* methods to be implemented. All existing classes in both the expression structure and the Visitor hierarchy need not to be modified and re-compiled. All the *evaluate* operations in the Visitor hierarchy are declared protected so detailed implementation information of code generation is encapsulated.

With the Reflective Visitor pattern, the system designer can easily add new operations to the object structure by simply defining new concrete Visitor classes, as what the GoF Visitor pattern does. On the other hand, the designer can also easily add new concrete Element classes by simply defining new Visitor subclasses in the Visitor hierarchy. 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 object structure. Since the interface and the implementation of the operations on the object structure are separated, the client is shielded from any potential changes of the implementation details.

Applicability

The Reflective Visitor pattern can be applied when:

1. The programming language that the designer uses to implement the Reflective Visitor design pattern should support reflection. For example, Java.

2. An object structure contains many classes of objects with differing interfaces, and the designer performs operations on these objects that depend on their concrete classes [3].

3. Distinct and unrelated operations need to be performed on objects in an object structure [3].

4. The object structure may be changed often to fit changing requirements. The designer don't want to redefine the interface and recompile all existing classes.

5. The designer may need to reuse the object structure in the future and thus wants to break the cyclic dependencies and de-couple the object structure and the Visitor hierarchy.

6. The designer wants to define a unified stable operation interface for the client and to encapsulate the implementation details.

7. The run-time efficiency is not a major concern in the design.

Structure

Figure 4 shows the structure of the Reflective Visitor design pattern.

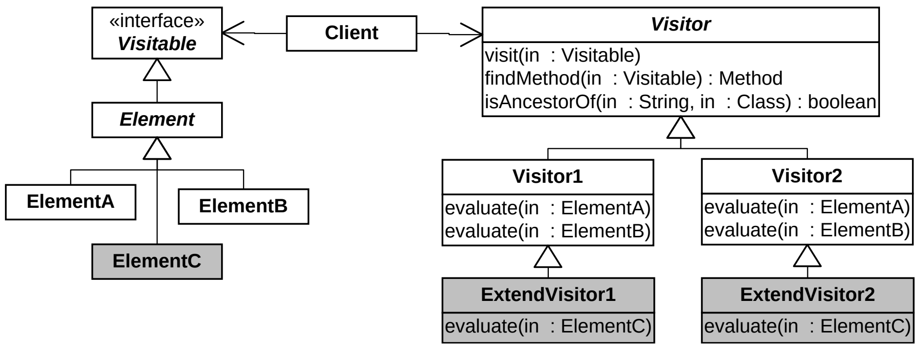

Figure 4: The Reflective Visitor Pattern Structure

Participants

Visitor (Visitor)

1. The abstract class `Visitor` is the facade and the root of the Visitor class hierarchy. All the concrete `Visitor` classes are derived from it.

2. The `Visitor` class defines a public *visit* operation, which is the unified operation interface for the `Visitor` class. The client invokes the *visit* method to execute the corresponding operations on the object structure.

3. The *visit* method takes a `Visitable` interface object as argument. It performs the dynamic dispatch for the concrete `Element` object. That is, the *visit* method finds the corresponding concrete *evaluate* operation from the `Visitor` hierarchy and invokes it at run time.

ConcreteVisitor (VM1CodeGenVisitor, VM2CodeGenVisitor)

1. The `ConcreteVisitor` defines a set of *evaluate* operations, each implements the specific behavior for the corresponding `ConcreteElement` class.

2. The *evaluate* operations are declared as protected so that the implementation information can be hidden from the outside of the system.

Visitable

The interface `Visitable` is the interface for all the classes that can be visited. It is an empty interface and provides the run time type information for the `Visitor`.

Element (Expression)

The `Element` class is the root of the `Element` class hierarchy to be visited. It implements the `Visitable` interface. All the concrete `Element` classes derive from the `Element` class and they have no knowledge about the `Visitor`.

ConcreteElement (AddExpr, SubExpr, MulExpr, DivExpr)

The `ConcreteElement` class is a descendant of the `Element` class. The `Element` class and all the concrete `Element` classes construct the Element class hierarchy.

Collaborations

A client who uses the Visitor pattern must create a `ConcreteVisitor` object and pass the `ConcreteElement` object to the `Visitor` for visiting.

The `Visitor` uses reflection to query the `ConcreteElement` class information and finds the corresponding *evaluate* method whose argument type is same as that of the `ConcreteElement`. The search process begins from the `ConcreteVisitor` class, and then traces up its ancestors until it reaches the root of the Visitor hierarchy. If the method is found, it is invoked. Otherwise, we assume that this *evaluate* method is defined for the ancestor classes of the `ConcreteElement`, so the search process repeats for these ancestors. If all the ancestors of the `ConcreteElement` have been tried and the corresponding *evaluate* method can not be found, an error is thrown. Figure 5 is the sequence diagram of the *visit* method.

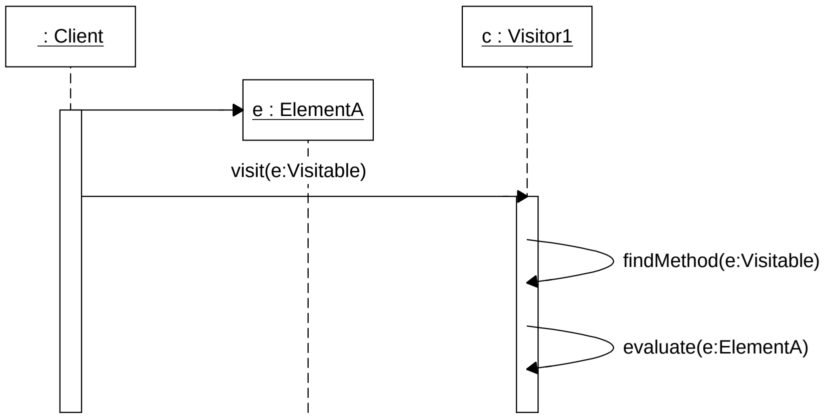

Figure 5: The Sequence Diagram for the Visiting Process

Consequences

Some of the benefits of the Reflective Visitor pattern are:

1. As that of the GoF Visitor 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 an object structure is to be added.

2. Adding a new Element class `ExtendElement` is easy. Since the `Visitor` is responsible for the dynamic dispatch, any operation operating on this new `ExtendElement` can be defined within a new subclass of the `ConcreteVisitor` without modifying the existing codes. The system's extensibility is then improved.

3. The cyclic dependencies are broken and the coupling between the object structure and the Visitor hierarchy is reduced. As the key of the standard Visitor pattern, the double-dispatch technique is used to bind the operation with the concrete element in the object structure at run time. But this technique reduces the system's reusability. With the reflection technique, the Reflective Visitor pattern can avoid the cyclic dependencies by performing the dynamic dispatch within the Visitor 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 object structure. Since the interface and the implementation of the operations on the object structure are separated, the client is shielded from any potential changes of the implementation details.

The Reflective Visitor pattern has some liabilities:

1. 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.

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

3. The use of reflection imposes a significant performance penalty and reduces the system efficiency [9]. This pattern can be considered to be used only in time non-critical systems.

Implementation

The abstract Visitor class declares a unique method *visit* that takes a `ConcreteElement` object for visiting. This *visit* method invokes the *findMethod* operation to fetch the corresponding *evaluate* method object through reflection. Then the *visit* method invokes the *evaluate* method object to execute the operation related to the `ConcreteElement` object.

The *findMethod* takes a `Visitable` interface object as argument. It queries the corresponding *evaluate* method object based on the method name "evaluate" and the `ConcreteElement` class object. The search process starts from the current `ConcreteVisitor` class and traces up until it reaches the root class (`Visitor`) in the Visitor hierarchy. If the corresponding *evaluate* method is found, the *findMethod* returns the method object. Otherwise, the search process repeats for the ancestors of this `ConcreteElement` until it reaches the root class (`Visitable`) in the Element hierarchy. If all the ancestors have been tried and the corresponding *evaluate* method can not be found, an error is thrown.

There is a nested loop statement in the method *findMethod*. The inner loop is used to search for an *evaluate* method with a given Element object as parameter. The outer loop assigns the Element object to the inner loop for search. The assignment principle is that the Element object to be visited is tried first, then the Element object whose declare type is the superclass of the current Element is tried until the corresponding *evaluate* method is found or an error is thrown if the search reaches the root Element interface `Visitable`. The nested loop statement guarantees that the searches trace up over the Visitor hierarchy for the `ConcreteElement` object and all its ancestors until the corresponding *evaluate* method is found.

The Visitor class 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 `ConcreteVisitor` class derives from the Visitor class. It declares an *evaluate* operation for each class of `ConcreteElement` that need to be visited. Each *evaluate* operation in the `ConcreteVisitor` takes a particular `ConcreteElement` as argument. The Visitor accesses the interface of the `ConcreteElement` directly, and the visitor-specific behavior for that corresponding `ConcreteElement` class is executed.

```
class ConcreteVisitor extends Visitor {
    protected void evaluate(ConcreteElement1 c1) {
        // perform the operation on ConcreteElement1;
    }
    protected void evaluate(ConcreteElement2 c2) {
        // perform the operation on ConcreteElement2;
    }
}
```

Sample Code

We'll use the Expression example defined in the Motivation section to illustrate the Reflective 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 1 is the class diagram for the Expression hierarchy. The interface `Visitable` 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 we add two extended expressions to the Expression hierarchy. They are classes `Variable` and `Assignment` and can be declared like:

```
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 Assignment extends Expression {
    protected Assignment(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 implementation of the abstract class `Visitor` has been showed in the Implementation section. The `CalculationVisitor` 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 `ExtendCalculationVisitor` 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(Assignment expr) throws NoSuchMethodException {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        visit(rvalue);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
}
```

Client Code

For example, 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 */ }
}
```

Known Uses

The Reflective Visitor pattern is applied to a compiler framework developed by the authors [5]. This framework is implemented in Java. The code generation part of the compiler framework is implemented using the Reflective Visitor pattern. The code generation operation is started with a direct call to the Visitor. The abstract syntax tree that generated by the syntactical analyzer is passed to the code generation (i.e. the Visitor). The later recursively visit each node in the abstract syntax tree to generate the corresponding code. With the Reflective Visitor pattern, the dispatch action is done by the Visitor itself. The abstract syntax tree includes no accept method and thus it can stand alone, which improve the reusability of the system.

The Reflective Visitor pattern is also used in the design and implementation of an extensible one-pass assembler developed by the authors [4]. This assembler is based on a virtual micro assembly language under a simple virtual processor (SVP) system and is implemented in Java.

Martin E. Nordberg III [8] describes an Extrinsic Visitor pattern, which focuses on breaking the cyclic dependencies between the Visitors and the Elements.

Jens Palsberg and C. Barry Jay [9] use the Java reflection technique in the Visitor pattern to break the double-dispatch between the dynamic linked list and the visitor Walkabout.

Jeremy Blosser [1] and Jeanne Sebring [10] also use the Java reflection to gain the flexibility to extend the object structure (Element hierarchy) in the Visitor pattern.

Related Patterns

Composite pattern [3]: The Reflective Visitor pattern can be used to recursively execute operations over a composite object implemented in the Composite pattern.

Interpreter pattern [3]: The Reflective Visitor pattern can work with the Interpreter pattern to do the interpreter.

GoF Visitor pattern [3] is used to represent an operation to be performed on the elements of an object structure. It is most likely to be used in an environment that this visited object structure is stable.

Vlissides Visitor pattern [12] defines a catch-all operation in the Visitor class to perform the run-time type tests that ensure the correct code generation operations to be invoked. It is best suitable in a situation where occasional extensions are occurred to the visited object structure.

Visser Visitor [11] is a variation on the Vlissides Visitor framework [13]. It defines generic counterparts `AnyVisitor` and `AnyVisiable` for Visitor and Element hierarchies respectively.

Sablecc Visitor pattern [2] allows the visited object structure to be extended without any limitation by performing a downcasting in the object structure. However, this approach introduces a deeper binding between the object structure and the Visitor hierarchy. It is used in situations where reusability of the object structure is not a major concern to the designer.

Acyclic Visitor [7] breaks the cyclic dependency. It allows new Elements to be added without changing the existing classes. This is done by defining 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.

Extrinsic Visitor pattern [8] removes the cyclic dependencies between the object structure and the Visitor hierarchy by defining a *dispatch* method in the Visitor to perform the dispatch action dynamically. Although the Extrinsic Visitor Pattern introduces a more flexible model, adding new Element classes to the object structure is still hard because all related Visitor classes have to redefined. The Extrinsic Visitor Pattern is limited to be implemented under a C++ development environment.

Walkabout Visitor pattern [9] removes the cyclic dependencies between the object structure 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 [1] and Jeanne Sebring [10] Visitor pattern also implement the dispatch action based on Java reflection. They support re-dispatch actions so that a complex multi-level composite hierarchy can be visited. The *accept* method is still used to implement the recursive traversal. Both the Blosser Visitor pattern and the Sebring 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 to encapsulate the implementation details.

Conclusion

The Reflective Visitor pattern improves the extensibility and reusability features upon the earlier implementations of the Visitor pattern. With the power of the reflection technique, the Reflective Visitor pattern can extend the object structure in a much easier way without changing the existing system. The Reflective Visitor pattern can be used in an environment that the implementation language supports reflection and the execution time is not a major concern.

Acknowledgements

We would like to thank Brian Marick, our EuroPLoP '2001 shepherd, and Doug Lea for their valuable comments for this paper.

References

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

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

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

[4] Yun Mai and Michel de Champlain. An Extensible One-Pass Assembler Framework. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, June 2000.

[5] Yun Mai and Michel de Champlain. Design A Compiler Framework in Java. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, November 2000.

[6] Yun Mai and Michel de Champlain. A Pattern Language to Visitors. *The 8th Conference PLoP '2001, Monticello, Illinois, USA*, September 2001.

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

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

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

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

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

[12] John Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, 1998.

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