

Software Design
CSE 335, Spring 2006

Visitors

Stephen Wagner

Michigan State University

Visitors

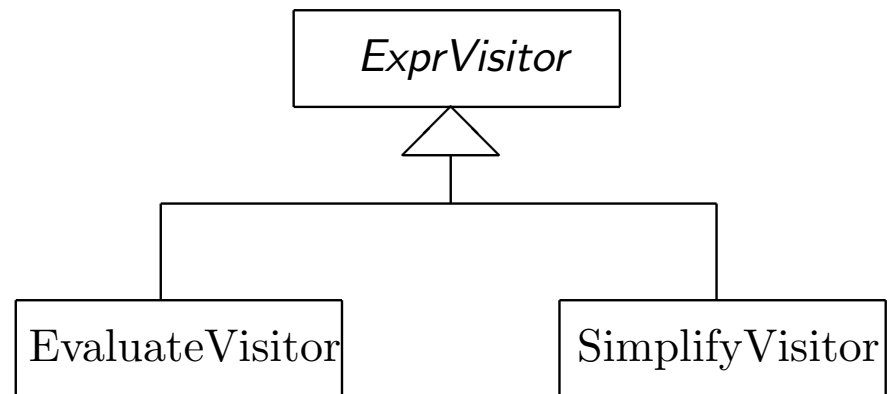
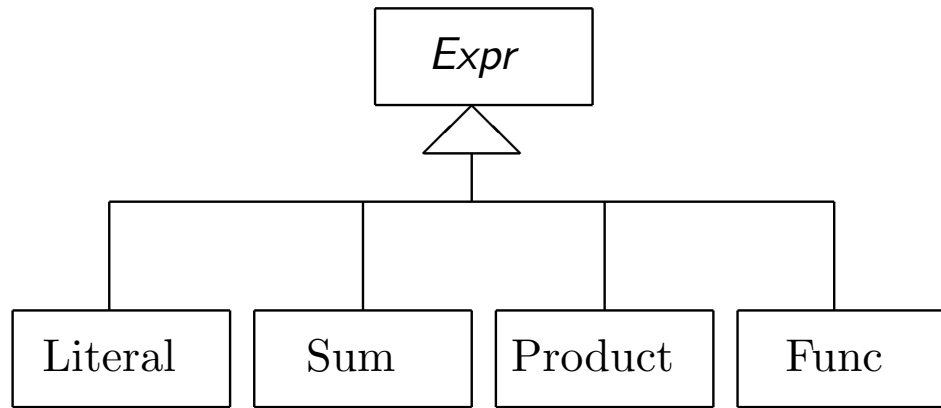
- Suppose we have a complicated expression tree:
 - add, subtract, multiply
 - literals, variables, arrays
 - min, max, arbitrary functions
 - control logic
- We can perform several operations
 - evaluate
 - type check
 - simplify

Visitors

- We would implement this with a hierarchy of nodes.
- Each operations could be implemented by a separate virtual function
- If we want to add an operation, we would have to modify all the existing classes
- The *visitor* pattern is a way to add operations without modify the existing classes

Visitors

- With the visitor pattern you define two class hierarchies
 - one for the elements being operated on (e.g. the expression nodes)
 - one for the visitors that define operations on the elements
- You create a new operation by adding a new class to the visitor class hierarchy
 - This will require writing lots of code
 - However we do not have to modify the expression node hierarchy



Visitors

- Each object structure (e.g. expression tree) will have an associated Visitor class
- The abstract visitor class declares a `VisitConcreteElement` operation for each class defining the object structure (e.g. `Product`).

```
class ExprVisitor {
public:
    virtual void VisitLiteral(Literal *);
    virtual void VisitSum(Sum *);
    virtual void VisitProduct(Product *);
    virtual void VisitFunc(Func *);
    // ....
protected:
    Visitor(); }
```

- We could use operator overloading to give all the `Visit` operations the same name.

Operating Overloading

```
class ExprVisitor {  
public:  
    virtual void Visit(Literal *);  
    virtual void Visit(Sum *);  
    virtual void Visit(Product *);  
    virtual void Visit(Func *);  
    // ....  
protected:  
    Visitor(); }  
}
```

- Be careful!
- All the Visit functions are different

Accept

- Each concrete class implements an Accept operation that calls the matching Visit... operation on the visitor for that concrete class.
- The operation that ends up getting called depends on both the class of the element and the class of the visitor

```
class Expr {
public:
    virtual ~Expr();
    virtual
        void Accept(ExprVisitor&)=0;
protected:
    Expr();
};

class Literal : public Expr {
public:
    Literal(Expr *, Expr *);
    virtual void Accept(ExprVisitor &v)
        { v.VisitLiteral(this); }
};
```

Accept

```
class Sum : public Expr {
public:
    Sum(Expr *, Expr *);
    virtual void Accept(ExprVisitor &);
private:
    Expr *left, *right;
}

void Sum::Accept (ExprVisitor & v)
{
    left->Accept(v);
    right->Accept(v);
    v.VisitSum(this);
}
```

Accept

```
class FuncExpr : public Expr {
public:
    FuncExpr();
    virtual void Accept(ExprVisitor &);
private:
    list<Expr *> parameters;
}

void FuncExpr::Accept (ExprVisitor & v)
{
    for (list<Expr *>::iterator it=parameters.begin();
         it!=parameters.end(); ++it)
        it->Accept(v);
    v.VisitFuncExpr(this);
}
```

Double Dispatch

- The Visitor pattern lets you add operations to classes without changing them
- Visitor achieves this by using a technique called *double-dispatch*
- **single-dispatch:**
 - the operation used to fulfill a request is determined by the name of the request and the type of the receiver
 - `a->request()` ;
 - C++ supports single-dispatch
- **double-dispatch:**
 - the operation used to fulfill a request is determined by the name of the request and the type of *two* receivers
 - key to Visitor pattern

Example

- We will construct an evaluate visitor for the expression trees
 - In this version, the tree structure will be responsible for determining the traversal
 - The visitor will maintain state
- Remember, the advantage of visitors is the ability to add new operations without modifying existing code
- Adding an operation will still require writing a lot of new code

Evaluate Visitor

```
class EvaluateVisitor : public ExprVisitor
{
    public:
        double  GetValue();

        virtual void VisitLiteral(Literal *);
        virtual void VisitSum(Sum *);
        virtual void VisitProduct(Product *);
        virtual void VisitMax(Max *);

    private:
        stack<double> values;
};
```

```
void EvaluateVisitor::VisitLiteral(Literal *l)
{
    values.push(l->value());
}
```

```
void EvaluateVisitor::VisitProduct(Product *l)
{
    double rightval, newval;
    rightval=values.top();
    values.pop();
    newval=values.top()*rightval;
    values.pop();
    values.push(newval);
}
```

```
void EvaluateVisitor::Max(Max *l)
{
    double maxval=values.top;
    values.pop();
    for (int i=1; i<l->parameters.size(); ++i)
    {
        if (values.top()>maxval) maxval=values.top();
        values.pop();
    }
    values.push(maxval);
}
```

Simplify

```
class SimplifyVisitor : public ExprVisitor
{
    public:
        bool check();

        virtual void VisitLiteral(Literal *);
        virtual void VisitSum(Sum *);
        virtual void VisitProduct(Product *);
        virtual void VisitMax(Max *);

    private:
        stack<Expr *> exprs;
};
```

```
void SimplifyVisitor::VisitLiteral(Literal *l)
{
    exprs.push(this->duplicate());
}

void SimplifyVisitor::VisitSum(Sum *l)
{
    Expr *left=exprs.top();
    exprs.pop();
    Expr *right=exprs.top();
    exprs.pop();
    if (left->evaluate(leftvalue) && right->evaluate(rightvalue))
        exprs.push_back(new Literal(leftvalue+rightvalue));
    else if (left->evaluate(leftvalue) && leftvalue==0)
        exprs.push_back(right);
    else if (right->evaluate(rightvalue) && rightvalue==0)
        exprs.push_back(left);
    else
        exprs.push_back(new Sum(left,right)); }
}
```

Visitors

- Visitors make adding new operations easy
 - You do not have to modify existing classes
 - You still have to write the code
- Visitors gather related operations and separates unrelated ones
 - All the code for an operation is defined in a single visitor class
 - Without a visitor, the code for an operation is spread across all the classes in the hierarchy
- Adding a new ConcreteElement class is hard
 - If the class hierarchy is not stable, a Visitor will be more work than it is worth

Visitors

- Visitors are not restricted to a class hierarchy

```
class Visitor {  
public:  
    void VisitTypeOne(TypeOne *);  
    void VisitTypeTwo(TypeTwo *);  
}
```

- TypeOne and TypeTwo do not have to belong to the same hierarchy
- Visitors can have state

Variations

- Who is responsible for traversing the structure?
 - The components of the structure
 - The visitor
- In previous examples, the nodes of the expression tree were responsible for traversing the tree
- What are the advantages and disadvantages of the different methods?

Advantages and Disadvantages

- Method One
 - The element classes already have information about structure
 - Only one traversal can be easily defined
- Method Two
 - The visitor will need to be able to access structure information
 - Simplifies Accept
 - Each visitor can define its own traversal

Binary Expressions Again

- This is actually much simpler if the visitor is responsible for the traversal.
- The visitor works by creating additional visitors.

```
class EvaluateVisitor : public ExprVisitor
{
    public:
        double  GetValue() { return value; }

        virtual void Visit(Literal *);
        virtual void Visit(Sum *);
        .....

    private:
        double value;
};
```

```
class Expr {
public:
    virtual ~Expr();
    virtual
        void Accept(ExprVisitor&)=0;
protected:
    Expr();
};
```

```
void FuncExpr::Accept (ExprVisitor & v)
{
    v.Visit(this);
}
```

```
class Literal : public Expr {
public:
    Literal(Expr *, Expr *);
    virtual void Accept(ExprVisitor &v)
        { v.Visit(this); }
};
```

```
void EvaluateVisitor::Visit(Literal *l)
{
    value=l->value();
}
```

```
void EvaluateVisitor::Visit(Product *l)
{
    EvaluateVisitor rightvisitor, leftvisitor;
    l->leftChild()->Accept(leftvisitor);
    l->rightChild()->Accept(rightvisitor);
    value=leftvisitor.GetValue()*rightvisitor.GetValue();
}
```

```
class Expr {
public:
    virtual ~Expr();
    virtual
        void Accept(ExprVisitor&)=0;
protected:
    Expr();
};
```

```
class FuncExpr : public Expr {
public:
    FuncExpr();
    virtual void Accept(ExprVisitor &)
        { v.Visit(this); }
private:
    list<Expr *> parameters;
}
```

```
class Literal : public Expr {
public:
    Literal(Expr *, Expr *);
    virtual void Accept(ExprVisitor &v)
        { v.Visit(this); }
};
```

Does this work?

```
class Expr {
public:
    virtual ~Expr();
    virtual
        void Accept(ExprVisitor& v)
            { v.Visit(this); }
protected:
    Expr();
};
```

Print Visitor

- The print visitor does not need any state (or more precisely, the state is maintained by the ostream).
- This is simple if the visitor is responsible for traversals.

```
void PrintVisitor::VisitLiteral(Literal *l)
{
    s << l->value();
}
```

```
void PrintVisitor::VisitProduct(Product *l)
{
    s << "(";
    l->leftChild()->Accept(this);
    s << "*";
    l->rightChild()->Accept(this);
    s << ")";
}
```

```
class PrintVisitor : public ExprVisitor
{
    public:
        virtual void Visit(Literal *);
        virtual void Visit(Sum *);
        .....

    private:
        ostream & s;
};
```

Simplify Visitor

```
class SimplifyVisitor : public ExprVisitor
{
    public:
        Expr *  GetExpr() { return theexpr; }

        virtual void Visit(Literal *);
        virtual void Visit(Sum *);
        .....
    protected:
        Expr *theexpr;
};
```

Simplify Visitor

```
void SimplifyVisitor::Visit(Sum *l)
{
    SimplifyVisitor rvisitor, lvisitor;
    l->leftChild()->Accept(lvisitor);
    l->rightChild()->Accept(rvisitor);
    if (lvisitor.GetExpr()->evaluate(lval) &&
        rvisitor.GetExp()->evaluate(rval))
        expr=new Literal(lval+rval);
    else if (lvisitor.GetExpr()->evaluate(lval) && lval==0)
        expr=new rvisitor.GetExpr();
    else if (rvisitor.GetExpr()->evaluate(rval) && rval==0)
        expr=new rvisitor.GetExpr();
    else
        expr=new Sum(lvisitor.GetExpr(),rvisitor.GetExpr())
}
```