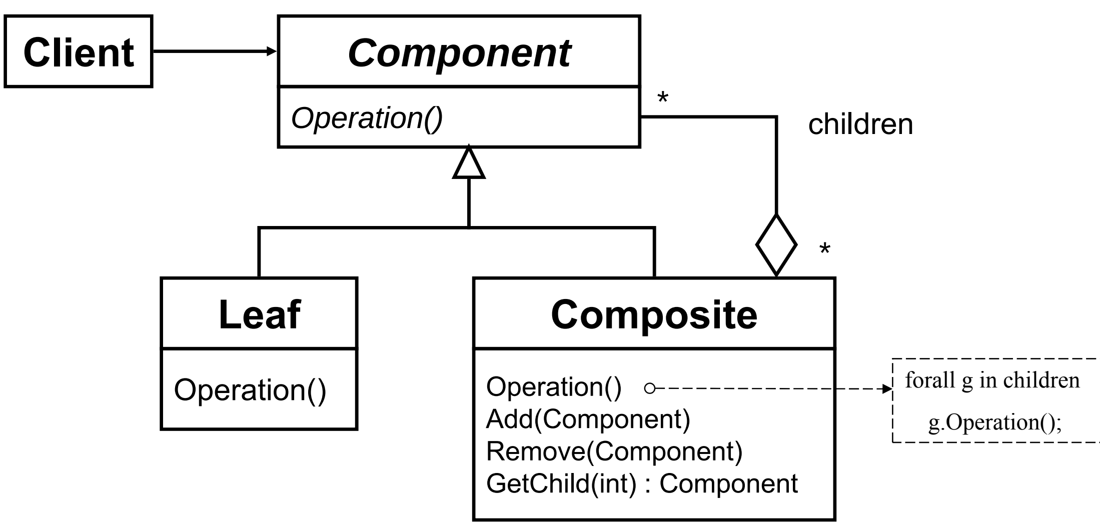
# Visitor Pattern

**Alex X. Liu**

# Review Composite Pattern




**Question: How to add a new operation?**

# Operations and Classes

- Reality: Is it possible to add new functions without changing an organization?
  - Yes. Method: outsourcing.
  - MSU wants to keep the buildings clean. There are two options:
    - 1. Hire and manage own janitors.
      - MSU needs to add them to the payroll system ← changing MSU existing system.
    - 2. Outsource to a company that specializes on office cleaning.
      - MSU can add any new functions by this outsourcing pattern.
- Software: Is it possible to add new operations to some classes without changing them?
  - Yes. Method: Visitor Pattern.
  - Idea: group the same operations into one class.
    - Like building a company that specializes on office cleaning.
  - Some classes have common operations.
    - Just like both MSU and UM need office cleaning.
- Changing a class vs. adding a new class:
  - We should avoid changing existing classes, which have been tested and used, as much as possible. Changing a class is error-prone and expensive.
  - Old classes have been tested. Don't touch them. Adding a new class means that you only need to test the new class.



# Recipe – Element and ConcreteElement

```
class Graphic{
public:
   virtual void Accept(Visitor*) = 0;
};

class Circle : public Graphic{
public:
   virtual void Accept(Visitor* v) { v->VisitCircle(this); }
};

class Picture: public Graphic{
public:
   virtual void Accept(Visitor* v) { v->VisitPicture(this);}
}
```

# Recipe – Visitor and ConcreteVisitor

```
class Circle, Picture; //Forward declaration
class Visitor {
public:
   virtual void VisitCircle(Circle*)=0;
   virtual void VisitPicture(Picture*)=0;
};
```

```
class PrintVistor : public Visitor{
protected:
   //state variables for storing intermediate results. For example, a stack, for a
    tree visitor.
public:
   virtual void VisitCircle(Circle* cp) {/may store something in state variables; };
   virtual void VisitPicture(Picture* pp) {
      forall children g do g->Accept(this);
      //may change state variables values based on their value};
   getVisitResult() {…};
};
```

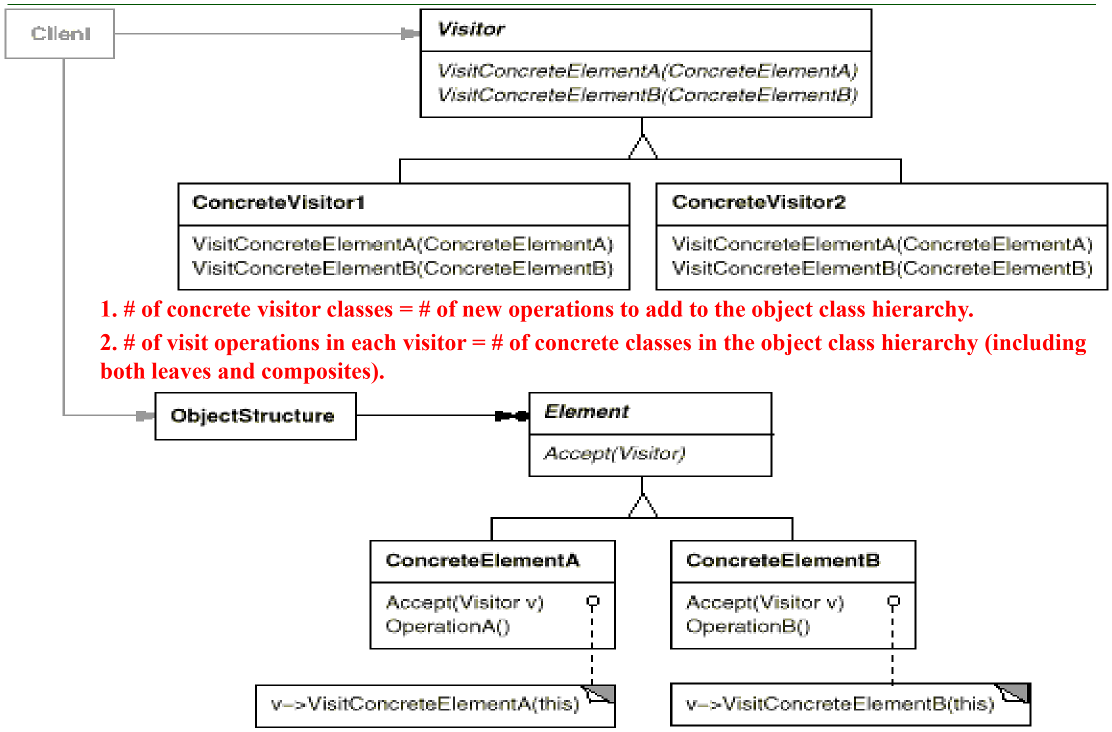


# Hooking Up

- Two class hierarchies
  - Object class hierarchy
  - Visitor class hierarchy
- Hooking up at run time:

```
// Create object trees
Circle aCircle;
Line aLine;
Rectangle aRec;
Picture pic1, pic2;
pic2.addChild (&aCircle);
pic2.addChild(&aLine);
pic2.addChild(&pic1);
pic2.addChild(&aRec);
// Create a PrintVistor objector
 PrintVistor pv;
// Hook up at run time
pic2.Accept(&pv);
```

# UML Diagram




1. # of concrete visitor classes = # of new operations to add to the object class hierarchy.
2. # of visit operations in each visitor = # of concrete classes in the object class hierarchy (including both leaves and composites).



# Applicability of Visitor Pattern

- Use the Visitor pattern when you want to add new operations without changing existing classes.
  - The classes defining the object structure rarely change, but you may want to define new operations over the structure.

# Tips

- For each class in the object class hierarchy, two important things:
  - 1. Add a hook for visitors: void Accept(Visitor*)
  - 2. Provide methods to access its data members.
    - If you outsource cleaning job to janitors, you have to give them keys to rooms.
- Thin ConcreteComposite, Fat ConcreteVisitor: In the Accept function of a ConcreComposite class, don't put any other code other than v->VisitConcreteCompositeA(this).
  - Always: virtual void Accept(Visitor* v) { v->VisitConcreteCompositeA(this);}
  - Reason 1: You may want to change the way that you visit the children!
  - Reason 2: Different ConcreteVisitors may visit children in different ways!
    - Preorder traversal, inorder traversal, postorder traversal
  - The visitor pattern example in the Gamma book is not recommended.
- In the ConcreteVisitor, a stack may be useful in storing state information of the visit.
- You need forward declaration to break circular dependency.
- Reading assignment: Gamma book "Visitor" chapter



# Example

- Design classes for representing a tree
  - Composite pattern (terminal node, nonterminal node)
- Design a visitor for a tree
  - For example, calculate the sum.
  - Assuming that non-terminal nodes have no values.

# Stack Based Implementation

```
class ConcreteVisitor: public Visitor{
protected:
   stack<int> m_stack;
public:
   virtual void visitTerminalNode( TerminalNode* trn) {
      m_stack.push( trn->getValue() ); };
   virtual void visitNonTerminalNode( NonTerminalNode* ntrn ) {
      //Visit every children, store state information in m_stack.
      for(int i=0; i< ntrn->getChildrenSize(); i++){
         ntrn->getChildren(i)->Accept(this);
      }
      //Get state information from m_stack, do calculation.
      int sum = 0;
      for(int i=0; i<ntrn->getChildrenSize(); i++ ){
         sum += m_stack.top();
          m_stack.pop();
      }
      //Store state information in m_stack.
     m_stack.push(sum);
      };
     int getResult() {
         int result =m_stack.top(); m_stack.pop(); return result;};
};
```

avoid side effect of one round of visit on another round of visit



# Non-stack Based Implementation

```
class ConcreteVisitor: public Visitor{
protected:
   int sum;
public:
   void ConcreteVisitor(){ sum=0;};

   virtual void visitTerminalNode( TerminalNode* trn) { sum += trn->getValue() ; };

   virtual void visitNonTerminalNode( NonTerminalNode* ntrn ) {
      for(int i=0; i< ntrn->getChildrenSize(); i++){
         ntrn->getChildren(i)->Accept(this);
      }
   };

   int getResult() {
      int result=sum;
      sum=0;
      return result;
   };
};
```