The Visitor Pattern

Motivating Example

Douglas C. Schmidt
Learning Objectives in This Lesson

- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.

- **Operation 1:** Print all values of nodes in tree
- **Operation 2:** Evaluate “yield” of nodes in tree
- **Operation 3:** Conduct semantic analysis of tree
- **Operation 4:** ...

```
5

- 

× 

+ 

3

4
```
Motivating the Need for the Visitor Pattern in the Expression Tree App

Douglas C. Schmidt
**Purpose:** Perform an extensible set of operations on an expression tree without requiring any changes to the tree itself.

**Visitor decouples expression tree structure from operations performed on it.**
Context: OO Expression Tree Processing App

- Adding new operations to an expression tree should require no changes to the tree’s structure & implementation.

Operation 1: Print all values of nodes in tree

Operation 2: Evaluate “yield” of nodes in tree

Operation 3: Conduct semantic analysis of tree

Operation 4: ...

Expression Tree:

- 5
- 3
+ 4
×
Problem: Non-Extensible Tree Operations

- Hard-coding operations in `Expression_Tree` or in `Component_Node` subclasses limits extensibility.
- e.g., adding new operations would violate the *Open/Closed Principle* since the `Expression_Tree` class API would change.

```cpp
class Expression_Tree {
    ...  
    Expression_Tree (Component_Node *root):
                : root_(root) {
    }
    ...  
    void print();
    int evaluate();
    ...  
```

Problem: Non-Extensible Tree Operations

- Hard-coding `dynamic_cast` to access `Expression_Tree` nodes limits extensibility.

```cpp
Expression_Tree expr_tree = ...;

cout << "Tree contents:" << endl;

for (auto iter = tree.begin(order); iter != tree.end(order); ++iter) {
    Expression_Tree node = *iter;
    if (dynamic_cast<Leaf_Node*>(node.get_root()))
        cout << (int) node.item() + " ";
    else
        cout << (char) node.item() + " ";
}
```

See earlier lesson on “The Iterator Pattern: Implementation in C++”
Problem: Non-Extensible Tree Operations

- Hard-coding `dynamic_cast` to access `Expression_Tree` nodes limits extensibility.

```cpp
Expression_Tree expr_tree = ...;

cout << "Tree contents:" << endl;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter) {
    Expression_Tree node = *iter;
    if (dynamic_cast<Leaf_Node *> (node.get_root()))
        cout << (int) node.item() + " ";
    else
        cout << (char) node.item() + " ";
}
```

*Code like this will cause trouble at some point since dynamic downcasting leads to maintainability & readability concerns.*

See [www.aristeia.com/EC3E/3E_item27.pdf](http://www.aristeia.com/EC3E/3E_item27.pdf)
Solution: Decouple Operations From Structure

- Create a hierarchy of visitors that define overloaded `visit()` methods to perform operations on each expression tree node implementation.
• Define an `accept()` method in the `Expression_Tree` class API that is passed an instance of a visitor implementation.

The `accept()` method on `Expression_Tree` forwards to the `accept()` method in the `Component_Node` implementation.

See earlier lesson on “The Bridge Pattern: Motivating Example”
During an iteration over the expression tree call `accept()` on each node & pass in the visitor instance, e.g.,

```cpp
Visitor &print_visitor = visitor_factory.make_visitor("print");
```

Use a factory method to create a print visitor

```cpp
Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");
```

```cpp
for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
    it->accept(print_visitor);
```

See earlier lesson on "The Factory Method Pattern!"
Solution: Decouple Operations From Structure

• During an iteration over the expression tree call accept() on each node & pass in the visitor instance, e.g.,

Visitor &print_visitor = visitor_factory.make_visitor("print");

Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");

Apply a Creational pattern to make an expression tree

for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
    it->accept(print_visitor);

See earlier lessons on “The Interpreter Pattern” & “The Builder Pattern”
Solution: Decouple Operations From Structure

- During an iteration over the expression tree call \texttt{accept()} on each node & pass in the visitor instance, e.g.,

\begin{verbatim}
Visitor &print_visitor = visitor_factory.make_visitor("print");

Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");
\end{verbatim}

\begin{verbatim}
for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
    it->accept(print_visitor);
\end{verbatim}

Apply a Creational pattern to make iterator for an expression tree

See earlier lesson on "The Factory Method Pattern!"
Solution: Decouple Operations From Structure

• During an iteration over the expression tree call `accept()` on each node & pass in the visitor instance, e.g.,

Visitor &print_visitor = visitor_factory.make_visitor("print");

Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");

for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
    it->accept(print_visitor);

The `accept()` method on `Expression_Tree` forwards to the `accept()` method in the `Component_Node` implementation.
Solution: Decouple Operations From Structure

- Have `accept()` call back to the `visitor.visit()` method, passing in the corresponding node in the expression tree to perform the operation, e.g.,

```cpp
class Leaf_Node : public Component_Node {
    void accept(Visitor &visitor) {
        visitor.visit(*this);
    }
    ...
}
```

This indirection & “double dispatching” avoids hard-coding operations into expression tree nodes & also eliminates the need for downcasts.

[en.wikipedia.org/wiki/Double_dispatch](en.wikipedia.org/wiki/Double_dispatch) has more info on double dispatching.
Solution: Decouple Operations From Structure

• Have `accept()` call back to the `visitor.visit()` method, passing in the corresponding node in the expression tree to perform the operation, e.g.,

```cpp
class Leaf_Node : public Component_Node {
    void accept(Visitor &visitor)
    {
        visitor.visit(*this);
    }
    ... 
}
```

Method overloading by `Component_Node` subclasses is “static polymorphism” that eliminates the need for ugly downcasts.

See [mrbool.com/static-polymorphism-in-java/29706](mrbool.com/static-polymorphism-in-java/29706)
Visitor Abstract Base Class Overview

- Specifies an extensible set of operations that can be performed on each subclass of Component_Node in an expression tree

**Class methods**

```cpp
virtual void visit(const Leaf_Node &) = 0
virtual void visit(const Composite_Negate_Node &) = 0
virtual void visit(const Composite_Add_Node &) = 0
virtual void visit(const Composite_Subtract_Node &) = 0
virtual void visit(const Composite_Divide_Node &) = 0
virtual void visit(const Composite_Multiply_Node &) = 0
```
Visitor Abstract Base Class Overview

- Specifies an extensible set of operations that can be performed on each subclass of Component_Node in an expression tree.

**Class methods**

An overloaded visit() method is defined by each subclass of Component_Node.

```cpp
virtual void visit(const Leaf_Node&) = 0
virtual void visit(const Composite_Negate_Node&) = 0
virtual void visit(const Composite_Add_Node&) = 0
virtual void visit(const Composite_Subtract_Node&) = 0
virtual void visit(const Composite_Divide_Node&) = 0
virtual void visit(const Composite_Multiply_Node&) = 0
```
Visitor Abstract Base Class Overview

• Specifies an extensible set of operations that can be performed on each subclass of Component_Node in an expression tree

Class methods

```cpp
virtual void visit(const Leaf_Node &) = 0
virtual void visit(const Composite_Negate_Node &) = 0
virtual void visit(const Composite_Add_Node &) = 0
virtual void visit(const Composite_Subtract_Node &) = 0
virtual void visit(const Composite_Divide_Node &) = 0
virtual void visit(const Composite_Multiply_Node &) = 0
```

• **Commonality**: provides a common set of visit() methods, one for each subclass of Component_Node

• **Variability**: Subclasses of this interface define specific behaviors for different types of visitors
Visitor Implementation Hierarchy Overview

- A class hierarchy that defines operations performed on implementations of `Component_Node` in an expression tree

Visitor subclasses define operations rather than the Expression_Tree API.

- `Visit(const Leaf_Node &)`
- `Visit(const Composite_Add_Node &)`
- `Visit(const Composite_Divide_Node &)`

...