Overview of the Expression Tree Processing App Case Study (Part 1)

Douglas C. Schmidt
Learning Objectives in This Lesson

- Understand the goals of the object-oriented (OO) expression tree case study.
Learning Objectives in This Lesson

- Understand the goals of the object-oriented (OO) expression tree case study.
- Recognize the key behavioral & structural properties in the expression tree domain.
Lesson Introduction
Lesson Introduction

• While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities.

“Sitting & thinking” is not sufficient...
Lesson Introduction

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities.
- Instead, it’s usually better to see how patterns can help improve nontrivial programs.
Lesson Introduction

• While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities.

• Instead, it’s usually better to see how patterns can help improve nontrivial programs, e.g.,
  • Easier to write & read;
  • Easier to maintain & modify;
  • More efficient & robust.
Lesson Introduction

• While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities.
• Instead, it’s usually better to see how patterns can help improve nontrivial programs.
• This lesson describes a realistic—yet tractable—expression tree processing app we’ll use as a case study throughout the course.
Lesson Introduction

• While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities.

• Instead, it’s usually better to see how patterns can help improve nontrivial programs.

• This lesson describes a realistic—yet tractable—expression tree processing app we’ll use as a case study throughout the course.

• This case study applies many “Gang of Four” (GoF) patterns using C++ & STL.

See en.wikipedia.org/wiki/Design_Patterns
Expression Tree Processing App
Case Study Goals
Expression Tree Processing App Case Study Goals

- Develop an OO expression tree processing app using patterns & frameworks.

<table>
<thead>
<tr>
<th>Design Problem</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-extensible &amp; error-prone designs</td>
<td>Composite</td>
</tr>
<tr>
<td>Minimizing impact of variability</td>
<td>Bridge</td>
</tr>
<tr>
<td>Inflexible expression input processing</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Inflexible interpreter output</td>
<td>Builder</td>
</tr>
<tr>
<td>Scattered request implementations</td>
<td>Command</td>
</tr>
<tr>
<td>Inflexible creation of variabilities</td>
<td>Factory Method</td>
</tr>
<tr>
<td>Inflexible expression tree traversal</td>
<td>Iterator</td>
</tr>
<tr>
<td>Obtrusive behavior changes</td>
<td>Strategy</td>
</tr>
<tr>
<td>Non-extensible tree operations</td>
<td>Visitor</td>
</tr>
<tr>
<td>Incorrect user request ordering</td>
<td>State</td>
</tr>
<tr>
<td>Non-extensible operating modes</td>
<td>Template Method</td>
</tr>
<tr>
<td>Minimizing global variable liabilities</td>
<td>Singleton</td>
</tr>
</tbody>
</table>

Naturally, these patterns apply to more than expression tree processing apps!
Expression Tree Processing App Case Study Goals

- This app uses expression trees to remove ambiguity in algebraic expressions.

See en.wikipedia.org/wiki/Binary_expression_tree
Expression Tree Processing App Case Study Goals

- Compare/contrast algorithmic decomposition & object-oriented (OO) approaches.

Despite decades of OO focus, algorithmic decomposition is still surprisingly common.

**Algorithmic Decomposition**

- **Tree Node**
  - 0|1|2

**Pattern- & Object-Oriented Decomposition**

- **Expression_Tree**
  - Component_Node
  - Composite_Unary_Node
  - Leaf_Node
  - Composite_Binary_Node
  - Composite_Negate_Node
  - Composite_Add_Node
  - Composite_Subtract_Node
  - Composite_Multiply_Node
  - Composite_Divide_Node

Expression Tree Processing App Case Study Goals

- Demonstrate *scope, commonality, & variability* (SCV) analysis as a means to achieve systematic software reuse.

  - **Identify common elements of a domain & define stable interfaces.**
  - **Identify variable elements of a domain & define stable interfaces.**

See [www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf)
Expression Tree Processing App Case Study Goals

- Demonstrate *scope, commonality, & variability* (SCV) analysis as a means to achieve systematic software reuse.
- Apply SCV in the context of the expression tree processing app.
Expression Tree Processing App Case Study Goals

• Show how to implement pattern-oriented OO frameworks & generic programs in C++.

```cpp
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
    iter != tree.end(order);
    ++iter)
    (*iter).accept(print_visitor);

for_each(tree.begin(order),
         tree.end(order),
         [&print_visitor](auto tree) {
             tree.accept(print_visitor);
         });
```
Expression Tree Processing App Case Study Goals

- Show how to implement pattern-oriented OO frameworks & generic programs in C++.

```cpp
Expression_Tree tree = ...;
Visitor print_visitor = ...;
for (auto iter = tree.begin (order);
    iter != tree.end (order);
    ++iter)
    (*iter).accept (print_visitor);

for_each(tree.begin(order),
    tree.end(order),
    [&print_visitor](auto tree) {
        tree.accept (print_visitor);
    });
```

C++-style GoF *Iterator* pattern with for loop

See [en.wikipedia.org/wiki/Iterator_pattern](en.wikipedia.org/wiki/Iterator_pattern)
Show how to implement pattern-oriented OO frameworks & generic programs in C++.

Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin (order);
     iter != tree.end (order);
     ++iter)
(*iter).accept (print_visitor);

for_each(tree.begin(order),
         tree.end(order),
         [&print_visitor](auto tree) {
             tree.accept (print_visitor);
         });
Overview of the Expression Tree Processing Domain

Douglas C. Schmidt
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing operators & operands.

See en.wikipedia.org/wiki/Binary_expression_tree for expression tree information.
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing *operators* & *operands*.
  - Operators are *interior nodes* in the tree, i.e.,
    - *Binary* & *unary nodes*
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing *operators & operands*.
  - Operators are *interior nodes* in the tree
  - Operands are *exterior nodes* in the tree, i.e.,
    - *Leaf nodes*
Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities

See en.wikipedia.org/wiki/Operator_associativity & en.wikipedia.org/wiki/Arity
Operators have different precedence levels, different associativities, & different arities, e.g.,
- Precedence defines which operator to perform first to evaluate a mathematical expression.
- Multiplication takes precedence over addition.

Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities, e.g.,
  - Precedence defines which operator to perform first to evaluate a mathematical expression.
  - Multiplication takes precedence over addition
  - Operator locations in a tree unambiguously designate precedence

Example: 3 + 4 is performed before −5 × 7.
Operators have different precedence levels, different associativities, & different arities, e.g.,

- Precedence defines which operator to perform first to evaluate a mathematical expression.
- Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses.

\[
5 + 3 - 4 = (5 + 3) - 4
\]

See [en.wikipedia.org/wiki/Operator_associativity](en.wikipedia.org/wiki/Operator_associativity)
Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities, e.g.,
  - Precedence defines which operator to perform first to evaluate a mathematical expression.
  - Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses.
  - Arity defines the number of operands an operator takes.
    - Multiplication & addition operators have two arguments (arity == 2)

See [en.wikipedia.org/wiki/Arity](en.wikipedia.org/wiki/Arity)
• Operators have different precedence levels, different associativities, & different arities, e.g.,

• Precedence defines which operator to perform first to evaluate a mathematical expression.

• Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses.

• Arity defines the number of operands an operator takes.
  • Multiplication & addition operators have two arguments (arity == 2)
  • The unary minus operator has one argument (arity == 1)
Overview of Expression Tree Processing Domain

- Operands can be integers, doubles, variables, etc.
- We'll just handle integers in this case study.

![Expression Tree Diagram]

- Binary Nodes
- Unary Node
- Leaf Nodes (Integers)
Overview of Expression Tree Processing Domain

- Operands can be integers, doubles, variables, etc.
  - We'll just handle integers in this case study.
  - It’s easy to extend the app to handle other types.
Trees may be “evaluated” via different traversal orders, e.g.,

- “In-order traversal” = \(-5 \times (3+4)\)
- “Pre-order traversal” = \(\times -5 + 34\)
- “Post-order traversal” = \(5 - 34 + \times\)
- “Level-order traversal” = \(\times + 534\)

See [en.wikipedia.org/wiki/Binary_expression_tree#Traversal](en.wikipedia.org/wiki/Binary_expression_tree#Traversal) for more information.