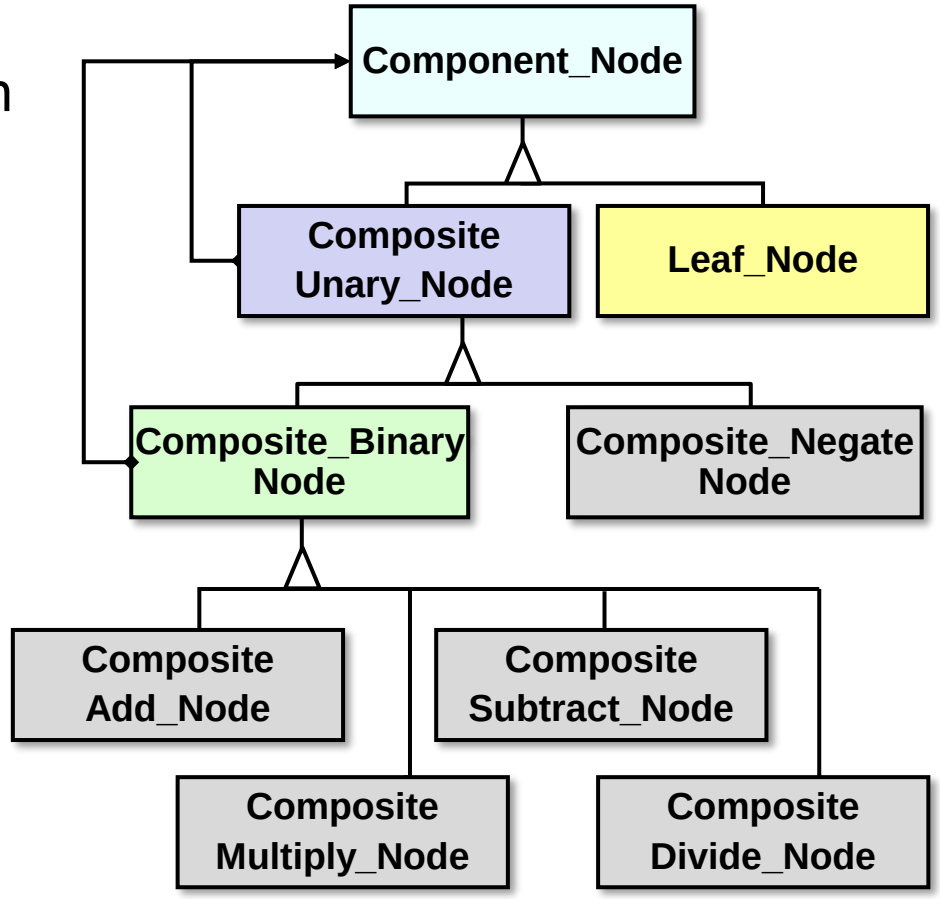
The Composite Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Composite* pattern can be applied to make the expression tree object structure more uniform & extensible.

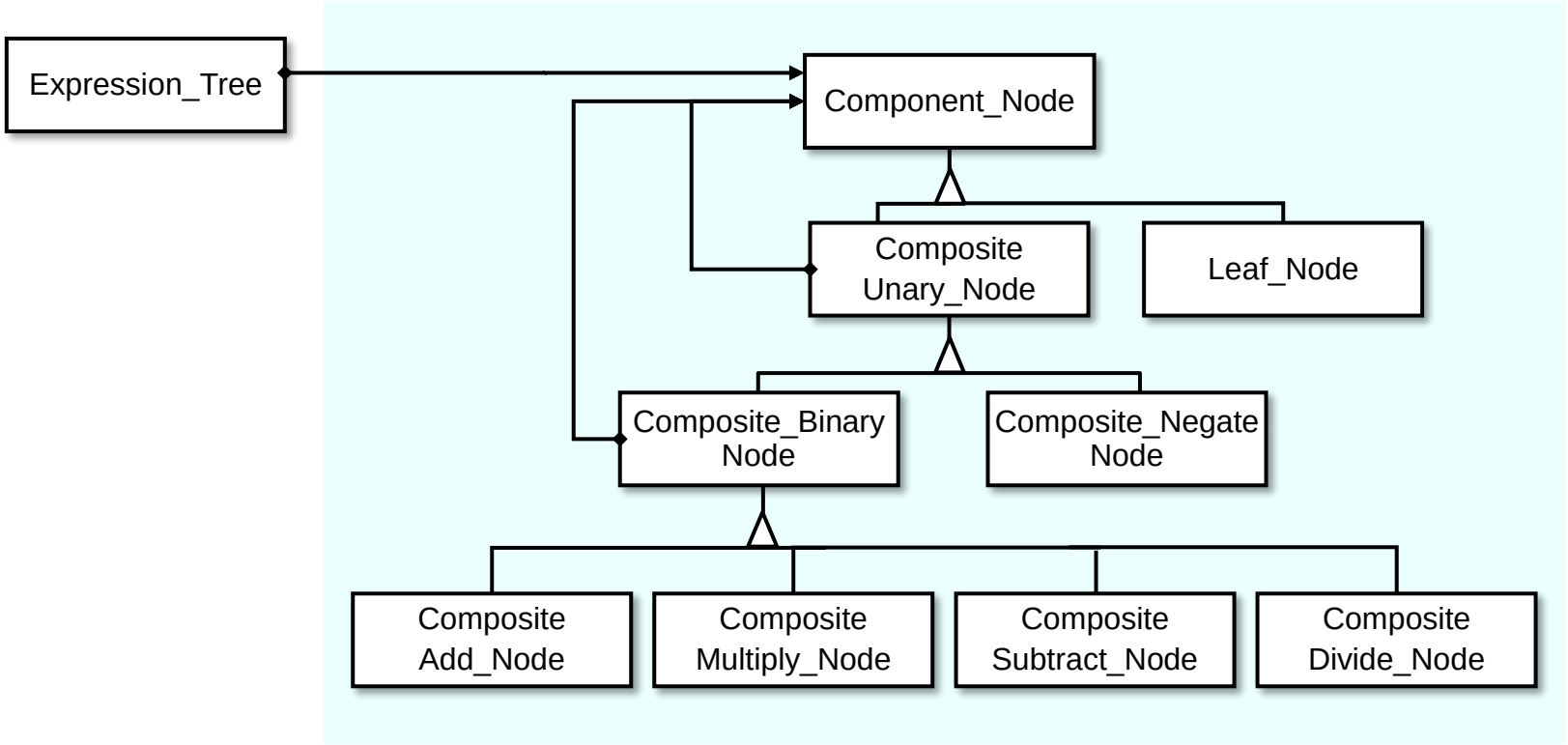


Douglas C. Schmidt

Motivating the Need for the Composite Pattern in the Expression Tree App

A Pattern for Structuring the Expression Tree

***Purpose**: Define the key internal data structure for the expression tree.*

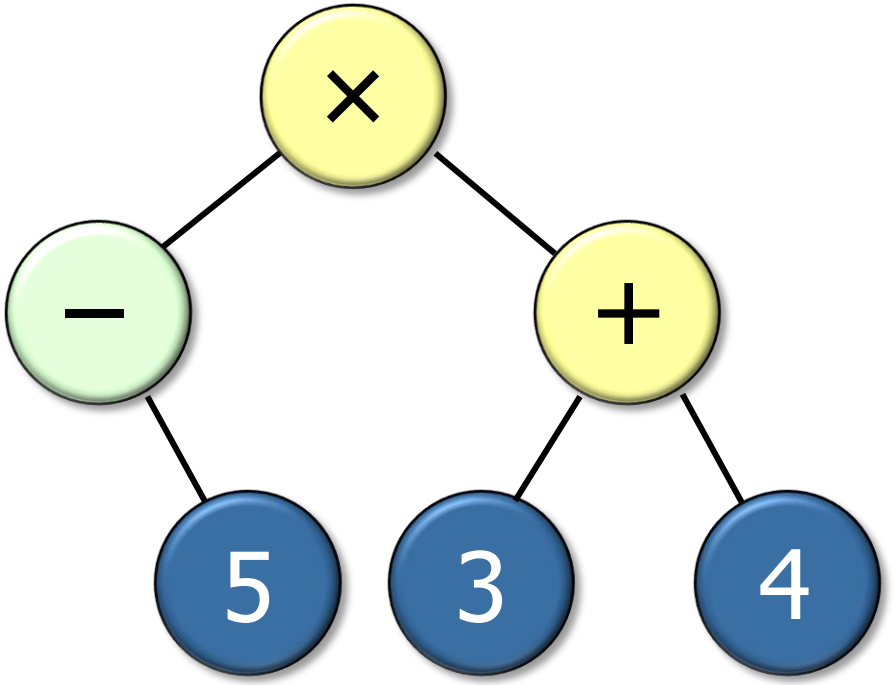


Composite

Composite simplifies adding new types of nodes (& new node operations).

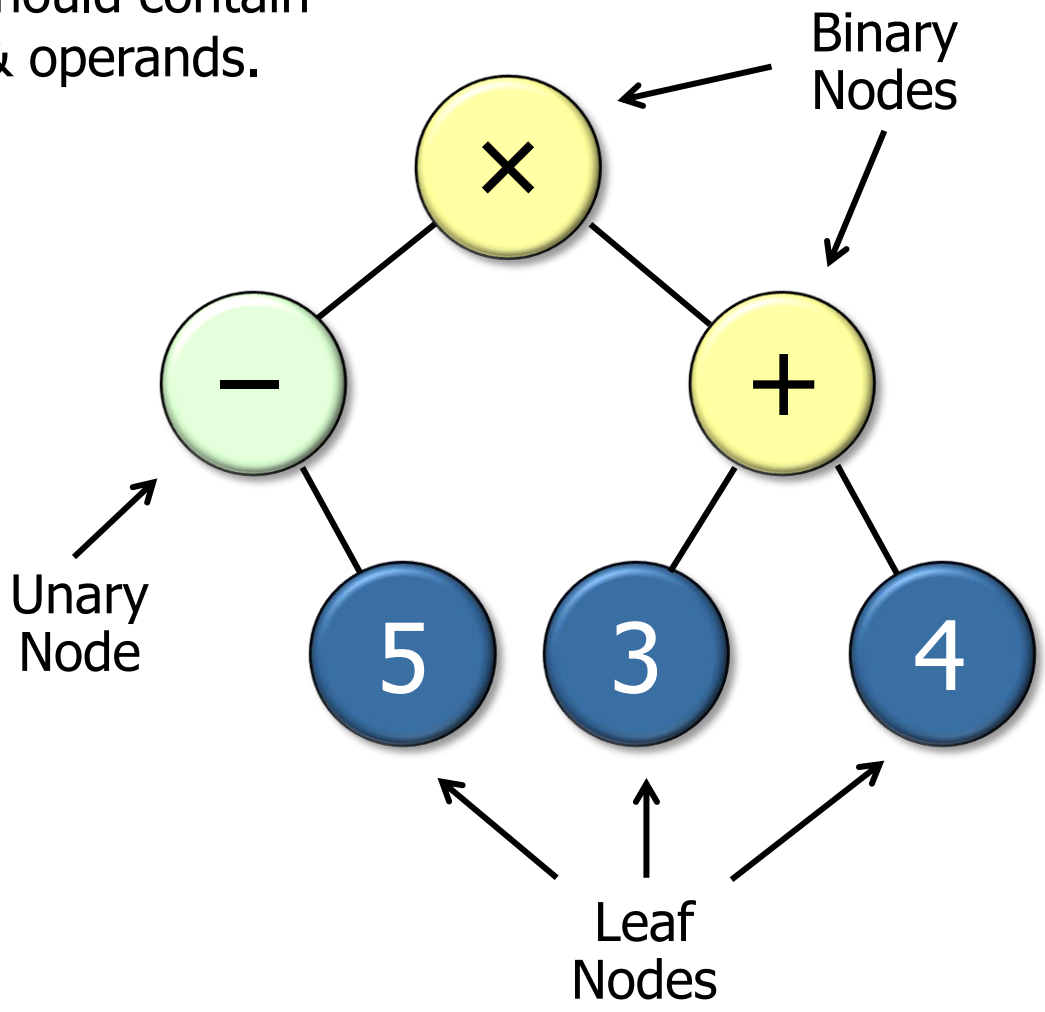
Context: OO Expression Tree Processing App

- The design of an expression tree should reflect its "physical" structure.



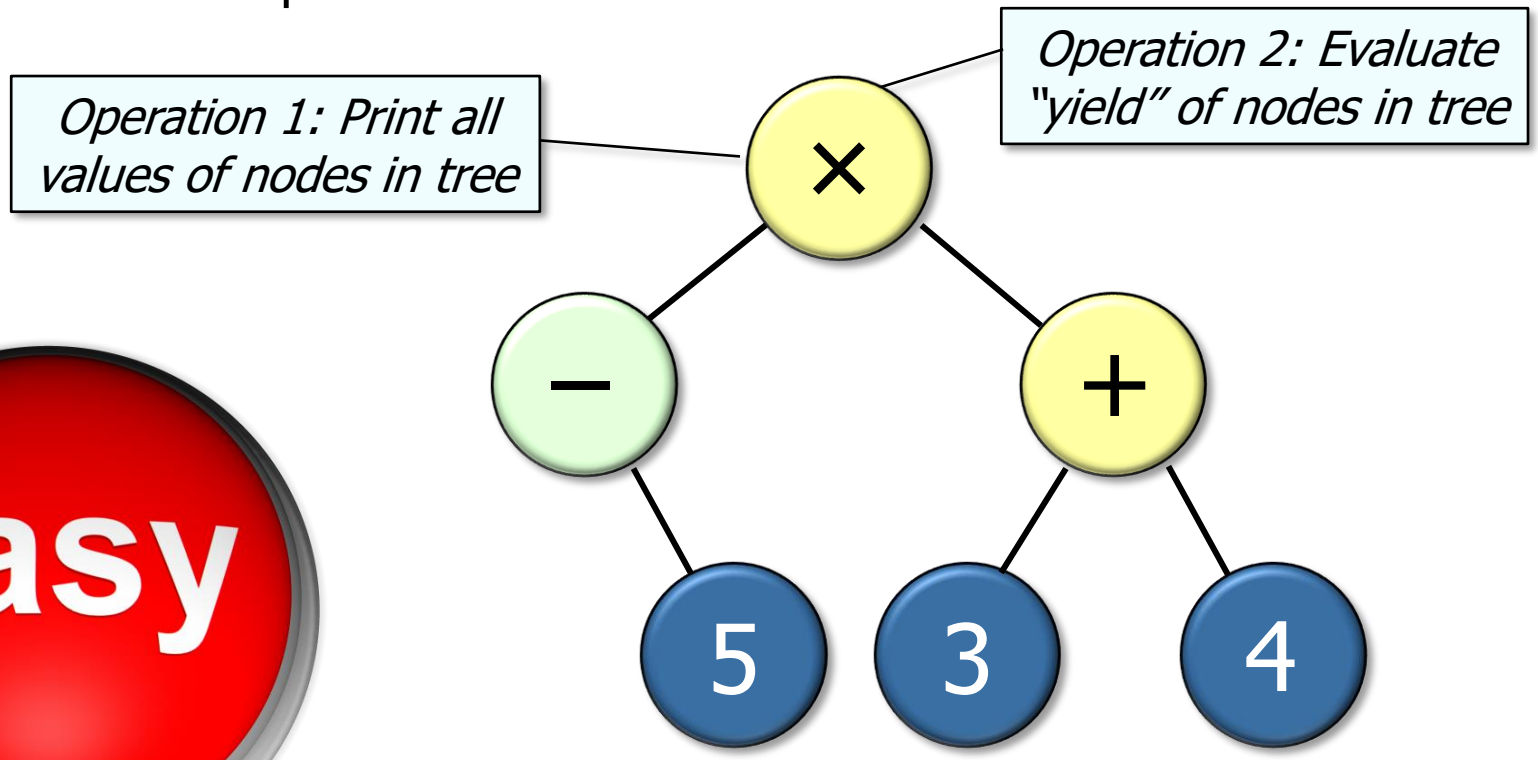
Context: OO Expression Tree Processing App

- The design of an expression tree should reflect its "physical" structure.
  - e.g., the tree structure should contain binary/unary operators & operands.



Context: OO Expression Tree Processing App

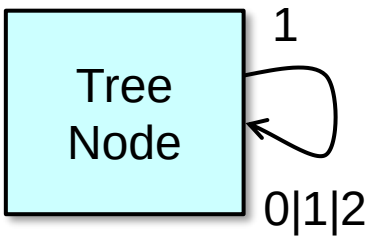
- Adding new operations on tree nodes should require little/no modifications to the tree's structure & implementation.



See upcoming lesson on "The *Visitor* Pattern."

Problem: Non-Extensible & Error-Prone Designs

- Tightly coupling expression tree data structures & functionality impedes extensibility.

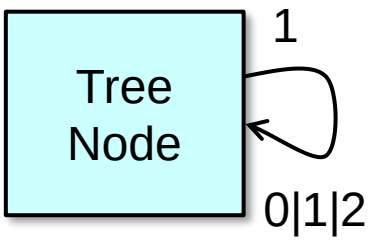


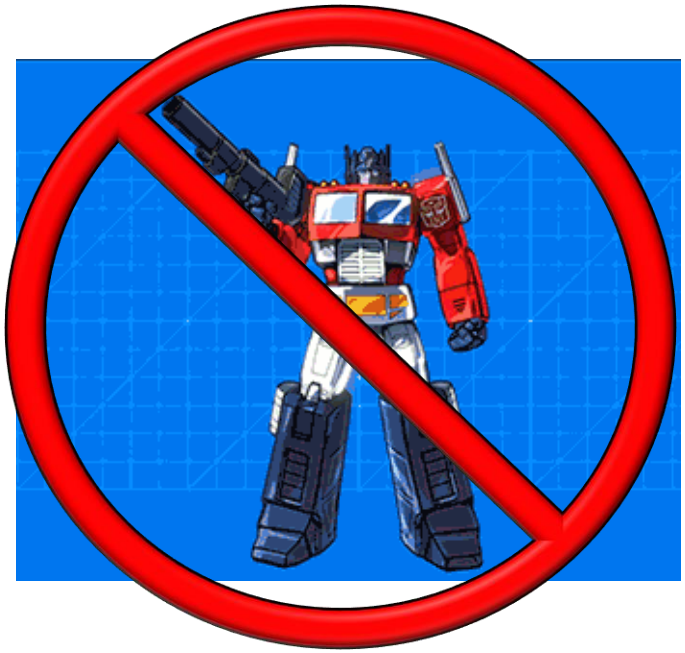

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[3]; int num_;
  } o_;
#define num_ o_.num_
#define op_  o_.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
             *r_;} binary_;
  } c_;
#define unary_ c_.unary_
#define binary_ c_.binary_
} Tree_Node;
```

See lesson on "*Evaluating the Algorithmic Decomposition of the Expression Tree Processing App*"

Problem: Non-Extensible & Error-Prone Designs

- Tightly coupling expression tree data structures & functionality impedes extensibility.



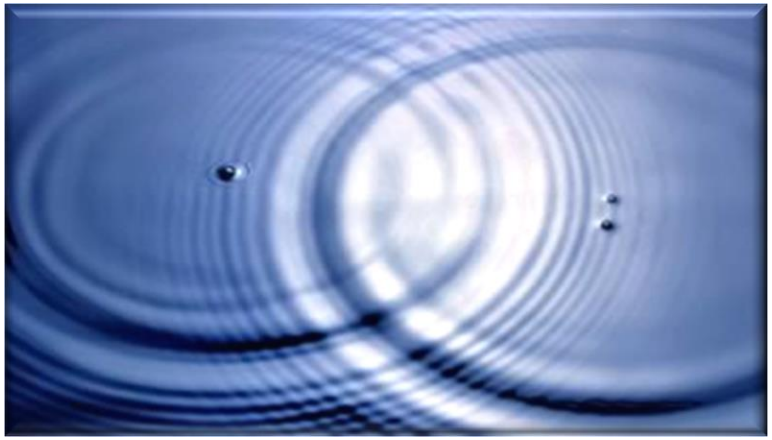


```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[3]; int num_;
  } o_;
#define num_ o_.num_
#define op_  o_.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
             *r_;} binary_;
  } c_;
#define unary_ c_.unary_
#define binary_ c_.binary_
} Tree_Node;
```

Lack of extensibility was a major limitation with algorithmic decomposition.

Problem: Non-Extensible & Error-Prone Designs

- Tightly coupling expression tree data structures & functionality impedes extensibility.
  - e.g., adding new types of nodes or new node operations affects many parts of the program.

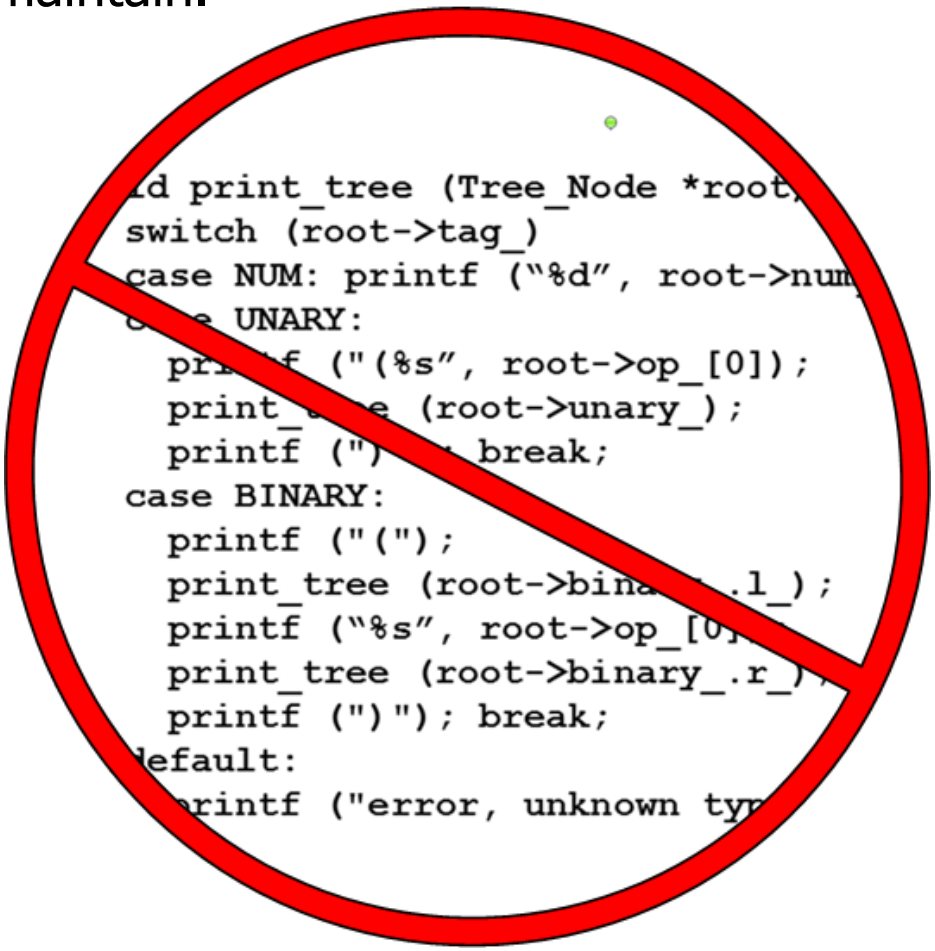


```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY,
         TERNARY } tag_;
  union {
    char op_[4];
    int num_;
  } o_;
  ...
  union {
  ...
    struct {
      Tree_Node *l_,
                *m_,
                *r_;
    } ternary_;
  } c_;
#define ternary_ c_.ternary_
} Tree_Node;
```

See lesson on "*Evaluating the Algorithmic Decomposition of the Expression Tree Processing App*"

Problem: Non-Extensible & Error-Prone Designs

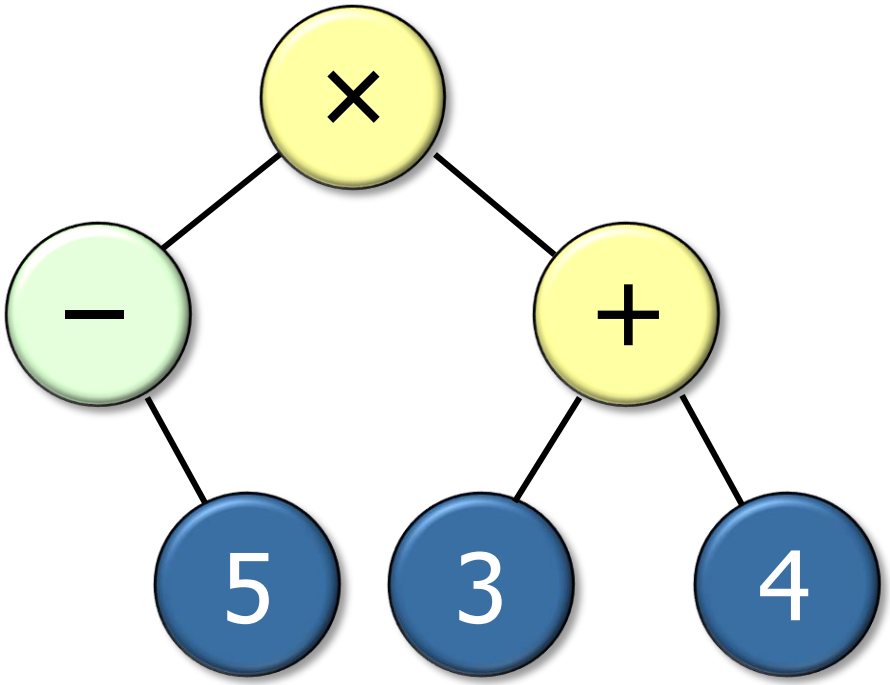
- Differentiating operators & operands via type tags & switch statements is tedious & error-prone to program & maintain.



See wiki.c2.com/?SwitchStatementsSmell

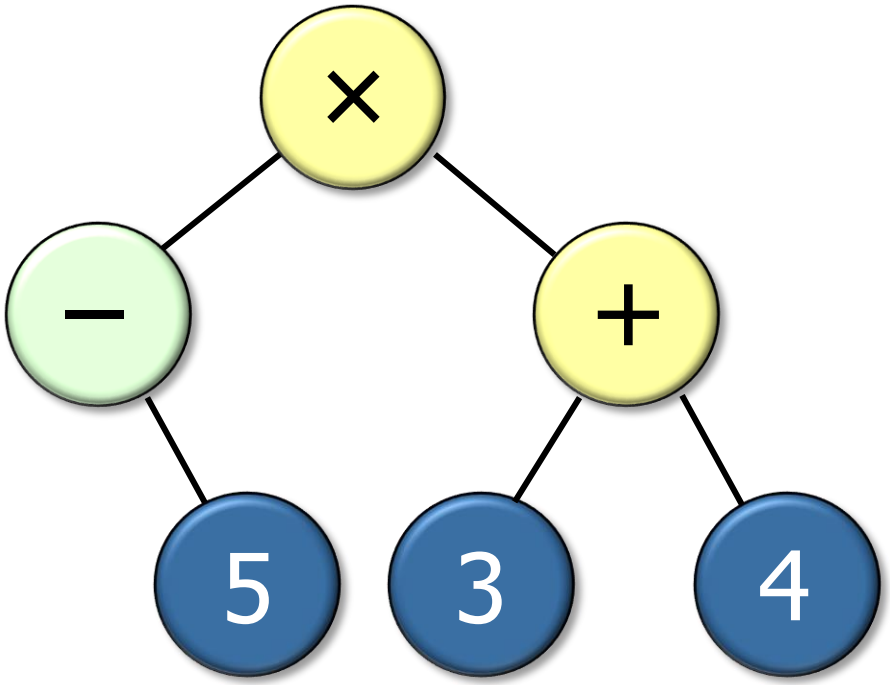
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes



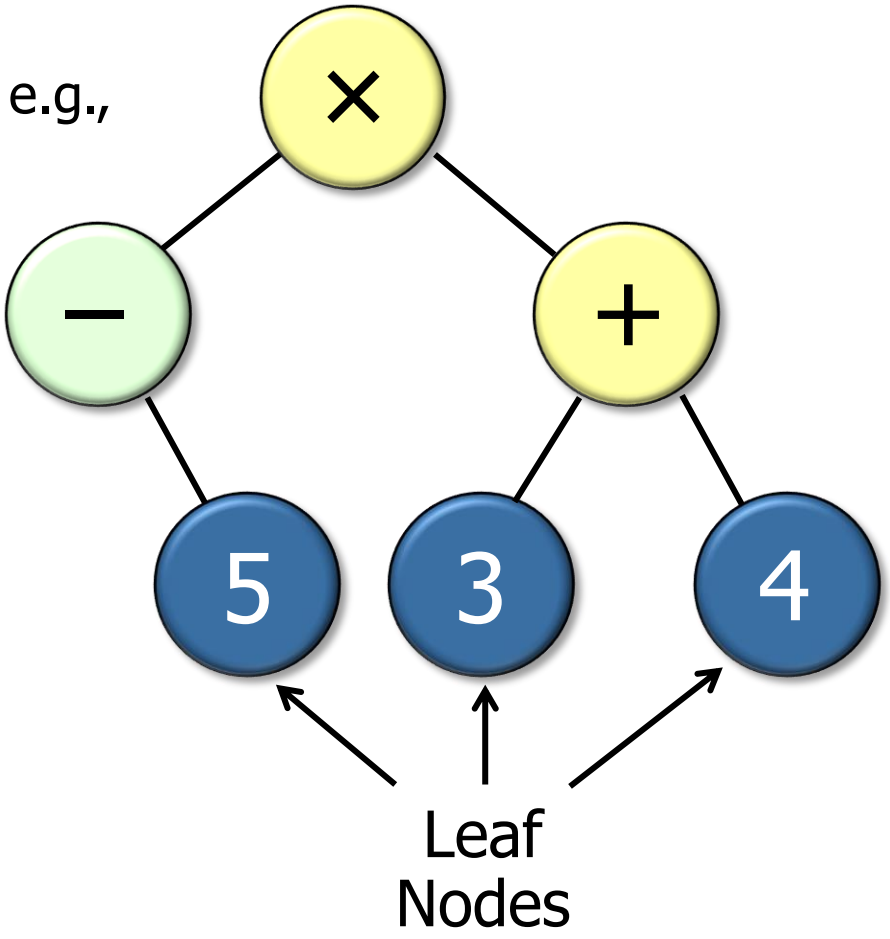
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.,
  - Structure nodes into a hierarchy that captures the properties of each node



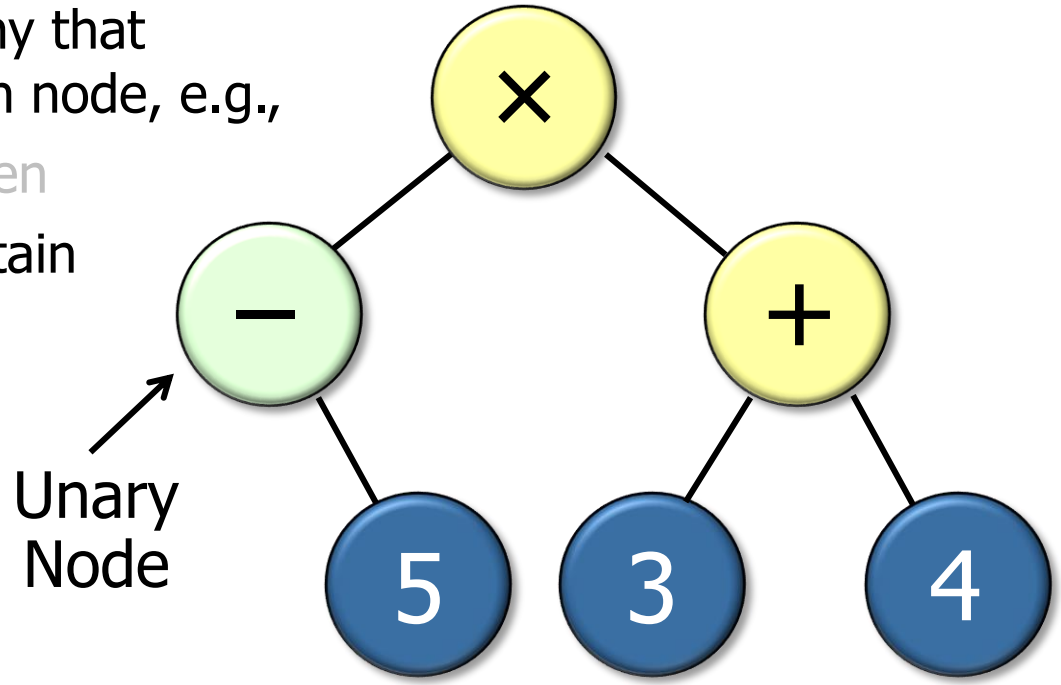
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.,
  - Structure nodes into a hierarchy that captures the properties of each node, e.g.,
    - Leaf nodes contain no children



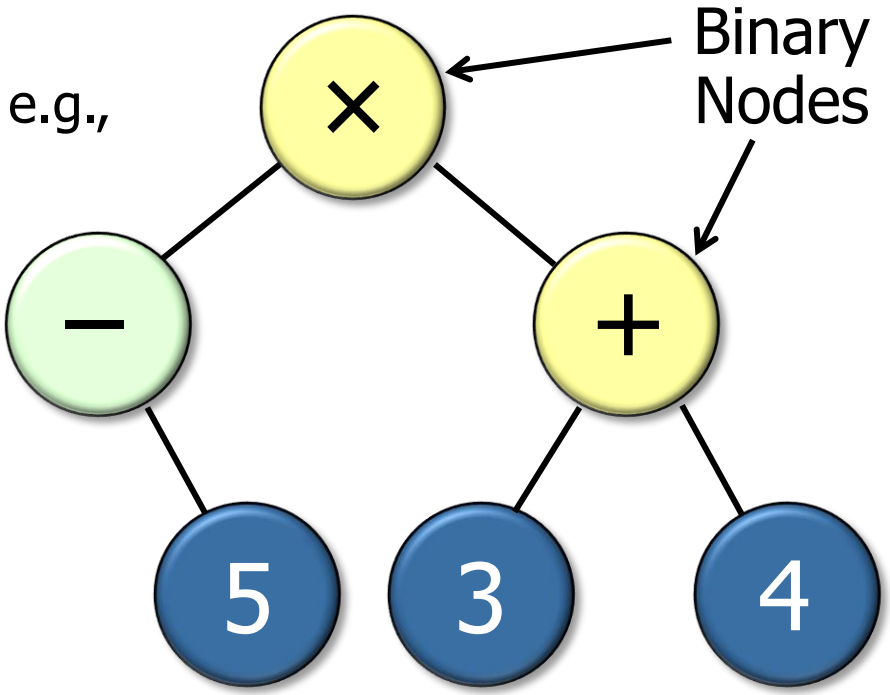
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.,
  - Structure nodes into a hierarchy that captures the properties of each node, e.g.,
    - Leaf nodes contain no children
    - Unary nodes recursively contain one child node



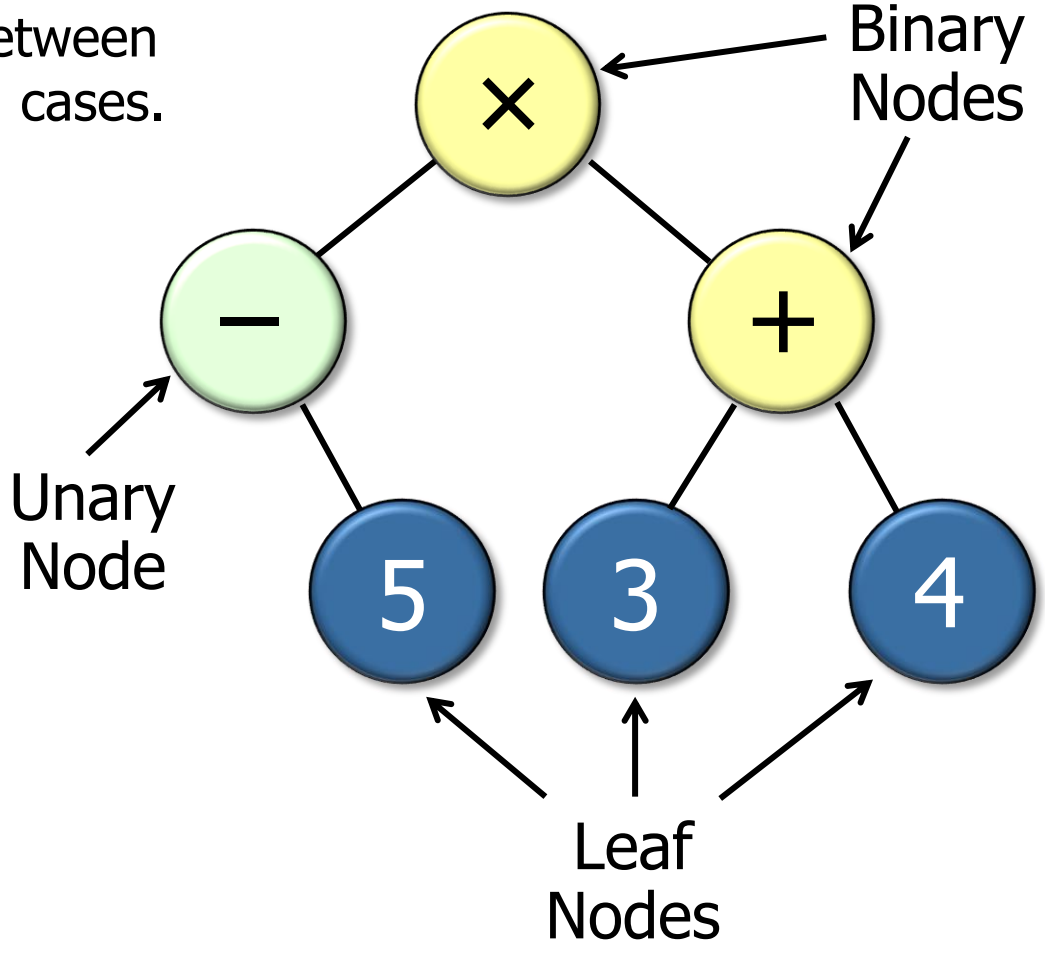
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.,
  - Structure nodes into a hierarchy that captures the properties of each node, e.g.,
    - Leaf nodes contain no children
    - Unary nodes recursively contain one child node
    - Binary nodes recursively contain two child nodes



Solution: Recursive Object Structure

- Treat operators & operands uniformly
  - e.g., minimize the distinction between "one vs. many" to avoid special cases.



Component_Node Class Overview

- Abstract base class for composable expression tree node objects

Class methods

```
                int  item()
Component_Node  *left()
Component_Node  *right()
               void  accept(Visitor &visitor)
```

Component_Node Class Overview

- Abstract base class for composable expression tree node objects

Class methods

These methods access relevant fields (may be no-ops for some implementations).

```
                int  item()
Component_Node  *left()
Component_Node  *right()
               void  accept(Visitor &visitor)
```

Component_Node Class Overview

- Abstract base class for composable expression tree node objects

Class methods

```
              int  item()
Component_Node *left()
Component_Node *right()
             void  accept(Visitor &visitor)
```

This hook method plays an essential role in *Iterator* & *Visitor* patterns.

See upcoming lessons on "The *Iterator* Pattern" & "The *Visitor* Pattern."

Component_Node Class Overview

- Abstract base class for composable expression tree node objects

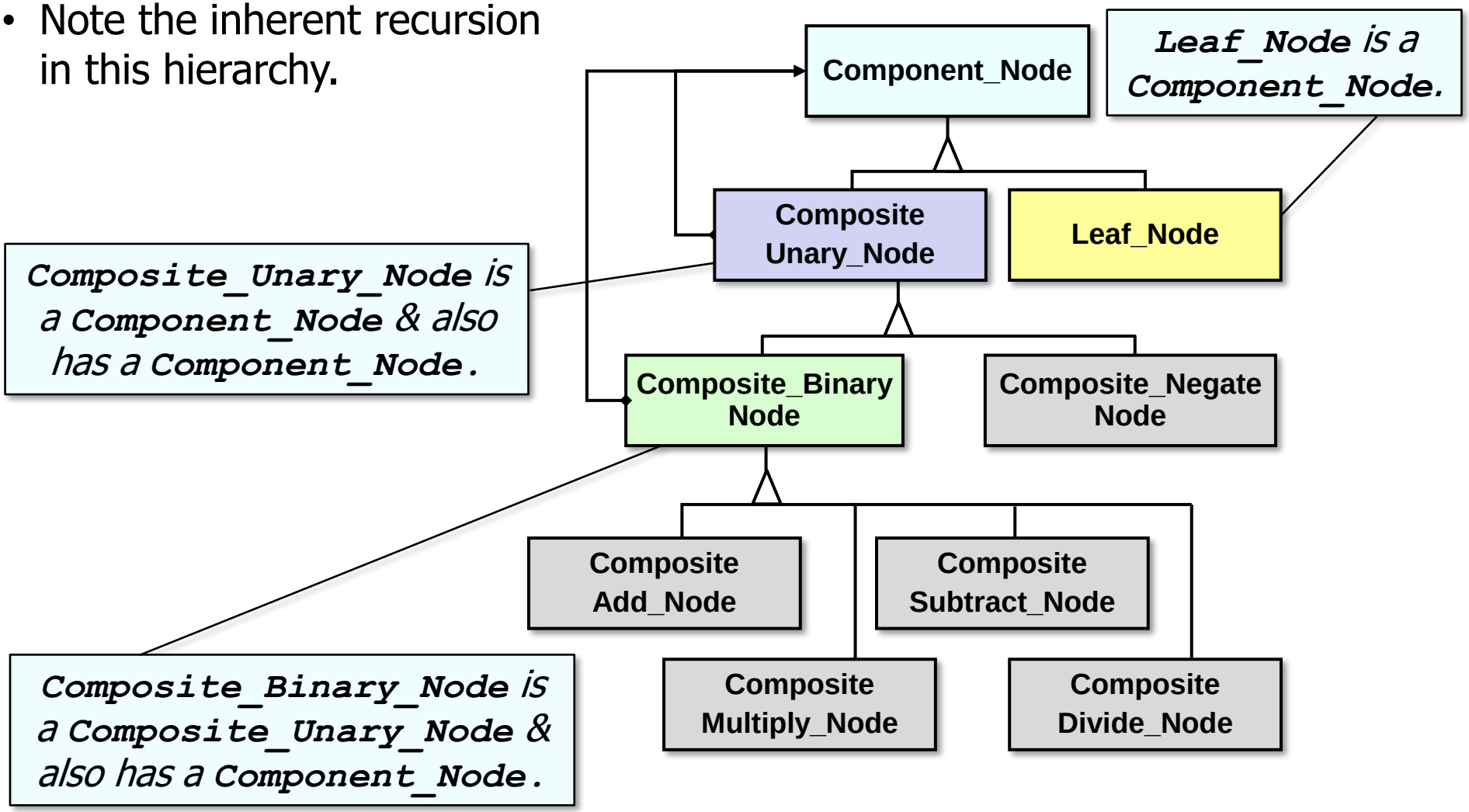
Class methods

```
                int  item()
Component_Node  *left()
Component_Node  *right()
               void  accept(Visitor &visitor)
```

- **Commonality**: the abstract base class used by all nodes in an expression tree
- **Variability**: each subclass defines the state & methods that can be customized for specific types of expression tree nodes

Component_Node Class Hierarchy Overview

- Note the inherent recursion in this hierarchy.



Component_Node Class Hierarchy Overview

- Note the inherent recursion in this hierarchy.