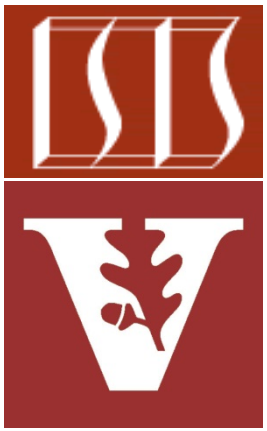
A Case Study of "Gang of Four" (GoF) Patterns: Part 3

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



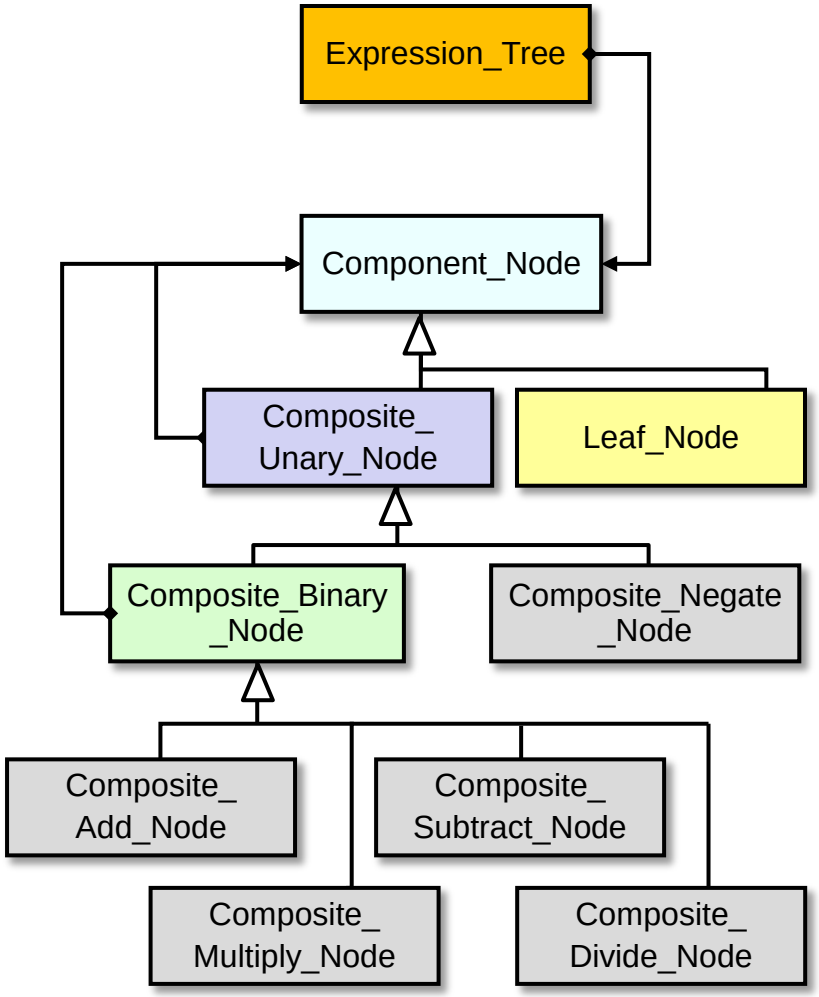
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

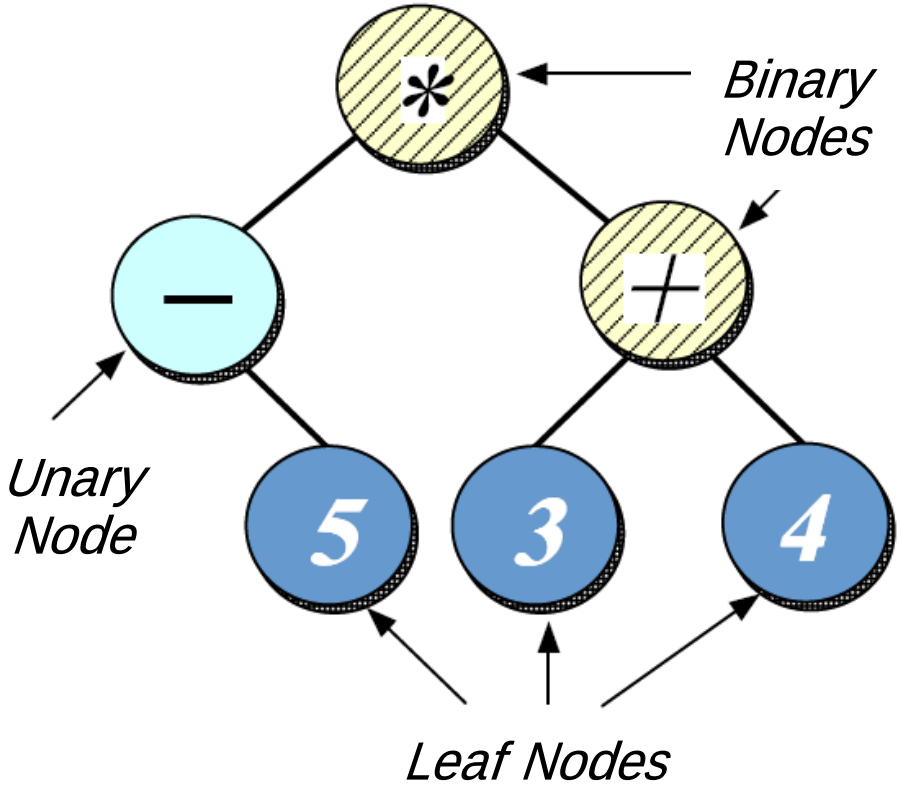
Topics Covered in this Part of the Module

- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques
- Present an OO design for the expression tree processing app



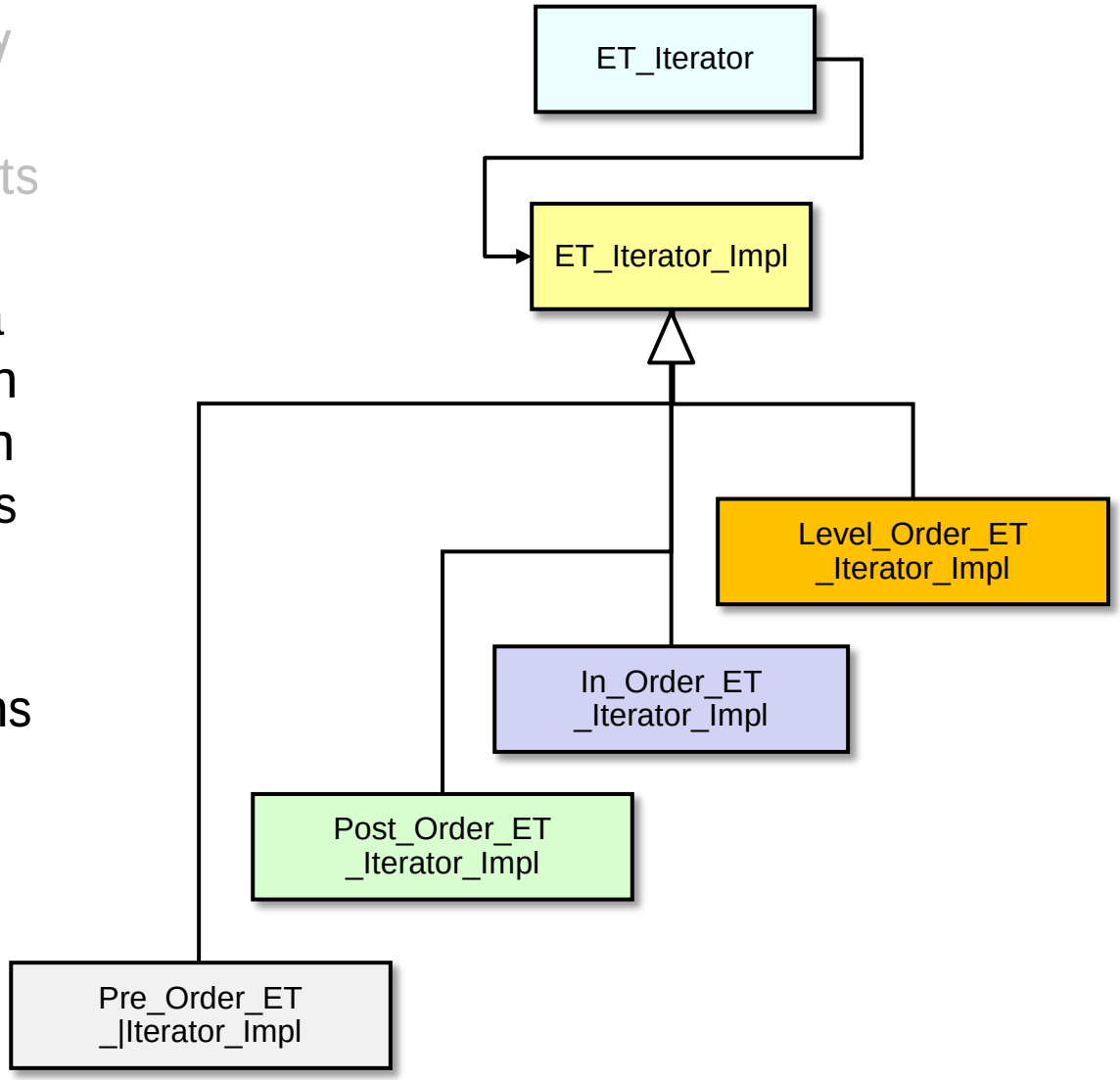
How to Design an Expression Tree Processing App

- Apply an *Object-Oriented* (OO) design based on modeling classes & objects in the application domain



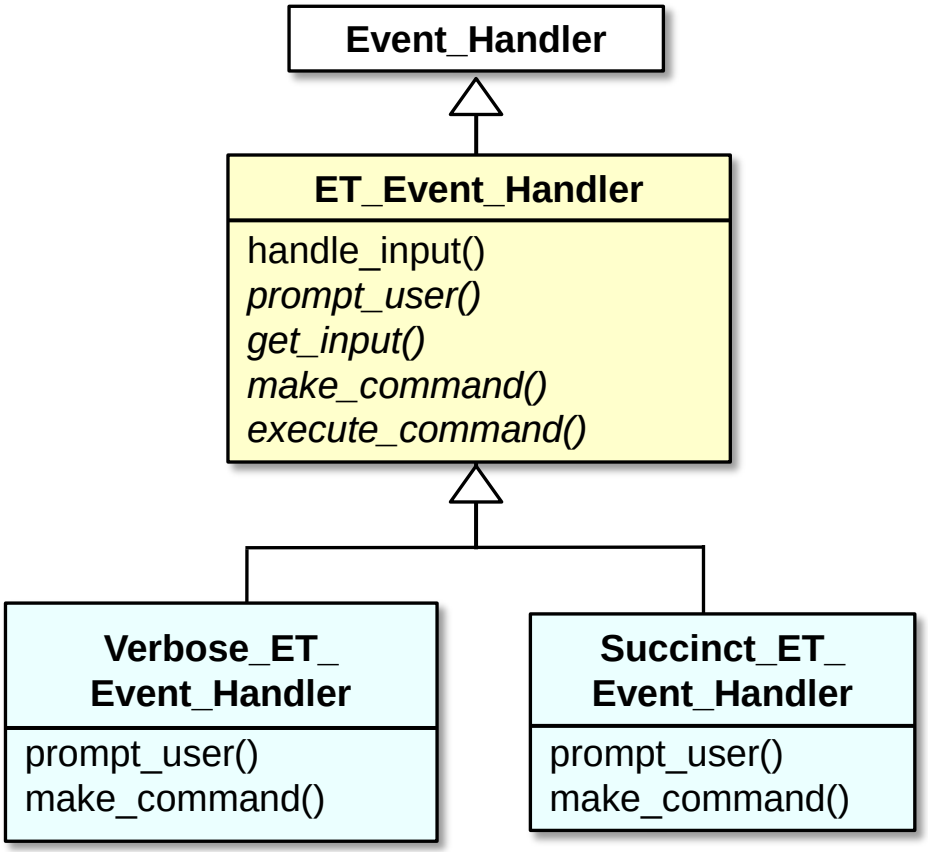
How to Design an Expression Tree Processing App

- Apply an *Object-Oriented* (OO) design based on modeling classes & objects in the application domain
- Employ "hierarchical data abstraction" where design components are based on stable *class* & *object* roles & relationships
  - Rather than functions corresponding to actions



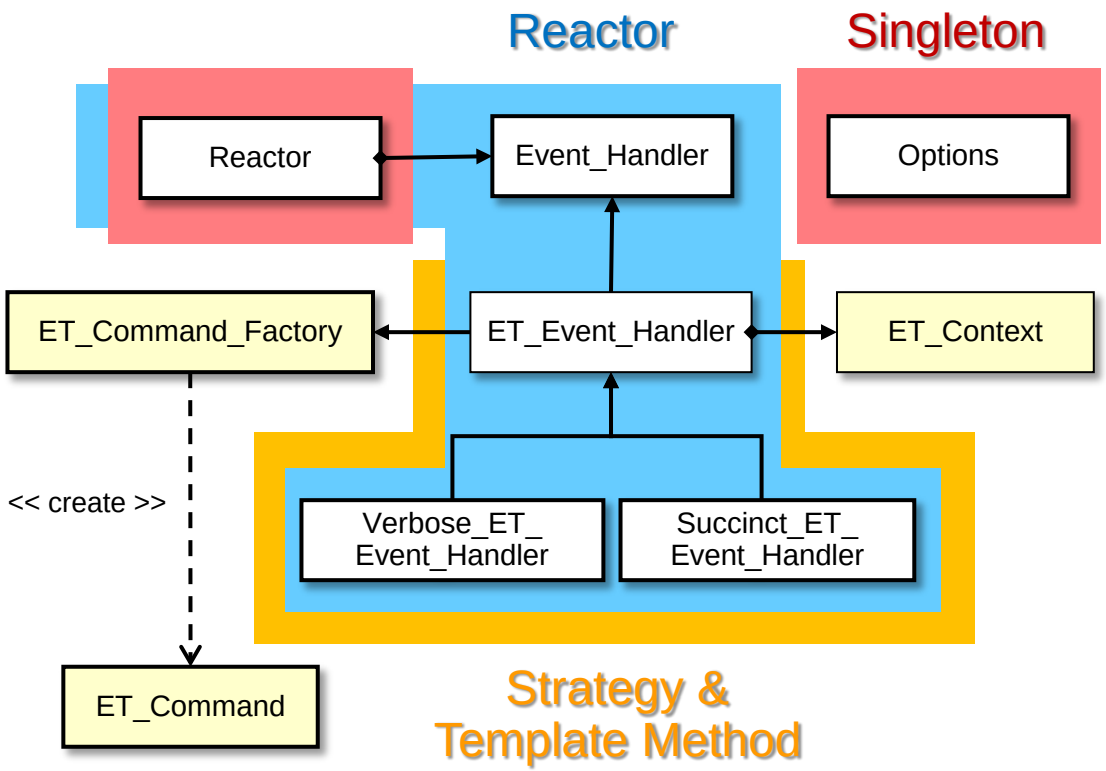
How to Design an Expression Tree Processing App

- Apply an *Object-Oriented* (OO) design based on modeling classes & objects in the application domain
- Employ "hierarchical data abstraction" where design components are based on stable *class* & *object* roles & relationships
- Associate actions with specific objects and/or classes of objects
  - Emphasize *high cohesion* & *low coupling*



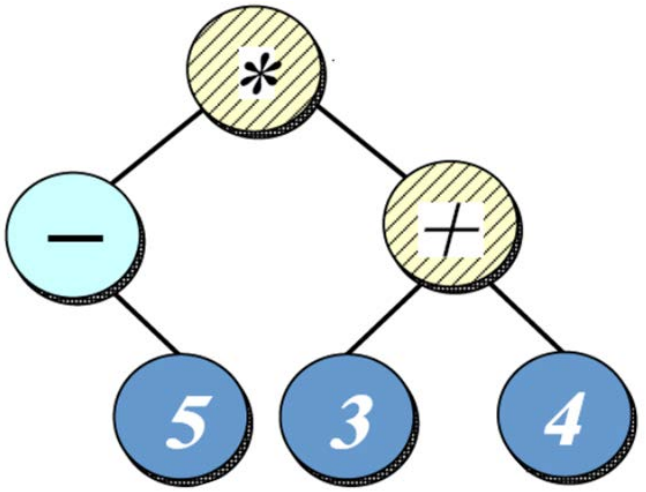
How to Design an Expression Tree Processing App

- Apply an *Object-Oriented* (OO) design based on modeling classes & objects in the application domain
- Employ "hierarchical data abstraction" where design components are based on stable *class* & *object* roles & relationships
- Associate actions with specific objects and/or classes of objects
- Group classes & objects in accordance to *patterns* & combine them to form *frameworks*



An OO Expression Tree Design Method

- Start with object-oriented (OO) modeling of the “expression tree” application domain

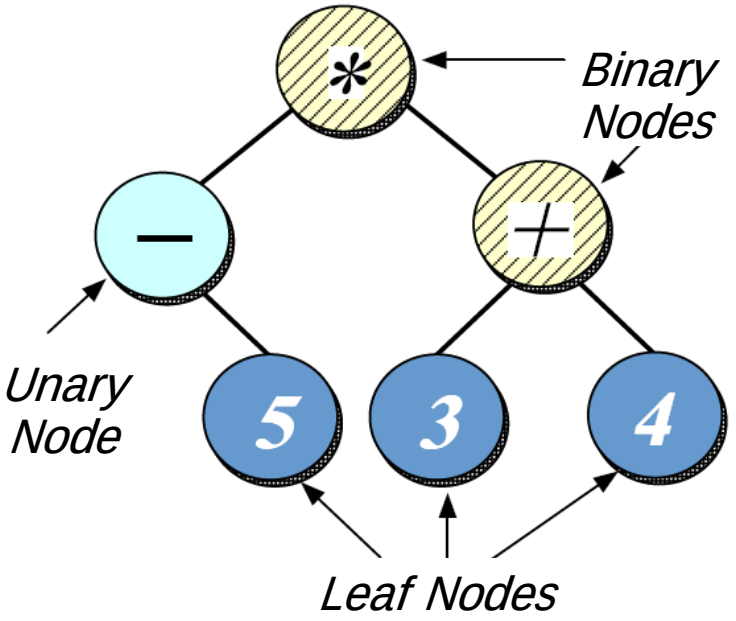


An OO Expression Tree Design Method

- Start with object-oriented (OO) modeling of the "expression tree" application domain

Application-dependent steps

- Model a *tree* as a collection of *nodes*

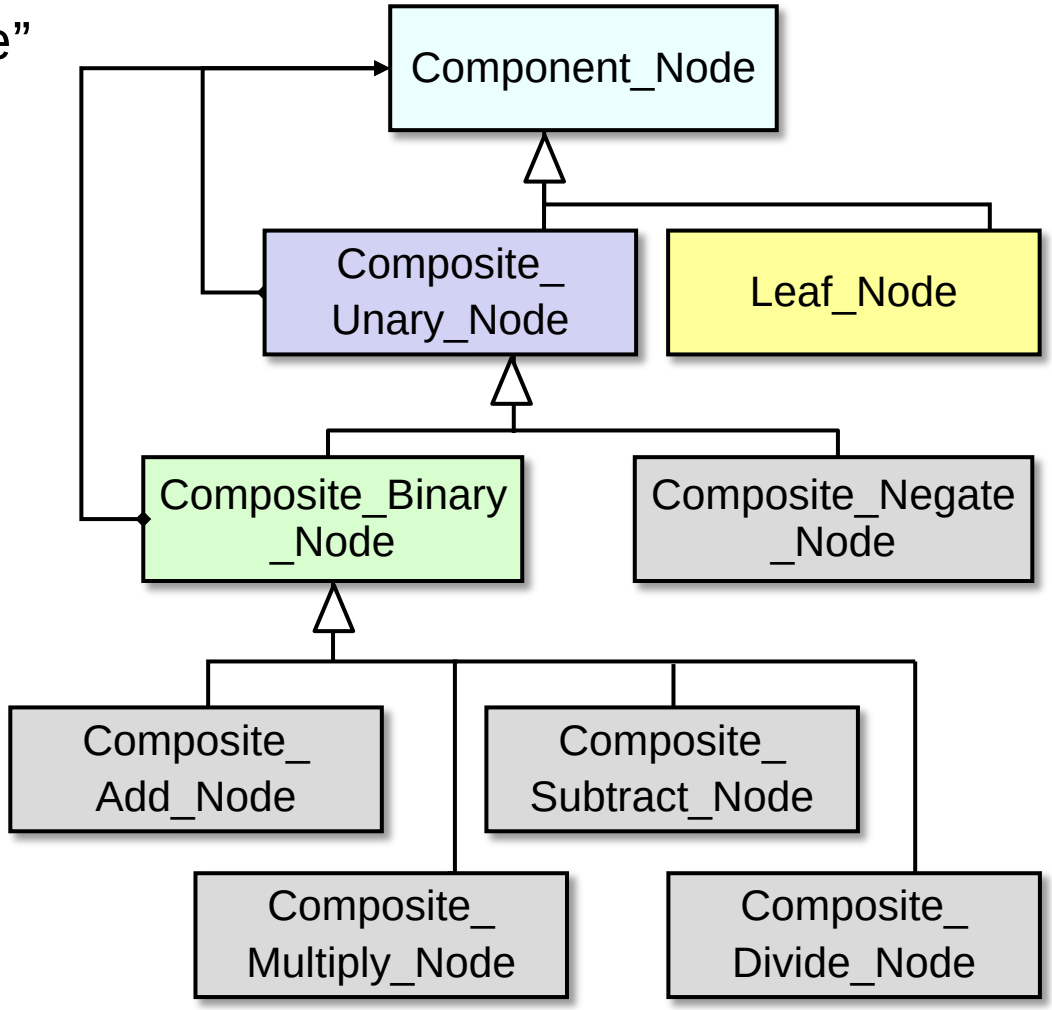


An OO Expression Tree Design Method

- Start with object-oriented (OO) modeling of the "expression tree" application domain

Application-dependent steps

- Model a *tree* as a collection of *nodes*
- Represent *nodes* as a hierarchy, capturing properties of each node
  - e.g., arities



An OO Expression Tree Design Method

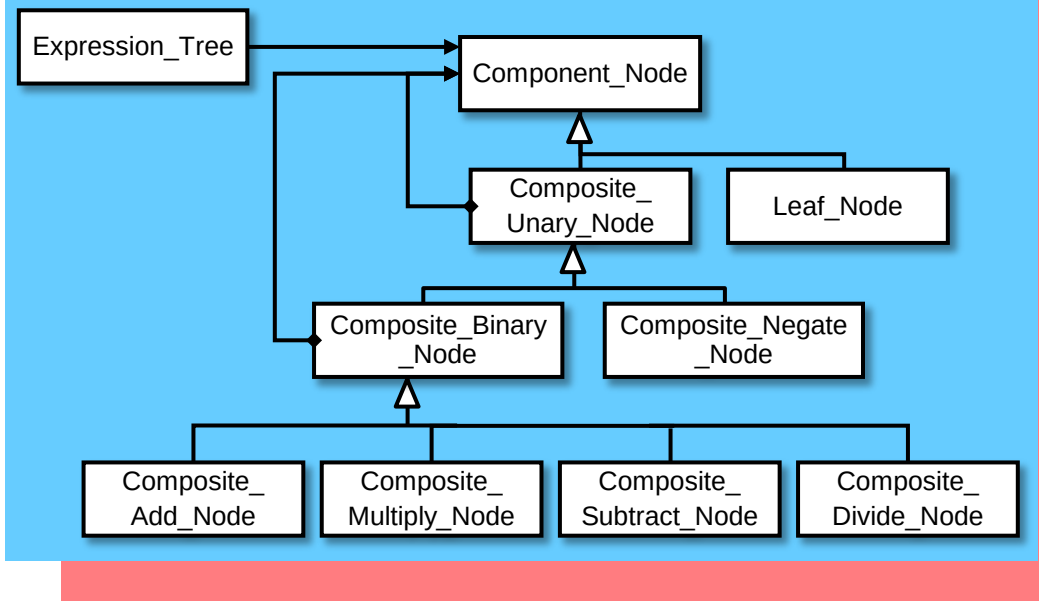
- Start with object-oriented (OO) modeling of the "expression tree" application domain

Application-dependent steps

- Model a *tree* as a collection of *nodes*
- Represent *nodes* as a hierarchy, capturing properties of each node
  - e.g., arities

Composite

Bridge



Application-independent steps

- Conduct *Scope, Commonality, & Variability* analysis to determine stable interfaces & extension points
- Apply "Gang of Four" (GoF) patterns to guide efficient & extensible development of framework components
- Integrate pattern-oriented language/library features w/frameworks

en.wikipedia.org/wiki/Design_Patterns has info on "Gang of Four" (GoF) book

C++ Pattern-Oriented Language/Library Features

- Over time, common patterns become institutionalized as programming language features

```
Expression_Tree expr_tree = …;
Print_Visitor print_visitor;
```

Visitor object (based on *Visitor* pattern)

C++ Pattern-Oriented Language/Library Features

- Over time, common patterns become institutionalized as programming language features

Traditional STL iterator loop

```
Expression_Tree expr_tree = …;
Print_Visitor print_visitor;

for (Expression_Tree::iterator iter =
           expr_tree.begin();
        iter != expr_tree.end();
        ++iter)
     (*iter).accept(print_visitor);
```

C++ Pattern-Oriented Language/Library Features

- Over time, common patterns become institutionalized as programming language features

```
Expression_Tree expr_tree = …;
Print_Visitor print_visitor;

for (Expression_Tree::iterator iter =
       expr_tree.begin();
     iter != expr_tree.end();
     ++iter)
  (*iter).accept(print_visitor);

std::for_each
  (expr_tree.begin(), expr_tree.end(),
   [&print_visitor]
   (const Expression_Tree &t)
   { t.accept(print_visitor);});
```

C++11 lambda expression

C++ Pattern-Oriented Language/Library Features

- Over time, common patterns become institutionalized as programming language features

```
Expression_Tree expr_tree = …;
Print_Visitor print_visitor;

for (Expression_Tree::iterator iter =
          expr_tree.begin();
       iter != expr_tree.end();
       ++iter)
   (*iter).accept(print_visitor);

std::for_each
  (expr_tree.begin(), expr_tree.end(),
   [&print_visitor]
   (const Expression_Tree &t)
   { t.accept(print_visitor);});
```

C++11 range-based for loop

```
for (auto &iter : expr_tree)
   iter.accept(print_visitor);
```

See en.wikipedia.org/wiki/C++11 for info on C++11

Java Pattern-Oriented Language/Library Features

- Over time, common patterns become institutionalized as programming language features

```
ExpressionTree exprTree = …;
ETVisitor printVisitor =
                    new PrintVisitor();

for (ComponentNode node : exprTree)
     node.accept(printVisitor);
```

Java for-each loop (assumes tree implements Iterable)

Java Pattern-Oriented Language/Library Features

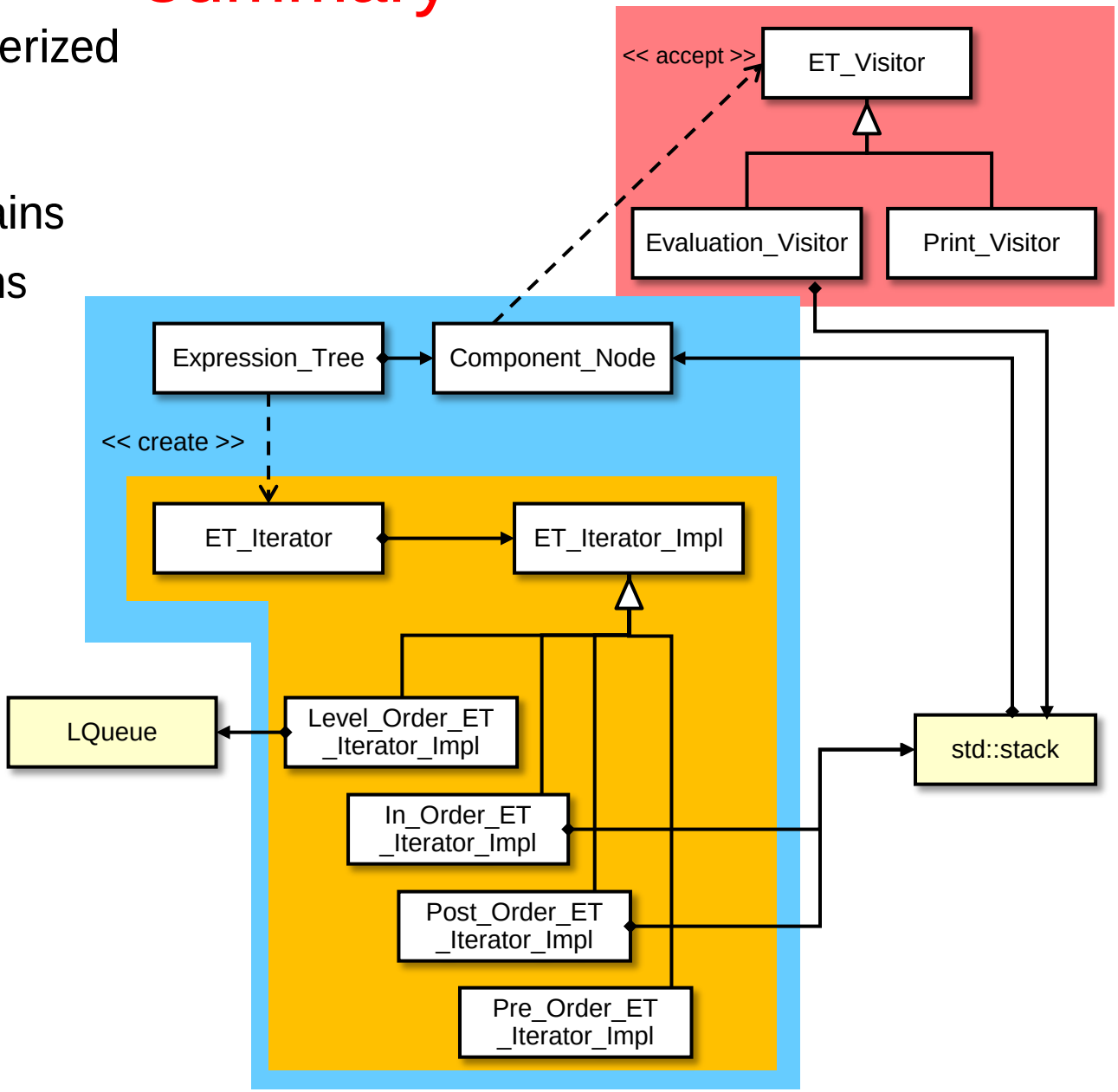
- Over time, common patterns become institutionalized as programming language features

```
ExpressionTree exprTree = …;
ETVisitor printVisitor =
                  new PrintVisitor();

for (ComponentNode node : exprTree)
    node.accept(printVisitor);

for (Iterator<ExpressionTree> iter =
       exprTree.iterator();
     iter.hasNext();
     )
   iter.next().accept
          (printVisitor);
```

Java iterator style

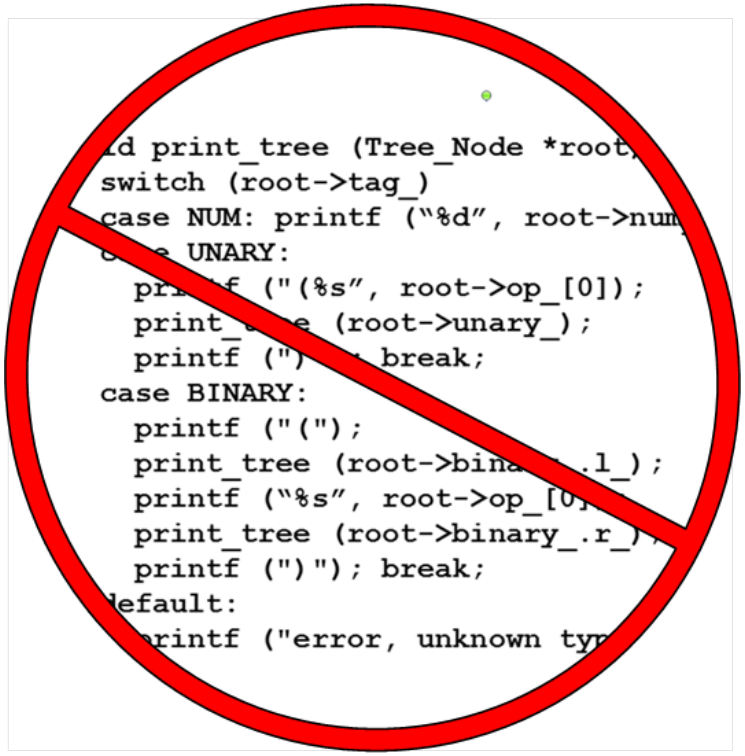
Summary

- OO designs are characterized by structuring software architectures around objects/classes in domains
  - Rather than on actions performed by the software



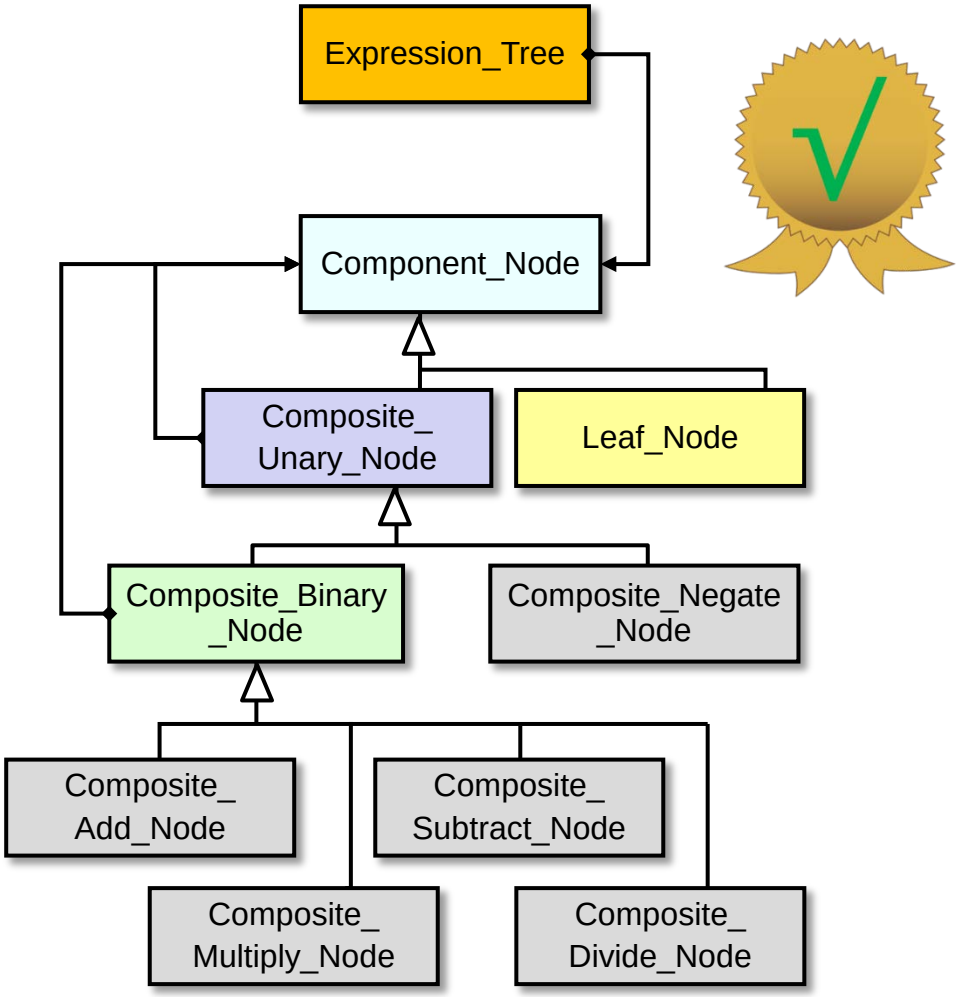
Summary

- OO designs are characterized by structuring software architectures around objects/classes in domains
  - Rather than on actions performed by the software
- Systems evolve & functionality changes, but well-defined objects & class roles & relationships are often relatively stable over time



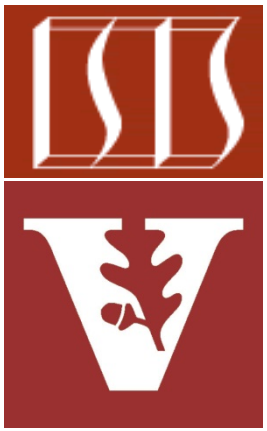
Summary

- OO designs are characterized by structuring software architectures around objects/classes in domains
  - Rather than on actions performed by the software
- Systems evolve & functionality changes, but well-defined objects & class roles & relationships are often relatively stable over time
- To obtain flexible & reusable software, therefore, it's better to base the structure on objects/classes rather than on actions



A Case Study of "Gang of Four" (GoF) Patterns: Part 4

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



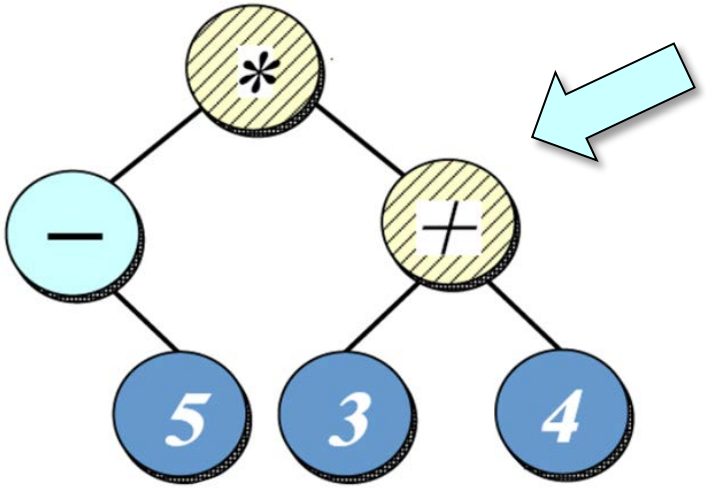
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques
- Present an OO design for the expression tree processing app
- Summarize the patterns in the expression tree design



Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible expression tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method
Ensuring correct protocol for commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL **std::for_each()** algorithm	Adapter

Outline of the Design Space for GoF Patterns

Abstract the process of instantiating objects

Describe how classes & objects can be combined to form larger structures

Concerned with communication between objects

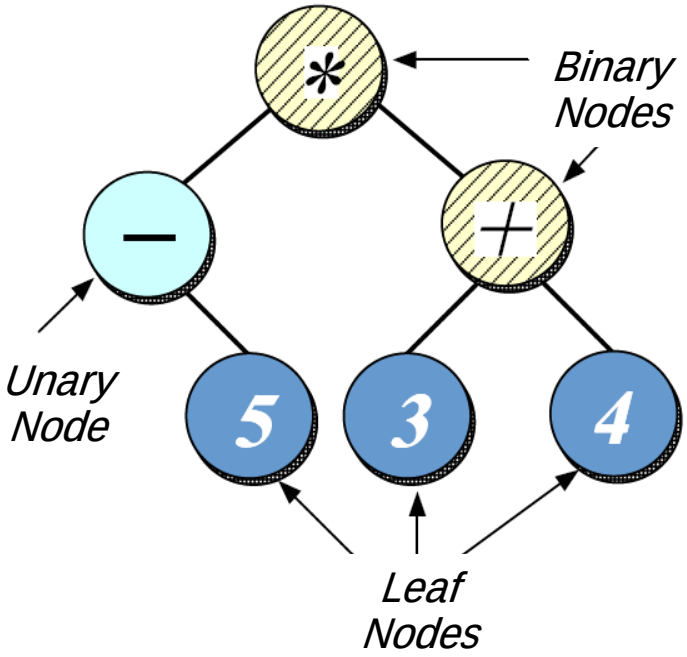
Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method √	Adapter √ (class)	Interpreter √ Template Method √
Object	Abstract Factory √ Builder √ Prototype √ Singleton √	Adapter (object) Bridge √ Composite √ Decorator Flyweight Façade Proxy	Chain of Responsibility Command √ Iterator √ Mediator Memento Observer State √ Strategy √ Visitor √

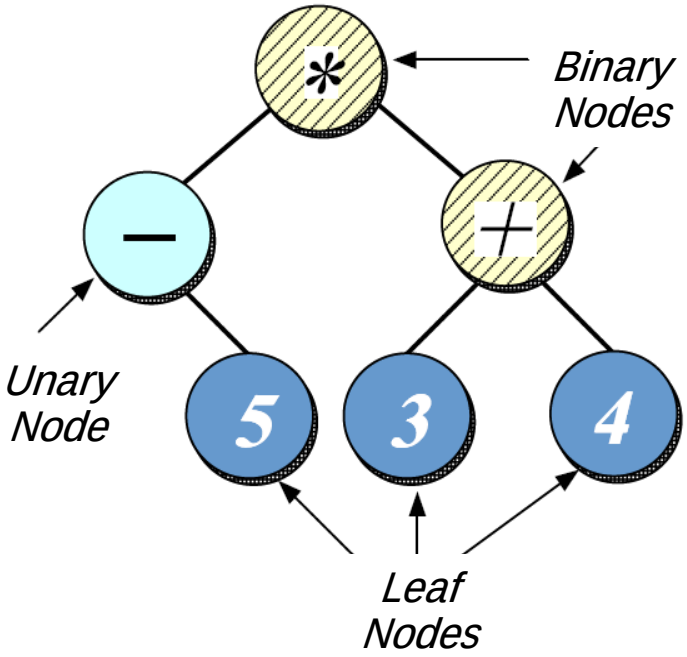
en.wikipedia.org/wiki/Design_Patterns has info on “Gang of Four” (GoF) book

Design Problems & Pattern-Oriented Solutions



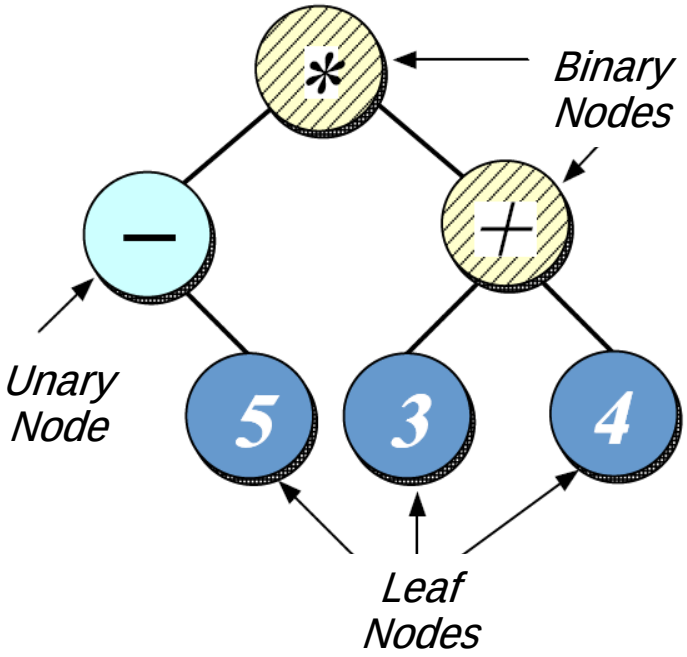
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



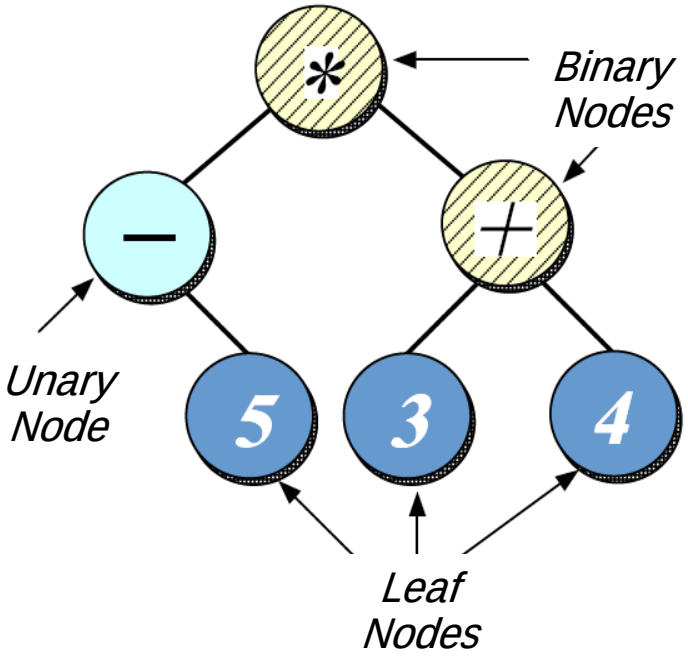
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



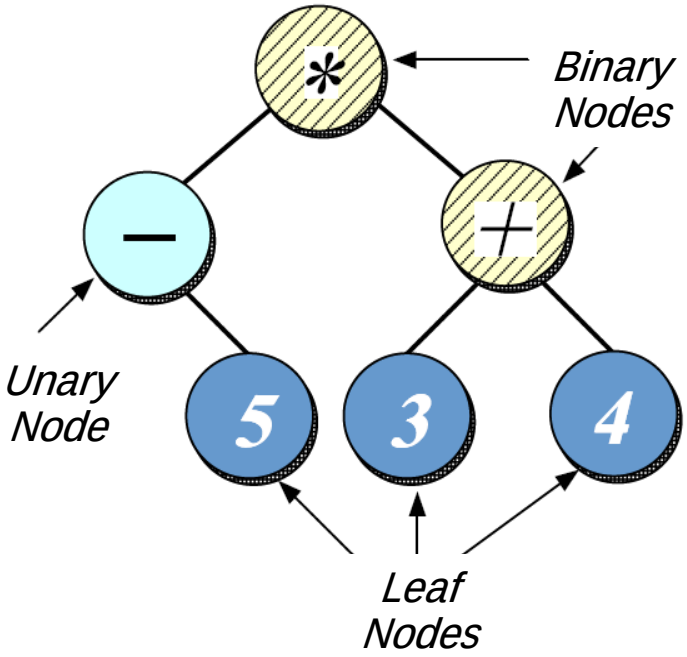
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



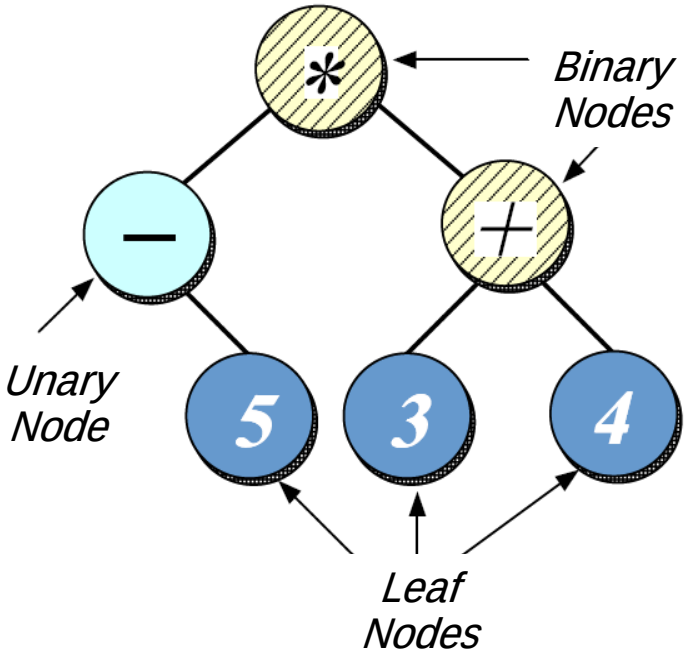
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



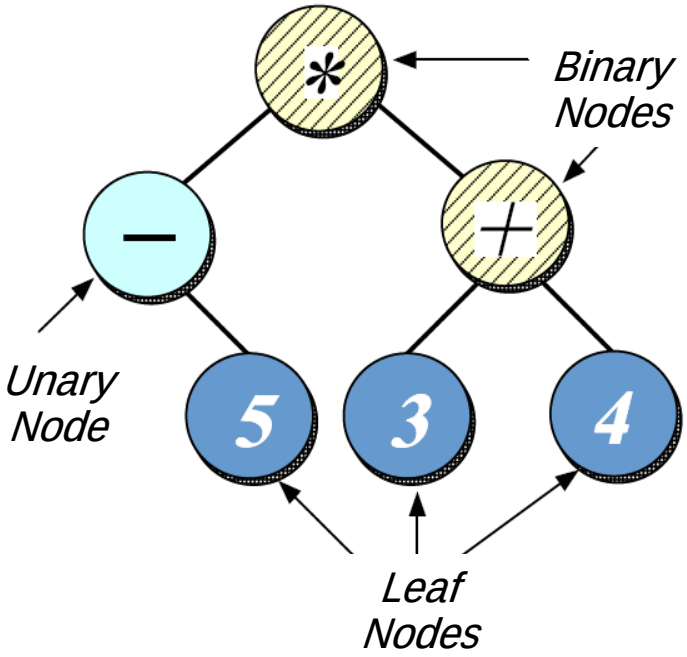
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



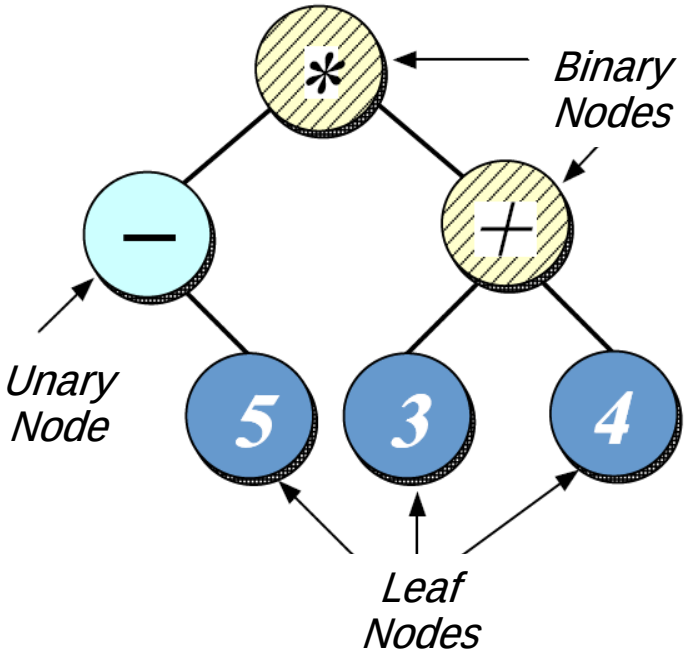
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

Design Problems & Pattern-Oriented Solutions



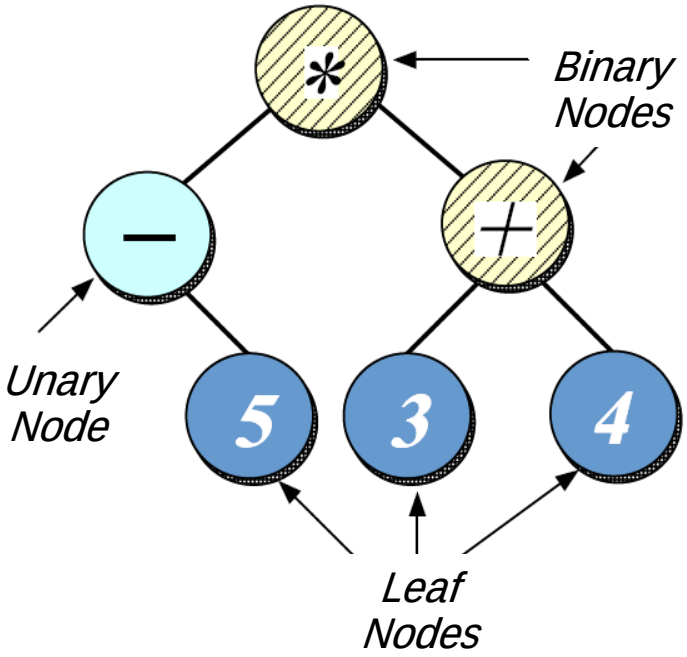
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method

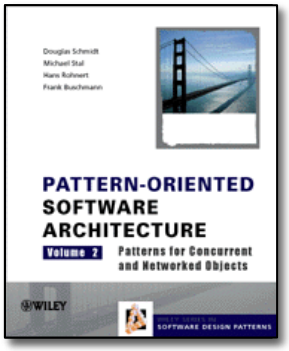
Design Problems & Pattern-Oriented Solutions



Design Problem	Pattern(s)
Ensuring correct protocol for processing commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL `std::for_each()` algorithm	Adapter

Design Problems & Pattern-Oriented Solutions

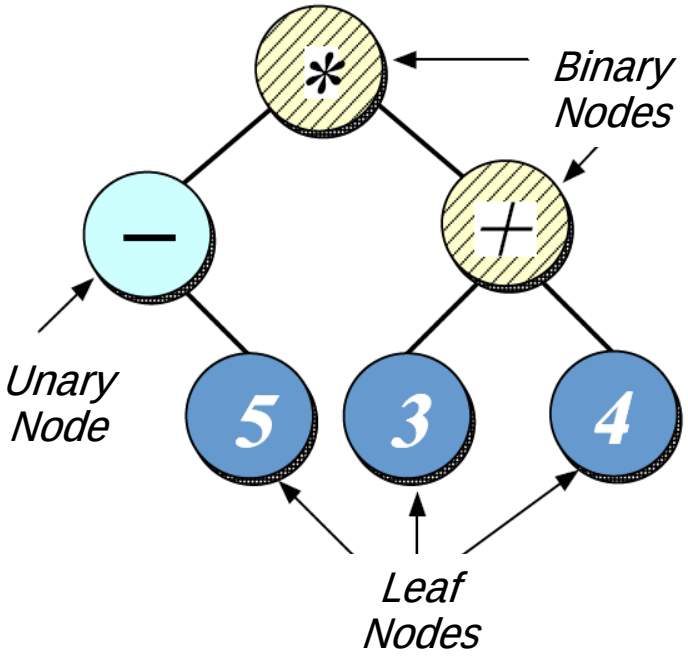




Design Problem	Pattern(s)
Ensuring correct protocol for processing commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL `std::for_each()` algorithm	Adapter

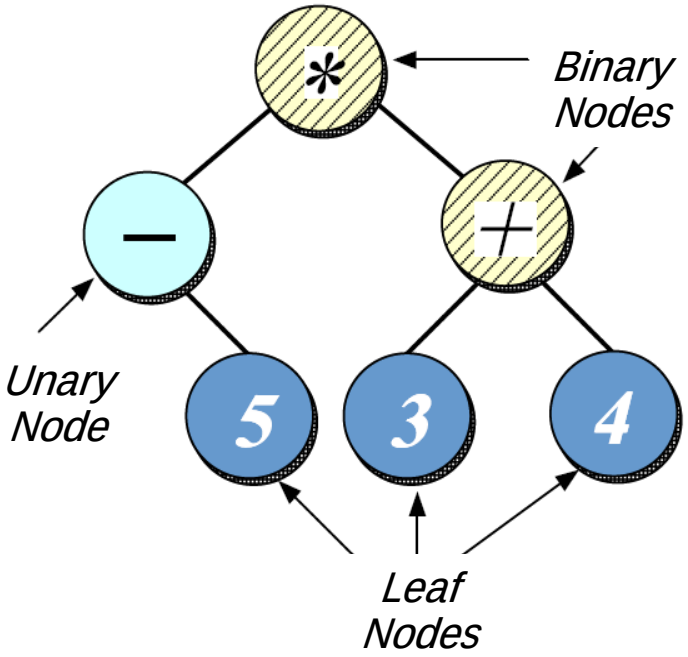
See www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf for the *Reactor* pattern

Design Problems & Pattern-Oriented Solutions



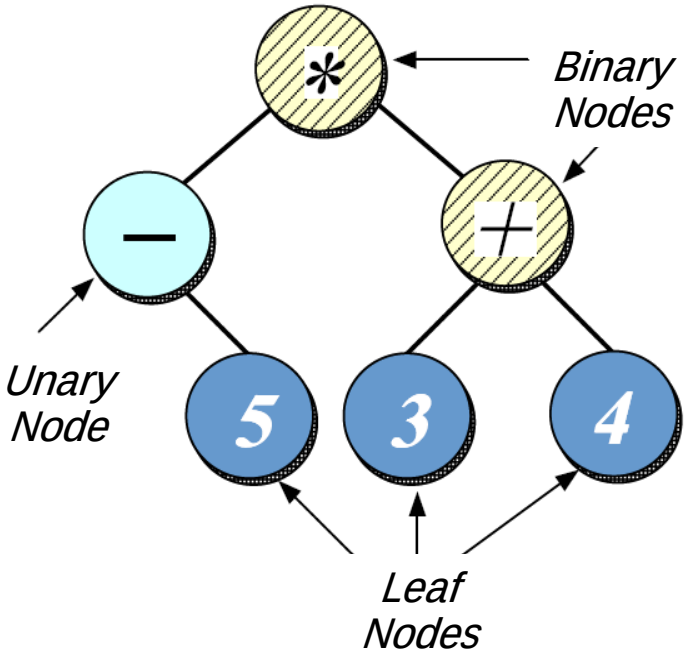
Design Problem	Pattern(s)
Ensuring correct protocol for processing commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL `std::for_each()` algorithm	Adapter

Design Problems & Pattern-Oriented Solutions



Design Problem	Pattern(s)
Ensuring correct protocol for processing commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL `std::for_each()` algorithm	Adapter

Design Problems & Pattern-Oriented Solutions



Design Problem	Pattern(s)
Ensuring correct protocol for processing commands	State
Structuring the application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL `std::for_each()` algorithm	Adapter

Naturally, these patterns apply to more than expression tree processing apps!

Summary

- GoF patterns provide elements of reusable object-oriented software that address limitations with algorithmic decomposition

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method √	Adapter (class) √	Interpreter √ Template Method √
Object	Abstract Factory √ Builder √ Prototype √ Singleton √	Adapter (object) √ Bridge √ Composite √ Decorator Flyweight Façade Proxy	Chain of Responsibility Command √ Iterator √ Mediator Memento Observer State √ Strategy √ Visitor √