

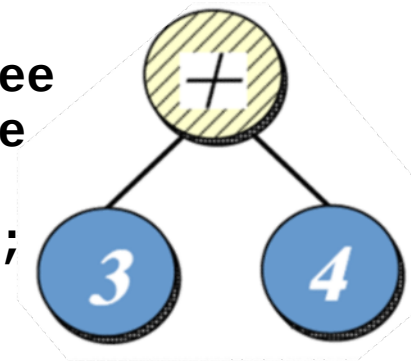
Problem: Encapsulating Variability & Simplifying Memory Management

Goals

- Help encapsulate sources of variability in expression tree construction & use

```
Expression_Tree expr_tree  
  (new Composite_Add_Node  
   (new Leaf_Node (3),  
    new Leaf_Node (4)));
```

Hide complex recursive internal structure behind a simple interface

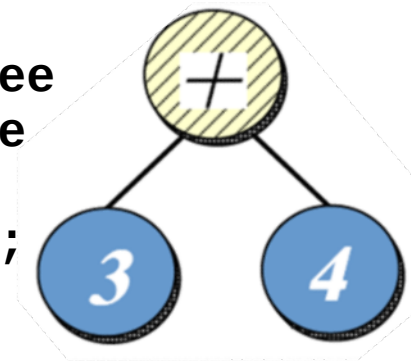


Problem: Encapsulating Variability & Simplifying Memory Management

Goals

- Help encapsulate sources of variability in expression tree construction & use

```
Expression_Tree expr_tree  
    (new Composite_Add_Node  
      (new Leaf_Node (3),  
        new Leaf_Node (4)));
```



```
Print_Visitor print_visitor;
```

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

Iterate through all elements in
expression tree without concern
for how tree is structured or what
traversal has been designated

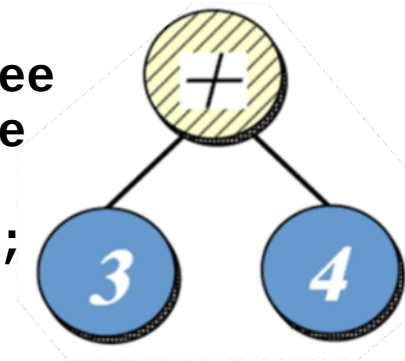


Problem: Encapsulating Variability & Simplifying Memory Management

Goals

- Help encapsulate sources of variability in expression tree construction & use

```
Expression_Tree expr_tree
    (new Composite_Add_Node
     (new Leaf_Node (3),
      new Leaf_Node (4)));
```



```
Print_Visitor print_visitor;
```

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

Note how C++11 auto keyword
can deduce type of Expression
_Tree iterator



Problem: Encapsulating Variability & Simplifying Memory Management

Goals

- Help encapsulate sources of variability in expression tree construction & use
- Simplify memory management
 - e.g., minimize use of “naked pointers” in C++ app code to avoid memory leaks stemming from exceptions

This is just asking for trouble in C++! 

```
Component_Node *l1 =  
    new Leaf_Node(5);  
Component_Node *l2 =  
    new Leaf_Node(3);  
Component_Node *l3 =  
    new Leaf_Node(4);  
Component_Node *u1 =  
    new Composite_Negate_Node(l1);  
Component_Node *b1 =  
    new Composite_Add_Node(l2, l3);  
Component_Node *b2 =  
    new Composite_Multiply_Node(u1,  
        b1);  
...  
delete b2;
```

Problem: Encapsulating Variability & Simplifying Memory Management

Goals

- Help encapsulate sources of variability in expression tree construction & use
- Simplify memory management
 - e.g., minimize use of “naked pointers” in C++ app code to avoid memory leaks stemming from exceptions

Constraints/forces

- Account for the fact that STL algorithms & iterators have “value semantics”

```
class Pre_Order_ET_Iterator_Impl
    : public ET_Iterator_Impl {
public:
    Pre_Order_ET_Iterator_Impl
        (const Expression_Tree &root) {
        stack_.push(root);
    }
```

Need to optimize this operation to avoid overhead of “deep copies”



```
// ...
private:
    std::stack<Expression_Tree> stack_;
};
```



Solution: Decouple Interface & Implementation(s)

- Create an interface class (**Expression_Tree**) used by clients

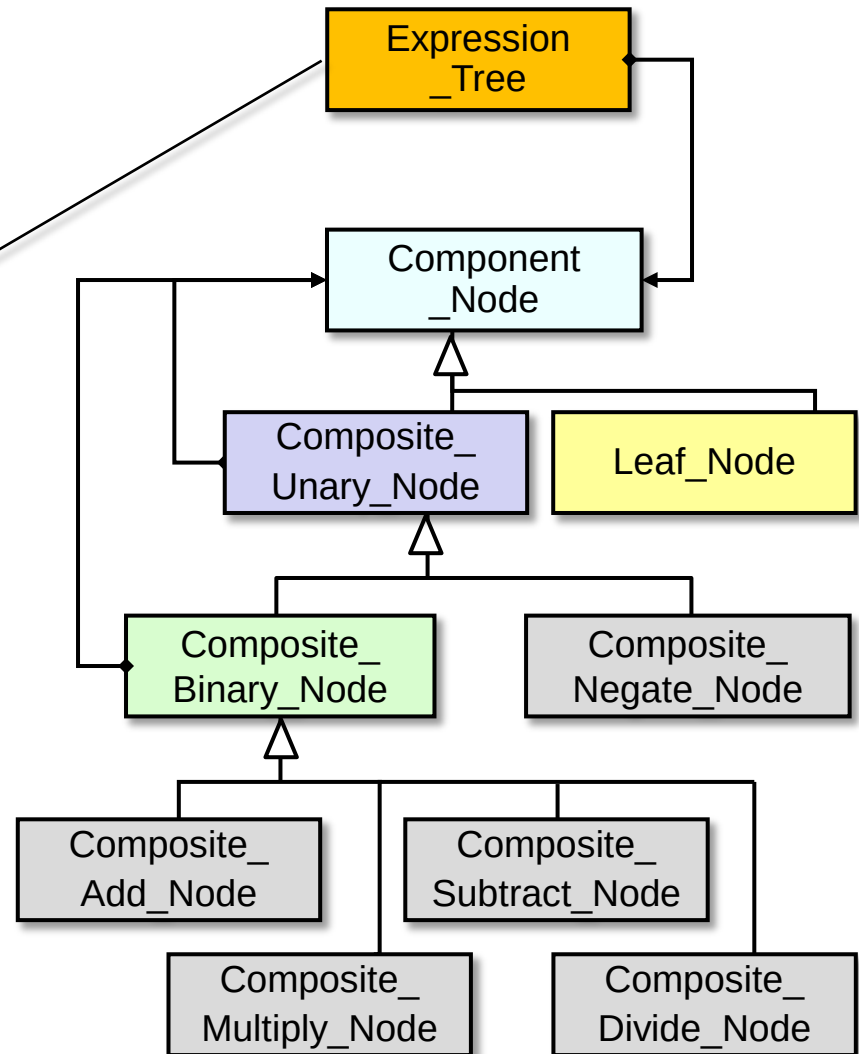


Expression
_Tree

Solution: Decouple Interface & Implementation(s)

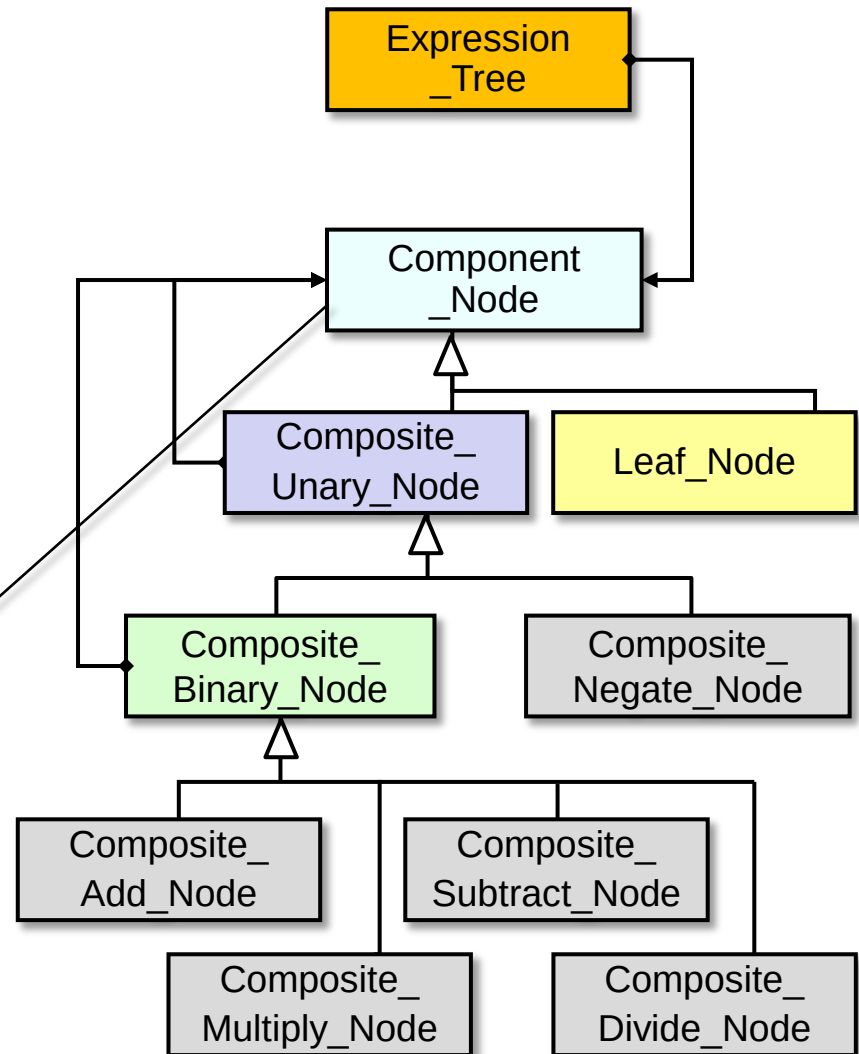
- Create an interface class (**Expression_Tree**) used by clients & an implementor hierarchy (**Component_Node** et al) that encapsulates variability

Interface class can use a C++ `std::shared_ptr` reference count object to help automate memory management



Solution: Decouple Interface & Implementation(s)

- Create an interface class (**Expression_Tree**) used by clients & an implementor hierarchy (**Component_Node** et al) that encapsulates variability



The Abstract Factory pattern can be used to produce the right implementor hierarchy (as seen later)

Expression_Tree Class Interface

- Interface for composite structure that contains all nodes in expression tree

Interface

Manage
reference
counts



```

typedef ET Iterator iterator
...
Expression_Tree(Component Node *root)
Expression_Tree(const Expression_Tree &t)
void operator=(const Expression_Tree &t)
~Expression_Tree()
bool is_null() const
const int item() const
Expression_Tree left()
Expression_Tree right()
void accept(ET Visitor &visitor)
iterator begin(const std::string &order = "")
iterator end(const std::string &order = "")
const iterator begin(const std::string order = "") const
const iterator end(const std::string &order = "") const
  
```



Plays essential role in
Iterator & Visitor
patterns (covered later)



Forward to
implementor
hierarchy

Expression_Tree Class Interface

- Interface for composite structure that contains all nodes in expression tree

Interface



Iterator trait

```
typedef ET Iterator iterator
```

```
...
```

```
Expression_Tree(Component_Node *root)
```

```
Expression_Tree(const Expression_Tree &t)
```

```
void operator=(const Expression_Tree &t)
```

```
~Expression_Tree()
```

```
bool is_null() const
```

```
const int item() const
```

```
Expression_Tree left()
```

```
Expression_Tree right()
```

Iterator

```
void accept(ET Visitor &visitor)
```

factories

```
iterator begin(const std::string &order = "")
```

```
iterator end(const std::string &order = "")
```

```
const_iterator begin(const std::string order = "") const
```

```
const_iterator end(const std::string &order = "") const
```

- Commonality:** Provides common interface for expression tree operations
- Variability:** The contents of the composite nodes in the expression tree will vary depending on the user input expression

Bridge

GoF Object Structural

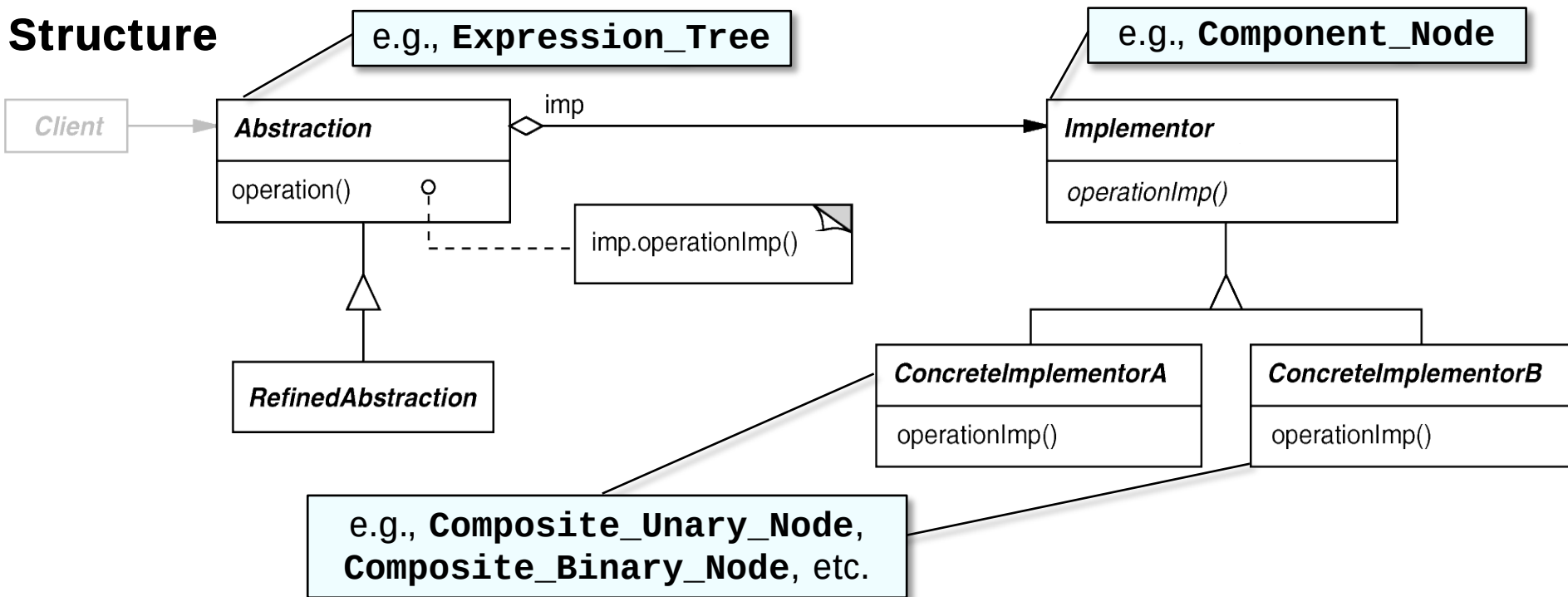
Intent

- Separate an interface from its implementation(s)

Applicability

- When interface & implementation should vary independently
- Require a uniform interface to interchangeable implementor hierarchies

Structure



Bridge

GoF Object Structural

Bridge example in C++

- Separate expression tree interface from the composite node implementations

```
class Expression_Tree {  
public:  
    Expression_Tree(Component_Node *root): root_(root) {}
```

root_ manages lifecycle of pointer parameter 

Interface forwards to implementor via shared_ptr 

```
    ...  
    void accept(ET_Visitor &v) { root_->accept(v); }
```

```
private:  
    std::shared_ptr <Component_Node> root_;
```

C++11/Boost "smart pointer" that handles reference counting 

Bridge

GoF Object Structural

Bridge example in C++

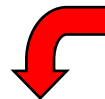
- Separate expression tree interface from the composite node implementations

```
class Expression_Tree {
public:
    Expression_Tree(Component_Node *root): root_(root) {}

    ...
    void accept(ET_Visitor &v) { root_->accept(v); }

private:
    std::shared_ptr <Component_Node> root_;

    ...
    Expression_Tree expr_tree
        (new Composite_Multiply_Node(u1, b1));
```



**expr_tree manages lifecycle of
composite via *Bridge* pattern**

Bridge

GoF Object Structural

Consequences

- + Abstraction interface & implementor hierarchy are decoupled
- + Implementors can vary dynamically
- + Enables efficient use of value-based STL algorithms & containers
- One-size-fits-all abstraction & implementor interfaces

Bridge

GoF Object Structural

Consequences

- + Abstraction interface & implementor hierarchy are decoupled
- + Implementors can vary dynamically
- + Enables efficient use of value-based STL algorithms & containers
- One-size-fits-all abstraction & implementor interfaces

Implementation

- Sharing implementors & reference counting
 - e.g., C++11/Boost **shared_ptr**
- Creating the right implementor
 - Often addressed by using factories
- Not as widely used in Java as in C++

Bridge

GoF Object Structural

Consequences

- + Abstraction interface & implementor hierarchy are decoupled
- + Implementors can vary dynamically
- + Enables efficient use of value-based STL algorithms & containers
- One-size-fits-all abstraction & implementor interfaces

Implementation

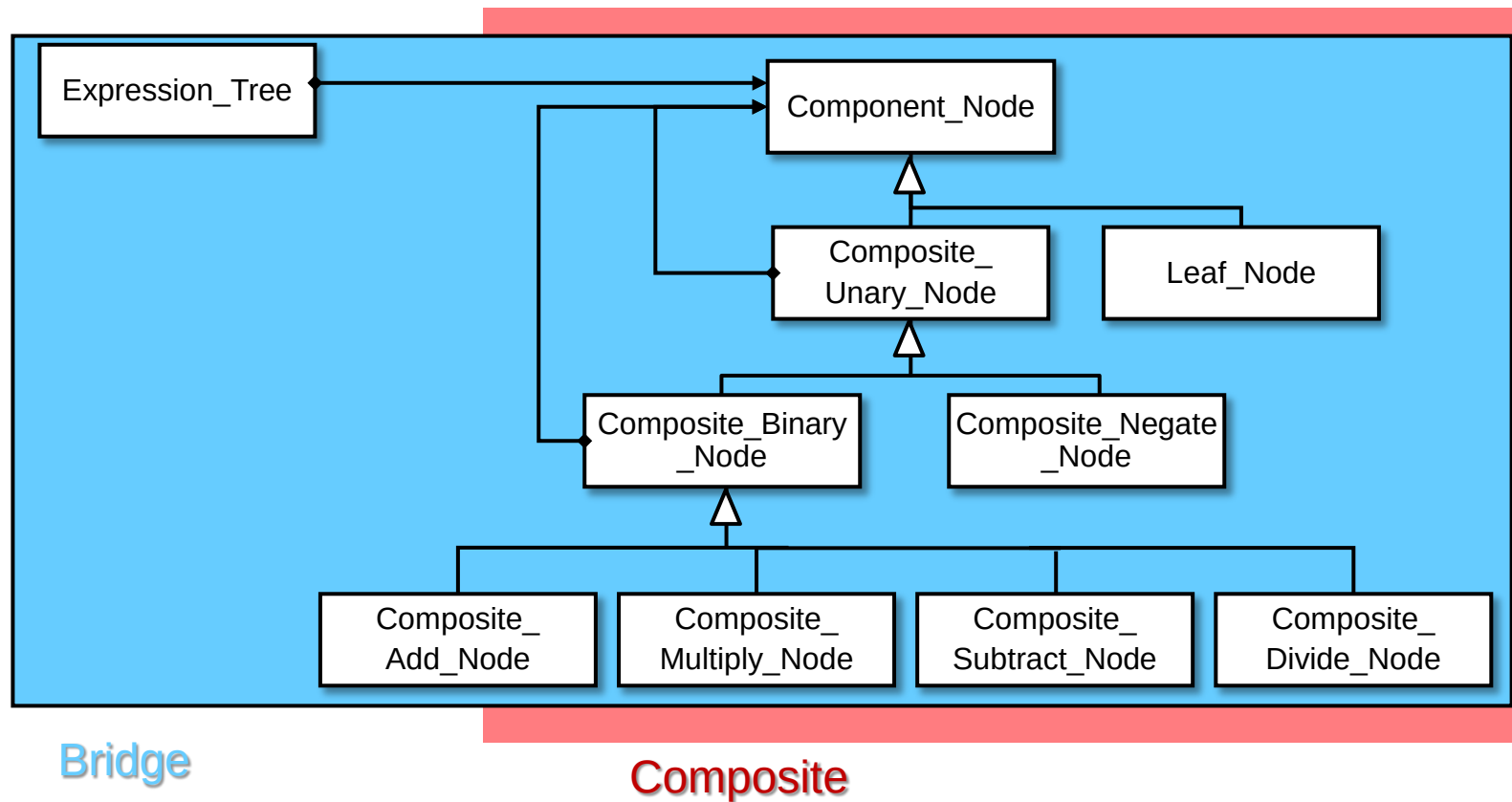
- Sharing implementors & reference counting
 - e.g., C++11/Boost `shared_ptr`
- Creating the right implementor
 - Often addressed by using factories
- Not as widely used in Java as in C++

Known Uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- AWT Component/ComponentPeer
- Java Socket/SocketImpl
- Java synchronizers
- ACE Reactor framework

Summary of Tree Structure & Access Patterns

Composite is used to define internal data structure for the expression tree processing app & *Bridge* simplifies access to this data structure for C++ code



Bridge Composite is an example of a “pattern compound”

A Case Study of “Gang of Four” (GoF) Patterns : Part 6

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

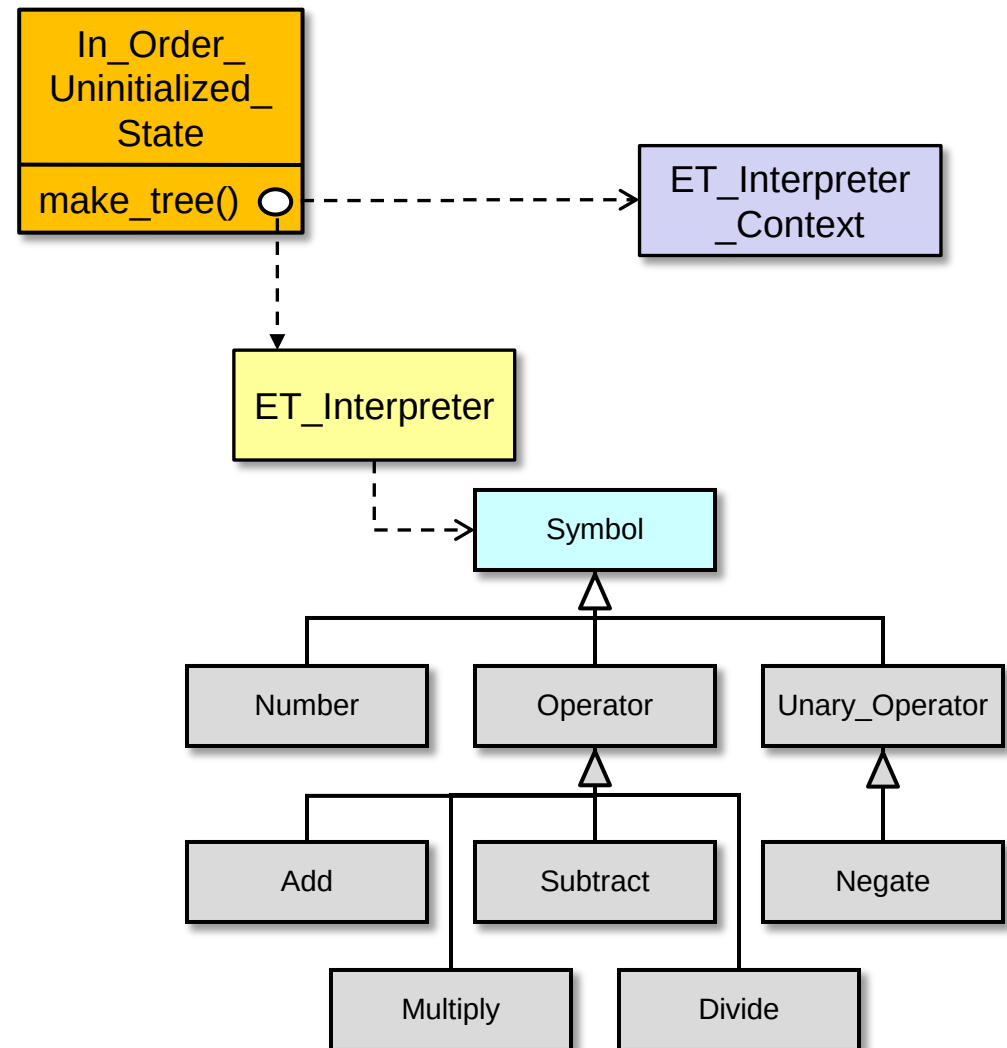
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



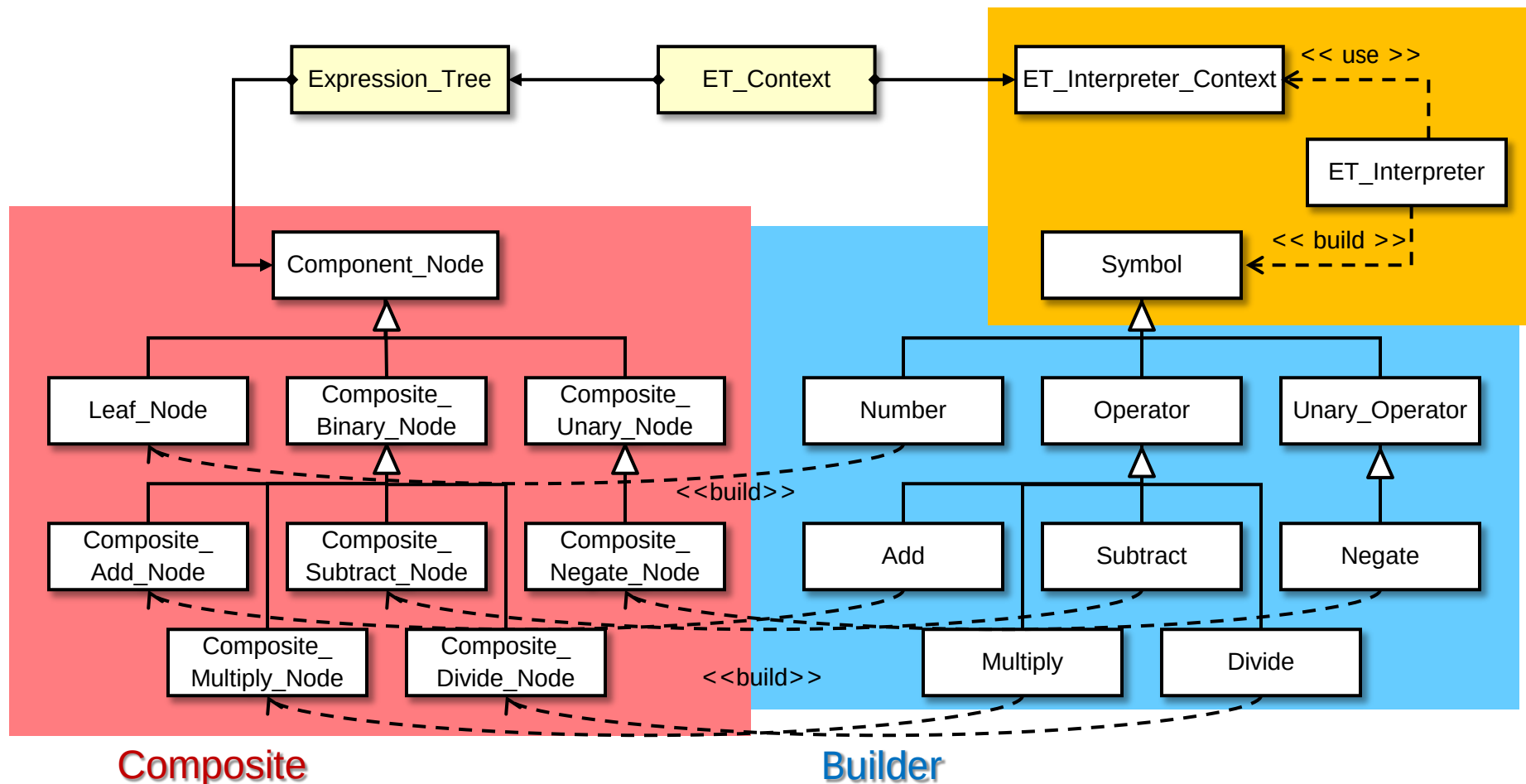
Topics Covered in this Part of the Module

- 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
- Explore patterns for
 - Tree structure & access
 - Tree creation



Overview of Tree Creation Patterns

Purpose: Parse user input expression & create the internal data structure for the expression tree



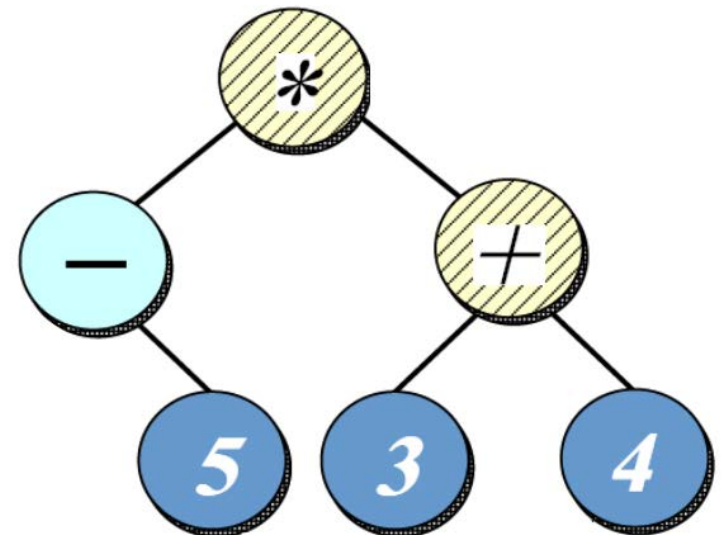
There are *many* classes, but only a handful of patterns involved in this design

Problem: Parsing Expressions & Creating an Expression Tree

Goals

- Simplify & centralize the creation of all nodes in the composite expression tree

“in-order” expression = $-5*(3+4)$



Problem: Parsing Expressions & Creating an Expression Tree

Goals

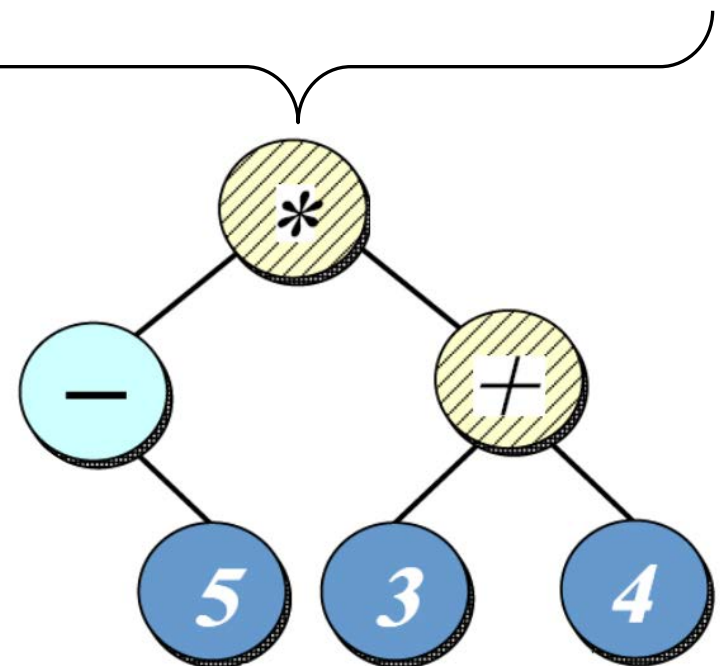
- Simplify & centralize the creation of all nodes in the composite expression tree
- Be extensible for future types of expression formats & optimizations

“pre-order” expression = $*-5+34$

“post-order” expression = $5-34+*$

“level-order” expression = $*-+534$

“in-order” expression = $-5*(3+4)$



Problem: Parsing Expressions & Creating an Expression Tree

Goals

- Simplify & centralize the creation of all nodes in the composite expression tree
- Be extensible for future types of expression formats & optimizations

Constraints/forces

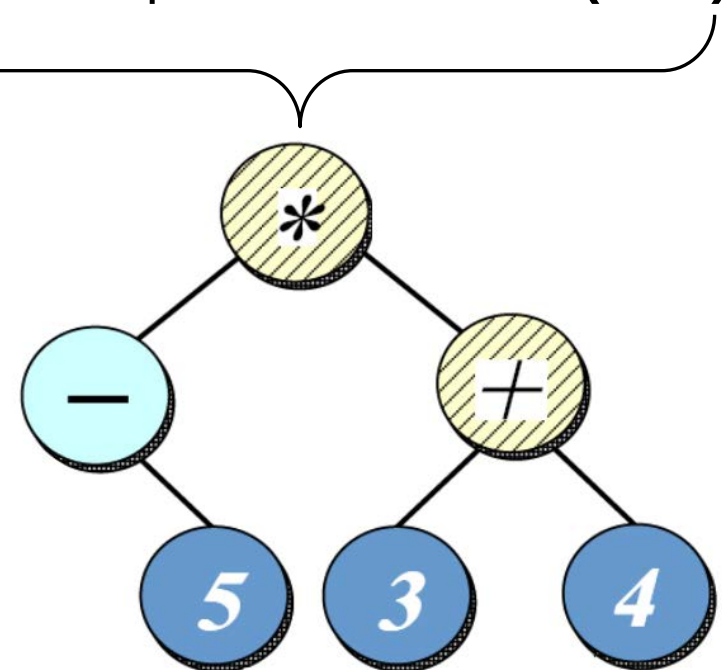
- Don't recode existing clients when new types of expression formats are added
- Add new user input expressions without recompiling existing code

“pre-order” expression = $*-5+34$

“post-order” expression = $5-34+*$

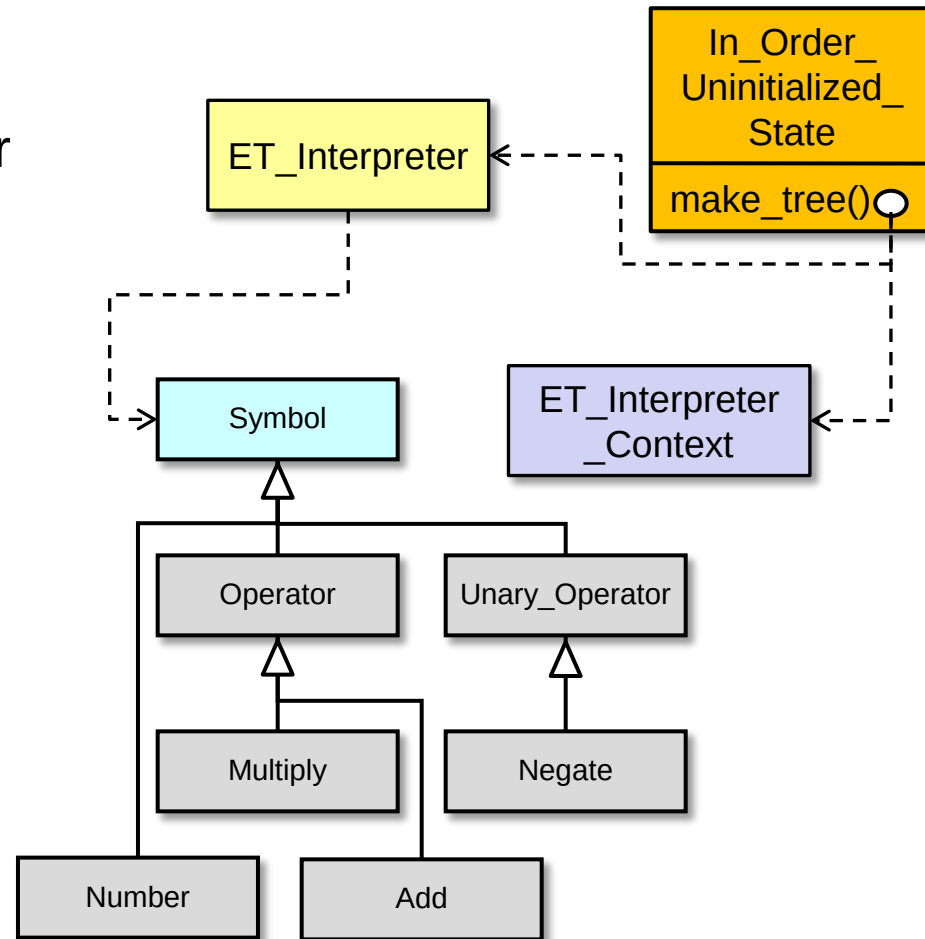
“level-order” expression = $*-+534$

“in-order” expression = $-5*(3+4)$



Solution: Build Expression Tree Using Interpreter

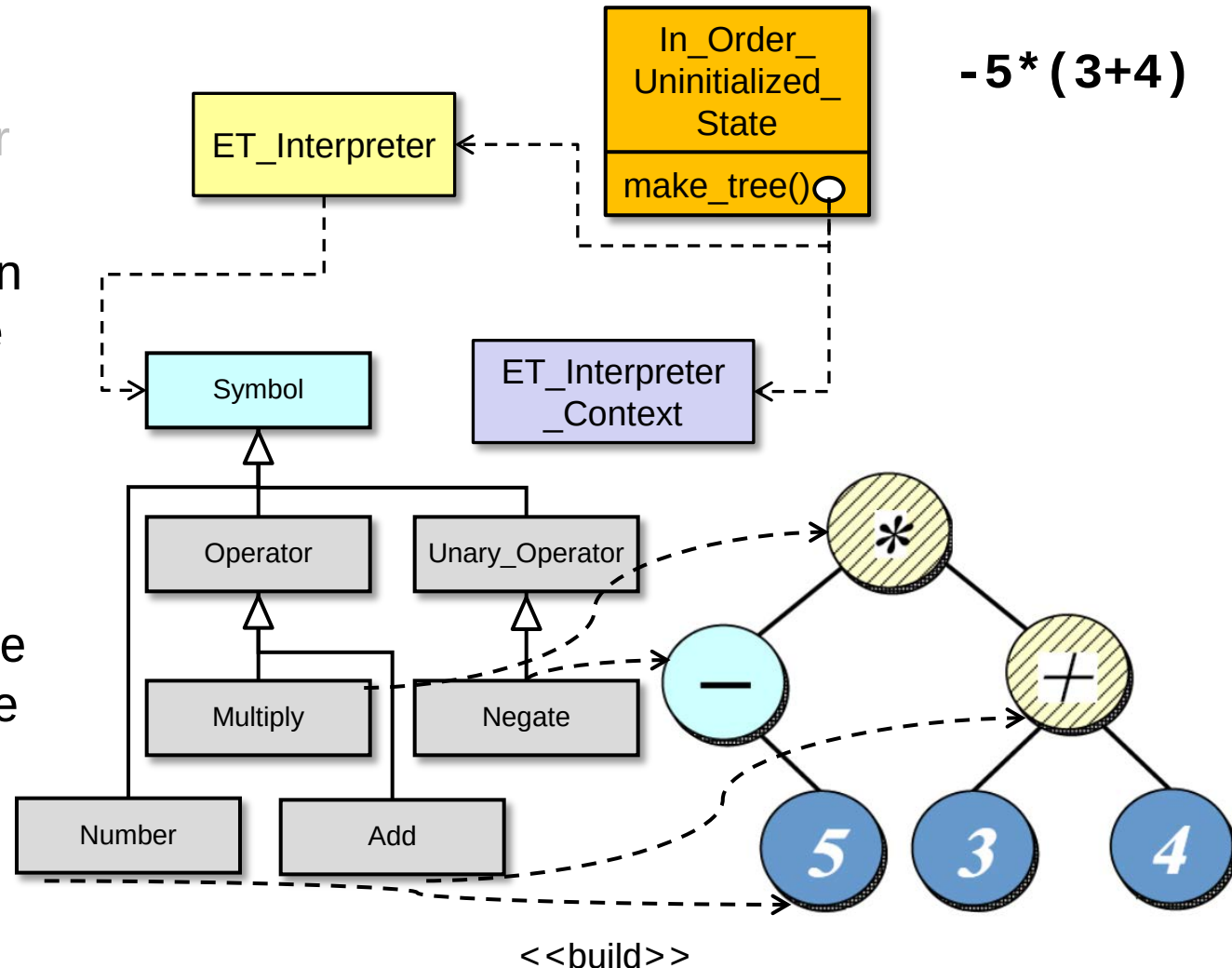
- Use an interpreter to create a *parse tree* corresponding to user input expression



-5*(3+4)

Solution: Build Expression Tree Using Interpreter

- Use an interpreter to create a *parse tree* corresponding to user input expression
- This parse tree is then traversed to build the appropriate type of nodes in the corresponding *expression tree*
- We create the entire parse tree to enable optimizations (as a future extension)



ET_Interpreter Class Interface

- Transforms expressions into a parse tree & then generates the corresponding expression tree

Interface

```
virtual ~ET_Interpreter()  
Expression_Tree interpret(ET_Interpreter_Context &context,  
                           const std::string &input)  
    <<uses>>  
ET_Interpreter_Context()  
~ET_Interpreter_Context()  
int get(std::string variable)  
void set(std::string variable,  
         int value)  
void print()  
void reset()
```



The context can be used to implement
setters/getters for variable leaf nodes

ET_Interpreter Class Interface

- Transforms expressions into a parse tree & then generates the corresponding expression tree

Interface

```

virtual ~ET_Interpreter()
Expression_Tree interpret(ET_Interpreter_Context &context,
                          const std::string &input)

```

<<creates>>

Abstract base class of
all parse tree nodes

→

```

Symbol(Symbol *left,
        Symbol *right)
virtual ~Symbol()
virtual int precedence()=0
virtual
Component_Node * build()=0

```

This method is used to build a component node
corresponding to the expression parse tree node



ET_Interpreter Class Interface

- Transforms expressions into a parse tree & then generates the corresponding expression tree

Interface

```

virtual ~ET_Interpreter()
Expression_Tree interpret(ET_Interpreter_Context &context,
                          const std::string &input)
    <<uses>>
    ET_Interpreter_Context()
    ~ET_Interpreter_Context()
    int get(std::string variable)
    void set(std::string variable,
             int value)
    void print()
    void reset()
    <<creates>>
    Symbol(Symbol *left,
            Symbol *right)
    virtual ~Symbol()
    virtual int precedence()=0
    virtual
    Component_Node * build()=0
  
```

- Commonality:** Provides a common interface building parse trees & expression trees from user input expressions
- Variability:** The structure of the parse trees & expression trees can vary depending on the format & contents of the expression input

Interpreter

GoF Class Behavioral

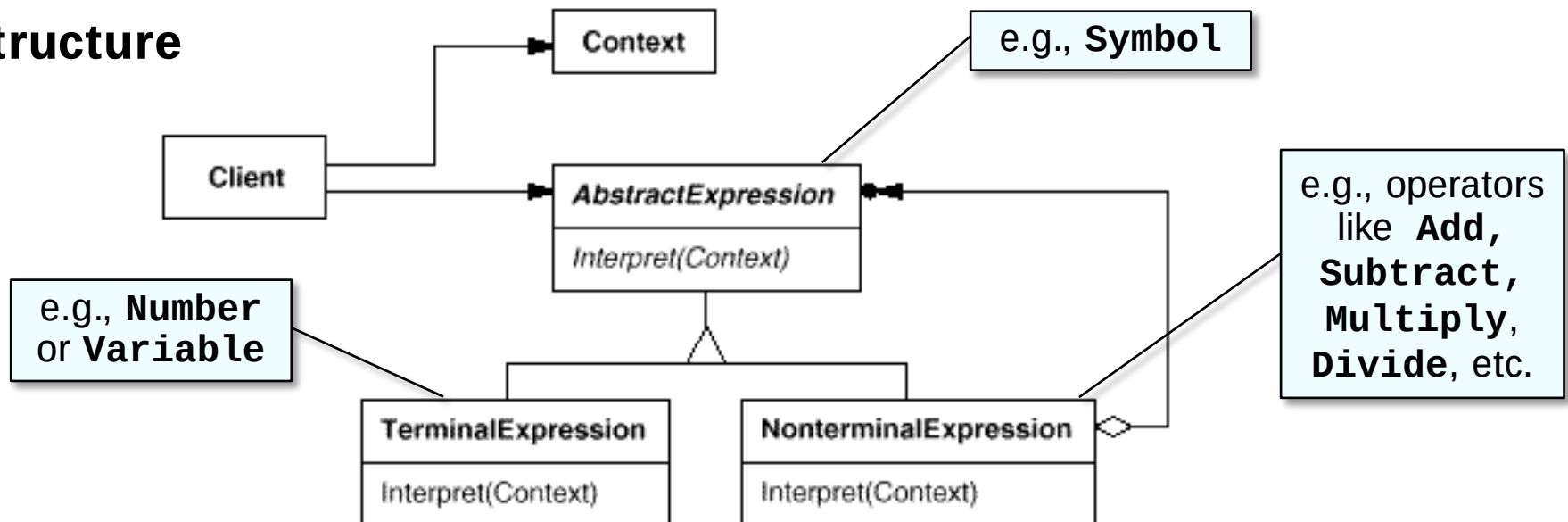
Intent

- Given a language, define a representation for its grammar, along with an interpreter that uses the representation to interpret sentences in the language

Applicability

- When the grammar is simple & relatively stable
- When time/space efficiency is not a critical concern

Structure



Interpreter

GoF Class Behavioral

Interpreter example in C++

- The `interpret()` method creates a parse tree from user's expression input

```
Expression_Tree ET_Interpreter::interpret
```

```
(ET_Interpreter_Context &context, const std::string &input) {  
    std::list<Symbol *> parse_tree;  Symbols added to parse tree as  
    ... interpreter recognizes them
```

```
    if(is_number(input[i])){  Handle operand  
        number_insert(input, parse_tree);
```

```
    } else if(input[i] == '+'){  Handle addition operator  
        Add *op = new Add(); op->add_precedence(accum_precedence);  
        precedence_insert(op, parse_tree);
```

```
    ...  
    } else if(input[i] == '(') {  Handle parenthesized expression  
        handle_parenthesis(context, input,  
                            accum_precedence, parse_tree);  
    }  
    ...
```

Interpreter

GoF Class Behavioral

Consequences

- + Simple grammars are easy to change & extend
 - e.g., All rules represented by distinct classes in a consistent & orderly manner
- + Adding another rule adds another class
- Complex grammars hard to create & maintain
 - e.g., More inter-dependent rules yield more inter-dependent classes

Complex grammars may require different approach (e.g., parser generators)

Interpreter

GoF Class Behavioral

Consequences

- + Simple grammars are easy to change & extend
 - e.g., All rules represented by distinct classes in a consistent & orderly manner
- + Adding another rule adds another class
- Complex grammars hard to create & maintain
 - e.g., More inter-dependent rules yield more inter-dependent classes

Implementation

- Express language rules, one per class
 - Literal translations expressed as *terminal expressions*
 - Alternations, repetitions, or sequences expressed as *nonterminal expressions*

Interpreter

GoF Class Behavioral

Consequences

- + Simple grammars are easy to change & extend
 - e.g., All rules represented by distinct classes in a consistent & orderly manner
- + Adding another rule adds another class
- Complex grammars hard to create & maintain
 - e.g., More inter-dependent rules yield more inter-dependent classes

Implementation

- Express language rules, one per class
 - Literal translations expressed as *terminal expressions*
 - Alternations, repetitions, or sequences expressed as *nonterminal expressions*

Known Uses

- Text editors & web browsers use *Interpreter* to lay out documents & check spelling
 - e.g., an equation in TeX is represented as a tree where internal nodes are operators & leaves are variables
- Smalltalk compilers
- The QOCA constraint-solving toolkit

Builder

GoF Object Creational

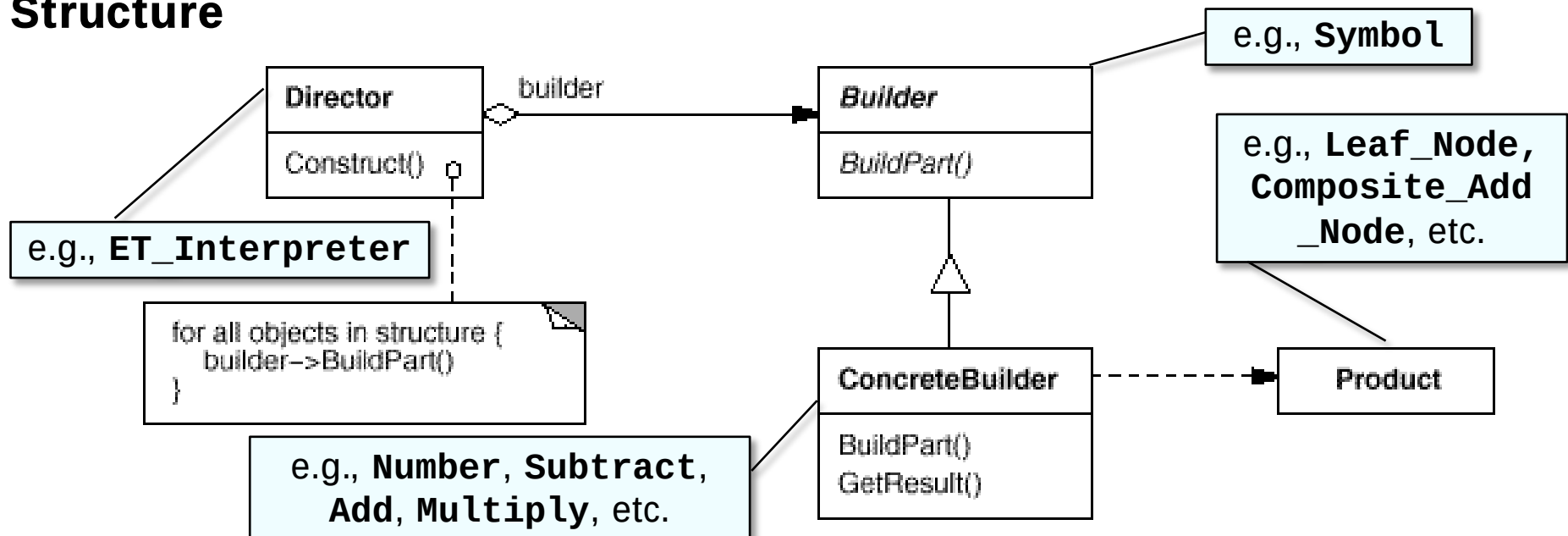
Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations

Applicability

- Need to isolate knowledge of the creation of a complex object from its parts
- Need to allow different implementations/interfaces of an object's parts

Structure



Builder

GoF Object Creational

Builder example in C++

- The `interpret()` method builds composite expression tree from parse tree

```
Expression_Tree ET_Interpreter::interpret  
    (ET_Interpreter_Context &context, const std::string &input) {  
    ...  
    return Expression_Tree(parse_tree.back()->build());  
}
```

Invoke a recursive expression tree build, starting with
the root symbol in parse tree created by the interpreter 

Builder

GoF Object Creational

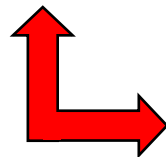
Builder example in C++

- The `interpret()` method builds composite expression tree from parse tree

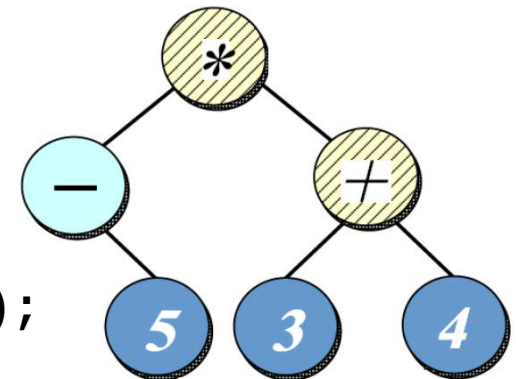
```
Expression_Tree ET_Interpreter::interpret
(ET_Interpreter_Context &context, const std::string &input) {
    ...
    return Expression_Tree(parse_tree.back()->build());
}
```

```
Component_Node *
Multiply::build() {
    return new Composite_Multiply_Node
        (left_->build(), right_->build());
}
```

Build equivalent
component nodes



```
Component_Node *Number::build() {
    return new Leaf_Node(item_);
}
```



Builder

GoF Object Creational

Consequences

- + Can vary a product's internal representation
- + Isolates code for construction & representation
- + Finer control over the construction process
- May involve a lot of classes

Builder

GoF Object Creational

Consequences

- + Can vary a product's internal representation
- + Isolates code for construction & representation
- + Finer control over the construction process
- May involve a lot of classes

Implementation

- The Builder pattern is a “factory pattern with a mission”
- A Builder pattern implementation exposes itself as a factory
 - It goes beyond conventional factory patterns by connecting various implementations together

Builder

GoF Object Creational

Consequences

- + Can vary a product's internal representation
- + Isolates code for construction & representation
- + Finer control over the construction process
- May involve a lot of classes

Implementation

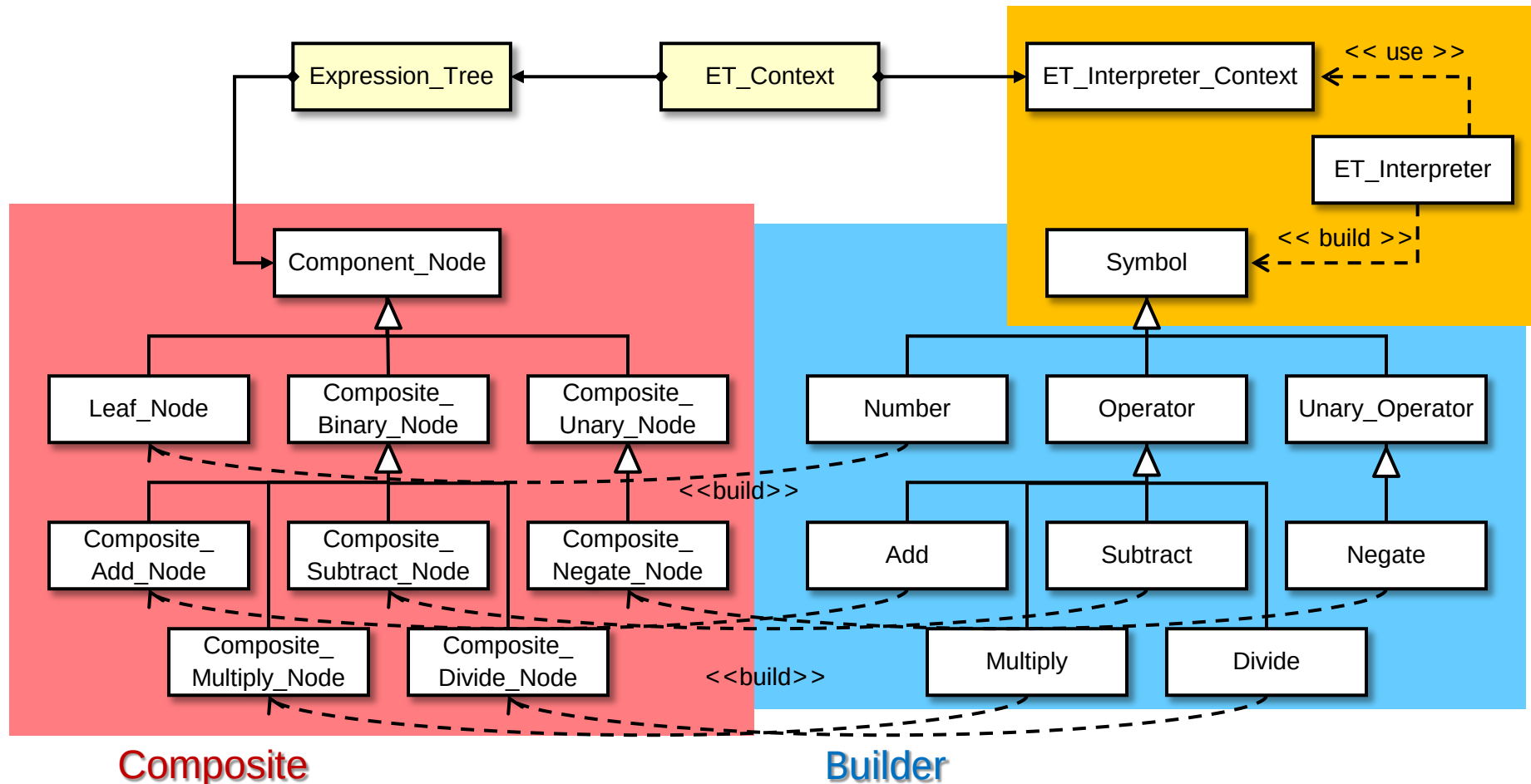
- The Builder pattern is a “factory pattern with a mission”
- A Builder pattern implementation exposes itself as a factory
 - It goes beyond conventional factory patterns by connecting various implementations together

Known Uses

- ET++ RTF converter application
- Smalltalk-80
- ACE Service Configurator framework

Summary of Tree Creation Patterns

Interpreter & Builder patterns create internal data structure for expression tree from user input



Interpreter, Builder, & Composite are powerful “pattern sequence”

