A Case Study of "Gang of Four" (GoF) Patterns : Part 9

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

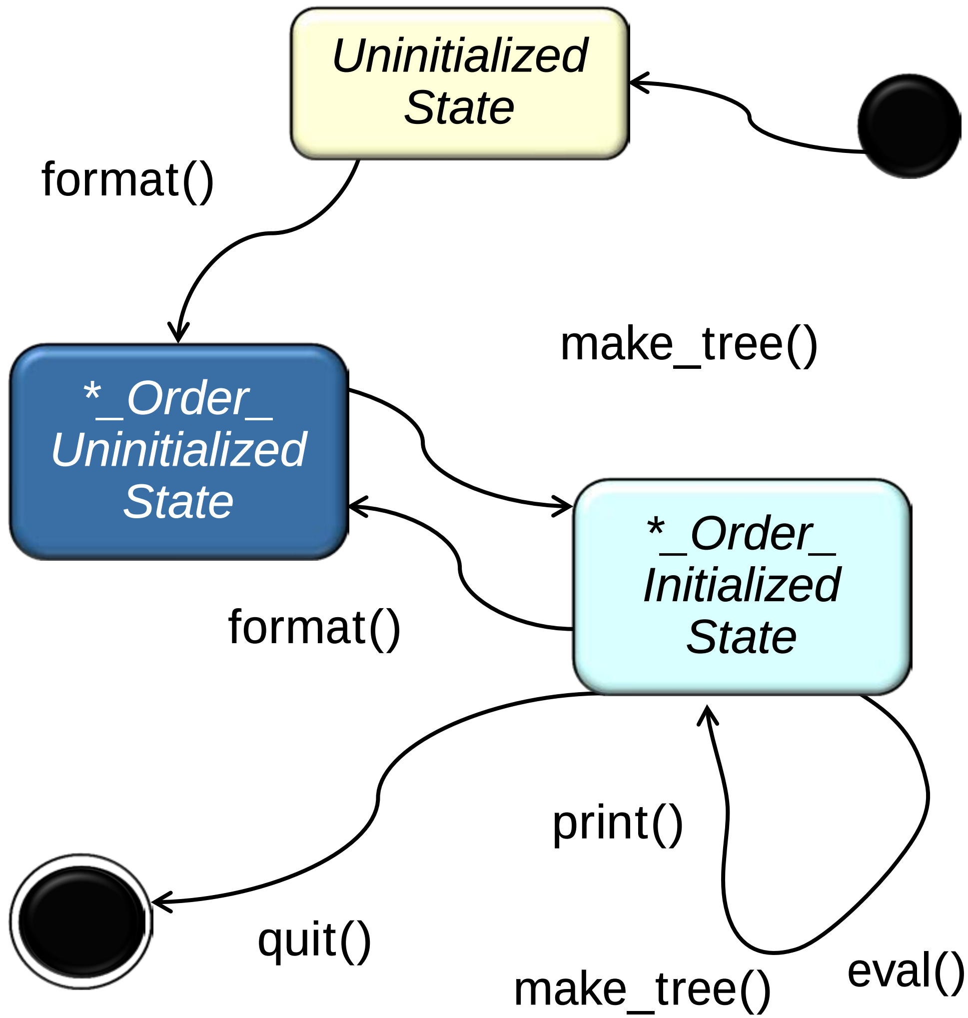
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

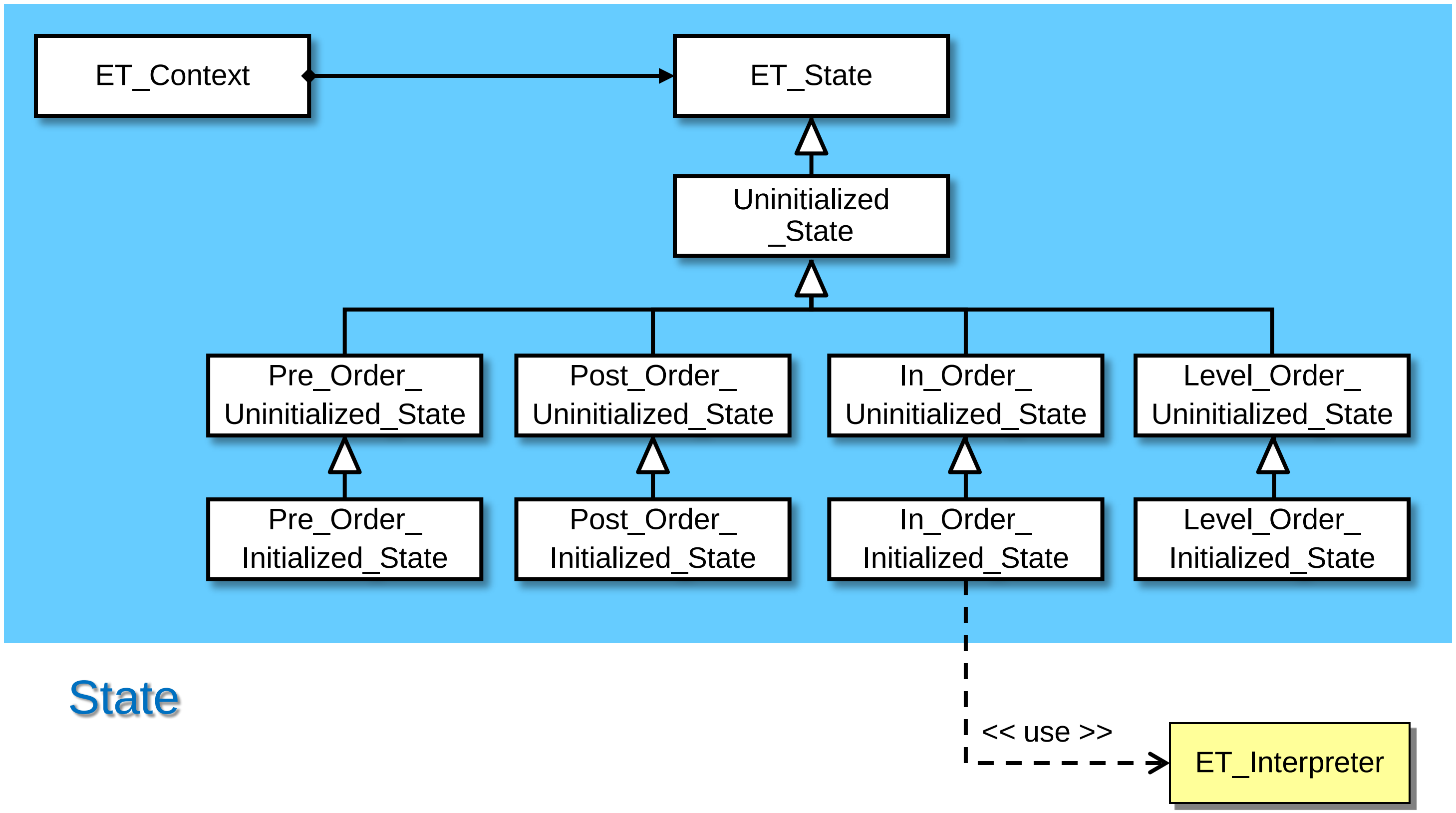
Topics Covered in this Part of the Module

- 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
  - Tree traversal
  - Commands & factories
  - Command ordering protocols



Overview of a Command Protocol Pattern

***Purpose**: Ensure user commands are performed in the correct order*



This pattern uses design of classes to explicitly order user commands correctly

Problem: Ensuring Correct Command Protocol

Goals

- Ensure that users follow the correct protocol when entering commands

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-
       order|level-order]
eval [post-order]
quit
> format in-order
> print in-order
Error: ET_State::print called in
invalid state
```

Protocol violation

Problem: Ensuring Correct Command Protocol

Goals

- Ensure that users follow the correct protocol when entering commands

Constraints/forces

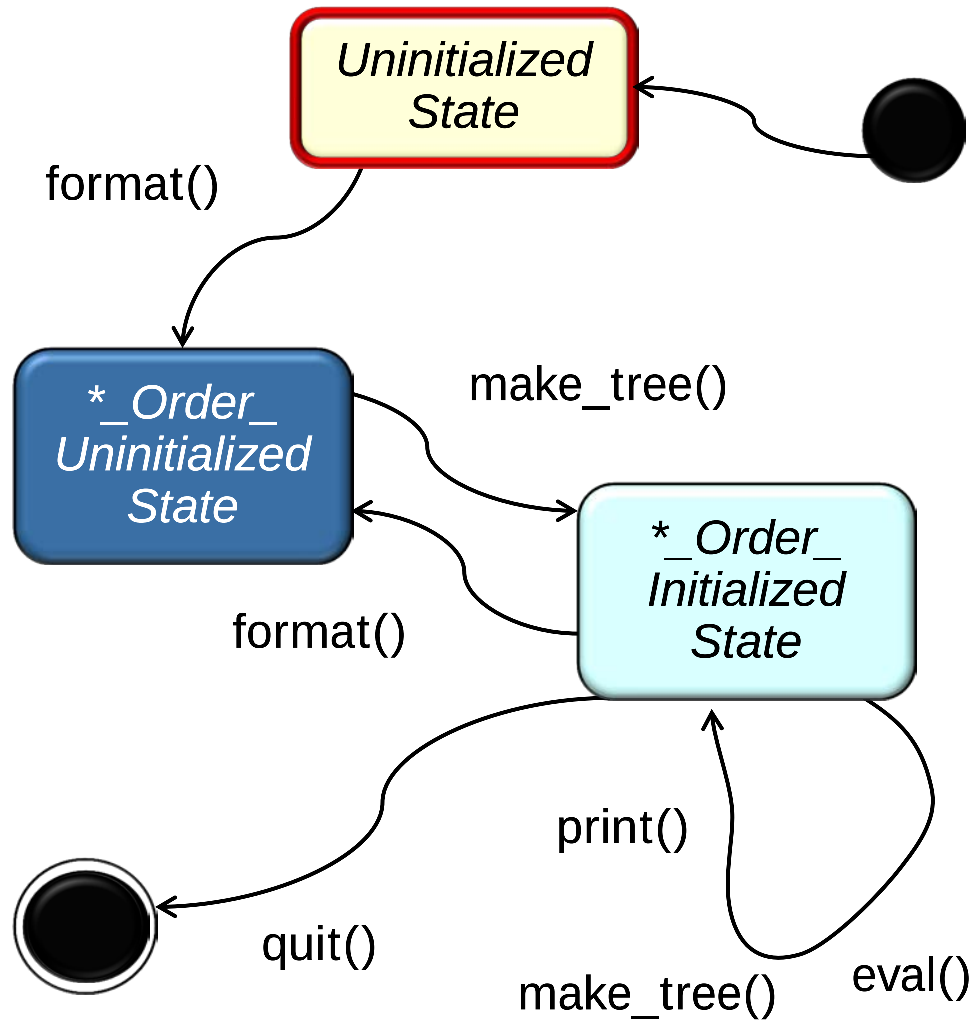
- Must consider context of previous commands to determine protocol conformance, e.g.,
  - **`format`** must be called first
  - **`expr`** must be called before **`print`** or **`eval`**
  - **`print`** & **`eval`** can be called in any order

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-
       order|level-order]
eval [post-order]
quit
> format in-order
> print in-order
Error: ET_State::print called in
invalid state
```

Protocol violation

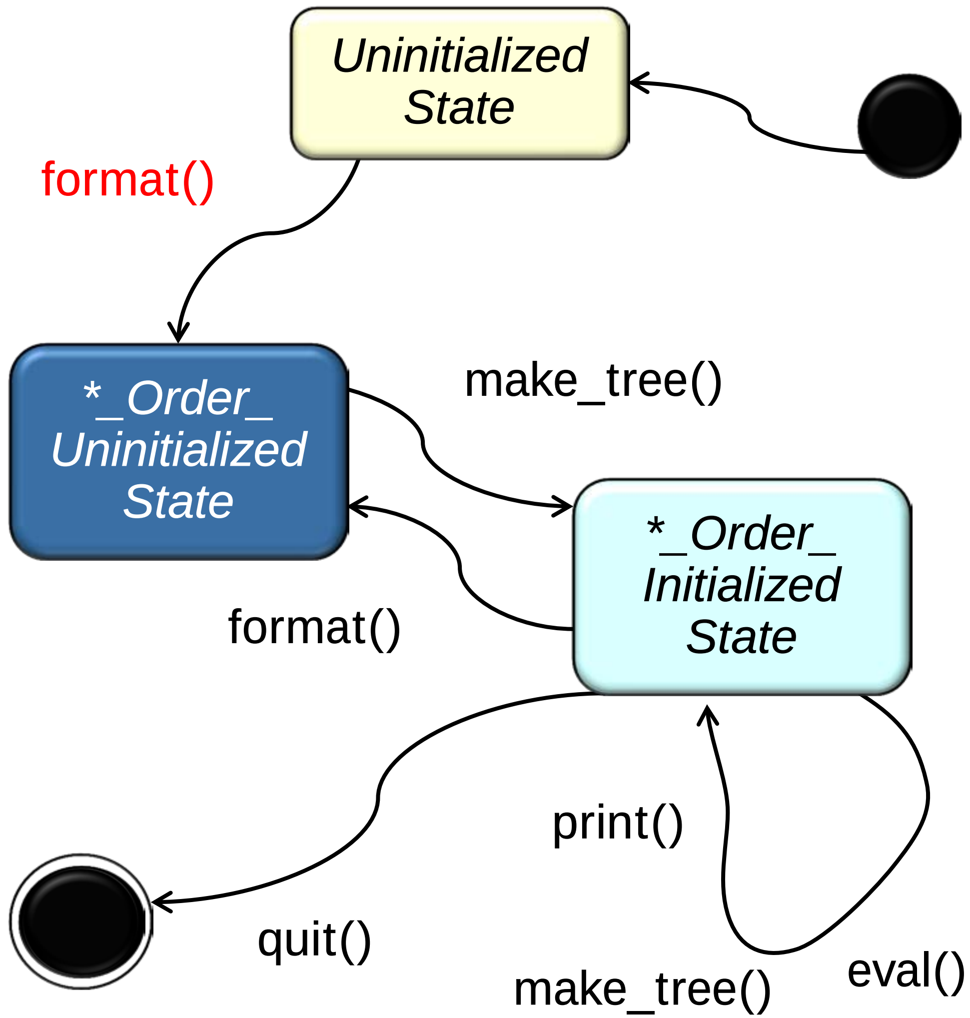
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



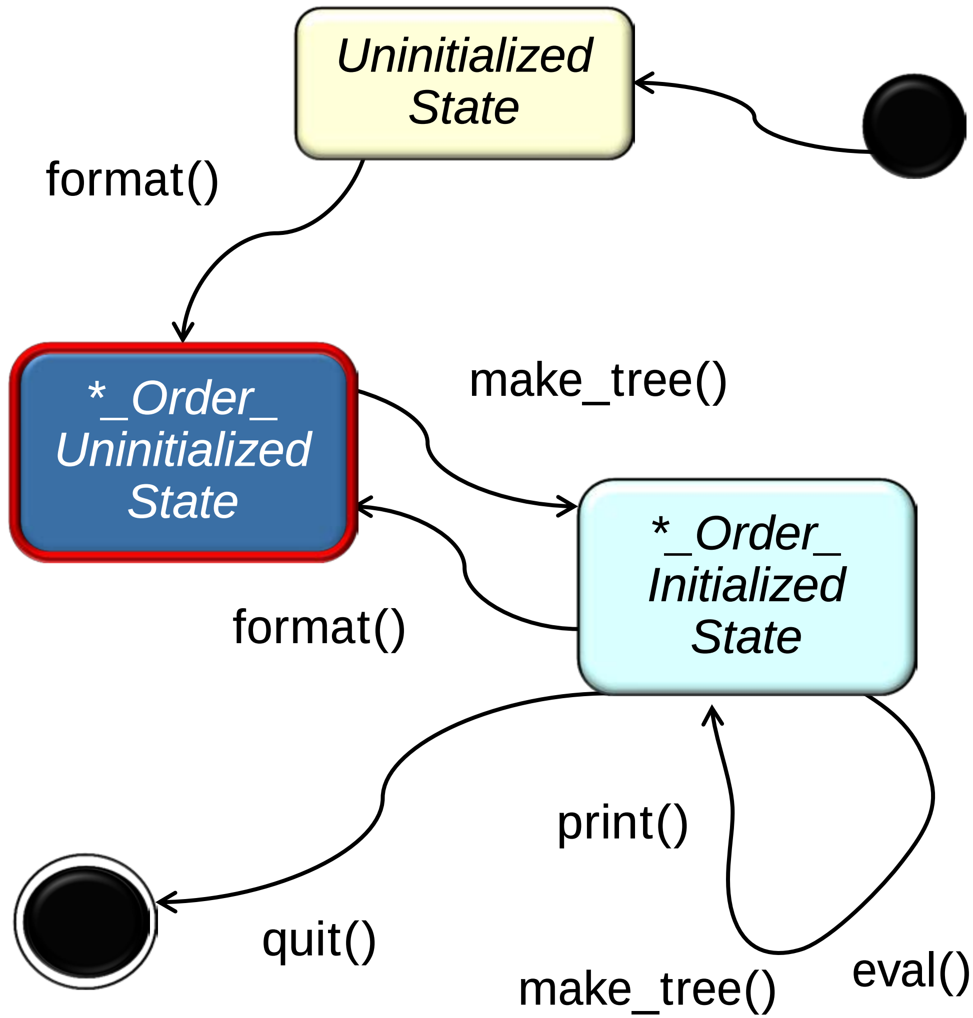
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



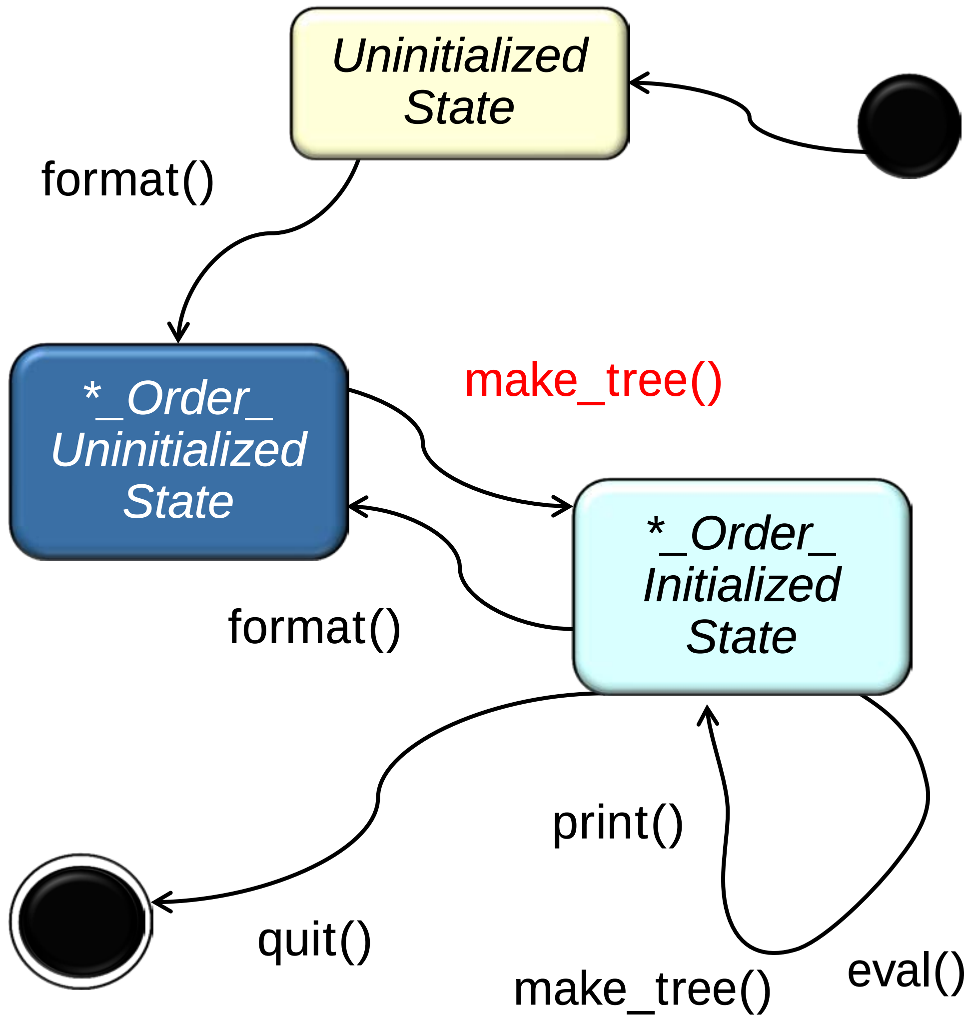
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



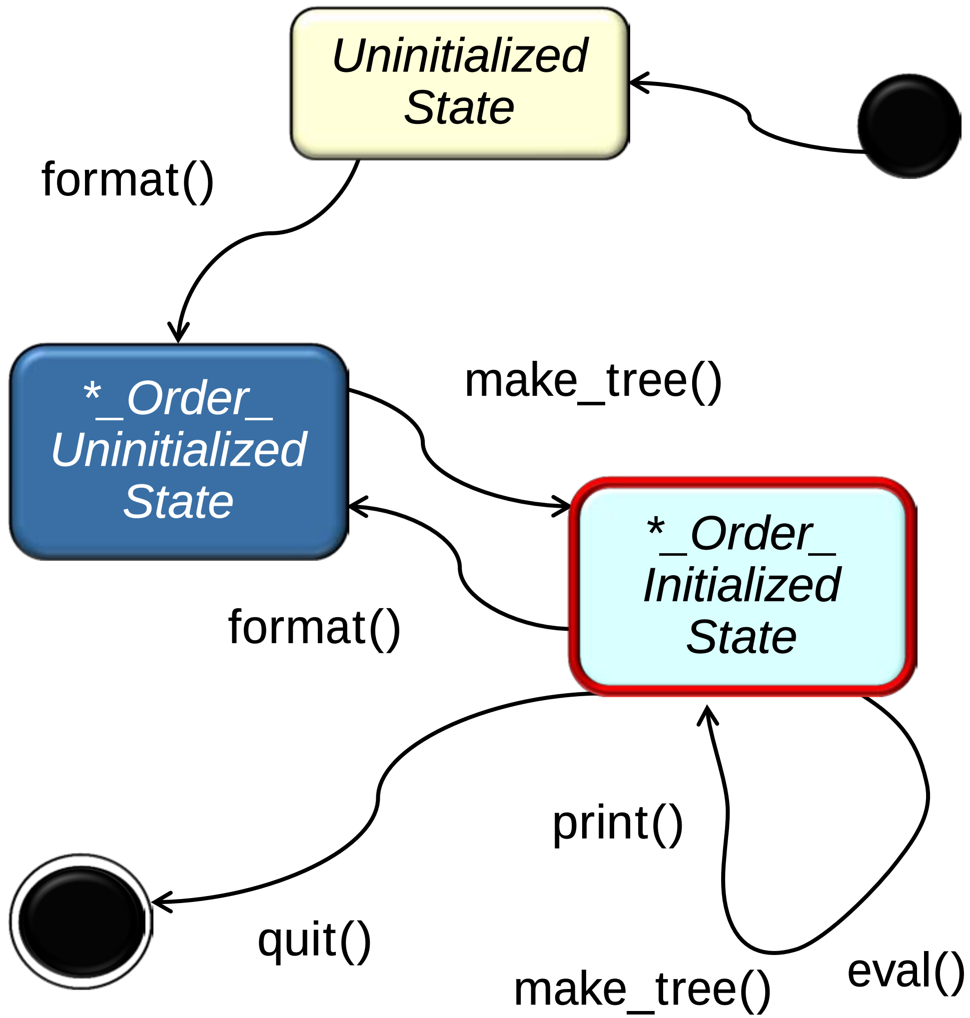
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



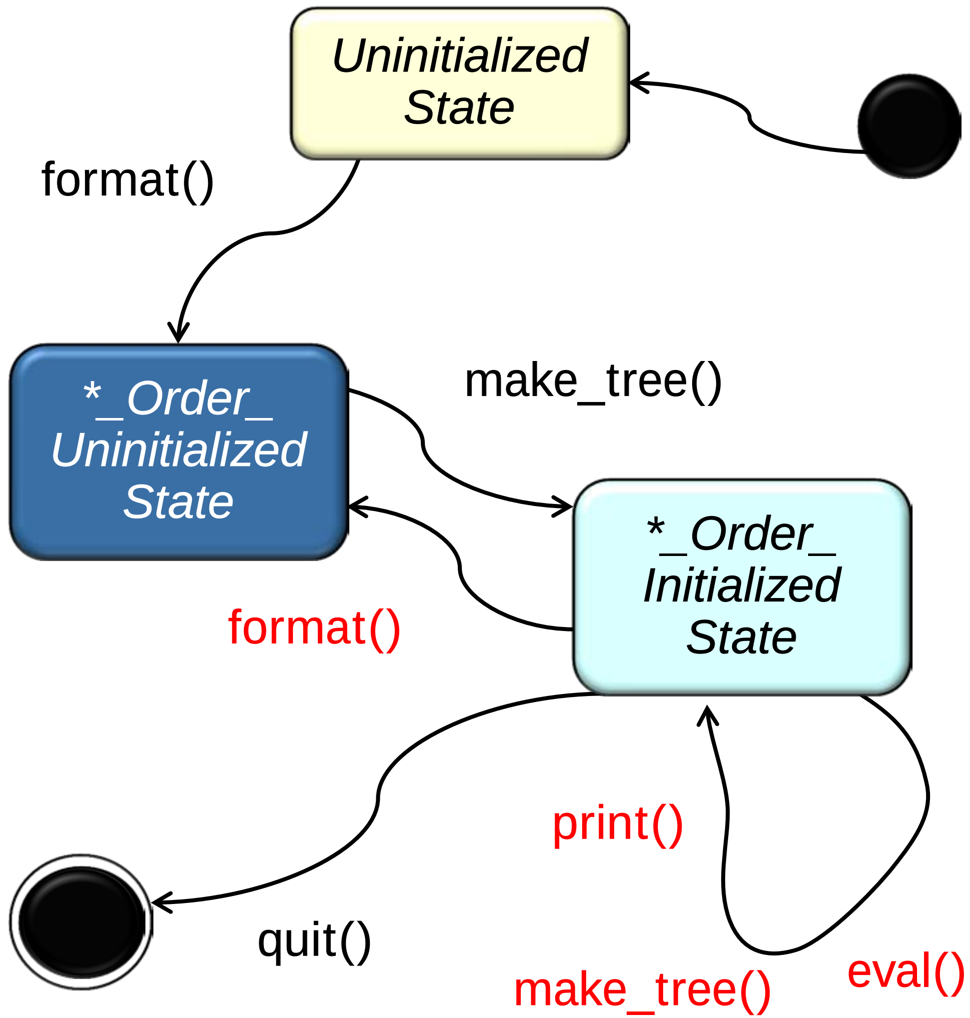
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



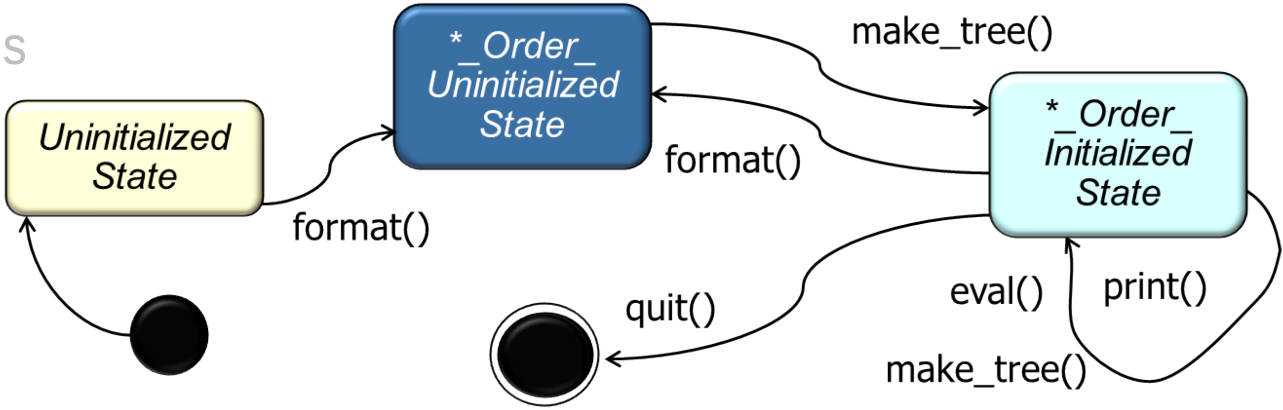
Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine

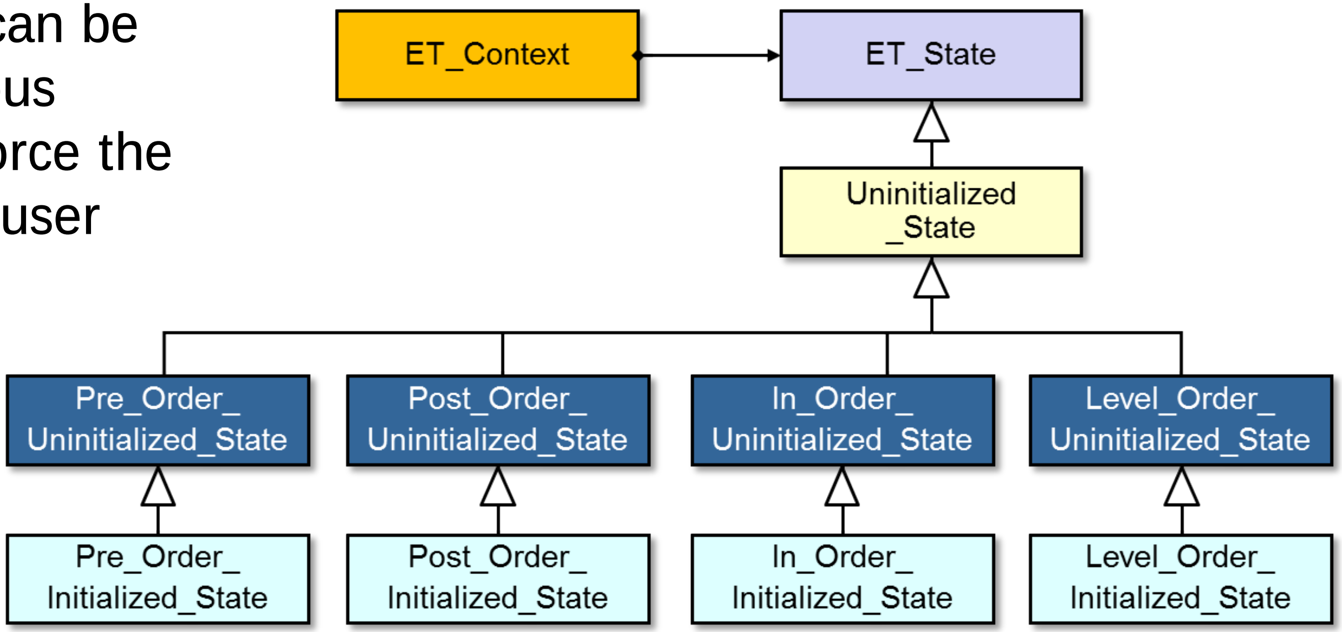


Solution: Encapsulate Command History as States

- Handling user commands depends on history of prior commands
  - This history can be represented as a state machine



- The state machine can be encoded using various subclasses that enforce the correct protocol for user commands



ET_Context also encapsulates variability & simplifies memory management

ET_Context Class Interface

- Interface used to ensure commands are invoked according to correct protocol

These methods correspond to user commands

Interface

```
              void format(const std::string &new_format)
              void make_tree(const std::string &expression)
              void print(const std::string &format)
              void evaluate(const std::string &format)
                   ...
        ET_State * state() const
              void state(ET_State *new_state)
Expression_Tree & tree()
              void tree(const Expression_Tree &new_tree)
```

Setter/getter for ET_State subclasses

- **Commonality**: Provides a common interface for ensuring that expression tree commands are invoked according to the correct protocol
- **Variability**: The implementations—& correct functioning—of the expression tree commands can vary depending on the requested operations & the current state

ET_State Class Interface

- Implementation used to define the various states that affect how users commands are processed

Interface

These methods are delegated from ET_Context methods

```
virtual void  format(ET_Context &context,
                     const std::string &new_format)
virtual void  make_tree(ET_Context &context,
                        const std::string &expression)
virtual void  print(ET_Context &context,
                    const std::string &format)
virtual void  evaluate(ET_Context &context,
                       const std::string &format)
```

- **Commonality**: Provides a common interface for ensuring that expression tree commands are invoked according to the correct protocol
- **Variability**: The implementations—& correct functioning—of the expression tree commands can vary depending on the requested operations & the current state

State GoF Object Behavioral

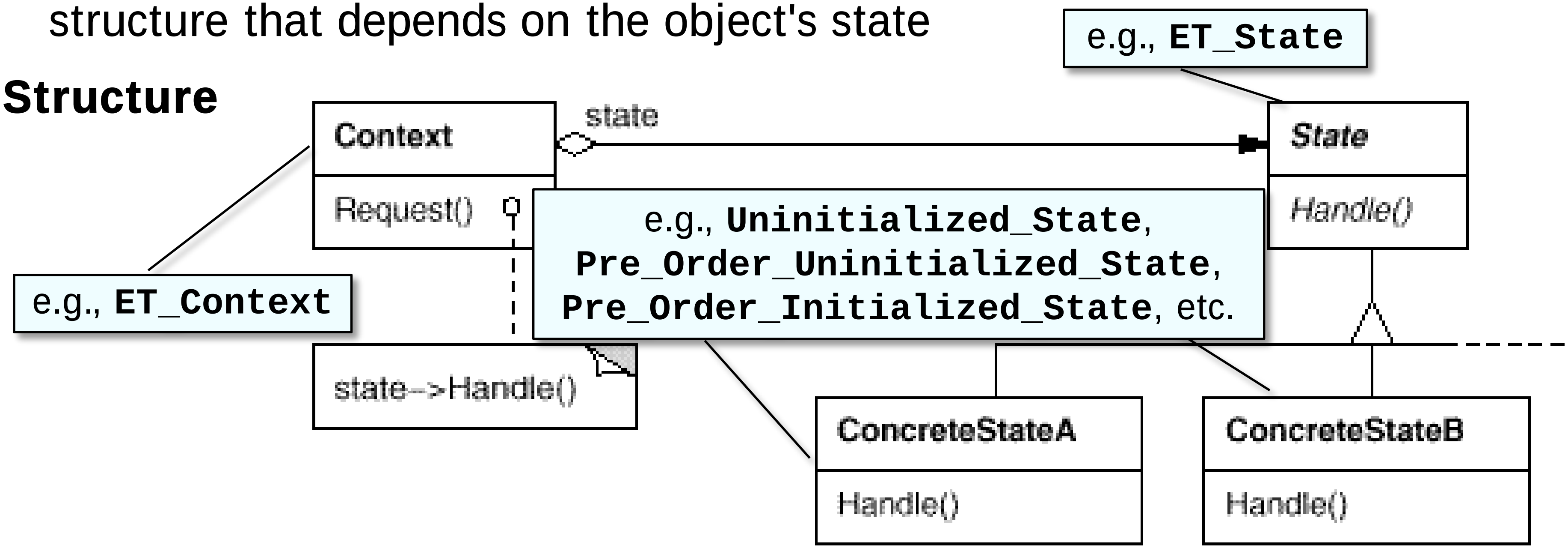
Intent

- Allow an object to alter its behavior when its internal state changes—the object will appear to change its class

Applicability

- When an object must change its behavior at run-time depending on which state it is in
- When several operations have the same large multipart conditional structure that depends on the object's state

Structure



State GoF Object Behavioral

State example in C++

- Allows **ET_Context** object to alter its behavior when its state changes

This method delegates to the ET_State object

```
void
ET_Context::make_tree(const std::string &expression) {
  state_->make_tree(*this, expression);
}

class Uninitialized_State : public State  {
public:
  virtual void make_tree(ET_Context &tc,
                        const std::string &expr)
  { throw Invalid_State("make_tree called in invalid state"); }
...
```

It's invalid to call make_tree() in this state

State GoF Object Behavioral

State example in C++

- Allows **ET_Context** object to alter its behavior when its state changes

This method delegates to the ET_State object

```
void
ET_Context::make_tree(const std::string &expression) {
  state_->make_tree(*this, expression);
}

class In_Order_Uninitialized_State : public Uninitialized_State {
public:
```

Calling make_tree() in this state initializes expression tree

```
  virtual void make_tree(ET_Context &et_context,
                         const std::string &expr) {
    ET_Interpreter interp;
    ET_Interpreter_Context interp_context;

    et_context.tree(interp.interpret (interp_context, expr));
    et_context.state(new In_Order_Initialized_State);
  }
...
```

Transition to the new state

State GoF Object Behavioral

Consequences

+ It localizes state-specific behavior & partitions behavior for different states
+ It makes state transitions explicit
+ State objects can be shared

– Can result in lots of subclasses that are hard to understand

State GoF Object Behavioral

Consequences

+ It localizes state-specific behavior & partitions behavior for different states
+ It makes state transitions explicit
+ State objects can be shared
– Can result in lots of subclasses that are hard to understand

Implementation

- Who defines state transitions?
- Consider using table-based alternatives
- Creating & destroying state objects

State GoF Object Behavioral

Consequences

+ It localizes state-specific behavior & partitions behavior for different states
+ It makes state transitions explicit
+ State objects can be shared
– Can result in lots of subclasses that are hard to understand

Implementation

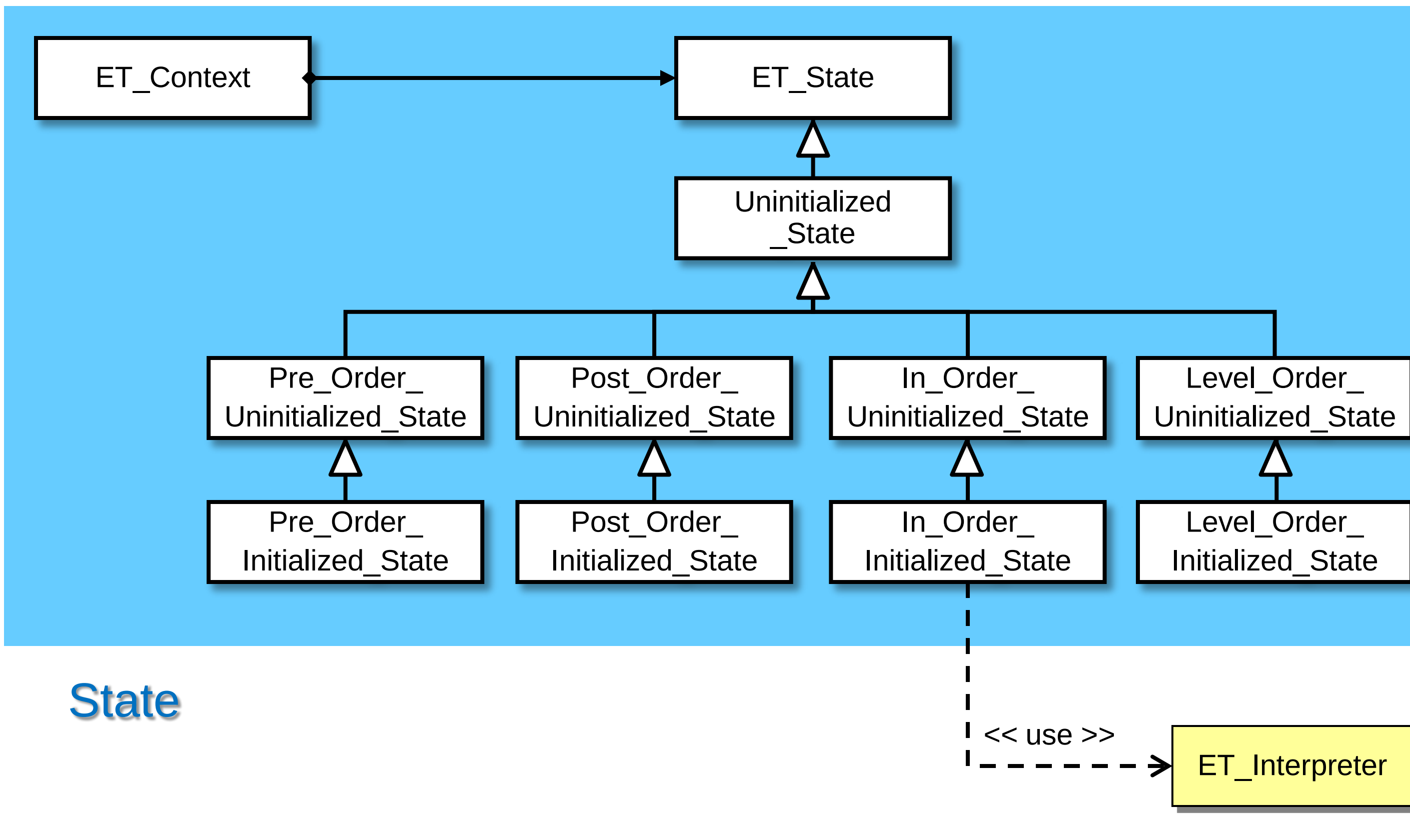
- Who defines state transitions?
- Consider using table-based alternatives
- Creating & destroying state objects

Known Uses

- The State pattern & its application to TCP connection protocols are characterized by Ralph Johnson & Johnny Zweig in their article "Delegation in C++," Journal of Object-Oriented Programming, 4(11):22-35, November 1991
- Unidraw & Hotdraw drawing tools

Summary of State Pattern

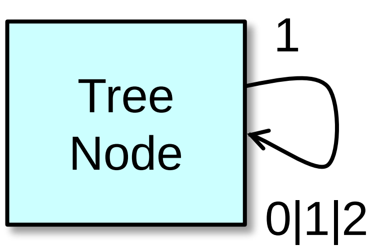
State ensures user commands are performed in the correct order

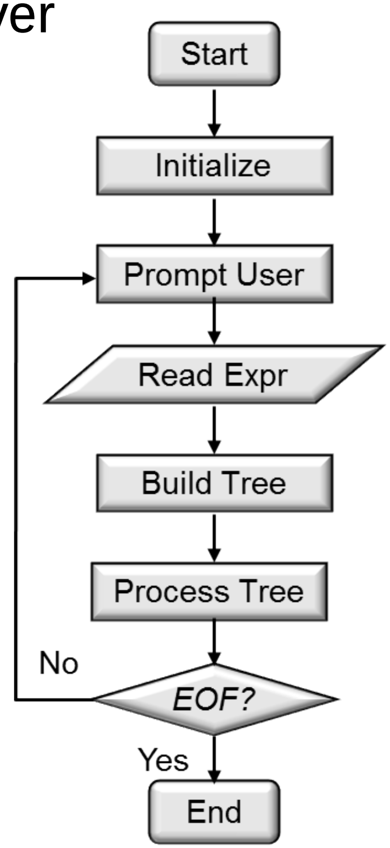


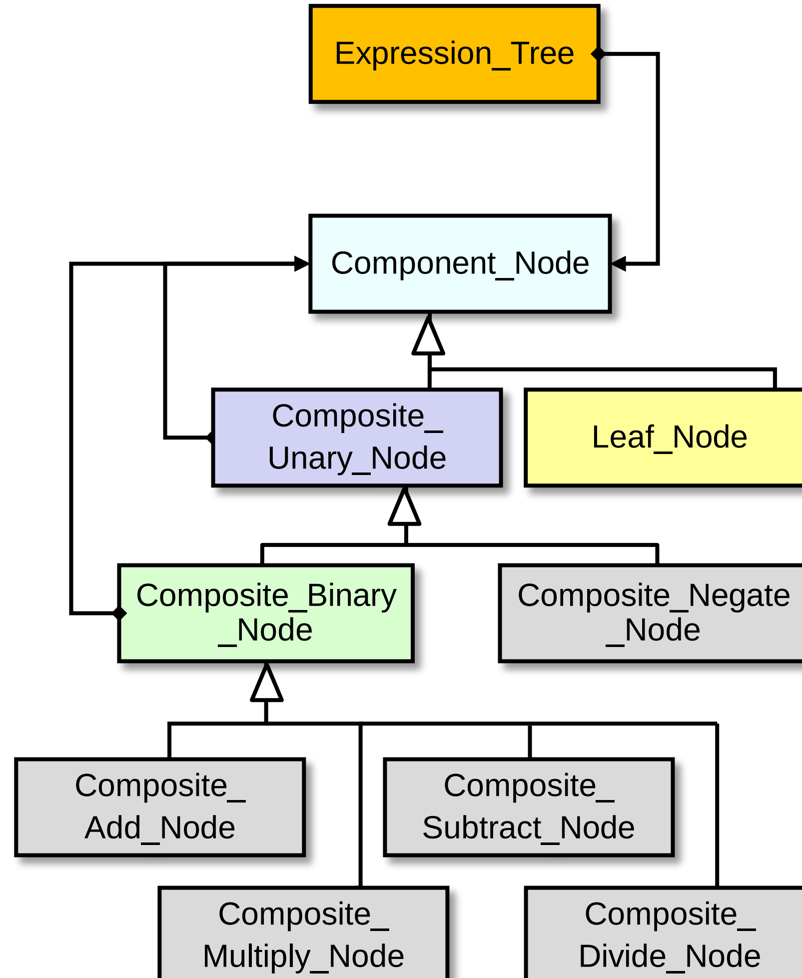
This pattern uses design of classes to explicitly order user commands correctly

Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvements over the original algorithmic decomposition

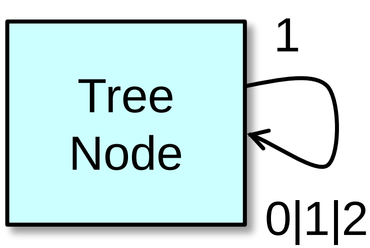


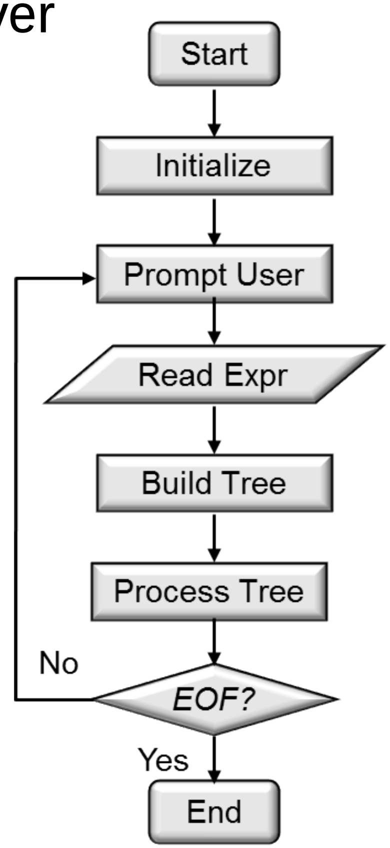


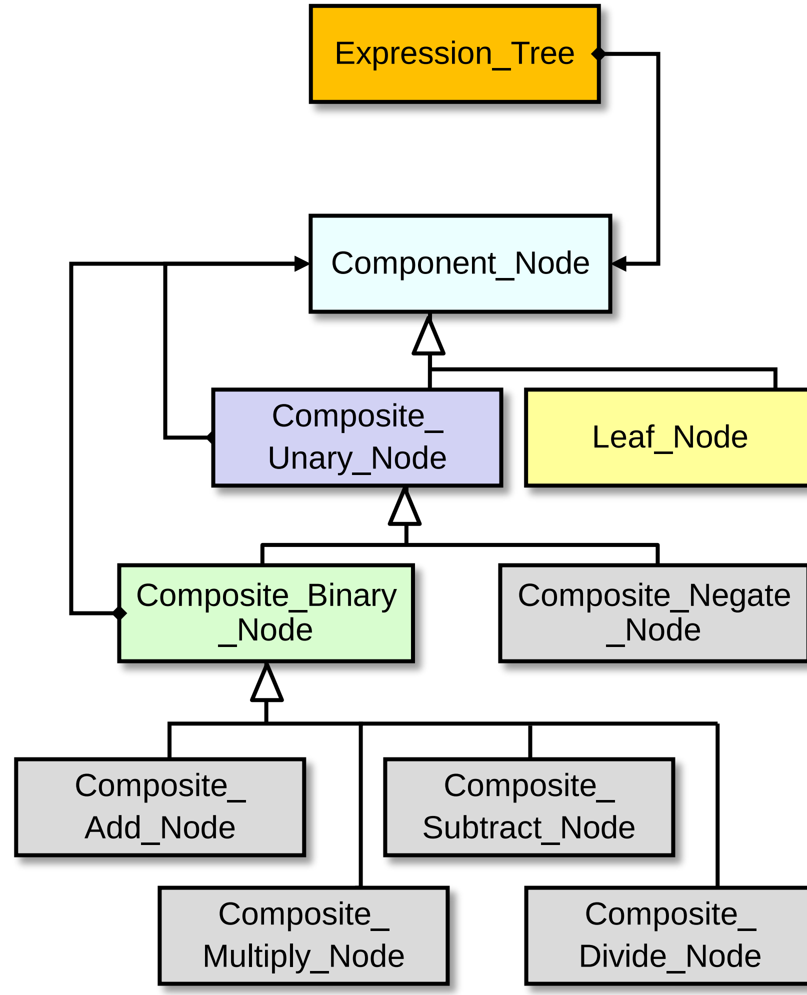


Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvements over the original algorithmic decomposition
    - Much more modular & extensible

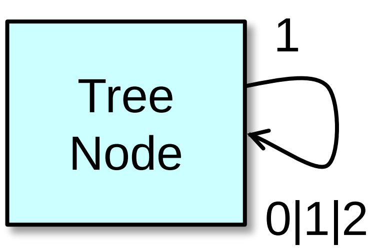


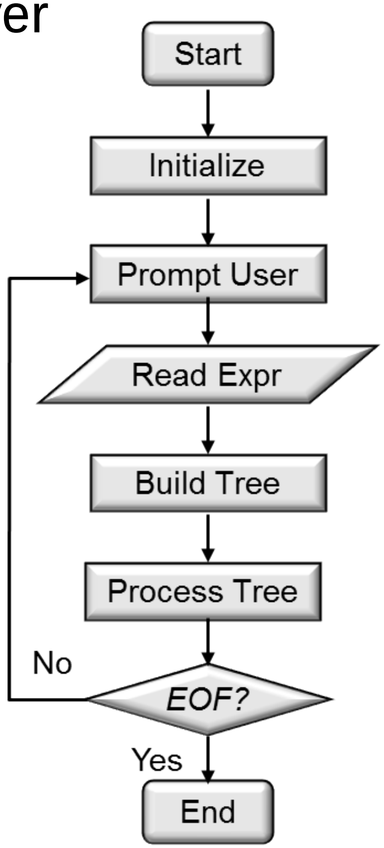


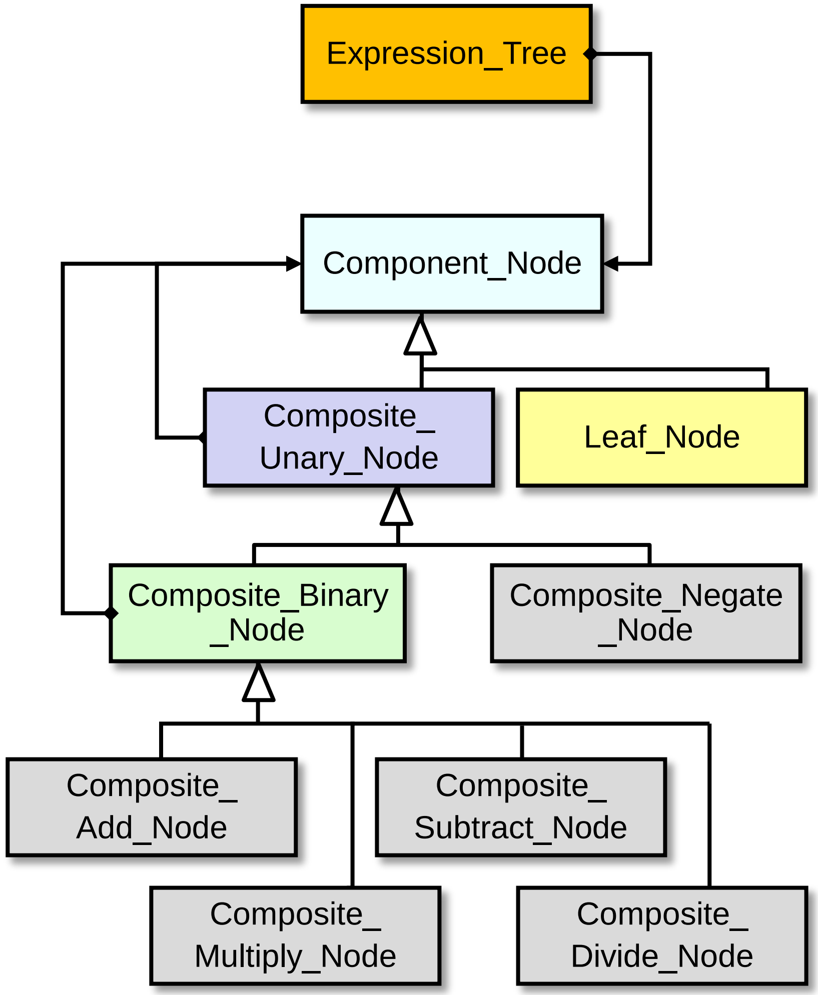


Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvements over the original algorithmic decomposition
    - Much more modular & extensible
  - Design matches the "domain" better

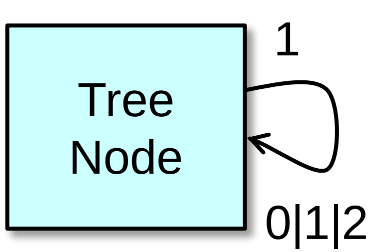


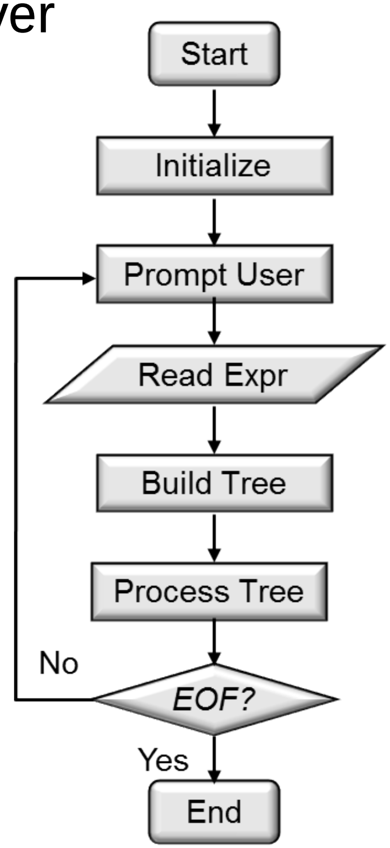


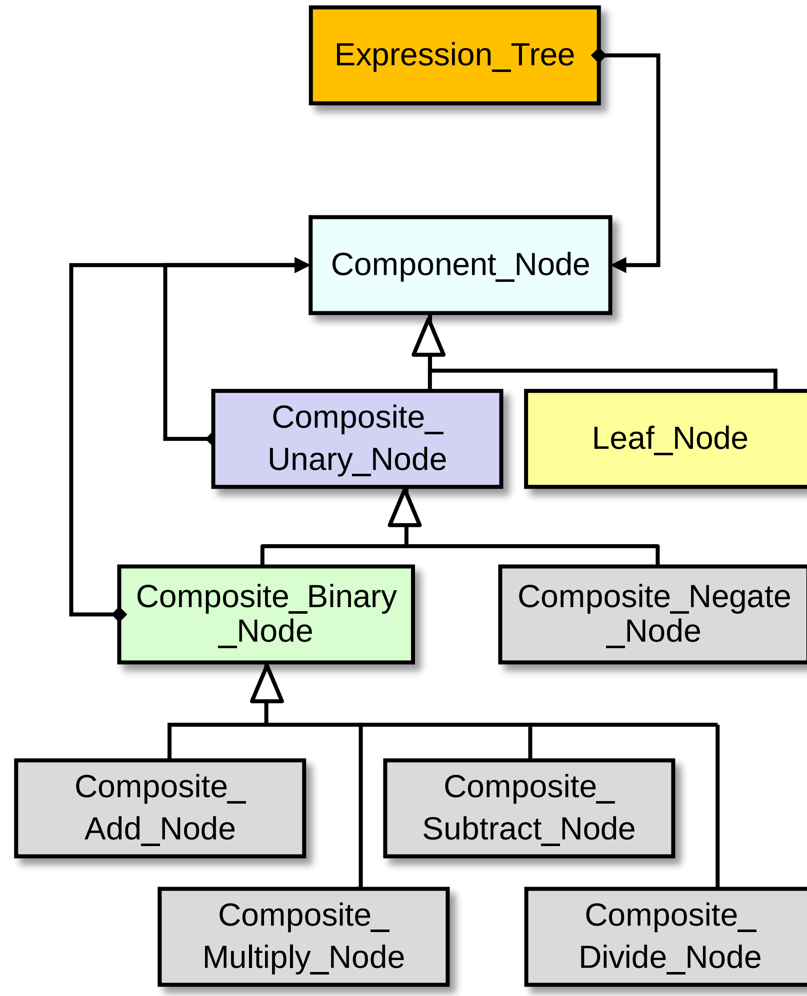


Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvements over the original algorithmic decomposition
    - Much more modular & extensible
    - Design matches the "domain" better
    - Less space overhead



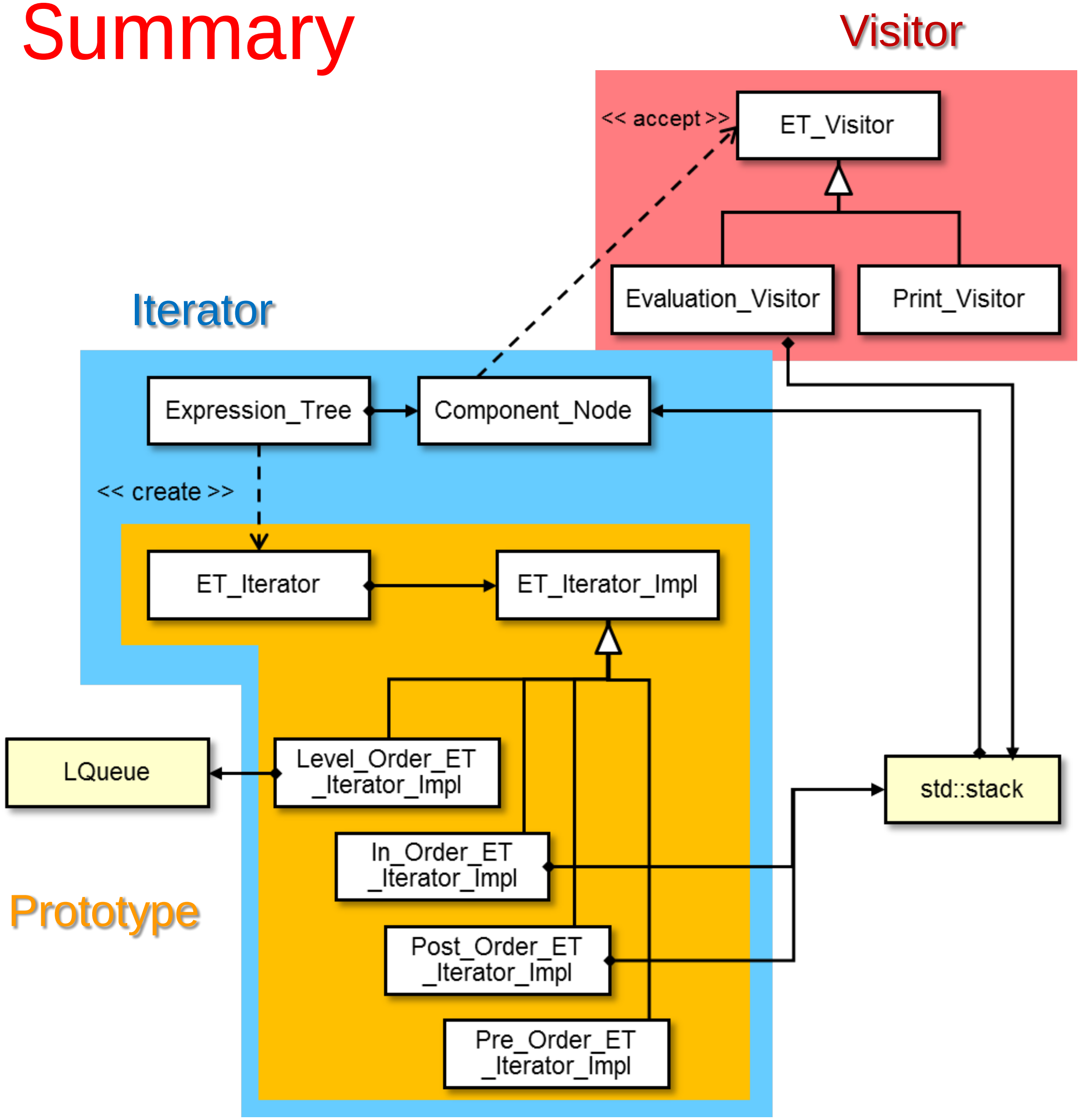




Koenig's *Ruminations on C++* book has another OO expression tree example

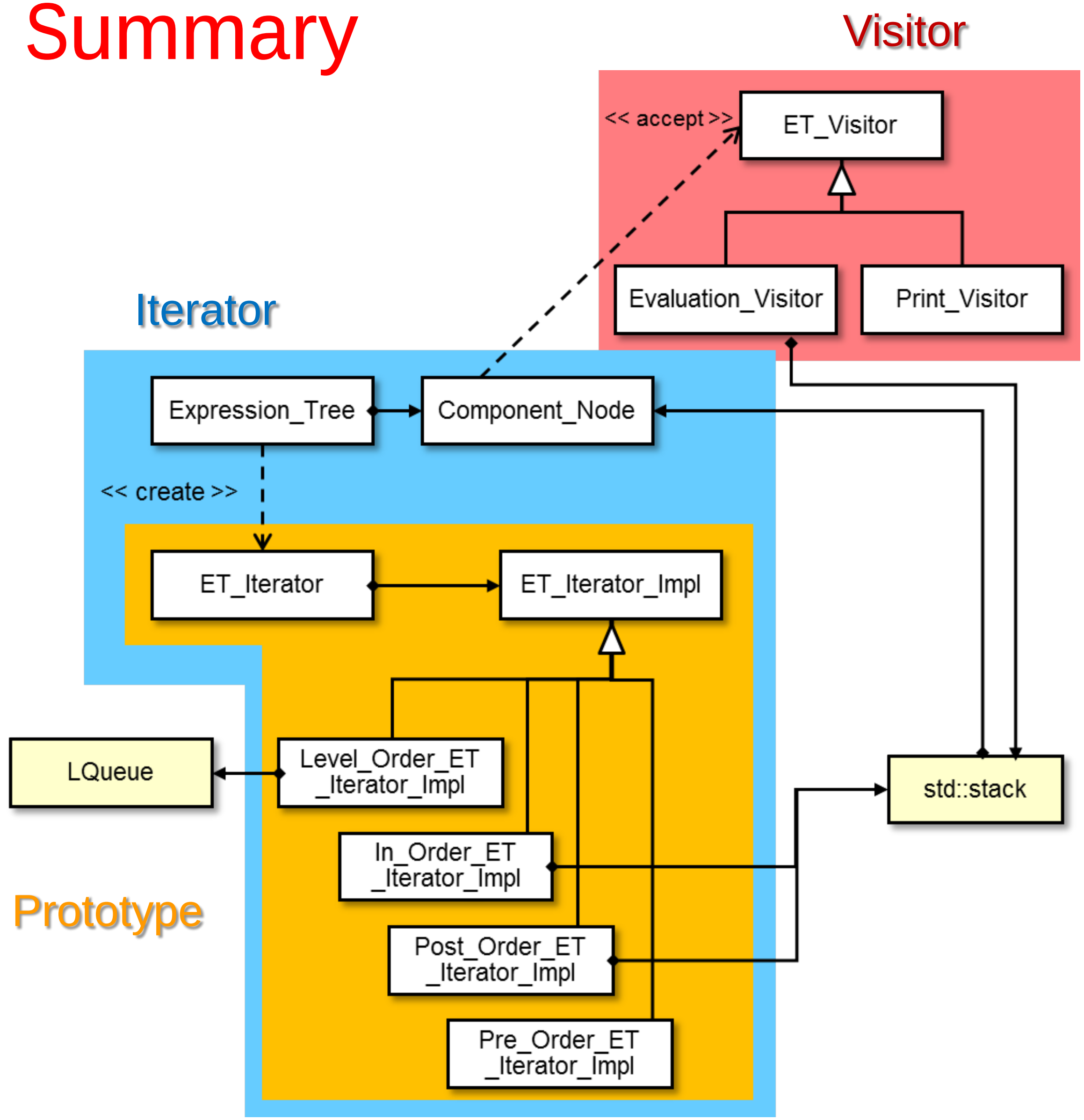
Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvement over the original algorithmic decomposition
  - Exhibits "high pattern density"



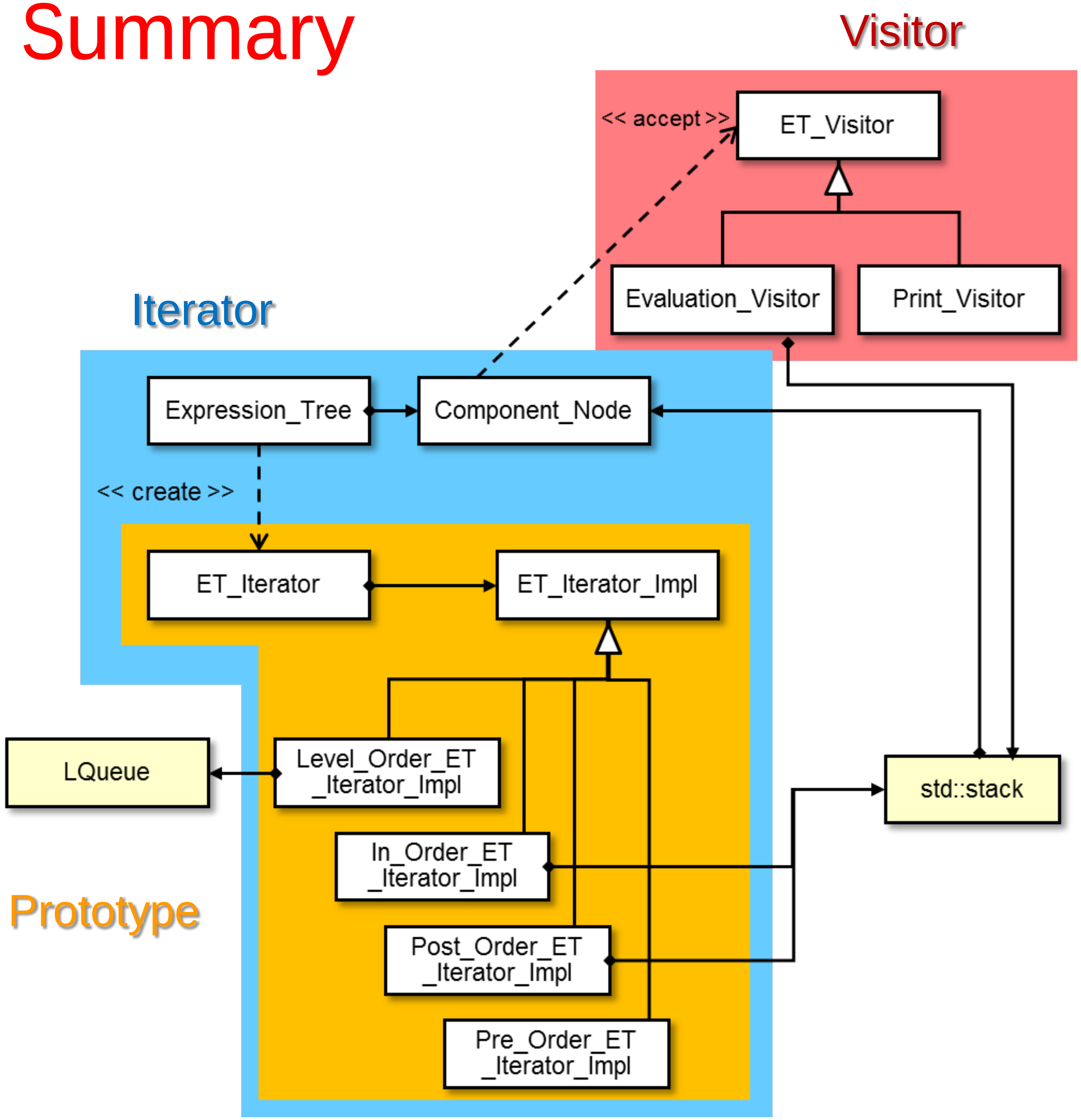
Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvement over the original algorithmic decomposition
  - Exhibits "high pattern density"
    - Nearly all classes & objects in design play a role in one or more patterns



Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvement over the original algorithmic decomposition
  - Exhibits "high pattern density"
    - Nearly all classes & objects in design play a role in one or more patterns
    - Patterns help clarify the relationships of myriad classes in the design



Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvement over the original algorithmic decomposition
  - Exhibits "high pattern density"
  - Same design can easily be realized in common OO programming languages

```
Expression_Tree expr_tree = ...;
Print_Visitor print_visitor;
```

C++11 range-based for loop

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

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

Java for-each loop

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

Summary

- Pattern-oriented expression tree processing app design has many benefits:
  - Major improvement over the original algorithmic decomposition
  - Exhibits “high pattern density”
  - Same design can easily be realized in common OO programming languages
    - C++ & Java solutions are nearly identical, modulo minor syntactical & semantic differences

```
Expression_Tree expr_tree = ...;
Print_Visitor print_visitor;
```

C++11 range-based for loop

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

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

Java for-each loop

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

See www.vincehuston.org/dp for many examples of patterns in C++ & Java