A Case Study of “Gang of Four” (GoF) Patterns : Part 7

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

```cpp
for (auto iter = expr_tree.begin(); iter != expr_tree.end(); ++iter)
    (*iter).accept(print_visitor);
```
Overview of Tree Traversal Patterns

**Purpose**: Traverse the expression tree & perform designated operations

Patterns decouple expression tree structure from operations performed on it.
Problem: Extensible Expression Tree Operations

Goals

• Create a framework for performing operations that affect nodes in a tree

Operation 1: Print all the values of the nodes in the tree

Operation 2: Evaluate the “yield” of the nodes in the tree
Problem: Extensible Expression Tree Operations

Goals

• Create a framework for performing operations that affect nodes in a tree

Constraints/forces

• Support multiple operations on the expression tree without tightly coupling operations with the tree structure
  • i.e., don’t have `print()` & `evaluate()` methods in the node classes
Solution (Part A): Encapsulate Traversal

**Iterator**
- Encapsulates a traversal algorithm without exposing representation details to callers
- e.g.,
  - “in-order iterator” = \(-5 \ast (3+4)\)
  - “pre-order iterator” = \(*-5+34\)
  - “post-order iterator” = \(5-34+*\)
  - “level-order iterator” = \(*-+534\)

```cpp
for (auto iter = expr_tree.begin();
     iter != expr_tree.end();
     ++iter)
    (*iter).accept(print_visitor);
```
Solution (Part A): Encapsulate Traversal

**Iterator**

- Encapsulates a traversal algorithm without exposing representation details to callers
- e.g.,
  - “in-order iterator” = \(-5 \times (3+4)\)
  - “pre-order iterator” = \(*-5+34\)
  - “post-order iterator” = \(5-34+*\)
  - “level-order iterator” = \(*+-+534\)

**Iterator Structure**

```
Bridge pattern encapsulates variability & simplifies memory management
```
ET_Iterator Class Interface

- Interface for iterator that traverses all nodes in an expression tree instance

This interface plays the role of the abstraction in the *Bridge* pattern

**Interface**

```
ET_Iterator (ET_Iterator_Impl *)
ET_Iterator (const ET_Iterator &)
Expression_Tree operator *()
const Expression_Tree operator *() const
   ET_Iterator & operator++()
      ET_Iterator operator++(int)
   bool operator==(const ET_Iterator &rhs)
   bool operator!=(const ET_Iterator &rhs)
```

Overloaded C++ operators conform to what’s expected from an STL iterator...

- **Commonality**: Provides a common interface for expression tree iterators that conforms to the standard STL iterator interface
- **Variability**: Can be configured with specific expression tree iterator algorithms via the *Abstract Factory* pattern
**ET_Iterator_Impl Class Interface**

- **Commonality**: Provides a common interface for implementing expression tree iterators that conforms to the standard STL iterator interface

- **Variability**: Can be subclassed to define various algorithms for accessing nodes in the expression trees in a particular traversal order

```cpp
ET_Iterator_Impl(const Expression_Tree &)
virtual ~ET_Iterator_Impl() = 0
virtual Expression_Tree operator *() = 0
virtual void operator++() = 0
virtual bool operator== (const ET_Iterator_Impl &)=0
virtual bool operator!= (const ET_Iterator_Impl &)=0
virtual ET_Iterator_Impl * clone() = 0
```

- *clone() is used by Prototype pattern*

---

**Interface**

- **Commonality**: Provides a common interface for implementing expression tree iterators that conforms to the standard STL iterator interface

- **Variability**: Can be subclassed to define various algorithms for accessing nodes in the expression trees in a particular traversal order

---

**This class doesn’t need to mimic the Bridge interface**

- **Commonality**: Provides a common interface for implementing expression tree iterators that conforms to the standard STL iterator interface

- **Variability**: Can be subclassed to define various algorithms for accessing nodes in the expression trees in a particular traversal order
Intent

- Access elements of an aggregate without exposing its representation

Applicability

- Require multiple traversal algorithms over an aggregate
- Require a uniform traversal interface over different aggregates
- When aggregate classes & traversal algorithm must vary independently

Structure

- **Aggregate**
  - createliterator()
  - e.g., Expression_Tree

- **ConcreteAggregate**
  - createliterator()
  - return new Concretelterator(this)

- **Client**

- **Iterator**
  - first()
  - next()
  - isDone()
  - currentItem()
  - e.g., ETIterator_Impl
  - e.g., Pre_Order_ETIterator_Impl, In_Order_ETIterator_Impl, etc.

- **Concretelterator**
Comparing STL iterators with GoF iterators

- STL iterators have “value-semantics”, e.g.:

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

- In contrast, GoF iterators have “pointer semantics”, e.g.:

  ```cpp
  Iterator *iter;
  for (iter = exprTree.createIterator();
      iter->done() == false;
      iter->advance())
    iter->currentElement()->accept(printVisitor);
  delete iter;
  ```

The **Bridge** pattern enables STL iterator use in expression tree processing app.
Comparing Java iterators with GoF iterators

• Java iterators are closer to GoF iterators than STL iterators are, e.g.:

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

• Here's the equivalent Java code for GoF-style iterators, e.g.:

```java
for (ETIterator iter = tree.createIterator();
     !iter.done();
    iter.advance())
    (iter.currentElement()).accept(printVisitor);
```

Often, the simplest way to use an iterator in C++/Java is to use a for loop
Comparing Java iterators with STL iterators

• Often, the simplest way to use an iterator in C++/Java is to use a for loop, e.g.:

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

  ```java
  for (ComponentNode node : exprTree)
    node.accept(printVisitor);
  ```
Iterator example in C++

- An STL `std::stack` can be used to traverse an expression tree in pre-order.

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

        virtual Expression_Tree operator*() { return stack_.top(); }

        virtual void operator++() {
            if (!stack_.is_empty()) {
                Expression_Tree current = stack_.top(); stack_.pop();
                if (!current.right().is_null())
                    stack_.push(current.right());
                if (!current.left().is_null())
                    stack_.push(current.left());
            }
        }

    ...

    The use of a stack simulates recursion, one item at a time.
```
<table>
<thead>
<tr>
<th>Iterator</th>
<th>GoF Object Behavioral</th>
</tr>
</thead>
</table>

**Consequences**

+ **Flexibility**: Aggregate & traversal are independent
+ **Multiplicity**: Multiple iterators & multiple traversal algorithms
  - **Overhead**: Additional communication between iterator & aggregate
    - Particularly problematic for iterators in concurrent or distributed systems
**Iterator**

**GoF Object Behavioral**

**Consequences**

+ *Flexibility*: Aggregate & traversal are independent
+ *Multiplicity*: Multiple iterators & multiple traversal algorithms
  
  - *Overhead*: Additional communication between iterator & aggregate
  - Particularly problematic for iterators in concurrent or distributed systems

**Implementation**

- Internal vs. external iterators
- Violating the object structure’s encapsulation
- Robust iterators
- Synchronization overhead in multi-threaded programs
- Batching in distributed & concurrent programs
Iterator

**Consequences**

+ **Flexibility**: Aggregate & traversal are independent
+ **Multiplicity**: Multiple iterators & multiple traversal algorithms
  - **Overhead**: Additional communication between iterator & aggregate
  - Particularly problematic for iterators in concurrent or distributed systems

**Implementation**

- Internal vs. external iterators
- Violating the object structure’s encapsulation
- Robust iterators
- Synchronization overhead in multi-threaded programs
- Batching in distributed & concurrent programs

---

GoF Object Behavioral

**Known Uses**

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator
- C++11 range-based for loops & Java for-each loops
- C buffered I/O
Solution (Part B): Decouple Operations from Expression Tree Structure

**Visitor**

- Defines action(s) at each step of traversal & avoids hard-coding action(s) into nodes
Solution (Part B): Decouple Operations from Expression Tree Structure

Visitor

- Defines action(s) at each step of traversal & avoids hard-coding action(s) into nodes
- Iterator calls `accept(ET_Visitor&)` method on each node in expression tree

```cpp
for (auto iter = expr_tree.begin(); iter != expr_tree.end(); ++iter)
  (*iter).accept(print_visitor);
```
Solution (Part B): Decouple Operations from Expression Tree Structure

**Visitor**

- Defines action(s) at each step of traversal & avoids hard-coding action(s) into nodes
- Iterator calls accept(ET_Visitor&) method on each node in expression tree
  
  ```
  for (auto iter = expr_tree.begin();
       iter != expr_tree.end();
       ++iter)
    (*iter).accept(print_visitor);
  ```

- accept() calls back on visitor, e.g.:
  ```
  void Leaf_Node::accept(ET_Visitor &v) {
    v.visit(*this);
  }
  ```

Note “static polymorphism” based on method overloading by type