Object-Oriented Design Case Study with C++

Douglas C. Schmidt

Professor
Department of EECS
Vanderbilt University
d.schmidt@vanderbilt.edu
(615) 343-8197

Case Study: Expression Tree Evaluator

- The following inheritance and dynamic binding example constructs expression trees
  - Expression trees consist of nodes containing operators and operands
    * Operators have different precedence levels, different associativities, and different arities, e.g.,
      - Multiplication takes precedence over addition
      - The multiplication operator has two arguments, whereas unary minus operator has only one
    * Operands are integers, doubles, variables, etc.
      - We’ll just handle integers in this example . . .

Expression Tree Diagram

Expression Tree Behavior

- Expression trees
  - Trees may be “evaluated” via different traversals
    * e.g., in-order, post-order, pre-order, level-order
  - The evaluation step may perform various operations, e.g.,
    * Traverse and print the expression tree
    * Return the “value” of the expression tree
    * Generate code
    * Perform semantic analysis
A typical algorithmic implementation uses a switch statement and a recursive function to build and evaluate a tree, e.g.,

```c
void print_tree (Tree_Node *root) {
    switch (root->tag_) {
        case NUM: printf ("%d", root->num_); break;
        case UNARY:
            printf ("(%s", root->op_[0]);
            print_tree (root->unary_);
            printf (")"); break;
        case BINARY:
            printf ("(");
            print_tree (root->binary_.l_);
            printf ("%s", root->op_[0]);
            print_tree (root->binary_.r_);
            printf (")"); break;
        default:
            printf (error, unknown type\n);
    }
}
```

Problems or limitations with the typical algorithmic approach include:

- Little or no use of encapsulation
- Incomplete modeling of the application domain, which results in:
  - Tight coupling between nodes and edges in union representation
  - Complexity being in algorithms rather than the data structures
- Data structures are "passive", and functions do most processing explicitly

Here's the memory layout of a struct `Tree_Node` object:

```c
typedef struct Tree_Node Tree_Node;
struct Tree_Node {
    enum { NUM, UNARY, BINARY } tag_;  
    short use_; /* reference count */
    union {
        char op_[2];
        int num_;  
    } o;
    #define num_ o.num_
    #define op_ o.op_
    union {
        Tree_Node *unary_;  
        struct { Tree_Node *l_, *r_; } binary_;  
    } c;
    #define unary_ c.unary_
    #define binary_ c.binary_
};
```
More Limitations with Algorithmic Approach

- The program organization makes it difficult to extend, e.g.,
  - Any small changes will ripple through the entire design and implementation
    * e.g., see the “ternary” extension below
  - Easy to make mistakes switching on type tags . . .
- Solution wastes space by making worst-case assumptions wrt structs and unions
  - This is not essential, but typically occurs
  - Note that this problem becomes worse the bigger the size of the largest item becomes!

OO Alternative

- Contrast previous algorithmic approach with an object-oriented decomposition for the same problem:
  - Start with OO modeling of the “expression tree” application domain, e.g., go back to original picture
  - Discover several classes involved:
    * class Node: base class that describes expression tree vertices:
      - class Int_Node: used for implicitly converting int to Tree node
      - class Unary_Node: handles unary operators, e.g., -10, +10, !a
      - class Binary_Node: handles binary operators, e.g., a + b, 10 - 30
    * class Tree: “glue” code that describes expression-tree edges, i.e., relations between Nodes
  - Note, these classes model entities in the application domain
    * i.e., nodes and edges (vertices and arcs)

Expression Tree Diagram

Relationships Between Tree and Node Classes
Design Patterns in the Expression Tree Program

- **Factory**
  - Centralize the assembly of resources necessary to create an object
    * e.g., decouple `Node` subclass initialization from use

- **Bridge**
  - Decouple an abstraction from its implementation so that the two can vary independently
    * e.g., printing contents of a subtree and managing memory

- **Adapter**
  - Convert the interface of a class into another interface clients expect
    * e.g., make `Tree` conform C++ iostreams

---

### C++ Node Interface

class Tree; // Forward declaration

// Describes the Tree vertices
class Node {
  friend class Tree;
  protected: // Only visible to derived classes
    Node (): use_ (1) {} // pure */ virtual void print (ostream & const) const = 0;
    // Important to make destructor virtual!
    virtual ~Node ();
private:
  int use_; // Reference counter.
};

---

### C++ Tree Interface

```cpp
#include "Node.h"

// Bridge class that describes the Tree edges and acts as a Factory.
class Tree {

public:
  // Factory operations
  Tree (int);
  Tree (const string &, Tree &);
  Tree (const string &, Tree &, Tree &);
  Tree (const Tree & t);
  void operator= (const Tree & t);
  ~Tree ();
  void print (ostream & const);

private:
  Node *node_; // pointer to a rooted subtree
};
```

---

### C++ Int_Node Interface

```cpp
#include "Node.h"

class Int_Node : public Node {

public:
  Int_Node (int k);
  virtual void print (ostream & stream) const;

private:
  int num_; // operand value.
};
```
C++ Unary_Node Interface

```cpp
#include "Node.h"

class Unary_Node : public Node {
public:
    Unary_Node (const string &op, const Tree &t);
    virtual void print (ostream &stream) const;
private:
    string operation_;   // The operation node
    Tree operand_;       // The operand node
};
```

C++ Binary_Node Interface

```cpp
#include "Node.h"

class Binary_Node : public Node {
public:
    Binary_Node (const string &op,
                 const Tree &t1,
                 const Tree &t2);
    virtual void print (ostream &s) const;
private:
    const string operation_;  // The operation node
    Tree left_;               // The left operand node
    Tree right_;              // The right operand node
};
```

C++ Int_Node Implementations

```cpp
#include "Int_Node.h"

Int_Node::Int_Node (int k): num_ (k) { }

void Int_Node::print (ostream &stream) const {
    stream << this->num_;  // Print the integer
}
```
### C++ Unary_Node Implementations

```cpp
#include "Unary_Node.h"

Unary_Node::Unary_Node (const string &op, const Tree &t1)
 : operation_ (op), operand_ (t1) { }

void Unary_Node::print (ostream &stream) const {
    stream << "(" << this->operation_ <<
    << this->operand_ // recursive call!
    << ")" ;
}
```

### C++ Binary_Node Implementation

```cpp
#include "Binary_Node.h"

Binary_Node::Binary_Node (const string &op,
 const Tree &t1,
 const Tree &t2):
 operation_ (op), left_ (t1), right_ (t2) {}

void Binary_Node::print (ostream &stream) const {
    stream << "(" << this->left_ // recursive call
    << " " << this->operation_  
    << " " << this->right_ // recursive call
    << ")" ;
}
```

---

### Initializing the Node Subclasses

- **Problem**
  - How to ensure the Node subclasses are initialized properly

- **Forces**
  - There are different types of Node subclasses
    - *e.g.,* take different number and type of arguments
  - We want to centralize initialization in one place because it is likely to change . . .

- **Solution**
  - Use a Factory pattern to initialize the Node subclasses

---

### The Factory Pattern

- **Intent**
  - *Centralize the assembly of resources necessary to create an object*
    - Decouple object creation from object use by localizing creation knowledge

- This pattern resolves the following forces:
  - Decouple initialization of the Node subclasses from their subsequent use
  - Makes it easier to change or add new Node subclasses later on
    - *e.g.,* Ternary nodes . . .

- A generalization of the GoF Factory Method pattern
**Structure of the Factory Pattern**

```
Factory
make_product() -> Product
```

**Using the Factory Pattern**

- The Factory pattern is used by the Tree class to initialize Node subclasses:

```cpp
Tree::Tree (int num) : node_ (new Int_Node (num)) {}
```

```cpp
Tree::Tree (const string &op, const Tree &t) : node_ (new Unary_Node (op, t)) {}
```

```cpp
Tree::Tree (const string &op,
            const Tree &t1,
            const Tree &t2):
    : node_ (new Binary_Node (op, t1, t2)) {}
```

**Printing Subtrees**

- **Problem**
  - How do we print subtrees without revealing their types?

- **Forces**
  - The Node subclass should be hidden within the Tree instances
  - We don’t want to become dependent on the use of Nodes, inheritance, and dynamic binding, *etc.*
  - We don’t want to expose dynamic memory management details to application developers

- **Solution**
  - Use the Bridge pattern to shield the use of inheritance and dynamic binding

**The Bridge Pattern**

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently

- This pattern resolves the following forces that arise when building extensible software with C++

  1. **How to provide a stable, uniform interface that is both closed and open, i.e.,**
     - interface is closed to prevent direct code changes
     - Implementation is open to allow extensibility
  2. **How to manage dynamic memory more transparently and robustly**
  3. **How to simplify the implementation of operator<<**
Illustrating the Bridge Pattern in C++

- The Bridge pattern is used for printing expression trees:

```cpp
void Tree::print (ostream &os) const {
    this->node_->print (os);
}
```

- Note how this pattern decouples the `Tree` interface for printing from the `Node` subclass implementation
  - *i.e.*, the `Tree` interface is *fixed*, whereas the `Node` implementation varies
  - However, clients need not be concerned about the variation . . .

Integrating with C++ I/O Streams

- **Problem**
  - Our `Tree` interface uses a `print` method, but most C++ programmers expect to use I/O Streams

- **Forces**
  - Want to integrate our existing C++ `Tree` class into the I/O Stream paradigm without modifying our class or C++ I/O

- **Solution**
  - Use the *Adapter* pattern to integrate `Tree` with I/O Streams
The Adapter Pattern

- **Intent**
  - Convert the interface of a class into another interface client expects
    * Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
- This pattern resolves the following force:
  1. How to transparently integrate the Tree with the C++ iostream operators

The Adapter pattern is used to integrate with C++ I/O Streams

```cpp
ostream &operator<<(ostream &s, const Tree &tree) {
    tree.print (s);
    // This triggers Node * virtual call via
    // tree.node_->print (s), which is
    // implemented as the following:
    // (*tree.node_->vptr[1]) (tree.node_, s);
    return s;
}
```

Note how the C++ code shown above uses I/O streams to “adapt” the Tree interface...
**C++ Tree Implementation**

- Reference counting via the “counted body” idiom

```cpp
tree::tree (const tree &t)
    node_ (t.node_) {
    // Sharing, ref-counting.
    ++this->node_->use_;}

void tree::operator= (const tree &t) {
    // order important here!
    ++t.node_->use_;  
    --this->node_->use_;  
    if (this->node_->use_ == 0)  
        delete this->node_;  
    this->node_ = t.node_;}
```

**C++ Tree Implementation (cont’d)**

```cpp
tree::~tree () {
    // Ref-counting, garbage collection
    --this->node_->use_;  
    if (this->node_->use_ <= 0)  
        delete this->node_;}
```

**C++ Main Program**

```cpp
#include <iostream.h>
#include "Tree.h"

int main (int, char **[]) {
    const tree t1 = tree ("*", tree ("-", 5),
                    tree ("+", 3, 4));
    cout << t1 << endl; // prints ((-5) * (3 + 4))
const tree t2 = tree ("*", t1, t1);
    // prints (((-5) * (3 + 4)) * ((-5) * (3 + 4))).
    cout << t2 << endl;
    return 0;
    // Destructors of t1 and t2 recursively  
} // delete entire tree when leaving scope.
```

**Expression Tree Diagram 1**

- Expression tree for t1 = ((-5) * (3 + 4))
Adding Ternary Nodes

- Extending the existing program to support ternary nodes is straightforward
  - i.e., just derive new class Ternary_Node to handle ternary operators, e.g., a == b ? c : d, etc.

```cpp
#include "Node.h"
class Ternary_Node : public Node {
  public:
    Ternary_Node (const string &op, const Tree &a, const Tree &b, const Tree &c);
    virtual void print (ostream &) const;
  private:
    const string operation_; 
    Tree left_, middle_, right_; 
};
```

C++ Ternary Node Implementation

```cpp
#include "Ternary_Node.h"
Ternary_Node::Ternary_Node (const string &op, const Tree &a, const Tree &b, const Tree &c)
  : operation_ (op), left_ (a), middle_ (b), right_ (c) {};

void Ternary_Node::print (ostream &stream) const {
  stream << this->operation_ << "("
  << this->left_ // recursive call
  << "," << this->middle_ // recursive call
  << "," << this->right_ // recursive call
  << ")";
}
```

C++ Ternary Node Implementation (cont’d)

```cpp
#include "Ternary_Node.h"

class Tree {
  // add 1 class constructor
  public:
    Tree (const string &op, const Tree &l, const Tree &m, const Tree &r)
      : node_ (new Ternary_Node (op, l, m, r)) {} 
  // Same as before . . .
```
Differences from Algorithmic Implementation

- On the other hand, modifying the original algorithmic approach requires changing (1) the original data structures, e.g.,

```c
struct Tree_Node {
    enum {
        NUM, UNARY, BINARY, TERNARY
    } tag_; // same as before
    union {
        // same as before. But, add this:
        struct {
            Tree_Node *l_, *m_, *r_;
        } ternary_;
    } c;
#define ternary_ c.ternary_
};
```

- and (2) many parts of the code, e.g.,

```c
void print_tree (Tree_Node *root) {
    // same as before
    case TERNARY: // must be TERNARY.
        printf ("(");
        print_tree (root->ternary_.l_);
        printf ("%c", root->op_[0]);
        print_tree (root->ternary_.m_);
        printf ("%c", root->op_[1]);
        print_tree (root->ternary_.r_);
        printf (")"); break;
    // same as before
}
```

Summary of Expression Tree Example

- OO version represents a more complete modeling of the application domain
  - e.g., splits data structures into modules that correspond to "objects" and relations in expression trees
- Use of C++ language features simplifies the design and facilitates extensibility
  - e.g., implementation follows directly from design
- Use of patterns helps to motivate, justify, and generalize design choices

Potential Problems with OO Design

- Solution is very “data structure rich”
  - e.g., requires configuration management to handle many headers and .cc files!
- May be somewhat less efficient than original algorithmic approach
  - e.g., due to virtual function overhead
- In general, however, virtual functions may be no less inefficient than large switch statements or if/else chains . . .
- As a rule, be careful of micro vs. macro optimizations
  - i.e., always profile your code!