Design Patterns Vol. 2

Commented C++ Code Examples

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

The Iterator

Let's code in *C++* the *Design Patterns Vol. 1* example for the *Iterator* pattern

The Iterator

We have a container: a binary tree

```
class Node {
  int mValue;
  Node* mChildren[2];

public:
  Node(int inValue);

  int value() const { return mValue; }

  Node*& child(int i) { return mChildren[i]; }

  const Node* child(int i) const { return mChildren[i]; }
};
```

Tree traversals

There are various way to traverse a binary tree

- Breadth-first order
- Depth-first order
- More ...

Iterator traversal

Let's define an interface for the tree traversal

```
class NodeIterator {
public:
  virtual ~NodeIterator();

  virtual void init(Node* inRoot) = 0;

  virtual bool hasNext() const = 0;

  virtual Node* next() = 0;
};
```

Iterator traversal

We can now define the breadth-first iterator

```
class BreadthFirstNodeIterator : public NodeIterator {
  std::deque<Node*> mQueue;

public:
  virtual ~BreadthFirstNodeIterator() { }

  virtual void init(Node* inRoot) {
    mQueue.clear();
    mQueue.push_back(inRoot);
  }

  virtual bool hasNext() const {
    return !mQueue.empty();
  }

  virtual Node* next() { ... }
};
```

Iterator traversal

We can now define the breadth-first iterator

```
class BreadthFirstNodeIterator : public NodeIterator {
  std::deque<Node*> mQueue;

public:
  virtual ~BreadthFirstNodeIterator() { }

  virtual void init(Node* inRoot) { ... }

  virtual bool hasNext() const { ... }

  virtual Node* next() {
    if (mQueue.empty())
      return 0;

    Node* lFront = mQueue.front();
    mQueue.pop_front();

    for(int i = 0; i < 2; ++i) {
      Node* lChild = lFront->child(i);
      if (lChild)
        mQueue.push_back(lChild);
    }

    return lFront;
  }
};
```

Iterator traversal

We can now define the depth-first iterator

```
class DepthFirstNodeIterator : public NodeIterator {
  std::vector<Node*> mStack;

public:
  virtual ~DepthFirstNodeIterator() { }

  virtual void init(Node* inRoot) {
    mStack.clear();
    mStack.push_back(inRoot);
  }

  virtual bool hasNext() const {
    return !mStack.empty();
  }

  virtual Node* next() { ... }
};
```

Iterator traversal

We can now define the depth-first iterator

```
class DepthFirstNodeIterator : public NodeIterator {
  std::vector<Node*> mStack;

public:
  virtual ~DepthFirstNodeIterator() { }

  virtual void init(Node* inRoot) { ... }

  virtual bool hasNext() const { ... }

  virtual Node* next() {
    if (mStack.empty())
      return 0;

    Node* lTop = mStack.back();
    mStack.pop_back();

    for(int i = 0; i < 2; ++i) {
      Node* lChild = lTop->child(1 - i);
      if (lChild)
        mStack.push_back(lChild);
    }

      return lTop;
  }
};
```

Usage

How does it look like when we use one of our iterators ?

```
Node* IRoot = ...

inIterator->init(IRoot);
while(inIterator->hasNext()) {
  Node* INode = inIterator->next();
  cout << INode->value() << ' ';
}
```

Conclusion

We have successfully

- Encapsulated the concept of *traversal* of a tree
- The algorithm and the data structure are separated

Table of Contents

The Visitor

Let's code a *C++* example for the *Visitor* pattern

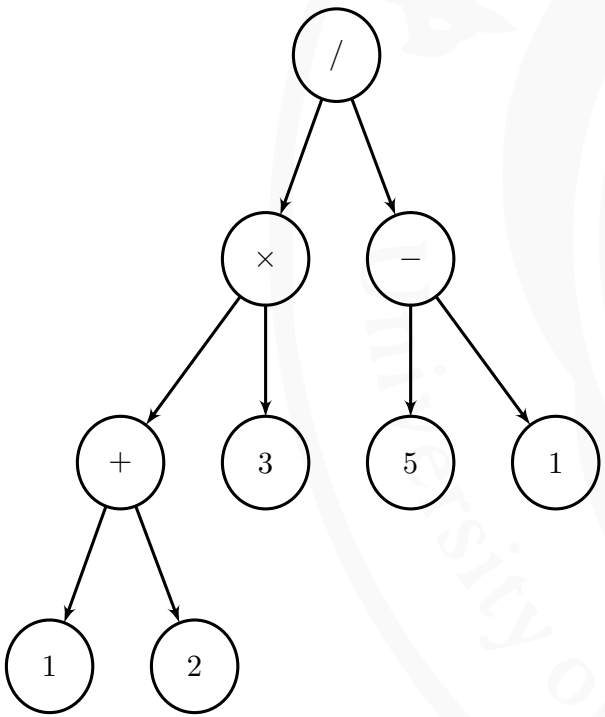
Arithmetic expressions

Let's play with arithmetic expressions

$$((2+1)\times 3)/(5-1)$$

Arithmetic expressions

Arithmetic expressions can be represented as a tree



$$((2+1)\times 3)/(5-1)$$

Arithmetic expressions

With arithmetic expressions, we want to

1. print it
2. compute its result
3. more, and we don't know exactly what

We are going to need nodes, lot of nodes

```
class Node {
public:
  virtual ~Node();

  virtual void accept(NodeVisitor& inVisitor) const = 0;

  virtual int nbChildren() const = 0;

  virtual const Node* child(int index) const = 0;
};
```

Nodes which shall not have any children, ever

```
class LeafNode : public Node {
public:
  virtual ~LeafNode() { }

  virtual int nbChildren() const { return 0; }

  virtual const Node* child(int index) const { return 0; }
};
```

2 types of nodes are *LeafNodes*

```
class IntNode : public LeafNode {
  long mValue;

public:
  IntNode(long inValue) : mValue(inValue) { }

  virtual ~IntNode() { }

  long value() const { return mValue; }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

2 types of nodes are *LeafNodes*

```
class FloatNode : public LeafNode {
  double mValue;

public:
  FloatNode(double inValue) : mValue(inValue) { }

  virtual ~FloatNode() { }

  double value() const { return mValue; }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

Nodes which shall have 2 children

```
class BinaryNode : public Node {
  Node* mChildren[2];

public:
  virtual ~BinaryNode() { }

  virtual int nbChildren() const { return 2; }

  virtual const Node* child(int index) const { return mChildren[index]; }

protected:
  BinaryNode(Node* inLeft, Node* inRight) {
    mChildren[0] = inLeft;
    mChildren[1] = inRight;
  }
};
```

4 types of nodes are *BinaryNodes*

```
class AddNode : public BinaryNode {
public:
  AddNode(Node* inLeft, Node* inRight) : BinaryNode(inLeft, inRight) { }

  virtual ~AddNode() { }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

4 types of nodes are *BinaryNodes*

```
class SubNode : public BinaryNode {
public:
  SubNode(Node* inLeft, Node* inRight) : BinaryNode(inLeft, inRight) { }

  virtual ~SubNode() { }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

4 types of nodes are *BinaryNodes*

```
class MulNode : public BinaryNode {
public:
  MulNode(Node* inLeft, Node* inRight) : BinaryNode(inLeft, inRight) { }

  virtual ~MulNode() { }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

4 types of nodes are *BinaryNodes*

```
class DivNode : public BinaryNode {
public:
  DivNode(Node* inLeft, Node* inRight) : BinaryNode(inLeft, inRight) { }

  virtual ~DivNode() { }

  virtual void accept(NodeVisitor& inVisitor) const {
    inVisitor->accept(*this);
  }
};
```

The *NodeVisitor* interface

```
class NodeVisitor {
public:
  virtual ~NodeVisitor();

  virtual void visit(const AddNode& inNode) = 0;

  virtual void visit(const DivNode& inNode) = 0;

  virtual void visit(const FloatNode& inNode) = 0;

  virtual void visit(const IntNode& inNode) = 0;

  virtual void visit(const MulNode& inNode) = 0;

  virtual void visit(const SubNode& inNode) = 0;
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) {
    inNode.child(0)->accept(*this);
    mStream << '+';
    inNode.child(1)->accept(*this);
  }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) {
    inNode.child(0)->accept(*this);
    mStream << '-';
    inNode.child(1)->accept(*this);
  }
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) {
    mStream << inNode.value();
  }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) {
    mStream << inNode.value();
  }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) {
    mStream << '(';
    inNode.child(0)->accept(*this);
    mStream << ")*(";
    inNode.child(1)->accept(*this);
    mStream << ')';
  }

  virtual void visit(const SubNode& inNode) { ... }
};
```

The *PrettyPrinter*, a *NodeVisitor* for printing

```
class PrettyPrinter : public NodeVisitor {
  std::ostream& mStream;

public:
  PrettyPrinter(std::ostream& inStream) : mStream(inStream) { }

  virtual ~PrettyPrinter() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) {
    mStream << '(';
    inNode.child(0)->accept(*this);
    mStream << ")/(";
    inNode.child(1)->accept(*this);
    mStream << ')';
  }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  std::vector<double> mStack;

  double pop() {
    double lValue = mStack.back();
    mStack.pop_back();
    return lValue;
  }

  void push(double inValue) { mStack.push_back(inValue); }

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }

  double compute(Node* inNode) { ... }
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }

  double compute(Node* inNode) {
    mStack.clear();
    inNode->accept(*this);
    return mStack.back();
  }
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) {
    inNode.child(0)->accept(*this);
    inNode.child(1)->accept(*this);

    double lA = pop();
    double lB = pop();
    push(lA + lB);
  }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }

  ...
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) {
    inNode.child(0)->accept(*this);
    inNode.child(1)->accept(*this);

    double lA = pop();
    double lB = pop();
    push(lA - lB);
  }

  ...
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) {
    inNode.child(0)->accept(*this);
    inNode.child(1)->accept(*this);

    double lA = pop();
    double lB = pop();
    push(lA * lB);
  }

  virtual void visit(const SubNode& inNode) { ... }

  ...
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) {
    inNode.child(0)->accept(*this);
    inNode.child(1)->accept(*this);

    double lA = pop();
    double lB = pop();
    push(lA / lB);
  }

  virtual void visit(const FloatNode& inNode) { ... }

  virtual void visit(const IntNode& inNode) { ... }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }

  ...
};
```

The *Computer*, a *NodeVisitor* for computing

```
class Computer : public NodeVisitor {
  ...

public:
  virtual ~Computer() { }

  virtual void visit(const AddNode& inNode) { ... }

  virtual void visit(const DivNode& inNode) { ... }

  virtual void visit(const FloatNode& inNode) {
    mStack.push_back(inNode.value());
  }

  virtual void visit(const IntNode& inNode) {
    mStack.push_back(inNode.value());
  }

  virtual void visit(const MulNode& inNode) { ... }

  virtual void visit(const SubNode& inNode) { ... }

  ...
};
```

Usage

How does it look like when we use one of our visitors ?

```
Node* IRoot = ...

PrettyPrinter IPrinter(cout);
IRoot->accept(IPrinter);

Computer IComputer;
cout << '=' << IComputer.compute(IRoot) << endl;
```

Conclusion

We have successfully

- Separated structure and algorithms for it
- Adding new algorithms does not pollute interfaces
- But adding new types a bit inconvenient ...