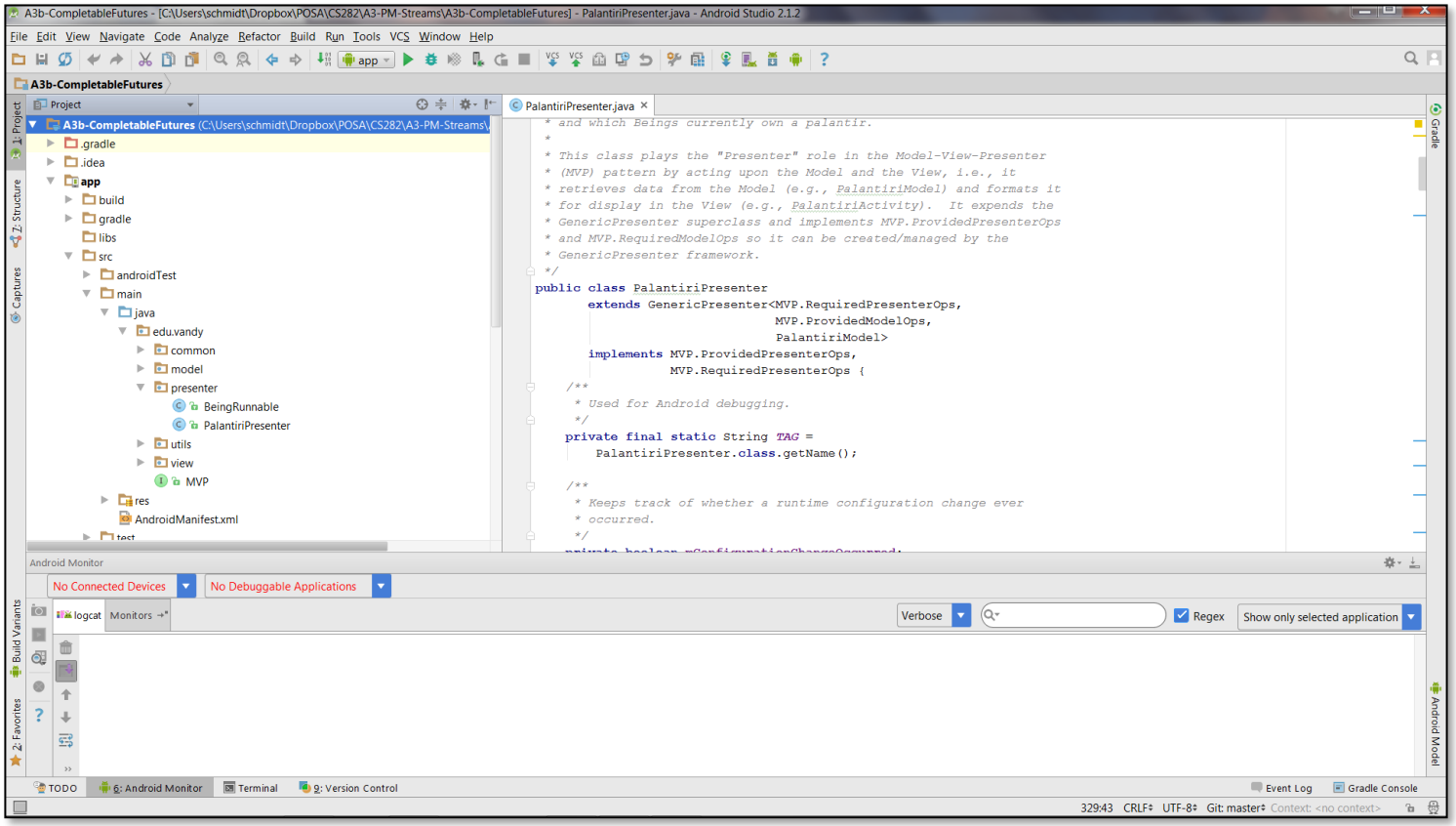
The Visitor Pattern

Implementation in C++

Douglas C. Schmidt

Learning Objectives in This Lesson

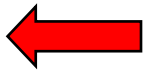
- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.
- Understand the *Visitor* pattern.
- Know how to implement the *Visitor* pattern in C++.



Visitor implementation in C++ (1/2)

- The **`Print_Visitor`** class prints character code or value for each node.

```
class Print_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Divide_Node &);
  // etc.
};
```

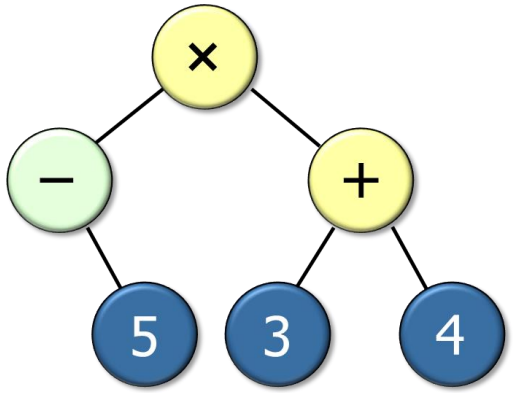


for all relevant `Component_Node` subclasses

Visitor implementation in C++ (1/2)

- The **`Print_Visitor`** class prints character code or value for each node.

```
class Print_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Divide_Node &);
  // etc.
};
```



- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitor_factory.make_visitor("print");
```

Factory method creates a print visitor

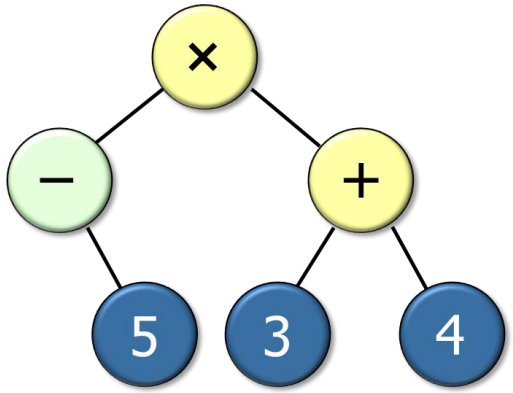
```
Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");


for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
  it->accept(print_visitor);
```

This visitor is stateless.

Visitor implementation in C++ (1/2)

- The **`Print_Visitor`** class prints character code or value for each node.

```
class Print_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Divide_Node &);
  // etc.
};
```



- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitor_factory.make_visitor("print");


Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");
```

Creational pattern makes an expression tree

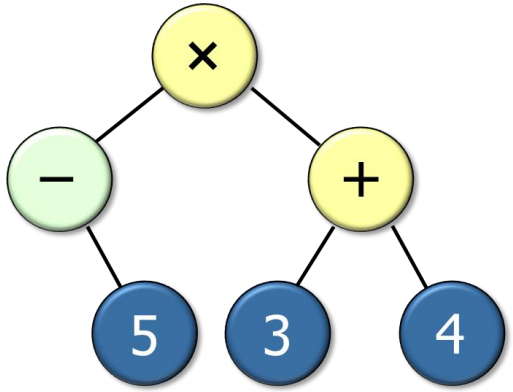
```
for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
  it->accept(print_visitor);
```

See earlier lessons on "*The Interpreter Pattern*" & "*The Builder Pattern*"

Visitor implementation in C++ (1/2)

- The **`Print_Visitor`** class prints character code or value for each node.

```
class Print_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Divide_Node &);
  // etc.
};
```



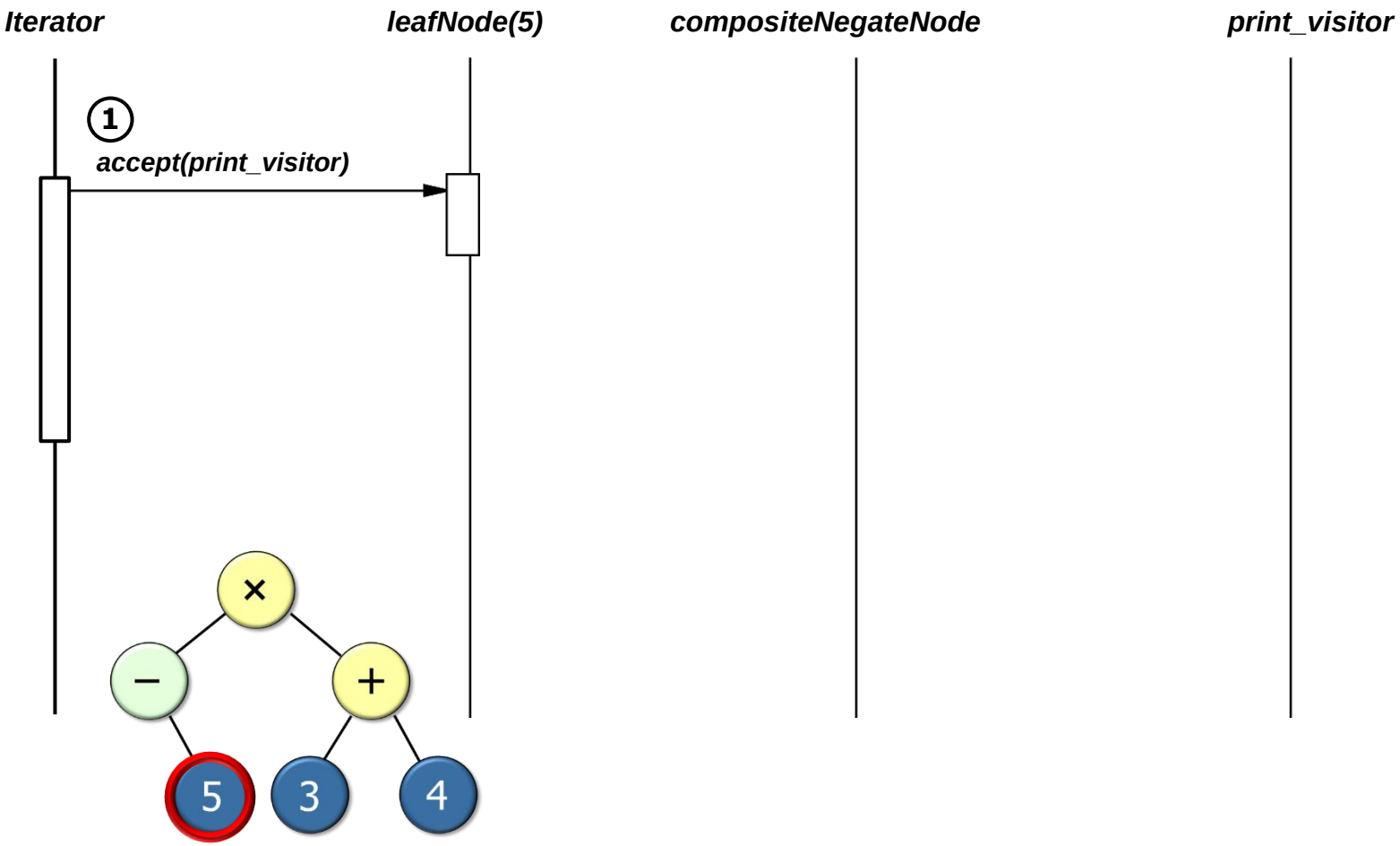
- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitor_factory.make_visitor("print");


Expression_Tree tree = make_expression_tree("-5 * (3 + 4)");


for(auto it = tree.begin("post-order");
    it != tree.end("post-order");
    ++it)
  it->accept(print_visitor);
```

`accept()` **forwards to implementor's** `accept()`**, which calls** `visit(*this)`

Visitor implementation in C++ (1/2)

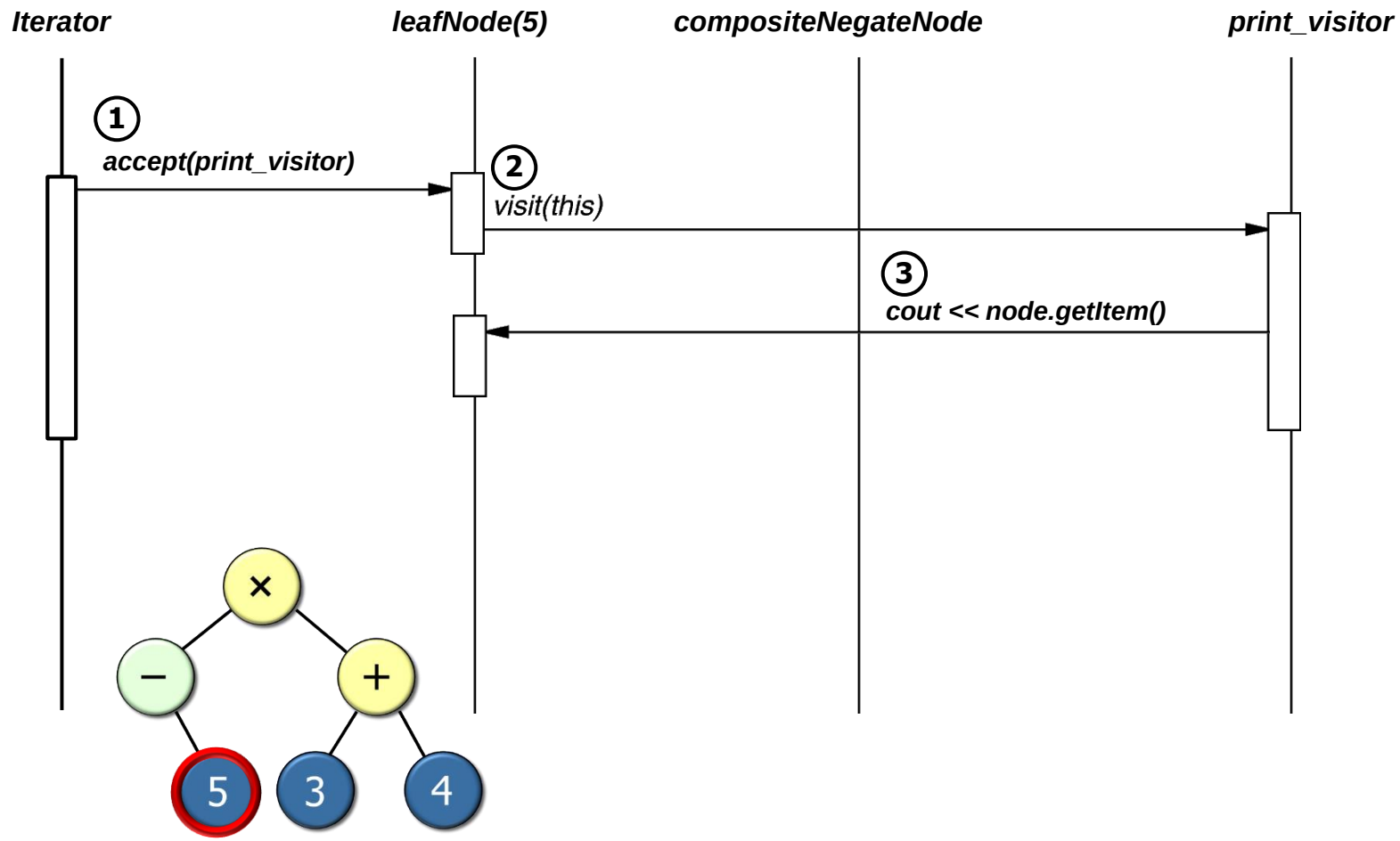
- Iterator controls the order in which `accept()` is called on each node in the tree.



This example is based on a "post-order" traversal.

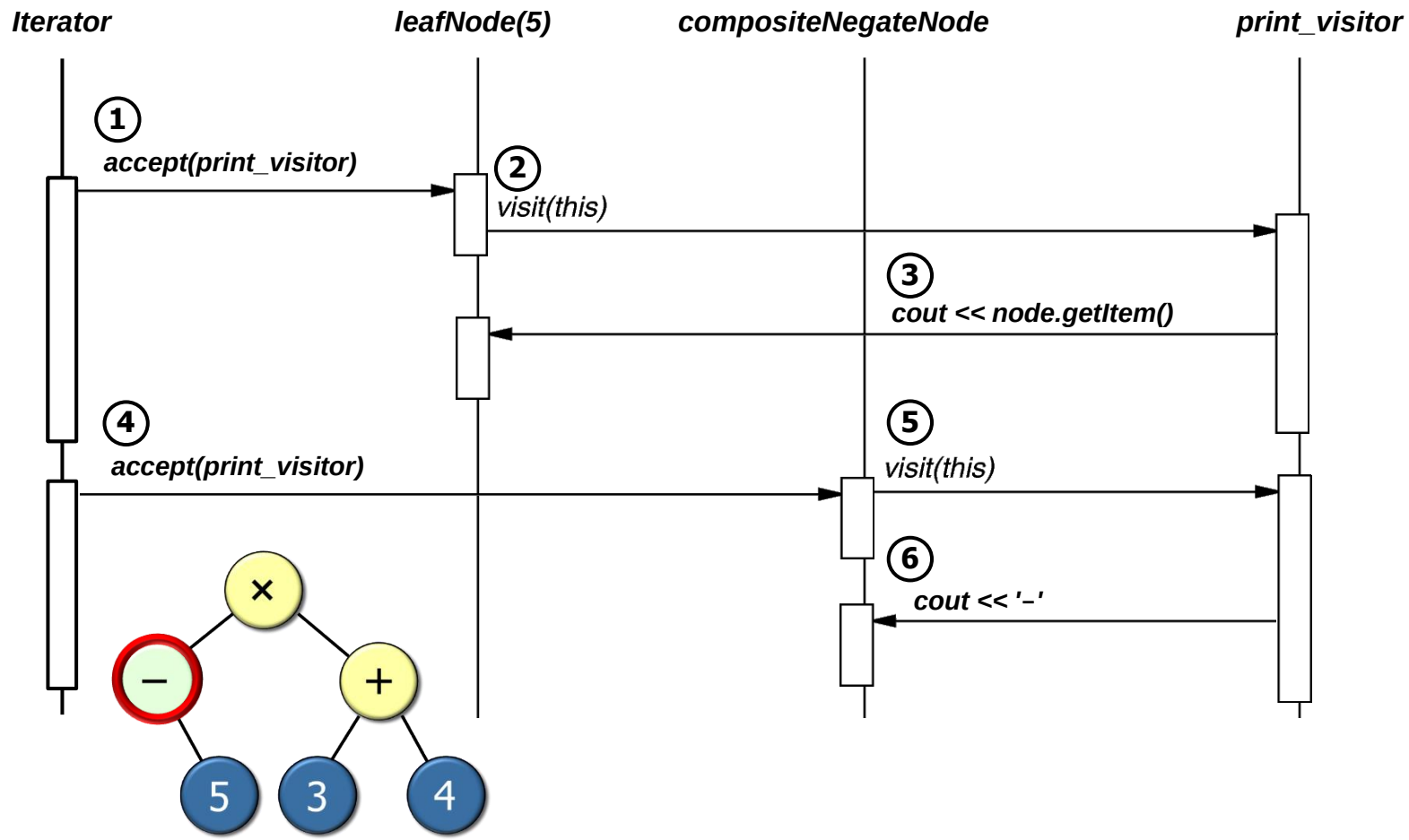
Visitor implementation in C++ (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.



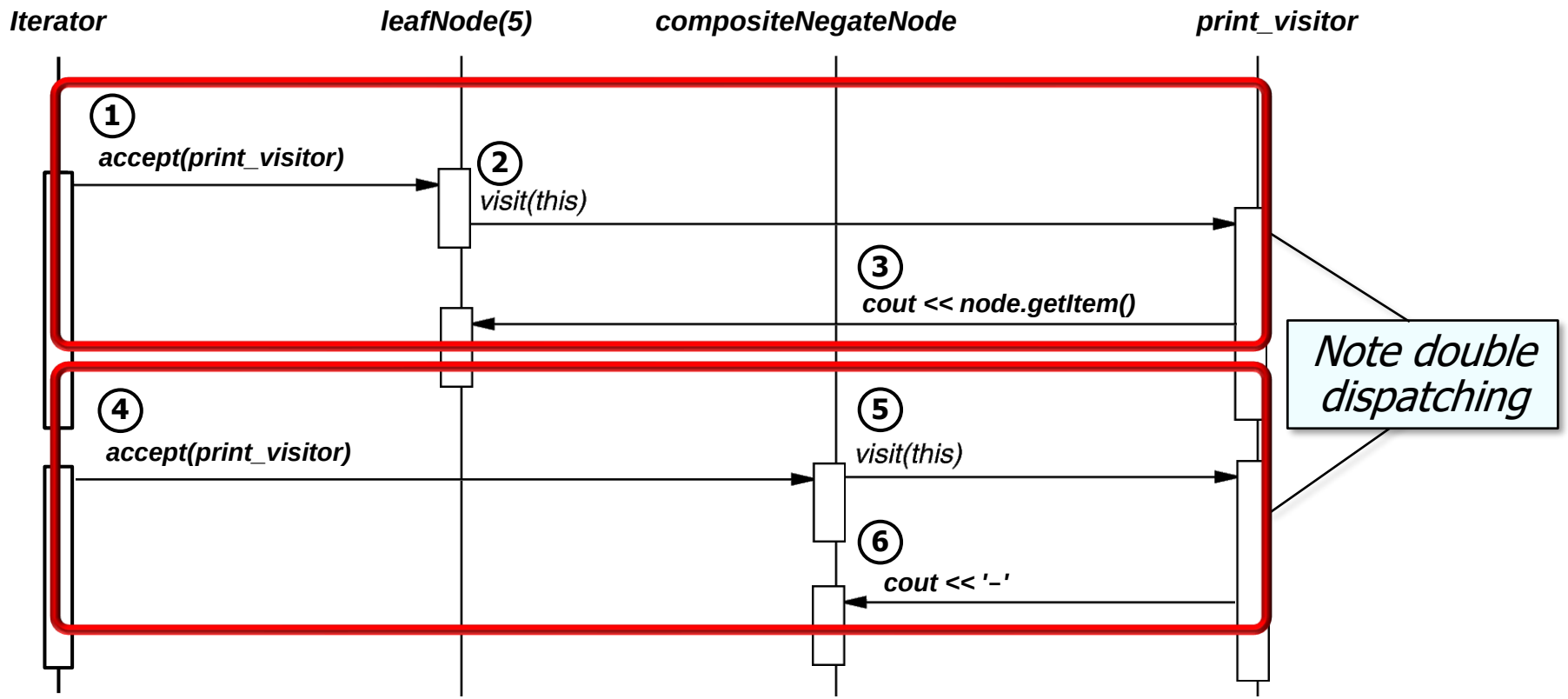
Visitor implementation in C++ (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.



Visitor implementation in C++ (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.

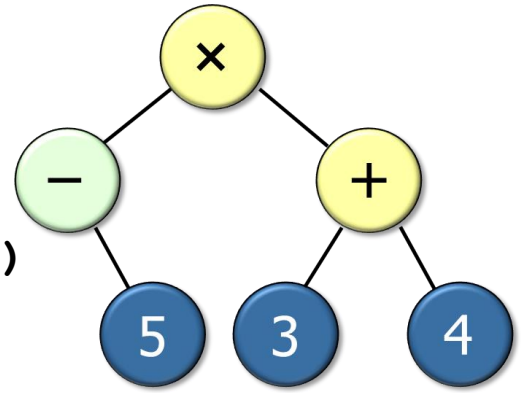


en.wikipedia.org/wiki/Double_dispatch has more info on double dispatching.

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
```

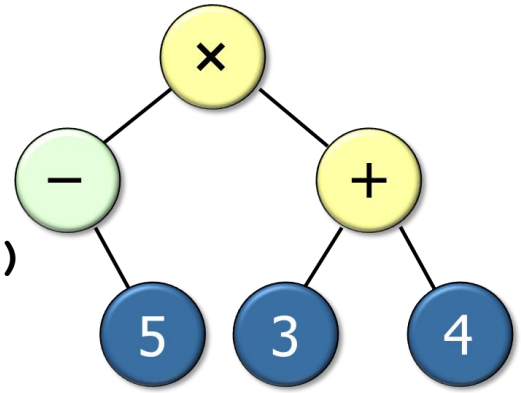


5~34+×

Visitor implementation in C++ (2/2)

- `Evaluation_Visitor` visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

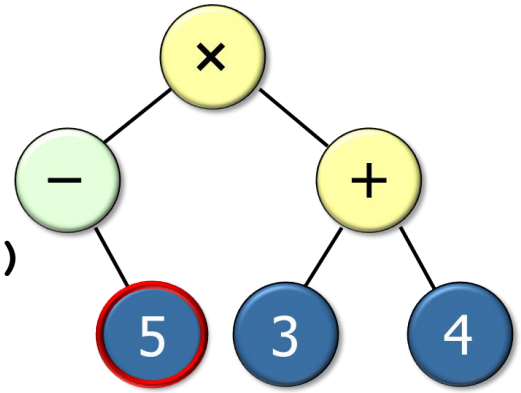
Stores post-order expression tree values processed incrementally during the iteration.

This visitor is stateful.

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



`5~34+×`

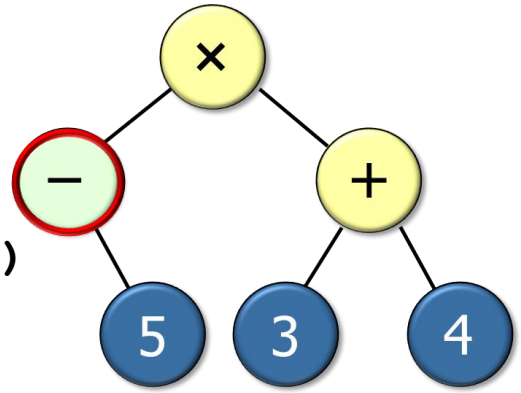
visit() method behavior

1. S = [5] `push(node.getItem())`

Visitor implementation in C++ (2/2)

- `Evaluation_Visitor` visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

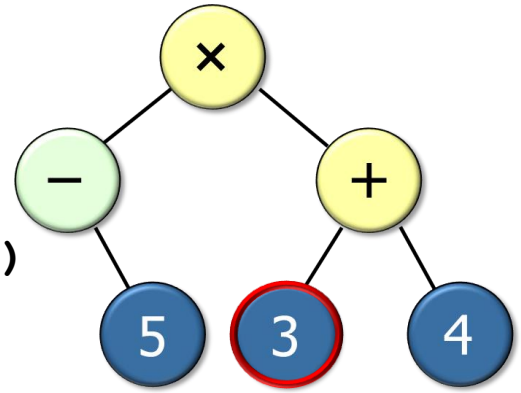
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

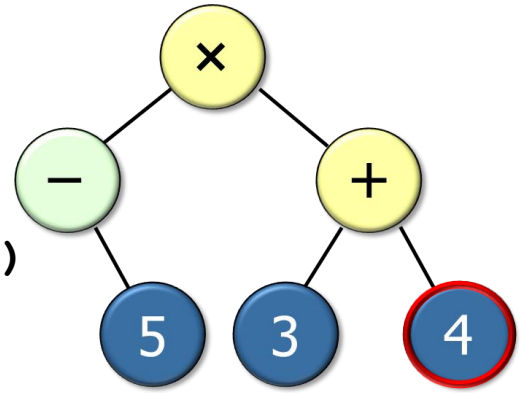
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] **`push(node.getItem())`**

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

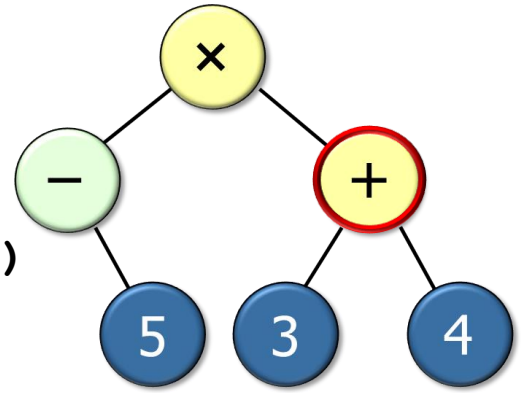
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.getItem())`
4. S = [-5, 3, 4] **`push(node.getItem())`**

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

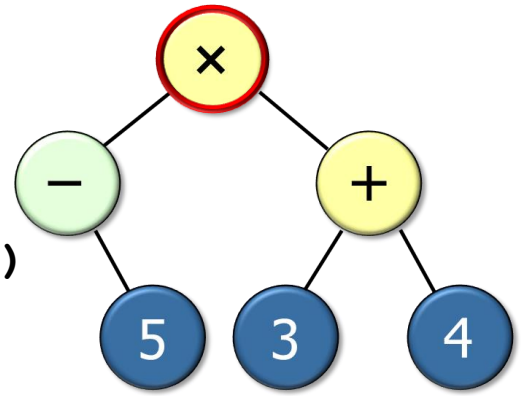
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.getItem())`
4. S = [-5, 3, 4] `push(node.getItem())`
5. S = [-5, 7] `push(pop()+pop())`

Visitor implementation in C++ (2/2)

- `Evaluation_Visitor` visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

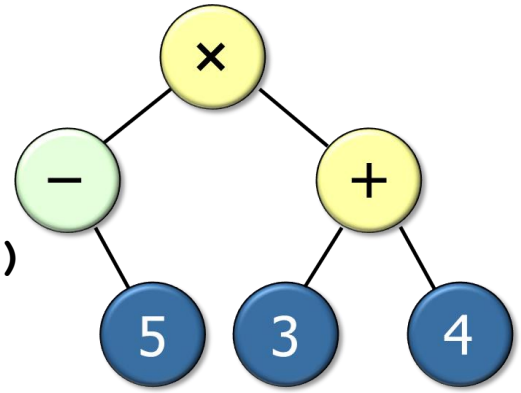
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.getItem())`
4. S = [-5, 3, 4] `push(node.getItem())`
5. S = [-5, 7] `push(pop()+pop())`
6. S = [-35] `push(pop()*pop())`

Visitor implementation in C++ (2/2)

- **`Evaluation_Visitor`** visits nodes via *post-order* iterator to compute yield

```
class Evaluation_Visitor : public Visitor {
  void visit(const Leaf_Node &);
  void visit(const Composite_Negate_Node &);
  void visit(const Composite_Add_Node &);
  void visit(const Composite_Multiply_Node &)
  // etc.
  ...
  stack<int> stack_;
}
```



5~34+×

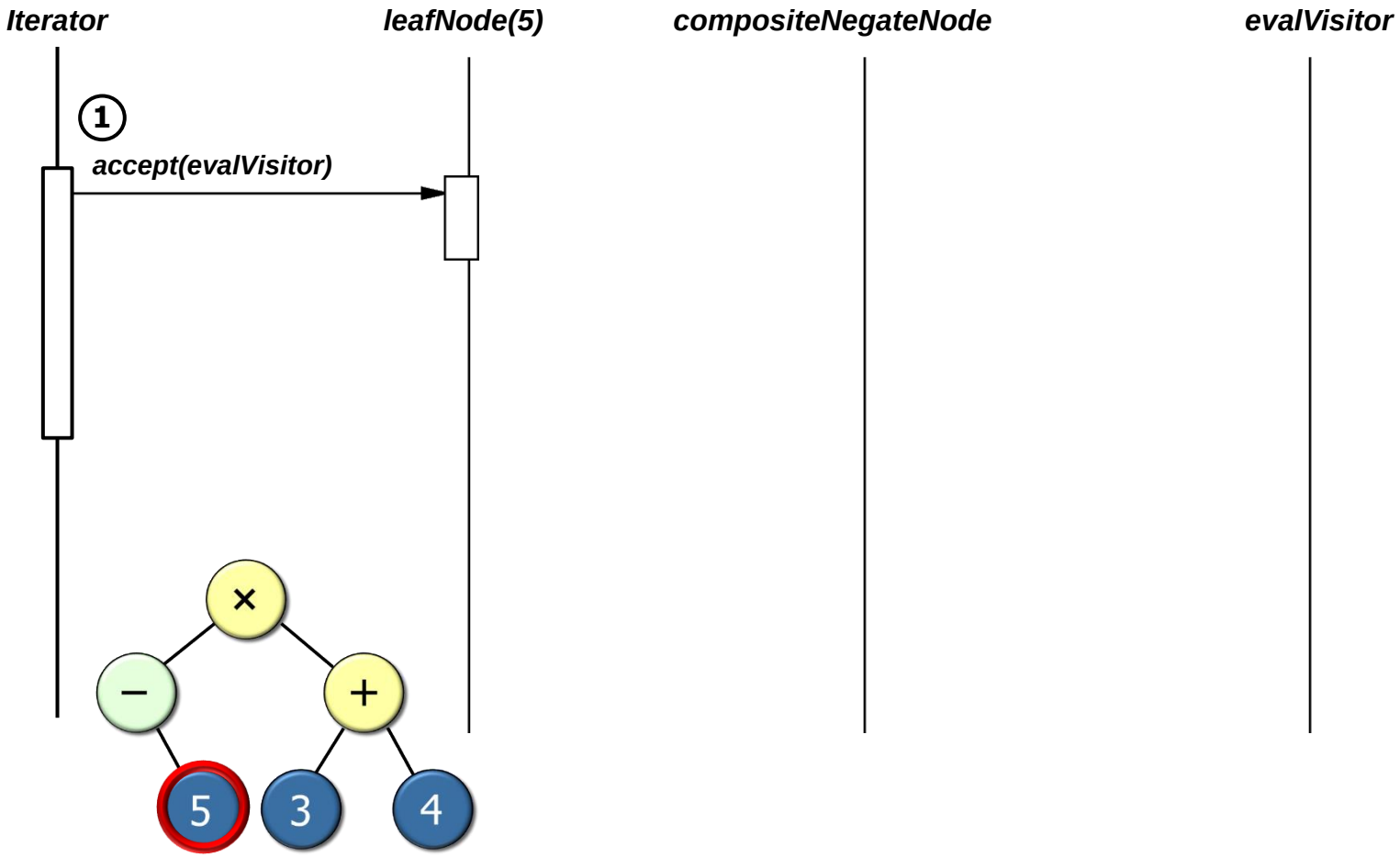
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.getItem())`
4. S = [-5, 3, 4] `push(node.getItem())`
5. S = [-5, 7] `push(pop()+pop())`
6. S = [-35] `push(pop()*pop())`

The final top stack item contains the "yield" of the expression tree.

Visitor implementation in C++ (2/2)

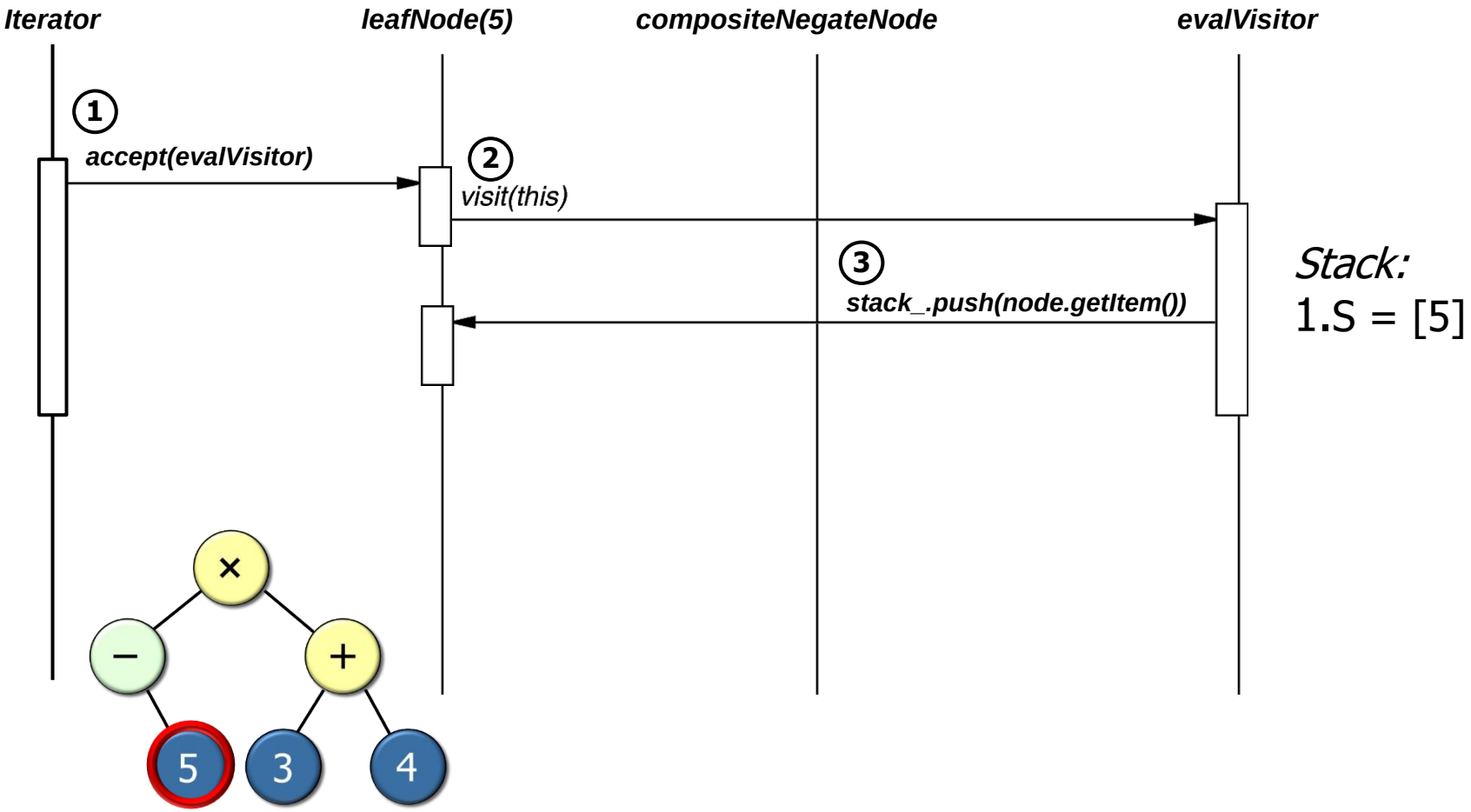
- Iterator controls the order in which `accept()` is called on each node in the tree.



This example is based on a "post-order" traversal.

Visitor implementation in C++ (2/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired evaluation action.



Visitor implementation in C++ (2/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired evaluation action.