The Iterator Pattern

Other Considerations

Douglas C. Schmidt
Learning Objectives in This Lesson

• Recognize how the *Iterator* pattern can be applied to access all nodes in an expression tree flexibly & extensibly.
• Understand the structure & functionality of the *Iterator* pattern.
• Know how to implement the *Iterator* pattern in C++.
• Be aware of other considerations when applying the *Iterator* pattern.
Consequences

+ **Flexibility**
  - Aggregate & traversal objects are decoupled & can (co)evolve separately
Consequences

+ **Flexibility**

  - Aggregate & traversal objects are decoupled & can (co)evolve separately

Adding new traversal algorithms shouldn’t affect the expression tree elements.
Consequences

+ **Flexibility**

- Aggregate & traversal objects are decoupled & can (co)evolve separately

---

**Adding new subclasses of**

*Composite_Binary_Node*

*shouldn’t affect the iterators.*
Consequences

+ Multiplicity
  - Supports multiple iterators & multiple traversal algorithms

These traversals can all occur simultaneously on the same expression tree instance.

- "In-order" traversal = $-5 \times (3+4)$
- "Pre-order" traversal = $\times -5+34$
- "Post-order" traversal = $5-34+\times$
- "Level-order" traversal = $\times --+534$

Later we’ll apply the Strategy pattern to support multiple traversal algorithms.
Consequences

- **Overhead**
  - Additional communication between iterator & aggregate

**Diagram:**

```
Aggregate
  CreatelIterator()

ConcreateAggregate
  CreatelIterator()

Client

Iterator
  First()
  Next()
  IsDone()
  CurrentItem()

ConcreateIterator

return new ConcreateIterator(this)
```

*Significant overhead can occur if there is a distribution or user/kernel boundary crossing.*

This overhead is quite problematic for iterators in concurrent or distributed systems.
Consequences

- **Dependencies**
  - The iterator implementation may depend on the aggregate’s implementation
Consequences

- Dependencies
  - The iterator implementation may depend on the aggregate’s implementation

Adding a new subclass for Composite_Ternary_Node may affect the iterators.
Implementation considerations

• Iterator style
  • Java iterators vs. GoF iterators
    • Java iterators are similar—but not identical to—GoF iterators, e.g.,
      
      ```java
      for (Iterator<ExpressionTree> it = exprTree.iterator();
           it.hasNext();)
        doSomethingWithIterator(it.next());
      ```

See [docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)
Implementation considerations

• Iterator style
  
• Java iterators vs. GoF iterators
  
  • Java iterators are similar—but not identical to—GoF iterators, e.g.,

        for(Iterator<ExpressionTree> it = exprTree.iterator();
            it.hasNext();)
        doSomethingWithIterator(it.next());

  
  • Here’s the equivalent Java code for GoF-style iterators

        for(GoFIterator it = tree.createIterator();
            !it.done();
            it.advance())
        doSomethingWithIterator(it.currentElement());
Implementation considerations

- Iterator style
  - Java iterators vs. C++ STL iterators
    - C++ Standard Template Library (STL) iterators mimic native C/C++ pointer arithmetic syntax/semantics

```cpp
for (auto it = expr_tree.begin ();
     it != expr_tree.end ();
     ++it)
  do_something_with_iterator (*it);
```

See [www.geeksforgeeks.org/iterators-c-stl](http://www.geeksforgeeks.org/iterators-c-stl)
Implementation considerations

- Iterator style
  - Java iterators vs. C++ STL iterators
    - C++ Standard Template Library (STL) iterators mimic native C/C++ pointer arithmetic syntax/semantics
      ```cpp
      for (auto it = expr_tree.begin ();
           it != expr_tree.end ();
           ++it)
        do_something_with_iterator (*it);
      ```
    - Java iterators are closer to the GoF Iterator pattern
      ```java
      for (Iterator<ExpressionTree> it = exprTree.iterator ();
           it.hasNext();)
        doSomethingWithIterator(it.next());
      ```
Implementation considerations

- Iterator style
  - Java iterators vs. C++ STL iterators
  - Java also supports a “Spliterator” (splitable iterator)

```java
Consumer<Expression_Tree> action;

for (Spliterator<Expression_Tree> s = exprTree.spliterator();
     split.tryAdvance(action);)
    doSomethingWithSpliterator(s);
```

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

Create a spliterator for an expression tree
Implementation considerations

• Iterator style

  • Java iterators vs. C++ STL iterators

  • Java also supports a “Spliterator” (splitable iterator)

```java
Consumer<Expression_Tree> action;

for (Spliterator<Expression_Tree> s = exprTree.spliterator();
     split.tryAdvance(action);)
    doSomethingWithSpliterator(s);
```

`tryAdvance()` combines `hasNext()` & `next()`. 
Implementation considerations

- Internal iterators vs. external iterators

```java
List<URL> newUrls = urlList
    .stream()
    .filter(s -> s.contains("cse.wustl"))
    .map(s -> s.replace("cse.wustl", "dre.vanderbilt"))
    .map(rethrowFunction(URL::new))
    .collect(toList());
```

```java
List<URL> newUrls =
    new ArrayList<URL>();
...
for (Iterator<List> i = newUrls.iterator(); i.hasNext(); ) {
    String url = i.next();
    if (!url.contains("cse.wustl")) continue;
    else
        newUrls.add(new URL(url.replace("cse.wustl", "dre.vanderbilt")));
}
```

See www.javabrahman.com/java-8/java-8-internal-iterators-vs-external-iterators
Implementation considerations

• Internal iterators vs. external iterators

```java
List<URL> newUrls = urlList
    .stream()
    .filter(s -> s.contains("cse.wustl"))
    .map(s -> s.replace("cse.wustl", "dre.vanderbilt"))
    .map(rethrowFunction(URL::new))
    .collect(toList());
```

Java external iterators are more flexible, but are more complicated to program.

```java
List<URL> newUrls =
    new ArrayList<URL>();
...
for (Iterator<List> i = newUrls.iterator(); i.hasNext(); ) {
    String url = i.next();
    if (!url.contains("cse.wustl")) continue;
    else
        newUrls.add(new URL(url.replace("cse.wustl", "dre.vanderbilt")));
}
```
Implementation considerations

- Internal iterators vs. external iterators

Java internal iterators are easier to program, but are less flexible.

```java
List<URL> newUrls = urlList
    .stream()
    .filter(s -> s.contains("cse.wustl"))
    .map(s -> s.replace("cse.wustl", "dre.vanderbilt"))
    .map(rethrowFunction(URL::new))
    .collect(toList());

List<URL> newUrls =
    new ArrayList<URL>();
...
for (Iterator<List> i = newUrls.iterator(); i.hasNext(); ) {
    String url = i.next();
    if (!url.contains("cse.wustl")) continue;
    else
        newUrls.add(new URL(url.replace("cse.wustl", "dre.vanderbilt")));
}
Implementation considerations

• Robust iterators

• Enable insertions & deletions on the aggregate during the iteration process
Implementation considerations

- Violating the aggregate’s encapsulation

```java
private class Itr
    implements Iterator<E> {
    int cursor
    int lastRet = -1;
    int expectedModCount = modCount;

    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData =
            ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new
                ConcurrentModificationException();
        cursor = i + 1;
        return (E)elementData[lastRet = i];
    }
```

*Itr.next() hard-codes a dependency on the ArrayList implementation.*

See [share/classes/java/util/ArrayList.java](share/classes/java/util/ArrayList.java)
## Implementation considerations

- Overhead & behavior in concurrent programs

<table>
<thead>
<tr>
<th>Feature</th>
<th>Fail Fast Iterator</th>
<th>Fail Safe Iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throws Concurrent Modification Exception</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Clone object</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory overhead</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Examples</td>
<td>HashMap, Vector, ArrayList, HashSet</td>
<td>CopyOnWriteArrayList, ConcurrentHashMap</td>
</tr>
</tbody>
</table>

Implementation considerations

- Batching in programs that cross distribution or user/kernel boundaries

Batch Iterator is a pattern compound that minimizes the impact of latency.

See www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf
Known uses

- Unidraw Iterator
- C++ STL iterators
- C buffered I/O
- C++11 range-based for loops & Java for-each loops
- JDK Iterator, Iterable, & Spliterator

### Interface Iterator\(<E>\>

**Type Parameters:**

\(E\) - the type of elements returned by this iterator

**All Known Subinterfaces:**

ListIterator\(<E>\), PrimitiveIterator\(<T, T_{CONS}>\),
PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt,
PrimitiveIterator.OfLong, XMLEventReader

**All Known Implementing Classes:**

BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

### Interface Spliterator\(<T>\>

**Type Parameters:**

\(T\) - the type of elements returned by this Spliterator

**All Known Subinterfaces:**

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,
Spliterator.OfPrimitive\(<T, T_{CONS}, T_{SPLITR}>\)

**All Known Implementing Classes:**

Spliterators.AbstractDoubleSpliterator,
Spliterators.AbstractIntSpliterator,
Summary of the Iterator Pattern

- **Iterator creates objects that traverse the Composite-based expression tree & access each of its elements one at a time.**

We’ll combine *Iterator* with other patterns to further improve our app design.