The Iterator Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

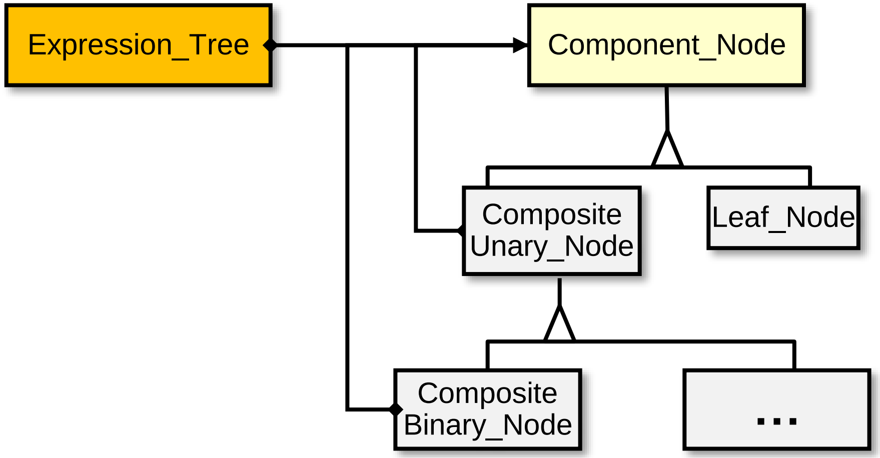
- Recognize how the *Iterator* pattern can be applied to access all nodes in an expression tree flexibly & extensibly.
- Understand the structure & functionality of the *Iterator* pattern.
- Know how to implement the *Iterator* pattern in C++.
- Be aware of other considerations when applying the *Iterator*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

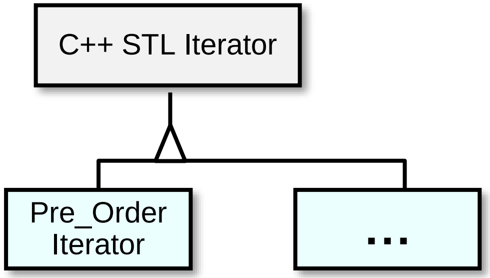
Iterator GoF Object Behavioral

Consequences

+ *Flexibility*

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



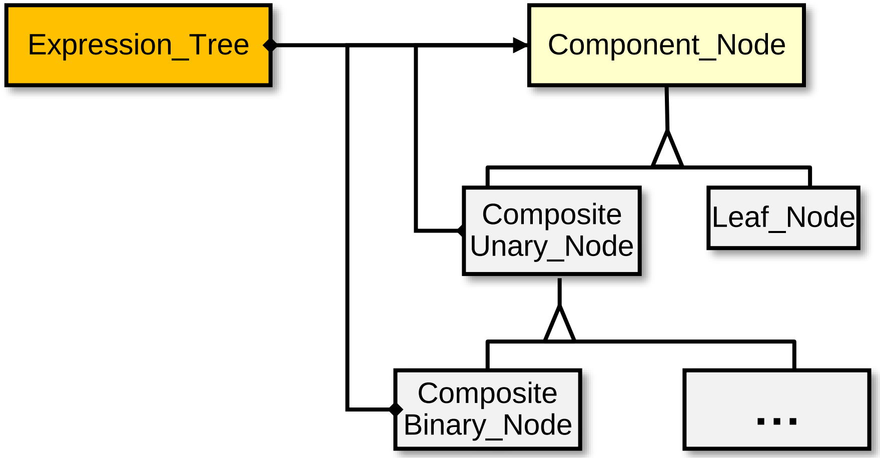


Iterator GoF Object Behavioral

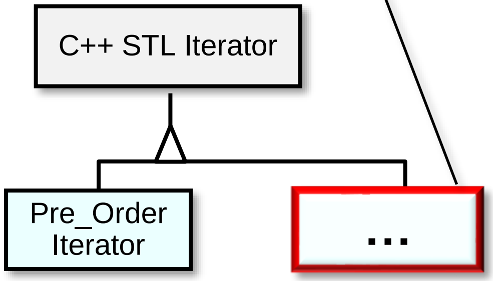
Consequences

+ *Flexibility*

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



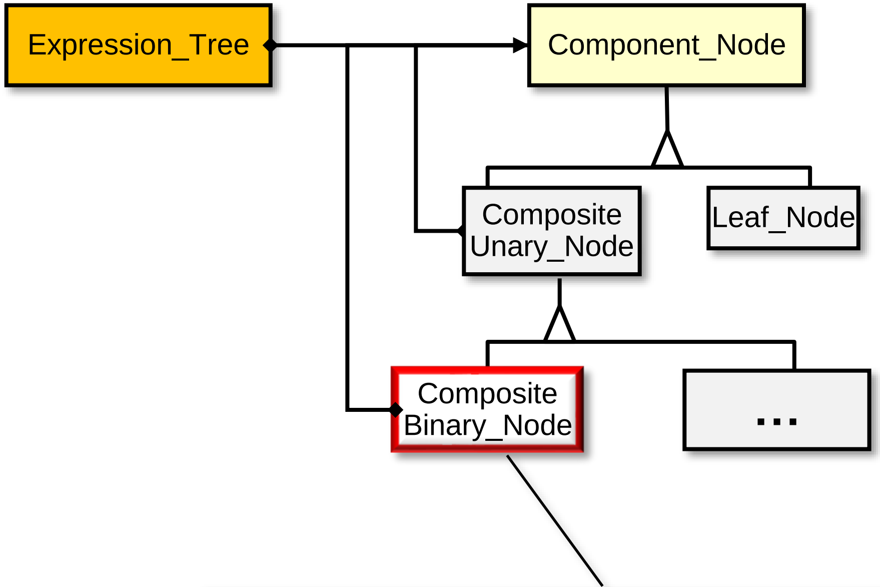
Adding new traversal algorithms shouldn't affect the expression tree elements.

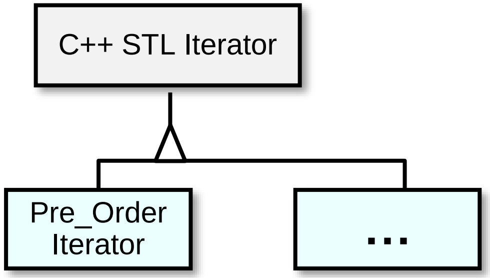


Consequences

+ *Flexibility*

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





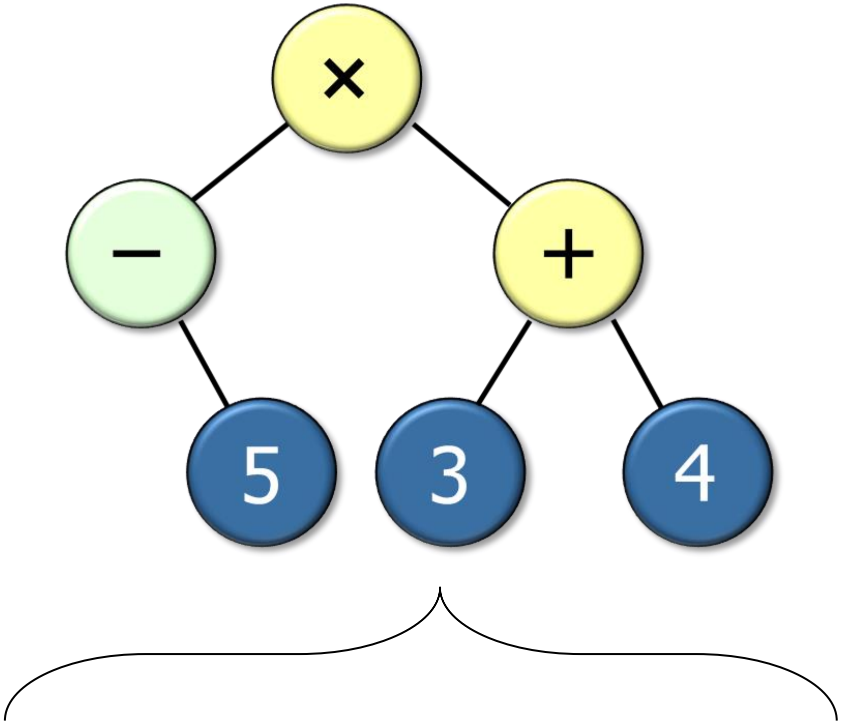
Adding new subclasses of ***`Composite_Binary_Node`*** *shouldn't affect the iterators.*

Iterator GoF Object Behavioral

Consequences

+ *Multiplicity*

  - Supports multiple iterators & multiple traversal algorithms



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

- "In-order" traversal = **−5×(3+4)**
- "Pre-order" traversal = **×−5+34**
- "Post-order" traversal = **5−34+×**
- "Level-order" traversal = **×−+534**

Later we'll apply the *Strategy* pattern to support multiple traversal algorithms.

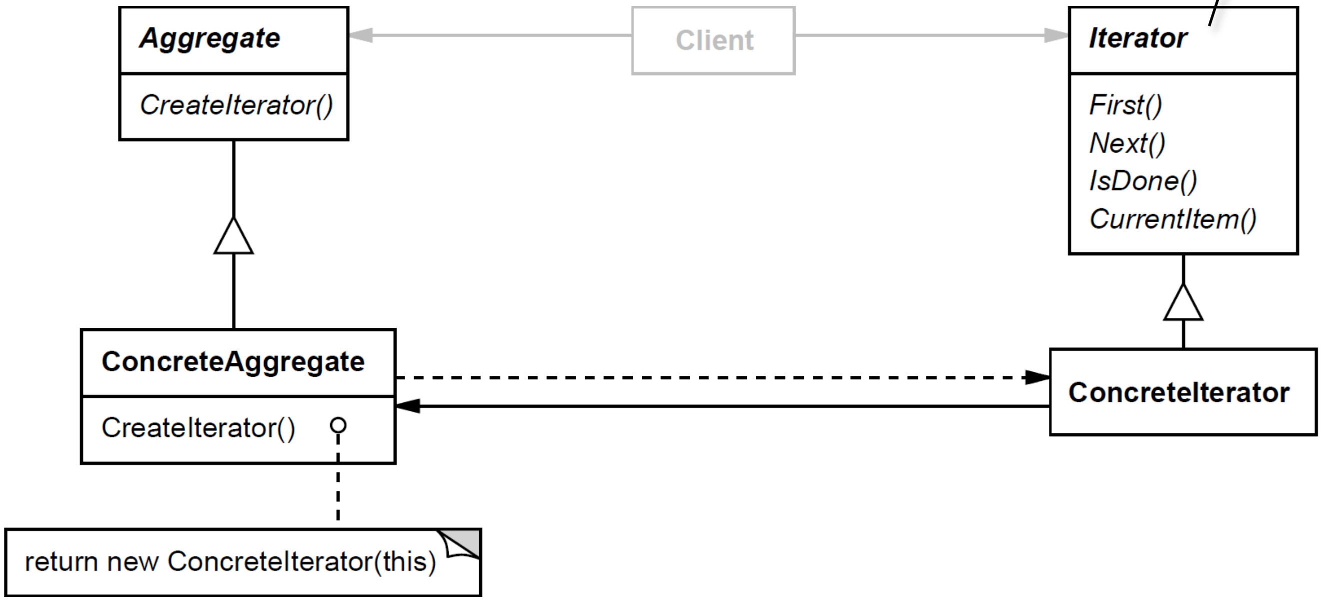
Iterator GoF Object Behavioral

Consequences

– *Overhead*

- Additional communication between iterator & aggregate

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



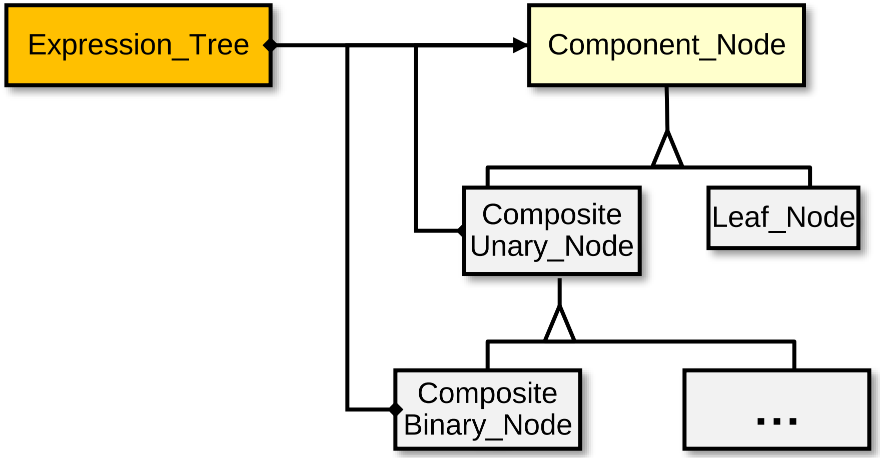
This overhead is quite problematic for iterators in concurrent or distributed systems.

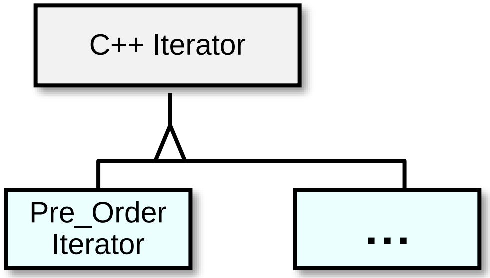
Iterator GoF Object Behavioral

Consequences

– *Dependencies*

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



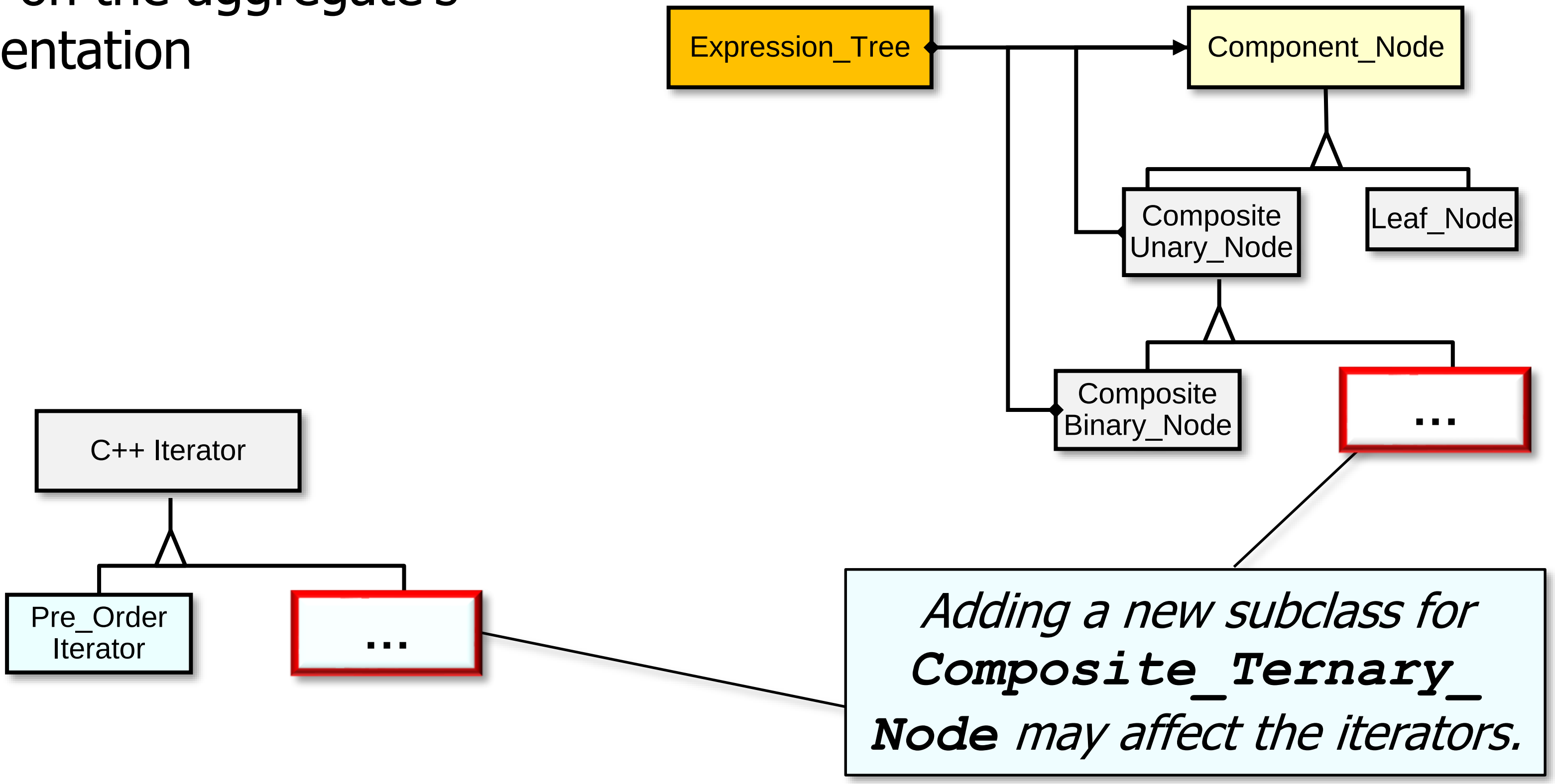


Iterator GoF Object Behavioral

Consequences

– *Dependencies*

- The iterator implementation may depend on the aggregate's implementation



Adding a new subclass for **`Composite_Ternary_Node`** *may affect the iterators.*

Iterator GoF Object Behavioral

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());
```

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

Iterator GoF Object Behavioral

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());
```

Iterator GoF Object Behavioral

Implementation considerations

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

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

See 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

```
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

```
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)

```
Consumer<Expression_Tree> action;

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

Create a spliterator for an expression tree

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

Iterator GoF Object Behavioral

Implementation considerations

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

```
Consumer<Expression_Tree> action;

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

tryAdvance() combines hasNext() & next()

Iterator GoF Object Behavioral

Implementation considerations

- Internal iterators vs. external iterators

```
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")));
}
```

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

Implementation considerations

- Internal iterators vs. external iterators

```
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.

```
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

```
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());
```

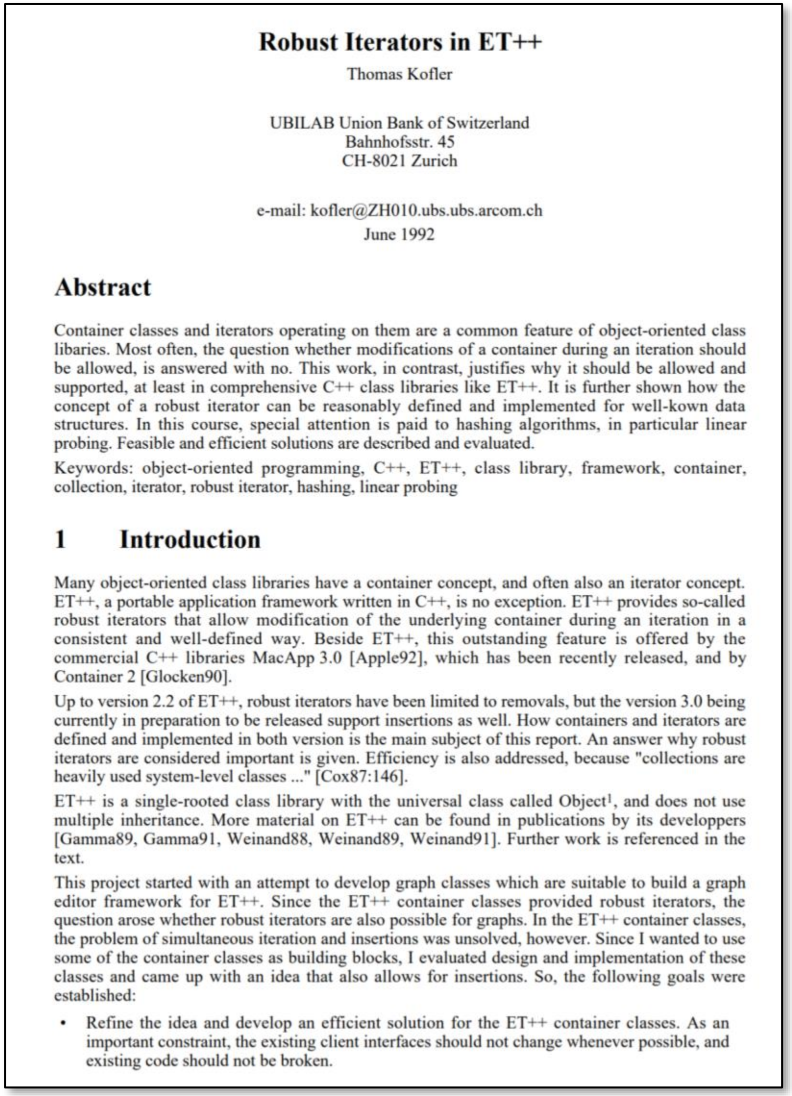
Java internal iterators are easier to program, but are less flexible.

```
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")));
}
```

Iterator GoF Object Behavioral

Implementation considerations

- Robust iterators
  - Enable insertions & deletions on the aggregate during the iteration process



Robust Iterators in ET++

Thomas Kofler

UBILAB Union Bank of Switzerland
Bahnhofsstr. 45
CH-8021 Zurich

e-mail: kofler@ZH010.ubs.ubs.arcom.ch

June 1992

Abstract

Container classes and iterators operating on them are a common feature of object-oriented class libaries. Most often, the question whether modifications of a container during an iteration should be allowed, is answered with no. This work, in contrast, justifies why it should be allowed and supported, at least in comprehensive C++ class libraries like ET++. It is further shown how the concept of a robust iterator can be reasonably defined and implemented for well-kown data structures. In this course, special attention is paid to hashing algorithms, in particular linear probing. Feasible and efficient solutions are described and evaluated.

Keywords: object-oriented programming, C++, ET++, class library, framework, container, collection, iterator, robust iterator, hashing, linear probing

1 Introduction

Many object-oriented class libraries have a container concept, and often also an iterator concept. ET++, a portable application framework written in C++, is no exception. ET++ provides so-called robust iterators that allow modification of the underlying container during an iteration in a consistent and well-defined way. Beside ET++, this outstanding feature is offered by the commercial C++ libraries MacApp 3.0 [Apple92], which has been recently released, and by Container 2 [Glocken90].

Up to version 2.2 of ET++, robust iterators have been limited to removals, but the version 3.0 being currently in preparation to be released support insertions as well. How containers and iterators are defined and implemented in both version is the main subject of this report. An answer why robust iterators are considered important is given. Efficiency is also addressed, because "collections are heavily used system-level classes ..." [Cox87:146].

ET++ is a single-rooted class library with the universal class called Object[1], and does not use multiple inheritance. More material on ET++ can be found in publications by its developpers [Gamma89, Gamma91, Weinand88, Weinand89, Weinand91]. Further work is referenced in the text.

This project started with an attempt to develop graph classes which are suitable to build a graph editor framework for ET++. Since the ET++ container classes provided robust iterators, the question arose whether robust iterators are also possible for graphs. In the ET++ container classes, the problem of simultaneous iteration and insertions was unsolved, however. Since I wanted to use some of the container classes as building blocks, I evaluated design and implementation of these classes and came up with an idea that also allows for insertions. So, the following goals were established:

- Refine the idea and develop an efficient solution for the ET++ container classes. As an important constraint, the existing client interfaces should not change whenever possible, and existing code should not be broken.

See www.ubilab.org/publications/print_versions/pdf/robustiter-structprog93.pdf

Iterator GoF Object Behavioral

Implementation considerations

- Violating the aggregate’s encapsulation

```
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

Iterator GoF Object Behavioral

Implementation considerations

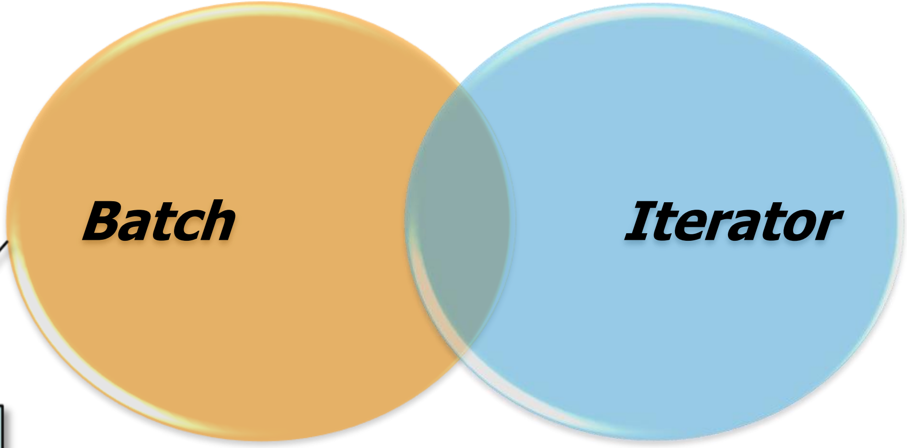
- Overhead & behavior in concurrent programs

	Fail Fast Iterator	**Fail Safe Iterator**
Throw Concurrent Modification Exception	Yes	No
Clone object	No	Yes
Memory overhead	No	Yes
Examples	HashMap, Vector, ArrayList, HashSet	CopyOnWriteArrayList, ConcurrentHashMap

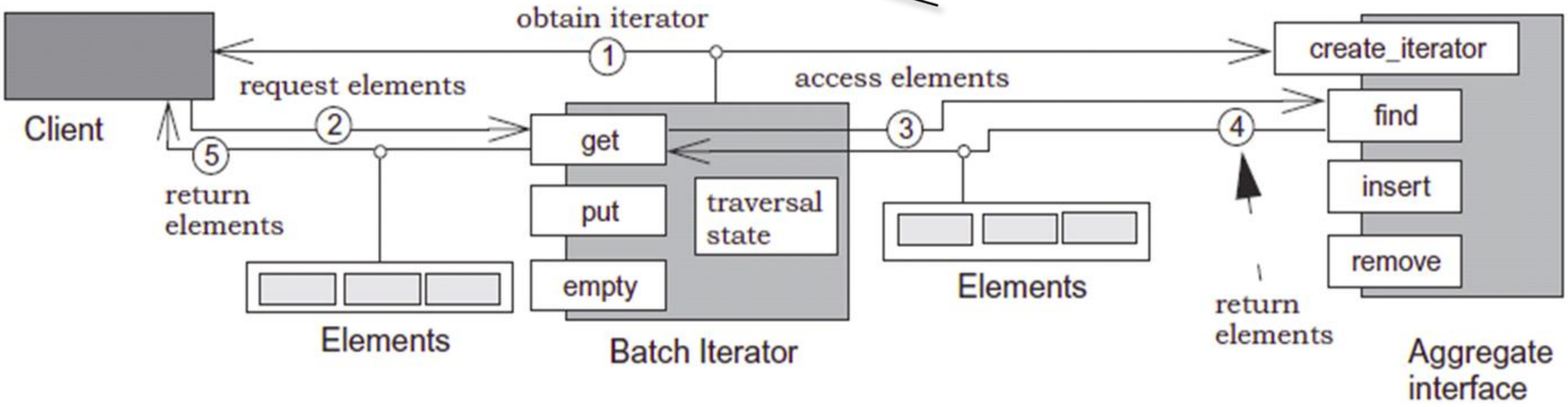
See javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html

Implementation considerations

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



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

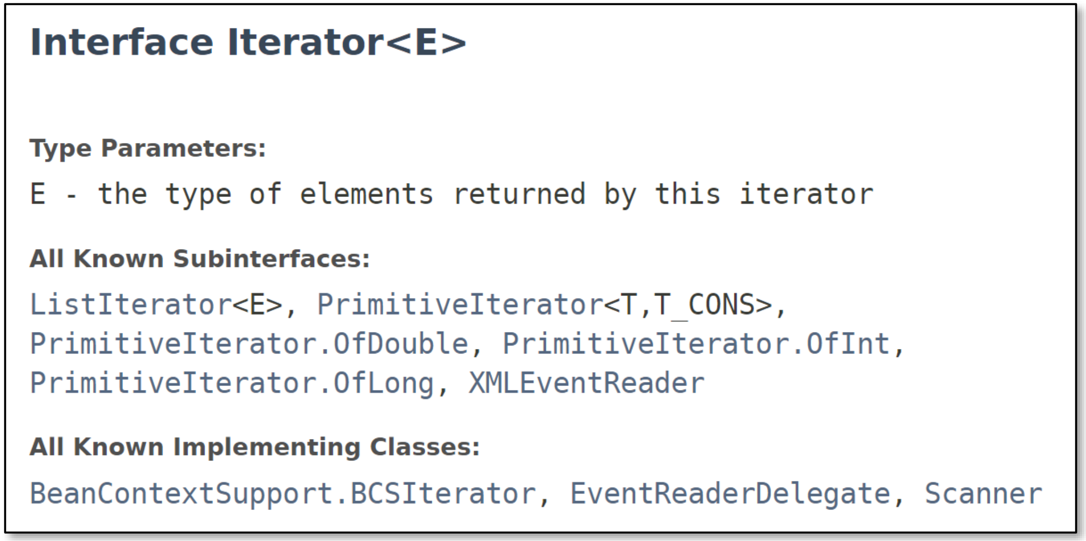


See www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf

Iterator GoF Object Behavioral

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:

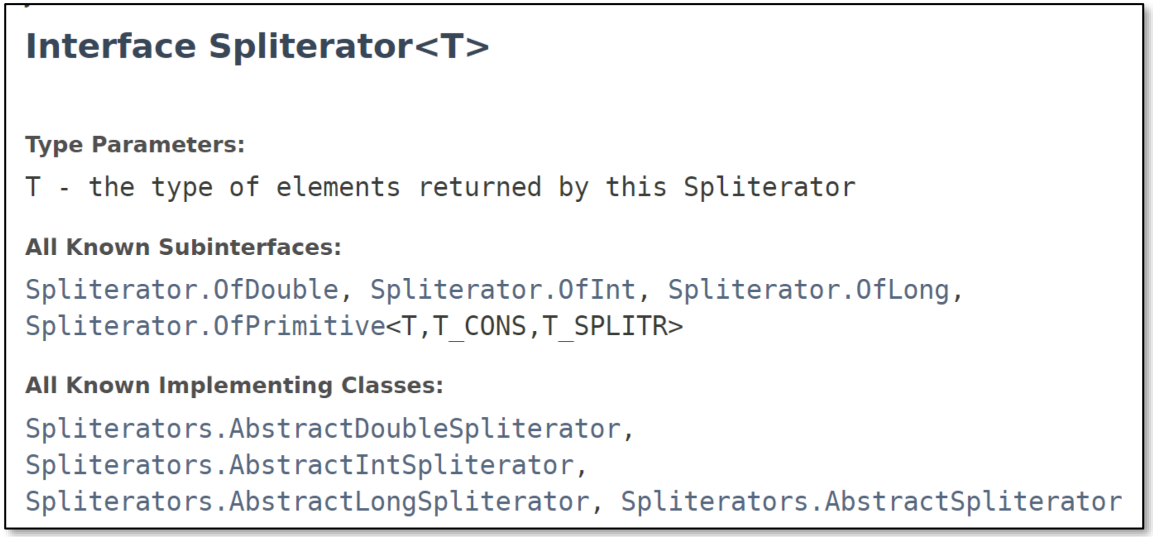
`E - the type of elements returned by this iterator`

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:

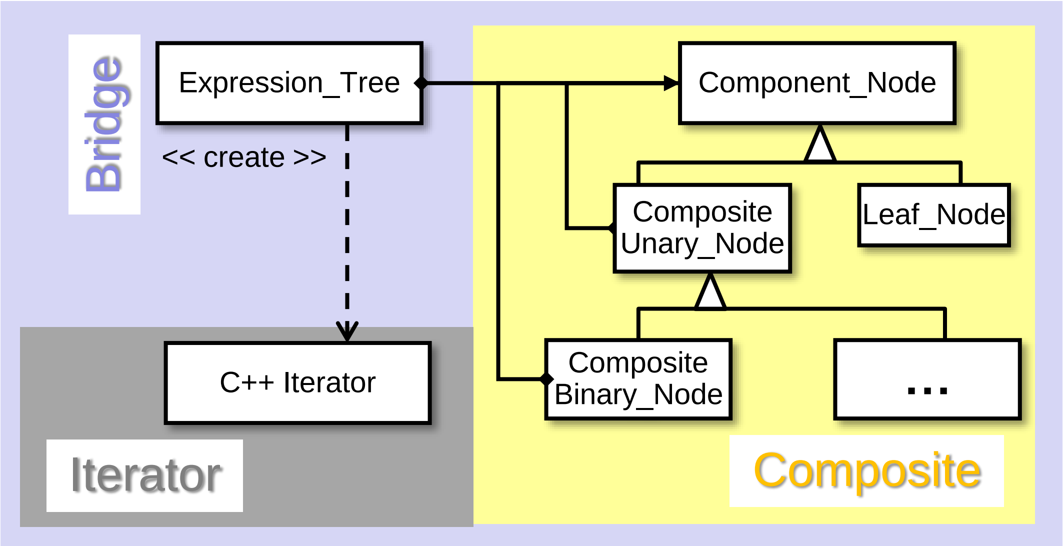
`Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong, Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>`

All Known Implementing Classes:

`Spliterators.AbstractDoubleSpliterator, Spliterators.AbstractIntSpliterator, Spliterators.AbstractLongSpliterator, Spliterators.AbstractSpliterator`

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.