Contents

Iterator ........................................................................................................... 2
Issues .............................................................................................................. 12
  Concrete vs. Polymorphic Iterators ....................................................... 12
  Who Controls the iteration? ................................................................. 13
  Who Defines the Traversal Algorithm? ............................................... 15
  How Robust is the iterator? ................................................................. 16
  Iterators and Privileged Access ........................................................ 17
  Iterators for Composites ................................................................. 18
  Null Iterator ...................................................................................... 18

References

Design Patterns: Elements of Reusable Object-Oriented Software,
Gamma, Helm, Johnson, Vlissides, 1995, pp. 257-271

Reading

Design Patterns: pp. 257-271

Copyright ©, All rights reserved. 2002 SDSU & Roger Whitney, 5500 Campanile Drive, San
Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.
Iterator

Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.

Java Examples

Enumeration, Iterator, and Streams in Java are iterators.

    Vector listOfStudents = new Vector();
    // code to add students not shown
    Iterator list = listOfStudents.iterator();
    while ( list.hasNext() )
        Console.println( list.next() );

    Hashtable anIndex = new HashMap();
    // code to add elements to the hashMap not shown
    Iterator list = anIndex.values().iterator();
    while ( list.hasNext() )
        Console.println( list.next() );
Smalltalk Examples

Streams, do:, select:, reject:, collect:, detect:, inject:into: are iterators in Smalltalk

| sum |
sum := 0.

#( 1 7 2 3 9 3 50) do: [:each | sum := sum + each squared].
^sum

#( 1 7 2 3 9 3 50) do:
  [:partialSum :number | partialSum + number squared]

'this is an example' select: [:each | each isVowel ]

| input words word |
input := 'juliusceasar' asFilename readStream.
words := Set new.
word := OrderedCollection new.
input do:
  [:each |
    (each isSeparator | ('.,;!-[])' includes: each))
    ifTrue:
      [words add:(( String withAll: word) asLowercase.
        word := OrderedCollection new.]}
    ifFalse:[ word add: each]].
input close.
words size
void print(ArrayList list)
{
    for( int k = 0; k < list.size(); k++ )
        System.out.println( list.get(k) );
}

void print(LinkedList list)
{
    Node current = list.first();
    System.out.println( current );
    while (current.hasNext() )
    {
        current = current.next();
        System.out.println( current );
    }
}

void print(Collection list)
{
    Iterator items = list.iterator();
    while (items.hasNext() )
    {
        System.out.println( items.next() );
    }
}

print: aCollection
    aCollection do:
        [:each |
            Transcript
                show: each;
            cr]
What's The Big Deal?

Iterators abstract out underlying representation of collection

Programmer does not have to know implementation details of each type of collection

Can write code that works for wide range of collects

Do not have to change code if change the type of collection used
Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

Declare variables to be instances of the abstract class not instances of particular classes

**Benefits of programming to an interface**

Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects. Clients only know about the abstract classes (or interfaces) that define the interface.
Programming to an Interface
Java Collections

Collection
- Set
  - SortedSet
    - TreeSet
  - HashSet
- List
  - ArrayList
  - LinkedList
  - Vector
- Map
  - HashMap
  - Hashtable
  - SortedMap
    - WeakHashMap
    - TreeMap

Class
- Interface
  - Implements
  - Extends
Java Iterators & Arrays?

Arrays are common collections

How can one get an iterator on a Java array?

How would you pass an array to the following function?

```java
void print(Collection list )
{
    Iterator items = list.iterator();
    while (items.hasNext() )
    {
        System.out.println( items.next() );
    }
}
```
Java Iterators in Practice

void printA(Collection list )
{
    Iterator items = list.iterator();
    while (items.hasNext() )
        System.out.println( items.next() );
}

void printB(String[] list)
{
    for (int k = 0; k < list.length; k++)
        System.out.println( list[k] );
}

Programmers are not used to programming to an interface

printA requires as much typing as printB
Smalltalk Iterators in Practice

printA: aCollection
  1 to: aCollection size do: [:index |
    Transcript
    show: (aCollection at: index);
    cr.

printB: aCollection
  aCollection do: [:each |
    Transcript
    show: each;
    cr.

printB requires less typing

Programmers use iterators just to avoid extra work
Sample Implementation of Java Enumerator

class VectorIterator implements Enumeration {
    Vector iteratee;
    int count;

    VectorIterator(Vector v) {
        iteratee = v;
        count = 0;
    }

    public boolean hasMoreElements() {
        return count < iteratee.elementCount;
    }

    public Object nextElement() {
        synchronized (iteratee) {
            if (count < iteratee.elementCount)
                return iteratee.elementData[count++];
        }
        throw new NoSuchElementException("VectorIterator");
    }
}

The iterator is using privileged access to Vectors fields
Issues
Concrete vs. Polymorphic Iterators
Concrete

Use Explicit Iterator Type

Reader iterator = new StringReader( "cat" );
int c;
while ( -1 != (c = iterator.read() ))
  System.out.println( (char) c);

Polymorphic

Actual type of iterator is not known

Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();
while ( list.hasNext() )
  Console.println( list.next() );

Polymorphic iterators can cause problems with memory leaks in C++ because they are on the heap!
Who Controls the iteration?
External (Active)

Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();

while ( list.hasNext() )
    Console.println( list.next() );

Iteration control code is repeated for each use of the iterator
Who Controls the iteration?
Internal (Passive)

'this is an example' select: [:each | each isVowel ]

Control code is inside the iterator

Programmer

• Does not repeat control code
• Can focus on what to do not how to do it
Who Defines the Traversal Algorithm?
Object being Iterated

Iterator can store where we are

In a Vector this could mean the index of the current item

In a tree structure it could mean a pointer to current node and stack of past nodes

BinaryTree searchTree = new BinaryTree();

// code to add items not shown

Iterator aSearch = searchTree.getIterator();
Iterator bSearch = searchTree.getIterator();
Object first = searchTree.nextElement( aSearch );
Object stillFirst = searchTree.nextElement( bSearch );

Iterator

Makes it easier to have multiple iterator algorithms on same type

On Vector class, why not have a reversedIterator which goes backwards?

In a complex structure the iterator may need access to the iteratee's implementation
How Robust is the iterator?

What happens when items are added/removed from the iteratee while an iterator exists?

Vector listOfStudents = new Vector();

// code to add students not shown

Enumeration list = listOfStudents.elements();
Iterator failFastList = listOfStudents.iterator();

listOfStudents.add( new Student( "Roger" ) );

list.hasMoreElements();
failFastList.hasNext(); //Exception thrown here
**Additional Iterator Operations**

Augmenting basic iteration operations may improve their usefulness

`previous()`  
back up one location

`add(Object item)`  
add item to the iteratee at current location

`remove()`  
remove the current item from the iteratee

`skipTo(some location, item or condition)`  
go to the location indicated

`mark()`  
mark current location for future return

**Iterators and Privileged Access**

An iterator may need privileged access to the aggregate structure for traversal
Iterators for Composites

Traversing a complex structure like a graph, tree, or composite can be difficult

An internal iterator can use recursion to keep track of where to go next

For example using a depth-first search algorithm on graph

If each element in the aggregate “knows” how to traverse to the next element and previous element, than an external iterator can be used

Null Iterator

A Null iterator for the empty aggregates can be useful