Iterator Design Pattern [Gamma et al]

Traditionally aggregate objects are identical group of type like objects. As the iterator abstraction is fundamental to an emerging methodology known as “Generic programming” which advocates the notion to separate the algorithms from data structures. The intent is to motivate component based development and reduce configuration management. Therefore this pattern provides a way to access and pass through collection without exposing its internal architecture. This pattern provides a mechanism to traverse the complex structures as well regardless of the matter how the internal representation. Iterator is commonly used in frameworks and libraries. It falls into the three categories as shown in the figure.

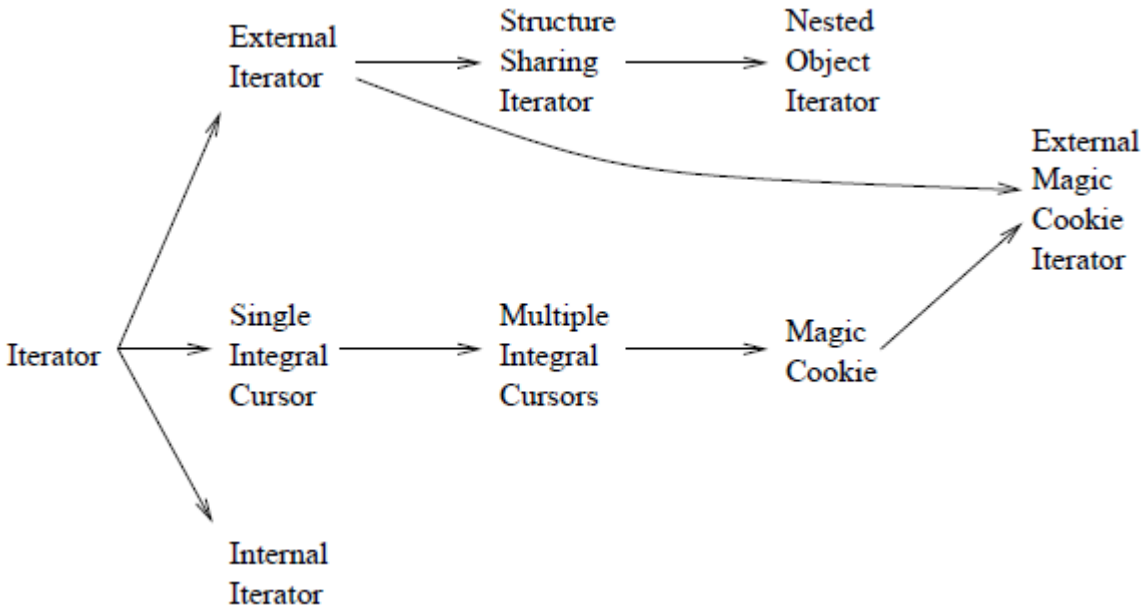
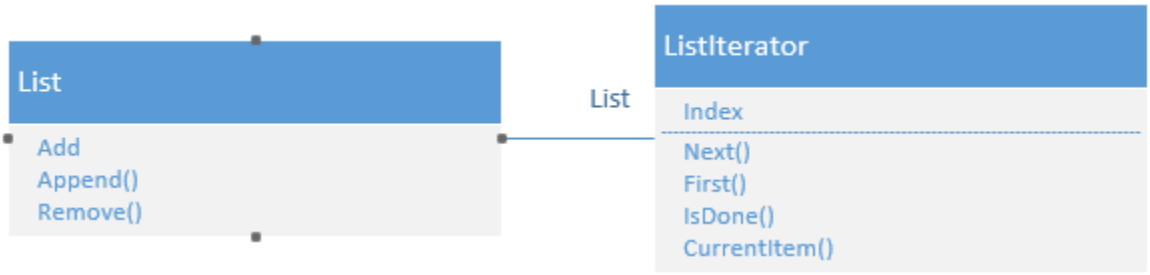


Figure 1.7 Iterator Categories

Following are the variants of Iterator design pattern:

External Iterators [60]

The aggregate object provides a simple interface for adding, removing, retrieving elements from the complex structure. Sequential access to the elements of the aggregate object is provided by external iterator. This variant is commonly used in applications because of easy to implement nature. To iterate over a collection using an external iterator, a client must create the iterator and then loop, processing elements until no more are forthcoming.




```
List list = new LinkedList();
for(Iterator i = list.iterator(); i.hasNext();)
 {System.out.println(i.next());};
```

Figure 1.8 UML and source code of External Iterator

Static Structure Iterators [60]

This variant overcomes the performance penalty of the external iterator discussed above. A structure sharing iterator provides exactly the same interface to clients as a completely external iterator. Although it must be constructed slightly differently, but to iterate over a collection. Given below is the highlighted implementation:

```
class SharingListIterator implements Iterator {
 Link ilink;

 SharingListIterator(LinkedList list) {
    this.ilink = list.link;}

 public boolean hasNext() {return ilink != null;};
 public Object next() {
    Object rv = ilink.value;
    ilink = ilink.next; return rv;};
}
```

Figure 1.9 source code of Static structure iterator

Nested Object Iterator [60]

This variant is implemented by making the iterator an instance of an inner class declared inside the aggregate's class. This will avoid aggregate object's representation from outside. Below is the implementation aspect of nested object iterator:

```
class NestedLinkedList implements List {
 Link link;
 Iterator iterator() {return new NestedListIterator();

// other methods deleted

    class NestedListIterator implements Iterator {
     Link ilink;

     NestedListIterator() {this.ilink = link;}
     public boolean hasNext() {return ilink != null;};
     public Object next() {
        Object rv = ilink.value; ilink = ilink.next;
        return rv;};
    }
}
```

Figure 1.10 Null Object Iterator

Note: <u>The difference between the two is in the way they deal with the aggregate's encapsulation | a structure sharing iterator requires that the aggregate hand out a reference to its internal state to an external object, while a nested object iterator is an internal object that explicitly offers extra public services to the aggregate object's clients.</u>

Single Integral Iterator [60]

In this variant, the iterator behavior is implemented into main aggregate class. A single integral cursor allows a single client to iterate over an aggregate, and, because the cursor is completely contained within the aggregate, the iteration can be implemented efficiently without breaching encapsulation. The main disadvantage is it supports only one traversal at a time. Below is the highlighted change:

```
interface SingleCursorList {
 Object    get(int index);
 Object    set(int index, Object element);
 // many other List methods deleted

 void      start();
 boolean   hasNext();
 Object    next();
}
```

Figure 1.11 Single Integral Iterator

Multiple Integral Iterator [60]

The single cursor design can be extended to support multiple simultaneous iterations by storing more than one cursor inside the aggregate. Other than supporting multiple simultaneous traversals, a multiple integral cursor iterator has similar advantages and disadvantages to a single integral cursor iterator in particular, it does not breach the aggregate's encapsulation. Below is the implementation aspect:

```
interface MultipleCursorList {
 // List methods deleted

 int       openCursor();            // allocates cursor
 boolean   hasNext(int cursor);
 Object    next(int cursor);
 void      closeCursor(int cursor); // releases cursor
}
```

Figure 1.12 Multiple Integral Iterator

Magic Cookie [60]

A multiple cursor iterator must store some internal state for each active cursor, and also maintains a client visible key that is used to access each cursor's state. These two objects can be unified by turning the internal cursor into a magic cookie [61]. In many senses, a magic cookie design is simply another version of a multiple integral cursor design which is efficient, supports multiple traversals, is easily made robust and safe for concurrency, but does not support the standard external iterator interface. Below is the implementation:

```
class MagicCookieList extends LinkedList {
 // List methods deleted

 Cookie    cookie() {return new Cookie();};
 boolean   hasNext(Cookie cookie) {
    return cookie.ilink != null;};
 Object    next(Cookie cookie) {
    Object rv = cookie.ilink.value;
    cookie.ilink = cookie.ilink.next; return rv; };
 class Cookie {
     private Link ilink;
     Cookie() {ilink = link;};
 }
}
```

Figure 1.13 Magic Cookie variant

External Magic Cookie Iterator [60]

This variant is an attempt to remove the drawback of the magic cookie iterator shown above to implement the iterator interface. Traditional magic cookie iterator doesn't support interface of basic external iterator. The implementation is highlighted as follows:

```
class CookieIterator extends Cookie implements Iterator {
 public boolean hasNext() {
   return MagicCookieList.this.hasNext(this);};
 public Object next() {
   return MagicCookieList.this.next(this);};
}
```

Figure 1.14 External Magic Iterator

Internal Iterator [60]

This variant manages the flow and control during iteration. The client provides a specific operation to execute an element, and the iterator applies that operation to each element in the aggregate object. An internal iterator is not a refined object, the client manages the control flow and explicitly manages individual elements. Internal iterators are thread safe in nature. Below is the source code demonstration:

```
class LinkedListInternal extends LinkedList {
  public void run(Block b) {
     Link l = link;
     while (l != null) {b.value(l.value); l = l.next; };
  }
}
```

Client Request

```
LinkedListInternal list = new LinkedListInternal();
list.run(new Block() {
  public void value(Object o) {
    System.out.println(o);};});
```

Figure 1.15 Internal Iterator

Super Iterator [62]

The Super Iterator pattern, like the standard Iterator pattern, traverses an unknown data structure without exposing that structure. With the standard Iterator pattern, clients must create a different iterator for each new structure, and the object returned must be of the specific type stored in the structure, even when they share a common super class. With the Super Iterator pattern, the object

returned is of the common super class, and the iterator itself need not be altered when adding a new subtype with custom data structures. The client, however, must change two lines of code to load and instantiate the new subclass. Below is the UML Diagram:

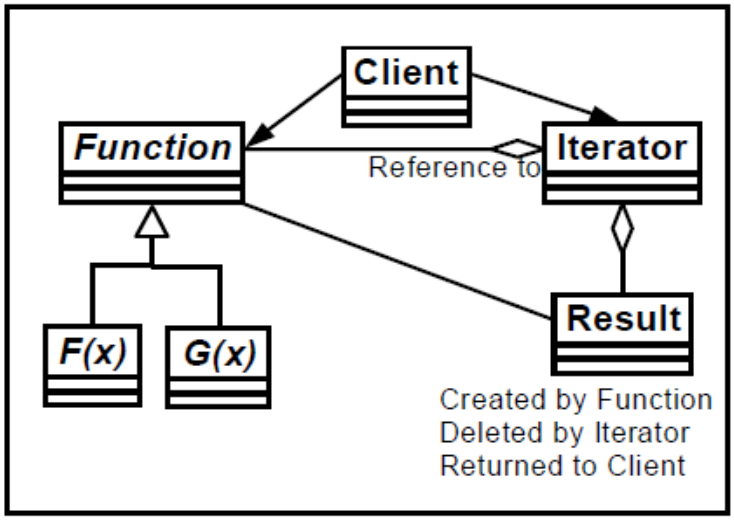


Figure 1.16 UML Diagram of Super Iterator