ITERATOR

Object Behavioral

Intent

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

Also Known As

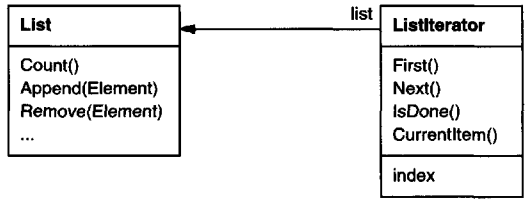
Cursor

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an **iterator** object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with the following relationship between them:



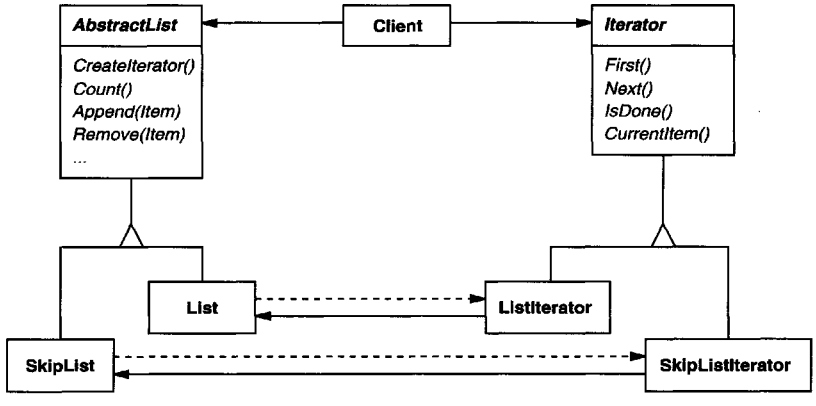
Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially. The CurrentItem operation returns the current element in the list, First initializes the current element to the first element, Next advances the current element to the next element, and IsDone tests whether we've advanced beyond the last element—that is, we're finished with the traversal.

Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface. For example, FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints.

Notice that the iterator and the list are coupled, and the client must know that it is a *list* that's traversed as opposed to some other aggregate structure. Hence the client commits to a particular aggregate structure. It would be better if we could change the aggregate class without changing client code. We can do this by generalizing the iterator concept to support **polymorphic iteration**.

As an example, let's assume that we also have a SkipList implementation of a list. A skiplist [Pug90] is a probabilistic data structure with characteristics similar to balanced trees. We want to be able to write code that works for both List and SkipList objects.

We define an AbstractList class that provides a common interface for manipulating lists. Similarly, we need an abstract Iterator class that defines a common iteration interface. Then we can define concrete Iterator subclasses for the different list implementations. As a result, the iteration mechanism becomes independent of concrete aggregate classes.



The remaining problem is how to create the iterator. Since we want to write code that's independent of the concrete List subclasses, we cannot simply instantiate a specific class. Instead, we make the list objects responsible for creating their corresponding iterator. This requires an operation like CreateIterator through which clients request an iterator object.

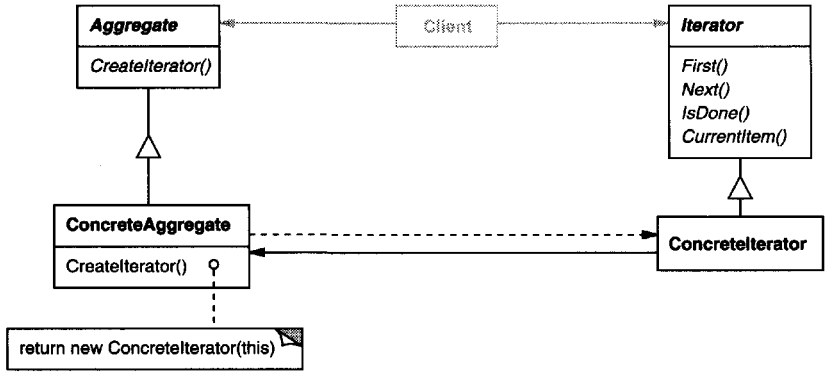
CreateIterator is an example of a factory method (see Factory Method (107)). We use it here to let a client ask a list object for the appropriate iterator. The Factory Method approach give rise to two class hierarchies, one for lists and another for iterators. The CreateIterator factory method "connects" the two hierarchies.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure



Participants

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

The Iterator pattern has three important consequences:

1. *It supports variations in the traversal of an aggregate.* Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.
2. *Iterators simplify the Aggregate interface.* Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. *More than one traversal can be pending on an aggregate.* An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation

Iterator has many implementation variants and alternatives. Some important ones follow. The trade-offs often depend on the control structures your language provides. Some languages (CLU [LG86], for example) even support this pattern directly.

1. *Who controls the iteration?* A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**.[2] Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.

   External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like C++ that does not provide anonymous functions, closures, or continuations like Smalltalk and CLOS. But on the other hand,

[2] Booch refers to external and internal iterators as **active** and **passive** iterators, respectively [Boo94]. The terms "active" and "passive" describe the role of the client, not the level of activity in the iterator.

internal iterators are easier to use, because they define the iteration logic for you.

2. *Who defines the traversal algorithm?* The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.[3]

   If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

3. *How robust is the iterator?* It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might end up accessing an element twice or missing it completely. A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.

   A **robust iterator** ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. There are many ways to implement robust iterators. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.

   Kofler provides a good discussion of how robust iterators are implemented in ET++ [Kof93]. Murray discusses the implementation of robust iterators for the USL StandardComponents' List class [Mur93].

4. *Additional Iterator operations.* The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem.[4] Some additional operations might prove useful. For example, ordered aggregates can have a Previous operation that positions the iterator to the previous element. A SkipTo operation is useful for sorted or indexed collections. SkipTo positions the iterator to an object matching specific criteria.

5. *Using polymorphic iterators in C++.* Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack.

---

[3] Cursors are a simple example of the Memento (283) pattern and share many of its implementation issues.

[4] We can make this interface even *smaller* by merging Next, IsDone, and CurrentItem into a single operation that advances to the next object and returns it. If the traversal is finished, then this operation returns a special value (0, for instance) that marks the end of the iteration.

Polymorphic iterators have another drawback: the client is responsible for deleting them. This is error-prone, because it's easy to forget to free a heap-allocated iterator object when you're finished with it. That's especially likely when there are multiple exit points in an operation. And if an exception is triggered, the iterator object will never be freed.

The Proxy (207) pattern provides a remedy. We can use a stack-allocated proxy as a stand-in for the real iterator. The proxy deletes the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions. This is an application of the well-known C++ technique "resource allocation is initialization" [ES90]. The Sample Code gives an example.

6. *Iterators may have privileged access.* An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. We can express this close relationship in C++ by making the iterator a `friend` of its aggregate. Then you don't need to define aggregate operations whose sole purpose is to let iterators implement traversal efficiently.

   However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid this problem, the Iterator class can include `protected` operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and *only* Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.

7. *Iterators for composites.* External iterators can be difficult to implement over recursive aggregate structures like those in the Composite (163) pattern, because a position in the structure may span many levels of nested aggregates. Therefore an external iterator has to store a path through the Composite to keep track of the current object. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.

   If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and children, then a cursor-based iterator may offer a better alternative. The cursor only needs to keep track of the current node; it can rely on the node interface to traverse the Composite.

   Composites often need to be traversed in more than one way. Preorder, postorder, inorder, and breadth-first traversals are common. You can support each kind of traversal with a different class of iterator.

8. *Null iterators.* A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal; that is, its IsDone operation always evaluates to true.

   NullIterator can make traversing tree-structured aggregates (like Composites) easier. At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator

as usual. But leaf elements return an instance of NullIterator. That lets us implement traversal over the entire structure in a uniform way.

Sample Code

We'll look at the implementation of a simple List class, which is part of our foundation library (Appendix C). We'll show two Iterator implementations, one for traversing the List in front-to-back order, and another for traversing back-to-front (the foundation library supports only the first one). Then we show how to use these iterators and how to avoid committing to a particular implementation. After that, we change the design to make sure iterators get deleted properly. The last example illustrates an internal iterator and compares it to its external counterpart.

1. *List and Iterator interfaces.* First let's look at the part of the List interface that's relevant to implementing iterators. Refer to Appendix C for the full interface.

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

The `List` class provides a reasonably efficient way to support iteration through its public interface. It's sufficient to implement both traversals. So there's no need to give iterators privileged access to the underlying data structure; that is, the iterator classes are not friends of `List`. To enable transparent use of the different traversals we define an abstract `Iterator` class, which defines the iterator interface.

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

2. *Iterator subclass implementations.* `ListIterator` is a subclass of `Iterator`.

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

The implementation of `ListIterator` is straightforward. It stores the `List` along with an index `_current` into the list:

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

`First` positions the iterator to the first element:

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

`Next` advances the current element:

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

`IsDone` checks whether the index refers to an element within the List:

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

Finally, `CurrentItem` returns the item at the current index. If the iteration has already terminated, then we throw an `IteratorOutOfBounds` exception:

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

The implementation of ReverseListIterator is identical, except its `First` operation positions `_current` to the end of the list, and `Next` decrements `_current` toward the first item.

3. *Using the iterators.* Let's assume we have a `List` of `Employee` objects, and we would like to print all the contained employees. The `Employee` class supports this with a `Print` operation. To print the list, we define a `PrintEmployees` operation that takes an iterator as an argument. It uses the iterator to traverse and print the list.

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

Since we have iterators for both back-to-front and front-to-back traversals, we can reuse this operation to print the employees in both orders.

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

4. *Avoiding commitment to a specific list implementation.* Let's consider how a skiplist variation of `List` would affect our iteration code. A `SkipList` subclass of `List` must provide a `SkipListIterator` that implements the `Iterator` interface. Internally, the `SkipListIterator` has to keep more than just an index to do the iteration efficiently. But since `SkipListIterator` conforms to the `Iterator` interface, the `PrintEmployees` operation can also be used when the employees are stored in a `SkipList` object.

```
SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

Although this approach works, it would be better if we didn't have to commit to a specific `List` implementation, namely `SkipList`. We can introduce an `AbstractList` class to standardize the list interface for different list implementations. `List` and `SkipList` become subclasses of `AbstractList`.

To enable polymorphic iteration, `AbstractList` defines a factory method `CreateIterator`, which subclasses override to return their corresponding iterator:

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

An alternative would be to define a general mixin class `Traversable` that defines the interface for creating an iterator. Aggregate classes can mix in `Traversable` to support polymorphic iteration.

`List` overrides `CreateIterator` to return a `ListIterator` object:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Now we're in a position to write the code for printing the employees independent of a concrete representation.

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

5. *Making sure iterators get deleted.* Notice that `CreateIterator` returns a newly allocated iterator object. We're responsible for deleting it. If we forget, then we've created a storage leak. To make life easier for clients, we'll provide an `IteratorPtr` that acts as a proxy for an iterator. It takes care of cleaning up the `Iterator` object when it goes out of scope.

   `IteratorPtr` is always allocated on the stack.[5] C++ automatically takes care of calling its destructor, which deletes the real iterator. `IteratorPtr` overloads both `operator->` and `operator*` in such a way that an `IteratorPtr` can be treated just like a pointer to an iterator. The members of `IteratorPtr` are all implemented inline; thus they can incur no overhead.

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }
```

[5] You can ensure this at compile-time just by declaring private `new` and `delete` operators. An accompanying implementation isn't needed.

```
    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }
private:
    // disallow copy and assignment to avoid
    // multiple deletions of _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};
```

`IteratorPtr` lets us simplify our printing code:

```
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

6. *An internal ListIterator.* As a final example, let's look at a possible implementation of an internal or passive `ListIterator` class. Here the iterator controls the iteration, and it applies an operation to each element.

   The issue in this case is how to parameterize the iterator with the operation we want to perform on each element. C++ does not support anonymous functions or closures that other languages provide for this task. There are at least two options: (1) Pass in a pointer to a function (global or static), or (2) rely on subclassing. In the first case, the iterator calls the operation passed to it at each point in the iteration. In the second case, the iterator calls an operation that a subclass overrides to enact specific behavior.

   Neither option is perfect. Often you want to accumulate state during the iteration, and functions aren't well-suited to that; we would have to use static variables to remember the state. An `Iterator` subclass provides us with a convenient place to store the accumulated state, like in an instance variable. But creating a subclass for every different traversal is more work.

   Here's a sketch of the second option, which uses subclassing. We call the internal iterator a `ListTraverser`.

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

   `ListTraverser` takes a `List` instance as a parameter. Internally it uses an external `ListIterator` to do the traversal. `Traverse` starts the traversal

and calls `ProcessItem` for each item. The internal iterator can choose to terminate a traversal by returning `false` from `ProcessItem`. `Traverse` returns whether the traversal terminated prematurely.

```
template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```

Let's use a `ListTraverser` to print the first 10 employees from our employee list. To do it we have to subclass `ListTraverser` and override `ProcessItem`. We count the number of printed employees in a `_count` instance variable.

```
class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }

protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}
```

Here's how `PrintNEmployees` prints the first 10 employees on the list:

```
List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Note how the client doesn't specify the iteration loop. The entire iteration logic can be reused. This is the primary benefit of an internal iterator. It's a bit more work than an external iterator, though, because we have to define a new class. Contrast this with using an external iterator:

```
ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
    count++;
    i.CurrentItem()->Print();

    if (count >= 10) {
        break;
    }
}
```

Internal iterators can encapsulate different kinds of iteration. For example, `FilteringListTraverser` encapsulates an iteration that processes only items that satisfy a test:

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

This interface is the same as `ListTraverser`'s except for an added `TestItem` member function that defines the test. Subclasses override `TestItem` to specify the test.

`Traverse` decides to continue the traversal based on the outcome of the test:

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
```

```
                if (result == false) {
                    break;
                }
            }
        }
        return result;
    }
```

A variant of this class could define `Traverse` to return if at least one item satisfies the test.[6]

Known Uses

Iterators are common in object-oriented systems. Most collection class libraries offer iterators in one form or another.

Here's an example from the Booch components [Boo94], a popular collection class library. It provides both a fixed size (bounded) and dynamically growing (unbounded) implementation of a queue. The queue interface is defined by an abstract Queue class. To support polymorphic iteration over the different queue implementations, the queue iterator is implemented in the terms of the abstract Queue class interface. This variation has the advantage that you don't need a factory method to ask the queue implementations for their appropriate iterator. However, it requires the interface of the abstract Queue class to be powerful enough to implement the iterator efficiently.

Iterators don't have to be defined as explicitly in Smalltalk. The standard collection classes (Bag, Set, Dictionary, OrderedCollection, String, etc.) define an internal iterator method `do:`, which takes a block (i.e., closure) as an argument. Each element in the collection is bound to the local variable in the block; then the block is executed. Smalltalk also includes a set of Stream classes that support an iterator-like interface. ReadStream is essentially an Iterator, and it can act as an external iterator for all the sequential collections. There are no standard external iterators for nonsequential collections such as Set and Dictionary.

Polymorphic iterators and the cleanup Proxy described earlier are provided by the ET++ container classes [WGM88]. The Unidraw graphical editing framework classes use cursor-based iterators [VL90].

ObjectWindows 2.0 [Bor94] provides a class hierarchy of iterators for containers. You can iterate over different container types in the same way. The ObjectWindow iteration syntax relies on overloading the postincrement operator `++` to advance the iteration.

Related Patterns

Composite (163): Iterators are often applied to recursive structures such as Composites.

[6] The `Traverse` operation in these examples is a Template Method (325) with primitive operations `TestItem` and `ProcessItem`.

Factory Method (107): Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.

Memento (283) is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.