Design Patterns Study Group

Iterator Pattern

Fred Stluka

April 30, 1998

Name

- Iterator
- AKA: Cursor

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- An object <u>behavioral</u> pattern

Motivation -- Approach 1: Direct access, no encapsulation

- Client code for Array:

```
for (I = 0; I < MAX; I++) {
    ProcessItem (arr[I]);
}
```

- Client code for Linked List:

```
p = pList;
while (p) {
    ProcessItem (*p);
    p = p->Next;
}
```

- Client code for Binary Tree:

```
[more complicated, recursive algorithm]
```

Motivation -- Approach 1: Direct access, no encapsulation

- Pro: Simple, familiar, easy to understand
- Con: No encapsulation of data structure to prevent corruption
- Con: Different client code for different data structures
- Con: Can’t change data structure without re-coding client

Motivation -- Approach 2: Iteration methods on Aggregate

- Aggregate class

```
class List {
        ...
        void        First();
        void        Next();
        bool        IsDone();
        Item        CurrentItem();
        void        AddItem(Item i);
        void        RemoveItem();
        Item        FindItem(char* pName);
}
```

Motivation -- Approach 2: Iteration methods on Aggregate

- Client code (for list, array, tree, …)

```
pList->First();
while (!pList->IsDone()) {
        ProcessItem (pList->CurrentItem());
        pList->Next();
}
```

Motivation -- Approach 2: Iteration methods on Aggregate

- Pro: (All pros from previous approach)
- Pro: Encapsulation of data structure
- Pro: Same client code for all data structures (list, array, tree, ...)

Motivation -- Approach 2: Iteration methods on Aggregate

- Con: No multiple concurrent traversals
  - Searching for duplicates, etc.
- Con: No multiple types of traversal
  - backward, forward, preorder, postorder, inorder
- Con: Traversal algorithm not reusable
- Con: Iteration methods intermixed with other methods

Motivation -- Approach 3: Separate Iterator

- Aggregate class

```
class List { ...
        int       Count();
        Item      Get(int pos);
        void      AddItem(Item i, int pos);
        void      RemoveItem(int pos);
        Item&     FindItem(char* pName); ... }
```

- Iterator class

```
class Iterator { ...
                  Iterator(List* list);
        void      First();
        void      Next();
        bool      IsDone();
        Item      CurrentItem(); ... }
```

Motivation -- Approach 3: Separate Iterator

- Client code (for list, array, tree, …)

```
Iterator i(pList);
i->First();
while (!i->IsDone()) {
        ProcessItem (i->CurrentItem());
        i->Next();
}
```

Motivation -- Approach 3: Separate Iterator

- Pro: (All pros from previous approach)
- Pro: Multiple concurrent traversals via multiple instances of iterator
- Pro: Multiple types of traversal via multiple iterator classes
- Pro: Traversal algorithm reusable
- Pro: Iteration methods factored out

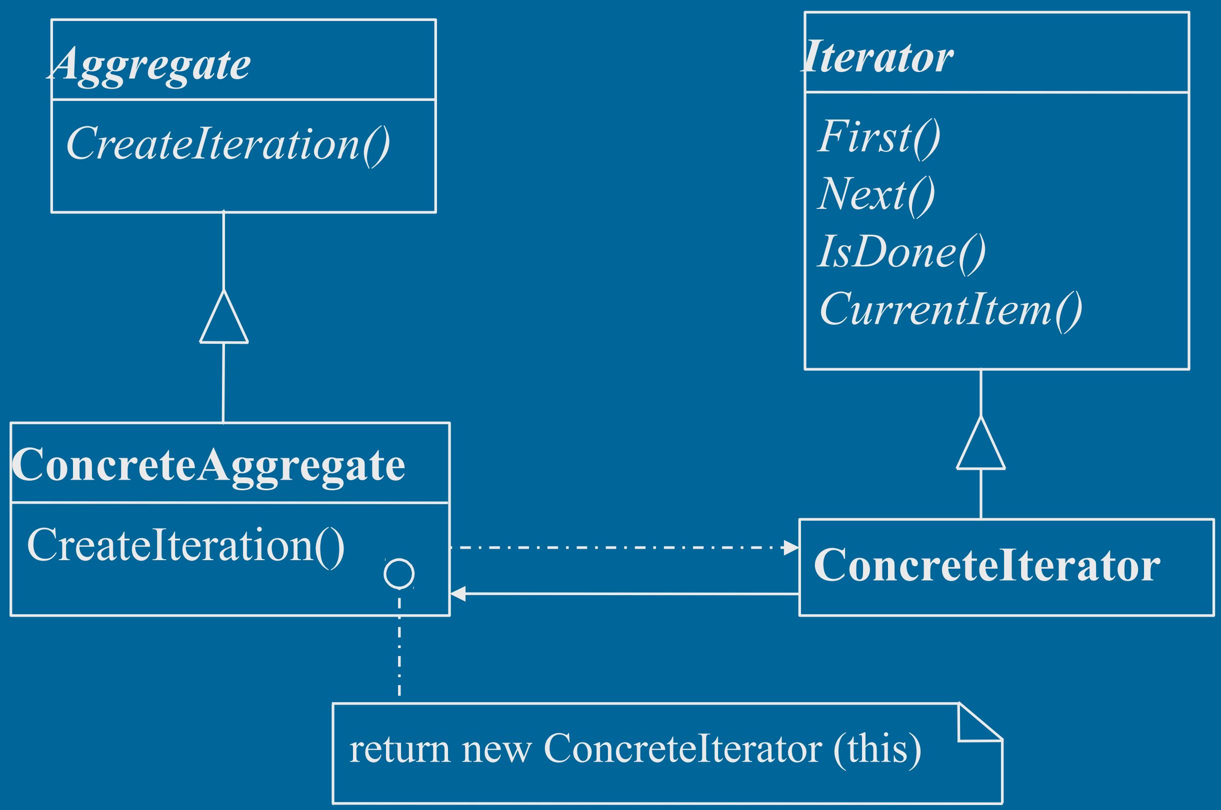
Motivation -- Approach 3: Separate Iterator

- Con: Iterator needs access to items
  - Get, Count
- Con: Need way to associate Iterator with Aggregate
  - Parameter to Iterator constructor
- Con: How to efficiently store position?
  - Pos parameter to Get, AddItem, RemoveItem, ...
  - Especially recursive Aggregates

Applicability

- Access to contents of black-box aggregate
- Polymorphic iteration
  - Same interface for list, tree, ...
- Multiple traversals
  - Nested or concurrent
  - Forward, reverse, preorder, inorder, postorder
- Complex traversal algorithm
  - Reuse the algorithm on multiple data structures

Structure


Aggregate
CreateIteration()
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIteration()
ConcreteIterator
return new ConcreteIterator (this)

Participants

- Iterator
  - Defines interface for accessing and traversing elements
- ConcreteIterator
  - Maintains position and determines next element
- Aggregate
  - Defines interface for creating Iterator
- ConcreteAggregate
  - Creates appropriate ConcreteIterator

Collaborations

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

Consequences

- Separation of data structure from traversal
- Multiple concurrent traversals
  - Current position recorded in each iterator, not in the aggregate
- Multiple traversal orders
- Traversal algorithm reusable
- Simplifies interface of Aggregate
  - Moves First(), Next(), IsDone() etc. to Iterator class

Implementation: Internal ("Passive") Iterators

- Previous discussion covers "external" ("active") iterators
- "Internal" ("passive") Iterator class

```
typedef bool (*FUNCPTR)(Item);
class Iterator { ...
        Iterator(List* list);
    bool   Traverse(FUNCPTR fp);
    ... }
```

- Client code (for list, array, tree, …)

```
Iterator i(pList);
i->Traverse(ProcessItem);
```

Implementation: Internal (“Passive”) Iterators

- Pro: Simpler to use, no risk of infinite loop
- Pro: Manages complex position well

Implementation: Internal ("Passive") Iterators

- Con: Hides complex position from client
- Con: Less flexible (like "for" loop)
- Con: No synchronized traversals (MergeSort)
- Con: Info accumulated during traversal must be stored globally or statically (or passed as Iterator parameter)
- See also: http://sw-eng.falls-church.va.us/AdaIC/docs/style-guide/83style/style-t.txt

Implementation: Modifications During Iteration

- Items added during iteration
  - Mathematical “closure” algorithm relies on hitting added items later.
  - Other algorithms rely on not hitting them.
  - Prioritized list relies on hitting high priority added items immediately, and low priority added items later.
- Items deleted during iteration
  - Common mistake is to iterate list, deleting items.
  - Don’t allow this to crash your iterator.
- See also: http://sw-eng.falls-church.va.us/AdaIC/docs/style-guide/83style/style-t.txt

Implementation: Polymorphic Iterators

- Polymorphic Iterators are heap-based (dynamically allocated by CreateIterator and passed to client).
- Memory leak if client fails to deallocate.
- Use Proxy pattern to do deallocation in destructor of stack-based proxy class.

Implementation: Privileged access

- How does Iterator access items in Aggregate without making such access available to all clients?
- "Friend" access in C++ requires knowledge of all Iterators by Aggregate.
- "Protected" access in C++ requires Iterator to be a subclass of Aggregate.

Implementation:
Full “iterator” vs. mere “cursor”

- Previous discussion has been on iterators
- “Cursors” are lightweight iterators that record the current position but not the algorithm for getting to the next item. The Aggregate does that part.
- This dodges the problem of privileged access

Implementation: Recursive aggregates

- How to efficiently maintain position in a recursive aggregate like a tree? Can't keep pointer into guts of data structure without special access. Can't use a simple index without forcing Aggregate to re-traverse to the right node at each iteration.

Implementation: Associating Iterator & Aggregate

- How to associate the Iterator with the Aggregate?
  - Aggregate creates Iterator of the right type and passes itself as a parameter to the constructor.
    - Con: Aggregate must know all Iterator types.
  - Client creates both and passes one to the other.
    - Con: Client must know appropriate pairs.

Known Uses

- Booch components, 1987 (active/passive)
- VB “For Each”, Form_Unload (passive)
- C++ STL
- Smalltalk collection classes
- Windows “RegEnumKey” API (active)
- Windows “EnumWindows” API (passive)
- All black-box aggregates

Related Patterns

- Composite
  - Used to implement recursive Aggregates
- Factory Method
  - Used in Aggregate to create Iterator
- Memento
  - Used in Iterator to store position

Questions

- Example of PreOrderIterator on pg 68?

Design Patterns Study Group

Iterator Pattern

Fred Stluka
April 30, 1998

Name

- Iterator
- AKA: Cursor

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- An object behavioral pattern

Motivation -- Approach 1: Direct access, no encapsulation

- Client code for Array:

```
for (I = 0; I < MAX; I++) {
    ProcessItem (arr[I]);
}
```

- Client code for Linked List:

```
p = pList;
while (p) {
    ProcessItem (*p);
    p = p->Next;
}
```

- Client code for Binary Tree:

```
[more complicated, recursive algorithm]
```

Motivation -- Approach 1: Direct access, no encapsulation

- Pro: Simple, familiar, easy to understand
- Con: No encapsulation of data structure to prevent corruption
- Con: Different client code for different data structures
- Con: Can’t change data structure without re-coding client

Motivation -- Approach 2: Iteration methods on Aggregate

- Aggregate class

```
class List {
        ...
        void        First();
        void        Next();
        bool        IsDone();
        Item        CurrentItem();
        void        AddItem(Item i);
        void        RemoveItem();
        Item        FindItem(char* pName);
}
```

Motivation -- Approach 2: Iteration methods on Aggregate

- Client code (for list, array, tree, …)

```
pList->First();
while (!pList->IsDone()) {
        ProcessItem (pList->CurrentItem());
        pList->Next();
}
```

Motivation -- Approach 2: Iteration methods on Aggregate

- Pro: (All pros from previous approach)
- Pro: Encapsulation of data structure
- Pro: Same client code for all data structures (list, array, tree, ...)

Motivation -- Approach 2: Iteration methods on Aggregate

- Con: No multiple concurrent traversals
  - Searching for duplicates, etc.
- Con: No multiple types of traversal
  - backward, forward, preorder, postorder, inorder
- Con: Traversal algorithm not reusable
- Con: Iteration methods intermixed with other methods

Motivation -- Approach 3: Separate Iterator

- Aggregate class

```
class List { ...
        int         Count();
        Item        Get(int pos);
        void        AddItem(Item i, int pos);
        void        RemoveItem(int pos);
        Item&       FindItem(char* pName); ... }
```

- Iterator class

```
class Iterator { ...
                    Iterator(List* list);
        void        First();
        void        Next();
        bool        IsDone();
        Item        CurrentItem(); ... }
```

Motivation -- Approach 3: Separate Iterator

- Client code (for list, array, tree, …)

```
Iterator i(pList);
i->First();
while (!i->IsDone()) {
        ProcessItem (i->CurrentItem());
        i->Next();
}
```

Motivation -- Approach 3: Separate Iterator

- Pro: (All pros from previous approach)
- Pro: Multiple concurrent traversals via multiple instances of iterator
- Pro: Multiple types of traversal via multiple iterator classes
- Pro: Traversal algorithm reusable
- Pro: Iteration methods factored out

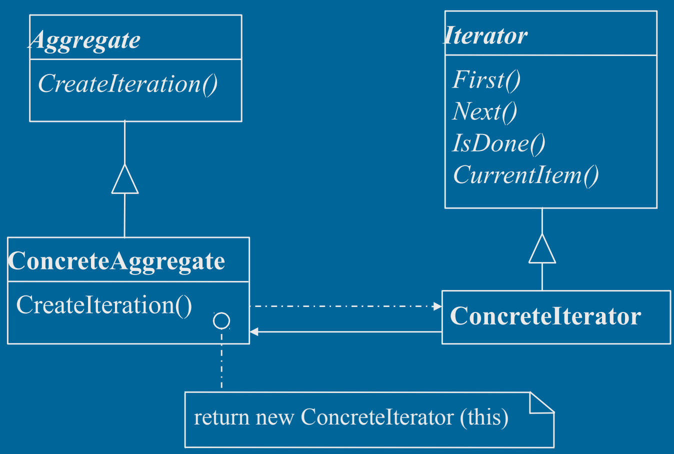
Motivation -- Approach 3: Separate Iterator

- Con: Iterator needs access to items
  - Get, Count
- Con: Need way to associate Iterator with Aggregate
  - Parameter to Iterator constructor
- Con: How to efficiently store position?
  - Pos parameter to Get, AddItem, RemoveItem, ...
  - Especially recursive Aggregates

Applicability

- Access to contents of black-box aggregate
- Polymorphic iteration
  - Same interface for list, tree, ...
- Multiple traversals
  - Nested or concurrent
  - Forward, reverse, preorder, inorder, postorder
- Complex traversal algorithm
  - Reuse the algorithm on multiple data structures

Structure



Aggregate
CreateIteration()
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIteration()
ConcreteIterator
return new ConcreteIterator (this)

Participants

- Iterator
  - Defines interface for accessing and traversing elements
- ConcreteIterator
  - Maintains position and determines next element
- Aggregate
  - Defines interface for creating Iterator
- ConcreteAggregate
  - Creates appropriate ConcreteIterator

Collaborations

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

Consequences

- Separation of data structure from traversal
- Multiple concurrent traversals
  - Current position recorded in each iterator, not in the aggregate
- Multiple traversal orders
- Traversal algorithm reusable
- Simplifies interface of Aggregate
  - Moves First(), Next(), IsDone() etc. to Iterator class

Implementation: Internal (“Passive”) Iterators

- Previous discussion covers “external” (“active”) iterators
- “Internal” (“passive”) Iterator class

```
typedef bool (*FUNCPTR)(Item);
class Iterator { ...
        Iterator(List* list);
    bool   Traverse(FUNCPTR fp);
    ... }
```

- Client code (for list, array, tree, …)

```
Iterator i(pList);
i->Traverse(ProcessItem);
```

Implementation: Internal (“Passive”) Iterators

- Pro: Simpler to use, no risk of infinite loop
- Pro: Manages complex position well

Implementation: Internal (“Passive”) Iterators

- Con: Hides complex position from client
- Con: Less flexible (like “for” loop)
- Con: No synchronized traversals (MergeSort)
- Con: Info accumulated during traversal must be stored globally or statically (or passed as Iterator parameter)
- See also: http://sw-eng.falls-church.va.us/AdaIC/docs/style-guide/83style/style-t.txt

Implementation: Modifications During Iteration

- Items added during iteration
  - Mathematical "closure" algorithm relies on hitting added items later.
  - Other algorithms rely on <u>not</u> hitting them.
  - Prioritized list relies on hitting high priority added items immediately, and low priority added items later.
- Items deleted during iteration
  - Common mistake is to iterate list, deleting items.
  - Don't allow this to crash your iterator.
- See also: http://sw-eng.falls-church.va.us/AdaIC/docs/style-guide/83style/style-t.txt

Implementation: Polymorphic Iterators

- Polymorphic Iterators are heap-based (dynamically allocated by CreateIterator and passed to client).
- Memory leak if client fails to deallocate.
- Use Proxy pattern to do deallocation in destructor of stack-based proxy class.

Implementation: Privileged access

- How does Iterator access items in Aggregate without making such access available to all clients?
- “Friend” access in C++ requires knowledge of all Iterators by Aggregate.
- “Protected” access in C++ requires Iterator to be a subclass of Aggregate.

Implementation: Full “iterator” vs. mere “cursor”

- Previous discussion has been on iterators
- “Cursors” are lightweight iterators that record the current position but not the algorithm for getting to the next item. The Aggregate does that part.
- This dodges the problem of privileged access

Implementation: Recursive aggregates

- How to efficiently maintain position in a recursive aggregate like a tree? Can't keep pointer into guts of data structure without special access. Can't use a simple index without forcing Aggregate to re-traverse to the right node at each iteration.

Implementation: Associating Iterator & Aggregate

- How to associate the Iterator with the Aggregate?
  - Aggregate creates Iterator of the right type and passes itself as a parameter to the constructor.
    - Con: Aggregate must know all Iterator types.
  - Client creates both and passes one to the other.
    - Con: Client must know appropriate pairs.

Known Uses

- Booch components, 1987 (active/passive)
- VB “For Each”, Form_Unload (passive)
- C++ STL
- Smalltalk collection classes
- Windows “RegEnumKey” API (active)
- Windows “EnumWindows” API (passive)
- All black-box aggregates

Related Patterns

- Composite
  - Used to implement recursive Aggregates
- Factory Method
  - Used in Aggregate to create Iterator
- Memento
  - Used in Iterator to store position

Questions

- Example of PreOrderIterator on pg 68?