CSCI 253

Object Oriented Design:
Iterator Pattern
George Blankenship

Overview

Creational Patterns
- 🖫 Singleton
- 🗐 Abstract factory
- 🗐 Factory Method
- 🗐 Prototype
- 🗐 Builder

Structural Patterns
- 🗐 Composite
- 🗐 Façade
- 🗐 Proxy
- 🗐 Flyweight
- 🖫 Adapter
- 🖫 Bridge
- 🖫 Decorator

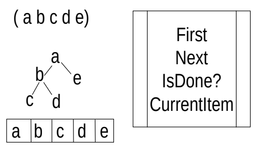
Behavioral Patterns
- 🖫 Chain of Respons.
- 🖫 Command
- 🖫 Interpreter
- 🗐 Iterator
- 🖫 Mediator
- 🖫 Memento
- 🗐 Observer
- 🗐 State
- 🗐 Strategy
- 🖫 Template Method
- 🗐 Visitor

The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Iterator Pattern: The Problem

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



-

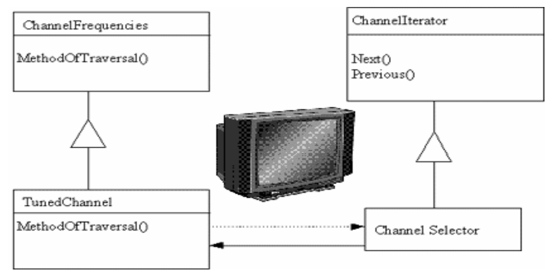
• support multiple types of traversals of aggregate objects

• provide a uniform interface for traversing aggregate structures (polymorphic iteration)

Also known as : Cursor

xxx Pattern George Blankenship 4

Channel Changer



xxx Pattern George Blankenship 5

Iterator Operation

- Add a create_iterator() method to the "collection" class, and grant the "iterator" class privileged access.
- Design an "iterator" class that can encapsulate traversal of the "collection" class.
- Clients ask the collection object to create an iterator object.
- Clients use the first(), is_done(), next(), and current_item() protocol to access the elements of the collection class.

xxx Pattern George Blankenship 6

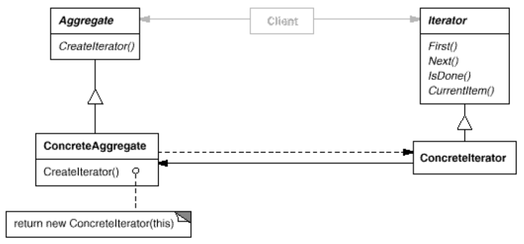
Linked List

```
class ListEntry {
 protected:
     ListEntry *Next; // next entry in the liLked  ist (orÀN   ¦)
     void *Trace; // pointer to list head if queued (or NUL)
public:
     ListEntry(void);
     ~ListEntry(void);
      void SetNextEntry(ListEntry *Entry);  // set next entry in list
      ListEntry *GetNextEntry(void);          // get next entry in list
};

class ListHead {
 protected:
      ListEntry *Head, // start of the linked list (or NUL)
                     *Tail; // end of the linked list (or NUL)
public:
       ListHead(void) {Head = Tail = NUL;}
       ~ListHead(void);
       void AddEntry(ListEntry *Entry);       // add an entry to linked list
       ListEntry *RemoveFirstEntry(void);   // remove the first entry in the list
       ListEntry *GetFirstEntry(void);         // get first entry in the l st
     };
```

xxx Pattern George Blankenship 7

The Iterator Pattern: Structure



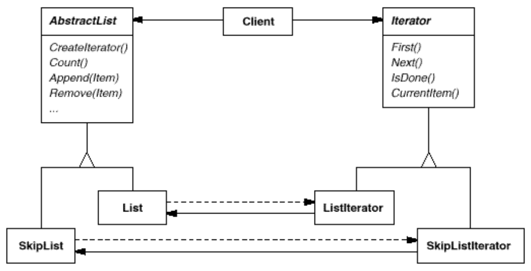
xxx Pattern George Blankenship 8

The Iterator Pattern
Participants & Collaboration

- 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 defining an Iterator object
- ConcreteAggregate
  - Implements the iterator creation interface to return an instance of the proper ConcreteIterator
- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal

xxx Pattern George Blankenship 9

The Iterator Pattern: Collaboration



xxx Pattern George Blankenship 10

The Iterator Pattern Consequences

- Supports variation in the traversal of an aggregate: complex aggregates can be traversed in many ways, iterators make it easy to change the traversal algorithm by just using a different iterator instance
- Simplifies the Aggregate interface: the Aggregate interface is not to be cluttered with various types of traversal support
- More than one traversal can be pending on the same aggregate: an iterator keeps track of its own traversal state, therefor more than one traversal can be in progress at the same time

xxx Pattern George Blankenship 11

The Iterator Pattern Implementation Control

- An internal iterator has full control over the complete iteration, the clients hands an operation to perform and the iterator applies that operation to each element in the aggregate. With an external iterator the client controls the iteration, i.e. the client advances the traversal by requesting the next element explicitly. External iterators are more flexible (compare two collections on equality is practically impossible with internal operators). Internal iterators are easier to use but are weak in languages that do not support functions as first class objects.
- The iterator can be responsible for the traversal algorithm in which case it is easy to use different iteration algorithms on the same aggregate and to use the same algorithm on different aggregates. The aggregate itself can be responsible for the traversal algorithm and the iterator just stores the state of the iteration (cursor)

xxx Pattern George Blankenship 12

The Iterator Pattern Implementation Robustness and Polymorphism

- A robust iterator ensures that insertion and removals do not interfere with traversal (and it does so without copying the aggregate). Robust iterators can for example be implemented by registering the iterators with the aggregate and make the aggregate adjust the internal state of the iterators upon insertion or deletion of elements
- Polymorphic have a cost, they require the iterator object to be allocated dynamically by a factory method. Hence if there is no need for polymorphism use concrete iterators which can be allocated on the stack. Polymorphic iterators have another drawback, they must be deleted by the client code which is error prone. The proxy pattern offers a solution here: use a stack allocated proxy for the real iterator object, make the proxy ensure proper clean up in its destructor so that when the proxy goes out of scope the iterator object is deleted

The Iterator Pattern Implementation Access and Null Entities

- An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. In they can be made friends so that the aggregate does not have to define operations with the sole purpose of making the traversal efficient. Adding new traversals becomes difficult since the aggregates interface has to change then to let in another friend.
- A Null Iterator is a degenerated iterator which is helpful for handling boundery conditions. NullIterators make traversing tree-like recursive structures (like Composites) easier. At each point in the traversal the current node is asked for the iterator for its children. An aggregate element returns a concrete iterator, a leaf element a NullIterator