Iterator Design Pattern

Overview

- the iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation

- an aggregate object is an object that contains other objects for the purpose of grouping those objects as a unit
  - also called a container or a collection
  - examples are a linked list and a hash table

- iterators are generally used to traverse a container to access its elements

- it is a very commonly used design pattern in Java with the collection framework
  - used to access the elements of a collection object

- the pattern hides the actual implementation of traversal through the collection
  - client programs just use iterator methods

Examples

- suppose there are two companies, company A and company B

- company A stores its employee records (name, etc.) in a linked list

- company B stores its employee data in a big array

- one day the two companies decide to work together
  - the iterator pattern will allow us to have a common interface through which we can access data for both companies
  - we will simply call the same methods without rewriting any code

- another example would be in a college
  - the arts department may use an array data structure
  - the science department may use a linked list data structure to store their students' records

- the main administrative department will access those data through common methods using the iterator
  - it does not care which data structure is used by individual departments

More Overview

- as mentioned, this pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers

- an iterator object is responsible for keeping track of the current element
  - it knows which elements have been traversed already

- once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with any of these aggregates

- the iterator allows different traversal methods (forwards and backwards)

- allows multiple traversals to be in progress concurrently

- places the task of traversal on the iterator object, not on the aggregate
  - simplifies the aggregate interface and implementation
  - places the responsibility where it should be
    - keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration

When to use the iterator pattern?

- when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program
  - logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily

- use the pattern to support multiple traversals of aggregate objects

- use the pattern to support polymorphic iteration

Implementation consequences

- the implementation supports variations in the traversal of an aggregate
  - 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

- more than one traversal can be pending on an aggregate
  - an iterator keeps track of its own traversal state
  - you can have more than one traversal in progress at once