COMP2511

Iterator Pattern

Prepared by

Dr. Ashesh Mahidadia

Iterator Pattern: Intent and Motivation

- The intent of the Iterator design pattern is to:

  "*Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*" [GoF]

- Exposing representation details of an aggregate breaks its encapsulation.

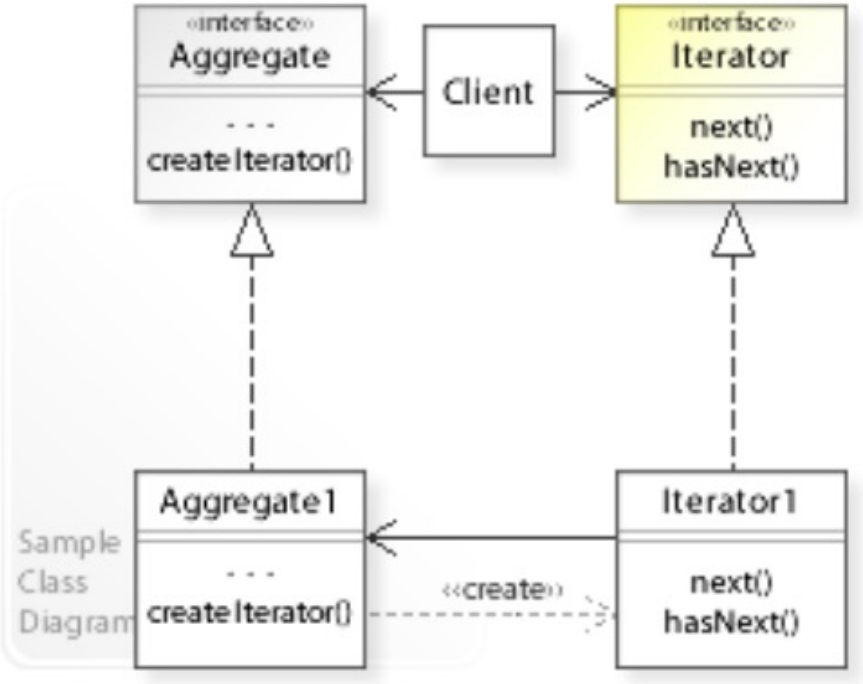
- Problem to address:

  How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?

- "But you probably don't want to bloat the List [Aggregate] interface with operations for different traversals, even if you could anticipate the ones you will need." [GoF, p257]

Iterator Pattern: Possible Solution

- Encapsulate the access and traversal of an aggregate in a separate Iterator object.
- Clients request an Iterator object from an aggregate (say by calling createIterator()) and use it to access and traverse the aggregate.
- Define an interface for accessing and traversing the elements of an aggregate object (next(), hasNext()).
- Define classes (Iterator1,...) that implement the Iterator interface.



Iterator Pattern: Possible Solution

- An iterator is usually implemented as inner class of an aggregate class. This enables the iterator to access the internal data structures of the aggregate.
- New access and traversal operations can be added by defining new iterators.
  For example, traversing back-to-front: previous(), hasPrevious().
- An aggregate provides an interface for creating an iterator (createIterator()).
- Clients can use different Iterator objects to access and traverse an aggregate object in different ways.
- Multiple traversals can be in progress on the same aggregate object (simultaneous traversals). However, need to consider concurrent usage issues!

Iterator Pattern: Java Collection Framework

The Java Collections Framework provides,

- a general purpose *iterator*

  next(), hasNext(), remove()

- an extended *listIterator*

  next(), hasNext(), previous(), hasPrevious(), remove(), ...

Example: Custom Iterator

```
Hashtable<String, MenuItem> menuItems =
        new Hashtable<String, MenuItem>();
```

```
public Iterator<MenuItem> createIterator() {
    return menuItems.values().iterator();
}
```

Using or forwarding an **iterator** method from a collection (i.e. Hashtable, ArrayList, etc.)

Implement **Iterator** interface, and provide the required methods (and more if required).

```
public class DinerMenuIterator implements Iterator<MenuItem> {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public MenuItem next() {
        MenuItem menuItem = list[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= list.length || list[position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

Read the example code discussed/developed in the lectures, and also provided for this week

Demo: Iterator Pattern

Demo …

End