Problem No.09

Problem Name: Implement and design Iterator design pattern(for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Iterator Design Pattern. The pattern provides a way to sequentially access elements of a collection without exposing its underlying representation.

Theory:

The Iterator Pattern is a behavioral design pattern that allows clients to traverse elements in a collection (list, array, tree, etc.) without knowing how the collection is implemented. It promotes encapsulation and a standard way of iteration.

Key Components:

1. Iterator: Defines the interface for accessing elements.
2. ConcreteIterator: Implements the iterator interface.
3. Aggregate (Collection): Defines an interface for creating an iterator.
4. ConcreteAggregate: Implements the collection and returns a concrete iterator.

This pattern simplifies traversal logic and promotes reusability.

Application:

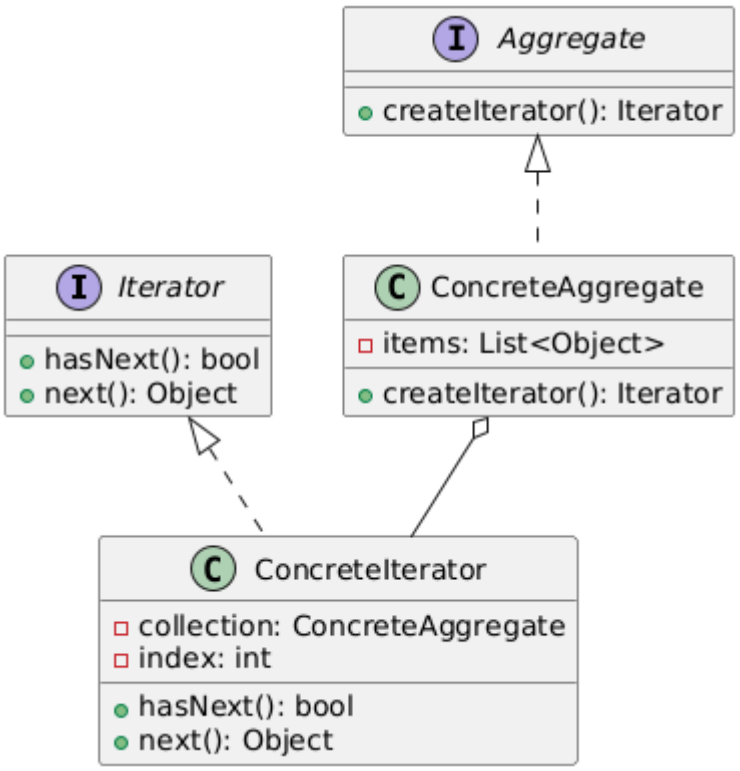
The Iterator pattern is useful when:

1. You want to access elements of a collection without exposing its structure.
2. You want multiple traversals of a collection.
3. You need a uniform way to iterate different types of collections.

Examples:

1. Standard Template Library (STL) iterators in C++.
2. Iterators in Java (Iterator, Iterable).
3. Iterators in Python (__iter__, __next__).

UML Design:



Implementation in C++:

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

// Iterator Interface
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual int next() = 0;
    virtual ~Iterator() = default;
};

// Aggregate Interface
class Aggregate {
public:
    virtual Iterator* createIterator() = 0;
    virtual ~Aggregate() = default;
};

// Concrete Aggregate
class NumberCollection : public Aggregate {
private:
    vector<int> items;
public:
    void addItem(int item) { items.push_back(item); }
```

```
    vector<int>& getItems() { return items; }

    Iterator* createIterator() override;
};

// Concrete Iterator
class NumberIterator : public Iterator {
private:
    NumberCollection& collection;
    size_t index;
public:
    NumberIterator(NumberCollection& c) : collection(c), index(0) {}

    bool hasNext() override {
        return index < collection.getItems().size();
    }

    int next() override {
        return collection.getItems()[index++];
    }
};

// Implementation of createIterator
Iterator* NumberCollection::createIterator() {
    return new NumberIterator(*this);
}

// Demo
int main() {
    NumberCollection numbers;
    numbers.addItem(10);
    numbers.addItem(20);
    numbers.addItem(30);
    numbers.addItem(40);

    Iterator* it = numbers.createIterator();
    cout << "Iterating over collection:" << endl;
    while (it->hasNext()) {
        cout << it->next() << " ";
    }
    cout << endl;

    delete it;
    return 0;
}
```

Result Discussion

Sample Output:

```
Iterating over collection:
10 20 30 40
```

Discussion:

1. NumberCollection stores integers internally but hides its representation.
2. NumberIterator provides a sequential way to access collection elements.
3. Client code iterates through the collection without knowing it uses a vector.

Conclusion:

The **Iterator Design Pattern** provides a standard way to access elements of a collection sequentially without exposing its internal representation. In this lab:

- We designed a custom collection and iterator.
- We demonstrated sequential access to elements.
- We achieved encapsulation and flexibility in traversal.

This pattern is essential in object-oriented design for simplifying access to complex data structures.