CS335 Software Engineering Process

Lab 8

Iterator Design Pattern

In this lab, we are going to study another useful design pattern – Iterator design pattern.

Introduction

An aggregate object, such as a list, should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

Iterator Design Pattern

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.

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents. The term container, used above, can simply be defined as a collection of data or objects. The objects within the container could in turn be collections, making it a collection of collections.

The Iterator pattern enables a client object to traverse through this collection of objects (or the container) without having the container to reveal how the data is stored internally. To accomplish this, the Iterator pattern suggests that a Container object should be designed to provide a public interface in the form of an Iterator object for different client objects to access its contents. An Iterator object contains public methods to allow a client object to navigate through the list of objects within the container.

Implementation

Let us implement the Iterator Design Pattern using a *Shape* class. We will store and iterate the Shape objects using an iterator.

```
public class Shape {
	private int id;
	private String name;
	public Shape(int id, String name){
	this.id = id;
		this.name = name;
	}
	public int getId() {
		return id;
	} // download iterator.zip to see the full source code.
```

The simple Shape class has an id and name as its attributes.

```
public class ShapeStorage {
	private Shape []shapes = new Shape[5];
	private int index;
	public void addShape(String name){
		int i = index++;
		shapes[i] = new Shape(i,name);
	}

	public Shape[] getShapes(){
	return shapes;
	}
}
```

The above class is used to store the Shape objects. The class contains an array of Shape type, for simplicity we have initialized that array up to 5. The *addShape* method is used to add a Shape object to the array and increment the index by one. The *getShapes* method returns the array of Shape type.

```
import java.util.Iterator;
public class ShapeIterator implements
Iterator<Shape>{   private Shape [] shapes;  int pos;

        public ShapeIterator(Shape []shapes){
        this.shapes = shapes;
        }

        @Override
        public boolean hasNext() {
                if(pos >= shapes.length || shapes[pos] == null)
                        return false;
        return true;
        } //download iterator.zip to see the full source code.
```

The above class is an Iterator to the Shape class. The class implements the Iterator interface and defines all the methods of the Iterator interface.

The hasNext method returns a boolean if there's an item left. The next method returns the next item from the collection and the remove method remove the current item from the collection.

```
public class TestIteratorPattern {
        public static void main(String[] args) {
                ShapeStorage storage = new ShapeStorage();
storage.addShape("Polygon");            storage.addShape("Hexagon");
        storage.addShape("Circle");
storage.addShape("Rectangle");
storage.addShape("Square");
```

```
            ShapeIterator iterator = new ShapeIterator(storage.getShapes());
            while(iterator.hasNext()){
                System.out.println(iterator.next());
            }
            System.out.println("Apply removing while iterating...");
            iterator = new ShapeIterator(storage.getShapes());
            while(iterator.hasNext()){
                System.out.println(iterator.next());
                iterator.remove();
            }
        }

}
```

In the above class, we have created a ShapeStorage object and stores the Shape objects in it. Next, we created a ShapeIterator object and assigned it the shapes. We iterated twice, first without calling the remove method and then with the remove method.

When to use the Iterator Design Pattern

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Exercise 1. Download iterator.zip, compile and run the source code.

Exercise 2. Add a method called *contains* in the ShapeIterator. This method checks whether a particular shape exists in the collection (at the name level only). The method signature is as follows: *public boolean contains (String shape)*

Exercise 3. Add a method called *prev* in the ShapeIterator. The method iterates the collection in the reverse order. The method signature is as follows:

public Shape prev ()

Note that you may create *hasPrev()* to make sure we have enough elements for us to move backward .