Iterator Pattern

Scenario:

Imagine you are developing a document editor where a document can contain various elements like paragraphs, images, and tables.

You want to allow users to easily navigate through these elements, perhaps to apply formatting, search for content, or extract information.

Purpose:

The Iterator Pattern provides a way to access the elements of a collection sequentially without exposing its underlying structure.

When to Use:

- When you need to provide a way to traverse a complex collection of objects without exposing its internal structure.
- When you want to provide multiple ways to traverse a collection (e.g., forward, backward).
- When different clients need to traverse the same collection independently.

Key Concepts:

- Iterator: An interface that defines the methods needed to traverse a collection.
- Concrete Iterator: Implements the Iterator interface to traverse a specific collection.
- Aggregate: An interface that defines a method to create an iterator for the collection.
- Concrete Aggregate: Implements the Aggregate interface to return an instance of the concrete iterator.

How It Works:

- Iterator Interface: Defines methods for iterating over a collection.
- Concrete Iterator: Implements the Iterator interface and contains the logic to traverse the specific collection.
- Aggregate Interface: Defines a method to create an iterator for the collection.
- Concrete Aggregate: Implements the Aggregate interface and returns an instance of the concrete iterator.

Implementation Example:

Here is how the implementation might look in Java:

```
// Iterator Interface
interface Iterator<E> {
    boolean hasNext();
    E next();
}

// Concrete Iterator
class DocumentIterator implements Iterator<DocumentElement> {
    private List<DocumentElement> elements;
    private int position = 0;

    public DocumentIterator(List<DocumentElement> elements) {
        this.elements = elements;
    }

    @Override
    public boolean hasNext() {
        return position < elements.size();
    }

    @Override
    public DocumentElement next() {
        return elements.get(position++);
    }
}

// Aggregate Interface
interface Document {
    Iterator<DocumentElement> createIterator();
}

// Concrete Aggregate
class TextDocument implements Document {
    private List<DocumentElement> elements = new ArrayList<>();

    public void addElement(DocumentElement element) {
        elements.add(element);
    }

    @Override
    public Iterator<DocumentElement> createIterator() {
        return new DocumentIterator(elements);
    }
}


// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        TextDocument document = new TextDocument();
        document.addElement(new Paragraph("First Paragraph"));
        document.addElement(new Image("/images/sample.png"));
        document.addElement(new Paragraph("Second Paragraph"));

        Iterator<DocumentElement> iterator = document.createIterator();

        while (iterator.hasNext()) {
            DocumentElement element = iterator.next();
            element.render();
        }
    }
}
```

Key Points:

- Iterator Pattern provides a standardized way to traverse a collection of objects without exposing its structure.
- Encapsulation: Keeps the traversal logic separate from the collection's internal structure.
- Flexibility: Allows different traversal strategies without modifying the collection.