Iterator Design Pattern

The Iterator Pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This pattern is particularly useful when you need to traverse a collection of objects in a uniform manner.

Use Case:

Imagine you are developing a playlist application where users can add songs to a playlist. The application needs a way to iterate through the songs in the playlist, regardless of how the playlist is implemented (e.g., as an array, list, or set).

Components:

1. Iterator (SongIterator): Defines an interface for accessing and traversing elements.
2. Concrete Iterator (PlaylistIterator): Implements the Iterator interface and provides the concrete implementation for traversing the aggregate.
3. Aggregate (Playlist): Defines an interface for creating an Iterator object.
4. Concrete Aggregate (SongPlaylist): Implements the Aggregate interface and returns an instance of the appropriate Concrete Iterator.

Example: Playlist Application

1. Iterator Interface (SongIterator):

```java
public interface SongIterator {
    boolean hasNext();
    Song next();
}
```

2. Concrete Iterator (PlaylistIterator):

```java
import java.util.List;

public class PlaylistIterator implements SongIterator {
    private List<Song> songs;
    private int position;

    public PlaylistIterator(List<Song> songs) {
        this.songs = songs;
        this.position = 0;
    }

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

    @Override
    public Song next() {
        return hasNext() ? songs.get(position++) : null;
    }
}
```

3. Aggregate Interface (Playlist):

```java
public interface Playlist {
```

```
    SongIterator createIterator();
}
```

4. Concrete Aggregate (SongPlaylist):

```java
import java.util.ArrayList;
import java.util.List;

public class SongPlaylist implements Playlist {
    private List<Song> songs;

    public SongPlaylist() {
        this.songs = new ArrayList<>();
    }

    public void addSong(Song song) {
        songs.add(song);
    }

    @Override
    public SongIterator createIterator() {
        return new PlaylistIterator(songs);
    }
}
```

5. Client Code:

```java
public class IteratorPatternDemo {
    public static void main(String[] args) {
        SongPlaylist playlist = new SongPlaylist();

        playlist.addSong(new Song("Song 1"));
        playlist.addSong(new Song("Song 2"));
        playlist.addSong(new Song("Song 3"));

        SongIterator iterator = playlist.createIterator();

        while (iterator.hasNext()) {
            Song song = iterator.next();
            System.out.println("Playing: " + song.getTitle());
        }
    }
}
```

6. Supporting Class (Song):

```java
public class Song {
    private String title;

    public Song(String title) {
        this.title = title;
```

```
    }

    public String getTitle() {
        return title;
    }
}
```

Key Points:

- Uniform Traversal: The Iterator Pattern provides a uniform way to traverse a collection of objects, regardless of its underlying structure.
- Encapsulation: The underlying representation of the collection is encapsulated, and clients can interact with the iterator without needing to know how the collection is implemented.
- Flexibility: New types of collections can be added without modifying existing client code, as long as they provide an appropriate iterator.