Lecture 15

Dr. Umair Rehman

Agenda

- Behavioral Design Patterns
  - Iterator Pattern
  - Direct Access Iteration vs Iterator Design Pattern

Behavioral Design Patterns

- Focused on managing communication and interactions between objects.
- Promoting flexibility and efficient communication.
- By using behavioral patterns
  - Objects can work together more seamlessly
  - Maintaining clear responsibilities and reducing dependencies

Popular Behavioral Design Patterns

- Iterator: Sequential access to collection.
- Chain of Responsibility: Pass request along handlers.
- Command: Encapsulate actions as objects.
- Observer: Notify on state changes.
- State: Alter behavior by state.
- Strategy: Switchable algorithms at runtime.
- Template Method: Algorithm skeleton, modifiable steps.

Iterator

- Iterator: object to traverse a collection
- Enables element-by-element access
- Hides collection's structure
- Standardized sequential access
- Operates without storage knowledge

Iterator

- In simpler terms, think of an iterator as a pointer
  - Helps you go through each item in a collection, one at a time.

Uniform Interface

- Standard Methods: Common methods like `next()`, `hasNext()`
- Consistency: Works across different collection types
- Ease of Use: Simplifies element access for any structure
- Abstraction: Hides data handling details
- Flexibility: Enables swapping collections with minimal changes

Why Use an Iterator?

- You can iterate through different types of collections (arrays, lists, trees, etc.) using the same method calls.

Example

- Imagine a book collection in a library
- If you wanted to access each book one by one
- An iterator would be the librarian who hands you one book at a time
- You don’t need to know the exact layout of the books on the shelf
  - You just rely on the librarian to hand you each book in order.

Example

- `getIterator()`: Gets collection's iterator
- `hasNext()`: Checks for more
- `booksnext()`: Retrieves next book, advances iterator

```
1. Iterator* iterator = library.getIterator();
2.
3. while (iterator->hasNext()) {
4.     Book book = iterator->next();
5.     book.display();
6. }
7.
```

Iterator in C++ Standard Library

- C++ STL: Iterators used extensively
- STL Containers: `std::vector`, `std::list`, `std::map`, etc.
- Standardized Iteration: Provides consistent access to elements

Iterator in C++ Standard Library

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Iterator to go through each element
    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";  // Dereference the iterator to access the value
    }

    return 0;
}

```

Iterator in C++ Standard Library

- `it` is short for "iterator" and is widely used for readability and convention.
- `it` provides direct, sequential access to container elements.
- Using iterators (`begin()` and `end()`) is the standard method in the STL
  - Flexibility and compatibility across container types
- Dereferencing: *it accesses the element at the iterator's current position

Iterators

- Allows element-by-element traversal in a collection.
- Hides the underlying data storage details.
- Provides a standard interface (like `next()`and `hasNext()`)
- Separates traversal logic from data structure.
- Works across different data structures seamlessly.
- Simplifies code and reduces potential iteration errors.
- Lazy Evaluation: Fetches elements when needed, optimizing memory

Direct Access Iteration

- Iterating over elements in a collection by directly
  - Accessing elements based on their index or position
  - Rather than sequentially or through an iterator.

Direct Access Iteration – Example

- Create a console-based Song Playlist application in C++.

- The application should allow users to
  - create a playlist
  - add songs to it,
  - play each song sequentially

Direct Access Iteration

```
 1. #include <iostream>
 2. #include <vector>
 3. #include <string>
 4.
 5. // Playlist class to manage a list of songs
 6. class Playlist {
 7. public:
 8.     // Nested Song class representing each song in the playlist
 9.     class Song {
10.         std::string title;
11.         std::string artist;
12.
13.     public:
14.         // Constructor to initialize Song with title and artist
15.         Song(const std::string& songTitle, const std::string& songArtist)
16.             : title(songTitle), artist(songArtist) {}
17.
18.         std::string getTitle() const { return title; }
19.         std::string getArtist() const { return artist; }
20.
21.         // Displays the song details (simulating "playing" the song)
22.         void play() const {
23.             std::cout << "Playing: " << title << " by " << artist << std::endl;
24.         }
25.     };
```

Direct Access Iteration

```
26.
27. private:
28.     std::vector<Song> songs;  // Directly exposes this structure to the client
29.
30. public:
31.     // Method to add a new song to the playlist
32.     void addSong(const std::string& title, const std::string& artist) {
33.         songs.emplace_back(title, artist);
34.     }
35.
36.     // Method to return the vector of songs directly
37.     const std::vector<Song>& getSongs() const {
38.         return songs;  // Directly exposes the collection of songs to the client
39.     }
40. };
```

Direct Access Iteration

```
42. int main() {
43.     // Create a playlist and add songs
44.     Playlist playlist;
45.     playlist.addSong("Let it Be", "The Beatles");
46.     playlist.addSong("Imagine", "John Lennon");
47.     playlist.addSong("Bohemian Rhapsody", "Queen");
48.
49.     // Access the internal collection of songs directly
50.     const std::vector<Playlist::Song>& songs = playlist.getSongs();
51.
52.     // Client code iterates over the vector of songs directly
53.     for (size_t i = 0; i < songs.size(); ++i) {
54.         songs[i].play();  // Accesses each song and plays it directly
55.     }
56.
57.     return 0;
58. }
59.
```

Direct Access Iteration

- Client manually loops over collection.
- Uses for or while loops on arrays, vectors
- Directly interacts with structure.
- Dependency Risk: Tied to collection type (e.g., array, linked list).

Iterator Design Pattern

- Iterator Pattern: Provides sequential access to collection elements.
- Hides collection's internal structure.
- Works across various collection types.
- Simplifies traversal without exposing data structure details.

Iterator Pattern– Example

- Create a console-based Song Playlist application in C++.

- The application should allow users to
  - create a playlist
  - add songs to it,
  - play each song sequentially

Iterator Interface

- Defines the Iterator interface standard methods
- `hasNext()` checks if there are more elements in the collection
- `next()` returns the current element and advances the iterator

Iterator Interface

```
 1. #include <iostream>
 2. #include <vector>
 3. #include <string>
 4.
 5. // Iterator interface for Songs
 6. class Iterator {
 7. public:
 8.     virtual bool hasNext() const = 0;      // Checks if more elements are left
in the collection
 9.     virtual const class Playlist::Song& next() = 0; // Returns the current
element and advances to the next
10.     virtual ~Iterator() = default;         // Virtual destructor for safe
deletion of derived objects
11. };
```

Iterable Interface

- Defines the `Iterable` interface with a `getIterator()` method.

- Any collection that implements `Iterable`
  - Must provide a way to return an iterator

Iterable Interface

```
12.
13. // Iterable interface for collections
14. class Iterable {
15. public:
16.     virtual Iterator* getIterator() const = 0;  // Returns an iterator for the
collection
17.     virtual ~Iterable() = default;              // Virtual destructor for safe
deletion of derived objects
18. };
19.
```

Concrete Iterable (Playlist)

- Implements the `Iterable` interface, allowing it to provide an iterator.

- Contains a nested Song class representing each item in the `collection.addSong()` adds a new Song to the playlist.

Concrete Iterable (Playlist)

```
20. // Concrete Iterable: Playlist
21. class Playlist : public Iterable {
22. public:
23.     // Nested Song class representing each song in the playlist
24.     class Song {
25.         std::string title;  // Stores the title of the song
26.         std::string artist; // Stores the artist's name
27.
28.     public:
29.         // Constructor to initialize Song with title and artist
30.         Song(const std::string& songTitle, const std::string& songArtist)
31.             : title(songTitle), artist(songArtist) {}
32.
33.         std::string getTitle() const { return title; }  // Returns the song title
34.         std::string getArtist() const { return artist; } // Returns the artist name
35.
36.         // Displays the song details (simulating "playing" the song)
37.         void play() const {
38.             std::cout << "Playing: " << title << " by " << artist << std::endl;
39.         }
40.     };
```

Concrete Iterable (Playlist)

```
42. private:
43.     std::vector<Song> songs;  // Vector to store a list of Song objects
44.
45. public:
46.     // Method to add a new song to the playlist
47.     void addSong(const std::string& title, const std::string& artist) {
48.         songs.emplace_back(title, artist);  // Adds a new Song object to the songs vector
49.     }
```

Concrete Iterator (`PlaylistIterator`)

- Implements the Iterator interface specifically for Playlist.
- Keeps a reference to the Playlist
  - an index for tracking the current position.
- `hasNext()` checks if more elements are available.
- `next()` returns the current Song and advances the position.

Concrete Iterator (PlaylistIterator)

```
51.     // Concrete Iterator implementation for Playlist
52.     class PlaylistIterator : public Iterator {
53.         const Playlist& playlist;  // Reference to the playlist being iterated over
54.         size_t index;              // Tracks the current position in the playlist
55.
56.     public:
57.         // Constructor initializing with a reference to the playlist and starting index
58.         PlaylistIterator(const Playlist& pl) : playlist(pl), index(0) {}
59.
60.         // Checks if there are more songs in the playlist
61.         bool hasNext() const override {
62.             return index < playlist.songs.size();  // Returns true if current index is within bounds
63.         }
64.
65.         // Returns the current song and moves to the next song
66.         const Song& next() override {
67.             return playlist.songs[index++];  // Returns the song at the current index and increments index
68.         }
69.     };
70.
71.     // Method to get an iterator for the playlist
72.     Iterator* getIterator() const override {
73.         return new PlaylistIterator(*this);  // Returns a new PlaylistIterator initialized with this playlist
74.     }
75. };
```

PlalistIterator

- Initialization:
  - The client calls `getIterator()` on the Playlist object, which creates a new `PlaylistIterator` starting at `index = 0`.
- Iteration (Looping with `hasNext()` and `next()`):
  - The client checks `hasNext()`:
    - `If index < playlist.songs.size()`, `hasNext()` returns true, meaning there are more songs.
    - If `index` reaches `playlist.songs.size()`, `hasNext()` returns false, indicating the end of the playlist.
- The client calls `next()`:
  - `next()` returns the song at the current index, then increments index.
  - This process repeats, allowing the client to access each song in order.

Client Code:

- Creates a `Playlist` and adds songs to it.
- Retrieves an iterator for the Playlist and uses it to access each Song sequentially.
- Calls `play()` on each Song, displaying its details.

Iterator in C++ Standard Library

```
77. // Client code that uses the Playlist and its iterator
78. int main() {
79.     Playlist playlist;  // Create a Playlist object
80.
81.     // Add some songs to the playlist
82.     playlist.addSong("Imagine", "John Lennon");           // Adds "Imagine" by John Lennon
83.     playlist.addSong("Hey Jude", "The Beatles");          // Adds "Hey Jude" by The Beatles
84.     playlist.addSong("Bohemian Rhapsody", "Queen");       // Adds "Bohemian Rhapsody" by Queen
85.
86.     // Get an iterator for the playlist
87.     Iterator* iterator = playlist.getIterator();          // Creates an iterator for traversing the playlist
88.
89.     // Use the iterator to play each song in the playlist
90.     while (iterator->hasNext()) {                         // Loop while there are more songs in the playlist
91.         const Playlist::Song& song = iterator->next();    // Get the next song and advance the iterator
92.         song.play();                                      // Call the play() method to display song details
93.     }
94.
95.     delete iterator; // Clean up the iterator after use to avoid memory leaks
96.     return 0;        // End of program
97. }
98.
```

PlalistIterator

- First Call:
  - `hasNext()` returns `true` (since `index = 0` and there are 3 songs).
  - `next()` returns "Imagine" and increments `index` to 1.
- Second Call:
  - `hasNext()` returns true (since `index = 1`).
  - `next()` returns "Hey Jude" and increments `index` to `2.`
- Third Call:
  - `hasNext()` returns true (since `index = 2`).
  - next() returns "Bohemian Rhapsody" and increments index to 3.
- End of Iteration:
  - `hasNext()` now returns false (since `index = 3`, which is equal to the total number of songs).
  - The loop stops because there are no more songs to iterate over.

Pros & Cons Iterator Design Pattern

- Pros
  - Hides collection details.
  - Consistent Interface: Standard `hasNext()`, `next()`.
  - Simplifies Client Code: No traversal logic in client.
  - Flexible Traversal: Supports various traversal orders.
  - Concurrent Traversal: Multiple independent iterators.
- Cons
  - Added Complexity: Extra class and logic.
  - Performance Overhead: Extra method calls.

Conclusion

- Behavioral Design Patterns
  - Iterator Pattern
  - Direct Access Iteration vs Iterator Design Pattern