

Iterator

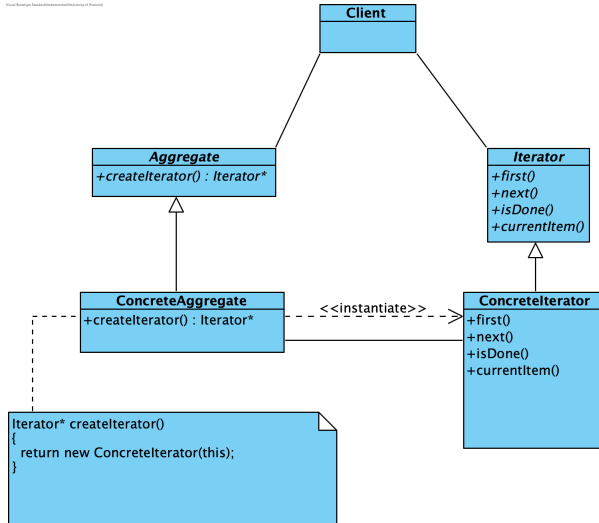
Department of Computer Science
University of Pretoria

1 September 2023

Name and Classification: Iterator (Object Behavioural)

Intent: “Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation” GoF(257)

Small Knowledge Base for the Knowledge Engineering of Formal



Iterator

- Defines an interface for accessing and traversing elements.

Concrete Iterator

- Implements the Iterator interface
- Keeps track of the current position in the traversal of the aggregate

Aggregate

- Defines an interface for creating an Iterator object

Concrete Aggregate

- Implements the Iterator creation Interface to return an instance of the proper concrete iterator.

Separation of concerns: The separation of the aggregate - and how the aggregate behaves - from how the aggregate is traversed.

By moving the mechanism of iteration outside the aggregate, all aggregates can be traversed in a uniform manner.

The C++ Containers library – referred to as the Standard Template Library (STL) prior to C++11 – provide efficient implementations for many data structures (for example: array, vector, deque, forward_list, list set, map, stack etc.)

In C++ STL
A Queue iterator
Covariant return type

Member function table

- functions present in C++03
- functions present since C++11
- functions present since C++17
- functions present since C++20

[illegible]

<http://en.cppreference.com/w/cpp/container>

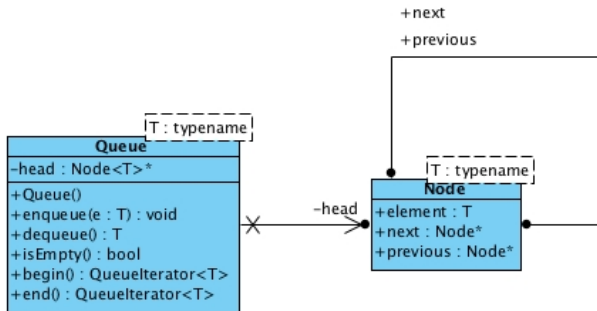
From the Observer example:

```
...  
// Create the container  
vector<PitCrew*> pitcrew;  
...  
// Insert elements into the container  
pitcrew.push_back(person);  
...  
// Create an iterator  
vector<PitCrew*>::iterator it = pitcrew.begin();  
// Iterate through the container  
while ((it != pitcrew.end()) && (! found)) {  
    if (*it == person) {  
        found = true;  
        pitcrew.erase(it);  
    }  
    ++it;  
}  
// OR  
vector<PitCrew*>::iterator it = pitcrew.begin();  
for (it = pitcrew.begin(); it != pitcrew.end(); ++it){  
    (*it)->update();  
}
```

Using templates in programs to enable the “structure” to cater for different types of objects as long as they adhere to certain traits.

Examples of traits are: Assignable,
Constructible...

Visual Paradigm Standard(University of Pretoria)



```
#ifndef _NODE_H
#define _NODE_H
    template <typename T>
    class Node {
    public:
        T element;
        Node *next;
        Node *previous;
    };

#endif
```

```
#ifndef _QUEUE_H
#define _QUEUE_H

#include "Node.h"
#include "QueueIterator.h"

template <typename T>
class Queue{
    friend class QueueIterator<T>;
public:
    Queue();
    void enqueue(T e);
    T dequeue();
    bool isEmpty();
    QueueIterator<T> begin();
    QueueIterator<T> end();
private:
    Node<T>* head;
};

#include "Queue.cpp" // Necessary - working with templates

#endif
```

```
#ifndef _QUEUE_C
#define _QUEUE_C

#include "Node.h"
#include "Queue.h"
#include "QueueIterator.h"

template <typename T>
Queue<T>::Queue(){
    head = 0;
}
...
```

```
...
template <typename T>
void Queue<T>::enqueue(T e){
    Node<T>* n = new Node<T>();
    n->element = e;
    if (isEmpty()) {
        n->next = n;
        n->previous = n;
    } else {
        n->next = head;
        n->previous = head->previous;
        head->previous->next = n;
        head->previous = n;
    }
    head = n;
}
...
```



```
...
template <typename T>
T Queue<T>::dequeue(){
    if (isEmpty())
        return 0;
    else if (head->previous == head) {
        Node<T> *tmp= head;
        head = 0;
        return tmp->element;
    } else {
        Node<T> *tmp = head->previous;
        head->previous = head->previous->previous;
        head->previous->next = head;
        return tmp->element;
    }
}
...
```

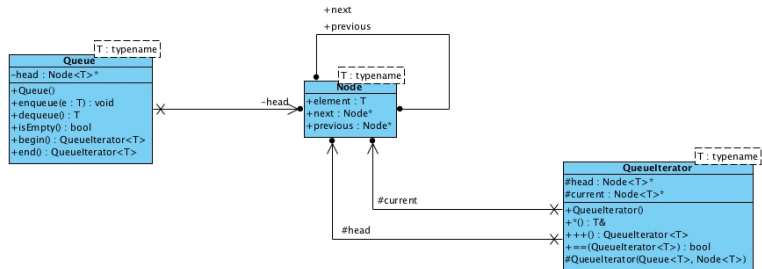
```
...
template <typename T>
bool Queue<T>::isEmpty(){
    return head == 0;
}

template <typename T>
QueueIterator<T> Queue<T>::begin(){
    return QueueIterator<T> (*this, head);
}

template <typename T>
QueueIterator<T> Queue<T>::end(){
    return QueueIterator<T> (*this, head->previous);
}

#endif
```

Visual Paradigm Standard (University of Pretoria)



Note, this code has collapsed both the aggregate (Queue is both the Aggregate and ConcreteAggregate participants) and iterator (QueueIterator represents both the Iterator and ConcreteIterator participants) hierarchies. Refer to Figure 2 in the notes for an example with two complete aggregate and iterator hierarchies.

```
#ifndef _QUEUEITERATOR_H
#define _QUEUEITERATOR_H

template <typename T>
class Queue;
template <typename T>
class Node;

template <typename T>
class QueueIterator {
    friend class Queue<T>;
public:
    QueueIterator();
    T& operator*();
    QueueIterator<T> operator++();
    bool operator==(const QueueIterator<T>&) const;
protected:
    QueueIterator(const Queue<T>&, Node<T>*);
    Node<T>* head;
    Node<T>* current;
};
#include "QueueIterator.cpp"
#endif
```

```
#ifndef _QUEUEITERATOR_C
#define _QUEUEITERATOR_C

#include <iostream>
using namespace std;
#include "QueueIterator.h"
#include "Queue.h"
#include "Node.h"

template <typename T>
QueueIterator<T>::QueueIterator() : head(0), current(0) {}

template <typename T>
QueueIterator<T>::QueueIterator(const Queue<T>& source,
                                Node<T>* p) : head(source.head), current(p) {}

template <typename T>
T& QueueIterator<T>::operator*(){
    return current->element;
}

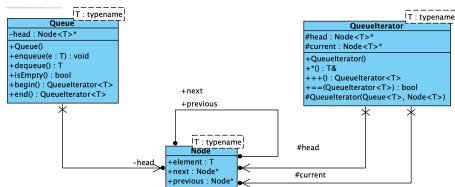
...
```

```
...
template<typename T>
QueueIterator<T> QueueIterator<T>::operator++(){
    if (this != nullptr)
        this->current = this->current->next;
    return *this;
}

template <typename T>
bool QueueIterator<T>::operator==(const QueueIterator<T>& rhs)
                                const{
    return current == rhs.current;
}

#endif
```

Now having seen a user-defined queue and its corresponding iterator, how would you abstract the design to include other data structures?



Concept - **covariant return types**. Replace the return type with a more specialised type in a derived class.

```
class A {  
public:  
    A* someFunction() {  
        return new A;  
    }  
};  
  
class B : public A {  
public:  
    B* someFunction() {  
        return new B;  
    }  
};
```