Iterator

Linda Marshall

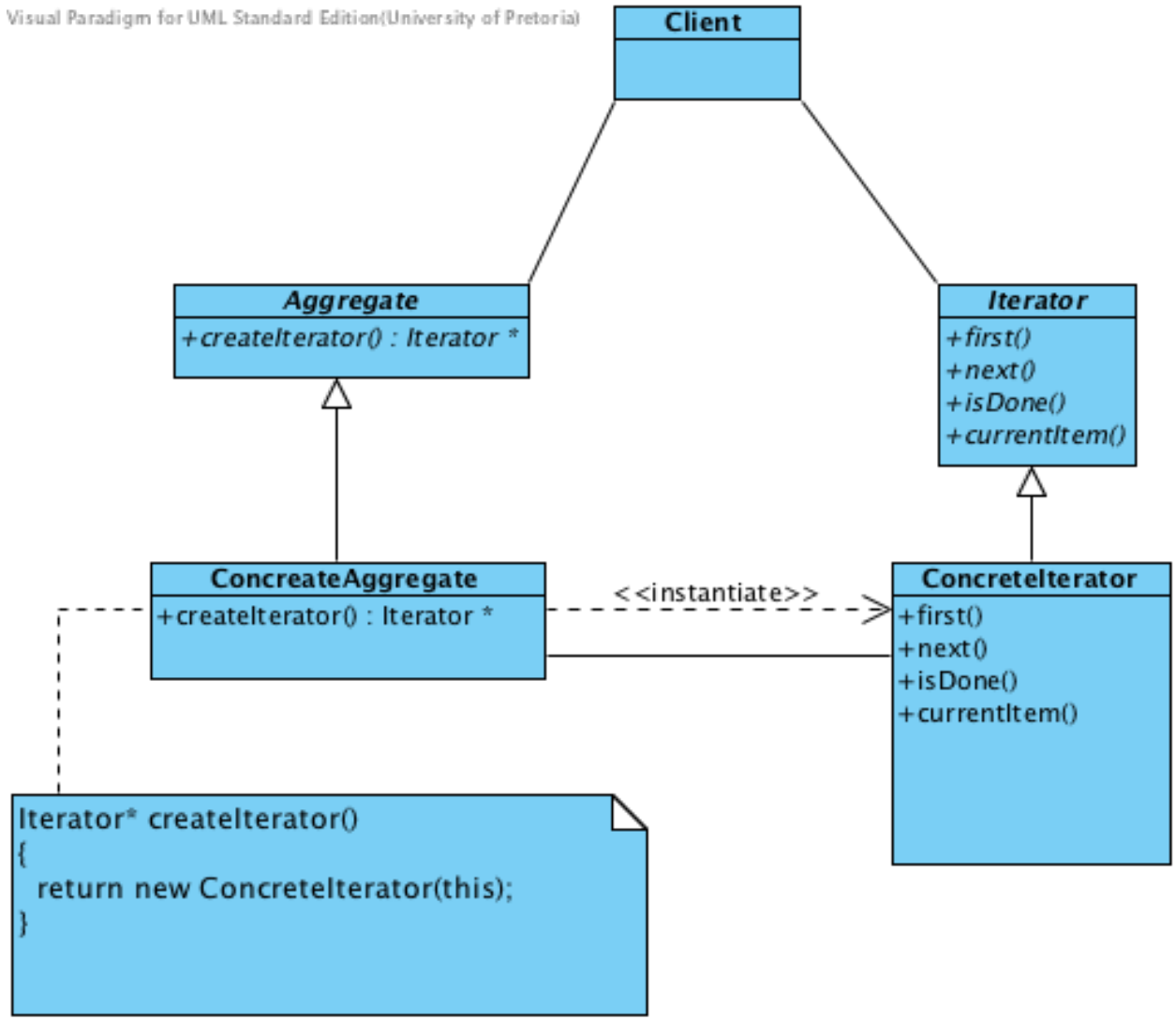
Department of Computer Science
University of Pretoria

28 September 2021

Name and Classification: Iterator (Object Behavioural)

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

"Provide a way to access the elements of an aggregate object sequentially without exposing its underlaying representation" GoF(257)


Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
Aggregate
+createIterator() : Iterator *
Iterator
+first()
+next()
+isDone()
+currentItem()
ConcreateAggregate
+createIterator() : Iterator *
<<instantiate>>
ConcreteIterator
+first()
+next()
+isDone()
+currentItem()
Iterator* createIterator()
{
return new ConcreteIterator(this);
}

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.

- **Factory Method**: Both Iterator and Factory Method use a subclass to decide what object should be created. In fact `createIterator()` is an example of a factory method.
- **Memento**: The memento pattern is often used in conjunction with the iterator pattern. An iterator can use a memento to capture the state of the aggregate. This memento is stored inside the iterator to be used for traversing the aggregate.
- **Adapter**: Both patterns provides an interface through which operations are performed. They differ in the reason for providing this interface.
- **Composite**: Recursive structures such as composites usually need iterators to traverse them sequentially. Although recursive traversal might be very easy to implement without extending the composite pattern, its is strongly advised to create a composite iterator.

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

Iterators provide a universal interface with which these containers can be accessed.

Member function table

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

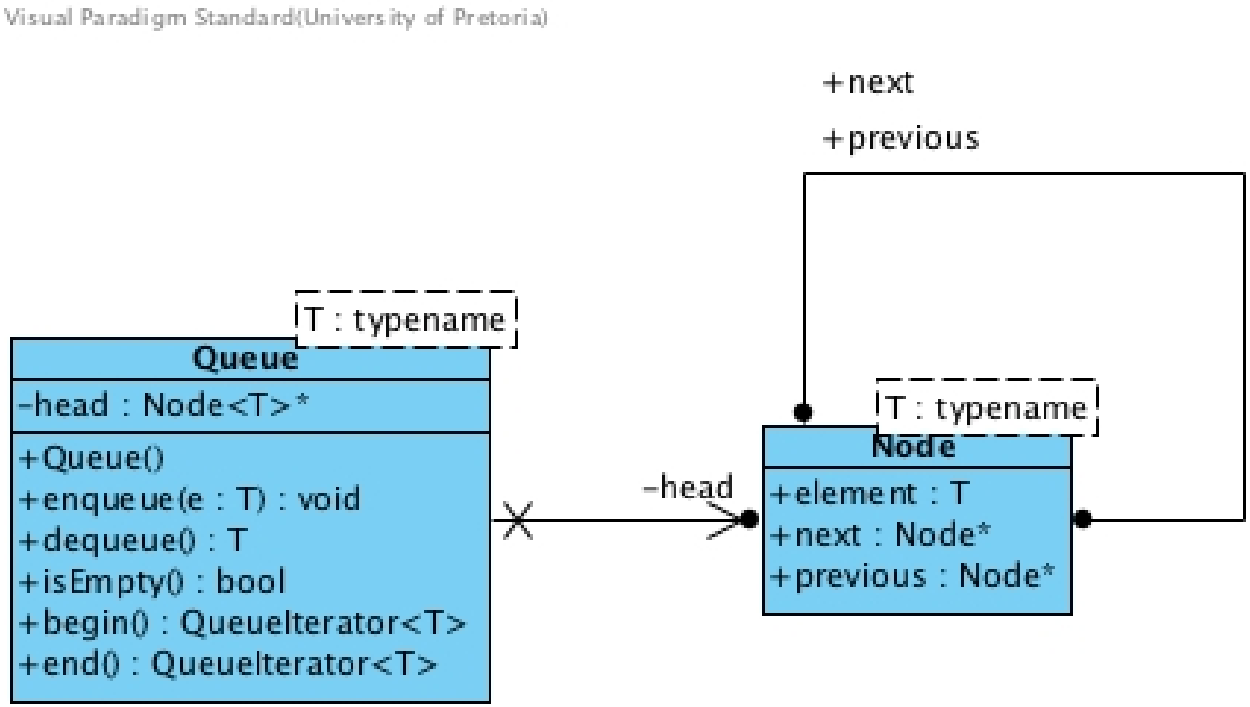
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Header		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	data	data	data														
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity								bucket_count	bucket_count	bucket_count	bucket_count			
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	insert_or_assign								insert_or_assign				insert_or_assign				
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	try_emplace								try_emplace				try_emplace				
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front				push_front	push_front	push_front										
	emplace_front				emplace_front	emplace_front	emplace_front										
	pop_front				pop_front	pop_front										pop	pop
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
	merge				merge	merge	merge	merge	merge	merge	merge	merge	merge	merge			
	extract						extract	extract	extract	extract	extract	extract	extract	extract			
List operations	splice				splice_after	splice											
	remove				remove	remove											
	remove_if				remove_if	remove_if											
	reverse				reverse	reverse											
	unique				unique	unique											
	sort				sort	sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	contains						contains	contains	contains	contains	contains	contains	contains	contains			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

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 is certain traits.
Examples of traits are: Assignable, Constructible...


Visual Paradigm Standard(University of Pretoria)
+next
+previous
T : typename
Queue
-head : Node<T>*
+Queue()
+enqueue(e : T) : void
+dequeue() : T
+isEmpty() : bool
+begin() : QueueIterator<T>
+end() : QueueIterator<T>
-head
T : typename
Node
+element : T
+next : Node*
+previous : Node*

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

#endif
```

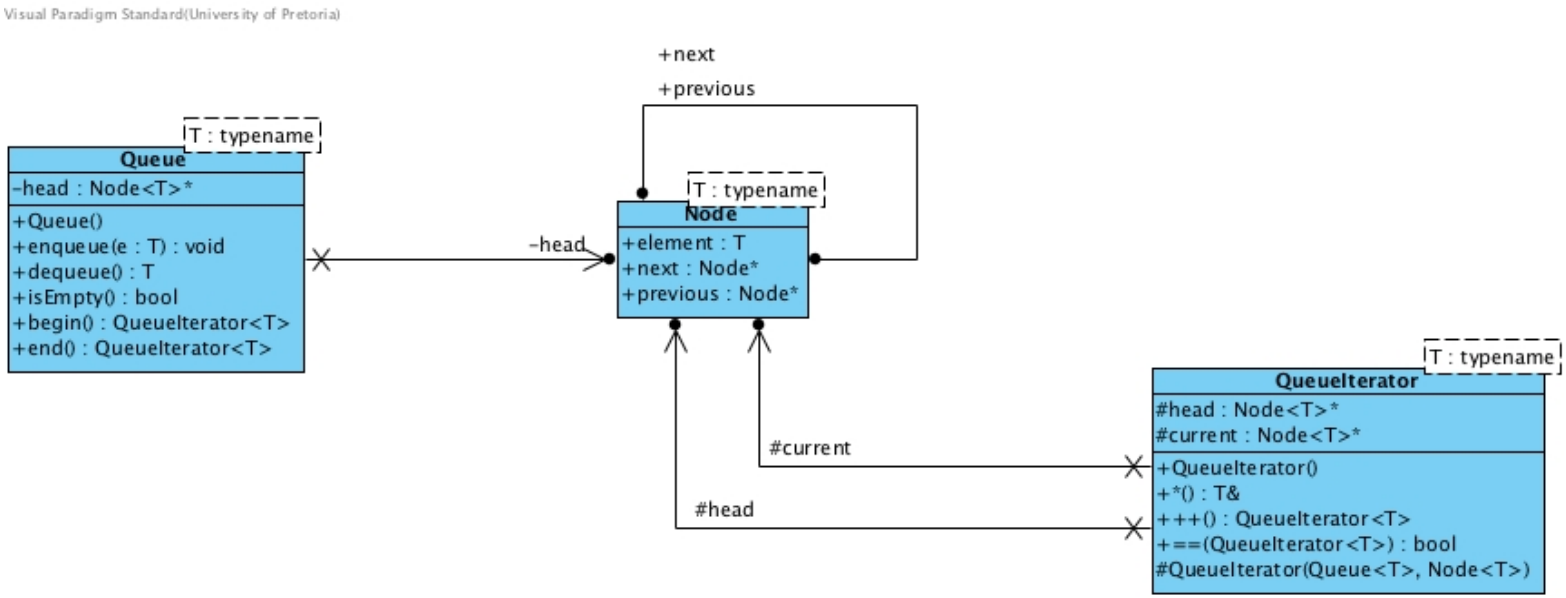
```
#ifndef _QUEUE_H
#define _QUEUE_H

    #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
```



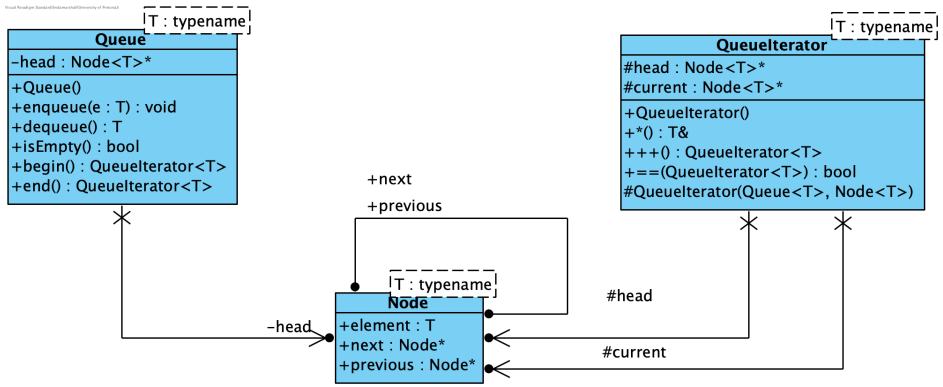
Note, this code has collapsed both the aggregate (Queue is both the Aggregate and ConcreteAggregate participants) and iterator (QueueIterator represents both the Iterator and ConcreteIterator participants) hierarchies. Refer to Figure 2 in the notes for an example with two complete aggregate and iterator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