



Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University



17. Iterator Pattern

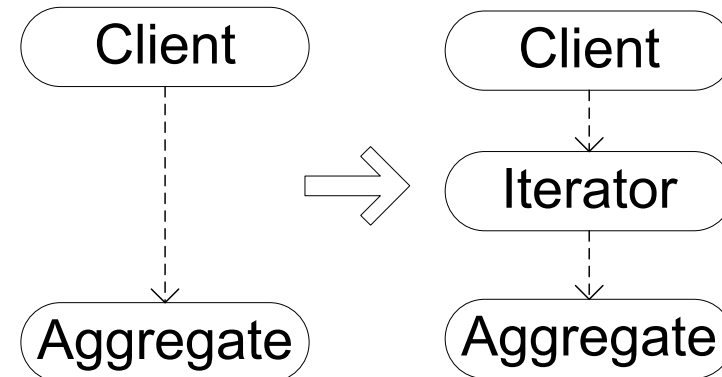


Intent

- Provide a way to **access** the elements of an aggregate object **sequentially** without exposing its underlying **representation**.
 - Cursor
 - 迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象。
-

Intent for OCP

- Traversal mechanism is unchanged, but the traversed aggregate is changed. The code in client side should be modified because different aggregates have different traversal interface.
- Aggregate is unchanged, but traversal mechanism is changed. For example, add filtering algorithm. The interface of aggregate should be modified to introduced the new traversal approaches.

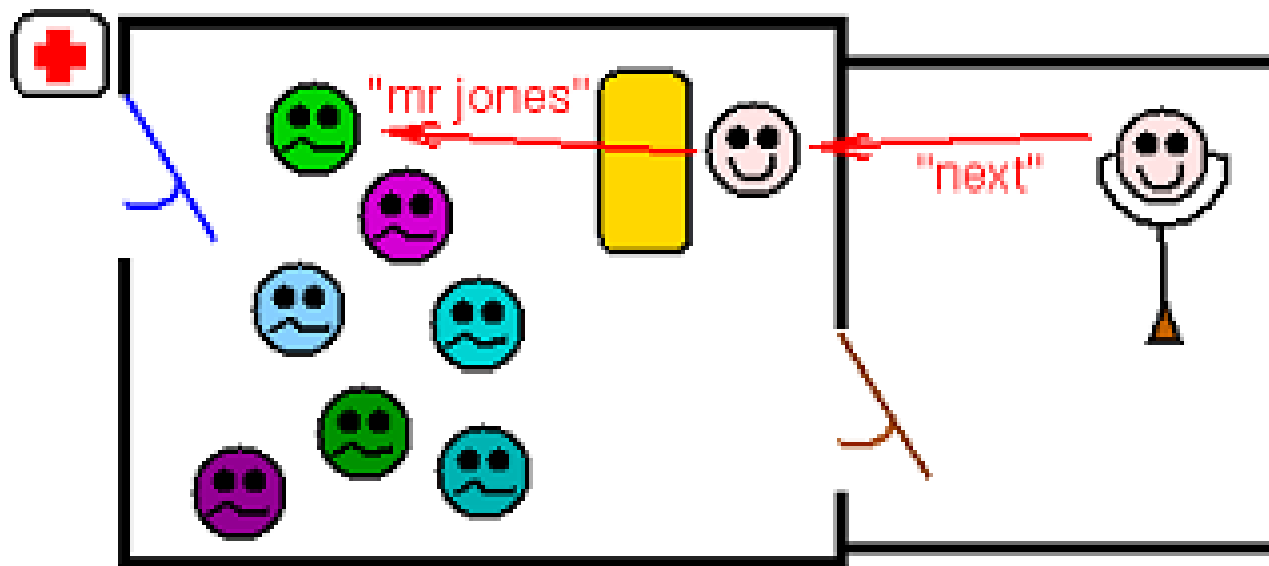




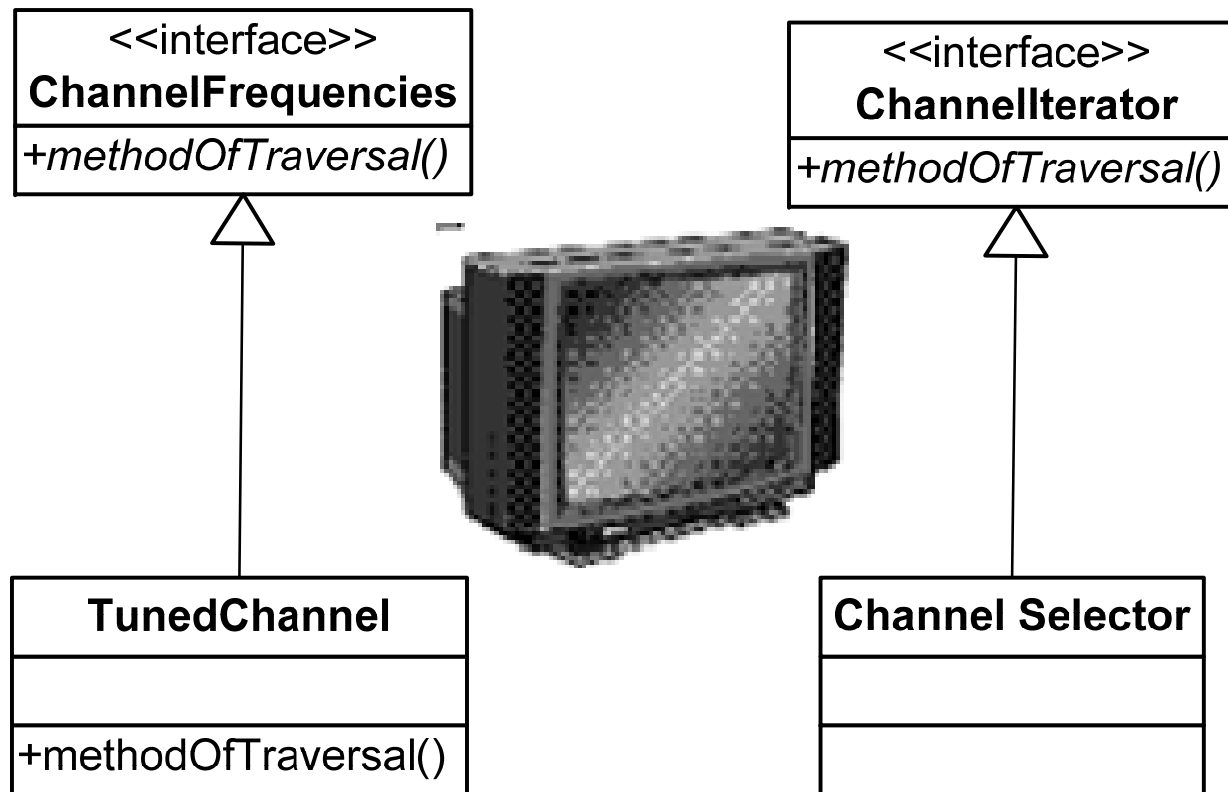
Traversal mechanism

- Forward: The directions that the elements increased;
 - Forward Iteration
 - Backward: The directions that the elements decreased
 - Backward Iteration
-

Example



Example





Example

```
interface Channel extends Comparable<Channel> {  
    public String getName();  
    public void setName(String name);  
}  
  
class ChannelImpl implements Channel {  
    private String name;  
    public ChannelImpl() {  
    }  
    public ChannelImpl(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int compareTo(Channel channel) {  
        return this.getName().compareTo(channel.getName());  
    }  
    public String toString() {  
        return this.name;  
    }  
}
```




1 - Public Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
    public E getChannel(int index);  
    public int size();  
}
```

```
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
}
```



1 - Public Iterator

```
class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Channel getChannel(int index) {
        return channelList.get(index);
    }
    public int size() {
        return channelList.size();
    }
    public Iterator<Channel> iterator() {
        return new ChannelItr(this);
    }
}
```



1 – Public Iterator

```
class ChannelItr implements Iterator<Channel> {  
    private ChannelSet<Channel> channelSet;  
    private int current = 0;  
    public ChannelItr(ChannelSet<Channel> channelSet) {  
        this.channelSet = channelSet;  
    }  
    public boolean hasNext() {  
        return current < channelSet.size();  
    }  
    public Channel next() {  
        return channelSet.getChannel(current++);  
    }  
}
```



1 – Public Iterator

```
class Client {  
    public void testChannelIterator() {  
        ChannelSet<Channel> channelSet = new ChannelSetImpl();  
        channelSet.addChannel(new ChannelImpl("CCTV-1"));  
        channelSet.addChannel(new ChannelImpl("CCTV-2"));  
        channelSet.addChannel(new ChannelImpl("CCTV-3"));  
        Iterator<Channel> it = channelSet.iterator();  
        while (it.hasNext()) {  
            System.out.println(it.next());  
        }  
    }  
}
```




2 - Private Passive (Internal) Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
}  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
}
```

2 - Private Passive (Internal) Iterator

```
class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private int current = 0;
        private Itr() {
            Collections.sort(channelList);
        }
        public boolean hasNext() {
            return current < channelList.size();
        }
        public Channel next() {
            return channelList.get(current++);
        }
    }
}
```



3 - Private **Active (External)** Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
}  
interface Iterator<E extends Channel> {  
    public int size();  
    public E getChannel(int index);  
}
```

3 - Private Active (External) Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private Itr() {
            Collections.sort(channelList);
        }
        public int size() {
            return channelList.size();
        }
        public Channel getChannel(int index) {
            return channelList.get(index);
        }
    }
}
```


3 - Private **Active (External)** Iterator

```
class Client {  
    public void testChannelIterator() {  
        ChannelSet<Channel> channelSet = new ChannelSetImpl();  
        channelSet.addChannel(new ChannelImpl("CCTV-1"));  
        channelSet.addChannel(new ChannelImpl("CCTV-2"));  
        channelSet.addChannel(new ChannelImpl("CCTV-3"));  
        Iterator<Channel> it = channelSet.iterator();  
        for (int i = 0; i < it.size(); i++) {  
            System.out.println(it.getChannel(i));  
        }  
    }  
}
```

4 - Private Passive (Internal) Static (Copied) Iterator

```
private class Itr implements Iterator<Channel> {  
    private int current = 0;  
    private List<Channel> copy;  
    private Itr() {  
        copy = new ArrayList<Channel>();  
        copy.addAll(channelList);  
        Collections.sort(channelList);  
    }  
    public boolean hasNext() {  
        return current < channelList.size();  
    }  
    public Channel next() {  
        return channelList.get(current++);  
    }  
}
```

5 - Private Passive (Internal) Fastfail Iterator

```
private class Itr implements Iterator<Channel> {
    private int originalSize;
    private int current = 0;
    private Itr() {
        originalSize = channelList.size();
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        checkModify();
        return current < channelList.size();
    }
    public Channel next() {
        checkModify();
        return channelList.get(current++);
    }
    public void checkModify() {
        if (originalSize != channelList.size()) {
            throw new RuntimeException(
                "Iterator is invalid, the aggregate is modified!");
        }
    }
}
```



6 - Private Passive (Internal) **Robust** Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public void removeChannel(int index);  
    public Iterator<E> iterator();  
}  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
}
```

6 - Private Passive (Internal) Robust Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    private List<Itr> iterators;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
        iterators = new ArrayList<Itr>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public void removeChannel(int index) {
        if (index >= channelList.size()) {
            return;
        }
        channelList.remove(index);
        for (Itr it : iterators) {
            if (index <= it.current) {
                it.current--;
            }
        }
    }
    public Iterator<Channel> iterator() {
        Itr it = new Itr();
        iterators.add(it);
        return it;
    }
}
```

6 - Private Passive (Internal) Robust Iterator

```
private class Itr implements Iterator<Channel> {  
    private int current = 0;  
    private Itr() {  
        Collections.sort(channelList);  
    }  
    public boolean hasNext() {  
        boolean hasNext = current < channelList.size();  
        if (!hasNext) {  
            iterators.remove(this);  
        }  
        return hasNext;  
    }  
    public Channel next() {  
        return channelList.get(current++);  
    }  
}
```



7 - Private Passive (Internal) Editable Iterator

```
interface ChannelSet<E extends Channel> {  
    public Iterator<E> iterator();  
}  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
    //add to the last  
    public void add(E channel);  
    //remove the element which returned by next()  
    public void remove();  
    //replace the element which returned by next()  
    public void replace(E channel);  
}
```




7 - Private Passive (Internal) Editable Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private int current = 0;
        public boolean hasNext() {
            return current < channelList.size();
        }
        public Channel next() {
            return channelList.get(current++);
        }
        public void add(Channel channel) {
            channelList.add(channel);
        }
        public void remove() {
            channelList.remove(--current);
        }
        public void replace(Channel channel) {
            channelList.set(current - 1, channel);
        }
    }
}
```



8 - Private Passive (Internal) Iterator **with additional operations**

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
}  
  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public boolean hasPrevious();  
    public int currentIndex();  
    // All return current item  
    public E next();  
    public E previous();  
    // return the first or last item  
    public E first();  
    public E last();  
    public E moveTo(int index);  
}
```



8 - Private Passive (Internal) Iterator **with additional operations**

```
private class Itr implements Iterator<Channel> {
    private int current = 0;
    private Itr() {
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        return current < channelList.size();
    }
    public boolean hasPrevious() {
        return current >= 0;
    }
    public int currentIndex() {
        return current;
    }
    public Channel next() {
        return channelList.get(current++);
    }
    public Channel previous() {
        return channelList.get(current--);
    }
    public Channel first() {
        current = 0;
        return channelList.get(current);
    }
    public Channel last() {
        current = channelList.size() - 1;
        return channelList.get(current);
    }
    public Channel moveTo(int index) {
        if (index >= 0 && index < channelList.size()) {
            current = index;
            return channelList.get(current);
        }
        throw new RuntimeException("input index is out of range");
    }
}
```

9 - Self Iterator

```
interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
}
interface ChannelSet<E extends Channel> extends Iterator<E> {
    public void addChannel(E channel);
}
class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    private int current = 0;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public boolean hasNext() {
        return current < channelList.size();
    }
    public Channel next() {
        return channelList.get(current++);
    }
}
```



9 - Self Iterator

```
class Client {  
    public void testChannelIterator() {  
        ChannelSet<Channel> channelSet = new ChannelSetImpl();  
        channelSet.addChannel(new ChannelImpl("CCTV-1"));  
        channelSet.addChannel(new ChannelImpl("CCTV-2"));  
        channelSet.addChannel(new ChannelImpl("CCTV-3"));  
        Iterator<Channel> it = channelSet;  
        while (it.hasNext()) {  
            System.out.println(it.next());  
        }  
    }  
}
```

10 – Iterator without Aggregate

```
interface ChannelSet<E extends Channel> {
    public void addChannel(E channel);
    public Iterator<E> iterator();
}


interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
}

class CCTVChannelSet implements ChannelSet<Channel> {
    private Channel CCTV1 = new ChannelImpl("CCTV-1");
    private Channel CCTV2 = new ChannelImpl("CCTV-2");
    private Channel CCTV3 = new ChannelImpl("CCTV-3");
    private Channel CCTV4 = new ChannelImpl("CCTV-4");
    private Channel CCTV5 = new ChannelImpl("CCTV-5");
    private final int CCTVChannelSize = 5;
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    public void addChannel(Channel channel) {
    }
}
```

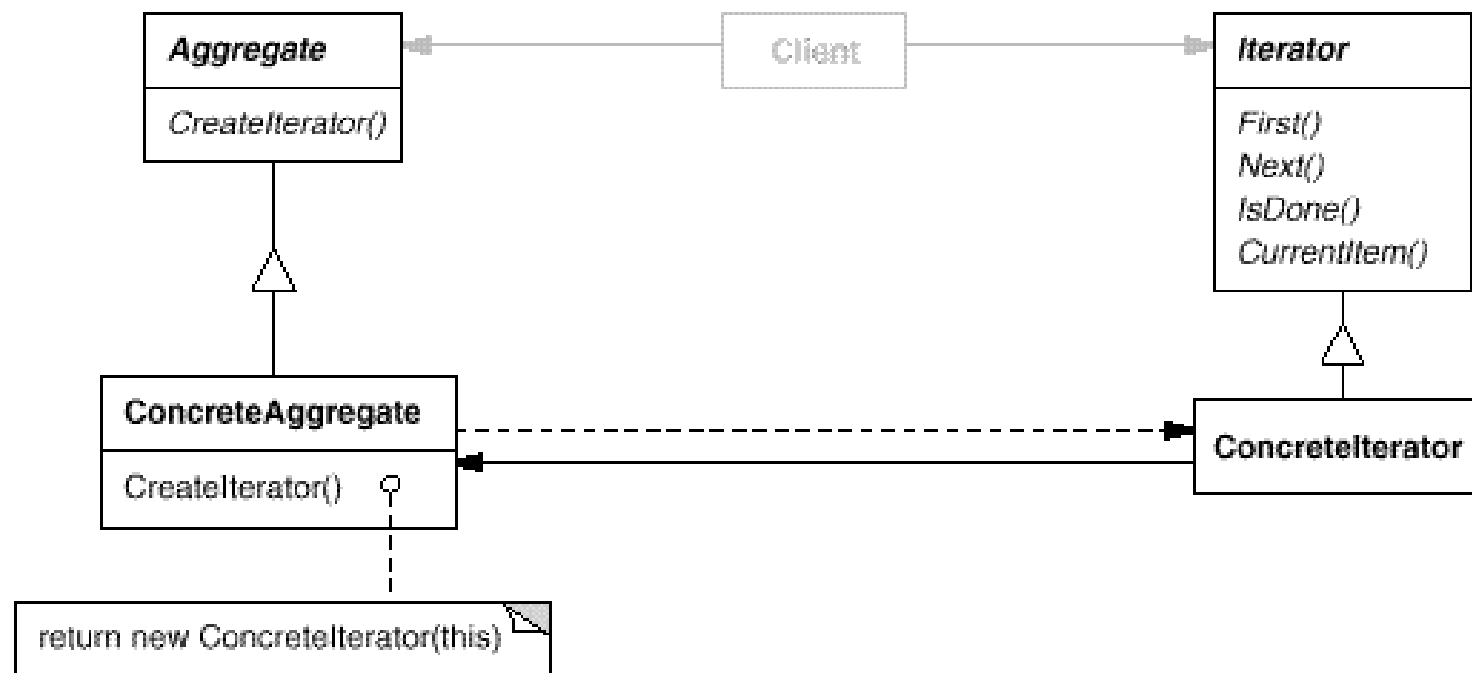


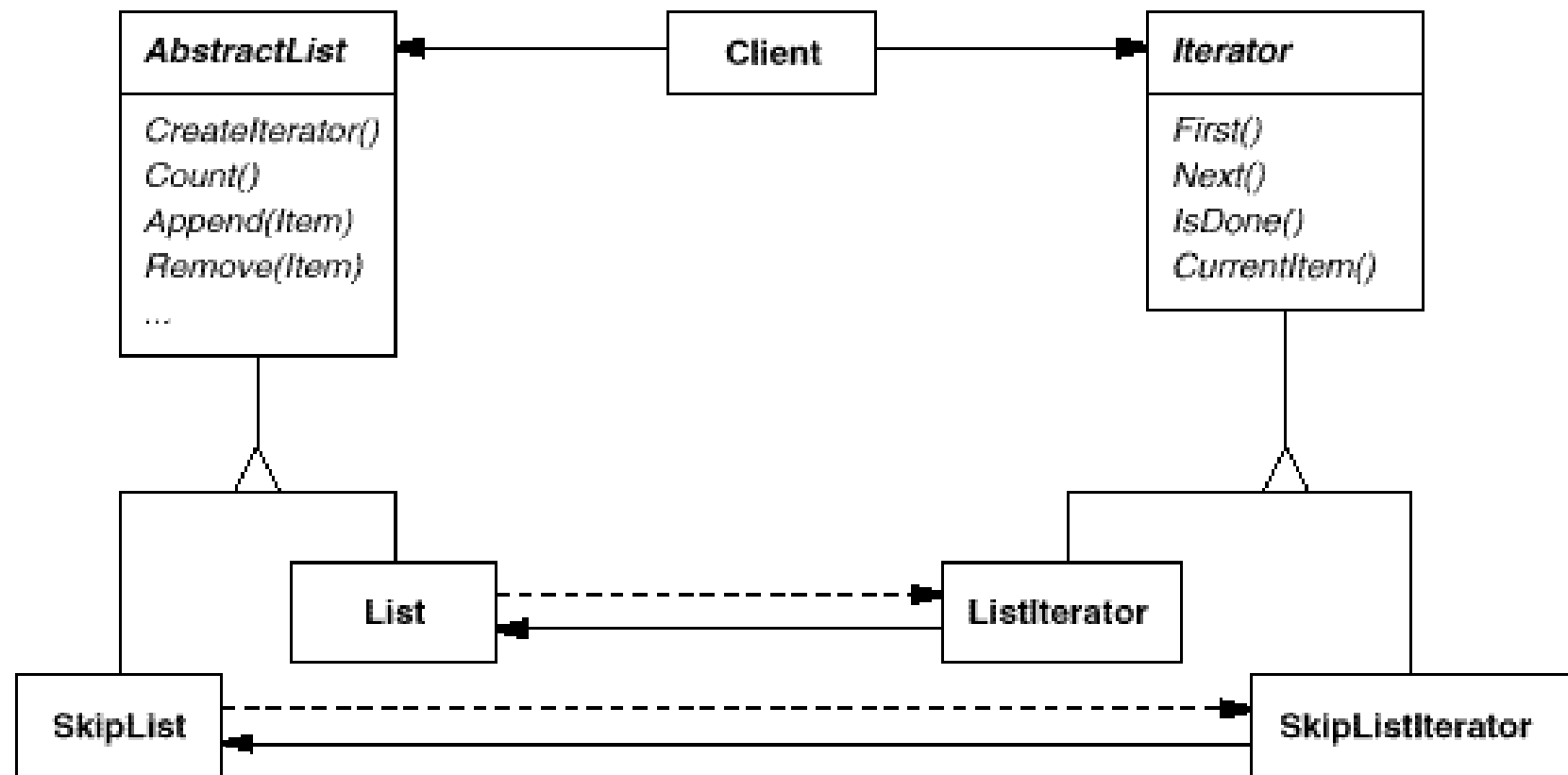
10 – Iterator without Aggregate

```
private class Itr implements Iterator<Channel> {  
  
    private int current = 0;  
    public boolean hasNext() {  
        return current < CCTVChannelSize;  
    }  
    public Channel next() {  
        switch (current++) {  
            case 0:  
                return CCTV1;  
            case 1:  
                return CCTV2;  
            case 2:  
                return CCTV3;  
            case 3:  
                return CCTV4;  
            case 4:  
                return CCTV5;  
            default:  
                return null;  
        }  
    }  
}
```



Structure







Participants

■ Iterator

- Defines an interface for accessing and traversing elements.

■ ConcreteIterator

- 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.

■ ConcreteAggregate

- Implements the **Iterator creation interface** to return an instance of the proper **ConcreteIterator**.
-



Consequences – advantages

- It supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways.
 - It supports multiple traversals of an aggregate. Many **iterators** can traverse the aggregate together.
 - **Iterators** simplify the **Aggregate** interface.
 - More than one traversal can be pending on an aggregate.
-



Consequences – drawbacks

- Because **Iterators** relay on linear traversal, the traversed aggregate seems to be an ordered sequence, on the contrary, it is not true in most cases.
 - The elements returned by **Iterators** is implicit type (object or general type) . So that client must know the true type of the elements explicitly.
-



Applicability

- To access an aggregate object's contents without exposing its internal representation.
 - To support multiple traversals of aggregate objects.
 - To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).
-



Implementation 1:

Where the **concrete iterator** is defined?

- **Public Iterator: Concrete iterator** is defined as a class independent from aggregate.
 - More straightforward;
 - Polymorphic iteration;
 - Can storing multiple cursor for different clients.
 - Need the aggregate expose the details, thus break the encapsulation.
 - **Private Iterator: Concrete iterator** is defined as a inner class in the aggregate.
 - Less straightforward;
 - Protect the encapsulation of aggregate;
 - Suggested in most cases.
-



Implementation 2: Who controls the iteration?

- **Active Iterator (External Iterator):** The client controls the iteration;
 - Clients that use an **active iterator** must advance the traversal and request the next element explicitly from the **iterator**.
 - more flexible than **passive iterators**;
 - **Passive Iterator (Internal Iterator):** The **iterator** controls the iteration;
 - The client hands an **passive iterator** an operation to perform, and the **iterator** applies that operation to every element in the aggregate.
 - Easier to use, because it define the iteration logic for you.
-



Implementation 3:

Who defines the traversal algorithm?

- The aggregate might define the traversal algorithm and use the **iterator** to store just the state of the iteration (**cursor**), it points to the current position in the aggregate.
 - The **iterator** is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates.
 - Defining the **iterator** in aggregate's the inner class if traversal algorithm might need to access the private variables of the aggregate.
-



Implementation 3:

How robust is the **iterator**?

- It can be dangerous to modify an aggregate while you're traversing it.
 - **Copied Iterator**: A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.
 - **Robust iterator**: Ensures that insertions and removals won't affect traversal, and it does it without copying the aggregate.
 - Robust **iterator** rely on registering the **iterator** with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of **iterators** it has produced, or it maintains information internally to ensure proper traversal.
-



Static Iterator and Dynamic Iterator

- **Static Iterator:** A **copied iterator** which contains a snapshot of the aggregate when **iterator** is created. New changes are invisible to the traversal approach.
 - **Dynamic Iterator:** **Dynamic Iterator is opposed to the static one.** Any changes to the aggregate are allowed and available when traversing the aggregate.
 - Completely **Dynamic Iterator** is not easy to be implemented.
-



Fail Fast

- **Fail-fast** is a property of a system or module with respect to its response to failures.
 - A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure.
 - Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly-flawed process.
 - **FailFast Iterator** throws an exception when the aggregate is changed during iteration.
-



Implementation 4: Additional **Iterator** operations.

- The minimal interface to **Iterator** consists of the operations **First**, **Next**, **IsDone**, and **CurrentItem**.
 - Some additional operations might prove useful.
 - **Frist**
 - **Last**
 - **Previous**
 - **SkipTo**
-



Implementation 4: Additional **Iterator** operations

■ **Filter Iterator**

- A common iterator traverse each element of an aggregate.
 - A **filter iterator** compute the element of the aggregate and return the elements which match a certain condition.
-



Implementation 5: polymorphic iterators

- **Polymorphic iterators** allows the traversal algorithm operate on different aggregates which is assigned dynamically.
 - **Polymorphic iterators** have a drawback that client is responsible for deleting them.
-



Implementation 7:

Iterators for composites.

- External **iterators** can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates.
 - An **active (external) iterator** has to store a path through the Composite to keep track of the current object.
 - It's easier just to use an **passive (internal) iterator**. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.
 - Composites often need to be traversed in more than one way.
 - Pre-order, post-order, in-order, and breadth-first traversals are common.
-



Implementation 8: Null iterators.

- A **Null Iterator** is a degenerate **iterator** that's helpful for handling boundary conditions.
 - By definition, a **Null Iterator** is always done with traversal; that is, its **IsDone** operation always evaluates to true.
 - **NullIterator** can make traversing tree-structured aggregates (like Composites) easier.
-

Example: Java Collection

- Interface `Collection<E>`

- `Vector`, `ArrayList<E>`, `LinkedList<E>`, `Stack<E>`, `Queue<E>`, `HashSet<E>`, `TreeSet<E>`, `HashMap<K,V>`, `TreeMap<K,V>`, `Hashtable<K,V>`.


```
java.util.concurrent.LinkedBlockingQueue
java.util.concurrent.BlockingQueue
java.util.concurrent.ArrayBlockingQueue
java.util.concurrent.LinkedBlockingQueue
java.util.concurrent.PriorityBlockingQueue
- -
```

- Interface `Iterator<E>`

- Interface `ListIterator<E>` **extends** `Iterator<E>`

- Interface `Iterable<T>`

```
/** Implementing this interface allows an object to be the target of
 * the "foreach" statement.
 * @since 1.5
 */
public interface Iterable<T> {
    Iterator<T> iterator();
}
```


```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

```
public interface ListIterator<E> extends Iterator<E> {  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove();  
    void set(E e);  
    void add(E e);  
}
```



ListIterator<E>

- **void add(E o)**
 - Inserts the specified element into the list.
 - The element is inserted immediately before the next element that would be returned by next, if any, and after the next element that would be returned by previous, if any.
 - The new element is inserted before the implicit cursor: a subsequent call to next would be unaffected, and a subsequent call to previous would return the new element.
 - **void remove()**
 - Removes from the list the last element that was returned by next or previous
 - This call can only be made once per call to next or previous.
 - It can be made only if ListIterator.add has not been called after the last call to next or previous.
-



java.util.AbstractList
java.util.AbstractList.Itr
java.util.AbstractList.ListItr


```
package java.util;


public abstract class AbstractList<E> extends AbstractCollection<E> implements
    List<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    public ListIterator<E> listIterator() {
        return listIterator(0);
    }

    public ListIterator<E> listIterator(final int index) {
        if (index < 0 || index > size())
            throw new IndexOutOfBoundsException("Index: " + index);

        return new ListItr(index);
    }
}
```






```
private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastRet = -1;
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor != size();
    }

    public E next() {
        checkForComodification();
        try {
            E next = get(cursor);
            lastRet = cursor++;
            return next;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
}
```



```
public void remove() {
    if (lastRet == -1)
        throw new IllegalStateException();
    checkForComodification();

    try {
        AbstractList.this.remove(lastRet);
        if (lastRet < cursor)
            cursor--;
        lastRet = -1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException e) {
        throw new ConcurrentModificationException();
    }
}

final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
}
```

```

private class ListItr extends Itr implements ListIterator<E> {
    ListItr(int index) {
        cursor = index;
    }
    public boolean hasPrevious() {
        return cursor != 0;
    }
    public E previous() {
        checkForComodification();
        try {
            int i = cursor - 1;
            E previous = get(i);
            lastRet = cursor = i;
            return previous;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    public int nextIndex() {
        return cursor;
    }
    public void set(E e) {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();

        try {
            AbstractList.this.set(lastRet, e);
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
    public void add(E e) {
        checkForComodification();
        try {
            AbstractList.this.add(cursor++, e);
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
}

```

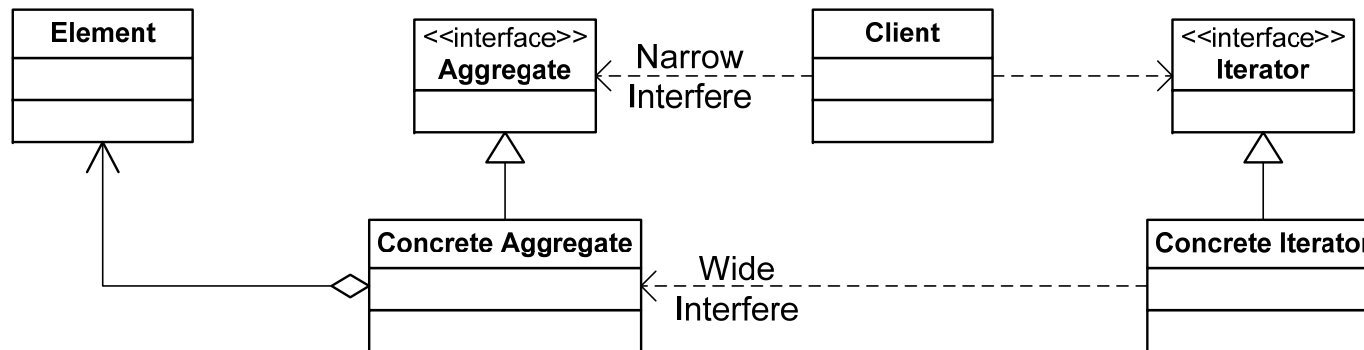


Extension 1: **Wide** and **Narrow** interface of an aggregate

- Wide interfere: the interfere which allow the client modify the elements of aggregate.
 - White-box Aggregate
 - Narrow interfere: the interfere which only allow the client traverse the elements of aggregate.
 - Black-box Aggregate
-

Extension 1

- An aggregate should provide an wide interface to its **iterator**, and narrow interface to its clients.
- Let **concrete iterator** be and inner class of an aggregate



Extension 2: Enumeration<E> vs Iterator<E>

| <<interface>> Iterator |
|--|
| + <i>hasNext()</i> : <i>boolean</i> + <i>next()</i> : <i>E</i> + <i>remove()</i> |

| <<interface>> Enumeration |
|--|
| + <i>hasMoreElements()</i> : <i>boolean</i> + <i>nextElement()</i> <i>E</i> |

- **Iterator** simplifies the interfaces;
- **Enumeration** does not supports Fail Fast;
- **Enumeration** is twice as fast as **Iterator** and uses very less memory;
- **Iterator** is much safer as compared to **Enumeration** because it always denies other threads to modify the collection object which is being iterated by it;



Let's go to next...