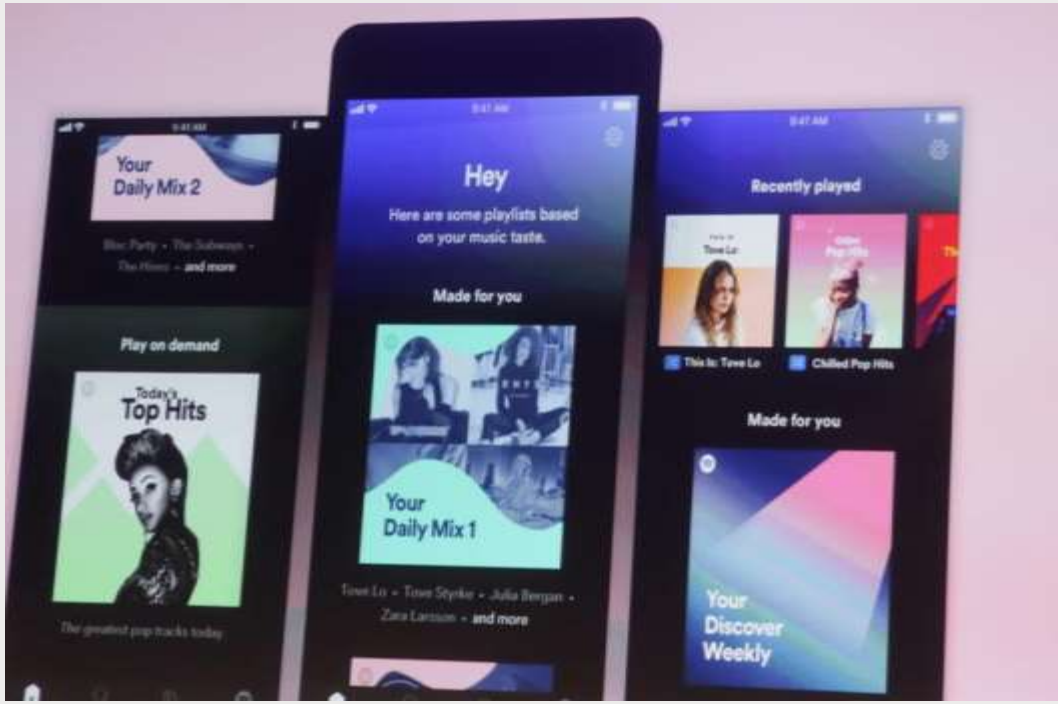
ITERATOR

Motivation - problems

- Library
  - *We want to see books in the catalog system.*
  - *The collection of books is too large.*

- Music streaming platform
  - *Songs in the music library.*
  - *See songs in a recommendation playlist.*
  - *The collection could be practically endless.*

- Social media
  - *We want to see news, images on the feed.*
  - *The collection is too big.*
  - *New posts may be added.*

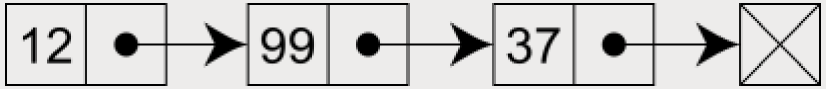


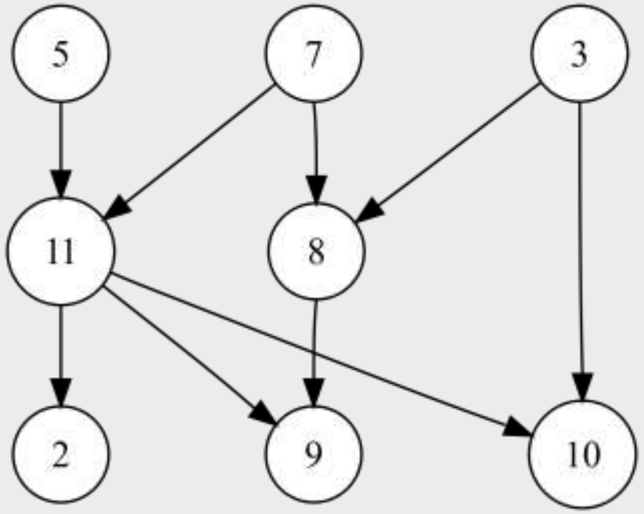
Motivation - solution

- Iterator pattern design
- Items shown in way which is
  - *Manageable*
  - *Easy to navigate*
  - *Easy Adding/removing*
- Iterator provides
  - *Display the items one at a time*
    - Posts, Songs, Books
  - *Keeping track of the current item*
  - *Access next item*
  - *New item added*

Motivation

- Aggregated collections
  - *Array*
  - *List*
  - *Dag*
  - *Linked list*
  - *Dictionary*
  - *HashSet*





Index	0	1	2	3	4
number	10	20	30	40	50

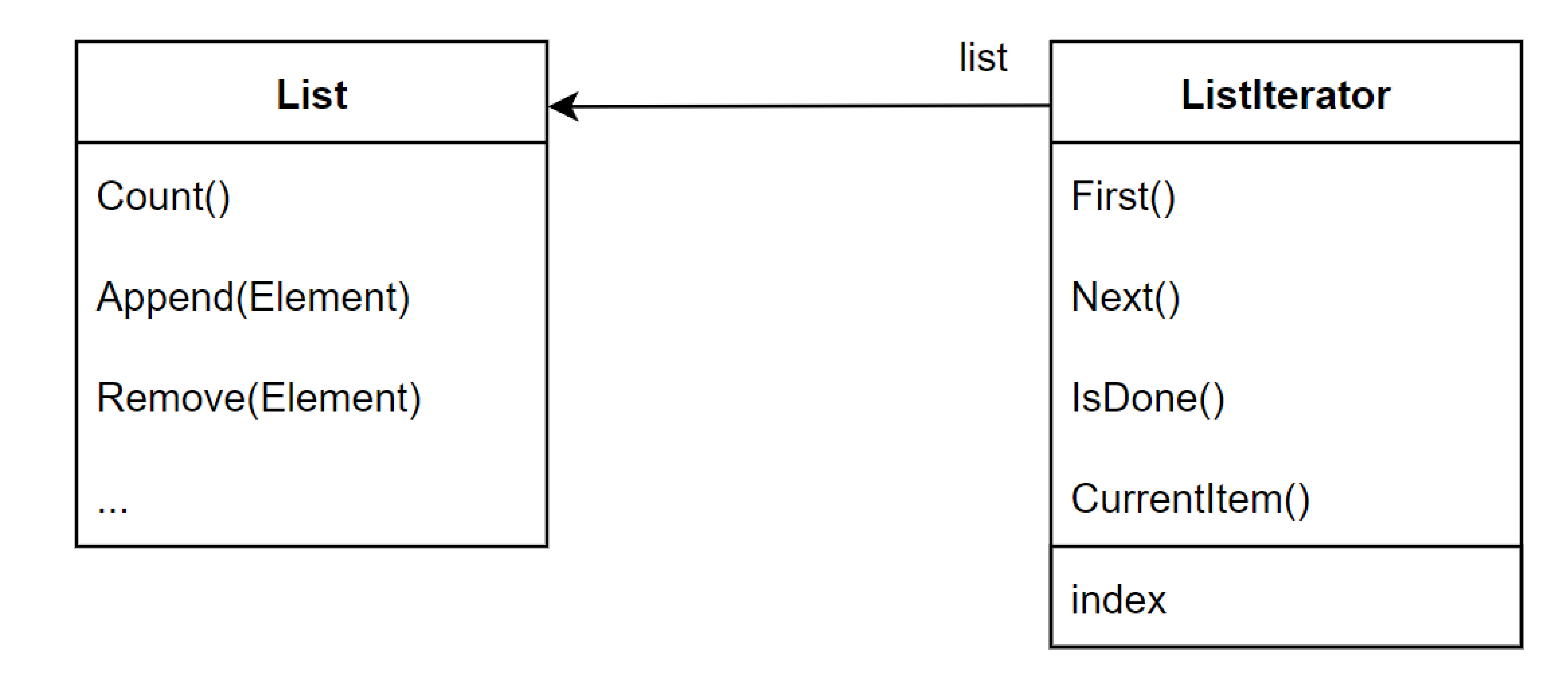
What is an iterator ?

- Simple and united interface for accessing items of a collection
- Without exposing its internal structure (encapsulation)
- Iteration interface taken out of collection interface
  - *Lets us traverse collection in different ways*
  - *Change in collection without need to change our code*
  - *Allows for more traversals at the same time*
  - *Lazy evaluation*

Interface Responsibilities

- Keeping track of the current element
- Knows already traversed elements
- Lets us traverse next element

Example – basic relationship



- Sequential access
- Iterator chooses the way to iterate
  - *FilteringListIterator*
  - *Provides desired items*

- Other possible methods
  - *Previous()*
  - *SkipTo()*
  - *Reset()*

Example code

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
  ListIterator(const List<Item>* aList);
  virtual void First();
  virtual void Next();
  virtual bool IsDone() const;
  virtual Item CurrentItem() const;
private:
  const List<Item>* _list;
  long _current;
};
```

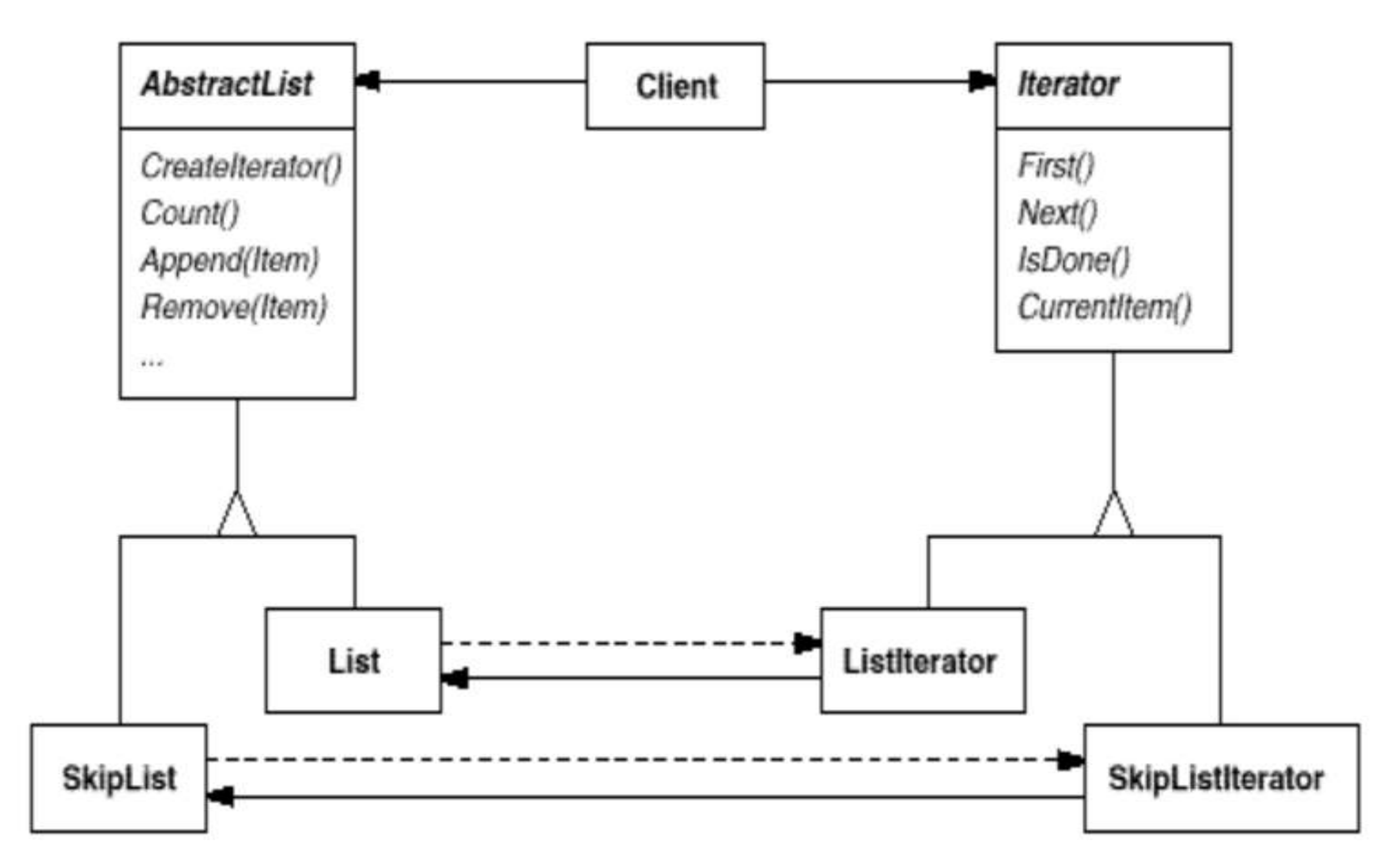
```
struct Employee {
  std::string _name;

  void Print() {
    std::cout << _name << std::endl;
  }
};

void PrintEmployees(ListIterator<Employee*> &it) {
  for(it.First(); !it.IsDone(); it.Next()) {
    it.CurrentItem()->Print();
  }
}
```

- PrintEmployees still coupled with List Collection
  - *Uses ListIterator*
  - *We want to change the collection*
    - Can’t without changing our code

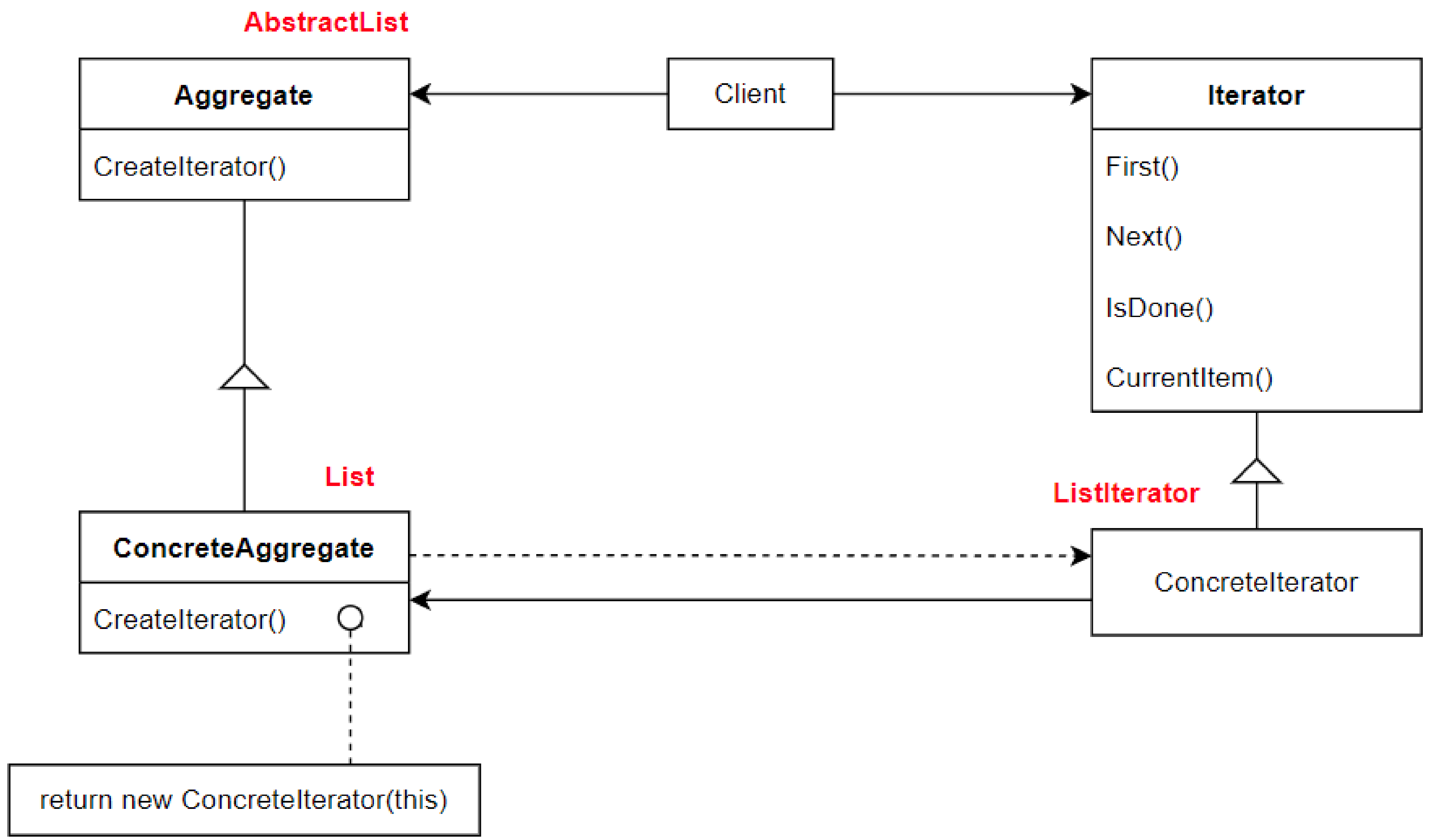
Generalization



- Abstract Iterator
  - *Common interface for collection*
- Abstract List
  - *Common interface for iterating over a collection*
- No longer dependand on a specific collection

How do we create the iterator ?

- We want the code to be independent of the concrete collection (polymorphic iteration)
- Collections are responsible for creating iterator
- CreateIterator
- Factory Method
- Connects the two hierarchies



- Interfaces and their implementation
- 2 separate hierarchies

Iterator Factory Method

- Connects the two hierarchies
- Dynamic iterator allocation
- Collection’s responsibility to create the iterator
- Our code is independent of specific collection usage

C++11

```
auto it = employees.begin();
it->PrintEmployees();
```

JAVA

```
Iterator<Employee> it = employees.iterator();
PrintEmployees(it);
```

C#

```
IEnumerator<Employee> it = employees.GetEnumerator();
PrintEmployees(it);
```

Consequences

- Supports variations in the traversal of an aggregate
- Simpliefies the Aggregate interface
- More than one traversal pending

Implementation

```
public class Node {
    public int data;
    public Node left, right;
}
```

- Variation in traversal

```
public class InorderIterator {
    private Node current, rightMost;
    public InorderIterator(Node root);
    public bool HasNext();
    public Node Next();
    public Node CurrentItem();
}
```

```
public class PostorderIterator {
    private Node current, rightMost;
    public PostorderIterator (Node root);
    public bool HasNext();
    public Node Next();
    public Node CurrentItem();
}
```

Division

- Who controls the process of iteration ?

1. External (pull) iterator

- More flexible
- User controls the traversal
- User asks for next item
- Harder to implement
- Easier to use

```
iterator = words.GetEnumerator();
while(iterator.MoveNext()){
    Console.WriteLine(iterator.Current);
}
```

2. Internal (push) iterator

- Less flexible
- Iterator/collection controls the traversal
- User specifies action on each element
- Easier to implement
- Harder to use
- Especially stronger in languages which support lambda functions and expressions

```
Words.ForEach(
    x => Console.WriteLine(x)
);
```

Division

- Who defines the traversal algorithm ?

1. Collection - Cursor

- Defined by collection.
- Iterator only stores the state of iteration.
- Next invoked on collection.
- Next changes state of iterator.

2. Iterator

- Easier to change the iteration algorithm.
- Reusability.
- Violates the encapsulation of collection.

Solution ?

C# - private nested class

C++ - friend class

Polymorphism

```
template <typename Item>
class AbstractList {
public:
  virtual Iterator<unique_ptr<Item>> CreateIterator() const = 0;
  // ...
};
```

```
template <typename Item>
auto <Item>::CreateIterator() const ->
unique_ptr<Iterator<Item>> {
  return make_unique<ListIterator<Item>>(this);
}
```

```
unique_ptr<AbstractList<unique_ptr<Employee>>> employees;
// ...
unique_ptr<Iterator<unique_ptr<Employee>>> iterator =
employees->CreateIterator();

PrintEmployees(*iterator);
```

```
void PrintEmployees (Iterator<unique_ptr<Employee>>& it) {
  for (it.First(); !it.IsDone(); it.Next()) {
    it.CurrentItem()->Print();
  }
}
```

Aditional operations

- Minimal interface
  - *First, Next, IsDone, CurrentItem*

- Previous
- SkipTo

- Dangerous to modify the collection inside iteration
  - *Iteration over copy of the collection – too expensive*
  - *Robust iterator*
    - Modifications do not interfere with traversal
    - Without copying collections
    - Collection adjusts the state of iterator
    - Or maintains information

- What can be iterated ?
- Virtual or endless collections
  - *Data from external source*
- Transform iterator
  - *Function applied to value*
- Numbers generated by iterator – items not in memory
  - *Fibonacci*
  - *Ranom*

Iterators for composites

Often need to be traversed in more than one way. (inorder, preorder, postorder, ...)

- External iterators – difficult to implement over recursive collections
  - *Need to store the recursive path*
- Internal iterator easier to implement
  - *Can remember the path by calling itself*
  - *Stored in the call stack*
- Smarter to use the cursor
  - *If the collection provides interface for moving between nodes*

NullIterator

- Handling edge cases
- IsDone is always true

- Traversing over a tree
- Leafs always return NullIterator

- Makes the traversal more uniform

Related patterns

- Composite – Iterators often applied to various recursive structures
- Factory Method – Polymorphic iterators
- Momento – Captures the state of the iterator
- Proxy – resource management