Behavioral Design Pattern

Instructor: Mehroze Khan

Behavioral Design Pattern

- Behavioral design patterns are concerned with algorithms and the **assignment of responsibilities** between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the **patterns of communication** between them.

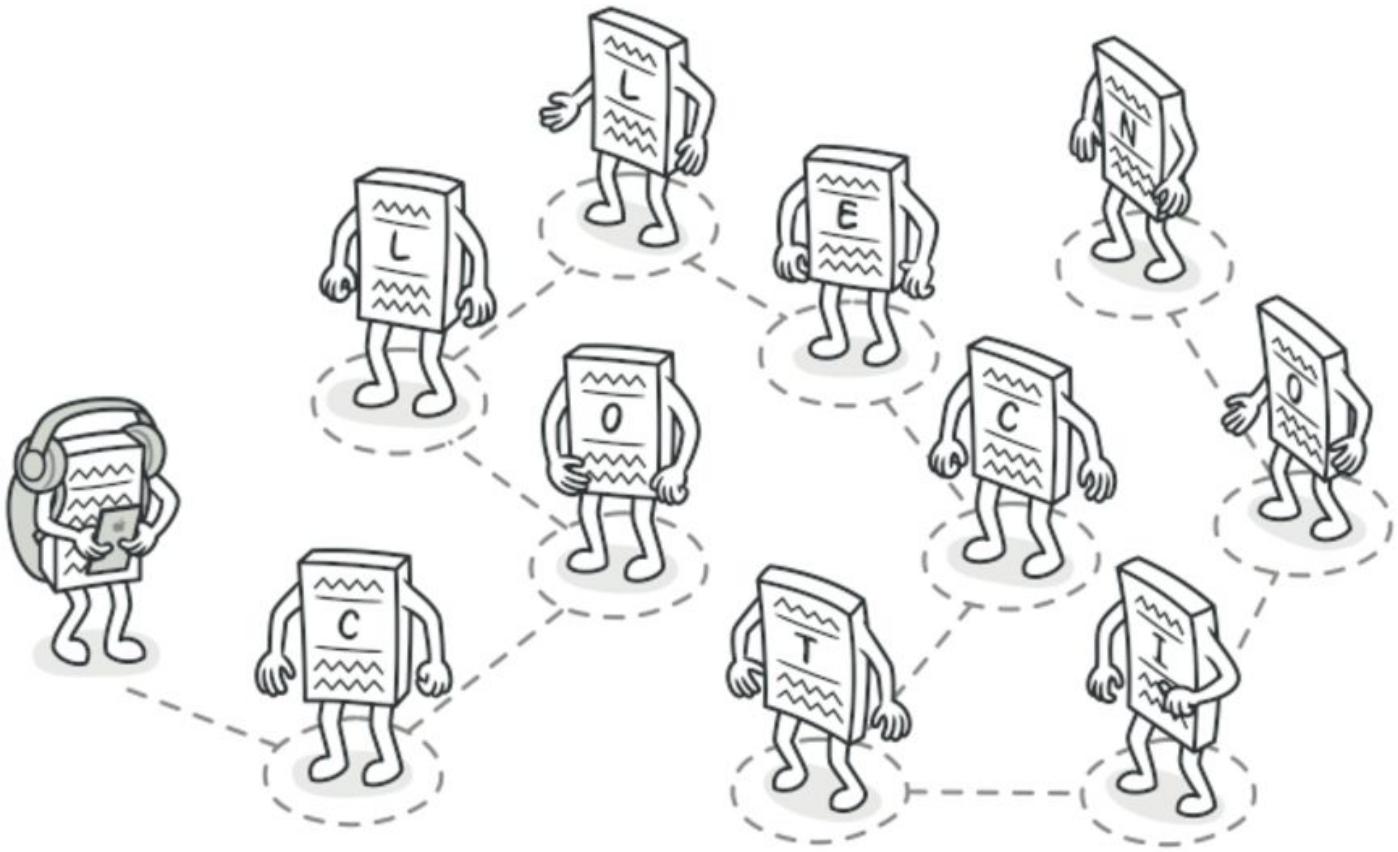
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Iterator Pattern

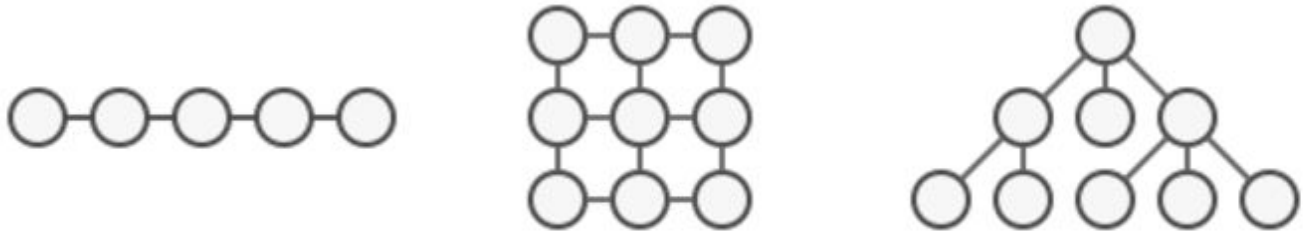
Iterator Pattern

Intent:

- **Iterator** is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



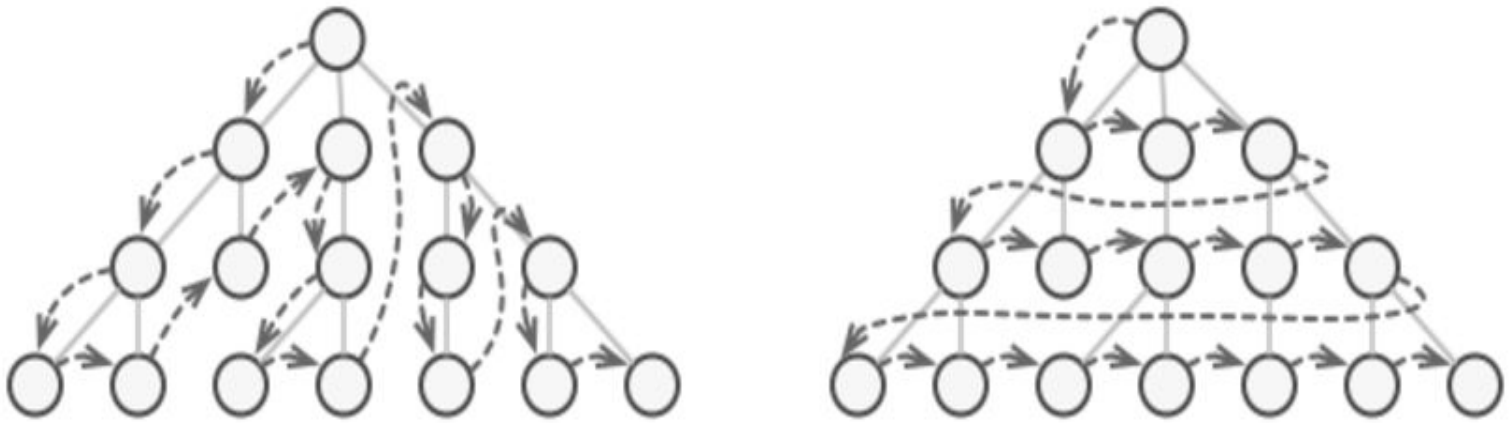
Problem



Various types of collections.

- Most collections store their elements in simple lists. However, some of them are based on **stacks**, **trees**, **graphs** and other complex data structures.
- But no matter how a collection is structured, there should be a way to go through each element of the collection without accessing the same elements over and over.
- This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements.
- But how do you **sequentially traverse elements of a complex data structure**, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.

Problem

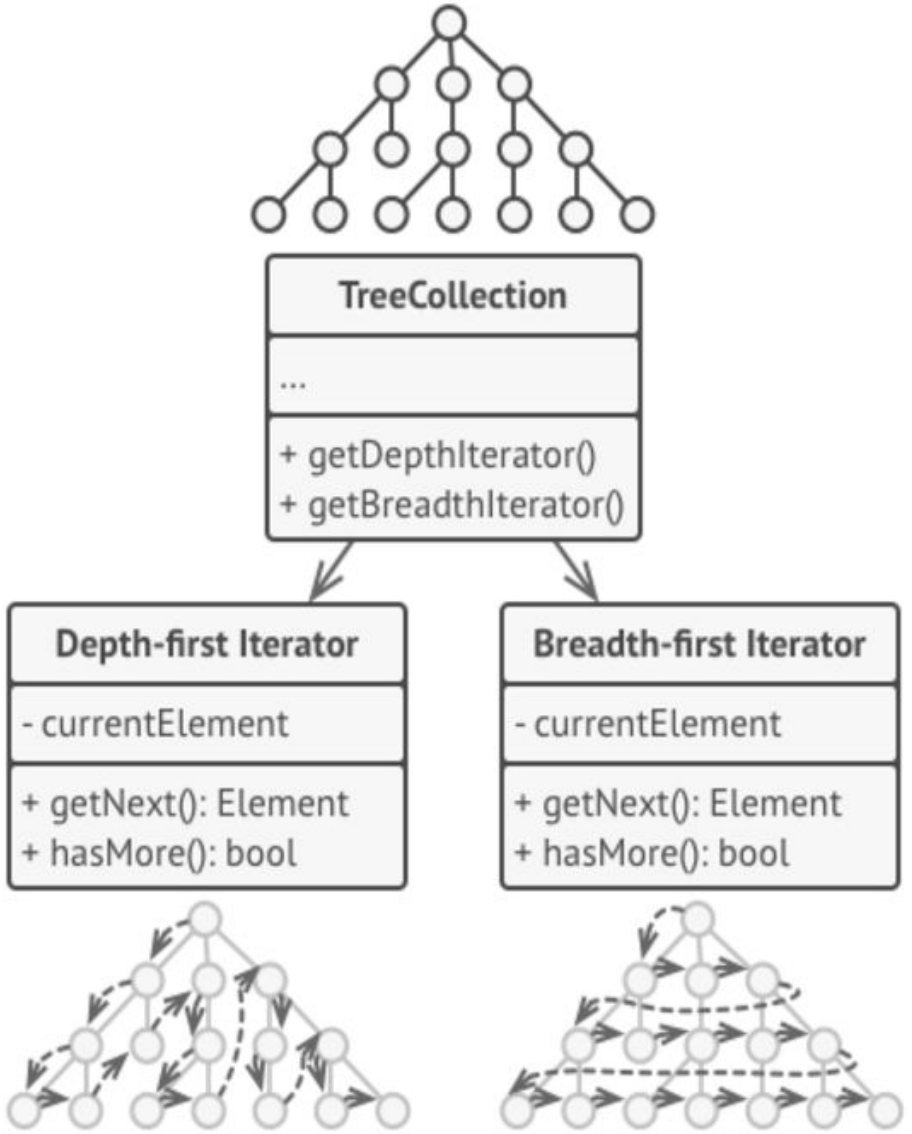


The same collection can be traversed in several different ways.

- Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is **efficient data storage**.
- Client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

Solution

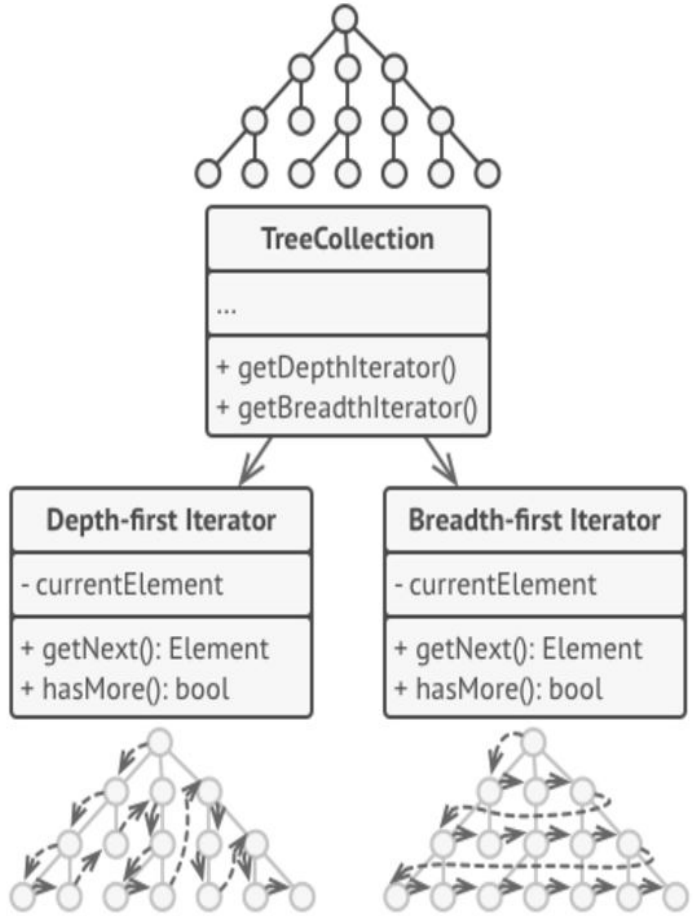
- The main idea of the Iterator pattern is to **extract the traversal behavior of a collection** into a separate object called an **iterator**.



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

Solution

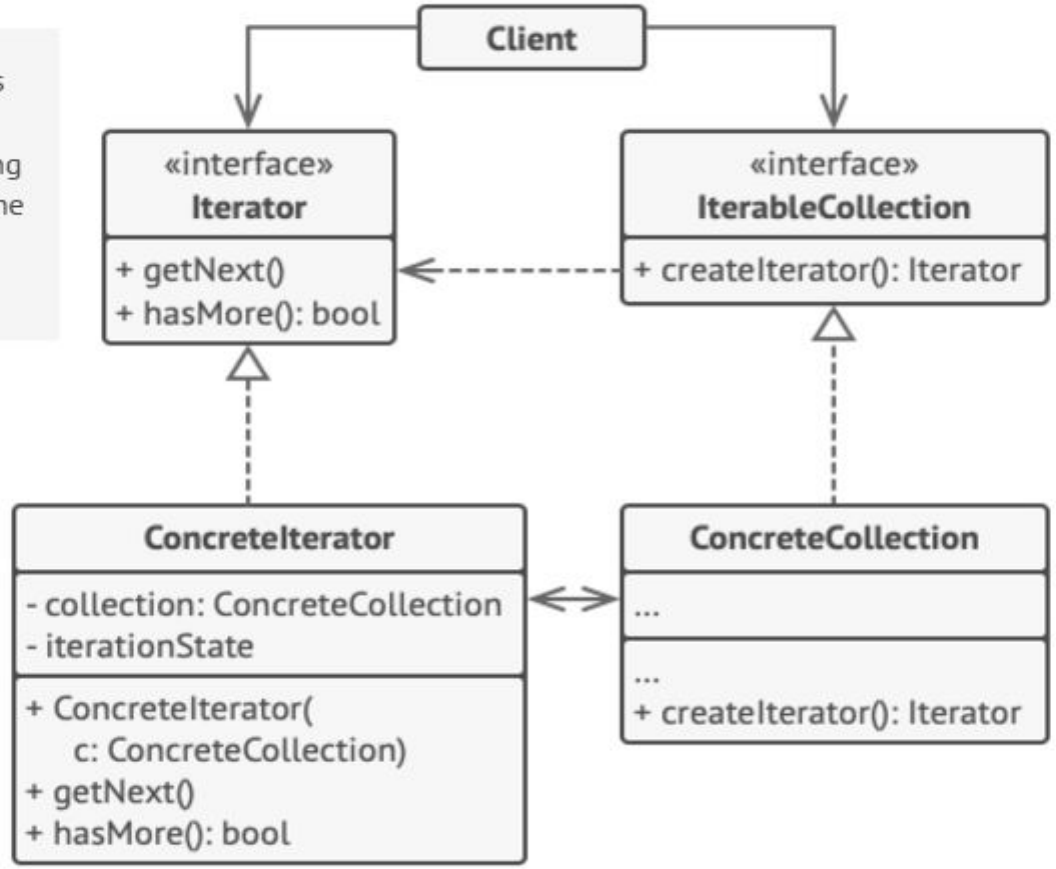
- An iterator object encapsulates all of the **traversal details**, such as the current position and how many elements are left till the end.
- Several iterators can go through the same collection at the same time, independently of each other.
- Iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.
- All iterators must implement the **same interface**. This makes the client code compatible with any collection type or any traversal algorithm.
- If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.



Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

Structure

1 The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.



2 **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.

3 The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

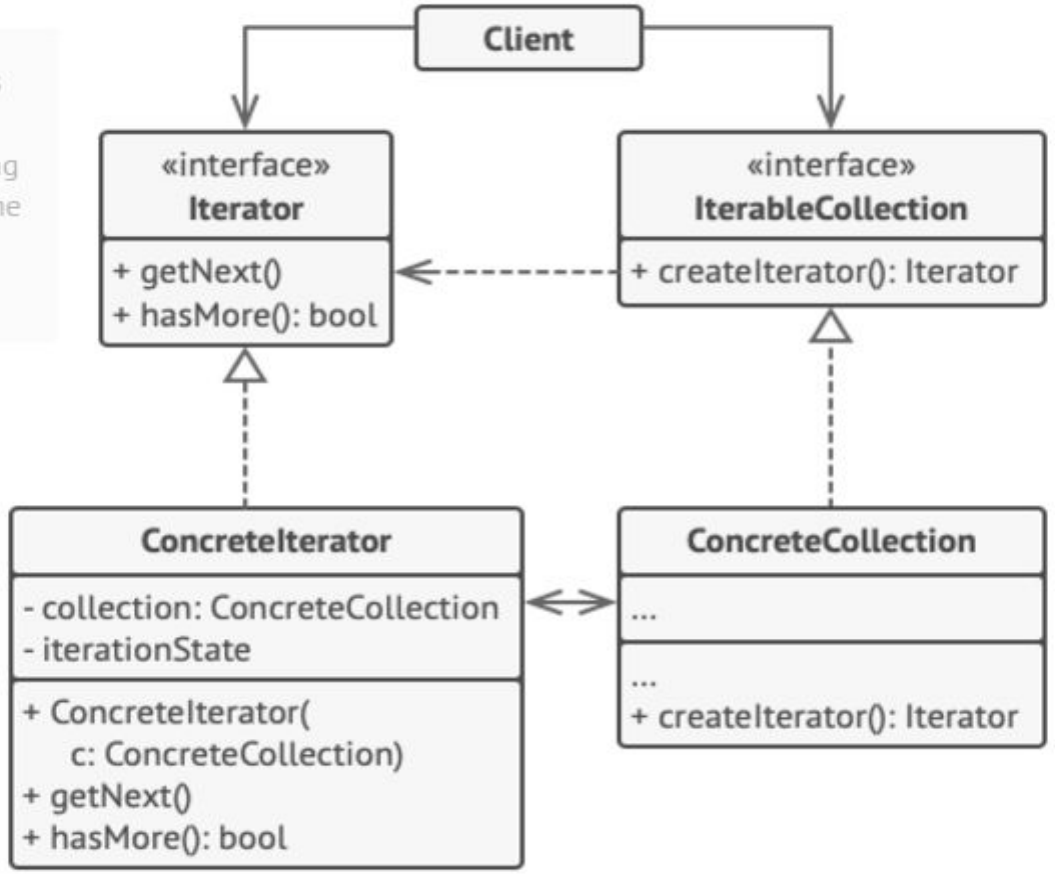
4 **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

5 The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structure

1 The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.



2 **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.

3 The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

4 **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

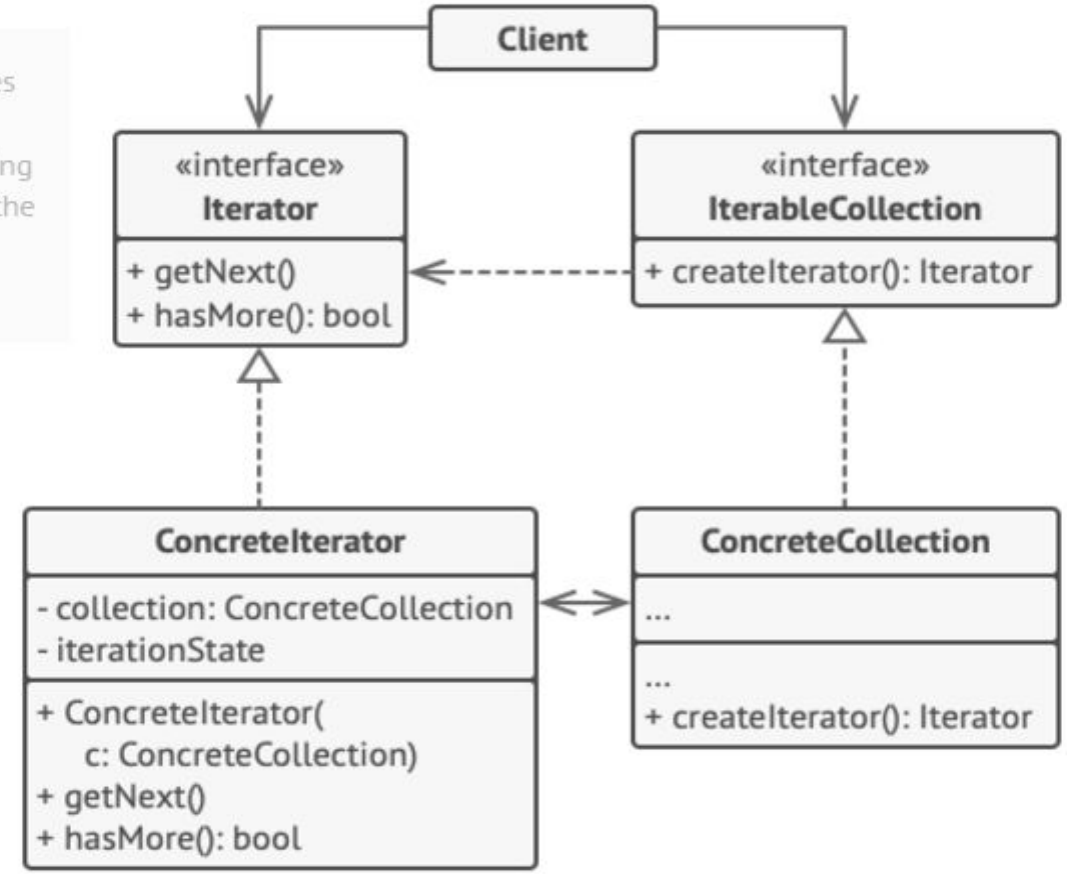
5 The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structure

1 The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

2 **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.



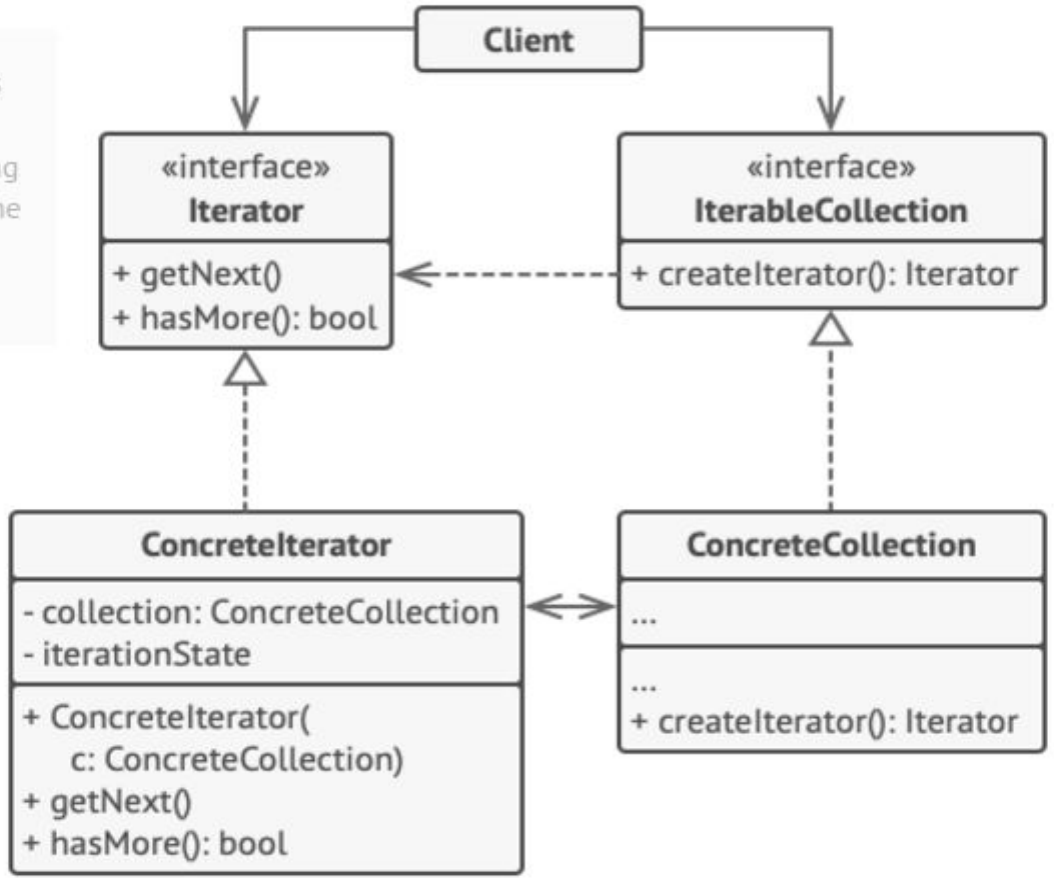
3 The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

4 **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

5 The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structure



1 The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

2 **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.

3 The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

4 **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

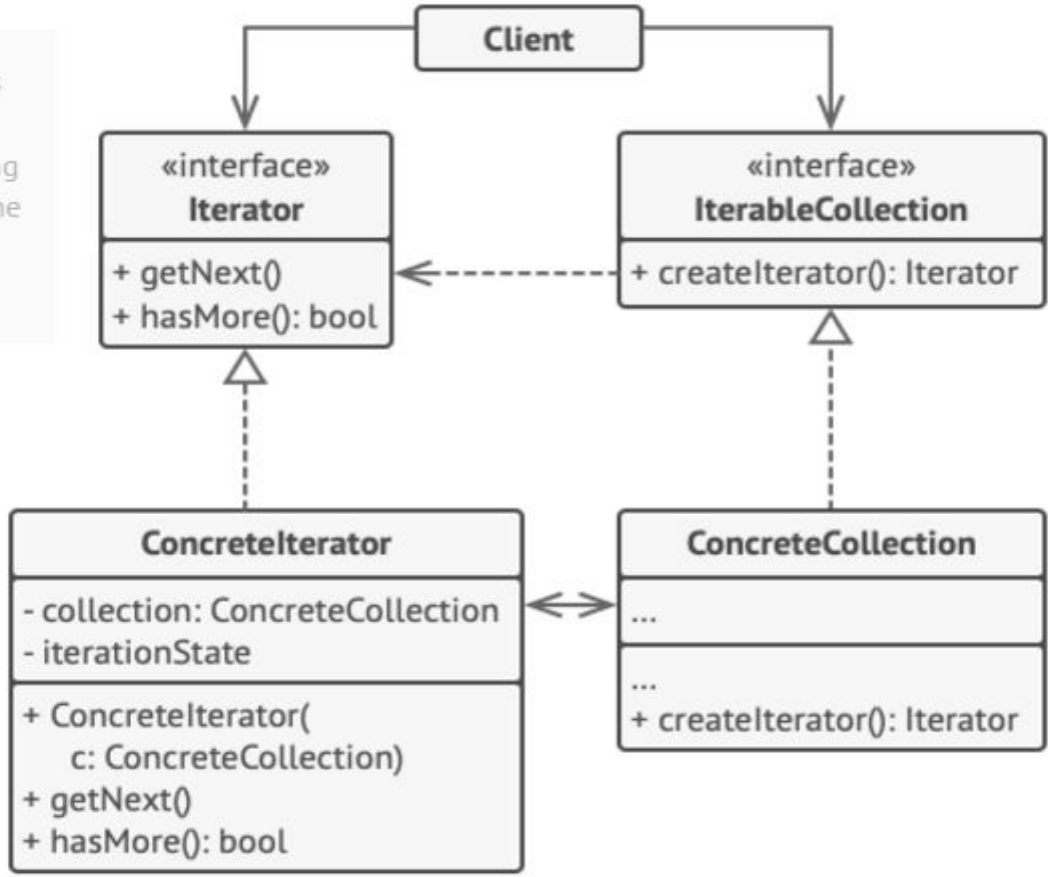
5 The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structure

1 The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

2 **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.



3 The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

4 **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

5 The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Example

```cpp
// Iterator Interface
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual int next() = 0;
};

// Collection Interface
class Collection {
public:
    virtual Iterator* createIterator() = 0;
};
```

Example

```
// Concrete Collection: VectorCollection
class VectorCollection : public Collection {
private:
    vector<int> items;

public:
    void addItem(int item) {
        items.push_back(item);
    }

    Iterator* createIterator() override {
        return new VectorIterator(items);
    }
};
```

Example

```
// Iterator for VectorCollection
class VectorIterator : public Iterator {
private:
    vector<int>& collection;
    size_t position;

public:
    VectorIterator(vector<int>& collection) {
        collection = collection;
        position = 0;
    }

    bool hasNext() override {
        return position < collection.size();
    }

    int next() override {
        if (hasNext()) {
            return collection[position++];
        } else {
            throw out_of_range("No more elements.");
        }
    }
};
```

Example

```
// Concrete Collection: StackCollection
class StackCollection : public Collection {
private:
    stack<int> items;

public:
    void addItem(int item) {
        items.push(item);
    }

    Iterator* createIterator() override {
        return new StackIterator(items);
    }
};
```

Example

```
// Iterator for StackCollection
class StackIterator : public Iterator {
private:
    stack<int> tempStack;

public:
    StackIterator(stack<int>& collection) {
        // Copy elements to a temporary stack for iteration
        tempStack = collection;
    }

    bool hasNext() override {
        return !tempStack.empty();
    }

    int next() override {
        if (hasNext()) {
            int value = tempStack.top();
            tempStack.pop();
            return value;
        } else {
            throw out_of_range("No more elements.");
        }
    }
};
```

Example

```
// Client Code
int main() {
    // Using VectorCollection
    VectorCollection vectorCollection;
    vectorCollection.addItem(1);
    vectorCollection.addItem(2);
    vectorCollection.addItem(3);

    cout << "VectorCollection Items:" << endl;
    Iterator* vectorIterator = vectorCollection.createIterator();
    while (vectorIterator->hasNext()) {
        cout << vectorIterator->next() << " ";
    }

    // Using StackCollection
    StackCollection stackCollection;
    stackCollection.addItem(10);
    stackCollection.addItem(20);
    stackCollection.addItem(30);

    cout << "StackCollection Items:" << endl;
    Iterator* stackIterator = stackCollection.createIterator();
    while (stackIterator->hasNext()) {
        cout << stackIterator->next() << " ";
    }
}
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.

Helping Links:

- https://refactoring.guru/design-patterns/