WIKIPEDIA

Iterator pattern

In object-oriented programming, the **iterator pattern** is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

For example, the hypothetical algorithm *SearchForElement* can be implemented generally using a specified type of iterator rather than implementing it as a container-specific algorithm. This allows *SearchForElement* to be used on any container that supports the required type of iterator.

Contents

Overview

The Iterator [1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Iterator design pattern solve? [2]

- The elements of an aggregate object should be accessed and traversed without exposing its representation (data structures).
- New traversal operations should be defined for an aggregate object without changing its interface.

Defining access and traversal operations in the aggregate interface is inflexible because it commits the aggregate to particular access and traversal operations and makes it impossible to add new operations later without having to change the aggregate interface.

What solution does the Iterator design pattern describe?

- Define a separate (iterator) object that encapsulates accessing and traversing an aggregate object.
- Clients use an iterator to access and traverse an aggregate without knowing its representation (data structures).

Different iterators can be used to access and traverse an aggregate in different ways.
New access and traversal operations can be defined independently by defining new iterators.

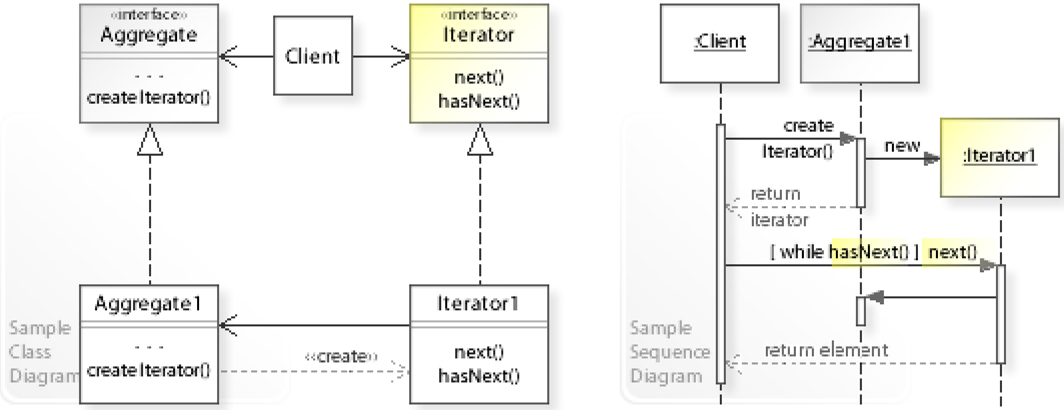
See also the UML class and sequence diagram below.

Definition

The essence of the Iterator Pattern is to "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.".[3]

Structure

UML class and sequence diagram

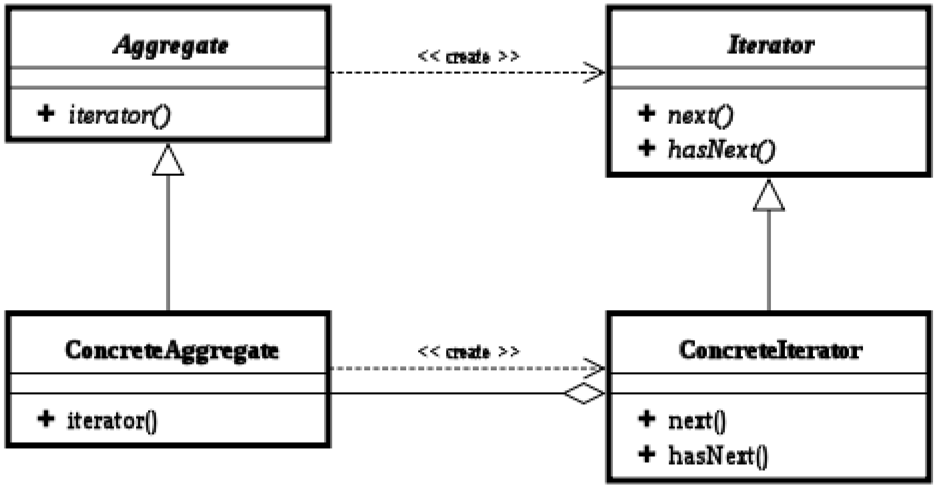


A sample UML class and sequence diagram for the Iterator design pattern. [4]

In the above UML class diagram, the `Client` class refers (1) to the `Aggregate` interface for creating an `Iterator` object (`createIterator()`) and (2) to the `Iterator` interface for traversing an `Aggregate` object (`next(),hasNext()`). The `Iterator1` class implements the `Iterator` interface by accessing the `Aggregate1` class.

The UML sequence diagram shows the run-time interactions: The `Client` object calls `createIterator()` on an `Aggregate1` object, which creates an `Iterator1` object and returns it to the `Client`. The `Client` uses then `Iterator1` to traverse the elements of the `Aggregate1` object.

UML class diagram



The iterator pattern

Language-specific implementation

Some languages standardize syntax. C++ and Python are notable examples.

C#

.NET Framework has special interfaces that support a simple iteration: `System.Collections.IEnumerator` over a non-generic collection and `System.Collections.Generic.IEnumerator<T>` over a generic collection.

C# statement `foreach` is designed to easily iterate through the collection that implements `System.Collections.IEnumerator` and/or `System.Collections.Generic.IEnumerator<T>` interface. Since C# v2, `foreach` is also able to iterate through types that implement `System.Collections.Generic.IEnumerable<T>` and `System.Collections.Generic.IEnumerator<T>` [5]

Example of using `foreach` statement:

```
var primes = new List<int>{ 2, 3, 5, 7, 11, 13, 17, 19};
long m = 1;
foreach (var p in primes)
    m *= p;
```

C++

C++ implements iterators with the semantics of pointers in that language. In C++, a class can overload all of the pointer operations, so an iterator can be implemented that acts more or less like a pointer, complete with dereference, increment, and decrement. This has the advantage that C++ algorithms such as `std::sort` can immediately be applied to plain old memory buffers, and that there is no new syntax to learn. However, it requires an "end" iterator to test for equality, rather than allowing an iterator to know that it has reached the end. In C++ language, we say that an iterator models the iterator concept.

Java

Java has the `Iterator (https://docs.oracle.com/javase/10/docs/api/java/util/Iterator.html)` interface.

A simple example showing how to return integers between [start, end[using an `Iterator`

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class RangeIteratorExample {
    public static Iterator<Integer> range(int start, int end) {
        return new Iterator<>() {
            private int index = start;

            @Override
            public boolean hasNext() {
                return index < end;
            }

            @Override
            public Integer next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return index++;
            }
        };
    }

    public static void main(String[] args) {
        var iterator = range(0, 10);
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // or using a lambda
        iterator.forEachRemaining(System.out::println);
    }
}
```

As of Java 5, objects implementing the `Iterable (https://docs.oracle.com/javase/10/docs/api/java/lang/Iterable.html)` interface, which returns an `Iterator` from its only method, can be traversed using Java's foreach loop syntax. The `Collection (https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html)` interface from the Java collections framework extends `Iterable`.

Example of class `Family` implementing the `Iterable` interface:

```
import java.util.Iterator;
import java.util.Set;

class Family<E> implements Iterable<E> {
    private final Set<E> elements;

    public Family(Set<E> elements) {
      this.elements = Set.copyOf(elements);
    }

    @Override
    public Iterator<E> iterator() {
        return elements.iterator();
    }
}
```

The class `IterableExample` demonstrates the use of class `Family` :

```
public class IterableExample {
    public static void main(String[] args) {
        var weasleys = Set.of(
            "Arthur", "Molly", "Bill", "Charlie",
            "Percy", "Fred", "George", "Ron", "Ginny"
            );
        var family = new Family<>(weasleys);

        for (var name : family) {
            System.out.println(name + " Weasley");
        }
    }
}
```

Output:

```
Ron Weasley
Molly Weasley
Percy Weasley
Fred Weasley
Charlie Weasley
George Weasley
Arthur Weasley
Ginny Weasley
Bill Weasley
```

JavaScript

JavaScript, as part of ECMAScript 6, supports the iterator pattern with any object that provides a `next()` method, which returns an object with two specific properties: `done` and `value`. Here's an example that shows a reverse array iterator:

```
function reverseArrayIterator(array) {
    var index = array.length - 1;
    return {
       next: () =>
          index >= 0 ?
           {value: array[index--], done: false} :
           {done: true}
    }
}

const it = reverseArrayIterator(['three', 'two', 'one']);
console.log(it.next().value);  //-> 'one'
console.log(it.next().value);  //-> 'two'
console.log(it.next().value);  //-> 'three'
console.log(`Are you done? ${it.next().done}`);  //-> true
```

Most of the time, though, what you want is to provide Iterator[6] semantics on objects so that they can be iterated automatically via `for...of` loops. Some of JavaScript's built-in types such as `Array`, `Map`, or `Set` already define their own iteration behavior. You can achieve the same effect by defining an object's meta `@@iterator` method, also referred to by `Symbol.iterator`. This creates an Iterable object.

Here's an example of a range function that generates a list of values starting from `start` to `end`, exclusive, using a regular `for` loop to generate the numbers:

```
function range(start, end) {
  return {
    [Symbol.iterator]() { //#A
```

```
      return this;
    },
    next() {
      if(start < end) {
        return { value: start++, done:false }; //#B
      }
      return { done: true, value:end }; //#B
    }
  }
}

for(number of range(1, 5)) {
   console.log(number);   //-> 1, 2, 3, 4
}
```

The iteration mechanism of built-in types, like strings, can also be manipulated:

```
let iter = ['I', 't', 'e', 'r', 'a', 't', 'o', 'r'][Symbol.iterator]();
iter.next().value; //-> I
iter.next().value; //-> t
```

PHP

PHP supports the iterator pattern via the Iterator interface, as part of the standard distribution.[7] Objects that implement the interface can be iterated over with the `foreach` language construct.

Example of patterns using PHP:

```
<?php

// BookIterator.php

namespace DesignPatterns;

class BookIterator implements \Iterator
{
    private $i_position = 0;
    private $booksCollection;

    public function __construct(BookCollection $booksCollection)
    {
        $this->booksCollection = $booksCollection;
    }

    public function current()
    {
        return $this->booksCollection->getTitle($this->i_position);
    }
```

```php
    public function key()
    {
        return $this->i_position;
    }

    public function next()
    {
        $this->i_position++;
    }

    public function rewind()
    {
        $this->i_position = 0;
    }

    public function valid()
    {
        return !is_null($this->booksCollection->getTitle($this->i_position));
    }
}
```

```php
<?php

// BookCollection.php

namespace DesignPatterns;

class BookCollection implements \IteratorAggregate
{
    private $a_titles = array();

    public function getIterator()
    {
        return new BookIterator($this);
    }

    public function addTitle($string)
    {
        $this->a_titles[] = $string;
    }

    public function getTitle($key)
    {
        if (isset($this->a_titles[$key])) {
            return $this->a_titles[$key];
        }
        return null;
    }

    public function is_empty()
    {
        return empty($this->$a_titles);
    }
}
```

```php
<?php

// index.php

require 'vendor/autoload.php';
use DesignPatterns\BookCollection;

$booksCollection = new BookCollection();
$booksCollection->addTitle('Design Patterns');
$booksCollection->addTitle('PHP7 is the best');
$booksCollection->addTitle('Laravel Rules');
$booksCollection->addTitle('DHH Rules');

foreach($booksCollection as $book){
    var_dump($book);
}
```

OUTPUT

```
string(15) "Design Patterns"
string(16) "PHP7 is the best"
string(13) "Laravel Rules"
string(9) "DHH Rules"
```

Python

Python prescribes a syntax for iterators as part of the language itself, so that language keywords such as `for` work with what Python calls sequences. A sequence has an `__iter__()` method that returns an iterator object. The "iterator protocol" requires `next()` return the next element or raise a `StopIteration` exception upon reaching the end of the sequence. Iterators also provide an `__iter__()` method returning themselves so that they can also be iterated over e.g., using a `for` loop. Generators are available since 2.2.

In Python 3, `next()` was renamed `__next__()`.[8]

See also

- Composite pattern
- Container (data structure)
- Design pattern (computer science)
- Iterator
- Observer pattern

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 257ff. ISBN 0-201-63361-2.
2. "The Iterator design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=b04&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
3. Gang Of Four
4. "The Iterator design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=b04&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.
5. https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/foreach-in
6. "Iterators and generators" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators). Retrieved 18 March 2016.
7. "PHP: Iterator" (http://www.php.net/manual/en/class.iterator.php). Retrieved 23 June 2013.
8. "Python v2.7.1 documentation: The Python Standard Library: 5. Built-in Types" (https://docs.python.org/library/stdtypes.html). Retrieved 2 May 2011.

External links

- Object iteration (http://us3.php.net/manual/en/language.oop5.iterations.php) in PHP
- Iterator Pattern (http://www.dofactory.com/Patterns/PatternIterator.aspx) in C#
- Iterator pattern in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Iterator.xml)
- SourceMaking tutorial (http://sourcemaking.com/design_patterns/iterator)
- Design Patterns implementation examples tutorial (http://patterns.pl/iterator.html)
- Iterator Pattern (http://c2.com/cgi/wiki?IteratorPattern)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Iterator_pattern&oldid=879803199"

This page was last edited on 23 January 2019, at 14:00 (UTC).