Iterating collections

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFSEN02-2

Introduction

Introduction

Lecture topics

- A taxonomy of design patterns
- Iterating collections
- Concrete examples of the iterator design pattern
- Conclusions

A taxonomy of design patterns

A taxonomy of design patterns

A taxonomy of design patterns

- After having seen the first design pattern, we can add some depth to the discussion
- Design patterns have been grouped in several specific categories (we will show at least one design pattern per category):

- Behavioral
- Structural
- Creational

A taxonomy of design patterns

Behavioral patterns

- Design patterns for identifying the fundamental communication behavior between entities
- Among such patterns we find:

- **Visitor pattern**
- State pattern
- Strategy pattern
- **Null Object pattern**
- Iterator pattern
- etc..

A taxonomy of design patterns

Structural patterns

- Design patterns that ease the design of an application by identifying a simple way to implement relationships between entities
- Among such patterns we find:

- Adapter pattern
- Decorator pattern
- Proxy pattern
- etc..

A taxonomy of design patterns

Creational patterns

- Design patterns that deal with entities creation mechanisms, trying to create entities in a manner suitable to the situation
- They make it possible to have "polymorphic" constructors
- Among such patterns we find:

- Factory method pattern
- Lazy initialization pattern
- Singleton pattern
- etc..

A taxonomy of design patterns

Software development principles

- Even more abstractly, design patterns are all rooted in the same principles
- These principles make it possible to derive old and new patterns
- THey are refinements of the broader principles of encapsulation and loose coupling

A taxonomy of design patterns

Software development principles

- Such principles are:

DRY : Is an acronym for the design principle "Don't Repeat Yourself"

KISS : Is an acronym for the design principle "Keep it simple, Stupid!"

SOLID : Is an acronym for Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion

A taxonomy of design patterns

DRY

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
- Violations of DRY are typically referred to as *WET*(write everything twice) solutions

A taxonomy of design patterns

KISS

- It states that most systems work best if they are kept simple rather than made complicated
- Simplicity should be a key goal in design and unnecessary complexity should be avoided
- See λ-calculus / stack heap: complex system from single rules

A taxonomy of design patterns

SOLID

- S : a class should have only a single responsibility
- O : entities should be "open" for extensions, but "closed" for modification
- L : objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program
- I : many "specific" interfaces are better than one general-purpose interface
- D : high-level modules should not dependent from the low-levels; both should depend on abstractions. Abstractions should not depend on details and vice-versa

A taxonomy of design patterns

Software development principles

- In this course we will always try, when introducing a design pattern, to present it along with its principles

Iterating collections

Iterating collections

Introduction

- Today we are going to study collections
- In particular, we are going to study how to access the elements of a collection without exposing its underlying representation (methods and fields)
- How? By means of a design pattern: the iterator (a behavioral design pattern)

Iterating collections

Different implementations for different collections

- Stream of data
- Records of a database
- List of cars
- Array of numbers
- Array of Array of pixels (a matrix)
- Option (zero or one elements
- etc..

What do we do with collections?

- However, all collections, from options to arrays, exhibit similarities
- The *general* idea is going through all the elements one by one until there are no more to see

Iterating collections

What do we do with collections?

- Unfortunately, every collection has its own different implementation
- This is an issue
- Why?

Iterating collections

What do we do with collections?

- Unfortunately, every collection has its own different implementation
- This is an issue
- Why?
- **Because we would have to write specific access/iteration code for each collection**

What do we do with collections?

- Unfortunately, every collection has its own different implementation
- This is an issue
- Why?
- **Because we would have to write specific access/iteration code for each collection**
- **For example: how hard would it have been to change all custom lists into Python standard lists in DEV2?**

Similar collections, but with different implementation

- Take for example a linked list and an array:
- The former is a dynamic data structure made of linked nodes
- The latter is a static compact data structure with a fixed number of elements

Iterating collections

Iterating lists

- Iterating a list requires a variable that references the current node in the list
- To move to the next node we need to manually update such variable, by assigning to it a reference to the next node

```
LinkedList<int> list_of_numbers = new LinkedList<int>();
...
while (list_of_numbers.Tail != null) {
  ...
  list_of_numbers = list_of_numbers.Tail;
 }
...
```

Iterating collections

Which in Java then becomes:

```
LinkedList<Integer> list_of_numbers = new LinkedList<Integer>();
...
while (list_of_numbers.Tail != null) {
...
  list_of_numbers = list_of_numbers.Tail;
 }
...
```

Iterating array

- Iterating an array requires a variable (an index) containing a number representing the position of the current visited element
- To move to the next element we need to manually update the index, increasing it by one

```
int[] array_of_numbers = new int[5];
...
int index;
for(index = 0;(index <= array_of_numbers.Length);index = (index + 1)){
  ...
 }
...
```

Iterating collections

Iterating array

- Iterating an array requires a variable (an index) containing a number representing the position of the current visited element
- To move to the next element we need to manually update the index, increasing it by one

```
int[] array_of_numbers = new int[5];
...
int index;
for(index = 0;(index <= array_of_numbers.Length);index = (index + 1)){
  ...
 }
...
```

- What about all other collections?
- Maps, sets, trees, etc..

Which in Java then becomes:

```
int[] array_of_numbers = new int[5];
...
int index;
for(index = 0;(index <= array_of_numbers.Length);index = (index + 1)){
  ...
 }
...
```

Iterating collections

The need for different collections

- A collection has its own purpose: for example arrays are very performant in retrieving data at specific positions, linked lists allow fast insertions, etc..
- But then how can we hide the implementation details so that iterating collections becomes trivial if the specifics are not relevant?

Iterating collections

Issues

- Repeating code is problematic (DRY: do not repeat iteration logic)
- Knowing too much about a data structure increases coupling, making code more complex (KISS: keep iteration superficially simple)

Our goal

- We try to achieve a mechanism that abstracts our concrete collections from their iteration algorithms
- Iteration is a behavior common to all collections: only its implementation changes

How do we achieve it?

- We wish to delegate the implementation of such algorithms to each concrete collection
- We control such algorithms by means of a common/shared interface

Iterating collections

What follows?

- When developers need to iterate a collection they simply use the interface provided by the chosen collection
- Such interface hides the internals of a collection and provides a clean interaction surface for iterating it

The iterator design pattern

- Is a design pattern that captures the iteration mechanism
- We will now study it in detail and provide a series of examples

The iterator design pattern

The iterator design pattern

The iterator design pattern

- We will define an interface capturing the basics to all container iterations:

- get current item
- move to next item
- check if next item exists

The iterator design pattern

The iterator design pattern

- Is an interface `Iterator<T>` containing the following method signature

```
interface Iterator<T> {
  IOption<T> GetNext();
}
```

- `GetNext` returns `Some<T>` if there is an item to fetch
- It moves to the next item
- It returns `None<T>` if there are no more elements to fetch

The iterator design pattern

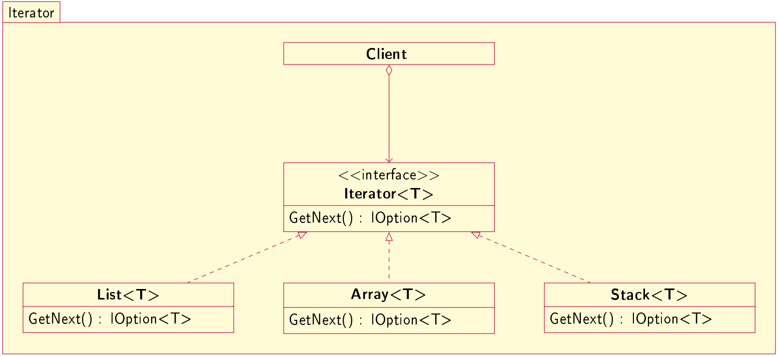
Which in Java then becomes:

```
interface Iterator<T> {
  IOption<T> GetNext();
}
```

Implementing the Iterator<T>

- At this point every collection that wants to provide a disciplined and controlled iteration mechanism has to either implements such interface or provide a way to adapt to it
- Iterating a collection with 5, 3, 2 will return: Some(5), Some(3), Some(2), None(), None(), None(), ..., None()

The iterator design pattern



The iterator design pattern

Implementing the Iterator<T>

- We now show a series of collections implementing such an interface

Natural numbers

- The natural numbers are all integers greater than or equal to 0
- `GetNext` increases a counter `n` (starting from -1) and returns it within a `Some`

```
class NaturalList : Iterator<int> {
  private int current = -1;
  IOption<int> GetNext() {
    current = (current + 1);
    return new Some<int>(current);
  }
}
```

The iterator design pattern

IterableList<T>

- Dealing with a list requires to deal with references
- We hide such complexity, which is error-prone, by means of our iterator
- We use the unsafe version of the list with `IsNone`, `GetValue`, and `GetTail` for simplicity; a visit would be better

IterableList<T>

- Our iterable list now takes as input an object of type list
- `GetNext` returns `None` at the end of the list (when the tail is `None`), otherwise it moves to the next node and returns its value wrapped inside a `Some`

```
class IterableList<T> : Iterator<T> {
  private List<T> list;
  public IterableList(List<T> list) {
    this.list = list;
  }
  IOption<T> GetNext() {
    if list.IsNone() {
      return new None<T>();
    }
    else{
      List<T> tmp = list;
      list = list.GetTail();
      return new Some<T>(tmp.GetValue());
    }
  }
}
```

The iterator design pattern

Which in Java then becomes:

```
class IterableList<T> implements Iterator<T> {
  private List<T> list;
  public IterableList(List<T> list) {
    this.list = list;
  }
  IOption<T> GetNext() {
    if list.IsNone() {
      return new None<T>();
    }
    else{
      List<T> tmp = list;
      list = list.GetTail();
      return new Some<T>(tmp.GetValue());
    }
  }
}
```

The iterator design pattern

IterableArray<T>

- Dealing with an array requires to deal with its indexes
- We hide such complexity, which is error-prone, by means of our iterator

The iterator design pattern

IterableArray<T>

- Our iterable array takes as input an object of type array
- GetNext returns None at the end of the array, otherwise it increases the index and returns the value of the array at position index wrapped inside a Some

```
class IterableArray<T> : Iterator<T> {
  private T[] array;
  private int index = -1;
  public IterableArray(T[] array) {
    this.array = array;
  }
  IOption<T> GetNext() {
    if ((index + 1) >= array.Length) {
      return new None<T>();
    }
    else{
      index = (index + 1);
      return new Some<T>(array[index]);
    }
  }
}
```

The iterator design pattern

Other collections

- Each container will then store its own reference to a collection, plus the current iterated element
- The plumbing is trivial per container
- We obviously cannot show them all

The iterator in literature

- In the literature we often find another formulation

```
interface TraditionalIterator<T> {
  void MoveNext();
  bool HasNext();
  T GetCurrent();
}
```

- As we can see, this is less safe, since now we have to *carefully* manipulate three methods (instead of one as for `Iterator<T>`)

The iterator design pattern

Which in Java then becomes:

```
interface TraditionalIterator<T> {
  void MoveNext();
  bool HasNext();
  T GetCurrent();
}
```

The iterator design pattern

Improving the TraditionalIterator<T> safeness

- Adapting our `TraditionalIterator` will require us to define an adapter `MakeSafe` that implements our `Iterator` by coordinating method calls to an underlying `Iterator`[a]

[a] Adapter allows to automatically convert back/forth between iterators

```
class MakeSafe<T> : Iterator<T> {
  private TraditionalIterator<T> iterator;
  public MakeSafe(TraditionalIterator<T> iterator) {
    this.iterator = iterator;
  }
  IOption<T> GetNext() {
    if iterator.HasNext() {
      iterator.MoveNext();
      return new Some<T>(iterator.GetCurrent());
    }
    else{
      return new None<T>();
    }
  }
}
```

The iterator design pattern

Which in Java then becomes:

```
class MakeSafe<T> implements Iterator<T> {
  private TraditionalIterator<T> iterator;
  public MakeSafe(TraditionalIterator<T> iterator) {
    this.iterator = iterator;
  }
  IOption<T> GetNext() {
    if iterator.HasNext() {
      iterator.MoveNext();
      return new Some<T>(iterator.GetCurrent());
    }
    else{
      return new None<T>();
    }
  }
}
```

Conclusions

Conclusions

Conclusions

- Iterating collections is a time consuming, error-prone activity, since collections come with different implementations each with its own complexity
- Iterators are a mechanism that hides the complexity of a collection and provides a clean interaction surface to iterate them
- This mechanism not only reduces the amount of code to write (achieving then the DRY principle), but also reduces the amount of coupling

This is it!

The best of luck, and thanks for the attention!