# Iterating collections

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

# Dev 4 - Lecture topics

# Dev 4 - Lecture topics

## Lecture topics

- Breaking down design patterns, an introduction
- Iterating collections
- Concrete examples of the iterator design pattern
- Conclusions

# Breaking down design patterns

## Introduction

- After having seen the 1st design pattern, we can add some depth to the discussion
- Design patterns have been grouped in several specific categories:
- Behavioral
- Structural
- Creational
- In this course we will always try, when introducing a design pattern, to picture it with respect to such categories

# Breaking down design patterns

## Behavioral patterns

- Are design patterns for identifying the fundamental communication behavior between entities
- Among such pattern we find:

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

# Breaking down design patterns

## Structural patterns

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

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

# Breaking down design patterns



## Creational patterns

- Are 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 "virtual" constructors
- Among such pattern we find:

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

# Breaking down 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

# Breaking down 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

# Breaking down 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)
- We will see how the iterator provides a clean, almost trivial, general way representation for iterating collections

# 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 (an option essentially is a "one-element' collections")
- etc..

# Iterating collections

## What do we do with collections?

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

## What do we do with collections?

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

## 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 code for each collection

### 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. A linked list potentially might contains infinite nodes
- The latter is a static compact data structure. In an array the maximum number of elements is fixed

# 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 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 variable, increasing it by one.

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

### The need for different collections

- A collection has its own use: 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 make code more complex (KISS: keep iteration simple)

# Iterating collections



## 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

# Iterating collections



## 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

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

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

# The iterator design pattern

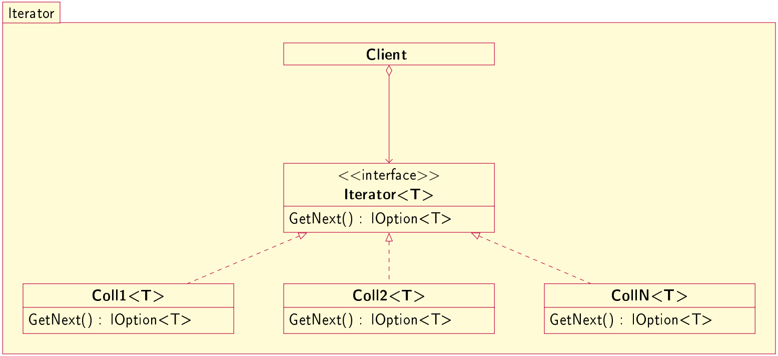


## Implementing the Iterator<T>

- At this point every collection that wants to provide a disciplined and controlled iteration mechanism has to implement such interface

# The iterator design pattern





# The iterator design pattern



## Implementing the Iterator<T>

- We now show a series of collections implementing such interface

## Natural numbers

- The natural numbers are all integers greater than or equal to 0
- We now wish to define a collection containing all natural numbers
- To do so we define a data structure that implements our iterator
- And starting from -1 (the successor of it is 0, the first natural number), which is stored in a field called `current`, whenever we call the `GetNext` method we increase such `current` and returns its value

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

# The iterator design pattern

## Array<T>

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

# The iterator design pattern



## Array<T>

- Our “new” array takes as input an object of type array
- A field `index` keeps track of the current index
- Whenever the `GetNext` method is called we check whether we reached the end of the array: if so we return `None`, otherwise we increase the index and return the value of the array at position `index` wrapped inside a `Some` object

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

# The iterator design pattern



## The iterator in literature

- In literature it is often the case to see our Iterator as an interface containing the following signatures

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

- The main big difference now is that whenever we need to move throughout our collection we have to coordinate `GetCurrent`, `HasNext`, and `MoveNext`
- As we can see, this adds a layer of complexity to the iteration and is error prone, since now we have to *carefully* manipulate three methods (instead of one as for `Iterator<T>`)
- In what follows we show how to make `UnsafeIterator`

## Improving the UnsafeIterator<T> safeness

- Adapting our `UnsafeIterator` will require us to define an adapter `AdapterIterator` that implements our `Iterator`
- The `AdapterIterator` takes as input an `UnsafeIterator` and whenever the `GetNext` method is called it calls `GetCurrent` and `MoveNext` accordingly

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

## Iterating an IOption<T>

- We could also make our `IOption<T>` iterable
- To do so, we have to return `Some` only the first time we it and `None` for all successive iterations
- Note, if we iterate a `None` entity we return `None`

```
class IOptionAdapter<T> : Iterator<T> {
  private IOption<T> option;
  private bool visited = false;
  public IOptionAdapter(IOption<T> option) {
    this.option = option;
  }
  IOption<int> GetNext() {
    if visited {
      return new None<T>();
    }
    else{
      visited = true;
      return option.Visit<IOption<T>>(() => new None<T>(),t => new Some<T>(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!