

# The decorator design pattern

The INFDEV team

Hogeschool Rotterdam  
Rotterdam, Netherlands

# INFDEV02-4

# Introduction

## Introduction

- Today we are going to study a behavioral pattern: the decorator design pattern
- Sometimes, we need to modify behaviors of an instance dynamically

## Introduction

- Hand made combinations could be a solution, but excessive inheritance is a pitfall, since the amount of combinations could be huge:

- Car → with turbo → new class
- Car → with truck → new class
- Car → with extra seat → new class
- Car → with turbo + extra seat → new class
- etc.

## Introduction

- The decorator pattern (also known as wrapper) solves this issue
- How? By emulating polymorphism through composition

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

# Case study

## Case study

- Consider again the iterator interface

```
1 interface Iterator<T> {  
2     IOption<T> GetNext();  
3 }
```



## Case study

- Consider again the natural numbers implementation

```
1 class Naturals : Iterator<int> {  
2     private int current;  
3     public Naturals() {  
4         current = 1;  
5     }  
6     IOption<int> GetNext() {  
7         current = (current + 1);  
8         return new Some<int>(current);  
9     }  
10 }
```

## First task: selecting only natural even numbers

- We wish now to iterate only the even numbers of our natural number list

```
1 class Evens : Iterator<int> {  
2     private int current;  
3     public Evens() {  
4         current = -1;  
5     }  
6     IOption<int> GetNext() {  
7         current = (current + 1);  
8         if(((current % 2) == 0)) {  
9             return new Some<int>(current);  
10        }  
11        else{  
12            return this.GetNext();  
13        }  
14    }  
15 }
```

## Another task: iteration with offset

- We wish now to iterate our natural number list and while iterating it add an offset to each element

```
1 class Offset : Iterator<int> {  
2     private int current;  
3     private int offset;  
4     public Offset(int offset) {  
5         current = -1;  
6         this.offset = offset;  
7     }  
8     IOption<int> GetNext() {  
9         current = (current + 1);  
10        return new Some<int>((current + offset));  
11    }  
12 }
```

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## UML

- Lets give a look to the UML of our classes made so far..

# Case study

The  
decorator  
design  
pattern

The INFDEV  
team

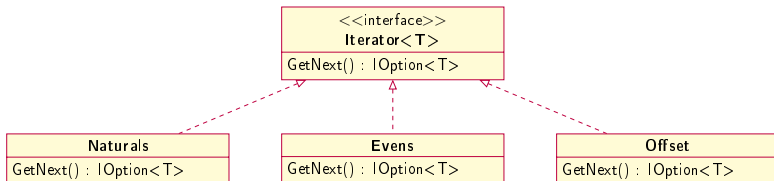
INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions



## A new task: iteration over evens with offset

- We wish now to iterate only even numbers of our natural number list and for each number add an offset
- Yes, we need another class...

```
1 class EvensFrom : Iterator<int> {  
2     private int current;  
3     private int offset;  
4     public EvensFrom(Offset offset) {  
5         current = -1;  
6         this.offset = offset;  
7     }  
8     IOption<int> GetNext() {  
9         current = (current + 1);  
10        if(((current % 2) == 0)) {  
11            return new Some<int>((current + offset));  
12        }  
13        else{  
14            return this.GetNext();  
15        }  
16    }  
17 }
```

## UML discussion

- As we can see our class hierarchy is growing “horizontally”, because of lacks of reuse
- Let us see the UML again

# Case study

The  
decorator  
design  
pattern

The INFDEV  
team

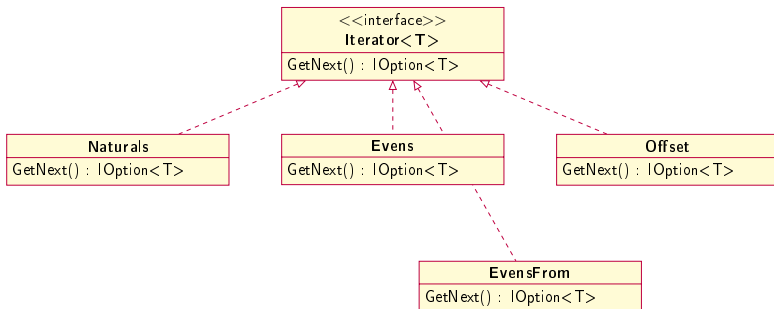
INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions





## Iterating a range between two integers

- Imagine if we now wish to implement a new data structure `Range` that takes two integers `A` and `B` (where  $A \leq B$ )
- We want `Range` to support the `Offset` and `Even` behaviors
- We have to literally duplicate everything and to implement all possible combinations

## Considerations

- Polymorphism solves our problem, but adds another one.  
Too many combinations
- Every change/add requires lots of work
- Behavioral commonalities are not taken into consideration

## Considerations

- How can we group such behaviors (offset and even) to define them once and use them everywhere?
- A possible solution would see our natural number implementing offset and even

```
1 class EvensFrom : Naturals , Offset , Evens {  
2     ...  
3 }
```

- This solution is not good, since the responsibilities are now not clear, see SOLID

## Considerations

- Abstract classes with a series of booleans, which we can use as “switchers” to select the appropriate algorithm, could be another solution
- But fields do not force appropriate behavior for each of the roles

```
1 class Naturals : Iterator<int> {  
2     private bool isEven;  
3     private bool isOffset;  
4     ...  
5 }
```

- This solution is not good, since the responsibilities are now not clear, see SOLID

## Considerations

- We need a better mechanism. Ideally we wish:
  - To define once our naturals
  - To define once our even behavior
  - To define once our offset behavior
  - To apply the above behaviors “on demand”, and not to all instances of natural lists
  - To combine the above behaviors without defining new behaviors

## Idea

- A interesting solution could be to built a *proxy*, like adapter, but with the possibility to add semantics!

## Idea

- We define an intermediate entity `Decorator`, which inherits our iterator and also contains an instance of it (`decorated_item`)
- Note `GetNext` acts as a proxy by simply calling `decorated_item.GetNext()` and returning its result
- `Decorator` is abstract, so you cannot create it without a “concrete” behavior

```
1 abstract class Decorator : Iterator<int> {  
2     protected Iterator<int> decorated_item;  
3     public Decorator(Iterator<int> decorated_item) {  
4         this.decorated_item = decorated_item;  
5     }  
6     protected abstract IOption<int> GetNext();  
7 }
```

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Idea

- We can think of the decorator as an iterator containing elements, but which does not know how to iterate them
- A concrete decorator needs a specific specification of how to iterate



## Idea

- We now declare concrete two decorators `Even` and `Offset` that extend our `Decorator`
- `Even` and `Offset` are unaware (and they do not need to be) of whether they are going to deal with all natural numbers, just a range of them, or something else

## Idea: even class

- In the following you find the code for Even

```
1 class Even : Decorator {  
2     public Even(Iterator<int> collection) : base(collection) {  
3     }  
4     public override IOption<int> GetNext() {  
5         Option<int> current = base.decorated_item.GetNext();  
6         if(current.IsNone()) {  
7             return new None<int>();  
8         }  
9         else{  
10            if (((current.GetValue() % 2) == 0)) {  
11                return new Some<int>(current.GetValue());  
12            }  
13            else{  
14                return this.GetNext();  
15            }  
16        }  
17    }  
18 }
```

## Idea: even class, with lambdas

- In the following you find the code for Even, with lambdas

```
1 class Even : Decorator {  
2     public Even(Iterator<int> collection) : base(collection) {  
3     }  
4     public override IOption<int> GetNext() {  
5         Option<int> current = base.decorated_item.GetNext();  
6         current.Visit(() => new None<int>(), current =>  
7 { if(((current % 2) == 0)) {  
8     return new Some<int>(current);  
9 }  
10    else{  
11        return this.GetNext();  
12    }  
13    });  
14    }  
15 }
```

## Idea: offset class

- In the following you find the code for Offset

```
1 class Offset : Decorator {
2     private int offset;
3     public Offset(Offset offset) {
4         this.offset = offset;
5     }
6     public override IOption<int> GetNext() {
7         Option<int> current = base.decorated_item.GetNext();
8         if (current.IsNone()) {
9             return new None<int>();
10        }
11        else{
12            return new Some<int>((current.GetValue() + offset));
13        }
14    }
15 }
```

## Idea: offset class, with lambdas

- In the following you find the code for Offset, with lambdas

```
1 class Offset : Decorator {  
2     private int offset;  
3     public Offset(Offset offset) {  
4         this.offset = offset;  
5     }  
6     public override IOption<int> GetNext() {  
7         Option<int> current = base.decorated_item.GetNext();  
8         current.Visit(() => new None<int>(), current => new Some<int>((current +  
9             offset)));  
10    }
```

## Idea

- With Even and Offset we managed to capture a reusable behavior that works with any collection made of numbers
  - They work on Range and Numbers
  - They work on any other collection of ints, even with those built with decorators
- Naturals → Evens → Offset
  - etc.

## Idea

- The following are all examples of how to use our new data structures:
- `Iterator<int> ns1 = new Even(new Naturals())`
- `Iterator<int> ns2 = new Offset(new Naturals())`
- `Iterator<int> ns2 = new Offset(new Even(new Naturals()))`
- `Iterator<int> ns3 = new Offset(new Even(new Range(5, 10)))`

## Idea

- Note how decomposability helps to keep the interaction surface clean and reusable and the implementation compact
- We now show the UML of our code



# Case study

The  
decorator  
design  
pattern

The INFDEV  
team

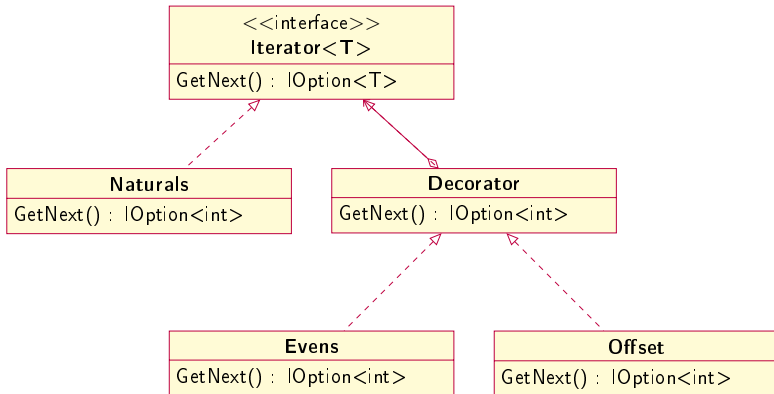
INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions



## Considerations

- The pattern seen so far follows a specific design pattern that is called the **Decorator design pattern** (a behavioral design pattern)
- We now study its formalization and add some final considerations

# The decorator design pattern

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Formalism

- Given a polymorphic type  $I$  (to instantiate)
- Given a concrete implementations of  $I$ :  $C$
- A decorator  $D$  is an entity that implements  $I$  and references an instance of  $I$
- A concrete decorator  $CD$  extends  $D$
- As concrete  $CD$ 's come with difference semantics, every  $CD$  is tasked to apply its semantics by overriding methods of the inherited  $D$

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

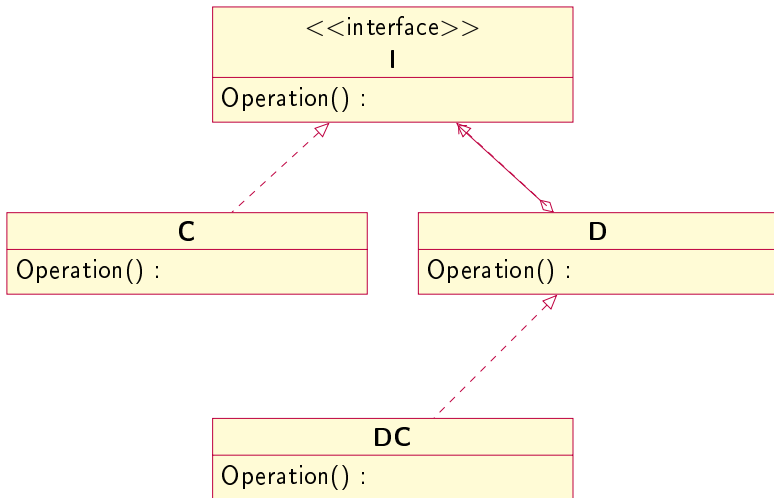
INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions



# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Generic decorators

- We can build generic decorators
- Genericity comes from lambdas, which act as “holes” in their behaviors
- Concrete decorators can be defined by specifying the underlying iterator + the lambdas

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Filter

- A Filter is a generic decorator that skips some elements
- We can use it to express our Evens, see code below

```
1 class Filter : Decorator {
2     Func<int, bool> p;
3     public Filter(Iterator<int> collection, Func<int, bool> p) : base(
4         collection) {
5         p = this.p;
6     }
7     public override IOption<int> GetNext() {
8         Option<int> current = base.decorated_item.GetNext();
9         if (current.IsNone()) {
10             return new None<int>();
11         }
12         else {
13             if (p.Invoke(current.GetValue())) {
14                 return new Some<int>(current.GetValue());
15             }
16             else {
17                 return this.GetNext();
18             }
19         }
20     }
21 }
```

```
Iterator<int> numbers = new Filter(new Range(0,5), n => ((n % 2) == 0));
```

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Filter, with lambdas

- A Filter is a generic decorator that skips some elements
- We can use it to express our Evens, see code below (with lambdas)

```
1 class Filter : Decorator {  
2     Func<int, bool> p;  
3     public Filter(Iterator<int> collection, Func<int, bool> p) : base(  
4         collection) {  
5         p = this.p;  
6     }  
7     public override IOption<int> GetNext() {  
8         Option<int> current = base.decorated_item.GetNext();  
9         current.Visit(() => new None<int>(), current =>  
10 { if (p.Invoke(current)) {  
11     return new Some<int>(current);  
12 }  
13 else {  
14     return this.GetNext();  
15 }  
16 }  
17 }  
18 Iterator<int> numbers = new Filter(new Range(0,5), n => ((n % 2) == 0));
```



# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Map

- A Map transforms all elements one after the other
- We can use it to express our Offset, see code below

```
1 class Map : Decorator {
2     Func<int, int> t;
3     private int offset;
4     public Map(Offset offset) {
5         this.offset = offset;
6     }
7     public override IOption<int> GetNext() {
8         Option<int> current = base.decorated_item.GetNext();
9         if(current.IsNone()) {
10             return new None<int>();
11         }
12         else{
13             return new Some<int>(t.Invoke(current.GetValue()));
14         }
15     }
16 }
17 Iterator<int> numbers = new Map(new Range(0,5), n => (current + 1));
```

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Map, with lambdas

- A Map transforms all elements one after the other
- We can use it to express our Offset, see code below (with lambdas)

```
1 class Map : Decorator {  
2     Func<int, int> t;  
3     private int offset;  
4     public Map(Offset offset) {  
5         this.offset = offset;  
6     }  
7     public override IOption<int> GetNext() {  
8         Option<int> current = base.decorated_item.GetNext();  
9         current.Visit(() => new None<int>(), current => new Some<int>(t.Invoke(  
10             current)));  
11     }  
12     Iterator<int> numbers = new Map(new Range(0,5), n => (current + 1));
```

# The decorator design pattern

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

## Map and Filter

- Of course we can combine them, so to express our EvensFrom
- `Iterator<int> numbers = new Filter(new Map(new Range(0,5), n => n + 1), n => n % 2 == 0);`

# Conclusions

## Conclusions

- Sometimes, we need to apply behaviors instances dynamically
- Hand made combinations could be a solution, but the number of combinations is huge
- The decorator pattern solves this issue by emulating a customizable, dynamic form of polymorphism through composition
- In short, a decorator allows behavior to be added to an individual object, without affecting other objects

The  
decorator  
design  
pattern

The INFDEV  
team

INFDEV02-4

Introduction

Case study

The  
decorator  
design  
pattern

Conclusions

The best of luck, and thanks for the  
attention!