The decorator design pattern

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Introduction

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 $\rightarrow$ with turbo $\rightarrow$ new class
  - Car $\rightarrow$ with truck $\rightarrow$ new class
  - Car $\rightarrow$ with extra seat $\rightarrow$ new class
  - Car $\rightarrow$ with turbo + extra seat $\rightarrow$ new class
  - etc.

Introduction

Introduction

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

Case study

Case study

Case study

- Consider again the iterator interface

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

Case study

- Consider again the natural numbers implementation

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

Case study

First task: selecting only natural even numbers

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

```
class Evens : Iterator<int> {
  private  int current;
  public Evens() {
    current = -1;
  }
  IOption<int> GetNext() {
    current = (current + 1);
    if(((current % 2) == 0)) {
      return new Some<int>(current);
    }
    else{
      return this.GetNext();
    }
  }
}
```

Another task: iteration with offset

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

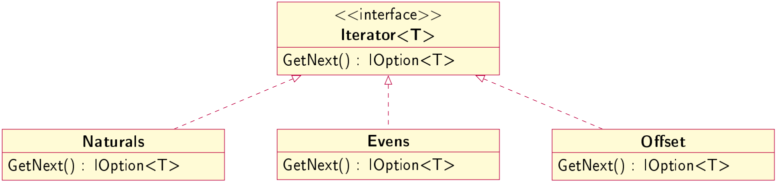
```
class Offset : Iterator<int> {
  private int current;
  private int offset;
  public Offset(int offset) {
    current = -1;
    this.offset = offset;
  }
  IOption<int> GetNext() {
    current = (current + 1);
    return new Some<int>((current + offset));
  }
}
```

Case study

UML

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

Case study


<<interface>>
Iterator<T>
GetNext() : IOption<T>
Naturals
GetNext() : IOption<T>
Evens
GetNext() : IOption<T>
Offset
GetNext() : IOption<T>

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...

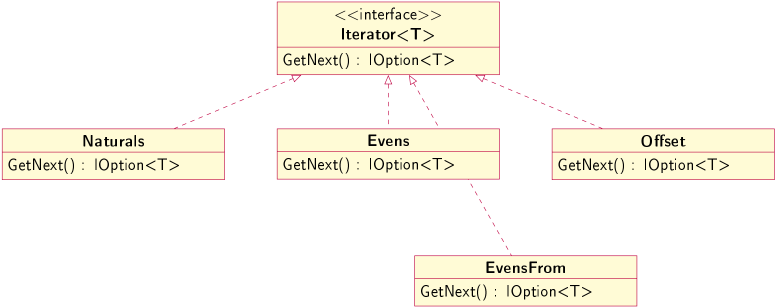
```
class EvensFrom : Iterator<int> {
  private int current;
  private int offset;
  public EvensFrom(Offset offset) {
    current = -1;
    this.offset = offset;
  }
  IOption<int> GetNext() {
    current = (current + 1);
    if(((current % 2) == 0)) {
      return new Some<int>((current + offset));
    }
    else{
      return this.GetNext();
    }
  }
}
```

Case study

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


<<interface>>
Iterator<T>
GetNext() : IOption<T>
Naturals
GetNext() : IOption<T>
Evens
GetNext() : IOption<T>
Offset
GetNext() : IOption<T>
EvensFrom
GetNext() : IOption<T>

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 <= B)
- We want `Range` to support the `Offset` and `Even` behaviors
- We have to literally duplicate everything and to implement all possible combinations

Case study

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

Case study

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

```
class EvensFrom : Naturals, Offset, Evens {
  ...
}
```

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

Case study

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

```
class Naturals : Iterator<int> {
  private bool isEven;
  private bool isOffset;
  ...
}
```

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

Case study

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

Case study

Idea

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

Case study

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

```
abstract class Decorator : Iterator<int> {
  protected Iterator<int> decorated_item;
  public Decorator(Iterator<int> decorated_item) {
    this.decorated_item = decorated_item;
  }
  protected abstract  IOption<int> GetNext();
}
```

Case study

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

Case study

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

Case study

Idea: even class

- In the following you find the code for Even

```
class Even : Decorator {
  public Even(Iterator<int> collection) : base(collection) {
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    if(current.IsNone()) {
      return new None<int>();
    }
    else{
      if(((current.GetValue() % 2) == 0)) {
        return new Some<int>(current.GetValue());
      }
      else{
        return this.GetNext();
      }
    }
  }
}
```

Idea: even class, with lambdas

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

```
class Even : Decorator {
  public Even(Iterator<int> collection) : base(collection) {
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    current.Visit(() => new None<int>(),current =>
{ if(((current % 2) == 0)) {
  return new Some<int>(current);
}
else{
  return this.GetNext();
}
 });
  }
}
```

Case study

Idea: offset class

- In the following you find the code for `Offset`

```
class Offset : Decorator {
  private int offset;
  public Offset(Offset offset) {
    this.offset = offset;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    if(current.IsNone()) {
      return new None<int>();
    }
    else{
      return new Some<int>((current.GetValue() + offset));
    }
  }
}
```

Case study

Idea: offset class, with lambdas

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

```
class Offset : Decorator {
  private  int offset;
  public Offset(Offset offset) {
    this.offset = offset;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    current.Visit(() => new None<int>(),current => new Some<int>((current +
         offset)));
  }
}
```

Case study

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` $\rightarrow$ `Evens` $\rightarrow$ `Offset`
- etc.

Case study

Idea

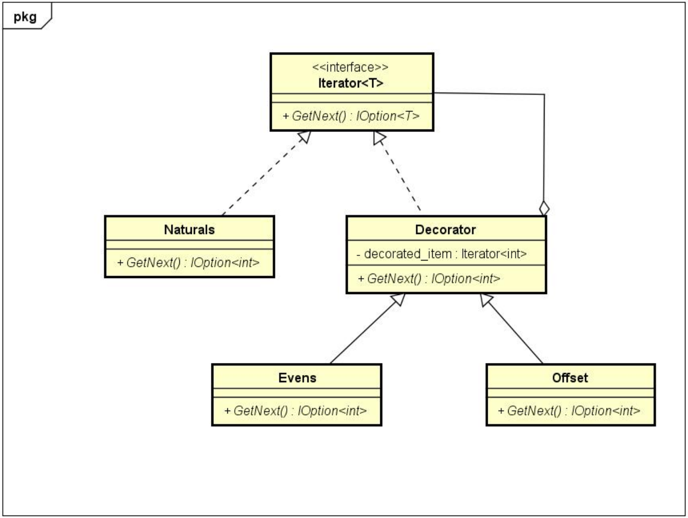
- The following are all examples of how to use our new data structures:
- `Iterator<int> ns1 = new Even(new Naturals())`
- `Iterator<int> ns2 = new Offset(new Naturals())`
- `Iterator<int> ns2 = new Offset(new Even(new Naturals()))`
- `Iterator<int> ns3 = new Offset(new Even(new Range(5, 10)))`

Case study

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

Idea


pkg
<<interface>>
Iterator<T>
+ GetNext() : IOption<T>
Naturals
+ GetNext() : IOption<int>
Decorator
- decorated_item : Iterator<int>
+ GetNext() : IOption<int>
Evens
+ GetNext() : IOption<int>
Offset
+ GetNext() : IOption<int>

Case study

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

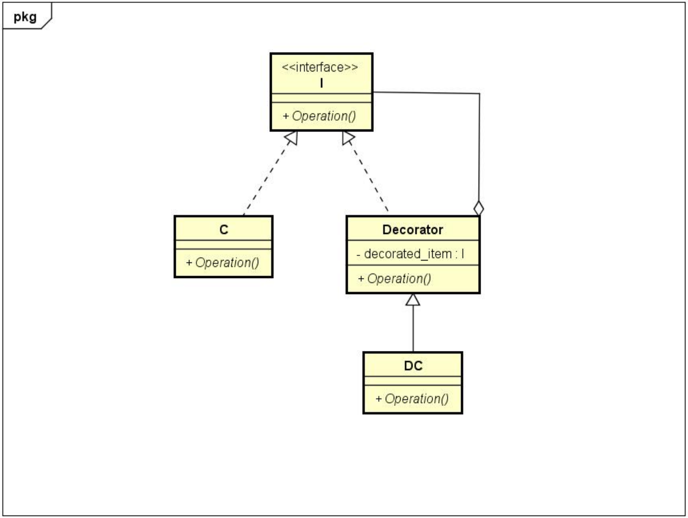
The decorator design pattern

The decorator design pattern

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

Formalism


pkg
<<interface>>
I
+ Operation()
C
+ Operation()
Decorator
- decorated_item : I
+ Operation()
DC
+ Operation()

The decorator design pattern

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

Filter

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

```
class Filter : Decorator {
  Func<int, bool> p;
  public Filter(Iterator<int> collection,Func<int, bool> p) : base(
       collection) {
    p = this.p;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    if(current.IsNone()) {
      return new None<int>();
    }
    else{
      if(p.Invoke(current.GetValue())) {
        return new Some<int>(current.GetValue());
      }
      else{
        return this.GetNext();
      }
    }
  }
}
Iterator<int> numbers = new Filter(new Range(0,5),n => ((n % 2) == 0));
```

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)

```
class Filter : Decorator {
  Func<int, bool> p;
  public Filter(Iterator<int> collection,Func<int, bool> p) : base(
       collection) {
    p = this.p;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    current.Visit(() => new None<int>(),current =>
{ if(p.Invoke(current)) {
  return new Some<int>(current);
}
else{
  return this.GetNext();
}
 });
  }
}
Iterator<int> numbers = new Filter(new Range(0,5),n => ((n % 2) == 0));
```

The decorator design pattern

Map

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

```
class Map : Decorator {
  Func<int, int> t;
  private int offset;
  public Map(Offset offset) {
    this.offset = offset;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.decorated_item.GetNext();
    if(current.IsNone()) {
      return new None<int>();
    }
    else{
      return new Some<int>(t.Invoke(current.GetValue()));
    }
  }
}
Iterator<int> numbers = new Map(new Range(0,5),n => (current + 1));
```

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)

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

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

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

This is it!

The best of luck, and thanks for the attention!