The adapter design pattern

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Introduction

Introduction

- Today we are going to study the a behavioral pattern: the decorator design pattern
- Sometimes, we need to modify behaviors of an instance dynamically
- Sub-classing could be a solution, but excessive sub-classing is a pitfall
- The decorator pattern (also known as wrapper) solves this issues
- How? By limiting sub-typing through effective aggregation

Case study

Case study

Case study

- Consider again the iterator interface

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

Case study

Case study

- Consider again the natural numbers implementation

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

Case study

Another task: iteration with offset

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

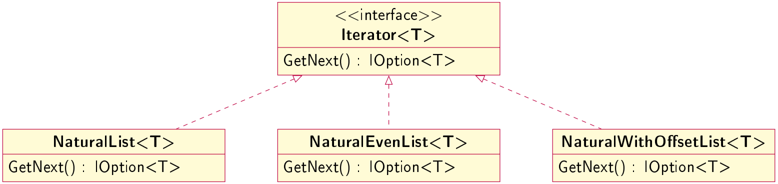
```
class NaturalWithOffsetList : Iterator<int> {
  private  int current;
  private  int offset;
  public NaturalWithOffsetList(int offset) {
    current = -1;
    this.offset = offset;
  }
  IOption<int> GetNext() {
    current = (current + 1);
    return new Some<int>((current + offset));
  }
}
```

Case study

UML

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

Case study


<<interface>>
Iterator<T>
GetNext() : IOption<T>
NaturalList<T>
GetNext() : IOption<T>
NaturalEvenList<T>
GetNext() : IOption<T>
NaturalWithOffsetList<T>
GetNext() : IOption<T>

Case study

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...

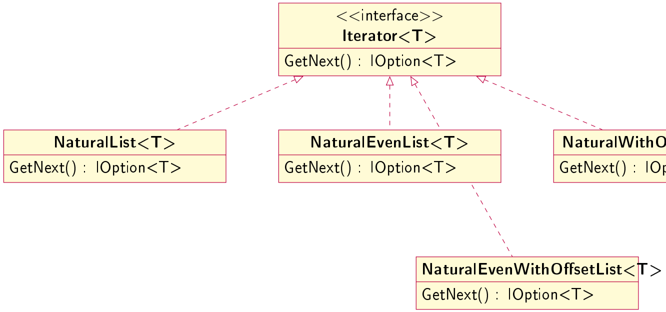
```
class NaturalEvenWithOffsetList : Iterator<int> {
  private int current;
  private int offset;
  public NaturalEvenWithOffsetList(Offset offset) {
    current = -1;
    this.offset = offset;
  }
  IOption<int> GetNext() {
    current = (current + 1);
    if(((current % 2) == 0)) {
      return new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

Case study

UML discussion

- As we can see our sub-classes are growing “horizontally”, since no reuse is used
- Lets give a the UML again

Case study



<<interface>>
Iterator<T>
GetNext() : IOption<T>
NaturalList<T>
GetNext() : IOption<T>
NaturalEvenList<T>
GetNext() : IOption<T>
NaturalWithOffsetList<T>
GetNext() : IOption<T>
NaturalEvenWithOffsetList<T>
GetNext() : IOption<T>

Case study

Iterating a range between two integers

- Imagine if we now wish to implement a new data structure RangeBetween that takes two integers A and B (where A <= B)
- We want `Range` to as well to support the offset and even behavior
- It is a very time consuming task, as we have to literally duplicate everything and to implement all possible combinations

Case study

Considerations

- Sub-typing solves our problem, but adds another one. Too many repetitions
- Every change/add requires lots of work
- Behaviors commonalities are not taken into consideration

Case study

Considerations

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

```
class NaturalEvenWithOffsetList : NaturalList, Offset, Evens {
  ...
}
```

Case study

Considerations

- This solution is not good, since the responsibilities are now not clear(see SOLID)
- Moreover, the resulting structure will become a big, bulky class (a class whose functionality does not adapt to each instance)

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

Case study

Considerations

- We need a better mechanism. Ideally we wish:
-
  - To define once our natural list
  - To define once our the even behavior
  - To define once our the 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 possible could be that 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> {
  private  Iterator<int> decorated_item;
  public Decorator(Iterator<int> decorated_item) {
    this.decorated_item = decorated_item;
  }
  public IOption<int> GetNext() {
    return decorated_item.GetNext();
  }
}
```

Case study

Idea

- We can think of the decorator as an iterator containing elements, but it does not know how to iterate such elements (indeed we could make the above `GetNext` also abstract)
- A decorator needs someone who teaches him how to iterate
- `Decorator` is only aware that the instances to iterate are those referenced by `decorated_item`

Case study

Idea

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

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.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 Even

```
class Offset : Decorator {
  private  int offset;
  public Offset(Offset offset) {
    this.offset = offset;
  }
  public override IOption<int> GetNext() {
    Option<int> current = base.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
- Indeed, they are concrete decorators. Their code is always run after a `GetNext` over a collection of numbers
- Does this happen to all collections of numbers? No, only on those who are requesting this behavior

Case study

Idea

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

Idea

- UML

Case study

Considerations

- Of course we still have sub-typing, but it is limited as our decorator help us to take out concerns (our even and offset) that are not part of the natural list description, so concerns can be reused/adapted/combined/etc.

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 show its formalization and add some final considerations

The decorator design pattern

The decorator design pattern

Formalism

- Formalism

The decorator design pattern

Formalism

- UML

The decorator design pattern

Improving our initial solution

- We can further improve the above solution. How? By making offset and even "general" as offset simply transforms and even simply filters
- Indeed, offset is a concrete **Map**
- CODE

The decorator design pattern

Improving our initial solution

- Indeed, even is a concrete **Filter**
- CODE

Conclusions

HOGESCHOOL ROTTERDAM

This is it!

The best of luck, and thanks for the attention!