

Adapting interfaces

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Adapting interfaces

Introduction

Lecture topics

- Issues arising from importing and using entities
- The adapter design pattern
- Examples and considerations
- Conclusions

Introduction

- Today we are going to study code adapters
- In particular, we are going to study how to make existing classes work with others without modifying their code
- How? By means of a design pattern: the adapter (a behavioral design pattern)
- We will see the adapter provides a clean and general mechanism that allows an interface of an existing class to be used as another interface

Adapting existing classes

- It is often the case where we need to adapt existing entities to other
- For example we wish to treat an option by means of an iterator, traditional iterator as a safe iterator, or a class belonging to a closed library with the interface required by our application
- With the only constraint that we cannot change the original implementation and the original functionalities
- Why?

Adapting existing classes

- It is often the case where we need to adapt existing entities to other
- For example we wish to treat an option by means of an iterator, traditional iterator as a safe iterator, or a class belonging to a closed library with the interface required by our application
- With the only constraint that we cannot change the original implementation and the original functionalities
- Why?
- Otherwise we might break other programs depending on such implementation

An example of similar but incompatible classes

- Consider the following two classes LegacyLine and LegacyRectangle
- Both implementing a draw method

```
1 class LegacyLine {
2     public void Draw(int x1,int y1,int x2,int y2) {
3         Console.WriteLine("line from (" + x1 + ',' + y1 + ") to (" + x2 + ',' +
4             y2 + ')');
5     }
6     class LegacyRectangle{
7         public void Draw(int x,int y,int w,int h){
8             Console.WriteLine("rectangle at (" + x + ',' + y + ") with width " + w +
9                 " and height " + h);
10        }
11    }
```


Consuming our LegacyLine and LegacyRectangle

- We now wish to consume instances of LegacyLine and LegacyRectangle in our application
- We group such instances within the same collection, so to deal with them all at once, thus to avoid duplication
- The collection is of type Object. Why? Because LegacyLine and LegacyRectangle do not share a same type

```
1  LinkedList<Object> shapes = new LinkedList<Object>();
2  shapes.Add(new LegacyLine());
3  shapes.Add(new LegacyRectangle());
4  while (shapes.Tail != null) {
5      if shapes.Head.getClass().getName().equals("LegacyLine") {
6          (LegacyLine)shapes.Head.Value.Draw(...);
7      }
8      if shapes.Head.getClass().getName().equals("LegacyRectangle") {
9          (LegacyRectangle)shapes.Head.Value.Draw(...);
10     }
11     shapes = shapes.Tail;
12 }
```

Issues with consuming LegacyLine and LegacyRectangle

- As we can see from the example consuming instances of such classes is complex and error-prone
- We could of course apply a visitor, but in this case it is not possible, since we cannot touch the their implementation
- We wish now to reduce such complexity and to achieve safeness

Consuming “safely” LegacyLine and LegacyRectangle: idea

- A solution would be to define an intermediate layer that mediates for us with instances of both LegacyLine and LegacyRectangle
- For this implementation we need first to define an interface Shape with one method signature Draw

```
1 interface Shape {  
2     void Draw(int x1,int y1,int x2,int y2);  
3 }
```

An adapter for our LegacyLine

- We declare a class Line that takes as input a LegacyLine object
- Whenever the Draw method is called also the Draw of the LegacyLine object is called

```
1 class Line : Shape {  
2     private LegacyLine underlyingLine;  
3     public Line(LegacyLine line) {  
4         this.underlyingLine = line;  
5     }  
6     public void Draw(int x1,int y1,int x2,int y2) {  
7         underlyingLine.Draw (...);  
8     }  
9 }
```

An adapter for our LegacyRectangle

- We apply the same mechanism to our LegacyRectangle

```
1 class Rectangle : Shape {  
2     private LegacyRectangle underlyingRectangle;  
3     public Rectangle(LegacyRectangle rectangle) {  
4         this.underlyingRectangle = rectangle;  
5     }  
6     public void Draw(int x1,int y1,int x2,int y2) {  
7         underlyingRectangle.Draw(...);  
8     }  
9 }
```

Consuming “safely” LegacyLine and LegacyRectangle

- The main program will now define a list of shapes containing instances of shapes each referencing an instance of either LegacyLine or LegacyRectangle

```
1 LinkedList<Shape> shapes = new LinkedList<Shape>();  
2 shapes.Add(new Line(new LegacyLine()));  
3 shapes.Add(new Rectangle(new LegacyRectangle()));  
4 while (shapes.Tail != null) {  
5     shapes.Head.Value.Draw(...);  
6     shapes = shapes.Tail;  
7 }
```

Considerations

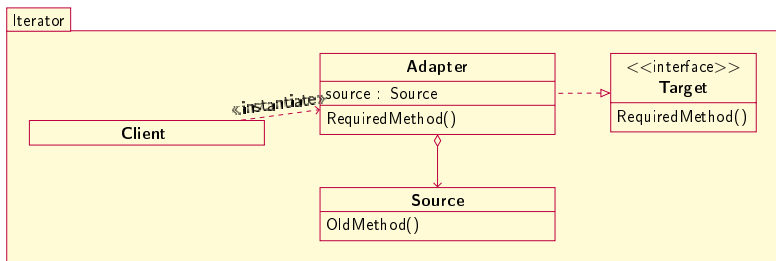
- As we can see our program now manages instances of both `LegacyLine` and `LegacyRectangle` without requiring to manually deal with their details
- This makes the code not only more maintainable but also safer, since the original implementation remains the same
- This example
- In this way our program deals with objects of type `Rectangle` and `Line` as if they are concrete `LegacyLine` and `LegacyRectangle` objects without changing its concrete functionalities

The adapter design pattern

- Is a design pattern that abstracts the just described mechanism
- By means of adapters, it allows to convert the interface of a class into another one that a client expects without changing its functionalities
- In what follows we will study such design pattern and provide a general formalization

The adapter design pattern structure

- Given two different interfaces `Source` and `textttTarget`
- An `textttAdapter` is built to, carefully, adapt `textttSource` to `textttTarget`
- The `textttAdapter` implements `textttTarget` and contains a reference to `textttSource`
- A `textttClient` interacts with the `textttAdapter` whenever it needs to interact with `textttSource` as if it is an instance of `textttTarget`
- In the following we provide a UML for such structure



Iterating an `IOption<T>`

- Adapters successfully achieve the task of making an source interface “behaving” as another
- It is the case of the option introduced in the previously
- As an option can be seen as a collection that might contains at most one element
- We can treat such option as an iterator
- How? By means of an adapter

Iterating an IOption<T>

- In this case Target is Iterator<T>, Source is IOption<T>, and Adapter is IOptionIterator<T>
- Now, GetNext returns Some only the at the first iteration and None for the rest
- Note, if we iterate a None entity we return None

```
1 class IOptionIterator<T> : Iterator<T> {
2     private IOption<T> option;
3     private bool visited = false;
4     public IOptionIterator(IOption<T> option) {
5         this.option = option;
6     }
7     IOption<int> GetNext() {
8         if visited {
9             return new None<T>();
10        }
11        else{
12            visited = true;
13            return option.Visit<IOption<T>>(() => new None<T>(),t => new Some<T>(t
14                ));
15        }
16    }
17 }
```

Considerations about reversibility

- Adapters as we can see successfully achieve the task of adapting imported and custom client interfaces without changing the imported interface
- However, it is important that adapting does not change the intended behavior of the imported interface

Considerations about reversibility

- Consider the `TraditionalIterator` and `Iterator` example
- Adapting a `TraditionalIterator` to an `Iterator` does not change the order of iteration (see previous class)
- But, adapting an `Iterator` to `TraditionalIterator` changes the order of iteration

Considerations about reversibility

- An adapter does not add or remove information, in order to preserve the correctness of the involved interface adapters
- Adapting interface should not affect their intended logical mechanisms
- Adapters are simply “bridges” to let abstractions vary independently
- Thus, **Safe(Unsafe(list)) = list** and **Unsafe(Safe(list)) = list**

Conclusion

- Code comes in different forms
- For many cases code cannot be changed: like a library, a toolkit, etc..
- Sometimes it is hard to make such code work in a specific target application (for example because it is written at a different time)
- The adapter pattern allows the adaptation of such code in a way that makes the resulting solution safe and maintainable
- How? By providing an custom adapter that mediated between the targeted client and the code to adapt

This is it!

Adapting
interfaces

The INFDEV
team

Adapting
interfaces

Introduction

The best of luck, and thanks for the
attention!