

Adapting interfaces

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Introduction

Lecture topics

- Issues arising from connecting domains
- The adapter design pattern
- Examples and considerations
- Conclusions

Issues:

- Independent domains each based on its interface(s)
- No shared code, so they cannot communicate directly
- Semantically compatible: we want to connect them

Examples:

- Legacy systems
- Different frameworks
- Closed libraries
- Etc..

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Adapter

Introduction

- Today we are going to study adapters
- In particular, we are going to study how to make existing classes work within other domains without modifying their code
- How? By means of a design pattern: the adapter (a behavioral design pattern)
- A clean and general mechanism that allows an instance of an interface to be used where another interface is expected

Adapting existing classes

- A further constraint is that we cannot change the original implementation
- Why?

Adapting existing classes

- A further constraint is that we cannot change the original implementation
- Why?
- *We might break other programs depending on such implementation*

Examples:

- An option as an iterator
- A traditional iterator as a safe iterator
- A class belonging to a closed library with the interface required by our application
- A shape in another drawing library
- ...

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 }
7 class LegacyRectangle{
8     public void Draw(int x,int y,int w,int h){
9         Console.WriteLine("rectangle at (" + x + ', ' + y + ") with width " + w +
10            " and height " + h);
11     }
12 }
```

Consuming our LegacyLine and LegacyRectangle

- Suppose we wished to build a drawing system
- We need to group lines and rectangles together, plus our own classes
- Cast to Object?

```
1 List<Object> shapes = new List<Object>();  
2 shapes.Add(new LegacyLine());  
3 shapes.Add(new LegacyRectangle());  
4 shapes.Add(new NonLegacyCircle());  
5 foreach(Object shape in shapes){  
6     if(shape is NonLegacyCircle) {  
7         (NonLegacyCircle)shape.Draw(...);  
8     }  
9     if(shape is LegacyLine) {  
10        (LegacyLine)shape.Draw(...);  
11    }  
12    if(shape is LegacyRectangle) {  
13        (LegacyRectangle)shape.Draw(...);  
14    }  
15 }
```

Issues with consuming LegacyLine and LegacyRectangle

- This technique is complex and error-prone
- We cannot even apply a visitor, since we cannot touch the implementation of Legacy*
- We wish now to reduce such complexity and to achieve safety

Safely consuming LegacyLine and LegacyRectangle: idea

- We define a mediating layer that abstracts instances of both LegacyLine and LegacyRectangle
- For this implementation we first define an interface Shape with one method signature Draw
- This interface defines the entry of our own domain

```
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
- Line exists both in the legacy and our new domain

```
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 }
```

- Our drawing system can now define a list of shapes

```
1 List<Shape> shapes = new List<Shape>();  
2 shapes.Add(new Line(new LegacyLine()));  
3 shapes.Add(new Rectangle(new LegacyRectangle()));  
4 shapes.Add(new NonLegacyCircle());  
5 foreach(Shape shape in Shapes){  
6     shape.Draw(...);  
7 }
```

- We could even extend our Shape with a visitor

```
1 interface Shape {  
2     void Draw(int x1,int y1,int x2,int y2);  
3     U Visit<U>(Func<U> onLegacyLine,Func<U> onLegacyRectangle,Func<U>  
4         onNonLegacyCircle);  
}
```

Considerations

- Instances of both `LegacyLine` and `LegacyRectangle` are now harmoniously integrated with our own framework
- Code is more maintainable, and we have not changed (and potentially broken) the legacy implementations
- Only requirement is that we never manipulate legacy instances directly, but go through `Rectangle` and `Line`
- `Rectangle` and `Line` are **adapters**

Adapter

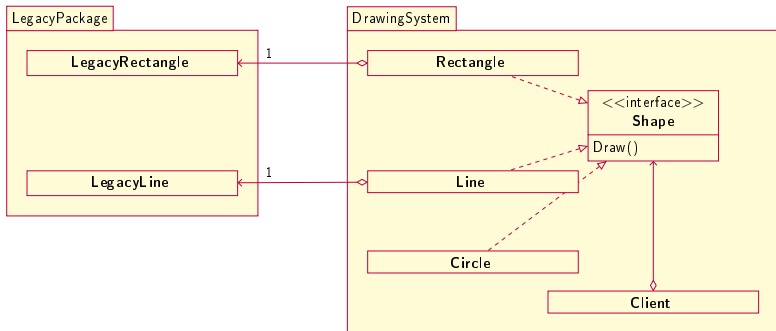
Adapting
interfaces

The INFDEV
team

INFDEV02-4
Introduction

Adapter

The adapter
design
pattern



The adapter design pattern

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

General idea

- By means of adapters, we “convert” the interface of a class into another, without touching the class sources
- In what follows we will study such design pattern and provide a general formalization

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

The adapter design pattern structure

- Given two different interfaces Source and Target
- An Adapter is built to adapt Source to Target
- The Adapter implements Target by means of a reference to Source
- A Client interacts with the Adapter whenever it a Target, but we have a Source
- In the following we provide a UML for such structure

The adapter design pattern

Adapting
interfaces

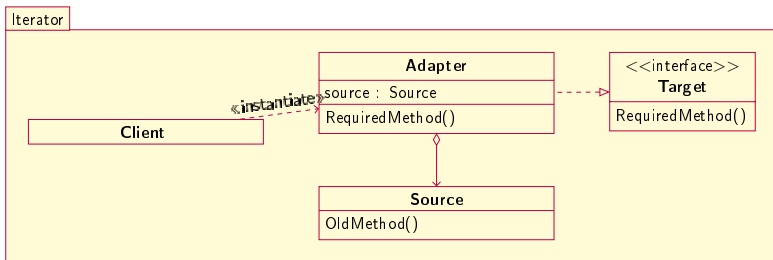
The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern



The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Example:

- Consider the `Option` data type
- It is a collection of sorts
- It could be iterated, but it does not implement an iterator!

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Making Option “iterable”, a naive approach:

- Without adapter, we need Option<T> to implement Iterator<T>

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

```
1 interface Option<T> : Iterator<T> {  
2     ...  
3 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Making Some “iterable”, a naive approach

- Calling `GetNext` on `Some` returns only once its `Value` within a `Some`

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

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Making None “iterable”, a naive approach

- Calling `GetNext` returns always `None`

```
1 class None<T> : Iterator<T> {  
2     Option<T> GetNext() {  
3         return new None<T>();  
4     }  
5     ...  
6 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Making an option “iterable”: considerations

- Is it always needed for the option to be iterable?

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Making an option “iterable”: considerations

- Is it always needed for the option to be iterable?
- No!
- According to the single responsibility principle of SOLID, Option should not include considerations regarding iteration^a
- Adapter solution is better, as it allows to extend option to any additional services required without changing the option data structure

^aThat is why we presented all the iterators through adapter in the previous lecture.

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Iterating an `Option<T>` with adapters

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

```
1 class IOptionIterator<T> : Iterator<T> {  
2     private Option<T> option;  
3     private bool visited = false;  
4     public IOptionIterator(Option<T> option) {  
5         this.option = option;  
6     }  
7     Option<T> GetNext() {  
8         if(visited) {  
9             return new None<T>();  
10        }  
11        else{  
12            visited = true;  
13            if(option.IsSome()) {  
14                return new Some<T>(option.GetValue());  
15            }  
16            else{  
17                return new None<T>();  
18            }  
19        }  
20    }
```


The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Iterating an Option<T>

- Which with visitor becomes:

```
1 class IOptionIterator<T> : Iterator<T> {
2     private Option<T> option;
3     private bool visited;
4     public IOptionIterator(Option<T> option) {
5         this.option = option;
6         this.visited = false;
7     }
8     Option<T> GetNext() {
9         if(visited) {
10             return new None<T>();
11         }
12         else{
13             visited = true;
14             return option.Visit<Option<T>>(() => new None<T>(), t => new Some<T>(t)
15                 );
16         }
17     }
18 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Considerations about bijective adapters

- Adapters map behaviors across domains
- Adapting may not change or add behaviors

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Considerations about bijective adapters

- Consider the `TraditionalIterator` and `Iterator` example

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

TraditionalIterator and Iterator

- What was the point of one and the other?
- We need both, as they both make sense within their respective contexts!

```
1 interface TraditionalIterator<T> {  
2     bool MoveNext();  
3     T GetCurrent();  
4 }  
5 interface Iterator<T> {  
6     IOption<T> GetNext();  
7 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Which in Java then becomes:

```
1 interface TraditionalIterator<T> {  
2     bool MoveNext();  
3     T GetCurrent();  
4 }  
5 interface Iterator<T> {  
6     IOption<T> GetNext();  
7 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- How do we bridge the two worlds?
- With an adapter per direction:
- MakeSafe that makes TraditionalIterator behave as Iterator
- MakeUnsafe that makes Iterator behave as TraditionalIterator

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

```
1 class MakeSafe<T> : Iterator<T> {  
2     private TraditionalIterator<T> iterator;  
3     public MakeSafe(TraditionalIterator<T> iterator) {  
4         this.iterator = iterator;  
5     }  
6     Option<T> GetNext() {  
7         if(iterator.MoveNext()) {  
8             return new Some<T>(iterator.GetCurrent());  
9         }  
10        else{  
11            return new None<T>();  
12        }  
13    }  
14 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Which in Java then becomes:

```
1 class MakeSafe<T> implements Iterator<T> {  
2     private TraditionalIterator<T> iterator;  
3     public MakeSafe(TraditionalIterator<T> iterator) {  
4         this.iterator = iterator;  
5     }  
6     Option<T> GetNext() {  
7         if(iterator.MoveNext()) {  
8             return new Some<T>(iterator.GetCurrent());  
9         }  
10        else{  
11            return new None<T>();  
12        }  
13    }  
14 }
```


The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

```
1  class MakeUnsafe<T> : TraditionalIterator<T> {  
2      private _current T;  
3      private Iterator<T> iterator;  
4      public MakeUnsafe(Iterator<T> iterator) {  
5          this.iterator = iterator;  
6      }  
7      T GetCurrent() {  
8          return _current;  
9      }  
10     bool MoveNext() {  
11         Option<T> opt = iterator.GetNext();  
12         if(opt.IsSome()) {  
13             _current = iterator.GetValue();  
14             return true;  
15         }  
16         else{  
17             return false;  
18         }  
19     }  
20 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Which in Java then becomes:

```
1  class MakeUnsafe<T> implements TraditionalIterator<T> {  
2      private _current T;  
3      private Iterator<T> iterator;  
4      public MakeUnsafe(Iterator<T> iterator) {  
5          this.iterator = iterator;  
6      }  
7      T GetCurrent() {  
8          return _current;  
9      }  
10     bool MoveNext() {  
11         Option<T> opt = iterator.GetNext();  
12         if(opt.IsSome()) {  
13             _current = iterator.GetValue();  
14             return true;  
15         }  
16         else{  
17             return false;  
18         }  
19     }  
20 }
```

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- What is the behavior of new MakeSafe(new MakeUnsafe(it)) for a generic iterator it?

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- What is the behavior of `new MakeSafe(new MakeUnsafe(it))` for a generic iterator `it`?
- No change! The two behave exactly the same!

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- What is the behavior of new `MakeUnsafe(new MakeSafe(it))` for a generic iterator `it`?

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- What is the behavior of new MakeUnsafe(new MakeSafe(it)) for a generic iterator it?
- No change! The two behave exactly the same!

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Bridging TraditionalIterator and Iterator

- This semantic neutrality is common to all adapters: no information is added or removed
- Adapters preserve the full behavior of the adapted interface
- Adapters are simply “bridges” between domains, and contain no domain logic themselves

The adapter design pattern

Adapting
interfaces

The INFDEV
team

INFDEV02-4

Introduction

Adapter

The adapter
design
pattern

Conclusion

- Code usually is partitioned in (closed) domains
- Sometimes it cannot be changed: a library, a framework, etc..
- Sometimes it is hard to make existing code work in a specific target application (for example because it is written with other conventions or is simply legacy)
- The adapter pattern allows the adaptation of such code in a way that makes the resulting solution flexible and safe
- How? By providing a neutral adapter that mediates between the target and source domains

The best of luck, and thanks for the
attention!