Adapting interfaces

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFSEN02-2

Introduction

Introduction

Lecture topics

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

Introduction

Issues:

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

Introduction

Examples:

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

Adapter

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

Adapter

Adapting existing classes

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

Adapter

Adapting existing classes

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

Adapter

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
- ...

Adapter

An example of similar but incompatible classes

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

```
class LegacyLine {
  public void Draw(int x1,int y1,int x2,int y2) {
    Console.WriteLine("line␣from␣(" + x1 + ',' + y1 + ")␣to␣(" + x2 + ',' + y2 + ')'");
␣␣}
}
class␣LegacyRectangle␣{
␣␣public␣void␣Draw(int␣x,int␣y,int␣w,int␣h)␣{
␣␣␣␣Console.WriteLine("rectangle at ("␣+␣x␣+␣','␣+␣y␣+␣") with width "␣+␣w␣+␣" and height "␣+␣h);
␣␣}
}
```

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`?

```
List<Object> shapes = new List<Object>();
shapes.Add(new LegacyLine());
shapes.Add(new LegacyRectangle());
shapes.Add(new NonLegacyCircle());
foreach(Object shape in shapes){
  if(shape is NonLegacyCircle) {
    (NonLegacyCircle)shape.Draw(...);
  }
  if(shape is LegacyLine) {
    (LegacyLine)shape.Draw(...);
  }
  if(shape is LegacyRectangle) {
    (LegacyRectangle)shape.Draw(...);
  }
 }
```

Adapter

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

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

Adapter

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

```
class Line : Shape {
  private LegacyLine underlyingLine;
  public Line(LegacyLine line) {
    this.underlyingLine = line;
  }
  public void Draw(int x1,int y1,int x2,int y2) {
    underlyingLine.Draw(...);
  }
}
```

Adapter

An adapter for our LegacyRectangle

- We apply the same mechanism to our LegacyRectangle

```
class Rectangle : Shape {
  private LegacyRectangle underlyingRectangle;
  public Rectangle(LegacyRectangle rectangle) {
    this.underlyingRectangle = rectangle;
  }
  public void Draw(int x1,int y1,int x2,int y2) {
    underlyingRectangle.Draw(...);
  }
}
```

Adapter

- Our drawing system can now define a list of shapes

```
List<Shape> shapes = new List<Shape>();
shapes.Add(new Line(new LegacyLine()));
shapes.Add(new Rectangle(new LegacyRectangle()));
shapes.Add(new NonLegacyCircle());
foreach(Shape shape in Shapes){
  shape.Draw(...);
 }
```

Adapter

- We could even extend our Shape with a visitor

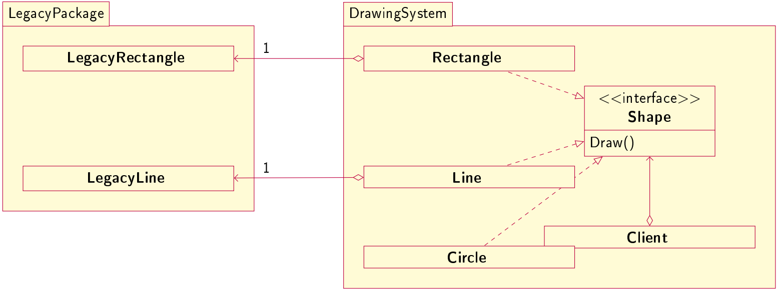
```
interface Shape {
  void Draw(int x1,int y1,int x2,int y2);
  U Visit<U>(Func<U> onLegacyLine,Func<U> onLegacyRectangle,Func<U>
       onNonLegacyCircle);
}
```

Adapter

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


LegacyPackage
LegacyRectangle
LegacyLine
1
1
DrawingSystem
Rectangle
<<interface>>
Shape
Draw()
Line
Client
Circle

The adapter design pattern

The adapter design pattern

General idea

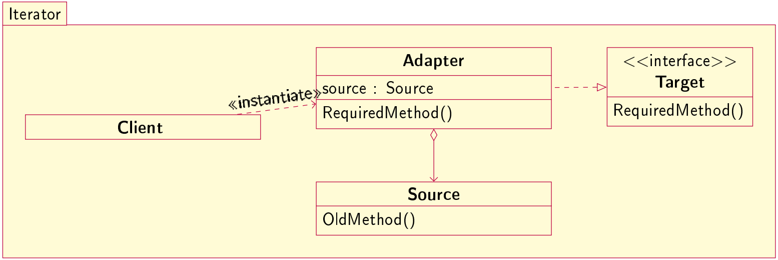
- By means of adapters, we "convert" the interface of a class into another, without touching the class sources
- In what follows we will study such design pattern and provide a general formalization

The adapter design pattern

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 texttSource
- A `Client` interacts with the `Adapter` whenever it a `Target`, but we have a `Some`
- In the following we provide a UML for such structure

The adapter design pattern



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 interator!

The adapter design pattern

Making Option "iterable", a naive approach:

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

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

```
interface Option<T> : Iterator<T> {
  ...
}
```

Adapting interfaces
The INFDEV team
INFSEN02-2
Introduction
Adapter
The adapter design pattern

Making Some “iterable”, a naive approach

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

```
class Some<T> : Iterator<T> {
  private T value;
  private bool visited = false;
  public Some(T value) {
    this.value = value;
  }
  Option<T> GetNext() {
    if(visited) {
      return new None<T>();
    }
    else{
      visited = true;
      return new Some<T>(value);
    }
  }
  ...
}
```

The adapter design pattern

Making None “iterable”, a naive approach

- Calling `GetNext` returns always `None`

```
class None<T> : Iterator<T> {
  Option<T> GetNext() {
    return new None<T>();
  }
  ...
}
```

Making an option "iterable": considerations

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

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

[a] That is why we presented all the iterators through adapter in the previous lecture.

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`

```
class IOptionIterator<T> : Iterator<T> {
  private Option<T> option;
  private bool visited = false;
  public IOptionIterator(Option<T> option) {
    this.option = option;
  }
  Option<T> GetNext() {
    if(visited) {
      return new None<T>();
    }
    else{
      visited = true;
      if(option.IsSome()) {
        return new Some<T>(option.GetValue());
      }
      else{
        return new None<T>();
      }
    }
```

Iterating an Option<T>

- Which with visitor becomes:

```
class IOptionIterator<T> : Iterator<T> {
  private Option<T> option;
  private bool visited = false;
  public IOptionIterator(Option<T> option) {
    this.option = option;
  }
  Option<T> GetNext() {
    if(visited) {
      return new None<T>();
    }
    else{
      visited = true;
      return option.Visit<Option<T>>(() => new None<T>(),t => new Some<T>(t)
           );
    }
  }
}
```

Considerations about bijective adapters

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

The adapter design pattern

Considerations about bijective adapters

- Consider the `TraditionalIterator` and `Iterator` example

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!

```
interface TraditionalIterator<T> {
  void MoveNext();
  bool HasNext();
  T GetCurrent();
}
interface Iterator<T> {
  IOption<T> GetNext();
}
```

HOGESCHOOL ROTTERDAM

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


```
class MakeSafe<T> : Iterator<T> {
  private TraditionalIterator<T> iterator;
  public MakeSafe(TraditionalIterator<T> iterator) {
    this.iterator = iterator;
  }
  Option<T> GetNext() {
    if(iterator.MoveNext()) {
      return new Some<T>(iterator.GetCurrent());
    }
    else{
      return new None<T>();
    }
  }
}
```

Which in Java then becomes:

```
class MakeSafe<T> implements Iterator<T> {
  private TraditionalIterator<T> iterator;
  public MakeSafe(TraditionalIterator<T> iterator) {
    this.iterator = iterator;
  }
  Option<T> GetNext() {
    if(iterator.MoveNext()) {
      return new Some<T>(iterator.GetCurrent());
    }
    else{
      return new None<T>();
    }
  }
}
```

The adapter design pattern


```
class MakeUnsafe<T> : TraditionalIterator<T> {
  private _current T;
  private Iterator<T> iterator;
  public MakeUnsafe(Iterator<T> iterator) {
    this.iterator = iterator;
  }
  T GetCurrent() {
    return _current;
  }
  bool MoveNext() {
    Option<T> opt = iterator.GetNext();
    if(opt.IsSome()) {
      _current = iterator.GetValue();
      return true;
    }
    else{
      return false;
    }
  }
}
```

The adapter design pattern

Which in Java then becomes:

```
class MakeUnsafe<T> implements TraditionalIterator<T> {
  private _current T;
  private Iterator<T> iterator;
  public MakeUnsafe(Iterator<T> iterator) {
    this.iterator = iterator;
  }
  T GetCurrent() {
    return _current;
  }
  bool MoveNext() {
    Option<T> opt = iterator.GetNext();
    if(opt.IsSome()) {
      _current = iterator.GetValue();
      return true;
    }
    else{
      return false;
    }
  }
}
```

Bridging TraditionalIterator and Iterator

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

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

Bridging TraditionalIterator and Iterator

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

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

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

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

HOGESCHOOL ROTTERDAM

This is it!

The best of luck, and thanks for the attention!