Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Introduction

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Dev 4

Dev 4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Lecture topics

- Intro to DEV4
- Design patterns introduction
- The visitor design pattern
- Course agenda
- Conclusions

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Intro to DEV4

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Exam

- written exam
- 3 open questions
- stack/heap, type system, and design patterns
- no grade: go (score$\geq$75) or no go (otherwise)

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Exercises

- exercises to prepare step-by-step
- builds up to actual practicum
- there is no grade for this

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Assignments

- a connected series of programming tasks
- build a GUI framework
- **mandatory**, but with no direct grade

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Oral

- the oral is entirely based on the assignments
- we remove some pieces of code from the working solutions and you fill them back in
- the oral gives you the final grade for the course

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Expected study effort

- between 10 and 20 **net**[a] hours a week
- read every term on the slides and every sample
- if you do not understand it perfectly, either ask a teacher, google, or brainstorm with other students
- every sample of code on the slides you should both **understand** and **try out** on your machine

[a] No, 9gag does not count even if the slides are open on another monitor

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

What you have done so far

- Encapsulation, polymorphism, subtyping, generics, etc.;
- Ways to express interactions among entities.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

What is coupling?

- Interactions between entities affect maintainability;
- The more the interactions, the higher the likelihood of having bugs;
- This phenomenon is known as coupling.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

What is coupling?

- If changing something in a program requires changing something else, then we have coupling.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Sort of coupling

- High, which is undesirable;
- Low, which is our target.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

High-coupling

- As the number of interaction between two classes **A** and **B** increases, the coupling between them increases as well;
- This translates into: whenever **A** changes, the likelihood to erroneously change **B** is "high";
- Threfore, likely more bugs.

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

High-coupling

- The class `Driver` contains a field of type `Car`
- The class `Driver` has visibility of all `Car` public methods and fields, such as the cilinders status;
- Move is really the only relevant bit here

```
1  class Driver {
2    private Car car;
3    void Drive() {
4      public this.car.Move();
5    }
6  }
7  class Car {
8    public CilindersStatus cilinders;
9    public void Move() {
10     ...
11   }
12   ...
13 }
```

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Low-coupling

- The number of interaction between two classes **A** and **B** is limited to a series of methods provided by an interface;
- This translates into: whenever **A** changes, the likelihood to erroneously change **B** is "low", since **A** knows little about **B**.

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Low-coupling

- The class `Driver` contains a polymorphic type `Vehicle`
- The interaction between `Driver` and `Car` is restricted to the interface method `Move`;
- No cilinders;
- Also electric cars (no cilinders in electric cars).

```
1  class Driver {
2    private Vehicle vehicle;
3    void Drive() {
4      public this.vehicle.Move();
5    }
6  }
7  interface Vehicle {
8    void Move();
9  }
10 class Car : Vehicle {
11   public void Move() {
12     ...
13   }
14 }
```

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Low vs High coupling

- As the number of entities increases, the number of interactions increases;
- More precisely, given N classes, it is:

$$I \simeq \left(\sum_{i=2}^{N} \frac{N!}{2(N-i)!} \right)$$

- It is a very big number!

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Low vs High coupling

- Consider a very simple program with only 4 classes
- This number is given by

$$I \simeq \frac{4!}{2(4-2)!} + \frac{4!}{2(4-3)!} + \frac{4!}{2(4-4)!} = 30$$

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Achieving low-coupling

- Maintaining code is hard and expensive
- Low coupling results in easily maintainable code
- What seems desirable when dealing with software development is to keep coupling between entities as low as possible

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern

Maintainability in code

- Is an important aspect in development;
- It affects costs of fixing bugs and changing functionalities.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Polymorphism for reducing coupling in programs

- We can control interactions by means of an interface that hides the specifics of some classes
- Every entity interacts with another only through small "windows" (defined as interfaces), each exposing a specific and controlled behavior.

Intro to DEV4

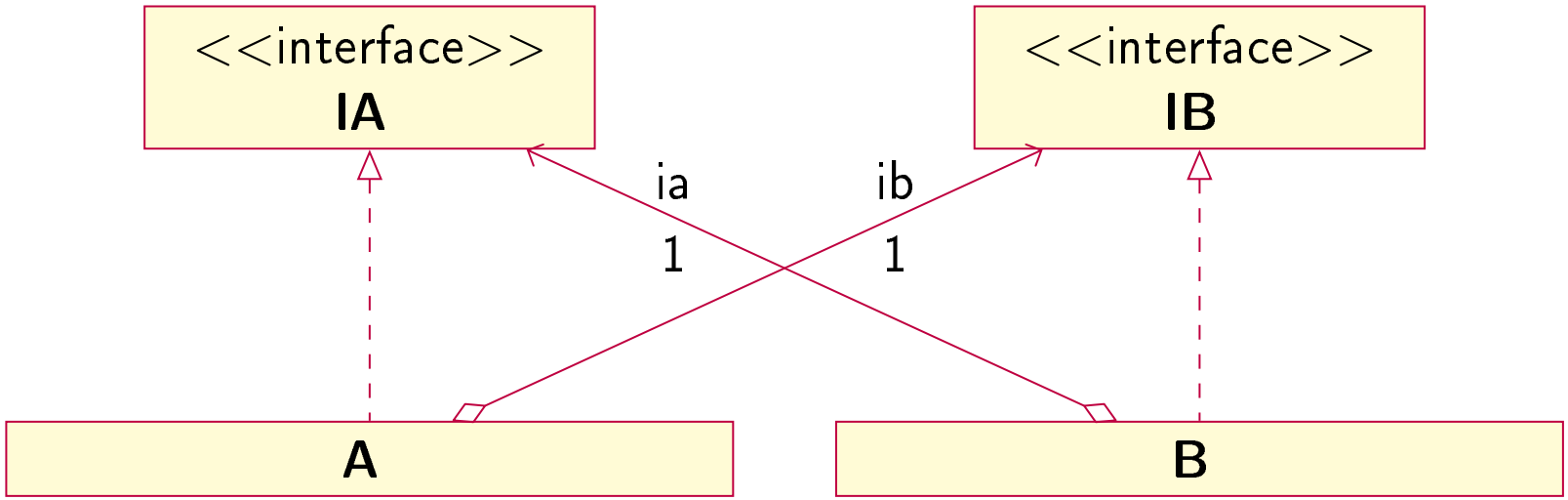
Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

A general view of low-coupling

- Given two classes `A` and `B`;
- `A` interacts with an `I_B` interface, whenever `A` needs to interact with an instance of type `B`;
- `B` interacts with an `I_A` interface, whenever `B` needs to interact with an instance of type `A`.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course



Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

```
1  class Driver {
2    private Vehicle vehicle;
3    void Drive() {
4      public this.vehicle.Move();
5    }
6  }
7  interface Vehicle {
8    void Move();
9  }
10 class Car : Vehicle {
11   private Engine engine;
12   public void Move() {
13     ...
14   }
15 }
```

- The driver (class B) can interact with a vehicle (interface IA);
- The engine, which should not be accessible outside the car, is not mentioned in the interface, so the driver cannot interact with it.

Intro to DEV4

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Recurrent patterns in objects interactions

- Disciplined interactions such as the one above tend to exhibit some recurring high level structures;
- Such structures are known under the umbrella term of **design patterns**.

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Design Patterns

- Design patterns in short are: ways to capture recurring patterns for expressing controlled interactions between entities;
- We will now see a specific example of such a pattern.

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Our first design pattern

Our first design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Choosing in the presence of polymorphism

- As you already know polymorphism is a powerful mechanism that allows decomposition and code reuse;
- Sometimes though, we need to go "back" from general instances to concrete ones[a].

[a] Cat is Animal. Cat is specific. Animal is general.

Our first design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Why is choosing concrete types so problematic?

- Mainly because a general type has no information about what classes are implementing it.

```
1  interface Vehicle {
2    void Move();
3  }
4  class Car : Vehicle {
5    ...
6  }
7  class Bike : Vehicle {
8    ...
9  }
```

- Given an instance `v` of type `Vehicle`, what can we say about the concrete type of `v`?
- Is it a `Car` or a `Bike`?
- What if we want to turn on the airco of `v` if it is a `Car`?

Our first design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Safe choice in the presence of polymorphism

- We need a mechanism that allows us to manipulate polymorphic instances as if they were concrete;
- Concrete instances are the only ones who know their identity, so we allow them to choose from a series of given "options".

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting Option's

Visiting Option's

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

The IOption<T> data structure

- Is used when an actual value of type `T` might or might not be variable;
- It is also called "reified null" or "null object".

Visiting Option's

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Examples of usage

- The following code illustrates the use of the option type;
- In this case we are capturing the number 5 within a Some<int> object;

```
1 IOption<int> a_number = new Some<int>(5);
```

Examples of usage

- In this case we capture the "nothing" common to all values of type int within a None<int> object;

```
1 IOption<int> another_number = new None<int>();
```

Visiting Option's

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Some<T> and None<T>

- Both types implement the `IOption<T>` data structure;

```
1  class Some<T> : IOption<T> {
2    public T value;
3    public Some(T value) {
4      this.value = value;
5    }
6    ...
7  }
```

```
1  class None<T> : IOption<T> {
2    public None() {
3    }
4    ...
5  }
```

Visiting Option's

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

IOption<T>

- Is an interface that represents both absence and presence of data of type `T`;
- We cannot give direct access to the `T` value here as `None` could not implement it!

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

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting Options without lambdas

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting an IOption<T>

- We add a method `Visit` to the interface that accepts as input a "Visitor" (an `IOptionVisitor<T, U>`) and returns a generic result;
- The visitor object will able to identify the concrete type of the option (`Some` or `None`) and manipulate it accordingly[a].

[a] **Note**, in many literature this `Visit` method is generally called `Accept`

```
1 interface Option<T> {
2   U Visit<U>(IOptionVisitor<T, U> visitor);
3 }
```

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

What is an IOptionVisitor<T, U>?

- An interface that provides a series of methods, one for each concrete class;
- In our case we have two signatures one for visiting a concrete `Some` instance and one for the `None`.

```
1 interface OptionVisitor<T, U> {
2   U OnSome<U>(T value);
3   U OnNone<U>();
4 }
```

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern

A concrete visitor - PrettyPrinterIOptionVisitor<int, string>

- Provides a pretty printer for options containing integers.

```
1 class PrettyPrinterOptionVisitor : IOptionVisitor<int, string> {
2   public string OnSome<string>(int value) {
3     return value.ToString();
4   }
5   public string OnNone<string>() {
6     return "I'm␣none..";
7   }
8 }
```

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern

Visiting a None<T>

- When visited, None informs its visitor of its identity by calling on onNone.

```
1 class None<T> : IOption<T> {
2   public U Visit<U>(IOptionVisitor<T, U> visitor) {
3     return visitor.onNone();
4   }
5 }
```

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting a Some<T>

- When visited, Some informs its visitor of its identity by calling on onSome.

```
1 class Some<T> : IOption<T> {
2   public T value;
3   public Some(T value) {
4     this.value = value;
5   }
6   public U Visit<U>(IOptionVisitor<T, U> visitor) {
7     return visitor.onSome(this.value);
8   }
9 }
```

Visiting Options without lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Testing out our IOption<T>

- The next line shows how to use our option to capture numbers and define operations over it;
- More precisely we instantiate a `PrettyPrinterOptionVisitor`, which is then used to visit a `Some` containing the number 5.

```
1 IOptionVisitor<int, int> opt_visitor = new PrettyPrinterIOptionVisitor<int,
      string>();
2 IOption<int> number = new Some<int>(5);
3 number.Visit(opt_visitor);
```

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting Options lambdas

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting an IOption<T>

- Visiting also can be simplified;
- We give directly the methods to choose from;
- One less interface and trivial classes.

```
1 interface Option<T> {
2   U Visit<U>(Func<U> onNone,Func<T, U> onSome);
3 }
```

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Visiting a None<T>

- None simply selects onNone.

```
1 class None<T> : IOption<T> {
2   public U Visit<U>(Func<U> onNone,Func<T, U> onSome) {
3     return onNone();
4   }
5 }
```

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern

Visiting a Some<T>

- Some simply selects onSome.

```
1  class Some<T> : IOption<T> {
2    private T value;
3    public Some(T value) {
4      this.value = value;
5    }
6    public U Visit<U>(Func<U> onNone,Func<T, U> onSome) {
7      return onSome(value);
8    }
9  }
```

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Testing out our IOption<T>

- String conversion is now very streamlined.

```
1 IOption<int> number = new Some<int>(5);
2 int inc_number = number.Visit(() => "I am None...",x => x.toString());
```

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

A concrete visitor - LambdaIOptionVisitor<T, U>

- We can adapt the "non-lambda" visitor that we say earlier so that it accepts lambda's as well.

```
1  class LambdaOptionVisitor<T, U> : IOption<T> {
2    private Func<T, U> oneSome;
3    private Func<U> onNone;
4    public LambdaOptionVisitor(Func<T, U> onSome,Func<U> onNone) {
5      this.onNone = onNone;
6      this.onSome = onSome;
7    }
8    public U onSome<U>(T value) {
9      return onSome(value);
10   }
11   public U onNone<U>() {
12     return onNone();
13   }
14 }
```

Visiting Options lambdas

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

More sample

- Can be found on GIT under the folder: Design Patterns Samples CSharp and also Java.

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

The visitor design pattern

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

The general idea

- What we have seen so far is an example implementing the *visitor* design pattern;
- It allows the recovery of "lost-type" information from a general instance back to specifics;
- The recovery is based on the actually activation of one of the multiple concrete options available.

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

How do we define it (lambda version)? (Step 1)

- Given: $C_1, ..., C_n$ classes implementing a common interface I;
- Every class C_i has fields $f^i_1, .., f^i_{m_i}$

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

How do we define it (lambda version)? (Step 2)

- We now add to I a method `Visit` that returns a result of type `U`;
- `Visit`, which is the common to all classes implementing I, picks the right option based on its concrete shape;
- Since we do not know what the visit will result in, then we return a result of a generic type `U`

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

How do we define it (lambda version)? (Step 2)

- The `Visit` method accepts as input one function per concrete implementation;
- Each such function depends on the fields of the concrete instance and produces a result of type `U`.

```
1  interface I
2    {
3      U Visit<U>(Func<FieldsC_1, U> onC_1,
4                 ...,
5                 Func<FieldsC_N, U> onC_N);
6    }
```

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

How do we implement it (lambda version)? (Step 3)

- Every class implementing the interface `I` has the task now to implement the `Visit` method, by selecting and calling the appropriate argument.

```
1  class C1
2        : I
3    {
4      F_1 f1;
5      ...
6      F_m fm;
7      U Visit<U>(Func<FieldsC1, U> onC1,
8                 ...,
9                 Func<FieldsCN, U> onCN){
10       onC1(f1,..,fm);
11     }
12   }
```

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

How do we use it (lambda version)? (Step 4)

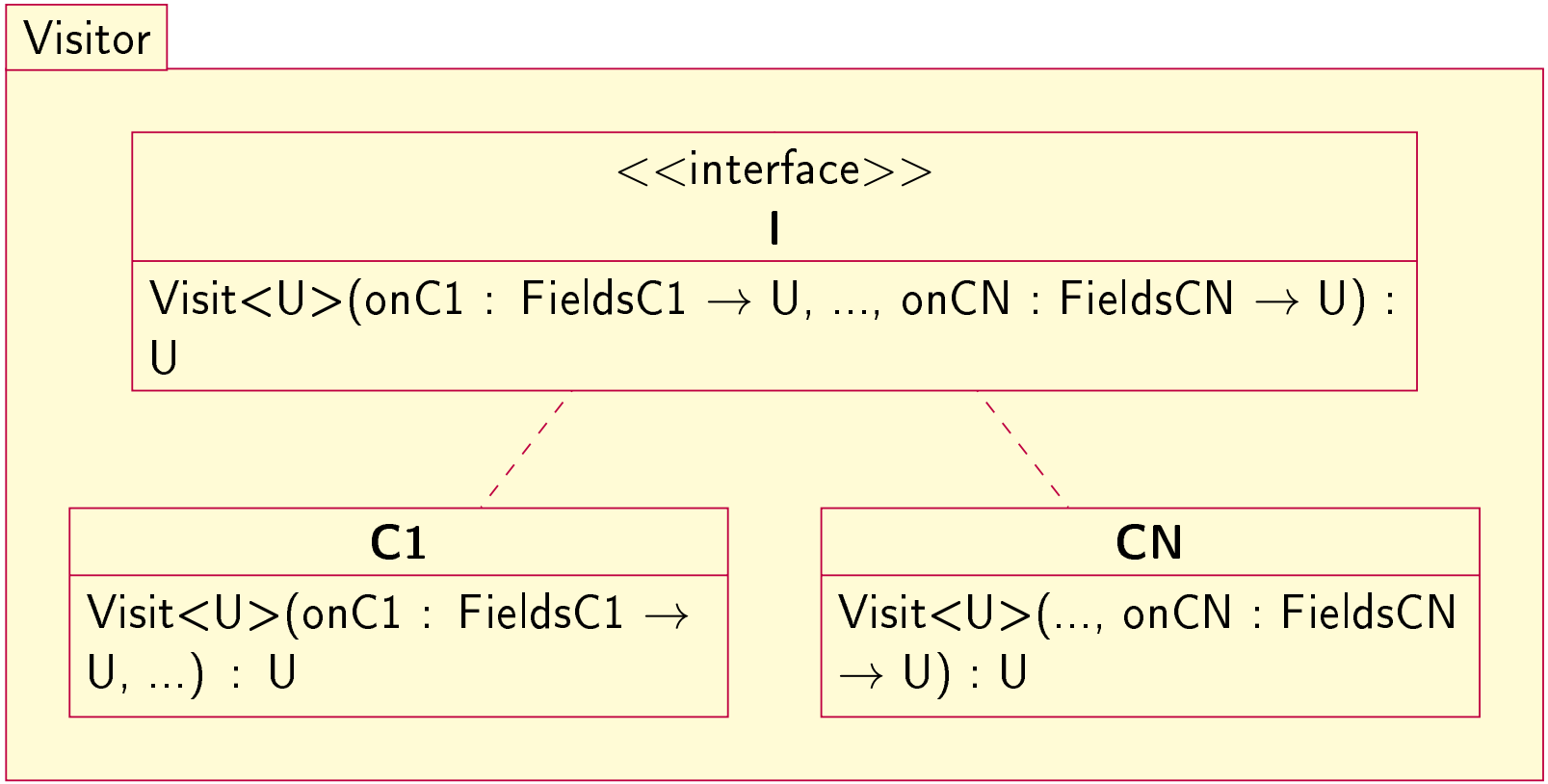
- Every time we want to consume an instance of type `M` we have to `Visit` it.

```
1  I i = ...;
2  ...
3  U result =
4    m.Visit(
5      i_1 => b_1,
6      ...,
7      i_N => b_n);
```

- Every argument of the visit becomes a function that is triggered depending on the concrete type of `i`;
- `i_i` are the fields of the concrete instance C_i;
- `b_i` is the block of code to run when a visit on an instance of a concrete type C_i is needed.

The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course



The visitor design pattern

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Final considerations

- The visitor patterns provides us with a mechanism to safely manipulate polymorphic instances;
- This mechanism is transparent and safe, as there always will be an appropriate function to call;
- The instance itself is able to select the proper implementation among the input arguments of the visitor method without any complexity or risks.

Introduction

The INFDEV team

Dev 4

Intro to DEV4

Our first design pattern

Visiting Option's

Visiting Options without lambdas

Visiting Options lambdas

The visitor design pattern

Course

Course structure

Course structure

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Final considerations

- Lectures
  - Intro to design patterns - Visiting polymorphic instances (1 lecture) TODAY
  - Iterating collections - Iterator (1 lecture)
  - Entities construction and event management - Factory + Observer (1 lecture)
  - Building state machines - Strategy (1 lecture)
  - Extending behaviors - Decorator over Strategy (1 lecture)
  - Composing behaviours - Adapter over Strategy and input (1 lecture)
  - Live coding class (1 optional lecture)
- Assignment

Course structure

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Conclusions

- Coupling in code is dangerous;
- Unmanaged interactions might introduce bugs;
- Interfaces are powerful means to control interactions.

Course structure

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

Conclusions

- Software engineering techniques (called design patterns) have been developed to achieve low-coupling by effectively using interfaces;
- This is going to be the topic of this course;
- We will study a series of basic design patterns, used in many applications.

This is it!

Introduction
The INFDEV team
Dev 4
Intro to DEV4
Our first design pattern
Visiting Option's
Visiting Options without lambdas
Visiting Options lambdas
The visitor design pattern
Course

The best of luck, and thanks for the attention!