Introduction

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

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

Introduction

What you have done so far?

- Encapsulation, polymorphism, subtyping, generics, etc.;
- Powerful ways to express interactions among objects.

Introduction

What we have not told you?

- Maybe you have already noticed it;
- Interactions affect coupling.

Introduction

What is coupling?

- If changing one module in a program requires changing another module, then we have coupling.

Introduction

High-coupling

- As the interaction surface between two classes **A** and **B** increases, the coupling between them increases as well;
- This translates into: whenever **A** changes the chance to erroneously change **B** is "high";
- Thus, the amount of bugs.

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

Introduction

Low-coupling

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

```
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 }
```

Low vs High coupling

- As the amount of entities increase, the of amount interactions increases (especially if the interfaces are not clear or not used at all);
- How much?
- It is a very big number (we are talking about an exponential function) depending on the amount of interacting objects;
- More precisely, given C classes, it is:

$$O\left(\sum_{1<k\leq C} \frac{C!}{2(C-k)!}\right)$$

Finding the right amount of coupling

- One could argue that: to avoid coupling we can put everything in one big class;
- Unfortunately this is completely true, since we can have coupling also within a single class.

Achieving low-coupling

- What seems desirable when dealing with software development is to keep coupling (our interactions) among entities as low as possible;
- Why?
- To mainly keep code maintainable.

Introduction

Maintainability in code

- Is an important aspect in development;
- It affects costs, code customization, bug fixing, etc.

Achieving low-coupling

- How how can we reduce the interaction surface among objects??
- We can use polymorphism, as seen in the last example, as a tool for specifying interaction surfaces.

Low-coupling a general view

- 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`.
- UML MISSING

Introduction

Polymorphism for taming coupling in programs

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

Introduction


```
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 can yes interact with a vehicle, but only with its public `Move` method;
- The `engine`, which should not be accessible outside the car, is not mentioned in the interface, so the driver cannot interact with it.

Introduction

Recurrent patterns in objects interactions

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

Introduction

Design Patterns

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

Our first design pattern

Our first design pattern

Choosing in the presence of polymorphism

- As you already know polymorphism is a powerful mechanism that allows decomposition and code reuse;
- However, polymorphism becomes dangerous when given a general[a] instance we have to choose what its specific shape is.

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

Our first design pattern

Why is choosing concrete types so dangerous?

- 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 lights of the car of `v`?

Our first design pattern

Safe choice in the presence of polymorphism

- We need a mechanism that allows us to manipulate polymorphic instances as id 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".

Visiting Option's without lambda's

Visiting Option's without lambda's

Safe choice in the presence of polymorphism

- UML + Examplanation

Visiting Option's with lambda's

Visiting Option's with lambda's

Safe choice in the presence of polymorphism

- UML + Examplanation

The visitor design pattern

The visitor design pattern

The general idea

- What we have seen so far are all examples 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 actualy activation of one of the multiple "options";
- The options can be instances of some concrete visitor interface, or (more elegantly) lambda's;
- We will for now on focus on the lambda implementation.

The visitor design pattern

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

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

- We add to M a method `Visit` with returning type `U` (parametric as well) accepting as many arguments as the possible concrete classes.
- Every argument is a function implementing operations that depend on a concrete instance.

```
1 interface M<Input_1, Input_2, ..., Input_N>
2   {
3     U Visit<U>(Func<Input_1, U> onObj1,
4                ...,
5                Func<Input_N, U> onObjN);
6   }
```

The visitor design pattern

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

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

```
1  class Input1<Input_1, Input_2, ..., Input_N>
2            : M<Input_1, Input_2, ..., Input_N>
3    {
4      Input_1 value;
5      U Visit<U>(Func<Input_1, U> onObj1,
6                 ...,
7                 Func<Input_N, U> onObjN){
8        onObj1(this.value);
9      }
10   }
```

The visitor design pattern

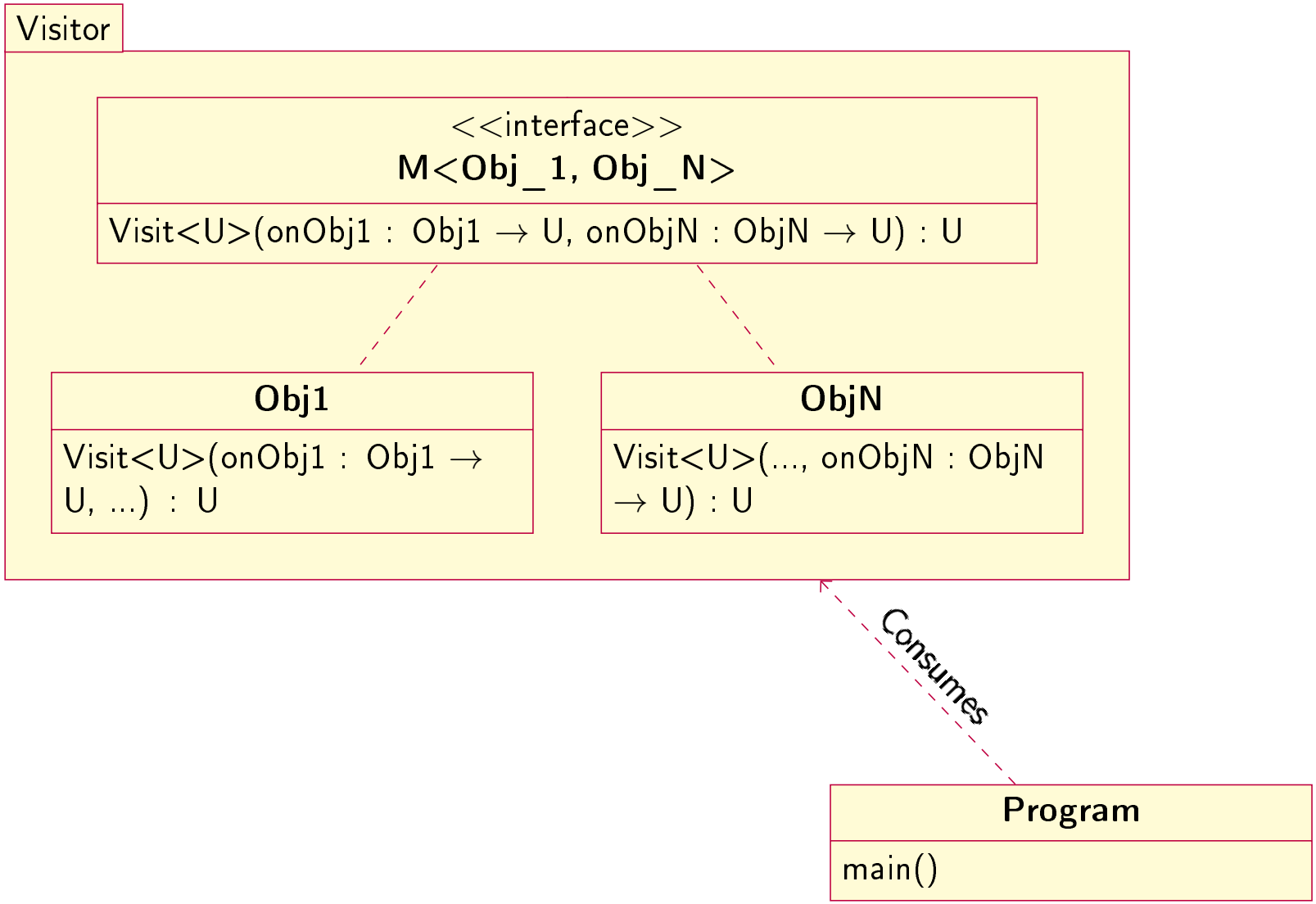
How do we use it (lambda version)? (Step 4)

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

```
1 M<Input_1, Input_2, ..., Input_N> m;
2 ...
3 m.Visit(
4  i_1 => reaction in case m is of type Input_1,
5  ...,
6  i_N => reaction in case m is of type Input_N);
```

- Every argment of the visit now becomes a code that is triggered depending on the concrete type of `m`.

The visitor design pattern



The visitor design pattern

Final considerations

- The visitor patterns allows provides us a mechanism that allows us to manipulate polymorphic instances without running into a wrong choice.
- From the caller point of view: this mechanism is transparent and safe. No matter what is the concrete instance, there will be always a proper implementation capturing its behavior.
- From the callee point of view: no one better the instance iself is able to select the proper implementaiton among the input arguments of the visitor method.

The visitor design pattern

Conclusions

- Coupling in code is dangerous.
- Unmanaged interactions might potentially introduce bugs and so make the code unmaintainable.
- Interfaces as mean to control interactions.
- Software engineering techniques (called design patterns) have been developed to achieve low-coupling by effectively using interfaces.

The visitor design pattern

Conclusions

- Coupling in code is dangerous.
- Unmanaged interactions might potentially introduce bugs and so make the code unmaintainable.
- Interfaces as mean to control interactions.
- Software engineering techniques (called design patterns) have been developed to achieve low-coupling by effectively using interfaces.
- This is going to be the topic for this course.
- We will study a series of basic design patterns, used in many realities, through concrete applications given as class examples and as homeworks.

Course structure

Course structure

Conclusions

- Lectures
  - Intro to design patterns (1 lecture) TODAY
  - Entities construction - Factory (1 lecture)
  - Generalizing behaviors - Adapter (1 lecture)
  - Extending/Composing behaviors - Decorator (1 lecture)
  - Composing patterns - MVC, MVVM (1 lecture)
  - Live coding class (1 lecture)
- Assignment
  - Build a GUI application containing interactive buttons.

This is it!

The best of luck, and thanks for the attention!