Introduction

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

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?

- Interactions between program modules affect maintainability
- The higher the interactions, the higher is the chance of having bugs
- This phenomenon is knowkn as coupling

What is coupling?

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

Introduction

High-coupling

- As the amount of interaction 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";
- More bugs

High-coupling

- The class `Driver` contains a field of type `Car`
- The class `Driver` has visibility of all `Car` public methods and fields
- The interaction between `Driver` and `Car` should be limited to the `Move` method

```
class Driver {
  private Car car;
  void Drive() {
    public this.car.Move();
  }
}
class Car {
  public void Move() {
...
  }
}
```

Low-coupling

- The amount 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 chance to erroneously change **B** is “low”, since **A** know little about **B**.

Introduction

Low-coupling

- The class `Driver` contains a polymorphic type `Vehicle`
- The interaction between `Driver` and `Car` is restricted to the interface method `Move`

```
class Driver {
  private Vehicle vehicle;
  void Drive() {
    public this.vehicle.Move();
  }
}
interface Vehicle {
  void Move();
}
class Car : Vehicle {
  public void Move() {
...
  }
}
```

Low vs High coupling

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

$$I \sim \left(\sum_{k=2}^{C} \frac{C!}{2(C-k)!} \right)$$

Low vs High coupling

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

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

Introduction

- One could argue that: to avoid coupling we can put everything in one big class;
- Unfortunately this does not solve the problem, since we can have coupling also within a single class.
- Parts of the class `Driver` still have complete visibility on the rest of the class

```
class Driver {
  private Vehicle vehicle;
  void Drive() {
    public this.vehicle.Move();
  }
  public void Move() {
...
  }
}
```

Achieving low-coupling

- Maintaining code is hard and expensive
- Low coupling = easily maintainable code
- What seems desirable when dealing with software development is to keep coupling (our interactions) among entities as low as possible

Maintainability in code

- Is an important aspect in development;
- It affects costs, code customization, bug fixing, etc.
- Maintainable code = low chance of bugs and smaller effort in making changes

Introduction

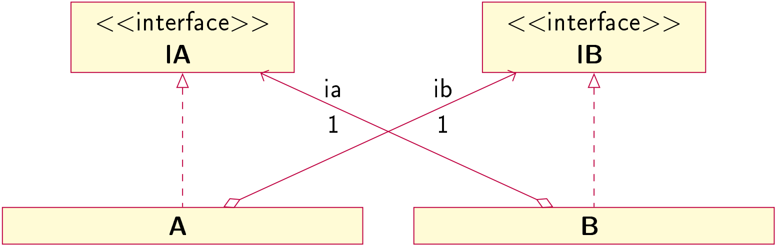
Polymorphism for taming coupling in programs

- 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 specific and controlled behavior.

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`.

Introduction



```
class Driver {
  private Vehicle vehicle;
  void Drive() {
    public this.vehicle.Move();
  }
}
interface Vehicle {
  void Move();
}
class Car : Vehicle {
  private Engine engine;
  public void Move() {
...
  }
}
```

- 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**.

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.

```
interface Vehicle {
  void Move();
}
class Car : Vehicle {
...
}
class Bike : Vehicle {
...
}
```

- 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 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".

Visiting Option's

Visiting Option's

The Option data structure

- Is used when an actual value might not exist for a named value or variable;
- An option has an underlying type and can hold a value of that type, or it might not have a value.

Visiting Option's

Example 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;

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

Example of usage

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

```
Option<int> another_number = new None<int>();
```

Visiting Option's

Some<T> and None<T>

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

```
class Some<T> : Option<T> {
...
}
```

```
class None<T> : Option<T> {
...
}
```

- `Some<T>` is a container of data, of type `T`, which is ready to get consumed; and
- `None<T>` is a container of data, of type `T`, which is not ready to get consumed yet.

Visiting Option's

- Is an interface that represents both the absence and presence of data of type T

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

Visiting an Option<T>

- As option represents a generic container for any type of objects, we need a mechanism that allows us to manipulate its content regardless its concrete data type;
- We add a method to our interface called `Visit` that accepts as inputs a series of options (in the shape of lambdas) and a generic result;
- Each option will be selected by exactly one of the possible concrete types;
- We decided a propri that the first argument is meant for the class `None<T>` while the second one for the `Some<T>`

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

Visiting Option's

Visiting a None<T>

- When visiting an object of type `None<T>` we first select the input reserved for it then we return the result of its call;

```
class None<T> : Option<T> {
  public U Visit<U>(Func<U> onNone,Func<T, U> onSome) {
    return onNone();
  }
}
```

Visiting Option's

Visiting a Some<T>

- When instantiating a Some<T> a data of type T is passed and stored inside a field value;
- When visiting an object of type Some<T> we first select the input reserved for it then we return the result of its call with value given as input;
- We pass value to the lambda, since it might be transformed/consumed by it;

```
class Some<T> : Option<T> {
  private T value;
  public Some(T value) {
    this.value = value;
  }
  public U Visit<U>(Func<U> onNone,Func<T, U> onSome) {
    return onSome(value);
  }
}
```

Visiting Option's

- The next line shows how to use our option to capture numbers and define operations over it;
- More precisely we define a Some containing the number 5 with the following operations:
  - The first lambda runs an exception, since we are trying to read a data that is not ready (None represents a `null` object);
  - The second lambda gets as input the `value` stored into Some and increments it by 1.

```
Option<int> number = new Some<int>(5);
int inc_number = number.Visit(() => "Throw␣exception..",x => (x + 1));
```

Visiting Option's

Testing out our Option<T>

- The next line shows an example with a `None` object;
- Visiting such object will indeed cause an exception;
- As we see we managed to define operations on the fly over polimorphic data types in a controlled way;
- This design will work properly (regadless the data type captured by `T`) as long as there are always options to choose.

```
Option<int> number = new None<int>();
int inc_number = number.Visit(() => "Throw␣exception..",x => (x + 1));
```

Visiting Option's

More sample

- Can be found on GIT under the folder: Design Patterns Samples C.

The visitor design pattern

The visitor design pattern

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 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.

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 $f_i^1, .., f_i^{m_i}$

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

- We now add to I a method `Visit` that returns an result of type `U`;
- `Visit`, which is method common to all classes implementing I, picks the right option based on its concrete shape;
- And since we do not know the visit result it returns a result of type generic sU

The visitor design pattern

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

- The `Visit` method accepts as input arguments as many as the possible concrete classes;
- Every argument is a function that depends on the fields of the concrete instance and produces a result of type `U`.

```
interface I<FieldsC1, FieldsC2, ..., FieldsCN>
  {
    U Visit<U>(Func<FieldsC1, U> onC1,
                 ...,
                 Func<FieldsCN, U> onCN);
  }
```

Introduction

The INFDEV team

Introduction

Our first design pattern

Visiting Option's

The visitor design pattern

Course structure

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.

```
class C1<FieldsC1, FieldsC2, ..., FieldsCN>
      : I<FieldsC1, FieldsC2, ..., FieldsCN>
  {
    Input_1 value;
    U Visit<U>(Func<FieldsC1, U> onC1,
                   ...,
                   Func<FieldsCN, U> onCN){
      onC1(this.value);
    }
  }
```

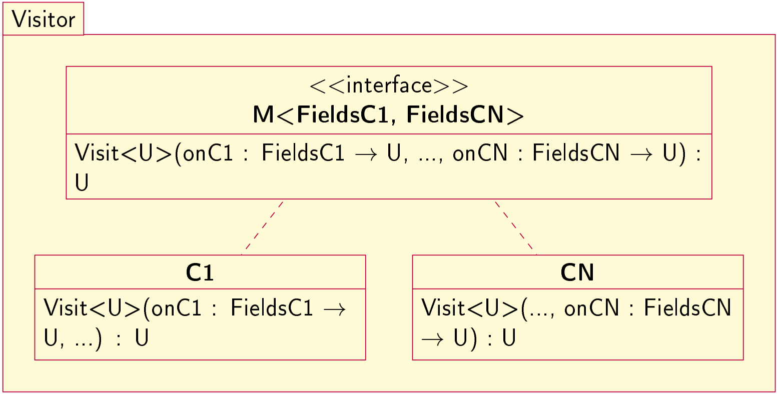
How do we use it (lambda version)? (Step 4)

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

```
I<FieldsC_1, FieldsC_2, ..., FieldsC_N> i;
...
m.Visit(
 i_1 => b_1,
 ...,
 i_N => b_n);
```

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

The visitor design pattern



The visitor design pattern

Final considerations

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

Course structure

Course structure

Final considerations

- 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.

Course structure

Conclusions

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

Course structure

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 for this course;
- We will study a series of basic design patterns, used in many applications.

This is it!

The best of luck, and thanks for the attention!