INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Frédéric Boisguérin
fboisgue@insa-toulouse.fr

Object-Oriented Programming

Outline

1. Introduction
2. Pragmatic programming
3. User interaction
4. Concurrent programming
5. Network programming

Introduction

“Object-Orientation is a dimension orthogonal to the imperative/logic/functional dimension.”

Martin Odersky

OO is not only

A bag of solutions and technology

...But it is

A way to **understand**, **describe** and **communicate**

about **a domain**

and a **concrete implementation** of that domain.

What does *Object* mean?

Pokemon
Name: Pikachu
Type: Electric
Health: 70
attack()
dodge()
evolve()

State

Behavior

Imperative style

```java
public class ImperativeCodeStyle {
    public static void main(String[] args) {
        double height = 10;
        double width = 15;
        double surface = height * width;
        System.out.println("Surface of the sheet: " + surface);
    }
}
```

⇒ Lack of semantic about what is a sheet of paper.
The height and the width are properties of a same *object*, but it does not appears clearly...

OOP style

```java
public class OopCodeStyle {
    public static void main(String[] args) {
        SheetOfPaper sheetOfPaper = new SheetOfPaper(10, 15);
        System.out.println("Surface of the sheet: " + sheetOfPaper.getSurface());
    }
    public static class SheetOfPaper {
        private final double height;
        private final double width;
        public SheetOfPaper(double height, double width) {
            this.height = height;
            this.width = width;
        }
        private double getSurface() {
            return height * width;
        }
    }
}
```

⇒ Both data and process are *encapsulated* in the *SheetOfPaper* class

The joy of OO programming

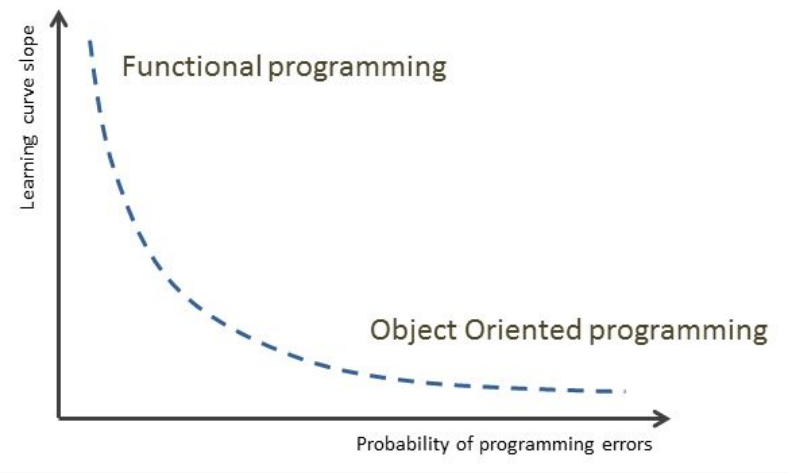
OO ⇒ no boring mathematical laws, no esoteric terms (like in OCaml or Prolog)

Everyone can study and learn an object oriented programming language

As human beings living in the *lucky* part of the world, we understand what a Car type means and why it owns attributes of type Engine, Wheel and Body.

Simplicity often leads to tradeoffs...

Simpler ⇒ less constrained ⇒ less formality



The more a programming language is easy to learn,
the easier it is to make mistakes using it

Different kind of “objects”

Values

- Model unchanging quantities and measurements
- Equal to others by their state, not identity
- No “setters”

Objects

- Have an identity
- Might change state over time
- Model *computational processes*

No difference in Java
(but in Kotlin, for example)

The core of OO

Encapsulation, abstraction, inheritance, polymorphism

Encapsulation and information hiding

```
public class Foo {
    private double value;
    public Foo(double value) {
        this.value = value;
    }
    public double asDouble() {
        return value;
    }
    public void setValue(double value){
        this.value = value;
    }
}
```

```
public class Banana {
    final Foo start;
    final Foo end;
    public Banana(Foo start, Foo end) {
        this.start = start;
        this.end = end;
    }
    public double size() {
        return end.asDouble() - start.asDouble();
    }
}
```

Encapsulation and information hiding

Encapsulation

Ensures that the **behavior of an object can only be affected through its API**.

It lets us control how much a change to one object will impact other parts of the system by ensuring that there are no unexpected dependencies between unrelated components.

Information hiding

Conceals how an object implements its functionality behind the abstraction of its API.

It lets us work with higher abstractions by ignoring lower-level details that are unrelated to the task at hand.

Tell, Don’t Ask...

```
((EditSaveCustomizer) master).getModelisable()
  .getDockablePanel()
    .getCustomizer()
      .getSaveItem().setEnabled(false);
```

Tell, Don’t Ask...

```
((EditSaveCustomizer) master).getModelisable()
  .getDockablePanel()
    .getCustomizer()
      .getSaveItem().setEnabled(false);
```

```
master.disableSavingOfCustomizations();
```

...But sometimes ask

```
public class Train {
   private final List<Carriage> carriages;
   private int percentReservedBarrier = 70;
   // ...
   public void reserveSeats(ReservationRequest request) {
      for (Carriage carriage : carriages) {
         if (carriage.getSeats().getPercentageReserved() < percentReservedBarrier) {
            request.reserveSeatsIn(carriage);
            return;
         }
      }
      request.cannotFindSeats();
   }
}
```

...But sometimes ask

```
public class Train {
   private final List<Carriage> carriages;
   private int percentReservedBarrier = 70;
   // ...
   public void reserveSeats(ReservationRequest request) {
      for (Carriage carriage : carriages) {
         if (carriage.hasSeatsAvailableWithin(percentReservedBarrier)) {
            request.reserveSeatsIn(carriage);
            return;
         }
      }
      request.cannotFindSeats();
   }
}
```

Encapsulation and data hiding lead us to Abstraction

Why ?

- Client/user perspective : interest for what a program does, not how
- Do not give the details ⇒ well structured code

How ?

- Naming
- Separation of concerns

Inheritance (concrete class)

```java
public class InsaStudent {
    private final String name;
    public InsaStudent(String name) {
        this.name = name;
    }
}
```

<< extends >>

<< extends >>

```java
public class DgeiStudent extends InsaStudent {
    public DgeiStudent(String name) {
        super(name);
    }
    public void learnToWriteCleanCode(){
        System.out.println("I'm listening to my awesome teacher");
    }
}
```

```java
public class GcStudent extends InsaStudent {
    public GcStudent(String name) {
        super(name);
    }
    public void goToBarInsa(){
        System.out.println("Baaazoom bazoom bazoom");
    }
}
```

Inheritance (abstract class)

Sometimes the parent class is not very concrete...

```
public abstract class Diploma {
    private final LocalDate deliveryDate;
    public Diploma(LocalDate deliveryDate) {
        this.deliveryDate = deliveryDate;
    }
}
```

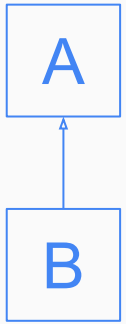
<< extends >>

<< extends >>

```
public class TOEIC extends Diploma {
    private final LocalDate expirationDate;
    public TOEIC(LocalDate deliveryDate,
                 LocalDate expirationDate) {
        super(deliveryDate);
        this.expirationDate = expirationDate;
    }
}
```

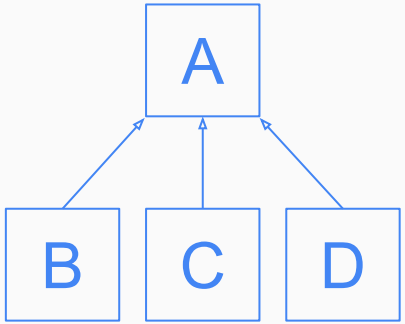
```
public class EngineeringDiploma extends Diploma {
    private final String schoolName;
    public EngineeringDiploma(LocalDate deliveryDate,
                              String schoolName) {
        super(deliveryDate);
        this.schoolName = schoolName;
    }
}
```

Different kinds of inheritance

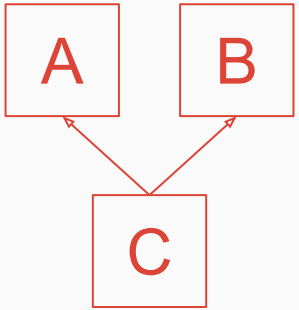


Single

Multi level



Hierarchical



Multiple

⇒ not allowed in Java

Interfaces

Tent

Camping car
???

Car

Interfaces

```java
public interface Habitable {
    boolean canFit(int inhabitants);
}
```

```java
public interface Movable {
    void moveForward(int distanceInKilometers);
}
```

```java
public class CampingCar implements Movable, Habitable {
    private final int capacity;
    private int kilometersCounter = 0;
    public CampingCar(int capacity) {
        this.capacity = capacity;
    }
    @Override
    public boolean canFit(int inhabitants) {
        return inhabitants < capacity;
    }
    @Override
    public void moveForward(int distanceInKilometers) {
        kilometersCounter += distanceInKilometers;
    }
}
```

Java syntax

- Keywords
  - Visibility : public / protected / private
  - void
  - null
  - Modifiers : static, final
- Flow control (if, for, while, do…)
- Boolean expressions (||, &&, ==, equals()...)
- Variable types : primitives (int, double, byte) *vs.* references

Question: `[new String("hello")] == [new String("hello")]` *?*

Model data

- Native arrays (double[], Object[]...)
- API Collections
  - List : ArrayList, LinkedList, Stack…
  - Map : HashMap*, TreeMap…
  - Set : HashSet*, TreeSet…

* hashCode() and equals()

Pragmatic programming

Data model, TDD, Refactoring, Design Patterns

Why?

More than 40 years of software :

- Still more bugs ⇒ lack of tests
- Not understandable code ⇒ lack of refactoring

Let’s make code a better place to play !

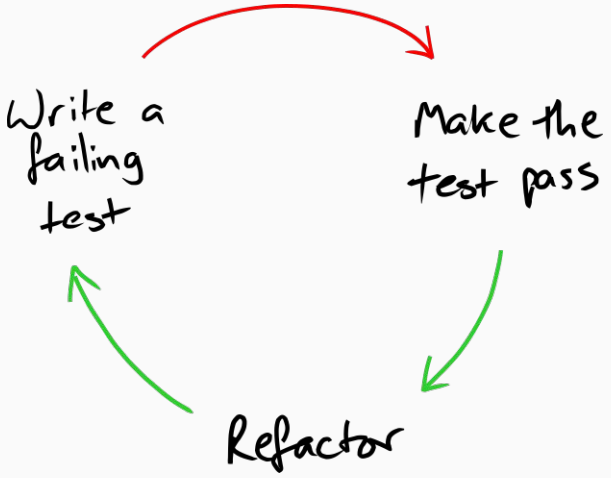
Définition de la folie :
C'est de refaire toujours la même chose, et d'attendre des résultats différents.
- Albert Einstein -

Version control

Test Driven Development

1. Write a failing test
2. Write the simplest code to pass the test
3. Refactor your code: remove duplication, rename variables, extract methods...

JUnit

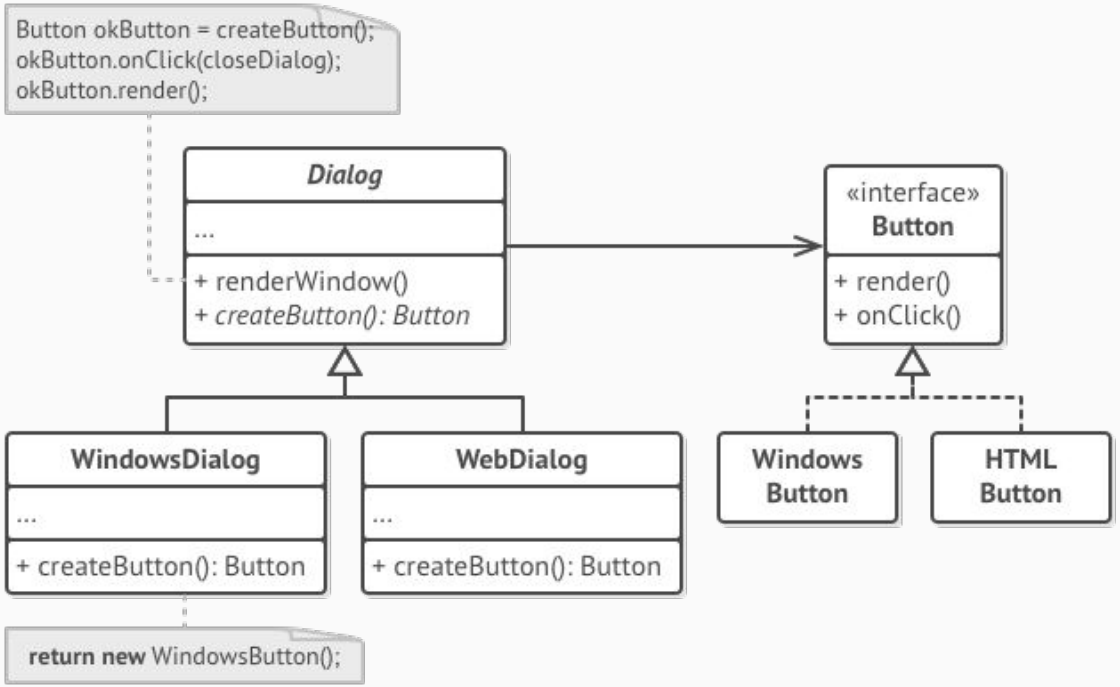


Design patterns

- Creational (e.g. Factory)
- Structural (e.g. Adapter)
- Behavioral (e.g. Chain of Responsibility)

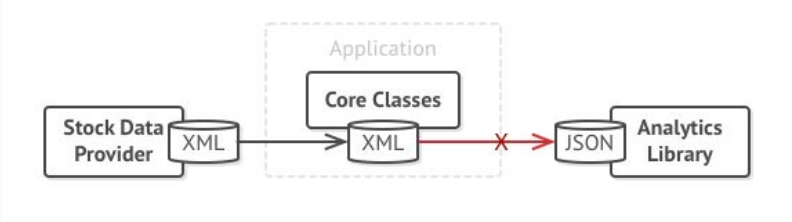
Many other examples on : https://refactoring.guru/design-patterns

Design pattern “Factory method”

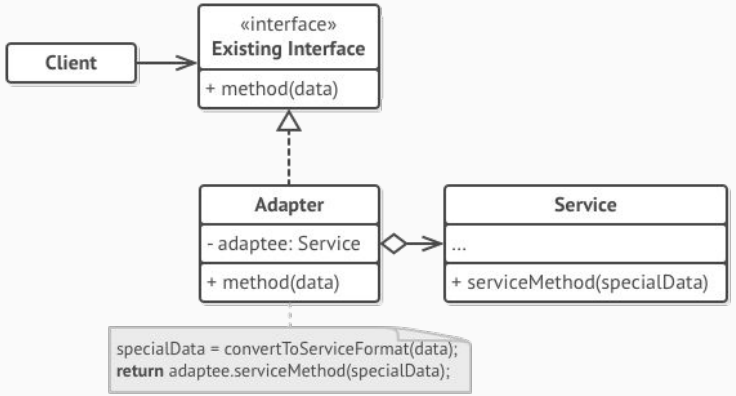


Design pattern “Adapter”

Problem



Solution



Design pattern “Chain of responsibility”