



DESIGN PATTERN



VERS UN CODE IMMuable

QU'EST-CE QU'UN DESIGN PATTERN ?

- Une solution simple à un problème récurrent.
- Conceptualisé dans: "*A Pattern Language: Towns, Buildings, Construction*", Christopher Alexander.
- Repris dans: "*Design patterns: Elements of Reusable Object-Oriented Software*", GoF.

A Pattern Language

Towns · Buildings · Construction



Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

LE GANG OF
FOUR (GOF)

Erich Gamma

Richard Helm

Ralph Johnson

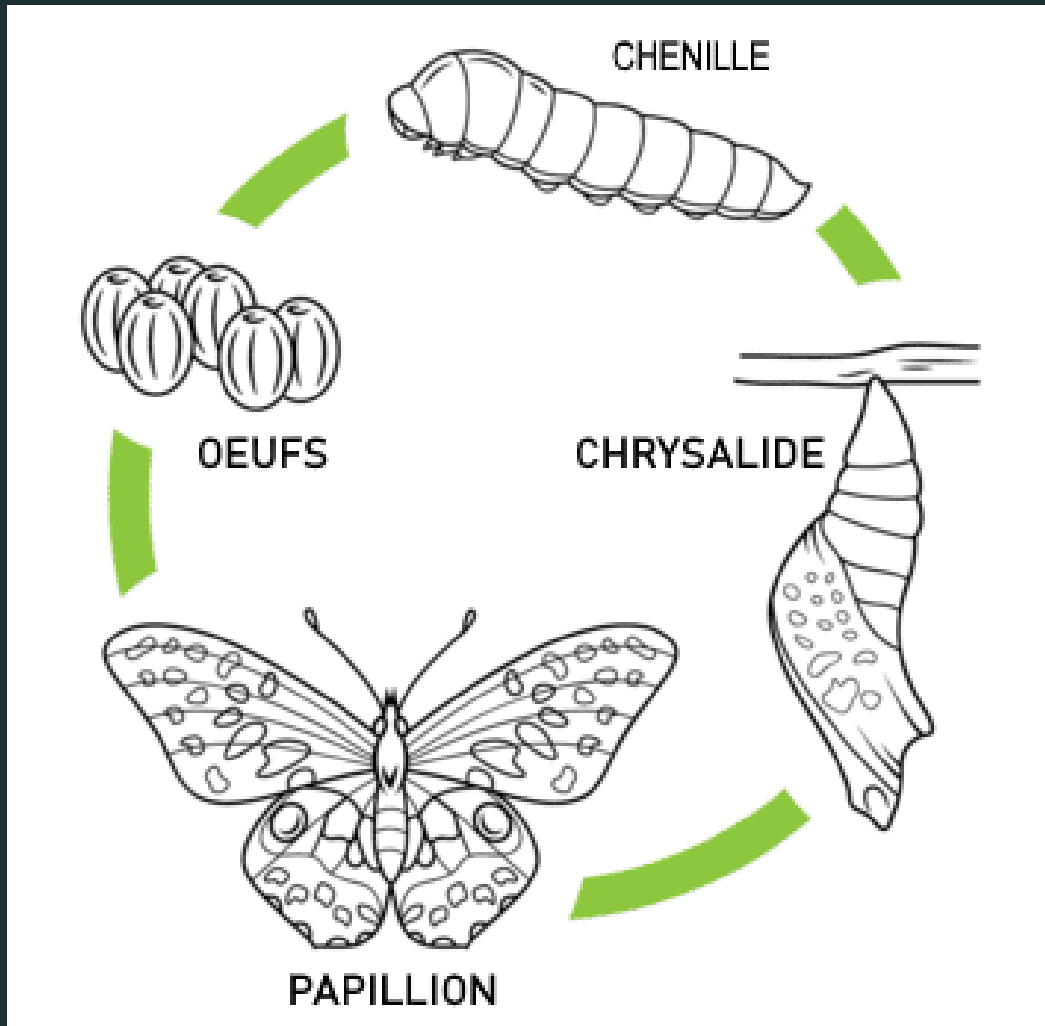
John Vlissides

DESIGN PATTERNS

- 23 patterns divisés en 3 familles.
- Créateurs: Instanciation des objets.
- Structuraux: Relations entre les classes dans une structure large.
- Comportementaux:
Communication entre les objets.

IMMUABLE VS MUABLE





MUABLE

- Quelque chose qui peut muer.
- Quelque chose qui est sujet au changement.

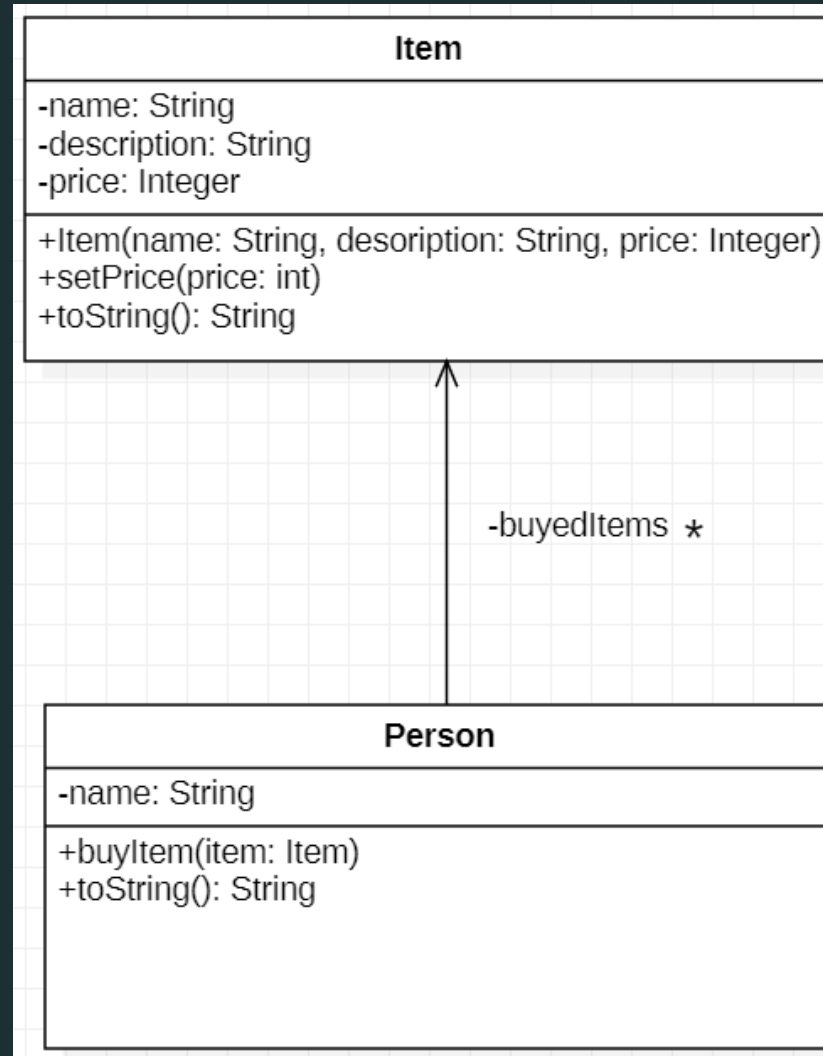
IMMUABLE

- Quelque chose de non muable.
- Quelque chose qui ne peut muer.
- Quelque chose qui n'est pas sujet au changement.

PROBLEMATIQUE: BOUTIQUE EN LIGNE

- Oncle Bob a une boutique: www.achete-si-tes-un-pigeon.com.
- Il veut que les clients puissent:
- *Accéder à la boutique sur mobile.*
- Acheter des articles depuis l'application.

DIAGRAMMES DE CLASSES



IMPLÉMENTATION

```
public class Item {
    private String name;
    private String description;
    private Integer price;
    public Item(String name, String description, Integer price) {
        this.name = name;
        this.description = description;
        this.price = price;
    }
    public void setPrice(int price){
        this.price = price;
    }
    @Override
    public String toString(){
        return name + "(" + description + ") " + price.toString() + "€";
    }
}
```

```
public class Person {
    private String name;
    ArrayList<Item> boughtItems;

    public Person(String name) {
        this.name = name;
        boughtItems = new ArrayList<>();
    }

    public void buyItem(Item item) {
        boughtItems.add(item);
    }

    @Override
    public String toString(){
        String message = name;
        if (boughtItems.isEmpty()){
            message += " n'a rien acheté";
        }
        else{
            message += " a acheté:\n";
            for (Item item: boughtItems) {
                message += "   -" + item.toString() + "\n";
            }
        }
        return message;
    }
}
```

```

public class Main {
    public static void main(String[] args) {
        Person mathias = new Person("Mathias");
        Person enzo = new Person("Enzo");
        Item tv = new Item("Télévion", "écran qui peut servir à regarder des chaînes", 200);
        Item watch = new Item("Montre", "bien positionnel qui affiche l'heure", 2000);

        mathias.buyItem(watch);
        mathias.buyItem(tv);

        enzo.buyItem(tv);

        System.out.println(mathias);
        System.out.println(enzo);
    }
}

```

Résultat

Mathias a acheté:

- Montre (bien positionnel qui affiche l'heure) 2000€
- Télévion (écran qui peut servir à regarder des chaînes) 200€

Enzo a acheté:

- Télévion (écran qui peut servir à regarder des chaînes) 200€

Que se passe-t-il si le prix d'un article change ?

```

public class Main {
    public static void main(String[] args) {
        Person mathias = new Person("Mathias");
        Person enzo = new Person("Enzo");
        Item tv = new Item("Télévion", "écran qui peut servir à regarder des chaînes", 200);
        Item watch = new Item("Montre", "bien positionnel qui affiche l'heure", 2000);

        mathias.buyItem(watch);
        mathias.buyItem(tv);

        tv.setPrice(150);

        enzo.buyItem(tv);

        System.out.println(mathias);
        System.out.println(enzo);
    }
}

```

Mathias a acheté:

- Montre (bien positionnel qui affiche l'heure) 2000€
- Télévion (écran qui peut servir à regarder des chaînes) 150€

Enzo a acheté:

- Télévion (écran qui peut servir à regarder des chaînes) 150€

LIMITES DE LA SOLUTION

- Ne permet pas de modifier le prix d'un article sans affecter l'objet instancié de la classe *Item*



LE PATTERN IMMuable

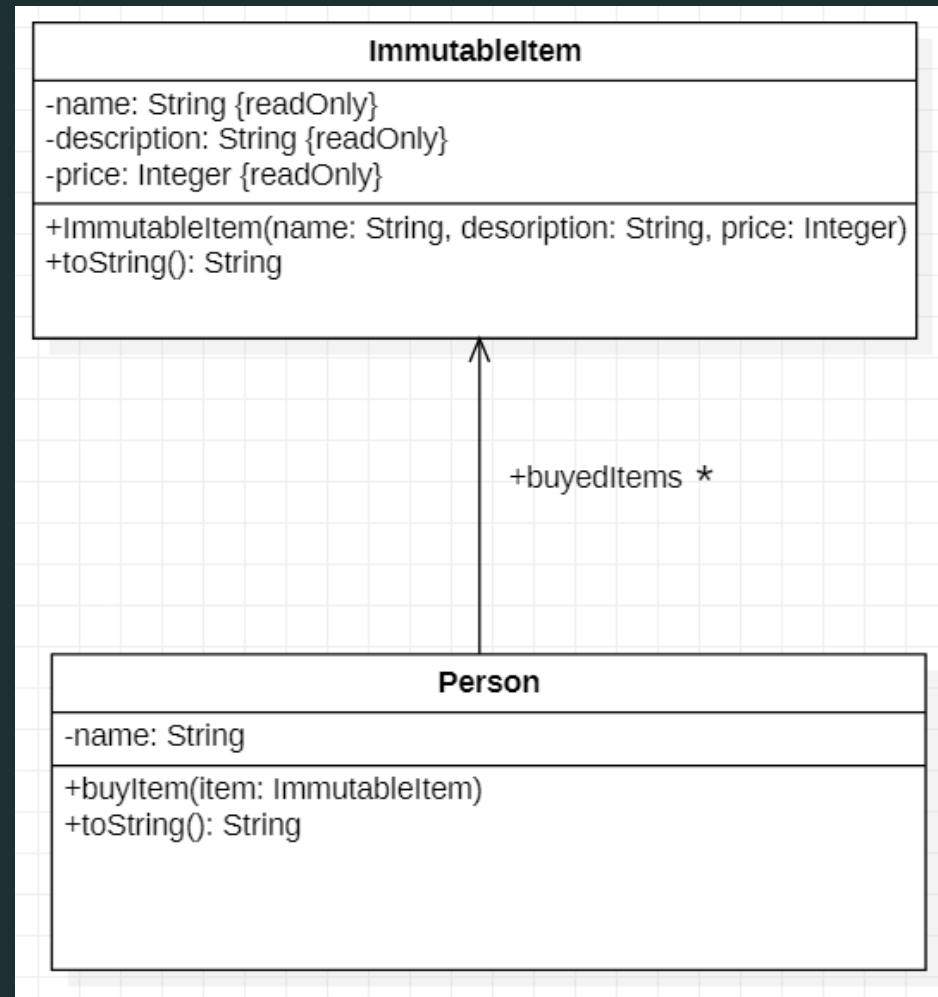
- Classe immuable
- Protection

S.O.L.I.D COMME UN ROC !

- S : Single Responsibility Principle
- O : Open Closed Principle
- L : Liskov Substitution Principle
- I : Interface Segregation Principle
- D : Dependency Inversion Principle

- Faible couplage
- Indirection
- Protection des variations

DIAGRAMMES DE CLASSES



IMPLÉMENTATION

```
public final class ImmutableItem {
    private final String name;
    private final String description;
    private final Integer price;

    public ImmutableItem(String name, String description, Integer price) {
        this.name = name;
        this.description = description;
        this.price = price;
    }

    // aucun setter

    @Override
    public String toString(){
        return name + "(" + description + ")" + price.toString() + "€";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person mathias = new Person("Mathias");
        Person enzo = new Person("Enzo");
        ImmutableItem tv = new ImmutableItem("Télévion", "écran qui peut servir à regarder des chaînes", 200);
        ImmutableItem watch = new ImmutableItem("Montre", "bien positionnel qui affiche l'heure", 2000);

        mathias.buyItem(watch);
        mathias.buyItem(tv);

        tv = new ImmutableItem("Télévion", "écran qui peut servir à regarder des chaînes", 150);

        enzo.buyItem(tv);

        System.out.println(mathias);
        System.out.println(enzo);
    }
}
```

Mathias a acheté:


- Montre (bien positionnel qui affiche l'heure) 2000€
- Télévion (écran qui peut servir à regarder des chaînes) 200€

Enzo a acheté:

- Télévion (écran qui peut servir à regarder des chaînes) 150€

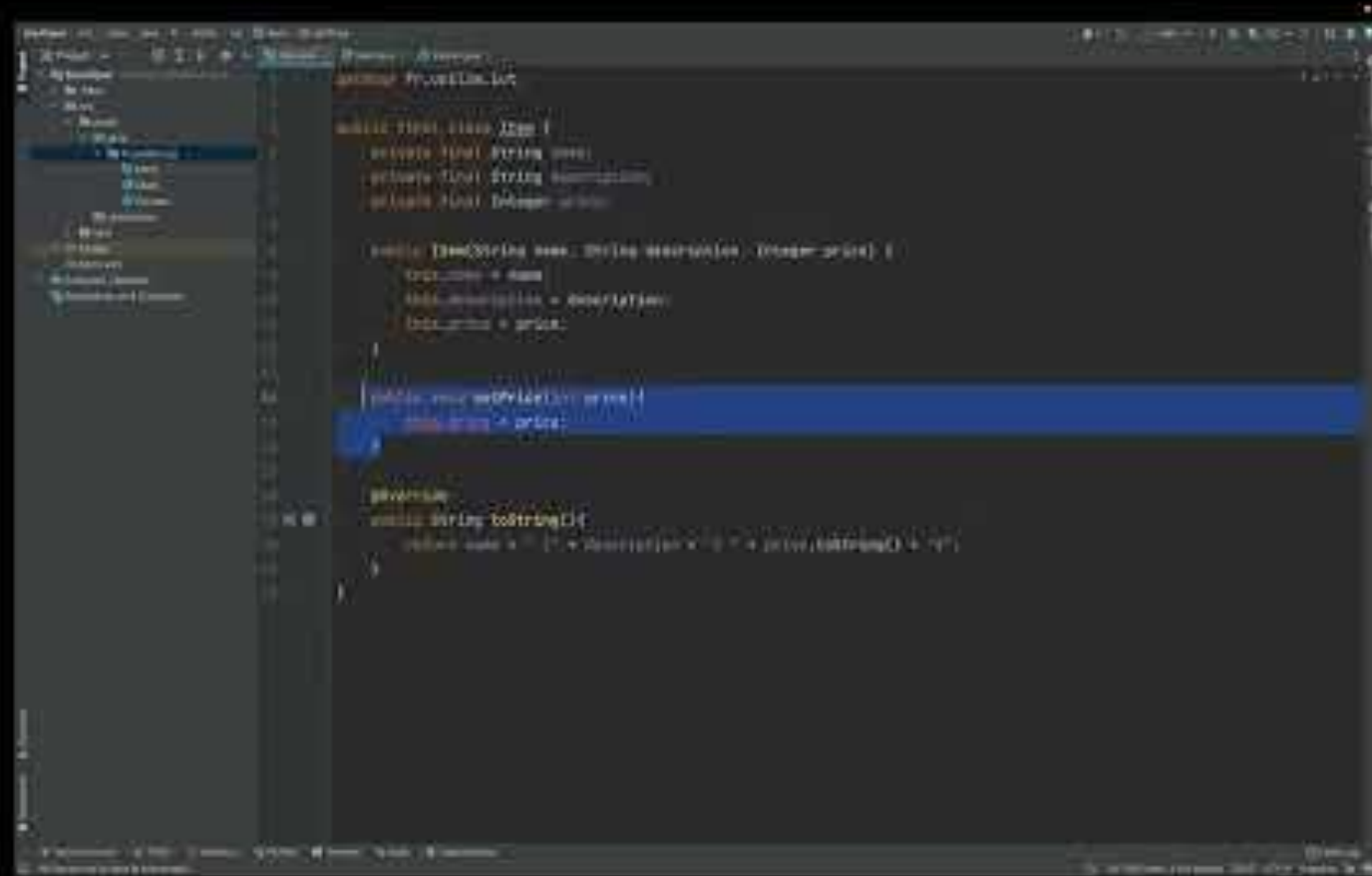
RECORD (JDK 14)

```
public final class ImmutableItem {  
    2 usages  
    private final String name;  
    2 usages  
    private final String description;  
    2 usages  
    private final Integer price;  
  
    8 usages  
    public ImmutableItem(String name, String description, Integer price) {  
        this.name = name;  
        this.description = description;  
        this.price = price;  
    }  
  
    @Override  
    public String toString(){  
        return name + " (" + description + ") " + price.toString() + "€";  
    }  
}
```



```
public record ImmutableItem(String name, String description, Integer price) {  
    @Override  
    public String toString(){  
        return name + " (" + description + ") " + price.toString() + "€";  
    }  
}
```

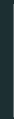
LIVE CODING



LIMITES DE L'IMMUABILITÉ

- Peut être coûteuse en ressources et en mémoire

PATTERNS
SIMILAIRES



Read-Only

MERCI !

- Test tes connaissances:
shorturl.at/npqNU



SOURCES

- <https://www.digitalocean.com/community/tutorials/how-to-create-immutable-class-in-java>
- <https://www.geeksforgeeks.org/create-immutable-class-java/>
- <https://docs.oracle.com/en/java/javase/14/language/records.html>
- <https://dev-vibe.medium.com/object-immutability-pattern-the-only-constant-in-the-ever-changing-world-of-programming-c5fb6d224260>
- <https://lkumarjain.blogspot.com/2016/02/immutable-design-pattern.html>
- <https://www.youtube.com/watch?v=T43EdFSqCmo>