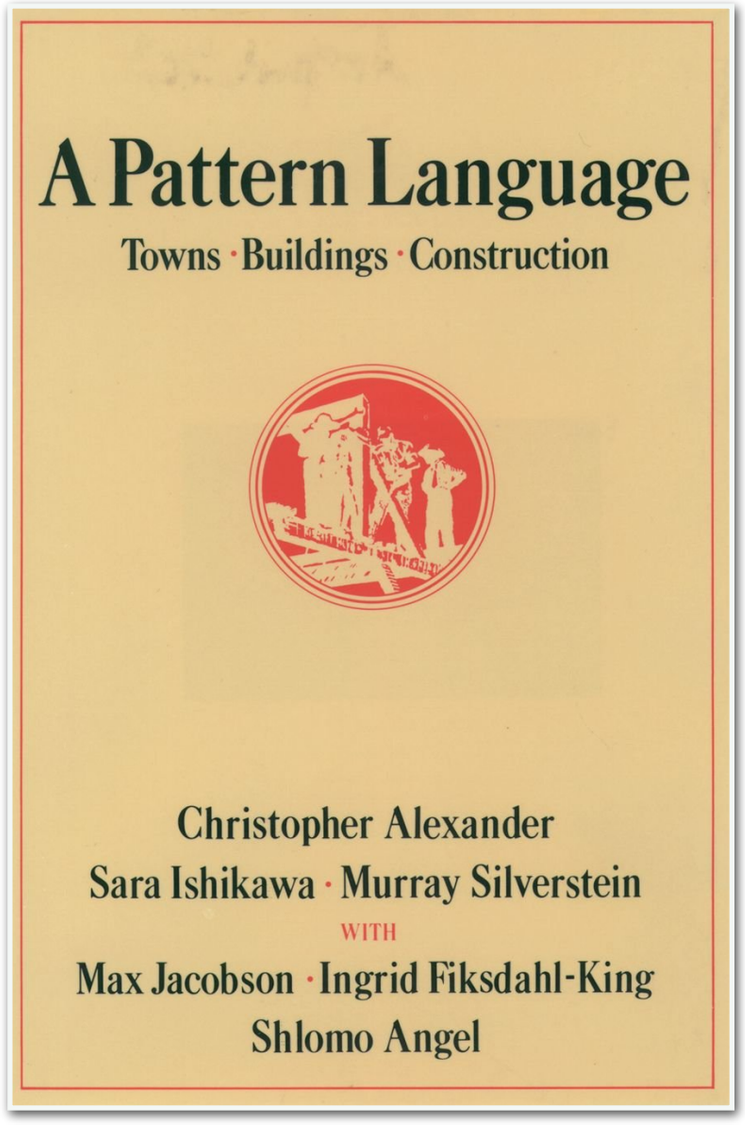
Vers un code Immutable

Par : BERTUCAT Hector, FIAU Maël, ROUSSELET Guillaume, Sarribouette Goran

Introduction

Qu’est-ce qu’un design Pattern ?

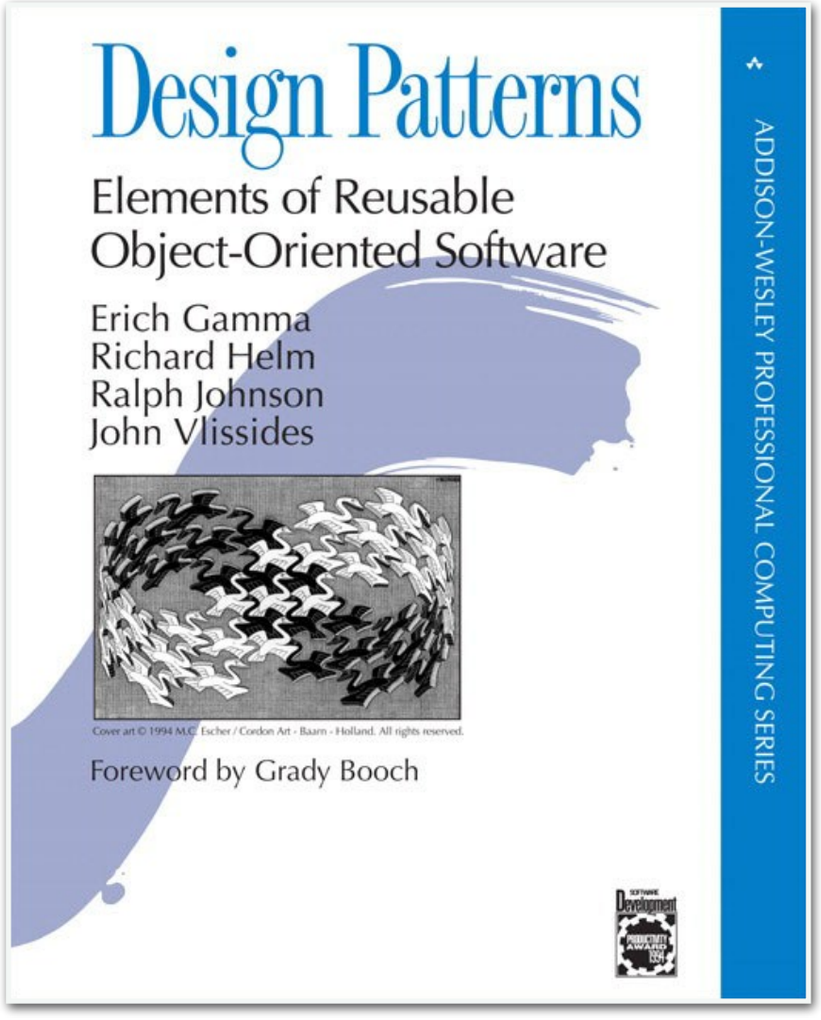
Qu’est-ce qu’un design Pattern ?



Qu’est-ce qu’un design Pattern ?

- Accélérer le processus de développement
- Anticiper les problèmes
- Améliorer la lisibilité du code

Qu’est-ce qu’un design Pattern ?



Qu’est-ce qu’un design Pattern ? - Principe SOLID

- **S**ingle Responsability Principle (*SRP*)
- **O**pen Close Principle (*OCP*)
- **L**iskov Substitution Principle (*LSP*)
- **I**nterface Segregation Principle (*ISP*)
- **D**ependency Inversion Principle (*DIP*)

Comment mettre en oeuvre l’immutabilité ?

Plan

1. Qu’est qu’un code Immutable ?
2. L’intérêt de l’immutabilité
3. Comment mettre en oeuvre l’immutabilité

1 - Qu’est-ce qu’un code Immutable ?

Qu’est-ce qu’un code Immutable ?

Définition

Un objet immuable, est un objet dont l'état ne peut pas être modifié après sa création.

Qu’est-ce qui définit une classe immuable ?

- Des attributs immuables
- Une classe « final »
- Pas de setters
- Ni de méthodes qui auraient la même fonction

Comment faire si un objet contient une variable qui est une référence à une classe mutable ?

- Pas de méthodes modifiant cet objet
- Pas de partage de référence à cette objet (getters ou setters)
  Sinon : copie défensive

2 - L’intérêt de l’immutabilité

Avantages

Fiabilité

• Les classes immuables sont par nature thread-safe, elles sont conseillées en environnement multi-thread

•Elles peuvent être mises en cache côté client sans risque de désynchronisation

•Elles peuvent être utilisées sans risque comme clé dans des HashMap ou dans des HashSets, car leur HashCode est constant

Avantages

Lisibilité

• Quand ces objets sont utilisés comme variables d’une classe, leur initialisation n’a pas à être fait à partir d’une copie défensive

•Une classe immutable, est plus simple et plus lisible. Pas de setter etc..

•L’invariant de classe n’a besoin d’être validé qu’à la création de l’objet

•Il n’est pas nécessaire de leur créer un constructeur par copie ou d’implémenter l’interface Clonable (spécifique Java)

Avantages

Optimisation

• Il est possible pour le hashcode de ces classes, d’utiliser la technique de la lazy initialisation et de le garder en cache

Inconvénient

Coût

• Si l’on souhaite modifier un objet, il faut le recréer ce qui peut être plus couteux qu’une simple modification

3 - Rendre un code immuable

Comment mettre en oeuvre l'immutabilité ?

Classe Etudiant

```java
package pattern.immuable;

import java.util.List;

public class Etudiant {
        private String nom;
    private Adresse address;
    private List<String> matieres;

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public Adresse getAddress() {
        return address;
    }

    public void setAddress(Adresse address) {
        this.address = address;
    }

    public List<String> getMatieres() {
        return matieres;
    }

    public void setMatieres(List<String> matieres) {
        this.matieres = matieres;
    }
}
```

Classe Adresse

```java
package pattern.immuable;

public class Adresse {
        private String pays;
    private String ville;

    public Adresse(String pays, String ville) {
        this.pays = pays;
        this.ville = ville;
    }

    public String getPays() {
        return pays;
    }

    public void setPays(String pays) {
        this.pays = pays;
    }

    public String getVille() {
        return ville;
    }

    public void setVille(String ville) {
        this.ville = ville;
    }
}
```

Comment mettre en oeuvre l'immutabilité ?

La valeur des attributs peut être changé
Empêchons le changement de valeur en enlevant les setters

```java
package pattern.immuable;

import java.util.List;

public class Etudiant {
    private String nom;
    private Adresse address;
    private List<String> matieres;

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public Adresse getAddress() {
        return address;
    }

    public void setAddress(Adresse address) {
        this.address = address;
    }

    public List<String> getMatieres() {
        return matieres;
    }

    public void setMatieres(List<String> matieres) {
        this.matieres = matieres;
    }
}
```

Comment mettre en oeuvre l’immutabilité ?

Les setters sont enlevés, mais comment changer la valeur des attributs ?

```java
package pattern.immuable;

import java.util.List;

public class Etudiant {
    private String nom;
    private Adresse address;
    private List<String> matieres;

    public String getNom() {
        return nom;
    }

    public Adresse getAddress() {
        return address;
    }

    public List<String> getMatieres() {
        return matieres;
    }
}
```

Comment mettre en oeuvre l’immutabilité ?

Le constructeur est maintenant créé

```java
package pattern.immuable;

import java.util.List;

public class Etudiant {
    private String nom;
    private Adresse address;
    private List<String> matieres;

public Etudiant(String nom, Adresse adresse, List<String> matieres ) {
     this.nom = nom;
     this.adresse = adresse
     this.matieres = matieres;
    }

    public String getNom() {
        return nom;
    }

    public Adresse getAddress() {
        return address;
    }

    public List<String> getMatieres() {
        return matieres;
    }
}
```

Est-ce terminé ?…
Pas réellement…

Notre immutabilité peut être contournée par l’héritage.

Comment mettre en oeuvre l'immutabilité ?

Un exemple.

```java
public class EtudiantIut extends Etudiant{
    private String nom;

    public EtudiantIut(String nom, Adresse adresse, List<String>
 matieres) {
        super(nom, adresse, matieres);
        this.nom = nom;
    }
    public void setNom(String nouveauNom) {
     this.nom = nouveauNom;
    }
    @Override
    public String getNom() {
        return this.nom;
    }
}
```

```java
public static void main(String[] args) {

        EtudiantIut guillaume = new EtudiantIut("Guillaume
Rousselet", new Adresse("France", "Limoges"), new
 ArrayList<String>());

        Etudiant etudiant = (Etudiant) guillaume;

        System.out.println(etudiant.getNom());

        guillaume.setNom("Un autre nom");

        System.out.println(etudiant.getNom());

}
```

```
> Guillaume Rousselet
> Un autre nom
```

Empêcher le contournement via l’héritage

```
public class Etudiant { . . . }
```

Comment mettre en oeuvre l’immutabilité ?

Empêcher le contournement via l’héritage

```
public final class Etudiant { . . . }
```

Comment mettre en oeuvre l'immutabilité ?

Un exemple.

```java
public class EtudiantIut extends Etudiant{
    private String nom;

    public EtudiantIut(String nom, Adresse adresse, List<String>
 matieres) {
        super(nom, adresse, matieres);
        this.nom = nom;
    }
    public void setNom(String nouveauNom) {
     this.nom = nouveauNom;
    }
    @Override
    public String getNom() {
        return this.nom;
    }
}
```

```java
public static void main(String[] args) {

        EtudiantIut guillaume = new EtudiantIut("Guillaume
Rousselet", new Adresse("France", "Limoges"), new
 ArrayList<String>());

        Etudiant etudiant = (Etudiant) guillaume;

        System.out.println(etudiant.getNom());

        guillaume.setNom("Un autre nom");

        System.out.println(etudiant.getNom());

}
```

> Erreur

Le passage de valeur

```
public static void main(String[] args) {
        Adresse adresse = new Adresse("France", "Limoges");
        List<String> matieres = new ArrayList<String>();
        matieres.add("POO");
        matieres.add("Maths");

        Etudiant guillaume = new Etudiant("Guillaume Rousselet", adresse, matieres);

        System.out.println(guillaume.getNom());
        System.out.println(guillaume.getAdresse().getPays());
        System.out.println(guillaume.getAdresse().getVille());
        System.out.println(guillaume.getMatieres());

        System.out.println();

        adresse.setPays("Canada");
        adresse.setVille("Montréal");
        matieres.remove("Maths");

        System.out.println(guillaume.getNom());
        System.out.println(guillaume.getAdresse().getPays());
        System.out.println(guillaume.getAdresse().getVille());
        System.out.println(guillaume.getMatieres());


}
```

Comment mettre en oeuvre l’immutabilité ?

Pour corriger ce problème de passage de valeur par référence

```
public Etudiant(String nom, Adresse adresse, List<String> matieres ) {
      this.nom = nom;
      this.adresse = adresse;
      this.matieres = matieres;
}
```

Comment mettre en oeuvre l'immutabilité ?

Il est nécessaire d'initialiser les attributs
non primitifs via le constructeur en les copiant

```
public Etudiant(String nom, Adresse adresse, List<String> matieres ) {
      this.nom = nom;
      this.adresse = new Adresse(adresse.getPays(), adresse.getVille());
      this.matieres = new ArrayList<String>(matieres);
}
```

```
> Guillaume Rousselet
> France
> Limoges
> POO, Maths
```

```
> Guillaume Rousselet
> France
> Limoges
> POO, Maths
```

Comment mettre en oeuvre l'immutabilité ?

Le comportement du main est maintenant comme souhaité…
Mais il peut aussi être contourné via les getters

```
adresse.setPays("Canada");
adresse.setVille("Montréal");
matieres.remove("Maths");
```

Comment mettre en oeuvre l'immutabilité ?

En changeant le code et en utilisant les getters…

```
guillaume.getAdresse().setPays("Canada");
guillaume.getAdresse().setVille("Montréal");
guillaume.getMatieres().remove("Maths");
```

```
> Guillaume Rousselet
> France
> Limoges
> POO, Maths
```

```
> Guillaume Rousselet
> Canada
> Montréal
> POO
```

Comment mettre en oeuvre l'immutabilité ?

Pour corriger ce problème, à la manière des setters
Les getters doivent renvoyer une copie des attributs non primitifs

```
public Adresse getAddress() {
        return addresse;
    }
```

```
public List<String> getMatieres() {
    return matieres;
}
```

Comment mettre en oeuvre l'immutabilité ?

Pour corriger ce problème, à la manière des setters
Les getters doivent renvoyer une copie des attributs non primitifs

```
public Adresse getAddress() {
        return new Addresse(adresse.getPays(), adresse.getVille());
    }
```

```
public List<String> getMatieres() {
    return new ArrayList<String>(matieres);
}
```

Comment mettre en oeuvre l’immutabilité ?

La dernière modification à effectuer est de modifier l’état des attributs

```
public final class Etudiant {
    private String nom;
    private Adresse adresse;
    private List<String> matieres;

    . . .
}
```

Comment mettre en oeuvre l’immutabilité ?

En les déclarant en tant que final

```
public final class Etudiant {
    private final String nom;
    private final Adresse adresse;
    private final List<String> matieres;

    . . .
}
```

Étapes vers l'immutabilité

1. Enlever les setters
2. Ajouter chaque attribut au(x) constructeur(s)
3. Définir la classe comme *final* afin de la protéger de l'héritage
4. Initialiser chaque attribut non primitifs via le constructeur en effectuant une copie
5. Effectuer la duplication de chaque objet non primitifs retournés par les getters.
6. Définir chaque attribut de notre classe comme final.

Conclusion

- État non modifiable après la création

- Plusieurs étapes (code mutable -> code immutable)

Sitographie

1ère Partie :

https://wodric.com/classe-immutable/

https://fr.wikipedia.org/wiki/Objet_immuable

https://jobprod.com/astucedecode-10-les-objets-immutables

https://www.freecodecamp.org/news/write-safer-and-cleaner-code-by-leveraging-the-power-of-immutability-7862df04b7b6/

2ème Partie :

https://wodric.com/classe-immutable/

https://web.maths.unsw.edu.au/~lafaye/CCM/java/javaconst.htm

https://perso.telecom-paristech.fr/hudry/coursJava/avance/dupliquer.html

https://gfx.developpez.com/tutoriel/java/immuables/

3ème Partie :

https://medium.com/@mykola.shumyn/immutable-classes-in-java-76635df0356d

En avant pour le QCM !

bit.ly/immutabilite