Java : design patterns

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Design pattern en Java

Plusieurs traductions possibles

- Motifs de conception
- Patrons de conception
- Masques de conception
- Modèles de conception
- ...

Design pattern en Java

L'architecte anglais **Christopher Alexander** (A Pattern Language 1977)

Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière.

Design pattern en Java

L'architecte anglais Christopher Alexander (A Pattern Language 1977)

Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière.

L'idée a été reprise et détaillée par la suite dans le livre *Elements of Reusable Object-Oriented Software* en 1995 par le **Gang of Four** (GoF, la bande des quatre en français) Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides).

Design pattern en Java

Donc, un design pattern est

- une solution réutilisable à un problème récurrent de conception de logiciel
- un modèle de conception qui a fait ses preuves et qui peut être appliqué dans différentes situations

Design pattern en Java

Les design patterns, pourquoi ?

- **Amélioration de la lisibilité et de la maintenabilité du code** : en utilisant des patterns reconnus, le code devient plus facile à comprendre et à modifier.
- **Réduction des erreurs** : les patterns ont été conçus pour résoudre des problèmes courants, ce qui réduit le risque d'introduire des bugs.
- **Facilitation de la collaboration** : les développeurs qui connaissent les mêmes patterns peuvent travailler ensemble plus efficacement.
- **Promotion de bonnes pratiques** : les patterns encouragent l'adoption de bonnes pratiques de programmation.

Design pattern en Java

Classification de Design Patterns

- **Patterns de création** (fabrication, construction...) : description de la façon dont un (ou plusieurs) objet(s) peu(ven)t être créé(s), initialisé(s), et configuré(s)
- **Patterns structurels** : description de la manière dont doivent être liés des objets afin de rendre ces connections indépendantes des évolutions futures de l'application
- **Patterns comportementaux** : description de l'interconnexion entre objets
- **Patterns architecturaux** : interaction et structure globale des systèmes logiciels à un niveau plus élevé

Design pattern en Java

Design patterns : exemples

- **Patterns de création** : Singleton, Factory, Builder, Prototype...
- **Patterns structurels** : Adapter, Bridge, Composite, Decorator...
- **Patterns comportementaux** : Iterator, Observer, Visitor, Strategy...
- **Patterns architecturaux** : MVC, IoC (DI)

Design pattern en Java

Problématique et solution

- Une classe qu'on ne peut instancier qu'une seule fois
- L'unicité de l'instance est complètement contrôlée par la classe elle-même
- Mettre en privé le constructeur ? $\Rightarrow$ insuffisant car la classe n'est plus instanciable
- Un traitement additionnel par une méthode autre que le constructeur

Design pattern en Java

Exemple

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        System.out.println("Appelé une seule fois");
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Design pattern en Java

Pour tester

```
public class Main {

    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
    }

}
```

Design pattern en Java

Pour tester

```
public class Main {

    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
    }

}
```


Résultat

```
Appelé une seule fois
```

Design pattern en Java

Remarque

Évidemment, ce motif peut facilement être étendu pour permettre la création d'un nombre bien déterminé d'instances.

Design pattern en Java

Problématique et solution

- On veut automatiser par exemple l'instanciation d'une classe (création d'objets)
- Ceci peut nous aider par exemple à remplir une table de notre base de données avec des données aléatoires pour qu'on puisse tester notre application
- Déléguer la création d'objets à une classe Factory qui fabriquera des objets en fonction de nos paramètres

Design pattern en Java

Considérons l'interface `Forme`

```
public interface Forme {
    double calculerSurface();
}
```

Design pattern en Java

La classe `Rectangle`

```
public class Rectangle implements Forme {
    private double largeur;
    private double hauteur;

    public Rectangle(double largeur, double hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    @Override
    public double calculerSurface() {
        return largeur * hauteur;
    }
}
```

Design pattern en Java

La classe `Triangle`

```
public class Triangle implements Forme {
    private double base;
    private double hauteur;

    public Triangle(double base, double hauteur) {
        this.base = base;
        this.hauteur = hauteur;
    }

    @Override
    public double calculerSurface() {
        return (base * hauteur) / 2;
    }
}
```

Design pattern en Java

La classe `Cercle`

```
public class Cercle implements Forme {
    private double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    @Override
    public double calculerSurface() {
        return Math.PI * rayon * rayon;
    }
}
```

Design pattern en Java

Et la classe `FormeFactory`

```
public class FormeFactory {
    public static Forme createForme(String type, double... dimensions)
       {
        switch (type.toLowerCase()) {
        case "rectangle":
            return new Rectangle(dimensions[0], dimensions[1]);
        case "triangle":
            return new Triangle(dimensions[0], dimensions[1]);

        case "cercle":
            return new Cercle(dimensions[0]);

        default:
            throw new IllegalArgumentException("Forme inconnue : " +
               type);
        }
    }
}
```

Design pattern en Java

Pour tester

```
public class Main {

    public static void main(String[] args) {
        Forme rectangle = FormeFactory.createForme("rectangle", 4, 5);
        System.out.println("Surface du rectangle: " + rectangle.calculerSurface());

        Forme triangle = FormeFactory.createForme("triangle", 3, 4);
        System.out.println("Surface du triangle: " + triangle.calculerSurface());

        Forme cercle = FormeFactory.createForme("cercle", 2.5);
        System.out.println("Surface du cercle: " + cercle.calculerSurface());
    }

}
```

Design pattern en Java

Pour tester

```
public class Main {

    public static void main(String[] args) {
        Forme rectangle = FormeFactory.createForme("rectangle", 4, 5);
        System.out.println("Surface du rectangle: " + rectangle.calculerSurface());

        Forme triangle = FormeFactory.createForme("triangle", 3, 4);
        System.out.println("Surface du triangle: " + triangle.calculerSurface());

        Forme cercle = FormeFactory.createForme("cercle", 2.5);
        System.out.println("Surface du cercle: " + cercle.calculerSurface());
    }

}
```


Résultat

```
Surface du rectangle: 20.0
Surface du triangle: 6.0
Surface du cercle: 19.634954084936208
```

Design pattern en Java

Problématique et solution

Utile lorsque la création d'un objet requiert plusieurs étapes ou lorsqu'il existe de nombreuses variations possibles dans la configuration de cet objet.

Design pattern en Java

Commençons par définir une classe `Computer`

```
public class Computer {
    private String processor;
    private int ram;

    // + getters et setters

    @Override
    public String toString() {
        return "Computer [processor=" + processor + ", ram=" +
           ram + "]";
    }

}
```

Design pattern en Java

Dans `Computer`, déclarons `ComputerBuilder` comme classe interne static

```
public static class ComputerBuilder {
    private String processor;
    private int ram;

    public ComputerBuilder setProcessor(String processor) {
        this.processor = processor;
        return this;
    }

    public ComputerBuilder setRam(int ram) {
        this.ram = ram;
        return this;
    }

    public Computer build() {
        return new Computer(this);
    }
}
```

Design pattern en Java

Dans `Computer`, ajoutons un constructeur privé prenant comme paramètre un `ComputerBuilder`

```
private Computer(Builder builder) {
    this.processor = builder.processor;
    this.ram = builder.ram;
}
```

Design pattern en Java

Dans `Computer`, ajoutons un constructeur privé prenant comme paramètre un `ComputerBuilder`

```
private Computer(Builder builder) {
    this.processor = builder.processor;
    this.ram = builder.ram;
}
```

Et pour faciliter l'utilisation, ajoutons une méthode static `builder`

```
public static Builder builder() {
    return new Builder();
}
```

Design pattern en Java

Pour tester

```
Computer computer = Computer.builder()
    .setProcessor("Intel i7")
    .setRam(16)
    .build();
System.out.println(computer);
```

Design pattern en Java

Pour tester

```
Computer computer = Computer.builder()
    .setProcessor("Intel i7")
    .setRam(16)
    .build();
System.out.println(computer);
```


Résultat

```
Computer [processor=Intel i7, ram=16]
```

Design pattern en Java

Problématique et solution

- On veut utiliser une classe existante mais sa structure ne coïncide pas avec celle attendue
  - nom de méthodes
  - nombre d'attributs ou leurs types
- utiliser un adaptateur de cette classe sans avoir besoin de la modifier

Design pattern en Java

Considérons l'interface `Forme`

```
public interface Forme {
    void dessiner();
}
```

Design pattern en Java

La classe `Rectangle`

```
public class Rectangle implements Forme {

    @Override
    public void dessiner() {
        System.out.println("Dessiner un rectangle");
    }
}
```

Design pattern en Java

La classe `Rectangle`

```
public class Rectangle implements Forme {

    @Override
    public void dessiner() {
        System.out.println("Dessiner un rectangle");
    }
}
```

La classe `Cercle`

```
public class Cercle implements Forme {

    @Override
    public void dessiner() {
        System.out.println("Dessiner un cercle");
    }
}
```

Design pattern en Java

La classe `Triangle` qui n'a pas la même structure (n'implémente pas `Forme`

```
public class Triangle {

    public void drawTriangle() {
        System.out.println("Dessiner un triangle");
    }
}
```

Design pattern en Java

La classe `TriangleAdapter` **qui adapte** `Triangle` **à une** `Forme`

```
public class TriangleAdapter implements Forme {

    private Triangle triangle;

    public TriangleAdapter(Triangle triangle) {
        this.triangle = triangle;
    }

    @Override
    public void dessiner() {
        triangle.drawTriangle();
    }
}
```

Design pattern en Java

Pour tester

```
Forme rectangle = new Rectangle();
Forme cercle = new Cercle();
Forme triangleAdapter = new TriangleAdapter(new Triangle());

rectangle.dessiner();
cercle.dessiner();
triangleAdapter.dessiner();
```

Design pattern en Java

Pour tester

```
Forme rectangle = new Rectangle();
Forme cercle = new Cercle();
Forme triangleAdapter = new TriangleAdapter(new Triangle());

rectangle.dessiner();
cercle.dessiner();
triangleAdapter.dessiner();
```


Résultat

```
Dessiner un rectangle
Dessiner un cercle
Dessiner un triangle
```

Design pattern en Java

Problématique et solution

- on veut assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance
- l'objet, dont on veut suivre le changement de ses états, doit implémenter une interface qui offre trois méthodes permettant de s'abonner (s'attacher, s'inscrire...), se désabonner et notifier les abonnés
- les objets qui veulent être informés doivent implémenter une interface qui offre une méthode `update()`

Design pattern en Java

Considérons la classe `News` suivante

```
public class News {

    private String titre;
    private String categorie;
    private String details;

    public News(String titre, String categorie, String details) {
        this.titre = titre;
        this.categorie = categorie;
        this.details = details;
    }

    @Override
    public String toString() {
        return "News [titre=" + titre + ", categorie=" + categorie + ", details=" + details + "
           ]";
    }

}
```

Design pattern en Java

Commençons par l'interface `Observer` qui la methode `update` qui sera utilisée pour informer les observateurs

```
public interface Observer {
    void update(News news);
}
```

Design pattern en Java

Commençons par l'interface `Observer` qui la methode `update` qui sera utilisée pour informer les observateurs

```
public interface Observer {
    void update(News news);
}
```

Et l'interface `Subject` qui décrit les opérations qu'un `Subject` doit prendre en charge

```
public interface Subject {

    void subscribe(Observer o);

    void unsubscribe(Observer o);

    void notifyObservers();
}
```

Design pattern en Java

Créons la classe `LaProvence` **qui implémente** `Subject`

```
public class LaProvence implements Subject {

    private News news;
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void subscribe(Observer o) {
        observers.add(o);
    }

    @Override
    public void unsubscribe(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }
    public void setNews(News news) {
        this.news = news;
        notifyObservers();
    }
}
```

Design pattern en Java

La classe `Smartphone` **qui implémente** `Observer`

```
public class Smartphone implements Observer {

    private News news;

    @Override
    public void update(News news) {
       this.news = news;
        System.out.println("Nouvelle info sur smartphone : " + news);
    }

}
```

Design pattern en Java

Et `Tablette` qui implémente aussi `Observer`

```
public class Tablette implements Observer {

    private News news;

    @Override
    public void update(News news) {
        this.news = news;
        System.out.println("Nouvelle info sur tablette : " + news);
    }

}
```

Design pattern en Java

Pour tester

```
LaProvence laProvence = new LaProvence();
Observer smartObserver = new Smartphone();
Observer tabObserver = new Tablette();
laProvence.subscribe(tabObserver);
laProvence.subscribe(smartObserver);
laProvence.setNews(new News("Victoire OM", "Foot", "5-0"));
```

Design pattern en Java

Pour tester

```
LaProvence laProvence = new LaProvence();
Observer smartObserver = new Smartphone();
Observer tabObserver = new Tablette();
laProvence.subscribe(tabObserver);
laProvence.subscribe(smartObserver);
laProvence.setNews(new News("Victoire OM", "Foot", "5-0"));
```


Résultat

```
Nouvelle info sur tablette : News [titre=Victoire OM, categorie=Foot, details=5-0]
Nouvelle info sur smartphone : News [titre=Victoire OM, categorie=Foot, details=5-0]
```

Design pattern en Java

Autres détails et exemples

- http://design-patterns.fr/
- https://fr.wikipedia.org/wiki/Patron_de_conception