Règles de bases

Introduction

Ces règles fondamentales constituent le socle d'une programmation propre et maintenable. Leur application rigoureuse permet de :

- **Réduire la dette technique** : En évitant les mauvaises pratiques qui s'accumulent et compliquent la maintenance future
- **Faciliter la collaboration** : Un code bien structuré est plus facile à comprendre et à modifier par d'autres développeurs
- **Améliorer la testabilité** : Des composants à responsabilité unique avec des dépendances claires sont plus simples à tester
- **Favoriser l'évolutivité** : Un code respectant ces principes s'adapte plus facilement aux changements de spécifications
- **Diminuer les coûts de maintenance** : La clarté et la cohérence du code réduisent le temps nécessaire pour comprendre et modifier le code existant

Ces principes ne sont pas de simples recommandations stylistiques, mais des fondements éprouvés qui garantissent la pérennité et la qualité des applications. Leur application cohérente permet d'éviter les pièges courants du développement logiciel et d'assurer que votre code reste compréhensible et maintenable à long terme.

1. DRY : Don’t Repeat Yourself

Cette règle encourage à éviter la duplication de code.
La violation de ce principe augmente considérablement le coût de la maintenance, car elle nécessite des interventions répétées dans plusieurs endroits du code source.
Pour appliquer cette règle, il est essentiel d'écrire des méthodes concises avec des responsabilités bien définies, puis de les réutiliser.

Exemple :

```
// Mauvais exemple : duplication de code
public class Calculateur {
    public int addition(int a, int b) {
        return a + b;
    }

    public int soustraction(int a, int b) {
        return a - b;
    }
}

// Bon exemple : réutilisation de méthode
public class Calculateur {
    public int calculer(int a, int b, BiFunction<Integer, Integer, Integer>
operation) {
```

```
            return operation.apply(a, b);
        }
    }
```

2. KISS : Keep It Simple, Stupid!

Assurez-vous de répondre aux besoins de la manière la plus simple possible, sans recourir à des abstractions inutiles ou complexes.
L'application de ce principe garantit un code plus lisible et plus facile à maintenir.

I **Exemple :**

```
// Mauvais exemple : code complexe inutile
public int calculerSomme(int[] nombres) {
    int somme = 0;
    for (int i = 0; i < nombres.length; i++) {
        somme += nombres[i];
    }
    return somme;
}

// Bon exemple : code simple et lisible
public int calculerSomme(int[] nombres) {
    return Arrays.stream(nombres).sum();
}
```

3. YAGNI : You Aren't Gonna Need It

Ce principe rejoint le précédent en insistant sur l'importance de ne pas développer de fonctionnalités sans un besoin fonctionnel réel.
Évitez la surconception et concentrez-vous uniquement sur ce qui est nécessaire.

Exemple :

```
// Mauvais exemple : ajout d'une fonctionnalité inutile
public class Utilisateur {
    private String nom;
    private String adresse;
    private String numeroDeFax; // Inutile si non requis
}

// Bon exemple : se concentrer sur les besoins actuels
public class Utilisateur {
    private String nom;
    private String adresse;
}
```

4. les 5 principes SOLID

a. S : Single Responsability principle (SRP)

Chaque composant, classe ou fonction doit avoir une seule responsabilité.

Exemple :

```
// Mauvais exemple : une classe avec plusieurs responsabilités
public class GestionnaireUtilisateur {
    public void enregistrerUtilisateur() { /* ... */ }
    public void envoyerEmail() { /* ... */ }
}

// Bon exemple : séparation des responsabilités
public class GestionnaireUtilisateur {
    public void enregistrerUtilisateur() { /* ... */ }
}

public class ServiceEmail {
    public void envoyerEmail() { /* ... */ }
}
```

b. O : Open/Closed principle (OCP)

Chaque composant doit être ouvert à l'extension, mais fermé à la modification.

Exemple :

```
// Mauvais exemple : modification d'une classe existante
public class Calculateur {
    public int calculer(int a, int b, String operation) {
        if (operation.equals("addition")) {
            return a + b;
        } else if (operation.equals("soustraction")) {
            return a - b;
        }
        return 0;
    }
}

// Bon exemple : extension via des classes
public interface Operation {
    int appliquer(int a, int b);
}

public class Addition implements Operation {
    public int appliquer(int a, int b) {
        return a + b;
    }
```

```
}

public class Calculateur {
    public int calculer(int a, int b, Operation operation) {
        return operation.appliquer(a, b);
    }
}
```

c. L : Liskov substitution principle (LSP)

L'utilisation d'une implémentation d'une classe abstraite ou d'une interface ne doit pas altérer le contrat défini par cette classe abstraite ou interface.

Exemple :

```
// Mauvais exemple : une sous-classe qui viole le contrat
public class Rectangle {
    public void setLargeur(int largeur) { /* ... */ }
    public void setHauteur(int hauteur) { /* ... */ }
}

public class Carre extends Rectangle {
    @Override
    public void setLargeur(int largeur) {
        super.setLargeur(largeur);
        super.setHauteur(largeur); // Comportement inattendu
    }
}

// Bon exemple : respecter le contrat
public interface Forme {
    int calculerSurface();
}

public class Rectangle implements Forme {
    private int largeur, hauteur;
    public int calculerSurface() {
        return largeur * hauteur;
    }
}

public class Carre implements Forme {
    private int cote;
    public int calculerSurface() {
        return cote * cote;
    }
}
```

d. I : Interface segregation principle (ISP)

Un client ne doit jamais être contraint de dépendre d'une interface qu'il n'utilise pas.

Exemple :

```
// Mauvais exemple : interface trop large
public interface Machine {
    void imprimer();
    void scanner();
    void faxer();
}

// Bon exemple : interfaces spécifiques
public interface Imprimante {
    void imprimer();
}

public interface Scanner {
    void scanner();
}
```

e. D : Dependency inversion principle (DIP)

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Exemple :

```
// Mauvais exemple : dépendance directe sur une classe concrète
public class Service {
    private MySQLDatabase database = new MySQLDatabase();
}

// Bon exemple : dépendance sur une abstraction
public interface Database {
    void connecter();
}

public class MySQLDatabase implements Database {
    public void connecter() { /* ... */ }
}

public class Service {
    private Database database;

    public Service(Database database) {
        this.database = database;
    }
}
```