Les Principes SOLID en Java

Découvrez les principes SOLID, essentiels pour un code propre.

MA **par Mohamed Alouani**

Principe de Responsabilité Unique (SRP)

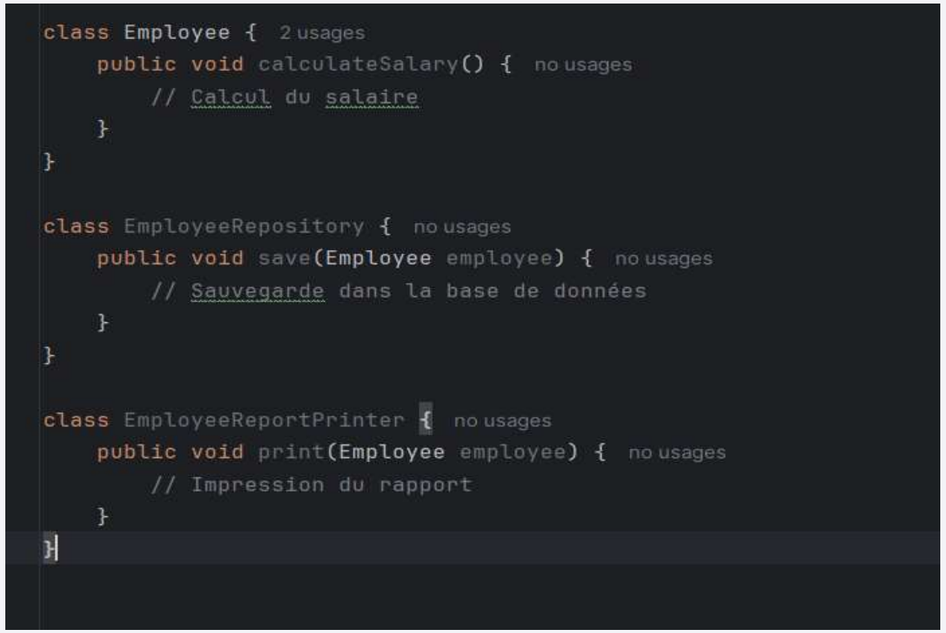
- Chaque classe doit avoir une seule raison de changer.

Mauvais pratique

```
class Employee {  no usages
    public void calculateSalary() {  no usages
        // Calcul du salaire
    }
    public void saveToDatabase() {  no usages
        // Sauvegarde dans la base de données
    }
    public void printReport() {  no usages
        // Impression du rapport
    }
}
```

Problème : Cette classe gère à la fois la logique métier, la persistance et l'affichage, ce qui la rend difficile à maintenir.

Bon pratique



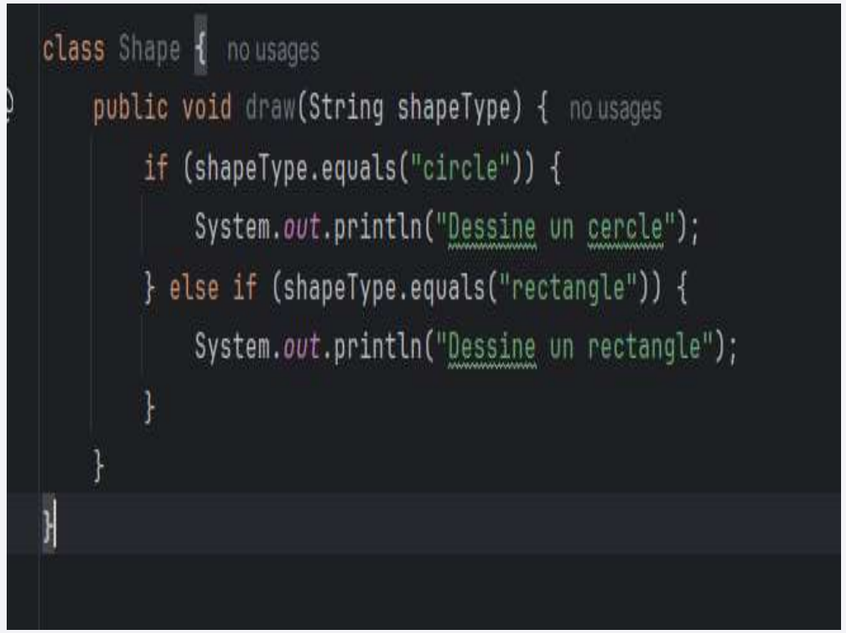
```
class Employee {  2 usages
    public void calculateSalary() {  no usages
        // Calcul du salaire
    }
}

class EmployeeRepository {  no usages
    public void save(Employee employee) {  no usages
        // Sauvegarde dans la base de données
    }
}

class EmployeeReportPrinter {  no usages
    public void print(Employee employee) {  no usages
        // Impression du rapport
    }
}
```

Avantage : Chaque classe a une responsabilité claire.

Principe Ouvert/Fermé (OCP)

- Étendre le comportement sans modifier le code existant.

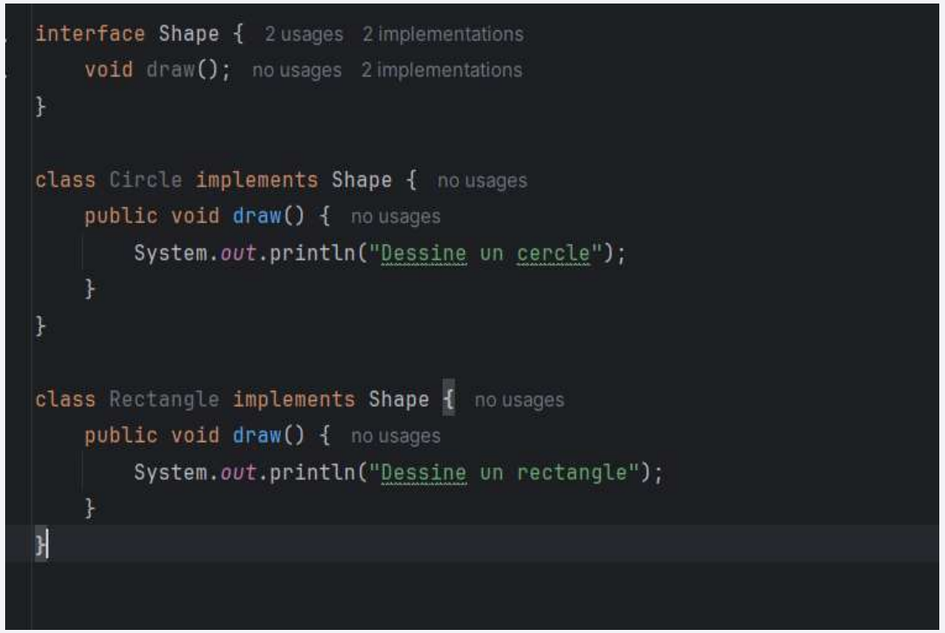
Mauvais pratique



```
class Shape {  no usages
    public void draw(String shapeType) {  no usages
        if (shapeType.equals("circle")) {
            System.out.println("Dessine un cercle");
        } else if (shapeType.equals("rectangle")) {
            System.out.println("Dessine un rectangle");
        }
    }
}
```

Problème : Chaque nouvelle forme oblige à modifier la méthode

Bon pratique



```
interface Shape {  2 usages  2 implementations
    void draw();  no usages  2 implementations
}

class Circle implements Shape {  no usages
    public void draw() {  no usages
        System.out.println("Dessine un cercle");
    }
}

class Rectangle implements Shape {  no usages
    public void draw() {  no usages
        System.out.println("Dessine un rectangle");
    }
}
```

Avantage : Pour ajouter une nouvelle forme, il suffit d'ajouter une nouvelle classe, pas de modifier les existantes.

Principe de Substitution de Liskov (LSP)

- Les objets d’une classe dérivée doivent pouvoir remplacer les objets de la classe mère sans altérer le comportement.

Mauvais pratique

```
class Bird {  1 usage  1 inheritor
    public void fly() {  no usages  1 override
        System.out.println("Je peux voler");
    }
}

class Ostrich extends Bird {  no usages
    @Override  no usages
    public void fly() {
        throw new UnsupportedOperationException("Je ne peux pas voler");
    }
}
```

Problème : Ostrich ne respecte pas le comportement attendu de Bird.

Bon pratique

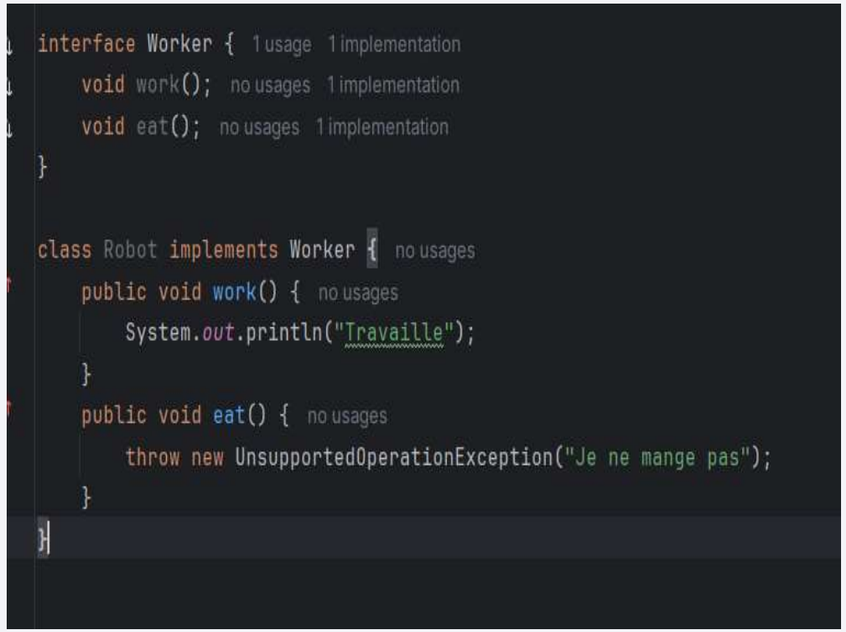
```
interface Bird {}  2 usages  3 implementations

interface FlyingBird extends Bird {  1 usage  1 implementation
    void fly();  no usages  1 implementation
}

class Sparrow implements FlyingBird {  no usages
    public void fly() {  no usages
        System.out.println("Je peux voler");
    }
}

class Ostrich implements Bird {  no usages
    // N'a pas la méthode fly, donc pas de problème
}
```

Avantage : On ne force pas Ostrich à implémenter une méthode qu’elle ne peut pas respecter.

Principe de Ségrégation d'Interface (ISP)

- Les clients ne doivent pas être forcés d’implémenter des interfaces qu’ils n’utilisent pas.

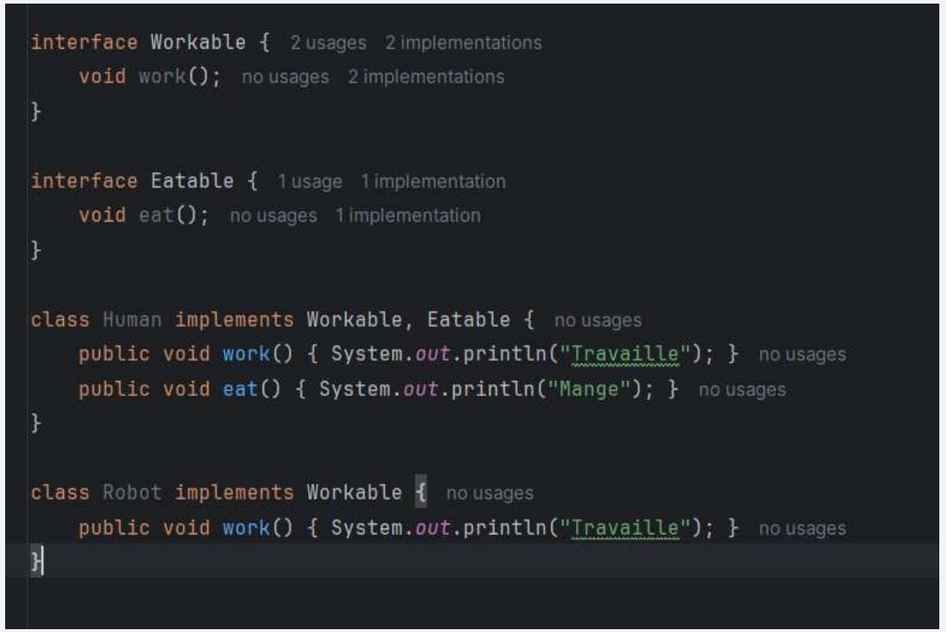
Mauvais pratique



```
interface Worker {  1 usage  1 implementation
    void work();  no usages  1 implementation
    void eat();  no usages  1 implementation
}

class Robot implements Worker {  no usages
    public void work() {  no usages
        System.out.println("Travaille");
    }
    public void eat() {  no usages
        throw new UnsupportedOperationException("Je ne mange pas");
    }
}
```

Problème : Robot est obligé d’implémenter la méthode eat() alors qu’il ne mange pas.

Bon pratique



```
interface Workable {  2 usages  2 implementations
    void work();  no usages  2 implementations
}

interface Eatable {  1 usage  1 implementation
    void eat();  no usages  1 implementation
}

class Human implements Workable, Eatable {  no usages
    public void work() { System.out.println("Travaille"); }  no usages
    public void eat() { System.out.println("Mange"); }  no usages
}

class Robot implements Workable {  no usages
    public void work() { System.out.println("Travaille"); }  no usages
}
```

Avantage : Chaque classe implémente uniquement ce qui est nécessaire.

Principe d'Inversion de Dépendance (DIP)

- Dépendre des abstractions, pas des classes concrètes.

Mauvais pratique

```
class MySQLDatabase {  2 usages
    public void save(String data) {  1 usage
        System.out.println("Sauvegarde dans MySQL: " + data);
    }
}

class UserService {  no usages
    private MySQLDatabase db = new MySQLDatabase();  1 usage

    public void saveUser(String user) {  no usages
        db.save(user);
    }
}
```

Problème : UserService dépend directement de MySQLDatabase, ce qui rend le changement difficile.

Bon pratique

```
interface Database {  3 usages  1 implementation
    void save(String data);  1 usage  1 implementation
}

class MySQLDatabase implements Database {  no usages
    public void save(String data) {  1 usage
        System.out.println("Sauvegarde dans MySQL: " + data);
    }
}

class UserService {  no usages
    private Database db;  2 usages

    public UserService(Database db) {  no usages
        this.db = db;
    }

    public void saveUser(String user) {  no usages
        db.save(user);
    }
}
```

Avantage : UserService ne dépend que d'une abstraction, ce qui facilite le remplacement de la base de données.

Résumé des principes SOLID

Principe	Ce qu’il dit	Pourquoi utile
SRP	Une classe = une responsabilité	Facilite la maintenance
OCP	Ouvert à l’extension, fermé à la modif.	Moins de bugs en production
LSP	Sous-classes substituables	Préserve le comportement attendu
ISP	Interfaces spécifiques	Réduit le code inutile
DIP	Dépendre des abstractions	Meilleure flexibilité et tests

Avantages des Principes SOLID

Couplage Réduit

Classes faiblement liées facilitent les modifications.

Tests Faciles

Composants isolés simplifient les tests unitaires.

Réutilisation Optimale

Code modulaire et adaptable dans divers contextes.

Maintenabilité

Projets plus robustes et évolutifs sur le long terme.

Conclusion

Les principes SOLID sont une boussole pour un code Java sain.

Ils augmentent qualité, robustesse et adaptabilité du logiciel.

Adoptez-les avec pragmatisme pour vos projets actuels et futurs.