DIAGRAMME DE CLASSE DES PRINCIPES SOLID

AVANT ET APRES LE REFACTORING

Ce document propose une explication simple, progressive et illustrée des cinq principes **SOLID**, à partir d'extraits de code volontairement incorrects. L'objectif est de montrer concrètement les erreurs de conception les plus courantes, d'expliquer clairement pourquoi ces codes posent problème, puis de présenter des solutions plus propres et plus maintenables.

1. Single Responsibility Principle (SRP)

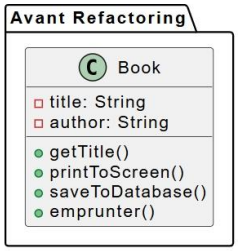
Extrait du code erroné:

```
// Classe qui gère plusieurs responsabilités
public class Book {
private String title;
private String author;
private String content;
// Gestion des données
public String getTitle() { return title; }
public String getAuthor() { return author; }
public String getContent() { return content; }

public void printToScreen() {
System.out.println(title + " - " + author);
}
public void saveToDatabase() {
System.out.println("Sauvegarde en base de données");
}
}
```

Problème

La classe **Book** fait trop de choses à la fois. Elle gère à la fois les données du livre, son affichage et sa sauvegarde. Cela signifie que la moindre modification liée à l'affichage ou à la base de données oblige à modifier cette même classe.

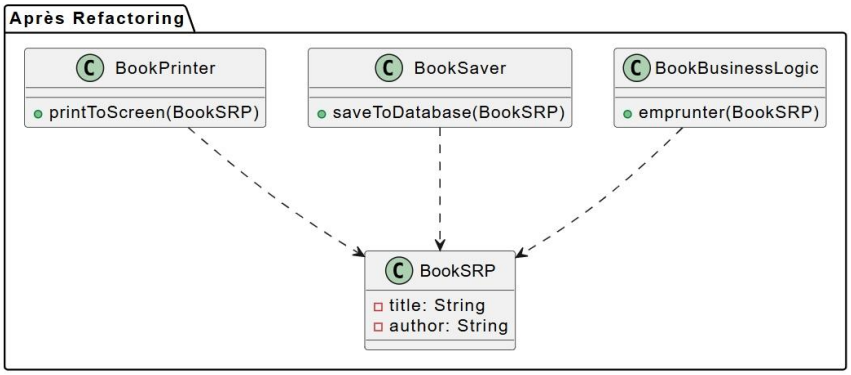
En pratique, ce type de conception rend le code difficile à maintenir, fragile face aux changements et source d'erreurs. C rt couplage et d'une mauvaise séparation des responsabilités.



Solution

La solution consiste à séparer clairement les rôles. Chaque classe doit avoir une seule responsabilité bien définie :

- **Book** : données métier,
- **BookPrinter** : affichage,
- **BookS**



2. Principe LSP

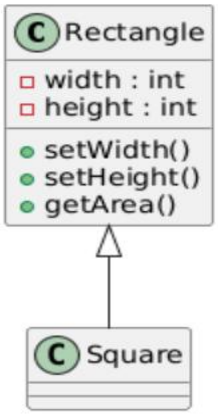
```
class Rectangle {
protected int width;
protected int height;
public void setWidth(int width) {
this.width = width;
}
public void setHeight(int height) {
this.height = height;
}
public int getArea() {
return width * height;
}
}
class Square extends Rectangle {
@Override
public void setWidth(int width) {
this.width = width;
this.height = width;
}
@Override
public void setHeight(int height) {
```

```
this.width = height;
this.height = height;}}
```

Problème

Même si, en théorie, un carré est un type particulier de rectangle, cette relation ne fonctionne pas correctement en programmation. Lorsqu'un Square est utilisé à la place d'un Rectangle, le comportement obtenu n'est plus celui attendu.

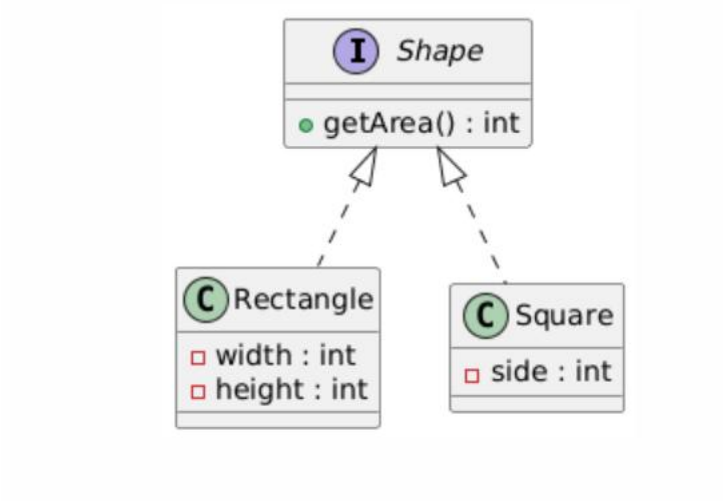
Cela entraîne des ré le le contrat défini par la classe parente. L cté, ce qui viole le principe de substitu



Solution

Supprimer l'héritage incorrect et utiliser :

- une interface commune Shape, des implémentations indépendantes Rectangle et Square.



3. Open / Closed Principle (OCP)

Extrait du code erroné

```
public void processPayment(String paymentType, double amount) {
if (paymentType.equals("mobile")) {
// paiement mobile
} else if (paymentType.equals("card")) {
// paiement carte
}
}
```

Problème

À chaque fois qu'un nouveau moyen de paiement doit être ajouté, il est nécessaire de modifier la méthode existante en ajoutant une nouvelle condition. Ce fonctionnement est risqué, car une modification peut casser un comportement déjà fonctionnel.

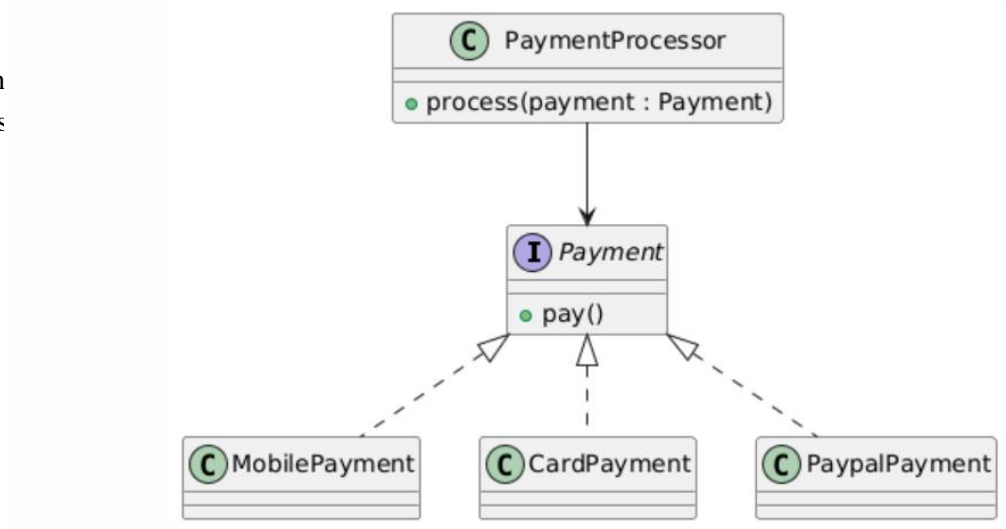
Le code devient alors rigide, difficile à faire évoluer et coûteux à maintenir.

Solution

La bonne approche consiste à utiliser une abstraction et le polymorphisme. Chaque moyen de paiement est représenté par une classe indépendante qui implémente une même interface :

- une interface PaymentStrategy,

Ain
exis



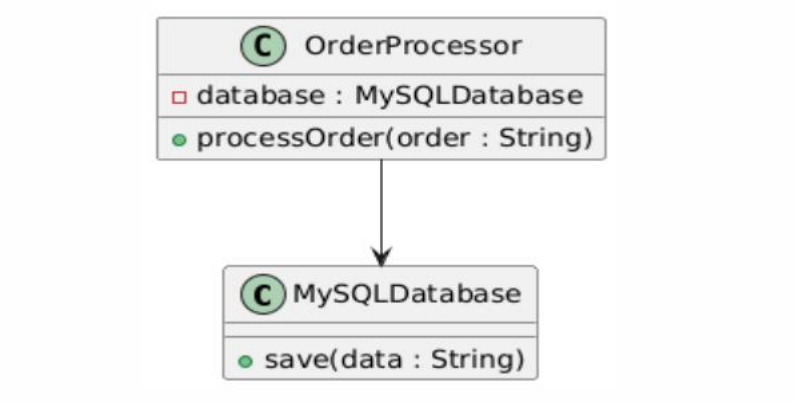
4. Principe DIP

```
class MySQLDatabase {

    public void save(String data) { }

}

class OrderProcessor {

    private MySQLDatabase database = new MySQLDatabase();



    public void process(String order) {

        database.save(order);

    }

}
```

Problème

La classe OrderProcessor, qui contient la logique métier principale, dépend directement d'une implémentation précise de base de données. Ce choix rend l'application peu flexible.

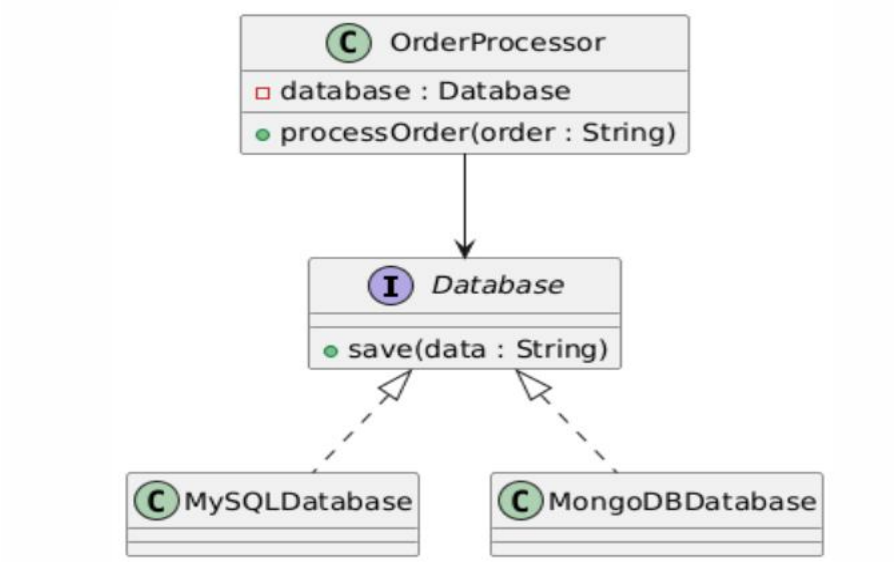
Changer de technologie de stockage oblige à modifier le code métier, ce qui est contraire aux bonnes pratiques de conception. (MySQLDatabase). Il est impossible de changer de base de données sans modifier le code métier.



Solution

Introduire une abstraction **Database** :

- **OrderProcessor** dépend de l'interface,
- les bases de données concrètes implémentent cette interface.



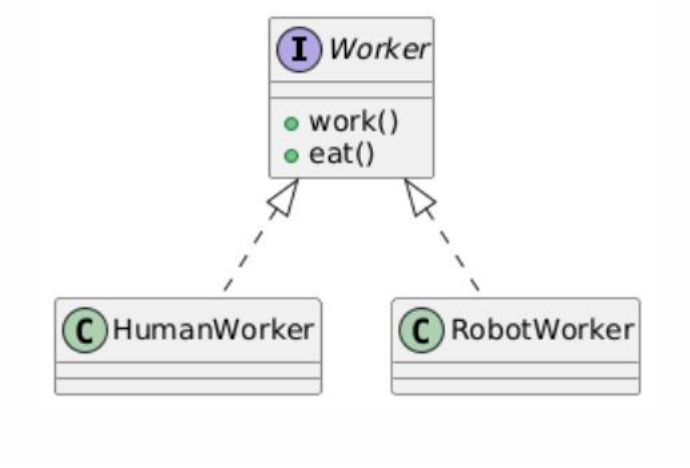
5. Principe ISP

```
interface Worker {
    void work();
    void eat();
}
class RobotWorker implements Worker {
    public void work() { }
    public void eat() {
        throw new UnsupportedOperationException(); }}
```

Problème

Certaines classes sont obligées d'implémenter des méthodes qui n'ont aucun sens pour elles. Par exemple, un robot n'a pas besoin de manger, pourtant il est contraint d'implémenter la méthode **eat()**.

Cela conduit soit à du code inutile, soit à des exceptions lancées à l'exécution, ce qui est un mauvais design.

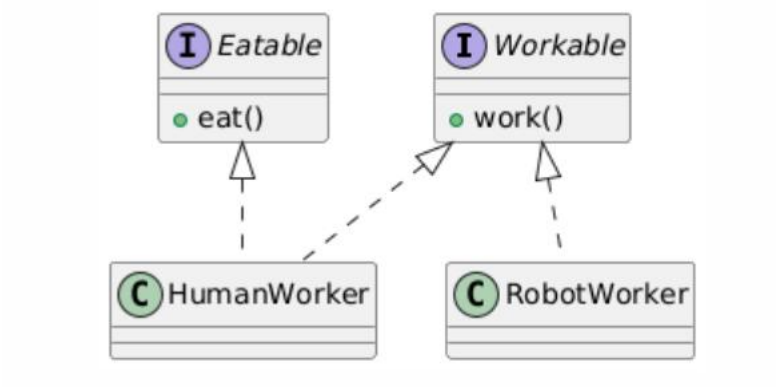


Solution

Découper l'interface en interfaces plus petites et spécifiques :

- **Workable,**
- **Eatable.**

Chaque classe n'implémente que les méthodes dont elle a réellement besoin.



Conclusion

Les principes SOLID permettent d'éviter les erreurs classiques de conception rencontrées dans de nombreuses applications. En les appliquant correctement, le code devient plus clair, plus facile à comprendre et surtout plus simple à faire évoluer.

Une bonne application de ces principes améliore la maintenabilité, réduit le risque d'erreurs lors des modifications et favorise la réutilisation du code. C'est pour cette

raison que les principes SOLID occupent une place centrale dans la conception logicielle moderne.