Lecon3: Principe de conception SOLID en poo

SOLID est un ensemble de cinq principes de conception qui aident à développer un code plus lisible, évolutif et maintenable. Voici les acronymes pour chaque principe :

1. SRP - Single Responsibility Principle (Principe de Responsabilité Unique) :

Principe

Une classe doit avoir une seule responsabilité, c'est-à-dire qu'elle ne doit être responsable que d'une seule tâche ou fonctionnalité.

Exemple en java

```java
public class Order {
    private int orderId;
    private int customerId;
    private Date orderDate;
    private List<OrderItem> orderItems;

    public double calculateOrderTotal() {
        double orderTotal = 0;
        for (OrderItem item : orderItems) {
            orderTotal += item.getPrice() * item.getQuantity();
        }
        return orderTotal;
    }

    public void saveOrder() {
        // Code pour sauvegarder la commande dans la base de données
    }
}
```

Dans cet exemple, la classe `Order` a une seule responsabilité, c'est-à-dire de représenter une commande. La méthode `calculateOrderTotal()` calcule le montant total de la commande et la méthode `saveOrder()` sauvegarde la commande dans la base de données. Cette classe respecte donc le principe de responsabilité unique.

2. OCP (Open/Closed Principle) : Principe ouvert/fermé

principe

Les classes devraient être ouvertes à l'extension mais fermées à la modification.

Exemple en java

```java
public interface Shape {
    double calculateArea();
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double calculateArea() {
        return width * height;
    }
}

public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

Dans cet exemple, l'interface `Shape` définit une méthode `calculateArea()` pour calculer l'aire d'une forme. Les classes `Rectangle` et `Circle` implémentent cette interface et fournissent une implémentation pour la méthode `calculateArea()`. Si une nouvelle forme doit être ajoutée, il suffit de créer une nouvelle classe qui implémente l'interface `Shape` et fournit une implémentation appropriée pour la méthode `calculateArea()`. Ainsi, la classe `Shape` est ouverte aux extensions mais fermée aux modifications.

3. LSP (Liskov Substitution Principle) : Principe de substitution de Liskov

Principe

les objets d'une classe dérivée doivent être utilisables en lieu et place des objets de la classe de base sans que cela ne provoque d'erreurs ou de comportements inattendus

Exemple en java

```java
public class Bird {
    public void fly() {
        System.out.println("I can fly.");
    }
}

public class Ostrich extends Bird {
    public void fly() {
        throw new UnsupportedOperationException("Ostriches cannot fly.");
    }
}
```

Dans cet exemple, la classe `Ostrich` hérite de la classe `Bird` mais ne peut pas voler. Cependant, elle redéfinit la méthode `fly()` et lève une exception `UnsupportedOperationException`. Cette classe respecte donc le principe de substitution de Liskov, qui stipule qu'une instance de la classe dérivée doit pouvoir être utilisée à la place de l'instance de la classe de base sans altérer le comportement du programme.

4. Interface Segregation Principle (ISP) : Principe de ségrégation des interfaces

principe

Aucune classe ne doit être forcée d'implémenter des méthodes dont elle n'a pas besoin

Exemple en java

```
interface OnlineOrder {
  void placeOrder();
}

interface StoreOrder {
  void cancelOrder();
}

class OnlineClient implements OnlineOrder {
  public void placeOrder() {
    // implémentation pour passer une commande en ligne
  }
}

class StoreClient implements StoreOrder {
  public void cancelOrder() {
    // implémentation pour annuler une commande en magasin
  }
}
```

Supposons que nous ayons une interface générique `Order` pour passer une commande qui a deux méthodes `placeOrder()` et `cancelOrder()`. Dans une entreprise, il y a différents types de clients comme les clients en ligne, les clients en magasin, etc. Nous pouvons implémenter les clients à partir de l'interface `Order` mais cela signifierait que chaque client doit également implémenter les deux méthodes `placeOrder()` et `cancelOrder()` même si l'un d'entre eux n'est pas nécessaire pour un type spécifique de client.

Pour résoudre ce problème, nous pouvons diviser l'interface `Order` en deux interfaces distinctes : `OnlineOrder` et `StoreOrder`. L'interface `OnlineOrder` aura la méthode `placeOrder()` tandis que l'interface `StoreOrder` aura la méthode `cancelOrder()`. De cette façon, chaque client pourra implémenter l'interface qui convient le mieux à ses besoins.

5. Dependency Inversion Principle (DIP) : Principe d'inversion de dépendance

principe

les classes de haut niveau ne doivent pas dépendre de classes de bas niveau, mais plutôt d'abstractions

Exemple en java

Supposons que nous ayons une application de service de messagerie qui permet aux utilisateurs de se connecter, d'envoyer des messages, de recevoir des messages et de voir leur historique de

messages. Pour implémenter cette application, nous avons une classe `UserService` qui contient des méthodes pour gérer les utilisateurs, une classe `MessageService` pour gérer les messages, et une classe `HistoryService` pour gérer l'historique des messages.

Au lieu de créer une dépendance directe entre ces services, nous pouvons créer une interface commune `IMessageService` qui expose les méthodes communes à tous les services de messagerie, et nous faisons en sorte que chaque service implémente cette interface. De même, nous pouvons créer une interface `IUserService` qui expose les méthodes communes à tous les services utilisateur, et nous faisons en sorte que chaque service utilisateur implémente cette interface.

```java
public interface IMessageService {
    public void sendMessage(String message);
    public List<String> getMessages();
}

public interface IUserService {
    public void addUser(String username, String password);
    public void deleteUser(String username);
}

public class UserService implements IUserService {
    // Implémentation des méthodes addUser() et deleteUser()
}

public class MessageService implements IMessageService {
    // Implémentation des méthodes sendMessage() et getMessages()
}

public class HistoryService implements IMessageService {
    // Implémentation des méthodes sendMessage() et getMessages()
}

public class MessagingApp {
    private IUserService userService;
    private IMessageService messageService;

    public MessagingApp(IUserService userService, IMessageService messageService) {
        this.userService = userService;
        this.messageService = messageService;
    }

    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
    }

    public List<String> getMessages() {
        return this.messageService.getMessages();
    }

    public void addUser(String username, String password) {
        this.userService.addUser(username, password);
    }

    public void deleteUser(String username) {
        this.userService.deleteUser(username);
    }
}

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

```
        IUserService userService = new UserService();
        IMessageService messageService = new MessageService();

        MessagingApp messagingApp = new MessagingApp(userService, messageService);

        messagingApp.addUser("john_doe", "password123");
        messagingApp.sendMessage("Hello world!");
        List<String> messages = messagingApp.getMessages();
        messagingApp.deleteUser("john_doe");
    }
}
```

Dans cet exemple, la classe `MessagingApp` dépend des interfaces `IUserService` et `IMessageService`, plutôt que des implémentations concrètes de ces interfaces. Cela permet de faciliter la maintenance et la modification de l'application, car les services peuvent être remplacés ou modifiés sans avoir à modifier la classe MessagingApp.