Université Sultan Moulay Slimane

Ecole Nationale des Sciences Appliquées de Khouribga

Département : Mathématiques & informatique

Filière : IID (3ème année)

Module : Génie logiciel

Professeur : M. Nasri Mohammed

**TP2 : Principes SOLID**

L’objectif de ce TP est de manipuler les principes SOLID, corriger une mauvaise conception en appliquant l’ensemble de ces principes.

Principe 1 : Single Responsability

Soit la classe suivante :

public class Employee {

private String name;

private int id;

private double salary;

public Employee(String name, int id, double salary) {

this.name = name;

this.id = id;

this.salary = salary;

}

public void saveToDatabase() {

// Code pour sauvegarder l'employé dans la base de données.

}

public void printEmployeeReport() {

// Code pour générer le rapport de l'employé.

}

}

Cette classe ne respecte pas le principe Single Responsability. Corrigez-la.

Princile 2 : Open-Closed

Soient les classes suivantes :

public class EmployeeMySqlDao {

private Connection connection; // Initialisation de la connexion à la base de données MySQL.

public EmployeeMySqlDao() {

// Initialisation de la connexion MySQL.

}

public void saveEmployee(Employee employee) {

// Code pour sauvegarder un employé dans MySQL.

}

public Employee getEmployeeById(int employeeId) {

// Code pour récupérer un employé par ID depuis MySQL.

}

public List<Employee> getAllEmployees() {

// Code pour récupérer tous les employés depuis MySQL.

}

}

Si nous voulons ajouter une classe pour PostgreSql ou SqlServer, nous devons modifier cette classe ou en créer des nouvelles, ceci ne respecte pas le principe Open-Close, corrigez.

Principe 3 : Liskov Substitution

**class** CoffeeShop {

String takeaway() { **return** ""; }

//...

}

**class** A **extends** CoffeeShop {

String takeaway() {

**return** "Delivery at most 30 minutes";

}

}

**class** B **extends** CoffeeShop {

String takeaway() {

**throw** **new** RuntimeException("We do not have takeaway service");

}

}

Dans cet exemple, la classe **B** ne peut pas remplacer sa classe mère, la fonction **takeaway** ne peut donc pas être membre de **CoffeeShop** (le principe de Liskov Substitution est violé), proposez une solution.

// Classe de base pour l'accès à la base de données

public class DatabaseAccessor {

protected String connectionString;

public DatabaseAccessor(String connectionString) {

this.connectionString = connectionString;

}

public void connect() {

// Code pour établir une connexion à la base de données.

}

public void executeQuery(String query) {

// Code pour exécuter une requête SQL générique.

}

}

// Classe d'accès à la base de données MySQL

public class MySqlDatabaseAccessor extends DatabaseAccessor {

public MySqlDatabaseAccessor(String connectionString) {

super(connectionString);

}

public void executeMySqlQuery(String query) {

// Code spécifique pour exécuter une requête SQL dans MySQL.

}

}

// Classe d'accès à la base de données Oracle

public class OracleDatabaseAccessor extends DatabaseAccessor {

public OracleDatabaseAccessor(String connectionString) {

super(connectionString);

}

public void executeOracleQuery(String query) {

// Code spécifique pour exécuter une requête SQL dans Oracle.

}

}

Principe 4 : Interface Segregation

**interface** ICoffeeShop{

//traditional shops

**void** brewByEspressoMachine();

**void** brewMachinePourOver();

//third wave shops

**void** brewByHandHeldEspressoMaker();

**void** brewManualPourOver();

//both

**void** brewFilterCoffee();

}

**class** Traditional **implements** ICoffeeShop {

**public** **void** brewByEspressoMachine() {

//...

}

**public** **void** brewMachinePourOver() {

//...

}

**public** **void** brewFilterCoffee() {

//...

}

**public** **void** brewByHandHeldEspressoMaker() {

**throw** **new** RuntimeException("We don't brewByHandHeldEspressoMaker");

}

**public** **void** brewManualPourOver() {

**throw** **new** RuntimeException("We don't brewManualPourOver");

}

}

**class** ThirdWave **implements** ICoffeeShop {

**public** **void** brewByEspressoMachine() {

**throw** **new** RuntimeException("We don't brewByEspressoMachine");

}

**public** **void** brewMachinePourOver() {

**throw** **new** RuntimeException("We don't brewMachinePourOver");

}

**public** **void** brewFilterCoffee() {

//...

}

**public** **void** brewByHandHeldEspressoMaker() {

//...

}

**public** **void** brewManualPourOver() {

//...

}

}

Dans cet exemple, nous avons une interface **ICoffeeShop** qui sert pour tous les types de CoffeShop, les clients de cette interface (qui sont les classes qui l’implémentent) doivent implémenter toutes les méthodes même celles qui ne les concernent pas. Quelle solution proposez-vous ?

Principe 5 : Dependency Inversion

**class** CoffeeShop {

**void** getPayment() {

}

**void** deliverCoffee() {

}

}

**class** Customer {

**void** makePayment() {

}

**void** receiveCoffee() {

}

}

**class** Delivery {

Customer customer;

CoffeeShop coffeeShop;

Delivery(Customer customer, CoffeeShop coffeeShop) { }

**void** deliver() {

customer.makePayment();

coffeeShop.getPayment();

coffeeShop.deliverCoffee();

customer.receiveCoffee();

}

}

Dans cet exemple, la classe **Delivery** est liée aux deux classes **Customer** et **CoffeeShop**. Ce qui va à l’encontre avec le principe de l’inversion de dépendance. Proposez une solution pour respecter ce principe.