**Tài Liệu Về SOLID**

# Giới Thiệu:

Lập trình hướng đối tượng (object-oriented programming – OOP) là một trong những mô hình lập trình được sử dụng nhiều nhất. Các tính chất đặc biệt khiến việc hướng đối tượng trở nên hiệu quả đó là:

* Tính trừu tượng (abstraction): Tạo ra các lớp trừu tượng mô hình hoá các đối tượng trong thế giới thực.
* Tính đóng gói (Encapsulation): Các thực thể của lớp trừu tượng có các giá trị thuộc tính riêng biệt.
* Tính kế thừa (Inheritance): Các đối tượng có thể dễ dàng kế thừa và mở rộng lẫn nhau.
* Tính đa hình (Polymorphism): Có thể thực hiện một hành động đơn theo nhiều cách thức khác nhau tuỳ theo loại đối tượng cụ thể đang được gọi.

Tính kế thừa:

* Kế thừa trong Java là một tính năng của lập trình hướng đối tượng, cho phép một lớp (class) mới tiếp nhận các thuộc tính và phương thức của một lớp khác. Lớp mới được gọi là lớp con (subclass), và lớp mà nó kế thừa được gọi là lớp cha (superclass). Kế thừa giúp tái sử dụng mã nguồn, giảm độ lặp lại và tăng tính mô-đun của chương trình.
* Kế thừa chia làm 5 loại:
* Đơn kế thừa: được sử dụng khi ta muốn tạo ra một lớp con duy nhất, kế thừa từ một lớp cha duy nhất.
* Kế thừa đa cấp: được sử dụng khi ta cần tạo ra các lớp con phức tạp và đa dạng, các lớp con này có thể kế thừa các thuộc tính và phương thức từ nhiều lớp cha khác nhau.
* Kế thừa phân cấp:  một lớp cha được kế thừa bởi nhiều lớp con khác nhau.
* Đa kế thừa: là khả năng kế thừa các tính năng từ nhiều lớp cha khác nhau vào một lớp con. (Không được sử dụng vì dễ gây mâu thuẫn khi các lớp cha có phương thức giống nhau)
* Kế thừa hỗn hợp: là sự kết hợp giữa kế thừa đa cấp (multilevel inheritance) và kế thừa đa dạng (multiple inheritance).
* Ví dụ:

// Lớp cơ bản (Base class)

class Animal {

void eat() {

System.out.println("This animal is eating.");

}

}

// Lớp con (Subclass) kế thừa từ lớp Animal

class Dog extends Animal {

@Override

void eat() {

System.out.println("The dog is eating dog food.");

}

}

// Lớp con (Subclass) kế thừa từ lớp Animal

class Cat extends Animal {

@Override

void eat() {

System.out.println("The cat is eating cat food.");

}

}

// Lớp chính để kiểm tra

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat();

Cat cat = new Cat();

cat.eat();

}

}

* Tính đa hình:

Ví dụ:

// Lớp cơ bản (Base class)

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

// Lớp con (Subclass) cụ thể hóa hành vi

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

}

// Lớp con (Subclass) cụ thể hóa hành vi

class Cat extends Animal {

@Override

void makeSound() {

System.out.println("Cat meows");

}

}

// Kiểm tra tính đa hình

public class Main {

public static void main(String[] args) {

Animal myDog = new Dog();

myDog.makeSound(); // Dog barks

Animal myCat = new Cat();

myCat.makeSound(); // Cat meows

}

}

* Tính đóng gói:

Ví dụ:

public class Person {

// Các thuộc tính được đóng gói

private String name;

private int age;

// Phương thức công khai để truy cập và thay đổi giá trị thuộc tính

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age > 0) {

this.age = age;

}

}

}

* Tính trừu tượng:

Ví dụ:

// Lớp trừu tượng mô hình hóa phương tiện giao thông

abstract class Vehicle {

abstract void move();

}

// Lớp con cụ thể hóa phương tiện giao thông là xe hơi

class Car extends Vehicle {

@Override

void move() {

System.out.println("Car is moving");

}

}

// Lớp con cụ thể hóa phương tiện giao thông là xe đạp

class Bicycle extends Vehicle {

@Override

void move() {

System.out.println("Bicycle is moving");

}

}

// Kiểm tra tính trừu tượng

public class Main {

public static void main(String[] args) {

Vehicle car = new Car();

car.move();

Vehicle bicycle = new Bicycle();

bicycle.move();

}

}

Các tính chất đặc biệt này của OOP giúp chúng ta xây dựng được các chương trình giải quyết được nhiều vấn đề cụ thể khác nhau trong thế giới thực. Hầu hết lập trình viên đều đã biết các tính chất này của OOP, nhưng cách thức để phối hợp các tính chất này với nhau để tăng hiệu quả của ứng dụng thì không phải ai cũng nắm được. Một trong những chỉ dẫn để giúp chúng ta sử dụng được OOP hiệu quả hơn đó là nguyên tắc SOLID.

SOLID là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc thiết kế hướng đối tượng. Giúp cho lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain. Nó được đưa ra bởi Robert C. Martin và Michael Feathers. 5 nguyên tắc đó bao gồm:

* Single responsibility priciple (SRP)
* Open/Closed principle (OCP)
* Liskov substitution principe (LSP)
* Interface segregation principle (ISP)
* Dependency inversion principle (DIP)

## Single responsibility priciple

Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.

Một class chỉ nên giữ 1 trách nghiệm duy nhất. Vi nếu 1 class có quá nhiều chức năng sẽ trở nên cồng kềnh, khó đọc, khó maintain. Mà đối với ngành IT việc requirement thay đổi, cần thêm sửa chức năng là rất bình thường, nên việc code trong sáng, dễ đọc dễ hiểu là rất cần thiết.

Ví dụ: Một nhân viên trong công ty phải làm các công việc như developer, tester, salemans. Khi đó nếu chúng ta cho 1 lớp nhiều chức năng ví dụ:

class Employee

{

string position;

function developSoftware(){};

function testSoftware(){};

function saleSoftware(){};

}

Điêù này sẽ khiến cho việc sau này cập nhật thêm 1 số vị trí mới chúng ta lại phải thay đổi lại lớp Employee sẽ ảnh hưởng và rất dễ xảy ra sai sót.

Áp dụng nguyên tắc Single Responsibility: mỗi lớp 1 trách nhiệm. Ta sẽ tạo 1 lớp trừu tượng là “Employee” có phương thức là working (), từ đây bạn kế thừa ra 3 lớp cụ thể là Developer, Tester và Salesman. Ở mỗi lớp này bạn sẽ implement phương thức working () cụ thể tuy theo nhiệm vụ của từng người. Khi đó chúng ta sẽ bị tình trạng dùng nhầm phương thức nữa.

## Open/Closed principle

Không được sửa đổi một Class có sẵn, nhưng có thể mở rộng bằng kế thừa.

Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) chứ không nên sửa đổi class cũ. Việc này dẫn đến tình trạng phát sinh nhiều class, nhưng chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới, nơi chứa các chức năng mới.

Thông thường việc mở rộng thêm chức năng thì phải viết thêm code, vậy để thiết kế ra một module có thể dễ dàng mở rộng nhưng lại hạn chế sửa đổi code ta cần làm gì. Cách giải quyết là tách những phần dễ thay đổi ra khỏi phần khó thay đổi mà vẫn đảm bảo không ảnh hưởng đến phần còn lại.

Ví dụ:

abstract class Connection()

{

public abstract function doConnect();

}

class SqlServer extends Connection

{

public function doConnect()

{

//connect with SqlServer

}

}

class MySql extends Connection

{

public function doConnect()

{

//connect with MySql

}

}

class ConnectionManager

{

public function doConnection(Connection $connection)

{

//something

//.................

//connection

$connection->doConnect();

}

}

* Áp dụng Abstract thiết kế lại các lớp SqlServer, MySql, Oracle…
* Các lớp này đều có chung nhiệm vụ tạo kết nối đến csdl tương ứng có thể gọi chung là Connection.
* Cách thức kết nối đến csdl thay đổi tùy thuộc vào từng loại kết nối nhưng có thể gọi chung là doConect.
* Vậy ta có lớp cơ sở Connection có phương thức doConnect, các lớp cụ thể là SqlServer, MySql, Oracle… kế thừa từ Connection và overwrite lại phương thức doConnect phù hợp với lớp đó.

Việc này sẽ giúp code dễ quản lý hơn khi thêm các phương thức mới…

## Liskov substitution principle

Các đối tượng (instance) kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.

## Interface segregation principle

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể.

Nguyên lý này rất dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ rất vất vả vì các class implement interface này sẽ bắt buộc phải phải thực thi toàn bộ các method của interface. Ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement và quản lý sẽ dễ hơn.

**Ví dụ**: Tách interface Worker thành các interface nhỏ hơn như Workable và Eatable.

interface Workable {

void work();

}

interface Eatable {

void eat();

}

class Robot implements Workable {

public void work() {

// Robot làm việc

}

}

class Human implements Workable, Eatable {

public void work() {

// Con người làm việc

}

public void eat() {

// Con người ăn

}

}

## Dependency inversion principle

Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại (Các class giao tiếp với nhau thông qua interface (abstraction), không phải thông qua implementation.)

Có thể hiểu nguyên lí này như sau: những thành phần trong 1 chương trình chỉ nên phụ thuộc vào những cái trừu tượng (abstraction). Những thành phần trừu tượng không nên phụ thuộc vào các thành phần mang tính cụ thể mà nên ngược lại.

Những cái trừu tượng (abstraction) là những cái ít thay đổi và biến động, nó tập hợp những đặc tính chung nhất của những cái cụ thể. Những cái cụ thể dù khác nhau thế nào đi nữa đều tuân theo các quy tắc chung mà cái trừu tượng đã định ra. Việc phụ thuộc vào cái trừu tượng sẽ giúp chương trình linh động và thích ứng tốt với các sự thay đổi diễn ra liên tục.

**Ví dụ**: Sử dụng interface để tách biệt sự phụ thuộc giữa các lớp

interface Keyboard {

void type();

}

class MechanicalKeyboard implements Keyboard {

public void type() {

// Gõ bàn phím cơ

}

}

class Computer {

private Keyboard keyboard;

public Computer(Keyboard keyboard) {

this.keyboard = keyboard;

}

public void type() {

keyboard.type();

}

}

public class Main {

public static void main(String[] args) {

Keyboard keyboard = new MechanicalKeyboard();

Computer computer = new Computer(keyboard);

computer.type();

}

}

# Tổng Kết:

SOLID là 5 nguyên tắc cơ bản trong việc thiết kế phần mềm. Nó giúp chúng ta tổ chức sắp xếp các function, method, class một cách chính xác hơn. Làm sao để kết nối các thành phần, module với nhau.

**Rõ ràng, dễ hiểu**

Teamwork là điều không thể tránh trong lập trình. Áp dụng SOLID vào công việc bạn sẽ tạo ra các hàm tốt, dễ hiểu hơn. Giúp cho bạn và đồng nghiệp đọc hiểu code của nhau tốt hơn.

**Dễ thay đổi**

SOLID giúp tạo ra các module, class rõ ràng, mạch lạc, mang tính độc lập cao. Do vậy khi có sự yêu cầu thay đổi, mở rộng từ khách hàng, ta cũng không tốn quá nhiều công sức để thực hiện việc thay đổi.

**Tái sử dụng**

SOLID khiến các lập trình viên suy nghĩ nhiều hơn về cách viết phần mềm, do vậy code viết ra sẽ mạch lạc, dễ hiểu, dễ sử dụng.