**Single Responsibility Principle**

**Nội dung:**

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

**Ví dụ làm rõ:**

**Ví dụ 1:**

Giả sử ta có class sau:

class Customer {

public void Add() {

try {

// Database code goes here

} catch (Exception ex) {

System.IO.File.WriteAllText(@"C:\log.txt", ex.ToString());

}

}

}

Class *Customer* ngoài việc thực hiện các xử lý đến đối tượng *Customer* còn thực hiện cả việc ghi log. Công việc ghi log thì quan trọng nhưng thực hiện nó như trên thì không tốt. Rõ ràng class *Customer* chỉ nên làm những việc như là validation dữ liệu, xử lý logic liên quan tới các dữ liệu của *Customer*.

Điều này sẽ gây ra khó khăn khi chúng ta muốn thay đổi việc ghi log, vì khi muốn đổi cách thức ghi log file chúng ta phải vào class *Customer* để sửa. Việc này thì không tốt chút nào.

Giải pháp:

Để đảm bảo nguyên lý này thì chúng ta sẽ chuyển đoạn code ghi log sang class *FileLogger*, class này chỉ làm việc với log mà thôi.

class FileLogger {

public void Handle(string message) {

System.IO.File.WriteAllText(@"c:\log.txt", message);

}

}

class Customer {

private FileLogger logger = new FileLogger();

publicvirtual void Add() {

try {

// Database code goes here

} catch (Exception ex) {

logger.Handle(ex.ToString());

}

}

}

Mọi thứ đã trở nên rõ ràng hơn trước. Class *Customer* chỉ làm việc với đối tượng customer và class *FileLogger* sẽ chuyên tâm làm việc với nhiệm vụ ghi log. Ở đây có thể dễ dàng thấy được lợi ích của việc tách 2 class này ra.

**Ví dụ 2:**

Hình dung cụ thể rằng mỗi nhân viên của một công ty phần mềm giữ 1 trong 3 vai trò sau: lập trình phần mềm (developer), kiểm thử phần mềm (tester), bán phần mềm (salesman). Mỗi nhân viên dựa vào chức vụ sẽ làm công việc tương ứng. Hình thành class Employee ban đầu như sau:

class Employee

{

string position;

function developSoftware(){};

function testSoftware(){};

function saleSoftware(){};

}

Khi ta thiết kế class *Employee* với thuộc tính position (developer, tester, salesman) và các phương thức *developSoftware()*, *testSoftware()* và *saleSoftware()* như thế này nếu có thêm một chức vụ nữa là quản lí nhân sự, thì ta phải vào sửa lại class *Employee* và thêm phương thức mới. Nếu có thêm nhiều chức vụ nữa thì sao? Khi đó các đối tượng được tạo ra sẽ dư thừa rất nhiều phương thức: chẳng hạn Developer thì đâu cần dùng hàm *testSoftware()* và *saleSoftware()*, chẳng may dùng lầm phương thức sẽ gây ra hậu quả khôn lường.

Giải pháp:

Để đảm bảo nguyên lý này, *Employee* sẽ là lớp trừu tượng có phương thức *working()*, tạo ra 3 class cụ thể là *Developer*, *Tester* và *Salesman* kế thừa lớp trừu tượng này. Mỗi class sẽ được implement phương thức *working()* cụ thể tùy theo vai trò của từng class. Khi đó chúng ta sẽ không bị tình trạng dùng sai phương thức.

**Kết luận:**

Về bản chất nguyên lý này chỉ là hướng dẫn không phải là nguyên tắc tuyệt đối. Có những trường hợp như các class Helper xét cho cùng toàn bộ các hàm trong class này đều thực hiện những tác vụ nhỏ nên nếu số lượng hàm ít vẫn có thể cho các hàm này vào cùng một class. Tuy nhiên khi số lượng hàm tăng lên quá nhiều thì nên cân nhắc sử dụng nguyên lý này để chia nhỏ module thuận tiện cho việc quản lý.

Việc hiểu và áp dụng nguyên này giúp cho việc viết code dễ đọc, dễ hiểu, dễ quản lý hơn. Tuy nhiên nguyên lý đơn nhiệm là nguyên lý đơn giản nhưng khó áp dụng đúng, việc xác định khi nào cần áp dụng khi nào không còn phụ thuộc vào việc người code xác định được đúng chức năng của module đang làm.

**Nguồn tham khảo**

<http://code4fun.vn/code4fun/page/17/>

<http://sieudaochichcode.com/2017/08/19/solid-tim-hieu-solid-de-tro-thanh-developer-gioi/>

<https://toidicodedao.com/2015/03/24/solid-la-gi-ap-dung-cac-nguyen-ly-solid-de-tro-thanh-lap-trinh-vien-code-cung/>

<https://nhungdongcodevui.com/2017/03/18/solid-la-gi-nguyen-tac-1-don-nhiem-single-responsibility-principle/>