ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

NGUYÊN LÝ SOLID

TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

NHẬP MÔN CÔNG NGHỆ PHẦN MỀM

Giảng viên: Đặng Việt Dũng

NHÓM:

22521074 - Nguyễn Hùng Phát
22520506 - Lê Minh Hùng
22252083 - Văn Công Gia Luật
22521189 - Thái Ngọc Quân
22521708 - Trần Phương Vy

NGÀY 8 THÁNG 5 NĂM 2024

NỘI DUNG

- Giới thiệu chung
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Câu hỏi và trả lời

GIỚI THIỆU CHUNG

SOLID là từ viết tắt của năm nguyên tắc thiết kế nhằm làm cho các thiết kế hướng đối tượng trở nên linh hoạt, dễ hiểu và dễ bảo trì hơn

S	Single responsibility principle
O	Open/closed principle
L	Liskov substitution principle
I	Interface segregation principle
D	Dependency inversion principle

SINGLE RESPONSIBILITY PRINCIPLE

Một lớp chỉ nên chịu trách nhiệm duy nhất một nhiệm vụ cụ thể

SINGLE RESPONSIBILITY PRINCIPLE

Một lớp chỉ nên chịu trách nhiệm duy nhất một nhiệm vụ cụ thể

Lợi ích:

- **Kiểm thử dễ dàng hơn:** một lớp có một trách nhiệm sẽ có ít trường hợp kiểm thử hơn
- **Liên kết dễ dàng hơn:** ít chức năng trong một lớp sẽ có ít phụ thuộc hơn
- **Tổ chức hơn:** các lớp nhỏ, được tổ chức tốt dễ tìm kiếm hơn so với lớp khổng lồ

SINGLE RESPONSIBILITY PRINCIPLE

```java
public class NhanVien {
  private String ten;
  private String chucVu;
  private double luong;

  // ...

  public void taoHoaDon(KhachHang khachHang, List<SanPham> sanPham) {
    HoaDon hoaDon = new HoaDon(khachHang, sanPham);
    // Tính toán tổng số tiền
    hoaDon.setTongSoTien(tinhToanTongSoTien(sanPham));
    // Lưu hóa đơn
    luuHoaDon(hoaDon);
  }
}
```

Vi phạm SRP vì có hai trách nhiệm:

- Quản lý thông tin nhân viên (tên, chức vụ, lương)
- Tạo hóa đơn

SINGLE RESPONSIBILITY PRINCIPLE

```
public class HoaDon {
    private KhachHang khachHang;
    private List<SanPham> sanPham;
    private double tongSoTien;

    // ...

    public void luuHoaDon() {
        // Lưu hóa đơn vào cơ sở dữ liệu
    }
}
```

Tuân theo SRP vì có một trách nhiệm duy nhất:

Quản lý thông tin hóa đơn (khách hàng, sản phẩm, số tiền)

```
public class Nhanvien
{
    private string Ten;
    private string ChucVu;
    private double Luong;
}
```

Bằng cách chia nhỏ lớp NhanVien thành hai lớp nhỏ hơn, chúng ta đã cải thiện tính mô đun, khả năng bảo trì và khả năng kiểm tra của mã.

```
public class Hoadon{
    private Nhanvien nhanvien;
    private KhachHang khachhang;
    private List<SanPham> sanpham;

    public void HoaDonDangKy(NhanVien nhanvien, KhachHang khachhang, List<SanPham> sanpham)
    {
        this.nhanvien = nhanvien;
        this.khachhang = khachhang;
        this.sanpham = sanpham
    }
    public void taoHoaDon()
    {
        Hoadon hd = new Hoadon(khachhang, sanpham);
        //Tính toán tổng số tiền
        hoadon.setTongSoTien(TinhToanTongSoTien(sanpham));
        //Lưu hóa đơn
        LuuHoaDon(hd);
    }
}
```

OPEN/CLOSED PRINCIPLE

Các classes, modules, functions, etc ... nên được mở để mở rộng (thêm mới chức năng) nhưng đóng lại để sửa đổi (không được phép sửa đổi mã nguồn gốc)

OPEN/CLOSED PRINCIPLE

Các classes, modules, functions, etc ... nên được mở để mở rộng (thêm mới chức năng) nhưng đóng lại để sửa đổi (không được phép sửa đổi mã nguồn gốc)

Lợi ích:

- **Dễ bảo trì:** việc sửa đổi mã hiện có có thể dẫn đến lỗi và khó khăn trong việc theo dõi thay đổi. OCP giúp giảm thiểu việc sửa đổi mã, do đó cải thiện tính bảo trì
- **Dễ mở rộng:** OCP cho phép dễ dàng thêm chức năng mới mà không ảnh hưởng đến mã hiện có. Điều này giúp phần mềm dễ dàng thích ứng với các yêu cầu mới
- **Tăng tính linh hoạt:** OCP giúp phần mềm linh hoạt hơn, dễ dàng thay đổi và tùy chỉnh

OPEN/CLOSED PRINCIPLE

```
public abstract class HinhHoc
{
    public abstract double TinhDienTich();
}
```

```
public class HinhVuong : HinhHoc
{
    private double canh;

    public HinhVuong(double canh)
    {
        this.canh = canh;
    }

    public override double TinhDienTich()
    {
        return canh * canh;
    }
}
```

Giả sử chúng ta muốn thêm một lớp mới **HinhTron** để tính diện tích hình tròn. Theo OCP, chúng ta có thể tạo một lớp HinhTron mới mà **không** cần sửa đổi các lớp HinhHoc hoặc HinhVuong hiện có.

OPEN/CLOSED PRINCIPLE

```
public abstract class HinhHoc
{
    public abstract double TinhDienTich();
}
```

```
public class HinhTron : HinhHoc
{
    private double banKinh;

    public HinhTron(double banKinh)
    {
        this.banKinh = banKinh;
    }

    public override double TinhDienTich()
    {
        return Math.PI * banKinh * banKinh;
    }
}
```

Bằng cách áp dụng OCP, chúng ta đã tạo ra một thiết kế **linh hoạt** và dễ **mở rộng**. Chúng ta có thể dễ dàng thêm các lớp mới để tính diện tích các hình dạng khác mà không cần sửa đổi mã hiện có.

LISKOV SUBSTITUTION PRINCIPLE

Các đối tượng của lớp con có thể thay thế các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình

LISKOV SUBSTITUTION PRINCIPLE

Các đối tượng của lớp con có thể thay thế các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình

Lợi ích:

- **Tăng tính linh hoạt:** LSP giúp cho phần mềm linh hoạt hơn, dễ dàng thay đổi và tùy chỉnh.
- **Giảm lỗi:** Việc thay thế các đối tượng của lớp cha bằng các đối tượng của lớp con không nên dẫn đến lỗi trong chương trình.
- **Tăng tính bảo trì:** LSP giúp tạo ra phần mềm dễ bảo trì hơn, do giảm thiểu việc sửa đổi mã hiện có.

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public virtual void SetHeight(int height)
    {
        this.Height = height;
    }
    public virtual void SetWidth(int width)
    {
        this.Width = width;
    }
    public virtual int CalculateArea()
    {
        return this.Height * this.Width;
    }
}

public class Square : Rectangle
{
    public override void SetHeight(int height)
    {
        this.Height = height;
    }

    public override void SetWidth(int width)
    {
        this.Width = width;
    }
}
```

```
Rectangle rect = new Rectangle();
rect.SetHeight(10);
rect.SetWidth(5);
System.Console.WriteLine(rect.CalculateArea()); // Kết quả là 5 * 10

// Below instantiation can be returned by some factory method
Rectangle rect1 = new Square();
rect1.SetHeight(10);
rect1.SetWidth(5);
System.Console.WriteLine(rect1.CalculateArea());
// Kết quả là 5 x 10. Nếu đúng phải là 5x5
```

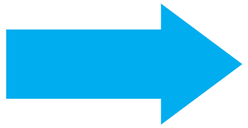
Do hình vuông có 2 cạnh bằng nhau, mỗi khi set độ dài 1 cạnh thì ta set luôn độ dài của cạnh còn lại. Tuy nhiên, khi chạy thử, hành động này đã thay đổi hành vi của của class Rectangle, dẫn đến vi phạm LSP.

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public virtual void SetHeight(int height)
    {
        this.Height = height;
    }
    public virtual void SetWidth(int width)
    {
        this.Width = width;
    }
    public virtual int CalculateArea()
    {
        return this.Height * this.Width;
    }
}

public class Square : Rectangle
{
    public override void SetHeight(int height)
    {
        this.Height = height;
    }

    public override void SetWidth(int width)
    {
        this.Width = width;
    }
}
```



Sửa:

```
public class Square : Rectangle
{
    public override void SetHeight(int height)
    {
        this.Height = height;
        this.Width = height;
    }

    public override void SetWidth(int width)
    {
        this.Height = width;
        this.Width = width;
    }
}
```

INTERFACE SEGREGATION PRINCIPLE

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

Lợi ích:

- **Rõ ràng và dễ hiểu**: Bằng cách chia nhỏ các giao diện thành các phần nhỏ hơn, mỗi phần phục vụ một mục đích cụ thể, code trở nên rõ ràng và dễ hiểu hơn.
- **Tính linh hoạt:** Điều này tăng tính linh hoạt của hệ thống, cho phép các sử dụng chỉ những phương thức cần thiết mà không bị ảnh hưởng bởi các phương thức không liên quan.
- **Dễ bảo trì và mở rộng**: Khi có nhu cầu thay đổi hoặc mở rộng chức năng, việc chỉnh sửa các giao diện nhỏ hơn thường dễ dàng hơn và an toàn hơn so với việc chỉnh sửa một giao diện lớn.
- **Tăng hiệu suất**: Với ISP, các khách hàng chỉ phải tải và triển khai các phương thức mà họ cần, giảm bớt việc tải những phần không cần thiết của giao diện. Điều này có thể tăng hiệu suất và tối ưu hóa tài nguyên của hệ thống.
- **Giảm rủi ro**: Chia nhỏ giao diện giúp giảm sự phụ thuộc vào các phương thức không cần thiết, giảm rủi ro khi có thay đổi hoặc bổ sung chức năng mới.

INTERFACE SEGREGATION PRINCIPLE

```
public interface IStudy
{
    void StudyEnglish();
    object StudyProgrammingLanguage();
}

public class NormalStudent : IStudy
{
    public void StudyEnglish()
    {

    }

    public object StudyProgrammingLanguage()
    {
        return null;
    }
}

public class InformationTechnologyStudent : IStudy
{
    public void StudyEnglish()
    {

    }

    public object StudyProgrammingLanguage()
    {
        return "Studying programming language for IT students";
    }
}
```

Nếu sau này muốn thêm môn học mới cho từng hệ sinh viên ?

Nếu viết chung vào interface Study thì phải phát sinh việc phải implement nhiều hàm không cần thiết.

=> Tách việc học thành phần học chung, phần học riêng

INTERFACE SEGREGATION PRINCIPLE

```
public interface IStudy
{
    void StudyEnglish();
    void StudyMath();
}

public interface IInformationTechnologyStudy
{
    void StudyProgrammingLanguage();
}

public interface IEconomicsStudy
{
    void StudyPhilosophy();
}
```

```
public class EconomicsStudy : IStudy, IEconomicsStudy
{
    public void StudyEnglish()
    {

    }

    public void StudyMath()
    {

    }

    public void StudyPhilosophy()
    {

    }
}

public class InformationTechnologyStudy : IStudy, IInformationTechnologyStudy
{
    public void StudyEnglish()
    {

    }

    public void StudyMath()
    {

    }

    public void StudyProgrammingLanguage()
    {

    }
}
```

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).

DEPENDENCY INVERSION PRINCIPLE

Lợi ích:

- **Giảm sự phụ thuộc:** DIP giảm sự phụ thuộc giữa các module trong hệ thống, đặc biệt là giữa các module cấp cao và cấp thấp. Thay vì trực tiếp phụ thuộc vào các module cụ thể, các module cấp cao chỉ phụ thuộc vào giao diện trừu tượng, giúp giảm sự ràng buộc và làm cho mã nguồn dễ dàng bảo trì và mở rộng.
- **Tính linh hoạt:** Tính linh hoạt: DIP tạo ra một lớp trung gian trừu tượng giữa các module cấp cao và cấp thấp. Điều này cho phép thay đổi hoặc thay thế các module cấp thấp mà không ảnh hưởng đến các module cấp cao, giúp tăng tính linh hoạt của hệ thống.
- **Dễ kiểm soát và thử nghiệm**: Bằng cách phân tách các module cấp cao và cấp thấp thông qua giao diện trừu tượng, DIP làm cho việc kiểm soát và thử nghiệm các module trở nên dễ dàng hơn. Bạn có thể dễ dàng thay thế các module cấp thấp bằng các bản giả lập hoặc mock trong các ca kiểm thử.
- **Tái sử dụng code:** DIP tạo điều kiện thuận lợi cho việc tái sử dụng code. Vì các module cấp cao không phụ thuộc vào các module cụ thể, chúng có thể tái sử dụng và triển khai lại trong các bối cảnh khác nhau mà không cần sửa đổi code cấp cao.
- **Tách biệt logic:** Bằng cách phân tách các module thành các lớp trừu tượng và cụ thể, DIP giú
tách biệt logic của các module, làm cho code trở nên dễ đọc và dễ hiểu hơn.

DEPENDENCY INVERSION PRINCIPLE

```
public class EmailSender
{
    public void SendEmail()
    {
        //Send email
    }
}

public class Notification
{
    private EmailSender _email;
    public Notification()
    {
        _email = new EmailSender();
    }

    public void Send()
    {
        _email.SendEmail();
    }
}
```

Nếu muốn gửi cả SMS và Email đến cho user thì làm thế nào ?

Nếu tạo lớp SMSSender và chỉnh sửa class Notification thì vi phạm một lúc hai nguyên tắc: Dependency Inversion và Open close principle.

=> phải Refactoring code theo chiều hướng giảm sự phụ thuộc cứng bằng cách tạo ra một interface ISender dùng chung giữa hai class EmailSender và SMSSender.

DEPENDENCY INVERSION PRINCIPLE

```
public class EmailSender : ISender
{
    public void Send(){
        //Send email
    }
}

public class SMSSender : ISender
{
    public void Send(){
        //Send SMS
    }
}
public class Notification
{
    private ICollection<ISender> _senders;
    public Notification(ICollection<ISender> senders){
        _senders = senders;
    }
    public void Send(){
        foreach (var message in _senders){
            message.Send();
        }
    }
}
```

```
public interface ISender
{
    void Send();
}
```

Giờ đây class Notificaiton phụ thuộc mềm vào interface ISender, nếu khách hàng yêu cầu thêm một phương thức để chuyển tin nhắn ta có thể thêm vào dễ dàng bằng cách sử dụng interface ISender.

Thank you!

DO YOU HAVE ANY QUESTIONS?