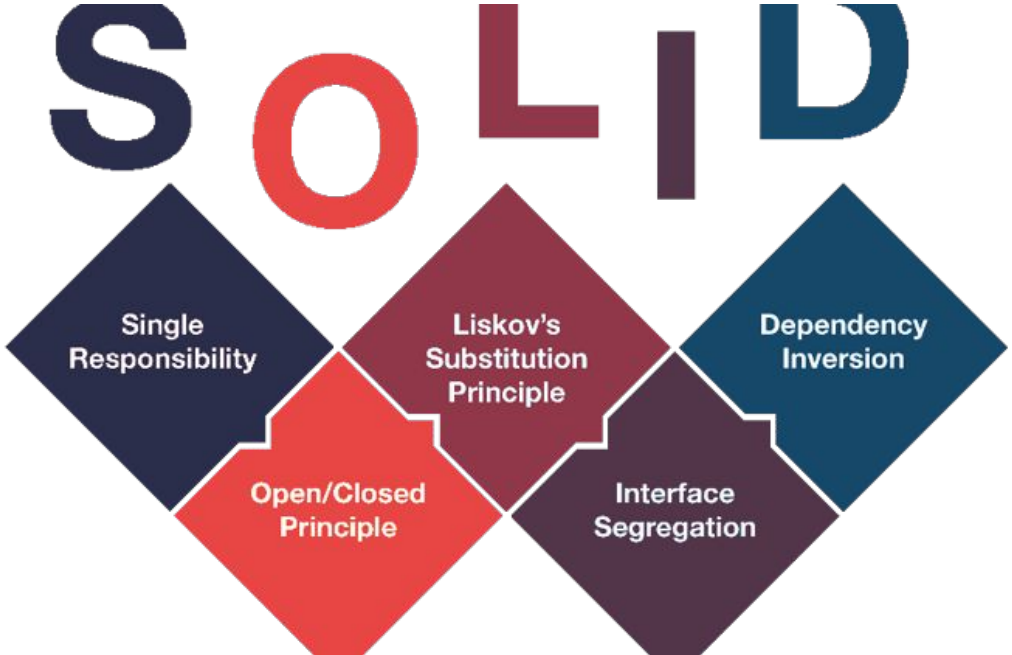
Bài 19: Các nguyên tắc thiết kế hướng đối tượng (SOLID)

Nguyễn Hoàng Anh

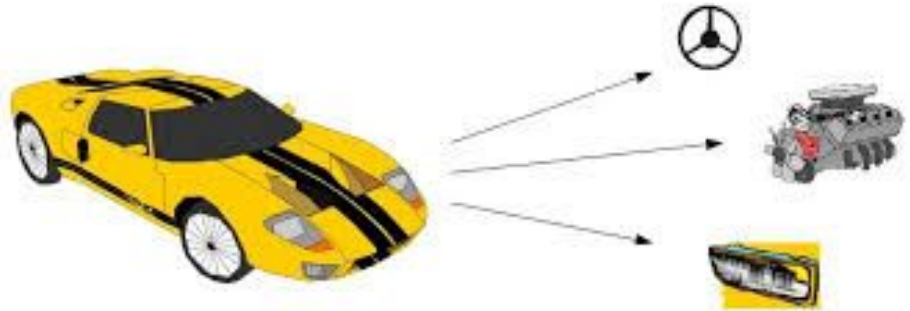
Giới thiệu

- **SOLID** là tập hợp 5 nguyên tắc **thiết kế** phần mềm giúp tạo ra mã nguồn **dễ bảo trì**, **mở rộng** và linh hoạt hơn.



Nguyên tắc đơn trách nhiệm

- **S - Single Responsibility Principle (SRP) - Nguyên tắc đơn trách nhiệm**
  - **Mỗi phần** của chương trình chỉ nên làm **một việc duy nhất**. Nếu một phần có nhiều nhiệm vụ, nó sẽ khó bảo trì và dễ gây lỗi khi thay đổi..
  - Giúp mã nguồn dễ bảo trì, dễ mở rộng và dễ kiểm thử hơn.



Nguyên tắc đơn trách nhiệm

```
class Sensor
{
    public:
        // Xử lý dữ liệu
        void processData(){}

        // Lưu trữ dữ liệu
        void saveData(){}

        // Gửi dữ liệu đi
        void sendData(){}
};
```

```
// Class xử lý dữ liệu
class Process
{
    public:
        void processData(){}
};

// Class lưu trữ dữ liệu
class Save
{
    public:
        void saveData(){}
};

// Class gửi dữ liệu
class Send
{
    public:
        void sendData(){}
};
```

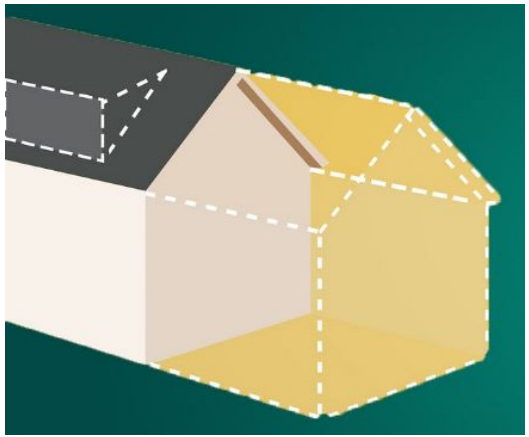
Nguyên tắc đơn trách nhiệm

```
class SpeedSensor
{
public:
    double getSpeed()
    {
        // đọc dữ liệu cảm biến
        return 80.5;
    }

    void controlEngine(double speed)
    {
        if (speed > 100){
            cout << "Giảm tốc độ động cơ" << endl;
        }
    }
};
```

```
#include <iostream>
using namespace std;

class SpeedSensor
{
    public:
        double getSpeed(){ return 40; }
};
class EngineController
{
    public:
        void adjustSpeed(double speed)
        {
            if (speed <= 40){
                cout << "Tăng tốc độ động cơ" << endl;
            } else if (speed >= 100){
                cout << "Giảm tốc độ động cơ" << endl;
            }
        }
};
int main()
{
    SpeedSensor speedSensor;
    double speed = speedSensor.getSpeed();
    EngineController engine;
    engine.adjustSpeed(speed);
    return 0;
}
```

Nguyên tắc đóng mở

- **O - Open/Closed Principle (OCP) - Nguyên tắc đóng mở**
  - Cho phép **mở rộng** nhưng **không sửa đổi mã cũ**. Khi cần thêm tính năng mới, hãy thêm mã mới thay vì chỉnh sửa mã hiện có.
  - Tránh sửa đổi mã nguồn cũ, giúp phần mềm ổn định hơn.



Nguyên tắc đóng mở

```
class SpeedAlert
{
    public:
        void sendAlert(double speed)
        {
            if (speed > 100)
            {
                cout  << "Cảnh báo: Vượt quá tốc độ!" << endl;
            }
        }
};
```

Ví dụ: có một hệ thống cảnh báo tốc độ, ban đầu chỉ hỗ trợ cảnh báo bằng âm thanh. Nếu muốn thêm cảnh báo bằng đèn LED, ta phải sửa code trong **sendAlert()**, vi phạm OCP.

Nguyên tắc đóng mở

```
class SpeedAlert
{
    public:
        void sendAlert(double speed)
        {
            if (speed > 100)
            {
                cout  << "Cảnh báo: Vượt
quá tốc độ!" << endl;
            }
        }
};
```

```
class Device_Alert{
    public:
        virtual void sendAlert(double speed) = 0;
};
```

```
class Sound_Alert : public Device_Alert{
    public:
        void sendAlert(double speed) override{
            if (speed > 100){
                cout << "Cảnh báo: Vượt quá tốc độ! (Âm
thanh)" << endl;
            }
        }
};
```

```
class LED_Alert : public Device_Alert{
    public:
        void sendAlert(double speed) override{
            if (speed > 100){
                cout << "Cảnh báo: Vượt quá tốc độ! (Đèn LED)"
<< endl;
            }
        }
};
```

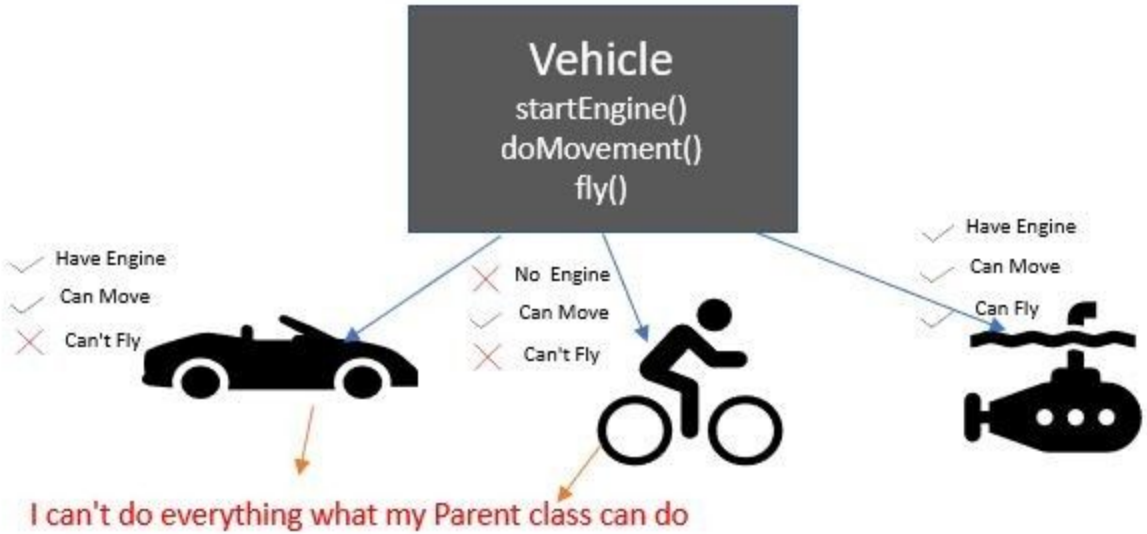
```
void handleSpeedAlert(Device_Alert *alert, double speed)
{
        alert->sendAlert(speed);
}
```

Nguyên tắc đóng mở

```
class Payment
{
    public:
        void processPayment(string type)
        {
            if (type == "CreditCard")
            {
                /* Xử lý thẻ tín dụng */
            }
            else if (type == "PayPal")
            {
                /* Xử lý PayPal */
            }
        }
};
```

```
class PaymentMethod
{
    public:
        virtual void pay() = 0;
};

class CreditCard : public PaymentMethod
{
    public:
        void pay() override { /* Xử lý thẻ tín dụng */ }
};

class PayPal : public PaymentMethod
{
    public:
        void pay() override { /* Xử lý PayPal */ }
};
```

Nguyên tắc thay thế Liskov



Nguyên tắc thay thế Liskov

- **L - Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov**
  - Một lớp con có thể thay thế lớp cha mà **không làm thay đổi tính đúng đắn** của chương trình**.**
  - Đảm bảo tính kế thừa không phá vỡ hành vi của hệ thống.

Nguyên tắc thay thế Liskov

```
class Rectangle{
    private:
        int width, height;

    public:
        virtual void setWidth(int w){
            width  = w;
        }

        virtual void setHeight(int h){
            height = h;
        }
};

class Square : public Rectangle{
    public:
        void setWidth(int w)  override{
            width = height = w;
        }

        void setHeight(int h) override{
            width = height = h;
        }
};
```

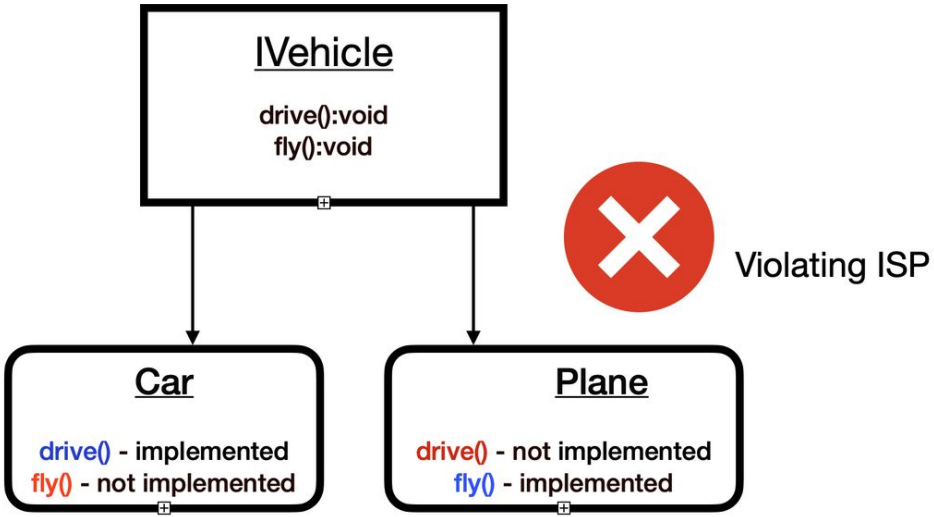
```
class Shape{
    public:
        virtual int getArea() = 0;
};

class Rectangle : public Shape {
    private:
        int width, height;

    public:
        void setDimensions(int w, int h){
            width = w; height = h;
        }

        int getArea() override{
            return width * height;
        }
};

class Square : public Shape {
    private:
        int side;

    public:
        void setSide(int s){
            side = s;
        }

        int getArea() override{
            return side * side;
        }
};
```

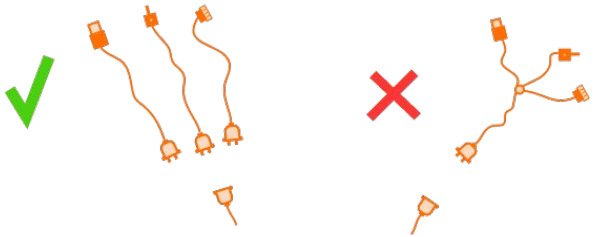
Nguyên tắc thay thế Liskov

```
class Car{
    public:
        virtual void refuel(){
            cout << "Đổ xăng" << endl;
        }
};

class ElectricCar : public Car{
    public:
        void refuel() override{
            // Xe điện không thể đổ xăng
            cout << "Xe điện không dùng xăng!" <<
endl;
        }
};
```

```
class Vehicle{
    public:
        virtual void rechargeOrRefuel() = 0;
};

class GasCar : public Vehicle{
    public:
        void rechargeOrRefuel() override{
            cout << "Đổ xăng" << endl;
        }
};

class ElectricCar : public Vehicle{
    public:
        void rechargeOrRefuel() override{
            cout << "Sạc pin" << endl;
        }
};
```

Nguyên tắc phân tách giao diện





Nguyên tắc phân tách giao diện

- **I - Interface Segregation Principle (ISP) - Nguyên tắc phân chia interface**
  - Một class không nên bị **ép buộc** triển khai những phương thức mà nó không sử dụng.
  - Tránh việc các class con phải cài đặt các phương thức không liên quan.

Nguyên tắc phân tách giao diện

```
class IVehicle
{
    public:
        virtual void drive() = 0;        // phương tiện có thể lái
        virtual void refuel() = 0;       // tiếp nhiên liệu
        virtual void loadCargo() = 0;    // chở hàng hóa
};

class Bike : public IVehicle
{
    public:
        void drive() override
        {
            cout << "Lái xe đạp" << endl;
        }

        void refuel() override {}        // Không hợp lý với xe đạp
        void loadCargo() override {}
}
```

Nguyên tắc phân tách giao diện

```
// Interface cho phương tiện có thể lái
class IDriveable
{
    public:
        virtual void drive() = 0;
};

// Interface cho phương tiện có thể lái
class IFuelable
{
    public:
        virtual void refuel() = 0;
};

// Interface cho phương tiện có thể lái
class ICargo
{
    public:
        virtual void loadCargo() = 0;
};
```

```
// Xe ô tô
class Car : public IDriveable, public IFuelable, public ICargo
{
public:
    void drive() override{ cout << "Lái ô tô" << endl; }

    void refuel() override{ cout << "Đổ xăng" << endl; }

    void loadCargo() override{ cout << "Chở hàng" << endl; }
};

// xe đạp
class Bike : public IDriveable, public ICargo
{
    public:
        void drive() override{ cout << "Lái xe đạp" << endl; }

        void loadCargo() override{ cout << "Chở hàng" << endl; }
};
```

Nguyên tắc đảo ngược phụ thuộc

- **D - Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược sự phụ thuộc**
  - Các phần quan trọng trong chương trình không nên dựa trực tiếp vào chi tiết cụ thể mà nên dựa vào một **giao diện chung**. Điều này giúp phần mềm linh hoạt hơn, dễ mở rộng và thay đổi mà không làm ảnh hưởng đến các phần khác.
  - Ví dụ: thay vì một công tắc chỉ bật được đèn, ta làm nó điều khiển bất kỳ thiết bị nào bằng cách dùng một giao diện chung.

Nguyên tắc đảo ngược phụ thuộc

```
class LightBulb
{
    public:
        void turnOn() { /* Bật đèn */ }
};

class Switch
{
    private:
        LightBulb bulb;

    public:
        void operate() { bulb.turnOn(); }
};
```

```
class Device{
    public:
        virtual void turnOn() = 0;
};

class LightBulb : public Device{
    public:
        void turnOn() override { /* Bật đèn */ }
};

class Fan : public Device{
    public:
        void turnOn() override { /* Bật quạt */ }
};

class Switch{
    private:
        Device *device;

    public:
        Switch(Device *d) : device(d){}
        void operate() { device->turnOn(); }
};
```

Nguyên tắc đảo ngược phụ thuộc

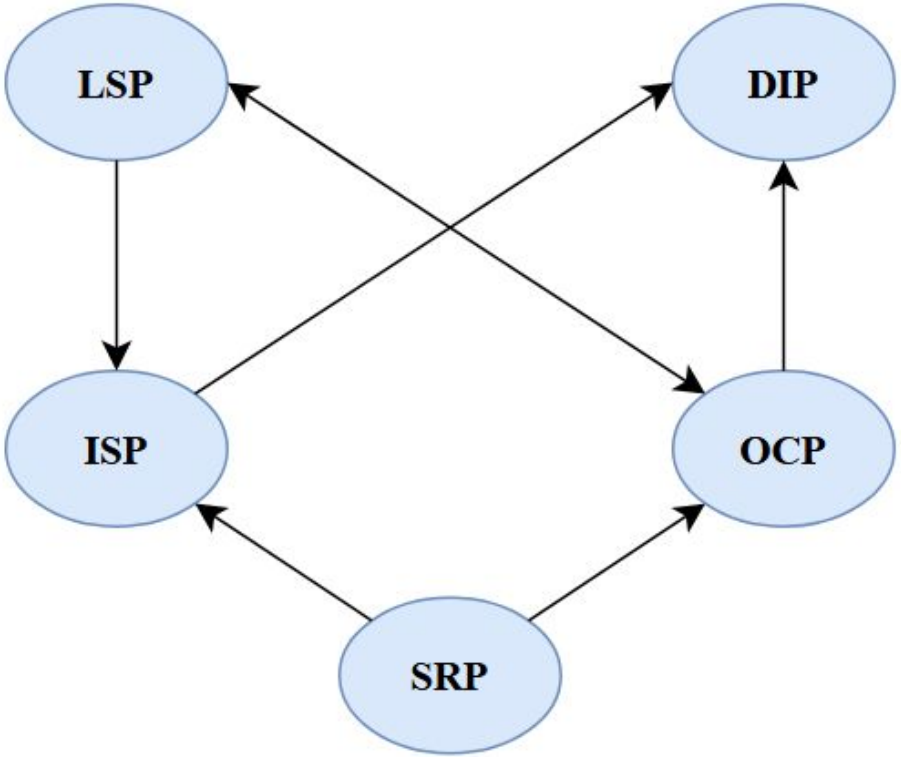
```
class SpeedSensor
{
    public:
        double getSpeed() { return 80.0; }
};

class SpeedController
{
    private:
        SpeedSensor sensor;

    public:
        void control()
        {
            if (sensor.getSpeed() > 100)
            {
                cout << "Giảm tốc độ" << endl;
            }
        }
};
```

```
class ISensor{
    public:
        virtual double getSpeed() = 0; // Interface cảm biến tốc độ
};

// Cảm biến tốc độ từ bánh xe
class SpeedSensor : public ISensor{
    public:
        double getSpeed() override { return 80.0; }
};

// Cảm biến tốc độ từ GPS
class GPSSpeedSensor : public ISensor{
    public:
        double getSpeed() override { return 85.5; }
};

class SpeedController{
    private:
        ISensor* sensor;

    public:
        SpeedController(ISensor* s) : sensor(s){}

        void control(){
            double speed = sensor->getSpeed();

            cout << "Tốc độ: " << speed << " km/h" << endl;
```

Tóm tắt

- SRP giúp chia nhỏ thành nhiều module.
- OCP giúp mở rộng mà không sửa đổi cái cũ.
- LSP đảm bảo lớp con kế thừa không phá vỡ chương trình.
- ISP giúp interface gọn gàng.
- DIP giúp kết nối các module qua abstraction.

Mối liên hệ



Ví dụ

Xây dựng ứng dụng quản lý siêu thị, nơi người quản lý có thể quản lý các sản phẩm với các tính năng cơ bản:

- Thêm sản phẩm
- Hiển thị danh sách sản phẩm
- Áp dụng giảm giá
- Lưu thông tin sản phẩm

Ví dụ

Ứng dụng được chia thành các phần chính:

1. **Class Product**: Lưu thông tin sản phẩm như ID, tên, giá.
2. **Interface IDiscount**: Cung cấp phương thức giảm giá.
3. **Interface IProductStorage, IShippable**: Quản lý lưu trữ sản phẩm và các sản phẩm có thể vận chuyển.
4. **Class MemoryStorage**: Lưu sản phẩm vào bộ nhớ RAM.
5. **Class PhysicalProduct** và **DigitalProduct**: Phân loại sản phẩm.

Ví dụ

🔹 **TẠO CLASS PRODUCT**

👉 **Mục tiêu:**

- Chỉ lưu thông tin sản phẩm, không chứa logic quản lý.

Ví dụ

🔹 **PHÂN LOẠI SẢN PHẨM ĐÚNG CÁCH**

👉 **Mục tiêu:**

- Khi có nhiều loại sản phẩm khác nhau, ta cần đảm bảo rằng việc mở rộng không làm hỏng hệ thống.
- Ví dụ, một sản phẩm kỹ thuật số (như eBook) không cần thuộc tính "kích thước" hay "cân nặng" như sản phẩm vật lý (như hộp sữa).
- Các sản phẩm mới thêm vào phải có thể thay thế sản phẩm cũ mà không gây lỗi.

Ví dụ

🔹 **THÊM GIẢM GIÁ CHO SẢN PHẨM**

👉 **Mục tiêu:**

- Dễ dàng thêm các loại giảm giá mà không cần sửa code cũ.

Ví dụ

🔹 **QUẢN LÝ LƯU TRỮ SẢN PHẨM**

👉 **Mục tiêu:**

- Tránh phụ thuộc trực tiếp vào kiểu lưu trữ.
- Có thể thay đổi cách lưu trữ (RAM, Database, File) dễ dàng.

Ví dụ

🔹 **TÁCH NHỎ INTERFACE**

👉 **Mục tiêu:**

- Không ép mọi sản phẩm phải có cùng tính năng.

Ví dụ

```cpp
#include <iostream>
#include <string>
using namespace std;

// Chứa thông tin sản phẩm
class Product
{
    protected:
        int id;
        string name;
        double price;

    public:
        Product(int id, string name, double price)
            : id(id), name(name), price(price){}

        virtual void display() const
        {
            cout << "ID: " << id << " - Name: " << name << " - Price: " << price << endl;
```