Nguyên tắc SOLID và ví dụ trong C#

Giới thiệu

Nếu bạn tìm kiếm Google với từ khóa ‘SOLID’, bạn sẽ thấy hàng tấn kết quả cả bài viết, video hay hướng dẫn về vấn đề này. Vậy nên mục đích của bài viết này mình sẽ hướng dẫn một cách đơn giản nhất có thể và đưa ra những ví dụ đơn giản nhưng hiệu quả. Các ví dụ này được viết bằng C#. Nhưng các bạn đừng lo, các ngôn ngữ hiện đại có cách viết gần tương tự nhau thôi.

Tổng quan về SOLID

SOLID là 5 nguyên tắc cơ bản, giúp xây dựng một kiến trúc phần mềm tốt. Bạn có thể thấy tất cả các design pattern đều dựa trên các nguyên tắc này. SOLID được ghép lại từ 5 chữ viết tắt đầu tiên của 5 nguyên tắc này:

1. **S** *is single responsibility principle (SRP)*
2. **O** *stands for open closed principle (OCP)*
3. **L** *Liskov substitution principle (LSP)*
4. **I** *interface segregation principle (ISP)*
5. **D** *Dependency injection principle (DIP)*

1. Single responsibility principle (SRP)

Mỗi một class chỉ nên đảm nhận một trọng trách, và chỉ nên có một lý do duy nhất để thay đổi. Để giải thích kỹ hơn mình xin lấy ví dụ. Bạn có một công cụ được kết hợp bởi rất nhiều các công cụ nhỏ khác nhau như dao, cắt móng tay, tuốc nơ vít….Bạn có muốn mua cái này? Tôi không nghĩ là muốn mua. Bởi vì có một vấn đề với nó, nếu bạn muốn thêm bất cứ một công cụ nào vào nó, bạn cần phải

thay đổi toàn bộ cấu tạo của nó, điều này là không ổn. Đây là một kiến trúc tồi với bất cứ hệ thống nào. Tốt hơn hết nếu bấm móng tay chỉ nên sử dụng để bấm móng tay, hoặc dao chỉ nên sử dụng để thái rau.

Sau đây là 1 ví dụ cho nguyên tắc này:

```
namespace SRP
{
   public class Employee
   {
      public int Employee_Id { get; set; }
      public string Employee_Name { get; set; }


      /// <summary>
      /// This method used to insert into employee table
      /// </summary>
      /// <param name="em">Employee object</param>
      /// <returns>Successfully inserted or not</returns>
      public bool InsertIntoEmployeeTable(Employee em)
      {
         // Insert into employee table.
         return true;
      }
```

```
        /// <summary>
        /// Method to generate report
        /// </summary>
        /// <param name="em"></param>
        public void GenerateReport(Employee em)
        {
            // Report generation with employee data using crystal report.
        }
    }
}
```

Class ‘Employee’ có 2 trách nhiệm, một là trách nhiệm thao tác với cơ sở dữ liệu và cái kia là tạo ra báo cáo. Lớp Employee không nên đảm nhận việc tạo ra báo cáo vì giả sử đến một ngày khách hàng yêu cầu phải tạo ra báo cáo trong Excel hoặc bất cứ định dạng nào khác, class này lại phải thay đổi cho phù hợp. Điều này là không tốt.

Vì thế để tuân theo SRP, một class chỉ nên đảm nhận một trách nhiệm, chúng ta nên viết sang một class khác cho việc tạo báo cáo, vậy khi có bất cứ sự thay đổi nào với việc tạo báo cáo, sẽ không ảnh hưởng đến class Employee.

```
public class ReportGeneration
{
    /// <summary>
    /// Method to generate report
```

```
    /// </summary>
    /// <param name="em"></param>
    public void GenerateReport(Employee em)
    {
        // Report reneration with employee data.
    }
}
```

2. Open closed principle (OCP)

Giờ chúng ta sẽ xem xét class ‘ReportGeneration’ nhưng một ví dụ cho nguyên tắc thứ 2. Bạn có đoán ra được vấn đề ở class này không?

```
public class ReportGeneration
{
    /// <summary>
    /// Report type
    /// </summary>
    public string ReportType { get; set; }

    /// <summary>
    /// Method to generate report
    /// </summary>
```

```
   /// <param name="em"></param>
   public void GenerateReport(Employee em)
   {
      if (ReportType == "CRS")
      {
          // Report generation with employee data in Crystal Report.
      }
      if (ReportType == "PDF")
      {
         // Report generation with employee data in PDF.
      }
    }
}
```

Tuyệt!!! Bạn đã đúng, quá nhiều mệnh đề IF và nếu bạn muốn thêm một loại report khác ví dụ như Excel, bạn cần viết thêm 1 lần if nữa. Class này nên khuyến khích mở rộng nhưng phải tránh việc chỉnh sửa. Làm sao để làm được điều này?

```
public class IReportGeneration
   {
      /// <summary>
      /// Method to generate report
```

```
    /// </summary>
    /// <param name="em"></param>
    public virtual void GenerateReport(Employee em)
    {
        // From base
    }
}
/// <summary>
/// Class to generate Crystal report
/// </summary>
public class CrystalReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate crystal report.
    }
}
/// <summary>
/// Class to generate PDF report
/// </summary>
```

```
public class PDFReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate PDF report.
    }
}
```

Nếu bạn muốn đưa ra một định dạng báo cáo khác, bạn chỉ cần kế thừa từ interface IReportGeneration. Vì IReportGeneration là interface nên nó chưa triển khai chi tiết method, nó sẽ giúp bạn giải quyết việc này.

3. Liskov substitution principle (LSP)

Nguyên tắc này đơn giản nhưng rất khó để hiểu. *Class con không nên phá vỡ các định nghĩa và hành vi của class cha*. Điều này có nghĩa là gì? Chúng ta lại lấy ví dụ với Employee để giúp bạn hiểu về nguyên tắc này. Bạn hãy xem hình bên dưới. Employee là lớp cha của Casual và Contractual. Hai class này kết thừa từ Employee.

Bạn hãy xem code:

```
public abstract class Employee
{
    public virtual string GetProjectDetails(int employeeId)
    {
        return "Base Project";
    }
    public virtual string GetEmployeeDetails(int employeeId)
    {
        return "Base Employee";
    }
}
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
// May be for contractual employee we do not need to store the details into
database.
```

```
    public override string GetEmployeeDetails(int employeeId)
    {
        return "Child Employee";
    }
}
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
    // May be for contractual employee we do not need to store the details into datab
ase.
    public override string GetEmployeeDetails(int employeeId)
    {
        throw new NotImplementedException();
    }
}
```

Có ổn không? Hãy xem đoạn có dưới đây và nó đã vi phạm nguyên tắc này:

```
List<Employee> employeeList = new List<Employee>();
```

```
employeeList.Add(new ContractualEmployee());
employeeList.Add(new CasualEmployee());
foreach (Employee e in employeeList)
{
   e.GetEmployeeDetails(1245);
}
```

Giờ Tôi đoán bạn đã hiểu vấn đề. Vâng, với Contractual employee, bạn sẽ ăn một exception khi method GetEmployeeDetails(int employeeId) chưa được triển khai, và điều vày vi phạm LSP. Vậy giải pháp là gì? Tách chúng ra thành 2 interface khác nhau. Một là Iproject, hai là Iemployee và triển khai theo từng type khác nhau:

```
public interface IEmployee
{
   string GetEmployeeDetails(int employeeId);
}

public interface IProject
{
   string GetProjectDetails(int employeeId);
}
```

Giờ contractual employee sẽ triển khai IEmployee nhưng không có IProject. Điều này giúp tuân theo nguyên tắc LSP.

4. Interface segregation principle (ISP)

Nguyên tắc này nói rằng bất cứ một client nào không nên triển khai một interface không phù hợp với nó. Điều này có nghĩa, giả sử có một CSDL để lưu trữ tất cả các loại của nhân viên (cố định, tạm thời), vậy cách tiếp cận tốt nhất là gì?

```
public interface IEmployee
{
   bool AddEmployeeDetails();
}
```

Tất cả các class Employee sẽ kế thừa từ interface này để lưu dữ liệu? Điều này ổn không? Bây giờ bạn hãy giả sử công ty một nào nào đó sẽ nói cho bạn rằng họ muốn lấy ra chỉ những nhân viên cố định. Bạn sẽ làm gì? Thêm phương thức vào interface?

```
public interface IEmployeeDatabase
{
   bool AddEmployeeDetails();
   bool ShowEmployeeDetails(int employeeId);
}
```

Nhưng chúng ta sẽ phá vỡ một số thứ. Chúng ta đang tập trung vào class nhân viên không cố định để hiển thị chi tiết của họ từ CSDL. Vậy giải pháp đưa ra là sẽ đưa chúng ra một interface khác.

```
public interface IAddOperation
{
   bool AddEmployeeDetails();
}
public interface IGetOperation
{
   bool ShowEmployeeDetails(int employeeId);
}
```

Và các nhân viên không cố định sẽ chỉ triển khai interface IAddOperation và các nhân viên cố định sẽ triển khai cả 2 interface.

5. Dependency inversion principle (DIP)

Nguyên tắc này nói cho bạn rằng bạn không nên viết code gắn chặt với nhau bởi vì sẽ là cơn ác mộng cho việc bảo trì khi ứn dụng trở lên lớn dần. Nếu một class phụ thuộc một class khác, bạn sẽ cần phải thay đổi class đó nếu một trong những class phụ thuộc phải thay đổi. Chúng ta nên cố gắng viết các class ít phụ thuộc nhất có thể.

Giả sử chúng ta có một hệ thống thông báo sau khi lưu vài thông tin vào DB.

```
public class Email
{
    public void SendEmail()
    {
        // code to send mail
    }
}


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


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

Giờ class Notification hoàn toàn phụ thuộc vào class Email, vì nó chỉ gửi một loại của thông báo. Nếu bạn muốn thêm một cách thông báo mới như SMS chẳng hạn? Chúng ta cũng phải thay đổi cả hệ thống thông báo? Đó gọi là liên kết chặt (tightly coupled). Bạn có thể làm gì để giúp nó giảm phụ thuộc vào nhau. OK, bạn xem ví dụ sau đây:

```
public interface IMessenger{
   void SendMessage();
}
public class Email : IMessenger{
   public void SendMessage()
   {
      // code to send email
   }
}
public class SMS : IMessenger{
   public void SendMessage(){
      // code to send SMS
   }
}
public class Notification{
   private IMessenger _iMessenger;
```

```
    public Notification(){
        _ iMessenger = new Email();
    }
    public void DoNotify(){
        _ iMessenger.SendMessage();
    }
}
```

Class Notification vẫn phụ thuộc vào Email class. Nhưng giờ chúng ta sử dụng depedency Injection để làm cho chúng giảm sự phụ thuộc. Có 3 loại DI, Contructor Injection, Property Injection và Method Injection.

Constructor Injection

```
public class Notification{
    private IMessenger _iMessenger;
    public Notification(Imessenger pMessenger){
        _ iMessenger = pMessenger;
    }
    public void DoNotify(){
        _ iMessenger.SendMessage();
    }
}
```

Property Injection

```
public class Notification{
   private IMessenger _iMessenger;
   public Notification(){
   }
   public IMessenger MessageService{
      private get;
      set
     {
          _ iMessenger = value;
      }
    }
   public void DoNotify(){
       _ iMessenger.SendMessage();
   }
}
```

Method Injection

```
public class Notification{
   public void DoNotify(IMessenger pMessenger){
      pMessenger.SendMessage();
   }
}
```

Vậy SOLID sẽ giúp chúng ta viết code độc lập giảm sự phụ thuộc giữa các module, giúp nâng cao hiệu quả trong việc bảo trì, tránh nhiều rủi ro hơn.

Nguồn: https://tedu.com.vn/design-pattern/nguyen-tac-solid-va-vi-du-trong-c-49.html