**Programing Principles**

Trong lập trình thì cách viết code nên được rõ ràng để dễ bảo trì, kiểm tra, sau đây là một số nguyên tắc lập trình giúp hoàn thiện việc code.

# **ĐỊnh nghĩa**

## **SOLID**

SOLID là từ viết tắt của năm nguyên tắc thiết kế OOD

1. Single responsibility principle:

Một class chỉ nên giữ 1 trách nhiệm duy nhất (chỉ có thể sửa đổi class với 1 lý do duy nhất). Nếu xuất hiện đoạn code mà không thuộc về trách nhiệm của lớp thì nên tách đoạn code đó ra một xử lý khác.

Text

Description automatically generated

*Hình 1: Đoạn code chưa áp dụng SRP*

*Text

Description automatically generated*

*Hình 2: Đoạn code sau khi áp dụng SRP*

1. Open/ Closed Principle:

Có thể thoải mái mở rộng 1 class nhưng không được sửa đổi bên trong class đó. Theo nguyên lý này thì mỗi khi muốn thêm chức năng, cho chương trình chúng ta nên viết class mới mở rộng class cũ bằng cách kế thừa class cũ, không nên sửa đổi class cũ.

Text

Description automatically generated

*Hình 3: Đoạn code chưa áp dụng nguyên tắc*

Quá nhiều mệnh đề IF và nếu muốn thêm một loại report khác ví dụ như Excel, thì cần viết thêm 1 lần if nữa, như vậy đã vi phạm vào nguyên tắc. Để 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.

Text

Description automatically generated

*Hình 4: Áp dụng nguyên tắc*

1. Liskov Substitution Principle:

Trong một chương trình, các object của class con có thể thay thế class mà không làm thay đổi tính đúng đắn của chương trình. Hiểu đơn giản là lớp con khi override các method lớp cha thì không nên break funtionality của lớp cha vì những thay đổi từ design hay từ khách hàng.

Text

Description automatically generated

*Hình 5: Code vi phạm LSP*

Contractual employee, sẽ bị exception khi method GetEmployeeDetails(int employeeId) chưa được triển khai, và điều này vi phạm LSP. Để giải quyết vấn đề này thì tách chúng ra thành 2 interface khác nhau. Một là Iproject, hai là Iemployee và triển khai theo từng type khác nhau:

Text

Description automatically generated

*Hình 6: Tách ra 2 Interface*

1. Interface segregation principle:

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể, nếu ta tạo ra 1 interface bự (hơn 100 method chẳng hạn), mỗi class sẽ phải implement toàn bộ 100 method đó, kể những method không bao giờ sử dụng đến. Nếu áp dụng chia interface này ra thành nhiều interface nhỏ, các class chỉ cần implement những interface có chức năng mà chúng cần, không cần phải implement những chức năng thừa nữa.

Text

Description automatically generated

*Hình 7:Chưa áp dụng nguyên tắc ISP*

Ví dụ động vật nào cũng có thể ăn, uống, ngủ, ta thiết kế interface IAnimal. Khi ta muốn thêm 1 số loài động vật mới và tính năng vào, ta phải thêm có thêm method vào trong interface như: bơi lội, bay, săn mồi, … Điều này làm interface phình to ra. Khi một loài động vật kế thừa interface, nó phải implement luôn cả những hàm không dùng đến. Sau khi tách interface IAnimal ra thành các interface nhỏ

Text

Description automatically generated

*Hình 8: Sau khi áp dụng ISP*

1. Dependency Inversion Principle:

Diagram

Description automatically generated

Một hệ thống với các module/class ràng buộc với nhau chặt chẽ như trên, sẽ rất dễ dẫn đến "sụp đổ". Nguyên tắc DIP gồm 2 phần:

* Các module cấp cao không nên phụ thuộc vào các module 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, không phải thông qua implementation).

Nói đơn giản là không nên viết code gắn chặt với nhau bởi vì sẽ rất khó cho việc bảo trì khi ứng dụng trở lên lớn dần. Nếu một class phụ thuộc một class khác, sẽ cần phải thay đổi class đó nếu một trong những class phụ thuộc phải thay đổi. Nên cố gắng viết các class ít phụ thuộc nhất có thể.

Để ít phụ thuộc nhất có thể, sử dụng Depedency Injection (DI), có 3 loại DI là Contructor Injection, Property Injection và Method Injection.

Lấy ví dụ về ổ cứng của máy tính, có thể dùng loại ổ cứng thể rắn SSD đời mới để chạy cho nhanh, tuy nhiên cũng có thể dùng ổ đĩa quay HDD thông thường. Nhà sản xuất Mainboard không thể nào biết người dùng sẽ dùng ổ SSD hay loại HDD đĩa quay thông thường. Tuy nhiên họ sẽ luôn đảm bảo rằng người dùng có thể dùng bất cứ thứ gì họ muốn, miễn là ổ đĩa cứng đó phải có chuẩn giao tiếp SATA để có thể gắn được vào bo mạch chủ. Ở đây chuẩn giao tiếp SATA chính là interface, còn SSD hay HDD đĩa quay là implementation cụ thể.

## **KISS**

KISS là viết tắt của câu “Keep It Short and Simple” hoặc “Keep It Simple and Straightforward” hoặc “Keep It Small and Simple”. Chung quy lại nguyên tắc KISS có hàm ý là hướng về sự đơn giản và rõ ràng trong mọi vấn đề. Trong lập trình KISS nghĩa là làm cho code trở nên đơn giản và dễ nhìn. Số lượng dòng code của một class hay method ở con số hàng chục, ngắn gọn nhất có thể. Không viết class và method theo kiểu tổng hợp hay tất cả trong một.

Text

Description automatically generated

*Hình 9: Code chưa sử dụng nguyên tắc KISS*

*Text

Description automatically generated*

*Hình 10: Đoạn code sau khi áp dụng KISS*

## **YAGNI**

Nguyên tắc YAGNI là viết tắt của “You Aren’t Gonna Need It”, ngắn gọn là “đừng tự vẽ việc cho mình”. Hàm ý trong lập trình không nên thêm một số chức năng mới mà bản thân người lập trình nghĩ là cần thiết, có thể sẽ dùng trong tương lai. Không nên lãng phí thời gian cho việc đó vì chức năng mà người lập trình thêm mới đó có thể không hài lòng khách hàng và cấp trên hoặc thậm chí sẽ chẳng bao giờ dùng đến. Tập trung vào các task hiện tại thay vì đi làm một cái mới mà chưa được sự đồng ý của cấp trên và khách hàng hơn nữa là chưa chắc sẽ dùng đến chức năng đó.

## **DRY**

Nguyên tắc DRY là viết tắt của “Don’t Repeat Yourself” có nghĩa nếu chúng ta đang muốn viết nhiều đoạn code giống nhau ở nhiều chỗ khác nhau, thay vì copy và paste đoạn code đó, thì hãy đưa đoạn code đó vào một phương thức riêng sau đó gọi phương thức này từ những chỗ chúng ta cần gọi. Nguyên tắc này giống như tính chất kế thừa trong lập tình hướng đối tượng.

Ví dụ có một đối tượng là Calculator, đối tượng này có 2 nhiệm vụ chính là tính tổng và tính trung bình cộng của 2 số a và b. Trường hợp nếu muốn in ra tổng của 2 số mỗi khi thực hiện 2 phép tính trong đối tượng Calculator, phải sửa lại 2 phương thức này.

A screenshot of a computer

Description automatically generated with medium confidence

*Hình 11: Đoạn code chưa áp dụng DRY*

Mỗi khi muốn chỉnh sửa gì đó liên quan đến những đoạn code giống nhau, phải chỉnh sửa từng chỗ một.

A screenshot of a computer

Description automatically generated with medium confidence

*Hình 12: Đoạn code áp dụng DRY*

Để in ra tổng của 2 số, chỉ cần sửa phương thức total() là xong

# **Ưu và nhược điểm**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **SOLID** | **KISS** | **YAGNI** | **DRY** |
| **Ưu điểm** | * Áp dụng trong lập trình OOP. * Áp dụng đúng quy tắc sẽ đạt được 3 yếu tố: understandable, maintainable, flexible. * 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. * Khả năng tái cấu trúc linh hoạt. | * Các vấn đề được giải quyết nhanh hơn, tránh được các issue phức tạp. * Code dễ sử dụng, dễ test * Và dễ dàng bảo trì, nâng cấp code khi cần | * Người lập trình tập trung vào chính những task của mình, không miên man làm mất thời gian cũng như những rủi ro đến từ việc suy nghĩ rằng “sẽ có ích trong tương lai” | * Code ít hơn, giúp tiết kiệm thời gian, công sức, dễ bảo trì và cũng giảm nguy cơ gây lỗi. |
| **Nhược điểm** | * Chỉ nên sử dụng cho những hệ thống lớn. * Sử dụng nhiều Interface hơn. * Mất nhiều thời gian để phát triển và làm tăng chi phí dự án. | * Khi kết hợp với nguyên tắc YAGNI thì khả năng người lập trình dễ bị thụ động trong việc phát triển phần mềm | * Đối lập với SOLID, dễ dẫn đến lối suy nghĩ tính năng vừa đủ, kiến trúc tối giản hết mức: Không cần flexible, không dự trù vấn đề busniess sẽ phát triển, yêu cầu đến đâu thì kiến trúc đến đó. | * Trường hợp đoạn mã phức tạp, và được sử dụng lại nhiều nơi thì khi sử dụng nguyên tắc DRY sẽ dễ bị phạm phải lỗi nợ kỹ thuật (Technical Debt) |

# **Trường hợp nên sử dụng nguyên tắc nào**

|  |  |  |  |
| --- | --- | --- | --- |
| **SOLID** | **KISS** | **YAGNI** | **DRY** |
| Sử dụng phổ biến trong lập trình OOP, phối hợp với 4 đặc tính của lập trình OOP để tăng hiệu quả của ứng dụng | * Áp dụng để đặt tên hàm, biến, method, class, comment rõ nghĩa. * Chia task ra thành các task nhỏ hơn để code. * Giới hạn mỗi method chỉ nên từ 30-50 lines, và mỗi method chỉ nên thực hiện 1 chức năng. * Không nên sử dụng nhiều câu lệnh If trong 1 method, hãy tách nó ra thành các method khác nhau. | * Không có gì sai với tư tưởng "thực dụng" khi áp dụng YAGNI trong công việc tuy nhiên Không cần xây dựng sớm các tính năng giả định, thay vào đó, hãy luôn chuẩn bị tinh thần cho mọi sự thay đổi bằng cách làm source code linh hoạt và trừu tượng hơn | * Khi phải thay đổi function/class, liệu bạn có muốn tất cả những chỗ chúng được dùng cùng cập nhật? * Hai đoạn code này giống nhau có phải là trùng hợp không? Có khả năng chúng không như vậy trong tương lai? |

# **kết luận**

Dù áp dụng nguyên tắc nào đi nữa thì mục đích chung vẫn là giúp cho chương trình hoạt động với hiệu suất tốt, đoạn code dễ nhìn, dễ hiểu. Trường hợp hệ thống phát triển, mở rộng và bảo trì thì cũng trở nên đơn giản hơn, tránh rối loạn.