**SOLID PRINCIPLES LÀ GÌ?**

Hầu như mỗi chúng ta khi đến với lập trình đều đã từng nghe đến khái niệm OOP (Object-Oriented Programming). Và các khái niệm, tính chất cơ bản của OOP xuất hiện ở mọi nơi khi nhắc đến OOP đó là: Abstraction, encapsulation, inheritance, polymorphism. Nếu các bạn chưa nắm được các tính chất này, hãy tìm hiểu trước khi đọc bài viết này vì SOLID là từ viết tắt của 5 nguyên lý thiết kế trong OOP. Những nguyên lý này được đúc kết, rút ra từ những thành công và thất bại của hàng ngàn dự án. Hiểu rõ và áp dụng được các nguyên lý này vào code sẽ giúp bạn cải thiện khả năng tư duy và viết code của bản thân.

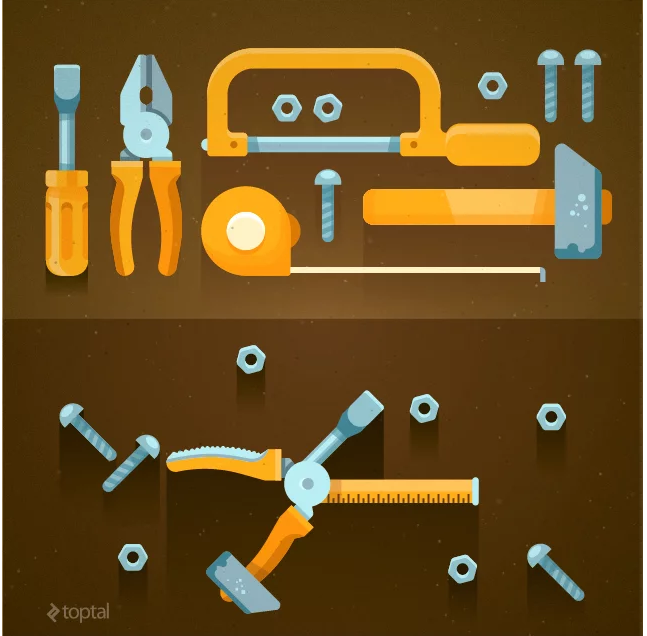
## **1. Single Responsibility Principle**

Nguyên lý đầu tiên tương ứng với chữ S trong SOLID. Nguyên lý này được phát biểu rằng:

Chỉ có thể sửa đổi class với một lý do duy nhất – Điều này có nghĩa rằng mỗi class chỉ nên giữ một trách nhiệm duy nhất

**Giải thích nguyên lý:**

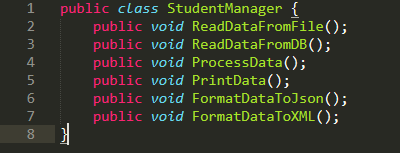
Nguyên lý trên nói rằng mỗi class chỉ nên giữ một trách nhiệm hay chức năng duy nhất, điều này không có nghĩa class đó chỉ có một method duy nhất mà là các method nằm trong class đó phải liên quan trực tiếp tới chức năng của class đó.



Trước tiên, hãy nhìn vào phần trên của tấm hình, ta thấy các vụng như thước đo, kềm, búa, cưa,… Mỗi vật dụng đều có một chức năng riêng biệt, vật dụng nào bị hỏng ta có thể sửa vật dụng đó mà không gây ảnh hưởng gì đến các vật dụng khác.

Giờ đến phần dưới của bức hình, ta có thể thấy kềm, tua-vít, thướt và búa đang được tích hợp lại thành một vật dụng, điều này ban đầu nhìn có vẻ ổn vì nó tiện lợi cho việc sử dụng vì nó có nhiều chức năng thay vì phải có 4, 5 vật dụng như phần trên. Nhưng vấn đề ở đây là nếu như một trong các chức năng của vật dụng tích hợp này bị hỏng thì ta sẽ phải tháo gỡ ra tất cả những thứ được tích hợp và, điều này dẫn đến việc khó sửa chữa và một chức năng hư có thể sẽ kéo theo những chức năng khác không hoạt động được.

Ví dụ trên liên quan trực tiếp đến việc thiết kế code của chúng ta, chúng ta cùng xem qua 1 class vi phạm nguyên lý



Class StudentManager giữ tới 4 trách nhiệm là: Đọc dữ liệu từ database, xử lý, format và xuất ra dữ liệu. Do đó, lỡ như thay đổi cách đọc dữ liệu hay format dữ liệu thì sẽ phải sửa đổi class này. Chưa kể đến việc khi có thêm nhiều chức năng class này sẽ bị phình to ra, các đoạn code đọc dữ liệu hay xuất ra dữ liệu sẽ nằm rải rác trong class, rất khó để bảo trì và nâng cấp.

Để giải quyết vấn đề này, ta chỉ cần tách ra làm nhiều class, mỗi class có một chức năng riêng. Chẳng hạn ở ví dụ trên ta sẽ tách ra thành 4 class là: DataReader, DataFormater, DataProcessor, DataPrinter. Tuy có nhiều class hơn nhưng việc bảo trì, nâng cấp sẽ đơn giản hơn.

## **2. Open-closed principle**

Đây là nguyên lý thứ hai, tương ứng với chữ O trong SOLID. Phát biểu nguyên lý này như sau:

Một class có thể được thoải mái mở rộng, nhưng không được sửa đổi bên trong nó.

**Giải thích nguyên lý:**

Theo nguyên lý này, một class cần đáp ưng được hai yêu cầu sau:

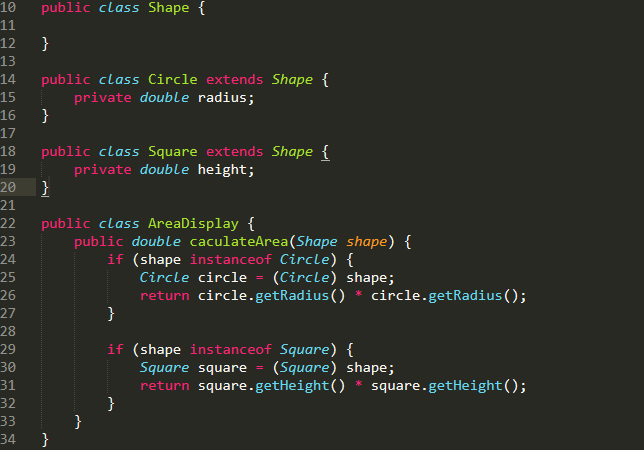
* Dễ mở rộng: Có thể dễ dàng nâng cấp, thêm tính năng khi có yêu cầu
* Không được sửa đổi: Không được hoặc hạn chế tối thiểu việc sửa đổi các module có sẵn.

Hai thuộc tính trên đi chung với nhau nghe có vẽ vô lý nhưng trên thực tế, rất nhiều phần mềm đã được thiết kế theo nguyên tắc này. Không nói đâu xa đó chính là ổ cắm điện mà nhà nào cũng có.

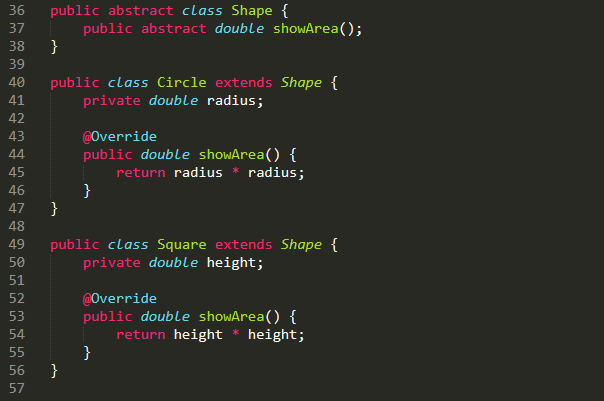


Có thể thấy, một chiếc ổ cắm điện thiết kế ra không cho phép người dùng có thể sửa đổi nó, bạn muốn cắm một phích cắm có 3 chân vào một ổ cắm chỉ cho phép 2 chân là điều không thể, bạn muốn thêm một ô cắm cho môt ổ cắm điện là hoàn toàn bất khả thi. Nhưng ổ cắm điện này đã được thiết kế cho phép việc chúng ta có thể mở rộng. Chúng ta có thể dùng một bổ chuyển đổi để cắm phích cắm có 3 chân vào ổ cắm chỉ cho phép phích cắm có 2 chân, chúng ta có thể có nhiều hơn số ô cắm được thiết kế sẵn bằng cách cắm thêm một ổ cắm khác vào.

Giờ chúng ta hãy cùng tìm hiểu nguyên lý ngày trong lập trình.



Đoạn code trên đã vị phạm nguyên lý, tại sao? Giả sử giờ chúng ta muốn thêm class Triangle, vậy để tính diện tích của hình tam giác, ta phải thêm một hàm if để kiểm tra và ép kiểu về Triangle. Do đó, áp dụng nguyên lý Open-closed, chúng ta sẽ cải tiến đoạn code này lại như sau:



Dễ dàng nhìn ra ở đoạn code trên, khi ta thêm vào class Triangle, ta chỉ cần kế thừa lại class Shape và viết lại phương thức tính diện tích, điều này không làm ảnh hưởng đến bất kì class nào.

## **3. Liskov Substitution Principle**

Nguyên lý thứ ba, tương ứng với chữ L trong SOLID. Nguyên lý này nói rằng:

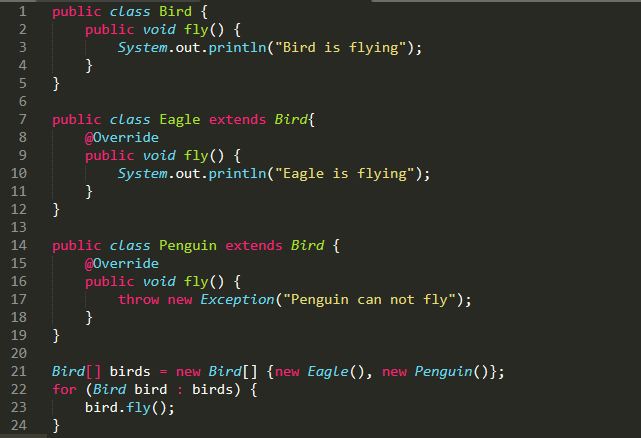
Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắng của chương trình.



**Giải thích nguyên lý:**

Mình sẽ lấy một ví dụ về nguyên lý này để chúng ta dễ hình dung. Hãy tưởng tượng chúng ta đang có một class Chó là class cha và class này có hành động là BảoVệNhà. Các class ChóPull, ChóNhật, ChóXù khi kế thừa class Chó sẽ kế thừa luôn hành động BảoVệNhà và không có vấn đề gì xảy ra. Tuy nhiên nếu ta có thêm class ChóĐồChơi và class này cũng kế thừa class Chó. Và rồi một ngày kia chẳng may ChóPull bị anh Exciter cướp đi mất, ta đem ChóĐồChơi vào thay thế hành động BảoVệNhà của ChóPull. Và tất nhiên, chương trình sẽ xảy ra lỗi vì ChóĐồChơi thì làm sao mà BảoVệNhà được.

Giờ hãy cùng nhìn qua một ví dụ khác về sự vi phạm nguyên lý này trong lập trình:



Vậy với đoạn code trên, ta thấy khi method fly() của class Penguin được gọi sẽ quăng ra Exception vì class Penguin không thay thế được class cha của nó là Bird.

Đây là nguyên lý rất dễ vi phạm. Trong đời sống, chim cánh cụt là chim nhưng không có nghĩa là chim cánh cụt nên kế thừa chim trong lập trình. Vậy nên chỉ nên cho class A kế thừa class B khi class A thay thế được cho class B.

## **4. Interface Segregation Principle**

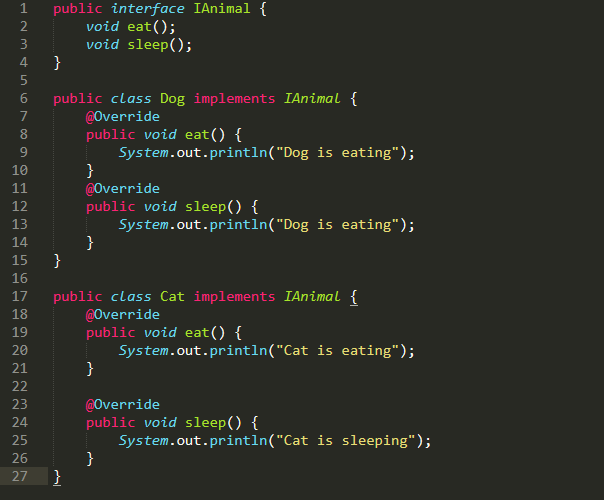
Nguyên lý thứ tư, tương ứng với chữ I trong SOLID. Nguyên lý này quy định:

Không nên dùng một interface lớn mà nên tách ra thành các interface nhỏ với nhiều mục đích cụ thể.

**Giải thích nguyên lý:**

Nguyên lý trên có vẻ dễ hiểu hơn so với các nguyên lý khác nếu như bạn đã hiểu rõ khái niệm interface. Mình sẽ đi luôn vào vấn đề mà nguyên lý này có thể giải quyết. Hãy thử nghĩ nếu chúng ta có một interface với cả vài chục method, khi một class implements interface này thì việc implements sẽ khá mất thời gian và ngoài ra một số method chúng ta không cần dùng đến nhưng vẫn phải implements. Do đó, nguyên lý này ra đời nhằm mục đích chúng ta nên tách interface lớn này ra thành những interface nhỏ gồm các method có liên quan tới nhau để việc implements và quản lý dễ dàng hơn.

Giờ hãy cùng xem qua một ví dụ vi phạm nguyên lý này:



Ta có thể nhận thấy ban đầu cách implement trên hoàn toàn không có vấn đề gì, nhưng khi ta thêm các động vật có tính chất riêng vào như Cá có thể bơi, Chim có thể bay. Điều này dẫn tới interface IAnimal bị phình to ra, và các class đã implements interface IAnimal phải implements thêm luôn cả những phương thức không dùng đến.



Vậy giờ làm thế nào để xử lý vấn đề này, rất đơn giản phải không nào? Chúng ta chỉ cần gom những phương thức có liên quan với nhau tách ra thành những interface nhỏ hơn.



Việc chia thành nhiều interface nhỏ sẽ làm giảm thiểu việc implements thừa các phương thức không cần thiết. Tuy nhiên sẽ làm tang số lượng interface, chúng ta cần chú ý để phân chia sao cho hợp lý.

## **5. Dependency Inversion Principle.**

Đây là nguyên lý thứ năm, cũng là nguyên lý cuối cùng ứng với chữ D trong SOLID. Nội dung của nguyên lý này có phần dài hơn và khó hiểu hơn so với các nguyên lý trên:

1. Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2. 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.)

**Giải thích nguyên lý:**

Trong phần nội dung nguyên lý, mình có dùng chữ module nhưng bạn cứ hình dung nó như là một class cho dễ hiểu.

Với cách viết code thông thường, theo cách mà mỗi người mới học lập trình đều gặp phải đó là các module cấp cao sẽ gọi các module cấp thấp. Điều này dẫn đến các module cấp cao sẽ phụ thuộc vào các module cấp thấp, và ta gọi sự phụ thuộc này là dependency. Khi module cấp thấp thay đổi dẫn đến việc thay đổi theo của các module cấp cao. Một thay đổi nhỏ của module cấp thấp cũng có thể kéo theo hàng loạt thay đổi, điều này gây khó khăn trong việc bảo trì và nâng cấp code.

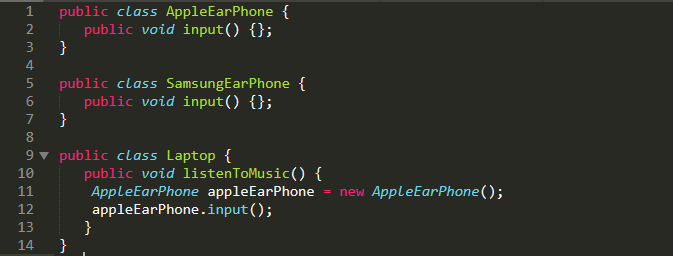
Vậy theo nguyên lý, ta sẽ giải quyết vấn đề này bằng cách các module cấp cao lẫn cấp thấp đều phụ thuộc vào abstract, ở đây là một interface không đổi. Ta có thể dễ dàng thay thế hoặc sửa đổi các module cấp thấp mà không làm ảnh hưởng đến module cấp cao.



Nhìn hình trên ta có thể dễ nhận biết bên trái là tai nghe của Iphone còn bên phải là tai nghe của Samsung. Điểm chung của hai loại tai nghe trên là nó có chung một kiểu jack cắm đầu vào. Khi ta cắm một trong hai loại tai nghe trên vào laptop, laptop chỉ quan tâm đến việc tai nghe có jack cắm phù hợp với khe cắm, không quan tâm tới việc tai nghe này được sản xuất bởi Apple hay là Samsung nên bạn thích nghe loại tai nghe nào chỉ việc chuyển đổi qua lại.

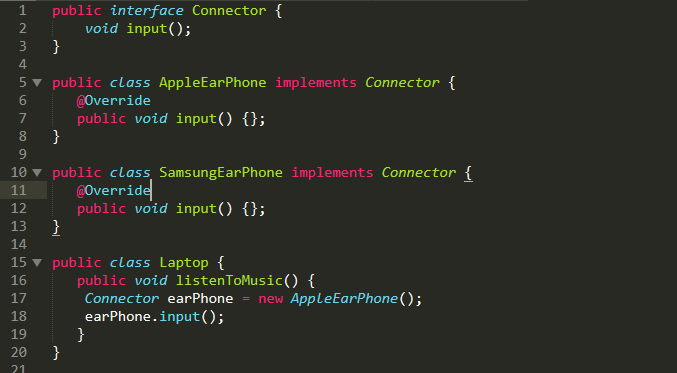
Đưa ví dụ trên vào trong code, laptop là module cấp cao, interface chính là jack cắm của tai nghe và các module cấp thấp là tai nghe Samsung và Apple. Hai module cấp thấp này đều implements interface jack cắm. Module cấp cao là laptop chỉ quan tâm tới tai nghe có implement jack cắm phù hợp hay không, không quan tâm tới việc tai nghe này có xuất sứ nguồn gốc ở đâu. Điều này rất dễ cho việc sửa đổi qua lại.

Giờ mình sẽ thể hiện ví dụ trên thông qua code:



Đoạn code trên đang vi phạm nguyên lý, chẳng hạn AppleEarPhone của bạn bị hư, bạn cần chuyển qua sử dụng SamsungEarPhone và việc chuyển đổi sẽ khó khan, phức tạp hơn.

Giờ mình sẽ áp dụng nguyên lý để cải tiến đoạn code trên.



Trong thực tế, người ta áp dụng Dependency Injection để đảm bảo nguyên lý Dependency Inversion trong code.

Vậy là mình vừa đi qua hết những nguyên lý trong SOLID PRINCIPLES. Các nguyên lý này chỉ là hướng dẫn, giúp code của bạn tốt hơn, rõ ràng và dễ bảo trì hơn. Tuy nhiên, không phải cứ cứng nhắc áp dụng nhiều nguyên lý này vào code thì code sẽ tốt, ngược lại sẽ khiến code của bạn rườm rà, dài dòng và khó quản lý.

Bài viết này mình có tham khảo từ:

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