## SOLID là gì?

**SOLID** là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc thiết kế hướng đối tượng. Giúp cho lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain. Nó được đưa ra bởi [Robert C. Martin](http://www.goodreads.com/author/show/45372.Robert_C_Martin) và Michael Feathers. 5 nguyên tắc đó bao gồm:

* **S**ingle responsibility priciple (SRP)
* **O**pen/Closed principle (OCP)
* **L**iskov substitution principe (LSP)
* **I**nterface segregation principle (ISP)
* **D**ependency inversion principle (DIP)

### **Single responsibility priciple**

**Nội dung:** Mỗi class chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.

Nguyên lý đầu tiên ứng với chữ S trong SOLID, có ý nghĩa là một class chỉ nên giữ một trách nhiệm duy nhất. Một class có quá nhiều chức năng sẽ trở nên cồng kềnh và trở nên khó đọc, khó maintain. Mà đối với ngành IT việc requirement thay đổi, cần thêm sửa chức năng là rất bình thường, nên việc code trong sáng, dễ đọc dễ hiểu là rất cần thiết.

Ví dụ: Hình dung rằng nhân viên của một công ty phần mềm cần phải làm 1 trong 3 việc sau đây: lập trình phần mềm (developer), kiểm tra phần mềm (tester), bán phần mềm (salesman). Mỗi nhân viên sẽ có một chức vụ và dựa vào chức vụ sẽ làm công việc tương ứng. Khi đó bạn có nên thiết kế class “Employee” với thuộc tính “position” và 3 phương thức developSoftware(), testSoftware() và saleSoftware() không?

class Employee

{

string position;

function developSoftware(){};

function testSoftware(){};

function saleSoftware(){};

}

Câu trả lời là KHÔNG. Thử hình dung nếu có thêm một chức vụ nữa là quản lí nhân sự, ta sẽ phải sửa lại class “Employee”, thêm phương thức mới vào sao? Nếu có thêm 10 chức vụ nữa thì sao? Khi đó các đối tượng được tạo ra sẽ dư thừa rất nhiều phương thức: Developer thì đâu cần dùng hàm testSoftware() và saleSoftware() đúng không nào, lỡ may dùng lầm phương thức cũng sẽ gây hậu quả khôn lường.

**Áp dụng nguyên tắc Single Responsibility**: mỗi class 1 trách nhiệm. Ta sẽ tạo 1 class trừu tượng là “Employee” có phương thức là working(), từ đây bạn kế thừa ra 3 class cụ thể là Developer, Tester và Salesman. Ở mỗi class này bạn sẽ implement phương thức working() cụ thể tuy theo nhiệm vụ của từng người. Khi đó chúng ta sẽ bị tình trạng dùng nhầm phương thức nữa.

### **Open/Closed principle**

**Nội dung:** Không được sửa đổi một Class có sẵn, nhưng có thể mở rộng bằng kế thừa.

Nguyên lý thứ 2 ứng với chữ O trong SOLID.Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) chứ không nên sửa đổi class cũ. Việc này dẫn đến tình trạng phát sinh nhiều class, nhưng chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới, nơi chứa các chức năng mới.

Thông thường việc mở rộng thêm chức năng thì phải viết thêm code, vậy để thiết kế ra một module có thể dễ dàng mở rộng nhưng lại hạn chế sửa đổi code ta cần làm gì. Cách giải quyết là tách những phần dễ thay đổi ra khỏi phần khó thay đổi mà vẫn đảm bảo không ảnh hưởng đến phần còn lại.

**Ví dụ:** Ta cần 1 class đảm nhận việc kết nối đến Database. Thiết kế ban đầu chỉ có SQL Server và MySQL. Thiết kế ban đầu có dạng như sau:

class ConnectionManager

{

public function doConnection(Object $connection)

{

if($connection instanceof SqlServer) {

//connect with SqlServer

} elseif($connection instanceof MySql) {

//connect with MySql

}

}

}

Sau đó yêu cầu đặt ra phải kết nối thêm đến Oracle và một vài Database khác.  
Để thêm chức năng ta phải thêm vào code những khối esleif khác, việc này làm code cồng kềnh và khó quản lý hơn.

* Giải pháp:
  + Áp dụng Abstract thiết kế lại các class SqlServer, MySql, Oracle…
  + Các class này đều có chung nhiệm vụ tạo kết nối đến database tương ứng có thể gọi chung là Connection.
  + Cách thức kết nối đến database thay đổi tùy thuộc vào từng loại kết nối nhưng có thể gọi chung là doConect.
  + Vậy ta có class cơ sở Connection có phương thức doConnect, các class cụ thể là SqlServer, MySql, Oracle… kế thừa từ Connection và overwrite lại phương thức doConnect phù hợp với class đó.

Thiết kế sau khi làm lại có dạng như sau:

abstract class Connection()

{

public abstract function doConnect();

}

class SqlServer extends Connection

{

public function doConnect()

{

//connect with SqlServer

}

}

class MySql extends Connection

{

public function doConnect()

{

//connect with MySql

}

}

class ConnectionManager

{

public function doConnection(Connection $connection)

{

//something

//.................

//connection

$connection->doConnect();

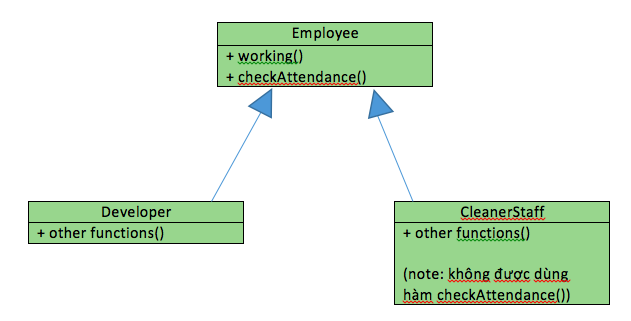
}

}

Với thiết kế này khi cần kết nối đến 1 loại database mới chỉ cần thêm 1 class mới kế thừa Connection mà không cần sửa đổi code của class ConnectionManager, điều này thỏa mãn 2 điều kiện của nguyên lý OCP.

### **Liskov substitution principle**

**Nội dung:** Các đối tượng (instance) kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.



Nguyên tắc thứ 3, ứng với chữ L trong SOLID. Minh hoạ một trường hợp vi phạm nguyên tắc Liskov substitution. Nếu thiết kế class như thế này, thì class CleanerStaff sẽ dùng được hàm checkAttendance(), mà điều này là không đúng, nên đây sẽ là một kiểu thiết kế sai nguyên tắc.

Quay trở lại ví dụ class Emloyee trong phần 1, ta giả sử có công ty sẽ điểm danh vào mỗi buổi sáng, và chỉ có các nhân viên thuộc biên chế chính thức mới được phép điểm danh. Ta bổ sung phương thức checkAttendance() vào class Employee.

Hình dung có một trường hợp sau: công ty thuê một nhân viên lao công để làm vệ sinh văn phòng, mặc dù là một người làm việc cho công ty nhưng do không được cấp số ID nên không được xem là một nhân viên bình thường, mà chỉ là một nhân viên thời vụ, do đó sẽ không được điểm danh.

**Nguyên tắc này nói rằng**: Nếu chúng ta tạo ra một class CleanerStaff kế thừa từ class Employee, và implement hàm working() cho class này, thì mọi thứ đều ổn, tuy nhiên class mới này cũng lại có hàm checkAttendance() để điểm danh, mà như thế là sai quy định dẫn đến chương trình bị lỗi. Như vậy, thiết kế class CleanerStaff kế thừa từ class Employee là không được phép.

Có nhiều cách để giải quyết tình huống này ví dụ như tách hàm checkAttendance() ra một interface riêng và chỉ cho các class Developer, Tester và Salesman implements interface này.

### **Interface segregation principle**

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

Nguyên lý này rất dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ rất vất vả vì các class impliment interface này sẽ bắt buộc phải phải thực thi toàn bộ các method của interface. Ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement và quản lý sẽ dễ hơn.

Ví dụ: Chúng ta có một interface Animal như sau:

interface Animal {

void Eat();

void Run();

void Fly();

}

Chúng ta có 2 class Dog và Snake implement interface Animal. Nhưng thật vô lý, Dog thì làm sao có thể fly(), cũng như Snake không thể nào run() được? Thay vào đó, chúng ta nên tách thành 3 interface như thế này:

interface Animal {

void Eat();

}

interface RunnableAnimal extends Animal {

void Run();

}

interface FlyableAnimal extends Animal {

void Fly();

}

### **Dependency inversion principle**

**Nội dung:**

1.Các module cấp cao không nên phụ thuộc vào các modules 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 (abstraction), không phải thông qua implementation.)

Có thể hiểu nguyên lí này như sau: những thành phần trong 1 chương trình chỉ nên phụ thuộc vào những cái trừu tượng (abstraction). Những thành phần trừu tượng không nên phụ thuộc vào các thành phần mang tính cụ thể mà nên ngược lại.

Những cái trừu tượng (abstraction) là những cái ít thay đổi và biến động, nó tập hợp những đặc tính chung nhất của những cái cụ thể. Những cái cụ thể dù khác nhau thế nào đi nữa đều tuân theo các quy tắc chung mà cái trừu tượng đã định ra. Việc phụ thuộc vào cái trừu tượng sẽ giúp chương trình linh động và thích ứng tốt với các sự thay đổi diễn ra liên tục.

**Ví dụ:** Ổ cứng của máy tính, bạn 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 bạn sẽ dùng ổ SSD hay loại HDD đĩa quay thông thường. Tuy nhiên họ sẽ luôn đảm bảo rằng bạn có thể dùng bất cứ thứ gì bạn 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.

Trong khi lập trình cũng vậy, khi áp dụng nguyên lý này, ở những class trừu tượng cấp cao, ta thường sử dụng interface nhiều hơn thay vì một kiểu kế thừa cụ thể. Ví dụ, để kết nối tới Database, ta thường thiết kế class trừu tượng DataAccess có các phương thức phương thức chung như save(), get(), … Sau đó tùy vào việc sử dụng loại DBMS nào (vd: MySql, MongoDB, …) mà ta kế thừa và implement những phương thức này. Tính chất đa hình của OOP được vận dụng rất nhiều trong nguyên lý này.

## KISS là gì?

***“K****eep****I****t****S****imple,****S****tupid!”*

KISS có nhiều biến thể khác nhau như "Keep It Short and Simple", "Keep It Simple and Straightforward" và "Keep It Small and Simple".

Tóm lại, hàm ý của nó vẫn hướng về một sự đơn giản và rõ ràng trong mọi vấn đề. Và như vậy, sự đơn giản là mục đích trọng tâm trong thiết kế, còn những cái phức tạp không cần thiết thì nên tránh.

Trong lập trình, KISS nghĩa là hãy làm cho mọi thứ (mã lệnh của bạn) trở nên đơn giản và dễ nhìn hơn. Hãy chia nhỏ vấn đề và giải quyết từng cái. Hãy để số lượng dòng code của một lớp hay phương thức ở con số hàng chục thôi.

Đừng viết những lớp hay phương thức theo kiểu tổng hợp hay lẫn lộn (tất cả trong một), hãy để mọi thứ thật đơn giản để bạn luôn có thể hiểu được, và kết hợp chúng với nhau để giải quyết được các bài toán lớn.

### **Ví dụ 1:** Ta có 1 method:

public int AddTwoNumber(int a, int b) => a + b;

Khi bạn đọc nó bạn có thể hiểu luôn method đó làm cái gì. Với những method có logic phức tạp hơn, dài dòng hơn thì bạn cần phải biết đặt tên biến, tên hàm, comment thế nào để cho code của bạn dễ hiểu.

### **Ví dụ 2:** Ta có 2 method đều thực hiện trả về true nếu số truyền vào là số chẵn:

// method 1

public static boolean IsEven(int input) => (input & 1) == 0;

// method 2

public static boolean IsEven(int input) => (input % 2) == 0;

Rõ ràng, method 1 nhanh hơn method 2 vì nó thực hiện trực tiếp với bit. Nhưng method 2 lại dễ hiểu hơn, người đọc không cần nhớ lại toán tử ‘&’ thực hiện như nào.

Ở đây tôi chọn method 2 vì nó dễ hiểu, rõ ràng. Còn method 1 có nhanh hơn nhưng tính tổng thể thì nó không nhanh hơn quá nhiều chỉ  tính bằng mini giây (chỉ khi nào ta thực hiện lập trình nhúng, big data… thì có lẽ lúc đó người ta sẽ xem xét lại.)

### **Ví dụ 3:** bạn có biết hàm tìm kiếm string cài đặt sẵn trong java được viết theo thuật toán ‘Naive Search’, trong khi bạn chỉ cần lên google là có hàng tá thuật toán cùng chức năng với ‘Naive Search’ mà có tốc độ cao hơn nhiều nhưng người ta vẫn dùng ‘Naive Search’ vì nó dễ hiểu, dễ cài đặt. Còn nếu bạn muốn nhanh hơn thì tất nhiên phải viết lại method mới phù hợp cho mục đích của mình rồi.

Với cấp độ cao hơn, khi mà người ta áp dụng các pattern, framework… bản chất code đã trở nên phức tạp hơn nhiều nên tùy theo nhu cầu (khả năng mở rộng, chịu tải, nâng cấp version…), thực sự cần thì người ta mới áp dụng. Chứ mấy cái app con con bạn cũng áp dụng vào vào thì nó sẽ khiến bạn tốn thời gian viết code, thời gian debug… Nếu bạn đọc code của đứa nào đó thấy khó hiểu thì đừng vội cho đó là cao siêu, hãy hỏi nó sao code khó hiểu thế.

### **Áp dụng KISS vào code của bạn.**

* Đặt tên hàm, biến, method, class, package, comment rõ nghĩa.
* Không lạm dụng framework, pattern.
* Chia task của bạn thành các task nhỏ hơn để code 1 cách rõ ràng.
* Giới hạn mỗi method chỉ nên từ 30-50 lines, và mỗi method chỉ nên thực hiện 1 chức năng.
* Hiểu rõ các vấn đề trước khi code.
* Không nên sử dụng nhiều câu lệnh If trong 1 method, hãy tách nó ra thành các method khác nhau.

### **Khi áp dụng được nguyên tắc KISS rồi bạn sẽ thấy nó có rất nhiều lợi ích:**

* Các vấn đề được giải quyết nhanh hơn, tránh được các issue phức tạp
* Code dễ sử dụng, dễ test
* Bản thân code đã chính là tài liệu, comment cho chính nó
* Và dễ dàng bảo trì, sửa lại code khi cần

## DRY là gì?

*“****D****on’t****R****epeat****Y****ourself”*

Nguyên tắc DRY là đừng bao giờ lặp lại code. Bạn cũng có thể đã nghe về sự so sánh: DRY cod vs WET code (Don’t Repeat Yourself and Write Everything Twice).

Bạn đã bao giờ viết các đoạn code giống nhau nằm ở các phần, module khác nhau của project? bạn đã bao giờ có 2 màn hình giống nhau nhưng lại dùng tới 2 đoạn code để hiển thị 2 màn hình đó?

OK, Nguyên tắc DRY chính là đang nói về vấn đề này.

Đừng lặp lại code ở đây là không lặp lại các đoạn code giống nhau, các method thực hiện chức năng như nhau, cố gắng gom chúng lại 1 cách gọn gàng và có thể dùng lại khi cần.

### **Ví dụ 1: Ta có 2 method**

public void print(User user) {

System.out.println("first name: "+user.getFirstName());

System.out.println("last name: "+user.getLastName());

System.out.println("age: "+user.getAge());

System.out.println("email: "+user.getEmail());

System.out.println("address: "+user.getAddress());

System.out.println("gender: "+user.getGender());

System.out.println("exprience: "+user.getExperience());

// do something: print user info

}

public void preview(User user) {

System.out.println("first name: "+user.getFirstName());

System.out.println("last name: "+user.getLastName());

System.out.println("age: "+user.getAge());

System.out.println("email: "+user.getEmail());

System.out.println("address: "+user.getAddress());

System.out.println("gender: "+user.getGender());

System.out.println("exprience: "+user.getExperience());

// do something: review user

}

public void print(User user) {

System.out.println("first name: "+user.getFirstName());

System.out.println("last name: "+user.getLastName());

System.out.println("age: "+user.getAge());

System.out.println("email: "+user.getEmail());

System.out.println("address: "+user.getAddress());

System.out.println("gender: "+user.getGender());

System.out.println("exprience: "+user.getExperience());

// do something: print user info

}

public void preview(User user) {

System.out.println("first name: "+user.getFirstName());

System.out.println("last name: "+user.getLastName());

System.out.println("age: "+user.getAge());

System.out.println("email: "+user.getEmail());

System.out.println("address: "+user.getAddress());

System.out.println("gender: "+user.getGender());

System.out.println("exprience: "+user.getExperience());

// do something: review user

}

Hai method trên thực hiện 2 chức năng khác nhau nhưng có rất nhiều code bị lặp lại, đây là case bị lặp code cơ bản ít người mắc phải. Ta có thể tránh lặp code trường hợp này bằng cách tách phần in thông tin user ra thành 1 method và gọi tới nó:

public void showUser(User user) {

System.out.println("first name: "+user.getFirstName());

System.out.println("last name: "+user.getLastName());

System.out.println("age: "+user.getAge());

System.out.println("email: "+user.getEmail());

System.out.println("address: "+user.getAddress());

System.out.println("gender: "+user.getGender());

System.out.println("exprience: "+user.getExperience());

}

public void print(User user) {

showUser(user);

// do something: print user info

}

public void preview(User user) {

showUser(user);

// do something: review user

}

Sau khi tách ra method showUser thì ta có thể dùng lại nó hoặc nếu cần chỉnh sửa, lỗi trong việc in ra thông tin user thì ta chỉ cần sửa method showUser là được.

### **Ví dụ 2:** Dưới đây là 1 đoạn code thực hiện tìm kiếm đối tượng User theo name, email, theo cả name và email.

public List<User> findByName(String name) {

if (name == null) {

return null;

}

String sql = "SELECT u FROM User u WHERE u.name like %"+name+"% ";

// do something

}

public List<User> findByEmail(String email) {

if (email == null) {

return null;

}

String sql = "SELECT u FROM User u WHERE u.email like %"+email+"% ";

// do something

}

public List<User> findByNameAndEmail(String name, String email) {

if (name == null || email == null) {

return null;

}

String sql = "SELECT u FROM User u WHERE u.name like %"+name+"% AND u.email like %"+email+"%";

// do something

}

Cái gì thế này? có vẻ như cả 3 method này đều thực hiện một việc khá giống nhau. Sao không chuyển thành hàm dưới đây:

public List<User> find(String name, String email) {

if (name == null && email == null) {

return null;

}

String sql = "SELECT u FROM User u WHERE u.name like %"+name+"% AND u.email like %"+email+"%";

if (name != null && email == null) {

sql = "SELECT u FROM User u WHERE u.name like %"+name+"% ";

}

if (name == null && email != null) {

sql = "SELECT u FROM User u WHERE u.email like %"+email+"% ";

}

// do something

}

Ngắn gọn hơn đúng không, lại vừa có thể search theo name, theo  email và theo cả 2.

Vậy là việc suy tính trước method của mình sẽ làm gì sẽ giúp bạn trành việc lặp code, phải viết thêm method mới. (Chú ý, giả sử bạn có method findByName, bây giờ cần thêm method search theo email thì phải giữ nguyên method findByName và viết mới method find chứ không được xóa method cũ đi nếu không nó sẽ vi phạm nguyên tắc  “open for extension but closed for modification” tức là ưu tiên thêm mới và hạn chế sửa đổi).

## YANGI là gì?

*“****Y****ou* ***A****ren't* ***G****onna* ***N****eed* ***I****t”*

Đôi khi, là những lập trình viên, chúng ta cố gắng nghĩ cách để tiến về phía trước, về tương lai của dự án, Làm thêm một vài tính năng "Trong trường hợp chúng ta cần chúng" hoặc nghĩ rằng "cuối cùng chúng ta sẽ cần chúng". Chỉ một từ thôi: Wrong! Bạn không cần nó, bạn không cần làm điều này và trong nhiều trường hợp ... "Bạn thực sự không cần".

YAGN là một nguyên tắc đằng sau extreme programming (XP) practice của " Do the Simplest Thing That Could Possibly Work ". Ngay cả khi những nguyên tắc này là một phần của XP, Nó được áp dụng trong tất cả các phương pháp và quá trình của nhà phát triển. Bằng cách thực hiện các ý tưởng của "You Aren't Gonna Need It". Bạn sẽ tiết kiệm được thời gian của chính bạn và cũng có thể tiến lên phía trước với những dự án hiệu quả.

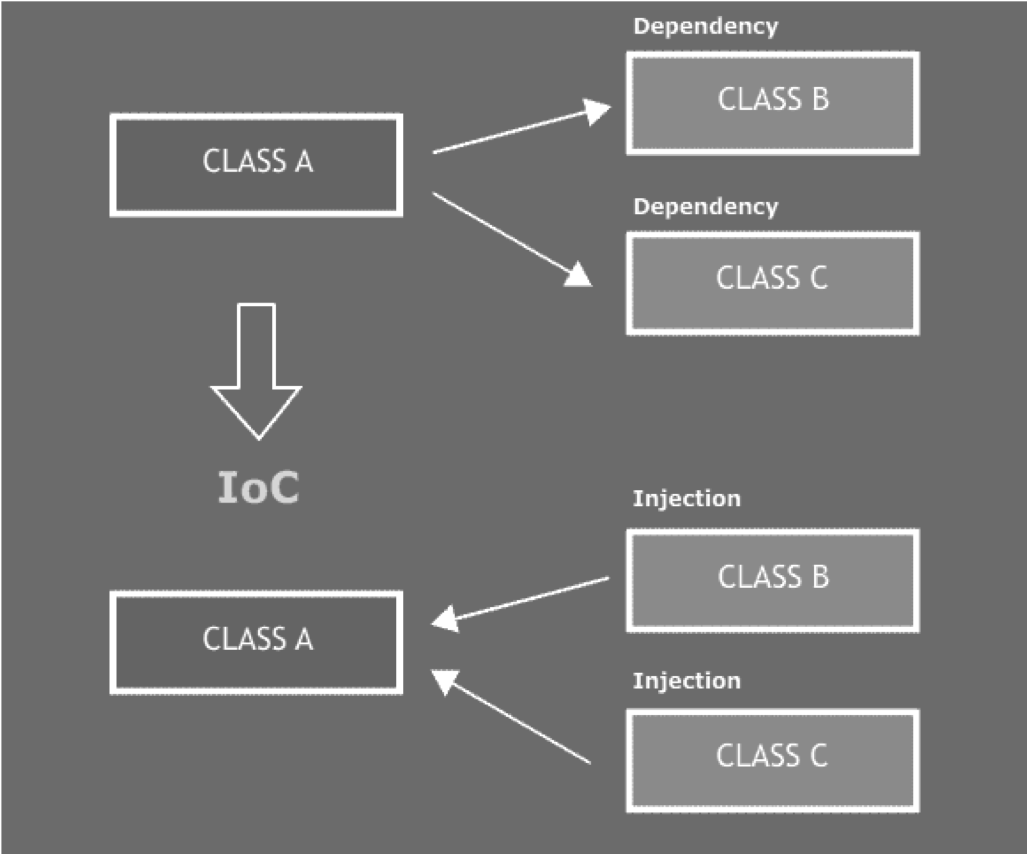
Khi bạn cảm thấy lo lắng, khó hiểu để làm thêm vài tính năng mở rộng thì trong lúc này là không cần thiết, nhưng bạn nghĩ rằng chúng sẽ hữu ích trong tương lai, Hãy bình tĩnh và xem lại những công việc đang chờ bạn xử lý ngay lúc này. Bạn không thể lãng phí thời gian để làm những tính năng mà bạn có lẽ cần sửa, thay đổi, xóa vì chúng không phù hợp với nhu cầu của sản phẩm.

## F.I.R.S.T

FIRST ở đây là nguyên tắc dung trong UnitTest. Nhằm tạo được các Test sạch. FIRST tuân theo năm quy tắc tạo thành từ viết tắt, Cụ thể như sau:

1. **Fast** Kiểm tra phải nhanh chóng. Nó nên chạy thật nhanh. Khi các bài kiểm tra chạy chậm, bạn sẽ không muốn chạy chúng thường xuyên. Nếu bạn không chạy chúng thường xuyên, bạn sẽ không phát hiện ra các vấn đề đủ sớm để khắc phục chúng một cách dễ dàng. Bạn sẽ không cảm thấy thoải mái khi xóa code. Cuối cùng thì code sẽ bắt đầu không được bảo trì.
2. **Independent** Các bài kiểm tra độc lập không nên phụ thuộc vào nhau. Một thử nghiệm không nên thiết lập các điều kiện cho thử nghiệm tiếp theo. Bạn sẽ có thể chạy từng bài kiểm tra một cách độc lập và chạy các bài kiểm tra theo bất kỳ thứ tự nào bạn muốn. Khi các bài kiểm tra phụ thuộc vào nhau, thì bài kiểm tra đầu tiên không thành công sẽ gây ra hàng loạt các thất bại phía dưới, làm cho việc chẩn đoán trở nên khó khăn và che giấu các lỗi phía dưới.
3. **Repeatable** Các thử nghiệm lặp lại nên được lặp lại trong mọi môi trường. Bạn sẽ có thể chạy các bài kiểm tra trong môi trường sản xuất, trong môi trường QA và trên máy tính xách tay của mình khi đang đi tàu về nhà mà không có mạng. Nếu các bài kiểm tra của bạn không thể lặp lại trong bất kỳ môi trường nào, thì bạn sẽ luôn có lý do tại sao chúng thất bại. Bạn cũng sẽ thấy mình không thể chạy các bài kiểm tra khi môi trường không có sẵn.
4. **Self-Validating** Tự xác thực Các bài kiểm tra phải có đầu ra boolean. Đạt hoặc không đạt. Bạn không cần phải đọc qua tệp nhật ký để biết liệu các bài kiểm tra có vượt qua hay không. Bạn không cần phải so sánh thủ công hai tệp văn bản khác nhau để xem liệu các bài kiểm tra có vượt qua hay không. Nếu các bài kiểm tra không tự xác thực thì việc thất bại có thể trở nên bị động và việc chạy các bài kiểm tra có thể yêu cầu đánh giá thủ công lãng phí thời gian.
5. **Timely** Đúng lúc Các bài kiểm tra cần được viết đúng thời hạn. Các bài kiểm tra đơn vị nên được viết ngay trước code product khiến chúng vượt qua. Nếu bạn viết các bài kiểm tra sau code product, thì bạn có thể thấy code product khó kiểm tra. Bạn có thể quyết định rằng một số code product quá khó để kiểm tra. Bạn không thể thiết kế code product để có thể kiểm tra được.

## IoC là gì?

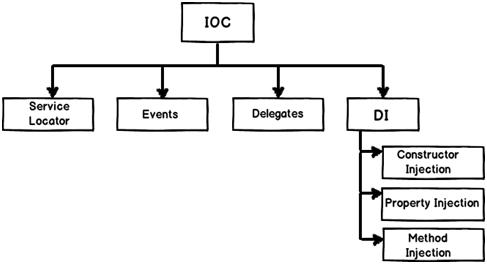


**Inversion of Control (IoC)** là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.

Hình bên là Class A ở hai trường hợp, áp dụng IoC và không. Lớp này nó làm việc phụ thuộc vào hai lớp Class B và Class C (B và C gọi là các dependency)

Ở mô hình không IoC, Class A khi cần chủ động tạo ra đối tượng lớp Class B và Class C (nó nắm quyền khởi tạo, điều khiển)

Với mô hình **IoC** thì class A không tự khởi tạo cũng không chịu trách nhiệm quản lý Class B, Class C. Nó nhận được hai dependency này từ bên ngoài thông qua một cơ chế nào đó (như bằng setter, bằng tham số hàm tạo lớp A, bằng gán thuộc tính ...)



Triển khai nguyên lý **IoC** thường thực hiện bởi các Framework theo từng loại ngôn ngữ lập trình, các mô hình lập trình (pattern) triển khai từ **IoC** như:

* Service Locator
* Events
* DI (Dependency Inject)

### **Mô hình Service Locator**

Service Locator - Một mô hình triển khai nguyên lý IoC. Sử dụng mô hình này có nghĩa cần có cơ chế sao cho tất các các tiến trình (các đối tượng, dịch vụ ...) được đưa vào quản lý bởi một trung tâm được gọi là Service Locator, từ trung tâm đăng ký (registry) này các lớp muốn sử dụng dịch vụ nào thì Locator sẽ cung cấp (khởi tạo nếu chưa, và trả về dịch vụ cho đối tượng sử dụng).

### **Mô hình Dependency Inject - DI**

Dependency injection (DI) là một mô hình triển khai từ nguyên lý IoC, là một kỹ thuật trong lập trình trong đó một đối tượng cung cấp những phụ thuộc (dependency - là đối tượng, dịch vụ, chức năng) của đối tượng khác. Injection - Bơm vào (tiêm vào) ám chỉ một phụ thuộc (đối tượng, dịch vụ) đưa vào đối tượng để đối tượng sử dụng nó.

## IoC Container

IoC Container (a.k.a. DI Container) là một framework để thực thi dependency injection tự động. Nó quản lý việc tạo đối tượng và life-time của đối tượng đó, đồng thời đưa các dependenciesvào class.

IoCContainer tạo một object của class được chỉ định và cũng đưa vào tất cả các dependency objects thông qua một hàm tạo(Contructer), một propertyhoặc một method tại thời điểm runtime và xử lý nó vào thời điểm thích hợp. Điều này được thực hiện để chúng ta không phải tạo và quản lý các object theo cách thủ công.

Tất cả các container phải hỗ trợ dễ dàng cho DI lifecycle như sau**:**

* 1. **Register**: container phải biết phụ thuộc nào cần khởi tạo khi nó gặp một kiểu cụ thể. Quá trình này được gọi là đăng ký. Về cơ bản, nó phải bao gồm một số cách để đăng ký ánh xạ kiểu.

* 1. **Resolve**: Khi sử dụng IoC container, chúng ta không cần phải tạo các đối tượng theo cách thủ công. Hộp chứa làm điều đó cho chúng tôi. Đây được gọi là resolve. container phải bao gồm một số phương pháp để giải quyết loại được chỉ định; vùng chứa tạo một đối tượng thuộc kiểu được chỉ định, đưa vào các phụ thuộc bắt buộc nếu có và trả về đối tượng.
  2. **Dispose**: container phải quản lý thời gian tồn tại của các đối tượng phụ thuộc. Hầu hết các IoCcontainer bao gồm các bộ quản lý filetime khác nhau để quản lý vòng đời của một đối tượng và loại bỏ nó.

## Yêu cầu:

## Sinh viên tiến hành áp dụng các nguyên tắc trên vào code sau này.