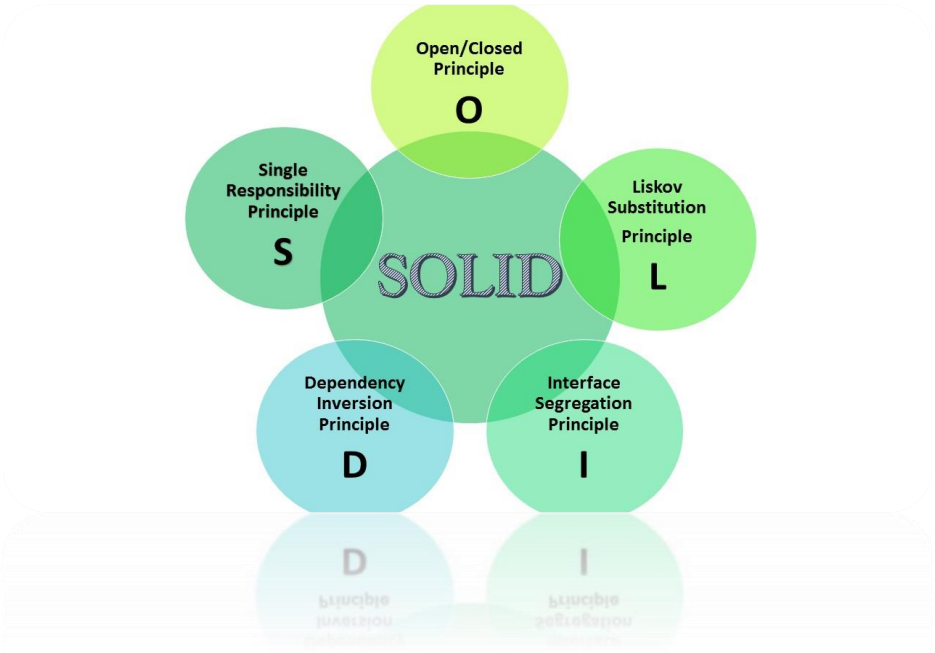
PHÁT TRIỂN HỆ THỐNG THÔNG TIN HIỆN ĐẠI

SEMINAR – SOLID in Java

Nhóm Awakening-2



PTHTTTHD-SEMINAR-SOLID

AWAKENING-2

6/10/2017

MỤC LỤC

PTHTTTHD-SEMINAR-SOLID

AWAKENING-2

6/10/2017

MSSV	Họ tên
1312636	Nguyễn Hoàng Quốc Trung
1412112	Trà Ngô Ngọc Dương
1412183	Phạm Quốc Hoàng
1412199	Hà Ngọc Huy

Thông tin thành viên

I. Nguyên lí SOLID

OOP (Object Oriented Programming) là một mô hình lập trình được sử dụng nhiều nhất và hiệu quả nhất hiện nay.

OOP có 4 tính chất đặc trưng sau đây:

- Encapsulation (Tính đóng gói)
- Abstraction (Tính trừu tượng)
- Inheritance (Tính thừa kế)
- Polymorphism (Tính đa hình)

Trong thực tế, sản phẩm phần mềm luôn có sự thay đổi và mở rộng chức năng theo thời gian do đó cần có những quy trình phát triển phần mềm, khâu phân tích thiết kế đóng vai trò vô cùng quan trọng trong việc tạo ra các kiến trúc phần mềm dễ dàng đáp ứng thay đổi nhất.

SOLID là 5 nguyên tắc cơ bản, giúp xây dựng một kiến trúc phần mềm tốt. Tất cả các design pattern đều dựa trên bộ nguyên tắc này. Các nguyên tắc này là một tập hợp con của nhiều nguyên tắc do Robert C. Martin đề xướng. Từ SOLID được giới thiệu bởi Michael Feathers

Một dự án áp dụng SOLID sẽ có code dễ đọc hiểu, dễ bảo trì và mở rộng. SOLID bao gồm 5 nguyên tắc:

- Single responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

1. Single Responsibility Principle

Nội dung:

Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.

Ví dụ làm rõ:

Ví dụ 1:

Giả sử ta có class sau:

```
class Customer {
  public void Add() {
    try {
      // Database code goes here
    } catch (Exception ex) {
      System.IO.File.WriteAllText(@"C:\log.txt", ex.ToString());
    }
  }
}
```

Class *Customer* ngoài việc thực hiện các xử lý đến đối tượng *Customer* còn thực hiện cả việc ghi log. Công việc ghi log thì quan trọng nhưng thực hiện nó như trên thì không tốt. Rõ ràng class *Customer* chỉ nên làm những việc như là validation dữ liệu, xử lý logic liên quan tới các dữ liệu của *Customer*.

Điều này sẽ gây ra khó khăn khi chúng ta muốn thay đổi việc ghi log, vì khi muốn đổi cách thức ghi log file chúng ta phải vào class *Customer* để sửa. Việc này thì không tốt chút nào.

Giải pháp:

Để đảm bảo nguyên lý này thì chúng ta sẽ chuyển đoạn code ghi log sang class *FileLogger*, class này chỉ làm việc với log mà thôi.

```
class FileLogger {
  public void Handle(string message) {
    System.IO.File.WriteAllText(@"c:\log.txt", message);
  }
```

```
}
class Customer {
  private FileLogger logger = new FileLogger();
  publicvirtual void Add() {
    try {
      // Database code goes here
    } catch (Exception ex) {
      logger.Handle(ex.ToString());
    }
  }
}
```

Mọi thứ đã trở nên rõ ràng hơn trước. Class *Customer* chỉ làm việc với đối tượng customer và class *FileLogger* sẽ chuyên tâm làm việc với nhiệm vụ ghi log. Ở đây có thể dễ dàng thấy được lợi ích của việc tách 2 class này ra.

Ví dụ 2:

Hình dung cụ thể rằng mỗi nhân viên của một công ty phần mềm giữ 1 trong 3 vai trò sau: lập trình phần mềm (developer), kiểm thử phần mềm (tester), bán phần mềm (salesman). Mỗi nhân viên dựa vào chức vụ sẽ làm công việc tương ứng. Hình thành class Employee ban đầu như sau:

```
class Employee
{
  string position;
  function developSoftware(){};
  function testSoftware(){};
  function saleSoftware(){};
}
```

Khi ta thiết kế class *Employee* với thuộc tính position (developer, tester, salesman) và các phương thức *developSoftware()*, *testSoftware()* và *saleSoftware()* như thế này nếu có thêm một chức vụ nữa là quản lí nhân sự, thì ta phải vào sửa lại class *Employee* và thêm phương thức mới. Nếu có thêm nhiều

chức vụ nữa thì sao? Khi đó các đối tượng được tạo ra sẽ dư thừa rất nhiều phương thức: chẳng hạn Developer thì đâu cần dùng hàm *testSoftware()* và *saleSoftware()*, chẳng may dùng lầm phương thức sẽ gây ra hậu quả khôn lường.

Giải pháp:

Để đảm bảo nguyên lý này, *Employee* sẽ là lớp trừu tượng có phương thức *working()*, tạo ra 3 class cụ thể là *Developer*, *Tester* và *Salesman* kế thừa lớp trừu tượng này. Mỗi class sẽ được implement phương thức *working()* cụ thể tùy theo vai trò của từng class. Khi đó chúng ta sẽ không bị tình trạng dùng sai phương thức.

Kết luận:

Về bản chất nguyên lý này chỉ là hướng dẫn không phải là nguyên tắc tuyệt đối. Có những trường hợp như các class Helper xét cho cùng toàn bộ các hàm trong class này đều thực hiện những tác vụ nhỏ nên nếu số lượng hàm ít vẫn có thể cho các hàm này vào cùng một class. Tuy nhiên khi số lượng hàm tăng lên quá nhiều thì nên cân nhắc sử dụng nguyên lý này để chia nhỏ module thuận tiện cho việc quản lý.

Việc hiểu và áp dụng nguyên này giúp cho việc viết code dễ đọc, dễ hiểu, dễ quản lý hơn. Tuy nhiên nguyên lý đơn nhiệm là nguyên lý đơn giản nhưng khó áp dụng đúng, việc xác định khi nào cần áp dụng khi nào không còn phụ thuộc vào việc người code xác định được đúng chức năng của module đang làm.

2. Open/Closed Principle

Thoải mái mở rộng một Class, nhưng không được sửa đổi bên trong của Class đó (software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification)

Nghĩa là khi muốn phát triển thêm chức năng cho chương trình, nên viết class mở rộng của class cha (Kế thừa hoặc sở hữu – là thuộc tính của Class mới). Từ

đó làm phát sinh nhiều class mới nhưng chỉ cần tập trung test các class mới này, nơi có các chức năng mới.

Ví dụ: Bạn có chiếc máy ảnh có đèn flash nhưng công suất yếu, bạn muốn đèn sáng hơn thì tốt nhất là ráp thêm một cái flash rời, chứ không nên mở cái flash bên trong máy ra để thay bóng đèn công suất cao hơn.

Code ví dụ

Trường hợp vi phạm nguyên tắc O

```
// class Xe
public class Vehicle {

    private int power;
    private int suspensionHeight;

    public int getPower() {
        return power;
    }

    public int getSuspensionHeight() {
        return suspensionHeight;
    }

    public void setPower(int power) {
        this.power = power;
    }

    public void setSuspensionHeight(int suspensionHeight) {
        this.suspensionHeight = suspensionHeight;
    }

}
```

```
// chế độ chạy
public enum DrivingMode {
    SPORT, COMFORT
}
```

```java
// Xử lí Sự kiện
public class EventHandler {

    private Vehicle vehicle;

    public EventHandler(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public void changeDrivingMode(DrivingMode drivingMode){
        switch (drivingMode){
            case SPORT:
                vehicle.setPower(500);
                vehicle.setSuspensionHeight(10);
                break;
            case COMFORT:
                vehicle.setPower(400);
                vehicle.setSuspensionHeight(20);
                break;
            default:
                vehicle.setPower(400);
                vehicle.setSuspensionHeight(20);
                break;
            // Khi thêm chế độ chạy mới thì cần phải thay đổi 2 class DrivingMode và EventHandler
        }
    }
}
```

Giải pháp

- Class Vehicle vẫn giữa lại như cũ.
- Tạo ra interface DrivingMode chứa 2 method như bên dưới

```java
public interface DrivingMode {

    int getPower();
    int getSuspensionHeight();
}
```

- Tạo các class Comfort, Economy, Sportimplement interface DrivingMode. Các class này chứa các thuộc tính và viết lại method của interface DrivingMode

```java
public class Comfort implements DrivingMode {

    private static final int POWER = 400;
    private static final int SUSPENSION_HEIGHT = 20;

    @Override
    public int getPower() {
        return POWER;
    }

    @Override
    public int getSuspensionHeight() {
        return SUSPENSION_HEIGHT;
    }
}
```

```java
public class EventHandler {

    private Vehicle vehicle;

    public EventHandler(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public void changeDrivingMode(DrivingMode drivingMode){
        vehicle.setPower(drivingMode.getPower());
        vehicle.setSuspensionHeight(drivingMode.getSuspensionHeight());
        // Từ bây giờ khi tạo mới một chế độ ví dụ ECONOMY thì chỉ cần tạo ra class ECONOMY
    }
}
```

```java
public class Economy implements DrivingMode{

    private static final int POWER = 300;
    private static final int SUSPENSION_HEIGHT = 20;

    @Override
    public int getPower() {
        return POWER;
    }

    @Override
    public int getSuspensionHeight() {
        return SUSPENSION_HEIGHT;
    }
}
```

```java
public class Sport implements DrivingMode {

    private static final int POWER = 500;
    private static final int SUSPENSION_HEIGHT = 10;

    @Override
    public int getPower() {
        return POWER;
    }

    @Override
    public int getSuspensionHeight() {
        return SUSPENSION_HEIGHT;
    }
}
```

3. Liskov Substitution Principle

Nội dung:

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

Ví dụ làm rõ:

Ví dụ 1:

Hình vuông là hình chữ nhật đặc biệt có chiều cao bằng chiều rộng. Vậy class *Square* có phải con của class *Rectangle*?

Xem xét mối quan hệ kế thừa giữa hình vuông và hình chữ nhật như sau:

```
class Rectangle
{
  protected $m_width;
  protected $m_height;

  public function setWidth(int $width) {
     $this->m_width = $width;
  }

  public function setHeight(int $height) {
     $this->m_height = $height;
  }

  public function getWidth() {
     return $this->m_width;
  }

  public function getHeight() {
     return $this->m_height;
  }

  public function getArea() {
     return $this->m_width * $this->m_height;
  }
}

class Square extends Rectangle
{
  public function setWidth(int $width) {
     $this->m_width = $width;
     $this->m_height = $width;
```

```
    }

    public function setHeight(int $height) {
        $this->m_height = $height;
        $this->m_width = $height;
    }
}
```

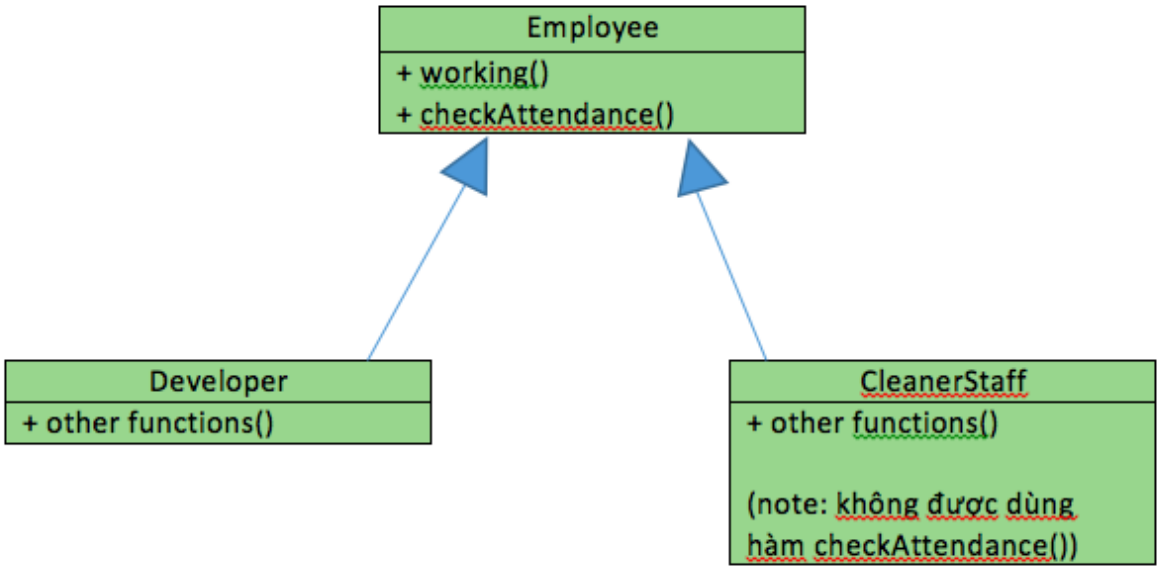
Theo Liskov Substitution Principle thì class *Square* có thể thay thế class *Rectangle* nên ta thực hiện việc test như sau:

```
class Test
{
    public function checkArea(Rectangle $r)
    {
        $r->setWidth(10);
        $r->setHeight(5);

        if($r->getArea() == 50) {
            return 'true';
        }

        return 'false';
    }
}

$test = new Test;
echo 'Test with Rectangle: '.$test->checkArea(new Rectangle).PHP_EOL;// Test with Rectangle: true
echo 'Test with Square: '.$test->checkArea(new Square).PHP_EOL;// Test with Square: false
```

Phương thức *Test* hoạt động đúng (50) khi $r là một thể hiện của *Rectangle* nhưng khi thay thế $r là thể hiện của *Square* thì kết quả là false (25). Lớp Square đã làm mất đi tính đúng đắn của chương trình.

Vậy hình vuông không phải là 1 hình chữ nhật? Xét về mặt hành vi thì hình vuông không phải là hình chữ nhật vì hành vi của hình vuông không thỏa mãn yêu cầu của hàm checkArea.

Ví dụ 2:



Trường hợp vi phạm Liskov Substitution Principle

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 này cũng vẫn sẽ 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* implement.

Ví dụ 3

Ví dụ Class Ostrich kế thừa class Bird nhưng làm thay đổi hình vi của hàm fly()

```
class Bird {
  public void fly(){}
  public void eat(){}
}
class Crow extends Bird {}
class Ostrich extends Bird{
  public void fly(){
    throw new UnsupportedOperationException();
  }
}

public BirdTest{
  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird());
    birdList.add(new Crow());
    birdList.add(new Ostrich());
    letTheBirdsFly ( birdList );
  }
  static void letTheBirdsFly ( List<Bird> birdList ){
    for ( Bird b : birdList ) {
      b.fly();// throw Exception when bird is Ostrich
    }
  }
}
```

Giải pháp

Tạo ra 2 lớp kế thừa là Chim bay được và Chim không bay được

```
class Bird{
  public void eat(){}
}
class FlightBird extends Bird{
  public void fly()()
}
class NonFlight extends Bird{}
```

Kết luận:

Trong lập trình chỉ cho class A kế thừa class B khi class A thay thế được cho class B. Cần xem xét các đối tượng về mặt hành vi chứ không nên sử dụng những mối quan hệ trong đời thật.

4. Interface Segregation Principle

Nội dung:

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

Diễn giải:

Giả như ta có 1 interface lớn, khoảng 100 methods. Việc implement sẽ rất vất vả vì các class implement 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.

Hình dung cụ thể khi thiết kế 1 lớp Ultilities, sẽ có hàng trăm các method khác nhau, các method này hỗ trợ các tính toán và xử lí bổ trợ liên quan tới user, report, database… Khi method trở nên quá nhiều thì chúng ta nên tách interface lớn thành các interface nhỏ hơn, ví dụ như: UserUltilities, ReportUltilities, DBUltilities,… như vậy sẽ dễ dàng quản lí và sử dụng hơn.

Trong thực tế, bằng việc tách ra các interface nhỏ hơn, ta sẽ dễ dàng áp dụng các quy tắc khác trong SOLID hơn, điều đó làm code trở nên trong sáng và hiệu quả hơn.

Ví dụ làm rõ:

Xây dựng 1 interface *Phone*, các class *SmartPhone*, *PhoneBox* implement *Phone*.

```
interface Phone
{
  public function call();
  public function sms();
  public function picture();
}

class SmartPhone implements Phone
{
  public function call()
  {
    // do call
  }

  public function sms()
  {
    // send sms
  }

  public function picture()
  {
    // take picture
  }
}

class PhoneBox implements Phone
```

```
{
  public function call()
  {
     // do call
  }

  public function sms()
  {
     throw new Exception("PhoneBox can't send SMS");
  }

  public function picture()
  {
     throw new Exception("PhoneBox can't take picture");
  }
}
```

Nếu class *Phone* có thêm các phương thức mới thì các class *SmartPhone*, *PhoneBox* sẽ phải thay đổi theo trong khi nó không cần hoặc không thể thực hiện.

Giải pháp:

- Chia nhỏ interface *Phone* thành những interface mang nhiệm vụ đặc thù.
- Các lớp *SmartPhone*, *PhoneBox* implement interface đặc thù mà nó cần.

```
interface Callable()
{
  public function call();
}

interface Smsable()
{
  public function sms();
}
```

```
interface Pictureable()
{
  public function sms();
}

class SmartPhone implements Callable, Smsable, Pictureable
{
  public function call()
  {
    // do call
  }

  public function sms()
  {
    // send sms
  }

  public function picture()
  {
    // take picture
  }
}

class PhoneBox implements Callable
{
  public function call()
  {
    // do call
  }
}
```

Như vậy khi mở rộng ta thêm các interface đặc thù với chức năng mở rộng thì sẽ không ảnh hưởng đến các lớp dẫn xuất.

Kết luận:

Interface Segregation Principle (ISP) có mối liên hệ với Open - Close Principle (OCP). Vi phạm ISP có khả năng dẫn tới sự vi phạm OCP.

Những lớp trừu tượng chứa quá nhiều thuộc tính và chức năng gọi là những lớp bị ô nhiễm (polluted). Các lớp dẫn xuất phụ thuộc vào polluted interface làm tăng sự kết dính giữa các thực thể. Khi nâng cấp, sửa đổi đòi hỏi các interface này thay đổi, các lớp dẫn xuất này buộc phải thay đổi theo, điều này vi phạm nguyên lý OCP.

5. Dependency Inversion Principle

Các modules cấp cao không nên phụ thuộc vào các modules 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.)

Ví dụ

Một hệ thống máy tính sẽ có mainboard là thành phần chính, bộ phận này kết nối các thành phần khác trong hệ thống (như CPU, Ram, ổ cứng, …) lại với nhau để tạo nên một hệ thống hoạt động hoàn chỉnh và thống nhất. Một mainboard có khả năng kết nối nhiều loại Ram, nhiều loại ổ cứng, … Dù những nhà sản xuất Ram hay mainboard là độc lập nhau hoàn toàn và cũng có rất nhiều loại mainboard và ổ cứng khác nhau, nhưng các bộ phận này được kết nối với nhau rất dễ dàng. Do mọi linh kiện máy tính dù cho có cấu tạo chi tiết khác nhau (implement khác nhau), nhưng luôn giao tiếp với nhau thông qua các chuẩn đã định sẵn (abstraction), cụ thể là mainboard giao tiếp với ổ đĩa cứng thông qua chuẩn kết nối chung SATA.

Ví dụ code

Trường hợp vi phạm

```
class Worker {

    public void work() {

        // ....working

    }

}
class Manager {

    Worker worker;


    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```

Giải pháp

Class Manager giao tiếp với worker thông qua Interface IWorker và worker implement IWorker.

```
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker  implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

II. Spring Framework

1. Sự ra đời của Spring

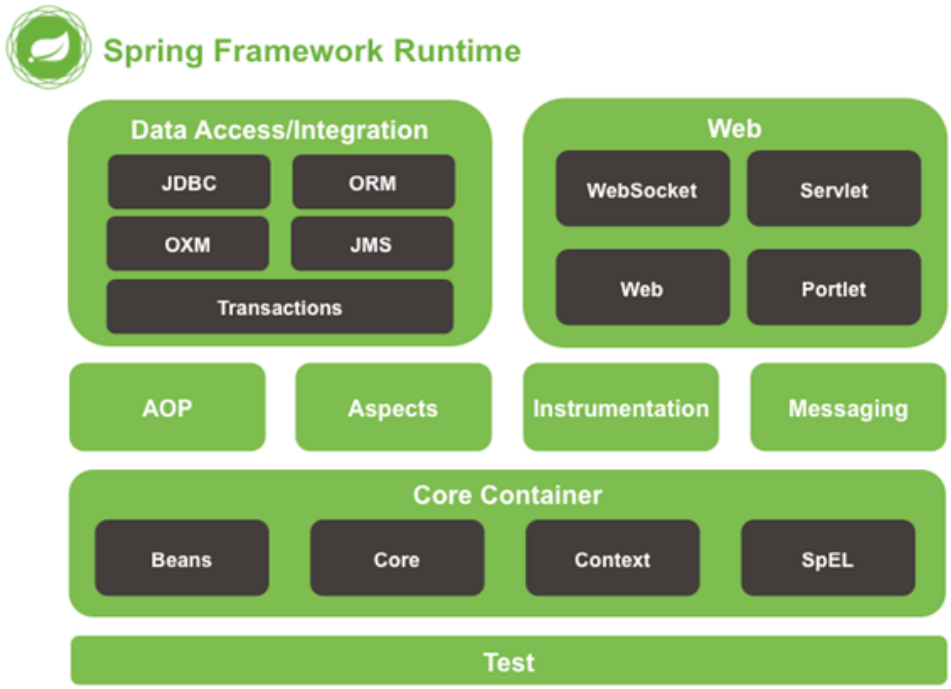
Vào năm 2002, Spring Framework phát hành phiên bản đầu tiên bởi Rod Johnson. Hiện nay, Spring đã trở thành framework mã nguồn mở phổ biến nhất để xây dựng các ứng dụng doanh nghiệp. Trên 50% các ứng dụng web Java hiện nay đang sử dụng Spring.

Để ngăn chặn sự phức tạp trong phát triển các ứng dụng, Spring Framework thường dựa trên các quan điểm như sau:

- Đơn giản hóa công việc phát triển thông qua việc sử dụng các đối tượng Java đơn giản POJO (Plain Old Java Object)
- Nới lỏng ràng buộc giữa các thành phần thông qua việc sử dụng Dependency Injection và viết các interface
- Tiếp cận lập trình khai báo bằng cách sử dụng các quy tắc (convention) và các khía cạnh (aspect) chung
- Giảm thiểu các mã nghi thức và soạn sẵn (boilerplate) thông qua việc sử dụng các khuôn mẫu (template) và các khía cạnh

2. Spring Core

Spring Core chính là thành phần trung tâm, cốt lõi của Spring Framework. Đây chính là nền tảng để xây dựng nên các thành phần khác.



Lược đồ mô tả mối quan hệ giữa Spring Core với các thành phần khác trong Spring Framework

3. Spring Bean

Spring Bean là trung tâm của Spring Core và là trái tim của một ứng dụng Spring. Spring BeanSpring Framework được thiết kế từ lõi bằng cách sử dụng các POJO hay các Spring Bean. Spring Bean có thể được hiểu là các đối tượng Java đơn giản. Điều này khá tương đồng với nguyên lý thiết kế "Đơn trách nhiệm" (single responsibility principle) của Robert C Martin trong lập trình hướng đối tượng

4. Dependency Injection (DI)

Dependency Injection là một sức mạnh nổi bật của Spring Framework. Dependency Injection là một mẫu thiết kế phần mềm mà các đối

tượng phụ thuộc sẽ được inject vào một lớp nào đó. Dependency Injection là một implementation cụ thể của khái niệm Inversion of Control (đảo ngược điều khiển)

Xét một ví dụ đơn giản về Dependency Injection như sau: Bạn có một web controller có nhiệm vụ lưu thông tin gửi từ form người dùng. Theo nguyên lý Đơn trách nhiệm (single responsibility principle), bạn không muốn lớp controller tương tác với cơ sở dữ liệu. Vì thế, bạn sẽ sử dụng một lớp service để làm công việc này. Như vậy, controller của bạn sẽ chỉ phải xử lý dữ liệu của form (get form data, validate data, ...) rồi gọi một phương thức của lớp service được inject để lưu dữ liệu. Controller không cần lo về kết nối cơ sở dữ liệu. Cũng như service không cần phải biết request có những thông tin gì.

5. Spring Context

Spring Context mang mọi thứ lại với nhau. Spring Context kế thừa các tính năng của Spring Bean và bổ sung các hỗ trợ cho internationalization (ví dụ như các resource bundle), event propagation, resource loading ... Ngoài ra, Spring Context cũng hỗ trợ các tính năng của Java EE như EJB, JMX và truy cập từ xa cơ bản. Interface ApplicationContext là tiêu điểm của Spring Context.

6. Spring Expression Language (SpEL)

Spring Expression Language là một ngôn ngữ ngắn gọn giúp cho việc cấu hình Spring Framework trở nên linh hoạt hơn

7. Các dự án trong Spring Framework

Spring Framework là một tập hợp của nhiều dự án con. Spring Core như chúng ta đã tìm hiểu là nền tảng của các dự án trong Spring Framework. Một một dự án sẽ đảm nhận một chức năng riêng trong việc xây dựng các ứng dụng doanh nghiệp.

8. Các dự án lớn trong Spring

Spring MVC

Spring MVC được thiết kế dành cho việc xây dựng các ứng dụng nền tảng web. Đây là một dự án chúng ta không thể bỏ qua khi xây dựng các ứng dụng Java web

Spring Data

Cung cấp một cách tiếp cận đúng đắn để truy cập dữ liệu từ cơ sở dữ liệu quan hệ, phi quan hệ, map-reduce và hơn thế nữa.

Spring Security

Dự án này cung cấp các cơ chế xác thực (authentication) và phân quyền (authorization) cho ứng dụng của bạn.

Spring Boot

Spring Boot là một framework giúp phát triển cũng như chạy ứng dụng một cách nhanh chóng.

Spring Batch

Dự án này giúp chúng ta dễ dàng tạo các lịch trình (scheduling) và tiến trình (processing) cho các công việc xử lý theo mẻ (batch jobs).

Spring Integration

Spring Integration là một implementation của Enterprise Integration Patterns (EIP). Dự án này thiết kế một kiến trúc hướng thông điệp hỗ trợ việc tích hợp các hệ thống bên ngoài.

Spring XD

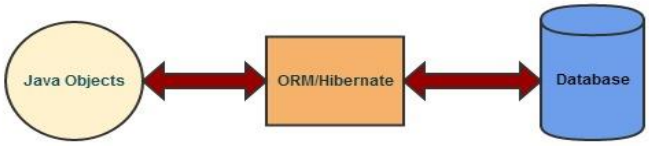
Mục tiêu của dự án này là đơn giản hóa công việc phát triển các ứng dụng Big Data.

Spring Social

Dự án này sẽ kết nối ứng dụng của bạn với các API bên thứ ba của Facebook, Twitter, Linkedin ...

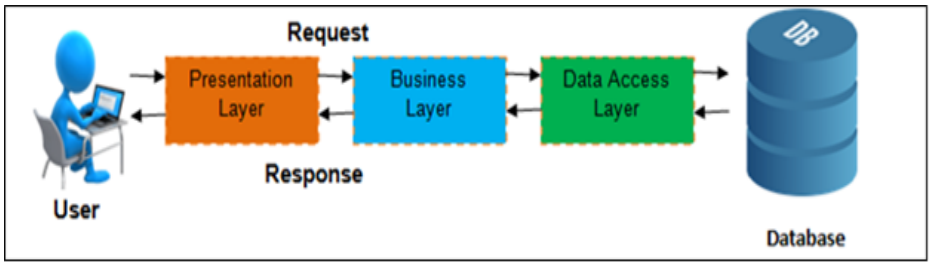
III. Hibernate

1. Giới thiệu về Hibernate

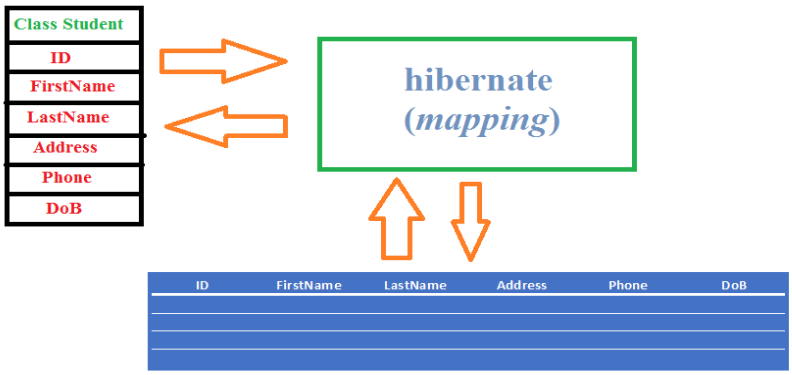


mô hình mapping của hibernate

Mô hình lập trình hiện đại thường chia kiến trúc ứng dụng thành 3 lớp như sau: lớp giao diện người dùng (UI layer), lớp xử lý nghiệp vụ (business layer) và lớp chứa dữ liệu (data layer). Trong đó, business layer có thể bị chia nhỏ thành 2 layer con là business logic layer và persitence layer. Persistence layer chịu trách nhiệm giao tiếp với data. Persistence layer sẽ đảm nhiệm các nhiệm vụ mở kết nối, truy xuất và lưu trữ dữ liệu vào các RDBMS



Mô hình kiến trúc 3 lớp của ứng dụng



ví dụ map Class Students.java với table Student in RDBMS với hibernate

Hibernate là một giải pháp ORM cho Java , một dự án open source chuyên nghiệp, một framework cho pesistence layer.

Hibernate ánh xạ các object Java với các table trong cơ sở dữ liệu và ánh xạ giữa các kiểu dữ liệu trong Java với các kiểu dữ liệu SQL. Hibernate giúp giảm thiểu các công việc liên quan đến nhiệm vụ xử lý dữ liệu thông thường trong phát triển ứng dụng.

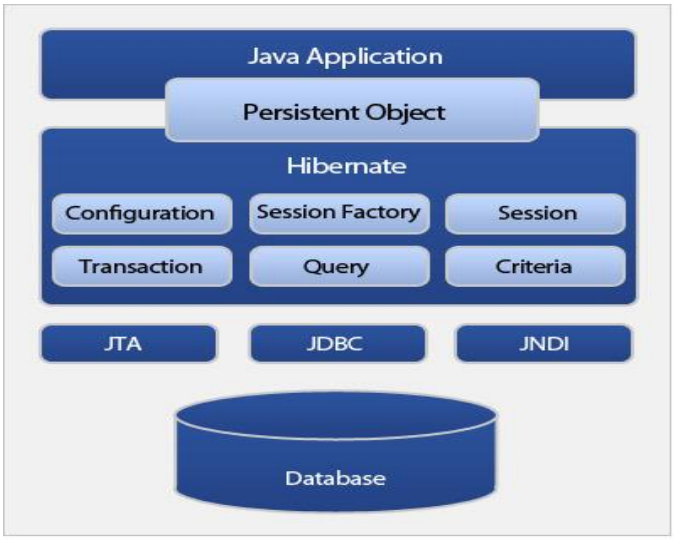
Hibernate nằm giữa các đối tượng Java truyền thống và cơ sở dữ liệu để giải quyết tất cả các công việc trong lớp persistence dựa trên mô hình kỹ thuật ORM.

Ưu điểm của Hibernate

- Hibernate chịu trách nhiệm ánh xạ giữa các lớp Java đến các bảng trong CSDL dùng các file XML
- Cung cấp các API đơn giản để lưu trữ, truy xuất trực tiếp các đối tượng Java và CSDL.
- Nếu có bất kỳ thay đổi nào trong CSDL thì chỉ cần thay đổi file XML.
- Cung cấp đầy đủ các tiện ích, tính năng truy vấn dữ liệu đơn giản, hiệu quả.
- Thao tác, xử lý được các quan hệ phức tạp của các đối tượng trong CSDL.
- Giảm thiểu tối đa sự truy cập đến CSDL nhờ có chiến lược tìm, nạp thông minh.
- Hỗ trợ hầu hết các RDBMS như: MySQL, PostgreSQL, Oracle,…
- Hỗ trợ một số công nghệ như: XDoclet Spring, J2EE, Eclipse plug-ins, Maven,…

2. Kiến trúc Hibernate

Hibernate sử dụng các file cấu hình để cung cấp các dịch vụ và đối tượng persistence cho ứng dụng



Kiến trúc của Hibernate

Hibernate dùng rất nhiều loại Java API có sẵn như JDBC, Java Transaction API (JTA) và Java Naming and Directory Interface (JNDI). JDBC cung cấp các chức năng làm việc với CSDL quan hệ ở mức thô sơ, trừu tượng, cho phép bất kỳ một CSDL nào dùng JDBC driver đều được hỗ trợ bởi Hibernate. JNDI và JTA cho phép hibernate được tích hợp với các server ứng dụng J2EE.

Configuration Object

Configuration object là file cấu hình hoặc file thuộc tính theo yêu cầu của hibernate. Configuration object cung cấp hai thành phần quan trọng sau:

- Database Connection: gồm các tập tin hibernate.properties và hibernate.cfg.xml
- Class Mapping Setup: Tạo sự kết nối giữa các lớp trong java và các bảng trong CSDL

SessionFactory Object

Configuration object được dùng để tạo ra các đối tượng SessionFactory, cho phép các đối tượng session được khởi tạo. SessionFactory thường được tạo ra từ lúc ứng dụng được khởi tạo và được giữ lại về sau.

Session Object

Một session được sử dụng để lấy kết nối vật lí với CSDL. Session được thiết kế để được khởi tạo cho mỗi lần tương tác với CSDL. Các đối tượng persistent được lưu trữ và truy cập thông qua đối tượng Session. Một session không nên mở trong thời gian dài bởi vì lý do không an toàn, mà nên được tạo và hủy chúng khi cần thiết.

Transaction Object

Một transaction đại diện cho một phiên giao dịch với CSDL. Transaction trong hibernate được thực thi bởi một trình quản lý giao dịch (từ JDBC hoặc JTA). Đây là một đối tượng tùy chọn và các ứng dụng hibernate có thể không chọn sử dụng nó, thay vào đó có thể quản lý transaction trong các đoạn code.

Query Object

Các đối tượng Query dùng SQL hoặc Hibernate Query Language (HQL) để truy xuất dữ liệu từ CSDL và tạo ra các đối tượng. Một query thường được sử dụng để ràng buộc các tham số truy vấn, giới hạn số lượng kết quả truy vấn và thực thi câu truy vấn.

Criteria Object

Criteria thường được sử dụng để tạo và thực thi các điều kiện truy vấn hướng đối tượng để truy xuất các đối tượng.

IV. Nguồn tham khảo

- https://spring.io/
- https://en.wikipedia.org/wiki/Hibernate_(framework)
- http://hibernate.org/orm/
- https://toidicodedao.com/2015/03/24/solid-la-gi-ap-dung-cac-nguyen-ly-solid-de-tro-thanh-lap-trinh-vien-code-cung/
- http://sieudaochichcode.com/2017/08/19/solid-tim-hieu-solid-de-tro-thanh-developer-gioi/