

 **SOICT** 25  
ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG


# Bài 10: Các nguyên lý thiết kế

1

1

## Nội dung

- ❖ 1. Nguyên lý GRASP
- ❖ 2. Nguyên lý tri thức tối thiểu
- ❖ 3. Nguyên lý SOLID


 **SOICT** 25  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

2

2

## Nội dung

- ❖ 1. Nguyên lý GRASP
- ❖ 2. Nguyên lý tri thức tối thiểu
- ❖ 3. Nguyên lý SOLID


 **SOICT** 25  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

3

3

## GRASP

- ❖ GRASP - General Responsibility Assignment Software Patterns (or Principles): Nguyên lý/mẫu thiết kế phần mềm phân định trách nhiệm
- ❖ GRASP: tất cả các nguyên lý GRASP đều hướng tới trọng tâm là phân trách nhiệm cho ai
- ❖ GRASP không liên quan tới SOLID

 **SOICT** 25  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

4

4

## GRASP

Có 9 mẫu (nguyên lý) sử dụng trong GRASP

1. Information expert
2. Creator
3. Controller
4. Low coupling
5. High cohesion
6. Indirection
7. Polymorphism
8. Pure fabrication
9. Protected variations- Don't Talk to Strangers

Craig Larman: *Apply UML and Patterns – An Introduction to Object-Oriented Analysis and Design*  
SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

5

5

## Chuyên gia thông tin

- ❖ Cho đối tượng o, những nhiệm vụ nào có thể được giao cho o?
- ❖ Nguyên lý **chuyên gia thông tin**: Trao nhiệm vụ cho lớp có đầy đủ thông tin để hoàn thành nhiệm vụ đó.
- ❖ Lớp có thể đã có sẵn thông tin để trả lời. Hoặc lớp có thể trao đổi với các lớp khác để lấy đủ thông tin cần thiết để thực hiện nhiệm vụ

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

6

6

## Ví dụ

- ❖ Cần lấy tất cả các videos trong VideoStore.
- ❖ Vì VideoStore biết về tất cả videos, trách nhiệm lấy tất cả các videos trong VideoStore được giao cho lớp VideoStore.
- ❖ VideoStore là lớp chuyên gia thông tin

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

7

7

## Ví dụ

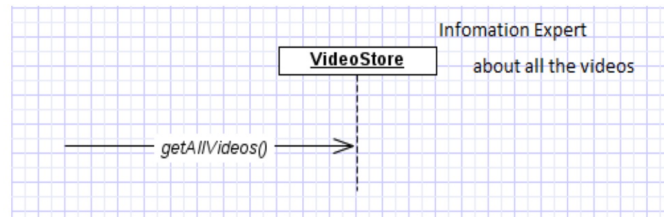


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

8

8

## Ví dụ



9

## Khởi tạo

- ❖ Ai sẽ chịu trách nhiệm khởi tạo một đối tượng?
- ❖ Quyết định dựa trên quan hệ và tương tác giữa các đối tượng
- ❖ B sẽ khởi tạo A nếu:
  - B chứa A
  - B lưu lại A
  - B dùng A như thành phần thiết yếu
  - B khởi tạo dữ liệu cho A



10

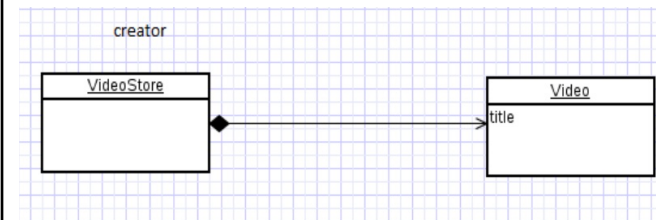
## Ví dụ

- ❖ Xét ví dụ với VideoStore và Video
- ❖ VideoStore có quan hệ kết tập với Video. VideoStore chứa các Video
- ❖ Do đó, ta có thể khởi tạo đối tượng Video trong lớp VideoStore



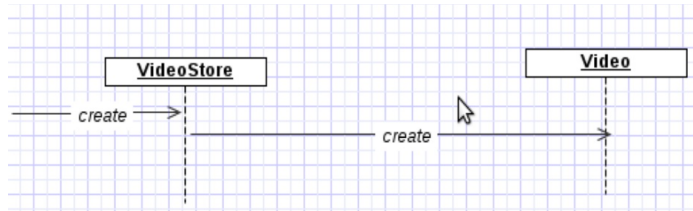
11

## Ví dụ



12

## Ví dụ



13

## Điều khiển

- ❖ Đối tượng nào sẽ chịu trách nhiệm chuyển request từ các đối tượng UI tới các đối tượng nghiệp vụ
- ❖ Khi một yêu cầu đến từ đối tượng tầng UI, mẫu điều khiển giúp chúng ta quyết định đối tượng đầu tiên nhận yêu cầu này là đối tượng nào
- ❖ Đối tượng điều khiển - controller object: nhận yêu cầu từ tầng UI, và điều phối các đối tượng khác ở tầng nghiệp vụ việc thực hiện công việc (Đôi khi phải cần nhiều đối tượng phối hợp thực hiện 1 công việc)



14

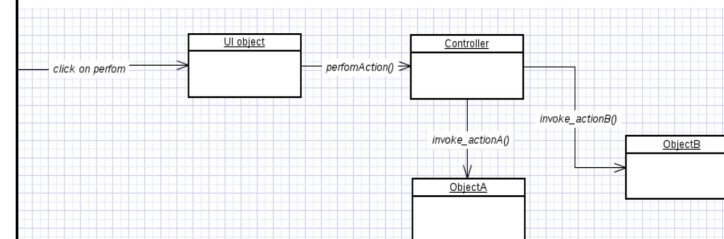
## Điều khiển

- ❖ Một đối tượng sẽ là đối tượng controller nếu
  - Đối tượng đại diện cho toàn bộ hệ thống (facade controller)
  - Đối tượng đại diện cho 1 nghiệp vụ use case, xử lý một chuỗi các thao tác (use case or session controller).
- ❖ Lợi ích
  - Có thể tái sử dụng lớp controller
  - Dễ quản lý trạng thái trong use case
  - Có thể điều khiển chuỗi các hoạt động



15

## Ví dụ



16

## Điều khiển quá tải - Bloated Controllers

- ❖ Lớp Controller là quá tải - bloated, nếu
  - Lớp có quá nhiều trách nhiệm.
    - Giải pháp – thêm các controllers khác
  - Lớp điều khiển xử lý luôn quá nhiều nhiệm vụ thay vì chuyển các nhiệm vụ đó cho các lớp khác
    - Giải pháp – chuyển nhiệm vụ cho các lớp khác



17

## Kết nối lỏng lẻo - Low Coupling

- ❖ Mức độ các đối tượng liên kết với nhau?
- ❖ Coupling – một đối tượng phụ thuộc vào các đối tượng khác.
- ❖ Khi một đối tượng thay đổi, sẽ ảnh hưởng đến các đối tượng phụ thuộc.
- ❖ Kết nối lỏng lẻo - Giảm tác động của thay đổi.
- ❖ Kết nối lỏng lẻo - phân công trách nhiệm để đảm bảo kết nối lỏng lẻo .
- ❖ Giảm thiểu sự phụ thuộc do đó giúp hệ thống dễ bảo trì, hiệu quả và dễ tái sử dụng mã nguồn



18

## Low coupling

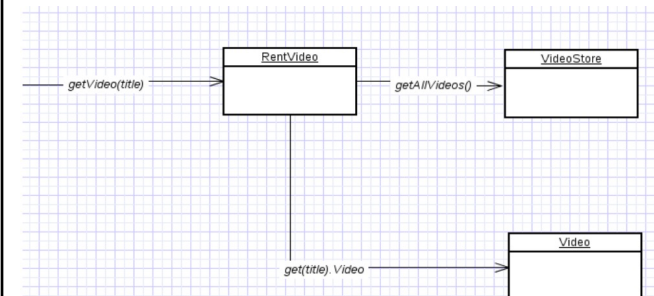
- ❖ 2 lớp là phụ thuộc nếu
  - lớp này liên kết với lớp khác
  - lớp này kế thừa lớp khác Two elements are coupled, if



19

## Ví dụ

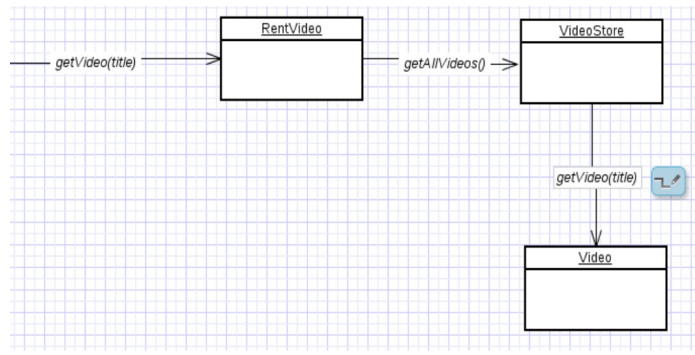
- ❖ RentVideo phụ thuộc vào cả VideoStore và Video  
→ high coupling



20

## Ví dụ

- ❖ VideoStore và Video phụ thuộc vào nhau, Rent phụ thuộc vào VideoStore. Do đó, kết nối là lỏng lẻo hơn (low coupling)



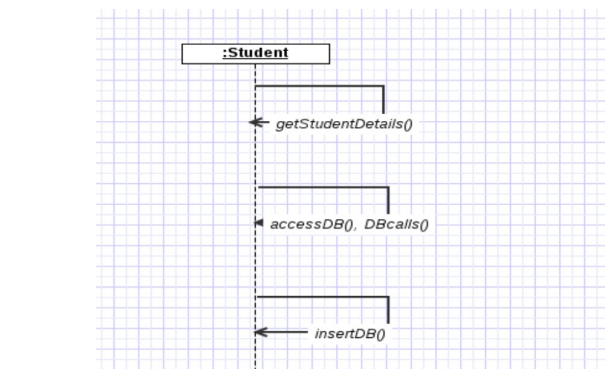
21

## Tính nhất quán cao - High Cohesion

- ❖ Mức độ liên quan giữa các operation trong một lớp?
- ❖ Các chức năng liên quan tới nhau trong một đơn vị quản lý
- ❖ Cần đảm bảo tính high cohesion
- ❖ Xác định rõ ràng nhiệm vụ của phần tử
- ❖ Lợi ích
  - Dễ hiểu dễ bảo trì
  - Tăng tính tái sử dụng mã nguồn
  - Góp phần làm giảm tính coupling

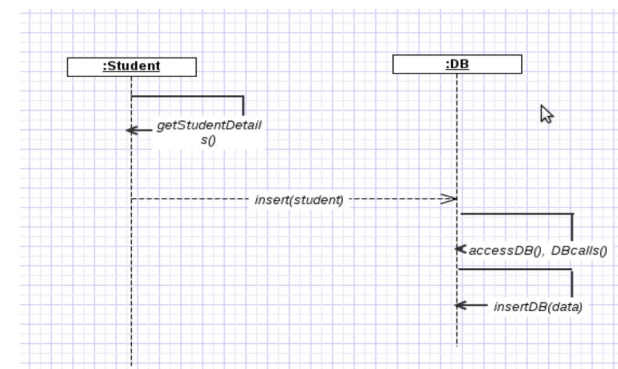
22

## Ví dụ



23

## Ví dụ



24

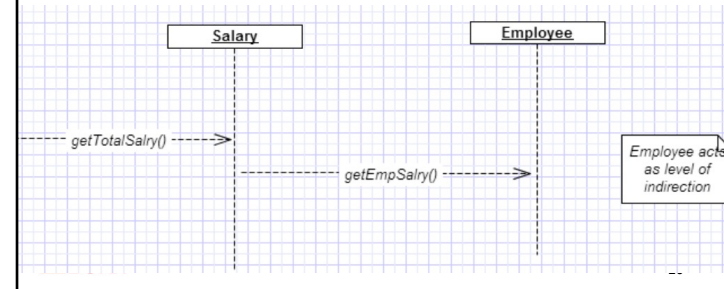
## Gián tiếp - Indirection

- ❖ Cách thức tránh phụ thuộc trực tiếp giữa các phần tử?
- ❖ Indirection sẽ đưa ra một đơn vị trung gian để thực hiện giao tiếp giữa phần tử, do đó các phần tử không bị trực tiếp phụ thuộc vào nhau
- ❖ Lợi ích: low coupling
- ❖ Ví dụ: Adapter, Facade, Observer

25

## Ví dụ

- ❖ Sử dụng đa hình với Employee là phần tử trung gian



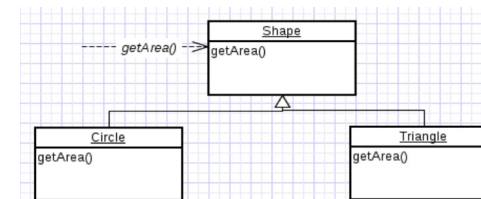
26

## Đa hình - Polymorphism

- ❖ Khi muốn hành vi khác nhau tùy theo kiểu cụ thể của đối tượng thì làm thế nào?
- ❖ Sử dụng đa hình
- ❖ Lợi ích: xử lý đơn giản và dễ dàng

27

## Ví dụ



28

## Pure Fabrication

- ❖ Vấn đề: phân trách nhiệm cho lớp nào, khi mà áp dụng các nguyên lý ở trên lại dẫn đến vi phạm nguyên lý high cohesion và low coupling
- ❖ Giải pháp: tạo một lớp riêng, độc lập và gán trách nhiệm cho lớp đó. Lớp này thường gọi là lớp tiện ích/lớp dịch vụ.

29

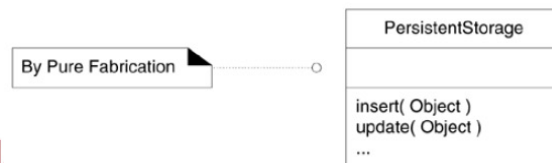
## Ví dụ 1

- ❖ Cần lưu đối tượng lớp **Sale** vào CSDL. Theo nguyên lý **Information Expert**, nhiệm vụ lưu được gán cho chính lớp **Sale**, vì lớp này có tất cả dữ liệu cần lưu. Nhưng
  - Sale có nhiều nhiệm vụ → không đảm bảo tính high cohesion
  - Sale phụ thuộc vào các lớp tiện ích để lưu DB → tăng tính coupling
  - Thao tác save còn lặp lại nhiều lần với các đối tượng khác (như lớp Customer) → không có tính tái sử dụng

30

## Ví dụ 1

- ❖ Giải pháp: tạo một lớp mới (PersistentStorage) chịu trách nhiệm lưu trữ đối tượng Sale vào CSDL
- ❖ Lợi ích:
  - Lớp Sale vẫn đảm bảo có thiết kế tốt, có tính high cohesion và low coupling.
  - Lớp PersistentStorage có tính cohesion khá cao, nhiệm vụ là lưu trữ/thao tác với CSDL
  - Lớp PersistentStorage tổng quát, có tính tái sử dụng cao



31

## Ví dụ 2

```
interface IForeignExchange {
    List<ConversionRate> getConversionRates();
}

class ConversionRate {
    private String from;
    private String to;
    private double rate;
    public ConversionRate(String from, String to, double rate) {
        this.from = from;
        this.to = to;
        this.rate = rate;
    }
}

public class ForeignExchange implements IForeignExchange {
    public List<ConversionRate> getConversionRates() {
        List<ConversionRate> rates = ForeignExchange.getConversionRatesFromExternalApi();
        return rates;
    }

    private static List<ConversionRate> getConversionRatesFromExternalApi() {
        // Communication with external API. Here is only mock.
        List<ConversionRate> conversionRates = new ArrayList<ConversionRate>();
        conversionRates.add(new ConversionRate("USD", "EUR", 0.88));
        conversionRates.add(new ConversionRate("EUR", "USD", 1.13));
        return conversionRates;
    }
}
```

32



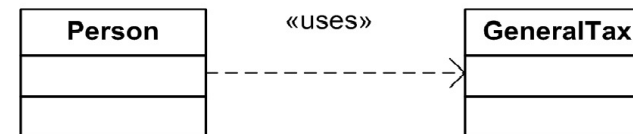
### Thích ứng với các thay đổi - Protected Variation

- ❖ Vấn đề: Thiết kế 1 thành phần ntn để khi thành phần đó thay đổi, ít gây ảnh hưởng nhất không mong muốn nhất tới các thành phần khác
- ❖ Giải pháp: Xác định các điểm tương lai sẽ có sự thay đổi/không ổn định, tạo interface tương ứng bọc lại



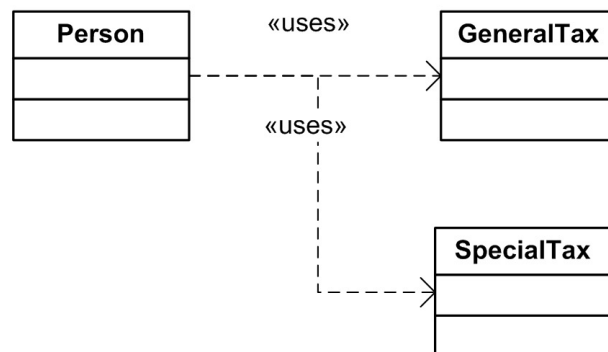
33

### Ví dụ (1/3)



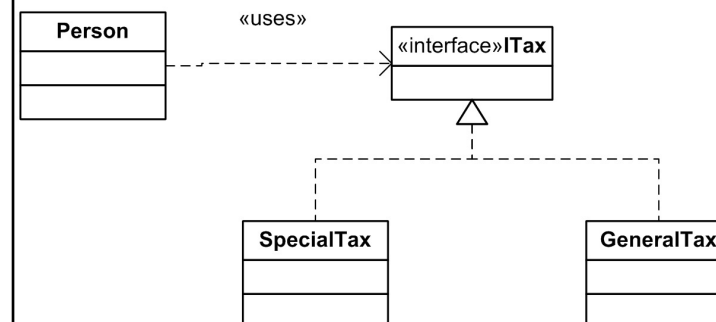
34

### Ví dụ (2/3)



35

### Ví dụ (3/3)



36

## Nội dung

- ❖ 1. Nguyên lý GRASP
- ❖ **2. Nguyên lý tri thức tối thiểu**
- ❖ 3. Nguyên lý SOLID

## Nguyên lý Demeter - nguyên lý tri thức tối thiểu (Karl Lieberherr)

- ❖ Một đối tượng khi tương tác với các đối tượng khác, chỉ nên biết ít nhất có thể về nội bộ cấu trúc của các đối tượng đó (giúp đảm bảo tính low cohesion)
  - Ý tưởng ... “chỉ nói chuyện với người bạn trực tiếp của bạn”
- ❖ Ví dụ code tồi:  

```
general.getColonel().getMajor(m).getCaptain(cap)
    .getSergeant(ser).getPrivate(name).digFoxHole();
```
- ❖ Ví dụ code tốt  

```
general.superviseFoxHole(m, cap, ser, name);
```

## Nguyên lý tri thức tối thiểu

- ❖ Trong phương thức M của đối tượng O chỉ được gọi các phương thức của các loại đối tượng sau:
  - Chính đối tượng O
  - Các đối tượng trong tham số của M
  - Các đối tượng được tạo/khởi tạo trong M
  - Các đối tượng thuộc tính của O

Lưu ý: không cần máy móc tuân thủ trong mọi trường hợp! Nhưng cần cân nhắc, sẽ có thiết kế tốt hơn

## Không đảm bảo nguyên lý tri thức tối thiểu

- objectA.getObjectB().getObjectC().doSomething();
- ❖ objectA về sau có thể sẽ không còn tham chiếu tới ObjectB nữa.
  - ❖ Phương thức doSomething() trong ObjectC có thể sẽ không còn tồn tại nữa.
  - ❖ Sẽ gặp các loại lỗi như NullPointerException, NoMethodError nếu như ObjectB và ObjectC bị null.
  - ❖ Khi đóng gói objectA để tái sử dụng, bạn sẽ cần phải kèm ObjectB, ObjectC với nó => Sự phụ thuộc lẫn nhau giữa các thành phần trong hệ thống tăng cao. (tightly coupled)

## Lợi ích

- ❖ Class sẽ loosely coupled hơn, những thành phần trong hệ thống sẽ ít phụ thuộc nhau hơn.
- ❖ Đóng gói và tái sử dụng sẽ dễ dàng hơn.
- ❖ Việc test sẽ dễ hơn nhiều, phần setup của test sẽ đơn giản hơn.
- ❖ Ít lỗi hơn.

41

## Ví dụ 1

```
public class Customer {  
    private String firstName;  
    private String lastName;  
    private Wallet myWallet;  
  
    public String getFirstName(){  
        return firstName;  
    }  
    public String getLastName(){  
        return lastName;  
    }  
    public Wallet getWallet(){  
        return myWallet;  
    }  
}
```

42

## Ví dụ 1

```
public class Wallet {  
    private float value;  
    public float getTotalMoney() {  
        return value;  
    }  
    public void setTotalMoney(float newValue) {  
        value = newValue;  
    }  
    public void addMoney(float deposit) {  
        value += deposit;  
    }  
    public void subtractMoney(float debit) {  
        value -= debit;  
    }  
}
```

43

## Ví dụ 1

```
// code from some method inside the Paperboy class...  
payment = 2.00; // "I want my two dollars!"  
Wallet theWallet = myCustomer.getWallet();  
if (theWallet.getTotalMoney() > payment) {  
    theWallet.subtractMoney(payment);  
} else {  
    // come back later and get my money  
}
```

*Is this Bad? Why?*

44

## Ví dụ 1 – Cải tiến

```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public float getPayment(float bill) {
        if (myWallet != null) {
            if (myWallet.getTotalMoney() > bill) {
                theWallet.subtractMoney(payment);
                return payment;
            }
        }
    }
}
```

45

## Ví dụ 1 – Cải tiến

```
// code from some method inside the Paperboy class...
payment = 2.00; // "I want my two dollars!"
paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
    // say thank you and give customer a receipt
} else {
    // come back later and get my money
}
```

*Why Is This Better?*

46

## Ví dụ 2

```
public class Band {
    private Singer singer;
    private Drummer drummer;
    private Guitarist guitarist;
}
```

47

## Ví dụ 2

```
class TourPromoter {
    public String makePosterText(Band band) {
        String guitaristsName = band.getGuitarist().getName();
        String drummersName = band.getDrummer().getName();
        String singersName = band.getSinger().getName();
        StringBuilder posterText = new StringBuilder();

        posterText.append(band.getName())
        posterText.append(" featuring: ");
        posterText.append(guitaristsName);
        posterText.append(", ");
        posterText.append(singersName);
        posterText.append(", ")
        posterText.append(drummersName);
        posterText.append(", ")
        posterText.append("Tickets £50.");

        return posterText.toString();
    }
}
```

48

## Ví dụ 2 – Cải tiến

```
public class Band {
    private Singer singer;
    private Drummer drummer;
    private Guitarist guitarist;

    public String[] getMembers() {
        return {
            singer.getName(),
            drummer.getName(),
            guitarist.getName();
        }
    }
}
```

## Ví dụ 2 – Cải tiến

```
public class TourPromoter {
    public String makePosterText(Band band) {
        StringBuilder posterText = new StringBuilder();

        posterText.append(band.getName());
        posterText.append(" featuring: ");
        for(String member: band.getMembers()) {
            posterText.append(member);
            posterText.append(", ");
        }
        posterText.append("Tickets: £50");

        return posterText.toString();
    }
}
```

## Nội dung

- ❖ 1. Nguyên lý GRASP
- ❖ 2. Nguyên lý tri thức tối thiểu
- ❖ **3. Nguyên lý SOLID**

## SOLID

*“Principles Of OOD”, Robert C. Martin (“Uncle BOB”)*

- S – Single-responsibility principle
- O – Open-closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – Dependency Inversion Principle

### 3.1. Nguyên lý một nhiệm vụ Single-responsibility Principle

- ❖ Mỗi lớp chỉ có một và duy nhất một lý do để thay đổi
- ❖ Cách nói khác: mỗi lớp chỉ có một nhiệm vụ duy nhất
- ❖ Thảo luận:
  - Mỗi lớp chỉ có một phương thức?
  - Tại sao mỗi lớp chỉ nên có một nhiệm vụ?

53

### Ví dụ 1

```
package test;
public class UserSettingService {
    public void
    changeEmail(User user) {
        if (checkAccess(user)) {
            //Grant option to change
        }
    }
    public boolean
    checkAccess(User user) {
        //Verify if the user is valid.
    }
}
```

54

### Ví dụ 1 - Mã nguồn sửa đổi

```
public class UserSettingService {
    public void changeEmail(User user) {
        if (SecurityService.checkAccess(user)) {
            //Grant option to change
        }
    }
}

public class SecurityService {
    public boolean checkAccess(User user) {
        //check the access.
    }
}
```

55

### Ví dụ 2

```
public class Employee{
    private String employeeId;
    private String name;
    private String address;
    private Date dateOfJoining;
    public boolean isPromotionDueThisYear(){
        //promotion logic implementation
    }
    public Double calcIncomeTaxForCurrentYear(){
        //income tax logic implementation
    }
    //Getters & Setters for all the private attributes
}
```

56

## Ví dụ 2 - Mã nguồn sửa đổi

```
public class HRPromotions{
    public boolean isPromotionDueThisYear(Employee emp){
        /*promotion logic implementation using the employee information
        passed*/
    }
}

public class FinITCalculations{
    public Double calcIncomeTaxForCurrentYear(Employee emp){
        /*income tax logic implementation using the employee information passed
        */
    }
}

public class Employee{
    private String employeeId;
    private String name;
    private String address;
    private Date dateOfJoining;
    //Getters & Setters for all the private attributes
}
```

57

## 3. 2. Nguyên lý đóng mở Open-closed Principle

- ❖ Các thực thể phần mềm (lớp, module, phương thức, ...) nên là **MỞ** cho các mở rộng nhưng **ĐÓNG** cho các sửa đổi (*open for extension, but closed for modification*)
  - “Open for extension”: một module (class) phải cung cấp các điểm mở rộng, cho phép thay đổi hành vi của nó
  - Closed for modification”: Mã nguồn của module không cần thay đổi để cài đặt sự mở rộng đó

58

58

## Ví dụ - HealthInsuranceSurveyor

```
public class HealthInsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("Validating ...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

59

59

## ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyor) {
        if(surveyor.isValidClaim()){
            System.out.println("Valid claim. Processing claim for approval....");
        }
    }
}
```

60

60

## ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("Valid claim. Processing ...");
        }
    }
    public void processVehicleClaim (VehicleInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("Valid claim. Processing ...");
        }
    }
}
```

61

## Mã nguồn sửa đổi

```
public abstract class InsuranceSurveyor {
    public abstract boolean isValidClaim();
}

public class HealthInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}

public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("VehicleInsuranceSurveyor: Validating claim...");
        /*Logic to validate vehicle insurance claims*/
        return true;
    }
}
```

62

## ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processClaim(InsuranceSurveyor
surveyor){
        if(surveyor.isValidClaim()){
            System.out.println("Valid claim. Processing ...");
        }
    }
}
```

63

## ClaimApprovalManagerTest

```
public class ClaimApprovalManagerTest {
    @Test
    public void testProcessClaim() throws Exception {
        HealthInsuranceSurveyor healthInsuranceSurveyor =
            new HealthInsuranceSurveyor();
        ClaimApprovalManager claim = new ClaimApprovalManager();
        claim1.processClaim(healthInsuranceSurveyor);

        VehicleInsuranceSurveyor vehicleInsuranceSurveyor =
            new VehicleInsuranceSurveyor();
        ClaimApprovalManager claim2 = new ClaimApprovalManager();
        claim2.processClaim(vehicleInsuranceSurveyor);
    }
}
```

64



## Ví dụ 2

```
public class Rectangle{
    private double length;
    private double width;
}

public class AreaCalculator{
    public double calculateRectangleArea(Rectangle
rectangle) {
    return rectangle.getLength() *rectangle.getWidth();
}
}
```



## Thêm lớp Circle

```
public class Circle{
    private double radius;
}

public class AreaCalculator{
    public double calculateRectangleArea(Rectangle
rectangle){
    return rectangle.getLength() *rectangle.getWidth();
}
    public double calculateCircleArea(Circle circle){
    return 3.14159*circle.getRadius()*circle.getRadius();
}
}
```



## Mã nguồn sửa đổi

```
public interface Shape{
    public double calculateArea();
}

public class Rectangle implements Shape{
    double length;
    double width;
    public double calculateArea(){
    return length * width;
}
}

public class Circle implements Shape{
    public double radius;
    public double calculateArea(){
    return 3.14159 *radius*radius;
}
}
```

## AreaCalculator

```
public class AreaCalculator{
    public double calculateShapeArea(Shape shape){
    return shape.calculateArea();
}
}
```

