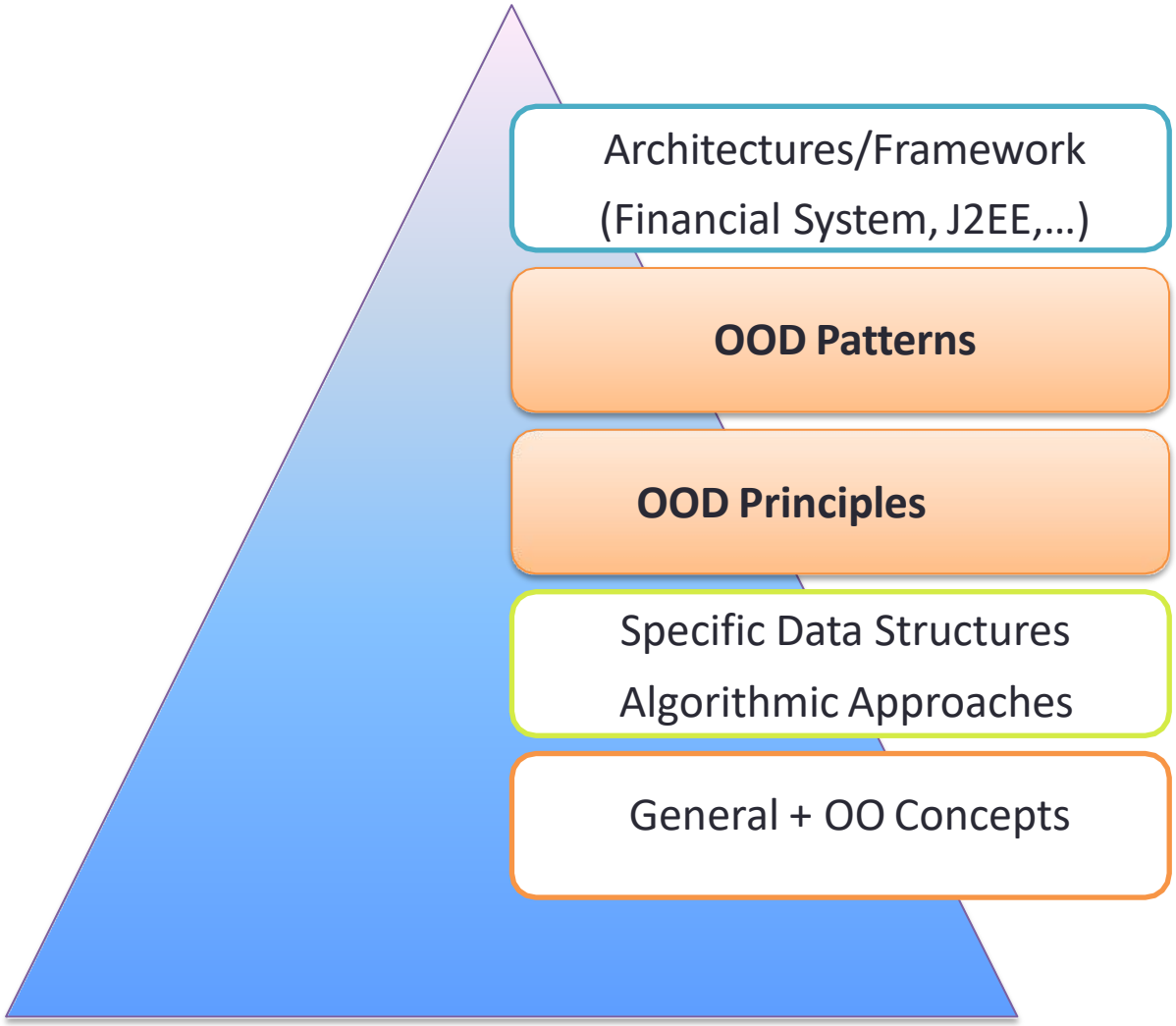
ITSS SOFTWARE DEVELOPMENT

11. DESIGN PRINCIPLES

Review: Design levels



Nội dung

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

Nội dung

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

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

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*

Chuyên gia thông tin

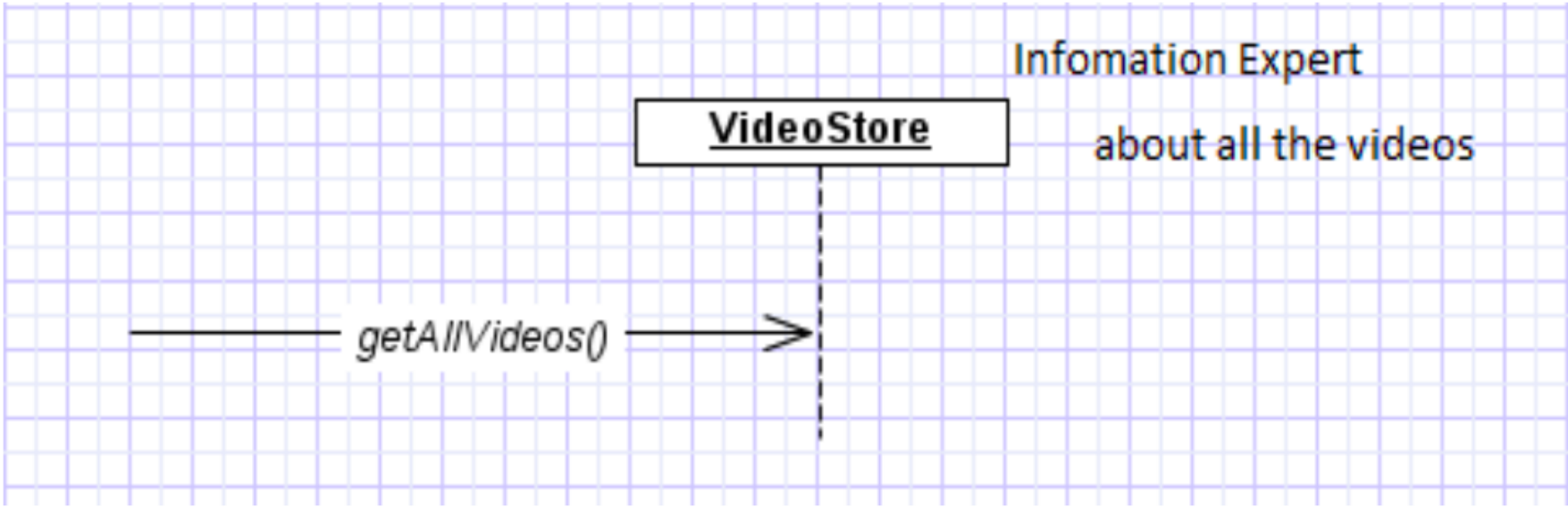
- Cho đối tượng o, những nhiệm vụ nào có thể được giao cho o?
- Nguyên lý **chuyên gia thông tin**: Trao nhiệm vụ cho lớp có đầy đủ thông tin để hoàn thành nhiệm vụ đó.
- Lớp có thể đã có sẵn thông tin để trả lời. Hoặc lớp có thể trao đổi với các lớp khác để lấy đủ thông tin cần thiết để thực hiện nhiệm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

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

Ví dụ

Ví dụ


Infomation Expert
VideoStore
about all the videos
getAllVideos()

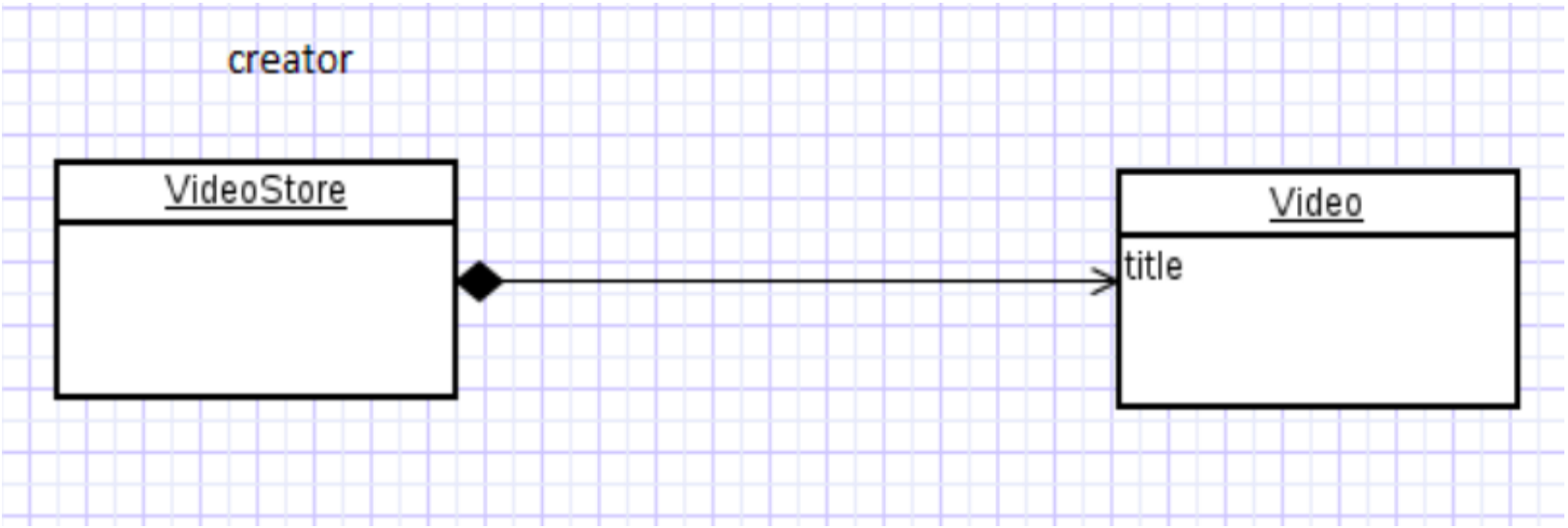
Khởi tạo

- 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

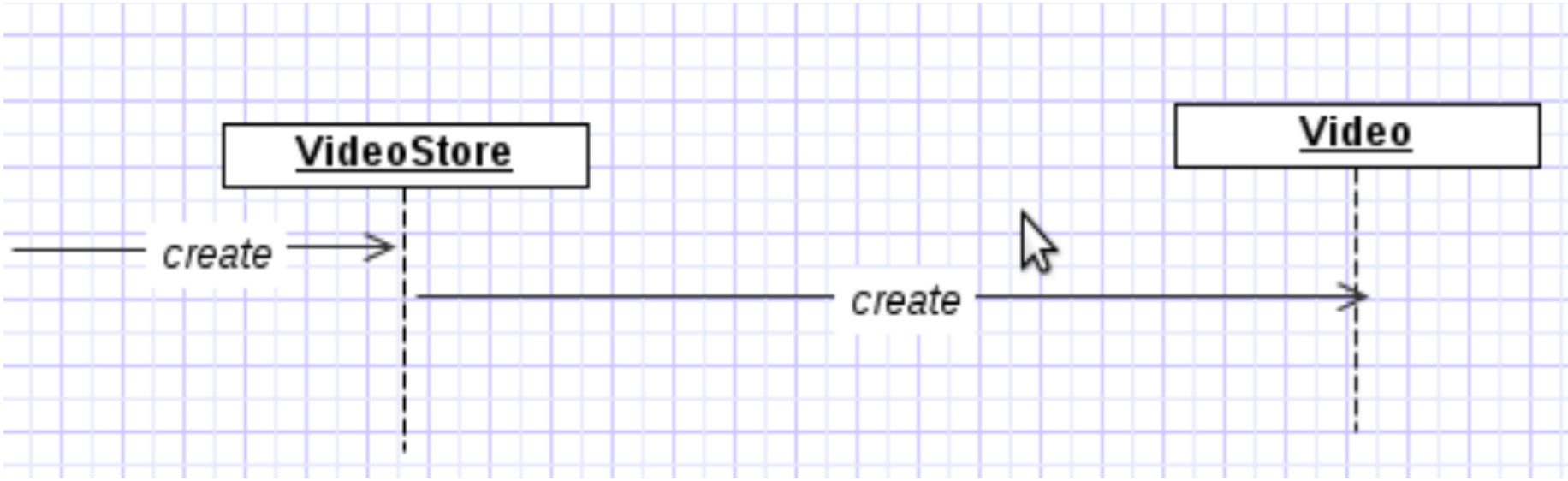
Ví dụ

- 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

Ví dụ



Ví dụ



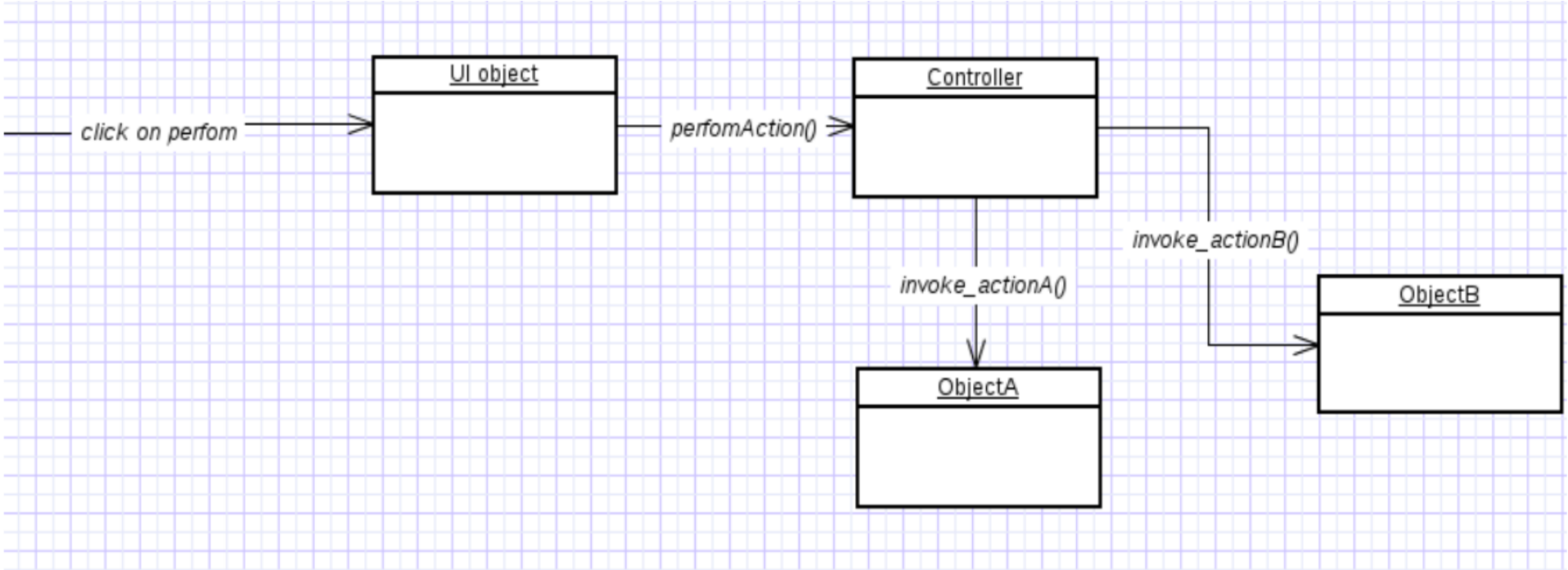
Điều khiển

- 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)

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

Ví dụ


UI object
click on perfom
perfomAction()
Controller
invoke_actionB()
invoke_actionA()
ObjectB
ObjectA

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

Kết nối lỏng lẻo - Low Coupling

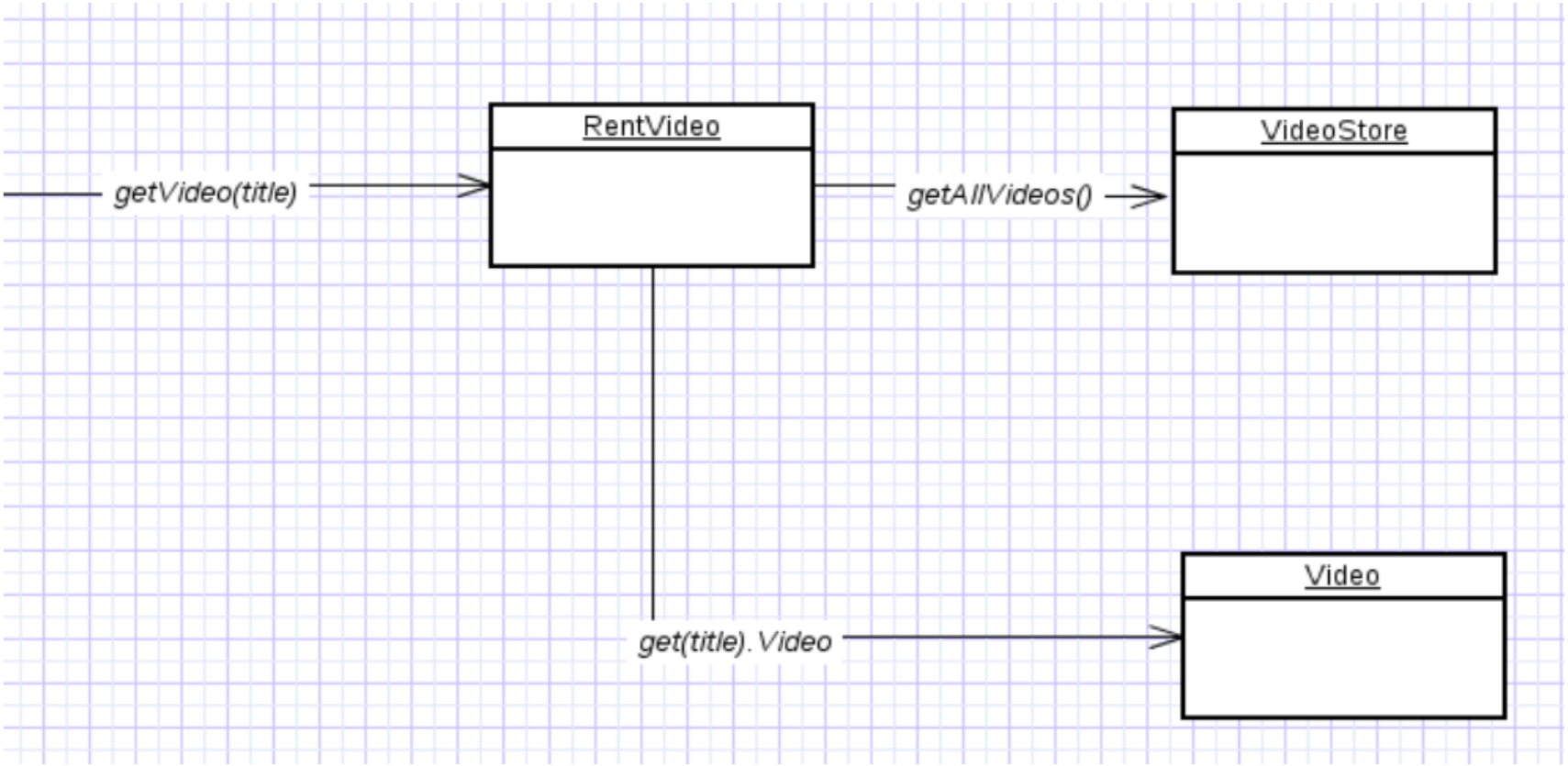
- Mức độ các đối tượng liên kết với nhau?
- Coupling – một đối tượng phụ thuộc vào các đối tượng khác.
- Khi một đối tượng thay đổi, sẽ ảnh hưởng đến các đối tượng phụ thuộc.
- Kết nối lỏng lẻo - Giảm tác động của thay đổi.
- Kết nối lỏng lẻo - phân công trách nhiệm để đảm bảo kết nối lỏng lẻo .
- Giảm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

Low coupling

- 2 lớp là phụ thuộc nếu
  - lớp này liên kết với lớp khác
  - lớp này kế thừa lớp khác

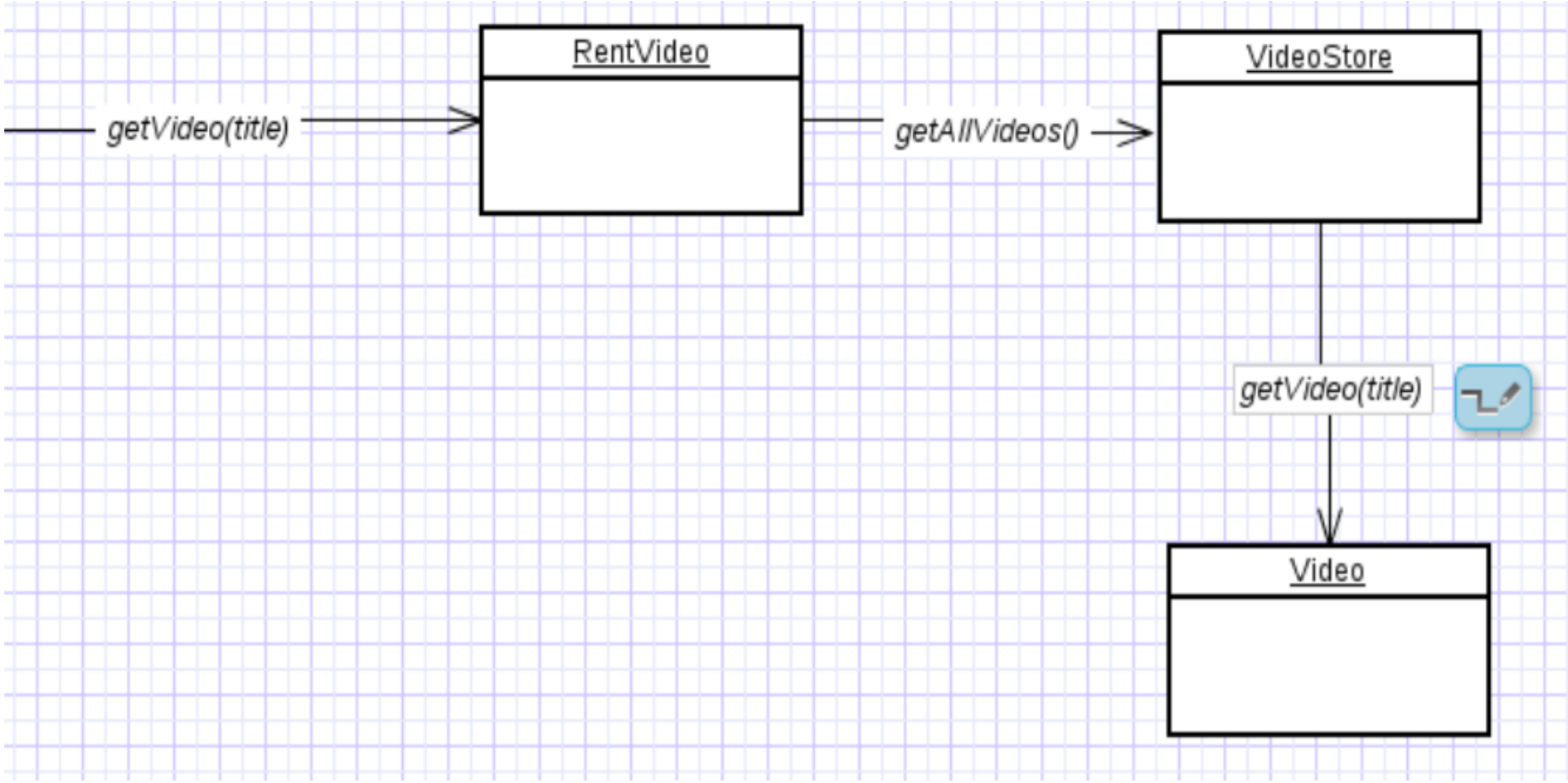
Ví dụ

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



Ví dụ

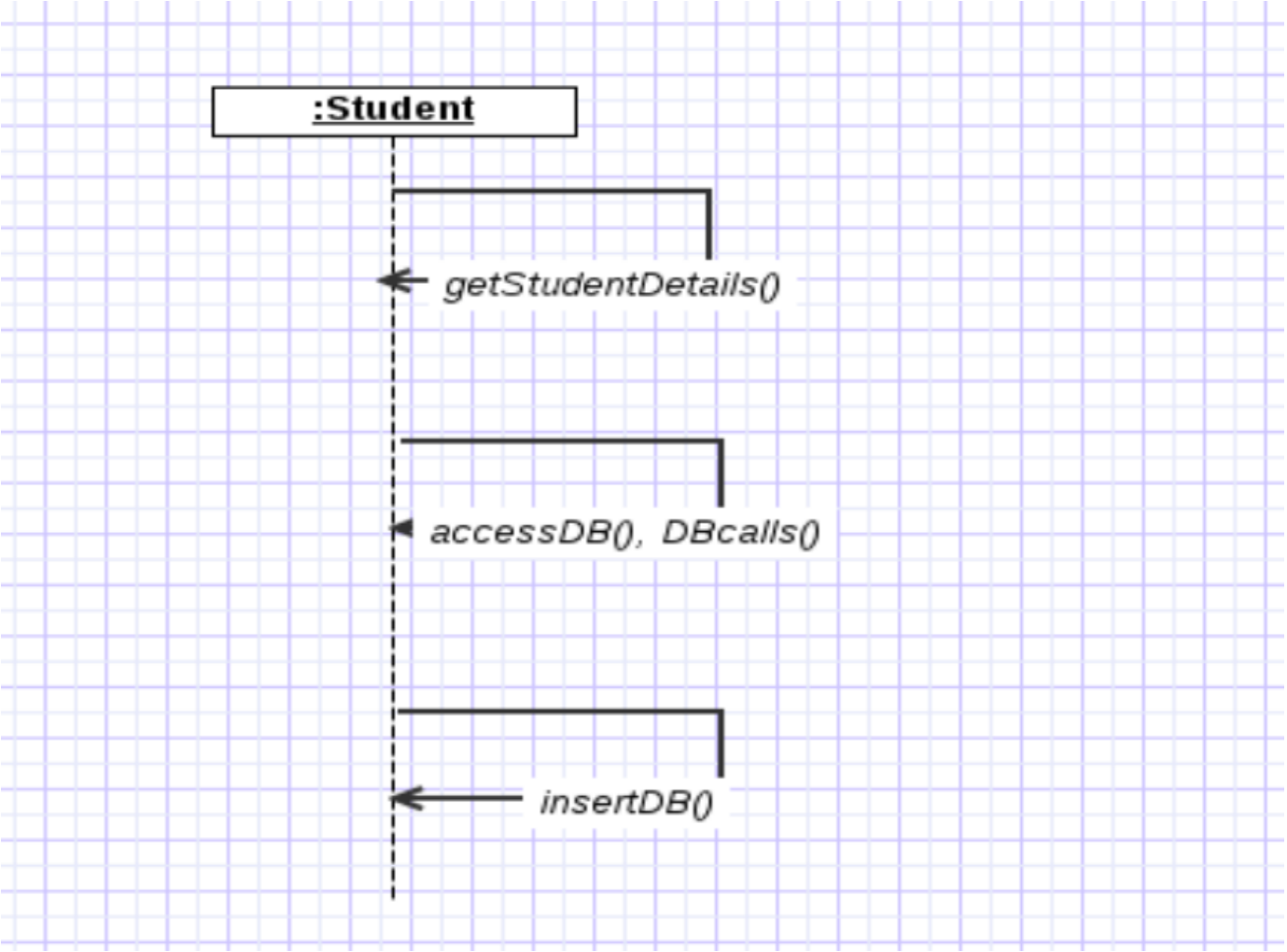
- VideoStore và Video phụ thuộc vào nhau, Rent phụ thuộc vào VideoStore. Do đó, kết nối là lỏng lẻo hơn (low



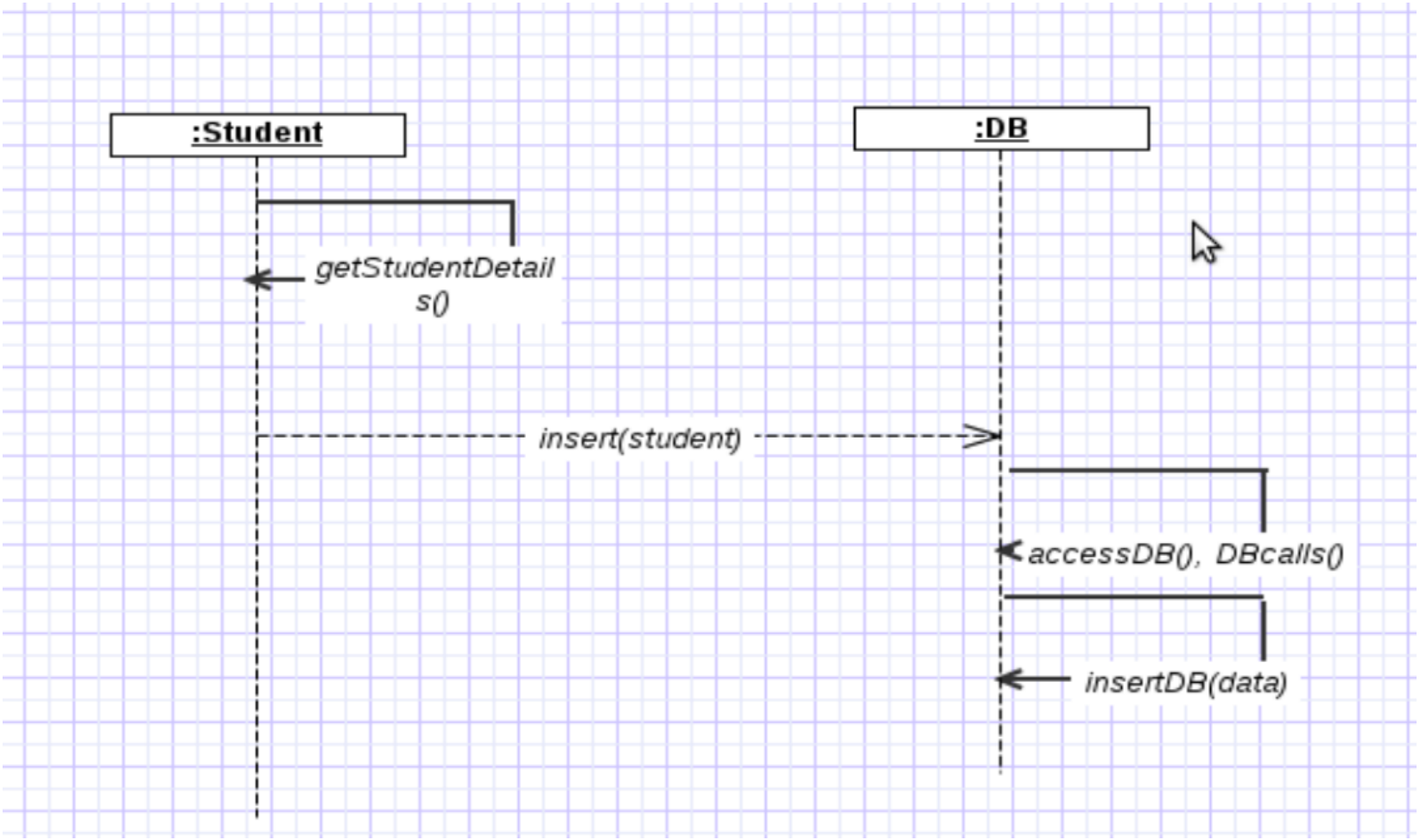
Tính nhất quán cao - High Cohesion

- 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

Ví dụ



Ví dụ

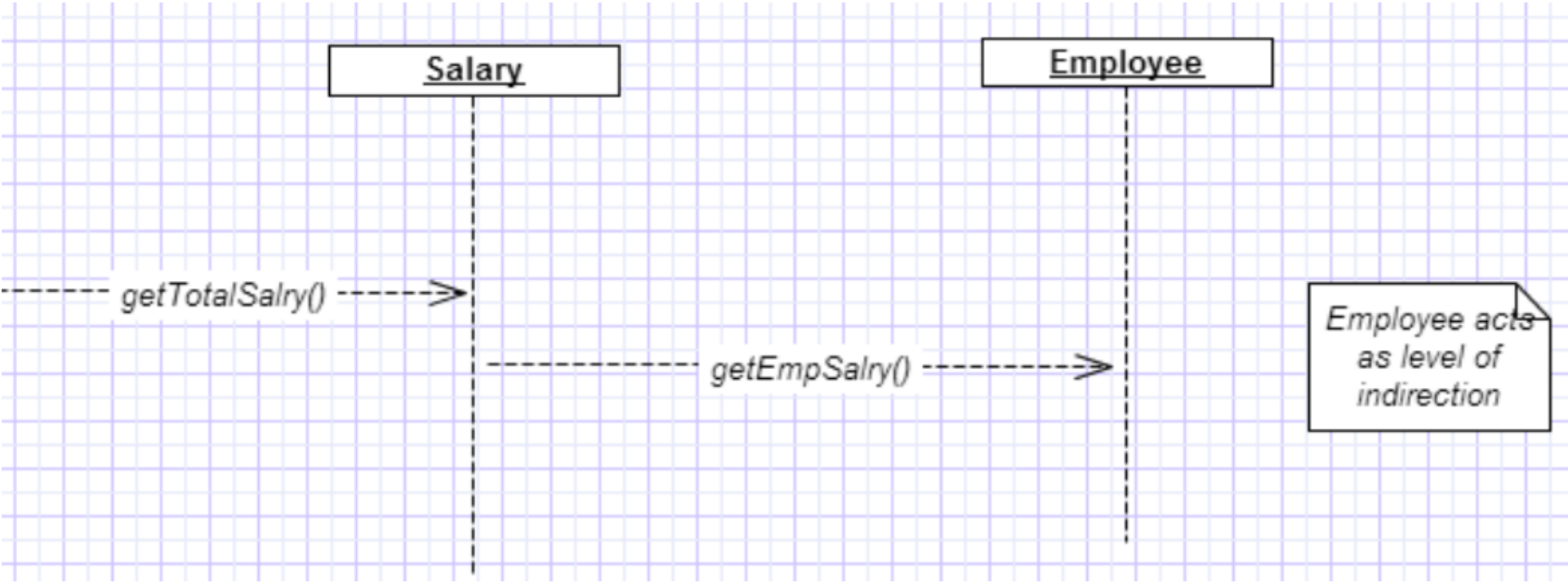

:Student
:DB
getStudentDetail
s()
insert(student)
accessDB(), DBcalls()
insertDB(data)

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, Obserever

Ví dụ

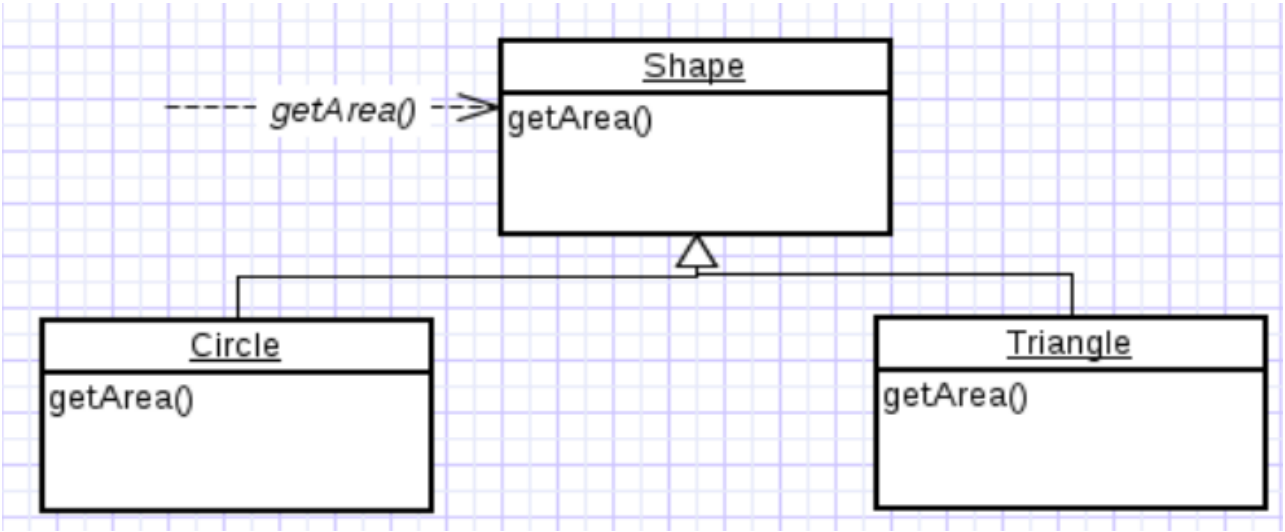
- Sử dụng đa hình với Employee là phần tử trung gian



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

Ví dụ



Pure Fabrication

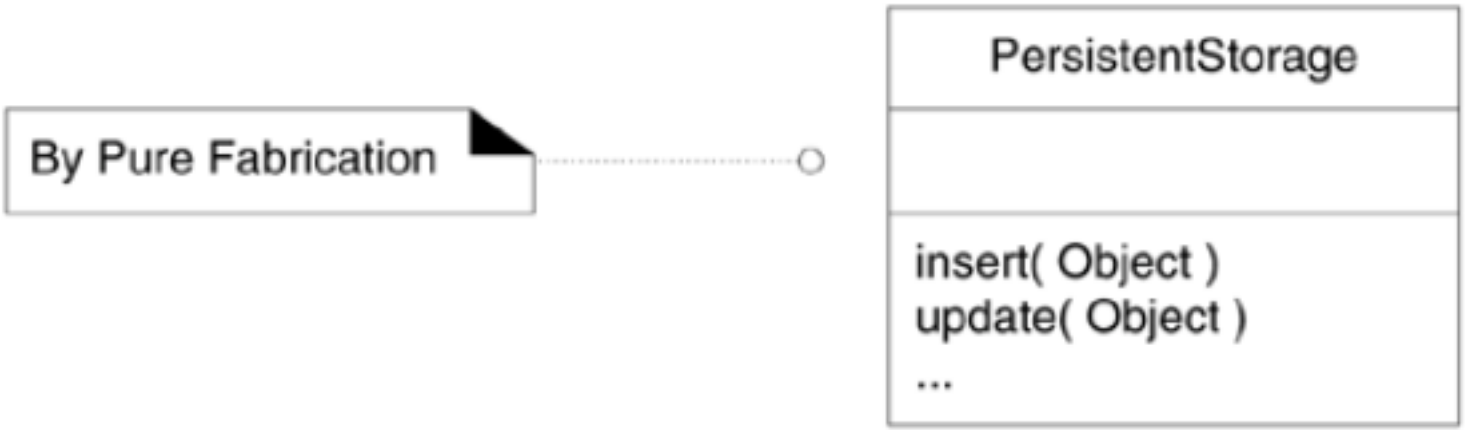
- Vấn đề: phân trách nhiệm cho lớp nào, khi mà áp dụng các nguyên lý ở trên lại dẫn đến vi phạm nguyên lý high cohesion và low coupling
- Giải pháp: tạo một lớp riêng, độc lập và gán trách nhiệm cho lớp đó. Lớp này thường gọi là lớp tiện ích/lớp dịch vụ.

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

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



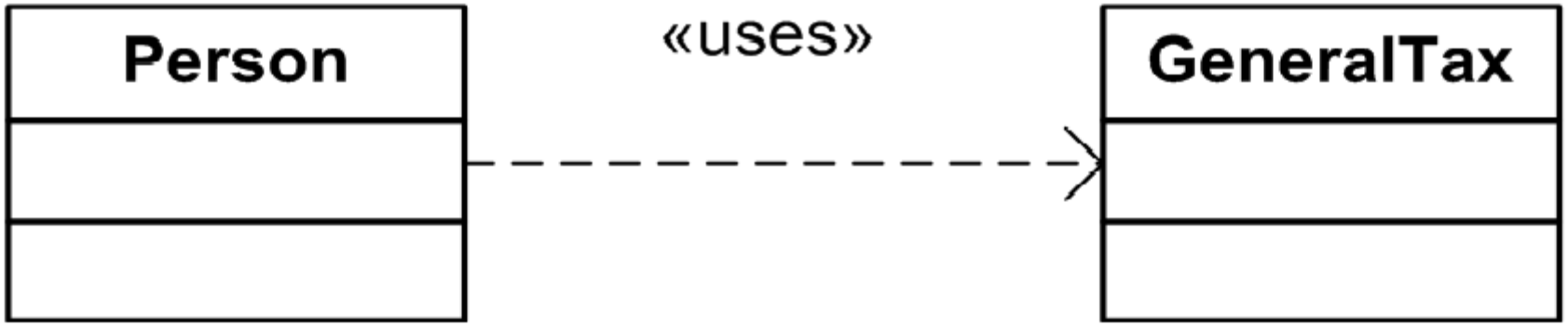
Ví dụ 2

```
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;
    }
}
```

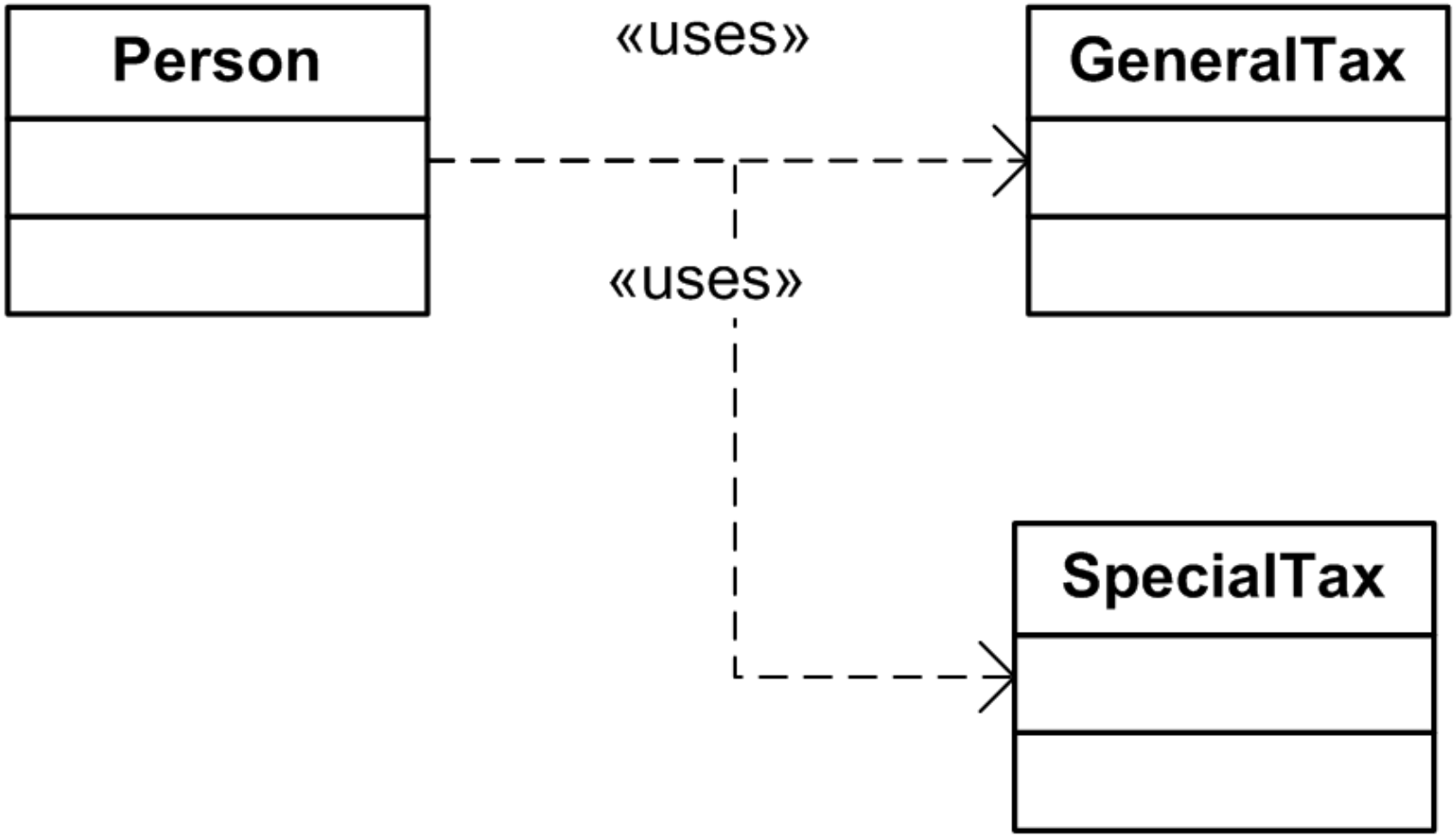
Thích ứng với các thay đổi - Protected Variation

- 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

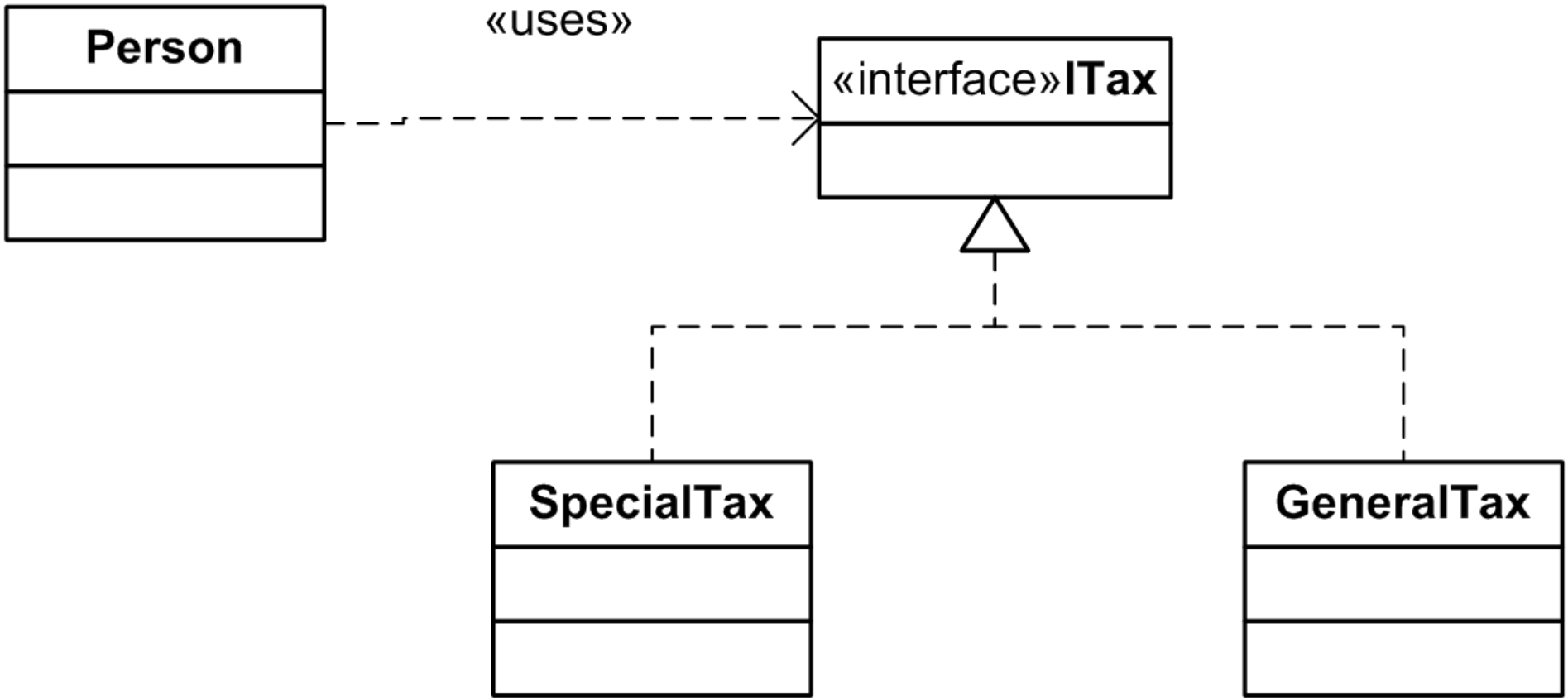
Ví dụ (1/3)



Ví dụ (2/3)



Ví dụ (3/3)



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 nhưNullPointerException, 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.

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;
   }
}
```

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;
   }
}
```

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?

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;
                     }
            }
   }
}
```

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?

Ví dụ 2

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

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();
  }
}
```

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**

S.O.L.I.D Principles of OOD

- SRP: The Single Responsibility Principle
- OCP: The Open Closed Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

Principles of OO Class Design

SRP: The Single Responsibility Principle

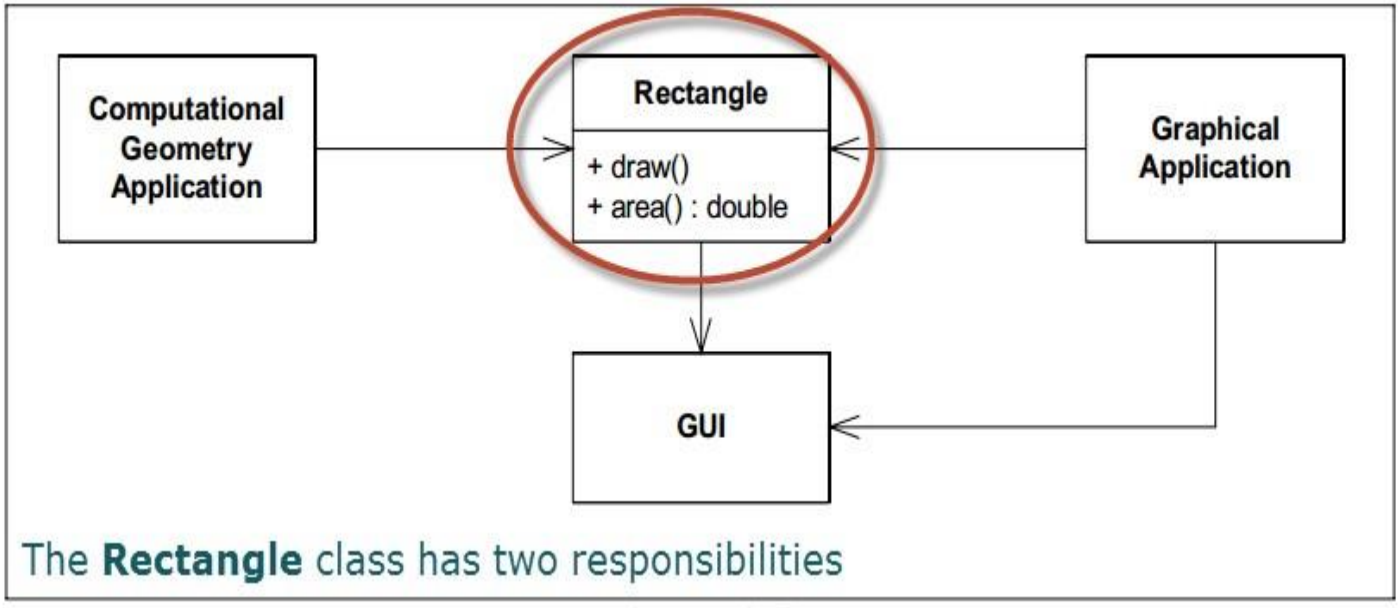
"There should never be more than one reason for a class to change"

Or

"A class should have one, and only one type of responsibility."

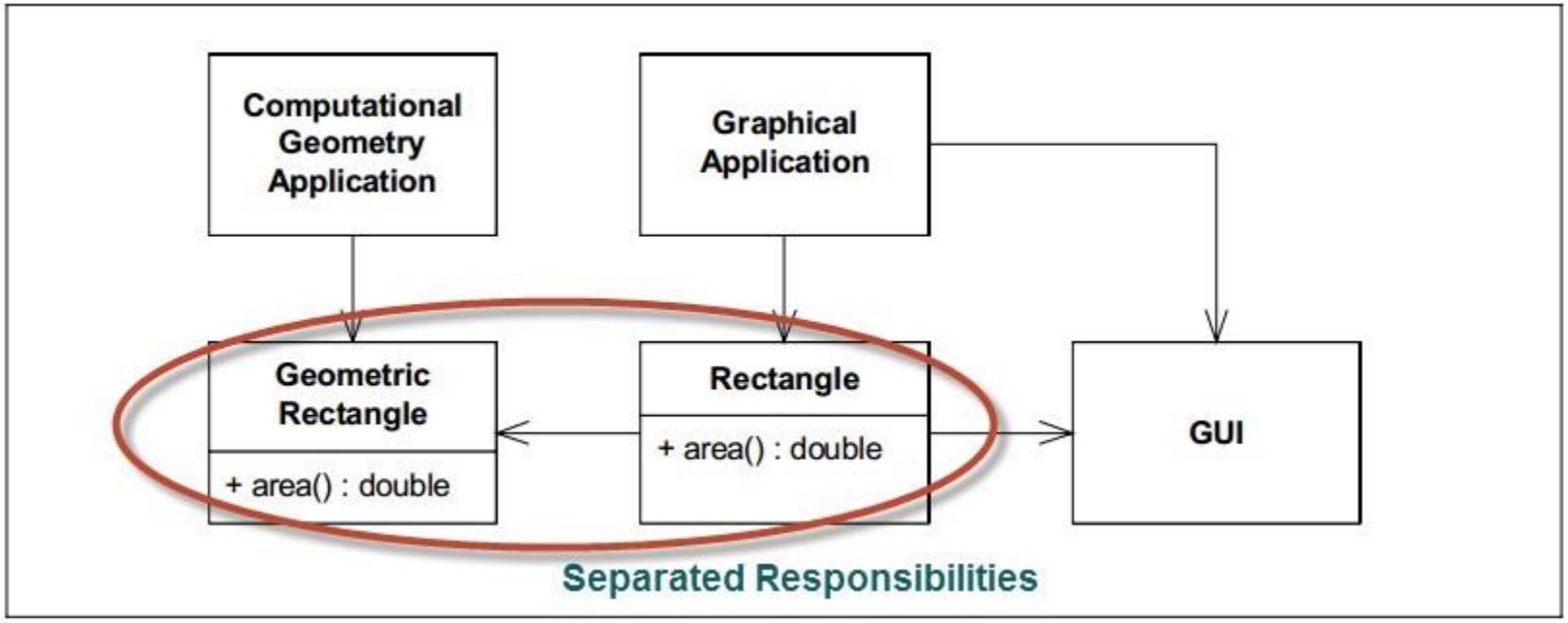
Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- Two applications are using this Rectangle class:
  - Computational Geometry Application uses this class to calculate the Area
  - Graphical Application uses this class to draw a Rectangle in the UI



Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- A better design is to separate the two responsibilities into two completely different classes



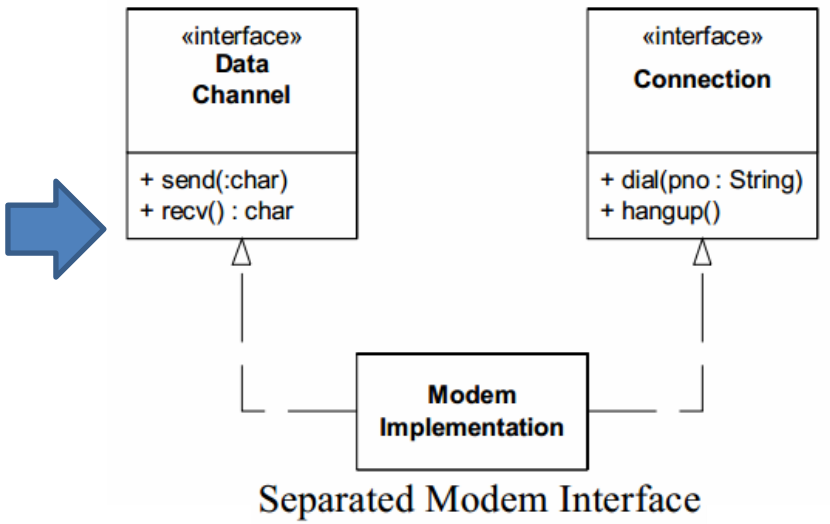
- Why is it important to separate these two responsibilities into separate classes?

Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- What is a Responsibility?
  - A reason for change
- "Modem" sample
  - dial & hangup functions for managing connection
  - send & recv functions for data communication

➔ Should separate into 2 repositories!

```
Modem.java -- SRP Violation
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



Separated Modem Interface

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.
   }
}
```

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.
  }
}
```

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
}
```

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
}
```

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

Principles of OO Class Design
OCP: The Open Closed Principle

"Software entities(classes, modules, functions, etc.) should be open for extension, but closed for modification."

Bertrand Meyer, 1988

Or

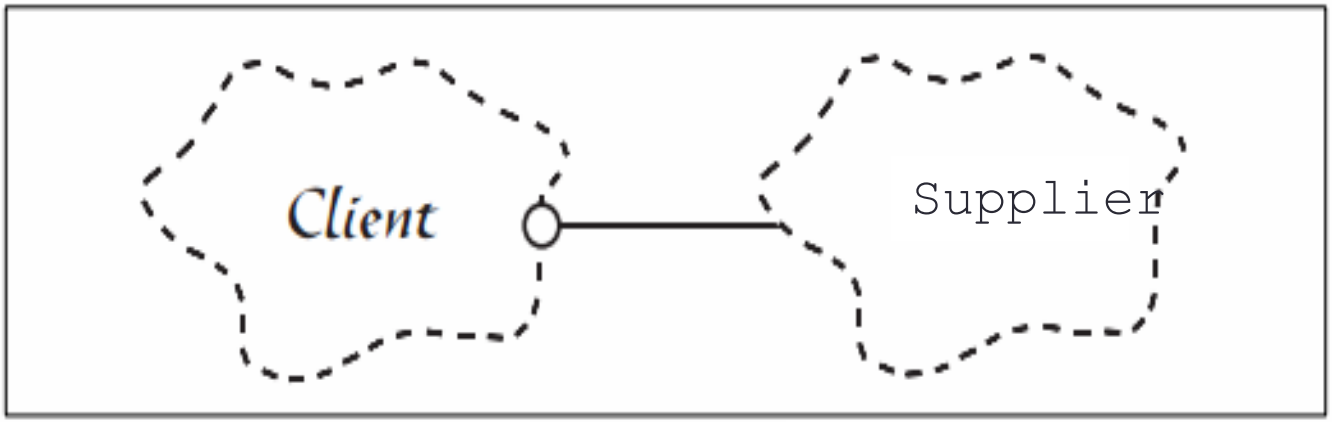
"You should be able to extend a classes behavior, without modifying code"

- "Open for Extension"
  - The behavior of the module/class can be extended
  - The module behave in new and different ways as the requirements changes, or to meet the needs of new aplications
- "Closed for Modification"
  - The source code of such a module is inviolate
  - No one is allowed to make source code changes to it

Principles of OO Class Design
OCP: The Open Closed Principle (cont)

- Client & Supplier classes are concrete
- If the Supplier implementation/class is changed, Client also needs change.

➔ How to resolve this problem?

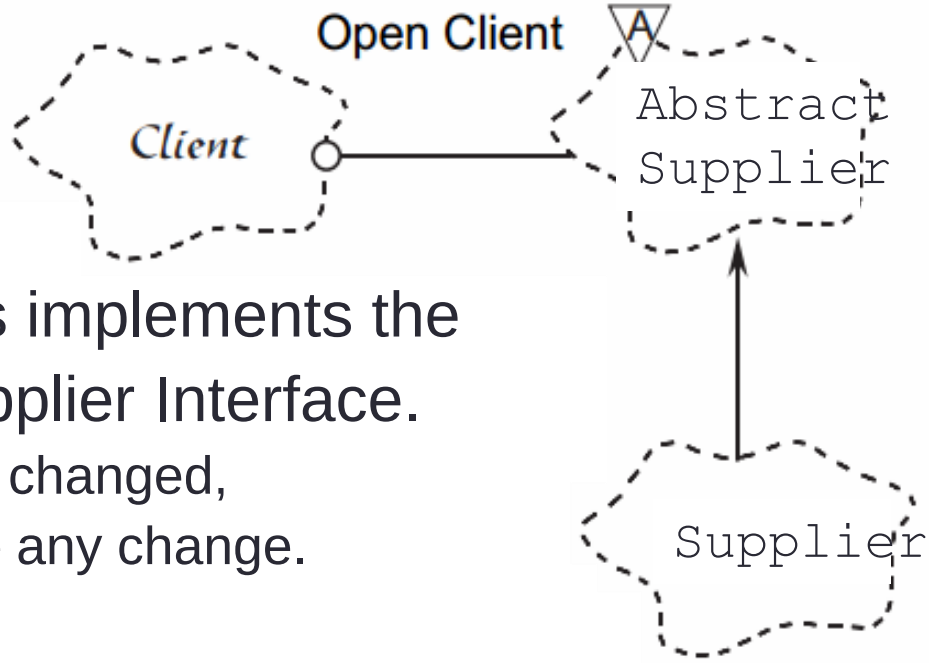


Closed Client

Principles of OO Class Design
OCP: The Open Closed Principle (cont)

- Change to support Open-Closed Principle.

➔ Abstraction is the key.



- The Concrete Supplier class implements the Abstract Supplier class / Supplier Interface.
  - The Supplier implementation is changed,
  - the Client is likely not to require any change.

➔ The Abstract Supplier class here is closed for modification and the Concrete class implementations here are Open for extension.

Ví dụ - HealthInsuranceSurveyor

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

ClaimApprovalManager

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

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 ...");
        }
    }
}
```

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;
    }
}
```

ClaimApprovalManager

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

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);
    }
}
```

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();
    }
}
```

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

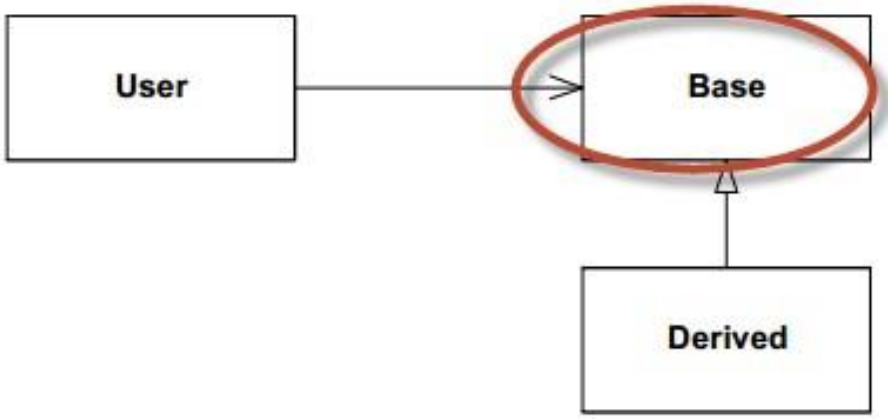
Principles of OO Class Design
LSP: The Liskov Substitution Principle

- *"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."*
  - Or

"Subclasses should be substitutable for their base classes."

User, Based, Derived, example.

```
void User(Base& b);

Derived d;
User(d);
```



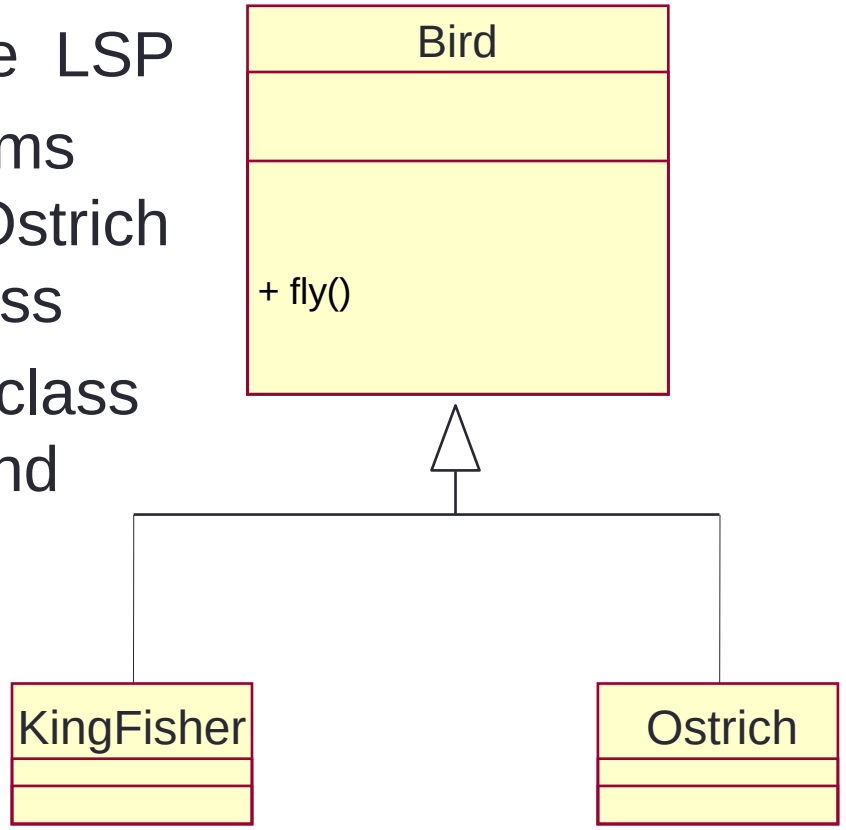
Principles of OO Class Design

LSP: The Liskov Substitution Principle (cont)

- Ostrich is a Bird (definitely!!!)
- Can it fly? No! => Violates the LSP

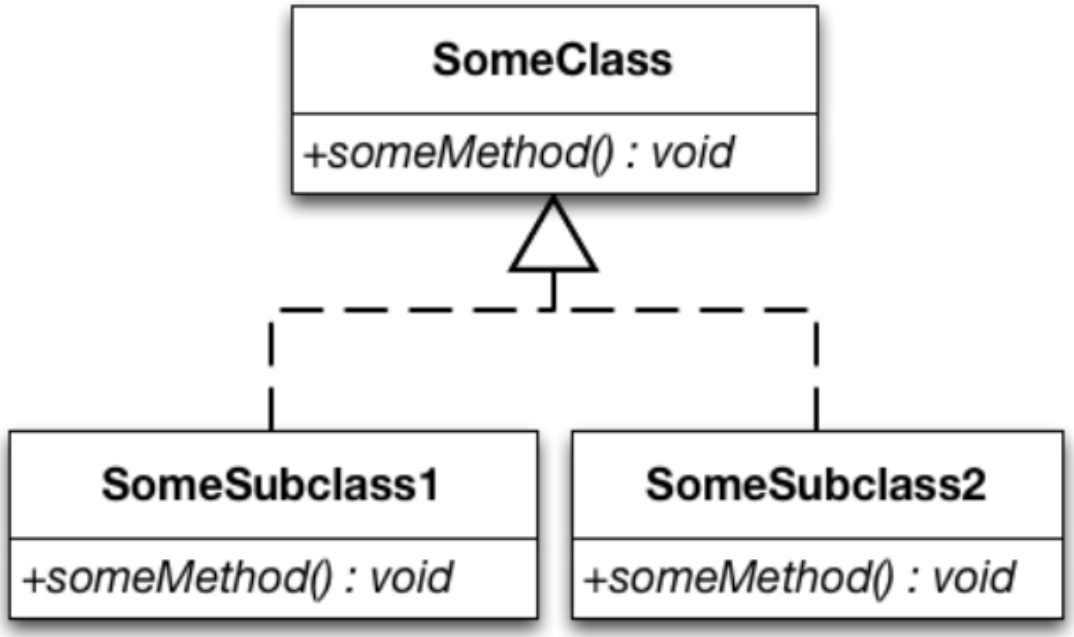
➔Even if in real world this seems natural, in the class design, Ostrich should not inherit the Bird class

➔There should be a separate class for birds that can't really fly and Ostrich inherits that.



Principles of OO Class Design
LSP: The Liskov Substitution Principle (cont)

```
void clientMethod(SomeClass sc) {
    …
    sc.someMethod();
    …
}
```



Principles of OO Class Design
LSP: The Liskov Substitution Principle (cont)

- “Inheritance” ~ "is a” relationship
  - But, easy to get carried away and end up in wrong design with bad inheritance.

  ➔ The LSP is a way of ensuring that inheritance is used correctly
- Why The LSP is so important? If not LSP,
  - Class hierarchy would be a **mess** and if subclass instance was passed as parameter to methods method, strange behavior might occur.
  - Unit tests for the Base classes would never succeed for the subclass.

  ➔ LSP is just an extension of Open-Close Principle!!!

Ví dụ – Rectangle

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

    public void setWidth(int width){
        m_width = width;
    }
    public void setHeight(int height){
        m_height = height;
    }
    public int getWidth(){
        return m_width;
    }
    public int getHeight(){
        return m_height;
    }
    public int getArea(){
        return m_width * m_height;
    }
}
```

Square

```
class Square extends Rectangle {
    @override
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }
    @override
    public void setHeight(int height){
        m_width = height;
        m_height = height;
    }
}
```

Test

```
public class RectangleFactory {
    public static Rectangle generate(){
        return new Square(); // if we return new Rectangle(), everything is fine
    }
}

class LspTest {
    public static void main (String args[]) {
        Rectangle r = RectangleFactory.generate();

        r.setWidth(5);
        r.setHeight(10);
        // user knows that r it's a rectangle.
        // It assumes that he's able to set the width and height as for the base class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
```

Giải pháp 1 (chưa tối ưu) – Shape

```
public abstract class Shape {
    protected int mHeight;
    protected int mWidth;

    public abstract int getWidth();

    public abstract void setWidth(int inWidth);

    public abstract int getHeight();

    public abstract void setHeight(int inHeight);

    public int getArea() {
        return mHeight * mWidth;
    }
}
```

Rectangle

```
public class Rectangle extends Shape {
	@Override
	public int getWidth() {
		return mWidth;
	}

	@Override
	public int getHeight() {
		return mHeight;
	}

	@Override
	public void setWidth(int inWidth) {
		mWidth = inWidth;
	}

	@Override
	public void setHeight(int inHeight) {
		mHeight = inHeight;
	}
}
```

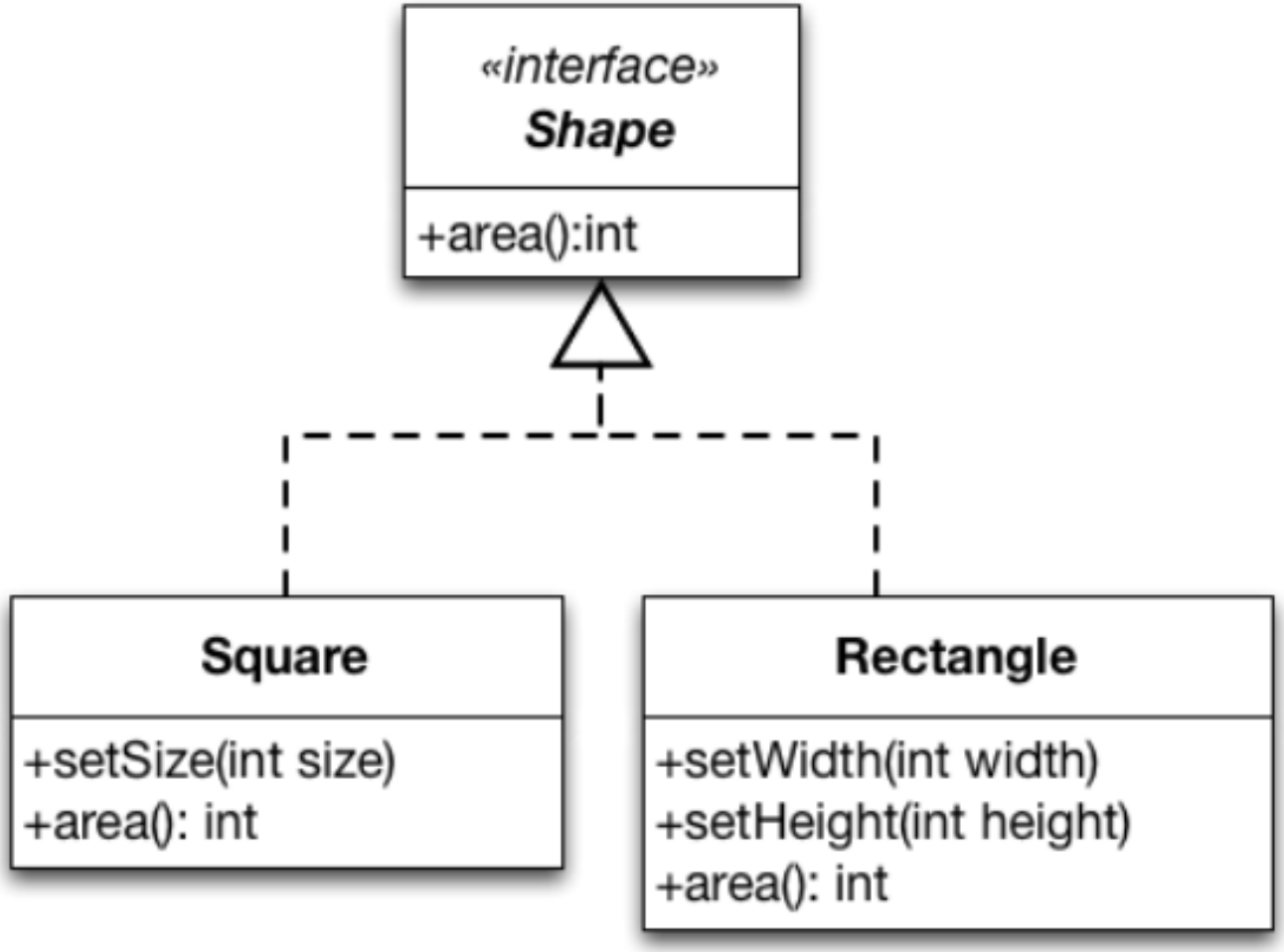
Square

```
public class Square extends Shape {
    @Override
    public int getWidth() {
        return mWidth;
    }
    @Override
    public void setWidth(int inWidth) {
        SetWidthAndHeight(inWidth);
    }
    @Override
    public int getHeight() {
        return mHeight;
    }
    @Override
    public void setHeight(int inHeight) {
        SetWidthAndHeight(inHeight);
    }
    private void setWidthAndHeight(int inValue) {
        mHeight = inValue;
        mWidth = inValue;
    }
}
```

Test

```
public class ShapeFactory {
    public static Shape generate(){
        return new Square();
    }
}

class LspTest {
    public static void main (String args[]) {
        Shape s = ShapeFactory.generate();

        s.setWidth(5);
        s.setHeight(10);

        System.out.println(r.getArea());
    }
}
```

Giải pháp 2: giải pháp tối ưu



Ví dụ 2 – Project

```
public class Project  {
    public ArrayList<ProjectFile> projectFiles;

    public void loadAllFiles()  {
        for (ProjectFile file: projectFiles) {
            file.loadFileData();
        }
    }

    public void saveAllFiles() {
        for (ProjectFile file: projectFiles) {
            file.saveFileData();
        }
    }
}
```

ProjectFile

```
public class ProjectFile {
    public string filePath;

    public byte[] fileData;

    public void loadFileData() {
        // Retrieve FileData from disk
    }

    public void saveFileData() {
        // Write FileData to disk
    }
}
```

ReadOnlyFile

```
public class ReadOnlyFile extends ProjectFile {
    @Override
    public void saveFileData() throws new InvalidOPException {
        throw new InvalidOPException();
    }
}
```

Project

```
public class Project  {
    public ArrayList<ProjectFile> projectFiles;

    public void loadAllFiles()  {
        for (ProjectFile file: projectFiles) {
            file.loadFileData();
        }
    }

    public void saveAllFiles() {
        for (ProjectFile file: projectFiles) {
            if (!file instanceOf ReadOnlyFile)
                file.saveFileData();
        }
    }
}
```

Project

```
public class Project  {
    public ArrayList<ProjectFile> allFiles;
    public ArrayList<WritableFile> writableFiles ;

    public void loadAllFiles()  {
        for (ProjectFile file: allFiles) {
            file.loadFileData();
        }
    }

    public void saveAllFiles() {
        for (ProjectFile file: writableFiles) {
            file.saveFileData();
        }
    }
}
```

ProjectFile

```
public class ProjectFile {
    public string filePath;

    public byte[] fileData;

    public void loadFileData() {
        // Retrieve FileData from disk
    }
}
```

WritableFile

```
public class WritableFile extends ProjectFile {
    public void saveFileData() {
        // Write FileData to disk
    }
}
```

Discussion

- Does method overriding break the Liskov substitution principle?

Ví dụ

```
public class Report{
  private Foo foo;
  public String toString(){
        return "";
  }
}
```

Ba lớp con

1. HTMLReport
2. XMLReport
3. TextReport

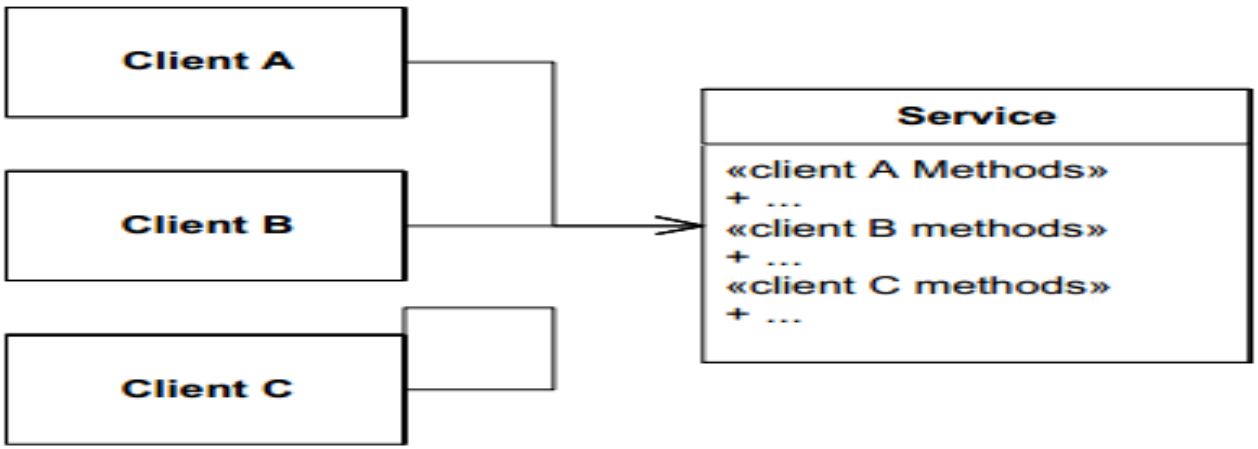
- Hợp đồng:
  - toString() trả về một xâu, là biểu diễn của Foo ở một format nào đó (và trả về xâu rỗng nếu format chưa xác định).
  - toString() không làm thay đổi đối tượng Report
  - toString() không tung ra một ngoại lệ (Exception) mới

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

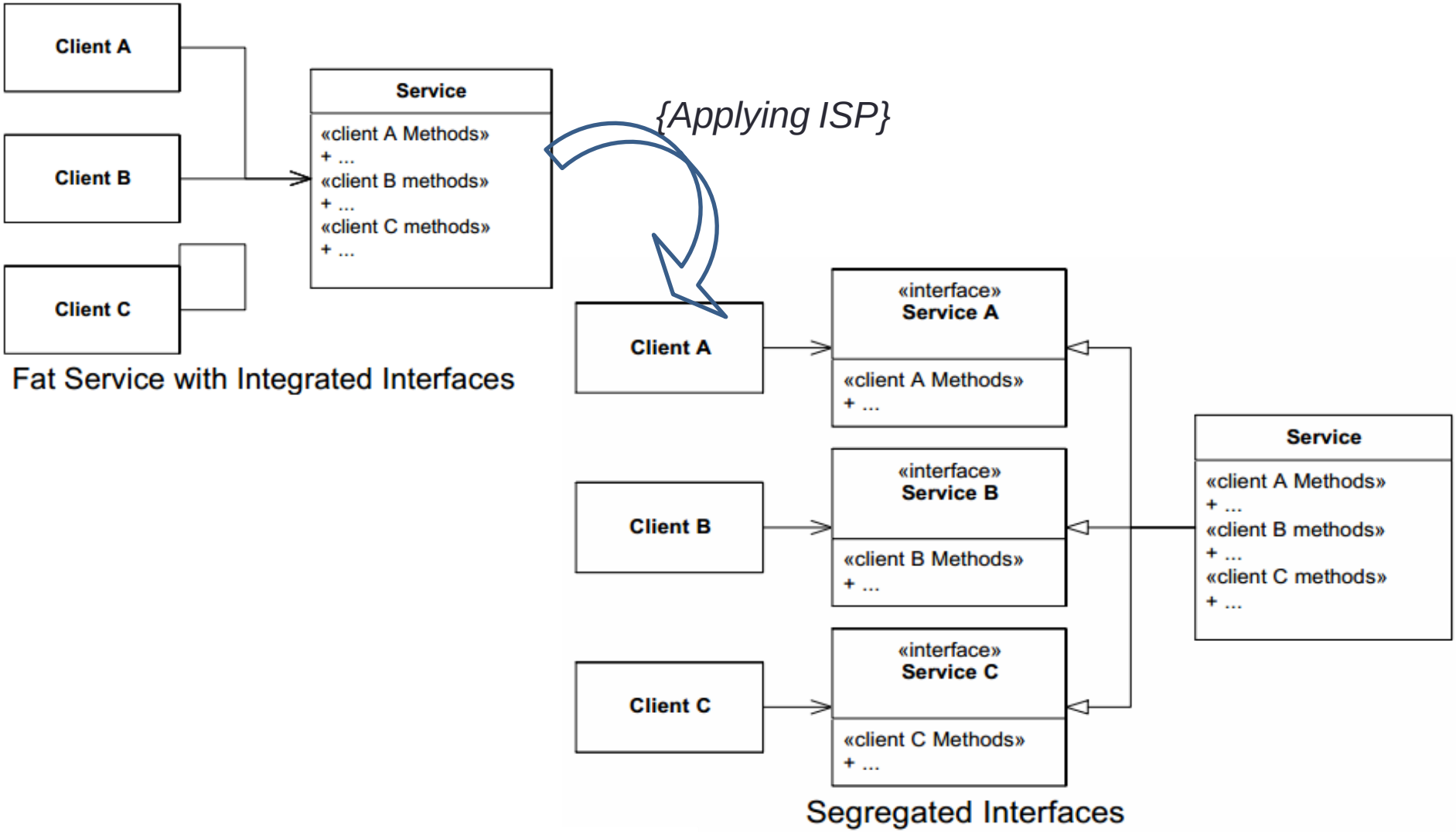
Principles of OO Class Design
ISP: The Interface Segregation Principle

- *"Client should not be forced to depend upon interface that they do not use."*
- *Or*
- *"Many client specific interfaces are better than one general purpose interface."*



Fat Service with Integrated Interfaces

Principles of OO Class Design
ISP: The Interface Segregation Principle



Principles of OO Class Design

ISP: The Interface Segregation Principle (cont.)

- Interfaces with too many methods are less re-usable.
- Such "fat interfaces“ with additional useless methods lead to inadvertent coupling between classes.
- Doing this also introduce unnecessary complexity and reduces maintainability or robustness in the system.

➔ The ISP ensures that, Interfaces are developed so that, each of them have their own responsibility and thus they are re-usable.

Ví dụ – Toy

```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}
```

ToyHouse

```
public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

Mã nguồn sửa đổi

```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
}


public interface Movable {
    void move();
}


public interface Flyable {
    void fly();
}
```

ToyHouse

```
public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public String toString(){
        return "ToyHouse: Toy house- Price: "+price+" Color: "+color;
    }
}
```

```
public class ToyCar implements Toy, Movable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
       this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyCar: Start moving car.");
    }
    @Override
    public String toString(){
        return "ToyCar: Moveable Toy car- Price: "+price+" Color: "+color;
    }
}
```

```
public class ToyPlane implements Toy, Movable, Flyable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyPlane: Start moving plane.");
    }
    @Override
    public void fly(){
        System.out.println("ToyPlane: Start flying plane.");
    }
    @Override
    public String toString(){
        return "ToyPlane: Moveable and flyable toy plane- Price: "+price+" Color: "+color;
    }
}
```

```
public class ToyBuilder {
    public static ToyHouse buildToyHouse(){
        ToyHouse toyHouse=new ToyHouse();
        toyHouse.setPrice(15.00);
        toyHouse.setColor("green");
        return toyHouse;
    }
    public static ToyCar buildToyCar(){
        ToyCar toyCar=new ToyCar();
        toyCar.setPrice(25.00);
        toyCar.setColor("red");
        toyCar.move();
        return toyCar;
    }
    public static ToyPlane buildToyPlane(){
        ToyPlane toyPlane=new ToyPlane();
        toyPlane.setPrice(125.00);
        toyPlane.setColor("white");
        toyPlane.move();
        toyPlane.fly();
        return toyPlane;
    }
}
```

Nguyên lý phân tách giao diện và nguyên lý một nhiệm vụ

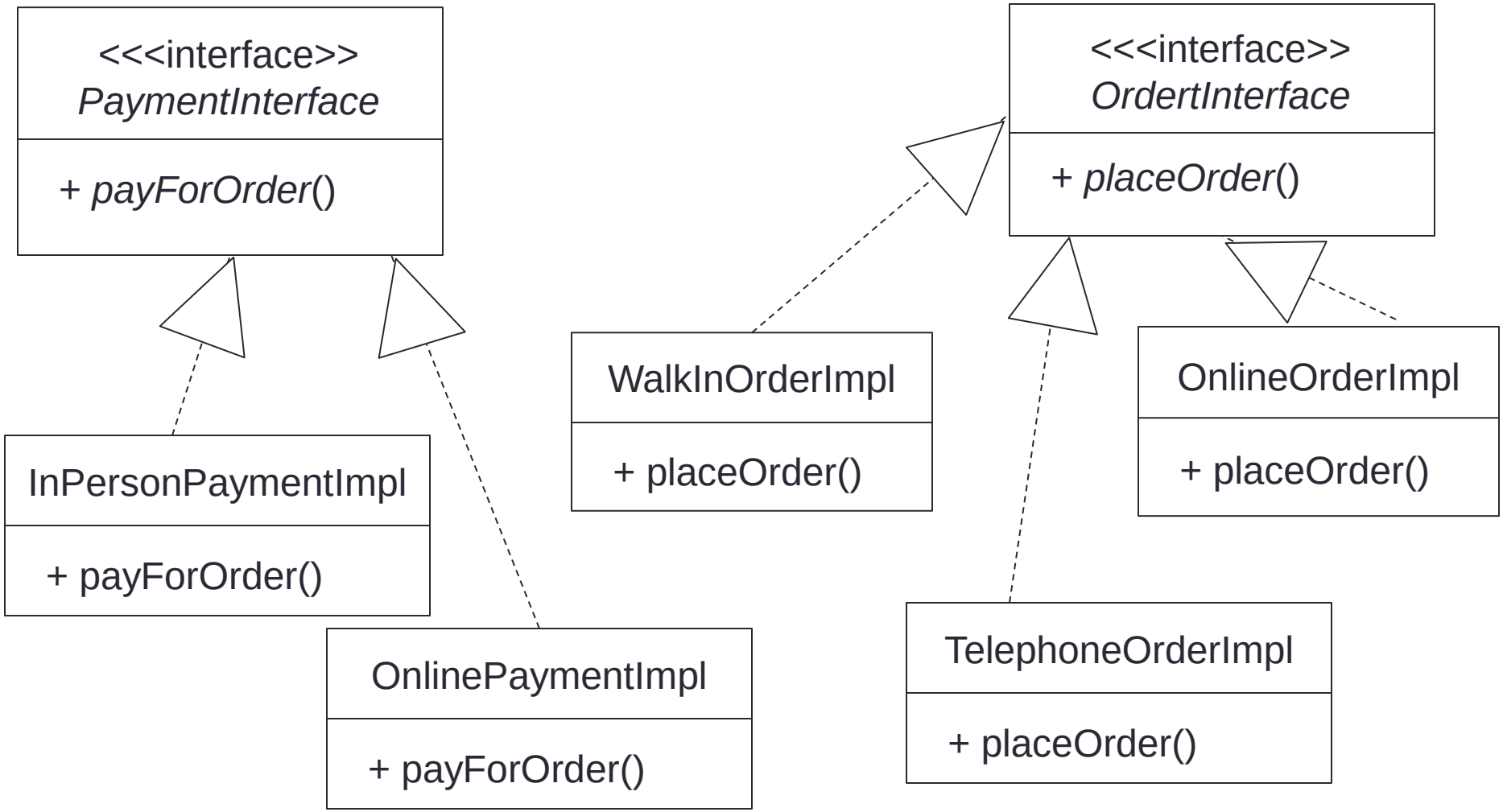
- Cùng mục đích: đảm bảo tính chất tập trung, nhỏ gọn, kết dính cao (highly cohesive) cho các các thành phần phần mềm
- Nguyên lý một nhiệm vụ áp dụng cho lớp
- Nguyên lý phân tách giao diện áp dụng cho giao diện

Ví dụ 2 – RestaurantInterface

```
public interface RestaurantInterface {
    public void acceptOnlineOrder();
    public void takeTelephoneOrder();
    public void payOnline();
    public void walkInCustomerOrder();
    public void payInPerson();
}
```

```
public class OnlineClientImpl implements RestaurantInterface {
    @Override
    public void acceptOnlineOrder() {
        // logic for placing online order
    }

    @Override
    public void takeTelephoneOrder() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }

    @Override
    public void payOnline() {
        // logic for paying online
    }

    @Override
    public void walkInCustomerOrder() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }

    @Override
    public void payInPerson() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
}
```

Refactored code



Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

Principles of OO Class Design
DIP: The Dependency Inversion Principle

"High level modules should not depend upon low level modules. Both should depend upon abstractions"

Or

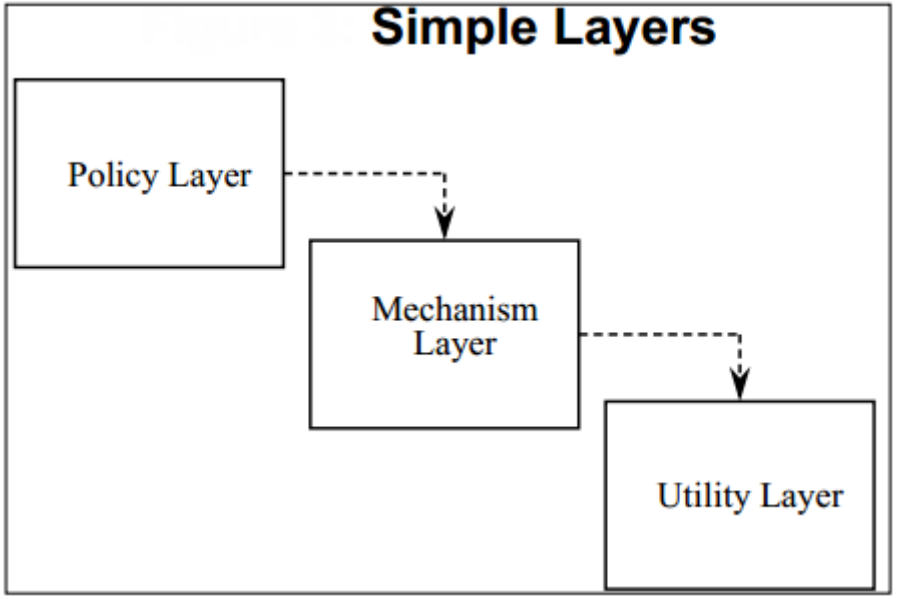
"Abstractions should not depend upon details. Details should depend upon abstraction."

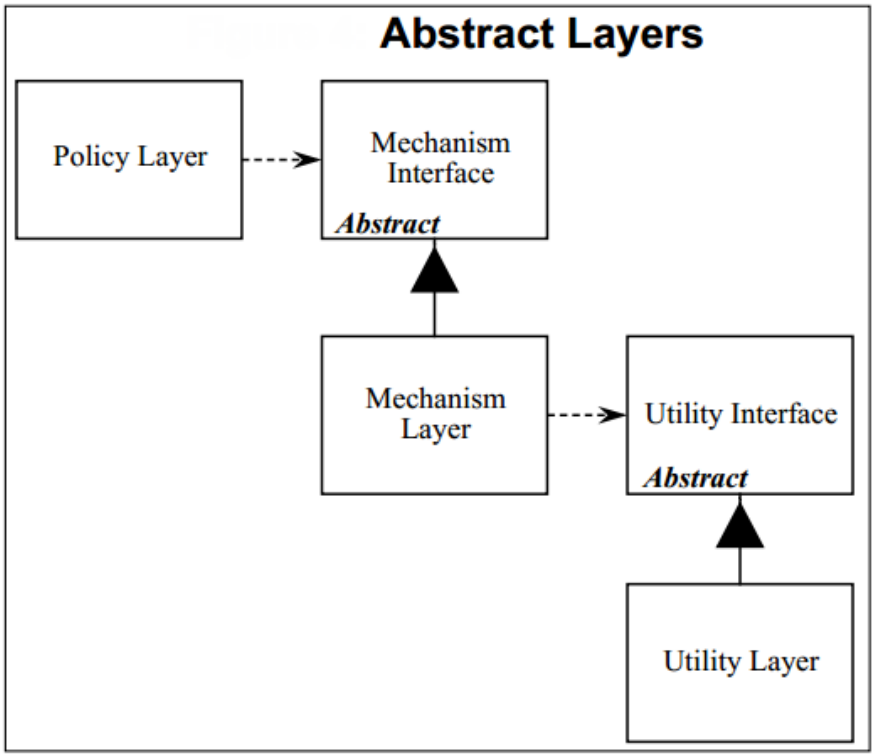
Or

"Depend upon Abstractions. Do not depend upon concretions."

Principles of OO Class Design
DIP: The Dependency Inversion Principle

- Strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.
  - A well designed object-oriented application.
  - E.g. Layers of application





Ví dụ – LightBulb

```
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```

Interface ISwitchable

```
public interface ISwitchable {
    public void turnOn();
    public void turnOff();
}
```

ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            client.turnOff();
            this.on = false;
        } else {
              client.turnOn();
              this.on = true;
        }
    }
}
```

LightBulb

```
public class LightBulb implements ISwitchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

Fan

```
public class Fan implements ISwitchable {
    @Override
    public void turnOn() {
        System.out.println("Fan: Fan turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("Fan: Fan turned off...");
    }
}
```

ElectricPowerSwitchTest

```
public class ElectricPowerSwitchTest {

    @Test
    public void testPress() throws Exception {
        ISwitchable switchableBulb=new LightBulb();
        ElectricPowerSwitch bulbPowerSwitch =
            new ElectricPowerSwitch(switchableBulb);
        bulbPowerSwitch.press();
        bulbPowerSwitch.press();

        ISwitchable switchableFan=new Fan();
        ElectricPowerSwitch fanPowerSwitch =
            new ElectricPowerSwitch(switchableFan);
        fanPowerSwitch.press();
        fanPowerSwitch.press();
    }
}
```

ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            client.turnOff();
            this.on = false;
        } else {
              client.turnOn();
              this.on = true;
        }
   }
}
```

Any problem?

ISwitch

```
public interface ISwitch {
    boolean isOn();
    void press();
}
```

ElectricPowerSwitch

```
public class ElectricPowerSwitch implements ISwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            client.turnOff();
            this.on = false;
        } else {
              client.turnOn();
              this.on = true;
        }
    }
}
```

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

Design exercise

- Write a typing break reminder program
  - Offer the hard-working user occasional reminders of the health issues, and encourage the user to take a break from typing
- Naive design
  - Make a method to display messages and offer exercises
  - Make a loop to call that method from time to time

(Let's ignore multi-threaded solutions for this discussion)

TimeToStretch suggests exercises

```
public class TimeToStretch {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }
  public void suggestExercise() {
  ...
  }
}
```

Timer calls run() periodically

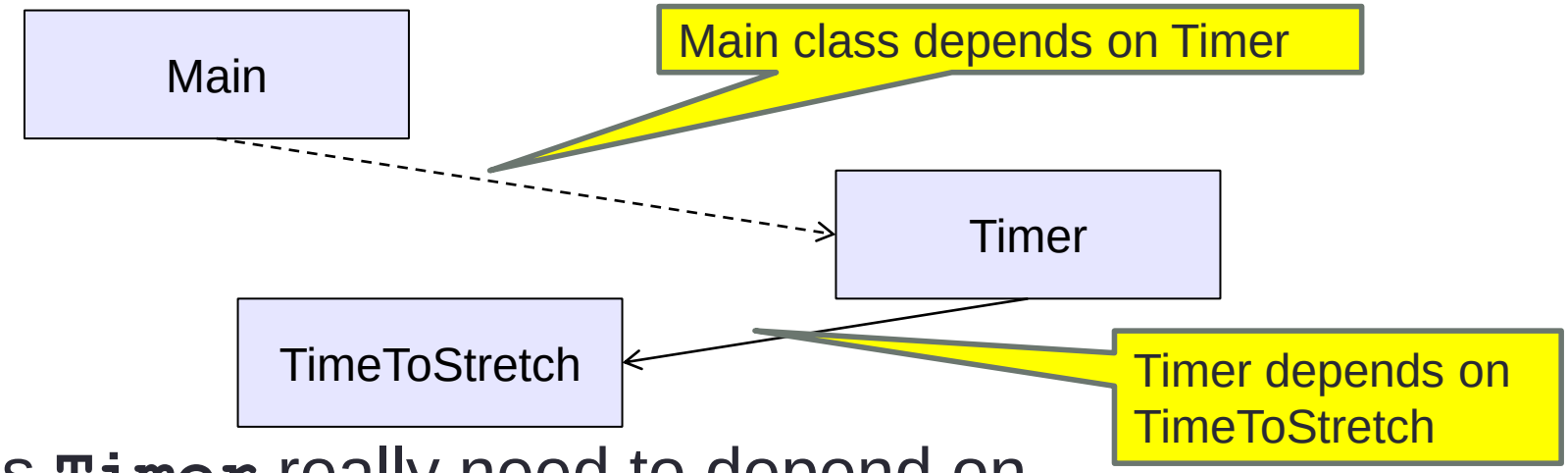
```
public class Timer {
  private TimeToStretch tts = new TimeToStretch();
  public void start() {
    while (true) {
      ...
      if (enoughTimeHasPassed) {
        tts.run();
      }
      ...
    }
  }
}
```

Main class puts it together

```
class Main {
  public static void main(String[] args) {
    Timer t = new Timer();
    t.start();
  }
}
```

Module dependency diagram

- An arrow in a module dependency diagram indicates "depends on" or "knows about" – simplistically, "any name mentioned in the source code"



- Does **`Timer`** really need to depend on **`TimeToStretch`**?
- Is Timer re-usable in a new context?

Decoupling

- **`Timer`** needs to call the **`run`** method
  - **`Timer`** doesn't need to know what the **`run`** method does
- Weaken the dependency of **`Timer`** on **`TimeToStretch`**
- Introduce a weaker specification, in the form of an interface or abstract class

```
public abstract class TimerTask {
    public abstract void run();
}
```

- **`Timer`** only needs to know that something (e.g., **`TimeToStretch`**) meets the **`TimerTask`** specification

TimeToStretch (version 2)

```
public class TimeToStretch extends TimerTask {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }

  public void suggestExercise() {
    ...
  }
}
```

Timer v2

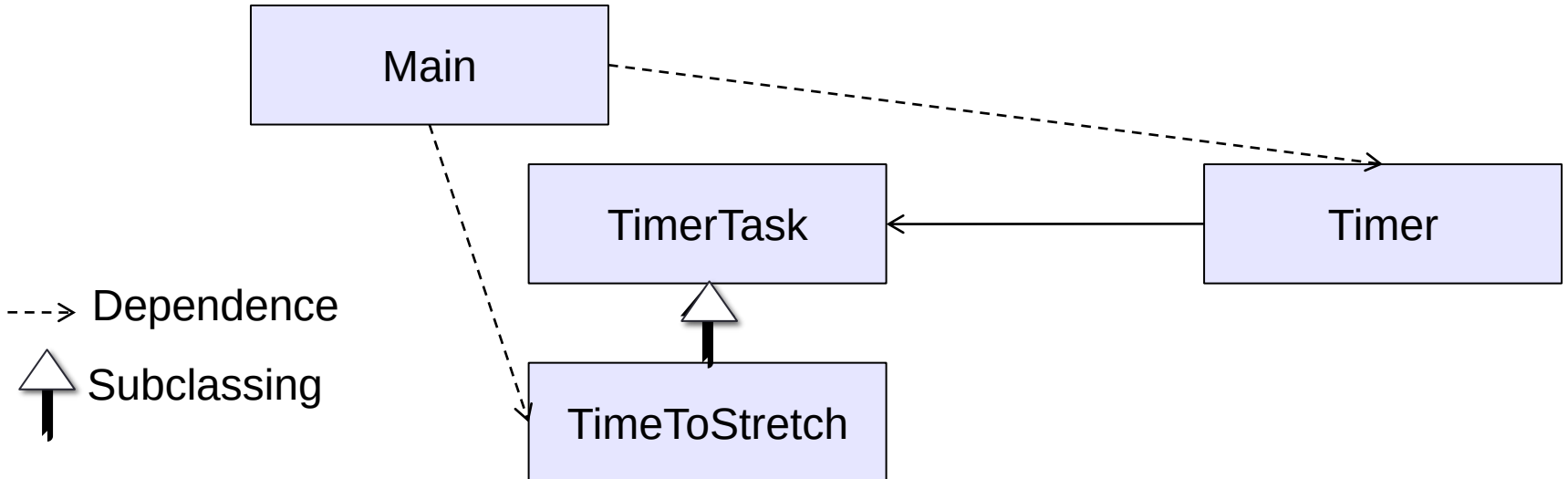
```
public class Timer {
  private TimerTask task;
  public Timer(TimerTask task) { this.task = task; }
  public void setTask(TimerTask task){this.task = task;}
  public void start() {
    while (true) {
      ...
       if (enoughTime)
              task.run();
    }
  }
}
```

- Main creates the **`TimeToStretch`** object and passes it to **`Timer`**

```
Timer t = new Timer(new TimeToStretch());
t.start();
t.setTask(new TimeToSave());
t.start();
```

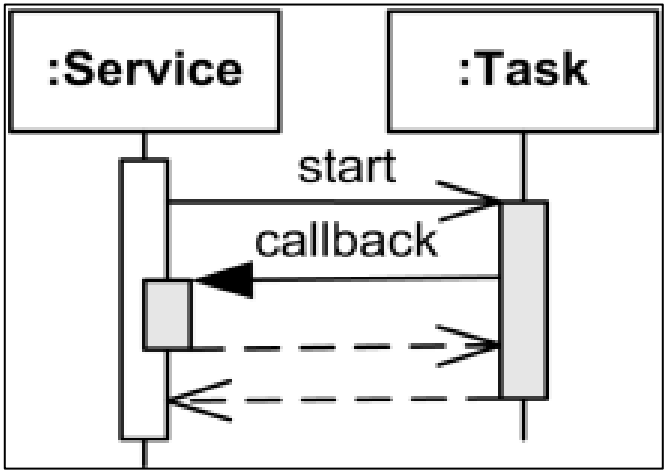
Module dependency diagram

- **`Main`** still depends on **`Timer`** (is this necessary?)
- **`Main`** depends on the constructor for **`TimeToStretch`**
- **`Timer`** depends on **`TimerTask`**, not **`TimeToStretch`**
  - Unaffected by implementation details of **`TimeToStretch`**
  - Now **`Timer`** is much easier to reuse



callbacks

- **`TimeToStretch`** creates a **`Timer`**, and passes in a reference to itself so the **`Timer`** can call it back
- This is a *callback* – a method call from a module to a client that notifies about some condition
- Use a callback to invert a dependency
  - Inverted dependency: **`TimeToStretch`** depends on **`Timer`** (not vice versa)
  - Side benefit: **`Main`** does not depend on **`Timer`**



A synchronous callback.
Time increases downward.
Solid lines: calls
Dotted lines: returns

TimeToStretch v3

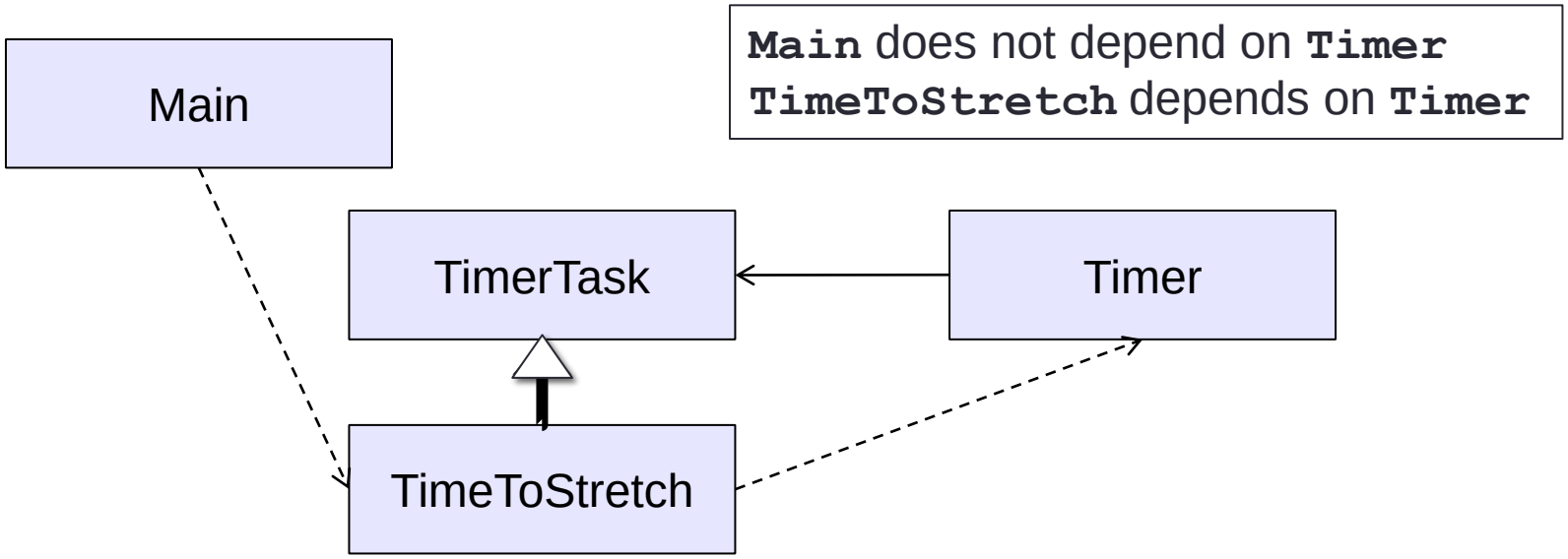
```
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    ...
}
```

Register interest with the timer

Callback entry point

Main v3

- `TimeToStretch tts = new TimeToStretch();`
  `tts.start();`
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between `Timer` and `TimeToStretch` (compared to v1)



How do we design classes?

- One common approach to class identification is to consider the specifications
- In particular, it is often the case that
  - *nouns* are potential classes, objects, fields
  - *verbs* are potential methods or responsibilities of a class

Design exercise

- Suppose we are writing a birthday-reminder application that tracks a set of people and their birthdays, providing reminders of whose birthdays are on a given day

- What classes are we likely to want to have? Why?

Class shout-out about classes

More detail for those classes

- What fields do they have?
- What constructors do they have?
- What methods do they provide?
- What invariants should we guarantee?

In small groups, ~5 minutes