Trường Đại học Công nghiệp Thực phẩm Tp. Hồ Chí Minh

Bộ môn Công nghệ Phần mềm

Lập trình hướng đối tượng

ĐẠI HỌC CHÍNH QUY

Trường Đại học Công nghiệp Thực phẩm Tp. Hồ Chí Minh

Bộ môn Công nghệ Phần mềm

CHƯƠNG 6

MỘT SỐ NGUYÊN LÝ VÀ PATTERN CƠ BẢN

ĐẠI HỌC CHÍNH QUY

03/05/2023

MỤC TIÊU

1. Trình bày được các nguyên lý trong lập trình hướng đối tượng.
2. Vận dụng được các pattern cơ bản vào trong phân tích thiết kế hướng đối tượng.

NỘI DUNG

4.1. Các nguyên lý lập trình

4.2. Các pattern thiết kế hướng đối tượng

4.1. CÁC NGUYÊN LÝ LẬP TRÌNH

4.1.1. Nguyên lý đơn nhiệm

4.1.2. Nguyên lý đóng – mở

4.1.3. Nguyên lý thay thế Liskov

4.1.4. Nguyên lý chia tách giao diện

4.1.5. Nguyên lý phụ thuộc đảo

GIỚI THIỆU

- Một sản phẩm làm ra luôn luôn có sự thay đổi và mở rộng chức năng theo thời gian.
- Trong quy trình phát triển phần mềm, khâu phân tích và thiết kế là cực kỳ quan trọng → một phần mềm có thể dễ dàng đáp ứng với thay đổi nhất.
- Để thiết kế một phần mềm có độ linh hoạt cao, cần áp dụng các kiến thức về **Design pattern các nguyên tắc trong thiết kế và lập trình.**

SOLID

5 nguyên tắc của thiết kế hướng đối tượng

Single responsibility principle

Open/closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

1. SINGLE RESPONSIBILITY PRINCIPLE

- **Nguyên lý đơn nhiệm**
- Nội dung: **Một class chỉ nên giữ 1 trách nhiệm duy nhất.** Nghĩa là: *chỉ có thể sửa đổi class với một lý do duy nhất*.

1. SINGLE RESPONSIBILITY PRINCIPLE

Ví dụ 1:

```
public class Person
{
    private string name;
    private string surname;
    private string email;
    public Person()...
    public Person(string name, string surname, string email)
    {
        this.name = name;
        this.surname = surname;
        if (isValid(this.email) == true)
            this.email = email;
        else {
            this.email = string.Empty;
            throw new Exception("email is not valid");
        }
    }
    public bool isValid(string email)
    {
        //kiem tra email;
        return true;
    }
}
```

Không liên quan với hành vi của một người

1. SINGLE RESPONSIBILITY PRINCIPLE

➤ **Loại bỏ** trách nhiệm xác thực email từ lớp Person và **tạo lớp Email** có trách nhiệm đó.

```
public class Email
{
    private string email;
    public Email(string email)
    {
        if (isValid(email) == true)
            this.email = email;
        else
            throw new Exception("Email khong hop le");
    }

    private bool isValid(string email)
    {
        //kiem tra email
        return true;
    }
}
```

1. SINGLE RESPONSIBILITY PRINCIPLE

- Lớp Person trở thành

```
public class Person
{
    private string name;
    private string surname;
    private Email email;
    public string Name...
    public string Surname...
    public Email Email...
    public Person()...
    public Person(string name, string surname, Email email)
    {
        this.name = name;
        this.surname = surname;
        this.email = email;
    }
}
```

2. OPEN/CLOSED PRINCIPLE

- **Nguyên lý đóng/mở**
- **Nội dung**: **Có thể thoải mái mở rộng 1 class, nhưng không được sửa đổi bên trong class đó.**
- **Open for extension**

  Khả năng sử dụng lại class, function. Đồng thời có thể thêm các tính năng mới.
- **Close for modification**

  Không (hạn chế) sửa đổi (update, delete) class, function đã tạo.

2. OPEN/CLOSED PRINCIPLE

Ví dụ 2: Ta có class Shape, và 2 class *Square*, *Circle kế thừa từ Shape*. Class *AreaDisplay* tính diện tích các hình này và in ra.

```
public class Shape
{
    //cai dat lop shape
}
```

```
public class Square:Shape
{
    private double edge;
    public double Edge
    {
        get { return edge; }
        set { edge = value; }
    }
}
```

```
public class Circle:Square
{
    private double radious;
    public double R
    {
        get { return radious; }
        set { radious = value; }
    }
}
```

2. OPEN/CLOSED PRINCIPLE

Ví dụ 2: Ta có class Shape, và 2 class *Square*, *Circle kế thừa từ Shape*. Class *AreaDisplay* tính diện tích các hình này và in ra.

```
public class AreaDisplay
{
    public static void showArea(List<Shape> shapes)
    {
        foreach (Shape shape in shapes)
        {
            if (shape is Square)
            {
                Square square = (Square)shape;
                double area = Math.Pow(square.Edge, 2);
                Console.WriteLine(area);
            }
            if (shape is Circle)
            {
                Circle circle = (Circle)shape;
                double area = Math.Pow(circle.R, 2) * Math.PI;
                Console.WriteLine(area);
            }
        }
    }
}
```

2. OPEN/CLOSED PRINCIPLE

Ví dụ 2: Ta có class Shape, và 2 class *Square*, *Circle kế thừa từ Shape*. Class *AreaDisplay* tính diện tích các hình này và in ra.

```
public class AreaDisplay
{
    public static void showArea(List<Shape> shapes)
    {
        foreach (Shape shape in shapes)
        {
            if (shape is Square)
            {
                Square square = (Square)shape;
                double area = Math.Pow(square.Edge, 2);
                Console.WriteLine(area);
            }
            if (shape is Circle)
            {
                Circle circle = (Circle)shape;
                double area = Math.Pow(circle.R, 2) * Math.PI;
                Console.WriteLine(area);
            }
        }
    }
}
```

Thêm nhiều class (hình), giải quyết như thế nào?

2. OPEN/CLOSED PRINCIPLE

Áp dụng OCP, ta sẽ cải tiến lại như sau, bằng cách sử dụng **Lớp trừu tượng** và chuyển phương thức tính diện tích vào mỗi class.

```
public abstract class Shape
{
    public abstract double calcualteArea();
}
```

```
public class Square:Shape
{
    private double edge;
    public double Edge
    {
        get { return edge; }
        set { edge = value; }
    }
    public override double calcualteArea()
    {
        return Math.Pow(edge, 2);
    }
}
```

2. OPEN/CLOSED PRINCIPLE

Khi đó class AreaDisplay trở thành:

```
public class AreaDisplay
{
    public static void showArea(List<Shape> shapes)
    {
        foreach (Shape shape in shapes)
        {
            Console.WriteLine(shape.calcualteArea());
        }
    }
}
```

In ra diện tích của các hình thêm vào mà không cần sửa source code của lớp.

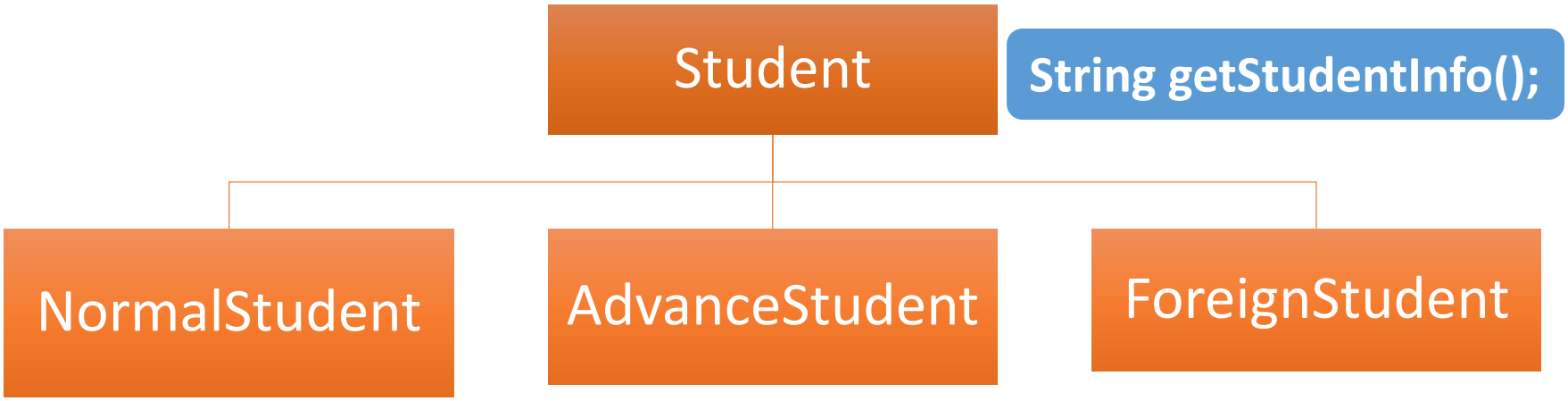
3. LISKOV SUBSTITUTION PRINCIPLE

- **Nguyên lý thay thế Liskov**
- **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.**

Nghĩa là: các lớp con có thể thực thi được và đúng những functions mà lớp cha đã cung cấp trước đó.

3. LISKOV SUBSTITUTION PRINCIPLE

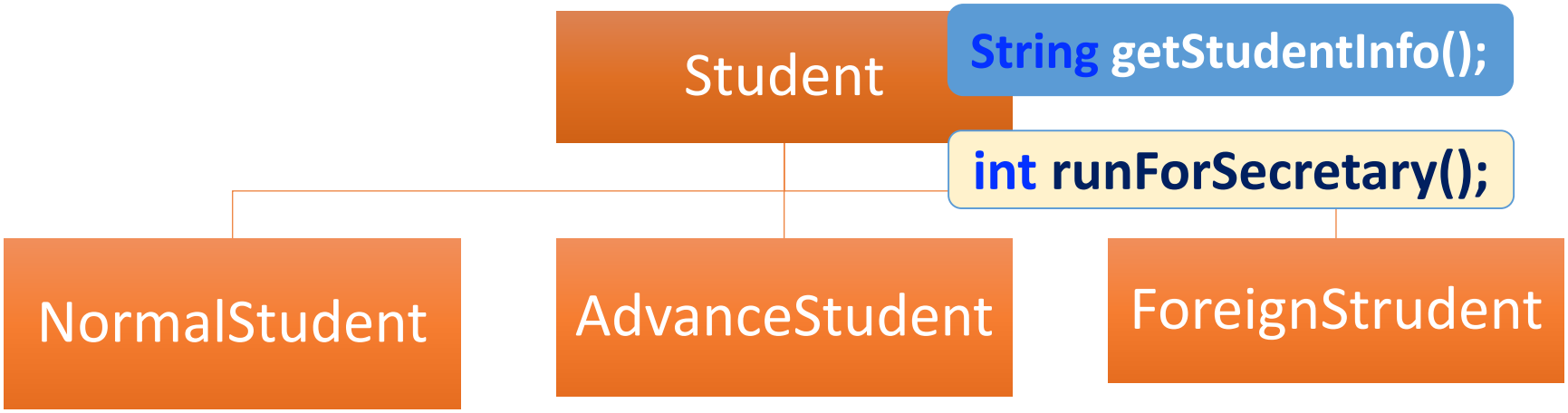
Ví dụ 3: Mô hình tổ chức các lớp sinh viên



Mở rộng một tính năng như sau: **ứng cử vào chức bí thư Đoàn khoa**

3. LISKOV SUBSTITUTION PRINCIPLE

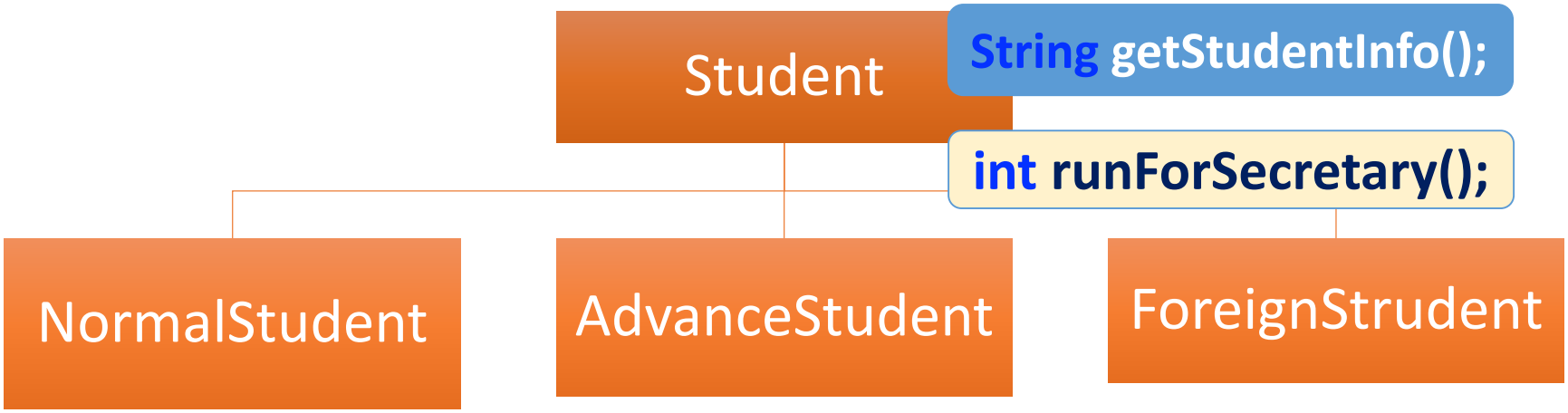
Ví dụ 3: Mô hình tổ chức các lớp sinh viên



Nhưng giả sử rằng chỉ có sinh viên người trong nước được ứng cử vào chức bí thư Đoàn.

3. LISKOV SUBSTITUTION PRINCIPLE

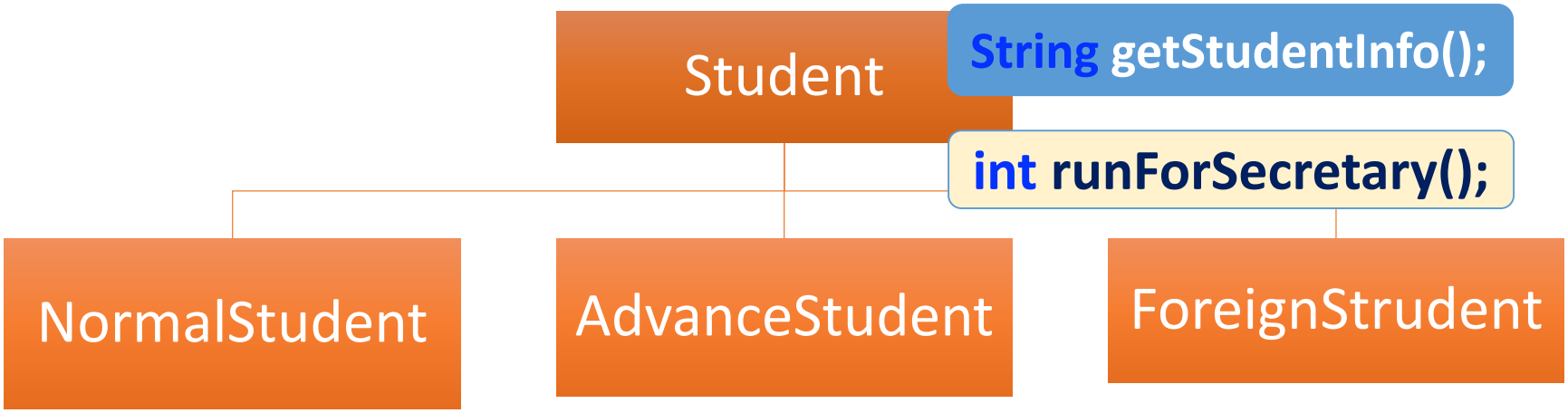
Ví dụ 3: Mô hình tổ chức các lớp sinh viên



Lớp ForeignStudent là sẽ không có chức năng này, viết code trong function **runForSecretary()** như thế nào? → đưa ra một ngoại lệ.

3. LISKOV SUBSTITUTION PRINCIPLE

Ví dụ 3: Mô hình tổ chức các lớp sinh viên



Không sử dụng "tính đa hình" trong lập trình hướng đối tượng một cách đúng đắn → **Vi phạm LSP**.

3. LISKOV SUBSTITUTION PRINCIPLE

Ví dụ 3: Mô hình tổ chức các lớp sinh viên không vi phạm LSP

```
interface iNationalSocialActivity
{
    int runForSecretary();
}
```

```
public class NormalStudent:Student,iNationalSocialActivity
{
    //code
}
```

```
class AdvanceStudent : Student, iNationalSocialActivity
{
    //code
}
```

```
public class ForeignStudent:Student
{
    //code
}
```

4. INTERFACE SEGREGATION PRINCIPLE

- Nguyên lý chia tách giao diện
- Nội dung: Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể.

4.INTERFACE SEGREGATION PRINCIPLE

- Các class kế thừa (implements) interface sẽ bắt buộc phải phải **thực thi toàn bộ** các method của interface, bao gồm cả những method mà class không cần dùng.
- Tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc thực thi và quản lý sẽ dễ hơn.

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Thiết kế interface (I_Animal) cho động vật nào cũng có thể ăn, uống, ngủ.

```
public interface I_Animal {
    void Eat();
    void Drink();
    void Sleep();
}
```

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Lớp Dog thực thi I_Animal.

```
public class Dog : I_Animal
{
  public void Eat () {}
  public void Drink () {}
  public void Sleep () {}
}
```

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Lớp Cat thực thi I_Animal.

```
public class Cat : I_Animal
{
  public void Eat () {}
  public void Drink () {}
  public void Sleep () {}
}
```

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Người dùng muốn thêm một số động vật mới có những tính năng như bơi, bay, ... → bổ sung thêm các method Swim(), Fly() vào I_Animal.

```
public interface I_Animal {
        void Eat();
        …
        void Swim();
        void Fly();
}
```

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Lớp Dog trở thành:

```
public class Dog : I_Animal
{
  public void Eat () {}
  public void Drink () {}
  public void Sleep () {}
  public void Swim () {}
  public void Fly() throws Exception {
      throw new Exception("Chó không biết bay");
  }
}
```

4.INTERFACE SEGREGATION PRINCIPLE

Ví dụ 4: Giải pháp → Tách interface *I_Animal* ra thành các interface nhỏ như sau:

```
public interface I_Animal {
  void Eat();
  void Drink();
  void Sleep();
}
public interface I_Bird {
  void Fly();
}
public interface I_Fish {
  void Swim();
}
```

4.INTERFACE SEGREGATION PRINCIPLE

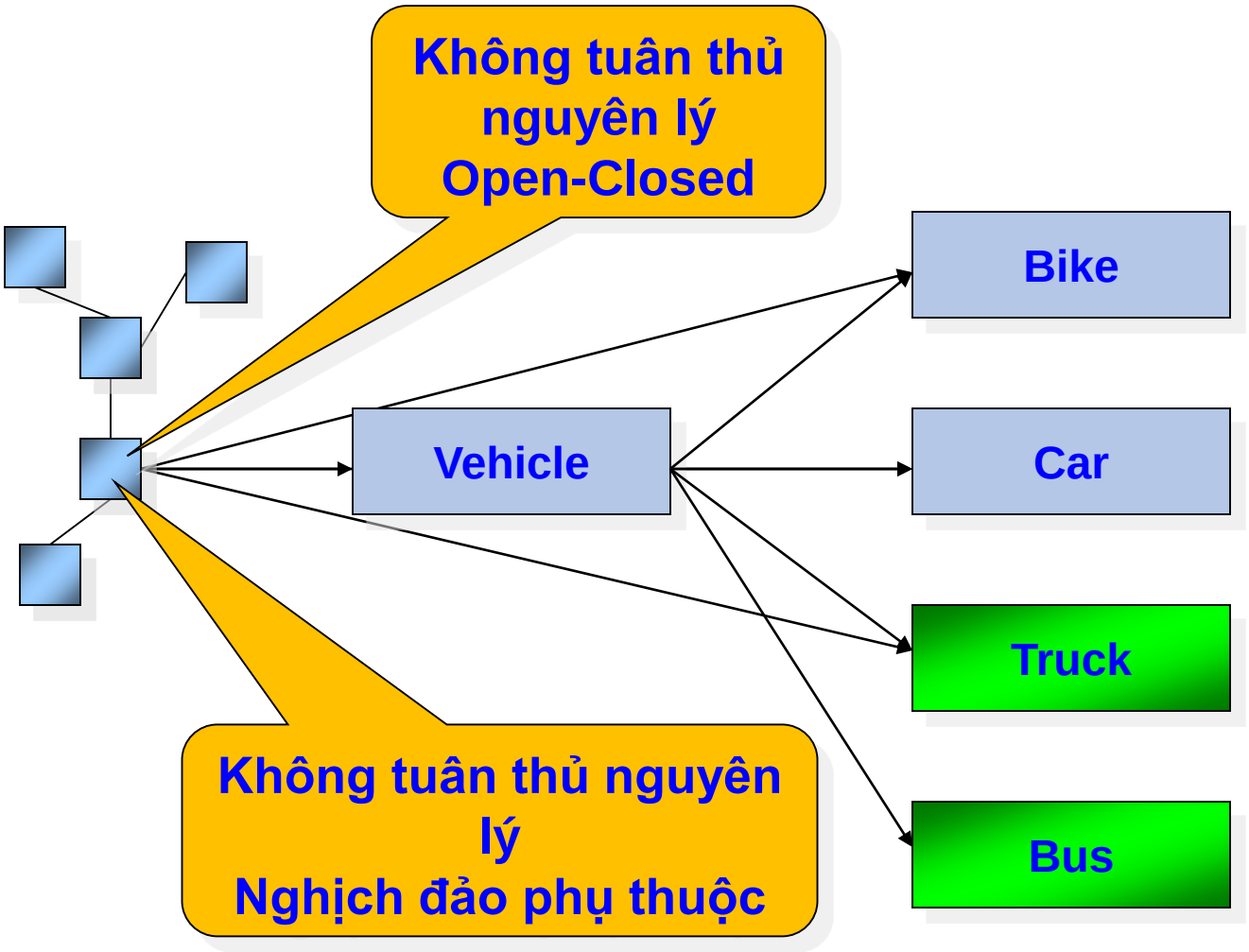
Ví dụ 4: Khi đó, các class chỉ cần kế thừa những interface có chúc năng

```
public class Dog : IAnimal {

}

public class FlappyBird : IAnimal, IBird {

}

...
```

5. DEPENDENCY INVERSION PRINCIPLE

- **Nguyên lý phụ thuộc đảo (DIP)**
- **Nội dung: Các thành phần trong phần mềm không nên phụ thuộc vào những cái riêng, cụ thể (details) mà ngược lại nên phụ thuộc vào những cái chung, tổng quát (abstractions) của những cái riêng, cụ thể đó.**

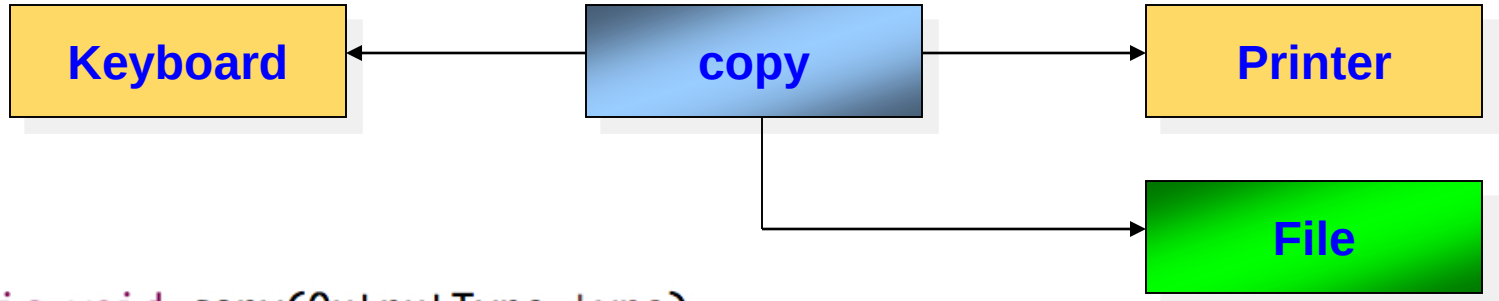
5. DEPENDENCY INVERSION PRINCIPLE



5. DEPENDENCY INVERSION PRINCIPLE

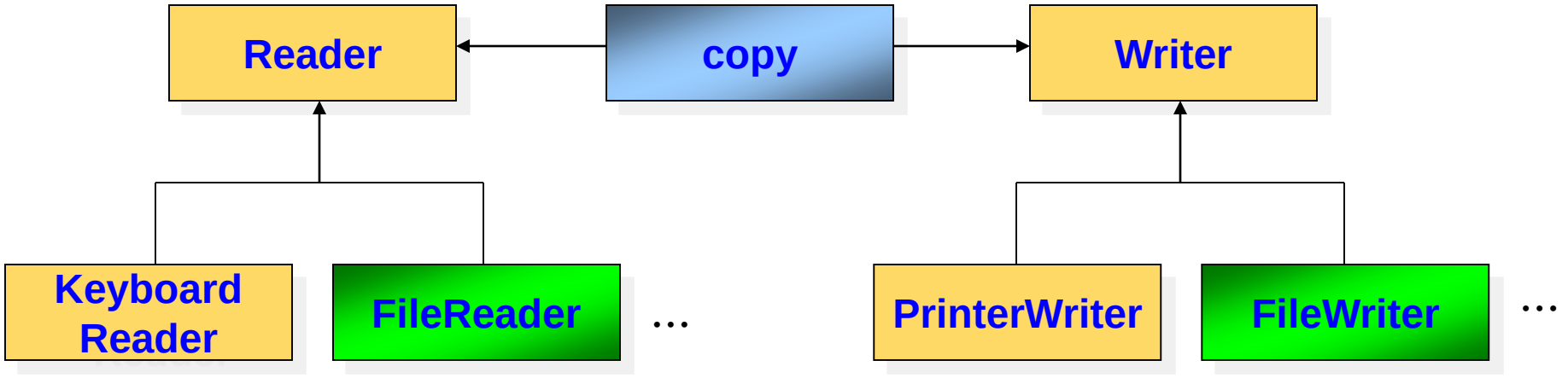

```
3  public class Test {
4
5      public void copy()
6      {
7          KeyBoard    keyboard = new KeyBoard();
8          Printer printer = new Printer();
9          char        c;
10
11         while ((c = keyboard.read()) != 'q')
12             printer.write(c);
13     }
14 }
```

5. DEPENDENCY INVERSION PRINCIPLE




```
public void copy(OutputType type)
{
    KeyBoard    keyboard = new KeyBoard();
    Printer printer = new Printer();
    File file = new File();
    char c;

    while (( c = keyboard.read()) != 'q')
        if (type instanceof Printer)
            printer.write(c);
        else if (type instanceof File)
            file.write(c);
}
```

5. DEPENDENCY INVERSION PRINCIPLE



NỘI DUNG

4.1. Các nguyên lý lập trình

4.2. Các pattern thiết kế hướng đối tượng

DESIGN PATTERN LÀ GÌ?

- Là một giải pháp tổng thể cho các vấn đề chung.
- Là template mô tả cách giải quyết một vấn đề.
- Chỉ ra mối quan hệ và sự tương tác giữa các lớp hay các đối tượng.
- Giúp thiết kế linh hoạt, dễ dàng thay đổi và bảo trì hơn.

VAI TRÒ DESIGN PATTERN

- Giúp cho việc bảo trì, phát triển phần mềm dễ dàng hơn.
- Cải thiện được kỹ năng lập trình hướng đối tượng, hạn chế các lỗi tiềm ẩn.
- Hiểu rõ hơn cơ chế hoạt động của các ngôn ngữ lập trình.

Cần phải nắm vững về OOP:

abstract class, interface, static,…

PHÂN LOẠI PATTERN

- Chia làm 3 loại chính
  - **Creational Pattern (nhóm khởi tạo)**
  - **Structural Pattern (nhóm cấu trúc)**
  - **Behavioral patterns (nhóm ứng xử)**
- Có tổng cộng khoảng 32 (9+11+12) design pattern thông dụng.

4.2. CÁC PATTERN THIẾT KẾ HĐT

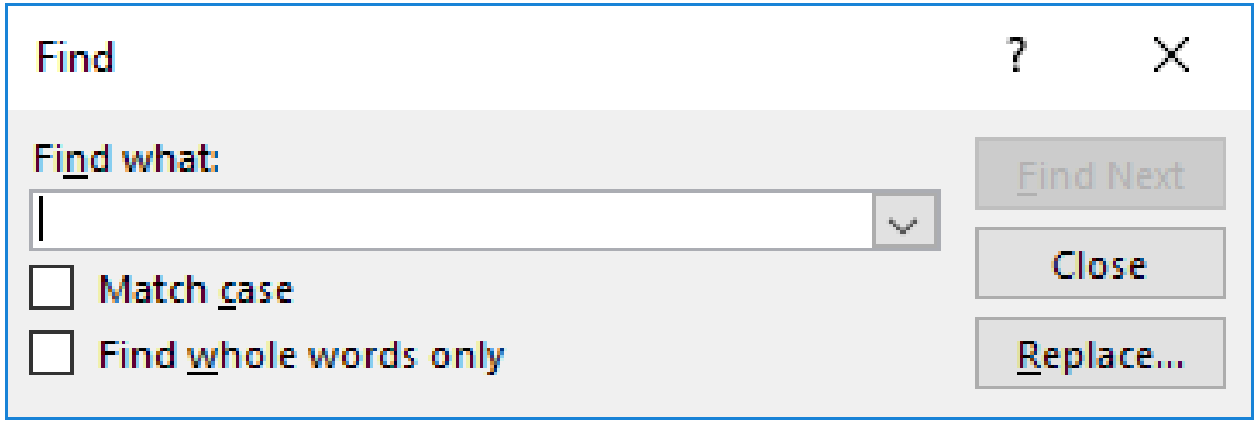
4.2.1. Singleton Pattern

4.2.2. Composite Pattern

4.2.3. Strategy

4.2.1. SINGLETON

- Là pattern đảm bảo rằng một lớp chỉ có một thể hiện (instance) duy nhất
- Cung cấp một cổng giao tiếp chung nhất để truy cập vào lớp đó.



4.2.1. SINGLETON

- Ví dụ 1: Có thể có rất nhiều máy in (Printer) trong hệ thống nhưng chỉ có thể tồn tại duy nhất một Phần quản lý máy in (Sprinter Spooler)

4.2.1. SINGLETON

- Ví dụ 2: Trong ứng dụng có chức năng bật, tắt nhạc nền của một ứng dụng. Trong setting của app cho phép người dùng quản lí việc mở hay tắt nhạc. Không thể tạo 1 instance để mở nhạc rồi sau đó lại tạo 1 instance khác để tắt nhạc => cần sử dụng singleton để quản lí việc này.

4.2.1. SINGLETON

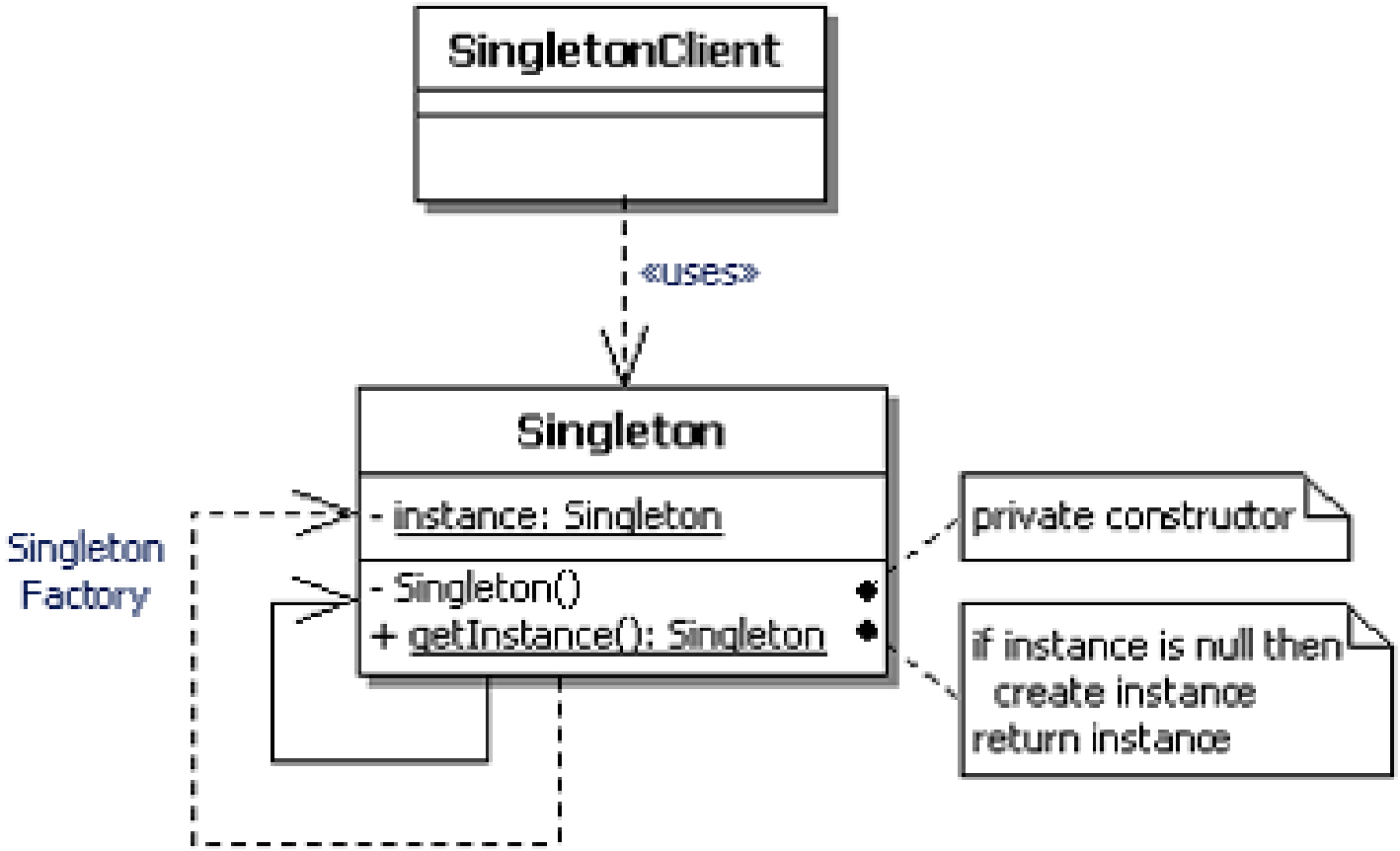
- **Vai trò**
  - Quản lý việc truy cập tốt hơn vì chỉ có một thể hiện đơn nhất.
  - Quản lý số lượng thể hiện (đối tượng) của một lớp
  - Khả chuyển hơn so với việc dùng một lớp có thuộc tính là static

4.2.1. SINGLETON

- **Cách tạo Singleton pattern**
  - Xây dựng Private constructor để đảm bảo rằng class lớp khác không thể truy cập vào constructor và tạo ra instance mới.
  - Sử dụng private static để đảm bảo thể hiện của class là duy nhất.
  - Tạo một public static menthod getInstance() trả về instance giúp client có thể truy cập vào đối tượng

4.2.1. SINGLETON

- **Cách tạo Singleton pattern**



4.2.1. SINGLETON

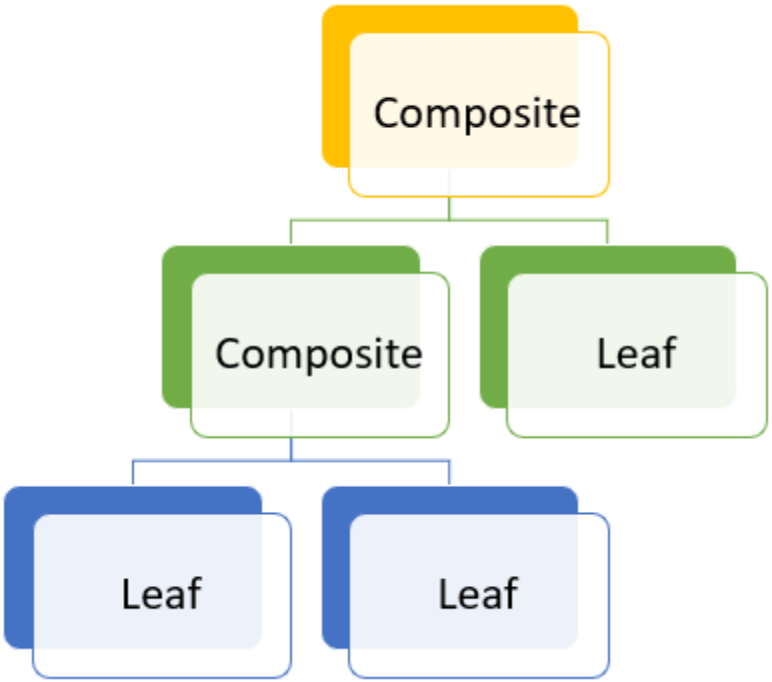
- **Cách tạo Singleton pattern**

```
3  public class Singleton {
4      public static Singleton instance;
5      private Singleton(){
6          //khoi tao du lieu tai day
7      }
8      public static Singleton getInstance()
9      {
10         if(instance==null)
11             instance=new Singleton();
12         return instance;
13     }
14     public void methodDoTask(){
15         //xu ly cong viec tai day
16     }
17 }
```

4.2.2. COMPOSITE PATTERN

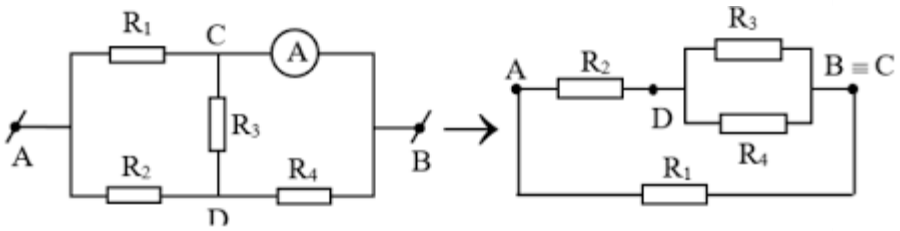
- Composite là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern).
- Composite Pattern là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn.
- Composite cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.

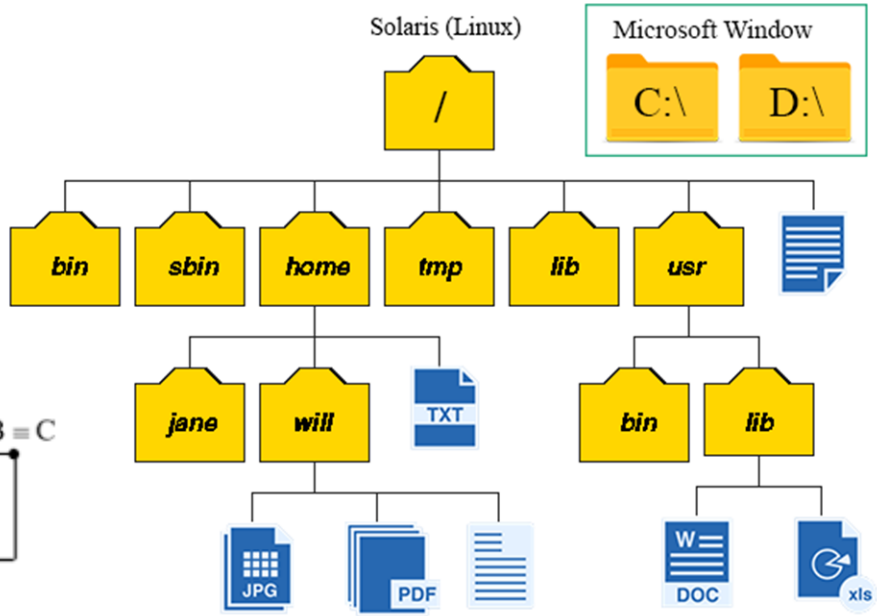
4.2.2. COMPOSITE PATTERN



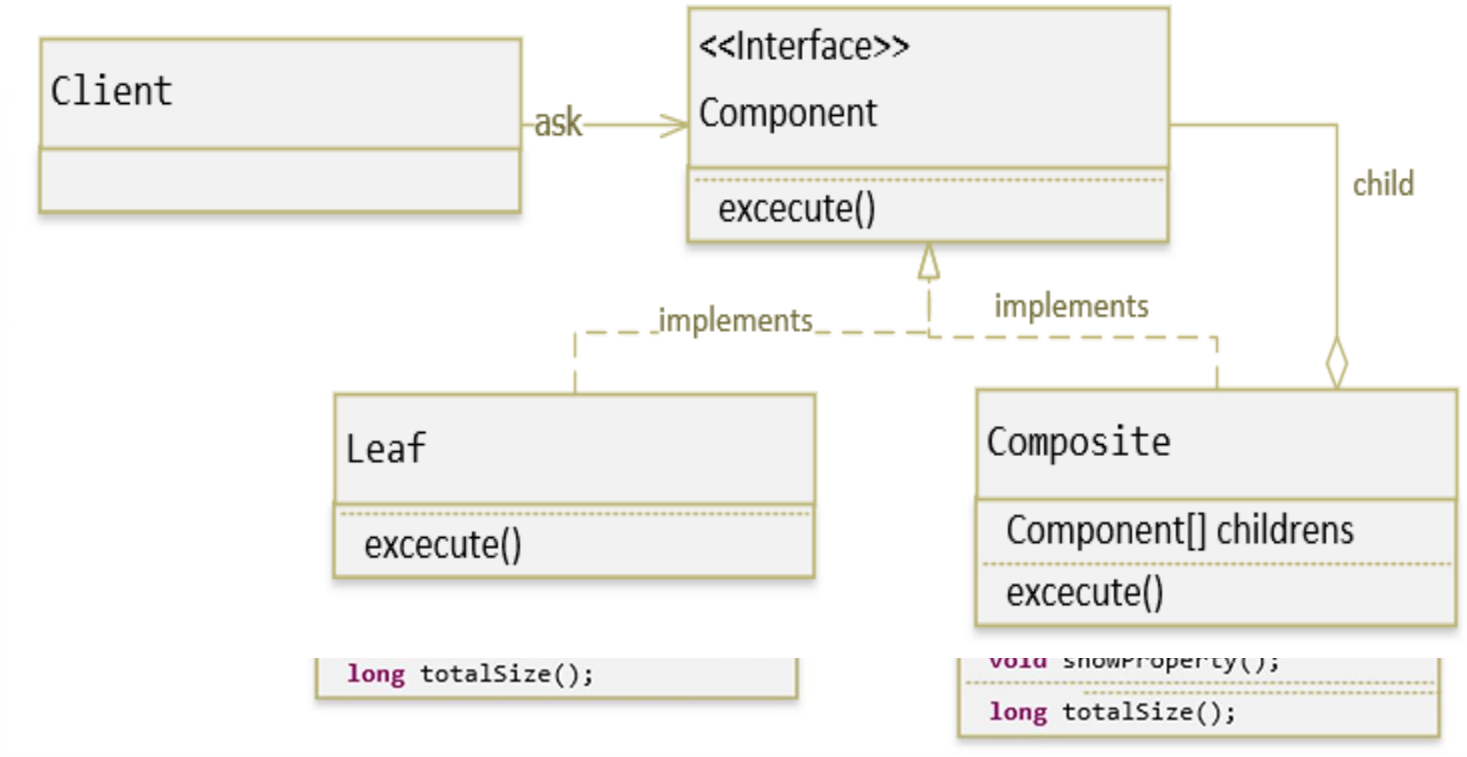
4.2.2. COMPOSITE PATTERN

- VD1: Thiết kế lớp cho chương trình quản lý thư mục, tập tin trong máy tính
- VD2: Thiết kế lớp cho chương trình mô tả mạnh điện trong thực tế.





4.2.2. COMPOSITE PATTERN



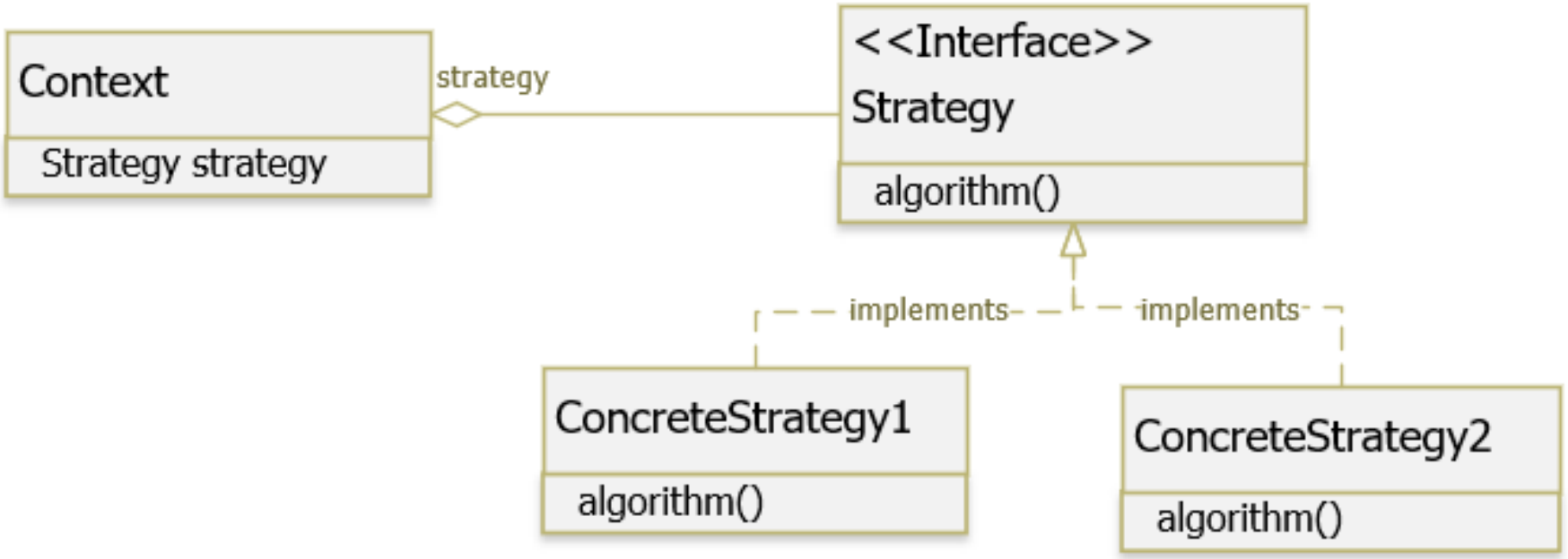
4.2.2. COMPOSITE PATTERN

- Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
- Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

4.2.3. STRATEGY PATTERN

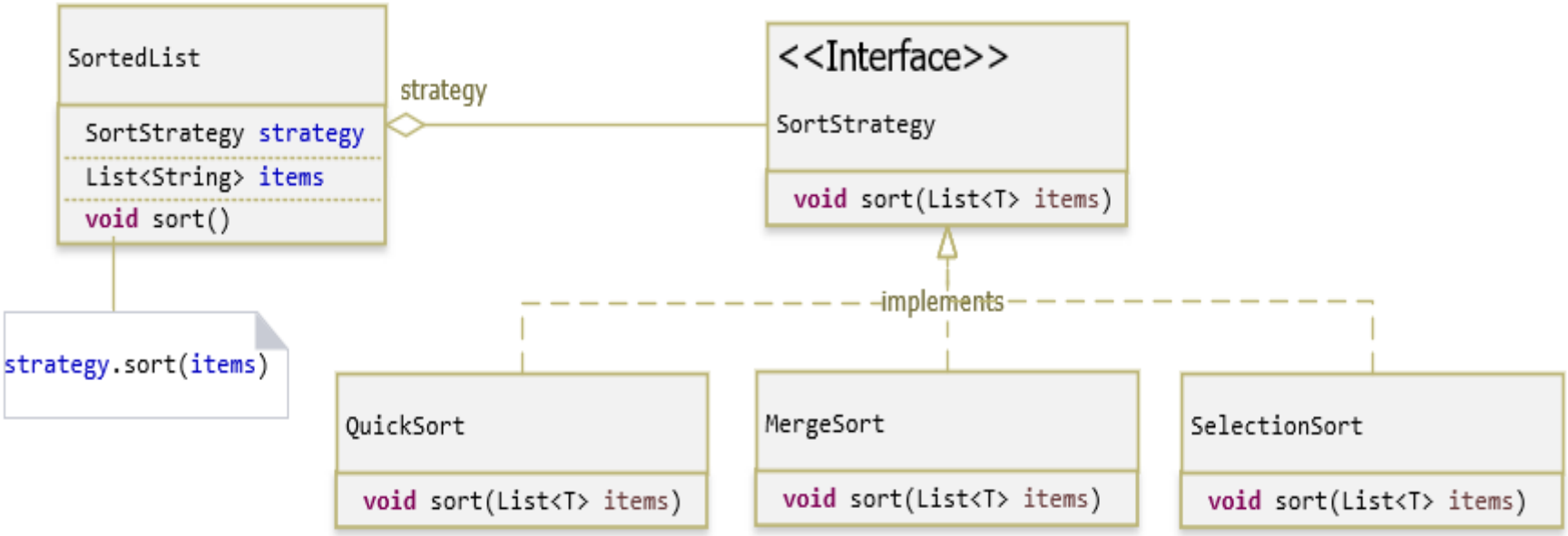
- Strategy Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).
- Strategy cho phép định nghĩa tập hợp các thuật toán, dễ dàng thay đổi linh hoạt các thuật toán bên trong object.
- Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.
- Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng.

4.2.3. STRATEGY PATTERN



4.2.3. STRATEGY PATTERN

- VD: Chương trình cung cấp nhiều giải thuật sắp xếp khác nhau: quick sort, merge sort, selection sort, heap sort, tim sort, ….



4.2.3. STRATEGY PATTERN

- Có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
- Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
- Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
- Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.