

# SYSTEM MODELLING

## OOAD

Hoan Ng

# Overview

- Tổng quan về phương pháp OOAD
- Các bước của phương pháp OOAD
  - Phân tích hướng đối tượng
  - Thiết kế hướng đối tượng
  - Lập trình hướng đối tượng
- ▶ Các nguyên tắc hướng đối tượng
  - ▶ 4 nguyên tắc cơ bản của OOP
  - ▶ 5 nguyên tắc hàng đầu của OOD
- ▶ So sánh giữa hướng đối tượng và hướng cấu trúc
- ▶ Tài liệu tham khảo

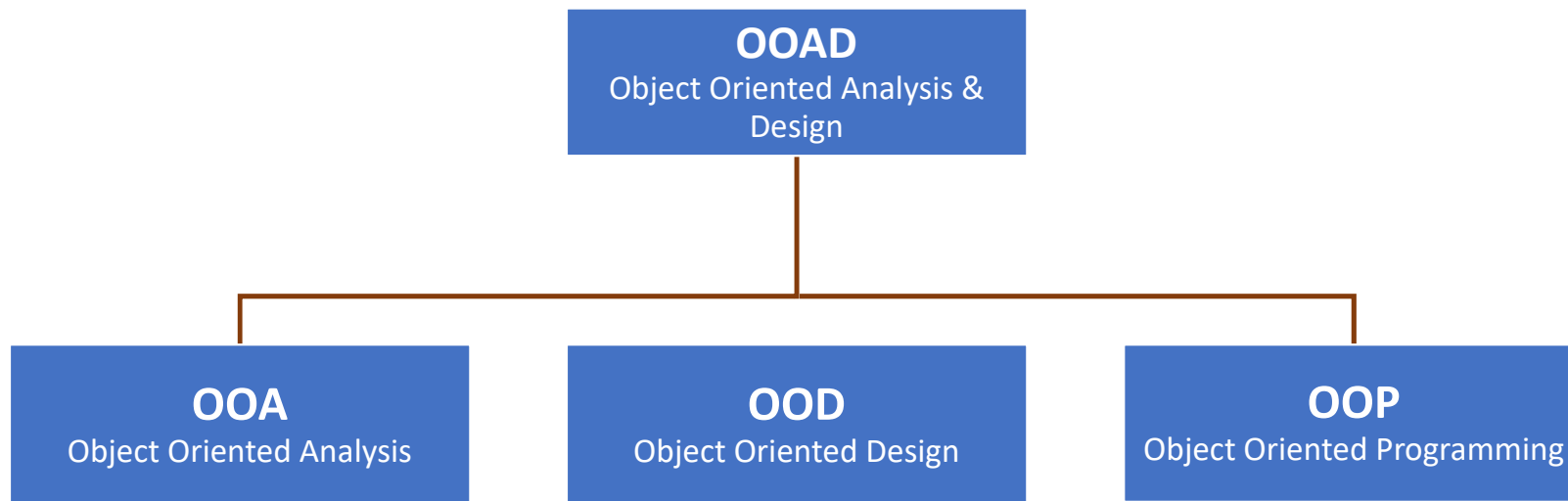
# Tổng quan về phương pháp OOAD

- OOAD –là viết tắt của **O**bject-**O**riented **A**nalysis and **D**esign (Phân tích và thiết kế hướng đối tượng)
- Trong đó, mô hình hệ thống được xem như là một nhóm các đối tượng có tương tác với nhau.
- Mỗi đối tượng đại diện cho một số thực thể đáng quan tâm, và được đặc trưng bởi các lớp, các trạng thái (dữ liệu), và hành vi.

# Tổng quan về phương pháp OOAD

- OOAD là một trong ba nhánh chính của phân tích và thiết kế hệ thống, bao gồm:
  - **Phương pháp hướng đối tượng (OOAD)**
    - Phương pháp hướng cấu trúc
    - Phương pháp hướng thích nghi
- Sử dụng bộ ngôn ngữ UML (*ngôn ngữ mô hình hóa thống nhất*)

# Các bước trong phương pháp OOAD



# Các bước trong phương pháp OOAD

## Object Oriented Analysis – OOA:

Giai đoạn này trình bày vấn đề bằng các thuật ngữ tương ứng với các đối tượng có thực, và phải được định nghĩa sao cho người không chuyên cũng có thể hiểu được, đồng thời vẫn giữ nguyên các mẫu hình về cấu trúc, quan hệ cũng như hành vi của chúng

- **Xác định use case**
  - *Use case là một văn bản hoặc một câu chuyện diễn tả cách làm việc của hệ thống*
- **Xác định domain model**
  - *Domain model là mô hình các khái niệm, sự vật quan trọng trong miền ứng dụng và quan hệ ràng buộc giữa chúng*

# Object Oriented Analysis – OOA:

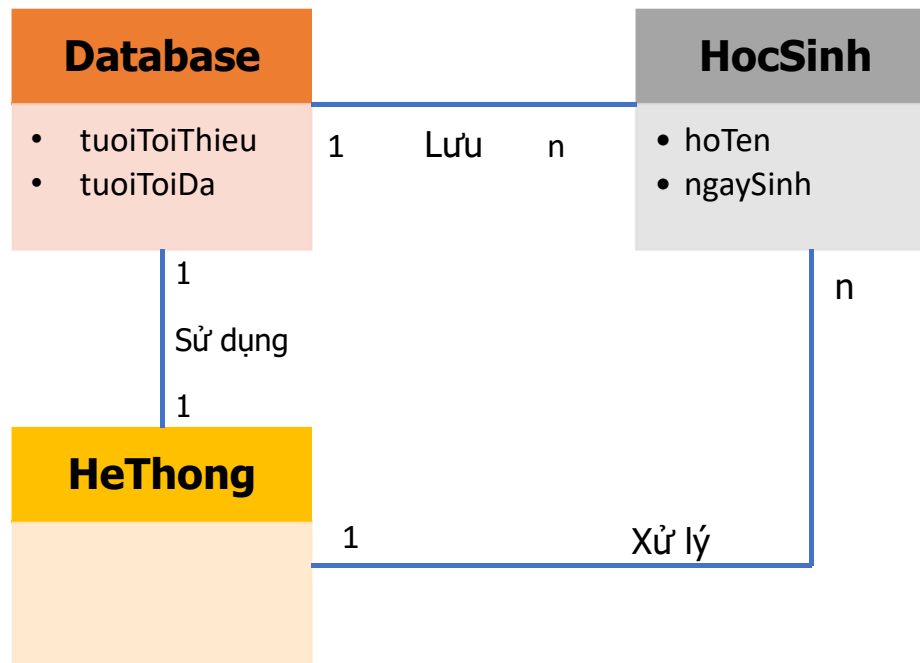
Vd 1:

Use case:

Nhập 1 học sinh mới vào danh sách:

- Người dùng nhập Họ tên, ngày sinh của học sinh
- Hệ thống tính tuổi của học sinh và kiểm tra xem độ tuổi có từ 15-20 không
- Nếu nằm trong độ tuổi 15-20 thì nhận vào danh sách học sinh của trường
- Nếu không thì báo về người dùng là độ tuổi không phù hợp

Domain model:



# Object Oriented Analysis – OOA:

## Vd 2:

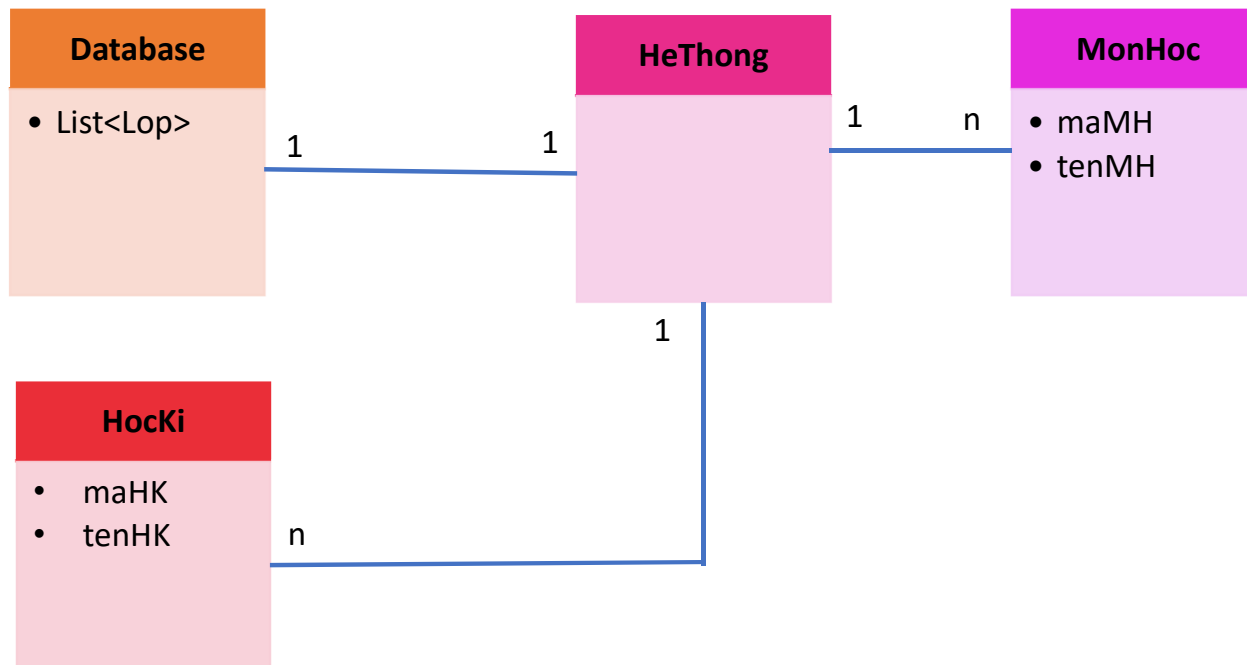
Use case: Người dùng cần tra Danh sách các lớp đã học môn nào đó trong học kì nào đó

1. Người dùng nhập mã môn học, mã học kì cần tra
2. Hệ thống dựa theo mã đó tra cứu trong Danh sách các lớp của trường
3. Trả về cho hệ thống các lớp nào có học môn đó trong học kì đó



# Object Oriented Analysis – OOA:

Domain model:



## Các bước trong phương pháp OOAD

### Object Oriented Design – OOD:

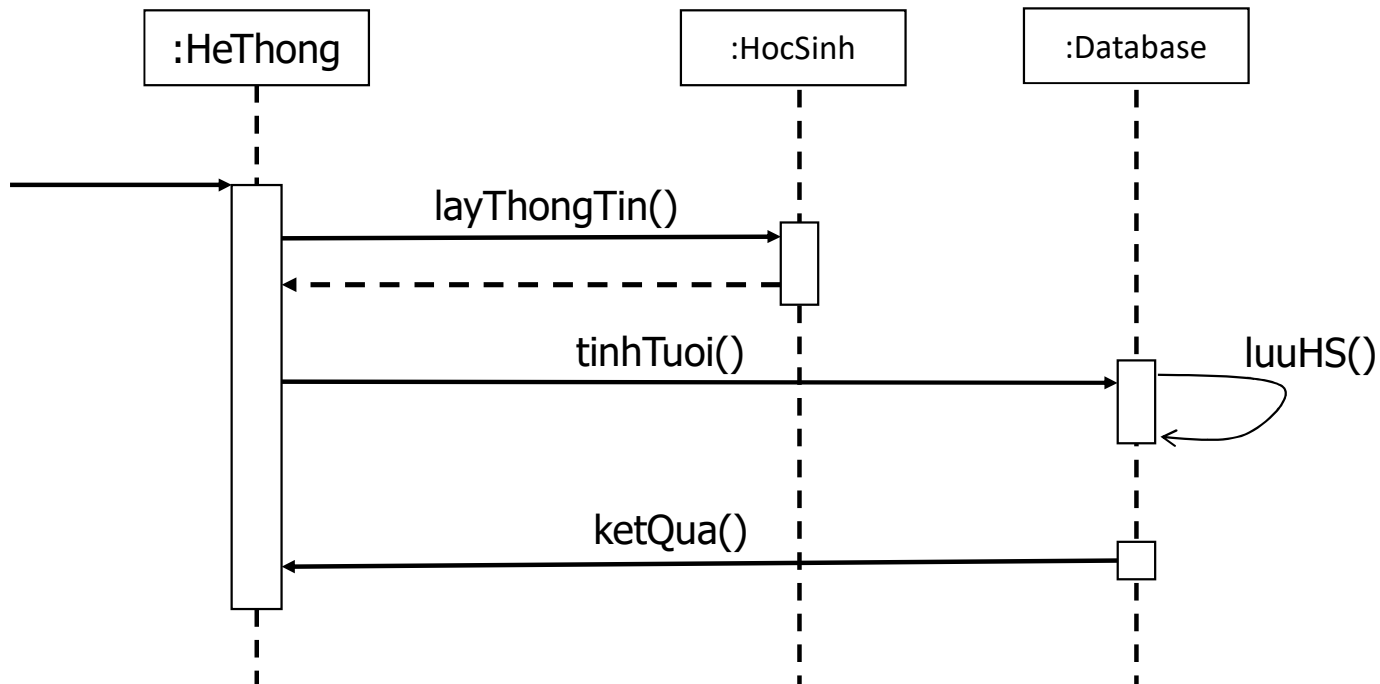
Là giai đoạn tổ chức chương trình thành các tập hợp đối tượng , mỗi đối tượng trong đó là thực thể của một lớp. Các lớp là thành viên của một cây cấu trúc với mối quan hệ thừa kế.

- **Xác định Interaction Diagram**
  - *Thể hiện dòng thông điệp đi giữa các object*
- **Xác định Design Model**
  - *Xác định thêm các thuộc tính, phương thức nhằm giải bài toán phần mềm*

# Object Oriented Design – OOD:

Vd 1:

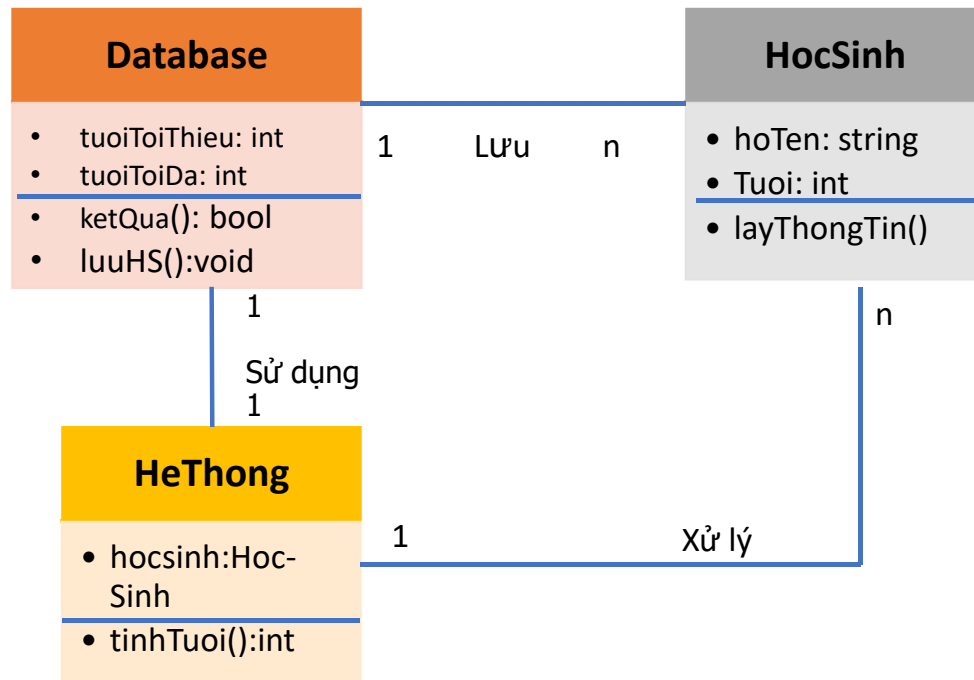
Interaction Diagram:



# Object Oriented Design – OOD:

Vd 1:

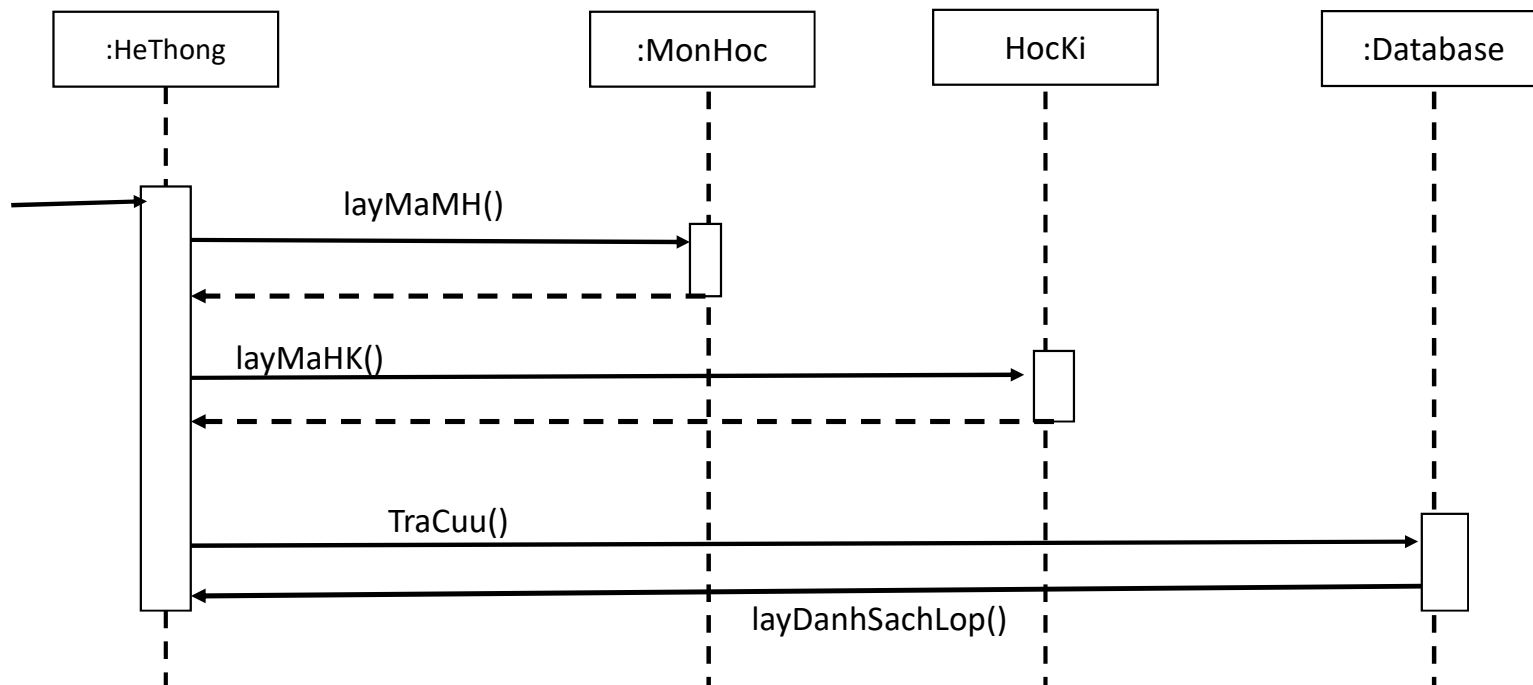
Design Model:



# Object Oriented Design – OOD:

Vd 2:

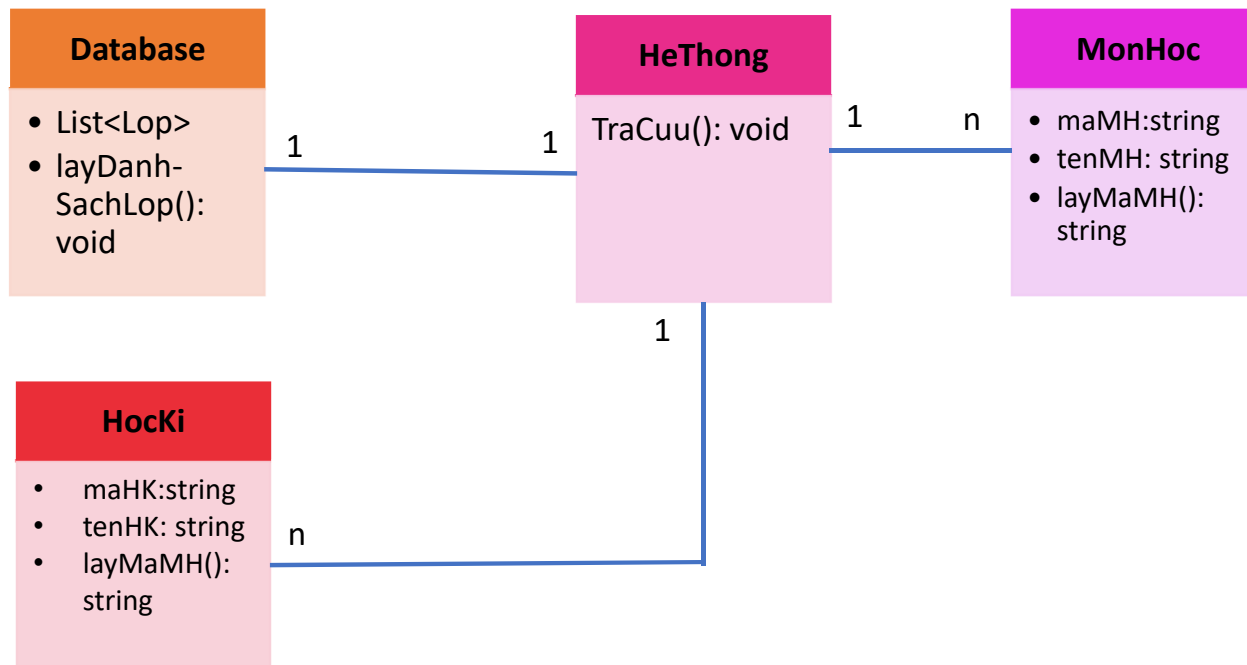
Interaction Diagram:



# Object Oriented Design – OOD:

Vd 2:

Design Model:



# Các bước trong phương pháp OOAD

## Object Oriented Programming – OOP:

Là giai đoạn hiện thực hóa phần mềm dựa trên kết quả của OOD. Sử dụng các ngôn ngữ lập trình hỗ trợ hướng đối tượng để chuyển hóa mô hình thành code.

Một số ngôn ngữ hỗ trợ hướng đối tượng:

- C++
- Java
- C#
- Perl
- Python
- PHP

# Các nguyên tắc của hướng đối tượng

## 4 nguyên tắc căn bản của OOP:

- Abstraction : tính trừu tượng.
- Encapsulation : tính đóng gói.
- Inheritance : tính kế thừa.
- Polymorphism : tính đa hình.



# Các nguyên tắc của hướng đối tượng

## Tính trừu tượng:

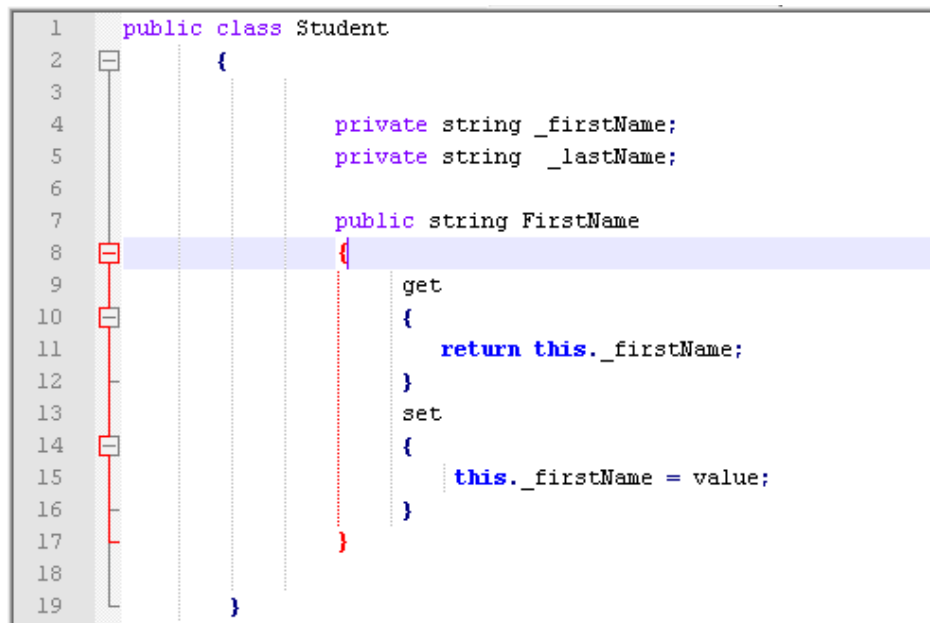
Khi thiết kế các đối tượng, ta cần rút tĩa ra những đặc trưng của chúng, rồi trừu tượng hóa thành các interface, và thiết kế xem chúng sẽ tương tác với nhau như thế nào. Nói cách khác, chúng ta định ra các interface và các contract mà chúng cần thỏa mãn.

```
1  public class Student
2  {
3      public string FirstName{get;set;}
4      public string LastName{get;set;}
5      public int Age{get;set;}
6      public static string Class
7      {
8          get;set;
9      }
10     public string override ToString()
11     {
12         return FirstName + " " + LastName;
13     }
14 }
15
```

# Các nguyên tắc của hướng đối tượng

## Tính đóng gói:

Tính chất này không cho phép người sử dụng thay đổi trạng thái nội tại của một đối tượng. Chỉ có các phương thức nội tại của đối tượng cho phép thay đổi trạng thái của nó. Việc cho phép môi trường bên ngoài tác động lên các dữ liệu nội tại của một đối tượng theo cách nào là hoàn toàn tùy thuộc vào người viết mã. Đây là tính chất đảm bảo sự toàn vẹn của đối tượng.



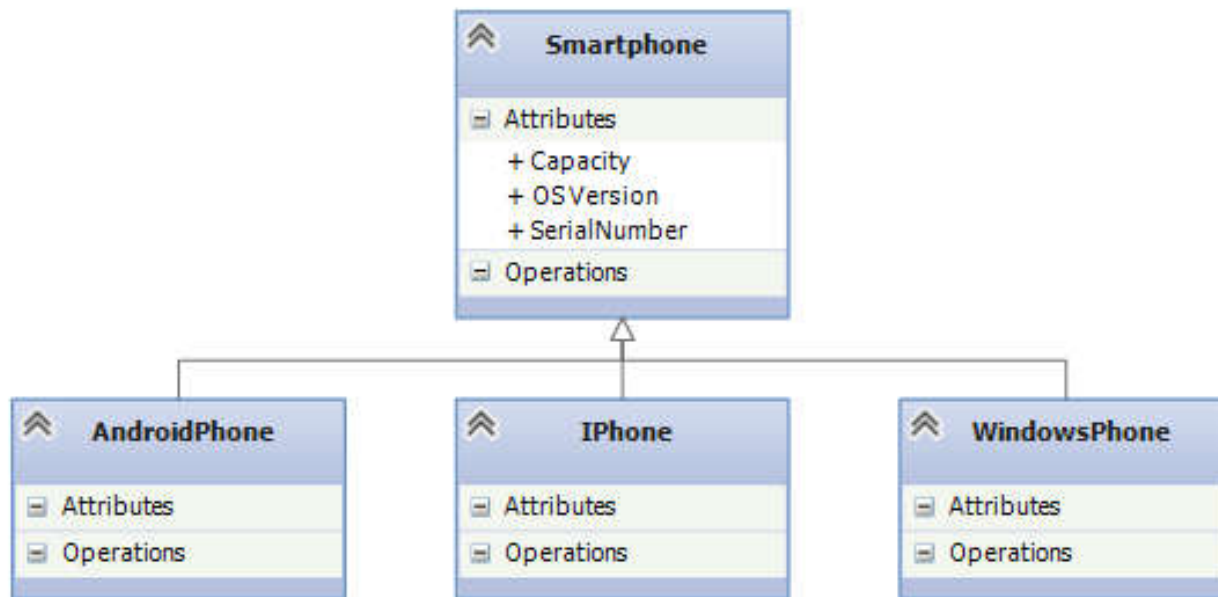
The image displays a UML class diagram and the corresponding C# code for a `Student` class, illustrating the principle of encapsulation. The UML diagram on the left shows the class structure with private attributes `_firstName` and `_lastName`, and a public attribute `FirstName`. The C# code on the right defines the `Student` class with these attributes and a `FirstName` property that uses `get` and `set` methods to access the private `_firstName` attribute.

```
1 public class Student
2 {
3
4     private string _firstName;
5     private string _lastName;
6
7     public string FirstName
8     {
9         get
10        {
11            return this._firstName;
12        }
13        set
14        {
15            this._firstName = value;
16        }
17    }
18 }
19
```

# Các nguyên tắc của hướng đối tượng

## Tính kế thừa:

Đặc tính này cho phép một đối tượng có thể có sẵn các đặc tính mà đối tượng khác đã có thông qua kế thừa. Điều này cho phép các đối tượng chia sẻ hay mở rộng các đặc tính sẵn có mà không phải tiến hành định nghĩa lại



# Các nguyên tắc của hướng đối tượng

## Tính kế thừa:

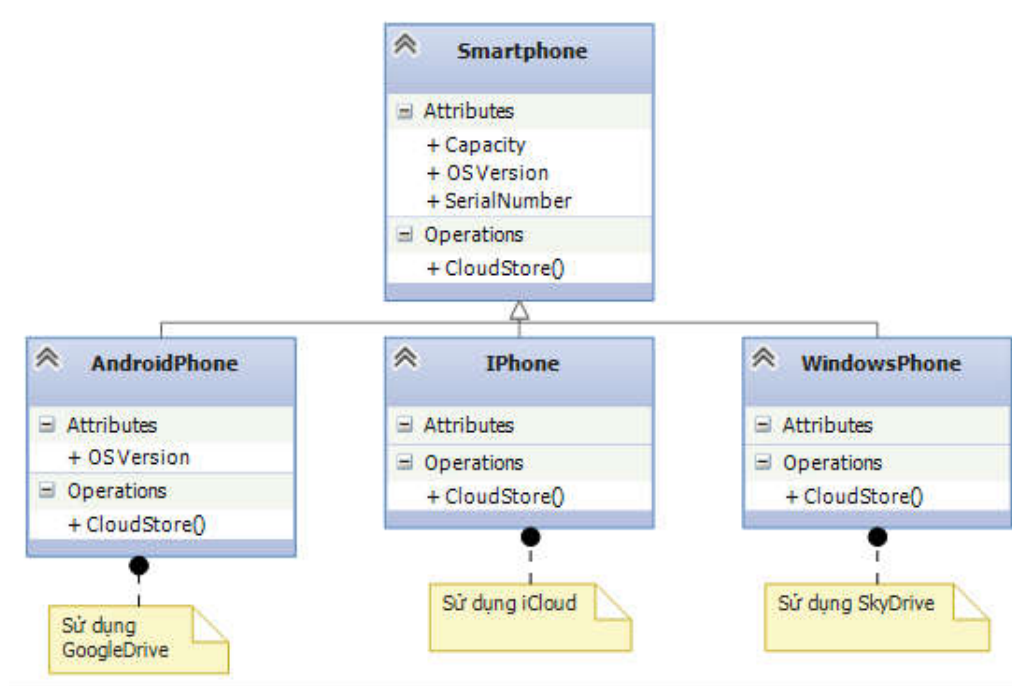
Nếu các chức năng của lớp cha đã được định nghĩa đầy đủ thì lập trình viên sẽ không phải làm bất cứ việc gì với lớp con. Còn nếu một lớp con muốn chức năng khác so với định nghĩa ở lớp cha thì nó có thể dễ dàng ghi đè (override) chức năng đã được định nghĩa trên lớp cha



# Các nguyên tắc của hướng đối tượng

## Tính đa hình:

Đa hình là khái niệm mà hai hoặc nhiều lớp có những phương thức giống nhau nhưng có thể thực thi theo những cách thức khác nhau.



# Áp dụng

## Great software in 3 easy steps

1. Make sure your software does what the customer wants it to do.

It may not seem easy now, but we'll show you how OOAD and some basic principles can change your software forever.

← This step focuses on the customer. Make sure the app does what it's supposed to do FIRST. This is where getting good requirements and doing some analysis comes in.

2. Apply basic OO principles to add flexibility.

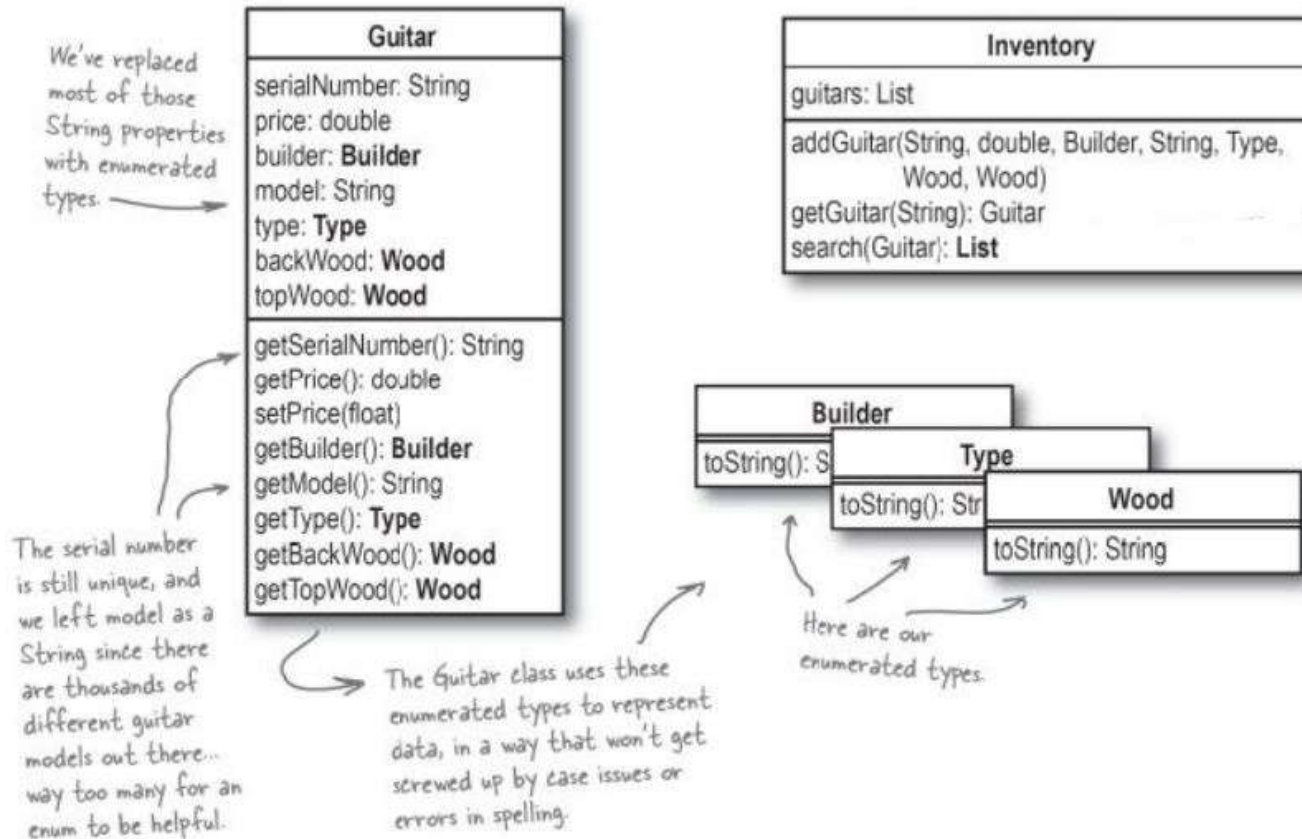
Once your software works, you can look for any duplicate code that might have slipped in, and make sure you're using good OO programming techniques.

3. Strive for a maintainable, reusable design.

Got a good object-oriented app that does what it should? It's time to apply patterns and principles to make sure your software is ready to use for

# Áp dụng

Vd1:



# Áp dụng

Here's the test program, updated to use the new version of Rick's search tool.

```
public class FindGuitarTester {  
  
    public static void main(String[] args) {  
        // Set up Rick's guitar inventory  
        Inventory inventory = new Inventory();  
        initializeInventory(inventory);  
  
        Guitar whatErinLikes = new Guitar("", 0, Builder.FENDER,  
                                           "Stratocaster", Type.ELECTRIC,  
                                           Wood.ALDER, Wood.ALDER);  
  
        List matchingGuitars = inventory.search(whatErinLikes);  
        if (!matchingGuitars.isEmpty()) {  
            System.out.println("Erin, you might like these guitars:");  
            for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {  
                Guitar guitar = (Guitar)i.next();  
                System.out.println("  We have a " +  
                                   guitar.getBuilder() + " " + guitar.getModel() + " " +  
                                   guitar.getType() + " guitar:\n  " +  
                                   guitar.getBackWood() + " back and sides,\n  " +  
                                   guitar.getTopWood() + " top.\n  You can have it for only $" +  
                                   guitar.getPrice() + "!\n  ----");  
            }  
        } else {  
            System.out.println("Sorry, Erin, we have nothing for you.");  
        }  
    }  
}
```

We're using enumerated types in this test drive. No typing mistakes this time!

In this new version, we need to iterate over all the choices returned from the search tool.

This time we get a whole list of guitars that match the client's specs.

```
class  
FindGuitar  
{  
    main()  
}
```

FindGuitarTester.java



# Áp dụng

Problem?



If you've got an object that is being used with no-value or null properties often, you've probably got an object doing more than one job. If you rarely have values for a certain property, why is that property part of the object? Would there be a better object to use with just a subset of those properties?

Solve

**Encapsulation**  
allows you to  
group your  
application into  
logical parts.

Anytime you see  
duplicate code, look for a  
place to encapsulate!

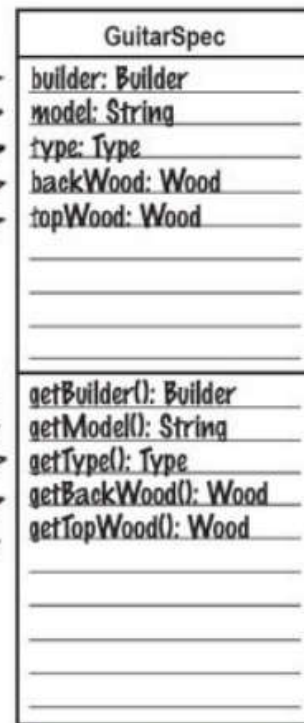
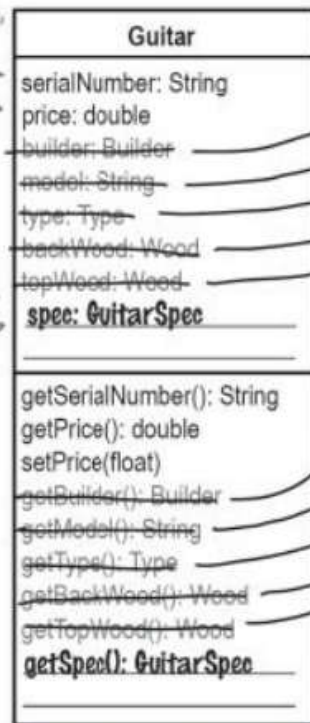
# Áp dụng

These two properties are still unique to each Guitar, so they stay.

These are the properties that Rick's clients supply to search(), so we can move them into GuitarSpec.

We also need a reference to a GuitarSpec object for each guitar.

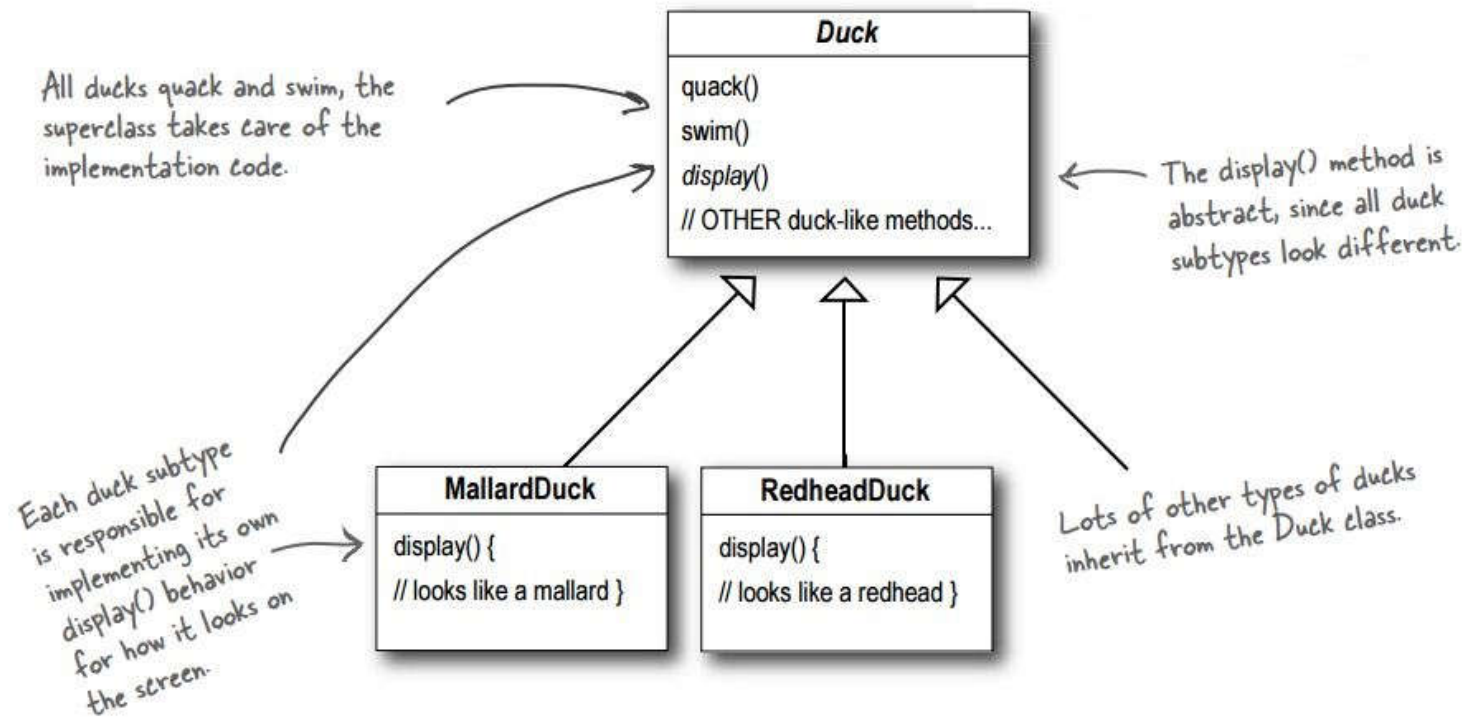
The methods follow the same pattern as the properties: we remove any duplication between the client's specs and the Guitar object.



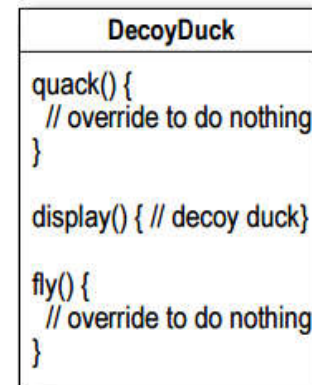
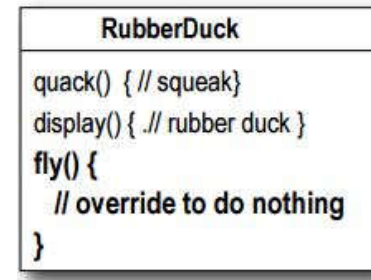
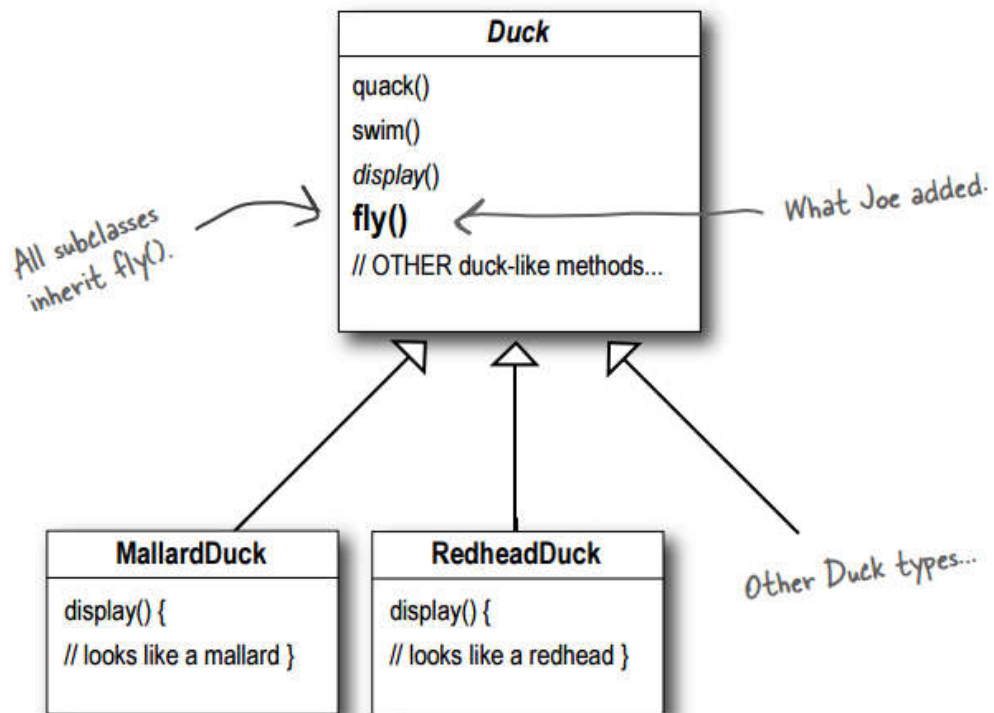
We've removed duplicated code by moving all the common properties—and related methods—into an object that we can use for both search requests and guitar details.

# Áp dụng

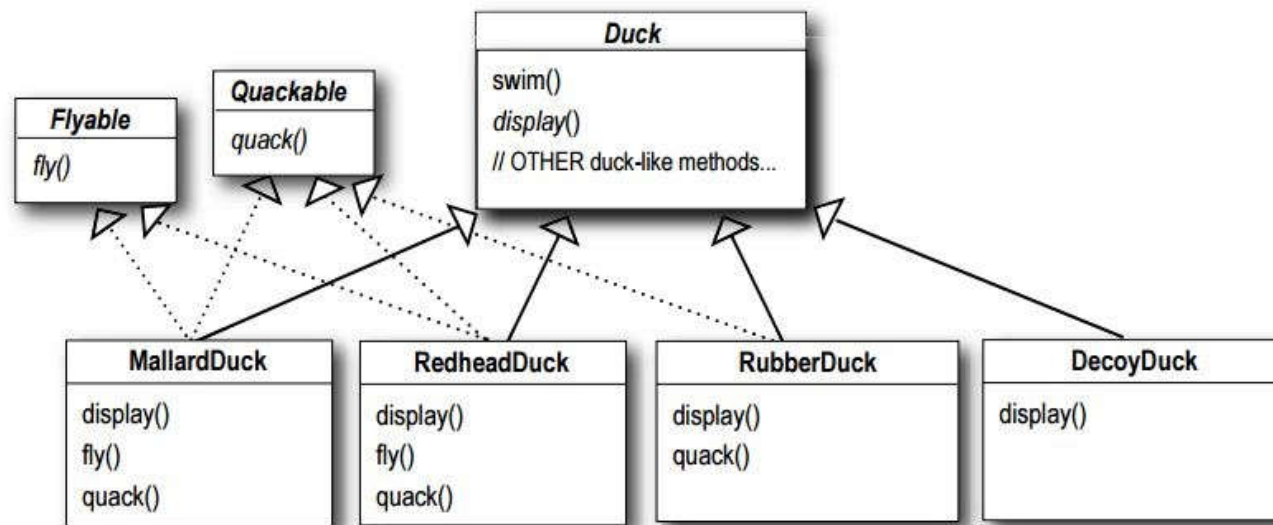
Vd2:



# Áp dụng



## Áp dụng



# Áp dụng



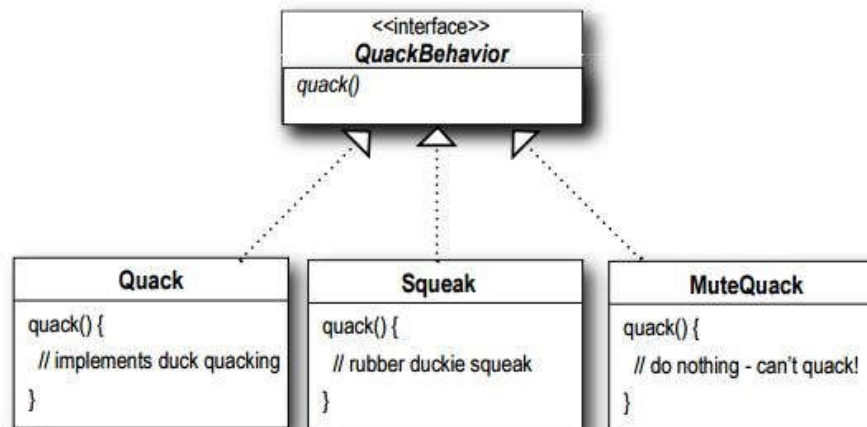
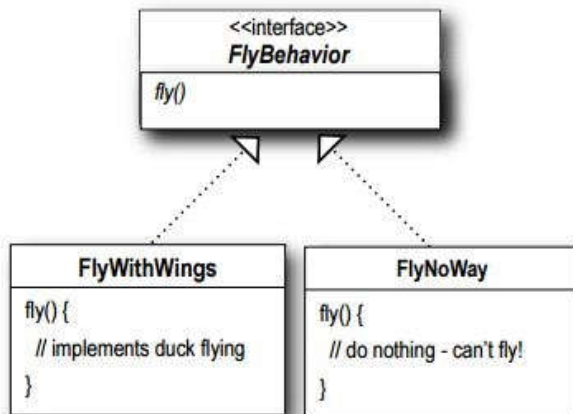
## Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.



## Design Principle

Program to an interface, not an implementation.

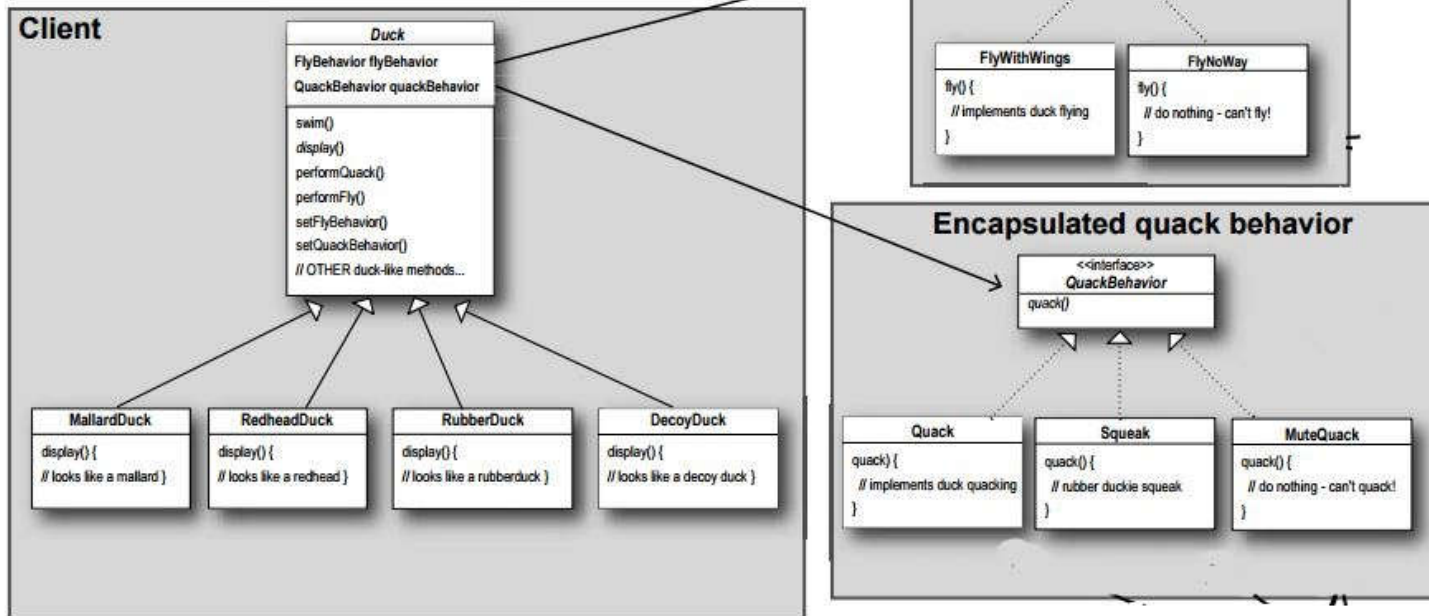


# Áp dụng



## Design Principle

*Favor composition over inheritance.*



# Các nguyên tắc của hướng đối tượng

## S.O.L.I.D: The First 5 Principles of Object Oriented Design

S.O.L.I.D là một từ viết tắt cho 5 nguyên tắc hàng đầu của thiết kế hướng đối tượng (OOD), được đề xuất bởi Robert C. Martin, thường được gọi là Bác Bob.


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




## S – Single-responsibility principle

Một class chỉ nên giữ một trách nhiệm duy nhất


```
public class ReportManager()  
{  
    public void ReadDataFromDB();  
    public void ProcessData();  
    public void PrintReport();  
}
```




```
public class ReadDataFromDB()  
{  
}
```



```
public class ProcessData()  
{  
}
```



```
public class PrintReport()  
{  
}
```



## O – Open-closed principle

Có thể thoải mái mở rộng 1 class,  
nhưng không được sửa đổi bên trong class đó

Mỗi khi ta muốn thêm chức năng,... cho chương trình, chúng ta nên viết class mới mở rộng class cũ ( bằng cách kế thừa class cũ) không nên sửa đổi class cũ.

```
class Customer
{
    private int _CustType;

    public int CustType
    {
        get { return _CustType; }
        set { _CustType = value; }
    }


    public double getDiscount(double TotalSales)
    {
        if (_CustType == 1)
        {
            return TotalSales - 100;
        }
        else
        {
            return TotalSales - 50;
        }
    }
}
```




## O – Open-closed principle

```
class Customer
{
    public virtual double getDiscount(double TotalSales)
    {
        return TotalSales;
    }
}

class SilverCustomer : Customer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 50;
    }
}
```



```
class goldCustomer : SilverCustomer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 100;
    }
}
```



## L – Liskov substitution principle

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

```
class Enquiry : Customer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 5;
    }

    public override void Add()
    {
        throw new Exception("Not allowed");
    }
}
```



## L – Liskov substitution principle


```
List<Customer> Customers = new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

foreach (Customer o in Customers)
{
    o.Add();
}
```



```
class Enquiry : Customer
{
    public override double getDiscount(double
    {
        return base.getDiscount(TotalSales)
    }

    public override void Add()
    {
        throw new Exception("Not allowed");
    }
}
```

 **Exception was unhandled**

Not allowed

**Troubleshooting tips:**

[Get general help for this exception.](#)

[Search for more Help Online...](#)

**Exception settings:**

☐ Break when this exception type is

## L – Liskov substitution principle

```
interface IDiscount
{
    double getDiscount(double TotalSales);
}

interface IDatabase
{
    void Add();
}
```

```
class Enquiry : IDiscount
{
    public double getDiscount(double TotalSales)
    {
        return TotalSales - 5;
    }
}
```

```
class Customer : IDiscount, IDatabase
{
    private MyException obj = new MyException();

    public virtual void Add()
    {
        try
        {
            // Database code goes here
        }
        catch (Exception ex)
        {
            obj.Handle(ex.Message.ToString());
        }
    }

    public virtual double getDiscount(double TotalSales)
    {
        return TotalSales;
    }
}
```

## L – Liskov substitution principle


```
List<IDatabase> Customers = new List<IDatabase>();  
Customers.Add(new SilverCustomer());  
Customers.Add(new goldCustomer());  
Customers.Add(new Enquiry());  
  
foreach (IDatabase o in Customers)  
{  
    o.Add();  
}
```




## I – Interface segregation principle

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

```
interface IDatabase
{
    void Add(); // old client are happy with these.
    void Read(); // Added for new clients.
}
```




```
interface IDatabaseV1 : IDatabase
{
    void Read();
}
```



```
class CustomerwithRead : IDatabase, IDatabaseV1
{
    public void Add()
    {
        Customer obj = new Customer();
        Obj.Add();
    }

    public void Read()
    {
        // Implements logic for read
    }
}
```





## I – Interface segregation principle

```
IDatabase i = new Customer(); // 1000 happy old clients not touched  
i.Add();  
  
IDatabaseV1 iv1 = new CustomerWithread(); // new clients  
iv1.Read();
```



## D – Dependency Inversion Principle

1. Các module cấp cao không nên phụ thuộc vào các modules cấp thấp.  
Cả 2 nên phụ thuộc vào abstraction.
2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại.  
(Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

```
public class ComputerShop
{
    public enum ComputerType
    {
        Laptop, Desktop, Tablet // Violation of OCP
    }

    public ComputerShop()
    {
    }

    public string GetMyComputerDescription(ComputerType cmptype)
    {
        var myComp = string.Empty;
        if (ComputerType.Desktop == cmptype)
        {
            myComp = new Desktop().GetComputer();
        }
        else if (ComputerType.Laptop == cmptype)
        {
            myComp = new Laptop().GetComputer();
        }
        else if (ComputerType.Tablet == cmptype) // Violation of OCP
        {
            myComp = new Tablet().GetComputer();
        }
        return myComp;
    }
}
```



```
public interface IComputer
{
    string GetComputerDescription();
}

public class Desktop : IComputer
{
    public string GetComputerDescription()
    {
        return " You get a Desktop";
    }
}

public class Laptop:IComputer
{
    public string GetComputerDescription()
    {
        return " You get a Laptop";
    }
}

public class Tablet : IComputer
{
    public string GetComputerDescription()
    {
        return " You get a new Tablet yahoooooooooooooooooooo";
    }
}

public class ComputerShop
{
    public string GetMyComputerDescription(IComputer cmptype)
    {
        // No matter how many types of computer comes
        var myComp = cmptype.GetComputerDescription();
        return myComp;
    }
}

var computer = new ComputerShop();
computer.GetMyComputerDescription(new Tablet());
```



## So sánh giữa hướng đối tượng và hướng cấu trúc

Phương pháp phân tích thiết kế hướng cấu trúc và hướng đối tượng có sự khác biệt nhau rất lớn

Phương pháp	Hướng đối tượng	Hướng cấu trúc
<b>Cách tiếp cận</b>	- Tập trung vào cả hai khía cạnh của hệ thống là dữ liệu và hành động.	- Chỉ tập trung vào dữ liệu hoặc hành động
	- Phân chia chương trình thành các phần nhỏ gọi là đối tượng. Mỗi đối tượng bao gồm đầy đủ cả dữ liệu và hành động liên quan đến đối tượng đó.	- Phân chia chương trình chính thành nhiều chương trình con thực một công việc xác định.
	- Tiếp cận từ dưới lên (bottom-up) . Bắt đầu từ những thuộc tính cụ thể của từng đối tượng sau đó tiến hành trừu tượng hóa thành các lớp (Đối tượng).	- Tiếp cận từ trên xuống (top-down). Phân rã các bài toán thành bài toán nhỏ hơn đến khi nhận được các bài toán có thể cài đặt được.

Phương pháp	Hướng đối tượng	Hướng cấu trúc
<b>Đặc trưng đóng gói</b>	- Dữ liệu được đóng gói để hạn chế truy nhập tự do, trực tiếp vào dữ liệu	- Đặc trưng là cấu trúc dữ liệu và giải thuật, mối quan hệ chặt chẽ của giải thuật vào cấu trúc dữ liệu
	- Cho phép sử dụng lại mã nguồn để tiết kiệm tài nguyên và công sức lập trình.	- Hạn chế tái sử dụng mã.
<b>Lĩnh vực áp dụng</b>	- Phương pháp hướng đối tượng thường được áp dụng cho các bài toán lớn, phức tạp, hoặc có nhiều luồng dữ liệu khác nhau mà phương pháp cấu trúc không thể quản lý được. Khi đó người ta dùng phương pháp hướng đối tượng để tận dụng khả năng bảo vệ giữ liệu ngoài ra còn tiết kiệm công sức và tài nguyên .	- Phương pháp hướng cấu trúc thường phù hợp với nhiều bài toán nhỏ, có luồng dữ liệu rõ ràng, cần phải t duy giải thuật rõ ràng và người lập trình có khả năng tự quản lý được mọi truy cập đến các dữ liệu của chương trình.

Phương pháp	Hướng đối tượng	Hướng cấu trúc
<b>Ưu nhược điểm</b>	<p>Ưu điểm:</p> <ul style="list-style-type: none"> <li>+ Gần gũi với thế giới thực.</li> <li>+ Tái sử dụng dễ dàng.</li> <li>+ Đóng gói che giấu thông tin làm cho hệ thống tin cậy hơn.</li> <li>+ Thừa kế làm giảm chi phí, hệ thống có tính mở cao hơn</li> <li>+ Xây dựng được hệ thống phức tạp.</li> </ul>	<p>- Ưu điểm:</p> <ul style="list-style-type: none"> <li>+ Tư duy phân tích thiết kế rõ ràng.</li> <li>+ Chương trình sáng sửa dễ hiểu.</li> <li>+ Phân tích được các chức năng của hệ thống .</li> <li>+ Dễ theo dõi luồng dữ liệu.</li> </ul>
	<p>- Nhược điểm:</p> <ul style="list-style-type: none"> <li>+ Phương pháp này khá phức tạp, khó theo dõi được luồng dữ liệu do có nhiều luồng dữ liệu ở đầu vào. Hơn nữa giải thuật lại không phải là vấn đề trọng tâm của phương pháp này.</li> </ul>	<p>- Nhược điểm:</p> <ul style="list-style-type: none"> <li>+ Không hỗ trợ việc sử dụng lại. Các chương trình hướng cấu trúc phụ thuộc chặt chẽ vào cấu trúc dữ liệu và bài toán cụ thể, do đó không thể dùng lại modul nào đó trong phần mềm này cho phần mềm khác với các yêu cầu về dữ liệu khác.</li> <li>+ Không phù hợp cho phát triển các phần mềm lớn.</li> <li>+ khó quản lý mối quan hệ giữa các modul và dễ gây ra lỗi trong phân tích cũng như khó kiểm thử và bảo trì.</li> </ul>

Hướng đối tượng	Hướng cấu trúc
Mọi hoạt động (planning, analysis,...) đều chạy trong từng giai đoạn (vòng lặp)	Mỗi hoạt động (planning, analysis,...) chiếm hẳn 1 giai đoạn trong SDLC
Các giai đoạn gồm: <ul style="list-style-type: none"> <li>• Inception</li> <li>• Elaboration</li> <li>• Construction</li> <li>• Transition</li> </ul>	Các giai đoạn gồm: <ul style="list-style-type: none"> <li>• Planning</li> <li>• Analysis</li> <li>• Design</li> <li>• Implementation</li> <li>• Installation/ Testing</li> </ul>
Được áp dụng khi mục tiêu của phần mềm chưa rõ ràng, hệ thống linh hoạt	Được áp dụng khi mục tiêu của phần mềm đã rõ ràng, cố định
Công cụ sử dụng: UML (Use case, Class Diagram)	Công cụ sử dụng: Data Flow Diagram, Entity-Relationship Diagrams
Thích nghi với thực tế	Cố gắng dự đoán thực tế



# Tài liệu tham khảo

1. [https://www.tutorialspoint.com/object\\_oriented\\_analysis\\_design/index.htm](https://www.tutorialspoint.com/object_oriented_analysis_design/index.htm)
2. <http://www.codeproject.com/Articles/1137299/Object-Oriented-Analysis-and-Design>
3. [https://en.wikipedia.org/wiki/Object-oriented\\_analysis\\_and\\_design](https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design)
4. <http://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp><https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>
5. <https://tungnt185.wordpress.com/2013/11/07/cac-nguyen-ly-cua-lap-trinh-huong-doi-tuong-object-oriented-programming/>
6. <https://robot.bolink.org/ebooks/Head%20First%20Object%20Oriented%20Analysis%20and%20Design.pdf>
7. [https://people.ucalgary.ca/~far/Lectures/SENG401/PDF/OOAD\\_with\\_UML.pdf](https://people.ucalgary.ca/~far/Lectures/SENG401/PDF/OOAD_with_UML.pdf)
8. [https://www.u-cursos.cl/usuario/777719ab2ddbdb16d99df29431d3036/mi\\_blog/r/head\\_first\\_design\\_patterns.pdf](https://www.u-cursos.cl/usuario/777719ab2ddbdb16d99df29431d3036/mi_blog/r/head_first_design_patterns.pdf)