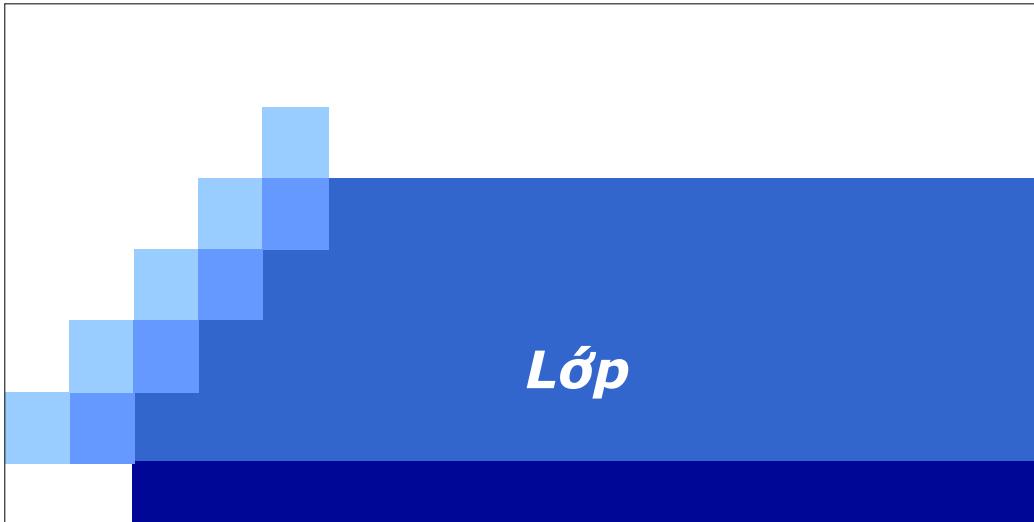


Review

# LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Kiến thức cơ bản

**Giảng viên: Huỳnh Tuấn Anh**  
**Khoa CNTT- Đại học Nha Trang**



*Lớp*

- ❖ Lớp
- ❖ Lớp trừu tượng
- ❖ Giao diện, thực thi giao diện
- ❖ Thừa kế
- ❖ Tính đa hình

## Lớp (class)

- ❖ Class: Sự đóng gói dữ liệu và các phương thức (method) xử lý trên dữ liệu đó
- ❖ Thành phần của lớp
  - Fields
  - Methods
  - Events
  - Properties
  - Các lớp lồng bên trong

## Khai báo lớp trong java

[access modifier] [static] [abstract]

**class\_name** [extends] [implements]

{

//class-members

...

}

5

## access-modifier trong java

### Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

6

## Khai báo lớp trong C#

attributes? unsafe? access-modifier?[ abstract| sealed ]?  
 class class-name  
 [: base-class | : interface+ | : base-class, interface+ ]?  
 { class-members }

5

## Khai báo lớp trong C#

attributes? unsafe? access-modifier?[ abstract| sealed ]?  
 class class-name



Bố từ	Giới hạn truy cập
<b>public</b>	Không hạn chế.
<b>private</b>	Chỉ có thể thấy được ở lớp hiện tại
<b>protected</b>	Chỉ có thể thấy được ở lớp hiện tại và lớp dẫn xuất
<b>internal</b>	Chỉ có thể thấy được trong cùng gói assembly (hợp ngữ) hiện tại
<b>protected internal</b>	Có thể thấy được ở lớp dẫn xuất (trong hay ngoài khối hợp ngữ) và bất cứ lớp nào trong cùng khối hợp ngữ hiện tại

7

8

## Khai báo lớp trong C#

attributes? unsafe? access-modifier?[ abstract| sealed ]?  
 class class-name  
 [: base-class | : interface+ | : base-class, interface+ ]?  
 { class-members }

9

## static member

- ❖ Là thành phần của một lớp
- ❖ Các biến, phương thức tĩnh được truy cập thông qua tên lớp
  - static member hoạt động giống như biến, phương thức toàn cục trong lập trình cấu trúc.
  - Phương thức tĩnh không thể truy cập đến một thành viên non-static trong cùng lớp
- ❖ Khai báo (trong c#/java):
  - [bỏ từ truy cập] <static> <tên kiểu> <tên thành viên>
  - Ví dụ:
    - public static int count;
    - public static void test() {...}

10

## properties trong c#

- ❖ Đặc tính mới trong một số ngôn ngữ LT HDT dùng để truy cập đến các trường dữ liệu bên trong lớp

❖ Ví dụ: C#  
 public class test  
 {  
 private int x;  
 public int X  
 {  
 get {return x;}  
 set {x=value;}  
 }  
 }



### C# 4.0

```
public class test
{
    public int x {get;set;}
}
```

11

## getter, setter trong java

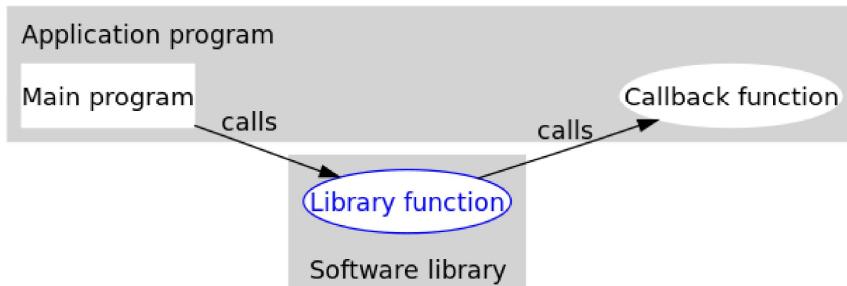
- ❖ Trong java không hỗ trợ khái niệm properties như C#. Thay vào đó java đưa ra khái niệm getter, setter
- ❖ Để truy cập an toàn vào dữ liệu của lớp, nên khai báo các biến dữ liệu của lớp là private và thực hiện gán giá trị cho các biến thông qua các setter, lấy giá trị của biến thông qua các getter

```
public class SV {
    String name;
    int tuoi;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getTuoi() {
        return tuoi;
    }
    public void setTuoi(int tuoi) {
        this.tuoi = tuoi;
    }
}
```

12

## Callback, Delegate, Event

- ❖ Callback: Là một đoạn mã, có thể là một phương thức, được xem như là một tham số đối với một đoạn mã khác muốn thực hiện (execute) tham số này tại một thời điểm thích hợp nào đó.



13

## Callback, Delegate, Event trong C#

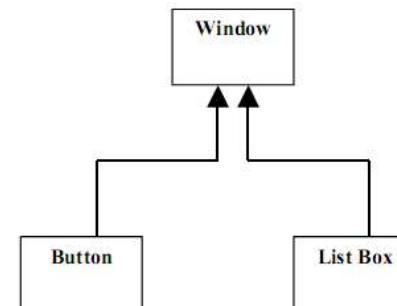
- ❖ Delegate: Từ khóa định nghĩa kiểu trong C# dùng để hỗ trợ cơ chế CALLBACK
  - Khai báo:  
access-modifier delegate return-type **delegate\_name(argument-list);**
- ❖ Event: Khai báo sự kiện để gọi phương thức ở thành phần Client
  - Cú pháp:  
access-modifier **event delegate\_name event\_name;**
- ❖ Sử dụng Event:
  - Library function: **event\_name(passed\_arguments);**
  - Client: **event\_name += callback\_function\_name;**

14

## Kế thừa, đa hình (inheritance, polymorphism)

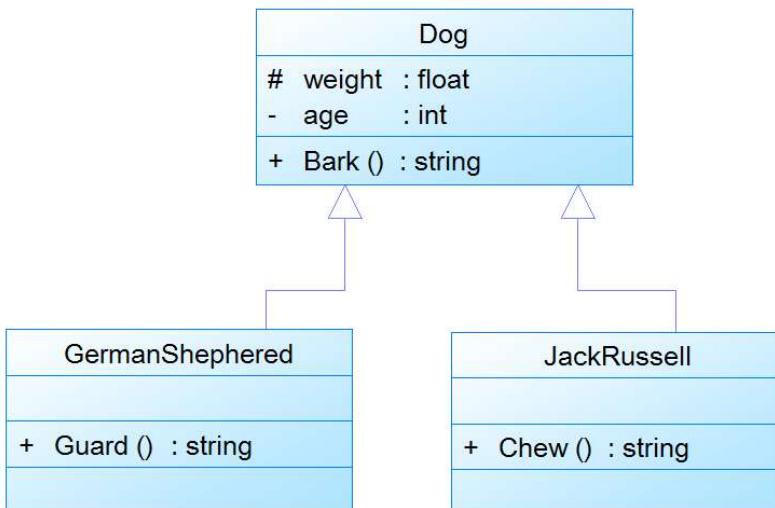
## inheritance

- ❖ cho phép một số phương thức (method), biến thành viên ở lớp cơ sở (base class) được sử dụng ở lớp dẫn xuất (derived class, subclass)



16

## inheritance



17

## Đa kế thừa, đơn kế thừa

- ❖ Đa kế thừa (Multi-inheritance): Một lớp được thừa kế từ nhiều lớp cơ sở
  - C++
- ❖ Đơn kế thừa (single inheritance): Một lớp chỉ được phép thừa kế từ một lớp cơ sở
  - C#, Java
- ❖ Cú pháp khai báo:
  - Java: <bố từ truy cập> tên lớp **extends** <tên lớp cơ sở>
  - C#: <bố từ truy cập> tên lớp: <tên lớp cơ sở>
  - C++ tên lớp: [kiểu thừa kế] <tên lớp cơ sở,>\*
    - public
    - protected
    - private

18

## inherited members' access modifier

base class	derived class
protected	private
public	public
internal	internal
protected internal	<b>protected internal</b> (nếu trong cả 2 lớp base và derived cùng khôi hợp ngữ) <b>private</b> (nếu derived class khác khôi hợp ngữ với base class)

19

## access-modifier trong java

### Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

20

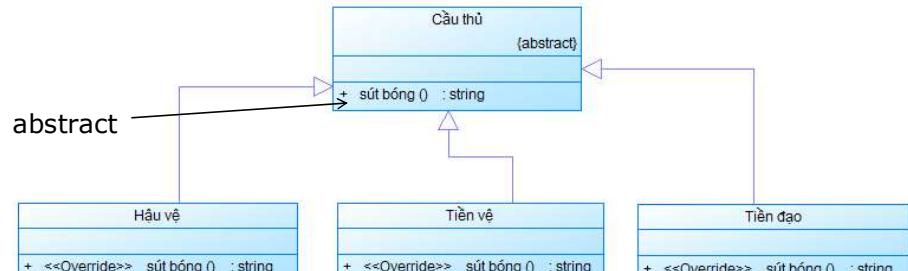
## Polymorphism

- ❖ Polymorphism=Một chức năng hay đối tượng được thực thi theo nhiều cách khác nhau
- ❖ Polymorphism có thể được áp dụng cho:
  - Objects
  - Operations (methods)

21

## polymorphic object

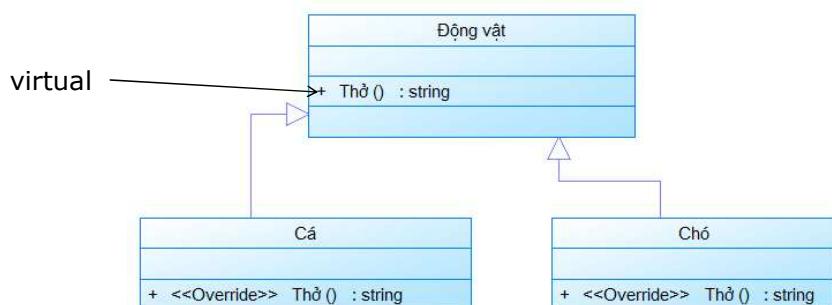
- ❖ polymorphic object: đối tượng của một lớp con là dẫn xuất của một super class
- ❖ Ví dụ:



22

## polymorphic method

- ❖ polymorphic operation: Một phương thức có thể thực hiện, trả về các kết quả khác nhau tùy thuộc vào lớp đối tượng thực hiện nó
- ❖ Ví dụ:



23

## Lớp trừu tượng (abstract class)

- ❖ Được khai báo với từ khóa abstract
  - ví dụ: abstract class A{...}
- ❖ Được thiết lập như là cơ sở cho các lớp dẫn xuất
  - Có một hay nhiều phương thức trừu tượng
  - Các phương thức trừu tượng chỉ có phần khai báo
  - Sự thực thi các phương thức trừu tượng trong lớp trừu tượng được giao cho lớp dẫn xuất
  - Không thể tạo một instance cho lớp trừu tượng
- ❖ Khi một lớp có một phương thức được khai báo là abstract phía trước chỉ dẫn truy cập thì lớp đó là lớp trừu tượng
- ❖ Lớp dẫn xuất từ lớp trừu tượng nhưng không thực thi phương thức trừu tượng cũng được xem là lớp trừu tượng.
  - Khi đó việc thực thi phương thức trừu tượng sẽ được giao cho các lớp con của lớp đó

24

## polymopic method

- ❖ Từ khóa virtual, override trong c#:
  - Phương thức với từ khóa virtual ở lớp base có thể bị ghi đè (overridden) ở lớp dẫn xuất
    - Ví dụ: public virtual void test() {...}
  - Từ khóa override: dù để ghi đè phương thức virtual trong lớp base
    - ví dụ: public override void test() {...}
  - Chú ý:
    - Nếu phương thức virtual bị ghi đè thì nó sẽ bị phương thức override thay thế ở lớp derived
    - Nếu không bị ghi đè thì phương thức virtual sẽ được sử dụng ở lớp derived
    - Phương thức virtual và override phải cùng tên

25

## Từ khóa new và override

- ❖ override: Phủ quyết một cách tường minh một phương thức virtual ở lớp base
- ❖ new: Chỉ ra rằng phương thức ở lớp derived không phủ quyết bất cứ phương thức nào ở lớp base
- ❖ Trong Java, Override là một chỉ dẫn dùng để chỉ ra rằng phương thức đang ghi đè một phương thức ở lớp cha
  - VD:
 

```
@Override
public void method1() {...}
```

26

## Lớp cô lập (sealed class)

- ❖ Được khai báo với từ khóa sealed
- ❖ Không cho phép các lớp dẫn xuất từ nó

27

## Lớp lồng nhau

- ❖ Lớp có thể chứa lớp bên trong nó
  - Lớp bên trong: nested/Inner class
  - Lớp ngoài: Outer class
- ❖ Nested class có thể truy cập đến tất cả các thành viên của lớp ngoài
  - Phương thức của lớp nested có thể truy cập đến biến thành viên private của lớp ngoài
  - Nested class có thể ẩn với các lớp khác bên ngoài lớp chứa nó khi được khai báo với từ khóa private

28

## Giao diện (interface)

- ❖ Chứa các khai báo của các phương thức qui định cho các lớp thực thi giao diện
  - Các phương thức đều được mặc định là public
  - Không khai báo bô từ truy cập cho các phương thức trong giao diện
  - Khi thực thi giao diện, các phương thức phải được khai báo public
  - Khi một lớp thực thi một giao diện, nó phải thực thi tất cả các phương thức trong giao diện đó
  - Nội dung thực hiện các phương thức do các lớp thực thi giao diện qui định → **polymorphism**

- ❖ Khai báo giao diện:

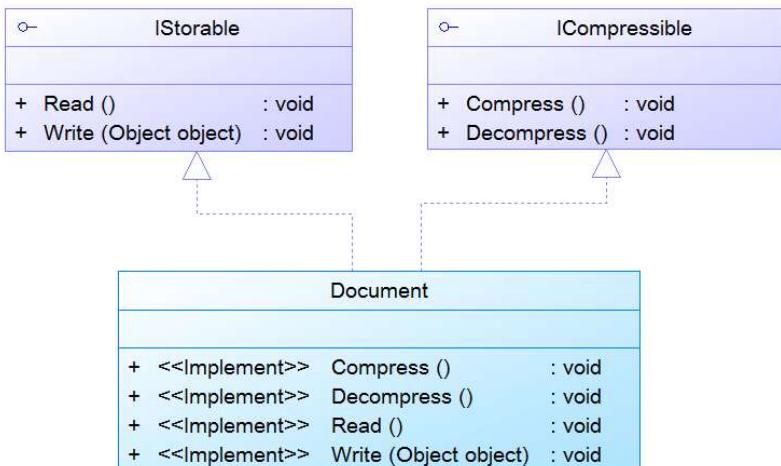
- C#
- java

29

## Sử dụng interface, abstract class

- ❖ Thực thi giao diện:

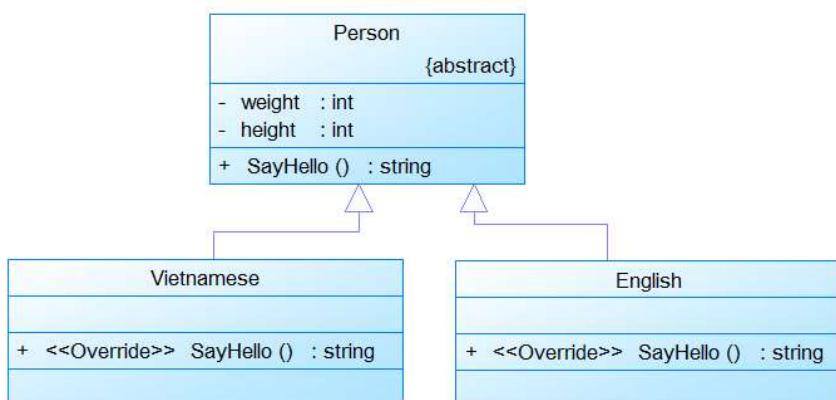
- Một lớp có thể thực thi một hoặc nhiều giao diện



30

## Sử dụng interface, abstract class

- ❖ C#/JAVA: Một lớp chỉ có thể thừa kế một lớp trừu tượng



Sử dụng:

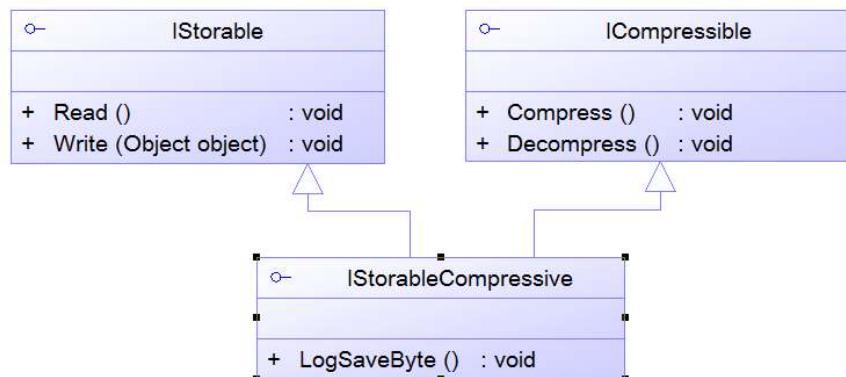
```

Person person=new English();
hoặc:
English en=new English();
Person person= en as Person;
  
```

31

## Mở rộng interface, abstract class

- ❖ Có thể thêm các thành viên mới cho một interface bằng cách thừa kế



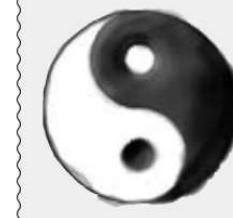
- ❖ Có thể mở rộng một abstract class bằng cách tạo lớp abstract dẫn xuất từ lớp đó

32

## Hạn chế của abstract class

- ❖ Abstract class không bắt buộc các lớp dẫn xuất ghi đè các phương thức abstract của nó.
- ❖ C#/JAVA: Một lớp dẫn xuất chỉ có thể thừa kế từ một base class
- ❖ Khắc phục những hạn chế của các abstract class bằng cách thay thế chúng bằng các interface

33

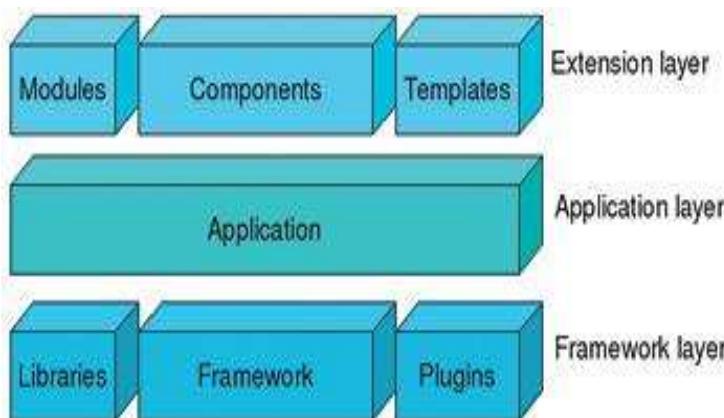


### **Design Principle**

*Program to an interface, not an implementation.*

34

## Lập trình module



35

## Lập trình module

- ❖ Kiến trúc mô-đun (module) cho phép chia nhỏ bài toán (hay yêu cầu) của phần mềm thành các phần hầu như không trùng lắp
  - Hỗ trợ làm việc song song trên các module
  - Dễ bảo trì
  - Khả năng tái sử dụng các thành phần của hệ thống
  - Khả năng mở rộng tốt hơn.
- ❖ Flex, Ruby: cho phép biên dịch các module một cách độc lập và có thể gắn kết vào hệ thống lúc thực thi.
- ❖ C#, C++: hỗ trợ cơ chế như thư viện liên kết động (DLL) để biên dịch các module thành các thư viện độc lập và có thể gắn kết động vào hệ thống.

36

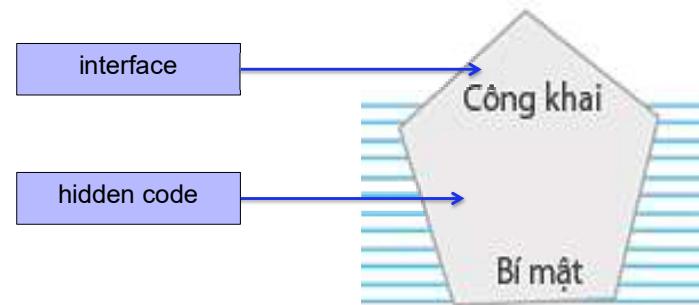
## interface: Công cụ giao tiếp của các modul

- ❖ Các module được gắn kết với nhau trong chương trình thông qua các "interface".
  - interface của module mô tả những thành phần được cung cấp và cần được cung cấp.
  - Các thành phần này được các module khác "thấy" và sử dụng.
- ❖ Interface của modul khác với interface của C# hay Java
  - Interface là những thành phần được các modul khác nhìn thấy
  - Để dễ dàng bảo trì nên dùng khái niệm Interface của NNLT

37

...

- ❖ interface của module nên được thiết kế chỉ bao gồm những phần hầu như không thay đổi,
  - Những thành phần này được gọi là thành phần "công khai".
  - Những chi tiết ẩn bên dưới các interface thường được gọi là các thành phần "bí mật" hoặc "riêng tư"



38

## Tài liệu tham khảo

- ❖ Jesse Liberty and Donald Xie. Programming C# 3.0. O'Reilly 2008
- ❖ Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First Design pattern. O'Reilly 2006.

39

## *Design patterns*

**Giảng viên: Huỳnh Tuấn Anh  
Khoa CNTT - Đại học Nha Trang**

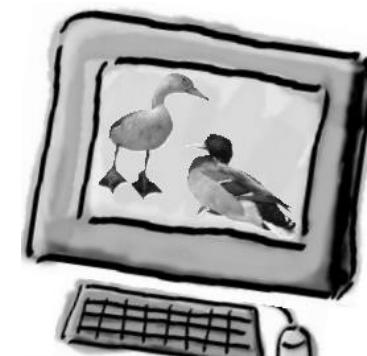
## *Tổng quan về design patterns*

- ❖ Tổng quan về design patterns
- ❖ Nhóm: Creational patterns
- ❖ Nhóm Structural patterns
- ❖ Nhóm Behavioral patterns

2

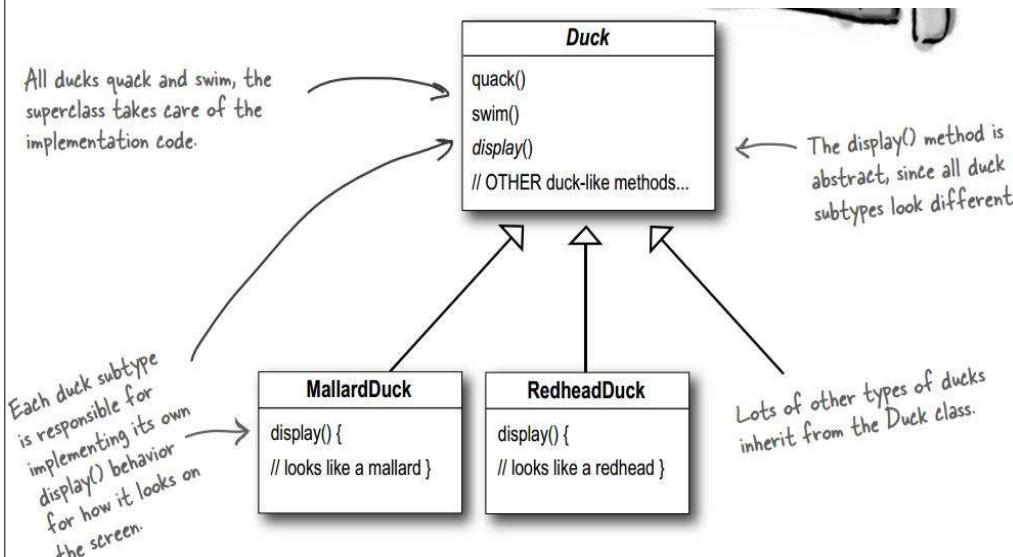
## Vấn đề mở đầu:

- ❖ Game: SimUDuck
  - Trò chơi có các nhân vật là các con vịt có khả năng bơi và kêu “quack quack”



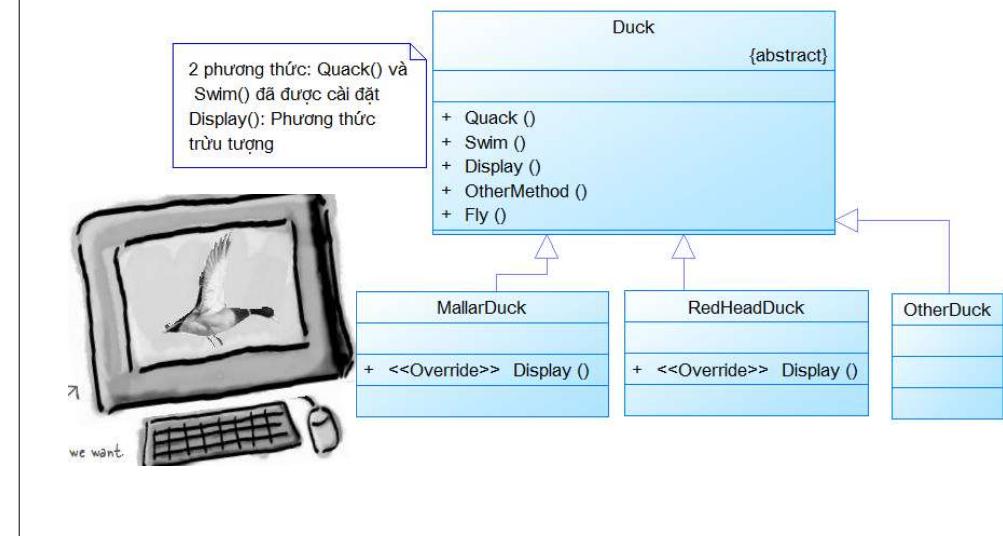
4

## Thiết kế trò chơi bằng phương pháp hướng đối tượng



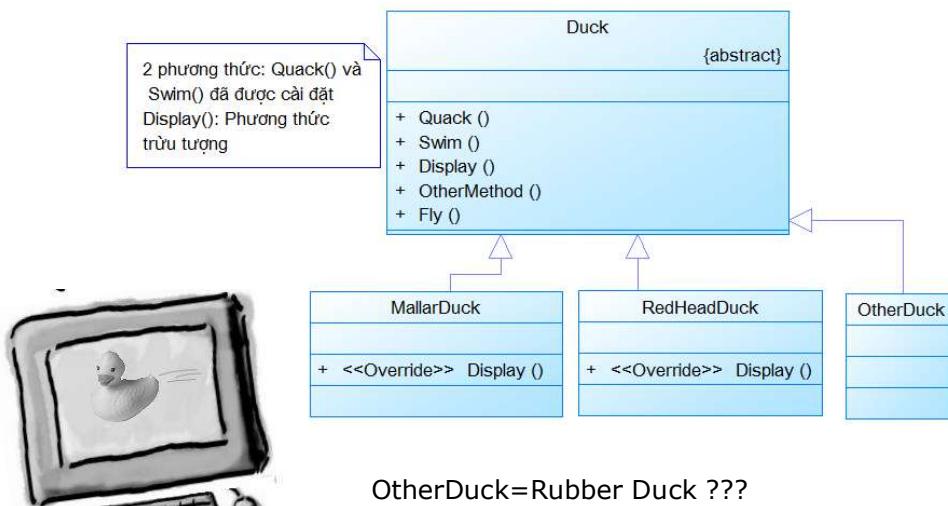
## Vấn đề mở đầu:

Vấn đề: Mở rộng trò chơi để cạnh tranh:  
Vịt có thể biết bay!!!



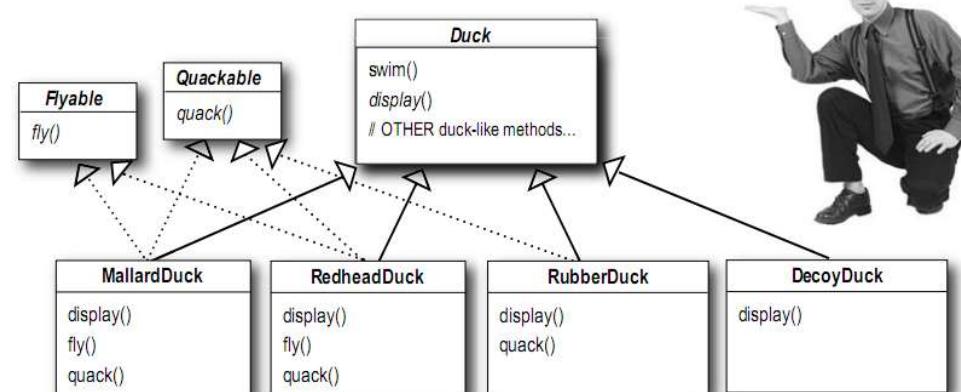
## Các ý tưởng giải quyết vấn đề

- 1<sup>st</sup> idea: Cài đặt thêm phương thức Fly() vào lớp Duck



## Các ý tưởng giải quyết vấn đề

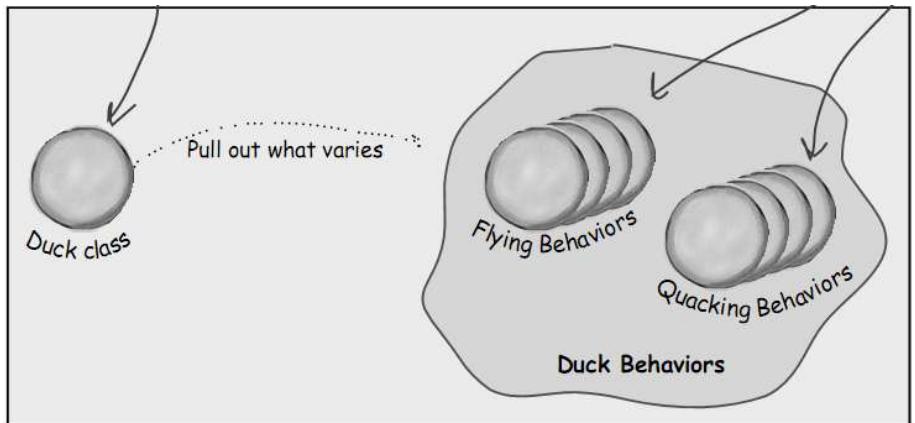
- 2<sup>nd</sup> idea: interface?



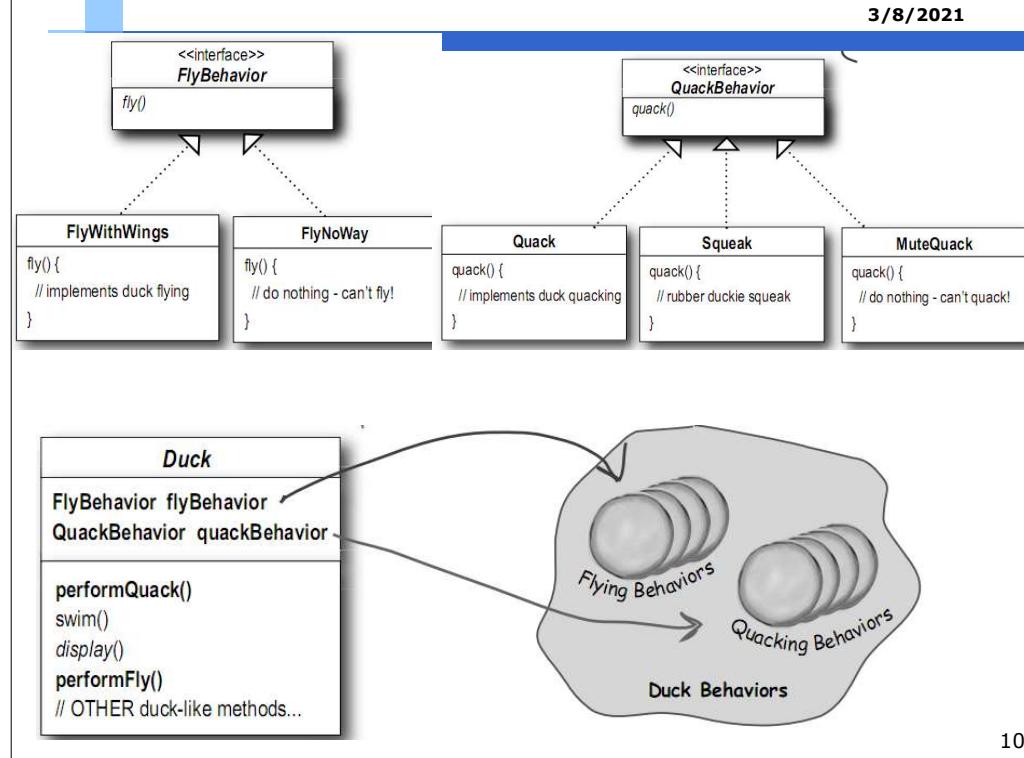
Nếu có nhiều loại vịt bay hoặc kêu giống nhau???

## Các ý tưởng giải quyết vấn đề

- 3<sup>rd</sup> idea: Tách rời Duck và các hành vi của chúng!



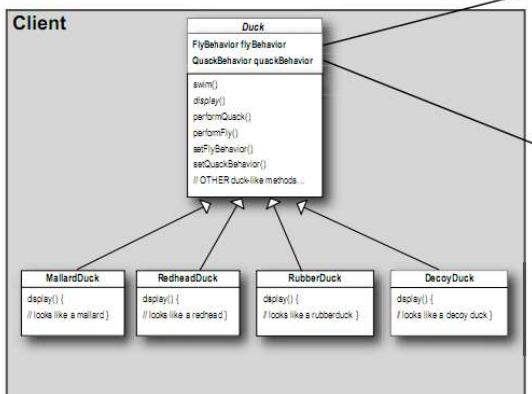
9



10

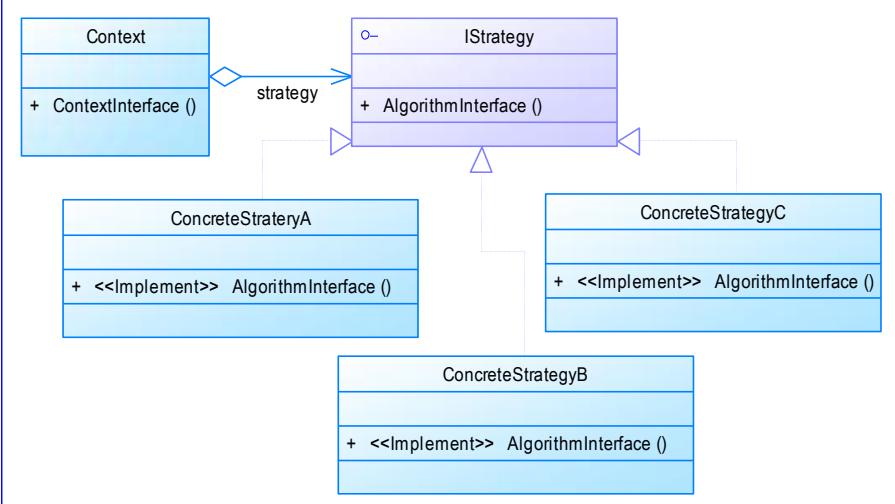
## Strategy pattern

Client makes use of an encapsulated family of algorithms for both flying and quacking.



11

## Strategy pattern



12

## Nguyên tắc đầu tiên trong thiết kế

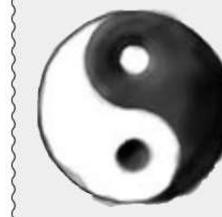


### **Design Principle**

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

13

- ❖ HAS-A can be better than IS-A



### **Design Principle**

*Favor composition over inheritance.*

14

## Khái niệm design patterns

- ❖ Bắt nguồn từ thuật ngữ ngành xây dựng của Christopher Alexander, dùng để :
  - Mô tả các vấn đề thường xuyên gặp phải, lặp đi lặp lại trong quá trình xây dựng các công trình.
- ❖ Đưa ra một giải pháp cơ bản để giải quyết các vấn đề đó.
  - Áp dụng giải pháp đó hàng ngàn lần trong xây dựng mà không lần nào giống lần nào !
- ❖ Định nghĩa của GoF : Gồm những mô tả cho việc giao tiếp giữa các đối tượng và lớp đối tượng mà sau đó được tùy biến, chỉnh sửa để giải quyết một vấn đề thiết kế chung chung trong một ngữ cảnh cụ thể.

15

## Khái niệm design patterns

- ❖ Định nghĩa của Ian Graham (OMG) :
  - Là một quá trình xây dựng các kiến trúc thiết kế nhỏ có thể dùng lại.
  - Các kiến trúc này cung cấp một giải pháp cơ bản, được dùng đi dùng lại trong các ngữ cảnh khác nhau nhưng đều tuân theo một “luật” nào đó.
  - Các Pattern là trùu tượng và không thể dùng ngay được (unpluggable), việc cài đặt, thực thi chúng sẽ thể hiện khác nhau với mỗi ứng dụng

16

## Khái niệm design patterns

❖ Dùng để xây dựng các phần mềm :

- Có thể sử dụng lại được.
- Tính linh động cao.
- Chuyên biệt hóa công việc các thành viên.
- Dễ kiểm thử, bảo trì, quản lý.

❖ Nhanh chóng tiếp cận và sử dụng lại hiệu quả các thành phần phần mềm được xây dựng với kỹ thuật Design Patterns :

- Framework : Struts, MVC, JSF, các Portal...
- Toolkit : JFC, BDK...
- Các thành phần điều khiển miễn phí

17

## Khái niệm design patterns

❖ Bản chất:

- Mục đích là chỉ ra giải pháp để tạo ra các thành phần dùng lại được và dễ dàng bảo trì, thậm chí là cho cả tài liệu của thành phần đó.
- Các phần mềm được tạo nhờ kỹ thuật Design Patterns phải được phân tách độc lập với nhau trên một số phương diện. Việc phân tách độc lập phần nào sẽ áp dụng Pattern phù hợp cho phần đó.
- Như vậy, việc sử dụng lại như thế nào chính là bản chất của kỹ thuật Design Patterns.

18

## Nguyên tắc thiết kế

❖ Design Principle: “Depend upon abstractions. Do not depend upon concrete classes.”

→ Program to an interface, not an implementation

19

## Phân loại các mẫu

- ❖ “Gang of Four” đã đưa ra 23 Pattern cơ bản. 23 Pattern này được phân loại theo mục đích và phạm vi áp dụng.
- ❖ Sau này các hãng phần mềm (Microsoft, Sun...) và các cá nhân đã “tạo ra” các Pattern mới, mỗi hãng lại có một “phong cách” khác nhau, ảnh hưởng tới ứng dụng của họ.
- ❖ Các Pattern mới có thể là sự kết hợp của một vài các Pattern cơ bản, các Pattern có được thừa nhận hay không là tùy thuộc và tính linh động, mở rộng và khả chuyển của nó.

20

## Phân loại các mẫu

### Theo mục đích :

- Creational : Áp dụng trong quá trình khởi tạo, độc lập việc khai báo, khởi tạo một đối tượng mới.
- Structural : Áp dụng trong việc tạo cấu trúc giữa các lớp hay đối tượng, mối liên quan giữa chúng.
- Behavioral : Áp dụng trong việc độc lập các hành vi đối tượng, các thuật toán thực hiện hành vi đó.

### Theo phạm vi áp dụng :

- Cho phạm vi đối tượng (objects).
- Cho phạm vi lớp đối tượng (classes).

21

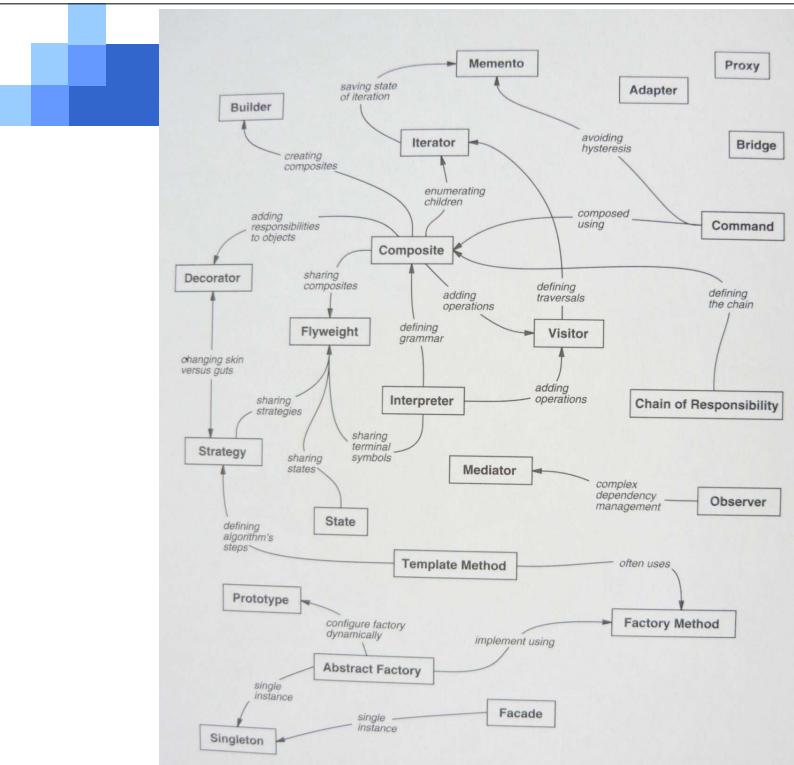
		Mục đích		
		Creational	Structural	Behavioral
Phạm vi áp dụng	Class	-Factory method	-Adapter	-Interpreter -Template method
	Object	-Abstract Factory -Builder -Prototype -Singleton	-Adapter -Bridge -Composite -Decorator -Facade -Proxy	-Chain of Responsibility -Command -Iterator -Mediator -Memento -Flyweight -Observer -State -Strategy -Visitor

22

## Phân loại mẫu

- ❖ Khó có thể tìm được ứng dụng chỉ áp dụng một Pattern và cũng không thể tìm được một ứng dụng dùng cả 23 Pattern.
- ❖ Một vài Pattern thường đi kèm với nhau như Prototype và Singleton.
- ❖ Một vài Pattern khác là thay thế lẫn nhau, chúng không thể dùng đồng thời như Prototype và Abstract Factory.
- ❖ Mục đích việc phân loại các Pattern:
  - Làm cho việc tìm hiểu và sử dụng từng Pattern nhanh hơn, đúng hơn.
  - Giúp tăng tốc việc tìm kiếm Pattern thích hợp ( phù hợp với vấn đề cụ thể của ứng dụng đang phát triển)

23



24



25

## Tài liệu tham khảo

- ❖ Eric Freeman, Elisabeth Freeman. Head First Design pattern. O'Reilly 2006.
- ❖ Bài giảng Design pattern của PGS.TS Huỳnh Quyết Thắng, Đại học BKHN
- ❖ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995

26

# ***Creational patterns***

**Giảng viên: Huỳnh Tuấn Anh  
Khoa CNTT - Đại học Nha Trang**

## **Creational patterns**

- ❖ Creational patterns
  - Thiết kế các khuôn mẫu tạo lập.
  - Thiết kế các khuôn mẫu tạo lập → Trừu tượng hóa tiến trình khởi tạo.
  - Giúp tạo ra một hệ thống các đối tượng độc lập với việc thao tác trên các đối tượng đó.
- ❖ Lớp & Đối tượng:
  - Lớp → Thừa kế
  - Đối tượng → Đại diện (interface, abstract classs)
- ❖ Các khuôn mẫu khởi tạo sẽ có trách nhiệm khởi tạo các đối tượng cụ thể thích hợp và gán cho đại diện của đối tượng

2

## **Creational patterns**

- ❖ Sử dụng những lớp căn bản để xây dựng những lớp lớn hơn → Thao tác khởi tạo không còn đơn giản là khởi tạo cho 1 đối tượng nữa.
- ❖ Với 1 hàm khởi tạo cho 1 đối tượng cụ thể, khi cần tái sử dụng:
  - Viết lại một số thao tác → Phức tạp
  - Cách khác → Sử dụng Patterns.

## **Creational patterns**

- ❖ Factory method
- ❖ Abstract Factory
- ❖ Builder Pattern
- ❖ Prototype pattern
- ❖ Singleton Pattern

3

4

## Factory method

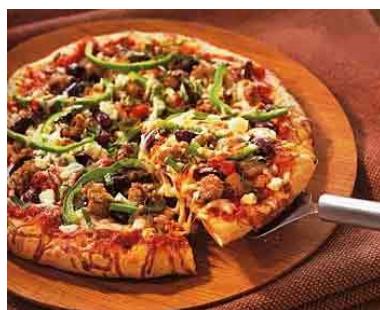
### Factory method

- ❖ Toán tử `new`: Dùng để tạo thể hiện của một đối tượng
  - Duck duck = new MallardDuck();
- ❖ Trường hợp có thêm một ConcreteClass được định nghĩa:
  - Việc sử dụng giao diện cho phép định nghĩa thêm lớp mới thực thi nó rất dễ dàng → “*Open for extension*”
  - Sử dụng toán tử `new` ở phần client để tạo các đối tượng đòi hỏi phải viết lại mã lệnh → mã lệnh không có tính chất “*close for modification*”.
- ❖ *Làm thế nào để tách rời phần khởi tạo thể hiện đối tượng của chương trình với các phần còn lại của chương trình? → Factory method*

6

### Pizza Example

```
Pizza orderPizza()
{
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```



Nhiều loại Pizza?

### Pizza Example

```
Pizza orderPizza(String type)
{
    Pizza pizza;
    if (type.equals("cheese"))
    {
        pizza = new CheesePizza();
    }
    else if (type.equals("greek"))
    {
        pizza = new GreekPizza();
    }
    else if (type.equals("pepperoni"))
    {
        pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Phần mã lệnh thường xuyên thay đổi

Phần thay đổi và phần cố định của mã lệnh nằm chung → khó khăn trong việc nâng cấp, không có tính reusable

7

8

## Pizza Example



```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese"))
            pizza = new CheesePizza();
        else if (type.equals("pepperoni"))
            pizza = new PepperoniPizza();
        else if (type.equals("clam"))
            pizza = new ClamPizza();
        else if (type.equals("veggie"))
            pizza = new VeggiePizza();
        return pizza;
    }
}
```

```
public class PizzaStore {
    SimplePizzaFactory factory;

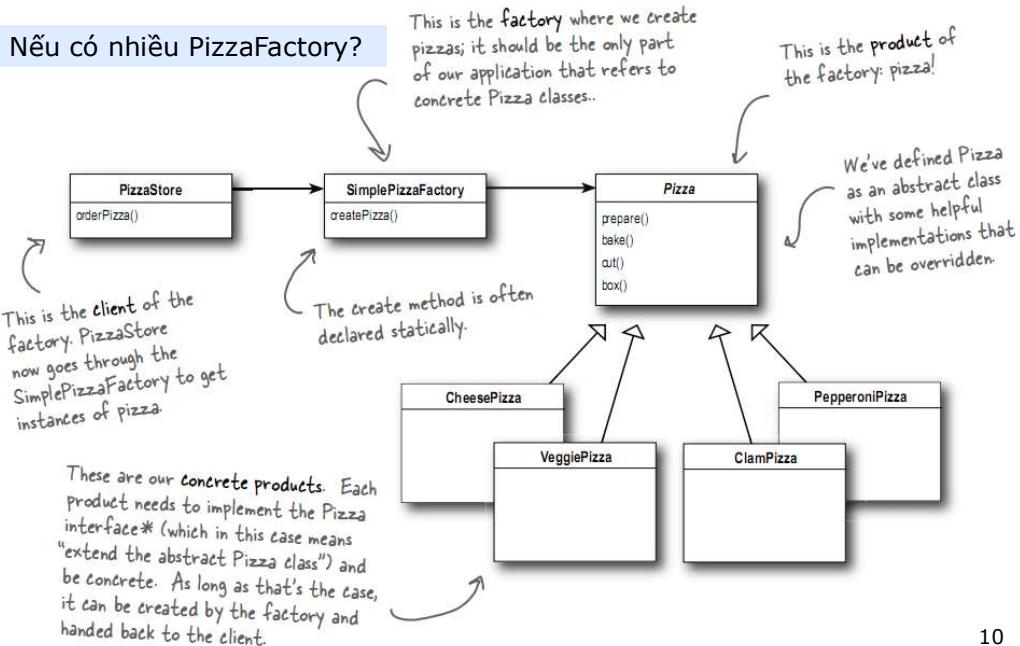
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // Các phương thức khác
}
```

9

## Pizza Example: simple factory

Nếu có nhiều PizzaFactory?



10

## pizza example: simple factory

❖ 1<sup>st</sup> idea: Xây dựng nhiều cặp lớp tương tự như SimplePizzaFactory và PizzaStore

- Mã lệnh ở client:

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("cheese");
```

- Mã lệnh không mèm dẻo: Do có sự liên hệ chặt chẽ giữa các PizzaStore và PizzaFactory

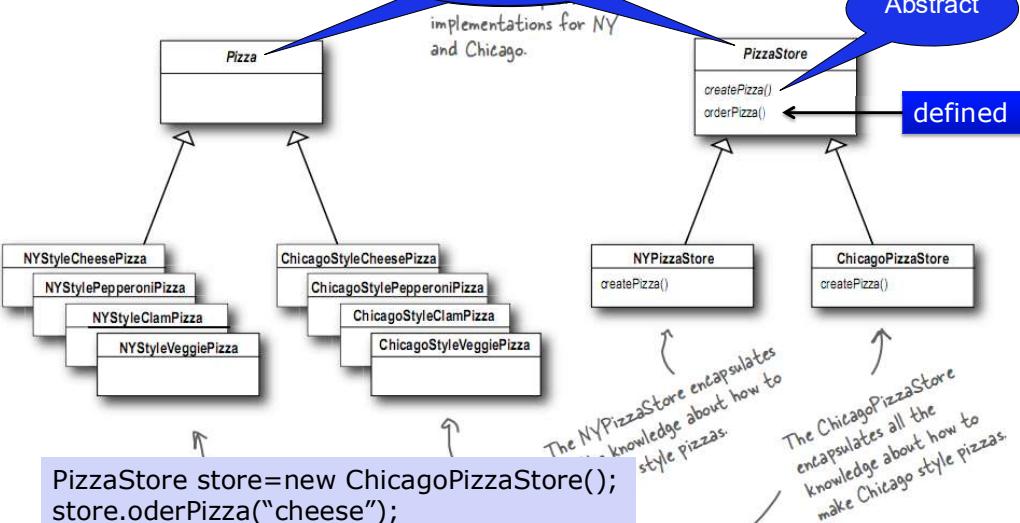
❖ 2<sup>nd</sup> idea: Gộp 2 lớp PizzaFactory và PizzaStore

- Xây dựng lớp trùu tượng PizzaStore
- Các subclass thực thi interface PizzaStore
- Các subclass quyết định loại pizza nào được tạo ra

11

## Pizza Example: Factory method diagram

The Product classes

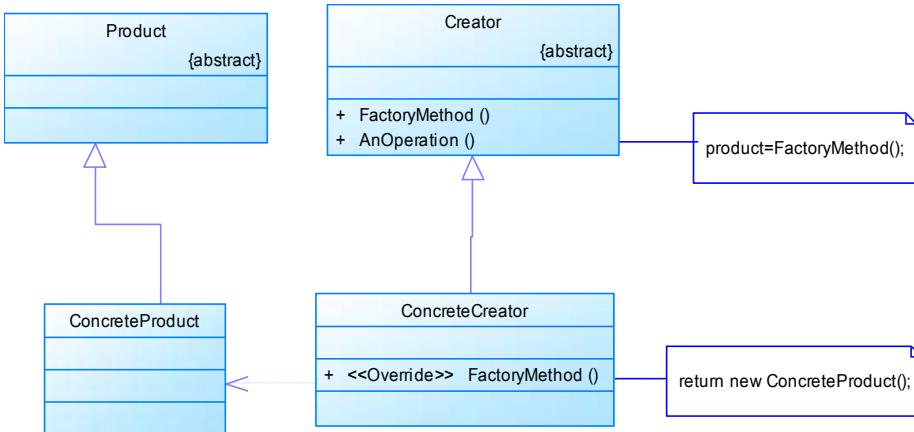


12

## Factory method

❖ **Mục đích:** Định nghĩa một giao diện để tạo đối tượng, nhưng để cho các lớp con quyết định lớp nào sẽ được khởi tạo.

❖ **Cấu trúc:**



13

## Questions

- ❖ Factory method có ưu điểm gì khi chỉ có một ConcreteCreator?
- ❖ Trong trường hợp nào ta phải tách riêng quá trình khởi tạo các đối tượng ra khỏi việc xử lý các đối tượng đó?
- ❖ Nêu ưu và khuyết điểm khi cài đặt phương thức FactoryMethod là static.
- ❖ FactoryMethod và Creator chỉ có thể luôn luôn là Abstract?
- ❖ Có thể thay các lớp Creator và Product là các interface được không?

14



### Design Principle

*Depend upon abstractions. Do not depend upon concrete classes.*

## Abstract factory

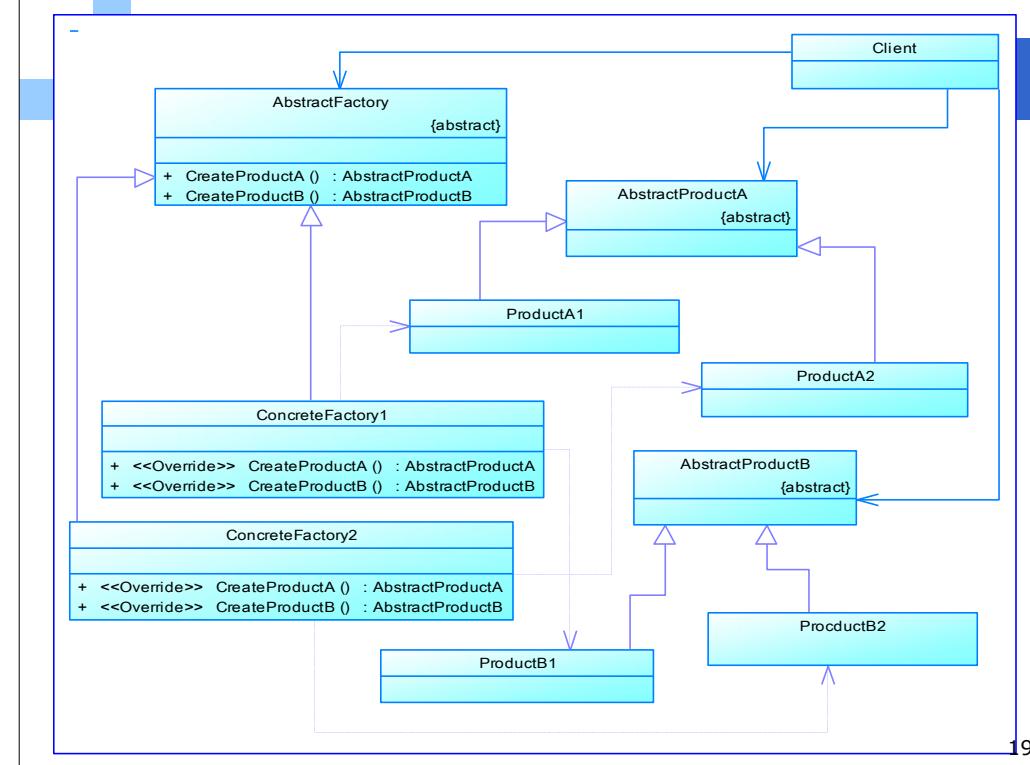
16

## Abstract factory

### Mục đích:

- Cung cấp một giao diện (interface) cho việc tạo một tập đối tượng có liên quan hay phụ thuộc nhau mà không cần chỉ định các lớp cụ thể của chúng

18



19

## Pizza Example

The abstract **PizzaIngredientFactory** is the interface that defines how to make a family of related products  
- everything we need to make a pizza.

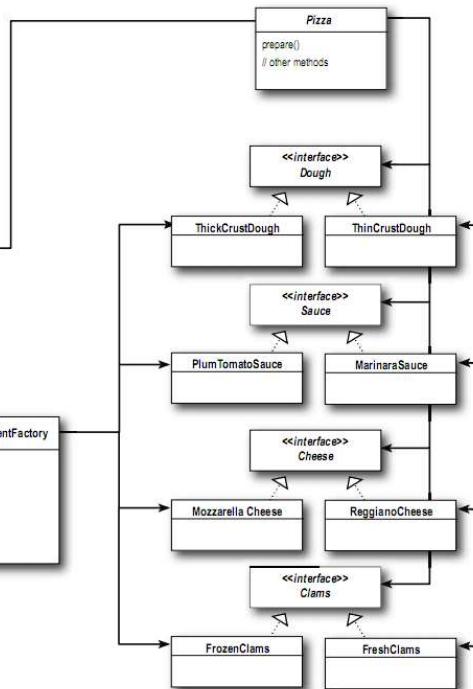
**NYPizzaIngredientFactory**

```

<<interface>>
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
  
```



The job of the concrete pizza factories is to make pizza ingredients.  
Each factory knows how to create the right



20

## Abstract factory vs Factory method

### Abstract Factory

Tạo đối tượng mà không chỉ ra lớp cụ thể của nó

Tách rời mã lệnh giữa client và các concrete class cần để tạo đối tượng cho các lớp đó

Tạo một tập các đối tượng

Sử dụng khi client tạo một tập các đối tượng cùng với nhau

Các đối tượng được tạo từ một đối tượng concreteFactory

interface AbstractFactory thay đổi nếu thêm một loại product mới

### Factory method

tạo một đối tượng

Sử dụng khi client chỉ tạo một đối tượng

Đối tượng được tạo từ lớp thực thi của lớp Creator

interface Creator không thay đổi khi thêm product mới

- phương thức **CreateProduct()** thường được thực thi như một Factory method
- Sử dụng Abstract Factory khi tạo một tập các loại product ít thay đổi

21

## Singleton pattern

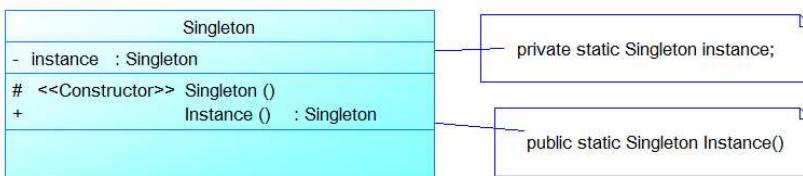
### Singleton pattern

#### ❖ Mục đích:

- Đảm bảo chỉ có một thể hiện của một lớp được tạo, và cho phép sự truy cập toàn cục đến đối tượng đó.
- Việc tạo đối tượng của một lớp phải do chính lớp đó đảm nhận

23

#### ❖ Cấu trúc



24

### Singleton pattern

#### ❖ Mã lệnh ()

```
class Singleton
{
    // Fields
    private static Singleton instance;

    // Constructor
    protected Singleton() {}

    // Methods
    public static Singleton Instance()
    {
        // Uses "Lazy initialization"
        if( instance == null )
            instance = new Singleton();

        return instance;
    }
}
```

Sử dụng: Singleton s = Singleton.Instance();

25

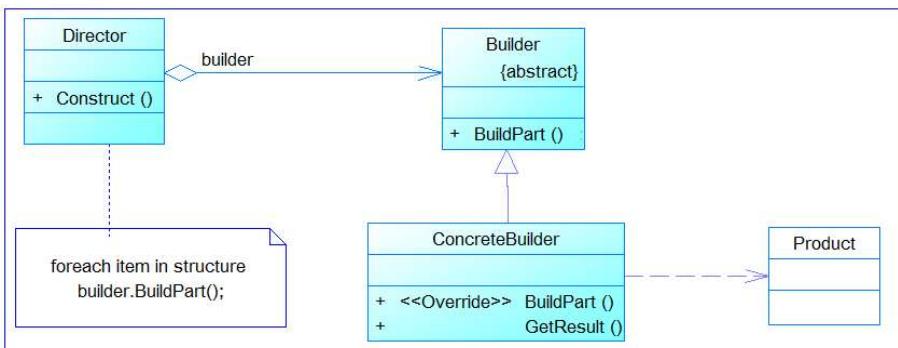
## Questions

- ❖ Tại sao phương thức khởi tạo của lớp Singleton lại được khai báo *protected* ?
- ❖ Tại sao biến instance được khai báo *static* ?
- ❖ Phương thức Instance() có thể không khai báo *static* được không?
- ❖ Nêu một số ví dụ về các trường hợp sử dụng Singleton pattern

26

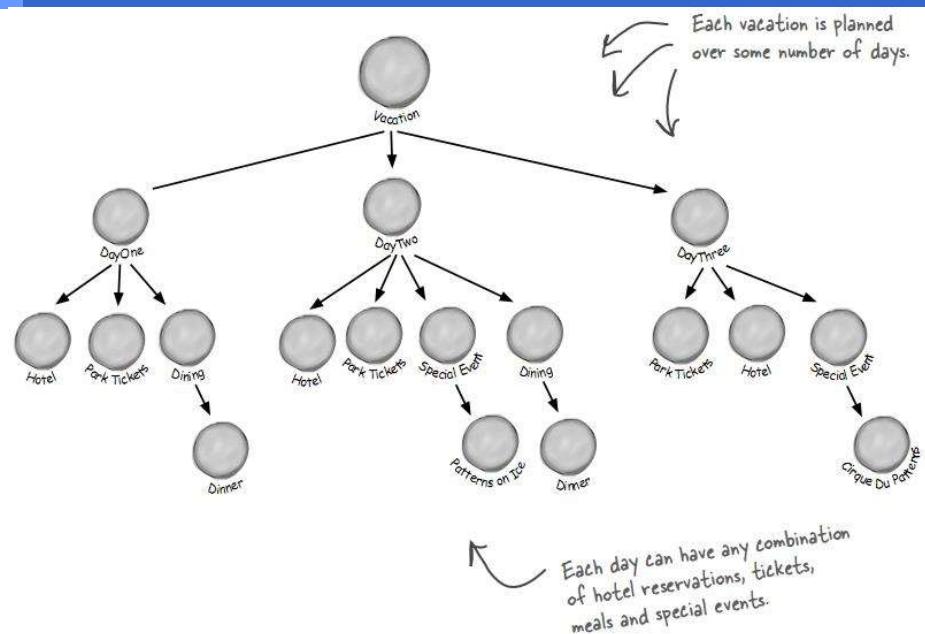
## Builder pattern

- ❖ **Mục đích:** Đóng gói việc xây dựng một product và cho phép nó được xây dựng qua nhiều bước

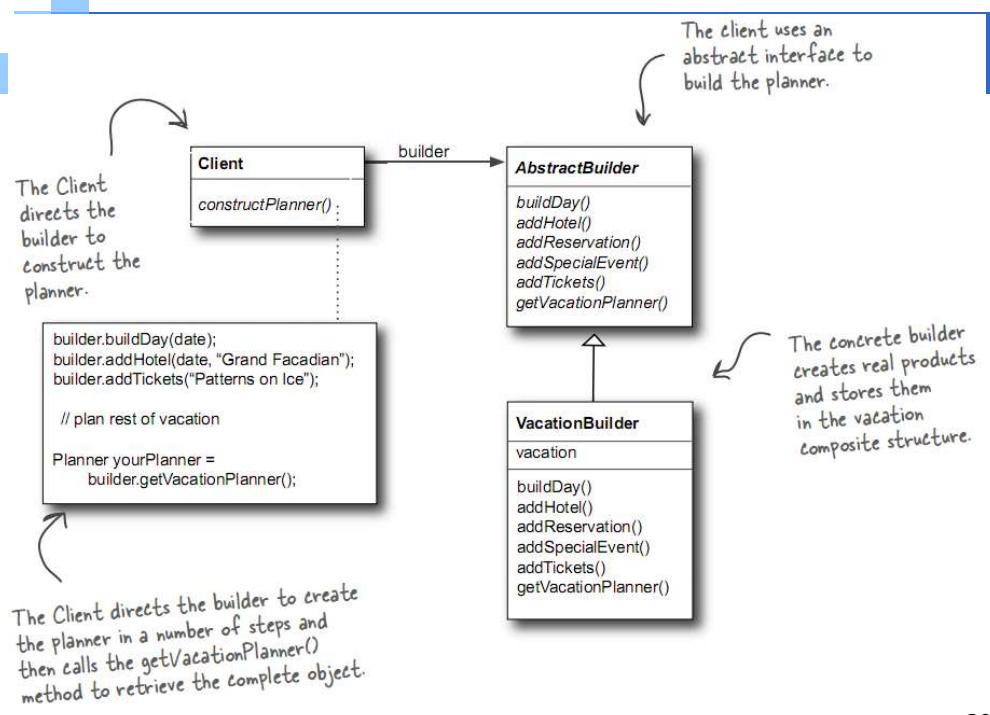


28

## Example: Vacation planner



29



30

## Đánh giá Builder pattern

### Ưu điểm

- Đóng gói cách xây dựng một đối tượng phức tạp
- Cho phép một đối tượng được xây dựng qua nhiều bước và tiến trình khác nhau
- Giấu biểu diễn bên trong của product đối với client
- Sự thực thi một product có thể được hoán chuyển in, out bởi vì client chỉ thấy một giao diện trừu tượng

### Khuyết điểm

- Việc xây dựng các đối tượng đòi hỏi nhiều tri thức hơn khi dùng Builder pattern so với dùng Factory

### Sử dụng

- Được sử dụng cho việc xây dựng các cấu trúc phức tạp bao gồm nhiều thành phần

31

## Questions

- So sánh 2 pattern Abstract factory và Builder
- Có thể sử dụng Builder pattern để thay thế cho Abstract factory được không?

32

## Prototype pattern

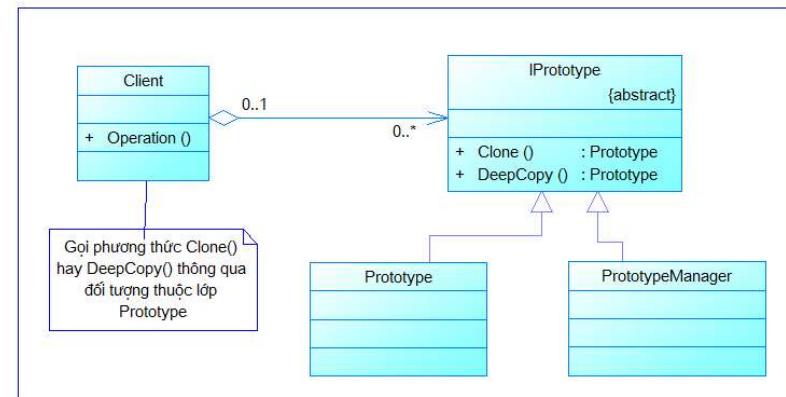
## Prototype pattern

### Mục đích:

- Tạo một đối tượng của một lớp phức tạp hay một lớp mà việc khởi tạo tốn kém nhiều chi phí
- Cho phép tạo một đối tượng mới bằng cách sao chép một đối tượng sẵn có

34

## prototype pattern



35

## Prototype pattern

### Lớp trùu tượng IPrototype

- Phương thức Clone(): Chỉ tạo một tham chiếu tới một đối tượng
  - Không lưu giữ được các giá trị của một đối tượng khi đối tượng đó bị thay đổi.
- Phương thức DeepCopy(): Tạo một bản sao thật sự của đối tượng
  - Lưu giữ được các giá trị của đối tượng
- Các phương thức Clone(), DeepCopy() nên được cài đặt trong lớp trùu tượng IPrototype

36

## Prototype pattern

### Ưu điểm

- Che giấu sự phức tạp của việc tạo một đối tượng mới đối với Client
- Cung cấp sự chọn lựa cho Client để tạo các đối tượng mà không cần biết dạng của chúng.
- Trong nhiều trường hợp việc copy một đối tượng hiệu quả hơn là việc khởi tạo nó

### Khuyết điểm

- Đôi khi việc tạo bản copy của một đối tượng có thể rất phức tạp

### Sử dụng:

- Prototype pattern được sử dụng khi việc tạo thể hiện của một lớp là phức tạp và tốn kém nhiều chi phí

37

## Questions

- ❖ Có thể dùng interface để thay thế cho lớp Abstract IPrototype hay không?

38

## Tài liệu tham khảo

- ❖ Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First Design pattern. O'Reilly 2006.
- ❖ **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
- ❖ <http://www.dofactory.com/Patterns/Patterns.aspx>

39

# *Structural pattern*

**Giảng viên: Huỳnh Tuấn Anh  
Khoa CNTT - Đại học Nha Trang**

## Structural pattern

- ❖ Liên quan tới cách tổ chức các đối tượng để hình thành các cấu trúc lớn hơn
- ❖ Mô tả cách để kết hợp các đối tượng để thực hiện một chức năng mới

- ❖ Adapter pattern
- ❖ Bridge pattern
- ❖ Composite pattern
- ❖ Decorator pattern
- ❖ Façade pattern
- ❖ Proxy pattern

# *Decorator pattern*

## Example

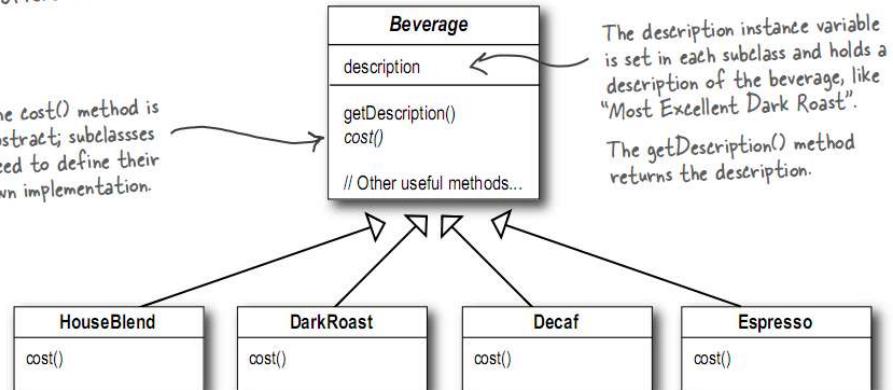


5

## Example: Cost of Beverage

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The `cost()` method is abstract; subclasses need to define their own implementation.



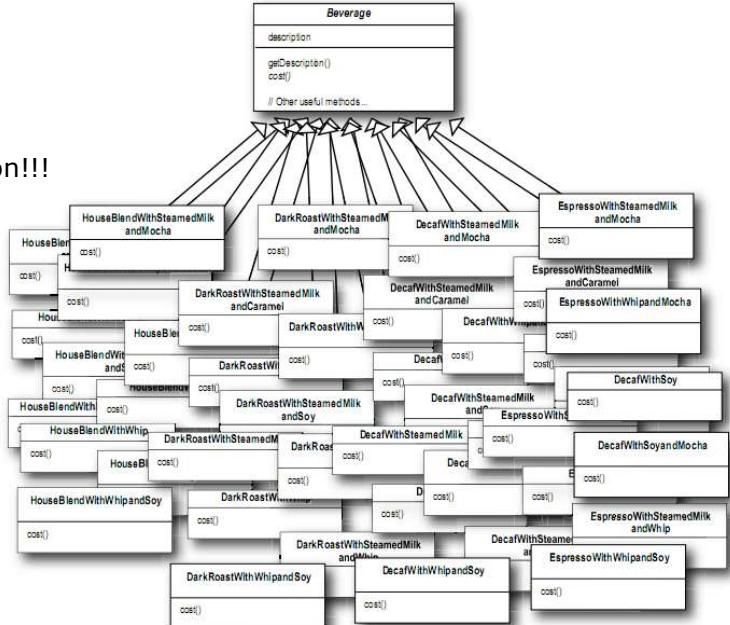
Thêm một số thành phần phụ vào 4 loại nước đã có ?

Each subclass implements `cost()` to return the cost of the beverage.

6

### 1<sup>st</sup> idea

Class explosion!!!



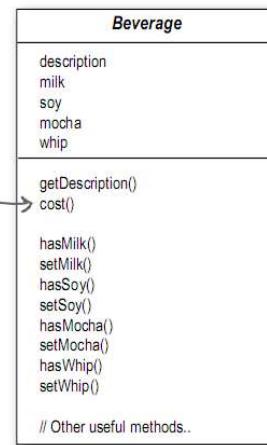
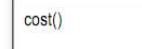
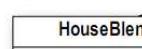
7

### 2<sup>nd</sup> idea

Now let's add in the subclasses, one for each beverage on the menu:

The superclass `cost()` will calculate the costs for all of the condiments, while the overridden `cost()` in the subclasses will extend that functionality to include costs for that specific beverage type.

Each `cost()` method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of `cost()`.



8

❖ The Open-Closed Principle:

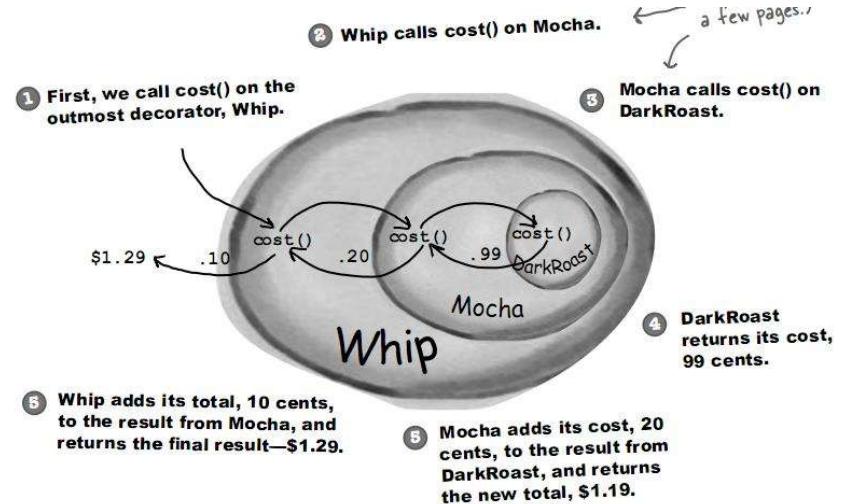
## Design Principle

*Classes should be open for extension, but closed for modification.*



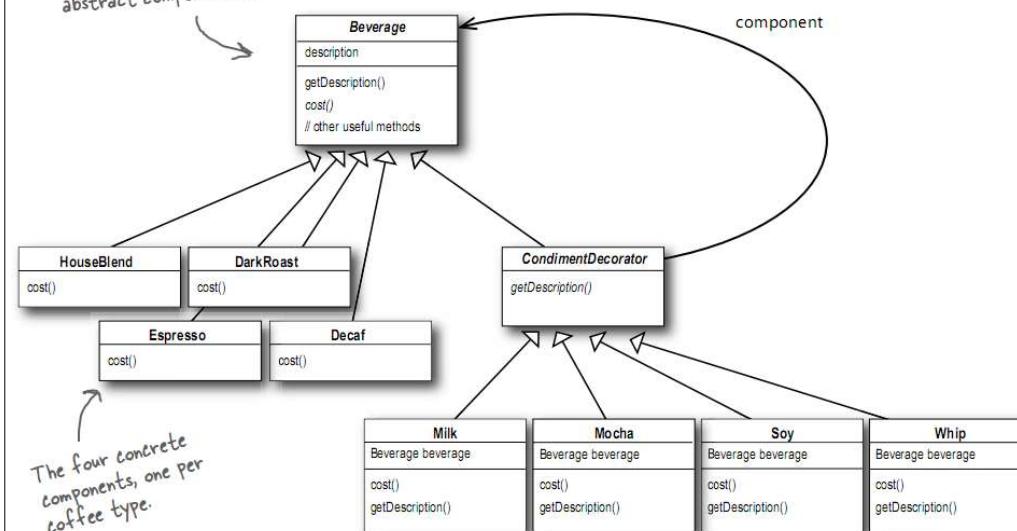
9

❖ 3<sup>rd</sup> idea



10

Beverage acts as our abstract component class.



11

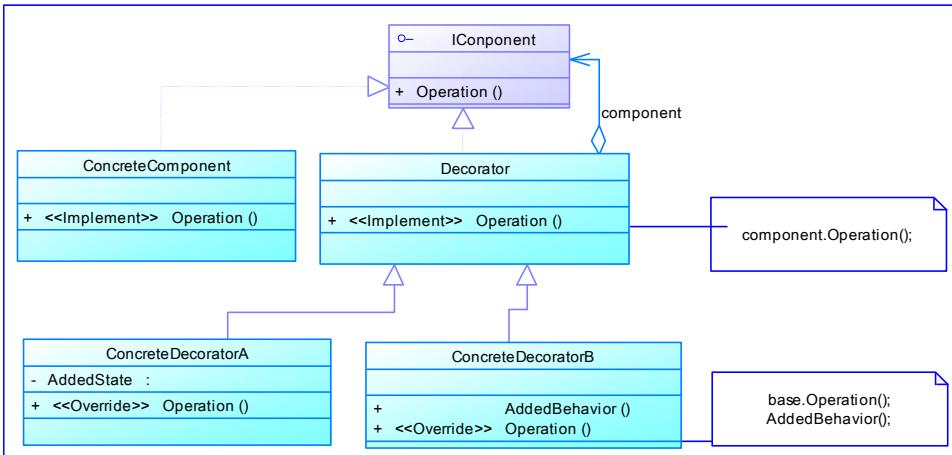
## Decorator pattern

❖ Mục đích:

- Cho phép thêm mới các trạng thái và hành vi vào một đối tượng lúc run-time bằng cách dùng kỹ thuật subclassing để mở rộng các chức năng của lớp.

12

## ❖ Cấu trúc



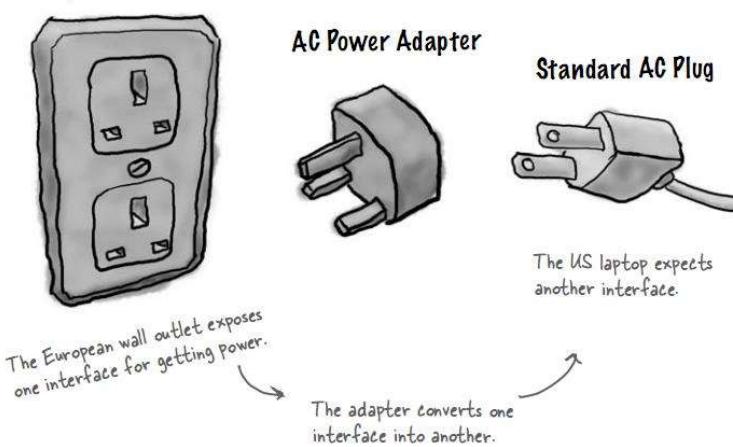
13

# Questions

- ❖ Vai trò của lớp Decorator, có thể không cần dùng lớp Decorator được không?
  - ❖ Nếu mối liên hệ giữa ConcreteComponent và ConcreteDecorator

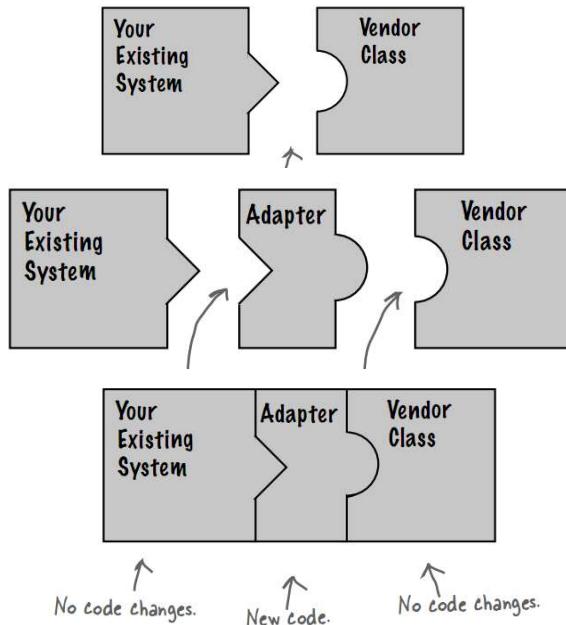
## *Adapter pattern*

## Adapter pattern: Example



16

## Adapter pattern: Example



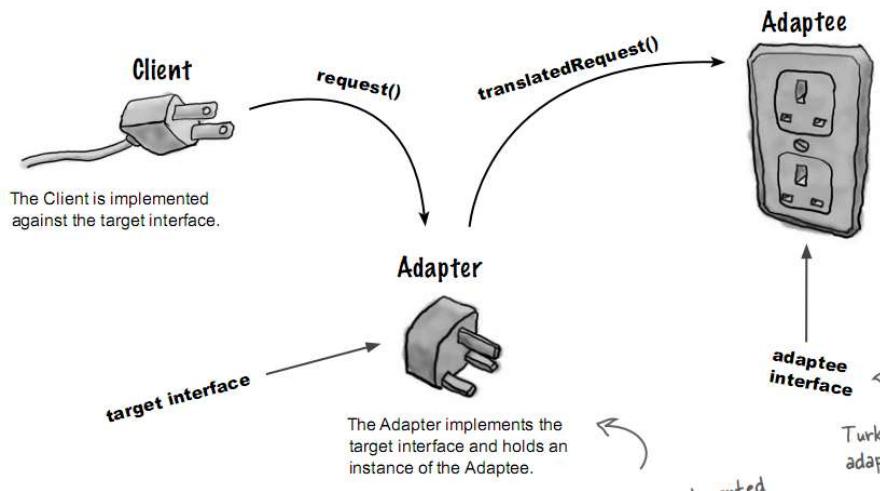
17

## Adapter pattern

### Mục đích:

- Chuyển giao diện của một lớp thành một giao diện khác mà client sử dụng
- Cho phép các lớp có giao diện không tương thích cùng làm việc với nhau.

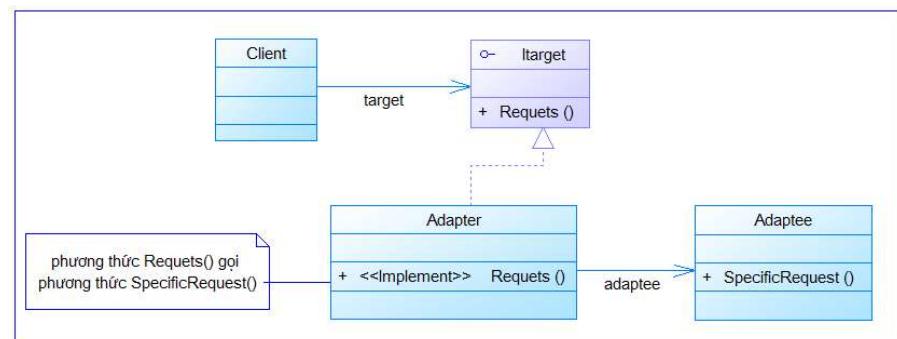
## Adapter pattern: Idea



19

## Adapter pattern

### Cấu trúc:



20

## Questions

- ❖ Vai trò của lớp Adapter ?
- ❖ Adapter chỉ được sử dụng cho một lớp Adaptee duy nhất?

21

## Façade pattern

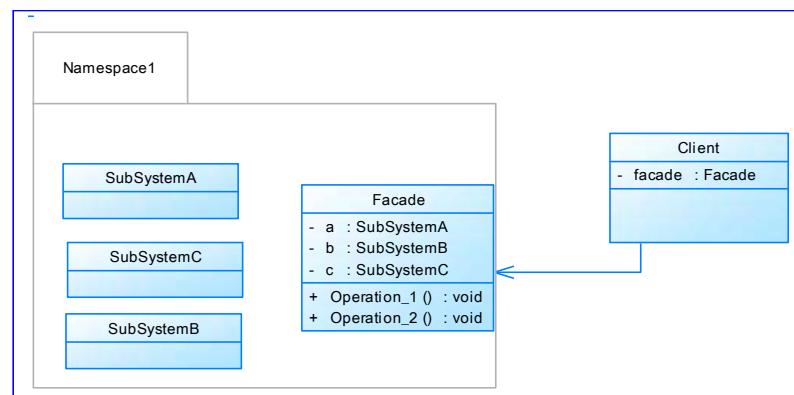
### Façade pattern

- ❖ Mục đích
  - Cung cấp một interface hợp nhất cho một tập các interface trong một subsystem
  - Định nghĩa một interface ở mức cao làm cho việc sử dụng subsystem trở nên dễ dàng hơn

23

### Façade pattern

- ❖ Cấu trúc:



24

- ❖ Each unit should only talk to its friends; don't talk to strangers.
- ❖ Reduce the interactions between objects to just a few close “friend”



### **Design Principle**

*Principle of Least Knowledge -  
talk only to your immediate friends.*

25

## **Proxy pattern**

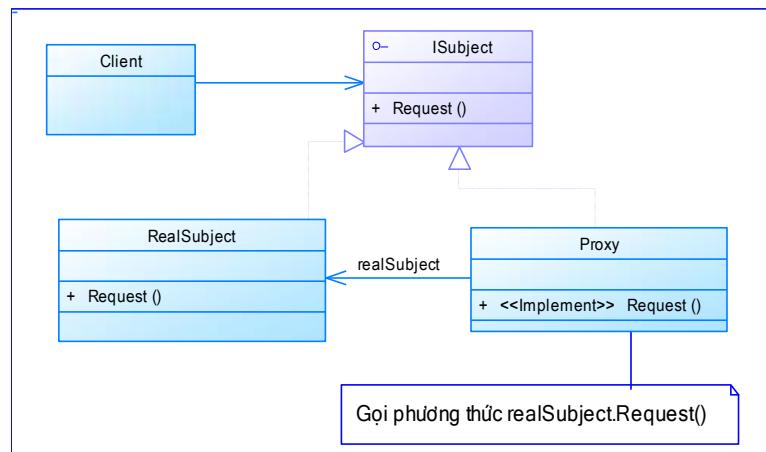
### **Proxy pattern**

- ❖ Mục đích:
  - Cung cấp một đối tượng thay thế hay một trình giữ chỗ cho một đối tượng khác để kiểm soát việc truy cập tới đối tượng đó.
  - Sử dụng Proxy pattern để tạo một đối tượng đại diện để kiểm soát truy cập tới một đối tượng khác:
    - Ở xa
    - Expensive to create
    - Cần được bảo vệ

27

### **Proxy pattern**

- ❖ Cấu trúc

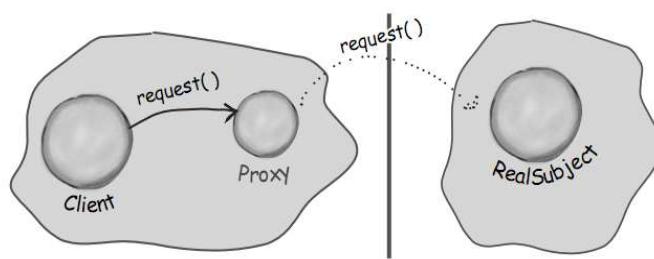


28

## Remote proxy

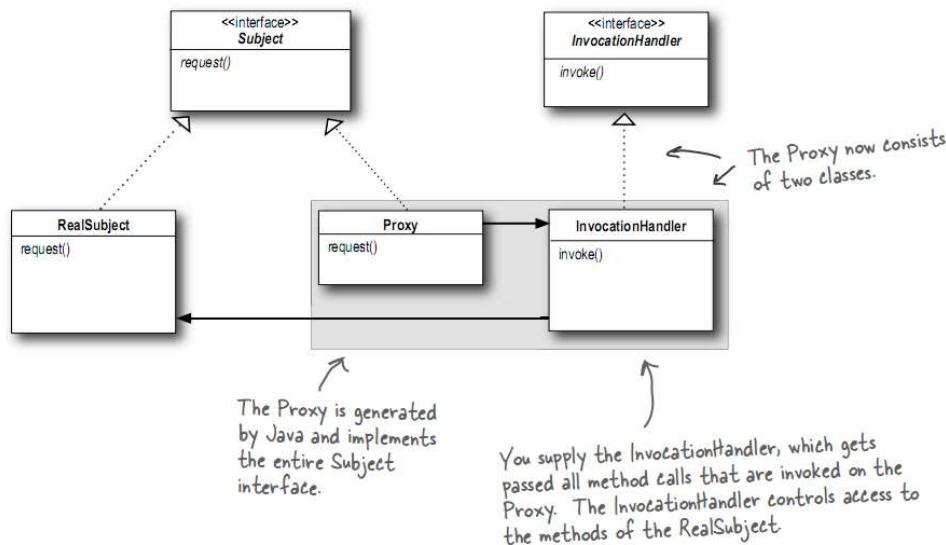
### Remote Proxy

With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.



29

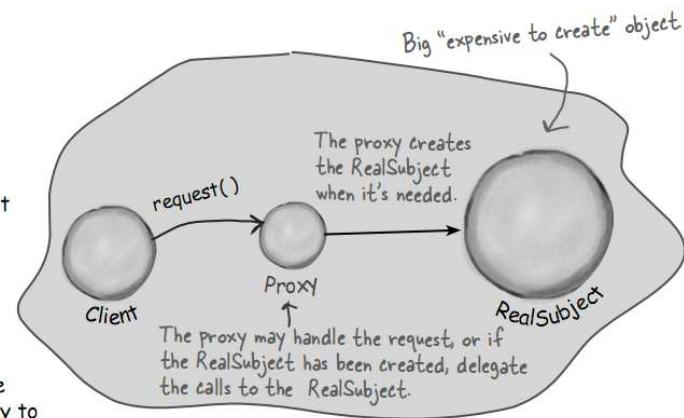
## Protection proxy



31

### Virtual Proxy

Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.



30

## Questions

32

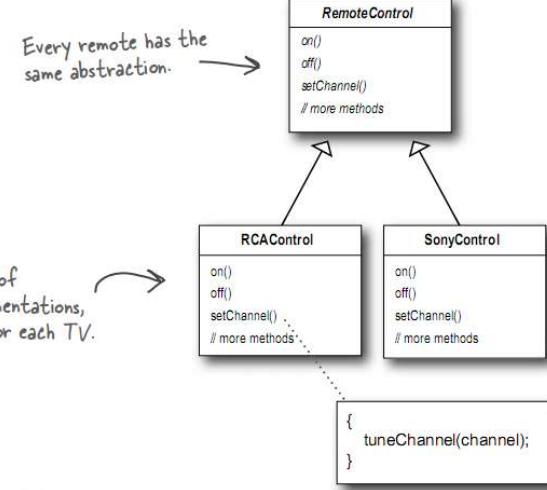
## Bridge pattern

### Example: Remote Control

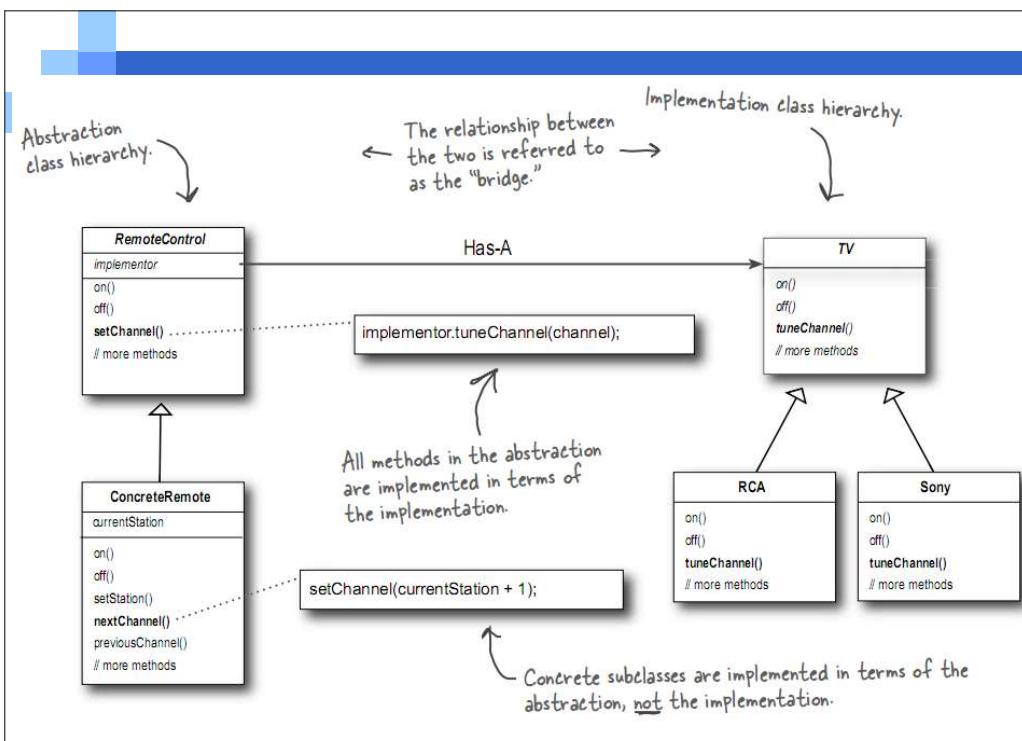
- Các loại remote TV có cùng chung một giao diện trừu tượng và nhiều cách thực thi khác nhau cho nhiều loại TV

#### Vấn đề:

- Remote lần đầu tiên thiết kế không chuẩn và cần phải cải tiến
  - Cả TV và Remote đều thay đổi



34



35

### Bridge pattern

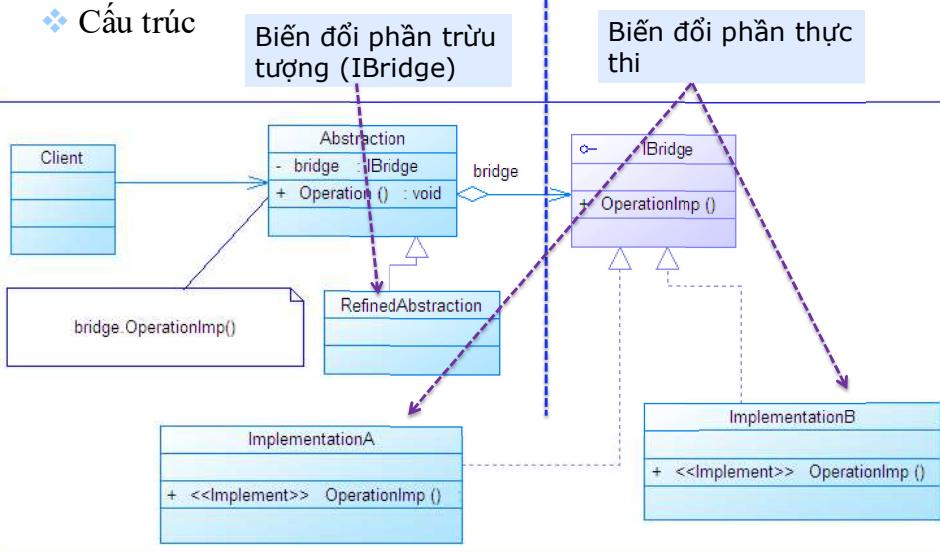
#### Mục đích:

- Tách rời phần trừu tượng ra khỏi sự thực thi của nó sao cho cả hai có thể biến đổi không phụ thuộc nhau

36

## Bridge pattern

### Cấu trúc



37

## Bridge pattern

### Ưu điểm:

- Tách rời sự thực thi sao cho chúng không kết nối vĩnh viễn với một giao diện
- Phần Abstraction và phần Implementation có được mở rộng không phụ thuộc nhau
- Thay đổi các lớp concrete abstraction không ảnh hưởng đến client

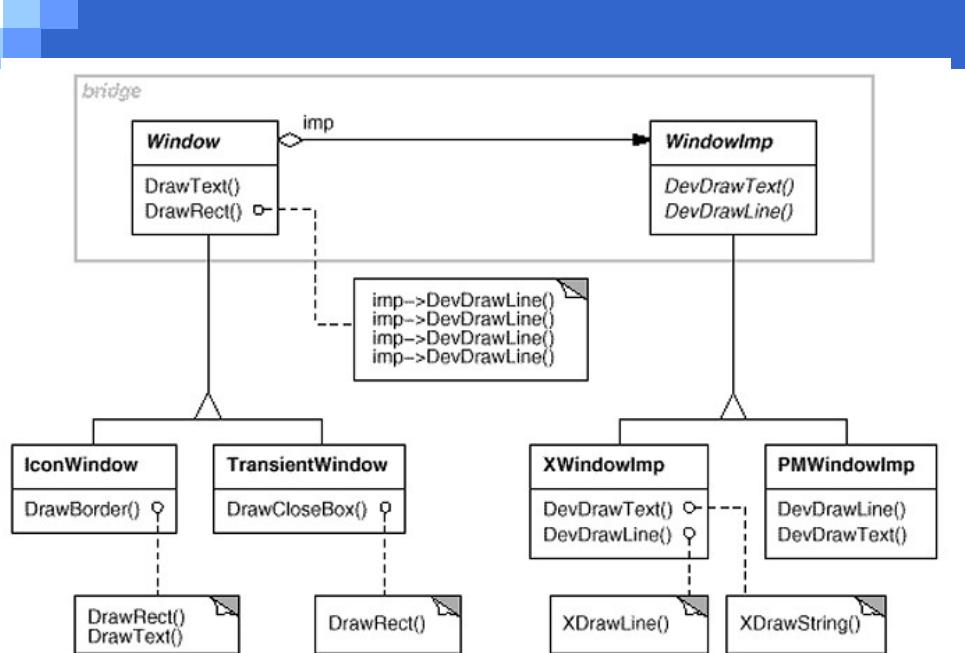
### Hạn chế:

- Gia tăng sự phức tạp

### Sử dụng:

- Sử dụng khi cần biến đổi interface và implementation theo các cách khác nhau
- Trong các hệ thống đồ họa và hệ thống của windows phải chạy trên nhiều nền tảng khác nhau.

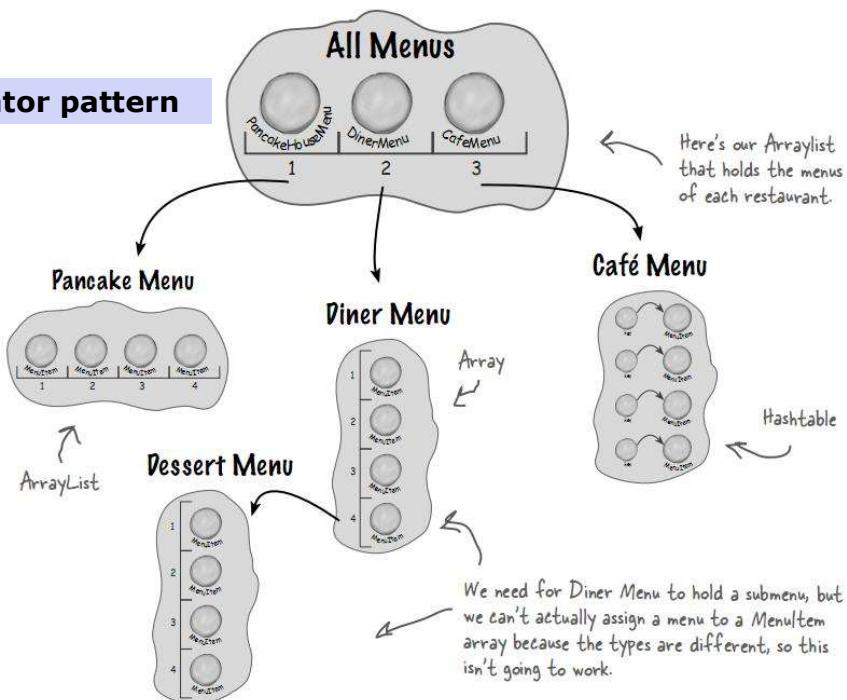
38



39

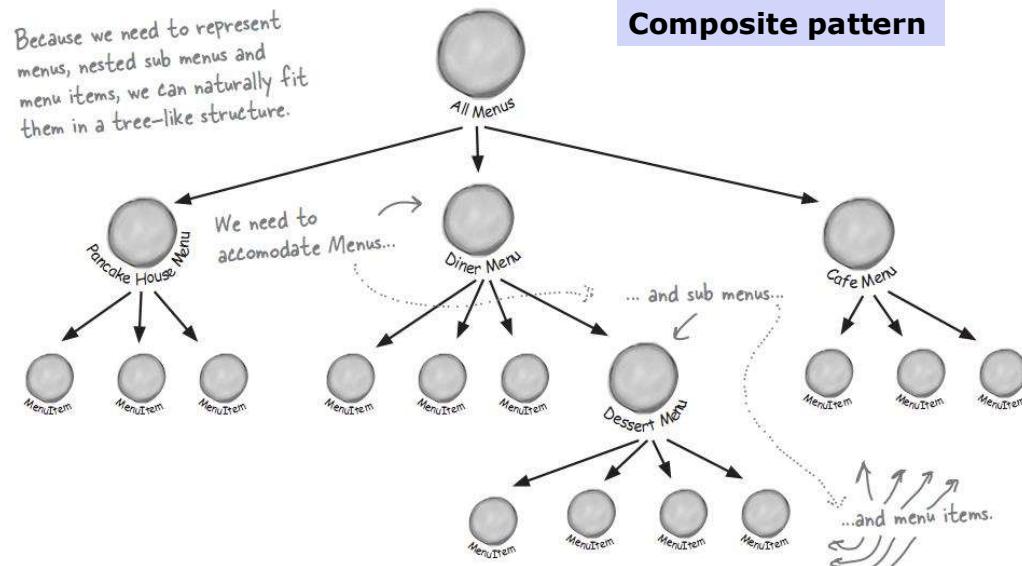
## Composite pattern

## Iterator pattern



Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.

## Composite pattern



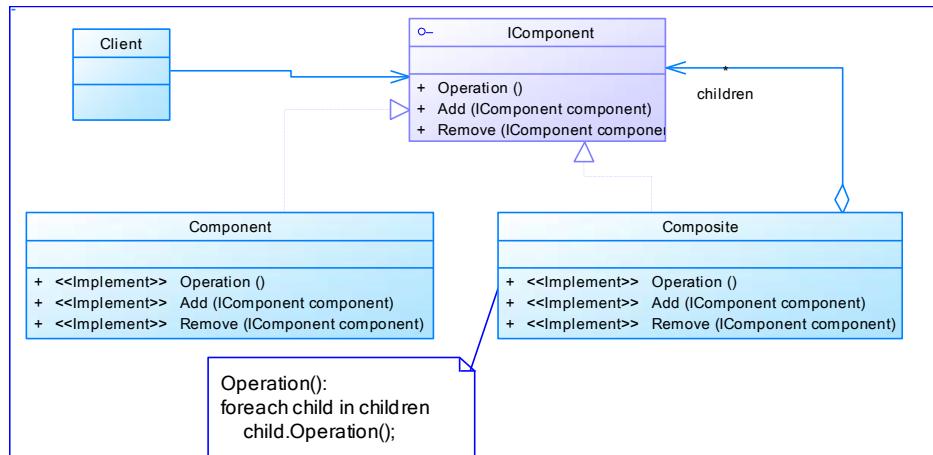
## Composite pattern

### Mục đích:

- Sắp xếp các đối tượng vào các cấu trúc cây để biểu diễn các phân cấp part-whole giữa các đối tượng
- Đối xử với các đối tượng riêng lẻ và nhóm các đối tượng theo cách giống nhau

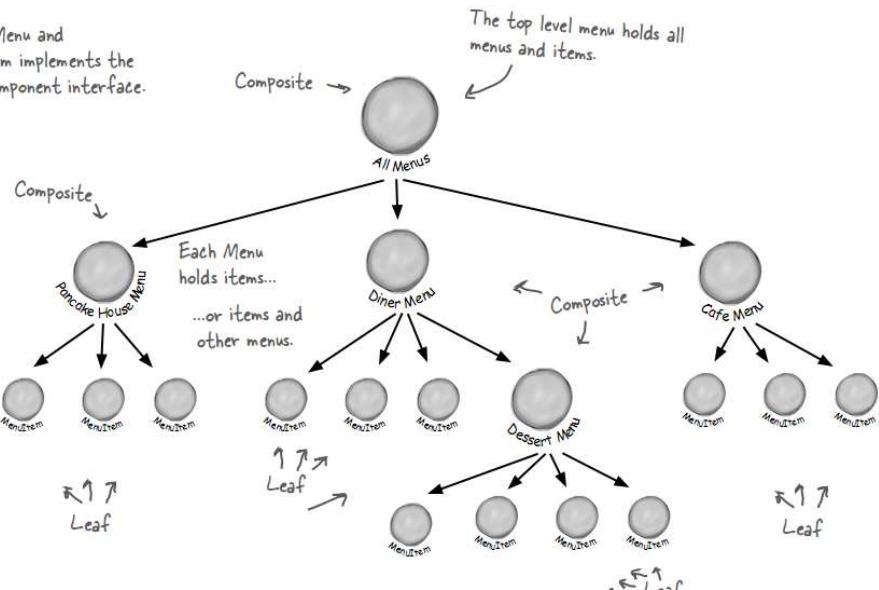
## Composite pattern

### Cấu trúc:



## Composite pattern: Example

Every Menu and MenuItem implements the MenuItem interface.



45

## Composite pattern

- ❖ Composite Pattern cho phép xây dựng các cấu trúc của các đối tượng dưới dạng cây với các node là:
  - composition of objects
  - individual objects
- ❖ Sử dụng cấu trúc Composite ta có thể áp dụng cùng một Operation cho cả hai loại đối tượng Composite và Individual  
→ Làm mất đi sự khác biệt đối với hai loại đối tượng

## Questions

- ❖ So sánh giữa Iterator Pattern và Composite Pattern. Có thể thay thế Composite Pattern bằng Iterator Pattern được không

47

## Tài liệu tham khảo

- ❖ Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First Design pattern. O'Reilly 2006.
- ❖ **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
- ❖ <http://www.dofactory.com/Patterns/Patterns.aspx>

46

48

## ***Behavioral patterns***

**Giảng viên: Huỳnh Tuấn Anh**  
**Khoa CNTT - Đại học Nha Trang**

2

### **Behavioral patterns**

- ❖ Liên quan tới các giải thuật và các quan hệ giữa các đối tượng, cách các đối tượng giao tiếp với nhau

## **Behavioral patterns**

- ❖ Chain of Responsibility.
- ❖ Command pattern
- ❖ Interpreter pattern
- ❖ Iterator pattern
- ❖ Mediator pattern
- ❖ Memento pattern
- ❖ Observer pattern
- ❖ State pattern
- ❖ Strategy pattern
- ❖ Template Method
- ❖ Visitor pattern

3

## ***Observer pattern***

## observer pattern

- Mục đích: Định nghĩa một phụ thuộc one-to-many giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc nó được thông báo và được cập nhật một cách tự động.

5

## Các ví dụ về observer pattern

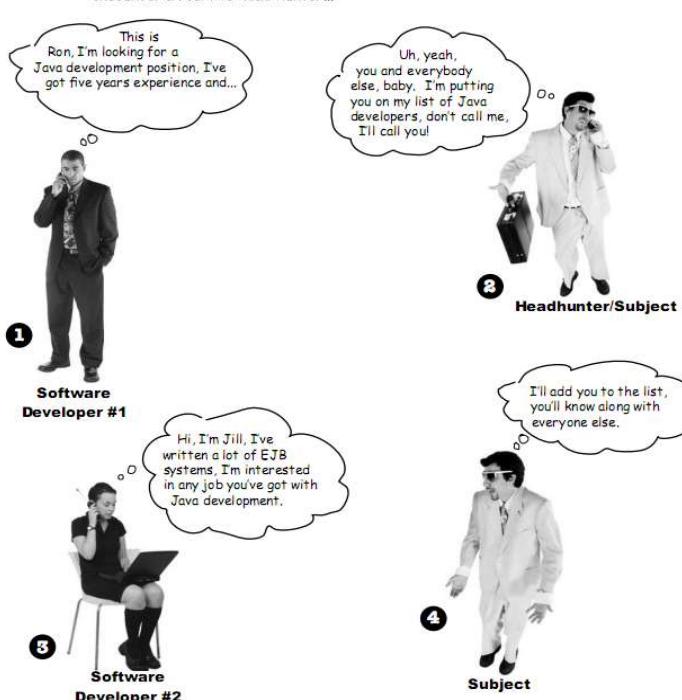
- Chương trình bảng tính Excel
- Data binding
- Đặt mua báo dài hạn
  - Độc giả đăng ký mua báo dài hạn với tòa soạn
  - Khi có một tờ báo mới xuất bản thì nó được đại lý phân phối đến độc giả.
  - Publishers + Subscribers = Observer Pattern**



6

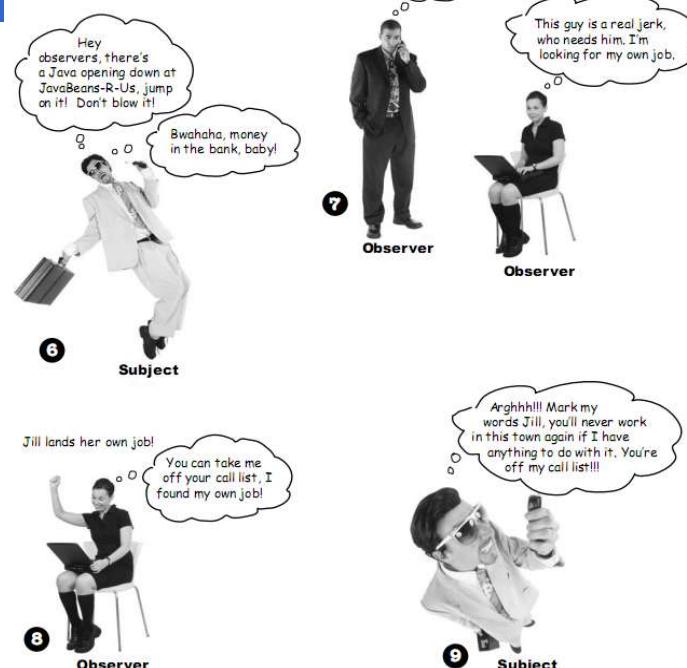
### Five minute drama: a subject for observation

In today's skit, two post-bubble software developers encounter a real live head hunter ...



7

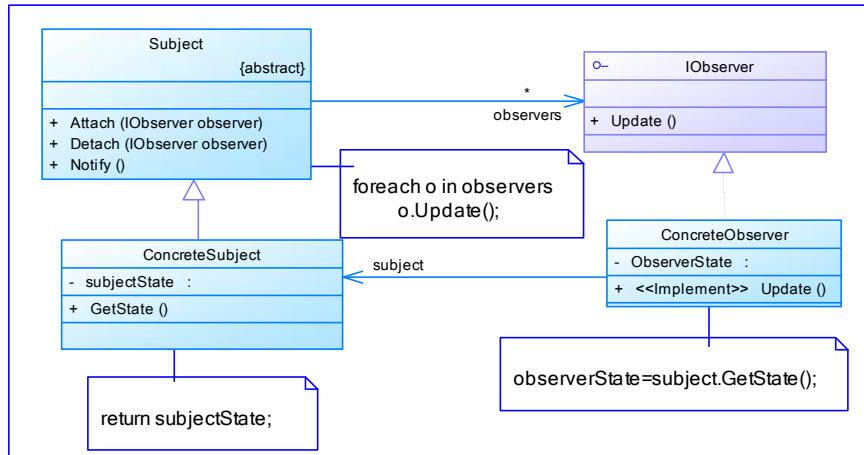
Meanwhile for Ron and Jill life goes on; if a Java job comes along, they'll get notified, after all, they are observers.



8

## Observer pattern

### Cấu trúc



9

## Questions

- ❖ Giải thích vai trò của hàm `Update()` từ đó cho biết vai trò của **IObserver**
- ❖ Có thể thay thế **Observer pattern** bằng mô hình các đối tượng dùng chung dữ liệu được không?
- ❖ Loosely coupled design: Hãy phân tích mối quan hệ giữa 2 **ConcreteSubject** và **ConcreteObserver**
- ❖ Viết mã lệnh cho cấu trúc của **Observer pattern**

10

## Bài tập

- ❖ Tìm hiểu mô hình lập trình RxJava
- ❖ Tìm hiểu mô hình lập trình RxJava cho Android

11

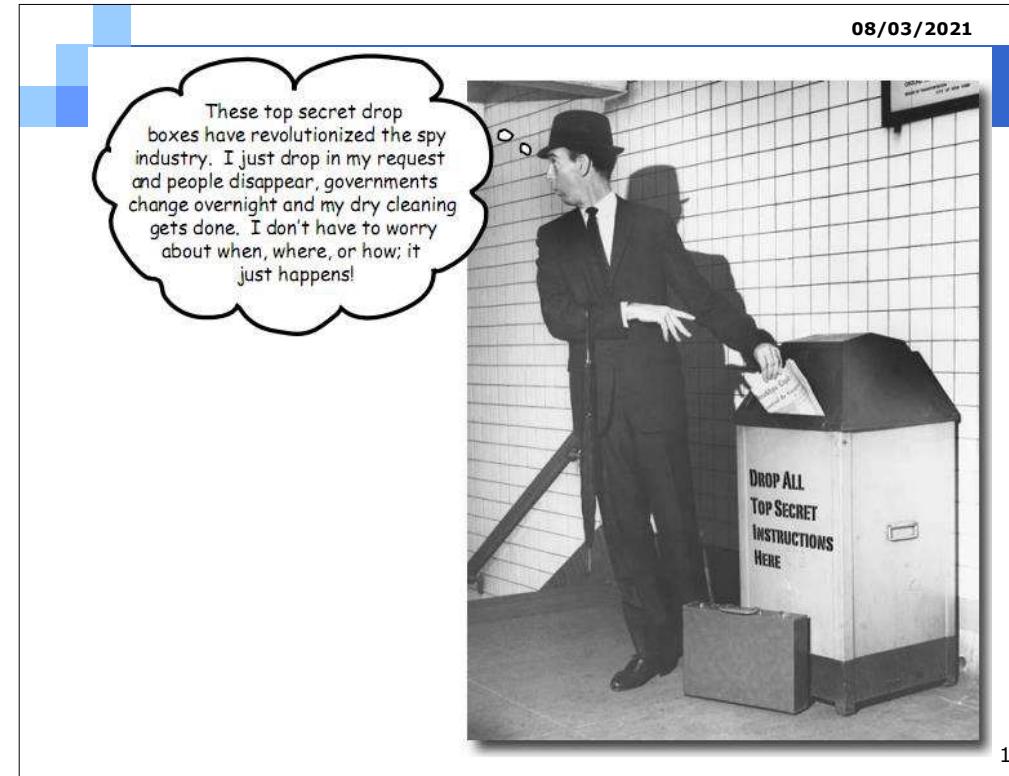
## Design Principle

*Strive for loosely coupled designs between objects that interact.*



12

## Command pattern



14

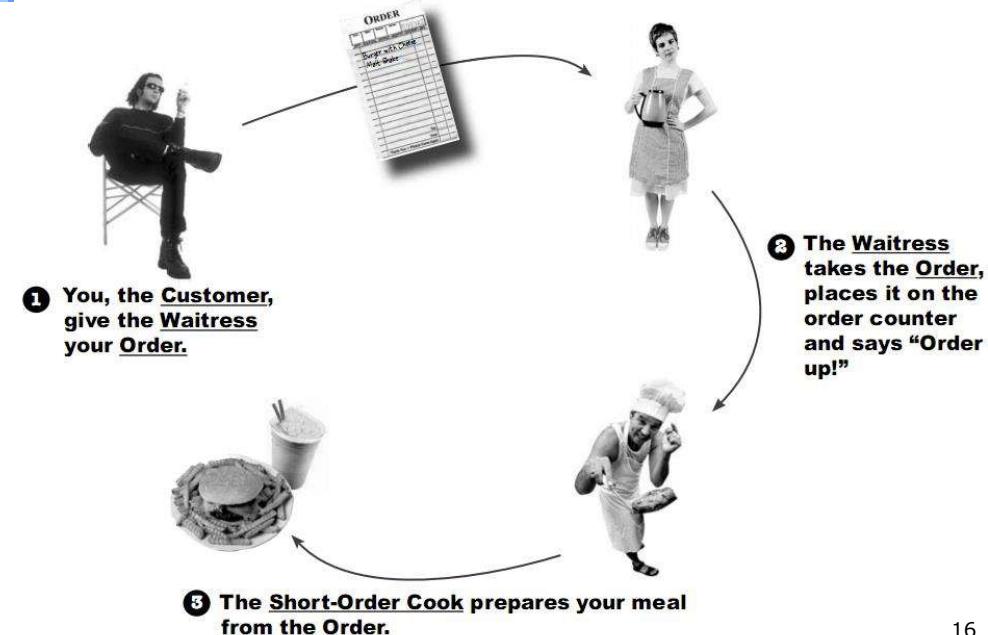
## Command pattern

08/03/2021

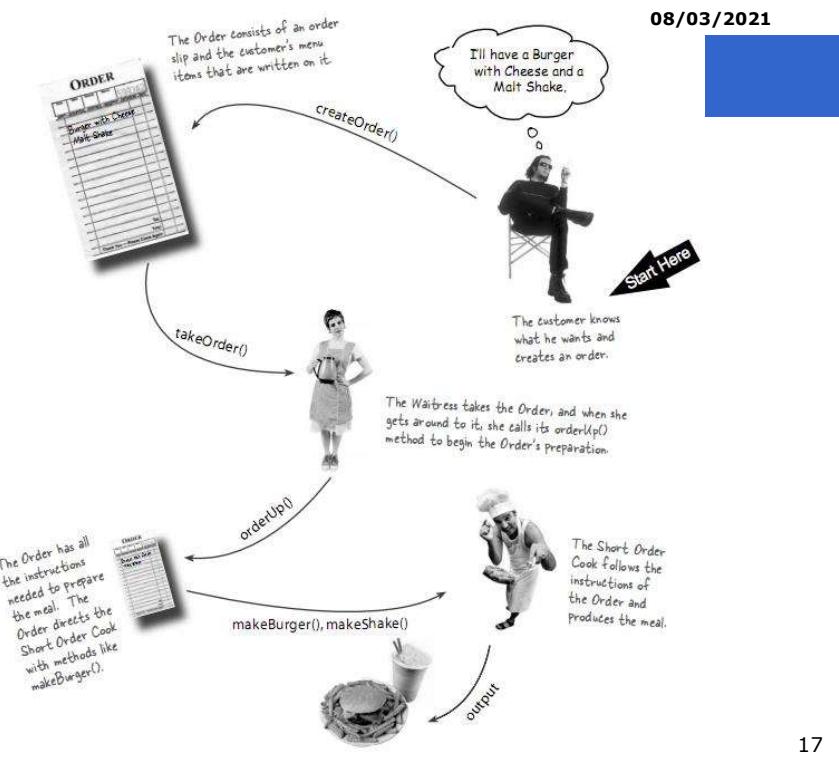
- ❖ Mục đích:
  - Đóng gói requests thành một đối tượng.
  - Cho phép tham số hóa các client với các requests khác nhau
  - Tách rời request một hành động ra khỏi đối tượng thực hiện hành động đó

15

## Example: Diner operates

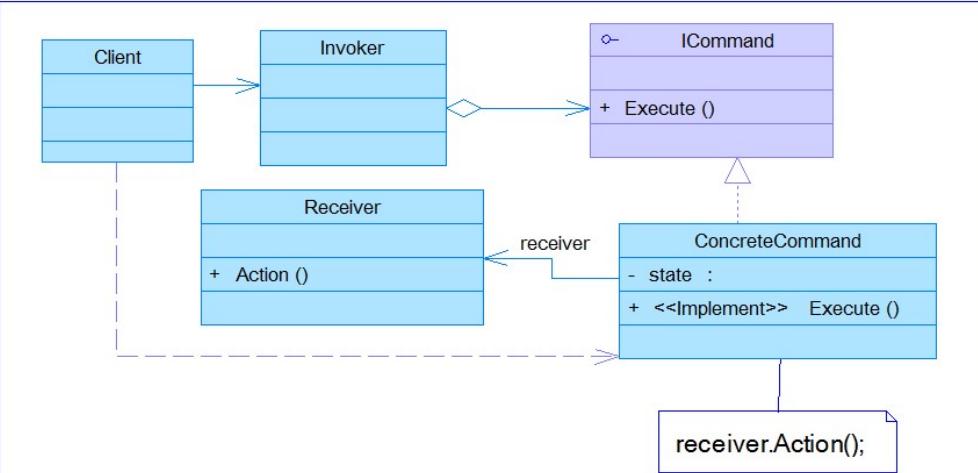


16



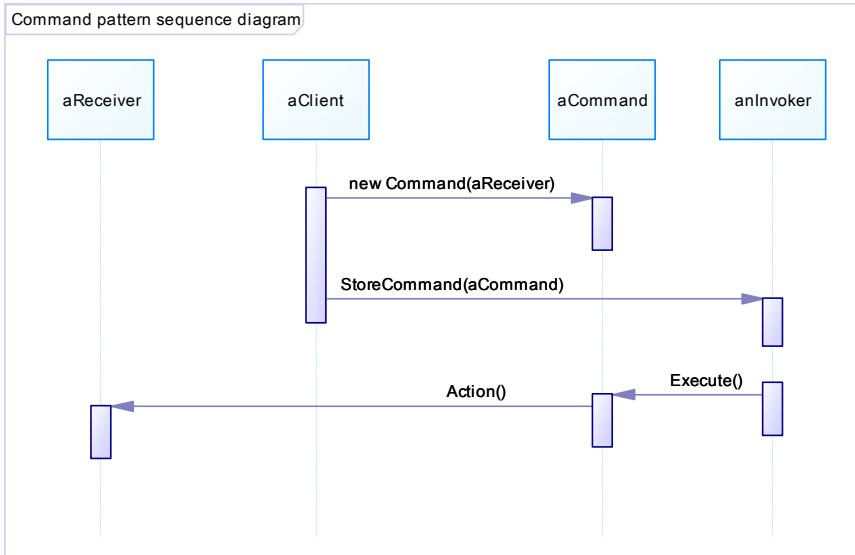
## Command pattern

❖ Cấu trúc:



18

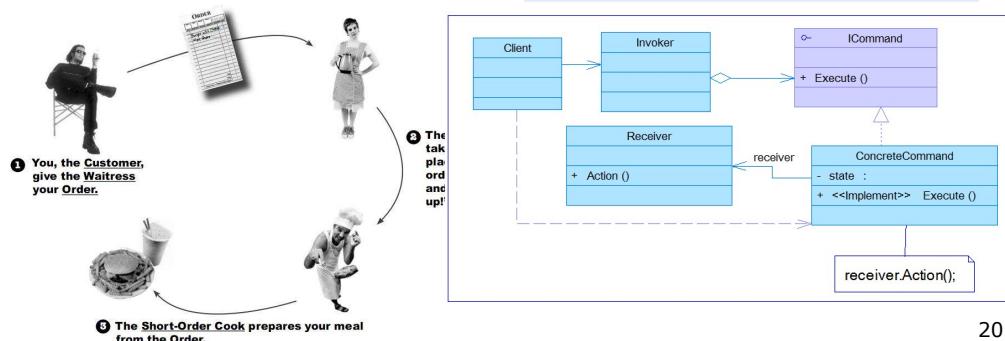
## Collaboration



19

## Matching between Command & Diner Example

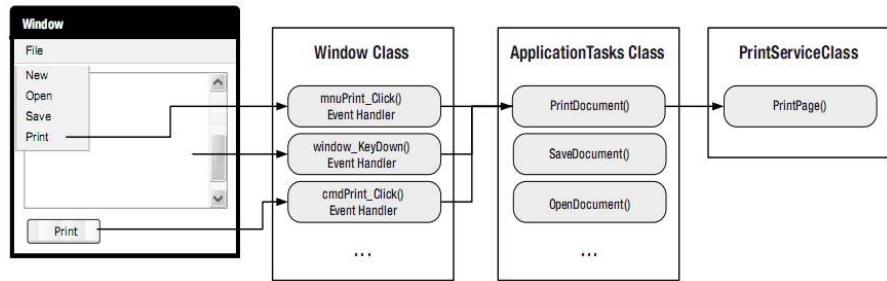
Diner Example	Command pattern
Customer	Client
Order	Command
Waitress	Invoker
TakeOrder()	setCommand()
Short Order Cook	Receiver
orderUp()	Execute()



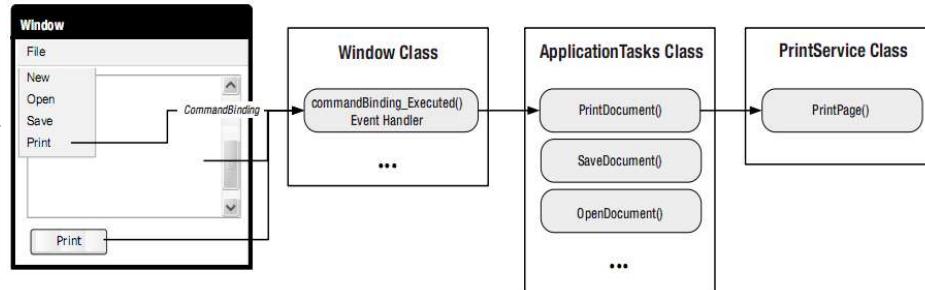
20

## Ứng dụng

Windows Form



WPF

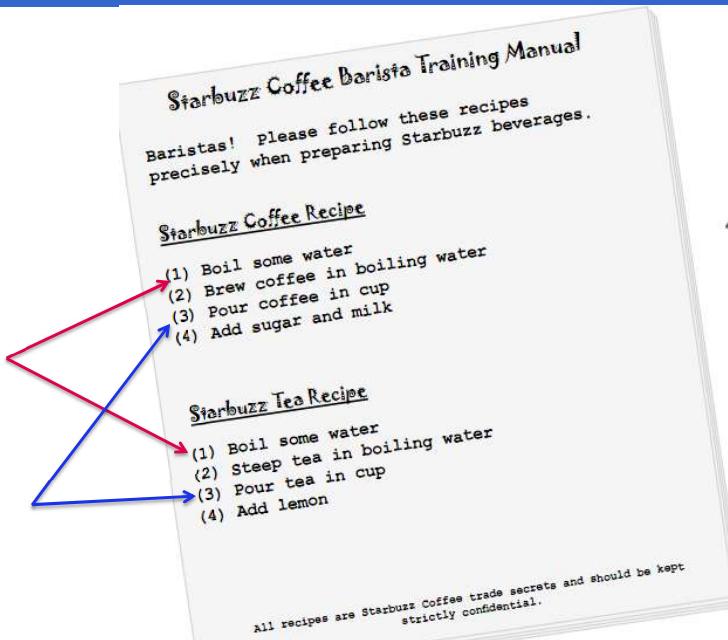


## Questions

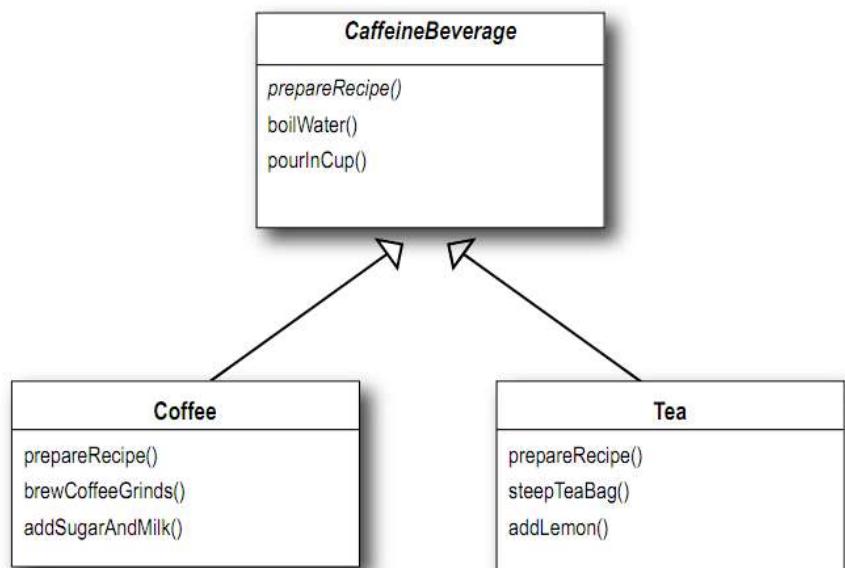
- ❖ Làm thế nào Command pattern tách rời sự phụ thuộc giữa *invoke of a request* và *receiver of the request*?

## Template method

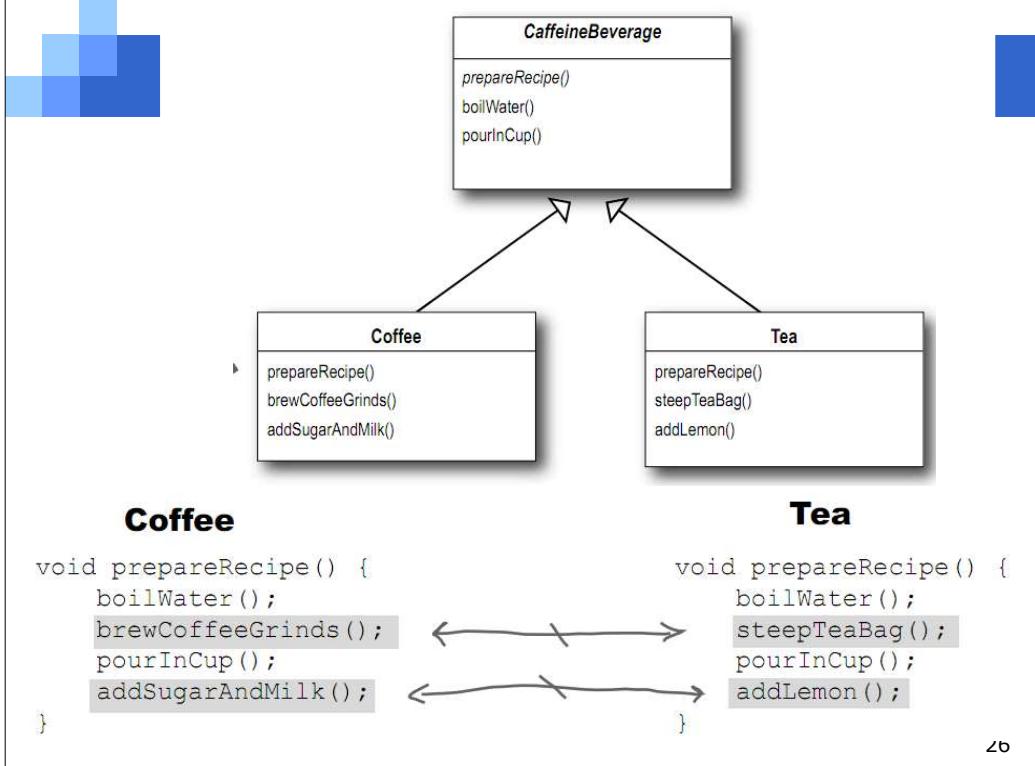
## Tea and Coffee examples



## Tea and Coffee example

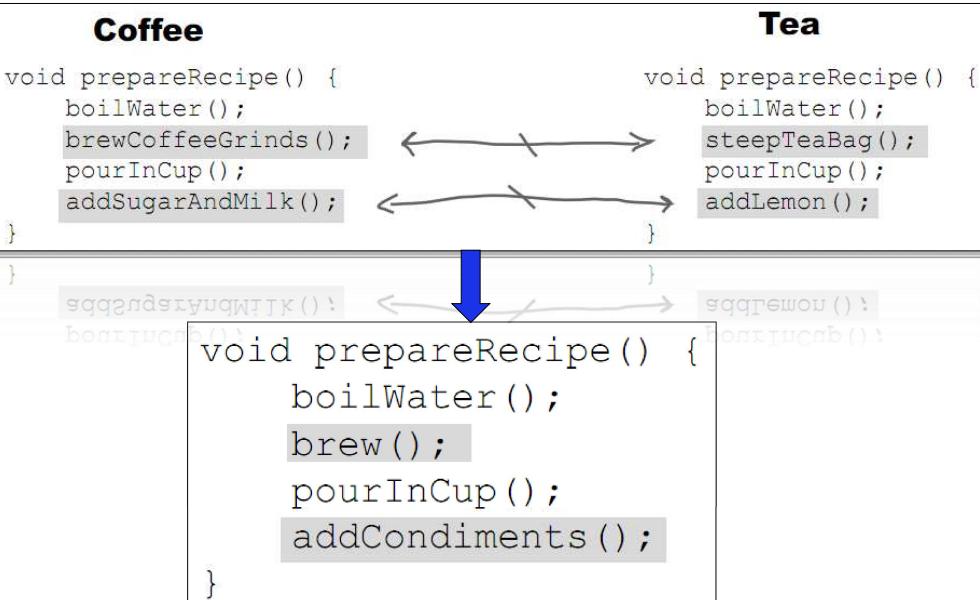


25

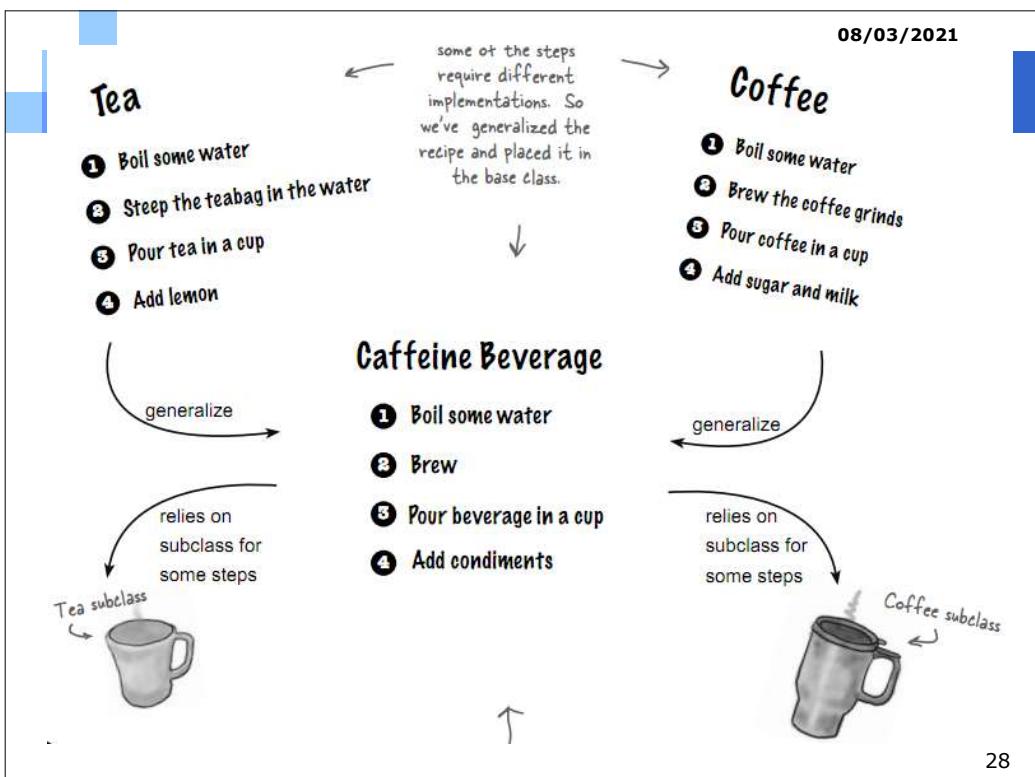


26

## Tea and Coffee example...

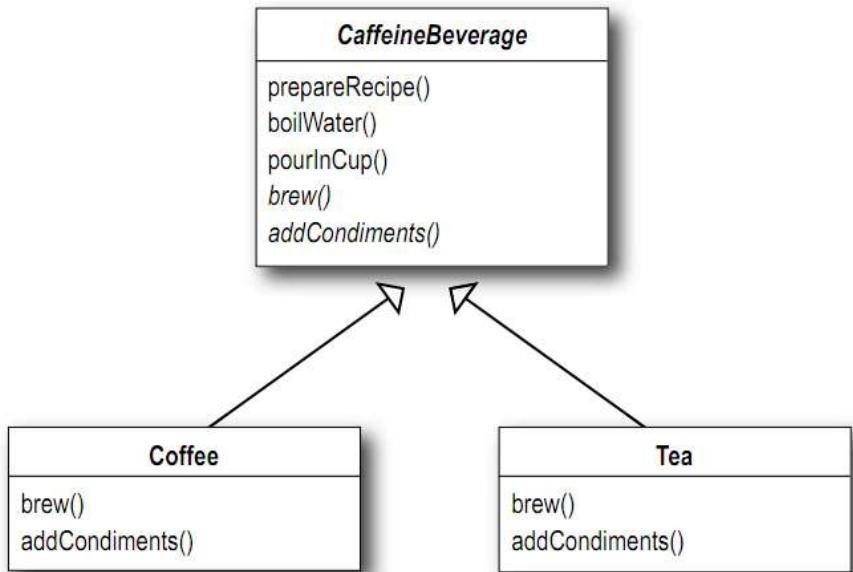


27



28

## Tea and Coffee example...



29

## template method

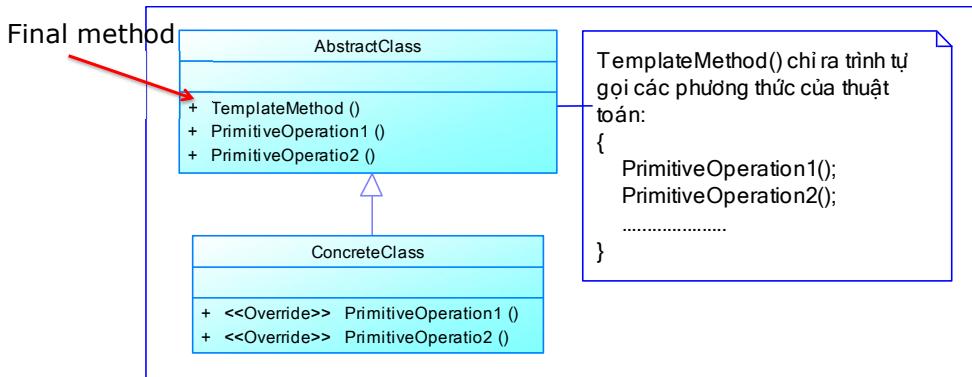
### ❖ Mục đích:

- Định nghĩa khung sườn của một thuật toán bao gồm nhiều bước trong một phương thức.
- Một số bước được khai báo abstract ở lớp cơ sở và ủy quyền cho lớp con cài đặt
- Việc cài đặt các bước ở lớp con không làm ảnh hưởng đến cấu trúc của thuật toán.

30

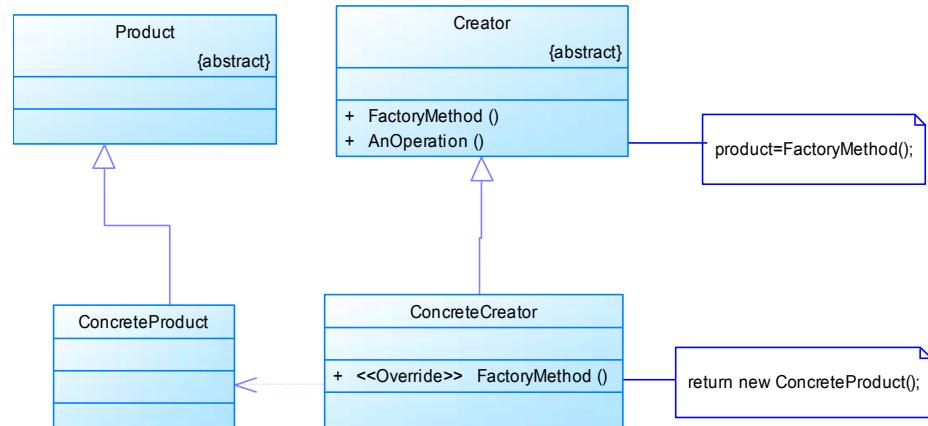
## Template method

### ❖ Cấu trúc



31

## Liên hệ giữa factory method và template method

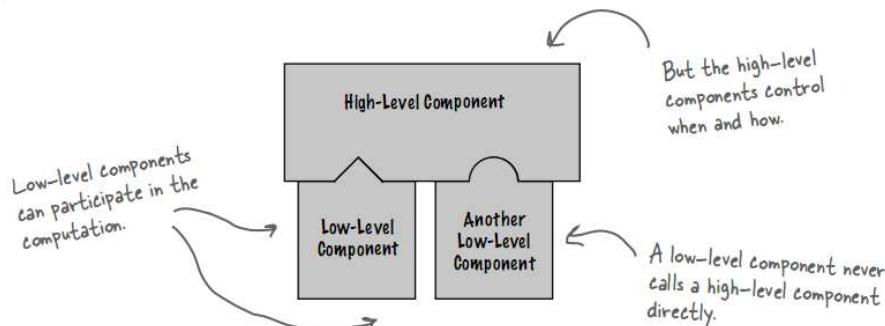


32



## The Hollywood Principle

*Don't call us, we'll call you.*



33

## Questions

- ❖ Có thể dùng interface để thay thế AbstractClass không? Nếu được hãy vẽ lại sơ đồ cấu trúc của template method?
- ❖ Các ConcreteClass có cần phải thực thi toàn bộ các phương thức của lớp AbstractClass?
- ❖ So sánh Template method với Strategy pattern, factory method

35

## State pattern

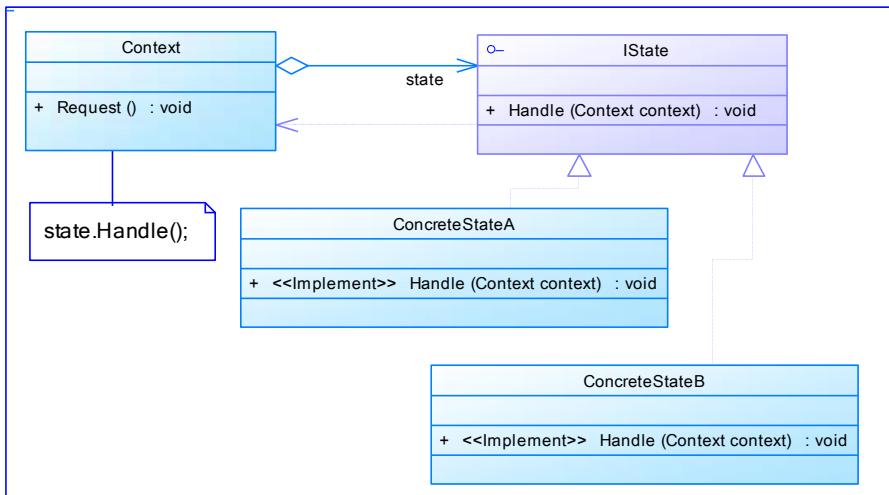
## State pattern

- ❖ **Mục đích:** Cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi. Đối tượng đó sẽ xuất hiện để thay đổi lớp của nó.

37

## State pattern

- ❖ Cấu trúc:



38

## State pattern: Sử dụng

- ❖ Một đối tượng có nhiều trạng thái
  - Hành vi của đối tượng phụ thuộc vào các trạng thái của đối tượng và có thể thay đổi lúc run-time
  - Nhiều operation phụ thuộc vào trạng thái của đối tượng. Thông thường chỉ vài operation cùng phụ thuộc vào trạng thái của đối tượng.
- ❖ Sử dụng State pattern:
  - Đối tượng → Context
  - Mỗi trạng thái → ConcreteState
    - ConcreteState chứa các operation phụ thuộc vào trạng thái
  - Chuyển trạng thái của đối tượng: ConcreteState, Context
    - Handle(Context context)

39

## Questions

- ❖ Trong State pattern class diagram, IState là một interface, có thể thay thế IState bằng một abstract class được không? Nếu được hãy chỉ ra khi nào thì thay thế
- ❖ State pattern làm tăng số lớp cài đặt của ứng dụng. Tại sao ta vẫn sử dụng State pattern trong OOD (Object Oriented Design)
- ❖ Giả sử trong Context có biến s kiểu IState. Giải thích vì sao một lớp ConcreteState có thể thay đổi trạng thái của biến s.
- ❖ Client có thể tương tác với biến kiểu IState trong context được không? giải thích.

40

## Chain of Responsibility

## Chain of Responsibility

❖ Mục đích:

- Tách rời người gửi và người thực hiện request

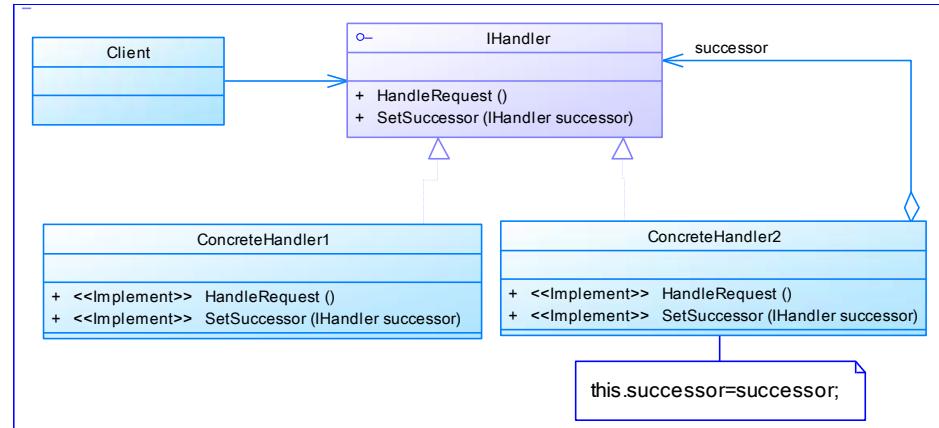
❖ Ý tưởng:

- Kết nối các đối tượng thực hiện request thành một chuỗi
- Chuyển request dọc theo chuỗi cho đến khi gặp được đối tượng có khả năng xử lý được nó

42

## Chain of Responsibility

❖ Cấu trúc:



43

## Question

- ❖ Có thể thay đổi Chain of Responsibility bằng cách dùng cấu trúc lệnh if...else của các ngôn ngữ lập trình được không?

44

## Sử dụng

- ❖ “Người gửi” không biết phải gửi request cho “người nhận” nào trong tập các “người nhận”
- ❖ Chuỗi các “người nhận” có thể thay đổi lúc run-time

45

# Iterator pattern

08/03/2021

They want to use my Pancake House menu as the breakfast menu and the Diner's menu as the lunch menu. We've agreed on an implementation for the menu items...

Lou



Network  
Internet access

47

## Iterator

### Vấn đề:

- Các tập hợp khác nhau được biểu diễn theo các cách khác nhau
- Client truy cập tới các phần tử của tập hợp theo một cách duy nhất
- Client không cần biết cấu trúc của từng tập hợp cụ thể

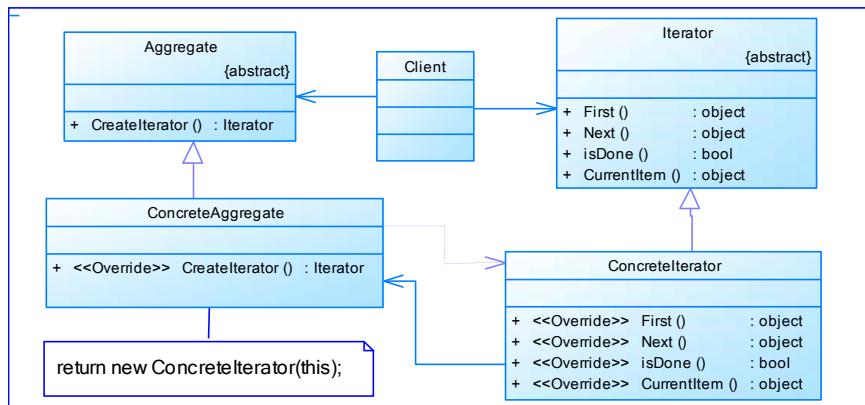
08/03/2021

## Iterator

### Mục đích:

- Cung cấp một cách truy cập các phần tử của một tập hợp một cách tuần tự mà không cần biết cấu trúc của tập hợp đó.

### Cấu trúc



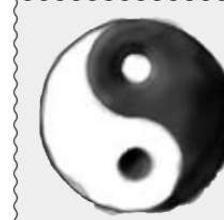
48

49

## Questions

- ❖ Có thể gộp chung hai lớp Iterator và Aggregate được không?  
Nếu ưu và khuyết điểm của cách làm này

50



### **Design Principle**

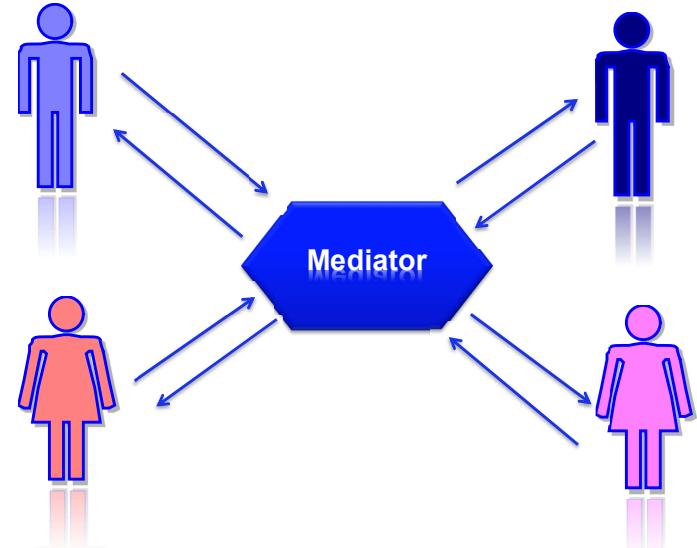
*A class should have only one reason to change.*

**Mỗi class chỉ nên thiết kế với một single responsibility**

51

## **Mediator pattern**

### **Mediator: Example**

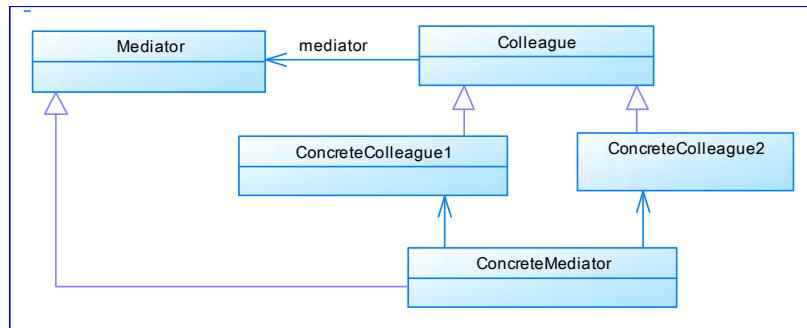


53

## Mediator pattern

❖ Mục đích:

- Đóng gói các cách tương tác giữa một tập các đối tượng



54

## Mediator

❖ Ưu điểm

- Gia tăng việc tái sử dụng các đối tượng được hỗ trợ bởi Mediator bằng cách tách rời chúng ra khỏi hệ thống
- Đơn giản hóa việc duy trì hệ thống bằng cách tập trung logic điều khiển
- Đơn giản hóa và giảm sự thay đổi các message được gửi giữa các đối tượng trong hệ thống

❖ Hạn chế

- Mediator có thể rất phức tạp nếu không được thiết kế một cách phù hợp.

❖ Sử dụng

- Mediator thường được sử dụng để phối hợp các thành phần GUI với nhau

55

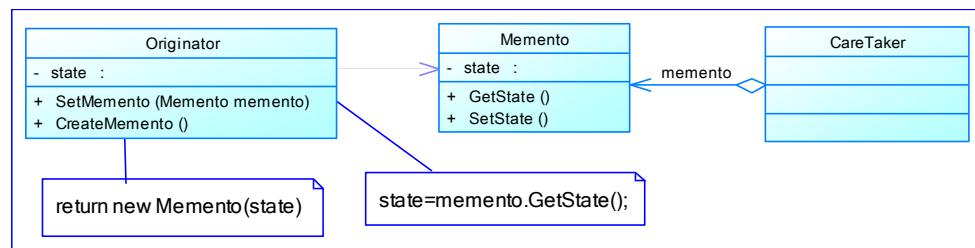
## Memento pattern



## Memento pattern

❖ Mục đích:

- Lưu lại trạng thái của một đối tượng để khôi phục lại sau này mà không vi phạm nguyên tắc đóng gói.

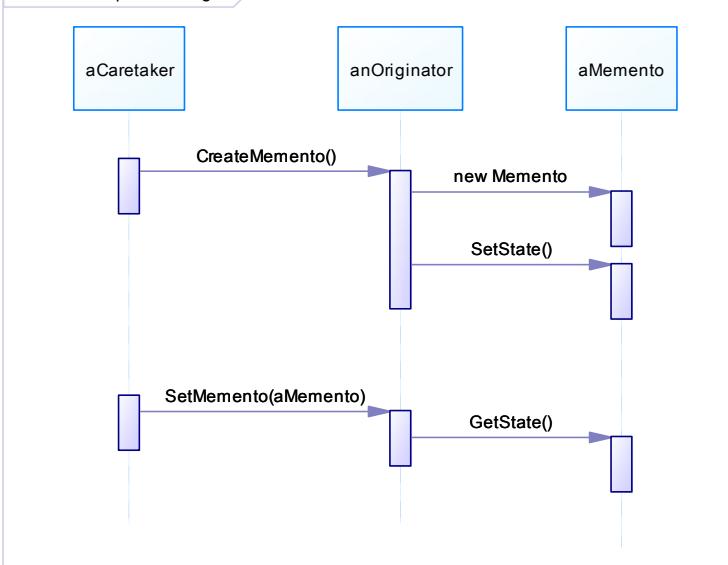


57

## Collaboration

08/03/2021

Memento sequence diagram



58

## Visitor pattern

## Vấn đề

- ❖ Xử lý thông tin của một phần tử trong một cấu trúc đối tượng cho trước
- ❖ Việc xử lý thông tin như thế nào chưa thể xác định lúc compile-time

60

## Visitor pattern

- ❖ Mục đích:
  - Thực hiện một thao tác trên một phần tử của một cấu trúc phức hợp (composite structure)
  - Định nghĩa phương thức mới thao tác trên một phần tử của cấu trúc mà không cần thay đổi các lớp đã được định nghĩa trên cấu trúc đó

61

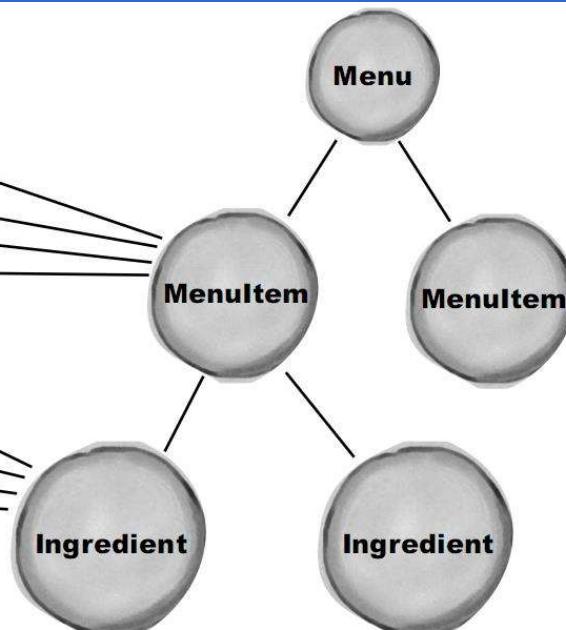
## Visitor pattern: Example

// new methods

getHealthRating  
getCalories  
getProtein  
getCarbs

// new methods

getHealthRating  
getCalories  
getProtein  
getCarbs



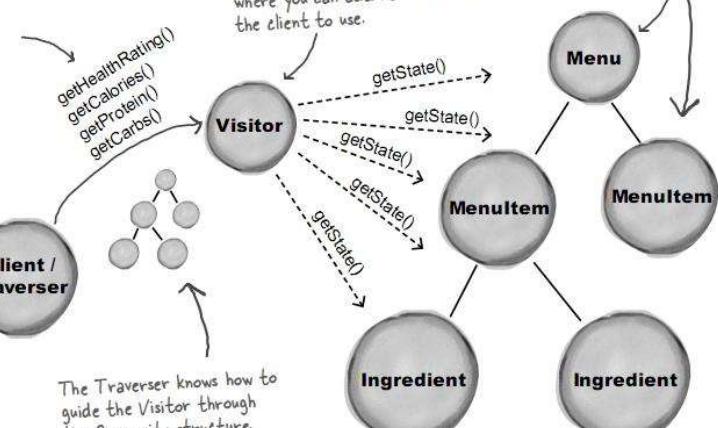
62

## Visitor pattern: Example

The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.

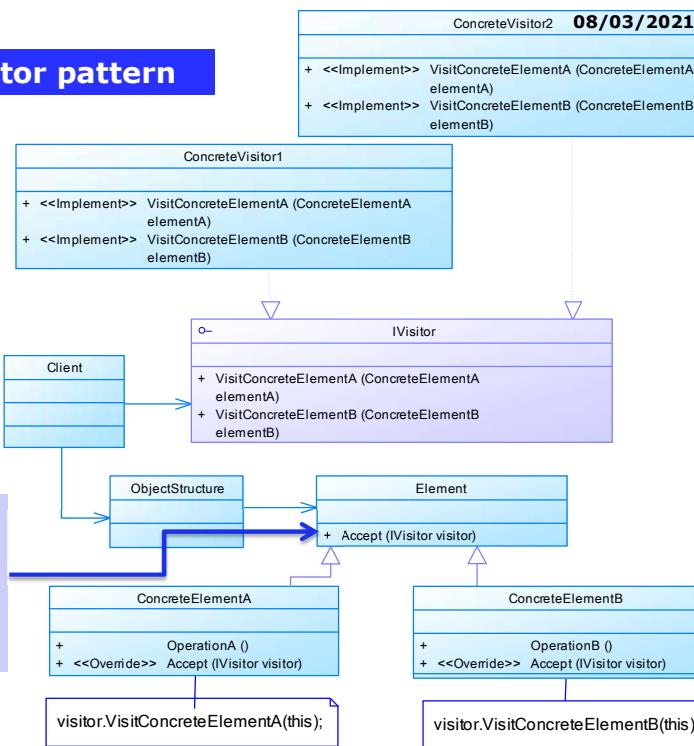
The Visitor needs to be able to call getState() across classes, and this is where you can add new methods for the client to use.

All these composite classes have to do is add a getState() method (and not worry about exposing themselves).



63

## Giải pháp: Visitor pattern



64

## Visitor pattern

### ❖ Ưu điểm

- Cho phép thêm các thao tác xử lý tới một Composite structure mà không cần thay đổi cấu trúc đó
- Việc thêm mới một operation khá dễ dàng
- Mã lệnh của các operation được thực hiện bởi Visitor được tập trung

### ❖ Hạn chế

- Tính chất đóng gói (encapsulation) của các lớp bị phá vỡ khi sử dụng Visitor
- Việc thay đổi nó để phù hợp với cấu trúc Composite gấp nhiều khó khăn do phải dùng hàm duyệt cấu trúc của Composite.

65

## Tài liệu tham khảo

- ❖ Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates.  
Head First Design pattern. O'Reilly 2006.
- ❖ **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
- ❖ <http://www.dofactory.com/Patterns/Patterns.aspx>