**HEAD FIRST DESIGN PATTERNS**

NỘI DUNG TÌM HIỂU

Chapter 1. Intro to Design Patterns – Strategy Pattern

Chapter 2. The Observer Pattern

Chapter 3. The Decorator Pattern

Chapter 4. The Factory Pattern

Chapter 5. The Singleton Pattern

Chapter 6. The Command Pattern

Chapter 7. The Adapter and Facade Pattern

Chapter 8. The Template Method Pattern – Encapsulating Algorithms

Chapter 9. The Iterator and Composite Pattern

Chapter 10. The State Pattern – State of Things

Chapter 11. The Proxy Pattern – Controlling Object Access

Chapter 12. The Compound Pattern – Patterns of Patterns

Chapter 13. Patterns in the Real World

Phụ lục 1. Tại sao sử dụng Design Pattern?

Phụ lục 2: Tổng hợp các nguyên tắc thiết kế (Design Principle)

**Chapter 1. Intro to Design Patterns – Strategy Pattern**

Bắt đầu bằng ứng dụng đơn giản SimUDuck: trò chơi mô tả đặc điểm riêng (display), mô phỏng tiếng kêu (quack), khả năng bơi lội (swim), và khả năng bay (fly) của các loài vịt khác nhau.

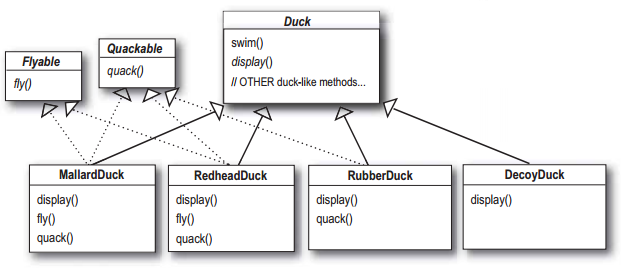
Đầu tiên, lập trình viên Joe thiết kế phần mềm sử dụng Inheritance (kế thừa):

Joe tạo một superclass **Duck** chứa tất cả các phương thức mong muốn. Tiếp theo, với mỗi loài vịt anh ta tạo một subclass tương ứng kế thừa từ superclass Duck, như Mallarduck, RedheadDuck, RubberDuck,...



Ta có thể thấy cách thiết kế trên có một vấn đề. Khi thêm một phương thức như ***fly()*** vào supperclass Duck, tất cả các subclass cũng sở hữu phương thức này. Ví dụ như ở trên, subclass *DecoyDuck* không thể quack hay fly, do đó, hai phương thức này cần phải được overrride. Đây là một bài toán khó khi có một số lượng lớn subclass phải maintain hay thêm vào các subclass mới.

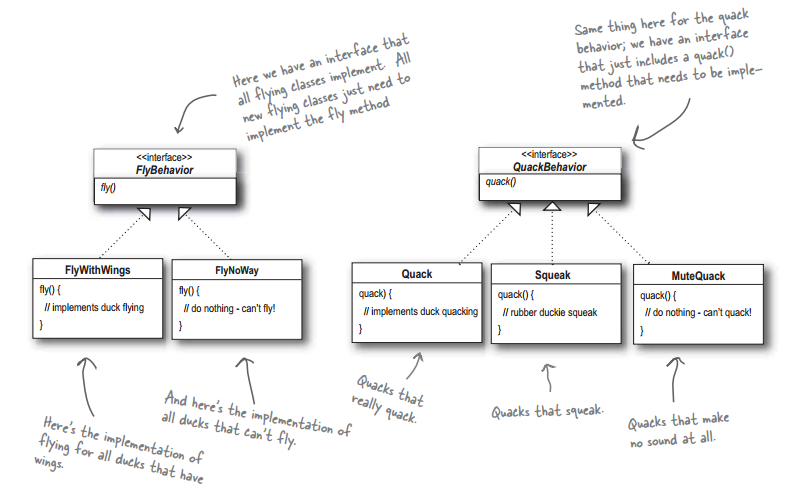
Do đó, Joe áp dụng Interface để giải quyết hạn chế trên như sau:



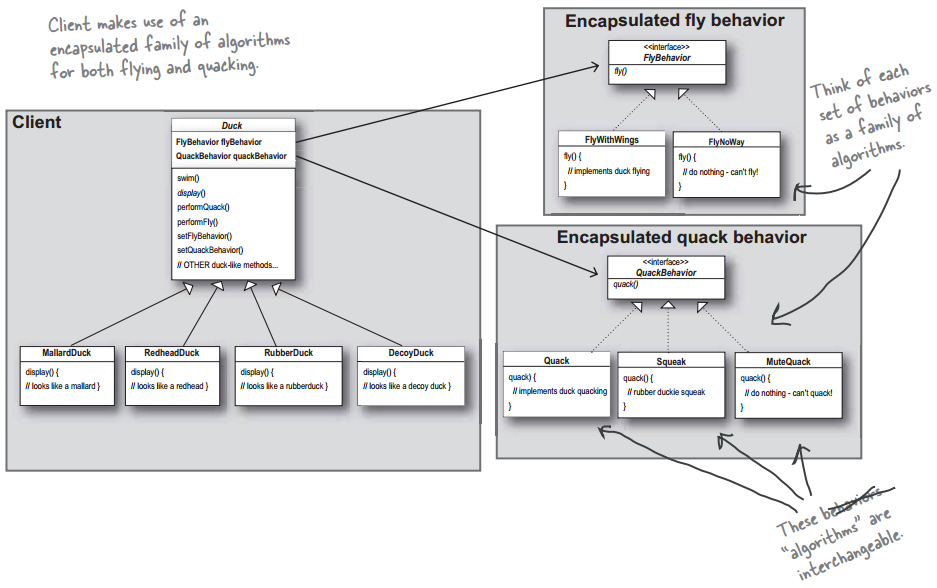
Nguyên tắc thiết kế thứ 1: Xác định các thuộc tính sẽ thay đổi, sau đó tách và “đóng gói” chúng ta ra khỏi các thuộc tính không đổi.

Nguyên tắc thiết kế thứ 2: Lập trình một interface, không phải một implementation.

Áp dụng vào Duck app, Joe tạo ra 2 interface là FlyBehavior và QuackBehavior.



Superclass Duck implement interface bằng cách khai báo hai biến flyBehavior và quackBehavior; cùng với hai phương thức performFly() và performQuack().



Bây giờ, Joe bắt tay vào code chương trình áp dụng Interface FlyBehavior.

1. FlyBehavior Interface (FlyBehavior.java) and two behavior implementation classes FlyWithWings.java and FlyNoWays.java

|  |
| --- |
| **public** **interface** FlyBehavior {  **public** **void** fly();  } |
| **public** **class** FlyWithWings **implements** FlyBehavior {  **public** **void** fly() {  System.***out***.println("I believe that I can fly");  }  } |
| **public** **class** FlyNoWay **implements** FlyBehavior {  **public** **void** fly() {  System.***out***.println("Sorry I cannot fly");  }  } |

2. Duck class (Duck.java) and RedheadDuck class (RedheadDuck.java)

|  |
| --- |
| **public** **class** Duck {  FlyBehavior flyBehavior;    **public** Duck() {  }    **public** **void** performFly() {  flyBehavior.fly();  }    **public** **void** setFlyBehavior(FlyBehavior fb) {  flyBehavior = fb;  }  } |
| **public** **class** RedheadDuck **extends** Duck {  **public** RedheadDuck() {  **this**.flyBehavior = **new** FlyWithWings();  }    **public** String toString() {  **return** "I am a real RedheadDuck";  }  } |

3. DuckSimulator main class (DuckSimulator.java)

|  |
| --- |
| **public** **class** DuckSimulator {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  Duck duck = **new** RedheadDuck();  System.***out***.println(duck);  duck.performFly();    // Change flyBehavior  duck.setFlyBehavior(**new** FlyNoWay());  duck.performFly();  }  } |

4. Output result

|  |
| --- |
| I am a real RedheadDuck  I believe that I can fly  Sorry I cannot fly |

Ta có thể thấy ưu điểm của thiết kế sử dụng interface này:

+ Tính đóng gói: Duck class chỉ đơn giản gọi hàm performFlyBehavior(), nó không biết hàm này thực hiện những công việc gì.

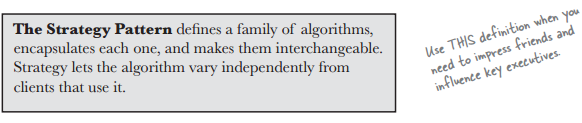
+ Tính mềm dẻo: ta có thể thay đổi behavior trong lúc chạy chương trình (setFlyBehavior).

Nguyên tắc thiết kế thứ ba: Favor composition over inheritance. Hay “HAS-A” có thể tốt hơn “IS-A”.

Khi bạn đặt hai class với nhau tức là bạn đang sử dụng composition. Như cách Joe sử dụng thuộc tính kiểu FlyBehavior trong Duck class. Thay vì kế thừa các thuộc tính từ Duck class, các subclass như RedheadDuck thiết lập hành vi của chúng bằng cách kết hợp với đối tượng cần thiết (như là FlyNoWay hay FlyWithWings).

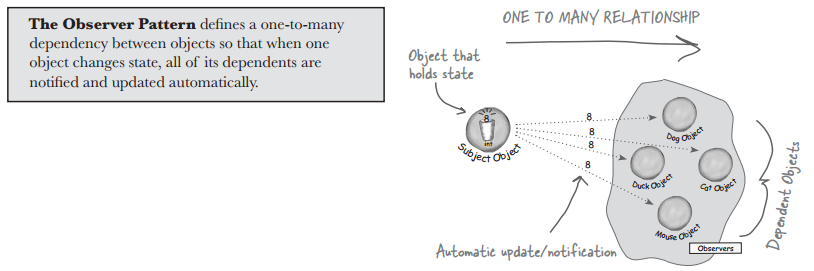
Pattern đầu tiên:

Lúc Joe sử dụng Interface để cải tiến SimUDuck app, anh ta đã sử dụng Strategy Pattern.





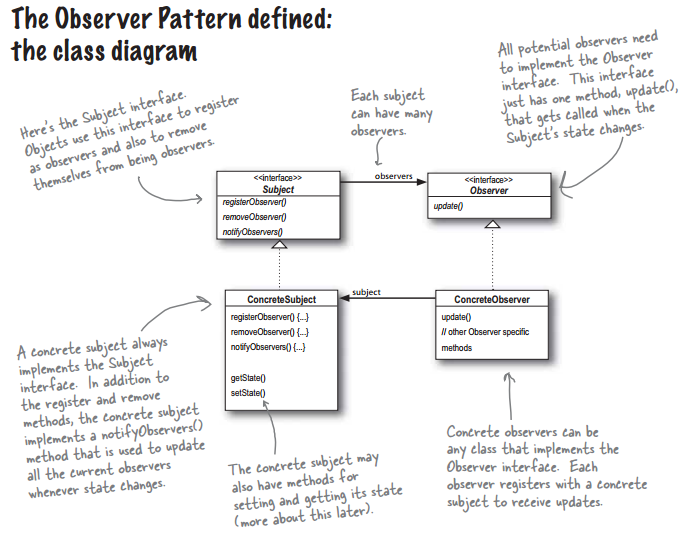
**Chapter 2. The Observer Pattern**



Subject và các Observers định nghĩa thành mối quan hệ one-to-many.

Các Observers phụ thuộc vào subject, khi trạng thái của subject thay đổi, các observer được thông báo. Tùy thuộc vào loại thông báo mà Observer có thể quyết định cập nhật giá trị hay không.

Observer Pattern class diagram sử dụng Object interface và Observer interface:



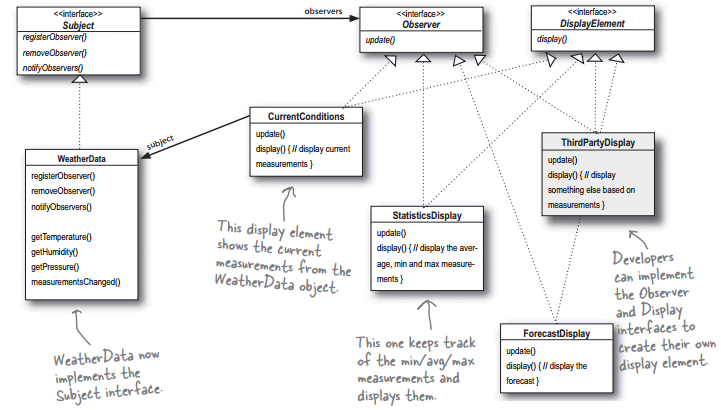
Minh họa về ứng dụng thông báo thời tiết.

Nhiệm vụ của Joe lúc này là thiết kế một chương trình sử dụng WeatherData Object để cập nhật 3 thông tin thời tiết (nhiệt độ, độ ẩm, và áp suất) từ đó hiển thị thông tin:

+ CurrentConditions: tình trạng thời tiết hiện tại.

+ StaticsDisplay: chỉ số thời tiết như nhiệt độ trung bình, nhỏ nhất, lớn nhất.

+ ForecastDisplay: dự báo thời tiết.



Joe bắt tay vào code và xử lý trước với chức năng CurrentConditionsDisplay như sau:

1. Bắt đầu với các interfaces

|  |
| --- |
| **public** **interface** Observer {  **public** **void** update(**float** temp, **float** humidity, **float** pressure);  }  **public** **interface** Subject {  **public** **void** registerObserver(Observer o);  **public** **void** removeObserver(Observer o);  **public** **void** notifyObservers();  }  **public** **interface** DisplayElement {  **public** **void** display();  } |

2. Joe áp dụng interface Subject vào class WeatherData

Anh ta sử dụng một ***ArrayList*** để lưu trữ danh sách các observers.

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** WeatherData **implements** Subject {  **private** ArrayList observers;  **private** **float** temperature;  **private** **float** humidity;  **private** **float** pressure;    **public** WeatherData() {  observers = **new** ArrayList();  }    **public** **void** registerObserver(Observer o) {  observers.add(o);  }    **public** **void** removeObserver(Observer o) {  **int** i = observers.indexOf(o);  **if** (i>=0) {  observers.remove(i);  }  }    **public** **void** notifyObservers() {  **for** (**int** i = 0; i < observers.size(); i++) {  Observer observer = (Observer) observers.get(i);  observer.update(temperature, humidity, pressure);  }  }    **public** **void** measurementChanged() {  notifyObservers();  }    **public** **void** setMeasurements(**float** temperature, **float** humidity, **float** pressure) {  **this**.temperature = temperature;  **this**.humidity = humidity;  **this**.pressure = pressure;  measurementChanged();  }  } |

3. Joe viết class CurrentConditionsDisplay

Class này hiển thị thông tin nhiệu độ và độ ẩm mỗi khi có thay đổi (không cần hiển thị áp suất)

|  |
| --- |
| **public** **class** CurrentConditionsDisplay **implements** Observer, DisplayElement {  **private** **float** temperature;  **private** **float** humidity;  **private** Subject weatherData;    **public** CurrentConditionsDisplay(Subject weatherData) {  **this**.weatherData = weatherData;  weatherData.registerObserver(**this**);  }    **public** **void** update(**float** temperature, **float** humidity, **float** pressure) {  **this**.temperature = temperature;  **this**.humidity = humidity;  display();  }    **public** **void** display() {  System.***out***.println("Current condition: " + temperature +  "C degrees and " + humidity + "% humidity");  }  } |

4. Cuối cùng, Joe viết trạm thời tiết WeatherStation, thực hiện cập nhật các thông số

|  |
| --- |
| **public** **class** WeatherStation {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  WeatherData weatherData = **new** WeatherData();    CurrentConditionsDisplay currentDisplay =  **new** CurrentConditionsDisplay(weatherData);  weatherData.setMeasurements(30, 80, 24.6f);  weatherData.setMeasurements(40, 90, 24.6f);  weatherData.setMeasurements(35, 70, 24.6f);  }  } |

5. Kết quả khi chạy chương trình

|  |
| --- |
| Current condition: 30.0C degrees and 80.0% humidity  Current condition: 40.0C degrees and 90.0% humidity  Current condition: 35.0C degrees and 70.0% humidity |

**Using Java’s built-in Observer Pattern**

Suy nghĩ kĩ hơn, Joe nhận ra với cách code như trên, mỗi lần có sự thay đổi, cả 3 thông số nhiệt độ, độ ẩm và áp suất được gửi đến Observer.Trong khi CurrentConditionsDisplay chỉ cần lấy 2 thông số là nhiệt độ và độ ẩm. Câu hỏi đặt ra là làm thế nào để Observer có thể chủ động lấy các thông tin mà nó cần?

Joe tìm thấy rằng Java hỗ trợ sẵn class ***Observer*** và ***Observable***. Hai class này chứa các API thực hiện nhiệm vụ tương tự các phương thức registerObserver(), removeObserver(), notifyObservers(), etc.

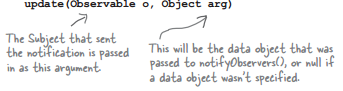
Khi một đối tượng thuộc class ***Observable*** gửi thông báo:

+ Đầu tiên nó phải gọi hàm ***setChanged()*** để báo hiệu rằng trạng thái đối tượng thay đổi.

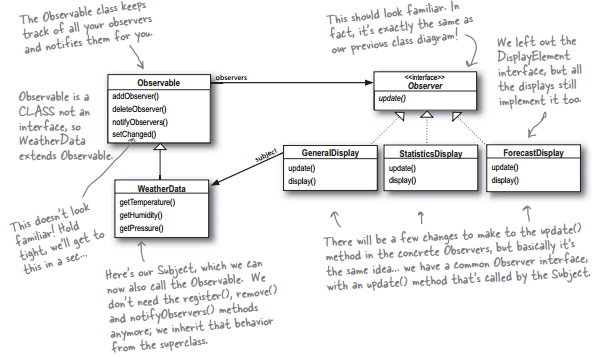
+ Sau đó nó gọi hàm ***notifyObservers()*** hoặc ***notifyObservers(Object arg)***.

Khi một đối tượng thuộc class ***Observer*** nhận thông báo:

+ Thay đổi phương thức update một chút, ***update(Observable o, Object arg).***



Joe thiết kế lại hệ thống áp dụng 2 class trên như sau:



Tiếp theo là coding.

1. Thay đổi Subject WeatherData sử dụng java.util.Observable

|  |
| --- |
| **import** java.util.Observable;  **public** **class** WeatherData **extends** Observable {  **private** **float** temperature;  **private** **float** humidity;  **private** **float** pressure;    **public** WeatherData() { }    **public** **void** measurementChanged() {  setChanged();  notifyObservers();  }    **public** **void** setMeasurements(**float** temperature, **float** humidity, **float** pressure) {  **this**.temperature = temperature;  **this**.humidity = humidity;  **this**.pressure = pressure;  measurementChanged();  }    **public** **float** getTemperature() {  **return** temperature;  }  **public** **float** getHumidity() {  **return** humidity;  }  **public** **float** getPressure() {  **return** pressure;  }  } |

2. Thay đổi Observer CurrentConditionsDisplay sử dụng Observable và Observer.

|  |
| --- |
| **import** java.util.Observable;  **import** java.util.Observer;  **public** **class** CurrentConditionsDisplay **implements** Observer, DisplayElement{  Observable observable;  **private** **float** temperature;  **private** **float** humidity;    **public** CurrentConditionsDisplay(Observable observable) {  **this**.observable = observable;  observable.addObserver(**this**);  }    **public** **void** update(Observable obs, Object arg) {  **if** (obs **instanceof** WeatherData) {  WeatherData weatherData = (WeatherData) obs;  **this**.temperature = weatherData.getTemperature();  **this**.humidity = weatherData.getHumidity();  display();  }  }    **public** **void** display() {  System.***out***.println("Current conditions: " + temperature  + "C degrees and " + humidity + "%humidity");  }  } |

Joe nhận ra một số điểm trừ của Observable:

+ Observable là một class. Giải thích vì sao class là một nhược điểm ở phần kinh nghiệm rút ra.

+ Observable bảo mật các phương thức quan trọng. Ví dụ API setChanged(). Joe không thể tạo một instance của class Observable và kết hợp (compose) nó vào đối tượng của anh ta. Joe không thể gọi phương thức setChanged() theo cách này, anh ta phải kế thừa nó. Điều này vi phạm nguyên tắc thiết kế là ưu tiên composition hơn inheritance.

Một số kinh nghiệm rút ra:

Nguyên tắc thiết kế thứ 4: Strive for loosely coupled designs between objects that interact.

Thiết kế liên kết lỏng cho phép chúng ta xây dựng hệ thống OO mềm dẻo, dễ dàng thay đổi bởi sự phụ thuộc giữa các đối tượng đã được tối thiểu hóa.

1. Lợi ích của Observer Pattern:

+ Thứ nhất, dễ dàng thêm vào một Observer mới hay bỏ đi một Observer trong danh sách ngay lúc Runtime bằng các phương thức registerObserver() và removeObserver().

+ Thứ hai, chúng ta không phải sửa đổi subject để thêm vào một loại Observer mới.

+ Thứ ba, chúng ta có thể tái sử dụng Subject hoặc Observers độc lập.

+ Thứ tư, thay đổi subject hoặc một observer sẽ không ảnh hưởng đến nhau.

+ Thứ năm, với việc khai bao các getter ở Subject (e.g. getTemperature) và truyền Object arg, ta có thể dễ dàng thêm bớt các thông số cần thiết ví dụ như thông số windpower mà không cần sửa đổi các Observer hiện tại.

2. Tại sao thiết kế Observer phải là một Interface. Bởi vì Java không hỗ trợ đa kế thừa (multi-inheritance) nên nếu thiết kế Observer là một class thì không thể thêm vào danh sách Observer một Class-có-sẵn-và-đã-kế-thừa-từ-một-superclass. Nó không thể extend thêm Oberser class được nữa.

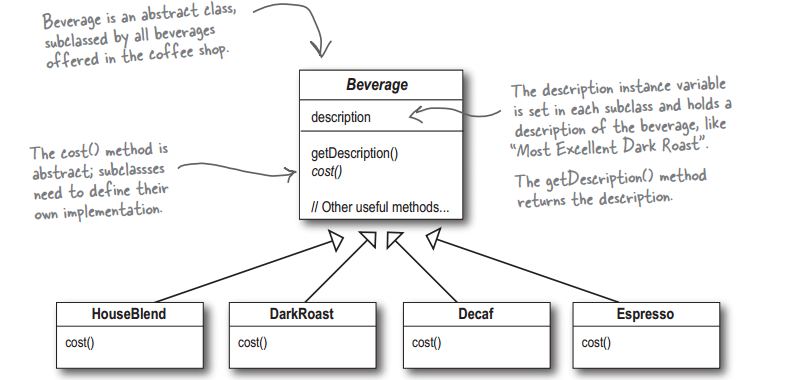
3. Áp dụng ở trong Android Framework:

+ Broadcast Receiver. Ví dụ LowBattery Broadcast.

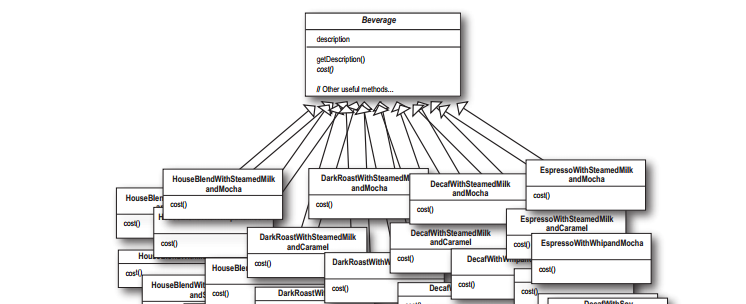
+ Listener. Ví dụ class SensorActivity implements SensorEventListener. Chú ý, còn có một abstract class của SensorActivity nhằm nhiệm vụ cung cấp các API cho upper layers. Số lượng API là hạn chế hơn so với API cho framework.

**Chapter 3. The Decorator Pattern**

Joe nhận được yêu cầu từ công ty Starbuzz Coffee, thiết kế hệ thống tính giá tiền cho các đơn hàng gọi đồ uống. Ban đầu số lượng sản phẩm của Starbuzz còn ít nên họ tự thiết kế như sau:

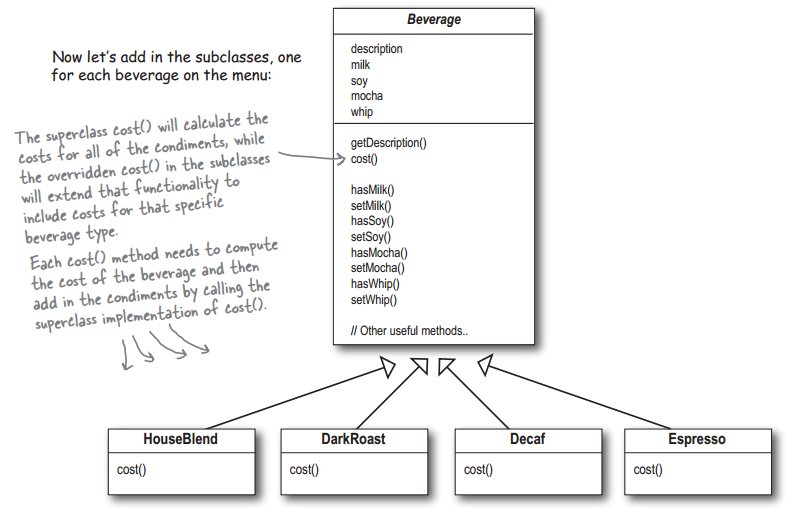


Tuy nhiên, Starbuzz ngày càng nghĩ ra nhiều loại đồ uống, đồng thời thêm thắt các gia vị như milk, soy, mocha. Khách hàng có thể tùy ý chọn loại gia vị vào đồ uống của mình. Mỗi loại gia vị lại có một giá thành khác nhau, dẫn đến việc tính toán không còn đơn giản. Số lượng class gia tăng chóng mặt:



Starbuzz tự nghĩ cách cải tiến cho mô hình này. Họ thêm vào superclass Beverage các biến để cho biết milk, soy, mocha và các phương thức tương ứng hasMilk(), hasSoy(), hasMocha() biểu thị có các loại gia vị đó hay không.

Đồng thời giá của mỗi loại gia vị được lưu bằng các thuộc tính milkCost, soyCost, MochaCost.



Code tính toán giá đồ uống:

|  |
| --- |
| **public** **class** Beverage {  **public** **float** cost() {  **float** condimentCost = 0.0;  **if** (hasMilk()) {  condimentCost += milkCost;  }  **if** (hasSoy()) {  condimentCost += soyCost;  }  **if** (hasMocha()) {  condimentCost += mochaCost;  }  **return** condimentCost;  }  } |
| **public** **class** DarkRoast **extends** Beverage {  **public** DarkRoast() {  description = "Most Excellent Dark Roast";  }    **public** **float** cost() {  **return** 1.99 + **super**.cost();  }  } |

Joe nhanh chóng nhận ra nhược điểm của thiết kế này:

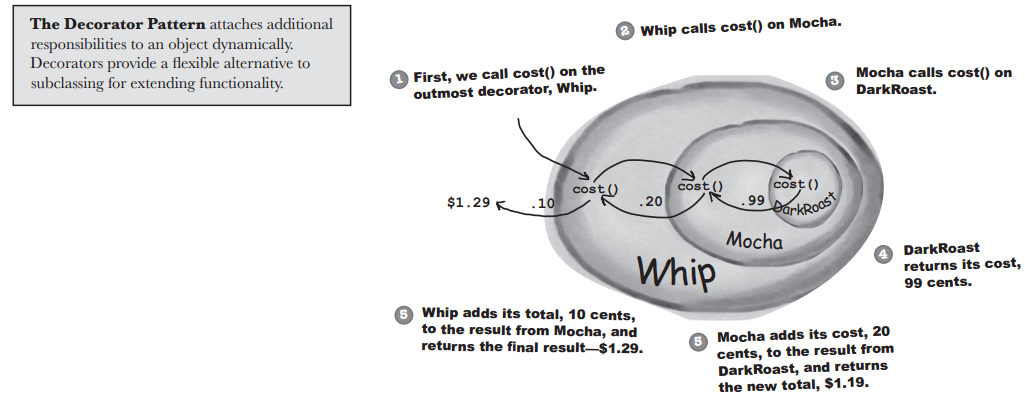
+ Nếu có thêm một gia vị mới, Starbuzz sẽ phải sửa phương thức cost() ở superclass Beverage.

+ Các loại đồ uống như DarkRoast kế thừa từ superclass Beverage nên mặc định nhận tất thuộc tính và phương thức liên quan đến các loại gia vị, mặc dầu đó là loại gia vị không cần thiết.

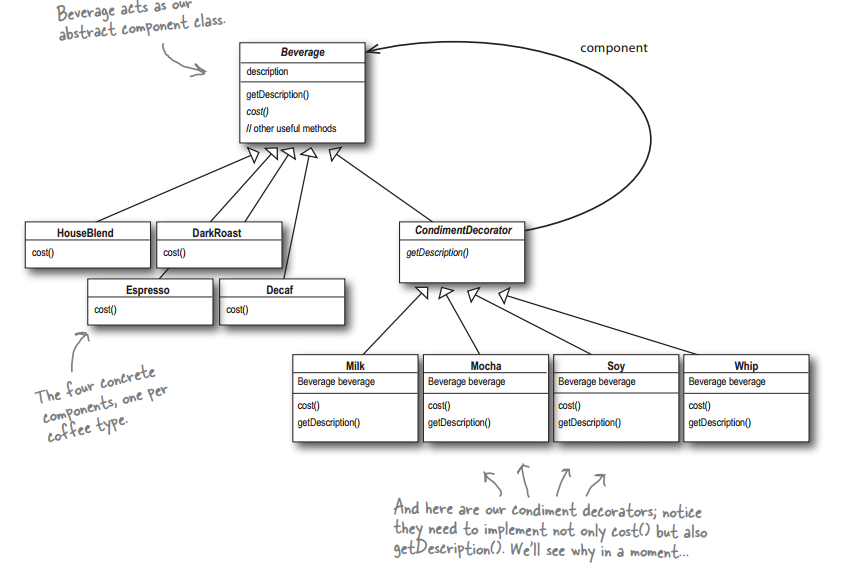
+ Nếu khách hàng muốn 2 lần Mocha thì code sẽ phải thiết kế lại bước tính toán.

Nguyên tắc thiết kế thứ 5. Classes should be open for extension, but close for modification.

Joe nghĩ tới **Decorator Pattern**



Anh ta thiết kế lại hệ thống cho Starbuzz với Decorator Pattern



Các loại gia vị không kế thừa trực tiếp từ superclass Beverage mà thông qua một superclass trung gian ***ComdimentDecorator***, thể hiện các đặc tính chung của gia vị.

Joe bắt tay vào code

1. Abstract class Beverage và CondimentDecorator

|  |
| --- |
| **public** **abstract** **class** Beverage {  String description = "Unknown Beverage";    **public** String getDescription() {  **return** description;  }    **public** **abstract** **double** cost();  } |
| **public** **abstract** **class** CondimentDecorator **extends** Beverage {  **public** **abstract** String getDescription();  } |

2. Lớp đồ uống kế thừa từ abstract class Beverage: Espresso, DarkRoast, HouseBlend, etc.

|  |
| --- |
| **public** **class** Espresso **extends** Beverage {  **public** Espresso() {  description = "Espresso";  }    **public** **double** cost() {  **return** 1.99;  }  } |
| **public** **class** DarkRoast **extends** Beverage {  **public** DarkRoast() {  description = "Dark Roast Coffee";  }    **public** **double** cost() {  **return** 1.49;  }  } |

3. Lớp gia vị kế thừa từ abstract class CondimentDecorator: Mocha, Whip

|  |
| --- |
| **public** **class** Mocha **extends** CondimentDecorator {  Beverage beverage;    **public** Mocha(Beverage beverage) {  **this**.beverage = beverage;  }    **public** String getDescription() {  **return** beverage.getDescription() + ", Mocha";  }    **public** **double** cost() {  **return** .20 + beverage.cost();  }  } |
| **public** **class** Whip **extends** CondimentDecorator {  Beverage beverage;    **public** Whip(Beverage beverage) {  **this**.beverage = beverage;  }    **public** String getDescription() {  **return** beverage.getDescription() + " , Whip";  }    **public** **double** cost() {  **return** 0.4 + beverage.cost();  }  } |

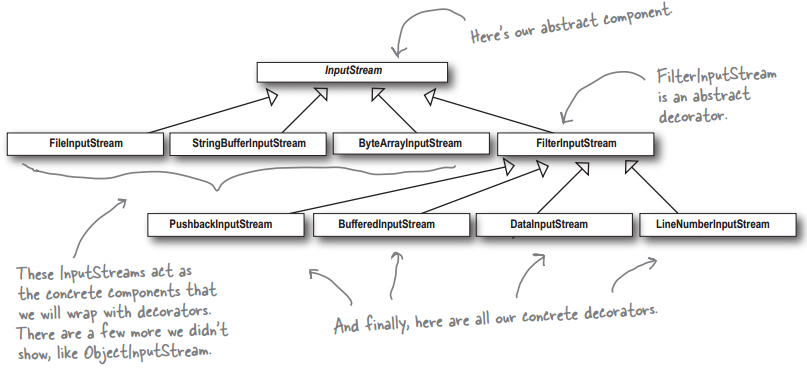
4. Order được thực hiện ở main class StarbuzzCoffeeStore

|  |
| --- |
| **public** **class** StarbuzzCoffeeStore {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  Beverage beverage = **new** Espresso();  System.***out***.println(beverage.getDescription() + " $" + beverage.cost());    beverage = **new** Mocha(beverage);  beverage = **new** Whip(beverage);  beverage = **new** Mocha(beverage);  System.***out***.println(beverage.getDescription() + " $" + beverage.cost());    }  } |

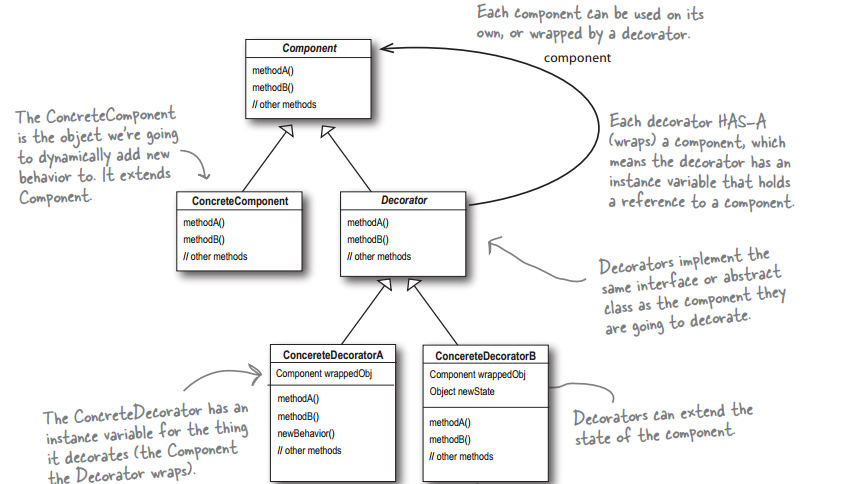
5. Kết quả khi chạy chương trình

|  |
| --- |
| Espresso $1.99  Espresso, Mocha , Whip, Mocha $2.79 |

**Java IO decorator**



Mô hình tổng quát của Decorator Pattern



Kinh nghiệm rút ra:

1. Ưu điểm của Decorator Pattern:

+ Tính đóng gói: Khi cần thêm loại đồ uống hoặc gia vị hoặc thay đổi giá cả thì không ảnh hưởng tới các lớp hiện tại.

+ Dễ dàng thêm bớt gia vị vào đối tượng đồ uống ngay lúc Runtime.

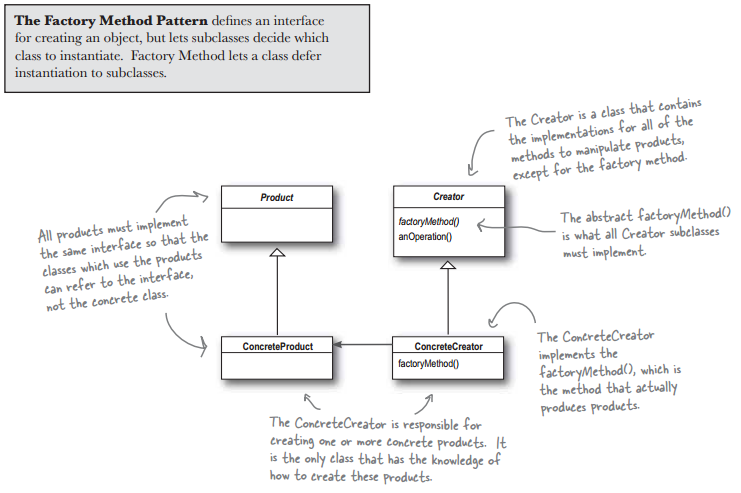
2. Áp dụng Decorator ở Android Framework: abstract superclass View.

+ Text View là ConcreteComponent.

+ ViewGroup Decorator.

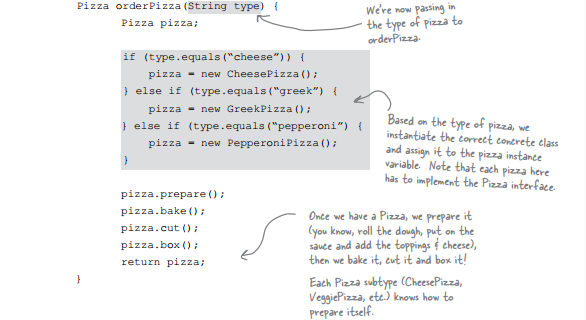
+ ConcreteDecorators: Linear Layout, etc.

**Chapter 4. The Factory Pattern**



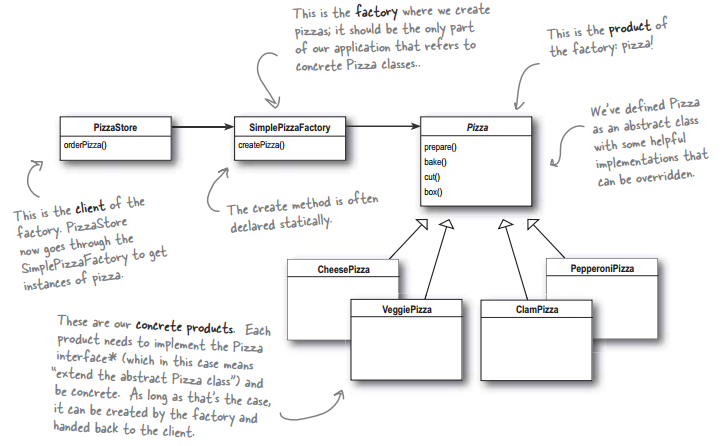
Lần này Joe nhận được yêu cầu từ cửa hàng Pizza. Họ cho biết các loại Pizza có 4 bước sản xuất: prepare, bake, cut, box. Tuy nhiên, mỗi loại lại có thuộc tính riêng: name, dough, sauce và toppings.

Ban đầu chỉ có 1 số lượng ít loại pizza, lập trình viên cũ đã code như sau:



Có thể nhận thấy ngay là nếu thêm hay loại bỏ một loại Pizza, lập trình viên phải sửa code ở đoạn if else. Điều này vi phạm nguyên tắc số 5 (Classes should be open for extension, but close for modification).

Một phương pháp trực quan nhất để cải tiến là di chuyển đoạn code khởi tạo loại Pizza vào một class mới, gọi là ***SimplePizzaFactory***.



Joe bắt tay code thử với mô hình trên

1. Abstract class Pizza và các sub class Pizza type (CheesePizza, ClamPizza)

|  |
| --- |
| **import** java.util.ArrayList;  **public** **abstract** **class** Pizza {  String name;  String dough;  String sauce;  ArrayList toppings = **new** ArrayList();    **void** prepare() {  System.***out***.println("Preparing " + name);  System.***out***.println("Tossing dough...");  System.***out***.println("Adding sauce...");  System.***out***.println("Adding toppings:");  **for** (**int** i = 0; i < toppings.size(); i++) {  System.***out***.println(" " + toppings.get(i));  }  }  **void** bake() {  System.***out***.println("Bake for 25 minutes at 350");  }  **void** cut() {  System.***out***.println("Cutting the pizza into diagonal slices");  }  **void** box() {  System.***out***.println("Place pizza in official PizzaStore box");  }  **public** String getName() {  **return** name;  }  } |
| **public** **class** CheesePizza **extends** Pizza {  **public** CheesePizza() {  name = "Cheese Pizza";  dough = "Thin Crust Dough";  sauce = "Mariana Sauce";  toppings.add("Grated Reggiano Cheese");  }  } |
| **public** **class** ClamPizza **extends** Pizza {  **public** ClamPizza() {  name = "Clam Pizza";  dough = "Thick Crust Dough";  sauce = "Tomato Sauce";  toppings.add("Fresh Clam");  toppings.add("Dry Onion");  }  } |

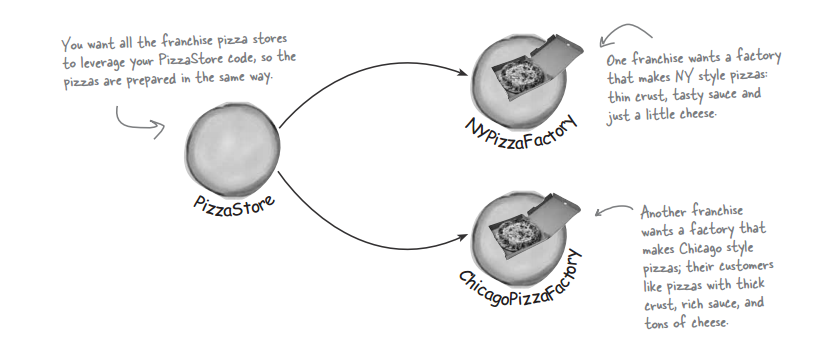
2. Class PizzaStore và class SimplePizzaFactory

|  |
| --- |
| **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;  }  } |
| **public** **class** SimplePizzaFactory {    **public** Pizza createPizza(String type) {  Pizza pizza = **null**;    **if** (type.equals("cheese")) {  pizza = **new** CheesePizza();  } **else** **if** (type.equals("clam")) {  pizza = **new** ClamPizza();  }  **return** pizza;  }  } |

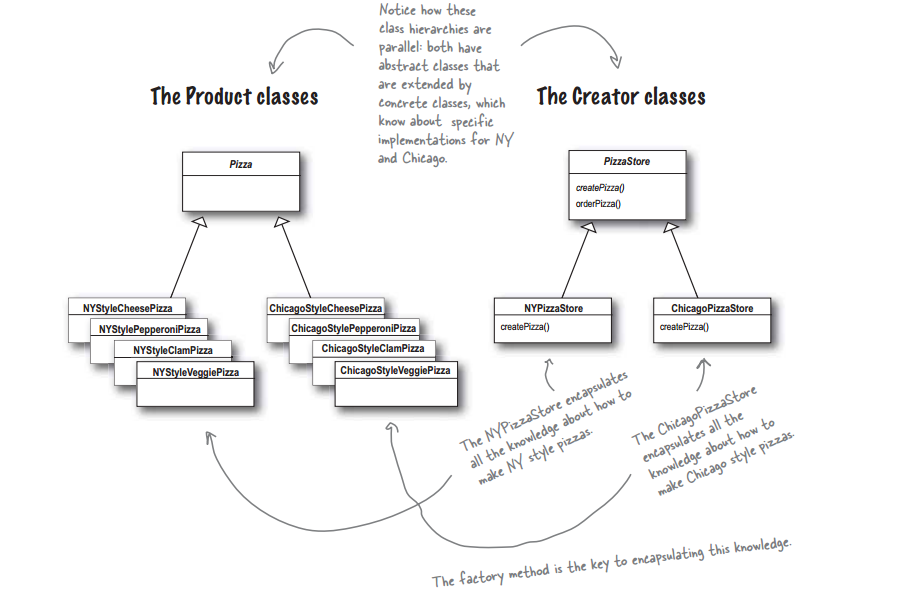
3. Chương trình main PizzaTestDrive và kết quả chạy

|  |
| --- |
| **public** **class** PizzaTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  SimplePizzaFactory factory = **new** SimplePizzaFactory();  PizzaStore pizzaStore = **new** PizzaStore(factory);    Pizza pizza = pizzaStore.orderPizza("cheese");  System.***out***.println("Tung ordered a " + pizza.getName());  System.***out***.println();    pizza = pizzaStore.orderPizza("clam");  System.***out***.println("Th ordered a " + pizza.getName());  }  } |
| Preparing Cheese Pizza  Tossing dough...  Adding sauce...  Adding toppings:  Grated Reggiano Cheese  Bake for 25 minutes at 350  Cutting the pizza into diagonal slices  Place pizza in official PizzaStore box  Tung ordered a Cheese Pizza  Preparing Clam Pizza  Tossing dough...  Adding sauce...  Adding toppings:  Fresh Clam  Dry Onion  Bake for 25 minutes at 350  Cutting the pizza into diagonal slices  Place pizza in official PizzaStore box  Th ordered a Clam Pizza |

Bài toán mới xuất hiện, do cửa hàng Pizza làm ăn phát đạt nên họ mở chi nhánh ở NY và Chicago. Do khẩu vị mỗi vùng là khác nhau nên từ mỗi loại Pizza ban đầu, cần phải tách thành 2 loại Pizza cho mỗi vùng. Ví dụ *CheesePizza* tách thành *NYStyleCheesePizza* và *ChicagoStyleCheesePizza*.



Bây giờ Joe phát triển SimplePizzaFactory thành 2 class riêng biệt NYPizzaStore và ChicagoPizzaStore. Thiết kế theo Factory Pattern gồm Creator classes và Product classes:



Tiếp theo Joe thay đổi code một chút so với SimplePizzaFactory:

1. Giữ nguyên abstract class Pizza; Thay thế mỗi class Pizza cũ (CheesePizza, ClamPizza) thành 2 class NYStyle và ChicagoStyle

|  |
| --- |
| **public** **class** NYStyleCheesePizza **extends** Pizza {  **public** NYStyleCheesePizza() {  name = "NY Style Sauce and Cheese Pizza";  dough = "Thin Crust Dough";  sauce = "Mariana Sauce";  toppings.add("Grated Reggiano Cheese");  }  } |
| **public** **class** NYStyleClamPizza **extends** Pizza {  **public** NYStyleClamPizza() {  name = "NY Style Clam Pizza";  dough = "No Dough";  sauce = "Orange Sauce";  toppings.add("Fresh Clam");  }  } |
| **public** **class** ChicagoStyleCheesePizza **extends** Pizza {  **public** ChicagoStyleCheesePizza() {  name = "Chicago Style Deep Dish Cheese Pizza";  dough = "Extra Thick Crust Dough";  sauce = "Plum Tomato Sauce";  toppings.add("Shredded Mozzarella Cheese");  }  **void** cut() {  System.***out***.println("Cutting the pizza into square slices");  }  } |
| **public** **class** ChicagoStyleClamPizza **extends** Pizza {  **public** ChicagoStyleClamPizza() {  name = "Chicago Style Clam Pizza";  dough = "Thich Crust Dough";  sauce = "Coconut Sauce";  toppings.add("Cold Clam");  }  } |

2. Chuyển class PizzaStore thành abstract và thay thế class SimplePizzaFactory bằng class NYStylePizzaStore và ChicagoStylePizzaStore.

|  |
| --- |
| **public** **abstract** **class** PizzaStore {  **abstract** Pizza createPizza(String type);    **public** Pizza orderPizza(String type) {  Pizza pizza;    pizza = createPizza(type);    pizza.prepare();  pizza.bake();  pizza.cut();  pizza.box();  **return** pizza;  }  } |
| **public** **class** NYStylePizzaStore **extends** PizzaStore {  Pizza createPizza(String item) {  **if** (item.equals("cheese")) {  **return** **new** NYStyleCheesePizza();  } **else** **if** (item.equals("clam")) {  **return** **new** NYStyleClamPizza();  } **else** **return** **null**;  }  } |
| **public** **class** ChicagoStylePizzaStore **extends** PizzaStore {  Pizza createPizza(String item) {  **if** (item.equals("cheese")) {  **return** **new** ChicagoStyleCheesePizza();  } **else** **if** (item.equals("clam")) {  **return** **new** ChicagoStyleClamPizza();  } **else** **return** **null**;  }  } |

3. Chương trình main PizzaTestDrive và kết quả chạy

|  |
| --- |
| **public** **class** PizzaTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  PizzaStore nyStore = **new** NYStylePizzaStore();  PizzaStore chicagoStore = **new** ChicagoStylePizzaStore();    Pizza pizza = nyStore.orderPizza("cheese");  System.***out***.println("Tung ordered a " + pizza.getName());  System.***out***.println();    pizza = chicagoStore.orderPizza("cheese");  System.***out***.println("Th ordered a " + pizza.getName());  }  } |
| Preparing NY Style Sauce and Cheese Pizza  Tossing dough...  Adding sauce...  Adding toppings:  Grated Reggiano Cheese  Bake for 25 minutes at 350  Cutting the pizza into diagonal slices  Place pizza in official PizzaStore box  Tung ordered a NY Style Sauce and Cheese Pizza  Preparing Chicago Style Deep Dish Cheese Pizza  Tossing dough...  Adding sauce...  Adding toppings:  Shredded Mozzarella Cheese  Bake for 25 minutes at 350  Cutting the pizza into square slices  Place pizza in official PizzaStore box  Th ordered a Chicago Style Deep Dish Cheese Pizza |

**Dependency Inversion Principle**

Nguyên tắc thiết kế thứ 6: Depend upon abstractions. Do not depend on concrete classes.

+ High-level components shouldn’t depend on low-level components.

+ Make high and low level components depend on abstractions.

**Hướng dẫn hỗ trợ áp dụng Inversion Principle**

+ No class variable should hold a reference to a concrete class (Nếu bạn muốn “new”, xây dựng một factory)

+ No class should derive from a concrete class (Derive from an abstraction – an abstract class or interface)

+ No method should override an implemented method any of its base classes. (Nếu bạn override một phương thức đã được định nghĩa, thì super class chưa phải là một abstraction tốt).

**Building the ingredient factories – Abstract Factory Pattern**

Joe nhận thấy rằng, với cùng một loại bánh Pizza, kiểu NY sẽ khác kiểu Chicago tuy có cùng thành phần gia vị (dough, sauce, cheese, etc), tuy nhiên lại khác nhau ở loại gia vị mỗi thành phần. Ví dụ cùng là dough nhưng NYStyle sử dụng *Thin Crust Dough* trong khi ChicagoStyle sử dụng *Thick Crust Dough*.

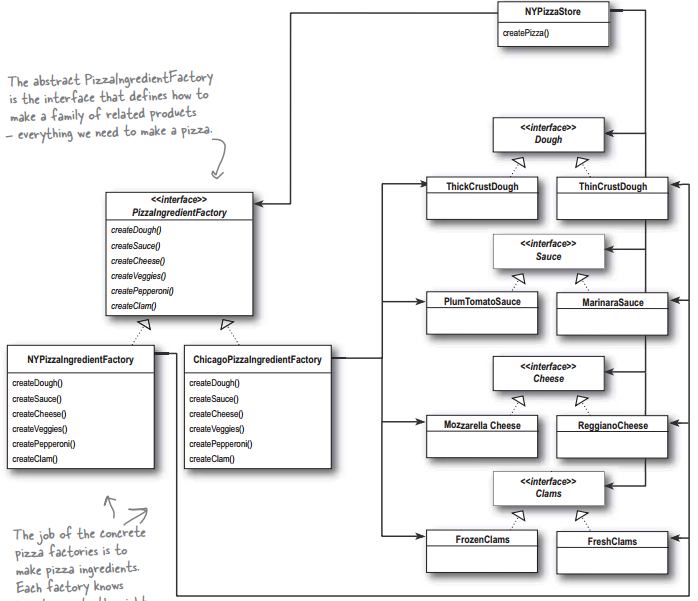
Anh ta thiết kế một nhà máy nhằm cung cấp các loại gia vị khác nhau cho từng vùng.



Joe tạo PizzaIngredientFactory

+ Xây dựng một nhà máy cho mỗi vùng (NYPizzaIngredientFactory và ChicagoPizzaIngredientFactory)

+ Xây dựng một tập hợp các lớp gia vị có thể sử dụng bởi bất kỳ nhà máy nào.



1. Interface PizzaIngredientFactory và 2 concrete class NY, Chicago PizzaIngredientFactory

Mỗi vùng lại có một kiểu khác nhau trong cùng một loại gia vị.

|  |
| --- |
| **public** **class** NYPizzaIngredientFactory **implements** PizzaIngredientFactory {  **public** Dough createDough() {  **return** **new** DoughThinCrust();  }    **public** Sauce createSauce() {  **return** **new** SauceMariana();  }  **public** Cheese createCheese() {  **return** **new** CheeseReggiano();  }  } |
| **public** **class** ChicagoPizzaIngredientFactory **implements** PizzaIngredientFactory {  **public** Dough createDough() {  **return** **new** DoughThickCrust();  }    **public** Sauce createSauce() {  **return** **new** SaucePlumTomato();  }  **public** Cheese createCheese() {  **return** **new** CheeseMorrazella();  }  } |

2. Thêm các interface và concrete class gia vị (Dough, Sauce, Cheese)

|  |
| --- |
| **public** **interface** Dough {  } |
| **public** **class** DoughThinCrust **implements** Dough {  **public** String toString() {  **return** "Thin Crust Dough";  }  } |
| **public** **class** DoughThickCrust **implements** Dough {  **public** String toString() {  **return** "Thick Crust Dough";  }  } |

3. Thay đổi abstract class Pizza và các concrete class (CheesePizza, ClamPizza) để sử dụng factory ingredient

|  |
| --- |
| **public** **abstract** **class** Pizza {  String name;  Dough dough;  Sauce sauce;  Cheese cheese;    **abstract** **void** prepare();    **void** bake() {  System.***out***.println("Bake for 25 minutes at 350");  }    **void** cut() {  System.***out***.println("Cutting the pizza into diagonal slices");  }    **void** box() {  System.***out***.println("Place pizza in official PizzaStore box");  }    **void** setName(String name) {  **this**.name = name;  }    String getName() {  **return** name;  }  } |
| **public** **class** CheesePizza **extends** Pizza {  PizzaIngredientFactory ingredientFactory;    **public** CheesePizza(PizzaIngredientFactory ingredientFactory) {  **this**.ingredientFactory = ingredientFactory;  }    **void** prepare() {  System.***out***.println("Preparing " + name);  dough = ingredientFactory.createDough();  sauce = ingredientFactory.createSauce();  cheese = ingredientFactory.createCheese();  System.***out***.println(" Dough: " + dough.toString() + " Sauce: " + sauce.toString() +  " Cheese: " + cheese.toString());  }  } |
| **public** **class** ClamPizza **extends** Pizza {  PizzaIngredientFactory ingredientFactory;    **public** ClamPizza(PizzaIngredientFactory ingredientFactory) {  **this**.ingredientFactory = ingredientFactory;  }    **void** prepare() {  System.***out***.println("Preparing " + name);  dough = ingredientFactory.createDough();  sauce = ingredientFactory.createSauce();  cheese = ingredientFactory.createCheese();  System.***out***.println(" Dough: " + dough.toString() + " Sauce: " + sauce.toString() +  " Cheese: " + cheese.toString());  }  } |

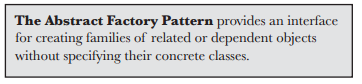
4. Giữ nguyên abstract class PizzaStore và chỉnh sửa concrete class (NYPizzaStore)

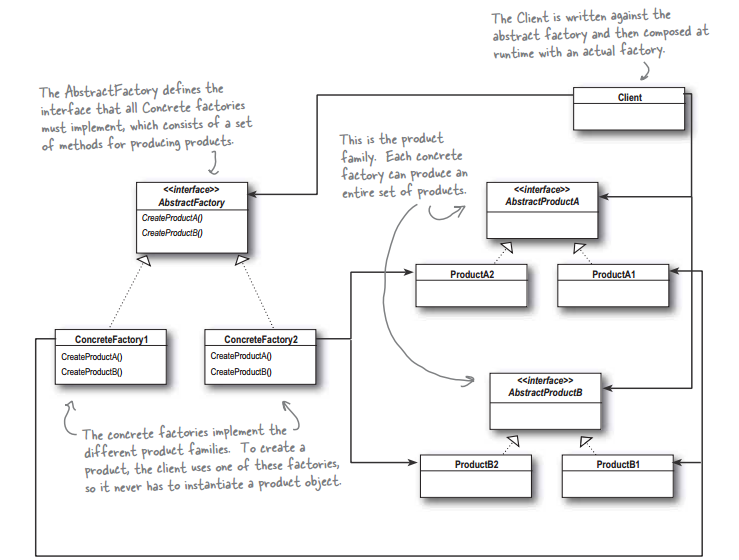
|  |
| --- |
| **public** **class** NYPizzaStore **extends** PizzaStore {  **protected** Pizza createPizza(String item) {  Pizza pizza = **null**;  PizzaIngredientFactory ingredientFactory = **new** NYPizzaIngredientFactory();    **if** (item.equals("cheese")) {  pizza = **new** CheesePizza(ingredientFactory);  pizza.setName("NY Style Cheese Pizza");  } **else** **if** (item.equals("clam")) {  pizza = **new** ClamPizza(ingredientFactory);  pizza.setName("NY Style Clam Pizza");  }  **return** pizza;  }  } |

5. Kết quả khi chạy PizzaTestDrive

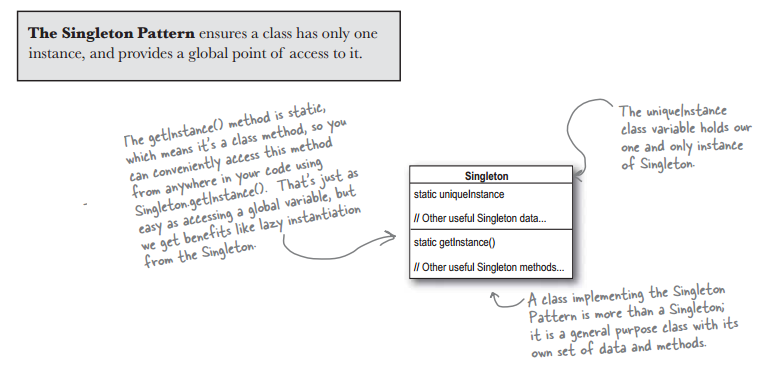
|  |
| --- |
| **public** **class** PizzaTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  PizzaStore nyStore = **new** NYPizzaStore();  PizzaStore chicagoStore = **new** ChicagoPizzaStore();    Pizza pizza = nyStore.orderPizza("cheese");  System.***out***.println("Tung ordered a " + pizza.getName());  System.***out***.println(pizza.dough.toString());  System.***out***.println();    pizza = chicagoStore.orderPizza("cheese");  System.***out***.println("Th ordered a " + pizza.getName());  System.***out***.println(pizza.dough.toString());  }  } |
| Preparing NY Style Cheese Pizza  Dough: Thin Crust Dough Sauce: Mariana Sauce Cheese: Reggiano Cheese  Bake for 25 minutes at 350  Cutting the pizza into diagonal slices  Place pizza in official PizzaStore box  Tung ordered a NY Style Cheese Pizza  Thin Crust Dough  Preparing Chicago Style Cheese Pizza  Dough: Thick Crust Dough Sauce: Plum Tomato Sauce Cheese: Mozzarella Cheese  Bake for 25 minutes at 350  Cutting the pizza into diagonal slices  Place pizza in official PizzaStore box  Th ordered a Chicago Style Cheese Pizza  Thick Crust Dough |

Thực chất để xây dựng PizzaIngredientFactory, Joe đã áp dụng **Abstract Factory Pattern**





**Chapter 5. The Singleton Pattern**



Về cơ bản, Singleton Pattern có mục đích đảm bảo rằng một object chỉ được khởi tạo duy nhất 1 lần. (Tương tự như biến toàn cục).

Có 3 cách để implement Singleton Pattern.

Thứ nhất, sử dụng synchronize.

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *uniqueInstance*;    **private** Singleton() {};    **public** **static** **synchronized** Singleton getInstance() {  **if** (*uniqueInstance* == **null**) {  *uniqueInstance* = **new** Singleton();  }  **return** *uniqueInstance*;  }  } |

Nhược điểm của cách này là làm giảm hiệu suất của chương trình. Máy luôn phải synchronized mỗi khi gọi phương thức getInstance().

Thứ hai, eagerly created instance - chủ động khởi tạo instance ngay khi class được load.

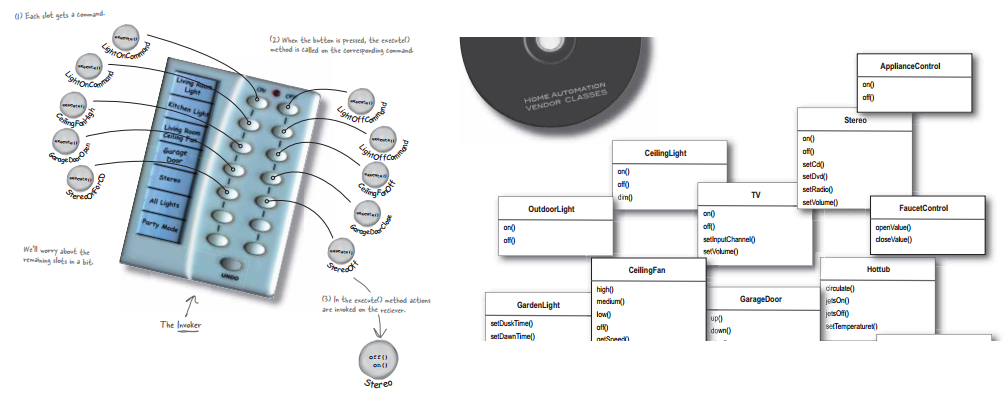
|  |
| --- |
| **public** **class** Singleton2 {  **private** **static** Singleton2 *uniqueInstance* = **new** Singleton2();    **private** Singleton2() {}    **public** **static** Singleton2 getInstance() {  **return** *uniqueInstance*;  }  } |

Thứ ba, double-checked looking – đầu tiên kiểm tra instance đã được tạo hay chưa. Nếu chưa thì sử dụng synchronized. Như vậy chỉ synchronized một lần đầu tiên.

|  |
| --- |
| **public** **class** Singleton3 {  **private** **volatile** **static** Singleton3 *uniqueInstance*;    **private** Singleton3() {}    **public** **static** Singleton3 getInstance() {  **if** (*uniqueInstance* == **null**) {  **synchronized** (Singleton3.**class**) {  **if** (*uniqueInstance* == **null**) {  *uniqueInstance* = **new** Singleton3();  }  }  }  **return** *uniqueInstance*;  }  } |

**Chapter 6. The Command Pattern**

Hôm nay, lập trình viên Joe nhận được yêu cầu từ công ty smart-home. Họ có 1 bảng điều khiển gồm nhiều slot. Mỗi slot có 2 chức năng điều khiển là On hoặc Off một thiết bị nào đó trong nhà, ví dụ như TV, đèn gara, stereo, etc.



Đầu vào: các class có sẵn từ vendor của các thiết bị TV, Light, Stereo, etc.

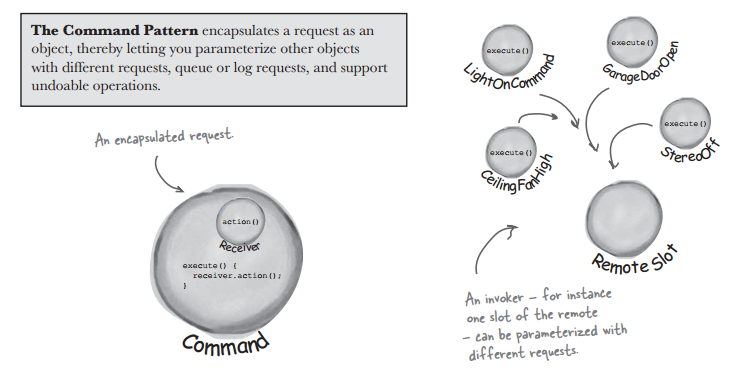
Cần thiết kế:

+ Các class On/Off Command cho mỗi thiết bị, sử dụng các hàm mà vendor cung cấp sẵn.

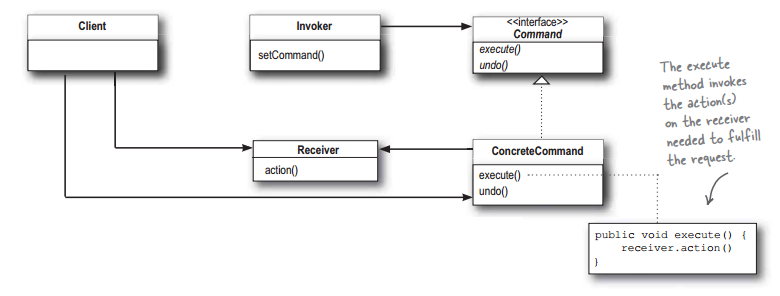
+ Thiết kế RemoteControl – cho phép setCommand vào slot, gọi phương thức xử lý khi ấn On, Off, Undo.

+ Main Remote Client.

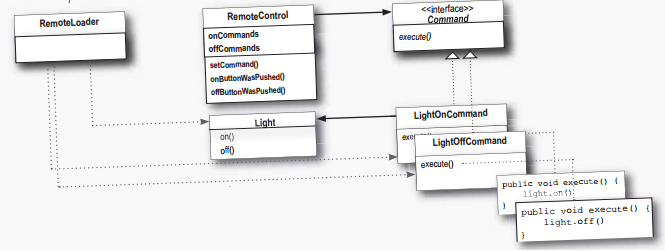
Để hoàn thành những yêu cầu trên, Joe đã tham khảo Command Pattern.



Sơ đồ UML tổng quát của Command Pattern



Joe áp dụng vào bài toán của mình như sau



Joe bắt tay vào code

1. Các class vendor cung cấp cho thiết bị

|  |
| --- |
| **public** **class** Light {  String name;  **public** Light(String name) {  **this**.name = name;  }    **public** **void** on() {  System.***out***.println(name + " Light is on");  }    **public** **void** off() {  System.***out***.println(name + " Light is off");  }  } |
| **public** **class** GarageDoor {  **public** GarageDoor() {}    **public** **void** up() {  System.***out***.println("Garage door moves up");  }    **public** **void** down() {  System.***out***.println("Garage door moves down");  }    **public** **void** stop() {  System.***out***.println("Garage door is stop");  }  } |

2. Interface Commands và các On/Off class tương ứng với mỗi thiết bị

|  |
| --- |
| **public** **interface** Command {  **public** **void** execute();  **public** **void** undo();  } |
| **pub**lic **class** LightOnCommand **implements** Command {  Light light;    **public** LightOnCommand(Light light) {  **this**.light = light;  }    **public** **void** execute() {  light.on();  }    **public** **void** undo() {  light.off();  }  } |
| **public** **class** GarageDoorOpenCommand **implements** Command {  GarageDoor garageDoor;    **public** GarageDoorOpenCommand(GarageDoor garageDoor) {  **this**.garageDoor = garageDoor;  }    **public** **void** execute() {  garageDoor.up();  garageDoor.stop();  }    **public** **void** undo() {  garageDoor.down();  garageDoor.stop();  }  } |

3. Class RemoteControl : setCommands, on/off/undoButtonWasPushed.

|  |
| --- |
| **public** **class** RemoteControl {  Command[] onCommands;  Command[] offCommands;  Command undoCommand;    **public** RemoteControl() {  onCommands = **new** Command[7];  offCommands = **new** Command[7];    Command noCommand = **new** NoCommand();  **for** (**int** i = 0; i < 7; i++) {  onCommands[i] = noCommand;  offCommands[i] = noCommand;  }  undoCommand = noCommand;  }    **public** **void** setCommand(**int** slot, Command onCommand, Command offCommand) {  onCommands[slot] = onCommand;  offCommands[slot] = offCommand;  }    **public** **void** onButtonWasPushed(**int** slot) {  onCommands[slot].execute();  undoCommand = onCommands[slot];  }    **public** **void** OffButtonWasPushed(**int** slot) {  offCommands[slot].execute();  undoCommand = offCommands[slot];  }    **public** **void** undoButtonWasPushed() {  undoCommand.undo();  }    **public** String toString() {  StringBuffer stringBuffer = **new** StringBuffer();  stringBuffer.append("\n------ Remote Control ------\n");  **for** (**int** i = 0; i < onCommands.length; i++) {  stringBuffer.append("[slot " + i + "] " + onCommands[i].getClass().getName()  + " " + offCommands[i].getClass().getName() + "\n");  }  **return** stringBuffer.toString();  }  } |

4. Main class RemoteClient và kết quả

|  |
| --- |
| **public** **class** RemoteClient {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  RemoteControl remoteControl = **new** RemoteControl();    Light livingRoomLight = **new** Light("Living Room");  Light bedRoomLight = **new** Light("bedRoomLight");  GarageDoor garageDoor = **new** GarageDoor();    LightOnCommand livingRoomLightOn = **new** LightOnCommand(livingRoomLight);  LightOffCommand livingRoomLightOff = **new** LightOffCommand(livingRoomLight);  LightOnCommand bedRoomLightOn = **new** LightOnCommand(bedRoomLight);  LightOffCommand bedRoomLightOff = **new** LightOffCommand(bedRoomLight);  GarageDoorOpenCommand garageDoorOpen = **new** GarageDoorOpenCommand(garageDoor);  GarageDoorCloseCommand garageDoorClose = **new** GarageDoorCloseCommand(garageDoor);    remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);  remoteControl.setCommand(1, bedRoomLightOn, bedRoomLightOff);  remoteControl.setCommand(2, garageDoorOpen, garageDoorClose);    System.***out***.println(remoteControl);    remoteControl.onButtonWasPushed(0);  remoteControl.OffButtonWasPushed(0);  remoteControl.undoButtonWasPushed();  remoteControl.onButtonWasPushed(2);  remoteControl.OffButtonWasPushed(2);  remoteControl.onButtonWasPushed(3);  }  } |
| ------ Remote Control ------  [slot 0] RemoteControl.LightOnCommand RemoteControl.LightOffCommand  [slot 1] RemoteControl.LightOnCommand RemoteControl.LightOffCommand  [slot 2] RemoteControl.GarageDoorOpenCommand RemoteControl.GarageDoorCloseCommand  [slot 3] RemoteControl.NoCommand RemoteControl.NoCommand  [slot 4] RemoteControl.NoCommand RemoteControl.NoCommand  [slot 5] RemoteControl.NoCommand RemoteControl.NoCommand  [slot 6] RemoteControl.NoCommand RemoteControl.NoCommand  Living Room Light is on  Living Room Light is off  Living Room Light is on  Garage door moves up  Garage door is stop  Garage door moves down  Garage door is stop  There is no command |

**Party Mode**

Sau khi nhận được chương trình của Joe, vị khách hàng “khó tính” phản hồi thêm yêu cầu cải tiển. Họ muốn có một slot dành cho Party, khi ấn nút On thì một chuỗi các thiết bị được bật lên (đèn sáng, cửa gara mở, TV bật); ngược lại khi ấn nút Off thì các thiết bị đó được tắt đi. Thay vì phải ấn nhiều nút như hiện nay, họ muốn gói gọn trong 1 hành động. Joe đã suy nghĩ và đi đến sử dụng Macro Commands.

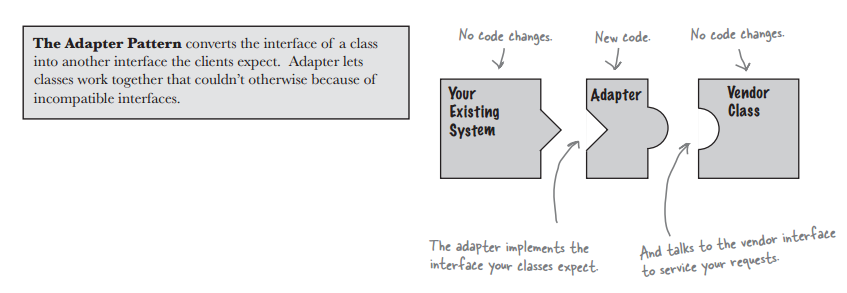
1. MacroCommand

|  |
| --- |
| **public** **class** MacroCommand **implements** Command {  Command[] commands;    **public** MacroCommand(Command[] commands) {  **this**.commands = commands;  }    **public** **void** execute() {  **for** (**int** i = 0; i < commands.length; i++) {  commands[i].execute();  }  }    **public** **void** undo() {  **for** (**int** i = 0; i < commands.length; i++) {  commands[i].undo();  }  }  } |

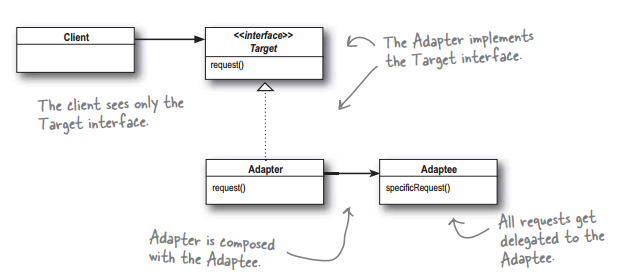
2. MainClass RemoteClientWithMacro và kết quả:

|  |
| --- |
| **public** **class** RemoteClientWithMacro {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  RemoteControl remoteControl = **new** RemoteControl();    Light livingRoomLight = **new** Light("Living Room");  Light bedRoomLight = **new** Light("bedRoomLight");  GarageDoor garageDoor = **new** GarageDoor();    LightOnCommand livingRoomLightOn = **new** LightOnCommand(livingRoomLight);  LightOffCommand livingRoomLightOff = **new** LightOffCommand(livingRoomLight);  LightOnCommand bedRoomLightOn = **new** LightOnCommand(bedRoomLight);  LightOffCommand bedRoomLightOff = **new** LightOffCommand(bedRoomLight);  GarageDoorOpenCommand garageDoorOpen = **new** GarageDoorOpenCommand(garageDoor);  GarageDoorCloseCommand garageDoorClose = **new** GarageDoorCloseCommand(garageDoor);    Command[] partyOn = { livingRoomLightOn, bedRoomLightOn, garageDoorOpen};  Command[] partyOff = {livingRoomLightOff, bedRoomLightOff, garageDoorClose};    MacroCommand partyOnMacro = **new** MacroCommand(partyOn);  MacroCommand partyOffMacro = **new** MacroCommand(partyOff);    remoteControl.setCommand(0, partyOnMacro, partyOffMacro);    System.***out***.println("------ Pushing Macro On ------");  remoteControl.onButtonWasPushed(0);  System.***out***.println("------ Pushing Macro On ------");  remoteControl.OffButtonWasPushed(0);  }  } |
| ------ Pushing Macro On ------  Living Room Light is on  bedRoomLight Light is on  Garage door moves up  Garage door is stop  ------ Pushing Macro On ------  Living Room Light is off  bedRoomLight Light is off  Garage door moves down  Garage door is stop |

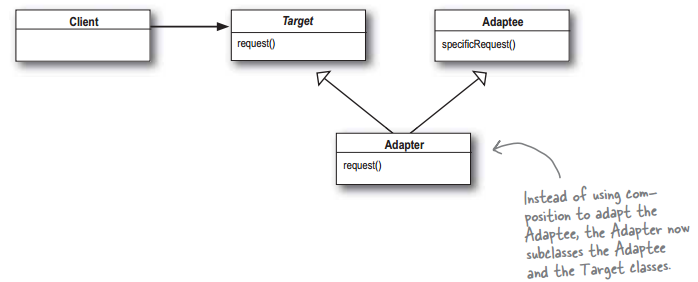
**Chapter 7. The Adapter and Facade Pattern**



Object Adapter Pattern



Class Adapter Patter



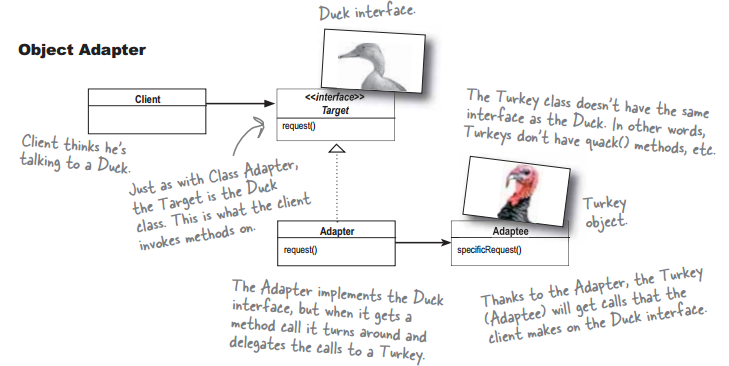
Nhược điểm của class adapter là Java không hỗ trợ đa thừa kế. Tuy nhiên nó có một ưu điểm là Adapter có thể gọi trực tiếp hoặc tùy ý override phương thức request() từ Adaptee.

Khi đã biết Adapter interface giúp biến đổi một interface này thành một interface khác, Joe đưa ra một ví dụ về 2 interface Duck và Turkey.

Interface Duck có 2 phương thức là quack() và fly(). Dĩ nhiên duck có thể bay một quãng rất dài.

Interface Turkey có 2 phương thức là gobble() và fly(). Tuy nhiên turkey chỉ bay được một quãng ngắn.

Joe viết một Class ***Turkey Adapter*** nhằm biến đổi Turkey về thành Duck.



1. Interface Duck và class MallardDuck

|  |
| --- |
| **public** **interface** Duck {  **public** **void** quack();  **public** **void** fly();  } |
| **public** **class** MallardDuck **implements** Duck {  **public** **void** quack() {  System.***out***.println("quack");  }  **public** **void** fly() {  System.***out***.println("I'm flying");  }  } |

2. Interface Turkey và class WildTurkey

|  |
| --- |
| **public** **interface** Turkey {  **public** **void** gobble();  **public** **void** fly();  } |
| **public** **class** WildTurkey **implements** Turkey {  **public** **void** gobble() {  System.***out***.println("gobble gobble");  }  **public** **void** fly() {  System.***out***.println("I'm flying a short distance");  }  } |

3. TurkeyAdapter, biến đổi turkey thành duck.

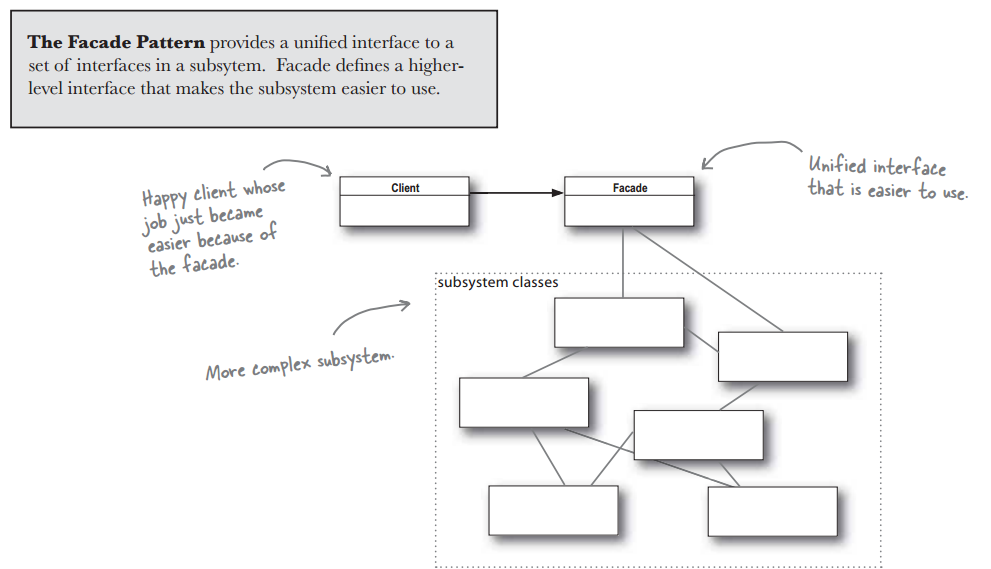
Vì turkey chỉ bay được quãng ngắn nên hàm fly được gọi 3 lần.

|  |
| --- |
| **public** **class** TurkeyAdapter **implements** Duck {  Turkey turkey;    **public** TurkeyAdapter(Turkey turkey) {  **this**.turkey = turkey;  }    **public** **void** quack() {  turkey.gobble();  }    **public** **void** fly() {  **for** (**int** i = 0; i < 3; i++) {  turkey.fly();  }  }  } |

4. Main class DuckTestDrive và kết quả

|  |
| --- |
| **public** **class** DuckTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  MallardDuck duck = **new** MallardDuck();  WildTurkey turkey = **new** WildTurkey();  Duck turkeyAdapter = **new** TurkeyAdapter(turkey);    System.***out***.println("\nThe Duck says...");  *testDuck*(duck);  System.***out***.println("\nThe Turkey Adapter says...");  *testDuck*(turkeyAdapter);  }    **static** **void** testDuck(Duck duck) {  duck.quack();  duck.fly();  }  } |
| The Duck says...  quack  I'm flying  The Turkey Adapter says...  gobble gobble  I'm flying a short distance  I'm flying a short distance  I'm flying a short distance |

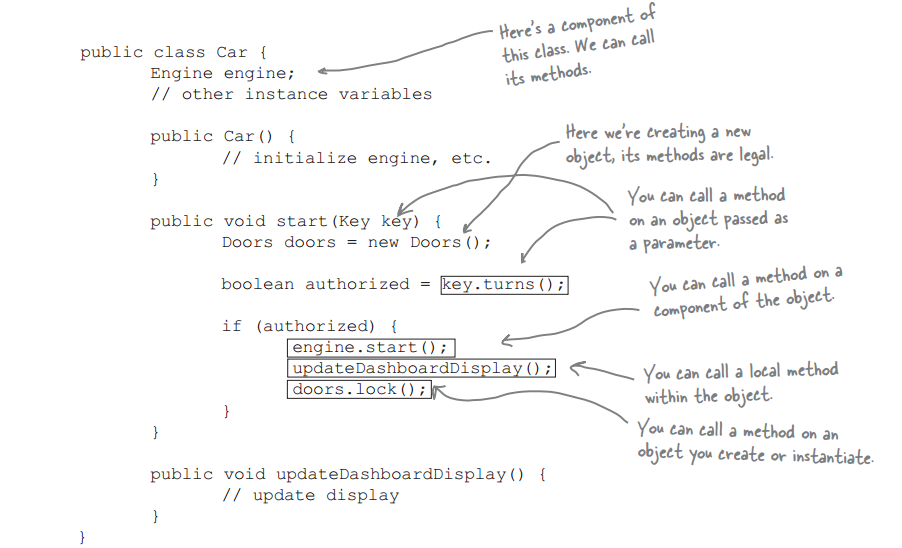
**The Facade Pattern defined**



Nguyên tắc thiết kế thứ 7: Principle of Least Knowledge – talk only to your immediate friends.

Khi bạn thiết kế một hệ thống, với mỗi object, hãy cẩn thận với số lượng class mà nó tương tác qua lại và cách mà chúng tương tác với nhau.

Lời khuyên, từ một phương thức trong object, chúng ta chỉ nên gọi một trong các phương thức sau:

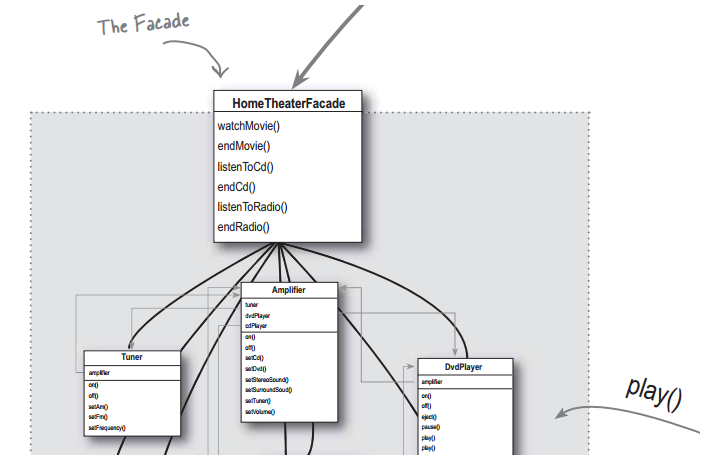


Bài toán thiết kế rạp chiếu phim tại nhà.

Một rạp chiếu phim mini gồm nhiều thiết bị như đèn, máy chiếu, ampli, dvdPlayer, etc. Mỗi thiết bị lại có một số thao tác điều khiển nhất định được cung cấp bởi nhà sản xuất.

Khi muốn xem một bộ phim, người dùng phải thao tác rất nhiều thiết bị, ví dụ như, hạn chế độ sáng của đèn, bật máy chiếu, chỉnh máy chiếu sang chế độ wide screen, bật ampli, cho dvd vào, chạy ampli, chỉnh âm thanh thích hợp... Tương tự, khi xem xong phim, người dùng cũng phải lật ngược lại các thao tác trên.

Áp dụng Facade Pattern, Joe có thể tạo ra một class ***HomeTheaterFacade***. Ở đó, Joe định nghĩa ra các hành động mà người dùng mong muốn thực hiện, như xem phim, kết thúc phim, nghe nhạc, nghe radio,... Mỗi hành động lại bao gồm N thao tác với các thiết bị máy móc. Như vậy, về phía người dùng, họ chỉ quan tâm là muốn thực hiện hành động nào, còn chuỗi thao tác với máy móc sẽ tuần tự thực hiện.



Joe tiến hành code.

1. Các class điều khiển thiết bị được nhà sản xuất cấp sẵn

|  |
| --- |
| **public** **class** Light {  **public** **void** on() {  System.***out***.println("Light is on");  }  **public** **void** dim(**int** brightness) {  System.***out***.println("Light is set to brightness = " + brightness);  }  **public** **void** off() {  System.***out***.println("Light is off");  }  } |
| **public** **class** Projector {  **public** **void** on() {  System.***out***.println("Projector is on");  }  **public** **void** off() {  System.***out***.println("Projector is off");  }  **public** **void** wideScreenMode() {  System.***out***.println("Projector is set to wide screen mode");  }  } |
| **public** **class** Amplifier {  **public** **void** on() {  System.***out***.println("Amplifier is on");  }  **public** **void** off() {  System.***out***.println("Amplifier is off");  }  **public** **void** setDvd(String dvd) {  System.***out***.println("Amplifier is playing " + dvd);  }  **public** **void** setVolume(**int** volume) {  System.***out***.println("Amplifier set volume = " + volume);  }  } |

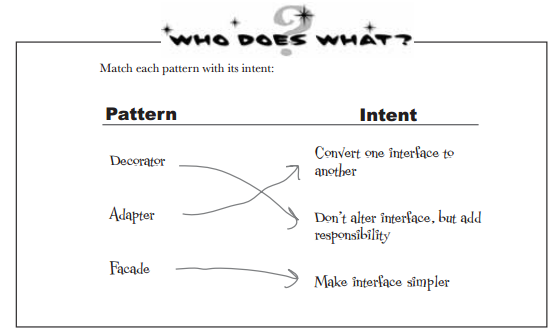
2. Class HomeTheaterFacade: định nghĩa các phương thức watchMovie(), endMovie()

|  |
| --- |
| **public** **class** HomeTheaterFacade {  Light light;  Projector projector;  Amplifier amp;    **public** HomeTheaterFacade(Light light, Projector projector, Amplifier amp) {  **this**.light = light;  **this**.projector = projector;  **this**.amp = amp;  }    **public** **void** watchMovie(String movie) {  light.dim(10);  projector.on();  projector.wideScreenMode();  amp.on();  amp.setDvd(movie);  amp.setVolume(24);  }    **public** **void** endMovie() {  light.on();  projector.off();  amp.off();  }  } |

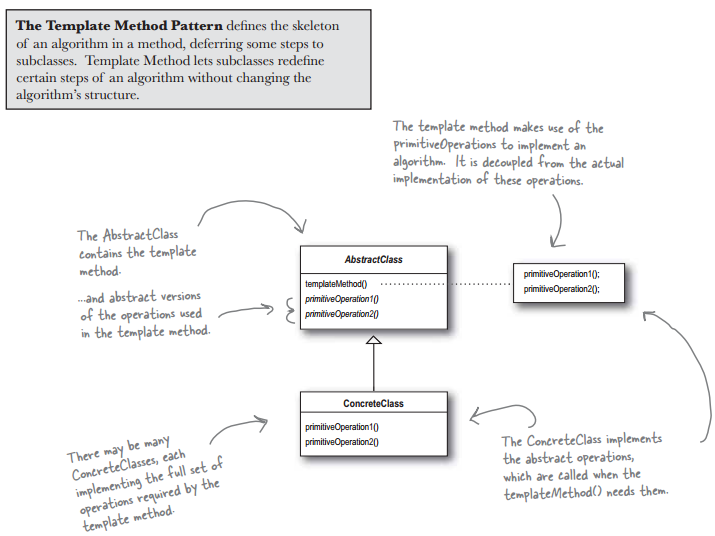
3. Main class HomeTheaterTestDrive và kết quả thực thi

|  |
| --- |
| **public** **class** HomeTheaterTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  Light light = **new** Light();  Projector projector = **new** Projector();  Amplifier amp = **new** Amplifier();    HomeTheaterFacade homeTheaterFacade = **new** HomeTheaterFacade(light, projector, amp);    System.***out***.println("--- Watch movie ---");  homeTheaterFacade.watchMovie("Lord of the Rings");  System.***out***.println("--- End movie ---");  homeTheaterFacade.endMovie();  }  } |
| --- Watch movie ---  Light is set to brightness = 10  Projector is on  Projector is set to wide screen mode  Amplifier is on  Amplifier is playing Lord of the Rings  Amplifier set volume = 24  --- End movie ---  Light is on  Projector is off  Amplifier is off |

**Phân biệt 3 pattern Decorator, Adapter và Facade**

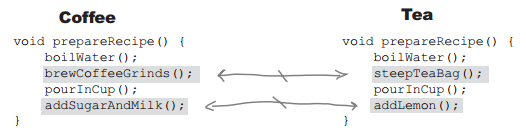


**Chapter 8. The Template Method Pattern – Encapsulating Algorithms**



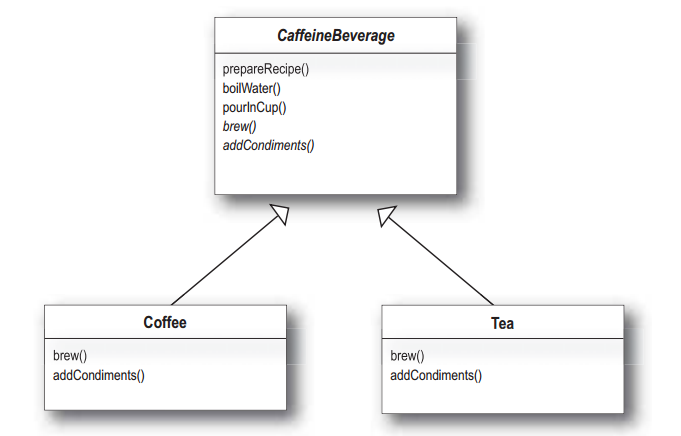
Template Method tập trung vào việc tạo một bản mẫu cho một thuật toán. Bản mẫu là một phương thức định nghĩa lần lượt từng bước của thuật toán (templateMethod() - AbstractClass), trong đó một số bước được khai báo abstract và sẽ được định nghĩa cụ thể bởi các subclass. Điều này đảm bảo sự toàn vẹn của thuật toán, trong khi các subclass vẫn có thể áp dụng một vài điểm khác nhau.

Joe làm ví dụ minh họa với cách pha Cà phê và Trà. Các bước pha được liệt kê như sau:



Joe chỉ ra rằng thuật toán pha đồ uống đều có 4 bước giống nhau, trong đó 2 bước boilWater() và pourInCup() là giống nhau nên sẽ được định nghĩa sẵn ở superclass CaffeinBeverage; 2 bước còn lại khác nhau ở cách áp dụng cụ thể, chúng được khai báo là 2 abstract class brew() và addCondiments(). Các subclass Coffee và Tea sẽ override chúng.

Anh ta thiết kế sơ đồ UML



Trong code, Joe áp dụng thêm hook (phương thức customerWantsCondiment - boolean), cho phép người dùng lựa chọn addCondiment hay không.

1. Superclass CaffeinBeverage

Định nghĩa thuật toán pha đồ uống trong prepareRecipe().

Phương thức customerWantsCondiments() mặc định trả về true.

|  |
| --- |
| **public** **abstract** **class** CaffeineBeverage {  **final** **void** prepareRecipe() {  boilWater();  brew();  pourInCup();  **if** (customerWantsCondiments()) {  addCondiments();  }  }    **abstract** **void** brew();  **abstract** **void** addCondiments();    **void** boilWater() {  System.***out***.println("Boiling water");  }    **void** pourInCup() {  System.***out***.println("Pouring into Cup");  }    **boolean** customerWantsCondiments() {  **return** **true**;  }  } |

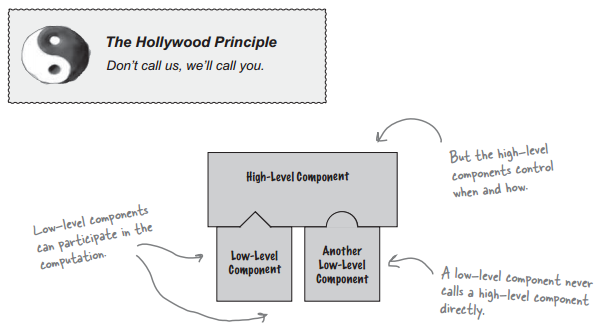
2. Class Coffee và Tea – implement các abstract method brew() và addCondiments() theo cách riêng

|  |
| --- |
| **public** **class** Coffee **extends** CaffeineBeverage {  **public** **void** brew() {  System.***out***.println("Dripping Coffee through filter");  }  **public** **void** addCondiments() {  System.***out***.println("Adding Sugar and Milk");  }  } |
| **public** **class** Tea **extends** CaffeineBeverage {  **public** **void** brew() {  System.***out***.println("Steeping the tea");  }    **public** **void** addCondiments() {  System.***out***.println("Adding Lemon");  }  } |
| **public** **class** CoffeeWithHook **extends** CaffeineBeverage {  **public** **void** brew() {  System.***out***.println("Dripping Coffee through filter");  }    **public** **void** addCondiments() {  System.***out***.println("Adding Sugar and Milk");  }    **public** **boolean** customerWantsCondiments() {  String answer = getUserInput();    **if** (answer.toLowerCase().startsWith("y")) {  **return** **true**;  } **else** {  **return** **false**;  }  }    **private** String getUserInput() {  String answer = **null**;    System.***out***.println("Would you like milk and sugar with your coffee (y/n)? ");    BufferedReader in = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **try** {  answer = in.readLine();  } **catch** (IOException ioe) {  System.***err***.println("IO error trying to read your answer");  }    **if** (answer == **null**) {  **return** "no";  }  **return** answer;  } |

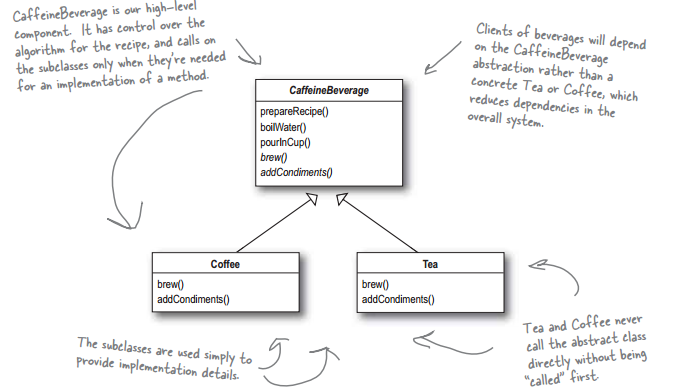
3. Main class BeverageTestDrive

|  |
| --- |
| **public** **class** BeverageTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  Tea myTea = **new** Tea();  Coffee myCoffee = **new** Coffee();  CoffeeWithHook myCoffeeWithHook = **new** CoffeeWithHook();    System.***out***.println("--- Make a cup of Tea ---");  myTea.prepareRecipe();  System.***out***.println("--- Make a cup of Coffee ---");  myCoffee.prepareRecipe();  System.***out***.println("--- Make a cup of Coffee with Hook ---");  myCoffeeWithHook.prepareRecipe();  }  } |
| --- Make a cup of Tea ---  Boiling water  Steeping the tea  Pouring into Cup  Adding Lemon  --- Make a cup of Coffee ---  Boiling water  Dripping Coffee through filter  Pouring into Cup  Adding Sugar and Milk  --- Make a cup of Coffee with Hook ---  Boiling water  Dripping Coffee through filter  Pouring into Cup  Would you like milk and sugar with your coffee (y/n)?  y  Adding Sugar and Milk |

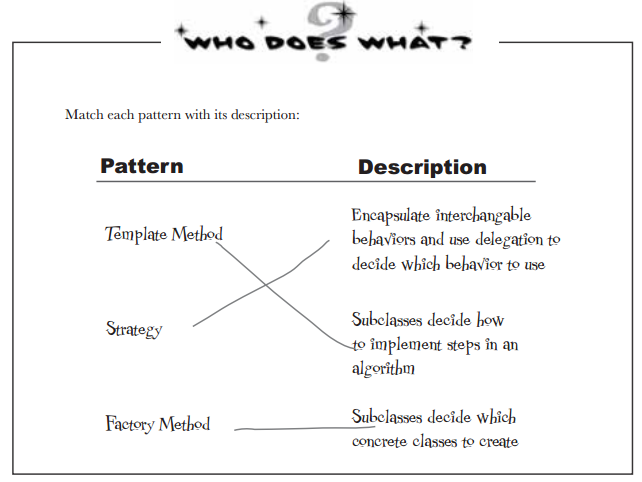
**The Hollywood Principle**



Nguyên lý Hollywood đã được áp dụng ở bài toán pha cà phê

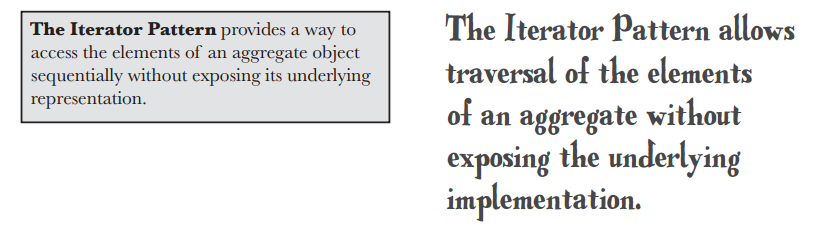


Phân biệt Template Method, Strategy và Factory Method pattern



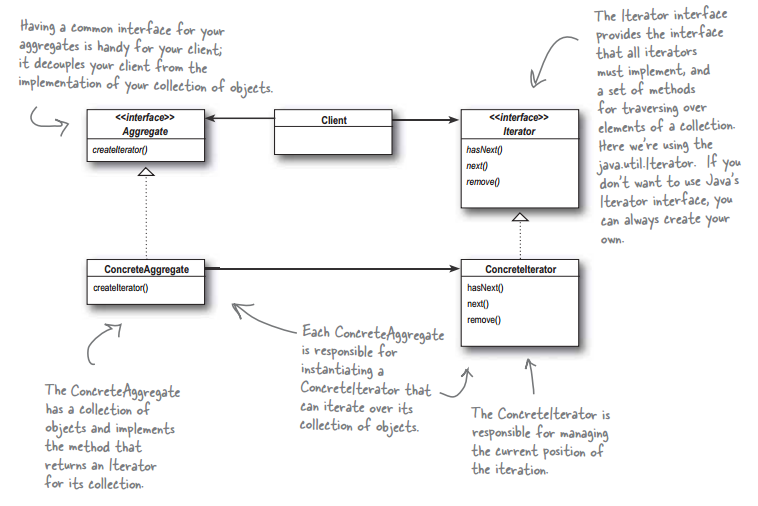
**Chapter 9. The Iterator and Composite Pattern**

Định nghĩa Iterator Pattern



Iterator Pattern cho phép Client duyệt và truy cập các phần tử của tất cả object theo một cách đồng nhất, mà không cần quan tâm đến cấu trúc dữ liệu của từng object.

Biểu diễn UML tổng quát của Iterator Pattern



Bài toán Joe nhận được từ một nhà hàng có hai đầu bếp là Louis và Mel. Louis chịu trách nhiệm về thực đơn bữa sáng – PancakeMenu; Mel chịu trách nhiệm về thực đơn bữa trưa – DinerMenu. Trong thực đơn, các món ăn đều có định dạng thống nhất, gồm 4 thông tin được lưu trữ:

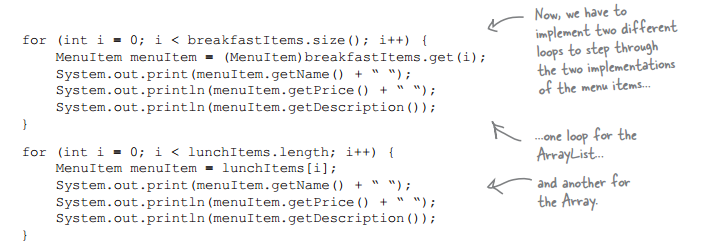
|  |
| --- |
| String name; // Ten mon an  String description; // Mo ta mon an  **boolean** vegetarian; // Do chay hay khong  **double** price; // Gia tien |

Hiện tại 2 đầu bếp đang bất đồng quan điểm về cách lưu trữ của thực đơn:

+ ArrayList: Louis sử dụng ArrayList để lưu trữ món ăn cho pancakeMenu, vì nó có thể dễ dàng mở rộng.

+ Array thường: Mel sử dụng array để lưu trữ món cho dinerMenu, giúp anh ta kiểm soát MAX\_ITEMS.

Đứng trên vai trò của người bồi bàn, Joe nhận thấy việc liệt kê các món ăn từ hai menu là không hề đồng nhất; Tuy cùng một mục đích nhưng Joe phải duyệt hai lần cho hai cấu trúc khác nhau.



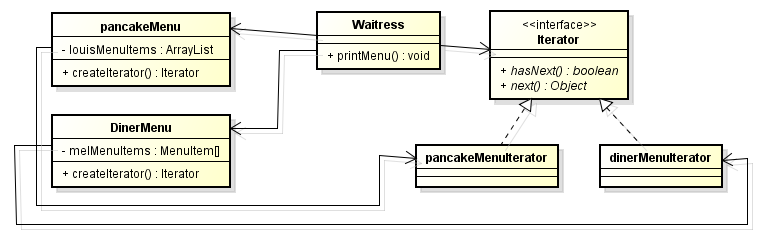
Joe áp dụng Iterator vào thiết kế bài toán.

Đầu tiên anh ta thêm vào Iterator interface gồm 2 phương thức hasNext() và next(); và tạo ra các Class pancakeMenuIterator, dinerMenuIterator.

+ hasNext: xác định có phần tử tiếp theo trong Iterator hay không.

+ next: trả về một Object là phần tử tiếp theo

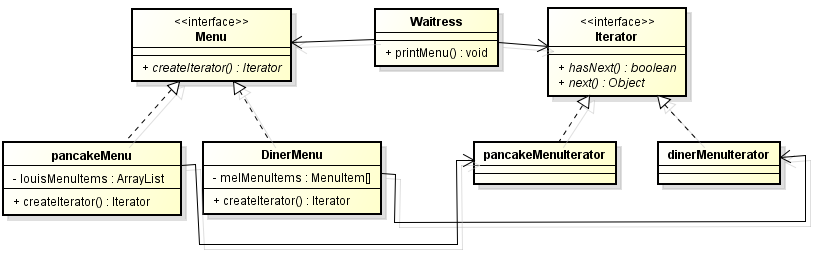
+ createIterator: chuyển các cấu trúc dữ liệu ArrayList hay array thường về cấu trúc Iterator đồng nhất.



Cải tiến thêm, Joe tạo thêm một interface Menu có chứa phương thức createIterator. pancakeMenu và DinerMenu implement Menu interface, công việc của Waitress trở nên dễ dàng hơn, không cần quan tâm đến cụ thể từng loại Menu nữa.

|  |
| --- |
| // Before  PancakeMenu pancakeMenu;  DinerMenu dinerMenu;  // After  Menu pancakeMenu;  Menu dinerMenu; |

Sơ đồ UML hoàn thiện



Cuối cùng, Joe code các class:

1. MenuItem class – mô tả món ăn

|  |
| --- |
| **public** **class** MenuItem {  String name; // Ten mon an  String description; // Mo ta mon an  **boolean** vegetarian; // Do chay hay khong  **double** price; // Gia tien    **public** MenuItem(String name, String description,  **boolean** vegetarian, **double** price) {  **this**.name = name;  **this**.description = description;  **this**.vegetarian = vegetarian;  **this**.price = price;  }    **public** String getName() {  **return** name;  }    **public** String getDescription() {  **return** description;  }    **public** **boolean** isVegetarian() {  **return** vegetarian;  }    **public** **double** getPrice() {  **return** price;  }  } |

2. Interface Menu và class pancakeMenu, dinerMenu

|  |
| --- |
| **public** **interface** Menu {  **public** Iterator createIterator();  } |
| **public** **class** PancakeMenu **implements** Menu {  ArrayList louisMenuItems;    **public** PancakeMenu() {  louisMenuItems = **new** ArrayList();    addItem("Banh my thit", "thit ba chi nuong", **false**, 1.50);  addItem("Banh ran", "Nhan dau xanh", **true**, 0.50);  addItem("Banh my goi", "Chi co bot my", **true**, 0.75);  }    **public** **void** addItem(String name, String description,  **boolean** vegetarian, **double** price) {  MenuItem menuItem = **new** MenuItem(name, description, vegetarian, price);  louisMenuItems.add(menuItem);  }    **public** Iterator createIterator() {  **return** **new** PancakeIterator(louisMenuItems);  }  } |
| **public** **class** DinerMenu **implements** Menu {  **static** **final** **int** ***MAX\_ITEMS*** = 6;  **int** numberOfItems = 0;  MenuItem[] melMenuItems;    **public** DinerMenu() {  melMenuItems = **new** MenuItem[***MAX\_ITEMS***];    addItem("Com binh dan", "Com canh ca thit", **false**, 2.19);  addItem("Bun cha", "Thit nuong hoac nem", **false**, 1.79);  addItem("Bun dau", "dau phu cham mam tom", **true**, 1.39);  }    **public** **void** addItem(String name, String description,  **boolean** vegetarian, **double** price) {  MenuItem menuItem = **new** MenuItem(name, description, vegetarian, price);  **if** (numberOfItems >= ***MAX\_ITEMS***) {  System.***out***.println("Sorry, menu is full!");  } **else** {  melMenuItems[numberOfItems] = menuItem;  numberOfItems++;  }  }    **public** Iterator createIterator() {  **return** **new** DinerMenuIterator(melMenuItems);  }  } |

3. Interface Iterator và class pancakeIterator, dinerIterator

|  |
| --- |
| **public** **interface** Iterator {  **boolean** hasNext();  Object next();  } |
| **public** **class** PancakeIterator **implements** Iterator {  ArrayList items;  **int** position = 0;    **public** PancakeIterator(ArrayList items) {  **this**.items = items;  }  @Override  **public** **boolean** hasNext() {  **if** (position < items.size()) {  **return** **true**;  } **else** {  **return** **false**;  }  }  @Override  **public** Object next() {  MenuItem item = (MenuItem) items.get(position);  position++;  **return** item;  }  } |
| **public** **class** DinerMenuIterator **implements** Iterator {  MenuItem[] items;  **int** position = 0;    **public** DinerMenuIterator(MenuItem[] items) {  **this**.items = items;  }    @Override  **public** **boolean** hasNext() {  **if** (position >= items.length || items[position] == **null**) {  **return** **false**;  } **else** {  **return** **true**;  }  }  @Override  **public** Object next() {  // **TODO** Auto-generated method stub  MenuItem item = items[position];  position++;  **return** item;  }  } |

4. Class Waitress

|  |
| --- |
| **public** **class** Waitress {  Menu pancakeMenu;  Menu dinerMenu;    **public** Waitress(Menu pancakeMenu, Menu dinerMenu) {  **this**.pancakeMenu = pancakeMenu;  **this**.dinerMenu = dinerMenu;  }    **public** **void** printMenu() {  Iterator pancakeIterator = pancakeMenu.createIterator();  Iterator dinerIterator = dinerMenu.createIterator();    System.***out***.println("MENU\n---\nBREAKFAST");  printMenu(pancakeIterator);  System.***out***.println("\n LUNCH");  printMenu(dinerIterator);  }    **public** **void** printMenu(Iterator iterator) {  **while** (iterator.hasNext()) {  MenuItem item = (MenuItem) iterator.next();  System.***out***.print(item.getName() + ", ");  System.***out***.print(item.getPrice() + " -- ");  System.***out***.println(item.getDescription());  }  }  } |

5. Main class MenuTestDrive và output

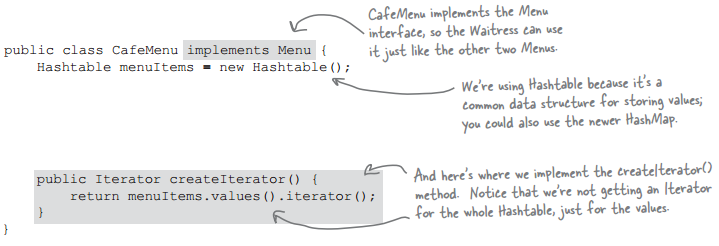
|  |
| --- |
| **public** **class** MenuTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  PancakeMenu pancakeMenu = **new** PancakeMenu();  DinerMenu dinerMenu = **new** DinerMenu();    Waitress waitress = **new** Waitress(pancakeMenu, dinerMenu);  waitress.printMenu();  }  } |
| MENU  ---  BREAKFAST  Banh my thit, 1.5 -- thit ba chi nuong  Banh ran, 0.5 -- Nhan dau xanh  Banh my goi, 0.75 -- Chi co bot my  LUNCH  Com binh dan, 2.19 -- Com canh ca thit  Bun cha, 1.79 -- Thit nuong hoac nem  Bun dau, 1.39 -- dau phu cham mam tom |

Suy nghĩ mở rộng:

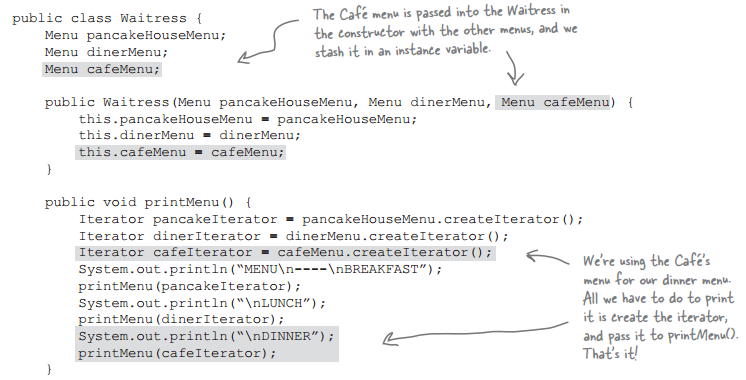
+ Cửa hàng muốn thêm một class CafeMenu kiểu Hashtable. Hãy suy nghĩ xem code hiện tại cần phải thay đổi những gì?

Hashtable là một cấu trúc phức tạp hơn ArrayList do nó hỗ trợ cả keys và values, nhưng chúng ta vẫn có thể lấy một iterator cho values (MenuItems).

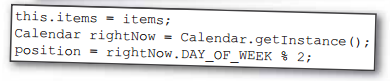
Class CafeMenu implement interface Menu:



Thay đổi ở class Waitress:



+ Cửa hàng muốn thay đổi DinerMenu. Danh sách món sẽ được chia thành thực đơn của ngày chẵn và ngày lẻ trong tuần. Hãy suy nghĩ cách áp dụng nó?



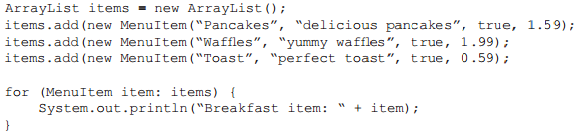
**Iteration và Collections trong Java 5**

Java 5 có một dạng mới của for gọi là **for/in**, cho phép bạn duyệt một collection hay array.

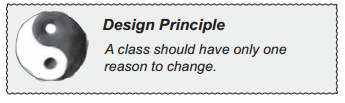
Cấu trúc của nó như sau:



Ví dụ, duyệt một ArrayList sử dụng for/in:

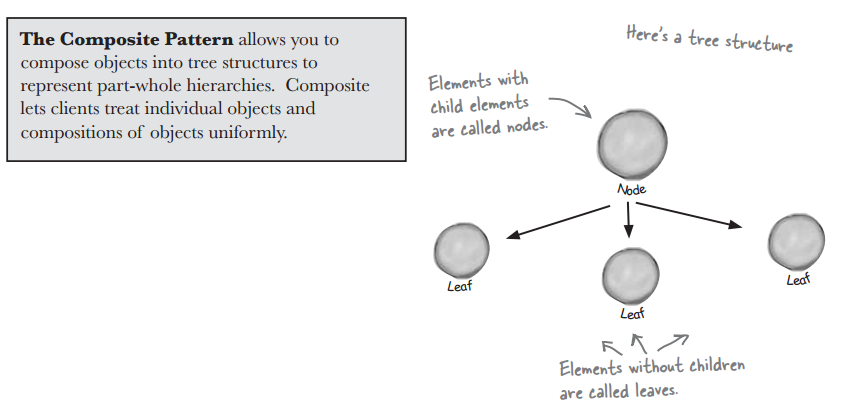


**Nguyên lý thiết kế:**



Một class không chỉ thực hiện các phương thức của chính nó, mà còn thực hiện một vài nhiệm vụ khác (ví dụ như iterator), nghĩa là chúng ta đã cho class đó hai lí do để thay đổi. Điều này làm tăng khả năng class sẽ phải sửa đổi trong tương lai. Giải pháp? Nguyên lý hướng dẫn ta chỉ định một nhiệm vụ cho một class (và chỉ duy nhất một).

**Composite Pattern defined**

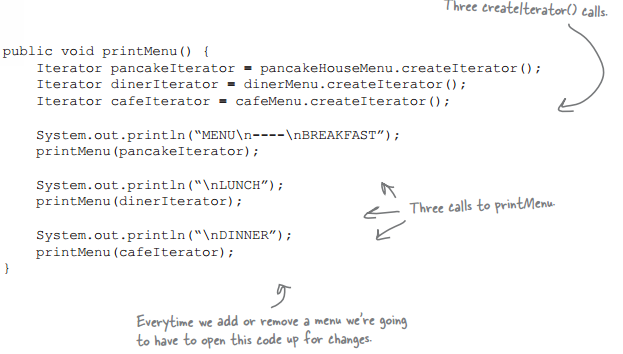


Trước khi đến với ví dụ về Composite Pattern, ta nhắc lại về nhiệm vụ liệt kê các thực đơn của Waitress trong ví dụ Iterator. Hiện tại chúng ta đang đối xử mỗi menu như là một object riêng biệt, do đó chúng dẫn tới một số điểm cần phải cải tiến:

+ Mỗi menu cần phải gọi phương thức createIterator() một lần.

+ Để liệt kê danh sách món trong mỗi menu cũng cần gọi phương thức printMenu() tương ứng.

+ Khi cần thêm hoặc bớt menu thì phương thức printMenu() sẽ phải sửa đổi.

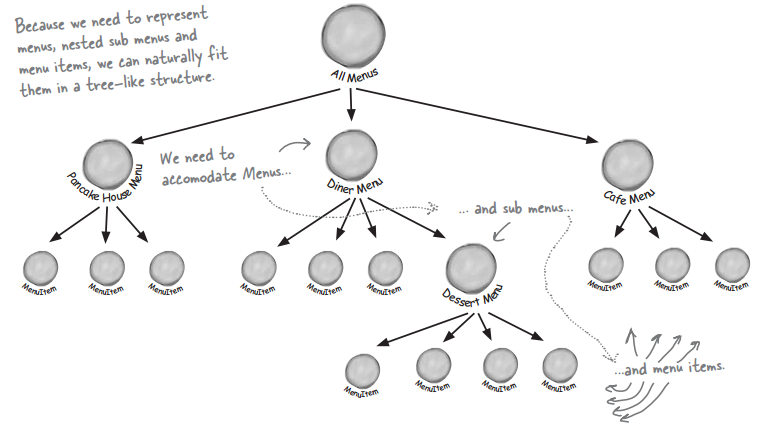


Composite Design Pattern cho phép chúng ta xây dựng một cấu trúc cây Menu, thực hiện các công việc:

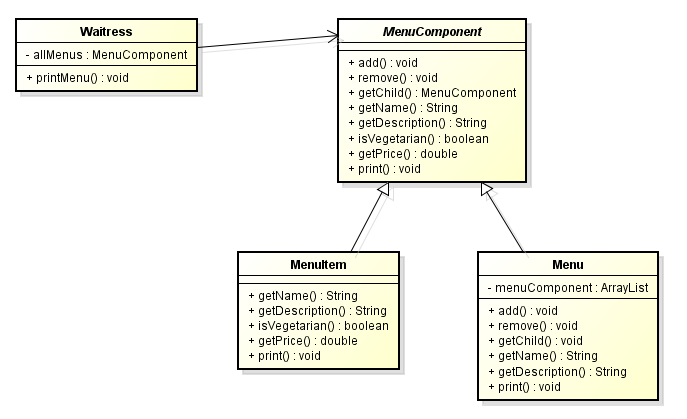
+ Lưu trữ tất cả các menu, submenu và menu item.

+ Có thể duyệt tất cả các item có trong menu một cách thuận tiện, không mất công gọi nhiều lần.

+ Có thể linh hoạt duyệt một submenu cụ thể và các child menu của nó.



Thiết kế sơ đồ UML của bài toán Menu:

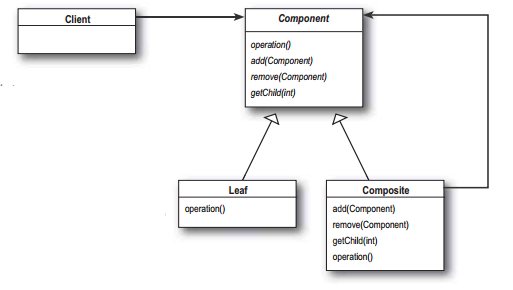


Sơ đồ UML tổng quát của Composite Pattern:

+ Component là một abstract class, ngoài việc quy định các phương thức operation chung, nó còn quy định thêm 3 phương thức: add(), remove(), và getChild().

+ Nếu thừa kế của **Component** là một class **Leaf**, nó chỉ cần định nghĩa các operation.

+ Nếu thừa kế của **Component** là một class **Composite**, nó có chứa một ArrayList chứa các Component con. Đó có thể là một Leaf hoặc một Composite con.



Áp dụng composite menu

1. Abstract class MenuComponent

|  |
| --- |
| **public** **abstract** **class** MenuComponent {    **public** **void** add(MenuComponent menuComponent) {  **throw** **new** UnsupportedOperationException();  }    **public** **void** remove(MenuComponent menuComponent) {  **throw** **new** UnsupportedOperationException();  }    **public** MenuComponent getChild(**int** i) {  **throw** **new** UnsupportedOperationException();  }    **public** String getName() {  **throw** **new** UnsupportedOperationException();  }    **public** String getDescription() {  **throw** **new** UnsupportedOperationException();  }    **public** **boolean** isVegetarian() {  **throw** **new** UnsupportedOperationException();  }    **public** **double** getPrice() {  **throw** **new** UnsupportedOperationException();  }    **public** **void** print() {  **throw** **new** UnsupportedOperationException();  }  } |

2. Class MenuItem – Leaf

|  |
| --- |
| **public** **class** MenuItem **extends** MenuComponent {  String name;  String description;  **boolean** vegetarian;  **double** price;    **public** MenuItem(String name, String description,  **boolean** vegetarian, **double** price) {  **this**.name = name;  **this**.description = description;  **this**.vegetarian = vegetarian;  **this**.price = price;  }    **public** String getName() {  **return** name;  }    **public** String getDescription() {  **return** description;  }    **public** **boolean** isVegetarian() {  **return** vegetarian;  }    **public** **double** getPrice() {  **return** price;  }    **public** **void** print() {  System.***out***.print(" " + getName());  **if** (isVegetarian()) {  System.***out***.print("(v)");  }  System.***out***.println(", " + getPrice());  System.***out***.println(" -- " + getDescription());  }  } |

3. Class Menu – Composite

|  |
| --- |
| **public** **class** Menu **extends** MenuComponent {  ArrayList menuComponents = **new** ArrayList();  String name;  String description;    **public** Menu(String name, String description) {  **this**.name = name;  **this**.description = description;  }    **public** **void** add(MenuComponent menuComponent) {  menuComponents.add(menuComponent);  }    **public** **void** remove(MenuComponent menuComponent) {  menuComponents.remove(menuComponent);  }  **public** MenuComponent getChild(**int** i) {  **return** (MenuComponent) menuComponents.get(i);  }    **public** String getName() {  **return** name;  }    **public** String getDescription() {  **return** description;  }    **public** **void** print() {  System.***out***.print("\n" + getName());  System.***out***.println(", " + getDescription());  System.***out***.println("------------------------");    Iterator iterator = menuComponents.iterator();  **while** (iterator.hasNext()) {  MenuComponent menuComponent =  (MenuComponent) iterator.next();  menuComponent.print();  }  }  } |

4. Class Waitress – Client

Thực sự công việc của Waitress rất nhẹ nhàng như mục đích thiết kế theo Composite Pattern.

|  |
| --- |
| **public** **class** Waitress {  MenuComponent allMenus;    **public** Waitress(MenuComponent allMenus) {  **this**.allMenus = allMenus;  }    **public** **void** printMenu() {  allMenus.print();  }  } |

5. Main class MenuTestDrive và kết quả

|  |
| --- |
| **public** **class** MenuTestDrive {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  MenuComponent pancakeMenu =  **new** Menu("PANCAKE MENU", "Breakfast");  MenuComponent dinerMenu =  **new** Menu("DINER MENU", "Lunch");  MenuComponent cafeMenu =  **new** Menu("CAFE MENU", "Dinner");  MenuComponent dessertMenu =  **new** Menu("DESSERT MENU", "Dessert of course!");    MenuComponent allMenus =  **new** Menu("ALL MENUS", "All menus combined.");    allMenus.add(pancakeMenu);  allMenus.add(dinerMenu);  allMenus.add(cafeMenu);    pancakeMenu.add(**new** MenuItem("banh my", "thit nuong", **false**, 0.99));  pancakeMenu.add(**new** MenuItem("Banh ran", "nhan dau xanh", **true**, 0.49));    dinerMenu.add(**new** MenuItem("Com binh dan", "Com canh ca", **false**, 2.99));  dinerMenu.add(dessertMenu);  dessertMenu.add(**new** MenuItem("Sua chua", "nguyen chat", **true**, 0.29));  dessertMenu.add(**new** MenuItem("Kem", "Trang Tien", **true**, 0.59));    cafeMenu.add(**new** MenuItem("Den da", "nguyen chat", **true**, 1.49));    Waitress waitress = **new** Waitress(allMenus);  waitress.printMenu();  }  } |
| ALL MENUS, All menus combined.  ------------------------  PANCAKE MENU, Breakfast  ------------------------  banh my, 0.99  -- thit nuong  Banh ran(v), 0.49  -- nhan dau xanh  DINER MENU, Lunch  ------------------------  Com binh dan, 2.99  -- Com canh ca  DESSERT MENU, Dessert of course!  ------------------------  Sua chua(v), 0.29  -- nguyen chat  Kem(v), 0.59  -- Trang Tien  CAFE MENU, Dinner  ------------------------  Den da(v), 1.49  -- nguyen chat |

**Flashback to Iterator**

Bài toán đặt ra: Chúng ta đã sử dụng Iterator trong phương thức print(). Bây giờ ta sẽ áp dụng Iterator vào các class Menu và MenuItem, với 2 mục đích chính:

+ Cho phép Waitress có thể duyệt toàn bộ Menu và liệt kê các món có chung một đặc điểm nào đó, ví dụ, liệt kê các món ăn chay (thuộc tính *vegetarian* bằng true).

+ Cho phép Waitress có thể duyệt một menu con tùy ý, ví dụ duyệt Diner Menu.

Bằng cách áp dụng cấu trúc stack và đệ quy, ta sử dụng Composite Iterator:

|  |
| --- |
| **public** **class** CompositeIterator **implements** Iterator {  Stack stack = **new** Stack();    **public** CompositeIterator(Iterator iterator) {  stack.push(iterator);  }    **public** **boolean** hasNext() {  **if** (stack.empty()) {  **return** **false**;  } **else** {  Iterator iterator = (Iterator) stack.peek();  **if** (!iterator.hasNext()) {  stack.pop();  **return** hasNext();  } **else** {  **return** **true**;  }  }  }  **public** Object next() {  **if** (hasNext()) {  Iterator iterator = (Iterator) stack.peek();  MenuComponent component = (MenuComponent) iterator.next();  **if** (component **instanceof** Menu) {  stack.push(((Menu) component).createIterator());  }  **return** component;  } **else** {  **return** **null**;  }  }  } |
| **public** **class** NullIterator **implements** Iterator {  @Override  **public** **boolean** hasNext() {  **return** **false**;  }  @Override  **public** Object next() {  **return** **null**;  }  } |

Phương thức abstract *createIterator()* được thêm vào abstract class MenuComponent và nó được định nghĩa khác nhau ở trong class Menu và MenuItem.

|  |
| --- |
| **public** **abstract** **class** MenuComponent {  // other code doesn't change  **public** **abstract** Iterator createIterator();  } |
| **public** **class** MenuItem **extends** MenuComponent {  // other code doesn't change  **public** Iterator createIterator() {  **return** **new** NullIterator();  }  } |
| **public** **class** Menu **extends** MenuComponent {  // other code doesn't change  **public** Iterator createIterator() {  **return** **new** CompositeIterator(menuComponents.iterator());  }  } |

Công việc của Waitress chính là tạo Composite Iterator của allMenus và duyệt nó để tìm ra các món ăn có thuộc tính *vegetarian* bằng true.

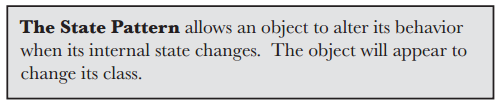
|  |
| --- |
| **public** **class** Waitress {  // other code doesn't change  **public** **void** printVegetarianMenu() {  Iterator iterator = allMenus.createIterator();    System.***out***.println("\nVEGETARIAN MENU\n------");  **while** (iterator.hasNext()) {  MenuComponent menuComponent = (MenuComponent) iterator.next();  System.***out***.println(menuComponent.getName());  **try** {  **if** (menuComponent.isVegetarian()) {  menuComponent.print();  }  } **catch** (UnsupportedOperationException e) {}  }  }  } |

Main class đơn giản chỉ cần gọi phương thức *printVegetarianMenu()* của instance waitress.

Tuy nhiên, không hiểu sao danh sách các món trong DessertMenu bị liệt kê 2 lần.

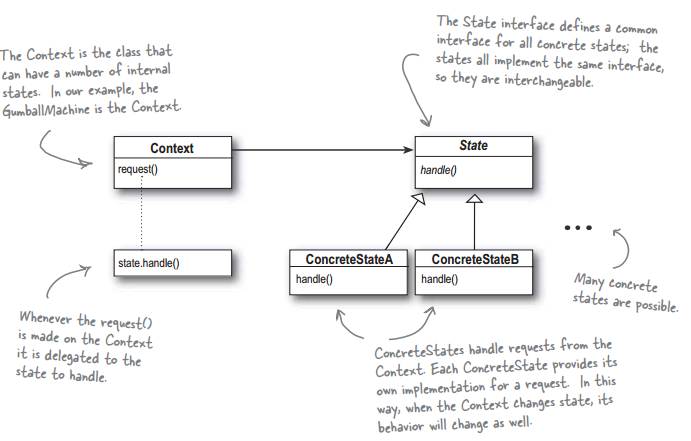
|  |
| --- |
| **public** **class** MenuTestDrive {  **public** **static** **void** main(String[] args) {  // other code doesn't change  Waitress waitress = **new** Waitress(allMenus);  //waitress.printMenu();  waitress.printVegetarianMenu();  }  } |
| VEGETARIAN MENU  ------  PANCAKE MENU  banh my  Banh ran  Banh ran(v), 0.49  -- nhan dau xanh  DINER MENU  Com binh dan  DESSERT MENU  Sua chua  Sua chua(v), 0.29  -- nguyen chat  Kem  Kem(v), 0.59  -- Trang Tien  Sua chua => Bị gọi và in ra lần thứ 2 các MenuItem trong DessertMenu.  Sua chua(v), 0.29  -- nguyen chat  Kem  Kem(v), 0.59  -- Trang Tien  CAFE MENU  Den da  Den da(v), 1.49  -- nguyen chat |

**Chapter 10. The State Pattern – State of Things**



State Pattern đóng gói trạng thái thành các lớp riêng biệt và ủy quyền cho Object biểu diễn một trạng thái hiện tại. Các trạng thái đều có chung một bộ các phương thức, tuy nhiên biểu hiện của mỗi phương thức là khác nhau ở hai trạng thái khác nhau. Khi thực hiện một phương thức, nó có thể dẫn đến thay đổi trạng thái, tức là thay đổi toàn bộ hành vi của Object.

Sơ đồ tổng quát UML:

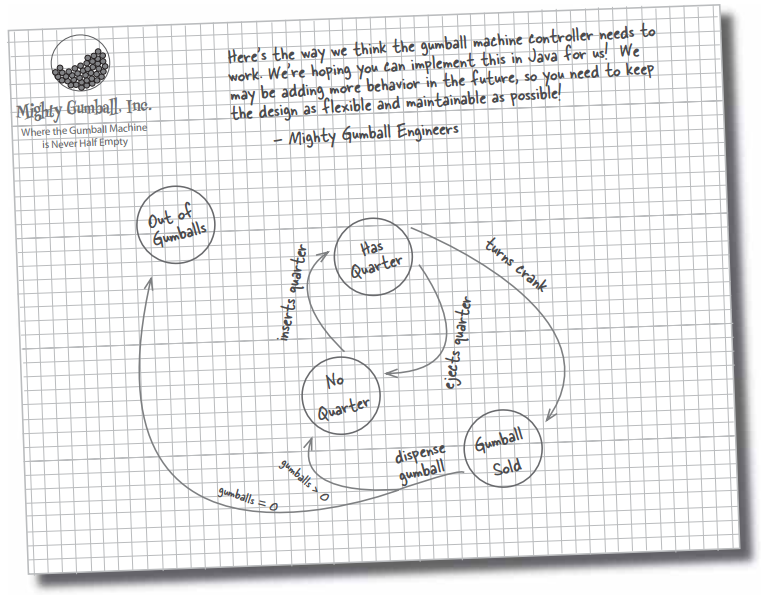


Đồng hành cùng Joe giải quyết yêu cầu từ công ty kẹo cao su Mighty Gumball. Họ yêu cầu Joe lập trình cho hoạt động của máy bán kẹo cao su.

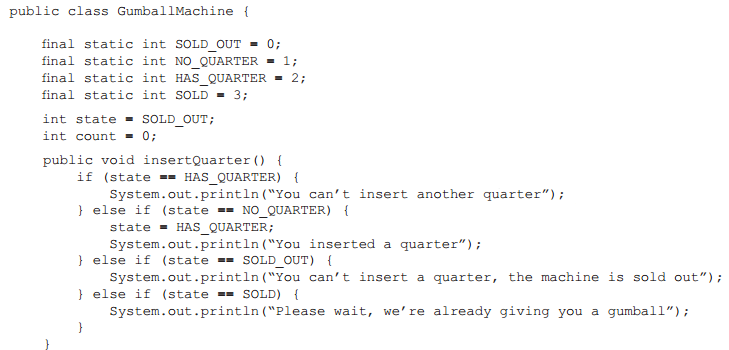
+ Từ phương diện người dùng có các hành động sau: insertQuarter, ejectQuarter, turnCrank và dispense.

+ Các trạng thái của gumballMachine: SoldOutState, NoQuarterState, HasQuarterState, SoldState.

Tương ứng với mỗi hành động của người dùng, tùy vào trạng thái hiện tại mà gumballMachine phải đưa ra phản ứng tương ứng. Ví dụ như, khi đang ở trạng thái *NoQuarterState*, người dùng *insertQuarter* thì máy phải báo người dùng đã cho xu vào và chuyển sang trạng thái *HasQuarterState*.

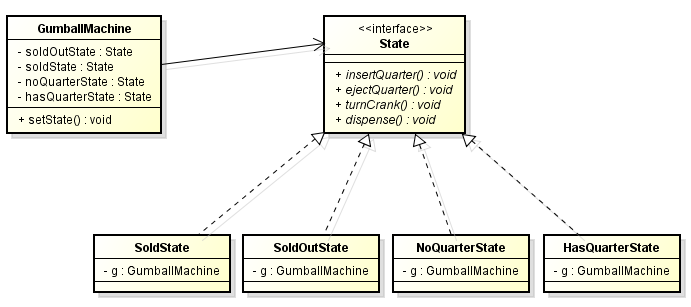


Cách đơn giản nhất để giải quyết yêu cầu là sử dụng một biến để lưu trữ trạng thái hiện tại curState. Khi người dùng thực hiện hành vi, sử dụng các câu lệnh if ... else ... để đưa ra phản ứng.



Hãy tưởng tượng những khó khăn khi công ty yêu cầu thêm bớt trạng thái và thay đổi hành vi cũ.

Áp dụng State Pattern, coi mỗi trạng thái là một Class, Joe thiết kế sơ đồ như sau:



Joe code để cho mọi người thấy cái hay của mô hình thiết kế này:

1. Interface State và các trạng thái.

Mỗi trạng thái sẽ định nghĩa tương ứng với 4 hành vi của người dùng sẽ thực hiện nhiệm vụ gì.

|  |
| --- |
| **public** **interface** State {  **public** **void** insertQuarter();  **public** **void** ejectQuarter();  **public** **void** turnCrank();  **public** **void** dispense();  } |
| **public** **class** SoldOutState **implements** State {  GumballMachine gumballMachine;    **public** SoldOutState(GumballMachine gumballMachine) {  **this**.gumballMachine = gumballMachine;  }    **public** **void** insertQuarter() {  System.***out***.println("You can't insert a quarter. Sold out!");  }  **public** **void** ejectQuarter() {  System.***out***.println("Sold out. You haven't insert a quarter");  }  **public** **void** turnCrank() {  System.***out***.println("Sold out. You turned but there's no gumball");  }  **public** **void** dispense() {  System.***out***.println("Sold out. No gumball dispense");  }  } |
| **public** **class** SoldState **implements** State {  GumballMachine gumballMachine;    **public** SoldState(GumballMachine gumballMachine) {  **this**.gumballMachine = gumballMachine;  }    **public** **void** insertQuarter() {  System.***out***.println("Please wait, gumball is coming out");  }  **public** **void** ejectQuarter() {  System.***out***.println("Sorry, you already turned the crank");  }  **public** **void** turnCrank() {  System.***out***.println("Turning twice doesn't get you another gumball!");  }  **public** **void** dispense() {  gumballMachine.releaseBall();  **if** (gumballMachine.numberBall > 0) {  gumballMachine.setState(gumballMachine.noQuarterState);  } **else** {  System.***out***.println("This is the last gumball!");  gumballMachine.setState(gumballMachine.soldOutState);  }  }  } |
| **public** **class** NoQuarterState **implements** State {  GumballMachine gumballMachine;    **public** NoQuarterState(GumballMachine gumballMachine) {  **this**.gumballMachine = gumballMachine;  }    **public** **void** insertQuarter() {  System.***out***.println("You inserted a quarter");  gumballMachine.setState(gumballMachine.hasQuarterState);  }  **public** **void** ejectQuarter() {  System.***out***.println("You haven't inserted a quarter");  }  **public** **void** turnCrank() {  System.***out***.println("You turned, but there's no quarter");  }  **public** **void** dispense() {  System.***out***.println("You need to pay first");  }  } |
| **public** **class** HasQuarterState **implements** State {  GumballMachine gumballMachine;    **public** HasQuarterState(GumballMachine gumballMachine) {  **this**.gumballMachine = gumballMachine;  }    **public** **void** insertQuarter() {  System.***out***.println("You can't insert another quarter");  }  **public** **void** ejectQuarter() {  System.***out***.println("Quarter returned");  gumballMachine.setState(gumballMachine.noQuarterState);  }  **public** **void** turnCrank() {  System.***out***.println("You turned...");  gumballMachine.setState(gumballMachine.soldState);  }  **public** **void** dispense() {  System.***out***.println("No gumball dispensed. Turn crank first");  }  } |

2. Class GumballMachine

Chứa instance của tất cả các trạng thái và một instate *curState* để lưu trạng thái hiện tại. Khi cần thay đổi trạng thái thì gọi phương thức *setState()* để thiết lập lại *curState*. Đồng thời nó cũng định nghĩa 4 hành vi người dùng bằng cách gọi đến phương thức tương ứng trong *curState*.

|  |
| --- |
| **public** **class** GumballMachine {  State soldOutState;  State noQuarterState;  State hasQuarterState;  State soldState;    State curState = soldOutState;  **int** numberBall = 0;    **public** GumballMachine(**int** numberGumballs) {  soldOutState = **new** SoldOutState(**this**);  noQuarterState = **new** NoQuarterState(**this**);  hasQuarterState = **new** HasQuarterState(**this**);  soldState = **new** SoldState(**this**);    **this**.numberBall = numberGumballs;  **if** (numberBall > 0) {  curState = noQuarterState;  }  }    **public** **void** insertQuarter() {  curState.insertQuarter();  }    **public** **void** ejectQuarter() {  curState.ejectQuarter();  }    **public** **void** turnCrank() {  curState.turnCrank();  curState.dispense();  }    **public** **void** releaseBall() {  System.***out***.println("A gumball comes rolling out the slot...");  **if** (numberBall > 0) {  numberBall--;  }  }    **public** **void** setState(State state) {  curState = state;  }    **public** String toString() {  **return** "There are " + numberBall + " gumballs left in the Machine\n";  }  } |

3. Main class GumballMachineTestDrive và kết quả

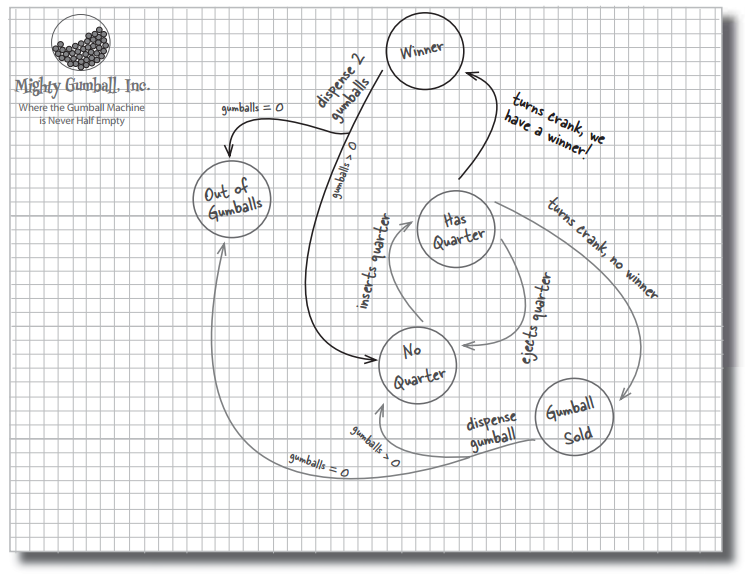
|  |
| --- |
| **public** **class** GumballMachineTestDrive {  **public** **static** **void** main(String[] args) {  GumballMachine gumballMachine = **new** GumballMachine(5);    System.***out***.println(gumballMachine);    System.***out***.println("------Try 1...");  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  System.***out***.println(gumballMachine);    System.***out***.println("------Try 2...");  gumballMachine.insertQuarter();  gumballMachine.ejectQuarter();  gumballMachine.turnCrank();  System.***out***.println(gumballMachine);    System.***out***.println("------Try 3...");  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  gumballMachine.ejectQuarter();  System.***out***.println(gumballMachine);    System.***out***.println("------Try 4...");  gumballMachine.insertQuarter();  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  gumballMachine.insertQuarter();  gumballMachine.turnCrank();  System.***out***.println(gumballMachine);  }  } |
| There are 5 gumballs left in the Machine  ------Try 1...  You inserted a quarter  You turned...  A gumball comes rolling out the slot...  There are 4 gumballs left in the Machine  ------Try 2...  You inserted a quarter  Quarter returned  You turned, but there's no quarter  You need to pay first  There are 4 gumballs left in the Machine  ------Try 3...  You inserted a quarter  You turned...  A gumball comes rolling out the slot...  You inserted a quarter  You turned...  A gumball comes rolling out the slot...  You haven't inserted a quarter  There are 2 gumballs left in the Machine  ------Try 4...  You inserted a quarter  You can't insert another quarter  You turned...  A gumball comes rolling out the slot...  You inserted a quarter  You turned...  A gumball comes rolling out the slot...  This is the last gumball!  You can't insert a quarter. Sold out!  Sold out. You turned but there's no gumball  Sold out. No gumball dispense  There are 0 gumballs left in the Machine |

**You know it was coming... a change request!**

Đã vào mùa khuyến mãi, Mighty Gumball Company yêu cầu Joe thêm hoạt động cho máy chọn ra người mua may mắn.

“Be a Winner! ONE in TEN get a FREE GUMBALL”.

Nghĩa là với mỗi lần người dùng mua kẹo, xác suất 1/10 họ sẽ được bonus thêm một viên kẹo nữa.



Hãy xem, Joe cần thay đổi những gì ở chương trình điều khiển gumballMachine hiện tại.

1. Thêm trạng thái WinnerState

Phương thức dispense sẽ gọi release 2 lần thay vì 1 lần như ở trạng thái HasQuarterState.

|  |
| --- |
| **public** **class** WinnerState **implements** State {  GumballMachine gumballMachine;    **public** WinnerState(GumballMachine gumballMachine) {  **this**.gumballMachine = gumballMachine;  }    **public** **void** insertQuarter() {  System.***out***.println("Winner! Don't insert another quarter");  }  **public** **void** ejectQuarter() {  System.***out***.println("Winner! Don't eject quarter");  }  **public** **void** turnCrank() {  System.***out***.println("Winner! Be patient. Don't turn crank");  }  **public** **void** dispense() {  System.***out***.println("YOU'RE A WINNER! You get 2 gumballs");    gumballMachine.releaseBall();  **if** (gumballMachine.numberBall == 0) {  System.***out***.println("But unluckly, there is only 1 lefts");  gumballMachine.setState(gumballMachine.soldOutState);  } **else** {  gumballMachine.releaseBall();  **if** (gumballMachine.numberBall > 0) {  gumballMachine.setState(gumballMachine.noQuarterState);  } **else** {  System.***out***.println("Oops, out of gumballs!");  gumballMachine.setState(gumballMachine.soldOutState);  }  }  }  } |

2. Bổ sung trạng thái này vào class GumballMachine.

Đừng quên bổ sung vào Constructor.

|  |
| --- |
| **public** **class** GumballMachine {  State winnerState;    **public** GumballMachine(**int** numberGumballs) {  winnerState = **new** WinnerState(**this**);  ... |

3. Thay đổi hoạt động của phương thức *turnCrank()* trong class HasQuarterState.

Đây là phương thức duy nhất cần phải thay đổi.

|  |
| --- |
| **public** **class** HasQuarterState **implements** State {  Random randomWinner = **new** Random(System.*currentTimeMillis*());  GumballMachine gumballMachine;  **public** **void** turnCrank() {  System.***out***.println("You turned...");  **int** winner = randomWinner.nextInt(10);  **if** ((winner == 0) && (gumballMachine.numberBall > 1)) {  gumballMachine.setState(gumballMachine.winnerState);  } **else** {  gumballMachine.setState(gumballMachine.soldState);  }  } |

4. Kết quả chạy lại main class GumballMachineTestDrive. Chúc may mắn!!!

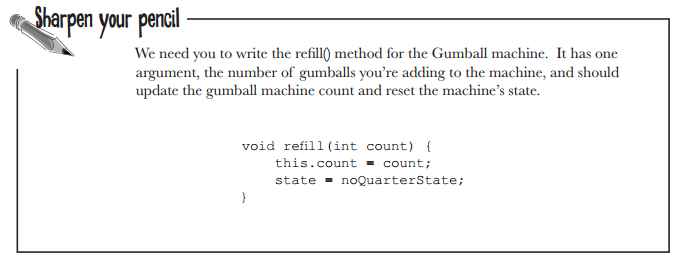
|  |
| --- |
| ------Try 3...  You inserted a quarter  You turned...  YOU'RE A WINNER! You get 2 gumballs  A gumball comes rolling out the slot...  A gumball comes rolling out the slot...  You inserted a quarter  You turned...  A gumball comes rolling out the slot...  You haven't inserted a quarter  There are 1 gumballs left in the Machine |

Tổng hợp, rút ra các ưu điểm to lớn của State Pattern như sau:

+ Khi một State muốn thay đổi hành vi, chỉ cần thay đổi phương thức trong chính State đó, không ảnh hưởng tới class GumballMachine hay các State khác.

+ Khi muốn thêm bớt một State, tất cả các việc phải làm là tạo State đó, thêm nó vào GumballMachine (2 dòng – định nghĩa state và constructor), thay đổi phương thức các State cũ có liên quan tới State mới. Ví dụ khi thêm **WinnerState**, cần thay đổi phương thức *turnCrank()* trong **HasQuarterState**.

+ Suy nghĩ, nếu công ty đưa thêm một phương thức mới, **refill()**, thực hiện thêm một số gumball cho máy thì những gì cần thay đổi?



**Phân biệt State Pattern và Strategy Pattern**

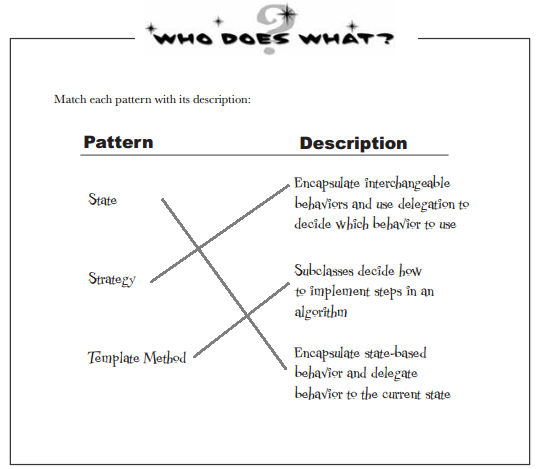
**Strategy**: Encapsulate interchangable behaviors and use deligation to decide which behavior to use. Cho phép object phối hợp các hành vi hoặc thuật toán khác nhau thông qua composition và delegation.

**State**: Encapsulate state-based behavior and delegate behavior to current state. Giúp các class thể hiện các hành vi khác nhau trong các trạng thái khác nhau.

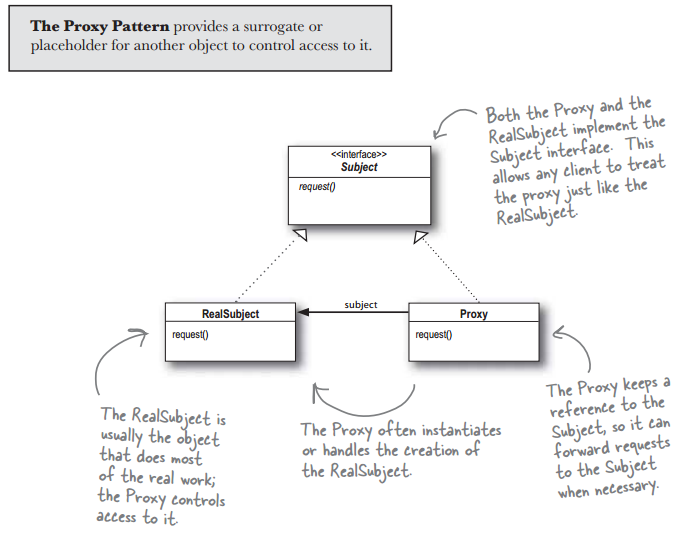
Hai pattern này có điểm tương đồng nhau, tuy nhiên cách client sử dụng composition và delegation là khác nhau.

+ Strategy khởi tạo một class bao gồm một số strategy object (*quackBehavior*, *flyBehavior*) và mỗi object áp dụng một vài hành vi khác nhau. Ví dụ trong quackBehavior có các hành vi: Realducks áp dụng real quack, rubber duck là squeack quack. Main class sẽ chịu trách nhiệm gọi và thay đổi các hành vi này; tự thân một hành vi không thể chuyển thành hành vi khác.

+ State: Định nghĩa một tập hợp các trạng thái. Khi class được khởi tạo, nó được gán một trạng thái để bắt đầu (*curState*). Khi một phương thức của trạng thái cũ được thực hiện, nó định nghĩa sẵn việc chuyển đổi sang trạng thái mới nếu cần thiết (state transition). Ví dụ khi gọi *insertQuarter()* thì trạng thái của curState được chuyển từ **NoQuarter** được chuyển sang trạng thái **HasQuarter**.



**Chapter 11. The Proxy Pattern – Controlling Object Access**



Trong Proxy Pattern, một class đại diện cho 1 chức năng của một lớp khác. Ta có thể hiểu, ta thuê một luật sư về đại diện cho mình, vị luật sư này có thể dùng các quyền của mình một cách gián tiếp để nói cho các người khác một cách chính xác vì luật sư am hiểu về pháp luật hơn mình.

**Khi nào dùng nó?**

* Khi bạn muốn bảo vệ quyền truy xuất vào các phương thức của thực thể.
* Bổ sung trước khi thực hiện phương thức của thực thể.
* Tạo đối tượng với chức năng được nâng cao theo yêu cầu.

**Có các loại Proxy nào?**

* Remote Proxy: quản lý giao tiếp giữa một client và một remote object.
* Virtual Proxy: điều khiển truy cập tới một object (thường object này khá “đắt” nếu khởi tạo nhiều)
* Protection Proxy: điều khiển truy cập tới các phương thức của một object dựa trên caller.
* Ngoài ra còn có nhiều loại khác như: caching proxy, synchoronization proxy, firewall proxy, copy-on-write proxy, ...

Cấu trúc của ***Proxy Pattern*** tương tự ***Decorator Pattern***, nhưng chúng khác nhau ở mục đích. Decorator Pattern có mục đích thêm hành vi cho một object, trong khi Proxy điều khiển các truy cập vào object.

**Dùng nó như thế nào?**

Ví dụ trong sách HeadFirst khá là phức tạp và đòi hỏi cài đặt nhiều phụ trợ, nên ta tham khảo các ví dụ đơn giản hơn ở các bài viết trên Internet.

Chúng ta tạo một *Image* interface và các class cụ thể áp dụng nó.

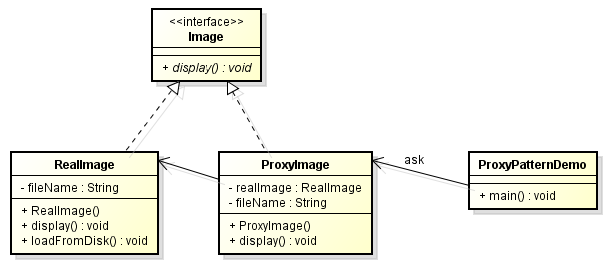
+ Interface *Image*: định nghĩa phương thức display() hiển thị hình ảnh.

+ Class *RealImage*: chứa một thuộc tính là fileName, là tên file ảnh của object. Khi khởi tạo, fileName sẽ được nhập vào và load từ ổ cứng. RealImage áp dụng interface Image, định nghĩa phương thức display.

+ *ProxyImage* là một lớp proxy giúp giảm truy cập bộ nhớ để tải ảnh (loadFromDisk) và hiển thị khi cần.

+ *ProxyPatternDemo*, là lớp main, sử dụng ProxyImage để tạo một đối tượng Image.

Sơ đồ UML



1. Tạo Interface Image

|  |
| --- |
| **public** **interface** Image {  **public** **void** display();  } |

2. Tạo lớp cụ thể RealImage và ProxyImage

|  |
| --- |
| **public** **class** RealImage **implements** Image {  **private** String fileName;    **public** RealImage(String fileName) {  **this**.fileName = fileName;  loadFromDisk(fileName);  }  @Override  **public** **void** display() {  System.***out***.println("Displaying " + fileName);  }    **private** **void** loadFromDisk(String fileName) {  System.***out***.println("Loading " + fileName);  }  } |
| **public** **class** ProxyImage **implements** Image {  **private** RealImage realImage;  **private** String fileName;    **public** ProxyImage(String fileName) {  **this**.fileName = fileName;  }  @Override  **public** **void** display() {  **if** (realImage == **null**) {  realImage = **new** RealImage(fileName);  }  realImage.display();  }    } |

3. Tạo lớp ProxyPatternDemo sử dụng ProxyImage để lấy đối tượng RealImage khi được yêu cầu.

|  |
| --- |
| **public** **class** ProxyPatternDemo {  **public** **static** **void** main(String[] args) {  Image image = **new** ProxyImage("test\_10mb.jpg");    // image will be loaded from disk  image.display();  System.***out***.println("");    // image will not be loaded from disk  image.display();  }  } |

4. Kết quả

Object Image “test\_10mb” chỉ cần được tải từ ổ đĩa lần ở lần đầu tiên.

|  |
| --- |
| Loading test\_10mb.jpg  Displaying test\_10mb.jpg  Displaying test\_10mb.jpg |

Ví dụ 2: Điều khiển phương thức đọc một file dựa vào User. Chỉ cho phép username là “Tung” thực hiện phương thức đọc file.

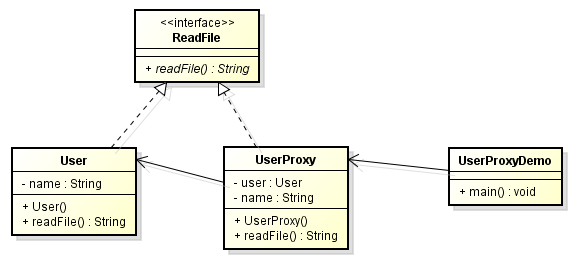
+ Interface ReadFile: khai báo phương thức readFile().

+ Class User: định nghĩa hoạt động của phương thức readFile() và trả về “File readed”.

+ Class UserProxy: Kiểm tra thuộc tính name có phải là “Tung” hay không và đưa ra phản ứng thích hợp.

+ Class UserProxy Demo: main class. Tạo các object UserProxy có name khác nhau và gọi readFile().

Sơ đồ UML



1. Interface ReadFile

|  |
| --- |
| **public** **interface** ReadFile {  **public** String readfile();  } |

2. Concrete class User và UserProxy

|  |
| --- |
| **public** **class** User **implements** ReadFile {  **private** String name;    **public** User(String name) {  **this**.name = name;  }  @Override  **public** String readfile() {  **return** "File readed";  }    } |
| **public** **class** UserProxy **implements** ReadFile {  **private** User user;  **private** String name;    **public** UserProxy(String name) {  **this**.name = name;  }  @Override  **public** String readfile() {  **if** (name.equals("Tung")) {  user = **new** User(name);  **return** user.readfile();  }  **return** "You can't read file";  }  } |

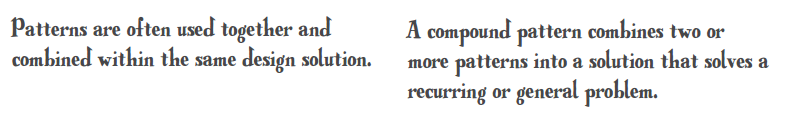
3. Main class UserProxyDemo

|  |
| --- |
| **public** **class** UserProxyDemo {  **public** **static** **void** main(String[] args) {    UserProxy user1 = **new** UserProxy("Tung");  System.***out***.println(user1.readfile());  UserProxy user2 = **new** UserProxy("Thao");  System.***out***.println(user2.readfile());  }  } |

4. Kết quả

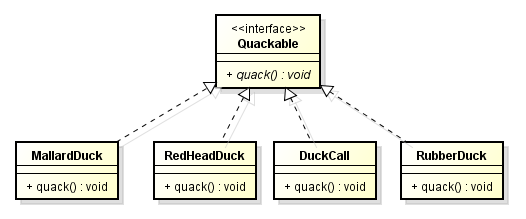
|  |
| --- |
| File readed  You can't read file |

**Chapter 12. The Compound Pattern – Patterns of Patterns**



Quay trở lại làm việc với vịt, lần này chúng ta tìm hiểu cách mà các pattern cùng hợp tác và tồn tại trong cùng một chương trình.

Đầu tiên, ***State Pattern*** – Mỗi loài vịt đều có một kiểu kêu, được định nghĩa bằng phương thức *quack()* nằm trong interface *Quackable*.

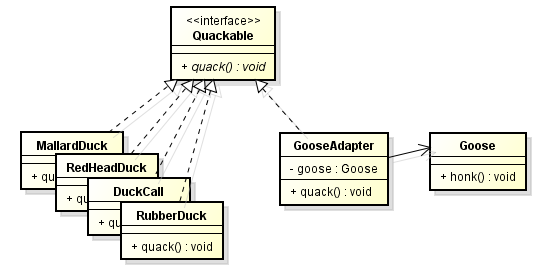


Mỗi class là một loài vịt implement Quackable định nghĩa tiếng kêu của riêng nó. MallardDuck và RedHeadDuck đều kêu “quack”, DuckCall là giả tiếng vịt kêu “Kweak”, còn RubberDuck là “Squeak”.

Ở chương trình chính DuckSimulator, khi cần thiết chúng ta chỉ đơn giản gọi duck.quack cho tất cả các instance.

|  |
| --- |
| **public** **void** simulate() {  Quackable mallardDuck = **new** MallardDuck();  Quackable redheadDuck = **new** RedHeadDuck();  Quackable duckCall = **new** DuckCall();  Quackable rubberDuck = **new** RubberDuck();    simulate(mallardDuck);  simulate(redheadDuck);  simulate(duckCall);  simulate(rubberDuck);  }    **public** **void** simulate(Quackable duck) {  duck.quack();  } |

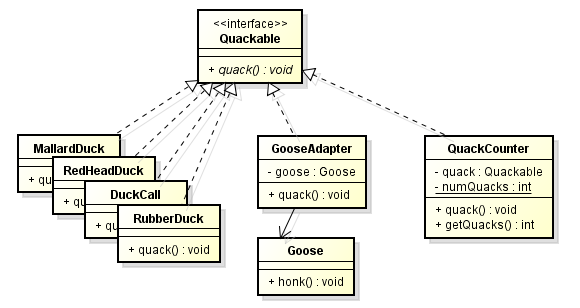
Thứ hai, ***Adapter Pattern*** – Chương trình được thêm vào một class *Goose*, loài ngỗng lại sử dụng phương thức *honk()* để biểu thị tiếng kêu. Lúc này, để đồng nhất các class về cùng một phương thức quack() mà không thay đổi class Goose, chúng ta tạo một class mới *GooseAdapter*.



Trong DuckSimulator, object GooseAdapter được xem như là một instance của Quackable và đối xử giống như những instance khác.

|  |
| --- |
| **public** **void** simulate() {  Quackable gooseDuck = **new** GooseAdapter(**new** Goose());  simulate(gooseDuck);  } |

Thứ ba, ***Decorator Pattern*** – Các nhà vịt học (Quackologist) muốn đếm số lần kêu của riêng loài vịt (không tính ngỗng), và không được thay đổi các class Duck có sẵn.

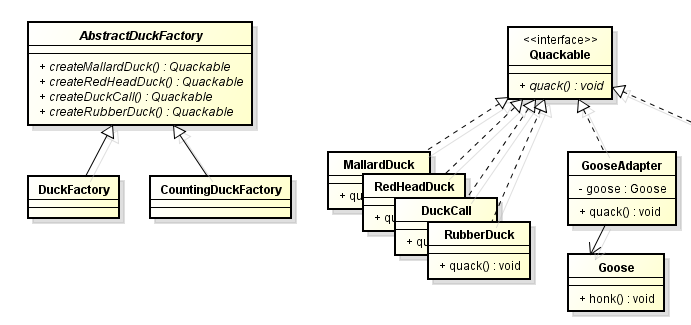


Chúng ta thêm một class QuackCounter, trong đó có chứa một thuộc tính kiểu statics là *numQuacks*. Mỗi khi phương thức quack() được gọi, numQuacks sẽ được tăng lên một. Phương thức *getQuacks()* trả về giá trị numQuacks hiện tại.

Thay đổi một chút ở chương trình DuckSimulator, mỗi Duck object được bọc trong một object QuackCounter:

|  |
| --- |
| **public** **void** simulate() {  Quackable mallardDuck = **new** QuackCounter(**new** MallardDuck());  Quackable redheadDuck = **new** QuackCounter(**new** RedHeadDuck());  Quackable duckCall = **new** QuackCounter(**new** DuckCall());  Quackable rubberDuck = **new** QuackCounter(**new** RubberDuck());  Quackable gooseDuck = **new** GooseAdapter(**new** Goose());  System.***out***.println("The ducks quacked:" + QuackCounter.*getQuacks*() + " times");  } |

Thứ tư, ***Abtract Factory Pattern*** – hãy để ý cách khởi tạo các loài vịt ở trên, rất khó để quan sát và không đóng kín (user biết cách khởi tạo). Chúng ta sẽ đóng kín nó bằng một abstract factory class.

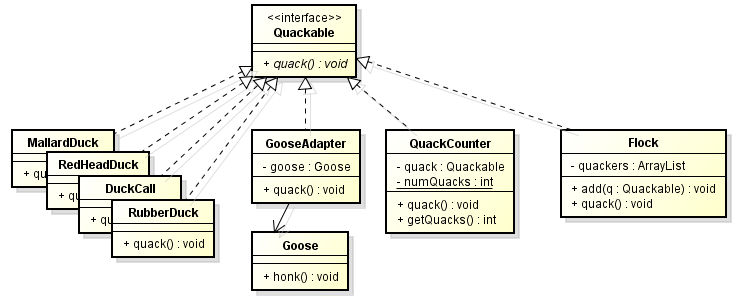


Ở đây có 2 class Factory là *DuckFactory*, tạo các loại vịt không bao gồm QuackCounter và *CountingDuckFactory* chứa QuackCounter.

Sau khi sửa code ở DuckSimulator, ta thấy cách khởi tạo vịt được giấu kín:

|  |
| --- |
| **public** **class** DuckSimulator {  **public** **static** **void** main(String[] args) {  DuckSimulator simulator = **new** DuckSimulator();  AbstractDuckFactory duckFactory = **new** CountingDuckFactory();    simulator.simulate(duckFactory);  }  **public** **void** simulate(AbstractDuckFactory duckFactory) {  Quackable mallardDuck = duckFactory.createMallardDuck();  Quackable redheadDuck = duckFactory.createRedHeadDuck();  Quackable duckCall = duckFactory.createDuckCall();  Quackable rubberDuck = duckFactory.createRubberDuck();  Quackable gooseDuck = **new** GooseAdapter(**new** Goose()); |

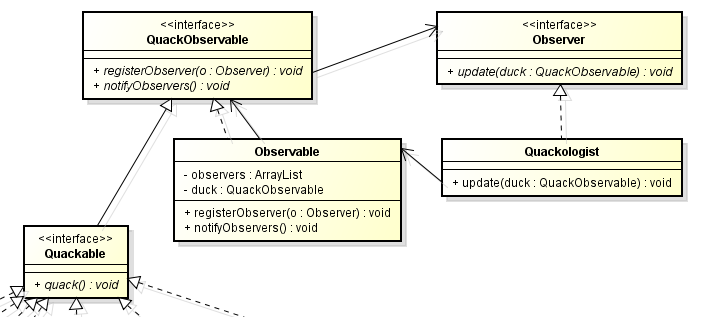
Thứ năm, ***Iterator và Composite Pattern*** – Gom tất cả vịt và ngỗng vào một cây quản lý duy nhất. Hiện tại các instance đang được nằm riêng rẽ nhau. Chúng ta sử dụng composite để tập hợp chúng lại.



Thêm code vào DuckSimulator, chúng ta còn có thể tạo thêm một nhánh riêng của đàn MallardDuck:

|  |
| --- |
| **public** **void** simulate(AbstractDuckFactory duckFactory) {  /\* Iterator and Composite \*/  Flock flockofDucks = **new** Flock();  flockofDucks.add(redheadDuck);  flockofDucks.add(duckCall);  flockofDucks.add(rubberDuck);  flockofDucks.add(gooseDuck);    Flock flockofMallards = **new** Flock();  Quackable mallardOne = duckFactory.createMallardDuck();  Quackable mallardTwo = duckFactory.createMallardDuck();  Quackable mallardThree = duckFactory.createMallardDuck();  Quackable mallardFour = duckFactory.createMallardDuck();    flockofMallards.add(mallardOne);  flockofMallards.add(mallardTwo);  flockofMallards.add(mallardThree);  flockofMallards.add(mallardFour);  flockofDucks.add(flockofMallards);    System.***out***.println("\nDuck Simulator: Whole Flock Simulation");  simulate(flockofDucks);  System.***out***.println("\nDuck Simulator: Mallard Duck Simulation");  simulate(flockofMallards);  } |

Thứ 6, ***Observer Pattern*** – Giúp các chuyên gia Quackologist nhận được thông báo khi vịt kêu.



Thêm ở main class DuckSimulator rất đơn giản:

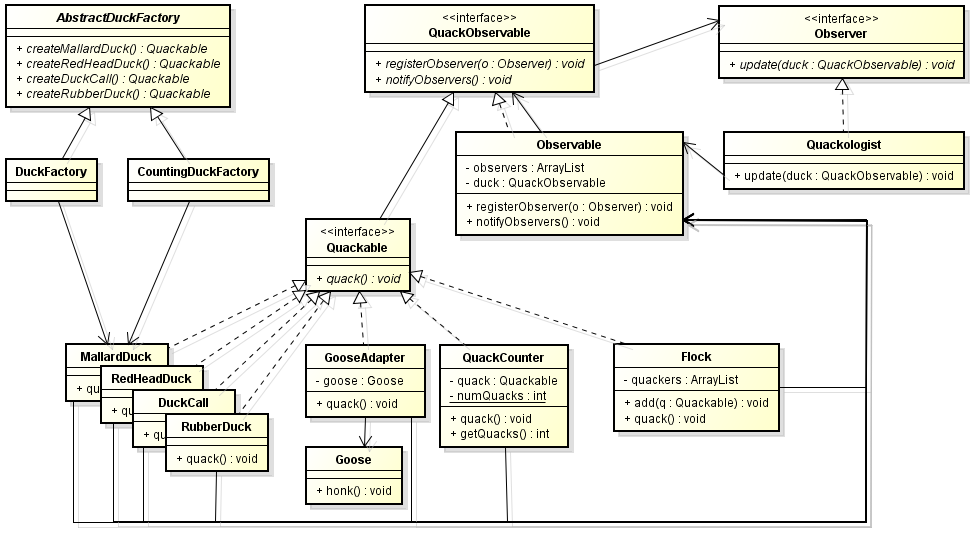
|  |
| --- |
| **public** **void** simulate(AbstractDuckFactory duckFactory) {  // Create duck factory and ducks    // Create flocks    // Observer  Quackologist quackologist = **new** Quackologist();  flockofDucks.registerObserver(quackologist); |

Mỗi concrete class của duck hay goose adapter đều chứa một thuộc tính *Observable* và implement 2 phương thức *registerObserver()* và *notifyObservers()*. Gọi flockofDucks.registerObserver(quackologist), tức là chúng ta đang duyệt cả cây flockofDucks và gọi từng phương thức của mỗi nốt trên cây đó, đồng thời đăng kí một Observer là *Quackologist*.

Mỗi lần phương thức quack() được thực hiện, nó sẽ gọi hàm notifyObservers(), từ đó Quackologist sẽ nhận được thông tin.

Như vậy chúng ta đã sử dụng 6 pattern (Strategy, Adapter, Decorator, Abstract Factory, Composite, và Observer) để tạo nên Compound pattern nhằm giải quyết các yêu cầu của chương trình Duck.

Trước khi đi đến code cụ thể, chúng ta theo dõi sơ đồ tổng quát một lần nữa:



Code

1. Quackable interface và các Duck class

|  |
| --- |
| **public** **interface** Quackable **extends** QuackObservable {  **public** **void** quack();  } |
| **public** **class** MallardDuck **implements** Quackable {  Observable observable;    **public** MallardDuck() {  observable = **new** Observable(**this**);  }    @Override  **public** **void** quack() {  System.***out***.println("Quack");  notifyObservers();  }  @Override  **public** **void** registerObserver(Observer observer) {  observable.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  observable.notifyObservers();  }  } |
| **public** **class** RedHeadDuck **implements** Quackable {  Observable observable;    **public** RedHeadDuck() {  observable = **new** Observable(**this**);  }    @Override  **public** **void** quack() {  System.***out***.println("Quack");  notifyObservers();  }  @Override  **public** **void** registerObserver(Observer observer) {  observable.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  observable.notifyObservers();  }    } |
| **public** **class** DuckCall **implements** Quackable {  Observable observable;    **public** DuckCall() {  observable = **new** Observable(**this**);  }    @Override  **public** **void** quack() {  System.***out***.println("Kwak");  notifyObservers();  }  @Override  **public** **void** registerObserver(Observer observer) {  observable.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  observable.notifyObservers();  }  } |
| **public** **class** RubberDuck **implements** Quackable {  Observable observable;    **public** RubberDuck() {  observable = **new** Observable(**this**);  }    @Override  **public** **void** quack() {  System.***out***.println("Squeak");  notifyObservers();  }  @Override  **public** **void** registerObserver(Observer observer) {  observable.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  observable.notifyObservers();  }  } |

2. Class Goose và Goose Adapter

|  |
| --- |
| **public** **class** Goose {  **public** **void** honk() {  System.***out***.println("Honk");  }  } |
| **public** **class** GooseAdapter **implements** Quackable {  Goose goose;  Observable observable;    **public** GooseAdapter(Goose goose) {  **this**.goose = goose;  observable = **new** Observable(**this**);  }    @Override  **public** **void** quack() {  goose.honk();  notifyObservers();  }  @Override  **public** **void** registerObserver(Observer observer) {  observable.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  observable.notifyObservers();  }  } |

3. Class QuackCounter (Decorator)

|  |
| --- |
| **public** **class** QuackCounter **implements** Quackable {  Quackable duck;  **static** **int** *numberofQuacks*;  **public** QuackCounter(Quackable duck) {  **this**.duck = duck;  }    @Override  **public** **void** quack() {  duck.quack();  *numberofQuacks*++;  }    **public** **static** **int** getQuacks() {  **return** *numberofQuacks*;  }  @Override  **public** **void** registerObserver(Observer observer) {  duck.registerObserver(observer);  }  @Override  **public** **void** notifyObservers() {  duck.notifyObservers();  }  } |

4. AbstractDuckFactory và 2 subclass DuckFactory và CountingDuckFactory

|  |
| --- |
| **public** **abstract** **class** AbstractDuckFactory {  **public** **abstract** Quackable createMallardDuck();  **public** **abstract** Quackable createRedHeadDuck();  **public** **abstract** Quackable createDuckCall();  **public** **abstract** Quackable createRubberDuck();  } |
| **public** **class** DuckFactory **extends** AbstractDuckFactory {  @Override  **public** Quackable createMallardDuck() {  **return** **new** MallardDuck();  }  @Override  **public** Quackable createRedHeadDuck() {  **return** **new** RedHeadDuck();  }  @Override  **public** Quackable createDuckCall() {  **return** **new** DuckCall();  }  @Override  **public** Quackable createRubberDuck() {  **return** **new** RubberDuck();  }  } |
| **public** **class** CountingDuckFactory **extends** AbstractDuckFactory {  @Override  **public** Quackable createMallardDuck() {  **return** **new** QuackCounter(**new** MallardDuck());  }  @Override  **public** Quackable createRedHeadDuck() {  **return** **new** QuackCounter(**new** RedHeadDuck());  }  @Override  **public** Quackable createDuckCall() {  **return** **new** QuackCounter(**new** DuckCall());  }  @Override  **public** Quackable createRubberDuck() {  **return** **new** QuackCounter(**new** RubberDuck());  }    } |

5. Class Flock (Composite)

|  |
| --- |
| **public** **class** Flock **implements** Quackable {  ArrayList quackers = **new** ArrayList();    **public** **void** add(Quackable quacker) {  quackers.add(quacker);  }  @Override  **public** **void** quack() {  Iterator iterator = quackers.iterator();  **while** (iterator.hasNext()) {  Quackable quacker = (Quackable) iterator.next();  quacker.quack();  }  }  @Override  **public** **void** registerObserver(Observer observer) {  Iterator iterator = quackers.iterator();  **while** (iterator.hasNext()) {  Quackable quacker = (Quackable) iterator.next();  quacker.registerObserver(observer);  }  }  @Override  **public** **void** notifyObservers() { }  // Each Quackable has its own notification,  // so Flock doesn't have to worry about it.    } |

6. Interface QuackObservable, Observer và class Observable và Quackologist

|  |
| --- |
| **public** **interface** QuackObservable {  **public** **void** registerObserver(Observer observer);  **public** **void** notifyObservers();  } |
| **public** **class** Observable **implements** QuackObservable {  ArrayList observers = **new** ArrayList();  QuackObservable duck;    **public** Observable(QuackObservable duck) {  **this**.duck = duck;  }  @Override  **public** **void** registerObserver(Observer observer) {  observers.add(observer);  }  @Override  **public** **void** notifyObservers() {  Iterator iterator = observers.iterator();  **while** (iterator.hasNext()) {  Observer observer = (Observer) iterator.next();  observer.update(duck);  }  }    } |
| **public** **interface** Observer {  **public** **void** update(QuackObservable duck);  } |
| **public** **class** Quackologist **implements** Observer {  **public** **void** update(QuackObservable duck) {  System.***out***.println("Quackologist: " + duck + " just quacked.");  }  } |

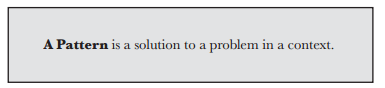
7. Main class DuckSimulator

|  |
| --- |
| **public** **class** DuckSimulator {  **public** **static** **void** main(String[] args) {  DuckSimulator simulator = **new** DuckSimulator();  AbstractDuckFactory duckFactory = **new** CountingDuckFactory();    simulator.simulate(duckFactory);  }  **public** **void** simulate(AbstractDuckFactory duckFactory) {  Quackable mallardDuck = duckFactory.createMallardDuck();  Quackable redheadDuck = duckFactory.createRedHeadDuck();  Quackable duckCall = duckFactory.createDuckCall();  Quackable rubberDuck = duckFactory.createRubberDuck();  Quackable gooseDuck = **new** GooseAdapter(**new** Goose());    System.***out***.println("\nDuck Simulator");  System.***out***.println("With Compound Pattern: S, A, D, F, C, O");    //Quackable mallardDuck = new QuackCounter(new MallardDuck());  //Quackable redheadDuck = new QuackCounter(new RedHeadDuck());  //Quackable duckCall = new QuackCounter(new DuckCall());  //Quackable rubberDuck = new QuackCounter(new RubberDuck());  //Quackable gooseDuck = new GooseAdapter(new Goose());    //simulate(mallardDuck);  //simulate(redheadDuck);  //simulate(duckCall);  //simulate(rubberDuck);  //simulate(gooseDuck);    /\* Iterator and Composite \*/  Flock flockofDucks = **new** Flock();  flockofDucks.add(redheadDuck);  flockofDucks.add(duckCall);  flockofDucks.add(rubberDuck);  flockofDucks.add(gooseDuck);    Flock flockofMallards = **new** Flock();  Quackable mallardOne = duckFactory.createMallardDuck();  Quackable mallardTwo = duckFactory.createMallardDuck();  Quackable mallardThree = duckFactory.createMallardDuck();  Quackable mallardFour = duckFactory.createMallardDuck();    flockofMallards.add(mallardOne);  flockofMallards.add(mallardTwo);  flockofMallards.add(mallardThree);  flockofMallards.add(mallardFour);    flockofDucks.add(flockofMallards);      // Observer  Quackologist quackologist = **new** Quackologist();  flockofDucks.registerObserver(quackologist);    System.***out***.println("\nDuck Simulator: Whole Flock Simulation");  simulate(flockofDucks);    System.***out***.println("\nDuck Simulator: Mallard Duck Simulation");  simulate(flockofMallards);    System.***out***.println("\nDuck Simulator: With QuackCounter Decorator");  System.***out***.println("The ducks quacked:" + QuackCounter.*getQuacks*() + " times");  }    **public** **void** simulate(Quackable duck) {  duck.quack();  }  } |

Kết quả khi chạy chương trình

|  |
| --- |
| Duck Simulator  With Strategy, Adapter, Decorator, Factory, Composite and Observer  Duck Simulator: Whole Flock Simulation  Quack  Quackologist: RedHeadDuck@b44e46 just quacked.  Kwak  Quackologist: DuckCall@104e3b2 just quacked.  Squeak  Quackologist: RubberDuck@15f550a just quacked.  Honk  Quackologist: GooseAdapter@6b2d4a just quacked.  Quack  Quackologist: MallardDuck@91993f just quacked.  Quack  Quackologist: MallardDuck@4af82c just quacked.  Quack  Quackologist: MallardDuck@19619aa just quacked.  Quack  Quackologist: MallardDuck@ac736f just quacked.  Duck Simulator: Mallard Duck Simulation  Quack  Quackologist: MallardDuck@91993f just quacked.  Quack  Quackologist: MallardDuck@4af82c just quacked.  Quack  Quackologist: MallardDuck@19619aa just quacked.  Quack  Quackologist: MallardDuck@ac736f just quacked.  Duck Simulator: With QuackCounter Decorator  The ducks quacked: 11 times |

**Chapter 13. Patterns in the Real World**



Một **context** là tình huống để áp dụng pattern.

Một **problem** ám chỉ đến mục tiêu mà bạn cố đạt được trong context đó.

Một **solution** là kết quả thu được khi bạn thiết kế để ai cũng có thể áp dụng và đạt mục tiêu.

Ví dụ:

Context: Tôi vẫn cắm chìa khóa trong xe ô tô khi đi ra ngoài và đóng cửa.

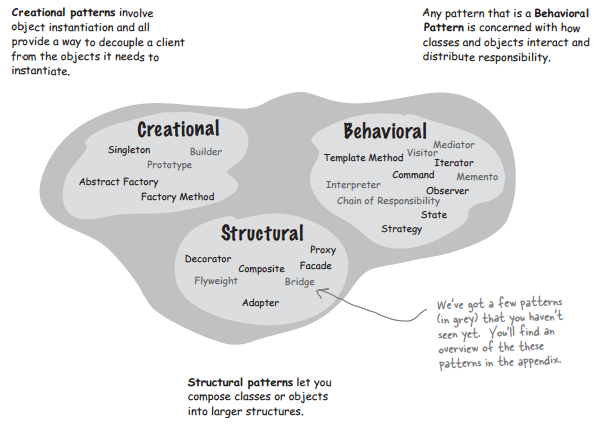
Problem: Làm thế nào để tôi có thể đến công ty đúng giờ?

Solution: Đập vỡ cửa kính ô tô, mở cửa và vào xe, khởi động động cơ và lái xe đến công ty.

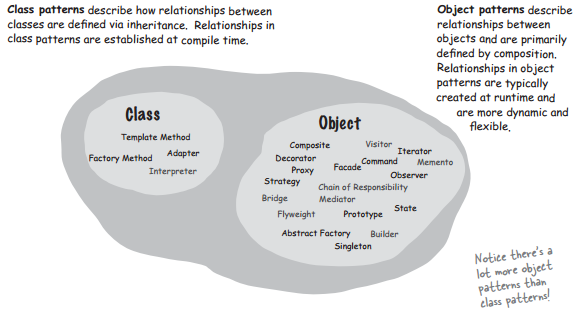
**Tổ chức phân loại Design Pattern**

Chia các pattern thành 3 nhóm chính: Creational, Structural, và Behavioral.

Có thể bạn thấy một pattern có thể thuộc nhiều hơn một nhóm, nhưng có thể xếp như sau:



Hoặc có thể chia thành nhóm làm việc với class hoặc object:

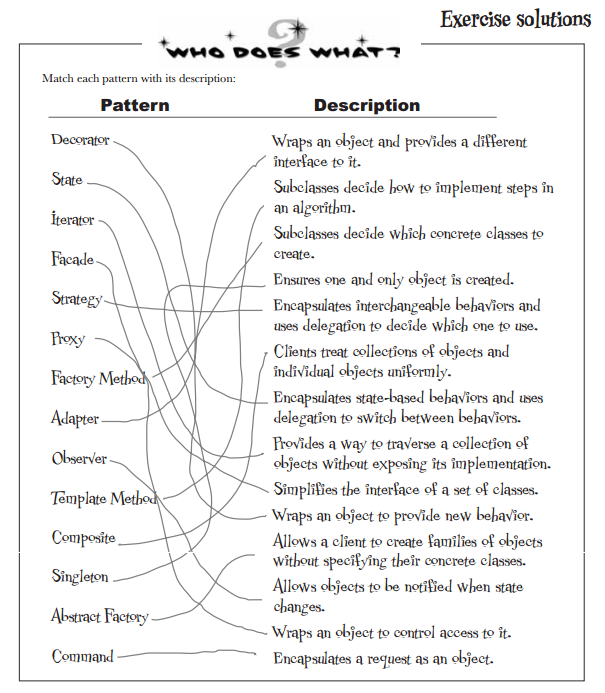


**Thinking in Patterns: Keep it simple (KISS)**

Design Pattern không phải là một công cụ. Hãy để nó đến tự nhiên khi bạn thiết kế và tìm cách tối ưu thiết kế của bạn nếu cần thiết.

|  |  |  |
| --- | --- | --- |
|  |  |  |

Tổng kết định nghĩa design pattern



**Phụ lục 1. Tại sao sử dụng Design Pattern?**

Nguồn: http://laptrinh.vn/d/4346-design-pattern-la-gi-huong-dan-design-pattern.html

Design pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình phát triển đã qua kiểm nghiệm. Thiết kế phần mềm hiệu quả đòi hỏi phải cân nhắc các vấn đề nảy sinh trong quá trình hiện thực hóa (implementation). Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, đồng thời dễ dàng nâng cấp, bảo trì về sau.

Nếu bạn quy ước đặt tên tốt, một mẫu thiết kế có thể được một lập trình viên khác dễ dàng nhận ra.

**Khi nào nên sử dụng design pattern?**

Đó là khi bạn muốn giữ cho chương trình của mình thực sự đơn giản. Việc sử dụng các design pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ các cách giải quyết cho những vấn đề đã có lời giải.

Cuốn *“Head First Design Patterns”* cung cấp cái nhìn tổng quan về design pattern. Hệ thống các mẫu design pattern hiện có **23 mẫu** được định nghĩa trong cuốn *“Design Pattern Elements of Reusable Object Oriented Software”*. Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại. Hệ thống các mẫu design pattern được chia thành 3 nhóm: *nhóm Creational (5 mẫu)*, *nhóm Structural (7 mẫu)*, và *nhóm Behavioral (11 mẫu)*.

|  |  |  |
| --- | --- | --- |
| STT | Tên | Mục đích |
| **Nhóm Creational (nhóm kiến tạo)** | | |
| 1 | Abstract Factory | Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần quy định lớp hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng.  Tần suất sử dụng: cao |
| 2 | Builder | Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau.  Tần suất sử dụng: trung bình thấp |
| 3 | Factory Method | Định nghĩa interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng. Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con.  Tần suất sử dụng: cao |
| 4 | Prototype | Quy định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này.  Tần suất sử dụng: trung bình |
| 5 | Singleton | Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.  Tần suất sử dụng: trung bình cao |
| **Nhóm Structural (Nhóm cấu trúc)** | | |
| 6 | Adapter | Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng thích hợp.  Tần suất sử dụng: cao trung bình |
| 7 | Bridge | Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt; mục đích để cả hai bộ phần (ngữ nghĩa và cài đặt) có thể thay đổi độc lập với nhau.  Tần suất sử dụng: trung bình |
| 8 | Composite | Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách đồng nhất.  Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau => khả năng tổng quát hóa trong code của client => dễ phát triển, nâng cấp, bảo trì  Tần suất sử dụng: cao trung bình |
| 9 | Decorator | Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) ngay trong lúc chạy (runtime)  Tần suất sử dụng: trung bình. |
| 10 | Facade | Cung cấp một interface thuần nhất cho một tập hợp các interface trong một hệ thống con. Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.  Tần suất sử dụng: cao |
| 11 | Flyweight | Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng “cỡ nhỏ” (chẳng hạng paragraph, dòng, cột, ký tự...).  Tần suất sử dụng: thấp |
| 12 | Proxy | Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thể gọi là proxy.  Tần suất sử dụng: cao trung bình |
| **Nhóm Behavioral (nhóm tương tác)** | | |
| 13 | Chain of  Responsibility | Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp;  Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cững giữa phần tử gửi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có cơ hội xử lý request.  Liên kết các đối tượng nhận request thành 1 dây chuyền rồi pass request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.  Tần suất sử dụng: trung bình thấp |
| 14 | Command | Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gửi đi như các đối tượng. Đóng gói request vào trong một Object, nhờ đó có thể thông số hóa chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo...  Tần suất sử dụng: cao trung bình |
| 15 | Interpreter | Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ  Tần suất sử dụng: thấp |
| 16 | Iterator | Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array,...) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.  Tần suất sử dụng: cao |
| 17 | Mediator | Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.  Tần suất sử dụng: trung bình thấp |
| 18 | Memento | Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.  Tần suất sử dụng: thấp |
| 19 | Observer | Định nghĩa sự phụ thuộc một-nhiều (one-many) giữa các đối tượng sao cho khi một đối tượng thay đổi thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo  Tần suất sử dụng: cao |
| 20 | State | Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi, ta có cảm giác như class của đối tượng bị thay đổi.  Tần suất sử dụng: trung bình |
| 21 | Strategy | Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán. Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng.  Tần suất sử dụng: cao trung bình |
| 22 | Template method | Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.  Tần suất sử dụng: trung bình |
| 23 | Visitor | Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.  Tấn suất sử dụng: thấp |

**Phụ lục 2: Tổng hợp các nguyên tắc thiết kế**

