1. Abstract Factory 抽象工廠

Provide an interface for creating families of related or
dependent objects without specifying their concrete classes.

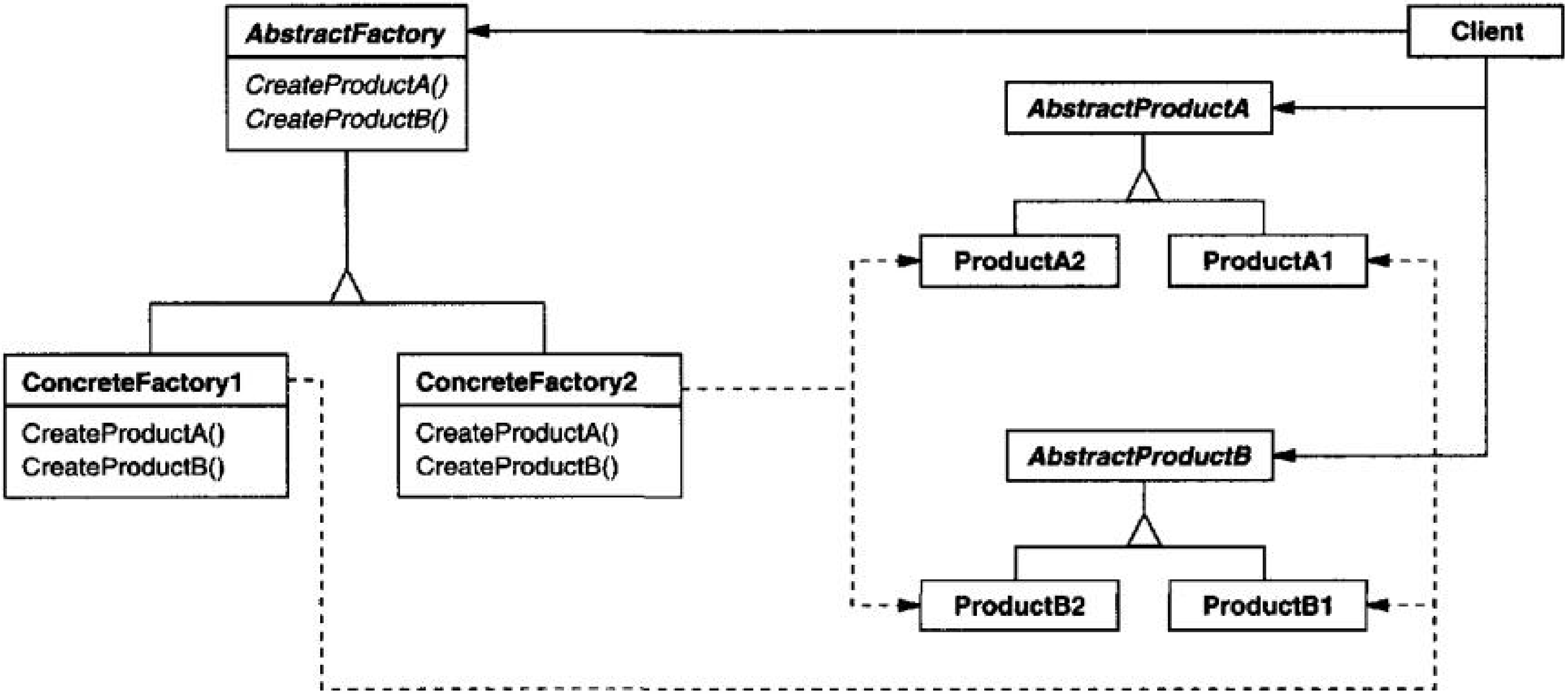
提供用於創建相關或從屬對象族的接口，而無需指定其具體類。

相似產品 不同工廠生產 變成不同產品

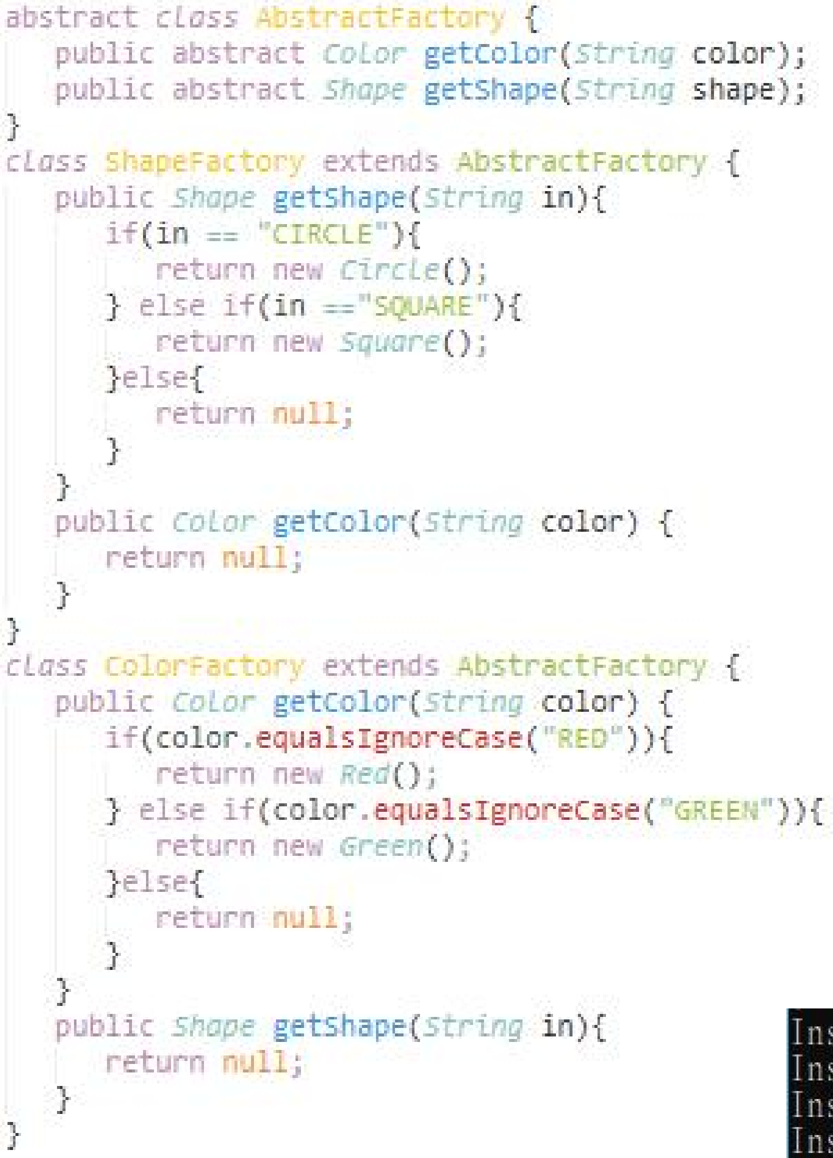
Nike的拖鞋(A1)
原本是Nike工廠生產(F1)

換成Addias工廠生產(F2)
變成addias拖鞋(A2)

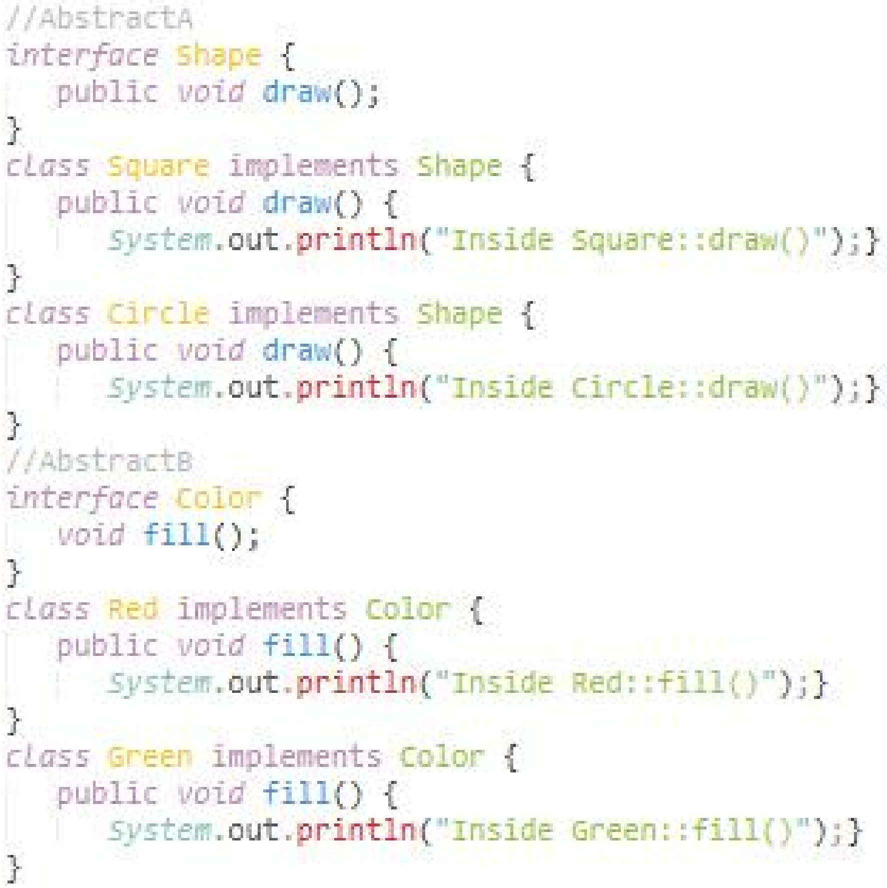
以圖來說有A、B兩種產品
Factory1生產的是產品A1、B1
Factory2生產的是產品A2、B2


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

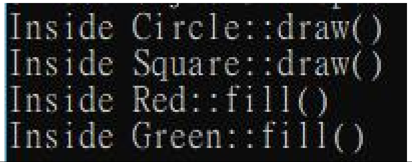
範例1.利用抽象工廠模式畫圖、著色



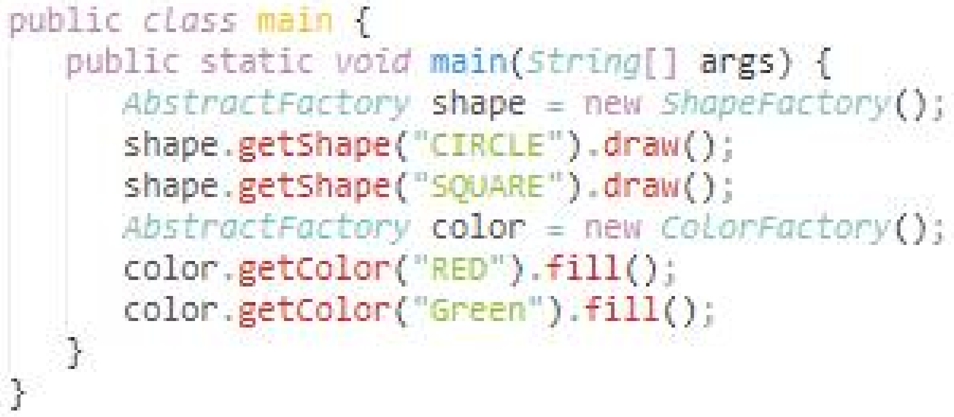
```java
abstract class AbstractFactory {
    public abstract Color getColor(String color);
    public abstract Shape getShape(String shape);
}
class ShapeFactory extends AbstractFactory {
    public Shape getShape(String in){
        if(in == "CIRCLE"){
            return new Circle();
        } else if(in =="SQUARE"){
            return new Square();
        }else{
            return null;
        }
    }
    public Color getColor(String color) {
        return null;
    }
}
class ColorFactory extends AbstractFactory {
    public Color getColor(String color) {
        if(color.equalsIgnoreCase("RED")){
            return new Red();
        } else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }else{
            return null;
        }
    }
    public Shape getShape(String in){
        return null;
    }
}
```



```java
//AbstractA
interface Shape {
    public void draw();
}
class Square implements Shape {
    public void draw() {
        System.out.println("Inside Square::draw()");}
}
class Circle implements Shape {
    public void draw() {
        System.out.println("Inside Circle::draw()");}
}
//AbstractB
interface Color {
    void fill();
}
class Red implements Color {
    public void fill() {
        System.out.println("Inside Red::fill()");}
}
class Green implements Color {
    public void fill() {
        System.out.println("Inside Green::fill()");}
}
```



```
Inside Circle::draw()
Inside Square::draw()
Inside Red::fill()
Inside Green::fill()
```



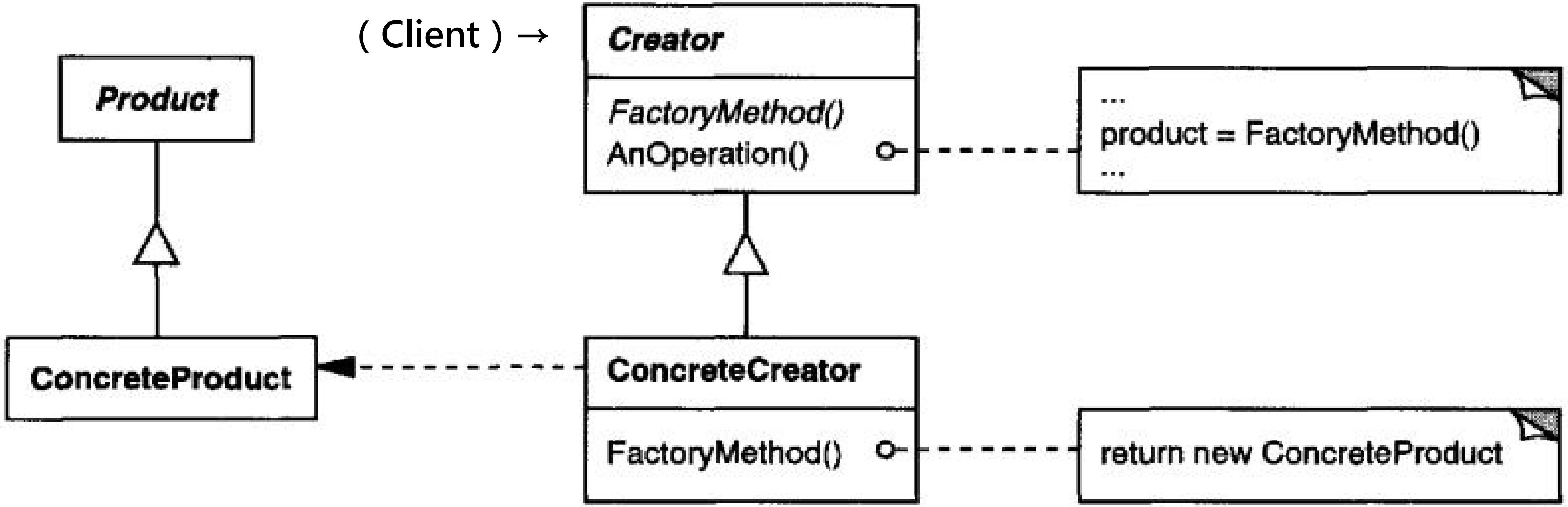
```
public class main {
    public static void main(String[] args) {
        AbstractFactory shape = new ShapeFactory();
        shape.getShape("CIRCLE").draw();
        shape.getShape("SQUARE").draw();
        AbstractFactory color = new ColorFactory();
        color.getColor("RED").fill();
        color.getColor("Green").fill();
    }
}
```

2. Factory Method 工廠方法

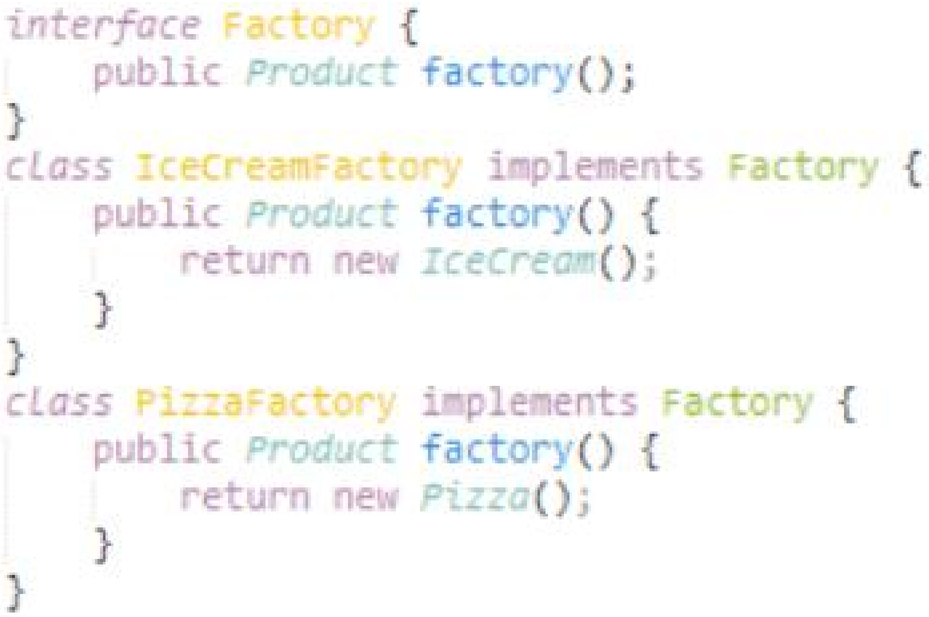
Define an interface for creating an object, but let subclasses decide which class to instantiate.
Factory Method lets a class defer instantiation to subclasses.

定義用於創建對象的接口，但讓子類決定實例化哪個類。
Factory Method允許類將實例化延遲到子類。

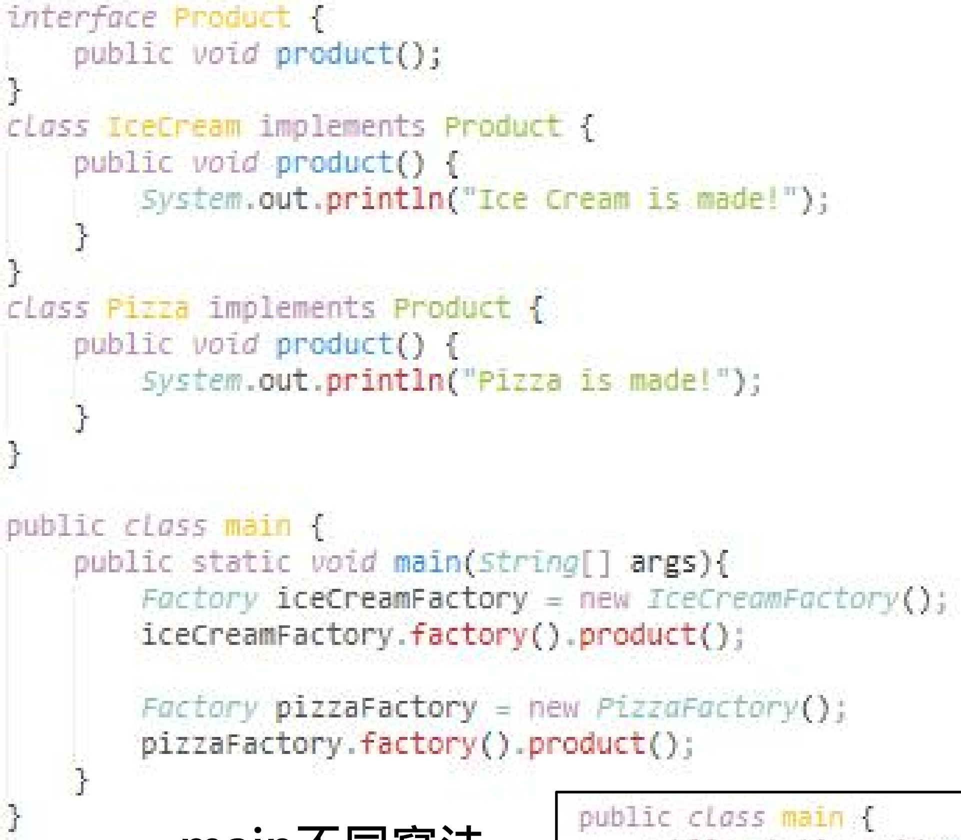
一個工廠對應一個ConcreteFactory
多一個產品就要多一種ConcreteFactory


(Client) →
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
Product
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

範例1.Factory Method :我想吃冰淇淋跟披薩

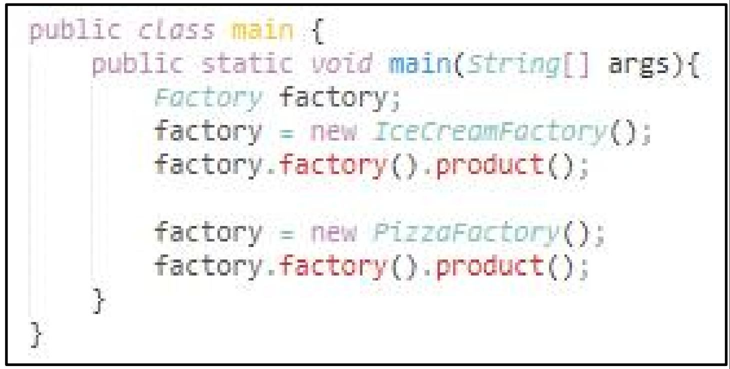


```
interface Factory {
    public Product factory();
}
class IceCreamFactory implements Factory {
    public Product factory() {
        return new IceCream();
    }
}
class PizzaFactory implements Factory {
    public Product factory() {
        return new Pizza();
    }
}
```

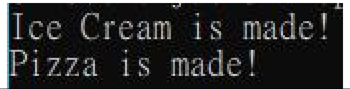


```
interface Product {
    public void product();
}
class IceCream implements Product {
    public void product() {
        System.out.println("Ice Cream is made!");
    }
}
class Pizza implements Product {
    public void product() {
        System.out.println("Pizza is made!");
    }
}

public class main {
    public static void main(String[] args){
        Factory iceCreamFactory = new IceCreamFactory();
        iceCreamFactory.factory().product();

        Factory pizzaFactory = new PizzaFactory();
        pizzaFactory.factory().product();
    }
}
```

main不同寫法→

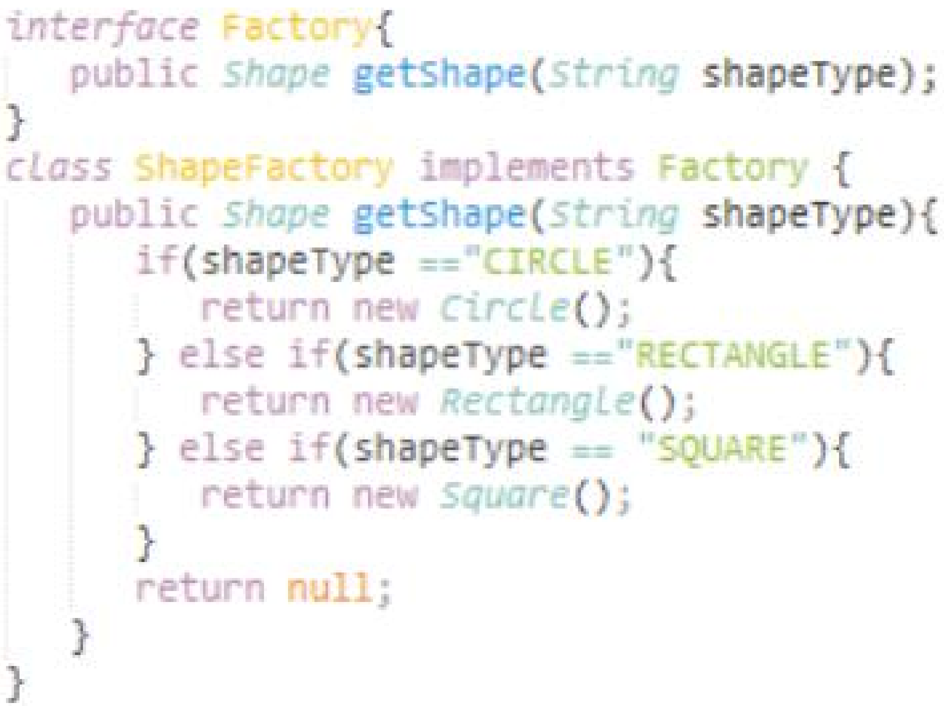


```
public class main {
    public static void main(String[] args){
        Factory factory;
        factory = new IceCreamFactory();
        factory.factory().product();

        factory = new PizzaFactory();
        factory.factory().product();
    }
}
```

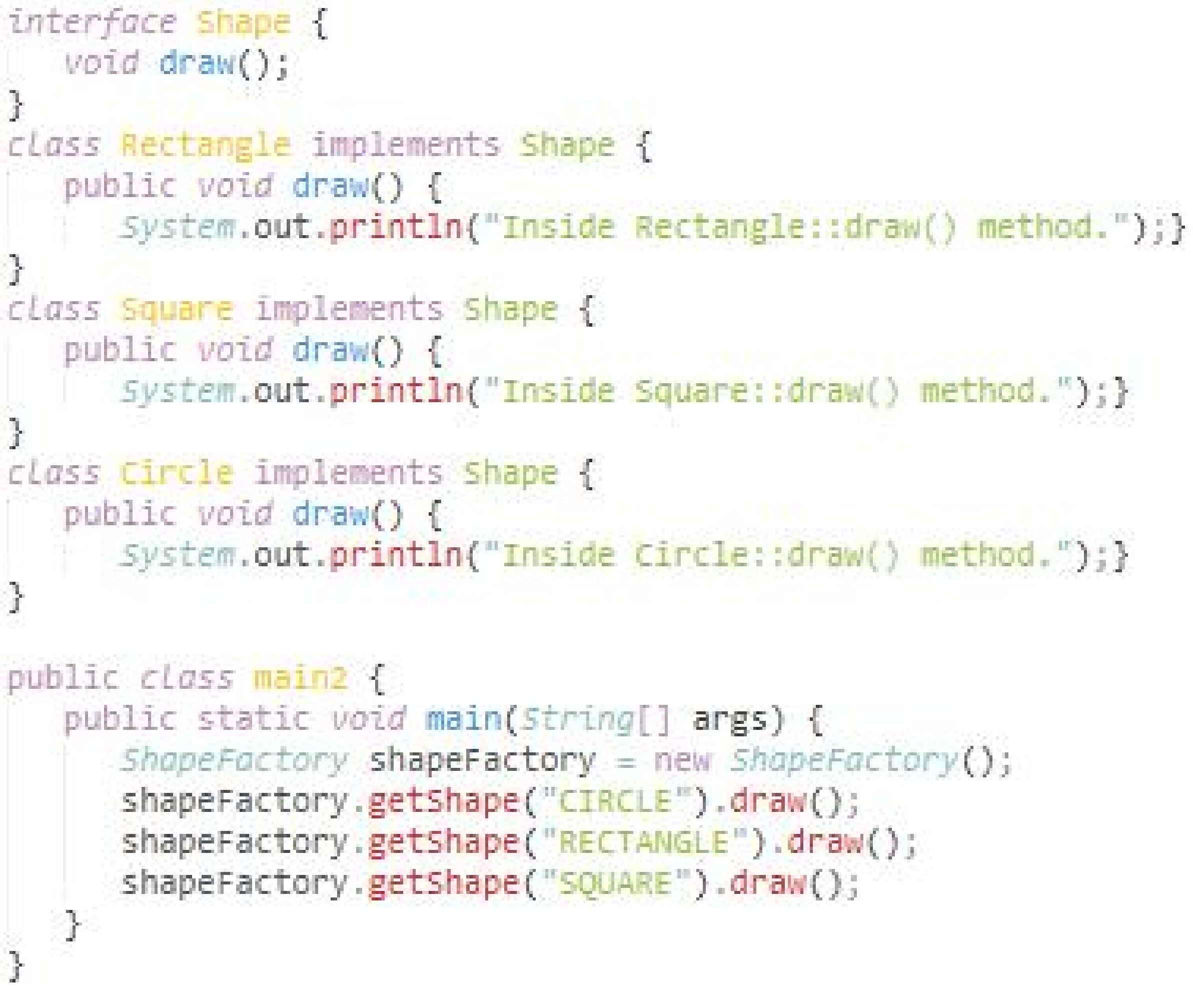


```
Ice Cream is made!
Pizza is made!
```

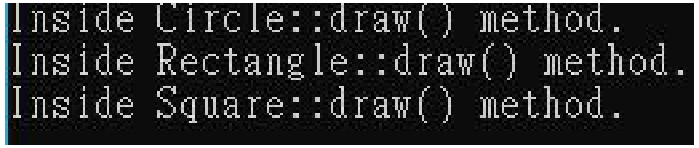
範例2.我想建造一個____的形狀



```java
interface Factory{
   public Shape getShape(String shapeType);
}
class ShapeFactory implements Factory {
   public Shape getShape(String shapeType){
      if(shapeType =="CIRCLE"){
         return new Circle();
      } else if(shapeType =="RECTANGLE"){
         return new Rectangle();
      } else if(shapeType == "SQUARE"){
         return new Square();
      }
      return null;
   }
}
```



```java
interface Shape {
   void draw();
}
class Rectangle implements Shape {
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");}
}
class Square implements Shape {
   public void draw() {
      System.out.println("Inside Square::draw() method.");}
}
class Circle implements Shape {
   public void draw() {
      System.out.println("Inside Circle::draw() method.");}
}

public class main2 {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
      shapeFactory.getShape("CIRCLE").draw();
      shapeFactory.getShape("RECTANGLE").draw();
      shapeFactory.getShape("SQUARE").draw();
   }
}
```



```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

3. Builder 建造者模式

Separate the construction of a complex object from its representation
so that the same construction process can create different representations.

將復雜對象的構造與其表示分開，以便將其複制相同的施工過程可以創建不同的表示。

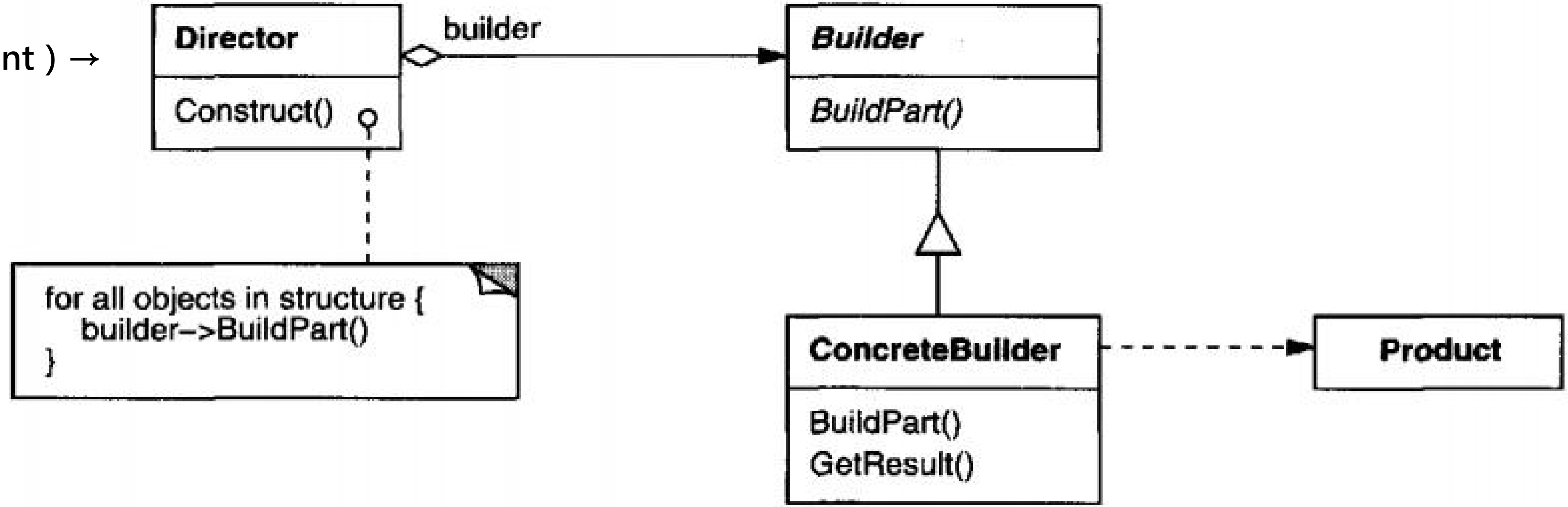
這個模式用來建造由複雜組成的產品
Builder可以製造許多part 組成Product

每一個ConcreteBulider
會根據Director的規定建立產品

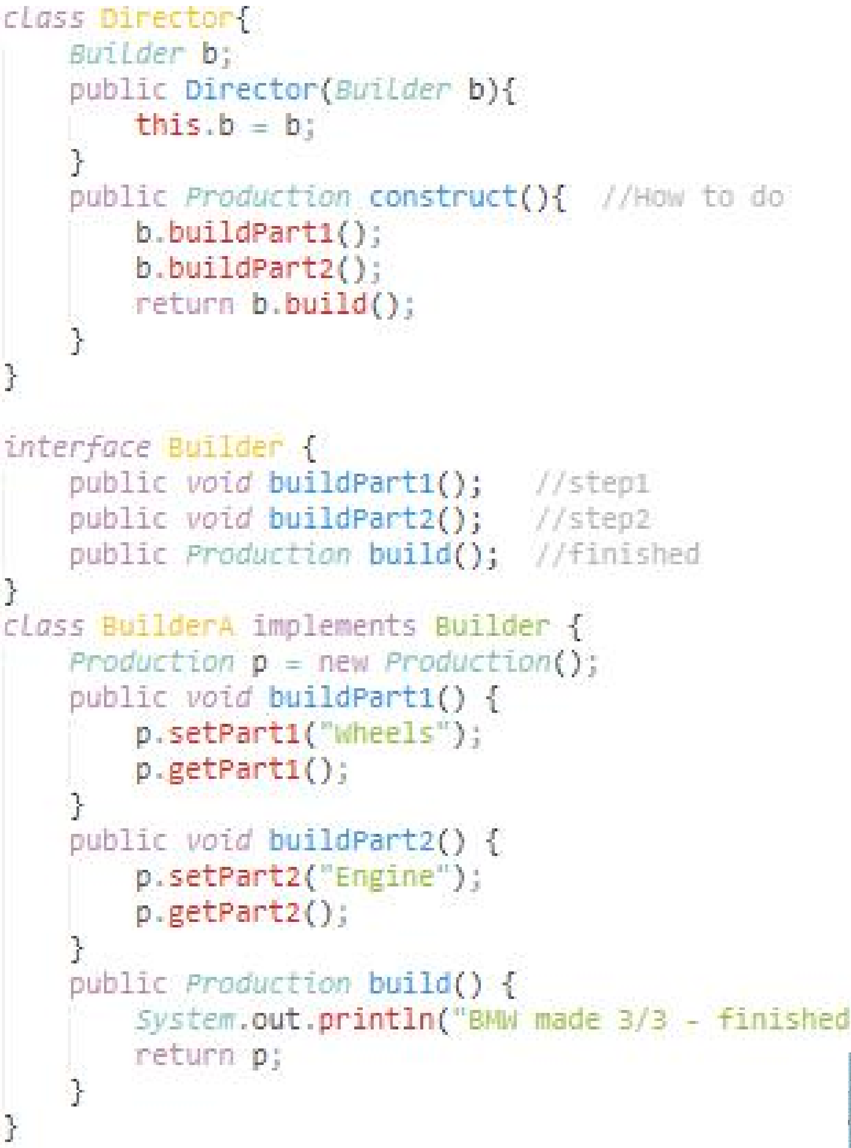
Director裡面會規定如何建造
再透過ConcreteBulider來取得產品

Director知道When to do.
Builder 知道 How to do.

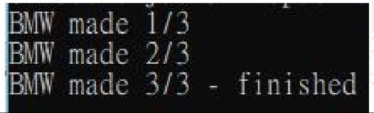
(Client) →


Director
Construct()
builder
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product

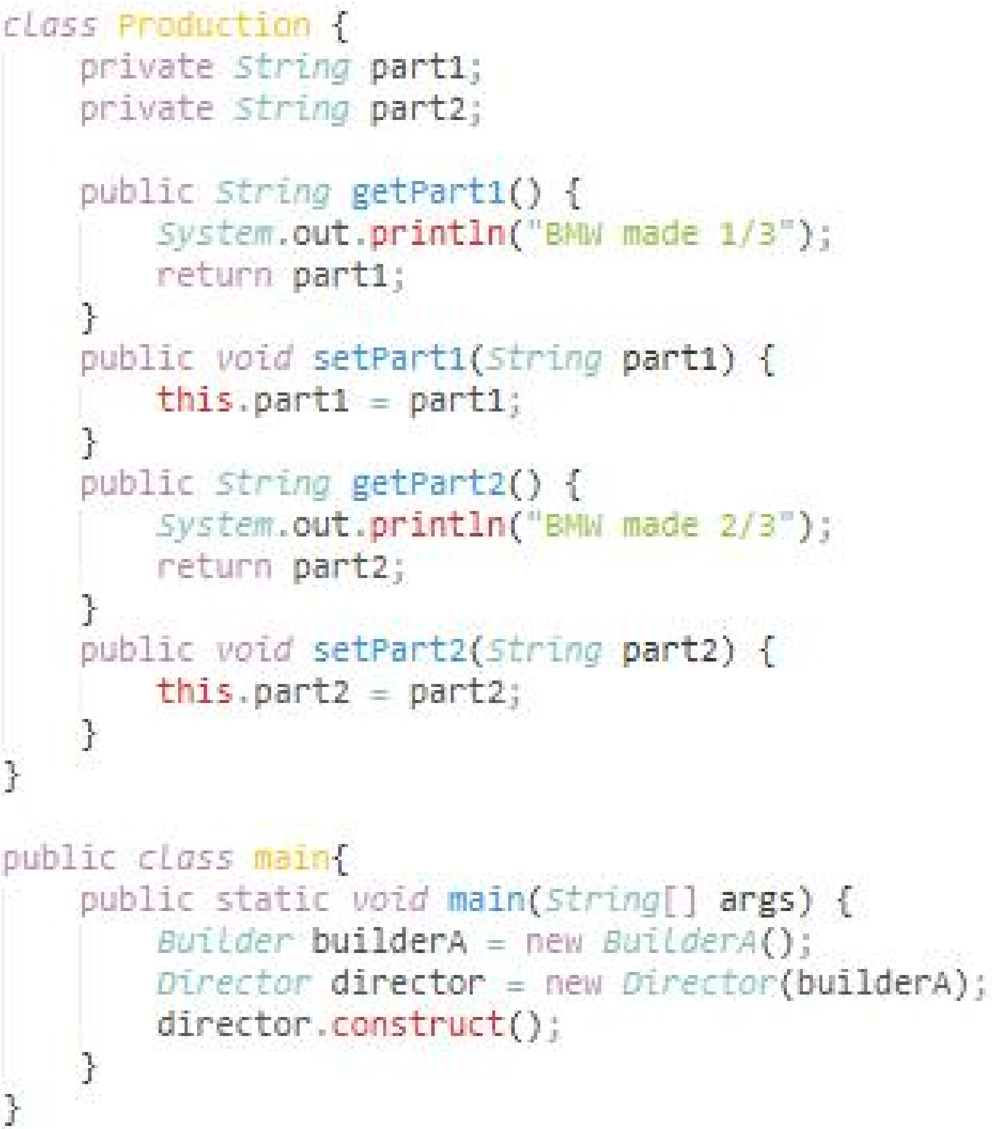
範例1.我現在要製造一輛車



```
class Director{
    Builder b;
    public Director(Builder b){
        this.b = b;
    }
    public Production construct(){  //How to do
        b.buildPart1();
        b.buildPart2();
        return b.build();
    }
}

interface Builder {
    public void buildPart1();   //step1
    public void buildPart2();   //step2
    public Production build();  //finished
}
class BuilderA implements Builder {
    Production p = new Production();
    public void buildPart1() {
        p.setPart1("Wheels");
        p.getPart1();
    }
    public void buildPart2() {
        p.setPart2("Engine");
        p.getPart2();
    }
    public Production build() {
        System.out.println("BMW made 3/3 - finished");
        return p;
    }
}
```

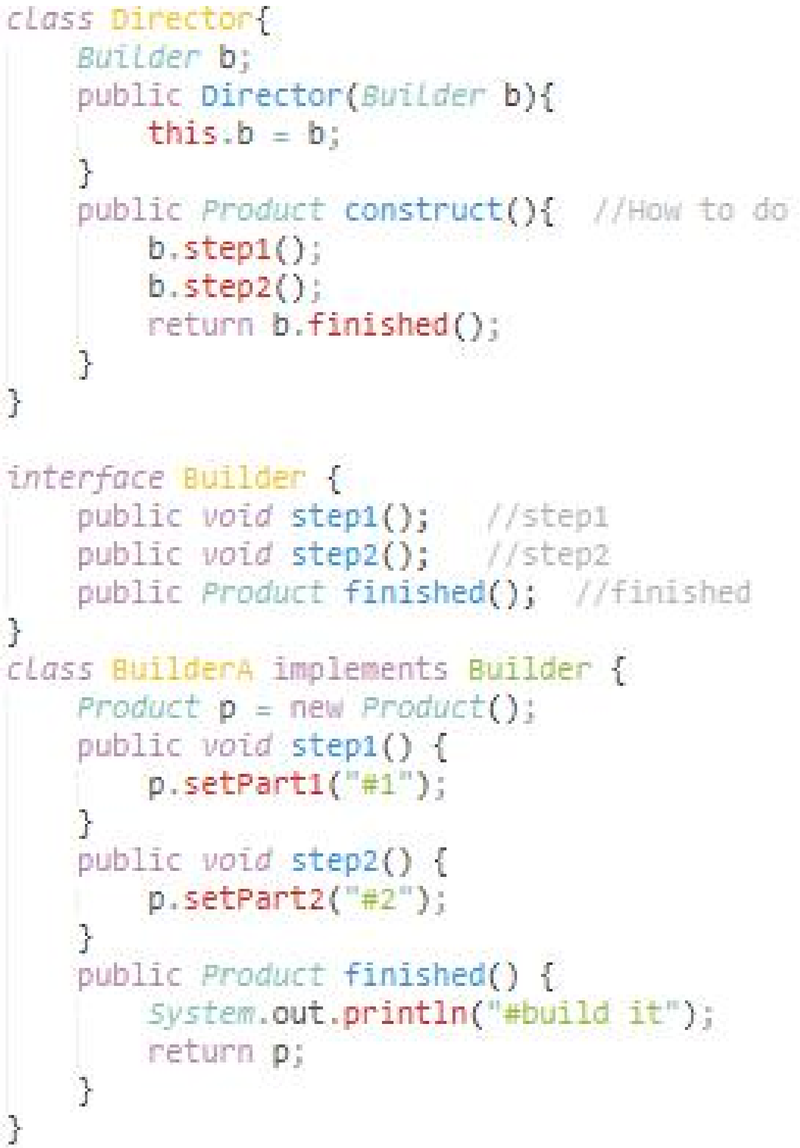


```
BMW made 1/3
BMW made 2/3
BMW made 3/3 - finished
```

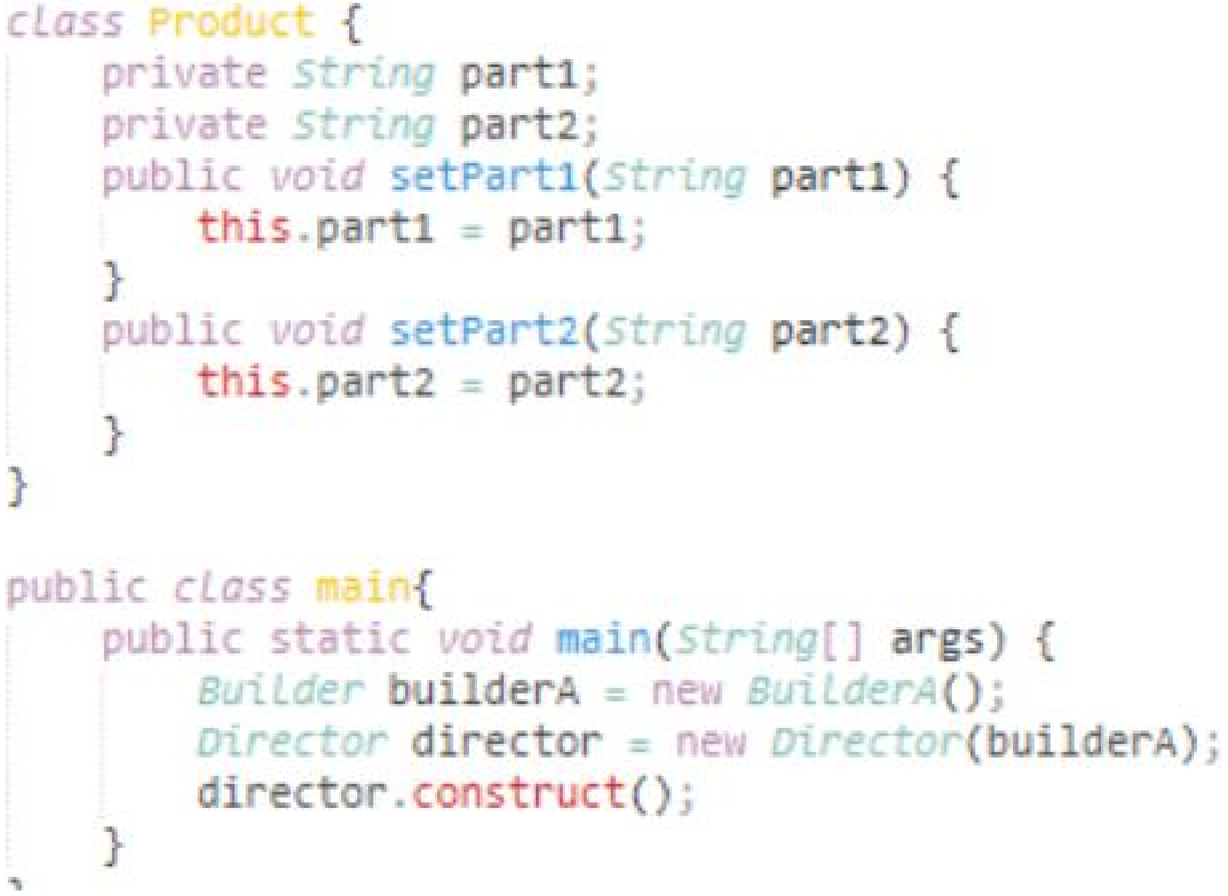


```
class Production {
    private String part1;
    private String part2;

    public String getPart1() {
        System.out.println("BMW made 1/3");
        return part1;
    }
    public void setPart1(String part1) {
        this.part1 = part1;
    }
    public String getPart2() {
        System.out.println("BMW made 2/3");
        return part2;
    }
    public void setPart2(String part2) {
        this.part2 = part2;
    }
}

public class main{
    public static void main(String[] args) {
        Builder builderA = new BuilderA();
        Director director = new Director(builderA);
        director.construct();
    }
}
```

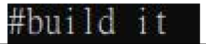
範例2.基本架構



```java
class Director{
    Builder b;
    public Director(Builder b){
        this.b = b;
    }
    public Product construct(){  //How to do
        b.step1();
        b.step2();
        return b.finished();
    }
}

interface Builder {
    public void step1();   //step1
    public void step2();   //step2
    public Product finished();  //finished
}
class BuilderA implements Builder {
    Product p = new Product();
    public void step1() {
        p.setPart1("#1");
    }
    public void step2() {
        p.setPart2("#2");
    }
    public Product finished() {
        System.out.println("#build it");
        return p;
    }
}
```



```java
class Product {
    private String part1;
    private String part2;
    public void setPart1(String part1) {
        this.part1 = part1;
    }
    public void setPart2(String part2) {
        this.part2 = part2;
    }
}

public class main{
    public static void main(String[] args) {
        Builder builderA = new BuilderA();
        Director director = new Director(builderA);
        director.construct();
    }
}
```



```
#build it
```

4. Prototype 原型模式

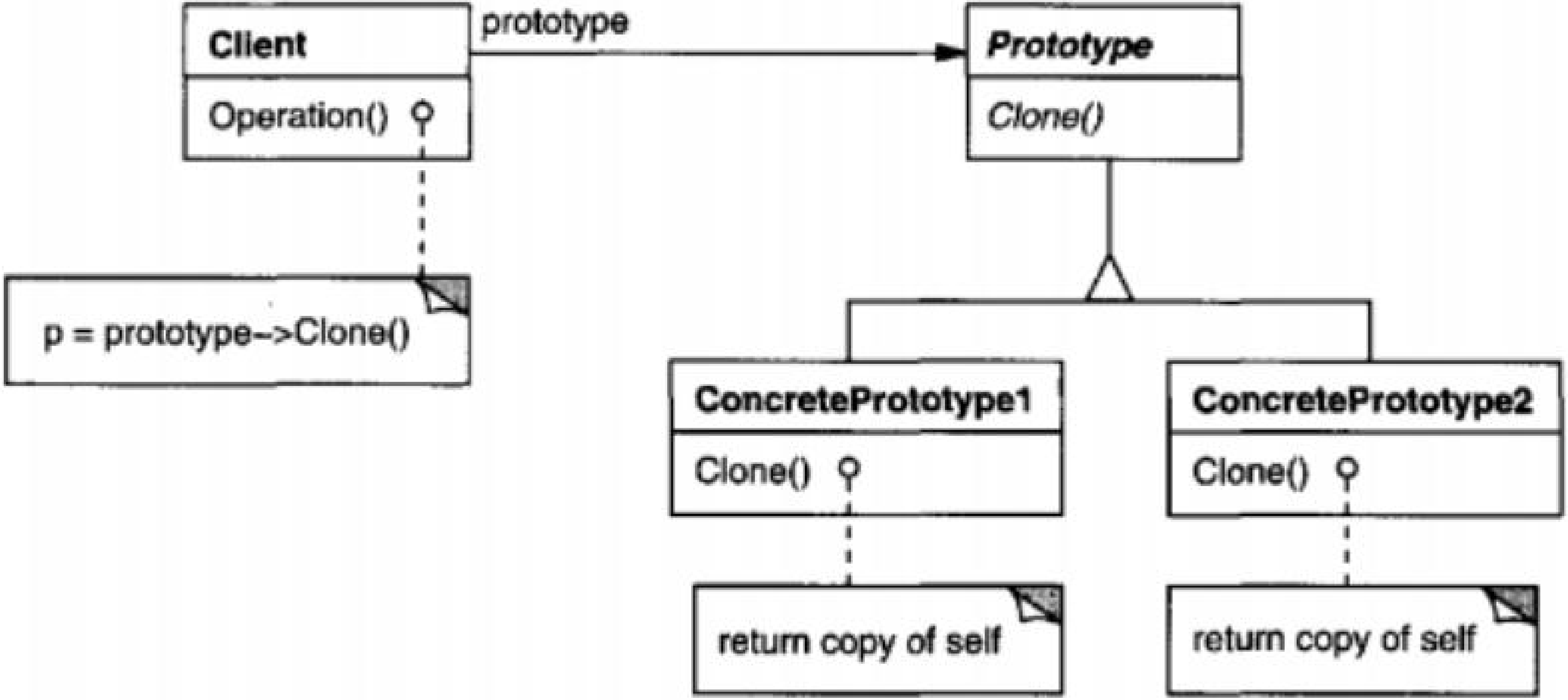
Specify the kinds of objects to create using a prototypical instance,
and create new objects by copying this prototype.

使用原型實例指定要創建的對象類型，然後創建新對象複製此原型的對象。

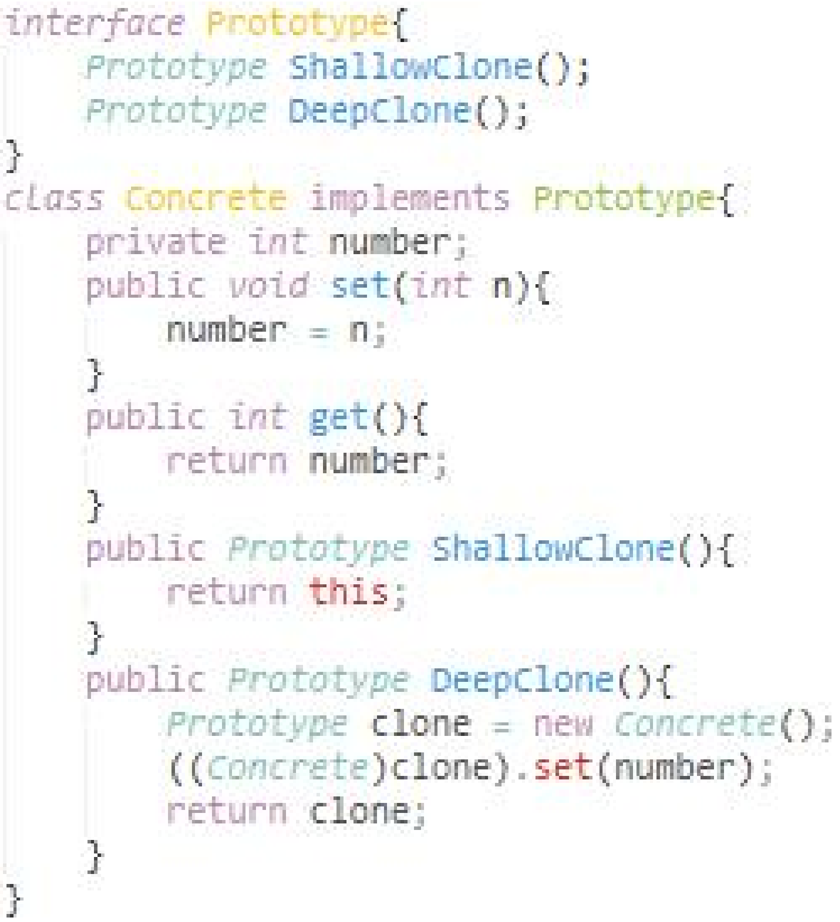
這個Pattern簡單來說就是用來複製

Prototype是原體
ConcretePrototype就是複製體
很簡單，問題在於如何去複製

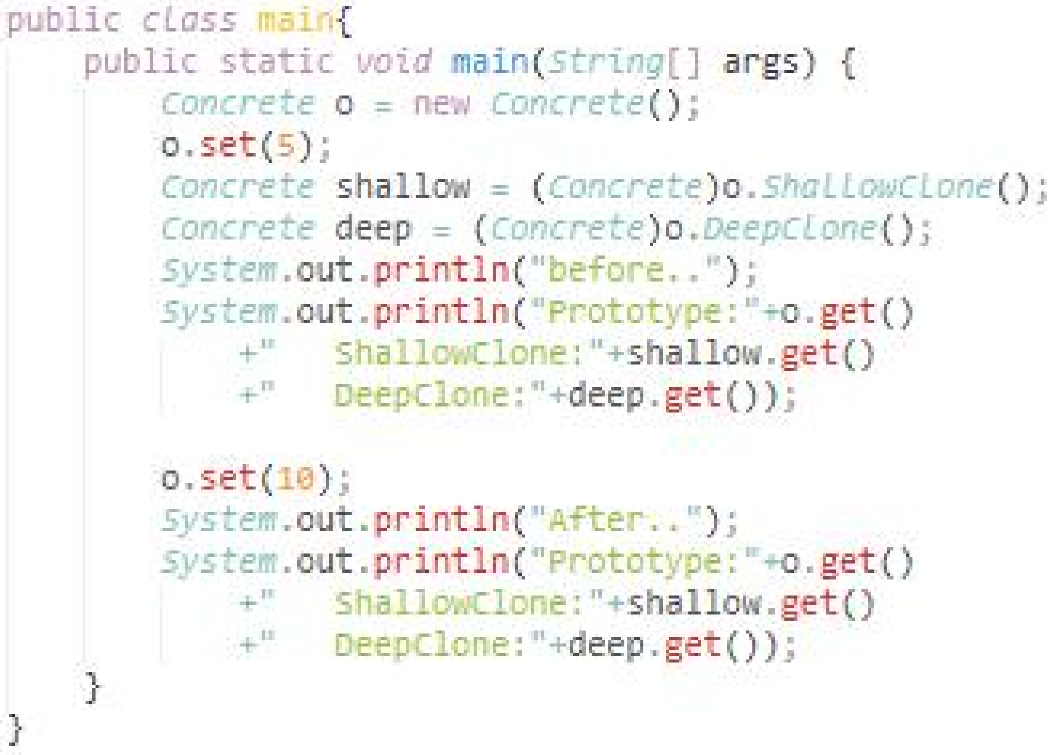
Shallow Copy 當原型被修改，複製體也會跟著改
Deep Copy 當原型被修改，複製體不會跟著改


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

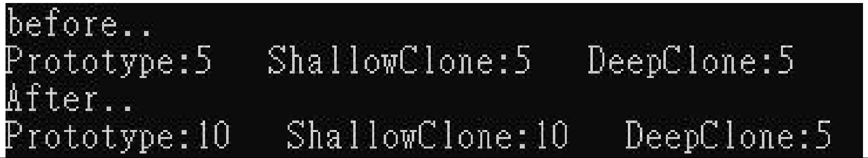
範例1.



```java
interface Prototype{
    Prototype ShallowClone();
    Prototype DeepClone();
}
class Concrete implements Prototype{
    private int number;
    public void set(int n){
        number = n;
    }
    public int get(){
        return number;
    }
    public Prototype ShallowClone(){
        return this;
    }
    public Prototype DeepClone(){
        Prototype clone = new Concrete();
        ((Concrete)clone).set(number);
        return clone;
    }
}
```



```java
public class main{
    public static void main(String[] args) {
        Concrete o = new Concrete();
        o.set(5);
        Concrete shallow = (Concrete)o.ShallowClone();
        Concrete deep = (Concrete)o.DeepClone();
        System.out.println("before..");
        System.out.println("Prototype:"+o.get()
            +"   ShallowClone:"+shallow.get()
            +"   DeepClone:"+deep.get());

        o.set(10);
        System.out.println("After..");
        System.out.println("Prototype:"+o.get()
            +"   ShallowClone:"+shallow.get()
            +"   DeepClone:"+deep.get());
    }
}
```



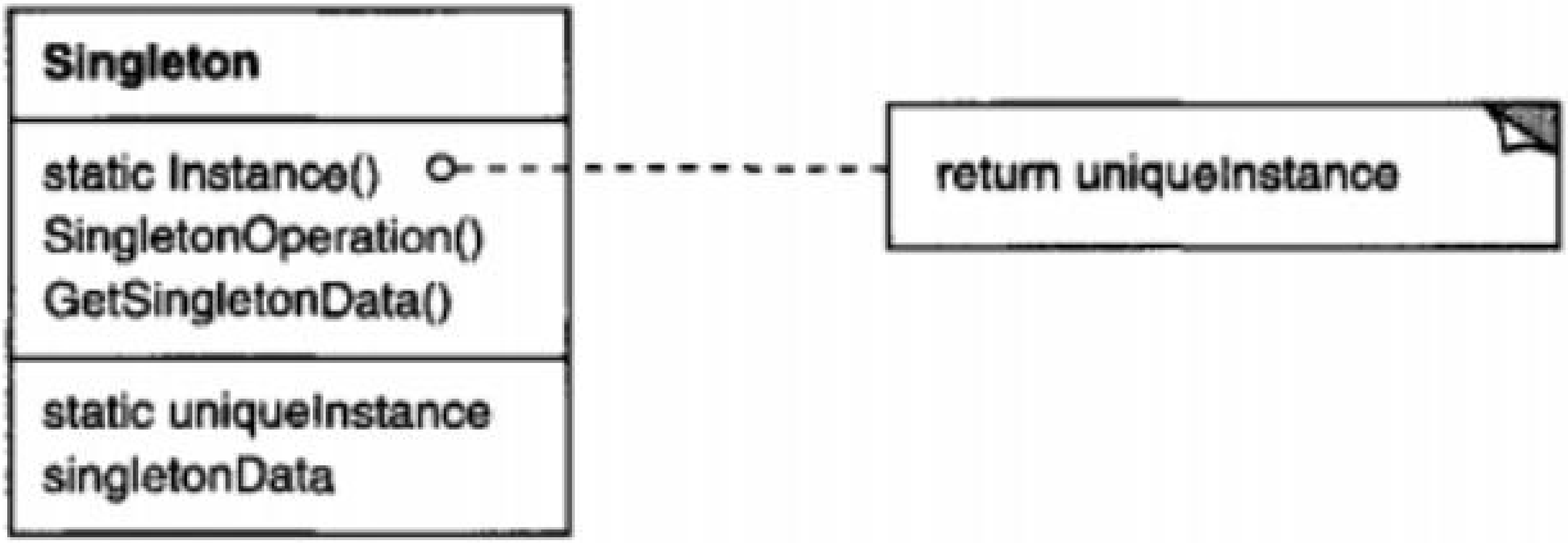
```
before..
Prototype:5   ShallowClone:5   DeepClone:5
After..
Prototype:10   ShallowClone:10   DeepClone:5
```

5. Singleton 單例模式

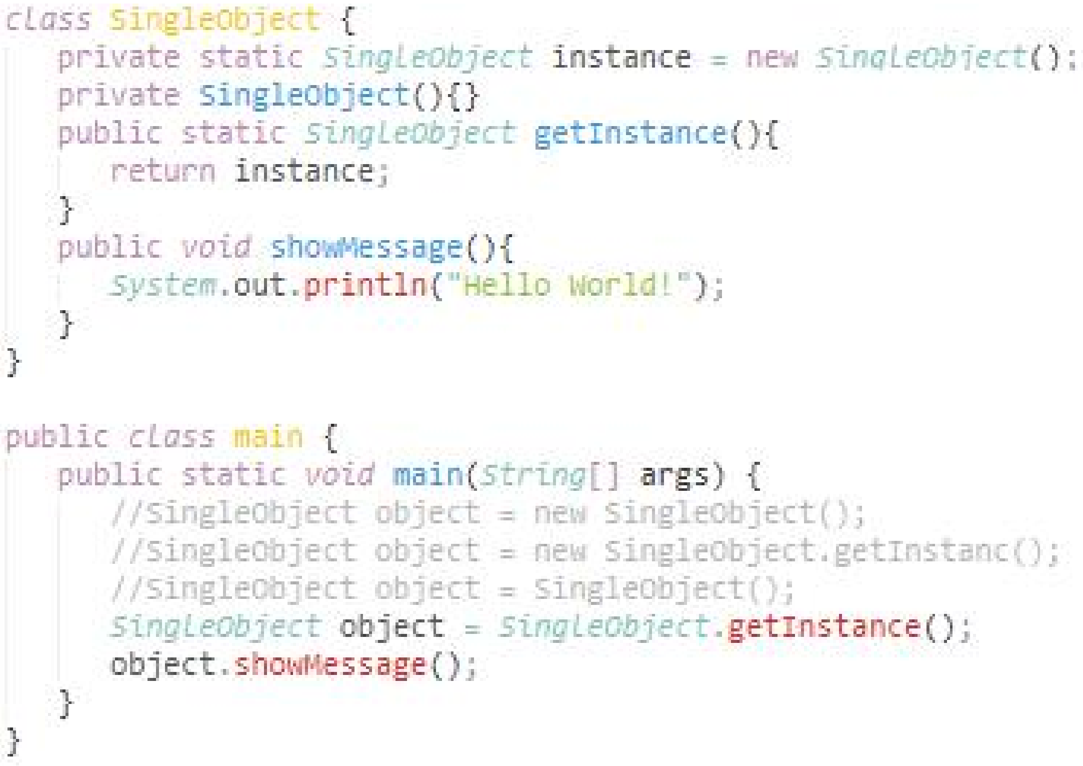
Ensure a class only has one instance, and provide a global point of access to it.

確保一個類只有一個實例，並提供一個全局訪問點。

Singleton就是要確保物件只有**一個**實例可以被重複使用
所以常常是被使用率極高(重複存取)的原件才會這樣做


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

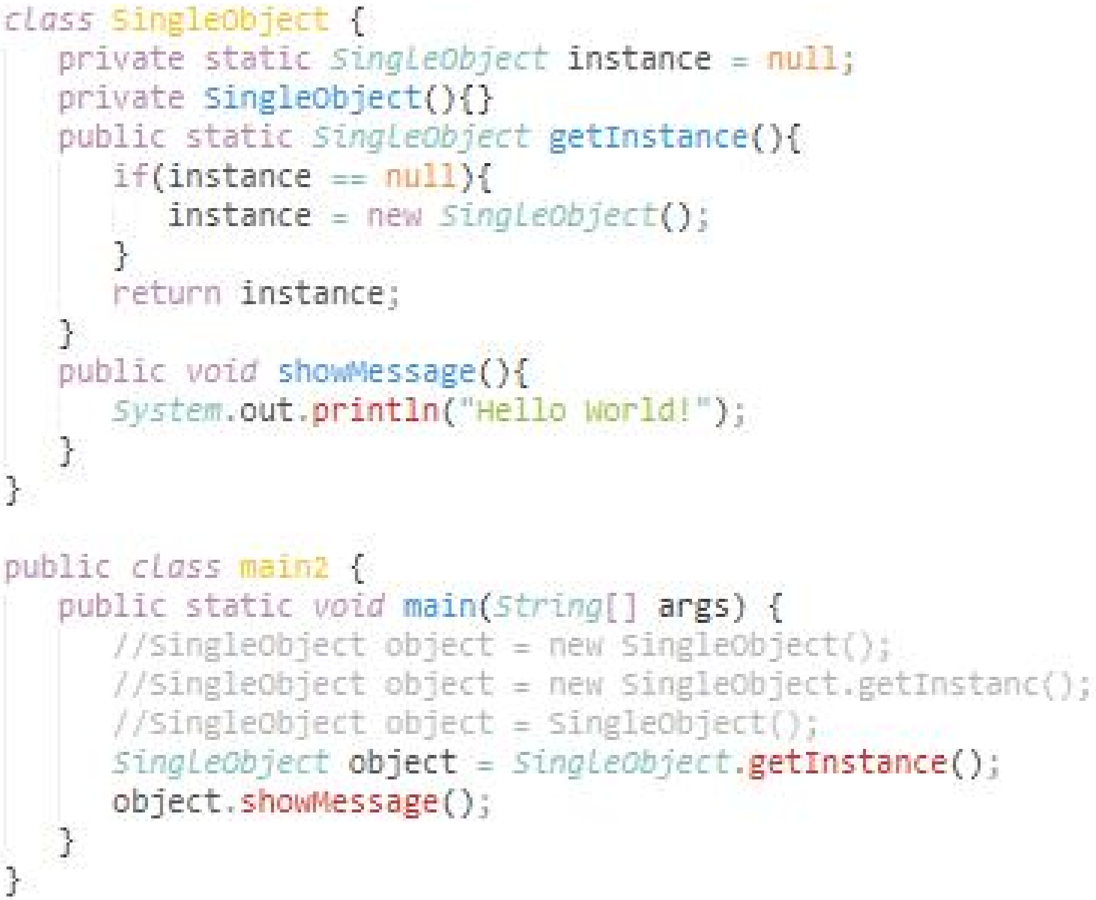
範例1. EagerSingleton



```
class SingleObject {
    private static SingleObject instance = new SingleObject();
    private SingleObject(){}
    public static SingleObject getInstance(){
        return instance;
    }
    public void showMessage(){
        System.out.println("Hello World!");
    }
}

public class main {
    public static void main(String[] args) {
        //SingleObject object = new SingleObject();
        //SingleObject object = new SingleObject.getInstanc();
        //SingleObject object = SingleObject();
        SingleObject object = SingleObject.getInstance();
        object.showMessage();
    }
}
```

在一開始就先建造好實例
反正反覆存取一定會用到
需要的時候直接回傳同一個

範例2. LazySingleton



```
class SingleObject {
    private static SingleObject instance = null;
    private SingleObject(){}
    public static SingleObject getInstance(){
        if(instance == null){
            instance = new SingleObject();
        }
        return instance;
    }
    public void showMessage(){
        System.out.println("Hello World!");
    }
}

public class main2 {
    public static void main(String[] args) {
        //SingleObject object = new SingleObject();
        //SingleObject object = new SingleObject.getInstanc();
        //SingleObject object = SingleObject();
        SingleObject object = SingleObject.getInstance();
        object.showMessage();
    }
}
```

若為否則在此時建造實體
在取得實體時判斷是否已經建造

6. Adapter 轉接器模式

Convert the interface of a class into another interface clients expect.
Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
將類的接口轉換為客戶期望的另一個接口。 適配器由於不兼容的接口，類無法協同工作。

Adapter模式分為三種：

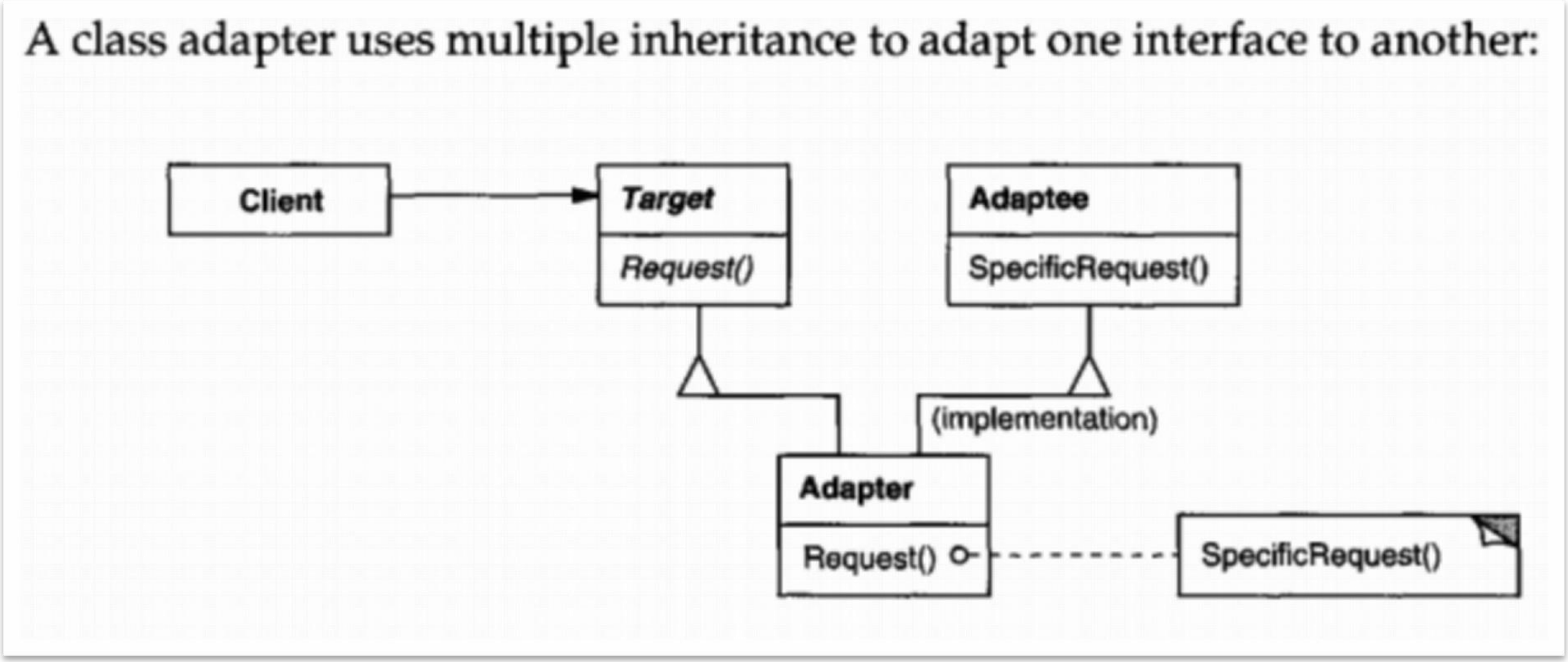
1.類(class)，
Class方式只是換成了用多重繼承的方式
讓Adapter多繼承Adaptee。
【Java不支援多重繼承故不討論】

2.對象(object)，
Object方式就是把Adaptee放在Adapter中繼承Target
使用者就會認為Adapter是Target
Adapter再用Adaptee的方法來做成Target的方法

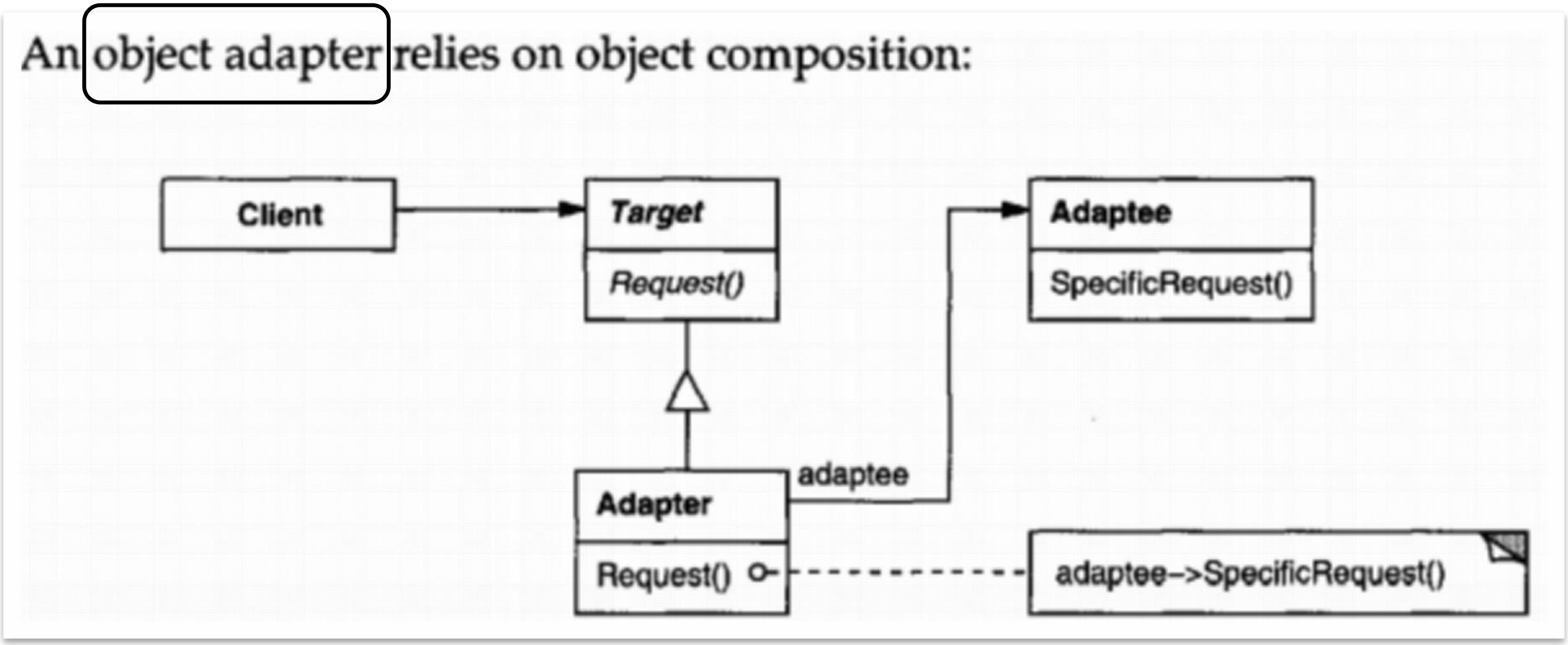
3.接口(interface)，
【較少使用故不討論】

【Adapter vs .Proxy】

Adapter是解決現有對象在新環境中的不足，
Proxy是解決直接訪問對象時出現的問題，



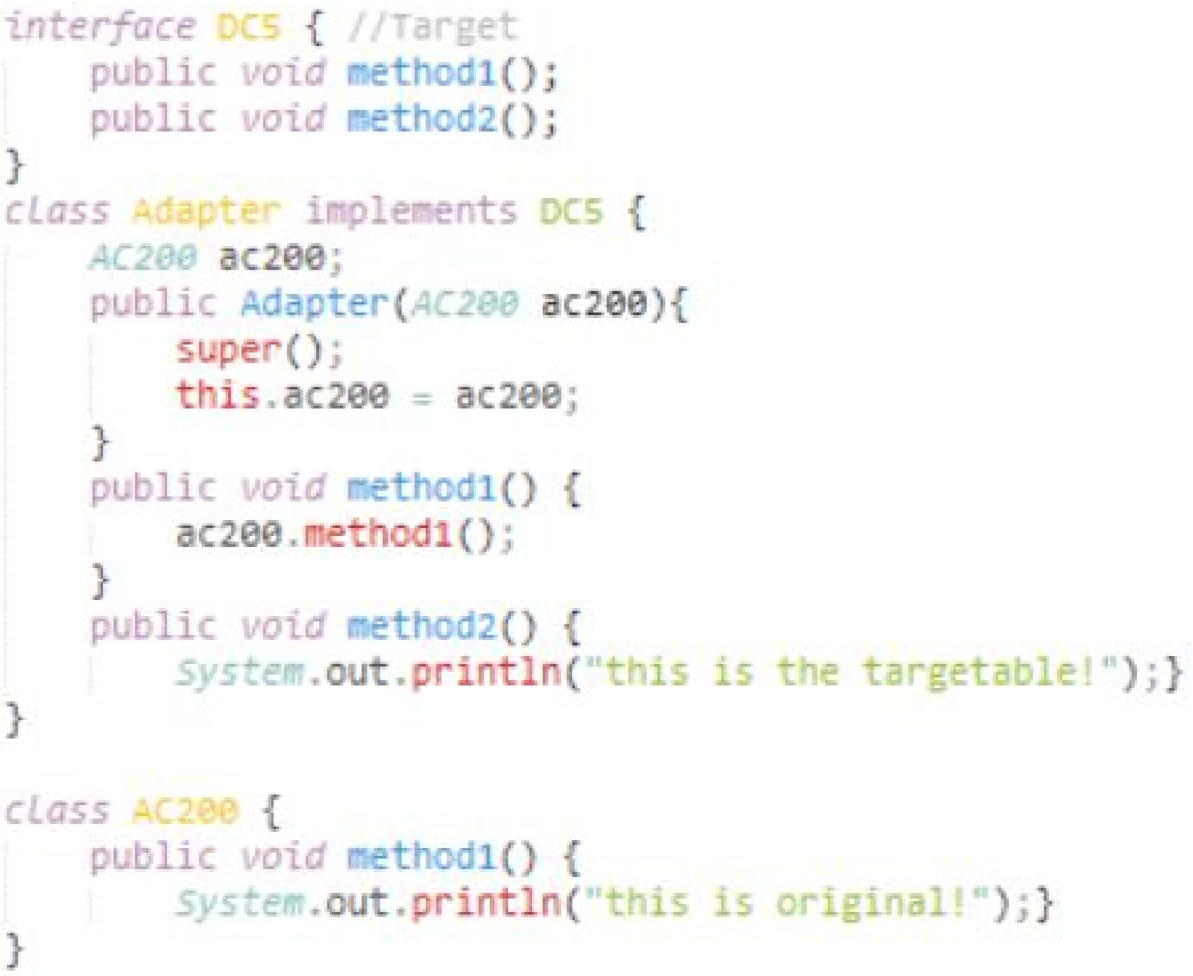
A class adapter uses multiple inheritance to adapt one interface to another:



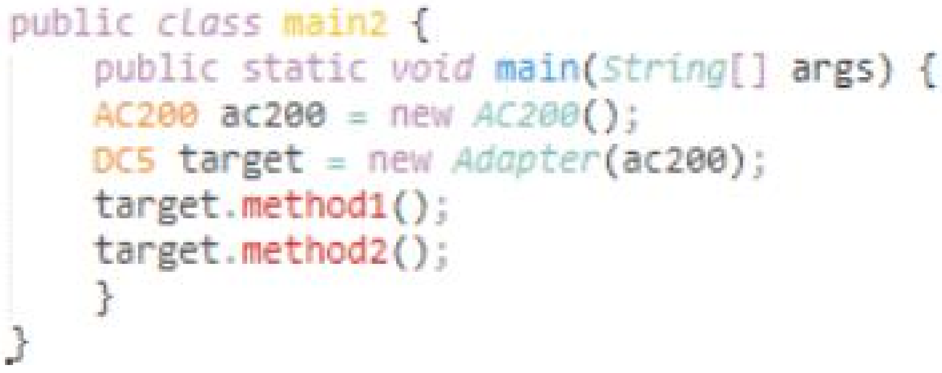
An object adapter relies on object composition:

範例1. 手機充電要將AC 220V 轉為 DC 5V充電

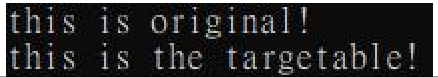
我們用Object Adapter寫一個電源的轉接頭，將AC220v —>



```
interface DC5 { //Target
    public void method1();
    public void method2();
}
class Adapter implements DC5 {
    AC200 ac200;
    public Adapter(AC200 ac200){
        super();
        this.ac200 = ac200;
    }
    public void method1() {
        ac200.method1();
    }
    public void method2() {
        System.out.println("this is the targetable!");}
}

class AC200 {
    public void method1() {
        System.out.println("this is original!");}
}
```

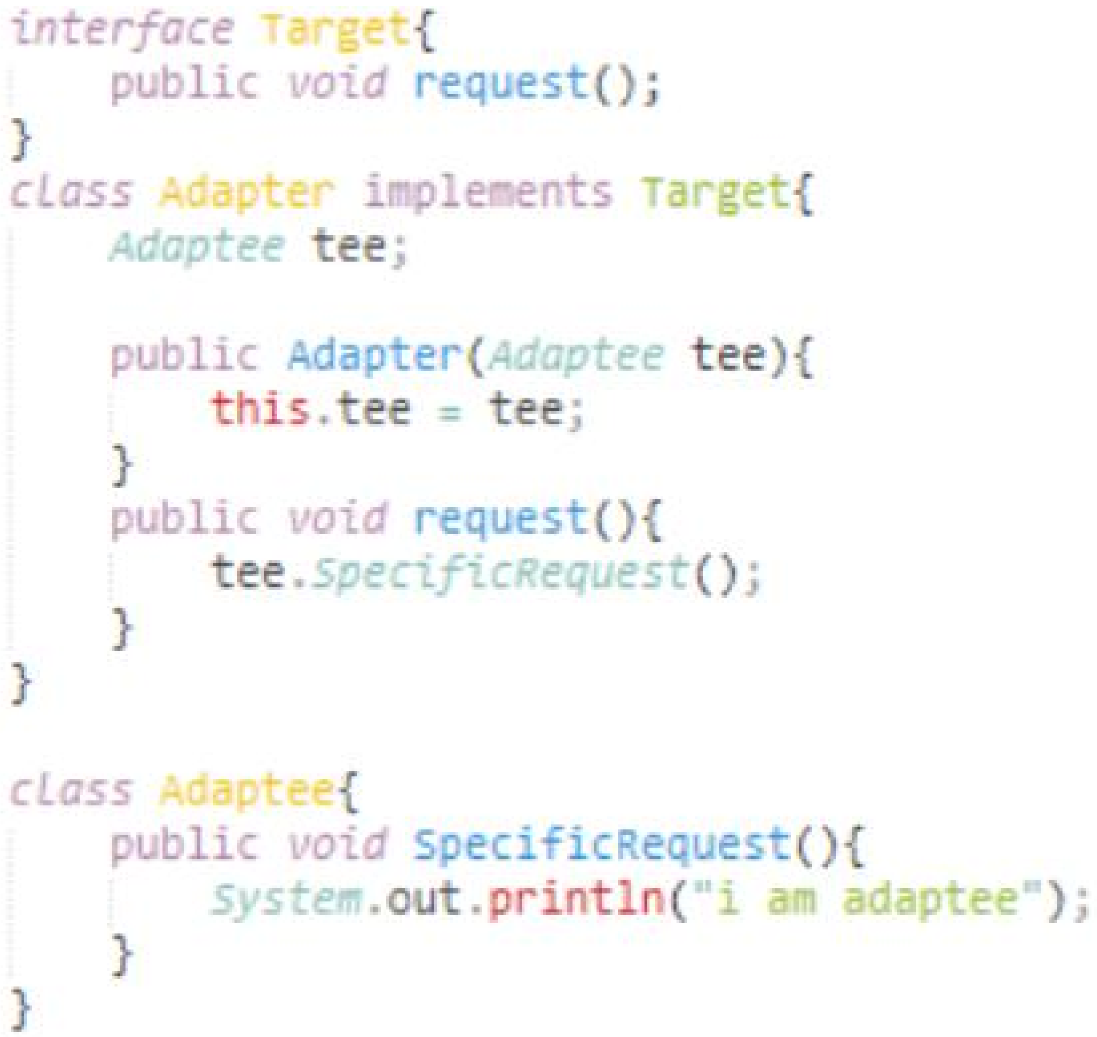


```
public class main2 {
    public static void main(String[] args) {
    AC200 ac200 = new AC200();
    DC5 target = new Adapter(ac200);
    target.method1();
    target.method2();
    }
}
```

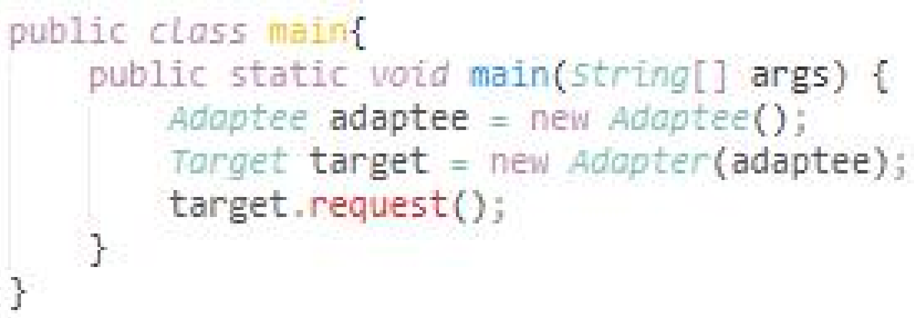


```
this is original!
this is the targetable!
```

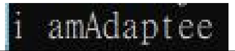
範例2.基本架構



```
interface Target{
    public void request();
}
class Adapter implements Target{
    Adaptee tee;

    public Adapter(Adaptee tee){
        this.tee = tee;
    }
    public void request(){
        tee.SpecificRequest();
    }
}

class Adaptee{
    public void SpecificRequest(){
        System.out.println("i am adaptee");
    }
}
```



```
public class main{
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new Adapter(adaptee);
        target.request();
    }
}
```



```
i amAdaptee
```

7. Bridge 橋接模式

Decouple an abstraction from its implementation so that the two can vary independently.

將抽象與其實現分離，以使兩者可以獨立變化。

假設,左邊(Abstraction)有三種框架
右邊(Implementor)有四種實作

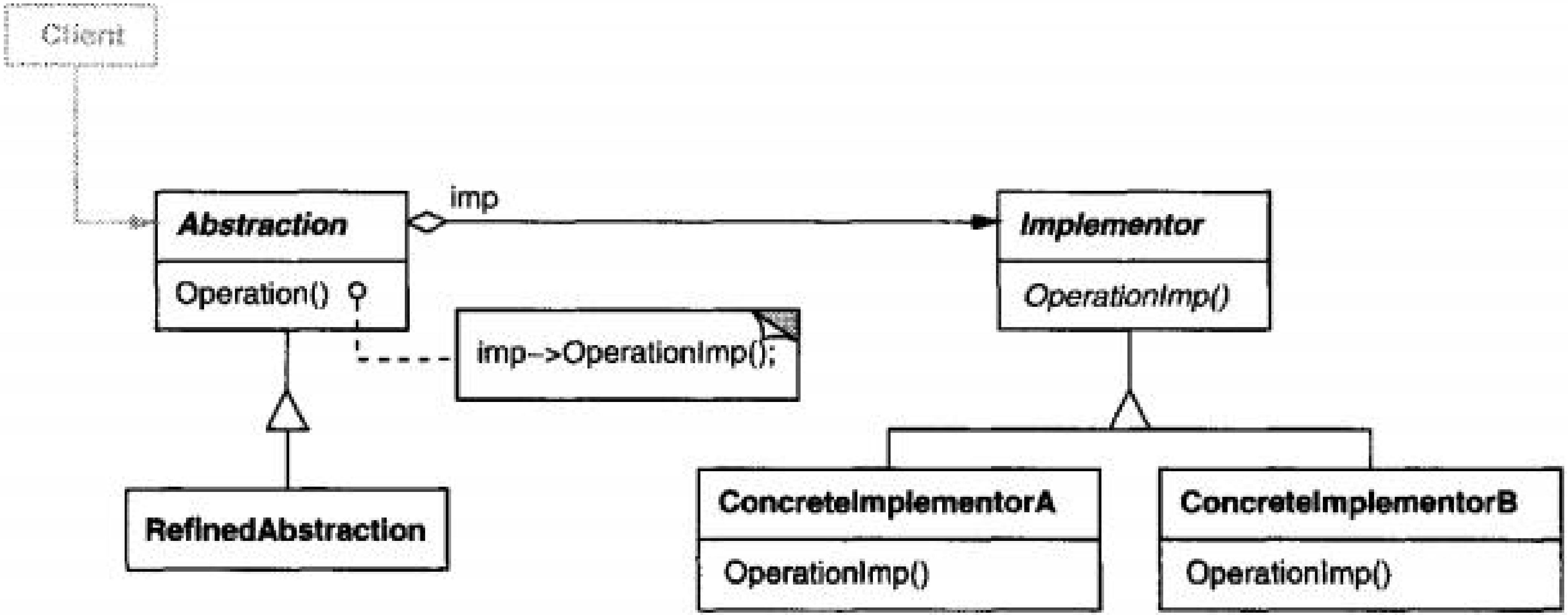
那總共就有3x4=12種變化
所以只要一方增加了就可以多出很多種變化
例如：軟體可以選擇要用什麼語言

【Strategy vs Bridge】

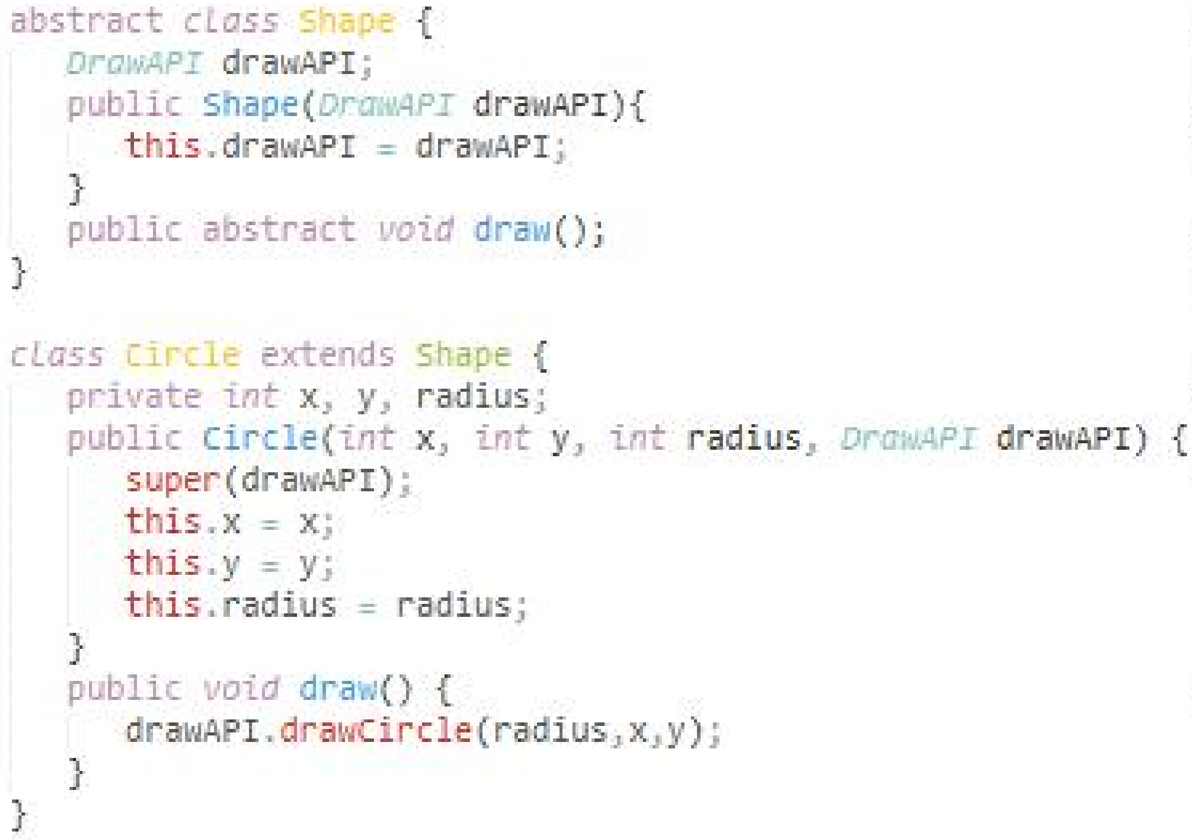
Strategy是Behavioral pattern
強調使用者可以選擇怎樣的方式去做

Bridge是Structural pattern
強調把架構跟實作分離

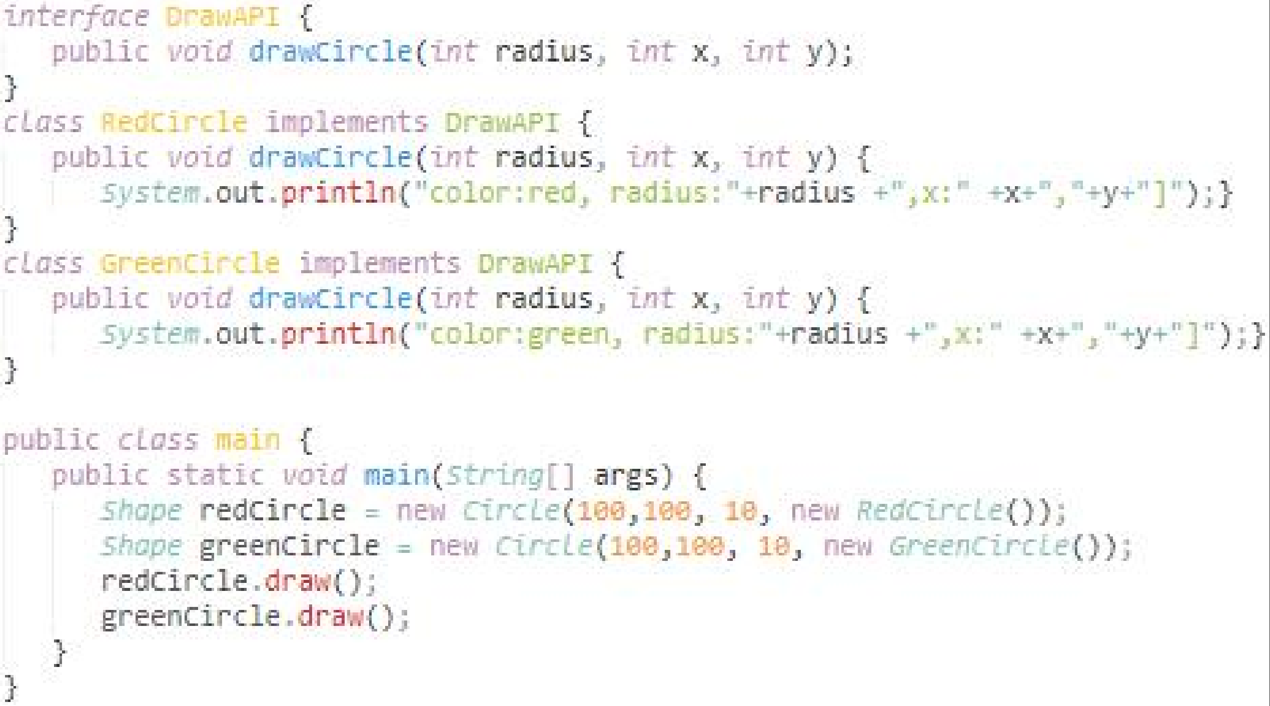
小敏會強調Bridge是架構和實作的所有組合都能夠實現


Client
Abstraction
Operation()
imp
imp->OperationImp();
RefinedAbstraction
Implementor
OperationImp()
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

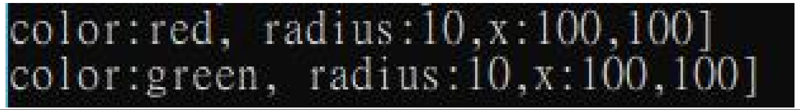
範例1. 我要進行座標繪圖



```java
abstract class Shape {
    DrawAPI drawAPI;
    public Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}

class Circle extends Shape {
    private int x, y, radius;
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public void draw() {
        drawAPI.drawCircle(radius,x,y);
    }
}
```

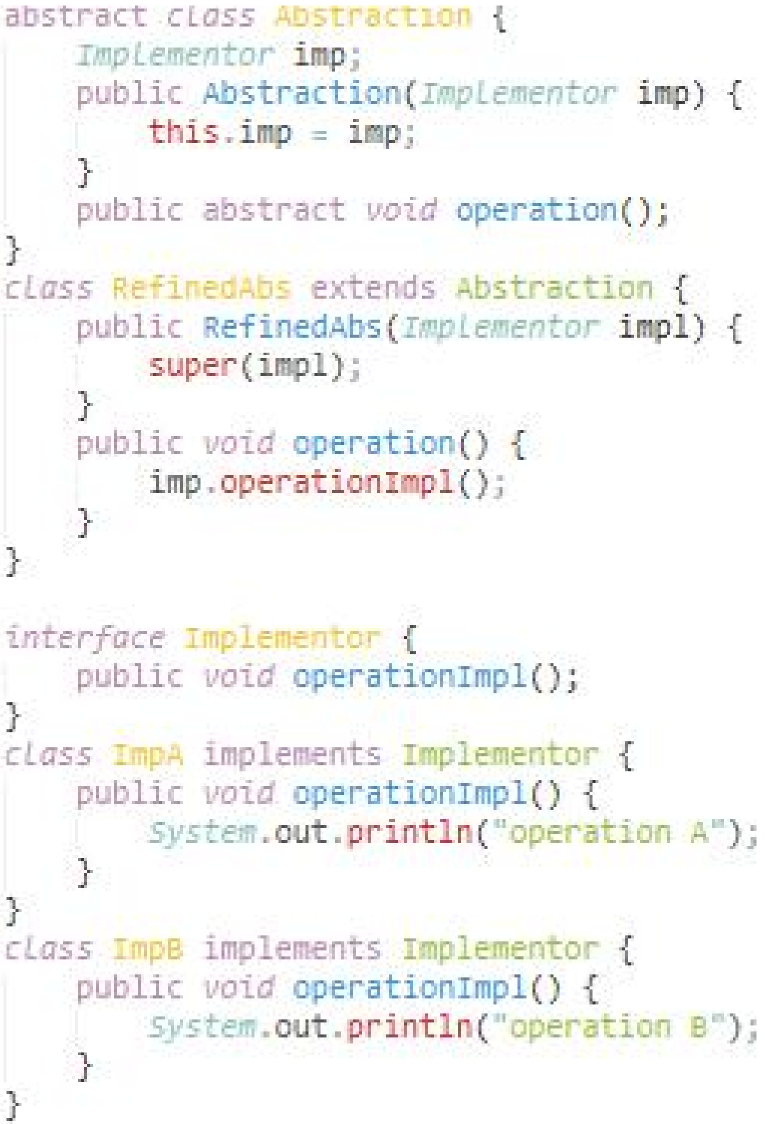


```java
interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
class RedCircle implements DrawAPI {
    public void drawCircle(int radius, int x, int y) {
        System.out.println("color:red, radius:"+radius +",x:" +x+","+y+"]");}
}
class GreenCircle implements DrawAPI {
    public void drawCircle(int radius, int x, int y) {
        System.out.println("color:green, radius:"+radius +",x:" +x+","+y+"]");}
}

public class main {
    public static void main(String[] args) {
        Shape redCircle = new Circle(100,100, 10, new RedCircle());
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
        redCircle.draw();
        greenCircle.draw();
    }
}
```

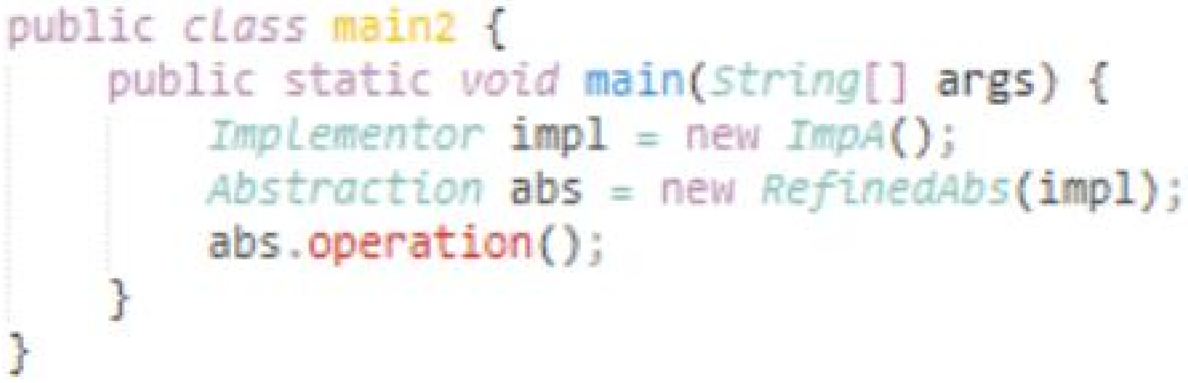


```
color:red, radius:10,x:100,100]
color:green, radius:10,x:100,100]
```

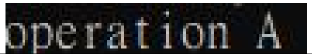
範例2. 基本架構



```
abstract class Abstraction {
    Implementor imp;
    public Abstraction(Implementor imp) {
        this.imp = imp;
    }
    public abstract void operation();
}
class RefinedAbs extends Abstraction {
    public RefinedAbs(Implementor impl) {
        super(impl);
    }
    public void operation() {
        imp.operationImpl();
    }
}

interface Implementor {
    public void operationImpl();
}
class ImpA implements Implementor {
    public void operationImpl() {
        System.out.println("operation A");
    }
}
class ImpB implements Implementor {
    public void operationImpl() {
        System.out.println("operation B");
    }
}
```



```
public class main2 {
    public static void main(String[] args) {
        Implementor impl = new ImpA();
        Abstraction abs = new RefinedAbs(impl);
        abs.operation();
    }
}
```



```
operation A
```

8. Composite 組合模式

Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

將對象組織成樹狀結構產生出階層關係。讓外界一致性(都視為Component)對待個別類別物件和組合類別物件。

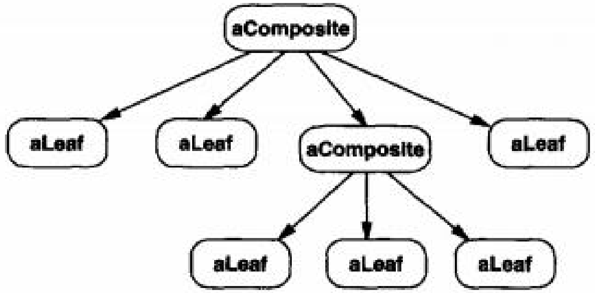
1.先作好方法，防止Leaf使用到功能時不會錯誤 可以作空→{}

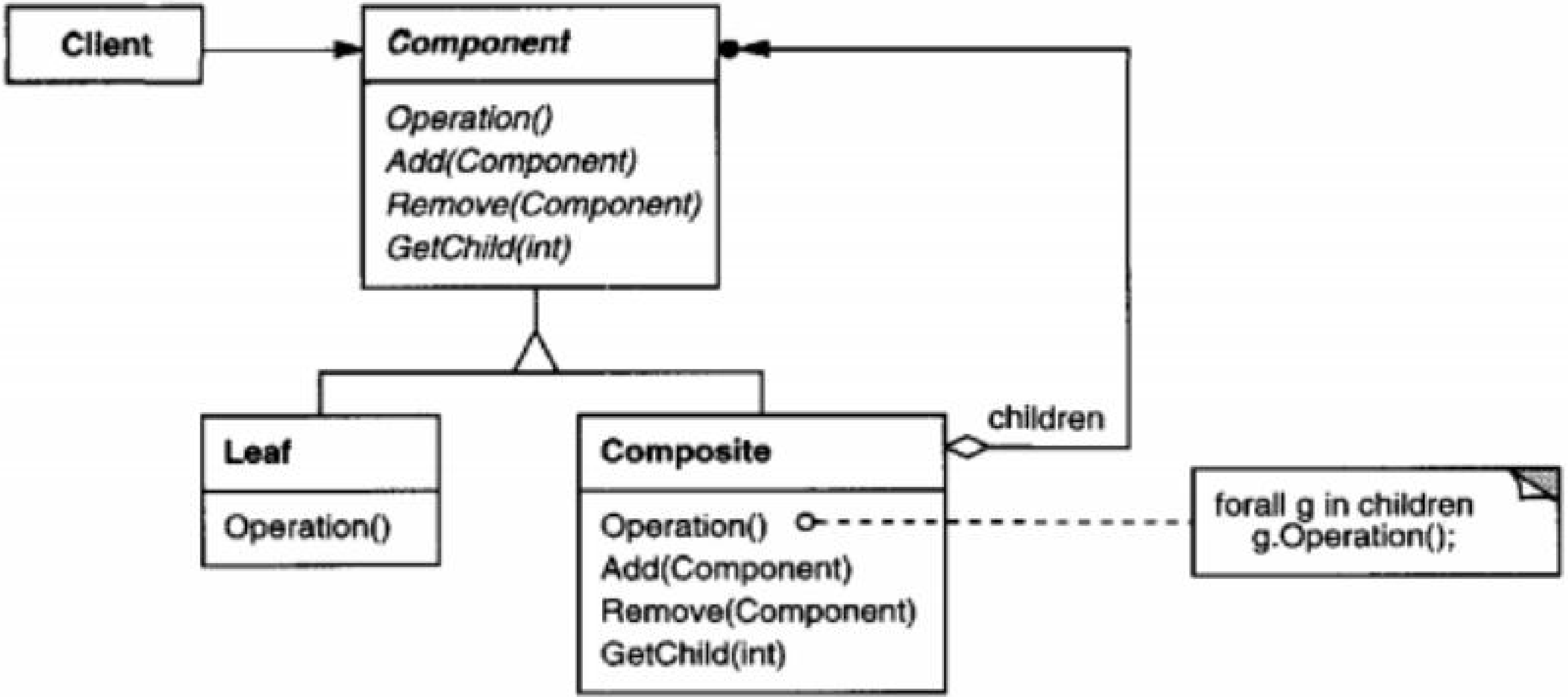
2.Leaf (rice,noodle)

3.建立 composite (menu) ，組合需要ArrayList儲存多個元件，
因為Composite、Leaf 本身繼承，都可以視為Component

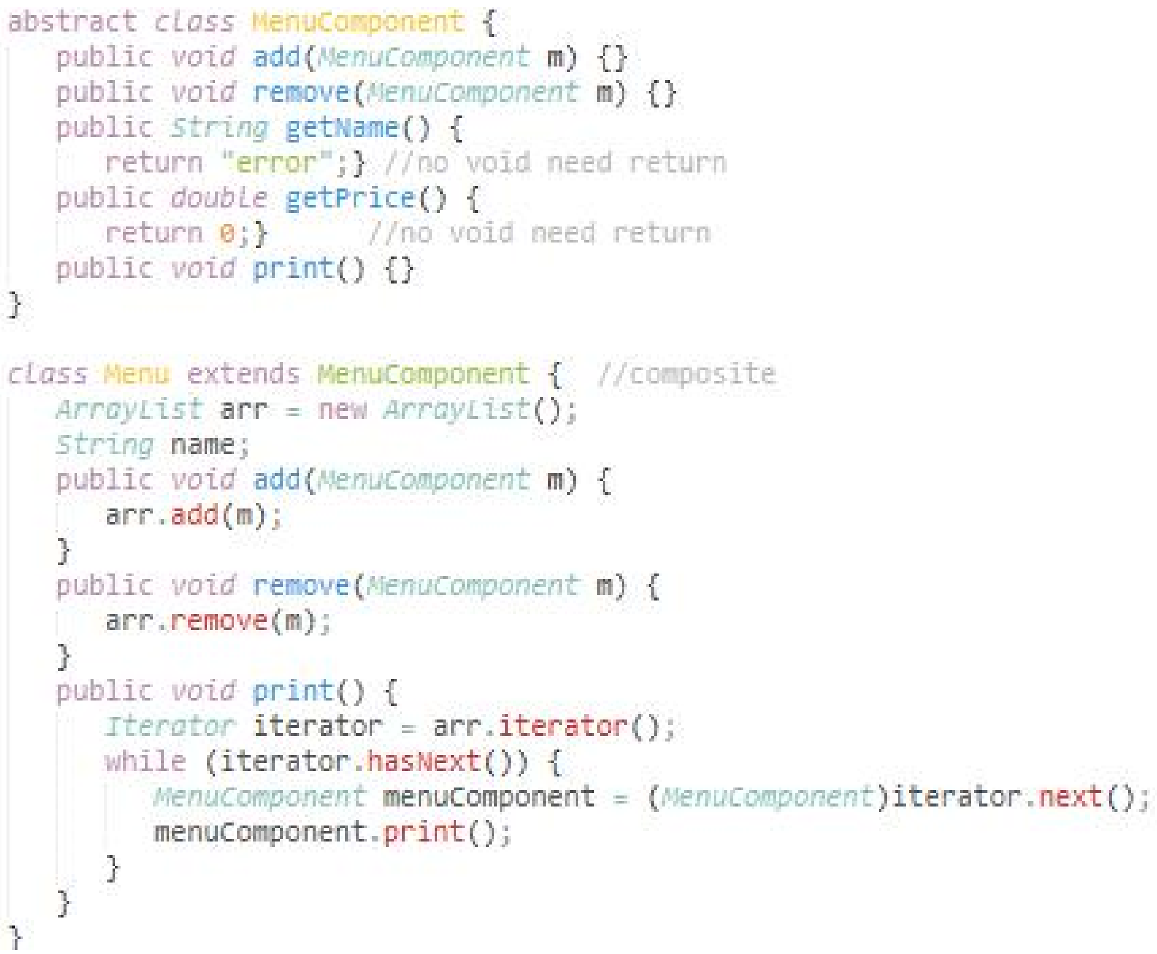
4.成功建立樹

A typical Composite object structure might look like this:

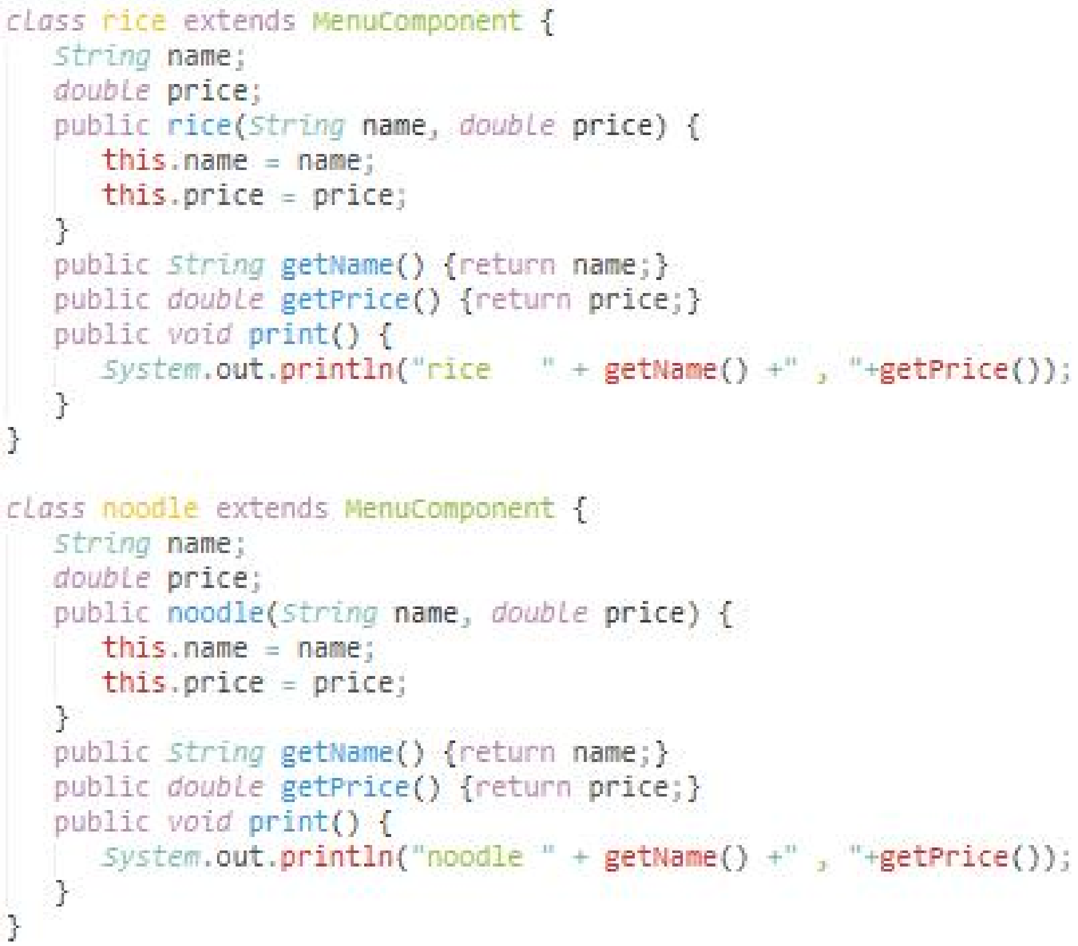



Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

範例1. 基本架構

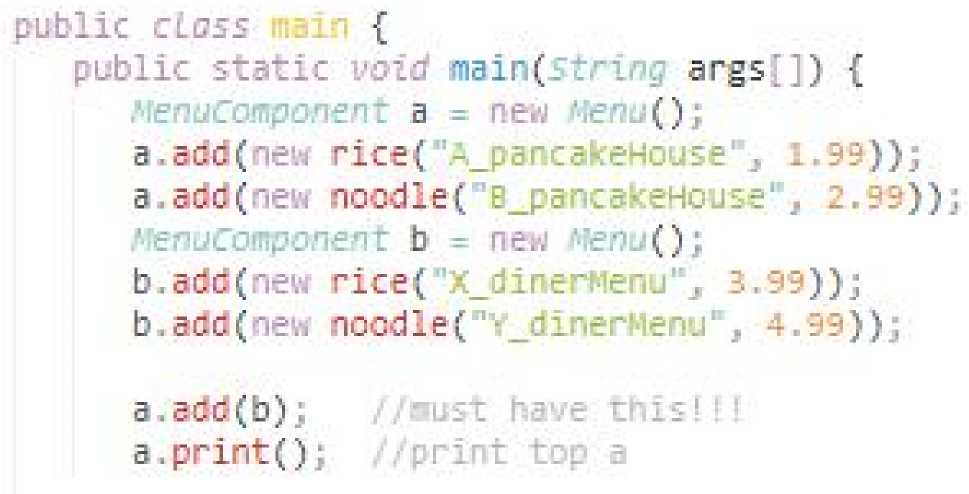


```
abstract class MenuComponent {
    public void add(MenuComponent m) {}
    public void remove(MenuComponent m) {}
    public String getName() {
        return "error";} //no void need return
    public double getPrice() {
        return 0;}      //no void need return
    public void print() {}
}

class Menu extends MenuComponent {  //composite
    ArrayList arr = new ArrayList();
    String name;
    public void add(MenuComponent m) {
        arr.add(m);
    }
    public void remove(MenuComponent m) {
        arr.remove(m);
    }
    public void print() {
        Iterator iterator = arr.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent = (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

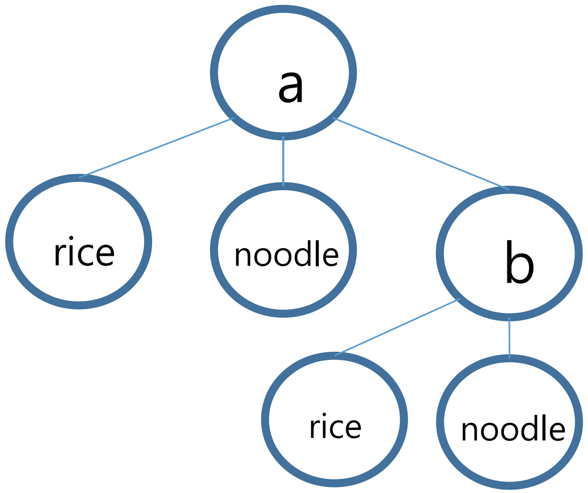


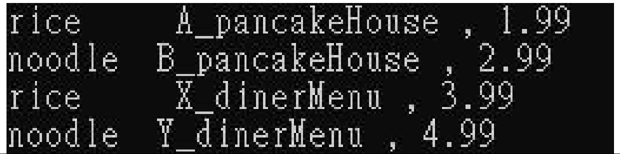
```
class rice extends MenuComponent {
    String name;
    double price;
    public rice(String name, double price) {
        this.name = name;
        this.price = price;
    }
    public String getName() {return name;}
    public double getPrice() {return price;}
    public void print() {
        System.out.println("rice    " + getName() +" , "+getPrice());
    }
}

class noodle extends MenuComponent {
    String name;
    double price;
    public noodle(String name, double price) {
        this.name = name;
        this.price = price;
    }
    public String getName() {return name;}
    public double getPrice() {return price;}
    public void print() {
        System.out.println("noodle " + getName() +" , "+getPrice());
    }
}
```

```
a(MenuComponent)  - b(Menu)    - rice
                  - rice       - noodle
                  - noodle
```



```java
public class main {
    public static void main(String args[]) {
        MenuComponent a = new Menu();
        a.add(new rice("A_pancakeHouse", 1.99));
        a.add(new noodle("B_pancakeHouse", 2.99));
        MenuComponent b = new Menu();
        b.add(new rice("X_dinerMenu", 3.99));
        b.add(new noodle("Y_dinerMenu", 4.99));

        a.add(b);   //must have this!!!
        a.print();  //print top a


    }
}
```



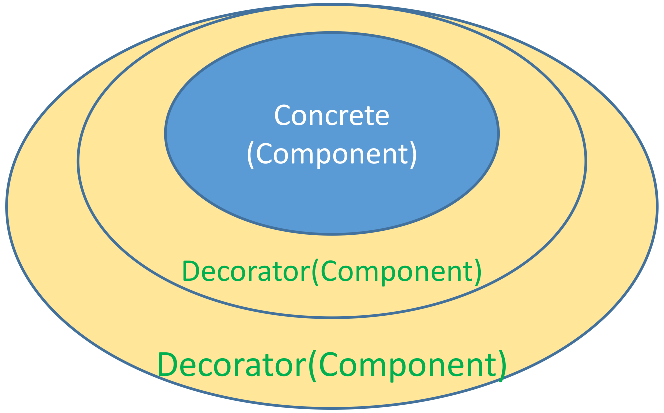


```
rice      A_pancakeHouse , 1.99
noodle  B_pancakeHouse , 2.99
rice      X_dinerMenu , 3.99
noodle  Y_dinerMenu , 4.99
```

9. Decorator 裝飾模式

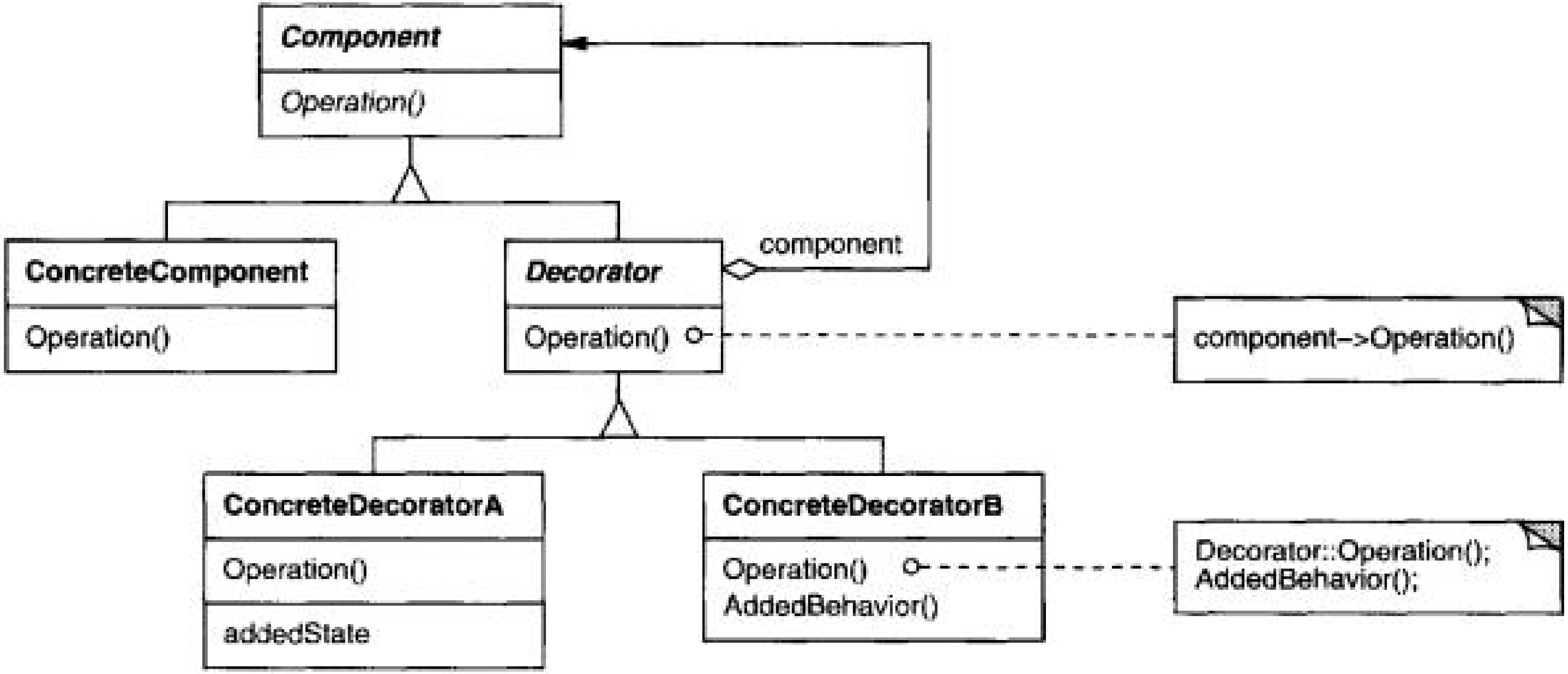
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

動態地將附加職責附加到對象。 裝飾者提供了一個靈活的子類化替代方法，用於擴展功能。

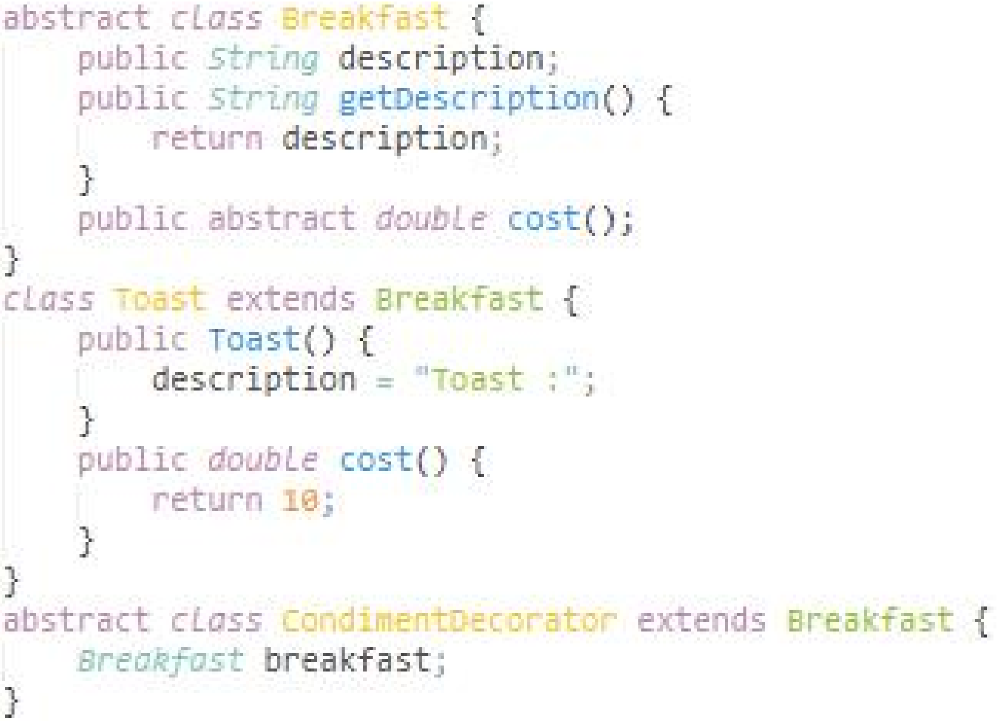


跟Composite相似的結構 重點在可以一層一層的疊上去
用Decorator去包裝ConcreteComponent

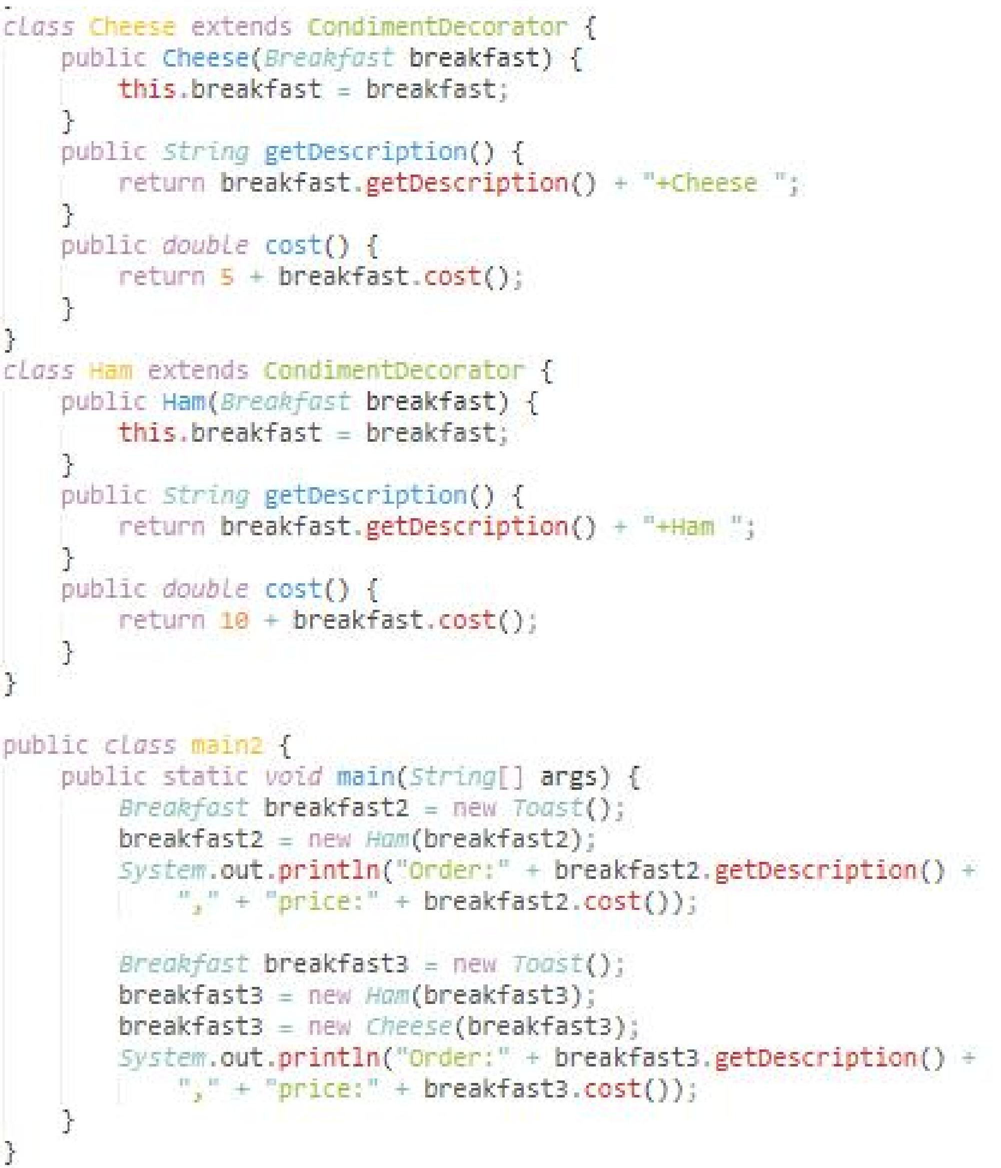
讓原本的Component很容易擴充而不用修改原本寫好的


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

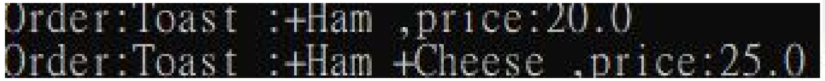
範例1.早餐店吐司加火腿跟起司



```java
abstract class Breakfast {
    public String description;
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
class Toast extends Breakfast {
    public Toast() {
        description = "Toast :";
    }
    public double cost() {
        return 10;
    }
}
abstract class CondimentDecorator extends Breakfast {
    Breakfast breakfast;
}
```

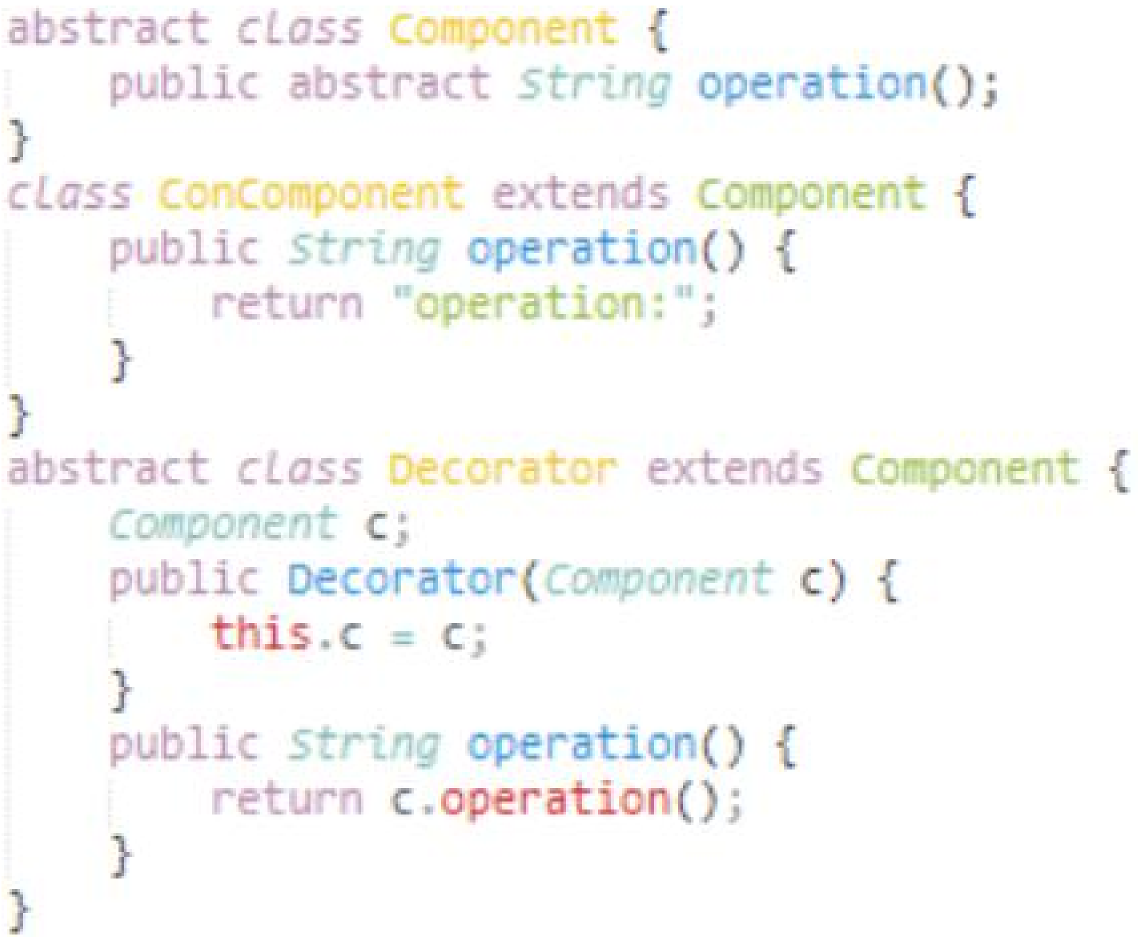


```java
class Cheese extends CondimentDecorator {
    public Cheese(Breakfast breakfast) {
        this.breakfast = breakfast;
    }
    public String getDescription() {
        return breakfast.getDescription() + "+Cheese ";
    }
    public double cost() {
        return 5 + breakfast.cost();
    }
}
class Ham extends CondimentDecorator {
    public Ham(Breakfast breakfast) {
        this.breakfast = breakfast;
    }
    public String getDescription() {
        return breakfast.getDescription() + "+Ham ";
    }
    public double cost() {
        return 10 + breakfast.cost();
    }
}

public class main2 {
    public static void main(String[] args) {
        Breakfast breakfast2 = new Toast();
        breakfast2 = new Ham(breakfast2);
        System.out.println("Order:" + breakfast2.getDescription() +
            "," + "price:" + breakfast2.cost());

        Breakfast breakfast3 = new Toast();
        breakfast3 = new Ham(breakfast3);
        breakfast3 = new Cheese(breakfast3);
        System.out.println("Order:" + breakfast3.getDescription() +
            "," + "price:" + breakfast3.cost());
    }
}
```

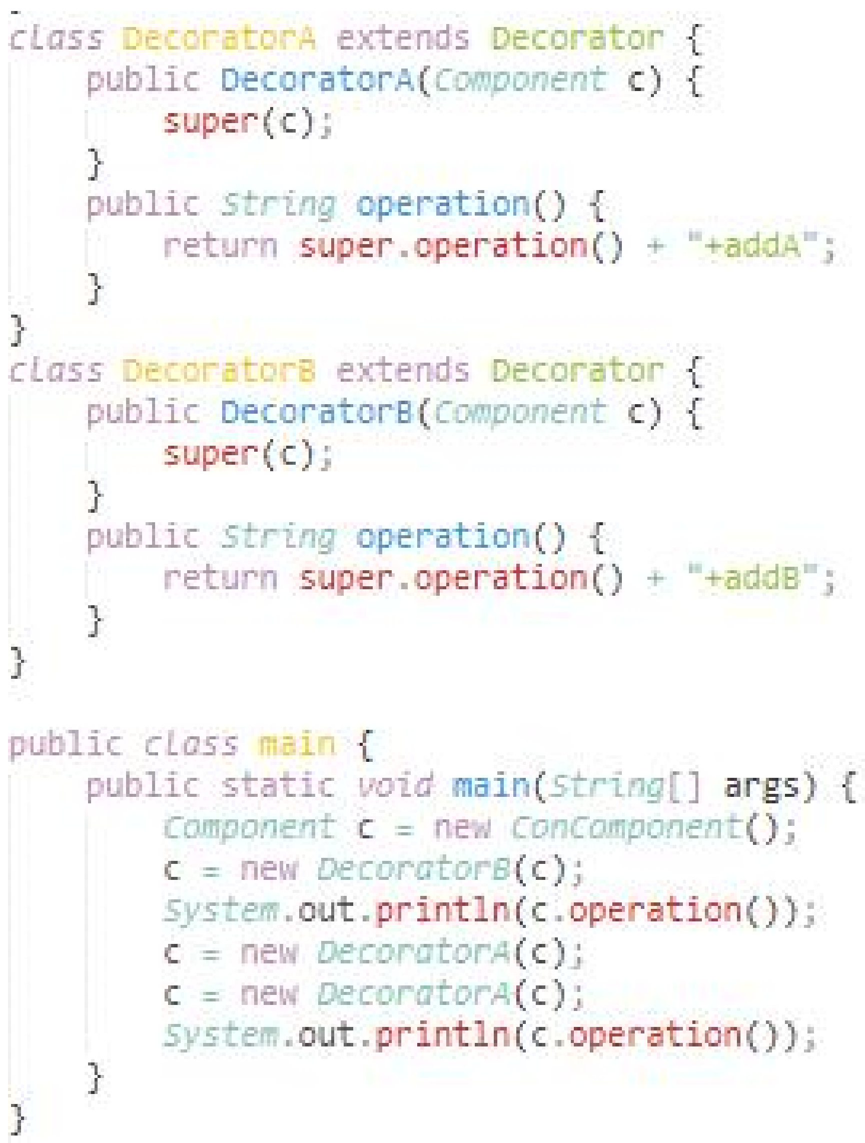


```
Order:Toast :+Ham ,price:20.0
Order:Toast :+Ham +Cheese ,price:25.0
```

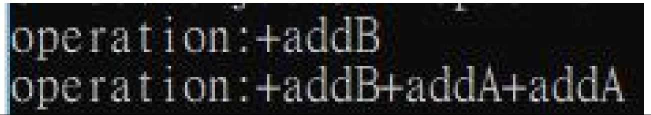
範例2. 程式結構



```java
abstract class Component {
    public abstract String operation();
}
class ConComponent extends Component {
    public String operation() {
        return "operation:";
    }
}
abstract class Decorator extends Component {
    Component c;
    public Decorator(Component c) {
        this.c = c;
    }
    public String operation() {
        return c.operation();
    }
}
```



```java
class DecoratorA extends Decorator {
    public DecoratorA(Component c) {
        super(c);
    }
    public String operation() {
        return super.operation() + "+addA";
    }
}
class DecoratorB extends Decorator {
    public DecoratorB(Component c) {
        super(c);
    }
    public String operation() {
        return super.operation() + "+addB";
    }
}

public class main {
    public static void main(String[] args) {
        Component c = new ConComponent();
        c = new DecoratorB(c);
        System.out.println(c.operation());
        c = new DecoratorA(c);
        c = new DecoratorA(c);
        System.out.println(c.operation());
    }
}
```



```
operation:+addB
operation:+addB+addA+addA
```

10. Facade 外觀模式

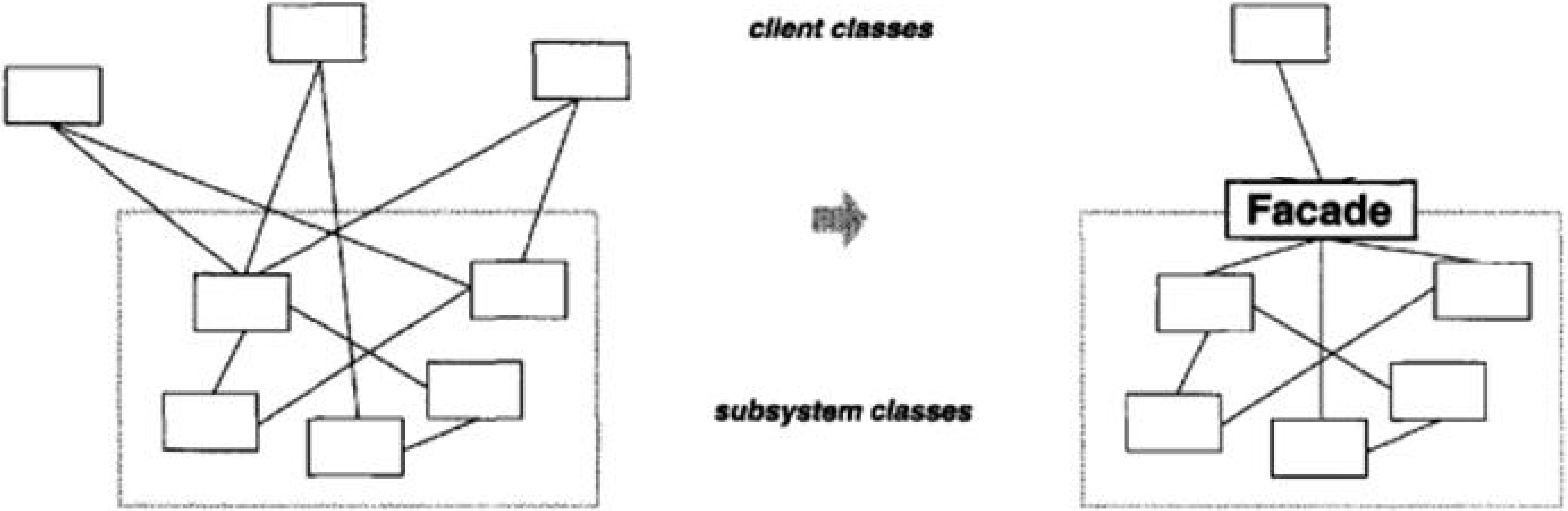
Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem easier to use.

為子系統中的一組接口提供統一接口。 Facade定義了一個更高級別的界面，使子系統更易於使用。

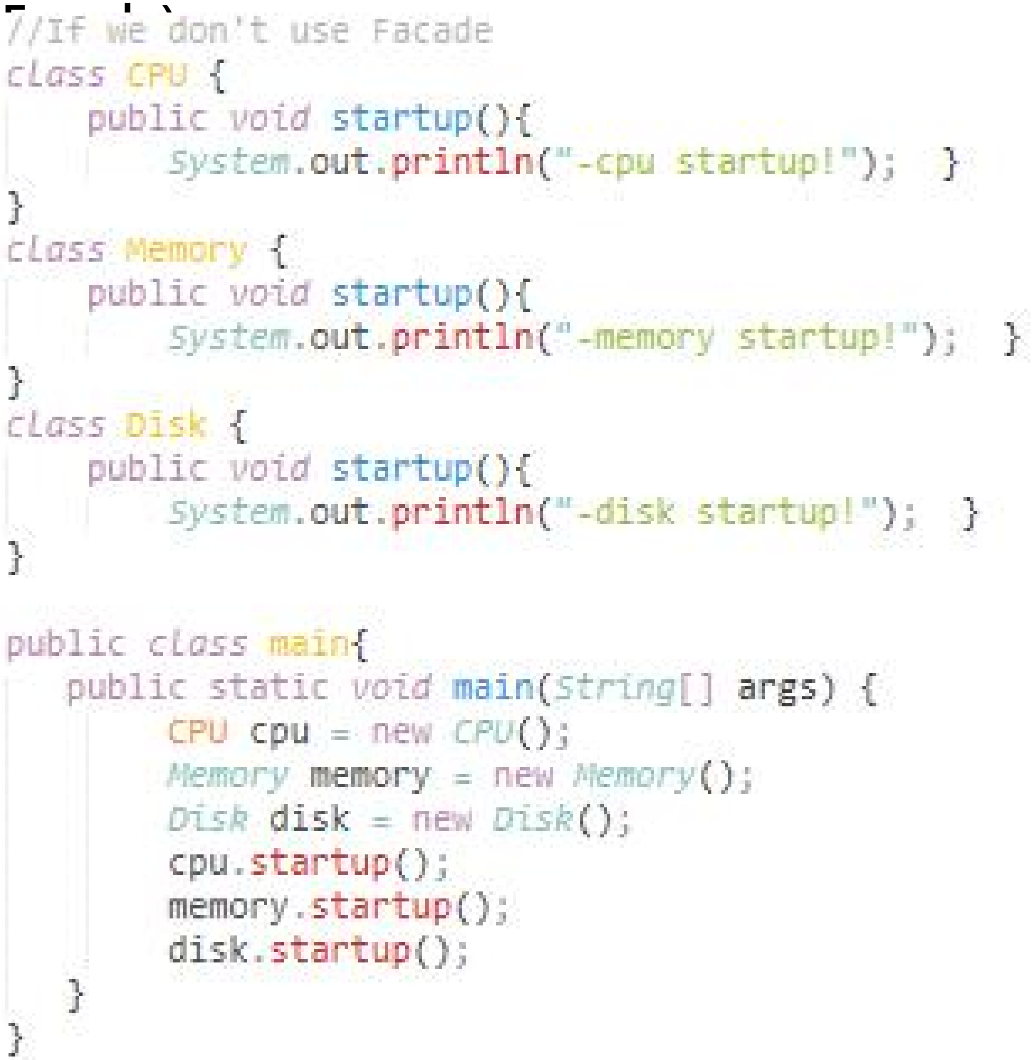
- 把原本客戶端跟子系統
  間錯綜複雜的關係簡化

- 讓客戶端透過一個介面
  就能夠使用所有的功能

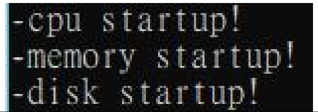
- 客戶端也不會知道有多少
  子系統在運作只會知道介面


client classes
Facade
subsystem classes

範例1. 透過一個介面使用所有功能(如果沒有使用Facade)

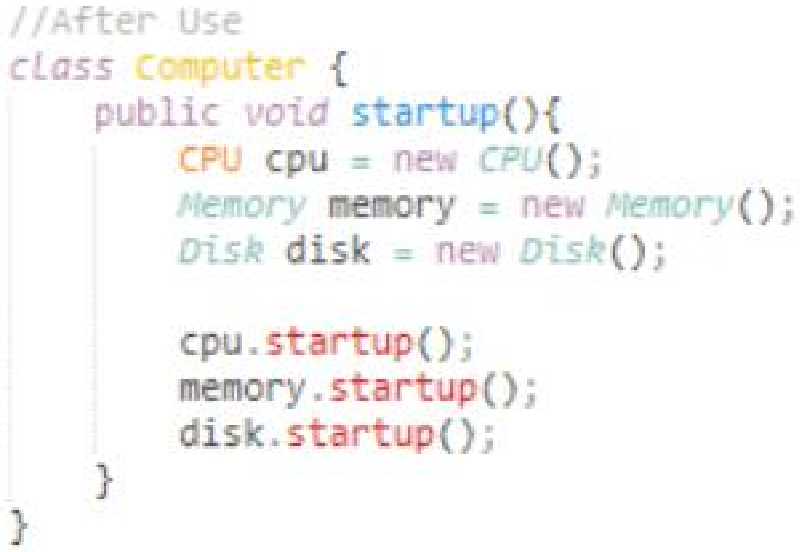


```java
//If we don't use Facade
class CPU {
    public void startup(){
        System.out.println("-cpu startup!");  }
}
class Memory {
    public void startup(){
        System.out.println("-memory startup!");  }
}
class Disk {
    public void startup(){
        System.out.println("-disk startup!");  }
}

public class main{
   public static void main(String[] args) {
        CPU cpu = new CPU();
        Memory memory = new Memory();
        Disk disk = new Disk();
        cpu.startup();
        memory.startup();
        disk.startup();
   }
}
```

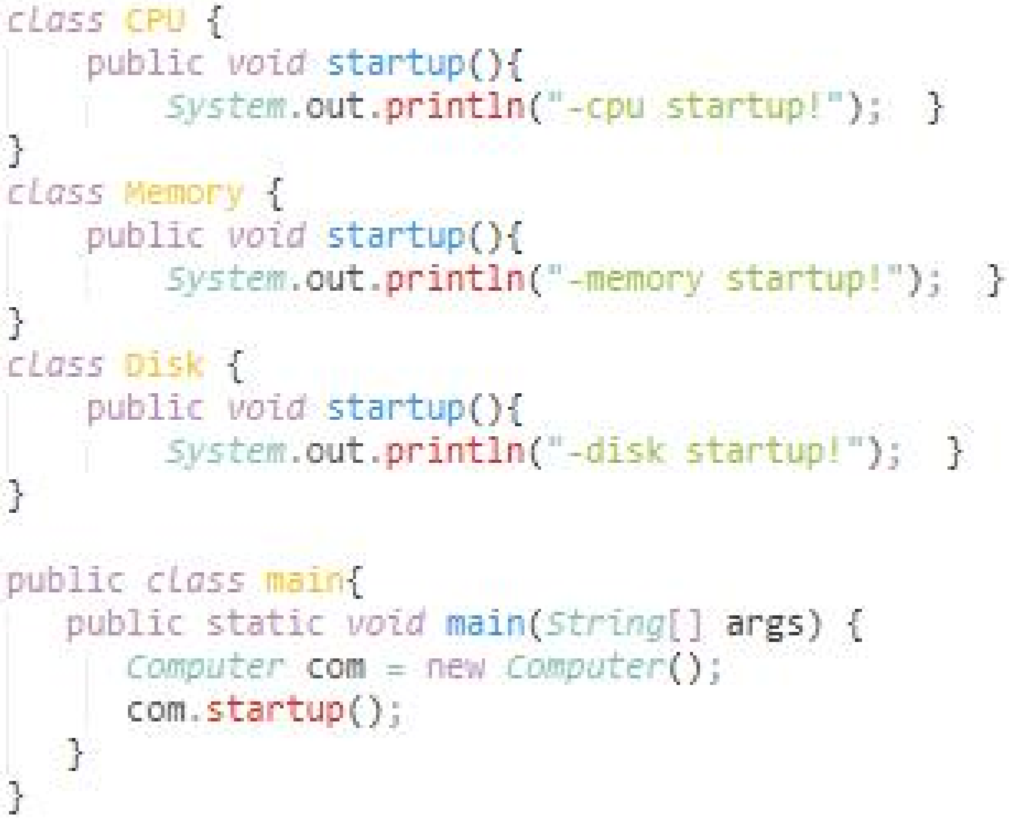


```
-cpu startup!
-memory startup!
-disk startup!
```

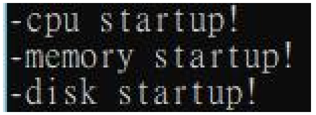
範例2. 透過一個介面使用所有功能(使用Façade)



```
//After Use
class Computer {
    public void startup(){
        CPU cpu = new CPU();
        Memory memory = new Memory();
        Disk disk = new Disk();

        cpu.startup();
        memory.startup();
        disk.startup();
    }
}
```



```
class CPU {
    public void startup(){
        System.out.println("-cpu startup!");  }
}
class Memory {
    public void startup(){
        System.out.println("-memory startup!");  }
}
class Disk {
    public void startup(){
        System.out.println("-disk startup!");  }
}

public class main{
   public static void main(String[] args) {
      Computer com = new Computer();
      com.startup();
   }
}
```



```
-cpu startup!
-memory startup!
-disk startup!
```

11. Flyweight 享元模式

Use sharing to support large numbers of fine-grained objects efficiently.

使用共享可以有效地支持大量細粒度對象。

共享物件，用來盡可能減少記憶體使用量
以及分享資訊給儘可能多的相似物件。

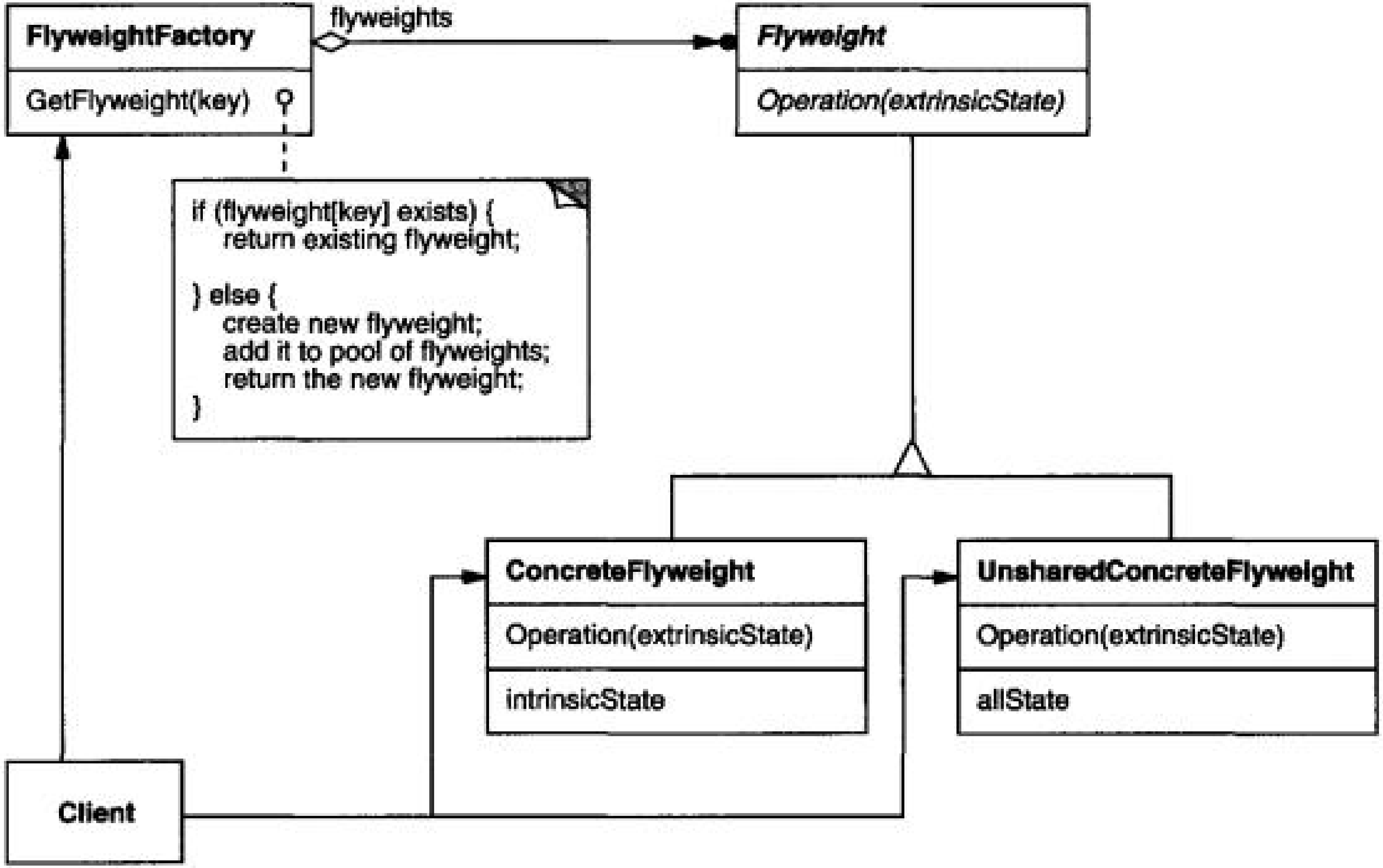
- Intrinsic: 可被共享的
  Extrinsic:不被共享的

- Flyweight(所有具體享元的父類別)
  為這些類規定需要實現的公共接口。

- ConcreteFlyweight
  實現Flyweight接口，並為內部狀態拉回存儲空間。

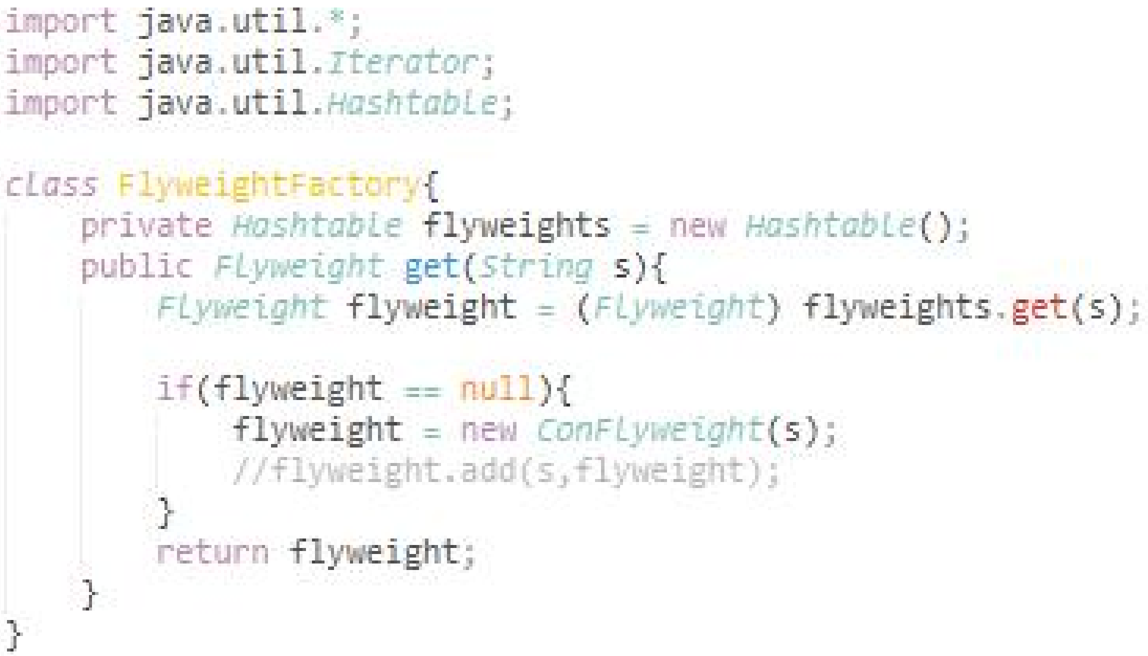
- FlyweightFactory：
  負責創建和管理享元角色。

- Client：
  需要存儲所有享元對象的外部狀態。


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;

} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

範例1. 基本架構

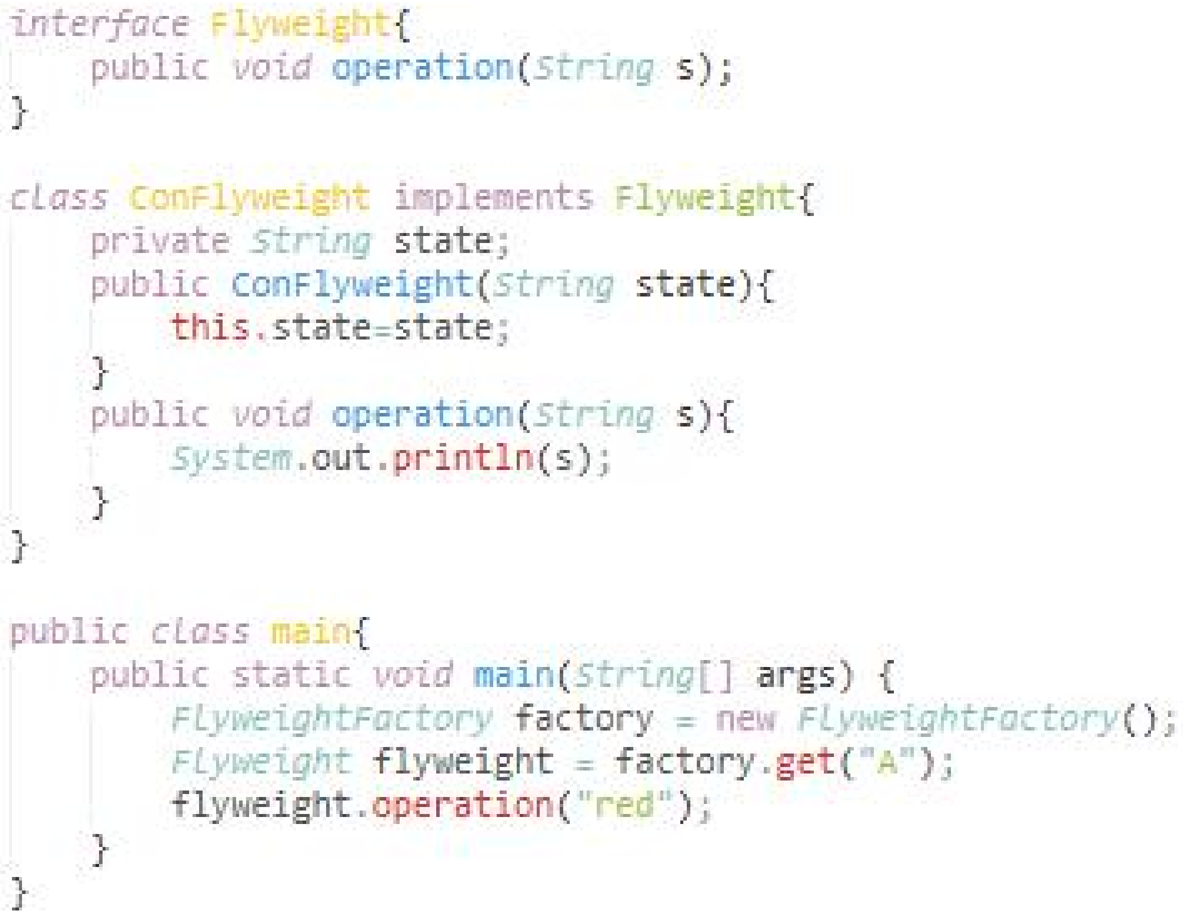


```
import java.util.*;
import java.util.Iterator;
import java.util.Hashtable;

class FlyweightFactory{
    private Hashtable flyweights = new Hashtable();
    public Flyweight get(String s){
        Flyweight flyweight = (Flyweight) flyweights.get(s);

        if(flyweight == null){
            flyweight = new ConFlyweight(s);
            //flyweight.add(s,flyweight);
        }
        return flyweight;
    }
}
```

ConcreteFlyweight:

儲存在內部的state
使用時可以共享出去

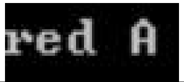
不被共享s的在使用時才取得



```
interface Flyweight{
    public void operation(String s);
}

class ConFlyweight implements Flyweight{
    private String state;
    public ConFlyweight(String state){
        this.state=state;
    }
    public void operation(String s){
        System.out.println(s);
    }
}

public class main{
    public static void main(String[] args) {
        FlyweightFactory factory = new FlyweightFactory();
        Flyweight flyweight = factory.get("A");
        flyweight.operation("red");
    }
}
```

Factory使用Hashtable存放Flyweight物件

要取得Flyweight時先從HashTable裡面找
找不到再建造一個新的，減少記憶體空間的使用



```
red A
```

12. Proxy 代理模式

Provide a surrogate or placeholder for another object to control access to it.

為另一個對象提供代理或占位符以控制對它的訪問。

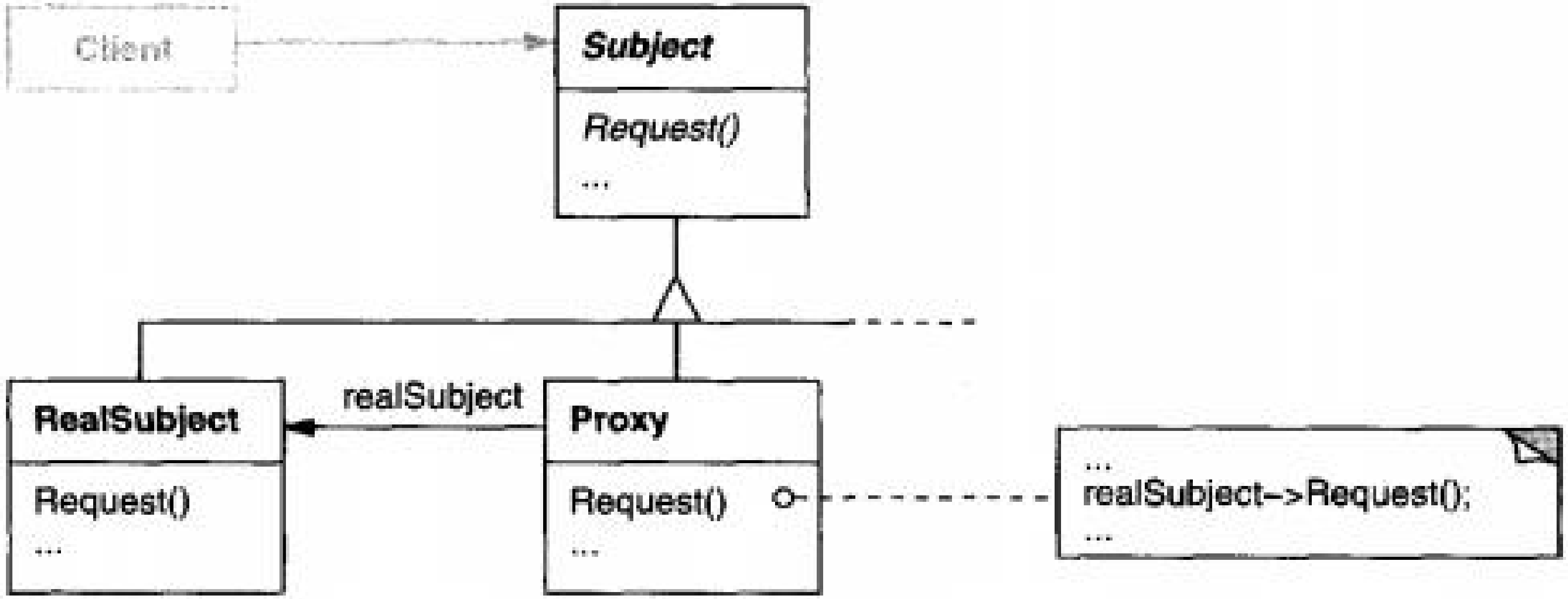
透過代理人來負責所有的事情
根據功能不同大致可以分四種

- 虛擬(Virtual Proxy)
  用比較不消耗資源的代理物件來代替實際物件
  實際物件只有在真正需要才會被創造
- 遠程(Remote Proxy)
  在本地端提供一個代表物件來存取遠端網址的物件
- 保護(Protect Proxy)
  限制其他程式存取權限
- 智能(Smart Reference Proxy)
  為被代理的物件增加一些動作

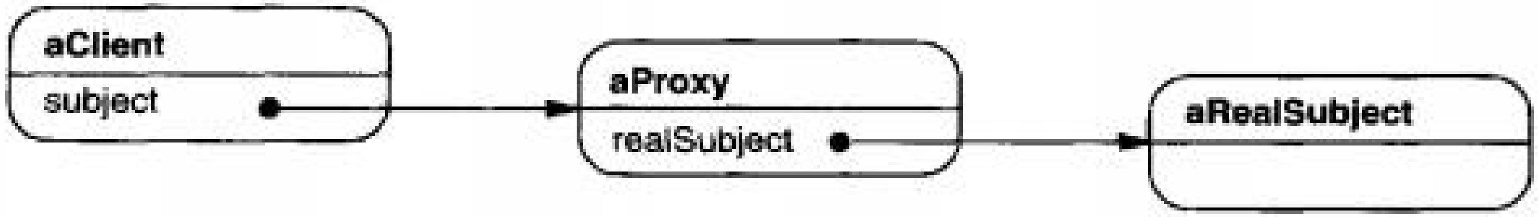
【Proxy vs Decorator】
相同點在都需要實現同一個接口或者繼承同一個抽像類，
並且代理角色和裝飾角色都持有被角色的引用。

代理模式是對整個對象的行為控制和限制，而非針對功能，
Ex.代理/老闆請財務處理事務，有必要的事情才到達老闆

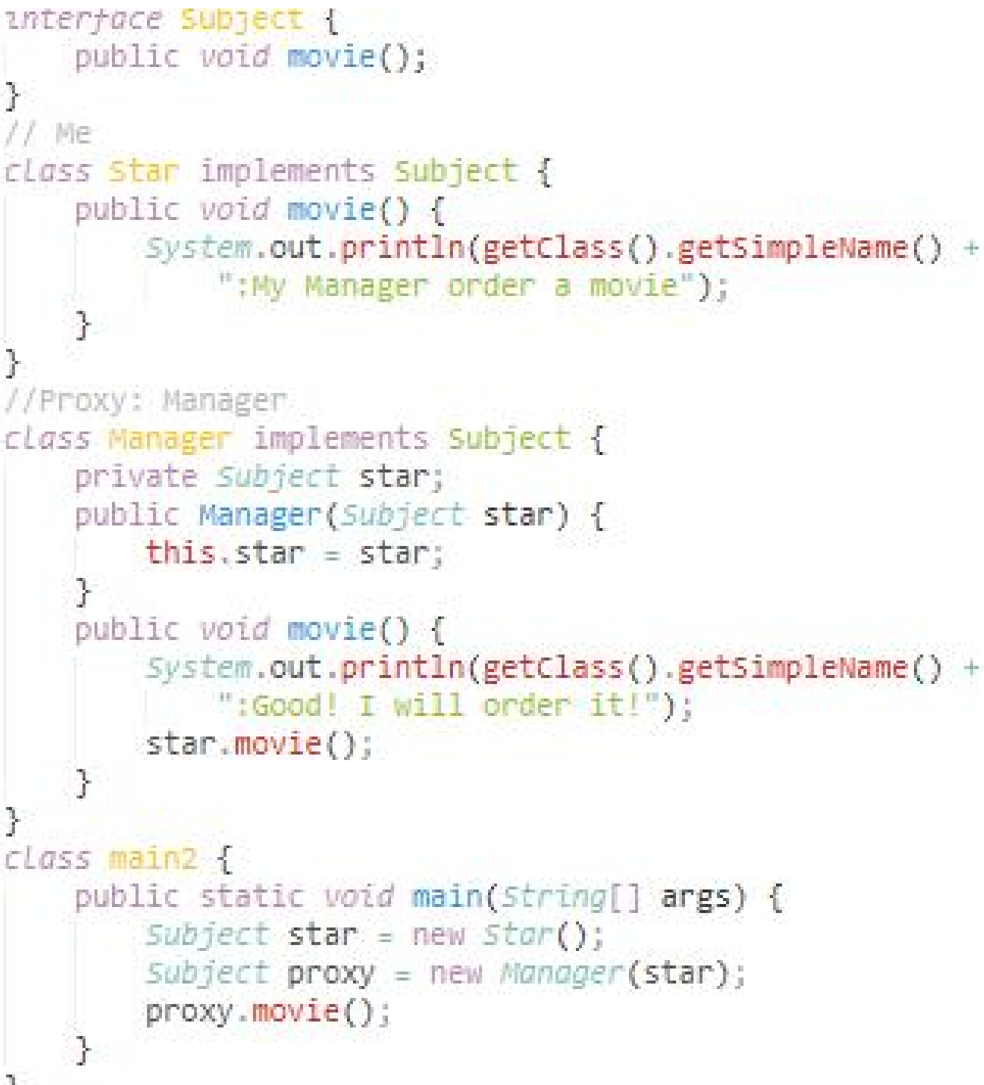
裝飾模式針對的是增加對象的職能，也就是屬性或者方法。
EX. 裝飾/老闆身上加，老闆還要懂財務會計。



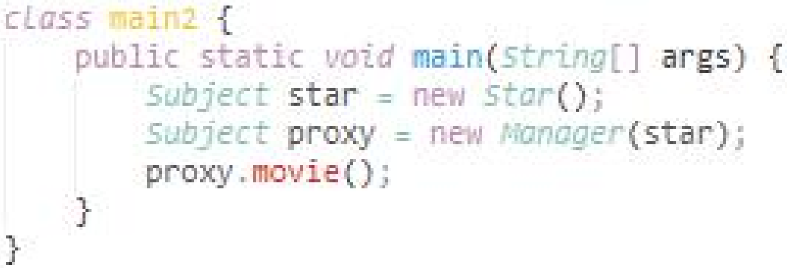
Here's a possible object diagram of a proxy structure at run-time:



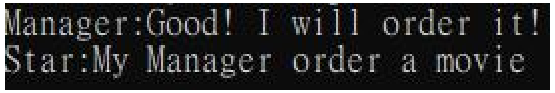
範例1. 我的經紀人幫我接下電影



```
interface Subject {
    public void movie();
}
// Me
class Star implements Subject {
    public void movie() {
        System.out.println(getClass().getSimpleName() +
            ":My Manager order a movie");
    }
}
//Proxy: Manager
class Manager implements Subject {
    private Subject star;
    public Manager(Subject star) {
        this.star = star;
    }
    public void movie() {
        System.out.println(getClass().getSimpleName() +
            ":Good! I will order it!");
        star.movie();
    }
}
class main2 {
    public static void main(String[] args) {
        Subject star = new Star();
        Subject proxy = new Manager(star);
        proxy.movie();
    }
}
```

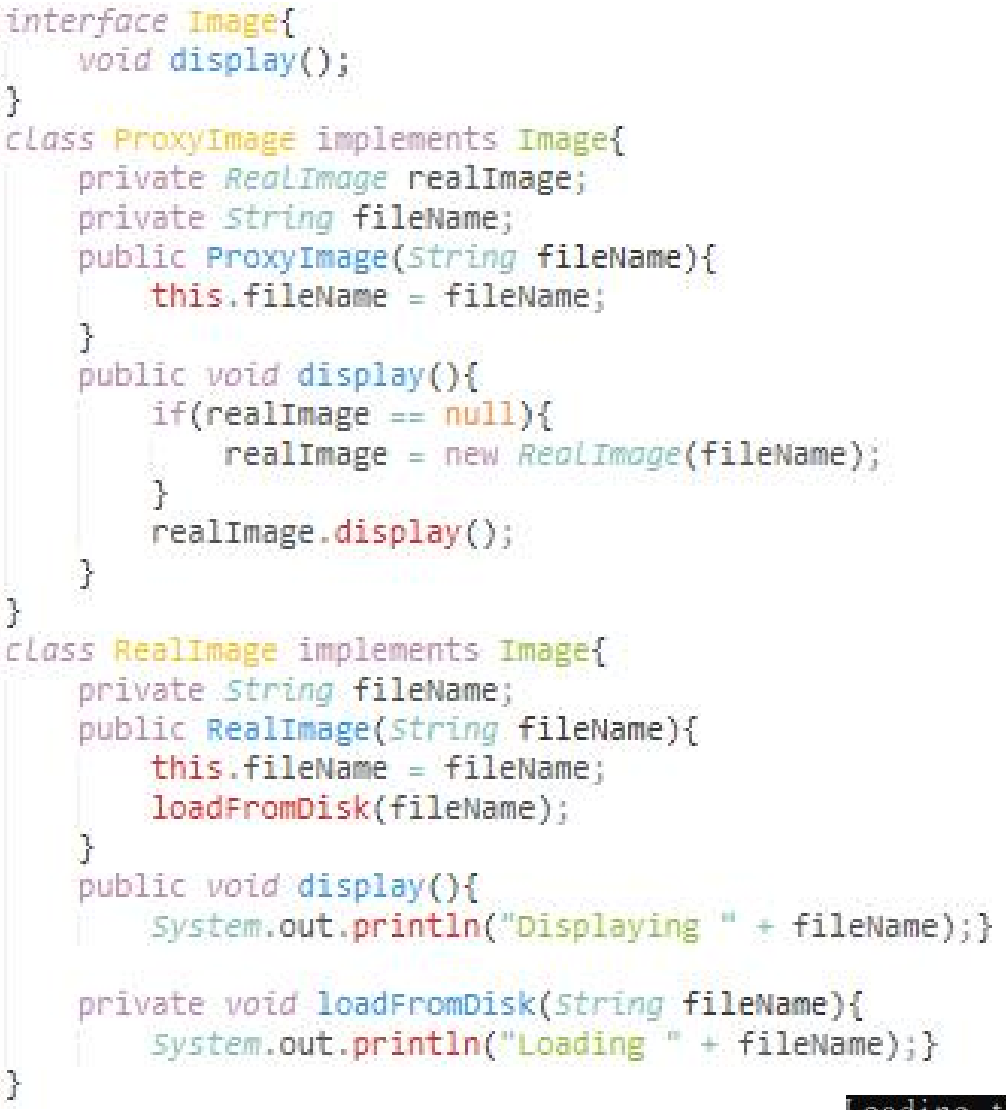


```
class main2 {
    public static void main(String[] args) {
        Subject star = new Star();
        Subject proxy = new Manager(star);
        proxy.movie();
    }
}
```

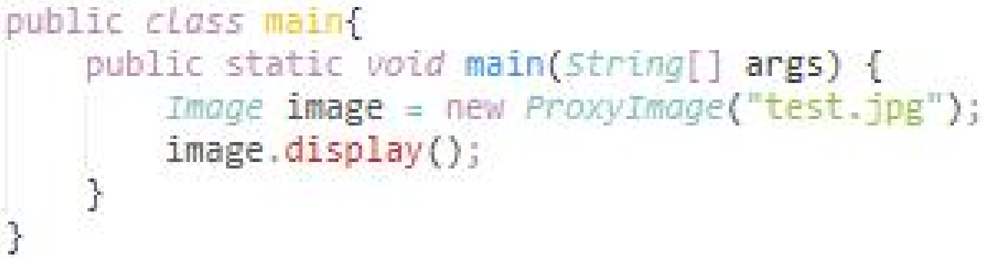


```
Manager:Good! I will order it!
Star:My Manager order a movie
```

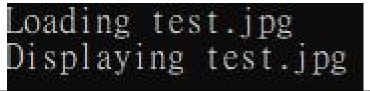
範例2. 讀取照片



```
interface Image{
    void display();
}
class ProxyImage implements Image{
    private RealImage realImage;
    private String fileName;
    public ProxyImage(String fileName){
        this.fileName = fileName;
    }
    public void display(){
        if(realImage == null){
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
class RealImage implements Image{
    private String fileName;
    public RealImage(String fileName){
        this.fileName = fileName;
        loadFromDisk(fileName);
    }
    public void display(){
        System.out.println("Displaying " + fileName);}

    private void loadFromDisk(String fileName){
        System.out.println("Loading " + fileName);}
}
```



```
public class main{
    public static void main(String[] args) {
        Image image = new ProxyImage("test.jpg");
        image.display();
    }
}
```

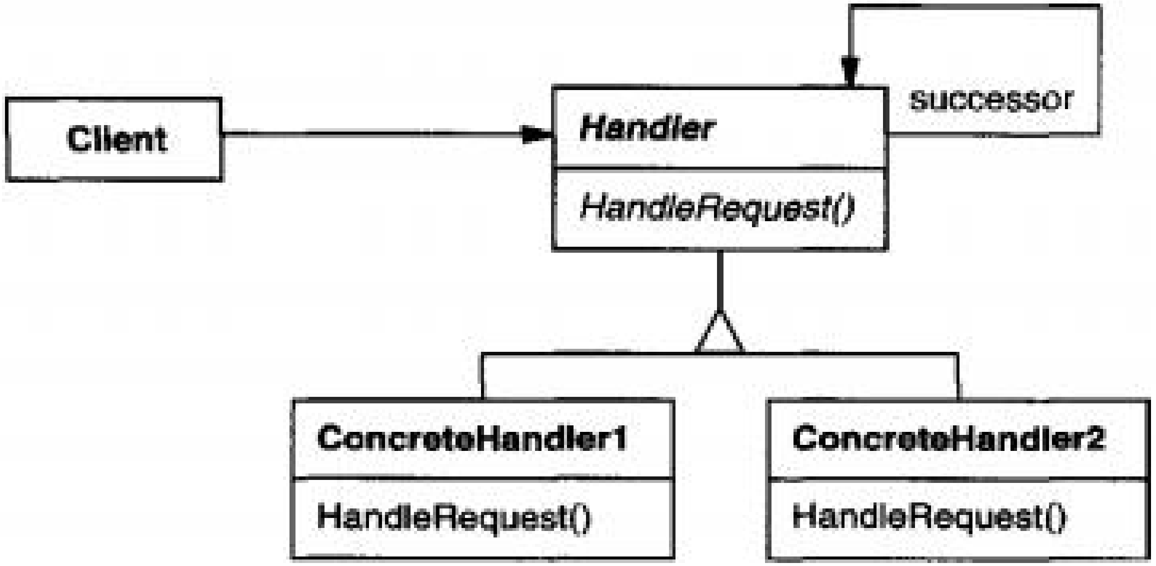


```
Loading test.jpg
Displaying test.jpg
```

13. Chain of Responsibility 責任鏈模式

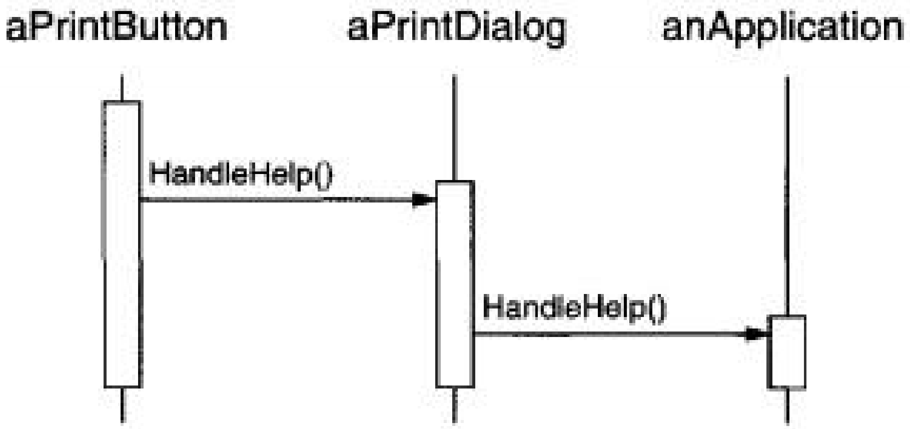
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

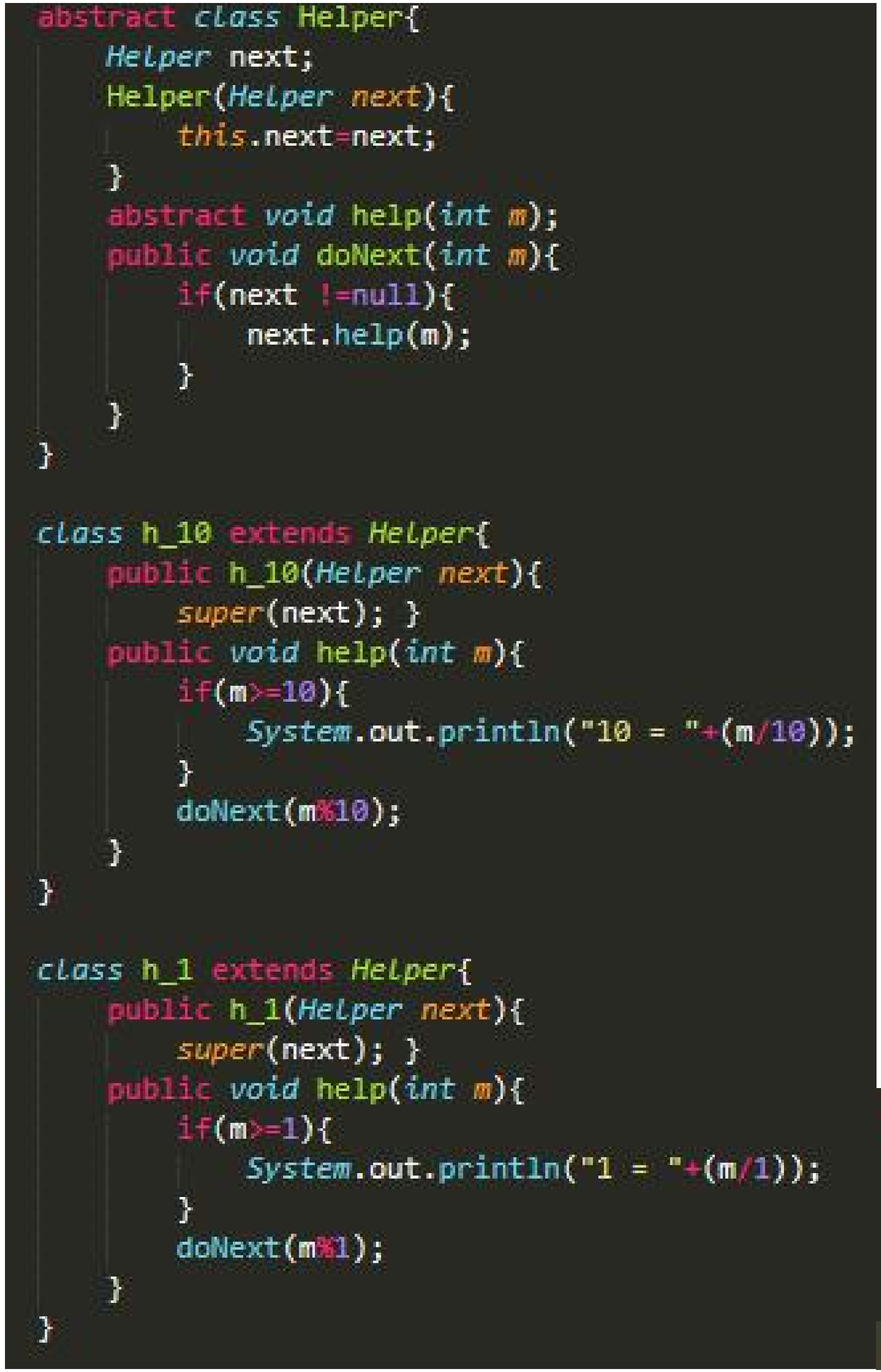
避免通過提供多個請求將發送者的請求與其接收者耦合對象有機會處理請求。
鏈接接收對象並通過沿鏈請求，直到對象處理它。



這個Pattern很簡單
就是把很多個處理器串在一起
當這個處理器無法處理就交給下一個

可以處理越多事情的Handler放越後面





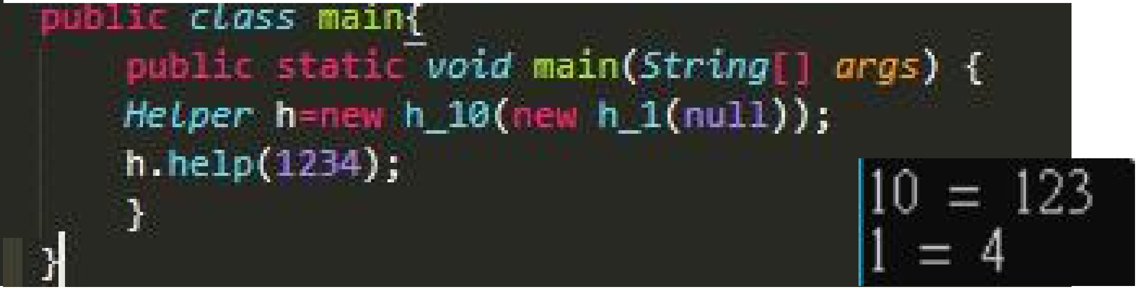
```java
abstract class Helper{
    Helper next;
    Helper(Helper next){
        this.next=next;
    }
    abstract void help(int m);
    public void doNext(int m){
        if(next !=null){
            next.help(m);
        }
    }
}

class h_10 extends Helper{
    public h_10(Helper next){
        super(next); }
    public void help(int m){
        if(m>=10){
            System.out.println("10 = "+(m/10));
        }
        doNext(m%10);
    }
}

class h_1 extends Helper{
    public h_1(Helper next){
        super(next); }
    public void help(int m){
        if(m>=1){
            System.out.println("1 = "+(m/1));
        }
        doNext(m%1);
    }
}
```

1.每個Helper裡面要放下一個Helper
在建構子中放置下一個Helper

2.判斷下一個Helper 是不是Null
不是就交給下一位 next.help

3.零錢處理器現在這個是處理10的，只負責10塊
用取餘數，把10取完後交給next

4.把所有Helper串起，沒有下一個就放null



```java
public class main{
    public static void main(String[] args) {
    Helper h=new h_10(new h_1(null));
    h.help(1234);
    }
}
```

```
10 = 123
1 = 4
```

14. Command 命令模式

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

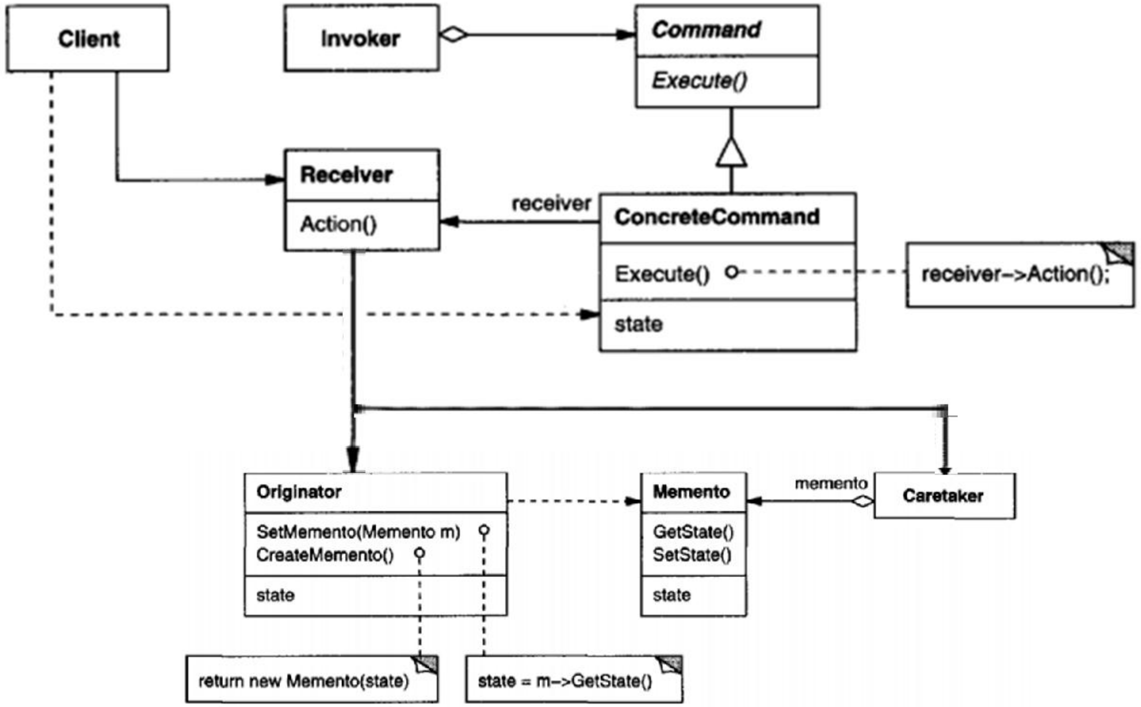
將請求封裝為對象，從而允許您使用參數化客戶戶端不同的請求，隊列或日誌請求，並支持可撤銷的操作。

把常用、常常重複的指令包裝成Command
需要用到的時候就透過Command來執行

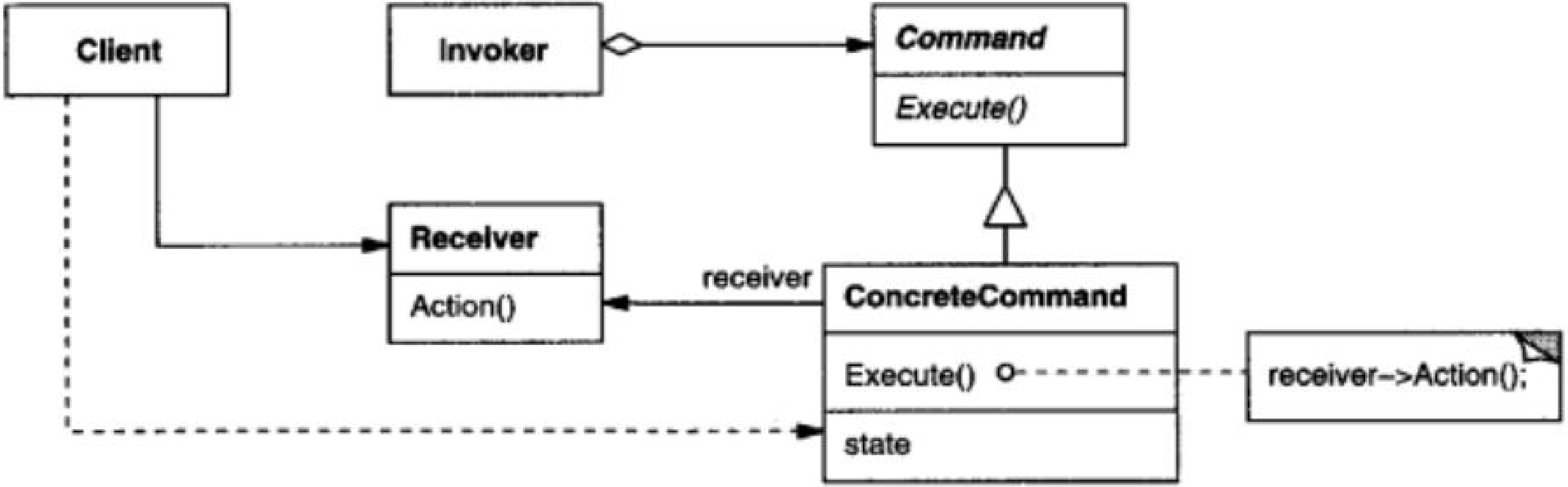
-Invoker 用來執行Command的
-Command 操作Reciver的
-Reciver是真正執行功能的

- Client -> Invoker ->command -> Receiver
- 客人 服務生 點餐 廚師

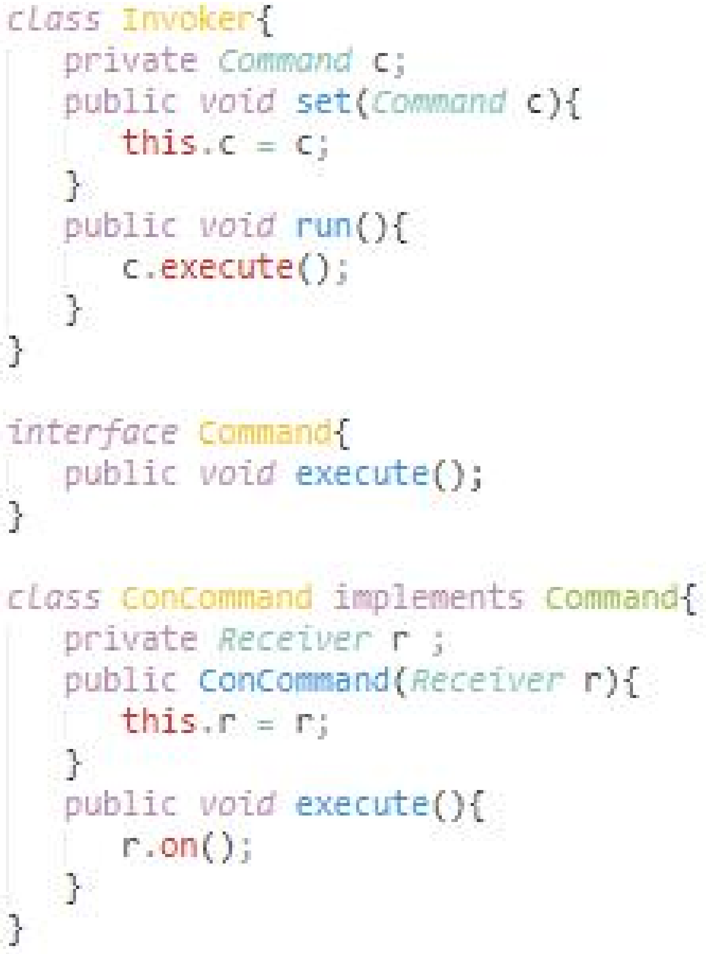
【Command + Memento】



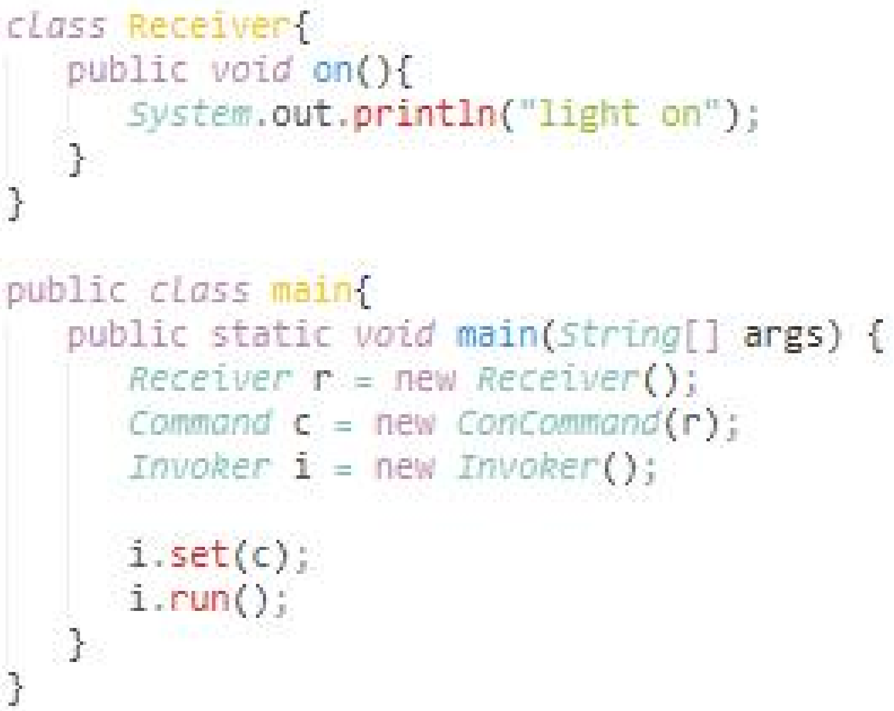
(Client) → (Get. Set.)



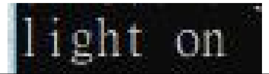
範例1.我要開燈 (執行一個動作)



```
class Invoker{
    private Command c;
    public void set(Command c){
        this.c = c;
    }
    public void run(){
        c.execute();
    }
}

interface Command{
    public void execute();
}

class ConCommand implements Command{
    private Receiver r ;
    public ConCommand(Receiver r){
        this.r = r;
    }
    public void execute(){
        r.on();
    }
}
```

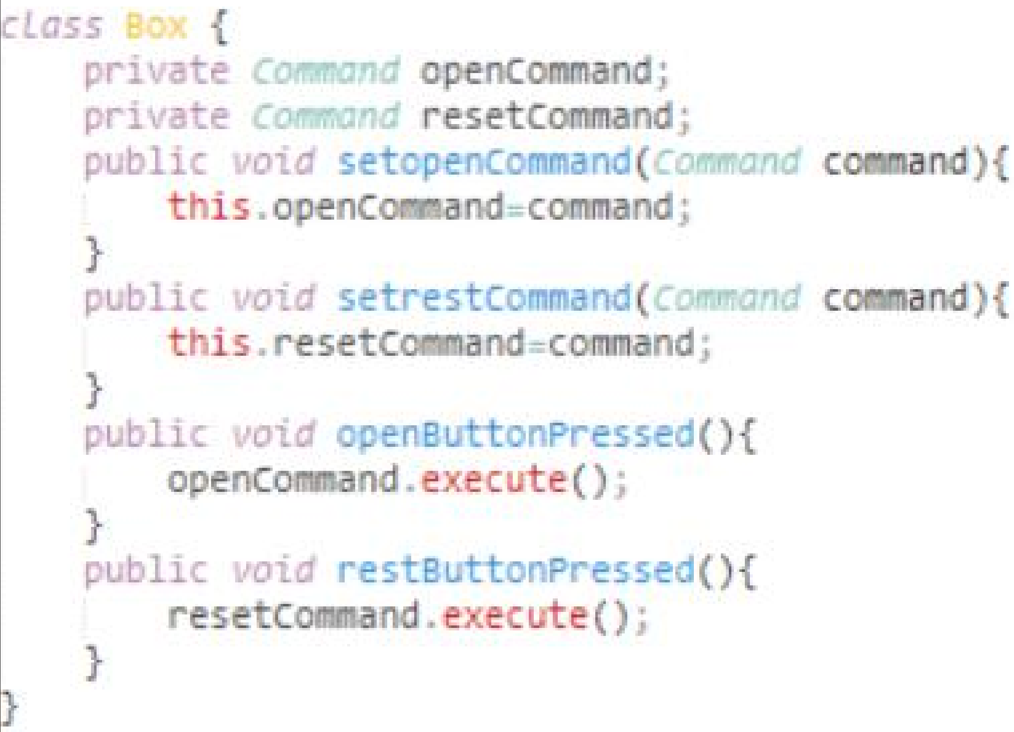


```
class Receiver{
    public void on(){
        System.out.println("light on");
    }
}

public class main{
    public static void main(String[] args) {
        Receiver r = new Receiver();
        Command c = new ConCommand(r);
        Invoker i = new Invoker();

        i.set(c);
        i.run();
    }
}
```

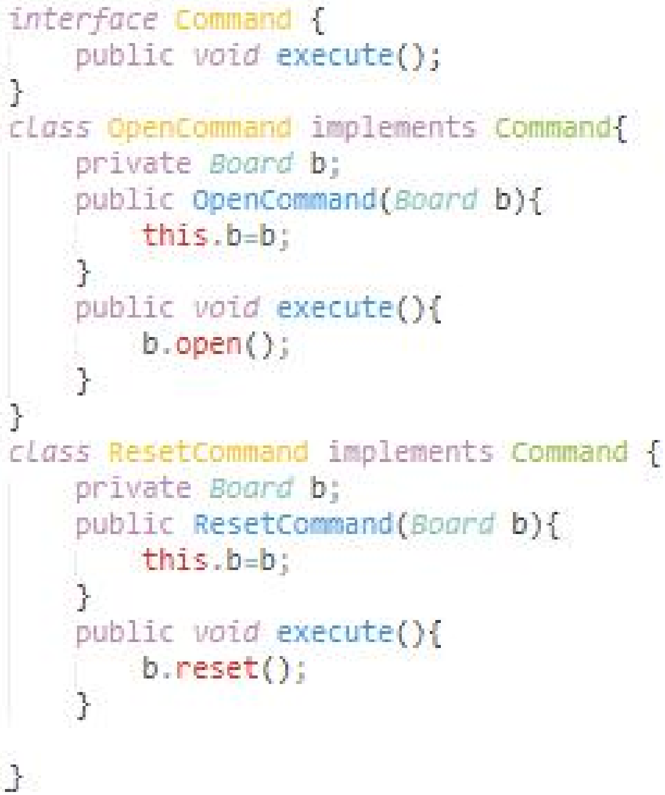


```
light on
```

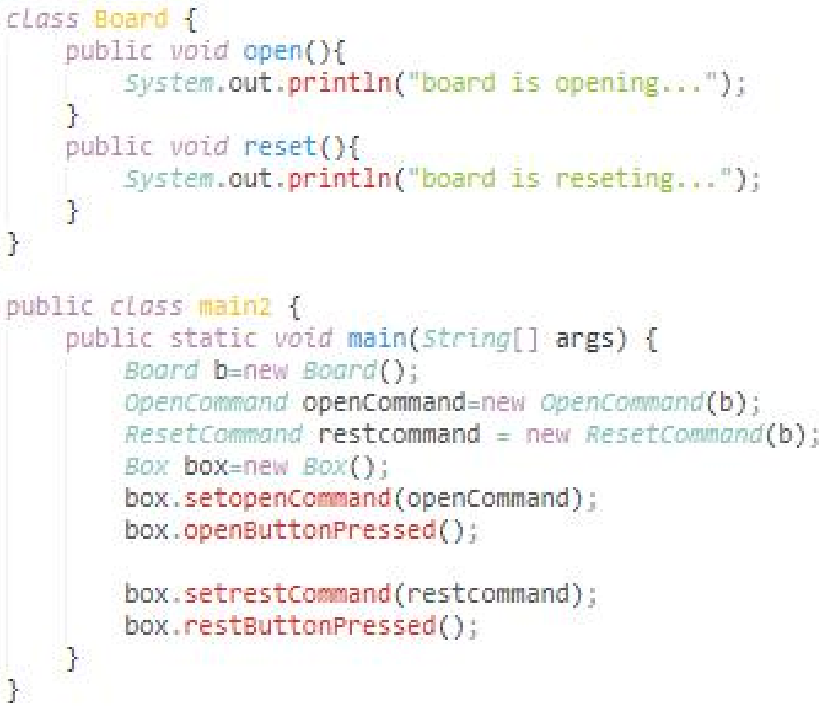
範例2.我要讓主機板開機 & 重開機 (執行兩個動作)



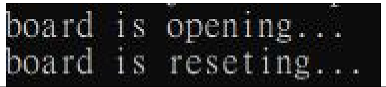
```java
class Box {
    private Command openCommand;
    private Command resetCommand;
    public void setopenCommand(Command command){
        this.openCommand=command;
    }
    public void setrestCommand(Command command){
        this.resetCommand=command;
    }
    public void openButtonPressed(){
        openCommand.execute();
    }
    public void restButtonPressed(){
        resetCommand.execute();
    }
}
```



```java
interface Command {
    public void execute();
}
class OpenCommand implements Command{
    private Board b;
    public OpenCommand(Board b){
        this.b=b;
    }
    public void execute(){
        b.open();
    }
}
class ResetCommand implements Command {
    private Board b;
    public ResetCommand(Board b){
        this.b=b;
    }
    public void execute(){
        b.reset();
    }

}
```



```java
class Board {
    public void open(){
        System.out.println("board is opening...");
    }
    public void reset(){
        System.out.println("board is reseting...");
    }
}

public class main2 {
    public static void main(String[] args) {
        Board b=new Board();
        OpenCommand openCommand=new OpenCommand(b);
        ResetCommand restcommand = new ResetCommand(b);
        Box box=new Box();
        box.setopenCommand(openCommand);
        box.openButtonPressed();

        box.setrestCommand(restcommand);
        box.restButtonPressed();
    }
}
```

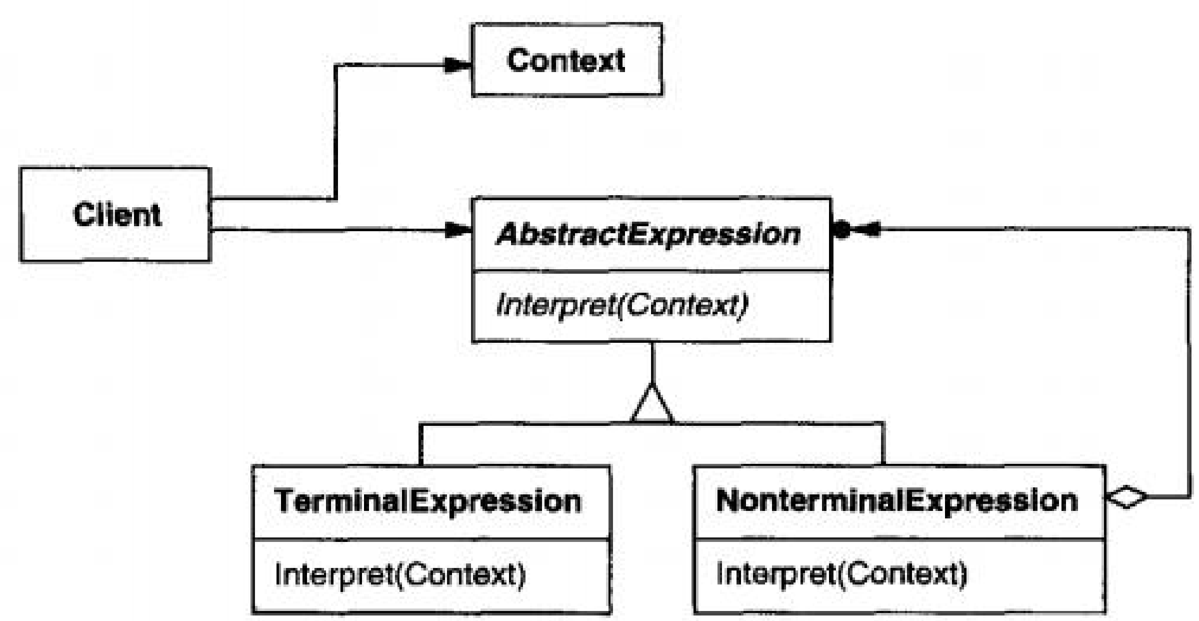


```
board is opening...
board is reseting...
```

15. Interpreter 解釋器模式 ***

Given a language, define a represention forits grammar along with an interpreter
that uses the representation to interpret sentences in the language.

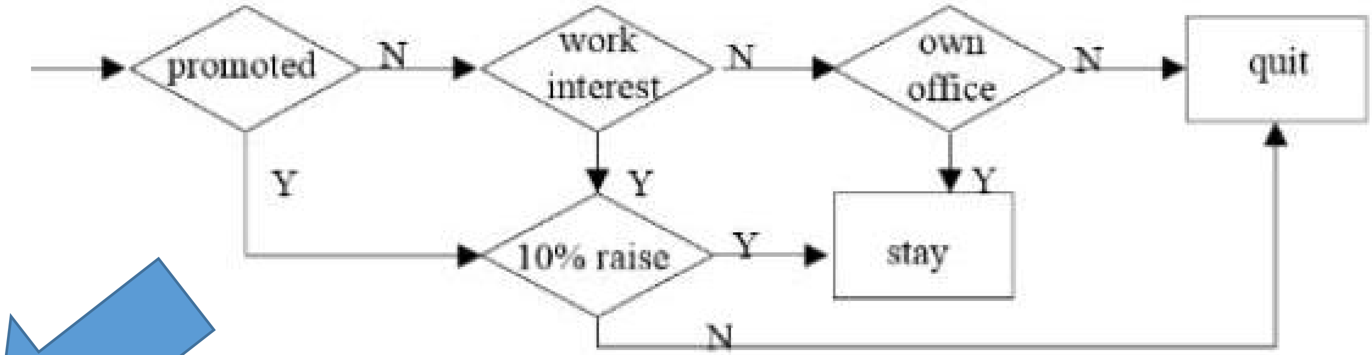
給定一種語言，定義其語法的表示以及解釋器
使用表示來解釋語言中的句子。

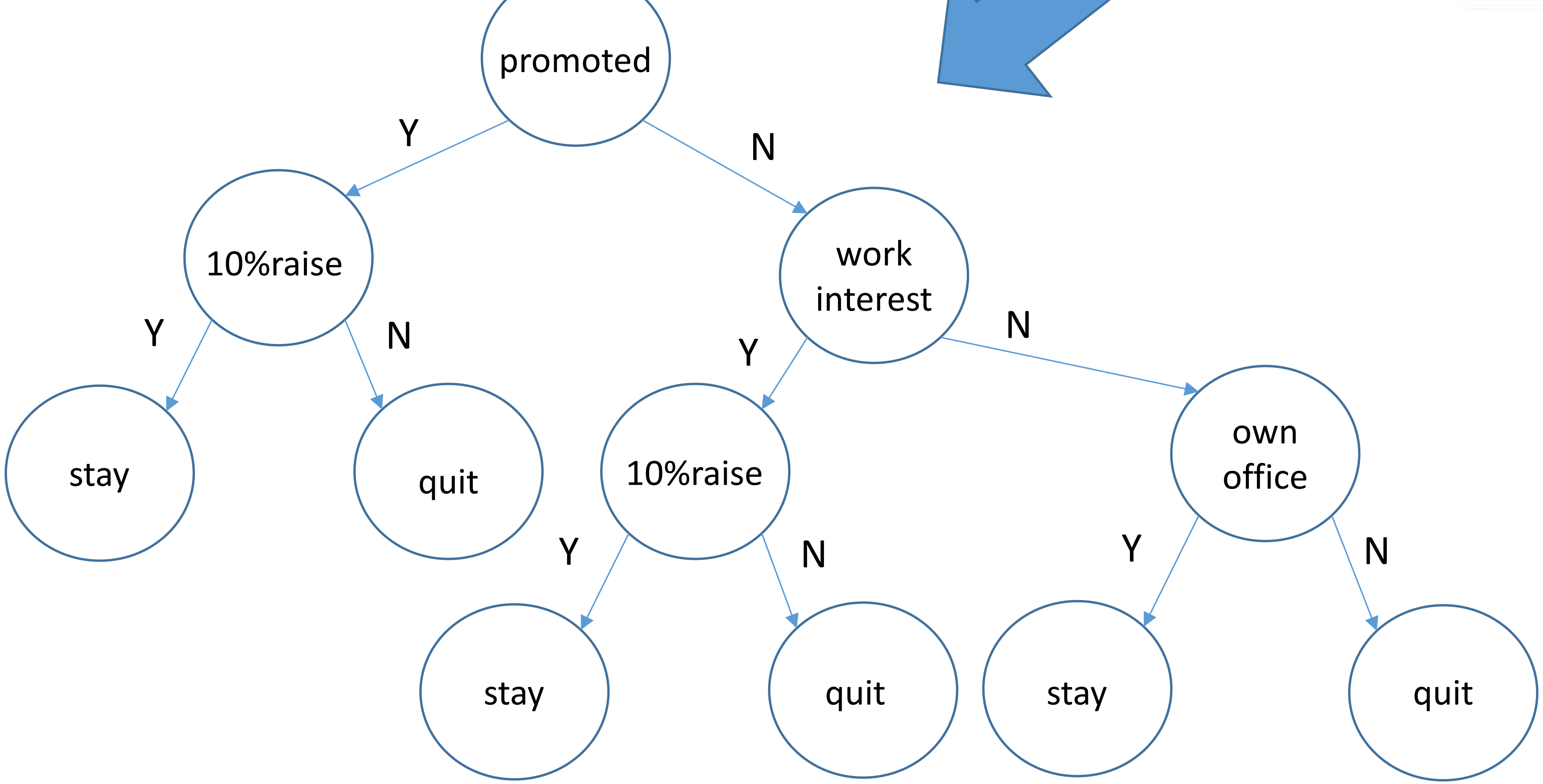


在處理一些複雜的問題上
我們希望可以透過分析器把問題丟進去
答案會自然產生

然而當問題變數改變時
不必修改原本寫好的分析器
就會使用到Interpreter

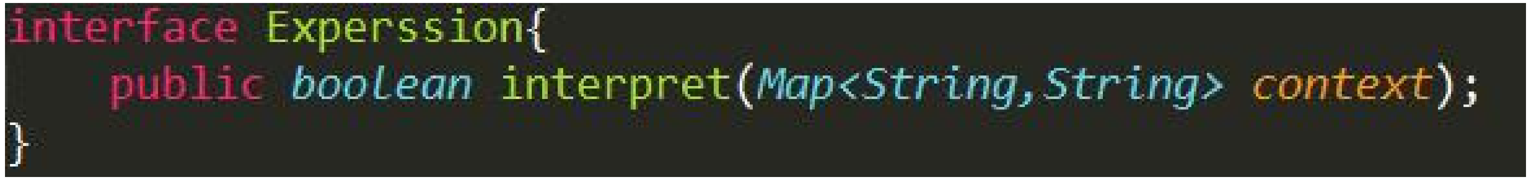
我們試著為這個決策原則建立一棵樹
並使用Interpreter求出答案





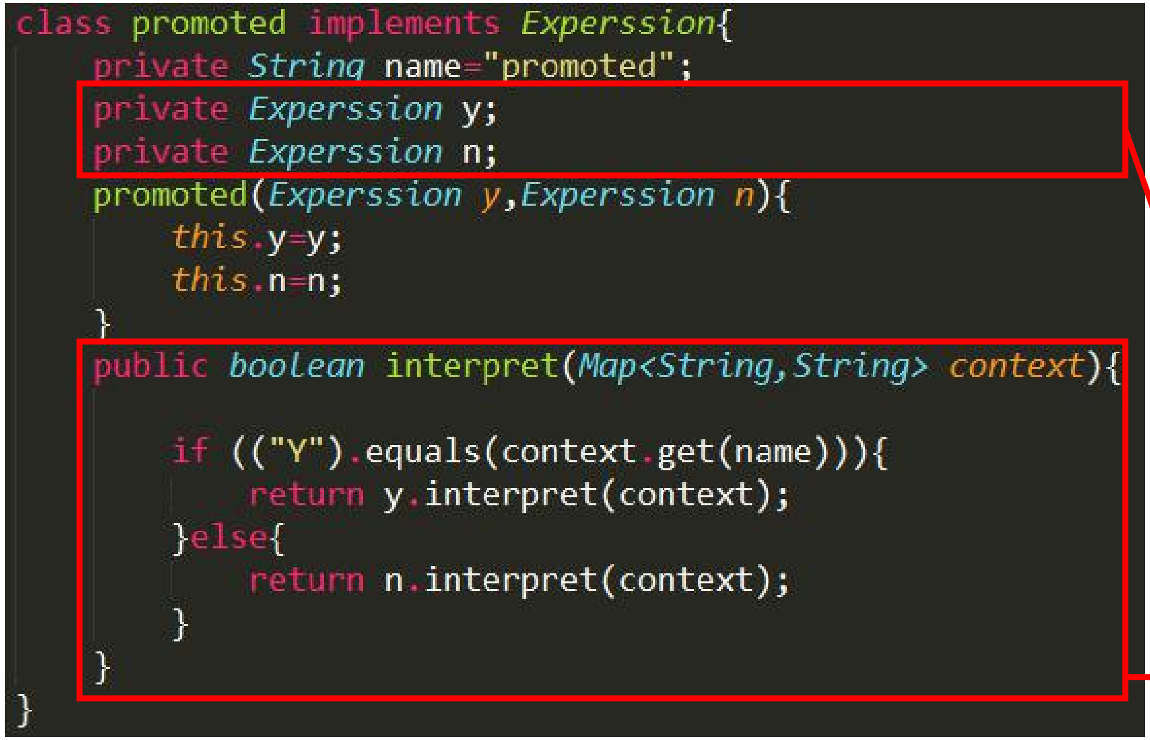
轉換成程式怎麼寫？

1.看到的任何樹節點都是一個Experssion

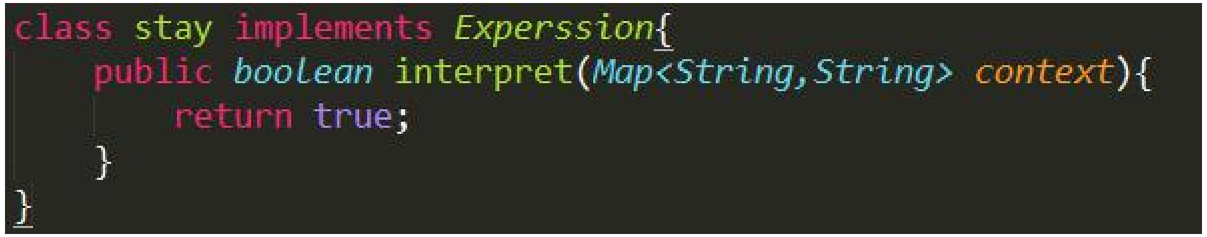


```
interface Experssion{
    public boolean interpret(Map<String,String> context);
}
```

2.總共有六個情況 ，
四個nonTerminal(promoted、raise、workinterest、ownoffice)，
兩個Terminal(Stay、Quit)



```
class promoted implements Experssion{
    private String name="promoted";
    private Experssion y;
    private Experssion n;
    promoted(Experssion y,Experssion n){
        this.y=y;
        this.n=n;
    }
    public boolean interpret(Map<String,String> context){

        if (("Y").equals(context.get(name))){
            return y.interpret(context);
        }else{
            return n.interpret(context);
        }
    }
}
```



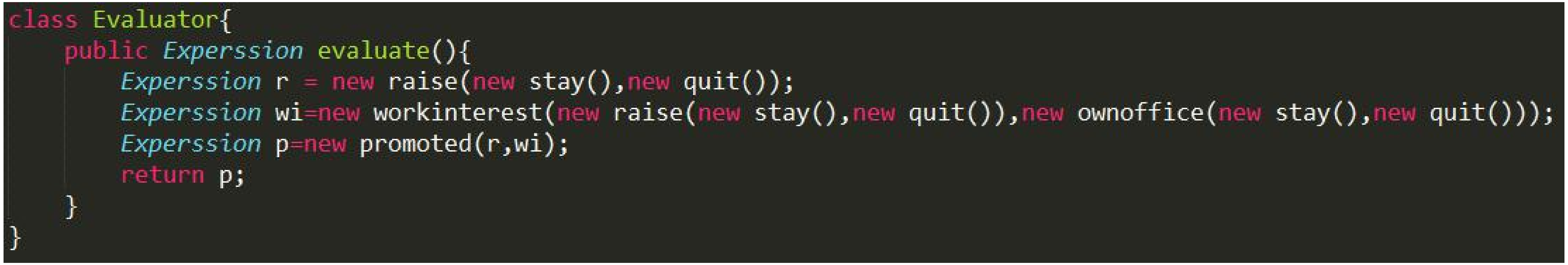
```
class stay implements Experssion{
    public boolean interpret(Map<String,String> context){
        return true;
    }
}
```

Terminal節點，輸出結果。

nonTerminal會有左右節點
(依情況而定，有時候可能是單邊節點)

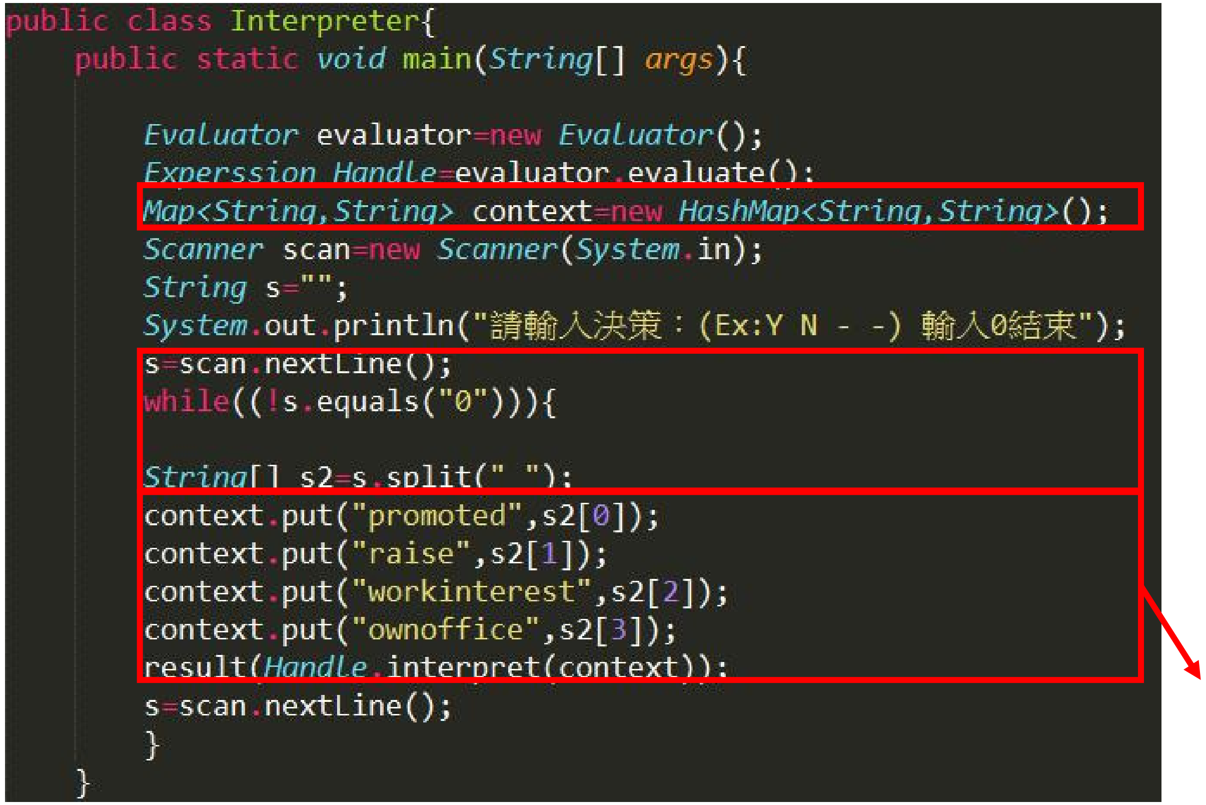
解析的方法，根據此節點的結果分別繼續往左右節點走，直到Terminal節點得出結果。

3.Interpreter最重要的地方，建立結構(樹)

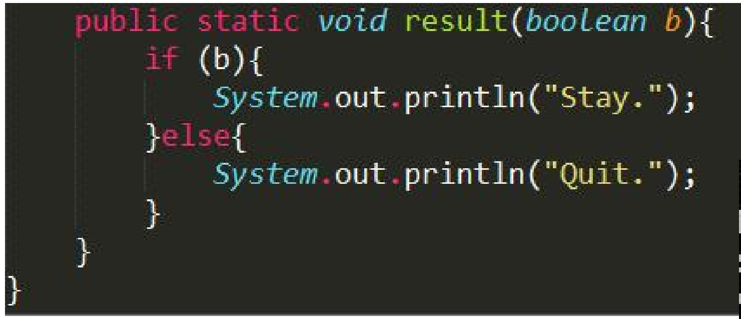


```
class Evaluator{
    public Experssion evaluate(){
        Experssion r = new raise(new stay(),new quit());
        Experssion wi=new workinterest(new raise(new stay(),new quit()),new ownoffice(new stay(),new quit()));
        Experssion p=new promoted(r,wi);
        return p;
    }
}
```

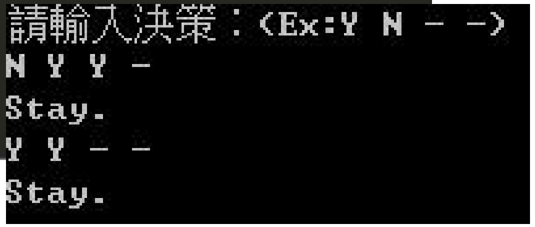
4.輸入與執行



```
public class Interpreter{
    public static void main(String[] args){

        Evaluator evaluator=new Evaluator();
        Experssion Handle=evaluator.evaluate();
        Map<String,String> context=new HashMap<String,String>();
        Scanner scan=new Scanner(System.in);
        String s="";
        System.out.println("請輸入決策：(Ex:Y N - -) 輸入0結束");
        s=scan.nextLine();
        while((!s.equals("0"))){

        String[] s2=s.split(" ");
        context.put("promoted",s2[0]);
        context.put("raise",s2[1]);
        context.put("workinterest",s2[2]);
        context.put("ownoffice",s2[3]);
        result(Handle.interpret(context));
        s=scan.nextLine();
        }
    }
```



```
    public static void result(boolean b){
        if (b){
            System.out.println("Stay.");
        }else{
            System.out.println("Quit.");
        }
    }
}
```



```
請輸入決策：(Ex:Y N - -)
N Y Y -
Stay.
Y Y - -
Stay.
```

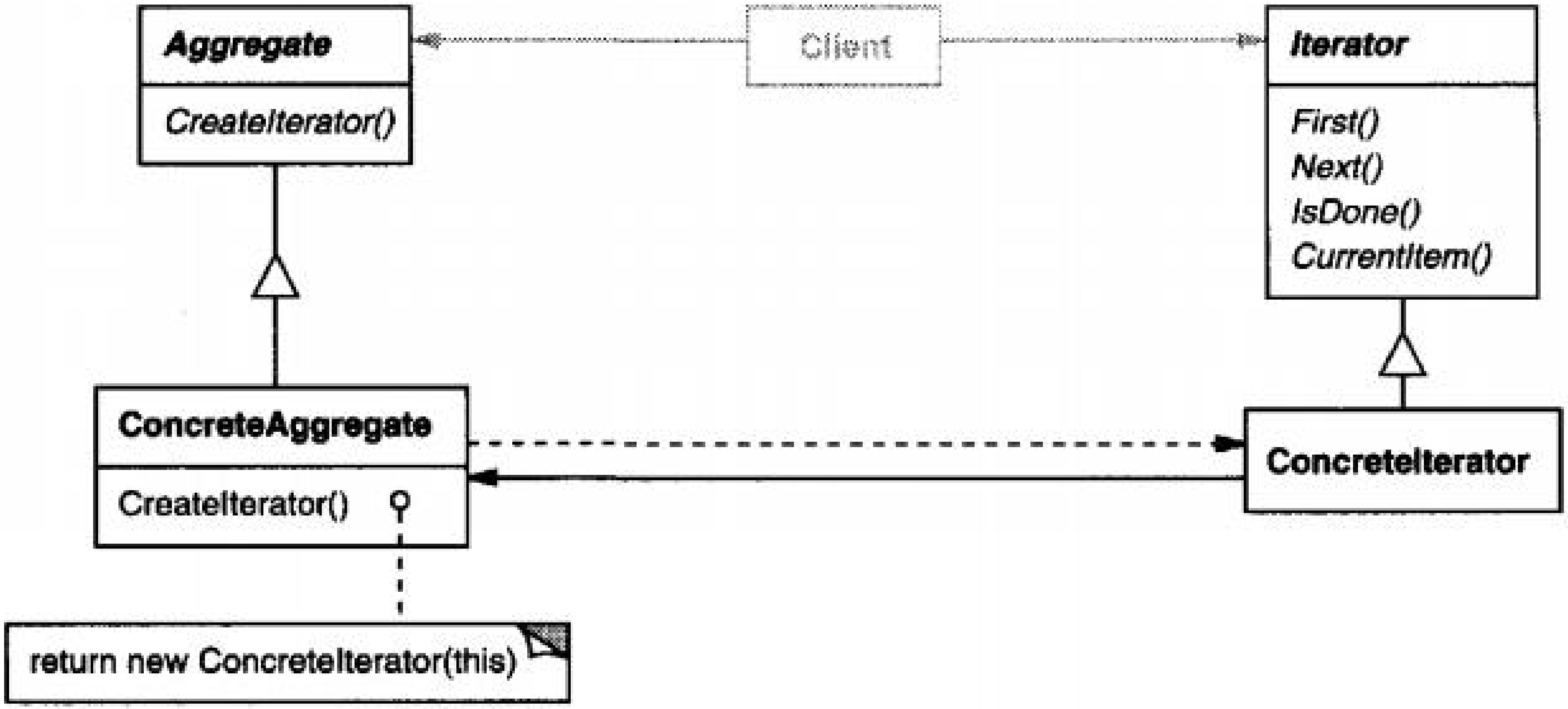
Map分別用來存四個NonTerminal的情境抉擇狀況
輸入並用空格分割
將輸入分別填入Map中，並根據result結果輸出答案

16. Iterator 走訪器模式

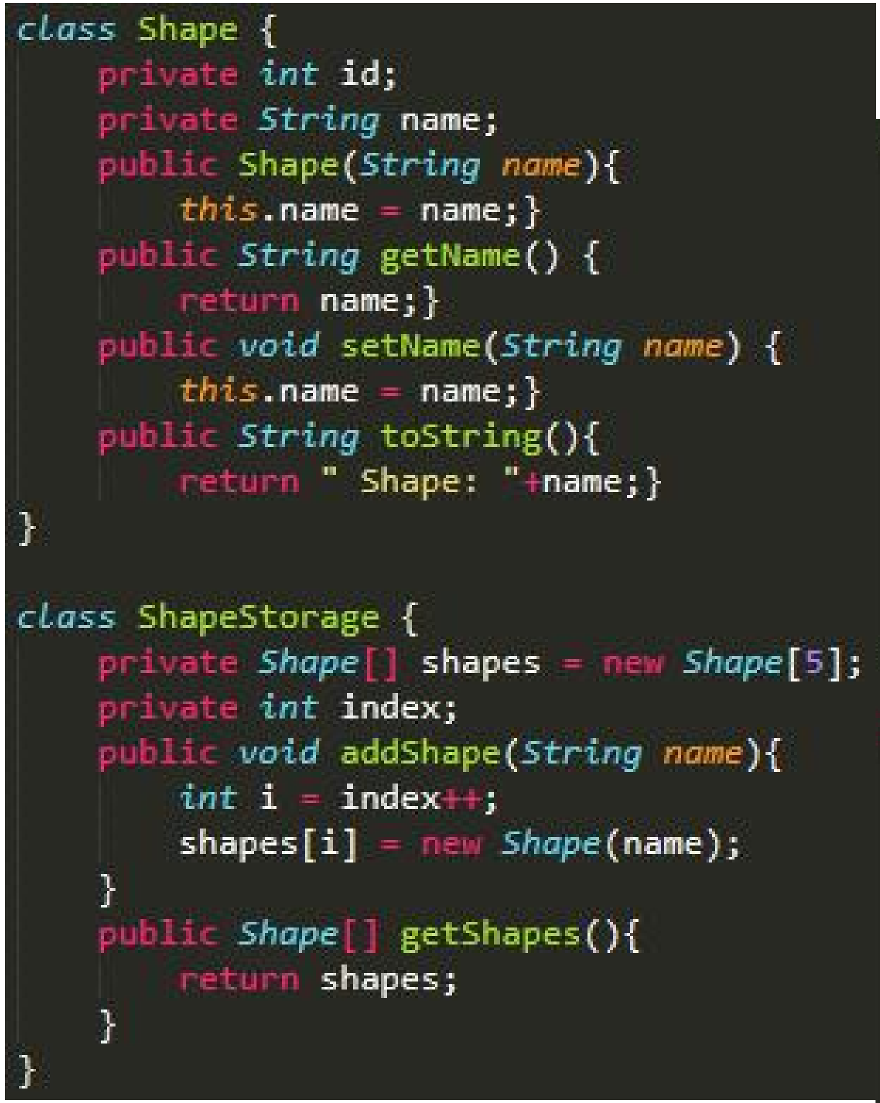
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供一種順序訪問聚合對像元素的方法揭露其潛在的代表性。

提供方法走訪集合內的物件
走訪過程不需知道集合內部的結構

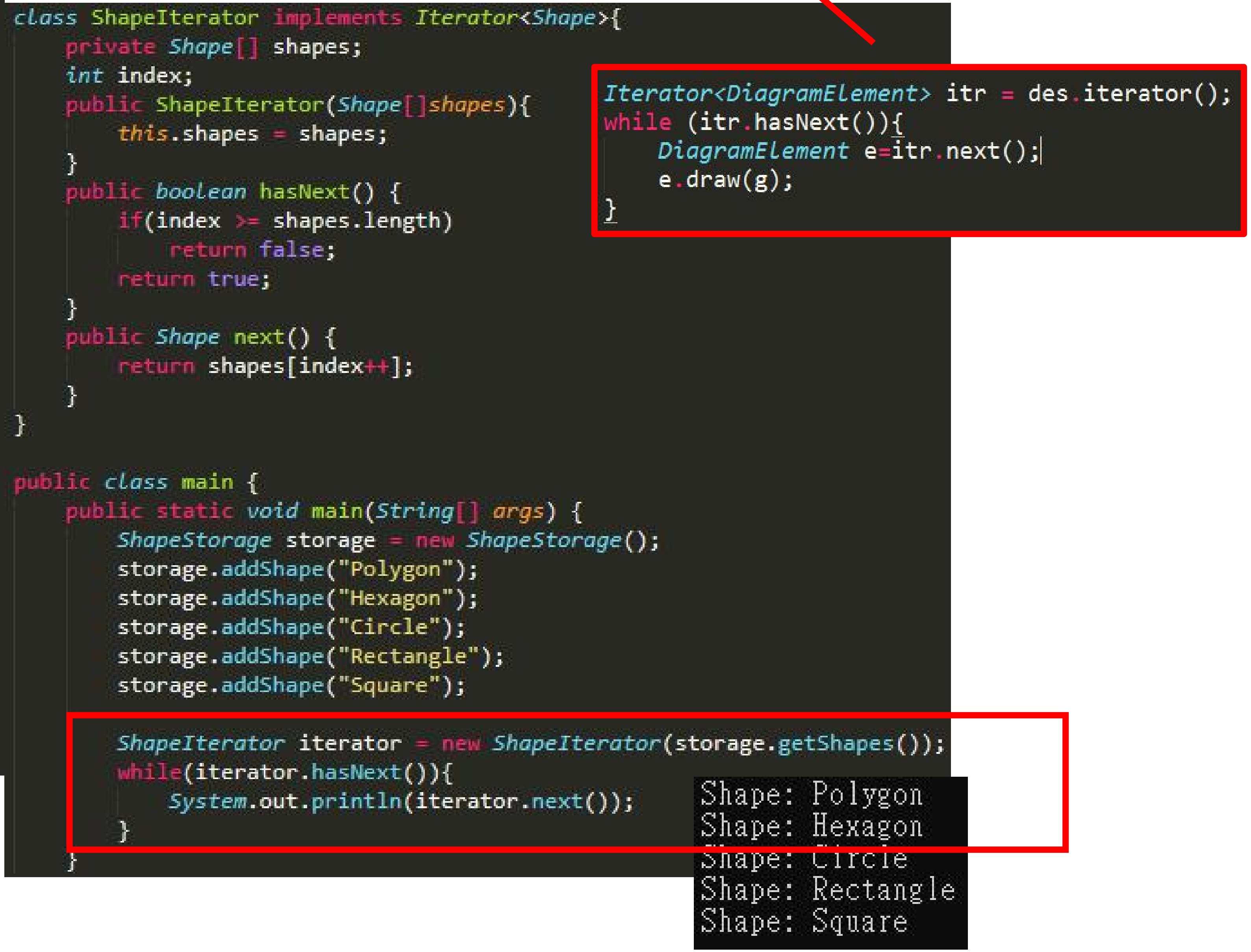

Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

import java.util.Iterator;



```
class Shape {
    private int id;
    private String name;
    public Shape(String name){
        this.name = name;}
    public String getName() {
        return name;}
    public void setName(String name) {
        this.name = name;}
    public String toString(){
        return " Shape: "+name;}
}

class ShapeStorage {
    private Shape[] shapes = new Shape[5];
    private int index;
    public void addShape(String name){
        int i = index++;
        shapes[i] = new Shape(name);
    }
    public Shape[] getShapes(){
        return shapes;
    }
}
```

Java的Collection物件都有內建iterator()方法
直接拿來用吧..



```
class ShapeIterator implements Iterator<Shape>{
    private Shape[] shapes;
    int index;
    public ShapeIterator(Shape[]shapes){
        this.shapes = shapes;
    }
    public boolean hasNext() {
        if(index >= shapes.length)
            return false;
        return true;
    }
    public Shape next() {
        return shapes[index++];
    }
}

public class main {
    public static void main(String[] args) {
        ShapeStorage storage = new ShapeStorage();
        storage.addShape("Polygon");
        storage.addShape("Hexagon");
        storage.addShape("Circle");
        storage.addShape("Rectangle");
        storage.addShape("Square");

        ShapeIterator iterator = new ShapeIterator(storage.getShapes());
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
```

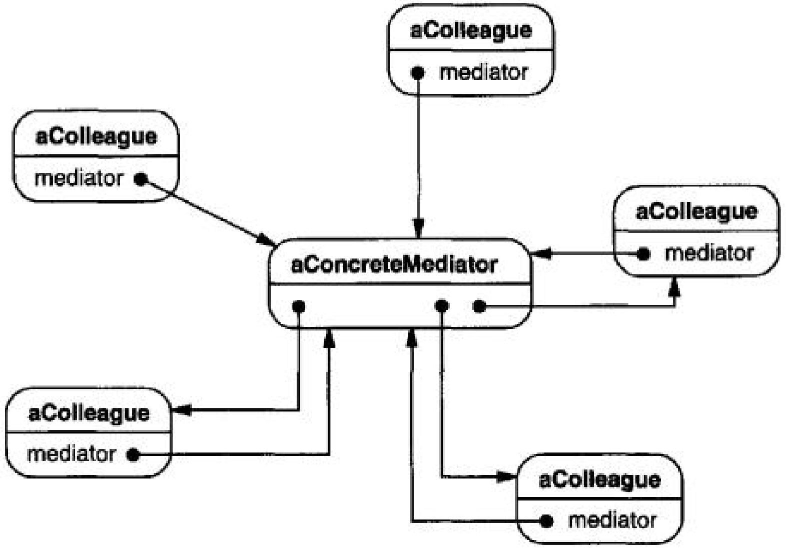
```
Iterator<DiagramElement> itr = des.iterator();
while (itr.hasNext()){
    DiagramElement e=itr.next();
    e.draw(g);
}
```

```
Shape: Polygon
Shape: Hexagon
Shape: Circle
Shape: Rectangle
Shape: Square
```

17. Mediator 中介者模式

Define an object that encapsulate show a set of objects interact.
Mediator promotes loose coupling by keeping objects from referring to each other explicitly,
and it lets you vary their interaction independently.

定義一個封裝顯示一組對象交互的對象。
Mediator通過保持對像明確地相互引用來促進鬆散耦合，它可以讓你獨立改變他們的互動。



把原本彼此錯亂的溝通透過一個中介者來管理
把直接的溝通交給中介者來轉達
Mediator和Colleague之間會互相知道

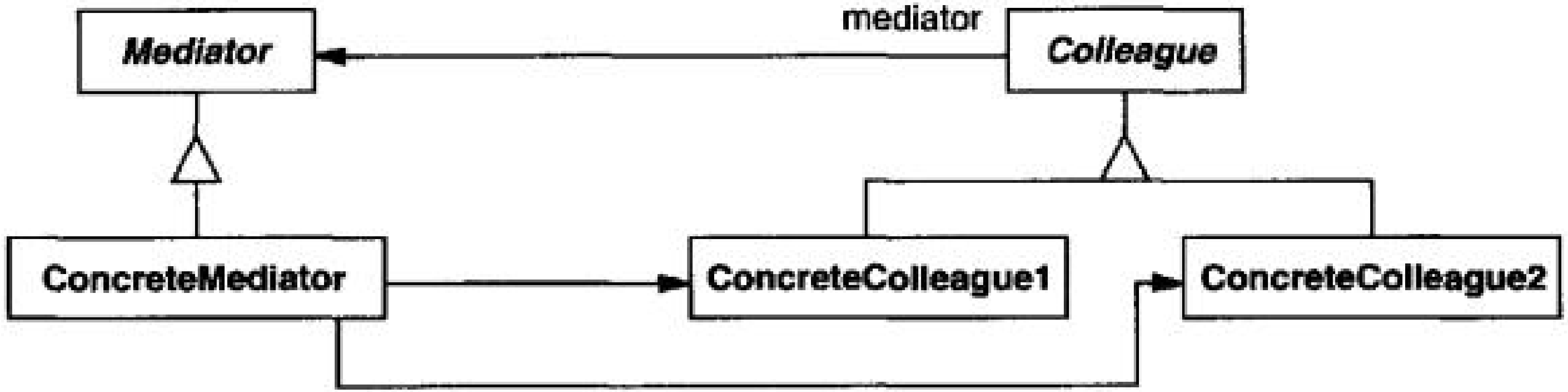
【Façade vs Mediator】

Façade 希望提供一個介面對外提供操作

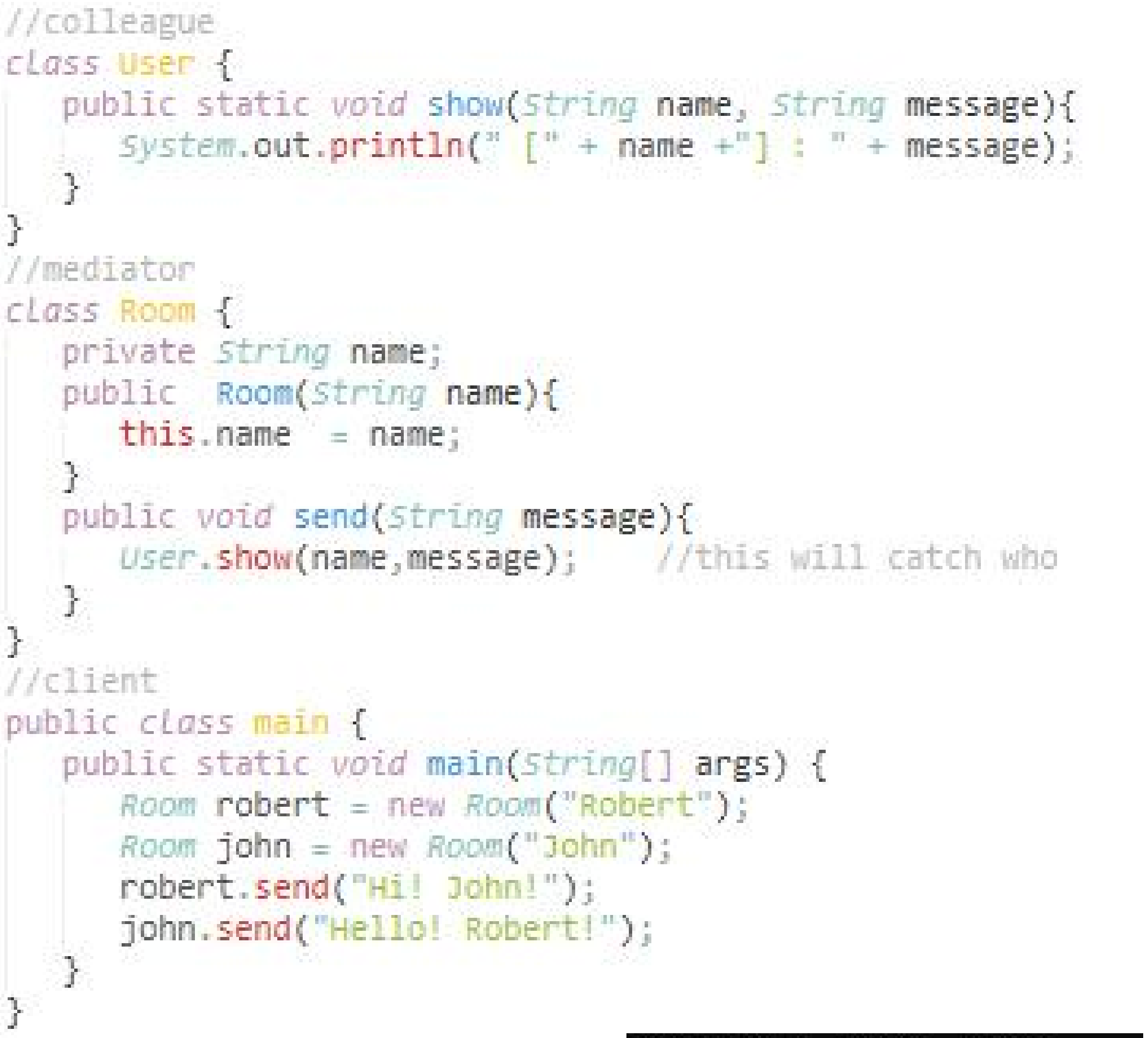
使用者不用知道任何底下的子系統運作(我按開機不管裡面)

Mediator / 當系統內部之間運作非常複雜

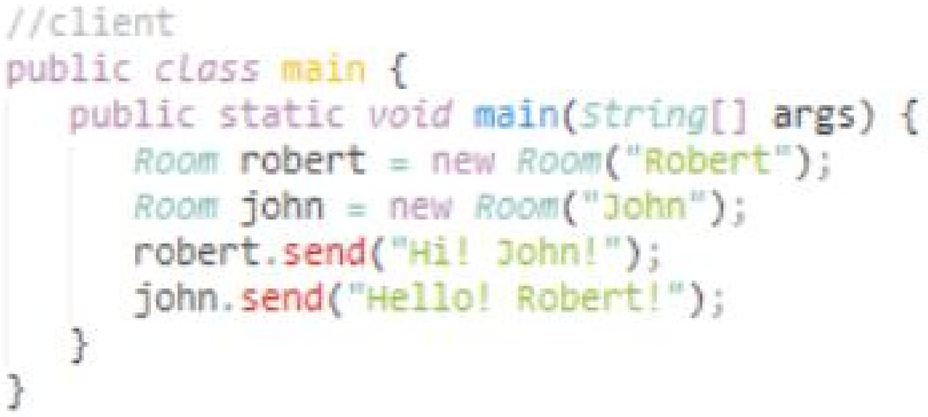
彼此的耦合太高的時候需要一個中介者來調節系統之間的運作


mediator
Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

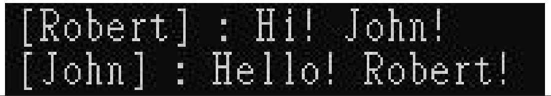
範例1.聊天室



```java
//colleague
class User {
    public static void show(String name, String message){
        System.out.println(" [" + name +"] : " + message);
    }
}
//mediator
class Room {
    private String name;
    public  Room(String name){
        this.name  = name;
    }
    public void send(String message){
        User.show(name,message);    //this will catch who
    }
}
//client
public class main {
    public static void main(String[] args) {
        Room robert = new Room("Robert");
        Room john = new Room("John");
        robert.send("Hi! John!");
        john.send("Hello! Robert!");
    }
}
```



```java
//client
public class main {
    public static void main(String[] args) {
        Room robert = new Room("Robert");
        Room john = new Room("John");
        robert.send("Hi! John!");
        john.send("Hello! Robert!");
    }
}
```



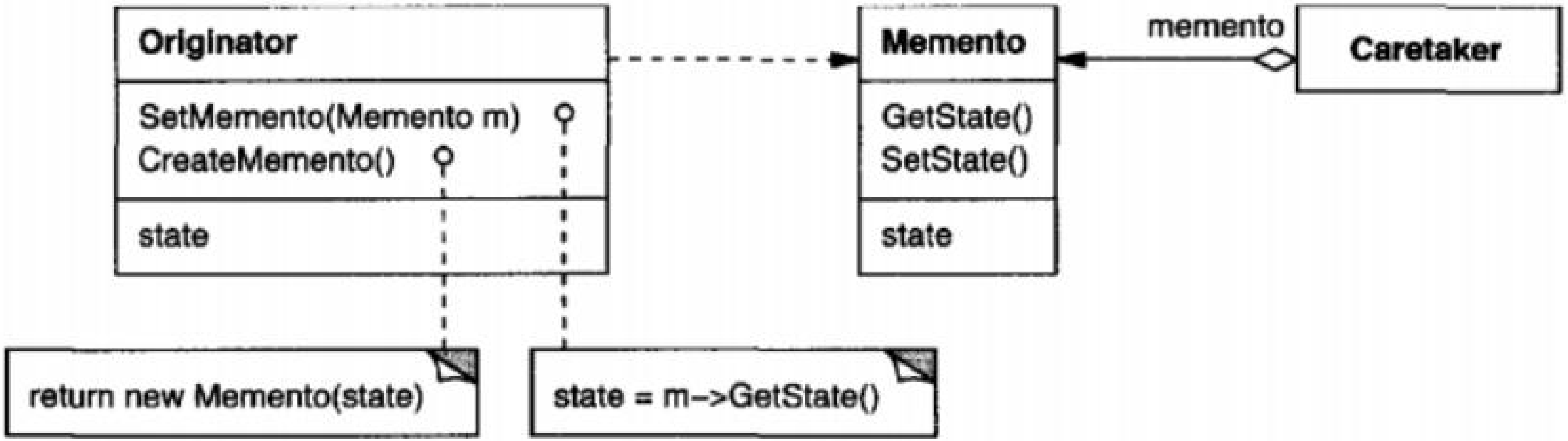
```
[Robert] : Hi! John!
[John] : Hello! Robert!
```

18. Memento 備忘錄模式

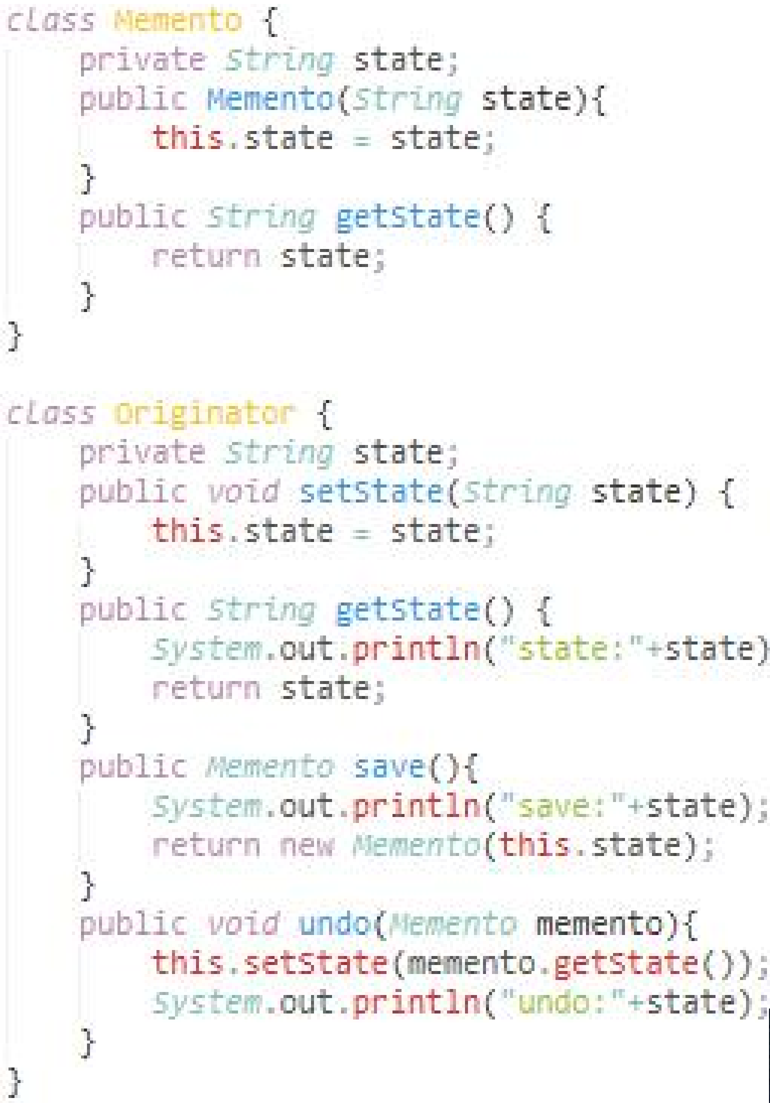
Without violating encapsulation, capture and externalize an object's internal state
so that the object can be restored to this state later.

在不違反封裝的情況下，捕獲並外化對象的內部狀態
以便稍後可以將對象恢復到此狀態。

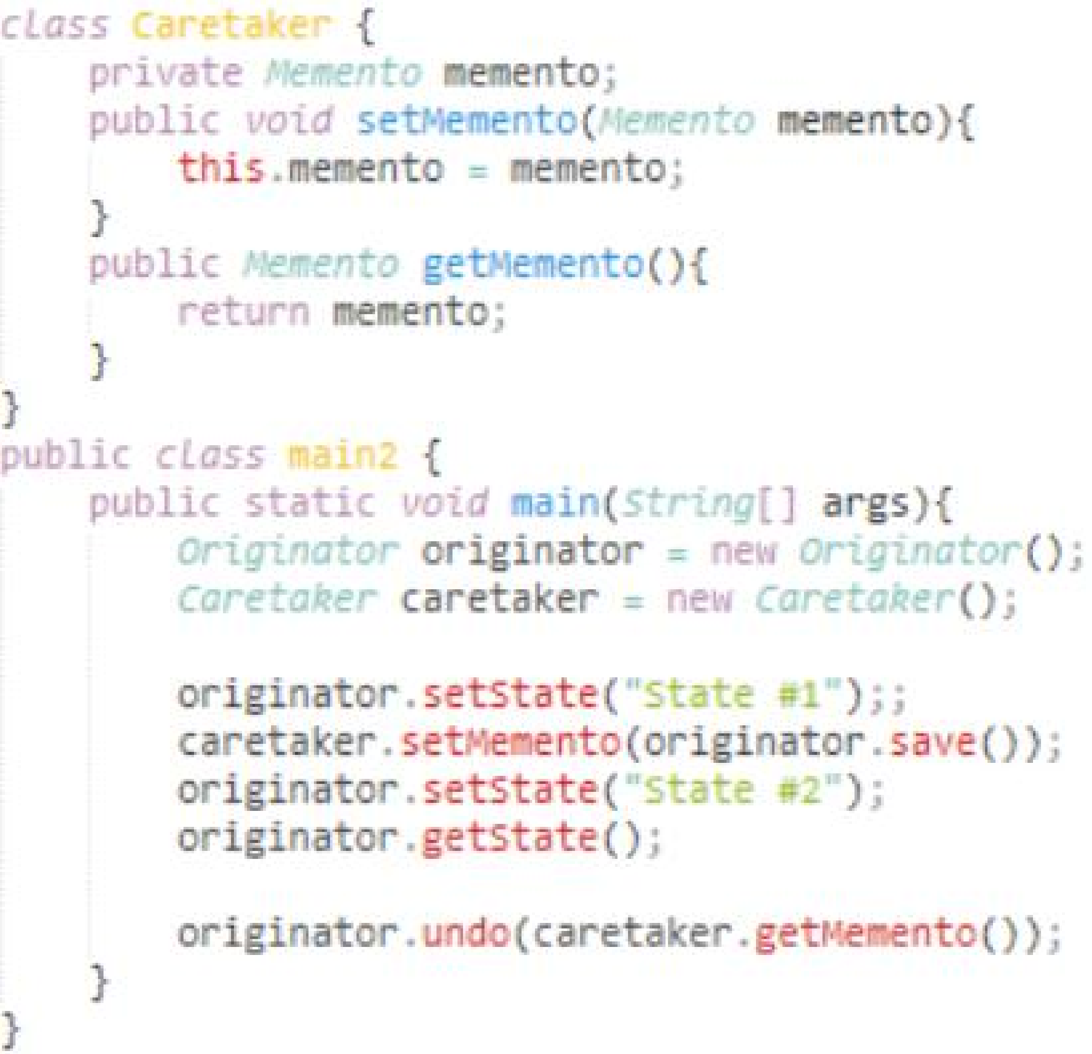
1. **Memento** 備份**state (**便條紙**)**
2. **Originator** 建造**memento (**紙上的內容**)**
3. **Caretaker** 儲存這些備份 **(**便條紙一疊**)**


Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

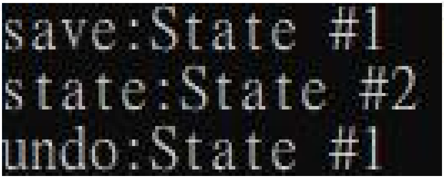
範例1. 基本架構



```java
class Memento {
    private String state;
    public Memento(String state){
        this.state = state;
    }
    public String getState() {
        return state;
    }
}

class Originator {
    private String state;
    public void setState(String state) {
        this.state = state;
    }
    public String getState() {
        System.out.println("state:"+state);
        return state;
    }
    public Memento save(){
        System.out.println("save:"+state);
        return new Memento(this.state);
    }
    public void undo(Memento memento){
        this.setState(memento.getState());
        System.out.println("undo:"+state);
    }
}
```

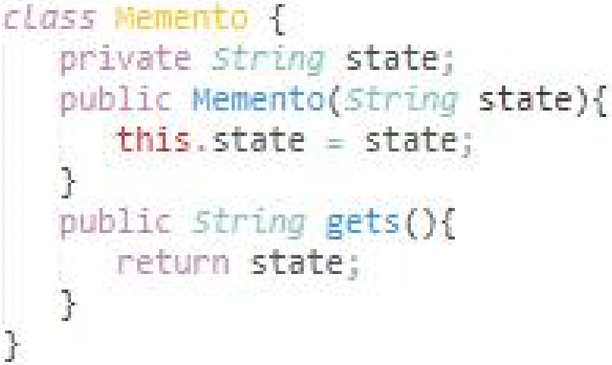


```java
class Caretaker {
    private Memento memento;
    public void setMemento(Memento memento){
        this.memento = memento;
    }
    public Memento getMemento(){
        return memento;
    }
}
public class main2 {
    public static void main(String[] args){
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        originator.setState("State #1");;
        caretaker.setMemento(originator.save());
        originator.setState("State #2");
        originator.getState();

        originator.undo(caretaker.getMemento());
    }
}
```

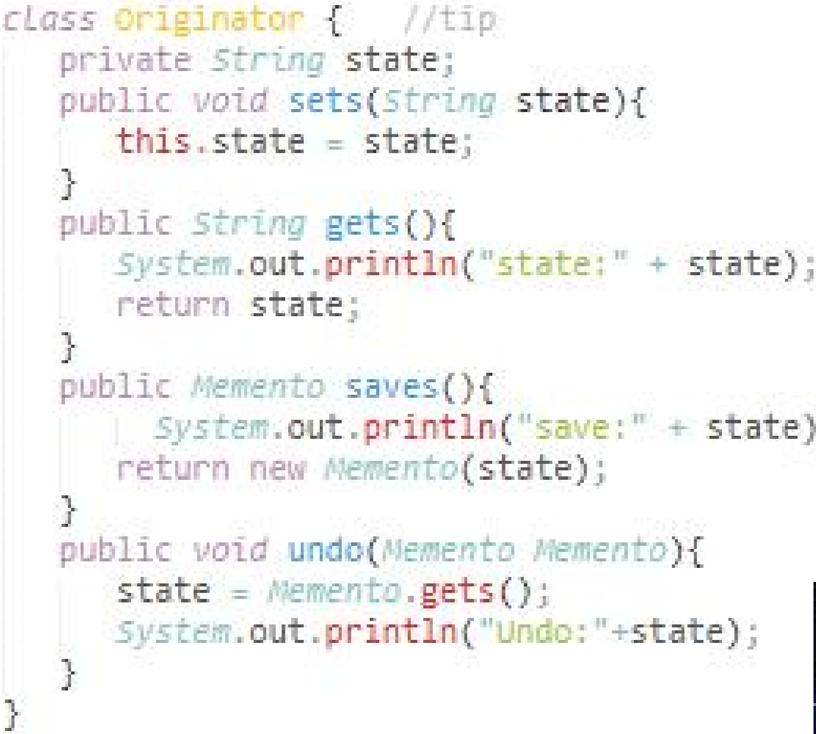


```
save:State #1
state:State #2
undo:State #1
```

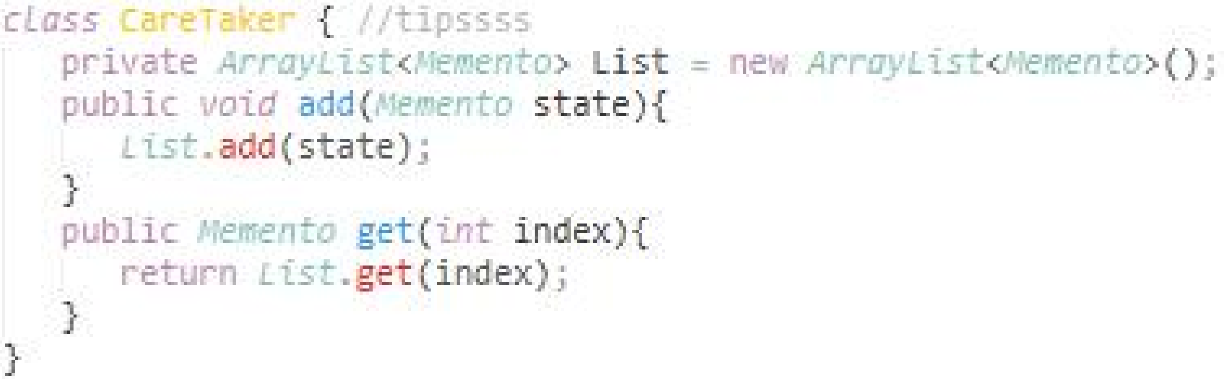
範例2. 搭配ArrayList的使用



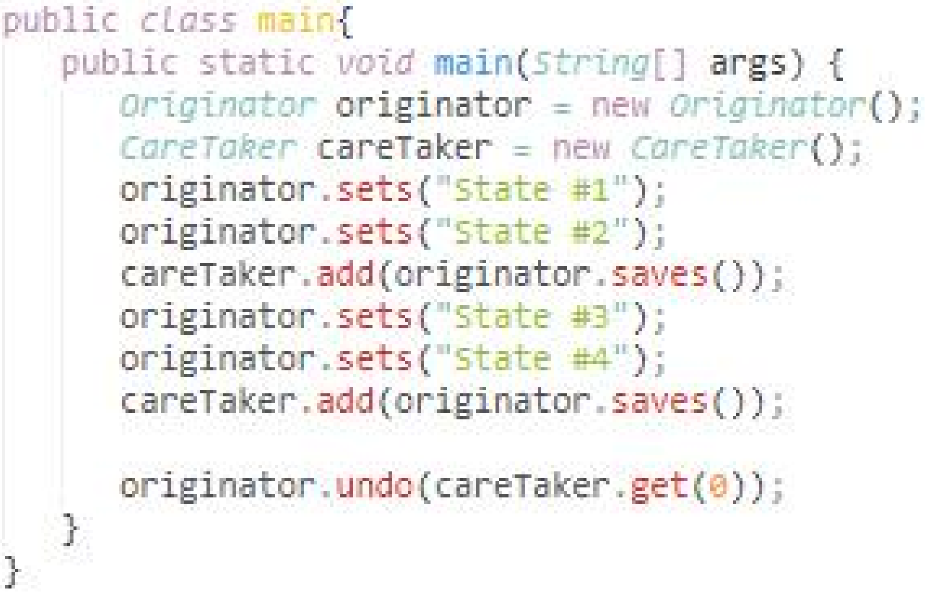
```
class Memento {
    private String state;
    public Memento(String state){
        this.state = state;
    }
    public String gets(){
        return state;
    }
}
```



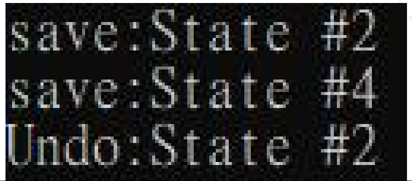
```
class Originator {   //tip
    private String state;
    public void sets(String state){
        this.state = state;
    }
    public String gets(){
        System.out.println("state:" + state);
        return state;
    }
    public Memento saves(){
        System.out.println("save:" + state);
        return new Memento(state);
    }
    public void undo(Memento Memento){
        state = Memento.gets();
        System.out.println("Undo:"+state);
    }
}
```



```
class CareTaker { //tipssss
    private ArrayList<Memento> List = new ArrayList<Memento>();
    public void add(Memento state){
        List.add(state);
    }
    public Memento get(int index){
        return List.get(index);
    }
}
```



```
public class main{
    public static void main(String[] args) {
        Originator originator = new Originator();
        CareTaker careTaker = new CareTaker();
        originator.sets("State #1");
        originator.sets("State #2");
        careTaker.add(originator.saves());
        originator.sets("State #3");
        originator.sets("State #4");
        careTaker.add(originator.saves());

        originator.undo(careTaker.get(0));
    }
}
```



```
save:State #2
save:State #4
Undo:State #2
```

19. Observer 觀察者模式

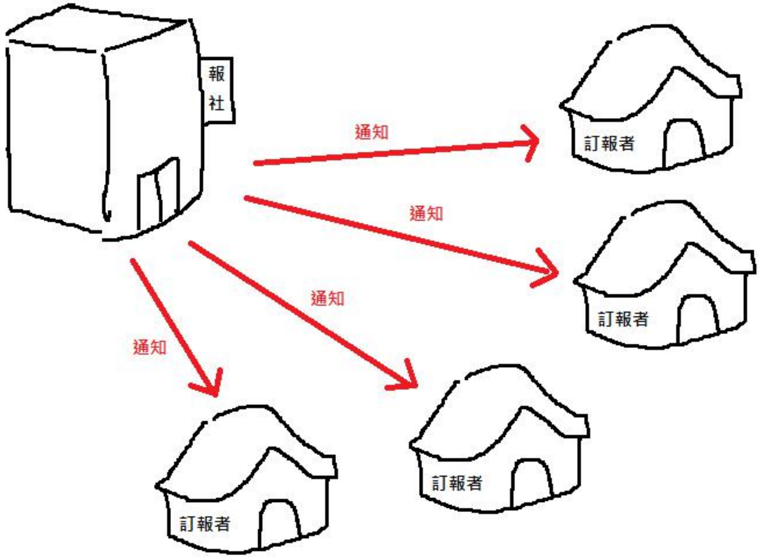
Define a one-to-many dependency between objects so that when one object changes state,
all its dependents are notified and updated automatically.

定義對象之間的一對多依賴關係，以便一個對象改變狀態時，
所有家屬都會自動得到通知和更新。

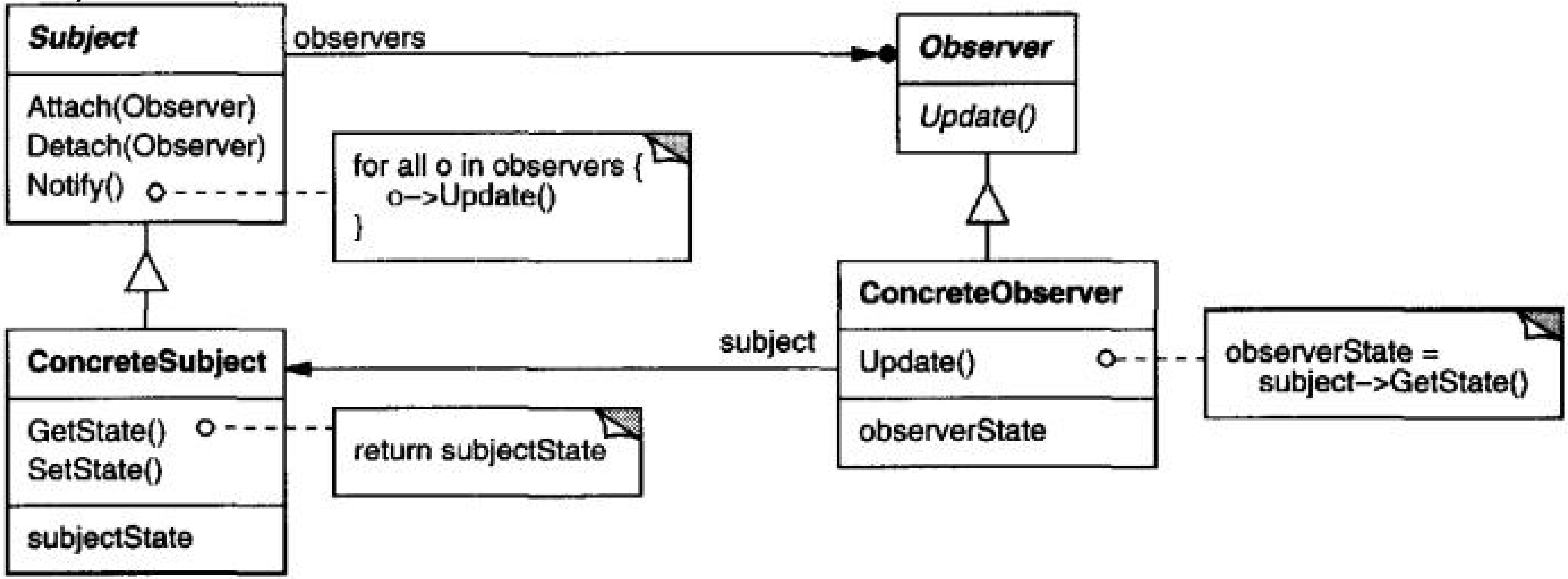
Observer很簡單
就是當Subject改變時
要通知所有的Observer要更新了

常跟代理模式對比：
Observer和Proxy主要區別在功能不一樣，

Observer強調的是被觀察者反饋結果，
Proxy是同根負責做同樣的事情。



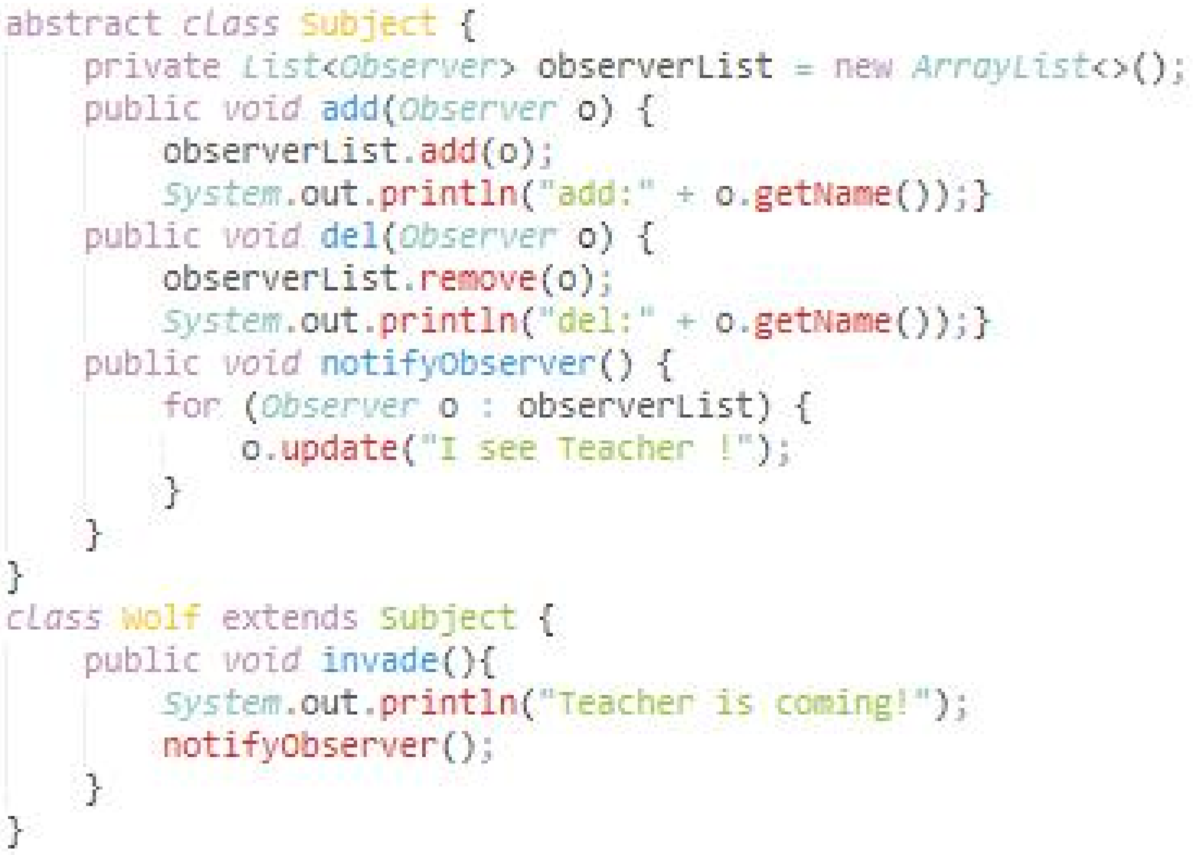
(Client) →



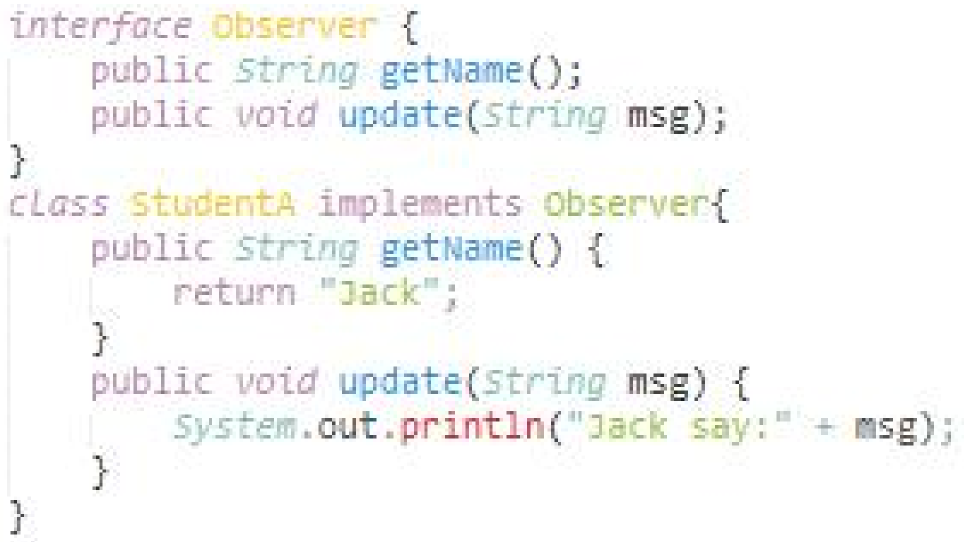
Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

Oberver 必備: import java.util.ArrayList;
import java.util.List;

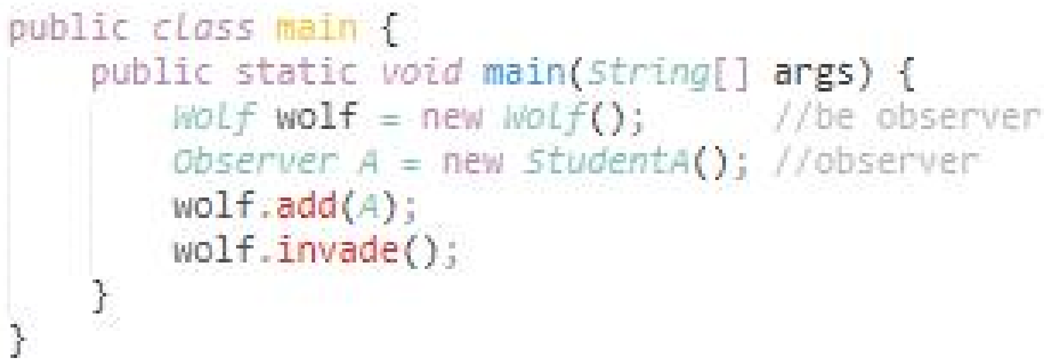
範例1. 高中時期，老師(被觀察者)準備走進教室



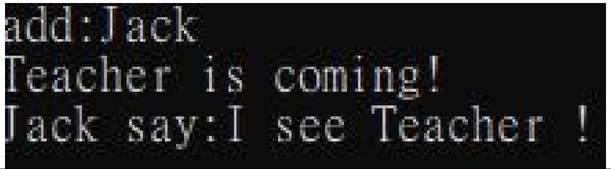
```
abstract class Subject {
    private List<Observer> observerList = new ArrayList<>();
    public void add(Observer o) {
        observerList.add(o);
        System.out.println("add:" + o.getName());}
    public void del(Observer o) {
        observerList.remove(o);
        System.out.println("del:" + o.getName());}
    public void notifyObserver() {
        for (Observer o : observerList) {
            o.update("I see Teacher !");
        }
    }
}
class Wolf extends Subject {
    public void invade(){
        System.out.println("Teacher is coming!");
        notifyObserver();
    }
}
```



```
interface Observer {
    public String getName();
    public void update(String msg);
}
class StudentA implements Observer{
    public String getName() {
        return "Jack";
    }
    public void update(String msg) {
        System.out.println("Jack say:" + msg);
    }
}
```

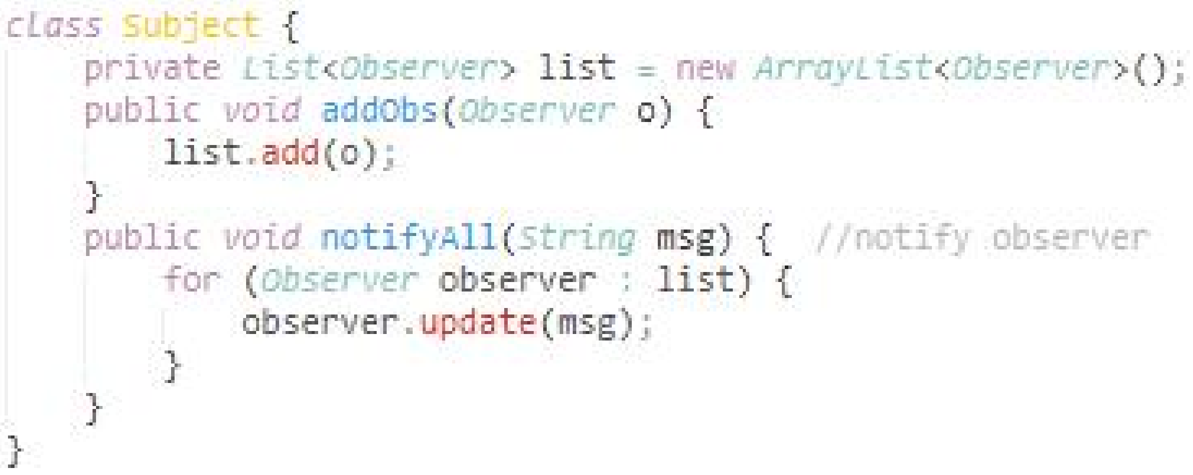


```
public class main {
    public static void main(String[] args) {
        Wolf wolf = new Wolf();      //be observer
        Observer A = new StudentA(); //observer
        wolf.add(A);
        wolf.invade();
    }
}
```

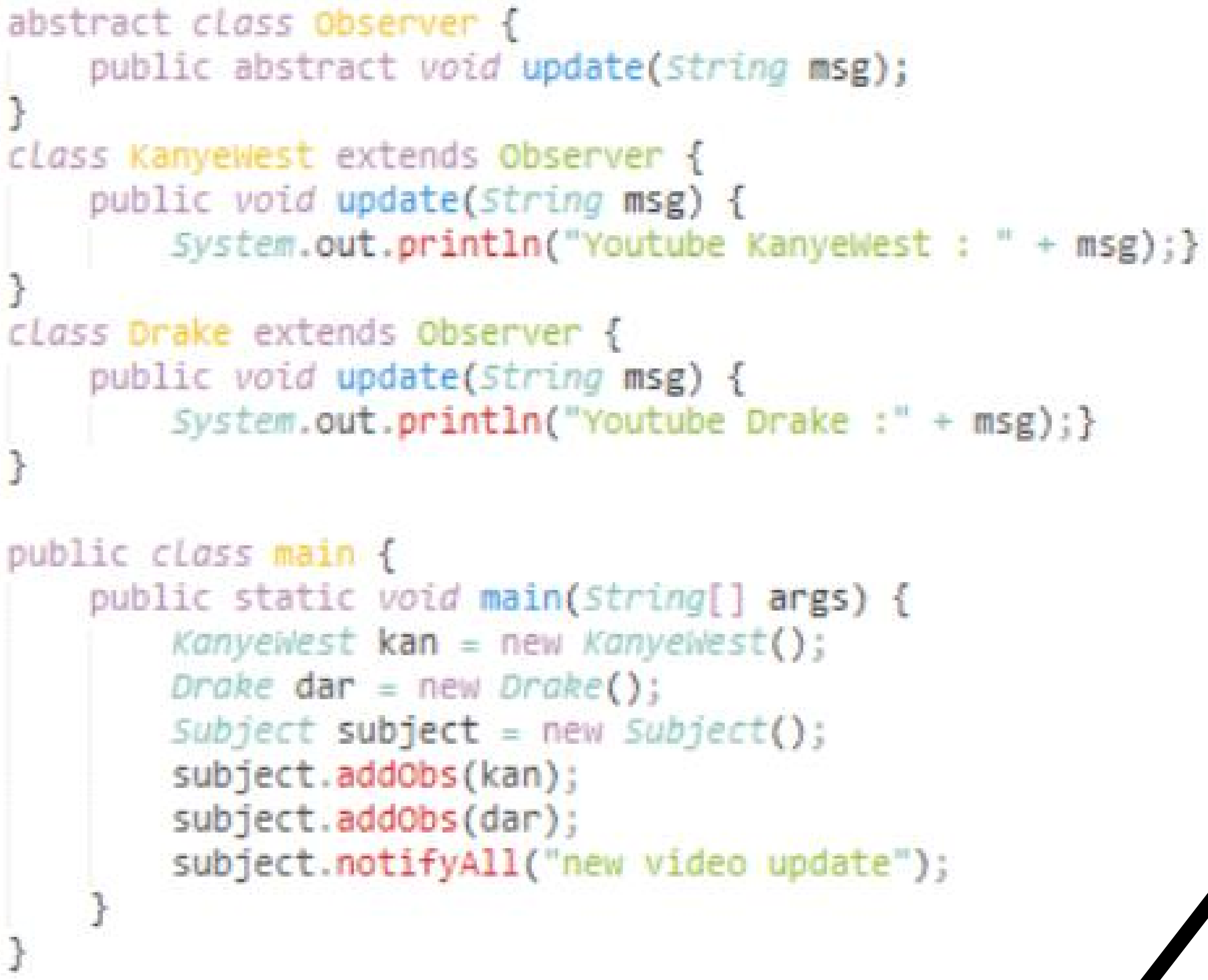


```
add:Jack
Teacher is coming!
Jack say:I see Teacher !
```

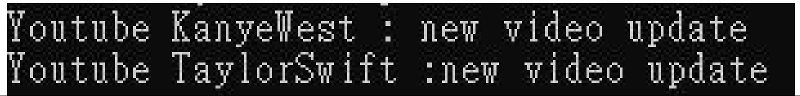
範例2. 我訂閱的Youtuber(observer)發布新影片



```
class Subject {
    private List<Observer> list = new ArrayList<Observer>();
    public void addObs(Observer o) {
        list.add(o);
    }
    public void notifyAll(String msg) {  //notify observer
        for (Observer observer : list) {
            observer.update(msg);
        }
    }
}
```



```
abstract class Observer {
    public abstract void update(String msg);
}
class KanyeWest extends Observer {
    public void update(String msg) {
        System.out.println("Youtube KanyeWest : " + msg);}
}
class Drake extends Observer {
    public void update(String msg) {
        System.out.println("Youtube Drake :" + msg);}
}

public class main {
    public static void main(String[] args) {
        KanyeWest kan = new KanyeWest();
        Drake dar = new Drake();
        Subject subject = new Subject();
        subject.addObs(kan);
        subject.addObs(dar);
        subject.notifyAll("new video update");
    }
}
```



```
Youtube KanyeWest : new video update
Youtube TaylorSwift :new video update
```

20. State 狀態模式

Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class.

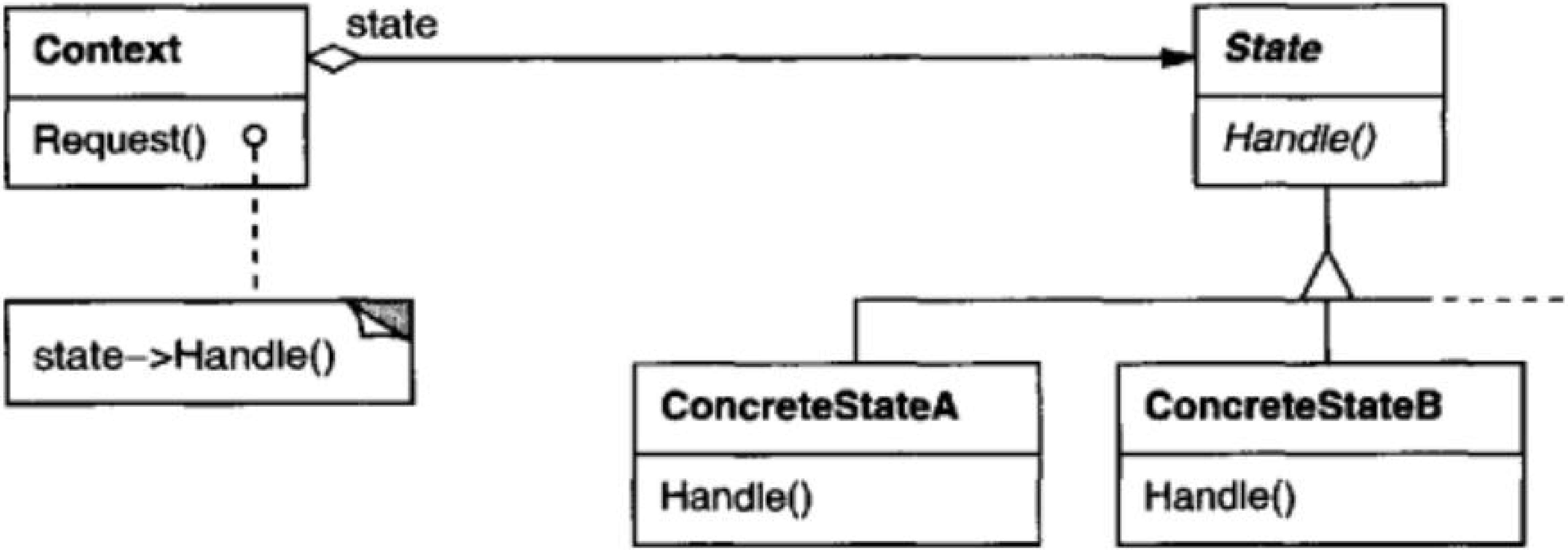
允許對像在其內部狀態更改時更改其行為。該對像似乎會更改其類。

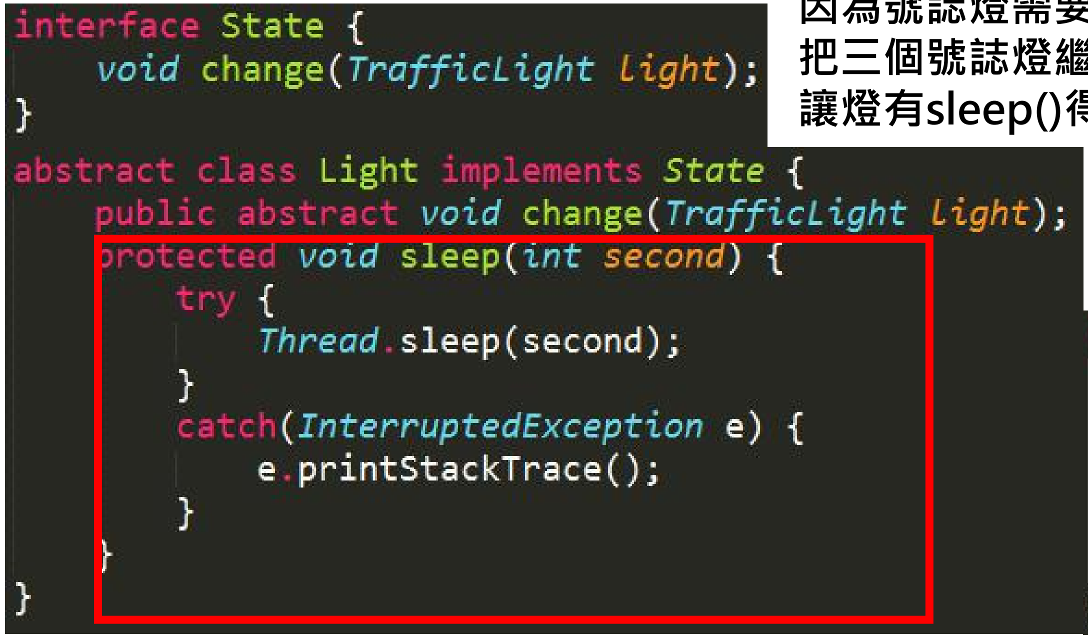
一個物件的行為會因為物件自身狀態不同
而表現出不同的反應動作。

例如一個自動販賣機物件，「選擇貨品」的功能。
會因為顧客有沒有投錢、投多少錢，而有不同反應。

「結構跟Strategy一模一樣 不過目的不一樣」

- State是由自己轉變到下一個State
- Strategy是由使用者決定要切換到哪一個方法

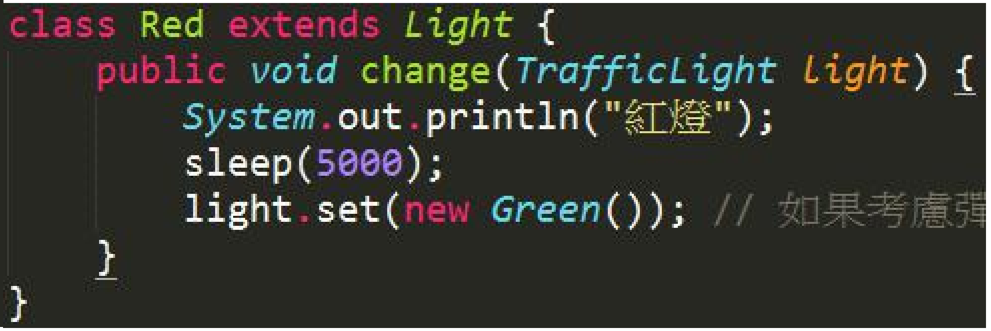

Context
Request()
state
state->Handle()
State
Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()



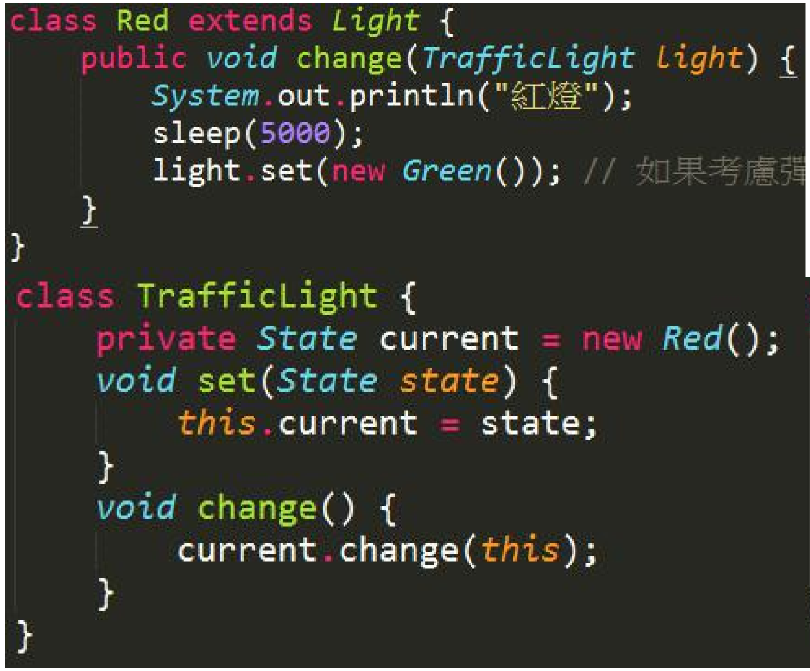
```java
interface State {
    void change(TrafficLight light);
}
abstract class Light implements State {
    public abstract void change(TrafficLight light);
    protected void sleep(int second) {
        try {
            Thread.sleep(second);
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

因為號誌燈需要讀秒倒數所以
把三個號誌燈繼承一個Light
讓燈有sleep()得方法可以倒數

實作State的change方法
每個燈的sleep時間不一樣
Sleep完之後會切換燈號到下一個狀態

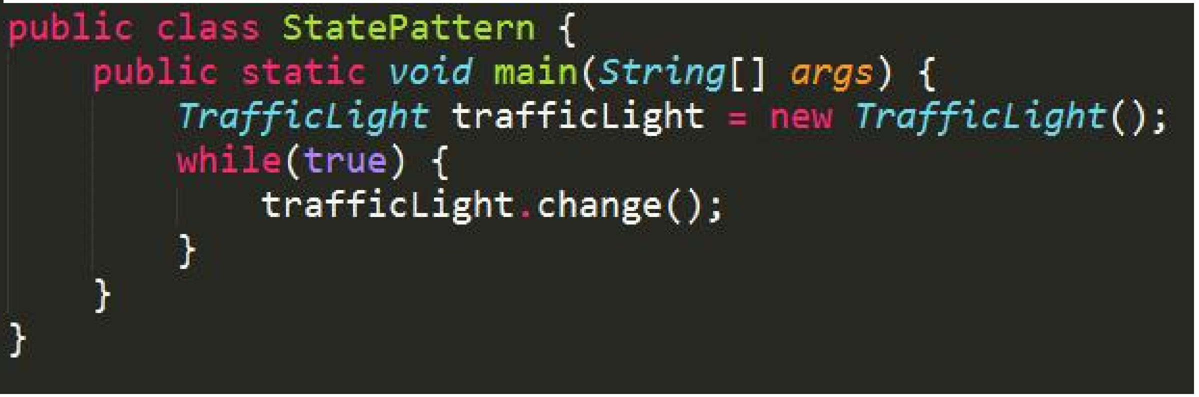


```java
class Red extends Light {
    public void change(TrafficLight light) {
        System.out.println("紅燈");
        sleep(5000);
        light.set(new Green()); // 如果考慮彈
    }
}
```

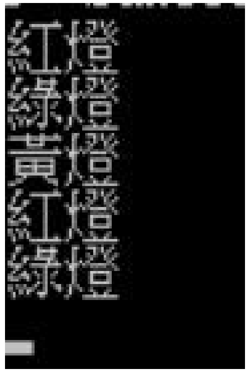


```java
class Red extends Light {
    public void change(TrafficLight light) {
        System.out.println("紅燈");
        sleep(5000);
        light.set(new Green()); // 如果考慮彈
    }
}
class TrafficLight {
    private State current = new Red();
    void set(State state) {
        this.current = state;
    }
    void change() {
        current.change(this);
    }
}
```

2.再來號誌燈會使用State的change方法
當State被改變，change的實際方法也改變
3.在Sleep完之後會重新把號誌燈Set到下一個綠燈狀態
1.迴圈內會一直呼叫號誌燈要Change



```java
public class StatePattern {
    public static void main(String[] args) {
        TrafficLight trafficLight = new TrafficLight();
        while(true) {
            trafficLight.change();
        }
    }
}
```



```
紅燈
綠燈
黃燈
紅燈
綠燈
```

21. Strategy 策略模式

Define a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.

定義一系列算法，封裝每個算法，並使它們可互換。
策略允許算法獨立於使用它的客戶端。

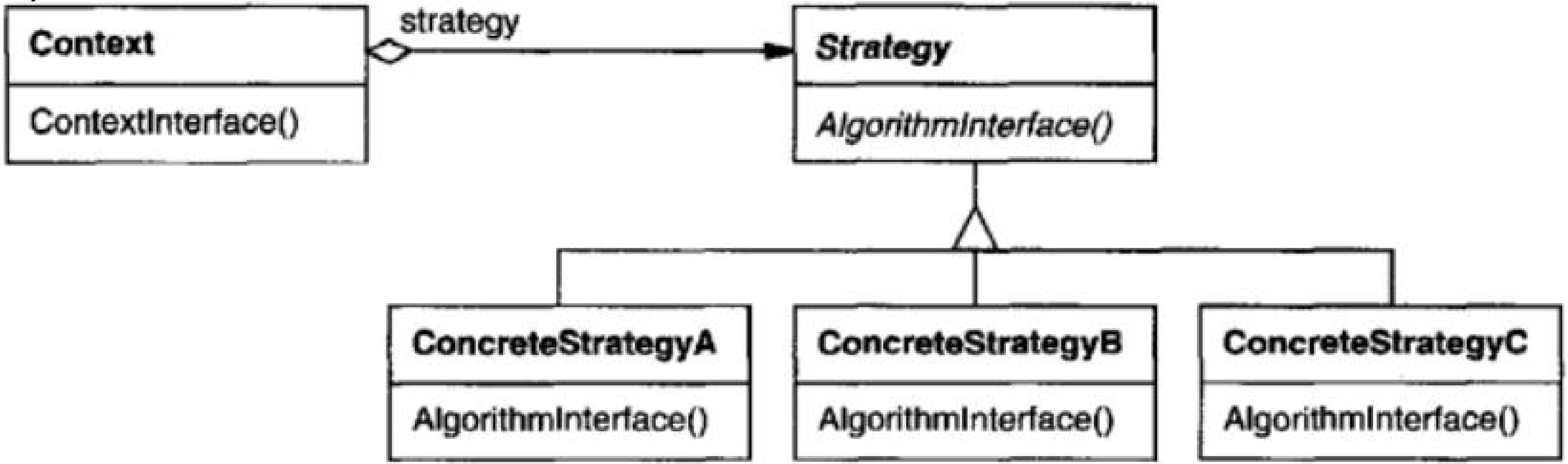
為了達到相同的目的，物件可以因地制宜，
讓行為擁有多種不同的實作方法。
比如每個人都要「交個人所得稅」，但是「在美國交個人所得稅」
和「在中國交個人所得稅」就有不同的算稅方法。

將實作的方法獨立出來成為一個Class，
透過繼承可以在不修改原程式碼的情況下擴充新程式碼

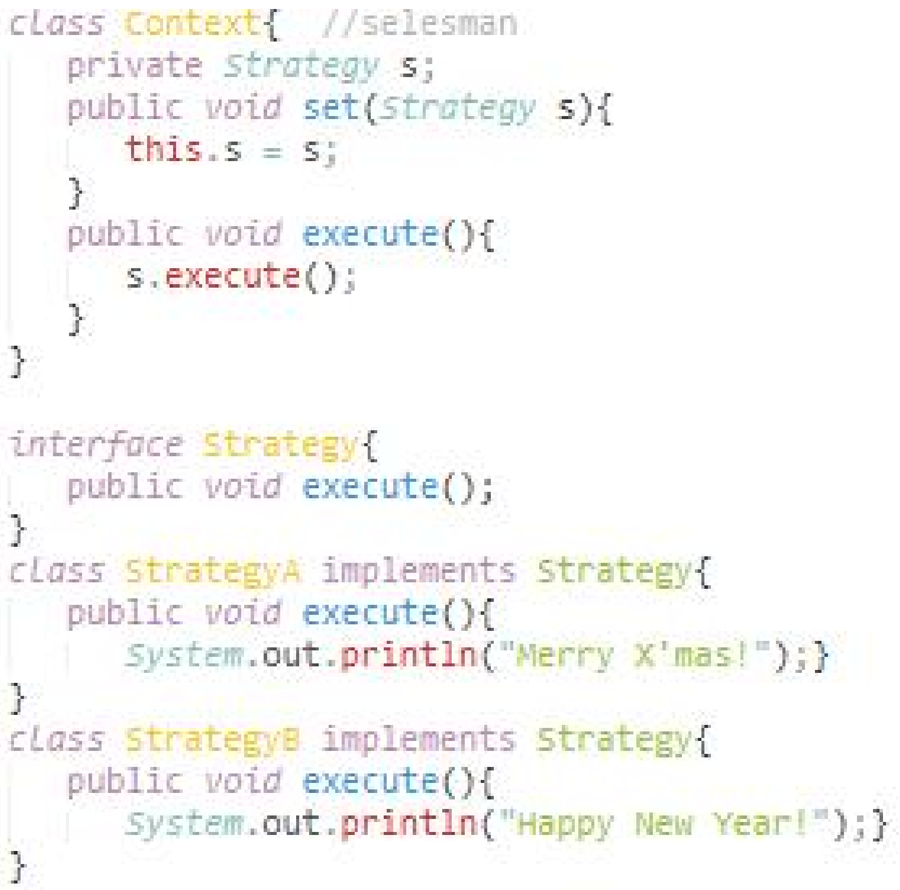
- 吻合OCP原則(Open-Closed Principl)
- Runtime下改變(Override)
- Dynamic Binding

OCP=軟體實體必須能夠延伸但不能修改，”對擴展開放，修改則封閉”

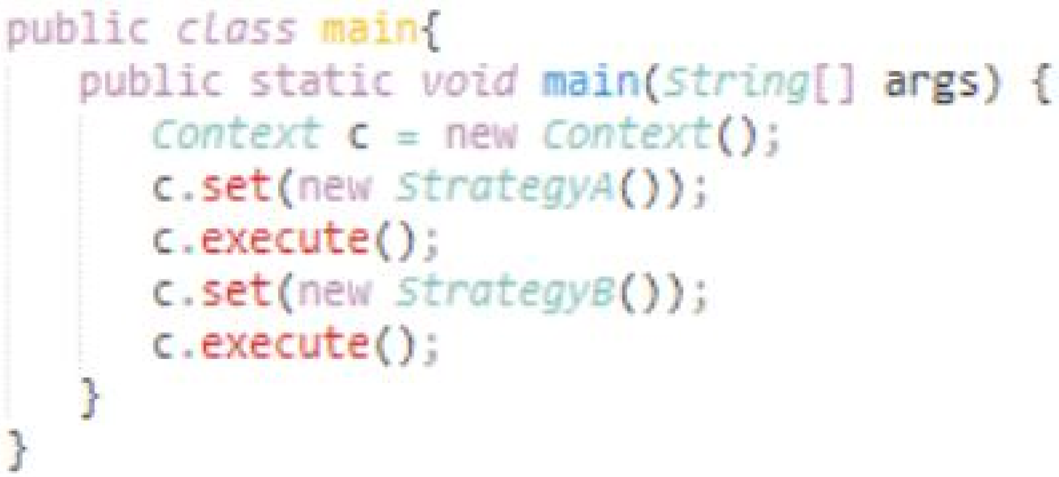
(Client) →


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

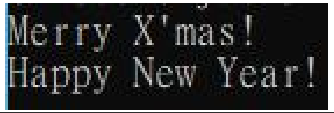
範例1. 節慶到了，要叫業務員跑業務



```
class Context{  //selesman
    private Strategy s;
    public void set(Strategy s){
        this.s = s;
    }
    public void execute(){
        s.execute();
    }
}

interface Strategy{
    public void execute();
}
class StrategyA implements Strategy{
    public void execute(){
        System.out.println("Merry X'mas!");}
}
class StrategyB implements Strategy{
    public void execute(){
        System.out.println("Happy New Year!");}
}
```



```
public class main{
    public static void main(String[] args) {
        Context c = new Context();
        c.set(new StrategyA());
        c.execute();
        c.set(new StrategyB());
        c.execute();
    }
}
```

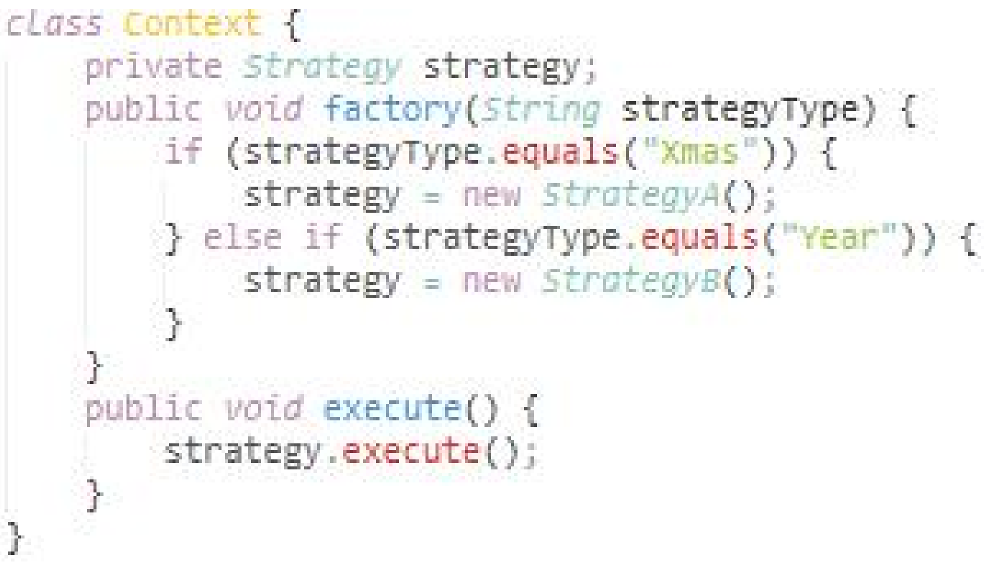


```
Merry X'mas!
Happy New Year!
```

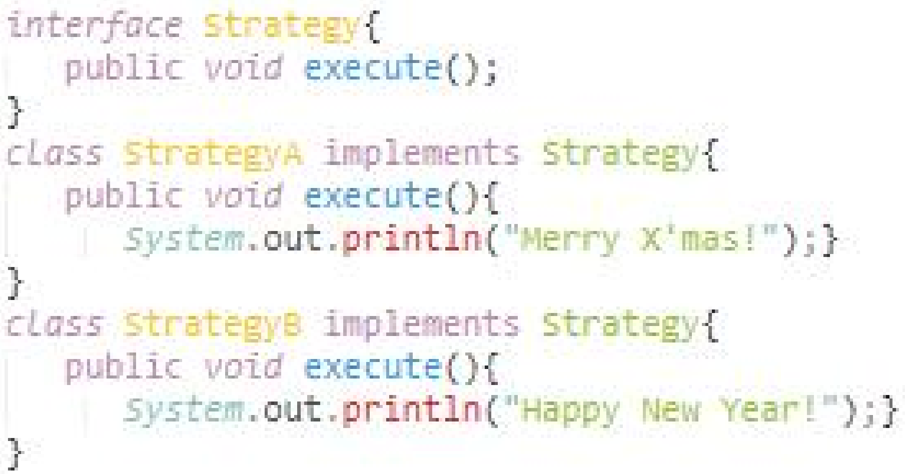
策略模式的缺點，就是所有的策略都必須暴露讓客戶端自行選擇策略使用。

改善這個缺陷需要跟簡單工廠模式結合混編

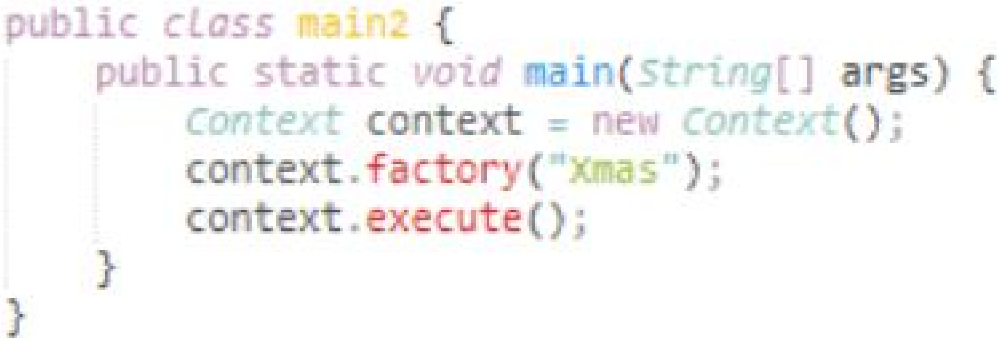
範例2. 節慶到了，要叫業務員跑業務 (Factory + Strategy)



```java
class Context {
    private Strategy strategy;
    public void factory(String strategyType) {
        if (strategyType.equals("Xmas")) {
            strategy = new StrategyA();
        } else if (strategyType.equals("Year")) {
            strategy = new StrategyB();
        }
    }
    public void execute() {
        strategy.execute();
    }
}
```



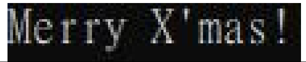
```java
interface Strategy{
   public void execute();
}
class StrategyA implements Strategy{
   public void execute(){
      System.out.println("Merry X'mas!");}
}
class StrategyB implements Strategy{
   public void execute(){
      System.out.println("Happy New Year!");}
}
```



```java
public class main2 {
    public static void main(String[] args) {
        Context context = new Context();
        context.factory("Xmas");
        context.execute();
    }
}
```

策略模式的缺點，就是所有的策略都必須暴露，
讓客戶端自行選擇策略使用。

改善這個缺陷需要跟簡單工廠模式結合混編



```
Merry X'mas!
```

22. Template 樣板模式

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
Template Method lets subclasses redefine certain steps of an algorithm
without changing the algorithm's structure.

在操作中定義算法的骨架，將一些步驟推遲到子類。
模板方法允許子類重新定義算法的某些步驟而不改變算法的結構。

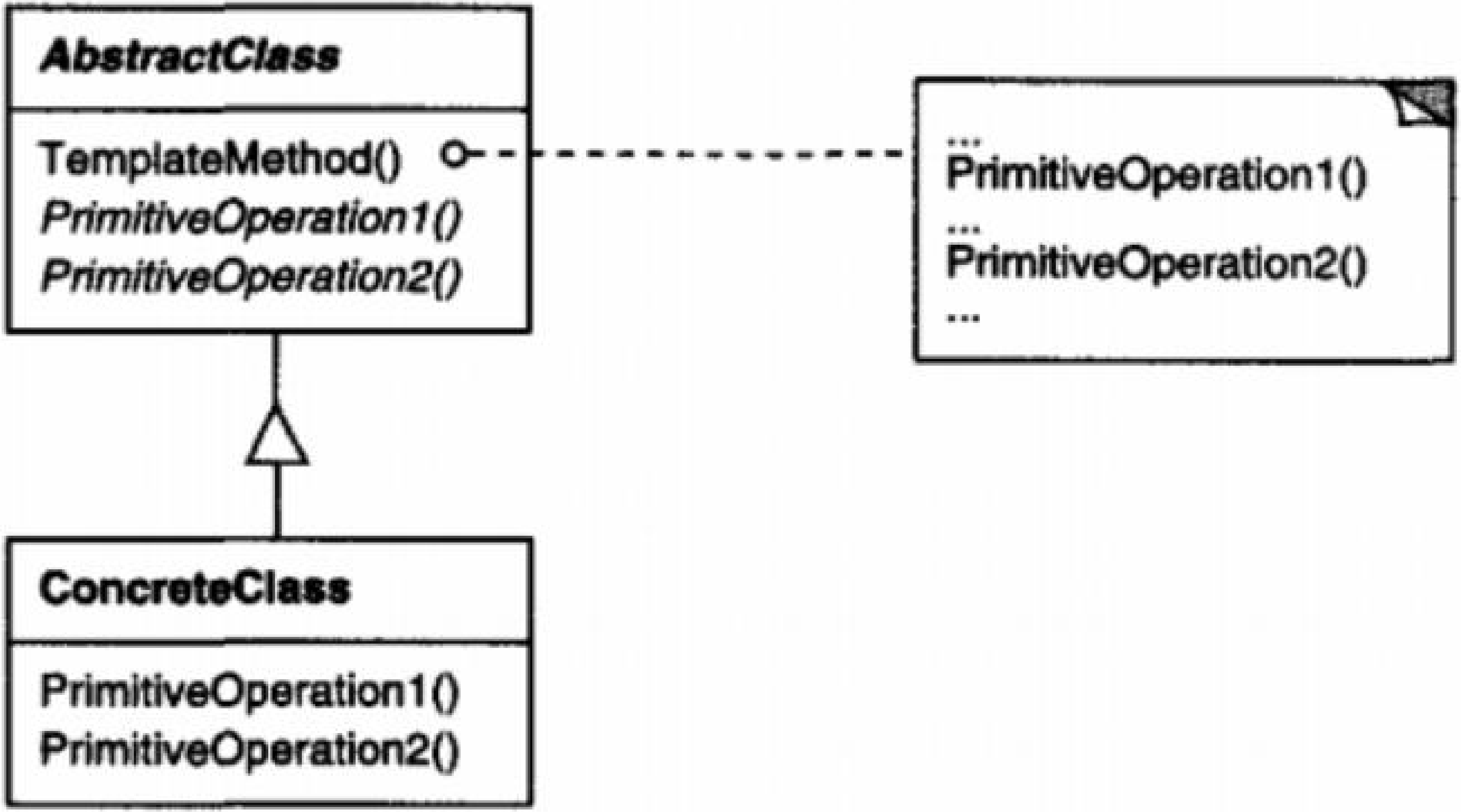
Template 顧名思義就是提供一個固定的樣板
你可以自行修改樣板的方法來達到不一樣的效果

- Template Method /
  必須是Final因為要定義好執行順序
- Primitive Method /
  是Abstract必須被複寫的方法
- Concrete Method /
  偶爾會有如果是通用的方法就可以先實作好
- Hook有兩種說法
  1.(老師)說的他是一個Boolean值，當Template執行
  可以根據這掛勾決定要不要執行這一段
  2.(網路)說Hook是一個預設為空的方法(Concrete)，
  子類別可以選擇是否覆寫這個方法來擴充功能

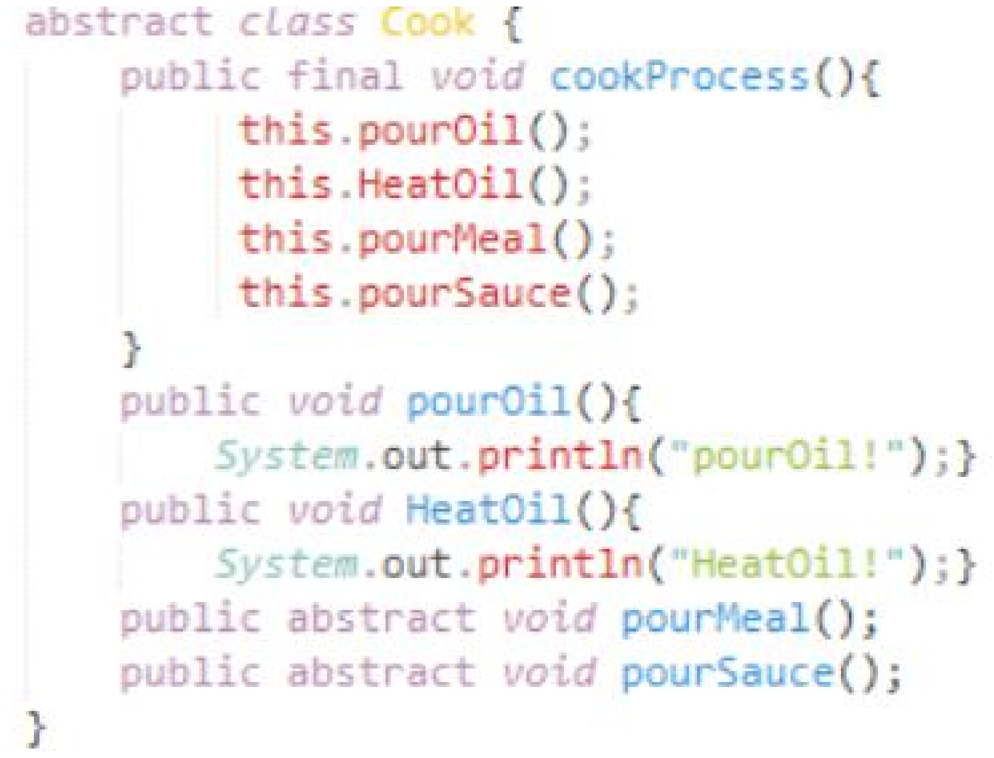
【Template vs Strategy】

模板模式(Compiler Time)
一定是按照次序執行，任何重載不會影響到這個次序；
流程中的某幾個節點會被替換，但順序不變

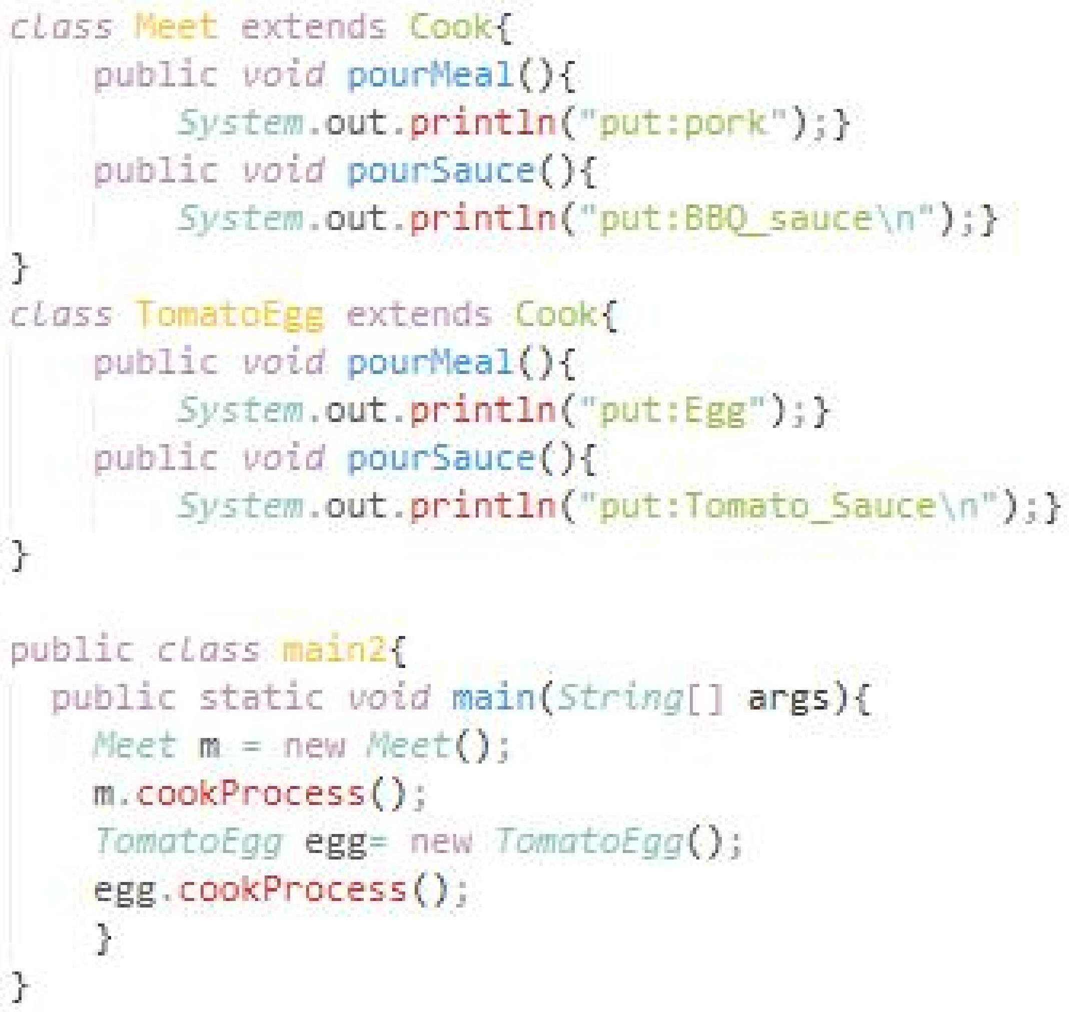
策略模式(Runtime)
它只提供了某個情景下的執行策略，執行順序不做需求；
整個流程都是可以被替換的


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()

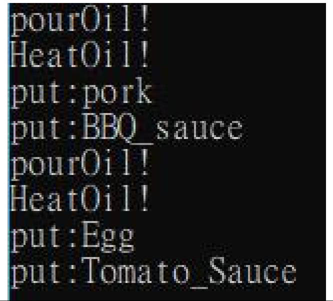
範例1. 煮飯流程(下油、加熱、選肉、選醬)



```java
abstract class Cook {
    public final void cookProcess(){
        this.pourOil();
        this.HeatOil();
        this.pourMeal();
        this.pourSauce();
    }
    public void pourOil(){
        System.out.println("pourOil!");}
    public void HeatOil(){
        System.out.println("HeatOil!");}
    public abstract void pourMeal();
    public abstract void pourSauce();
}
```

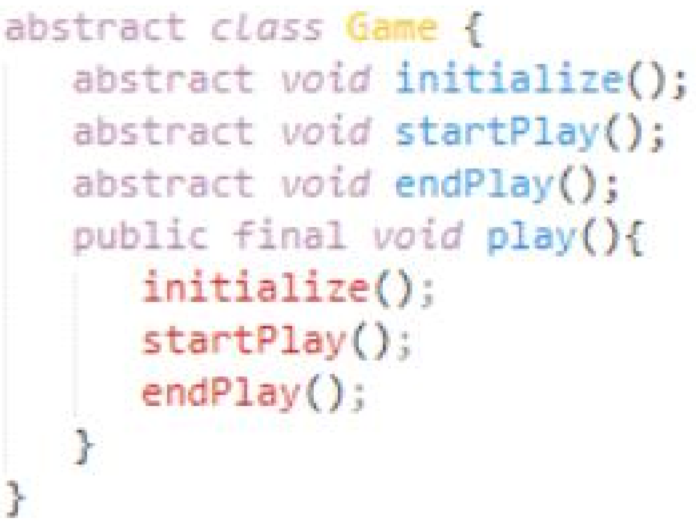


```java
class Meet extends Cook{
    public void pourMeal(){
        System.out.println("put:pork");}
    public void pourSauce(){
        System.out.println("put:BBQ_sauce\n");}
}
class TomatoEgg extends Cook{
    public void pourMeal(){
        System.out.println("put:Egg");}
    public void pourSauce(){
        System.out.println("put:Tomato_Sauce\n");}
}

public class main2{
  public static void main(String[] args){
    Meet m = new Meet();
    m.cookProcess();
    TomatoEgg egg= new TomatoEgg();
    egg.cookProcess();
    }
}
```

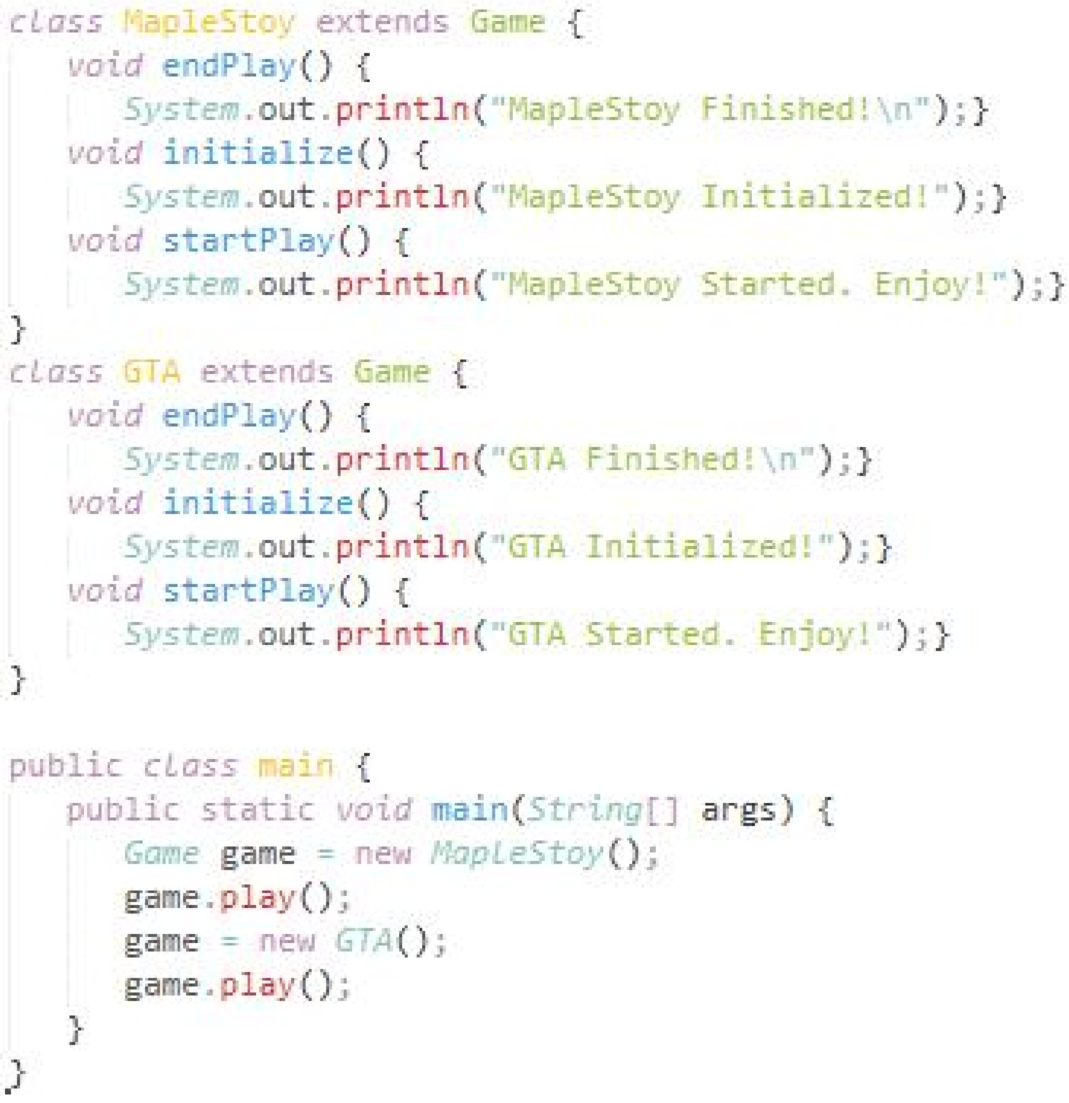


```
pourOil!
HeatOil!
put:pork
put:BBQ_sauce
pourOil!
HeatOil!
put:Egg
put:Tomato_Sauce
```

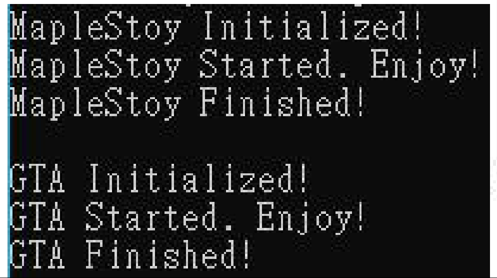
範例2. 玩遊戲流程(啟動/開始/關閉)



```java
abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();
    public final void play(){
        initialize();
        startPlay();
        endPlay();
    }
}
```



```java
class MapleStoy extends Game {
    void endPlay() {
        System.out.println("MapleStoy Finished!\n");}
    void initialize() {
        System.out.println("MapleStoy Initialized!");}
    void startPlay() {
        System.out.println("MapleStoy Started. Enjoy!");}
}
class GTA extends Game {
    void endPlay() {
        System.out.println("GTA Finished!\n");}
    void initialize() {
        System.out.println("GTA Initialized!");}
    void startPlay() {
        System.out.println("GTA Started. Enjoy!");}
}

public class main {
    public static void main(String[] args) {
        Game game = new MapleStoy();
        game.play();
        game = new GTA();
        game.play();
    }
}
```



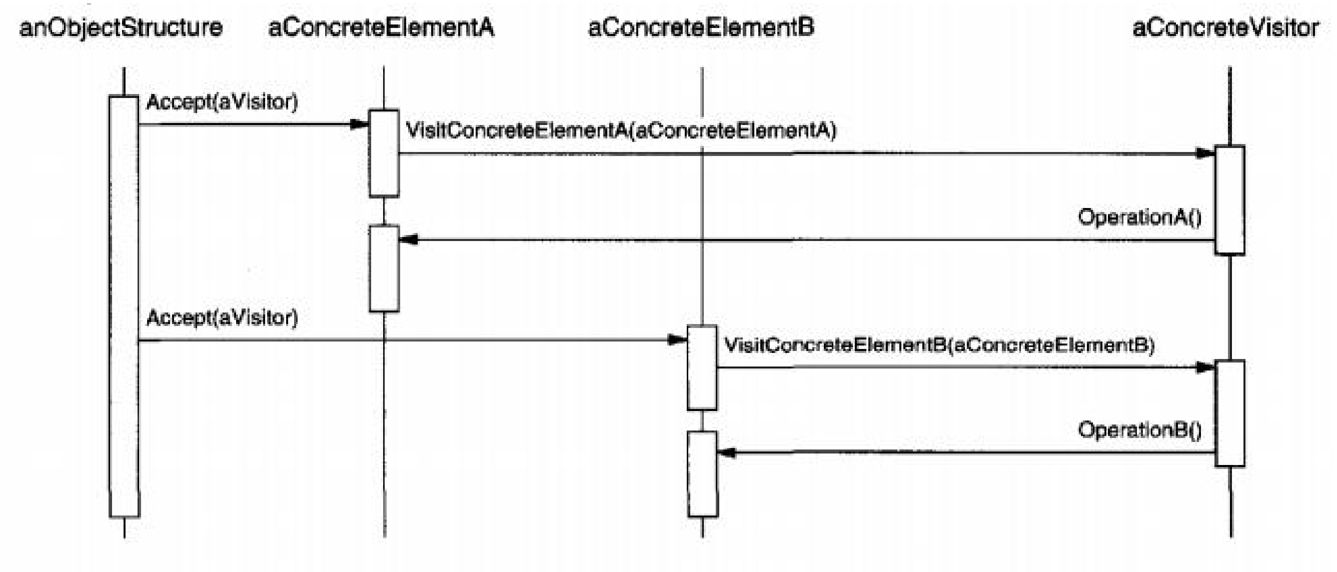
```
MapleStoy Initialized!
MapleStoy Started. Enjoy!
MapleStoy Finished!

GTA Initialized!
GTA Started. Enjoy!
GTA Finished!
```

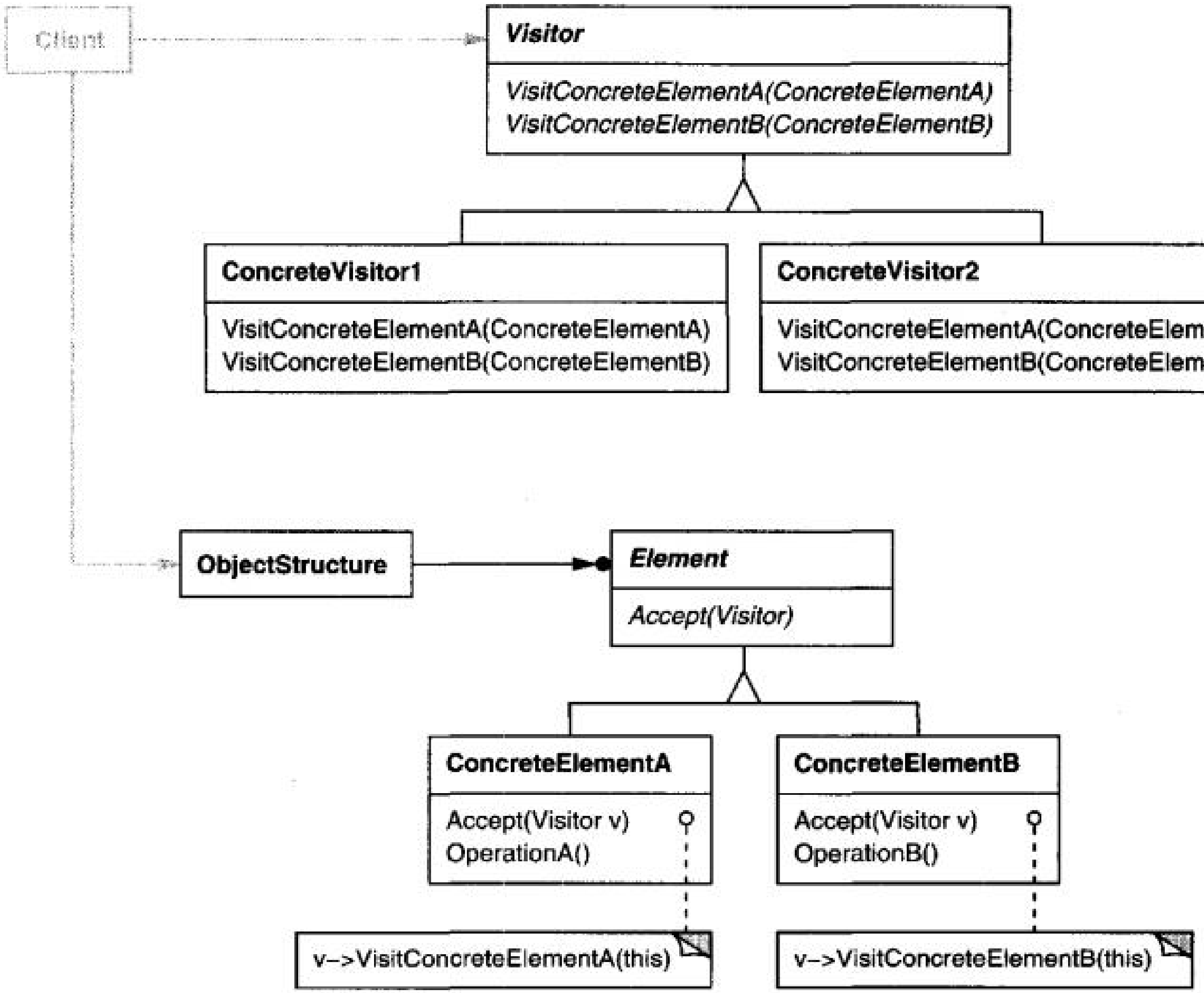
23. Visitor 參訪者模式

Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.

表示要對對象結構的元素執行的操作。
訪問者允許您定義新操作，而無需更改其操作的元素的類。

當你有很多元件(element)且數量固定
而這些元件常常需要被執行某些操作就可以使用Visitor
透過訪問者的方式來對這些元件進行操作

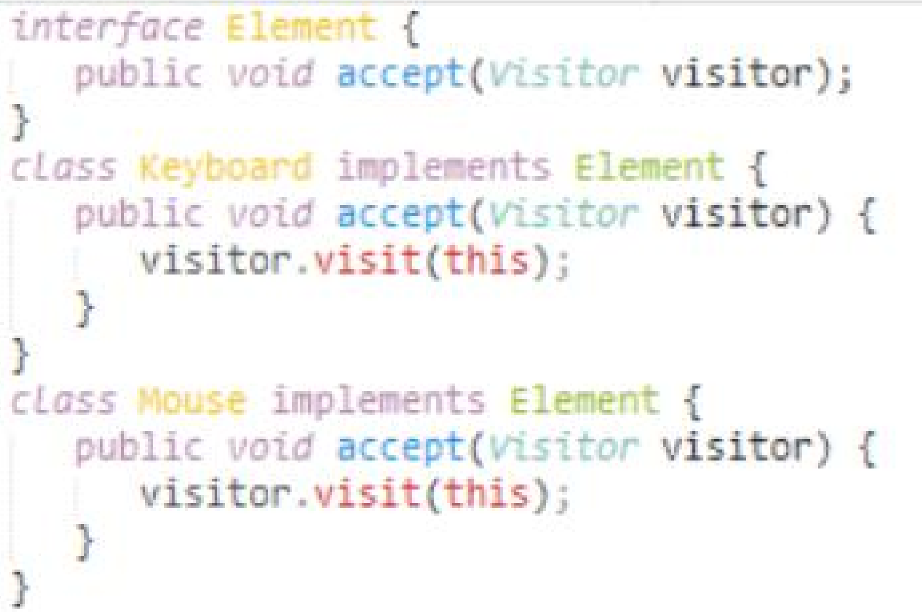



Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

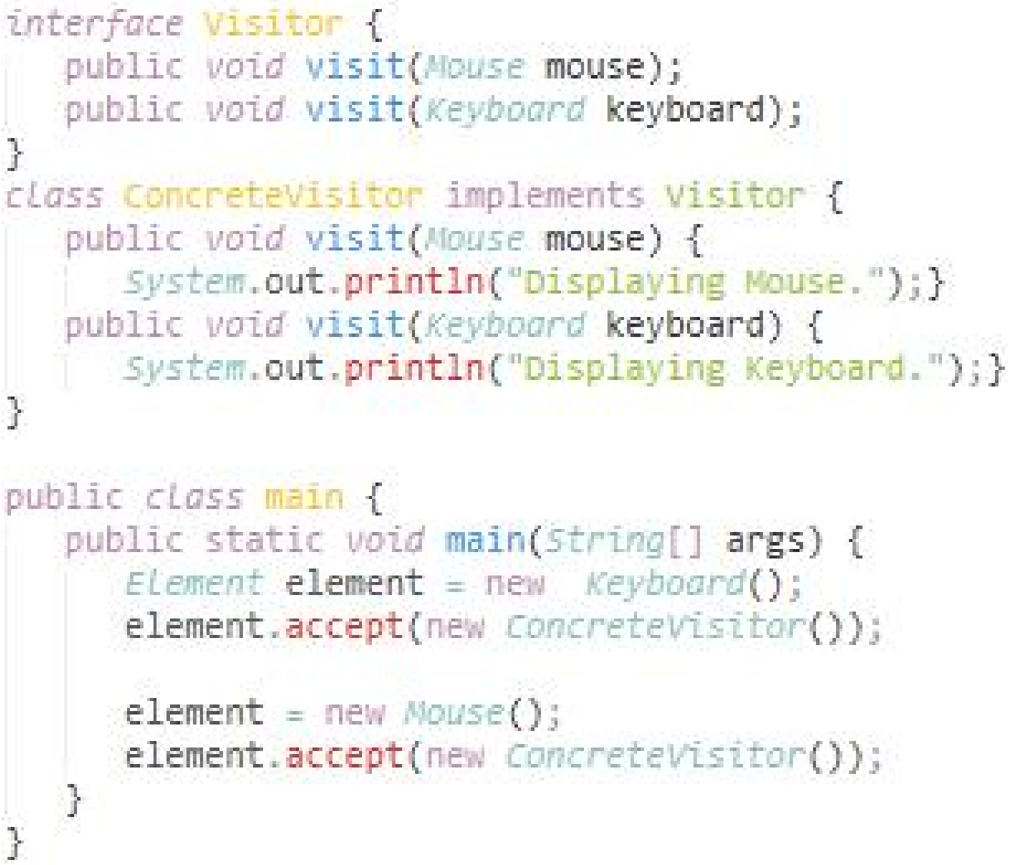
1.先把Element跟Visitor的方法定義好
Element只有accept的方法(Override)

2.Visitor則要根據有幾個Element就會有幾個Visit方法(Overload)

範例1.

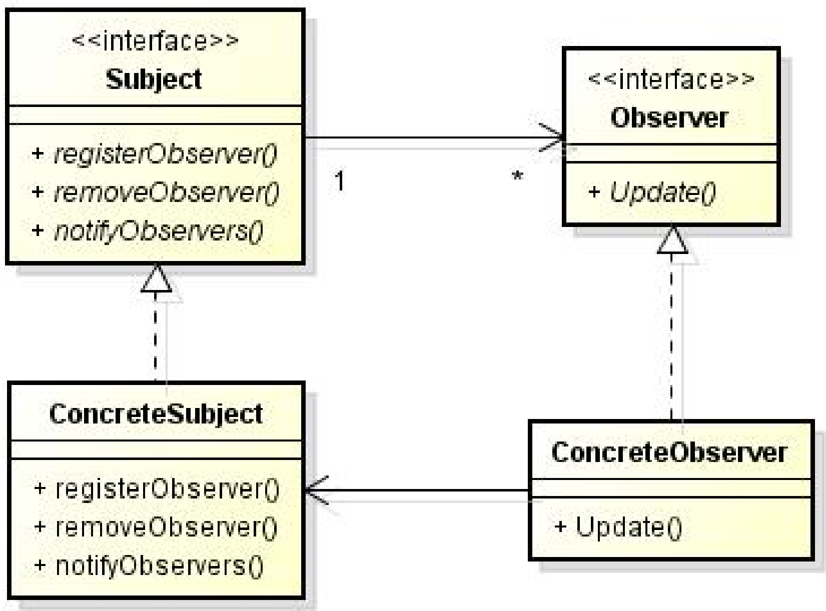


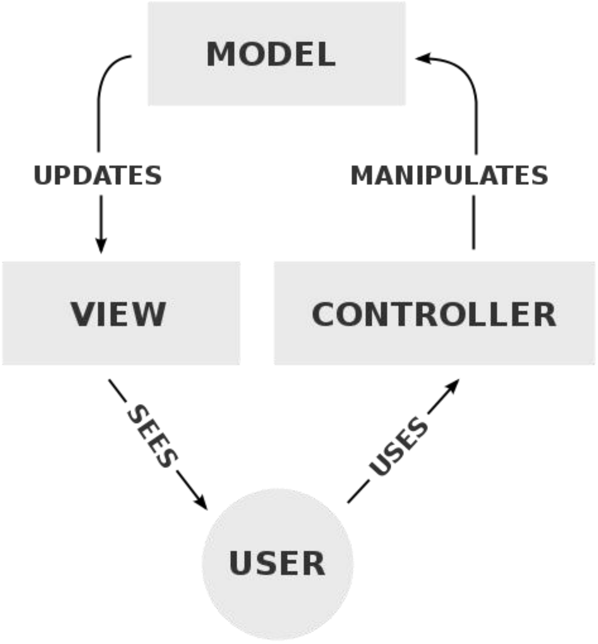
```
interface Element {
    public void accept(Visitor visitor);
}
class Keyboard implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
class Mouse implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```



```
interface Visitor {
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
}
class ConcreteVisitor implements Visitor {
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");}
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");}
}

public class main {
    public static void main(String[] args) {
        Element element = new  Keyboard();
        element.accept(new ConcreteVisitor());

        element = new Mouse();
        element.accept(new ConcreteVisitor());
    }
}
```

24. Model-View-Control 模式





簡單來說

View
就是使用者可以看到的東西

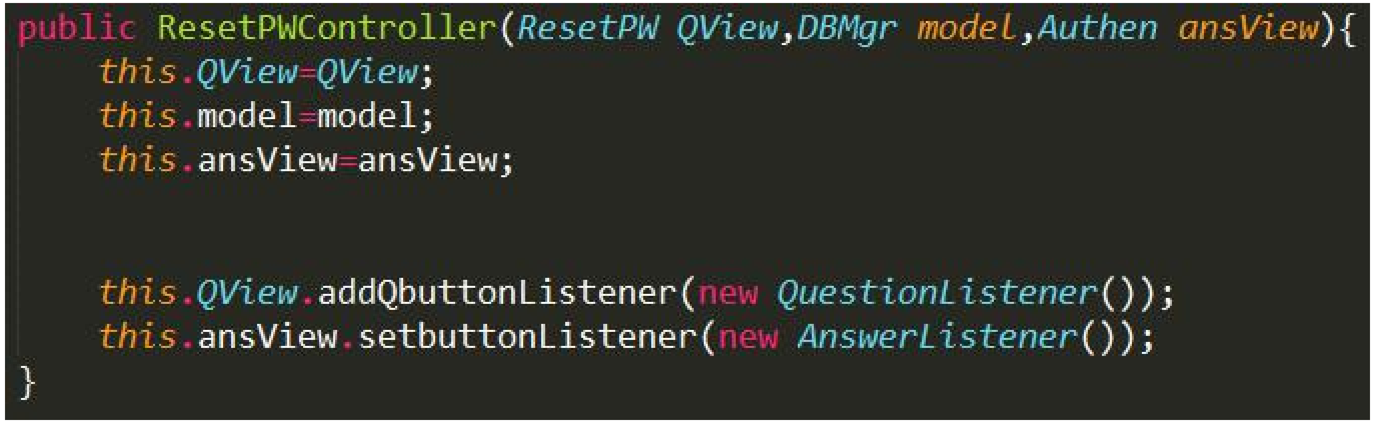
Controller
是當使用者透過View操作之後負責命令Model去做事情

Model
就是負責處理事情，處理完後再更新View到最新狀態

Q:當使用者透過View操作的時候Controller要怎麼知道呢?

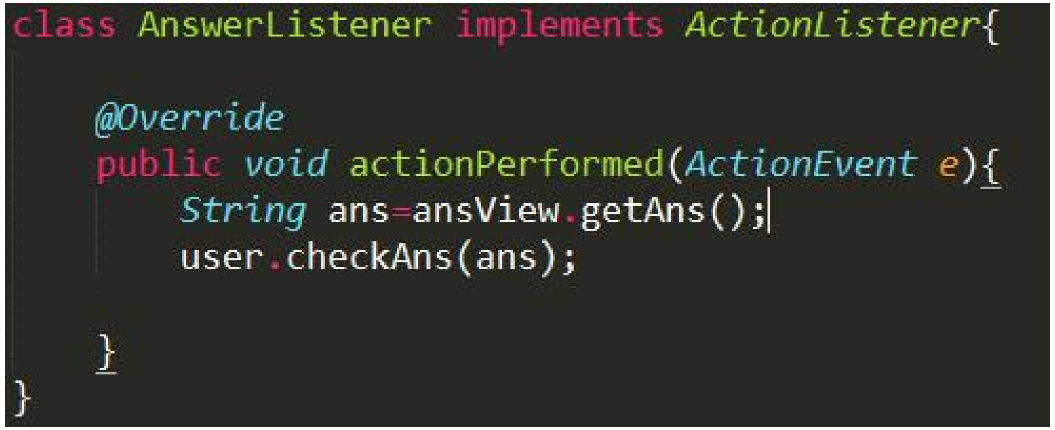
A:透過ActionListener在View上註冊Controller的Listener監聽View的動作

↓透過View的方法註冊進去



```
public ResetPWController(ResetPW QView,DBMgr model,Authen ansView){
    this.QView=QView;
    this.model=model;
    this.ansView=ansView;


    this.QView.addQbuttonListener(new QuestionListener());
    this.ansView.setbuttonListener(new AnswerListener());
}
```

↓Controller內已經實作好的Listenter



```
class AnswerListener implements ActionListener{

    @Override
    public void actionPerformed(ActionEvent e){
        String ans=ansView.getAns();
        user.checkAns(ans);

    }
}
```

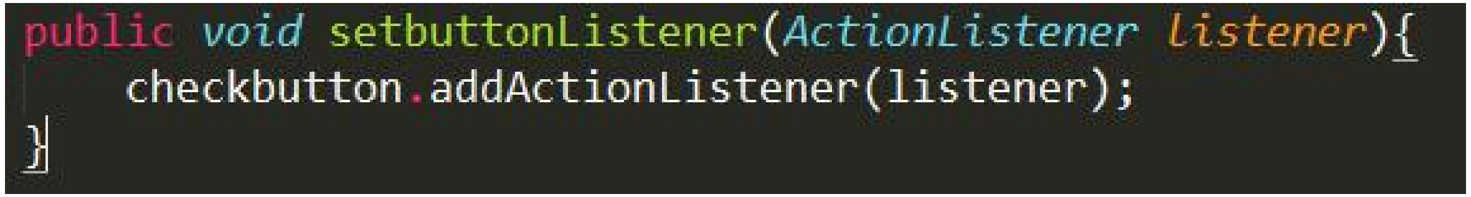
Q:Model要怎麼通知View要更新?

A:用Observer Pattern，model是Subject，View是observer

Subject(model)裡註冊observer(view)，更新時就會通知observer(view)需要更新

我給的MVC Code裡面有可以參考

↓View裡面的方法，實際上是註冊到Button



```
public void setbuttonListener(ActionListener listener){
    checkbutton.addActionListener(listener);
}
```

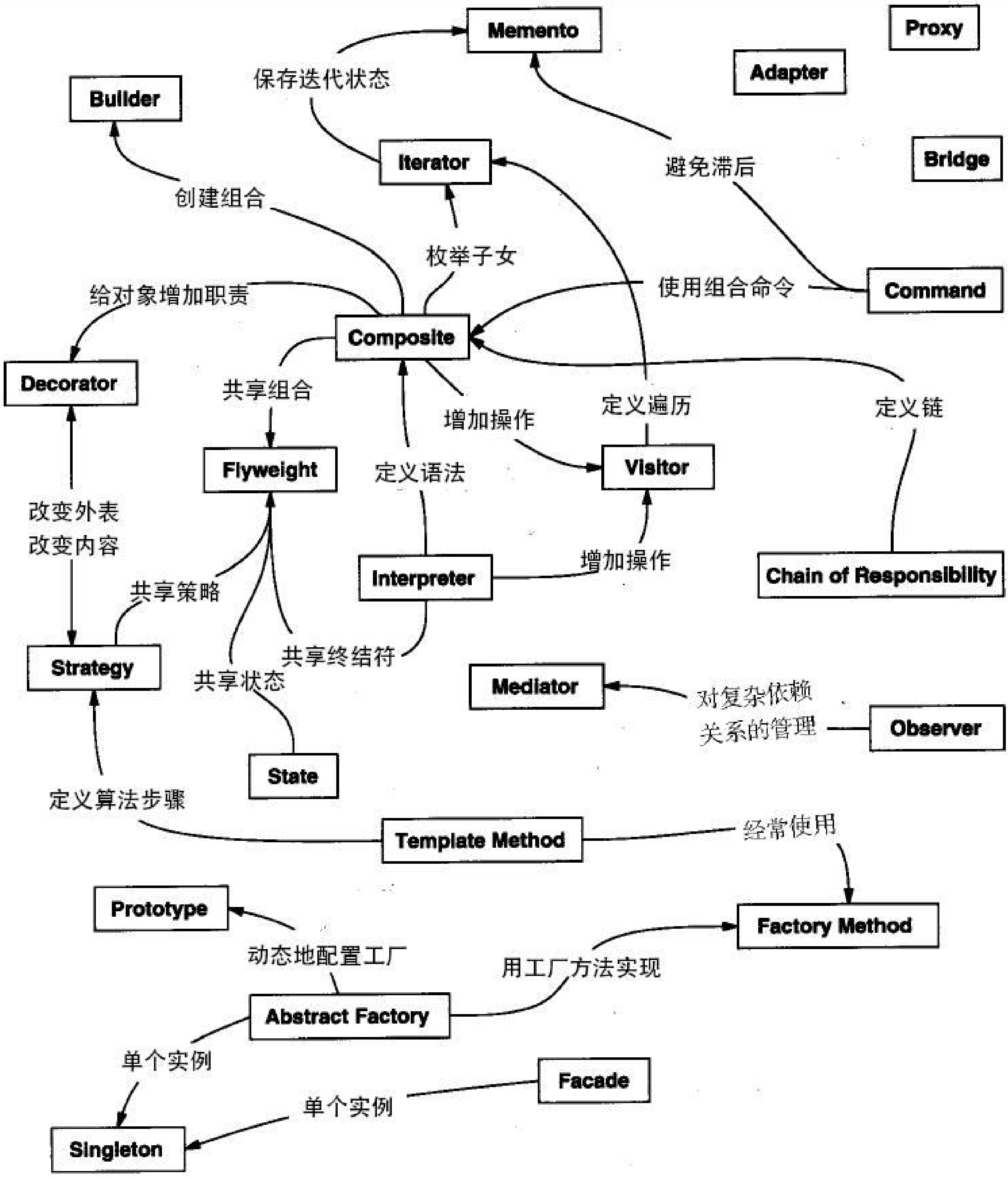
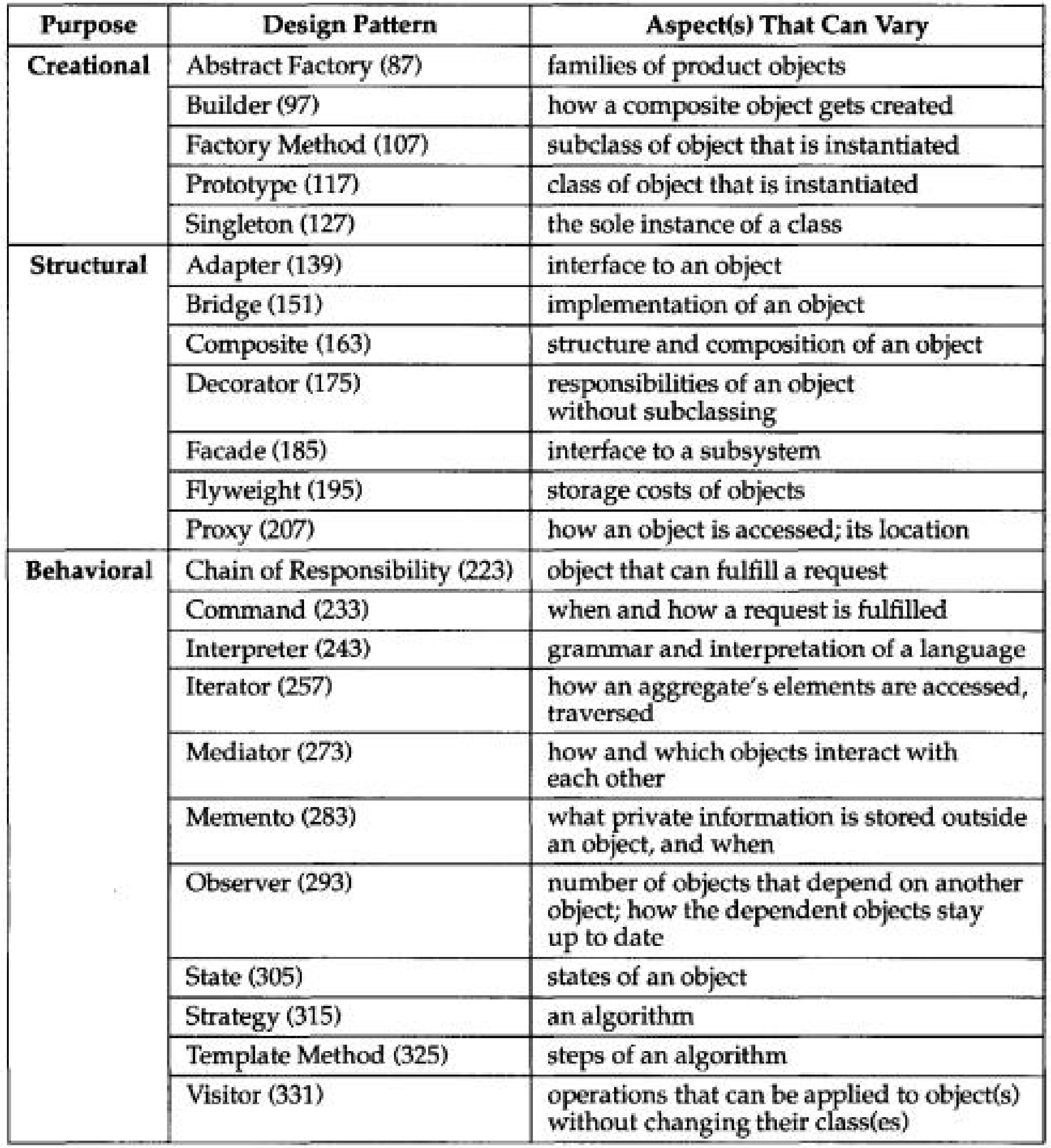

Memento
Proxy
保存迭代状态
Adapter
Builder
Iterator
避免滞后
Bridge
创建组合
枚举子女
给对象增加职责
使用组合命令
Command
Composite
Decorator
共享组合
定义遍历
定义链
增加操作
Flyweight
定义语法
Visitor
改变外表
改变内容
增加操作
Chain of Responsibility
共享策略
Interpreter
共享终结符
Strategy
共享状态
Mediator
对复杂依赖
关系的管理
Observer
State
定义算法步骤
Template Method
经常使用
Prototype
Factory Method
动态地配置工厂
用工厂方法实现
Abstract Factory
单个实例
Facade
单个实例
Singleton

图　设计模式之间的关系



Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

Creational Patterns

Structural Patterns

BehavioralPatterns

可以參考

https://openhome.cc/Gossip/DesignPattern/

http://design-patterns.readthedocs.io/zh_CN/latest/index.html

https://en.wikipedia.org/wiki/Design_Patterns

https://skyyen999.gitbooks.io/-study-design-pattern-in-java/content/

https://zhuanlan.zhihu.com/shejimoshi

https://blog.csdn.net/zhengzhb/article/details/7697549