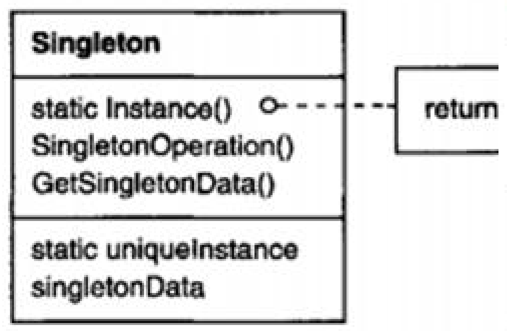
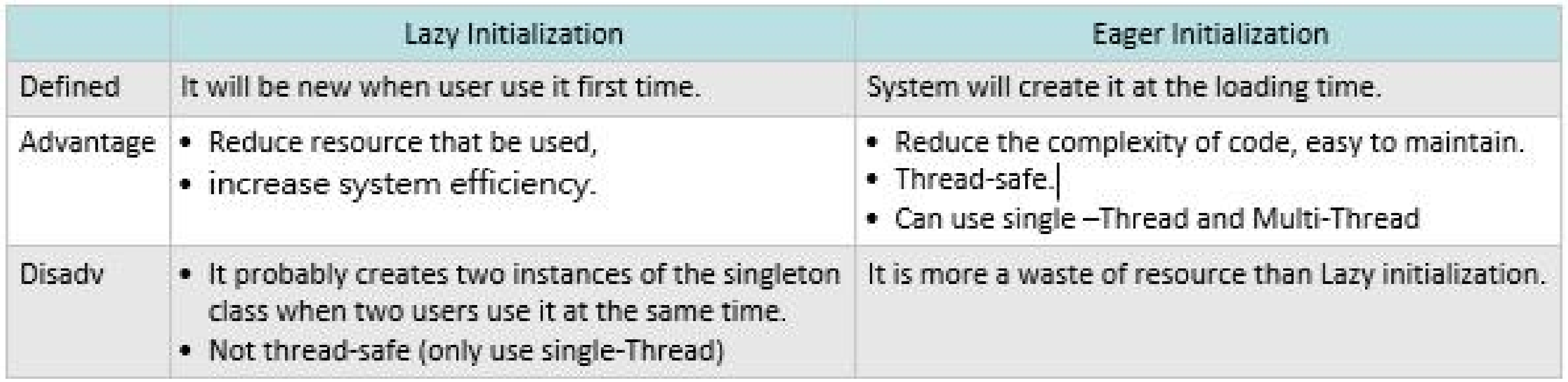
Singleton 單例模式 (P)

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。
(当您想控制实例数目，节省系统资源的时候。)

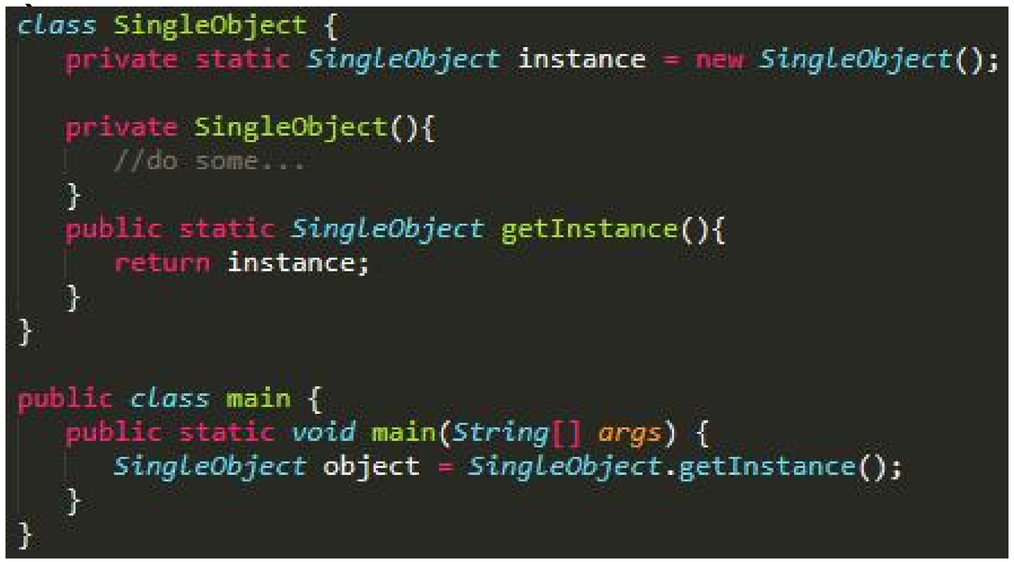
Singleton就是要確保物件只有**一個**實例可以被重複使用
所以常常是被使用率極高(重複存取)的原件才會這樣做





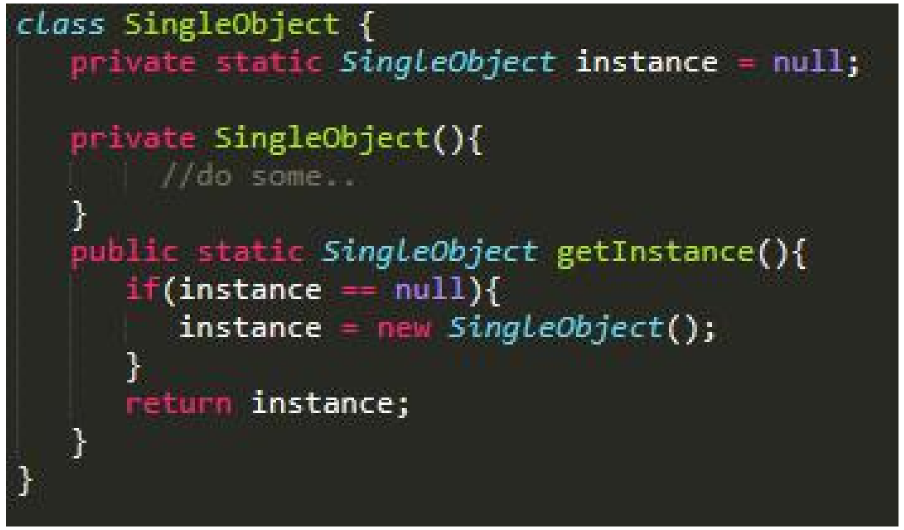
	Lazy Initialization	Eager Initialization
Defined	It will be new when user use it first time.	System will create it at the loading time.
Advantage	• Reduce resource that be used, • increase system efficiency.	• Reduce the complexity of code, easy to maintain. • Thread-safe. • Can use single –Thread and Multi-Thread
Disadv	• It probably creates two instances of the singleton class when two users use it at the same time. • Not thread-safe (only use single-Thread)	It is more a waste of resource than Lazy initialization.

EagerSingleton (減少複雜 / 容易維護)



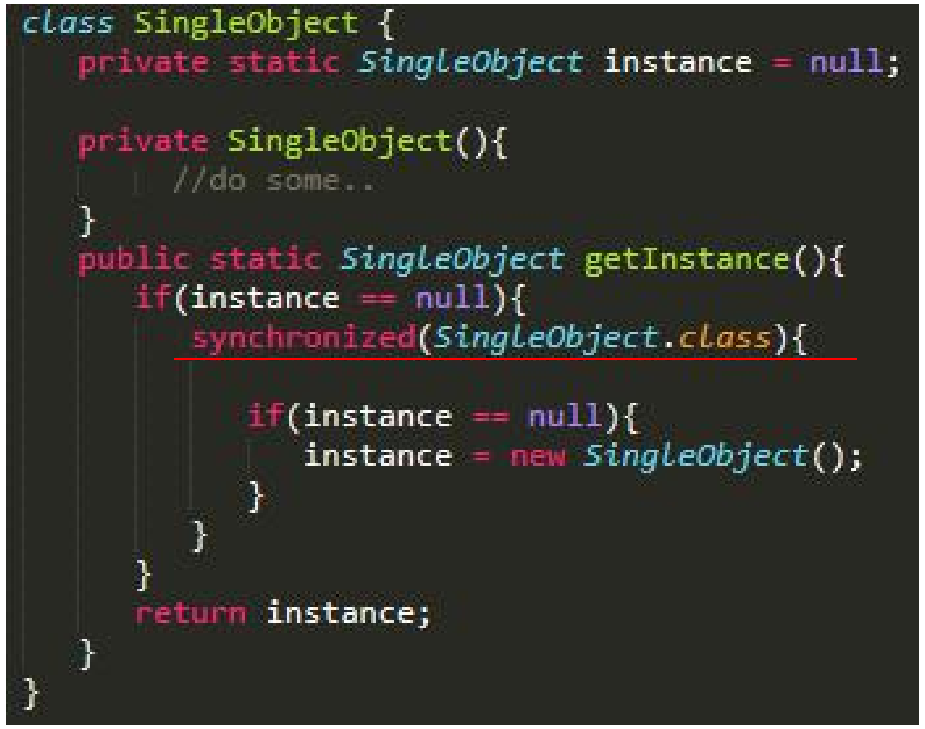
```java
class SingleObject {
   private static SingleObject instance = new SingleObject();

   private SingleObject(){
      //do some...
   }
   public static SingleObject getInstance(){
      return instance;
   }
}

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

LazySingleton (減少資源 / 提高效率)



```java
class SingleObject {
   private static SingleObject instance = null;

   private SingleObject(){
      //do some..
   }
   public static SingleObject getInstance(){
      if(instance == null){
         instance = new SingleObject();
      }
      return instance;
   }
}
```

Double-check-locking (解決Lazy)
解決可能同時創造2個 instances 的問題



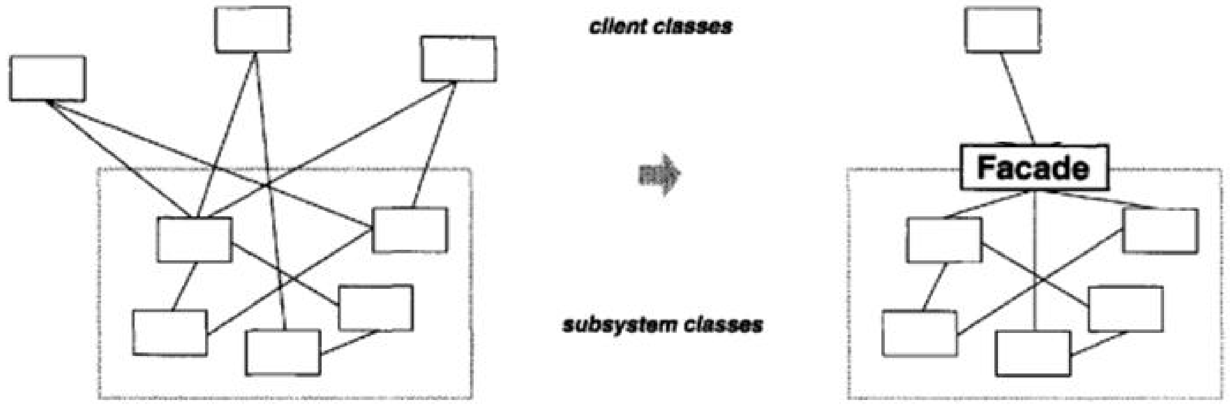
```java
class SingleObject {
   private static SingleObject instance = null;

   private SingleObject(){
      //do some..
   }
   public static SingleObject getInstance(){
      if(instance == null){
         synchronized(SingleObject.class){

            if(instance == null){
               instance = new SingleObject();
            }
         }
      }
      return instance;
   }
}
```

Facade 外觀模式 (P)

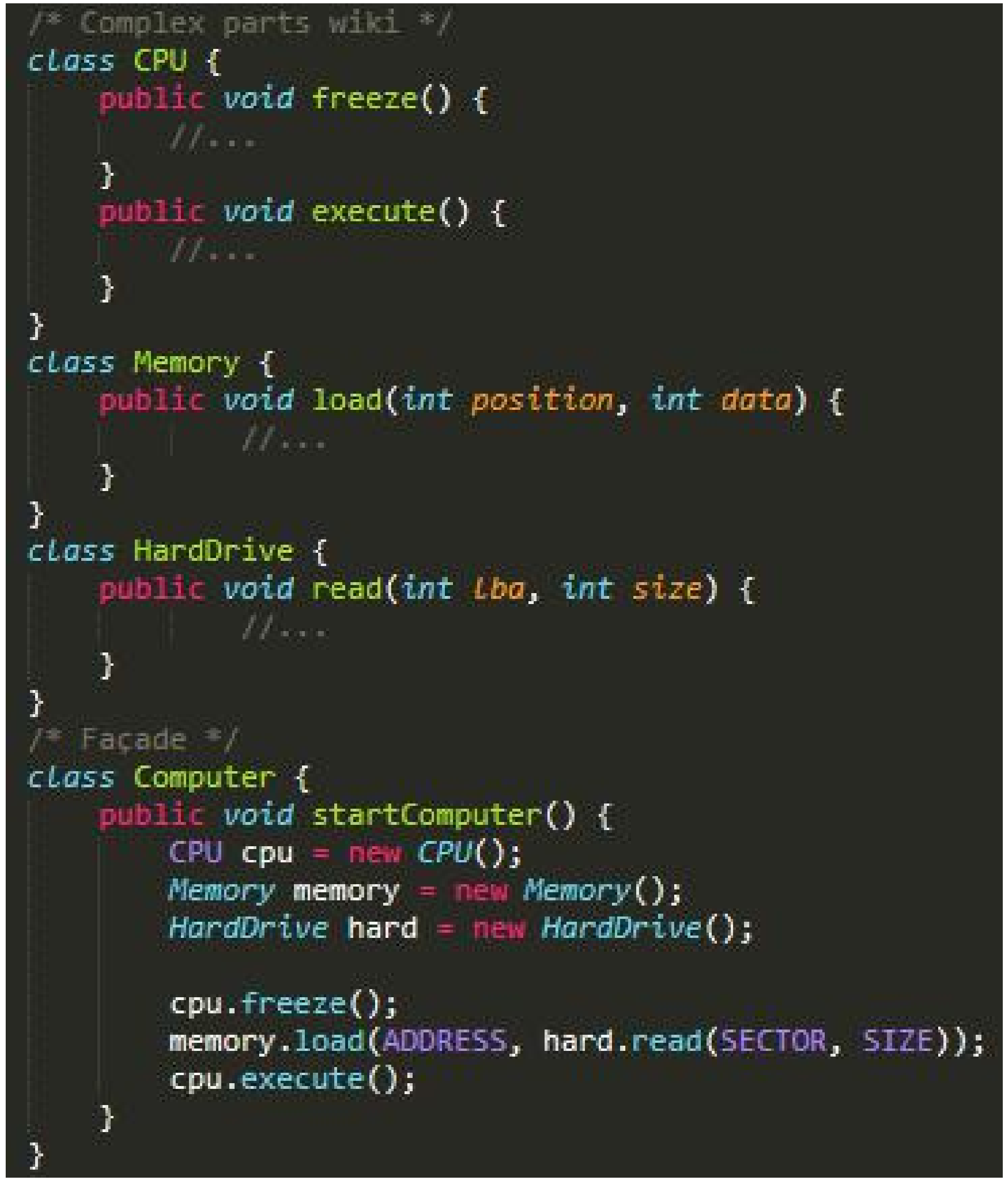
Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem easier to use.

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。
(降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。)

- 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

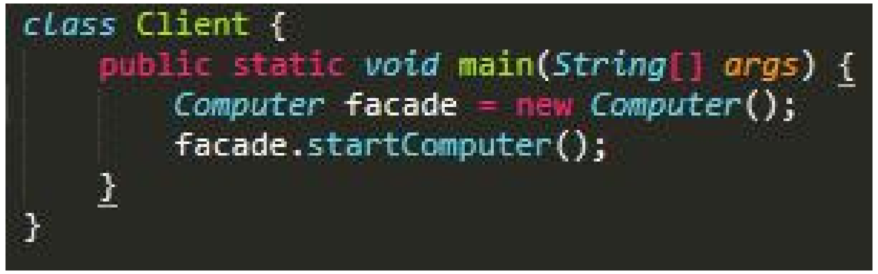


Complex (複雜功能)



```
/* Complex parts wiki */
class CPU {
    public void freeze() {
        //...
    }
    public void execute() {
        //...
    }
}
class Memory {
    public void load(int position, int data) {
            //...
    }
}
class HardDrive {
    public void read(int lba, int size) {
            //...
    }
}
/* Façade */
class Computer {
    public void startComputer() {
        CPU cpu = new CPU();
        Memory memory = new Memory();
        HardDrive hard = new HardDrive();

        cpu.freeze();
        memory.load(ADDRESS, hard.read(SECTOR, SIZE));
        cpu.execute();
    }
}
```

Client 只需用到Façade

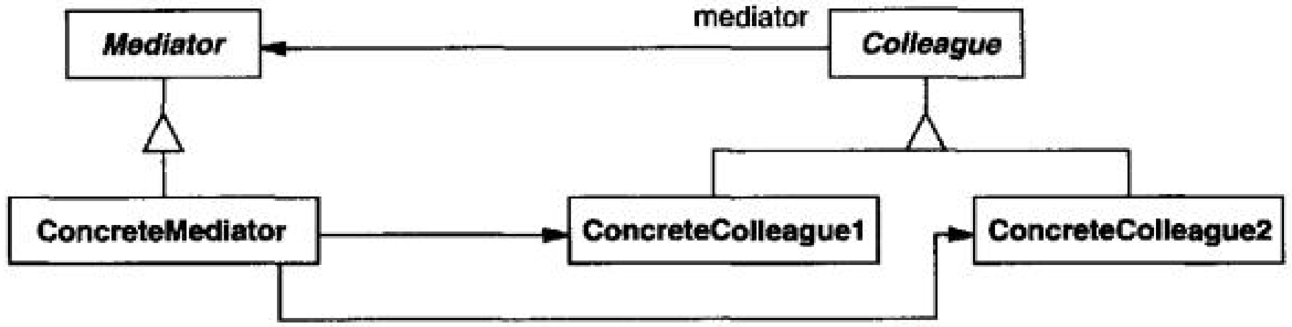


```
class Client {
    public static void main(String[] args) {
        Computer facade = new Computer();
        facade.startComputer();
    }
}
```

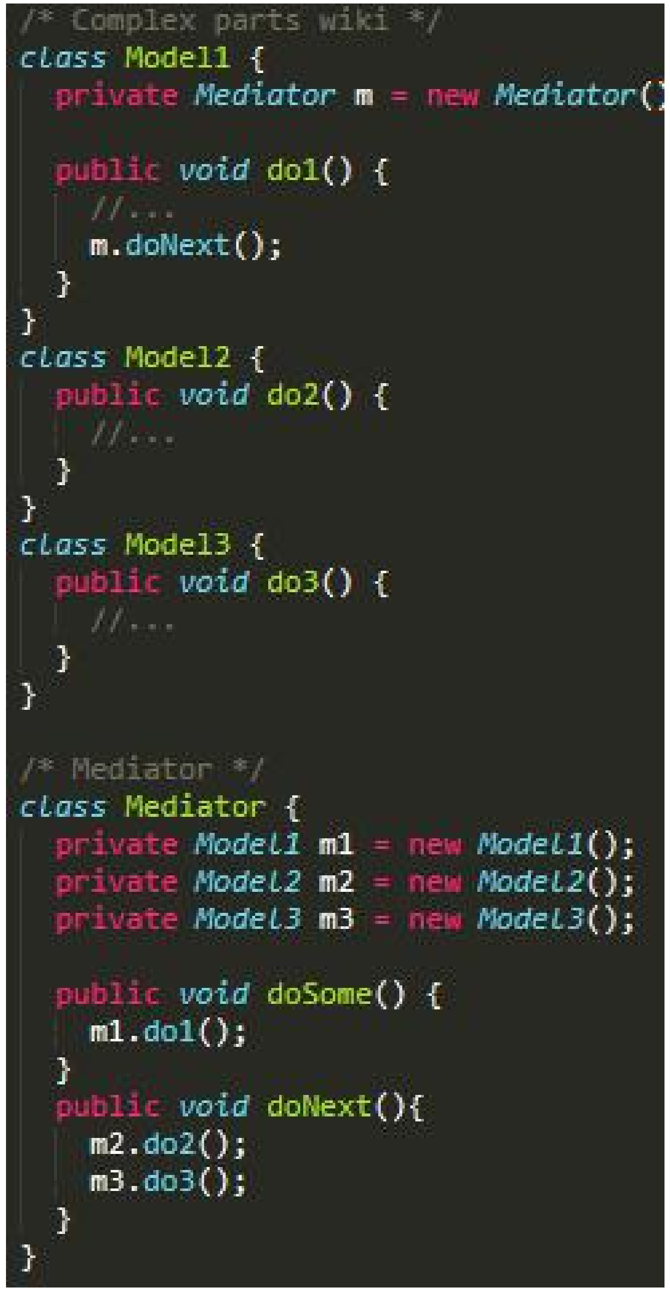
Mediator 中介者模式 (P)

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。
(多个类相互耦合，形成了网状结构時)



Mediator



```
/* Complex parts wiki */
class Model1 {
  private Mediator m = new Mediator()

  public void do1() {
    //...
    m.doNext();
  }
}
class Model2 {
  public void do2() {
    //...
  }
}
class Model3 {
  public void do3() {
    //...
  }
}

/* Mediator */
class Mediator {
  private Model1 m1 = new Model1();
  private Model2 m2 = new Model2();
  private Model3 m3 = new Model3();

  public void doSome() {
    m1.do1();
  }
  public void doNext(){
    m2.do2();
    m3.do3();
  }
}
```

【Façade vs Mediator】

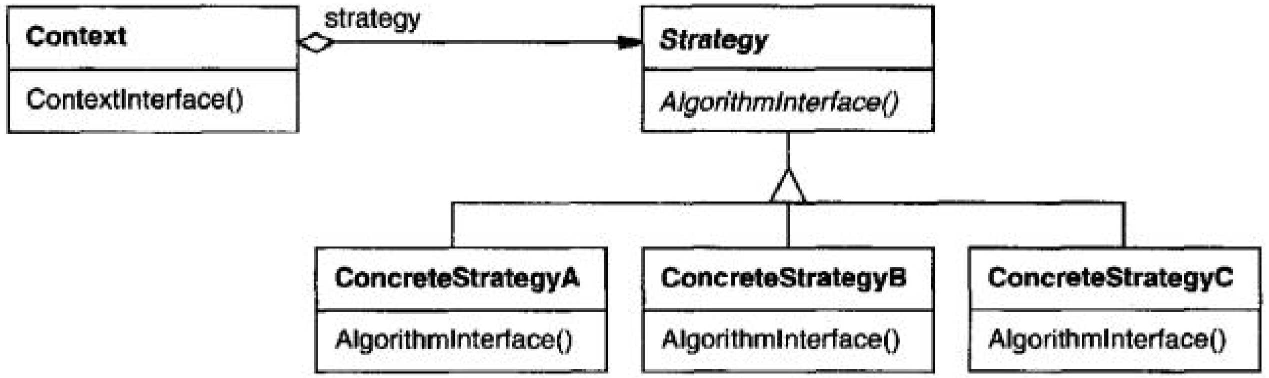
- Façade 提供一個介面對外提供操作，使用者不用知道任何底下的子系統運作
- Mediator 可以解決耦合太高問題，讓中介者調節所有決定與操作

Strategy 策略模式 (EP)

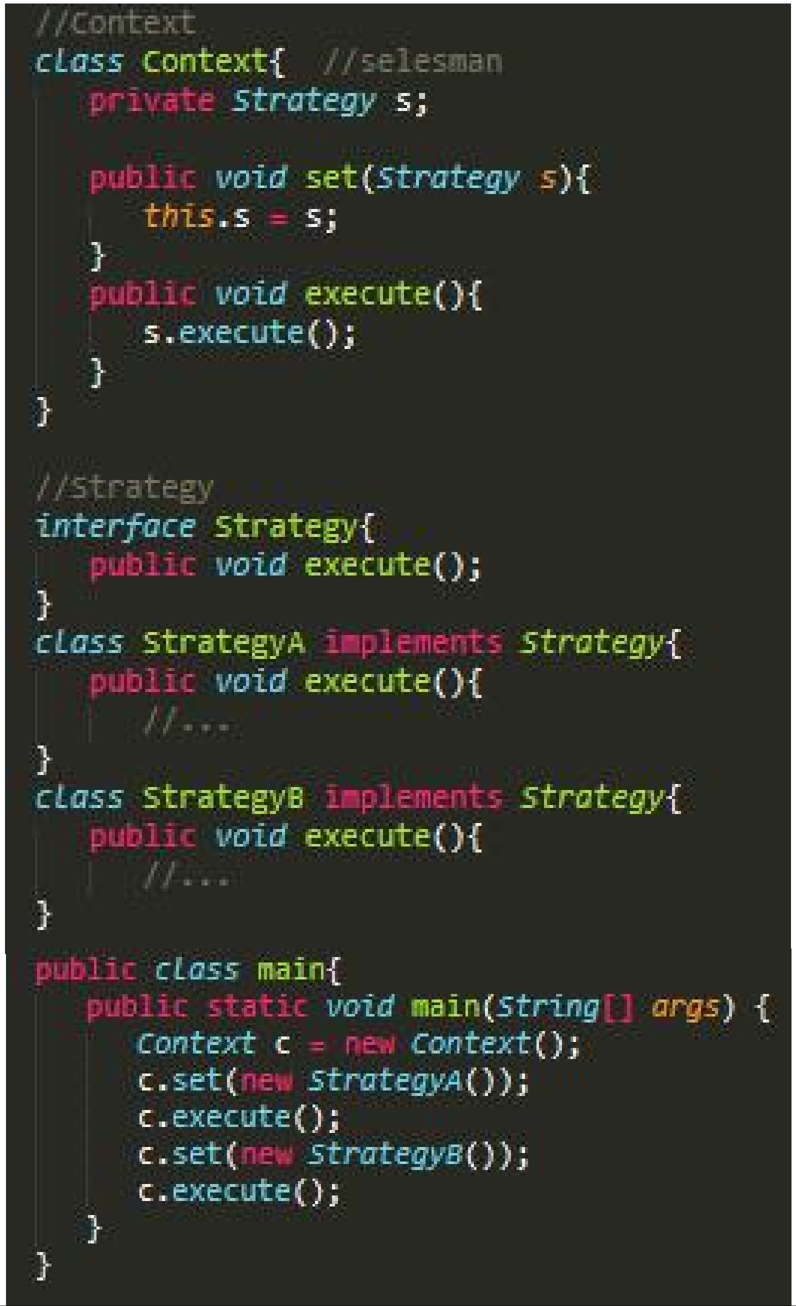
Define a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.

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。
(解決在有多种算法相似的情況下，难以维护的問題)

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



Strategy



```java
//Context
class Context{  //selesman
    private Strategy s;

    public void set(Strategy s){
        this.s = s;
    }
    public void execute(){
        s.execute();
    }
}

//Strategy
interface Strategy{
    public void execute();
}
class StrategyA implements Strategy{
    public void execute(){
        //...
}
class StrategyB implements Strategy{
    public void execute(){
        //...
}
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();
    }
}
```

【Strategy vs Bridge】

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

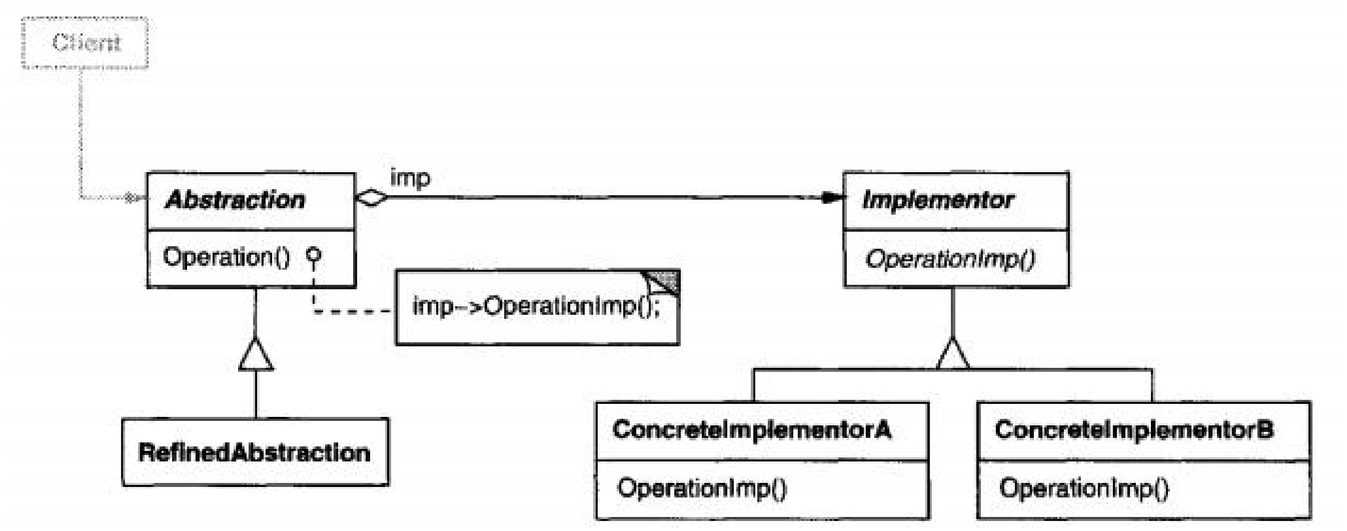
策略模式的缺點，就是所有的策略都必須暴露(讓客戶端選擇)
改善這個缺陷需要跟簡單工廠模式結合混編

Bridge 橋接模式 (AEP)

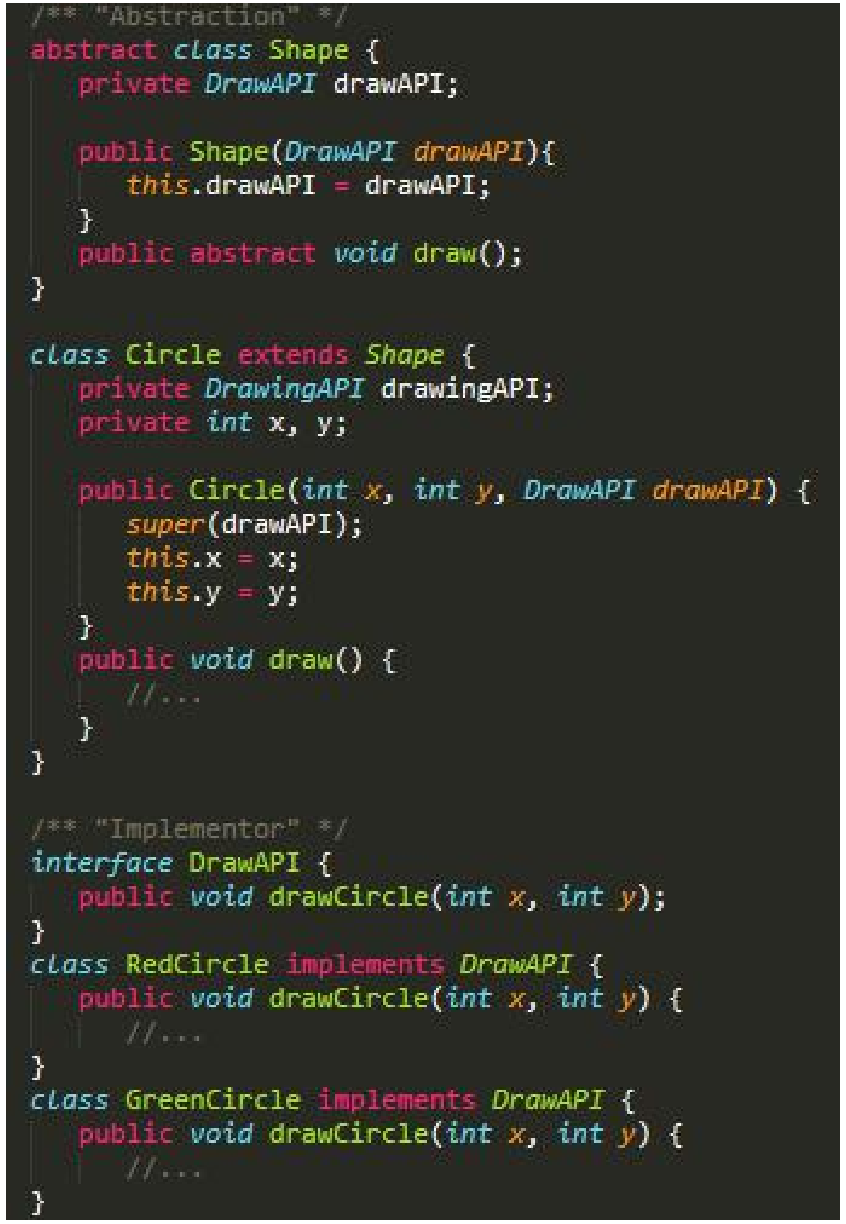
Decouple an abstraction from its implementation so that the two can vary independently.

將抽象與其實現分離，以使兩者可以獨立變化。
(多种变化的情况下，用继承会造成类爆炸问题)

Bridge 把對象和行爲、具體特徵分離開來，使它們可以各自獨立的變化。
如「圓」、「三角」歸於抽象的「形狀」之下，
「畫圓」、「畫三角」歸於行爲的「畫圖」類之下，然後由「形狀」調用「畫圖」。

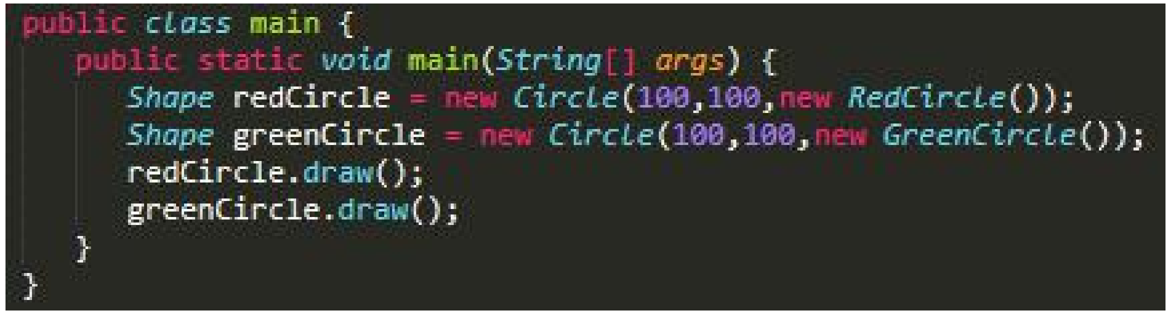


Bridge



```java
/** "Abstraction" */
abstract class Shape {
   private DrawAPI drawAPI;

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

class Circle extends Shape {
   private DrawingAPI drawingAPI;
   private int x, y;

   public Circle(int x, int y, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
   }
   public void draw() {
      //...
   }
}

/** "Implementor" */
interface DrawAPI {
   public void drawCircle(int x, int y);
}
class RedCircle implements DrawAPI {
   public void drawCircle(int x, int y) {
      //...
}
class GreenCircle implements DrawAPI {
   public void drawCircle(int x, int y) {
      //...
}
```

Client

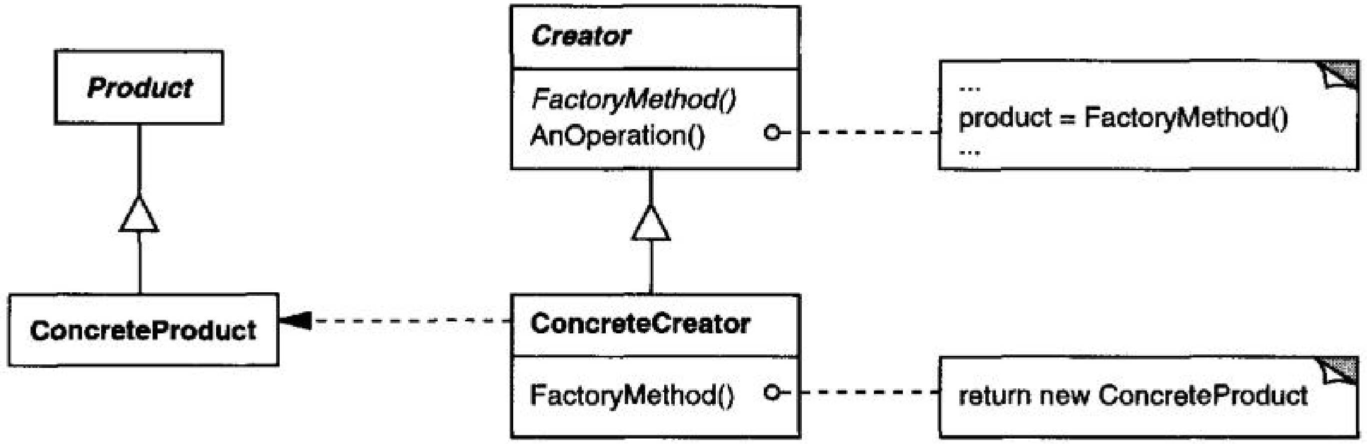


```java
public class main {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100,new RedCircle());
      Shape greenCircle = new Circle(100,100,new GreenCircle());
      redCircle.draw();
      greenCircle.draw();
   }
}
```

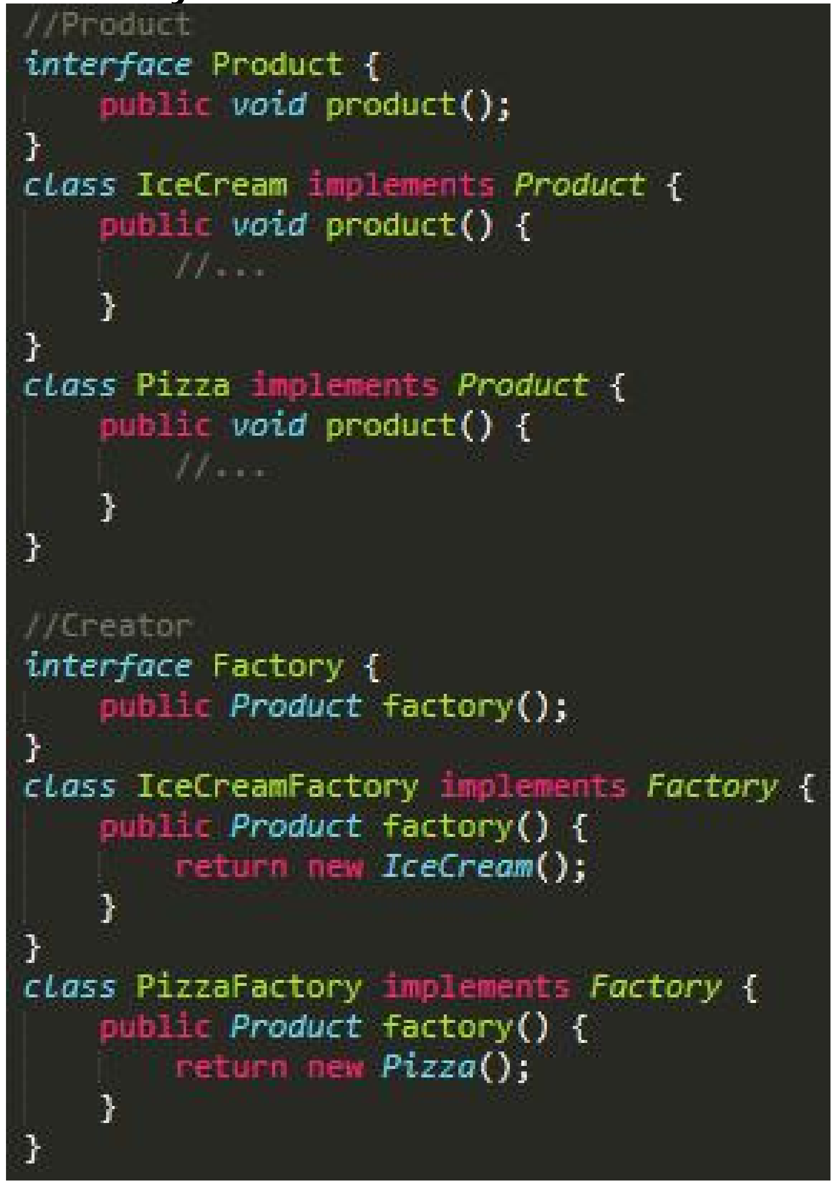
Factory Method 工廠方法 (AEP)

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

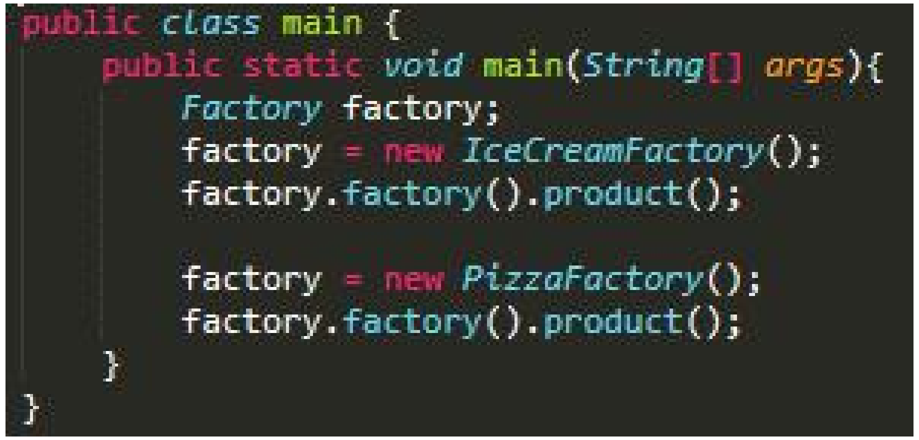
定義用於創建對象的接口，但讓子類決定實例化哪個類。
Factory Method允許類將實例化延遲到子類。
(明确地计划不同条件下创建不同实例时)



Factory



```
//Product
interface Product {
    public void product();
}
class IceCream implements Product {
    public void product() {
        //...
    }
}
class Pizza implements Product {
    public void product() {
        //...
    }
}

//Creator
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();
    }
}
```

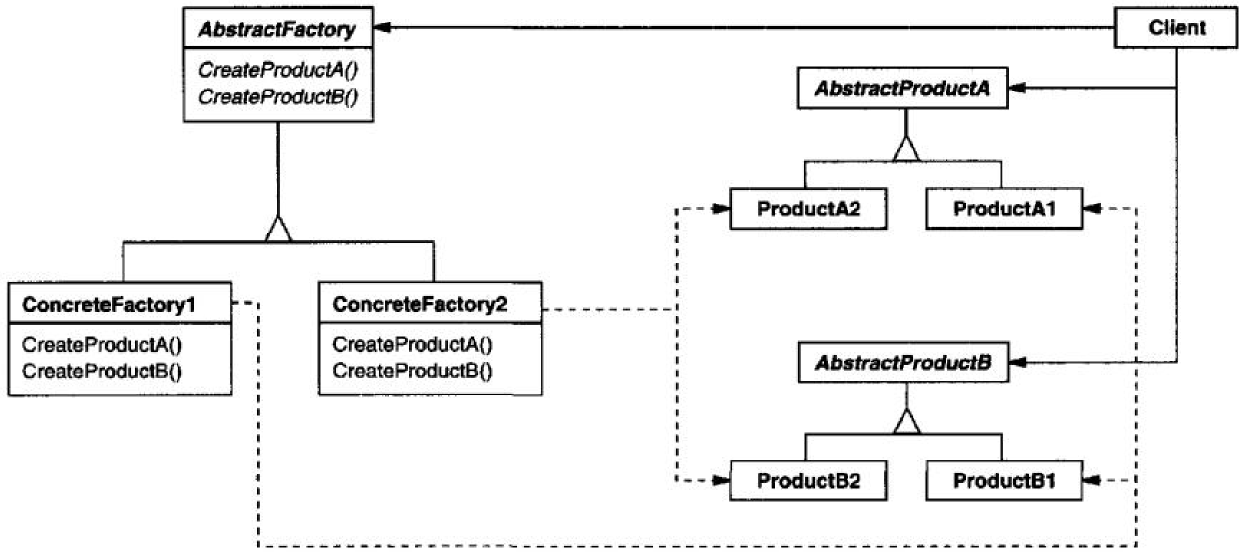


```
public class main {
    public static void main(String[] args){
        Factory factory;
        factory = new IceCreamFactory();
        factory.factory().product();

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

Abstract Factory 抽象工廠 (AEP)

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。
(产品有多于一个的产品族)

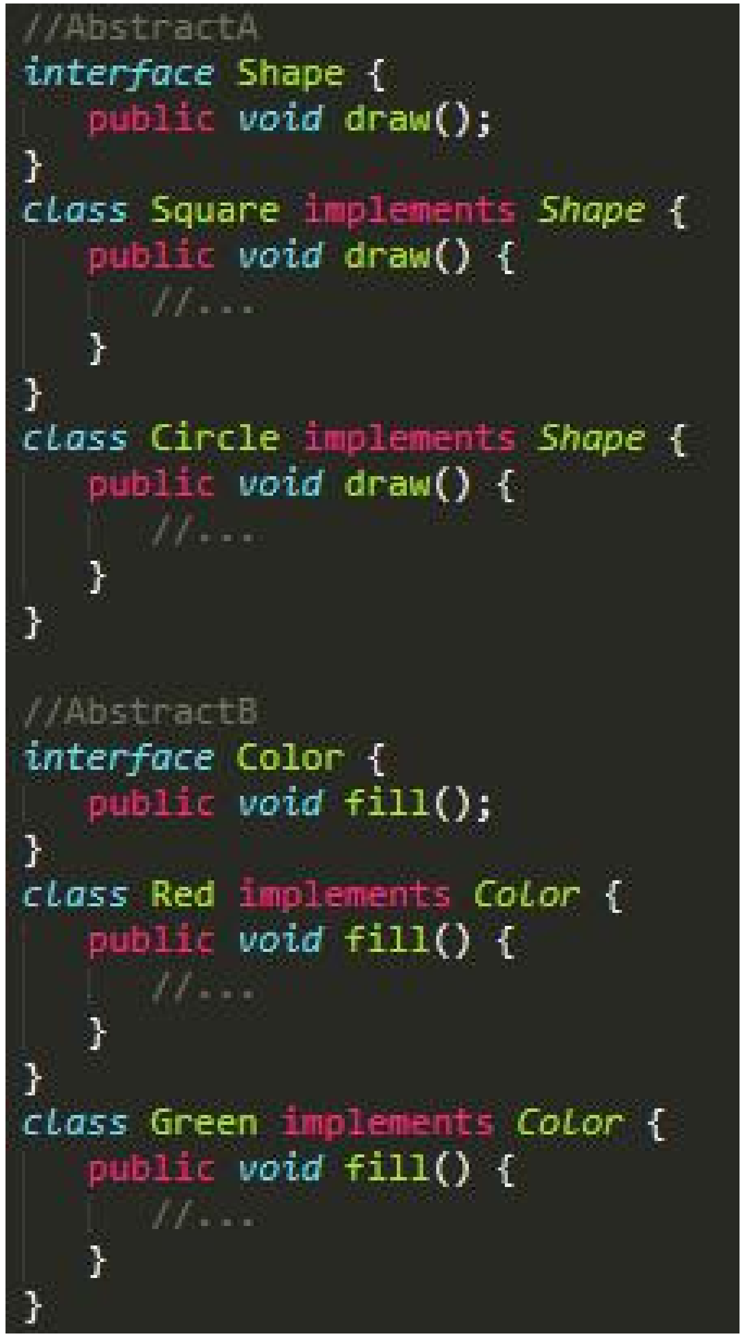


相似產品 不同工廠生產 變成不同產品

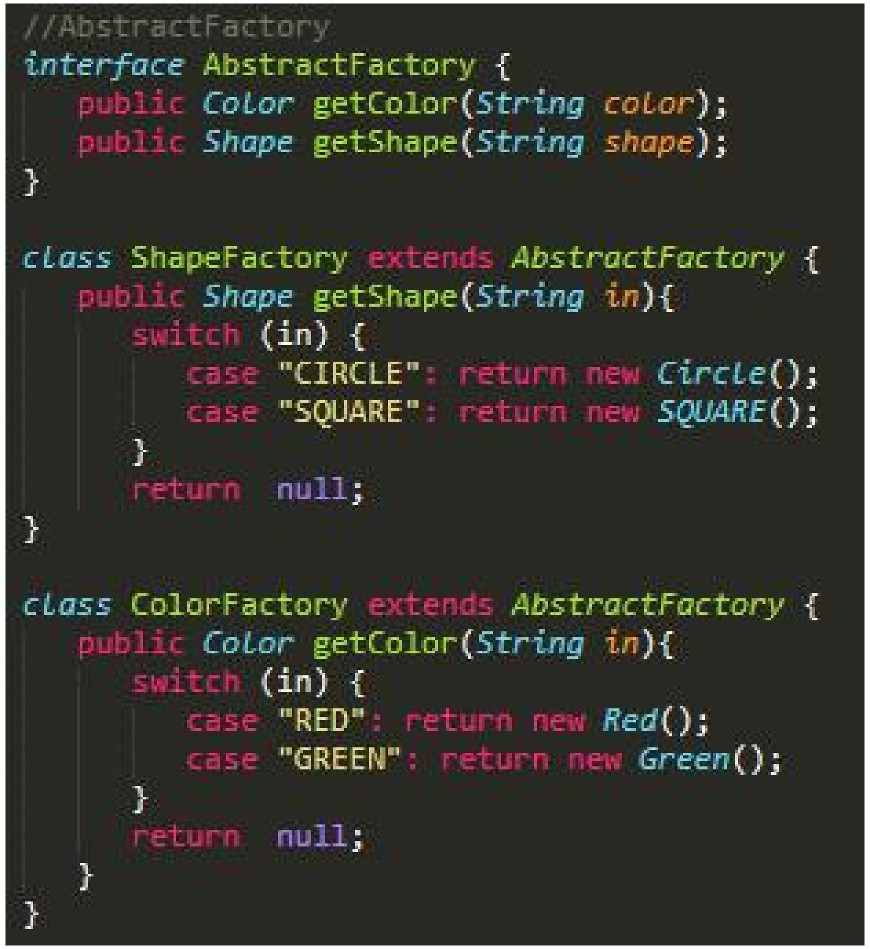
Nike的拖鞋(A1)
原本是Nike工廠生產(F1)

換成Addias工廠生產(F2)
變成addias拖鞋(A2)

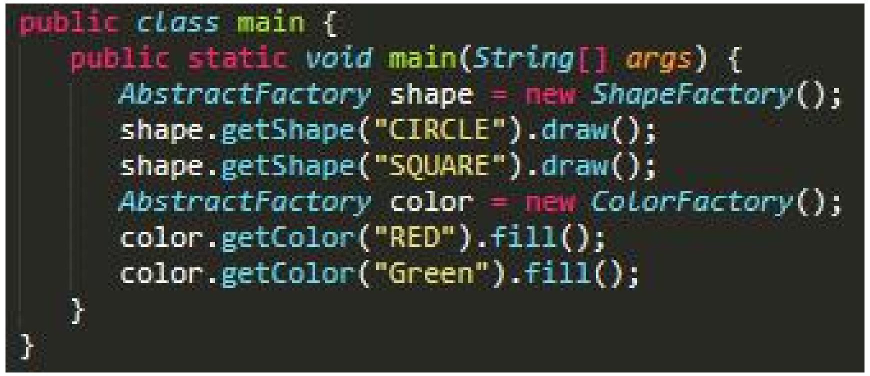
以圖來說有A、B兩種產品
Factory1生產的是產品A1、B1
Factory2生產的是產品A2、B2



```
//AbstractA
interface Shape {
    public void draw();
}
class Square implements Shape {
    public void draw() {
        //...
    }
}
class Circle implements Shape {
    public void draw() {
        //...
    }
}

//AbstractB
interface Color {
    public void fill();
}
class Red implements Color {
    public void fill() {
        //...
    }
}
class Green implements Color {
    public void fill() {
        //...
    }
}
```



```
//AbstractFactory
interface AbstractFactory {
    public Color getColor(String color);
    public Shape getShape(String shape);
}

class ShapeFactory extends AbstractFactory {
    public Shape getShape(String in){
        switch (in) {
            case "CIRCLE": return new Circle();
            case "SQUARE": return new SQUARE();
        }
        return  null;
}

class ColorFactory extends AbstractFactory {
    public Color getColor(String in){
        switch (in) {
            case "RED": return new Red();
            case "GREEN": return new Green();
        }
        return  null;
    }
}
```



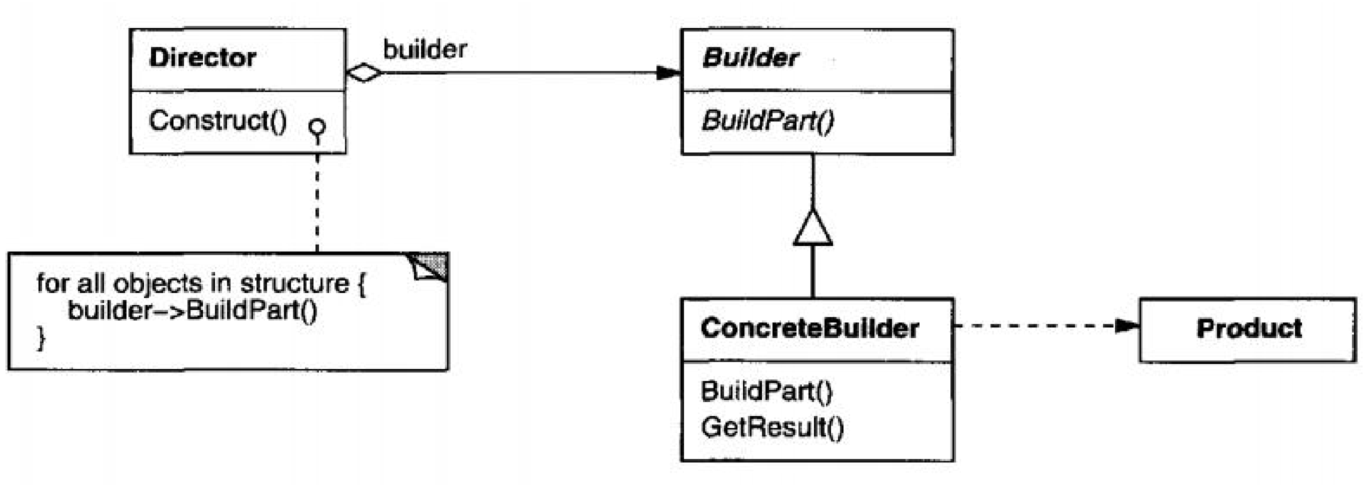
```
public class main {
   public static void main(String[] args) {
      AbstractFactory shape = new ShapeFactory();
      shape.getShape("CIRCLE").draw();
      shape.getShape("SQUARE").draw();
      AbstractFactory color = new ColorFactory();
      color.getColor("RED").fill();
      color.getColor("Green").fill();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

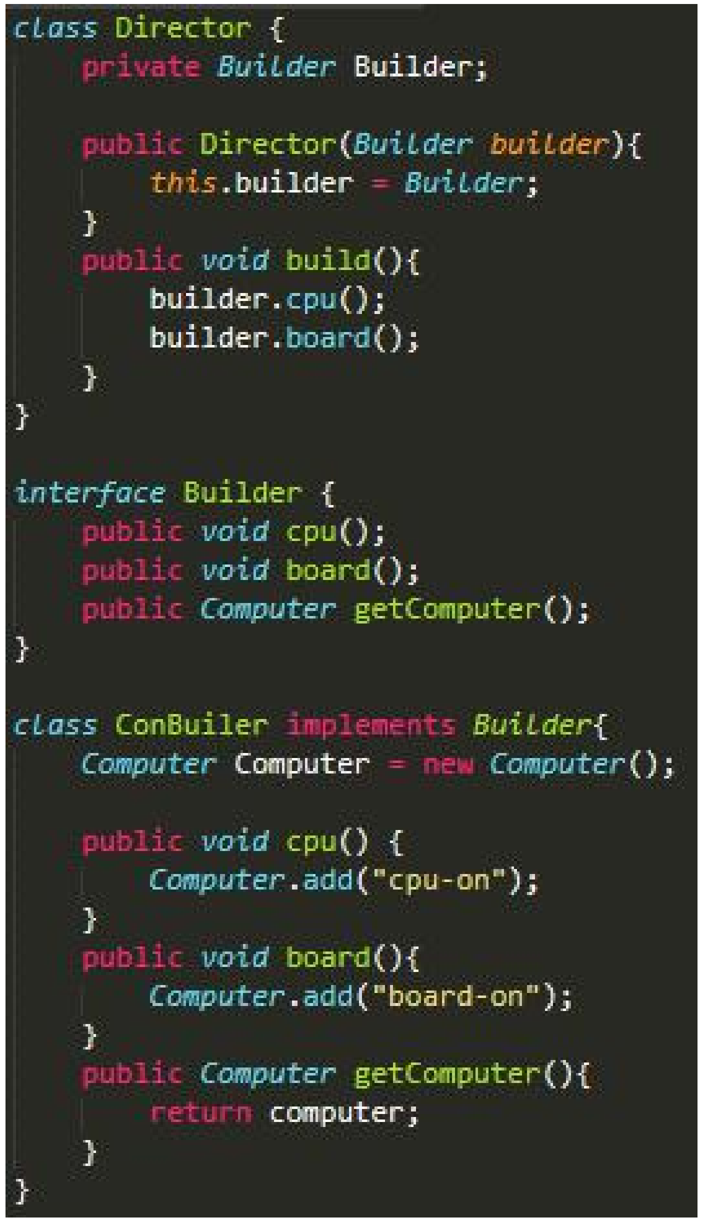
Builder 建造者模式 (AEP)

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

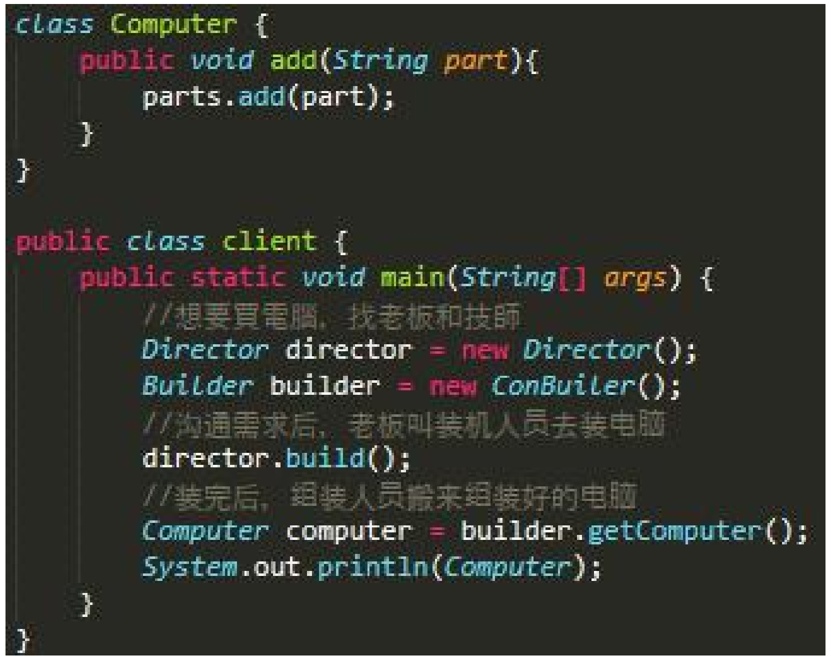
將複雜對象的構造與其表示分開，以便將其複制相同的施工過程可以創建不同的表示。
(主要解决在软件系统中，創造 “复杂对象” ，其通常由各个部分的子对象用一定的算法构成)

情境：我去找老闆和技師，告訴 Director 下個禮拜要電腦 (When)
老闆知道了請 Builder 技師去做(How)，技師創造出了Product





```java
class Director {
    private Builder Builder;

    public Director(Builder builder){
        this.builder = Builder;
    }
    public void build(){
        builder.cpu();
        builder.board();
    }
}

interface Builder {
    public void cpu();
    public void board();
    public Computer getComputer();
}

class ConBuiler implements Builder{
    Computer Computer = new Computer();

    public void cpu() {
        Computer.add("cpu-on");
    }
    public void board(){
        Computer.add("board-on");
    }
    public Computer getComputer(){
        return computer;
    }
}
```

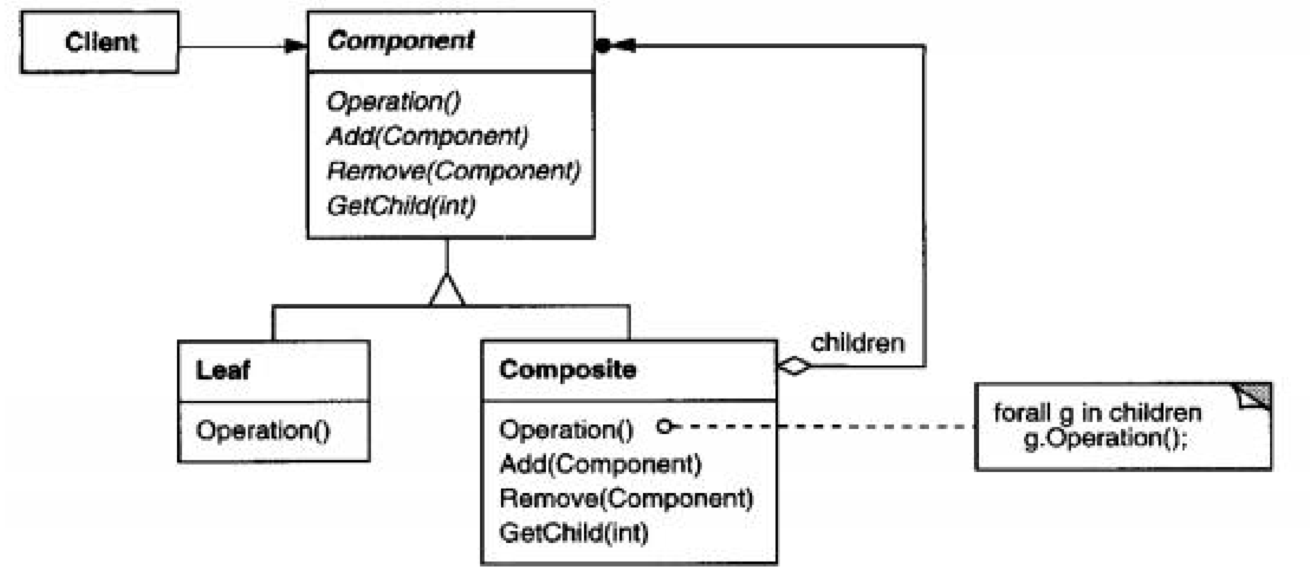


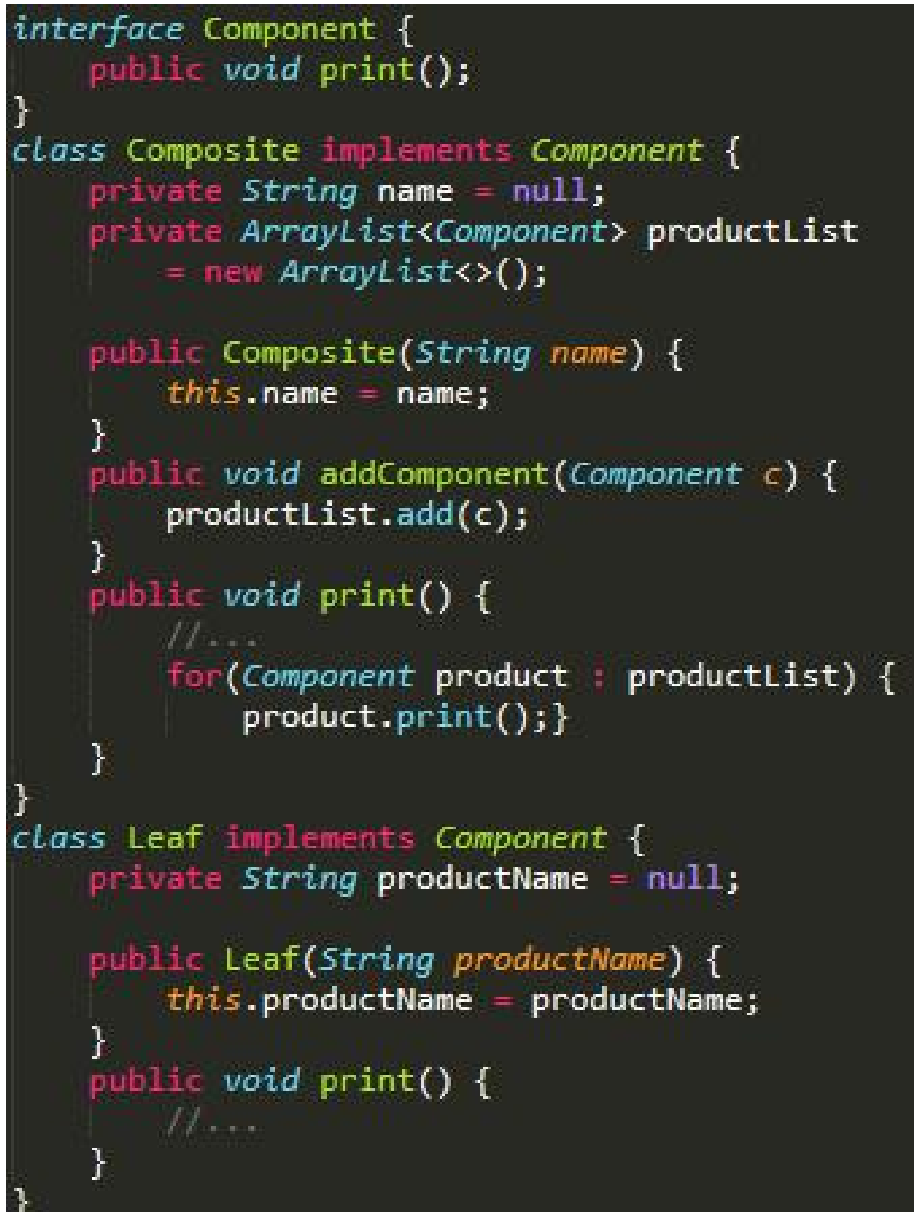
```java
class Computer {
    public void add(String part){
        parts.add(part);
    }
}

public class client {
    public static void main(String[] args) {
        //想要買電腦，找老板和技師
        Director director = new Director();
        Builder builder = new ConBuiler();
        //沟通需求后，老板叫装机人员去装电脑
        director.build();
        //装完后，组装人员搬来组装好的电脑
        Computer computer = builder.getComputer();
        System.out.println(Computer);
    }
}
```

Composite 組合模式 (EP)

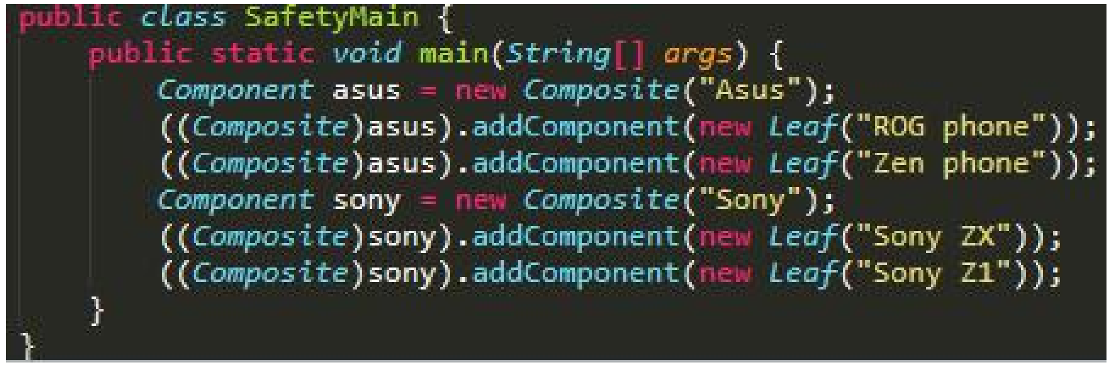
Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

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。
(客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。)





```java
interface Component {
    public void print();
}
class Composite implements Component {
    private String name = null;
    private ArrayList<Component> productList
        = new ArrayList<>();

    public Composite(String name) {
        this.name = name;
    }
    public void addComponent(Component c) {
        productList.add(c);
    }
    public void print() {
        //...
        for(Component product : productList) {
            product.print();}
    }
}
class Leaf implements Component {
    private String productName = null;

    public Leaf(String productName) {
        this.productName = productName;
    }
    public void print() {
        //...
    }
}
```



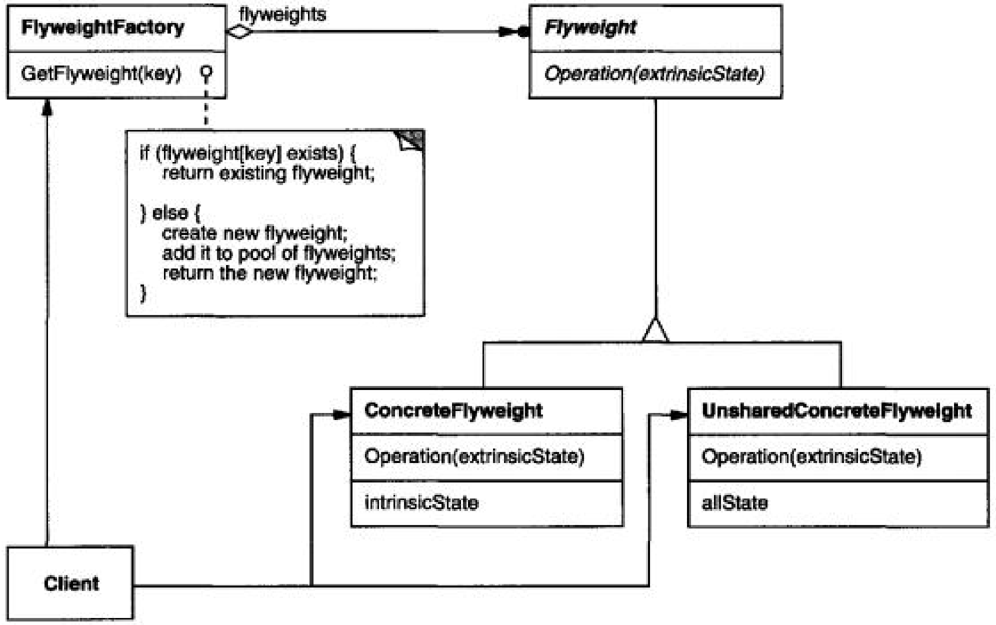
```java
public class SafetyMain {
    public static void main(String[] args) {
        Component asus = new Composite("Asus");
        ((Composite)asus).addComponent(new Leaf("ROG phone"));
        ((Composite)asus).addComponent(new Leaf("Zen phone"));
        Component sony = new Composite("Sony");
        ((Composite)sony).addComponent(new Leaf("Sony ZX"));
        ((Composite)sony).addComponent(new Leaf("Sony Z1"));
    }
}
```

Flyweight 享元模式 (P)

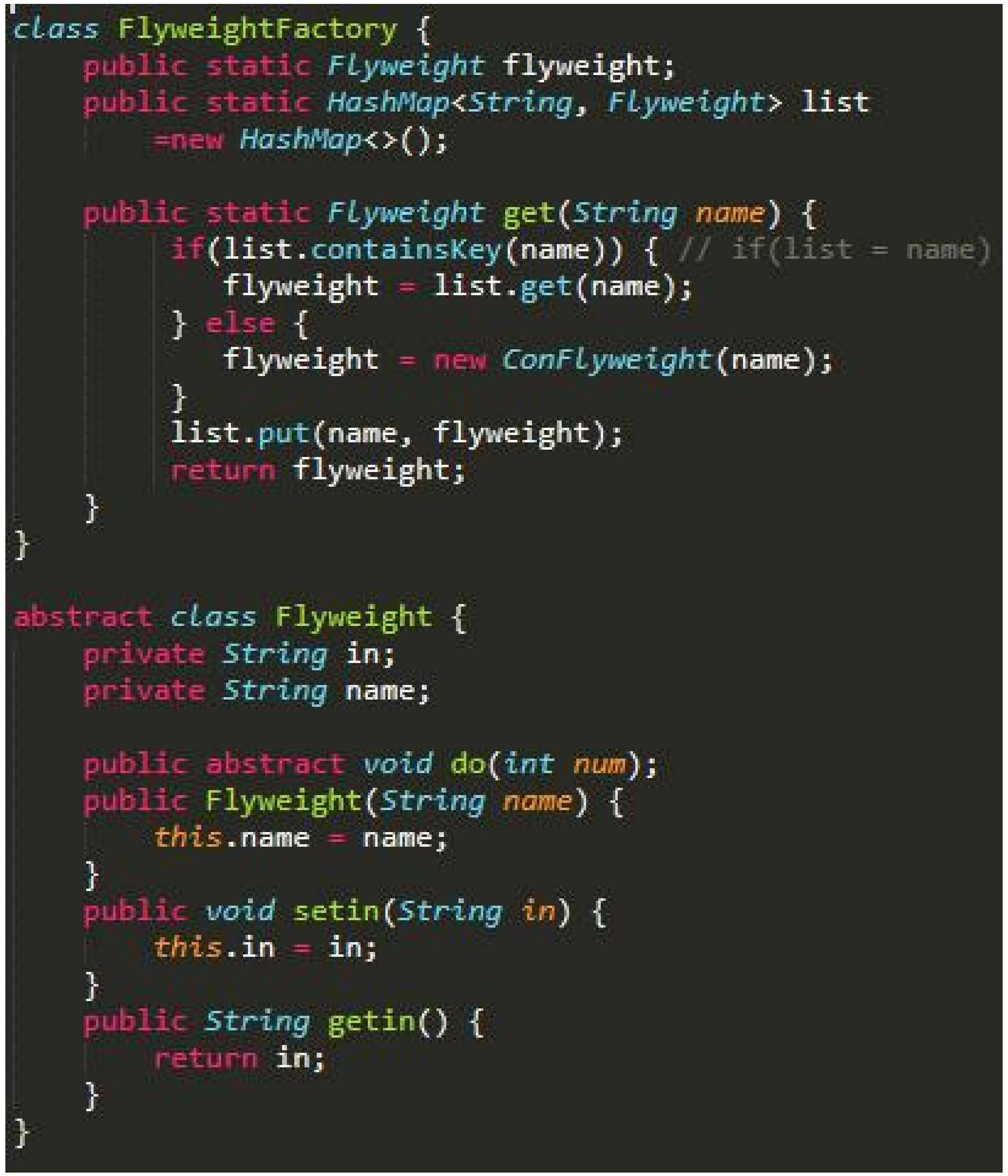
Use sharing to support large numbers of fine-grained objects efficiently.

使用共享可以有效地支持大量細粒度對象。
(在有大量对象时，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。)

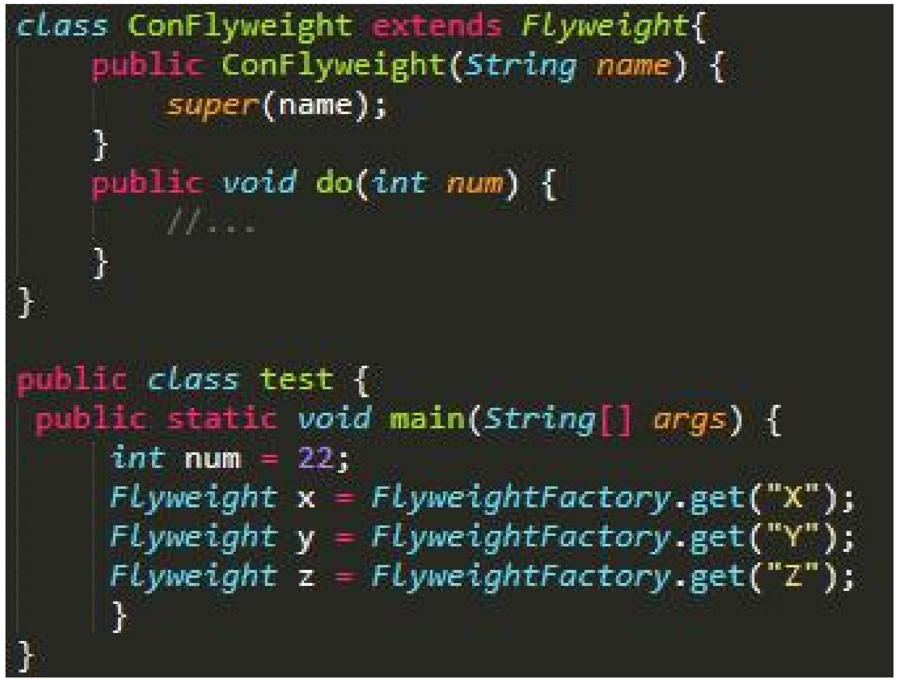
Intrinsic: 可被共享的
Extrinsic:不被共享的


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;
}
```



```
class FlyweightFactory {
    public static Flyweight flyweight;
    public static HashMap<String, Flyweight> list
        =new HashMap<>();

    public static Flyweight get(String name) {
        if(list.containsKey(name)) { // if(list = name)
            flyweight = list.get(name);
        } else {
            flyweight = new ConFlyweight(name);
        }
        list.put(name, flyweight);
        return flyweight;
    }
}

abstract class Flyweight {
    private String in;
    private String name;

    public abstract void do(int num);
    public Flyweight(String name) {
        this.name = name;
    }
    public void setin(String in) {
        this.in = in;
    }
    public String getin() {
        return in;
    }
}
```

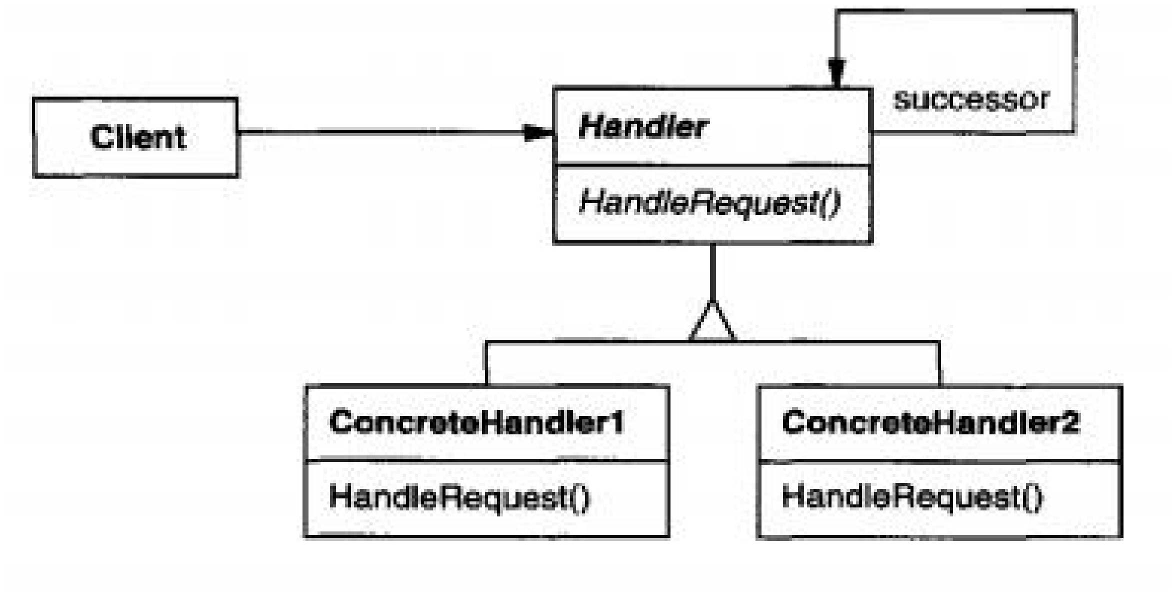


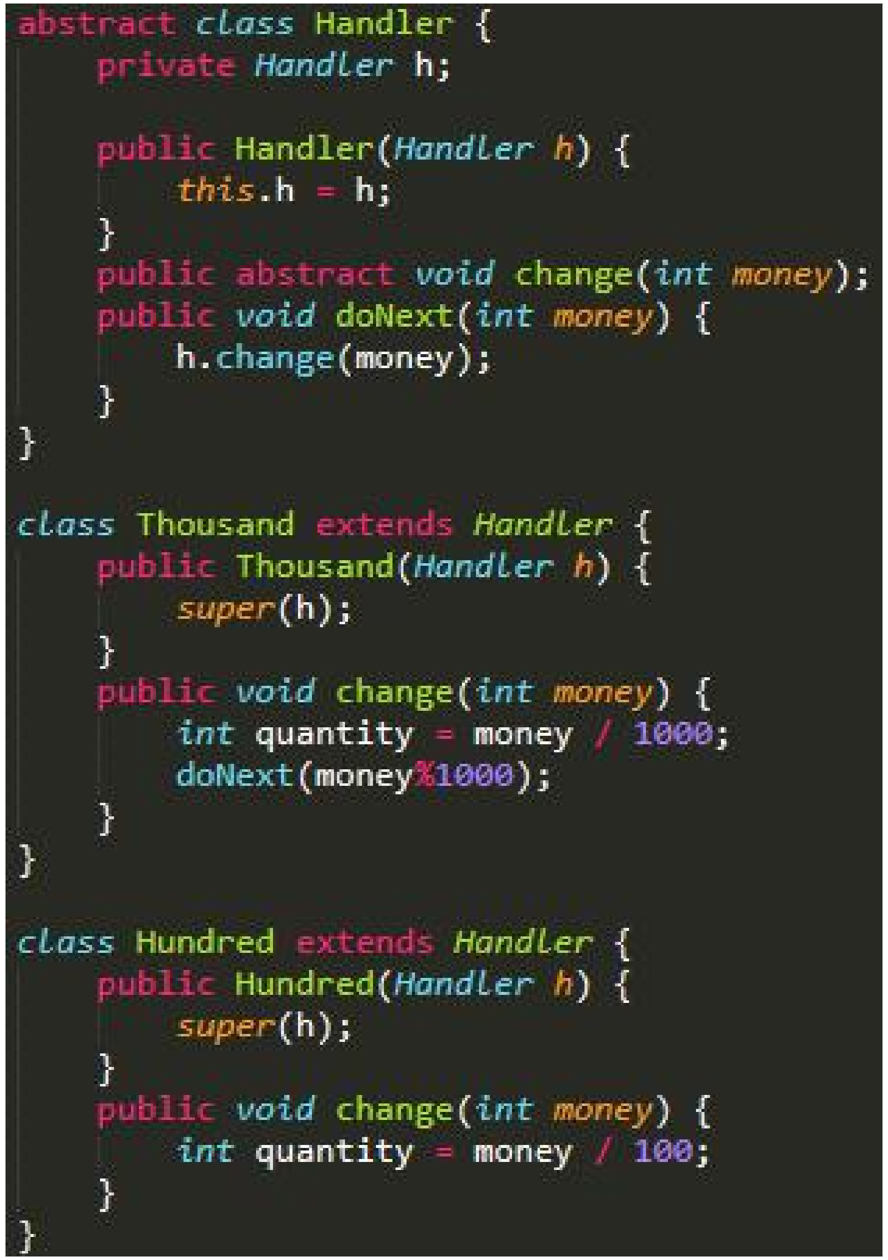
```
class ConFlyweight extends Flyweight{
    public ConFlyweight(String name) {
        super(name);
    }
    public void do(int num) {
        //...
    }
}

public class test {
 public static void main(String[] args) {
     int num = 22;
     Flyweight x = FlyweightFactory.get("X");
     Flyweight y = FlyweightFactory.get("Y");
     Flyweight z = FlyweightFactory.get("Z");
     }
}
```

Chain of Res 責任鏈模式 (EP)

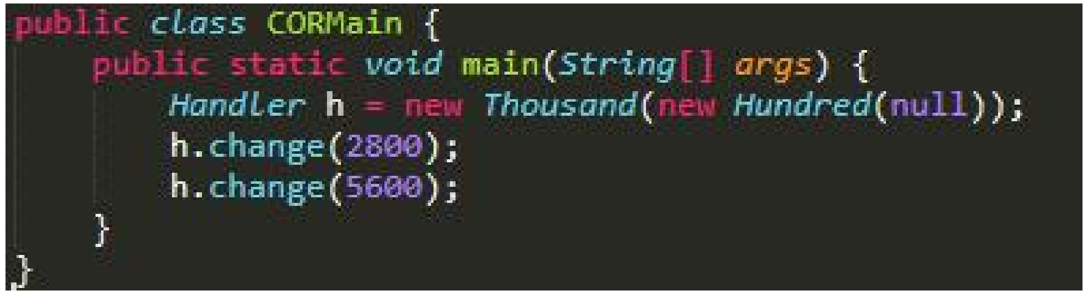
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

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。
(处理者负责处理请求，客户只需要将请求发送到职责链上即可，将请求的发送者和请求的处理者解耦。)





```
abstract class Handler {
    private Handler h;

    public Handler(Handler h) {
        this.h = h;
    }
    public abstract void change(int money);
    public void doNext(int money) {
        h.change(money);
    }
}

class Thousand extends Handler {
    public Thousand(Handler h) {
        super(h);
    }
    public void change(int money) {
        int quantity = money / 1000;
        doNext(money%1000);
    }
}

class Hundred extends Handler {
    public Hundred(Handler h) {
        super(h);
    }
    public void change(int money) {
        int quantity = money / 100;
    }
}
```

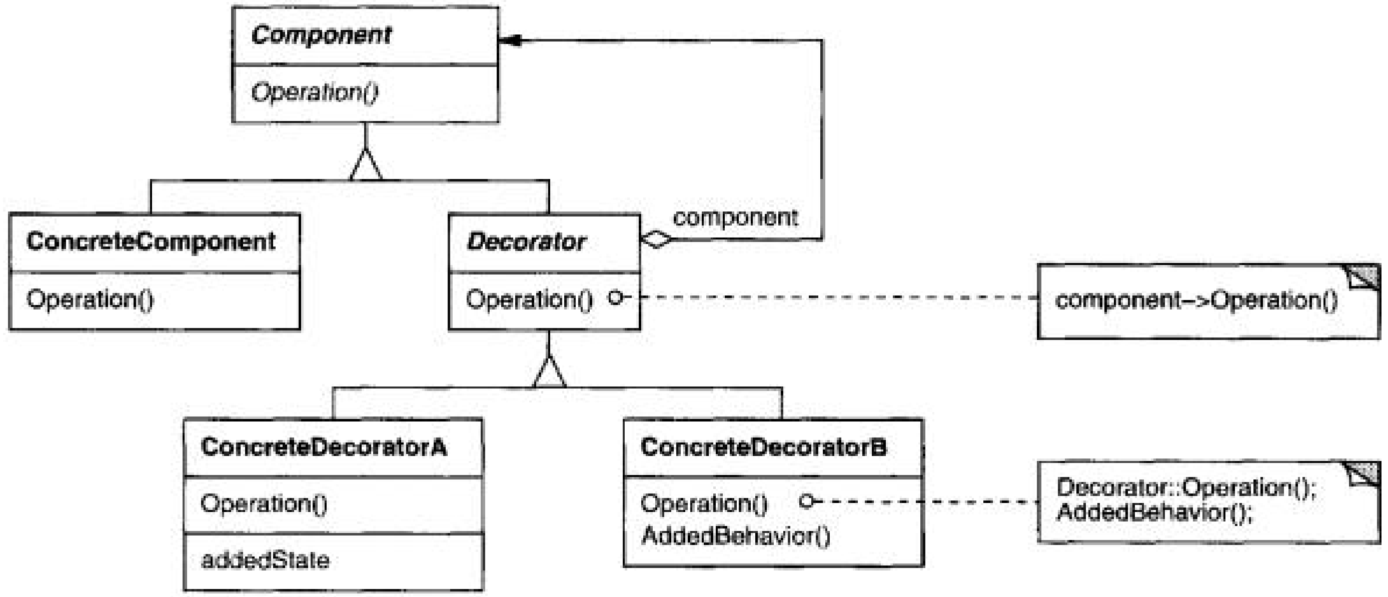


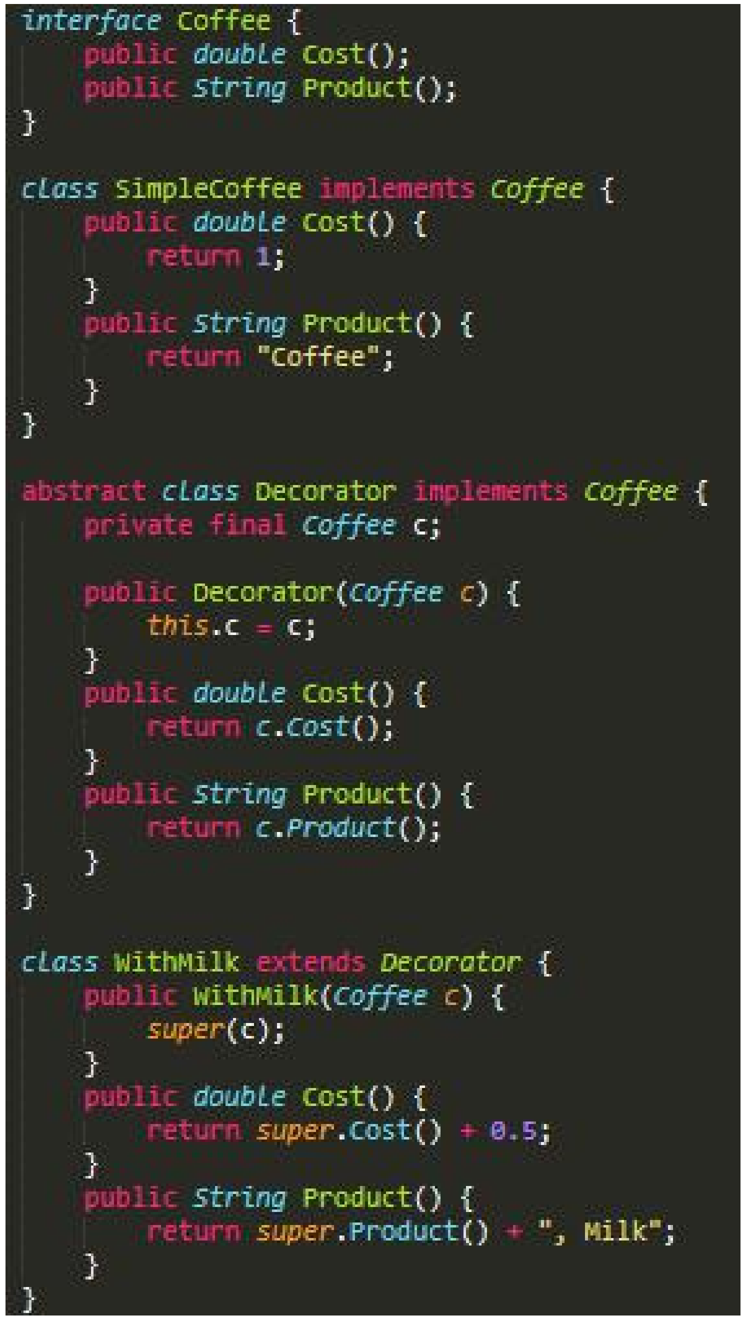
```
public class CORMain {
    public static void main(String[] args) {
        Handler h = new Thousand(new Hundred(null));
        h.change(2800);
        h.change(5600);
    }
}
```

Decorator 裝飾模式 (EP)

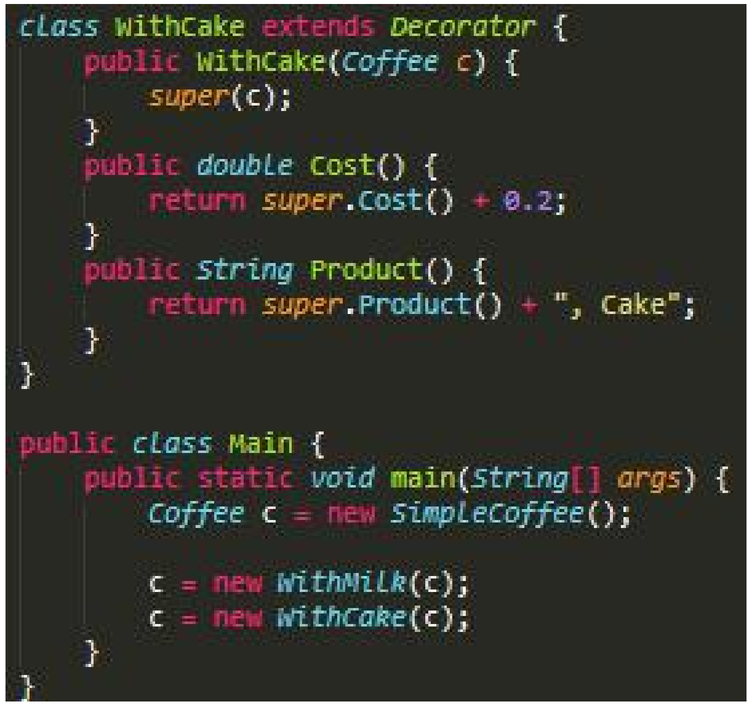
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

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。
(一般的扩展一个类经常使用继承方式实现，由于继承，子类会很膨胀。可在不想增加很多子类的情况下扩展类。)
跟Composite相似的結構 重點在可以一層一層的疊上，容易擴充而不用修改





```java
interface Coffee {
    public double Cost();
    public String Product();
}

class SimpleCoffee implements Coffee {
    public double Cost() {
        return 1;
    }
    public String Product() {
        return "Coffee";
    }
}

abstract class Decorator implements Coffee {
    private final Coffee c;

    public Decorator(Coffee c) {
        this.c = c;
    }
    public double Cost() {
        return c.Cost();
    }
    public String Product() {
        return c.Product();
    }
}

class WithMilk extends Decorator {
    public WithMilk(Coffee c) {
        super(c);
    }
    public double Cost() {
        return super.Cost() + 0.5;
    }
    public String Product() {
        return super.Product() + ", Milk";
    }
}
```



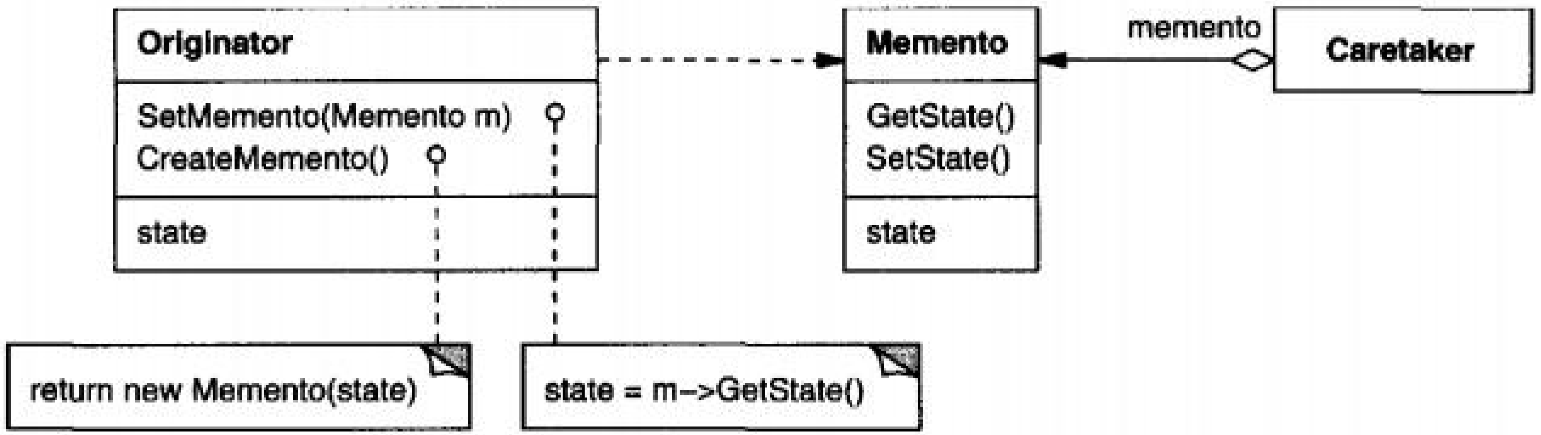
```java
class WithCake extends Decorator {
    public WithCake(Coffee c) {
        super(c);
    }
    public double Cost() {
        return super.Cost() + 0.2;
    }
    public String Product() {
        return super.Product() + ", Cake";
    }
}

public class Main {
    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();

        c = new WithMilk(c);
        c = new WithCake(c);
    }
}
```

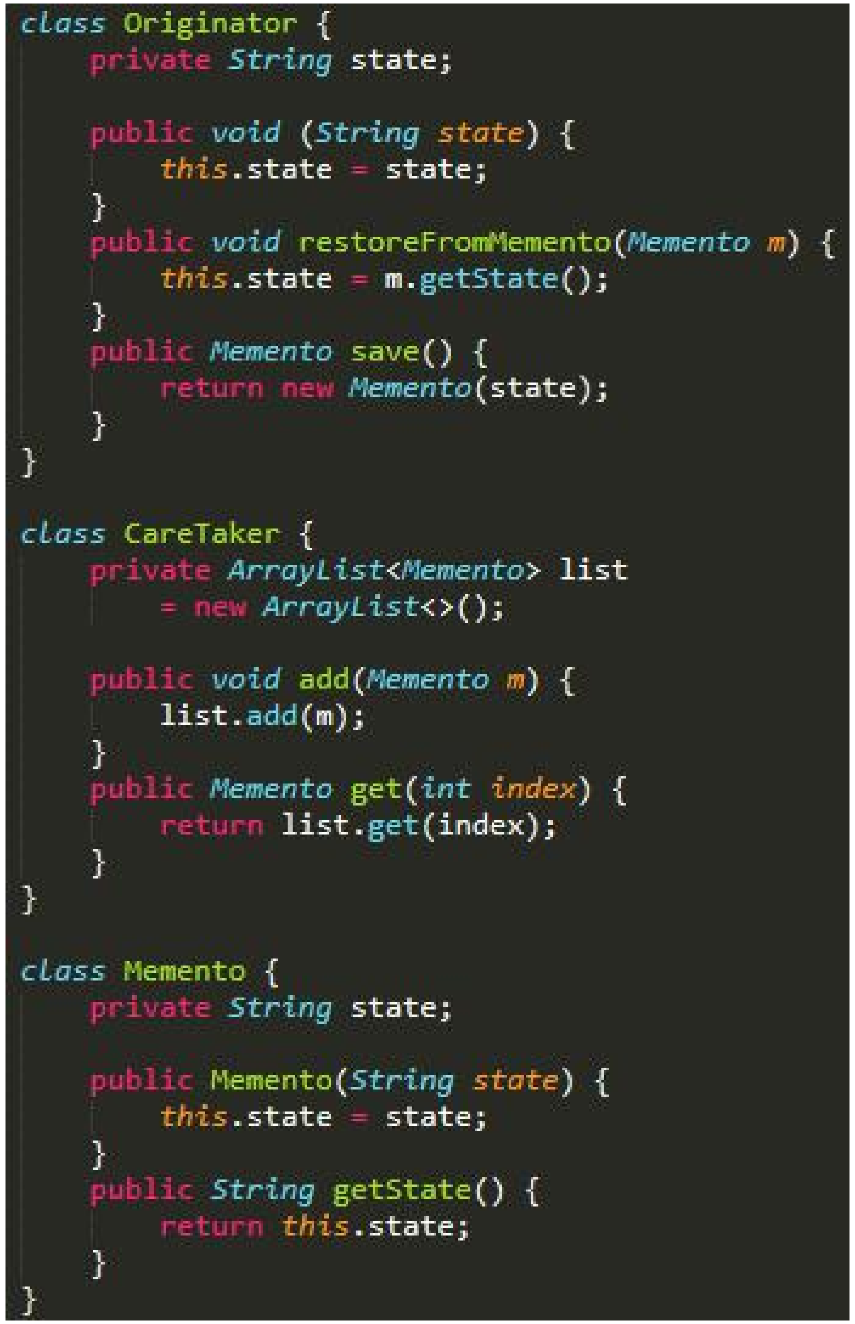
Memento 備忘錄模式 (EP)

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

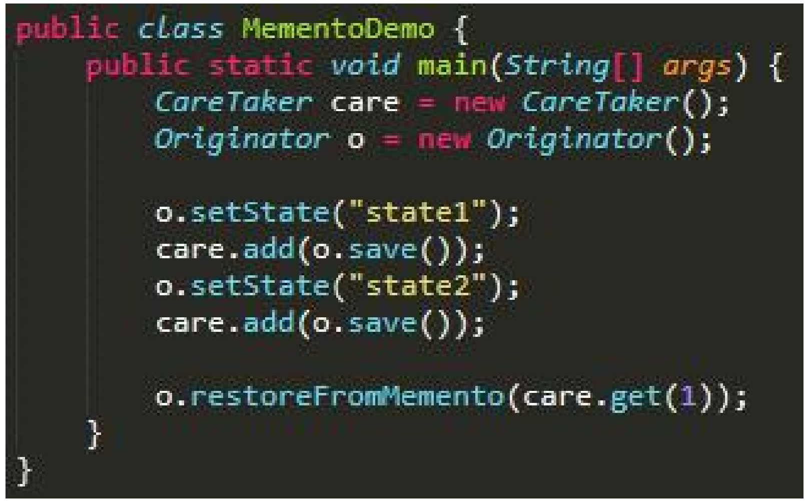
在不違反封裝的情況下，捕獲並外化對象的內部狀態，以便稍後可以將對象恢復到此狀態。
(不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。)

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





```java
class Originator {
    private String state;

    public void (String state) {
        this.state = state;
    }
    public void restoreFromMemento(Memento m) {
        this.state = m.getState();
    }
    public Memento save() {
        return new Memento(state);
    }
}

class CareTaker {
    private ArrayList<Memento> list
        = new ArrayList<>();

    public void add(Memento m) {
        list.add(m);
    }
    public Memento get(int index) {
        return list.get(index);
    }
}

class Memento {
    private String state;

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

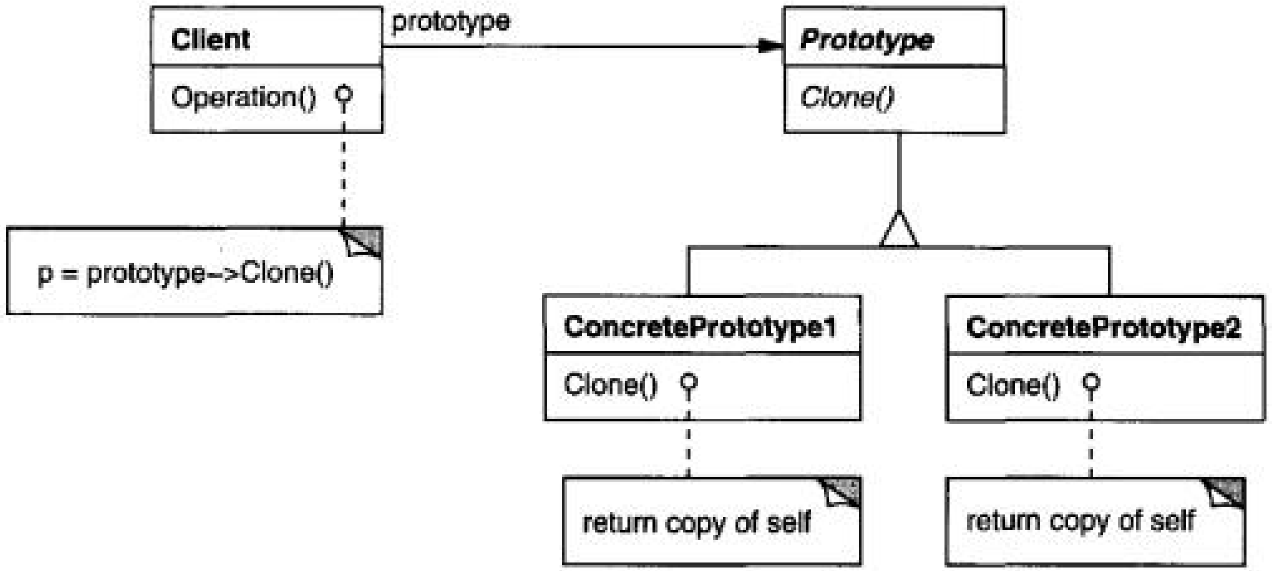


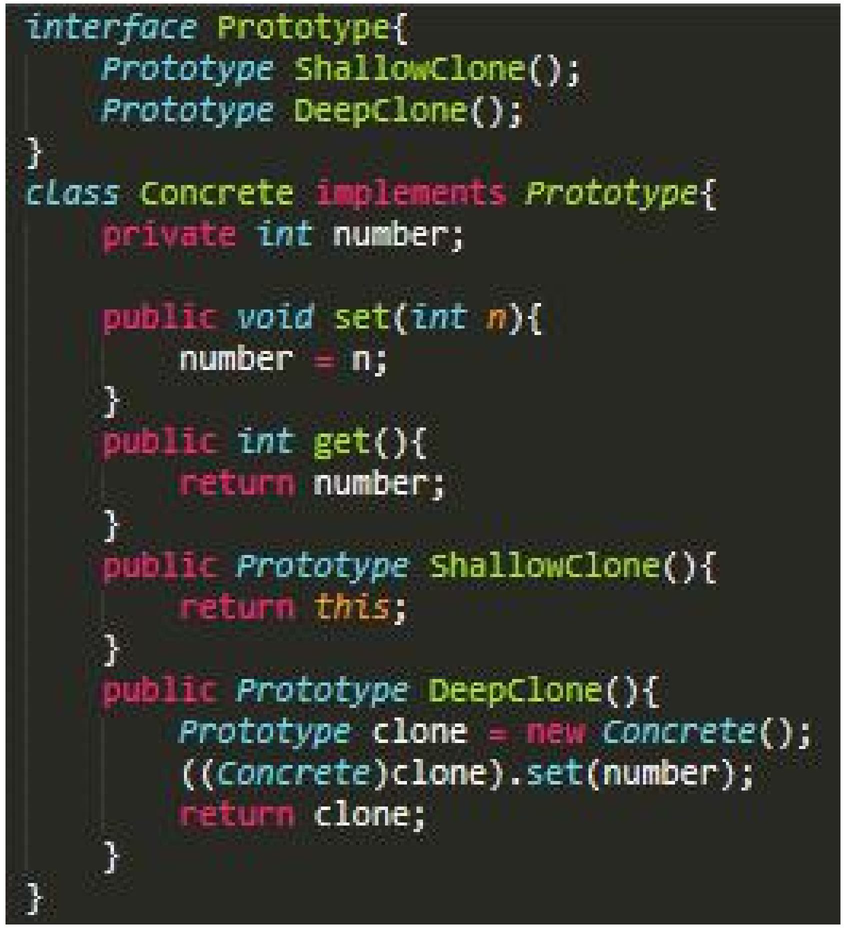
```java
public class MementoDemo {
    public static void main(String[] args) {
        CareTaker care = new CareTaker();
        Originator o = new Originator();

        o.setState("state1");
        care.add(o.save());
        o.setState("state2");
        care.add(o.save());

        o.restoreFromMemento(care.get(1));
    }
}
```

Prototype 原型模式 (EP)

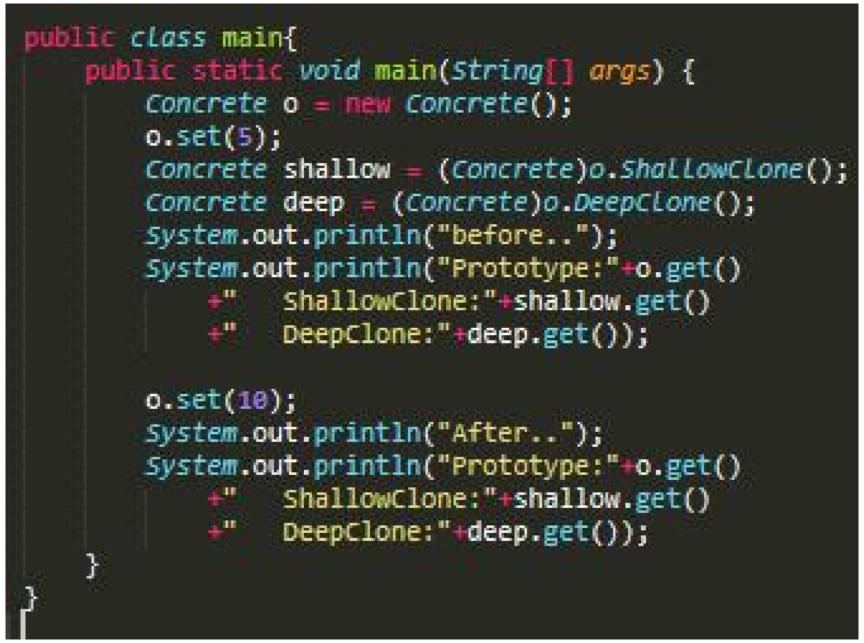
Specify the kinds of objects to create using a prototypical instance,
and create new objects by copying this prototype.

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。
(利用已有的一个原型对象，快速地生成和原型对象一样的实例。)
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





```
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;
    }
}
```



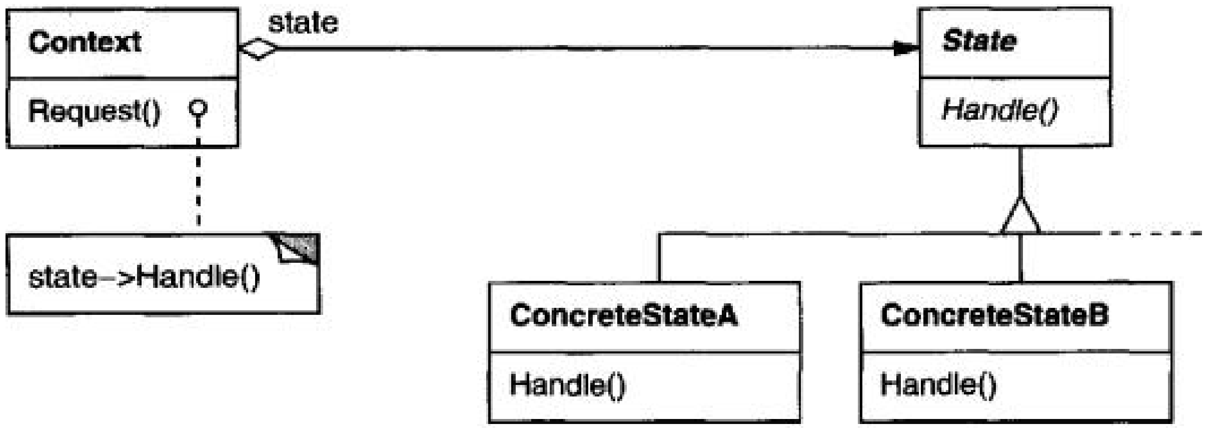
```
public class main{
    public static void main(String[] args) {
        Concrete o = new Concrete();
        o.set(5);
        Concrete shallow = (Concrete)o.ShallowClone();
        Concrete deep = (Concrete)o.DeepClone();
        System.out.println("before..");
        System.out.println("Prototype:"+o.get()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

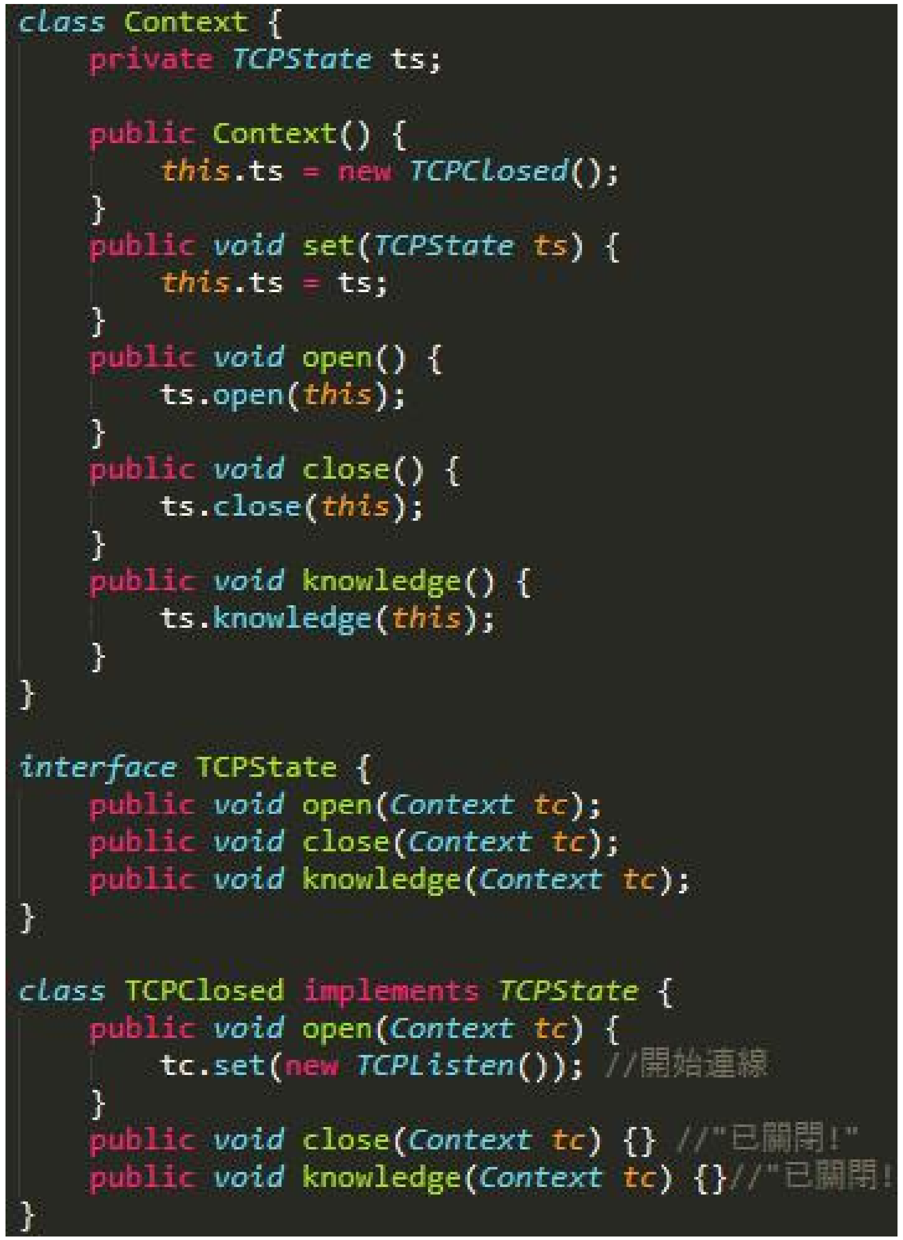
State 狀態模式 (EP)

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

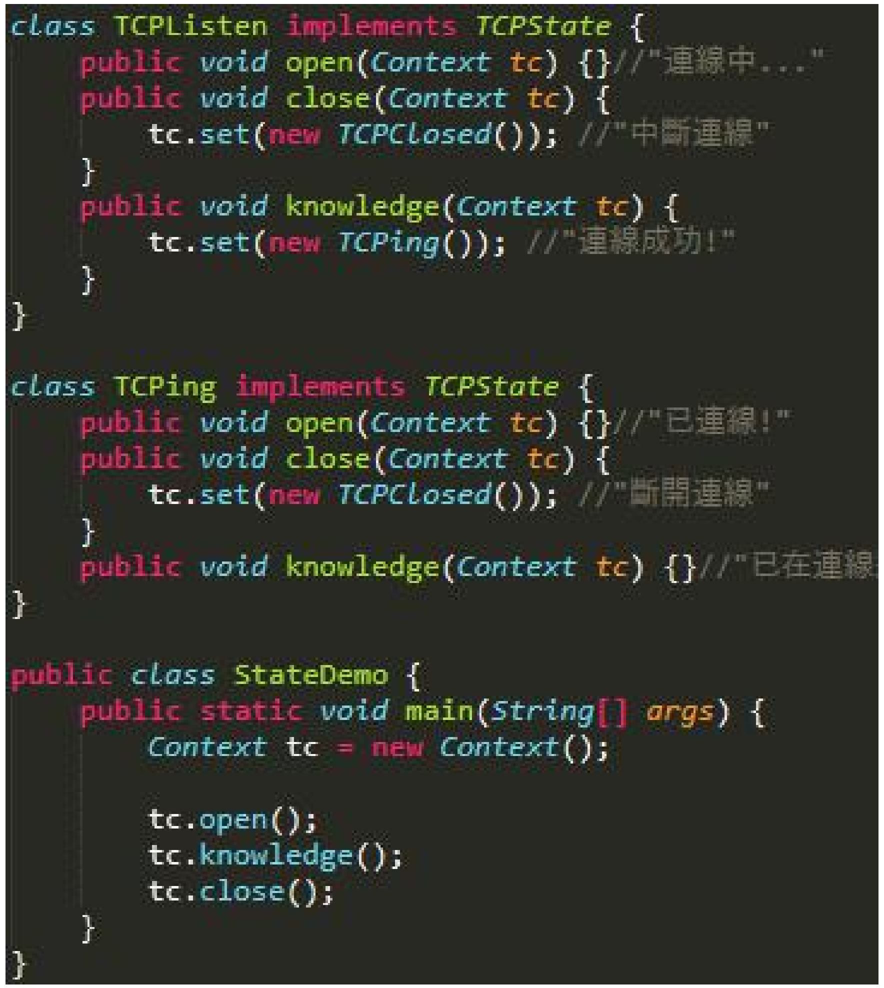
允許對像在其內部狀態更改時，更改其行為。該對象看起來好像更改了他的類。
(可以根据它的状态改变而改变它的相关行为。)

自動販賣機(物件)，會因有沒有投錢(狀態)有不同反應。
「結構跟Strategy一模一樣 不過目的不一樣」
- State 由自己轉變到下一個State
- Strategy 由使用者決定切換到哪一個方法





```java
class Context {
    private TCPState ts;

    public Context() {
        this.ts = new TCPClosed();
    }
    public void set(TCPState ts) {
        this.ts = ts;
    }
    public void open() {
        ts.open(this);
    }
    public void close() {
        ts.close(this);
    }
    public void knowledge() {
        ts.knowledge(this);
    }
}

interface TCPState {
    public void open(Context tc);
    public void close(Context tc);
    public void knowledge(Context tc);
}

class TCPClosed implements TCPState {
    public void open(Context tc) {
        tc.set(new TCPListen()); //開始連線
    }
    public void close(Context tc) {} //"已關閉!"
    public void knowledge(Context tc) {}//"已關閉!
}
```



```java
class TCPListen implements TCPState {
    public void open(Context tc) {}//"連線中..."
    public void close(Context tc) {
        tc.set(new TCPClosed()); //"中斷連線"
    }
    public void knowledge(Context tc) {
        tc.set(new TCPing()); //"連線成功!"
    }
}

class TCPing implements TCPState {
    public void open(Context tc) {}//"已連線!"
    public void close(Context tc) {
        tc.set(new TCPClosed()); //"斷開連線"
    }
    public void knowledge(Context tc) {}//"已在連線
}

public class StateDemo {
    public static void main(String[] args) {
        Context tc = new Context();

        tc.open();
        tc.knowledge();
        tc.close();
    }
}
```

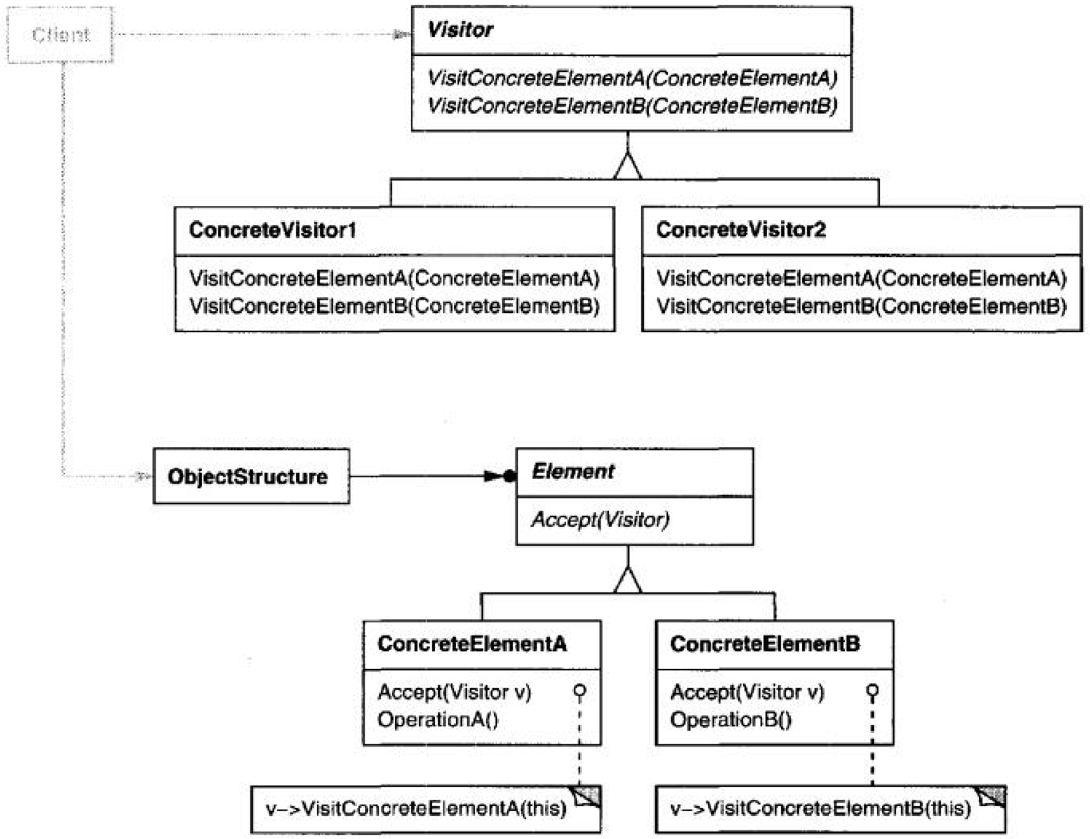
QUIZ2 考到這

Visitor 參訪者模式(EP)

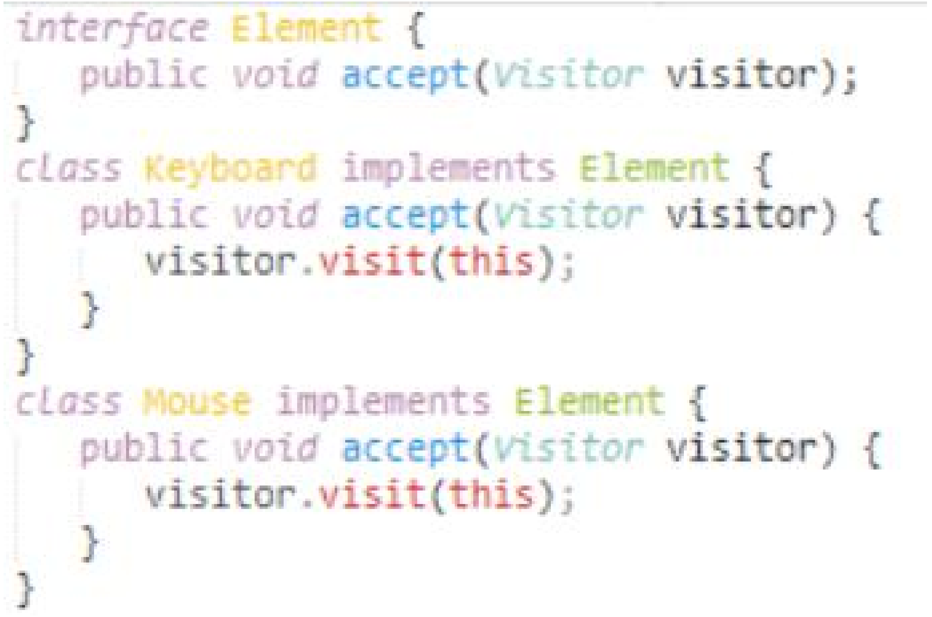
Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.

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。

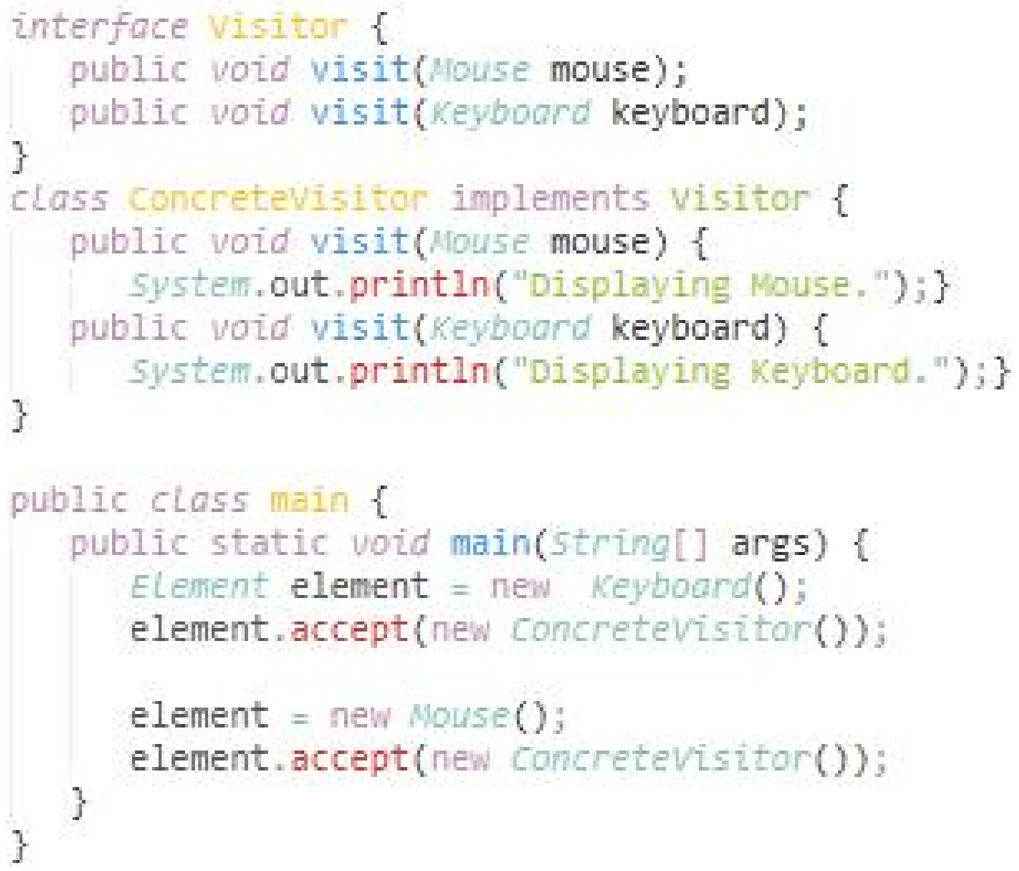
當你有很多元件(element)且數量固定
而這些元件常常需要被執行某些操作就可以使用Visitor
透過訪問者的方式來對這些元件進行操作


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)

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());
    }
}
```

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)**

Proxy 代理模式 (P)

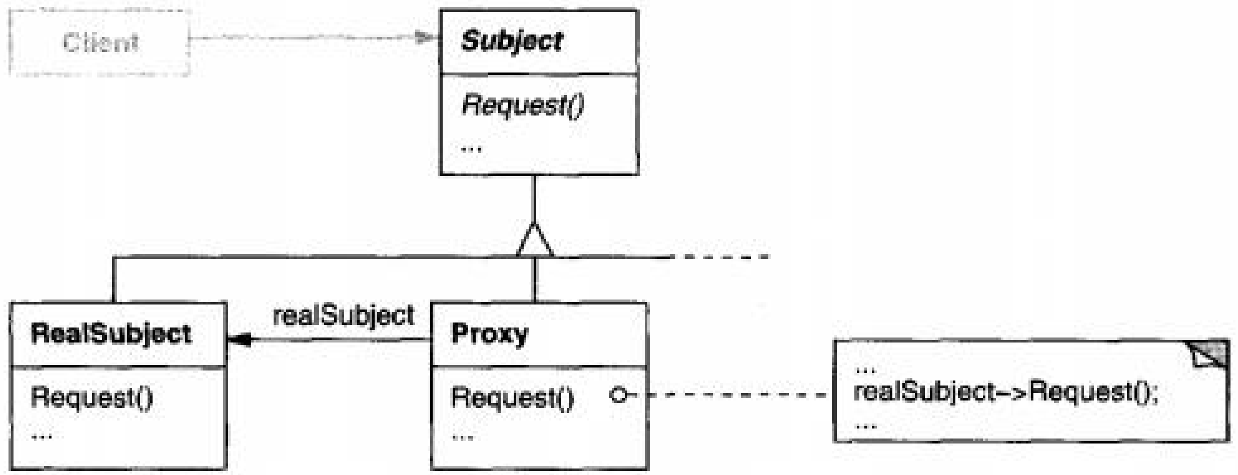
【Proxy vs Decorator】
相同：都需要實現同一個接口、繼承同一個抽像類。
代理：對整個對象的行為控制和限制 (老闆請財務處理事務)
裝飾：針對的是增加對象的職能 (老闆身上加，老闆還要懂財務會計)

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

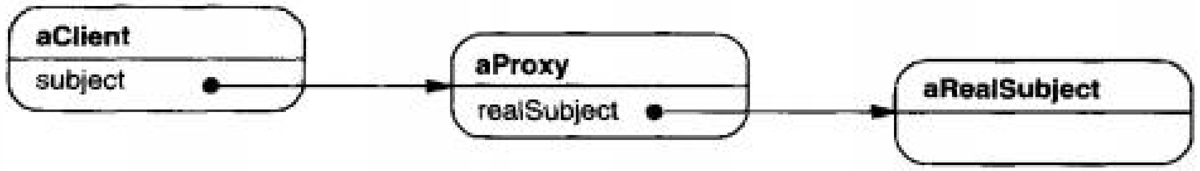
為另一個對象提供代理或占位符以控制對它的訪問。

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

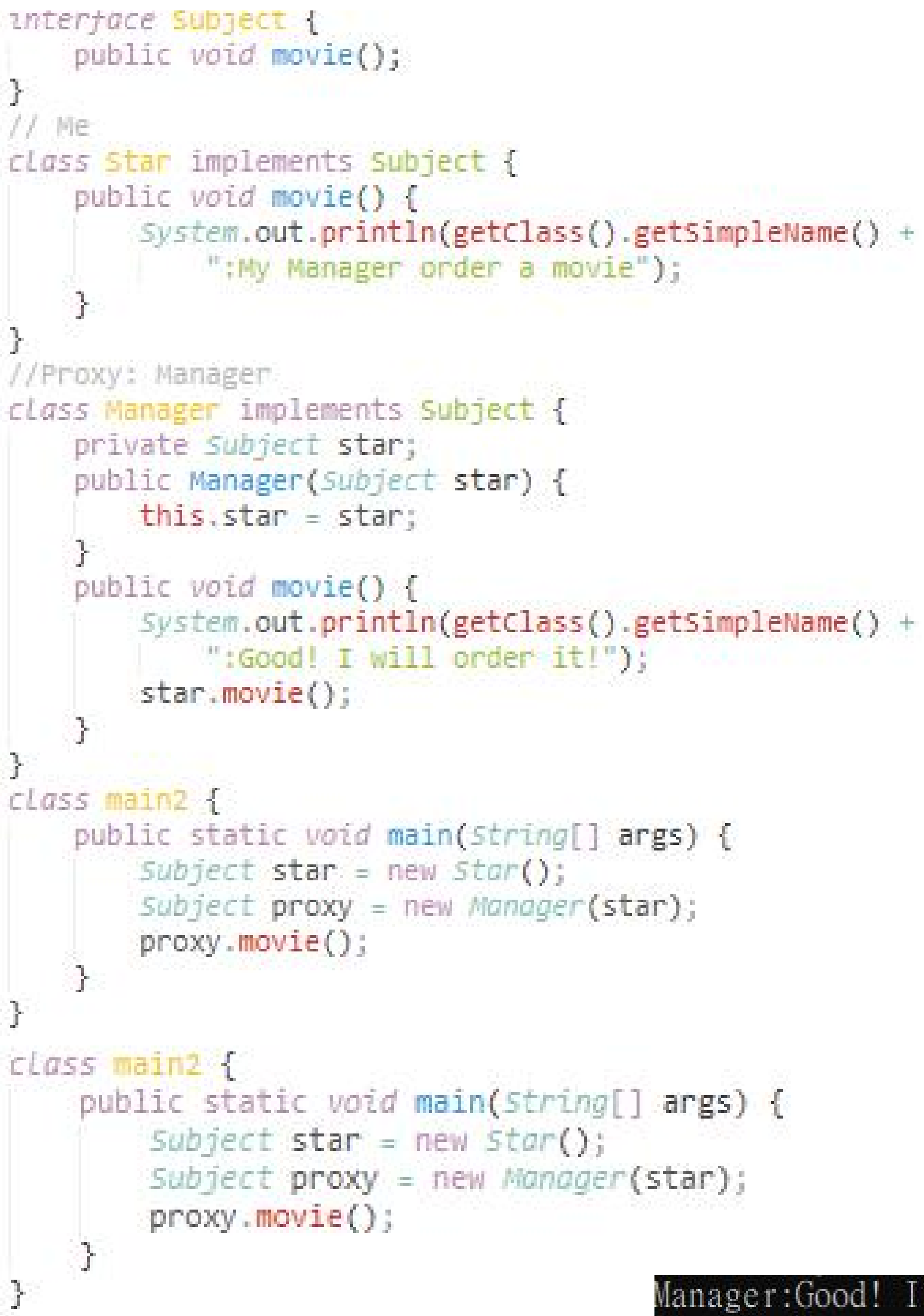
- 虛擬(Virtual Proxy)
  用比較不消耗資源的代理物件來代替實際物件
  實際物件只有在真正需要才會被創造
- 遠程(Remote Proxy)
  本地端提供物件來存取遠端網址的物件
- 保護(Protect Proxy)
  限制其他程式存取權限
- 智能(Smart Reference Proxy)
  為被代理的物件增加一些動作



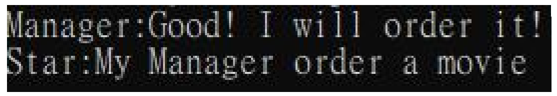
Here's a possible object diagram of a proxy structure at run-time:



範例1. 我的經紀人幫我接下電影

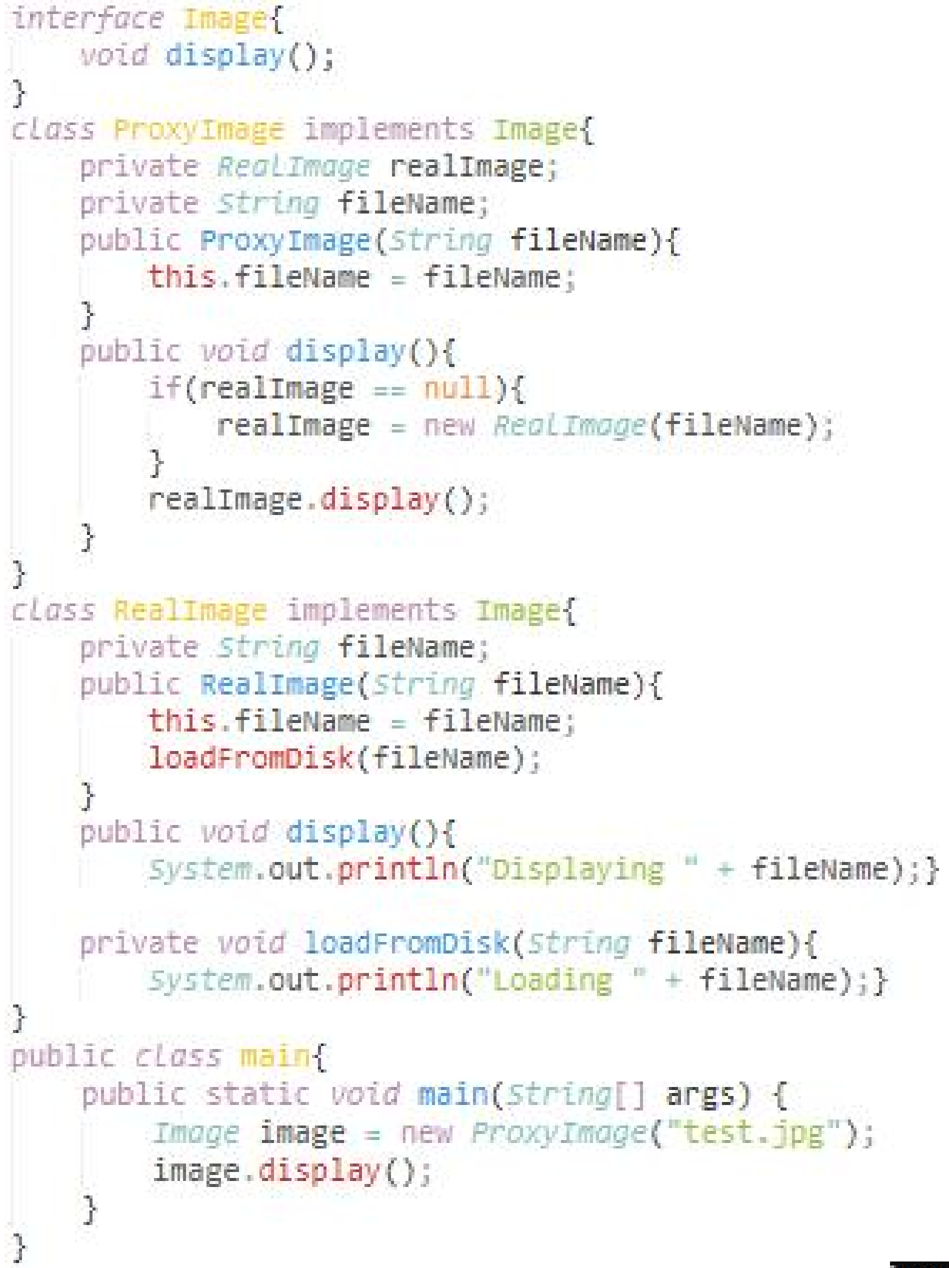


```java
interface Subject {
    public void movie();
}
// Me
class Star implements Subject {
    public void movie() {
        System.out.println(getClass().getSimpleName() +
            ":My Manager order a movie");
    }
}
//Proxy: Manager
class Manager implements Subject {
    private Subject star;
    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();
    }
}
```

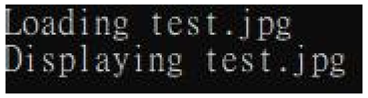


```
Manager:Good! I will order it!
Star:My Manager order a movie
```

範例2. 讀取照片



```java
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();
    }
}
```



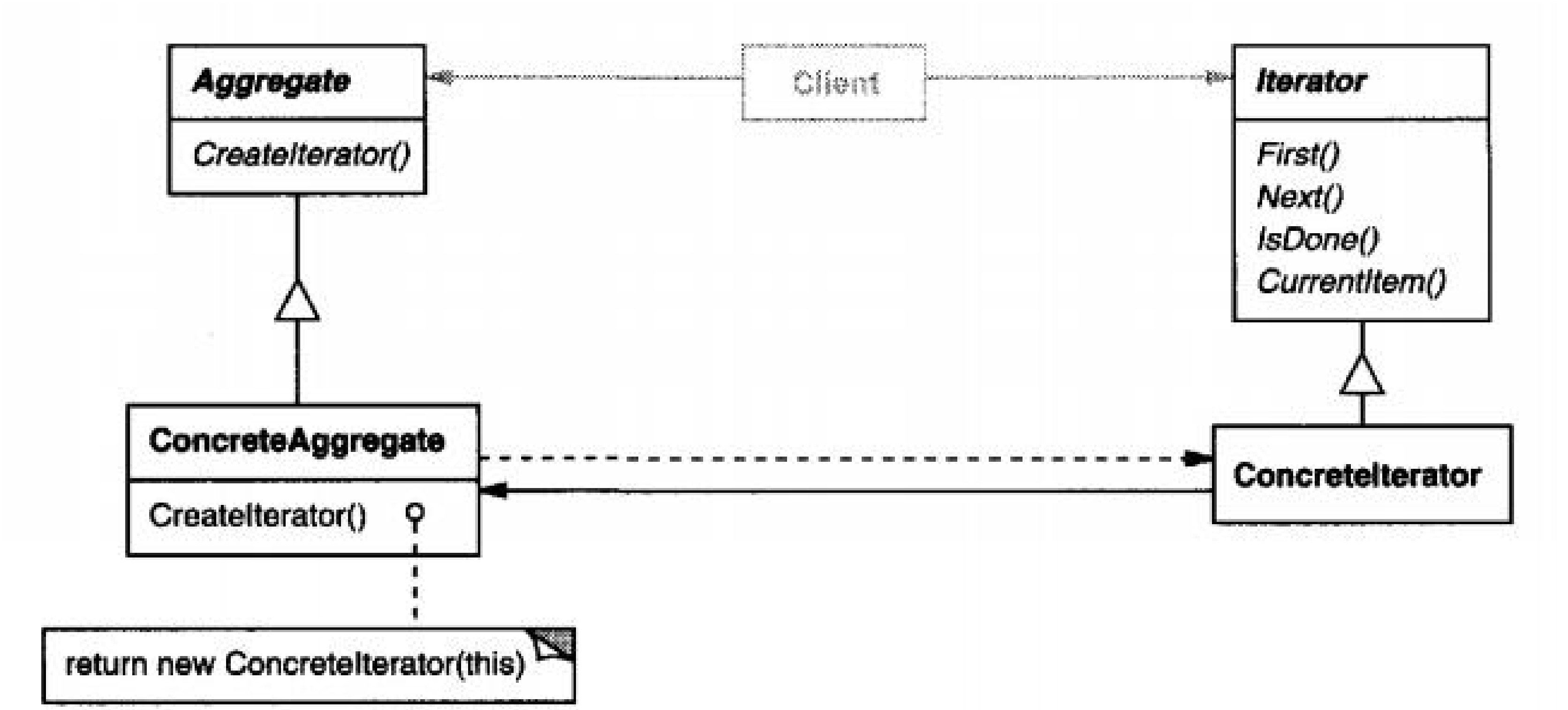
```
Loading test.jpg
Displaying test.jpg
```

Iterator 走訪器模式 (?)

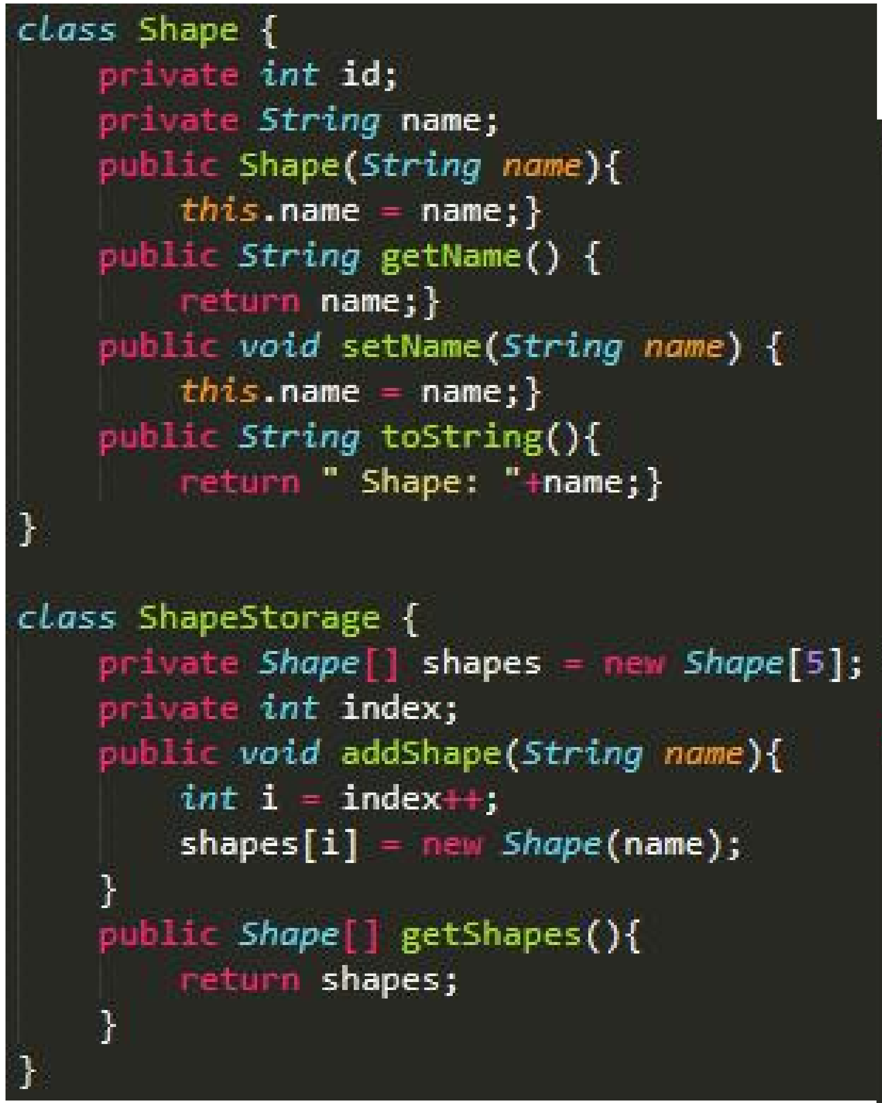
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

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

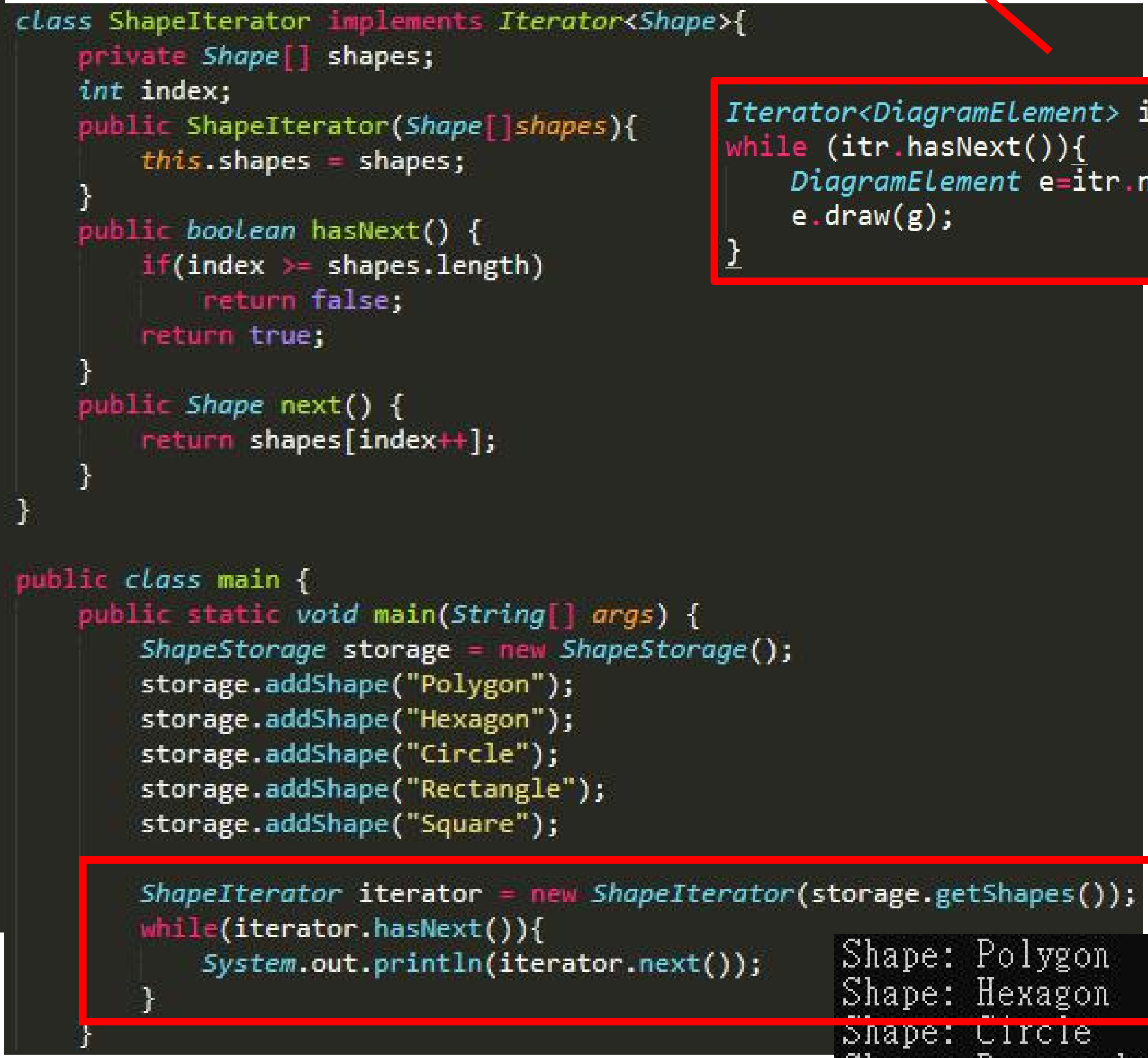


import java.util.Iterator;

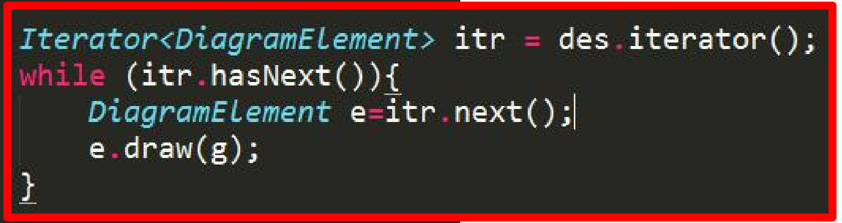


```java
class Shape {
    private int id;
    private String name;
    public Shape(String name){
        this.name = name;}
    public String getName() {
        return name;}
    public void setName(String name) {
        this.name = name;}
    public String toString(){
        return " Shape: "+name;}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

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;
    }
}
```

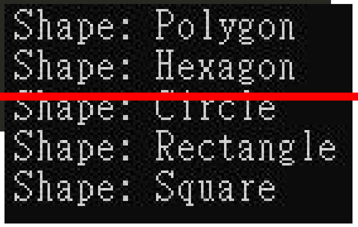
Java的Collection物件都有內建iterator()方法
直接拿來用吧..



```java
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());
        }
    }
```



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



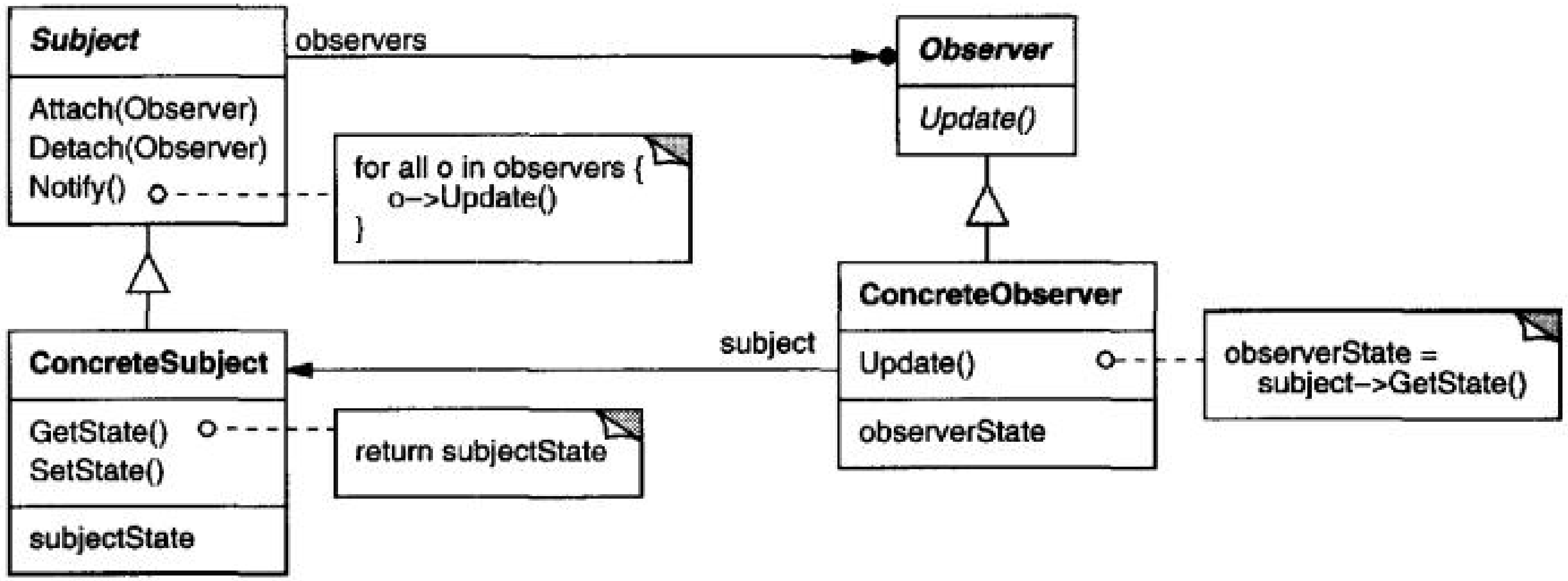
```
Shape: Polygon
Shape: Hexagon
Shape: Circle
Shape: Rectangle
Shape: Square
```

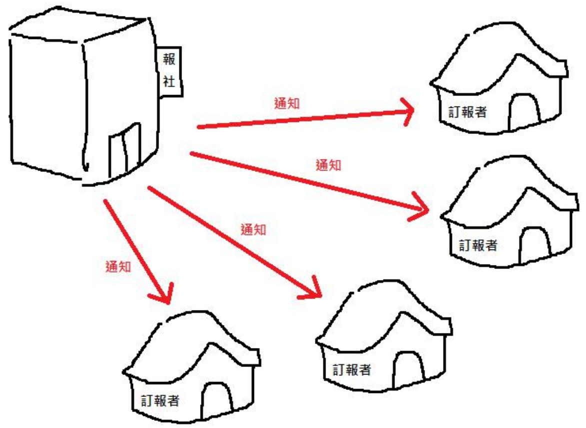
Observer 觀察者模式 (EP)

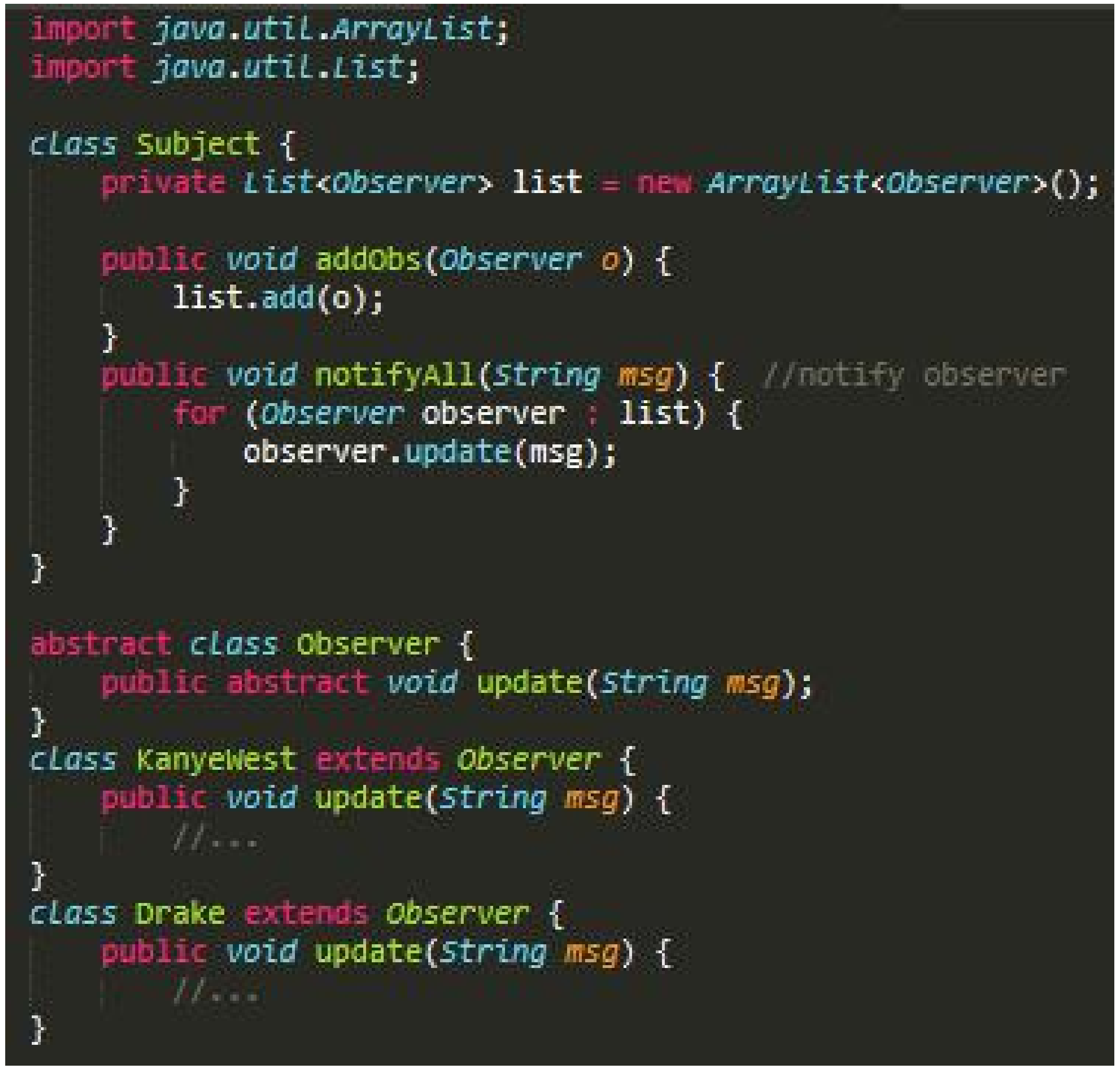
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

定義對象一對多關係，當一個對象改變狀態時，所有家屬都會自動得到通知和更新。
(一个对象的状态发生改变，所有的依赖对象都将得到通知，进行广播)

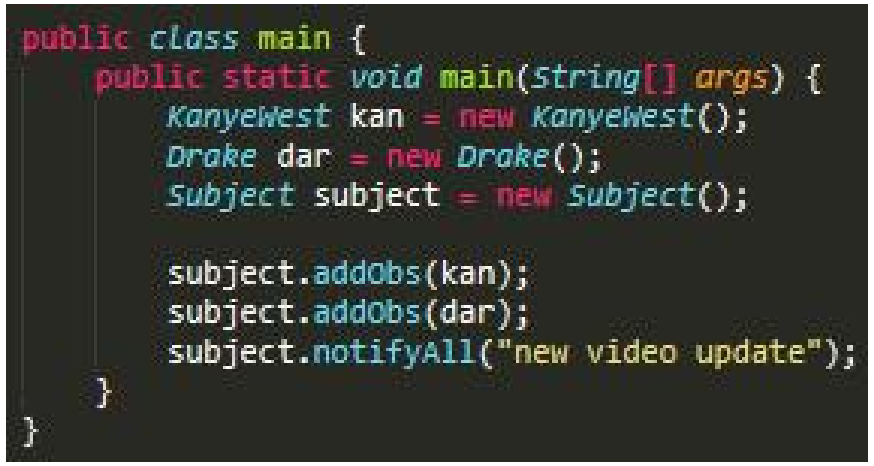
常跟 Proxy 對比：主要區別在功能不一樣
- Observer 強調被觀察者反饋結果
- Proxy 同根負責做同樣的事情







```java
import java.util.ArrayList;
import java.util.List;

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) {
        //...
}
class Drake extends Observer {
    public void update(String msg) {
        //...
}
```



```java
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");
    }
}
```

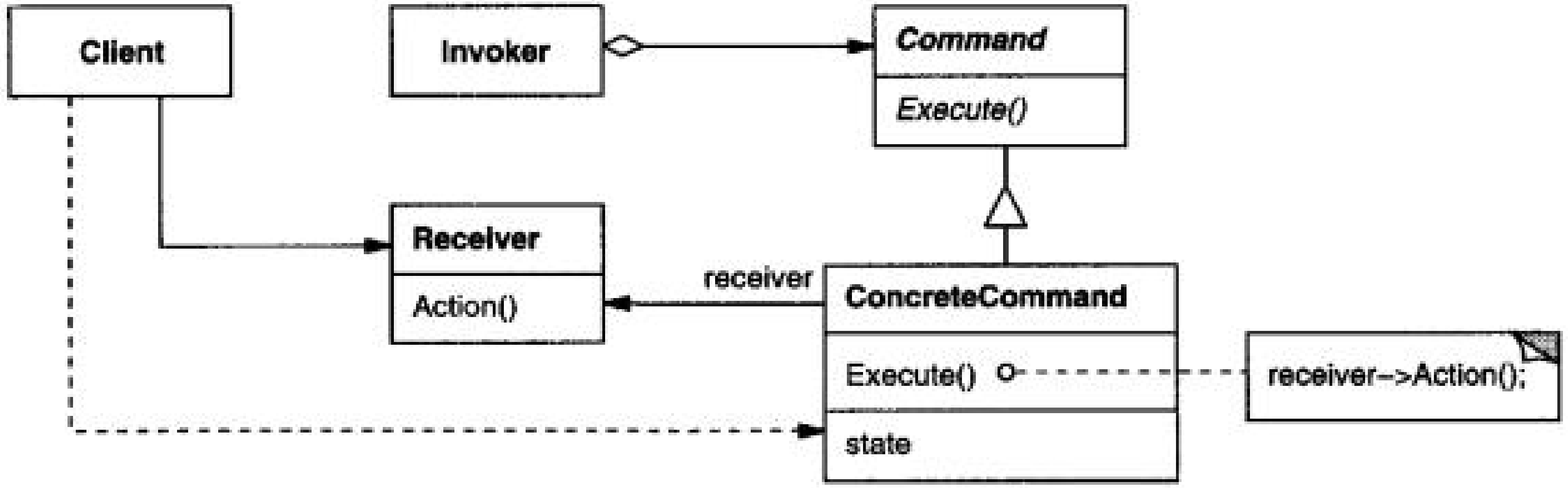
Command 命令模式 (EP)

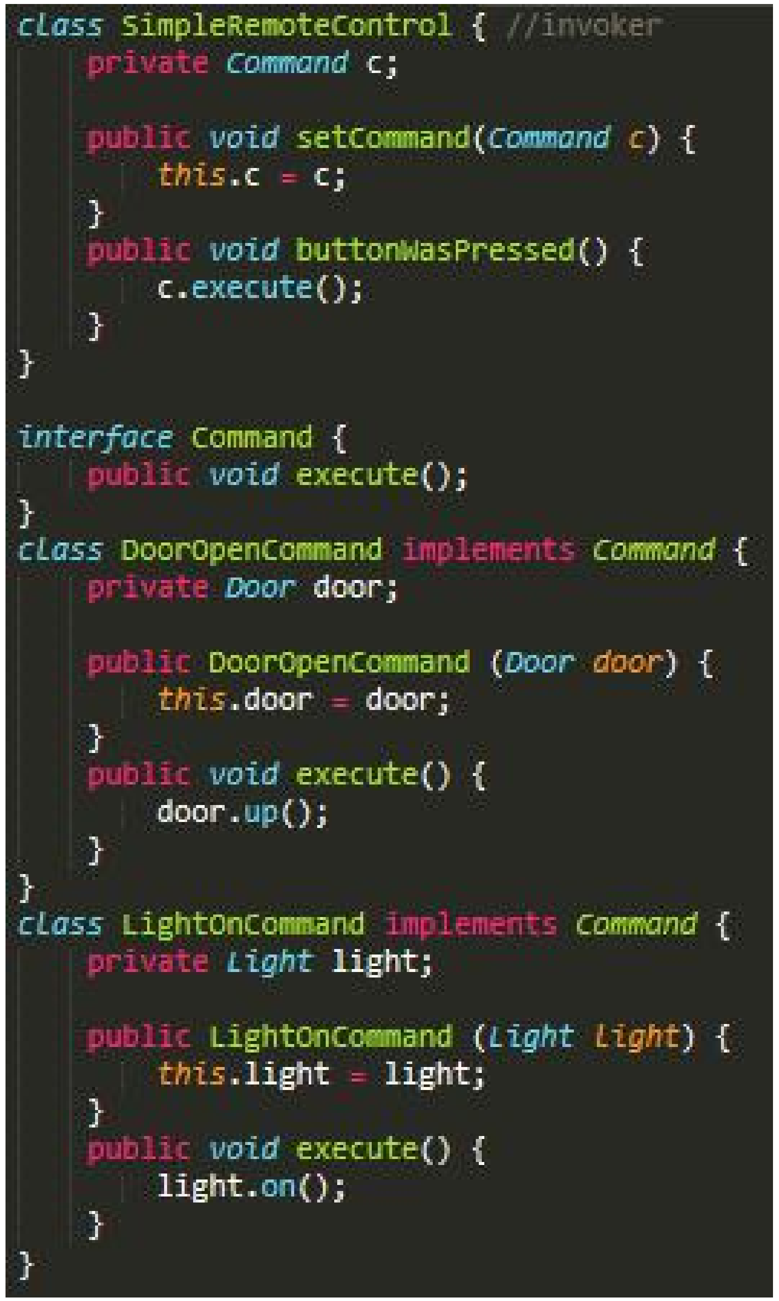
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

將訊息封裝為對象，將訊息利用參數化來處理
(通过调用者调用接受者执行命令，顺序：调用者→接受者→命令)

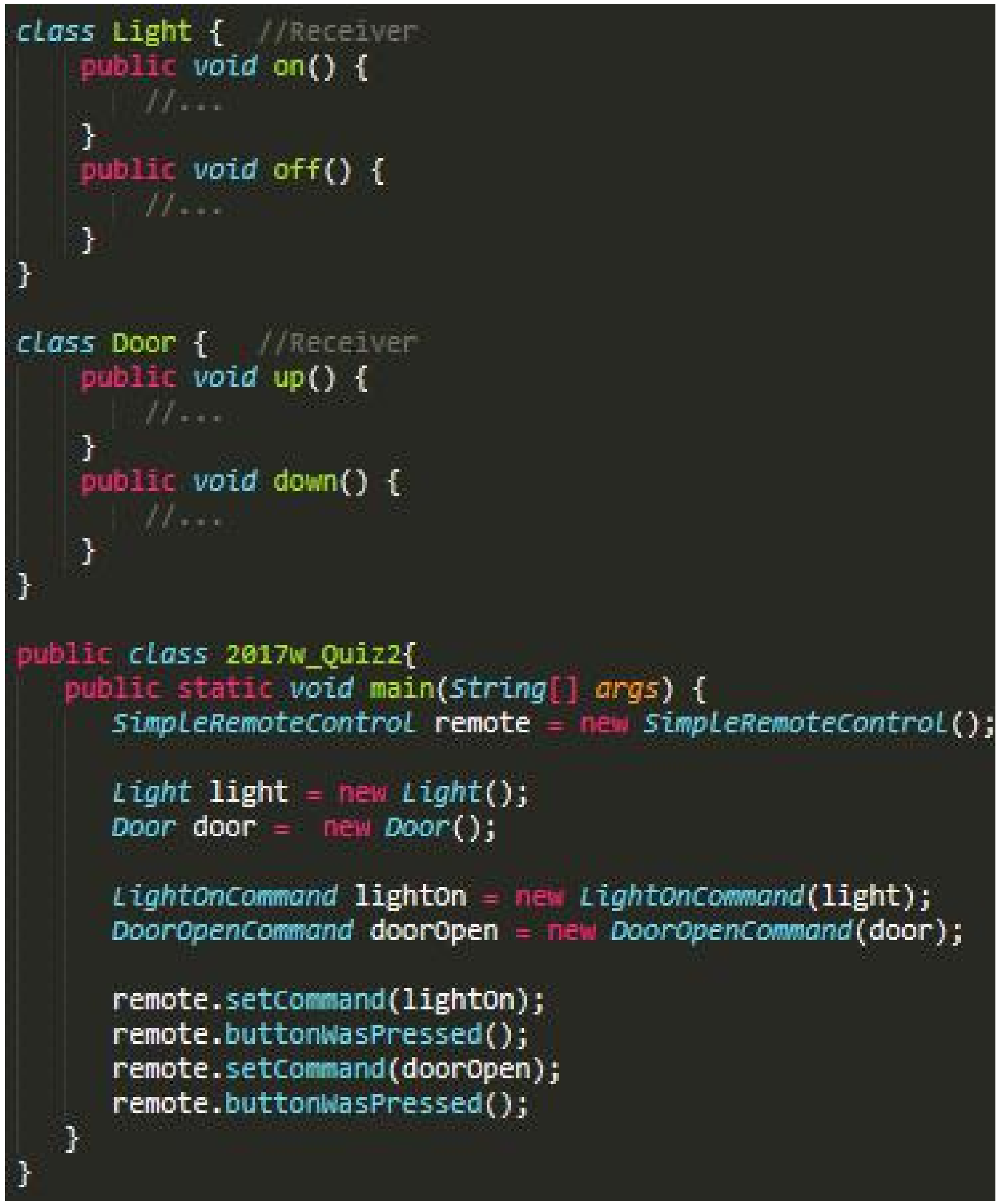
把常用、常常重複的指令包裝成Command
- Reciver是真正執行功能 (Problem Solver)

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





```
class SimpleRemoteControl { //invoker
    private Command c;

    public void setCommand(Command c) {
        this.c = c;
    }
    public void buttonWasPressed() {
        c.execute();
    }
}

interface Command {
    public void execute();
}
class DoorOpenCommand implements Command {
    private Door door;

    public DoorOpenCommand (Door door) {
        this.door = door;
    }
    public void execute() {
        door.up();
    }
}
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand (Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
}
```



```
class Light {  //Receiver
    public void on() {
        //...
    }
    public void off() {
        //...
    }
}

class Door {   //Receiver
    public void up() {
        //...
    }
    public void down() {
        //...
    }
}

public class 2017w_Quiz2{
   public static void main(String[] args) {
      SimpleRemoteControl remote = new SimpleRemoteControl();

      Light light = new Light();
      Door door =  new Door();

      LightOnCommand lightOn = new LightOnCommand(light);
      DoorOpenCommand doorOpen = new DoorOpenCommand(door);

      remote.setCommand(lightOn);
      remote.buttonWasPressed();
      remote.setCommand(doorOpen);
      remote.buttonWasPressed();
   }
}
```

Template 樣板模式 (AEP)

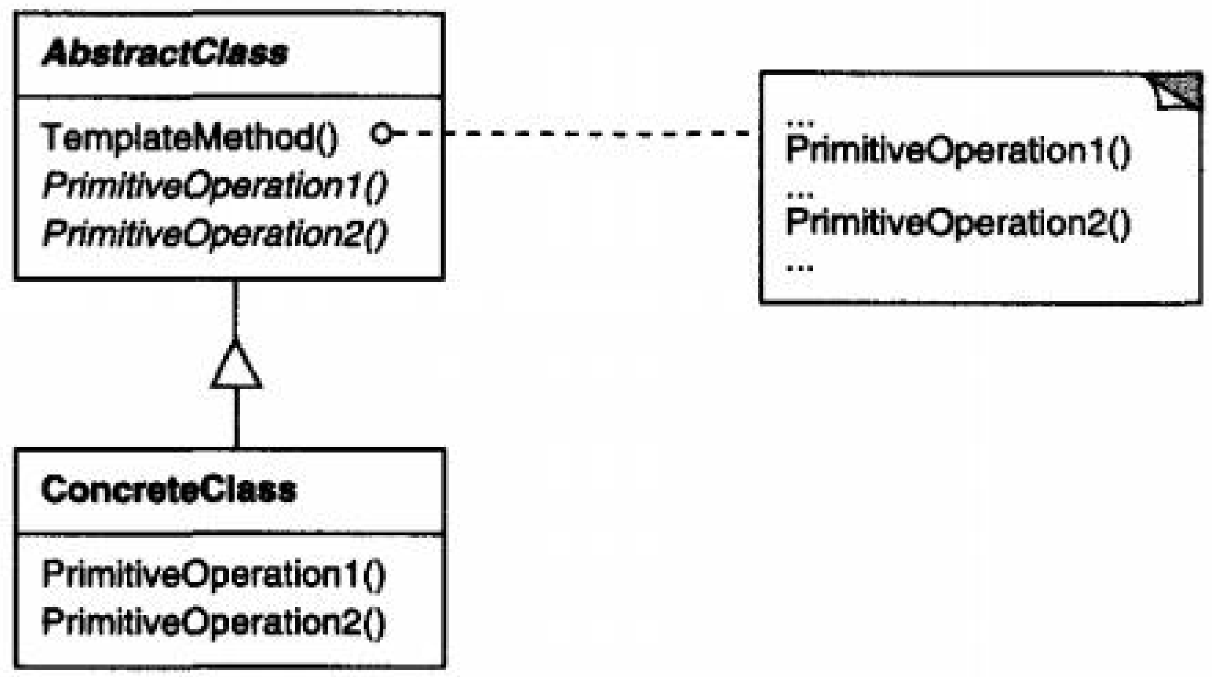
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

在操作中定義算法的骨架，將一些步驟推遲到子類。允許子類重新定義算法而不改變算法的結構。
(一些方法通用，却在每一个子类都重新写了这一方法。)

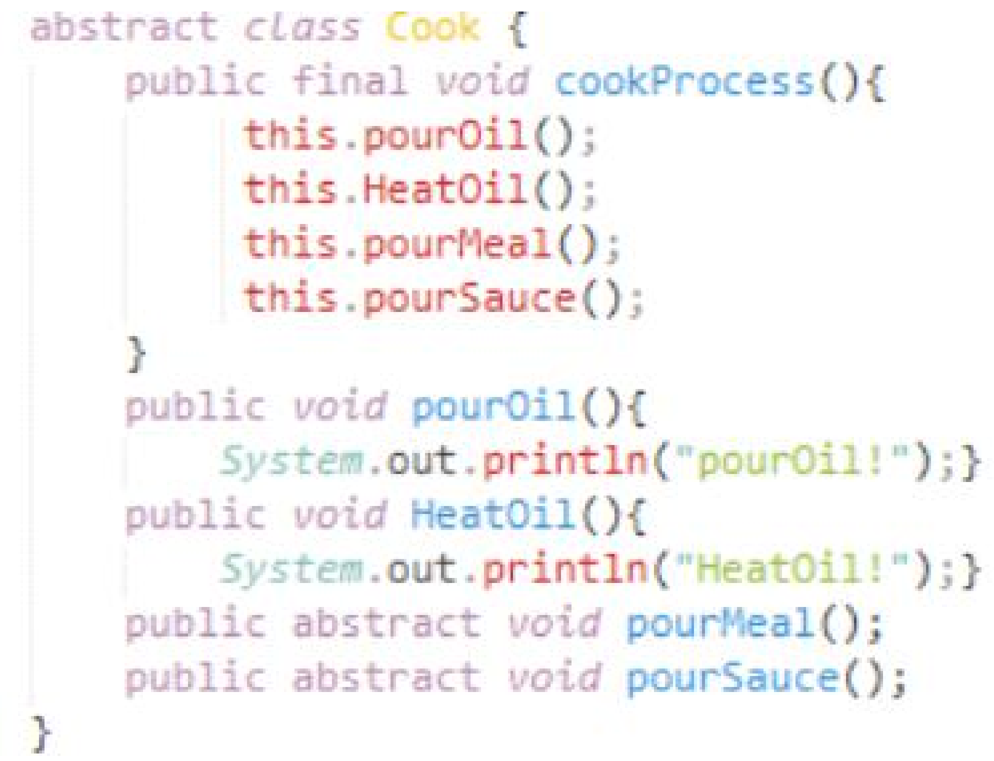
【Method】
Template / 必須是Final因為要定義好執行順序
Primitive / 是Abstract必須被複寫的方法
Concrete / 偶爾會有如果是通用的方法就可以先實作好

【Template vs Strategy】
模板模式(Compiler Time)
一定是按照順序執行，任何重載不會影響到這個次序
策略模式(Runtime)
只提供情景下策略，順序不要求，整個流程都可以被替換

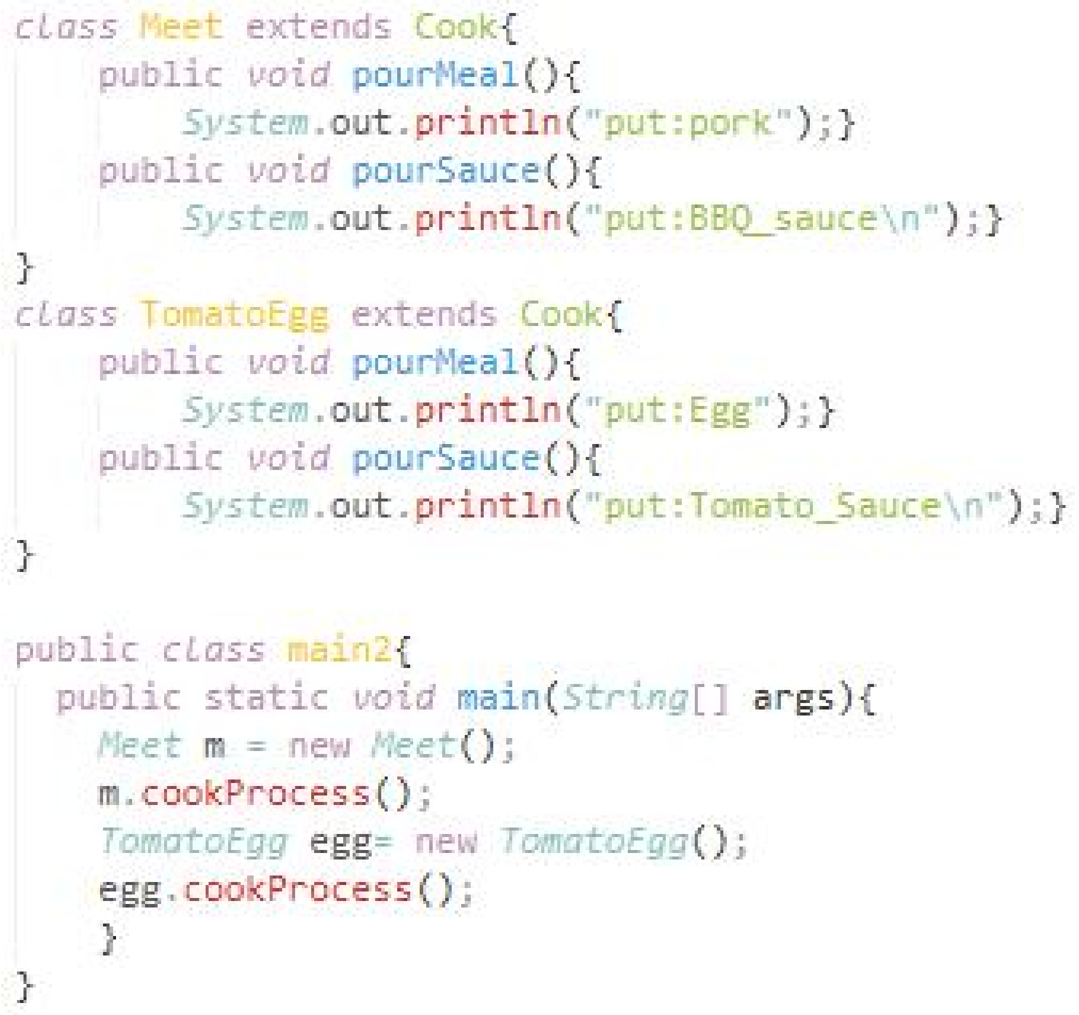
【Hook】
【好萊屋行為】



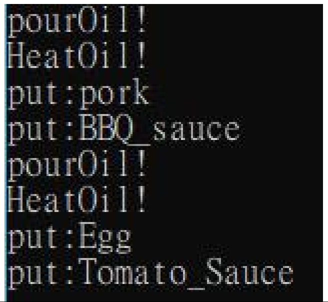
範例1. 煮飯流程(下油、加熱、選肉、選醬)



```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();
}
```

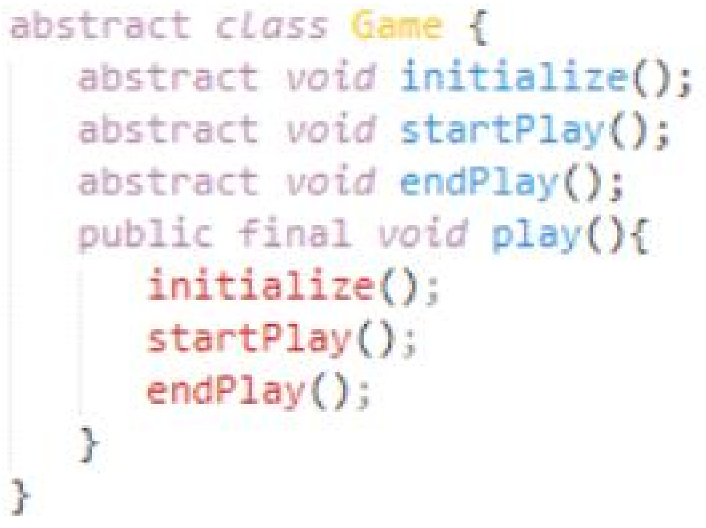


```java
class Meet extends Cook{
    public void pourMeal(){
        System.out.println("put:pork");}
    public void pourSauce(){
        System.out.println("put:BBQ_sauce\n");}
}
class TomatoEgg extends Cook{
    public void pourMeal(){
        System.out.println("put:Egg");}
    public void pourSauce(){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

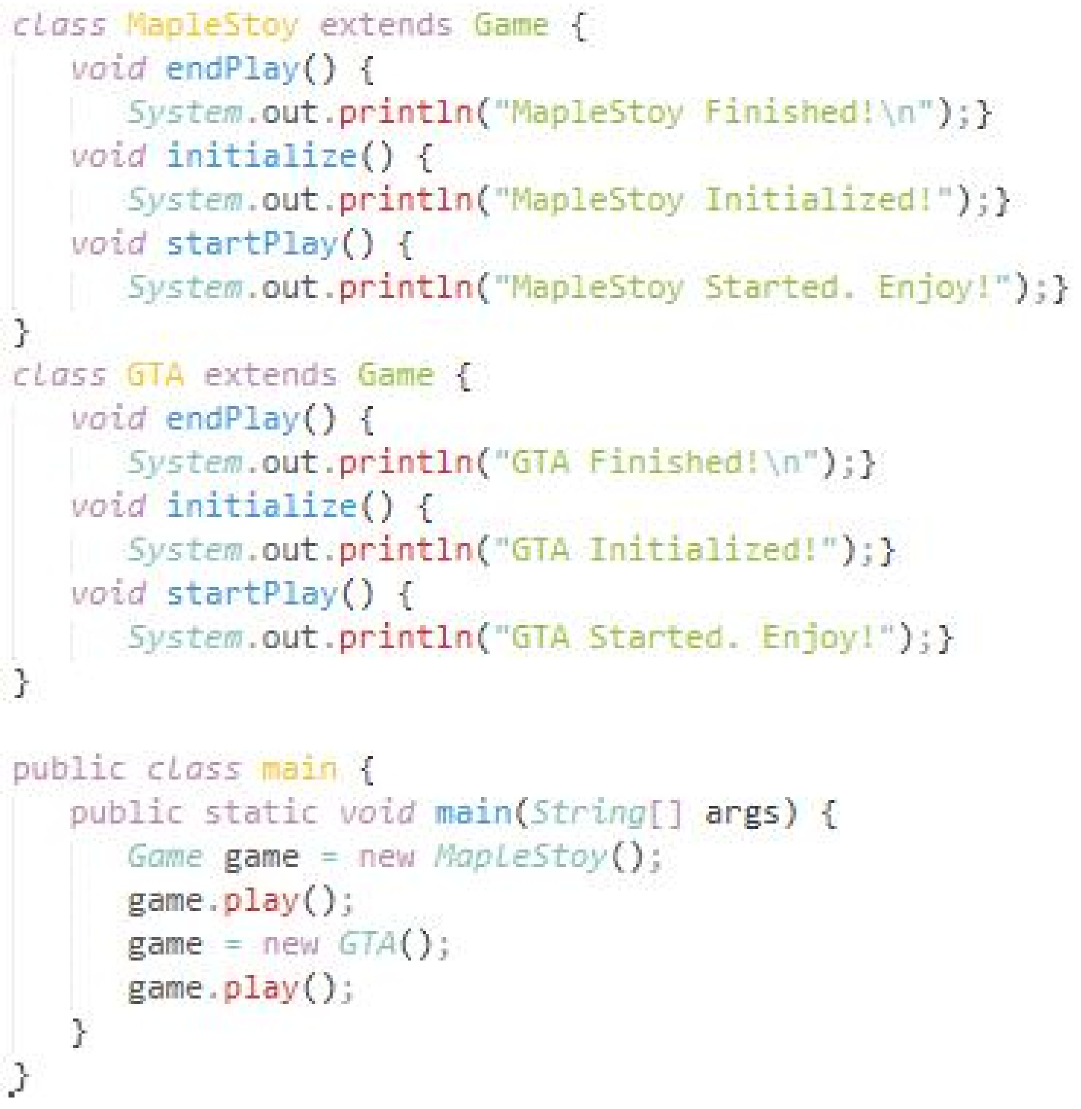


```
pourOil!
HeatOil!
put:pork
put:BBQ_sauce
pourOil!
HeatOil!
put:Egg
put:Tomato_Sauce
```

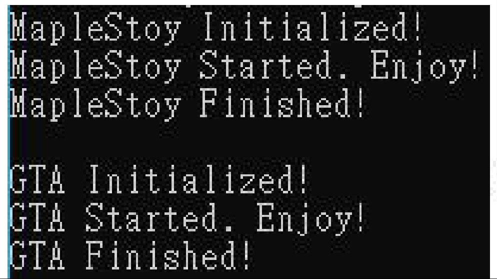
範例2. 玩遊戲流程(啟動/開始/關閉)



```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!
```

Adapter 轉接器模式 (ACEP)

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模式分為兩種：

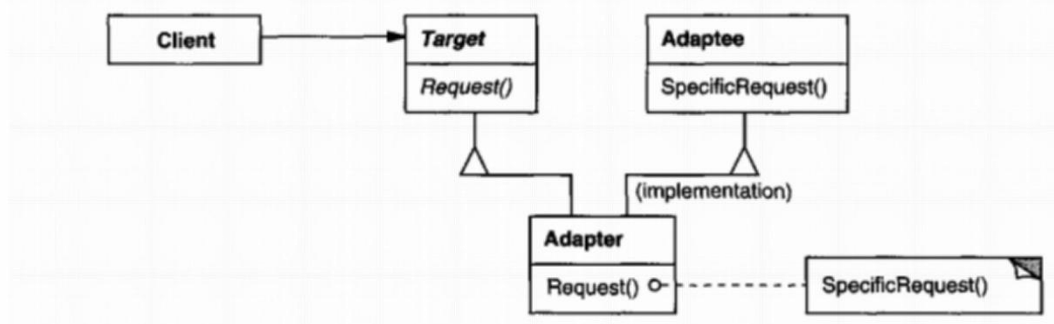
1.類(class)，
  Class方式只是換成了用多重繼承的方式
  讓Adapter多繼承Adaptee。
  【Java不支援多重繼承故不討論】

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

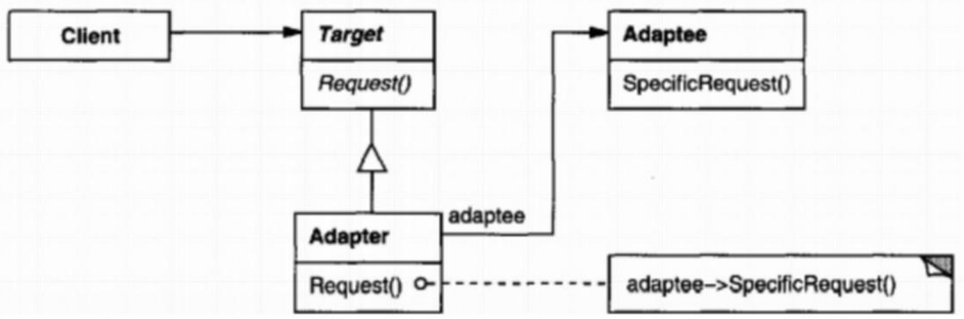
【Adapter vs .Proxy】

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，
【Pluggable Adapters】

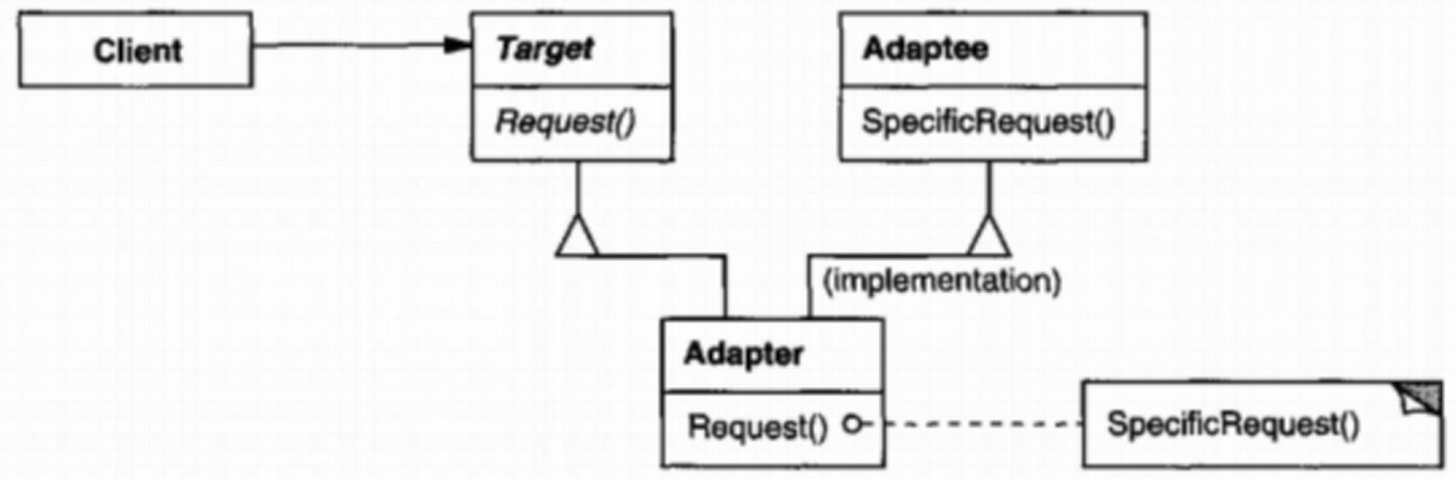
A class adapter uses multiple inheritance to adapt one interface to another:



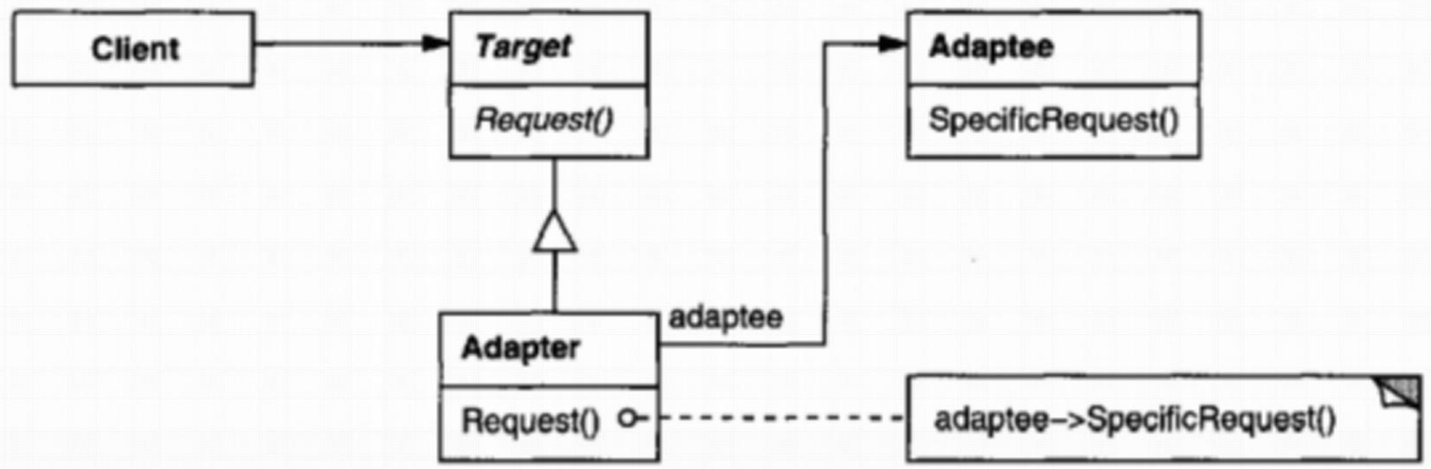
An object adapter relies on object composition:



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

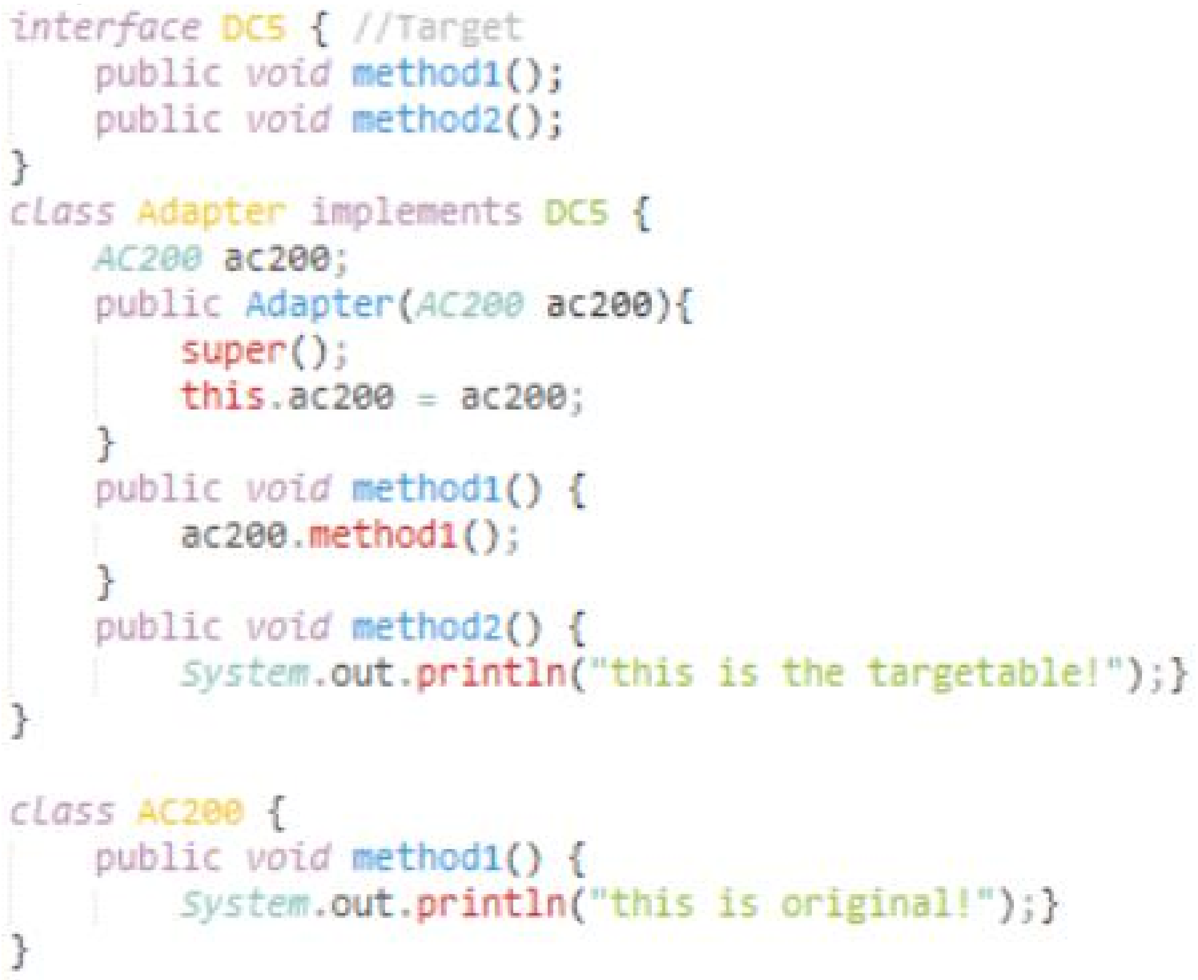


An object adapter relies on object composition:

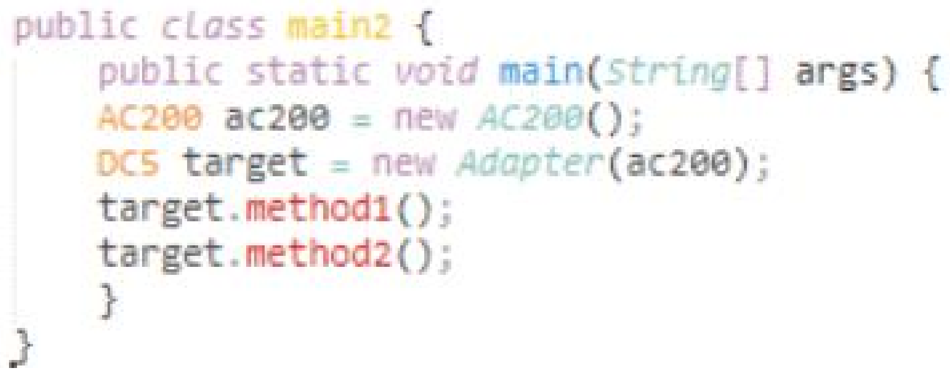


範例1. 手機充電要將AC 220V 轉為 DC 5V充電

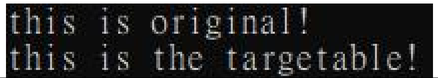
我們用Object Adapter寫一個電源的轉接頭，將AC220v —>



```
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!");}
}
```

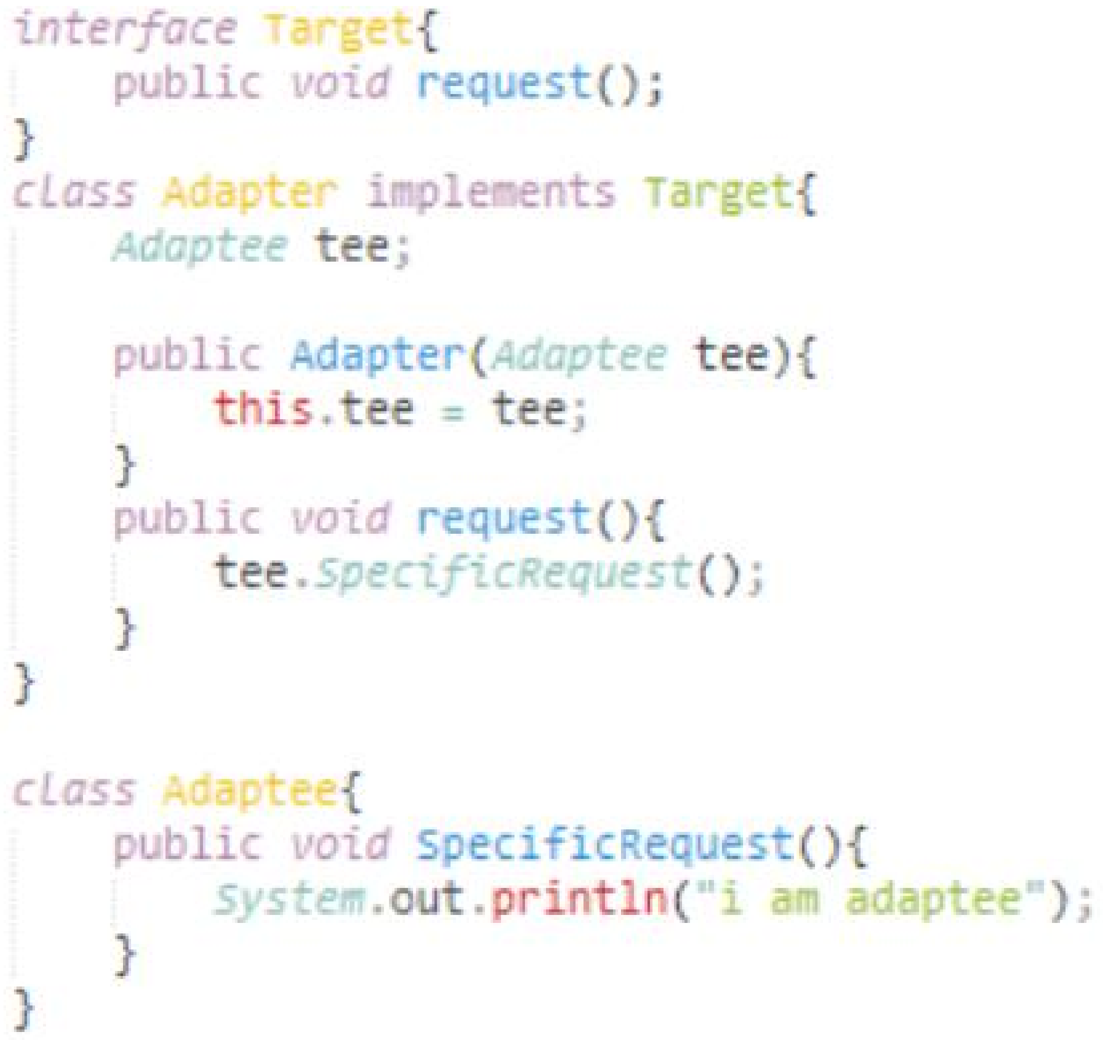


```
public class main2 {
    public static void main(String[] args) {
    AC200 ac200 = new AC200();
    DC5 target = new Adapter(ac200);
    target.method1();
    target.method2();
    }
}
```

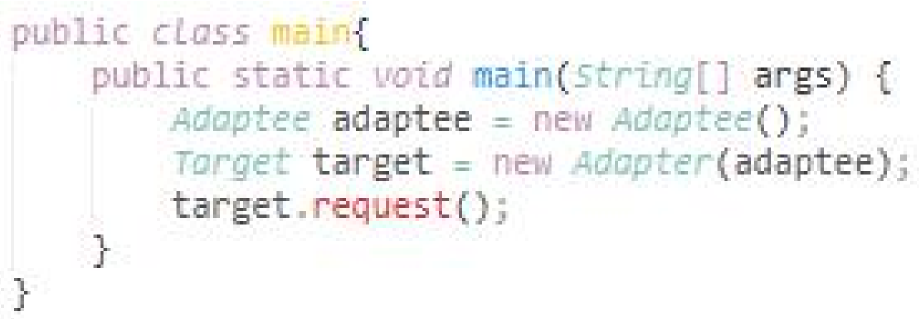


```
this is original!
this is the targetable!
```

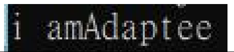
範例2.基本架構



```
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();
    }
}
```

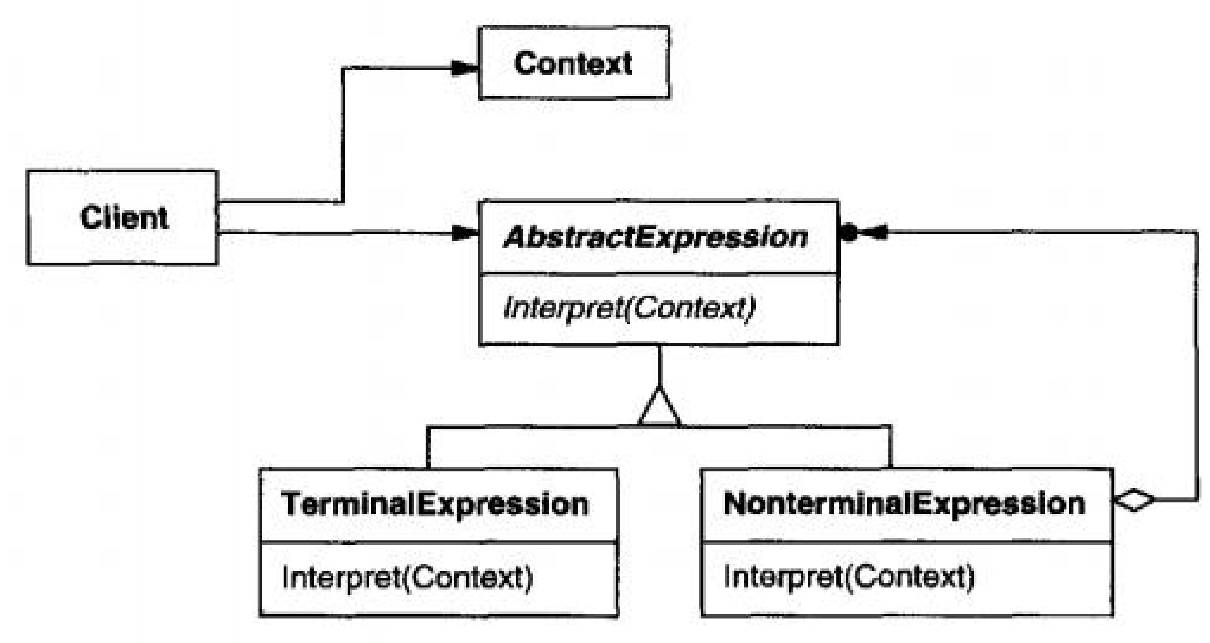


```
i amAdaptee
```

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.

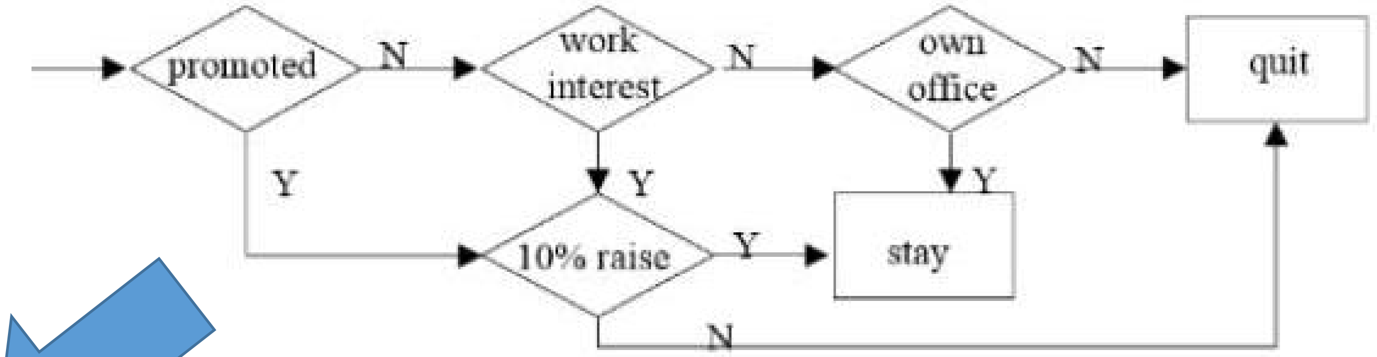
給定一種語言，定義其語法的表示以及解釋器
使用表示來解釋語言中的句子。

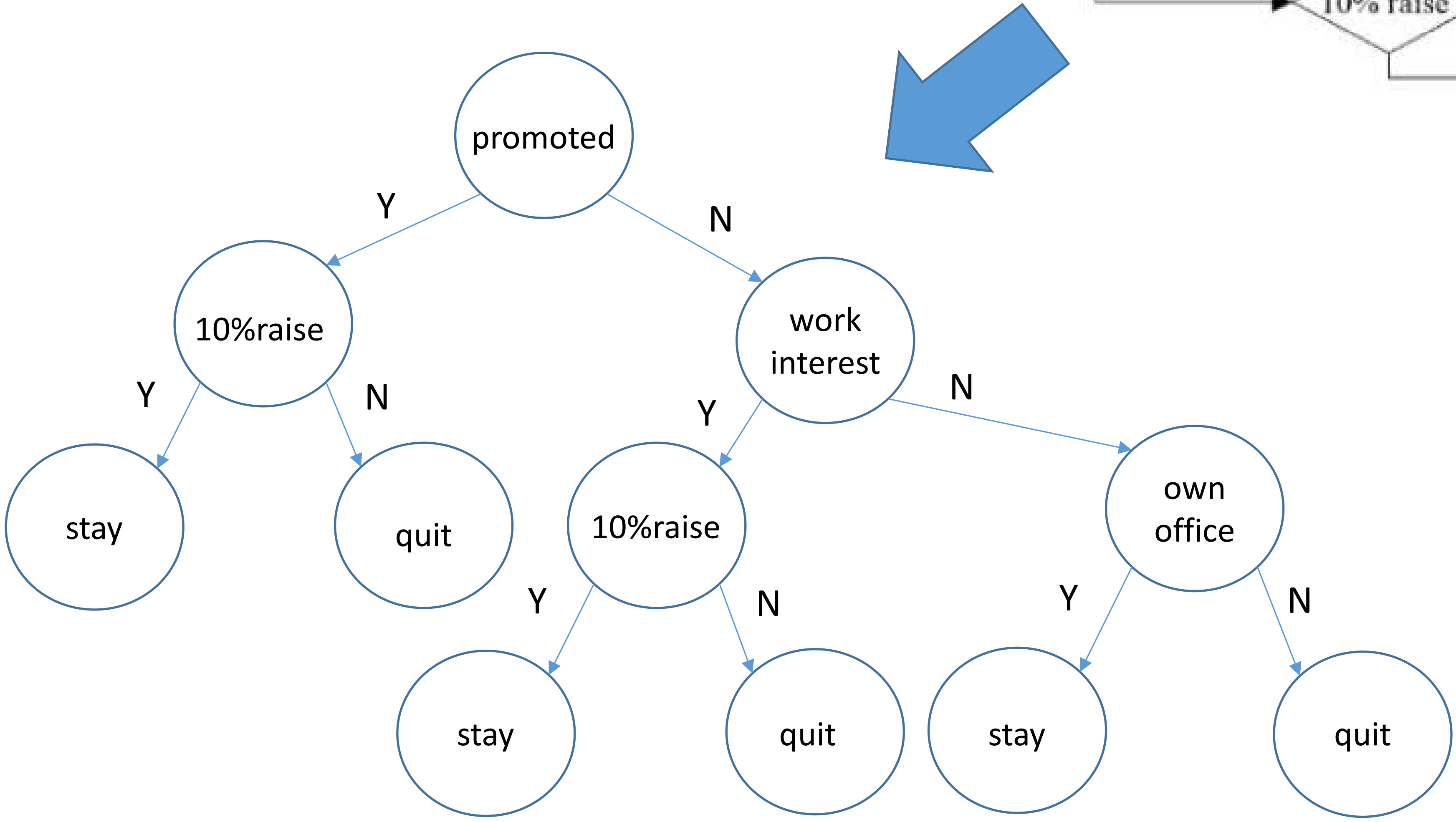


在處理一些複雜的問題上
我們希望可以透過分析器把問題丟進去
答案會自然產生

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

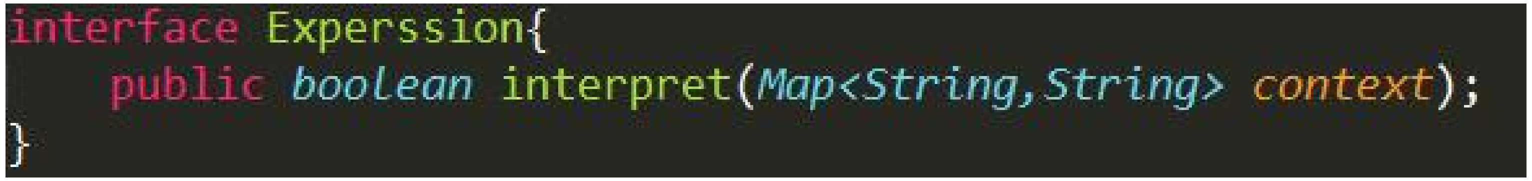
我們試著為這個決策原則建立一棵樹
並使用Interpreter求出答案





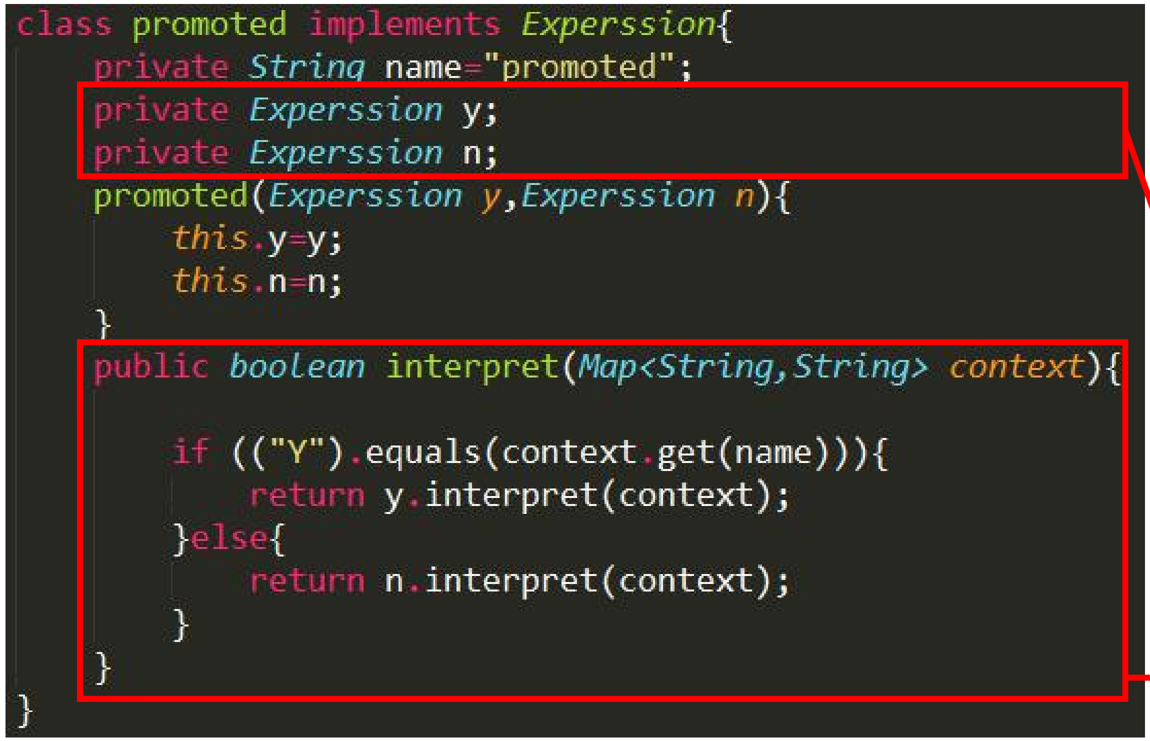
轉換成程式怎麼寫？

1.看到的任何樹節點都是一個Experssion

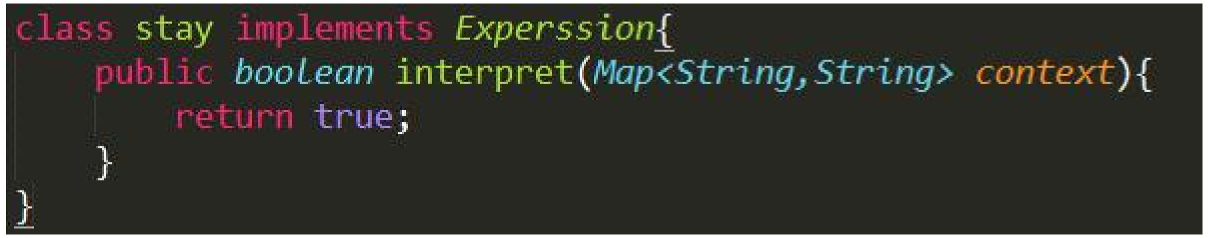


```
interface Experssion{
    public boolean interpret(Map<String,String> context);
}
```

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);
        }
    }
}
```



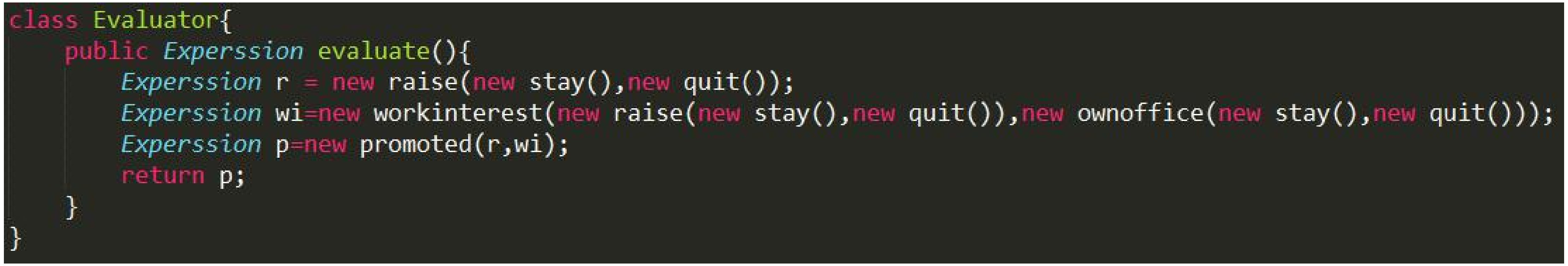
```
class stay implements Experssion{
    public boolean interpret(Map<String,String> context){
        return true;
    }
}
```

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)

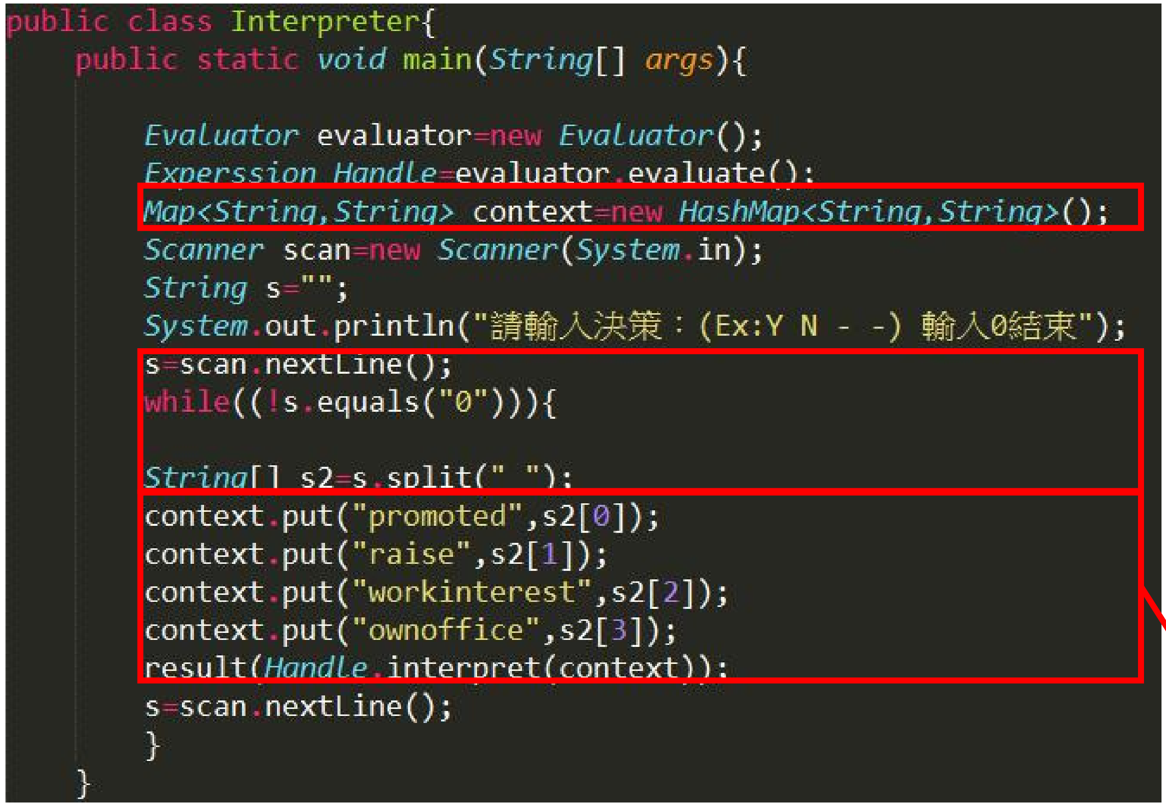
解析的方法，根據此節點的結果分別繼續往左右節點走，直到Terminal節點得出結果。

3.Interpreter最重要的地方，建立結構(樹)

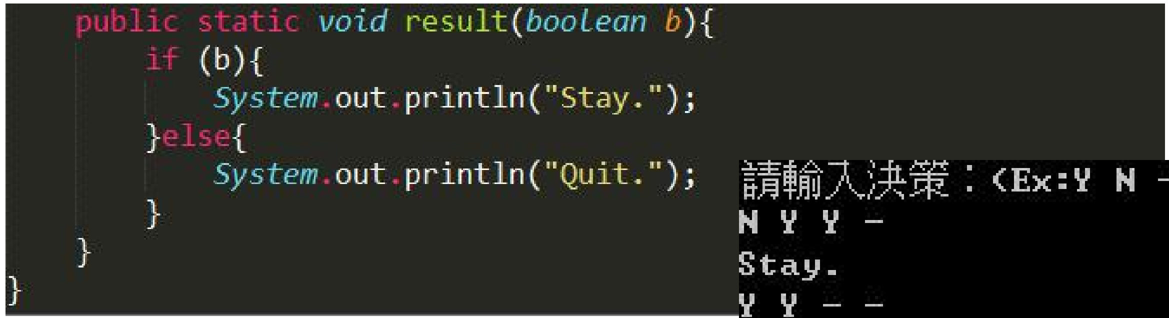


```
class Evaluator{
    public Experssion evaluate(){
        Experssion r = new raise(new stay(),new quit());
        Experssion wi=new workinterest(new raise(new stay(),new quit()),new ownoffice(new stay(),new quit()));
        Experssion p=new promoted(r,wi);
        return p;
    }
}
```

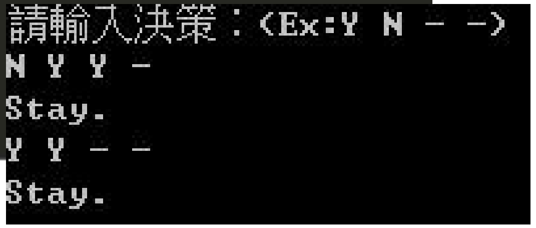
4.輸入與執行



```
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();
        }
    }
```

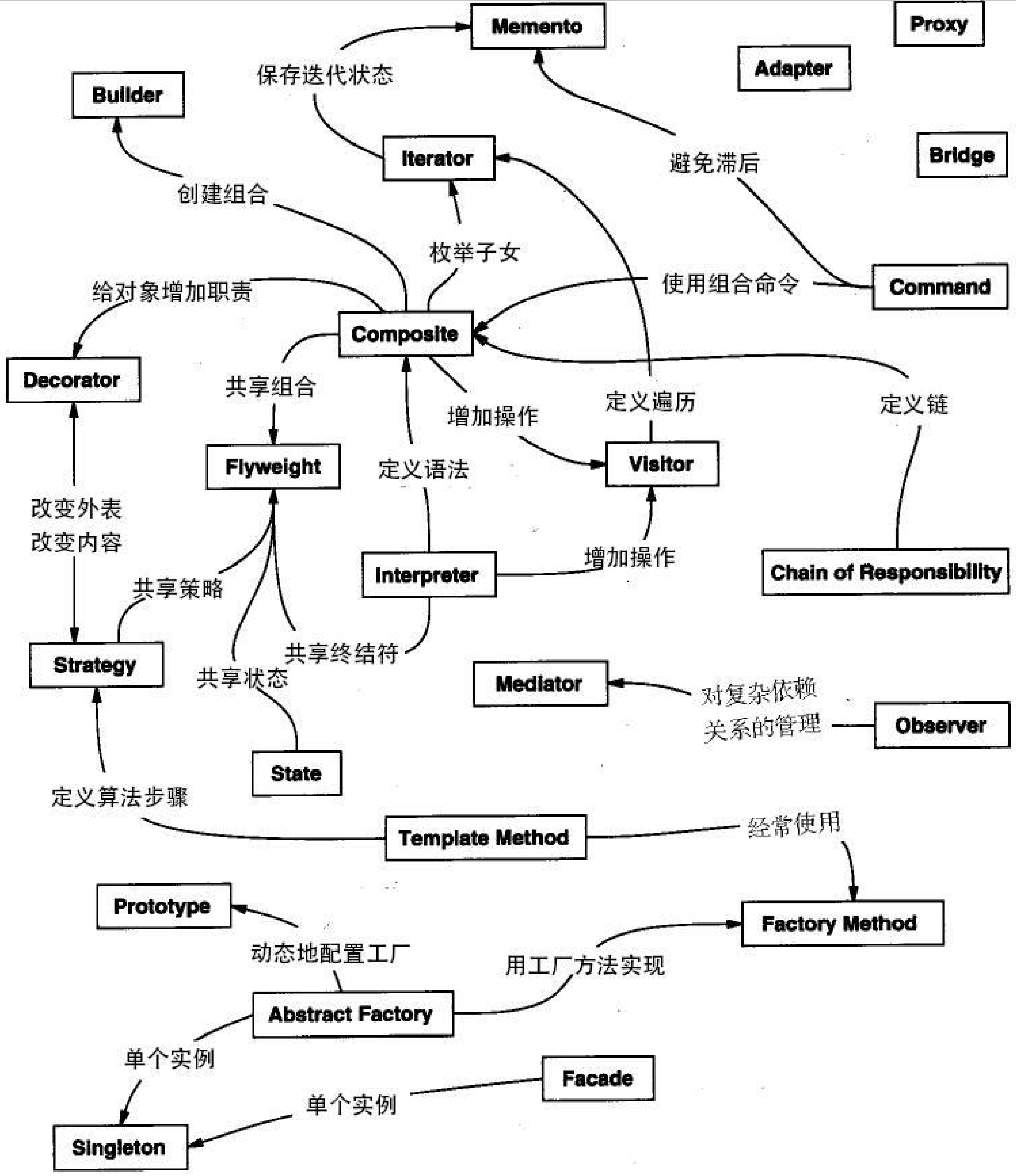


```
    public static void result(boolean b){
        if (b){
            System.out.println("Stay.");
        }else{
            System.out.println("Quit.");
        }
    }
}
```



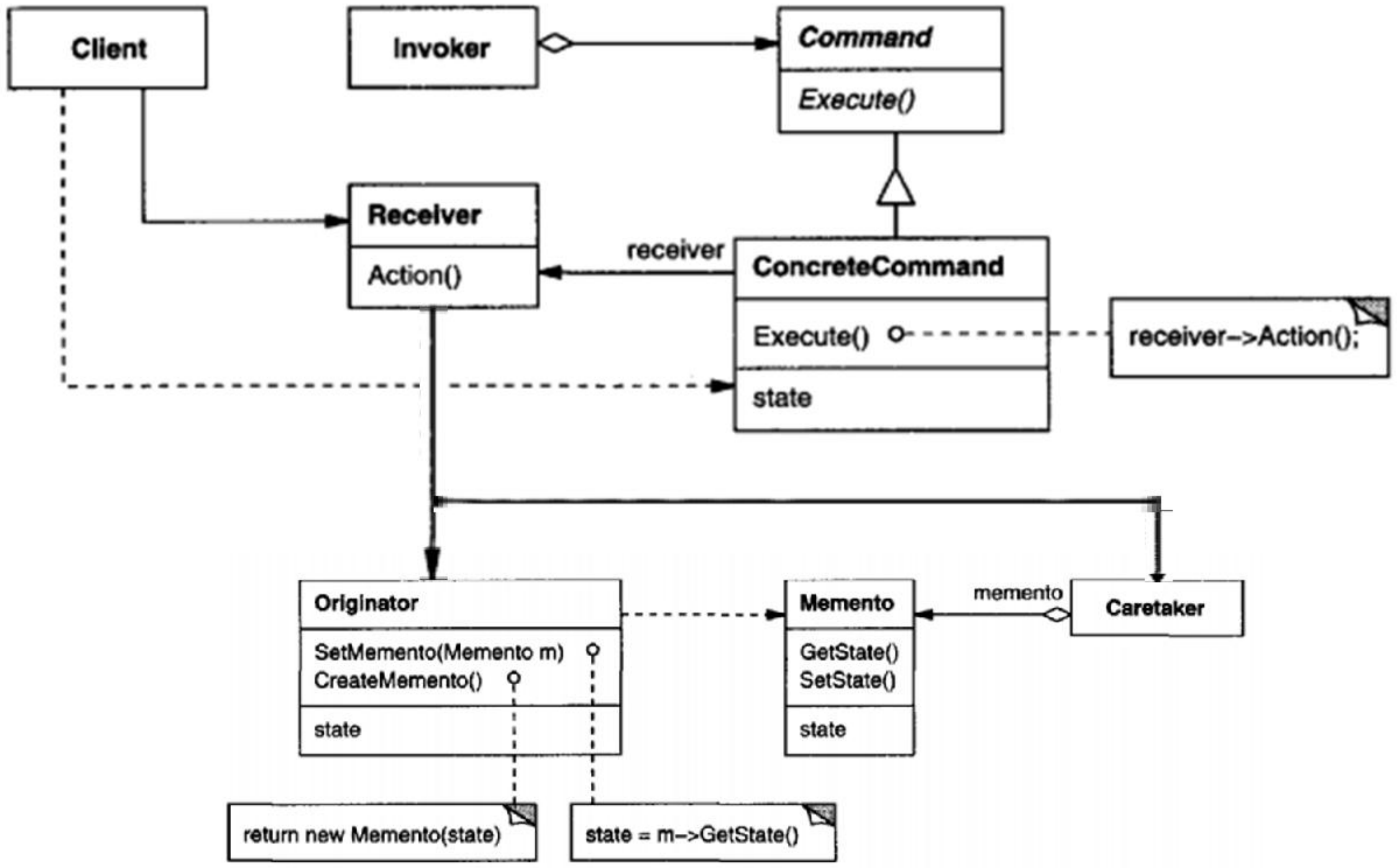
```
請輸入決策：(Ex:Y N - -)
N Y Y -
Stay.
Y Y - -
Stay.
```

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

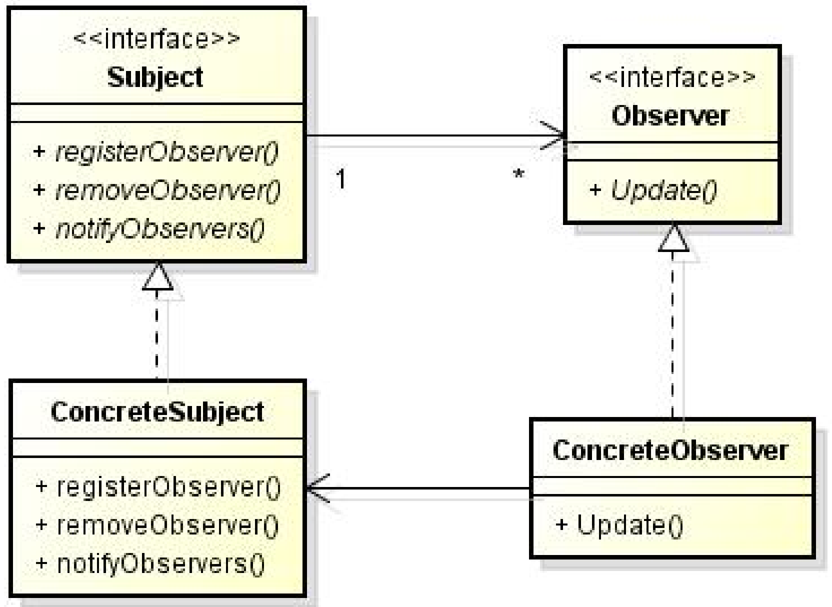


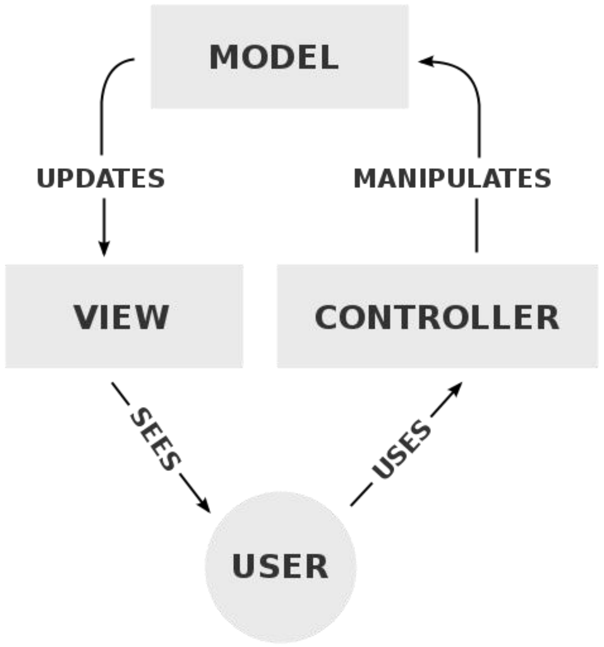
Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance 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)

Command + Memento



Model-View-Control 模式





簡單來說

View
就是使用者可以看到的東西

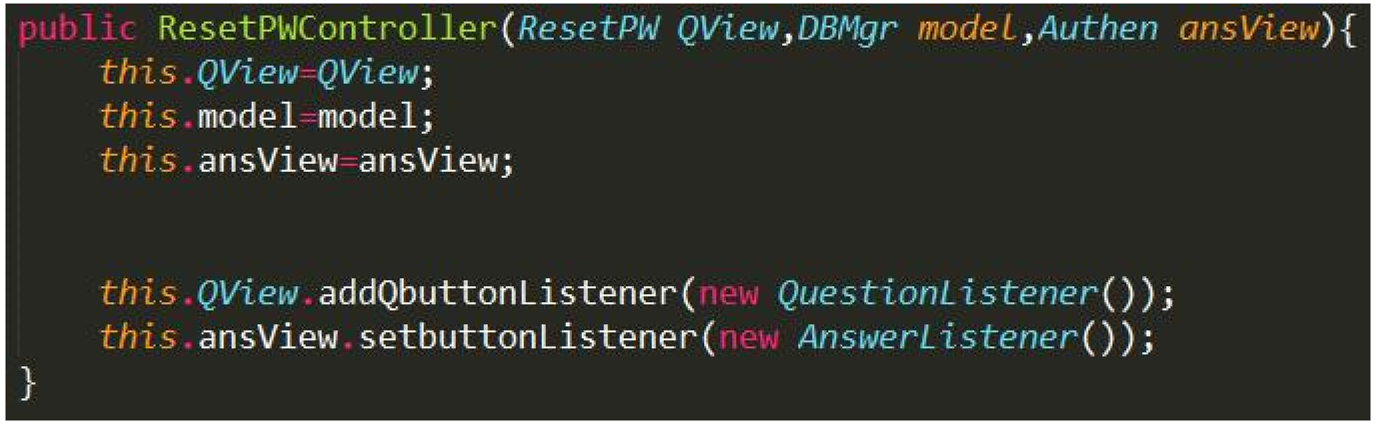
Controller
是當使用者透過View操作之後負責命令Model去做事情

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?

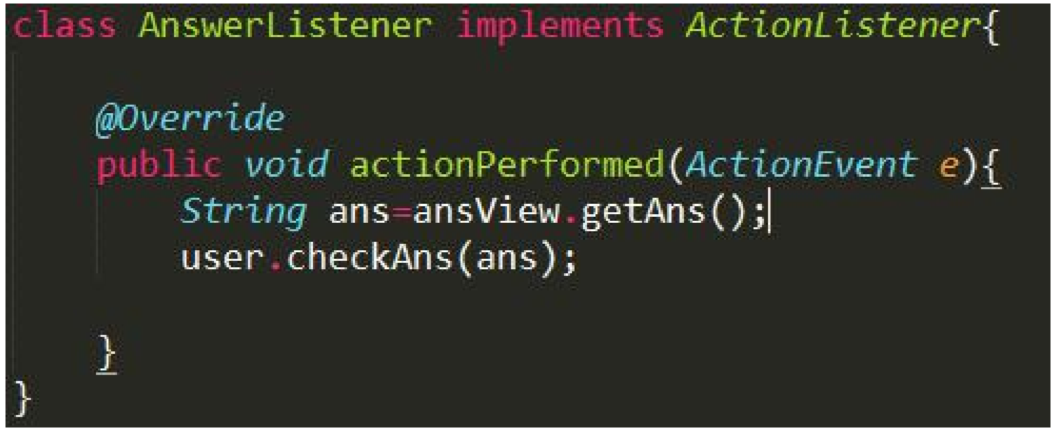
A:透過ActionListener在View上註冊Controller的Listener監聽View的動作

↓**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);

    }
}
```

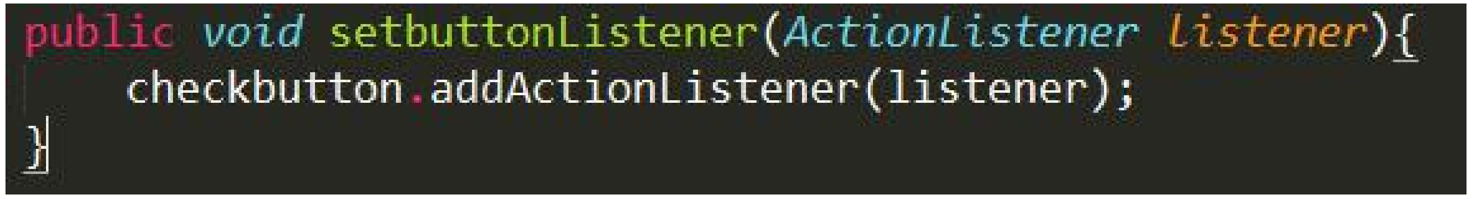
Q:Model要怎麼通知View要更新?

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);
}
```