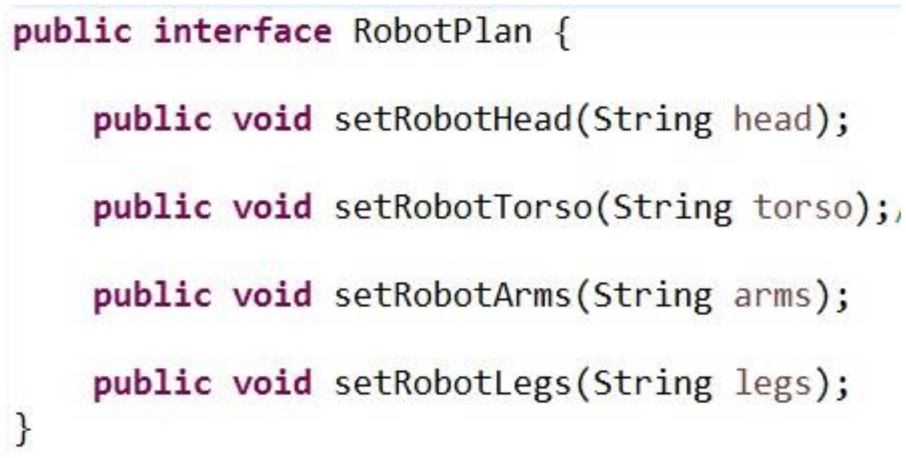
1. Builder Pattern :

   使用時機 ：將一個 [複雜] 產品建造過程，分離出來 便且 標準化步驟

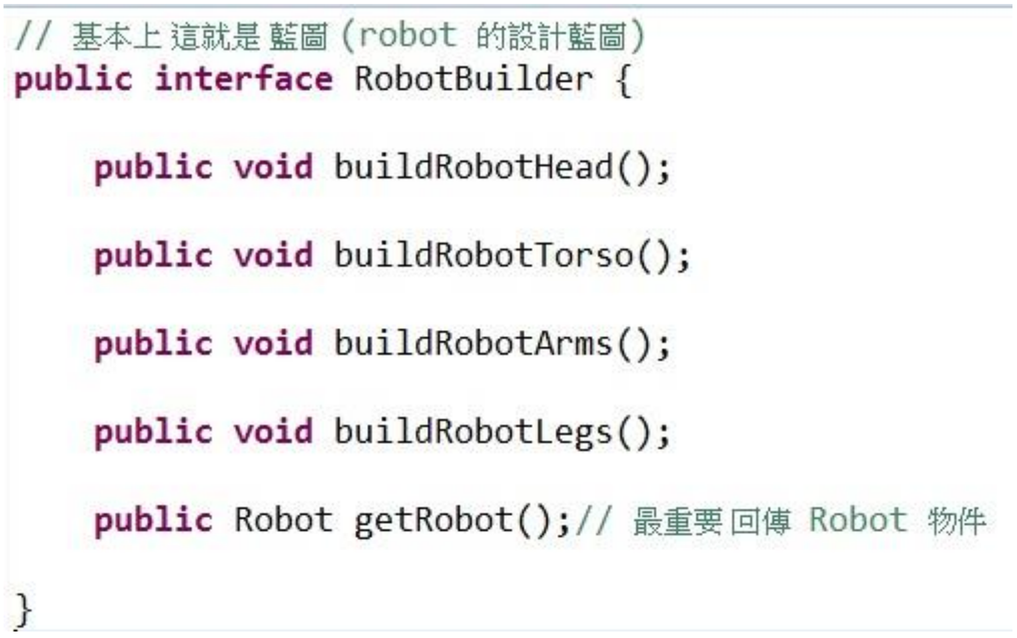
   範例 ：生產 Root

   (1) Robot 的元件



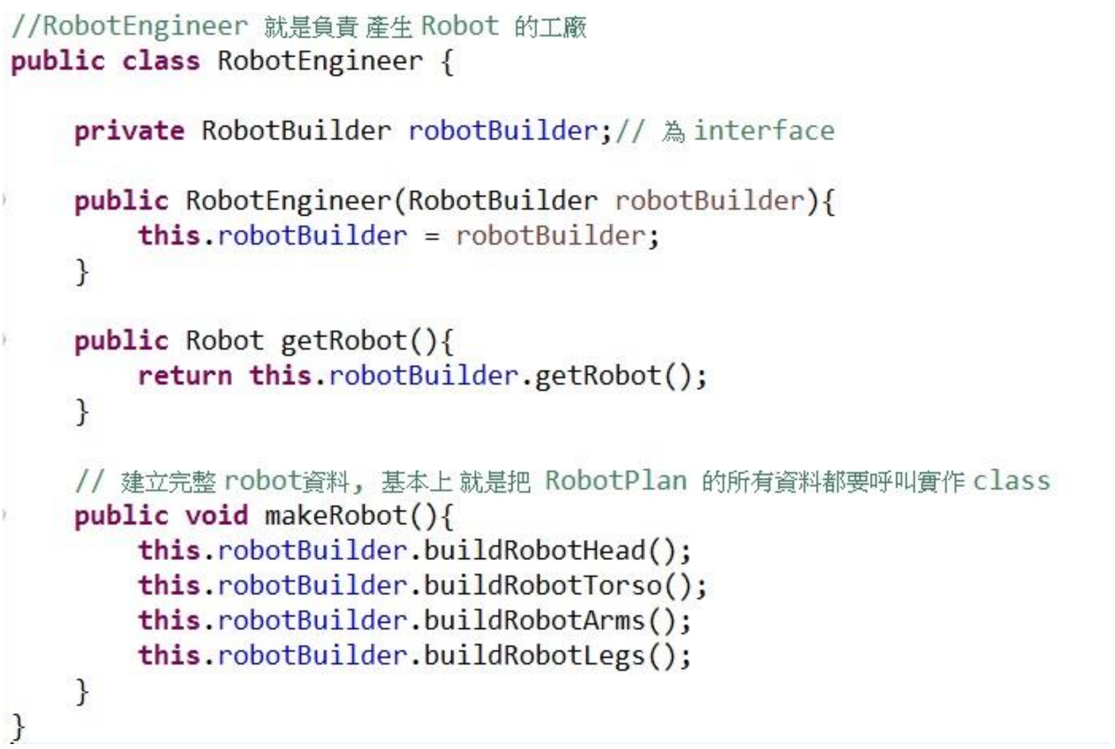
```
public interface RobotPlan {

    public void setRobotHead(String head);

    public void setRobotTorso(String torso);

    public void setRobotArms(String arms);

    public void setRobotLegs(String legs);
}
```

   (2) Builder class



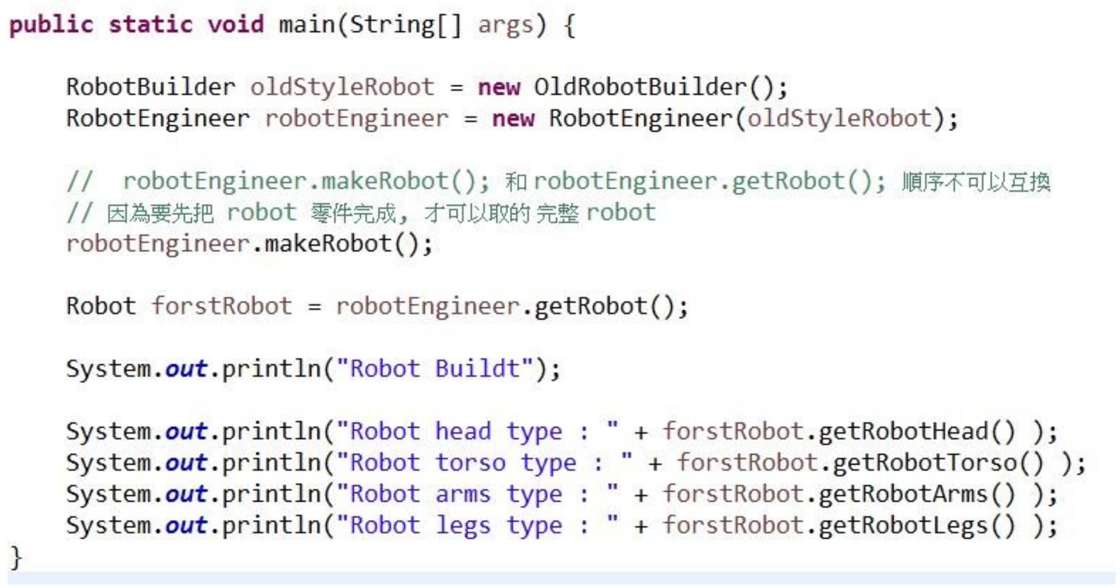
```
// 基本上這就是藍圖(robot 的設計藍圖)
public interface RobotBuilder {

    public void buildRobotHead();

    public void buildRobotTorso();

    public void buildRobotArms();

    public void buildRobotLegs();

    public Robot getRobot();// 最重要回傳 Robot 物件

}
```

(3) 工廠 ： 制定 標準化步驟



```java
//RobotEngineer 就是負責 產生 Robot 的工廠
public class RobotEngineer {

    private RobotBuilder robotBuilder;// 為 interface

    public RobotEngineer(RobotBuilder robotBuilder){
        this.robotBuilder = robotBuilder;
    }

    public Robot getRobot(){
        return this.robotBuilder.getRobot();
    }

    // 建立完整 robot資料, 基本上 就是把 RobotPlan 的所有資料都要呼叫實作 class
    public void makeRobot(){
        this.robotBuilder.buildRobotHead();
        this.robotBuilder.buildRobotTorso();
        this.robotBuilder.buildRobotArms();
        this.robotBuilder.buildRobotLegs();
    }
}
```

(4) 應用



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

    RobotBuilder oldStyleRobot = new OldRobotBuilder();
    RobotEngineer robotEngineer = new RobotEngineer(oldStyleRobot);

    //  robotEngineer.makeRobot(); 和 robotEngineer.getRobot(); 順序不可以互換
    // 因為要先把 robot 零件完成, 才可以取的 完整 robot
    robotEngineer.makeRobot();

    Robot forstRobot = robotEngineer.getRobot();

    System.out.println("Robot Buildt");

    System.out.println("Robot head type : " + forstRobot.getRobotHead() );
    System.out.println("Robot torso type : " + forstRobot.getRobotTorso() );
    System.out.println("Robot arms type : " + forstRobot.getRobotArms() );
    System.out.println("Robot legs type : " + forstRobot.getRobotLegs() );
}
```

2. Bridge Pattern

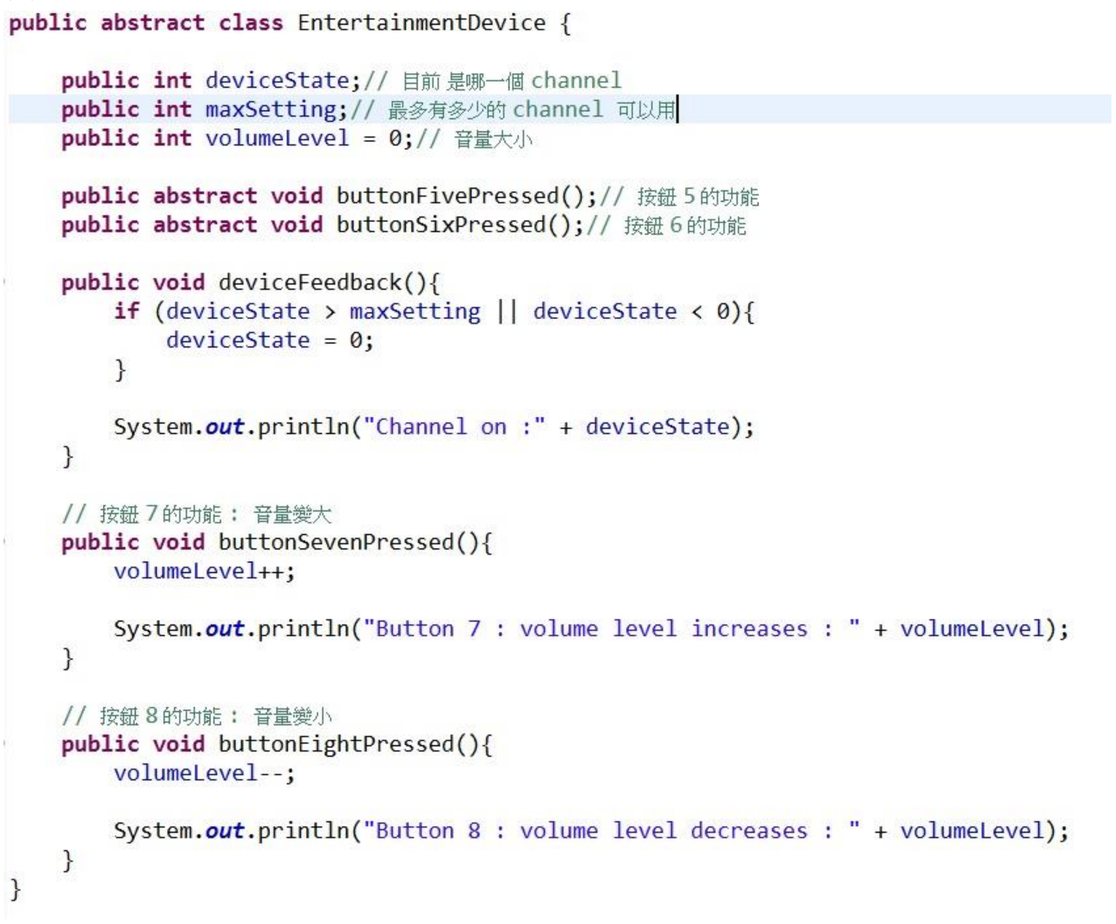
使用時機 ： 抽像物件 與 實作物件 可以 解耦合(decouple)，各自獨立變化

範例 ： 把 遙控器 和 娛樂設備 (TV/DVD)分離

更詳細解說 ： 請參考此中文網站

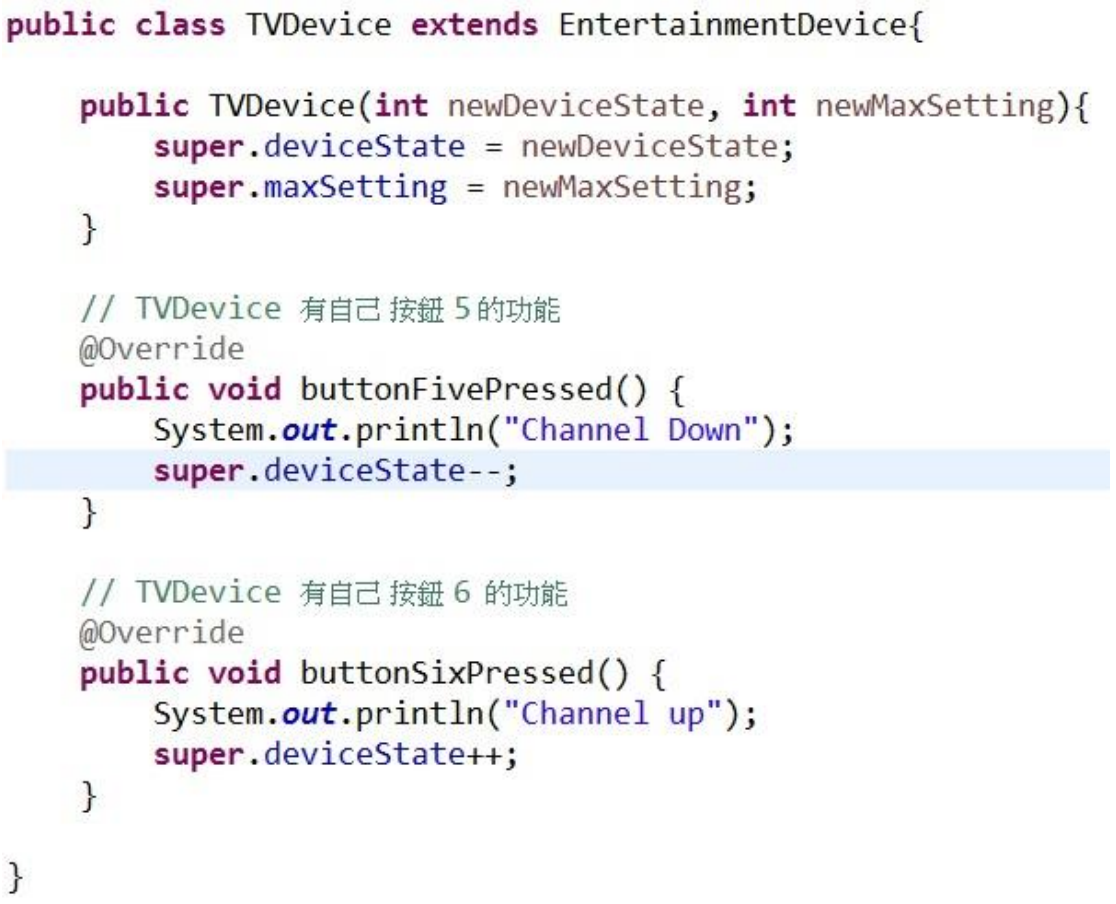
http://www.cnblogs.com/abcdwxc/archive/2007/09/05/882918.html

(1) 娛樂設備



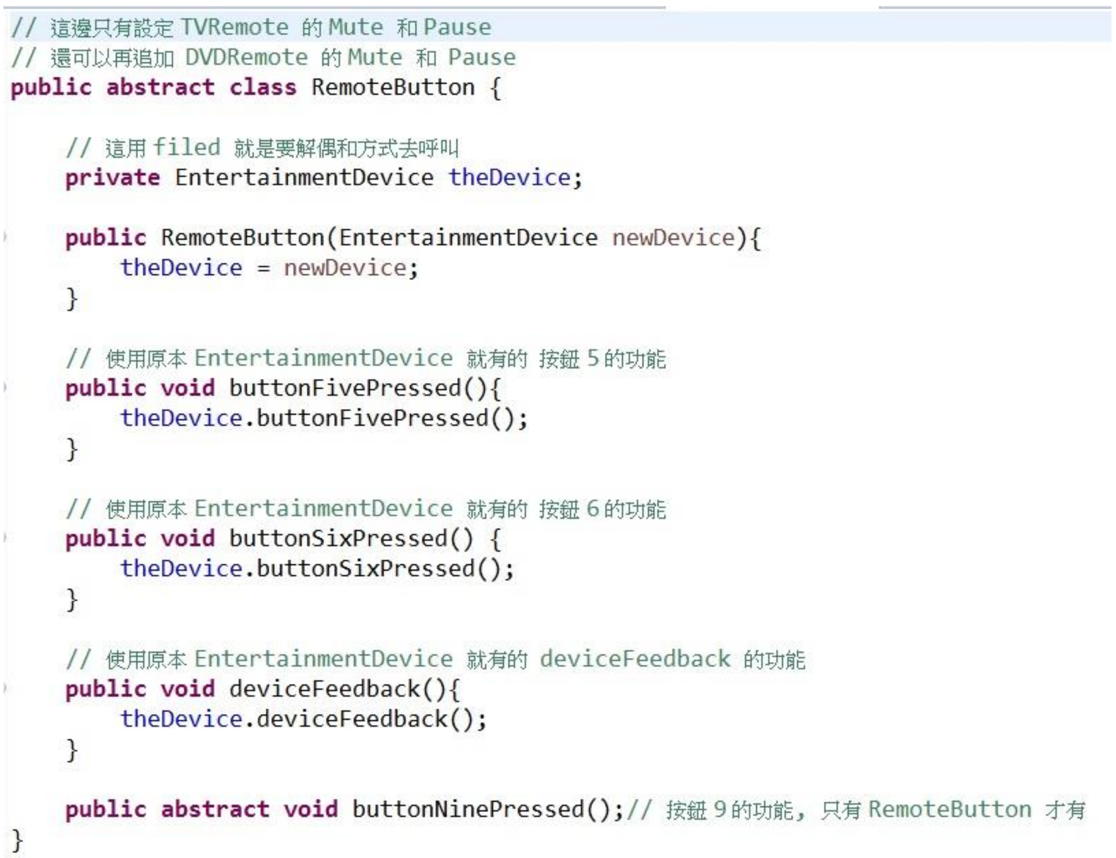
```java
public abstract class EntertainmentDevice {

    public int deviceState;// 目前 是哪一個 channel
    public int maxSetting;// 最多有多少的 channel 可以用
    public int volumeLevel = 0;// 音量大小

    public abstract void buttonFivePressed();// 按鈕 5 的功能
    public abstract void buttonSixPressed();// 按鈕 6 的功能

    public void deviceFeedback(){
        if (deviceState > maxSetting || deviceState < 0){
            deviceState = 0;
        }

        System.out.println("Channel on :" + deviceState);
    }

    // 按鈕 7 的功能： 音量變大
    public void buttonSevenPressed(){
        volumeLevel++;

        System.out.println("Button 7 : volume level increases : " + volumeLevel);
    }

    // 按鈕 8 的功能： 音量變小
    public void buttonEightPressed(){
        volumeLevel--;

        System.out.println("Button 8 : volume level decreases : " + volumeLevel);
    }
}
```

(2) TV



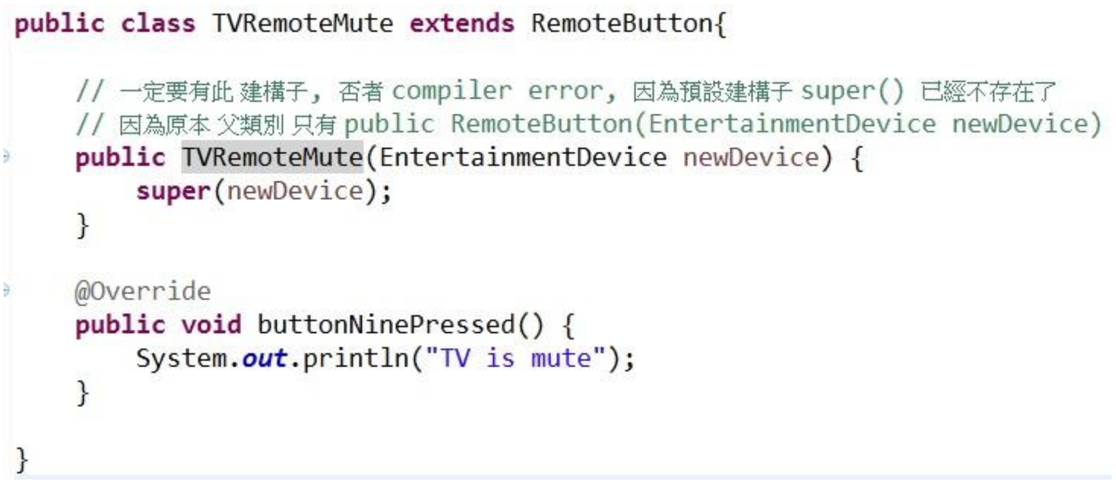
```java
public class TVDevice extends EntertainmentDevice{

    public TVDevice(int newDeviceState, int newMaxSetting){
        super.deviceState = newDeviceState;
        super.maxSetting = newMaxSetting;
    }

    // TVDevice 有自己 按鈕 5 的功能
    @Override
    public void buttonFivePressed() {
        System.out.println("Channel Down");
        super.deviceState--;
    }

    // TVDevice 有自己 按鈕 6 的功能
    @Override
    public void buttonSixPressed() {
        System.out.println("Channel up");
        super.deviceState++;
    }

}
```

(3) Remote controller

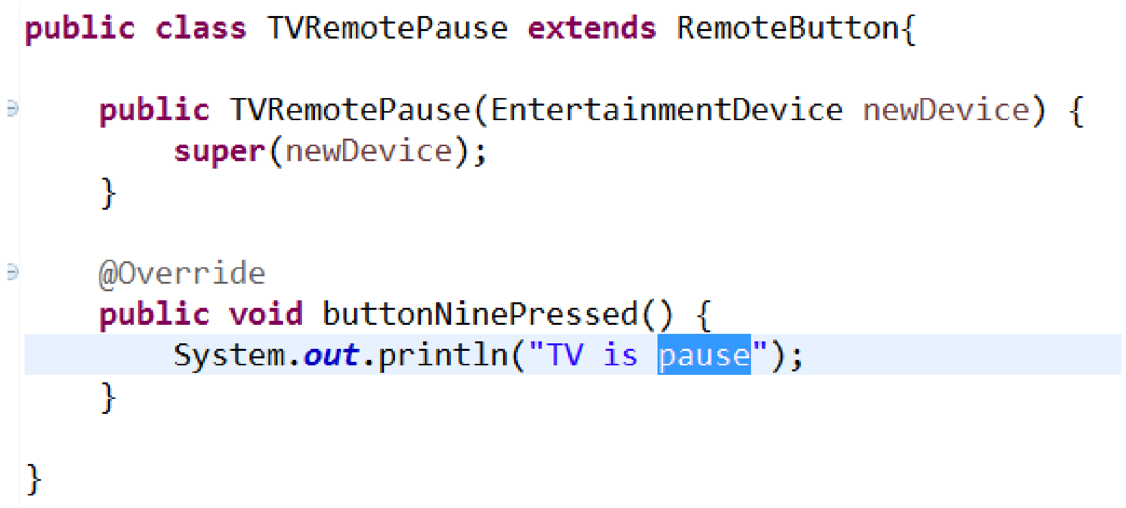


```java
// 這邊只有設定 TVRemote 的 Mute 和 Pause
// 還可以再追加 DVDRemote 的 Mute 和 Pause
public abstract class RemoteButton {

    // 這用 filed 就是要解偶和方式去呼叫
    private EntertainmentDevice theDevice;

    public RemoteButton(EntertainmentDevice newDevice){
        theDevice = newDevice;
    }

    // 使用原本 EntertainmentDevice 就有的 按鈕 5 的功能
    public void buttonFivePressed(){
        theDevice.buttonFivePressed();
    }

    // 使用原本 EntertainmentDevice 就有的 按鈕 6 的功能
    public void buttonSixPressed() {
        theDevice.buttonSixPressed();
    }

    // 使用原本 EntertainmentDevice 就有的 deviceFeedback 的功能
    public void deviceFeedback(){
        theDevice.deviceFeedback();
    }

    public abstract void buttonNinePressed();// 按鈕 9 的功能, 只有 RemoteButton 才有
}
```

(4) 兩個不同的 Remote controller

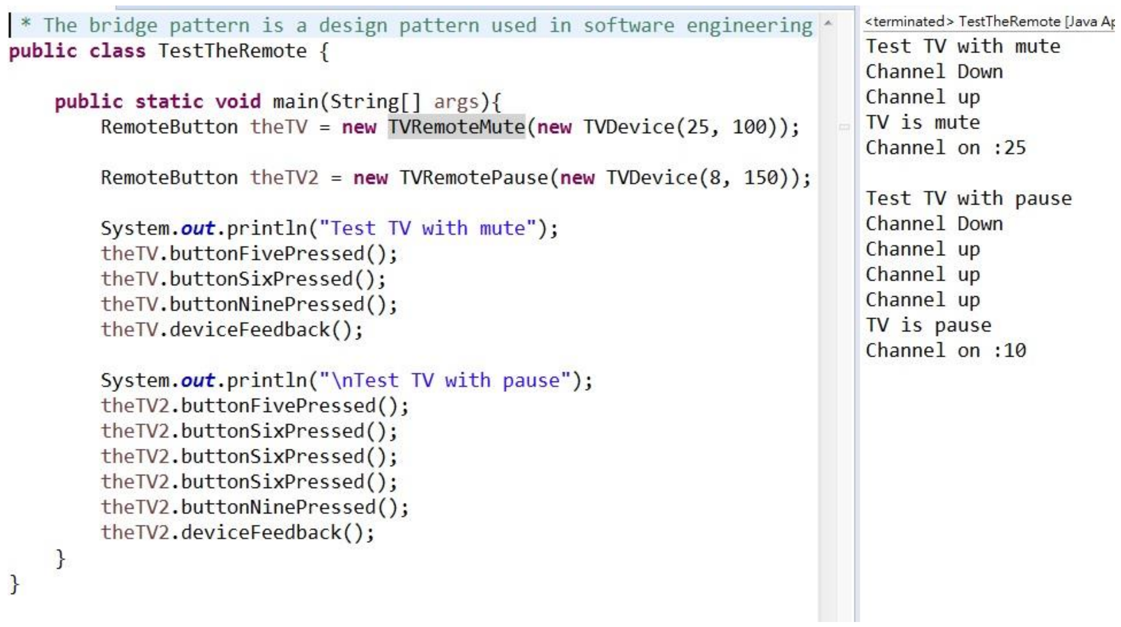


```java
public class TVRemoteMute extends RemoteButton{

    // 一定要有此 建構子, 否者 compiler error, 因為預設建構子 super() 已經不存在了
    // 因為原本 父類別 只有 public RemoteButton(EntertainmentDevice newDevice)
    public TVRemoteMute(EntertainmentDevice newDevice) {
        super(newDevice);
    }

    @Override
    public void buttonNinePressed() {
        System.out.println("TV is mute");
    }

}
```



```java
public class TVRemotePause extends RemoteButton{

    public TVRemotePause(EntertainmentDevice newDevice) {
        super(newDevice);
    }

    @Override
    public void buttonNinePressed() {
        System.out.println("TV is pause");
    }

}
```

(5) 應用



```java
 * The bridge pattern is a design pattern used in software engineering
public class TestTheRemote {

    public static void main(String[] args){
        RemoteButton theTV = new TVRemoteMute(new TVDevice(25, 100));

        RemoteButton theTV2 = new TVRemotePause(new TVDevice(8, 150));

        System.out.println("Test TV with mute");
        theTV.buttonFivePressed();
        theTV.buttonSixPressed();
        theTV.buttonNinePressed();
        theTV.deviceFeedback();

        System.out.println("\nTest TV with pause");
        theTV2.buttonFivePressed();
        theTV2.buttonSixPressed();
        theTV2.buttonSixPressed();
        theTV2.buttonSixPressed();
        theTV2.buttonNinePressed();
        theTV2.deviceFeedback();
    }
}
```

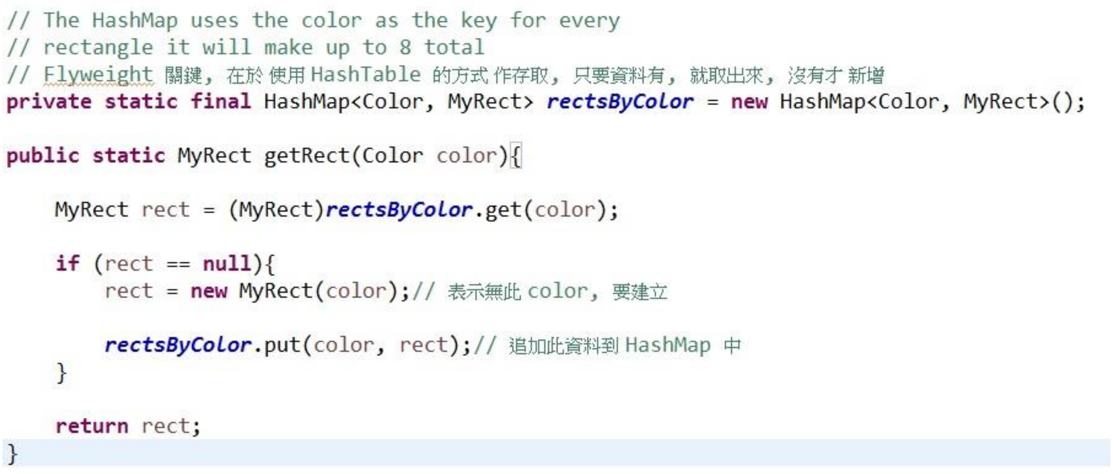
<terminated> TestTheRemote [Java Ap

```
Test TV with mute
Channel Down
Channel up
TV is mute
Channel on :25

Test TV with pause
Channel Down
Channel up
Channel up
Channel up
TV is pause
Channel on :10
```

3. Flyweight Pattern

使用時機 ：許多物件都會包含同一物件許多次的時候使用時候，利用物件共享的方式以節省實體空間

關鍵技術 ：使用 HashTable 的方式 作存取，這邊建議用 Generic！因為都是同一個 class，只要資料有，就取出來，沒有才 新增



```java
// The HashMap uses the color as the key for every
// rectangle it will make up to 8 total
// Flyweight 關鍵，在於 使用 HashTable 的方式 作存取，只要資料有，就取出來，沒有才 新增
private static final HashMap<Color, MyRect> rectsByColor = new HashMap<Color, MyRect>();

public static MyRect getRect(Color color){

    MyRect rect = (MyRect)rectsByColor.get(color);

    if (rect == null){
        rect = new MyRect(color);// 表示無此 color, 要建立

        rectsByColor.put(color, rect);// 追加此資料到 HashMap 中
    }

    return rect;
}
```

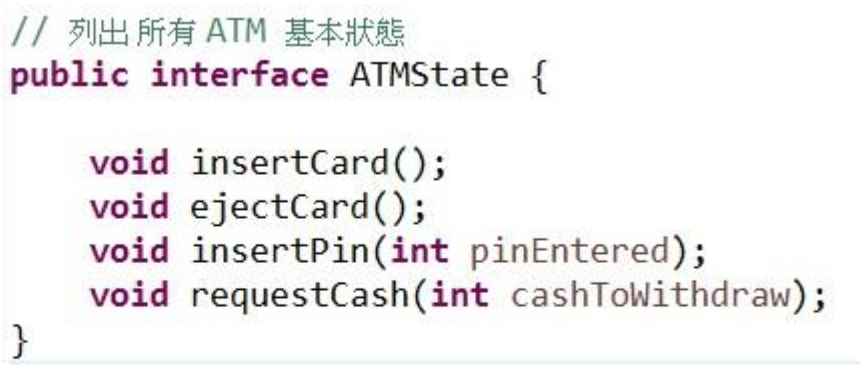
4. State Pattern

使用時機: 減少使用一個很大的 switch 或一連串的 if else

關鍵技術 ：使用 overloading 方法，且回傳值 都是相同的 型別

範例 ：ATM 狀態

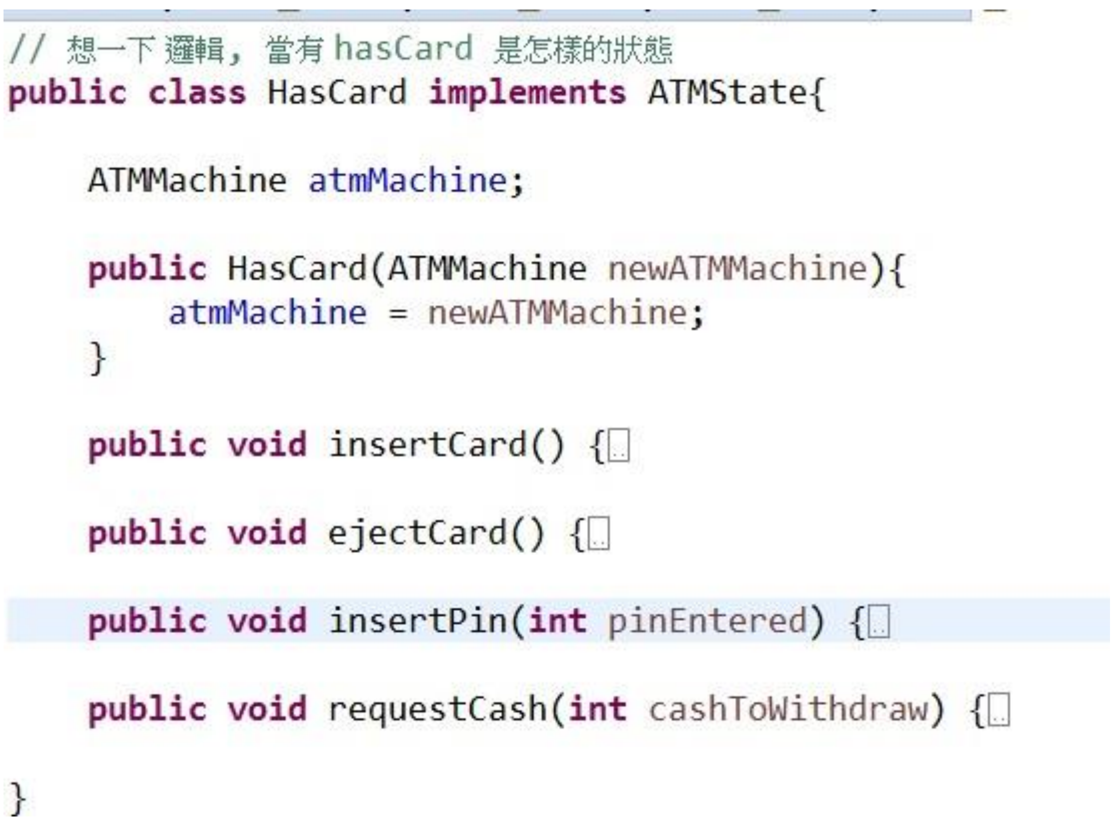
(1) ATM state



```java
// 列出 所有 ATM 基本狀態
public interface ATMState {

    void insertCard();
    void ejectCard();
    void insertPin(int pinEntered);
    void requestCash(int cashToWithdraw);
}
```

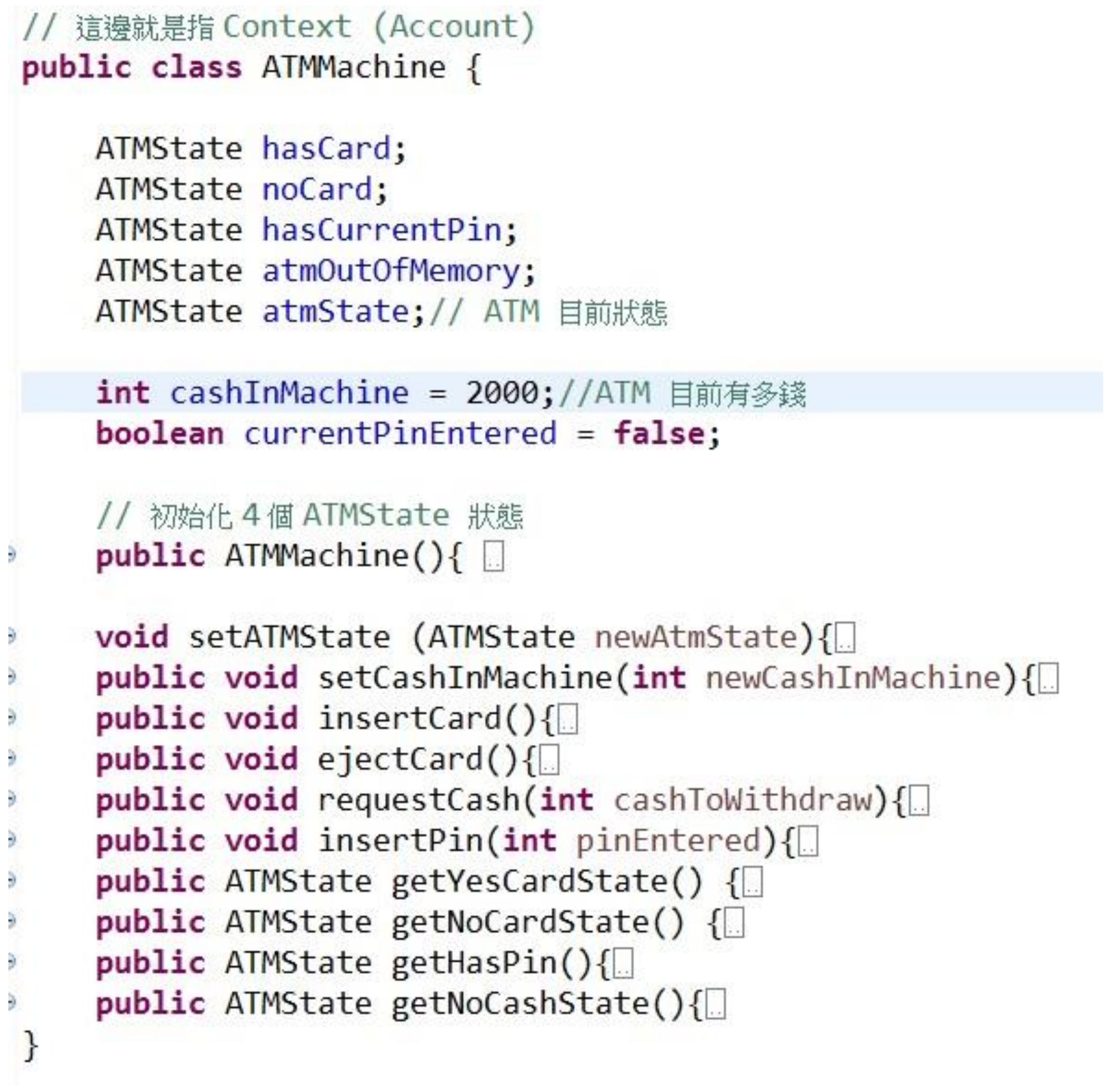
(2) 不同的情況，有不同狀態，假設這邊有 4 個情況

HasCard, HasPin, NoCard 和 NoCash 基本上都是 繼承 ATMState 只是實作狀態不太一樣而已



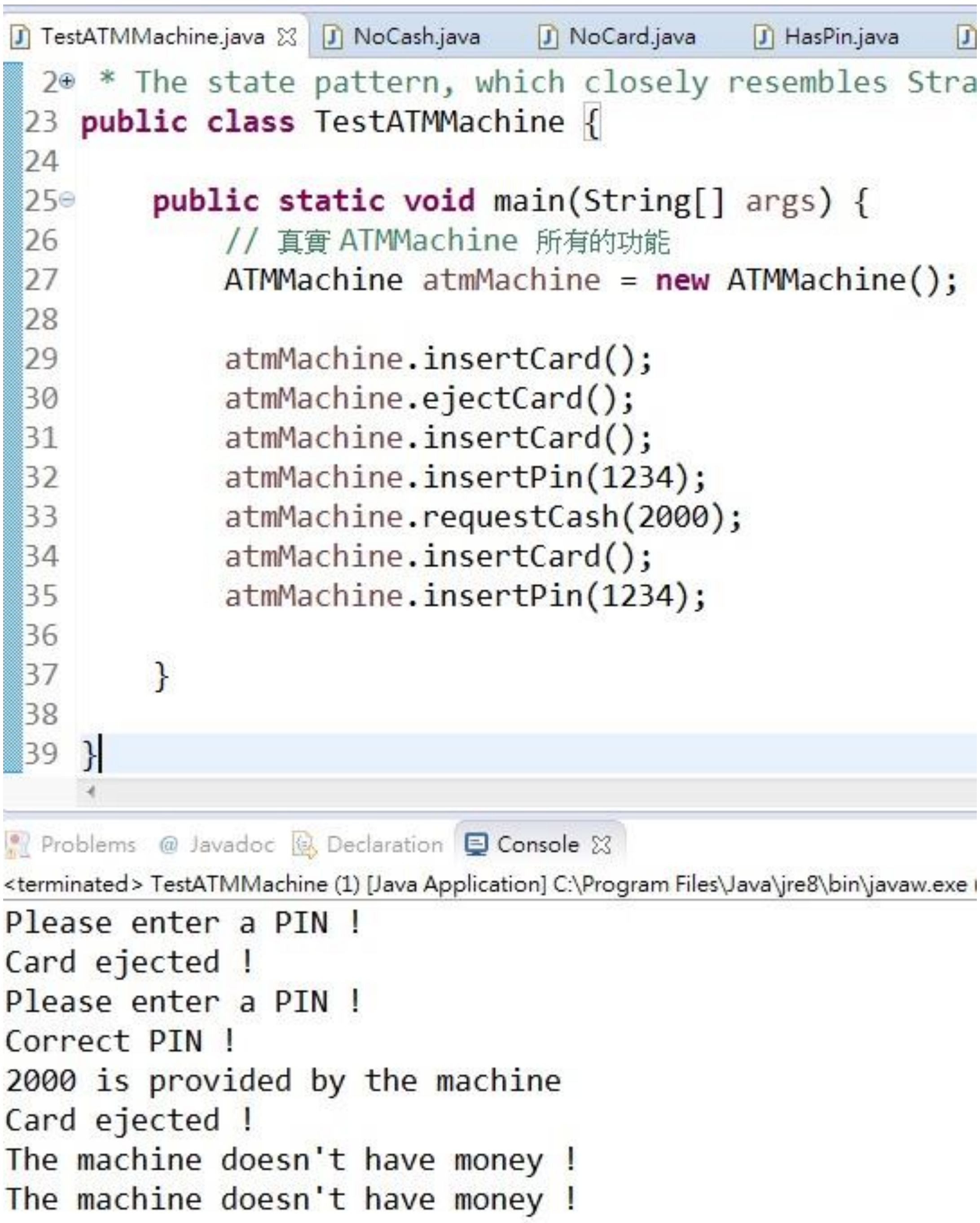
```java
// 想一下 邏輯, 當有 hasCard 是怎樣的狀態
public class HasCard implements ATMState{

    ATMMachine atmMachine;

    public HasCard(ATMMachine newATMMachine){
        atmMachine = newATMMachine;
    }

    public void insertCard() {□

    public void ejectCard() {□

    public void insertPin(int pinEntered) {□

    public void requestCash(int cashToWithdraw) {□

}
```

(3) ATM machine



```java
// 這邊就是指 Context (Account)
public class ATMMachine {

    ATMState hasCard;
    ATMState noCard;
    ATMState hasCurrentPin;
    ATMState atmOutOfMemory;
    ATMState atmState;// ATM 目前狀態

    int cashInMachine = 2000;//ATM 目前有多錢
    boolean currentPinEntered = false;

    // 初始化 4 個 ATMState 狀態
    public ATMMachine(){

    void setATMState (ATMState newAtmState){
    public void setCashInMachine(int newCashInMachine){
    public void insertCard(){
    public void ejectCard(){
    public void requestCash(int cashToWithdraw){
    public void insertPin(int pinEntered){
    public ATMState getYesCardState() {
    public ATMState getNoCardState() {
    public ATMState getHasPin(){
    public ATMState getNoCashState(){
}
```

(4) 應用



```java
 * The state pattern, which closely resembles Stra
public class TestATMMachine {

    public static void main(String[] args) {
        // 真實 ATMMachine 所有的功能
        ATMMachine atmMachine = new ATMMachine();

        atmMachine.insertCard();
        atmMachine.ejectCard();
        atmMachine.insertCard();
        atmMachine.insertPin(1234);
        atmMachine.requestCash(2000);
        atmMachine.insertCard();
        atmMachine.insertPin(1234);

    }

}
```

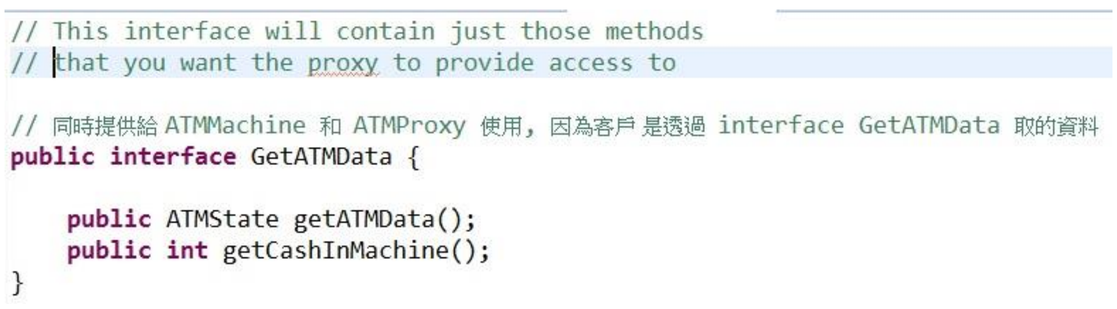
```
<terminated> TestATMMachine (1) [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe
Please enter a PIN !
Card ejected !
Please enter a PIN !
Correct PIN !
2000 is provided by the machine
Card ejected !
The machine doesn't have money !
The machine doesn't have money !
```

5. Proxy Pattern

使用時機 ： 當作其他外來物件的存取門戶,一般用來提供 security(只開放部分功能) 或 limit access 功能

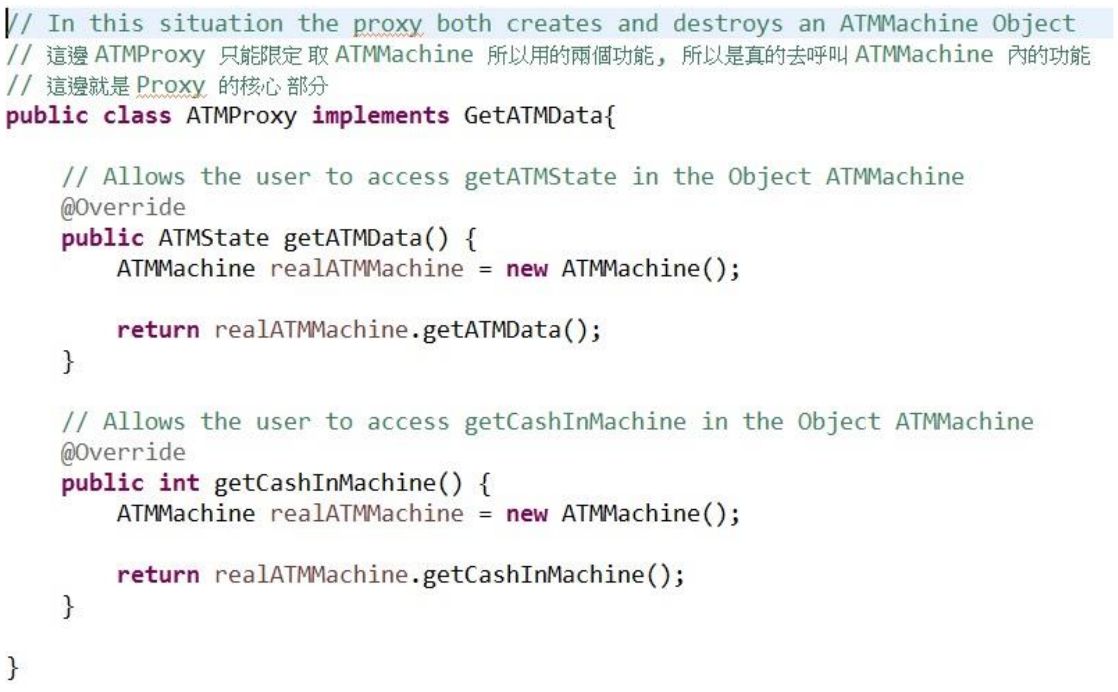
補充 : Proxy Pattern 可以和 State Pattern 互相搭配使用

(1) 定義只要開放的功能



```java
// This interface will contain just those methods
// that you want the proxy to provide access to

// 同時提供給 ATMMachine 和 ATMProxy 使用, 因為客戶 是透過 interface GetATMData 取的資料
public interface GetATMData {

    public ATMState getATMData();
    public int getCashInMachine();
}
```

(2) proxy class

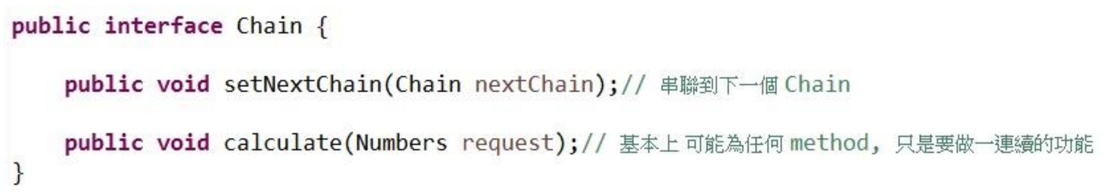


```java
// In this situation the proxy both creates and destroys an ATMMachine Object
// 這邊 ATMProxy 只能限定 取 ATMMachine 所以用的兩個功能, 所以是真的去呼叫 ATMMachine 內的功能
// 這邊就是 Proxy 的核心 部分
public class ATMProxy implements GetATMData{

    // Allows the user to access getATMState in the Object ATMMachine
    @Override
    public ATMState getATMData() {
        ATMMachine realATMMachine = new ATMMachine();

        return realATMMachine.getATMData();
    }

    // Allows the user to access getCashInMachine in the Object ATMMachine
    @Override
    public int getCashInMachine() {
        ATMMachine realATMMachine = new ATMMachine();

        return realATMMachine.getCashInMachine();
    }

}
```

6. Chain of Responsibility Pattern
   使用時機 ： 當有幾個物件 都 要求處理的需求，但處理的能範圍或權限不同, 下放到 可以處理 此物件的程序

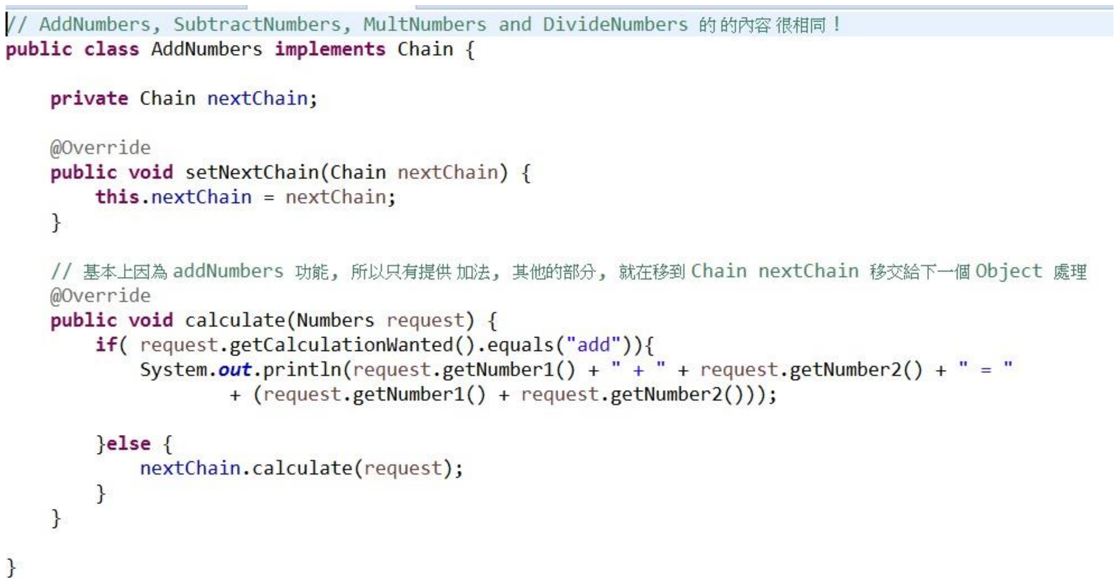
範例 ： 把 加法, 減法, 乘法 和 除法, 做成一個 Chain, 且 每一個加法, 減法, 乘法 和 除法 都有 自己的 class

(1) Chain 的介面



```java
public interface Chain {

    public void setNextChain(Chain nextChain);// 串聯到下一個 Chain

    public void calculate(Numbers request);// 基本上 可能為任何 method， 只是要做一連續的功能
}
```

(2) 加法 class, 減法, 乘法 和 除法 都採用類似作法, 唯一要注意的一點就是 ： 最後一個 chain 的不需要在呼叫 Chain

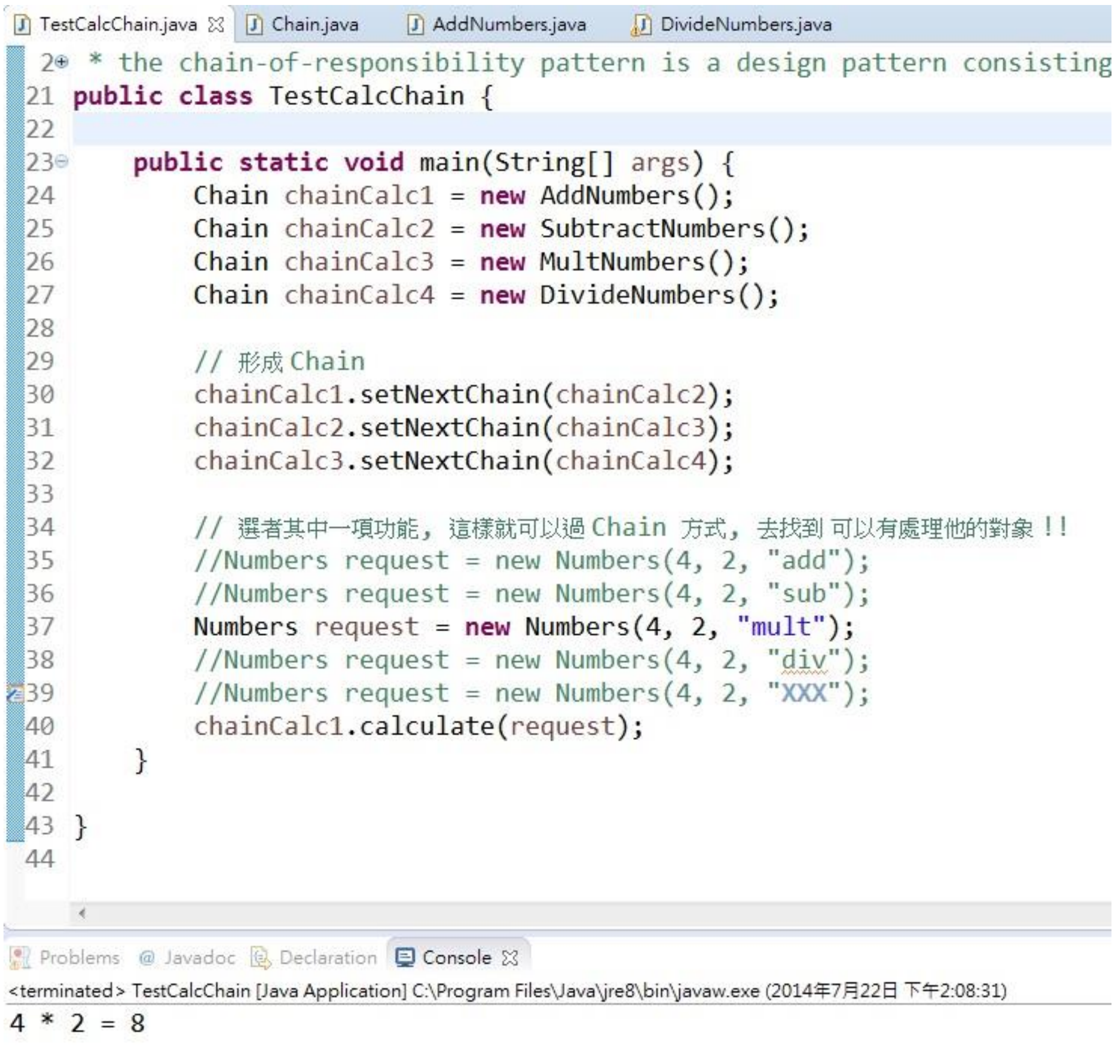


```java
// AddNumbers, SubtractNumbers, MultNumbers and DivideNumbers 的 的內容 很相同！
public class AddNumbers implements Chain {

    private Chain nextChain;

    @Override
    public void setNextChain(Chain nextChain) {
        this.nextChain = nextChain;
    }

    // 基本上因為 addNumbers 功能， 所以只有提供 加法， 其他的部分， 就在移到 Chain nextChain 移交給下一個 Object 處理
    @Override
    public void calculate(Numbers request) {
        if( request.getCalculationWanted().equals("add")){
            System.out.println(request.getNumber1() + " + " + request.getNumber2() + " = "
                    + (request.getNumber1() + request.getNumber2()));

        }else {
            nextChain.calculate(request);
        }
    }

}
```

除法(最後一個 chain 的 class, 不需要在呼叫 Chain)

```java
// AddNumbers, SubtractNumbers, MultNumbers and DivideNumbers 的的內容很相同！
public class DivideNumbers implements Chain{

    private Chain nextChain;

    @Override
    public void setNextChain(Chain nextChain) {
        this.nextChain = nextChain;

    }

    // 只有處理 除法 功能
    @Override
    public void calculate(Numbers request) {
        if( request.getCalculationWanted().equals("div")){
            System.out.println(request.getNumber1() + " / " + request.getNumber2() + " = "
                    + (request.getNumber1() / request.getNumber2()));

        }else { // 這視最後的 chain, 如果都沒有就是 無此功能
            System.out.println("only works for add, sub, mult and div");
            //nextChain.calculate(request);// 此功能不需要
        }
    }
}
```

(2) 應用



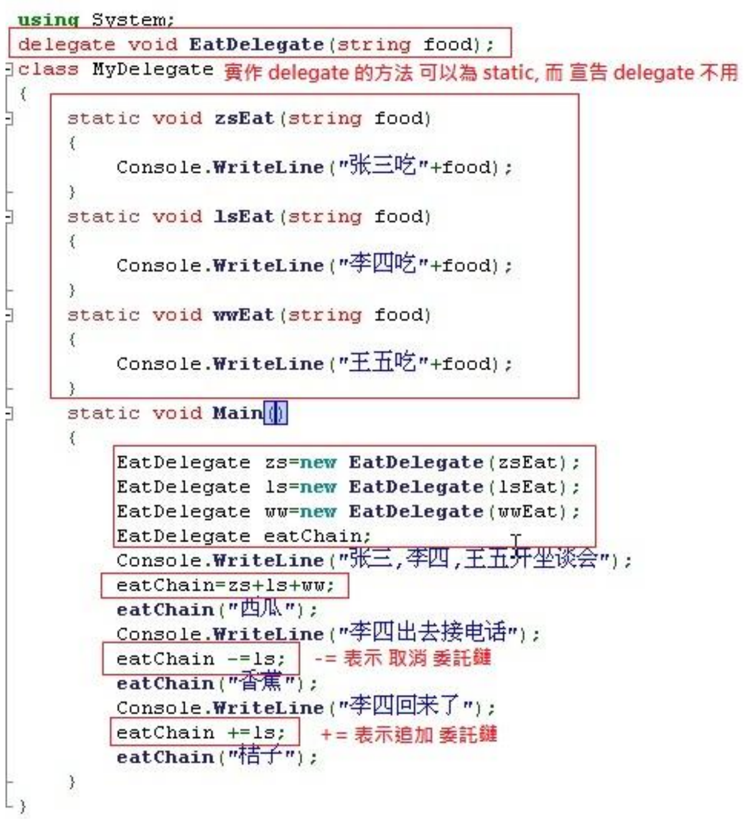
TestCalcChain.java   Chain.java   AddNumbers.java   DivideNumbers.java

```java
 * the chain-of-responsibility pattern is a design pattern consisting
public class TestCalcChain {

    public static void main(String[] args) {
        Chain chainCalc1 = new AddNumbers();
        Chain chainCalc2 = new SubtractNumbers();
        Chain chainCalc3 = new MultNumbers();
        Chain chainCalc4 = new DivideNumbers();

        // 形成 Chain
        chainCalc1.setNextChain(chainCalc2);
        chainCalc2.setNextChain(chainCalc3);
        chainCalc3.setNextChain(chainCalc4);

        // 選者其中一項功能, 這樣就可以過 Chain 方式, 去找到 可以有處理他的對象！！
        //Numbers request = new Numbers(4, 2, "add");
        //Numbers request = new Numbers(4, 2, "sub");
        Numbers request = new Numbers(4, 2, "mult");
        //Numbers request = new Numbers(4, 2, "div");
        //Numbers request = new Numbers(4, 2, "XXX");
        chainCalc1.calculate(request);
    }

}
```

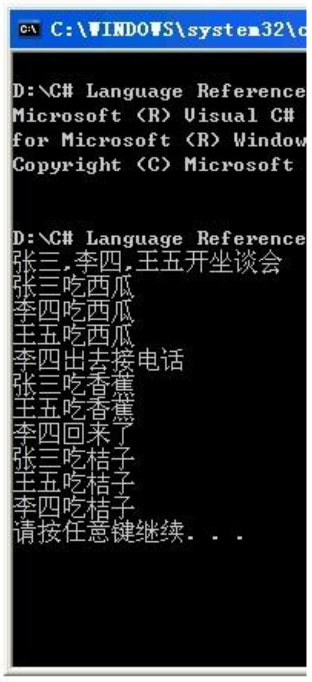
Problems @ Javadoc Declaration Console

```
<terminated> TestCalcChain [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (2014年7月22日 下午2:08:31)
4 * 2 = 8
```

補充 Chain of Responsibility 和 C# 中 委託鏈 概念很類似
但是 C# 委託鏈中用 += 就可以追加(多個執行), -= 就可以做取消
下面為 C# 委託鏈 使用範例



```
using System;
delegate void EatDelegate(string food);
class MyDelegate
{
    static void zsEat(string food)
    {
        Console.WriteLine("张三吃"+food);
    }
    static void lsEat(string food)
    {
        Console.WriteLine("李四吃"+food);
    }
    static void wwEat(string food)
    {
        Console.WriteLine("王五吃"+food);
    }
    static void Main()
    {
        EatDelegate zs=new EatDelegate(zsEat);
        EatDelegate ls=new EatDelegate(lsEat);
        EatDelegate ww=new EatDelegate(wwEat);
        EatDelegate eatChain;
        Console.WriteLine("张三,李四,王五开坐谈会");
        eatChain=zs+ls+ww;
        eatChain("西瓜");
        Console.WriteLine("李四出去接电话");
        eatChain -=ls;
        eatChain("香蕉");
        Console.WriteLine("李四回来了");
        eatChain +=ls;
        eatChain("桔子");
    }
}
```



```
C:\WINDOWS\system32\c

D:\C# Language Reference
Microsoft (R) Visual C#
for Microsoft (R) Window
Copyright (C) Microsoft

D:\C# Language Reference
张三,李四,王五开坐谈会
张三吃西瓜
李四吃西瓜
王五吃西瓜
李四出去接电话
张三吃香蕉
王五吃香蕉
李四回来了
张三吃桔子
王五吃桔子
李四吃桔子
请按任意键继续. . .
```

7. Interpreter Pattern :
使用時機 : 定義一個語言和他的文法表示, 而 Interpreter 就是用來解析 語言
範例 : IPhone 上 Siri 的語音助理
生活範例 : 問今天天氣如何 ? 系統回答 : 2014/7/22 等, 這就是 Interpreter Pattern 概念

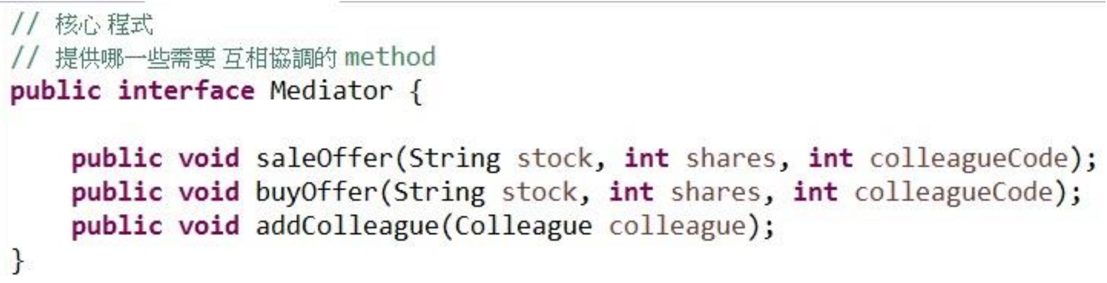
補充 : 基本上此 Pattern 使用率不高, 要做 語意 分析 很耗時

8. Mediator Pattern

使用時機 ： 解決物件之間，交錯複雜的關係，透過 Mediator(中介者) 來達到降低 物件之間的 耦合 程度

理由 ： 有時候 物件之間 太過複雜，反而 不好維護 系統

(1) Mediator 介面 (要自己定義哪些需要 method 溝通)



```
// 核心 程式
// 提供哪一些需要 互相協調的 method
public interface Mediator {

    public void saleOffer(String stock, int shares, int colleagueCode);
    public void buyOffer(String stock, int shares, int colleagueCode);
    public void addColleague(Colleague colleague);
}
```

基本上 這部分實作的程式碼差異很大，因為每個人對於降低 物件之間的耦合 的實作 有不同看法,但是 唯一不變就是 Mediator 的概念，所以不列出相關 code

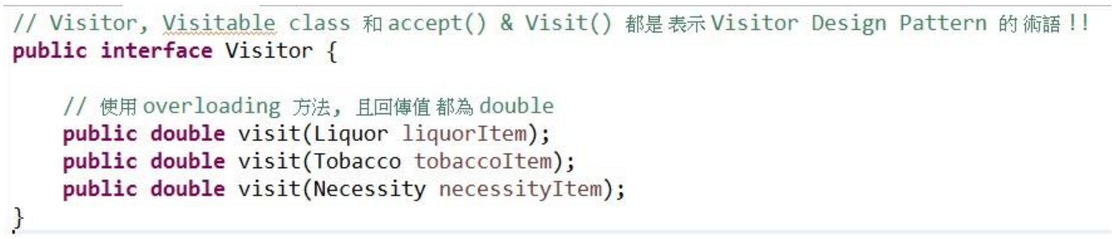
9. Visitor Pattern

使用時機 ： 封裝 某種 資料結構 的「元素」 之上的操作，但是 只有元素的行為」常會增減，但是不改變「元素」(看 source code 就知道)

關鍵技術 ： 兩個介面 Visitor, Visitable 和 overloading 概念

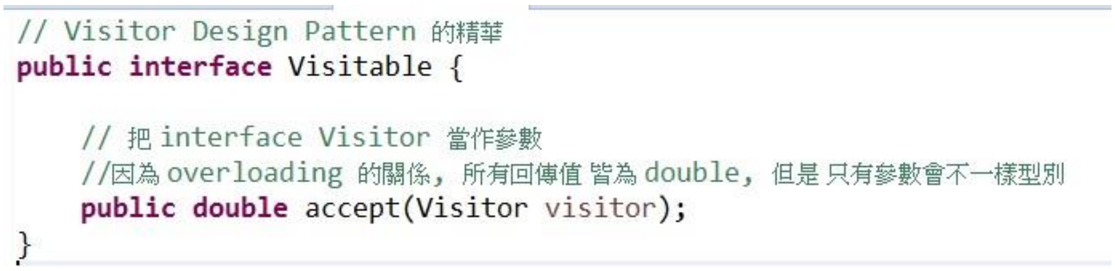
此範例 ：3 個產品(Tobacco, Liquor 和 Necessity) 在 兩個不同時間點 稅 不相同 (假日:TaxHolidayVisitor 和 平時:TaxVisitor)

(1) Visitor 介面



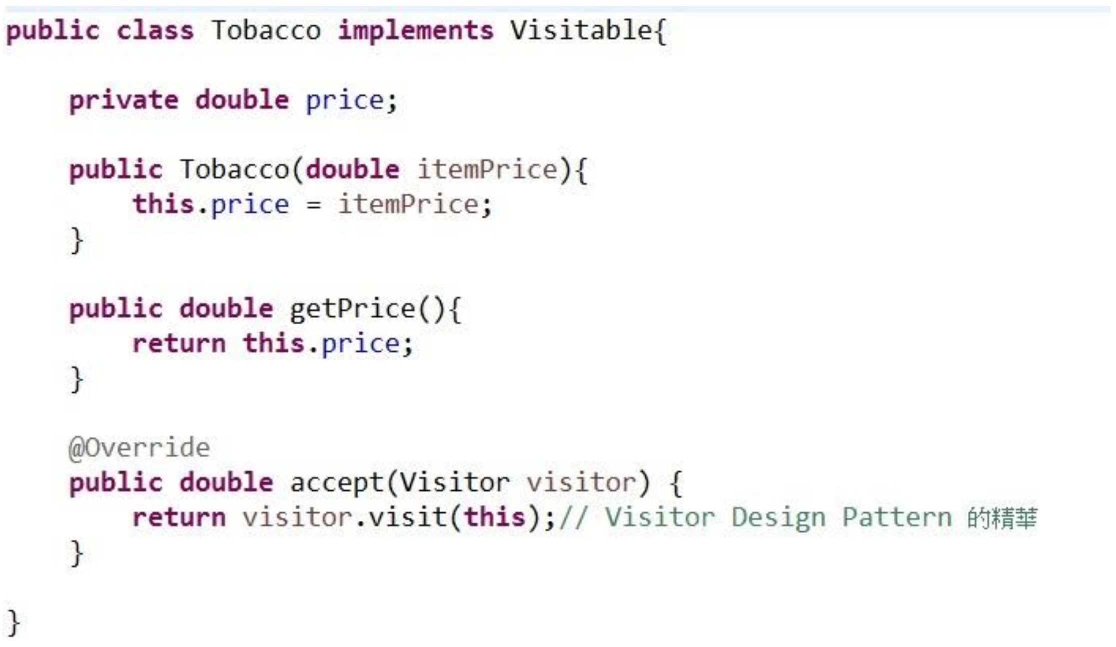
```
// Visitor, Visitable class 和 accept() & Visit() 都是 表示 Visitor Design Pattern 的 術語 !!
public interface Visitor {

    // 使用 overloading 方法, 且回傳值 都為 double
    public double visit(Liquor liquorItem);
    public double visit(Tobacco tobaccoItem);
    public double visit(Necessity necessityItem);
}
```

(2) visitable (設計精華)



```
// Visitor Design Pattern 的精華
public interface Visitable {

    // 把 interface Visitor 當作參數
    //因為 overloading 的關係, 所有回傳值 皆為 double, 但是 只有參數會不一樣型別
    public double accept(Visitor visitor);
}
```

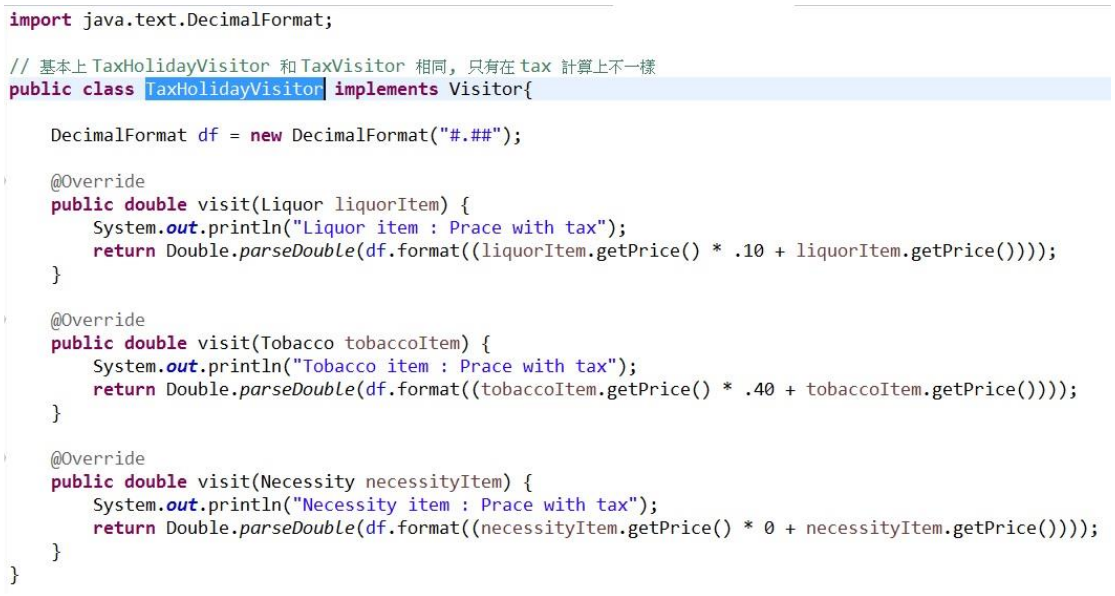
(3) 3 個產品 class, 繼承 Visitable (只列出 Tobacco, 因為剩下兩個相同)



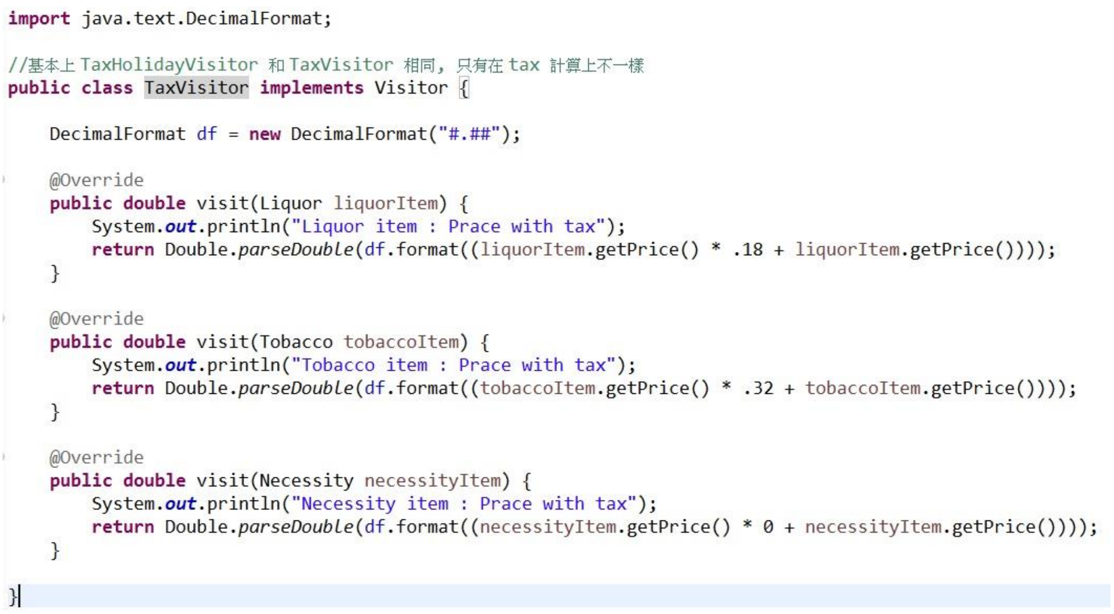
```java
public class Tobacco implements Visitable{

    private double price;

    public Tobacco(double itemPrice){
        this.price = itemPrice;
    }

    public double getPrice(){
        return this.price;
    }

    @Override
    public double accept(Visitor visitor) {
        return visitor.visit(this);// Visitor Design Pattern 的精華
    }

}
```

(4) 注意 比對 TaxHolidayVisitor 和 TaxVisitor 的不同，但是都繼承 Visitor : 封裝 某種 資料結構 的「元素」 之上的操作，但是 只有元素的行為」常會增減，但是不改變「元素」

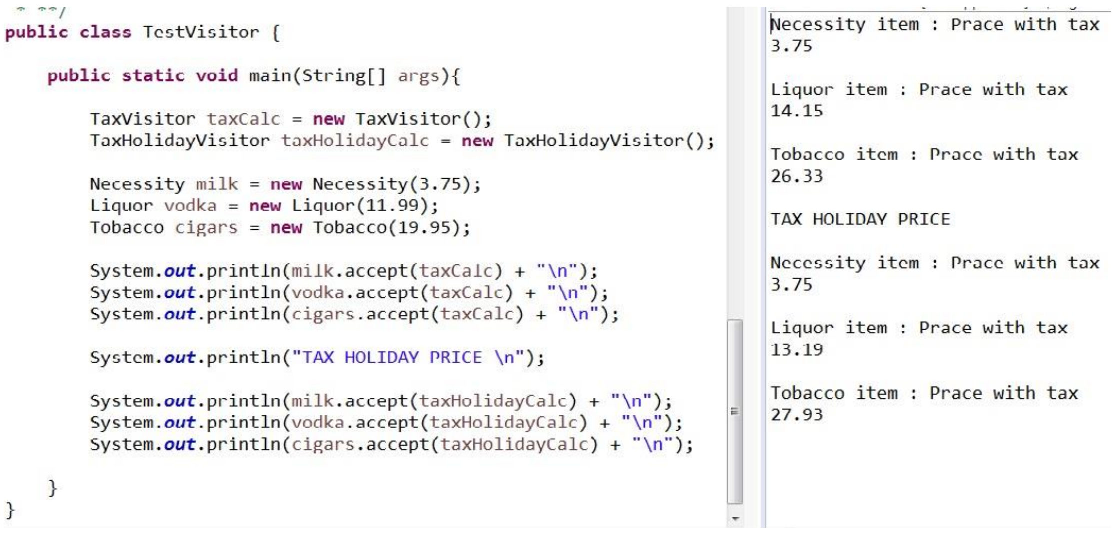
TaxHolidayVisitor


```
import java.text.DecimalFormat;

// 基本上 TaxHolidayVisitor 和 TaxVisitor 相同，只有在 tax 計算上不一樣
public class TaxHolidayVisitor implements Visitor{

    DecimalFormat df = new DecimalFormat("#.##");

    @Override
    public double visit(Liquor liquorItem) {
        System.out.println("Liquor item : Prace with tax");
        return Double.parseDouble(df.format((liquorItem.getPrice() * .10 + liquorItem.getPrice())));
    }

    @Override
    public double visit(Tobacco tobaccoItem) {
        System.out.println("Tobacco item : Prace with tax");
        return Double.parseDouble(df.format((tobaccoItem.getPrice() * .40 + tobaccoItem.getPrice())));
    }

    @Override
    public double visit(Necessity necessityItem) {
        System.out.println("Necessity item : Prace with tax");
        return Double.parseDouble(df.format((necessityItem.getPrice() * 0 + necessityItem.getPrice())));
    }
}
```

TaxVisitor


```
import java.text.DecimalFormat;

//基本上 TaxHolidayVisitor 和 TaxVisitor 相同，只有在 tax 計算上不一樣
public class TaxVisitor implements Visitor {

    DecimalFormat df = new DecimalFormat("#.##");

    @Override
    public double visit(Liquor liquorItem) {
        System.out.println("Liquor item : Prace with tax");
        return Double.parseDouble(df.format((liquorItem.getPrice() * .18 + liquorItem.getPrice())));
    }

    @Override
    public double visit(Tobacco tobaccoItem) {
        System.out.println("Tobacco item : Prace with tax");
        return Double.parseDouble(df.format((tobaccoItem.getPrice() * .32 + tobaccoItem.getPrice())));
    }

    @Override
    public double visit(Necessity necessityItem) {
        System.out.println("Necessity item : Prace with tax");
        return Double.parseDouble(df.format((necessityItem.getPrice() * 0 + necessityItem.getPrice())));
    }

}
```

(5) 應用



```
 * **/
public class TestVisitor {

    public static void main(String[] args){

        TaxVisitor taxCalc = new TaxVisitor();
        TaxHolidayVisitor taxHolidayCalc = new TaxHolidayVisitor();

        Necessity milk = new Necessity(3.75);
        Liquor vodka = new Liquor(11.99);
        Tobacco cigars = new Tobacco(19.95);

        System.out.println(milk.accept(taxCalc) + "\n");
        System.out.println(vodka.accept(taxCalc) + "\n");
        System.out.println(cigars.accept(taxCalc) + "\n");

        System.out.println("TAX HOLIDAY PRICE \n");

        System.out.println(milk.accept(taxHolidayCalc) + "\n");
        System.out.println(vodka.accept(taxHolidayCalc) + "\n");
        System.out.println(cigars.accept(taxHolidayCalc) + "\n");

    }
}
```

```
Necessity item : Prace with tax
3.75

Liquor item : Prace with tax
14.15

Tobacco item : Prace with tax
26.33

TAX HOLIDAY PRICE

Necessity item : Prace with tax
3.75

Liquor item : Prace with tax
13.19

Tobacco item : Prace with tax
27.93
```

10. Factory Pattern (很適合電玩設計)

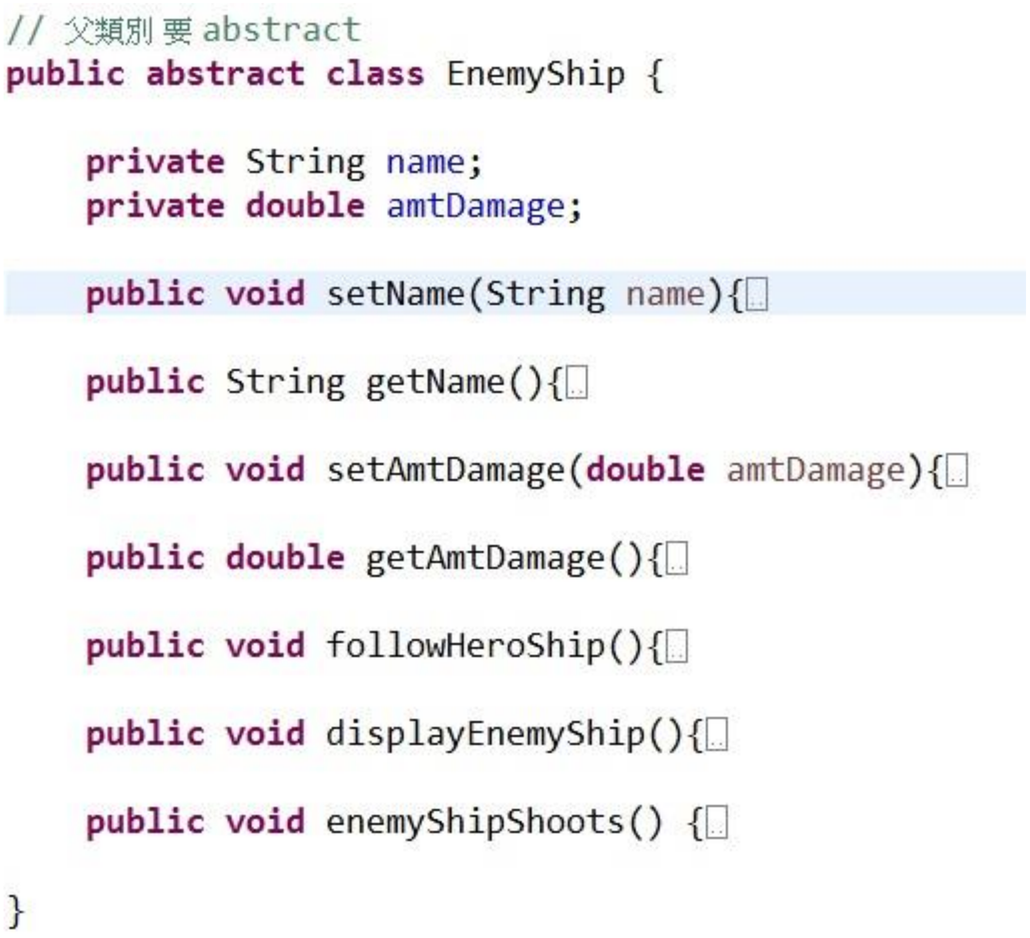
(1) 整合 Strategy Pattern 和 Template Method Pattern 使用, 關於 Strategy Pattern 和 Template Method Pattern 請參考之前 Design Pattern 文章
(2) Factory Pattern 可以在 細分為 Factory Method Pattern 和 Abstract Factory Pattern (個人認為 Abstract Factory Pattern 最複雜的其中之一)

10-1 Factory Method Pattern

使用時機 : 當 Factory 無法預期以後會需要什麼樣的 component 時，將責任延遲到 concreteProcuct(子類別) 身上

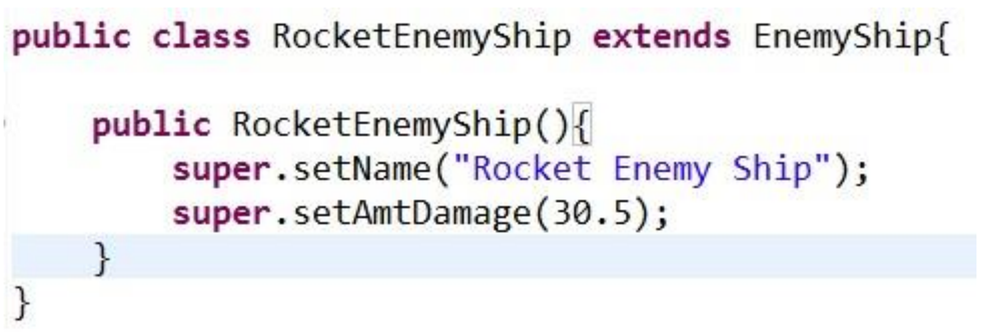
範例 :(電玩設計)三種戰艦 RocketEnemyShip, UFOEnemyShip, BigUFOEnemyShip

(1) 三個 EnemyShip 的 父類別, 必為 abstract class (Template Method Pattern)



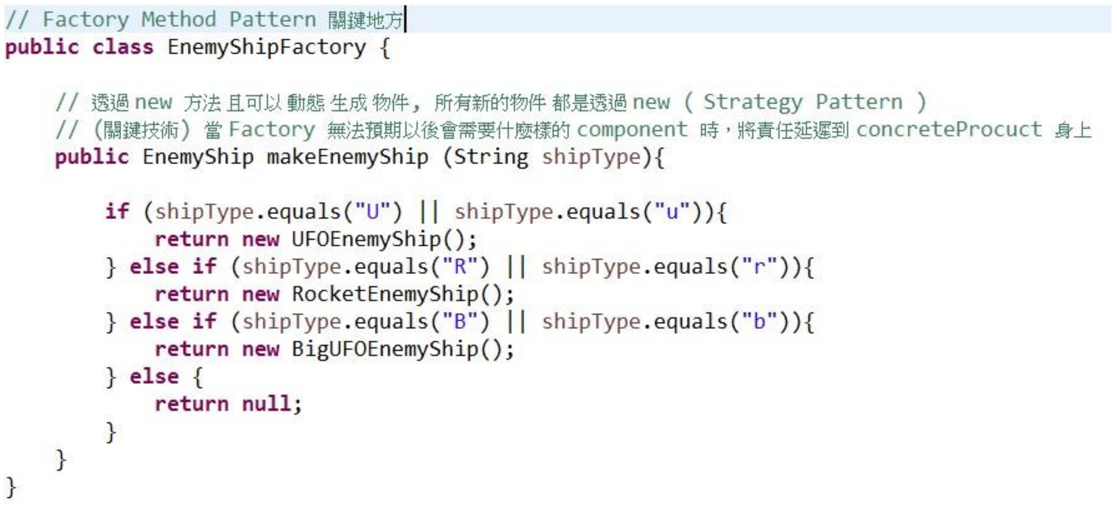
```java
// 父類別 要 abstract
public abstract class EnemyShip {

    private String name;
    private double amtDamage;

    public void setName(String name){□

    public String getName(){□

    public void setAmtDamage(double amtDamage){□

    public double getAmtDamage(){□

    public void followHeroShip(){□

    public void displayEnemyShip(){□

    public void enemyShipShoots() {□

}
```

(2) 三種戰艦 : 基本上都繼承 EnemyShip 內容就只有 建構子中 setName( ) 和 setAntDamage( ) 不同, 只用 RocketEnemyShip 表示



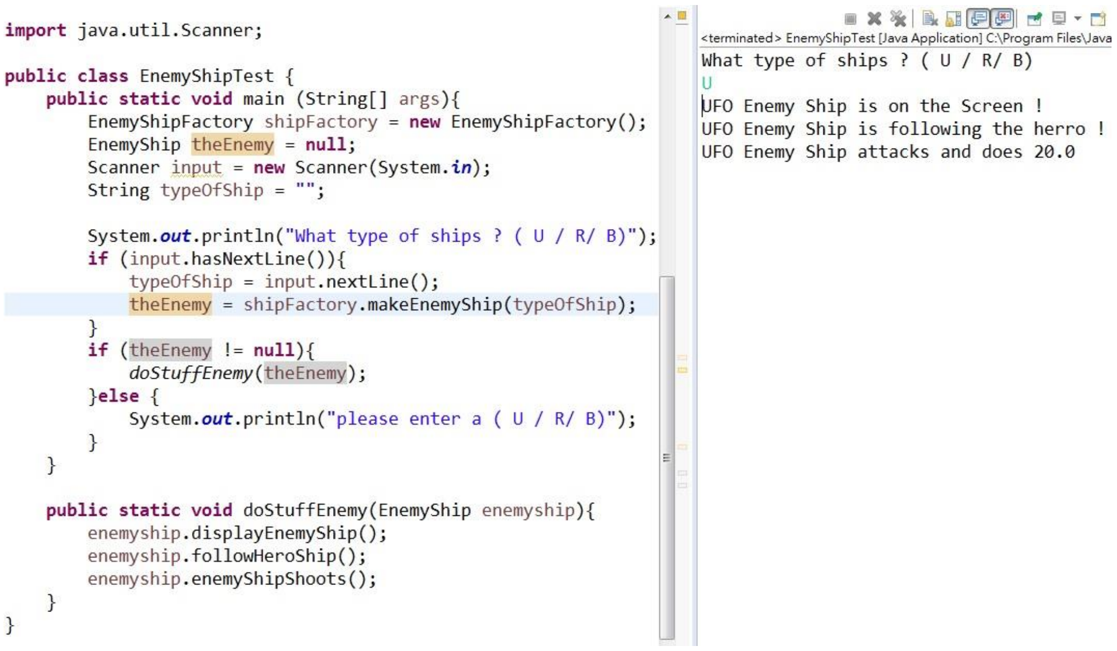
```java
public class RocketEnemyShip extends EnemyShip{

    public RocketEnemyShip(){
        super.setName("Rocket Enemy Ship");
        super.setAmtDamage(30.5);
    }
}
```

(3) Factory (關鍵技術的地方，整合 Strategy Pattern)



```java
// Factory Method Pattern 關鍵地方
public class EnemyShipFactory {

    // 透過 new 方法 且可以 動態 生成 物件，所有新的物件 都是透過 new ( Strategy Pattern )
    // (關鍵技術) 當 Factory 無法預期以後會需要什麼樣的 component 時，將責任延遲到 concreteProcuct 身上
    public EnemyShip makeEnemyShip (String shipType){

        if (shipType.equals("U") || shipType.equals("u")){
            return new UFOEnemyShip();
        } else if (shipType.equals("R") || shipType.equals("r")){
            return new RocketEnemyShip();
        } else if (shipType.equals("B") || shipType.equals("b")){
            return new BigUFOEnemyShip();
        } else {
            return null;
        }
    }
}
```

(4) 應用



```java
import java.util.Scanner;

public class EnemyShipTest {
    public static void main (String[] args){
        EnemyShipFactory shipFactory = new EnemyShipFactory();
        EnemyShip theEnemy = null;
        Scanner input = new Scanner(System.in);
        String typeOfShip = "";

        System.out.println("What type of ships ? ( U / R/ B)");
        if (input.hasNextLine()){
            typeOfShip = input.nextLine();
            theEnemy = shipFactory.makeEnemyShip(typeOfShip);
        }
        if (theEnemy != null){
            doStuffEnemy(theEnemy);
        }else {
            System.out.println("please enter a ( U / R/ B)");
        }
    }

    public static void doStuffEnemy(EnemyShip enemyship){
        enemyship.displayEnemyShip();
        enemyship.followHeroShip();
        enemyship.enemyShipShoots();
    }
}
```

```
<terminated> EnemyShipTest [Java Application] C:\Program Files\Java
What type of ships ? ( U / R/ B)
U
UFO Enemy Ship is on the Screen !
UFO Enemy Ship is following the herro !
UFO Enemy Ship attacks and does 20.0
```

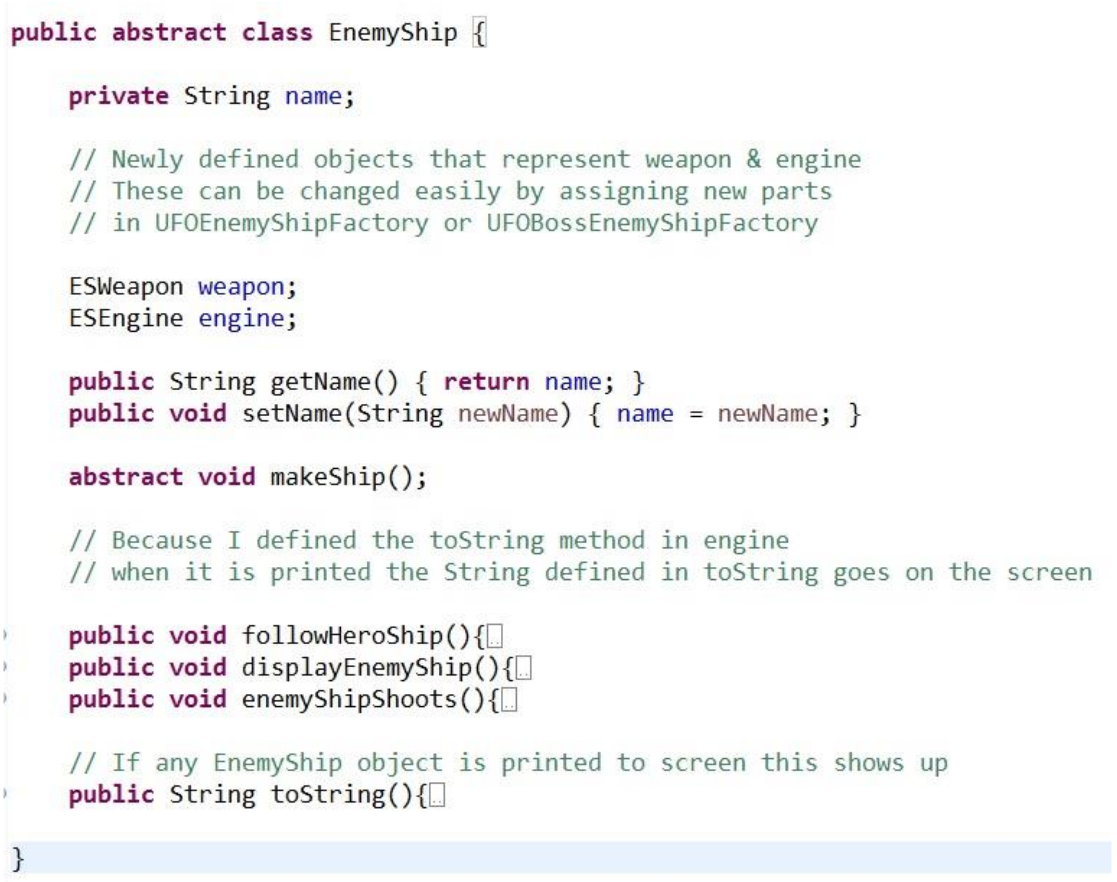
10-2 Abstract Factory Pattern
使用時機 ： 將一個 產品 把內部 元件，可以達到 隨時抽換 且可以簡單的一次抽換

更詳細解說 ： 請參考此中文網站
http://blog.monkeypotion.net/gameprog/pattern/abstract-factory

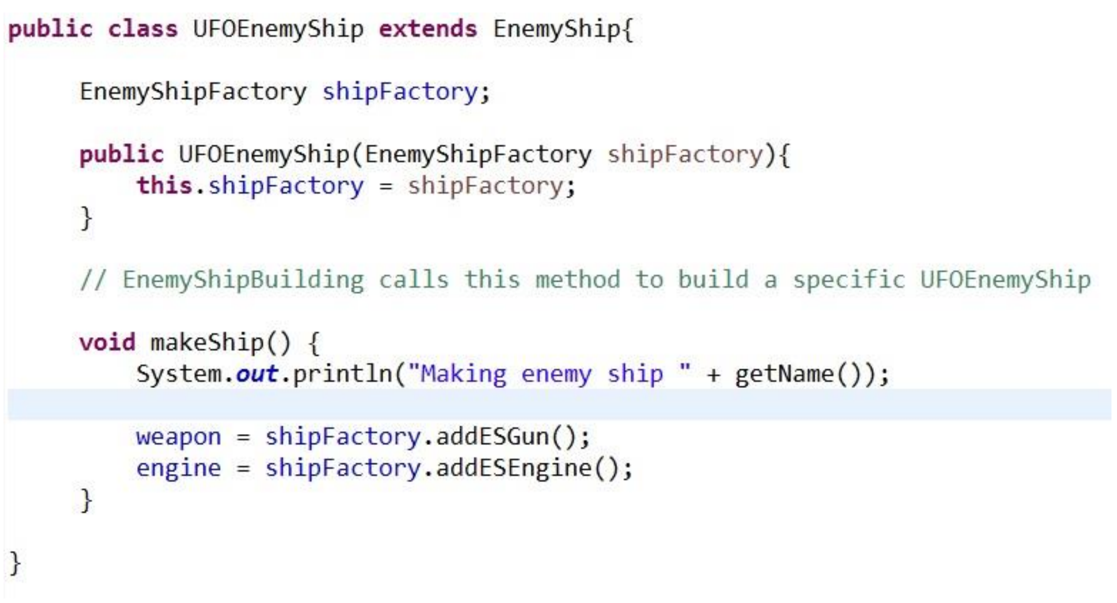
Abstract Factory 和 Factory Method 最大的不同點 ：
Abstract Factory:生成的物件，還可以 特製化 ！ 不需要 每一個都相同 !! 把產品細分化到每一個物件，之後 在透過物件 繼承 去實作 不同的 實體部分
Factory Method： 生成的物件，都是一樣的 !!

沿用 Factory Method Pattern 範例 ： 把 中的 Weapon 和 Engine 在抽出來，也就是說，同一個 UFOEnemyShip 還可以有不同的 Weapon 和 Engine 作抽換

(1) 父類別 EnemyShip (Weapon 和 Engine 在抽出來)，且把 makeShip( ) 改成抽象方法，因為預計要包含 Weapon 和 Engine 做調用

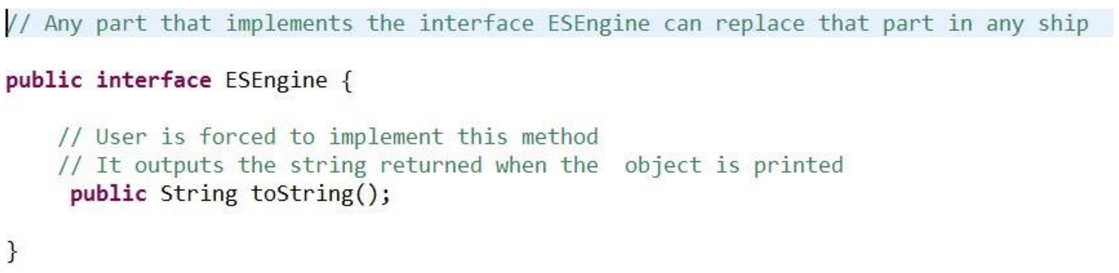

```java
public abstract class EnemyShip {

    private String name;

    // Newly defined objects that represent weapon & engine
    // These can be changed easily by assigning new parts
    // in UFOEnemyShipFactory or UFOBossEnemyShipFactory

    ESWeapon weapon;
    ESEngine engine;

    public String getName() { return name; }
    public void setName(String newName) { name = newName; }

    abstract void makeShip();

    // Because I defined the toString method in engine
    // when it is printed the String defined in toString goes on the screen

    public void followHeroShip(){
    public void displayEnemyShip(){
    public void enemyShipShoots(){

    // If any EnemyShip object is printed to screen this shows up
    public String toString(){

}
```

(2) UFOEnemyShip & UFOBossEnemyShip &都繼承 EnemyShip 內含 Factory 和 makeShip 方法, 所以只有顯示 UFOEnemyShip 內容

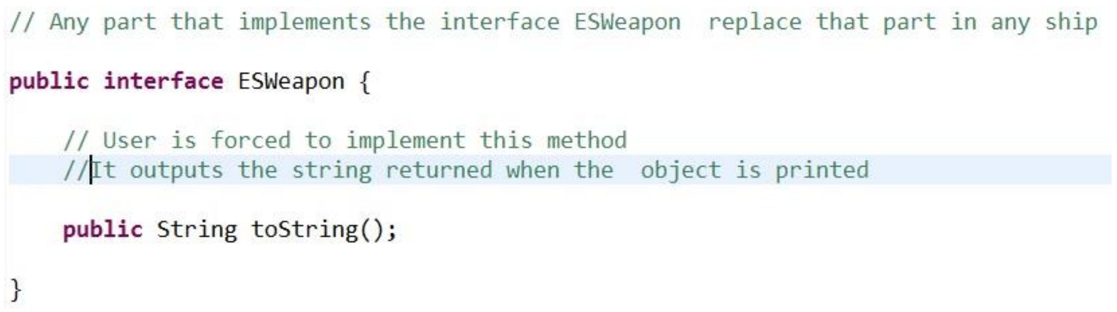


```java
public class UFOEnemyShip extends EnemyShip{

    EnemyShipFactory shipFactory;

    public UFOEnemyShip(EnemyShipFactory shipFactory){
        this.shipFactory = shipFactory;
    }

    // EnemyShipBuilding calls this method to build a specific UFOEnemyShip

    void makeShip() {
        System.out.println("Making enemy ship " + getName());

        weapon = shipFactory.addESGun();
        engine = shipFactory.addESEngine();
    }

}
```

(3) Weapon & Engine 介面

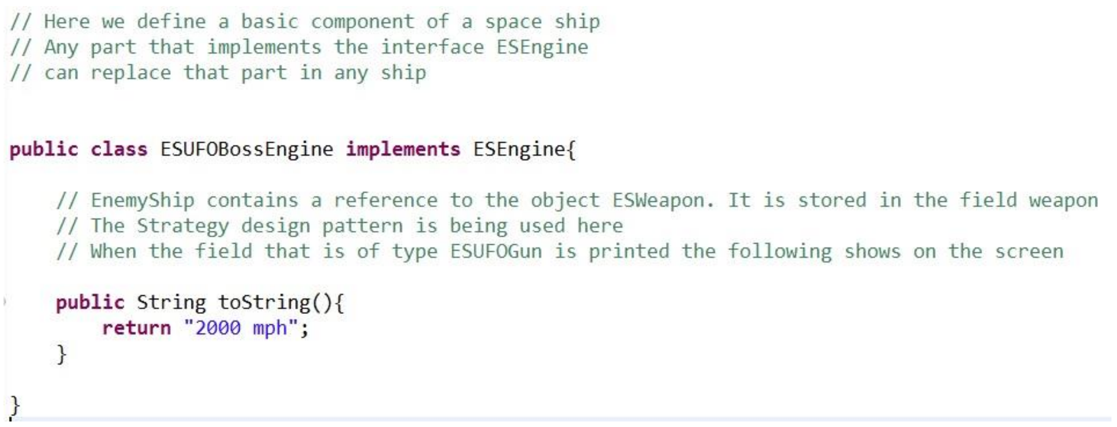


```java
// Any part that implements the interface ESEngine can replace that part in any ship

public interface ESEngine {

    // User is forced to implement this method
    // It outputs the string returned when the  object is printed
     public String toString();

}
```

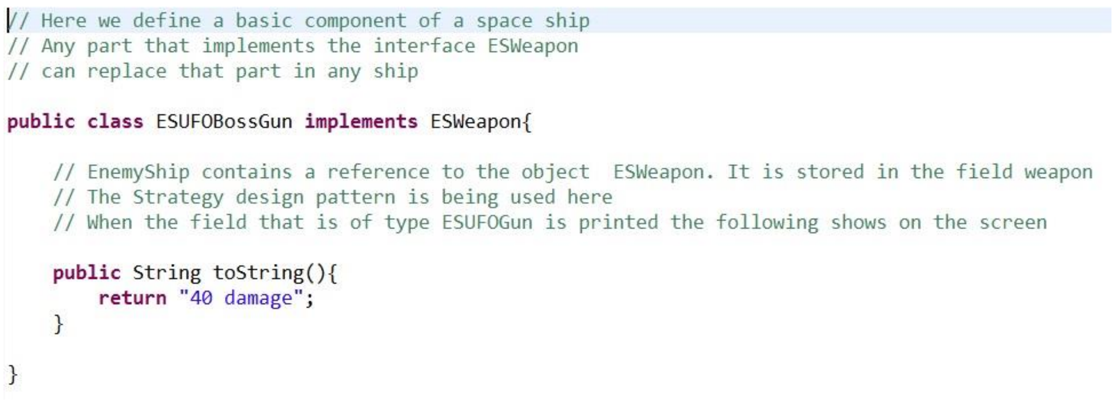


```java
// Any part that implements the interface ESWeapon  replace that part in any ship

public interface ESWeapon {

    // User is forced to implement this method
    //It outputs the string returned when the  object is printed

    public String toString();

}
```

(4) 把 UFOBossEnemyShip 中 分別 實作兩個 Weapon & Engine Interface, 另外 UFOBossEnemyShip 也需要分別 實作兩個 Weapon & Engine Interface, 所以只顯示 UFOBossEnemyShip 版本

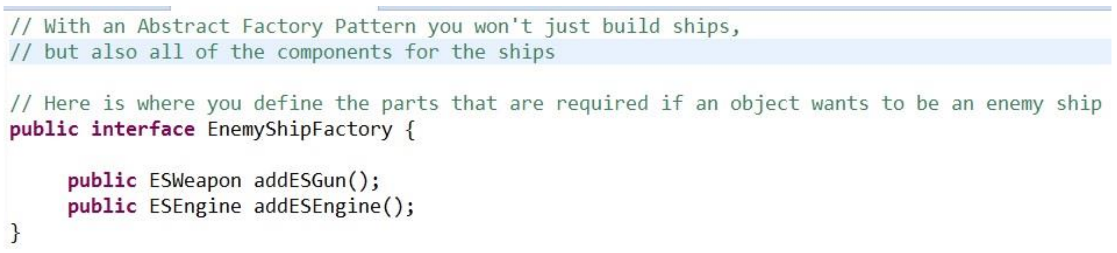


```java
// Here we define a basic component of a space ship
// Any part that implements the interface ESEngine
// can replace that part in any ship


public class ESUFOBossEngine implements ESEngine{

    // EnemyShip contains a reference to the object ESWeapon. It is stored in the field weapon
    // The Strategy design pattern is being used here
    // When the field that is of type ESUFOGun is printed the following shows on the screen

    public String toString(){
        return "2000 mph";
    }

}
```



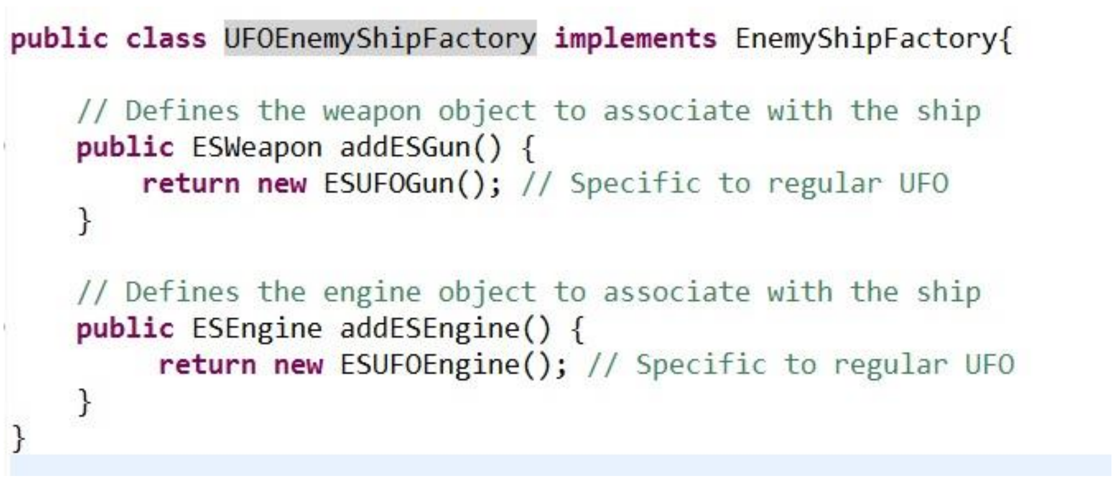
```java
// Here we define a basic component of a space ship
// Any part that implements the interface ESWeapon
// can replace that part in any ship

public class ESUFOBossGun implements ESWeapon{

    // EnemyShip contains a reference to the object  ESWeapon. It is stored in the field weapon
    // The Strategy design pattern is being used here
    // When the field that is of type ESUFOGun is printed the following shows on the screen

    public String toString(){
        return "40 damage";
    }

}
```

(7) Factory (只有限定 Weapon 和 Engine)



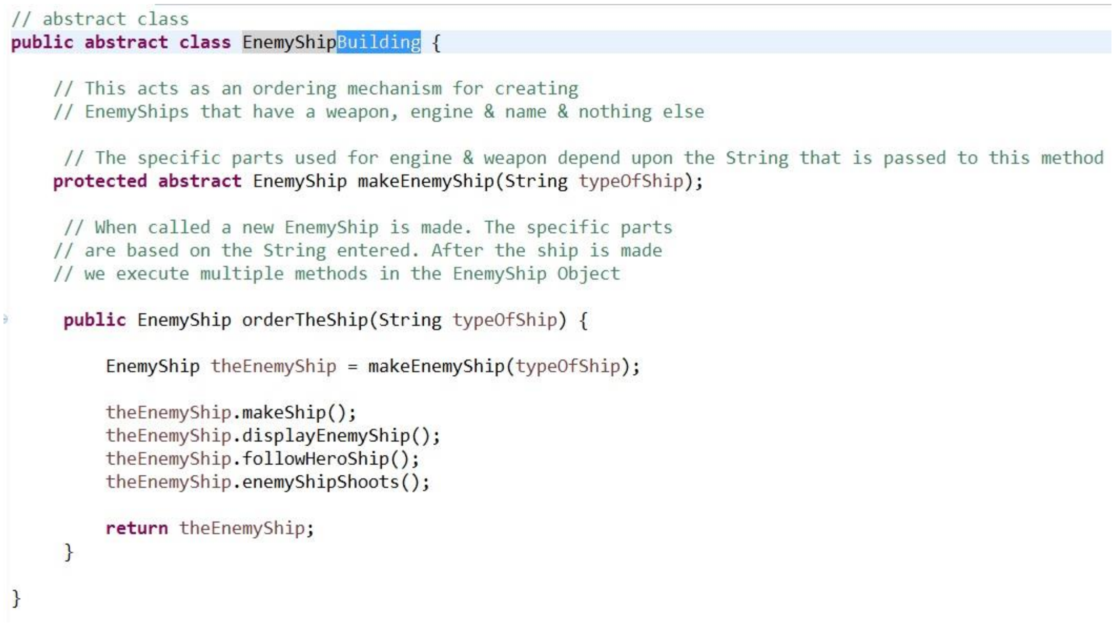
```java
// With an Abstract Factory Pattern you won't just build ships,
// but also all of the components for the ships

// Here is where you define the parts that are required if an object wants to be an enemy ship
public interface EnemyShipFactory {

     public ESWeapon addESGun();
     public ESEngine addESEngine();
}
```

(8) UFOEnemyShip & UFOBossEnemyShip 有個自 Factory, 都繼承 EnemyShipFactory, 所以只有顯示 UFOEnemyShip 版的 Factory



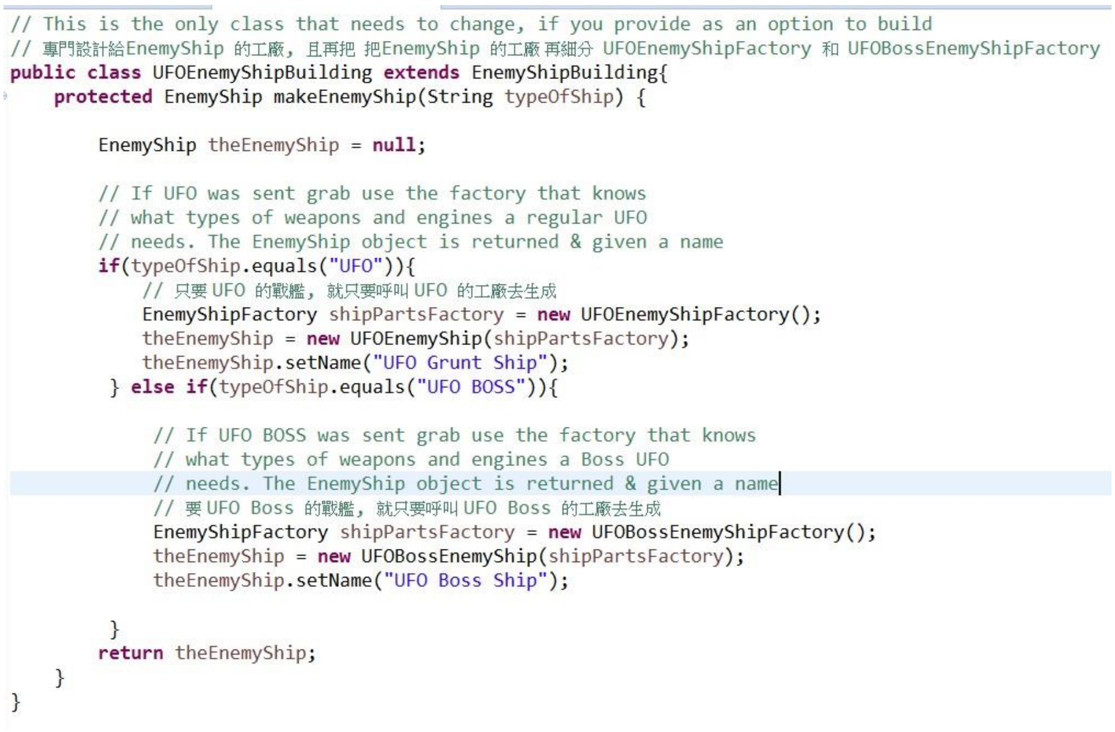
```java
public class UFOEnemyShipFactory implements EnemyShipFactory{

    // Defines the weapon object to associate with the ship
    public ESWeapon addESGun() {
        return new ESUFOGun(); // Specific to regular UFO
    }

    // Defines the engine object to associate with the ship
    public ESEngine addESEngine() {
         return new ESUFOEngine(); // Specific to regular UFO
    }
}
```

(9) EnemyShip 的設計藍圖



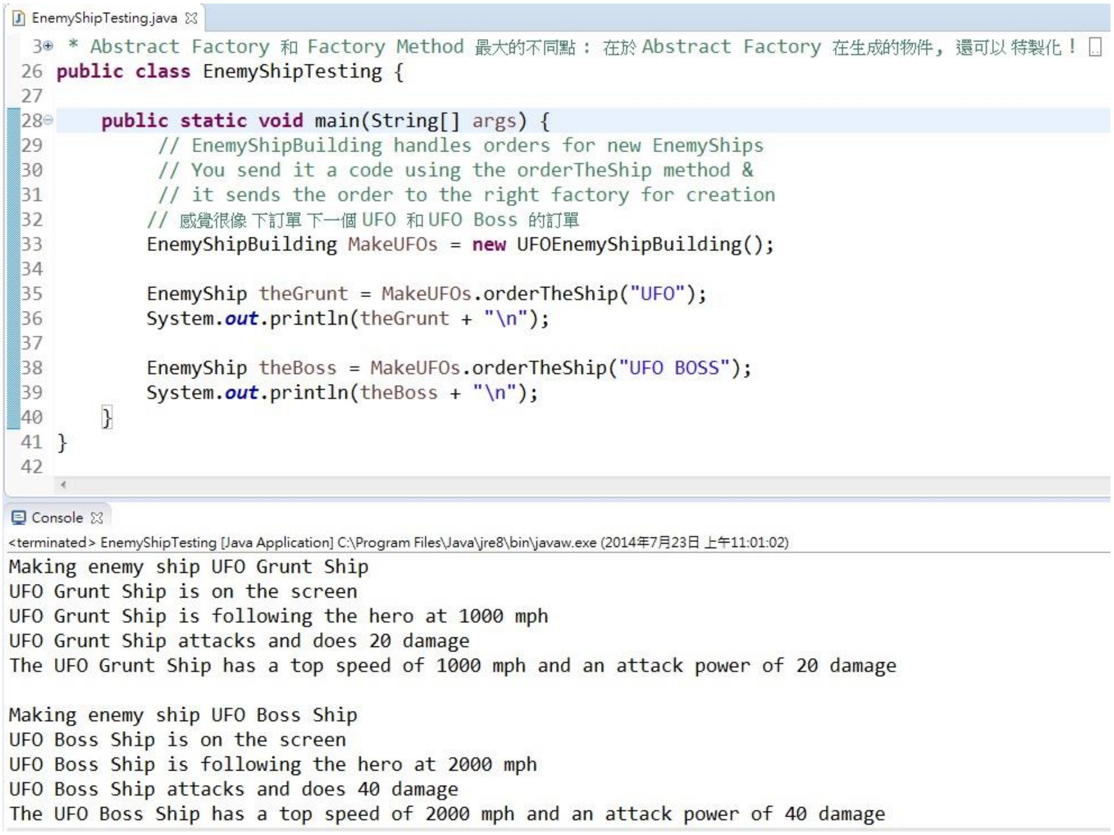
```java
// abstract class
public abstract class EnemyShipBuilding {

    // This acts as an ordering mechanism for creating
    // EnemyShips that have a weapon, engine & name & nothing else

     // The specific parts used for engine & weapon depend upon the String that is passed to this method
    protected abstract EnemyShip makeEnemyShip(String typeOfShip);

     // When called a new EnemyShip is made. The specific parts
    // are based on the String entered. After the ship is made
    // we execute multiple methods in the EnemyShip Object

     public EnemyShip orderTheShip(String typeOfShip) {

         EnemyShip theEnemyShip = makeEnemyShip(typeOfShip);

         theEnemyShip.makeShip();
         theEnemyShip.displayEnemyShip();
         theEnemyShip.followHeroShip();
         theEnemyShip.enemyShipShoots();

         return theEnemyShip;
     }

}
```

(10) 實作 EnemyShip 的設計藍圖 ： 也就是說 呼叫 UFOEnemyShip & UFOBossEnemyShip 有個自 Factory



```java
// This is the only class that needs to change, if you provide as an option to build
// 專門設計給EnemyShip 的工廠，且再把 把EnemyShip 的工廠 再細分 UFOEnemyShipFactory 和 UFOBossEnemyShipFactory
public class UFOEnemyShipBuilding extends EnemyShipBuilding{
    protected EnemyShip makeEnemyShip(String typeOfShip) {

        EnemyShip theEnemyShip = null;

        // If UFO was sent grab use the factory that knows
        // what types of weapons and engines a regular UFO
        // needs. The EnemyShip object is returned & given a name
        if(typeOfShip.equals("UFO")){
            // 只要 UFO 的戰艦，就只要呼叫 UFO 的工廠去生成
            EnemyShipFactory shipPartsFactory = new UFOEnemyShipFactory();
            theEnemyShip = new UFOEnemyShip(shipPartsFactory);
            theEnemyShip.setName("UFO Grunt Ship");
         } else if(typeOfShip.equals("UFO BOSS")){

             // If UFO BOSS was sent grab use the factory that knows
             // what types of weapons and engines a Boss UFO
             // needs. The EnemyShip object is returned & given a name
             // 要 UFO Boss 的戰艦，就只要呼叫 UFO Boss 的工廠去生成
             EnemyShipFactory shipPartsFactory = new UFOBossEnemyShipFactory();
             theEnemyShip = new UFOBossEnemyShip(shipPartsFactory);
             theEnemyShip.setName("UFO Boss Ship");

         }
        return theEnemyShip;
    }
}
```

(11) 應用



EnemyShipTesting.java

```java
 * Abstract Factory 和 Factory Method 最大的不同點 : 在於 Abstract Factory 在生成的物件，還可以 特製化 !
public class EnemyShipTesting {

    public static void main(String[] args) {
         // EnemyShipBuilding handles orders for new EnemyShips
         // You send it a code using the orderTheShip method &
         // it sends the order to the right factory for creation
        // 感覺很像 下訂單 下一個 UFO 和 UFO Boss 的訂單
        EnemyShipBuilding MakeUFOs = new UFOEnemyShipBuilding();

        EnemyShip theGrunt = MakeUFOs.orderTheShip("UFO");
        System.out.println(theGrunt + "\n");

        EnemyShip theBoss = MakeUFOs.orderTheShip("UFO BOSS");
        System.out.println(theBoss + "\n");
    }
}
```

Console

```
<terminated> EnemyShipTesting [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (2014年7月23日 上午11:01:02)
Making enemy ship UFO Grunt Ship
UFO Grunt Ship is on the screen
UFO Grunt Ship is following the hero at 1000 mph
UFO Grunt Ship attacks and does 20 damage
The UFO Grunt Ship has a top speed of 1000 mph and an attack power of 20 damage

Making enemy ship UFO Boss Ship
UFO Boss Ship is on the screen
UFO Boss Ship is following the hero at 2000 mph
UFO Boss Ship attacks and does 40 damage
The UFO Boss Ship has a top speed of 2000 mph and an attack power of 40 damage
```

總結 Abstract Factory 的程式碼 ：

1. Weapon & Engine 都有個自的 實作 class 對應到 UFOEnemyShip 和 UFOBossEnemyShip
2. 另外 UFOEnemyShip 和 UFOBossEnemyShip 都又自己專屬的 factory (因為要針對 元件作 特製化) ，這點就和 Factory Method 不同, 在 Factory Method 中大家都只有共用一個 factory