

# 物件導向軟體工程

(自製上課講義，請勿散播)

薛念林  
逢甲大學資訊工程系

December 21, 2015





# Contents

<b>1 蹤馬步：物件導向程式設計</b>	<b>9</b>
1.1 萬相皆物：類別	10
1.2 陰隱陽顯：封裝	12
1.3 虛虛實實：抽象與繼承	26
1.4 一法多形：多型	30
1.5 無色無相：介面	31
1.6 範例	35
1.7 練習	40
<b>2 圖模術：UML</b>	<b>49</b>
2.1 模組	50
2.2 UML 簡介	51
2.3 使用案例圖	54
2.4 類別圖	56
2.5 狀態圖	70
2.6 循序圖	75
2.7 程式碼對應	78
2.8 練習	83
<b>3 心法一：軟體設計原則</b>	<b>87</b>

---

3.1 涼渭分明：模組化原則	88
3.2 私財勿露：資訊隱藏原則	90
3.3 防護變異原則	91
3.4 生人勿語：迪密特原則	92
3.5 無雙無對：不重複原則	93
3.6 不變應萬變：開畢原則	95
3.7 練習	101
<b>4 心法二：物件導向設計原則</b>	<b>103</b>
4.1 異中求同：一般化原則	104
4.2 委以重任：善用包含/委託	104
4.3 空為上：善用介面	108
4.4 代父從軍：Liskov 取代原則	112
4.5 介面分割原則	115
4.6 所依皆幻：相依反轉原則	118
4.7 控制反轉原則	123
4.8 練習	124
<b>5 把脈清毒：程式碼重整</b>	<b>129</b>
5.1 把脈：21 個程式臭味	130
5.2 清毒：程式碼重整	131
5.3 練習	134
<b>6 秘笈：設計樣式</b>	<b>135</b>
6.1 簡介	136
6.2 設計樣式的分類	139
6.3 對軟體工程的協助	140

6.4 樣式的選擇與採用 . . . . .	143
6.5 工程師的武林世界 . . . . .	144
6.6 練習 . . . . .	145
<b>7 乾坤挪移：Adaptor</b>	<b>147</b>
7.1 目的與動機 . . . . .	148
7.2 結構與方法 . . . . .	149
7.3 範例 . . . . .	152
7.4 練習 . . . . .	156
<b>8 虛實分離：Bridge</b>	<b>159</b>
8.1 目的與動機 . . . . .	160
8.2 結構與方法 . . . . .	161
8.3 範例 . . . . .	164
8.4 練習 . . . . .	166
<b>9 小器晚成：Factory Method</b>	<b>169</b>
9.1 目的與動機 . . . . .	170
9.2 結構與方法 . . . . .	171
9.3 範例 . . . . .	173
9.4 練習 . . . . .	177
<b>10 一式多款：Abstract Factory</b>	<b>179</b>
10.1 目的與動機 . . . . .	180
10.2 結構與方法 . . . . .	181
10.3 範例 . . . . .	183
10.4 練習 . . . . .	188

<b>11 眾觀其變：Observer</b>	<b>191</b>
11.1 目的與動機 . . . . .	192
11.2 結構與方法 . . . . .	193
11.3 範例 . . . . .	195
11.4 練習 . . . . .	201
<b>12 獨一無二：Singleton</b>	<b>205</b>
12.1 目的與動機 . . . . .	206
12.2 結構與方法 . . . . .	206
12.3 範例 . . . . .	209
12.4 練習 . . . . .	210
<b>13 一法萬策：Strategy</b>	<b>213</b>
13.1 目的與動機 . . . . .	214
13.2 結構與方法 . . . . .	216
13.3 範例 . . . . .	218
13.4 結語 . . . . .	219
13.5 練習 . . . . .	219
<b>14 神行百變：Decorator</b>	<b>221</b>
14.1 目的與動機 . . . . .	222
14.2 結構與方法 . . . . .	226
14.3 範例 . . . . .	229
14.4 練習 . . . . .	230
<b>15 剛中帶柔：Template Method</b>	<b>233</b>
15.1 目的與動機 . . . . .	234
15.2 結構與方法 . . . . .	234

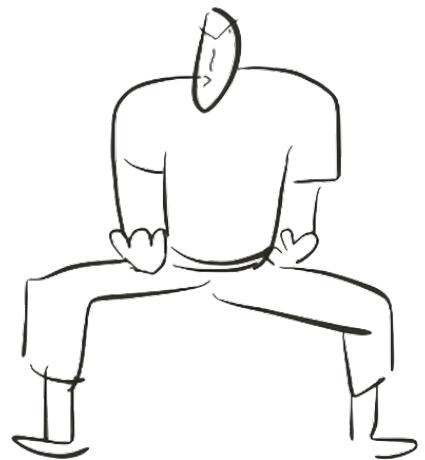
15.3 比較 . . . . .	237
15.4 練習 . . . . .	237
<b>16 三足鼎立：MVC</b>	<b>241</b>
16.1 目的與動機 . . . . .	242
16.2 範例 . . . . .	244
16.3 Web MVC . . . . .	251
16.4 練習 . . . . .	254
<b>17 隻手乾坤：Composite</b>	<b>257</b>
17.1 目的與動機 . . . . .	258
17.2 結構與方法 . . . . .	259
17.3 範例 . . . . .	260
17.4 練習 . . . . .	263
<b>18 逍遙遊：Iterator</b>	<b>267</b>
18.1 目的與動機 . . . . .	268
18.2 範例 . . . . .	270
18.3 練習 . . . . .	271
<b>19 物換星移：State</b>	<b>275</b>
19.1 目的與動機 . . . . .	275
19.2 結構與方法 . . . . .	276
19.3 範例 . . . . .	279
19.4 練習 . . . . .	280
<b>20 七星聚會：Mediator</b>	<b>283</b>
20.1 目的與動機 . . . . .	284

---

20.2 結構與方法 . . . . .	284
20.3 範例 . . . . .	287
20.4 練習 . . . . .	292
<b>21 編綿不絕：Chain of Responsibility</b>	<b>293</b>
21.1 動機與目的 . . . . .	294
21.2 結構與方法 . . . . .	294
21.3 練習 . . . . .	300
<b>22 清風拂山崗</b>	<b>301</b>
22.1 回顧 . . . . .	302
22.2 練習 . . . . .	305
<b>23 陣法：物件導向開發方法</b>	<b>307</b>
23.1 步步為營瀑布陣 . . . . .	307
23.2 先聲奪人雛形陣 . . . . .	307
23.3 反覆突破敏捷陣 . . . . .	307
<b>24 附錄</b>	<b>311</b>
24.1 ChessGame: version 1: simple . . . . .	311
24.2 ChessGame: version 2: Abstraction . . . . .	314
24.3 ChessGame: version 3: Factory method . . . . .	318
24.4 ChessGame: version 4: Strategy . . . . .	323

# Chapter 1

繫馬步：物件導向程式設計



## 1.1 萬相皆物：類別

物件導向程式設計的主體是物件類別（class），它們合作完成一個系統的運作。就像人類世界的主體是人，它們合作以完成一件工作。所以在設計物件導向程式時，你可以有點擬人化的思考，這樣有幫助你思考整個程式的架構。

以物件為抽象的中心，而非動作。例如在一個象棋系統中，以物件的角度你會先想到 Chess, ChessBoard, ChessGame 等物件，棋子的移動是附屬在 Chess 中。

`chess.move(12)`

如果你是用功能導向，一開始想到的是 move

`move(chess, 12)`

又例如傳統的行動電話設計，“撥電話”和“傳簡訊”是功能，被分別設計在手機重要選單上。後來新的手機都有“聯絡人”，把這兩個功能放到聯絡人裡面，可以說是一種物件設計的思維。

類別 vs. 物件 => 設計 vs. 實體

**物件的生成** 類別是抽象的、被設計出來的，物件則是具體被生成出來的，它在執行時具體的佔據了記憶體，可以執行程式。類別則不能執行，它是你設計出來一字字的程式碼。這就像是『“人”是抽象的，“張三李四”才是具體的人』。人是類別，張三李四則是物件。人不會跑、張三李四會跑。但上帝 (?) 設計人的時候，讓它具備了眼睛、嘴巴等特性，具備了走路跑步等動作，張三李四就依造『人』這樣的形被創造出來了。

```

1 // 透過 new 生成物件
2 Person changThree = new Person("張三", 24);

```

物件雖然有相同的方法（或演算法），但表現出來的行為不一定會一樣，因為它在執行的過程中會參考到物件本身的資料（狀態），所以表現出來的行為還是不同的。

類別通常代表一個真實事件的物體、概念、事務，通常是一個名詞。例如車子、房子等物體；老師、學生等概念。它通常是一個名詞，有一些特性，有一些能力或功能。許多寫傳統 C 習慣的人換到 OOP 常會把一個方法當成一個類別，可以先用詞態來檢驗：類別通常是名詞。

**類別也是型態 (type)**。大家應該都知道 int 是一個型態，當我們宣告一個 age 是 int 的形態，就表示它有 4 type 來儲存這個變數的值。class 也是型態喔，所以你也可以宣告一個

`age` 是一個 `Age` 的類別型態（前提是你要先設計、宣告一個 `Age` 類別）。相對於類別型態，`int`, `double`, `long` 等就稱為基本型態 (primitive type)。

**屬性與方法** 一個類別可以有很多的屬性 (attribute) 和方法 (method)。例如一個車子類別，它可以有顏色、汽缸大小等屬性；也可以有移動、停止、左轉、右轉等方法。一個類別所產生的物件，都具有相同的特性，包含屬性與方法，但內容可能不同。例如人都該有眼睛嘴巴，但大小形狀可能不同。

```

1  public class Bicycle {
2      // Bicycle 有三個屬性
3      int cadence;
4      int gear;
5      int speed;
6      // Bicycle 有四個方法
7      public void setCadence(int newValue) {
8          cadence = newValue;
9      }
10     public void setGear(int newValue) {
11         gear = newValue;
12     }
13     public void applyBrake(int decrement) {
14         speed -= decrement;
15     }
16     public void speedUp(int increment) {
17         speed += increment;
18     }
19 }
```

繼承後也可以新增屬性與方法：

```

1  public class MountainBike extends Bicycle {
2      // 繼承以後又多了一些屬性
3      public int seatHeight;
4      // 繼承以後又多了一些方法
5      public void setHeight(int newValue) {
6          seatHeight = newValue;
7      }
8  }
```

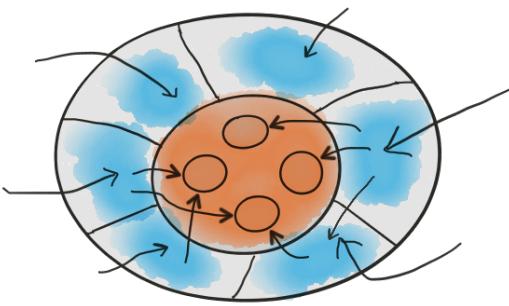


圖 1.1: 封裝

**類別命名** Java 的命名慣例是採用全名。例如學生就取為 Student, 不要叫 STD。不清楚的名字在以後很難維護，別人看不懂，你自己也看不懂。第一個字母大寫 Student, 不要 student。ChessBoard, 不要 chessboard, 不要 Chessboard, Chess\_Board...

## 1.2 陰隱陽顯：封裝

封裝討論一個類別應該具備哪些屬性、方法及它們的存取權。

### 1.2.1 成員變數

成員變數（members variables）就是剛剛說的屬性。有很多其它的名字：實體變數 instance variables、欄位 fields、或是特性 properties。成員變數表達一個類別特殊的屬性或特質。程式設計師和藝術家都是人，但前者有一個屬性「會的程式語言」，後者沒有，這就是前者的特性、屬性。又例如你要設計一個武俠遊戲，每一個人物可能都會有一個屬性「武功」來儲存它會的武功；「生命值」來表達它目前的生命力等。

成員變數的值表達「狀態」。例如棋局 class 有一個成員變數 gameStatus, 它的值可以是

- GAME\_OVER: 表達遊戲結束
- START: 表達遊戲開始
- BLACK: 表達目前由黑方下
- RED: 表達目前有白方下

每一個值表達不同的狀態。保護屬性的值，盡可能讓它維持私有。不要讓人家侵門踏戶的修改你的值，這樣這個物件的行為全亂啦。

```

1   class Bicycle {
2       private int speed; // 把 speed 告訴為私有 (private)
3   }

```

車子的速度應該由車子自己來控制，如果其它的物件隨便來亂改的話，系統很容易出錯。

想想看：宣告為 private 後誰可以修改？外面的類別如果真的需要改的時候怎麼辦？例如人要控制車速，人要怎麼修改 speed？

除了成員變數，還有其它的變數。

- 區域變數 local variable (或 method variable)。在方法內的變數。
- 參數 (parameter)：在方法內宣告的變數。

```

1   class A {
2       String name; //name 是成員變數
3       // y 是參數
4       public void m1(int y) {
5           int x; // x 是區域變數
6       }
7   }

```

**成員變數命名 Variable naming** 全名、第一個字小寫，例如：好的寫法：name, speed, numberofCar。不好的寫法：Name, \_name, NumberOfCar, number\_of\_car。

**靜態變數 Static variable (class variable)** 又稱為**類別變數**，表示是所有的物件共用的。我們如果沒有特別註明它是靜態變數，它就是非靜態變數 – 每個物件會有自己的一個空間來存變數的值。

例如 Bicycle 有一個變數 size (輪圈) 是一個非靜態變數，當我們產生 b1, b2, b3 等腳踏車時

```

1   class Bicycle {
2       private int size;
3       ...
4   }
5
6   Bicycle b1=new Bicycle(), b2=new Bicycle(), b3=new Bicycle();

```

b1, b2, b3 都有各自的空間來存 size 喔。反之，若是宣告為 static，則大家共用一份。

```

1   Class Bicycle {
2       private static int count=0;
3       ...
4       Bicycle () {
5           count++;
6       }
7   }
```

上述的程式，每當我們生成一個 bicycle，count 就會加一，所以 count 可以用來計算物件的數量。

**常數**：不僅共用，也不能修改。Math.PI 就可以取到這個值了（不需要生成物件）。

```

1   class Math {
2       static final double PI = 3.141592653589793;
3   }
```

定義常數，也是一種物件封裝的風格。不需要去記 PI 的值為多少，你會想到 Math 裡去找它，這就是好封裝的效果。

## 1.2.2 存取權

除了私有 private 以外，還有其它的存取修飾（存取權）：

- public: 所有的類別都可以存取。
- private: 只有該類別可以存取。
- package: 同一個 package 內的類別可以存取。當我們沒有寫任何 access modifier 時就是 package。
- protected: 子類別可以存取。

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
package	Y	Y	N	N
private	Y	N	N	N

**Getter, Setter** getter 一種特別的方法，專門讓其它類別來取得該屬性的值。例如

```

1  class Person {
2      private int age;
3      getAge() { //getAge 是一種 getter
4          return age;
5      }
6  }

```

**setter** 也是一種特別的方法，專門讓其它類別來設定該屬性。例如

```

1  //可以在 setter 裡面先做一些檢查
2  setAge(int _age) { //setAge 是一種 setter
3      this.age = _age;
4  }

```

想想看：有 getter, setter, 為什麼不直接宣告為 public 就好了？

### 1.2.3 方法

方法 (method) 表示一個物件的能力 (capability) 或責任 (responsibility)。能力和責任是一體兩面。在現實生活好像也是吧！是先有能力還是先有責任呢？

當我們在設計一個系統，我們定義出一些類別，指定這些類別應該具備的功能或能力。沙盤推演一下，這樣功能是否能夠滿足全部的需求？如果可以，分配下去。A 類別就必須完成這樣的功能，對它來說，是它應該具備的責任。

在程式的世界中，怎麼把這能力實作出來靠的我們所設計的演算法，也就是執行它的方法。但擬人化的說法是：能力與責任。

物件透過送訊息 (message) 來呼叫方法

```

1  //ChessBoard 送了一個訊息 move 給 chess1
2  class ChessBoard
3      play() {
4          chess1.move(2,3);
5      }

```

送訊息也是一種擬人化的說法，其實就是呼叫物件 chess1 去執行 move 的方法。

方法是一個外部物件 B 可以和本身物件 A 溝通的介面

這裡的介面是一般名詞，不是 java 中的 interface。物件 A 雖然有很多能力和方法，但未必全都會開放給外部的物件呼叫使用。A 提供了一些可呼叫的方法，其實也就是開放了一些介面，讓它物件可以存取我、可以使喚我的介面。

想像一個物件全部的方法都是 private 的（沒有溝通介面），別人都不能呼叫，這樣的物件就沒什麼用了。就像人不能孤獨的過生活不和人溝通交流吧！

**方法可能改變物件狀態** 假設一個 Date 類別，裡面有 month, date, year 三個屬性，提供 *setMonth(int)* 的介面修改月份。一旦修改 Date 的值就改變了，也就是狀態改變了。

為什麼說改變狀態 (state)，而不說改變物件的屬性值？這是因為狀態是程式設計重要的觀念。物件的行為是倚賴於狀態的，假設我們的程式邏輯是依據春夏秋冬的狀態來做事，那們 Date 的狀態就是春、夏、秋、冬：

```

1  if 春
2      耕作;
3  else if 夏
4      耘
5  else if 秋
6      收
7  else if 冬
8      藏

```

對農作的系統來說，從一月一日變成一月二日並不具意義，重點是春夏秋冬的變化。對於另一個應用程式，例如星座相關的系統，則狀態又變成 12 個星座了。所以我們在談論系統設計時，多半談**狀態**。

方法，提供了一個介面讓其它的物件來改變本身的值，也就有可能改變狀態，就會改變系統可能的反應了，不得不注意。

**方法的成員** 一個 method 可以由以下六個元素組成：存取權 (public, protected, private)、回傳型態、方法名稱、參數列、例外的形態、方法本身。

**參數** Formal parameter: 被呼叫端，如 public double myMethod(int p1, int p2, double p3)。律定了對呼叫端的要求。Actual parameter (also called argument): 呼叫端，例如 “myMethod(12, 20, 23.0)”。

**型態轉換** 當 parameter 的形態與 argument 的形態不同時，會進行 type casting (型態轉換)，如下：左邊的可轉換成右邊的，但右邊的不能轉成左邊的：

byte, short, int, long, float, double, char。

**Accessor 和 Mutator** Accessor 和 mutator 是兩種比較特殊的方法，Accessor 是獲取物件屬性值的方法，例如 `getAge()`, `getHeight()` 等。Mutator 是修改物件屬性的方法，例如 `setHeight(int)`, `setAddress(String)` 等。mutator 會把物件內部的屬性的值設為所傳進來的參數，但通常會做一些檢查，以控制修改是否合適，例如 `setHeight` 若帶進來的參數值大於 250 公分顯然不太合理，就可以拋出例外、回傳 `false` 或終止程式的執行。又稱為 Getter/Setter。

Mutator 方法也可以回傳 boolean

### 前置條件、後置條件

方法模組設計以前，先好好的想一想：這個方法的前置條件是什麼？後置條件是什麼

- 前置條件 (precondition): 功能能正常運作的條件。例如一個除法  $div(x/y)$ , 前置條件是  $y \neq 0$ 。
- 後置條件 (postcondition): 功能完成後必須滿足的條件。例如遞增排序後，必定會滿足  $a[0] \leq a[1]$  的條件。

後置條件和 invariant (不變的規則) 是不同的。Invariant 是指在此應用領域中不會改變的規則，當它被違反了，有可能就是程式出錯了。例如計算年齡的程式,  $age > 0$ ,  $age < 200$  是一個不變的規則，我們可以在程式進行過程中不斷的檢驗這樣的規則，以降低程式出錯的機率日後除錯的 effort。

```
1 assert age>0 && age <200
```

直接把屬性設為公開，和透過 accessor, modifier 來公開有什麼不同？

有了前置後置條件的觀念後，上面的問題就迎刃而解了。我們可以在 modifier 中加入前置與後置條件的檢查，這是直接修改公開屬性無法做的功能。

```
1 boolean setHeight(int h) {
2     if (h>250)
3         return false;
4     else {
5         this.height = h;
6         return true;
}
```

```

7      }
8  }
```

## 靜態方法

靜態方法（static method）又稱為類別方法（非靜態方法又稱為物件方法 instance method）。靜態方法不會存取到物件的狀態（物件的屬性值），也就是說，它的運作和物件的非靜態屬性完全沒有關係。

假設一個 Person class, 內部有一個 birthday 的屬性，及一個 getAge() 的方法：

```

1 int getAge() {
2     return (now() - birthday);
3 }
```

因為這個方法會使用到物件的屬性（非靜態屬性），所以它是 instance method。反之，假設 getAge() 是這樣設計的：

```

1 static int getAge(Date birth) {
2     return (now() - birth);
3 }
```

可以看得到它完全不會用到物件的屬性，所以可以宣告為 static method。它的呼叫方法：

`Person.getAge(new Date(12,1,1990));`

注意到它並不需要產生一個物件才去呼叫，直接用類別就可以呼叫。（為什麼？）

以下幾點注意：

- 靜態方法不可存取物件變數 (instance variable)
- 靜態方法不可呼叫物件方法 (instance method)
- 靜態方法可以呼叫靜態方法

**公用程式** 很多有用的公用程式（utility function）都被設計成靜態方法，因為它主要的功能是提供一些便捷的運算： $input \Rightarrow (computation) \Rightarrow output$ 。例如 Math 這個類別裡面提供了大量的數學運算的方法，都是靜態的：`sin(double)`, `cos(double)` 等。記得它們都是有帶參數的。

找找看，Java API 中還有哪些 utility function? Integer 這個 class, 裡面有哪些靜態方法？

`Integer.parseInt()`

**包裝類別 (wrapper class)** 把每一種基本型態包裝成一個類別型態，例如 int 包成 Integer, double 包成 Double 等。

```

1  Integer i = new Integer(20);
2  //透過 intValue 來做開箱 (unboxing)
3  int j = i.intValue();
4  Integer s = 30; //自動包裝
5  int k = i; //自動開箱

```

包裝類別內有很多的常數：例如 Integer.MAX\_VALUE, Integer.MIN\_VALUE, Double.MAX\_VALUE, Double.MIN\_VALUE 等。也包含很多的靜態方法可以把字串變成基本型態。

```

1  // parseInt 是靜態方法，把字串轉成整數
2  int i = Integer.parseInt("23");
3  double d = Double.parseDouble("23.4");
4  // 把整數轉成字串
5  String s = Integer.toString(100);

```

## 傳參數

基本上 java 都是傳值 (call by value)，所以當我們傳一個基本型態的值時

```

1  int a = 100;
2  m1(a);

```

是把 a 的值傳過去給方法 m1。m1 修改了參數 arg 的值也不會對 a 有所影響。

```

1  m1(int arg) {
2      arg = 200;
3  }

```

也就是說執行到 m1 時，會提出一個空間來存 100，讓 arg 指到這個空間。a 與 arg 有各自的空間，兩不相擾。

如果傳的是物件型態，就要注意值可能會被方法修改

```

1  Person p = new Person("Jack");
2  m2(p);

```

這時候一樣是傳值，可是 p 的值可不是 Jack 阿（物件型態的值可沒那麼單純），而是存放 p 物件的位址，假設是 10A20 好了。

```

1  setName(People p2) {
2      p2.setName("Nick");
3 }
```

這時候 p2 會被建立出來，裡面放的 p1 的值（也就是，拷貝一份 p1 給 p2），所以 p2 的值也是 10A20 了。執行 p2.setName("Nick") 時，就是要原來叫 Jack 的改名為 Nick 了，因為 p1 和 p2 都是指到同一個物件阿。

**變數只是一個物件參考、一個別名，它們可能指到同一個實體。**

所以如果你不想你的物件值受到可能的修改，記得先複製一份，再傳進去。

```

1 Person p2 = new Person(p1);
2 setName(p2);
```

p1 與 p2 有各自的空間。

## Overloading 方法

同一個類別可以有多個相同名稱的方法，只要參數不同即可。如下：

```

1 public class DataArtist {
2     ...
3     public void draw(String s) {
4         ...
5     }
6     public void draw(int i) {
7         ...
8     }
9     public void draw(double f) {
10        ...
11    }
12 }
```

注意不能方法參數一樣，只是傳回參數型態不同。下方的程式碼會導致編譯錯誤。

```

1 public void draw(double f) {
2     ...
3 }
4
```

```

5   public double draw(double f) {
6       ...
7   }

```

**任意數量的參數** 有時候我們不知道呼叫端會傳多少參數，可利用... 來允許傳任何數量的參數。

```

1  public Polygon polygonFrom(Point... corners) {
2
3      int numberofsides = corners.length;
4      double squareofSide1, lengthofSide1;
5      squareofSide1 = (corners[1].x - corners[0].x)
6          * (corners[1].x - corners[0].x)
7          + (corners[1].y - corners[0].y)
8          * (corners[1].y - corners[0].y);
9      lengthofSide1 = Math.sqrt(squareofSide1);
10     ...
11
12 }

```

記得要先用.length 來取得數量。

**應用程式介面 API** 和屬性的存取權控制一樣，method 也可以有 public, protected, package, private 等控制權。

## 1.2.4 建構子

建構子的觀念

- 也是一種方法 method
- 沒有回傳值，也不用 void
- 可以有很多不同參數的建構子
- 建構子可以用來初始化，不同參數的建構子可以做不同的初始化

```

1  class ClassName {
2      public ClassName( parameter ) {
3          ...

```

```

4      }
5  }
```

預設建構子是沒有參數的建構子。但一旦宣告了有參數的建構子，沒有參數的建構子就消失了。

```

1   class P {
2 }
```

上面的 P 並沒有宣告建構子，但你可以這樣呼叫

```
P p1 = new P();
```

因為預設有一個沒有參數的建構子。但如果你宣告了一個有參數的建構子

```

1 class P {
2     public void P(int x) {
3         ...
4     }
5 }
```

則 `p1 = new P()` 就會造成 compile 錯誤。因為我們認為設計者想要強制物件的生成一定要做某個與 x 相關的初始化，所以預設的就會被取消。當然如果你想要也可以加回來：

```

1 class P {
2     public void P(int x) {
3         ...
4     }
5     public void P() { //加上預設建構子
6         ...
7     }
8 }
```

建構子可以呼叫其它的方法。因為建構子內常常做初始化，這個動作可以交給 mutator 來做。這樣前置條件的設計可以寫在一個地方就好了。

**複製建構子** 我們可以依照我們的需求生成一個我們想要的手機，只要把相關的參數帶進去就好了。

```

1 class Phone {
2     int memory, size;
```

```

3     public Phone (int memory, int size) {
4         this.memory = memory;
5         this.size = size;
6     }
7 }
```

有時候想要複製一樣的手機，

```

1 class Phone {
2     int memory, size;
3     public Phone (Phone p) {
4         this.memory = p.memory;
5         this.size = p.size;
6     }
7 }
```

生成時只要呼叫 Phone p2 = new Phone(p1) 就可以生成一個和 p1 一模一樣的手機。上述的建構子就稱為複製建構子。它的形態是：

```

1 class ClassName {
2     public ClassName(ClassName para) {
3         ...
4     }
5 }
```

Copy constructor 很好用也很重要。設計類別時可多加利用。一個類別可以有很多的建構子。

```

1 public Bicycle(int startSpeed, int startGear) {
2     gear = startGear;
3     speed = startSpeed;
4 }
5 public Bicycle() {
6     gear = 1;
7     speed = 0;
8 }
9
10 public Bicycle(Bicycle b) {
11     gear = b.gear;
```



圖 1.2: 鋼鐵物件：屬性不可被修改的物件，所以可以大方的給別的物件參考

```

12     speed = b.speed;
13 }
```

### 1.2.5 私有外洩

我們常會把變數宣告為私有，但它真的有達到私有的效果嗎？不要以為宣告為私有就很安全了。如果是基本型態，宣告為私有就可確保不會有私有外洩。但如果是類別型態，就要小心了 !!

如何避免私有外洩（privacy leak）？

- 為你的類別設計拷貝建構子（copy constructor）
- 不要直接回傳物件參考，要拷貝，在回傳拷貝的參考。

記住要回傳拷貝的參考，不要直接回傳該物件的參考。

**鋼鐵類別 Immutable class** **Immutable** 是不可修改的意思，但在這裡我翻譯為鋼鐵類別，比較能夠傳達它的意義。鋼鐵類別 - 一旦它的值被設定了，就不能修修改了。很驚訝吧，**String** 是一個鋼鐵類別。

```

1 String a = "this is a book"
2 a = a.append(", not a pen");
```

各位看上面的 **a** 字串不是變化了嗎？其實真實的運作是先產生一個新的字串，它的內容是“**this is a book, not a pen**”，然後再把 **a** 指到這個字串，原來的字串內容併沒有修改喔。

鋼鐵類別是一個沒有提供可以改變內部值的方法的類別

鋼鐵類別有什麼好處？因為它的值不會被修改，所以也就不用擔心私有漏出時被亂改了。

不要寫一個會回傳非鋼鐵物件的方法。如果你想要回傳的是一個非鋼鐵物件，你應該使用拷貝建構子先生成一個物件，再這個物件回傳。打個比喻好了，你的筆記本借給同學，難保不會被同學修改的亂七八糟，怎們辦？

- 拿去影印一份，在給同學。不管同學怎麼修改，你的原稿都是好的。這就是 copy and write
- 你的原稿是用鋼板刻的 - 這樣你的同學就沒有辦法在上面塗鴉啦。這就是 immutable class

即便 Person 宣告生日 birthday 為私有，還是有可能被修改到

```

1   class Person {
2       public Date getBirthDate() {
3           return birthday; /* 危險
4       }
5   }
```

改成下面的：

```

1   public Date getBirthDate() {
2       // 回傳拷貝參考
3       return new Date(birthday);
4   }
```

但前提是 `Date(Date)` 的拷貝建構子，如下：

```

1   class Date {
2       public Date(Date d) {
3           this.year = d.getYear();
4           this.month = d.getMonth();
5           this.date = d.getDate();
6       }
7   }
```

**深拷貝 Deep copy** 拷貝後的物件和原有的物件不會有任何共同的參考，但有一個例外：  
共同的參考是一個鋼鐵物件。

圖 1.3: 預防私有外洩：Copy and write

## 1.3 虛虛實實：抽象與繼承

### 1.3.1 類別繼承

當類別 B 繼承類別 A 時，表示 B 具備 A 的特性，不用再重複的寫一次，並且可以擴充自己的特性。

```

1  class A {
2      public void m1() { ... }
3  }
4
5  class B extends A { //類別繼承
6      public void m2() { //新增方法
7          ...
8      }
9  }
10
11 B b = new B();
12 b.m1(); => ok
13 b.m2(); => ok

```

A 稱為父類別，B 稱為子類別

```

1  class Vehicle {
2      int speed;
3  }
4
5  class Bike extends Vehicle {
6      int seatHeight;
7
8      public Bike(int seatHeight) {
9          super();
10         this.seatHeight = seatHeight;
11     }
12
13     void setHeight(int) {
14         this.seatHeight = seatHeight;

```

```

15      }
16  }
```

注意事項：子類別建構子會呼叫父類別建構子

### 1.3.2 覆蓋

```

1  class A {
2      public void m1() { print A }
3  }
4
5  class B extends A {
6      public void m1() { //覆蓋方法
7          print B
8      }
9      public void m2() { //新增方法
10         ...
11     }
12 }
13
14 A a = new A();
15 a.m1() => print A
16
17 B b = new B();
18 b.m1(); => print B
```

### 1.3.3 型態轉換

不要讓 compiler 不開心。假設 B 是 A 的子類別。

```

1  A a = new A(); //當然 ok
2  A a = (A) new B(); //ok, upcasting
3  A a = new B(); //ok, upcasting, (A) 可以省略
4
5  B b = new B(); //當然 ok
6  B b = new A(); //compiler error
7  B b = (B) new A(); //downcasting, runtime error
8
9  A a = new B();
```

```
10    B b = (B)a; //downcasting, compiler, runtime ok
```

B 繼承 A 後具備 A 的特性，所以 B 可以做 A 所有的事，反之 A 無法做所有 B 的事。我們姑且以「弱 A」「強 B」來命名。

**B b = new A()**：把一個弱 A 當成一個強 B，產生錯誤。

例子：

```
1  Cat mao = new Cat();
2  Animal mimi = mao; //upcasting (把一個比較低階的物件給比較高階的類別)
3  //給 mao 多取一個名字 mimi，並告訴大家 mimi 是一個動物
4
5  Cat jaja = mimi; //==> 錯誤
6  Cat jaja = (Cat) mimi; //downcasting
7  //mimi 又多了一個名字 jaja，並且告訴大家 jaja 是一支貓
```

### 1.3.4 實例探討

進階字串處理器 EnhancedStringTokenizer。除了可以做字串的解析以外，還可以回傳目前解析的字串集（以陣列的方式回傳）。

```
1  import java.util.StringTokenizer;
2
3  public class EnhancedStringTokenizer extends StringTokenizer {
4      private String[] a;
5      private int count;
6
7      // enhance the constructor
8      public EnhancedStringTokenizer(String theString) {
9          super(theString);
10         a = new String[countTokens()];
11         count = 0;
12     }
13
14     // enhance nextToken
15     public String nextToken() {
```

```

16         String token = super.nextToken();
17         a[count++] = token;
18         return token;
19     }
20
21     // new method
22     public String[] tokensSoFar() {
23         String[] arrayToReturn = new String[count];
24         for (int i = 0; i < count; i++)
25             arrayToReturn[i] = a[i];
26         return arrayToReturn;
27     }
28 }
29
30 class Main {
31     public static void main(String args[]) {
32         String s = "I\u2022love\u2022apple";
33         EnhancedStringTokenizer tokenizer = new
34             EnhancedStringTokenizer(s);
35         while (s.hasMoreTokens()) {
36             System.out.println(s.nextToken());
37             printSoFar(s.tokensSoFar());
38         }
39         static void printSoFar(String[] ss) {
40             for (String s: ss) System.out.println(s);
41         }
42     }

```

### 1.3.5 抽象方法與類別

抽象類別無法產生物件，但可被繼承，例如交通工具 Vehicle 可分為 Bike 和 Car，是一種完全分類，不會有物件從 Vehicle 產生出來。

抽象方法宣告方法的介面（參數及回傳型態），但不具備實作（implementation）。例如所有的交通工具都會向左轉，向右轉，但怎麼則由子類別自己定義。

```

1 abstract class Vehicle { //抽象類別
2     private String ID;
3     public abstract void turnLeft(); //抽象方法

```

```

4     public abstract void turnRight(); //抽象方法
5     public String getID() { //具體方法
6         return ID;
7     }
8 }
9
10    class Bike extends Vehicle {
11        public void turnLeft() {
12            ... //腳踏車的左轉方法
13        }
14        public void turnRight() {
15            ... //腳踏車的右轉方法
16        }
17    }
18
19    class Car extends Vehicle {
20        public void turnLeft() {
21            ... //汽車的左轉方法
22        }
23        public void turnRight() {
24            ... //汽車的右轉方法
25        }
26    }

```

抽象類別是一個半成品，等待子類別去完成。

## 1.4 一法多形：多型

一個方法（method）可以有很多的形式/實作方法。

```

1   class A {
2       void m1() {
3           print A;
4       }
5   }
6
7   class B extends A {
8       void m1() {

```

```

9         print B;
10    }
11 }
12
13 class Client {
14     void op1(A a) {
15         a.m1();
16     }
17 }
```

對 Client 的 op1 而言，a 可能是一個 A 的物件或是 B 的物件，取決於 runtime 時帶進的物件。不要以為 a 的 type 是 A，就認為它會執行 print A。runtime 時才做 binding，稱之為 dynamic binding。

```

1 Client c = new C();
2 c.op1(new A()) => print A
3 c.op1(new B()) => print B
```

## 1.5 無色無相：介面

介面定義一個規格，一個多個物件之間彼此溝通的規格，但他僅定義規格，並不描述其實作方法。Java 中介面的宣告如下：

```

1 interface E {
2     public void m1();
3     public void m2();
4 }
```

m1() m2() 都是抽象的，但我們不需要寫 abstract。請注意介面內只宣告它所提供的方法，及這些方法的使用方式 (signature，即該方法的參數型態即傳回型態)。所有的方法內皆沒有實作。當一類別實作一介面時，所使用的關鍵字是 implements。

當一個類別實踐一個介面，表示它必須實踐這個規格。D 必定要實作 m1() 與 m2()，因為這兩個方法都宣告在介面 E 中。

```

1 public class D implements E {
2     public void m1() {
3         ... // 實作
4     }
5     public void m2() {
```

```

6      ... // 實作
7      }
8  }
9
10 class Client { // Client 是介面的使用者
11     void m (E e) {
12         e.m1();
13     }
14 }
```

### 類別、抽象類別、介面

- 能做什麼，是類別
- 該做什麼，是介面
- 能做什麼，又該做什麼，是抽象類別

## 介面實踐與使用

- 一個好的建築，需要有一個人會蓋，一個人會欣賞。
- 一個介面，需要有類別去實作，也需要有 client 去使用。

只要能實踐 E 的物件，m1() 都可以呼叫使用，不論是 A 或 B 物件。

### 繼承和實作的異同

- 兩者都具備多型，也就是說，當 class C extends D implements E, 則 c instanceof D, c instanceof E 都是 true;
- extends 享受到 code reuse 的好處，但 implements 沒有（因為 interface 內沒有程式碼），它只有被規範“要去履行介面所定義的功能”。

“他一生下來，就背負著「皇帝」的使命，對它來說，是一種責任，一種規範。從這個觀點來看，皇帝是一個介面。

“他一生下來，就擁有龐大的繼承資源，即使什麼都不會做，很多是還是順理成章的完成了。從這個觀點來看，皇帝是一個父類別。

### 1.5.1 多重繼承

Java 所謂的多重繼承是指多重的介面繼承。一個類別可以實作很多的介面，但只能繼承一個類別。類別 G 繼承類別 C 並實作介面 E 與 F 是被允許的。

```

1  public class G extends C implements E, F {
2      public void op2() {
3          ...
4      }
5      public void op1() {
6          ...
7      }
8      public void op4() {
9          ...
10 }
11 }
```

請注意 C, E, F 中同時都定義了方法 op2(), 但這並不會造成任何的混淆，因為 op2() 都是抽象的，並沒有任何的實作。如果讓 G 同時繼承 C 與 A 則會編譯失敗，因為 Java 並不允許同時繼承兩個類別。

```

1 interface Vehicle {
2     //右轉最大角度
3     public final static int MAX_TURN_ANGLE = 60;
4     public void turnRight();
5 }
```

## 實例探討

**Comparable 介面** 任何物品只要符合 Comparable 的介面都是可以比較的。針對 Comparable 我們設計一個 best(x, y, z) 的方法來比較三個物品，該方法將回傳最『好』的物件。

```

1 interface Comparable {
2     public boolean betterThan(Comparable x);
3 }
4
5 class Util {
6     public static Object best(Comparable x, Comparable y,
7         Comparable z) {
8         if (x.betterThan(y)) {
```

```

8         x.betterThan(z) ? return x: return z;
9     }
10    else if (y.betterThan(z)) {
11        return y;
12    }
13    else return z;
14}
15}

```

如果我想比較水果，我該如何修改以下的 Fruit 類別？我們只要定義什麼是「好水果」即可，下面的例子是以 sweetDegree 作為好水果的標準。

```

1 class Fruit implements Comparable {
2     String name;
3     int price;
4     int sweetDegree;
5     int waterDegree;
6     public boolean betterThan(Comparable x) {
7         this.sweetDegree > ((Fruit)x).sweetDegree ? true: false;
8     }
9     public Fruit(int sweetDegree) {
10        this.sweetDegree = sweetDegree;
11    }
12}
13
14 class Main {
15     public static void main(String args[]) {
16         Fruit f1 = new Fruit(12), f2 = new Fruit(23), f3 = new
17             Fruit(9);
18         Fruit best = Util.best(f1, f2, f3);
19         // best 為何？
20     }
21}

```

設計一個 Student 的類別，也透過 Comparable 介面、Util.best 來比較學生。

## 1.5.2 介面內的常數

Interface 可以宣告常數

```

1 interface Vehicle {

```

```

2      //右轉最大角度
3      public final static int MAX_TURN_ANGLE = 60;
4      public void turnRight();
5  }
```

只能宣告常數，不能宣告 instance variable。

## 1.6 範例

### NNEntity

大家都寫過 99 乘法表。我們現在把這個程式「一般化」 – 變成 NN 乘法表。這不是很簡單嗎？參數變化就可以做到，何須物件導向？

那我們再把這個問題更抽象些：如果「乘」的主體不限定是整數呢？例如字串相乘，大家會問：什麼是字串相乘？沒關係我們先假設其效果就是字串相連。也可以顏色相乘，其效果就是顏色相混。能不能用物件的觀念來設計這個題目？讓重用性高一點？

```

1 package basic;
2
3 public class NNMultiplication {
4     public static void main( String[] args ) {
5         NNEntity[] xListA = { new NNInteger(2), new NNInteger(
6             3), new NNInteger(5), new NNInteger(6), new
7             NNInteger(10) };
8         NNEntity[] yListA = { new NNInteger(7), new NNInteger(
9             2), new NNInteger(3), new NNInteger(4), new
10            NNInteger(8) };
11         TableDisplayer.multiplyAndShow( xListA, yListA );
12
13         NNEntity[] xListB = { new NNString("Q"), new NNString("D"),
14             new NNString("T"), new NNString("H"), new
15             NNString("Z") };
16         NNEntity[] yListB = { new NNString("Y"), new NNString("D"),
17             new NNString("Z"), new NNString("V"), new
18             NNString("B") };
19         TableDisplayer.multiplyAndShow( xListB, yListB );
20
21         NNEntity[] xListC = { new NNColor("Red"), new NNColor("Green"),
22             new NNColor("Blue") };
23         NNEntity[] yListC = { new NNColor("Yellow"), new NNColor("Blue"),
24             new NNColor("Magenta") };
25         TableDisplayer.multiplyAndShow( xListC, yListC );
26     }
27 }
```

```
        Red" ), new NNColor( "Red" ), new NNColor( "Green" ), new
        NNColor( "Blue" ) };
14    NNEntity[] yListC = { new NNColor( "Green" ), new NNColor
        ( "Blue" ), new NNColor( "Red" ), new NNColor( "Blue" ),
        new NNColor( "Green" ) };
15    TableDisplayer . multiplyAndShow( xListC , yListC );
16}
17}
18
19 abstract class NNEntity {
20     public abstract NNEntity multiply( NNEntity otherone );
21 }
22
23 class NNInteger extends NNEntity {
24     private int number;
25
26     public NNInteger( int number ) {
27         this . number = number;
28     }
29
30     public NNInteger( NNInteger copy ) {
31         this ( copy . number );
32     }
33
34 // 數字相乘
35     public NNEntity multiply( NNEntity otherone ) {
36         if ( otherone == null ) {
37             return null;
38         } else if ( getClass() != otherone . getClass() ) {
39             return null;
40         } else {
41             NNInteger otherone2 = ( NNInteger ) otherone ;
42             return new NNInteger( this . number * otherone2 . number
43                     );
44         }
45     }
46
47     public String toString() {
48         return Integer . toString( number );
49     }
50 }
```

```
49     }
50
51     class NNString extends NNEntity {
52         private String words;
53
54         public NNString( String words) {
55             this.words = words;
56         }
57
58         public NNString( NNString copy) {
59             this(copy.words);
60         }
61
62         // 字串相連
63         public NNEntity multiply( NNEntity otherone) {
64             if (otherone == null) {
65                 return null;
66             } else if (getClass() != otherone.getClass()) {
67                 return null;
68             } else {
69                 NNString otherone2 = (NNString) otherone;
70                 return new NNString(this.words + otherone2.words);
71             }
72         }
73
74         public String toString() {
75             return words;
76         }
77     }
78
79     class NNCOLOR extends NNEntity {
80         private String color;
81
82         public NNCOLOR( String color) {
83             this.color = color;
84         }
85
86         public NNCOLOR(NNCOLOR copy) {
87             this(copy.color);
88         }
89     }
```

```
89  
90 // 顏色相混  
91 public NNEntity multiply(NNEntity otherone) {  
92     if (otherone == null) {  
93         return null;  
94     } else if (getClass() != otherone.getClass()) {  
95         return null;  
96     } else {  
97         NNColor otherone2 = (NNColor) otherone;  
98         if (this.color.equals("Red")) {  
99             if (otherone2.toString().equals("Red")) {  
100                 return new NNColor("Red");  
101             } else if (otherone2.toString().equals("Green")) {  
102                 return new NNColor("Yellow");  
103             } else  
104                 return new NNColor("Purple");  
105         } else if (this.color.equals("Green")) {  
106             if (otherone2.toString().equals("Red")) {  
107                 return new NNColor("Yellow");  
108             } else if (otherone2.toString().equals("Green")) {  
109                 return new NNColor("Green");  
110             } else  
111                 return new NNColor("Cyan");  
112         } else {  
113             if (otherone2.toString().equals("Red")) {  
114                 return new NNColor("Purple");  
115             } else if (otherone2.toString().equals("Green")) {  
116                 return new NNColor("Cyan");  
117             } else  
118                 return new NNColor("Blue");  
119         }  
120     }  
121 }  
122 public String toString() {  
123     return color;  
124 }  
125 }
```

```

126
127 class TableDisplayer {
128     public static void multiplyAndShow(NNEntity[] xList,
129                                         NNEntity[] yList) {
130         /* Multiply */
131         NNEntity[][] table = new NNEntity[yList.length][xList.
132                                                 length];
133         for (int i = 0; i < yList.length; i++) {
134             for (int j = 0; j < xList.length; j++) {
135                 table[i][j] = xList[j].multiply(yList[i]);
136             }
137             /* Show */
138             System.out.printf("%7s", " ");
139             for (int i = 0; i < xList.length; i++) {
140                 System.out.printf("%7s", xList[i]);
141             }
142             System.out.println();
143             for (int i = 0; i < yList.length; i++) {
144                 System.out.printf("%7s", yList[i]);
145                 for (int j = 0; j < xList.length; j++) {
146                     System.out.printf("%7s", table[i][j]);
147                 }
148                 System.out.println();
149             }
150         }
151     }

```

[\[Get the code\]](#)

說明 NNEntity 如何實踐物件特性：

- 封裝
- 物件是第一人稱
- 繼承、多型

## 1.7 練習

### 類別

**Ex 1:** 一個棋局遊戲中，可能會有哪些類別？每一個類別分別做什麼事？

**Ex 2:** 修正以下的錯誤程式：

```

1 // Filename: Temperature.java
2 PUBLIC CLASS temperature {
3     PUBLIC static void main( string args ) {
4         double fahrenheit = 62.5;
5         double celsius = f2c( fahrenheit );
6         System.out.println( fahrenheit + 'F' +
7             " = " + Celsius + 'C' );
8     }
9     double f2c( float fahr ) {
10        RETURN ( fahr - 32 ) * 5 / 9;
11    }
12 }
13 }
```

### 封裝

**Ex 3:** 以下和者為真

- 一個沒有屬性的類別所產生的物件是無狀態物件。
- 一個全都是 static variable 的類別所產生的物件是無狀態物件。

Hint: instance variable 表示物件的狀態; static variable 則否

**Ex 4:** 默寫存取表 (access level)

**Ex 5:** 一個棋局可能有那些狀態？它會和哪些棋局的屬性相關？

**Ex 6:** Student 可能有哪些狀態？

**Ex 7:** 一個手錶，可能有哪些狀態？

**Ex 8:** 一個 Date 的類別，裡面可能有什麼成員函數？其存取權如何設定？

**Ex 9:** 一個 Chess 類別，裡面可能定義成員函數？其存取權該如何設定？可能定義什麼常數？

**Ex 10:** 呼叫 b.m1() 後 age 的值為多少？

```

1   B b = new B();
2   class A {
3       private int age=12;
4   }
5   class B extends A {
6       public void m1() {
7           age++;
8       }
9   }

```

**Ex 11:** 何者不會造成程式編譯錯誤？

```

1   package p1 class A {
2       protected int n=0;
3   }
4   package p2 class B extends A {
5       (code)
6   }
7
8   (1) public void m1() {n++;}
9   (2) public void m1() {B b = new B(); b.n++;}
10  (3) public void m1() {A a = new A(); a.n++;}

```

**Ex 12:** 象棋系統中的 Chess, ChessBoard 可能具備哪些方法？哪些是靜態方法？為什麼？

**Ex 13:** 以下程式要將棋子 c 移到 loc，合理嗎？有何問題？

```

1   class ChessBoard {
2       void move (Chess c, Location loc) {
3           Chess c2 = new Chess(c);
4           c2.setLocation(loc);
5       }
6   }

```

**Ex 14:** 一個 Chess.move() 可能有幾種 overloading 的方法？

**Ex 15:** 寫出以下類別的建構子：ChessBoard, Chess

**Ex 16:** T/F: 建構子的名稱和類別名稱一樣

**Ex 17:** T/F: 一個類別建構子只能有一個

**Ex 18:** T/F: 建構子不會有回傳值 (只能用 void)

**Ex 19:** 以下程式否正確

```

1   class A {
2       private int a, b;

```

```

3     public A(int a, int b) {
4         this.a=a;
5         this.b=b;
6     }
7 }
8
9 class B extends A {
10    private int c;
11    public B(int a, int b, int c) {
12        super(a, b);
13        this.c = c;
14    }
15 }
```

**Ex 20:** 我們想把一個棋盤上的所有棋子做一個副本回傳會去，Chess 的 copy constructor 及以下的程式碼該如何設計？

```

1 class ChessBoard {
2     Chess[] getChesses() {
3     ?
4     }
5 }
6 class Chess {
7     private name;
8     int priority, loc;
9     ?
10 }
```

## 繼承

**Ex 21:** 以下和者正確：(1) 抽象類別有  $1..*$  個抽象方法 (2) 抽象類別有  $0..*$  個具體方法 (3) 具體類別不可以有任何抽象方法。

**Ex 22:** B 是 A 的子類別，下列 compile 是否可通過？問題在哪裡？

- (a) 設計端：public void m1(A a) {....}。呼叫端：m1(new B());
- (b) A a = new A();
- (c) A b = new B();
- (d) B c = new A();
- (e) B d = new B();

**Ex 23:** 下列何者正確？

1. public B do1() { return new A(); }
2. public String m1(int i) {return "1"; }
3. public private String m2(String s1, String s2) {return s2;}
4. public A do2() { return new B(); }

```

1   class A {
2       public int m1(int i) {return 1;}
3       protected String m2(String s1, String s2) {return s1;}
4       public String m3() {return "Hi"; }
5
6       public B do1() { return new B(); }
7       public A do2() { return new A(); }
8   }
9
10  class B extends A {
11      ???
12  }

```

## 多型

**Ex 24:** 以下會印出什麼

```

1   public class Game {
2       public static void main( String [] args ) {
3           ChessBoard cb = new LongChessBoard();
4           cb.show();
5       }
6   }
7   class ChessBoard{
8       public void show() { System.out.println ("一般象棋");}
9   }
10  class LongChessBoard extends ChessBoard{
11      private void show() { System.out.println("長棋");}
12  }

```

**Ex 25:** 以下會出現什麼訊息？(Hint 子類別會自動的呼叫父類別的預設建構子)

```

1   class A {
2       public A() {
3           System.out.println("hi");

```

```

4      }
5  }
6  class B extends A {
7  }
8
9  public class Main {
10         public static void main(String [] arg) {
11             B b = new B();
12         }
13     }

```

**Ex 26:** 回答以下問題：

```

1  class A {
2      int max(int x, int y) {
3          if (x>y) return x;
4          else return y;
5      }
6  }
7  class B extends A {
8      int max(int x, int y) {
9          return super.max(y, x) - 10;
10     }
11    }
12    A a = new B();
13    a.max(100,20)=?

```

**Ex 27:** 何者不會造成 compiler 錯誤

- (a) print(new Object());
- (b) print(new Employee());
- (c) print(12);
- (d) print("abc");
- (e) print(new Integer(12));

```
1  public static void print(Object x) { ... }
```

**Ex 28:** 這是 override 還是 overloading?

```

1  class Employee {
2      public boolean equals(Employee otherObject) {
3          ...
4      }
5  }

```

## 介面

**Ex 29:** 宣告一個 Moveable 的介面，應用在 Chess 系統。

**Ex 30:** 從以下項目比較 class, abstract class, interface：

- 可具備抽象方法
- 可具備”非”抽象方法
- 可生成物件
- 可具備 instance variable
- 可具備常數
- 可當成一種規範
- 是一種型態

**Ex 31:** 介面會有實作者 (implementer) 和用戶 (client)。下面程式中，哪個是實作者？哪個是用戶？

```

1  interface IA {
2      void m1();
3  }
4  class A implements IA {}
5  class B {
6      void m2(IA a) {
7          a.m1();
8      }
9  }
```

**Ex 32:** 請改以 interface 的方式重新設計：A 不要直接對 B，而是對「會做 m1()」的物件。

```

1  class A {
2      void op1(B b) {
3          ...
4          b.m1();
5      }
6  }
7  class B {
8      public void m1() {
9          ...
10     }
11 }
```

**Ex 33:** 下述程式為某系統的部分程式碼，是否會發生編譯錯誤？執行錯誤？

```

1  interface IA {
```

```

2     void m1();
3 }
4
5 class B {
6     void m1(IA a) {
7         a.m1();
8     }
9 }
```

**Ex 34:** 以下程式有哪些錯誤？

```

1 interface IA {
2     private int x = 100;
3     int y;
4     int z=100;
5     public static final int P = 200;
6     public void m1();
7
8     protected void m2();
9     int m3() {
10         return 100;
11     }
12
13     void m4();
14 }
```

15

```

16 interface IB extends IA {
17     void m5();
18 }
```

19

```

20 class B implements IA {
21     public void m1() {}
22     int m3() {}
23 }
```

24

```

25 class C implements IB {
26     public void m5() {}
27 }
```

28

```

29 abstract class D implements IA {
30     abstract public void m1();
```

```

31         void m4() {} //注意沒有 public
32     }
33
34     class D {
35         void m1() {
36             System.out.println(IA.z + " ");
37         }
38
39         void m2() {
40             IA a = new B();
41             a.m1();
42         }
43
44         void m3() {
45             IA a = new IA();
46             a.m1();
47         }
48     }

```

**Ex 35:** 請寫一個 SuperStringTokenizer, 它除了可以解析字元以外, 還會把解析的字元轉成大寫回傳回來。

- 透過繼承 SuperStringTokenizer 繼承 StringTokenizer 來實作。(注意:SuperStringTokenizer 將 override 父類別的 nextToken())
- \* 透過委託 SuperStringTokenizer 將”包含” StringTokenizer。你一樣要宣告一個 nextToken() 來傳回每一個大寫字元的 token。

**Ex 36:** 類別 Document 有一 text 屬性，記錄文件內容, toString() 會把所有內容回傳。Email 是 Document 的子類別，多了一些屬性: sender, recipient, title, text 分別表示信的傳送者，接受者標題與內容。toString() 會把這些組起來回傳。

設計一個方法檢查文件中是否包含某特定的關鍵字。(hint: 可應用 String.indexOf(String) 回傳字串在某字串的位置。)

```

1   public static boolean ContainsKeyword(?) {
2       ?
3   }

```

## 綜合

**Ex 37:** 依據以下需求撰寫程式：

- 建立一個 Shape 類別，其中要有一個字串來識別它是什麼圖形，要有一個方法來取得圖形的面積，要有一個方法印出圖形面積(這個方法會呼叫取得面積的方法並印出)
- 建立圓形、正方形、矩形、三角形等圖形的類別，繼承 shape 類別，它們分別會多一些各自的屬性(邊長、半徑……等)，override 取得面積的方法，要有建構子來
- 初始化建立物件時的屬性設定寫一支程式建立出圓形、正方形、矩形、三角形等物件，並輸出它們的屬性以及面積。

**Ex 38:** 應用 Comparable (內有 int compareTo(Object) 方法) 介面來寫一個排序的程式，並且用來排序以下的物件。

- 一個類別 Student, 裡面的屬性包含身高、體重、成績，如果「身高 + 成績 - 體重」比較較高，則較好。
- 一個類別 MobilePhone, 「面板 size \* 價格 \* 0.01」比較高的則比較好

設計一些測試案例來展示。

**Ex 39:** 考慮一個象棋短棋遊戲，32 個棋子會隨機的落在 4\*8 的棋盤上。透過 Chess 的建構子產生這些棋子並隨機編排位置，再印出這些棋子的名字、位置

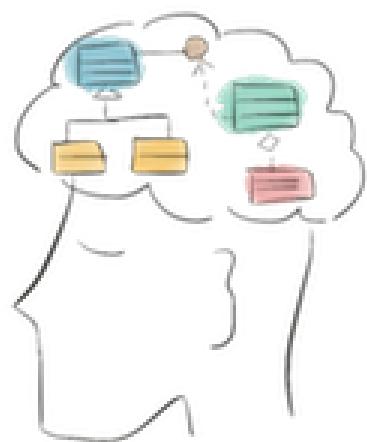
- ChessGame: void showAllChess(); void generateChess();
- Chess: Chess(name, weight, side, loc); String toString();

**Ex 40:** 同上，(1) 新增一個 Location 的類別來記錄位置; Player 記錄玩家資訊 (2) ChessGame 繼承一個抽象的 AbstractGame, AbstractGame 宣告一個抽象的方法 setPlayers(Player, Player)。

**Ex 41:** \* 撰寫一個簡單版、非 GUI 介面的 Chess 系統。使用者可以在 console 介面輸入所要選擇的棋子的位置(例如 3,2)，若該位置的棋子未翻開則翻開，若以翻開則系統要求輸入目的的位置進行移動或吃子，如果不成功則系統提示錯誤回到原來狀態。每個動作都會重新顯示棋盤狀態。

# Chapter 2

## 圖模術：UML



我們必須在一定的抽象程度下去思考一個系統，用機器語言來思考一個系統是最糟的方式，因為機器語言是給機器看的。高階的語言例如 Java 等物件導向的語言雖然高階，但仍適合機器看而非人類。物件導向的模組語言提供一個符合人類思考的抽象層級來幫助我們分析設計一個系統。目前最廣為使用的模組語言是統一模組語言 (Unified Modeling Language ; UML)，它是一種圖形化的模組語言，不同角色的工程師可以在不同的階段利用它來視覺化系統、分析系統、建構系統與製作系統文件。UML 是設計樣式的基礎，因為設計樣式在說明解決方式時都是以模組的方式呈現的。

## 2.1 模組

在建構一個大樓前，設計師會先將大樓的藍圖繪製出來，工程師依此藍圖建構大樓；在接待所中大樓的模型也會被建立，作為客戶決定是否購買該樓的參考。藍圖與模型都是一種模組。開發軟體系統前，我們會先建立若干的軟體模組：

- 模組是一種真實的簡化；模組是系統參與者溝通的媒介橋樑；
- 模組是處理複雜度的工具；
- 模組可以作為系統開發的藍圖或規格書。

模組與符號 (notation) 息息相關。符號可由圖形或文字所構成，用來表達一個模組的語法 (syntax)。對一個模組語言而言，每一個符號都必須有唯一且清楚的意義，稱為語意 (semantic)。例如在 UML 的類別模組中方塊 (符號) 代表一個類別 (語法)，而類別的意義是一群有相同特色與行為的物件的集合 (語意)。

模組的要素：要成為一個好的模組語言並不容易，一個好的模組語言應該具備以下的條件：

抽象力	模組是真實事物的抽象，它只呈現部分觀看者有興趣的一面，而忽略細節或其它的層面。一個方法論通常提供多個模組並可從不同的抽象來表達系統，例如行為模組，結構模組與功能模組等。
理解力	模組必須能夠容易閱讀，理解，以作為開發者及使用者之間溝通的橋樑，作為日後維護的依據。為了提升理解力，模組通常是圖形化的符號所組成的。
正確性	模組應該能清楚的表達它的語意。模組內每個符號應該有明確的定義。UML 的超模組 (meta-model) 即是用以定義模組語意的模組。具有正確性的模組才能作為系統設計與開發的規格。
預測力	模組必須某個程度的表示未來要開發的軟體系統的形式，結構或行為。
低成本	模組必須是方便建立的，因為它是作為溝通的橋樑，會經常的修改，如果建立它的成本很高，就喪失了模組的作用了。

一個系統或事物，用不同的抽象角度去看，就會得到不同的結果。從物件的抽象角度來看與從功能的角度來看，所得到的模組就完全不同。其他還有各種不同的模組：資料模組、流程模組、狀態模組等。

## 2.2 UML 簡介

物件導向模組語言最早出現在 1970 年代，當時一些學者為了因應各種不同的物件導向語言與越來越複雜的系統，紛紛提出各種不同的物件導向模組方法。從 1989 到 1994 年，物件導向模組方法由十多個增加到近五十個，其中較為出名包括 Booch，Jacobson 的 OOSE，Rumbaugh 的 OMT，Fusion，Shlaer-Mellor 與 Coad-Yourdon，每一種方法都有其優缺點。使用者開始對這些不統一的方法論感到無所適從，於是許多人開始思考一個統一的作法。

到了 90 年代中期，Booch，OOSE 與 OMT 三個方法的主要作者 Booch、Jacobson 與 Rumbaugh 開始整合其他方法的優點，希望能夠建立一個統一的模組語言。值得一提的是，這三個方法正好在物件導向方法中各自佔有其重要的地位，Booch 著重於系統的「設計」及「建構」階段；OMT 則對系統「分析」提出精闢的見解；而 Jacobson 的使用案例 (Use Case) 簡潔而明瞭，對於「需求擷取與分析」有良好的支援。這三個方法在軟體發展生命週期中各佔有不同的角色，由於他們的合作，奠定了 UML 的基礎，也使得物件導向方法論更趨完整而易於使用。

UML 的目標為：

- 提供容易使用、表達能力強的圖形化模組工具以建立可互相溝通、有意義的系統模組。

- 提供擴充性並讓開發者可自定其所需要的模組概念。
- 提供與程式語言 (language-independent) 無關與發展程序無關 (process-independent) 的規格語言。
- 提供一個正規 (formal) 的方式以定義此模組語言。
- 鼓勵模組工具市場的成長。
- 提倡高階的發展概念，例如元件 (component)、合作 (collaboration)、框架 (framework)、與樣式 (pattern)。

有句話說「瞎子摸象」，意思是不同的人去摸瞎子，都以為他所摸到就是象的全貌，但事實是他所認知只是整體的一部份。一個尚未開發完成的系統，對於系統開發人員而言，正如同瞎子摸象一般的困難，因此我們需要從各種不同的角度來看系統，才能正確的表達系統的全貌。

UML 提供許多模組圖，約略可分成靜態與動態 2 個層面，介紹如下：

- 類別圖 (Class Diagram)：描述類別屬性、作業員與類別間的關係。
- 物件圖 (Object Diagram)：描述物件間的關係。
- 元件圖 (Component Diagram)：描述系統的真實元件 (physical component) 與元件間的關係。
- 配置圖 (Deployment Diagram)：描述系統元件在硬體上的配置，與各硬體間的關係。
- 使用案例圖 (Use Case Diagram)：從使用者的觀點描述系統的功能。
- 循序圖 (Sequence Diagram)：以時間的順序為主描述物件的互動。
- 合作圖 (Collaboration Diagram)：以空間的配置為主描述物件的互動。
- 狀態圖 (Statechart Diagram)：以物件接受到訊息前後狀態的改變來描述物件的行為
- 活動圖 (Activity Diagram)：主要用途為用來表現活動與活動間的流程。

## 特性與優點

- 是一種圖形化的模組語言。不同角色的工程師可以在不同的步驟利用它來視覺化系統、規劃系統、建構系統與製作系統文件。UML 是一個模組語言。所謂的「語言」，提供了一群特定的字彙與組合這些字彙的規則來達到溝通的目的。模組語言亦是如此，但它著重於以圖形化的方式表達一個軟體系統的結構或行為。要特別注意的是，模組語言只是一種語言，描述系統的架構與行為，它並沒有規定分析師、程式設計師在什麼階段作什麼事，因為這是屬於系統發展程序的工作。這正也是 UML 的優點之一：將語言從發展程序中獨立出來，使此模組語言更具彈性。

- **UML 用以視覺化系統。**由於軟體的不可視性與複雜性，以程式碼來呈現軟體系統的是不可行的方法，圖形化的方式可以大大的提高系統的可閱讀性。
- **可視為規格書。**軟體開發如同建築設計，其過程中也必須將需求、分析、設計、實作、佈署等各項工作流程之構想與結果予以呈現。在軟體的開發過程中，最重要的兩個技術規格書為系統需求規格書 (System Requirement Specification ; SRS) 與系統設計規格書 (System Design Description ; SDD)。這兩個規格書分別在分析階段與設計階段結束後產出，用已說明需求與設計的內容。UML 的模組設計可以放在 SRS 與 SDD 中協助描述需求與設計。

```
1  1. 簡介( Introduction )
2    1.1 系統目的( System Objective )
3    1.2 系統範圍( System Scope )
4    1.3 詞語用義( Definition and Acronyms )
5    1.4 參考文獻( Referenced Documents )
6  2. 系統作業說明( Proposed System Description )
7    2.1 系統作業流程( Process Description of Proposed System )
8      UML – Activity Diagram
9    2.2 系統特色及預期效益( System Characteristics and Expected
10   Benefits )
11   2.3 系統限制( System Constraints )
12  3. 業務規則( Business Rules )
13  4. 系統需求說明
14    4.1 功能性需求( Functional Requirement )
15    4.2 功能性需求( Nonfunctional Requirement )
16    4.3 計限制( Design Constrains )
17  5. 系統模組( System Models )
18    5.1 需求定義 ( Requirement Definition )
19      UML – 使用案例圖
20        5.1.1 案例說明( Use Cases Description )
21          UML – 使用案例描述
22        5.1.2 軟體關係圖( Conceptual Data Model )
23    5.2 技術分析( Requirement Analysis )
24      5.2.1 靜態結構分析
25        UML – 類別圖
26      5.2.2 動態行為分析
27        UML – 循序圖
28        UML – 狀態圖
29  6. 系統與軟體驗收準則( System and Software Acceptance Criteria )
30    6.1 系統驗收準則( System Acceptance Criteria )
31    6.2 軟體驗收準則( Software Acceptance Criteria )
```

需求規格書主要用來定義使用者的需求，當規格書的內容被使用者確定後，開發團隊就可以依照此需求設計系統。下圖為一個設計規格書的樣版，UML 的模組可以被應用在設計規格書中協助定義設計的內容。與需求規格書比較，會發現許多部分所用的圖形是一樣的，差別在於設計規格書所描述的要更仔細。

- 1    1. 簡介 (Introduction)
  - 2    1.1 系統目的 (System Objective)
  - 3    1.2 系統範圍 (System Scope)
  - 4    1.3 詞語用義 (Definition and Acronyms)
  - 5    1.4 參考文獻 (Referenced Documents)
- 6    2. 系統架構設計 (Architecture Design)
  - 7    2.1 系統架構 (System Architecture)  
      UML 配置圖
  - 8    2.2 軟體架構 (Software Architecture)  
      UML 套件圖
- 9    3. 資料庫設計 (Physical Data Model)
- 10    4. GUI 設計 (Prototype)
- 11    5. 細部設計 (Detailed Design)
  - 12       UML – 類別圖
  - 13       UML – 元件圖
  - 14       UML – 循序圖
  - 15       UML – 狀態圖
- 16    6. 系統整合策略 (Integration Strategy)
- 17    7. 建構程序 (Build procedure)

### 使用 UML 於系統設計規格書

當系統設計規格書完成後，便可以依照此規格書實作。目前許多模組工具，例如 IBM 的 Rational Rose、Borland 的 Together 與 Sybase 的 Power Designer 等都提供可以從設計規格產生部分程式碼的功能，讓開發者可以很快的從設計階段銜接到實作階段。

## 2.3 功能情境：使用案例圖

使用案例圖可以用來表達系統主要的使用者是誰？他們使用系統的目的為何。使用案例圖包含以下的觀念：

- 角色 (Actor)。系統的使用角設，在 UML 稱為「Actor」。UML 之所以把它稱之為

Actor 而非「user」的原因是使用系統的「物體」不一定是「人」，它也可能是另一個系統或外在設備。比方我們設計一個公司的薪水管理系統，與這個互動的外在物體除了會計人員以外，銀行系統也是其一。在這種情況下，會計人員與銀行系統都是角色。另外要特別注意的是使用者是以使用系統的「角色」來定位的，而非個體。例如 John 在公司中同時是會計人員與公司員工，在薪水管理系統中我們找出的使用者是「會計人員」與「公司員工」兩個使用者，而非 John 一個人。

- 使用案例 (Use case)。描述一個角色使用系統的「互動過程」。說的白話一點就是「人機」之間的你來我往。
- 包含關係。使用案例之間的關係，圖中的 *UseCase<sub>2</sub>* 包含 *UseCase<sub>1</sub>* 表示 *UseCase<sub>1</sub>* 是部分的互動過程，抽取出來的目的是降低重複的描述。
- 擴充關係。使用案例之間的延伸關係，用來表示額外的互動或是例外處理。圖中的 *UseCase<sub>1</sub>* 表達的是一個基本的互動，例外或額外的互動描述在 *UseCase<sub>3</sub>*，這樣的好處的不會讓 *UseCase<sub>1</sub>* 顯得複雜而難以閱讀。

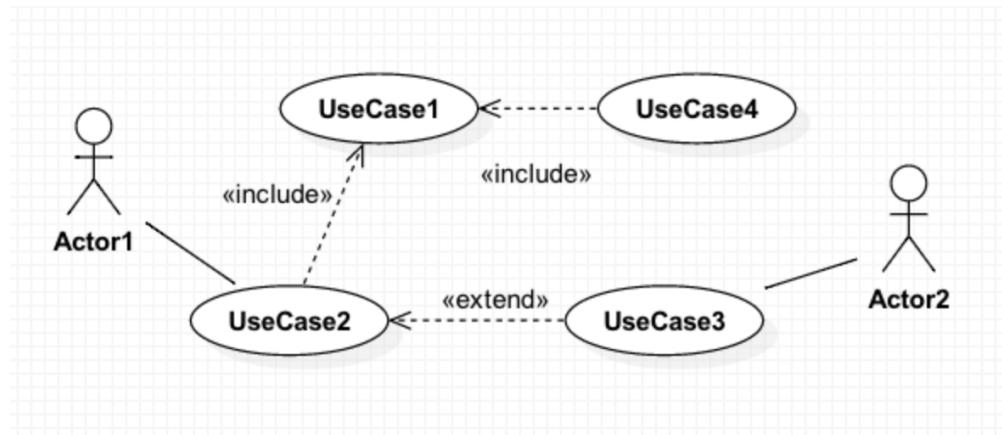


圖 2.1: 使用案例圖

### 使用者動作

1. 點選『查詢並預借媒體』
3. 要求列出所有媒體
5. 按下預借的控制鍵

### 系統動作

2. 顯示可能的查詢方式，例如直接輸入媒體代號、輸入關鍵字、或要求列出所有媒體
4. 系統列出所有媒體，包含媒體名稱、媒體數量、租借狀況等媒體特性。媒體旁有一個『預借』的控制鍵。
6. 系統檢查使用者可否借此媒體，若可以，則顯現租借成功的訊息。

注意使用案例

- 不是流程圖；
- 每個使用案例有一個完整的使用目的，他不是一個功能；
- 通常是動詞；
- 可以透過 use, extends 等關聯來組織使用案例。

## 2.4 靜態結構：類別圖

結構模組是物件導向系統中最常見的模組，它不僅可以描述問題領域的概念及系統靜態結構，更是物件導向程式設計的基礎。以下介紹類別圖的幾個用途。

### 2.4.1 類別圖的用途

類別圖是物件觀念中一個非常重要的角色，它提供以下的功能：

**概念模組 (Conceptual Modeling)** 在系統發展生命週期中，概念模組是位於需求擷取之後、物件分析之前，是需求分析一個很重要的步驟。概念模組是一個幫助分析師瞭解、分析問題領域 (problem domain) 的重要方法，這裡所謂的問題領域是指系統開發所涵蓋的企業規則、專業領域知識及相關概念之間的關係等。例如設計會計系統就必須瞭解會計領域的借貸原則、設備的折舊計算等；設計學術會議管理系統，就必須瞭解學術會議進行的流程與方法。

概念模組後所產生的規格稱為概念圖 (conceptual model)，主要由概念與關聯所構成。「概念」是問題領域中所涵蓋的想法、事件、關係、動作或真實的事物，只要它對這個問題領域而言是特別且具意義的，都有可能被模組在概念圖中。分析師可以從使用案例中去尋找概念，通常是文中的名詞。圖 2.2 是一個象棋系統的概念模組，Game 有一個 ChessBoard 和 Players，一個 ChessBoard 有 Chesses，Player 可以 select/move/eat Chesses，Game 和 Player 之間有 win/lose 的關係，ChessBoard 有大小、背景顏色的屬性，Chess 則有吃、移動、跳的行為。

**瞭解軟體系統的設計結構** 在物件導向系統中，功能是靠一群物件和其他元素 (介面、元件等) 共同合作達成的。在類別圖中，我們可以瞭解類別間靜態的合作關係。在往後談到設計樣式的章節中，我們多以類別圖來表現一個設計樣式的結構，也同時表現該設計樣式的精神。

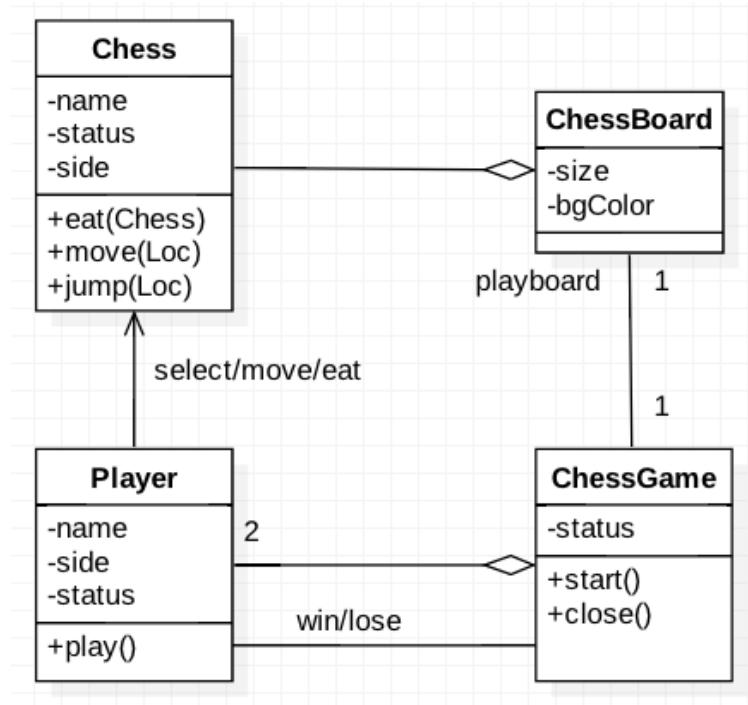


圖 2.2: 象棋系統的概念圖

UML 的類別圖不僅可以做為概念圖，也可以作設計之用。設計與概念模組的最大差別在於設計模組從軟體系統的角度來看模組，而非概念性的模組。所以，一些軟體特有的觀念，例如屬性的可視性、方法的參數型態及物件間的相依關係等都可以明確的表達出來。

類別圖在整個設計週期中是不斷被修改調整的，初期是作為概念模組，在設計階段時被精鍊 (refine) 為設計模組，最後到實作階段就成了真實存在的物件導向程式，如表 ??。

**資料庫綱要模組化 (Database Schema Modeling)** 有了剛剛概念模組的觀念後，其實就不難理解為什麼類別圖也可以分析資料庫的綱要，因為資料庫的綱要分析常常都是從概念模組開始的。由概念模組得到資料庫綱要的原則整理如下：

- 一個概念對應一個資料庫表格。
- 概念內的屬性對應資料庫表格內的欄位。
- 概念圖內的一對一關係與多對一關係對應資料庫表格內的外鍵 (foreign key)。
- 概念圖內多對多關係對應一個資料庫表格。

此時類別圖的角色是類似 ER 圖 (Entity Relationship Diagram) 的，主要差別在於類別是允許有方法，而 ER 圖中的 Entity 是沒有方法的。簡單的說，類別圖的表達能力較 ER diagram 強，可以用以設計資料庫。

	概念模組	設計模組	實際程式
時機	分析階段	設計階段	實作階段
目的	瞭解問題領域	物件如何合作以解決問題	物件如何實作以解決問題
重點	真實世界的概念，非軟體元素	架構的討論	軟體元素，例如 Button 真實的語法
結構	屬性、關聯、可不含方法	屬性、關聯、方法	屬性、方法、真實的演算法

表 2.1: 類別圖在軟體設計週期有不同的用途

### 2.4.2 物件與類別

在物件導向方法論中，物件是所有觀念的核心，因為系統的最基本組成單位是物件。相對於類別，物件是真實存在的事物，而類別是一群具有相同屬性與行為的物件的集合。類別可能是

物體	包含真實物體，如電腦、大樓、收據、人等，及抽象物體，如電腦系統內的游標、按鈕、面版等。
組織	如大學、公司、會計部門等。
角色	如執行長、會計師、工程師等。
概念	家族、流程、工作等。
事件	包含真實事件如選舉、意外等，及抽象事件如電腦系統內的滑鼠移動、鍵盤輸入等。

一般而言，類別都是名詞，具有屬性與提供服務，也具有明確定義的自身行為。但也有例外，尤其是在設計階段，為了讓系統更有彈性許多類別的宣告不一定會有屬性。我們會在稍後設計樣式的章節看到這些實例。

物件是類別的實體，類別則是物件的概念描述。電腦是一個類別，但你正在使用的電腦 – 一個真實存在的東西 – 是一個物件。同理，「人」是一個概念，抽象的代表一群有頭有手、能動能唱、具有思考能力的動物。人是一個類別，而「張三」，一個具體存在的事物則是人的實例，是一個物件。

**分類與實例化** 將一群物件依照他們的特性、行為的不同而分類分群的動作稱為分類，亦即由一群物件組複合成一個類別。反之，從一個類別生成物件的動作稱為實例化，被生成的物件稱為該類別的實例 (instance)。圖 2.3 中，類別與物件的「實例化」關係，可以用 *instanceOf* 來表達。

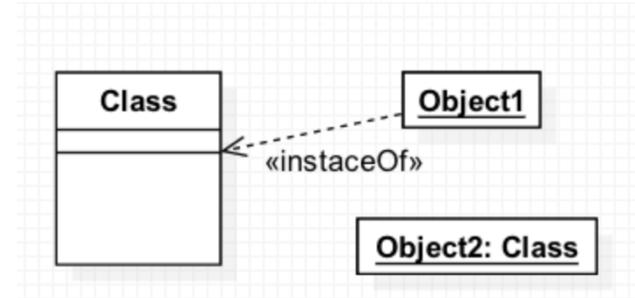


圖 2.3: 類別與物件的 UML 表示。物件有底線，類別則沒有。冒號表示物件的類別形態。`instanceOf` 表達物件與類別的關係。

一個類別表示一群具有相同屬性、關聯、方法及相同意義的物件。在 UML 中，類別用一個矩型表示，可以分為三個區塊 (圖 2.3)。在分析時，並不一定要將三個區塊都劃出來。通常我們會選擇性的隱藏部份的區塊以強調特別的模組特色。每個區塊的說明如下：

- 第一區塊「類別名稱」是不可缺少的區塊，它的主要作用是給類別一個清楚並且容易識別的名稱，通常是個名詞，例如汽車、電腦等都是合適的類別名稱。第一區塊除了描述類別名稱外，也可以加入 stereotype 來描述這個類別是屬於哪一類型的類別。
- 第二區塊是屬性區塊，用以描述該類別的相關屬性，例如汽車具備的屬性就有型號、汽缸數、里程數等相關的屬性。
- 第三區塊描述此類別的方法，用以描述可以作用在這個類別上的方法或此類別具備的功能。例如汽車具備發動、行駛、轉彎等方法或功能。

**屬性** 屬性是描述一個類別的特性。一個類別可以有很多或沒有任何的屬性。一旦你定義了一個類別的屬性，此類別的所有實例將會擁有這個屬性，並遵守這些特性的限制。舉例來說，一款車 (類別) 有有它的長度、氣缸數、配備、最高速度等，這些都是它的特性，屬於這款車的任何一輛車 (實例) 都將具備這些特性，但每個特色的值可能不一樣，例如 A 車的最高速是 100mile/hour，B 車的最高速是 120 mile/hour。

在 UML 中，屬性描述在第二個區塊。屬性的相關特性可定義如下：

[可視性] 屬性名稱 [多樣性] [: 型態] [= 初始值] [特性字串]

說明：在中括號 ([]) 內的表示可有可無，例如可視性、多樣性等。屬性名稱沒有被中括號刮起來，表示它是必要的。

以下幾個例子說明幾種類別的宣告方式：

telephone	只有屬性名稱
+telephone	屬性名稱、可視性
telephone: String	屬性名稱、型態
telephone[2..4]: Port	屬性名稱、多樣性、型態
telephone: String= null	屬性名稱、型態、預設名稱
telephone[2..4] {readOnly}	屬性名稱、多樣性、特性字串

在系統分析階段，通常只會有屬性名稱、型態被描述。其他特性，通常到設計階段或實作階段才作規劃。

**可視性 (visibility)** 可視性描述屬性是否能被其它的類別「看」的到(這是一個比較生動的字眼，真正的意義是其他類別使否能夠參考或修改此一屬性的值)。UML 定義了四個可視性：

- 私有 (private)：只有該類別本身可以使用此一屬性。UML 用「-」來表示。
- 保護 (protected)：只有該屬性的後代類別 (descendant of classes) 可以使用此一屬性。UML 用「#」來表示。
- 公開 (public)：其他類別都可以使用此屬性。UML 用「+」來表示。
- 套件 (package)：同一個套件的類別可以使用此屬性。UML 用「~」來表示。

**多樣性 (multiplicity)** 多樣性一般用於描述類別間的關聯在數量上的特性，亦可用在屬性上，用以描述一個屬性的數量。在下例圖 2.4 中，一個主機板有 USBport 這個屬性，其多樣性為 [2..4] 表示其擁有 USB port 的數目在二至四個。對應到程式碼，多樣性通常用陣列或 ArrayList 來表達。

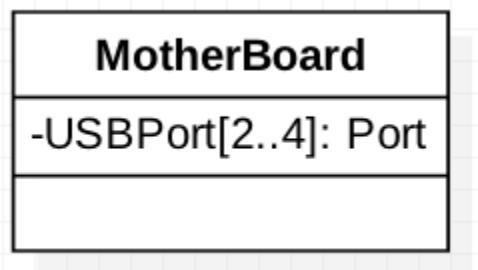


圖 2.4: 屬性的多樣性宣告

## 方法

方法用來描述一個類別的行為特性。方法通常都是動詞，代表該類別可以提供的服務。

- **功能**：執行該物件具備的功能。例如 Square 類別具有 *computeArea(int width, int height)* 來計算它的面積。
- **查詢**：查詢該物件的狀態，通常以 *is* 開頭，例如在 Book 類別中定義 *isBorrowed()* 來查詢一本書目前的借閱狀態。
- **狀態設定**：設定物件的值狀態，通常以 *set* 開頭，例如 *setColor()*、*setBackground()* 等。

UML 中方法的宣告語法如下

[可視性] 方法名稱 [(參數列)] [: 傳回型態] [(特性字串)]

display()	只有方法名稱
+ display()	方法名稱、可視性
display (i: integer, s:String)	方法名稱、型態
getPasswd(): String	方法名稱、傳回型態
display() {leaf}	方法名稱、特性字串

**可視性 (visibility)** 可視性描述方法是否能被其它的類別「看」得到，亦即，是否可以呼叫此方法。與屬性一樣，UML 對方法定義了四個可視性：

- 私有 (private)：只有該類別本身可以呼叫此方法。UML 用「-」來表示。請注意私有的方法是連子類別也不能呼叫使用的。
- 保護 (protected)：只有該類別或其後代類別 (descendant of classes) 可以呼叫此方法。UML 用「#」來表示。
- 公開 (public)：其他類別都可以呼叫此方法。UML 用「+」來表示。
- 套件 (package)：同一個套件的類別可以呼叫此方法。UML 用「~」來表示。

```

1  class TestVisibility {
2      void testPrivateAttribute() {
3          People p = new People(); //宣告一個 People 的類別
4          p.SSN="S123456789"; //錯誤！SSN 是一個私有屬性，不可以直接存取
5      }
6  }
```

在類別內的方法區塊中，除了描述一個個的方法外，一樣可以有 stereotype 來做分類。圖 2.5 中的 <> 表示一個 Stereotype。<> constructor>> 表示 Book(): void 與 Book(name): void 都是建構子。

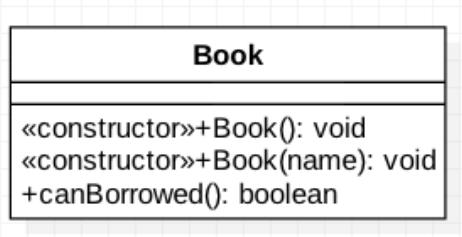


圖 2.5: 刻版 (Stereotype)

### 2.4.3 繼承樹

物件不會單獨的存在，它會和系統中的其他物件產生關係。在 UML 中，關係可以分為三大類：關聯 (association)、一般化關係 (generalization) 與相依關係 (dependency)。

**一般化關係** 描述一般化事物 (general element) 與相對特殊事物 (specific element) 之間的關係。比方說機車與汽車都是車子的一種，車子即是機車與汽車的一般化，而機車與汽車則相對性的為其特殊化。我們稱特殊化的類別為一般化類別的子類別 (subclass/child class)；反之，一般化類別為特殊化類別的父類別 (superclass/parent class)。在 UML 中，一般化的關係用一個三角形表示，如圖 2.6 所示。

**特殊化 (specialization)** 將一群類別共同的特性與行為抽取出來獨立成為一個類別的動作稱為一般化；反之，將一個類別分成許多子類別動作稱為特殊化。圖 2.6 中，車子可以分為卡車、腳踏車與汽車就是一種特殊化。從另一個角度來看，卡車、腳踏車與汽車都是車子，這則是一種一般化。一般化與特殊化是一體兩面、共同存在的。

在物件導向的系統設計中，我們可以先建立一個較為抽象的類別，透過特殊化逐步的建立子類別，進而完成一個類別階層 (class hierarchy)；亦可以先建立一群較明細的類別並透過一般化來建立類別階層。雙方向並行的方式也是常見的。

**繼承** 在物件導向技術中，一般化關係伴隨著繼承 (inheritance) 的機制。繼承使得子類別擁有父類別的功能，而不需再實作一次。如汽車的例子中，當我們宣告 Bus 繼承 Vehicle

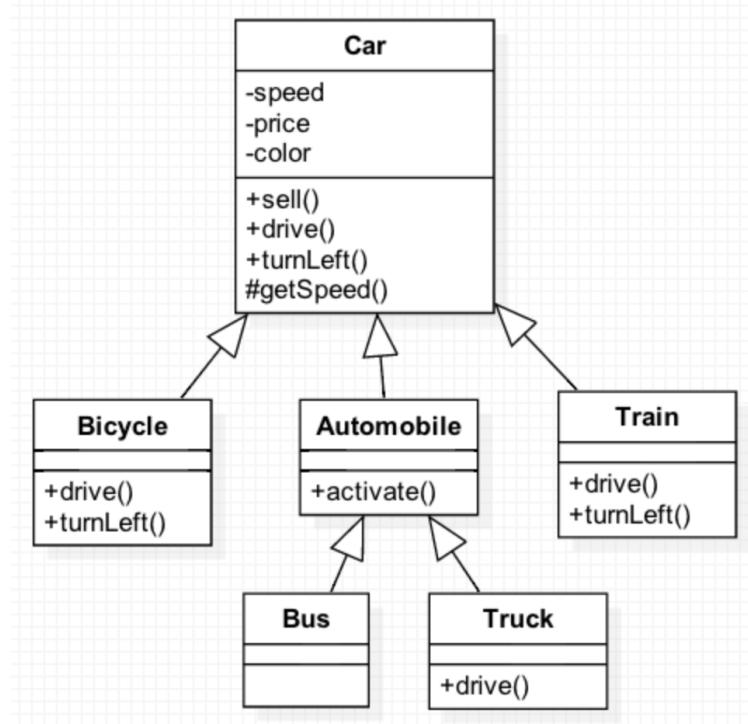


圖 2.6: 汽車的繼承樹

時，Automobile 就具備所有 Car 的特色 – 如汽缸數、速度、與價格等；也同時具備所有 VehicleCar 的功能，如啟動、開車、左右轉等。

```

1 Bus b = new Bus(); //宣告一個 Bus 物件 b
2 b.turnLeft(); //儘管 Bus 內沒有宣告 turnLeft()，它還是此功能。
  
```

繼承與可視性是兩碼子的事，雖然 Vehicle 具備速度的屬性，但這並不意味著 Bus 中可以存取速度這個屬性 – 如果速度在 Vehicle 中是宣告為私有。只要是私有的，就只有該類別能夠存取，即使是子類別也一樣不能存取。雖然子類別不可以存取速度這個屬性，但他仍然具備這個屬性 – 例如當任何一個車子的子類別的物件被建立時，速度的預設值都為 0。如果真的想要讓子類別存取父類別的屬性，有兩種方法：

- 將該屬性設為保護型（protected）的可視性。
- 建立一個保護型的「方法」來存取該屬性，例如在 Vechicle 中宣告 `protected int getSpeed()`。如此一來，Bus 可以透過 `getSpeed()` 來讀取速度。相較於前一種方式，這樣的好處是可以對該屬性多一層的控制。如果我們沒有建立 `setSpeed()`，那麼子類別就只能讀取速度，不能修改速度。

**擴充、修改與限制** 繼承後子類別可以具備父類別的特性，並且擴充、修改或限制其父類別的功能：

- **擴充 (extension)**：新增功能。例如 Bus 在繼承 Vehicle 以後新增了一個 pressStopRing() 的功能。
- **修改 (redefinition)**：保持功能的介面，但是重新定義實作該功能的實作。例如 Vehicle 有預設的 turnLeft() 的方法，但 Bus 重新定義 turnLeft() 的方法，這又稱為方法覆蓋 (Override)。
- **限制 (restriction)**：減少功能。父類別所提供的功能不被繼承，也就是說子類別並沒有該功能。由於限制會產生許多問題，所以目前物件導向程式語言並沒有提供。

**抽象類別** 抽象類別是一種不能產生物件的類別。為什麼要定義一個不能產生物件的類別呢？(1) 我們已經建立完整的分類，所有的物件一定屬於某一類的子類別，所以不應該由父類別生成物件，因此將父類別宣告為抽象的。(2) 抽象類別訂立了其他子類別的基本規格 (basic specification)，子類別必須定義部分方法的實作，特別是宣告在抽象父類別中的抽象方法。在 UML 中，抽象類別的類別名稱以斜體呈現，或是加上 `abstract` 的標記。圖 2.7 中的 `Icon` 被宣告為一個抽象的類別，因為我們不希望它能夠直接生成物件：它必須透過 `RectangleIcon` 或 `ImageIcon` 來生成。

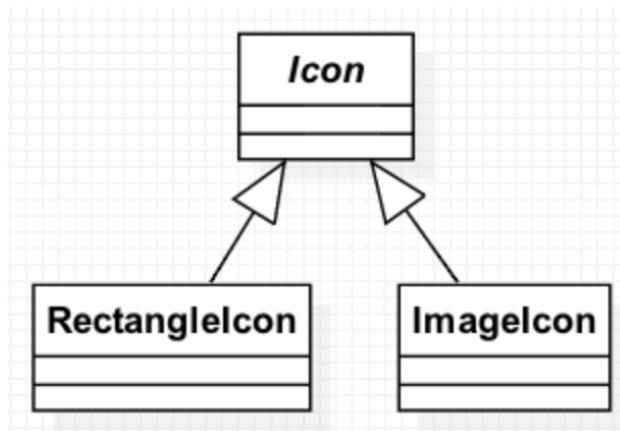


圖 2.7: 斜體字表示該類別是一個抽象類別

**抽象方法** 在 UML 中該方法會以斜體字表示。抽象方法是定義一個沒有實作 (implementation) 的方法。定義抽象方法的目的是作為一個「規格」讓子類別來依循，一個具有抽象方法的類別必定存在子類別可以實作該方法。例如 Car 中有定義一個抽象方法 `drive()`，其含意在「所有種類的車子應該具備開車功能」，而非「Car 定義了 `drive()` 的方法，可讓子類

別來重用」。UML 以斜體字來表示抽象方法，在 Car 中，drive()、turnLeft() 與 turnRight() 都是抽象方法，其實作留給子類別定義。以下觀念亦請注意：

- 抽象類別不能生成物件
- 具有抽象方法的類別一定是一個抽象類別
- 抽象類別不一定要是具有抽象方法

**多重繼承** 類別通常只有一個父類別，但也允許有多個父類別。在圖 2.8 中，汽車與帆船都有兩個父類別，在繼承上稱為多重繼承 (multiple inheritance)。同一個類別有兩種以上的分類方式時，可以在分類的符號 ( ) 旁寫上分類的基準為何 (discriminator)。在此例中，介質與動力為分類交通工具的兩個不同基準。

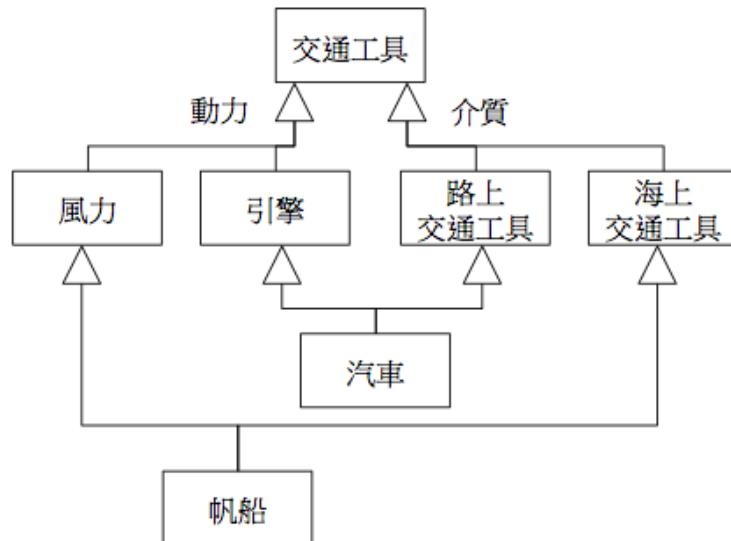


圖 2.8: 多重繼承

然而，多重繼承卻有時作上的困難。當類別 C 同時繼承類別 A 與類別 B 時，有可能 A 與 B 都同時有定義某個方法的實作，如此一來類別 C 就不知道該依循哪一個類別的實作了。Java 因此並沒有直接的支援多重繼承，而是以繼承介面的方式來達到概念上的多重繼承。這是因為介面內並沒有定義實作，所以即使繼承了很多的介面，也不會有混淆的情況發生。

**多型** 多型 (polymorphism) 表示 many form 的意思。在物件導向的設計裡，多型表示一個方法有多種定義 (形式)。多型與 Overloading 與 Overriding 相關。Overloading 表示一個相同的方法名稱可以因為參數型態的不同而有不同的定義，例如

```

1     getPrice(): void
2     getPrice( String name): void

```

`getPrice()` 有兩種不同的形式 – 也就是說有兩種定義。Overriding 則表示子類別可以與父類別有相同的方法、相同的參數列與相同的回傳型態（亦即重新定義父類別的方法）。在圖 4.14 中 `encrypt(String)` 的方法有三種定義，分佈在父類別與子類別之中。Document 物件要求 Encryption 物件協助加密，在它的 `encryptString()` 的方法中，它不需要指明到底加密者是 RSAEncryption 或 DesEncryption，只需說明要送訊息給 Encryption 的物件：

```

1 EncryptString(Encryption e) {
2     result = e.encrypt(source); // 加密的方法取決於 e 的型態
3     ...
4 }

```

#### 2.4.4 介面實作

介面定義一個規格，一個多個物件之間彼此溝通的規格，但他僅定義規格，並不描述其實作方法。UML 中的介面用一個圈圈來表示，如圖 2.9 中的 Comparable。介面的實作用一條實線來表示，如圖中的 String, Integer, Student 都實作了 Comparable 介面。

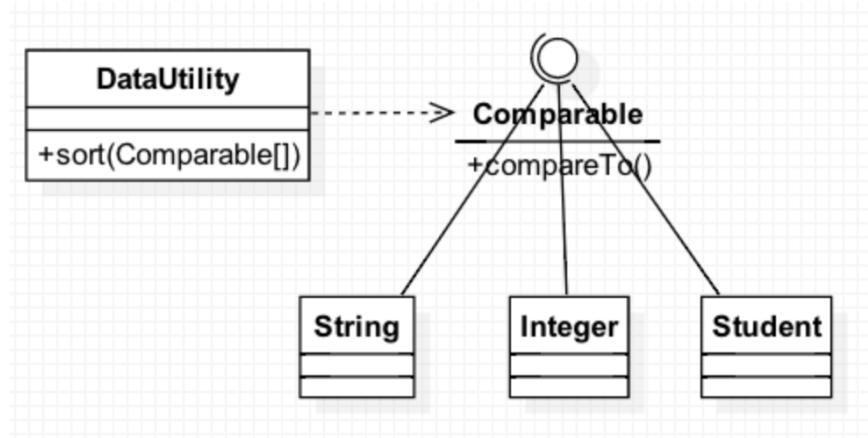


圖 2.9: 介面實作

介面的使用用一條虛線的「相依性」來表達，例如 `DataUtility` 中的 `sort(Comparable[])` 會將一群 `Comparable` 的物件做排序。

## 2.4.5 關聯

關聯主要在描述類別間的靜態結構關係。通常關聯都是二位關聯 (binary association)，表示「兩個」類別間的關係，用一條線連接兩個類別，如圖 2.10 說明了關聯的種類。

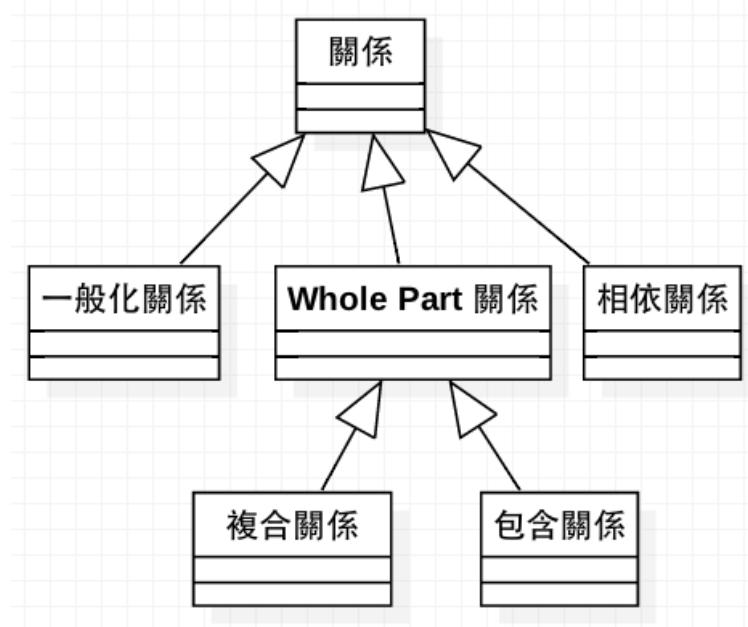


圖 2.10: 各種不同的關聯

當我們用關聯將兩個類別串起來時，代表它們的物件可能在某一段時間內有關係。比方說，「人為公司工作」，人與公司是兩個類別，而「工作」就是一個關聯。

**如何為關聯命名？** 一個好的命名可以讓分析師清楚的瞭解各個類別之間的關係，通常有兩個命名的方式，一個是直接對關連命名，另一種是在關連的兩端寫上角色。

角色命名可以明確的指出類別在此關聯上的角色。在圖 2.11 中，Professor 與 Course 有關聯，而且 Professor 在此關聯上的角色為 instructor，表示他為此關係上的講師身分。

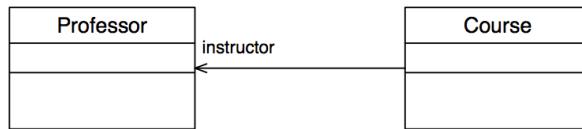


圖 2.11: 用「角色」為關聯命名

**Multiplicity** 每一個關聯的兩頭都該有一個 multiplicity value 描述一個物件在此關聯上可以和多少個物件相關。舉例來說，圖 2.12 表示 Company 在此關聯上的 multiplicity value 為 0..1，其意義為「任何 Person 只能在一家公司工作，或沒有在公司工作」。Person 的 multiplicity value 為 1..\*，表示「任何公司雇用一個以上的員工」。

其他 multiplicity value 所代表的意義描述如下：

1	恰好一個
0..1	零或一
M..N	M 到 N 個
	從零到任何正整數
0..*	從零到任何正整數 (強調包含零)
1..*	從一到任何正整數

在分析階段考慮物件間的 multiplicity 通常只指出是一對一、多對一或多對多，至於「多」到什麼程度、用何種方式實作都並不重要。在設計階段我們卻必須考慮這些問題，過分的高估 multiplicity value 會造成效率偏低，低估卻會造成執行時的錯誤。

**關聯類別 (Association Class)** 有時候關聯本身也有特性需要描寫。在上面的例子中，若我們需要分析每個人在公司的薪水、報到日期和職位時，就可以建立一個關聯類別：

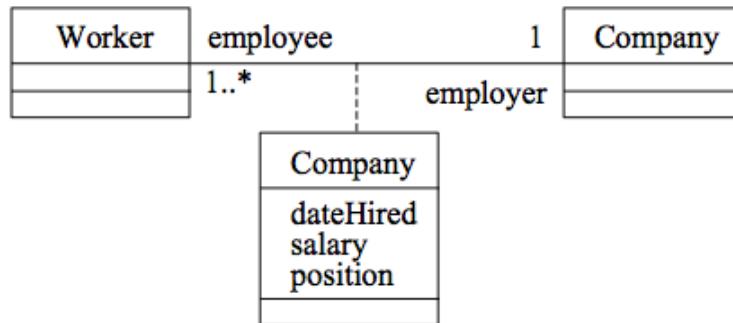


圖 2.12: 關聯類別

**瀏覽 (Navigation)** 在一般預設的情形下，關聯的瀏覽是雙向的，亦即關聯上的任一物件可以「瀏覽」另一物件。但在某些情況下，我們卻希望瀏覽是單向的。在圖 2.13 (c) 中，User 的物件可以找到相對應的 Password 物件，但卻不希望 Password 直接知道其相關的 User 物件為何。UML 用箭頭來表示瀏覽的方向，當沒有箭頭時，則表示該關聯為雙向瀏覽。

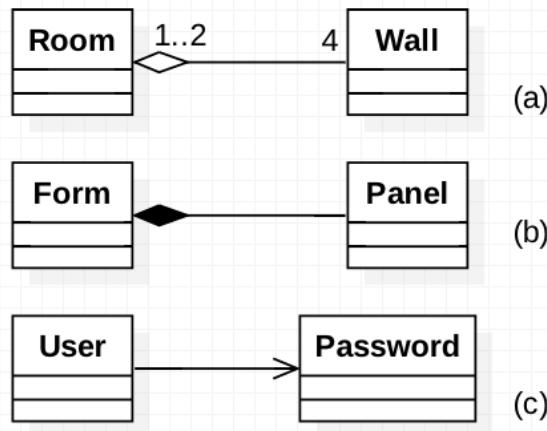


圖 2.13: 複合、包含、瀏覽關係

**Whole-Part 關係** 不論在現實生活中或軟體設計上都常常可以發現「某物體 B 是 A 物體的一部份」的例子，例如引擎是車子的一部份，房間是房子的一部份，Panel 是 Form 的一部份等。這種關係過去定義為單純的 Whole-Part 關係，UML 更進一步的討論，將之分為複合關係（Aggregation）與包含關係（Composition）關係，是管理軟體複雜性的一種重要機制。

複合關係表示單純的 Whole-Part 或 Ownership (擁有) 關係，一個 Part 可以有很多個 Whole 或 Owner，而且 Part 不會因為 Whole 的消失而消失。Room 與 Wall 就是一個典型的 Aggregation 關係，因為 Wall 是 Room 的一部份，但是 Wall 的生命週期並不決定於 Room 的生命週期，一個 Wall 也不僅是 Room 的一部份。

包含關係表現一種更強烈的 Whole-Part 關係，一個 Part 僅可以屬於一個 Whole，而且 Part 會隨著 Whole 的消失而消失。例如當我們在一個 Form 中加入一個 Panel 時，就建立一個包含關係，其中 Panel 是 Part，Form 是 Whole。一個 Panel 僅可以屬於一個 Form，而且 Panel 的生命週期是跟著 Form 的。

複合與包含在圖形上的差別在於關係上的菱形是否為實心：實心表示包含，空心表示複合。他們的差別整理如下：

意義	Multiplicity	生命週期
複合	Part 可屬於一個以上的 Whole	無關
包含	Part 僅屬於一個 Whole	Part 相依於 Whole

Navigation、Composition 與 Aggregation 這三個關聯是語意相異但實作相同。因為當這些關係被實作為程式碼後都是一個物件被宣告成另一個物件的屬性，在實作上無法區別

他們的不同。這也是為什麼 CASE (Computer-Aided Software Engineering, 電腦輔助軟體工程) 很難從原始碼中畫出物件導向模組的原因。

**依靠關係 (Dependency)** 依靠關係是一種單向關係，當類別 A 依靠類別 B 時，表示 B 的變動可能會影響到 A 的行為，另一種說法是 A 使用 (use)B 的規格。依靠關係不僅可以用在類別圖，也可以用在元件圖中表示軟體架構。在 UML 中，依靠關係是以有箭頭的虛線來表示。圖 2.14 表示類別 A 依存於類別 B：



圖 2.14: 依靠關係

Dependency 與 navigation 常令人混淆，為了清楚瞭解他們的差別，我們從物件間的可視性 (visibility) 來分析。一般而言，關係上的可視性可以區分為四種：

- 屬性可視 (attribute visibility)：B 是 A 的一個屬性。
- 參數可視 (parameter visibility)：B 是 A 某一個 method 的參數。
- 局部可視 (locally declared visibility)：B 是 A 某一個 method 內宣告的區域變數。
- 全域可視 (global visibility)：B 是全域變數。

在物件導向系統中，功能的完成是靠物件間相互傳遞訊息 (message) 所致。若 A 要送一個訊息給 B，則 A 必須要「看得到」B，也就是說，B 對 A 而言是可視的。一般而言，navigation 用以表示屬性可視，而 dependency 用以表示參數可視與局部可視。

有時候我們想要明確的指出關聯可視性，而不想用 dependency 來表示，UML 利用 stereotype 來表示關聯可視性。

## 2.5 動態行為：狀態圖

狀態圖是用來描述一個物件的行為。這裡所謂的行為指的是物件如何處理與回應事件。狀態圖主要是由狀態 (state) 與狀態轉移 (transition) 所構成的。

### 2.5.1 狀態

大部分的物件都是具備狀態性 (stateful) 物件。例如電腦可以有開機狀態或關機狀態；汽車有熄火、發動、啟動、前進或後退等狀態；視窗有開啟、關閉隱藏等狀態；滑鼠有移動、靜止等狀態。不同的狀態表現物件不同的反應：當我們在開機狀態下按下 on/off 的按鈕時，電腦會關機；但當電腦在關機狀態下我們同樣的按下 on/off 按扭時它卻會開機。同樣的事件電腦卻有不同的反應，這是因為電腦對事件的處理取決於其內部的狀態。狀態在 UML 用圓角方形表示。

狀態的特徵：

- 狀態通常經歷一段較長的時間。例如在電話系統中，「鈴聲響」、「拿起話筒」、「講電話」是接電話的三個基本步驟。其中，「鈴聲響」與「講電話」通常會被當成兩個狀態，因為它們會經歷一段較長的時間，「拿起話筒」是一瞬間的事，應當被表達成一個事件。事件會造成狀態的轉移。又例如執行一個運算在電腦的世界裡是一瞬間的事，通常不會當成一個狀態。
- 狀態必須是對系統有意義的。例如在媒體租借系統中，一個媒體可以有三種狀態：在圖書館中 (in library)、被預借中 (reserved) 與外借中 (borrowed)。從圖 2.15 的模組中，我們可以清楚的瞭解到媒體的行為：
  - 媒體可以透過「預借」→「外借」或直接「外借」的方式被借出；
  - 除非它在外借中的狀態，否則對它下達歸還的事件是沒有用的。

雖然我們只指出三個狀態，這並不表示媒體沒有其它的狀態 – 比方說「被閱讀中」、「圖書管理員貼條碼中」、「正被某人拿著」等等都是媒體可能處在的狀態，可是這些資訊對圖書管理而言是不重要的，所以不需模組出來。系統模組中的狀態狀態必須具備「對該系統設計有意義」的特性。

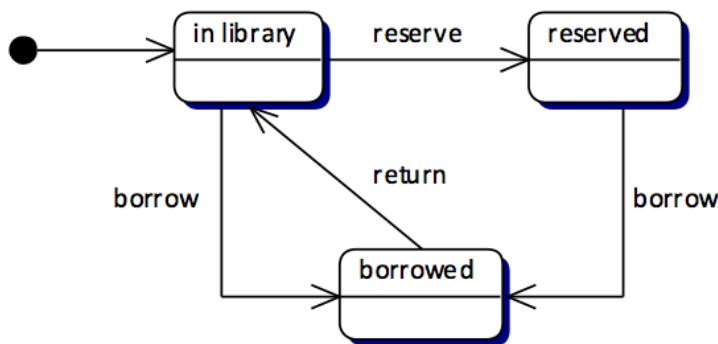


圖 2.15: 圖書館裡系統中圖書的狀態圖

**狀態與屬性** 物件的狀態與物件的屬性值有直接且緊密的關係。一個人的狀態可以分為「老」、「中」、「青」是因為這個狀態跟人的「年齡」的屬性相關；在上面的例子中，媒體的狀態可由屬性「借閱狀態」來加以模組。

### 2.5.2 狀態轉移

只有狀態是沒有辦法描述一個物件的行為的，物件會因為收到特定的事件而做了特定的工作而轉移到其它的狀態，這樣的轉移才構成所謂的行為。

轉移用具有方向性的連接線表示。一個轉移具有下面幾個部分：

- **原始狀態 (source state)**：此與目的狀態都是狀態轉移必要的元件。原始狀態是轉移前的狀態，當物件在原始狀態收到事件，且滿足轉移條件時就會進行狀態轉移，而進入目的狀況。
- **驅使事件 (event trigger)**：驅使物件做狀態轉移的事件稱為驅使事件。在上例中，「解聘」、「雇用」、「大於 60 歲」等都是驅使事件。在某些時候我們並不特別註明驅使事件：當原始狀態內的活動結束時會自動的驅使物件進入目的狀態，稱為**無驅使轉移** (triggerless transition)。
- **轉移條件 (guard condition)**：有時候物件收到驅使事件後不一定會做狀態轉移 – 除非它滿足某個條件，這個條件就稱為轉移條件 (guard condition; 守衛狀況)。轉移條件以 [condition] 來表示。
- **行動 (action)**：「行動」在 UML 的表示方法是緊接在事件之後，用「/」區隔開來。「event/action」所代表的意義是「當事件 event 發生後，物件會執行行動 action，而進入到下一個狀態」。比方說發生「掛斷電話」這個事件，會帶出「終止連線」的行動，而導致電話進入「idle」的狀態。
- **目的狀態 (target state)**：物件做完狀態轉移後進入的狀態。

簡言之：

一個物件在「原始狀態」時，接受到「驅使事件」且滿足「轉移條件」，就會執行「行動」，轉移到「目的狀態」。

### 2.5.3 複雜的狀態描述

完整的狀態包括五個部分：

- **名稱**：以文字的方式描述之。如果是一個「活動狀態」，即以活動名稱來命名。
- **進入行動 (entry action)**：表示物件進入此狀態馬上必須進行的活動。圖 2.16 中的  $op_1$  即為一個進入行動。
- **離開行動 (exit action)**：表示物件離開此狀態前必須要執行的行動。 $op_2$  即為一個離開行動。
- **狀態活動 (activity)**：表示物件在此狀態中一直在進行的作業。在 UML 中狀態活動是緊接在  $do/$  後，如  $op_4$  為一狀態活動。活動通常需要花一段時間才能完成，這與在狀態轉移上的行動 (action) 不同；行動通常與事件緊密相關，而且所花的時間非常短。
- **內部轉移 (internal transitions)**：當物件在某個狀態時，我們希望能夠模組它收到某個事件時該做的工作，但此工作並不會造成物件狀態的轉移。圖 2.16 中，物件收到事件  $i$  後會執行  $op_3$  且不會移到另一個狀態。要特別注意的是內部轉移與自我轉移 (self-transition) 不同，自我轉移指的是物件在目前的狀態後又重新進入該狀態，所以他會歷經離開行動與重新一次的進入行動，這些都是內部轉移所不會發生的。
- **子狀態 (substate)**：狀態內還有狀態稱之為子狀態。關於子狀態會在後面再做說明。
- **延遲的事件 (deferred events)**：在本狀態不做處理的事件。物件會將這些事件 queue 起來讓之後的狀態處理。

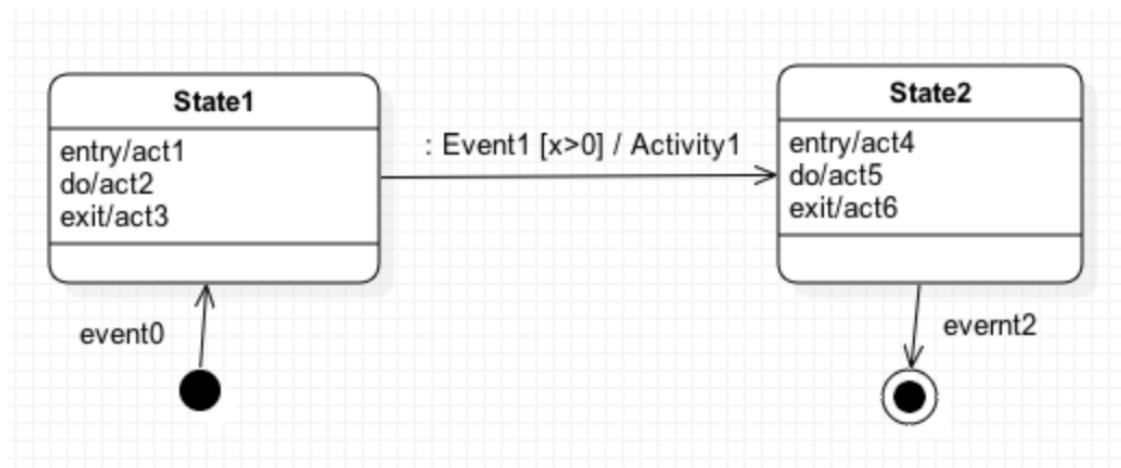


圖 2.16: State transition diagram

#### 2.5.4 一般化與合成化

如同 UML 的物件圖一般，為了使狀態圖更結構化，狀態圖也具有一般化 (generalization) 與合成化 (aggregation) 的觀念。狀態的一般化與合成化可以降低狀態圖的複雜度。

**狀態的一般化** 電腦的「開機狀態」又可細分為「操作狀態」、「休眠狀態」、「待命狀態」或「當機狀態」。

所謂的一般化是指狀態之間具備「是一種」(is-a) 的關係，例如車子的狀態分為前進檔、後退檔與空檔，而前進檔又可分為一檔、二檔與三檔；我們可以說一檔是一種 (is-a) 前進檔，或說一檔是前進檔的子狀態。一般化也同時代表著「或」的關係，當物件處於父狀態 ( $S$ ) 時它必定處在某一個子狀態中 (物件處於  $S_1$  或  $S_2$  中，假設  $S_1$  與  $S_2$  為  $S$  的子狀態)。在圖 2.17，車子的變速器處於前進檔時，它必定同時處於一檔或二檔或三檔。由於子狀態之間存在循序的關係，因此子狀態稱為「循序子狀態」(sequential substate)。

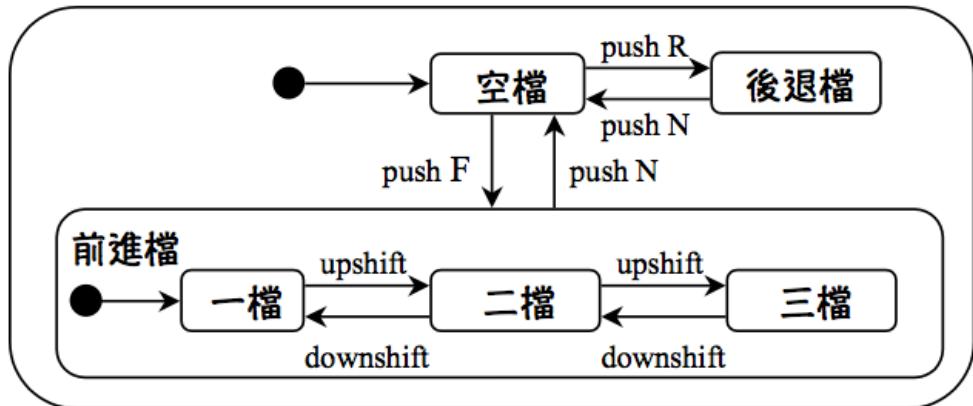


圖 2.17: 變速器的狀態圖

循序子狀態包含起始狀態表示當進入此狀態時即立刻進入的狀態。在上述的例子中，當在狀態空檔時發生  $pushF$  的事件，物件會立刻進入一檔的狀態，因為它是前進狀態的起始狀態。

狀態的一般化可以解決狀態圖複雜的問題，在圖 2.18 中可以發現  $S_1$  與  $S_2$  有相同的行为 – 當它們收到事件  $E$  時都會轉移到狀態  $S_3$ 。我們可以將之一般化為圖右的情形，可以明顯的看到狀態轉移由三個簡化為兩個。

雖然平版狀態（圖左）讀只比巢狀（圖右）的多了一個狀態轉移，可是可讀性明顯降低了很多。當狀態及狀態轉移數量多的時候，巢狀與非巢狀的複雜度就可以明顯的觀察出來。

**狀態的合成化** 狀態的合成表示一個狀態是由許多的子狀態所合成的。這種合成是一種「且」的關係，也就是說物件必須「同時」處在合成的子狀態中。因此這些子狀態稱為並行子狀態 (concurrent substate)。圖 2.19 中  $S_1$  與  $S_2$  同為  $S$  的並行子狀態，這表示當物件在  $S$  狀態時，它必定同時在  $S_1$  與  $S_2$  狀態 (回顧一下狀態的一般化，子狀態間的關係是或的關係)。

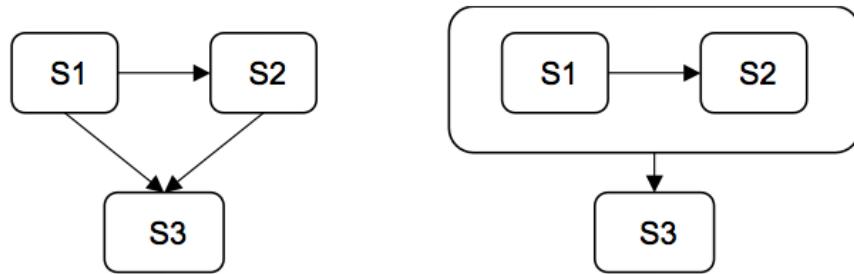


圖 2.18: 狀態一般化

因為合成化的關係，我們必須用一對或多個狀態 –  $(S_1, S_2)$  來表示該物件目前的狀態。圖中物件的起使狀態是  $(Z, A)$ ，受到  $E_2$  事件驅使後進入  $(X, A)$ ，若再受到事件  $E_4$  驅使後則進入  $(X, B)$ 。請注意即使只有一個子狀態作改變，我們仍須用一對狀態來描述。

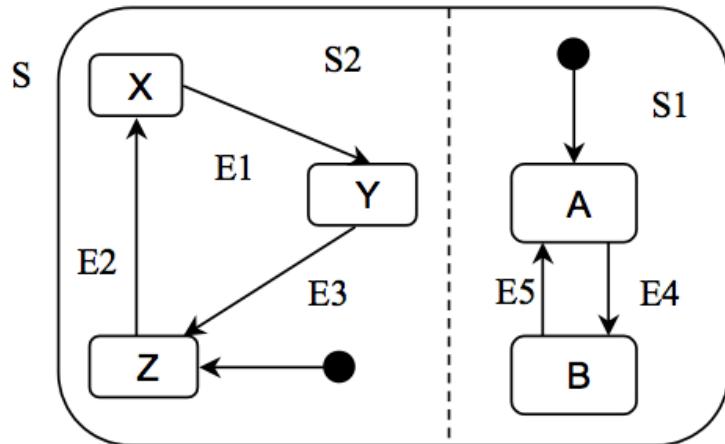


圖 2.19: 狀態合成化

狀態一般化與合成化都是有效解決狀態圖複雜的利器，下表比較兩者的不同：

## 2.6 物件互動：循序圖

循序圖與合作圖都表示物件交互合作的情形，但它的重點在於訊息傳遞的先後順序上。物件仍是以方形表示，每個物件並且有其生命線 (lifeline)，以虛線表示。

循序圖可以用來表達物件之間互動的順序。通常是針對一個情境的互動狀況。

- 物件。參與此活動的物件。在 UML 中以方形來表示，圖中的  $object_1, object_2,$

	狀態一般化	狀態合成化
子狀態名稱	循序子狀態	並行子狀態
子狀態與父狀態關係	是一種 (is-a)、一般化	部分 (part-of)、合成化
子狀態與子狀態間的關係	或	且

表 2.2: 狀態一般化與狀態合成化比較

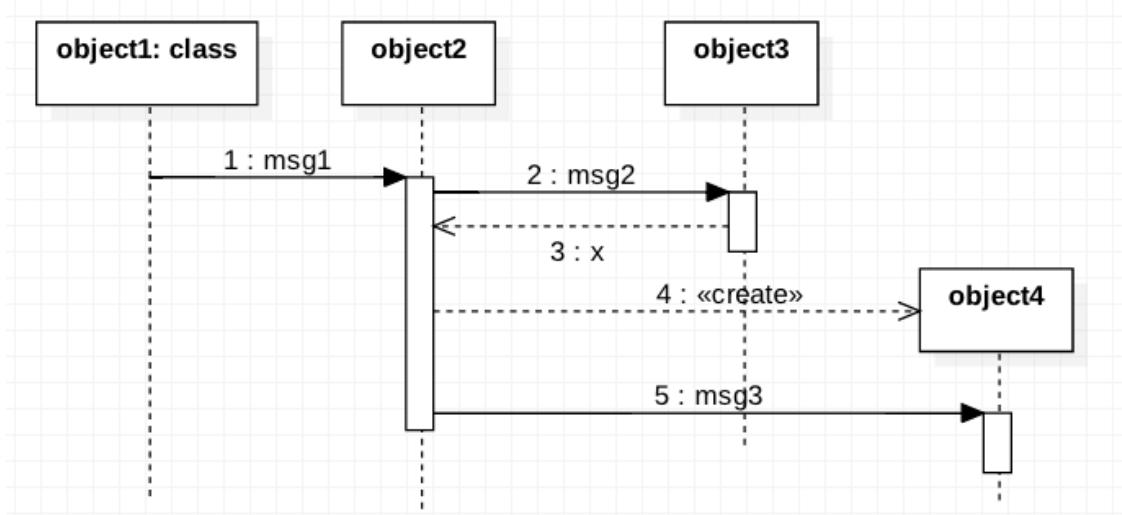


圖 2.20: 循序圖

*object<sub>3</sub>, object<sub>4</sub>* 皆為物件。

- 生命線。物件之間溝通的訊息以垂直的箭頭表示，訊息線的高低代表著訊息產出的先後順序，也因為如此，循序圖不需要以序號來表示其先後的關係。
- 活化段。物件 A 送訊息給物件 B 因此促發了物件 B 的一段工作，稱之為「活化段」(activation)，生命線上的長條方形即為活化段。物件 B 的活化段包含在物件 A 的活化段之中，代表執行的控制權由 A 轉移到 B，一直到 B 執行結束才將執行權歸還給 A。活化段的長短不表示工作的時間長短。
- 訊息傳遞。如圖中的 *msg<sub>1</sub>*, *msg<sub>2</sub>* 等。產生物件的訊息比較特別，透過 stereotype 來表示。訊息傳遞給物件後，物件會執行相對應的方法，如果有回傳值的話，就以虛線來表示，圖中的 *x* 表示回傳的值。

繪製循序圖是以完成某一個完成功能或情境為主體，可以幫忙定義物件的方法。

**象棋系統** 圖 2.21 是部分的循序圖，描述一個玩家先建立一個棋局遊戲 ChessGame, 接著另一個玩家加入。加入後 ChessGame 會建立 ChessBoard 來呈現整個棋盤，玩家接著對棋盤做互動，互動的事件會由棋盤轉換為對 ChessGame 有意義的指令，ChessGame 每一次做動作都會進行 checkWinner 來檢查是否勝負已定，如果已定就會傳訊息給 ChessBoard, 接著由 ChessBoard 公布訊息給玩家。

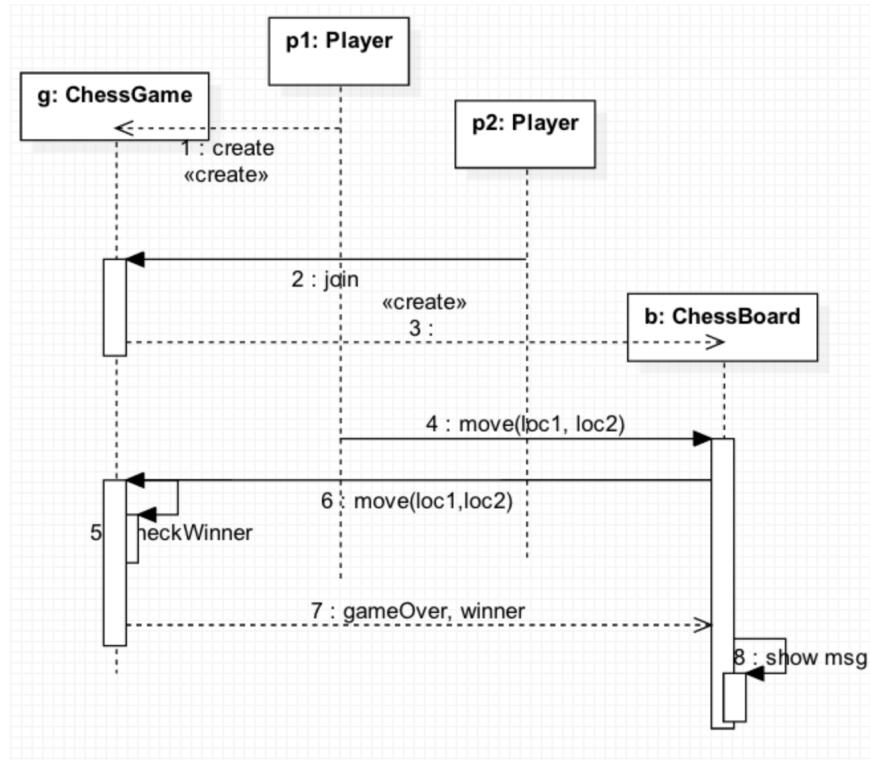
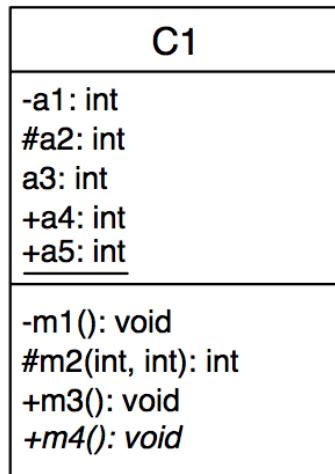


圖 2.21: 象棋系統的部份循序圖

## 2.7 程式碼對應

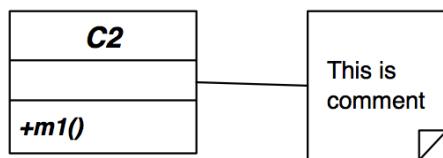
設計為程式開發鋪路 [?]。



其對應的程式碼如下：

```
1  class C1 {  
2      private int a1;  
3      protected int a2;  
4      int a3;  
5      public int a4;  
6      public static int a5;  
7  
8      private void m1() {…}  
9      protected int m2(int a, int b) {…}  
10     public void m3() {…}  
11     abstract public void m4() {…}  
12 }
```

註解



```

1 //This is comment
2 abstract class C2 {
3     public void m1();
4 }
```

## 關連

當兩邊的類別彼此都可看到對方時，我們可以用沒有箭頭的關係圖連接起來。



```

1 class ClassRoom {
2     Course course;
3 }
4 class Course {
5     ClassRoom room;
6 }
```

## Navigation

學生可以看到課程，但是課程看不到學生。



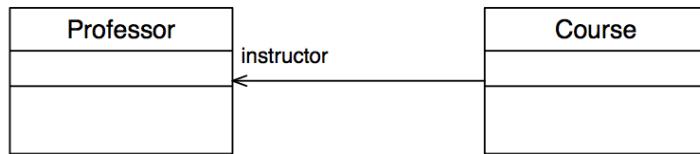
注意 Student 內有一個 course 的參考，但 Course 內沒有對 Student 的參考。

```

1 class Student {
2     Course course;
3 }
4 class Course {
5 }
```

## Role name

在 Association 的端點上的 role name, 可轉換為類別中的屬性。此屬性記錄參與此關係的另一個物件。

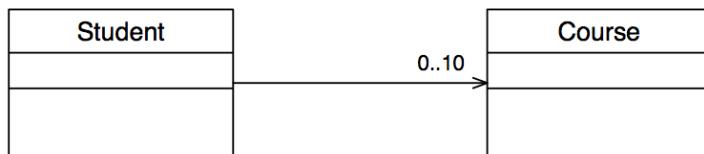


```

1   class Professor {
2   }
3   class Course {
4       Professor instructor;
5   }
  
```

## Multiplicity

一個學生可以 0 到 10 門課。這時候可以在關連端點上寫上數量 (multiplicity)。



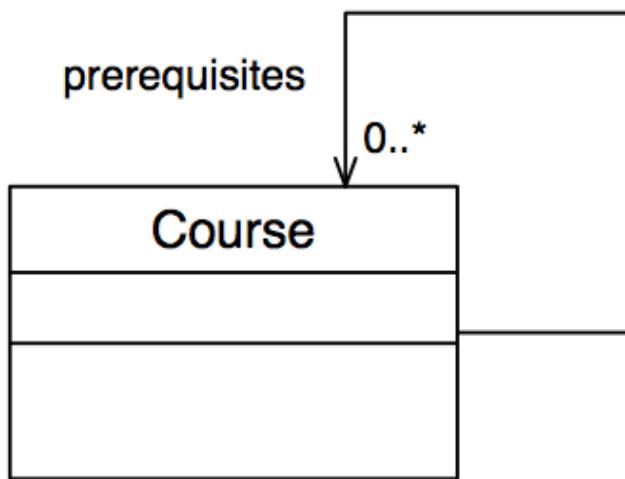
在實作上可以用陣列來實作這樣 [多數量] 的關係。

```

1   class Student {
2       Course[] courses = new Course[10];
3   }
4   class Course {
5   }
  
```

## Self association

同一個類別的物件彼此有關係。例如課與課之間有 [先修] 的關係。



在實作上仍然用物件參考來表示這個關係。因為數量是  $0..*$  表示我們並不確定明確的數量。這時候用 vector 這種動態陣列比較合適。

```

1  class Course {
2      Course[] prerequisites = new Vector();
3  }
  
```

## Aggregation



```

1  class Student {
2      Vector course;
3  }
4  class Course {
5      Student s;
6  }
  
```

## Composition

如果包含的關係是比較強烈的：當被包含者的生命週期是由包含者來控制時，則用實心的包含來表示。

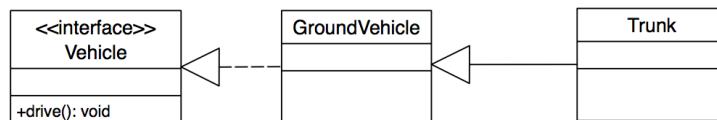


這時候 vector 的生成是當 Student 生成時就產生的。

```

1  class Student {
2      Vector course = new Vector();
3  }
4  class Course {
5      Student student;
6  }
  
```

## Interface implementation



```

1  interface Vehicle {
2      public void drive();
3  }
4  class GroundVehicle implements Vehicle {
5      public void drive() { ...
6      }
7  }
8  class Trunk extends GroundVehicle {
9      ...
10 }
  
```

## 2.8 練習

### UML 簡介

**Ex 1:** 室內設計圖有沒有什麼標準？用來畫畫你的房間。

**Ex 2:** 除了軟體工程以外，還有哪些工程也有圖模的概念？試說明之。

**Ex 3:** 一個好的模組語言，應該具備什麼特性？

### 使用案例圖

**Ex 4:** 使用案例的目的是描述系統的 (1) 類別架構 (2) 系統架構 (3) 系統功能 (4) 操作的情境

**Ex 5:** 關於一個象棋系統，設計其使用案例

**Ex 6:** 關於 ATM 提款，描述其設計案例

**Ex 7:** 關於 YouBike，描述其使用案例

### 類別圖

**Ex 8:** 類別圖不能表現一個物件類別的 (1) 屬性 (2) 功能 (3) 演算法 (4) 責任

**Ex 9:** \* 在武俠的世界中，有一些重要的「要素」，例如「人物」、「武功」、「武器」等。還有哪些要素？如果把每一個要素當成一個類別，他們會有哪些屬性？類別之間會有關係，例如「喬峰具備降龍十八掌」的「具備」關係、「黃蓉擁有打狗棒等」的「擁有」關係。請至少找出五個類別，分析他們之間可能的關係，以一些你看過的武俠小說為例來舉例。

**Ex 10:** 「老師」具備姓名、年資、專長等屬性，具備教書、研究等方法；「課程」具備名稱、代號、學分數等屬性；「學生」具備姓名、學校、年級等屬性，具備修課、問問題等方法。(1) 以 StarUML 為工具，建立一個 AnalysisModel，繪製這三個類別，及其屬性、方法 (2) 繪製這三個類別間的關係，除了用關係命名外，也用角色命名 (3) 繪製關係時，採用 StarUML 的 Explorer 面板，設定關係的 multiplicity (4) 把老師分為專任教師兼任教師，其中兼任教師學分數不可大於 6 學分，使用 Note link 來建立此限制

**Ex 11:** 同上，建立一個 DesignModel (1) 把上述的 AnalysisModel 複製到此 Model, 修改各屬性，為其加上型態 (2) 修改各方法，加上每個參數。

**Ex 12:** 考慮一個線上考試系統，繪製其類別圖。

**Ex 13:** 班級活動的籌劃，可能會牽涉班長、活動、旅行社、房間、交通等「物件」(可能更多或更少)，繪製其類別圖。

## 狀態圖

**Ex 14:** 狀態圖主要表現系統的 (1) 功能 (2) 操作情境 (3) 行為 (4) 物件結構

**Ex 15:** 看圖說故事，圖 2.22 表示什麼意義？有人訂了機票後，還可以取消航班嗎？

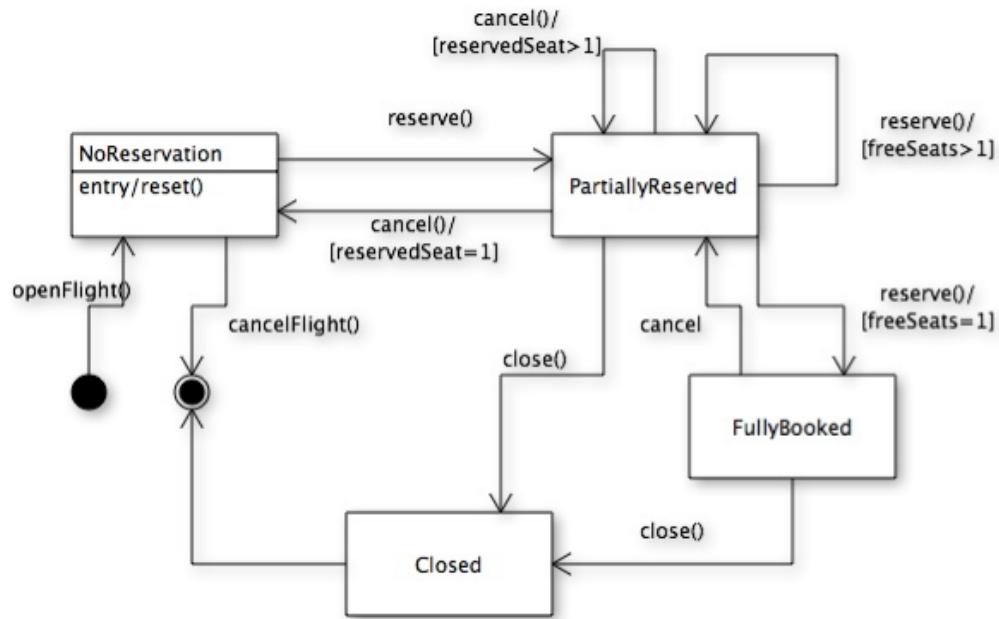


圖 2.22: 航班預定系統 - 航班狀態圖

**Ex 16:** 畫出 Chess, ChessGame 的狀態圖。

**Ex 17:** 考慮一個線上測驗系統，考試的狀態圖為何？哪些是狀態、狀態轉移、事件？有沒有 guard condition？有沒有活動狀態（activity state）？

**Ex 18:** 狀態的合成化與物件的合成有直接的關係。在車子的例子中，假設車子是由變速器、加速器與煞車所構成的，那麼車子的狀態恰是由這三個子物件的狀態所構成的，請繪製汽車的合成狀態圖。

## 循序圖

**Ex 19:** 繪製循序圖時，可能需要參考其他的圖型，但以下何者不太可能需要參考：(1) 使用案例 (2) 類別圖 (3) 系統配置圖 (4) 活動圖

**Ex 20:** 考慮以下的程式，繪製其循序圖

```

1  public class BillingDialog {
2      public static void main(String[] args) {
3          Bill yourBill = new Bill();
4          yourBill.inputTimeWorked();

```

```

5             yourBill.updateFee( );
6             yourBill.outputBill( );
7         }
8     }
9
10    public class Bill {
11        public static final double RATE = 150.00;
12        private int hours, minutes;
13        private double fee;
14
15        public void inputTimeWorked( ) {
16            System.out.println("請輸入工作幾小時幾分鐘（
17                空白分開）");
18            Scanner keyboard = new Scanner(System.in);
19            hours = keyboard.nextInt();
20            minutes = keyboard.nextInt();
21        }
22
23        private double computeFee(int hoursWorked, int
24            minutesWorked) {
25            minutesWorked = hoursWorked*60 + minutesWorked;
26            int quarterHours = minutesWorked/15;
27            return quarterHours*RATE;
28        }
29
30        public void updateFee( ) {
31            fee = computeFee(hours, minutes);
32        }
33
34        public void outputBill( ) {
35            System.out.println("你工作了" + hours + "小時" +
36                minutes + "分鐘");
37            System.out.println("共賺" + fee + "元");
38        }
39    }

```

**Ex 21:** 象棋系統中，Chess, ChessBoard, ChessGame 的循序圖為何？

**Ex 22:** 班級活動的籌劃，可能會牽涉班長、活動、旅行社等「物件」，繪製循序圖來完成相關的情境。

## 綜合

- Ex 23:** 使用 StarUML 工具，安裝 java code generation 的套件。繪製本節所講授的內容，並透過工具轉換為程式碼，並檢驗之。
- Ex 24:** 繪製 Chess 的類別圖，對應成程式碼，並完成程式碼。
- Ex 25:** 針對象棋系統，以 UML 設計其系統，包含使用案例圖、類別圖、狀態圖、循序圖等。
- Ex 26:** 設計一個角色扮演遊戲（例如金庸群俠傳），以 UML 設計其系統，包含使用案例圖、類別圖、狀態圖、循序圖等。

# Chapter 3

## 心法一：軟體設計原則



## 3.1 涼渭分明：模組化原則

把問題切割成若干適合管理的單元，再將之組合成需要的功能。

所謂的模組化，就是一種「切割、解決」(divide and conquer)的基本想法，也就是說，將一個大問題拆解成若干個小問題之後，透過逐一解決這些小問題，來解決整個大問題。

但要注意也不是切的越小就越好，不斷地拆解下去，會使得系統中的模組數量大幅增加，而一旦數量增加到一定程度之後，其管理與理解的負擔和成本就會隨之升高。

模組化可以帶來許多的好處。例如，模組化可以提昇重用性 (reusability)，因為，每個獨立的模組都有機會靈活運用到不同的專案中。模組化也可以帶來可擴充性和可測試性。

設計時，需讓模組具備的特質

- 可分解性 (Decomposability)。可以將一個大問題拆解成小問題，並且透過各個小問題的解法來解決大問題，而且在分解之後，可以將這些小問題指派給不同的人獨立去解決。「由上而下分解 (top-down decomposition)」即為如此，從高階的抽象化觀點出發，逐漸具體化到可以實作的程度。
- 可合成性 (Composability)。允許設計者將多個模組依自己的意念，組合在一塊。具可合成性的模組提供了重複使用能力，這使得模組可以應用在當初開發模組時的情境以外的其他情境，而且具備相同介面的模組間，還可以互相的抽換。
- 可理解性 (Understandability)。降低理解模組運用情境所需的知識。當一個模組所涉及的其他模組愈少時，那麼它的可理解性就愈高，因為，只需要了解這個模組的特性及行為，就可以了，無需一個牽連一個，使得運用單一模組時，還得同時了解眾多其餘的模組特性及行為。
- 連續性 (Continuity)。當規格有了小幅度的更動時，受到影響的模組的個數要愈少愈好。
- 保護性 (Protection)。保護性指的是，在某個模組內所發生的執行期錯誤，其影響到的模組應該愈少愈好，最好只影響到其本身。

### 3.1.1 低耦高聚原則

不同模組之間的相依性（耦合性）應該要儘量的低；同模組內成員相關性（內聚力）要儘量的高。

**耦合性** 耦合性可以是低耦合性（或稱為鬆散耦合），也可以是高耦合性（或稱為緊密耦合）。以下列出一些耦合性的分類，從高到低依序排列：

- 內容耦合（content coupling，耦合度最高）也稱為病態耦合（pathological coupling）是指一個模組依賴另一個模組的「內部」作業（例如，存取另一個模組的局域變數），因此修改第二個模組處理的資料也就影響了第一個模組。
- 共用耦合（common coupling）也稱為全局耦合（global coupling.）是指二個模組分享同一個「全局變數」，因此修改這個共享的資源也要更動所有用到此資源的模組。
- 控制耦合（control coupling）是指一個模組藉由傳遞「要做什麼（flag）」的資訊，控制另一個模組的流程。一個模組影響到另一個模組執行的流程的程度。
- 資料耦合（data coupling）是指模組藉由參數傳遞來相互合作，每一個資料都是最基本的資料（atomic data element）。資料耦合意味著：(1) 沒有多餘無用的資料; (2) 沒有控制參數; (3) 沒有外部、共享的資料結構;
- 無耦合：模組完全不和其他模組交換資訊。

緊密耦合的系統在開發階段有以下的缺點：

- 一個模組的修改會產生漣漪效應，其他模組也需隨之修改。
- 由於模組之間的相依性，模組的組合會需要更多的精力及時間。
- 由於一個模組有許多的相依模組，模組的可復用性低。

**內聚性** 內聚性（Cohesion）也稱為內聚力，是一軟體度量，是指機能相關的程式組合成一模組的程度，或是各機能凝聚的狀態或程度。是結構化分析的重要概念之一。

耦合性是一個和內聚性相對的概念。一般而言高內聚性代表低耦合性，反之亦然。內聚性在實務上可減少維護及修改的「好」軟體的特性為基礎。內聚性是指機能相關的程式組合成一模組的程度。應用在物件導向程式設計中，若服務特定型別的方法在許多方面都很類似，則此型別即有高內聚性。在一個高內聚性的系統中，代碼可讀性及復用的可能性都會提高，程式雖然複雜，但可被管理。

以下的情形會降低程式的內聚性：許多方法封裝在一型別內，可以藉由方法供外界使用，但方法彼此類似之處不多，對於內部資料的存取的共同性也很低。

低內聚性的缺點如下：

- 增加理解模組的困難度。
- 增加維護系統的困難度，因為一個邏輯修改會影響許多模組，而一個模組的修改會使得一些相關模組也要修改。

- 增加模組重用困難度，因為大部份的應用程式無法重用一個由許多不一定相關的機能組成的模組。

內聚性的分類如下，由低到高排列：

- 偶然內聚性（Coincidental cohesion，最低）指模組中的各部分只是剛好放在一起，各部分之間唯一的關係是其位在同一個模組中（例如：「Utility」模組，把一些常用的共用程式放一起）。
- 邏輯內聚性（Logical cohesion）只要在邏輯上分為同一類，不論各機能的本質是否有很大差異，就將這些機能放在同一模組中。
- 時間性內聚性（Temporal cohesion）相近時間點執行的程式，放在同一個模組中（例如在捕捉到一個異常後呼叫一函式，在函式中關閉已開啟的檔案、產生錯誤日誌、並告知使用者）。
- 程式內聚性（Procedural cohesion）依一組會依照固定順序執行的程式放在同一個模組中（例如一個函式檢查檔案的權限，之後開啟檔案）。
- 聯絡內聚性（Communicational cohesion）聯絡內聚性是指模組中的各部分因為處理相同的資料，因此放在同一個模組中。
- 依序內聚性（Sequential cohesion）依序內聚性是指模組中的各部分彼此的輸入及輸出資料相關，一模組的輸出資料是另一個模組的輸入，類似工廠的生產線。
- 功能內聚性（Functional cohesion，最高）模組中的各部分是因為它們都對模組中單一明確定義的任務有貢獻。

## 3.2 無雙無對：不重複原則

資料或是計算應該只存在一個地方，不要造成重複。

重複是邪惡的，很容易出錯。例如我們把一筆成績資料存在兩個不同的檔案  $a_1$  及  $a_2$ ，分別給  $m_1$  與  $m_2$  兩個模組來讀取，當資料修改成績時時必須同時修改  $a_1$  及  $a_2$  兩個檔案 - 一開始工程師可能還會記得這件事，但時間一久或交接沒有確實，就很容易忘了同步，造成程式的錯誤。不重複原則（Don't Repeat Yourself Principle; DRY）的原則就是不要描述再軟體設計時，不論是資料或是計算，都應該儘量的避免重複。

**Copy-Paste 程式開發** 工程師常有過這樣的經驗：需要某一段演算時，發現過去寫過的一段程式碼可以「再利用」，於是把它 copy 到現有的程式碼中，再修改掉部分不同的地方。

但日後發現共同的那部分的設計變了，絕大部分的時間你僅會修改其中的一個，而忽略掉另一個，這時候就造成「計算的不一致」，這是很多程式錯誤的來源。

一些建議：

- 不論多麼匆忙，不要走捷徑（copy paste, 資料重複）；
- 把共同的部份寫成一個方法，讓大家透過呼叫來共用；
- 不要因為方法短，就覺得麻煩或不需要；
- 透過參數化讓你的方法的重用性變高；
- 把你的方法抽象化到父類別。

## Point 實例

Point 是一個座標上的點，裡面儲存 x, y 軸的座標值。

```

1      class Point {
2          private double x, y;
3          void setX(double x) { this.x = x; }
4          void setY(double y) { this.y = y; }
5          double getX() { return x; }
6          double getY() { return y; }
7      }
```

如果我們又需要提供極座標（長度及角度）給其他物件，是否讓 Point 多宣告兩個變數來儲存呢？依據「不重複原則」，不要！

極座標可以由絕對座標計算出來，所以不要在用額外的欄位去儲存：需要的時候在計算即可。

```

1      private double x, y; //只儲存 x, y
2
3      double getRho() {
4          // 由 x, y 算出，不額外儲存 Rho
5          return Math.sqrt(x*x + y*y);
6      }
7      double getTheta() {
8          // 由 x, y 算出，不額外儲存 Theta
9          return Math.acos(x / rho);
10     }
11 }
```

```

12     void setRho(double rho) {
13         x = rho * Math.cos(theta);
14         y = rho * Math.sin(theta);
15     }
16     void setTheta(double theta) {
17         x = rho * Math.cos(theta);
18         y = rho * Math.sin(theta);
19     }

```

### 3.3 私財勿露：資訊隱藏原則

不要公開多餘的資訊與服務。

以一個 Client-Server 架構來說明資訊隱藏的概念，Client 根據規格對伺服器提出一個請求，伺服器給予 Client 一個適當的回應，此時伺服器端提供 Client 所需的資訊、知識、或者運算，客戶端不需要知道伺服器端如何執行運算，只要能夠取得它所需的資訊即可，此時伺服器端對客戶端做運算方法的資訊隱藏。

*Each module has a secret design involves a series of decision: for each such decision, wonder who needs to know and who can be kept in the dark.*

### Sort 實例

以下程式有何問題？

```

1  public void sort() {
2      for (int i=0; i< data.length; i++)
3          for (int j=0; j<data.length-i; j++)
4              if (data[j] > data[j+1])
5                  swap(j, j+1);
6  }
7
8  public void swap(int x; int y) {
9      int temp = data[x];
10     data[x] = data[y];
11     data[y] = temp;
12 }

```

## 3.4 生人勿語：迪密特原則

每個單元只和它的朋友交談，不和陌生單元交談。

迪密特<sup>1</sup>原則（Law of Demeter; LoD）又稱為最少知識原則（Principle of Least Knowledge），或「不要跟陌生人交談」(Don’t Talk to Strangers)，其意義為

- 每個單元對於其他的單元只能擁有有限的知識：只是與當前單元緊密聯繫的單元；或
- 每個單元只能和它的朋友交談：不能和陌生單元交談；或
- 只和自己直接的朋友交談。

設計系統時必須注意類別的數量，並且避免製造出太多類別之間的耦合關係。一個簡單的例子是，人可以命令一隻狗行走，但不要命令狗的腿行走。人跟狗有關係，狗跟他的腿有關係，整個系統有兩個關係；若人又跟狗的腿有關係，整個系統就會有三種關係。

```

1   class Register {
2       private Sale sale;
3       public void getAmout() {
4           Money amt = sale.getPayment().getTenderedAmount();
5       }
6   }
```

`sale.getPayment().getTenderedAmount()`如同人命令狗的腿行走一般，可以修改為：`sale.getTenderedAm`

一個物件盡可能的少知道其他物件，反之，物件本身提供最少訊息給其他物件。也就是說，一個物件本身提供最少的 public variable 及 public method 級外界使用。

什麼是「可以直接呼叫的朋友」？假設  $obj_1$  呼叫  $obj_2$

- 屬於同一個類別型態；
- $Class_1$  中有方法的參數型態是  $Class_2$ ；
- $Class_1$  中有一個成員變數的型態是  $Class_2$ ；
- $obj_2$  是由  $obj_1$  所建構的物件。

---

<sup>1</sup>希臘神話的農業女神，因曾孤獨的尋找其女兒，取其孤獨之意。

**Just do it, I don't like to know  
the STRANGER**



### 3.5 不變應萬變：開畢原則

在不修改程式的情況下擴充或修改程式。

開畢原則（Open for extension and Close for modification; OCP）的意義即為「擴充程式優於修改程式」。當我們完成一個程式或類別後，日後有很大的可能性要去擴充它的功能，這時候大部分的程式設計師會將原有的程式叫出來修改，修改程式會帶來許多問題：

- 修改者對原有程式了解不深，修改後的程式也許能滿足新的功能，舊的功能確有可能被破壞。
- 修改者通常在應急的心態下修改程式，沒有整體性的計劃，出現錯誤的機會高。
- 程式難以元件化或模組化，因此再使用的機會低。我們所希望的是：一個類別有其主要負責的功能，當它被完成時，它可以被封裝成一個元件，以後即使有新增的需求也只要擴充其原有的類別即可，而非去修改它。

把「東西」從程式碼中抽來出來就很容易達到 OCP 的要求，例如資料不要「寫死」在程式碼中，抽離出來到一個檔案或資料庫，執行前只要將資料和程式碼做一個綁定（bind）就可以了，日後抽換資料時，只要換一個檔案就好了。

有許多設計樣式均提供透過物件設計的技巧來擴充程式碼，而不會改變已經存在的程式碼之 OCP 技術，例如 Decorator、Factory Method、Observer、Template Method 等。

## Shape 實例

一個繪圖編輯器類別要繪製不同形狀的圖形，呼叫形狀 (Shape) 抽象類別提供的抽象方法為統一服務介面。設計形狀類別的繼承架構，實作不同形狀的繪製子類別，例如矩形 (Rectangle)、圓形 (Circle)、或其他形狀。若要增加需求繪製新的形狀，則設計新的繪製形狀的子類別，達到 OCP 開放原則；如此並不會修改到原先已經存在的類別程式碼，滿足 OCP 的關閉原則。其程式碼如下所示。

```

1  abstract class Shape {
2      abstract public void draw();
3  }
4  //draw the Rectangle
5  class Rectangle extends Shape {
6      public void draw() {
7          }
8      }
9  //draw the Circle
10 class Circle extends Shape {
11     public void draw() {
12         }
13     }
14 //draw another shape
15 ...

```

簡單來說，就是透過多型的方式來避免程式的修改。

## Price 實例

假設一個主機板的價格是透過計算每一個零件的總和而得的：

```

1  public double totalPrice(Part[] parts) {
2      double total = 0.0;
3      for (int i=0; i<parts.length; i++) {
4          total += parts[i].getPrice();
5      }
6      return total;
7  }

```

其中的 part 表示每一總可能的零件，part.getPrice() 獲得該零件的價格。倘若今天有新的需求變更：主機板必須漲價四成五、記憶體必須調漲兩成七。

**方案一** 下述的作法直接利用 `instanceOf` 來判斷是不是主機板、記憶體，雖然可以解決問題，但 `totalPrice()` 這個方法就被修改了。這與 OCP 原則相違背。

```

1   public double totalPrice(Part[] parts) {
2       double total = 0.0;
3       for (int i=0; i<parts.length; i++) {
4           if (parts[i] instanceof Motherboard)
5               total += (1.45 * parts[i].getPrice());
6           else if (parts[i] instanceof Memory)
7               total += (1.27 * parts[i].getPrice());
8           else
9               total += parts[i].getPrice();
10      }
11      return total;
12  }

```

**方案二** 下述的作法，透過繼承來修改價格。從 `getPrice()` 下手，如要修改價格策略時就修改 `getPrice()` 的內容。

```

1  public class Part {
2      private double basePrice;
3      public void setPrice(double price) {basePrice = price;}
4      public double getPrice() { return basePrice;}
5  }
6  public class ConcretePart extends Part {
7      public double getPrice() {
8          // return (1.45 * basePrice); //Premium
9          return (0.90 * basePrice); //Labor Day Sale
10     }
11 }

```

但這樣的方式還是很「髒」，`getPrice()` 會被不斷的修改。

**方案三** 下述的方法把價格策略抽象出來成為一個類別，不同的計價策略（例如打折策略）繼承價格策略後修改 `getPrice()` 的方法。

```

1  public class PricePolicy {
2      private double basePrice;
3      public void setPrice(double price) {
4          basePrice = price;

```

```

5         }
6     public double getPrice() {
7         return basePrice;
8     }
9 }
10
11 public class SalePrice extends PricePolicy{
12     private double discount;
13     public void setDiscount(double discount){
14         this.discount =discount; }
15     public double getPrice() {
16         return (basePrice * discount);
17     }
18 }
```

Part 物件在 getPrice() 時，是委由策略物件來做回傳：

```

1 public class Part {
2     private PricePolicy pricePolicy;
3     public void setPricePolicy(PricePolicy policy) {
4         pricePolicy =policy;
5     }
6     public void setPrice(double price) {
7         pricePolicy.setPrice(price);
8     }
9     public double getPrice() {
10        return pricePolicy.getPrice();
11    }
12 }
```

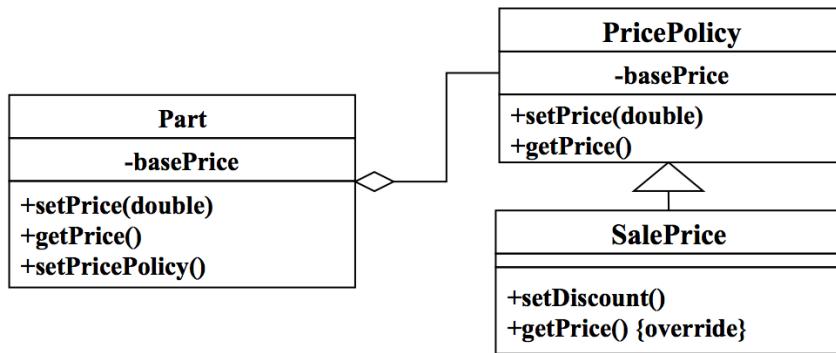
下圖為整個方法的架構圖：

## 迷宮實例

我們以 Gamma 一書所引用的迷宮的遊戲來說明。考慮一個迷宮的遊戲，其中包含 MazeGame、Maze、Room、Wall 及 Door 等類別。MazeGame 類別中有 createMaze() 方法以建立 Room、Wall、Door 等物件，並建立這些元件之間的關係。

```

1 public class MazeGame {
2     // Create the maze.
```



```

3   public Maze createMaze() {
4       Maze maze = new Maze();
5       Room r = new Room();
6       Wall w = new Wall();
7       Door d = new Door();
8       w.setDorr(d); //設定 r, w, d 之間的關聯
9       r.addWall(w, NORTH);
10      ...
  
```

**方案一** 假設現在我們想擴充系統的功能，讓迷宮是由 EnchantedRoom（有魔力的房間）、EnchantedWall、EnchantedDoor 等所構成，而這些類別都分別繼承自 Room、Wall、與 Door。為了要讓 MazeGame 用到新的房間、牆壁與門的元件，程式有必要修改：

```

1  public class MazeGame {
2      // Create the maze.
3      public Maze createMaze() {
4          Maze maze = new Maze();
5          Room r = new EnchantedRoom();
6          Wall w = new EnchantedWall();
7          Door d = new EnchantedDoor();
8          w.setDorr(d);
9          r.addWall(w, NORTH);
10         ...
  
```

然而，此方法卻違背了 OCP 的原則，因為我們去修改到了 MazeGame 了。因此，MazeGame 並不是一個很好的設計。我們可以考慮將建立 Room、Wall、Door 等工作抽象成一個方法，如下：

## 方案二 把物件的生成抽離出來變成一個方法

```

1  public class MazeGame {
2      // Create the maze.
3      public Maze createMaze() {
4          Maze maze = new Maze();
5          Room r = createRoom();
6          Wall w = createWall();
7          Door d = createDoor();
8          w.setDorr(d);
9          r.addWall(w, NORTH);
10         ...
11     }
12     public Room createRoom() {
13         return new Room();
14     }
15     ...
16 }
```

當有新的 MazeGame 時，只要建立新的子類別並讓它覆蓋 createRoom 等方法即可，不需修改原程式碼，如下：

```

1  public class EnchantedMazeGame extends MazeGame {
2      public Room createRoom() {
3          return new EnchantedRoom();
4      }
5      public Door createDoor() {
6          return new EnchantedDoor();
7      }
8      public Wall createWall() {
9          return new EnchantedWall();
10     }
11 }
```

其架構圖如圖 3.1 所示。

如此一來，在不修改 MazeGame 的原則下，我們可以透過新增一個 EnchantedMazeGame 的類別來滿足新功能。EnchantedMazeGame 雖然只單純的覆蓋方法 createRoom()，傳回一個 EnchantedRoom 的物件，卻可以在不改變 MazeGame 的條件下滿足需求。這樣的設計原則在物件導向的設計樣式中四處可見，這個例子我們在工廠方法一章中也會仔細的說明。雖然新的架構較為複雜，但符合了 OCP 的原則，以後擴充性會比較佳。

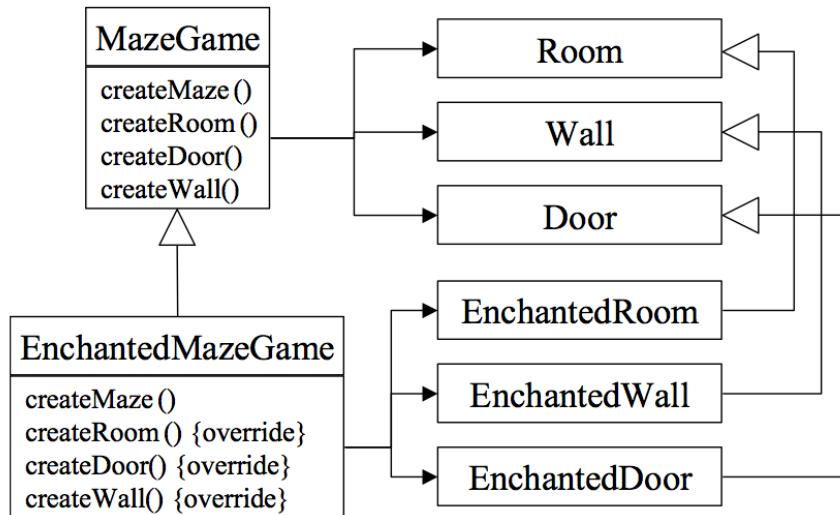


圖 3.1: Maze Game: OCP

## 3.6 防火牆：防護變異原則

針對不穩定或者是容易變動的部分進行防護，不能讓它對其他元件有非預期的影響。

保護變異（Protected Variation; PV）是一個重要且基本的軟體設計原則。設計一個物件、子系統、或系統時，針對其不穩定或者是容易變動的部分進行防護，不能讓它對其他元件有非預期的影響。亦即，對於一個元件的設計，往後若有需求變動，其增加或修改的設計，不能對其他元件有不良的影響。Larman 定義變動有兩種：<sup>2</sup>

- 變異點 (variation point)：在已存在或現存的系統或需求中的變動。
- 改善點 (evolution point)：將來可能產生的變動，但非現存的需求中要變動。

原先的設計是百分比的折扣策略，之後可能增加其他未知種類的折扣策略之設計，但此設計不能影響到其他已經設計的價格策略元件，這種變動屬於一種「變異點」的變動。

<sup>2</sup>Larman, Craig. Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. Pearson Education India, 2005.

## 3.7 練習

### 選擇/簡答

**Ex 1:** 為何使用「介面」可以降低耦合力？

**Ex 2:** 一個 Initialization 的模組中，會設定遊戲一開始需要執行的所有功能，他屬於哪一種內聚力

**Ex 3:** 大學裡的成績系統，如果是遇到「國文」科目的輸入時，會出現「活動加分」的項目，該分數是由學務處輸入的成績。你覺得這樣的設計是否具備高內聚低耦合？

**Ex 4:** 在一個線上考試系統中，可能的變異點是什麼？如何預防？

**Ex 5:** 在一個象棋系統中，可能的改善點是什麼？如何預防？

**Ex 6:** 說明 StringTokenizer 是如何運作的？我們可以猜測他內部有一個變數在記錄目前位置，每次執行 nextToken() 時他就會 +1，但為什麼我們不能看這個值？也不能修改這個值？其設計的原理為何？

### 設計

**Ex 7:** (OCP) 變動的計價策略，某物品的價格會隨著計價策略的改變而變動，例如假期特價 (HolidaySale) 或特別特價 (SpecialSale)。為了符合 OCP 原則，我們應該把計價抽離出來設計。(1) 請畫出 UML 設計圖。(2) 請寫出程式碼。以下為 Company 的主程式。

```

1   class Company {
2       public double totalPrice(Part[] parts) {
3           double total = 0.0;
4           for (int i=0; i<parts.length; i++) {
5               total += parts[i].getPrice();
6           }
7           return total;
8       }
9   }
```

Hint: 建立一個抽象的計價類別 PricePolicy；並且讓 Part 類別放一個這個類別的參考。

**Ex 8:** 我們寫一個 Date 的元件（模組），他應該有什麼功能？討論一下你的設計的內聚力好嗎？



# Chapter 4

## 心法二：物件導向設計原則



本章介紹物件導向特有的設計原則。

## 4.1 異中求同：一般化原則

Inheritance: Method of reuse in which new functionality is obtained by extending the implementation of an existing object. The generalization class (the superclass) explicitly captures the common attributes and methods. The specialization class (the subclass) extends the implementation with additional attributes and methods.

Advantages of Generalization/Inheritance

- New implementation is easy, since most of it is inherited
- Easy to modify or extend the implementation being reused

## 4.2 委以重任：善用包含/委託

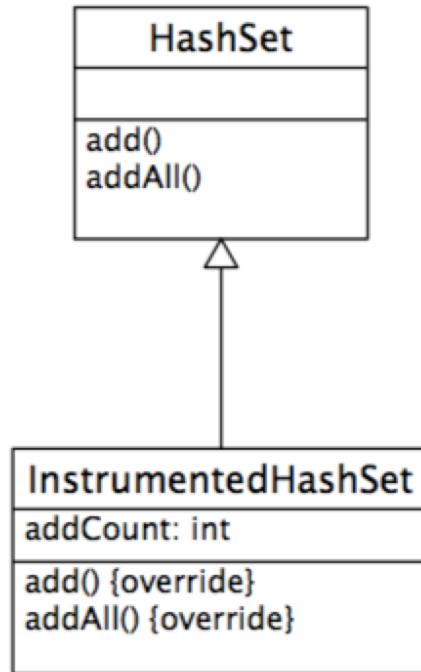
善用包含/委託的關係，它可以間接的實踐繼承，而且更有彈性。

### 繼承的缺點

- “White-box” reuse, since internal details of superclasses are often visible to subclasses
- Breaks encapsulation
- Static binding: Implementations inherited from superclasses can not be changed at runtime

### 4.2.1 破壞封裝性

```
1  public static class MyHashSet<E> extends HashSet<E> {
2      public int addCount = 0;
3      @Override
4      public boolean add(E a) {
5          addCount += 1;
6          return super.add(a);
7      };
8      @Override
```



```

9     public boolean addAll(Collection<? extends E> c) {
10    addCount += c.size();
11    return super.addAll(c);
12  }
13}
14}
  
```

以下是一個測試案例：set.addAll() 加上一個集合物件，裡面有三個元素。

```

1  public class BrokenEncapsulationTest {
2      @Test
3      public void testAddCount() {
4          MyHashSet<String> set =
5              new MyHashSet<String>();
6          set.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
7          assertEquals(3, set.addCount); //!! 錯了
8      }
9  }
  
```

我們以為 set.addCount 的值應該是 3, 但其實是 6 !! 為什麼？原來 HashSet 的 addAll() 會呼叫 add()。如果不知道父類別的程式的內容，就有可能會發生這樣的錯誤，知道父類別的內容，這樣就破壞了封裝性。

### 4.2.2 造成程式碼重複

想像一個播放器。一開始都是繼承 play()，但後來 PortableCassettePlayer 與 MP3Player 的播放方式已經改變，變成 xyz，所以進行 override。但 MP3 又不能繼承 PortableCassettePlayer，只好程式碼重複。

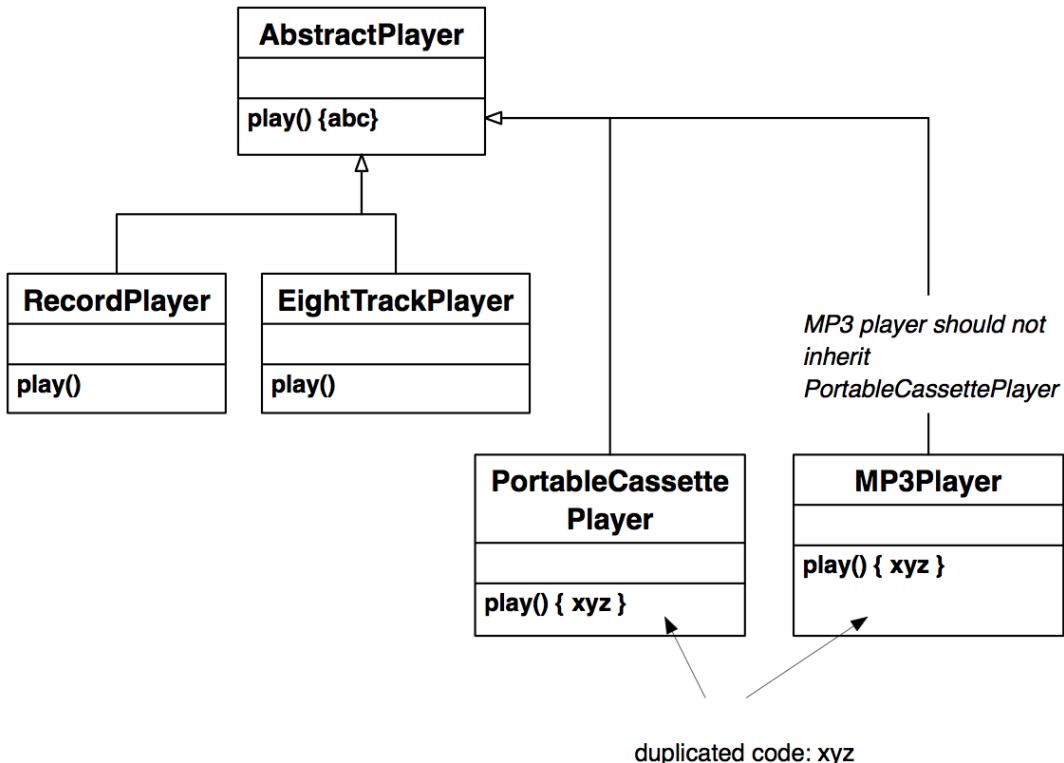


圖 4.1: 造成程式碼的重複

可以透過委託的方式來解決這個問題：

### 4.2.3 包含

- Method of reuse in which new functionality is obtained by creating an object composed of other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed

包含/委託的優點

- “Black-box” reuse, since internal details of contained objects are not visible

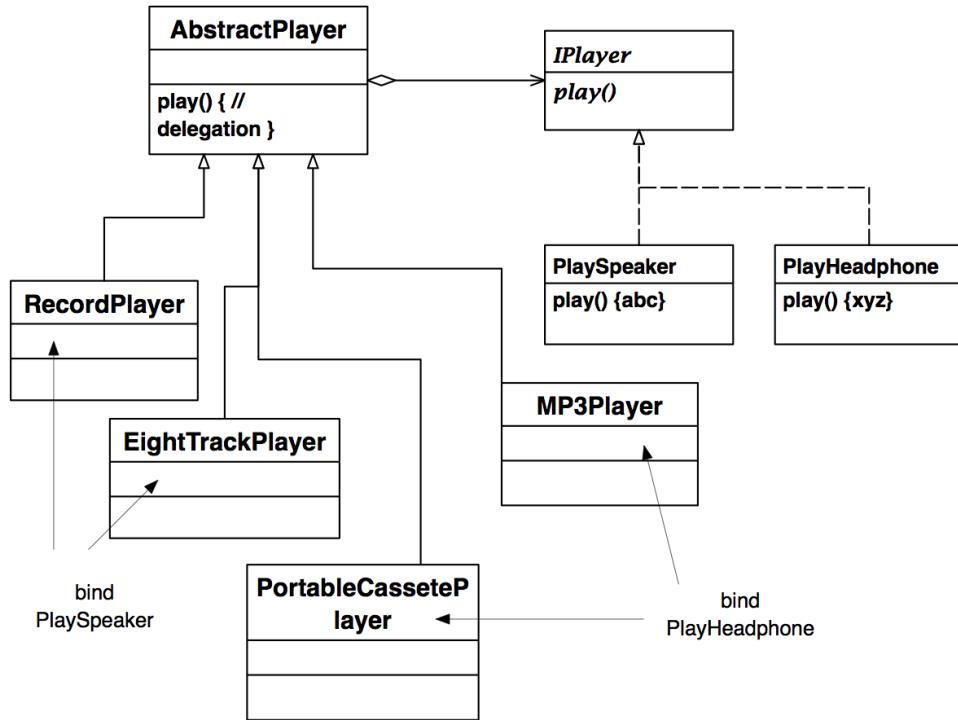


圖 4.2: 修改後：使用委託

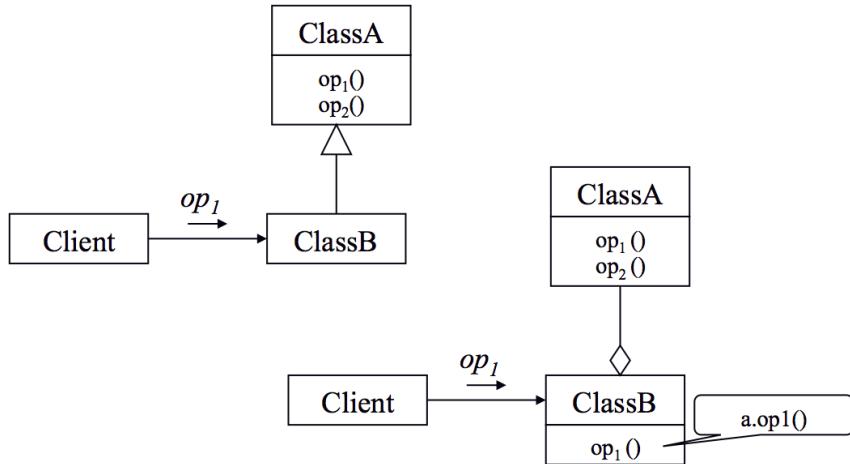


圖 4.3: Delegation

- Contained objects are accessed by the containing class solely through their interfaces
- Good encapsulation
- Dynamic binding

- The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type

缺點

- Resulting systems tend to have more objects
- Interfaces must be carefully defined in order to use many different objects as composition blocks

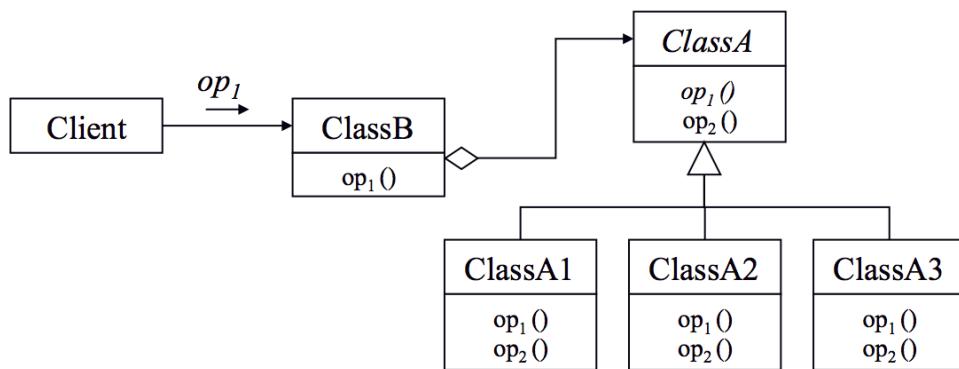


圖 4.4: 善用繼承與包含

### 4.3 空為上：善用介面

不要直接呼叫一個實作的類別。請呼叫一個介面，藉此降低模組之間的耦合力。

介面的主要目的在定義兩個物件之間溝通的規格。考慮一個 IDE 的介面是電腦主機板與 IDE 設備如硬碟的溝通橋樑。當 IDE 介面一被定義後，主機板的廠商可以依照此介面去設計他們的主機板，而不需要理會將來的硬碟是 IBM 或 Seagate 或 Quantum。相同的，硬碟廠商也可以依照 IDE 介面去設計他們的硬碟，而不需理會將來是哪一種主機板與其溝通。

軟體的設計亦是相同的道理。兩個模組之間可以先定好一個溝通的介面，而後各模組的負責人就可以依此介面分別去實作，之後在結合即可。這樣的好處除增加系統平行開發的可能之外，亦可以增加系統的彈性：模組 A 不需要明確的知道與其合作的哪一個模組（假設是模組 B），他只要知道與其合作的介面為何即可（假設為介面 I）。將來如果系統作修改或擴充，我們可以用符合介面 I 的模組 B' 來取代 B，而不需要修改任何模組 A 的程式碼。

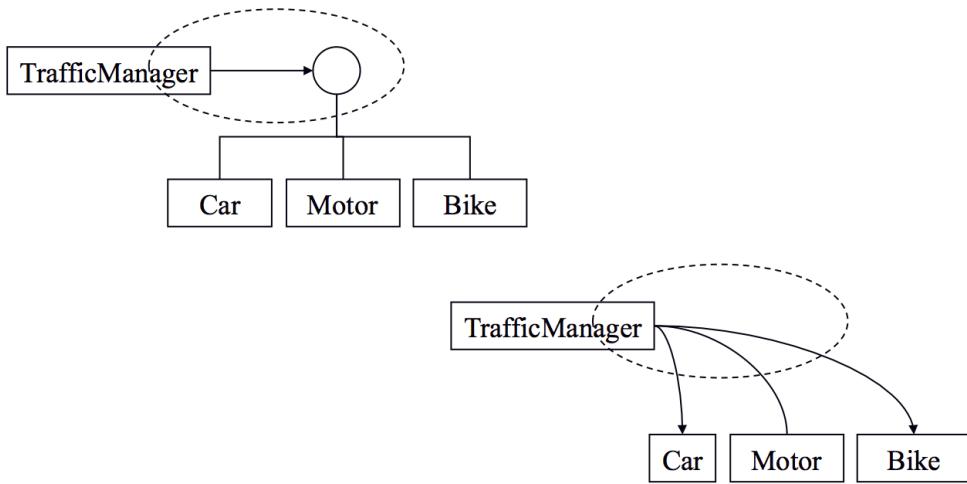


圖 4.5: Loose Coupling

介面內只定義該做的事，而沒有定義如何實作。在 Java 中，介面只定義了一群方法，卻沒有定義這些方法的實作。例如我們宣告一個交通工具的介面：

```

1  public interface IManeuverable {
2      public void left();
3      public void right();
4      public void forward();
5      public void backward();
6      public void setSpeed(double speed);
7      public double getSpeed();
8  }

```

**IManeuverable** 是一個介面，它定義一個『交通工具應有的功能』(能左右轉、前進、後退及設定速度)。此介面是交通管理系統 (**TrafficManager**) 與汽機車 (**Car**、**Motor**) 之間溝通的一個橋樑，汽機車實作此介面，且交通管系統使用此介面。

對 **TrafficManager** (介面的使用類別) 的開發者而言，不需要知道交通工具的實作，只需要知道要用的介面內有哪些方法可以呼叫即可。其部份的程式碼如下：

```

1 // TrafficManager 使用交通工具介面
2 class TrafficManager {
3     public void manage (IManeuverable c) {
4         c.setSpeed(35.0);
5         c.forward();
6         c.left();

```

```

7      }
8  }
```

在執行時，傳入的參數 `c` 可以是任何一個實作 `IManeuverable` 介面的類別的物件，例如 `Motor` 或 `Car` 的物件。這與前一節所提及的多型是完全相同的概念。亦即：若 `classA` 實作介面 `I`，則介面 `I` 的所有方法可以作用在任何 `classA` 的物件，及 `ClassA` 的子類別的物件上。

對 `Car`、`Motor`（介面的實作類別）等而言，他們必須實作 `IManeuverable` 所定義的所有方法，而不需要知道 `TrafficManager` 如何與其溝通。其部份的程式碼如下：

```

1  public class Car implements IManeuverable {
2      ...
3  }
4  public Motor implements IManeuverable {
5      ...
6  }
```

### 4.3.1 降低模組間的耦合力

為了清楚的表現介面所帶來的彈性，我們用比較的方式來說明它的好處。如果我們沒有宣告一個 `IManeuverable` 的介面，那們 `TrafficManager` 該如何與所有的交通工具溝通呢？

```

1  class TrafficManager {
2      public void manage (Car c) {
3          c.setSpeed(35.0);
4          c.forward();
5          c.left();
6      }
7      public void manage (Motor c) {
8          c.setSpeed(35.0);
9          c.forward();
10         c.left();
11     }
12     public void manage (Trunk c) {
13         c.setSpeed(35.0);
14         c.forward();
15         c.left();
16     }
17 }
```

	類別繼承	介面繼承
另名	實作繼承 (Implementation inheritance)	行為繼承 (Behavior Inheritance)
目的	子類別不需定義自己的實作，而是引用父類別的實作，亦即，再使用父類別的程式碼	子類別可以在任何地方取代父類別的工作或角色，亦即，提供一個抽象以做為多型之用
關係	是一種 (is a kind of)	支援實作 (is a kind of that support this interface)
取代	若 B 擴充 A，則 B 的物件可以取代 A 的物件	若 B 實作 A，則 B 的物件可以取代 A 的介面物件

表 4.1: 類別繼承與介面繼承

可以發現其實上述的三個方法所作的事都相同，只是對象不同而已。因為沒有宣告它們共同的介面，使我們做了很多重複的工作。不僅如此，模組與模組間的耦合力也因此增加了。

### 4.3.2 類別繼承與介面繼承

介面的作用在「將一群具有相同行為的類別抽象出來成為一個介面」，好讓其它的物件針對此抽象作處理，而非一個個的個別類別。這樣的機制所帶來的好處就是多型。我們也提到類別的繼承可以帶來多型，並討論它所可能帶來的困擾。如果類別的繼承也可以帶來多型，那我們為什麼要介面繼承？

在物件導向中，類別繼承 (class inheritance) 與介面繼承 (interface inheritance) 是特性類似卻又目的不同的兩個觀念，常常造成混淆與誤用。這是因為它們提供許多相同的架構與益處，我們用下表區別它們的差別。

在類別繼承中的父類別是有實作的，所以子類別繼承父類別的目的是為了擁有像父類別一般的功能。介面繼承的目的不是在擁有介面的能力 (事實上，介面根本沒有實作)，而在宣稱此類別將擁有介面內所宣稱的能力，但此能力必須類別自己實作，而非繼承自介面。綜合來說：類別繼承是為了再利用父類別的程式碼，介面繼承卻是為了提供一個抽象以做為多型之用。類別繼承的『附帶』好處是多型，但使用上要小心。

那們介面繼承會不會違反 LSP？因為介面內沒有任何的實作，行為上絕對不會與子類別相互矛盾或衝突。這點也給我們另一個啟示，如果我們只要一個抽象觀念而非實作上的繼承 – 用介面繼承，而不要用類別繼承。

## 4.4 代父從軍：Liskov 取代原則

引用到父類別的方法必須要能夠在不知道其子類別為何的情形下也能夠套用在子類別上。

Liskov 取代原則 (Liskov Substitution Principle) 主要探討的是子類別可否取代父類別的問題。這個問題的基本是物件的多型 (polymorphism)。什麼是多型？「相同的訊息可以送給不同的類別的物件，每一個物件會依其獨特性作出不同的反應」，或「相同的方法可以作用在不同的物件上」。例如在下例中，Circle 與 Rectangle 都是 Shape 的子類別，所以 Shape 中所定義的方法可以用在 Circle 或 Rectangle 中。假設類別 MyApp 的方法 paint 會要求一個 Shape 物件作繪圖的動作，如下：

```

1  class MyApp {
2      public void paint(Shape s) {
3          s.draw();
4      }
5      ...
6  }
```

第二行中的 paint() 的參數為 Shape，代表 MyApp 的物件可以送訊息給 Shape、Circle 或 Rectangle 的物件（因為這三個類別是 Shape 的子類別）。亦即，行 3 中的 s 物件在執行期間，可能是 Shape、Circle 或 Rectangle 的物件。方法 draw() 可以作用在多個類別 (Shape、Circle 或 Rectangle) 的物件上，故稱為多型。若從「取代」的角度來看，子類別 (Circle) 是父類別 (Shape) 功能的擴充，所以由子類別來取代父類別去執行的父類別動作 (draw()) 也沒有問題。多型的使用可以視為一種子類別的物件取代父類別物件工作的行為。

然而，繼承並非是萬無一失的，如果不小心謹慎的使用多型的技巧，可能會造成問題。以下我們以 2 個實例來說明不適當的繼承所帶來的問題。

### 正方形是一種矩形嗎？

從概念上來看，正方形 (Square) 是一種矩形 (Rectangle)，所以我們很自然的在它們之間宣告一個繼承關係。以下程式說明 Square 在繼承 Rectangle 以後所作的修改，因為 Square 的寬與高是相同的，所以 Square 必須覆蓋方法 setWidth() 與 setHeight() 用以保障長與寬都相同。

```

1  class Rectangle {
2      private int width, height;
```

```
3     public Rectangle (int w, int h) {
4         width = w;
5         height = h;
6     }
7     public setWidth(int w) {
8         width = w;
9     }
10    public setHeight(int h) {
11        height = h;
12    }
13 }
14
15 class Square extends Rectangle {
16     public Square (int s) { super (s, s); }
17     public setWidth(int w) {
18         super.setHeight(w);
19         super.setWidth(w);
20     }
21     public setHeight(int h) {
22         super.setHeight(h);
23         super.setWidth(h);
24     }
25 }
26
27 class App {
28     public void testLSP(Rectangle r) {
29         r.setWidth(4);
30         r.setHeight(5);
31         if (r.getArea() != 20)
32             System.out.println("Error");
33     }
34     public static void main(String args[]) {
35         Rectangle r = new Rectangle(3, 4);
36         testLSP(r);
37         Square s = new Square (5);
38         testLSP(s);
39     }
40 }
```

類別 App 中的方法 testLSP() 傳入 Rectangle 物件。對 paint() 而言，它所要處理的物件就是一個 Rectangle，所以在設定它的寬度與長度分別為 4 與 5 後，r 的面積就應該是  $20(4*5=20)$ 。然而事實並非如此 — 若傳進的物件是一個 Rectangle 時不會出錯，但若傳進的物件是一個 Square 時就會出錯 (面積會變成 25)。為什麼會這樣呢？依照多型的定義，用 Square 的物件來取代 Rectangle 的物件來運作應該不會有問題的呀？

Rectangle 的例子說明這個現象。子類別繼承父類別後會有擴充的屬性及功能，也就是說，物件的功能應該越多。但 Square 繼承 Rectangle 後，條件卻越來越緊 (Square 多了對屬性間的限制)，這時候用 Square 來取代 Rectangle 也會出現問題。

## Tree 是一種 Graph ?

從數學上來看，樹狀結構 (Tree) 與圖形結構 (Graph) 都是由點 (node) 與線 (edge) 所構成的結構，不同的是 Tree 要求任 2 點必須相通 (直接或間接)，而且任 2 點只有一個路徑 (也就是不可以有迴圈)。Graph 可以有迴圈的結構。從概念上及數學上來看樹狀結構 (Tree) 「是一種」 (is a kind of) 圖形結構 (Graph)，依據物件導向分析 aka 的關係，我們讓 Tree 繼承 Graph。

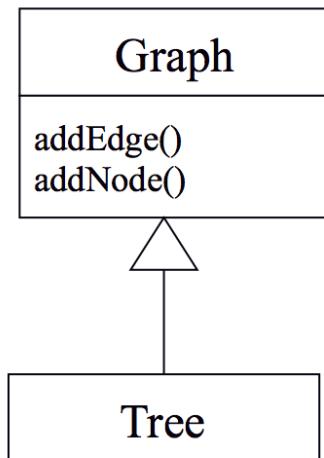


圖 4.6: Tree 是一種樹？

Graph 有 addNode() 與 addEdge() 2 個方法，分別用以新增點與線，這在 Graph 中是極為自然的。Tree 因為繼承自 Graph，也順理成章的擁有 addNode 與 addEdge() 2 種功能，然而問題就出在此處 - Tree 任意的新增與移除點或線後就不是一個 Tree 了，這 2 個方法在 Tree 中根本不自然，也不應該。怎麼會這樣呢？Tree 不是一種 Graph 嗎？為什麼繼承後會發生這種問題？

LSP 的定義如下：

引用到父類別的方法必須要能夠在不知道其子類別為何的情形下也能夠套用在子類別上。( Functions that use references to super classes must be able to use object of subclasses without knowing it! )

相對於 Rectangle 的例子：

Functions (paint) that use references (s) to super classes (Shape) must be able to use object of subclasses (Circle or Rectangle) without knowing it!

簡單的說，LSP 要求我們在建立類別階級時也必須同時考慮到他們之間的行為階級(子類別是否繼承父類別的行為)，若否，則不應建立類別階級。編譯器並不能幫我們檢查這一點，因為，只要程式語言提供多型的功能，用子類別的物件來取代父類別的物件來運作在編譯時是不會發生錯誤的。所以這個問題就必須留給設計者傷腦筋。原則是什麼？子類別的行為不能比父類別少、子類別的限制不能比父類別多、並且多用介面繼承，少用類別繼承。

## 4.5 穢纖合度：介面分割原則

介面分離原則 (Interface Segregation Principle) 採用多個分離的介面，比採用一個通用的涵蓋多個業務方法的介面要好。

在說明此原則之前，我們先以 Door 實例說明介面污染 (interface pollution)。

假設我們有一個抽象類別 Door，裡面定義了 lock() 與 unlock(), isDoorOpen() 等方法，分別表示關門，開門，是否開著等功能：

```

1      abstract class Door {
2          public abstract void Lock();
3          public abstract void Unlock();
4          public abstract boolean IsDoorOpen();
5      }

```

TimedDoor 是一個 Door 的子類別，當門開太久時，他就會警告。Timer 是一個專門來做警告的類別，裡面有一個 register() 的方法，

```

1  class Timer {
2      public void register(int timeout, TimerClient client);
3  }
4  class TimerClient {
5      public abstract void timeout (int timeOutId);
6  }

```

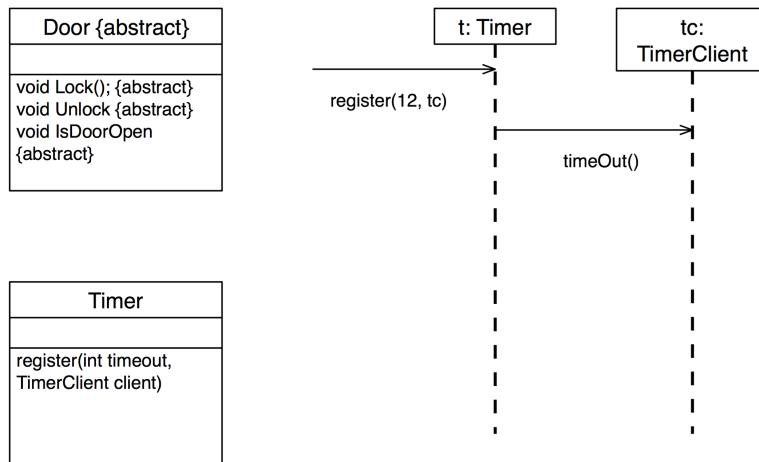


圖 4.7: Timer1

當 `timeout` 的時間到了，他就會通知 `client`。如果有一個物件想要被通知，他就可以呼叫 `register` 來做一個註冊，以便在 `timeout` 來時被通知。

因為 `TimedDoor` 需要被通知到，所以我們想把它設計成 `TimerClient` 的子類別，但 `TimedDoor` 已經是 `Door` 的子類別了，在 Java 中我們無法做多重繼承。

因此，可能有些設計者會讓 `Door` 繼承 `TimerClient`，如此一來，`TimedDoor` 自然地成為 `TimerClient` 的子類別，編譯器與執行上都沒有錯誤，但這樣的設計造成「`Door` 相依於 `TimerClient`」，這樣的相依性是沒有道理的，也造成了介面污染。假設 `TimerClient` 內有 `timeout()` 的抽象方法，`Door` 也需要時做此方法。

```

1  class Door extends TimerClient { //錯誤的設計！
2      public abstract void TimeOut (int timeOutId) {
3          }
4      }

```

一開始的設計者可能知道此方法僅是為了「讓編譯器通過」的權宜設計，但到了維護期，如果 `timeout()` 的介面有所更動而需要 `Door` 重新編寫時，維護者就不一定知道其意義

為何，而造成困擾。簡言之，目前的設計錯誤為：

TimedDoor 為了要成為一種 TimerClient，但本身又不能繼承 TimerClient，只好讓 Door 去繼承 TimerClient。

其實這個問題我們可以有兩個解決方法

- 把 TimerClient 設計成介面
- TimerDoor 透過委託的方式來呼叫 TimerClient

回到介面分離原則。當我們設計一個介面時，內部所包含的方法必須謹慎的考量，避免設計一個通用，包含很多方法的大介面。採用通用大介面通常會造成介面污染。當一個類別為了滿足其中一個方法而必須實作該介面時，就同時被迫實作其他的方法，而造成介面污染。

簡單的說，介面分離原則建議：*擁有許多方法的介面應該被分離不同的介面，每一個介面擁有一群緊密相關的方法，被一些特定的客端物件使用。*

*Interface pollution must be broken up into groups of methods, every group serves a different set of client.*

## 4.6 所依皆幻：相依反轉原則

高階模組不該相依於一個低階模組。兩者都應該相依於抽象。

**Dependency Inversion Principle (DIP):** *High level modules should not depend upon low level modules. All should depend on abstraction.*

1970 年代的軟體開發多遵從結構化分析，倡導由上而下的分解 (top down decomposition)，這樣的方法論鼓勵一個高階的模組相依於一個低階的模組。然而，高階模組包含著一個應用程式重要的策略決定與企業邏輯，如果它相依於一個低階的模組，那就代表策略的變更會因為低階的執行的不同而變更，這是不是適當的。

舉個例子來說，版本管理是重要必要的政策，但是因為工程師習慣用檔案管理系統或是 facebook 檔案分享來共享資料，因此「本公司採用 facebook 進行版本管理」，而放棄一個版本管理應有的功能與機制。這就是高階相依於低階。

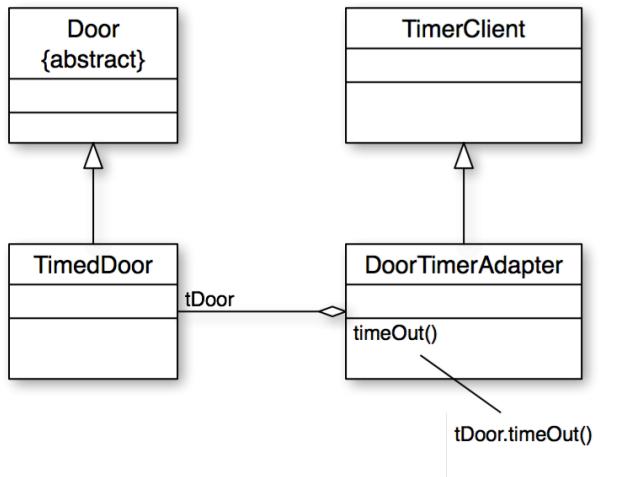


圖 4.8: Timer2

**Copy Program** 我們可能會寫出以下的 copy() 模組

```

1 void Copy() {
2     int c;
3     while ((c = ReadKeyboard()) != EOF)
4         WritePrinter(c);
5 }
```

這個程式看起來沒有什麼問題，但 copy 這個高階的動作就相依於 ReadKeyboard() 這個低階的動作了。例如今天我們要輸出的對象是一個 printer，程式就要做一些修改，如下：

```

1 void Copy(outputDevice dev) {
2     int c;
3     while ((c = ReadKeyboard()) != EOF)
4         if (dev == printer)
5             WritePrinter(c);
6         else
7             WriteDisk(c);
8 }
```

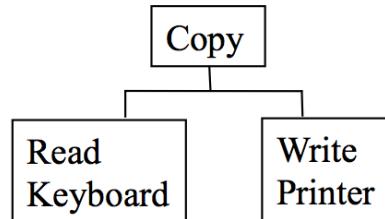
*High level module**Low level module*

圖 4.9: 高階模組呼叫（相依於）低階模組

一般來說，所謂的 copy 就是把資料從「來源」複製一份到「目的地」，這樣的政策是不會變的，不該因為目的地是 disk 或 printer 而有所改變。下方是一個符合 DIP 原則的程式：

```

1  abstract class Reader {
2      public abstract int Read();
3  }
4  abstract class Writer {
5      public abstract void Write(char);
6  }
7 //copy 相依於抽象的 Reader, Writer
8 void copy(Reader r, Writer w) {
9     int c;
10    while ((c=r.Read()) != EOF)
11        w.Write(c);
12 }
  
```

在這個程式中，copy 所牽涉到的來源與目的地分別是抽象的 Reader 和 Writer，就不會相依於 Printer, Scanner, Disk 等低階的物件了。若我們今天想要寫道 Disk，只要讓 Disk 去實作 Writer 再透過 dynamic binding 就可以達到上述的效果。

**Lamp Program** 接下來我們來看一個檯燈開關的例子。

```

1  class Lamp {
2      public void TurnOn() { ... }
3      public void TurnOff() { ... }
4  }
5
  
```

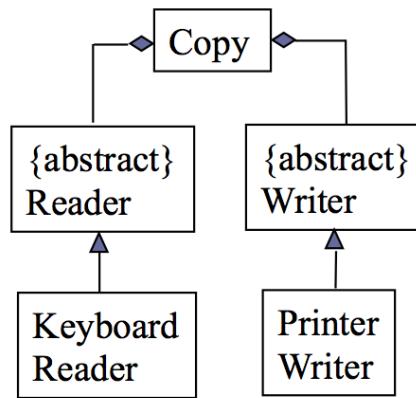


圖 4.10: 相依於抽象

```

6   class Button {
7       private Lamp lamp;
8       public Button(Lamp p) {lamp =p; }
9       public void Detect() {
10          boolean buttonOn = GetPhysicalState();
11          if (buttonOn)    lamp.TurnOn();
12          else           lamp.TurnOff();
13      }
14  }
  
```

這樣的程式不好！Button 相依於 Lamp, 也就是說它只能處理 Lamp 了。想想看我們從五金行買回來的開關面板可是沒有限定只能用來處理檯燈阿！以下的程式則相依於一個抽象。

```

1  abstract class ButtonClient {
2      public abstract void TurnOn();
3      public abstract void TurnOff();
4  }
5 // Button 相依於一個抽象通用的 ButtonClient
6  abstract class Button {
7      private ButtonClient bClient;
8      public Button(ButtonClient b) { //Bind 實際的 ButtonClient
9          bClient = b;
10     }
11     public void Detect() {
12         boolean buttonOn = GetState();
  
```

```

13             if (buttonOn)    bClient.TurnOn();
14             else      bClient.TurnOff();
15         }
16     public bool GetState();
17 }
18
19 // Lamp 自己定義開關
20 class Lamp extends ButtonClient {
21     public void TurnOn() { ... }
22     public void TurnOff() { ... }
23 }
24
25 class SquareButton extends Button {
26     ButtonClient client;
27     public SquareButton(ButtonClient b) {
28         client = b;
29     }
30     public bool GetState() { ... }
31 }
```

高階模組不應該依賴低階模組，兩者必須依賴抽象(即抽象層)，抽象不能依賴細節，但細節必須依賴抽象，抽象模組不應該根據低階模組來創造，這就是「依賴反轉原則」的概念。

### 4.6.1 相依注入

將相依性移除於模組之中，透過相依性注入 (dependency injection) 的方式來建立相依性。

#### Creation injection

考慮以下的程式 Customer 的物件相依於低階的 SQLServer 物件。

```

1 Customer obj = new Customer();
2 obj.add();
3 ..
4 class Customer {
5     private SQLServer db = new SQLServer();
```

```

6     public boolean validate() { ... }
7     public void add() {
8         if (validate()) db.add();
9     }
10    }

```

這是不好的，應該抽離出來。如下的程式讓 Customer 相依於一個抽象的 DBI 介面，但是何時建立 Customer 與 Oracle 的連接呢？在建立 Customer 時，我們決定採用 Oracle 這個物件，因此是在「物件建立時」決定這個相依性，稱之為 Creation Injection。

```

1 Customer obj = new Customer(new Oracle());
2 obj.add();
3 ...
4 class Customer {
5     private DBI db;
6     public Customer(DBI dbi) {
7         db = dbi;
8     }
9     public boolean validate() { ... }
10    public void add() {
11        if (validate()) db.add();
12    }
13 }

```

同理，在上一個 Button 的例子中，也是透過 creation injection 來決定相依性。

**Factory pattern** 有時候該相依物件的決定並不是單純的建構子所決定的，而是有另一個物件生成的

```

1 DBFactory dbf = new DBFactory(...);
2 Customer obj = new Customer(dbf);
3 ...
4 class Customer {
5     private DBI db;
6     public Customer(DBFactory dbf) {
7         db = dbFactory.createDB();
8     }
9     ...
10 }

```

我們在後續的設計樣式 Abstract Factory 就會看到這樣的手法。

### Setter injection

有時候在物件一開始生成之時上無法決定其相依物件，而是透過 `setter` 方法來設定其相依性，例如：

```

1 Customer obj = new Customer();
2 obj.setDB(new Oracle());
3 ...
4 class Customer {
5     private DBI db;
6     ...
7     public void setDB(DBI db) { //setter injection
8         this.db = db;
9     }
10 }
```

## 4.7 身不由己：控制反轉原則

應用程式（或客製化模組）的控制流程而來自於一個一般性、可重用的框架，而非模組本身。

*Inversion of Control (IoC): Custom-written portions of a computer program receive the flow of control from a generic, reusable framework.*

傳統的程式我們控制整個程式的流程，例如

```

1 System.out.println("input your name");
2 String name = scanner.nextLine();
3 name = processName(name);
4 System.out.println("input your name");
5 String address = scanner.nextLine();
6 address = processAddress(address);
7 show(name, address);
```

程式會照你所想的一步步的執行：你先輸入你的名字、然後住址，接著呈現出你的名字與住址。

但現今的 window 程式設計，其流程就並非如此，你（programmer）並不會知道使用者會先輸入 name 或是 address，你只要關心使用者按下確定的 button 時應該處理什麼就好了。這是因為流程由上一層的 Window 框架所決定，而非你的程式。

假設我們設計一個書籍閱讀的框架，所有的書籍其閱讀流程都是：openBook, readline, 然後 closeBook，我們把這個流程寫在抽象的 Book 「框架」中。

```

1  abstract class Book {
2      public void readBook() {
3          openBook();
4          readLine();
5          closeBook();
6      }
7      abstract void openBook();
8      abstract void readLine();
9      abstract void closeBook();
10 }
```

一本小說（Novel）的「閱讀流程」並不是由小說決定的，而是由上層的 Book 所決定的，小說只是去決定其 openBook, readLine 的方法：

```

1  class Novel extends Book {
2      void openBook() {...}
3      void readLine() {...}
4      void closeBook() {...}
5 }
```

對於小說、漫畫、論文而言，Book 就像是一個框架一般，固定（決定了）一些流程。我們在日後會說明 Template 設計樣式就是一種 IoC 的應用。

採用 IoC 的策略可以幫助我們建立一個可重複使用的程式碼框架，讓我們少寫很多的程式碼，並且確保程式一定的品質，在現今複雜應用程式中是不可或缺的。各位常常聽到的許多 Web framework, AP framework 都是應用了繼承與 IoC 的設計原則。

**好萊塢原則** 有時候 IoC 也會和好萊塢原則一起談，其意義為：「不要叫我們，我們有需要將會叫你」(Don't call us, we'll call you)<sup>1</sup>。你的應用程式不需要關注上層的流程是怎麼做的，寫好你的小程式（模組功能），該到你的流程就會叫你，不需囉嗦煩惱。

---

<sup>1</sup>許多人到好萊塢逐夢，經紀公司會叫你留下名片，叫你不要再打電話了，必要時它會打電話給你。

## 4.8 練習

### 委託

**Ex 1:** 請將下述程式從繼承改成委託的方式來撰寫

```

1   class B {
2       public void m1() {
3           ...
4       }
5   }
6   class A extends B {
7 }
```

### 善用介面

**Ex 2:** 只要是交通工具 Vehicle 就必須要能向左轉 (turnLeft) 向右轉 (turnRight) 停止 (stop) 或前進 (forward)。但如何實作 (implementation) 都必須由 Bike 或 Car 來決定。該怎麼設計類別結構？

**Ex 3:** 同上，VehicleManager 控制所有的交通工具: 紅燈停，綠燈行。

```

1   void control (Car[] cc) {
2       for (Car c: cc) {
3           if (isRed ()) {
4               c.stopCar ();
5           }
6           else (isGreen ()) {
7               c.driveCar ();
8           }
9       }
10    }
11
12   void controlBike (Bike[] bb) {
13       for (Bike b: bb) {
14           if (isRed ()) {
15               b.stopBiking ();
16           }
17           else (isGreen ()) {
18               b.startBiking ();
19           }
20    }
21 }
```

```

20      }
21  }
```

上面的程式不夠通用，可否用 interface 改寫成更通用些？

**Ex 4:** 同上，假設所有 Vehicle 的 trunRight() 實作方法都一樣，Vehicle 應該宣告為 abstract class 還是 interface

**Ex 5:** 應用 interface Comparable, 來設計一個「通用型的 getMax」，它可以找出任何陣列內最大的元素，例如可以找到最 max 的 People。

```

1  interface Comparable {
2      public int compareTo(Object other);
3  }
4
5  class GeneralMax {
6      public static ? getMax(?[] data) {
7          ?
8      }
9  }
10
11 class Main {
12     ?
13 }
14
15 class People ? {
16     ...
17 }
```

## DIP

**Ex 6:** 應用 DIP 的原則設計一個通用型的開關器 (ButtonPanel)，它可以用來開關所有電器 – 只要它有 on, off 的介面。注意 ButtonPanel 不能「看到很多」不同型態的電器，這樣偶合力會很高，也違反的 DIP 的原則。這個例子可以作為一個「GUI 元件」設計的練習。如果這個例子會作了，你大概可以設計軟體元件了。參考以下設計：

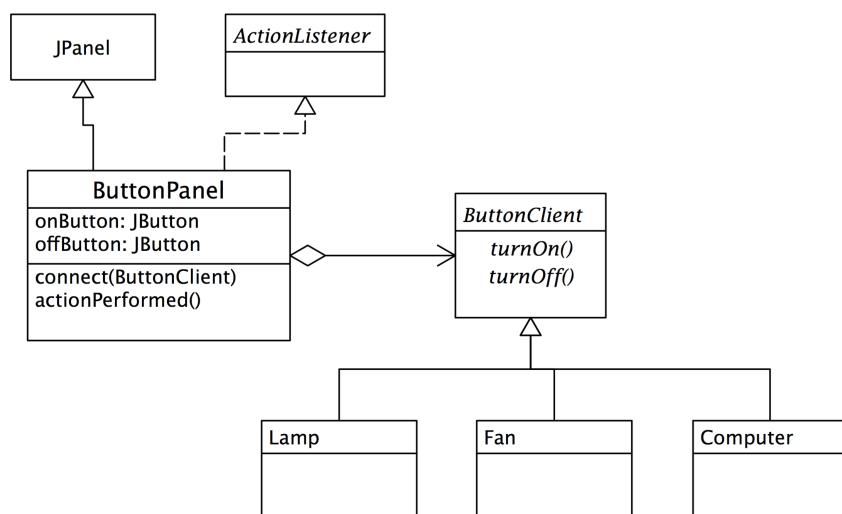


圖 4.11: DIP

**Ex 7:** (Homework) 請參考 DIP 原則 (Dependency Inversion Principle) 設計一個通用型的遙控器 RemoteController, 可以對電視 (TV) 或冷氣 (AirConditioner) 做開、關、上、下 (on, off, up, down) 等動作。TV 預設的頻道是第七台，上下會在 1-15 間變化。冷氣預設 25 度，會在 20-30 度間變化。使用 Swing 來呈現介面可以加分。(Hint: IRemoteControllable)

# Chapter 5

## 把脈清毒：程式碼重整



## 5.1 把脈：21 個程式臭味

以下是 Martin Fowler [?] 所整理的一些軟體臭味：

1. **重複的程式碼 Duplicated Code**。重複的程式碼應該抽出來集中在一個類別（或元件）中，需要的時候再去繼承或呼叫。重複的程式碼的維護性很低：我們常常改了這個忘了改另一個。
2. **冗長的方法 Long Method**。一個 1000 行的程式碼好不好看得懂？好不好維護？一個方法通常代表一個處理的流程或演算法，大到某個程度後我們就必須要學會「抽象」 - 把部分的流程抽象到另一個方法裡。
3. **大類別 Large Class**。同理，一個大類別包含很多的屬性、方法讓程式員難以理解類別的責任，應該把責任做分割，交給另一個類別來做。
4. **太長的參數列 Long Parameter List**。參數太多太長時（想像一下，10 個參數），參數代表一個方法可以調整的靈活度，有其複雜度，太多的參數令人難以理解該方法的彈性。一樣，我們應該把部分的參數彙整起來，抽象成另一個物件類別。
5. **發散變更 Divergent Change**：一個類別有太多的變更原因。一個類別最好能有一個專一的功能，也就是內聚力要高，方法之間是有關係的。這也表示不要炒大鍋菜一樣把不相關的功能都擠在一個類別內。
6. **散彈槍手術 Shotgun Surgery**：一個變更，要改很多其他模組。想像一下一個「班級人數」的值散落在 10 個模組裡，當這個人數值需要改變時，你需要翻箱倒櫃的修改這十個模組。如果這個值是放在一個設定檔中，模組啟動時去讀這個檔就好了，那麼你的修改就會少很多。
7. **依戀情結 Feature Envy**：一個計算要取好多其他類別的值。一個內聚力強的類別其計算會用到該類別的屬性，例如一個 Circle 的物件，有一個 area() 的方法來計算他的面積會用到他內部的屬性 radius，少許的引用其他類別的資料來運算當然也很正常，但如果過份的依賴大量其他的類別，那麼你的抽象化可能就出現了問題。是不是這個計算應該移到別的類別中？
8. **資料泥團 Data Clumps**：常常會一起出現的資料卻沒有抽象呈一個類別型態。例如說住址，有一群的城市、街道、ZIP code、號等字串出現，為什麼不把它抽象成 Address 的類別呢？
9. **基本型別偏執 Primitive Obsession**：不用小類別，堅持用基本型別。例如「錢」的這個變數，你可能會用 int money 來代表，又發現需要描述是哪個國家的幣值，所以又加了一個變數 String country。這會讓程式碼變得凌亂難懂，其實我們可以宣告一個小類別 Money，裡面包含這兩個屬性。
10. **Switch 敘述句**：太多的 switch case, 不會用多型。Switch 有一個缺點，當你要新增一個 case 選項時，你需要修改程式碼。如果我們本來就預期會有 case 的改變，那麼就

用多型吧，至少可以在修改程式的情況下，透過擴充來完成彈性的設計。

11. 平行繼承體系 **Parallel Inheritance Hierarchies**：可以說是散彈槍手術的一種：當你為某一個類別建立一個子類別時，你也需要為另一個類別建立子類別。假設你有兩個繼承樹，分別為 SalariedEmployee 與 HourlyEmployee，其下分別有 Engineer, Analyzer, Manager 等類別，當我們要新增一個 SoftwareEngineer 類別時，必須在這兩個繼承樹分別新增，這就是一種不好的設計。
12. 兀員類別 **Lazy Class**：沒什麼功能的類別。
13. 不實用的一般性 **Speculative Generality**: 預留的太多不實用的擴充點。為了滿足「擴充性上的彈性」，我們常常會把程式寫的複雜些<sup>1</sup>，但是仔細想一想，是不是這些擴充真的需要？還是為了設計而設計。
14. 暫時欄位 **Temporary Field**: 有一些暫時欄位只對某些特殊的方法或特殊時機才會用到，令人難以了解他的意義。例如為了不想傳太多參數就把某些值以 instance variable 的方式寫在類別內。
15. 過度的訊息串 **Message Chains** : a.getB().getC().getD() ... 當訊息串過長時可能是功能封裝的有問題，可能要考慮直接的訊息傳遞。
16. 過度的中間人 **Middle Man**：太多的方法都是透過委託來進行，該類別的方法封裝要考慮改寫。
17. 狹隘關係 **Inappropriate Intimacy**：類別間的過於親密的屬性存取，可以考慮分開來以 delegation 的方式來進行。例如不適當的宣告子類別，讓子類別任意存取父類別的資料。
18. Alternative Classes with Different Interfaces
19. Incomplete Library Class
20. Data Class
21. Comments

## 5.2 清毒：程式碼重整

### 5.2.1 方法組裝

1. Extract Method: 把方法萃取出來，形成另一個方法
2. Inline Method
3. Inline Temp
4. Replace Temp with Query

---

<sup>1</sup>可以看看 design pattern 的書

5. Introduce Explaining Variables：加上有意義的變數。特別是在一群條件判斷式中。
6. Split Temporary Variable
7. Remove Assignments to Parameters
8. Replace Method with Method Object
9. Substitute Algorithm：把演算法萃取出來。

### 5.2.2 特性移動

1. Move Method：移動方法
2. Move Field：移動屬性
3. Extract Class：萃取類別
4. Inline Class：
5. Hide Delegate：隱含委託
6. Remove Middle Man：移除中間人
7. Introduce Foreign Method：
8. Introduce Local Extension

### 5.2.3 資料組織

1. Self Encapsulate Field
2. Replace Data Value with Object
3. Change Value to Reference
4. Change Reference to Value
5. Replace Array with Object
6. Duplicate Observed Data
7. Change Unidirectional Association to Bidirectional
8. Change Bidirectional Association to Unidirectional
9. Replace Magic Number with Symbolic Constant
10. Encapsulate Field
11. Encapsulate Collection
12. Replace Record with Data Class
13. Replace Type Code with Class
14. Replace Type Code with Subclasses

15. Replace Type Code with State/Strategy

16. Replace Subclass with Fields

#### 5.2.4 條件簡化

1. Decompose Conditional
2. Consolidate Conditional Expression
3. Consolidate Duplicate Conditional Fragments
4. Remove Control Flag
5. Replace Nested Conditional with Guard Clauses
6. Replace Conditional with Polymorphism
7. Introduce Null Object
8. Introduce Assertion

#### 5.2.5 呼叫簡化

1. Rename Method
2. Add Parameter
3. Remove Parameter
4. Separate Query from Modifier
5. Parameterize Method
6. Replace Parameter with Explicit Methods
7. Preserve Whole Object
8. Replace Parameter with Method
9. Introduce Parameter Object
10. Remove Setting Method
11. Hide Method
12. Replace Constructor with Factory Method
13. Encapsulate Downcast
14. Replace Error Code with Exception
15. Replace Exception with Test

### 5.2.6 一般化處理

1. Pull Up Field
2. Pull Up Method
3. Pull Up Constructor Body
4. Push Down Method
5. Push Down Field
6. Extract Subclass
7. Extract Superclass
8. Extract Interface
9. Collapse Hierarchy
10. Form Template Method
11. Replace Inheritance with Delegation
12. Replace Delegation with Inheritance

### 5.2.7 大重整

1. Tease Apart Inheritance
2. Split an inheritance hierarchy that is doing two jobs at once
3. Convert Procedural Design to Objects
4. Separate Domain from Presentation
5. GUI classes that contain domain logic
6. Extract Hierarchy
7. Create a hierarchy of classes from a single class where the single class contains many conditional statements

## 5.3 練習

**Ex 1:** 說明何謂軟體重整合

# Chapter 6

## 秘笈：設計樣式



## 6.1 簡介

軟體的複雜度一直是軟體工程極力想解決的問題。在傳統的程序導向系統中，軟體的複雜度是透過程序抽象 (procedure abstraction) 的方式來解決的，也就是說，將系統作功能性的切割，分成較小、容易處理的單位；然而，程序性抽象的重用性 (reusability) 較差，較難以設計出高彈性 (flexible) 的系統。物件導向的設計概念以資料抽象 (data abstraction) 的方式來降低軟體的複雜度，可以開發出較有彈性、重用性較高的軟體系統。

開發物件導向的程式並非易事，要開發出高重用性的軟體更是困難。因此，Gamma 等人提出以設計樣式 (design pattern) 的方法來輔助物件導向軟體的開發 [1]。設計樣式是將過去有經驗的程式設計者的設計經驗抽鍊、整理、包裝成特定的框架，用以解決系統設計上的特定問題，例如擴展性問題、維護性問題與重用性問題等。設計樣式的優點可綜合如下：

- 充分發揮物件導向的特性。物件導向為程式語言帶來極大的好處，可是由於其技術瓶頸並不低，許多人仍用物件導向的程式語言寫傳統的程序化方法。設計樣式大量的使用繼承、委託、多型、介面、抽象類別的概念，讓程式設計師可以透過設計樣式更了解物件的技術。
- 發揮知識管理的精神。設計樣式是一種程式設計師的知識管理，他們將特別的設計問題用一種較為抽象、可套用的方式呈現，以便後者可以引用、解決相類似的問題。將程式設計經驗抽離成設計樣式的過程就是一種將知識由內潛轉為外顯的動作。
- 為高品質的軟體提供一個機會。軟體品質從 1960 年提出軟體危機後受到極度的重視，不斷的研究投入到這個領域。設計樣式所提倡的概念即是高品質的軟體，它重視非功能需求，例如高維護性、高重用性、高模組化、高擴充性、高移植性等，並且為這些品質要求提供解決方案，讓程式設計師有依循的管道。
- 設計樣式起源於 Christopher Alexander，他發現建築時所遭遇的問題常常是不斷重複發生的，於是將建築時候常遇見的這幾種問題分別搭配適合的解決方式，建立出一個良好的問題解決範本。軟體工程便將這樣的觀念應用在軟體設計的問題上，用以解決經常發生的困難問題上。設計樣式在軟體工程界真正的流行是在 Gamma 等四人 (有人稱之為四人幫 (Gang of Four ; GoF)) 合著的 *Design Patterns* 一書發表後 [1]。書中將設計軟體時候常見的幾種問題抽離成 23 種設計樣式，實際的說明如何應用這些樣式解決物件設計上的問題，或是應用這些樣式建立高品質的軟體。因為這本書，物件的技術得以更精緻化，樣式的觀念也隨之風行。也由於這 23 個樣式受到普遍的採用，許多人便將這些樣式冠稱為 GoF 設計樣式。

**設計樣式的結構** 樣式可簡易定義如下：A solution to a problem in a context (在某個情境下針對特定問題所提出的解決方案)。一般而言，每個設計樣式都會透過一個結構性的描述

來描述它的使用情境、欲解決的問題及解決的方法。每個樣式包含下列四項元素：

- 樣式名稱：用簡單易懂的命名來定義一個樣式，方便之後的溝通和使用。
- 問題描述：紀錄這個樣式所解決的問題和所面對的問題背景環境。這個部份將會描述樣式的情境與問題。
- 解決方式：詳細描述這個樣式內所使用到的技巧和元素，包含技術方法、元素結構及元素間的互動等。這個部份將會描述樣式的解決。
- 影響：使用這個樣式後可能可以推論的相關問題解決，使用後的好處壞處等，提供後來使用者可以有一個作出評估選擇的了解。

GoF 設計樣式則擴充這四項，提供以下的項目以詳加說明樣式的使用：

- 目的 (Intent)：該設計樣式所預期達到之目標。
- 動機 (Motivation)：該設計樣式的背景資訊、使用動機。
- 應用時機 (Applicability)：該設計樣式的適用時機、環境與限制。物件結構 (Structure)：該設計樣式之組成物件及他們之間的關係。通常採用 UML 中的類別圖來描述。
- 參與者 (Participants)：物件結構中每一物件所擔負之責任說明。合作方式 (Collaborations)：物件間相互合作之方法。通常採用 UML 中的互動圖來描述。
- 範例程式 (Sample code)：簡單卻充足的範例來呈現該設計樣式之實作。通常使用 C++ 或 Java。
- 結論 (Consequence)：該設計樣式的有缺點。
- 引用範例和過程 (Known uses)：別名。
- 相關樣式 (Related Patterns)：相關的樣式。

設計樣式的組成結構相當的符合知識管理架構。在 know what 方面，你必須先了解設計樣式可以解決什麼問題；你可以先翻看每一個樣式的使用目的、它的背景動機、它的使用時機等來分析有沒有樣式可以解決你的問題。在 know what 之後便是 know how – 了解如何引用這個設計樣式來解決你的問題。設計樣式中所提供的物件結構、相關參與者、他們的合作方式可以作為你設計的參考，透過這三者將系統的設計圖描繪出一個大致上的形狀。如果你實在不知道如何將這些組成物件實作出來，你還可以使用「範例程式」來協助。最後，仔細的研讀樣式使用上的優缺點來避免可能產生的問題。另外，樣式之間可能會有關係：例如複合樣式通常會與覆迴或拜訪者樣式合用；雛形樣式與抽象工廠功能類似，通常是兩者擇一採用。你可以在設計樣式的相關樣式一節中找到這個資訊。

以下列出了 GoF 的 23 個設計樣式的簡要說明：

1. 抽象工廠 (Abstract Factory)。在不需要指定明確的類別下，提供一個介面以建立一群相關的物件。因此，當系統預建立新的一群物件時，不需要改變既有的程式碼，只需擴充原來的類別即可。
2. 建築者 (Builder)。將一個複雜物件的建構與其表達分開，藉此，相同的建構程序可以用在不同的表達上，提供擴充上的彈性。
3. 工廠方法 (Factory Method)。定義一介面以生成物件，但將其生成延遲給子類別來作決定。
4. 雛形 (Prototype)。當直接生成物件的成本過高時，利用複製現有雛型實例的方式建立物件，而非採用生成的方法。
5. 單例 (Singleton)。確保一個類別只會生成單一的物件。
6. 轉接器 (Adaptor)。在不修改既有介面的情況下將一介面轉成另一個介面，藉以整合不同的物件。
7. 橋接 (Bridge)。將介面與實作分離，藉以提供介面與實作組合的多樣性。
8. 複合 (Composite)。將物件組合成樹狀的結構並同時具備部分 - 全部的包含關係。組合的結構讓客端的物件以相同的介面來看待個別物件與複合物件，藉此簡化客端物件與服務端物件的耦合力。
9. 裝飾品 (Decorator)。動態的增加物件的功能。相對於用繼承的方式來擴充功能，裝飾品提供更彈性的方法來擴充物件的功能。
10. 門戶 (Façade)。為一個子系統內眾多的服務提供一個統一的介面，藉以降低子系統間的耦合力。
11. 輕量 (Flyweight)。使用分享的方式來協助有效的管理輕量級物件的資源。
12. 代理人 (Proxy)。為某一物件提供一個中介控制的介面，以過濾對該物件的存取。
13. 責任鏈 (Chain of Responsibility)。避免將一個要求的提出者與接受者直接連結以降低他們之間的耦合力。責任鏈允許多個物件處理一個相同的要求。要求會在責任鏈中傳遞直至真正可以處理該要求的物件。
14. 命令 (Command)。將物件的需求封裝為一個類別，藉此提供更彈性的操作。例如將請求作排隊處理 (queue) 及提供請求回覆 (undo) 的功能。
15. 解析器 (Interpreter)。針對一個語言，提供該語言文法的表達法，以便於解析該語言內的結構與子句。
16. 覆迴 (Iterator)。提供一個較安全的方式以循序性的存取複合物件的內容 – 存取者不會知曉複合物件的內部的細節。
17. 調停者 (Mediator)。將物件的互動封裝為一個物件，藉以降低這群務間之件的耦合力。
18. 紀念品 (Memento)。在不破壞封裝性的前提下，紀錄並外顯化物件的狀態，以便該物件在之後可以回覆該狀態。

19. 觀察者 (Observer)。當一群物件間有一對多的相依關係時，當被依者物件的資料改變時，會通知其他依靠者物件以作出回應。
20. 狀態 (State)。將物件的狀態自物件本身獨立出來，以提高物件行為變化的彈性。
21. 策略 (Strategy)。將演算法自其使用者中獨立出來，藉此提高該演算法使用上的彈性。亦即，演算法的使用者可以在不修改自身程式的情況下更換演算法。
22. 樣板方法 (Template Method)。定義一個方法演算法的結構為若干個步驟的組合，但將每個步驟的真實演算法延遲到子類別定義，藉此提高演算法變化的彈性。
23. 拜訪者 (Visitor)。將方法自其會運作的物件中獨立出來，藉此，避免新增方法時對該物件結構作的改變。

## 6.2 設計樣式的分類

GoF 設計樣式可以由兩個層面來剖析與分類：用途 (purpose) 與範圍 (scope)。從用途來看，設計樣式可以分為生成 (creational) 用途、結構 (structural) 用途與行為 (behavioral) 用途；生成型的設計樣式抽象化生成的過程，結構型的樣式考慮類別和物件如何組成更大的結構，而行為型的樣式考慮物件間的責任分配以及互動的方式。從範圍來看，設計樣式可以分為類別範圍與物件範圍。類別型的樣式主要考慮以繼承的方式來組合物件或解決問題，而物件型的樣式則以組合 (composition) 方式來組合物件或解決問題。儘管如此，並不是表示物件型的樣式就不會用到繼承的技巧，事實上絕大部分的物件型樣式都是繼承與組合同時採用，只不過它們的重點在於組合，而非繼承。下表為此分類下的設計樣式整理。

	類別	物件
生成	將部分物件的生成延遲到子類別決定。樣式：抽象工廠	將部分物件的生成委託給其他物件生成。樣式：建築者、工廠方法、雛形、單例。
結構	使用繼承來組合介面或是實作。樣式：轉接器	使用物件組合實現新的功能及彈性，使得執行時可以更改物件間的組合。樣式：橋接器、複合、裝飾、外觀、輕量化、代理人
行為	使用繼承來分配類別間的行為。樣式：解譯、樣板方法	採用物件間的組合而非繼承，描述一群對等的物件如何協同合作以完成工作。樣式：責任鏈、命令、策略、訪問、覆迴、調停、紀念品、觀察者、狀態

## 6.3 對軟體工程的協助

設計樣式主要解決的設計階段的問題。我們可以將設計階段的問題簡單的歸類如下幾點 [3]：

- 基本問題：如何從分析階段成功的跨越到設計階段？亦即，如何精細化 (refine) 分析模組，使之符合設計上的考量？
- 品質問題：如何滿足非功能性需求以提升軟體的品質？

設計的問題沒有唯一與明確的解答，絕大部分的情況都必須視系統的特性而定。設計樣式，雖然不能解決所有的問題，但提供了絕佳的經驗分享與技巧指導。以下即針對上述三個問題來分析設計樣式的解決之道。

- 協助進入設計階段
- 協助解決非功能性需求
- 彈性化設計

**協助進入設計階段** 物件導向方法論最困難的工作之一是「如何決定物件」。亦即，我們如何將客戶的需求轉化為一群物件？在分析階段大部分的建議多是建議從問題敘述中找出名詞與動詞，並以「將名詞視為一個物件或屬性，動詞視為一個方法或關聯」等方法作為初步的分析方法。到了設計階段，由於許多細部技術的問題必須考量，物件的定義變得更為複雜，幾乎沒有什麼規則可以依循。設計樣式在此扮演著一個經驗分享的角色，描述由分析階段進入到設計階段可會遇到的問題，及相關的解決方式。

有一些設計活動是我們在設計階段必須要作的，用以解決基本的設計問題。例如系統分解 (decomposition)、物件分配 (object allocation)、存取控制 (access control)、控制流程 (control flow)、元件合成 (component composition) 等：

- 系統分解：用來降低系統的複雜度。將一個系統分解成數個較簡單的部分，稱之為子系統 (subsystem)，並將這些子系統製成一群各個相互合作的類別。設計樣式即可以幫助我們解決系統分解相關的問題，例如，當我們要將子系統與介面類別包裝起來時，外觀設計樣式 (Facade) 幫助我們降低其相依性。其他的架構性設計樣式 (architectural pattern)，如 MVC 可以協助此設計活動。
- 物件分配：用以將物件或子系統分散在不同的電腦上，以滿足高效能需求或是多個分散使用者間的相互聯繫。代理者設計樣式提供在本地端物件與遠端物件之間提供了一個代理者，所以適合用在此方面。例如，當一個物件為了提高設計的簡易度，一個物件必須存在於一個不同的位址，此時便可以應用遠端代理者 (Remote proxy) 來隱藏該物件；虛擬代理者 (Virtual proxy) 為了效能最佳化的要求而生成物件。

- 存取控制：用來提供一個更安全的多重使用者環境，我們在可在此活動中定義存取控制規則。保護代理者 (Protection proxy) 設計物件在代理者設計物件中扮演一個內部管理的角色來過濾出不適當的存取。
- 元件合成：當遭遇到重複性循環的問題可以應用現成的元件以降低開發成本。商業現成的元件可以被選出、採納並與系統結合在一起。適應者設計樣式 (Adapter) 可以將介面轉換成元件與系統間的黏著劑。

物件模組在整個軟體開發的流程中是不斷的精細化 (refine) 而成為一個真實可執行的系統的。對於沒有經驗的程式設計師而言，這樣的精細化是很困難的，需要一些建議與指引。下表列出部分的實例說明設計樣式如何輔助解決基本的設計問題：

設計活動	設計樣式	說明
系統分解	門戶	透過封裝一個帶有統一介面的子系統，減少類別間的相依性
系統分解	觀察者	減少資料物件以及邊界物件的相依性
物件定位	遠端代理者	提供一個代理人物件，以隱藏一個物件長駐在另一個不同位址的事實
物件定位	虛擬代理者	一經要求便創造所需求的物件以最佳化效能
存取控制	保護代理者	引進一個過濾器物件，預防不適當的存取
存取控制	覆迴	預防對於複合物件不適當的存取
元件組合	轉接器	引進一個類別當作擁有不同介面的元件間的黏著劑，而不是為了物件間的合作而更改它們的介面

**協助解決非功能性需求** 設計樣式可視為一種功能性需求 (functional requirement; FR) 的延伸，用以滿足非功能性的需求 (Non-functional requirement; NFR)[2]。換句話說，設計樣式除了提供基本的功能性需求外，它還同時滿足品質化、非功能性的需求。例如在觀察者設計樣式的目的描述如下：

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

分析此意圖，我們可了解此設計樣式的目的是解決一個主體物件與其相關相依物件之間的溝通問題。在功能性方面，它要求主體物件在物件更改狀態時能夠通知所有相依物件；在非功能性方面，它要求主體物件要在不知道相依物件的型態下自動的通知這些相依物件。從需求的角度來說，此設計樣式同時具備一個功能性的意圖 (FR-intent) 與延伸的非功能性意圖 (NFR-intent)。又例如 Abstract Factory 設計樣式意圖如下：

樣式	功能性	非功能性
抽象工廠	客戶端物件一次建立、使用一些相關物件(產品)	客戶端在不指定產品物件具體型態的情形下建立、使用產品物件，提高了可重用性
觀察者	當一個主題改變自己的狀態時，通知所有依靠它的物件	主題在不知道所有依靠它的物件的型態下，自動通知它們，因此提高了模組化
命令	系統支援可復原的工作	較容易增加新的工作，因此提高了可擴充性
覆迴	客戶端物件瀏覽集合容器物件	客戶端物件在不知道集合容器的內部結構下，能夠去瀏覽它的內容，因此提高了可維護性
策略	系統使用一演算法解決一特定問題	較容易使用新的演算法解決問題，因此提高了可擴充性

表 6.1: 設計樣式協助達成非功能性需求

*Provide an interface for creating families of related or dependent object without specifying their concrete classes (提供一個介面，用以創造相關或是相依物件的產品族，且不事先指定這些物件的具體類別)*

從“[功能性、非功能性]”的角度來看，此設計樣式在功能上是希望客端物件可以建立(或使用)一群相關的物件；其延伸的非功能性需求則要求在「不知曉這一群相關物件真正的型態下建立或使用這些物件」，據以達成可重用性。依據這樣的觀察，我們將設計樣式重新分析，將樣式的意圖區分為功能性意圖與非功能性意圖，藉此突顯該設計樣式對非功能需求的延伸。功能性意圖描述一個樣式要做什麼，而非功能性意圖描述其對一些品質屬性(quality attribute)的要求，像可重用性(reusability)、可維護性(maintainability)及可擴充性(extensibility)。

下表為以此角度分析下的部分實例。

相對於功能性需求與非功能性需求，我們可以建立功能性結構與非功能性結構。圖 6.1 為抽象工廠的實例。我們可以將圖(a)視為一個比較差的結構(因為他只能解決功能性意圖)，而圖(b)是比較好的結構，因為他可以同時達到功能性與非功能性的意圖。

**彈性化設計** 軟體的「修改」的難免的。需求的修改也無可避免的會造成設計的修改，因此，如何將修改所帶來的災害降到最低是軟體開發很重要的一環。彈性化的設計可部分的解決此一問題。欲達到彈性化設計，首先必須先將系統可能變化的部分抽離出來(當然，

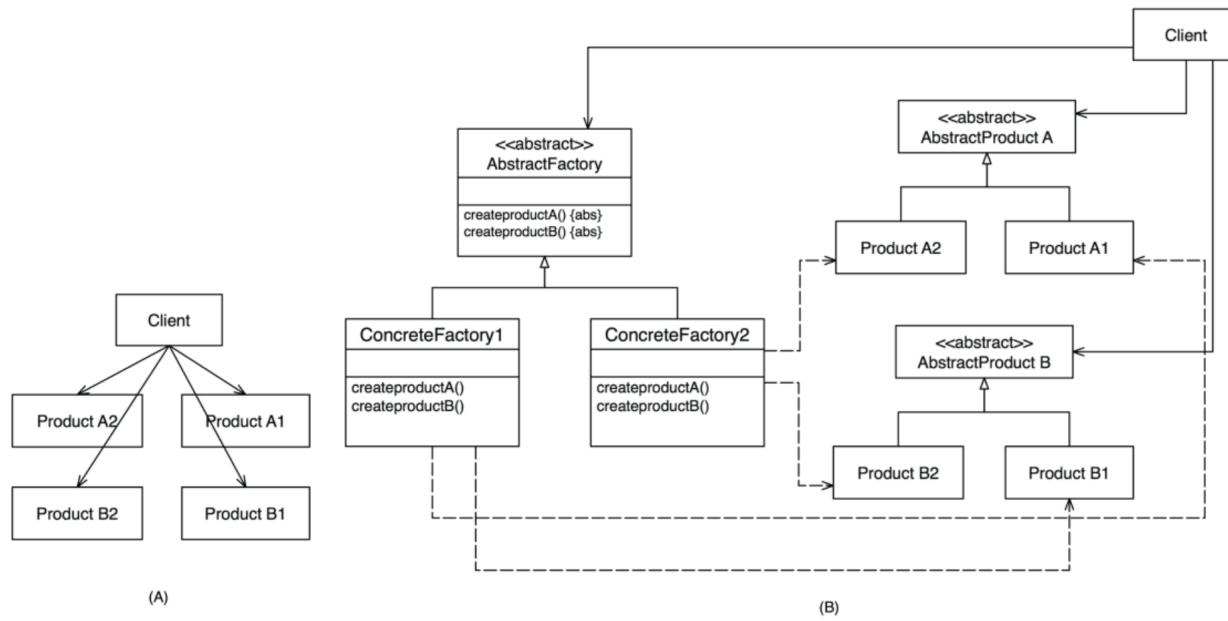


圖 6.1: 功能結構與非功能性結構

這需要一些系統分析的活動)，接下來便考慮如何用一些好的技巧來處理這些經常會變動的部分，使我們能在變動程式最少的情況下滿足這些變動。設計樣式提供了若干的協助。如下表，當我們意識到日後物件的實作很可能變動時，可以採用橋接器來應用；若是物件的狀態種類在日後很可能變動則採用狀態設計樣式。

## 6.4 樣式的選擇與採用

當我們遇到問題時，可以依照以下的步驟來選擇及採用設計樣式：

1. **瀏覽樣式的目的：**瀏覽並檢視設計樣式的「目的」，尋找可能的樣式以解決所遭遇的問題。樣式的目的只是概略性的描述，如果不確定該樣式是否可以真的可以應用，可以各進一步的參考它的「動機」、「應用時機」等。
2. **建立候選樣式群：**當我們找到可能解決問題的候選樣式後，可以透過樣式中的「相關樣式」找到其他可能的樣式，並因此列出一群候選樣式。
3. **檢驗樣式物件結構：**觀察候選樣式群中樣式的「結構」，尋找一個合適目前系統架構的樣式，亦即，該設計樣式的物件架構可以對應到目前系統的物件架構。樣式中「參與者」的敘述可以很快的協助你了解樣式的物件結構。如果物件結構所提供的訊息不夠，可以進一步的參考「物件合作」以了解這些物件間的行為關係。
4. **參考實作：**當決定採用哪個樣式後，便可以參照設計樣式中所提示的「範例程式碼」。

樣式	變動的部分	樣式	變動的部分
複合	物件的結構與複合關係	觀察者	依附於其他物件的物件數目
狀態	物件的狀態	策略	演算法
橋接	物件的實作	裝飾品	物件的功能
轉接器	物件的介面	抽象工廠	產品物件的家族
工廠方法	被生成物件的子類別	外觀	子系統的介面
輕量	物件的儲存成本	代理人	物件的存取方式、物件的定址方式
覆迴	複合式物件內的元素被存取的方式	命令	要求被履行的時機與方法
責任鏈	履行要求的物件		

表 6.2: 應用設計樣式來管理變動

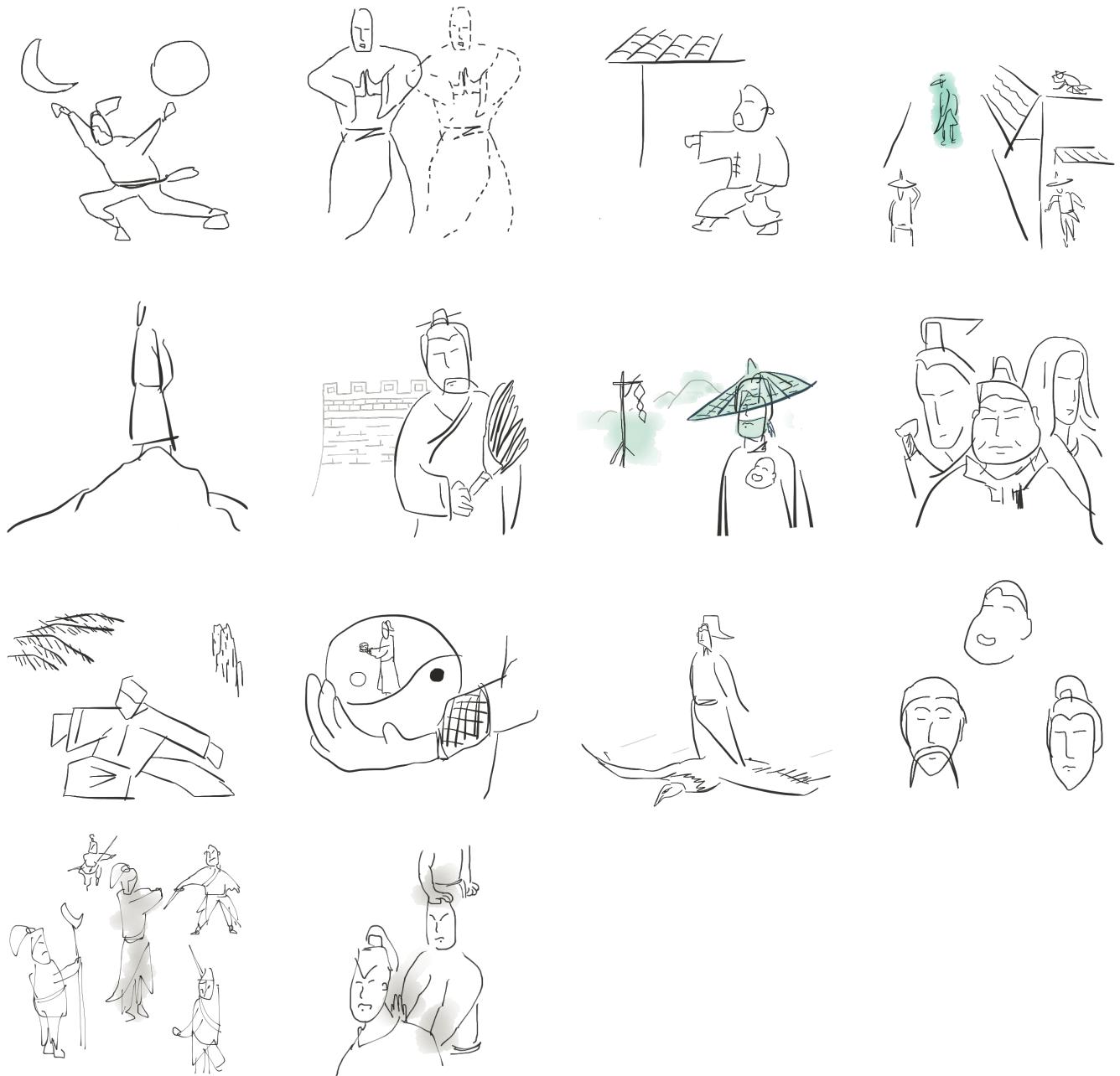
將程式應用到系統中。

對於熟悉物件技巧及設計樣式的設計者而言，他可以依照這樣的程序很快的找到他要的樣式並將之應用在系統中。然而，對於剛學習設計樣式的設計者而言，這樣的程序可能還是不夠的。學習者必須事前研讀過一些設計樣式，並實際的操作過一些例子，才能在初看到設計樣式的目的及結構時有所領會。因此，讀者應將設計樣式視為一種學習物件技巧的工具，平時就需去研讀瞭解，若等到要用到才去尋找，很難很快的找到。

## 6.5 工程師的武林世界

本講義並不會全部的設計樣式都介紹，會挑一些常見的。相信讀者若能仔細了解這些設計樣式，物件導向的觀念就通了，也不需要特別需要設計樣式了。

多數的軟體工程師都喜歡充滿幻想、奇幻的武俠世界，我也不例外。每學一個程式的技巧，就會覺得自己的武功更上一層樓般的喜悅。所以在這份講義裡特別把每個設計樣式加上一個武俠式招數，給它一張圖，藉此希望提昇大家的印象，至於圖像與樣式之間的連結，就留給大家聯想吧。



## 6.6 練習

**Ex 1:** 1994 年 Gamma 等人所出版的書籍中，共介紹多少 design pattern? (1) 15 個 (2) 23 個 (3) 78 個 (4) 108 個。

**Ex 2:** Design pattern 是有結構性的描述一個設計問題的解決方案，它通常包含哪四個項

目？

**Ex 3:** Gamma 從用途與範圍來分類設計樣式，用途分為哪三個？範圍分為哪兩個？

**Ex 4:** 設計樣式解決非功能性需求？請說明其原委。

# Chapter 7

## 乾坤挪移：Adaptor



## 7.1 目的與動機

轉換類別的介面成另一個介面所預期的樣式。Adapter 能夠讓不相容的介面，用轉接的方式合作，並且相容。

*Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.*

### 動機

各位都有要將三孔插頭插入二孔插座時的困擾吧！怎麼辦呢？除了將多餘的一角拔掉外，我們可以買一個三轉二的轉接器來做調整。在軟體的設計上，我們也常遇到過同樣的問題：

物件 A 在某個環境下使用介面  $I_1$  來達成某個功能，但換到另一個環境時，提供相同功能的物件 B 的介面卻不是  $I_1$ ，而是另一個不同的  $I_1$ 。在不修改 A 物件的呼叫與 B 物件的介面時 (正如同我們不願修改插頭與插座)，我們如何能讓物件 A 正確的呼叫我該功能？

有時候我們在引用一些類別介面時，有些功能無法引用，是因為介面不相容，我們可能有原始碼但卻不想更動到它的原始結構，或是我們並不知道它的原始碼，只知道它的操作方式。這時候使用轉接器 (Adapter) 設計樣式，做一個轉接的代理接口，就能讓兩個原本不相容的介面接合在一起。

### 應用時機

當你想使用手邊已存在的類別時，但它的介面並不相容於你所預期的樣式，或是你想發展一個可以 Reuse 的類別，讓它能夠和不相容介面相互合作。

## 7.2 結構與方法

### 結構

Adapter 所以分為 2 種，一為 (class Adapter)，一為 (object Adapter)。前者使用繼承 (inheritance) 的技巧，而後者使用委託 (delegation) 的技巧。

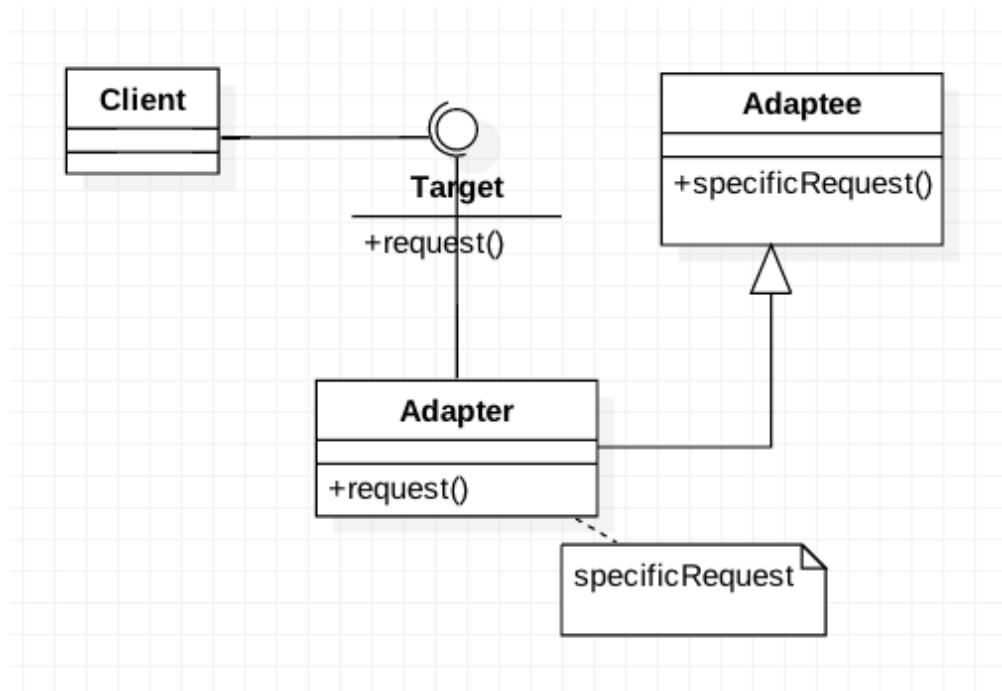


圖 7.1: Adaptor Design Pattern- Using Class Inheritance

### 程式範本

#### class adapter

程式 7.1: Class Adaptor 程式樣板

```

1 package adaptor.classadaptor;
2 // Adaptee 的 specificRequest() 對應到 Target 的 request()
3 interface Target {
4     public void request(Object arg);
5 }
6
7 class Adaptee {
  
```

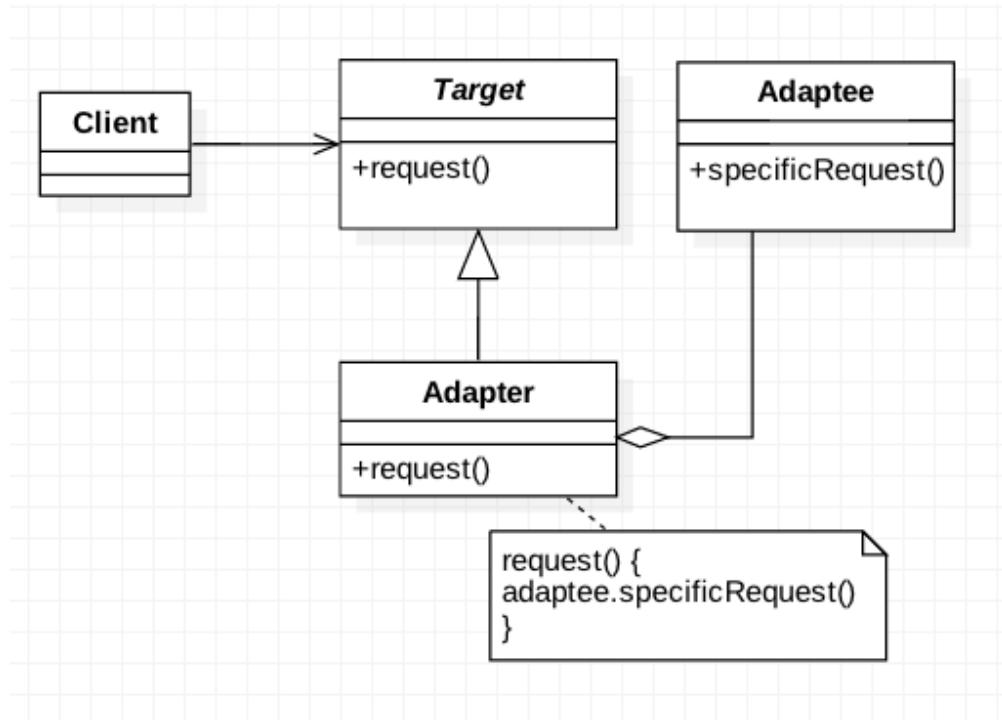


圖 7.2: Adaptor Design Pattern- Using Delegation

```

8     public void SpecificRequest(Object arg) {
9         //...
10    }
11 }
12
13 class Adapter extends Adaptee implements Target {
14     public void request(Object arg) {
15         this.SpecificRequest(arg);
16     }
17 }
18 }
19
20 class Client {
21     // t 可以是一個 Target, 或是一個 Adapter (實作了 Target)
22     public void makeRequest(Target t, Object o) {
23         t.request(o);
24     }
25 }
  
```

[\[Get the code\]](#)

## Object adapter

程式 7.2: Object Adaptor 程式樣板

```
1 package adaptor.objectadaptor;
2
3 // Adaptee 的 specificRequest() 對應到 Target 的 request()
4 class Target {
5     public void request(Object arg) {
6         // ...
7     }
8 }
9
10 class Adapter extends Target {
11     Adaptee adaptee;
12
13     public Adapter(Adaptee a) {
14         this.adaptee = a;
15     }
16
17     public void request(Object arg) {
18         adaptee.specificRequest(arg);
19     }
20 }
21
22 class Adaptee {
23     public void specificRequest(Object arg) {
24         // ...
25     }
26
27 }
28
29 class Client {
30     public void main(String args[]) {
31         Target t = new Adapter(new Adaptee());
32         t.request(new Integer(1));
33     }
34 }
```

[\[Get the code\]](#)

## 7.3 範例

### 電源轉接器

假設電腦是慣用的介面是 TwoPin, 但現在都改成 ThreePin 的插座了，請設計一個 2-3 Adaptor。

程式 7.3: 應用 Adaptor 的插座轉接範例

```

1   class TwoPin {
2       insert() { ... }
3   }
4
5   class ThreePin {
6       insertHole() { ... }
7   }
8
9   public class Adapter extends TwoPin {
10    private ThreePin threePin;
11    public Adapter(ThreePin pin) {
12        this.threePin = pin;
13    }
14    public void insert(String str) {
15        threePin.insertIntoHole(str);
16    }
17 }
18
19 public class Computer {
20     public static void main(String args[]) {
21         ThreePin threePin = new ThreePin();
22
23         TwoPin adapter = new Adaptor(threePin);
24         (new Computer()).powerOn(adapter);
25     }
26
27 // 我們不想要改變的程式，電腦的電線都是用 TwoPin 的介面
28     public void powerOn(TwoPin pin) {
29         pin.insert();
30         powerOnComputer();
31         ...
32     }
33 }
```

```

32         }
33     }

```

## Copy

考慮一個 VectorUtility 類別，其提供了一個 copy 的功能，可以將某個 Vector 複製到另一個 Vector，但前提是 Vector 內的元素必須符合 isCopyable 的介面：

```

1  class VectorUtility {
2      public static Vector copy(Vector vin) {
3          Vector vout = new Vector();
4          Enumeration e = new vin.elements();
5          while (e.hasMoreElements()) {
6              Copyable c = (Copyable)e.nextElement();
7              if (c.iscopyable) {
8                  vout.addElement(c);
9              }
10         }
11     }
12 }

```

例如 Book 符合 Copyable 的介面，則 VectorUtility 就可以複製一個以 Book 建立的 Vector：

```

1  Vector v = new Vector();
2  v.add(new Book("b1"));
3  v.add(new Book("b2"));
4  VectorUtility vu = new VectorUtility();
5  Vector v2 = vu.copy(v);

```

若我們要複製的東西是 Student，但 Student 並不支援 Copyable 介面，但提供一個相似的功能介面—isValid()。如何能讓 VectorUtility 也可以來 copy student 的 Vector 呢？

看看 Adapter 如何幫忙吧！

```

1  public class StudentAdapter implements Copyable {
2      private Student s;
3      public StudentAdapter(Student s) {
4          this.s = s;
5      }
6      public boolean isCopyable() {

```

```

7             return s.isValid();
8         }
9     }

```

其 UML 的結構如圖 7.3。

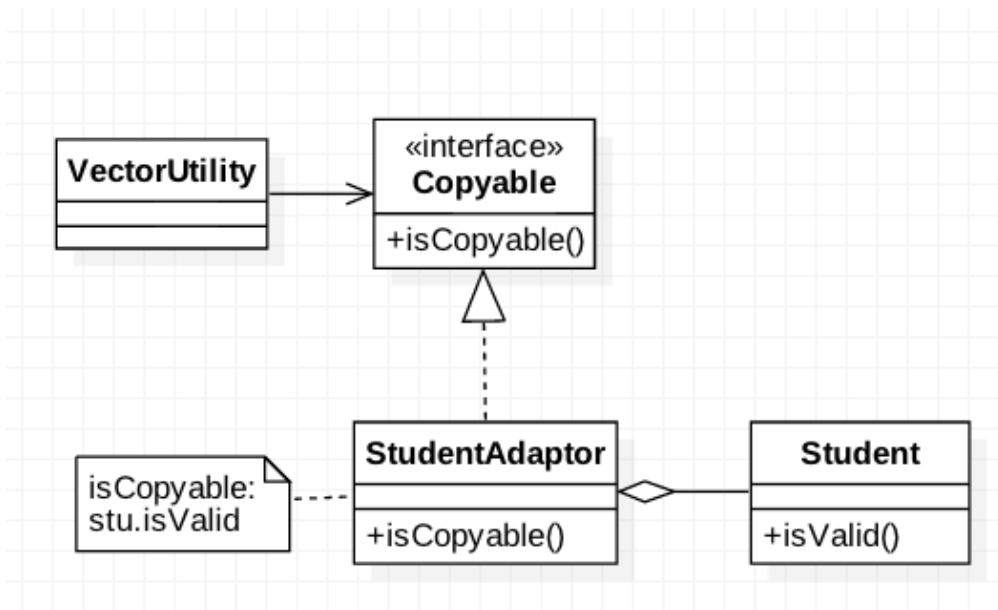


圖 7.3: 應用 Adaptor- Copyable

而使用 VectorUtility 來 copy 的方式如下：

```

1     Vector v = new Vector();
2     v.add(new StudentAdapter(new Student("s1")));
3     v.add(new StudentAdapter(new Student("s2")));
4     VectorUtility vu = new VectorUtility();
5     Vector v2 = vu.copy(v);

```

請注意 Vector 內加的物件是一個 StudentAdapter，如此才可以給 VectorUtility 判斷是否可 copy。各位會不會有個疑問：Vector 內放一群 StudentAdapter 作什麼？我的目的應該是放一群 Student 呀？可別忘了 StudentAdapter 內有一個 private 物件 Student，我們只要宣告一個介面讓外界取得到就好了。因此，將 StudentAdapter 設計如下：

```

1  class StudentAdapter implements Copyable {
2      private Student s;
3      public StudentAdapter(Student s) {
4          this.s = s;
5      }

```

```

6     public boolean isCopyable( ) {
7         return s.isValid( );
8     }
9     public Student getStudent( ) {
10        return s;
11    }
12 }
```

## Window Adaptor

Window adaptor 是另一種特殊的 adaptor.

熟悉 Java GUI 設計的人一定常利用 WindowAdapter 來做 closing 的動作：

```

1  public static void main( String arg[] ) {
2      Test application = new Test( );
3      application.addwindowlistener(
4          new WindowAdapter( ){
5              public void windowClosing (WindowEvent event){
6                  System.exit(0);
7              }
8          });
9      }
```

new WindowAdapter( ) 將會產生一個匿名類別，其為 WindowAdapter 的子類別，並修改 (Override)WindowAdapter 的方法 windowClosing。addWindowListener 的參數應該是一個 WindowListener 的物件，為何會是一個 WindowAdapter 呢？

其實 WinAdapter 實作 WindowListener :

```

1  class WindowAdaptor implements WindowListener{
2      public void windowActivated() {}
3      public void windowClosed() {}
4      public void windowClosing() {}
5  }
```

其實 WindowAdapter 內所有的 operation 都是空的。為什麼要做一個空的類別？因為如此一來 WindowAdapter 的子類別只需要 override 其想修改的 method 即可。倘若上例不用 Adapter 來做，則程式必須如此麻煩：

```
1  application.addWindowListener(
```

```

2     new WindowListener( ) {
3         public void windowActivated() {}
4         public void windowClosed() {}
5         public void windowClosing() {
6             System.exit(0);
7         }
8     }
9 );

```

各位注意到了嗎？即使 application 只想處理 windowClosing 而已，但因為它實作 WindowListener 就必須把所有的 event “照抄”一次 (內容都是空的)。所以，在此例中，WindowAdapter 做為 application 物件與 WindowListener 的轉接器。

## 7.4 練習

### 選擇/簡答

**Ex 1:** Adaptor 的目的:

- (a) 把兩個介面不相容的物件可以溝通合作
- (b) 讓一個類別只能產生一個物件
- (c) 讓一個物件可以有很多的觀者者，物件變動時，其觀察者物件可以跟著變動
- (d) 提供一個可以修改介面的介面，讓物件可以溝通

**Ex 2:** 在 Adaptor 中，與 client 溝通的物件為

- (a) Target
- (b) Adaptee

**Ex 3:** Adaptor 運用的技巧為

- (a) 讓 Target 包含一個 Adaptor 的物件，轉而呼叫 Adaptee 的方法
- (b) 設計一個 Target 的子類別 Adaptor，把 client 呼叫的方法轉而呼叫 Adaptee 的方法
- (c) 設計一個 Adaptor 類別，呼叫 Target 與 Adaptee，等於是作為兩者之間的中介，以降低耦合度

**Ex 4:** Adaptor 可分為 class adaptor 與 object adaptor。當 target 與 adaptee 都是類別 (非介面) 時，我們應該用哪一種？

- (a) class adaptor

(b) object adaptor

**Ex 5:** 當我們想要把 A.m1() 介面轉成 B.op1() 介面。回答問號? 的程式碼

```

1   class AdaptorA2B extends B {
2       A a;
3       public Adaptor(A a) {
4           ?1
5       }
6       public void op1() {
7           ?2
8       }
9   }
```

**Ex 6:** 說明物件轉接器和類別轉接器的差別

**Ex 7:** 說明 Client, Target, Adaptor, Adaptee 的關係

## 設計

**Ex 8:** 請設計一個 A 到 B, B 到 A 的雙向 Adaptor

```

1   interface A {
2       public void m1();
3   }
4   interface B {
5       public void op1();
6   }
7   class AdaptorAB ?1 {
8       ?2
9   }
```

**Ex 9:** Java 過去的集合型態 (collection) 都實踐 Enumeration 的介面，但新版的則開始使用 Iterator 的介面，我們需要一個 Enumeration 轉 Iterator 的 Adaptor，請設計之。(Target 為 Iterator)

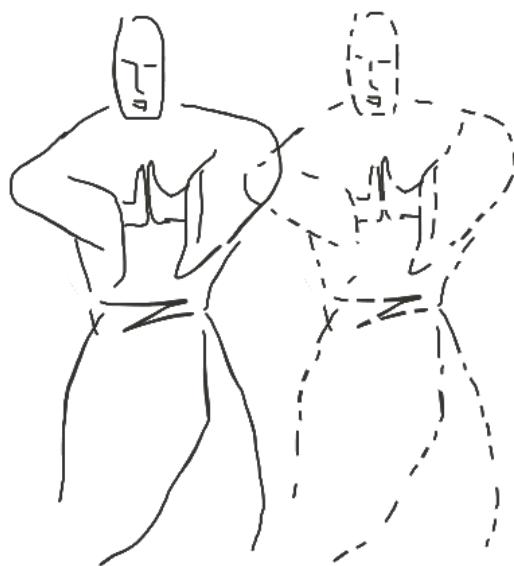
```

1   interface Enumeration<E> {
2       public boolean hasMoreElement();
3       public E nextElement();
4   }
5   interface Iterator<E> {
6       public boolean hasNext();
7       public E next();
```

```
8      }
9      class AdaptorIE ? {
10     ?
11 }
```

# Chapter 8

## 虛實分離：Bridge



## 8.1 目的與動機

把“抽象”和“實作”抽離開來，使得兩者可以獨立的變化

*Decouple an abstraction from its implementation allowing the two to vary independently.*

子類別的含意到底是什麼？為父類別實踐一個實作？還是表達一種特殊的抽象？如果當兩者都要時，該如何設計？

### 動機

當一個抽象有多個實作時方法時，通常我們會使用繼承來設計：每一個子類別表示一個不同的實作。但有時候這樣的方法沒有彈性。因為該抽象本身也可以分解成其他的類別，形成另外一個繼承結構。

例如，一個視窗可以有兩種不同的實作：*XWindow* 或是 *PMWindow*。當我們要把 *Window* 分成 *IconWindow* 和 *TransientWindow* 兩種不同的類別，那麼我們就需要設計  $2 * 2$  個類別。同理，如果當實作方面又多一個 *MacWindow*，抽象方面又多一個 *SquareWindow*，那我們就需要  $3 * 3$  個類別。類別會越來越多，有沒有可能簡化設計？

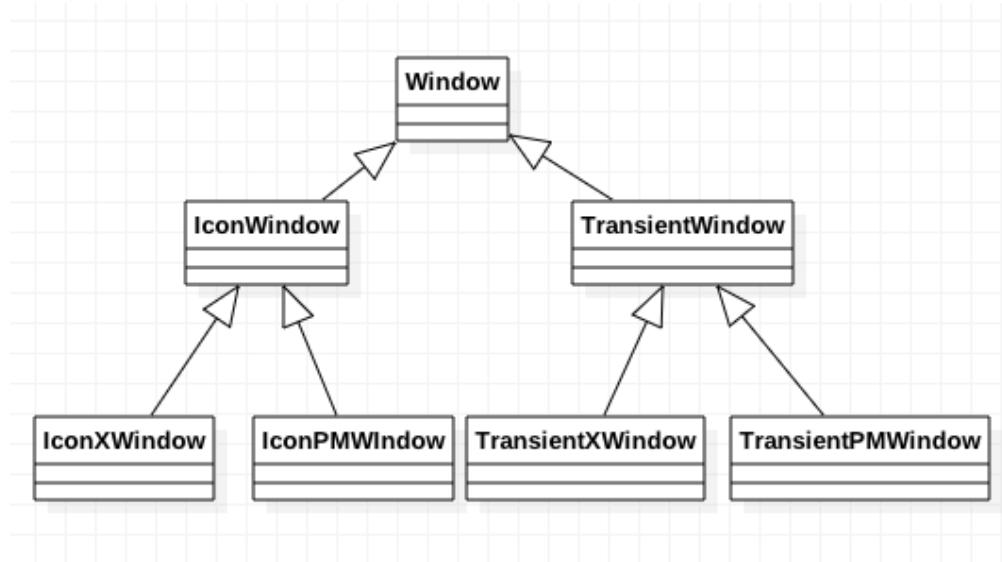


圖 8.1: Window 分類：沒有使用 Bridge 樣式

答案就是 Bridge 設計樣式。

## 應用時機

- 當我們想避免抽象和實作永遠的綁在一起時。
- 當抽象和實作都可能透過繼承來擴充時。
- 當更改實作時不會對該抽象有影響時。(例如 PMwindow 的實作方法改變了，但這不應該會引想到 IconWindow 的特性)

## 8.2 結構與方法

### 結構

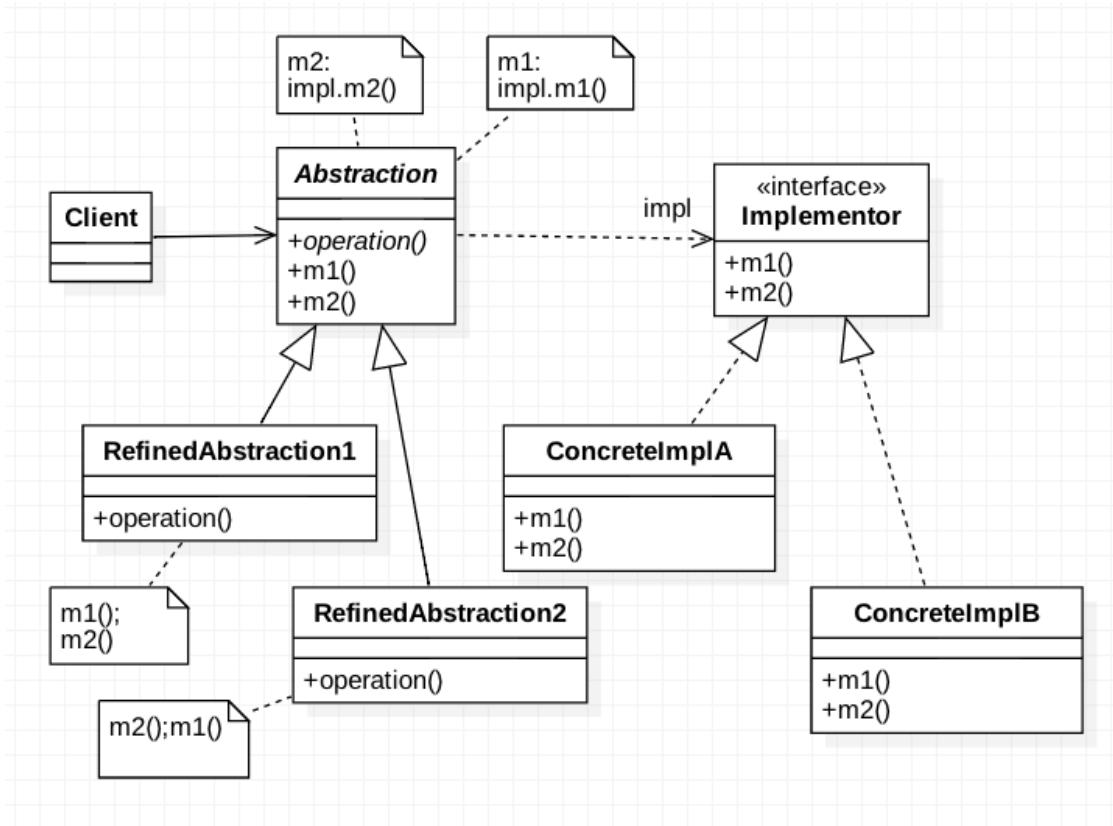


圖 8.2: Bridge Structure

- **Abstraction:** 問題空間的主要類別，它包含了這個概念主要的功能。**operation()** 及 **m1()**, **m2()** 都是這個概念的主要功能，但 **m1()**, **m2()** 這兩個方法是比較細微的方法，它的功能可以會被不同的實作方法來實踐。

- RefinedAbstraction: 上述概念的一個子分類，例如交通工具可以分為摩托車與汽車。這個類別可能會覆寫父類別的 `operation()`，用來表現這個子類別的特殊化。它的實踐可能是透過執行 `m1()`, `m2()` 等方法來實踐的。
- Implementor: 表示所有實踐者的抽象介面，它定義了所有實踐者必須履行的功能 (`m1()`, `m2()`)。注意 `Abstraction` 的 `m1()`, `m2()` 方法都是直接委託給這個介面下的實體來實踐的。
- ConcreteImplA: 真實的實踐者，例如在上述的例子中，`PMWindow` 或是 `XWindow`。

## 程式樣板

```

1  package bridge;
2
3  interface Implementor {
4      public void m1();
5      public void m2();
6  }
7
8  class ConcreteImplementorA implements Implementor {
9      public void m1() {
10         // ...
11     }
12
13     public void m2() {
14         // ...
15     }
16 }
17
18 class ConcreteImplementorB implements Implementor {
19     public void m1() {
20         // ...
21     }
22
23     public void m2() {
24         // ...
25     }
26 }
27
28 abstract class Abstraction {

```

```
29     Implementor imp;
30
31     public Abstraction(Implementor imp) {
32         this.imp = imp;
33     }
34
35     protected void m1() {
36         imp.m1();
37     }
38
39     protected void m2() {
40         imp.m2();
41     }
42
43     abstract void operation();
44 }
45
46 class RefinedAbstraction1 extends Abstraction {
47     public RefinedAbstraction1(Implementor imp) {
48         super(imp);
49     }
50
51     // 每個 RefinedAbstraction 執行 operation 的方式可能不同。
52     void operation() {
53         m1();
54         m2();
55         // ...
56     }
57 }
58
59 class RefinedAbstraction2 extends Abstraction {
60     public RefinedAbstraction2(Implementor imp) {
61         super(imp);
62     }
63
64     // 每個 RefinedAbstraction 執行 operation 的方式可能不同。
65     void operation() {
66         m2();
67         m1();
68         // ...
```

```

69         }
70     }

```

[\[Get the code\]](#)

## 8.3 範例

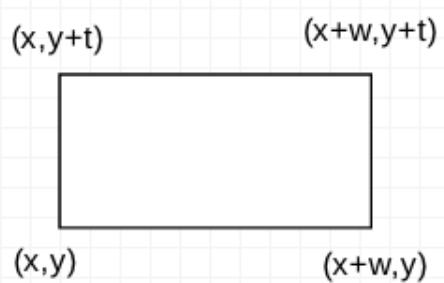
### Shape

```

1 abstract class Shape { //相當於 Abstraction
2     public abstract void draw(); //draw 相當於 operation()
3
4     ShapeImp imp; //相當於 Implementor
5     public Shape(ShapeImp imp) {
6         this.imp = imp;
7     }
8     //相當於 m1()
9     protected void drawLine(int x1, int y1, int x2, int y2) {
10        imp.drawLine(x1, y1, x2, y2);
11    }
12 }

```

draw() 是由 drawLine() 所組成的方法，而 drawLine 的真實實作是委託給 impl 來做的。 Rectangle 和 Triangle 都是形狀，其 draw() 各自不同，但都呼叫了 drawLine()。



```

1 //相當於 RefinedAbstraction1
2 class Rectangle extends Shape {
3     int x, y, w, t;
4

```

```

5      public Rectagle(int a, int b, int w, int t, ShapeImp imp) {
6          super(imp);
7          x=a; y=b; this.w=w; this.t=t;
8      }
9
10     //Rectagle 知道如何利用 drawLine 畫出方形。但他並沒有綁真正的實作。
11     public void draw() {
12         drawLine(x, y, x+w, y);
13         drawLine(x+w, y, x+w, y+t);
14         drawLine(x+w, y+t, x, y+t);
15         drawLine(x, y+t, x, y);
16     }
17 }
18
19 //相當於 RefinedAbstraction2
20 class Triangle extends Shape {
21     ...
22     public void draw() {
23         drawLine(x, y, x+w, y);
24         drawLine(x+w, y, x+w, y+t);
25         drawLine(x, y, x+w, y+t);
26     }
27 }
28
29 interface ShapeImp {
30     public void drawLine(x1, y1, x2, y2);
31 }
32
33 class Draw2D implements ShapeImp {
34     public void drawLine(x1, y1, x2, y2) {
35         // 畫出 2D 的線
36     }
37 }
38
39 class Draw3D implements ShapeImp
40     public void drawLine(x1, y1, x2, y2) {
41         // 畫出 3D 的線
42     }
43 }
44

```

```

45  class Demo {
46      public static void main( String args [] ) {
47          Shape r1 = new Rectage(1,1,2,3, new Draw2D());
48          r1.draw();
49
50          Shape r2 = new Rectage(1,1,2,3, new Draw3D());
51          r2.draw();
52
53          Shape s1 = new Triangle(1,1,2,3, new Draw3D());
54          s1.draw();
55      }
56  }

```

## 8.4 練習

**Ex 1:** Bridge 的目的為

- (a) 把抽象和實作抽離開來，使得兩者可以獨立的變化
- (b) 設計一個橋樑，讓兩個不同介面的物件可以相互合作
- (c) 設計一平台，使物件可以通過不同的管道重送訊息
- (d) 把狀態與介面抽離開來，使兩個物件可以獨立的變化。

**Ex 2:** Bridge 中，Abstract 和 Implementor 分別有 operation() 與 operationImpl() 方法，以下何者為真 (複選)：

- (a) operation() 表示一種抽象的方法或功能，operationImpl() 則是具體的實踐方法
- (b) operation() 會被 operationImpl() 所覆蓋
- (c) operation() 可能會呼叫多個 operationImpl()
- (d) operationImpl() 可能會呼叫多個 operation()
- (e) Abstraction 的子類別不可覆蓋 (override) operation()

**Ex 3:** 把圖 8.3 改用 Bridge 重新設計。

**Ex 4:** 參考講義 Shape 的例子，假設我們現有一個新的子類別平行四邊形 Parallelogram，其建構子的參數是傳入四個角的座標，請描述其 draw() 如何設計？請寫出程式碼。

```

1  class Parallelogram extends Shape {
2      public void draw(?);
3      ?
4  }
5

```

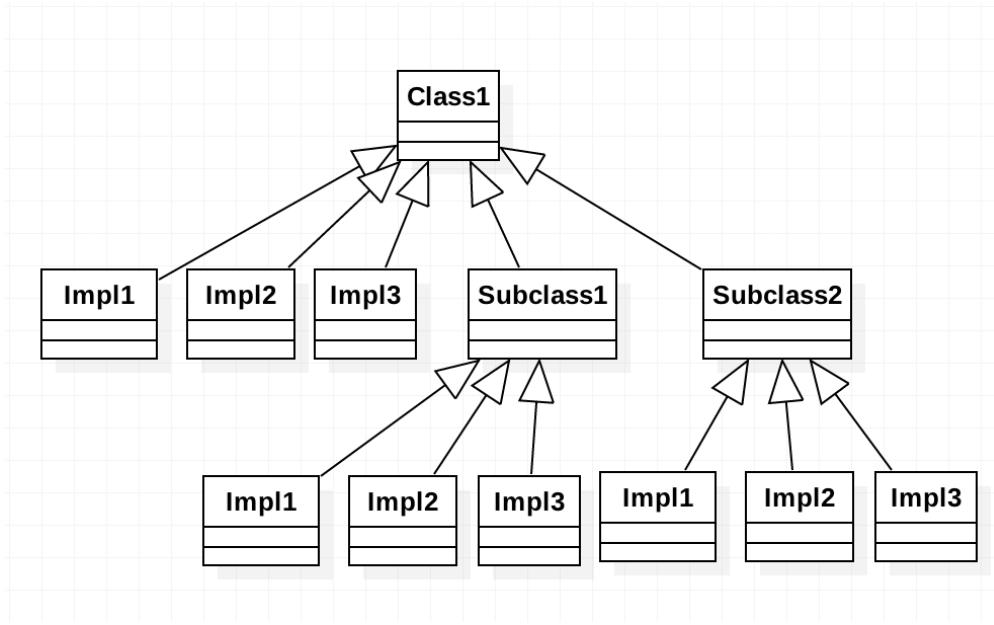


圖 8.3: 未使用 Bridge 的結構

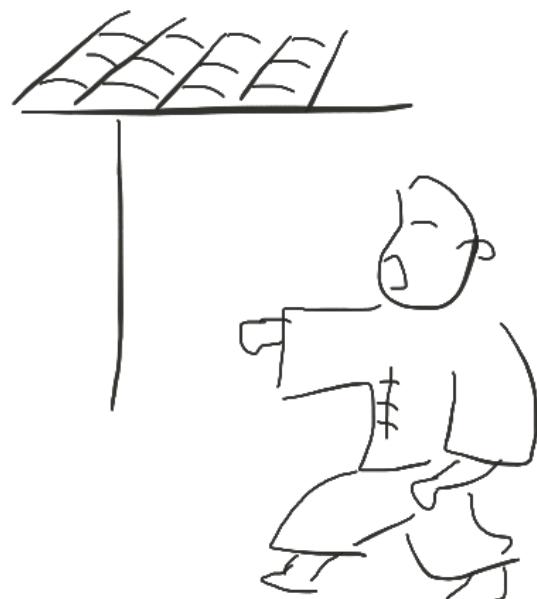
**Ex 5:** 一個概念可以分為三個子概念，從實作的角度來看有四種實作的方法。若我們不採用 Bridge 方法設計，需要設計幾個具體的類別？若採用 Bridge, 又需要幾個具體類別？

- (a) 3, 4
- (b) 12, 7
- (c) 7, 12
- (d) 4, 3



# Chapter 9

## 小器晚成：Factory Method



## 9.1 目的與動機

定義一介面以生成物件，但將其生成延遲給子類別來作決定。

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

### 動機

假設某一個 Application 物件可以產生 Document 物件以供其使用完成 Application 物件的工作。如果直接在 Application 某方法內生成文件物件，如下

```

1   class Application {
2       void operation1() {
3           doc = new Document();
4       }
5   }
```

則日後 Application 物件想建立不同型態的文件物件 (例如 HTML 文件、Word 文件) 時，則必須修改 operation1() 方法如下：

doc = new HTMLDocument();

or

doc = new WordDocument();

這樣的缺點是如果我們每一次有新的文件類別產生時就必須修改程式 (operation1()) 一次。我們可以將生成文件的動作抽象為一個方法，當日後有新的文件物件產出時，只要擴充 Application 類別即可，不需要修改文件物件。使用工廠方法的動機即為解決此類問題。下圖為採用此設計樣式後的結構。

### 應用時機

- 建立者 (Creator) 無法預期將產生何種物件，並希望其子類別來決定生產何種物件時。

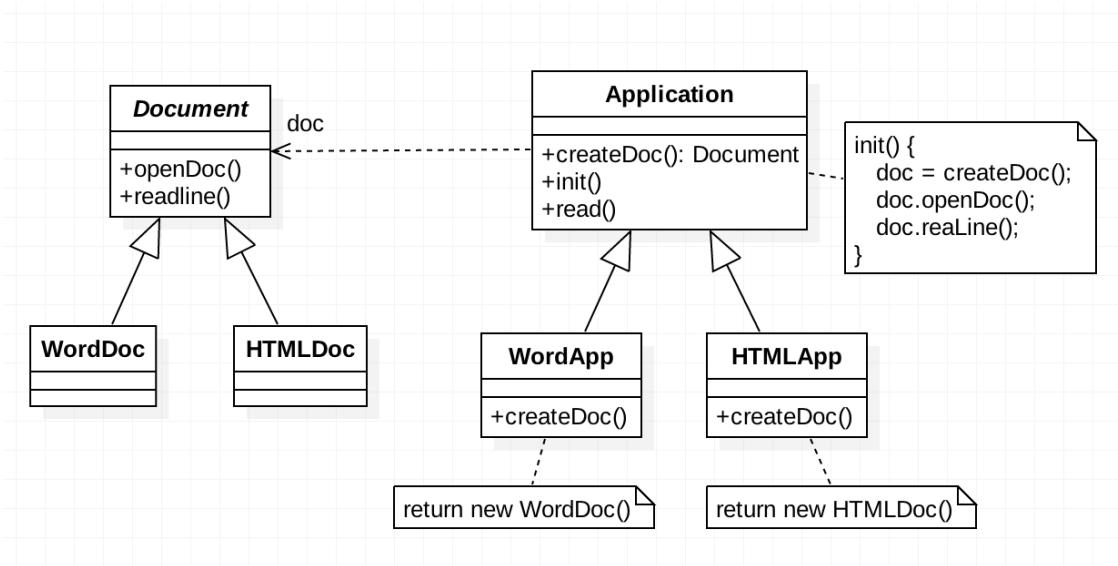


圖 9.1: Using factory method

## 9.2 結構與方法

### 參與者

- Creator (建立者): 宣告 Factory Method，回應一個物件型態的 Product。
- ConcreteCreator (實際建立者): 修改 Factory Method，回應一個新的 ConcreteProduct。與 Product 的關係為：Creator 會建立 Product 物件。
- Product (產品或零件): 定義物件的 Factory Method 型態介面。
- ConcreteProduct (實際產品): 實作的 Product 介面。

### 程式樣板

```

1 package factorymethod;
2
3 abstract class Creator {
4     Product p;
5
6     public void createProduct() {
7         p = factoryMethod();
8         // ...
9     }
  
```

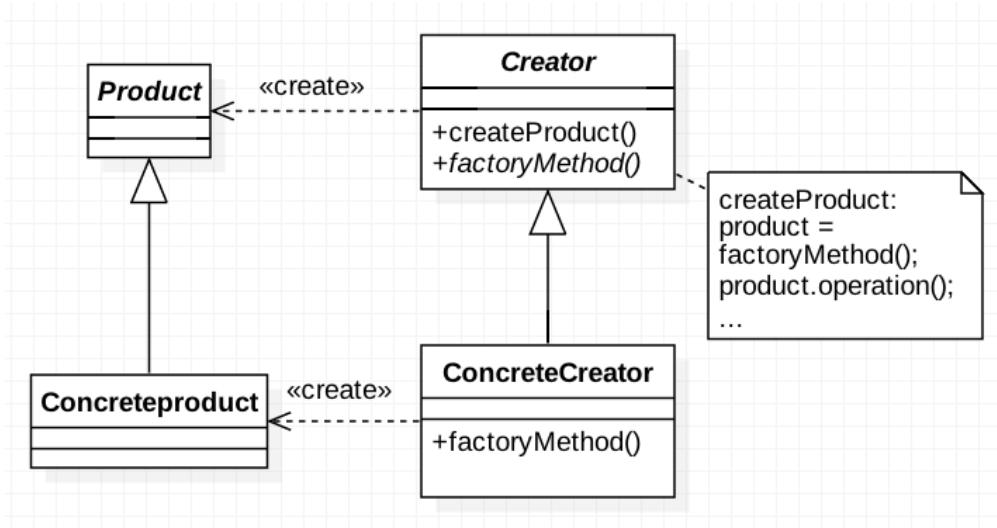


圖 9.2: Factory Method 結構

```

10
11     public abstract Product factoryMethod();
12 }
13
14 class ConcreteCreator extends Creator {
15     public Product factoryMethod() {
16         return new ConcreteProduct();
17     }
18 }
19
20 abstract class Product {
21     abstract void operation();
22 }
23
24 class ConcreteProduct extends Product {
25     void operation() {
26         // ...
27     }
28 }

```

[\[Get the code\]](#)

## 優缺點

- 優：增加程式的擴充性，避免生產特別用途的物件就需要修改程式（符合 OCP 原則）。
- 缺：生產者只為了生產另一個特別的產品時，就必須建立一個子類別，可能會產生過多的子類別而增加管理上的負擔。

## 9.3 範例

### 9.3.1 迷宮範例

接下來以 Gamma 書中的實例說明 Factory Method 的用處。考慮一個迷宮的程式。一個迷宮包含許多房間、牆、門等物件，所以當 MazeGame 在建立時，必須建立一些「零件」物件，如 Room、Door、Wall 等：

#### Solution 1: 未使用設計樣式

```

1  public class MazeGame {
2      // 建立一個迷宮
3      public Maze createMaze() {
4          //產生建立迷宮遊戲的所需零件物件，包括 Maze、Door、Room 等
5          Maze maze = new Maze();
6          Room r1 = new Room(1);
7          Room r2 = new Room(2);
8          Door door = new Door(r1, r2);
9          //建立各子物件之間的關聯
10         maze.addRoom(r1);
11         maze.addRoom(r2); r1.setSide(MazeGame.North, new Wall());
12         r1.setSide(MazeGame.East, door);
13         r1.setSide(MazeGame.South, new Wall());
14         r1.setSide(MazeGame.West, new Wall());
15         r2.setSide(MazeGame.North, new Wall());
16         r2.setSide(MazeGame.East, new Wall());
17         r2.setSide(MazeGame.South, new Wall());
18         r2.setSide(MazeGame.West, door);
19         return maze;
20     }
21 }
```

程式中的第 5-8 行建立 Room、Door 等零件物件，第 10-18 行則建立這些物件的關係。下圖描述此實例的架構。

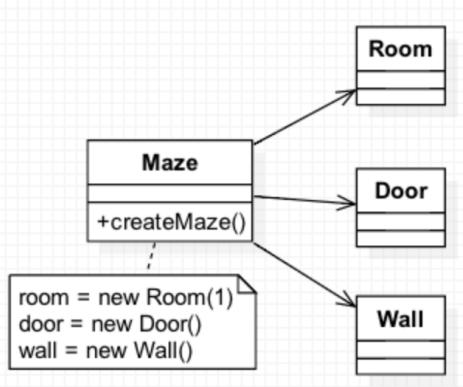


圖 9.3: No Factory Method

如果現在我們想擴充迷宮的功能，例如擴充迷宮內的房間是具備魔法的房間（enchanted room），我們可以建立一個一個新的類別 EnchantedRoom，令其繼承 Room，如下：

```

1   class EnchantedRoom extends Room {
2       ...
3   }
  
```

而 MazeGame 中的 createMaze 方法也作修改，使其建立的房間是 EnchantedRoom，不是 Room：

```

6     Room r1 = new EnchantedRoom(1);
7     Room r2 = new EnchantedRoom(2);
  
```

這樣的程式在執行上並不會有問題，只是它「違背了 OCP 的原則」 – 我們為了新增的功能而修改了原存在的類別內容。

我們可以使用工廠方法來解決這個問題，將「生產 Room 的工作」抽象為一個方法：

### Solution 2: 使用設計樣式

```

1  public class MazeGame {
2      // 建立一個迷宮
3      public Maze createMaze() {
4          //產生建立迷宮遊戲的所需零件物件，包括 Maze、Door、Room 等
5          Maze maze = makeMaze(); //! 採用抽象方法
6          Room r1 = makeRoom(1); //! 採用抽象方法
  
```

```

7     Room r2 = makeRoom(2); //! 採用抽象方法
8     Door door = makeDoor(r1, r2); //! 採用抽象方法
9     //建立各子物件之間的關聯
10    maze.addRoom(r1);
11    maze.addRoom(r2); r1.setSide(MazeGame.North, new Wall());
12    r1.setSide(MazeGame.East, door);
13    r1.setSide(MazeGame.South, new Wall());
14    r1.setSide(MazeGame.West, new Wall());
15    r2.setSide(MazeGame.North, new Wall());
16    r2.setSide(MazeGame.East, new Wall());
17    r2.setSide(MazeGame.South, new Wall());
18    r2.setSide(MazeGame.West, door);
19    return maze;
20 }
21 public Maze makeMaze() {
22     //將生產 Maze 物件的工作抽象成一個方法
23     return new Maze();
24 }
25 public Room makeRoom(int n) {
26     //將生產 Room 物件的工作抽象成一個方法
27     return new Room(n);
28 }
29 public Wall makeWall() {
30     return new Wall();
31 }
32 public Door makeDoor(Room r1, Room r2) {
33     //將生產 Door 物件的工作抽象成一個方法
34     return new Door(r1, r2);
35 }
36 }
```

請注意第 5-8 行的目的是在建立 Room、Door 等物件，與上一個程式的第 5-8 行的目的是一樣的，可是我們卻將生產 Door，Room 等工作包成一個方法 (makeDoor()、makeRoom() 等)，其目的在提供一個子類別覆蓋的機會。21-36 行是 factory method 的設計。請看新版的 EnchantedMazeGame：

```

1  public class EnchantedMazeGame extends MazeGame {
2      public Room makeRoom(int n) {
3          return new EnchantedRoom(n);
4      }
```

```

5     public Wall makeWall() {
6         return new EnchantedWall();
7     }
8     public Door makeDoor(Room r1, Room r2) {
9         return new EnchantedDoor(r1, r2);
10    }
11}

```

請將本程式的第 03 行與 solution 2 的第 27 行相比較，前者回傳一個 EnchantedRoom 物件，而後者回傳一個 Room 物件；相同的是 makeRoom 在這兩個程式中的介面都沒有變。所以，EnchantedMazeGame 擴充 MazeGame 後所做的事很單純，就是讓所有生產物件的方法生產新的物件。我們再做兩個觀察：

- OCP 的原則吻合了嗎？是的，我們的功能加強了，但沒有任何程式碼做了修改。我們新增 EnchantedMazeGame 與 EnchantedRoom 等類別。
- makeRoom() 的介面有相容嗎？是的，雖然 MazeGame.makeRoom() 的傳回型態是定義為 Room，而 EnchantedMazeGame.makeRoom() 是傳回 EnchantedRoom 物件，但因為 EnchantedRoom 繼承自 Room，因而型態上是相容的。

相對應的 UML 圖如下：你可以把這些類別內的方法寫上嗎？

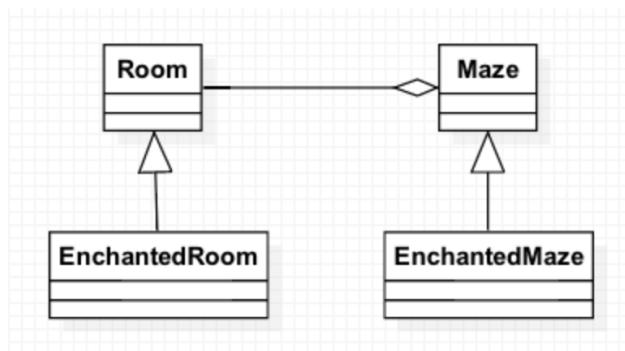


圖 9.4: Using Factory Method

如上的 UML 圖：新增類別以擴充功能：EnchantedMazeGame 與 EnchantedRoom

心得

我們獲得的心得是：在設計此某物件 (MazeGame) 時，如果知道它一定會建立某些零件物件 (Room) 時，為了想保留零件物件的彈性 (亦即可以建立零件物件的子類別物件)，我們可以將建立零件物件的功能特別獨立出來成為一個方法 (makeRoom())，這個方法就稱為 Factory Method(makeRoom())。

了解 Factory Method 後，Abstract Factory 各位就更容易瞭解了。相較於 Factory Method 將生產包裝成一個方法，Abstract Factory 將生產包裝成一個類別。我們將在下一章節介紹 Abstract Factory。

## 9.4 練習

**Ex 1:** Factory method 的目的

- (a) 可以一次產生很多物件
- (b) 確保一次只產生一個物件
- (c) 把物件生成延遲到子類別
- (d) 透過延遲生成物件來轉接物件介面

**Ex 2:** 下方程式碼中，程式 m2() 是一個 factory method, 下列何者正確？

- (a) ?1 為 A
- (b) ?1 為 B
- (c) 若?2 為 return new C(); 則 C 為 B 的子類別
- (d) ?2 為 return new A();

```

1   class A {
2       public void m1() {
3           ...
4           B b = m2();
5           ...
6       }
7       public ?1 m2() {
8           ?2
9       }
10  }
```

**Ex 3:** 說明 Factory Method 的目的

**Ex 4:** 以下程式中，哪一個方法可能是 factory method?

```

1   class A {
2       public void m1() {
3           ...
4       }
5       public B m2() {
6           return new B();
7   }
```

```

8     public int m3() {
9         int x = 0;
10        for ... .
11        ...
12        return x;
13    }
14 }
```

## 設計

**Ex 5:** 在考試系統中，可能進行各種不同的考試，例如小考 Quiz、正式考試 Exam、隨堂練習 Practice 等。ExamApp 這個物件會產生考試物件，並開始進行一連串的活動（產生考卷、進行考試、閱卷）。請應用 factory method 來協助設計，並且降低 ExamApp 與考試類型之間的耦合力。

**Ex 6:** 在象棋系統中，可以如何應用 factory method?

**Ex 7:** 針對每一個可能的產品我們都需要產生一個 ConcreteCreator, 有時候很麻煩，於是我們一個 ProductFactory 類別如下，Creater 只要把 ProductID 傳給 ProductFactory 就可以產生該物件回傳。

```

1  public class ProductFactory{
2      public Product createProduct(int ProductID){
3          if (ProductID==ID1)
4              return new ProductA();
5          if (ProductID==ID2)
6              return new ProductB();
7          return null;
8      }
9  }
```

這種方法又稱為 Parameterized Factory。請問可能會有什麼問題？

# **Chapter 10**

## **一式多款：Abstract Factory**

## 10.1 目的與動機

提供一個介面物件以建立一群相關的物件，但卻不明確的指明這些物件的所屬類別，用以增加建立這些物件時的彈性。

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

### 動機

考慮一個 Computer 的物件在運作的時候需要用到 CPU、Memory、MotherBoard 等零件物件。如果我們在方法 `make()` 中產生這些零件物件，如下：

```

1  class Computer {
2      void make() {
3          cpu = new PC();
4          memory = new Memory();
5          mb = new MotherBoard();
6      }
7  }
```

則日後 Computer 物件想建立不同型態的零件物件 (例如工作站 CPU、工作站 Memory、工作站主機板) 時，則必須修改 `make()` 方法如下：

```

1  cpu = new WorkStationCPU();
2  memory = new WorkStationMemory();
3  mb = new WorkStationMainBoard();
```

這樣的缺點是如果我們每一次有新的電腦類別產生時就必須修改程式 (`make()`) 一次。我們可以將生成這一群零件物件的動作抽象為一個工廠類別，當日後有新的零件物件產出時，只要擴充這個工廠類別即可，不需要修改原程式的程式碼。

### 應用時機

- 當系統的目標是生產具有許多類似的物件時，又有動態配置產品的需求時。
- 當所有生產的產品物件有一種家族系列的關係時 (family of product)。

## 10.2 結構與方法

### 結構

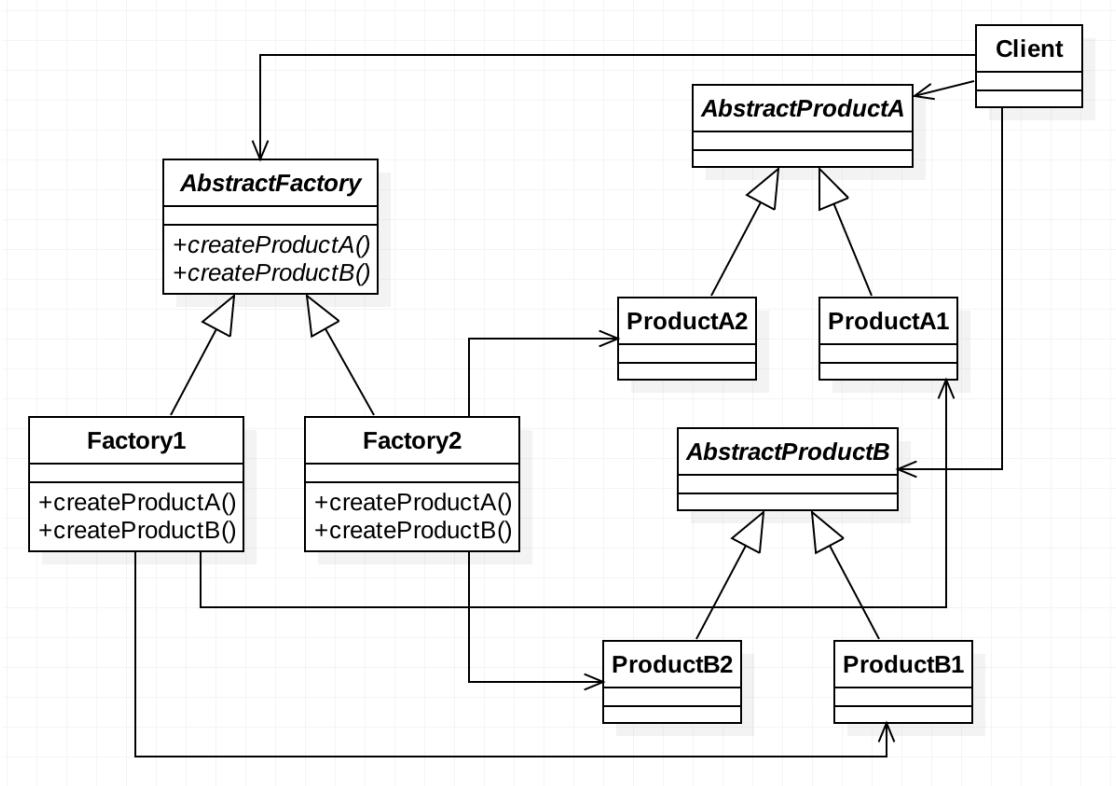


圖 10.1: 抽象工廠結構

抽象生成工廠的特色是系統可以拆解分成幾個家族，每個家族內有一些相近的成員類別。上圖為抽象生成工廠的架構圖，其中 `ConcreteFactory1` 與 `ConcreteFactory2` 為兩個不同的家族，每個家族在運作時，都會用到 `ProductA` 與 `ProductB` 等成員物件，但前者會用到 `ProductA1` 與 `ProductB1`，而後者會用到 `ProductA2` 與 `ProductB2`。

### 參與者

- 抽象工廠 (`AbstractFactory`)：宣告一個介面，宣告生成零件物件的方法。
- 實體工廠 (`ConcreteFactory`)：實際負責生產的物件。一個實體工廠負責生成一個家族的所有零件物件。
- 抽象產品 (`AbstractProduct`)：宣告某項零件物件的共同性質。
- 實體產品 (`ConcreteProduct`)：實際存在的零件類別。

- 使用者 (Client)：使用者。實際需要遇到抽象工廠所產生的零件物件的物件。

**DIP 原則** Client 只會看到抽象的物件 AbstractProductA, AbstractProductB, AbstractFactory 等類別，並不會看到比較低階的 ProductA2, ProductB1 等物件，這大大的降低了 client 與這些物件的耦合力 (coupling)。這也體現了「相依倒轉原則 (Dependency Inversion Principle; DIP)」。

**OCP 原則** 當我們有心的一組產品被開發出來，只需要透過繼承產生 ConcreteFactory3, ProductA3, ProductB3 即可，不需要修改 Client 中的程式碼。

## 程式樣板

程式 10.1: 抽象工廠程式樣板

```

1  class Client {
2      // 不同的 AbstractFactory f，可以使用不同的一組 Product (ex. A, B, ...)
3      void doSomeThing(AbstractFacotry f) {
4          AbstractProductA a = f.createProductA();
5          AbstractProductB b = f.createProductB();
6          ...
7      }
8  }
9
10 abstract class AbstractProductA {}
11 class ProductA1 extends AbstractProductA {}
12 class ProductA2 extends AbstractProductA {}
13
14 abstract class AbstractProductB {}
15 class ProductB1 extends AbstractProductB {}
16 class ProductB2 extends AbstractProductB {}
17
18 abstract class AbstractFacotry {
19     abstract AbstractProductA createProductA();
20     abstract AbstractProductB createProductB();
21 }
22
23 class ConcreteFactory1 {
24     AbstractProductA createProductA() {

```

```

25         return new ProductA();
26     }
27     AbstractProductB createProductB() {
28         return new ProductB();
29     }
30 }
31
32 class ConcreteFactory2 {
33     AbstractProductA createProductA() {
34         return new ProductA2();
35     }
36     AbstractProductB createProductB() {
37         return new ProductB2();
38     }
39 }

```

## 效益

- 抽象生成工廠最大的好處在於簡化家族間的切換。當系統想要使用某個家族類別時，只要傳入該家族類別的生成工廠即可，整個家族類別所需要的成員類別可以依序建立以供使用。比起 Factory Method 將生產只是包裝成一個方法，在 Abstract Factory 中，則是將生產包裝成一個一個的類別，更能夠表示出一個工廠生產產品的特性。

### 10.3 範例

AF 可以有很多的應用。再電腦工廠中... 在迷宮系統中，一般型態的迷宮是一個家族、有魔法的迷宮是一個家族。不論是哪一種家族，都需要用到零件物件如牆壁、房間、門等。

在象棋系統中，長棋是一個家族、短棋是一個家族。不論是哪一種家族，都需要用到零件物件如象棋規則、象棋棋盤、棋局管理等。

## 電腦工廠

```

1  class ComputerFactory {
2      public CPU makeCPU() {
3          return new CPU();

```

```

4      }
5      public Memory makeMemory() {
6          return new Memory();
7      }
8      public MotherBoard makeMotherBoard() {
9          return new MotherBoard();
10     }
11 }
12
13 class Computer {
14     public Computer createComputer(ComputerFactory factory) {
15         // 使用 factory 來產生所有的零件物件
16         CPU cpu = factory.makeCPU();
17         Memory memory = factory.makeMemory();
18         MotherBoard mb = factory.makeMotherBoard();
19     }
20 }
21
22 class WorkstationFactory {
23     public CPU makeCPU() {
24         return new WorkstationCPU();
25     }
26     public Memory makeMemory() {
27         return new WorkstationMemory();
28     }
29     public MotherBoard makeMotherBoard() {
30         return new WorkstationMotherBoard();
31     }
32 }

```

請注意 Workstation 的各零件是都是 Computer 的子類別

```

1     class WorkstationCPU extends CPU { ... }
2     class WorkstationMemory extends Memory { ... }
3     class WorkstationMotherBoard extends MotherBoard { ... }

```

當我們想要生產 workstation 時，只要帶入 WorkstationFactory 就好了：

```

1     ComputerFactory factory = new WorkstationFactory();
2     computer.createComputer(factory);

```

如果要生產 PC，則帶入預設的 ComputerFactory；

```

1 ComputerFactory factory = new ComputerFactory();
2 computer.createComputer(factory);

```

## 迷宮

首先我們以 Gamma 一書所提的迷宮程式來做介紹，在之前我們用 Factory Method 的方法去製作另一間 EnchantedRoom，EnchantedRoom 本身必須要有 EnchantedRoom 必須的 Wall 和 Door。但是當我們希望能夠動態搭配 Wall 和 Door，這時候 Factory Method 所提供的便不夠了，我們可以改用 Abstract Factory 來解決這個問題。

首先我們必須宣告一個 MazeFactory，負責去宣告 makeMaze()、makeRoom()、makeWall()、makeDoor() 等操作方法的介面：

程式 10.2: 沒有使用樣式的設計

```

1 class MazeFactory {
2     public Maze makeMaze() {
3         return new Maze();
4     }
5     public Room makeRoom(int n) {
6         return new Room(n);
7     }
8     public Wall makeWall() {
9         return new Wall();
10    }
11    public Door makeDoor(Room r1, Room r2) {
12        return new Door(r1, r2);
13    }
14 }

```

然後我們製作一個 MazeGame 負責去做一個主要操作的描述：

程式 10.3: 迷宮遊戲採用 Abstract Factory 的設計

```

1 class MazeGame {
2     static String North="north";
3     static String East="east";
4     static String South="south";
5     static String West="west";
6     Room r1, r2;
7     Door door;

```

```

8     Wall w1, w2, w3, w4, w5, w6;
9
10    MazeFactory factory;
11   public Maze createMaze() {
12       // use factory to create all products
13       Maze maze = factory.makeMaze();
14       r1 = factory.makeRoom(1);
15       r2 = factory.makeRoom(2);
16       door = factory.makeDoor(r1, r2);
17       w1 = factory.makeWall();
18       w2 = factory.makeWall();
19       w3 = factory.makeWall();
20       w4 = factory.makeWall();
21       w5 = factory.makeWall();
22       w6 = factory.makeWall();
23
24       // communication between products
25       maze.addRoom(r1);
26       maze.addRoom(r2);
27       r1.setSide(MazeGame.North, w1);
28       r1.setSide(MazeGame.East, door);
29       r1.setSide(MazeGame.South, w2);
30       r1.setSide(MazeGame.West, w3);
31       r2.setSide(MazeGame.North, w4);
32       r2.setSide(MazeGame.East, w1);
33       r2.setSide(MazeGame.South, w3);
34       r2.setSide(MazeGame.West, door);
35       return maze;
36   }
37 }
```

上述的程式應用了『包含』的關係來連接 MazeGame 和 MazeFactory。我們也可以把 MazeFactory 當成參數傳到 MazeGame 中，如下：

```

1  class MazeGame {
2      ...
3      public Maze createMaze(MazeFactory f) {
4          ...
5      }
6  }
```

除了 MazeFactory 外，我們亦做了一個 EnchantedMazeFactory 來表示出動態產生的結果。

程式 10.4: 透過擴充建立一個新的迷宮遊戲

```

1   class EnchantedMazeFactory extends MazeFactory {
2       public Maze makeMaze() {
3           return new EnchantedMaze();
4       }
5       public Room makeRoom(int n) {
6           return new EnchantedRoom(n);
7       }
8       public Wall makeWall() {
9           return new EnchantedWall();
10      }
11      public Door makeDoor(Room r1, Room r2) {
12          return new EnchantedDoor(r1, r2);
13      }
14  }
```

在這個系統的設計中，也許大家發現了一個奇怪的地方，也就是 MazeFactory 為何不是抽象的，不是應該抽像一個 MazeFactory，讓各種 Maze 的實際生成工廠去引用嗎？在這裡的 MazeFactory 其實是扮演了兩個角色，本身是抽象工廠，也同時是負責實際生產的工廠。

對照抽象工廠的架構，程式中的 MazeFactory 相當於 AbstractFactory，其他的對應關係如下：

- AbstractFactory : MazeFactory
- ConcreteFactory : EnchantedMazeFactory , MazeFactory
- AbstractProduct : 此例中沒有
- ConcreteProduct : Maze 、 Room 、 Wall 、 Door
- Client : MazeGame

[\[Get the code\]](#)

## 熱區與冰區

熱區 (Hot spot) 表示程式中會經常變動 (擴充) 的地方，冰區 (frozen spot) 則是不會變動的地方，也就是可以被重用 (reuse) 的地方。以上述的例子來看，冰區會是

MazeGame.createMaze() 這個方法，也就是說：生成的物件與他們之間關係的建立是不變。

熱區則是建立 factory 物件的主程式了。

```

1  class GameDemo {
2      public void static main( String [] args ) {
3          //f 可以變動
4          MazeFactory f = new AgentMazeFactory();
5          MazeGame = new MazeGame( f );
6      }
7  }
```

## 比較

在 Factory Method 中，我們介紹工廠方法將「物件的生成封裝成一個方法」，透過覆寫，我們可以在不需要修改程式碼的情況下讓系統使用新的類別。抽象工廠 (abstract factory) 和 Factory Method

比較：抽象工廠將「物件的生成封裝成一個類別」，而工廠方法是將「物件的生成封裝為一個方法」。

## 10.4 練習

### 選擇/簡答

**Ex 1:** Abstract factory 的目的為何？

- (a) 把物件的生成延遲到子類別;
- (b) 轉接兩個介面不同的物件
- (c) 把同一系列的物件群的生成委託給一個物件
- (d) 一次只能生成一個物件

**Ex 2:** 在 Abstract factory 設計樣式中，設計階段 client 會與哪些類別關聯

- (a) abstract factory
- (b) concrete factory
- (c) abstract product
- (d) concrete product

**Ex 3:** Abstract factory 樣式中，abstract factory 內宣告 m 個抽象方法，表示

- (a) 有 m 個系列
- (b) 有 m 個零件

**Ex 4:** 每一個 concrete factory 可以

- (a) 產生某一系列的某一個零件
- (b) 產生同一系列很多零件
- (c) 產生同一零件很多系列

**Ex 5:** Abstract factory 樣式中，有 n 個 concrete factory，表示

- (a) 有 n 個系列
- (b) 有 n 個零件

**Ex 6:** Abstract factory 和 factory method 的異同為何？

**Ex 7:** 不要看講義，畫出 abstract factory 的架構圖。

## 設計

**Ex 8:** 鞋子工廠，一定要製造鞋身 (shoes body)、鞋帶 (shoes strap)、鞋底 (shoes bottom) 三個零件，不同型態的鞋子，例如運動鞋 (sport shoes)、休閒鞋 (leisure shoes)、皮鞋 (leather shoes) 都會用到不同型態的零件。假設製造鞋子流程都是固定的，寫在 makeShoes() 方法中，而我們也希望重用這樣的流程，不想因為製造不同的鞋子就換修改到 makeShoes 的程式，因此我們採用 Abstract Factory 來設計。請畫出 UML 架構圖。

**Ex 9:** 同上，請撰寫此程式。

**Ex 10:** 以象棋系統為例，Abstract Factory 可能有什麼應用？



# Chapter 11

## 眾觀其變：Observer



## 11.1 目的與動機

定義一個「一對多」的相依關係，使得當「一」的物件狀態改變時，所有相依於「一」的「多」物件會被通知到並作適當的修改。

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

### 動機

考慮一個股票資料有三種呈現方式，股價及成交量修改後，其相關的呈現方式也要跟著改變

- 當股價變動時，跟著呼叫介面物件做修改。股價的資料屬於資料物件（model），介面物件屬於 view。資料物件直接呼叫介面物件是一種不好的設計，因為介面物件的變動性大，資料物件會因為介面物件的改變而需要做改變。
- 介面物件每隔一段時間去讀取資料物件。問題是：我們無法知道多久該去讀取一次。

#### Solution 1: No Observe

```

1      class Stock {
2          price ...;
3          public priceChange(int newPrice) {
4              this.price = newPrice;
5              //三個 view 三個不同的方法。耦合度高，不好的設計
6              view1.refresh();
7              view2.update();
8              view3.reload();
9          }
10     }

```

Solution 1 是最直覺的方法，當狀態改變時就叫每一個呈現去修改，但呈現是變動的，不該企業邏輯放在一起。如果我們增加一個新的介面、更換成 Android 或是 HTML 的介面是不是企業邏輯也要跟著修改呢？

**Solution 2** 讓 Viewer 定期的去取得資料的狀態，然後更新。這樣的問題是：我們該多久去取一次？1 秒？10 秒？時間過於密集可能會浪費頻寬、過於鬆散可能取得不正確的資料。

如何解決這個問題呢？答案是 Observer 設計模式。在 Observer 中，像股價等資料通常被稱主體 (Subject) 或被觀察者 (Observable)，而呈現方式則稱為觀察者 (Observer)。

## 應用時機

- 當後端資料有所變更，有必須即時的更新前端資料呈現。
- 當一個事務有兩個角度，其中一個角度相依於另一個。

## 11.2 結構與方法

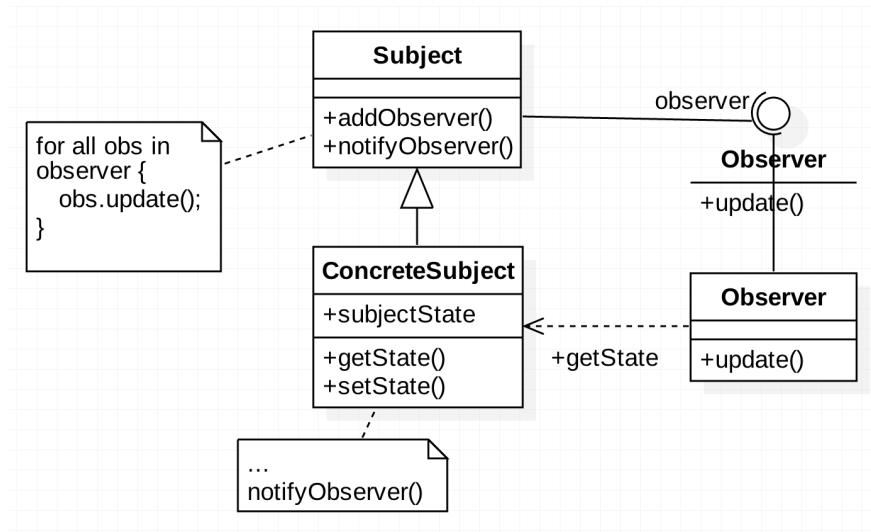


圖 11.1: Observer Structure

## 參與者

- **Subject (Observable)**：定義一個有多個觀察者的資料的基本資料型態與介面。其中的 `addObserver()` 表示加入一個新的 `observer`，而 `notifyObserver()` 表示要通知所有與其相關的觀察者。
- **ConcreteSubject (ConcreteObservable)**：實際的被觀察者。
- **Observer**：定義一個觀察者的基本結構與介面，其中的 `update()` 為觀察者收到被觀察者資料異動訊息時的處理程序。
- **ConcreteObserver**：實際的觀察者。

## 程式樣板

程式 11.1: Observer 程式樣板

```
1 package observer;
2
3 import java.util.Observable;
4
5 public class ObserverTemplate {
6
7     public static void main(String[] args) {
8         Subject s = new Subject();
9
10        View1 v1 = new View1();
11        View2 v2 = new View2();
12        s.addObserver(v1);
13        s.addObserver(v2);
14    }
15
16 }
17
18 class Subject extends java.util.Observable {
19     int data;
20
21     public Subject() {
22         data = 0;
23     }
24
25     public void setData(int newValue) {
26         data = newValue;
27         this.setChanged();
28         this.notifyObservers();
29     }
30
31 }
32
33 class View1 implements java.util.Observer {
34     public void update(Observable arg0, Object arg1) {
35         // update the view
36     }
37 }
```

```

37     }
38
39     class View2 implements java.util.Observer {
40         public void update(Observable o, Object arg) {
41             // update the view
42         }
43     }

```

[\[Get the code\]](#)

## 優點

分離了資料模組與呈現模組使得溝通能夠更容易廣泛的被應用，當資料變更不需觀察者做出更新動作才能更新，保持資料呈現的一致性。

## 11.3 範例

### 11.3.1 Observable 的應用

Java 已經針對這個設計樣式沒計了一個 API, 其中 **Observable** 相對於 Subject, **Observer** 則名稱不變。

```

1  public class Fruit extends Observable {
2      private String name;
3      private float price;
4
5      public Fruit(String name, float price) {
6          this.name = name;
7          this.price = price;
8          System.out.println("Fruit created: " + name + " at " +
9              price);
10     }
11
12     public String getName() {return name;}
13     public float getPrice() {return price;}
14
15     public void setPrice(float price) {
16         this.price = price;

```

```

16         setChanged();
17         notifyObservers(new Float(price));
18     }
19 }
```

為何需要先 `setChanged()` 再呼叫 `notifyObserver()`? 因為 `notifyObserver` 是 `public` 的，外部物件可以呼叫 `notifyObserver`，但物件的狀態可能沒有變化。`notifyObserver()` 會先檢查是否 `hasChanged()`，如果有才會呼叫 `update()`。`Fruit` 自己可以確定狀態改變時執行 `setChanged()` 以保證 `update()` 的執行。`notifyObservers()` 內部在執行完 `update()` 後也會呼叫 `clearChanged()`。有時候我們會變更一連串的狀態後才會 `setChanged()`，允許通知其他的 `Observers`。

接著來看看 `Observer` 這一端：

```

1 public class Monkey implements Observer {
2     float price;
3     public void update(Observable obj, Object arg) {
4         if (arg instanceof Float) {
5             price = ((Float)arg).floatValue();
6             System.out.println("水果價格變成" + price);
7         }
8     }
9 }
```

`WineMaker` 也關心價格波動，是另一個觀察者，當價格有變動時，它就會有所反應。

```

1 public class WineMaker implements Observer {
2     float originalPrice;
3     public void update(Observable obj, Object arg) {
4         if (arg instanceof Float) {
5             price = ((Float)arg).floatValue();
6             if ((price - originalPrice) / price < -0.1)
7                 System.out.println("Can make wine");
8             else
9                 System.out.println("too expensive");
10        }
11    }
12 }
```

來看看主程式

```
1 public class TestObservers {
```

```

2   public static void main(String args[]) {
3       // Create the Subject and Observers.
4       Fruit s = new Fruit("Grape", 1.29f);
5       Monkey jj = new Monkey();
6       WineMaker wm = new WineMaker();
7
8       // Add those Observers!
9       s.addObserver(jj);
10      s.addObserver(wm);
11
12      //make changes to the Subject.
13      s.setPrice(4.57f);
14      s.setPrice(9.22f);
15  }
16 }
```

## 委託的應用

如果 ConcreteSubject 已經有繼承了另一個類別了，無法繼承 Observable 那該怎麼辦？我們可以用委託的方式把 observable 委給 delegatedObservable。

```

1  // 水果是植物
2  public class Fruit extends Plant {
3      private String name;
4      private float price;
5      private DelegatedOBS observable;
6
7      public Fruit(String name, float price) {
8          this.name = name;
9          this.price = price;
10         System.out.println("Fruit created: " + name + " at " +
11             price);
12         observable = new DelegatedOBS();
13     }
14
15     public String getName() {return name;}
16     public float getPrice() {return price;}
17
18     public void setPrice(float price) {
```

```

18         this.price = price;
19         observable.setChanged();
20         observable.notifyObservers(new Float(price));
21     }
22     public Observable getObservable() {
23         return observable;
24     }
25 }
```

DelegatedOBS 是一個 Observable 的子類別：

```

1 // A subclass of Observable that allows delegation.
2 public class DelegatedOBS extends Observable {
3     public void clearChanged() {
4         super.clearChanged();
5     }
6
7     public void setChanged() {
8         super.setChanged();
9     }
10 }
```

大家會不會覺得奇怪，為什麼不直接委給 Observable，而是在宣告一個 DelegatedOBS，然後委給 DelegatedOBS？原來 Observable.setChanged() 被設定為 protected，如果沒有透過繼承是無法呼叫的，因此我們將之繼承後再開放為 public。

=> 想一下：為什麼 setChanged() 要宣告成 protected?

```

1 public class TestObservers2 {
2     public static void main(String args[]) {
3         // Create the Subject and Observers.
4         Fruit s = new Fruit("Grape", 1.29f);
5         Monkey jj = new Monkey();
6         WineMaker wm = new WineMaker();
7
8         // Add those Observers!
9         s.getObservable().addObserver(jj);
10        s.getObservable().addObserver(wm);
11
12        //make changes to the Subject.
13        s.setPrice(4.57f);
14        s.setPrice(9.22f);
```

```

15      }
16  }
```

=> 動一下：把上述的例子用 UML 來表示。

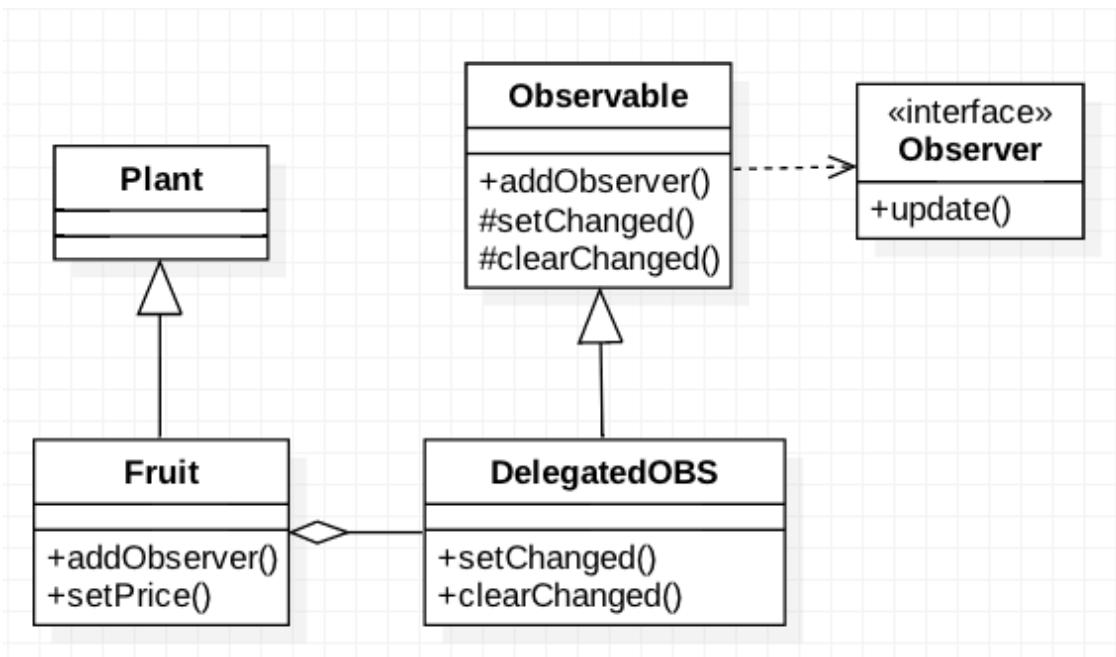


圖 11.2: Observer with delegation

**私有漏洞** 各位可以看到第 9-10 行的 `s.getObservable().addObserver(nameObs)`，先透過 `getObservable()` 獲得 `Observable` 物件，再透過它來作 `addObserver` 的動作。這樣的缺點是外界的物件很容易取得 `Observable` 的參考，就有可能拿著這個參考胡作非為（例如 `deleteObserver()`）。為了避免這種狀況，我們新增 `addObserver()` 這個方法，在裡面進行委託；並且移除 `getObservable()` 的方法，避免私有漏洞的可能。

```

1  public class Fruit extends Plant {
2      ...
3      public void addObserver(Observer o) {
4          observable.addObserver(o);
5      }
6      ...
7  }
```

### 11.3.2 ActionListener

JAVA 的 event model 與 Observer 的架構類似。

- AbstractButton => Observable; fireActionListener() => notifyObserver()
- ActionListener => Observer; actionPerformed() => update()

其中的 AbstractButton 就相當於 Observer 中的 Observable，而向它註冊的就是那些監聽事件發生的類別，也就是實作 ActionListener 的物件 (Event Handler)。由於 JButton 本身已是 AbstractButton 的子類別，我們只要直接在 JButton 的實作中加入事件監聽者即可：

```

1  public class TestEventManager extends JFrame{
2      private JButton b1;
3      public TestEventManager() {
4          b1 = new JButton("Button");
5          getContentPane().add(b1);
6
7          //相當於 addObserver()
8          b1.addActionListener(new Listener1());
9          b1.addActionListener(new Listener2());
10         ...
11     }
12 }
13
14 //ActionListener 相當於 Observer, actionPerformed 相當於 update()
15 class Listener1 implements ActionListener {
16     public void actionPerformed(ActionEvent e) {
17         System.out.println("Event happen");
18     }
19 }
20
21 class Listener2 implements ActionListener {
22     public void actionPerformed(ActionEvent e) {
23         System.out.println("Hello world");
24     }
25 }
```

AbstractButton 內的 fireActionPerformed() 相當於 Observable 內的 notifyObservers()，但我們不需要去呼叫它，因為當我們按下 Button 時會直接呼叫 fireActionPerformed()，進而呼叫所有的 ActionListener 內的 actionPerformed()。

有時候程式不是很複雜時，事件的發生與處理在同一個類別內，所以常可以看到這樣的程式碼：

```

1  class TestEventModel2 extends JFrame implements ActionListener {
2      ...
3      b1.addActionListener(this);
4      public void actionPerformed(ActionEvent e){
5          ...
6      }
7  }

```

此時的 TestEventModel2 同時兼具了 event source 與 event listener 的功能，亦即觀察者與被觀察者的雙重身分。

## 11.4 練習

### 選擇

**Ex 1:** Observer 設計樣式主要有兩個物件：Subject 與 Observer:

- (a) 一個 subject，多個 observer
- (b) 一個 observer 一個 subject
- (c) 一對一的關係
- (d) 多對多的關係。

**Ex 2:** 關於 Observer 樣式，何者為真：

- (a) Observer 變動時，Subject 被通知
- (b) Observer 定時查詢 Subject 狀態
- (c) Subject 定期查詢 Observer 狀態
- (d) Subject 變動時，Observer 會被通知

**Ex 3:** java API 中實踐 Subject 的類別為

- (a) Object
- (b) Subject
- (c) Observable
- (d) Observer

**Ex 4:** Java 的 Swing 架構使用 Observer，其中 ActionListener 相當於 Observer 樣式中的？

- (a) Subject

- (b) Observer
- (c) Concrete Observer
- (d) Concrete Subject

**Ex 5:** 同上，像 JButton 這一類的元件，相當於 Observer 樣式的？

- (a) Subject
- (b) Observer
- (c) Concrete Observer
- (d) Concrete Subject

## 簡答

**Ex 6:** 請寫出 java.util.Observer 此介面。注意參數的正確。

```

1  interface Observer {
2      ?
3  }
```

**Ex 7:** 請畫出 Observer 的架構

**Ex 8:** 應用 Observer 樣式時，擔任 Subject 的類別，當狀態改變時，要呼叫哪一個方法？

**Ex 9:** 以下 View1 是一個 Observer, ?1 和?2 為何

```

1  class View1 implements Observer {
2      public void update(?1 obs, ?2 obj) {
3          ...
4      }
5  }
```

**Ex 10:** Stock 是一個 Subject, 價格改變時會通知所有的 observer, 以下? 為何

```

1  class Stock extends Observable {
2      public void increasePrice() {
3          price++;
4          setChanged();
5          ?
6      }
7  }
```

**Ex 11:** 為何 Subject 可以通知所有不同的物件，又可以避免過高的耦合力？

**Ex 12:** 當 Subject 物件已經有必要的父類別時，無法再繼承 java.util.Observable, 該如何實踐 Observer 樣式？

**Ex 13:** 在透過委託來實踐 Observer 的方法中，為何 Subject 不直接委託給 Observable 物件，而是委託給 DelegatedObservable？

## 設計

**Ex 14:** 如果我們不透過 java API 來實踐 Observer，想要自己在 Stock 類別中實踐 Observable 的功能，該怎麼設計 addObserver(), notifyObserver() 等方法？資料結構該怎麼設計？

```

1   Hint:
2   class Stock {
3       Vector observers;
4       public Stock() {
5           observers = new Vector();
6       }
7       addObserver(?) {
8           ?
9       }
10      notifyObserver(?) {
11          ?
12      }
13  }
```

**Ex 15:** 假設你設計一個象棋遊戲，遊戲狀態有 waiting, started, end 三個狀態。當狀態改變時會傳給多個介面，如 PlayerView, CustomerView, AllGameStatusView 等三個介面做呈現。

- 請透過 java 的 Observable 來設計此問題。
- 若 ChessGame 本身已經繼承 Game, 無法在繼承 Observable, 該怎麼辦？

**Ex 16:** 股票（Stock）物件內包含昨日價格、現價與成交量三個屬性，現價與成交量每個 2 秒變動一次（請隨機產生在 7%, 10% 內的價格），請應用 Observer 設計樣式設計以下三個呈現：。

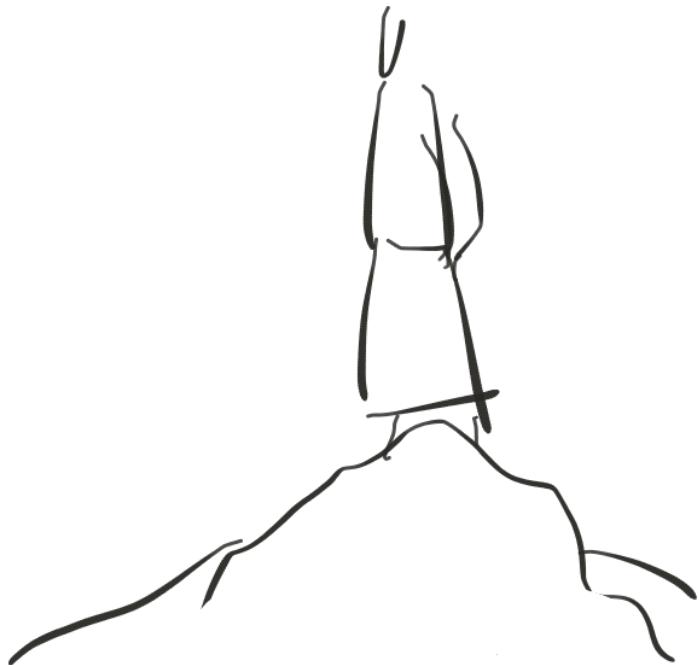
- CurrentPriceBoard: 呈現昨日價格 (Y)、目前價格 (C)、及波動百分比  $((C-Y)/C)$ 。
- AmountBoard: 呈現現價、成交量。
- GreenRedBoard: 最近三次的價格，如果連三漲，背景設為綠色，如果連三跌，背景設為紅色。否則維持原色（白色）。

**Ex 17:** 同上，(1) 不要透過繼承 observable 的方式來實踐 Observer 設計樣式。透過 delegation 的方式委託給 observable 來間接實踐 observable (2) 不用 java.util.Observable, 將 Observable 的功能直接寫在 Stock 中，並自己建立一個 Observer 的介面。



# Chapter 12

## 獨一無二：Singleton



## 12.1 目的與動機

確定一個類別只會有一個物件實體，並且提供一個可以存取該物件的統一方法。

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

### 動機

過去我一直很喜歡用的一個雲端產品「evernote」，它滿足我隨時隨地想作筆記的需求。但在 2015 年以後我對開始失望，其一的原因是它的附件功能，明明我是設定為「附件模式」，但他卻常常是以「內置展開」的模式，當我把一群物件放在一個筆記時，很難快速的看到第二筆以後的附件。我期望放很多檔案，但放很多檔案時卻操作不良。

第二個問題是重複打開同一附件時，每一次它都會產生一個「實體」。例如 project101.doc 就會產生很多份，我因此違反「資料唯一性」的原則。過去 evernote 並沒有這樣的問題，不知道為何它要改成這樣設計？

這就是 Singleton 設計樣式的動機。有時候我們在系統中只想要一個實體，唯一的一個。例如說我們只需要一個視窗管理員，只需要一個 RadioPlayer(要不然聲音就打架了)，或只需要一個產品的工廠物件（請參考 Abstract factory）。我們希望該唯一的物件很容易的被讀取到；並且確定不會有其他的物件被產生出來。

## 12.2 結構與方法

### 結構

### 優點

控制外界的物件只能參考到同一唯一的一個物件。

### 沒有繼承樹的獨體

這個設計的技巧在於

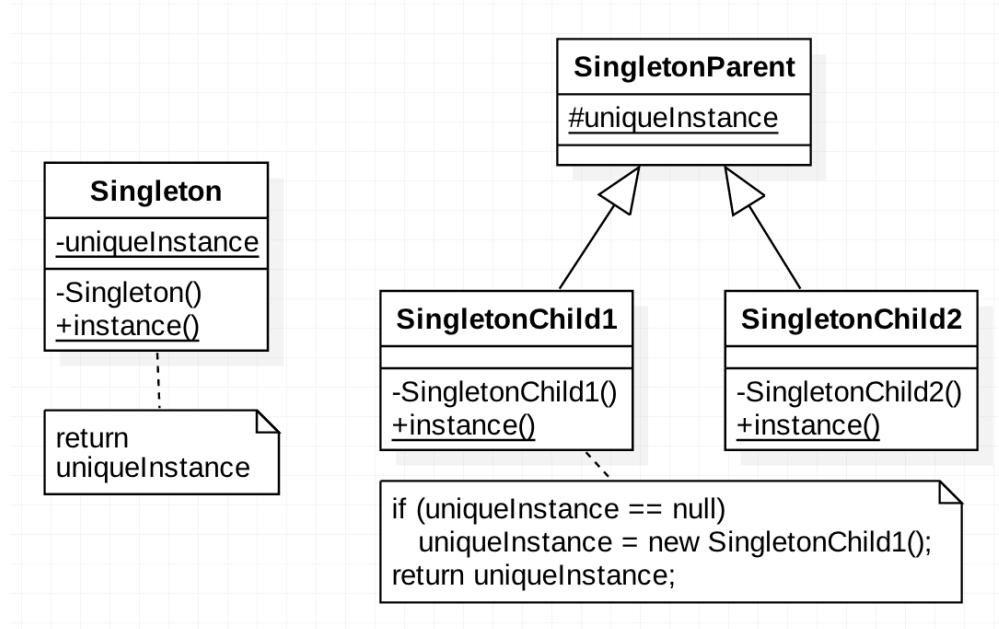


圖 12.1: Singleton Structure

- 宣告一個靜態的物件參考；
- 將原有的建構子宣告為私有的；
- 建立另外一個生成物件的方法，通常稱為 `instance()`，它判斷是否物件已經生成了，若已生成則不再生成，若未生成則生成一個。

以下是一個範例：

```

1  public class Singleton {
2      // 宣告為 static 讓物件唯一
3      private static Singleton uniqueInstance = null;
4      private int data = 0;
5
6      public static Singleton instance() {
7          if(uniqueInstance == null)
8              uniqueInstance = new Singleton();
9          return uniqueInstance;
10     }
11     // 把一般的建構子宣告為私有
12     private Singleton() {}
13 }
  
```

主程式

```

1  public class TestSingleton {
2      public static void main(String args[]) {
3          Singleton s = Singleton.instance();
4          s.setData(34);
5          System.out.println("s的參考為：" + s);
6          System.out.println("s的值為：" + s.getData());
7
8          Singleton s2 = Singleton.instance();
9          System.out.println("s2的參考為：" + s2);
10         System.out.println("s2的值為：" + s2.getData());
11     }
12 }
```

執行的結果

```

s 的參考為：Singleton@1cc810
s 的值為：34
s2 的參考為：Singleton@1cc810
s2 的值為：34
```

有上述的例子可以看到，不如我們呼叫多少次 `Singleton.instance`，回傳的都是相同的物件。

### 有繼承樹的獨體

如果在一個繼承樹中只允許產生一個物件，該怎麼設計？

- 把該物件的 `reference` 建立在父類別中，並且宣告為 `protected`; 如此一來，子類別都可共享這一份物件了。
- 父類別把元建構子宣告為 `private`, 因為不允許其他物件透過父類別來生成物件：一切都要從子類別來生成。
- 子類別的原有建構子宣告為 `private`，如此一來，其他物件無法透過建構子『偷生』其他的物件實體了。
- 因為父類別與子類別們都共享同一份物件參考，所以就可以控制只生一個物件了。

## 12.3 範例

### 迷宮範例

不論是 EnchantedMazeFactory 或 AgentMazeFactory 只能生成一個物件。

```

1  public abstract class MazeFactory {
2      //宣告成 protected 這樣子類別才看得到
3      protected static MazeFactory uniqueInstance = null;
4
5      //藏起來
6      protected MazeFactory() {}
7
8      // Return a reference to the single instance.
9      public static MazeFactory instance() {
10         return uniqueInstance;
11     }
12 }
13
14 public class EnchantedMazeFactory extends MazeFactory {
15     //參考到父類別的 uniqueInstance
16     public static MazeFactory instance() {
17         if(uniqueInstance == null)
18             uniqueInstance = new EnchantedMazeFactory();
19         return uniqueInstance;
20     }
21     private EnchantedMazeFactory() {}
22 }
23
24 public class AgentMazeFactory extends MazeFactory {
25     //同樣的參考到父類別的 uniqueInstance
26     public static MazeFactory instance() {
27         if(uniqueInstance == null)
28             uniqueInstance = new EnchantedMazeFactory();
29         return uniqueInstance;
30     }
31     private EnchantedMazeFactory() {}
32 }
```

## 比較

Factory method 和 abstract factory 都是討論設計的彈性，希望日後在功能擴充時減少程式碼的修改。Singleton 的目的在於解決設計上的問題，是少數設計樣式中不討論設計彈性的樣式。

## 12.4 練習

### 選擇/簡答

**Ex 1:** Singleton 的目的為

- (a) 快速的為陣列內每一個類別產生一個物件
- (b) 讓一個類別只能產生一個物件
- (c) 讓一個套件只能產生一個類別
- (d) 強迫一個類別只能有一個子類別

**Ex 2:** static method 的定義哪一個錯誤

- (a) 不需產生物件就可以呼叫
- (b) Singleton 產生物件的方法是呼叫 static method
- (c) 必須引用類別中的 instance variable
- (d) 可以引用類別中的 static variable

**Ex 3:** Singleton with subclassing 的意義為

- (a) 一個繼承樹上只能產生一個物件
- (b) 一個繼承樹上每一個類別都只能產生一個物件

**Ex 4:** 在象棋系統中，「將」「士」等每一個 Chess 都個別只會有一隻，所以 Chess 可以用 Singleton 來設計。

- (a) 對
- (b) 錯

**Ex 5:** Singleton 的目的為何？

**Ex 6:** Singleton 主要應用的物件技巧為何？

**Ex 7:** Singleton 應用在繼承樹時，主要應用的技巧為何？

## 設計

**Ex 8:** class Vehicle 有兩個子類別 Motor 與 Bike。若我們只想生成一個 Vehicle (不論他是 Motor, Bike)，請利用 Singleton 設計之。



# Chapter 13

## 一法萬策：Strategy



## 13.1 目的與動機

定義一群演算法，將每一個封裝成一個類別且使之可互換。使用 Strategy 讓演算法獨立於使用者。

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

### 動機

在很多種情況我們都會用到策略樣式。假設我們開發一個應用程式，其中會對某個陣列做排序。也許我們一開始會用氣泡排序法，且知道以後這個演算法可以改善，例如用 quick sort, selection sort。我們不希望抽換這些排序方法的時候會對其他程式造成影響，也就是說，我們應該符合 OCP 原則。我們該怎麼設計？

Solution: 使用 strategy 設計樣式，將各種不同的排序抽象出來：

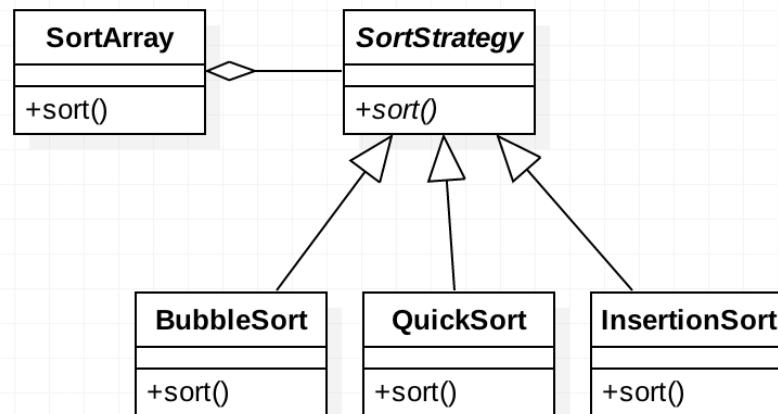


圖 13.1: Strategy Sort

### Sort

```
1  interface SortStrategy {
```

```
2     public int[] sort(int [] d);
3 }
4
5 class SortArray {
6     int [] d;
7     private SortStrategy sortStrategy;
8     public SortArray(SortStrategy s) {
9         this.sortStrategy = s;
10    }
11    public int[] doSort() {
12        return this.sortStrategy.sort(d);
13    }
14 }
15
16 class QuickSort implements SortStrategy {
17     public int[] sort(int[] d) {
18         //do ....
19         return ...
20     }
21 }
22
23 class SelectionSort implements SortStrategy {
24     public int[] sort(int[] d) {
25         //do ....
26         return ...
27     }
28 }
29
30 //main program to test
31 class StrategyExample {
32     public static void main(String[] args) {
33         SortArray context;
34         context = new SortArray(new QuickSort());
35         int [] resultA = context.doSort();
36
37         context = new SortArray(new SelectionSort());
38         int [] resultB = context.doSort();
39     }
40 }
```

## 13.2 結構與方法

### 結構

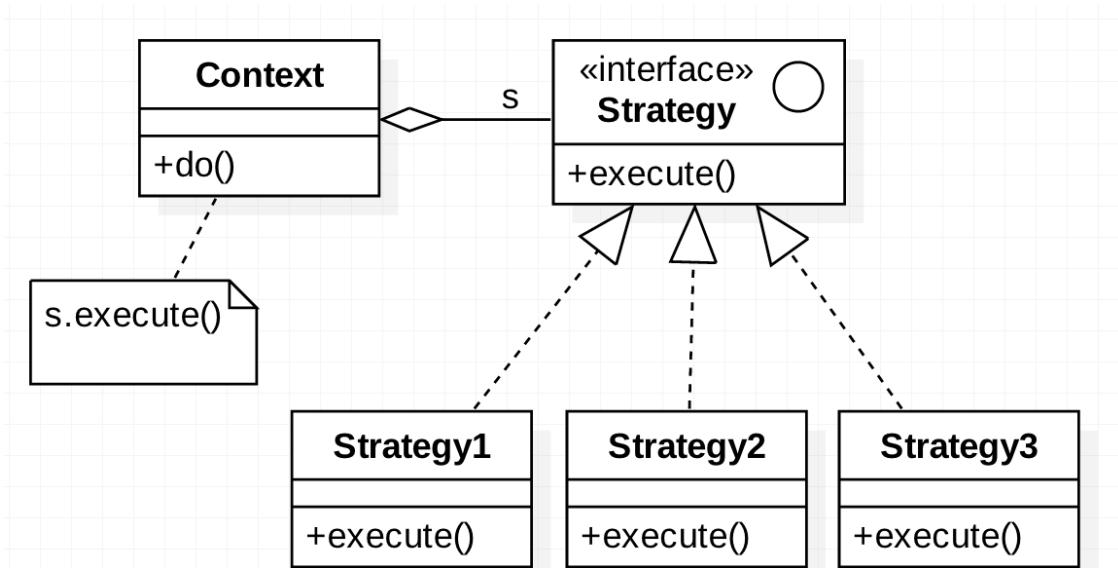


圖 13.2: Strategy Structure

### 程式樣板

程式 13.1: Strategy 程式樣板

```

1 package strategy;
2
3 class Context {
4     Strategy s;
5
6     public Context(Strategy s) {
7         this.s = s;
8     }
9     public void doIt() {
10         System.out.println("Doing something");
11         s.execute();
12     }
13 }
  
```

```

14
15  class Strategy1 implements Strategy {
16      public void execute() {
17          System.out.println("Using strategy 1");
18      }
19  }
20
21  class Strategy2 implements Strategy {
22      public void execute() {
23          System.out.println("Using strategy 2");
24      }
25  }
26
27  interface Strategy {
28      public void execute();
29  }
30
31  public class StrategyTemplate {
32
33      public static void main(String[] args) {
34          Strategy s1 = new Strategy1();
35          Context context = new Context(s1);
36          context.doIt();
37      }
38  }

```

[\[Get the code\]](#)

執行結果如下：

<p style="margin: 0;">Doing something</p> <p style="margin: 0;">Using strategy 1</p>
--

## 優點

- 消除大量的 if-else 等判斷句。過去我們可能用 if 來改變要使用的演算法。採用策略樣式，我們透過多型來達到此目的，避免過多的判斷句。
- 演算法的動態抽換。

## 缺點

- 此設計會造成比較多的類別。過去用 if 造成一個很大的方法，用 Strategy 可以避免這個大方法，但類別就會多一些。
- 所有的演算法必須符合相同的介面。因為有一個抽象的策略類別被繼承，大家都要實作相同的介面。

## 13.3 範例

### LayoutManager

Java 的 GUI 容器物件也是利用策略設計樣式來改變它的排版的。

```

1   Frame f = new Frame();
2   f.setLayout(new FlowLayout());
3   f.add(new Button("Press"));
4
5   f.setLayout(new BorderLayout());
6   ...

```

上述的 FlowLayout 就是一個 Strategy, 其他的 layout 例如 BorderLayout, GridLayout 等也都是具體的策略類別。

### Validator/Verifier

驗證器。各種不同的輸入需要做不同的驗證，我們可以把驗證器獨立於輸入元件，這樣輸入元件就可以客製化的設計驗證器了。例如生日格式的驗證、電話格式的驗證等都需要特別的驗證方式。

```
1 myTextField.setInputVerifier(new MyInputVerifier());
```

我們自己擴充 Java [InputVerifier](#) :

```

1 public class MyInputVerifier extends InputVerifier {
2     public boolean verify(JComponent input) {
3         String text = ((JTextField) input).getText();
4         try {
5             BigDecimal value = new BigDecimal(text);
6             return (value.scale() <= Math.abs(2));

```

```

7         } catch (NumberFormatException e) {
8             return false;
9         }
10    }
11 }
```

## 13.4 結語

Strategy 是換骨，Decorator 是換皮

## 13.5 練習

### 選擇/簡答

**Ex 1:** Strategy 設計樣式是把策略

- (a) 延遲到子類別決定
- (b) 委託給另一個物件
- (c) 包裝成一個複合物件
- (d) 限制只能產生一份演算法

**Ex 2:** 在 Strategy 中，若我們要擴充一個新的演算法

- (a) 新增一個 Strategy 介面
- (b) 新增一個實踐 Strategy 介面的類別
- (c) 在方法中新增一個 Strategy 參數
- (d) 宣告一個 static 方法

### 設計

**Ex 3:** 線上考試系統，選擇題的出題方式可以分為 (1) 按照原來項目順序出題 (2) 隨機把項目打亂出題。請問是否適合用 Strategy 設計？為什麼？該如何設計？

**Ex 4:** (Lab) 象棋遊戲 24.3 中，象棋的位置是在象棋生成的時候透過建構子生成的，這樣的方式似乎不太有彈性。我們希望能夠有一個彈性的設定位置策略，請改寫此程式，可以利用 Strategy 樣式。

**Ex 5:** 象棋系統中，吃子的行為可以封裝成為一個 Strategy 嗎？為什麼？還有其他的可能會用到 Strategy 嗎？試討論之。

**Ex 6:** class GradeBook 需要排序，如何應用 Strategy 讓排序演算法靈活變更？

```
1  class GradeBoook {  
2      ?  
3  }  
4  
5  interface ?  
6  
7  class ?
```

**Ex 7:** 擴充 Java InputVerifier 設計一個台灣身分證的 SSNVerifier。並應用這個 Verifier 在一個簡單應用程式。

# Chapter 14

## 神行百變：Decorator



## 14.1 目的與動機

可以動態的為一個物件加上功能（責任）。Decorator 提供一種有彈性的方法來透過繼承來擴充方法。

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

### 動機

假設我們要做一個文字視窗 (TextView)，並且提供各種不同的邊框 (border) 與捲軸 (scroll bar) 作為選擇。邊框的型態有：一般型 (Plain)、3D 型或花俏型 (Fancy)，捲軸的型態有：無捲軸、水平型 (Horizontal)、垂直型 (Vertical) 與水平垂直型。因此排列組合共有  $3 \times 4 = 12$  種型別的文字視窗。



圖 14.1: 由框和捲軸而成的 TextView

**Solution 1** 為了提供各種可能的 TextView，我們必須建立 12 種 TextView 的子類別。這種方法不但繁瑣，無法提供動態的物件生成，甚至連命名都很困難。

---

TextView-Plain	TextView-Plain-Vertical
TextView-Plain-Horizontal	TextView-Plain-Vertical-Horizontal
TextView-3D	TextView-3D-Vertical
TextView-3D-Horizontal	TextView-3D-Vertical-Horizontal
TextView-Fancy	TextView-Fancy-Vertical
TextView-Fancy-Horizontal	TextView-Fancy-Vertical-Horizontal

---

**Solution 2: Strategy 樣式** 我們可以透過 Strategy 樣式來解決這個問題，在 TextView 建立的時候帶入兩個參數，透過參數的組合來形成各種不同的 TextView。程式碼如下：

程式 14.1: 採用策略設計樣式的 TextView

```

1  public class TextView {
2      private Border border;
3      private Scrollbar sb;
4      public TextView(Border border, Scrollbar sb) {
5          this.border = border; this.sb = sb;
6      }
7      public void draw() {
8          border.draw();
9          sb.draw();
10         // Code to draw the TextView object itself.
11     }
12 }
```

此方法的缺點是缺乏彈性，如果我們在增加新的維度（除了 border, scrollbar 以外的維度），勢必要修該 TextView 的程式碼。

**Solution 3: Decorator 樣式** 如果我們使用 Decorator 設計樣式一切就會變的容易許多：對 TextView 而言，是否增加 Border 的功能或捲軸的功能都可以隨意增減，就像是裝飾品一般。新的架構如圖 14.2：

注意我們將各種 Border 與 Scrollbar 視為一種 Decorator，而每一個 Decorator 可包含一個以上的 Component，如 BorderDecorator 可能可以包含有 3D Border、Fancy Border 和 Plain Border 等個別裝飾品物件，而 ScrollDecorator 可以包含有垂直、水平的 Scroll Bar。這樣的架構方式可以讓動態生成的搭配裝飾更多樣性。讓我們來看 Border Decorator 中 PlainBorder 的程式片段：

程式 14.2: 採用 Decorator 樣式

```

1  abstract class Decorator extends AbstractTextView {
```

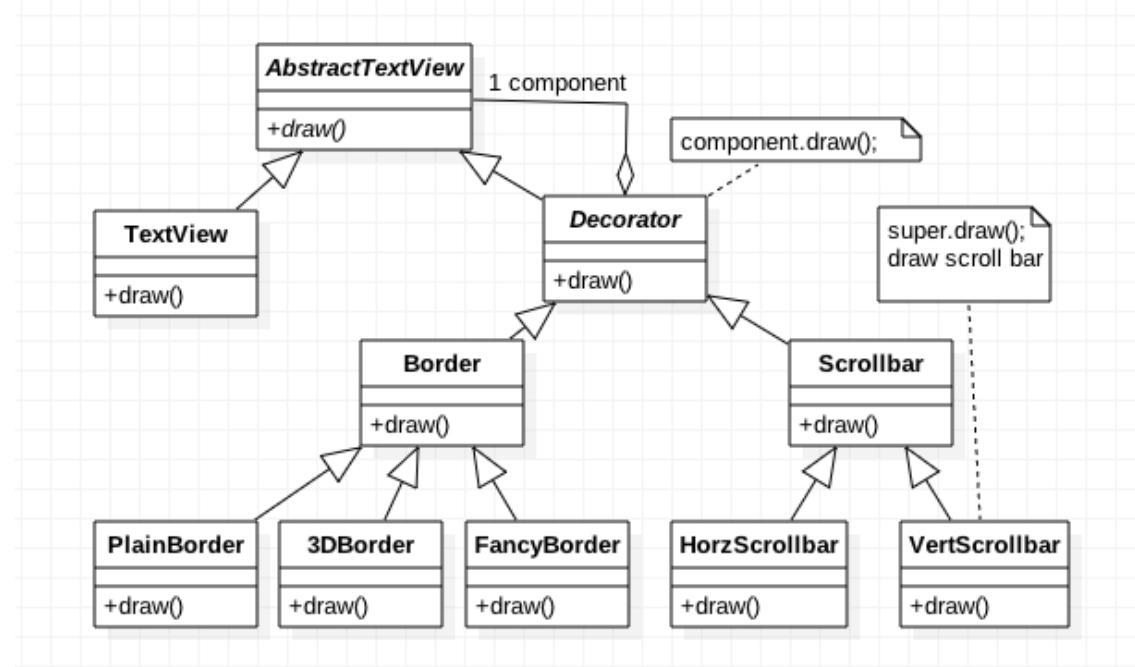


圖 14.2: 使用 Decorator 來實作 TextView

```

2     AbstractTextView tv;
3     // decorator 一定會有一個主要物件
4     public Decorator(AbstractTextView tv) {
5         this.tv = tv
6     }
7     public void draw() {
8         tv.draw();
9     }
10 }
11
12 abstract class BorderDecorator extends Decorator {
13     public BorderDecorator(AbstractTextView c) {
14         super(c);
15     }
16     public void draw() {
17         super.draw(); //先讓 component 繪出主要功能
18         //做一些 border 的準備
19     }
20 }
21
  
```

```

22  class PlainBorder extends BorderDecorator{
23      public PlainBorder (AbstractTextView c) {
24          super(c);
25      }
26      public void draw ( ) {
27          super.draw( );
28          //以下繪出一般型 (Plain) 邊框
29      }
30  }
31
32  public class Client {
33      public static void main( String[] args ) {
34          TextView data = new TextView();
35          AbstractTextView borderData = new FancyBorder(data);
36          AbstractTextView scrolledData = new VertScrollbar(data)
37          ;
38          AbstractTextView borderAndScrolledData = new
39          HorzScrollbar(borderData);
}

```

PlainBorder 的建構子將傳入一個 AbstractTextView 的變數，透過 super(c) 來設定其所包含的元件。在 draw() 時呼叫 super.draw() 會讓所包含的 textView 先做它的 draw() 再執行 PlainBorder 自身的繪圖。因此，當我們想要建立一個 PlainBorder 的 TextView 只要執行以下的命令：

```

1  TextView tv = new TextView ( );
2  PlainBorder plainTextView = new PlainBorder (tv) ;

```

從「裝飾品」的角度來看，PlainBorder 裝飾在 TextView 之上，當我們需求 PlainTV 繪圖時，它會先要求 tv 繪出基本的文字視窗，然後再將邊線繪上。如果我們要繪一個有邊框又有垂直捲軸的文字視窗呢？我們只要在 PlainTextView 上再點綴上一個垂直捲軸即可：

```

VerticalScrollBar verticalPlainTextView = new VerticalScrollBar
(plainTextView) ;

```

VerticalScrollBar 的建構子為 VerticalScrollBar(AbstractTextView c)，所以 PlainTV 可以順利的傳入 VerticalScrollBar 的建構子中。當 verticalPlainTextView 繪圖時，其過程：其中(1) 繪出一個文字視窗，(2) 繪出一個邊框，(3) 繪出垂直和水平的 Scroll Bar。看完此例，各位應了解 Decorator 的用途了。我們接著來看 Decorator 的基本結構吧。

## 14.2 結構與方法

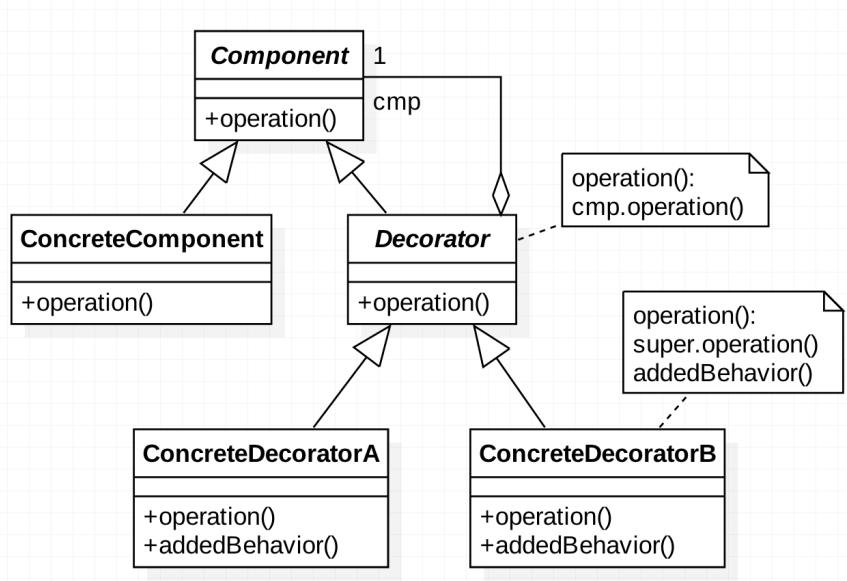


圖 14.3: Decorator

### 參與者

- Component：系統內元件的配置管理。
- ConcreteComponet：系統內主要的功能元件。
- Decorator：功能元件的裝飾品管理，管理每一個裝飾品功能物件，使它們能夠動態生成的方式加入到系統中。
- ConcreteDecoratorA：實際生成的裝飾品功能物件。

### 優點

透過裝飾品設計樣式，可以讓功能需求像是裝飾品一樣動態的加到系統中，而不會影響到系統本體結構，不會讓架構變的複雜，透過功能包裝成物件的方式可以讓架構更清晰，所屬物件負責工作更明確，進而提昇軟體品質。

### 程式樣板

## 程式 14.3: Decorator 程式樣板

```

1  package decorator;
2
3  abstract class Component {
4      abstract void op();
5  }
6
7  class ConcreteComponent extends Component {
8      void op() {
9          System.out.println("Basic_behavior");
10     }
11 }
12
13 abstract class Decorator extends Component {
14     Component c;
15
16     public Decorator(Component c) {
17         this.c = c;
18     }
19
20     void op() {
21         c.op();
22     }
23 }
24
25 class ConcreteDecorator1 extends Decorator {
26     public ConcreteDecorator1(Component c) {
27         super(c);
28     }
29
30     void op() {
31         super.op();
32         addedBehavior();
33     }
34
35     void addedBehavior() {
36         System.out.println("Added_behavior_1");
37     }
38 }
```

```

39
40  class ConcreteDecorator2 extends Decorator {
41      public ConcreteDecorator2(Component c) {
42          super(c);
43      }
44
45      void op() {
46          super.op();
47          addedBehavior();
48      }
49
50      void addedBehavior() {
51          System.out.println("Added behavior 2");
52      }
53  }
54
55  public class DecoratorTemplate {
56      public static void main(String[] args) {
57          Component cc = new ConcreteComponent();
58          cc.op();
59          Component c1 = new ConcreteDecorator1(new
60              ConcreteDecorator2(cc));
61          c1.op();
62      }
63  }

```

[\[Get the code\]](#)

執行結果如下：

Basic behavior Basic behavior Added behavior 2 Added behavior 1
--

如果我們有 Decorator 子類別 D1, D2, D3, D4, decorator 內的功能為 op()，假設每個 op() 都是先執行 super.op(), 再執行自身的行為（分別為 f1-f4），ConcreteComponent 的類別為 CC。則

```

1     Component d = new D2(new D1(new D3(new D4(new CC())))))

```

的生成方式，執行的順序為：CC.op(), f4, f3, f1, f2。

## 14.3 範例

### Java I/O

熟悉 JAVA I/O 的讀者對 Decorator 應該有似曾相識的感覺吧！我們先看 Input Stream 的類別圖（圖 14.4）：

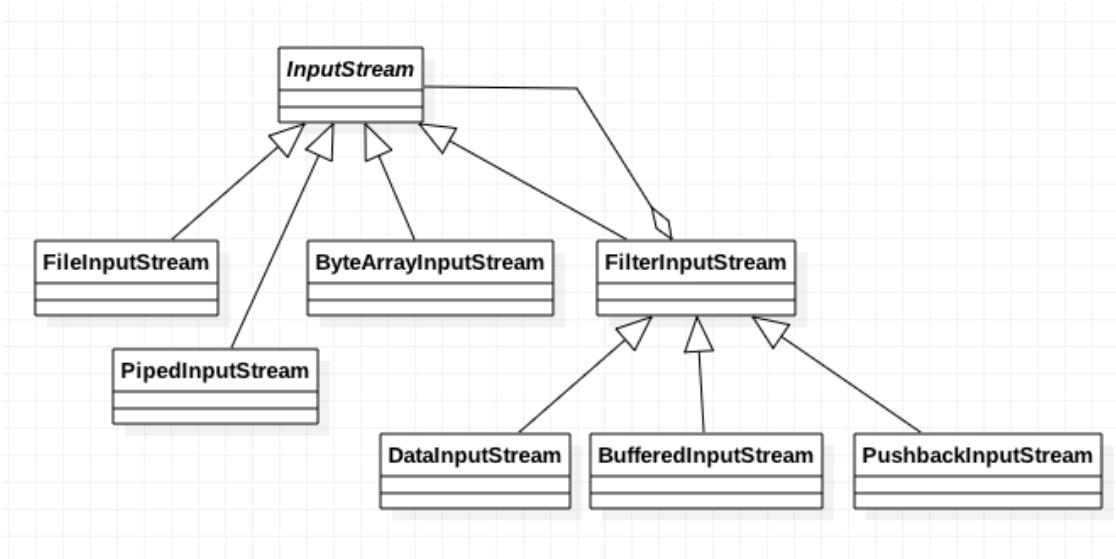


圖 14.4: Java IO- Using Decorator

在 JavaI/O 中 `InputStream` 和 `OutPutStream` 都有 `BufferredStream`、`DataStream`、`PushbackStream` 的功能需求：

- `BufferredInputStream`：支援 buffering 的功能
- `DataInputStream`：I/O 支援 JAVA 基本型態
- `PushbackInputStream`：支援 undo 的功能

但是如果靜態的生成方式必須作出  $2 * 2 * 2 = 8$  種不同搭配的類型，這樣讓系統架構變得很複雜龐大，修改維護上更是麻煩。

因此 JavaI/O 的結構使用裝飾品樣式去解決這樣的問題，`FilterInputStream` 的角色就相當於 Decorator，而 `BufferredInputStream`，`DataInputStream`，`PushbackStream` 等為 `Input-Stream` 類別的 ConcreteDecorator：

下列說明 InputStream 的用法：

```

1 //建立一個基本的檔案輸入通道
2 FileInputStream fin = new FileInputStream ("decorator.txt") ;
3
4 //建立一個支援 buffering 的檔案輸入通道 - 擴充 fin
5 BufferedInputStream bfin = new BufferedStream (fin) ;
6
7 //建立一個支援 buffering 及 undo 的檔案輸入通道 - 擴充 bfin
8 PushbackInputStream pbfin = new PushBackInputStream (bfin) ;

```

如果一開始就想建立一個支援 buffering 及 undo 的檔案輸入通道，可以簡化的寫：

```

PushBackInputStream in = new PushBackInputStream (new
    BufferedInputStream (new FileInputStream ("decorator.
        txt")));

```

注意：當骨幹建立以後，各點綴品的建立並沒有先作之別。也就是說，你可以先建立 pushback 的功能再建立 buffering 的功能。因此上式可以改寫成：

```

in = new BufferedInputStream (new PushBackInputStream (new
    FileInputStream ("decorator.txt")));

```

## 比較

**Strategy 換骨，Decorator 換皮**

**Decorator 像洋蔥**

**Decorator 像聖誕樹**

## 14.4 練習

**選擇/簡答**

**Ex 1:** Java 的 FileStream 用了 Decorator 設計樣式，其中 FilterStream 相當於此樣式中的

- (a) Client

- (b) Decorator
- (c) ConcreteDecorator
- (d) Component
- (e) ConcreteComponent

**Ex 2:** 關於 Decorator pattern, 下列何者為錯

- (a) Decorator 可以包含一個 Decorator 物件
- (b) Decorator 和 ConcreteComponent 有部分共同的方法，宣告在 Component 中
- (c) Decorator 和 ConcreteComponent 都可以包含 Component

**Ex 3:** 請說明 Strategy 和 Decorator 設計樣式的異同。

**Ex 4:** 有一個物件 A 可以從兩方面（或更多）去擴充，分別為 X, Y。假設 X 方面可以有  $X_1, X_2$  兩種選項，Y 有  $Y_1, Y_2, Y_3$  三種選項。

- 若以 Decorator 設計樣式來設計，該如何設計？請寫出一個簡易的 demo 程式
- 若改以 Strategy 設計樣式來設計，請寫出一個簡易的 demo 程式
- 以繼承的方法來設計，需要設計多少類別？

## 設計

**Ex 5:** 聖誕樹 (ChrismasTree) 上面有許多的裝飾品，包含鈴鐺 (Bell)，糖果 (Candy)，與禮物 (Gift)，請用 Decorator 樣式設計之。所有的聖誕樹都會支援 sing() 的方法：

- 聖誕樹：I am a Chrismas tree
- 有鈴鐺的聖誕樹：I have a bell, I am a Chrismas tree
- 有糖果和鈴鐺的聖誕樹：I have a candy, I have a bell, I am a Chrismas tree

依此類推。請寫出完整可以執行的程式。

**Ex 6:** Writer 可作輸出，FilterWriter 是一個 Decorator 的物件。設計以下的 Filter:

- LowerCaseFilter: 每個英文字都改成小寫
- UpperCaseFilter: 每個英文字都改成大寫
- CommaFilter: 遇到數字就加上千分號
- CountFilter: 在每行字後面加上單字的個數

```

1   Writer w = new BufferedWriter(new FileWriter("test.txt"))
2   ;
3   Writer w2 = new LowerCaseFilter(w);
4   w2.write("THIS is A Test 12309092"); //印出 this is a test
5   12309092

```

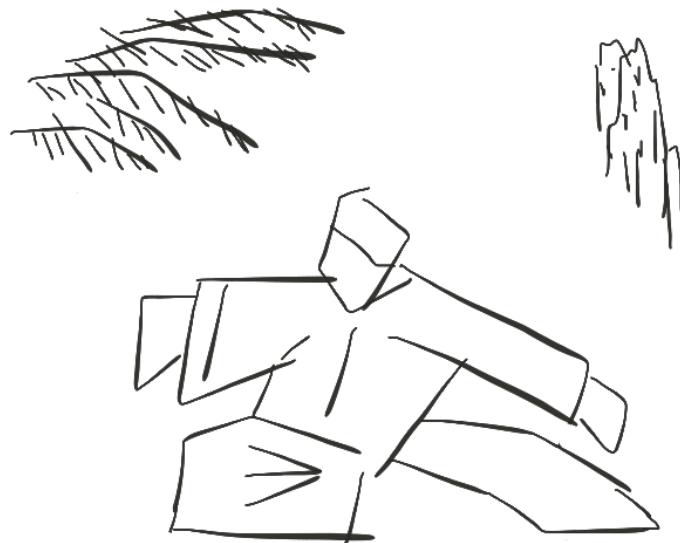
```
4   Writer w3 = new CommaFilter(w2);
5   w3.write("THIS is A Test 12309092"); //印出 this is a test
12,309,092
```

**Ex 7:** 泡咖啡了！我們有手工 (HandBlend)、深度烘胚 (DarkRoast)、低卡 Decaf、Espresso 等咖啡，而且每一種咖啡都可以加上 Milk, Mocha, Soy，當然每一個都是額外需要加費的。請用 Decorator 設計樣式設計之，注意 Coffee 是父類別，而我們需要 cost() 方法來回傳費用。畫出 UML 圖，寫出程式（請自己假設個別的價格）。

**Ex 8:** 象棋系統中，可否應用 Decorator 設計樣式？試說明之。

# Chapter 15

## 剛中帶柔：Template Method



## 15.1 目的與動機

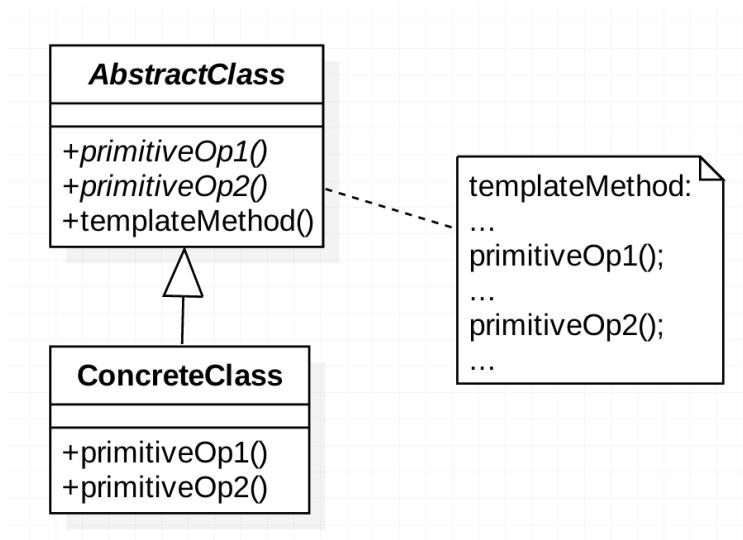
定義一個演算法的骨幹，把部分的步驟延遲到子類別來決定。

*To define the skeleton of an algorithm in an operation, deferring some steps to subclasses*

樣板樣式讓子類別重新定義一個演算法的部分細節，但不變更演算法的結構。整體演算法不變是「剛」，部分可以改是「柔」，所以說 template method 是一個剛中帶柔的方式。多半我們翻譯成樣板，或是樣板方法。

## 15.2 結構與方法

### 結構



### 參與者

- **AbstractClass:** `templateMethod()` 必須宣告為 `final`, 表示子類別不會變動整體的演算法。`primitiveOp1()`, `primitiveOp2()` 表示基礎方法，是 `templateMethod()` 會呼叫的方法，如果是強制要子類別覆蓋、重新定義的基礎方法就應該宣告為抽象。
- **ConcreteClass:** 表示真實運作的類別或系統，因為 **AbstractClass** 定義的方法 `templateMethod` 有部分的方法是抽象，在此類別會具體化這些方法。

## 應用性

- 演算法不變的部分僅做一次，將有變化的部份留給子類別來設計。
- 「一般化」的重整，達到避免程式碼重複的好處。
- 控制子類別的擴充，僅可以在固定的點做擴充。

## 優點

- Template method 是程式碼重用的基礎技巧。
- Template method 實踐 IoC (Inversion of Control; 控制倒轉) 的設計原則。

## 實作考量

- Template method 中的基礎方法 (primitive operation) 是否宣告為 `protected`? 如果有子類別呼叫的需求時可宣告為 `protected`。
- 一定要被覆蓋的基礎方法必須要宣告為抽象
- Template method 必須宣告為 `final`, 不可被子類別覆蓋。

## 範例

### Application opens documents

```

1  class Application {
2      final void openDocument ( String name ) {
3          if ( ! canOpenDocument ( name ) ) {
4              // cannot handle this document return;
5          }
6          Document doc = doCreateDocument ();
7          if ( doc ) {
8              doc . addDocument ( doc );
9              aboutToOpenDocument ( doc );
10             doc . open ();
11             doc . doRead ();
12         }
13     }
14     abstract boolean canOpenDocument ( String );

```

```

15     abstract aboutToOpenDocument( Document );
16     abstract Document doCreateDocument();
17 }
```

## Chess Game

設計一個通用的遊戲，playOneGame() 是一個 Template Method。

```

1 abstract class Game {
2     protected int playersCount;
3
4     /* A template method : */
5     final void playOneGame(int playersCount) {
6         this.playersCount = playersCount;
7         initializeGame();
8         int j = 0;
9         while (!endOfGame()) {
10            makePlay(j);
11            j = (j + 1) % playersCount;
12        }
13        printWinner();
14    }
15    abstract void initializeGame();
16    abstract void makePlay(int player);
17    abstract boolean endOfGame();
18    abstract void printWinner();
19 }
```

大富翁或是象棋遊戲都繼承 Game, 修改了部分的程式碼。

```

1 class Monopoly extends Game {
2     void initializeGame() {
3         // ...
4     }
5     void makePlay(int player) {
6         // ...
7     }
8     boolean endOfGame() {
9         // ...
10    }
```

```

11     void printWinner() {
12         // ...
13     }
14 }
15
16 class Monopoly extends Game {
17     void initializeGame() {
18         // ...
19     }
20     void makePlay(int player) {
21         // ...
22     }
23     boolean endOfGame() {
24         // ...
25     }
26     void printWinner() {
27         // ...
28     }
29 }

```

## 15.3 比較

Template method 是「框架」的基礎

## 15.4 練習

### 選擇/簡答

**Ex 1:** Template Method 的目的

- (a) 把兩個介面不相容的物件可以溝通合作
- (b) 把物件的生成延遲到子類別
- (c) 讓一個物件可以有很多的觀者者，物件變動時，其觀察者物件可以跟著變動
- (d) 定義一個演算法的架構，讓細部的做法延遲到子類別做決定

**Ex 2:** 在 Template Method 中，與 template method 通常宣告為

- (a) final

- (b) static
- (c) interface
- (d) generic type

**Ex 3:** Template Method 運用的技巧為

- (a) 把要延遲的程式碼定義為 final，讓子類別去定義
- (b) 把要延遲的程式碼包裝成方法，讓子類別去定義
- (c) 把要延遲的程式碼定義為 final，不讓子類別修改

**Ex 4:** Template method 和 factory method 有點類似

- (a) 前者會生成演算法，後者會生成物件
- (b) 前者運用多型，後者運用委託
- (c) 兩者都是延遲方法到子類別，前者延遲方法，後者延遲物件生成方法

**Ex 5:** 以下程式哪一個方法可能是 template method?

```

1   class A {
2       int a;
3       public void m1() {
4           ...
5           m2();
6           ...
7           ...
8       }
9       public abstract void m2();
10      public int m3() { return a; }
11      public final int m4() { return a*a; }
12  }
```

## 設計

**Ex 6:** 應用 Arrays.sort(Object[], Comparator) 把一群學生 (Student) 依據他們的身高做排序。  
討論：Arrays.sort 有應用到 Template 樣式嗎？有什麼好處？

**Ex 7:** 某一類遊戲的演法算法大概的邏輯是

- (a) 初始化遊戲
- (b) 等待玩家加入
- (c) 開始玩
- (d) 第三步驟一直重複，直到分出勝負

- (e) 印出勝利者
- (f) 印出遊戲資訊，例如遊戲時間

其中第 1, 3, 4 步驟在每個遊戲略有不同，由遊戲本身定義。請用 template method 定義 playGame() 方法。

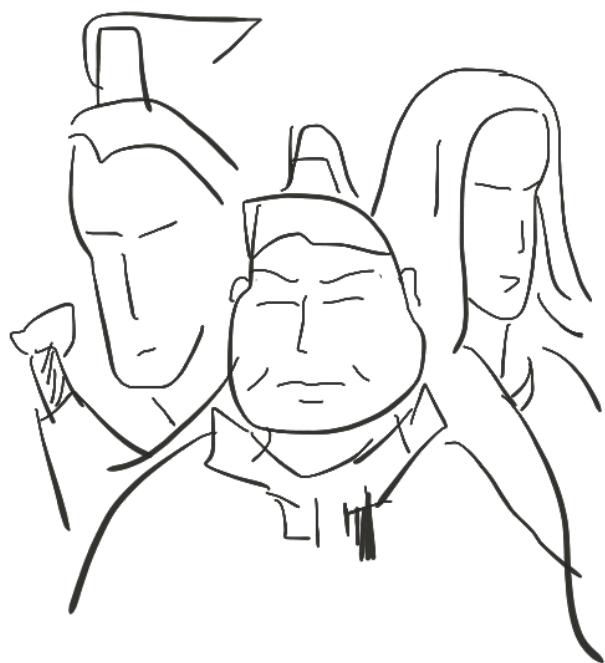
```
1  public ?1 void playGame() {  
2      ?  
3  }
```

**Ex 8:** 煮咖啡囉！咖啡沖泡法：1. 把水煮沸（boilWater）；2. 用沸水沖泡咖啡（brewCoffeeGrinds）；3. 把咖啡倒到杯子（pourInCup）；4. 加糖和奶精（addSugarAndMilk）。茶沖泡法：1. 把水煮沸（）；2. 用沸水浸泡茶葉（steerTeaBag）；3. 把茶倒到杯子（pourInCup）；4. 加檸檬（addLemon）。請利用 template method 達到重用性的設計。



# Chapter 16

## 三足鼎立：MVC



## 16.1 目的與動機

在系統架構中，將資料模組 (Model)、呈現模組 (View)、控制模組 (Controller) 三者分離，藉此降低系統內的功能邏輯和資料的耦合，使得架構上分工明確、促使開發人員專責分工，進而提昇系統的擴充性。

*Model-view-controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. [5]*

Model 與 View 的關係，和 Observer 設計樣式中 Subject 與 Observer 很類似，重點都是要把企業邏輯（或是資料主體）與呈現分離，並且當資料改變時，能夠即時的更新呈現。

問題是：誰去改變主體的狀態？

可能是使用者透過一個視窗介面輸入值變動後端的資料；可能是滑鼠的移動觸發資料的變動；可能是一個網路爬蟲在網路獵取資料經過計算後進行變動。像這樣的模組，它提供一個可以輸入的介面、過濾資料、轉換資料成可以變更資料主體的形式，在 MVC 架構中就是 Control 模組。你可以把它翻譯成控制器，控制所有對於資料主體進行變更的行為。

Controller 修改 Model, Model 通知 View

### 結構

- 資料模組 (Model)：只有包含應用程式的資料。當經由 Controller 做資料更改時，Model 會通知 View 做更動，以致於 View 能夠用已更動過的值來更換 View 它本身的呈現。在股票系統中，股票數值資料本身，例如股票的報價、序號、漲跌資料等。
- 呈現模組 (View)：負責資料的呈現。在股票系統中，資料呈現於網頁、手機、電視都是不同的呈現模組。
- 控制模組 (Controller)：定義使用者介面如何回應用戶的輸入事件，控制模組負責提供改變資料模組中的資料。在股票系統中，經由不同股票交易所，網站交易等輸入資料的動作。

MVC 模式的好處：

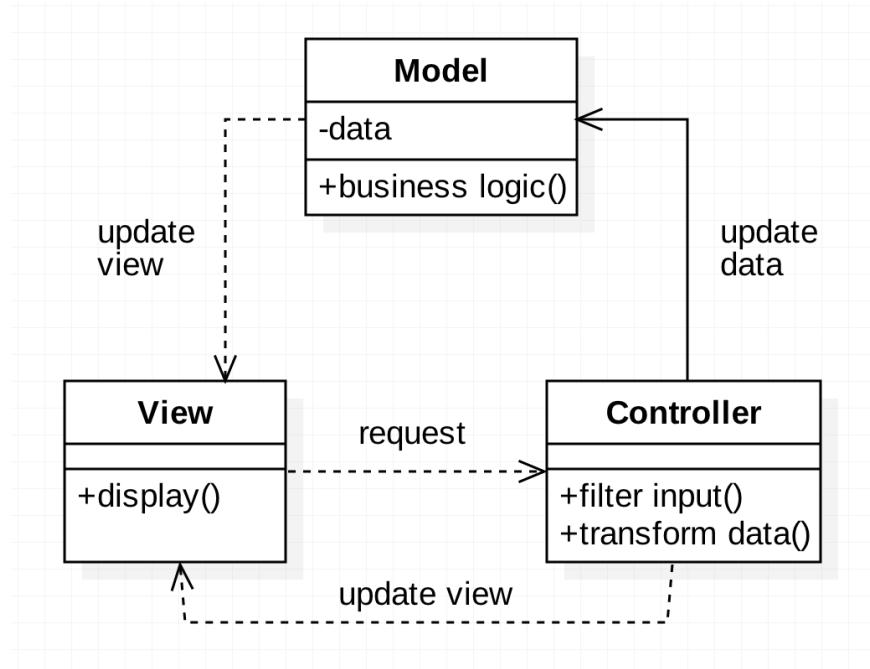


圖 16.1: MVC

- 藉由 Controller 協調，View 不會直接對 Model 作出任何變更，View 只負責呈現邏輯，即根據 Model 資料呈現結果，相關操作對應的企業邏輯（Business logic）不在 View 中。
- Controller 負責企業邏輯中請求的收集與驗證，更複雜的邏輯轉交 Model 處理，這使得 Controller 維持簡單，Model 也具有較高的重用性。Controller 也負責調配對應的 View，所以 Model 不會與 View 耦合。
- Model 不會知道或決定 View 實際呈現方式，因此 View 可以獨立於 Model，自由變化。
- Model 狀態有變化時，曾向 Model 註冊過的多個 View，都可獲得通知而得到同步更新效果。

在這樣的定義下，MVC 與 Observer 有很大的關係，MVC 中 Model 和 View 的關係，就相當於 Observer 樣式中，Subject 和 Observer 的關係。

### 16.1.1 程式樣板

```

1   Model m = new Model();
2   View1 v1 = new View1();
3   View2 v2 = new View2();
  
```

```

4
5     m.addView(v1);
6     m.addView(v2);
7
8     Controller c = new Controller(m);

```

也有可能是這樣

```

1     Model m = new Model();
2     View1 v1 = new View1(m);
3     View2 v2 = new View2(m);
4
5     Controller c = new Controller();
6     c.setM(m);

```

## 16.2 範例

我們先從這個簡單的介面程式來看看什麼是 MVC。

### Solution 1：無模組化

```

1  public class CounterGui extends JFrame {
2      //counter 是 MODEL
3      private int counter = 0;
4
5      //tf 是 VIEW
6      private TextField tf = new TextField(10);
7
8      public CounterGui(String title) {
9          super(title);
10         Panel tfPanel = new Panel();
11         tf.setText("0");
12         tfPanel.add(tf);
13         add("North", tfPanel);
14         Panel buttonPanel = new Panel();
15
16         //CONTROLLER
17         Button incButton = new Button("Increment");

```

```

18     incButton.addActionListener(new ActionListener() {
19         public void actionPerformed(ActionEvent e) {
20             counter++;
21             tf.setText(counter + "");
22         }
23     });
24 );
25 buttonPanel.add(incButton);
26
27 Button decButton = new Button("Decrement");
28 decButton.addActionListener(new ActionListener() {
29     public void actionPerformed(ActionEvent e) {
30         counter--;
31         tf.setText(counter + "");
32     }
33 }
34 );
35
36 buttonPanel.add(decButton);
37
38 add("South", buttonPanel);
39 }
40
41 public static void main(String[] argv) {
42     CounterGui cg = new CounterGui("CounterGui");
43     cg.setSize(300, 100);
44     cg.setVisible(true);
45 }
46 }
```

在這個例子中，Model (counter)、View (tf)、Controller (incButton) 都放在同一個類別內，並沒有分離。以下的作法，我們將 View 與 Model 分離。

## Solution 2：View 與 Model 分離

首先我們把 Counter 宣告成一個類別，它扮演 Model 的角色。Model 有自己要存儲的資料，和存取的介面。Model 獨立以後，很可能有多的 View 會相依於這個 Model，如果沒有處理好，就會出現資料不一致的狀況。

程式 16.1: 應用 Observer 的 MVC

```
1 package mvc.noobserver;
2
3 import java.awt.TextField;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.util.Observable;
7 import java.util.Observer;
8 import javax.swing.JButton;
9 import javax.swing.JFrame;
10
11 /* MODEL */
12 class Counter {
13     int counter = 0;
14
15     public void incCounter() {
16         counter++;
17     }
18
19     public int getCounter() {
20         return counter;
21     }
22 }
23
24 /* VIEW */
25 class CounterView extends JFrame {
26     static int id = 0; //just for setting location
27     private TextField tf = new TextField(10);
28
29     public CounterView(String title) {
30         super(title);
31         tf.setText("0");
32         add(tf);
33         this.setSize(200, 100);
34         this.setLocation(100 + (id++) * 200, 100);
35         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
36     }
37
38     public void update(int v) {
39         tf.setText(String.valueOf(v));
40     }
}
```

```
41     }
42
43     /* CONTROLLER */
44     class CounterController extends JFrame implements
45         ActionListener {
46         static int id = 0; //just for setting location
47         private JButton inc = new JButton("INC");
48         Counter c;
49         CounterView v;
50
51         public CounterController(Counter c, CounterView v) {
52             this.c = c;
53             inc.addActionListener(this);
54             this.v = v;
55             add(inc);
56             this.setTitle("Controller" + id);
57             this.setSize(200, 100);
58             this.setLocation(100 + (id++) * 200, 200);
59             this.setDefaultCloseOperation(EXIT_ON_CLOSE);
60         }
61
62         public void actionPerformed(ActionEvent arg0) {
63             c.incCounter();
64             v.update(c.getCounter());
65         }
66
67     public class NoObserverDemo {
68
69         public static void main(String[] args) {
70             Counter c = new Counter();
71
72             CounterView view1 = new CounterView("View0");
73             view1.setVisible(true);
74             CounterView view2 = new CounterView("View1");
75             view2.setVisible(true);
76
77             CounterController controller1 = new CounterController(c
78                 , view1);
79             controller1.setVisible(true);
```

```

79         CounterController controller2 = new CounterController(c
80             , view2);
81         controller2.setVisible(true);
82     }

```

[\[Get the code\]](#)

注意上述的做法「並不是由 MODEL 去通知 VIEW 狀態的改變」，而是由 Control 直接去改變，這會產生不一致的情況。當我們在 cv1 上一直增加 counter 的值時，cv2 的介面並不會反應。反之亦然。這個設計，雖然已經分離了 MVC，但其架構沒有使用 Observer 樣式，造成了一些問題。

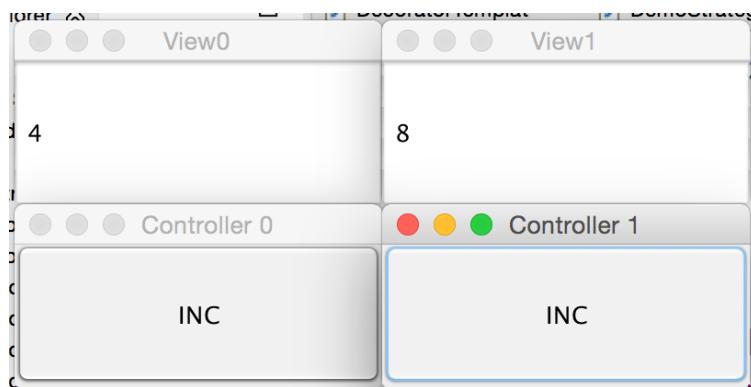


圖 16.2: 不一致的 View

我們可以應用 Observer 來改善這個問題：

### Solution 3：使用 Observer

程式 16.2: 應用 Observer 的 MVC

```

1 package mvc;
2
3 import java.awt.TextField;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.util.Observable;
7 import java.util.Observer;
8 import javax.swing.JButton;
9 import javax.swing.JFrame;

```

```
10
11  /* MODEL */
12  class Counter extends Observable {
13      int counter = 0;
14
15      public void incCounter() {
16          counter++;
17          setChanged();
18          notifyObservers(new Integer(counter));
19      }
20
21      public int getCounter() {
22          return counter;
23      }
24  }
25
26  /* VIEW */
27  class CounterView extends JFrame implements Observer {
28      static int id = 0; //just for setting location
29      private TextField tf = new TextField(10);
30
31      public CounterView(String title) {
32          super(title);
33          tf.setText("0");
34          add(tf);
35          this.setSize(200, 100);
36          this.setLocation(100 + (id++) * 200, 100);
37          this.setDefaultCloseOperation(EXIT_ON_CLOSE);
38      }
39
40      public void update(Observable arg0, Object arg1) {
41          int c = ((Integer) arg1).intValue();
42          tf.setText(String.valueOf(c));
43      }
44  }
45
46  /* CONTROLLER */
47  class CounterController extends JFrame implements
48      ActionListener {  
    static int id = 0; //just for setting location
```

```
49     private JButton inc = new JButton("INC");
50     Counter c;
51
52     public CounterController(Counter c) {
53         this.c = c;
54         inc.addActionListener(this);
55
56         add(inc);
57         this.setTitle("Controller" + id);
58         this.setSize(200, 100);
59         this.setLocation(100 + (id++) * 200, 200);
60         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
61     }
62
63     public void actionPerformed(ActionEvent arg0) {
64         c.incCounter();
65     }
66 }
67
68 public class MVCByObserver {
69     public static void main(String[] args) {
70         Counter c = new Counter();
71         CounterView view1 = new CounterView("View0");
72         view1.setVisible(true);
73         CounterView view2 = new CounterView("View1");
74         view2.setVisible(true);
75         c.addObserver(view1); //add view
76         c.addObserver(view2); //add view
77         CounterController controller1 = new CounterController(c
78             );
79         controller1.setVisible(true);
80         CounterController controller2 = new CounterController(c
81             );
82         controller2.setVisible(true);
83     }
84 }
```

[Get the code]

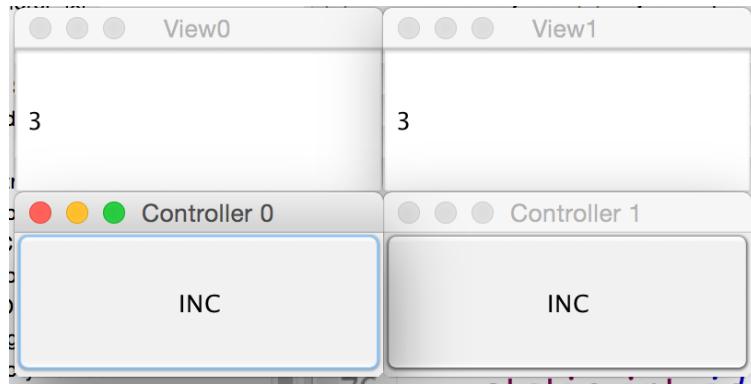


圖 16.3: 應用 Observer 的 MVC 範例

### 16.3 Web MVC

雖然我們看了應用 Observer 的 MVC 架構，但現實框架並不是全部的 MVC 都採用了 Observer 的架構。實際上 MVC 模式本身並沒有明確定義，因此許多設計僅採納 MVC 最大特徵「將系統畫分為 Model、View 和 Controller 三個部份」，並將三個部份的互動流程因實際應用場合進行調整，其中最為人熟悉的，就是 Web 應用程式經常採用的 Model 2 [4]，又稱 Web MVC，或由於過於流行所以大家就直接稱 MVC 了。

Web 在先天上很自然的會被畫分為 Model、View 和 Controller 三個部份，View 通常是客戶端的瀏覽器依所取得的 HTML 所繪製的介面，Controller 位於 Web 伺服器用來控制用戶所傳的命令；與資料庫互動的則是 Model；傳統 MVC 模式實作的桌面應用程式，MVC 三個部份通常位於同臺電腦，或者是透過持續網路連線互動。

然而，對於基於 HTTP 的 Web 應用程式來說，由於 HTTP 基於請求/回應（Request/Response）模型，沒有請求就不會有回應，因此無法實現傳統 MVC 中 Model 主動通知 View 的流程，如果 View 想根據 Model 最新狀態來呈現畫面，基本作法之一，是透過持續性地主動詢問（Poll）伺服端 Model 來達成。

基於 HTTP 限制而調整傳統 MVC 互動流程，得到的模式稱為 Model 2，除了 Model 無法主動通知 View 外，基本上它的優點與傳統 MVC 優點是相同的。Model 2 概念最早出現在 1998 年 JSP 規格中，規格書中提及兩個模型：

- **Model 1**。Model 1 中通常會將請求發給某個網頁動態元件（例如 JSP 或 Servlet），由它處理請求參數收集、驗證、企業處理以及畫面回應，有些企業邏輯亦可能會封裝成 Model（例如 JavaBean），然而前端元件亦要處理請求參數收集、驗證與畫面回應，程式碼與畫面設計勢必形成程式碼的混亂，也因而 Model 1 通常應用在規模較小、任務簡單的 Web 應用程式。

- Model 2。將架構分為 MVC 三大模組。(1) **Controller** 的職責是接受請求、驗證請求、判斷要轉發請求給哪個模型、判斷要轉發請求給哪個畫面。(2) **Model** 的職責是：保存應用程式狀態、執行應用程式企業邏輯。(3) **View** 的職責是：提取模型狀態、執行呈現邏輯、組織回應畫面。

### Model 2 的流程

1. 瀏覽器送出請求，由扮演 Controller 的 Servlet 程式接收。
2. Servlet 依據請求產生適當的 Model 物件，回傳運算結果。
3. 控制權由 Servlet forward 給 JSP 呈現模組。
4. JSP 程式向 Model 請求值，透過 JSP（與 HTML）語法作適當呈現。
5. 將畫面回應給 client 瀏覽器。

Model 2 中通常根據 URL 模式 (Pattern) 決定哪個 Controller 要接收 View 發送的請求參數，實際上在傳統 MVC 中，Controller 名稱與方法簽署 (Method signature) 就相當於 Model 2 的 URL 模式，而方法的參數名稱與實際接收的引數，就相當 HTTP 請求中的參數名稱與值。

HelloServlet.java 是程式的 Controller

```

1  import java.io.IOException;
2  import javax.servlet.ServletException;
3  import javax.servlet.annotation.WebServlet;
4  import javax.servlet.http.HttpServlet;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.http.HttpServletResponse;
7
8  @WebServlet(name="Hello", urlPatterns={"/hello.do"})
9  public class HelloServlet extends HttpServlet {
10     private Hello hello = new Hello();
11
12     @Override
13     protected void doGet(HttpServletRequest req,
14                           HttpServletResponse resp)
15             throws ServletException, IOException {
16         String name = req.getParameter("user");
17         String message = hello.doHello(name);
18         req.setAttribute("message", message);
19         req.getRequestDispatcher("hello.jsp").forward(req, resp);
20     }

```

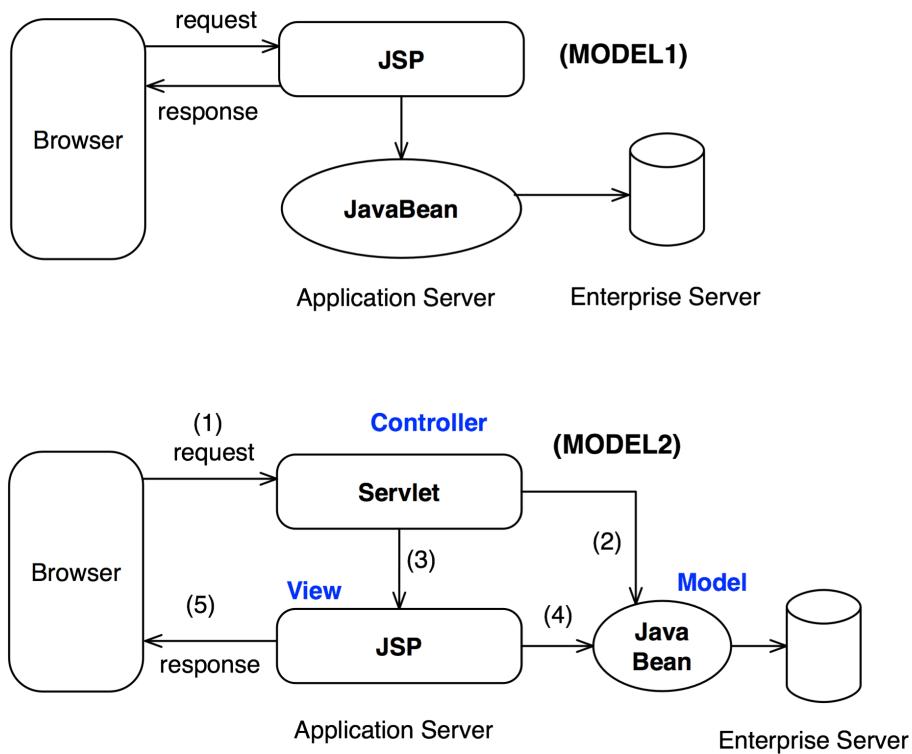


圖 16.4: JSP Model 1 and Model 2

```

19      }
20  }
```

上述程式可以看到 HelloServlet 扮演 Controller 的角色，第 10 行建立一個 Model 物件 Hello，16 行向此物件請求執行特定的企業邏輯（doHelp()），執行後的結果封裝在 req 中，透過 forward 傳到介面程式 hello.jsp。

Hello.java 是資料的一部分，以下我們用一個 HashMap 模擬一個資料庫。

```

1  import java.util.*;
2
3  public class Hello {
4      private Map<String, String> messages;
5
6      public String doHello(String user) {
7          String message = messages.get(user);
8          return message + ", " + user + "!";
}
```

```

9         }
10
11     public Hello() {
12         messages = new HashMap<String, String>();
13         messages.put("Nick", "Hello");
14         messages.put("Mary", "Welcome");
15         messages.put("John", "Hi");
16     }
17 }

```

依上，如果 user 請求參數是 Nick 就會取得 “Hello” 字串，如果是 Mary，就會取得 “Welcome” 字串，在取得訊息之後，先前擔任控制器的 Servlet 會轉發給畫面。Hello.jsp:

```

1 <html>
2   <head>
3     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
4     <title>\${param.user}</title>
5   </head>
6   <body>
7     <h1>\${message}</h1>
8   </body>
9 </html>

```

可以看得出來，此模組主要是 HTML 的呈現，加上適當的資料呈現。

## 16.4 練習

### 選擇/簡答

**Ex 1:** Observer 樣式中的 Subject 和 Observer 分別是 MVC 中的？

- (a) Controller-Model
- (b) Model-Controller
- (c) Model-View
- (d) View-Model

**Ex 2:** 關於 MVC，何者錯誤

- (a) Model 不應該相依於 Controller 與 View

- (b) View 會被 Model 通知狀態的改變，也可能會去讀取 Model 的狀態
- (c) Controller 過濾及轉換訊息，把修改的指令給 Model，讓 Model 變更狀態
- (d) Model 雖不可相依於 View, 但可相依於 Controller

**Ex 3:** 畫出 MVC 架構，並寫出樣板程式

**Ex 4:** 畫出 Counter v3 的 UML, 說明何者為 M, V, C

## 設計

**Ex 5:** (Lab) ChessGame v4 中 (see 24.4)，透過 ChessGame 呼叫 ChessBoard 改變棋子的狀態，並呈現在 ChessBoard 上

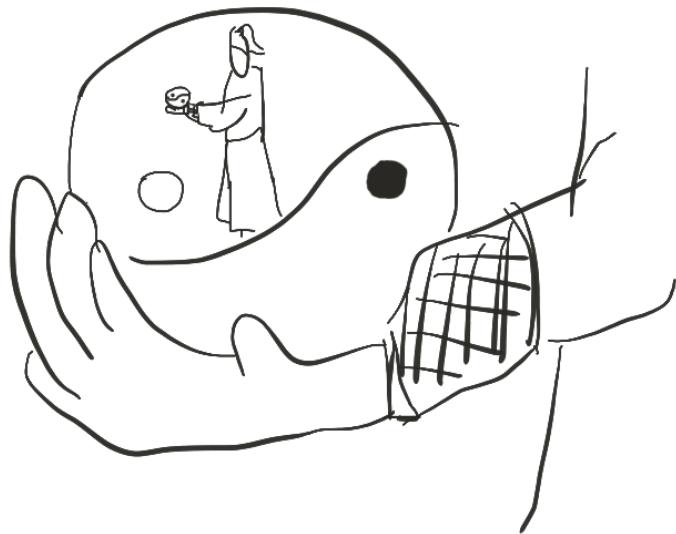
- (a) 下載此程式後，透過 StarUML 的 Reverse Code 自動產生 UML 的 Model。整理此 Model 提昇其可讀性。觀看程式碼與 UML model 了解此程式。
- (b) 了解後繪製 Sequence diagram, 了解程式的動態執行程序。
- (c) 一個棋盤應該只有一個 Chess 被選擇 (SELECTED)，目前的程式並不符合這樣的限制，請修改。
- (d) 同上，棋子的選擇應該和玩家有關係（黑邊/紅邊），請加上此限制。
- (e) ChessGame, Chess, ChessBoard 各自扮演 MVC 的哪個角色？
- (f) 如果我們有多個呈現介面，例如有一個 PlayerPanel 會呈現該玩家所有棋子的狀態、吃子狀態等。該如何設計比較好？透過 MVC 該如何設計？

**Ex 6:** 考試系統中的出題系統如何用 JSP model 來實踐？討論及畫出其架構圖。



# Chapter 17

## 隻手乾坤：Composite



## 17.1 目的與動機

複合設計樣式的目的是要單元物件與複合物件一視同仁，以降低程式的複雜度。所謂的複合物件就是它可以包含其他的複合物件或單元物件。

*To compose objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.*

容器，可以裝東西。也可以被裝。下面的圖形就是一個典型的複合物件，其中有顏色的圖示是一個基礎元件，而圓角矩型是一個容器，可以放圖示，也可以被放在一個個大的容器中。

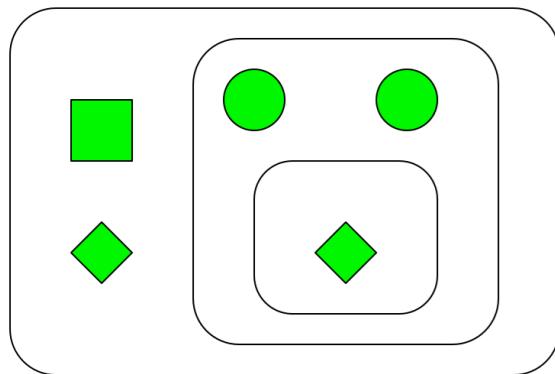


圖 17.1: 繪圖可視為一個 Composite 物件

## 應用

- 物件間有複合的關係。
- 我們不想區別單元物件與複合物件之間的差異，對他們有相同的呼叫方式。

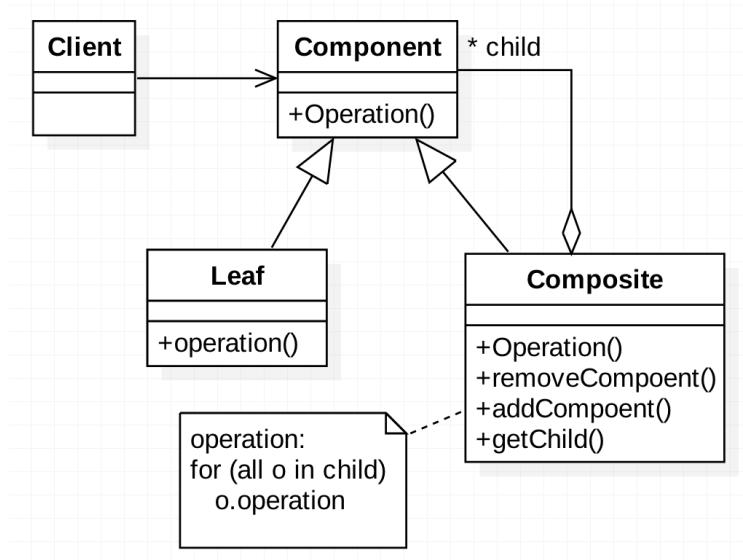


圖 17.2: Composite 結構

## 17.2 結構與方法

### 程式樣板

Java 已經有很好的集合物件可以幫我處理元件的管理，所以 Composite 內有一個 ArrayList 的物件，執行新增元件、刪除元件、列舉出元件的工作，也就是說，委託給 ArrayList。

```

1 // COMPONENT
2 abstract class Component {
3     abstract void op(); //OPERATION
4 }
5
6 // COMPOSITE
7 class Composite extends Component {
8     ArrayList<Component> list;
9
10    public Composite() {
11        list = new ArrayList<Component>();
12    }
13
14    void addComponent(Component c) {
15        list.add(c);
16    }
  
```

```

17
18 // OPERATION in COMPOSITE
19 void op() {
20     ListerIterator<Component> iterator = list.listIterator();
21     while (iterator.hasNext()) {
22         Component c = iterator.next();
23         c.op();
24     }
25 }
26
27
28 // LEAF
29 class Leaf extends Component {
30     // OPERATION in LEAF
31     void op() {
32         // ...
33     }
34 }
35
36 // CLIENT
37 class Main {
38     void m1(Component[] cc) {
39         for (Component c: cc)
40             c.op();
41     }
42 }

```

## 優點

- 容易新增新的元件。
- 讓客戶端的物件設計更為容易，因為不用區分單元物件與複合物件的差異。

## 17.3 範例

### GUI 元件範例

#### Solution 1：通通不一樣

```

1  public class Window {
2      Button[] buttons;
3      Menu[] menus;
4      TextArea[] textAreas;
5      WidgetContainer[] containers;
6
7      public void update() {
8          if (buttons != null) {
9              for (int k = 0; k < buttons.length; k++) {
10                  buttons[k].draw();
11                  if (menus != null) {
12                      for (int k = 0; k < menus.length; k++)
13                          menus[k].refresh();
14                  if (containers != null)
15                      for (int k = 0; k < containers.length; k++)
16                          containers[k].updateWidgets();
17              }
18          }
19      }
20  }

```

在這個例子中我們看到視窗物件把各種不同的介面元件視為不相同的物件，每一個呼叫的方式都不一樣。如此一來，這個程式的複雜度變高了，整個程式的耦合力也相對的高。

## Solution 2 : widget 與 container

Solution 1 的問題：違反 OCP 原則：如果我們想要加一個新種類的 widget，我們必須要修改 update()。違反開閉原則，當我們要新增新的元件、我們必須修改原來的程式碼。為了解決這個問題，程式碼修改如下：

```

1  public class Window {
2      Widget[] widgets;
3      WidgetContainer[] containers;
4
5      public void update() {
6          if (widgets != null)
7              for (int k = 0; k < widgets.length; k++)
8                  widgets[k].update();
9          if (containers != null)

```

```

10         for (int k = 0; k < containers.length; k++)
11             containers[k].updateWidgets()
12     }
13 }
```

在第二個解決方案中，我們把所有單元物件的介面統一，如此一來，程式變的更簡單了。但是單元物件與複合物件仍是不同的介面，處理上還是需要分開來。我們可以用 Composite 來改善。

### Solution 3: Composite Pattern!

```

1  public class Window {
2      Component[] components;
3      public void update() {
4          if (components != null)
5              for (int k = 0; k < components.length; k++)
6                  components[k].update();
7      }
8  }
```

在第三個方案中，所有的物件都視為 Component 了，統一的介面方法都是 update(), 程式是不是變得更簡潔了？

**通透與安全** 實作上有兩種選擇：通透與安全。

**通透 (Transparent)** : Component 具備 operation 以外，還具備了 add(Component), remove(Component) 等複合物件的方法。如此一來，Component 和 Composite 是沒有差異的，所以稱為 Transparent。但如此一來，Leaf 物件繼承了 add() 的功能顯得奇怪，通常我們會讓它沒有作用：

```

1  class Leaf extends Component {
2      public void add() {} // 沒有實作
3  }
```

但這是危險的，因為 Client 可能誤以為你有加入 Component 物件。

**安全 (Safe)** : Component 僅具備 operation 方法。Client 要新增元件時，必須先判斷它是不是一個複合物件：

```

1      if (component instanceof Composite) {
2          (Composite).add( ... )

```

## Java Collection

實作方面，我們通常會把一個 collection 物件（如 arraylist）埋在 composite 物件中，讓他來執行單元物件的管理。

```

1  class Graphic implements Component {
2      ArrayList<Component> gList;
3
4      public void add(Component c) {
5          gListc.add(c);
6      }
7
8      public void update() {
9          for (Component g: glist) {
10              g.update();
11          }
12      }
13  }

```

在 Java 中，複合物件的相關管理功能都會委託給 ArrayList 等 Collection 物件來幫忙管理。

## 17.4 練習

### 選擇/簡答

**Ex 1:** 在 Composite 設計樣式中，不包含哪一個角色

- (a) component
- (b) leaf
- (c) composite
- (d) decorator

**Ex 2:** 在 Composite 設計樣式中，composite 角色（複選）

- (a) 是 component 的子類別

- (b) 可以包含許多 component
- (c) 具備 addComponent() 的功能
- (d) 收到 client 訊息後，會轉給所有它包含的 component

**Ex 3:** 若 Component 有 addComponent() 等元件管理等功能，此方法稱為

- (a) Transparent
- (b) Safe

**Ex 4:** 關於 Composite, 何者對？

- (a) 對於 client, 複合物件與單元物件具備共通性，因此偶合力低
- (b) 不論是複合物件與單元物件都可以透過 addComponent() 加入元件
- (c) 複合物件可以包含單元物件，不可包含複合物件
- (d) 其觀念與遞迴概念相仿

**Ex 5:** 若 Component 內的方法為 update()，寫出 Composite 內的 update()

```

1   class Composite {
2       ?
3   }
```

**Ex 6:** Composite 和 Decorator 似乎都有動態包含、同時又具備繼承的關係，請問兩者的異同為何？

## 設計

**Ex 7:** 說明一個 Tree 是一個 Composite 的結構。一個系統的功能列透過 Tree 來展現，該如何用 Composite 來設計？

**Ex 8:** Ellipse, Rectangle, GraphicList 等三個類別，都實作了 Graphic 此介面，並且遵循了 Composite 設計樣式，請寫出 Ellipse, Rectangle, GraphicList 的關鍵程式碼。(GraphicList 是複合物件)

```

1   interface Graphic {
2       public void draw();
3   }
```

Hint: 在設計之前，先想想看 Composite 設計樣式包含哪些角色，對應於此問題的哪些物件？

**Ex 9:** 大學下有很多的院 (college)、系所 (department) 或一二級行政單位 (level1Office, level2Office)，院下面可以有系所；一級行政單位包含二級單位。當我們詢問該單位的人數時，會回傳該單位的人數及其所包含之單位的人數。

- 請應用 Composite 來設計此問題。(畫出 UML 圖)，要注意院不能包含二級單位。一級行政單位不能包含系。
- Hint: (1) 先找出什麼是複合物件。(2) 透過 ArrayList 幫你做複合物件該做的事。

**Ex 10:** (Lab) 在考試系統中，一個試卷可以只有一個題目，也可以包含其他試卷的題目，利用 Composite 如何來設計？假設一個題目的類別如下：

```
1  class Question {  
2      String description;  
3      public Question(String desc) {  
4          this.description = desc;  
5      }  
6      public void print() {  
7          System.out.println(desc);  
8      }  
9  }
```

下載 [Source Tree](#) 版本管理工具，並在 [GitHub](#) 中建立一個 open source 的專案，把你的程式碼 checkin 到你的 GitHub 中，注意必須有超過 3 次的 commit。



# Chapter 18

## 逍遙遊：Iterator



## 18.1 目的與動機

提供一個能夠方式去瀏覽某個集合體的內部資料，而不必去了解其內部的結構。

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

### 動機

在程式設計中，我們常常會遇到必須要叫出一一叫出陣列或集合物件內的內容，但基於資料隱藏的原則，這些集合物件的內容結構應該避免透露，也就是說暴露必要的，隱藏不必要的。Iterator 樣式的目的，便是透過抽離出瀏覽方法的方式，讓使用者不必去了解目標內容物的結構，就可以依著規定的方式呼叫出內容物件。

只可遠看，不可亵玩焉。

### 結構與方法

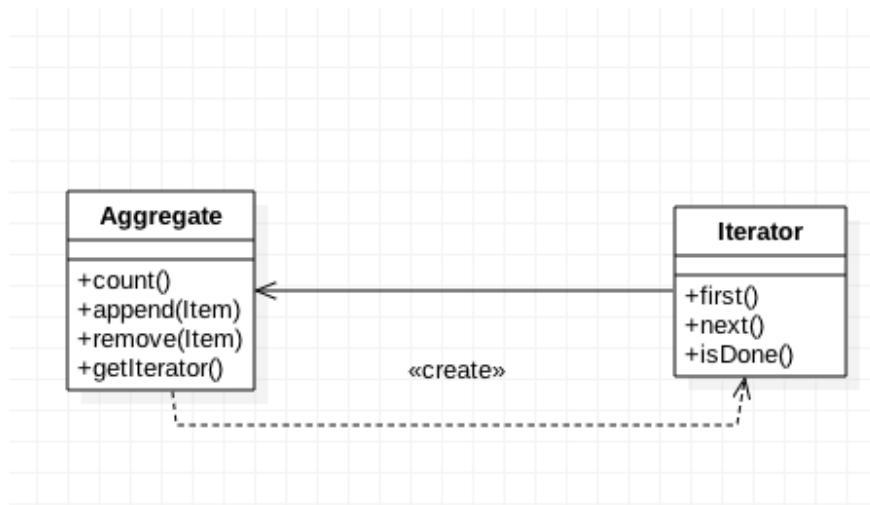


圖 18.1: Iterator 結構

不同結構的 Aggregate 需要不同的 Iterator 來瀏覽，因此可以運用 factory method 來依據不同型態的複合物件產生不同的瀏覽物件。例如 MapCollection 需要的瀏覽器是 MapIterator、ListCollection 需要的是 ListIterator。Collection 並沒有決定要產生哪一個 Iterator。如此的結構是為 Polymorphic Iterator，也是較為普遍的使用方法。

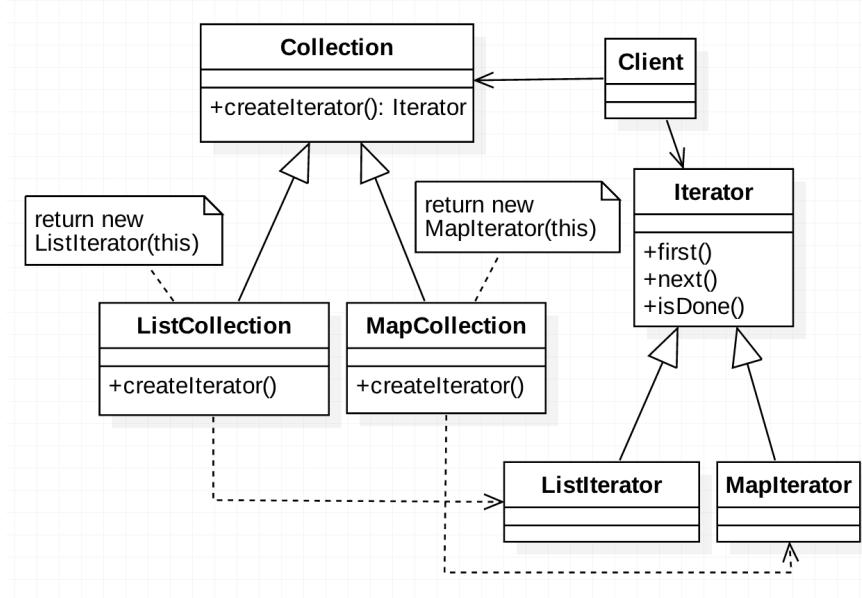


圖 18.2: Polymorphic iterator

## 參與者

- Aggregate/Collection：複合物件。定義如何取得瀏覽者物件的方法，例如 `createIterator()`，除了產生 `iterator` 以外，Aggregate 有對元素處理的方法，包含 `addElement`, `removeElement` 等。
- Iterator：瀏覽者介面，裡面定義一個複合物見的瀏覽方法，例如 `first`, `next` 等，還有判斷是否瀏覽完畢的檢查方法，例如 `isDone()`。
- ConcreteIterator：具體的瀏覽者類別，例如 `ListIterator`, `MapIterator` 等。
- ConcreteAggregate：具體的複合物件，例如 `ListCollection`, `MapCollection` 等。

## 程式樣板

```

1   Iterator it = aggregateObject.iterator();
2
3   while (it.hasNext()) {
4       Object = it.next();
5       ...
6   }
  
```

## 效益

可以有效的建立一個瀏覽物件的集合內容，不必因為新的內容增加而必須去新增方法去引用他，不必公開其內部結構。

## 18.2 範例

### Java Vector, ArrayList, HashMap

在 Java1.2 之後已經將 Iterator 的方法納入其中了，包含 Vector, ArrayList, HashMap 的瀏覽方法，如下：

```
5  public class IterationDemo {  
6  
7      public static void main( String[] args ) {  
8          // Vector 並且增加內容  
9          Vector<String> v = new Vector<String>();  
10         v.addElement( new String("Hello") );  
11         v.addElement( new String("Taichung") );  
12         v.addElement( new String("Have a nice day") );  
13         // 瀏覽 Vector 內的內容  
14         Iterator<String> it1 = v.iterator();  
15         System.out.print("Vector 内的內容為：" );  
16         traverse( it1 );  
17  
18         // ArrayList  
19         ArrayList<String> v2 = new ArrayList<String>();  
20         v2.add( new String("Hello") );  
21         v2.add( new String("Taipei") );  
22         v2.add( new String("Good morning") );  
23         // 瀏覽 ArrayList 內的內容  
24         Iterator<String> it2 = v2.iterator();  
25         System.out.print("\nArrayList 内的內容為：" );  
26         traverse( it2 );  
27  
28         // HashMap  
29         HashMap<String , Integer> v3 = new HashMap<String ,  
           Integer>();
```

```

30         v3.put("John", new Integer(172));
31         v3.put("Mary", new Integer(168));
32         v3.put("Nick", new Integer(180));
33         // 瀏覽 HashMap 的 Key
34         Iterator<String> it3 = v3.keySet().iterator();
35         System.out.print("\nHashMap 内的內容為:");
36         traverse(it3);
37     }
38
39     static void traverse(Iterator<String> e) {
40         while (e.hasNext()) {
41             System.out.print(e.next() + ", ");
42         }
43     }
44
45 }
```

Vector 內的內容為:Hello, Taichung, Have a nice day,  
ArrayList 內的內容為:Hello, Taipei, Good morning,  
HashMap 內的內容為:Mary, John, Nick,

[\[Get the code\]](#)

## 18.3 練習

### 選擇/簡答

**Ex 1:** 要取得一個集合物件內所有物件的，為何不直接從此集合物件取值？還需要先取得其瀏覽物件？

- (a) 瀏覽物件的功能較為強大
- (b) 瀏覽物件是 View, 透過如此可以將 Model 與 View 分離
- (c) 可以避免修改到集合物件的內容
- (d) 避免傳遞過多的資料，速度較快

**Ex 2:** Polymorphic iterator 是結合哪兩個樣式？

- (a) iterator
- (b) mediator

- (c) decorator
- (d) factory method

**Ex 3:** 查看 Java API, Iterator 包含哪些方法？

**Ex 4:** 下列的程式會計算一群學生的平均成績，你覺得有什麼問題？可以怎麼改善？

```

1   computeAverage( ArrayList<Student> s ) {
2       ...
3   }
4   class Student {
5       int score;
6       int getScore() { return score; }
7   }

```

**Ex 5:** 同上，如果我們用 Iterator 來做，「？」部分為何？

```

1   double getAverage( Iterator<Student> iterator ) {
2       ?
3   }

```

## 設計

**Ex 6:** 請設計 GradeIterator

```

1   interface GradeIterator {
2       boolean moreGrade();
3       Object nextGrade();
4   }
5
6   class Student {
7       private int[] grade;
8
9       public int[] getGrade() {
10          return grade;
11      }
12  }
13
14  StudentIterator implements GradeIterator {
15      Student st;
16      ?
17
18      public StudentIterator( Student st ) {

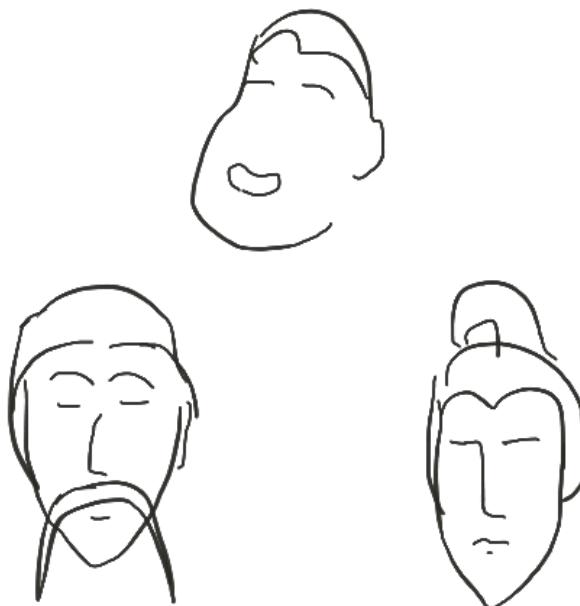
```

```
19         this.st = st;
20     }
21     boolean moreGrade() {
22     ?
23     }
24     Object nextGrade() {
25     ?
26     }
27 }
```



# Chapter 19

## 物換星移：State



### 19.1 目的與動機

將所有關於狀態的資訊與動作都包裝在一個狀態內，使物件的狀態較易擴充或修改。

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

把狀態封裝成一個類別。改變狀態的動作成為該類別內的方法。物件的行為通常取決於其內部的狀態。也就說，物件在不同的狀態時收到相同的訊息時，有可能會做出不同的動作(即執行不同的 operation)。因此，狀態是設計方法時的一種重要考量。在大部分的情況時，狀態的作用只隱含在程式，不會特別的出現在程式的結構。在物件導向的系統分析，我們常常使用狀態來表現物件的行為。不過這也有例外，State 設計樣式，就是特別將物件的狀態抽離出來成為一個類別。

## 適用時機

- 某一個事件處理有大量的條件式，且其判斷都取決於物件的狀態時，可使用此設計樣式。State 設計樣式將每一個判斷獨立成一個類別。
- 物件的狀態，及所相關的行為可能會擴充。

## 19.2 結構與方法

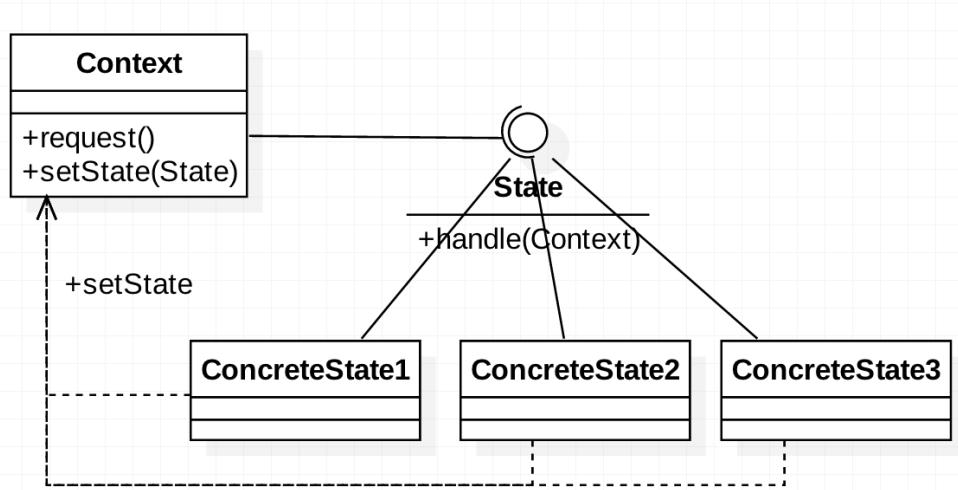


圖 19.1: State Structure

## 程式樣板

```

1  // 抽象的狀態物件，規範可能的行為
2  abstract class State {
3      // handle 是可能會造成狀態改變的行為
4      // Context 是擁有狀態的物件
5      abstract public void handle(Context c);
6  }
7
8  class ConcreteState1 extends State {
9      public void handle(Context c){
10         // change to ConcreteState2
11         c.setState(new ConcreteState2());
12     }
13 }
14
15 class ConcreteState2 extends State {
16     // change to ConcreteState1
17     public void handle(Context c){
18         c.setState(new ConcreteState1());
19     }
20 }
21
22 class Context {
23     State s;
24     public void request() {
25         // 把請求委託給 state 來處理
26         s.handle(this);
27     }
28
29     public void setState(State s) {
30         this.s = s;
31     }
32 }
```

注意 handle 必須包含一個 Context 的參數，這樣狀態轉換時才能呼叫 Context 的 setState() 來改變狀態。

## 參與者

- State：所有狀態類別的抽象父類別，用以定義狀態能夠接收的事件。
- Context：具有該狀態的物件或情境。
- ContextState：真實的狀態。方法 Handle() 將會是定義此狀態在執行 Handle() 後所到達的新狀態。

## 優點

- 將某個狀態的所有相關行為記錄在一個物件內。
- 可以避免用一大堆的 switch/if 來做狀態行為的轉換。
- 方便修改每一個狀態行為物件。
- 方便動態增加狀態行為物件。
- 更容易瞭解每一個狀態行為的意義。

## 缺點

會造成較多的物件去陳述各種狀態行為。

假設某個類別 StateClass 有三個狀態 S1、S2，S3，並具備以下的狀態轉移（行為）：

- s1 時收到 op1() 的事件，會移轉到 s2；收到 op2() 的事件，會移轉到 s3。
- s2 時收到 op1() 的事件，會移轉到 s3；收到 op2() 的事件，會移轉到 s1。
- s3 時收到 op1() 的事件，會移轉到 s1；收到 op2() 的事件，會移轉到 s2。

沒有用 State 樣式的程式：

```

1   class StateClass {
2       int s; //代表狀態的整數
3       public void op1(){
4           if (s == s1)
5               s = s2;
6           else if (s == s2)
7               s = s3;
8           else if (s == s3)
9               s = s1;
10      }
11      public void op2(){
12          if (s == s1)
13              s = s3;

```

```

14         else if ( s == s2 )
15             s = s1 ;
16         else if ( s == s3 )
17             s = s2 ;
18     }
19 }
```

想想看，如果用 State 樣式，該如何做？

## 19.3 範例

### 19.3.1 TCP/IP

TCP 傳輸控制協定的主要工作就是負責傳送與接收網路通訊過程的資料串，當接收到一個傳送資料的命令時，TCP 會建立網路要求，並將資料串分割成等長的數據封包，當獲得通訊認可時便將封包逐一傳出，並且監控目前網路狀態，必要的時候會暫停傳輸、重組封包、更換傳輸路徑等。

在 TCP 的連接控制的部分有許多的狀態處理動作，為了描述容易，我們只提出主要的三個狀態做為例子以描述 State 設計樣式的作用。這三個狀態分別是 Established、Listen、Closed，這些狀態會隨著方法的運作而作狀態的切換。我們也簡約的只定了三個方法：open、close、acknowledge。狀態的轉移如圖 13.4 所示，當 TCP 在 Closed 狀態接收到 open 的訊息後會轉移到 Listen 的狀態，等待客端的連接；在 Listen 時接收到 acknowledge 訊息後就會轉以到建立建立狀態：Established。我們可以用 State 設計樣式將每一個狀態判斷獨立成一個個的類別以增加設計的彈性。首先建立建立一個抽象的 TCPState 類別，描述會影響狀態改變的方法 (open、close、acknowledge)，接下來再讓每一個狀態擴充 TCPState 以描述各自的轉態轉移。

```

1 abstract public class TCPState {
2     public abstract void open(TCPConnection c);
3     public abstract void close(TCPConnection c);
4     public abstract void acknowledge(TCPConnection c);
5 }
```

使用 State 使其狀態類別可以宣告如下：

```

1 public class TCPClosed extends TCPState {
2
3     // TCPClosed 的狀態對 close 事件沒有任何反應
```

```

4         public void close(TCPConnection c) {}
5
6             // TCPClosed 的狀態對 acknowledge 事件沒有任何反應
7             public void acknowledge() {}
8
9             // 收到 Open 事件後會轉移到 Listen 的狀態
10            public void open(TCPConnection c) {
11                c.setState(new TCPListen());
12            }
13        }
14        public class TCPListen extends TCPState {
15            public void open(TCPConnection c) {}
16
17            public void close(TCPConnection c) {}
18
19            public void acknowledge(TCPConnection c) {
20                c.setState(new TCPEstablished());
21            }
22        }
23
24        public class TCPEstablished extends TCPState {
25            public void open(TCPConnection c) {}
26            public void acknowledge(TCPConnection c) {}
27            public void close(TCPConnection c) {
28                c.setState(new TCPClosed());
29            }
30        }

```

透過加入 State 樣式讓每一個狀態獨立，我們便能容易新增或修改每一個狀態本身，而不會因為一個修改的動作而讓系統必須要更改許多地方，甚至發生錯誤。另外，程式碼避免使用過多的 if/else，也可以避免錯誤。

## 19.4 練習

### 選擇/簡答

**Ex 1:** State 樣式中，保存物件狀態的類別為

- (a) Context

- (b) Observer
- (c) Strategy
- (d) Proxy

**Ex 2:** 為何 State.handle() 需要帶 Context 參數？

- (a) state 需要觀看 context 物件的值
- (b) state 需要設定 context 的新狀態
- (c) state 需要讀取 context 的狀態值

**Ex 3:** 關於 State 樣式，何者錯誤？

- (a) 把改變物件狀態的動作延遲到子類別
- (b) State 內所宣告的方法為可能改變狀態的方法
- (c) 若有 n 種可能狀態則會宣告 n 個 concrete state 類別
- (d) Context 把狀態改變的工作委託給 State 物件

**Ex 4:** (a) State 和 Strategy 有許多相同之處，兩者都利用了委託、繼承、多型的技巧。  
(b) 假設一個系統一次只能在一個狀態（通常都是如此），所以我們可以利用 Singleton 設計樣式在狀態物件上。  
(c) State 通常與 Observer 設計樣式合用，當情境（Context）改變時，就會通知狀態物件做修改，所以 Context 扮演 Subject 角色，State 扮演 Observer 角色。

## 設計

**Ex 5:** 有一個物件 A，當它接受到訊息 request，所表現出來的行為取決於他目前的狀態 s1, s2, s3。狀態轉移如下：

- s1 → s2
- s2 → s3
- s3 → s1

(1) 請用 if else 的方式（不要用 state 樣式）來進行狀態的改變；(2) 狀態的個數在日後可能是會變動的，所以我們不想要固定在物件 A 中，請用 State 設計樣式來設計此程式。

**Ex 6:** 圖 19.2 是物件 Context 的狀態行為，請用 State design pattern 設計之。

**Ex 7:** 描述一個 Stack 的狀態圖，利用 state 設計樣式來設計之。

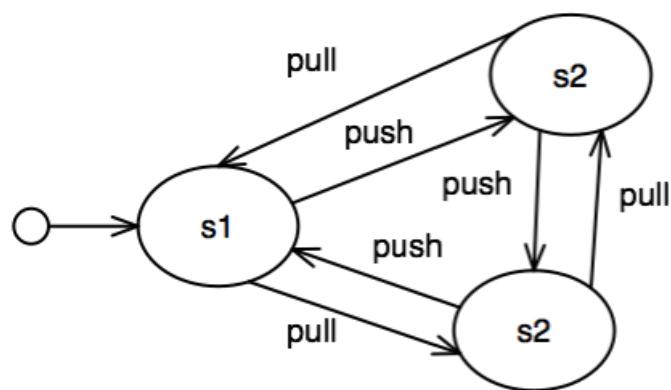


圖 19.2: Push

# Chapter 20

## 七星聚會 : Mediator



## 20.1 目的與動機

定義一個物件來封裝物件間的互動，避免讓物件直接對話藉此降低彼此耦合力。

*Define an object that encapsulates how 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.*

一群物件，統一透過一個物件來傳話。

### 應用時機

- 物件間有複雜的溝通行為；因此，想要重用這些物件變得困難，因為他和其他物間的耦合力太強了。
- 物件間的互動有變化時，方便修改，不會動到個別物件的內部

## 20.2 結構與方法

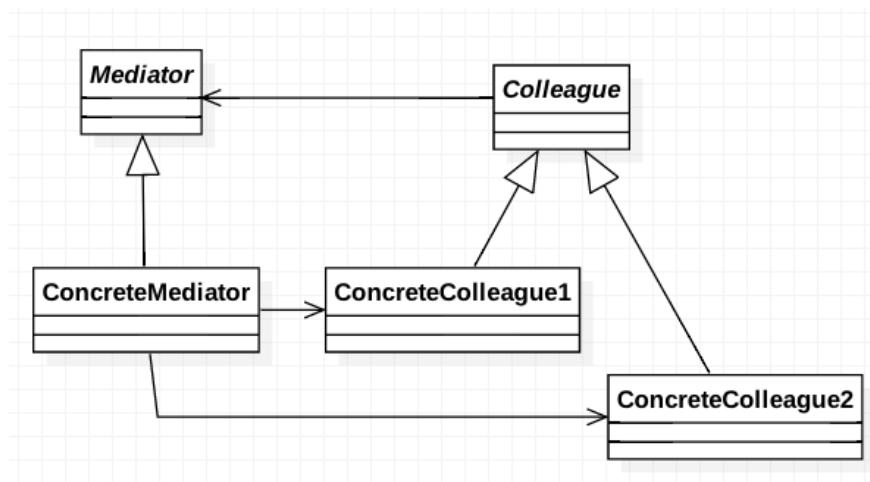


圖 20.1: CoR 結構

## 參與者

- Mediator: 抽象中介。定義同儕物件互動的介面
- ConcreteMediator: 實體中介。定義同儕物件間的互動行為
- Colleague: 同儕物件。與中介物件溝通的物件。

## 程式樣板

```

1  package mediator.template;
2
3  public class MediatorTemplate {
4
5      public static void main(String[] args) {
6          IMediator med = new Mediator();
7
8          // 生成 colleague 時，設定其 Mediator
9          Colleague1 c1 = new Colleague1(med);
10         Colleague2 c2 = new Colleague2(med);
11
12         // 送訊息給 c1, c2
13         c1.m1();
14         c2.m2();
15     }
16 }
17
18 interface IMediator {
19     void m1();
20
21     void m2();
22
23     void registerColleague1(Colleague1 c);
24
25     void registerColleague2(Colleague2 c);
26 }
27
28 class Mediator implements IMediator {
29     Colleague1 c1;
30     Colleague2 c2;
31 }
```

```
32      // 雖然 m1 是給 Colleague1 的訊息，但會轉給 mediator
33      // mediator 來決定會與哪些其他 Colleague 互動
34      public void m1() {
35          System.out.println("Mediator\u2022m1");
36          c2.op2();
37      }
38
39      public void m2() {
40          System.out.println("Mediator\u2022m2");
41          c1.op1();
42      }
43
44      // Mediator 必須識得每一個 Colleague
45      public void registerColleague1(Colleague1 c) {
46          this.c1 = c;
47      }
48
49      public void registerColleague2(Colleague2 c) {
50          this.c2 = c;
51      }
52
53  }
54
55  abstract class Colleague {
56      protected IMediator med;
57
58      public Colleague(IMediator med) {
59          this.med = med;
60      }
61  }
62
63  class Colleague1 extends Colleague {
64      public Colleague1(IMediator med) {
65          super(med);
66          med.registerColleague1(this);
67      }
68
69      void m1() {
70          System.out.println("Colleague1.m1");
71          med.m1();
```

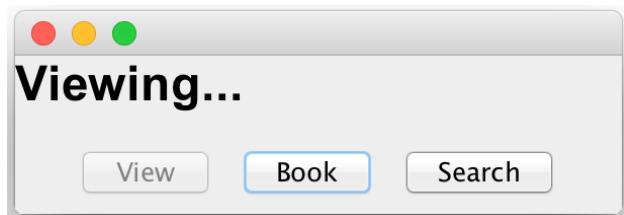
```

72         }
73
74     void op1() {
75         System.out.println("Colleague1.op1");
76     }
77 }
78
79 class Colleague2 extends Colleague {
80     public Colleague2(IMediator med) {
81         super(med);
82         med.registerColleague2(this);
83     }
84
85     void m2() {
86         System.out.println("Colleague2.m2");
87         med.m2();
88     }
89
90     void op2() {
91         System.out.println("Colleague2.op2");
92     }
93 }
```

[\[Get the code\]](#)

### 20.3 範例

想像一個 BookStore, 使用者可以上線點擊預定書籍、觀看書籍及搜尋書籍。



```

1 package mediator;
2
3 import java.awt.Font;
```

```
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class BookStoreDemo extends JFrame implements
13     ActionListener {
14     IMediator med = new BookStoreMediator();
15
16     BookStoreDemo() {
17         JPanel p = new JPanel();
18         p.add(new BtnView(this, med));
19         p.add(new BtnBook(this, med));
20         p.add(new BtnSearch(this, med));
21         getContentPane().add(new LblDisplay(med), "North");
22         getContentPane().add(p, "South");
23         setSize(300, 100);
24         setVisible(true);
25         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26     }
27
28     // 所有的 Colleague 都有統一的介面：Command
29     public void actionPerformed(ActionEvent ae) {
30         Command comd = (Command) ae.getSource();
31         comd.execute();
32     }
33
34     public static void main(String[] args) {
35         new BookStoreDemo();
36     }
37
38     // Colleague interface
39     interface Command {
40         void execute();
41     }
42
```

```
43 // Abstract Mediator, 定義所有 Mediator 的規格
44 interface IMediator {
45     public void book();
46
47     public void view();
48
49     public void search();
50
51     public void registerView(BtnView v);
52
53     public void registerSearch(BtnSearch s);
54
55     public void registerBook(BtnBook b);
56
57     public void registerDisplay(LblDisplay d);
58 }
59
60 // Concrete mediator
61 class BookStoreMediator implements IMediator {
62
63     BtnView btnView;
64     BtnSearch btnSearch;
65     BtnBook btnBook;
66     LblDisplay show;
67
68     // 註冊後 mediator 才知道 BtnView
69     public void registerView(BtnView v) {
70         btnView = v;
71     }
72
73     public void registerSearch(BtnSearch s) {
74         btnSearch = s;
75     }
76
77     public void registerBook(BtnBook b) {
78         btnBook = b;
79     }
80
81     public void registerDisplay(LblDisplay d) {
82         show = d;
```

```
83         }
84
85     // book 時所有需要溝通設定的都在這裡進行
86     public void book() {
87         btnBook.setEnabled(false);
88         btnView.setEnabled(true);
89         btnSearch.setEnabled(true);
90         show.setText("Booking... ");
91     }
92
93     // view 時所有需要溝通設定的都在這裡進行
94     public void view() {
95         btnView.setEnabled(false);
96         btnSearch.setEnabled(true);
97         btnBook.setEnabled(true);
98         show.setText("Viewing... ");
99     }
100
101    // search 時所有需要溝通設定的都在這裡進行
102    public void search() {
103        btnSearch.setEnabled(false);
104        btnView.setEnabled(true);
105        btnBook.setEnabled(true);
106        show.setText("Searching... ");
107    }
108 }
109
110 // A concrete colleague
111 class BtnView extends JButton implements Command {
112     IMediator med;
113
114     // colleague 也需要知道 mediator
115     BtnView(ActionListener al, IMediator m) {
116         super("View");
117         addActionListener(al);
118         med = m;
119         med.registerView(this);
120     }
121
122     // 轉呼叫 medicator 來處理
```

```
123     public void execute() {
124         med.view();
125     }
126 }
127
128 // A concrete colleague
129 class BtnSearch extends JButton implements Command {
130     IMediator med;
131
132     BtnSearch(ActionListener al, IMediator m) {
133         super("Search");
134         addActionListener(al);
135         med = m;
136         med.registerSearch(this);
137     }
138
139     public void execute() {
140         med.search();
141     }
142
143 }
144
145 // A concrete colleague
146 class BtnBook extends JButton implements Command {
147     IMediator med;
148
149     BtnBook(ActionListener al, IMediator m) {
150         super("Book");
151         addActionListener(al);
152         med = m;
153         med.registerBook(this);
154     }
155
156     public void execute() {
157         med.book();
158     }
159
160 }
161
162 class LblDisplay extends JLabel {
```

```

163     IMediator med;
164
165     LblDisplay(IMediator m) {
166         super("Just start... ");
167         med = m;
168         med.registerDisplay(this);
169         setFont(new Font("Arial", Font.BOLD, 24));
170     }
171 }
```

[\[Get the code\]](#)

## 20.4 練習

### 選擇/簡答

**Ex 1:** Mediator 的目的為何

- (a) 作為一群物件溝通的橋樑，藉此降低彼此的耦合度
- (b) Mediator 作為代理物件，藉此降低網路負擔，提昇效能
- (c) 一群物件不需要特別觀察其他物件的狀態，Mediator 會自動的通知
- (d) 統整相關物件的介面為唯一，藉此提昇 client 設計的簡潔性

**Ex 2:** \* 關於 Mediator 何者正確

- (a) Mediator 與所有的 Colleague 有 bi-direction 的 navigation 的關係
- (b) Colleague 彼此的關係是密切的，但透過 Mediator 的仲介，耦合度降低了
- (c) Mediator 透過委託的方式把訊息傳給其他 Colleague 物件處理事件
- (d) 塔台的運作可視為是一種 Mediator- 所有的交通工具與塔台溝通，而非彼此相乎溝通。

### 設計

**Ex 3:** 如果某個 Mediator 收到訊息後都必須通知其他所有的 Colleague, 那麼 Mediator 的角色和 Observer 樣式中的 Observable 是不是一樣？有何差異？Colleague 和 Observer 有何差異？

**Ex 4:** 在一個專案系統中，客戶、系統分析師、軟體工程師、系統維護工程師、使用者需要常常溝通，但彼此又不知道該與誰溝通，溝通的技巧為何。如何解決這個問題？哪一個設計樣式比較適合解決這個問題？樣式的角色對應為何？

# Chapter 21

## 綿綿不絕：Chain of Responsibility



## 21.1 動機與目的

為了降低訊息傳送者與接收者之間的耦合力，讓超過一個以上的物件來處理訊息。訊息鏈會把請求訊息一直傳遞下去直到有物件處理它。

*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.*

電腦機房常常會遇到很多異常現象：網路斷線了、Server 掛了、應用系統起不來、資料庫出現問題、被攻擊了等等事件。平時還好，同仁們都在，假日誰應該誰來處理呢？我們於是定下一個規則：通通給值班人員。於是所有的訊息都會到值班人員身上，但他並不是真正處理的人員，很多問題無法處理。當他無法處理時就往上回報給上面的上司，例如說是技術人員，技術人員無法處理時就往上報，可能是組長、然後是技術長、總經理等。

這樣的好處是：每一個人專注於訊息處理的形態與方法，而不用去理會訊息的流程。Java 1.0 對於事件的處理方式就是這樣的模式：

```

1  public boolean action(Event event, Object obj) {
2      if (event.target == test_btn)
3          doTestBtnAction();
4      else if (event.target == exit_btn)
5          doExitBtnAction();
6      else
7          return super.action(event, obj);
8      return true;
9  }

```

其中 `super` 就成為一個事件處理者的後繼者。

## 21.2 結構與方法

### 參與者

- Client: 事件的請求者
- Handler: 事件請求的統一介面。注意它會「包含」一個後繼者。
- ConcreteHandler: 具體的事件處理者，他會真的可以處理的事件進行處理，無法處理的轉移給後繼者處理。

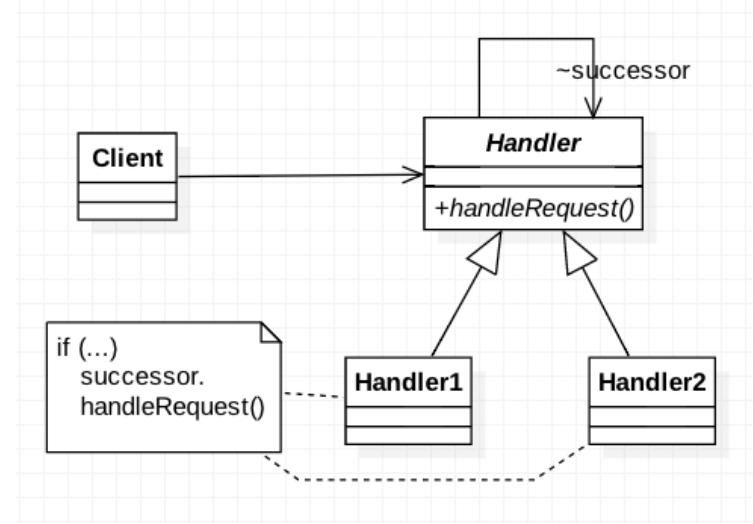


圖 21.1: CoR 結構

基本結構，針對一個問題來處理。

```

5   public class CoRDemo {
6
7       public static void main( String[] args ) {
8           Handler h2 = new Handler2( null );
9           Handler h1 = new Handler1( h2 );
10
11          // 第一個處理者都是 Handler1
12          h1.handleRequest( 100 );
13          h1.handleRequest( 0 );
14          h1.handleRequest( 10 );
15      }
16  }
17
18  abstract class Handler {
19      Handler successor;
20
21      public Handler( Handler h ) {
22          successor = h;
23      }
24
25      public void handleRequest( int i ) {
26          if ( successor != null )
27              successor.handleRequest( i );
  
```

```

28         else
29             System.out.println("No one can handle it");
30     }
31 }
32
33 class Handler1 extends Handler {
34     Random rn = new Random();
35     public Handler1(Handler h) {
36         super(h);
37     }
38
39     public void handleRequest(int x) {
40         int i = rn.nextInt(2);
41
42         // Handler 依據請求的參數值及自身的狀態來決定是否能夠處理
43         if (x > 10 || (i==1))
44             System.out.println("Handler1 will handle");
45         else
46             // 不能處理時就丟給下一個
47             super.handleRequest(x);
48     }
49 }
50
51 class Handler2 extends Handler {
52     public Handler2(Handler h) {
53         super(h);
54     }
55
56     public void handleRequest(int x) {
57         if (x < 10)
58             System.out.println("Handler2 will handle");
59         else
60             super.handleRequest(x);
61     }
62 }
```

[\[Get the code\]](#)

但如果我們要處理的問題型態有很多種呢？該怎麼辦？

方案一：單一介面 一個介面彙總所有的事件處理。

```

1  public interface Handler {
2      public void handleHelp();
3      public void handlePrint();
4      public void handleFormat();
5  }
6
7  public class ConcreteHandler implements Handler {
8      private Handler successor;
9      public ConcreteHandler(Handler successor) {
10         this.successor = successor;
11     }
12     public void handleHelp() {
13         // 本物件可以處理的。
14     }
15     public void handlePrint() {
16         // 級後繼者處理
17         successor.handlePrint();
18     }
19     public void handleFormat() {
20         // 級後繼者處理
21         successor.handleFormat();
22     }
23 }
```

問題：如果有新的問題型態，程式碼不好修改。好處是：強迫每個可能的處理者對問題「表態」。

方案二：多個介面 針對不同的訊息有不同的介面。

```

1  public interface HelpHandler {
2      public void handleHelp();
3  }
4  public interface PrintHandler {
5      public void handlePrint();
6  }
7  public interface FormatHandler {
8      public void handleFormat();
9  }
```

每一個處理針對他可能處理的事件進行實作，無法處理的交給他的後繼者。

```

1  public class ConcreteHandler implements HelpHandler,
2      PrintHandler, FormatHandler {
3      // 必須記錄每一個後繼者
4      private HelpHandler helpSuccessor;
5      private PrintHandler printSuccessor;
6      private FormatHandler formatSuccessor;
7
8      public ConcreteHandler (HelpHandler helpSuccessor,
9          PrintHandler printSuccessor, FormatHandler formatSuccessor
10         ) {
11         this.helpSuccessor = helpSuccessor;
12         this.printSuccessor = printSuccessor;
13         this.formatSuccessor = formatSuccessor;
14     }
15     public void handleHelp() {
16         // 本物件可以處理的。
17     }
18     public void handlePrint() {
19         printSuccessor.handlePrint();
20     }
21     public void handleFormat() {
22         formatSuccessor.handleFormat();
23     }
24 }
```

**方案三：參數** 只有一個介面，但帶一個參數，透過參數的判斷來決定處理的方式。

```

1  public interface Handler {
2      public void handleRequest(String request);
3  }
4
5  public class ConcreteHandler implements Handler {
6      private Handler successor;
7      public ConcreteHandler(Handler successor) {
8          this.successor = successor;
9      }
10     public void handleRequest(String request) {
11         if (request.equals("Help")) {
```

```

12         // 本物件可以處理的。
13     }
14     else
15         // 丟給後繼者
16         successor.handle(request);
17     }
18 }
```

方案四：封裝 把請求封裝成一個物件來處理。(參考 Command 設計樣式)。

```

1  public class Request {
2     private String type;
3     public Request(String type) {
4         this.type = type;
5     }
6     public String getType() { return type; }
7     public void execute() {
8         // 預設的執行方法
9     }
10 }
11
12 public class ConcreteHandler implements Handler {
13     private Handler successor;
14     public ConcreteHandler(Handler successor) {
15         this.successor = successor;
16     }
17     public void handleRequest(Request request) {
18         if (request instanceof HelpRequest) {
19             // 本物件可以處理
20         }
21         else if (request instanceof PrintRequest)
22             request.execute(); // 預設的處理方法
23     } else
24         successor.handle(request); // pass 紿後繼者
25     }
26 }
```

## 21.3 練習

### 選擇/簡答

**Ex 1:** Chain of responsibility 的目的為何？

- (a) 把物件串連起來，生成時一起生成。
- (b) 把事件的請求者與處理者抽離開來，降低彼此的耦合度。
- (c) 把物件串連起來，接收到事件請求時會通知所有的相關物件。
- (d) 設定一個事件的代理者，先由代理者處理，無法處理時在由真正的物件處理。

**Ex 2:** 關於 CoR 以下和者錯誤？

- (a) 每一個 Handler 生成時都需要指定一個後繼者。
- (b) 所有的 Hanlder 會實作同一個介面。
- (c) 不同型態的 Handler 不可相互成為後繼者。

**Ex 3:** CoR 和 Composite 看起來都一樣，都是有一個繼承、一個包含，兩者有和差異？

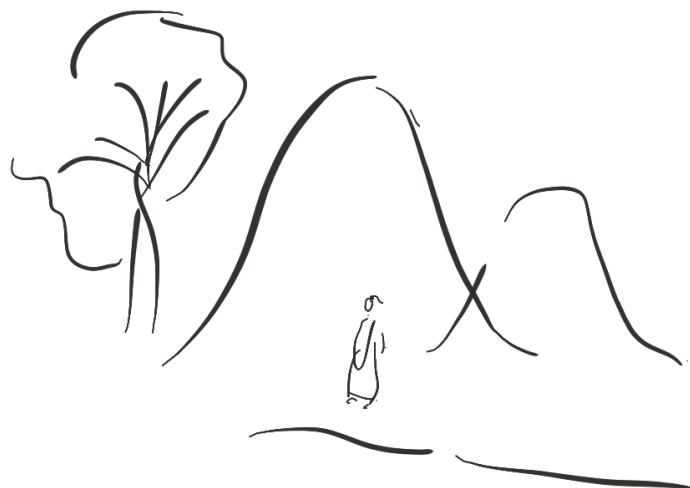
### 設計

**Ex 4:** 公司內有若干不同的角色，當遇到技術問題時解決的順序是：programmer, designer, architect。遇到管理問題的解決順序是：programmer, analyzer, manager, CEO。請利用 chain of responsibility 的方式來解決此問題。

```
1  interface handle {
2      public void handleTech();
3      public void handleMgmt();
4  }
```

## Chapter 22

清風拂山崗

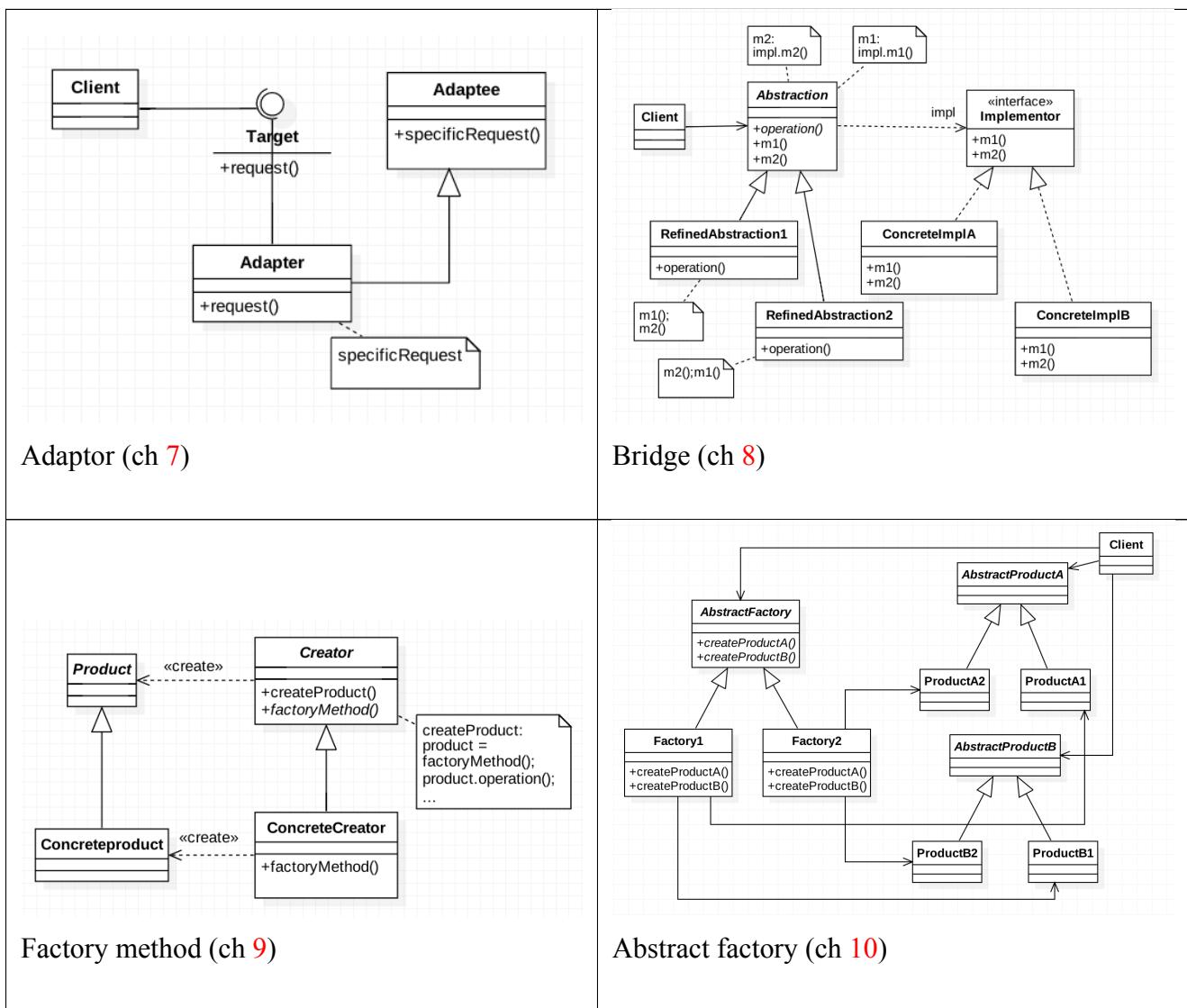


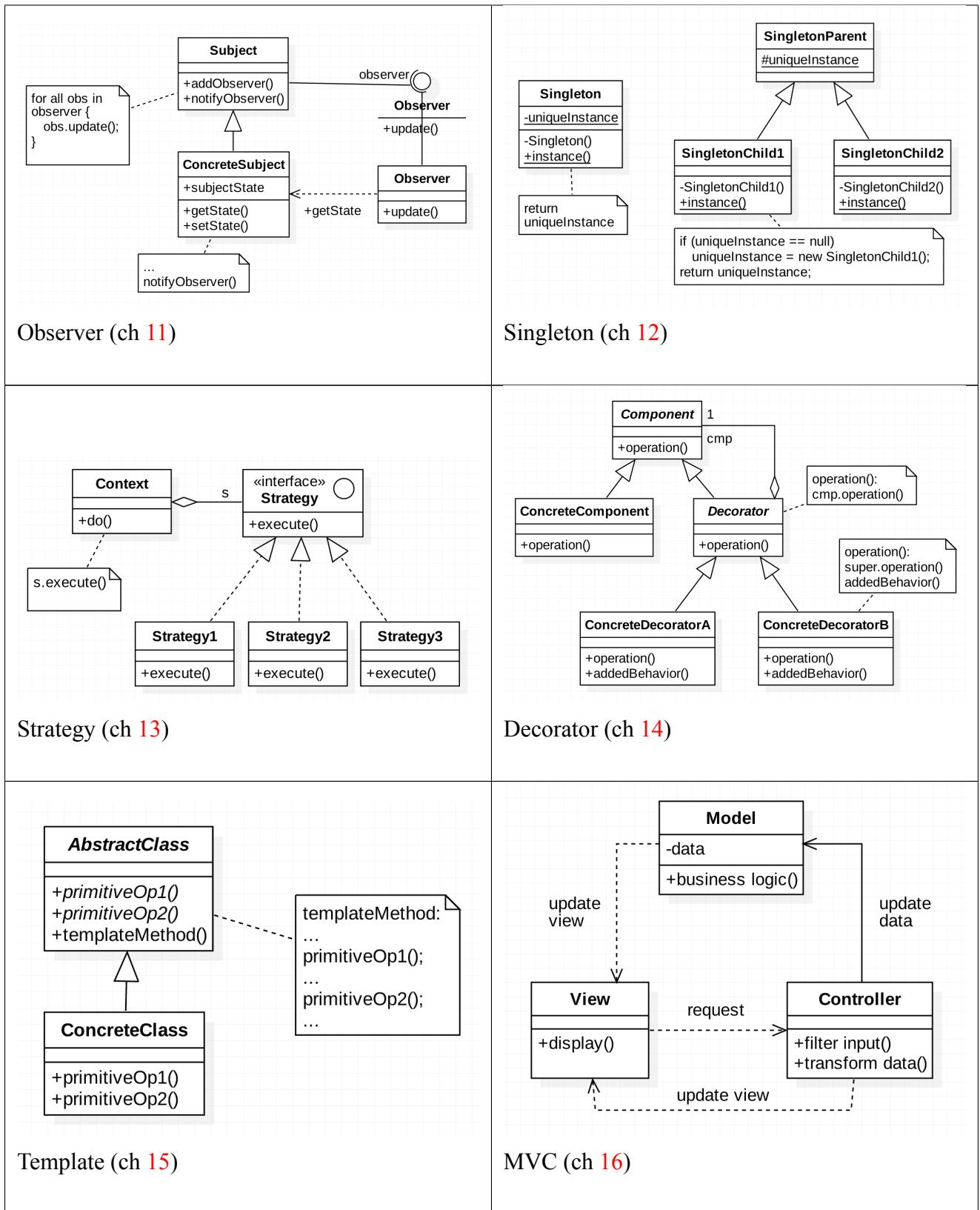
「它強任它強，清風拂山崗」在我看來這是武功的最高境界。

在軟體開發中最大的敵人，從軟體工程的角度來看是「Change」，從軟體技術的角度來看是「Function」。如果一個軟體工程師能夠瀟灑的說，「不論要多難的功能，不論需求規格在怎麼變，我都不怕」，真正做到「它強任它強，清風拂山崗」的境界了。

## 22.1 回顧

回顧我們學過的設計樣式，各位能夠看圖說故事嗎？





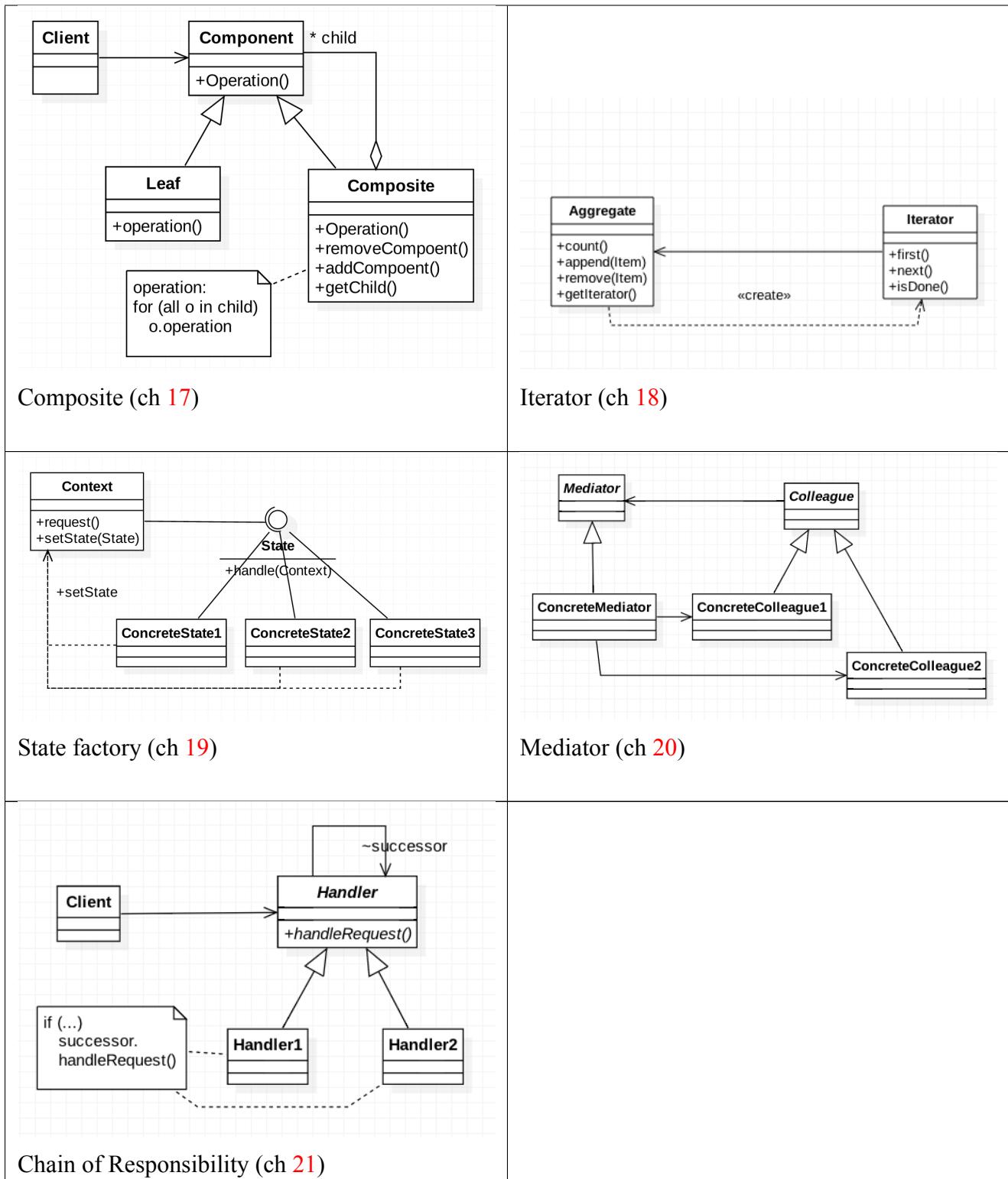


表 22.1: 設計樣式

## 22.2 練習

**Ex 1:** 比較 Composite 與 Decorator 設計樣式的異同



# **Chapter 23**

## **陣法：物件導向開發方法**

**23.1 步步為營瀑布陣**

**23.2 先聲奪人雛形陣**

**23.3 反覆突破敏捷陣**



# References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [2] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8):1430–1439, 2008.
- [3] Nien-Lin Hsueh, Jong-Yih Kuo, and Ching-Chiuan Lin. Object-oriented design: A goal-driven and pattern-based approach. *Software & Systems Modeling*, 8(1):67–84, 2009.
- [4] Wikipedia. Jsp model 2 architecture, 2015.
- [5] Wikipedia. Model–view–controller — wikipedia, the free encyclopedia, 2015. [Online; accessed 3-November-2015].



# Chapter 24

## 附錄

### 24.1 ChessGame: version 1: simple

應用物件導向的觀念，設計一個簡易的象棋系統，可以隨機的產出 32 的棋子並且排列於棋盤中，注意一個位置只能放一個棋子。

```
1 package chess.chessv1;
2
3 public class ChessGameV1 {
4     public static final int BLACK = 0, RED = 1;
5     Chess[] black;
6     Chess[] red;
7
8     public static void main(String args[]) {
9         // initialize the game
10        ChessGameV1 game = new ChessGameV1();
11        game.generateChess();
12        game.showAllChess();
13    }
14
15    void showAllChess() {
16        for (Chess c : black) {
17            System.out.println(c);
18        }
19        for (Chess c : red) {
20            System.out.println(c);
21        }
22    }
23}
```

```
22     }
23
24     void generateChess() {
25         black = new Chess[] { new Chess("將", 1, BLACK, 0),
26                             new Chess("士", 2, BLACK, 1),
27                             new Chess("士", 2, BLACK, 2),
28                             new Chess("象", 3, BLACK, 3),
29                             new Chess("象", 3, BLACK, 4),
30                             new Chess("車", 3, BLACK, 5),
31                             new Chess("車", 3, BLACK, 6),
32                             new Chess("馬", 3, BLACK, 7),
33                             new Chess("馬", 3, BLACK, 8),
34                             new Chess("包", 3, BLACK, 9),
35                             new Chess("包", 3, BLACK, 10),
36                             new Chess("卒", 3, BLACK, 11),
37                             new Chess("卒", 3, BLACK, 12),
38                             new Chess("卒", 3, BLACK, 13),
39                             new Chess("卒", 3, BLACK, 14),
40                             new Chess("卒", 3, BLACK, 15), };
41
42         red = new Chess[] { new Chess("帥", 1, RED, 16),
43                            new Chess("仕", 2, RED, 17),
44                            new Chess("仕", 2, RED, 18),
45                            new Chess("相", 3, RED, 19),
46                            new Chess("相", 3, BLACK, 20),
47                            new Chess("俌", 3, BLACK, 21),
48                            new Chess("俌", 3, BLACK, 22),
49                            new Chess("俌", 3, BLACK, 23),
50                            new Chess("俌", 3, BLACK, 24),
51                            new Chess("炮", 3, BLACK, 25),
52                            new Chess("炮", 3, BLACK, 26),
53                            new Chess("兵", 3, BLACK, 27),
54                            new Chess("兵", 3, BLACK, 28),
55                            new Chess("兵", 3, BLACK, 29),
56                            new Chess("兵", 3, BLACK, 30),
57                            new Chess("兵", 3, BLACK, 31), };
58
59         // change black's location
60         for (int i = 0; i < 16; i++) {
61             int target = (int) (Math.random() * 32);
```

```
62         if (target < 16) {
63             int originalLoc = black[i].loc;
64             black[i].loc = target;
65             black[target].loc = originalLoc;
66         } else if (target < 32) {
67             target == 16;
68             int originalLoc = black[i].loc;
69             black[i].loc = target;
70             red[target].loc = originalLoc;
71         } else {
72             System.out.println("LocationError");
73         }
74     }
75
76 }
77 }
78
79 class Chess {
80     public Chess(String name, int weight, int side, int loc) {
81         this.name = name;
82         this.weight = weight;
83         this.side = side;
84         this.loc = loc;
85     }
86
87     String name;
88     int weight;
89     int side;
90     int loc;
91
92     public String toString() {
93         return name + ", " + weight + ", " + loc;
94     }
95
96 }
```

[Get the code]

## 24.2 ChessGame: version 2: Abstraction

每個棋局都應該有玩家。改善 v1 的 ChessGame, 加上 Player 及 Location 的類別。

```

35             new Chess("象", 3, BLACK, 4),
36             new Chess("車", 3, BLACK, 5),
37             new Chess("車", 3, BLACK, 6),
38             new Chess("馬", 3, BLACK, 7),
39             new Chess("馬", 3, BLACK, 8),
40             new Chess("包", 3, BLACK, 9),
41             new Chess("包", 3, BLACK, 10),
42             new Chess("卒", 3, BLACK, 11),
43             new Chess("卒", 3, BLACK, 12),
44             new Chess("卒", 3, BLACK, 13),
45             new Chess("卒", 3, BLACK, 14),
46             new Chess("卒", 3, BLACK, 15), };

47
48     red = new Chess[] { new Chess("帥", 1, RED, 16),
49                         new Chess("仕", 2, RED, 17),
50                         new Chess("仕", 2, RED, 18),
51                         new Chess("相", 3, RED, 19),
52                         new Chess("相", 3, BLACK, 20),
53                         new Chess("俌", 3, BLACK, 21),
54                         new Chess("俌", 3, BLACK, 22),
55                         new Chess("俌", 3, BLACK, 23),
56                         new Chess("俌", 3, BLACK, 24),
57                         new Chess("炮", 3, BLACK, 25),
58                         new Chess("炮", 3, BLACK, 26),
59                         new Chess("兵", 3, BLACK, 27),
60                         new Chess("兵", 3, BLACK, 28),
61                         new Chess("兵", 3, BLACK, 29),
62                         new Chess("兵", 3, BLACK, 30),
63                         new Chess("兵", 3, BLACK, 31), };

64
65     // change black's location
66     for (int i = 0; i < 16; i++) {
67         int target = (int) (Math.random() * 32);
68         if (target < 16) {
69             int originalLoc = black[i].loc;
70             black[i].loc = target;
71             black[target].loc = originalLoc;
72         } else if (target < 32) {
73             target -= 16;
74             int originalLoc = black[i].loc;

```

```
75         black[i].loc = target;
76         red[target].loc = originalLoc;
77     } else {
78         System.out.println("LocationError");
79     }
80 }
81
82 }
83
84 @Override
85 void setPlayers(Player p1, Player p2) {
86     this.p1 = p1;
87     this.p2 = p2;
88 }
89
90 /**
91 * Show all chess in the game
92 */
93 void showAllChess() {
94     System.out.println("Players:" + p1 + " vs. " + p2);
95     for (Chess c : black) {
96         System.out.println(c);
97     }
98     for (Chess c : red) {
99         System.out.println(c);
100    }
101 }
102
103 }
104
105 class Chess {
106
107     public Chess(String name, int weight, int side, int loc) {
108         this.name = name;
109         this.weight = weight;
110         this.side = side;
111         this.loc = loc;
112     }
113
114     String name;
```

```

115     int weight;
116     int side;
117     int loc;
118
119     public String toString() {
120         return name + ",\t" + weight + ",loc=" + this.
121             getLocation();
122     }
123
124     public Location getLocation() {
125         return Location.getLocation(loc);
126     }
127
128     public int getLoc() {
129         return loc;
130     }
131
132     class Player {
133         String name;
134
135         public Player(String n) {
136             this.name = n;
137         }
138
139         public String toString() {
140             return name;
141         }
142     }
143
144     /*
145      * Location of a chess. It is composed of (x,y)
146      * x should be in the range (0,7), y: (0, 3)
147      */
148     class Location {
149         int x, y;
150
151         public Location(int x, int y) {
152             boolean xOK = (x >= 0 && x <= 7);
153             boolean yOK = (y >= 0 && y <= 3);

```

```

154         if (! (xOK && yOK)) {
155             System.out.println("Location Error:" + x + ", " + y
156                     );
157             System.exit(0);
158         }
159         this.x = x;
160         this.y = y;
161     }
162
163     public String toString() {
164         return "(" + x + ", " + y + ")";
165     }
166
167     public static Location getLocation(int loc) {
168         int x = (loc / 4);
169         int y = (loc % 4);
170         return new Location(x, y);
171     }
172 }
```

[\[Get the code\]](#)

### 24.3 ChessGame: version 3: Factory method

改善 ChessGame v2, 採用 Factory Method 的方法，使得 Chess 容易更換。

```

1   package chess.chessv3;
2
3   /*
4    * Force every chess game to have two players
5    */
6   abstract class AbstractGame {
7       abstract void setPlayers(Player p1, Player p2);
8   }
9
10  /*
11   * Version 3 chess game
12   */
13  public class ChessGameV3 extends AbstractGame {
```

```

14     public static final int BLACK = 0, RED = 1;
15     Chess[] black;
16     Chess[] red;
17     Player p1, p2;
18
19     public static void main( String args[] ) {
20         // initialize the game
21         ChessGameV3 game = new ChessGameV3();
22         game.setPlayers( new Player("Mary") , new Player("Jack") )
23             ;
24         game.generateAllChess();
25         game.showAllChess();
26     }
27
28     Chess makeChess( String name, int weight, int side, int loc )
29     {
30         return new Chess(name, weight, side, loc);
31     }
32
33     /*
34      * Using Factory method to generate all chess. This will
35      * make the
36      * chess generation more flexible, can be extended in the
37      * child class.
38      */
39
40     void generateAllChess() {
41         black = new Chess[] { makeChess("將", 1, BLACK, 0),
42                             makeChess("士", 2, BLACK, 1), makeChess("士",
43                                     2, BLACK, 2),
44                             makeChess("象", 3, BLACK, 3), makeChess("象",
45                                     3, BLACK, 4),
46                             makeChess("車", 3, BLACK, 5), makeChess("車",
47                                     3, BLACK, 6),
48                             makeChess("馬", 3, BLACK, 7), makeChess("馬",
49                                     3, BLACK, 8),
50                             makeChess("包", 3, BLACK, 9), makeChess("包",
51                                     3, BLACK, 10),
52                             makeChess("卒", 3, BLACK, 11), makeChess("卒",
53                                     3, BLACK, 12),
54                             makeChess("卒", 3, BLACK, 13), makeChess("卒",
55                                     3, BLACK, 14) };

```

```

        3 , BLACK, 14) ,
44      makeChess("卒", 3, BLACK, 15) , } ;

45
46      red = new Chess[] { makeChess("帥", 1, RED, 16) ,
47                          makeChess("仕", 2, RED, 17) , makeChess("仕", 2,
48                                         RED, 18) ,
49                          makeChess("相", 3, RED, 19) , makeChess("相", 3,
50                                         BLACK, 20) ,
51                          makeChess("傌", 3, BLACK, 21) , makeChess("傌",
52                                         3, BLACK, 22) ,
53                          makeChess("傌", 3, BLACK, 23) , makeChess("傌",
54                                         3, BLACK, 24) ,
55                          makeChess("炮", 3, BLACK, 25) , makeChess("炮",
56                                         3, BLACK, 26) ,
57                          makeChess("兵", 3, BLACK, 27) , makeChess("兵",
58                                         3, BLACK, 28) ,
59                          makeChess("兵", 3, BLACK, 29) , makeChess("兵",
60                                         3, BLACK, 30) ,
61                          makeChess("兵", 3, BLACK, 31) , } ;

62
63      // change black's location
64      for (int i = 0; i < 16; i++) {
65          int target = (int) (Math.random() * 32);
66          if (target < 16) {
67              int originalLoc = black[i].loc;
68              black[i].loc = target;
69              black[target].loc = originalLoc;
70          } else if (target < 32) {
71              target -= 16;
72              int originalLoc = black[i].loc;
73              black[i].loc = target;
74              red[target].loc = originalLoc;
75          } else {
76              System.out.println("LocationError");
77          }
78      }
79
80  }
81
82
83  @Override

```

```
76     void setPlayers(Player p1, Player p2) {
77         this.p1 = p1;
78         this.p2 = p2;
79     }
80
81     void showAllChess() {
82         System.out.println("Players:" + p1 + " vs. " + p2);
83         for (Chess c : black) {
84             System.out.println(c);
85         }
86         for (Chess c : red) {
87             System.out.println(c);
88         }
89     }
90
91 }
92
93 class Chess {
94     // name, weight, side (black/red), location (1-32)
95     public Chess(String name, int weight, int side, int loc) {
96         this.name = name;
97         this.weight = weight;
98         this.side = side;
99         this.loc = loc;
100    }
101
102    String name;
103    int weight;
104    int side;
105    int loc;
106
107    public String toString() {
108        return name + ",\t" + weight + ",loc=" + this.
109            getLocation();
110    }
111
112    public Location getLocation() {
113        return Location.getLocation(loc);
114    }
115}
```

```
115     public int getLoc() {
116         return loc;
117     }
118 }
119
120 class Player {
121     String name;
122
123     public Player(String n) {
124         this.name = n;
125     }
126
127     public String toString() {
128         return name;
129     }
130 }
131
132 /*
133 * Location of a chess. It is composed of (x,y) x should be in
134 * the range (0,3),
135 * y: (0, 7)
136 */
137 class Location {
138     int x, y;
139
140     public Location(int x, int y) {
141         boolean xOK = (x >= 0 && x <= 3);
142         boolean yOK = (y >= 0 && y <= 7);
143         if (!(xOK && yOK)) {
144             System.out.println("Location Error:" + x + ", " + y
145                 );
146             System.exit(0);
147         }
148         this.x = x;
149         this.y = y;
150     }
151
152     public String toString() {
153         return "(" + x + ", " + y + ")";
154     }
155 }
```

```

153
154     public static Location getLocation(int loc) {
155         int x = (loc / 8);
156         int y = (loc % 8);
157         return new Location(x, y);
158     }
159 }
```

[Get the code]

## 24.4 ChessGame: version 4: Strategy

Version 4 採用 **Strategy** 讓棋子的初始位置可以彈性設定。同時也新增了棋盤類別 **ChessBoard**, 記錄每一個位置對應的棋子, 提供介面讓使用者對某個位置進行選擇、吃子的動作。

卒	相	車	象	象	卒	帥	馬
馬	包	車	士	[將]		兵	卒
卒	仕	偶	包	兵	進	進	仕
士	兵	炮	兵	偶	炮	相	兵

圖 24.1: ChessGame v4 的執行結果

```

1 package chess.chessv4;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /*
7  * 規範每一個遊戲都應該有兩個玩家
8  */
9 abstract class AbstractGame {
10     abstract void setPlayers(Player p1, Player p2);
11 }
12
13 /*
14  * Version 4 採用 Strategy 讓棋子的初始位置可以彈性設定。
15  * 新增了棋盤類別 ChessBoard, 記錄每一個位置對應的棋子,
16  * 也提供介面讓使用者對某個位子進行選擇、吃子的動作。
17 }
```

```
16     */
17     public class ChessGameV4 extends AbstractGame {
18         public static final int BLACK = 0, RED = 1;
19         Chess[] chesses;
20         Player p1, p2;
21
22         public static void main(String args[]) {
23             // initialize the game
24             ChessGameV4 game = new ChessGameV4();
25             game.setPlayers(new Player("Mary"), new Player("Jack"))
26             ;
27             game.generateAllChess();
28             ChessBoard cb = new ChessBoard();
29             // game.setChessBoarding(cb, new SimpleChessBoarding());
30             game.setChessBoarding(cb, new RandomChessBoarding());
31
32             // 模擬一些使用者的行為
33             cb.select(12);
34             cb.kill(13);
35
36             cb.showBoard();
37         }
38
39         protected void setChessBoarding(ChessBoard cb,
40             ChessBoarding boarding) {
41             boarding.setLocation(chesses);
42             cb.putChess(chesses);
43         }
44
45         Chess makeChess(String name, int side) {
46             return new Chess(name, side);
47         }
48
49         /*
50          * 採用 Factory Method 來產生棋子
51          */
52         void generateAllChess() {
53             chesses = new Chess[] { makeChess("將", BLACK),
54                 makeChess("士", BLACK),
55                 makeChess("士", BLACK), makeChess("象", BLACK),
```

```

53     makeChess("象", BLACK), makeChess("車", BLACK),
54     makeChess("車", BLACK), makeChess("馬", BLACK),
55     makeChess("馬", BLACK), makeChess("包", BLACK),
56     makeChess("包", BLACK), makeChess("卒", BLACK),
57     makeChess("卒", BLACK), makeChess("卒", BLACK),
58     makeChess("卒", BLACK), makeChess("卒", BLACK),
59     makeChess("卒", RED), makeChess("仕", RED),
60             makeChess("仕", RED),
61             makeChess("相", RED), makeChess("相", BLACK),
62             makeChess("俌", BLACK), makeChess("俌", BLACK),
63             makeChess("俌", BLACK), makeChess("俌", BLACK),
64             makeChess("炮", BLACK), makeChess("炮", BLACK),
65             makeChess("兵", BLACK), makeChess("兵", BLACK),
66             makeChess("兵", BLACK), makeChess("兵", BLACK),
67             makeChess("兵", BLACK) };
68
69     void setPlayers(Player p1, Player p2) {
70         this.p1 = p1;
71         this.p2 = p2;
72     }
73
74     void showAllChess() {
75         System.out.println("Players: " + p1 + " vs. " + p2);
76         for (Chess c : chesses) {
77             System.out.println(c);
78         }
79     }
80 }
81
82 class Chess {
83     public static final int CREATED = 0;
84     public static final int SELECTED = 1;
85     public static final int KILLED = 2;
86
87     private String name;
88     private int weight;
89     private int side;
90     private int loc;
91     private int status;

```

```
92
93     public Chess( String name, int side) {
94         this.name = name;
95         this.side = side;
96         status = Chess.CREATED;
97     }
98
99     public String getName() {
100        return name;
101    }
102
103    public Location getLocation() {
104        return Location.getLocation(loc);
105    }
106
107    public int getLoc() {
108        return loc;
109    }
110
111    void setLoc(int i) {
112        this.loc = i;
113    }
114
115    public String toString() {
116        return name + "," + "loc=" + getLocation() + ",status"
117            +" " + status;
118    }
119
120    public void setStatus(int s) {
121        this.status = s;
122    }
123
124    // 是否被選了
125    public boolean isSelected() {
126        return status == Chess.SELECTED ? true : false;
127    }
128
129    class Player {
130        String name;
```

```
131
132     public Player(String n) {
133         this.name = n;
134     }
135
136     public String toString() {
137         return name;
138     }
139 }
140
141 /**
142 * Location of a chess. It is composed of (x,y) x should be in
143 * the range (0,3),
144 * y: (0, 7)
145 */
146 class Location {
147     int x, y;
148
149     public Location(int loc) {
150         this(loc / 8, loc % 8);
151     }
152
153     public Location(int x, int y) {
154         boolean xOK = (x >= 0 && x <= 3);
155         boolean yOK = (y >= 0 && y <= 7);
156         if (!(xOK && yOK)) {
157             System.out.println("LocationError:" + x + ", " + y
158                     );
159             System.exit(0);
160         }
161         this.x = x;
162         this.y = y;
163     }
164
165     public String toString() {
166         return "(" + x + ", " + y + ")";
167     }
168
169     public static Location getLocation(int loc) {
170         int x = (loc / 8);
```

```
169         int y = (loc % 8);
170         return new Location(x, y);
171     }
172
173     int getLoc() {
174         int result = 8 * x + y;
175         return result;
176     }
177 }
178
179 class ChessBoard {
180     // board 記錄每一個 Location 對應的棋子
181     Map<Location, Chess> board = new HashMap<Location, Chess>()
182         ;
183     Location[] locs;
184     Chess[] chesses;
185
186     // put the chess to the board
187     public void putChess(Chess[] chesses) {
188         this.chesses = chesses;
189         locs = new Location[chesses.length];
190         for (int i = 0; i < chesses.length; i++) {
191             locs[i] = new Location(i);
192         }
193         for (int i = 0; i < chesses.length; i++) {
194             Chess c = chesses[i];
195             int loc = c.getLoc();
196             board.put(locs[loc], c);
197         }
198     }
199
200     public void kill(int loc) {
201         Chess c = (Chess) board.get(locs[loc]);
202         c.setStatus(Chess.KILLED);
203         board.put(locs[loc], null);
204     }
205
206     public void select(int loc) {
207         Chess c = (Chess) board.get(locs[loc]);
208         c.setStatus(Chess.SELECTED);
```

```

208         }
209
210     void showBoard() {
211         for (int i = 0; i < locs.length; i++) {
212             Chess c = (Chess) (board.get(locs[i]));
213             boolean noChessOnLocation = (c == null) ? true :
214                 false;
215             if (noChessOnLocation)
216                 System.out.print("uu");
217             else if (c.isSelected())
218                 System.out.print("[" + c.getName() + "]");
219             else
220                 System.out.print(c.getName());
221             // new row
222             if ((i + 1) % 8 == 0)
223                 System.out.println("");
224             else
225                 System.out.print("\t");
226         }
227     }
228
229 /**
230 * 設定棋子位置的方法。Strategy 設計樣式中的 AbstractStrategy
231 */
232 interface ChessBoarding {
233     public void setLocation(Chess[] chesses);
234 }
235
236 /**
237 * 隨機的設定棋子的位置
238 */
239 class RandomChessBoarding implements ChessBoarding {
240
241     public void setLocation(Chess[] chesses) {
242         // initialize location
243         for (int i = 0; i < chesses.length; i++) {
244             chesses[i].setLoc(i);
245         }
246         // exchange location

```

```
247     for (int i = 0; i < chesses.length / 2; i++) {
248         int a = (int) (Math.random() * 32);
249         int b = (int) (Math.random() * 32);
250         Chess chessA = chesses[a];
251         Chess chessB = chesses[b];
252         int chessALocation = chessA.getLoc();
253         // switch the location
254         chessA.setLoc(chessB.getLoc());
255         chessB.setLoc(chessALocation);
256     }
257 }
258 }
259
260 /*
261 * 簡易的棋子排版，會依照 將士...卒帥仕...兵 的方式排版
262 */
263 class SimpleChessBoarding implements ChessBoarding {
264     public void setLocation(Chess[] chesses) {
265         for (int i = 0; i < chesses.length; i++) {
266             chesses[i].setLoc(i);
267         }
268     }
269 }
```

[Get the code]