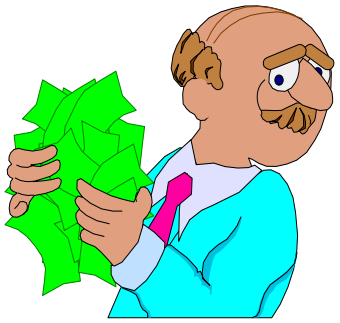
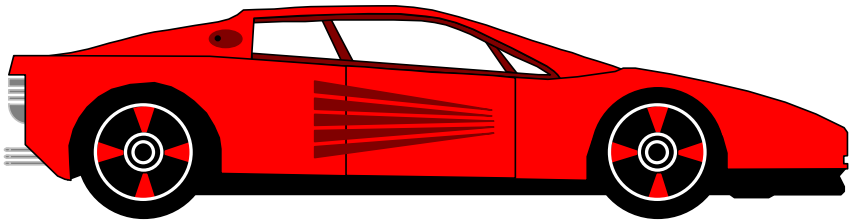
軟體設計課程大綱

- 軟體架構設計原則(Software Architecture Design Principle)
  - 設計介面再寫程式(Program To An Interface)
  - 整體-部分(Whole-part)與委任
  - 善用組合超越繼承(Favor Composition Over Inheritance)
  - 開放-封閉法則(Open-Closed Principle)
  - 繼承取代原則(Liskov Substitution Principle)
  - 介面分離原則(Interface Segregation Principle)
  - 依賴反轉原則(The Dependency Inversion Principle)

設計介面(Interface)後寫程式，
而非直接實做
Program To An Interface, Not An Implementation





?

介面範例 1

```
// Interface IManeuverable provides the specification for a maneuverable vehicle.
public interface IManeuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
public class Car implements IManeuverable
{ // Code here. }
public class Tank implements IManeuverable
{ // Code here. }
public class Boat implements IManeuverable
{ // Code here. }
public class Submarine implements IManeuverable
{ // Code here. }
```

介面範例 2

- 其他類別的物件，可以透過這個介面直接操作，而不需瞭解操作的物件到底是什麼物件 (car, boat, submarine)

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

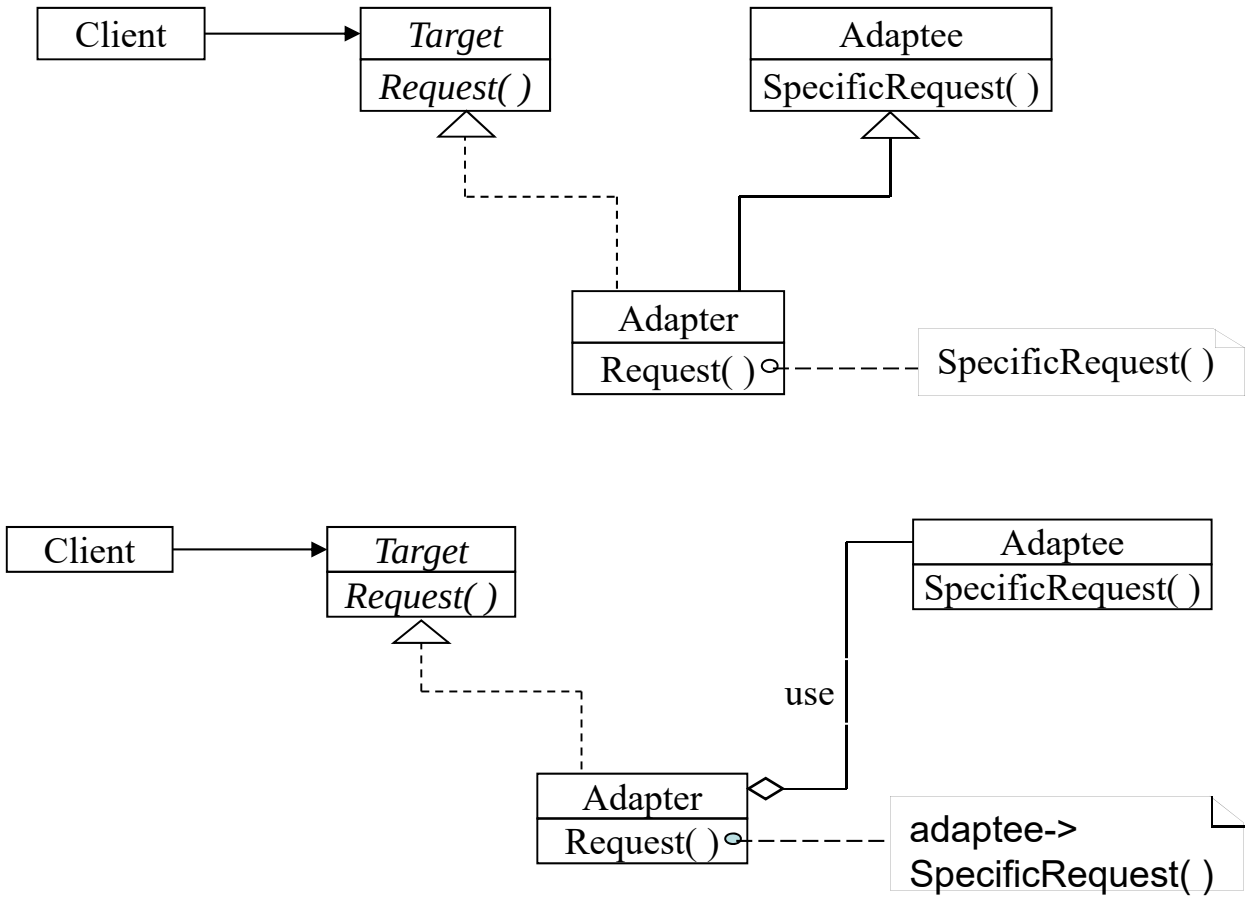
介面 1

- 介面是一組方法(method)
  - 介面表達"is a kind of that supports this interface“
  - 一個物件透過介面被其他物件存取
- 介面是一個資料型態(data type)
  - 不同的物件有相同的介面，
  - 一個物件可以實做多個介面
- 界面是熱抽換(plugability)-動態改變-的關鍵
- *介面繼承(Interface Inheritance)是一種契約繼承(contract Inheritance)*
- *類別繼承(Class Inheritance)是一種實做繼承(Implementation Inheritance)*

介面 2

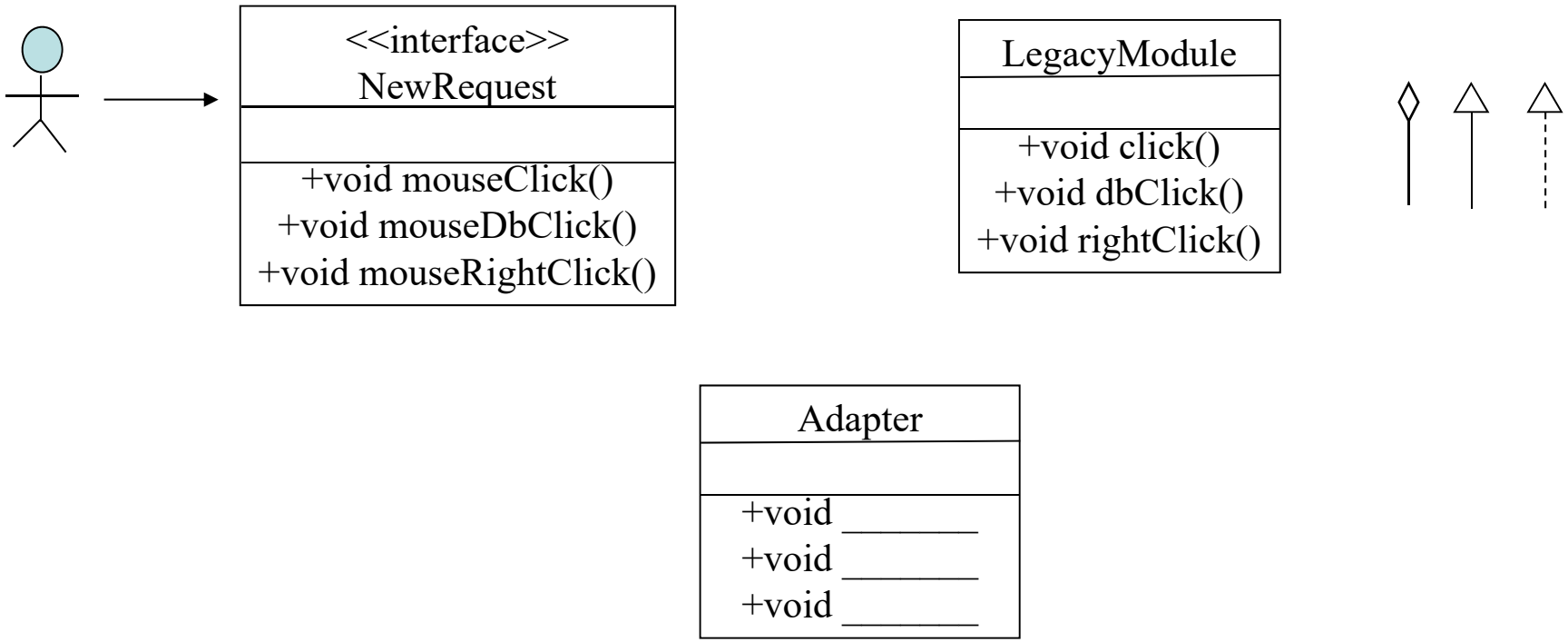
- 優點
  - 客戶端程式(Client)不需知道使用哪一種特殊類別物件。
  - 改善組合機會：被包含物件可使用任何類別實做，只要實做所需介面
  - 物件易被其他物件取代，可降低耦合(Loosens coupling)、提高重複使用性、彈性 。
- 缺點
  - 增加設計的複雜性

Adapter Pattern Structure



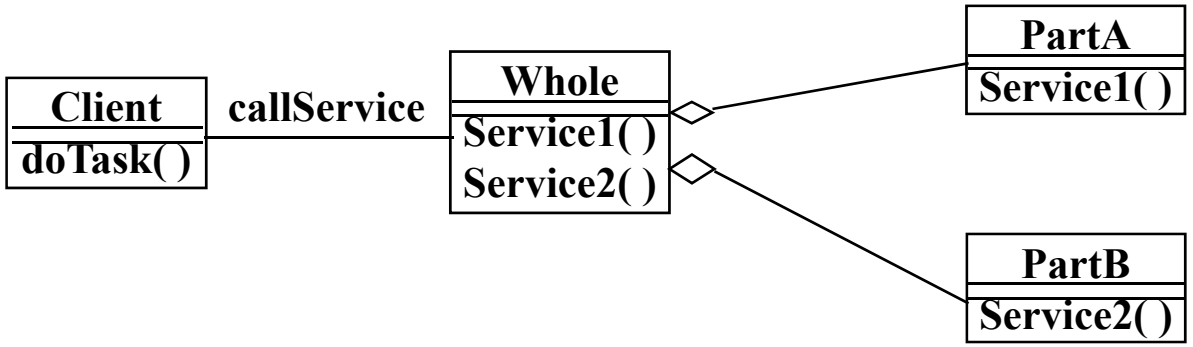
Exercise

- 目前已有的是LegacyModule的介面使用方式
- 如何設計，可以讓客戶端程式依NewRequest使用方式來使用



整體-部分與委任 1

- 問題
  - 複雜的物件由一些存在的物件組成
  - 支援重用性(reusability)、可改變性(changeability)
- 解決
  - 一個元件(component)封裝較小的物件，客戶端程式避免直接存取小物件。
  - 元件將責任委託給小物件。

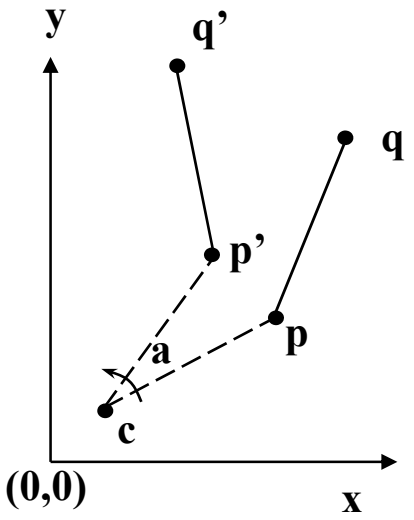


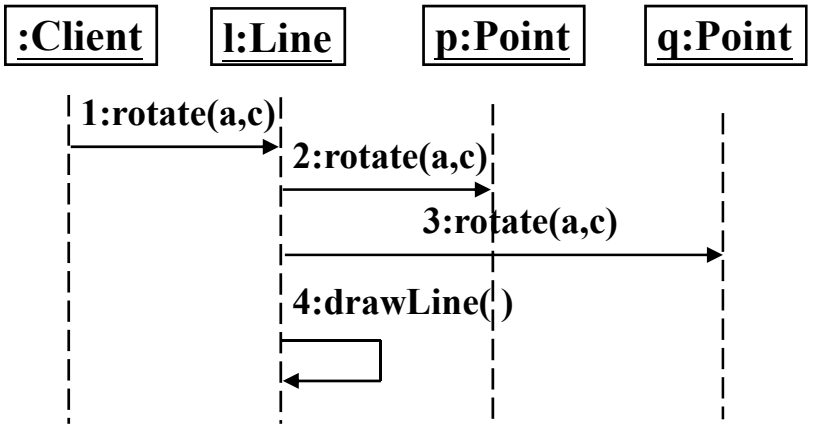
整體-部分與委任2

- 三種組合關係
  - 組合-部分(Assembly-Part)關係
    - 產品和他的零件，內部結構緊緊整合在一起
    - 例如：飛機和他的引擎
  - 容器-內容(Container-Contents)關係
    - 內容物沒有緊緊跟容器有內部結構關係，
    - 例如生日禮盒內有書、和生日卡
  - 集合-成員(Collection-Member)關係
    - 將相似的物件群組化，例如組織和他的成員、班級與同學

整體-部分與委任3

- 範例：CAD系統中，線條在二維座標中旋轉
  - 客戶端程式驅動line L的旋轉的功能，傳入角度和圓心
  - line L 呼叫點 p 和點 q 的旋轉方法
  - line L 使用新的座標p’ 和 q’的資訊來重畫





善用組合(Composition)特性
超越繼承
Favor Composition Over Inheritance

組合

- 重複(reuse)使用新功能的方法(Method)
  - 物件(object)藉由結合其他物件，達成委任(delegating)
- 優點
  - 主要的物件透過介面(interfaces)存取被包含的物件
  - 黑箱(Black-box)重複使用：被包含的物件內部是不可視(visible)的
  - 好的封裝性(Good encapsulation)
  - 較少的實做(implementation)依賴
  - 每一個類別聚焦於一個工作任務(task)
  - 在執行時期動態(dynamically)定義
- 缺點：系統有較多物件、介面必須很小心的定義

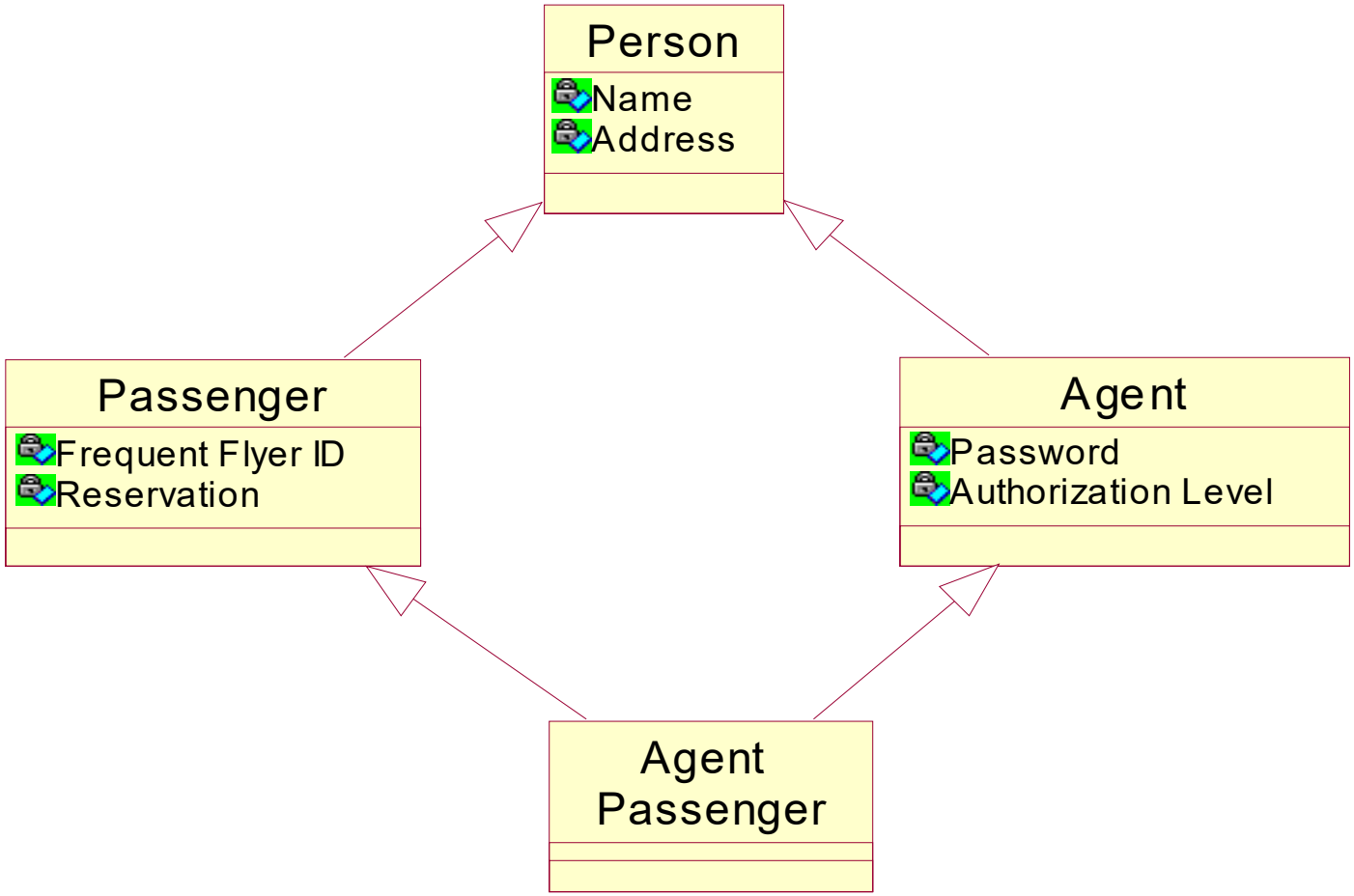
繼承(Inheritance)

- 重複(reuse)使用新功能的方法(Method)
  - 物件藉由繼承/擴充(extending)另一個物件的實做
  - 一般化(generalization)/父類別定義一般屬性和方法
  - 特殊化(specificalization)/子類別藉由額外屬性和方法實做
- 優點
  - 擴充繼承就可以實做新的功能，擴充比較簡便。
  - 容易修改或擴充可被重複使用的新功能
- 缺點
  - 白箱(White-box)重複使用：子類別可看到父類別實做細節
  - 破壞封裝性：子類別可修改父類別實做的細節
  - 父類別修改會影響到子類別
  - 從父類別實做繼承，無法在執行時期改變

Coad's Rules

- 滿足以下條件
  - 子類別表達 "is a special kind of" 不是 "is a role played by a" (上下關係不是子類別扮演父類別的一個角色)
  - 不要從一個單一的類別做多重類別的繼承
  - 子類別要擴充，而不是覆寫(overrides)或空化(nullifies)其父類別的責任
  - 應用(utility)類別不會有子類別
  - 真實的問題領域中，子類別特殊化一個角色(role)、交易(transaction)或設備(device)

Exercise

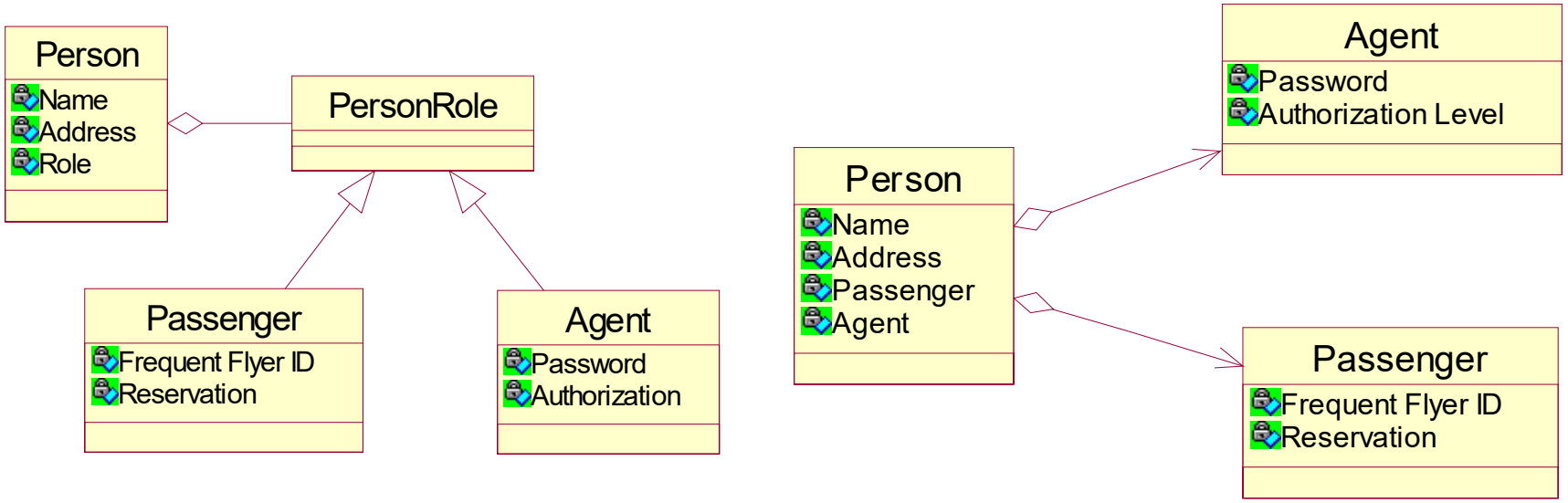

Person
Name
Address
Passenger
Frequent Flyer ID
Reservation
Agent
Password
Authorization Level
Agent
Passenger

Exercise

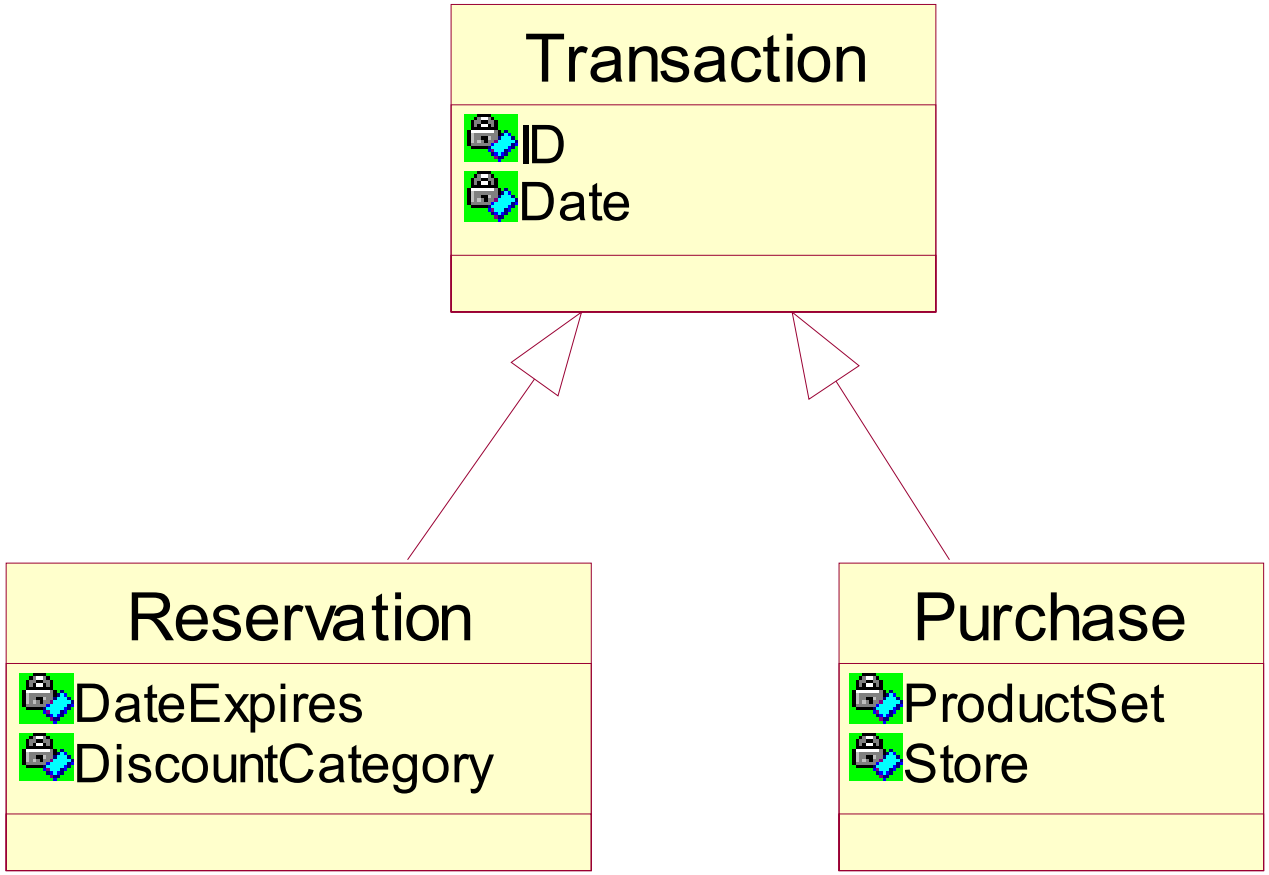
- “Is a special kind of” 不是 "is a role played by a"
  - **Fail**. Passenger是一個扮演的角色，agent 也是。
- 不需要變形
  - **Fail**. Person子類別的一個實體，在不同時間能從Passenger到Agent，然後再到Agent Passenger
- 擴充而不是覆寫或空化
  - **Pass**.
- 不是一個應用類別
  - **Pass**.
- 在真實問題領域，特殊化一個角色、交易、或設備
  - **Fail**. Person不是一個角色、交易或設備

Exercise

- "Is a special kind of" 不是 "is a role played by a" => Pass.
- 不需要變形 => Pass.
- 擴充而不是覆寫或空化 => Pass.
- 不是一個應用類別 => Pass.
- 在真實問題領域，特殊化一個角色、交易、或設備
  - Pass. PersonRole是一種角色



Exercise


Transaction
ID
Date
Reservation
DateExpires
DiscountCategory
Purchase
ProductSet
Store

繼承與組合

- 繼承與組合都是重複使用的方法
- 在物件導向發展中，繼承常常被過度使用
- 善用組合可以簡化重複使用的設計
- 繼承與組合要相輔相成

Composite Pattern

- 動機
  - 將物件組織成樹狀結構、「部份－全體」層級關係，讓外界以一致性的方式存取個別物件和整體物件
  - 圖形編輯器應用程式，常用小零件拼出複雜圖形；或將幾個小零件結合成較大群組物件，而群組物件又可當成零件使用
  - 替各種基本圖形定義類別(像Text, Line)，再定義其他作為容器用途的類別；
- 問題
  - 程式無法以一致方式對待基本圖形和容器物件

Composite Exercise

```
public abstract class Shape {
    public void draw(String fillColor);
}
public class Triangle extend Shape {
    public void draw(String fillColor) {
        System.out.println("Drawing Triangle with color "+fillColor);
    }
}
public class Circle extend Shape {
    public void draw(String fillColor) {
        System.out.println("Drawing Circle with color "+fillColor);
    }
}
```

Composite Exercise

- 畫出類別圖
- 客戶端程式如何使用此程式模組。

```
import java.util.ArrayList;
import java.util.List;
class Drawing implements Shape{
    private List<Shape> shapes
                = new ArrayList<Shape>();
    public void draw(String fillColor) {
        for(Shape sh : shapes)        {
            sh.draw(fillColor);
        }
    }
public void add(Shape s){
        this.shapes.add(s);
    }
public void remove(Shape s){
        shapes.remove(s);
    }
    public void clear(){
        System.out.println("Clearing");
        this.shapes.clear();
    }
}
```

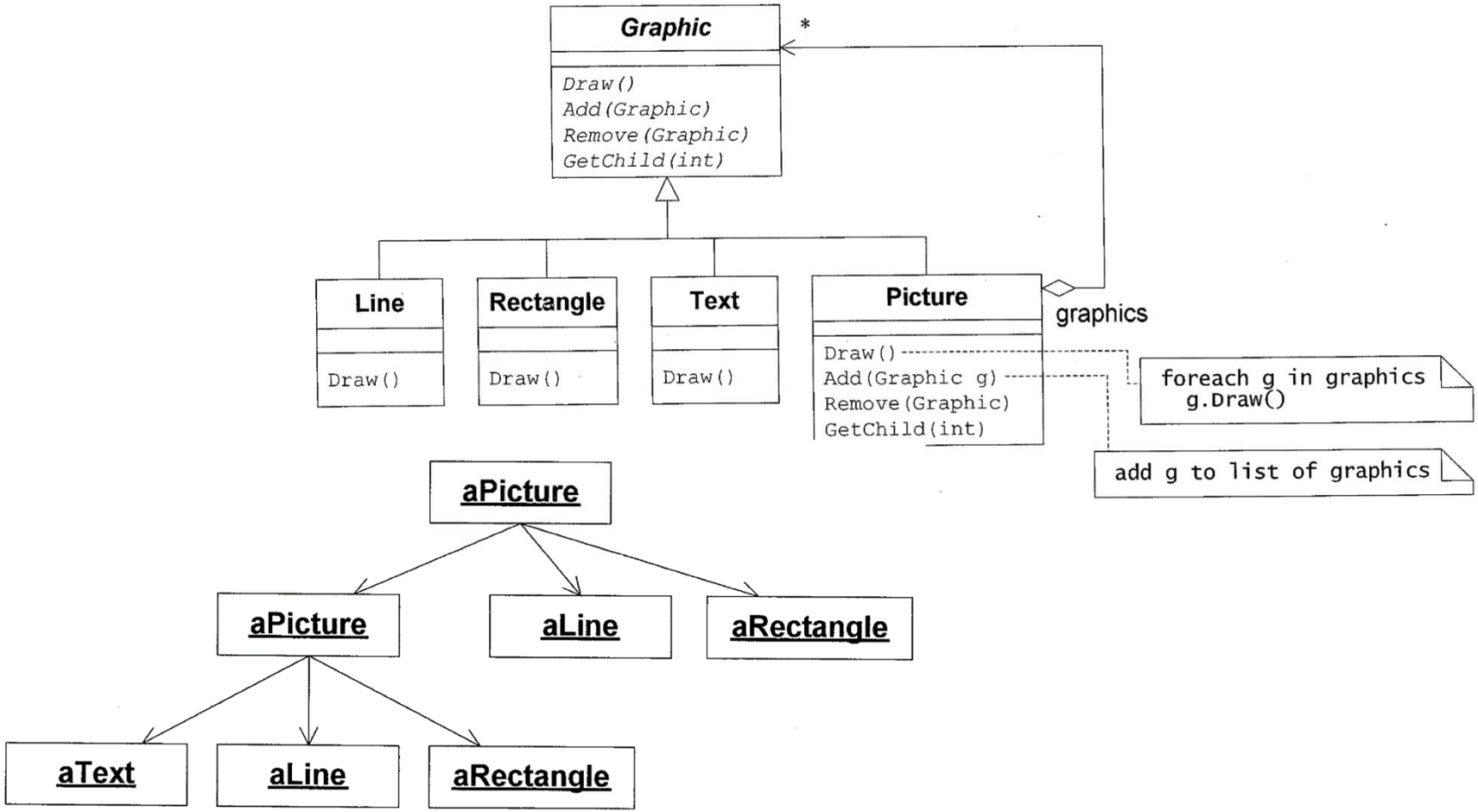
Composite Exercise

```
public class TestCompositePattern {
    public static void main(String[] args) {
        Shape tri = new Triangle();
        Shape tri1 = new Triangle();
        Shape cir = new Circle();
        Drawing drawing = new Drawing();
        drawing.add(tri1);          drawing.add(tri1);          drawing.add(cir);
        drawing.draw("Red");
        drawing.clear();
        drawing.add(tri);           drawing.add(cir);
        drawing.draw("Green");
    }
}
```

將印出甚麼結果~

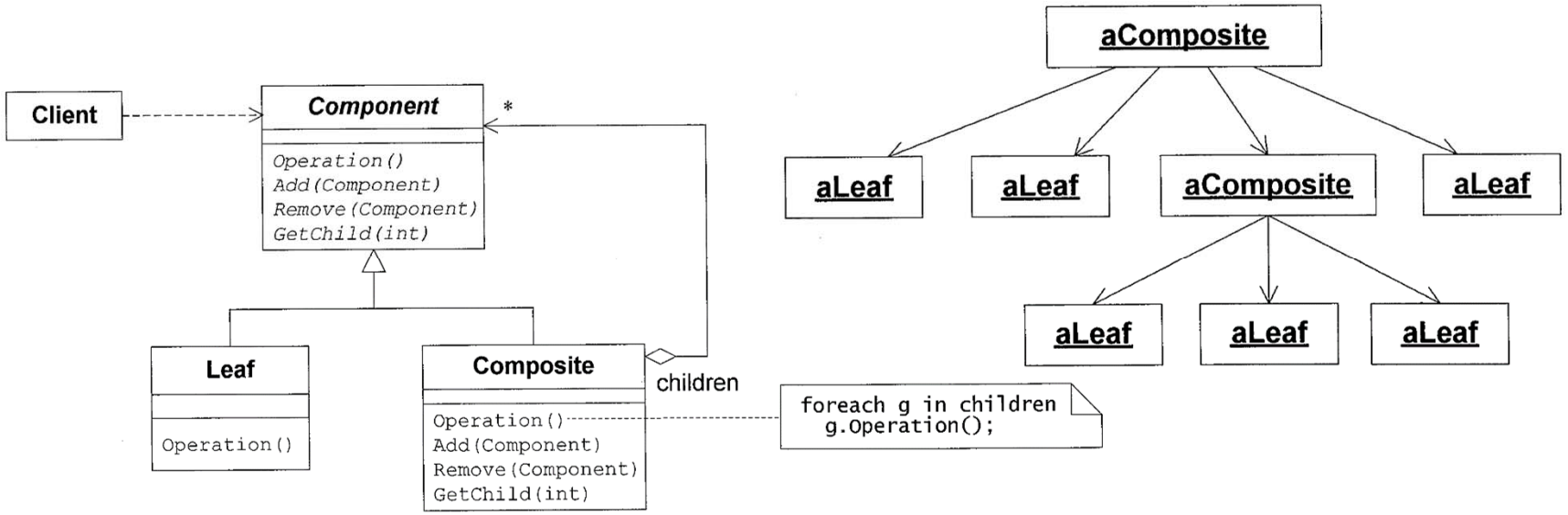
Composite

- 解決辦法



Composite

- 時機
  - 想表達出「部份-全體」的物件關係時
  - 想讓客戶碼毋須考慮基本物件與複合物件間的差異，能以一致方式處理複合結構裡的物件時
- 結構



Composite

- 參與者
  - Component (Graphic)
    - 宣告複合體內含物件之介面
    - 替所有類別所共有的操作，實作出合適的預設行為
    - 宣告存取及管理子節點的介面
    - 宣告存取父節點的介面
  - Leaf (Rectangle、Line、Text等等)
    - 代表複合結構之終端物件。不會有子節點
    - 定義基本物件的行為
  - Composite (Picture)
    - 定義含子結構的節點行為、儲存子節點
    - 實作出Component中與子節點有關的介面
  - Client: 透過Component介面操縱複合體的物件

Composite

- 合作方式
  - Client透過Component類別介面，與複合結構的物件互動。
  - 如果對象是Leaf，就直接處理。
  - 如果對象是Composite，就將訊息傳給子節點處理，在傳遞前或後也可能會做額外的事。
- 效果
  - 定義包含基本和複合物件的類別階層。
  - 簡化客戶程式碼：以一致方式處理複合結構和個別物件。
  - 更易增加新的Component類型。
  - 可能讓設計過於一般化。
  - 難以對Composite所能包含的Component類型設限，需以執行期檢驗方式來達成。

Composite Exercise

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
abstract class Employee {
   private String name;
   private double salary;
   public abstract void add(Employee employee);
   public abstract void remove(Employee employee);
   public abstract Employee getChild(int i);
   public abstract void myPrint();
   public void setName(String name) { this.name = name; }
   public void setSalary(double salary) { this.salary = salary; }
   public String getName() {    return name;    }
   public double getSalary() {   return salary;    }
   public void print() {
      System.out.println("Name ="+getName() +"\n"+"Salary ="+getSalary());
      myPrint();
   }
}
```

Composite Exercise

```
class Manager extends Employee{
   public Manager(String name,double salary){  setName(name); setSalary(salary);   }
   List<Employee> employees = new ArrayList<Employee>();
   public void add(Employee employee) { employees.add(employee);     }
   public Employee getChild(int i) {   return employees.get(i);    }
   public void myPrint() {
      System.out.println("--The management team--");
      Iterator<Employee> employeeIterator = employees.iterator();
      while(employeeIterator.hasNext()){
        Employee employee = employeeIterator.next();
        employee.print();
      }
      System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~");
   }
   public void remove(Employee employee) { employees.remove(employee);      }
}
```

Composite Exercise

```
 class Developer extends Employee{
    public Developer(String name,double salary){  setName(name);    setSalary(salary);   }
    public void myPrint() { /*leaf node is not applicable*/  }
    public void add(Employee employee) { /*leaf node is not applicable*/  }
    public Employee getChild(int i) { return null; /*leaf node is not applicable*/ }
    public void remove(Employee employee) { /*leaf node is not applicable*/  }
}
public class TestEmployee {
    public static void main(String[] args) {
        Employee manager1 = new Manager("Tom",80000);
        Employee manager2 = new Manager("Mary",70000);
        Employee developer1 = new Developer("John",50000);
        Employee developer2 = new Developer("Kevin",40000);
        manager1.add(developer1);
        manager2.add(developer2);
        manager1.add(manager2);
        manager1.print();
        manager2.print();
    }
}
```

Composite Exercise

- 請畫出類別圖

開放-封閉法則：

為擴充而開放，封閉任何修改

The Open-Closed Principle:

Open For Extension,

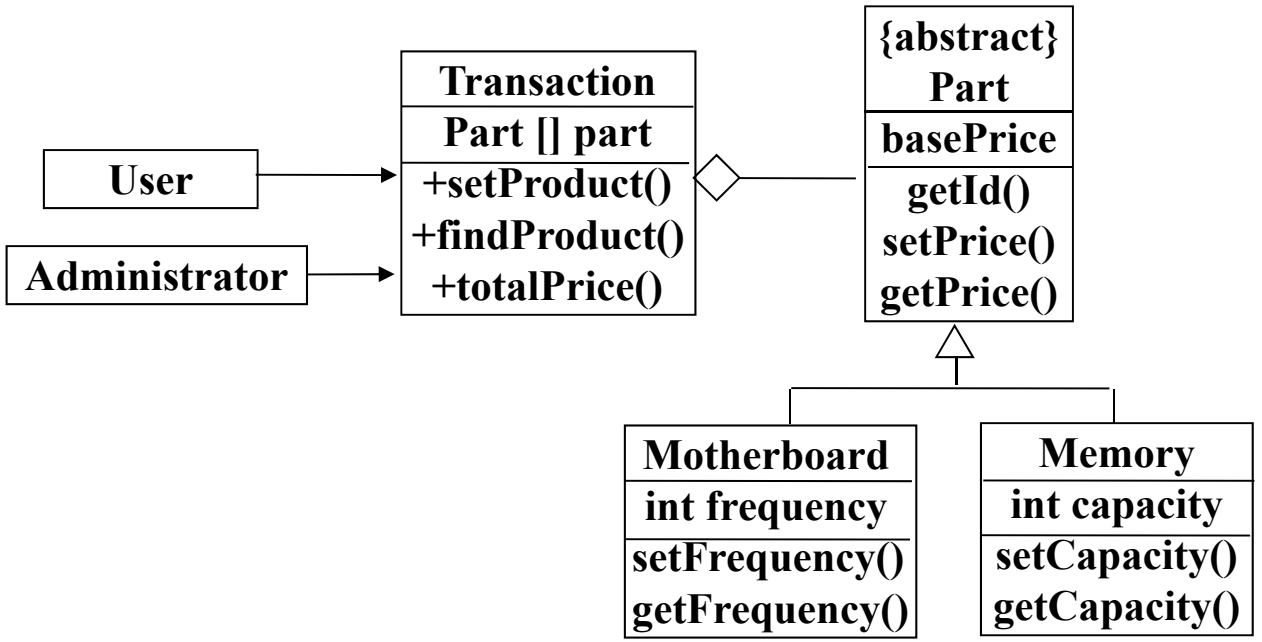
Yet Closed For Modification

開放封閉法則 1

- 為了擴充而開放
  - 模組的行為要能被擴充以增加新的功能
- 封閉修改
  - 增加新的程式碼，不要修改舊的已寫好的程式碼
- 達成方法
  - 抽象(Abstraction)、多型(Polymorphism)、繼承(Inheritance)、介面(Interfaces)
- 一個軟體系統不可能所有模組設計都符合OCP
  - 讓滿足OCP模組數量最大化，增加重複使用性及可維護性

開放封閉法則 2

- 電腦賣場，販賣各種不同電腦零件，每一位客戶一次購買許多電腦零件，計算客戶付款總價。賣場以後可能販賣其他新的電腦零件。



開放封閉法則 3

- Part是抽象基礎類別，使用多型可彈性增加新零件，不需修改存在的程式碼，此為 OCP！

```
public class Part {
    private double basePrice;
    public void setPrice(double price) {basePrice = price;}
    public double getPrice() { return basePrice;}
}
public double totalPrice(Part[] parts) {
   double total = 0.0;
   for (int i=0; i<parts.length; i++) {       total += parts[i].getPrice();      }
   return total;
}
```

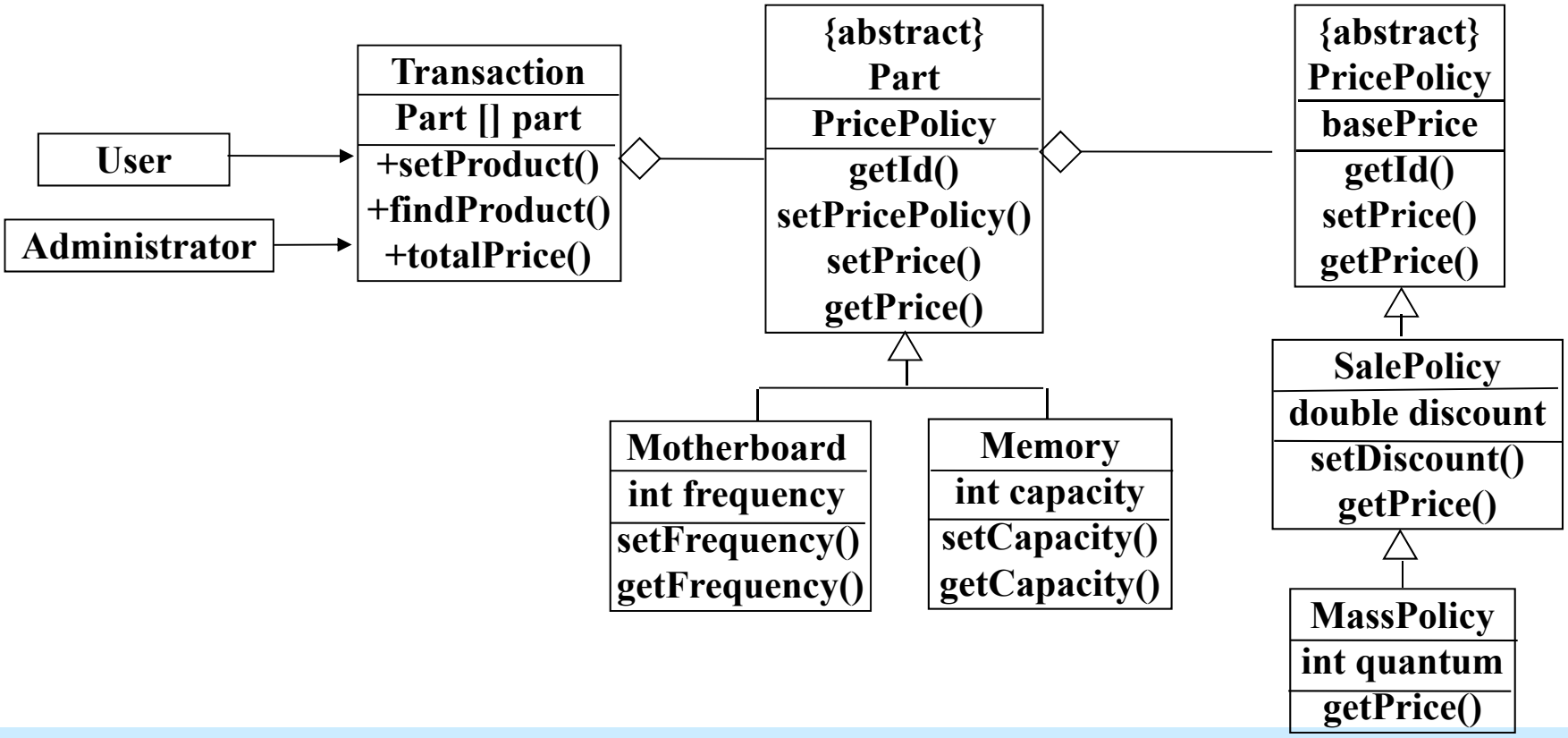
開放封閉法則 4

- 需求變更
  - 每次會計部門訂定新價格政策，就須改變 totalPrice()，或修改part子類別的價格，破壞 OCP!

```
public class ConcretePart extends Part {
   public double getPrice() {       // return (1.45 * basePrice);
      return (0.90 * basePrice); //Labor Day Sale
   }
}
public double totalPrice(Part[] parts) {
   double total = 0.0;
   for (int i=0; i<parts.length; i++) {
      if (parts[i] instanceof Motherboard)   total += (1.45 * parts[i].getPrice());
      else if (parts[i] instanceof Memory)   total += (1.27 * parts[i].getPrice());
      else   total += parts[i].getPrice();
   }
   return total;
}
```

開放封閉法則 5

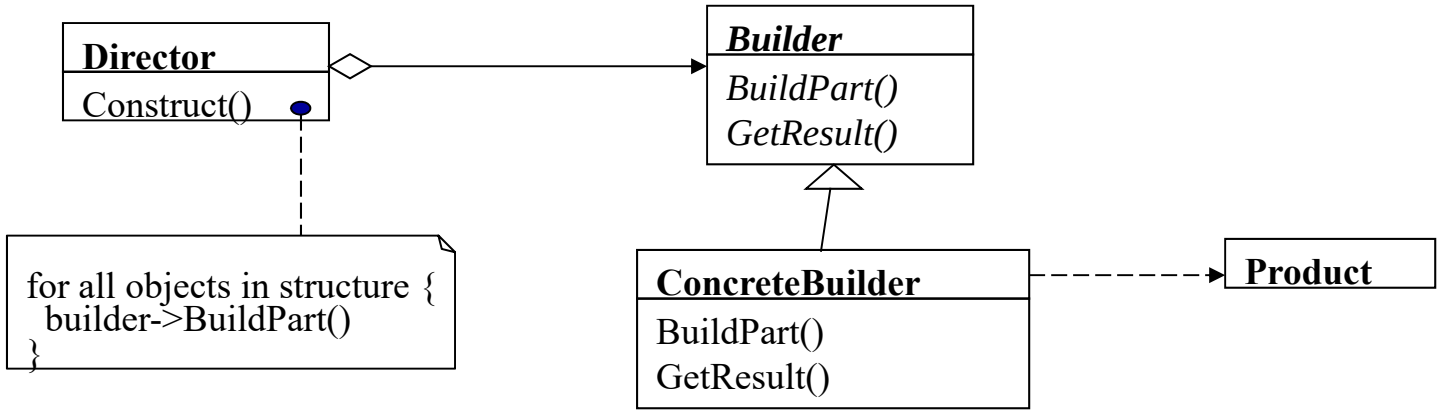
- 解決方案
  - 方案一：設計價格政策繼承架構，動態改變價格政策物件，設定價格
  - 方案二：隨時設定資料庫的價格。

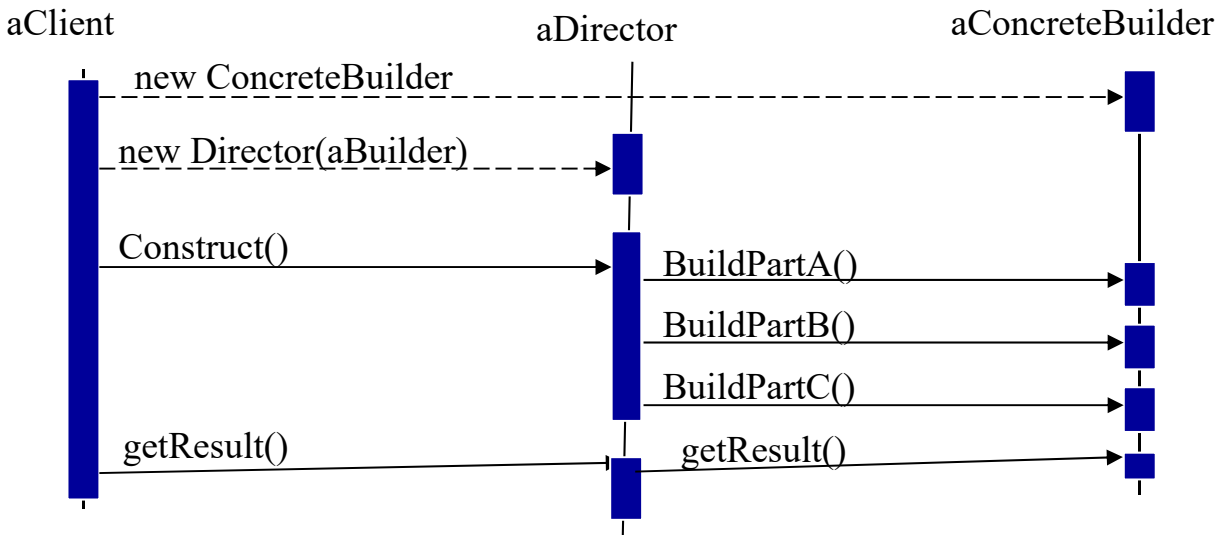


開放封閉法則 6

```
public class Part {
      private PricePolicy pricePolicy;
      public void setPricePolicy(PricePolicy policy)
         {pricePolicy =policy;}
      public void setPrice(double price) {pricePolicy.setPrice(price);}
      public double getPrice() { return pricePolicy.getPrice();}
}
public class PricePolicy {
      private double basePrice;
      public void setPrice(double price) {basePrice = price;}
      public double getPrice() { return basePrice;}
}
public class SalePrice extends PricePolicy{
      private double discount;
      public void setDiscount(double discount)
      { this.discount = discount; }
      public double getPrice() { return (basePrice * discount);}
}
```

Builder


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


aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
getResult()
getResult()

Exercise Builder

- 飲料店販賣各種咖啡與奶茶
  - 奶茶(MilkTea)製作方式
    - brew:泡紅茶
    - flavor:加入鮮奶
    - mix:灑上少許巧克力粉
  - 咖啡(Coffee)製作方式
    - brew: 研磨
    - flavor:加入奶精
    - mix:灑上少許肉桂粉
  - 一個飲料製作的abstract 類別 (Builder)
- 畫出以上類別圖
- 畫出BuilderDirector.java的完整類別圖

取代法則：

使用父類別的地方，

須能在不知道子類別的前提下使用子類別

The Liskov Substitution Principle:

Functions That Use References To Base (Super) Classes Must Be Able To Use Objects Of Derived (Sub) Classes Without Knowing It

[Barbara Liskov]

Liskov取代法則 1

- 當子類別替換基礎類別，使軟體功能不受影響，此子類別才算真正被複用，子類別才能在基礎類別上增加新的行為。
- LSP清楚指引多型！基礎(base)類別適用的地方，子類別一定適用，故子類別須包含全部基礎類別介面 。
- 違反LSP也違反OCP，因為要修改子類別的方法。
- 針對違反LSP設計時Refactoring方式，當classA錯誤繼承classB時
  - 建構新的抽象classC，作為2個具體classA, B的父類別
  - 重構為classB委派(Delegate) classA

Liskov取代法則 2

- 考慮以下Rectangle 類別:

```
public class Rectangle {
    private double width;
    private double height;
    public Rectangle(double w, double h) {
        width = w;
        height = h;
    }
    public double getWidth() {return width;}
    public double getHeight() {return height;}
    public void setWidth(double w) {width = w;}
    public void setHeight(double h) {height = h;}
    public double area() {return (width * height);
}
```

Liskov取代法則 3

- Square 類別

```
public class Square extends Rectangle {
    public Square(double s) {super(s, s);}
    public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w);
        }
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
}
```

Liskov取代法則 4

- 似乎一切完美，不過要使用如下方法檢查!

```
public class TestRectangle {
   public static void testLSP(Rectangle r) {
      r.setWidth(4.0);    r.setHeight(5.0);
      System.out.println("Width 4.0, Height 5.0" + ", Area is " + r.area());
      if (r.area() == 20.0)  System.out.println("Looking good!\n");
      else  System.out.println("What kind of rectangle is this?");
   }
   public static void main(String args[]) {
      Rectangle r = new Rectangle(1.0, 1.0);
      Rectangle s = new Square(1.0);
      testLSP(r);    testLSP(s);
   }
}
```

Liskov取代法則 5

- 測試程式的輸出：

  Width is 4.0 and Height is 5.0, so Area is 20.0

  Looking good!

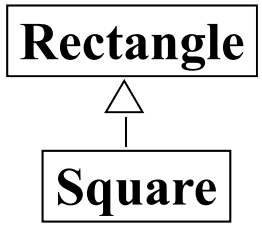
  Width is 4.0 and Height is 5.0, so Area is 25.0

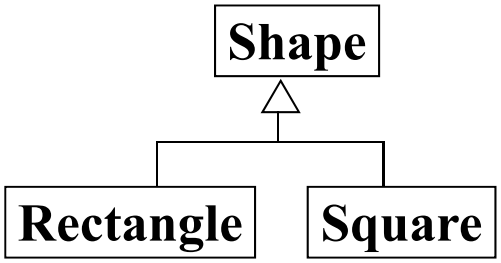
  What kind of rectangle is this??

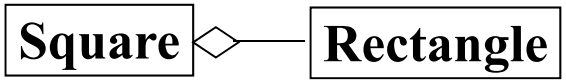
- 違反LSP!
  - Square 和 Rectangle 類別看似沒有問題，但經由傳送 Square 物件到此測試程式，暴露出違反 LSP
  - 數學上定義square 可能是rectangle。
  - 行為上，Square不是一個Rectangle，Square物件不能多型於Rectangle物件。繼承父類別的方法，須能用於子類別。

Liskov取代法則 6

- 通過LSP測試，保證父類別能使用的地方，子類別也可以使用。
- LSP 清楚指引 ISA 關係，所有子類別的行為須與父類別行為一致！
- Subtype 不能限制base type的介面功能！
- 錯誤繼承違反LSP，需Refactoring！





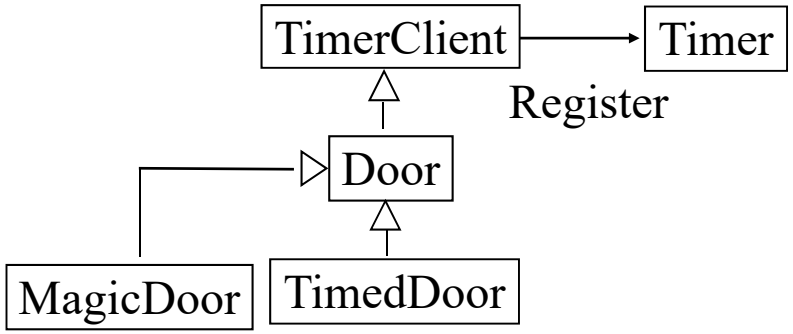


介面分離原則
Interface Segregation Principle

介面污染 1

- 需求：門開啟一段時間，需要Timer通知。

```
class Timer {
    public void regsiter(int timeout, TimerClient client) {  client.timeOut();   }
}
abstract class TimerClient {
    private Timer timer;
    public abstract  void timeOut();
    public void needNotify(int timeout, int timeOutId) {
                timer.register(timeout, this);
    }
}
abstract class Door extends TimerClient {
    public abstract void open();
    public abstract void close();
    public abstract bool isDoorOpen();
    public abstract bool enter();
}
```

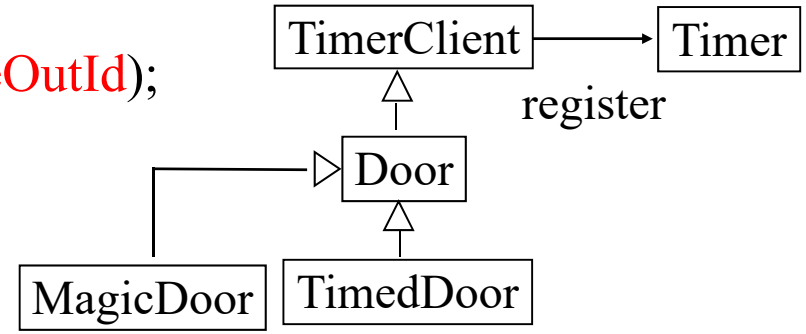


介面污染 2

- 需求變更
  - Timer使用者(TimedDoor)要登記超過一個timeout，當門一打開，即傳送登記訊息給Timer要求一個timeout通知。當timeout尚未過期，門又被開啟，就再登記一個 timeout通知。

```
class Timer {
    public void register(int timeout, int timeOutId, TimerClient client)
    { client.timeOut(timeOutId); }
}
abstract class TimerClient {
    public abstract  void TimeOut(int timeOutId);
}
```

timeout登記時，設計timeOutId識別碼，可得知要回應那個timeout要求。如此影響所有TimeClient使用者！



介面分離原則

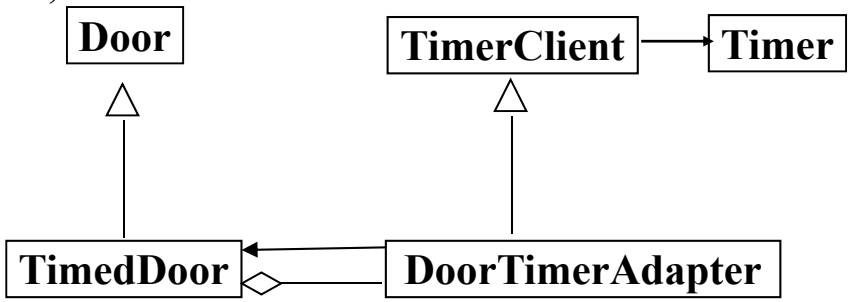
- 客戶端程式若依賴未使用到的介面，將造成介面污染/肥(FAT)介面，應被分解成幾群介面，每群服務一種客戶端
  - 客戶端程式間的依賴性應建立在最小的介面上
- ISP建議每個服務都有特定interface，依客戶型別分類，建立各種介面。
  - 若服務TimerClient的interface需變動，MagicDoor不受影響，也不用重新編譯或部署。
  - 維護程式時，現有類別或元件的介面往往會變動，迫使所有元件重新編譯及部署。
  - 應在現有物件加入新介面，而不是改變現有介面。

透過委任(Delegation)分離

```
abstract class TimerClient {
    private Timer timer;
    public abstract void timeOut(int timeOutId);
    public void setTimer(Timer timer) { this.timer = timer; }
    public void needNotify(int timeout, int timeOutId)
        {    timer.register(timeout, timeOutId, this);    }
}
class DoorTimerAdapter extends TimerClient {
    private TimedDoor timedDoor;
    public DoorTimerAdapter(TimedDoor door, Timer timer){
        timedDoor = door;
        setTimer(timer);
    }
    public void timeOut(int timeOutId) {
        timedDoor.doorTimeOut(timeOutId);
    }
}
```

透過委任(Delegation)分離

```
abstract class Door {
    private String state;
    public void close() { state = "CLOSE";  }
    public void open()  { state = "OPEN";  }
    public boolean isDoorOpen() {
        if (state.compareTo("OPEN")==0) return true;
        else return false;
    }
    public abstract void enter();
}
class TimedDoor extends Door {
   private TimerClient timerClient;
   private int timerID;
   private int timeout;
   TimedDoor(int timeout) { this.timeout = timeout; close(); timerID=0;}
   public void enter() {
      System.out.println("enter, register timer, door open");
      timerClient.needNotify(timeout, timerID++);
   }
   public void setAdapter(TimerClient timerClient) {  this.timerClient = timerClient;}
   public void doorTimeOut(int timeOutId) {   close();   System.out.println("timeout");  }
}
```



Exercise

- 畫出循序圖

```
public class TestTimerAdapter {
    public static void main(String[] args) {
        int timeout = 5;
        Timer timer = new Timer();
        TimedDoor timedDoor = new TimedDoor(timeout);
        TimerClient dTA = new DoorTimerAdapter(timedDoor, timer);
        timedDoor.setAdapter(dTA);
        System.out.println("Is door open: "+timedDoor.isDoorOpen());
        timedDoor.enter();
        System.out.println("Is door open: "+timedDoor.isDoorOpen());
    }
}
```

依賴反轉法則
The Dependency Inversion Principle

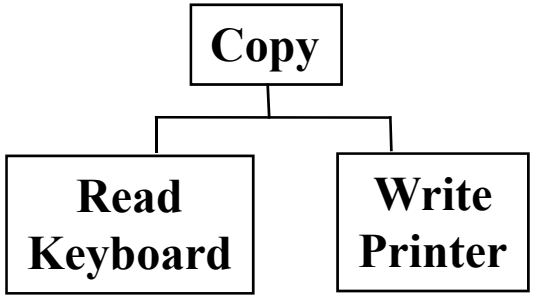
依賴反轉法則 1

- 高階模組不應依賴低階模組，兩者皆應依賴抽象模組。
  - 抽象為高階、共同策略。
  - 高階模組處理高階策略，較少處理實作細節
  - 物件導向架構，依賴抽象，不依賴含實作細節的模組。
  - 結構化設計，從上往下分解，可能造成高階模組依賴低階模組。此種依賴應被反轉。

"Copy"程式範例 1

- “Copy”模組封裝"如何複製"政策，應被重複使用

```
void Copy() {
  int c;
  while ((c = ReadKeyboard()) != EOF)
      WritePrinter(c);
}
```

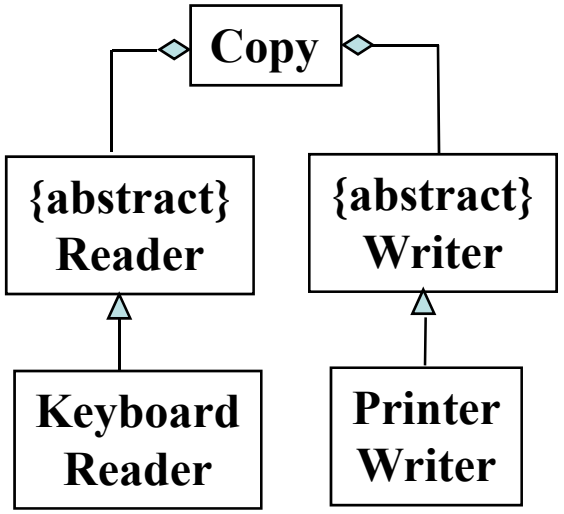


- 但"Copy"依賴"Write Printer"，因此無法在新需求下重用。
  - 更多設備如 keyboard, disk加入，因“Copy”依賴低階模組， 需加入許多if/else判斷，因此程式常須修改！

```
void Copy(outputDevice dev){
  int c;
  while ((c = ReadKeyboard()) != EOF)
      if (dev == printer) WritePrinter(c);
      else WriteDisk(c);
}
```

"Copy"程式範例 2

```
class Reader {   public int Read();   }
class Writer  {    public void Write(char);  }
void Copy(Reader r, Writer w)  {
   int c;
   while((c=r.Read()) != EOF)     w.Write(c);
}
```

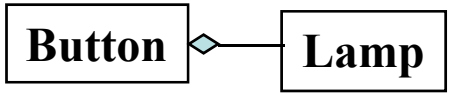



```
#include <stdio.h>
void Copy() {
   int c;
   while((c = getchar()) != EOF)   putchar(c);
}
```

- Device根據**stdio.h，是另一種依賴反轉。**

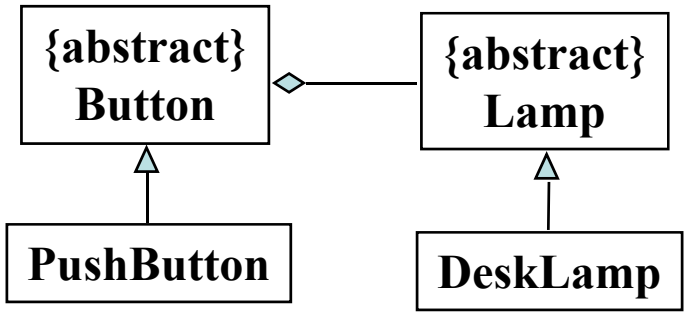
"Button"程式範例 1

```
class Lamp {
    public void TurnOn();
    public void TurnOff();
}
class Button {
    private Lamp itsLamp;
    public Button(Lamp l) {itsLamp =l; }
    public void Detect() {
        bool buttonOn = GetPhysicalState();
        if (buttonOn)   itsLamp.TurnOn();
        else     itsLamp.TurnOff();
    }
}
```



"Button"程式範例 2

```
abstract class Lamp {
    public void TurnOn();
    public void TurnOff();
}
abstract class Button {
    private Lamp  lamp;
    public Button(ButtonClient b) {
        lamp = b;
    }
    public void Detect() {
        bool buttonOn = GetState();
        if (buttonOn)      lamp.TurnOn();
        else               lamp.TurnOff();
    }
    public bool GetState();
}
class DeskLamp extends Lamp {
    public  void TurnOn();
    public  void TurnOff();
}
class PushButton extends Button {
    public PushButton(Lamp b);
    public bool GetState();}
```



依賴抽象(Abstraction)

- 抽象層含宏觀和重要商務邏輯，實做層含實作演算法
  - 錯誤設計讓抽象層依賴實做層，DIP可倒轉此現象，讓實作改變時，商業邏輯無須變動。
  - 具體(Concrete)類別應只實作介面和抽象類別中的抽象方法，不應設計多餘方法。
  - DIP假設所有具體類別都是會變更的
    - 例外：某些實做類別相當穩定，不需設計抽象類別，e.g. String。
- 建構object的方式違背DIP
  - List employees = new Vector();
  - 可使用abstract factory解決，但使用abstract factory會產生過多類別

Exercise Template 相同行為

- 人(People)帶寵物
  - 貓和狗有相同行為
    - 重複程式碼
  - 帶不同寵物，需變更play()

```
class Dog {
   private String _name;
   public Dog(String name) { _name = name;}
   public void eat() { System.out.println(name +"餵食"); }
   public void walk() {System.out.println(name+"帶去散步");}
   public void sleep() {System.out.println(name+"要睡覺");}
}
class Cat {
   public Cat(String name) { _name = name;}
   public void eat() { System.out.println(name +"餵食"); }
   public void walk() {System.out.println(name+"帶去散步");}
   public void sleep() {System.out.println(name+"要睡覺");}
}
```

```
class People {
   public void Play() {
      CallMyDog();
      // 變更帶寵物
      //CallMyCat();
   }
   private void CallMyDog(){
      Dog dog = new Dog("D");
      dog.eat();
      dog.walk();
      dog.sleep();
   }
   private void CallMyCat(){
      Cat cat = new Cat("C");
      cat.eat();
      cat.walk();
      cat.sleep();
   }
}
```

Exercise Factory Method

- 設計出寵物、貓、狗、People、PetFactory類別圖

```
abstract class Pet {
    private String _name;
    public void eat() { System.out.println(name +"餵食"); }
    public void walk() {System.out.println(name+"帶去散步");}
    public void sleep() {System.out.println(name+"要睡覺");}
}
class Dog extends Pet {
    public Dog(String name) { _name = name; }
}
class Cat extends Pet {
    public Cat(String name) { _name = name; }
}
```

```
public class PetFactory {
    public enum PetType {Dog, Cat}
    public static Pet CreatePet(PetType pType) {
        switch (pType)  {
            case PetType.Dog:
                return new Dog();
            case PetType.Cat:
                return new Cat();
            default:
                return null;
        }
    }
}
```

Exercise Factory Method

- 工廠模式
  - 增加新寵物 class Bird，不須修改 People、Pet、Dog、Cat。
  - 如何修改PetFactory的swicth case？
- 缺點
  - 還是需修改程式。
- 解決方法
  - 1.使用 DI實作 IoC。
  - 2.Abstract Factory

```
public void Play(PetFactory.PetType pType) {
    CallMyPet(pType);
}
private void CallMyPet(PetFactory.petType pType){
    Pet pet = PetFactory.CreatePet(pType);
    pet.eat();
    pet.walk();
    pet.sleep();
}
```

Abstract Factory農場系統 I 1

- 一個農場公司，專門銷售各類水果，包括葡萄Grape、草莓Strawberry、蘋果 Apple
- 水果與其他植物不同，水果是可採食。水果介面包括：種植plant()，生長grow()及收穫harvest()。蘋果是多年生植物，多出treeAge，描述蘋果樹的樹齡。葡萄分有籽和無籽兩種。
- Q:
  - 建立各種水果都適用的介面，與農場其他植物區分。
  - 畫出繼承架構類別圖。

Abstract Factory農場系統 I 2

- 農場園丁FruitGardener類別
  - 根據客戶端要求，建構不同的水果物件-Apple, Grape或Strawberry。
  - 若接到不合法要求，會拋出BadFruitException例外。
- Q: 畫出FruitGardener類別，以及其method的note。

農場系統I3

- 工廠類別集中所有產品建構邏輯，形成無所不知的全能類別，有人稱God Class。
- Problem:
  - 當產品類別有不同介面時，工廠類別需判斷什麼時候建構某種產品。如此和對哪種具體產品的判斷邏輯混合在一起，使系統進行功能擴展時較為困難。
  - 當產品類別有複雜的多層次層級結構時，工廠類別只有它自己。
  - 靜態方法無法由子類別繼承，因此工廠角色無法形成繼承層級結構。

農場系統 II

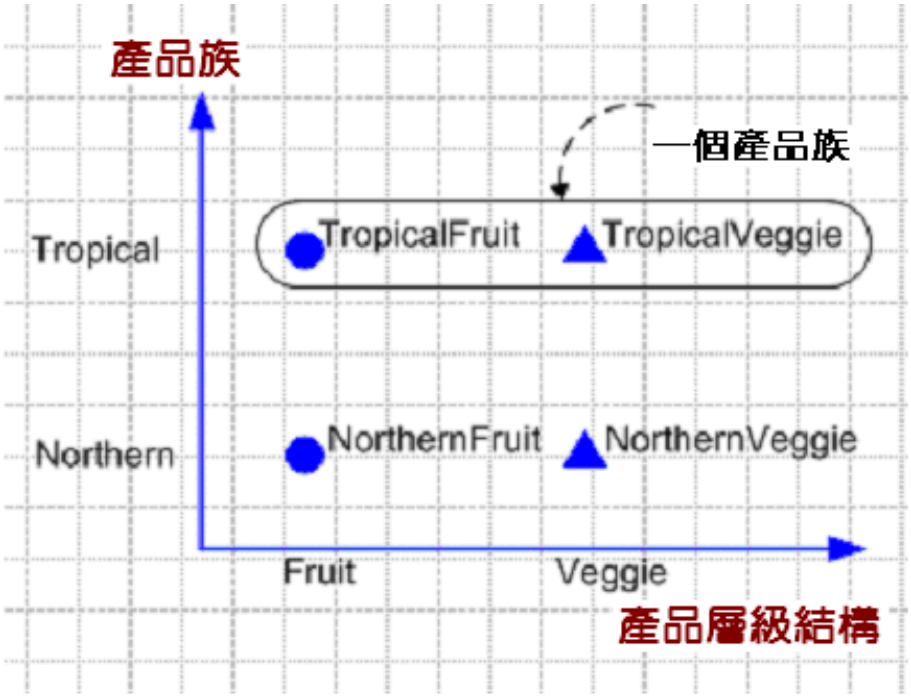
- 農場規模變大，管理專業化。每種農作物都有專門園丁管理，形成規模化和專業化生產。
  - 蘋果由蘋果園丁負責，草莓有草莓園丁負責。
  - 蘋果園丁、草莓園丁實作抽象的“水果園丁”介面。
- Q
  - 增加畫出園丁繼承架構類別圖

農場系統 III 1

- 農場公司新發展 - 引進塑膠大棚技術，在大棚裏種植熱帶（Tropical）和亞熱帶的水果和蔬菜。
  - 系統中，產品分成兩個結構：水果Fruit和蔬菜Veggie。
  - 系統產品分成兩個層級結構：Fruit, Veggie，及產品族：Tropical, Northern。
- Q
  - 畫出修改的水果類別架構圖

農場系統 III 2

- 四個點代表熱帶水果TropicalFruit、熱帶蔬菜TropicalVeggie、北方水果NorthernFruit及北方蔬菜NorthernVeggie。



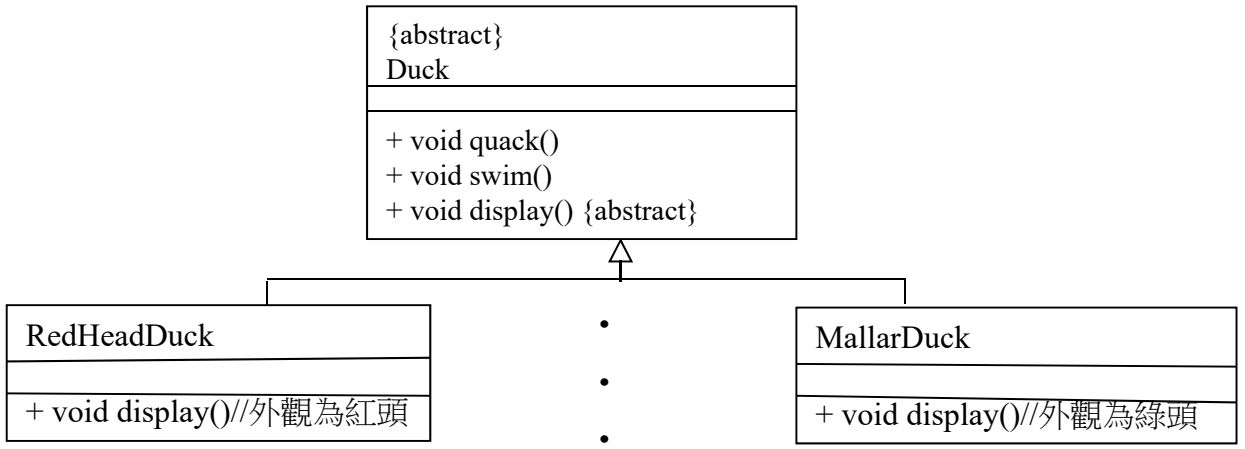
農場系統 III 3

- 種在田間的北方作物與種在大棚的熱帶作物，分屬兩個產品族。北方作物要種植一起，大棚作物要另外種植一起。
  - 使用一個工廠族封裝它們的建構過程。其層級結構應與產品族層級結構完全平行
- Q: 畫出水果與園丁類別架構圖

```
public class NorthernGardener implements Gardener {
  // 水果的工廠方法
  public Fruit createFruit(String name) {
      return new NorthernFruit(name);
   }
  // 蔬菜的工廠方法
  public Veggie createVeggie(String name) {
      return new NorthernVeggie(name);
   }
}
```

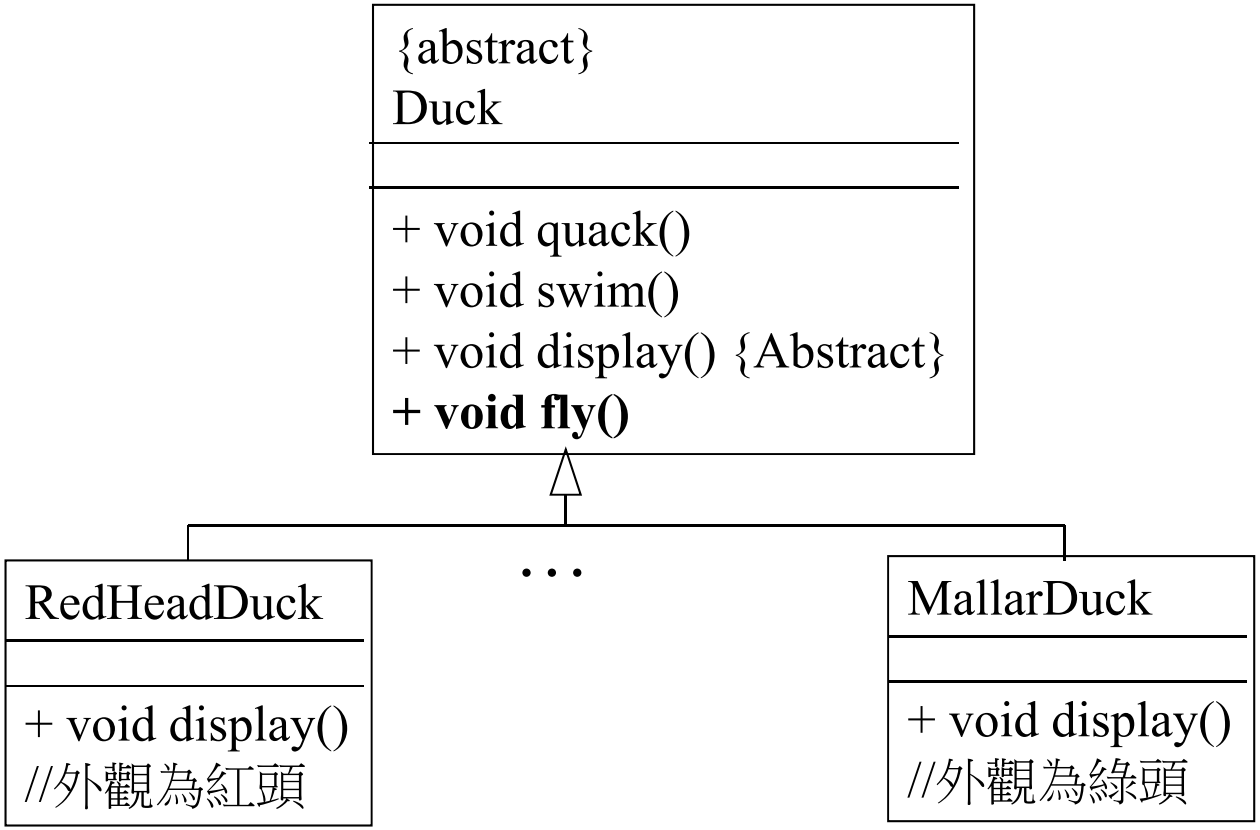
Template Pattern

- X公司做一套成功的模擬鴨子遊戲 SimUDuck
  - 遊戲中出現各種鴨子，一邊游泳戲水，一邊呱呱叫。
  - 系統設計使用 Object Orientation 技術，設計鴨子的Superclass，讓各種鴨子繼承。
  - 所有鴨子都會呱呱叫，也會游泳，所以Superclass設計quack(), swim() operation.
  - 由於每一種鴨子的外觀都不同，所以display是一個abstract operation。每個subclass都要實作display()



Fly Duck

- 公司主管John：新版需要會飛的鴨子
  - 軟體工程師Ken 保證，使用OO，只要一個星期



Fly Duck Problem

- 主管John對Ken：對客戶demo程式時，看到一隻會飛的「橡皮鴨子」，這是你開的玩笑嗎？你是否要去逛逛人力銀行網站。
- Ken忽略，並非所有Duck都會飛，在Duck加入fly() 時，某些subclass也具有此不當行為 - SimUDuck程式中有會飛的無生物。
- 對程式做局部修改，影響層面可能不只局部（會飛的橡皮鴨）。

Fly Duck Solution

- 問題
  - 不是所有Duck都會飛。
- 需求變更：橡皮鴨不會呱呱叫，只會吱吱叫。
- 需求增加：加入誘餌鴨(Decoy Duck) -假鴨，不會飛也不會叫。
- 解決方案
  - RubberDuck，將quack()覆寫成吱吱叫。將fly()覆寫成什麼事都不做。 fly() { // do nothing }
  - 誘餌鴨
    - fly() { // do nothing }
    - quack() { // do nothing }
    - display() { //誘餌鴨 }

Inheritance Problem

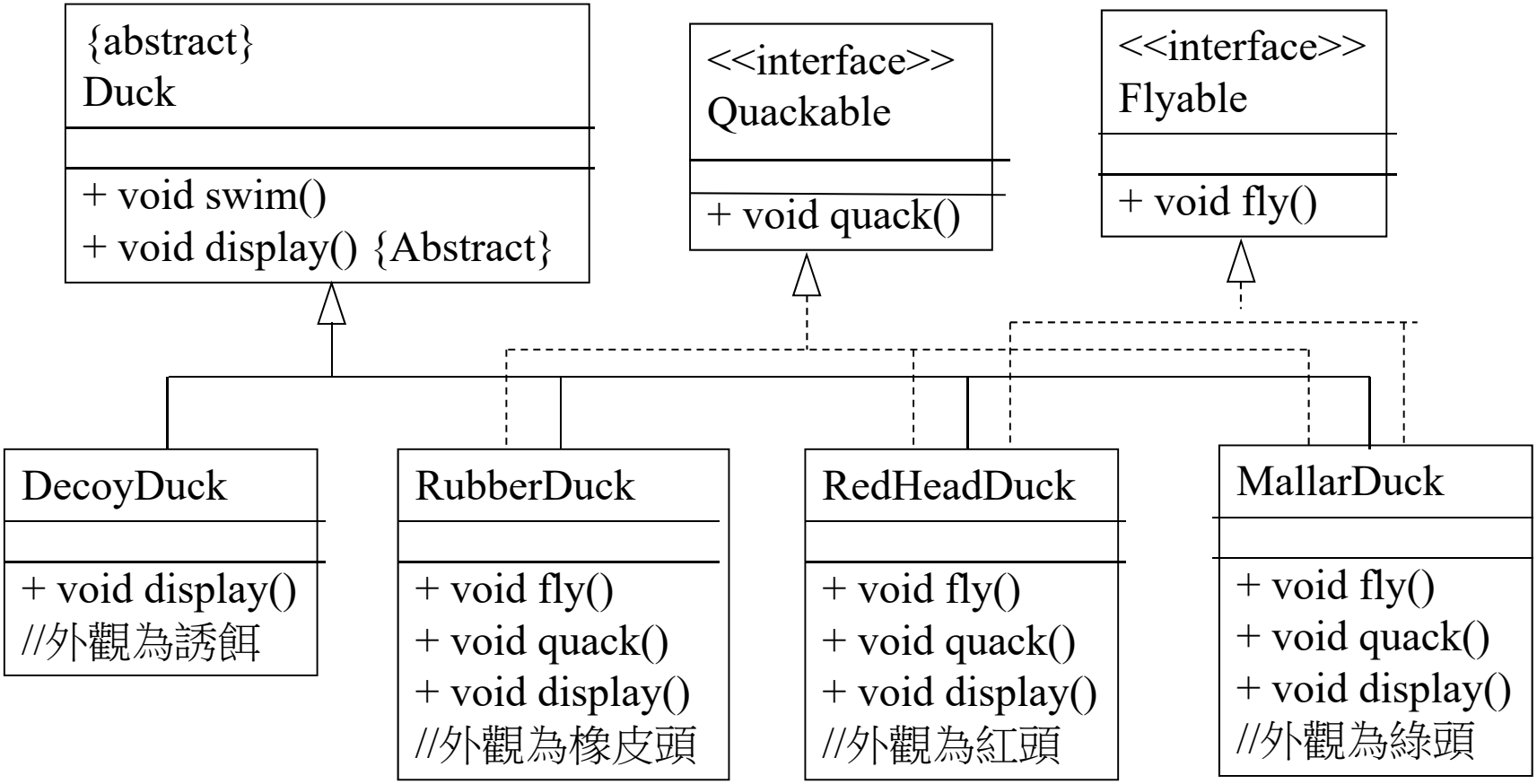
- 主管John：以後每六個月會有新Duck產品，新產品新的能力都還未定。
- Ken知道未來當有新Duck時，要被迫檢查行為是否適當，且可能重新覆寫fly()和quack()，這是無盡惡夢的開始。

Interface Solution 1

- Ken想到利用interface，
  - 將fly()取出，做成Flyable interface，只有會飛的鴨子才implement此方法。
  - 同樣設計Quackable interface，因為不是所有的鴨子都會叫。

```
class RubberDuck extends Duck implements Quackable{
    void quack(){.....}
    void display(){….}
}
```

Interface Solution 2



Exercise

- 主管John
  - 這是不好的方法，如果為新Duck 覆寫幾個方法很累人，interface 解決設計會造成以前所有的Duck都要重新實作fly()與quack()！
- 問題
  - 並非所有Duck都有fly()與quack()的行為，所以繼承不是好方法。
  - 而Flyable與Quackable可解決一部份問題 - 不會再有會飛的橡皮鴨，卻造成程式碼無法再利用，這只能算是從一個惡夢跳到另一個惡夢。
  - 甚至，以前會飛的Duck，飛行動作可能有多種變化。

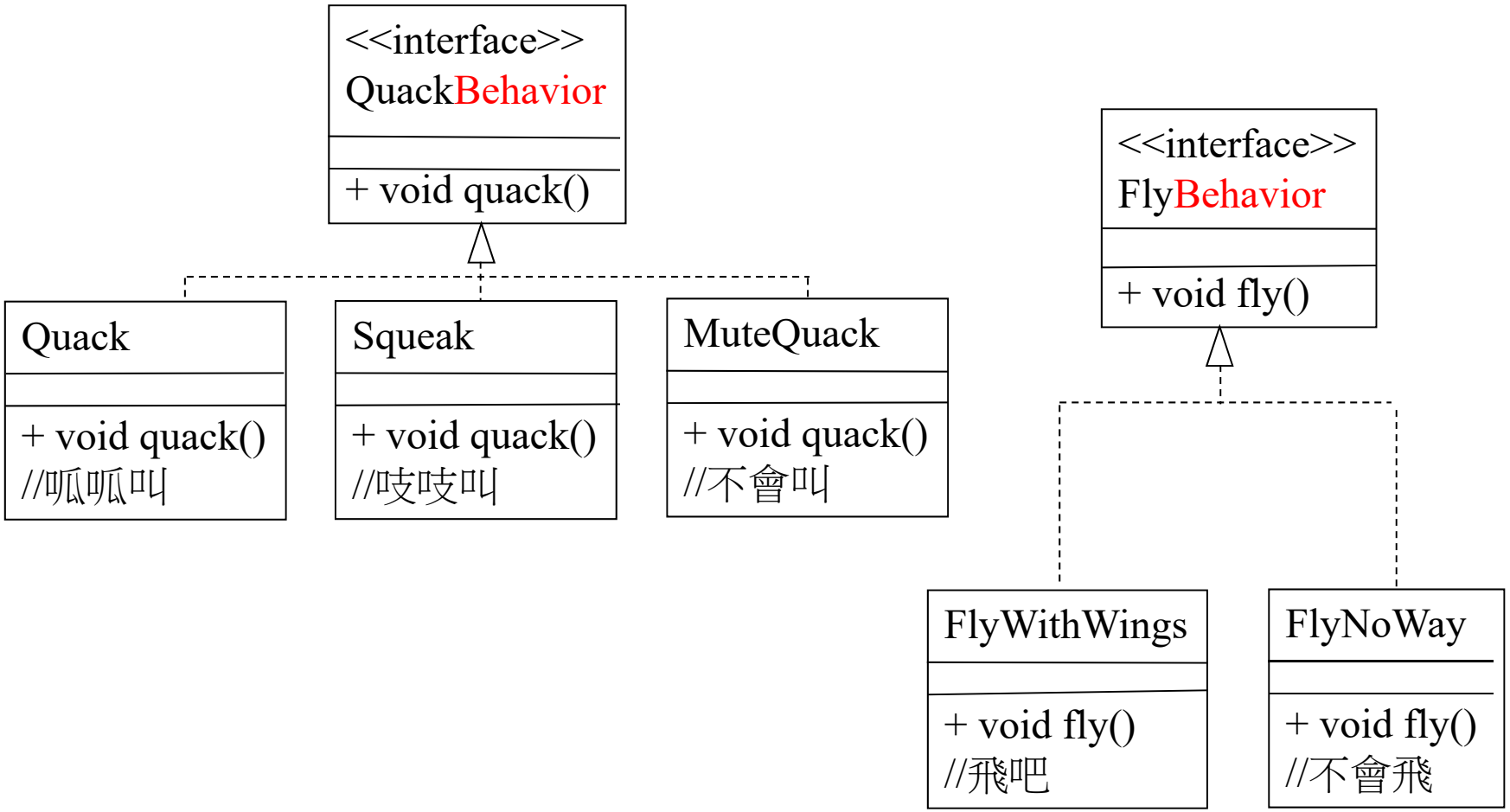
Exercise 軟體發展之需求變更

- 不管當初軟體設計多好，一陣子後軟體總需成長與改變。
  - 繼承缺失：改變Duck 父類別行為會影響所有種類Duck。
  - 介面問題：介面不具程式碼，無法達到程式碼再利用。
    - 無論何時需要修改某個行為，必須追蹤並修改每一個定義此行為的類別。
- 得到設計守則
  - 找出程式中可能需要更動之行為，獨立出來並封裝成類別階層。
  - 不要、和不需要更動的程式碼混在一起，以便以後可彈性地擴充此行為，而不影響不需要更動的其它部份。

Exercise Strategy 1

- Duck，除fly()和quack()會隨種類不同而改變外，沒有其他經常變動或修改的部分。
  - 把這二個行為從Duck分開，建立一組新類別代表每個行為。
- 使Duck行為有彈性：指定"行為"到Duck實體
  - e.g. 指定綠頭鴨的 fly 行為。
  - 在Duck 類別中包含設定行為的方法，可在「執行期」動態「改變」綠頭鴨的飛行行為。
- 多設計interface，並在subtype(實做的類別)實現其operation/method
  - 設計二個介面：FlyBehavior與QuackBehavior
    - 介面方法預設為 public abstract，變數預設為public static final。
    - 鴨子，有會呱呱叫的，有會吱吱叫的，有不會叫。設計三個類別，皆實作QuackBehavior：

Exercise Strategy 2



Exercise Strategy 3

- 整合Duck的行為
  - 分離Duck變動的行為，並設計class實作之，整合原本的Duck class
- Problem
  - 用一個class代表行為？class不是應代表某種東西，具備attribute/operation？
  - 需求增加：在SimUDuck系統中，加入一隻火箭動力的Duck，該怎麼做？
- Solution
  - Duck將fly()和quack()的動作，委任(delegate)行為類別處理，而非使用自己類別或子類別定義的方法

Exercise Strategy 4

- Duck加入二個instance variable，type 為FlyBehavior, QuackBehavior
  - type為interface - supertype，而非實作interface的class (Squeak, FlyNoWay...)
  - Advantage: 每個instance variable會利用polymorphism在run-time取到正確的行為
- 將Duck class與所有sub class中的fly()與quack()移除，因這些行為已被移到FlyBehavior與QuackBehavior class中。
- 將Duck class加上performFly()與performQuack()二個方法，取代原本的fly()與quack()。
- …

Exercise Strategy 5

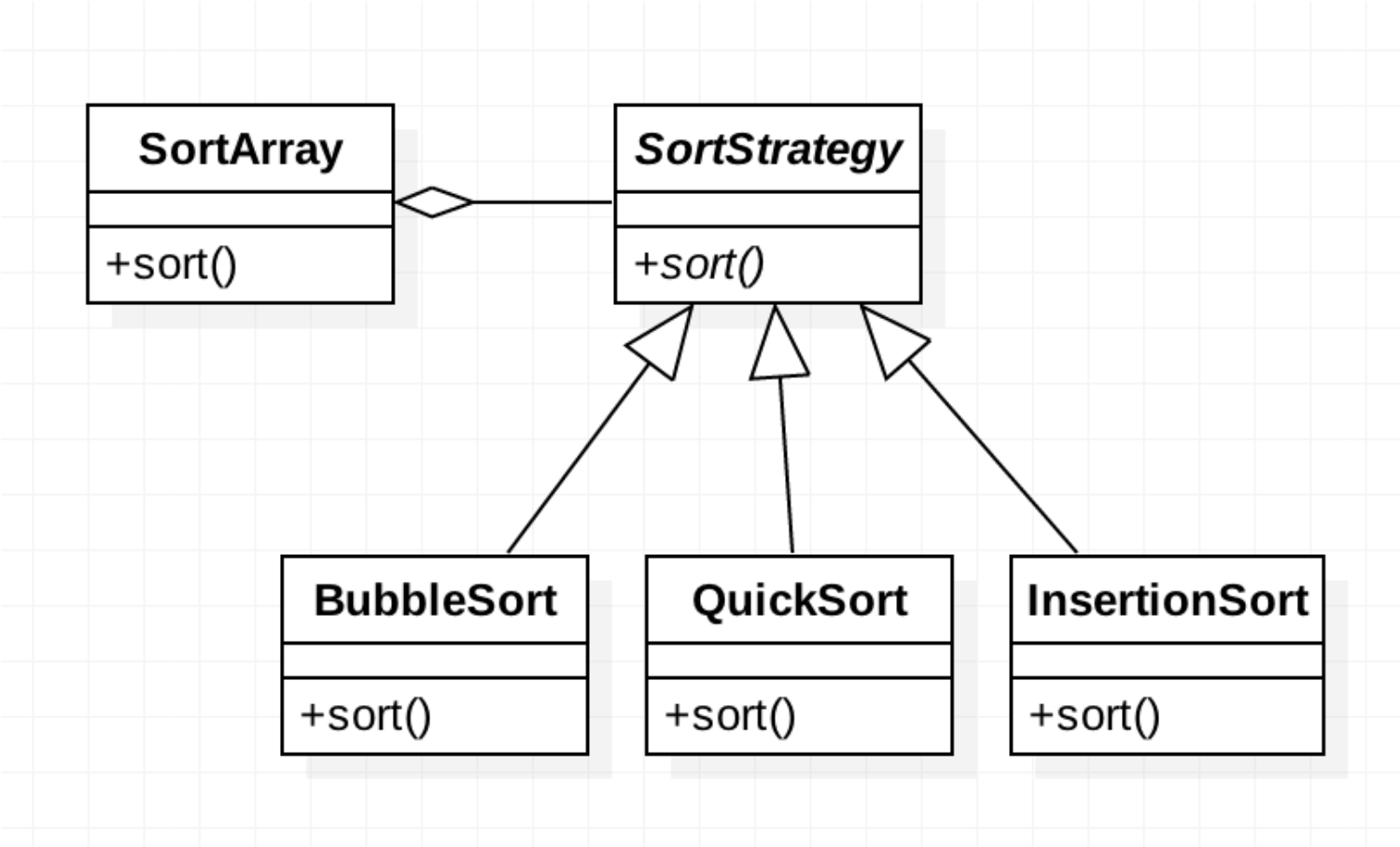
- 實作performQuack()的內容

```
public abstract class Duck {
        QuackBehavior quackBehavior; //每個鴨子都參考一個實作
        //QuackBehavior介面的物件
//. . .
public void setQuackBehavior(QuackBehavior q) {
        quackBehavior = q;
}
public void performQuack() {
        quackBehavior.quack(); //不親自處理呱呱叫的行為，而是
        //「委任」quackBehavior物件幫Duck處理
        //呱呱叫
}
```

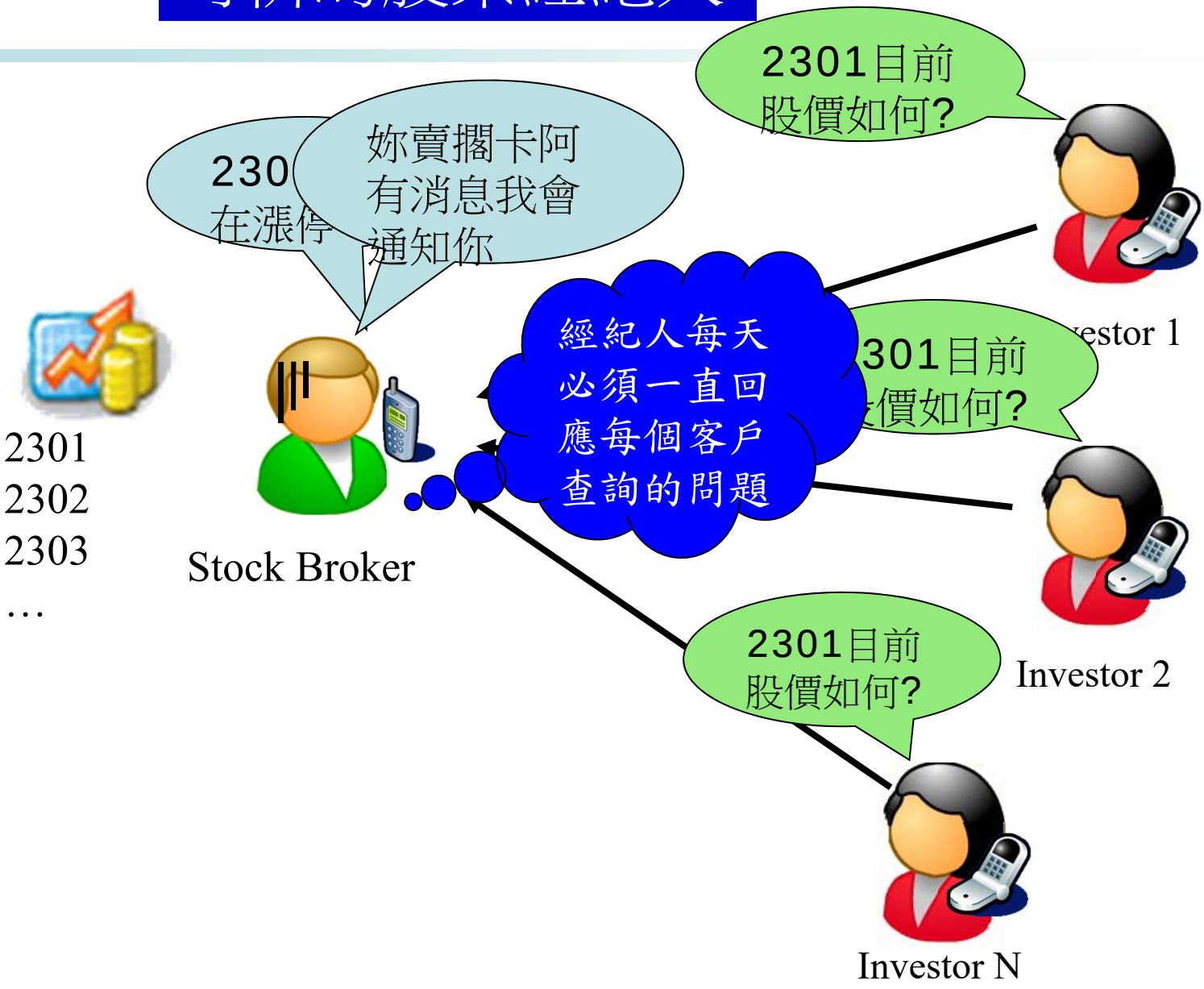
- 若需求增加：系統中加入一隻火箭動力Duck，怎麼做？

Exercise Strategy 6

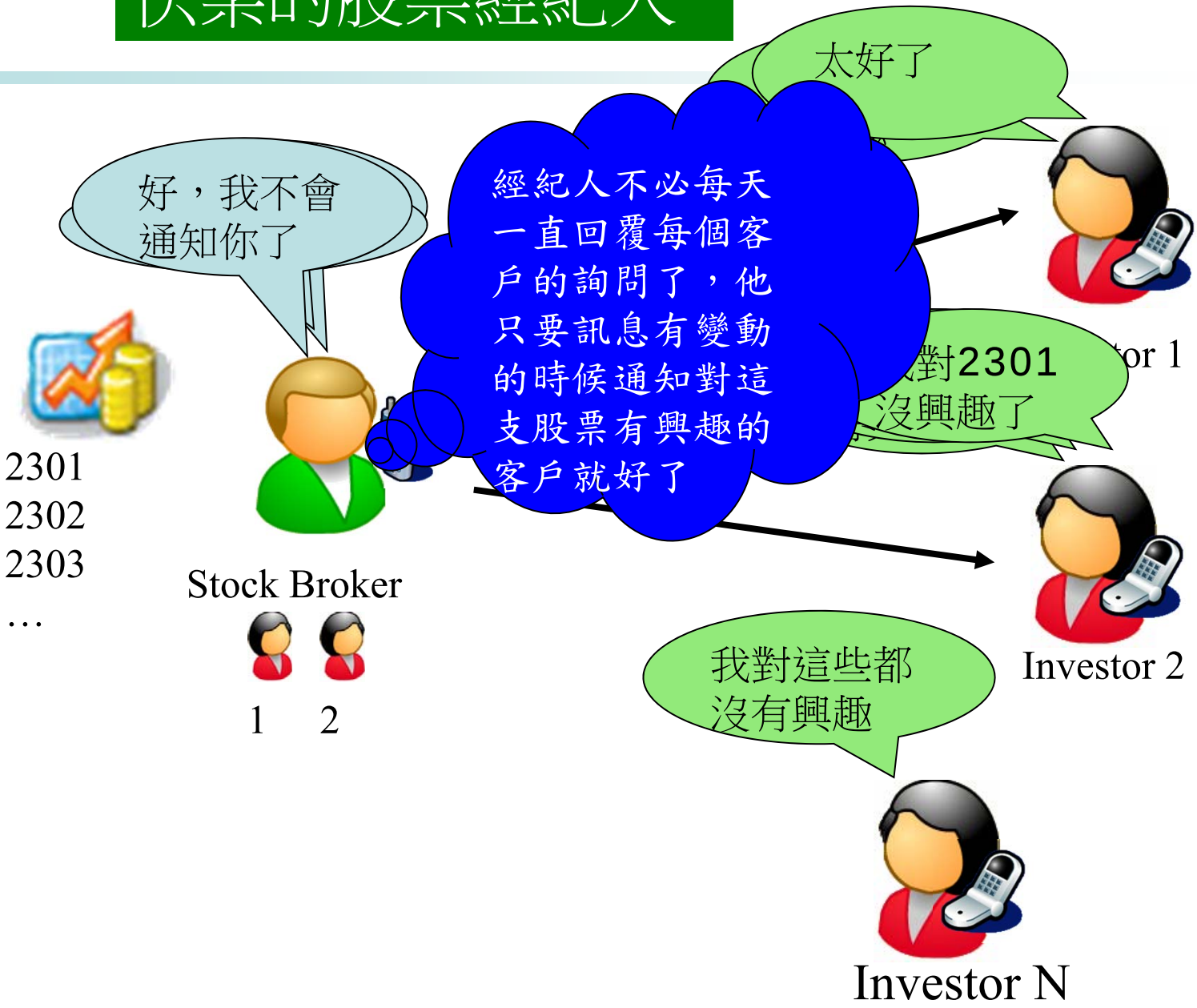
- 以下動態流程如何運作



可憐的股票經紀人


2301目前
股價如何?
妳賣擱卡阿
有消息我會
通知你
230
在漲停
經紀人每天
必須一直回
應每個客戶
查詢的問題
301目前
價如何?
2301
2302
2303
…
Stock Broker
Investor 1
Investor 2
2301目前
股價如何?
Investor N

快樂的股票經紀人


太好了
好，我不會通知你了
經紀人不必每天一直回覆每個客戶的詢問了，他只要訊息有變動的時候通知對這支股票有興趣的客戶就好了
對2301沒興趣了
or 1
2301
2302
2303
…
Stock Broker
1
2
Investor 2
我對這些都沒有興趣
Investor N

發生什麼事

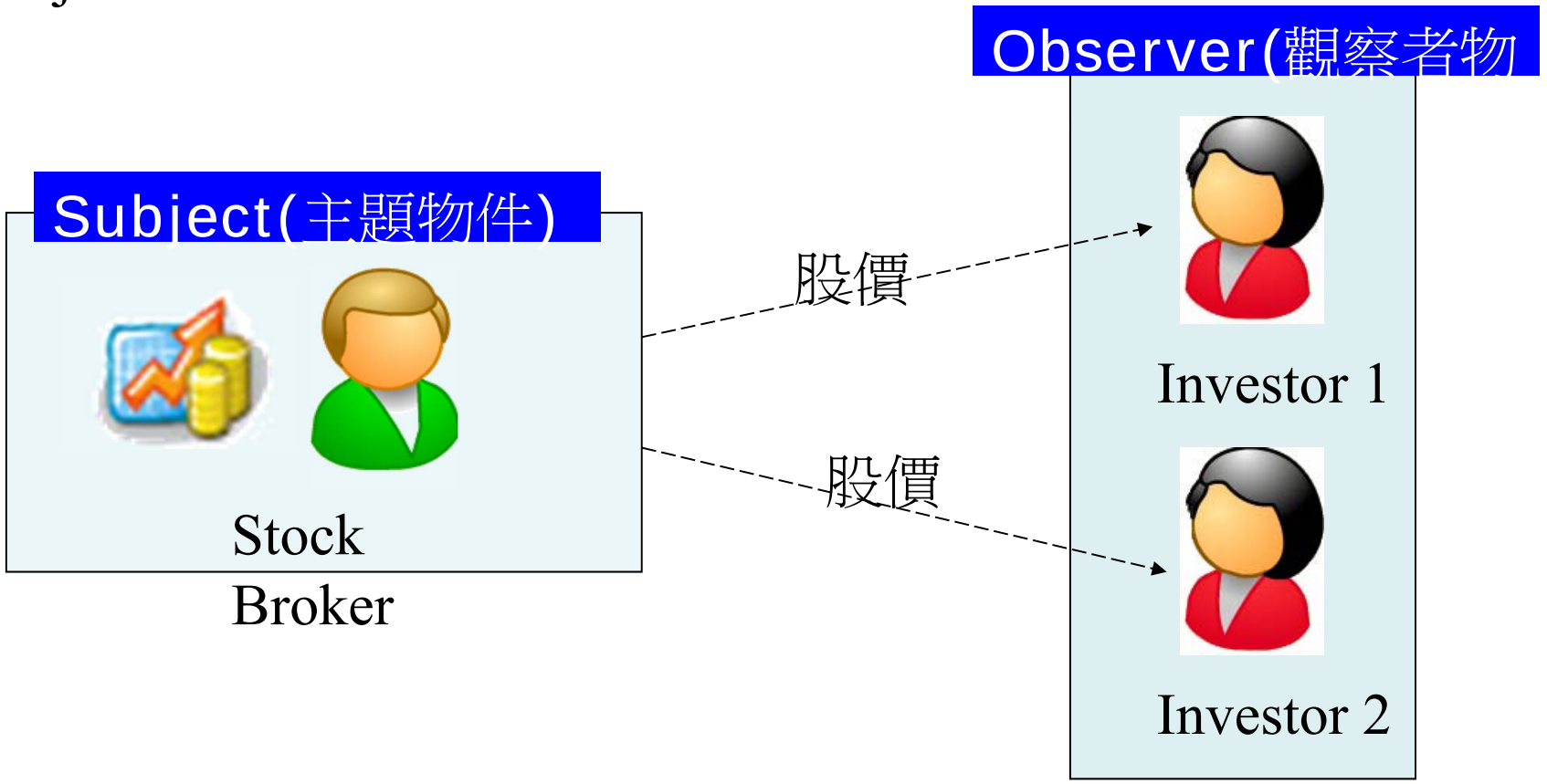
- 不希望一直去詢問有沒有新鮮事。
  - 當有興趣的事情發生時，可收到通知，當對這件事情沒興趣，就不再收到通知。
  - 隨時可加入對這件事情有興趣的人，不會影響這件事情的處理方法。
- 需求描述
  - 建立一個股票資訊系統，目前有免費會員和VIP尊榮會員可看到股票資訊，不同會員等級畫面顯示的狀況不一樣。當股價變動時，會員畫面要跟著改變。另外，須允許未來可擴充不同會員等級，甚至提供股票資訊給其他廠商。

觀察者樣式 1

- 定義物件之間一對多的關係。
- 當一個物件改變狀態，其他相依物件都會收到通知並自動更新訊息。
- 以報紙的訂閱為例
  - 報社的業務就是出版報紙
  - 妳向某家報社訂閱報紙，只要有報紙出版，就會送一份給妳，只要妳在訂閱戶名單上，就會一直收到報紙。
  - 當妳不想再看報紙的時候，取消訂閱，報社就不會再送報紙給妳。
  - 只要報社還在營運，就一直會有人或單位向他們訂閱報紙或取消訂閱報紙。

觀察者樣式 2

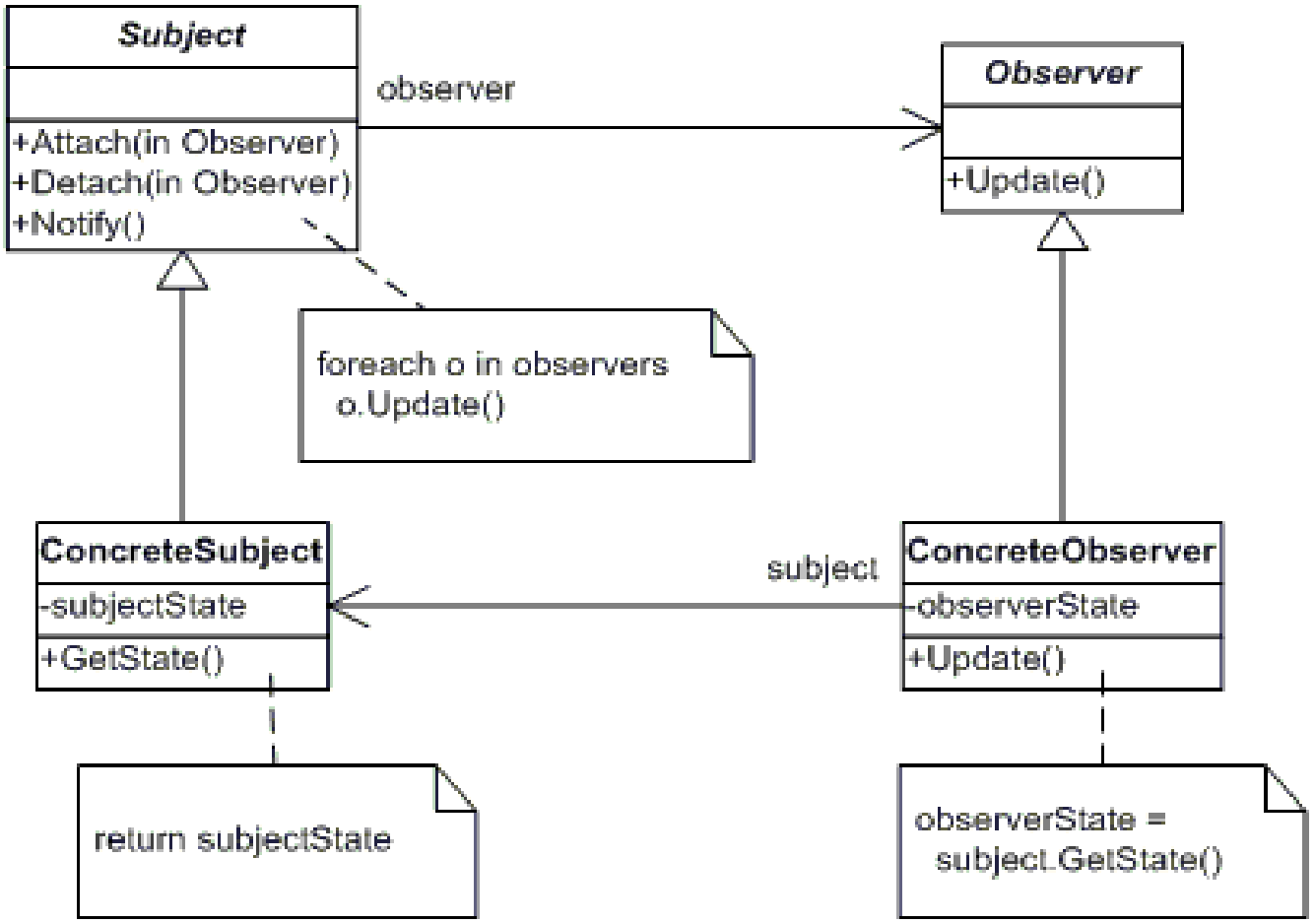
- 出版者+訂閱者=觀察者模式
- Subject + Observer



觀察者樣式 3

- 好的設計
  - 鬆綁兩個物件關係，他們依然可互動，但不清楚彼此細節。(鬆綁，演算法封裝)
  - 隨時都可加入新的Observer(彈性)
  - 有新型態的Observer出現時，並不需改變Subject的程式碼(符合Open-Closed Principle)
  - 可於不同地方單獨運用Subject或Observer，不需將兩者綁一起使用
  - 片面的改變Subject或Observer，並不會影響另一方
- UI 事件通知的方法，例如 Button 的Click()事件
- Model-View-Controller<=最早應用的案例在Smalltalk的GUI Framework

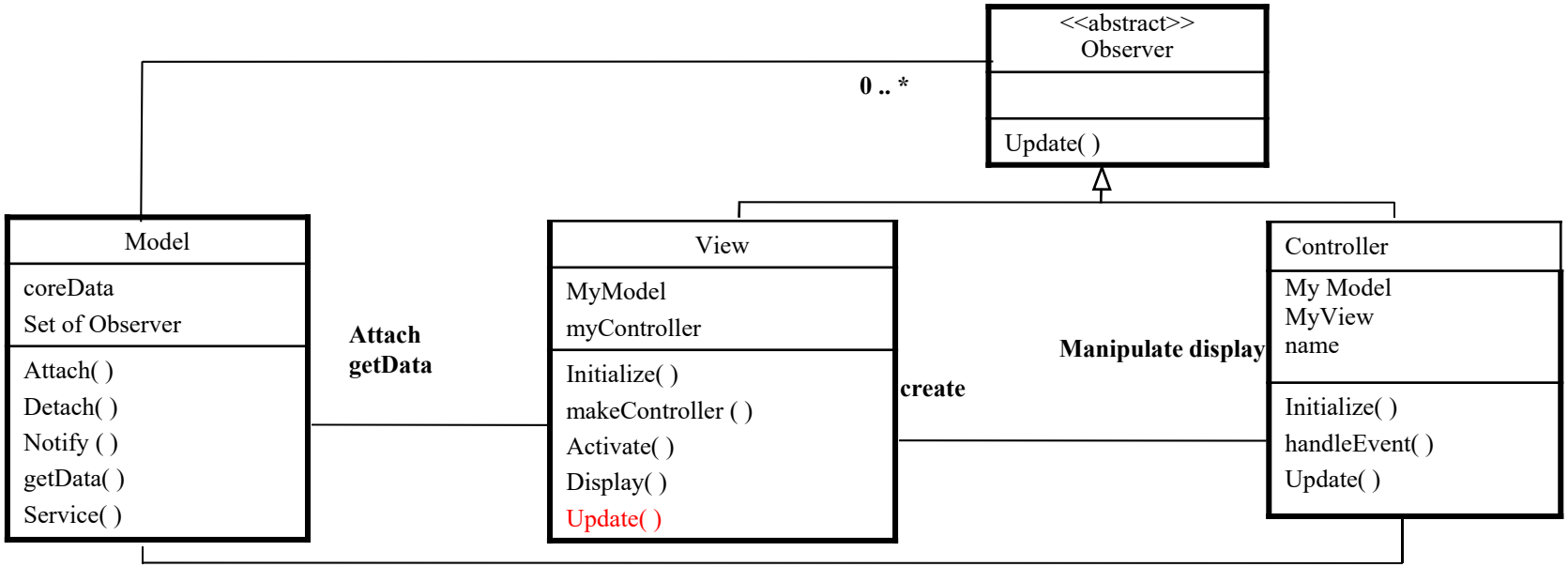
觀察者樣式 4

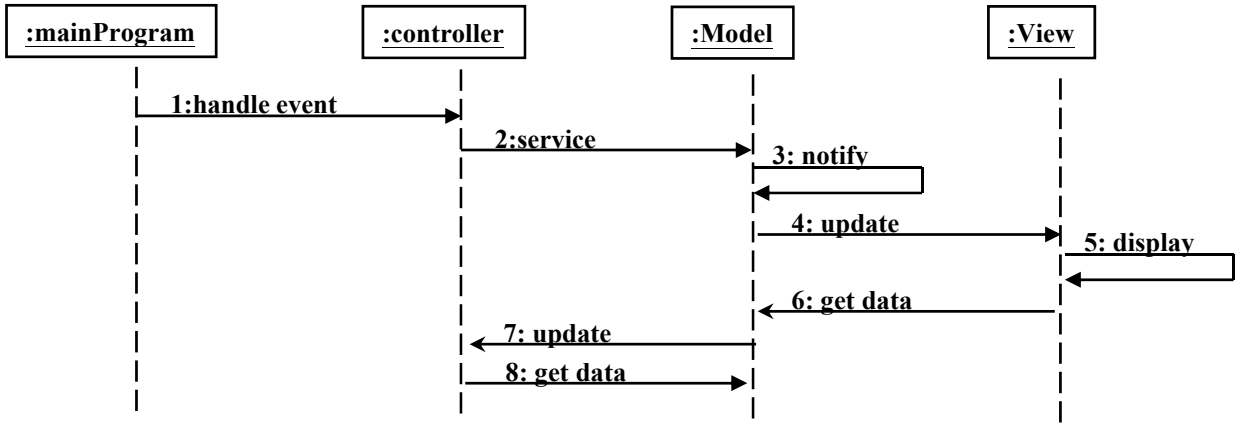

Subject
+Attach(in Observer)
+Detach(in Observer)
+Notify()
observer
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Exercise Observer

- 設計股票通知的觀察者(Observer)樣式
  - 類別圖與循序圖

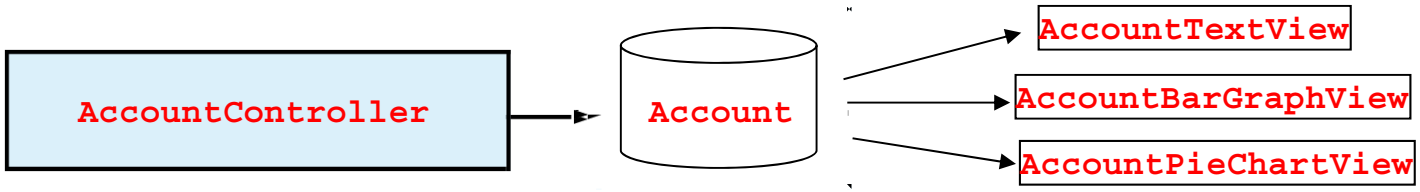
Model View Controller 3

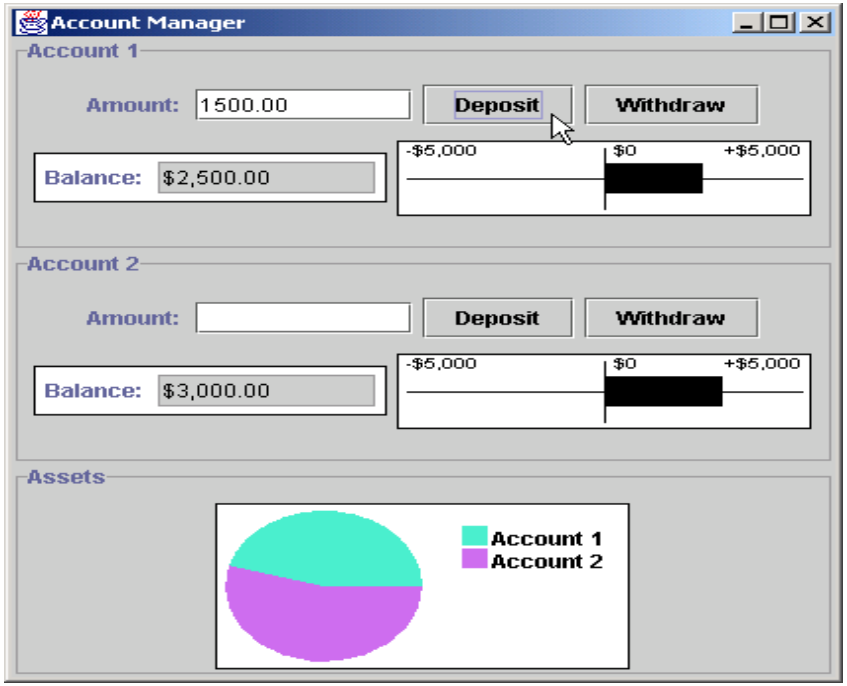




Exercise MVC

畫出類別架構圖





架構分析與設計

- 取得需求目標、規格限制與環境描述
  - 建構需求劇本分析模型
- 選架構風格/樣式，設計初步軟體架構規格，以利表達溝通
  - 動態流程模型
  - 靜態模組模型
- 針對候選軟體架構，評估品質屬性，實施損益權衡分析
  - 架構設計需滿足系統需求目標
  - 了解品質屬性對軟體架構影響，訂定品質屬性優先清單。
  - 良好軟體架構規格文件，由各利害關係人審查。
- 架構設計原則
  - 關注分離、抽象、模組化、高內聚、低耦合
  - 定義良好介面，封裝或隱藏可變更的部分。