

物件導向技能

練習簿

泰迪軟體

泰迪軟體科技 | [HTTP://TEDDYSOFT.TW](http://teddysoft.tw)

目錄

第一部 物件導向基礎觀念	3
1 封裝 (ENCAPSULATION)	4
練習 1-1 : (Java) 以下數字所標示之內容如何稱呼？例如，1 是 <i>data member</i> 或 <i>attribute</i> ，2 是 <i>access modifiers</i> 。	4
練習 1-2 : (Java) 以下數字所標示之內容如何稱呼？	5
練習 1-3 以下數字所標示之內容如何稱呼？	6
練習 1-4 : (Java) 以下數字所標示之內容如何稱呼？	7
練習 1-5 : (Java)	8
練習 1-6 : (Java)	8
2 多型 (POLYMORPHISM)	9
練習 2-1 : (C#)	9
3 繼承 (INHERITANCE 繼承)	10
練習 3-1 : 繼承有哪些好處？	10
重複使用程式碼	10
提高程式碼可讀性	10
實作多型	10
減化測試	10
提高可擴充性	10
將資訊隱藏在父類別中	10
其它_____ (自行填寫)	10
在這些好處中何者是繼承最主要的目的？	10
COLUMN A. 鴨子型別	11
In C#	14
In Java	15

物件導向技能

<i>In JavaScript</i>	16
<i>In Objective-C</i>	17
<i>In PHP</i>	19
第二部 依合約設計 (DESIN BY CONTRACT)	21
4 定義合約	22
<i>練習 4-1：定義 pop method 的 Pre-condition、Post-condition 和 Stack 的 class invariant</i>	22
<i>練習 4-2：合約應用</i>	23
5 子合約	24
<i>練習 5-1：從子合約的角度來解釋為什麼第 9 頁〈2 多型(Polymorphism)〉的例子是不洽當的透過繼承使用多型。</i>	24
<i>練習 5-2：從子合約的角度來判斷以下繼承是否洽當。</i>	25
第三部 物件導向設計原則	27
6 SOLID 原則	28
<i>練習 6-1：套用單一責任原則改寫以下測試案例。</i>	28
<i>練習 6-2：開放封閉原則</i>	29
第四部 物件導向分析與設計	30
7 尋找概念	31

第一部 物件導向基礎觀念

1 封裝 (Encapsulation)

練習 1-1：(Java) 以下數字所標示之內容如何稱呼？例如，1 是 data member 或 attribute，2 是 access modifiers。

```

3 public class Product {
4     public String name;
5     private int quantity;
6     private int itemPrice;
7 }
8
9 public Product(){}
10
11 public int getQuantity() { return quantity;}
12
13 public void setQuantity(int quantity) { this.quantity = quantity; }
14
15 public int getItemPrice(){ return itemPrice; }
16
17 public void setItemPrice(int itemPrice) { this.itemPrice = itemPrice; }
18
19 public double getPrice(){ return basePrice() * discountFactor(); }
20
21 private double basePrice() { return quantity * itemPrice; }
22
23 private double discountFactor() {
24     if (basePrice() > 1000) return 0.88;
25     else return 0.95;
26 }
  
```


Copyright@2012-2016 Teddysoft

```

9 @Test
10 public void testDiscountFactorIs0_88() {
11     Product p = new Product();
12     p.setItemPrice(500);
13     p.setQuantity(10);
14     assertEquals(4400, p.getPrice(), 0.000001);
15 }
16 }
  
```

練習 1-2：(Java) 以下數字所標示之內容如何稱呼？

```
public class FileUtil {  
    public boolean delete(String aFileName) throws IOException {  
        File file = new File(aFileName);  
        return delete(file);  
    }  
    public boolean delete(File aResource) throws IOException {  
        if (aResource.isDirectory()) {  
            File[] childFiles = aResource.listFiles();  
            for (int i = 0; i < childFiles.length; i++) {  
                delete(childFiles[i]);  
            }  
        }  
        return aResource.delete();  
    }  
}
```



練習 1-3 以下數字所標示之內容如何稱呼？

(Java)

```
public class FileSystem {  
    1 private static FileSystem mFileSystem = null;  
    2 private FileSystem(){};  
    3 public static synchronized FileSystem getInstance(){  
        if (null == mFileSystem) {  
            mFileSystem = new FileSystem();  
        }  
        return mFileSystem;  
    }  
}
```

(C#)

```
public class FileSystem  
{  
    1 private static Object lockThis = new Object();  
    private static FileSystem mFileSystem = null;  
    2 private FileSystem() {}  
    3 public static FileSystem GetInstance() {  
        lock(lockThis)  
        {  
            if (null == mFileSystem)  
            {  
                mFileSystem = new FileSystem();  
            }  
        }  
        return mFileSystem;  
    }  
}
```

練習 1-4：(Java) 以下數字所標示之內容如何稱呼？

```

7
8  public abstract class ConfigParser {
9
10  2  protected PersonData mPData = null;
11
12
13  3  public final PersonData doParse(){
14      readData();
15      parseToken();
16      buildModel();
17      validate();
18      return mPData;
19  }
20
21  4  abstract protected void readData();
22      abstract protected void parseToken();
23      abstract protected void buildModel();
24      abstract protected void validate();
25  }
26

```

```

11
12  public class DBConfigParser extends 5 ConfigParser{
13      private String mConnStr = null;
14
15  16  public DBConfigParser(String aConnString){
16      6  super();
17      mConnStr = aConnString;
18  }
19
20  20  @Override
21  21  protected void readData() {
22      System.out.println("Read config data "
23          + "from database: " + mConnStr);
24  }
25
26  26  @Override
27  27  protected void parseToken() {
28      System.out.println("parseToken...");
29  }

```


練習 1-5：(Java)

以下何者符合 command-query separation?

1

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
}
```

2

```
public interface Iterator<T> {  
    boolean hasNext();  
    void next();  
    T getCurrent();  
}
```

練習 1-6：(Java)

Stack 是否符合 command-query separation?

2

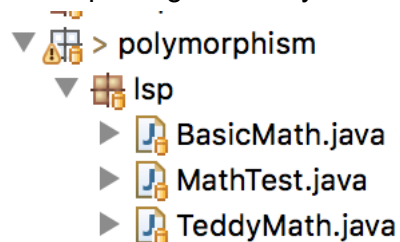
```
class Stack {  
    void push(Object arg);  
    Object pop();  
    int size();  
}
```

2 多型 (Polymorphism)

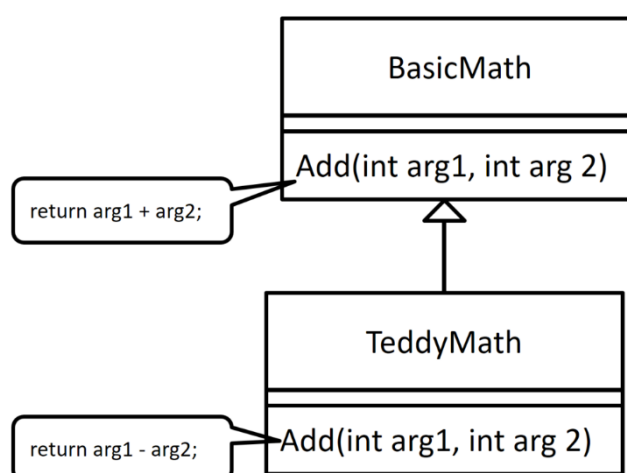
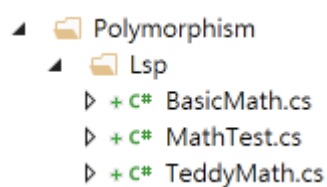
練習 2-1 : (C#)

以下使用透過繼承達到多型的使用方式是否正確？為什麼？

- Java : package tw.teddysoft.oo.polymorphism.lsp



- C# : namespace TW.Teddysoft.OO.Polymorphism.Lsp



3 繼承 (Inheritance 繼承)

練習 3-1：繼承有哪些好處？

- ☐ 重複使用程式碼
- ☐ 提高程式碼可讀性
- ☐ 實作多型
- ☐ 減化測試
- ☐ 提高可擴充性
- ☐ 將資訊隱藏在父類別中
- ☐ 其它_____ (自行填寫)

在這些好處中何者是繼承最主要的目的？

Column A. | 鴨子型別

在[程式設計](#)中，**鴨子型別**（英語：**duck typing**）是[動態型別](#)的一種風格。在這種風格中，一個物件有效的語意，不是由繼承自特定的類或實作特定的介面，而是由"**當前方法和屬性的集合**"決定。這個概念的名字來源於由 [James Whitcomb Riley](#) 提出的[鴨子測試](#)（見下面的「[歷史](#)」章節），「鴨子測試」可以這樣表述：

「當看到一隻鳥走起來像鴨子、游泳起來像鴨子、叫起來也像鴨子，那麼這隻鳥就可以被稱為鴨子。」

在鴨子型別中，關注的不是物件的類型本身，而是它是如何使用的。例如，在不使用鴨子型別的語言中，我們可以編寫一個函式，它接受一個類型為"鴨子"的物件，並呼叫它的"走"和"叫"方法。在使用鴨子型別的語言中，這樣的一個函式可以接受一個任意類型的物件，並呼叫它的"走"和"叫"方法。如果這些需要被呼叫的方法不存在，那麼將引發一個執行時錯誤。任何擁有這樣的正確的"走"和"叫"方法的物件都可被函式接受的這種行為引出了以上表述，這種決定類型的方式因此得名。

鴨子型別通常得益於"不"測試方法和函式中參數的類型，而是依賴文件、清晰的代碼和測試來確保正確使用。從靜態型別語言轉向動態型別語言的用戶通常試圖添加一些靜態的（在執行之前的）型別檢查，從而影響了鴨子型別的益處和可伸縮性，並約束了語言的動態特性。

概念範例

考慮用於一個使用鴨子型別的語言的以下虛擬碼：

```
function calculate(a, b, c) => return (a+b)*c

example1 = calculate (1, 2, 3)
example2 = calculate ([1, 2, 3], [4, 5, 6], 2)
example3 = calculate ('apples ', 'and oranges ', 3)

print to_string example1
print to_string example2
print to_string example3
```

在範例中，每次對 `calculate` 的呼叫都使用的物件（數字、列表和字串）在繼承關係中沒有聯繫。只要物件支援「+」和「*」方法，操作就能成功。例如，翻譯成 [Ruby](#) 或 [Python](#) 語言，執行結果應該是：

```
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
apples and oranges, apples and oranges, apples and oranges,
```

這樣，鴨子型別在不使用[繼承](#)的情況下使用了[多型](#)。唯一的要求是 `calculate` 函式需要作為參數的物件擁有「+」和「*」方法。以下範例（[Python](#) 語言）體現了鴨子測試。就 `in_the_forest` 函式而言，物件是一個鴨子：

```
class Duck:
    def quack(self):
        print "這鴨子在呱呱叫"
    def feathers(self):
        print "這鴨子擁有白色與灰色羽毛"

class Person:
    def quack(self):
        print "這人正在模仿鴨子"
    def feathers(self):
        print "這人在地上拿起 1 根羽毛然後給其他人看"

def in_the_forest(duck):
    duck.quack()
    duck.feathers()

def game():
    donald = Duck()
    john = Person()
    in_the_forest(donald)
    in_the_forest(john)

game()
```

靜態語言中的鴨子型別

一些通常的[靜態](#)語言如 [Boo](#) 和 [C#](#) 第四版，有一些額外的類型註解，它們指示編譯器將類的型別檢查安排在執行時而不是編譯時，並在編譯器的輸出中包含用於執行時型別檢查的代碼[\[3\]\[4\]](#)。這些附加的內容允許這些語言享受鴨子型別的大多數益處，僅有的缺點是需要在編譯時識別和指定這些動態類。

與其他型別系統的比較

[介面](#)可以提供鴨子型別的一些益處，但鴨子型別與之不同的是沒有顯式定義任何介面。例如，如果一個第三方 **Java** 庫實作了一個使用者不允許修改的類，使用者就無法把這個類別的例項用作一個自己定義的介面的實作，而鴨子型別允許這樣做。

批評

關於鴨子型別常常被參照的一個批評是它要求程式設計師在任何時候都必須很好地理解他/她正在編寫的代碼。在一個強靜態型別的、使用了類型繼承樹和參數型別檢查的語言中，給一個類提供未預測的物件類型更為困難。例如，在 **Python** 中，你可以建立一個稱為 **Wine** 的類，並在其中需要實作 **press** 方法。然而，一個稱為 **Trousers**（長褲）的類可能也實作 **press()** 方法。為了避免奇怪的、難以檢測的錯誤，開發者在使用鴨子型別時需要意識到每一個「**press**」方法的可能使用，即使在語意上和他/她所正在編寫工作的代碼沒有任何關係。

本質上，問題是：「如果它走起來像鴨子並且叫起來像鴨子」，它也可以是一隻正在模仿鴨子的龍。儘管它們可以模仿鴨子，但也許你不總是想讓龍進入池塘。

鴨子型別的倡者，如 [吉多·范羅蘇姆](#)，認為這個問題可以通過在測試和維護代碼庫前擁有足夠的瞭解來解決。

對鴨子型別的批評傾向於成為關於 [動態型別和靜態型別](#) 的爭論的更廣闊的觀點的特殊情形。

歷史

[Alex Martelli](#) 很早（2000 年）就在發布到 [comp.lang.python 新聞群組](#) 上的一則 [訊息](#) 中使用了這一術語。他同時對鴨子測試的錯誤的字面理解提出了提醒，以避免人們錯誤認為這個術語已經被使用。

「換言之，不要檢查它是不是一個鴨子：檢查它像不像一個鴨子地叫，等等。取決於你需要哪個像鴨子的行為的子集來使用語言。」

實作

In C#

As of [C# 4.0](#) the compiler and runtime collaborate to implement [dynamic member lookup](#).

In the following C# 6.0 code, the parameter *duck* of the method *Program.InTheForest* is declared as **dynamic**.

```
using static System.Console;

class Duck {
    public void Quack() => WriteLine("Quaaaaaack!");
    public void Feathers() => WriteLine("The duck has white and gray
feathers.");
}

class Person {
    public void Quack() => WriteLine("The person imitates a duck.");
    public void Feathers() => WriteLine("The person takes a feather from
the ground and shows it.");
}

class Program {
    static void InTheForest(dynamic duck) {
        duck.Quack();
        duck.Feathers();
    }

    static void Game() {
        Duck donald = new Duck();
        Person john = new Person();
        InTheForest(donald);
        InTheForest(john);
    }

    static void Main() {
        Game();
    }
}
```

In Java

In [Java](#) duck typing may be achieved with [reflection](#).

```
public class DuckTyping {  
    public static void main(String[] args) {  
        Duck duck = new Duck();  
        Person person = new Person();  
        try {  
            actAsADuck(duck);  
            actAsADuck(person);  
        } catch (NoSuchMethodException e) {  
            System.out.println("Method not found: " +  
e.getMessage());  
        } catch (Exception e) {  
            System.out.println("Execution failed: " + e);  
        }  
    }  
    static void actAsADuck(Object obj) throws Exception {  
        Class<?> objClass = obj.getClass();  
        objClass.getDeclaredMethod("walk").invoke(obj);  
        objClass.getDeclaredMethod("swim").invoke(obj);  
        objClass.getDeclaredMethod("quack").invoke(obj);  
    }  
}  
  
class Duck {  
    public void walk() {System.out.println("I'm a Duck, I can walk...");}  
    public void swim() {System.out.println("I'm a Duck, I can swim...");}  
    public void quack() {System.out.println("I'm a Duck, I can quack...");}  
}  
  
class Person {  
    public void walk() {System.out.println("I'm a Person, I can walk...");}  
    public void swim() {System.out.println("I'm a Person, I can swim...");}  
    public void talk() {System.out.println("I'm a Person, I can talk...");}  
}
```

Running the DuckTyping class will produce the following output:


```
I'm a Duck, I can walk...
I'm a Duck, I can swim...
I'm a Duck, I can quack...
I'm a Person, I can walk...
I'm a Person, I can swim...
Method not found: Person.quack()
```

In JavaScript

```
function Duck() {
  // Dynamically add functions to this object
  this.quack = function() { alert('Quaaaaaack!'); };
  this.feathers = function() { alert('The duck has white and gray feathers.');; };
}

function Person() {
  // Dynamically add functions to this object
  this.quack = function() { alert('The person imitates a duck.');; };
  this.feathers = function() { alert('The person takes a feather from the ground
and shows it.');; };
  this.name = function() { alert('John Smith');; };
}

function inTheForest(object) {
  if (object.quack) // Check that the .quack() function exists
    object.quack();
  if (object.feathers) // Check that the .feathers() function exists
    object.feathers();
}

function game() {
  var donald = new Duck();
  var john = new Person();
  inTheForest(donald);
}
```

```

    inTheForest(john);
}

// Execute upon page load
game();

```

The function `inTheForest()` above checks that the object passed to it has a member `quack` and a member `feathers` before attempting to invoke them. More explicit tests could also be made, wherein the runtime types of the members are also checked:

```

function inTheForest(object) {
    if (object.quack && typeof(object.quack) == 'function') // Check that
the .quack() function exists
        object.quack();
    if (object.feathers && typeof(object.feathers) == 'function') // Check
that the .feathers() function exists
        object.feathers();
}

```

In Objective-C

[Objective-C](#), a cross between [C](#) and [Smalltalk](#), allows one to declare objects of type 'id' and send any message to them (provided the method is declared somewhere), like in Smalltalk. The sender can test an object to see if it responds to a message, the object can decide at the time of the message whether it will respond to it or not, and, if the sender sends a message a recipient cannot respond to, an exception is raised. Thus, duck typing is fully supported by Objective-C.

```

#import Foundation;

@interface Duck : NSObject
@end

@implementation Duck
- (void)quack { NSLog(@"Quaaaack!"); }

```

```
@end

@interface Person : NSObject
@end

@implementation Person
- (void)quack { NSLog(@"The person imitates a duck."); }
@end

@interface Dog : NSObject
@end

@implementation Dog
- (void)bark { NSLog(@"Baaaaark!"); }
@end

void inTheForest(id duck) {
    if ([duck respondsToSelector:@selector(quack)]) {
        [duck quack];
    }
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        inTheForest([[Duck alloc] init]);
        inTheForest([[Person alloc] init]);
        inTheForest([[Dog alloc] init]);
    }
    return 0;
}
```

Output:

```
Quaaaack!
The person imitates a duck.
```

In PHP

[PHP](#) leans towards the Java convention of using inheritance and the user land type system (type hinting method arguments or using instanceof class or interface) in favour of duck typing. Below is an example of duck typing:

```
<?php
class Duck {
    function quack() { echo "Quack", PHP_EOL; }
    function fly() { echo "Flap, Flap", PHP_EOL; }
}
class Person {
    function quack() { echo "I try to imitate a duck quack", PHP_EOL; }
    function fly() { echo "I take an airplane", PHP_EOL; }
}
function in_the_forest($object) {
    $object->quack();
    $object->fly();
}
in_the_forest(new Duck);
in_the_forest(new Person);
```

Output:

```
Quack
Flap, Flap
I try to imitate a duck quack
I take an airplane
```

以上資料整理自維基百科：

- <https://zh.wikipedia.org/wiki/%E9%B8%AD%E5%AD%90%E7%B1%BB%E5%9E%8B>
- https://en.wikipedia.org/wiki/Duck_typing

第二部 依合約設計（**Desin By Contract**）

4 定義合約

練習 4-1: 定義 pop method 的 Pre-condition、Post-condition 和 Stack 的 class invariant

```
public interface Stack {  
  
    void push(Object arg);  
  
    Object pop();  
  
    Object top();  
  
    int size();  
}
```

練習 4-2：合約應用

你要設計一個讓使用者透過網頁輸入個人資料，然後將資料儲存到資料庫的功能。其中有一個欄位是輸入「年齡」，因為正常人類不可能超過 200 歲，所以你在資料庫中將年齡這個欄位的長度設為 `unsigned short (0-255)`。除非科學家發明了長生不老藥，否則 255 應該很夠用了。

你為這個功能設計了四個元件：

- **UI**：顯示網頁的程式碼。
- **UserBean**：用來將使用者輸入的資料由 Web UI 傳送到資料庫。
- **Servlet**：呼叫 **UserDAO** 將由 UI 端收到的 **UserBean** 存到資料庫中。
- **UserDAO (Data Access Object)**：負責透過 JDBC 或是 OR-Mapping 工具將 **UserBean** 存到資料庫中。如果儲存資料發生錯誤，將丟出 `exception`。

程式開發完畢，很幸運的你找來「彭祖」幫你作測試，「彭祖」在年齡欄位輸入 888，按下確定送出之後，畫面上看到由資料庫所發出的「`Exception: integer out of range`」。每個人都說不是它的問題：

- **UI**：這不是我的問題。誰叫你資料庫欄位設計的太小，改成 `unsigned int (0-65535)` 不就好了。
- **UserBean**：很明顯的這不是我的問題，因為我的 `setAge()` 和 `getAge()` 接受和傳回的都是 `Int`，範圍大於 888。
- **Servlet**：這絕對不是我的問題，我只是負責把收到的 **UserBean** 傳給 **UserDAO**。
- **UserDAO**：這肯定不是我的問題。依照規格，年齡欄位最大值就是 255。有人要輸入超過這個數值的資料，我的 `saveUser (UserBean aBean)` 函數當然會丟出例外，如果不丟出例外才是我的問題。

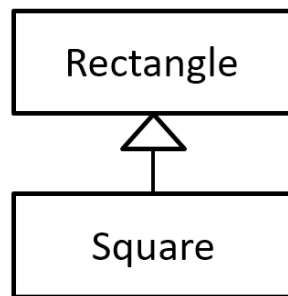
請套用 **Design By Contract** 來分析誰要為此例外負責，進一步探討系統要如何修改以避免該例外產生。

5 子合約

練習 5-1：從子合約的角度來解釋為什麼第 9 頁〈2 多型（Polymorphism）〉的例子是不恰當的透過繼承使用多型。

提示：寫出 BasicMath 類別的 add method 的 post-condition 與 MyMath 類別的 add method 的 post-condition。

練習 5-2：從子合約的角度來判斷以下繼承是否洽當。



```
public class Rectangle {  
  
    private Point topLeft;  
    private double width;  
    private double height;  
  
    public double getWidth() {  
        return width;  
    }  
    public void setWidth(double width) {  
        this.width = width;  
    }  
  
    public double getHeight() {  
        return height;  
    }  
    public void setHeight(double height) {  
        this.height = height;  
    }  
  
    public double area(){  
        return width * height;  
    }  
}
```

```
public class Square extends Rectangle {
```

- ⊖ **@Override**
public void setWidth(**double** width){
 super.setWidth(width);
 super.setHeight(width);
}
- ⊖ **@Override**
public void setHeight(**double** height){
 super.setHeight(height);
 super.setWidth(height);
}
}

```
@Test  
public void testRectangleArea() {  
  
    Rectangle r = new Rectangle();  
    r.setHeight(5);  
    r.setWidth(4);  
  
    assertEquals(20.0, r.area(), 0.000001);  
  
}
```

```
@Test  
public void testSquareArea() {  
  
    Rectangle r = new Square();  
    r.setHeight(5);  
    r.setWidth(4);  
  
    assertEquals(20.0, r.area(), 0.000001);  
  
}
```

第三部 物件導向設計原則

6 SOLID 原則

練習 6-1：套用單一責任原則改寫以下測試案例。

```
public void testWithdraw() {  
    // A normal withdraw amount  
    long timestamp = getTimeStamp(2012, 1, 11, 10, 0);  
    Assert.assertTrue(_account.withdraw(200.0, timestamp));  
    Assert.assertEquals(2800.0, _account.getBalance(), ACCEPTABLE_DELTA);  
  
    // A negative (unacceptable) withdraw amount  
    Assert.assertFalse(_account.withdraw(-200.0, timestamp));  
    Assert.assertEquals(2800.0, _account.getBalance(), ACCEPTABLE_DELTA);  
  
    // A withdraw amount that is larger than the balance  
    Assert.assertFalse(_account.withdraw(4000.0, timestamp));  
    Assert.assertEquals(2800.0, _account.getBalance(), ACCEPTABLE_DELTA);  
  
    // A withdraw amount that is the same as the balance (boundary test)  
    Assert.assertTrue(_account.withdraw(2800.0, timestamp));  
    Assert.assertEquals(0.0, _account.getBalance(), ACCEPTABLE_DELTA);  
  
    // Another boundary test  
    Assert.assertFalse(_account.withdraw(1, timestamp));  
    Assert.assertEquals(0.0, _account.getBalance(), ACCEPTABLE_DELTA);  
}
```

練習 6-2：開放封閉原則

1. 是不是系統支援多型就算滿足開放封閉原則?為什麼?
2. 你知道哪些設計模式或軟體架構符合開放封閉原則?

第四部 物件導向分析與設計

7 尋找概念

練習 7-1：請從以下使用案例（Use Case）找出概念

Use Case：執行同步設定

Primary Actor：*使用者*

Precondition：使用者已經新增一個以上的*同步設定*。

1. 這個 use case 開始於使用者手動執行同步設定。
2. 系統紀錄同步開始時間，並寫入同步紀錄檔中。
3. 系統列出來源目錄中，符合同步設定的檔案與目錄結構。
4. 系統列出遠端目錄中，符合同步設定的檔案與目錄結構。
5. 系統讀出「*同步參照表*」。
6. 系統比對來源目錄、遠端目錄並參考「*同步參照表*」中之資料，依此資料決定檔案之版本新舊、檔案新增或檔案刪除。
7. 依照步驟 6 比對之結果，開始同步檔案，一直到來源目錄與遠端目錄資料同步為止。每次完成同步一個檔案後即將結果寫入*同步紀錄檔*中。
8. 產生新的「*同步參照表*」，並覆蓋原有之資料。
9. 完成同步動作，紀錄完成時間，並寫入同步紀錄檔中。

Extensions：

4a 列出遠端目錄與檔案結構失敗：

4a1 顯示錯誤訊息，並停止同步動作。

7a 同步某檔案失敗：

4a1 紀錄錯誤訊息，並寫入同步紀錄檔中。

4a2 繼續同步下一個檔案。

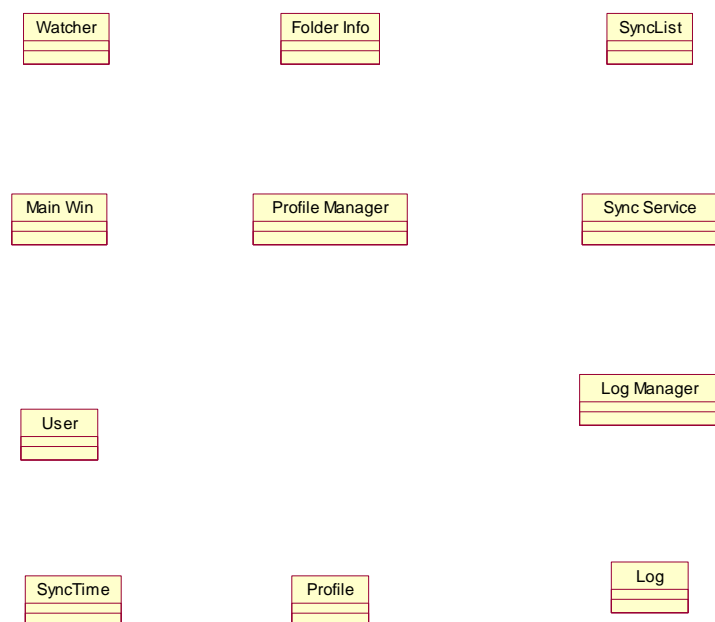
7b 網路斷線：

7b1 紀錄錯誤訊息，並寫入同步紀錄檔中。

7a2 停止同步動作，紀錄停止時間，並寫入同步紀錄檔中。

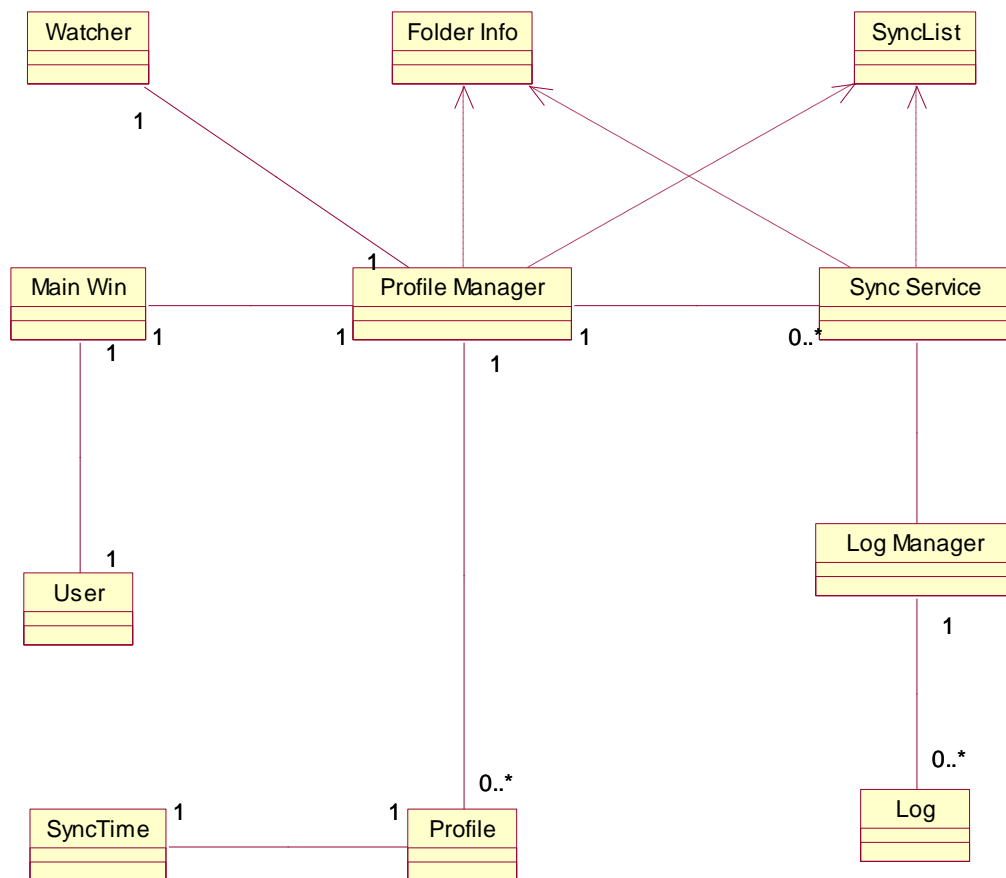
參考範例

Conceptual diagram showing only concepts

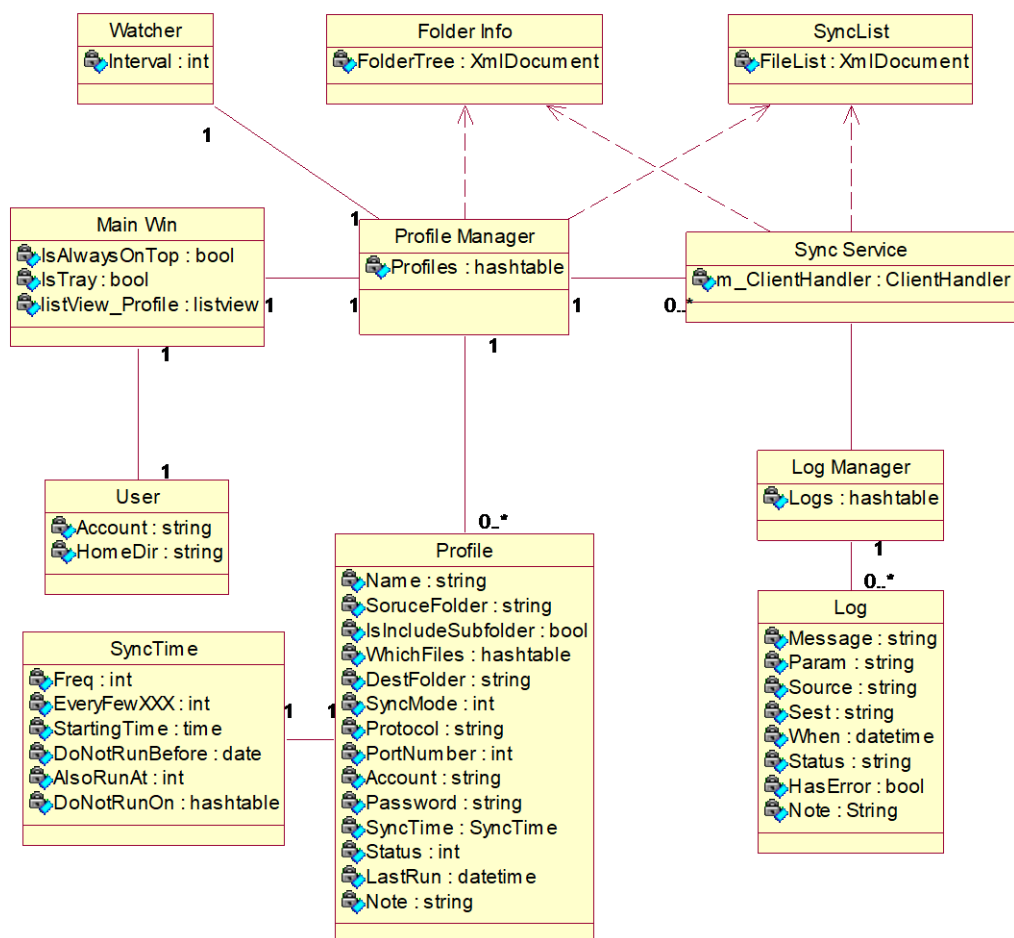


Classes	How it is extracted
Watcher	Noun phrase
MainWin	User interface
User	Noun phrase
SyncTime	Data object
FolderInfo	Noun phrase
ProfileManager	Containers of other things
Profile	Noun phrase
SyncList	Containers of other things
SyncService	System service
LogManager	Containers of other things
Log	Noun Phrase

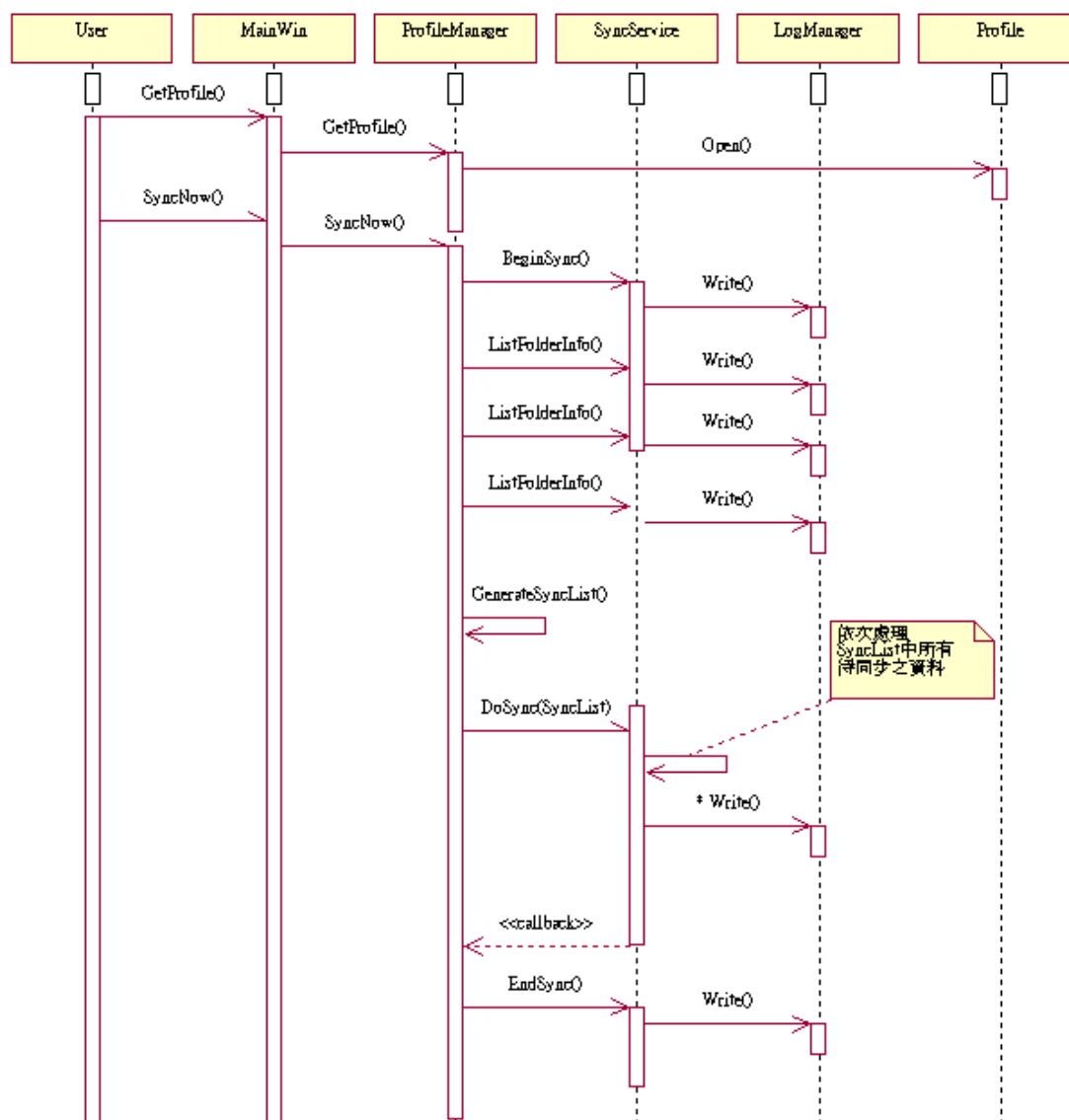
Conceptual diagram showing concepts with associations



Conceptual diagram showing concepts with association and attributes



System sequence diagram



除 User 外其餘皆為系統部分，真正的 system operators 只有 User→MainWin 之間的 GetProfile 和 SyncNow。

Contract

Contract : GetProfile

Operation : GetProfile (profileName : string)

Preconditions :

- 系統已成功啟動，MainWin 與 ProfileManager 都已存在。
- 系統中至少存在一個以上的 Profile。

Postconditions :

- A Profile instance prof was created.
- prof was associated with the ProfileManager.
- prof 的狀態將被設定（依據 profileName 找出狀態資料）。
- prof 於 MainWin 中將被以反白顯示。

Contract : SyncNow

Operation : SyncNow (profileName : string)

Preconditions :

- 已經選擇了某個 Profile。

Postconditions :

- prof 所紀錄之 SourceFolder 與 DestFolder，將依據 SyncMode 之設定方式，被同步成相同狀態。
- Log 被用來紀錄同步之過程。
- prof 的 LastRun 屬性將被設定為本次執行時間。
- prof 新的 LastRun 屬性將於 MainWin 中被重新顯示。
- FolderInfo（同步參照表）將被重新產生，紀錄最新之 SourceFolder 狀態。