

# 軟體重構

軟體架構的基石

BEGINNER

SKILLED

ADVANCED

SENIOR

EXPERT

# 毛豆

## ► 架構師

- 設計底層
- 減少開發複雜度
- 降低開發門檻

## ► 技能

- ASP.NET
- jQuery
- IoT



## ► K.NET 成員

## ► 創客閣樓成員

## ► Webduino 志工

## ► 中台灣碼農大食團 團長

## ► 常跑 MonoSpace

為何講這個主題

# 寫軟體

- ▶ 很多人都會
- ▶ 一開始應該都很快

# 維護軟體

- ▶ 就不那麼容易了
- ▶ 在維護階段，開發進度慢下來了？
- ▶ 維護？

# 隨著軟體的開發


- ▶ 錯誤、臭蟲的摘除
- ▶ 需求變更
- ▶ 客製化的要求
- ▶ 畫面的美觀

# 隨著軟體的開發

- ▶ 錯誤、臭蟲的摘除
- ▶ 需求變更
- ▶ 客製化的要求
- ▶ 畫面的美觀
- ▶ 一堆 `if / else`
- ▶ 垃圾 Method 變多
- ▶ 寫了客戶編號在程式裡
- ▶ 相同用途卻被拷貝多次
- ▶ 目的相似的東西越來越多

記得有個金句如是說：





“你的同事有嚴重暴力傾向、  
他對維護他人的爛程式沒耐心、  
而且他知道你的住址...”

我也忘了從哪邊看到的金句，我猜是約耳

寫程式時多點技巧，可以保命 XD

# 軟體架構 – 定義

- ▶ 是一個系統的草圖。
- ▶ 軟體架構描述的對象是直接構成系統的抽象組件。
- ▶ 各個組件之間的連接則明確和相對細緻地描述組件之間的通訊。
- ▶ 在實現階段，這些抽象組件被細化為實際的組件，比如具體某個類或者對象。
- ▶ 在面向對象領域中，組件之間的連接通常用介面來實現。

# 簡言之

- ▶ 軟體由許多元件構成。
- ▶ 怎麼讓元件適當地發揮，就是軟體架構。
  - ▶ 正確的角色
  - ▶ 適當的位置
  - ▶ 稜織合度的功能

# 架構的目的是什麼

- ▶ 可靠性

- ▶ Reliable

- ▶ 安全性

- ▶ Secure

- ▶ 可伸縮性

- ▶ Scalable

- ▶ 可定製化

- ▶ Customizable

- ▶ 可擴展性

- ▶ Extensible

- ▶ 可維護性

- ▶ Maintainable

- ▶ 客戶體驗

- ▶ Customer Experience

- ▶ 市場時機

- ▶ Time to Market

# 什麼是重構

讓我們開始吧

# 傳統寫程式

```
var loto = new Array();
var sel = new Array();
var s_msg = "";
var m_msg = "";

function chk_select() {
    chk = 0;
    s_msg = "自選：";
    for (i = 0; i <= 41; i++) {
        if (document.LOTO.SN[i].checked == true) {
            sel[chk] = i + 1;
            s_msg += sel[chk] + " ";
            chk++;
        }
    }
    if (chk != 6) return false;
    return true;
}
```

```
function chk_match() {
    chk = 0;
    m_msg = "對中：";
    for (i = 0; i <= 5; i++) {
        for (j = 0; j <= 5; j++) {
            if (loto[i] == sel[j]) {
                m_msg += loto[i] + " ";
                chk++;
            }
        }
    }

    msg = s_msg + "\n\n" + m_msg;
    if (chk == 6) { msg += "\n\n恭喜你中了頭獎"; }
    else if (chk == 5) {
        ok = 0;
        for (i = 0; i <= 5; i++) {
            if (loto[6] == sel[i]) ok = 1;
        }
        if (ok == 1) { msg += loto[6] + "\n\n恭喜你中了二獎"; }
        else { msg += "\n恭喜你中了三獎"; }
    }
    else if (chk == 4) { msg += "\n\n恭喜你中了四獎"; }
}
```





# 重構定義

- ▶ 對軟體程式碼做任何更動，以增加可讀性或者簡化結構而不影響輸出結果。



# 說白話

- ▶ 找出程式碼的壞味道
- ▶ 使用一系列的技巧，整理你的程式碼。使之井然有序，或增加彈性。
- ▶ 結構（外部行為）不允許改變：
  - ▶ 回傳值、參數數量、參數型別。



# SOLID 原則

- ▶ S：單一功能原則
- ▶ O：開閉原則
- ▶ L：里氏替換原則
- ▶ I：接口隔離原則
- ▶ D：依賴反轉原則

# SOLID 原則在說什麼

- ▶ 最小知識，不過度曝露細節
- ▶ 專注於自己的程式
- ▶ 要改程式前，想想能不能擴充
- ▶ 安心使用自己專用的方法
- ▶ 契約為重



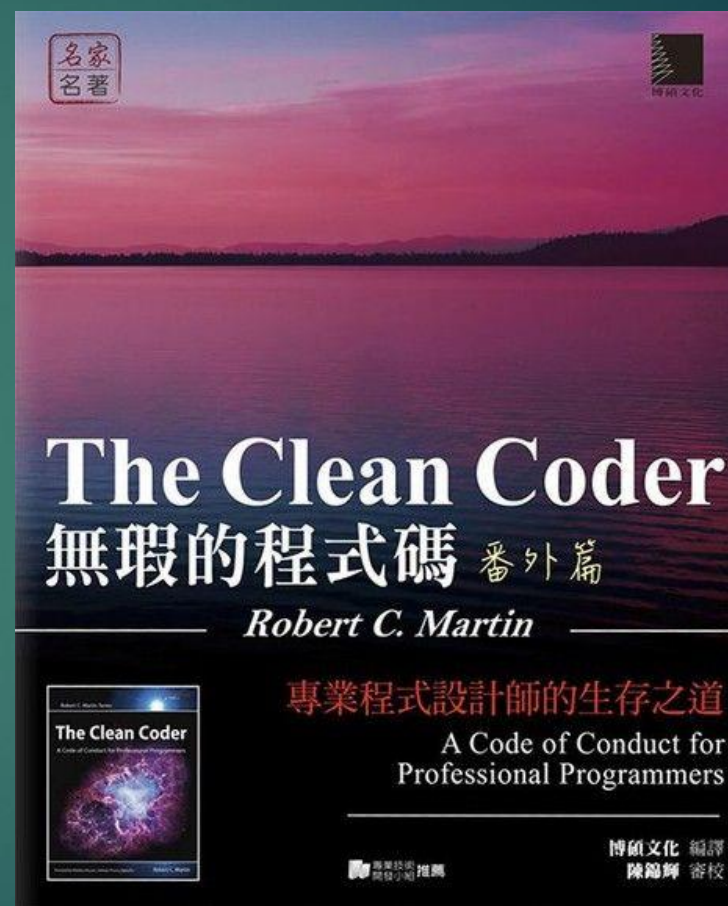
# 重構 – 常見的壞味道

1. Duplicated code (重複的程式碼)
2. Long method (過長函式)
3. Large class (過大類別)
4. Long parameter list (過長參數列)
5. Feature envy (依戀情節)
6. Data clumps (資料泥團)
7. Unsuitable naming (名不符實) ☹
8. Lack of comments (缺乏註解)
9. Inconsistent coding standard (不一致的程式碼標準) ☹
10. Fat view (臃腫的外表)
11. Unresolved warnings (忽視警告) ☹
12. Literal constants (字面常數)
13. Message chains(小心陌生人) ☹

# 這邊推薦一下書籍



# 這邊推薦一下書籍



# 可以怎麼進行重構？

## 基本原則

- ▶ 依需求切割
- ▶ 依用途切割
- ▶ 依流程切割
- ▶ 依變動頻率切割
- ▶ 依複用性切割

## 技巧

- ▶ 抽取或減少 Method
- ▶ 搬移程式
- ▶ 委派
- ▶ 打包物件



依需求切割

# 需求

- ▶ 專案中得達成的目標
  - ▶ 專用的、特定的、沒有它就沒價值的
  - ▶ 可以從中分析出商業邏輯
- ▶ 如果在做會計系統
  - ▶ 輸入會計科目、金額試算等就是需求
  - ▶ 會員管理、檔案匯出、自毀功能則不是
- ▶ 某個角度，需求也代表相關性

# 一個小小的範例

## ▶ Page 1

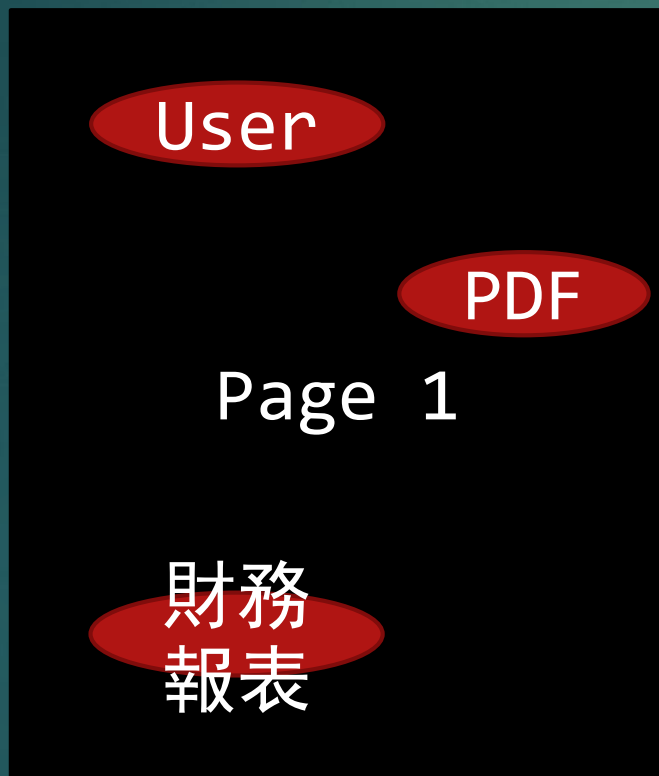
- ▶ 驗證管理者登入
- ▶ 匯出財務報表成 PDF
- ▶ 有問題就存 LOG

## ▶ Page 2

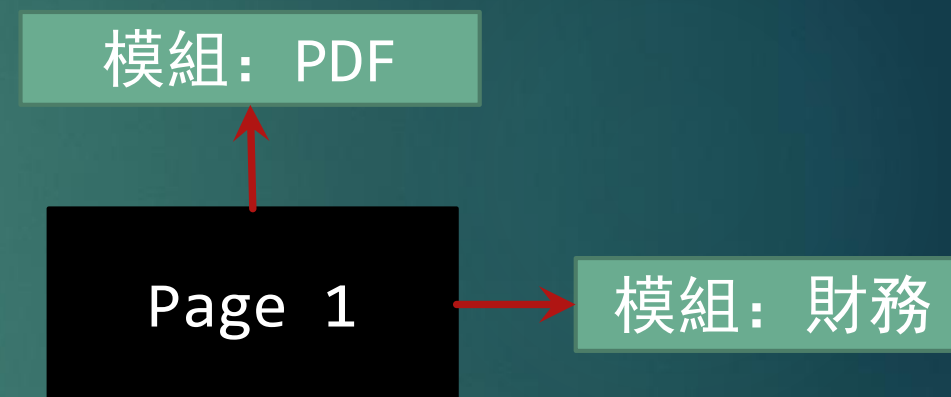
- ▶ 驗證管理者登入
- ▶ 查出產品清單
- ▶ 發送廣告電子報
- ▶ 有問題就存 LOG

# 依需求切割模組 – 結果

Old Style



New Style



依用途切割

# 用途

- ▶ 非專案目標，但具「必要」或是「輔助性質」
  - ▶ 會員、資料庫、 LOG
  - ▶ 網址處理、取日期

- 為什麼資料庫存取程式到處都是
- 為什麼計算利息和檔案存取在一起

# 和剛才範例一樣

## ▶ Page 1

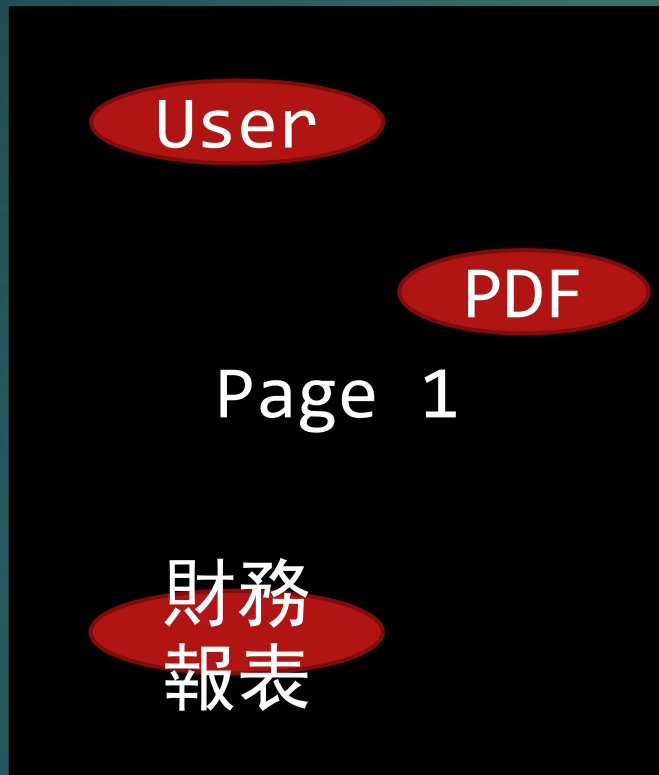
- ▶ 驗證管理者登入
- ▶ 匯出財務報表成 PDF
- ▶ 有問題就存 LOG

## ▶ Page 2

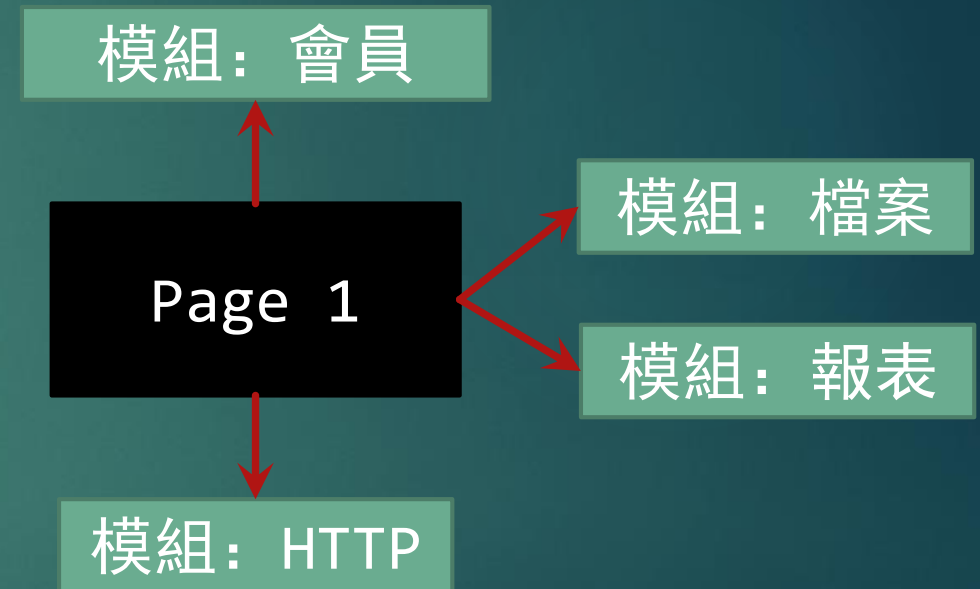
- ▶ 驗證管理者登入
- ▶ 查出產品清單
- ▶ 發送廣告電子報
- ▶ 有問題就存 LOG

# 依用途切割模組 – 結果

Old Style



New Style





# 依流程切割

# 流程

- ▶ 完成一件事的所有步驟及判斷
- ▶ 和商業邏輯有關係
- ▶ 像是
  - ▶ 把貨物放到地上 => 清點數量 => 回報上級

# 為什麼要依流程切割

- ▶ 重覆使用
- ▶ 減少相關程式
- ▶ 增加彈性

# 程式 - 常常變成這樣

- ▶ 收貨() {
  - ▶ 查詢資料庫;
  - ▶ 取得收貨單;
  - ▶ 清點數量;
  - ▶ 回寫資料庫;
  - ▶ 回報上級;
  - ▶ 發送 MAIL;
- ▶ }

# 我們可以怎麼做

- ▶ 收貨() {
  - ▶ 取得收貨單();
  - ▶ 清點數量(產品數量);
  - ▶ 回報(上級);
  - ▶ 回報(相關人);
- ▶ }

- ▶ 取得收貨單() {
  - ▶ 查詢資料庫;
  - ▶ 取得收貨單;
- ▶ }

- ▶ 清點數量(產品數量) {
  - ▶ 輸入數量;
  - ▶ 回寫資料庫;
- ▶ }

- ▶ 回報(接收人) {
  - ▶ 發送 MAIL 給接收人;
- ▶ }

# 為什麼要依流程切割

- ▶ 今天需求長這樣
- ▶ 把貨物放到地上 => 清點數量 => 回報上級

# 如果改成這樣呢？

## ▶ 原本：

▶ 把貨物放到地上 => 清點數量 => 回報上級

## ▶ 改為：

▶ 把貨物放到地上 => 清點數量 => 收入暫存貨架  
=> 品質檢驗 => 退貨 / 驗收完成 => 回報上級

# 嗯？這下有趣了...

```
▶ 收貨() {  
    ▶ 查詢資料庫;  
    ▶ 取得收貨單;  
    ▶ 清點數量;  
    ▶ 回寫資料庫;  
    ▶ 回報上級;  
    ▶ 發送 MAIL;  
▶ }
```



新程式要放哪裡呀？



# 可以再這樣，不過最後再說

```
▶ 收貨() {  
  ▶ 流程一();  
  ▶ 流程二();  
  ▶ 流程三();  
▶ }
```

```
▶ 取得收貨單() {  
  ▶ 查詢資料庫;  
  ▶ 取得收貨單;  
▶ }
```

```
▶ 清點數量(產品數量) {  
  ▶ 輸入數量;  
  ▶ 回寫資料庫;  
▶ }
```

```
▶ 回報(接收人) {  
  ▶ 發送 MAIL 給接收人;  
▶ }
```

依變動頻率切割

# 程式變動頻率

- ▶ 一段程式是否經常變動
- ▶ 原因有
  - ▶ 速度慢
  - ▶ 很難閱讀
  - ▶ SD 說要改（就像以前的我）

# 高頻率和低頻率

## 低頻率

- ▶ 資料庫連線底層
- ▶ LOG Method
- ▶ 查詢產品清單

## 高頻率

- ▶ MAIL TO 誰
- ▶ 文件電子簽核者
- ▶ 頁面名稱
- ▶ 帳號密碼
- ▶ 等等

# 會怎麼樣

- ▶ 相同的東西變動後一直測
  - ▶ LOG 等
- ▶ 可能會把對的東西改壞
- ▶ 搞不好順手加入了...
  - ▶ 可以運作，但你不該在這裡
  - ▶ 為了方便測試，把參數寫死後忘記了



# 可用技巧

- ▶ 將值抽到參數
  - ▶ 減少魔術數字
  - ▶ 讀取 Config 後再放入程式參數
- ▶ 建立數套程式，用參數決定
  - ▶ 不折扣 / 過季六折 / 三件八折

這篇不帶範例





依複用性切割

# 複用性

- ▶ 和依流程切割有點像
- ▶ 這段商業邏輯可能會在它處被使用
  - ▶ 查詢指定的數個產品
  - ▶ 加入時間戳及登入者
  - ▶ 假如 XXXX 節到了，或快到了

# 依複用性切割

- ▶ 存取權限不一樣
- ▶ 將可能重用的商業邏輯抽出
- ▶ 談複用性前，得瞭解一點委派 (Delegate)

# 委派 Delegate

- ▶ 代表自己只能做一半的事情
- ▶ 執行到一半時，呼叫外面的程式，再回歸自己
- ▶ 委派即事件
  - ▶ 但不只是 event

# Sample Code of 無委派版本

```
(function() {  
    Button1_Click_Event();  
})();  
  
function Button1_Click_Event() {  
    if(!window.confirm("確定要這麼做嗎?"))  
        return;  
  
    var cnt = 0;  
    for(var i = 1 ; i <= 100; i++) {  
        cnt += i;  
    }  
  
    console.log("Button1 Click Done;");  
}
```

# Sample Code of 有委派版本

```
(function () {  
    var event1 = Button1_Click_Event;  
  
    SimulateButtonClick(event1, "Button1 Event");  
})();
```

```
function Button1_Click_Event() {  
    var cnt = 0;  
    for (var i = 1; i <= 100; i++) {  
        cnt += i;  
    }  
}
```

```
function SimulateButtonClick(funcExecute, MethodName) {  
    if (!window.confirm("確定要這麼做嗎?"))  
        return;  
  
    funcExecute();  
  
    console.log(MethodName + " Done;");  
}
```

一般課本教的事件  
就是這裡

# 委派和複用性的關係

- ▶ 有些程式無法單獨切割成獨立方法
- ▶ 一定會遇到「做到一半時，需要告訴我怎麼做，最後我又會知道怎麼做」的程式
- ▶ 像 UI 就常常會遇到



# 商業邏輯複用性 – 案例

- ▶ 檢查是否有權做 XXX ，然後就要 000
- ▶ 如果 XXX 情況發生，得呼叫數量不定的方法
- ▶ 確定做 XXX ，但希望檢查可以動態一點
  - ▶ 今天檢查天氣
  - ▶ 明天檢查車況
    - ▶ 就是不想上班...

# 委派的優勢在哪？

- ▶ 委派 = 事件
- ▶ 處理非同步的程式特別好用
- ▶ 老做法：
  - ▶ 氣象局抓資料要 1 hr 或 3 sec
  - ▶ `Sleep(1 sec);`

END