# 程式碼品質批改原則

寫程式實作一個功能不難，但是要實作出**易懂**、**易改**程式碼則很難，這需要良好的程式寫作習慣，以提升程式碼的可讀性(Readability)。一個好的程式碼應該沒有**壞味道**(Bad smells [1])，因此，在程式作業中，以下壞味道將列入評分，以扣分方式執行(有☹標示加重扣分)。

## 常見的壞味道

### 1. Duplicated code (重複的程式碼) ☹

壞味道中最明顯的就是**重複的程式碼**，重複的程式碼會讓修改程式(包含除錯)變成一件困難的事情，因為當面臨變更時，必須在一個檔案的多個地方或是多個檔案中，找出重複的程式碼進行修改。為了避免重複的程式碼，如果你需要複製貼上一段程式碼時，請用 *Extract Method* 重構方法，將那段程式碼改寫成一個函式(method)，然後在要貼上的地方，直接呼叫該函式。如果程式碼之間相似但不完全相同，即複製貼上後要稍微修改，請利用函式的參數達成變動能力。

### 2. Long method (過長函式)

程式碼越長越難理解，也難以重複利用，因此常常需要寫大量的註解，且過長的函式需要經常捲動畫面來了解上下文，更重要的是長函式很難有合適的名稱。相對地，短函式(short methods)容易取名字，看到函式名稱就可以理解函式的功能，短函式往往各自獨立，因此更容易重複使用，因此建議用 *Extract Method* 重構手法，將過長函式拆開成為短函式。函式多長才算是過長呢？一般而言，一個畫面放不進去時，或大約 20 行以上時就算過長。另外，當需要為一段程式碼寫很多註解時，也許就應該將該段程式碼改寫成函式；或者，當需要寫迴圈處理資料時，也應該考慮將迴圈改寫成函式。

### 3. Large class (過大類別)

一個好的設計必須滿足 High cohesion 及 Low coupling 的設計準則，當有一個 Class 做太多事情時，不但 Class 會很大，而且違反 High cohesion 準則，甚至可能連帶使其他 Class 依賴這個 Class 產生不必要的 Coupling。通常一個大 class 都會有 Duplicated code 及 Long method 的壞味道，當發現一個 Class 有太多成員變數，或是有太多跟 Class 無關的函式時，就是一個過大類別的壞味道，可以使用 *Extract Class* 將一個 Class 拆成多個 Classes。

### 4. Long parameter list (過長參數列)

為了避免使用全域變數(Global variables)，一個好的函式，所需要的資料除了成員變數(Member data)外，全都以參數傳遞，但過長的參數列難以理解，呼叫函式時常常不知道要準備那些參數，而且一但需要更多參數時，修改函式的屬名(Signature)須連帶修改所有呼叫該函式的舊程式。此時可以使用 *Introduce Parameter Object* 重構方法將參數包裝成一個「參數物件」。

### 5. Feature envy (依戀情節)

物件設計的重點在於將資料和加諸其上的操作行為包裝在一起，一個典型的味道是某函式對於另一個 class 的興趣高於自己本身的 class，例如函式常常需要取得另一個 class 的多個成員變數，或是呼叫另一個 class 的多個成員函式。這個函式也許就不該屬於這個 class，此時，可以用 *Move Method* 將該函式移到另一個 class。

### 6. Data clumps (資料泥團)

資料項(Data items)就像小孩子：喜歡成群結隊地待在一塊兒。常常可以在許多地方看到相同的三或四筆資料項，而且為了完成該工作，這些資料項缺一不可，這些總是綁在一起出現的資料，應該放進屬於他們自己的物件中，此時可以使用 *Extract Class* 或 *Introduce Parameter Object* 將這些資料項包裝起來。

### 7. Unsuitable naming (名不符實) ☹

好的命名讓人易讀易懂，不管是 Class 的名稱，還是 Class 內部 member method 和 member data 的名稱，甚至是函式裡的 local variables 名稱，好的命名都可以提高可讀性，好的名稱必須是名符其實。

### 8. Lack of comments (缺乏註解)

良好的註解可以提高可讀性，因此在適當的地點寫上註解，可以幫助自己在以後維護時瞭解程式碼，以下幾個情況建議寫註解：(1) JavaDoc 式的註解，這類的註解在將來可以輸出成 API 文件，有的 IDE 甚至可以解析這些註解，並在呼叫時立即顯示用途；(2) 複雜的演算法；(3) 用寫註解輔助設計與思考。

### 9. Inconsistent coding standard (不一致的程式碼標準) ☹

我們的課程並不要求學生使用特定的**程式碼標準**(Coding standard)，但是，我們希望你的作業程式採用固定的程式碼標準。如果你不熟悉任何程式碼標準，請參考Java Coding Style Guide [2]或是GNU Coding Standard [3]，如果你覺得上述二者太過複雜，也可以自行制定自己的標準。但所謂的標準就是重頭到尾都是一致的，所以繳交的程式碼中不應該有不一致的情況發生，例如：一旦決定左大括號(Open brace)要換行並縮排，那所有程式碼中的左大括號就必須全部換行並縮排，沒有例外。其實IDE都有設定可以幫助自動格式化(Auto Format)，請善用IDE

避免被扣分。

### 10. Fat view (臃腫的外表)

設計視窗程式時，作業會要求使用 Model-View-Controller (MVC)或 Presentation Model 等設計樣式(Design pattern)，以提升程式品質。另外，為提升可測性(Testability)，View 必須越薄越好，也就是 View 程式中不應該含有任何邏輯，這些邏輯應該在 Model (或 Presentation model)中實作，View 只是呼叫 Model 的函式，如此一來，測試時就不需要 GUI，可以使用單元測試框架測試 Model。

### 11. Unresolved warnings (忽視警告) ☹

當使用 IDE (例如 Visual Studio)編譯時，通常有警告(Warnings)時仍然可以產生可執行檔，而且可以執行。雖然警告不是語法上的錯誤(Errors)，但警告可能代表隱藏的錯誤，或是執行期間才會發生的錯誤，因此，所有繳交的作業除了編譯應該沒有錯誤之外，也不允許有任何警告。

### 12. Literal constants (字面常數) ☹

寫程式時會多少都會用到常數(Constants)，若將常數值(Value)直接寫在程式中(稱為字面常數)，當需要維護時，往往看到常數值卻無法判斷這個值代表什麼意義。另外，當常數會被重複使用時，一旦要修改就必須在眾多檔案和程式碼中修改，非常麻煩。為避免使用字面常數，應該先用(static) const 或 define 關鍵字(視語言而異)定義常數，然後再使用之。

## 研究所相關的壞味道

研究所課程要求更為嚴格，因此除了常見的壞味道外，額外增加下列四個壞味道，以增加程式碼的設計完整度。

### 13. Speculative generality (夸夸其談未來性)

根據需求設計程式時，請遵守 Keep it simple and straightforward (KISS)準則，不要過度假設未來可能需要的功能，替程式碼加入過度複雜的設計，這些設計不但在未來可能連一次都沒使用過，反而增加理解程式碼的門檻。請注意，設計的標準在取得平衡，要加入任何的設計都要考量到增加的複雜度跟成本，如果不值得就別加，如果值得，那就放手做吧！

### 14. Switch statement (善用多型)

物件導向程式設計的一個明顯特徵就是：少用 switch-case。從本質上來說 switch 的問題就是重複，而且 switch 會散落在不同地點，當要新增一個 case 子句時，必須找到所有的 switch 並修改。事實上，物件導向中的多型(Polymorphism)

概念可為此帶來優雅的解決辦法。

### 15. Message Chains (小心陌生人)

如果程式碼中出現客戶向一個物件請求另一個物件，然後在向後者再請求另一個物件，然後在請求另一個物件...，這就是 Message Chain，Message Chain 中的物件對客戶來說應該是陌生人，卻因為 Message Chain 的關係，客戶必須依賴(產生 coupling)這些陌生人，一旦物件關係產生任何變化，客戶就不得不做出改變。此時可使用 *Hide Delegate* 讓替客戶與陌生人之間加入合適的 Middle Man。注意，有時 Message Chain 是可以接受的，但大多數是可以避免的。

### 16. Refused bequest (被拒絕的遺贈)

設計繼承架構時，subclass 會繼承 superclass 的函式與資料，但如果 subclass 不想或不需要繼承，或 subclass 只需要其中一個或少數透過繼承得到的函式或資料時該怎麼辦呢？此時，應該重新思考繼承體系架構的設計，可以使用 *Push Down Method* 和 *Push Down Field* 調整繼承架構，或使用 *Replace Inheritance with Delegation* 直接移除繼承架構。注意，**繼承應該只發生在類別之間是 Is-A 的關係**。

## Reference


[1] Martin Fowler, Ken Beck, John Brant, William Opdyke, and Don Roberts. Refactoring, Improving the Design of Existing Code. 1st Ed. Addison-Wesley, 1999. ISBN: 0201485672.

[2] Achut Reddy, Java Coding Style Guide, May 2000, http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf

[3] GNU Coding Standard, http://www.gnu.org/prep/standards/


## Revison History

| Date | Author | Description |
|---|---|---|
| 2010/09/13 | Pin-Ying Tu | First Draft |
| | | |
| | | |
| | | |