

程式碼品質批改原則

2015-09-16版

*Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.
- Martin Fowler.*

任何傻瓜都能寫出電腦能懂的程式碼。而只有好的程式設計師才能寫出人能懂的程式碼
- Martin Fowler.

*Quality is not an act. It is a habit.
- Aristotle*
品質不是動作，是一種習慣
- Aristotle

寫程式實作功能不難，但是要寫出易懂、易改的程式碼則很難，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。每一個壞味道都有標準的記錄方式。

我們使用「Dr. Smell」壞味道檢測工具掃描你的程式，你可以好好利用這個工具所抓到的壞味道並且一一修正。但是自動化掃描工具還是有些限制，對於及壞味道嚴重程度，例如 Long Method 可以有些容忍空間，遇到此狀況時，助教將會以人工判定是否確實為壞味道，如果不構成壞味道，助教會將其加入忽略清單。我們建議你可以直接將壞味道全數修正。另外，請注意，雖然 Dr. Smell 能檢測大部分的壞味道，但此工具無法檢測所有的壞味道，你仍然需要注意本文提及的壞味道規則，並自行修正。

助教批改作業時會使用標準方式記錄你的壞味道，使得你知道哪裡有問題。當你自已發現或被告知有壞味道時，應該盡快依範例說明的方法修改程式，去除壞味道。由於作業有連貫性，同一個壞味道必須盡早去除，才不會被重複扣分。另外，當進行 peer review 時，你必須使用標準方式記錄其他同學的壞味道，以供評分。

備註：作業評分時，View 的 Duplicated code 評分標準比較寬，而 Unit Test 的評分則只管標示⊗的項目。另外，當使用 Visual Studio 的 Designer 設計使用者介面時，請不用管 Designer 所產生及維護的 code 的壞味道，但是，各控制項(control)變數命名及其事件處理函式(Event Handler)命名，仍應遵守命名規則，並避免名不符實。

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. Fat view (臃腫的外表)
10. Unresolved warnings (忽視警告) ⊗
11. Literal constants (字面常數)
12. Message chains (小心陌生人) ⊗
13. Static methods and fields (濫用 Static)
14. Inconsistent coding standard (不一致的程式碼標準) ⊗

Duplicated code (重複的程式碼)

介紹

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。

範例

範例一：一段**完全重複的程式碼**在一個或多個類別或方法中出現。

```
1 public class ClassAReport
2 {
3     ...
4     public int CalculateAverage(List<Integer> scores)
5     {
6         int sum = 0;
7         int average;
8         for (int i = 0; i < scores.size(); i++)
9             sum += scores.get(i);
10            average = sum / scores.size();
11            return average;
12        }
13    ...
14 }
```

```
1 public class ClassBReport
2 {
3     ...
4     public int CalculateAverage(List<Integer> scores)
5     {
6         int sum = 0;
7         int average;
8         for (int i = 0; i < scores.size(); i++)
9             sum += scores.get(i);
10            average = sum / scores.size();
11            return average;
12        }
13    ...
14 }
```

重複的程式碼應抽出來改寫成類別或副程式讓其他程式共享。

```
1 public class AverageCalculator
2 {
3     ...
4     public int CalculateAverage(List<Integer> scores)
5     {
6         int sum = 0;
7         int average;
8         for (int i = 0; i < scores.size(); i++)
9             sum += scores.get(i);
10        average = sum / scores.size();
11        return average;
12    }
13    ...
14 }
```

Better Solution

範例二：一段相似度很高的程式碼在一個或多個類別或方法中出現。

```
1 public void DoSomething()
2 {
3     Person angelina = new Person();
4     angelina.SetFirstName("Angelina");
5     angelina.SetLastName("Jolie");
6     angelina.SetAge(16);
7     angelina.SetGender(Gender.Female);
8
9     Person james = new Person();
10    james.SetFirstName("James");
11    mary.SetMiddleName("Robert");
12    james.SetLastName("Chen");
    james.SetAge(17);
    james.SetGender(Gender.Male);
}
```

將相似的程式碼抽出為副程式讓其他程式共享

```
1 public Person createPerson(String firstName, String
2     middleName, String lastName, int age, Gender gender)
3 {
```

	<pre> 4 Person person = new Person(); 5 person.SetFirstName(firstName); 6 person.SetMiddleName(middleName); 7 person.SetLastName(lastName); 8 person.SetAge(age); 9 person.SetGender(gender); 10 return person; 11 } 12 13 public void DoSomething() 14 { 15 Person angelina = createPerson("angelina", null, "Jolie", 16, Gender.Female); 16 Person james = createPerson("James", "Robert", "Chen", 17, Gender.Male); 17 } </pre>
記錄 方式	<p>對於重複程式碼的每一行都指定為一個 Duplicate code 的壞味道，因此若有 n 斷重複的程式碼，且每一段有 m 行程式碼重複，則以 $((n-1)*m)$ 的個數為壞味道個數計算。</p> <p>以範例來看 ClassAReport 的第 4~11 行與 ClassBReport 的第 4~11 行為 Duplicate code，則指定 ClassBReport 的第 4~11 行為 Duplicate code。</p>

Long method (過長函式)

介紹

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，就是 Long Method。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。

範例

```

1  private void CreateSimpleTool()
2  {
3      ToolStrip tool = new ToolStrip();
4      tool.Parent = this;
5
6      toolUndo = new ToolStripButton("Undo", null,
UndoHandler);
7      toolUndo.Enabled = false;
8      toolUndo.Image = Resources.undo;
9      tool.Items.Add(toolUndo);
10
11     toolRedo = new ToolStripButton("Redo", null,
RedoHandler);
12     toolRedo.Enabled = false;
13     toolRedo.Image = Resources.redo;
14     tool.Items.Add(toolRedo);
15
16     tool.Items.Add(new ToolStripSeparator());
17
18     toolPointer = new ToolStripButton("Pointer", null,
Clicked_itemPointer);
19     toolPointer.Enabled = m.G_type_pointer;
20     toolPointer.Image = Resources.pointer;
21     tool.Items.Add(toolPointer);
22
23     toolRectangle = new ToolStripButton("Rectangle", null,
Clicked_itemRectangle);
24     toolRectangle.Image = Resources.rec;

```

	<pre> 25 tool.Items.Add(toolRectangle); 26 27 toolEllipse = new ToolStripButton("Ellipse", null, Clicked_itemEllipse); 28 toolEllipse.Image = Resources.elli; 29 tool.Items.Add(toolEllipse); 30 31 toolLine = new ToolStripButton("Line", null, Clicked_itemLine); 32 toolLine.Image = Resources.line; 33 tool.Items.Add(toolLine); 34 } </pre> <p>CreateSimpleTool 可以 17 行為切割點，分成兩個 sub method，CreateUndoRedoToolButton 與 CreateShapeToolButton，使得程式更容易閱讀理解，解決 Long method 的問題。</p>
記錄方式	<p>若有一個函式太長，則此函式的 Long method 壞味道個數為函式的總行數，但是註解不列入計算。以範例來看，CreateSimpleTool 為 Long method，則指定函式開頭那一行，private void CreateSimpleTool() 為 Long method，而此函式共有 34 行(含空白行)，故壞味道個數為 34。在判斷 Long method 時，20 行以內是綠燈，20~29 行是黃燈，由助教判定，30 行或以上時一律算 Long Method。</p>

Large class (過大類別)

介紹

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。

範例

```
1 public class 班長
2 {
3     public void 辦班遊()
4     {
5         規劃行程();
6         收錢();
7         ...
50     }
51
52     public void 維持秩序() {...}
110
111     public void 規劃行程() {...}
149
150     public void 收錢() {...}
250     ...
321 }
```

若是一個類別做了太多事情，導致 Large class，就必須將這種類別去做妥善的分工。

```
1 public class 班長
2 {
3     public void 辦班遊()
4     {
5         康樂.規劃行程();
6         總務.收錢();
7         ...
8     }
9
10     public void 維持秩序() {
11         風紀.維持秩序();
```

Better Solution

	12	}
	13	}
	1	public class 風紀
	2	{
	3	public void 維持秩序()
	4	{
	5	...
	6	}
	7	}
	8	public class 總務
	9	{
	10	public void 收錢()
	11	{
	12	...
	13	
	14	}
	15	public class 康樂
	16	{
	17	public void 規劃行程()
	18	{
	19	...
	20	}
	21	}
記錄 方式	<p>若一個類別太大(例如有數百行)，則在類別開頭的那一行指定一次 Large class，而註解不列入計算。以範例來看，班長這個類別總共有 321 行，為 Large class，則指定類別開頭的那一行，public class 班長，為 Large class。</p>	

Long parameter list (過長參數列)	
介紹	<p>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，而且一旦需要更多參數時，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 <i>Introduce Parameter Object</i> 重構方法將參數包裝成一個「參數物件」。</p>
範例	<div> <pre> 1 public class Member 2 { 3 public void CreateMember(4 Name name, 5 String country, 6 String postcode, 7 String city, 8 String street, 9 String house) 10 { 11 ... 12 } 13 }</pre> </div> <p>使用 Introduce Parameter Object 重構方法，將過長得參數包裝成一個「參數物件」。</p> <div> <pre> 1 public class Member 2 { 3 public void CreateMember(4 Name name, 5 Address address) 6 { 7 ... 8 } 9 }</pre> <div>Better Solution</div> </div>
記錄方式	<p>若一個函式的參數有相關性，可以包裝成單一物件，則依參數具有關聯性的個數為 Long parameter list 的壞味道個數。</p> <p>以範例來看，CreateMember 這個函式擁有 6 個參數，但是 country, postcode, city, street, house 具有關聯性，可以包裝成 address 物件，因此為 Long parameter list，則指定函式開頭的那一行，public void CreateMember() 為 Long parameter list，且壞味道個數為 6。</p>

	在判斷 Long parameter list 時，4 個參數或以內是綠燈，5 個參數是黃燈，由助教判定，6 個參數或以上時一律算 Long parameter list。	
Feature envy (依戀情節)		
介紹	物件設計的重點在於將資料與這些資料的操作行為封裝在一起，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。則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。	
範例	<pre>1 public void DoSomething() 2 { 3 ClassA a = new ClassA(); 4 int x = a.GetX(); 5 int y = a.GetY(); 6 int z = a.CalculateSomething(x + y, y); 7 a.SetZ(z); 8 }</pre> <p>使用 Move Method 將該函式移到另一個 class。</p> <pre>1 public ClassA 2 { 3 public void DoSomething() 4 { 5 z = CalculateSomething(x + y, y); 6 } 7 }</pre>	<div>Better Solution</div>
記錄方式	若一個函式對於另一個 class 的興趣高於自己本身的 class，則在函式依戀於其他類別的每一行程式碼的行數為 Feature envy 的壞味道個數。 以範例來看，DoSomething 這個函式中的運算依戀於 ClassA，屬於 Feature envy，則以此函式依戀其他類別的每一行程式碼的行數為壞味道個數計算，DoSomething() 中的第 3~7 行皆依戀於 ClassA，故此函式的 Feature envy 壞味道個數為 5。	

Data clumps (資料泥團)

介紹

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。

範例

```

1 public class Customer
2 {
3     private String _name;
4     private String _title;
5     private String _house;
6     private String _street;
7     private String _city;
8     private String _postcode;
9     private String _country;
10    ...
11 }
```

```

1 public class Staff
2 {
3     private String _lName;
4     private String _fName;
5     private String _house;
6     private String _street;
7     private String _city;
8     private String _postcode;
9     private String _country;
10    ...
11 }
```

使用 *Extract Class* 或 *Introduce Parameter Object* 將這些資料項包裝起來。

```

1 public class Address
2 {
3     private String _house;
4     private String _street;
5     private String _city;
6     private String _postcode;
7     private String _country;
8     ...
9 }
```

Better Solution

```

1 public class Customer
2 {
3     private String _name;
4     private String _title;
5     private Address _customerAddr;
6     ...
}
```

	<pre> 7 } 1 public class Staff 2 { 3 private String _lName; 4 private String _fName; 5 private Address _staffAddr; 6 ... 7 } </pre>
記錄 方式	<p>若在多個地方看見同一群 Data，則在每個地方的每一行 Data 各指定一次 Data clumps。以範例來看，類別 Customer 的第 5~9 行與類別 Staff 的第 5~9 行都有著同一群 Data，為 Data clumps，則在類別 Customer 的第 5~9 行與類別 Staff 的第 5~9 行的每一行都指定為一個 Data clumps 壞味道。</p>

Unsuitable naming (名不符實)

介紹 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。另外，拼字錯誤也屬於名不符實的範圍內。

範例

```

1 public class T
2 {
3     boolean _b = false;
4     public int Xyz(int x, int y,
5         int z)
6     {
7         int r = 0;
8         r = (x + y) * z / 2;
9         return r;
10    }
11 }
```

修改為合適的名稱。

```

1 public class Trapezoid
2 {
3     boolean _isIsosceles = false;
4     public int CalculateArea(int top, int bottom,
5         int height)
6     {
7         int area = 0;
8         area = (top + bottom) * height / 2;
9         return area;
10    }
11 }
```

Better Solution

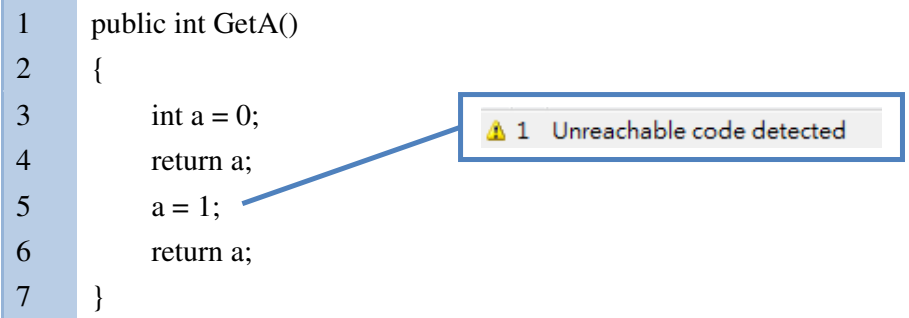
記錄方式 若命名不適當，則在此名稱第一次出現的那一行指定一次 **Unsuitable naming**，並且說明 **unsuitable naming** 是誰。主要有 **Class name**, **Member method**, **Member data**, **Local variable**。以範例來看，class name “T”不合適，則在第 1 行指定一次；variable “b”不合適，則在第 3 行指定一次；method “Xyz”不合適，則在第 4 行指定一次；又參數“x”，“y”，“z”不合適，則在第 4 行又指定兩次，而“z”因為換行了，所以在第 5 行再指定一次；variable “r”不合適，則在第 7 行又指定一次。

Lack of comments (缺乏註解)

介紹	<p>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。</p>
範例	<pre> 1 public class Trapezoid() 2 { 3 boolean _isIsoscle = false; 4 public int CalculateArea(int top, int bottom, 5 int height) 6 { 7 int area = 0; 8 area = (top + bottom) * height / 2; 9 return area; 10 } 11 }</pre> <p>為 CalculateArea 這個 method 加上註解</p> <pre> 1 public class Trapezoid() 2 { 3 boolean _isIsosceles = false; 4 //計算梯形面積並傳回計算結果 5 public int CalculateArea(int top, int bottom, 6 int height) 7 { 8 int area = 0; 9 area = (top + bottom) * height / 2; 10 return area; 11 } 12 }</pre> <p>Better Solution</p>
記錄方式	<p>除了單純的 get 與 set 的函式之外，為每一個 Member method 加上註解。若某個函式沒有註解，則在函式開頭的那一行指定一次 Lack of comments。以範例來看，CalculateArea 這個 method 缺少註解，則指定函式開頭那一行，public int CalculateArea(...)，為 Lack of comments。</p>

Unresolved warnings (忽視警告)

介紹	<p>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</p>
----	---

	(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存在。
範例	<pre> 1 public int GetA() 2 { 3 int a = 0; 4 return a; 5 a = 1; 6 return a; 7 } </pre> 
記錄方式	若程式中在很多行都有未解決的警告，則在每一個出現警告的那一行指定一次 Unresolved warnings 。以範例來看，第 5 行出現了警告，則指定第 5 行為 Unresolved warnings。

Fat view (臃腫的外表)	
介紹	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。
範例	<pre> 1 public class ViewForm() 2 { 3 private int _grade; 4 public ViewForm() 5 { 6 _grade = 59; 7 } 8 private void GetResultButton_Click(...) 9 { 10 If (_grade >= 60) 11 resultLabel.text = "pass"; 12 Else 13 resultLabel.text = "not pass"; 14 } 15 } </pre>

	之。
範例	<pre> 1 public double PotentialEnergy(double mass, double height) { 2 return mass * 9.81 * height; 3 }</pre> <p>定義常數。</p> <pre> 1 public double PotentialEnergy(double mass, double height) { 2 const double GRAVITATION = 9.81; 3 return mass * GRAVITATION * height; 4 }</pre>
記錄方式	<p>數字與字串都應先定義常數，而數字 0 與 1 可以不用定義。若程式中的某一行直接使用數字或字串，則在此行指定一次 Literal constants。注意，若是某一行(statement)有多個 Literal constants，則此行指定一次 Literal constants 即可，另外 View 裡面出現單次的 string Literal constants，可以不宣告 const。以範例來看，第 2 行直接使用 9.81 這個數字，則指定第 2 行，return mass * 9.81 * height;，為 Literal constants。</p>

Message chains (小心陌生人)	
介紹	<p>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 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。</p>
範例	<p>原本的兩個 class</p> <pre> 1 public class Person 2 { 3 Department _department; 4 public Department getDepartment() 5 { 6 return _department(); 7 } 8 }</pre> <pre> 1 public class Department 2 { 3 private person _manager</pre>

```

4      public Department(Person manager)
5      {
6          _manager = manager;
7      }
8      public Person getManager()
9      {
10         return _manager;
11     }
12 }

```

造成 Message Chain 的範例

```

1      public class Client
2      {
3          Person _john;
4          ....
11         public DoSomething
12         {
13             String _manager =
14                 _john.GetDepartment().GetManager().GetName();
15         }

```

如何修改：

在 Person 中加入一函式 getManager()

```

1      public class Person
2      {
3          Department _department;
4          ...
10         public Manager getManager()
11         {
12             return _department.getManager();
13         }
14     }

```

原本的 MessageChain 的 code 可修改為：

```

1      public class Client
2      {
3          Person _john;
4          ....
11         public DoSomething
12         {

```

	<pre> 13 String _manager = _john. GetManager().GetName(); 14 } 15 }</pre>
記錄方式	<p>若 Message Chains 的長度(深度)在兩層內是綠燈，例如：A.getName() 或 A.getB()，三層為黃燈，由助教判斷是否為 Message Chain，若判斷為壞味道，則在該行指定 Message Chain 並算一次壞味道，四層以上則是紅燈，在該行指定 Message Chain，每多一層多算一次壞味道，以範例來看原本的第 13 行有四層所以算一次。</p>

Static Methods and Fields (濫用 static)	
介紹	<p>一般而言，一個 class 的變數或函式並不需要設定為 static。當一個變數或函式設為 static 時，其使用就相當於 C 語言中的 global 變數或函式，可以讓我們很輕易地存取(不需要對方物件的 reference)，但是，眾所周知，我們應該避免使用 global 變數或函式，以免造成 code 維護與 debug 的困擾。因此，能不用 static 就盡量不要使用。以下是幾種適合使用 static 的情境，其他情況請勿使用 static。</p>
範例	<p>當宣告變數為 static 時，其值應該是常數，不能更動</p> <pre> 1 public class MyCircle 2 { 3 public static const float PI = 3.14159; 4 5 6 }</pre> <p>當宣告函式為 static 時，此函式應該是沒有狀態性的，也就是說，不論是在什麼情形下呼叫這個函式，只要參數相同，結果一定相同。</p> <pre> 1 public class MyMath 2 { 3 public static float Sqrt(float number) 4 { 5 int result = Math.Sqrt(number); 6 return result; 7 }</pre>

	<pre> 8 } </pre> <p>有時候，我們需要計算一個 class 總共用到幾個 instance，範例如下，然而這種計數器無法避免使用 static，因此不列入壞味道。</p> <pre> 1 public class Client 2 { 3 private static int InstanceCounter = 0; 4 11 public Client //建構子 12 { 13 InstanceCounter++; 14 } 15 } </pre>
記錄方式	除了上述例外，程式碼中不可出現 static 的變數或方法。非上述之 static 的使用，每一個紀錄為一個壞味道

Inconsistent coding standard (不一致的程式碼標準)	
介紹	我們的課程將要求學生使用特定的程式碼標準(Coding standard)，請看以下範例。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。
範例	<p>空白與空行</p> <p>在下列位置加上空白：</p> <ol style="list-style-type: none"> 函式若有兩個以上的參數，請在逗號後面加一個空白 do while 的 while 關鍵字前面與後面加一個空白 for, foreach, while, switch, if 等關鍵字與左括號之間加一個空白 <pre> 1 public int CalculateAverage(List<Integer> scores, int scoresSize) 2 { 3 int sum = 0; 4 int average = 0; 5 int i = 0; 6 if (scoresSize != 0) 7 { 8 do 9 { 10 sum += scores.get(i); </pre> <p>圖中標註了三個需要加空白的地方：</p> <ul style="list-style-type: none"> 標註 1：在 if 和 (之間。 標註 2：在 sum += scores.get(i); 後面。 標註 3：在 int i = 0; 後面。

```

11         i++;
12     } while (i < scoresSize)
13         average = sum / scoresSize;
14     }
15     return average;
16 }

```

4. for loop 裡面的三個敘述區，在分號後面加一個空白
5. for loop 裡面若有多個敘述，在逗號後面加一個空白
6. 條件敘述(?)中，問號後面加一個空白，冒號前後各加一個空白
7. 二元運算子的前面與後面各加一個空白
 - 位元運算： ^, &, |, >>, <<
 - 邏輯運算： &&, ||, ==, !=
 - 關係運算： <, <=, >, >=
 - 指派運算： =
 - 複合運算： +=, -=, /=, *=, >>=, <<=, &=, |=, ^=
 - 算術運算： +, -, *, /, %

```

1   for (int i = 0, j = 0; i < j + 1; i++, j++)
2   {
3       j = (i % 2 == 0) ? i : i * 2;
4   }

```

8. 繼承符號，冒號，的前面與後面各加一個空白
9. 陣列初始化的左大括號後面，右大括號前面

```

1   public class SubClass : BaseClass
2   {
3       int[] array = { 1, 2, 3 };
4       ...
5   }

```

在下列位置不加上空白：

10. 函式的第一個參數與左括號之間不加上空白
11. 函式名稱與左括號之間不加上空白
12. 函式最後一個參數與右括號之間不加上空白
13. 轉型(Type casting)宣告與變數間不加上空白
14. 每一行程式碼最後結束分號的前面不加上空白

15. 不亂加多個空白

```

1   public double CalculateArea(int top, int bottom, int height)
2   {

```

```

3      int area = 0;
4      area = (top + bottom) * height / 2 ;
5      return (double)area;
6  }
```

13

14, 錯誤示範

空行：

15, 錯誤示範

16. 每個函式之間加一個空行*

17. 一次不加多個空行(若發生時，則指定在多個空行的第一行即可)*

```

1  public class A
2  {
3      public void X()
4      {
5          ...
6      }
7
8      public void Y()
9      {
10         ...
11     }
12
13
14
15     public void Z()
16     {
17         ...
18     }
19 }
```

16

17, 錯誤示範

縮排與換行

1. 左大括號一律換行
2. 右大括號一律換行，並與左大括號對齊*
3. 左大括號後，一律換行撰寫程式碼
4. 縮排一律使用 tab(空 4 格)*
5. 程式碼依內容階層深度縮排*
6. 一行不能有多個 statements
7. if 和 else 後要換行(if 或 else 所屬的敘述不可與 if 或 else 同行)

```

1  public class Trapezoid()
2  {
```

1

3

```

3      boolean _isIsoscles = false;
4      public int calculateArea(int top, int bottom, int height)
5      {
6          if (top < 1 || bottom < 1 || height < 1 )
7              throw new ArgumentNullException("Error");
8          int area = 0; int a = 0;
9          area = (top + bottom) * height / 2;
10         return area;
11     }
12 }

```

Annotations in the image:

- Box 4 points to the `if` statement.
- Box 7 points to the `throw new ArgumentNullException("Error");` line.
- Box 5 points to the `return area;` line.
- Box 6 points to the calculation `area = (top + bottom) * height / 2;` with the note "6, 錯誤示範" (6, error example).
- Box 2 points to the closing brace of the `calculateArea` method.

類別宣告

1. 除非為類別常數(static class constants)，成員變數請勿宣告為 public
2. 類別名稱首字母大寫，若由多個單字組成，每個單字的第一個字母大寫，其餘小寫。不以底線區隔單字*
3. 一個 file 只能有一個 namespace
4. 一個 file 只能有一個 class

```

1      public class FixedForm
2      {
3          public static string TOHTML = "toHtml";
4          private int _tableRow;
5          private int _tableColumn;
6      }

```

Annotations in the image:

- Box 2 points to the `FixedForm` class name.
- Box 1 points to the opening brace of the class.

變數

1. Class 成員變數以底線開始，底線後第一個字母小寫，若由多個單字組成，從第二個單字開始，每個單字的第一個字母大寫，其餘小寫。不以底線區隔單字*
2. 區域變數，函式參數命名規則與 class 成員變數相似，僅差在不以底線開始*
3. 除了函式參數，其餘一行只宣告一個變數*
4. 定義字面常數的名稱的所有字母大寫，並以底線隔開單字*

```

1      public class FixedForm
2      {
3          public static string TOHTML_METHOD = "toHtml";
4          private int _tableRow;
5          private int _tableColumn;
6          public FixedForm(int row, int column)
7          {
8              _tableRow = row;

```

Annotations in the image:

- Box 4 points to the `FixedForm` class name.
- Box 1 points to the `TOHTML_METHOD` constant.
- Box 2 points to the `_tableColumn` variable.
- Box 3 points to the `FixedForm` constructor.



```
9         _tableColumn = column;
10     }
11 }
```

列舉(enum)

1. 一行只有一個項目。
2. 第一個字母為大寫，其餘小寫。
3. 最後一個項目不用加逗號。

```
1 enum Days
2 {
3     Mon,
4     Tue,
5     Wed,
6     Thu,
7     Fri,
8     Sat,
9     Sun
10 };

```





Delegate 與 Event

1. delegate 與 event 都宣告於 class 的最上方
2. delegate 的命名規則為第一個字母大寫，結尾為 EventHandler
3. event 的變數命名如一般 class 中的成員變數

```
1 public class Publisher
2 {
3     public delegate void SampleEventHandler(object sender,
4     SampleEventArgs e);
5     public event SampleEventHandler _sampleEvent;
6
7     protected virtual void RaiseSampleEvent()
8     {
9         if (_sampleEvent != null)
10         {
11             _sampleEvent (this,
12                             new SampleEventArgs("Hello"));
13         }
14     }
15 }

```



函式

1. 函式命名一律大寫開始，若由多個單字組成，每個單字的第一個字母大寫，其餘小寫。不以底線區隔單字*
2. 每個函式表示一種操作，因此一律使用動詞做為第一個單字*
3. Getter 除了取得 bool 值使用 IsXXX() 之外，其餘一律使用 GetXXX()*
4. Setter 一律使用 SetXXX()*
5. Getter 與 setter 一律使用 property

```
1 public class FixedForm
2 {
3     private int _tableRow;
4     private int _tableColumn;
5
6     public FixedForm(int row, int column)
7     {
8         _tableRow = row;
9         _tableColumn = column;
10    }
11
12    public void ShowTable()
13    {
14        ...
15    }
16
17    public void SetTableRow(int row)
18    {
19        _tableRow = row;
20    }
21
22    public int GetTableRow()
23    {
24        Return _tableRow;
25    }
26
27    public int TableColumn
28    {
29        set
30        {
31            _tableColumn = value;
32        }
33        get
34        {
35            Return _tableColumn;
```

	<pre> 36 } 37 } 38 }</pre>
記錄 方式	<p>若是不符合以上規定，則不符合一個位置，就指定一次 Inconsistent coding standard，並說明是哪一個項目。若為類別宣告，變數，函式等名稱相關，則指定它第一次出現的位置即可。若某一行(statement)有多個同一項目的 Inconsistent coding standard，則此行指定此項目的 Inconsistent coding standard 一次即可。以範例的第四段程式碼來看，第四行的位置，違反了空白與空行的第 14 個規定，就在第四行指定一次 Inconsistent coding standard，並且說明 空白與空行-14。</p>

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/>