類別

- **封裝**

一般來說我們會將不需要公開的變數或是函式保持私有的型態。

但如果因應測試需求，此部分不需要太過於堅持（可更改為 protected or package）。

- **類別要夠簡短**

同函式規則，類別要夠簡短。 >>>>> • **簡短**

準則一，要簡短。

準則二，比準則一更短。

嘿... 所以到底想要多短咧？

函式部分我們去計算**程式碼的行數**，而類別則是去計算**職責的數量**。

我們也可以透過觀察一個類別的名稱去判斷職責的數量。

manager 或是 super 相關的類別名稱可能就在暗示有過多的職責巨集。通常啦

- **單一職責原則**

Single Responsibility Principle (SRP) 單一職責原則。

此原則主張一個類別或是一個模組應該只有**一個修改的理由**，通常也是**唯一的職責**。

試著確認類別中職責歸屬通常可以抽離更多的抽象概念，這些概念就是更多的小型類別。

單一職責原則是一個極容易被遵守但也是極容易被忽略的概念。

讓軟體能夠運作以及**讓軟體保持整潔**是完全不同的事情。

通常~~人類腦袋記憶體有限~~一次只能執行上述兩者其一。

在時間有限下執行任務，**讓軟體能夠運作**絕對是優先考量。

但是在我們完成當下任務時，是否就覺得工作已經完成了？

~~你正在看我的筆記，我想你必須知道我一定會說真的還沒有。~~

你會接著執行下一個任務（即使是偷跑），而忘記回頭去檢視自己的程式是否乾淨且有組織性的拆解各個類別相關職責。

有部分開發者認為，當各個類別僅僅擁有單一職責，在一個大型系統中是否就會有過多的小型類別，且在了解系統的過程時難以一次看到全貌？在瀏覽程式的時候往往必須從 A類 跳到 B類 閱讀實作細節等瑣碎的過程。

Clean Code 的角度，在大型系統中大量的邏輯以及複雜性無可避免，所以首要目標就是**組織**各個類別。在規劃系統結構或是 package 的時候就必須使得未來的開發者能夠知道去某個地方尋找他所需要的元件。

Clean Code 喜歡數量多且擁有良好標記及定義的小型抽屜組合封裝大型功能，而不是在數量少的大型抽屜把所有的東西都塞進去。

Clean Code 主張，每個小類別

『封裝單一的職責』

『只有一個修改的理由』

『與其他少數幾個類別合作完成系統所要求的行為』

- **凝聚性**

當類別中的方法操作的實體變數越多，代表該方法越凝聚於該類別。若類別中的每個實體變數都被使用在每個方法，我們可以稱之為**最大凝聚性的類別**但是很難發生。

建議使類別凝聚性高一點，使得實體變數與方法互相依賴結合成一個**邏輯上的整體**。

當採取『類別需保持函式的簡短以及函式參數串列要夠少』的策略，實體變數可能也會跟著增加。當此情況發生的時候也幾乎代表著可以分離出一些更高凝聚性的小型類別。反之，當類別喪失凝聚性的時候，我們就應合理的拆解它們使得類別組織結構更為透明。