CHAPTER 12 軟體工程

12-1 寫程式

- 當開始接觸資訊科學，就開始在計算機程式課程中學習寫程式，我們學習變數的宣告，學習輸入輸出，學習使用迴圈，當然也學習編譯程式。
- 我們被教導如何寫好程式，可是卻很少被教導如何將程式寫得好。
- 很多人對於撰寫程式非常有興趣，總是想著如何以更簡短更省工的方式達到功能，並且在逐步減少程式碼行數中得到最大的成就感。

12-1 寫程式

- 在撰寫程式的經驗越來越多之後，很多人開始同意寫程式的最高準則，也就是K.I.S.S.。這四個字代表的意思是Keep It Simple and Stupid。
- 所謂K.I.S.S.準則是說：寫程式的時候，盡量讓程式碼看起來非常簡單而易懂，不必為了省幾行程式碼而過度用一些花俏的招數。理由很簡單，我們來回顧一下過往的經驗即可明瞭。

資 訊 科 技 專 欄

河內塔(Tower of Hanoi)

河內塔問題於1883年由一位法國數學家在報章上刊載出，成為一個動動腦問題。它有一個不可考的神秘傳說，源自古印度神廟中的一段故事。

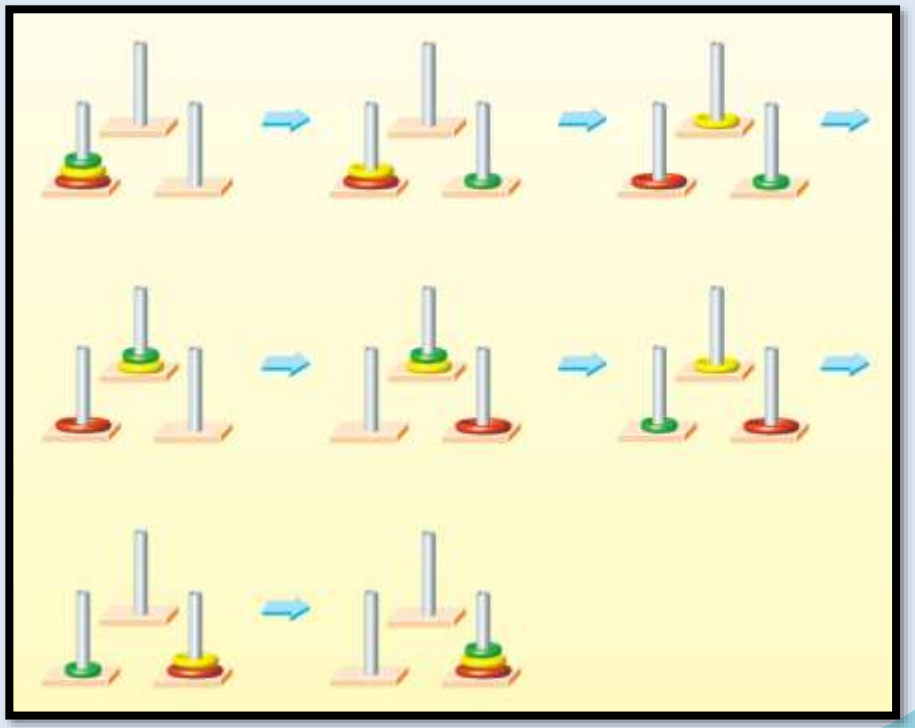
據說在古印度有一神廟是宇宙的中心，神廟內插有三條木柱，其中一個木柱由下到上套有共64個由大到小依序直徑遞減的環型金屬片。天神指示僧侶們要想辦法把這64個環型金屬片從原本的木柱移動到另一個木柱上面。

資 訊 科 技 專 欄

移動時必須一次只能動一個金屬片，並且所有的過程中，不論金屬片暫套在哪個木柱上，都得遵循下面的金屬片直徑得比在上面的金屬片直徑大的原則。直到僧侶們把64個金屬片都完整搬運到另一個木柱之後，西方極樂世界將到來。

資 訊 科 技 專 欄

這是一般人樂於鬥智的動動腦問題，也是學習演算法時很重要的一個演算法課題。



12-1 寫程式

- K.I.S.S法則中最重要的就是要維持程式碼簡單又乾淨，也就是讓程式碼擁有良好的**可讀性**(readability)，不僅是讓自己幾個月後還能夠理解程式碼在做些什麼，同時也要能讓他人輕易地了解自己寫的程式碼。
- 使得現任的人接手他人所寫的程式碼，或者程式碼交由別人改版或**除錯**(debug)。

12-1 寫程式

- 維持良好的可讀性包括變數名稱應該取得妥當，有些人草率地將變數命名為a、b、c這一類，或者是temp1、temp2等等，雖然在寫程式的當下較省功夫，不必花心思為變數取一個好名稱，可是日後再讀取該段程式碼時，勢必得花心思回想到底temp1所代表的意義是什麼？
- 為何要將temp2的值給temp1？舉最簡單的C程式碼例子如下：

12-1 寫程式

```
int function1  (int temp1)
int temp2;
       if  (temp1 == 1 || temp1 ==2)  temp2 = 1;
       else  temp2  =  function1  (temp1  -  2)  +
function1  (temp2 - 1) ;
       return temp2;
```

- 上面的程式碼，各位可以一眼即辨出這個函式在做什麼事情嗎？如果我們把它取上比較好的名字，並且加以有條理的排版，改寫為如下的情況呢？

12-1 寫程式

- 互相比較起來，下面這一段程式碼是不是讓人一目了然呢？

```
int fibonacci  (int n)
{
        int ans;
        if  (n == 1 || n == 2)
                ans = 1;
        else
                ans = fibonacci  (n - 2)  + fibonacci
 (n - 1) ;
        return  (ans) ;
}
```

12-1 寫程式

- 這個程式很清楚的將函式標明清楚，取名為fibonacci，至少告訴看到程式碼的人說：這一段函式與費伯納西有關。
- 再進一步看，可以看到是一個遞迴(recursive)函式，一開頭就先標明終止的條件就是當n為1或者是n為2的時候，如果不是屬於終止條件，則按照費伯納西數列的定義，令$a_n=a_{n-2}+a_{n-1}$運算。

12-1 寫程式

- 程式碼的撰寫中，應該要有統一的命名準則(naming)，譬如說變數名稱是否以大寫開頭？函數名稱是否以大寫開頭？
- 如果有變數名稱是由兩個單字串成，譬如說某一點的x座標，是要取成positionX呢？還是要取成position_x？這些應該都要有統一的準則，尤其是當多人合寫程式時，更應該統一以避免程式碼看起來雜亂無章。

12-1 寫程式

- 並且程式碼的排版也應該順著程式的語意而來，該換行的地方就該換行，該空白的地方就空白，也可以讓程式讀起來比較輕鬆不擁擠。
- 除了程式碼的可讀性之外，另外還有一大重點在於程式碼的**可靠性**(reliability)，一個好的程式員所撰寫的程式應該是要能夠很可靠，禁得起使用者各種出奇不意的使用操練。

12-1 寫程式

- 當撰寫大型程式的時候，每撰寫一小片段的程式就應該寫一些測試確保其有良好的可靠性，然後再繼續發展程式。
- 這樣的作法雖然囉嗦，可是可以盡可能確保每一小片段能不出錯。否則整個程式整合運作時，一旦程式直接當機，麻煩可就大了。因為程式碼長的不得了，也不知道程式當在哪一個區塊，要慢慢檢測出錯誤發生在哪裡是得花一番功夫的。

12-1 寫程式

- 理解可靠性的重要之後，我們再回過頭來檢視上面那段程式碼。仔細瞧瞧，會驚呼上面那樣的寫法其實是非常不可靠的程式碼。
- 你已經看出問題出在哪裡了嗎？當我們把n以5,7,8等等的正整數代進去都可以讓函式正常運作，可是如果不小心使得n的值為0，甚至是負數，會發生什麼慘劇呢？答案是這個函式將不停的遞迴下去呼叫，直到程式當掉為止。

12-1 寫程式

- 因此，最簡單的方法就是加入判斷句，避免n為負數的情況之下還遞迴呼叫，並且在螢幕上顯示出訊息，告訴使用者n的值應該要大於0。於是我們再將程式碼改寫如下：

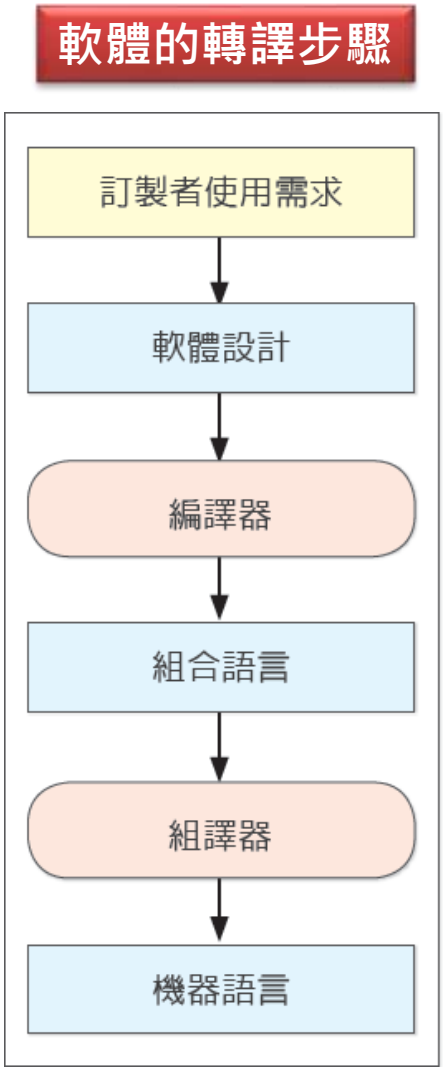
```
int fibonacci  (int n)
{
       int ans;
     if  (n < 1) {
         printf  ("n > 0 is needed\n") ;
         return -1;
     }
       if  (n == 1 || n == 2)
             ans = 1;
       else
             ans = fibonacci  (n - 2)  + fibonacci
 (n - 1) ;
       return  (ans) ;
}
```

12-1 寫程式

- 如此一來，就能夠避免輸入為負數時可能造成的錯誤。
- 然而實際上這樣還是不夠完善，譬如說，如果輸入是字母或者符號呢？因此寫程式的時候應該要小心考慮各種情況以增加程式的可靠性，隨著練習越多，就會越有經驗知道如何寫出可靠性高的程式碼，在此就不一一描述該如何補強。

12-1 寫程式

- 除了可讀性及可靠性之外，寫程式還該注意的地方是**註解**(comment)的撰寫。
- 程式碼寫好之後，經由**編譯器**(compiler)、**組譯器**(assembler)譯為機器看的**機器碼**(machine code)，而註解的地方則是寫給程式撰寫員看的。編譯器會忽略掉註解的地方進行編譯。



12-1 寫程式

- 在C語言中，註解是以/*...*/表示，在/*及*/中間的文字會被編譯器忽略。如果是單行的註解，則以//表示，在//後面的文字編譯器將忽略。
- 註解不外乎是在程式碼的開頭註明這份程式碼的功能說明，或者使用什麼特殊的演算法，撰寫的日期、改版的日期，版本的標示以及撰寫的程式開發人員為誰等等的資訊，或者是在函式之前用文字說明某一個函式的功能、變數的使用說明，或者是在程式語意較不顯著的地方用以補強說明，好使日後自己或他人還能很快地理解整個程式。

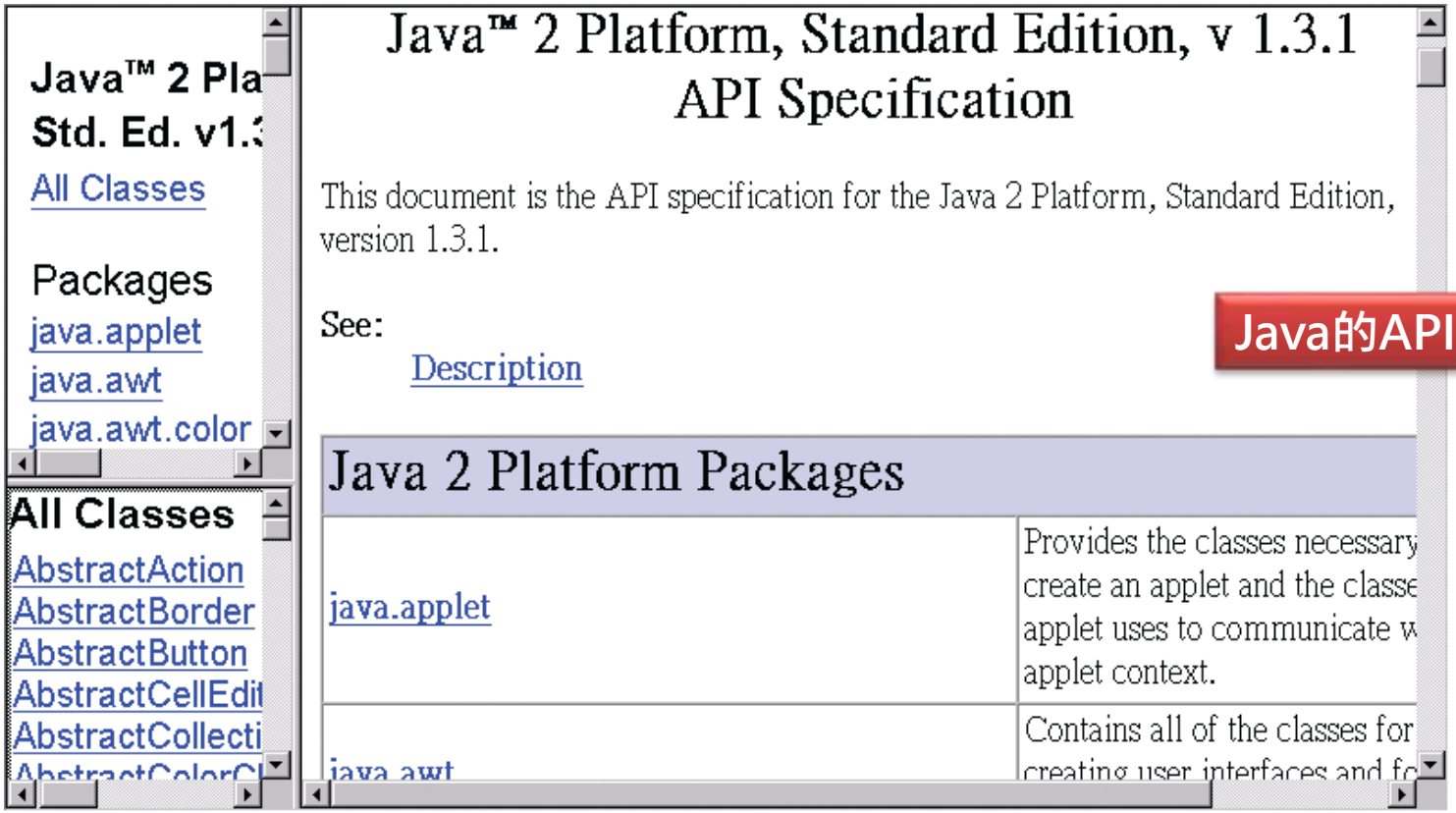
12-1 寫程式

- 有些時候所撰寫的函式不只是自己使用，也可能是發展供他人使用的函式，或者自己定義出一些類別(class)可供他人使用，在這種情況下，只要撰寫註解的方式合乎註解的特殊規定，就可以經由指令直接產生API(Application Programming Interface)的說明文件。
- 我們就把前面的程式碼加上註解如下：

12-1 寫程式

```
/ *
  * Computes the nth Fibonacci number
  * Fibonacci number：an = an-2 + an-1  (n>0，if
    n =1 or n =2，an = 1)
  * Version：1.4
  * Date：  2004/10/20
  * Author：Mary Lin
  */
int fibonacci  (int n)
{
       int ans;
     if  (n < 1) {
         printf ("n > 0 is needed\n") ;
         return -1;                              //means invalid input
     }
       if  (n == 1 || n == 2)                    //the terminal condition
          ans = 1;                               //a1 = 1, a2 = 1
       else
          ans = fibonacci (n - 2)  + fibonacci (n - 1) ;       //an=an-2+an-1
       return  (ans) ;
}
```

12-1 寫程式



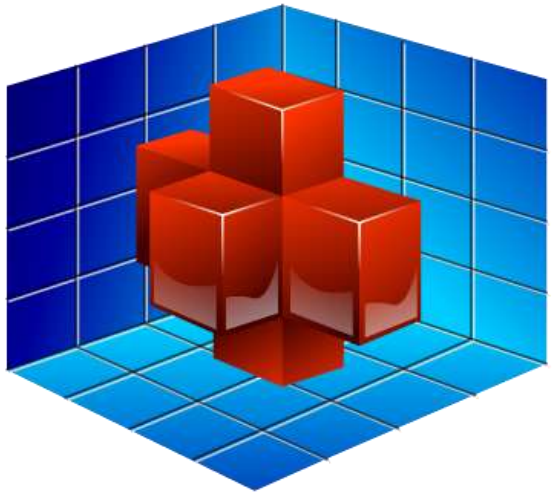
Java的API文件

12-1 寫程式

- 要把程式寫好其實需要練習與學習，有很多的技巧與方式有待大家去琢磨。
- 有些技巧是需要靠智慧去學習，而有些制式的項目則可以靠**電腦輔助軟體工程**(Computer-Aided Software Engineering；CASE)來幫忙，CASE可以針對程式碼的設計、程式碼的撰寫風格、所撰寫出的程式碼效能進行測試。

12-2 軟體開發生命週期

- 瀑布式模型
- 螺旋式模型
- 需求分析
- 設計
- 編碼
- 測試
- 維護



12-2 軟體開發生命週期

- 平常我們寫小程式，可能是看了作業規範之後，信手拈來就開始進行程式碼撰寫。
- 然而，當我們進行較大型的程式開發時，就無法如此隨心所欲。
- 為了使整個程式碼擁有較好的架構，我們會先花一些時間考量如何規劃。至於商用軟體的發展，整個過程就更為複雜了。

12-2 軟體開發生命週期

- 軟體開發生命週期(software development life cycle)中有幾個階段：需求分析、設計、編碼、測試、維護。
- 軟體開發生命週期循環的方式，一般分成兩種：

瀑布式模型 (waterfall model)

螺旋式模型 (spiral model)

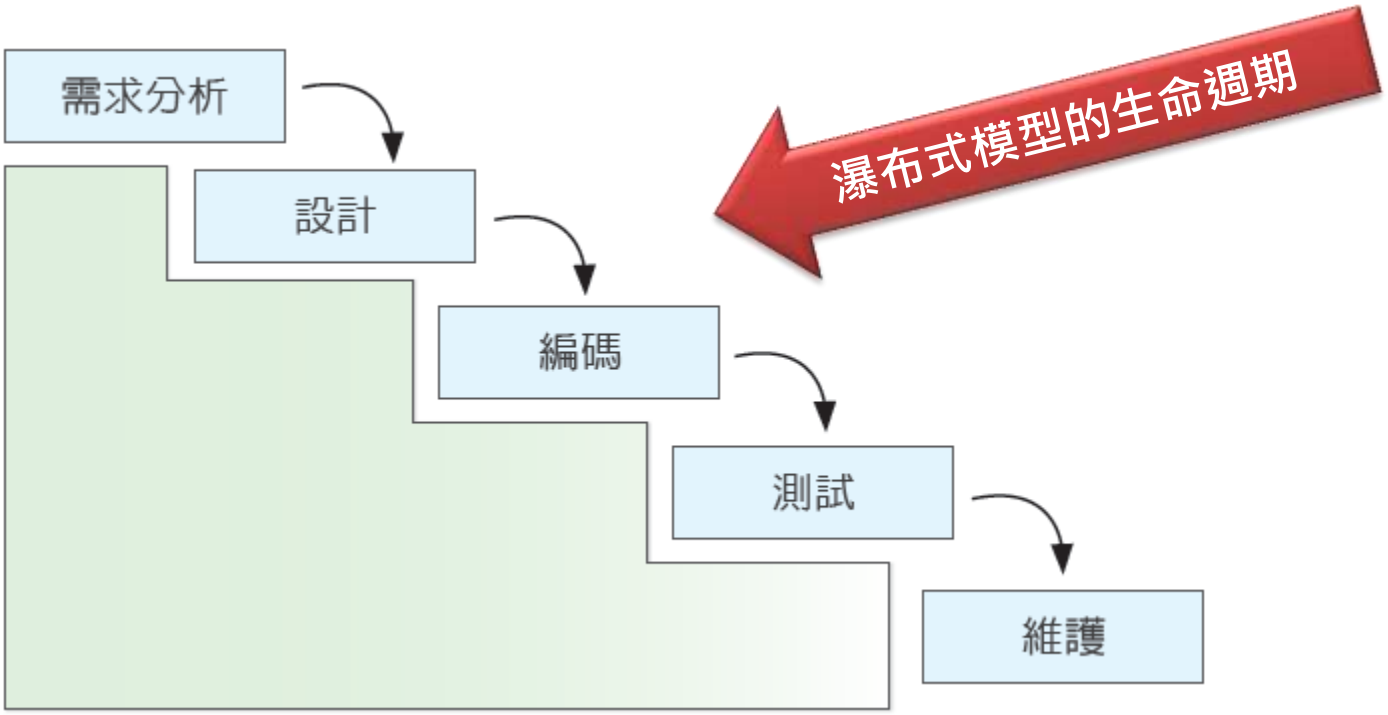
瀑布式模型

- 瀑布式模型是指軟體開發流程從需求分析、設計、編碼、測試、維護採用線性進行。
- 先整體分析客戶需求，接著進行設計，擬了架構之後進行程式碼開發，開發完成之後進行測試，最後上線並維護系統。
- 這樣的方法主要問題在於發掘可能性風險的時間點太晚。

瀑布式模型

- 直到產品測試完畢之後才與客戶進行展示，這樣做是非常冒險的。很多時候客戶能描述出他的需求，可是卻無法形容能夠滿足需求的成品樣貌為何。
- 直到最後看到成品時，第一種情況是感覺雖然所需求的功能都達到了，但是這並不是他真正所想要的；第二種情況則是客戶與專案負責人對於需求的溝通上一開始就發生誤解，實作出來的成品不符合客戶的要求，因此只好重新再來一次。

瀑布式模型



瀑布式模型

- 根據研究數據顯示，越晚發現缺失就得花上越多的成本來彌補，因此最後可能會因為成本太高而導致放棄專案進行。
- 不過這樣的發展模型可以適用於商用軟體的研發，因為線性流程無須往返，所費時間成本較低，並且公司內部本身對產品的樣貌無誤解。

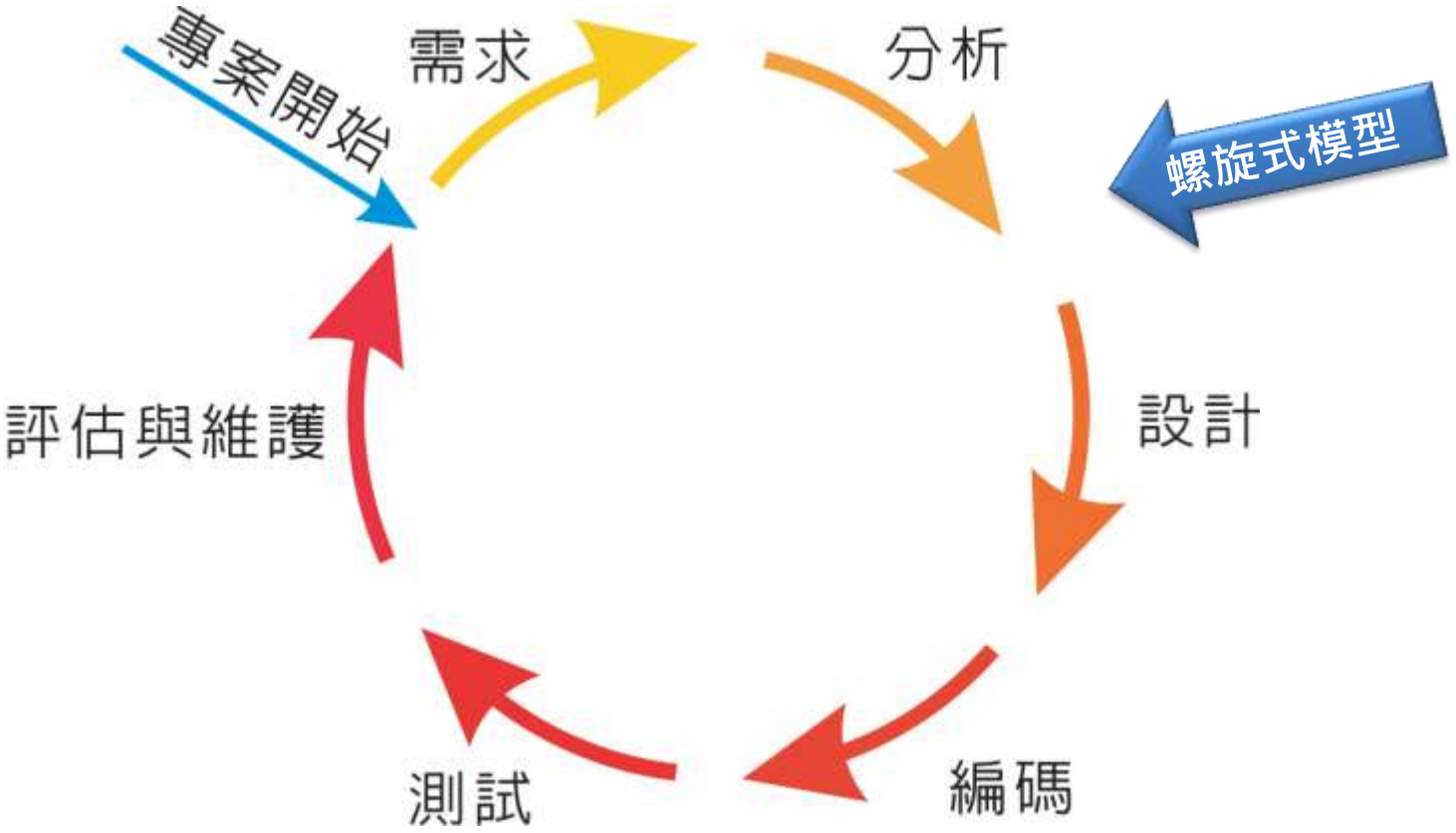
螺旋式模型

- 螺旋式的模型則是不停地反覆與漸增，螺旋式模型的「螺旋」是帶有慢慢擴張成長的意思。
- 它的關鍵在於先進行一點點分析，然後設計一點點，接著實作一點點，測試完成之後就進行評估，與客戶共同討論是否偏離了客戶的需求。
- 在整個發展的最初期就企圖盡可能地找到問題所在，以降低開發的風險。

螺旋式模型

- 完成最小規模的發展，並且通過測試檢驗強化軟體的可靠度，如果評估過後確認無誤，則再把規模擴大一些，進一步發展下去。
- 螺旋式模型的精神在於站穩一小步之後，再基於之前步伐的經驗邁出紮實的下一步。
- 每一次的新流程都會以過往流程的經驗為參考而進一步開發實作。

螺旋式模型



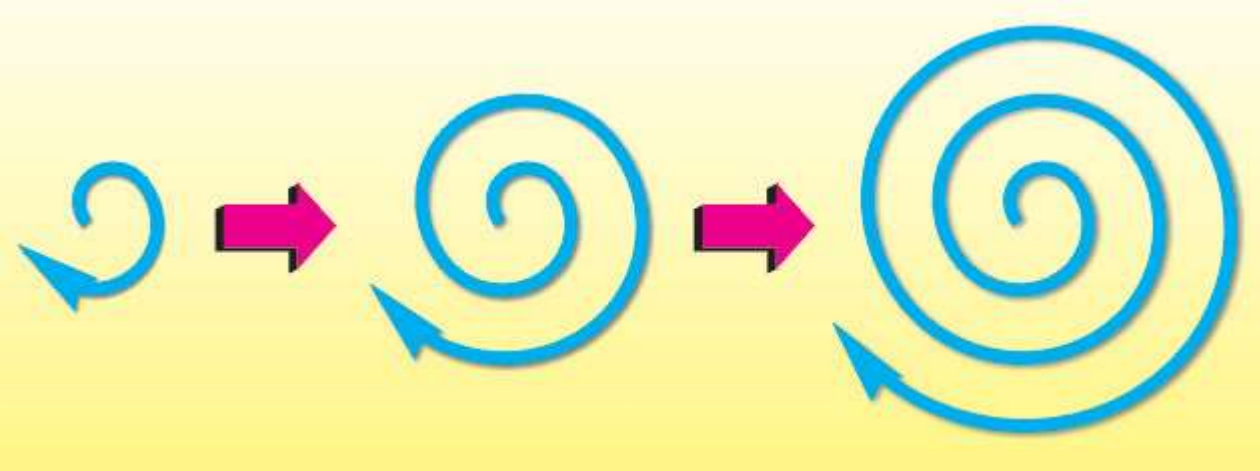
螺旋式模型

- 透過不停地往覆螺旋式前進的過程，每一步都可以充分掌握住使用者的需求，強化與使用者的互動，並且即時給予回應，就算是不得已需要更改需求，也能夠在早期就著手進行替換，使殺傷力降到最低。
- 這樣的開發方式雖然較為繁複，但是多付出一些成本總是比衝到最後一刻才赫然發現完全跑錯方向值得。

螺旋式模型

- 螺旋式的開發方式也可以用在開發不熟悉的領域。譬如假設我們對於網路程式開發不熟悉，可是卻要寫一個即時傳訊軟體，我們最好就採用螺旋式模型，一開始先理解如何透過socket傳遞訊息，如何透過socket接收訊息。
- 等到我們能夠掌握如何一對一往來訊息之後，再進而跨出下多交談的版本。

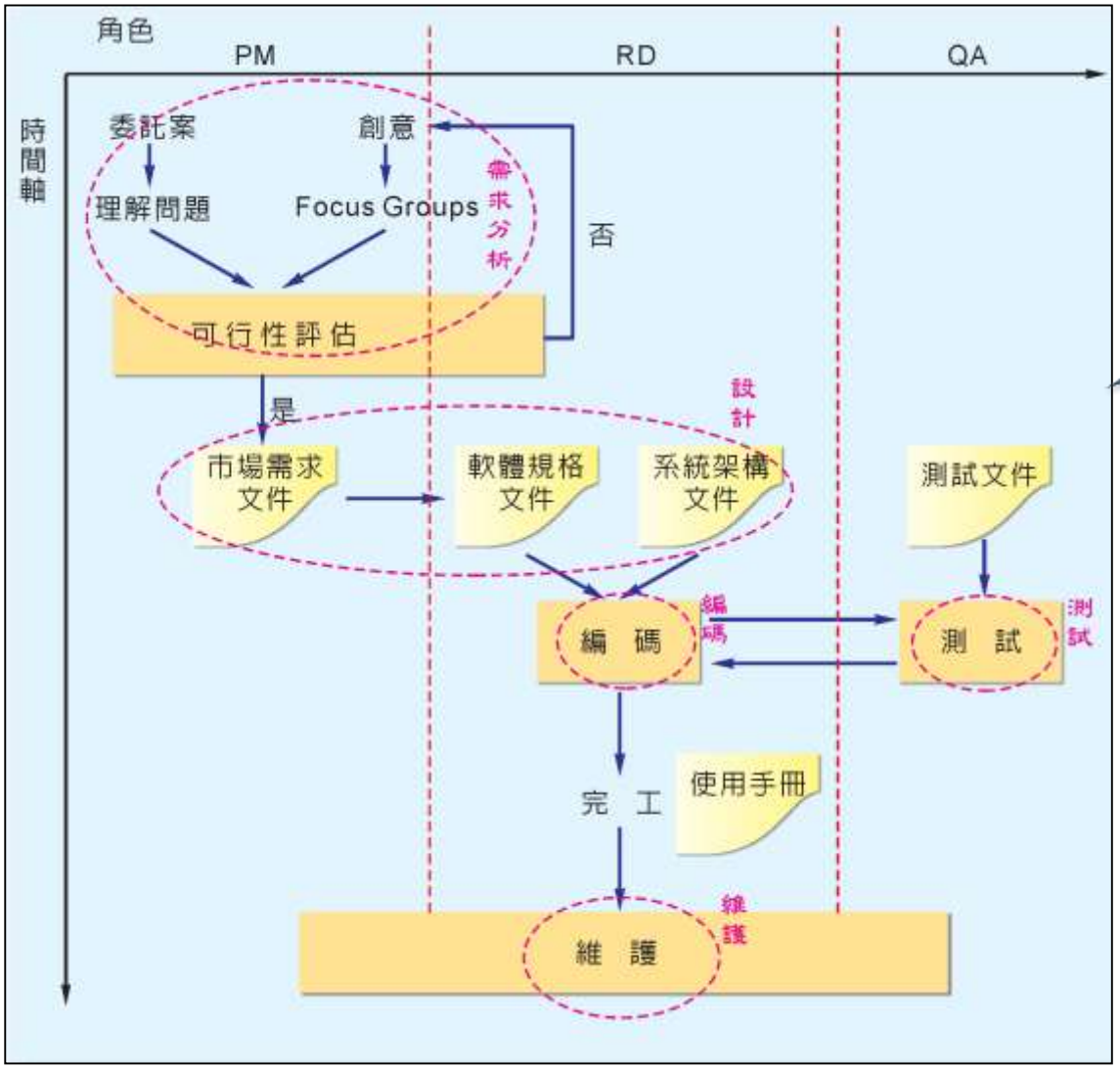
螺旋式模型



採用螺旋式模型發展，規模從小型擴展到大型

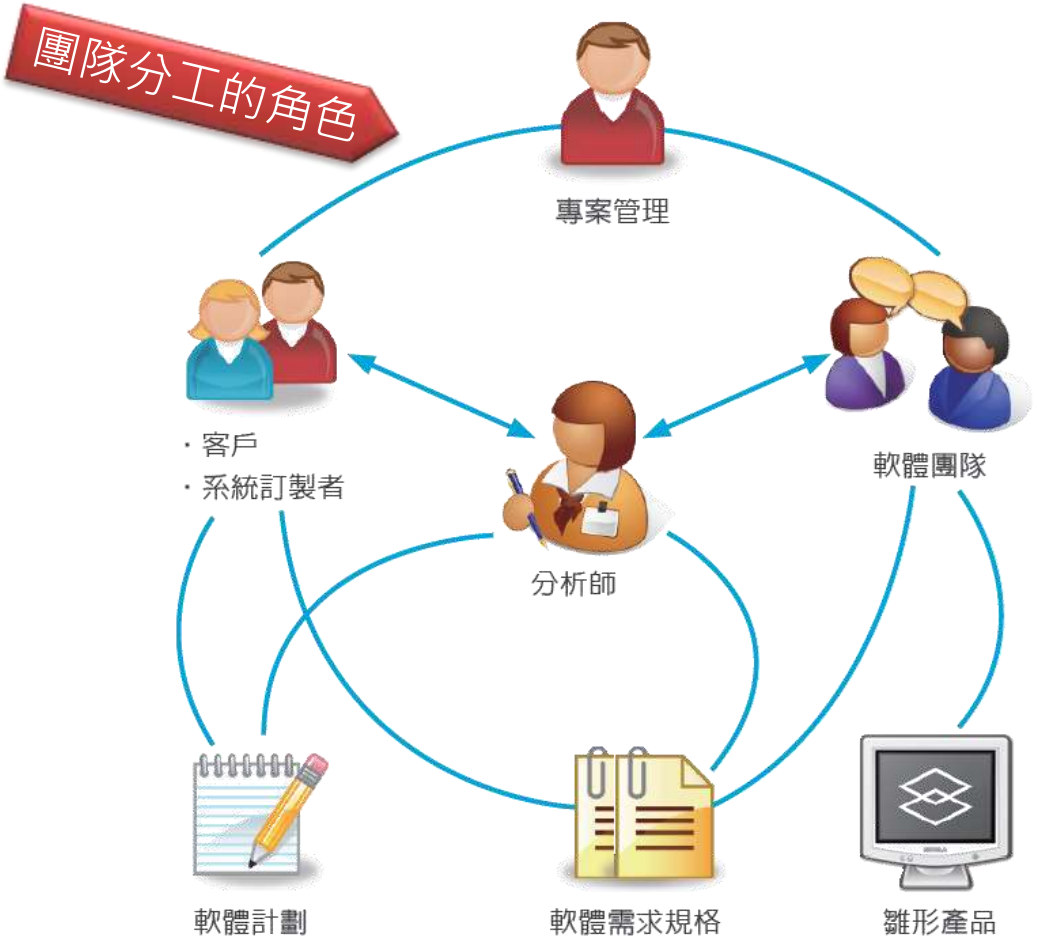
螺旋式模型

- 筆者以曾經參與過的e-learning輔助系統為背景，用二維的角度來討論，討論隨著時間的推進所該進行的事情，也討論什麼樣角色的人物負責什麼樣的任務。



軟體開發的過程及角色分工

螺旋式模型



假設一個開發團隊共有五人，其中分別是：
一位專案管理人(Project Manager；PM)
三位程式開發人員(Research and Devel opment ；RD)
一位測試工程師(Quality Assurance；QA)

需求分析

- 需求分析的工作是用來發掘問題的所在以及評估解決方案的程序。
- 在這個階段，專案管理人扮演極重要的角色。情況可能有兩種類型：

商業軟體

- 商用軟體情況下，可能是從一個有創意的點子開始，看到了一個可發展的產品。
- 為了確認這樣的點子能受歡迎而不是閉門造車，並且也為了了解使用者的想法，於是公司召集一群一般使用者進行Focus Group。
- Focus Group中，PM針對該軟體的特性準備一些問題，藉由受試者的回答整理出該軟體應有的功能。

商業軟體

- 以e-learning軟體為例，可能詢問的問題包括：

請問你過往有使用過線上數位學習嗎？如果有，能談談線上數位學習的好處嗎？如果沒有，請你陳述一下沒有使用過的理由。

請問你個人認為線上數位學習的效果如何？是否能夠保持專心？

請問你認為線上數位學習與傳統在課堂內教學的最大差異在哪邊？

商業軟體

- 透過一連串的問題，可以理解一般使用者的心聲、行為習慣，以及軟體的需求，也幫助設計產品功能特色時的取捨。
- Focus Group中的重點除了設計出好的問題之外，主持人應該謹記是導引受試者說出自己的心聲，而不是持有立場的導引受試者發聲。

軟體代工

- 在軟體代工的情況，軟體的發展是客戶有需求，所以找尋軟體代工幫助。
- 再以e-learning為例，可能是學校有計畫想要導入線上數位學習，或者是電子書包的應用，因此與軟體代工業者接洽。
- PM必須負責與客戶代表溝通，理解客戶的需求，排除造成矛盾的設計，並且充分與客戶解釋溝通。

軟體代工

- 在PM理解需求與規劃設計之後，此時RD就一起參與進行可行性評估，可行性評估包括技術上是否可行、成本是否能負荷、是否符合市場需求等等議題。
- 如果結論為可行，則PM開始撰寫市場需求文件(Market Requirement Document；MRD)，否則退回原點重新思量。

軟體代工

- PM必須要有好的整合歸納分析能力，在MRD中要條列出所開發的產品特性、開發此產品的想法及目標、使用者使用該產品的場景腳本(user story)、瞄準的顧客群為何、開發的需求、市場競爭力等等。
- 有了這份文件，可以讓上層決定是否要撥出時間人力等資源投入開發。

設計

- 有了MRD之後，RD群開始撰寫**軟體規格文件**(functional specification)。
- 根據MRD中所闡述的產品特性，RD具體地寫下整個系統如何運作，包含**使用者介面**(user interface)的設計、有哪些按鈕、功能列表中有哪些選項、如何操作才能執行功能。

設計

- 軟體功能文件所表述的是邏輯上的架構，以e-learning為例，描述的可能是使用者登入系統的畫面，登入後有什麼功能可以選用，每個功能該按哪幾個按鈕、輸入什麼，才可以達到功能。
- 除了描述邏輯上操作的軟體規格文件之外，RD們還得討論出整個系統架構， 撰寫出**系統架構文件**(system architecture specification)，在e-learning的例子中，必須決定系統的運作架構。

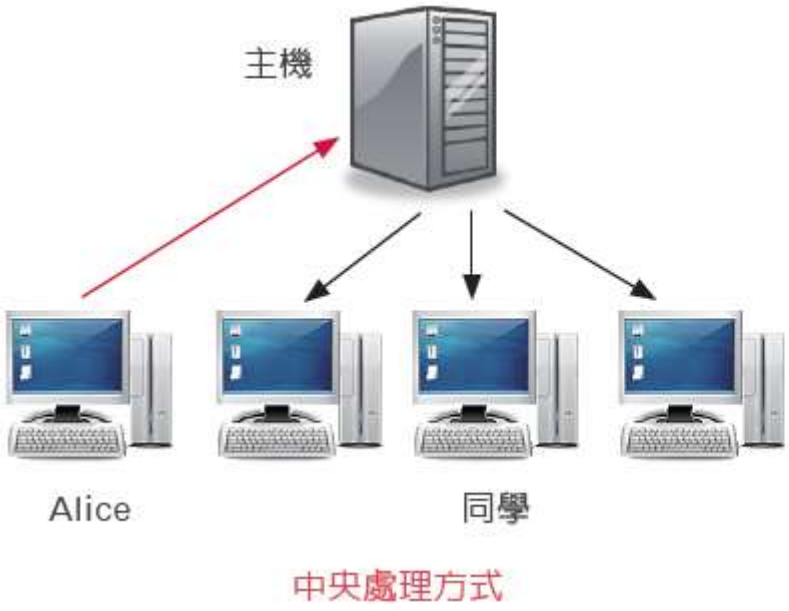
設計

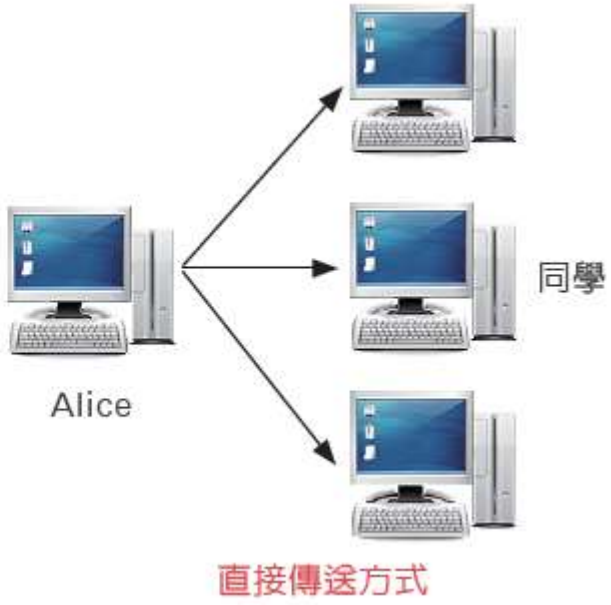
- 譬如說假設要做到同學們能夠相互交談，則我們至少就可以想到兩種系統設計方式，一種是採用中央處理方式，也就是有一個伺服器來處理所有的資訊，Alice要丟訊息給全班時，是Alice將訊息傳給主機，然後主機掌握住班上同學的所在網路位址，再廣播給班上的同學。
- 第二種方式是直接傳送方式，也就是Alice必須要知道班上所有同學的網路位址，然後傳遞訊息的時候，就直接由Alice發送訊息給大家。

設計

- 不同的系統架構就有不同的實作方式，RD工程師們必須要能夠分析不同架構的開發成本、運作效能等因素，最後決定出最好的版本，定案寫在系統架構文件中供開發時期的指南。
- 在此同時，QA也要開始根據所撰寫出的軟體規格文件而寫出測試文件(test plan)，內文寫出測試的模組、測試的項目、如何測試等計劃。

設計





針對同一功能，兩種可能的系統架構

編碼

- 有了軟體規格文件及系統架構文件之後，RD們就開始根據這兩份文件設計資料結構、程式架構，進而開始進行編碼的工作。
- 編碼設計的方式有兩種：

由上而下(Top-Down Approach)

由下而上(Bottom-Up Approach)

由上而下(Top-Down Approach)

- Top-Down方式有點像是Divide and Conquer的想法，把整個系統想清楚之後，切割成一個個的子系統，然後子系統完工之後再拼湊回來。
- 以e-learning為例，由上而下是先定好這個系統要有什麼功能，譬如分割成「上線名單顯示」、「共用白板討論」、「問與答機制」、「共同討論」。然後RD工程師先架設骨幹，之後分頭開始完成每一塊的功能，最後再把這些「肉」整合回「骨」之上，就算完工。

由上而下(Top-Down Approach)

- 這樣的進行方式比較適合整個系統架構已經明確了解，不會再有大變動時，較易成功。
- 用在寫小程式時，就像是有些人喜歡先把所有的架構給定好，宣告設計完所有的資料結構，然後也把程式模組化完畢，把所有的函式都宣告好。
- 等到這些骨幹架構都定了之後，再開始完成函式的實作、邏輯流程的設計。

由下而上(Bottom-Up Approach)

- Bottom-Up方式則是先針對每個需求各自開發，開發完畢之後，再把所有的元件整合起來，在開發的時候，無法看見最後的架構。
- 這有點像是我們小時候玩樂高蓋一個家園，我們可能想到什麼就先弄什麼，可能是先把房子周圍的籬笆蓋上，然後開始設計房子的長相，安插了房門窗戶之後，再設計庭院上放的信箱，最後再來設計花園。

由下而上(Bottom-Up Approach)

- 在蓋的時候，我們可能沒有完整地想過要蓋成什麼樣子，只知道要蓋房子，其他一切隨著興之所至而完成。
- 也有點像是蓋金字塔的時候，我們不可能一開始就拉起骨幹然後再把石頭填上去，而是採用慢慢把石頭從地面堆砌起的方式向上完成(bottomup)。

由下而上(Bottom-Up Approach)

- 這樣的方式適合應用在當系統的完成風貌還不夠清楚時；或者系統的規格隨著發展需求可能不停改變時，由於我們不能在一開始就清楚架設整個骨幹。
- 因此採用bottom-up的方式先針對已知的需求設計好，最後再把所有的元件兜起來。

由下而上(Bottom-Up Approach)

- 用在寫小程式的例子中，就像寫程式是一開始就直接下筆，需要用到什麼資料結構再回去宣告，每寫完一個小函式就先測試一下，邊設計邏輯的流程邊撰寫需要呼叫的函式，最後走完整個流程，程式大概也就完工了。

測試

- 測試是軟體發展中做到軟體品質保證(quality assurance)最重要的一環。雖然程式開發人員在撰寫程式的時候，會進行除錯好讓程式能夠運作。但是，RD在測試的時候幾乎都是一維的思考，譬如要開啟一個檔案，就先選功能列表，然後按開啟。
- 可是軟體交到使用者手上，使用者的行為模式很難預測，不小心亂按幾個按鈕之後才開啟檔案，是否能夠順利開檔就考驗著RD寫出來的程式是否夠可靠；或者當程式使用很久之後是否還能順利運作而不會常常當掉。

測試

- 軟體品質的保證要考慮到正確性、操作複雜度、執行效率、錯誤容忍度、壓力測試(針對系統測試其負荷量，譬如可承受多少人同時上線使用)、安全性等等議題。
- 這些就得交由QA如糾察隊一般使用RD交予的程式，根據測試文件進行測試。

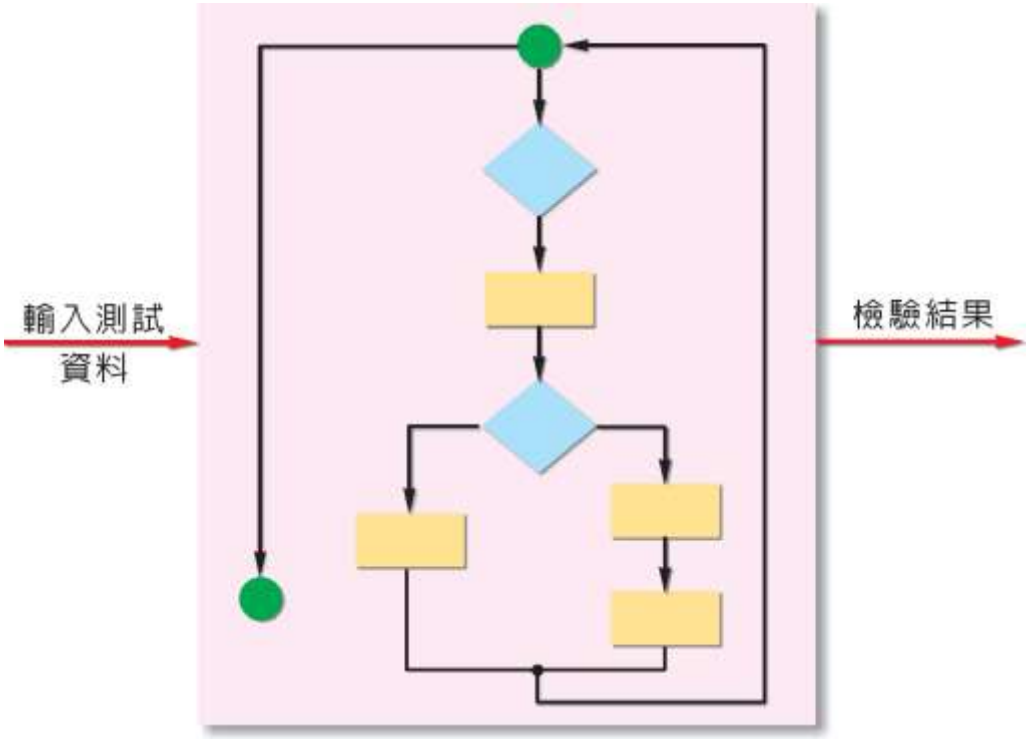
測試

- 越成熟的軟體公司，QA與RD的比例會越趨近於1：1，像是微軟就是這樣的例子；然而，發展中的公司，由於必須投入大量的人力資源開發，因此QA的數量就會比RD還少。
- 測試也可分成兩種方式：

白箱測試

- 白箱測試名為「白箱」是指程式碼就像是個箱子，在白箱測試的時候，QA工程師已經知道整個箱子的內容，也就是清楚程式的流程，根據程式的流程，每一條路徑都走過一遭，確定功能是否能正常運作。

白箱測試

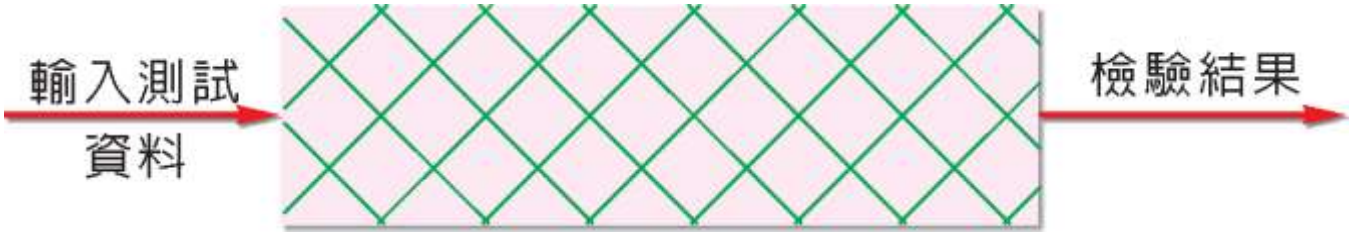


白箱測試中，QA根據程式的流程進行測試

黑箱測試

- 黑箱測試是指QA工程師不理會程式的邏輯流程，把程式碼當成一個黑箱子，以功能為依歸，猜測程式碼可能會在哪個環節發生錯誤，而抓出漏洞所在。
- 黑箱測試就比較像是一般使用者的行為，使用者不知道RD怎麼設計這個程式，只知道有哪些功能可以用，就依照自己的想法使用，可能會成功，也可能會發現程式的漏洞。

黑箱測試



黑箱測試中，QA把程式當作黑箱，不管內部如何運作，只管是否能正常運作

黑箱測試

- 這兩種測試方式由於使用的方法跟精神不同，因此可以抓出不同的錯誤，所以是互補的方法。
- 測試中如果發現仍有問題，必須將問題記錄下來，呈報給RD，告訴RD哪一個程式碼版本在什麼樣的情況之下會發生怎樣的錯誤。

黑箱測試

- 為了使RD與QA都不做白工，已發現的錯誤必須進行錯誤管理。
- 譬如說是使用Bugzilla軟體來進行錯誤的發佈、指出錯誤的嚴重性、更正、審核，同時也可以讓PM理解目前專案發展的進度。

Bugzilla，一個針對錯誤控管的軟體

維護

- 程式碼在RD與QA中來回翻修檢測校正之後，等到產品穩定下來，就可以交貨或壓片，然而軟體流程並不止於此。
- 除了開發出可用的程式之外，還應該撰寫出使用者手冊、管理者手冊等文件才算完整。並且在推出軟體之後還要進行維護。或是根據使用者的反應作為下一版軟體的變更目標，評估軟體的設計方式，或是幫助採買軟體的客戶維護系統運作。

維護

- 由於程式開發時很可能是人來人往，因此文件的撰寫與保持，對於軟體維護的重要性自是不在話下。
- 理解了以上所闡述的各個生命週期內涵，我們可以釐清幾個普遍容易對軟體發生的誤解。

維護

情況 1

- 軟體跟硬體除了一個看不到，一個看得到之外，有什麼大差別嗎？

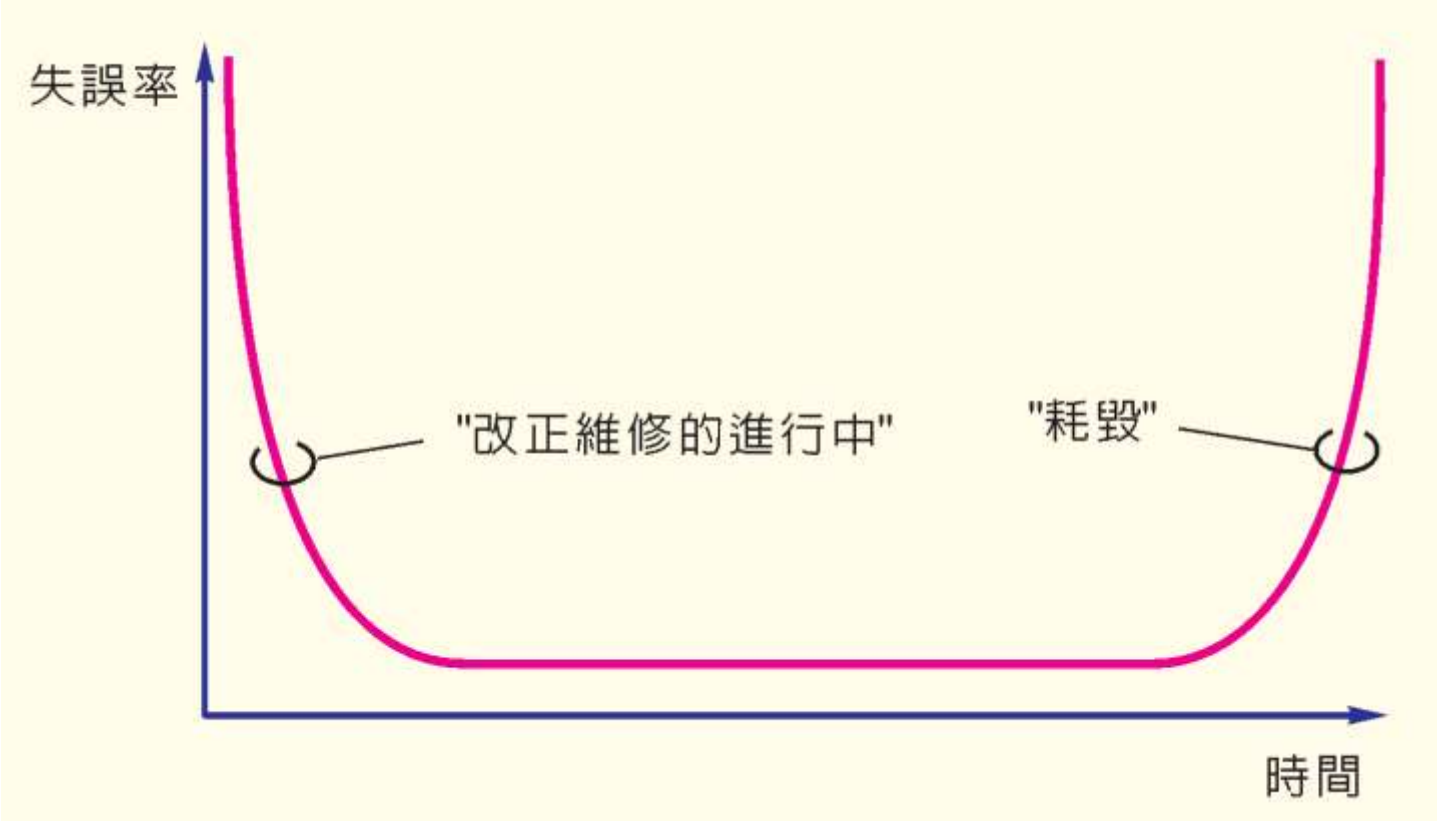
說明：

- 軟體跟硬體除了成品是否有實體之外，本質上與生命週期都有很大的差別。硬體生產的關卡在生產線上，廠房大小，能夠負荷多少生產線都是關鍵因素；而軟體的生產關鍵則在於開發設計時期，整個軟體的成本也幾乎都耗在開發上面，一旦完成之後，就只剩下壓片製成光碟，其成本比起硬體生產可說是極為低廉。

維護

- 此外，硬體的生命週期也與軟體大不相同。
- 硬體的品質效能失誤率呈現一凹曲線，稱之為「浴盆形曲線(bathtub curve)」，在初期由於設計及製造上的缺點導致失誤率較高，經由不斷的改良之後，能夠達到穩定的水平狀態而量產上市，然而經過使用者長期使用之後，因為硬體元件耗損折舊，失誤率再度上升，直到完全損壞。

維護



硬體的品質效能失誤率曲線

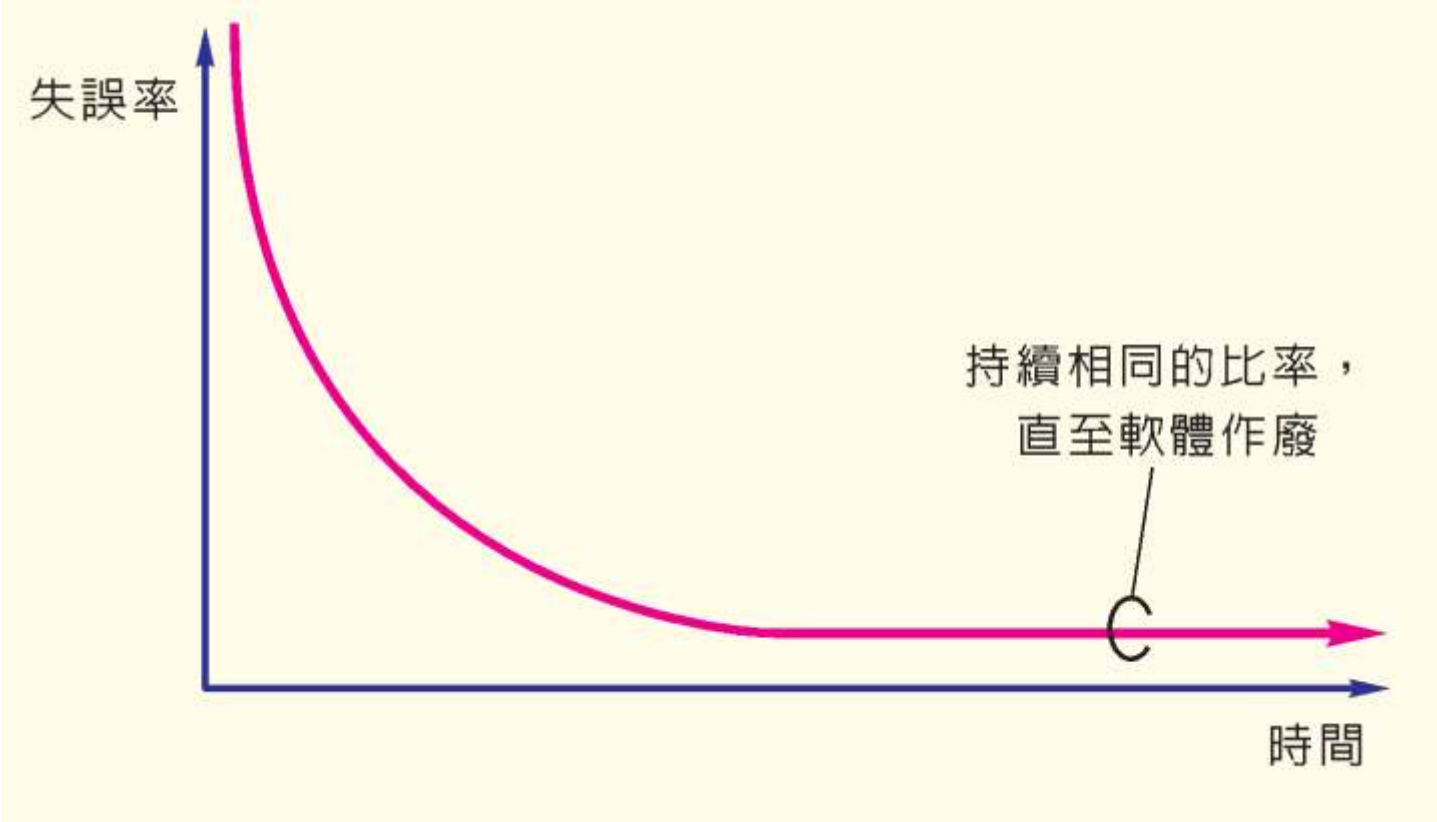
維護

- 然而軟體卻不會折舊，因此軟體的品質效能失誤率曲線可說是大不相同，程式開發早期時，由於程式中有許多設計不夠完善的地方未被一一揪出，因此很容易使用一下就當掉，隨著不斷的修改程式而使得效能逐趨穩定，最後就壓片販售，同一版本的效能將維持相同的水準，直到軟體作廢為止。不過，上面描述的是理想中的情況。
- 理想與現實總是有一大段差距，現實情況中，軟體會不停地被修改，每一次修改雖可以解決之前版本的一些錯誤，可是也可能因為程式碼的更動而引發更多的錯誤，再經過修改之後才能使效能回穩。

維護

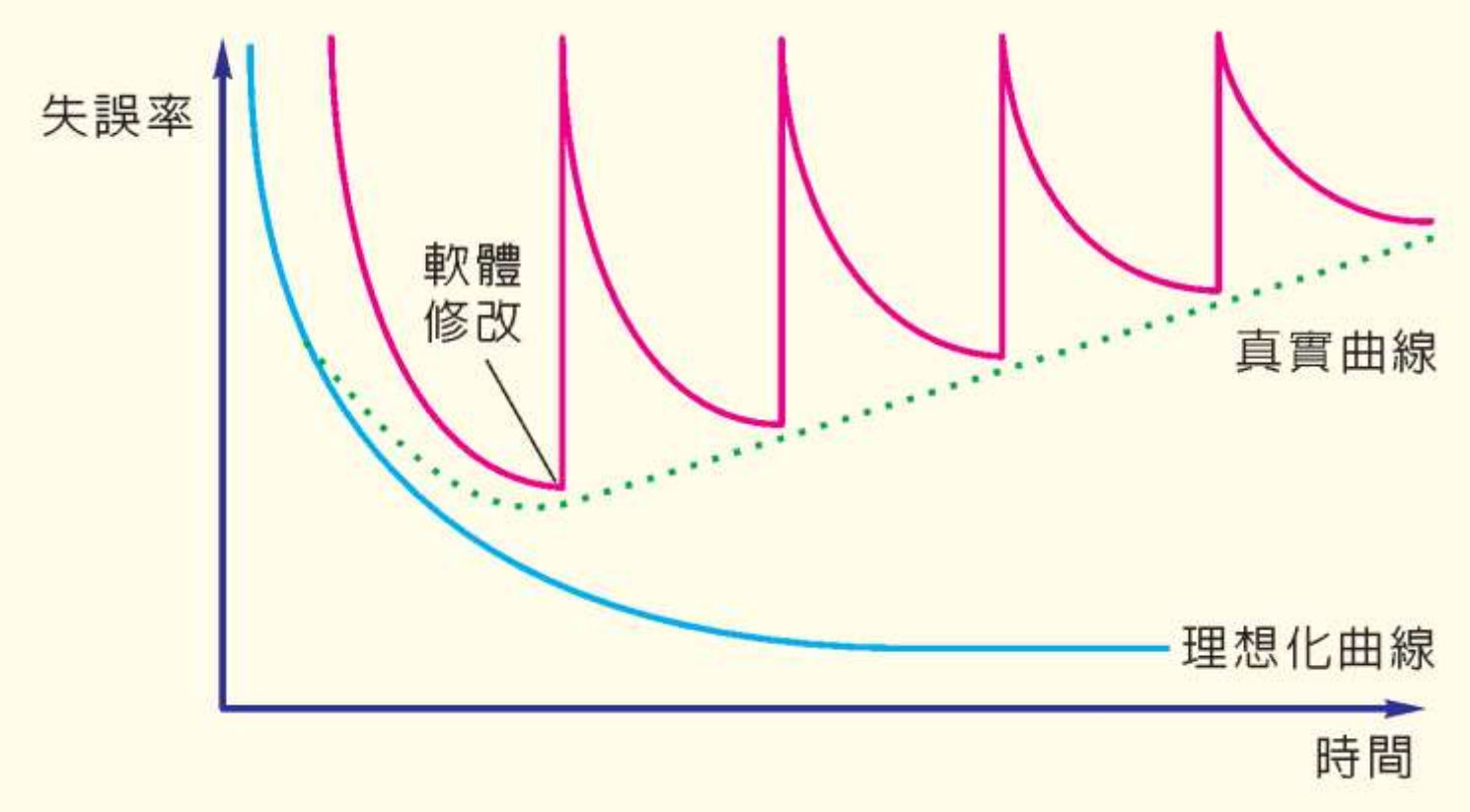
- 軟體會不停地修改，直到最後整個程式碼可能因為經過多人多年的變動，早已失去原本架構設計的模樣，雜亂無章至難以理解的境界，最後變成開發新的版本比維護改進舊有版本的成本更低，於是舊版本就被放棄了。

維護



理想中的軟體品質效能失誤率曲線

維護



現實情況的軟體效能失誤率曲線

維護

情況 2

- 假如軟體開發時進度落後了，再多加幾個人手不就可以了事嗎？

說明

- 軟體的開發與硬體開發大不相同。硬體開發時，如果工廠突然有大量訂單需要交貨，多開啟幾個生產線運作，產能就能明顯成長。

維護

- 軟體的開發並不像是製造業那樣機械性的程序，甚至有此一說：「對於進度落後的軟體專案，如果再加人手進去參與，只會讓進度更落後。」
- 乍聽之下似乎十分矛盾，其實卻十分合情合理，因為新人加入團隊之後，必須要花時間熟悉整個專案的架構內容、理解目前的情況，於是原本的開發團隊得停滯下來抽出時間與新人重新融合，所以整個進度反而變慢了。

維護

情況 3

- 軟體開發的方式為什麼需要更改或變動呢？難道程式設計會不同嗎？

說明

- 雖然都是進行軟體開發，可是隨著需求、品質及策略之不同，都應有不同的開發方式。商業軟體競爭是非常激烈的，並且軟體市場的特點是贏者壟斷(Winner Takes All)，也就是說通常第一名的佔有60%左右的市場，第二、三名共佔有30%，剩下的10%分給其他知名度低的廠商。

維護

- 因此，如何搶得先機非常重要，譬如微軟這幾年來推行Tablet PC，針對Tablet PC發展專屬的作業系統，雖然系統還不算是非常的完善、系統的效能也不算極佳，然而為了搶奪先機，就不能追求完美，得積極地先打出頭陣佔有市場，其他不夠完善的地方再推出二版改善。
- 針對桌上型電腦的作業系統，微軟就持保守的態度，最新一代的作業系統每每預計某年就要發表，卻因為種種因素而不斷延遲，微軟在這方面的策略可以較保守的理由自是因為已經擁有絕對優勢的市場佔有率。

維護

情況 4

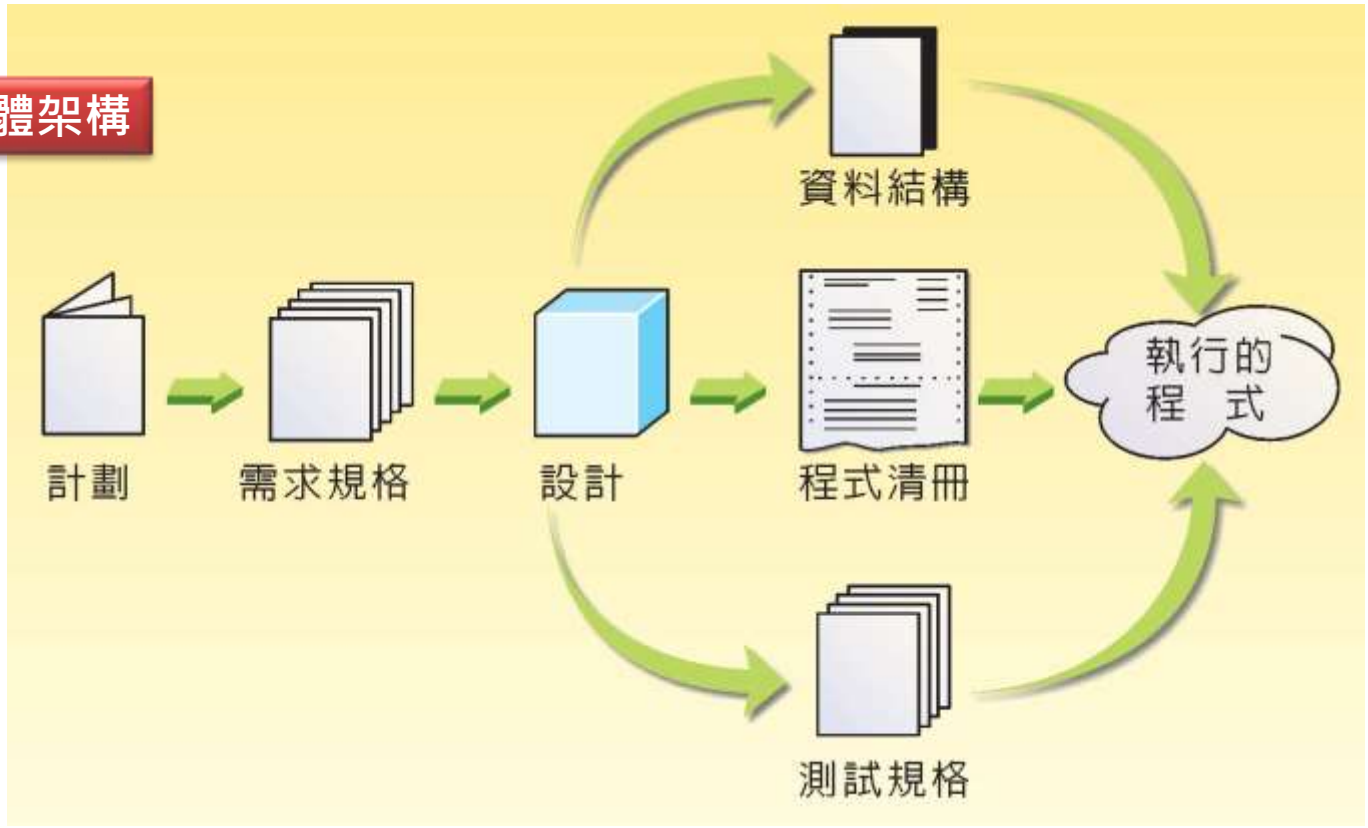
- 只要能夠展示出一套能運作的軟體，即能稱該專案成功，從此買賣雙方貨銀兩訖。

說明

- 一個完整而成功的軟體專案，包含著許多企劃、規格、程式碼、測試規格、說明文件、使用者手冊等等，能夠運作的軟體只是其中的一環而已。

維護

軟體架構



12-3 軟體品質認證

- 軟體產業蓬勃發展之後，軟體工作室便如雨後春筍般成立。組成的成員可能是一個功力深厚的工程師帶領剛畢業的學弟妹們；也可能是一群平凡而踏實的工程師；當然也可能是一整群天才型的程式開發人員。
- 寫程式的功力固然重要，然而軟體開發絕對不是把程式開發完就了結，開發出來的產品品質等問題都是很重要的議題。我們首先得確認一下自己對軟體的認知是否正確。

12-3 軟體品質認證

- 50年代時，人們認為軟體就是程式，直到70年代的時候，軟體的定義逐漸被改觀，人們開始認為軟體是由程式及開發它、使用它、維護它所需的一切文件共同組成。
- 並且在軟體專案中，軟體的開發固然非常重要，然而實際上線之後能否順利運作，大部分還是取決於維護的好壞。這是很多採買軟體方所忽略的重要環節。

12-3 軟體品質認證

- 筆者曾經參與**數位學習**(e-learning)輔助工具的發展，在與期望採用該系統的校方接洽中發現維護的重要性被嚴重忽略了。
- 買方看到軟體已能展示，能夠順利運作，就認為大功告成了。實際上，能夠展示(demo)與能夠使用可能相差十萬八千里。

12-3 軟體品質認證

- 展示的時候，由於是程式開發人員進行展示，因此程式開發人員自己可以避免讓程式當掉，程式開發人員會盡可能小心地讓展示順利；然而真正上線後，使用者的行為是無法預期的，使用者很可能會亂按按鍵、使用者很可能不按牌理出牌，這是第一個要注意的地方。
- 就算程式已開發成熟，還是不能採用貨銀兩訖的方式，因為系統的運作是需要維護的，不管是效能維護，管理維護或者是安全維護。

12-3 軟體品質認證

- 因此要委外開發的軟體，就要做好心理準備，除了開發的成本之外，仍要付出維護的成本。
- 如果只願意付出開發的成本而不願支付維護成本，很可能發生整個系統無法運作，最後連所付出的開發成本都浪費掉了。

12-3 軟體品質認證

- 這樣的觀點傳達出人們開始重視文件在軟體研發使用中的重要性，並且認為文件本身是軟體的組成份子之一。1983年時，IEEE明確地給軟體下了定義：

> 軟體是計算機程式、程序、規則，以及任何相關的文件以及在執行上所必須要用到的資料。
> "Software includes computer programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system" IEEE Standard Dictionary of Electrical and Electronics Terms, Third Edition

12-3 軟體品質認證

- 完善的軟體開發必須考量到軟體程序、開發時間控管、文件與程式都符合標準等等。
- 80年代中期，美國聯邦政府提出對軟體承包商的軟體發展能力進行評估的要求，委託卡內基美濃大學(Carnegie-Mellon University) 軟體工程研究所(Software Engineering Institute；SEI)進行研究，在Mitre公司的協助之下，於1987年發佈了能力成熟度框架(capability maturity framework)，以及一套成熟度問卷 (maturity questionnaire)。

12-3 軟體品質認證

- 這兩份文件提供了軟體過程評估及軟體能力評估，在當時提供美國軍方採買軟體時評估提供商的軟體品質。
- 1991年，SEI進一步將能力成熟度框架發展為**軟體能力成熟度模型** (software capability maturity model)，並且發佈最早期的CMM 1.0版本。

12-3 軟體品質認證

- 陸續又推出軟體工程、系統工程、及軟體採購等模型，並且於2000年底發表CMMI(Capability Maturity Model Integration)，成了目前軟體品質認證的主流。
- CMMI的精神致力於軟體發展過程的管理及工程能力的提昇與評估，CMMI可說是針對軟體開發及服務所訂出的標準。

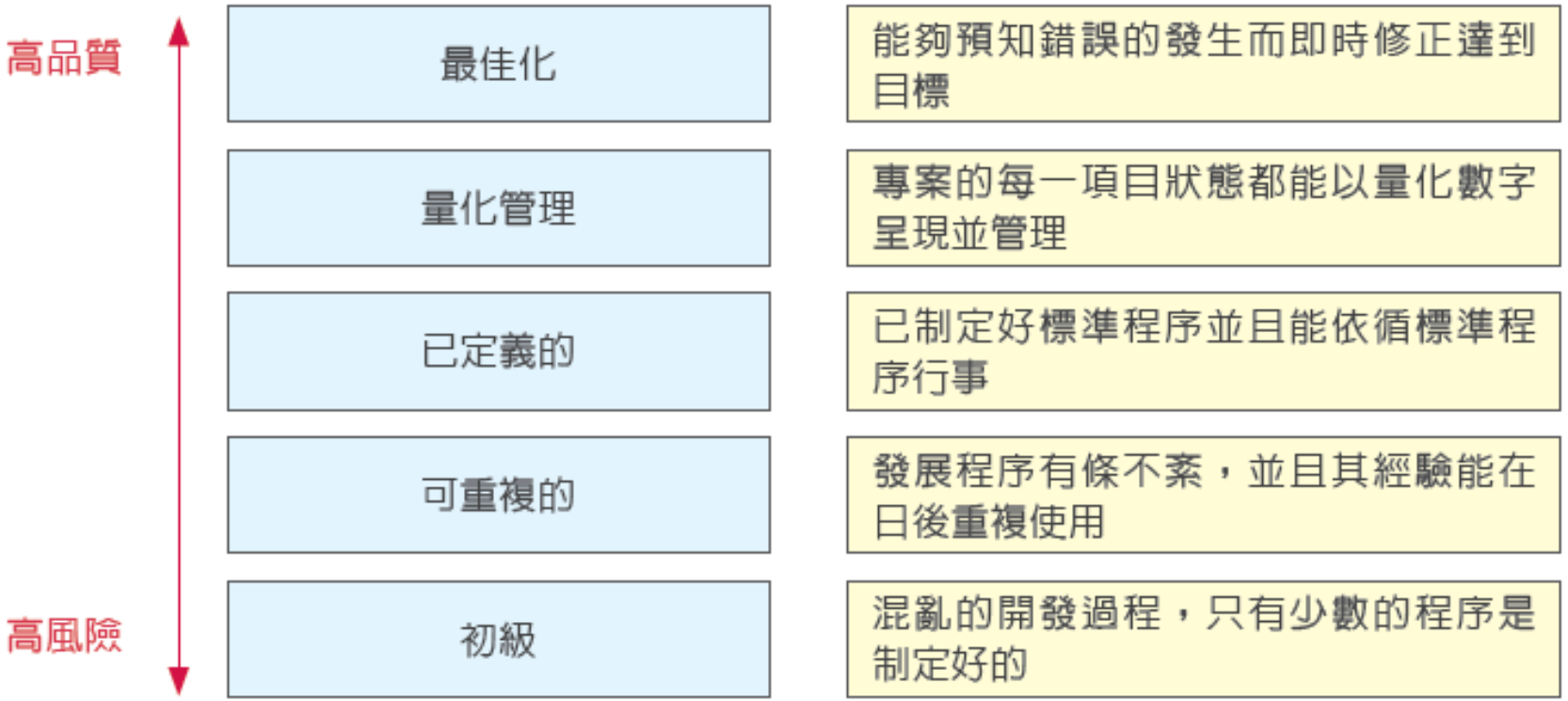
12-3 軟體品質認證

- CMMI強調軟體發展的成熟度，積極地闡明如何提高效能及改進過程；而ISO則是消極地描述可接受的最低標準。
- 軟體品質有了認證的依歸之後，對於軟體公司或是採買軟體方都是一項保障。
- CMMI分成五個等級，從第一級(level 1)到第五級。

12-3 軟體品質認證

- 基本上每間軟體公司一開始就是屬於第一級，一級只要提出申請即可列入，不需經過審查。
- 級數越高表示公司越成熟，到了第四級就可以達到量化的管理，而第五級則是可以預防缺失的發生。

12-3 軟體品質認證



CMMI的五個等級

12-3 軟體品質認證

第一級	• 初級（Initial）
第二級	• 可重複的（Repeatable）
第三級	• 已定義的（Defined）
第四級	• 量化管理（Quantitatively Managed）
第五級	• 最佳化（Optimizing）

第一級：初級(Initial)

- 軟體發展的過程是散亂的，有的時候甚至會陷入混亂(chaotic)的情況。
- 發展過程中只有少數的程序(process)是被定義好而執行的。
- 整個開發的成敗關鍵取決於少數精英的奮鬥及其突出的表現，並非仰賴團隊合作而成。
- 所有的開發經驗對於之後的開發並沒有留下可利用的價值。

第二級：可重複的(Repeatable)

- 已經擁有最基本的需求管理，撰寫出專案企劃書，能夠監控專案的發展，並且能夠針對成本、時間、功能進行量化的測量及分析。
- 針對發展的過程及出產的成品，能夠給予品質保證(Quality Assurance；QA)。能夠監控發展過程的需求。
- 所有的程序都能夠事先計劃，再執行，能夠量化測量執行情況，並且一切能在掌控之下。

第二級：可重複的(Repeatable)

- 在計劃中的進度檢視日期到來時，能夠看到工作的進度。
- 擁有簽章同意文件，相關主管在審視合格之後必須加以簽章負責。
- 產品能夠符合開發之初所訂下的規格標準及目標。
- 基本的開發原則能夠在日後遇到相似的專案時再度拿出來使用。

第三級：已定義的(Defined)

- 軟體發展中所有的程序，包含開發及管理等都已建檔文件化、標準化。
- 所有的程序都有其特殊意義並且確實被眾人了解。
- 能以團隊的方式進行開發，並且能夠進行開發風險管理。
- 組織內有訓練，並且能夠針對做出的決定進行分析及評估。

第三級：已定義的(Defined)

- 組織能夠發展出自己的一套標準程序，並且隨著時間的演進而修改。
- 所有項目的開發和維護都遵循標準程序而進行。
- 組織的標準程序能夠跨部門的一致執行。
- 所有的專案都必須經由簽章核准之後才能夠進行。

第四級：量化管理 (Quantitatively Managed)

- 能夠量化管理專案的進行。
- 能夠理解開發程序的好壞，量化測量產品的品質。

第五級：最佳化(Optimizing)

- 建立回饋機制，透過將過往開發經驗定量後，檢討並能衍生出創新的方法進行專案開發。
- 能夠明確指出開發過程的缺陷問題發生在哪兒。
- 定期檢討開發過程，並且根據成果檢討修改商業計劃。
- CMMI中，除了第一級之外，每一級都有該級的基本精神及訴求。

12-3 軟體品質認證

第二級是把重點放在建立基本的項目管理控制，這些項目包含需求管理、軟體專案企劃、軟體專案的跟蹤和監督、軟體轉包委外管理、軟體品質保證和軟體組態管理。

第三級訴求則是著眼於專案以及組織本身。整個組織必須能夠鋪陳好實行軟體工程的環境，使得軟體的開發及管理有一致性。

12-3 軟體品質認證

第四級 強調到達不論是軟體開發或者開發管理等各項目，都能量化來呈現出情況的境界。

第五級 是表達出要整個組織及企劃案都能不停地審視產生出創新想法，不斷檢討過往的缺失以避免未來失敗的發生。

12-3 軟體品質認證

- 推行CMMI軟體認證的好處在於提供軟體公司提高管理能力，提供自我評估的方式，提高生產力和品質，並且透過認證的保障，提升軟體公司的國際競爭力。
- 對於需要委外製作軟體的單位而言，透過CMMI則是可以理解軟體公司的開發能力及管理能力，甚至了解該公司是否能夠提供完善的上線維護服務，進而評估委外的風險。

12-3 軟體品質認證

- 台灣自認為是資訊王國，然而，精準地說，其實應該是資訊硬體王國，軟體的發展在台灣還不成氣候。很多公司積極導入CMMI認證，並且有些公司確實也因為導入CMMI後，得到跨國大廠牌的信任而多接到一些案子。
- 然而，由於CMMI認證標準之嚴苛，僅有少數幾家公司能夠得到CMMI第二級認證，第三級以上更是少之又少。目前世界上軟體代工最著名的國家是印度及中國。

12-3 軟體品質認證

- 在印度，幾乎每家軟體公司都有得到CMMI一級以上的認證，並且由於工資便宜，工程師訓練札實而嚴謹，並且語言溝通上零障礙而廣受歡迎。
- 然而，不同的國情、不同的公司針對軟體品質的保證，應該採用不同的作法。
- 針對軟體公司，我們可以粗淺地分成兩類型：

12-3 軟體品質認證

自己開發商品，將商品盒裝上架，公司自己負責行銷產品，譬如說是賣作業系統的微軟、賣防毒軟體的趨勢科技。這類型的公司就是將產品掛著自己的品牌名稱去行銷，因此CMMI的認證幫助並不大，銷路的好壞是取決於市場的反應，使用者只在意使用的軟體是否實用並且操作方便，開發過程中是否依循軟體工程的規劃對一般使用者而言並不重要。

另一類型的公司則是我們上面提到的軟體代工，這類型的公司若能取得CMMI的認證則能有顯著的加分效果。譬如說花旗銀行想要開發一套幫助客戶理財的軟體，由於花旗銀行本身並不是開發軟體的高手，所以可能委外開發，這個時候CMMI認證就能派上用場，有了認證就能在第一時間打敗沒有認證的公司，很快地就能夠得到大客戶的信任而接到專案。

12-3 軟體品質認證

- 軟體公司有兩大類型的訴求，行銷自己產品的公司追求的是創意，產品是否熱賣端看是否能夠切中使用者的需求，進而打敗目前市場上已有的軟體，能夠讓使用者心服口服放棄慣用的軟體，或者是安裝新的軟體使用前所未有的服務。
- 幫人代工的軟體公司著重於能夠與客戶妥善的溝通，完全理解客戶的需求，包括產品特性的需求及工作進度時辰上的需求，開發過程講究要能有條不紊，文件得完整無缺詳細說明，所有的事情都能井然有序地依照計畫時間完成上線，並且在上線之後提供諮詢及維護系統的服務。

12-3 軟體品質認證

- 對於後者這樣的情況，推動CMMI是合情合理的，因為兩者所強調的是不謀而合。而前者的狀況則應導入其他不同訴求的軟體認證標準，才能夠發揮最大好處。

12-3 軟體品質認證

- 軟體工程之難並不是在學習之上，而是在於實作時如何嚴守軟體工程的規範，如何細心地寫下一本本的文件，如何耐心地維護每一個版本的文件說明。

12-4 UML

- 軟體開發生命週期基本上包含了數個階段：需求分析、設計、編碼、測試、維護。其中最具代表性的開發流程，就是1970年代末期提出的瀑布式模型。
- 該模型提出的進行流程是線性的，也就是從需求分析開始依序進行每一階段；至於為了能夠順利的從前階段進入到後階段，需要大量的文件加以輔助和說明。

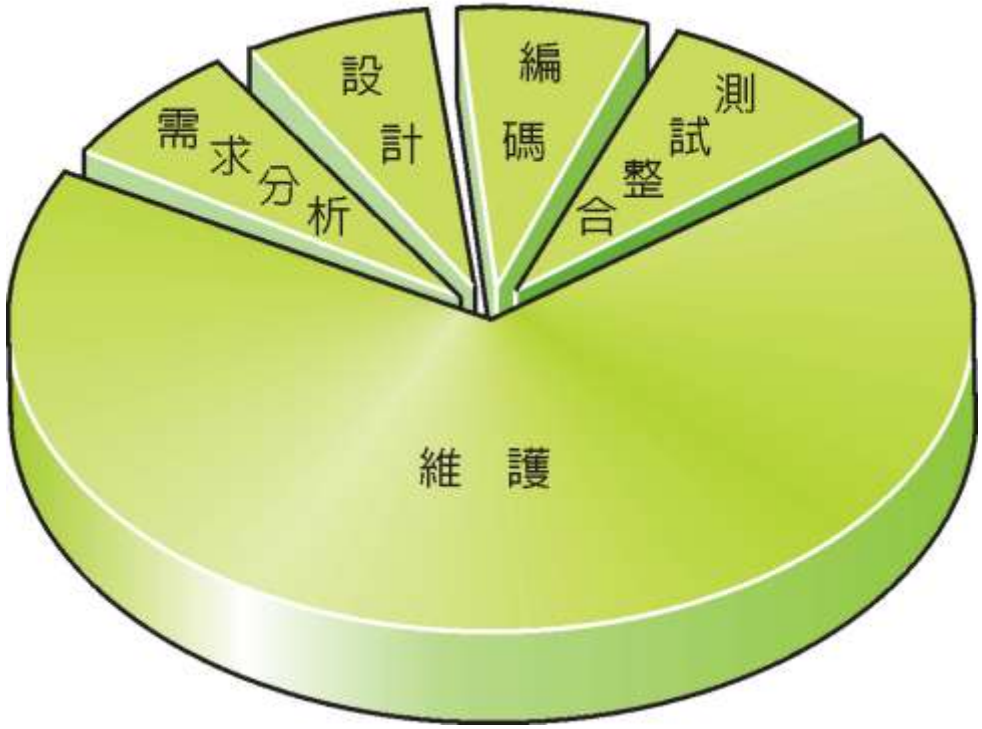
12-4 UML

- 瀑布式模型是傳統結構化(structured)設計的代表，它提供了詳細的準則，供程式設計師進行軟體的發展。
- 但是經過了數十年，結構化設計也開始顯露出它的不足。
- 其中最大的缺點，就是無法替軟體維護所需要的龐大花費帶來有效的解決方案。

12-4 UML

- 在整個軟體開發週期中，其實維護所需要的花費常常佔到整體的六、七成左右。
- 這是因為當一個軟體具有繼續使用的價值時，會不斷投入人力和經費去進行軟體的維護，或是修正之前沒發現的小錯誤，或是因應時空的改變導致對軟體功能需求的改變。
- 結果，維護的費用會比起一開始開發設計的費用更高出許多。而隨著軟體越來越複雜和龐大，這種情況也越來越嚴重。

12-4 UML



軟體生命週期中各階段花費的相對比例

12-4 UML

- 物件的觀念在1980年代末期開始受到重視，而比較成熟的物件導向模式則在1990年代被提出。
- 物件的好處，在於提供良好的模組化觀念，把相關資料的處理程序都定義在一起(也就是直接定義在物件上)。如此一來，若是日後要維護的時候，可以很輕易的找出要修改的地方，而不會淹沒在龐大的程式碼中。

12-4 UML

- 我們修改的地方，也會被侷限在局部的程式碼中，比較不會發生因為修改舊的錯誤，而產生新錯誤的問題。
- 在1990年代初期，比較受歡迎的物件導向設計方法，有OMT(Object Modeling Technique)和Booch’s technique這兩種。

12-4 UML

- 經過多年之後，這兩種方法終於被統合，連同一些新的改進方法，於1997年正式推出UML(Unified Modeling Language)1.0版。
- 自從UML推出之後，受到廣大的重視，很多軟體公司都開始採用UML來協助進行物件導向的設計，市面上也可以看到相當多專門的書籍。在此我們僅做簡單介紹，讓讀者有個初步的瞭解。

12-4 UML

- 所謂的物件導向(object-oriented)，就是以物件(object)為中心。
- 基本上，一個物件有下列特性：
  - 屬性(attribute)：定義物件的資料部分，或稱為物件的資料成員(data member)。
  - 方法(method)：定義物件的行為部分，或稱為物件的函數成員(function member)。
  - 封裝(encapsulation)：將物件行為和資料一起直接定義在物件上。

12-4 UML

- 所謂的「類別」，可看作是物件的集合，所以有時我們也稱物件為類別的「實例」(instance)。
- 譬如，我們可以定義「學生」為類別，而學號「B9201」的王雅蕙同學就是其中的一個物件，或稱實例。

12-4 UML

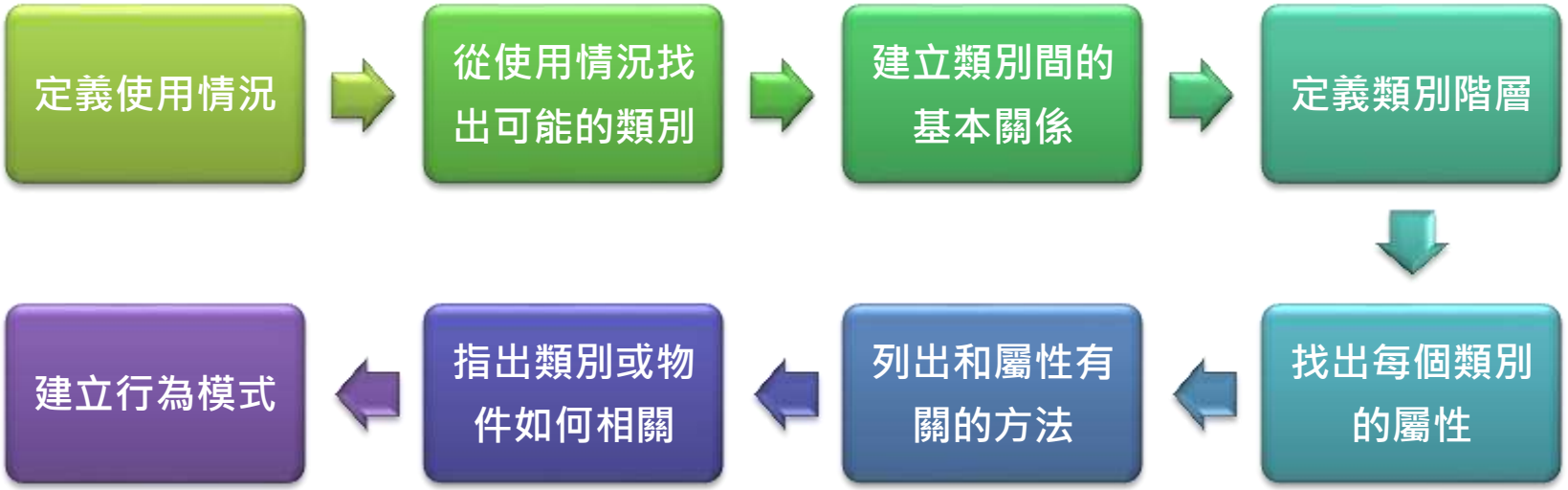
- 為一個應用程式分析資料時，往往會定義出很多類別，為了更進一步表示類別之間的關係，我們可以定義「類別階層」(class hierarchy)。
- 在類別階層中，比較大的集合稱作「上類別」(super class)，而它的部分集合稱作「下類別」(sub class)。譬如說，如果我們以「學生」類別代表整個學校的學生，若我們在其中希望針對資訊工程系的學生特別記錄相關資料，我們可以再定義一個「資工系學生」類別。

12-4 UML

- 由於資工系學生具有學生的所有特性，在類別階層中，我們就可以表示學生類別為「上類別」，資工系學生為「下類別」，而「資工系學生類別」裡的物件，直接具有「學生類別」定義的所有屬性和方法，這種性質稱作「繼承」(inheritance)。

12-4 UML

- **物件導向的分析**(Object-Oriented Analysis；OOA) 和 **物件導向的設計** (Object-Oriented Design；OOD)，基本上包含下列步驟：



12-4 UML

- 注意到物件導向的分析和設計並不是如瀑布式模式的線性過程，而是在必要的時候，會遞迴重複上列步驟數次，直到設計結果滿意為止。
- UML一個很大特色，就是提供了很多圖形化的工具，以下列舉幾個作為代表：

12-4 UML

類別圖(class diagram)

- 描述系統中有哪些類別，及類別內的資料和方法。

使用情況圖(use case diagram)

- 顯示使用者和系統之間的互動，特別是使用者執行系統時會進行的過程。

活動圖(activity diagram)

- 描述系統內各個元件間工作執行的流程。

實作圖(implementation diagram)

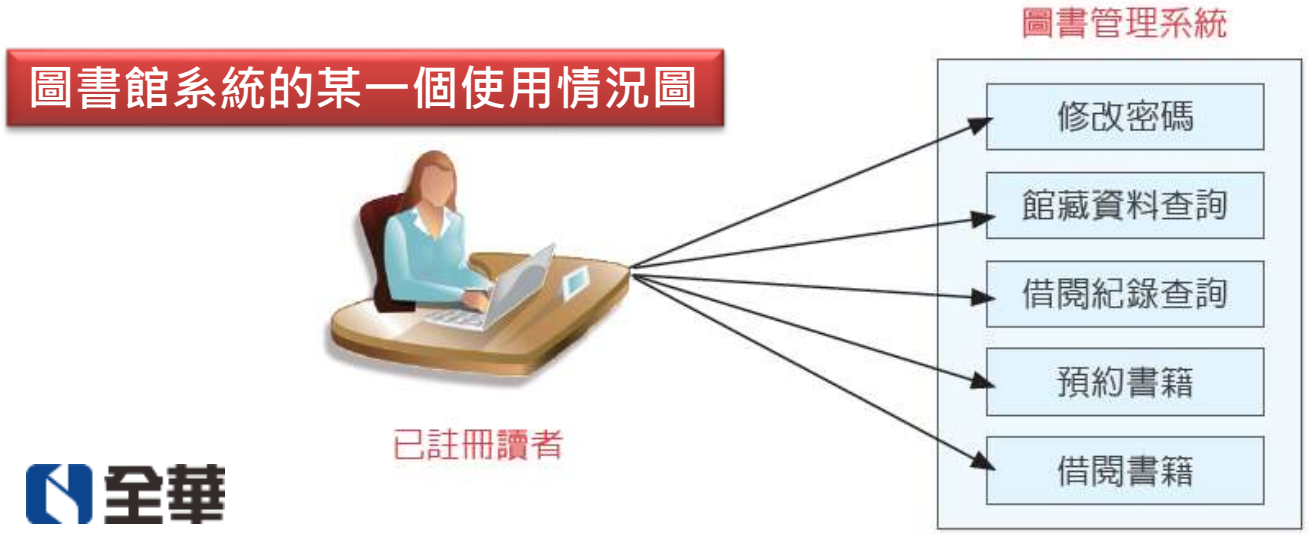
- 顯示系統架構內所設計的軟體元件和硬體元件，以及元件之間的互動。

12-4 UML

- 一開始，在描述使用情況時，我們可以利用圖形來表示。
- 我們描繪圖書館的「已註冊讀者」這種角色(role)的使用者，和圖書館系統之間可能進行的互動，也就是他可能執行的系統功能，包含了「修改密碼」、「館藏資料查詢」、「借閱紀錄查詢」、「預約書籍」、「借閱書籍」等這幾類。

12-4 UML

- UML也提供了所謂的「情節描述」(scenario)功能，也就是將一種可能使用系統的情形，從頭到尾一步步的用文字描述出來，類似寫電影的劇本一般。



12-4 UML

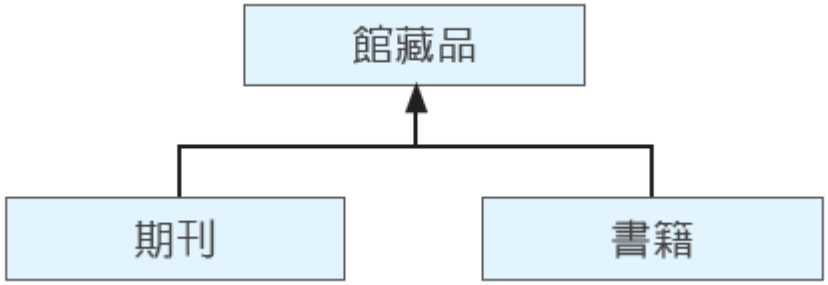
- 若將各類使用情況都詳細的列舉出來後，可從這些文字敘述和圖形找出可能的類別。
- 尋找的原則是：這些類別必須能夠將與系統有關的所有資訊都表達出來。
- 我們可以定義「館藏品」這個類別，來表示圖書館中所有收藏的紙本或印刷品；另外定義「書籍」，來表示一般可流通供讀者借閱的書本；而「期刊」則為定期出刊，限制在館內閱讀的印刷品。等到找出一些候選的類別後，再定義他們的屬性、方法，並歸納出類別階層等。

12-4 UML

- 在「類別圖」(class diagram)中，表示了系統中的三個類別及其構成的階層圖，在此，我們只標示了類別的名稱，還沒有定義其屬性和方法。
- 不過，透過一個三角形的符號，我們將「館藏品」定為「上類別」，然後「書籍」和「期刊」為其兩種下類別。

12-4 UML

- 除了表示階層之外，也可以在類別圖中表示類別之間的關係，像是一個「已註冊讀者」可以借閱10本「書籍」這樣的數量關係。



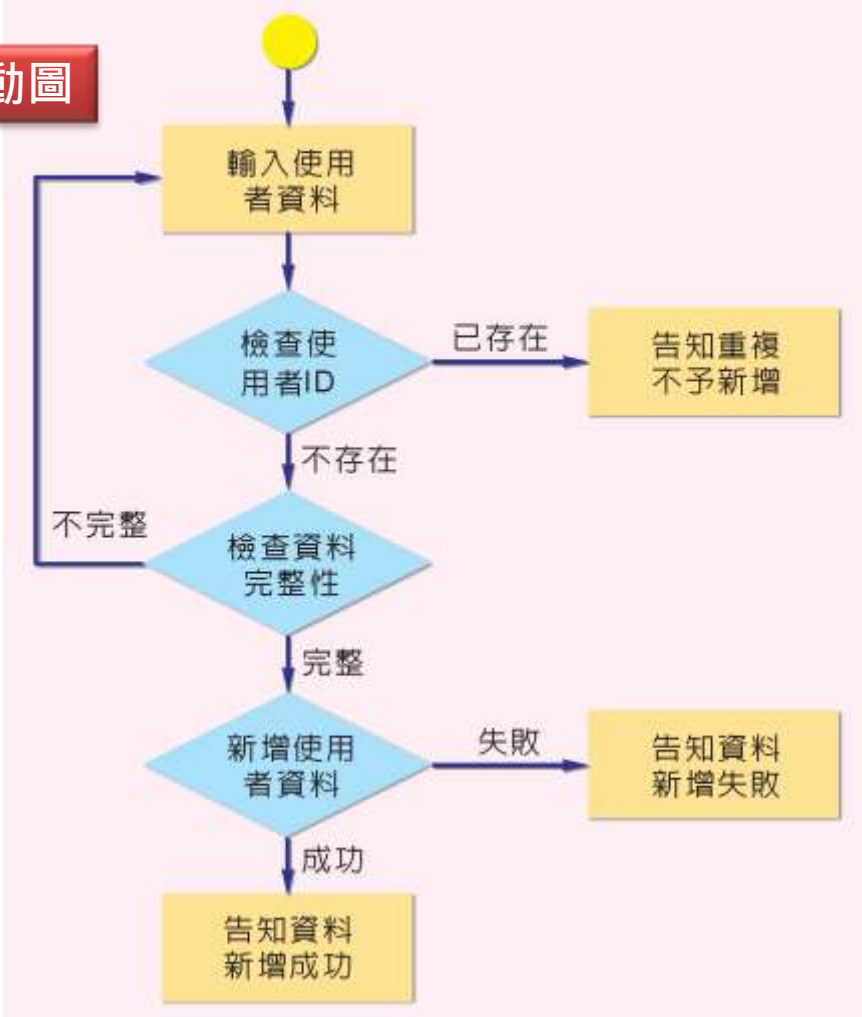
圖書館系統可能的類別圖

12-4 UML

- 在下圖中，我們描繪系統管理員在執行「新增使用者」這項功能時的「活動圖」。
- 根據該圖，我們可以看到，當使用者輸入其資料後，系統會先檢查他的ID，如果ID存在，則告知重複的訊息，並且不執行新增的動作；如果ID不存在，系統會判斷資料的完整性，再決定是執行新增的動作，或是再度回到「輸入使用者資料」的狀態。

12-4 UML

新增使用者的活動圖



12-4 UML

- 狀態圖(state diagram)和活動圖的差別，在於針對某一類別，描繪它可能的行為和被使用的情況，所產生的狀態改變。
- 透過這些行為和動態的分析，我們會再遞迴回去修改類別圖，將每個類別適當的方法(method)定義好，以確定這些功能會被適當的類別實踐(implement)出來。經過來來回回的分析，才會完成最後系統的物件設計。