第三章

CHAPTER 3

程式審視、排演與複查

Program Inspections, Walkthroughs, and Reviews

在軟體界的早期，原本都是認為寫程式純粹是為了餵給機器執行的，那時候並不會特別為了人在閱讀程式時的便利性而考慮，所以，要對程式進行測試的話，唯一的方法就是拿程式在機器上實際去跑。這種情形到了 1970 年代開始改觀，像是在 Weinberg 所著的 *The Psychology of Computer Programming*[1]一書中即強調並鼓吹程式閱讀方便的概念，如此程式在偵錯時會比較有效率。

如今，藉由肉眼閱讀來檢查程式已變為可行，以此為基礎的測試即稱為 non-computer-based testing (或稱「human testing」)，這也是本章的主題。經驗顯示，「human testing」技術在搜尋錯誤方面其實是非常有效的，所以也廣泛應用於許多的軟體專案之中，其應用的時機通常是在程式撰寫好之後到進行 computer-based testing 之前，computer-based testing 將在本章之後探討。另外還有一些類似的技巧可以應用在一些軟體開發的早期階段(例如每個設計階段結束之時)，不過這些方法已超過本書所要討論的範圍，有興趣的話可以參閱參考資料[2,3]。

別以為本章探討的方法從理論的角度上看來似乎並不怎麼正統(不正統係指相對而言，正統方法如程式正確性的數學證明等等，目前為止尚未成熟到可以實用的階段)，看到不正統這個字眼，你的反應可能覺得是那種旁門左道的方法，不屑一顧，然而事實上，我們在此將要討論的卻是真正能有效提昇生產力與可靠度的好方法，首先，由於可以提早發現錯誤，於是除錯的成本得以降低，除錯成功的機率得以提高，其次，程式設計師通常是越早發現錯誤在心理上會比較沒有

壓力，如果到了 computer-based testing 時再來除錯，那種壓力往往使程式設計師在除錯的時候更容易導致其他更多的錯誤。

審視與排演(INSPECTIONS AND WALKTHROUGHS)

程式碼審視和排演是「human testing」的兩個主要方法。我們在此先討論其共通性，後面再分別討論其差異處。

這兩種方法都是動用一個小組的人力召開會議去看同一份程式碼，會議的目的是挑錯，而非除錯，小組成員在會前也都要先行預習。

由於是一組人共同進行測試，所以符合上一章所提到的「程式設計師應避免測試他自己寫的程式」原則，事實上也的確比「desk-checking」(也就是原程式設計師自己檢視程式)有效，這是因為這兩種方法可以真正抓住錯誤的本質，所以除錯成本較省，也由於一次可抓出**一系列**的錯誤，所以可以將錯誤一次徹底解決，相對於 computer-based testing，則通常只能發現錯誤的**徵兆**，這些徵兆像是程式無法終止執行或是印表機印出一些無意義的東西等等，你有時只能根據徵兆去推敲真正的錯誤原因。

經驗告訴我們，一般的程式通常可以用這兩種方法找到 30%到 70%的邏輯設計和編寫程式碼方面的錯誤，不過對更高階的設計與需求分析層次的錯誤則比較沒有效。有份研究指出這兩種方法大約有 38%的偵錯率[4]，而 IBM 公司也曾指出他們測試結束後所找到的所有錯誤中有 80%是透過這兩種方式偵測出來的[5]，你可能認為這兩種方式只能找到簡單的錯誤，大錯誤可找不出來，但已有研究證實這是錯誤的想法[6]。另外還有份研究指出，human testing 對**某些**種類的錯誤比 computing-based testing **更加**有效[4]，**當然，其他種類的錯誤可能用 human testing 沒效，而 computing-based testing 則會比較有效**，因此，審視/排演與 computing-based testing 是互補的，僅單靠其中一種都是不夠的。

不論是新程式或是改寫過的程式，採取審視和排演同樣有效，尤其是改寫過的程式碼通常比寫新程式更容易滋生錯誤，對於這些改寫過的程式碼更要加強進行審視與排演。

程式碼審視(CODE INSPECTIONS)

程式碼審視是一個小組的人靠共同閱讀程式碼來偵錯的技術與程序[7]。這方面大部分的討論都是把焦點集中在程序或是需要填寫的表格等等，在此我打算先簡略說明一下程序，然後將焦點擺在實際的偵錯上。

小組通常包括四個成員，其中一人為主持人，通常是優秀的程式設計師，但非程式碼的原設計者，他也無須了解程式的細節，其責任包括會前分配程式碼、會議時控制流程、記錄會議中所發現到的錯誤、會後則追蹤錯誤的改正，所以主持人相當於品質控制者。其他小組成員包括程式碼的原設計者，還有曾經共同參予設計的其他工程師或是測試專家。

一般的程序是主持人在會議前幾天先分配程式碼與設計規格給其他小組成員，使得每個成員在會議之前都有時間預習。會議時則進行兩項活動：

1. 程式碼的原設計者先對他自己寫的程式加以說明，過程中允許發問並追捕錯誤。經驗告訴我們，通常許多錯誤都會由原設計者自己發現。換句話說，光是大聲地向別人說明自己程式的這種簡單做法，其實是很有效的偵錯技巧。

2. 程式碼將與根據過去所記錄的錯誤所設計出來的檢查列表(checklist)一起分析(這種檢查列表將在後面說明)。

主持人必須控制會議的討論是沿著主軸進行，使小組成員的關注焦點集中在發現錯誤，而非修正錯誤(除錯是會議之後原程式設計師的事)。

會後，原程式設計師收到了一份錯誤列表，如果錯誤很多，或是必須進行大幅修改，主持人也許有必要在錯誤修正後再準備開一次會。所有的錯誤也被分析、分類，同時也作為充實檢查列表的參考，使以後能更有效的進行審視會議。

再次強調，審視只專注於偵錯，而非除錯。盡可能不要被旁枝末節打斷而偏離主題，不過倒是有一篇文獻提出了與一般相反的看法[8]：

> 一開始大家都是將焦點集中在程式裡最基本的問題上面，隨著不斷討論，漸漸地，有些人就會開始注意到一些小細節，也許是其中的兩、三個人，包括程式的原設計師本人，開始修正一些原來的做法，以滿足一些原來他沒有考慮到的事情，這些小細節會將大家的注意焦點轉移到其他的觀點上面，待大家對原來的設計做了一些小修正，又會有第二個小細節的觀點被引發，像這樣，原設計便會經過許多其他觀點的考驗...由上述的說明可知，許多重要的問題都是源於一個看似沒啥重要的小細節的討論所引發。

審視過程應極力避免因被無關的議題打斷而偏離主題。進行審視所花費的時間最好是 90 到 120 分鐘，這是因為程式碼的審視非常耗費精神，若會議太長，進行到最後成效將會變差，通常一小時大約可以審視 150 行程式，所以大程式可能得分成幾次會議審視，每次只針對幾個模組或副程式。

注意！為有效進行審視，每個小組成員的態度很重要，如果原程式設計師把審視當作是別人來故意找碴，而在內心採取防禦的心態，其結果將難有良好的成效。正確的態度應該是對事不對人，原程式設計師應對審視有正面、建設性的認知[1]，目的是幫助他偵錯，增進他所寫程式的品質。也因為顧及原程式設計師自尊的緣故，有時審視會議的結果是保密的，只有參予者知曉。

除了偵錯，進行審視也會獲得額外的好處，從錯誤的發現中，每個成員都可以學到很多，得到許多不同意見的回饋，包括程式風格、演算法的選擇、或其他程式撰寫技術。還有一個好處是可以統計出最容易出錯的程式碼部分，這個部分是之後進行 computer-based testing 時應特別關注的部分。

審視的錯誤檢查列表(AN ERROR CHECKLIST FOR INSPECTIONS)

檢查列表是進行審視的重要工具，不幸地，一般檢查列表的內容對程式風格的注重反而多過對錯誤的注重(例如：「這些註解正確嗎？有意義嗎？」或是「THEN/ELSE 和 DO/END 有對齊嗎」)[9]，而對錯誤的檢查卻模稜兩可(例如：「程式有按照設計的要求撰寫嗎？」)，以下是一個檢查列表的範例，是我累積多年的經驗而整理出來的，這份列表大部分與所採用的程式語言無關，這表示可以用來檢測任何程式語言都會發生的錯誤。

【譯註】

以下這個 code inspection 的錯誤檢查列表是原作者在 20 年前用的，有些項目可能已經過時，一方面是因為某些種類的錯誤在目前流行的程式語言(如 C++)裡並不大容易發生，另一方面是由於當今編譯器的功能已非常進步，許多項目的檢查已可利用編譯器來代勞，不需再依靠人力檢查。不過，大部分項目所隱含的概念還是非常有價值的，你可以依你自己的經驗或專案特性，以此為基礎，設計出適合你自己的錯誤檢查列表。

資料參用錯誤(Data-Reference Errors)

1. 有任何並未設定初始值的變數嗎？變數未設初始值，但是每個參用到(reference to)這個變數的資料卻都假設這個變數已經有值，這種錯誤經常發生。
2. 所有參用到陣列的資料,所用的索引值是定義在該陣列維度(dimension)的範圍內嗎？
3. 所有參用到陣列的資料，所用的索引值是個整數值嗎？有些程式語言沒這問題，但若是有此問題，就可能會導致嚴重的後果。
4. 任何透過指標(pointer)參用到某一個變數的時候，該變數的儲存空間確實已配置成功了嗎？這種錯誤就是所謂的「dangling reference」，當一個指標參用到一個副程式裡的區域變數，然後這個指標又會在此副程式之外的地方使用，當此副程式結束(也就釋放了這個區域變數所配置的儲存空間)，問題就會發生。你應該養成使用指標之前確認指標所指的變數已配置儲存空間的習慣。
5. 如果一個配置的空間擁有不同的別名(alias)，那麼當透過這些別名來參用此配置空間內的資料時，有使用正確的屬性來看待這些資料嗎？這種錯誤有可能發生在 PL/I 中的 DEFINED 語法、Fortran 中的 EQUIVALENCE 語法、或是 Cobol 中的 REDEFINES 語法，例如：在一個 Fortran 的程式中，假設有一個實數變數 A 和一個整數變數 B，兩者利用 EQUIVALANCE 語法使得 A 和 B 事實上代表的是同一塊記憶體空間，當程式以 A 設了一個值，卻用 B 來參用這塊空間，就會發生儲存資料是浮點格式，卻用整數格式來解釋資料的錯誤。
6. 一筆資料中的變數，其型別或屬性確實符合所使用的資料結構嗎？這種錯誤可能發生在 PL/I 或 Cobol 的程式中,當程式以某一種資料結構讀取一筆資料，然而此實際上這筆資料當初並不是以這種資料結構來儲存，就有問題。
7. 記憶體配置語法的最小單位小於實際儲存的位址所能代表的最小單位嗎？例如：一個 PL/I 的程式在 IBM S/370 下執行，PL/I 中的字串允許以一個 bit 為單位來處理，無須考慮 byte boundary 的問題，然而 IBM S/370 中所有指到記憶體中的位址，卻都會以 byte boundary 為起始，當程式計算出此字串的位址，實際上卻是跟儲存在記憶體中的位址是有差異的，因此若使用此位址來參用此字串的時候，就會發生錯誤。
8. 指標的型別與它所指到的資料的型別是一致的嗎？例如：一個 PL/I 程式中的指標，其型別是代表某一個資料結構的指標，卻被用來指到另一個不同型別的資料結構。
9. 如果某一個資料結構將被許多個副程式參用到，這些副程式都會根據相同的定義來對待這個資料結構嗎？
10. 以索引值來存取字串時，該索引值會指到超過字串的範圍之外嗎？
11. 以索引值來存取陣列時，有任何「off by one」的問題嗎？

資料宣告錯誤(Data-Declaration Errors)

1. 所有的變數都經過正確的宣告嗎？這也許會通過編譯器的檢查，但卻是引發其他問題的根源，例如：一個 Fortran 的副程式接收了一個陣列的參數，然而此參數的型別事實上宣告的並不是個陣列(像是在使用 DIMENSION 語法的時候)，而是一個函式的指標，結果當參用這個陣列(像是 X=A(I))時，卻被解譯成呼叫一個函式。
2. 如果變數沒有做任何宣告，其預設的型別與屬性有被確實了解嗎？
3. 如果宣告變數時同時設定了初始值，此初始值確實是正確的嗎？大部分的程式語言中，陣列或字串的初始值設定都較為繁複，容易出錯。
4. 每一個變數都指定了正確的長度、型別與屬性嗎(例如：PL/I 中的 STATIC、AUTOMATIC、BASED、CONTROLLED 語法)？
5. 設定初始值的語法符合正確的需求嗎？例如：一個 Fortran 副程式在每次被呼叫的時候都必須重新設定某一個變數的值，那麼就必須使用設值(assignment)的語法，而不是用 DATA 語法，同樣的情形在 PL/I 的程式中，就是用 AUTOMATIC 語法，而不是用 STATIC 語法。
6. 有變數的名稱是極度類似的嗎(例如：VOLT 和 VOLTS)？這不是錯誤，但是容易搞混，搞混就容易導致別的錯誤。

計算錯誤(Computation Errors)

1. 有任何利用非數學的變數來作數學計算的情形嗎？
2. 有任何利用不同型別的變數來作數學計算的情形嗎？典型的例子就是一個浮點型別的變數和一個整數型別的變數執行加法運算，這也許不是錯誤，但使用這個運算之前，必須確實明瞭你所使用的程式語言在作型別轉換時所採用的規則，尤其是當這項規則非常複雜時(例如：PL/I)。例如：以下是一個 PL/I 的程式片斷：

```
DECLARE A BIT(1);
A=1;
```

   結果是，A 的值其實是 0，而不是 1。
3. 有任何利用不同長度的變數來作數學計算的情形嗎？尤其是 PL/I 的程式，例如：25+1/3，其計算出來的結果是 5.333...，而非 25.333...。
4. 設定給某一個變數的值，其大小有超過該變數的型別所能代表的範圍嗎？
5. 有任何運算會可能導致溢位(overflow)或不足位(underflow)的情況嗎？也就是說，運算出來的結果會是個合理但不正確的值，因為是它大於或小於電腦在運算時所能代表的範圍。

6. 有任何運算會可能導致除零的情況嗎？
7. 在你所使用的電腦下，以 2 為底(base-2 form)的計算結果是正確的嗎？也就是說，在 binary machine 中，可能會發生 10×0.1 的運算結果不是 1.0 的情況。
8. 在任何運算過程中，是否會發生變數的值將超過它所能代表的範圍的情形？例如：對於 PROBABILITY 這個變數，每次設值給它時，你都必須作一些檢查，以確保它的值始終是正的，且不大於 1.0。
9. 對於一個包含多個運算元(operator)的計算式，各個運算元的優先順序有被確實了解嗎？
10. 有任何不恰當的整數運算嗎？特別是整數除法，例如：假設 I 是一個整數變數，2*I/2 的結果是否會等於 I，其實是跟 I 是奇數或是偶數有關，也與乘法或除法的運算優先順序有關。

比較錯誤(Comparison Errors)

1. 任何使用不同型別的變數作比較的情形嗎(例如：將一個字元字串和一個指到某一塊記憶體的指標做比較)？
2. 任何使用不同長度的變數作比較的情形嗎？如果有，請確實明瞭轉換規則。
3. 所使用的比較運算元(comparison operator)是正確的嗎？程式設計師常常搞混一些像是**至多**、**至少**、**大於**、**不小於**、**小於等於**這些比較關係。
4. 任何布林比較運算是正確的嗎？程式設計師常常搞混「且(and)」、「或(or)」、「否(not)」的比較運算。
5. 任何執行布林運算的變數，其型別確實是布林值嗎？是否有將布林比較和布林運算搞混，這是常見的錯誤，例如：如果要判斷 I 是否介於 2 和 10 之間，2<I<10 便是錯誤的寫法，應該寫成 (2<I)&(I<10) 才對；如果要判斷 I 是否大於 X 或 Y，I>X|Y 便是錯誤的寫法，應該寫成 (I>X)|(I>Y) 才對；如果要判斷三個數字彼此是否相等，寫成 IF(A=B=C) 便有可能出錯；如果要表示出 X＞Y＞Z 這種關係，正確的寫法應該是 (X>Y)&(Y>Z)。
6. 有任何帶小數部分(fractional)或是浮點數字(floating-point number)的比較嗎？有些錯誤是源自於捨位(truncation)或是將以十為底(base-10)的數字轉換成以二為底(base-2)的數字的時候。
7. 當一個比較運算包含多個布林運算元時，它們的運算優先順序是正確的嗎？也就是說，當你看到 (A=2)&(B=2)|(C=3) 時，是否以了解 **and** 運算元和 **or** 運算元是最優先計算的。
8. 你明瞭所使用的編譯器對布林運算表示式的解譯規則嗎？例如：

```
IF(X!=0)&((Y/X)>Z)
```

這種寫法是否會被 PL/I 的程式編譯器所接受(也就是說，當 **and** 運算元的左邊為 false 時，是否真的就不再執行它右邊的運算)？這種寫法送到某些編譯器就有可能編譯出將導致除零情況的程式。

流向控制錯誤(Control-Flow Errors)

1. 如果程式裡有多個分支(branch)流向(像是 Fortran 中的 GOTO 語法)，代表各分支的索引值是否會超過它應該表示的範圍？例如：

```
GOTO (200,300,400),I
```

   其中 I 的值永遠只會是 1、2 或 3 嗎？
2. 每個迴圈最後都會終止嗎？你可能有必要利用一些方式來確保迴圈會終止。
3. 任何程式、模組、副程式最後都會終止嗎？
4. 有任何迴圈會發生永遠不會被執行到的情況嗎？例如：一個迴圈的前兩行程式為：

```
DO WHILE(NOTFOUND)
DO I=X TO Z
```

   其中，當 NOTFOUND 的起始值為 false，或是當 X 大於 Z 時，會造成什麼樣的執行結果呢？
5. 當一個迴圈由一個重複(iteration)運算的語法和另一個布林條件共同控制時(例如：一個搜尋迴圈)，「重複運算結束(loop fallthrough)」時會導致什麼樣的結果呢？例如：一個迴圈的第一行程式為：

```
DO I=1 TO TABLESIZE WHILE(NOTFOUND)
```

   其中，若 NOTFOUND 的值永遠不為 false 時，會造成什麼樣的執行結果呢？
6. 會導致「off by one」的反覆運算(iteration)嗎？
7. 如果採用了表示程式區段的語法(例如：PL/I 中的 DO/END 語法)，每一個程式區段都有用一個 END 語法作結尾嗎？每個 END 都對應到正確的程式區段嗎？
8. 有任何不充分的判斷嗎？例如：一個輸入的參數的值可能為 1、2 或 3，如果確定不是 1，也確定不是 2，就能保證它一定是 3 嗎？如果是的話，類似的這種假設是否合理？

介面錯誤(Interface Errors)

1. 呼叫模組時所輸入的參數個數是正確的嗎？順序也正確嗎？
2. 呼叫模組時所輸入的參數屬性(例如：型別和大小)是正確的嗎？
3. 呼叫模組時所輸入的參數的值所採用的單位是正確的嗎？例如：參數的值應該代表的是度，卻輸入一個單位為弳度的值。
4. 模組之間傳遞與接收的參數個數是否一致？
5. 模組之間傳遞與接收的參數屬性是否一致？
6. 模組之間傳遞與接收的參數的值所採用的單位是正確的嗎？
7. 當呼叫內建的函式，所輸入的參數個數、屬性、順序是正確的嗎？
8. 如果一個模組有多個進入點，有任何使用到的參數是當時採用的進入點不該使用到的參數嗎？以下面的 PL/I 程式碼為例，這種錯誤發生在第二個模組進入點：

```
A: PROCEDURE(W,X);
   W=X+1;
   RETURN;
B: ENTRY(Y,Z)
   Y=X+Z;
   END;
```

9. 有任何副程式企圖改變一個不該改變的參數的值嗎？
10. 如果使用了全域變數(global variable)(例如：PL/I 中的 EXTERNAL 語法，Fortran 中的 COMMAON 語法)，所有的模組都依照了共同的定義與屬性來參用它嗎？
11. 有任何將常數當作參數的情況嗎？例如一個 Fortran 程式：

```
CALL SUBX(J,3)
```

這是一種危險的寫法，因為如果在副程式 SUBX 中會指定一個值給它的第二個參數，常數 3 的值就會被更改。

輸出/輸入錯誤(Input/Output Errors)

1. 如果宣告了一個檔案，其屬性是正確的嗎？
2. 採用 OPEN 這種語法的時候，屬性是正確的嗎？

3. 儲存資料的格式與輸出/輸入的語法一致嗎？例如：Fortran 中的 FORMAT 語法事實上是需要搭配 READ 或 WRITE 語法來使用，而 PL/I 中也有類似的語法。
4. 存放輸出/輸入資料的空間大小是否符合每一筆記錄的大小？
5. 所有的檔案都是在 open 之後才開始使用嗎？
6. 偵測檔案結尾的機制是否正確？
7. 輸出/輸入的錯誤處理機制是否正確？
8. 程式中對文字檔列印或顯示的功能，是否有語意或文法上的錯誤？

其他(Other Checks)

1. 有任何從未使用到的變數嗎？
2. 確認所有變數的屬性是正確的，因為有些變數可能並沒有宣告，結果編譯器預設的屬性可能不是我們要的。
3. 如果程式通過了編譯，但是編譯器發出了「警告」或是一些「訊息」，這表示程式某些地方可能有問題或不恰當，你必須檢查這些地方。
4. 程式或模組夠強健(robust)嗎？也就是說，對於任何輸入值，它都會檢查其合理性嗎？
5. 有任何從未使用到的函式嗎？

以上的錯誤檢查列表可綜整成圖 3.1 和圖 3.2。

程式排演(WALKTHROUGHS)

如同審視，排演也是採取小組共同閱讀程式碼的方式，但在程序上稍有不同，所使用的偵錯技術也不同。

在進行一到兩個小時的排演會議之中，也是忌諱被旁枝末節岔題的。會議由三到五人參予，其中一人也是主持人，其職責類似在審視會議中的角色，另一人擔任紀錄，負責紀錄所發現到的錯誤，第三個人則是扮演「測試員」。通常第三到第五個人建議由以下幾種人擔任：(1)資深程式設計師，(2)程式語言專家，(3)新人(通常會有較新奇的想法，也較無偏見)，(4)將來要負責維護這份程式的人，(5)其他專案的成員，(6)同一專案小組的程式設計師。

資料參用檢查 (DATA REFERENCE)

1. 有任何並未設定初始值的變數嗎？
2. 陣列的索引值定義在合理的範圍內嗎？
3. 陣列的索引值是整數型態嗎？
4. 有「dangling reference」指標嗎？
5. 使用變數的別名(alias)時，與之搭配的屬性是正確的嗎？
6. 有用正確的資料結構來存取資料嗎？
7. 有適當地處理 byte boundary 的問體嗎？
8. 指標與它指到的資料，兩者型別一致嗎？
9. 多個副程式都會根據相同的定義來對待一個它們共同參用到的資料嗎？
10. 字串的索引值定義在合理的範圍內嗎？
12. 陣列的索引值有任何「off by one」的問題嗎？

資料宣告檢查 (DATA DECLARATION)

1. 所有的變數都經過正確的宣告嗎？
2. 明瞭沒有宣告的變數其預設的屬性嗎？
3. 陣列或字串的初始值有被正確的設定嗎？
4. 每一個變數都指定了正確的長度、型別與屬性嗎？
5. 設定初始值的語法符合正確的需求嗎？
6. 有名稱極為類似的變數嗎？

計算檢查 (COMPUTATION)

1. 有用非數學的變數來作數學計算嗎？
2. 有用不同型別的變數來作數學計算嗎？
3. 有用不同長度的變數來作數學計算嗎？
4. 設定給某一個變數的值，其大小有超過該變數的型別所能代表的範圍嗎？
5. 有任何會導致溢位或不足位的運算嗎？
6. 有任何會導致除零的運算嗎？
7. 任何以 2 為底的計算是正確的嗎？
8. 任何變數的值在運算的過程中是否會超過它所能代表的範圍？
9. 運算元的優先順序有被確實了解嗎？
10. 整數除法正確嗎？

比較運算檢查 (COMPARISON)

1. 有將不同型別的變數來作比較嗎？
2. 有將不同長度的變數來作比較嗎？
3. 所採用的比較運算元是正確的嗎？
4. 任何布林比較運算是正確的嗎？
5. 是否有將布林比較和布林運算搞混？
6. 有任何需要捨位(truncation)才能進行比較的運算嗎？
7. 明瞭比較運算的優先順序嗎？
8. 明瞭編譯器對布林運算表示式的解譯規則嗎？

圖 3.1 審視錯誤檢查列表(inspection error-checklist)，第一部份

流向控制檢查 (CONTROL FLOW)

1. 代表不同分支流向(branch)的索引值是否會超過它應該表示的範圍？
2. 每個迴圈最後都會終止嗎？
3. 任何模組、副程式最後都會終止嗎？
4. 有任何迴圈會發生永遠不會被執行到的情況嗎？
5. 迴圈 fallthrough 後的結果正確嗎？
6. 會導致「off by one」的反覆運算(iteration)嗎？
7. 每個 DO/END 對稱嗎？
8. 有任何不充分的判斷嗎？

介面檢查 (INTERFACES)

1. 呼叫模組時輸入的參數個數與順序都正確嗎？
2. 呼叫模組時所輸入的參數屬性正確嗎？
3. 參數的值所採用的單位正確嗎？
4. 模組之間傳遞與接收的參數個數是否一致？
5. 模組之間傳遞與接收的參數屬性是否一致？
6. 模組之間傳遞與接收的參數的值所採用的單位是正確的嗎？
7. 呼叫內建函式時所輸入的參數個數、屬性、順序是正確的嗎？
8. 有任何是當時採用的進入點不該使用到的參數嗎？
9. 有任何副程式企圖改變一個不該改變的參數的值嗎？
10. 所有的模組都依照共同的定義與屬性來參用同一個全域變數嗎？
11. 有任何將常數當作參數的情況嗎？

輸出/輸入檢查 (INPUT/OUTPUT)

1. 檔案的屬性正確嗎？
2. OPEN 語法的屬性正確嗎？
3. 資料格式與輸出/輸入的語法一致嗎？
4. 存放輸出/輸入資料的空間大小是否符合每一筆記錄的大小？
5. 檔案都是先 open 再使用嗎？
6. 偵測檔案結尾的機制是否正確？
7. 輸出/輸入的錯誤處理機制是否正確？
8. 輸出的文字資料是否有語意或文法上的錯誤？

其他 (OTHER CHECKS)

1. 有任何從未使用到的變數嗎？
2. 沒有經過宣告的變數，其預設的屬性是我們要的嗎？
3. 編譯器有發出任何警告或其他訊息嗎？
4. 程式裡對於任何輸入值都會檢查其合理性嗎？
5. 有任何從未使用到的函式嗎？

圖 3.2 審視錯誤檢查列表(inspection error-checklist)，第二部份

程式排演一開始是類似審視的程序，每個成員在會議前幾天便收到程式碼，先預習以大略明瞭程式的架構，但到了會議進行時則採取與審視不同的做法，其方法是由小組成員「扮演計算機」，由測試員提供一組測試樣本--包括具代表性的輸入資料與理論上應出現的輸出結果，這些測試樣本將依照程式的邏輯由人腦來執行，而執行過程中的狀態(例如一些變數的值)則可紀錄在紙或黑板上。

當然，測試樣本的設計不宜太複雜，因為人腦「執行」當然比機器慢，測試樣本只是用來引發大家對程式的邏輯與所用到假設提出疑問，因為實務上，錯誤多半是因提出的疑問而發現，而非測試樣本本身。

同樣，小組成員的態度很重要，評論應對事不對人。換句話說，所發現到的錯誤並不代表個人的缺陷，因為程式有錯本來就很正常。

會後的程序與審視相同，當然也會獲得與審視同樣的額外好處(找到具錯誤傾向的程式區段、從錯誤本身得到的啟示、程式風格、技術等等)。

DESK CHECKING

這裡要介紹的第三個 human testing 方法就是 desk checking，是個老方法，可以視為一人審視排演法。

對大多數人而言，desk checking 的效果並不大，原因之一是施行過程的好壞全靠程式設計師的個人素養，結果參差不齊，不易衡量好壞，第二個原因，也就是違反了上一章所提到的「程式設計師應避免測試他自己寫的程式」原則。或許你會說可以由另一個程式設計師來作 desk checking(例如：兩個程式設計師互換程式來 desk checking)，但即使如此，其效果仍不及於審視與排演法，原因是小組人員聚集在一起具有綜效(synergistic effect)，小組會議提供了一個良性的競爭環境，小組成員通常都喜歡炫耀錯誤是他發現的，所以錯誤被人發現時很容易就會被主動提出，這個效應在 desk checking 不會產生，所以偵錯的效果會較差。總之，desk checking 也許比不作測試好，但效果較差就是了。

PEER RATING

本章提出的最後一個 human testing 方法其實與測試程式沒有關係(因為這個方法原本的目的並不在偵錯)，提出來是因為它是一個不錯的閱讀程式碼的點子。

peer rating 是一種程式匿名評估技術(evaluating anonymous programs)[10]，可供程式設計師用來自我評估程式在品質、可維護性、擴充性、重用性與清晰度方面的好壞。

方法是，由一個程式設計師擔任監督者，招集約 6-20 個成員(最少 6 人才有匿名效果)，每個成員的技術背景最好差異不大(例如慣寫 Cobol 語言的人最好不要和組合語言專家放在一起)。每個成員自選兩份程式，一份是他認為寫得最好的，另一份則是自認為最差的。

待所有成員的程式都蒐集好之後，這些程式隨機分給每位成員，每個人被分到四份程式，其中兩份是「最好的」，另兩份是「最差的」，但這項資訊並不告知給每個人，然後每個成員對所分到的程式各花 30 分鐘閱讀。閱讀完之後，分別就所看的程式填寫評分表，每項評分項目的優劣程度用 1 到 7 之間的數字來表示(1 表示最好，7 表示最差)，評分項目例如：

程式容易瞭解嗎？
程式的高階設計架構明確且合理嗎？
程式的低階設計架構明確且合理嗎？
程式容易修改嗎？
你以寫出這份程式為榮嗎？

每個成員也必須寫下評語和改進建議。

最後，每個成員都會收到一份針對自己的兩份程式的匿名評分表，也有和其他程式相比較的統計資料，而相同程式的不同評分也將進行分析。上述方法的目的是提供程式設計師進行自我評估，可運用於實際專案或課堂上。

參考資料(REFERENCES)

[1]. G. M. Weinberg, *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold, 1971.

[2]. G. J. Myers, *Software Reliability: Principles and Practices.* New York: Wiley Interscience, 1976.

[3]. G. J. Myers, *Composite/Structured Design.* New York: Van Nostrand Reinhold, 1978.

[4]. G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Commun. ACM,* 21(9), 760-768(1978).

[5]. M. P. Perriens, “An Application of Formal Inspections to Top-Down Structured Program Development,” RADC-TR-77-212, IBM Federal Systems Div., Gaithersburg, Md., 1977(NTIS AD/A-041645).

[6]. M. L. Shooman and M. I. Bolsky, “Types, Distribution, and Test and Correction Times for Programming Errors,” *Proceedings of the 1975 International Conference on Reliable Software.* New York: IEEE, 1975, pp.347-357.

[7]. M. E. Fagan, “Design and Code Inspections to Reduce Errors in Program Development,” *IBM Systems J.,* 15(3), 182-211(1976).

[8]. R. D. Freeman, “An Experiment in Software Development,” *The Bell System Technical Journal, Special Safeguard Supplement,* S199-S209(1975).

[9]. J. Ascoly et al., “Code Inspection Specification,” TR-21.630, IBM System Communication Division, Kingston, N. Y. 1976.

[10]. N. Anderson and B. Shneiderman, “Use of Peer Ratings in Evaluating Computer Program Quality,” IFSM-TR-20, University of Maryland, 1977..