# 第二章

CHAPTER 2

# 測試程式的心理與經濟效益觀點

# The Psychology and Economics of Program Testing

對於測試，你可以從許多不同的角度切入，但我想最重要的就是經濟性(economics)觀點，以及人性心理(human psychology)的觀點，其議題包括了進行「完整」測試的可能性、該由誰來進行測試較為適合、以及該以什麼樣的心態來進行測試才較容易成功，而不是矇著頭完全只從技術性的觀點來考量。所以，本章先不討論技術，而是先討論心理與經濟效益的觀點。

首先，有一件事情相當重要，我必須花一些功夫來加以說明，一旦這件事弄清楚之後，隨之而來的討論都將會變得自然而簡單。

這件事就是對「測試」的定義。這裡特別強調定義的重要性，是因為失敗的測試往往肇因於大部份的人對測試的認知完全錯誤，例如：「測試就是用來驗證我們的程式沒有錯誤」、「測試的目的就是用來看看程式有沒有正確地執行」、「測試就是一種過程，使我們有信心說程式真的執行了我們要它做的事」。

以上對測試的想法是不正確的，它們甚至與正確的想法完全**相反**。想想看，測試是為了增進程式的價值(也就是說，測試是要耗費成本的，我們付出了代價當然要得到更棒的程式)，這價值可能指的是程式的品質或是可靠度，增進可靠度意指我們要儘可能地找到程式的錯誤並剔除錯誤，根據這個想法，我們不應該只是驗證程式有正確執行，相反的，在心理上，我們應該一開始就假設程式裡頭就隱藏著錯誤(對絕大部分的程式來說，這根本不只是假設，而是事實)，以這個假設為出發點，你才會儘可能地找出更多的錯誤。據此，一個合適的「測試」定

義是：

**為找出錯誤而執行程式的過程。**
*Testing is the process of executing a program with the intent of finding errors.*

聽起來有點微妙難懂，但這對於是否能進行一項成功的測試卻有著深遠的影響，人類的心理通常是高度目標導向的(goal oriented)，一旦在內心裡選擇了一個目標，就會在潛意識裡企圖滿足並達成這個目標，所以，如果我們的目標是去驗證程式有正確執行，那麼我們所選擇的測試資料在潛意識裡就會傾向於使程式執行成功，因此找到錯誤的機率將不高，相反地，如果心理上已經認定程式必定有錯誤，那麼我們所設計出來的測試資料就比較會有高度的機率來發現錯誤。後者的心態對測試而言，顯然是比較有價值的。

我們在上面對測試所做的定義，其實還隱含了許多的觀念，這些觀念遍佈在這本書中，例如：這樣的定義意指測試是種**毀滅性**(destructive)的行為，甚至是一種虐待程式的程序，以企圖使程式執行失敗，這可以解釋出發現錯誤為何是如此困難，因為這與一般的人性傾向於建設性(constructive)相違背。另外，這項定義也暗示了測試樣本該如何設計，以及應該由誰來測試程式。

還有一個方法可以加強我們對測試的正確觀念，就是，我們來探討一下什麼是「成功的」測試，以及什麼是「失敗的」測試，這兩個字眼是專案管理者常常用來判定測試好壞的字眼，按照字面上的意義，「成功」代表一切 OK，而「失敗」代表不符期望、有問題，於是，絕大部分的專案管理者都會把一切運作良好的測試歸類為「成功的測試」，而把發現到錯誤的測試歸類為「失敗的測試」，這就是用錯測試定義的結果。想想看，浪費了大筆的時間和金錢卻什麼錯誤都找不到的測試真的能歸類為「成功」嗎？

成功的測試應該是能找到錯誤的測試。

就好像病人給醫生看病，醫生也是先診斷、後醫療，而診斷就是一連串的測試，如果測試都正常，不知病因，便無從對症下藥，但病人感到不適卻是事實，因此我們說這個醫生診斷失敗，病人得向更好的醫生求診。好的醫生會一下子找到病人的毛病。那麼，我們通常把程式當病人還是當健康的人來看待呢？

另外，對於「測試就是用來驗證我們的程式沒有錯誤」這種說法，相信我，這是不可能的事情(這個問題即將在後面探討)，即使是一支小程式都不可能，糟的是，對於不可能的事情，我們通常都不會有把這件事情努力做好的企圖心，這種現象可以舉一個例子來說明，如果我們要求一個人要在 15 分鐘之內解出一個

字謎遊戲(crossword puzzle)，在前 10 分鐘，你通常可以觀察到這個人意興闌珊解不了多少字謎，因為反正他怎麼努力似乎也不可能在 15 分鐘之內完成，如果，我們一開始就給他 4 個小時，那麼你可能看到的是個充滿鬥志的人，甚至在前 10 分鐘就解出了大部分的字謎。所以，測試是必須定義清楚所需要做到的程度的，使它成為可行且務實的行動，這樣在心理上才會有正面效果。

還有，把測試視為「驗證程式執行了我們所預期它執行的行為」也是不恰當的，因為，程式也有可能**執行了我們並不預期它執行的行為**。例如第一章的三角形判斷程式，縱使在我們所能想得到的情況中，它都能正確地判斷不等邊三角形、等腰三角形、或是正三角形，但是我們沒有想到的情況呢？恐怕還是有錯誤的可能(例如：它可能判斷 1,2,3 為不等邊三角形，或是判斷 0,0,0 為正三角形)，這種錯誤往往藏在我們意想不到的地方，如果我們將測試的目標視為找到錯誤，而不只是驗證我們所預期的情況，就比較不會讓這種錯誤溜掉。

簡而言之，測試就是嘗試找到錯誤的毀滅性過程，成功的測試就是要想辦法讓程式當掉。當然，經由測試，最終的目的是建立對程式的信心，保證它會做該做的事，也絕不做它不該做的事，但這都必須從努力地發現錯誤開始，如果你想要信心十足地拍胸脯大聲保證「我的程式真的是完美無缺(error free)」，最好的方式就是企圖找到不完美之處，而不是只輸入一些資料來確定程式有正確執行。

## 測試的經濟效益觀點(THE ECONOMICS OF TESTING)

根據對測試所下的定義，我們接著要問，可能找得到**所有的**錯誤嗎？答案是否定的，即使是支小程式，這通常也是不切實際的，事實上，其實在大部分的情況下都不可能做得到，因為，測試是得花費成本的，你得考慮到經濟效益。我們依以下兩種測試方法來探討經濟效益的問題。

## 黑箱測試(Black-Box Testing)

**黑箱測試**(black-box)也叫**資料驅動**(data-driven)或**輸入/輸出驅動**(input/output-driven)測試。測試員完全不理會程式內部的結構與行為，只關心尋找程式未按規格運作的情況，因此測試資料純粹是根據規格所衍生出來的。

如果要用黑箱測試來找出所有的錯誤，方法就是普測**所有可能的輸入**(exhaustive input testing)，也就是將每一種可能輸入的情況都視為測試樣本。舉

例來說，或許你用了三個正三角形的測試樣本，程式也做出了正確的判斷，但並不保證其他你沒有使用到的測試樣本它都能做出正確的判斷，天知道3842,3842,3842 它會不會判斷成不等邊三角形，所以把程式當作是黑箱，除非你測過所有的情形，否則你無從證明起這樣的情況不會發生，你也無法斷定說程式已經沒有任何錯誤。

於是，你將發現，即使測試樣本的數目多到是個天文數字，還是不可能全測，更何況測試樣本還不僅只是侷限於**合理的**輸入，它應該是泛指所有**可能的**輸入，比如說 3,4,5 算是個合理的三角形，程式判定它是不等邊三角形，但是如果輸入的不是數字呢？像是 2,A,2，它不是個三角形，但是這種輸入卻是有可能的，程式是不是會將它判定成等腰三角形呢？你無從證明起，也無法一網打盡所有的類似情況。

聽起來實在很糟，測一個判斷三角形的小程式已這般棘手，如果要測一個編譯器呢？那豈不是要把所有可能的程式都要當作測試樣本了嗎？不單是合理正常的程式要測，以確保它能執行編譯的功能，甚至連不是程式的東西也要拿來測，才能確保這個編譯器不會做出不該做的事－例如，對一個不是程式的東西加以編譯。有「記憶」功能的程式更糟糕(例如：作業系統、資料庫系統、飛航記錄系統)，這些程式所執行的事情多半是前後具有關聯性的，於是，不只是單獨一項功能的合理輸入要測、不合理輸入要測，他們前前後後參雜在一起的所有情況也都要測。

說了一大堆，所要表達的就是這兩點：(1)保證程式絕無錯誤(error-free)是不可能的。(2)測試是要考慮到經濟效益層面的。也就是說，普測所有可能的情況是不切實際的，務實的做法應是儘可能在有限數目的測試樣本下來找到最多的錯誤。於是，我們可能會基於一些合理的假設來作測試(例如：如果程式會將 2,2,2 判定成正三角形，那我們便假設它也會將 3,3,3 判定成正三角形)，這也是在第四章探討測試樣本設計時所採取的一部份策略。

## 白箱測試(White-Box Testing)

**白箱測試**(white-box)又稱**邏輯驅動**(logic-driven)測試，測試人員必須檢測程式內部的結構與行為，測試資料則是根據程式的邏輯來設計(很不幸，我們通常卻是根據規格)。

如果要用白箱測試來找出所有的錯誤，要用什麼方法呢？你可能會說將每一行程式都讓它執行至少一遍，但很容易說明這麼做仍然是不夠的，原因將會在第

四章深入討論。唯一的方法就是普測**所有可能的執行路徑**(exhaustive path testing)，也就是將每一種程式可能執行的路徑流向都視為測試樣本。

於是，與黑箱測試類似，白箱測試的測試樣本數目也很可能多到是個天文數字，也幾乎不可能全測。舉一個例子來說明，如圖 2.1 所示，每一個圓圈代表一段循序性的程式碼(亦即其路徑僅有一種，中間不包含其他分支路徑)，箭頭代表控制路徑流向的轉移，如果程式由 A 到 B 是一個可能會執行 1 到 20 次的迴圈，像這樣的小程式會有多少可能的路徑呢？大約是 100 兆，算法是 $5^{20}+5^{19}+...+5^{1}$。

事實上，絕大多數的程式都遠比圖 2.1 複雜得多，所以普測所有可能的路徑就算可能，也不實際。

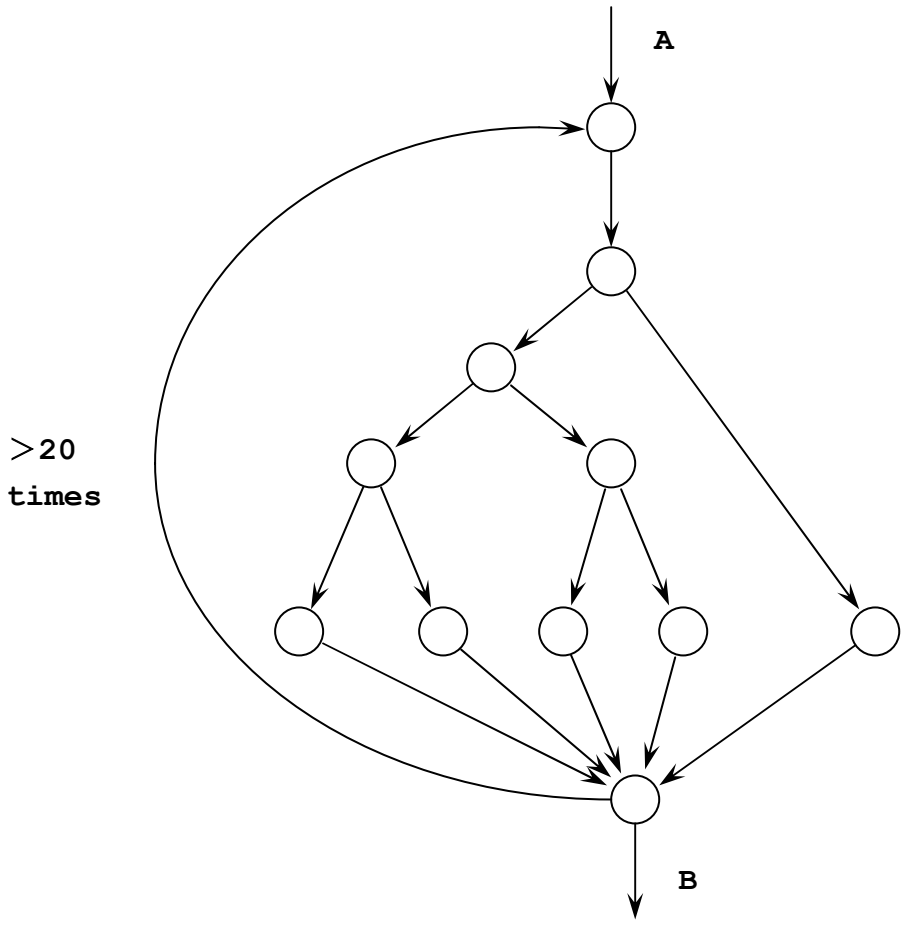


**圖 2.1 一個小程式的控制流程圖(control-flow)**

更糟的是，即使你能普測所有可能的路徑，測完了也不能保證程式沒有錯

誤，這可從三方面來解釋：第一、程式本身的設計就不符規格，例如規格是要求一個由小到大的排序，結果程式誤設計成由大到小的排序；第二、程式本身在邏輯上就**忽略掉某些該設計出來的路徑**；第三、源於**資料敏感**(data-sensitivity)的錯誤，例如我們要利用比較兩個數字的值來做收斂(convergence)的運算，也就是去判斷這兩個數字的差距是否小於某一個值，程式寫的像是這樣：

```
IF((A-B) < EPSILON)…
```

但如果事實上(A-B)是誤寫的，應寫成 ABS(A-B)，也就是忘了加上絕對值。以上三種錯誤，即使最完整的白箱測試也無能為力。

結論是，無論是何種普測都是不切實際的，但是，也許，將黑箱與白箱測試兩者合併使用，則設計出一種符合經濟效益的測試策略應是可行的方向，這些策略將在第四章仔細探討。

## 測試原則(TESTING PRINCIPLES)

依據心理觀點，以下列出一些極為重要的測試原則，這些原則多半顯而易見，但卻常常被忽略：

**每個測試樣本都必須定義所預期的輸出或結果。**
***A necessary part of a test case is a definition of the expected output or result.***

人都有選擇性知覺，也就是傾向於只看他想看的東西(the eye seeing what it wants to see)，而人都希望看到正確的結果，所以，如果測試樣本的結果沒有定義清楚，一些似是而非、看起來好像正確的錯誤很容易就會忽略過去。為此，測試樣本應明確包含兩個部分：輸入資料和與之對應的預期輸出結果。

理則學家 Copi 曾對這種人類的本性做過探討[1]：

> 人們往往不能接受無法迎合自己期望或偏見的事實，所以，對於那些似是而非的事情，抱持**一些**信念是必要的。

**程式設計師應避免測試他自己寫的程式。**
***A programmer should avoid attempting to test his or her own program.***

這是因為測試是種毀滅性的行為，但是寫程式卻是建設性的，你叫程式設計師的心態馬上從建設性切換成毀滅性是很困難的。

還有一個問題是，有的錯誤是源自於程式設計師本身對規格或解決方式的認知錯誤，由他自測程式，這項認知錯誤仍然存在。

此外，測試報告是屬於批評性的，程式設計師剛寫好他的程式後想辦法證明他寫的程式有一大堆錯誤，自己批評自己，這是違反人性的。

當然，並不表示程式設計師**不可能**自測程式，只是，如果由別人來測的話通常會比較有效也比較容易成功。但是如果是除錯，程式設計師當然是自己程式的最佳除錯者。

**程式設計團隊不應測試自己產出的程式。**
***A programming organization should not test its own programs.***

理由同上一個原則，另外就是通常程式設計團隊的績效是根據程式的完成日期與所花費的金錢，很難用程式的品質或可靠度來評估績效，而測試是要耗費時間與金錢的，所以程式設計團隊很可能為了績效而不重視測試。

同樣地，也並不表示程式設計團隊不可能自測程式，只是由別的團隊來測的話會比較有效。

**徹底檢視每一個測試結果。**
***Thoroughly inspect the results of each test.***

經驗顯示，有不少比例到最後才發現的錯誤是在老早的測試報告上就可以輕易發現的，只因之前的測試結果並沒有好好審視，才讓錯誤溜過。

**設計測試樣本時，不合理、不被預期的輸入情況，與合理、被預期的輸入情況同樣重要。**
***Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions.***

通常我們測試都傾向於輸入合理、被預期的資料，例如一個要輸入三角形三邊長來對三角形作分類的程式，很少人會輸入 1、2、5 來作測試。然而許多與眾不同的輸入情況卻往往比一般性的輸入更能發現錯誤。

**不要只看看程式是否做了它該做的事，更要看看程式是否做了它不該做的事。**

***Examining a program to see if it does not do what it is supported to do is only half of the battle. The other half is seeing whether the program does what it is not supported to do.***

這個原則是針對邊際效應(side effect)的問題，也就是程式在做完該做的事的同時，是否也造成了一些不該有的影響。例如一個發薪程式，除了產生正常的薪資資料，是否也為不存在的員工產生了薪資資料。

**除非你要丟棄程式，否則這個程式使用過的測試樣本不要丟棄。**

***Avoid throw-away test cases unless the program is truly a throw-away program.***

常見的情形是，測試員坐在終端機前天馬行空地輸入資料來測試程式，即使其中的測試樣本有了重大發現，卻沒有為後來的測試(例如修正錯誤後或程式改版後的測試)保留這些測試樣本，結果到時又必須重新建立一組測試樣本，然而這種重複性工作通常人們都傾向於避免它，於是後來的測試往往不及之前的測試來的嚴謹。

**不要基於程式沒有錯誤的假設來作測試。**

***Do not plan a testing effort under the tacit assumption that no errors will be found.***

這是弄錯測試定義的結果。

**還存在於程式裡的錯誤和已發現的錯誤數目成正比。**

***The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.***

這個違反常理的現象可用圖 2.2 來表示，比如說程式中包含 A 和 B 兩個模組，經過測試，在 A 模組中發現了五個錯誤，B 模組只發現了一個，這個原則告訴我們 A 模組將比 B 模組有更高的機率隱藏著尚未發現到的錯誤，因此，A 模組事實上是非常值得更進一步去進行更嚴格的測試。換句話說，錯誤有群聚的特性，雖然目前尚無人為此現象提供一個良好的解釋，但實務上卻是如此，例如在 IBM S/370 的作業系統中，被使用者發現的錯誤有 47%是落在 4%的程式碼之中。

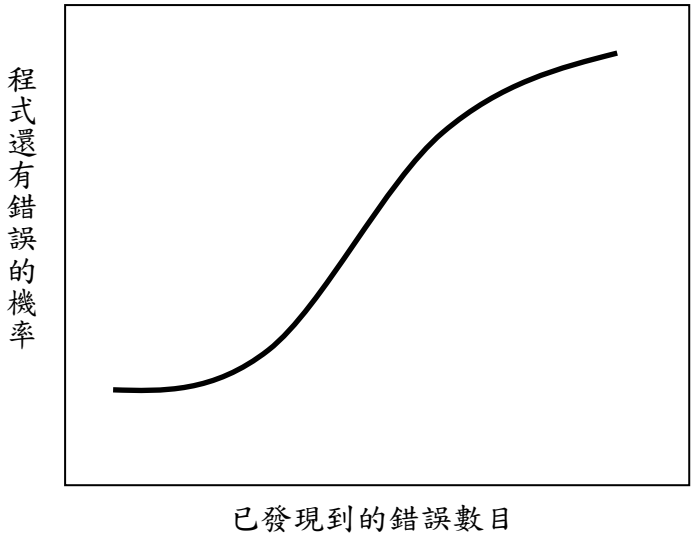


**圖 2.2 仍然隱藏在程式中的錯誤與已發現錯誤的關係**

這個現象為我們的測試過程提供了一個很好的指引，如果先前的測試在某些程式碼所發現的錯誤特別多，那麼之後的測試則應採取更嚴謹的態度來對待這些程式碼。

**測試，是一個極度需要創造力並發揮高度智慧的挑戰性工作。**
***Testing is an extremely creative and intellectually challenging task.***

測試一個大程式很可能比設計一個大程式需要更多的創造力。要偵測出所有的錯誤是不可能的，之後我們會探討許多實務上可行的測試方法，不過，它們都需要發揮創造力才能落實於軟體專案之中。

為了再強化這個章節所強調的概念，在此濃縮了最最最重要的三個原則來作為結論：

**測試是為找出錯誤而執行程式的過程。**
***Testing is the process of executing a program with the intent of finding errors.***

**最有可能找出新錯誤的測試樣本才是好的測試樣本。**
***A good test case is one that has a high probability of detecting an as-yet undiscovered error.***

**已找出新錯誤的測試樣本才是成功的測試樣本。**

***A successful test case is one that detects an as-yet undiscovered error.***

## 參考資料(REFERENCES)

[1]. I. M. Copi, *Introduction to Logic*. New York: Macmillan, 1968.