Chapter 8

軟體測試

本章內容

8.1 開發期間的測試
8.2 測試驅動式的開發方式
8.3 發行版本測試
8.4 使用者測試

- 測試是為了要證明程式有做到應該做的事，以及在程式正式上線使用前發現缺陷。
- 當你在測試軟體時，是使用編造的資料來執行程式，然後檢查測試結果是否有錯誤或異常，同時也收集與程式的非功能特性相關的資訊。

- 對軟體進行測試通常做兩件事：
  1. 向開發人員與客戶展示軟體的確符合需求。對於訂做軟體，這表示需求文件中的每條需求，應該至少設計一個測試。而假如是市售的軟體產品，這表示產品的發行版本中內含的所有個別功能，也應該經過測試。
  2. 找出有哪些輸入或輸入順序，會讓軟體行為不正確、非預期或不符合規格。這些是軟體缺陷（程式錯誤）造成的結果，測試軟體找出缺陷是為了發現不希望出現的系統行為，例如系統當機、與其他系統不該有的互動、不正確的計算，以及資料損毀等。

- 第一個就是所謂的確認測試 (validation testing)，這是使用一組事先準備好可以反映出系統預期使用方式的測試案例 (test case)，測試系統的行為是否正確。
- 第二個則是缺陷測試 (defect testing)，這裡的測試案例是設計成要暴露缺陷的，所以可能會刻意使用很少會出現的資料，而不是一般正常的資料。
- 當然這兩類測試方法並沒有明確的界線，在確認測試期間也可能找到系統的缺陷，而在缺陷測試期間的其中一些測試，也可能證明程式有符合需求。

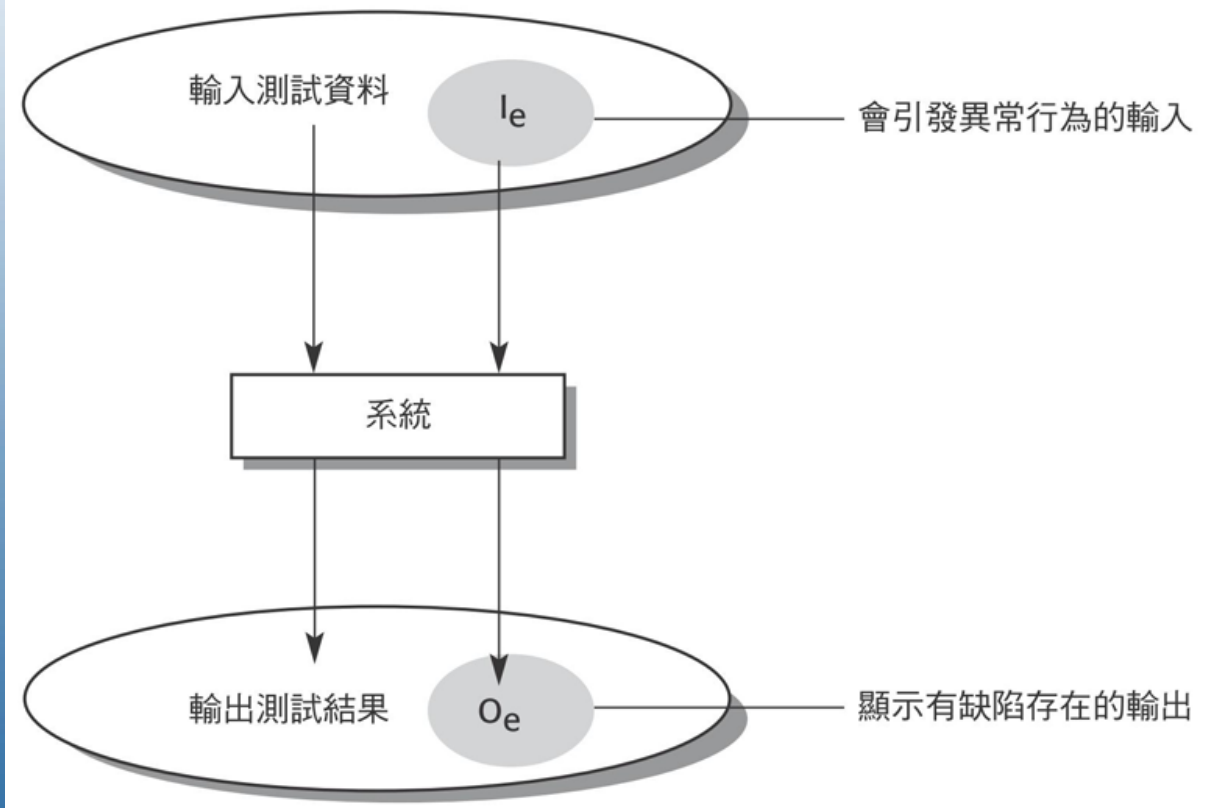

圖 8.1 程式測試的輸入輸出模型

- **確認 (validation)**：我們是否開發了對的產品？
  - (Are we building the right product?)
- **驗證 (verification)**：我們開發的產品正確嗎？
  - (Are we building the product right?)

- 驗證與確認程序的最終目的，是要建立軟體系統「有符合目的」的自信。所需的自信程度是依據系統的目的、系統使用者期望，以及系統目前的行銷環境而定：
  1. **軟體目的**：軟體越是關鍵，它的可靠度就越重要。
  2. **使用者期望**：許多使用者對他們使用的軟體品質普遍抱持較低的期望，而且對使用時出現的錯誤也不會感到驚奇。
  3. **行銷環境**：在很競爭的情況下，軟體公司可能會在產品尚未完成測試和除錯程序之前，就搶先發行以搶佔市場。

- 所謂的「靜態」(static) V & V技術，也就是不需要執行軟體就可以驗證它。

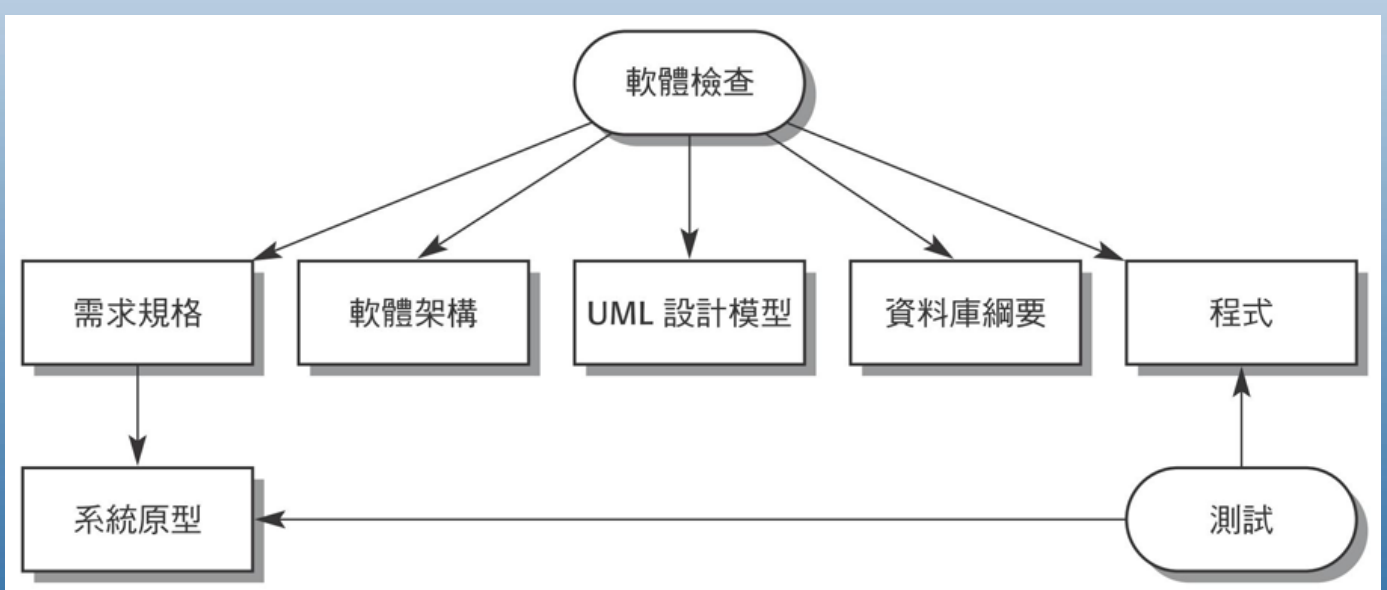

圖 8.2　軟體檢查與測試

- 軟體檢查勝過測試的主要優點有3個：
  1. 在測試期間，錯誤可能會掩蓋其他錯誤。因為檢查不會去執行系統，所以不必考慮錯誤彼此之間相互的影響。
  2. 尚未完成的系統也可以進行檢查，而且不必增加成本。
  3. 檢查除了可以找出程式缺陷之外，還可以一併檢查其他的品質特性，例如程式是否符合標準、它的可移植性和易維護性。

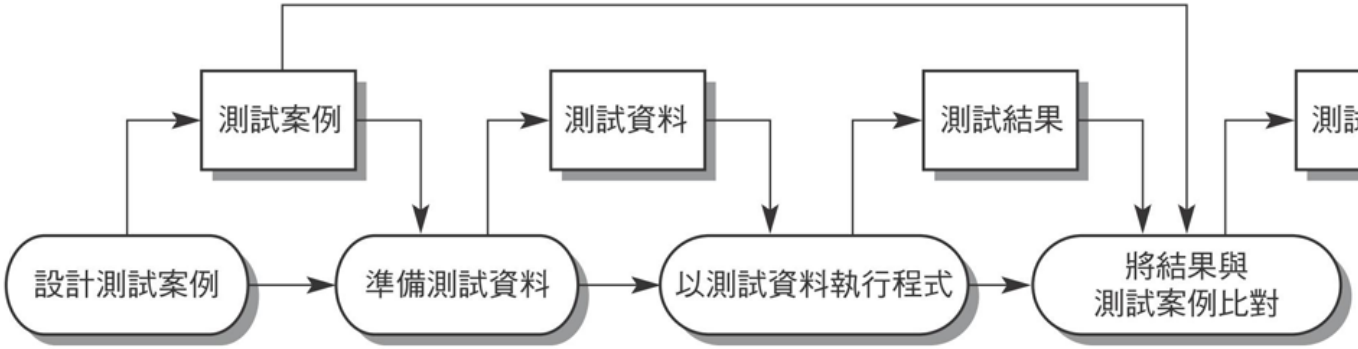

圖 8.3　軟體測試程序的模型

1. **開發期間的測試 (development testing)**：系統在開發期間進行測試，要找出程式錯誤和缺陷。
2. **發行版本測試 (release testing)**：由另外專門的測試團隊，在系統交付給使用者之前，先測試它的完整版本。
3. **使用者測試 (user testing)**：由使用者或潛在使用者在他們自己的環境中測試這個系統。

8.1 開發期間的測試

- 開發期間的測試包括由系統的開發團隊所進行的所有測試活動，此時軟體的測試人員通常就是負責開發此軟體的程式設計人員。
- 開發期間的測試分成以下3個階段：
  1. **單元測試 (unit testing)**：針對個別的程式單元或物件類別進行測試。單元測試應該針對物件或方法的功能做測試。
  2. **元件測試 (component testing)**：當一些個別單元被整合成複合元件時進行測試。元件測試主要是測試元件的介面。
  3. **系統測試 (system testing)**：當系統中某些或全部元件整合後，對系統整體做測試。系統測試應該主要是測試各元件彼此的互動。

8.1.1 單元測試

- 單元測試 (unit testing) 是測試程式個別元件 (如方法或物件類別) 的程序。
- 在測試物件類別時，所設計的測試應該涵蓋物件的所有功能，應該包括：
  - 測試與物件相關的所有運算動作
  - 設定和檢查與物件相關的所有屬性的值
  - 練習物件的所有可能狀態，這表示要模擬會造成狀態改變的所有事件

8.1.1 單元測試

- 舉例來說，以第7章的氣象站為例，這物件的屬性和運算動作如圖8.4所示。

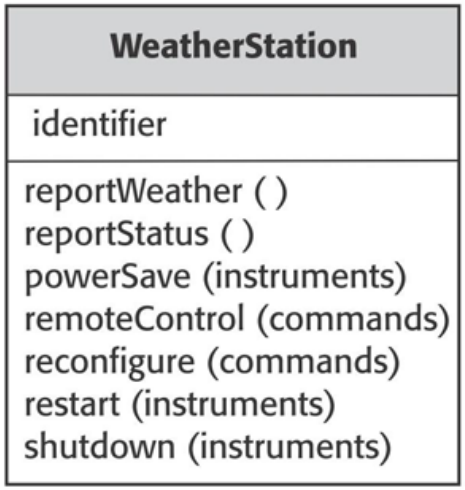

圖 8.4　氣象站的物件介面

8.1.1 單元測試

- 在測試氣象站的狀態時，可以使用如圖7.8的狀態模型。利用這個模型可以找出必須測試的狀態轉換序列 (sequence)，以及造成這些轉換的事件序列。

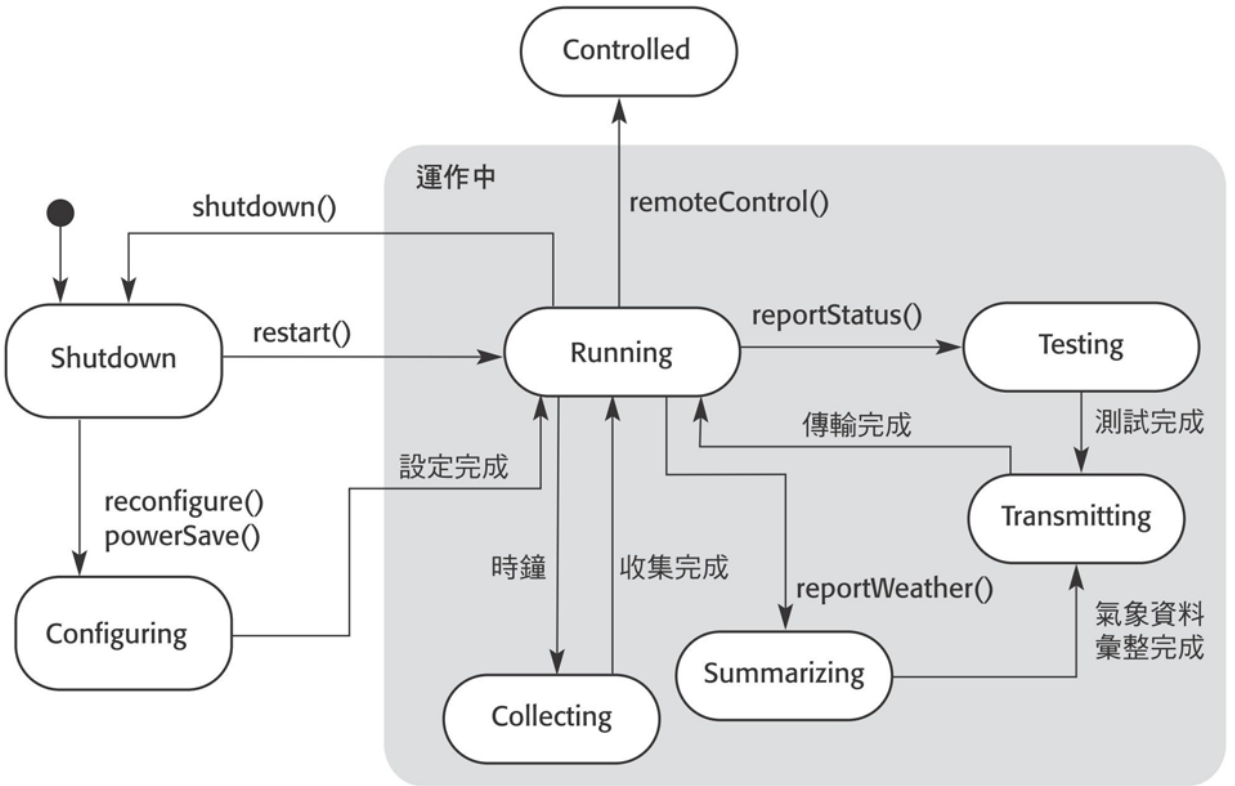

圖 7.8　氣象站的狀態圖

8.1.1 單元測試

- 氣象站需要測試的狀態序列範例有：
  - Shutdown→Running→Shutdown
    Configuring→Running→Testing→Transmitting→Running
    Running→Collecting→Running→Summarizing→Transmitting
    →Running
- 情況許可的話，應該儘量將單元測試自動化。在自動化單元測試時，你是利用某個測試自動化框架 (如JUnit) 來撰寫和測試你的程式測試 (Tahchiev et al. 2010)。單元測試框架提供通用的測試類別，你可延伸它來建立自己想要的測試案例。接著它們還能自動執行你設計的全部測試，並在GUI介面上回報測試是成功或失敗。

8.1.1 單元測試

- 每個自動化測試分成3個部分：
  1. **架設部分 (setup part)**：在此處初始化系統與測試案例，也就是各種輸入和預期的輸出。
  2. **呼叫部分 (call part)**：在此處呼叫被測試的物件或方法。
  3. **主張部分 (assertion part)**：在此處比較呼叫的結果和預期的結果。假如主張部分為真則測試成功；若為假則測試失敗。

8.1.2 挑選單元測試案例

- 選擇有效的單元測試案例是很重要的。這裡的「有效」(effective) 代表兩件事：
  1. 測試案例應該顯示出，在正常使用情況下，被測試的元件應該做該做的事。
  2. 如果元件中有缺陷，從測試案例應該可以看出來。
- 第一種是反映出程式的正常運作和顯示元件工作無誤。
- 另一種測試案例應該根據測試經驗，針對最常發生的問題來測試。例如故意使用不正確的輸入，測試程式是否有適當處理而不會讓元件當掉。

8.1.2 挑選單元測試案例

- 以下探討兩種能有效協助選擇測試案例的策略：
  1. **分割測試 (partition testing)**：先將輸入依照共同的特性而應該以相同方式來處理而分成幾組，再針對每一組輸入設計測試案例。
  2. **以原則為主的測試 (guideline-based testing)**：這是依據使用測試原則來選擇測試案例。這些原則反映出程式設計人員在開發元件時常犯的錯誤經驗。

8.1.2 挑選單元測試案例

- 程式的輸入資料和輸出結果，通常可根據一些共同的特性分成幾組。
- 因為它們有相同的行為，這些類別有時會被稱為等價分割區 (equivalence partition) 或定義域 (domain) (Bezier 1990)。
- 輸出的等價分割區則是具有共同特性的程式輸出。
- 在辨識出幾組分割區之後，就可以從這些分割區中選擇測試案例。

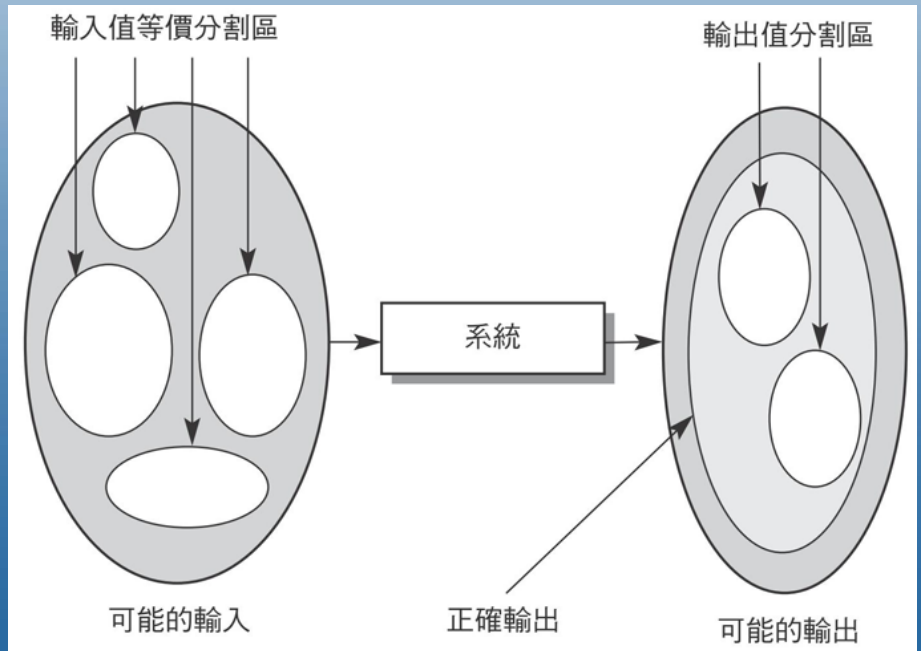

圖 8.5 等價分割動作

8.1.2 挑選單元測試案例

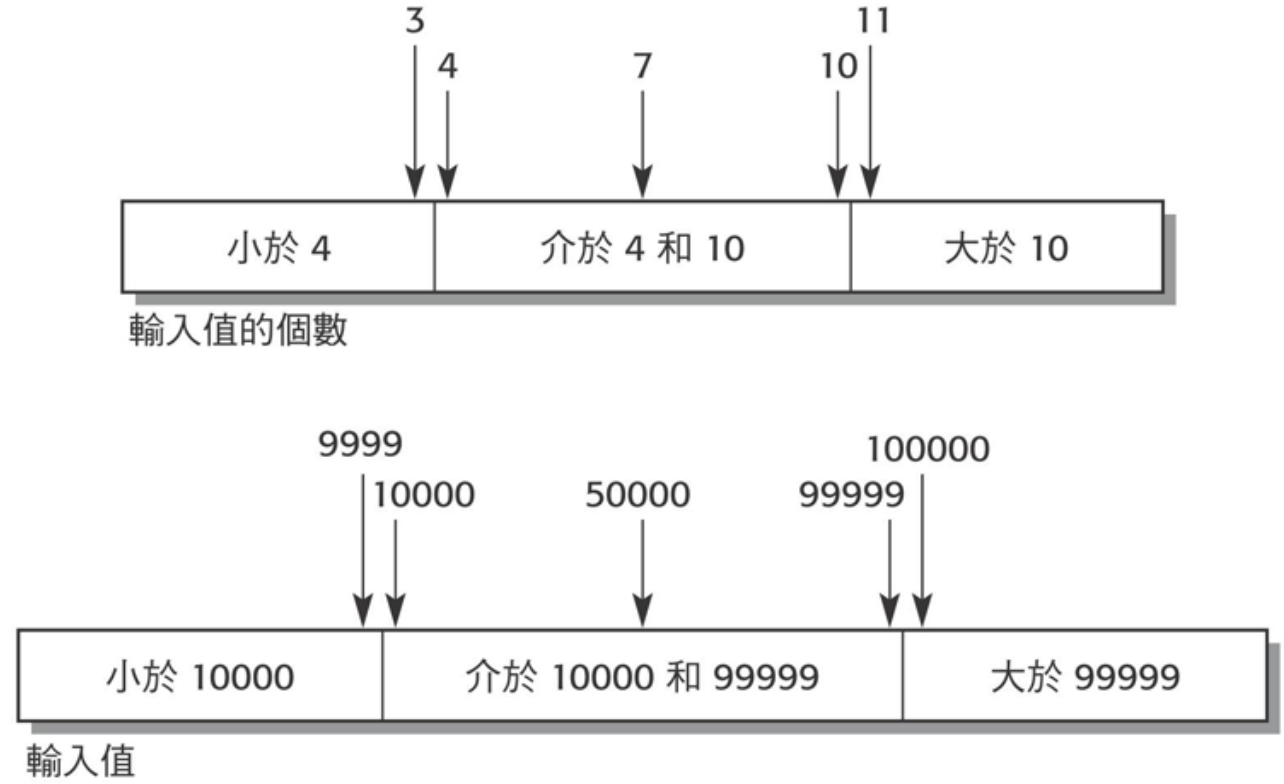

圖 8.6　等價分割區

8.1.2 挑選單元測試案例

- 有些原則能幫助找出缺陷：
  1. 使用只有一個元素的序列來測試軟體。
  2. 在不同的測試案例中使用不同長度的不同序列。
  3. 對序列的第一個、中間及最後一個元素進行測試，如此可以找出在分割區邊界上的問題。

8.1.2 挑選單元測試案例

- 例如Whittaker的書 (Whittaker 2009) 中整理出許多能用在設計測試案例的原則。其中一些最通用的原則如下：
  - 選擇會讓系統產生全部錯誤訊息的輸入
  - 設計輸入資料故意讓輸入緩衝區溢載
  - 重複多次相同的輸入或一系列輸入
  - 強制產生無效的輸出
  - 強制讓計算結果太大或太小

8.1.3 介面測試

- 軟體元件經常是由多個互動物件所組成的複合物件 (composite component)，例如氣象站系統的重設組態元件。要存取這些元件的功能是透過事先定義的元件介面 (參見第7章)。而測試這些複合物件主要是針對元件的介面是否符合規格來進行測試，至於元件中個別物件的單元測試，你可以假定已經完成。

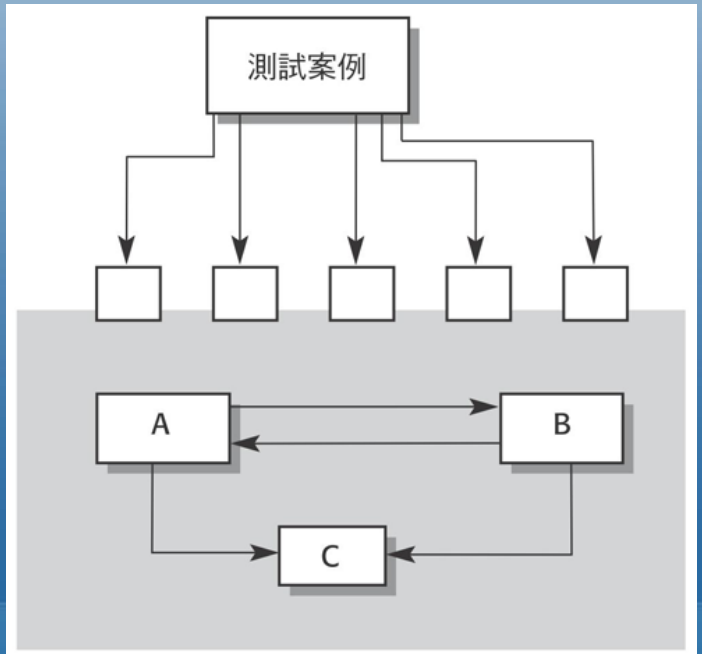

圖 8.7 介面測試

8.1.3 介面測試

- 程式元件之間有下列幾種不同類型的介面，會出現幾種不同類型的介面錯誤：
  1. **參數介面 (parameter interface)**：這些介面可以將某個元件的資料或函式參照位址傳給其他元件。
  2. **共用記憶體介面 (shared memory interface)**：這些介面可以讓各個元件共用同一塊記憶體。
  3. **程序介面 (procedural interface)**：這些介面可以將某一個元件封裝的一組程序，提供給其他元件呼叫使用。
  4. **訊息傳遞介面 (message passing interface)**：這些介面可以讓某一個元件向另一個元件，透過訊息的傳遞提出服務的請求。

8.1.3 介面測試

- 這些錯誤可以歸納為下列3類：
  1. **介面誤用 (interface misuse)**：元件在呼叫一些其他元件時，它的介面用法出錯。例如：參數順序不對。
  2. **介面誤解 (interface misunderstanding)**：呼叫端元件誤解被呼叫元件的介面規格，造成對被呼叫元件的行為有錯誤的假設。
  3. **時序錯誤 (timing error)**：這類錯誤常見於使用共用記憶體或訊息傳遞介面的即時系統中，這是因為資料的生產者和消費者可能以不同的速度在運作。

8.1.3 介面測試

- 介面測試有下列幾項一般原則：
  1. 在設計測試資料集時，儘量將傳給外部元件的參數值選擇合法範圍內的極端值。
  2. 一定要以空值 (null) 指標參數測試該介面。
  3. 必須設計能故意造成該元件執行失敗的測試案例。
  4. 在訊息傳遞系統中使用壓力測試 (stress testing)。
  5. 假如有許多元件是透過共用記憶體來溝通，請設計出讓這些元件以不同順序執行的測試案例。

8.1.4 系統測試

- 開發期間的系統測試，包括整合元件建立出系統的一個版本，接著測試整合後的系統。
- 系統測試要檢查這些元件是否相容、互動是否正確，以及是否在正確時間透過它們的介面傳輸正確的資料。它顯然與元件測試有重疊，不過有兩個重要的差異：
  1. 在系統測試期間，已經分開測試過的再利用元件和外購的現成系統，會與新開發的系統先整合，然後再測試完整的系統。
  2. 由不同團隊成員或子團隊開發的元件，可能會在這個階段整合。系統測試是集體而非個別的程序。有些公司可能另外編制一個測試團隊來做系統測試，設計人員和程式開發人員都不參與其中。

8.1.4 系統測試

- 由於系統測試的焦點是在互動行為，因此使用案例測試應該是最有效的方法。因為每個使用案例都是由數個元件或物件實作而成，測試使用案例會促使這些互動行為真正發生。假如有使用案例的序列圖，就可以看出牽涉在內的物件或元件。
- 以下我使用氣象站系統範例來說明。

8.1.4 系統測試

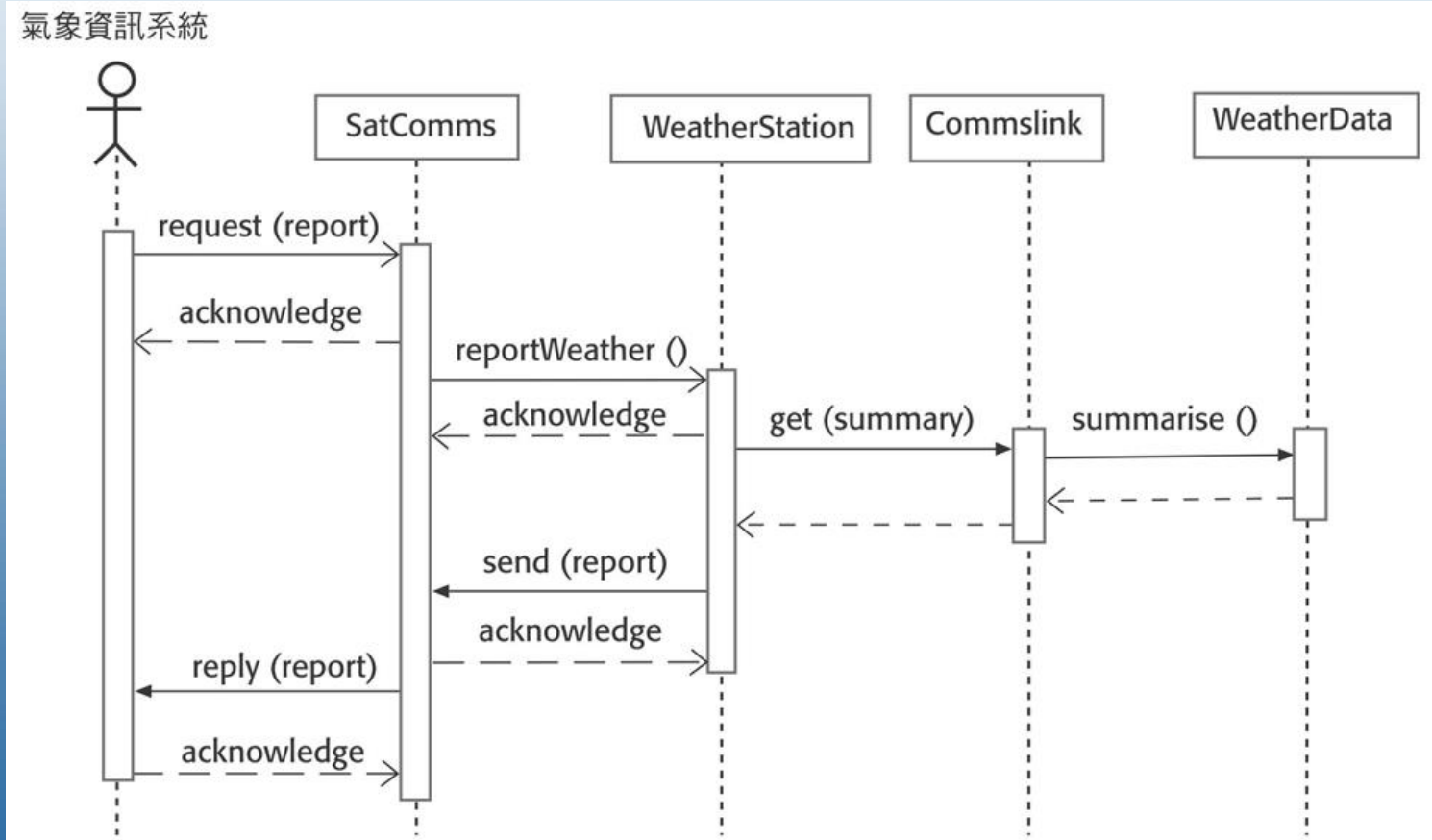

圖 8.8 收集氣象資料的序列圖

8.1.4 系統測試

- 序列圖可協助找出在設計測試案例時，所需要的輸入和即將建立的輸出：
  1. 某個請求報表 (report) 的輸入應該要有相關的回應 (acknowledgement)，報表最後也要傳回給該請求。在測試期間，應該建立可用來檢查報表是否正確編排的彙整資料。
  2. 對WeatherStation請求報表的輸入請求，結果應該會產生彙整的報表。你可以分開進行測試，先建立與彙整對應的原始資料，當作測試SatComms時的準備資料，然後再檢查WeatherStation物件是否能正確產生這份彙整報表。這個原始資料也可用來測試WeatherData物件。

8.1.4 系統測試

- 全面徹底的測試是不可能做到的，因為無法測試所有可能的程式執行順序，因此測試必須以一部分可能發生的測試案例為主。
  1. 所有能透過功能表存取的系統功能都應該測試到。
  2. 所有能透過相同功能表存取的功能組合，都應該測試到(例如設定文字格式)。
  3. 所有需要提供使用者輸入資料的功能，都必須測試正確與不正確的輸入資料。

8.2 測試驅動式的開發方式

- 測試驅動式開發 (test-driven development, TDD) 是一種程式開發方法，它是把測試和程式碼開發輪流交錯進行 (Beck 2002; Jeffries and Melnik 2007)。
- 程式碼是以增量式方式來開發，伴隨著開發該增量模組的測試，除非程式碼通過這個測試，否則就不能前進開發下一個增量模組。測試驅動式開發一開始是XP敏捷式開發方法的一部分，無論是敏捷式或計畫驅動式程序都可能使用它。

8.2 測試驅動式的開發方式

- 程序中的步驟如下：

1. 一開始先找出需要的增量模組功能。這通常比較小，可能幾行程式碼就可以實作出來。
2. 針對此功能撰寫測試並實作成為自動化測試。
3. 接著執行此測試，還有之前已經設計好的所有其他測試。一開始因為還沒有實作此功能，所以新測試會失敗。
4. 接下來實作此功能並重新執行測試。
5. 等到所有測試都執行成功，就可以前進實作下一個模組的功能。

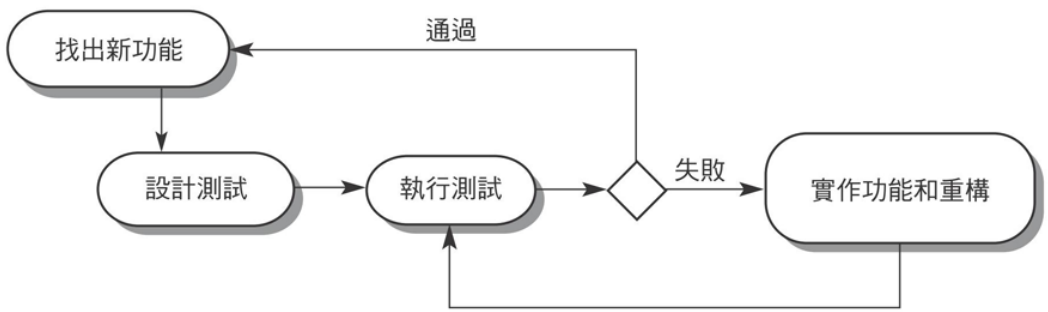

圖 8.9　測試驅動式開發

8.2 測試驅動式的開發方式

- 測試驅動式開發還有其他優點：
  1. **程式碼涵蓋**：每段程式碼都應該至少有一個相關聯的測試，因此對系統的所有程式碼都有確實被執行過會較有信心。
  2. **迴歸測試**：隨著程式開發，測試組也跟著開發。過程中一直都會執行迴歸測試 (regression test)。
  3. **簡化除錯**：當測試失敗，顯然是新寫的程式碼有問題，不必使用除錯工具也知道問題在哪裡。
  4. **系統文件**：測試本身就可以當作一種說明文件，描述程式碼應該做什麼。

8.2 測試驅動式的開發方式

- 迴歸測試是查驗確定這些修改沒有引進新的程式錯誤到系統中，而且新程式碼與舊有程式碼的互動也符合預期。
- 人工來進行迴歸測試，無論是時間和人力成本都很高。

8.3 發行版本測試

- 發行版本測試 (release testing) 是指針對某個將要給開發團隊以外的人使用的發行版本進行測試。
- 發行版本測試與開發程序期間的系統測試有兩個重要的區別：
  1. 系統開發團隊不應該負責發行版本測試。
  2. 發行版本測試是一個確認系統有符合需求，而且夠好可以讓客戶使用的程序 (確認測試)；而開發團隊的系統測試主要是找出系統的錯誤 (缺陷測試)。

8.3 發行版本測試

- 發行版本測試程序的主要目標，是讓系統出品商確認系統的確夠好能使用，如果是，它就可以當作產品發行或交付給客戶。
- 因此發行版本測試必須證明系統有達成指定的功能、執行效能和可信賴度，而且在正常使用下不會故障。
- 發行版本測試通常是一種黑箱 (black box) 測試程序，其測試是以系統規格為依據。

8.3.1 以需求為主的測試

- 以需求為主 (requirements-based) 的測試是一種確認測試而不是缺陷測試，因為目的是要證明系統的實作有適當的符合它的需求。
- 以下列Mentcare系統的需求為例，它是關於檢查用藥過敏的情況：
  - 假如病患已知對特定藥物會過敏，則在輸入含有該藥物的處方時，系統應該會送出警告訊息給使用者。
  - 假如醫師選擇忽略過敏警告，應該要輸入原因。

8.3.1 以需求為主的測試

- 可能需要開發一些相關的測試：
  1. 設計一筆沒有已知藥物過敏的病歷紀錄。開出已知有藥物過敏症存在的處方，檢查系統是否沒有發出警告訊息。
  2. 設計一筆有已知藥物過敏的病歷紀錄。開出已知病患會藥物過敏的處方，檢查系統是否有發出警告訊息。
  3. 設計一筆已知有兩種或多種藥物過敏的病歷紀錄。在處方中分別開出這幾種藥物，檢查系統是否有對這幾種藥物都發出正確的警告訊息。
  4. 在處方中同時開出兩種會引起病患過敏的藥物，檢查系統是否正確發出兩個警告訊息。

8.3.1 以需求為主的測試

5. 在處方中開出藥物讓系統發出警告訊息，並且駁回 (overrule) 這個警告訊息。請檢查系統是否有要求使用者必須提供資訊，解釋為什麼要駁回這個警告訊息。

8.3.2 情境式測試

- 情境測試 (scenario testing) 是一種發行版本測試，做法是設計典型的使用情境，並用它來開發系統的測試案例。

George 是一位心理保健護士，他的工作之一是到病患家中拜訪，確認治療情況和是否有藥物副作用。

在有安排家訪行程的工作日，George 會先登入 Mentcare 系統，並使用它列印出當天的探視時程，還有每位探視病患的摘要資訊。他會把當天病患的病歷下載到他的筆記型電腦，系統還會提示他輸入密碼來加密筆電上的病歷資料。

他今天要拜訪的其中一位病患是 Jim，他正接受憂鬱症的藥物治療。Jim 覺得藥物有幫助，但是它的副作用會讓他晚上睡不著。George 想看一下 Jim 的病歷，系統會提示他輸入密碼來解密才看得到。他查看目前的處方並查詢藥物的副作用，發現失眠是已知的副作用，因此他在 Jim 病歷上記下這個問題，並建議他來一趟診所調整藥物。Jim 同意了，因此 George 輸入一筆提醒打電話的紀錄，這樣當他回診所就會與醫師預約時間。最後他結束探訪而系統再度加密 Jim 的病歷。

最後當 George 回到診所，他將病患的探訪紀錄上傳到資料庫。系統則產生一份提醒打電話的清單給 George，清單上是需要他與醫師約診後再聯絡的病患。

圖 8.10 Mentcare 系統的使用情境之一

8.3.2 情境式測試

- 圖8.10是取材自Mentcare系統的一個可能的情境，描述一位家訪護士如何使用此系統。這個情境會測試Mentcare系統的好幾個功能：

1. 通過驗證手續才能登入系統
2. 下載病歷紀錄到筆電和上傳筆電的病歷紀錄
3. 安排家庭探視行程
4. 將行動裝置上的病歷紀錄加密和解密
5. 擷取和修改病歷紀錄
6. 連線到維護藥物副作用資訊的資料庫
7. 系統紀錄要提醒打電話給病患

8.3.3 效能測試

- 系統被整合完成之後，便可以針對系統的外顯性質進行測試，如執行效能和可靠性。
- 這通常會包括規劃一系列的測試，讓系統的負載穩定遞增，直到系統的執行效能慢到令人無法接受為止。
- 假如是效能測試，代表要給系統壓力，例如送入大量的要求，故意超出軟體的設計限制，並觀察系統的執行情況。因此這也叫做壓力測試 (stress testing)。

8.4 使用者測試

- 所謂的使用者測試 (user testing) 或客戶測試 (customer testing)，指的是在測試程序中，由使用者或客戶提供輸入，並提出系統測試意見的階段。
- 無論如何，即使有做過詳盡的系統和發行版本測試，使用者測試仍是絕對必要的。
- 理由是使用者的工作環境對於系統的可靠性、效能、易用性和堅固性有很大的影響。
- 實際上系統開發人員不可能完全複製系統的工作環境。

8.4 使用者測試

- 使用者測試分成3種不同的類型：
  1. **Alpha測試 (Alpha testing)**：經選擇的軟體使用者與開發團隊在一起工作，測試軟體的早期發行版本。
  2. **Beta測試 (Beta testing)**：提供一個發行版本，能讓人數較多的使用者群體驗，並向系統開發人員回報發現的問題。
  3. **驗收測試 (acceptance testing)**：由客戶測試系統決定要不要驗收通過並部署在客戶環境中。

8.4 使用者測試

- 驗收測試程序有6個階段如下：
  1. 定義驗收條件
  2. 規劃驗收測試
  3. 設計驗收測試
  4. 執行驗收測試
  5. 協商測試結果
  6. 接受或拒絕系統

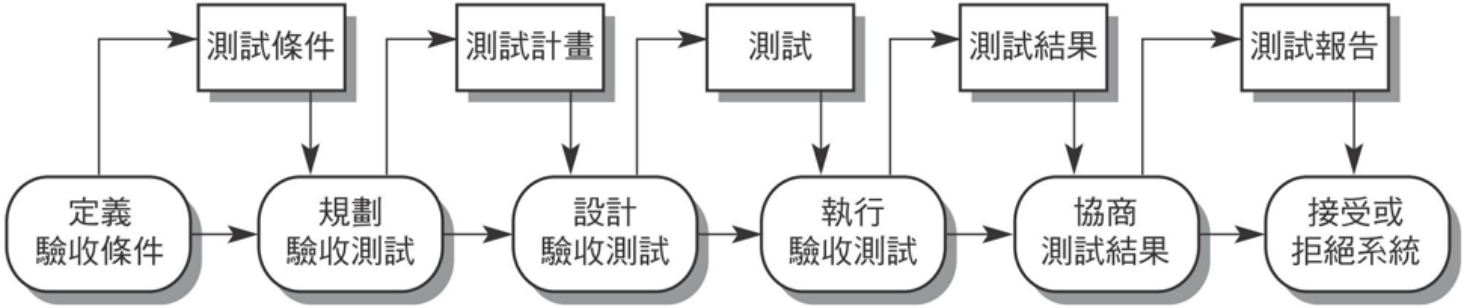

圖 8.11　驗收測試的程序

8.4 使用者測試

- 如果是像XP(極致程式設計)這樣的敏捷式方法，可能不會有另外分開的驗收測試活動。
- 要融入軟體開發團隊的使用者最好是「典型的」使用者，也就是對將來系統會如何使用有大概的瞭解。不過要找到這樣的使用者可能不容易，因而導致驗收測試未必能真正反映實務上系統將如何使用。