對本書的讚譽

這本書非常特別，它的各章互相銜接，積累出驚人的深度。準備好享受一場盛宴吧！

— 來自第二版前言，Robert C. Martin，cleancoder.com

本書是這個領域的經典，是學習單元測試的最佳途徑。

— Raphael Faria，LG Electronics

本書教導高效率單元測試的測試哲學及實作細節。

— Pradeep Chellappan，Microsoft

當我的團隊問我如何寫出正確的單元測試時，我會直接回答：看這本書！

— Alessandro Campeis，Vimar SpA

單元測試的最佳資源。

— Kaleb Pederson，Next IT Corporation

這是我讀過的單元測試指南中，最實用且最符合現況的一本。

— Francesco Goggi，FIAT

對想要認真學習或完善單元測試知識的 .NET 開發者而言，這是必讀之書。

— Karl Metivier，Desjardins Security Financial

本書簡介

關於學習這檔事，我聽過的最有智慧的說法之一是：「若要真正學會一件事，那就去教導它」（很遺憾我忘了是誰說的）。對我而言，於 2009 年撰寫這本書的第一版並出版它，無疑是一段真正的學習經歷。我寫這本書的初衷，是因為我對一次又一次回答相同的問題感到厭煩，但也有其他原因。我想嘗試一些新東西，想進行一項試驗，也想知道寫一本書（任何書）能夠讓我學到什麼。單元測試是我自認擅長的事情。但是，經驗越多，就越覺得自己知識淺薄。

現在回過頭看，第一版中的某些部分我已經無法認同，例如，**單元**是指「方法」，這大錯特錯。我在這本第三版的第 1 章中提到，單元是工作單元，它可能小至一個方法，也可能大至幾個類別（甚至可能是幾個編譯單元，assemblies），此外，我也改了其他的東西，等一下你就會知道。

第三版的新內容

在這本第三版裡，我們將 .NET 改成 JavaScript 和 TypeScript。我們當然也更新了所有相關的工具和框架，例如，我們用 Jest 來替代 NUnit 測試執行器和 NSubstitute，將它當成單元測試框架以及 mock 庫。

我們在探討「在組織層級上實踐單元測試」的那一章裡加入更多技術。

本書的程式在設計上有許多改變，主要與使用 JavaScript 這類的動態定型語言有關，但我們也利用 TypeScript 來討論靜態定型技術。

我們將測試的可信度、易維護性，和易讀性的探討擴展成獨立的三章，並加入關於測試策略的新章節，討論如何在不同的測試類型之間做出選擇，以及該使用哪些技術。

適合的讀者

本書適合需要撰寫程式並對學習單元測試的最佳實踐法有興趣的人。書中的範例都是用 JavaScript 和 TypeScript 來編寫的，因此 JavaScript 開發者會認為這些範例特別實用，但是我們教導的內容同樣適用於大多數的物件導向和靜態定型語言（例如 C#、VB.NET、Java 和 C++…等諸多語言）。無論你是架構師、開發者、團隊領導者、需要寫程式的 QA 工程師，還是剛學習程式設計的新人，這本書都適合你。

本書結構：路線圖

如果你從未寫過單元測試，最好從頭到尾閱讀這本書，以全面瞭解這項技術。如果你有經驗，你可以根據需要，自由地選擇章節閱讀。本書分為四部分。

第一部分帶你從零開始撰寫單元測試，直到熟練為止。第 1 章和第 2 章介紹基本知識，例如如何使用測試框架（Jest），並介紹自動測試概念，例如測試庫、斷言庫，和測試執行器。這部分也介紹斷言、忽略測試、工作單元測試、單元測試的三種最終結果，以及你需要做的三類測試：值測試、基於狀態的測試，和互動測試。

第二部分討論斷開依賴關係的進階技術：mock 物件、stub、分隔框架，以及重構程式碼來使用那些技術的模式。第 3 章介紹 stub 的概念，並展示如何手動建立和使用它們。第 4 章教你使用 mock 物件來進行互動測試。第 5 章將這兩個概念結合起來，展示分隔框架如何結合這兩個概念，並將它們自動化。第 6 章深入探討如何測試非同步程式碼。

第三部分討論如何組織測試程式碼、運行和重構測試程式碼結構的模式，以及編寫測試的最佳實踐法。第 7 章討論撰寫可信測試的技術。第 8 章介紹建立容易維護的單元測試的最佳實踐法。

第四部分討論如何在組織中進行變革，以及如何處理現有程式碼。第 9 章探討測試的易讀性。第 10 章展示如何擬定測試策略。第 11 章討論在組織中引入單元測試時，可能遇到的問題和解決方案，並回答在過程中你可能被問到的一些問題。第 12 章說明如何在遺留碼中引入單元測試，並提出幾種「決定從何處開始測試」的方法，以及討論一些測試「無法測試的程式碼（untestable code）」的工具。

> **注意** 關於測試驅動開發的精確含義有許多不同的觀點。有人說它是測試優先的開發，有人則認為它意味著你會有很多測試。有人說這是一種設計方式，有人則認為它是僅用一些設計來驅動程式碼行為的方法。在本書中，TDD 是指先寫出測試的開發方法，「設計」在這項技術中發揮漸進的功用（除了本節之外，本書不再討論 TDD）。

圖 1.8 和 1.9 是傳統的程式寫法和 TDD 之間的差異。TDD 與傳統開發不同，如圖 1.9 所示。你要先寫出一個失敗的測試，再編寫產品程式碼，讓那個測試通過，再進行程式碼重構，或編寫另一個失敗的測試。

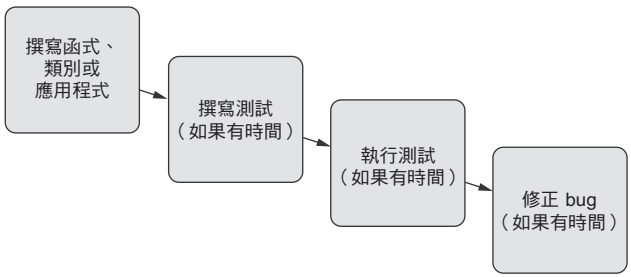

圖 1.8 傳統的單元測試寫法

本書的重點是寫出好的單元測試的技術，而不是 TDD，但我很喜歡 TDD。我曾經使用 TDD 寫出幾個重要的應用程式和框架，我也管理過善用 TDD 的團隊、教過數百門關於 TDD 和單元測試技術的課程和工作坊。我在職業生涯發現 TDD 有助於創造優質的程式、優質的測試，以及更好的設計。我相信 TDD 可以帶來好處，只是天下沒有白吃的午餐（你要付出學習時間、實作時間…等）。然而，如果你願意接受學習的挑戰，它絕對值得。

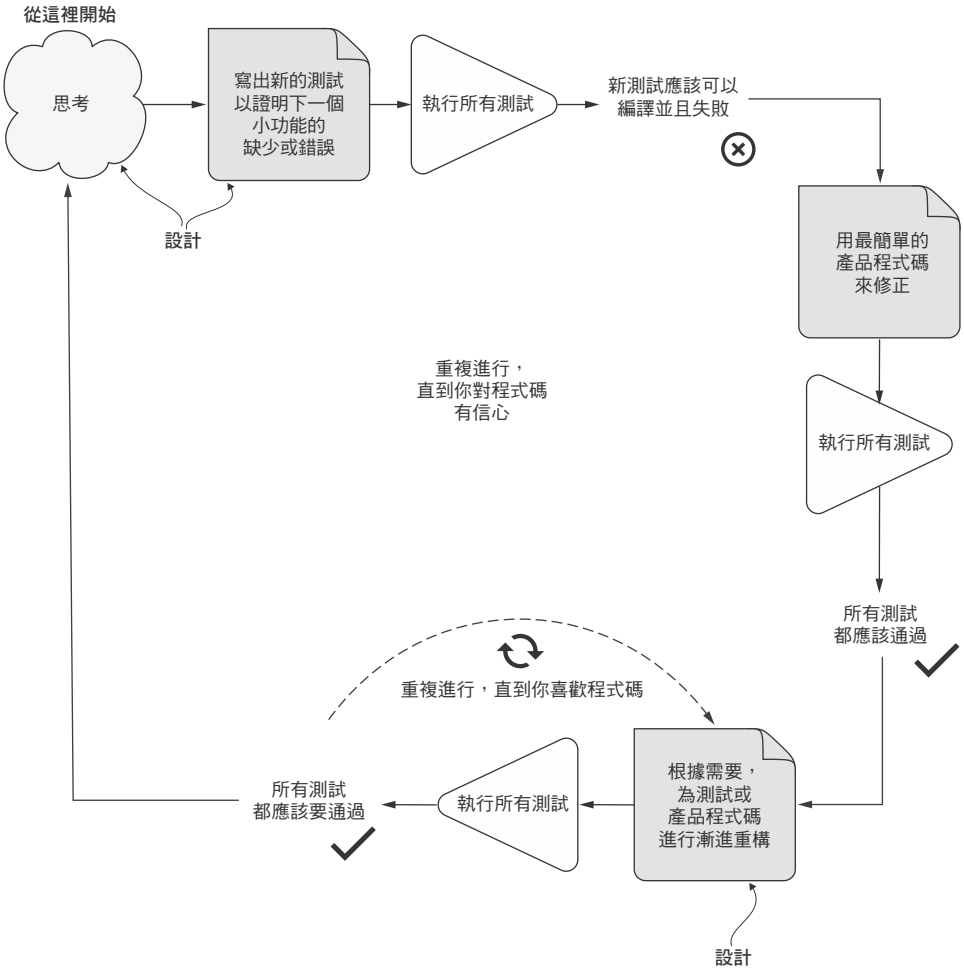

圖 1.9　測試驅動開發的全貌。注意這個過程的循環性：撰寫測試、撰寫程式碼、重構、撰寫下一個測試。它展示了 TDD 的漸進性質：用小步驟來產生可信且高品質的最終結果。

1.10.1 TDD：並非優良單元測試的替代品

切記，TDD 無法確保專案成功，也無法確保測試的穩健性或易維護性。我們很容易沉醉於 TDD 技術中，卻沒有注意單元測試的寫法，例如它們的名稱、易維護性，或易讀性，以及它們是否測試了正確的東西，或本身是不是有 bug。這就是我寫這本書的原因——因為把測試寫好是一種與 TDD 不一樣的技能。

TDD 的技巧非常簡單：

1. **寫一個失敗的測試來證明目前還缺少最終產品的程式碼或功能**。那個測試要寫得*彷彿*產品程式已經能夠正確運作一般，因此測試失敗意味著產品程式有 bug。我怎麼知道？因為測試被寫成當產品程式沒有 bug 時，它就會通過。

   在 JavaScript 之外的一些語言中，測試最初甚至可能無法編譯，因為程式碼還不存在。當它運行時，它就應該要失敗，因為產品程式仍然無法正確運作。這就是測試驅動設計思想中的許多「設計」發生之處。

2. **在產品程式中加入功能來滿足測試的期望，使測試通過**。產品程式應該盡量保持簡單。你不能修改測試，只能藉由修改產品程式來讓測試通過。

3. **重構程式碼**。當測試通過時，你可以繼續進行下一個單元測試，或重構程式碼（包括產品程式碼和測試程式），讓它更易讀、移除重複的程式碼…等。這是「設計」發生的另一處。我們進行重構，甚至可以重新設計組件，同時保留舊功能。

   重構應該小幅度、逐步推進，並且在每一個小步驟後運行所有測試，以確保修改沒有引起任何問題。你可以在編寫多個測試後或編寫每一個測試後進行重構。這是很重要的方法，因為它可以確保程式碼容易閱讀和維護，同時讓以前寫好的所有測試仍然通過。本書將用完整的一節（8.3）來討論重構。

> **定義**　重構是指修改程式碼而*不*改變其功能。如果你曾經修改方法的名稱，那就是重構了。如果你曾經將一個大方法分成多個較小的方法呼叫，你就重構過程式碼了。程式碼仍然做同一件事，但變得更容易維護、閱讀、偵錯和修改。

上述步驟聽起來或許很具技術性，但裡面蘊含著許多智慧。如果做得對，TDD 可以提升程式碼的品質、減少 bug、加強你對程式碼的信心、縮短尋找 bug 的時間、改善程式碼的設計，還有，讓你的主管更滿意。但如果做

3 使用 stub 來切斷依賴關係

本章內容

- 依賴項目的類型——mock、stub…等
- 使用 stub 的原因
- 函式注入技術
- 模組注入技術
- 物件導向注入技術

在上一章，我們使用 Jest 來編寫了第一個單元測試，並且更深入地探討測試本身的易維護性。當時的情境非常簡單，更重要的是，它是完全自成一體的。Password Verifier 不依賴外部模組，所以我們可以專心研究它的功能，不必擔心可能會干擾它的其他因素。

在上一章，我們使用前兩種退出點類型來作為範例：回傳值的退出點，和改變狀態的退出點。在這一章，我們將討論最後一種類型：**呼叫第三方**。我們也會介紹一個新需求：讓程式碼依賴時間。我們將探討處理這個問題的兩種方法——重構程式碼，和不進行重構的 monkey-patching（猴補丁）。

依賴外部模組或函式會讓測試難以編寫，讓測試更難重複使用，也可能導致測試不穩定。

我們在程式碼裡面依賴的外部事物稱為**依賴項目**（*dependency*）。在本章稍後，我會更詳細地定義它們。那些依賴項目可能是時間、非同步執行、使用檔案系統或網路，或只是使用某些難以設置，或可能需要花費大量時間來執行的東西。

3.1 依賴項目的類型

根據我的經驗，我們的工作單元可能使用兩種主要的依賴項目：

- **外出依賴項目**（*outgoing dependency*）——本身是工作單元**退出點**的依賴項目，例如呼叫 logger、將某些資料存入資料庫、發送 email、通知 API 或 webhook 有事情發生了…等。注意它們都有**動詞**：「呼叫」、「發送」和「通知」。它們從工作單元**往外**離開，類似一種射後不理的情境。每一個外出依賴項目都代表一個退出點，或工作單元裡的某個特定邏輯流程的結束。

- **入內依賴項目**（*incoming dependency*）——非退出點的依賴項目。它們不代表工作單元的最終行為的需求，只負責提供測試專用資料或行為給工作單元，例如資料庫查詢的結果、檔案系統裡的檔案的內容、網路回應…等。注意它們都是被動的資料，都是之前的操作結果，且**進入**工作單元。

圖 3.1 展示這兩種依賴項目。

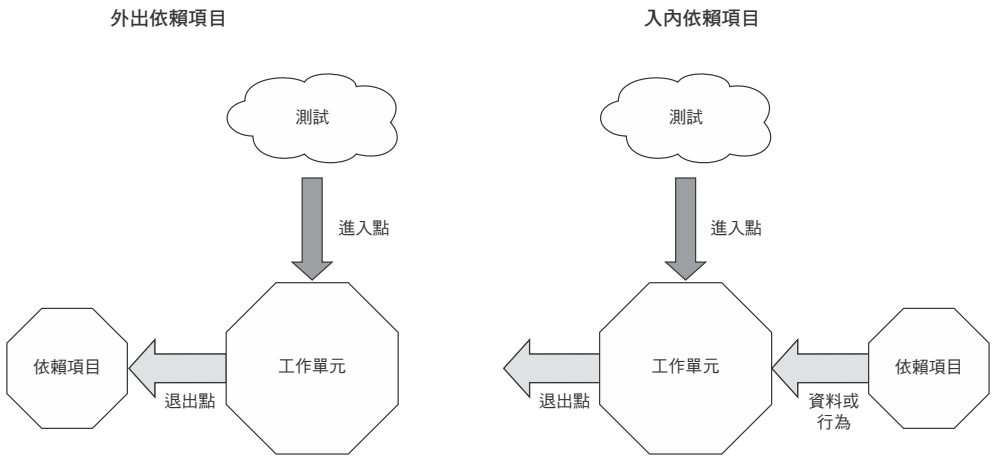

圖 3.1 在左圖，退出點被實作成呼叫一個依賴項目。在右圖，依賴項目提供間接的輸入或行為，且不是退出點。

以上所言是否意味著不該撰寫整合測試？不，我認為你絕對應該使用整合測試，只是不需要透過許多整合測試來加強你對程式碼的信心。整合測試未涵蓋的部分應該用低階測試來涵蓋，例如單元測試、API 測試或組件測試。我會在專門討論測試策略的第 10 章詳細討論這種測試策略。

6.2 讓程式更適合進行單元測試

如何使用單元測試來測試這些程式碼？接下來要告訴你一些讓程式碼更容易做單元測試的模式（即更容易注入，或避免依賴關係，以及檢查退出點）：

- *Extract Entry Point* 模式——將產品程式碼的純邏輯部分提取到它們自己的函式中，並將那些函式當成測試的進入點。
- *Extract Adapter* 模式——將本質上是非同步的東西提取出來並抽象化，以便使用同步的東西來替換它。

6.2.1 Extract Entry Point

在這個模式中，我們將一個特定的非同步工作單元分成兩部分：

- 非同步部分（保持不變）。
- 在非同步程序執行完成時呼叫的 callback。我們將它們提取出來，做成新函式，最終成為純邏輯工作單元的進入點，我們可以用純單元測試來呼叫它們。

圖 6.2 說明這個概念：在之前的圖中，我們有一個包含非同步程式碼的工作單元，那些程式碼與處理非同步結果的內部邏輯混合在一起，並透過 callback 函式或 promise 機制來回傳結果。在第 1 步，我們將邏輯提取到它自己的一個（或多個）函式中，這些函式僅使用非同步工作的結果作為輸入。在第 2 步，我們將這些函式外部化，這樣就可以將它們當成單元測試的進入點來使用。

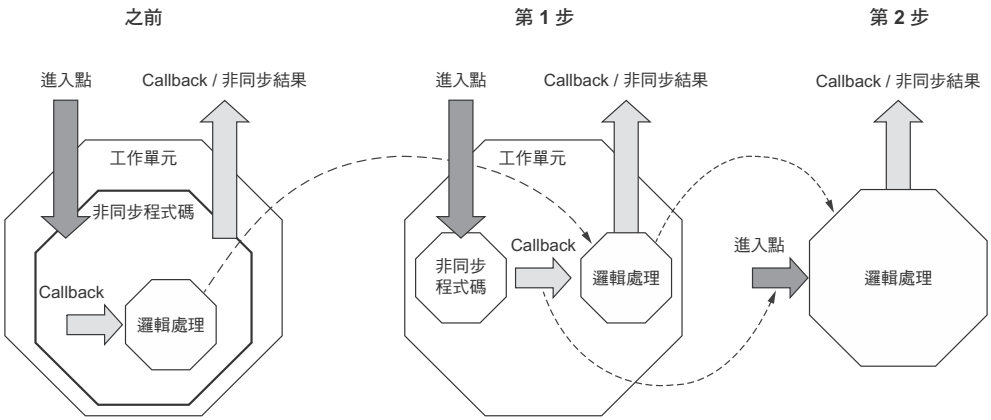

圖 6.2 將內部處理邏輯提取為一個單獨的工作單元有助於簡化測試，因為我們能夠同步驗證新的工作單元，而不涉及外部依賴項目。

這讓我們具備了測試非同步 callback 邏輯處理的關鍵能力（並且可以輕鬆模擬輸入）。同時，我們可以針對原始工作單元編寫高階的整合測試，以確保非同步協同工作（orchestration）也能夠正確運作。

如果我們僅為所有情境進行整合測試，最終會產生許多執行時間冗長且不穩定的測試。在新世界中，我們可以讓大多數的測試變得快速且一致，並在最上面加入一小層整合測試，以確保所有的協同工作都能正常運作。如此一來，我們就不會為了獲得信心，而犧牲速度和易維護性。

提取工作單元的範例

我們來將這個模式應用到範例 6.1 中的程式碼。圖 6.3 是我們將遵循的步驟：

❶ 之前狀態包含 `isWebsiteAlive()` 函式內的處理邏輯。

❷ 提取發生於「提取結果邊緣」的任何邏輯程式碼，並將它們放入兩個獨立的函式中：一個用來處理成功情況，另一個用來處理錯誤情況。

❸ 然後，我們將這兩個函式外部化，以便直接在單元測試中呼叫它們。

範例 8.8 需要按照特殊順序來執行的測試

```
describe("Test Dependence", () => {
  describe("loginUser with loggedInUser", () => {
    test("no user, login fails", () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(false);
    });

    test("can only cache each user once", () => {
      getUserCache().addUser({
        key: "a",
        password: "abc",
      });

      expect(() =>
        getUserCache().addUser({
          key: "a",
          password: "abc",
        })
      ).toThrowError("already exists");
    });

    test("user exists, login succeeds", () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);
    });
  });
});
```

使用者快取必須是空的

將使用者加入快取

快取裡面必須有使用者

注意，第一個和第三個測試都依賴第二個測試。第一個測試依賴第二個測試尚未執行，因為它需要空的使用者快取。另一方面，第三個測試依賴第二個測試將預期使用者填入快取。如果我們使用 Jest 的 `test.only` 關鍵字只執行第三個測試，該測試將會失敗：

```
test.only("user exists, login succeeds", () => {
   const app = new SpecialApp();
   const result = app.loginUser("a", "abc");
   expect(result).toBe(true);
 });
```

這種反模式通常在你試圖重複使用測試的一部分，而沒有將它提取出來做成輔助函式的時候發生。你會期待另一個測試先執行，以便節省一些設置工作。這種方法在有效時看似可行，但終究會出問題。

我們可以用以下的步驟來重構它：

- 提取一個用來加入使用者的輔助函式。
- 在多個測試中重複使用這個函式。
- 在測試之間重設使用者快取。

下面的範例展示如何重構測試以避免這個問題。

範例 8.9 重構測試，以移除順序依賴關係

```
const addDefaultUser = () =>
  getUserCache().addUser({
    key: "a",
    password: "abc",
  });

const makeSpecialApp = () => new SpecialApp();

describe("Test Dependence v2", () => {
  beforeEach(() => getUserCache().reset());
  describe("user cache", () => {
    test("can only add cache use once", () => {
      addDefaultUser();

      expect(() => addDefaultUser())
        .toThrowError("already exists");
    });
  });

  describe("loginUser with loggedInUser", () => {
    test("user exists, login succeeds", () => {
      addDefaultUser();
      const app = makeSpecialApp();

      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);
    });

    test("user missing, login fails", () => {
      const app = makeSpecialApp();
```

提取出來的函式，用來建立使用者

提取出來的工廠函式

在不同的測試之間重設使用者快取

嵌套的新 describe 函式

呼叫可重複使用的輔助函式

10.2.3 低階測試和高階測試脫節

這種模式乍看之下可能很健康，但實際上並非如此。它看起來可能像圖 10.4。

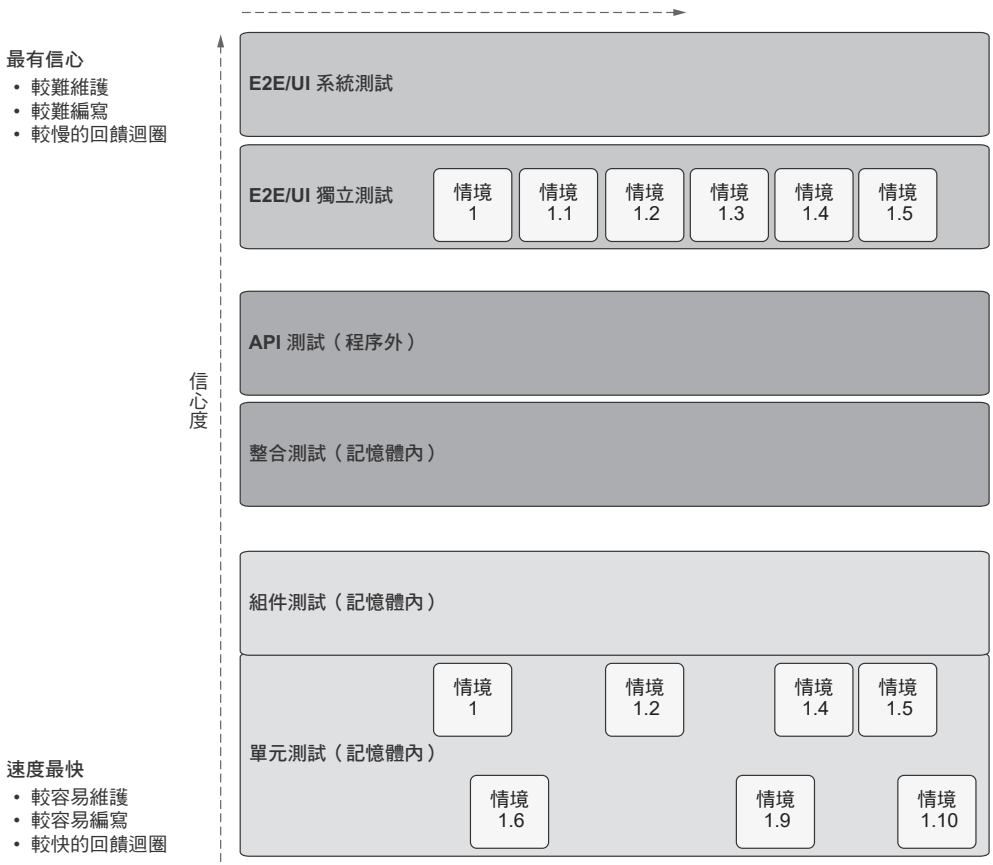

圖 10.4　低階測試和高階測試脫節

是的，你希望同時擁有低階測試（為了速度）和高階測試（為了信心）。但是當你在一個組織裡看到這種情況時，你可能也會遇到以下的一種或多種負面行為：

- 許多測試在多個階層中重複出現。

- 編寫低階測試的人與編寫高階測試的人不同。這意味著他們不在乎彼此的測試結果，可能用不同的管道來執行不同類型的測試。當一個管道是紅色的時，另一組人可能甚至不知道或不關心那些測試失敗。

- 我們遭遇雙輸的局面：在高階，我們有測試跑太久、難以維護、build whisperer 和不穩定的困擾；在低階，我們缺乏信心。而且由於缺乏溝通，我們無法從低階測試中獲得速度優勢，因為它們在高階測試中重複出現。我們也無法從高階測試中獲得信心，因為有大量的測試不穩定。

當我們有不同的測試和開發組織，而且那些開發組織有不同的目標和指標，以及不同的任務、管道、權限，甚至程式版本庫時，這種模式經常發生。公司規模越大，這種情況越有可能發生。

10.3 測試配方策略

為了讓組織使用的測試類型可以平衡，我提出使用**測試配方**這個策略。概念上，它就是為特定的功能建立非正式的測試計畫，這個計畫不但包含主要情境（也稱為**快樂路徑**（*happy path*）），也包含它的所有重要的變體（也稱為**邊緣情況**（*edge case*）），如圖 10.5 所示。清楚的測試配方可以明確地指出適合每一個情境的測試層。

10.3.1 如何撰寫測試配方

測試配方最好由兩個人一起擬定，其中一位從開發者的觀點，另一位從測試者的觀點。如果沒有測試部門，你可以指定兩位開發者，或一位開發者與一位資深開發者。將每一個情境對映到測試層是非常主觀的任務，因此安排兩對眼睛有助於互相檢查隱性的假設。

你可以將配方存成待辦事項清單裡的額外文字，或是任務追蹤板上的功能故事，不需要使用額外的工具來規劃測試。

建立測試配方的最佳時間是在開始製作一個功能之前。如此一來，測試配方就會成為該功能的「完成」定義的一部分，這意味著該功能在整個測試配方通過之前還不完整。