CD CH4

發凡

1. 手動 VS 自動，如何取捨？如何權衡？太多測試依賴手動，本章會分享有效的自動化測試策略以及操練 (practices)
2. Deming: 不要依賴大規模巡檢以成就品質，及早改善並且把品質建置進產品中。完整的自動化測試將測試每個程式更動，種種的測試如單元 (unit)、元件 (component)、驗收、手動、演示 (showcase)、可用性、探索性測試等等，都需要在專案中跟著開發持續改善
3. TDD: 越早越好，完整的測試覆蓋就是好的回歸測試基礎
4. 非功能性測試：容量、安全性，如同其他測試做CI，可以及早發現那些更動會導致效能問題或其他
5. 越早開始越好，專案中段才開始會有些慣性，不好追上，還是有些技巧可以讓舊有系統上軌道
6. 測試是信心的基礎：臭蟲少、降低維運成本、維持好名聲。更鼓勵好的操練，並隨時能提供最新文件以及技術規格
7. 只是九牛一毛，在此只舉梗概，更多細節請參考 Agile Testing, More Agile Testing 等書

測試種類 (四象限)

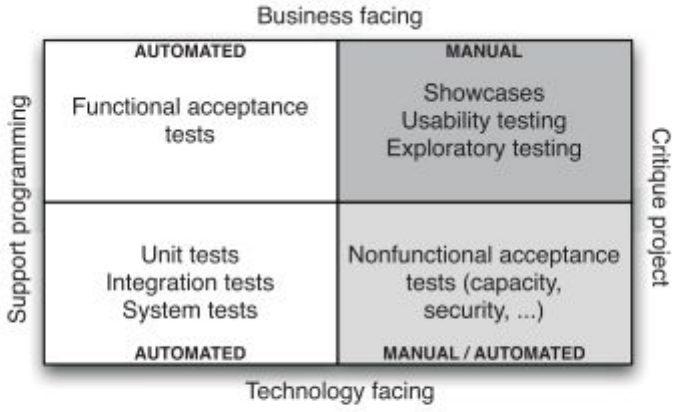

Figure 4.1 *Testing quadrant diagram, due to Brian Marick, based on ideas that were "in the air" at the time*

業務面向並支持開發流程測試

1. 功能性或驗收測試，確保驗收標準都達成，最好自動化並在開發前就做。驗收測試主要測試功能性、容量、可用性、安全性、可修改性、可用性等等，分為功能及非功能性(第四象限)
2. 在敏捷環境中，驗收測試舉足輕重，告訴開發者什麼時候做好了，並告訴使用者想要的功能已具備。Cucumber, JBehave, Concordian 與 Twist 等等的工具可以讓使用者寫(驗收)測試腳本，讓開發者及測試人員實作系統面的測試
3. 應用程式中的單一標準路徑：快樂路徑 (happy path)。Given, when, then：已知系統測試起始時狀態，當使用者做某些操作，然後系統會進入新狀態。

4. 有時候起始狀態會有一些異同，操作及結束狀態也會有些差異，這就是替代路徑 (alternate path)。應該出錯的，就是悲傷路徑 (sad path)。需要一些經驗做些分析才能分清這些路徑，並回饋做有效測試
5. 使用者驗收測試 (UAT)要在類上線環境中做，設置跟系統狀態都要盡量相同，外接服務就用 mock，自動化驗收測試也是要在類上線環境做

自動化驗收測試的好處

- 讓回饋迴路更快，開發者自己就可以知道東西壞掉沒，不需要找測試人員
- 降低測試人員的負擔
- 讓測試人員可以空出手腦來做探索性 (exploratory) 測試或是其他更高附加價值的活動
- 驗收測試就是最好的回歸測試，對大規模協作及複雜依存系統很有幫助
- BDD (behavior-driven development) 鼓勵分析師把需求寫成可執行的測試腳本，透過 Cucumber 或 Twist 做自動化，需求文件也可以一起進 CICD

探索性測試：有經驗有頭腦的測試人員，會一邊熟悉系統一邊學習，一邊設計測試，一邊執行測試，去探索自動化或舊測試未涉略之處女地，找到以前沒發現的 bug，補齊測試，或是發現一些未知或意料之外的行為，跟著開發一起迭代，更進一步了解真實需求或確認需求真實，是需要增減功能，這不是自動化能夠取代的測試，這需要創造性、洞察力及膽識，這不是black box testing，更不是ad-hoc testing，因為這一點都不sloppy，是很高技術含量的智慧行為，我們要找的 QA 就要具備或培養這樣的能力

6. 回歸測試 (regresson) 是橫跨四象限的，所有的自動化測試就是回歸測試，幫助你做重構 (refactoring/rearchitecting) 更有信心，確認系統行為沒破壞
7. 自動化測試維護起來可能昂貴，甚至有人反對大規模複雜的自動化測試，但依循一些好方法及操練，是可以好好權衡，發揮自動化的效益
8. 自動化不是一體適用，有些如可用性、畫面版位、探索性測試就無法自動化。不過自動化可以做準備工作：準備測試情境或是資料。一般來說自動化測試會涵蓋所有的快樂路徑 (happy path) 以及有限的其他重要部分。80% 以上的覆蓋率叫做完整 (comprehensive)，不過測試內容的品質更重要
9. 好的測試給你信心，只要測試通過，系統的行為就不會改變，能繼續作動正常，即使是整個部份抽換掉，或是重構
10. 何時該把測試自動化？經驗法則是如果測試已經重複了相當次數，並且有信心你不會花太多時間維護這些測試
11. 自動化的 cost 是什麼？是花在開發自動化的時間嗎？其實應該是這個自動化能找到的 bug (Cem Kaner) 為基準，如果自動化測試已經成熟，手動測試不過是自動的延伸，那麼自動化的代價就低；反之如果系統大翻，手動隨便就找到新 bug，自動化的代價就高
12. 驗收測試該直接打 UI 嗎?
    a. 驗收一般是端到端的測試，理想上應該直接對 UI 做測試
    b. 但是很多 UI 相關測試都跟 UI 綁得太緊，容易壞掉，一點點 UI 一動就壞，所以要權衡，到底是因為真的 bug 壞掉，還是只因為需求一動就壞了？
    c. 其實有很多方式解決這種問題：一個是在 UI 跟測試之間加抽象層界接，避免直接碰觸 UI；或是直接打 UI 下面的 API (UI 不應該綁到商業邏輯)，所以測試應該直接跟商業邏輯互動，第八章談更多
13. 最該自動化的部分就是快樂路徑，起碼每個需求就要有對應的快樂路徑驗收測試

14. 替代路徑跟悲傷路徑很難抉擇：如果系統穩定，就選替代路徑讓使用者的情境可以更完整被覆蓋；若是系統不穩定，就選悲傷路徑，儘快除錯

技術面向並支持開發流程的測試

1. 這些測試應該都是開發人員撰寫：單元、元件以及佈署測試。單元測試要簡單，速度要快，覆蓋率要高 (80%)
2. 太簡單的測試會放過元件間互動的問題，比如物件或是資料生命週期不同，管理不好就會出問題
3. 元件測試範圍較大，通常也執行較久，有時候也會被稱作整合測試 (本書為避免混淆叫元件測試)
4. 部署測試確認應用程式安裝無誤，設定無誤，界接無誤，並且活著

業務面向並批判專案的測試

1. 軟體專案應該是迭代的過程，其中使用者也淬鍊自己的需求分析，他們只是希望流程更正義更順暢
2. 演示也該迭代，而且是專案的脈動：我們是展示系統作動如期待給付錢的客戶。過程中可能會修正需求，或是使用者太喜歡了給太多意見，就要跟專案團隊討論該如何修整，以免貽誤時程或需求發散
3. 可用性測試要蒐集使用者與系統互動的資訊，確認系統有帶給使用者應有的價值，甚至是使用者自身在做需求分析時，也會太靠近問題而欠考慮可用性
4. Beta 測試：有些組織用金絲雀發布 (canary releasing)，會用 feature switch 跟 instrumentation 來觀察功能是否有符合期待決定去留

技術面向並批判專案的測試

1. 非功能性測試：常被低估誤解，想正名但是仍沿用因為大家都叫習慣了，因為業務面不是這麼重視，甚至也會被忽略，其實如容量或安全性等等的面向也很重要，應該及早定義好
2. 所用的工具及測試與功能性測試不同，也要跑比較久，並且要在特殊的環境還要具備特殊的知識才能跑，一般執行地比較不頻繁，並且會在流水線的後端
3. 隨工具演進，為了避免在上線前才發現效能問題，建議專案一開始就有這些測試的雛型，甚至在大型專案需要分配專門的時間與人力來做

測試替代 (test double)

1. 有時為了測試某個單元，會需要模擬其他部分，就是所謂的替代(替身)，之前有多個名詞如 mocks, stubs, dummies 等等，這些替代有不同的形態：
    - 傀儡 (dummy)：被傳遞來去，不會真的被用到，通常用來填 param list
    - 假 (fake) 物件：真有實作，不過通常取巧，不適合線上使用，如記憶體內的資料庫
    - 存根 (stub)：提供制式 IO，一般對該測試以外的部分不反應
    - 間諜 (spy)：紀錄如何被呼叫的資訊，如 email 服務紀錄多少信件被寄出
    - 模擬 (mock)：針對既定的需求提供 feed，有錯可以出 exception
2. 模擬 (mocks) 常被濫用，流於鎖定細節，所以很脆弱，應該要注意該單元如何與其他部分協同，第八章會更細究，還有其他參考文件 (Martin Fowler)

現實狀況與策略

新專案

1. 在起始階段，任何更動的代價都是低的，最好有一些基本的簡單測試，或是基礎建設，作為 CI 流程的預備，最好現在就開始把驗收測試自動化，因此會需要以下：
    - 選擇技術平台或測試工具
    - 建置簡單的自動化組建 (build)
    - 以 INVEST 原則定好 story (開發項目)：Independent (獨立)、Negotiable (可商榷)、Valuable (有價值)、Estimable (可預估)、Small (小單元) 以及 Testable (可測試)，都要有驗收標準
2. 據此就可以實踐嚴格的流程：
    - 客戶、分析師及測試人員定義驗收標準
    - 測試人員與開發者合作，據驗收標準把驗收測試自動化
    - 開發者開發功能以滿足驗收標準
    - 如果任何自動化測試失敗 (無論單元、元件或驗收測試)，開發者須優先修好
3. 越早開始越好，不然阻力很大，開發者很難買單，而且沒有測試支持功能，會陷入惡性循環
4. 大家都要和衷共濟，尤其是客戶，如果決定為了早點上線而犧牲品質，他們有權如此做，但是後果自負，必須告知客戶如此行的後果
5. 確保驗收標準具有業務價值，不要盲目把不好的驗收標準自動化，會導致測試很難維護，所以測試人員一開始就要參與需求撰寫
6. 如果有好的流程，開發者寫程式的方式也會被改變，若一開始就把驗收測試自動化，寫出來的程式會有比較好的封裝 (encapsulation)、清楚的意圖、各種考量也有清晰的區隔、並且代碼較能重複使用，的確是良性循環

專案中途

1. 常態：一開始都不會想做 CICD，求有，所以大家常常會發現我們會在缺乏資源的大型專案中頻繁開發，並且有交付時程壓力
2. 為最重要的最常用的使用案例做自動化測試，需要跟客戶討論需求，做好回歸測試保護核心功能，最起碼要涵蓋快樂路徑 (happy path)
3. 超乎常例地盡量涵蓋可能的互動，即使這麼做仍無法涵蓋某些細節異動，甚至某些操作會導致無效狀態，起碼能保證系統功能滿足業務的核心價值
4. 因為只自動化快樂路徑，所以仍需大量手動測試，若某些手動測試常做，而其涵蓋的功能不會經常異動，就把它自動化；反之可能要先註解掉 (一定要清楚註明為何註解掉，日後再回來打開)，或者乾脆刪掉，反正可以救回來
5. 有時程壓力時，就無法測試複雜的互動或情境，所以就用最簡單的測試腳本，儘可能跑不同的測試資料，涵蓋不同的系統狀態，12 章會有更詳細討論如何載入並管理這些測試資料

舊有系統

1. 簡單 (又有點爭議) 的定義是：沒有自動化測試的系統。所以經驗法則就是：一定要測試你的更動

2. 先弄自動化建置，再加自動化功能測試，如果有文件或開發過的同事仍在可以請教最好，通常現況不會這麼好
3. 客戶或贊助者會覺得這不是已經 QA 測過？都上線了還測什麼？不過最重要的核心功能還是要回歸測試保護，可以很容易跟他們說明
4. 跟客戶訪談，一定要把最重要的高價值功能讓測試涵蓋，以此為基礎再迭加，可以當作舊有系統的冒煙測試 (smoke test)
5. 一但有冒煙測試，就可以開發新功能，用層疊的方式加測試，第一層一定要簡單又快，確保現有功能沒壞，第二層就保護核心功能，儘可能讓新功能如之前提到的新專案一般做好自動化測試，當作是驗收標準
6. 做起來比說的難，能測試的系統一般要具備模組化並易測試的特徵，一般都沒有，所以很脆弱，容易相互影響，因此最好確認程式在有效狀態，有時間的話涵蓋替代路徑，並測試錯誤路徑，確保沒有意外
7. 確保有交付價值的功能有自動化測試涵蓋，而非底層架構，為底層架構測試常常會出錯，而且為其搭鷹架曠日廢時，除非你的系統要在不同的環境上運作，這時候搭配自動佈署到類上線環境會很有幫助，節省很多手工

整合測試

1. 常常跟元件測試搞混：整合測試多用來確認外部的依賴系統協同無誤，或是系統裡有多個散裝元件，其間需要複雜互動者，需要做整合測試：確認依存系統協同無誤
2. 跟驗收測試類似，直接打外部系統，或是打備援系統，或是打測試框架 (harness)
3. 最好不要真打外部系統 (除非真的上線系統)，或是可以用假交易(為測試故)，一般有兩種方法：
    - 用防火牆將外部系統隔離，這樣一來也可以測試沒有外部系統會如何
    - 在系統中有組態可以設定讓系統跟模擬的外部系統協同
4. 理想中最好有備援系統以利測試，不過實際上，可能還是要開發測試框架，當：
    - 外部系統在開發中，介面定好了，但是有可能異動
    - 外部系統已建置好，不過沒有測試實體，或是太慢無法負荷自動化測試
    - 測試系統已建置好，但回應無法預測，所以測試很難有效 (如股市即時動態)
    - 外部系統是另一個應用程式，很難安裝，甚至要透過 UI 手動設置
    - 需要為外部系統做功能或驗收測試，這應該用測試替代
    - 測試環境太輕量化，無法負荷 CI 系統持續測試，只能用作手動或探索性測試用途
5. 測試框架可以很複雜，如果需要記憶系統狀態的話，如此只能做黑箱測試，模擬外部系統互動，包含例外狀況 (遠端或基礎設施) 如：
    - 網路傳輸問題
    - 網路通訊協定問題
    - 應用通訊協定問題
    - 應用邏輯問題
6. 盡量測試系統的病理性，確保可以處理越多意外問題越好，如應用斷路器 (circuit breaker) 或隔離 (bulkheads)
7. 自動化整合測試也可用作佈署的冒煙測試
8. 上線計畫也要考慮外部系統整合，這些外部系統會增加風險：
    - 測試服務可用嗎？效能好嗎？
    - 這些服務提供者可以提供諮詢服務、檢修或是增加客製化功能嗎？
    - 我們可以取得這些系統的上線版來診斷容量或可用性問題嗎？
    - 這些 API 在我的開發語言或技術中可以容易整合嗎？或是需要專業技術？

    - 我們必須自己寫或是自己維護測試服務嗎？
    - 如果外部系統出狀況，我的系統會怎麼樣？
9. 此外，還需要考量整合層的工作以及運行時的組態，以及測試服務並測試策略，比如在做容量測試時

流程

1. 驗收測試可能曠日廢時，尤其當溝通不是那麼有效率的時候 (鉅細靡遺，還要送客戶審查等等)
2. 其實流程可以優化：
    - 在開發開始前跟利害關係人開一次會，找齊客戶、分析師以及測試人員，決定最重要的測試情境
    - Cucumber, JBehave, Concordion, and Twist 可以協助我們用自然語言把驗收標準寫下來，日後再實作
    - 或用 DSL (領域特定語言) 記下驗收標準
    - 先有最簡單的雛型，日後再迭加
3. 開發者跟測試人員要儘早緊密合作，加緊回饋循環，這樣的成品品質好，臭蟲少
4. 不要整個開發項目 (story) 做完才測試，做一點測一點，開發者做完了一點，馬上把機器交給測試人員做測試，開發者用另一台終端機修之前測試發現的問題，近距離合作，隨時協同

管理積存缺陷 (defect backlog)

1. 若用 TDD，當然理想中不會有 bug，而且自動化測試應該會及早抓到很多，不過探索性測試、演示或使用性測試可能還是會發現缺陷，就會積存 (backlog)
2. 當然有的派別認為 bug 隨時發現就即時修正，就不會積存
3. 如果已經有積存缺陷，只看組建 (build) 成功或失敗是沒有幫助的，最好畫個餅圖強調現在的問題，讓大家聚焦
4. 通常會每況愈下，為方便故會拖延，所以積壓缺陷，甚至功能已變更 (invalid)，或是客戶認為重要的問題被淹沒
5. 如果在另一個分支上開發又沒定期 merge 回來問題就會更嚴重，大家吵成一團，流程就壞了，14 章進階版本控管有詳細討論
6. 另一種方法是把缺陷當作功能來管理，讓客戶自己排定優先順序，可以分為「關鍵」(critical)、「障礙」(blockers)、「中等」(medium)、「低」(low) 等四個類別，根據發生頻率、嚴重性、以及是否有暫解來區分

總結

1. 每個人都要有隨時測試、隨時整合的心態，才能打造高品質產品，不要把測試拖到最後，或貶低測試專業。要建立快速的回饋迴路提高品質、生產力、以及專案的可測量性
2. 自動化及手動測試互為臂助，在系統的每個面向都做，提供有效迅速的回饋，及早發現，及早修正，最不浪費時間
3. 把測試整合進每個交付的環節，才是做完了 (done)
4. 測試是交付價值的礎石，要確保測試在流程中隨處可見