前言

我從 11 歲開始就一直在以某種形式接觸了程式的編寫和設計，所以在我找到第一份軟體工程師的工作時，早已寫過很多程式碼了。儘管如此，我很快發現編寫程式碼和軟體工程是不一樣的。以軟體工程師的身份來設計編寫程式意味著寫出來的程式碼必須對其他人是有意義的，也不會在別人稍微修改之後就變得不能用。這也代表真的有人（有時是很多人）在使用和依賴我寫的程式碼，所以出錯的後果是很嚴重的。

隨著軟體工程師的經驗愈來愈豐富，他們會了解到日常編寫程式碼中所做出的決定是很重要的，對軟體是否能正常運作、持續能運作以及能否由其他人來維護等都會產生重大影響。學習如何寫出好的程式碼（從軟體工程的角度來看）可能需要很多年的累積，而這些技能通常是在工程師從自己的錯誤中吸取教訓慢慢累積起來的，或是從與他們一起工作的更高階工程師所得到零碎的建議來吸收獲取。

本書的目標在為新進軟體工程師提供掌握這些技能的快速入門指引。書中的內容講述了寫出可靠、可維護和能適應需求不斷變化的程式碼時，所需要知道的最重要課程和理論基礎，希望能幫助讀者從中獲益。

關於本書

《Good Code, Bad Code》這本書介紹了專業軟體工程師用來建構生成可靠和可維護程式碼的關鍵概念和技術。本書不僅列舉了該做和不該做的事，還解釋了每個概念和技術背後的核心原理及相關取捨，讓讀者對於如何像經驗豐富的軟體工程師一樣思考和編寫程式有基本的了解。

本書適用對象

本書的目標讀者是已經能編寫程式碼但希望以專業軟體工程師的角度來提升其技能的所有人。對已有 3 年以內經驗的軟體工程師來說，這本書還是很有用的參考資料。對經驗更豐富的工程師來說，也許已經知道書中所談的內容，但我希望這些內容仍是他們用來指導新手的有用資源。

本書的組織結構：學習路線圖

本書共有 11 章，分成為三個部分。第一部分介紹一些理論的高層次概念，這些概念形塑了我們思考設計程式碼的方式。第二部分轉向較實務的課程主題，這裡各章內容是由一系列主題劃分，涵蓋某些特定的考量因素或技術。本書的第三部分也是最後一部分，所談到的是建立有效且可維護之單元測試的原則和實務作法。

本書各個部分的結構模式是先示範可能有問題的場景（和一些程式碼），然後展示解開這些問題的替代方案。從意義上來說，內容編排是從展示什麼是「壞（Bad）」程式碼到展示什麼是「好（Good）」程式碼，但需要注意的是，「**壞**」和「**好**」這兩個詞是主觀的，取決於上下文脈的關聯。正如本書主旨所強調的那樣，通常是需要考量細微的差異和權衡取捨，這也表示這種區分並不是那麼明確。

Part 1「理論篇」，為總體大方向和稍微理論的思維奠定基礎，這些思維形塑了作為軟體工程師應該要具備的編寫程式碼方法。

- 第 1 章介紹了**程式碼品質**的概念，內容是想要達成高品質程式碼所要了解和實現的一組實用目標。然後把這些目標擴展成六個「程式碼品質的支柱」，這些支柱就是用於日常編寫程式碼的高階策略。
- 第 2 章討論**抽象層**，指導我們把程式碼建構和拆分為不同部分時的基礎考慮因素。
- 第 3 章重點在考量其他工程師的重要性，這些工程師會用到我們的程式碼。這裡會談到定立**程式碼契約**以及如何仔細思考以防範錯誤。
- 第 4 章討論錯誤的發生和仔細思考如何發出信號並進行處置，這些都是編寫出良好程式碼的重要基礎。

Part 2「實務篇」，透過具體的技術和範例以更實用的方式說明程式碼品質的前五個支柱（第 1 章有講述程式碼品質的六個支柱）。

- 第 5 章介紹了怎麼讓程式碼具有可讀性，好讓其他工程師也能夠理解我們所編寫的程式碼。
- 第 6 章介紹了避免意外的驚喜，以其他工程師不會誤解的程式碼寫法來盡量減少出現錯誤的機會。
- 第 7 章介紹了讓程式碼不會誤用的寫法，讓工程師不會意外產生邏輯錯誤或違反假設的程式碼寫法，能盡量減少錯誤的機會。
- 第 8 章介紹了讓程式碼模組化的關鍵技術，這樣有助於確保程式碼能展現出清晰的抽象層，並能夠適應不斷變化的需求。
- 第 9 章介紹如何讓程式碼可以重複使用和泛化通用。避免重新發明輪子，又能在添加新功能或建構新特性時更輕鬆也更安全。

Part 3「單元測試篇」，介紹了編寫有效單元測試的關鍵原則和實務作法。

- 第 10 章介紹了許多影響我們對程式碼進行單元測試的原則和更高層級的考量因素。
- 第 11 章以第 10 章的原則為基礎，為編寫單元測試的實務提供了許多具體且實用的建議。

閱讀本書的理想方式是依序從頭讀到尾，因為本書前面部分的思維觀念會為後續內容提供基礎。但話雖如此，Part 2（和第 11 章）中的各個主題都是獨立

的，每個主題講述的內容也不多，因此即使分開閱讀也沒問題，其內容也很實用。這樣的編排結構是經過深思熟慮的，目的是為了提供有效的方法可向其他工程師快速解釋說明既有的最佳實務作法。這對於任何希望在程式碼審查中解釋某個特定概念，或在指導其他工程師時解釋特定概念是非常有用的。

關於程式碼

本書的目標讀者是使用靜態型別、物件導向程式語言的工程師，例如以下的程式語言都適用：Java、C#、TypeScript、JavaScript（ECMAScript 2015 或帶有靜態型別檢查器的更高版本）、C++、Swift、Kotlin、Dart 2 … 等等。本書中所談到的概念在使用上述任何一種語言進行程式碼編寫時都能廣泛適用。

不同的程式語言有不同的語法和範式來表達邏輯和程式結構。為了能在本書中提供適切的程式碼範例，有必要對某種語法和範式進行標準化。因此本書使用了虛擬程式碼，這種程式碼借鑒了多種不同語言的思維。虛擬程式碼的目標能明確、清晰表達且易於被大多數工程師識別。請記住這種功用導向的意圖，本書無意暗示某種語言較好或較差。

同樣地，在明確和簡潔之間權衡取捨時，虛擬程式碼範例往往較偏向在明確方面。以實例來說，程式會使用明顯的變數型別，而不會用 var 之類的關鍵字來推斷型別。另一個例子是使用 if 語法來處理 null 值，而不是使用更簡潔（但可能不太熟悉）的 null 值合併和 null 值條件運算子（請參考附錄 B）。在真實的程式碼庫中（以及本書之外的程式），工程師可能更希望強調簡潔性。

如何使用本書中的建議

在閱讀任何關於軟體工程的書籍或文章時，請記住，這是個主觀的課題，現實世界中的問題解決方案很少有百分之百明確的。根據我的經驗，優秀的工程師大會以健康的懷疑態度來對待閱讀的所有內容，並希望了解其背後的基本思維。由於大家看法不同並不斷開展，可用的工具和程式語言也會不斷更新改進。了解特定建議背後的原因、上下脈絡和限制，這對於知道何時去使用以及何時應該忽略是十分重要的。

本書的目標在彙整各種有用的主題和技術，以幫助指導工程師編寫出更好的程式碼。雖然考慮使用這些內容是明智的作法，但書中的內容不應該直接認定是

絕對可靠的，也不應該視為永遠無法打破的硬性規定。良好的判斷力是好的軟體工程中應具備的基本能力。

延伸閱讀

本書的目標是成為軟體工程師進入程式碼世界的墊腳石。本書會讓讀者對設計程式碼的方式、可能有問題的事物以及避免這些問題的技術有一個廣泛的了解。但學習的旅程不應該在這裡就結束，軟體工程是個龐大且不斷發展的學科領域，強烈建議讀者繼續延伸廣泛閱讀來掌握這門主題的新發展。除了閱讀網路文章和部落格之外，讀者可能還需要一些關於這個主題的有用書籍，整理如下所示：

- *Refactoring: Improving the Design of Existing Code*, second edition, Martin Fowler (Addison-Wesley, 2019)
- *Clean Code: A Handbook of Agile Software Craftsmanship*, Robert C. Martin (Prentice Hall, 2008)
- *Code Complete: A Practical Handbook of Software Construction*, second edition, Steve McConnell (Microsoft Press, 2004)
- *The Pragmatic Programmer: Your Journey to Mastery, 20th anniversary*, second edition, David Thomas and Andrew Hunt (Addison-Wesley 2019)
- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994)
- *Effective Java*, third edition, Joshua Bloch (Addison-Wesley, 2017)
- *Unit Testing: Principles, Practices and Patterns*, Vladimir Khorikov (Manning Publications, 2020)

1 程式碼品質

本章內容

- 程式碼品質很重要的原因
- 高品質程式碼要達成的四個目標
- 用來確保程式碼品質的六個高階策略
- 寫出高品質程式碼，從中長期來看是可以節省時間和精力的

在過往的經驗中，您可能已經用過上百甚至上千種不同的軟體了，不管是在您電腦上安裝的各個程式、手機上的各種 App，以及在系統背後自行提供服務的各類程式——我們一直都是在與軟體互動。

甚至還有許多我們所依賴的軟體是您沒有意識到。舉例來說，我們信任銀行擁有一套不錯的後端軟體系統，不會意外地把我們帳戶的錢轉移給別人，或者突然讓我們背負數百萬的債務。

有時候我們會遇到用起來十分愉悅的軟體，其功能完全符合我們的要求，幾乎不會出錯，而且也很容易上手。但有時候我們也會遇到使用起來很可怕的軟體，不斷出錯、老是當機，而且在使用上很不順利。

有些軟體顯然沒有那麼重要，就像手機上某個有 bug 的 App 可能很煩人，但不是世界末日。可若是從另一個角度來看，銀行後端系統中的 bug 有可能毀掉大家的生活。即使是看起來不太重要的軟體問題也有可能會毀掉整間企業。如果使用者發現某個軟體令人討厭或難以使用，那就有可能會改用其他替代軟體。

更高品質的程式碼往往會產生更可靠、更易維護且錯誤更少的軟體。許多關於提升程式碼品質的原則不僅談到要確保軟體最初設計時是以這樣的理念為目標，還希望隨著需求的開展和新應用場景的出現，這套軟體在整個生命週期中都能保持這種狀態。圖 1.1 說明了程式碼品質對軟體品質的影響。

好的程式碼顯然不是製作出好的軟體的唯一因素，但卻是主要的因素之一。我們能夠擁有世界上最好的產品和行銷團隊，部署在最好的平台上，並使用最好的框架來進行建構，但歸根結底，軟體的一切都是要有人寫出程式碼來讓它實現的。

工程師在編寫程式碼時所做的日常決策看起來可能很細微，有時甚至無關緊要，但他們卻共同決定了「軟體」的好壞。如果程式碼中含有 bug、配置出錯或沒有正確處理錯誤的情況，那麼由此搭建起來的軟體就很可能會發生錯誤和不穩定，並導致無法正常運作。

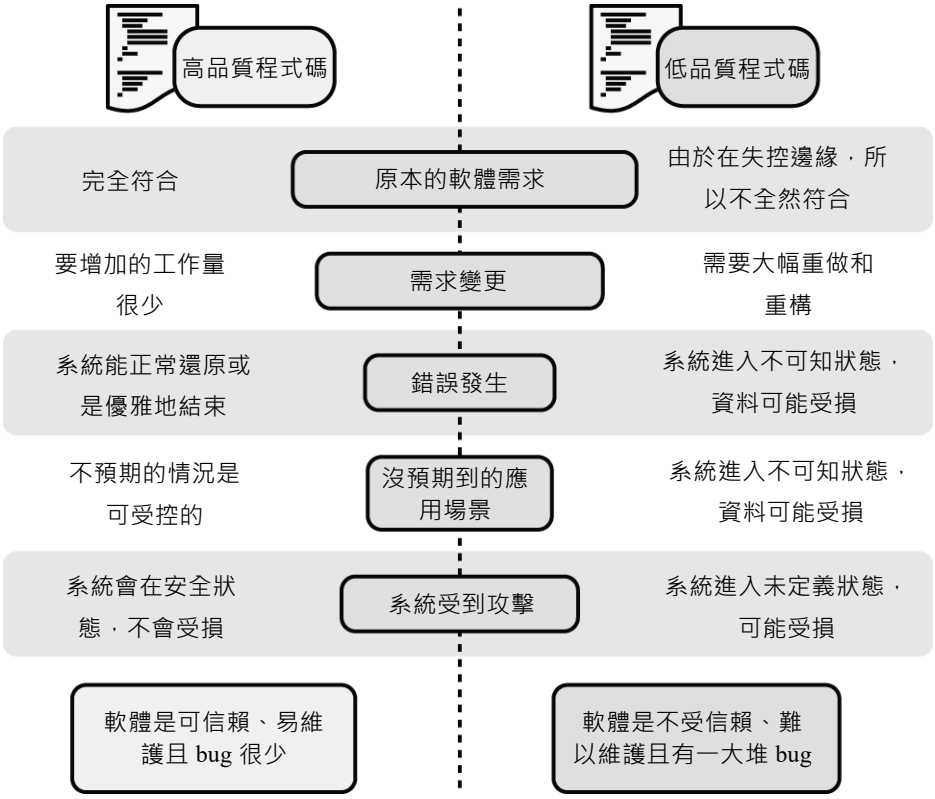


圖 1.1：高品質的程式碼會最大限度提升軟體的可靠性、可維護性，以及滿足需求的機會。低品質的程式碼往往會有相反的效果。

本章確定了高品質程式碼應該實現的四個目標，然後將其擴充為我們可以在日常工作中使用的六項高層次策略，以此來確保編寫的程式碼具備高品質。本書後面的章節會在逐一深入探討這些策略，並使用虛擬程式碼來呈現許多實例。

➢1.1 程式碼是怎麼變成軟體的

在我們深入討論程式碼品質之前，有必要簡單說明一下程式碼怎麼變成軟體。如果您已經熟悉了軟體開發和部署過程，那麼就可以直接跳到 1.2 小節。如果您只知道怎麼寫程式但還沒扮演過軟體工程師的角色，那麼本小節的內容會提供一個很好的綜觀概述。

軟體是由程式碼所組成，這很明顯，不需要特別說明。但大家不熟悉的（除非您已經有軟體工程師的經驗）是程式碼變成軟體在自然不受控的情景下（in the wild）執行的過程（送在使用者手中，或處理業務相關的工作）。

程式碼通常不會在工程師編寫的那一刻就能成為在自然不受控的情景下執行的軟體。一般都會有各種流程和測試檢查來確保程式碼的執行成效，以及不會破壞任何東西。這些相關作業通常被稱為軟體開發和部署過程。

不需要對這個過程有詳細的了解才能看懂本書的內容，但了解其大致的綜觀內容會有一定的幫助。首先是介紹一些術語：

- **程式碼庫（Codebase）**——可以用來建構軟體的程式碼庫。通常會由版本控制系統來管理，例如 git、subversion、perforce 等。
- **提交程式碼（Submitting code）**——有時也稱為「交付程式碼（committing code）」或「合併拉回請求（merging a pull request）」。程式設計師通常會對程式碼庫的本機副本內的程式碼進行修改。一旦修改到滿意程度，他們就會將程式提交到主程式碼庫。請留意：在某些設定中是由指定的維護者把修改拉回程式碼庫而不是由作者來提交。
- **程式碼審查（Code review）**——許多組織要求程式碼在提交到程式碼庫之前由另一名工程師進行審查校對。這有點像校驗程式碼，另一雙眼睛的校驗通常能發現原作者遺漏的問題。
- **預提交檢查（Pre-submit checks）**——有時也稱為「預合併掛附（pre-merge hooks）」、「預合併檢查（pre-merge checks）」或「預交付檢查（pre-commit checks）」。如果測試失效或程式碼不能編譯，都會阻止把修改內容提交到程式碼庫。
- **釋出版本（A release）**——由程式碼庫的快照建構的軟體版本。經過各種品質保證的檢查，然後將其釋放到外面（自然不受控的情景）。常聽到的「cutting a release」一詞，指的是對程式碼庫進行特定修訂並從中釋出發布的過程。
- **投入生產（Production）**——把軟體部署到伺服器或系統（而不是指送到客戶）時的術語。一旦軟體釋出並開始執行與作業相關的任務，就可以說軟體已正在投入生產執行中。

由程式碼變成外面現場可執行的軟體，其中的過程有很多變化，但關鍵步驟大致如下：

1. 工程師會在程式碼庫的本機副本中進行開發和修改。
2. 一旦修改到滿意，就會把這些修改發給程式碼審查。
3. 另一位工程師會審查程式碼並可能提出修改建議。
4. 一旦作者和審查者都修改到滿意時，程式碼就會提交到程式碼庫。
5. 釋出版本會定期從程式碼庫中發布出去，其釋出頻率會因組織和團隊的不同而有所不同（從隔幾分鐘到隔幾個月都有）。
6. 任何測試失效或沒有編譯的程式碼都會被阻止提交到程式碼庫或阻止其釋出發布。

圖 1.2 提供了典型軟體開發和部署過程的綜觀概要。不同的公司和團隊在這個過程會有個別的差異，過程中各個部分的自動化程度也可能會有很大的差異。

值得注意的是，軟體開發和部署過程本身就是個巨大的課題，很多都是用整本書來說明敘述的。圍繞這個課題的還有許多不同的框架和觀點，如果您對此有興趣，非常值得繼續閱讀其他更多相關內容。本書內容的主角並不是這些主題，因此不會更深入詳細介紹。就以本書的角度來看，您只需要了解程式碼怎麼變成軟體的綜觀概要即可。

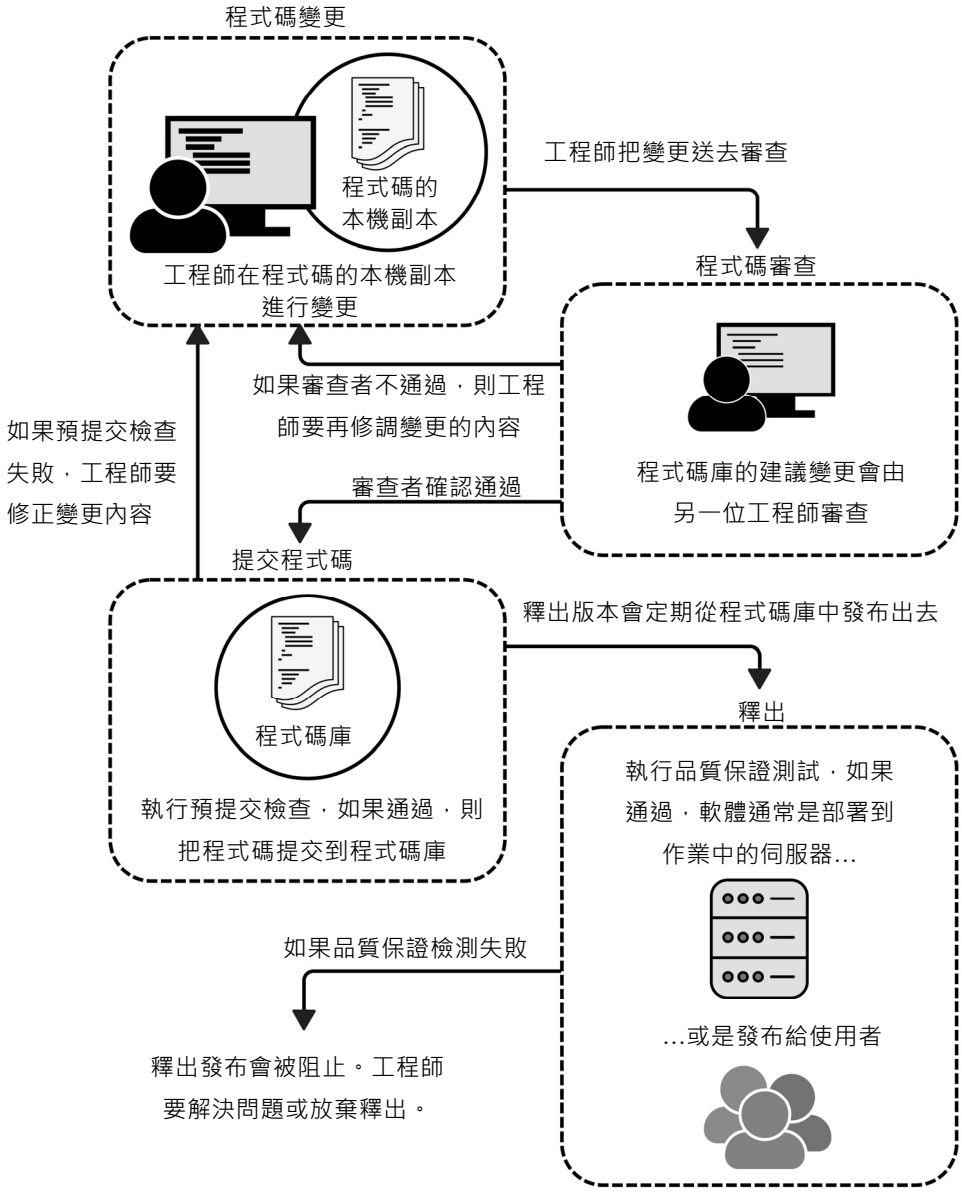


圖 1.2：典型軟體開發和部署過程的簡化圖解。在不同組織和團隊所使用的確切步驟和自動化水準可能會有很大差異。

➢1.2 程式碼品質的目標

如果我們想要買一台車，其「品質」可能是我們的主要考量因素之一。我們希望車子是：

- 安全的，
- 真的可以行駛，
- 不會分解，並且
- 行為可預測：當我們踩剎車時，汽車應該減速。

如果我們問某人什麼讓汽車品質更好，最有可能得到的答案之一就是它的製造很精良，這表示這台車設計精良，在行駛之前已經過安全性和可靠性測試，並且在組裝時正確良好。製作軟體其實也大同小異：要製作出高品質的軟體，我們需要確定它的建構是良好的。這就是程式碼品質的全部意義的所在。

「程式碼品質（code quality）」這個詞語有時會讓人覺得是挑剔一些瑣碎和不重要事情的建議。毫無疑問，您會不時遇到這種情況，但實際上這不是程式碼品質的問題。程式碼品質很大程度是以實務議題為根據，它有時關注小細節，有時關注大方向，但目標是一樣的：建構出更好的軟體。

話雖如此，程式碼品質仍是一個不好理解的概念。有時我們看到特定的程式碼時會想「糟糕」或「哇，這看起來很爛」，而有時偶然發現某些程式碼時會想「這太棒了！」。程式碼會引發這些類型反應的原因並不一定都很明顯，有時很可能只是一種沒有來由的直覺反應。

把程式碼定義為高品質或低品質，本質上就是一種主觀且決斷的事情。若想嘗試客觀地處理，我個人認為在編寫程式前先思考我真正想要實現的目標，這種方式很有用。在我看來，能幫助我實現目標的程式碼就是高品質的，而阻礙達成此目標的程式碼就是低品質的。

在編寫程式碼時，我希望實現的四個高層次目標是：

1. 應該能運作。
2. 應該要持續能運作。

3. 應該能適應不斷變化的需求。

4. 不用重新發明輪子。

接下來的幾個小節會更詳細解說這四個目標。

1.2.1 程式碼應該要能運作

這一點很明顯，並不需要特別說明，但我還是會補充一些內容。當我們編寫程式碼時，是試圖解決某個問題，例如實作出某項功能、修復某個錯誤或執行某項任務。程式碼的主要目標是要能運作，它應該要解決我們的問題。這也意味著程式碼沒有錯誤，因為錯誤的存在可能會阻止程式正常運作，進而無法完全解決問題。

在定義程式碼「運作（working）」的含義時，我們需要捕捉所有需求。舉例來說，如果我們正在解決的問題對效能（例如延遲或 CPU 使用率）特別敏感，那麼確保程式碼的執行效能就屬於「程式碼應該要能運作」這項目標，因為這是其中一項需求。這同樣適用於其他重要的考量因素，例如使用者隱私和安全性等。

1.2.2 程式碼應該要持續能運作

程式碼運作可能是非常短暫的事情，它今天可以運作，但我們如何確保明天或一年後仍然可以運作呢？很奇怪哦，為什麼程式碼會突然不能運作呢？關鍵是程式碼並不是獨立存在的，一不小心，程式很容易隨著周圍環境的變化而中斷運作。

- 程式碼可能會依賴於其他程式碼，而這些程式碼有可能會被修改、更新和變更。
- 任何新功能的需求都有可能代表著需要對程式碼進行修改。
- 我們試圖解決的問題可能會隨著時間推移而變化：消費者偏好、業務需求和技術考量等都會發生變化。

今天可以運作但明天因某些事情變化後就不能運作的程式碼是沒有用。建構可運作的程式碼通常很容易，但建立持續能運作的程式碼則困難許多。軟體工程

師在設計程式時最大考量之一就是要確保程式碼持續都能運作，這也是在編寫程式碼的所有階段都應該考量的事情。將其視為事後的再考量或是假設稍後只添加一些測試即可實現，這類想法通常是無效的方法。

1.2.3 程式碼應該能適應不斷變化的需求

某段程式只寫一次就不用再修改的情況非常罕見。軟體的持續開發可以維持幾個月、幾年，有時甚至是幾十年。在整個過程中，需求可能發生了變化：

- 商業現況變生轉變。
- 消費者偏好發生變化。
- 假設失效。
- 不斷添加新功能。

決定投入多少資源來讓程式碼具有適應性是個棘手的平衡取捨議題。一方面，我們知道軟體的需求會隨著時間的推移而演變（很少不會發生變化）。但另一方面，我們又無法確定會怎麼演變。我們不太可能精準預測出某段程式碼或軟體是怎麼隨時間演變的。但是我們不能因為不知道某些東西會如何演變，就完全忽略它會演變這個事實。為了說明這一點，讓我們思考兩種極端設想場景：

場景 A——我們嘗試準確預測未來需求可能會的發展，並設計程式碼能支持這些潛在的演變。這樣可能會花費數天或數週的時間來規劃出程式碼和軟體可能演變發展的所有方式。隨後還必須仔細考量編寫程式碼的每一個細節，以確保它能支持所有潛在的未來需求。這樣會大幅拖慢開發的速度，一個可能只需三個月就完成的軟體現在可能需要一年或更長時間才能搞定。到最後還可能是一場空，只是在浪費時間，因為競爭對手早在幾個月前就擊敗我們進入市場了，而我們對未來的預測還有可能是錯誤的。

場景 B——我們完全忽略了需求可能會演變的事實。編寫的程式碼只滿足現在的需求，也不花心力讓程式碼具有適應性。程式中大都是脆弱的假設，子問題的解決方案都捆綁在一起成為不可分割的大型程式碼區塊。我們在三個月內推出了軟體的第一個版本，但初始使用者的反饋表明，如果想要讓軟體成功，需要修改一些功能並且還要添加新功能。雖然需求的變化並不大，但是因為在當初寫程式時沒有考慮到適應性，我們唯一的選擇就是扔掉一切重新開始，然後

必須再花三個月的時間重新編寫軟體，如果需求又再次發生變化，我們又不得不再花三個月的時間重新來過。當我們在開發出滿足使用者需求的軟體時，競爭對手早就再次擊敗我們了。

情景 A 和情景 B 分別代表了兩個相反的極端情況。這兩種情況的結果都很糟糕，而且都不是建構軟體的有效方法。我們需要在這兩個極端中間找出一種平衡的方法。場景 A 與場景 B 之間的頻譜上的哪一點才是最佳的，並沒有單一的標準答案，這要取決於我們從事的專案類型以及公司的組織文化。

幸運的是，我們可以採用一些普遍適用的技術來確保程式碼具備適應性，而且無須確切知道未來會如何調整。我們會在本書介紹其中的各種技術。

1.2.4 不用重新發明輪子

當我們編寫程式碼來解決某個問題時，通常會把一個大問題分解成多個較小的子問題。舉例來說，假設我們編寫的程式是要載入某個影像檔，將其轉換為灰階影像，然後再儲存起來，我們需要解決的子問題如下：

- 從檔案載入一些位元組的資料。
- 把資料位元組解析為影像格式。
- 把影像轉換為灰階格式。
- 把影像轉換回位元組。
- 把這些位元組儲存回檔案。

上述的多個問題已經有其他人解決過了，舉例來說，從檔案載入一些位元組的處理可能是程式語言內建的功能。我們不用編寫自己的程式碼來與檔案系統進行低階處理。同樣地，只需匯入某個現有的程式庫就能取用現有功能來把位元組解析成影像。

如果我們真的去編寫自己的程式碼來與檔案系統進行低階處理，或是把位元組解析為影像，實際上就等於在重新發明輪子。使用現有解決方案而不是重新再發明一次，其理由有下列幾個：

- **節省時間和精力**——如果我們使用內建的檔案載入功能，可能只需幾行程式碼和幾分鐘的時間就搞定了。相比之下，重新編寫自己的程式碼來執行此項操作可能需要閱讀大量的檔案系統標準文件並寫出數千行程式碼，就算不花上幾週時間，也得要花上很多天才能完成。
- **降低出錯的機會**——如果已存在解決某個問題的現有程式碼，那麼這些程式應該已經過徹底測試，甚至已經被廣泛使用，所以程式碼中含有錯誤的可能性很低，就算有，也很可能已經被發現並修復了。
- **運用現有的專業知識**——把位元組解析為影像之程式碼的維護團隊可能是影像處理方面的專家。如果出現新版本的 JPEG 格式，他們會知道並更新其程式碼。重複使用他們已寫好的程式碼，我們能受益於他們的專業知識和未來的更新。
- **讓程式碼更容易理解**——如果有一套標準化的做事方法，那麼其他工程師很可能以前也見識過。大多數工程師都會在某個時間點進行讀取檔案的處理，此時能立即識別出程式內建的執行方式並了解其功能。如果我們為此編寫自訂的處理邏輯，那其他工程師不可能熟悉其作法，也無法立即知道我們自訂的程式是怎麼運作的。

不用重新發明輪子這個概念在相對的兩個面向都適用。如果有其他工程師已經編寫了解決子問題的程式碼，那麼我們應該呼叫此程式碼來用而不是再編寫自己的程式碼來解決問題。但同樣地，如果我們已編寫了程式碼來解決子問題，那麼我們在建構程式時應該要用便於其他工程師可重複使用的方式來編寫，如此一來，其他程式師就不需要重新再發明輪子了。

相同類型的子問題常會一直出現，因此在不同工程師和團隊之間分享程式碼的好處是顯而易見的。

➢1.3 程式碼品質的支柱

在前面內容看到的四個目標能協助我們把焦點放在怎麼從根本上去達成，但這裡並沒有提供關於日常編寫程式碼時該做什麼的具體建議。明確定出更具體的策略來幫助我們寫出滿足這些目標的程式碼是很有用的。本書就以六個策略來展開，我稱之為「程式碼品質的六大支柱」。我們會從每個支柱的高層次綜觀

描述開始，並在後面的章節提供具體的範例，展示如何把這些內容應用到日常編寫程式碼的工作中。

程式碼品質的六大支柱：

1. 讓程式碼可讀。
2. 避免意外驚訝。
3. 讓程式碼不會被誤用。
4. 讓程式碼模組化。
5. 讓程式碼可重用（reuse）和可泛化（generalizable）。
6. 讓程式碼可測試（testable）且能正確測試。

1.3.1 讓程式碼可讀

請看下面這段文字。故意寫得難以閱讀，所以不要浪費太多時間來了解。略讀一下看看您能吸收多少：

> 「取一個碗；我們現在將其稱為 A。拿一個平底鍋；現在稱為 B。將 B 裝滿水並放在爐灶上。取 A 放入奶油和巧克力，前者 100 克，後者 185 克，巧克力應該是 70%的黑巧克力。把 A 放在 B 上；再把 B 放在爐子上，直到 A 的內容物融化，然後從 B 取下 A。再拿一個碗；現在將其稱為 C。取 C 並在其中放入雞蛋、糖和香草精，第一個 2 個、第二個 185 克、第三個半茶匙。混合 C 的內容。等 A 的內容物冷卻後，將 A 的內容物加到 C 中並混合。取一碗；稱為 D。取 D 並將麵粉、可可粉和鹽放入其中，第一個 50 克、第二個 35 克，第三個半茶匙。將 D 的內容物徹底混合，然後過篩混入 C 中。將 D 的內容物充分混合。順便說一下，這是在製作巧克力布朗尼；我忘記先說明了嗎？取 D 加入 70 克巧克力片，將 D 的內容物適當混合好。拿出烤盤；稱為 E。在 E 上使用烘焙紙塗上油並排成一行。把 D 的內容物放入 E。我們將烤箱稱為 F。順便說一下，您應該將 F 預熱到 160° C。將 E 放入 F 中 20 分鐘，然後從 F 中取出 E。讓 E 冷卻幾個小時。」

現在有幾個問題要問：

- 這段文章的在說明什麼內容？

- 遵循這些說明後，我們最後會做出什麼東西？
- 製作的成分是什麼，需要的量又是多少？

這些問題都可以在上述文章中找到答案，但不容易找出，因為文字的可讀性很差。歸納出來的幾點是降低文字可讀性的理由：

- 沒有標題，所以我們必須通讀整篇文章才能弄清楚它是在談什麼內容。
- 這篇文章並沒有好好呈現一系列步驟（或子問題），反而是以一大堆文字來呈現。
- 談論之事物被無益且模糊的名稱來代替，例如用「A」這種名稱而不是「裝有融化奶油和巧克力的碗」。
- 資訊項目都放在遠離需要的地方：配料成分和數量分開列出、烤箱需要預熱這種重要說明卻只在最後才提到。

（如果您不想再看這段文字，沒關係，這是份巧克力布朗尼的食譜。附錄 A 中有一個更具可讀性的版本，如果您真的想要製作時可參考。）

閱讀一段寫得很糟糕的程式碼並試圖解決問題與剛剛閱讀布朗尼食譜的經歷很相像。尤其是我們可能很難理解程式碼的以下內容：

- 它能做什麼。
- 它是怎麼做到的。
- 它需要什麼成分（輸入或狀態）。
- 執行這段程式碼後我們會得到什麼。

在某個時機下，別的工程師很可能需要閱讀我們的程式碼並理解怎麼用它。如果我們的程式碼在提交之前必須經過程式碼審查，那表示馬上就需要審閱程式內容了。就算忽略程式碼審查，也可能在某個時間點有人會需要查閱我們的程式碼並試圖弄清楚它的功用。當需求發生變化或程式碼需要除錯時，查閱程式碼的動作就會發生。

如果我們的程式碼可讀性差，其他工程師不得不花費大量時間來解譯其內容。別人很有可能誤解其作用或錯過某些重要的細節。如果發生這種情況，那麼在程式碼審查期間可能無法發現錯誤，更有可能的是在其他人修改我們的程式碼

以加入新功能時引入新的錯誤。軟體能運作的每項功能都是由很多程式碼一起合作達成的。如果工程師無法理解其中某些程式碼的作用，那要怎麼確保整套軟體能正常運作呢？就像上述料理食譜一樣，程式碼需要具備可讀性。

在第 2 章的內容會看到定義正確的抽象層是怎麼協助我們解決這個問題。而在第 5 章則是介紹一些讓程式碼更具可讀性的特定技術。

1.3.2 避免意外驚訝

在生日時收到禮物或中獎是「驚喜」的例子。然而，當我們試圖完成一項特定的任務時，驚喜通常會變成「驚嚇」。

請想像一下，在您餓了的時候想要點一些披薩來吃。您拿出手機，找到披薩餐廳的號碼，然後撥號。線路沉默了很長一段時間，但最終接起來，另一端的聲音問您想要點什麼。

「請送一份大的瑪格麗特。」

「哦，好的，您的地址是什麼？」

半小時後，您的訂單送達，您打開袋子發現以下內容（圖 1.3）。

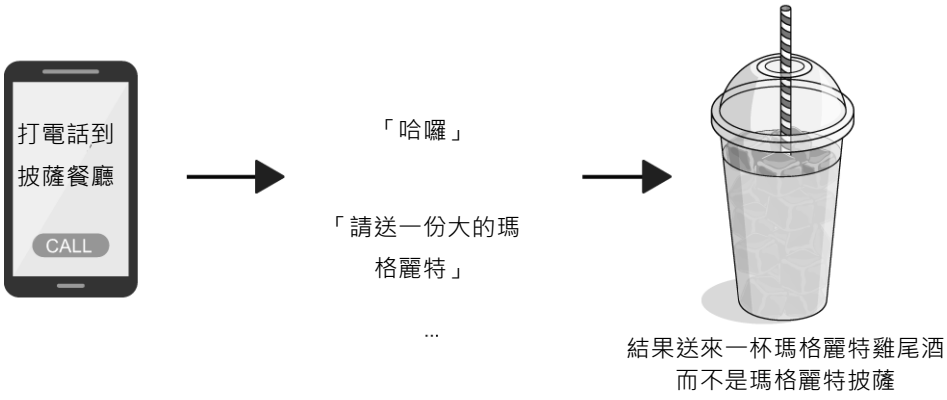


圖 1.3：如果您認為打電話過去的是一家披薩餐廳，但實際上卻是一家墨西哥餐廳，那麼您下的訂單可能仍然有效，但在交付時您會嚇到。

哇，這太令人驚訝了！顯然有人把這個「瑪格麗特」（比薩）誤認為那個「瑪格麗特」（雞尾酒），但這有點奇怪，因為披薩餐廳並不提供雞尾酒。

事實上，您在手機上使用的自訂撥號 App 增加了一個新的「聰明」功能。App 的開發者觀察到，當使用者打去的餐廳忙線時，80% 的使用會立即撥打別的餐廳，因此內建了一個方便、省時的功能：當您撥打後識別為餐廳且正在忙線，它會無縫接著撥打手機中的下一間餐廳號碼。

在上述的案例中，碰巧是接著撥打到您最喜歡的墨西哥餐廳，而不是您原本認定的披薩餐廳。這家墨西哥餐廳絕對有提供瑪格麗特雞尾酒，但沒有披薩。這套 App 的開發者立意良好，認為能讓使用者更輕鬆，但卻弄出令人驚訝的結果。打電話的心理狀況是根據聽到的內容來確定要發生的事情。如果我們聽到電話語音應答，這時的認知心理模型會認定是已經接通了原本撥打的號碼。

撥號 App 中的新功能超出了我們預期的認知，它打破了我們認知心理模型的假設，也就是如果有聲音接聽，表示已經接通了我們要撥打的號碼。自動在忙線時接著撥打很可能是個有用的功能，但因為它的行為超出了正常人的心理認知，所以這項功能需要明確指出發生了什麼情況，比如有一個音訊告知原本撥打的號碼忙線中，並詢問是否改撥到另一家餐廳。

把撥號 App 想像成是某段程式碼。另一位使用這段程式碼的工程師也會有一個認知心理模型，會用程式中的名稱、資料型別和通用慣例等線索來建構一個預期以什麼當作輸入、做什麼處理，以及會返回什麼的程式碼。如果我們的程式碼跳脫了這種認知心理模型，那麼它導致的錯誤會蔓延在軟體中。

在撥打電話到披薩餐廳的例子中，即使出現了意外，但訂餐的處理一切似乎都在進行：您點了一分瑪格麗特，餐廳很樂意送餐。直到經過一段時間的送餐之後，您才發現自己無意中點了雞尾酒而不是披薩。這類似於軟體系統中某些程式碼做了一些令人驚訝的事情：因為程式碼的呼叫方沒想到會這樣，所以也不知道導致的結果。一般來說，程式有一段時間會沒問題，但後來當發現程式處於無效狀態或向使用者返回奇怪的值時，事情早就不可收拾了。

即使立意良善美好，編寫出來的有用或聰明程式碼也可能有造成意外驚訝的風險。如果程式碼會引發令人驚訝的事情，那麼工程師就可能在無知或不預期的情況下誤用這些程式碼，這樣會導致系統手癆腳跛，出現一些奇怪的行為。出現的錯誤或許只是煩人的小問題，但也可能會導致破壞重要資料的災難性狀況。我們要警惕在程式碼中引起的意外驚訝狀況，並儘可能避免。

在第 3 章會說明如何考量程式碼契約，這是一種可以協助解決此問題的基本技術。第 4 章介紹了錯誤的狀況，如果出錯時沒有發出適當的信號或進行後續處理，這些錯誤可能就會導致意外和驚嚇。第 6 章的內容則著眼於一些更具體能避免意外驚訝的技巧。

1.3.3 讓程式碼不會被誤用

如果我們看電視背面的一些插孔，它可能看起來像圖 1.4 的樣子，會有一堆不同的插槽來讓我們插入相關功能的纜線。重要的是，插槽會以不同的形狀呈現，電視製造商的這種作法不會讓電源線誤插進 HDMI 插槽。

圖 1.4：電視背面有一堆不同形狀的插槽，不同功用的纜線不會插錯插槽

請想像一下，如果製造商沒有這樣做，而是讓每個插槽都製成相同的形狀。您認為會有多少人在電視背面摸索時不小心插錯纜線呢？如果有人錯把 HDMI 纜線插入電源插槽，結果也許還能運作，雖然很煩，但沒什麼大災難發生。但如果有人把電源線錯插入 HDMI 插槽，那就有可能會引發爆炸。

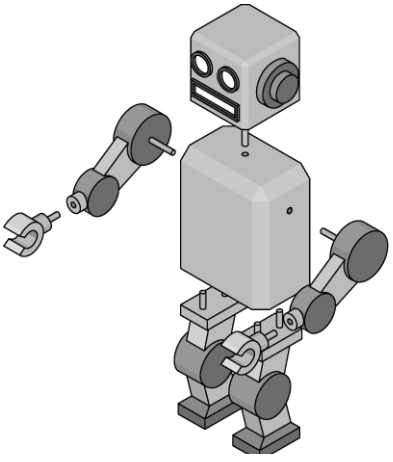
我們寫的程式碼經常被其他程式碼呼叫使用，有點像電視背面的情況。我們期望其他程式碼「插入」某些東西，例如輸入參數或在呼叫之前把系統置於某種狀態。如果把錯的東西插入我們的程式碼中，那麼程式有可能會當掉、讓系統崩潰、資料庫永久損壞或某些重要資料丟失。就算沒有當掉，程式碼也可能無法執行。我們的程式碼會被呼叫是有其緣由的，插入錯誤的東西那就有可能讓想要處理的重要工作無法執行，或是發生了一些奇怪的行為但沒有引起注意。

我們可以要讓程式碼難以或不可能被誤用來儘量提高其運作執行和持續能運作的機會。有許多實務方法可以做到這一點，第 3 章介紹了程式碼契約，是一種協助讓程式碼難以被誤用的基本技術（類似於避免意外驚訝中談到的內容）。第 7 章則介紹了許多讓程式碼難以誤用的更具體技術。

1.3.4 讓程式碼模組化

模組化（modularity）意味著某個物件或系統是由獨立交換或可替代的較小元件所組成。為了證明這一點以及說明模組化的好處，請參閱和思考圖 1.5 中的兩個玩具。

左邊的玩具是高度模組化的。頭部、手臂、手掌和腳都可以輕鬆獨立地交換或替代，而且不會影響玩具的其他部分。相反地，右邊的玩具是高度非模組化，不容易交換或替代這個玩具的頭部、手臂、手掌或腳。



模組化的玩具

非模組化的玩具

圖 1.5：模組化的玩具可以輕鬆重新配置。縫合的布玩具則很難重新配置。

模組化系統（如左側的玩具）的主要特徵之一是不同的元件會有定義明確的介面，相互交流的點盡可能少。如果我們把手掌視為一個元件，左側的玩具則只有一個與介面相互交流的點：一個栓釘和一個適合它的孔洞。右側玩具在手掌和玩具相連處是個非常複雜的介面：手掌和手臂上相連處是由 20 多圈織線交互連結在一起。

現在請想像一下，假設我們的工作是維護這些玩具，有一天經理告訴我們，現在有一項新的需求，手掌要有手指。那我們會更願意使用哪種玩具結構系統來完成工作呢？

對於左側的玩具，我們只要製造出新設計的手掌，隨後很容易就能把新手掌換接到玩具的手臂上。就算經理在兩週後又改變了主意，我們一樣可以輕鬆地把玩具恢復回以前原本的手掌配置。

以右側的玩具來看，我們可能需要用到剪刀，剪下 20 多圈線，然後再把新的手掌縫到玩具上。在此過程中有可能剪壞玩具，如果經理在兩週後又改變主意，我們需要以經歷相同的過程，花費同樣多的力氣來把玩具恢復到以前的配置狀態。

軟體系統和程式碼庫的關係與這些玩具的狀況非常相似。把一段程式碼分解為獨立的模組是很有益處的，兩個相鄰模組之間相互交流的地方只有一處，而且這裡使用的是定義良好的介面。這樣有助於確保程式碼更容易適應不斷變化的需求，因為對某項功能的更改不需要在整支程式中進行大量的更改。

模組化系統通常也更容易理解和推理，因為功能被分解成可管理的區塊，而且功能區塊之間的相互交流有很好地的定義和記錄。這樣增加了程式碼可以先運作並在未來持續能運作的機會，因為工程師不太可能誤解程式碼的作用。

在第 2 章會介紹建構清晰抽象層的基本技術是怎麼引導我們實現更模組化的程式碼。在第 8 章則會研究探討一些讓程式碼更加模組化的特定技術。

1.3.5 讓程式碼可重用和可泛化

可重用性和可泛化性是兩個相似但略有不同的概念：

- **可重用性（reusability）**是指某些東西可以在多種設想場景下用來解決相同的問題。手用電鑽可重複使用，可在牆壁、地板和天花板等不同場景進行鑽孔。問題都是一樣的（都想要鑽一個洞），但場景不同（鑽入牆壁、鑽入地板、鑽入天花板）。
- **可泛化性（generalizability）**是指某些東西可用於解決多個概念上相似但略有不同的問題。手用電鑽也具有可泛化通用性，因為它不僅可以用來鑽孔，還可以當作電動起子用來栓螺釘。鑽頭製造商認知到「旋轉」某物品這樣的動作是能適用在鑽孔和栓螺釘這種普遍狀況，因此開發了一種通用工具來解決這兩個狀況。

在以下狀況中，我們會馬上認識到這樣做的好處。請想像一下，如果我們需要用四種不同的工具：

- 一種只能在保持水平鑽孔的鑽頭，這表示它僅能用來鑽牆壁。
- 一種只能以 90° 角向下鑽孔的鑽頭，這表示它僅能用來鑽地板。
- 一種只能以 90° 角向上鑽孔的鑽頭，這表示它僅能用來鑽天花板。
- 一種只能當作電動起子把螺釘旋入物體。

我們需要花更多的錢來購買這四種工具，還必須隨身攜帶更多的東西，也必須為四種電池充電——這是一種浪費。值得慶幸的是，有人發明了一個既可重用又能可通用的電鑽，而我們只需要一個就能完成上述這些不同的工作。您應該已經猜到這裡所說的手用電鑽就是用來比喻程式碼的情況吧！

程式碼需要時間和精力來建立，一旦建立完成，它也需要持續的時間和精力來維護。建構程式碼也不是沒有風險的：無論我們多麼小心，寫出來的程式碼也都可能會含有些許錯誤，而且寫的程式碼愈多，錯誤就可能就愈多。這裡的重點是在程式碼庫中的程式行數越少越好。當工作似乎涉及到編寫程式碼的報酬時，這樣的說法似乎有點奇怪，但實際上我們是解決問題來獲得報酬的，而程式碼只是實現這個目標的一種手段。如果能以更少的精力來解決該問題，且減少無意間引入錯誤而造成其他問題的機會，那就更好了。

讓程式碼可重用和可泛化，這樣能讓我們（和其他人）在不同地方、不同場景中透過程式碼庫來使用它，並解決不同的問題。這樣的程式碼節省了時間和精力，並讓程式更可靠，若我們重用已經在自然狀態下測試過的邏輯，這也代表就算有任何錯誤也可能已經被發現和修復了。

更模組化的程式碼也更容易重用和泛化。模組化相關的章節內容也和可重用性和可泛化性主題有密切相關。此外，第 9 章的內容涵蓋了許多專門用於讓程式碼更具可重用性和可泛化通用性的技術和注意事項。

1.3.6 讓程式碼可測試且能正確測試

正如我們之前在軟體開發和部署圖（圖 1.2）中所看到的那樣，測試是確保錯誤和損壞的功能最後不會在外面執行的重要過程。測試是開發流程中兩個關鍵點的主要防禦機制（如圖 1.6）：

- 防止錯誤或損壞的功能提交到程式碼庫。
- 確保具有錯誤或損壞功能的版本會被阻止並且不會在最後釋出。

因此，測試是確保程式碼能正常運作且持續能運作的重要處理過程。

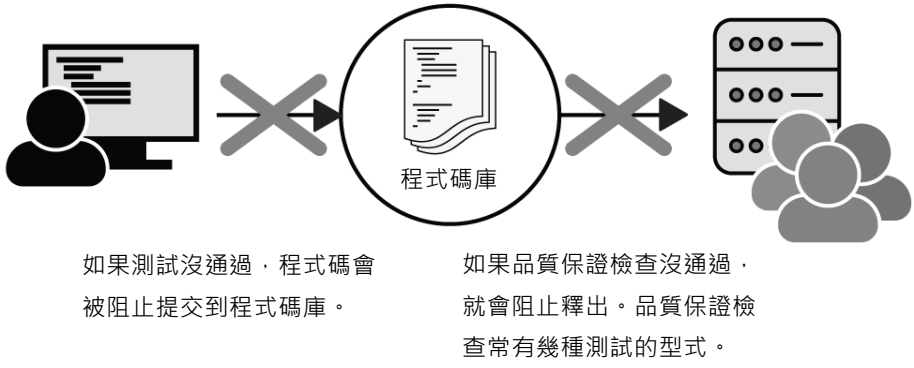


圖 1.6：測試可以最大程度地減少錯誤和損壞的功能進入程式碼庫的機會，也是確保有問題程式碼不會被釋出到外面是重要的過程。

「測試」在軟體開發過程中的重要性是毫無疑問的，在這之前已經講述過很多次了，很容易視為陳腔濫調，但測試真的十分重要。正如在整本書中很多地方都會看到這樣的觀點：

- 軟體系統和程式碼庫往往過於龐大和複雜，一個人無法了解每一個細節，
- 而人（就算是非常聰明的工程師）都會犯錯。

這或多或少是日常生活中的事實，除非我們用測試完全鎖定程式碼的功能，否則上述的問題會習慣性聯合起來影響我們（和我們的程式碼）。

程式碼品質支柱有兩個重要概念：讓程式碼「可測試」和「正確測試」。測試（testing）和可測試性（testability）是相關的，但兩者有不同的考量：

- **測試**——顧名思義，與測試程式碼或整個軟體有關。測試可以是手動的，也可以是自動的。身為工程師，我們通常會透過編寫測試程式碼來處理「真實」程式碼並檢查一切行為是否正常，這樣的方式能讓測試自動化。測試有分不同的等級，最常見的三個如下所示（請注意，下面並不是詳盡

的列表，測試的分類也有很多種方式，而且在不同的組織使用名稱也可能不同）。

- **單元測試（Unit tests）**——這些通常用來測試小型程式碼單元，例如單個函式或類別。單元測試是工程師在日常編寫程式碼中最常使用的測試級別，也是本書唯一會詳細介紹的測試級別。
- **整合測試（Integration tests）**——系統通常由多個元件、模組或子系統所組成。巴這些元件和子系統連接在一起的過程稱為**整合**。整合測試就要確保這些連接整合後能正確運作且持續能運作。
- **端對端測試（End-to-end tests）**——這些測試會從頭到尾貫穿整套軟體系統的典型流程（或工作流程）。如果線上購物商店的軟體有問題要測試，那麼其中一個端對端測試的例子可能是確認自動驅動 Web 網路瀏覽器來讓使用者順利完成購買的流程。

■ **可測試性**——這是指「真實的」程式碼（不是指測試程式碼），可測試性描述了程式碼適合被測試的程度。可測試性的概念也套用到子系統或系統等級。可測試性通常與模組化高度相關，愈模組化的程式碼（或系統）是愈容易測試。請想像一下，某家汽車製造商正在開發行人緊急煞停系統。如果系統不夠模組化，那麼測試它的唯一方法可能是要把它安裝在真實的汽車中，駕駛汽車到真正的行人前，然後檢查汽車是否自動煞停。如果是這種情況，那麼系統可以測試的場景是有限的，因為每次測試的成本都非常高：要先造一整台車、租用測試跑道、以真人假裝路上的行人。如果緊急煞停系統是一套可以在真實汽車之外執行的獨立模組，那麼它的可測試性就會大幅提高。這樣就可以透過行人走動的預錄視訊來進行測試，然後檢查系統是否正確輸出了用來煞停的信號。在這種情況下，以千百種不同行人走動的場景來測試就變得非常容易、便宜且安全。

如果程式碼不可測試，那就不太可能正確地測試它了。為了確保我們編寫的程式碼是可測試的，最好在寫程式時要不斷提醒自己「我們會如何測試它？」。因此，不應該把測試視為事後才要考量的事情：它在編寫程式的所有階段中都是不可少的基本部分。第 10 章和第 11 章都是討論關於測試的內容，但因為測試在編寫程式碼的各個階段都是不可或缺的，您會發現在本書的很多地方會一直提到「測試」這項工作。

NOTE **測試驅動開發** 因為測試是編寫程式碼時不可或缺的考量之一，所以有些工程師主張應該在編寫程式碼之前先寫出測試，這是測試驅動開發（Test-Driven Development，TDD）流程所倡導的實務之一。我們會在第 10 章（10.5 小節）對此進行更多討論。

軟體測試是個巨大的主題，坦白說，本書篇幅有限，其內容只能盡量說明和討論。在本書中，我們會介紹單元測試程式碼中一些最重要但經常被忽視的內容，因為這些內容通常在日常編寫程式碼的過程中最有用。但請注意，就算讀完整本書，所談到的也只是觸及軟體測試的一些皮毛而已。

➢1.4 編寫高品質的程式碼會拖慢開發的速度嗎？

這個問題的答案是，從短期來看，編寫高品質的程式碼似乎會減慢開發的速度。編寫高品質程式碼通常需要花費更多的思考和精力，而不僅僅是寫出我們腦中想到的第一件事。但是，如果我們編寫的不是小型、執行一次就扔掉的工具程式，那麼編寫高品質的程式碼一般都會加快中長期的開發時間。

請想像一下，假設我們想要在家中放一個架子。會有一種「正確」的方式可以做到這一點，另外也有一種「速成」的方式來做到這一點：

- **正確（proper）方式**——我們會鑽孔和以牆釘栓入磚牆的方式把支架固定到牆面上，然後再把架子的擱板安裝在支架上。耗時約 30 分鐘。
- **速成（hacky）方式**——我們買一些膠水，直接把架子黏在牆面上。耗時約 10 分鐘。

似乎以速成方式來放置架子可以節省 20 分鐘，也能省去使用鑽頭和起子的額外工作。我們選擇了使用速成的方式，現在讓我們考慮之後會發生什麼。

我們把架子黏在牆的表面，而這個牆面可能是一層石膏而已。石膏並不牢固，很容易開裂和大面積脫落。一旦開始使用架子，架子上放置的物品重量很可能會讓石膏開裂，架子掉落也會讓牆面的石膏大面積脫落。發生這種情況時就沒有架子可用了，另外還要重新粉刷和裝修牆面（這些工作就算不用幾天也可能

需要幾個小時才能完成）。就算出現某種奇蹟，架子沒有脫落，這種以快速方式黏上去的架子也可能為未來潛藏了問題。請想像以下幾個場景：

- 當發現架子沒有放得很平（發現錯誤）時：
  - 以支架式擱板來說，我們可以在支架和擱板之間加個較小的墊片來修正。耗時大約 5 分鐘。
  - 以膠水黏合的架子來說，我們需要把它從牆上拔下來，牆面一大塊石膏會脫落。我們現在補上石膏並重新粉刷牆面，然後再把架子黏回去。花費的時間：就算不花費幾天時間也要花費幾個小時。
- 我們決定要重新裝修房間（有新需求）：
  - 取下螺絲後再取下擱板和分開支架。重新裝修房間之後再把架子擱板裝回去。與架子相關的處理工作所用時間約 15 分鐘。
  - 若是以膠水黏合的架子，第一種選擇是把架子留在原位，然後冒著油漆滴在上面的風險來裝修房間，而且架子的邊緣油漆一定不會太整齊，必須在周圍修補或貼上壁紙。或者把架子從牆上拔下來，然後才重新粉刷裝修房間。這兩種選擇變成是隨便弄一下的重新裝修，或花費數小時或數天時間重新粉刷牆面。

您看明白了吧！乍看之下，把支架釘在牆面再放架子是浪費 20 分鐘沒必要的做法，但從長遠來看，這種方式很可能為我們節省了更多時間和麻煩。以未來重新裝修的案例來看，我們還能了解到若是從速成取巧的方案開始時，需要推動工作會更多，例如在重新裝修時要在架子周圍補漆或貼壁紙，而不是取下螺絲就能拆解架子。

編寫程式碼與這裡談的例子很相似。在不考慮程式碼品質的情況下，編寫程式時第一個想到的可能是要節省一些時間。雖然可以很快就得到一個程式碼庫，但它是脆弱且複雜的，也會變得越來越難以理解或推斷。想要在其中加入新功能或修復錯誤就變得越來越困難和緩慢，因為我們必須先損壞原本的東西和重新設計才能搞定。

您以前聽過「欲速則不達」這句話吧！這是生活中觀察許多事物得到的參考，過於倉促行事而沒有仔細考慮或正確處理事情，往往就會導致錯誤，反而降低整體的速度。「欲速則不達」詮釋了為什麼編寫高品質的程式碼反而能加快速度的原由，請不要把「匆忙」誤認為「速度」。

➢總結

- 要建立好的軟體，我們需要編寫高品質的程式碼。
- 在程式碼成為外面執行的軟體之前，通常必須通過幾個階段的檢查和測試（有時是手動的，有時是自動的）。
- 這些檢查有助於防止錯誤和把損壞的功能送到使用者或關鍵業務系統中。
- 在編寫程式碼的每個階段都要考量測試是很好的習慣，不應把測試看作是將來才要處理的工作。
- 編寫高品質的程式碼最初可能會減慢開發的速度，但從中長期來看，通常能加快開發時間。