叢書編輯前言

我的孫子正在學習寫程式。

是的，你沒看錯。我 18 歲的孫子正在學習電腦程式設計。誰在教他呢？他的姑姑，也就是我最小的女兒，她出生於 1986 年，而她在 16 個月前決定將職業生涯從化學工程轉向程式設計。他們倆為誰工作呢？我的大兒子，他和我的小兒子一起，正在創辦他的第二家軟體顧問公司。

是的，電腦軟體在這個家族的血脈中流動。而且，你想的沒錯，我從事程式設計工作已經很久很久了。

總之，我的女兒請我花一個小時和我的孫子一起，教授他電腦程式設計的基礎知識以及如何入門。於是我們開始了一個 Tuple 課程，我向他講授什麼是電腦、它們是如何出現的，以及早期的電腦是什麼樣子的，還有……嗯，你知道的，就是那些。

課程快要結束時，我使用 PDP-8 的組合語言編寫了將兩個二進位整數相乘的演算法。為那些不知道的人補充一下，PDP-8 並沒有乘法指令，你必須寫出一個演算法來將數字相乘。

事實上，PDP-8 甚至沒有減法指令；你必須使用 2 的補數（two's complement）並加上一個偽負數（讓讀者理解）。

在我即將完成這個程式範例時，我突然發現我把我孫子嚇壞了。我的意思是，我 18 歲的時候，這種技術細節會讓我激動不已，但對於他姑姑正試圖教他如何編寫簡單 Clojure 程式的一個 18 歲孩子來說，也許就沒有那麼大型的吸引力了。

總之，這讓我想到了程式設計實際上有多難。而它很難，真的很難。它可能是人類曾經嘗試過的最困難事情。

哦，我的意思不是說寫程式碼來計算一些質數（prime numbers），或 Fibonacci 數列，或簡單的氣泡排序（bubble sort）很困難，那並不是太難。但是一個空中交通管制系統（Air Traffic Control system）呢？一個行李管理系統（luggage management system）呢？一個材料清單系統（bill of materials system）？ *Angry Birds* 呢？現在這就很難了，真的、真的很難。

我認識 Mark Seemann 也有好幾年了，我不記得曾經真正見過他，可能是我們從未真正在同一個地方見過面，但我和他在專業新聞群組和社交網路上有相當多的互動。他是我最喜歡的持不同意見的人之一。

他和我在各種事情上都有不同的看法。我們在靜態與動態定型（static versus dynamic typing）上有所分歧。我們對於作業系統和語言有不同意見。我們在很多具有智力挑戰的事情上有意見分歧。但是，與 Mark 持不同意見是一件必須非常小心的事情，因為他的論證邏輯是無懈可擊的。

因此，當我看到這本書，我想到的是，讀過這本書並提出不同的想法，會帶來多大的樂趣。而那正是後來發生的事情。我讀完了它。我不同意其中的一些看法。而我在試圖找出一種方式，使我的邏輯超越他邏輯的過程中，得到很多樂趣。我想我甚至可能有在一兩個案例中成功做到這件事，至少在我的腦子裡是這樣，也許啦。

但這不是重點。重點在於，軟體是很困難的，而過去七十年的大部分時間都花在試圖找出使它變得更容易一點的方法。Mark 在這本書中所做的

就是收集這七十年來所有最好的想法，並將它們彙編到同一個地方。

不僅如此，他還將它們組織成一套經驗法則和技巧，並按照你會執行它們的順序來安排。這些經驗法則和技巧相互依存，幫助你在開發一個軟體專案的過程中，從一個階段走向另一個階段。

事實上，Mark 在本書的整個篇幅中開發了一個軟體專案，同時解釋了每個階段以及有利於該階段的經驗法則和技巧。

Mark 使用 C#（我不同意的事情之一 ;-)），但那不重要。程式碼很簡單，而且那些經驗法則和技巧適用於你可能使用的任何其他語言。

他涵蓋了諸如檢查表（Checklists）、TDD、Command Query Separation（命令查詢分離）、Git、Cyclomatic Complexity（循環複雜度）、Referential Transparency（參考透明性）、Vertical Slicing（垂直切片）、Legacy Strangulation（絞殺舊有程式碼）、Outside-In Development（從外而內型開發）等內容，僅舉幾例。

此外，在這些書頁中，到處都散落著一些精華。我的意思是，你正讀過去的時候，他會突然說出像這樣的一句話：「把你的測試函式旋轉 90 度，看看你能不能在 Arrange/Act/Assert 三要素的 Act 上取得平衡」或者「目標不是快速寫出程式碼，目標是可持續發展的軟體」或者「把資料庫綱目 commit 到 git」。

這些精華中，有些是深刻的，有些只是閒談，其他則是猜測，但所有的這些都是 Mark 多年來所獲得的深刻洞察力的例子。

所以請讀這本書。仔細閱讀它。思考過一遍 Mark 無懈可擊的邏輯。內化這些經驗法則和技巧。停下來思考那些突然出現在你面前的洞見寶石。那麼也許，當你為的孫子輩講課時，你就不會把他們嚇壞了。

—Robert C. Martin

序言

在 2000 年代的後半部，我開始擔任一家出版社的技術審閱者。在審閱了一些書之後，編輯就一本關於 Dependency Injection（依存性注入）的書聯繫了我。

這個序曲有點奇怪。通常，當他們為了一本書而與我聯繫時，都已經有了作者和目錄。然而，這一次，那些都沒有。編輯只是要求通個電話，討論這本書的主題是否可行。

我想了幾天，發現這個主體很有啟發性。同時，我看不出有必要寫一整本書。畢竟，這些知識就在那裡：部落格文章、圖書館文獻、雜誌文章，甚至有一些書籍都涉及到相關的主題。

經過反思，我意識到，雖然這些資訊都在那裡，但它們是分散在各處的，使用的術語也不一致，有時甚至是相互衝突的。收集這些知識並以一致的模式語言將之呈現出來，會是有價值的事情。

兩年後，我自豪地成為一本出版書籍的作者。

幾年過去了，我開始考慮再寫一本書。不是這本書，而是關於其他主題的書。然後我又有了第三個想法、第四個想法，但也不是這一本。

十年過去了，我開始意識到，當我向團隊提供諮詢，幫助他們寫出更好的程式碼時，我會建議一些從比我更睿智的人那裡學到的做法。而且，我再次意識到，這些知識大部分都已經存在了，但它們是分散的，很少有人明確地將這些點連接起來，形成一個關於如何開發軟體的連貫敘述。

根據我在第一本書中的經驗，我知道收集不同的資訊並以一致的方式呈現是有價值的。這本書是我在建立這樣的一種容器的嘗試。

誰應該讀這本書

本書針對的是至少有幾年專業經驗的程式設計師。我希望讀者有遭受過一些糟糕的軟體開發專案之折磨；有處理無法維護的程式碼的經驗。我還期望讀者能夠不停尋求改進。

核心的讀者群是「企業開發人員（enterprise developers）」：特別是後端開發者（back-end developers）。我職業生涯的大部分時間都在這個領域，所以這單純反映出我自己的專長。但如果你是前端開發者、遊戲程式設計師，開發工具工程師，或者其他完全不同的人，我希望你仍然會從閱讀本書中獲益良多。

你應該要能夠輕鬆地閱讀 C 族系中編譯式物件導向語言的程式碼。雖然我在職業生涯的大部分時間裡都是一名 C# 程式設計師，但我從帶有 C++ 或 Java[1] 範例程式碼的書中學到了很多。這本書扭轉了這種局面：它的範例程式碼以 C# 語言撰寫，但我希望 Java、TypeScript 或 C++ 開發者也能發現它的用處。

1　如果你對我指的是哪些書感到好奇，請看一下參考書目。

預備知識

這並不是一本初學者的書。雖然它涉及到如何組織和架構原始碼，但它並不包括最基本的細節。我希望你已經明白為什麼縮排（indentation）是有幫助的、為什麼冗長的方法（methods）容易出問題、全域變數是不好的，等等。我不指望你讀過 *Code Complete* [65]，但我假定你知道其中所涵蓋的一些基本概念。

給軟體架構師的注意事項

「架構師（architect）」這個詞對不同的人意味著不同的東西，即使是限定在軟體開發的情境脈絡之下。有些架構師專注於大局，他們幫助整個組織的成功完成大業。其他架構師則深入到程式碼中，主要關注特定源碼庫的可持續發展性。

就軟體架構師身分而言，我屬於後者。我的專長在於組織原始碼，使其利於處理長期的商業目標。我寫的是我所知道的東西，所以如果這本書對架構師有用處，這將會是那種類型的架構師。

你不會發現關於 Architecture Tradeoff Analysis Method（ATAM，架構取捨分析法）、Failure Mode and Effects Analysis（FMEA，失效模式和影響分析）、服務探索（service discovery）等內容。那些架構不在本書的範圍之內。

本書組織方式

雖然這是一本關於方法論的書，但我還是以圍繞一個貫穿全書的程式碼範例的方式來組織它。我決定這樣做是為了使閱讀體驗比典型的「模式目錄（pattern catalogue）」更有說服力。這個決定的一個後果是，當實務做法和經驗法則適合這裡敘述的故事（narrative）時，我就引入它們。這也是我一般在指導團隊時介紹這些技巧的順序。

這個故事是圍繞著實作餐廳預訂系統的一個範例源碼庫而展開的。該範例源碼庫的原始碼可在 informit.com/title/9780137464401 取用。

如果你想把這本書作為手冊使用，我在附錄中列出了所有實務做法的清單，以及你可以在書中哪裡讀到更多資訊。

關於程式碼風格（Code Style）

範例程式碼是用 C# 編寫的，這是近年來迅速發展的一個語言。它從函式型程式設計（functional programming）中吸取了越來越多的語法思想；作為一個例子，在我撰寫這本書的時候，**不可變的記錄型別**（*immutable record types*）已經推出。我決定忽略其中一些新的語言功能。

很久以前，Java 程式碼看起來很像 C# 程式碼。而另一方面，現代的 C# 程式碼則看起來不太像 Java。

我希望這些程式碼能被盡可能多的讀者所理解。就像我從有 Java 範例的書中學到很多東西一樣，我希望讀者能夠在不了解最新的 C# 語法的情況下使用這本書。因此，我試著僅使用一個保守的 C# 子集，其他程式設計師應該也讀得懂它。

這並沒有改變書中提出的概念。是的，在某些情況下，有可能出現針對 C# 的更簡潔的替代方案，但這只是意味著額外的改進是可行的。

是否要用 var

var 關鍵字在 2007 年被引入到 C# 中。它能讓你宣告一個變數而不明確指定其型別。取而代之，編譯器會從情境脈絡中推斷出其型別。確切地說，用 var 宣告的變數和以明確型別宣告的變數都一樣是靜態定型（statically typed）的。

在很長一段時間裡，這個關鍵字的使用是有爭議的，但現在大多數人都在使用它；我也是如此，但我偶爾會遇到一些抗拒。

雖然我在工作環境中使用 var，但為一本書寫程式碼是一種稍微不同的背景。在正常情況下，IDE 都不遙遠。現代開發環境可以迅速告訴你一個隱含型別的變數之型別，但一本書卻不能。

由於這個原因，我偶爾會選擇明確地為變數指定型別。大多數範例程式碼仍然使用 var 關鍵字，因為它使程式碼更簡短，而在印刷書中行寬是有限的。但在少數情況下，我刻意選擇明確宣告變數的型別，希望在書中讀到程式碼時更容易理解。

程式碼列表

列出的大部分程式碼都來自同一個範例源碼庫，它是一個 Git 儲存庫（repository），而程式碼範例取自不同的開發階段。每個這樣的程式碼列表都包括相關檔案的相對路徑。該檔案路徑的一部分是 Git 的 commit ID（提交 ID）。

舉例來說，列表 2.1 包括這個相對路徑：*Restaurant/f729ed9/Restaurant.RestApi/Program.cs*。這意味著這個例子取自 f729ed9 這個 commit ID，而其檔案是 Restaurant.RestApi/Program.cs。換句話說，要查看這個特定版本的檔案，你得 check out 那個 commit：

```
$ git checkout f729ed9
```

當你完成這些後，你就能在其完整的可執行情境中探索 `Restaurant.RestApi/Program.cs` 檔案。

關於參考書目的說明

參考書目包含各種資源，包括書籍、部落格文章和影片記錄。我的許多資源都在線上，所以我當然提供了 URL。我已經努力將我有理由相信會在 Internet 上穩定存在的大部分資源包括在內了。

儘管如此，事情還是會改變。如果你在未來讀到這本書，而某個 URL 已經失效了，可以試試網際網路的封存服務（internet archive service）。在我撰寫這篇文章時，https://archive.org 是最好的選擇，但該網站也可能在未來消失。

引述我自己

除了其他資源，參考書目中還列出了我自己的作品。我知道，就敘述說明而言，參考自己的話本身並不構成一種有效的論證。

我將自己的作品納入，並非作為一種行銷手段。取而代之，我包括這些資源，是為了可能對更多細節感興趣的那些讀者。當我引用自己時，原因是你可能會在我所指出的資源中找到一個擴展過的論點，或更詳細的程式碼範例。

1
藝術或科學？

你是科學家還是藝術家？是工程師還是工匠？是園丁還是主廚？詩人還是建築師？

你是程式設計師或軟體開發人員嗎？ 如果是，那麼你是什麼？

我對這些問題的回答是：**是的，以上皆非**。

雖然我自認為是一名程式設計師，但我對上述的所有事情都有一點了解。然而，這些都不是我。

像這樣的問題是很重要的。軟體開發行業大約有 70 年的歷史，而我們仍在摸索中。一個長期存在的問題是如何**思考**這件事。因此有了這些問題。軟體開發就像蓋房子嗎？它像創作一首詩嗎？

幾十年來，我們嘗試過各式各樣的隱喻，但都不盡如人意。開發軟體就像蓋房子，但也不完全是。開發軟體就像栽培一個花園，但也不全然如此。歸根究柢，這些隱喻都不適合。

但我相信，我們對軟體開發的思考方式決定了我們如何工作。

如果你認為開發軟體就像蓋房子一樣，你會犯下錯誤。

1.1 建造房子

幾十年來，人們一直把開發軟體比作建造房子。正如 Kent Beck 所說的：

> 「**不幸的是，軟體的設計一直被源於物理設計活動的隱喻所束縛。**」[5]

這是軟體開發中最普遍、最誘人，也是最礙事的隱喻之一。

1.1.1 專案的問題

如果你認為開發軟體類似於建造房子，你會犯的第一個錯誤就是把它當成一個**工程專案**（*project*）。一個專案有開始和結束，一旦你到達終點，工作就完成了。

只有不成功的軟體才會完結。成功的軟體是持久續存的。如果你有幸開發出成功的軟體，當完成一個發行版本後，你會繼續開發下一個版本，這可以持續好幾年。一些成功的軟體會持續存在幾十年[1]。

一旦你建好了房子，人們就可以搬進去。你需要維護它，但其成本只是建造它的一小部分。誠然，像這樣的軟體是存在的，特別是在企業領域。只要你建好了[2]一個內部業務線（line-of-business）的應用程式，它就完成了，而使用者也就被束縛其中了；當專案完成，這樣的軟體就進入了維護模式。

但大多數軟體並不是這樣的。與其他軟體競爭的軟體是永遠都不會完成

1　我使用 LaTeX 來撰寫這本書，這個程式是在 1984 年發行的！

2　我熱切希望永遠都不使用關於軟體開發的這一個動詞，但在這個特定的背景之下，它是合理的。

的。如果你陷入建造房子的隱喻中，可能會把它想成是一系列的專案；你可能計畫在九個月內發行產品的下一個版本，但卻驚恐地發現你的競爭對手每三個月就發佈一次改良版。

所以你努力地使該「專案」的時程縮短；當你終於能夠每三個月交付一次時，你的競爭對手卻達到了每月一次的發行週期。你可以看到這個的結局，對吧？

它的結局是 Continuous Delivery（持續交付）[49]。就是這樣，或者你最終倒閉了。根據研究，*Accelerate* 一書 [29] 令人信服地論證道，區分高績效和低績效團隊的關鍵是不加思索，毫不猶豫就發佈的能力。

若能做到這點，軟體開發**專案**的概念就不再有意義了。

1.1.2 階段的問題

另一個因為建造房子的隱喻而產生的誤解是，軟體開發應該分為不同**階段**（*phases*）進行。在建造房屋時，建築師（architect）首先得繪製計畫，然後你準備後勤工作，把材料搬到現場，然後才能開始建造。

當這個隱喻套用在軟體開發之上，你會任命一個**軟體架構師**（*software architect*），他的責任是製作一個計畫；只有在計畫完成後，才可以開始進行開發。這種關於軟體開發的觀點認為，規劃階段（planning phase）是發生智識活動的階段。根據這個隱喻，程式設計階段（programming phase）就像房子的實際建造階段。開發人員被看作是可互換的工人[3]，基本上只不過是光榮的打字員而已。

3　我對建築工人完全沒有貶義；我親愛的父親就是一名砌磚工人。

沒有什麼比這更不符合事實的了。正如 Jack Reeves 在 1992 年指出的那樣 [87]，軟體開發的**建造**（*construction*）階段就是你編譯原始碼（compile source code）的時候。

那幾乎就像免費的，與房屋的建造相當不同。所有的工作都發生在**設計**（*design*）階段，或者就像 Kevlin Henney 生動表達的那樣：

> 「**以毫不含糊的細節描述一個程式和程式設計的行為本身是相同的一個整體**」

在軟體開發中，沒有施工階段可言。這並不代表規劃工作沒有用，但它確實表明，建造房屋的隱喻頂多只能算是無益的。

1.1.3 依存關係

當你建造房屋時，物理現實施加了限制。你必須首先打好地基，然後立起牆壁，只有這樣你才能蓋上屋頂。換句話說，屋頂依存於（depends on）牆壁，而牆壁又依存於地基。

這個隱喻讓人們誤以為他們需要管理依存關係（dependencies）。我有過這樣的專案經理，他們製作了精緻的甘特圖（Gantt charts）來規劃一個專案。

我和許多團隊合作過，他們中的大多數人在開始任何新的開發專案時都會設計一個關聯式資料庫綱目（relational database schema）。資料庫是大多數線上服務的基礎，而那些團隊似乎無法擺脫這樣的觀念：你可以在擁有資料庫之前，先開發一個使用者介面。

有些團隊從來沒有設法生產出一個可以運作的軟體。在他們設計了資料庫之後，他們發現需要一個**框架**（*framework*）來配合它。所以他們開始

重新發明一種物件對關聯式映射器（object-relational mapper），也就是那個「電腦科學的越南」[70]。

建造房子的隱喻是有害的，因為它誤導了你，讓你以一種特定的方式思考軟體開發。你會錯過那些沒有看到的機會，因為你的觀點與現實不一致。實際的軟體開發，以隱喻的方式來講，就是**可以**從屋頂開始。你會在本書的後面看到這個的一個例子。

1.2 栽培一個花園

建造房子的隱喻是錯誤的，但或許其他隱喻更有效。園藝的隱喻（gardening metaphor）在 2010 年代得到了越來越多的關注。Nat Pryce 和 Steve Freeman 將他們的優秀著作取名為 *Growing Object-Oriented Software, Guided by Tests* [36] 並非偶然。

關於軟體開發的這種觀點認為，軟體是一種活的有機體，必須被照顧、耐心栽植和修剪。這是另一個令人信服的隱喻。你有沒有感覺到一個源碼庫有自己的生命呢？

從這個角度來看待軟體開發可能是有啟發性的。至少它迫使你改變觀點，並可能因此動搖你「軟體開發就像蓋房子」的信念。

透過將軟體視為一種活的有機體，園藝的隱喻強調了修剪（pruning）。如果任其生長，花園就會雜草叢生，毫無章法。為了從花園獲得價值，園丁必須在照護和支援必要的植物的同時，透過殺死雜草來進行維護。將之轉譯到軟體開發中，這有助於我們把焦點放在**防止**程式碼腐敗的活動，如重構（refactoring）和刪除死碼（dead code）。

我覺得這個隱喻不像造房子隱喻那般有問題，但我仍然不認為它描繪了整體畫面。

1.2.1 什麼讓花園成長？

我喜歡園藝隱喻對於能夠對抗混亂的活動之強調。就像必須對花園進行修剪和除草一樣，你必須對源碼庫進行重構並償還技術債（technical debt）。

然而，關於程式碼的來源，園藝隱喻就說得很少。在一個花園裡，植物會自動生長。它們所需要的只是養分、水和陽光。另一方面，軟體並不會自行生長。你不能只是把電腦、薯片和軟性飲料扔進一個黑暗的房間，並期望軟體能從中長出。你會缺少一個重要的成分：程式設計師（programmers）。

程式碼是由某個人寫的。這是一個主動的過程，對此園藝的隱喻並沒有什麼可說的。你如何決定寫什麼，以及不寫什麼？ 你怎麼決定**如何**架構一段程式碼？

如果我們希望改善軟體開發行業，我們也必須解決這些問題。

1.3 邁向工程

還有其他關於軟體開發的隱喻。例如，我已經提到了**技術債**（*technical debt*）這個詞，它意味著會計師的觀點。我還談到了**撰寫**（*writing*）程式碼的過程，這表明它與其他類型的寫作（authoring）有相似之處。很少有隱喻是完全錯誤的，但也沒有完全正確的。

我特別針對建房的隱喻是有理由的。一個原因是它是如此普遍。另一個原因是它看起來錯得離譜，以致於無法挽救。

1.3.1 軟體作為一種工藝

很多年前我就得出結論，建造房屋的隱喻是有害的。一旦你捨棄了一個觀點，你通常會去尋找一個新的觀點。我在**軟體工藝**（*software craftsmanship*）中找到了它。

把軟體開發看作是一門工藝（craft），當成本質上的**技術工作**（*skilled work*），這似乎很有說服力。雖然你**可以**接受電腦科學的教育，但你沒有必要這樣做。我就沒有[4]。

身為一名專業的軟體開發人員，你所需要的技能往往是情境式的。學習這個特定的源碼庫之結構為何。學習如何使用那個特定的框架。忍受在生產環境中花費三天的時間來排除一個錯誤的折磨。類似這樣的事情。

做得越多，你就越熟練。如果待在同一家公司，在同一個源碼庫中工作多年，你可能會成為一個專業的權威，但如果你決定去別的地方工作，這對你有幫助嗎？

透過從一個源碼庫移到下一個源碼庫，可以學習得更快。嘗試一些後端開發（back-end development），做一些前端開發（front-end development）；也許嘗試一些遊戲程式設計（game programming），或一些機器學習（machine learning）。這將使你接觸到廣泛的問題，而這些問題將作為經驗積累起來。

這與歐洲古老的 *journeyman years*（**學徒期滿的職工年**）傳統驚人地相似。像木匠或屋頂工人這樣的工匠會在歐洲各地旅行，在一個地方工作一段時間後再去下一個地方。這種方式使他們接觸到解決問題的替代做

4 如果你想知道，我確實有一個大學學位。那是經濟學學位，但除了在丹麥經濟事務部（Danish Ministry of Economic Affairs）工作過之外，我從未使用過它。

法，使他們的手藝變得更好。

把軟體開發人員想成這樣是很有說服力的。*The Pragmatic Programmer* 一書的副標題正是 *From Journeyman to Master* [50]。

如果這是真的，那麼我們就應該相應地架構我們的行業。我們應該有學徒（apprentices），與師傅（masters）一起工作。我們甚至可以組織公會（guilds）。

如果這是真的，那就會是這樣。

軟體工藝是另一個隱喻。我覺得它很有啟發性，但當你用亮光照耀一個主題時，也會產生陰影。光線越亮，陰影越暗，如圖 1.1 所示。

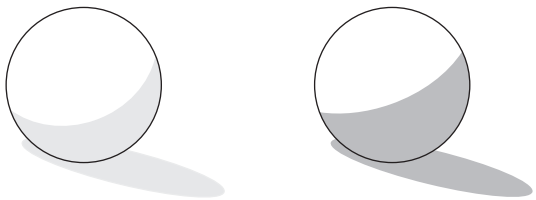

圖 1.1　你照在一個物體上的光越亮，陰影就越暗。

這畫面上還缺少了一些東西。

1.3.2 啟發式方法（Heuristics）

從某種意義上說，我的「軟體工藝」那幾年是一個徹底的幻滅期。當時我認為技能只不過是積累的經驗而已。在我看來，軟體開發是沒有方法論（methodology）的，一切都取決於環境。做事情的方法沒有對或錯。

也就是說，程式設計基本上是一種藝術（art）。

這很適合我。我一直都很喜歡藝術。當我年輕時，曾想成為一名藝術家[5]。

這種觀點的問題是，它似乎並不具有規模擴充性。為了「創造」新的程式設計師，你必須把他們當作學徒，直到他們學到足夠的知識，成為了學徒期滿的職工（journeymen）。從那時起，掌握技能還需要幾年的時間。

將程式設計視為一種藝術或工藝的另一個問題是，它也不符合現實。2010 年左右，我開始意識到 [106]，我在程式設計時遵循的是啟發式方法（heuristics），即經驗法則（rules of thumb）和一些指導方針（guidelines），這些都是可以教授的。

起初，我並沒有太注意到這點。然而，多年來，我經常發現自己處在指導其他開發者的位置上。當我那樣做的時候，我經常會制定以特殊方式編寫程式碼的理由。

我開始意識到，我的虛無主義可能是錯誤的。也許，指導方針可能是將程式設計變成一門工程學科的關鍵。

1.3.3 早期的軟體工程概念

軟體工程（software engineering）的概念可以追溯到 60 年代末[6]。它與當代的**軟體危機**（*software crisis*）有關，即人們逐漸意識到程式設計是很**困難**的。

5 我最古老的願望是成為一名歐洲傳統的漫畫家。後來，在我十幾歲的時候，我拿起吉他，夢想著成為一名搖滾明星。事實證明，雖然我喜歡繪畫和演奏，但我並不是特別有天賦。

6 這個詞可能出現得更早。我並不完全清楚，而且我當時並不在世，所以無法回想。不過，1968 年和 1969 年舉行的兩次北約（NATO）會議普及了軟體工程一詞，這一點似乎沒有爭議 [4]。

那時的程式設計師實際上對他們所做的事情有很好的把握。我們行業中的許多傑出人物都活躍在那些日子裡：Edsger Dijkstra、Tony Hoare、Donald Knuth、Alan Kay。如果你問當時的人們是否認為程式設計會在 2020 年代成為一門工程學科，他們可能會說是。

你可能已經注意到，我把軟體工程的概念當作一個理想的目標來討論，而不是日常軟體開發的一個事實。世界上有可能存在一些實際的軟體工程[7]，但根據我的經驗，大多數軟體開發是以不同的風格進行的。

並不是只有我覺得軟體工程仍然是一個未來的目標。Adam Barr 說得很好：

> **「如果你和我一樣，夢想有一天，軟體工程能以一種深思熟慮、有條不紊的方式進行研究，而提供給程式設計師的指導也能建立在實驗結果的基礎之上，而非根據搖擺不定的個人經驗。」**[4]

他解釋了軟體工程當初是如何走上正軌的，但後來發生的事情使其脫軌。Barr 說，當時所發生的意外就是個人電腦（personal computers）。它們創造了一代的程式設計師，他們在家裡自學程式設計。由於他們是在孤獨中擺弄電腦的，他們在很大程度上對已經存在的知識體系一無所知。

這種狀態似乎一直持續到今天。Alan Kay 就把這稱作一種**流行文化**（*Pop Culture*）：

> **「但流行文化對歷史抱有輕視的態度。流行文化主要關於身份認同和你在參與的感覺。它與合作、過去或未來無關：它是生活在當下。我認為大多數為錢寫程式碼的人也是如此。他們不知道 [他們的文化源自何處]」**[52]

7　NASA 似乎是軟體工程很有可能存在之地。

我們或許因為在軟體工程方面進展甚微，而浪費了五十年的時間，但我們可能在其他方面取得了進展。

1.3.4 隨著軟體工程往前邁進

工程師是做什麼的？工程師負責設計和監督事物的建造，從橋樑、隧道、摩天大樓和發電廠等大型結構，到微處理器等微小物體[8]。他們幫忙生產實體物品。

亞歷山德拉王后橋（Dronning Alexandrine's bridge），俗稱 *Mønbroen*。它於 1943 年建成，連接了丹麥的西蘭島（Sealand）和較小的默恩島（Møn）。

程式設計師不會那樣做。軟體是無形的。正如 Jack Reeves 所指出的[87]，由於沒有實物需要生產，所以建造過程幾乎是免費的。軟體開發主要是一種設計活動。當我們在編輯器中輸入程式碼時，就相當於工程師在畫圖紙，而非對應到工人在建構東西。

8 我有一個朋友，他受的教育是化學工程師。大學畢業後，他成為 Carlsberg 公司的一名釀酒師。工程師也釀製啤酒呢。

「真正」的工程師遵循通常會導致成功結果的方法論。這也是我們程式設計師想要做的，但我們必須小心翼翼地只複製那些在我們的環境下有意義的活動。當你設計一個實體物品，真正的建造是很昂貴的。我們不能建造一座橋，測試一段時間後才決定它不好，然後把它拆掉重新開始。因為現實世界的建造過程是昂貴的，工程師們從事計算和模擬。計算一座橋的強度比建造它需要更少的時間和材料。

有一個完整的工程學科與物流（logistics）有關。人們從事細緻的規劃，因為那是建造實物最安全且最不昂貴的方式。

那是工程中我們**不需要**複製的部分。

但還有很多其他的工程方法論可以啟發我們。工程師也做創造性、人性化的工作，但那往往是在一個框架內結構化進行的。特定的活動之後應該有其他的活動。他們對彼此的工作進行審查和簽字。他們遵循檢查表（checklists）[40]。

你也可以那樣做。

那就是這本書的內容。它是我發現有用的啟發式方法之導覽。恐怕這更接近於 Adam Barr 所說的**搖擺不定的個人經驗**（*the shifting sands of individual experience*），而不是一套有科學根據的定律。

我相信，這反映了我們行業的現狀。認為我們對任何事情都有堅定的科學證據的人都應該讀一讀 *The Leprechauns of Software Engineering*[13]。

1.4 結論

如果你考慮一下軟體開發的歷史，你可能會想到數量級規模的進步。然而，這些進步中有許多是硬體的進步，而非軟體的進步。儘管如此，在過去的五十年裡，我們還是見證了軟體開發的巨大進展。

今日，我們擁有比五十年前先進得多的程式語言，可以存取 Internet（包括以 Stack Overflow 形式出現的業界標準線上說明）、物件導向和函式型程式設計（functional programming），自動化的測試框架、Git、整合式開發環境，等等。

另一方面，我們仍在**軟體危機**中掙扎，儘管已經持續了半個世紀的東西是否可以被稱為危機是值得商榷的。

儘管做出了認真的努力，軟體開發行業仍然不像是一門工程學科。工程和程式設計之間存在著一些根本性的差異。除非我們理解這一點，否則我們無法取得進展。

好消息是，你可以做到許多工程師做的事情。有一種思維方式，以及一系列你可以遵循的流程。

正如科幻作家 William Gibson 所說：

> **「未來已然到臨，只是分佈得不是很均勻」**[9]

正如 *Accelerate* 一書所描繪的那樣，一些組織今天就使用了先進的技術，而其他組織則落在後面 [29]。未來確實是不均勻分佈的。好消息是，那些先進的理念可以自由取得。是否要開始使用它們，由你決定。

在第 2 章中，你將首次領略到你可以進行的具體活動。

9 這是那些來源不明的引言之一。這個想法和整體措辭是 Gibson 的，這似乎沒有爭議，但他究竟是在什麼時候第一次這樣說，就不清楚了 [76]。