對本書的讚譽

Kent Beck 的建議總是值得聆聽；幾十年來我一直在等待本書所提出的一些建議。本書把軟體的焦點，從各種不同的工具與技術，轉移到真正重要的東西——設計！設計就是我們用程式碼所繪製出來的形狀，*Kent* 則幫助我們繪製出更好的形狀。這是一本談論重要主題的重要書籍。

—*Dave Farley*，
Continuous Delivery 有限公司創辦人兼董事

許多程式碼庫真的很難理解，開發人員實在很難知道該從哪裡下手。本書針對各種不同程度的開發人員，提供了許多實用的技巧，可協助大家改進手邊正在處理的各種程式碼。

—*Sam Newman*，獨立顧問、技術專家，
同時也是《*Building Microservices*》（打造微服務）和
《*Monolith to Microservices*》（從龐然大物到微服務）這兩本書的作者

對於如何把複雜的程式碼轉化成更簡單的形式，*Kent Beck* 分享了許多簡單易懂的想法。這些想法其實都很簡單，但當你讀到這些想法時，心裡一定很想知道，為什麼這麼多的想法自己從來都沒想過。我要推薦這本書，給所有在意程式碼乾不乾淨、可讀性好不好的人。

—*Gergely Orosz*，*The Pragmatic* 公司的工程師

幾十年來，重構相關書籍一直都把重點聚焦於從上而下、以物件為導向的軟體設計理論。《先整理一下？》則打破常規，透過逐步改善現有程式碼的方式，提供了一套很實際的做法。

—*Maude Lemaire*，《*Refactoring at Scale*》（大規模重構）一書的作者

推薦序

這本薄薄的書，其實是整個系列中的第一本，目標讀者是專業程式設計師——這類軟體開發者對於自己的技術有著深厚而專業的興趣，希望可以透過小小的方法來改進自己的工作，以取得巨大的回報。作者 Kent Beck 是一位非常敬業的專業人士，他一直關注著各種細節，而且始終在思考著一些更宏大的問題，拓展更寬廣的視野。

從業的軟體開發人員通常很少關注理論，但 Kent 很清楚知道自己在說什麼，他把實務和理論結合成一本既有可讀性又很實用的程式碼整理（tidy）指南。

理論上來說，理論與實務應該沒有區別，但實務上來說，確實有區別。這個精闢的看法，廣泛流傳著各種不同的版本，有人甚至誤以為這是愛因斯坦或 Yogi Berra（洋基隊傳奇捕手）所說過的話。只有那種愛鑽牛角尖的文字大師（沒錯，這是有罪的！）才會在意正確的出處；其實這段話是出自耶魯大學的學生 Benjamin Brewster 在 1882 年版《*Yale Literary Magazine*》（耶魯文學雜誌）所寫的文章。多虧那些來自 QuoteInvestigator.com 的熱心文字極客們，我才能在這裡滿懷信心向讀者提供這個極客細節——他們的專業堅持，就是把細節做對重於一切。

為了把理論與實務相結合，Kent 從最底層的小小程式碼片段開始著手，一絲不苟地處理每一個小細節，然後再逐步提升到更大的視角，解釋如何創建出更清晰簡潔的程式碼，讓這些程式碼在面對無可避免的改變與修正時，能夠有更穩健可靠的處理方式。Kent 在編寫這份實務指南時，最後還借鑒現實世界裡的軟體開發經濟學，進一步與軟體工程的核心理論相互結合。

這個核心理論其實很簡單：電腦程式碼的複雜性，取決於如何把程式碼組織成好幾個部分，以及各部分之間的耦合程度，還有各部分自身的內聚程度。耦合（coupling）和內聚（cohesion）理論的源頭，通常都會引用 Ed Yourdon 和我所合著的《*Structured Design*》

（**結構化設計**，由 Yourdon Press 於 1975 出版；Prentice Hall 於 1979 再次出版），不過再往前也可以追溯到 1968 年在麻薩諸塞州劍橋所舉行的一次會議演講。「耦合」和「內聚」這兩個概念，在 1979 年的 Prentice Hall 版本差一點就被拿掉了。當時的編輯很想說服 Ed 和我拿掉這兩章，因為他們說「不會有人對理論感興趣的啦」。所幸這段軟體工程的歷史，作者還是占了上風，當時那些編輯確實想錯了。在此之後，經過半個世紀的實務驗證和好幾百項研究調查，這個理論終於得到了驗證。

耦合和內聚其實就是衡量電腦程式碼複雜性的一種方式，只不過它並不是從執行程式的電腦角度來衡量，而是從人類嘗試去理解程式碼的角度來衡量。如果想理解任何程式，不管是新建立的程式、還是改變或修正舊程式碼，你都必須仔細去瞭解你面前的片段程式碼，以及它所相連、所依賴、所影響或受其影響的其他片段程式碼。如果有某段程式碼本身完全緊密而連貫，可以視為一個整體來理解，就可以構成認知心理學家所說的「完形」（gestalt），要理解它也會比較容易一點。這就是所謂的內聚（cohesion）。如果與其他的程式碼片段沒什麼關係，或是關係相對比較薄弱，或是有嚴格的拘束，這樣理解起來也會比較容易。這就是所謂的耦合（coupling）。耦合和內聚與你的大腦如何處理複雜系統，其實是很有關係的。

這樣你懂了嗎？把概念整理之後，其實還蠻漂亮的。理論這東西就是這樣。接下來就可以進一步討論實際的細節了；其間還是會夾雜一些足夠的理論，讓一切看起來更加合理。Kent Beck 會很巧妙引導著你，循著這條道路前進。

—Larry Constantine
麻薩諸塞州的羅里鎮（Rowley）
2023 年 10 月 9 日

Larry Constantine 曾任葡萄牙 Madeira 大學和澳洲雪梨科技大學的教授。他發表了 200 多篇論文以及 30 幾本書籍，其中包括與 Lucy Lockwood 合著的 Jolt 獎作品《*Software for Use*》（**真正實用的軟體**；Addison Wesley，1999 年），以及他用 Lior Samson 這個筆名所寫的 15 部小說。

前言

《先整理一下？》要談的是……

「我必須修改這段程式碼，但它根本就一團亂。我應該先做什麼好呢？

「也許我應該在修改之前，先整理一下程式碼。也許吧。整理一下下就好。還是我不應該這麼做？

這些全都是你很可能會問自己的問題，如果答案很簡單，我就不必寫一本書來處理這些問題了。

《先整理一下？》這本書要談的是：

- 除了改變程式碼的行為之外，何時應該去整理那一團亂的程式碼
- 如何安全又有效率地整理凌亂的程式碼
- 「整理凌亂的程式碼」這件事，什麼時候該停手
- 為什麼整理是有意義的

軟體設計其實是人類關係的一種課題。在《先整理一下？》這本書中，我們首先會從你鏡子裡的那個人開始，談談程式設計師與自己的關係。為什麼我們不多花點時間，好好照顧自己呢？花點時間讓你的工作變得更輕鬆，不是很好嗎？我們何不先把協助使用者的工作擺到一旁，先跳進這個清理程式碼的兔子坑裡瞧一瞧呢？

我的任務就是協助極客們，能夠在這個世界感到更安心，而《先整理一下？》這本書，就是此任務的一部分。這也是我們面對凌亂的程式碼時，可以邁出的第一步。如果懂得好好善用軟體設計，它其實是一個能夠減輕世界痛苦的強大工具。但如果使用不當，它反而會成為另一種壓迫的工具，到頭來只會拖垮軟體開發的效率。

《先整理一下？》這本書只是我們聚焦於軟體設計，整個系列其中的第一本書籍。我想讓軟體設計變得更加平易近人、更有價值，所以我會先從你自己就能完成的軟體設計開始談起。後續幾本書則會探討軟體設計如何修復團隊裡程式設計師們之間的關係，然後再去解決真正大條的問題：業務和技術之間的關係。不過，一開始本書會先從更有利於我們自己日常工作的角度，來理解與實現軟體設計。

假設你有一個很大的函式，其中包含了許多行的程式碼。在進行修改之前，你一定要先讀懂程式碼，搞清楚它究竟在做什麼。在這個過程中，你可以看到自己會按照一定的邏輯，先把程式碼切成更小的區塊。你在提取出這一塊一塊的程式碼時，其實就是在進行整理。另外像是採用守衛語句（guard clause）、添加一些具有解釋效果的註解說明、使用輔助函式等等，也都屬於整理的做法。

《先整理一下？》這本書會把這些建議付諸實現——我們會把這些整理方式分開來細談，並建議你在何時何地可以採用這些整理的做法。因此，你並不需要一次就掌握所有的整理做法，只要先去嘗試一些對你而言有意義的整理做法就行了。《先整理一下？》這本書也會探討一些軟體設計背後的理論：耦合（coupling）、內聚（cohesion）、現金流的折現效應（discounted cash flows），以及所謂的選擇性（optionality）。

本書適用對象

本書很適合程式設計師、首席開發人員、實務軟體架構師以及技術管理者來閱讀。本書並不限定任何程式語言，所有的開發人員都能讀懂本書的概念，並把它應用到自己的專案中。本書假設讀者具有一般的程式設計能力，並不是完全的新手。

本書帶給你的收穫

讀完本書之後，你就會瞭解：

- 系統的「行為」改變與「結構」改變，兩者之間的根本區別
- 身為一個負責修改程式碼的孤獨程式設計師，如何發動你的魔法，讓你可以自由切換，去進行程式碼「結構」與「行為」上的改變

- 關於軟體設計的運作方式，以及影響軟體設計的力量，相應的理論基礎

而且你會得到以下的能力：

- 知道何時該先做整理、何時該後做整理，改善自己的程式設計體驗。
- 開始懂得運用一些比較小而安全的步驟，做出比較大的改變。
- 可以把設計視為一種包含各種不同動機的人類活動。

本書架構

《先整理一下？》的內容分成簡介和三大部分：

簡介

我會先簡單說明一下寫這本書的動機與原委，還有這本書的目標讀者，以及你可以期待的內容。然後我們就會直接進入主題。

第一部分：「整理」

整理就像是嬰兒版的微型重構。每一章的內容都很簡短，分別探討一種整理的做法。內容架構大致上就是：如果你看到「這樣」的程式碼，就把它改成「那樣」的程式碼。然後不用想太多，就可以讓它正式上線使用了。

第二部分：「管理」

接著我們會介紹整理的管理方式。關於整理的哲學，其中有一點就是，它永遠都不應該是個大問題。它絕不是那種必須回報、追蹤、計劃和排程的工作。你只是因為要修改程式碼，但發現程式碼很亂很難修改，所以你才決定先整理一下。即使這是日常工作的一部分，但它依然是一個需要好好思考如何進行改進的程序。

第三部分：「理論」

來在這裡，我終於可以展開翅膀，深入挖掘一些讓我感到很興奮的主題。「軟體設計其實是人類關係的一種課題」這句話是什麼意思呢？人類指的是誰呢？如何透過更好的軟體設計，更加滿足人們的需求？為什麼軟體的成本如此昂貴？我們有什麼能做的嗎？（劇透警告：就是軟體設計）耦合是啥？內聚是啥？冪次律（power law）又是什麼鬼東西？

我的目標就是讓讀者早上開始讀，下午就能做出更好的設計。然後每天都能把設計做得更好一點。很快地，軟體設計就不再是你的軟體價值傳遞鏈其中最薄弱的環節了

用同樣的寫法做同樣的事（Normalize Symmetries）

程式碼就像有機體，會不斷的成長。有些人會把「有機」視為一個貶義詞。對我來說，這實在不太合理。我們不太可能一次就編寫出自己所需要的全部程式碼。除非我們已經不再學習任何東西，這樣才有可能做到這種事。

在這樣的有機成長過程中，同樣的問題在不同的時間遇到不同的人，可能就會以不同的方式解決。這樣並沒有什麼問題，不過這樣確實會讓程式碼變得比較難讀懂。從閱讀程式碼的角度來看，你應該還是希望看到比較有一致性的寫法。因為這樣一來你只要看出程式碼裡出現某種特定的寫法，就可以很有自信直接跳到結論，因為你已經很清楚來龍去脈了。

舉個延遲初始化變數（lazily initialized variable）的例子。你可能看過好幾種不同的寫法：

```
foo()
    return foo if foo not nil
    foo := ...
    return foo

foo()
    if foo is nil
        foo := ...
    return foo

# 這個寫法要稍微想一下
foo()
    return foo not nil
```

```
        ? foo
        : foo := ...

# 這個寫法要多想兩下，假設賦值語句本身就是個表達式
foo()
    return foo := foo not nil
        ? foo
        : ...

# 這個寫法要多想好幾下，因為連條件判斷都沒了
foo()
    return foo := foo || ...
```

（再想想看，你能不能再發明或找出更多的變體寫法。）

所有這些不同的寫法，全都是在說「如果我們還沒計算並快取過 foo 的值，那就去進行計算並快取起來」。每一種寫法都有其優缺點。從讀者的角度來看，你應該很快就能適應其中的任何一種寫法。但如果有兩種或好幾種寫法交替使用，整件事就會變得很混亂。從讀者的角度來思考，如果你看到不同的寫法，心裡應該就會預期，那應該是代表不同的意思。但是這裡的不同寫法，反而會掩蓋掉「其實都是在做同一件事」的事實。

請你選定一種寫法就好。然後再把不同的變體寫法，全都轉換成你所選定的寫法。像這種非必要的變化形式，一次只需要整理一種就行了——舉例來說，你可以先針對「延遲初始化」，進行統一的整理。

有時候，有些額外的細節，反而會把一些共通點隱藏起來。你可以先把一些很相似但不完全相同的常式全部找出來。然後再嘗試把其中不同的部分，與相同的部分拆分開來。

第十一章

把程式碼切成一塊一塊的

本章的整理做法，可以說是最簡單的一種整理做法。你在讀一大段程式碼時，慢慢就會意識到，「哦，這個部分在做的是**這件事**，然後那個部分在做的是**那件事**。」只要在各個部分之間，放個空白行就行了。

我超級喜歡這種超級簡單的整理做法。這其實也是「先整理一下？」哲學的一部分——別讓軟體設計變成一件大事，否則你可能就不會去做了。軟體設計可以促進改變。軟體設計上小小的做法，就可以稍微讓改變變得更容易。

這其中還有一個很酷的東西——複利效應。軟體設計本身，也會讓更多的軟體設計變得更容易。這既是一種祝福，也是一種詛咒。你有可能會陷入設計的漩渦，而忘了原本是想去做改變。千萬別這樣。如果做得好，軟體設計應該要能夠促成那種真正會實現改變的軟體設計。

把程式碼切成一塊一塊之後，你還有許多道路可以繼續前進，例如具有解釋效果的變數（第 8 章）、提取輔助函式（第 12 章）、或是具有解釋效果的註解說明（第 14 章）。

第十五章

刪除掉多餘的註解

如果你看到一段註解說明，說的東西與程式碼本身完全相同，就把它刪除掉吧。

程式碼的目的，就是向其他的程式設計師解釋，你想讓電腦做什麼事。針對你這個寫程式碼的人，以及未來讀程式碼的人，註解說明與程式碼各自做出了不同的取捨。你可以用散文的形式，自由說明任何你想要解釋的東西。但另一方面，雖然狀況會改變，可是卻沒有任何機制可以幫你檢查註解說明的正確性，而隨著程式碼的發展，註解說明很有可能就會變得很多餘。

對於註解說明的溝通功能，有些人抱持著很狹隘的看法，堅持一定要遵守某一些教條式的規則，例如規定每個常式都一定要寫註解說明。正因為如此，所以才會出現下面這類的註解：

```
getX()
  # 送回 X
  return X
```

像這樣的註解說明，只有成本的付出，卻不會帶來任何好處。身為寫作者，你這樣只是在浪費讀者的時間而已——那些時間是拿不回來的。如果是完全多餘的註解說明，刪除掉就對了。

整理工作經常都會一個串一個、形成連鎖的效果。之前所做的整理，可能會讓某些註解說明變得很多餘。舉例來說，原本的程式碼可能像下面這樣：

```
if (generator)
    ... 對 generator 進行設定的一堆程式碼 ...
else
    # 沒有 generator，直接送回預設值
    return getDefaultGenerator()
```

用守衛語句來進行過整理之後，程式碼就會變成下面這樣：

```
if (! generator)
    # 沒有 generator，直接送回預設值
    return getDefaultGenerator()
... 對 generator 進行設定的一堆程式碼 ...
```

原本那行註解說明並不算多餘。它可以讓我們在讀完一堆不同背景條件（有 generator，需要進行設定）的整段程式碼之後，重新把注意力拉回到另一個背景條件（沒有 generator）的情況。不過在整理過之後，那行註解說明就會變成只是把程式碼的作用再簡單重新說一遍而已。所以，把它刪除掉就對了。再見，莎喲娜啦，慢走不送啦。

我們到了本書第二部分，還會再談談整理的這種連鎖效應。

把整理工作切分出來

我們暫時假設，你在寫程式的時候，採用的是拉取請求（PR） / 程式碼審查的開發模式（稍後我也會提出另一種替代做法）。你會把整理這件事，放在哪個位置呢？

下面就是一個「追著尾巴跑」（tail chasing）的醜陋做法：

1. 我把整理工作與行為上的改變混在一起。
2. 審查者向我抱怨，說我的 PR 拖得太長了。
3. 我把整理工作切分出來，讓它擁有自己的 PR，然後把它放在行為改變的前面（這種情況居多）或後面。
4. 審查者向我抱怨說，這種整理類 PR 根本就沒意義。
5. 回到 1。

整理這件事總要有個地方去做，要不然就是完全不做。究竟應該在哪裡做呢？直接跳到結論的話，就是：放到獨立的 PR 中，然後每個 PR 所做的整理盡量越少越好。

我們就來深入探討一下其中的取捨。剛開始學習如何做整理的人，好像都會經歷好幾個可預期的不同階段。在第一個階段，我們只懂得做出各種改變，而且一開始這一大堆的改變，並沒有什麼特別的區別（圖 16-1）。

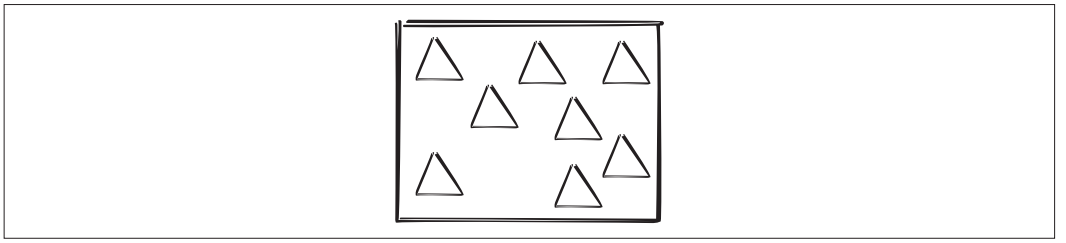

圖 16-1　一大堆的改變，沒什麼特別的區別

假設我們正在修正一個 `if` 語句，修到一半發現有個名稱錯了，於是就順手把它修正過來，然後再回頭繼續修正這個 `if` 語句。在這個階段，改變就只是改變而已。

學會整理的做法之後，感覺就好像顯微鏡底下的圖片焦點變清楚了。其中有一些改變，改變的是程式的行為屬性，也就是程式執行過程中可以觀察到的一些改變。另外還有一些改變，改變的是程式的結構。像這樣的改變，只有直接去查看程式碼才能看得出來：如圖 16-2 所示，B = 行為（Behavior），S = 結構（Structure）。

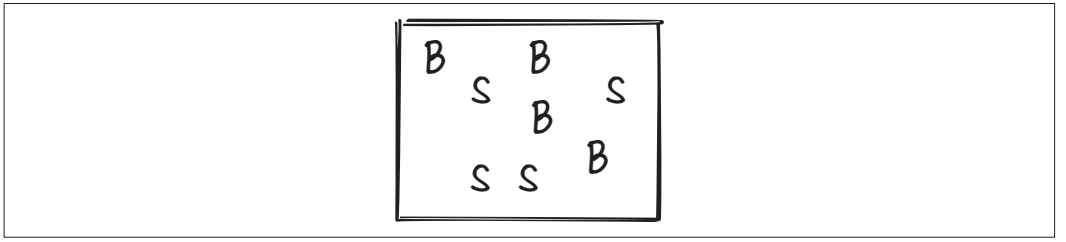

圖 16-2　行為上的改變和結構上的改變

在這個階段，我們對行為上的改變和結構上的改變，還沒有什麼特別的想法和做法——我們只是意識到，可以把改變分成兩種而已。

經過一段時間之後，我們就可以開始察覺出一些共通的流程。如果先把程式碼切成一塊一塊的，接下來經常就可以提取出一些具有解釋效果的輔助函式，再接下來我們就會更容易去做出行為上的改變。如此一來程式設計這件事，就變得更像是下一盤西洋棋，你可能在好幾步之前，就可以猜出整盤棋大概會怎麼發展了（圖 16-3）。

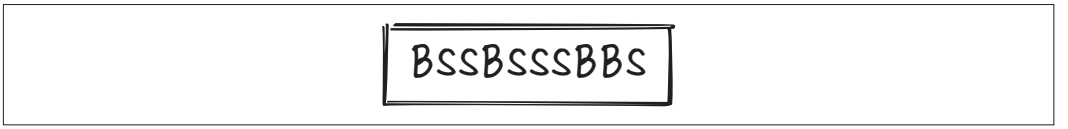

圖 16-3　行為上和結構上的改變，依序排列

請注意，這時候我們還只是用了一個很大的 PR，把所有改變全都包在一起。目前我們還只是處在「追著尾巴跑」這個循環的第 1 步而已。我們所做的每個行動都有其目的，要不是為了做出某個簡單的改變，要不就是為了更容易做出改變。不過，把所有這些全都混在一起，實在有點亂。審查者會抱怨，也是很正常的事。

於是，我們開始嘗試把改變區分成兩種獨立的 PR。一系列的整理（甚至只是一次小小的整理），會被放進一個獨立的 PR。行為上的改變，也會放進一個獨立的 PR。每次在「整理」和「行為改變」之間進行切換時，我們都會開一個新的 PR（圖 16-4）。

圖 16-4　行為上的改變和結構上的改變，會分別放進獨立的 PR

PR 應該如何整併或拆分，這也是一件需要權衡取捨的事情。你可以從動機的角度來思考。一個比較大的、包羅萬象的 PR，可以呈現出比較完整的情況，但對審查者來說，其中所包含的改變可能太多了，這樣就很難提供真正有用的回饋意見。比較小的 PR 則可以讓回饋意見限縮在比較小的範圍，但是也有可能讓審查者陷入瑣碎的枝末細節中。

審查過程如果拖得比較長，也會影響到大家的動機。如果程式碼審查的速度很快，就等於是鼓勵你多去建立一些更小的 PR。這種更有針對性的 PR 也會鼓勵審查者加快審查。同樣的道理，這個循環也可能反過來：比較慢的審查速度，只會鼓勵 PR 變得更大，而更大的 PR，也會拖慢後續的審查。

如果你可以做出讓人很放心的整理，懂得用小小的步驟去進行整理，而且會用絕對安全的方式進行整理，這樣我就會鼓勵你去嘗試，讓這種整理類的 PR 不再需要進行審查。這樣就可以進一步降低延遲的影響，從而激勵大家多去提交一些比較小的整理類 PR。