《重構》

Refactoring - Improving the Design of Existing Code 中文版

■敬告讀者

此處開放《重構》最後定稿之部分篇幅。此舉得到 繁體版出版人碁峰圖書公司 與簡體版出版人中國電力出版社 之鼎力支持，十分感謝。

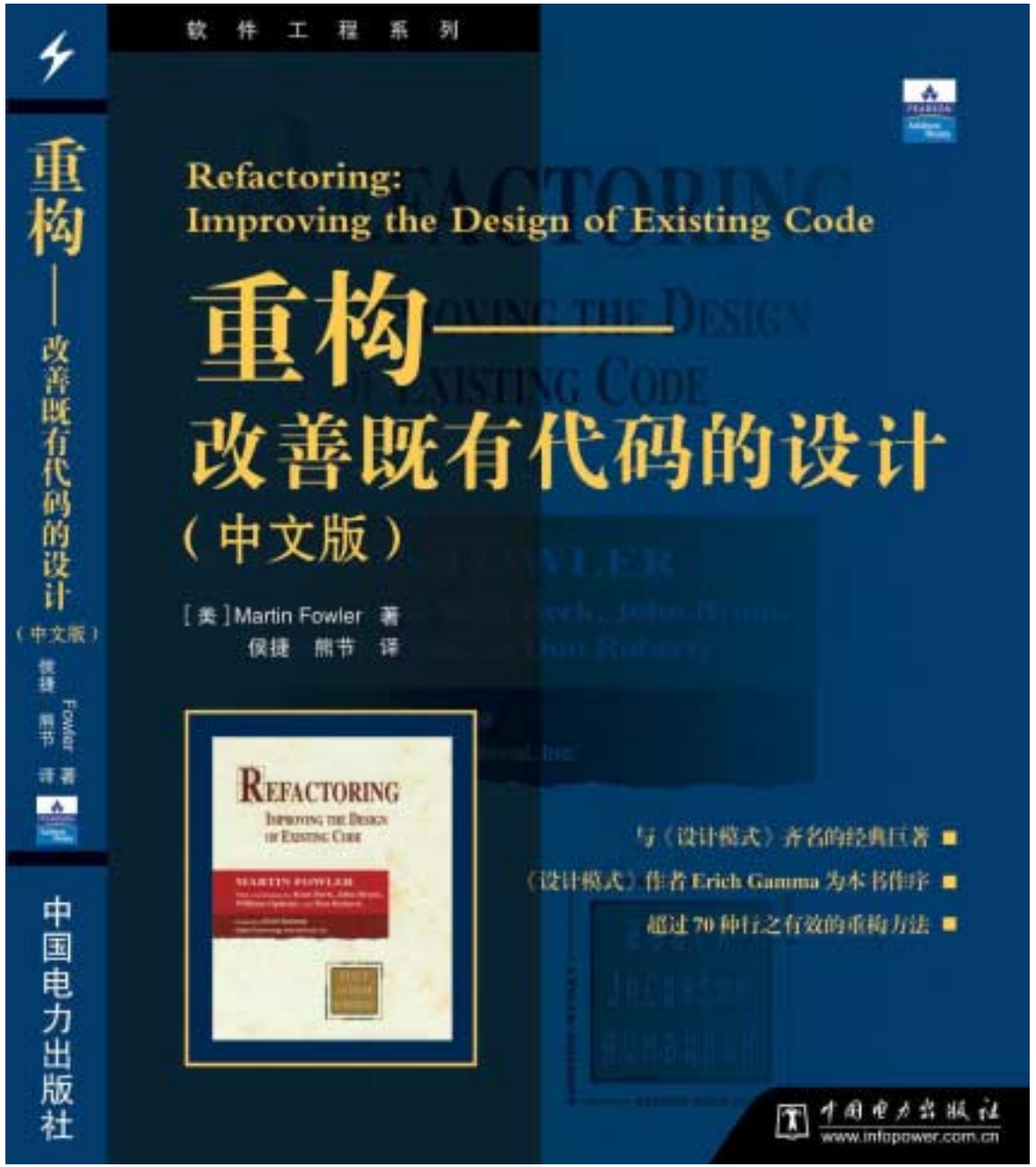
本次開放第 6 章（含）前所有內容及書後索引，達全書 1/3+ 篇幅。開放之 PDF 含標籤（目錄連結；只達「章」層級），請打開 PDF reader 之「導引框」，便可見到如下畫面：

請注意，標籤（目錄）完整，但內容只有前 6 章及索引。

本次開放以 繁/簡中文 讀者爲對象。由於我個人並不直接處理簡體版最終版面工作，因此手上無簡體版最終電子成品。我所開放的兩份成品，都使用繁體字，但區分爲「臺灣術語版」和「大陸術語版」。

您目前所見到的這一份成品，是「臺灣術語版」。enjoy it ☺

侯捷.2003/07/12


软 件 工 程 系 列
重构——改善既有代码的设计
（中文版）
侯捷 熊节 译 著
Martin Fowler
中国电力出版社
Refactoring:
Improving the Design of Existing Code
重构——
改善既有代码的设计
（中文版）
[美]Martin Fowler 著
侯捷 熊节 译
REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
与《设计模式》齐名的经典巨著
《设计模式》作者Erich Gamma为本书作序
超过70种行之有效的重构方法
中国电力出版社
www.infopower.com.cn

Refactoring: Improving the Design of Existing Code

重构——改善既有代码的设计（中文版）

当对象技术成为老生常谈之后——尤其在 Java 编程语言之中，新的问题也在软件开发社区中浮现了出来。缺乏经验的开发人员完成了大量粗劣设计，获得的程序不但缺乏效率，也难以维护和扩展。渐渐地，软件系统专家发现，与这些沿袭下来的，质量不佳的程序共处，是多么艰难。对象专家运用许多（而且日益更多）技术来改善既有程序的结构完善性与性能，已有数年之久，但是这些被称为“重构”（refactoring）的实践技术，一直（只）流传于专家领域内，因为没有人愿意将全部这些知识落写为所有开发人员可读的形式。这种情况如今终于结束。在本书中，知名的对象技术者 Martin Fowler 闯入新的领域，褪去晦涩名家实践手法的神秘面纱，展示软件从业人员领悟这种新过程的重大意义。

只要受过适度训练，一位技巧娴熟的系统程序员可以在拿到一个糟糕的设计之后，把它翻新为设计良好、稳健搞笑的代码。本书之中，Martin Fowler 告诉你重构机会通常可以在哪里找到，以及如何将一个糟糕的设计重新修订为一个良好的设计。每个重构步骤都十分简单——简单到了似乎不值得去做的程度。重构涉及将值域（field）从一个 class 搬移到另一个 class，或将某些代码拉出来就立为另一个函数（method），或甚至将某些代码上下移动于继承体系（hierarchy）之中。这每个别步骤虽然可能十分基本，积累下来的影响却能够彻底改善设计。重构已经被证明可以防止软件的腐朽与衰败。

除了讨论各式各样的重构技术，作者还提供了一份详细名录（catalog），其中有超过 70个已被证明效果的重构手法，以及宣布动的重点，看导你实施的时机，实施时的逐步指令，并各自携带一个例子，显示重构的运转。这些富有良好解说价值的实例都以 Java 写就，其中的观念适用于任何面向对象编程语言。

Martin Fowler 是一位独立咨询顾问，他运用对象技术解决企业问题已经超过 10 年。他的顾问领域包括健康管理、金融贸易，以及法人财务。他的客户包括 Chrysler, Citibank, UK National Health Service, Andersen Consulting, Netscape Communications。此外 Fowler 也是 objects、UML、patterns 技术的一位合格讲师，他是《Analysis Patterns》和《UML Distilled》的作者。

Kent Beck 是一位知名的程序员、测试员、重构员、作家、五弦琴专家。

John Brant 和 **Don Roberts** 是《Refactoring Browser for Smalltalk》的作者，此书可从 http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser 获得。他们两人也是咨询顾问，研究重构的实践与理论有六年之久。

William Opdyke 在伊利诺斯大学所做的 object-oriented frameworks（面向对象框架）博士研究，导出了重构领域的第一份重要出版品。他目前是 Lucent Technologies/Bell Laboratories 的一名卓越技术人员。

译者侯捷，致力计算面技术教育超过 10 年——以著作、翻译、评论、专栏、授课等多重形式，对于各种层级、各种定位、各种技术领域之 Framework Libraries 有深刻兴趣和钻研。

译者熊节，普通程序员，喜编程，乐此而不疲。酷爱读书，好求新知，记性好忘性大，故凡有所得必记诸文字，有小得，无大成。闲有点墨，心无大志，惟愿宁静流泊而已。夜阑人静，一杯清水，几本闲书，神交于各方名士，敝膝于天下同好，百愿足矣。

中文版（本书）支持网站：http://www.jjhou.com（繁体）http://jjhou.csdn.net（简体）

责任编辑/程 驰 乔 杰
封面设计/王红卿

www.PearsonEd.com

ISBN 7-5083-1554-5
定价：68.00 元

Refactorings（重構）列表

重構

— 改善既有程式的設計 —

Refactoring

Improving the Design of Existing Code

Martin Fowler 著

（以及 Kent Beck, John Brant , William Opdyke,
Don Roberts 對最後三章的貢獻）

侯捷 / 熊節 合譯

譯序

by 侯捷

看過鐵路道班工人嗎？提著手持式砸道機，機身帶著鈍鈍扁扁的鑽頭，在鐵道上、枕木間賣力地「砍劈鑽鑿」。他們在做什麼？他們在使路基上的碎石塊（道碴）因持續劇烈的震動而翻轉方向、滑動位置，甚至震碎爲更小石塊填滿縫隙，以求道碴更緊密契合，提供鐵道更安全更強固的體質。

當「重構」（refactoring）映入眼簾，我的大腦牽動「道班工人+電動砸道機+枕木道碴」這樣一幅聯想畫面。「重構」一詞非常清楚地說明了它自身的意義和價值：在不破壞可察功能的前提下，藉由搬移、提煉、打散、凝聚…，改善事物的體質。很多人認同這樣一個信念：「非常的建設需要非常的破壞」，但是現役的應用軟體、構築過半的專案、運轉中的系統，容不得推倒重來。這時候，在不破壞可察功能的前提下改善體質、強化當前的可讀性、爲將來的擴充性和維護性做準備、乃至於在過程中找出潛伏的「臭蟲」，就成了大受歡迎的穩步前進的良方。

作爲一個程式員，任誰都有看不順眼手上程式碼的經驗 — 程式碼來自你鄰桌那個菜鳥，或三個月前的自己。面臨此境，有人選擇得過且過；然而根據我對「程式員」人格特質的了解，更多人盼望插手整頓。挽起袖子劍及履及，其勇可嘉其慮未縝。過去或許不得不暴虎憑河，忍受風險。現在，有了嚴謹的重構準則和嚴密的重構手法，「穩定中求發展」終於有了保障。

是的，把重構的概念和想法逐一落實在嚴謹的準則和嚴密的手法之中，正是這本《*Refactoring*》的最大貢獻。重構?! 呵呵，上進的程式員每天的進行式，從來不新鮮，但要強力保證「維持程式原有的可察功能，不帶進新臭蟲」，重構就不能是一項靠著天份揮灑的藝術，必須是一項工程。

我對本書的看法

初初閱讀本書，屢屢感覺書中所列的許多重構目標過於平淡，重構步驟過於瑣屑。這些我們平常也都做、習慣大氣揮灑的動作，何必以近乎枯燥的過程小步前進？然後，漸漸我才體會，正是這樣的小步與緩步前進，不過激，不躁進，再加上完整的測試配套（是的，測試之於重構極其重要），才是「不帶來破壞，不引入臭蟲」的最佳保障。我個人其實不敢置信有誰能夠乖乖地按步遵循實現本書所列諸多被我（從人的角度）認爲平淡而瑣屑的重構步驟。我個人認爲，本書的最大價值，除了呼籲對軟體品質的追求態度，以及對重構「工程性」的認識，最終最重要的價值還在於：建立起吾人對於「目前和未來之自動化重構工具」的基本理論和實作技術上的認識與信賴。人類眼中平淡瑣屑的步驟，正是自動化重構工具的基礎。機器缺乏人類的「大局觀」智慧，機器需要的正是切割爲一個一個極小步驟的指令。一板一眼，一次一點點，這正是機器所需要的，也正是機器的專長。

本書第 14 章提到，Smalltalk 開發環境已含自動化重構工具。我並非 Smalltalk guy，我沒有用過這些工具。基於技術的飛快滾動（或我個人的孤陋寡聞），或許如今你已經可以在 Java, C++ 等物件導向編程環境中找到這一類自動化重構工具。

軟體技術圈內，重構（refactoring）常常被拿來與設計範式（design patterns）並論。書籍市場上，《*Refactoring*》也與《*Design Patterns*》齊名。GoF 曾經說『設計範式為重構提供了目標』，但本書作者 Martin 亦言『本書並沒有提供助你完成所有知名範式的重構手法，甚至連 GoF 的 23 個知名範式都沒有能夠全部涵蓋。』我們可以從這些話中理解技術的方向，以及書籍所反映的侷限。我並不完全贊同 Martin 所言『哪怕你手上有一個糟糕的設計或甚至一團混亂，你也可以藉由重構將它加工成設計良好的程式碼。』但我十分同意 Martin 說『你會發現所謂設計不再是一切動作的前提，而是在整個開發過程中逐漸浮現出來。』我比較擔心，閱歷不足的程式員在讀過本書後可能發酵出「先動手再說，死活可重構」的心態，輕忽了事前優秀設計的重要性。任何技術上的說法都必須有基本假設；雖然重構（或更向上說 XP，eXtreme Programming）的精神的確是「不妨先動手」，但若草率行事，代價還是很高的。重型開發和輕型開發各有所長，各有應用，世間並無萬應靈藥，任何東西都不能極端。過猶不及，皆不可取！

當然，「重構工程」與「自動化重構工具」可爲我們帶來相當大幅度的軟體品質提昇，這一點我毫無異議，並且非常期待☺。

關於本書製作

本書在翻譯與製作上保留了所有壞味道（bad smell）、重構（refactoring）、設計範式（design patterns）的英文名稱，並表現以特殊字型；只在封面內頁、目錄、小節標題中相應地給出一個根據字面或技術意義而做的中文譯名。各種「壞味道」名稱儘量就其意義選用負面字眼，如泥團、夸夸、過長、過大、過多、情結、偏執、驚悚、狎暱、純稚、冗員…。這些其實都是助憶之用，與茶餘飯後的談資（以及讀者批評的根據☺）。

原書各小節並無序號。爲求參考、檢索或討論時的方便，我爲譯本加上了序號。

本書保留相當份量的英文術語，時而英中並陳（英文爲主，中文爲輔）。這麼做的考量是，本書讀者不可能不知道 class, final, reference, public, package…這些簡短的、與 Java 編程息息相關的用詞。另一方面，我確實認爲，中文書內保留經過挑選的某些英文術語，有利於整體閱讀效果。

兩個需要特別說明的用詞是 Java 編程界慣用的 "field" 和 "method"。它們相當於 C++ 的 "data member" 和 "member function"。由於出現次數實在頻繁，爲降低中英夾雜程度，我把它們分別譯爲「欄位」和「函式」— 如果將 "method" 譯爲「方法」，恐怕術語突出性不高。此外，本書將「創造建立新物件」的 "create" 動作譯爲「創建」。「*static* 欄位與 *instance* 欄位」、「*reference* 物件與 *value* 物件」等等則保留部分英文，並選用如上的特殊字型。凡此種種，相信一進入書中您很快可以感受本書術語風格。

本書還有諸多地方採中英並陳（中文爲主，英文爲輔）方式，意在告訴讀者，我們（譯者）深知自己的不足與侷限，惟恐造成您對中譯名詞的誤解或不習慣，所以附上原文。

中文版（本書）已將英文版截至 2003/06/18 爲止之勘誤，修正於紙本。

一點點感想

Martin Fowler 表現於原書的寫作風格是：簡潔，愛用代名詞和略稱。這使得讀者往往需要在字面上揣度推敲。我期盼（並相信）經過技術意義的反芻、中英術語的並陳、中文表述的努力，中文版（本書）在閱讀時間、理解時間和記憶深度上，較之英文版，能夠爲以華文爲母語的讀者提高 10 倍以上的成效。

本書由我和熊節先生合譯。熊節負責第一個 pass，我負責後繼工作。中文版（本書）爲讀者帶來的閱讀和理解上的效益，熊節居於首功 — 雖說做的是第一個 pass，我從初稿品質便可看出他多次反覆推敲和文字琢磨的刻痕。至於整體風格、中英術語的選定、版面的呈現、乃至於全盤技術內涵的表現，如果有任何差錯，責任都是我的☺。

作爲一個資訊技術教育者，以及一個資訊技術傳播者，我在超過 10 年的著譯歷程中，觀察了不同級別的技術書品在讀書市場上的興衰起伏。這些適可反映大環境下技術從業人員及學子們的某些面向和取向。我很高興看到我們的中文技術書籍（著譯皆含）從早期盈盈滿滿的初階語言用書，逐漸進化到中高階語言用書、作業系統、技術內核、程式庫/框架、再至設計/分析、軟體工程。我很高興看到這樣的變化。我很高興看到《*Design Patterns*》、《*Refactoring*》、《*Agile*…》、《*UML*…》、《*XP*…》之類的書在中文書籍市場中現身，並期盼它們有豐富的讀者。

中文版（本書）支援網站有一個「術語 英中繁簡」對照表。如果您有需要，歡迎訪問，網址如下，並歡迎給我任何意見。謝謝。

侯捷 2003/06/18 于臺灣.新竹

jjhou@jjhou.com（電子郵箱）
http://www.jjhou.com（繁體）（術語對照表 http://www.jjhou.com/terms.htm）
http://jjhou.csdn.net（簡體）（術語對照表 http:// jjhou.csdn.net/terms.htm）

譯序

by 熊節

重構的生活方式

還記得那一天，當我把《*Refactoring*》的全部譯稿整理完畢，發送給侯老師時，心裡竟然不經意地有了一絲惘然。我是一隻習慣的動物，總是安於一種習慣的生活方式。在那之前的很長一段時間裡，習慣了每天晚上翻譯這本書，習慣了隨手把問題寫成 mail 發給 Martin Fowler 先生，習慣了閱讀 Martin 及時而耐心的回信，習慣了在那本複印的、略顯粗糙的書本上勾勾畫畫，習慣了躺在床上咀嚼回味那些帶有一點點英國紳士矜持口吻的詞句，習慣了背後嗡嗡作響的老空調…當深秋的風再次染紅了香山的葉，這種生活方式也就告一段落了。

只有幾位相熟的朋友知道我在翻譯這本書，他們不太明白爲什麼常把經濟學掛在嘴邊的我會樂於幫侯老師翻譯這本書 — 我自己也不明白，大概只能用愛好來解釋吧。既然已經衣食無憂，既然還有一點屬於自己的時間，能夠親手把這本《*Refactoring*》翻譯出來，也算是給自己的一個交代。

第一次聽到「重構」（refactoring）這個詞，是在 2001 年 10 月。在當時，它的思想足以令我感到震撼。軟體自有其美感所在。軟體工程希望建立完美的需求與設計，按照既有的規範編寫標準劃一的程式碼，這是結構的美；快速迭代和 RAD 顛覆「全知全能」的神話，用近乎刀劈斧砍（crack）的方式解決問題，在混沌的循環往復中實現需求，這是解構的美；而 Kent Beck 與 Martin Fowler 兩人站在一起，XP 那敏捷而又嚴謹的方法論演繹了重構的美 — 我不知道是誰最初把 refactoring 一詞翻譯爲「重構」，或許無心插柳，卻成了點睛之筆。

我一直是設計範式（design patterns）的愛好者。曾經在我的思想中，軟體開發應該有一個「理想國」—當然，在這個理想國維持著完美秩序的，不是哲學家，而

是 patterns。設計範式給我們的，不僅僅是一些問題的解決方案，更有追求完美「理型」的渴望。但是，Joshua Kerievsky 在那篇著名的《範式與 XP》（收錄於《極限編程研究》一書）中明白地指出：在設計前期使用 patterns 常常導致過度工程（over-engineering）。這是一個殘酷的現實，單憑對完美的追求無法寫出實用的程式碼，而「實用」是軟體壓倒一切的要素。從一篇《停止過度工程》開始，Joshua 撰寫了 "Refactoring to Patterns" 系列文章。這位猶太人用他民族性的睿智頭腦，敏銳地發現了軟體的後結構主義道路。而讓設計範式在飛速變化的 Internet 時代重新閃現光輝的，又是重構的力量。

在一篇流傳甚廣的帖子裡，有人把《*Refactoring*》與《*Design Patterns*》並列爲「Java 行業的聖經」。在我看來這種並列其實並不準確。實際上，儘管我如此喜愛這本《*Refactoring*》，但自從完成翻譯之後，我再也沒有讀過它。不，不是因爲我已經對它爛熟於心，而是因爲重構已經變成了我的另一種生活方式，變成了我每天的「麵包與黃油」，變成了我們整個團隊的空氣與水，以至於無須再到書中尋找任何「神諭」。而《*Design Patterns*》，我倒是放在手邊時常翻閱，因爲總是記得不那麼真切。

所以，在你開始閱讀本書之前，我有兩個建議要給你：首先，把你的敬畏扔到大西洋裡去，對於即將變得像空氣與水一樣普通的技術，你無須對它敬畏；其次，找到合適的開發工具（如果你和我一樣是 Java 人，那麼這個「合適的工具」就是 Eclipse），學會使用其中的自動測試和重構功能，然後再嘗試使用本書介紹的任何技術。懶惰是程式員的美德之一，絕不要因爲這本書讓你變得勤快。

最後，即使你完全掌握了這本書中的所有東西，也千萬不要跟別人吹噓。在我們的團隊裡，程式員常常會說：『如果沒有單元測試和重構，我沒辦法寫程式。』

好了，感謝你耗費一點點的時間來傾聽我現在對重構、對這本《*Refactoring*》的想法。Martin Fowler 經常說，花一點時間來重構是值得的，希望你會覺得花一點時間看我的文字也是值得的。

熊節 2003 年 6 月 11 日　夜 杭州

P.S. 我想借這個難得的機會感謝一個人：我親愛的女友馬姍姍。在北京的日子裡，是她陪伴著我度過每個日日夜夜，照顧我的生活，使我能夠有精力做些喜歡的事（包括翻譯這本書）。當我埋頭在螢幕前敲打鍵盤時，當我抱著書本冥思苦想時，她無私地容忍了我的痴迷與冷淡。謝謝你，姍姍，我永遠愛你。

目錄

Contents

序言

by Erich Gamma

重構（**refactoring**）這個概念來自 Smalltalk 圈子，沒多久就進入了其他語言陣營之中。由於重構是 framework（框架）開發中不可缺少的一部分，所以當 framework 開發人員討論自己的工作時，這個術語就誕生了。當他們精煉自己的 class hierarchies（類別階層體系）時，當他們叫喊自己可以拿掉多少多少行程式碼時，重構的概念慢慢浮出水面。framework 設計者知道，這東西不可能一開始就完全正確，它將隨著設計者的經驗成長而進化；他們也知道，程式碼被閱讀和被修改的次數遠遠多於它被編寫的次數。保持程式碼易讀、易修改的關鍵，就是重構 — 對 framework 而言如此，對一般軟體也如此。

好極了，還有什麼問題嗎？很顯然：重構具有風險。它必須修改運作中的程式，這可能引入一些幽微的錯誤。如果重構方式不恰當，可能毀掉你數天甚至數星期的成果。如果重構時不做好準備，不遵守規則，風險就更大。你挖掘自己的程式碼，很快發現了一些值得修改的地方，於是你挖得更深。挖得愈深，找到的重構機會就越多…於是你的修改也愈多。最後你給自己挖了個大坑，卻爬不出去了。爲了避免自掘墳墓，重構必須系統化進行。我在《*Design Patterns*》書中和另外三位（協同）作者曾經提過：design patterns（設計範式）爲 refactoring（重構）提供了目標。然而「確定目標」只是問題的一部分而已，改造程式以達目標，是另一個難題。

Martin Fowler 和本書另幾位作者清楚揭示了重構過程，他們爲物件導向軟體開發所做的貢獻，難以衡量。本書解釋重構的原理（principles）和最佳實踐方式（best practices），並指出何時何地你應該開始挖掘你的程式碼以求改善。本書的核心是一份完整的重構名錄（catalog of refactoring），其中每一項都介紹一種經過實證的程式碼變換手法（code transformation）的動機和技術。某些項目如 *Extract Method*

和 *Move Field* 看起來可能很淺顯，但不要掉以輕心，因爲理解這類技術正是有條不紊地進行重構的關鍵。本書所提的這些重構準則將幫助你一次一小步地修改你的程式碼，這就減少了過程中的風險。很快你就會把這些重構準則和其名稱加入自己的開發詞典中，並且朗朗上口。

我第一次體驗有紀律的、一次一小步的重構，是在 30000 英呎高空和 Kent Beck 共同編寫程式（譯註：原文爲 pair-programming，應該指的是 *eXtreme Programming* 中的所謂「成對/搭檔 編程」）。我們運用本書收錄的重構準則，保證每次只走一步。最後，我對這種實踐方式的效果感到十分驚訝。我不但對最後結果更有信心，而且開發壓力也小了很多。所以，我高度推薦你試試這些重構準則，你和你的程式都將因此更美好。

— *Erich Gamma*

Object Technology International, Inc.

前言

by Martin Fowler

從前，有位諮詢顧問參訪一個開發專案。系統核心是個 class hierarchy（類別階層體系），顧問看了開發人員所寫的一些程式碼。他發現整個體系相當凌亂，上層 classes 對自己的運作方式做了一些假設，這些假設被嵌入並被繼承下去。但是這些假設並不適合所有 subclasses，導致覆寫（overridden）行爲非常繁重。只要在 superclass 內做點修改，就可以減少許多覆寫必要。在另一些地方，superclass 的某些意圖並未被良好理解，因此其中某些行爲在 subclasses 內重複出現。還有一些地方，好幾個 subclasses 做相同的事情，其實可以把它們搬到 class hierarchy 的上層去做。

這位顧問於是建議專案經理看看這些程式碼，把它們整理一下，但是經理並不熱衷於此，畢竟程式看上去還可以執行，而且專案面臨很大的進度壓力。於是經理說，晚些時候再抽時間做這些整理工作。

顧問也把他的想法告訴了在這個 class hierarchy 上工作的程式員，告訴他們可能發生的事情。程式員都很敏銳，馬上就看出問題的嚴重性。他們知道這並不全是他們的錯，有時候的確需要借助外力才能發現問題。程式員立刻用了一兩天的時間整理好這個 class hierarchy，並刪掉了其中一半程式碼，功能毫髮無損。他們對此十分滿意，而且發現系統速度變得更快，更容易加入新 classes 或使用其他 classes。

專案經理並不高興。時程排得很緊，許多工作要做。系統必須在幾個月之後發佈，許多功能還等著加進去，這些程式員卻白白耗費兩天時間，什麼活兒都沒幹。原先的程式碼執行起來還算正常，他們的新設計顯然有點過於「理論」且過於「無瑕」。專案要出貨給客戶的，是可以有效執行的程式碼，不是用以取悅學究們的完美東西。顧問接下來又建議應該在系統的其他核心部分進行這樣的整理工作，

這會使整個專案停頓一至二個星期。所有這些工作只是爲了讓程式碼看起來更漂亮，並不能給系統添加任何新功能。

你對這個故事有什麼看法? 你認爲這個顧問的建議（更進一步整理程式）是對的嗎？你會因循那句古老的工程諺語嗎：「如果它還可以執行，就不要動它」。

我必須承認我自己有某些偏見，因爲我就是那個顧問。六個月之後這個專案宣告失敗，很大的原因是程式碼太複雜，無法除錯，也無法獲得可被接受的性能/效率。

後來，專案重新啓動，幾乎從頭開始編寫整個系統，Kent Beck 被請去做了顧問。他做了幾件迥異以往的事，其中最重要的一件就是堅持以持續不斷的重構行爲來整理程式碼。這個專案的成功，以及重構（refactoring）在這個成功專案中扮演的角色，鼓舞了我寫這本書的動機，如此一來我就能夠把 Kent 和其他一些人已經學會的「以重構方式改進軟體品質」的知識，傳播給所有讀者。

什麼是重構（Refactoring）？

所謂重構是這樣一個過程：「在不改變程式碼外在行爲的前提下，對程式碼做出修改，以改進程式的內部結構」。重構是一種有紀律的、經過訓練的、有條不紊的程式整理方法，可以將整理過程中不小心引入錯誤的機率降到最低。本質上說，重構就是「在程式碼寫好之後改進它的設計」。

「在程式碼寫好之後改進它的設計」？這種說法有點奇怪。按照目前對軟體開發的理解，我們相信應該先設計而後撰碼（coding）。首先得有一個良好的設計，然後才能開始撰碼。但是，隨著時間流逝，人們不斷修改程式碼，於是根據原先設計所得的系統，整體結構逐漸衰弱。程式碼品質慢慢沉淪，撰碼工作從嚴謹的工程墮落爲胡砍亂劈的隨性行爲。

「重構」正好與此相反。哪怕你手上有一個糟糕的設計，甚至是一堆混亂的程式碼，你也可以藉由重構將它加工成設計良好的程式碼。重構的每個步驟都很簡單，甚至簡單過了頭，你只需要把某個欄位（field）從一個 class 移到另一個 class，把某些程式碼從一個函式（method）拉出來構成另一個函式，或是在 class hierarchy 中把某些程式碼推上推下就行了。但是，聚沙成塔，這些小小的修改累積起來就可以根本改善設計品質。這和一般常見的「軟體會慢慢腐爛」的觀點恰恰相反。

透過重構（refactoring），你可以找出改變的平衡點。你會發現所謂設計不再是一切動作的前提，而是在整個開發過程中逐漸浮現出來。在系統構築過程中，你可以學習如何強化設計；其間帶來的互動可以讓一個程式在開發過程中持續保有良好的設計。

本書有些什麼?

本書是一本重構指南（guide to refactoring），爲專業程式員而寫。我的目的是告訴你如何以一種可控制且高效率的方式進行重構。你將學會這樣的重構方式：不引入臭蟲（錯誤），並且有條不紊地改進程式結構。

按照傳統，書籍應該以一個簡介開頭。儘管我也同意這個原則，但是我發現以概括性的討論或定義來介紹重構，實在不是件容易的事。所以我決定拿一個實例做爲開路先鋒。第 1 章展示一個小程式，其中有些常見的設計缺陷，我把它重構爲更合格的物件導向程式。其間我們可以看到重構的過程，以及數個很有用的重構準則。如果你想知道重構到底是怎麼回事，這一章不可不讀。

第 2 章涵蓋重構的一般性原則、定義，以及進行原因，我也大致介紹了重構所存在的一些問題。第 3 章由 Kent Beck 介紹如何嗅出程式碼中的「壞味道」，以及如何運用重構清除這些壞味道。「測試」在重構中扮演非常重要的角色，第 4 章介紹如何運用一個簡單的（源碼開放的）Java 測試框架，在程式碼中構築測試環境。

本書的核心部分，重構名錄（catalog of refactorings），從第 5 章延伸至第 12 章。這不是一份全面性的名錄，只是一個起步，其中包括迄今爲止我在工作中整理下來的所有重構準則。每當我想做點什麼 — 例如 *Replace Conditional with Polymorphism* — 的時候，這份名錄就會提醒我如何一步一步安全前進。我希望這是值得你日後一再回顧的部分。

本書介紹了其他人的許多研究成果，最後數章就是由他們之中的幾位所客串寫就。Bill Opdyke 在第 13 章記述他將重構技術應用於商業開發過程中遇到的一些問題。Don Roberts 和 John Brant 在第 14 章展望重構技術的未來 — 自動化工具。我把最後一章（第 15 章）留給重構技術的頂尖大師，Kent Beck。

在 Java 中運用重構

本書全部以 Java 撰寫實例。重構當然也可以在其他語言中實現，而且我也希望這本書能夠給其他語言使用者帶來幫助。但我覺得我最好在本書中只使用 Java，因爲那是我最熟悉的語言。我會不時寫下一些提示，告訴讀者如何在其他語言中進行重構，不過我真心希望看到其他人在本書基礎上針對其他語言寫出更多重構方面的書籍。

爲了最大程度地幫助讀者理解我的想法，我不想使用 Java 語言中特別複雜的部分。所以我避免使用 inner class（內隱類別）、reflection（反射機制）、thread（執行緒）以及很多強大的 Java 特性。這是因爲我希望儘可能清楚展現重構的核心。

我應該提醒你，這些重構準則並不針對並行（concurrent）或分散式（distributed）編程。那些主題會引出更多重要的事，超越了本書的關心範圍。

誰該閱讀本書?

本書瞄準專業程式員，也就是那些以編寫軟體爲生的人。書中的示例和討論，涉及大量需要詳細閱讀和理解的程式碼。這些例子都以 Java 完成。之所以選擇 Java，因爲它是一種應用範圍愈來愈廣的語言，而且任何具備 C 語言背景的人都可以輕易理解它。Java 是一種物件導向語言，而物件導向機制對於重構有很大幫助。

儘管關注對象是程式碼，重構（refactoring）對於系統設計也有巨大影響。資深設計師（senior designers）和架構規劃師（architects）也很有必要了解重構原理，並在自己的專案中運用重構技術。最好是由老資格、經驗豐富的開發人員來引入重構技術，因爲這樣的人最能夠良好理解重構背後的原理，並加以調整，使之適用於特定工作領域。如果你使用 Java 以外的語言，這一點尤其必要，因爲你必須把我給出的範例以其他語言改寫。

下面我要告訴你：如何能夠在不遍讀全書的情況下得到最多知識。

- 如果你想知道重構是什麼，請閱讀第 1 章，其中示例會讓你清楚重構過程。
- 如果你想知道爲什麼應該重構，請閱讀前兩章。它們告訴你「重構是什麼」以及「爲什麼應該重構」。

- 如果你想知道該在什麼地方重構，請閱讀第 3 章。它會告訴你一些程式碼特徵，這些特徵指出「這裡需要重構」。
- 如果你想真正（實際）進行重構，請完整閱讀前四章，然後選擇性地閱讀重構名錄（refactoring catalog）。一開始只需概略瀏覽名錄，看看其中有些什麼，不必理解所有細節。一旦真正需要實施某個準則，再詳細閱讀它，讓它來幫助你。名錄是一種具備查詢價值的章節，你也許並不想一次把它全部讀完。此外你還應該讀一讀名錄之後的「客串章節」，特別是第 15 章。

站在前人的肩膀上

就在本書一開始的此刻，我必須說：這本書讓我欠了一大筆人情債，欠那些在過去十年中做了大量研究工作並開創重構領域的人一大筆債。這本書原本應該由他們之中的某個人來寫，但最後卻是由我這個有時間有精力的人撿了便宜。

重構技術的兩位最早擁護者是 Ward Cunningham 和 Kent Beck。他們很早就把重構作爲開發過程的一個核心成份，並且在自己的開發過程中運用它。尤其需要說明的是，正因爲和 Kent 的合作，才讓我真正看到了重構的重要性，並直接激勵了我寫這一本書。

Ralph Johnson 在 University of Illinois, Urbana-Champaign（伊利諾大學烏爾班納分校）領導了一個小組，這個小組因其對物件技術（object technology）的實際貢獻而聞名。Ralph 很早就是重構技術的擁護者，他的一些學生也一直在研究這個課題。Bill Opdyke 的博士論文是重構研究領域的第一份詳細書面成果。John Brant 和 Don Roberts 則早已不滿足於寫文章了，他們寫了一個工具 — 重構瀏覽器（Refactoring Browser），對 Smalltalk 程式實施重構工程。

致謝

儘管有這些研究成果幫忙，我還需要很多協助才能寫出這本書。首先，並且也是最重要的，Kent Beck 給了我巨大的幫助。Kent 在底特律（Detroit）和我談起他正在爲 *Smalltalk Report* 撰寫一篇論文 [Beck, hanoi]，從此播下本書的第一顆種子。那篇論文不但讓我開始注意到重構技術，而且我還從中「偷」了許多想法放到本書第 1 章。Kent 也在其他地方幫助我，想出「程式碼味道」這個概念的是他，當我遇到各種困難時，鼓勵我的人也是他，常常和我一起工作助我完成這本書的，還

是他。我常常忍不住這麼想：他完全可以自己把這本書寫得更好。可惜有時間寫書的人是我，所以我也只能希望自己不要做得太差。

寫這本書的時候，我希望能把一些專家經驗直接與你分享，所以我非常感激那些花時間爲本書添加材料的人。Kent Beck, John Brant, William Opdyke 和 Don Roberts 編撰或合著了本書部分章節。此外 Rich Garzaniti 和 Ron Jeffries 幫我添加了一些有用的補充資料。

在任何像這樣的一本書裡，作者都會告訴你，技術審閱者提供了巨大的幫助。一如以往，Addison-Wesley 的 Carter 和他的優秀團隊是一群精明的審閱者。他們是：

- Ken Auer, Rolemodel Software, Inc.
- Joshua Bloch, Sun Microsystems, Java Software
- John Brant, University of Illinois at Urbana-Champaign
- Scott Corley, High Voltage Software, Inc.
- Ward Cunningham, Cunningham & Cunningham, Inc.
- Stéphane Ducasse
- Erich Gamma, Object Technology International, Inc.
- Ron Jeffries
- Ralph Johnson, University of Illinois
- Joshua Kerievsky, Industrial Logic, Inc.
- Doug Lea, SUNY Oswego
- Sander Tichelaar

他們大大提高了本書的可讀性和準確性，並且至少去掉了一些任何手稿都可能會有的潛在錯誤。在此我要特別感謝兩個效果顯著的建議，這兩個建議讓我的書看上去耳目一新：Ward 和 Ron 建議我以重構前後效果（包括程式碼和 UML 圖）並列的方式寫第 1 章，Joshua 建議我在重構名錄中畫出程式碼梗概（code sketches）。

除了正式審閱小組，還有很多非正式的審閱者。這些人或看過我的手稿，或關注我的網頁並留下對我很有幫助的意見。他們是 Leif Bennett, Michael Feathers, Michael Finney, Neil Galarneau, Hisham Ghazouli, Tony Gould, John Isner, Brian Marick, Ralf Reissing, John Salt, Mark Swanson, Dave Thomas 和 Don Wells。我相信肯定還有一些被我遺忘的人，請容我在此向你們道歉，並致上我的謝意。

有一個特別有趣的審閱小組，就是「惡名昭彰」☺ 的 University of Illinois at Urbana-Champaign 讀書小組。由於本書反映出他們的眾多研究成果，我要特別感謝他們的成就。這個小組成員包括 Fredrico "Fred" Balaguer, John Brant, Ian Chai, Brian Foote, Alejandra Garrido, Zhijiang "John" Han, Peter Hatch, Ralph Johnson, Songyu "Raymond" Lu, Dragos-Anton Manolescu, Hiroaki Nakamura, James Overturf, Don Roberts, Chieko Shirai, Les Tyrell 和 Joe Yoder。

任何好想法都需要在嚴酷的生產環境中接受檢驗。我看到重構對於 Chrysler Comprehensive Compensation（C3）系統起了巨大的影響。我要感謝那個團隊的所有成員：Ann Anderson, Ed Anderi, Ralph Beattie, Kent Beck, David Bryant, Bob Coe, Marie DeArment, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, Matt Saigeon, Don Thomas 和 Don Wells。和他們一起工作所獲得的第一手資料，鞏固了我對重構原理和利益的認識。他們在重構技術上不斷進步，極大程度地幫助我看到：一旦重構技術應用於歷時多年的大型專案中，可以起怎樣的作用。

再一次，我得到了 Addison-Wesley 的 J. Carter Shanklin 和其團隊的幫助，包括 Krysia Bebick, Susan Cestone, Chuck Dutton, Kristin Erickson, John Fuller, Christopher Guzikowski, Simone Payment 和 Genevieve Rajewski。與優秀出版商合作是一個令人愉快的經驗，他們會提供給作者大量的支援和幫助。

談到支援，爲一本書付出最多的，總是距離作者最近的人。對我來說，那就是我（現在）的妻子 Cindy。感謝妳，當我埋首工作的時候，妳還是一樣愛我。當我投入書中，總會不斷想起妳。

Martin Fowler
Melrose, Massachusetts
fowler@acm.org
http://www.martinfowler.com
http://www.refactoring.com

1

重構，第一個案例

Refactoring, a First Example

我該怎麼開始介紹重構（**refactoring**）呢？按照傳統作法，一開始介紹某個東西時，首先應該大致講講它的歷史、主要原理等等。可是每當有人在會場上介紹這些東西，總是誘發我的瞌睡蟲。我的思緒開始游蕩，我的眼神開始迷離，直到他或她拿出實例，我才能夠提起精神。實例之所以可以拯救我於太虛之中，因爲它讓我看見事情的真正行進。談原理，很容易流於泛泛，又很難說明如何實際應用。給出一個實例，卻可以幫助我把事情認識清楚。

所以我決定以一個實例作爲本書起點。在此過程中我將告訴你很多重構原理，並且讓你對重構過程有一點感覺。然後我才能向你提供一般慣見的原理介紹。

但是，面對這個介紹性實例，我遇到了一個大問題。如果我選擇一個大型程式，對程式本身的描述和對重構過程的描述就太複雜了，任何讀者都將無法掌握（我試了一下，哪怕稍微複雜一點的例子都會超過 100 頁）。如果我選擇一個夠小以至於容易理解的程式，又恐怕看不出重構的價值。

和任何想要介紹「應用於真實世界中的有用技術」的人一樣，我陷入了一個十分典型的兩難困境。我將帶引你看看如何在一個我所選擇的小程式中進行重構，然而坦白說，那個程式的規模根本不值得我們那麼做。但是如果我給你看的程式碼是大系統的一部分，重構技術很快就變得重要起來。所以請你一邊觀賞這個小例子，一邊想像它身處於一個大得多的系統。

1.1 起點

實例非常簡單。這是一個影片出租店用的程式，計算每一位顧客的消費金額並列印報表（statement）。操作者告訴程式：顧客租了哪些影片、租期多長，程式便根據租賃時間和影片類型算出費用。影片分爲三類：普通片、兒童片和新片。除了計算費用，還要爲常客計算點數；點數會隨著「租片種類是否爲新片」而有不同。

我以數個 classes 表現這個例子中的元素。圖 1.1 是一張 UML class diagram（類別圖），用以顯示這些 classes。我會逐一列出這些 classes 的程式碼。

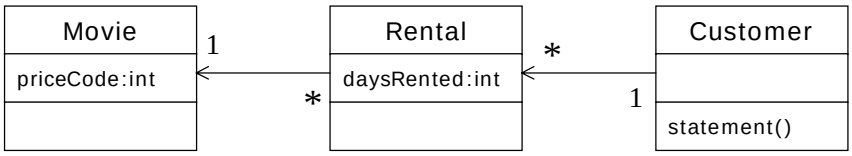


圖 1.1 本例一開始的各個 classes。此圖只顯示最重要的特性。圖中所用符號是 UML（Unified Modeling Language，統一建模語言，[Fowler, UML]）。

Movie（影片）

`Movie` 只是一個簡單的 data class（純資料類別）。

```
public class Movie {
  public static final int CHILDRENS = 2;
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;

  private String _title;   // 名稱
  private int _priceCode;  // 價格（代號）

  public Movie(String title, int priceCode){
     _title = title;
     _priceCode = priceCode;
  }
```

```
  public int getPriceCode(){
     return _priceCode;
  }

  public void setPriceCode(int arg){
     _priceCode = arg;
  }

  public String getTitle(){
     return _title;
  }
}
```

Rental（租賃）

`Rental` class 表示「某個顧客租了一部影片」。

```
class Rental {
  private Movie _movie;          // 影片
  private int _daysRented;       // 租期

  public Rental(Movie movie, int daysRented) {
    _movie = movie;
    _daysRented = daysRented;
  }

  public int getDaysRented() {
    return _daysRented;
  }

  public Movie getMovie() {
    return _movie;
  }
}
```

譯註：中文版（本書）支援網站提供本章重構過程中的各階段完整程式碼（共分七個階段），並含測試。網址見於封底。

Customer（顧客）

Customer class 用來表示顧客。就像其他 classes 一樣，它也擁有資料和相應的存取函式（accessor）：

```
class Customer {
  private String _name;                          // 姓名
  private Vector _rentals = new Vector();        // 租借記錄

  public Customer(String name) {
     _name = name;
  }

  public void addRental(Rental arg) {
     _rentals.addElement(arg);
  }

  public String getName() {
     return _name;
  }

  // 譯註：續下頁...
```

Customer 還提供了一個用以製造報表的函式（method），圖 1.2 顯示這個函式帶來的交互過程（interactions）。完整程式碼顯示於下一頁。

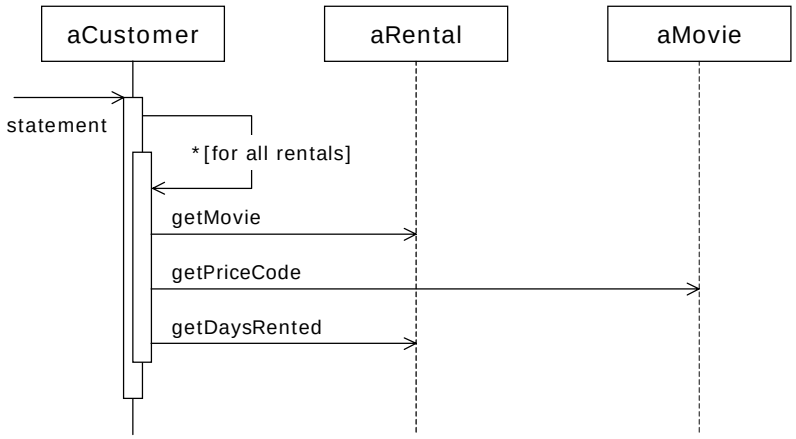


圖 1.2 statement() 的交互過程（interactions）

```
public String statement() {
  double totalAmount = 0;          // 總消費金額
  int frequentRenterPoints = 0;    // 常客積點
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";

  while(rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement(); // 取得一筆租借記錄

    //determine amounts for each line
    switch(each.getMovie().getPriceCode()) {  // 取得影片出租價格
      case Movie.REGULAR:           // 普通片
        thisAmount += 2;
        if(each.getDaysRented()>2)
          thisAmount += (each.getDaysRented()-2)*1.5;
        break;

      case Movie.NEW_RELEASE:       // 新片
        thisAmount += each.getDaysRented()*3;
        break;

      case Movie.CHILDRENS:         // 兒童片
        thisAmount += 1.5;
        if(each.getDaysRented()>3)
          thisAmount += (each.getDaysRented()-3)*1.5;
        break;
    }

    // add frequent renter points（累加 常客積點）
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE &&
       each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental（顯示此筆租借資料）
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  // add footer lines（結尾列印）
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}
```

對此起始程式的評價

這個起始程式給你留下什麼印象？我會說它設計得不好，而且很明顯不符合物件導向精神。對於這樣一個小程式，這些缺點其實沒有什麼關係。快速而隨性（quick and dirty）地設計一個簡單的程式並沒有錯。但如果這是複雜系統中具有代表性的一段，那麼我就真的要對這個程式信心動搖了。`Customer` 裡頭那個長長的 `statement()`做的事情實在太多了，它做了很多原本應該由其他 class 完成的事情。

即便如此，這個程式還是能正常工作。所以這只是美學意義上的判斷，只是對醜陋程式碼的厭惡，是嗎？在我們修改這個系統之前的確如此。編譯器才不會在乎程式碼好不好看呢。但是當我們打算修改系統的時候，就涉及到了人，而人在乎這些。差勁的系統是很難修改的，因爲很難找到修改點。如果很難找到修改點，程式員就很有可能犯錯，從而引入「臭蟲」（bugs）。

在這個例子裡，我們的用戶希望對系統做一點修改。首先他們希望以 HTML 格式列印報表，這樣就可以直接在網頁上顯示，這非常符合潮流。現在請你想一想，這個變化會帶來什麼影響。看看程式碼你就會發現，根本不可能在列印 HTML 報表的函式中復用（*reuse*）目前 `statement()`的任何行爲。你惟一可以做的就是編寫一個全新的 `htmlStatement()`，大量重複 `statement()`的行爲。當然，現在做這個還不太費力，你可以把 `statement()`複製一份然後按需要修改就是。

但如果計費標準發生變化，又會發生什麼事？你必須同時修改 `statement()`和 `htmlStatement()`，並確保兩處修改的一致性。當你後續還要再修改時，剪貼（copy-paste）問題就浮現出來了。如果你編寫的是一個永不需要修改的程式，那麼剪剪貼貼就還好，但如果程式要保存很長時間，而且可能需要修改，剪貼行爲就會造成潛在的威脅。

現在，第二個變化來了：用戶希望改變影片分類規則，但是還沒有決定怎麼改。他們設想了幾種方案，這些方案都會影響顧客消費和常客積點的計算方式。作爲一個經驗豐富的開發者，你可以肯定：不論用戶提出什麼方案，你惟一能夠獲得的保證就是他們一定會在六個月之內再次修改它。

爲了應付分類規則和計費規則的變化，程式必須對 `statement()`作出修改。但如果我們把 `statement()`內的程式碼拷貝到用以列印 HTML 報表的函式中，我們就必須確保將來的任何修改在兩個地方保持一致。隨著各種規則變得愈來愈複雜，適當的修改點愈來愈難找，不犯錯的機會也愈來愈少。

你的態度也許傾向於「儘量少修改程式」：不管怎麼說，它還執行得很好。你心裡頭牢牢記著那句古老的工程學格言：「如果它沒壞，就別動它」。這個程式也許還沒壞掉，但它帶來了傷害。它讓你的生活比較難過，因爲你發現很難完成客戶所需的修改。這時候就該重構技術粉墨登場了。

如果你發現自己需要爲程式添加一個特性，而程式碼結構使你無法很方便地那麼做，那就先重構那個程式，使特性的添加比較容易進行，然後再添加特性。

1.2 重構的第一步

每當我要進行重構的時候，第一個步驟永遠相同：我得爲即將修改的程式碼建立一組可靠的測試環境。這些測試是必要的，因爲儘管遵循重構準則可以使我避免絕大多數的臭蟲引入機會，但我畢竟是人，畢竟有可能犯錯。所以我需要可靠的測試。

由於 `statement()`的運作結果是個字串（string），所以我首先假設一些顧客，讓他們每個人各租幾部不同的影片，然後產生報表字串。然後我就可以拿新字串和手上已經檢查過的參考字串做比較。我把所有測試都設置好，俾得以在命令列輸入一條 Java 命令就把它們統統執行起來。執行這些測試只需數秒鐘，所以一如你即將見到，我經常執行它們。

測試過程中很重要的一部分，就是測試程式對於結果的回報方式。它們要不說"OK"，表示所有新字串都和參考字串一樣，要不就印出一份失敗清單，顯示問題字串的出現行號。這些測試都屬於自我檢驗（self-checking）。是的，你必須讓測試有能力自我檢驗，否則就得耗費大把時間來回比對，這會降低你的開發速度。

進行重構的時候，我們需要倚賴測試，讓它告訴我們是否引入了臭蟲。好的測試是重構的根本。花時間建立一個優良的測試機制是完全值得的，因爲當你修改程式時，好測試會給你必要的安全保障。測試機制在重構領域的地位實在太重要了，我將在第 4 章詳細討論它。

重構之前，首先檢查自己是否有一套可靠的測試機制。這些測試必須有自我檢驗（self-checking）能力。

1.3 分解並重組 `statement()`

第一個明顯引起我注意的就是長得離譜的 `statement()`。每當看到這樣長長的函式，我就想把它大卸八塊。要知道，程式碼區塊愈小，程式碼的功能就愈容易管理，程式碼的處理和搬移也都愈輕鬆。

本章重構過程的第一階段中，我將說明如何把長長的函式切開，並把較小塊的程式碼移至更合適的 class 內。我希望降低程式碼重複量，從而使新的（列印 HTML 報表用的）函式更容易撰寫。

第一個步驟是找出程式碼的邏輯泥團（*logical clump*）並運用 *Extract Method*（110）。本例一個明顯的邏輯泥團就是 `switch` 述句，把它提煉（*extract*）到獨立函式中似乎比較好。

和任何重構準則一樣，當我提煉一個函式時，我必須知道可能出什麼錯。如果我提煉得不好，就可能給程式引入臭蟲。所以重構之前我需要先想出安全作法。由於先前我已經進行過數次這類重構，所以我已經把安全步驟記錄於書後的重構名錄（refactoring catalog）中了。

首先我得在這段程式碼裡頭找出函式內的區域變數（local variables）和參數（parameters）。我找到了兩個：`each` 和 `thisAmount`，前者並未被修改，後者會被修改。任何不會被修改的變數都可以被我當成參數傳入新的函式，至於會被修改的變數就需格外小心。如果只有一個變數會被修改，我可以把它當作回返值。`thisAmount` 是個暫時變數，其值在每次迴圈起始處被設爲 0，並且在 `switch` 述句之前不會改變，所以我可以直接把新函式的回返值賦予它。

下面兩頁展示重構前後的程式碼。重構前的程式碼在左頁，重構後的程式碼在右頁。凡是從函式提煉出來的程式碼，以及新程式碼所做的任何修改，只要我覺得不是明顯到可以一眼看出，就以粗體字標示出來特別提醒你。**本章剩餘部分將延續這種左右比對形式**。

```
public String statement() {
  double totalAmount = 0;          // 總消費金額
  int frequentRenterPoints = 0;    // 常客積點
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";

  while(rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement(); // 取得一筆租借記錄

    //determine amounts for each line
    switch(each.getMovie().getPriceCode()) {  // 取得影片出租價格
      case Movie.REGULAR:           // 普通片
        thisAmount += 2;
        if(each.getDaysRented()>2)
            thisAmount += (each.getDaysRented()-2)*1.5;
        break;

      case Movie.NEW_RELEASE:       // 新片
        thisAmount += each.getDaysRented()*3;
        break;

      case Movie.CHILDRENS:         // 兒童片
        thisAmount += 1.5;
        if(each.getDaysRented()>3)
            thisAmount += (each.getDaysRented()-3)*1.5;
        break;
    }

    // add frequent renter points（累加 常客積點）
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental（顯示此筆租借資料）
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  // add footer lines（結尾列印）
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}
```

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = amountFor(each);    // 計算一筆租片費用

    // add frequent renter points
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }
  // add footer lines
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}

private int amountFor(Rental each) {     // 計算一筆租片費用
  int thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:          // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
         thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:      // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:        // 兒童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
         thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

每次做完這樣的修改之後，我都要編譯並測試。這一次起頭不算太好 — 測試失敗了，有兩筆測試數據告訴我發生錯誤。一陣迷惑之後我明白了自己犯的錯誤。我愚蠢地將 amountFor() 的回返值型別宣告爲 int，而不是 double。

```
private double amountFor(Rental each) {  // 計算一筆租片費用
  double thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:          // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:      // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:        // 兒童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

我經常犯這種愚蠢可笑的錯誤，而這種錯誤往往很難發現。在這裡，Java 無怨無尤地把 double 型別轉換爲 int 型別，而且還愉快地做了取整數動作 [Java Spec]。還好此處這個問題很容易發現，因爲我做的修改很小，而且我有很好的測試。藉著這個意外疏忽，我要闡述重構步驟的本質：由於每次修改的幅度都很小，所以任何錯誤都很容易發現。你不必耗費大把時間除錯，哪怕你和我一樣粗心。

重構技術係以微小的步伐修改程式。如果你犯下錯誤，很容易便可發現它。

由於我用的是 Java，所以我需要對程式碼做一些分析，決定如何處理區域變數。如果擁有相應的工具，這個工作就超級簡單了。Smalltalk 的確擁有這樣的工具：Refactoring Browser。運用這個工具，重構過程非常輕鬆，我只需標示出需要重構的程式碼，在選單中點選 *Extract Method*，輸入新的函式名稱，一切就自動搞定。而且工具絕不會像我那樣犯下愚蠢可笑的錯誤。我非常盼望早日出現 Java 版本的重構工具!

現在，我已經把原本的函式分爲兩塊，可以分別處理它們。我不喜歡 amountFor() 內的某些變數名稱，現在是修改它們的時候。

下面是原本的程式碼：

```
private double amountFor(Rental each) {  // 計算一筆租片費用
  double thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:          // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:      // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:        // 兒童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

下面是易名後的程式碼：

```
private double amountFor(Rental aRental) {    // 計算一筆租片費用
     double result= 0;
     switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:         // 普通片
           result += 2;
           if (aRental.getDaysRented() > 2)
               result += (aRental.getDaysRented() - 2) * 1.5;
           break;
        case Movie.NEW_RELEASE:     // 新片
           result += aRental.getDaysRented() * 3;
           break;
        case Movie.CHILDRENS:       // 兒童片
           result += 1.5;
           if (aRental.getDaysRented() > 3)
               result += (aRental.getDaysRented() - 3) * 1.5;
           break;
     }
     return result;
  }
```

易名之後我需要重新編譯並測試，確保沒有破壞任何東西。

更改變數名稱是值得的行爲嗎？絕對值得。好的程式碼應該清楚表達出自己的功能，變數名稱是程式碼清晰的關鍵。如果爲了提高程式碼的清晰度，需要修改某些東西的名字，大膽去做吧。只要有良好的搜尋/替換工具，更改名稱並不困難。語言所提供的強型檢驗（strong typing）以及你自己的測試機制會指出任何你遺漏的東西。記住：

任何一個傻瓜都能寫出計算機可以理解的程式碼。惟有寫出人類容易理解的程式碼，才是優秀的程式員。

程式碼應該表現自己的目的，這一點非常重要。閱讀程式碼的時候，我經常進行重構。這樣，隨著對程式的理解逐漸加深，我也就不斷地把這些理解嵌入程式碼中，這麼一來才不會遺忘我曾經理解的東西。

搬移「金額計算」程式碼

觀察 amountFor()時，我發現這個函式使用了來自 Rental class 的資訊，卻沒有使用來自 Customer class 的資訊。

```
class Customer...
  private double amountFor(Rental aRental) {
          double result = 0;
          switch (aRental.getMovie().getPriceCode()) {
             case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                   result += (aRental.getDaysRented() - 2) * 1.5;
                break;
             case Movie.NEW_RELEASE:
                result += aRental.getDaysRented() * 3;
                break;
             case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                   result += (aRental.getDaysRented() - 3) * 1.5;
                break;
          }
          return result;
       }
```

這立刻使我懷疑它是否被放錯了位置。絕大多數情況下，函式應該放在它所使用的資料的所屬 object（或說 class）內，所以 `amountFor()`應該移到 `Rental` class 去。爲了這麼做，我要運用 *Move Method*（142）。首先把程式碼拷貝到 `Rental` class 內，調整程式碼使之適應新家，然後重新編譯。像下面這樣：

```
class Rental...
       double getCharge() {
          double result = 0;
          switch (getMovie().getPriceCode()) {
             case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                   result += (getDaysRented() - 2) * 1.5;
                break;
             case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
             case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                   result += (getDaysRented() - 3) * 1.5;
                break;
          }
          return result;
       }
```

在這個例子裡，「適應新家」意味去掉參數。此外，我還要在搬移的同時變更函式名稱。

現在我可以測試新函式是否正常工作。只要改變 `Customer.amountFor()`函式內容，使它委託（*delegate*）新函式即可：

```
class Customer...
       private double amountFor(Rental aRental) {
          return aRental.getCharge();
       }
```

現在我可以編譯並測試，看看有沒有破壞了什麼東西。

下一個步驟是找出程式中對於舊函式的所有引用（*reference*）點，並修改它們，讓它們改用新函式。下面是原本的程式：

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = amountFor(each);

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                    each.getDaysRented() > 1)
                 frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;

    }
```

本例之中，這個步驟很簡單，因爲我才剛剛產生新函式，只有一個地方使用了它。一般情況下你得在可能運用該函式的所有 classes 中搜尋一遍。

```
class Customer
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = each.getCharge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                    each.getDaysRented() > 1)
                 frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;

    }
```

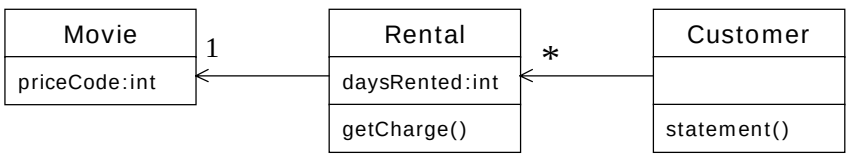


圖 1.3　搬移「金額計算」函式後，所有 classes 的狀態（*state*）

做完這些修改之後（圖 1.3），下一件事就是去掉舊函式。編譯器會告訴我是否我漏掉了什麼。然後我進行測試，看看有沒有破壞什麼東西。

有時候我會保留舊函式，讓它呼叫新函式。如果舊函式是一個 public 函式，而我又不想修改其他 class 的介面，這便是一種有用的手法。

當然我還想對 `Rental.getCharge()`做些修改，不過暫時到此爲止，讓我們回到 `Customer.statement()`：

```
    public String statement() {
       double totalAmount = 0;
       int frequentRenterPoints = 0;
       Enumeration rentals = _rentals.elements();
       String result = "Rental Record for " + getName() + "\n";
       while (rentals.hasMoreElements()) {
          double thisAmount = 0;
          Rental each = (Rental) rentals.nextElement();

          thisAmount = each.getCharge();

          // add frequent renter points
          frequentRenterPoints ++;
          // add bonus for a two day new release rental
          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                  each.getDaysRented() > 1)
               frequentRenterPoints ++;

          //show figures for this rental
          result += "\t" + each.getMovie().getTitle()+ "\t" +
                    String.valueOf(thisAmount) + "\n";
          totalAmount += thisAmount;

       }
       //add footer lines
       result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
       result += "You earned " + String.valueOf(frequentRenterPoints) +
                 " frequent renter points";
       return result;

    }
```

下一件引我注意的事是：thisAmount 如今變成多餘了。它接受 each.getCharge() 的執行結果，然後就不再有任何改變。所以我可以運用 *Replace Temp with Query*（120）把 thisAmount 除去。

```
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
               each.getDaysRented() > 1) frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints)
                + " frequent renter points";
        return result;

    }
}
```

做完這份修改，我立刻編譯並測試，保證自己沒有破壞任何東西。

我喜歡儘量除去這一類暫時變數。暫時變數往往形成問題，它們會導致大量參數被傳來傳去，而其實完全沒有這種必要。你很容易失去它們的蹤跡，尤其在長長的函式之中更是如此。當然我這麼做也需付出效率上的代價，例如本例的費用就被計算了兩次。但是這很容易在 Rental class 中被最佳化。而且如果程式碼有合理的組織和管理，最佳化會有很好的效果。我將在 p.69 的「重構與效率/性能」一節詳談這個問題。

提煉「常客積點計算」程式碼

下一步要對「常客積點計算」做類似處理。點數的計算視影片種類而有不同，不過不像收費規則有那麼多變化。看來似乎有理由把積點計算責任放在 Rental class 身上。首先我們需要針對「常客積點計算」這部分程式碼（以下粗體部分）運用 *Extract Method*（110）重構準則：

```
    public String statement() {

        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
              && each.getDaysRented() > 1) frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints)
            + " frequent renter points";
        return result;

    }
}
```

再一次我又要尋找區域變數。這裡再一次用到了 each，而它可以被當做參數傳入新函式中。另一個暫時變數是 frequentRenterPoints。本例中的它在被使用之前已經先有初值，但提煉出來的函式並沒有讀取該值，所以我們不需要將它當作參數傳進去，只需對它執行「附添賦值動作」（*appending assignment*, operator+=）就行了。

我完成了函式的提煉，重新編譯並測試；然後做一次搬移，再編譯、再測試。重構時最好小步前進，如此一來犯錯的機率最小。

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }

class Rental...
        int getFrequentRenterPoints() {
            if ((getMovie().getPriceCode() == Movie.NEW_RELEASE)
                && getDaysRented() > 1)
                return 2;
            else
                return 1;
        }
```

我利用重構前後的 UML（Unified Modeling Language，統一建模語言）圖形（圖 1.4 至圖 1.7）總結剛才所做的修改。和先前一樣，左頁是修改前的圖，右頁是修改後的圖。

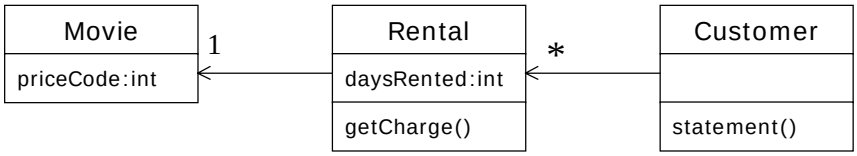


圖 1.4 「常客積點計算」函式被提煉及搬移之前的 class diagrams

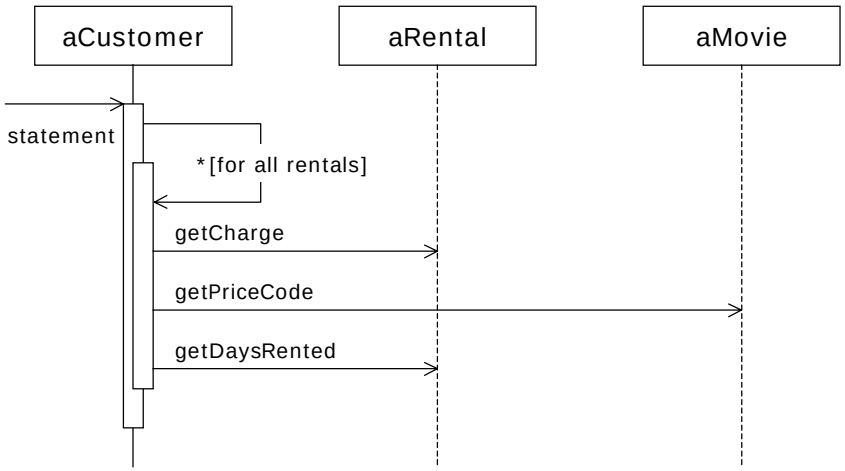


圖 1.5 「常客積點計算」函式被提煉及搬移之前的 sequence diagrams

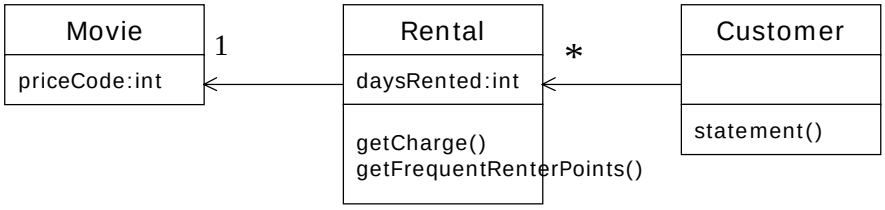


圖 1.6 「常客積點計算」函式被提煉及搬移之後的 class diagrams

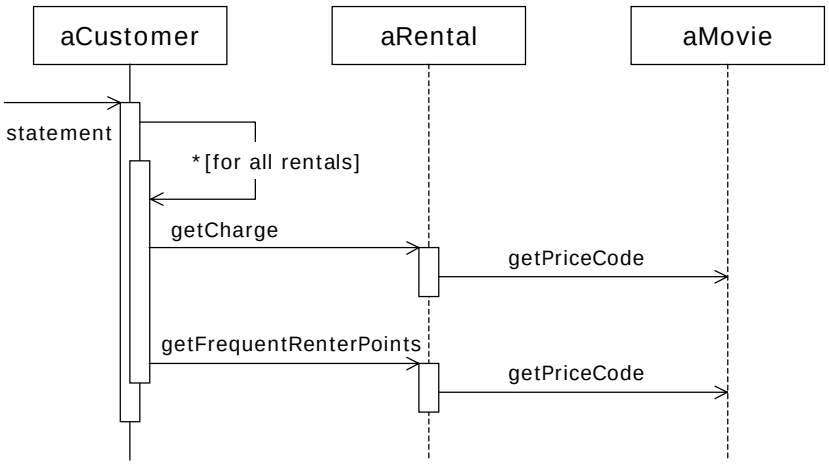


圖 1.7 「常客積點計算」函式被提煉及搬移之後的 sequence diagrams

去除暫時變數

正如我在前面提過的，暫時變數可能是個問題。它們只在自己所屬的函式中有效，所以它們會助長「冗長而複雜」的函式。這裡我們有兩個暫時變數，兩者都是用來從 Customer 物件相關的 Rental 物件中獲得某個總量。不論 ASCII 版或 HTML 版都需要這些總量。我打算運用 *Replace Temp with Query*（120），並利用所謂的 ***query method*** 來取代 totalAmount 和 frequentRentalPoints 這兩個暫時變數。由於 class 內的任何函式都可以取用（呼叫）上述所謂 ***query methods***，所以它能夠促進較乾淨的設計，而非冗長複雜的函式：

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }
```

首先我以 Customer class 的 getTotalCharge()取代 totalAmount：

```
class Customer...

    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        //add footer lines
        result += "Amount owed is " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
                  String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }

    // 譯註：此即所謂 query method
    private double getTotalCharge() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getCharge();
        }
        return result;
    }
```

這並不是 *Replace Temp with Query*（120）的最簡單情況。由於 totalAmount 在迴圈內部被賦值，我不得不把迴圈複製到 ***query method*** 中。

重構之後，重新編譯並測試，然後以同樣手法處理 frequentRenterPoints：

```
class Customer...
    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        //add footer lines
        result += "Amount owed is " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }
```

```
public String statement() {
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + getName() + "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      //show figures for this rental
      result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.getCharge()) + "\n";
   }

   //add footer lines
   result += "Amount owed is " +
             String.valueOf(getTotalCharge()) + "\n";
   result += "You earned " +
             String.valueOf(getTotalFrequentRenterPoints()) +
             " frequent renter points";
   return result;
}

// 譯註：此即所謂 query method
private int
   int result = 0;
   Enumeration rentals = _rentals.elements();
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getFrequentRenterPoints();
   }
   return result;
}
```

圖 1.8 至圖 1.11 分別以 UML class diagram（類別圖）和 interaction diagram（交互作用圖）展示 `statement()`重構前後的變化。

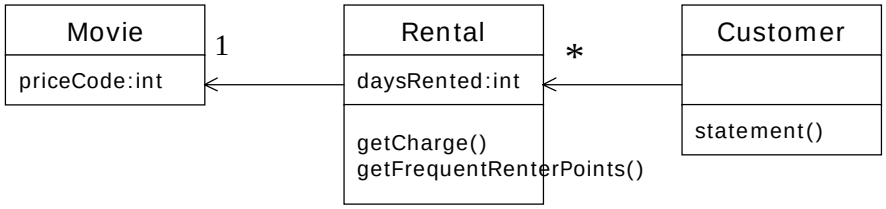


圖 1.8 「總量計算」函式被提煉前的 class diagram

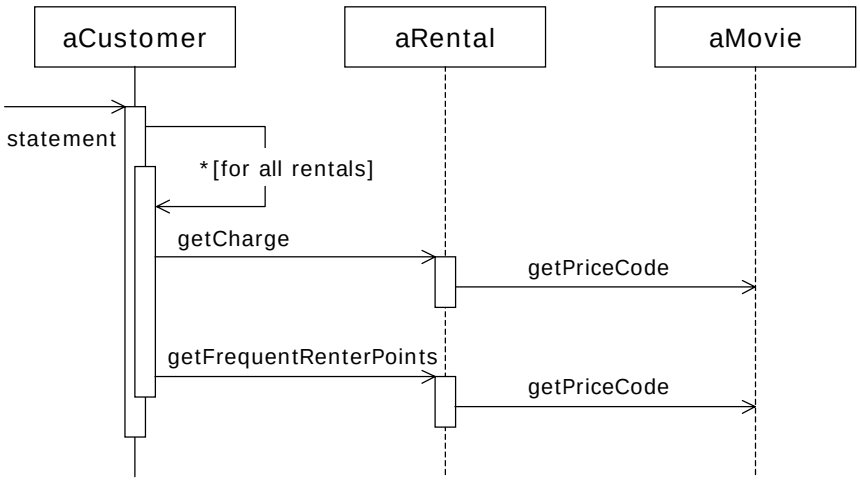


圖 1.9 「總量計算」函式被提煉前的 sequence diagram

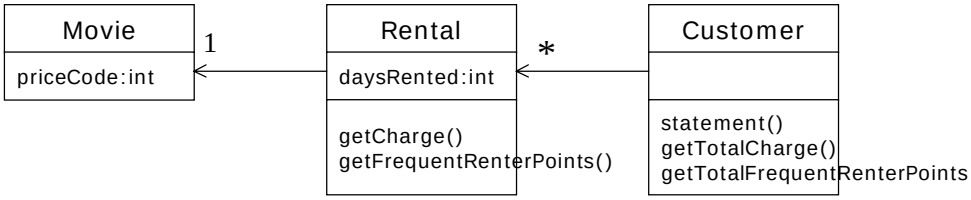


圖 1.10 「總量計算」函式被提煉後的 class diagram

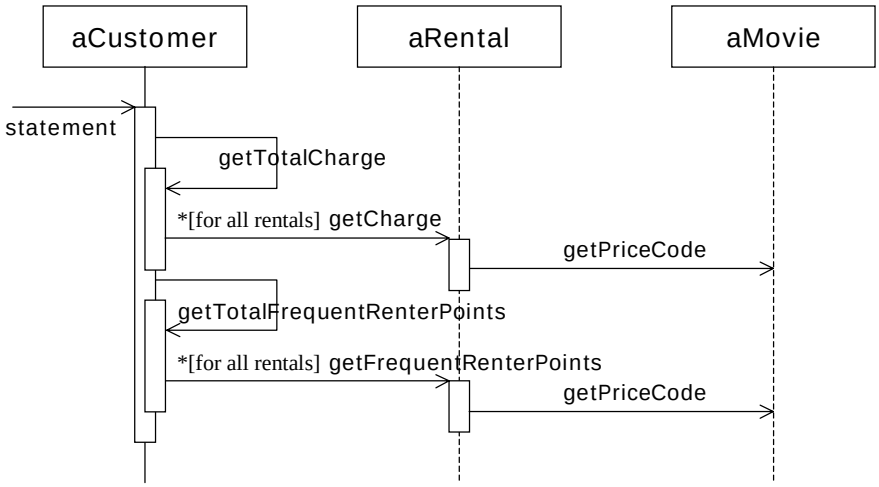


圖 1.11 「總量計算」函式被提煉後的 sequence diagram

做完這次重構，有必要停下來思考一下。大多數重構都會減少程式碼總量，但這次卻增加了程式碼總量，那是因爲 Java 1.1 需要大量述句（statements）來設置一個總和（*summing*）迴圈。哪怕只是一個簡單的總和迴圈，每個元素只需一行程式碼，週邊的支援程式碼也需要六行之多。這其實是任何程式員都熟悉的習慣寫法，但程式碼數量還是太多了。

這次重構存在另一個問題，那就是效率。原本程式碼只執行 `while` 迴圈一次，新版本要執行三次。**如果** `while` 迴圈耗時很多，就**可能**大大降低程式的效率。單單爲了這個原因，許多程式員就不願進行這個重構動作。但是請注意我的用詞：**如果**和**可能**。除非我進行評測（*profile*），否則我無法確定迴圈的執行時間，也無法知道這個迴圈是否被經常使用以至於影響系統的整體效率。重構時你不必擔心這些，最佳化時你才需要擔心它們，但那時候你已處於一個比較有利的位置，有更多選擇可以完成有效最佳化（見 p.69 的討論）。

現在，`Customer` class 內的任何程式碼都可以取用這些 ***query methods*** 了。如果系統他處需要這些資訊，也可以輕鬆地將 ***query methods*** 加入 `Customer` class 介面。如果沒有這些 ***query methods***，其他函式就必須了解 `Rental` class，並自行建立迴圈。在一個複雜系統中，這將使程式的編寫難度和維護難度大大增加。

你可以很明顯看出來，`htmlStatement()`和 `statement()`是不同的。現在，我應該脫下「重構」的帽子，戴上「添加功能」的帽子。我可以像下面這樣編寫 `htmlStatement()`，並添加相應測試：

```
public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rentals for <EM>" + getName() +
                        "</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
               Rental each = (Rental) rentals.nextElement();
               //show figures for each rental
               result += each.getMovie().getTitle()+ ": " +
                 String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result +=  "<P>You owe <EM>" +
                   String.valueOf(getTotalCharge()) +
                   "</EM><P>\n";
        result += "On this rental you earned <EM>" +
               String.valueOf(getTotalFrequentRenterPoints()) +
               "</EM> frequent renter points<P>";
        return result;
}
```

透過計算邏輯的提煉，我可以完成一個 `htmlStatement()`，並復用（*reuse*）原本`statement()`內的所有計算。我不必剪剪貼貼，所以如果計算規則發生改變，我只需在程式中做一處修改。完成其他任何類型的報表也都很快而且很容易。這次重構並不花很多時間，其中大半時間我用來弄清楚程式碼所做的事，而這是我無論如何都得做的。

前述有些重構碼係從 ASCII 版本裡頭拷貝過來 — 主要是迴圈設置部分。更深入的重構動作可以清除這些重複程式碼。我可以把處理表頭（header）、表尾（footer）和報表細目的程式碼都分別提煉出來。在 *Form Template Method*（345）實例中，你可以看到如何做這些動作。但是，現在用戶又開始嘀咕了，他們準備修改影片分類規則。我們尚未清楚他們想怎麼做，但似乎新分類法很快就要引入，現有的分類法馬上就要變更。與之相應的費用計算方式和常客積點計算方式都還待決定，現在就對程式做修改，肯定是愚蠢的。我必須進入費用計算和常客積點計算中，把「因條件而異的程式碼」（譯註：指的是 `switch` 述句內的 `case` 子句）替換掉，這樣才能爲將來的改變鍍上一層保護膜。現在，請重新戴回「重構」這頂帽子。

1.4 運用多型（polymorphism）取代與價格相關的條件邏輯

這個問題的第一部分是 `switch` 述句。在另一個物件的屬性（*attribute*）基礎上運用 `switch` 述句，並不是什麼好主意。如果不得不使用，也應該在物件自己的資料上使用，而不是在別人的資料上使用。

```
class Rental...
  double getCharge() {
     double result = 0;
     switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
           result += 2;
           if (getDaysRented() > 2)
              result += (getDaysRented() - 2) * 1.5;
           break;
        case Movie.NEW_RELEASE:
           result += getDaysRented() * 3;
           break;
        case Movie.CHILDRENS:
           result += 1.5;
           if (getDaysRented() > 3)
              result += (getDaysRented() - 3) * 1.5;
           break;
     }
     return result;
  }
```

這暗示 `getCharge()`應該移到 `Movie` class 裡頭去：

```
class Movie...
  double getCharge(int daysRented) {
     double result = 0;
     switch (getPriceCode()) {
        case Movie.REGULAR:
           result += 2;
           if (daysRented > 2)
              result += (daysRented - 2) * 1.5;
           break;
        case Movie.NEW_RELEASE:
           result += daysRented * 3;
           break;
        case Movie.CHILDRENS:
           result += 1.5;
           if (daysRented > 3)
              result += (daysRented - 3) * 1.5;
           break;
     }
     return result;
  }
```

爲了讓它得以運作，我必須把「租期長度」作爲參數傳遞進去。當然，「租期長度」來自 `Rental` 物件。計算費用時需要兩份資料：「租期長度」和「影片類型」。爲什麼我選擇「將租期長度傳給 `Movie` 物件」而不是「將影片類型傳給 `Rental`物件」呢？因爲本系統可能發生的變化是加入新影片類型，這種變化帶有不穩定傾向。如果影片類型有所變化，我希望掀起最小的漣漪，所以我選擇在 `Movie` 物件內計算費用。

我把上述計費方法放進 `Movie` class 裡頭，然後修改 `Rental` 的 `getCharge()`，讓它使用這個新函式（圖 1.12 和圖 1.13）：

```
class Rental...
  double getCharge() {
        return _movie.getCharge(_daysRented);
  }
```

搬移 `getCharge()`之後，我以相同手法處理常客積點計算。這樣我就把根據影片類型而變化的所有東西，都放到了影片類型所屬的 class 中。以下是重構前的程式碼：

```
class Rental...
 int getFrequentRenterPoints() {
     if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          getDaysRented() > 1)
         return 2;
     else
         return 1;
 }
```

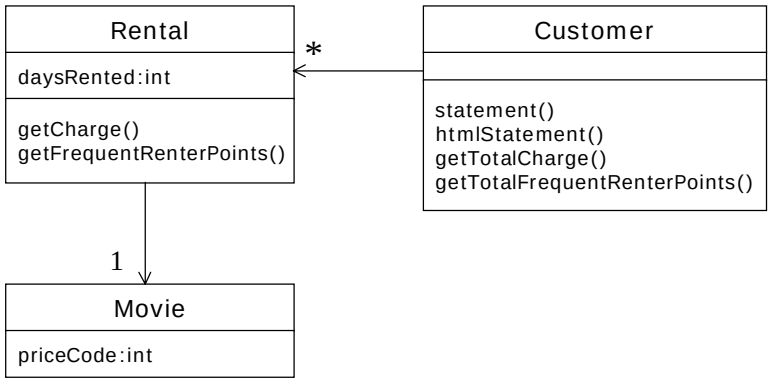


圖 1.12　本節所討論的兩個函式被移到 `Movie` class 內之前，系統的 class diagram。

重構如下：

```
class Rental...
  int getFrequentRenterPoints() {
      return _movie.getFrequentRenterPoints(_daysRented);
  }

class Movie...
  int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
  }
```

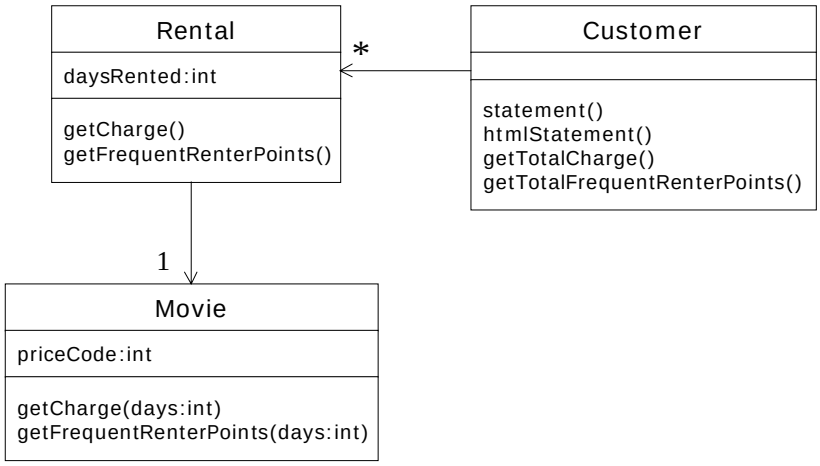


圖 1.13　本節所討論的兩個函式被移到 `Movie` class 內之後，系統的 class diagram。

終於…我們來到繼承（inheritance）

我們有數種影片類型，它們以不同的方式回答相同的問題。這聽起來很像 subclasses 的工作。我們可以建立 `Movie` 的三個 subclasses，每個都有自己的計費法（圖 1.14）。

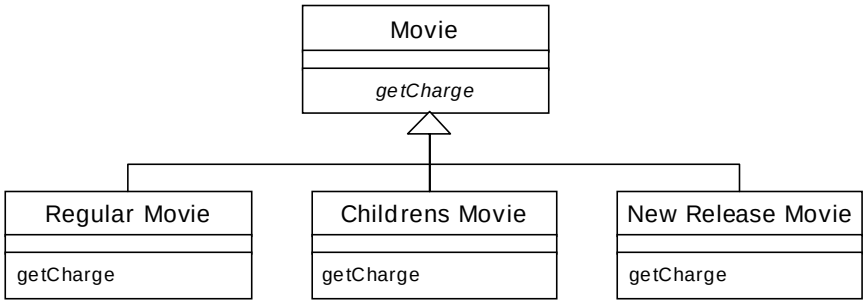


圖 1.14　以繼承機制表現不同的影片類型

這麼一來我就可以運用多型（polymorphism）來取代 `switch` 述句了。很遺憾的是這裡有個小問題，不能這麼幹。一部影片可以在生命週期內修改自己的分類，一個物件卻不能在生命週期內修改自己所屬的 class。不過還是有一個解決方法：***State*** pattern（範式）[Gang of Four]。運用它之後，我們的 classes 看起來像圖 1.15。

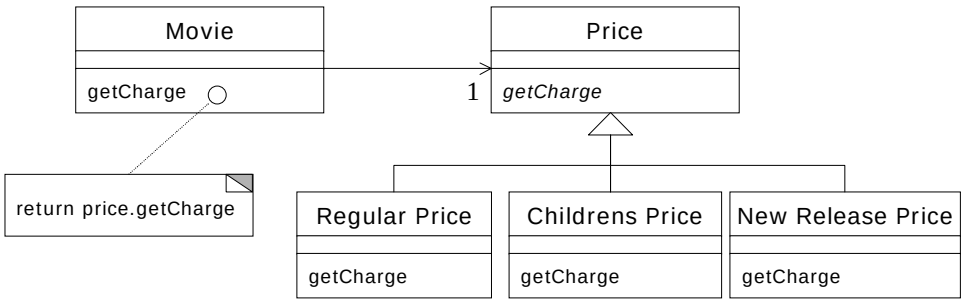


圖 1.15　運用 ***State*** pattern（範式）表現不同的影片

加入這一層間接性，我們就可以在 `Price` 物件內進行 subclassing 動作（譯註：一如圖 1.15），於是便可在任何必要時刻修改價格。

如果你很熟悉 Gang of Four 所列的各種範式（patterns），你可能會問：『這是一個 ***State*** 還是一個 ***Strategy***？』答案取決於 `Price` class 究竟代表計費方式（此時我喜歡把它叫做 `Pricer` 或 `PricingStrategy`），或是代表影片的某個狀態（*state*。例如「*Star Trek X* 是一部新片」）。在這個階段，對於範式（和其名稱）的選擇反映出你對結構的想法。此刻我把它視爲影片的某種狀態（*state*）。如果未來我覺得 ***Strategy*** 能更好地說明我的意圖，我會再重構它，修改名字，以形成 ***Strategy***。

爲了引入 ***State*** 範式，我使用三個重構準則。首先運用 *Replace Type Code with State/Strategy*（227），將「與型別相依的行爲」（type code behavior）搬移至 ***State*** 範式內。然後運用 *Move Method*（142）將 `switch` 述句移到 `Price` class 裡頭。最後運用 *Replace Conditional with Polymorphism*（255）去掉 `switch` 述句。

首先我要使用 *Replace Type Code with State/Strategy*（227）。第一步驟是針對「與型別相依的行爲」使用 *Self Encapsulate Field*（171），確保任何時候都透過 **getting** 和 **setting** 兩個函式來運用這些行爲。由於多數程式碼來自其他 classes，所以多數函式都已經使用 **getting** 函式。但建構式（constructor）仍然直接存取價格代號（譯註：程式中的 `_priceCode`）：

```
class Movie...
     public Movie(String name, int priceCode) {
          _title = name;
          _priceCode = priceCode;
     }
```

我可以用一個 **setting** 函式來代替：

```
class Movie
     public Movie(String name, int priceCode) {
        _title = name;
        setPriceCode(priceCode);    // 譯註：這就是一個 set method
     }
```

然後編譯並測試，確保沒有破壞任何東西。現在我加入新 class，並在 `Price` 物件中提供「與型別相依的行爲」。爲了實現這一點，我在 `Price` 內加入一個抽象函式（abstract method），並在其所有 subclasses 中加上對應的具體函式（concrete method）：

```
abstract class Price {
       abstract int getPriceCode();       // 取得價格代號
}
class ChildrensPrice extends Price {
       int getPriceCode() {
              return Movie.CHILDRENS;
       }
}
class NewReleasePrice extends Price {
       int getPriceCode() {
              return Movie.NEW_RELEASE;
       }
}
class RegularPrice extends Price {
       int getPriceCode() {
              return Movie.REGULAR;
       }
}
```

現在我可以編譯這些新 classes 了。

現在，我需要修改 Movie class 內的「價格代號」存取函式（*get/set* 函式，如下），讓它們使用新 class。下面是重構前的樣子：

```
public int getPriceCode() {
        return _priceCode;
}
public setPriceCode (int arg) {
        _priceCode = arg;
}
private int _priceCode;
```

這意味我必須在 `Movie` class 內保存一個 `Price` 物件，而不再是保存一個 `_priceCode` 變數。此外我還需要修改存取函式（譯註：即 **get/set** 函式）：

```
class Movie...
  public int getPriceCode() {            // 取得價格代號
     return _price.getPriceCode();
  }
  public void setPriceCode(int arg) {    // 設定價格代號
      switch (arg) {
          case REGULAR:         // 普通片
               _price = new RegularPrice();
               break;
          case CHILDRENS:       // 兒童片
               _price = new ChildrensPrice();
               break;
          case NEW_RELEASE:     // 新片
               _price = new NewReleasePrice();
               break;
          default:
             throw new IllegalArgumentException("Incorrect Price Code");
       }
  }
  private Price _price;
```

現在我可以重新編譯並測試，那些比較複雜的函式根本不知道世界已經變了個樣兒。

現在我要對 getCharge()實施 *Move Method*（142）。下面是重構前的程式碼：

```
class Movie...
     double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
           case Movie.REGULAR:
              result += 2;
              if (daysRented > 2)
                 result += (daysRented - 2) * 1.5;
              break;
           case Movie.NEW_RELEASE:
              result += daysRented * 3;
              break;
           case Movie.CHILDRENS:
              result += 1.5;
              if (daysRented > 3)
                 result += (daysRented - 3) * 1.5;
              break;
        }
        return result;
     }
```

搬移動作很簡單。下面是重構後的程式碼：

```
class Movie...
     double getCharge(int daysRented) {
            return _price.getCharge(daysRented);
     }

class Price...
     double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
     return result;
}
```

搬移之後，我就可以開始運用 *Replace Conditional with Polymorphism*（255）了。下面是重構前的程式碼：

```
class Price...
     double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
           case Movie.REGULAR:
              result += 2;
              if (daysRented > 2)
                 result += (daysRented - 2) * 1.5;
              break;
           case Movie.NEW_RELEASE:
              result += daysRented * 3;
              break;
           case Movie.CHILDRENS:
              result += 1.5;
              if (daysRented > 3)
                 result += (daysRented - 3) * 1.5;
              break;
        }
        return result;
     }
```

我的作法是一次取出一個 case 分支，在相應的 class 內建立一個覆寫函式（overriding method）。先從 RegularPrice 開始：

```
class RegularPrice...
    double getCharge(int daysRented) {
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    }
```

這個函式覆寫（*overrides*）了父類別中的 case 述句，而我暫時還把後者留在原處不動。現在編譯並測試，然後取出下一個 case 分支，再編譯並測試。（爲了保證被執行的的確是 subclass 程式碼，我喜歡故意丟一個錯誤進去，然後讓它執行，讓測試失敗。噢，我是不是有點太偏執了？）

```
class ChildrensPrice...
    double getCharge(int daysRented) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }

class NewReleasePrice...
    double getCharge(int daysRented) {
        return daysRented * 3;
    }
```

處理完所有 case 分支之後，我就把 Price.getCharge()宣告爲 abstract：

```
class Price...
    abstract double getCharge(int daysRented);
```

現在我可以運用同樣手法處理 `getFrequentRenterPoints()`。重構前的樣子如下（譯註：其中有「與型別相依的行爲」，也就是「判斷是否爲新片」那個動作）：

```
class Movie...
     int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
             return 2;
        else
             return 1;
     }
```

首先我把這個函式移到 `Price` class 裡頭：

```
Class Movie...
     int getFrequentRenterPoints(int daysRented) {
       return _price.getFrequentRenterPoints(daysRented);
     }
Class Price...
     int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
             return 2;
        else
             return 1;
     }
```

但是這一次我不把 superclass 函式宣告爲 abstract。我只是爲「新片類型」產生一個覆寫函式（*overriding method*），並在 superclass 內留下一個已定義的函式，使它成爲一種預設行爲。

```
// 譯註：在新片中產生一個覆寫函式（overriding method）
Class NewReleasePrice
     int getFrequentRenterPoints(int daysRented) {
       return (daysRented > 1) ? 2: 1;
     }

// 譯註：在 superclass 內保留它，使它成爲一種預設行爲
Class Price...
    int getFrequentRenterPoints(int daysRented){
       return 1;
    }
```

引入 ***State*** 範式花了我不少力氣，值得嗎？這麼做的收穫是：如果我要修改任何與價格有關的行爲，或是添加新的定價標準，或是加入其他取決於價格的行爲，程式的修改會容易得多。這個程式的其餘部分並不知道我運用了 ***State*** 範式。對於我目前擁有的這麼幾個小量行爲來說，任何功能或特性上的修改也許都稱不上什麼困難，但如果在一個更複雜的系統中，有十多個與價格相關的函式，程式的修改難易度就會有很大的區別。以上所有修改都是小步驟進行，進度似乎太過緩慢，但是沒有任何一次我需要打開除錯器（debugger），所以整個過程實際上很快就過去了。我書寫本章所用的時間，遠比修改那些程式碼的時間多太多了。

現在我已經完成了第二個重要的重構行爲。從此，修改「影片分類結構」，或是改變「費用計算規則」、改變常客積點計算規則，都容易多了。圖 1.16 和圖 1.17 描述 ***State*** 範式對於價格資訊所起的作用。

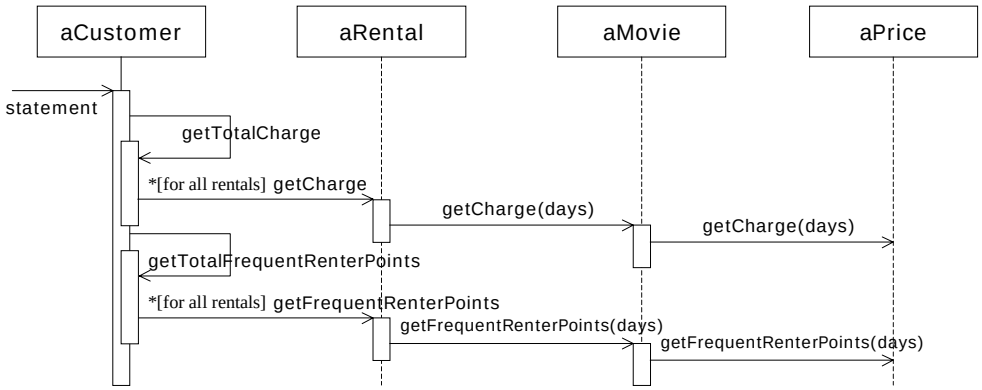


圖 1.16　運用 ***State*** pattern（範式）當時的 Interaction diagram

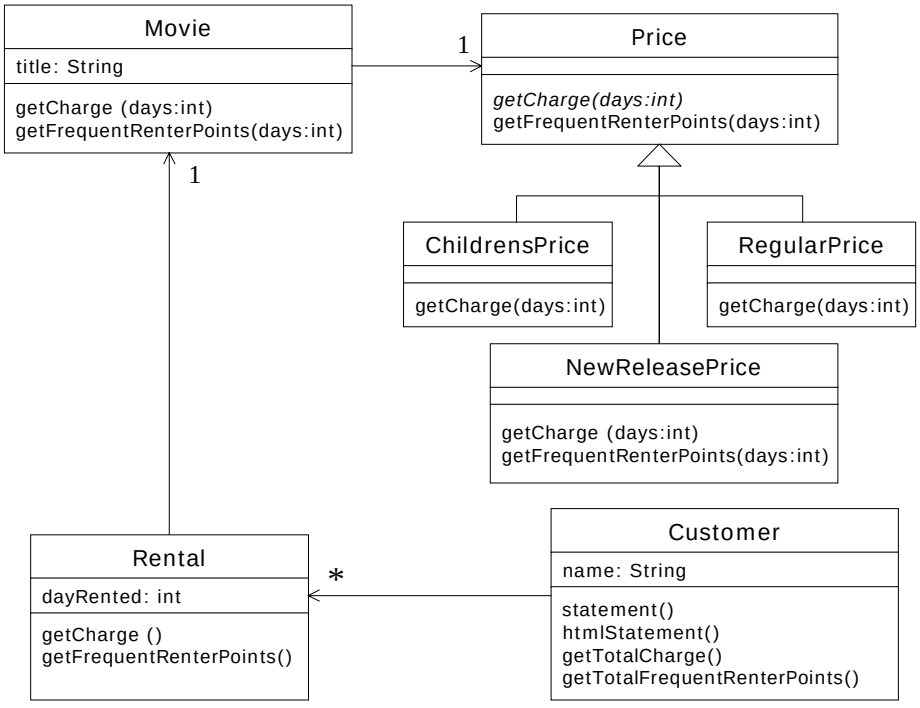


圖 1.17　加入 ***State*** pattern（範式）之後的 class diagram

1.5 結語

這是一個簡單的例子，但我希望它能讓你對於「重構是什麼樣子」有一點感覺。例中我已經示範了數個重構準則，包括 *Extract Method*（110）、*Move Method*（142）、*Replace Conditional with Polymorphism*（255）、*Self Encapsulate Field*（171）、*Replace Type Code with State/Strategy*（227）。所有這些重構行爲都使責任的分配更合理，程式碼的維護更輕鬆。重構後的程式風格，將十分不同於程序式（*procedural*）風格，後者也許是某些人習慣的風格。不過一旦你習慣了這種重構後的風格，就很難再回到（再滿足於）結構化風格了。

這個例子給你上的最重要一課是「重構的節奏」：測試、小修改、測試、小修改、測試、小修改…。正是這種節奏讓重構得以快速而安全地前進。

如果你看懂了前面的例子，你應該已經理解重構是怎麼回事了。現在，讓我們了解一些背景、原理和理論（不太多！）。

譯註：中文版（本書）支援網站提供本章重構過程中的各階段完整程式碼（共分七個階段），並含測試。網址見於封底。

2

重構原則

Principles in Refactoring

前面所舉的例子應該已經讓你對重構（**refactoring**）有了一個良好的感受。現在，我們應該回頭看看重構的關鍵原則，以及重構時需要考慮的某些問題。

2.1 何謂重構

我總是不太樂意爲什麼東西下定義，因爲每個人對任何東西都有自己的定義。但是當你寫一本書時，你總得選擇自己滿意的定義。在重構這個題目上，我的定義以 Ralph Johnson 團隊和其他相關研究成果爲基礎。

首先要說明的是：視上下文不同，「重構」這個詞有兩種不同的定義。你可能會覺得這挺煩人的（我就是這麼想），不過處理自然語言本來就是件煩人的事，這只不過是又一個實例而已。

第一個定義是名詞形式：

> 重構（名詞）：對軟體內部結構的一種調整，目的是在不改變「軟體之可察行爲」前提下，提高其可理解性，降低其修改成本。

你可以在後續章節中找到許多重構範例，諸如 *Extract Method*（110）和 *Pull Up Field*（320）等等。一般而言重構都是對軟體的小改動，但重構可以包含另一個重構。例如 *Extract Class*（149）通常包含 *Move Method*（142）和 *Move Field*（146）。

「重構」的另一個用法是動詞形式：

> 重構（動詞）：使用一系列重構準則（手法），在不改變「軟體之可察行爲」前提下，調整其結構。

所以，在軟體開發過程中，你可能會花上數小時的時間進行重構，其間可能用上數十個不同的重構準則。

曾經有人這樣問我：『重構就只是整理程式碼嗎？』從某種角度來說，是的！但我認爲重構不止於此，因爲它提供了一種更高效且受控的程式碼整理技術。自從運用重構技術後，我發現自己對程式碼的整理比以前更有效率。這是因爲我知道該使用哪些重構準則，我也知道以怎樣的方式使用它們才能夠將錯誤減到最少，而且在每一個可能出錯的地方我都加以測試。

我的定義還需要往兩方面擴展。首先，重構的目的是使軟體更容易被理解和修改。你可以在軟體內部做很多修改，但必須對軟體「可受觀察之外部行爲」只造成很小變化，或甚至不造成變化。與之形成對比的是「效率最佳化」。和重構一樣，效率最佳化通常不會改變組件的行爲（除了執行速度），只會改變其內部結構。但是兩者出發點不同：效率最佳化往往使程式碼較難理解，但爲了得到所需的效率你不得不那麼做。

我要強調的第二點是：重構不會改變軟體「可受觀察之行爲」— 重構之後軟體功能一如以往。任何用戶，不論終端用戶或程式員，都不知道已有東西發生了變化。（譯註：「可受觀察之行爲」其實也包括效率，因爲效率是可以被觀察的。不過我想我們無需太挑剔這些用詞。）

兩頂帽子

上述第二點引出了 Kent Beck 的「兩頂帽子」比喻。使用重構技術開發軟體時，你把自己的時間分配給兩種截然不同的行爲：「添加新功能」和「重構」。添加新功能時，你不應該修改既有程式碼，只管添加新功能。透過測試（並讓測試正常執行），你可以衡量自己的工作進度。重構時你就不能再添加功能，只管改進程式結構。此時你不應該添加任何測試（除非發現先前遺漏的任何東西），只在絕對必要（用以處理介面變化）時才修改測試。

軟體開發過程中，你可能會發現自己經常變換帽子。首先你會嘗試添加新功能，然後你會意識到：如果把程式結構改一下，功能的添加會容易得多。於是你換一頂帽子，做一會兒重構工作。程式結構調整好後，你又換上原先的帽子，繼續添加新功能。新功能正常工作後，你又發現自己的編碼造成程式難以理解，於是你又換上重構帽子…。整個過程或許只花十分鐘，但無論何時你都應該清楚自己戴的是哪一頂帽子。

2.2 為何重構？

我不想把重構說成治百病的萬靈丹，它絕對不是所謂的「銀子彈」[1]。不過它的確很有價值，雖不是一顆銀子彈卻是一把「銀鉗子」，可以幫助你始終良好地控制自己的程式碼。重構是個工具，它可以（並且應該）爲了以下數個目的而被運用：

「重構」改進軟體設計

如果沒有重構，程式的設計會逐漸腐敗變質。當人們只爲短期目的，或是在完全理解整體設計之前，就貿然修改程式碼，程式將逐漸失去自己的結構，程式員愈來愈難透過閱讀源碼而理解原本設計。重構很像是在整理程式碼，你所做的就是讓所有東西回到應該的位置上。程式碼結構的流失是累積性的。愈難看出程式碼所代表的設計意涵，就愈難保護其中設計，於是該設計就腐敗得愈快。經常性的重構可以幫助程式碼維持自己該有的形態。

同樣完成一件事，設計不良的程式往往需要更多程式碼，這常常是因爲程式碼在不同的地方使用完全相同的述句做同樣的事。因此改進設計的一個重要方向就是消除重複程式碼（Duplicate Code）。這個動作的重要性著眼於未來。程式碼數量減少並不會使系統執行更快，因爲這對程式的執行軌跡幾乎沒有任何明顯影響。然而程式碼數量減少將使未來可能的程式修改動作容易得多。程式碼愈多，正確的修改就愈困難，因爲有更多程式碼需要理解。你在這兒做了點修改，系統卻不如預期那樣工作，因爲你未曾修改另一處 — 那兒的程式碼做著幾乎完全一樣的事情，只是所處環境略有不同。如果消除重複程式碼，你就可以確定程式碼將所有事物和行爲都只表述一次，惟一一次，這正是優秀設計的根本。

[1] 譯註：「銀子彈」（silver bullet）是美國家喻戶曉的比喻。美國民間流傳月圓之夜狼人出沒，只有以純銀子彈射穿狼人心臟，才能制服狼人。

「重構」使軟體更易被理解

從許多角度來說，所謂程式設計，便是與計算機交談。你編寫程式碼告訴計算機做什麼事，它的回應則是精確按照你的指示行動。你得及時填補「想要它做什麼」和「告訴它做什麼」之間的縫隙。這種編程模式的核心就是「準確說出吾人所欲」。除了計算機外，你的源碼還有其他讀者：數個月之後可能會有另一位程式員嘗試讀懂你的程式碼並做一些修改。我們很容易忘記這第二位讀者，但他才是最重要的。計算機是否多花了數個鐘頭進行編譯，又有什麼關係呢？如果一個程式員花費一周時間來修改某段程式碼，那才關係重大 — 如果他理解你的程式碼，這個修改原本只需一小時。

問題在於，當你努力讓程式運轉的時候，你不會想到未來出現的那個開發者。是的，是應該改變一下我們的開發節奏，對程式碼做適當修改，讓程式碼變得更易理解。重構可以幫助我們讓程式碼更易讀。一開始進行重構時，你的程式碼可以正常執行，但結構不夠理想。在重構上花一點點時間，就可以讓程式碼更好地表達自己的用途。這種編程模式的核心就是「準確說出你的意思」。

關於這一點，我沒必要表現得如此無私。很多時候那個「未來的開發者」就是我自己。此時重構就顯得尤其重要了。我是個很懶惰的程式員，我的懶惰表現形式之一就是：總是記不住自己寫過的程式碼。事實上對於任何立可查閱的東西我都故意不去記它，因爲我怕把自己的腦袋塞爆。我總是儘量把該記住的東西寫進程式裡頭，這樣我就不必記住它了。這麼一來我就不必太擔心 Old Peculier （譯註：一種有名的麥芽酒）[Jackson] 殺光我的腦細胞。

這種可理解性還有另一方面的作用。我利用重構來協助我理解不熟悉的程式碼。當我看到不熟悉的程式碼，我必須試著理解其用途。我先看兩行程式碼，然後對自己說：『噢，是的，它做了這些那些…』。有了重構這個強大武器在手，我不會滿足於這麼一點腦中體會。我會真正動手修改程式碼，讓它更好地反映出我的理解，然後重新執行，看它是否仍然正常運作，以此檢驗我的理解是否正確。

一開始我所做的重構都像這樣停留在細枝末節上。隨著程式碼漸趨簡潔，我發現自己可以看到一些以前看不到的設計層面的東西。如果不對程式碼做這些修改，也許我永遠看不見它們，因爲我的聰明才智不足以在腦子裡把這一切都想像出來。Ralph Johnson 把這種「早期重構」描述爲「擦掉窗戶上的污垢，使你看得更遠」。研究程式碼時我發現，重構把我帶到更高的理解層次上。如果沒有重構，我達不到這種層次。

「重構」助你找到臭蟲（bugs）

對程式碼的理解，可以幫助我找到臭蟲。我承認我不太擅長除錯。有些人只要盯著一大段程式碼就可以找出裡面的臭蟲，我可不行。但我發現如果我對程式碼進行重構，我就可以深入理解程式碼的作爲，並恰到好處地把新的理解反饋回去。搞清楚程式結構的同時，我也清楚了自己所做的一些假設，從這個角度來說，不找到臭蟲都難矣。

這讓我想起了 Kent Beck 經常形容自己的一句話：『我不是個偉大的程式員；我只是個有著一些優秀習慣的好程式員而已。』重構能夠幫助我更有效地寫出強固穩健（robust）的程式碼。

「重構」助你提高編程速度

終於，前面的一切都歸結到了這最後一點：重構幫助你更快速地開發程式。

聽起來有點違反直覺。當我談到重構，人們很容易看出它能夠提高品質。改善設計、提昇可讀性、減少錯誤，這些都是提高品質。但這難道不會降低開發速度嗎？

我強烈相信：良好設計是快速軟體開發的根本。事實上擁有良好設計才可能達成快速的開發。如果沒有良好設計，或許某一段時間內你的進展迅速，但惡劣的設計很快就讓你的速度慢下來。你會把時間花在除錯上面，無法添加新功能。修改時間愈來愈長，因爲你必須花愈來愈多的時間去理解系統、尋找重複程式碼。隨著你給最初程式打上一個又一個的補釘（patch），新特性需要更多程式碼才能實現。真是個惡性循環。

良好設計是維持軟體開發速度的根本。重構可以幫助你更快速地開發軟體，因爲它阻止系統腐敗變質，它甚至還可以提高設計品質。

2.3 何時重構？

當我談論重構，常常有人問我應該怎樣安排重構時間表。我們是不是應該每兩個月就專門安排兩個星期來進行重構呢？

幾乎任何情況下我都反對專門撥出時間進行重構。在我看來，重構本來就不是一件「特別撥出時間做」的事情，重構應該隨時隨地進行。你不應該爲重構而重構，你之所以重構，是因爲你想做別的什麼事，而重構可以幫助你把那些事做好。

三次法則（The Rule of Three）

Don Roberts 給了我一條準則：第一次做某件事時只管去做；第二次做類似的事會產生反感，但無論如何還是做了；第三次再做類似的事，你就應該重構。

事不過三，三則重構。（*Three strikes and you refactor.*）

添加功能時一併重構

最常見的重構時機就是我想給軟體添加新特性的時候。此時，重構的第一個原因往往是爲了幫助我理解需要修改的程式碼。這些程式碼可能是別人寫的，也可能是我自己寫的。無論何時只要我想理解程式碼所做的事，我就會問自己：是否能對這段程式碼進行重構，使我能更快理解它。然後我就會重構。之所以這麼做，部分原因是爲了讓我下次再看這段程式碼時容易理解，但最主要的原因是：如果在前進過程中把程式碼結構理清，我就可以從中理解更多東西。

在這裡，重構的另一個原動力是：程式碼的設計無法幫助我輕鬆添加我所需要的特性。我看著設計，然後對自己說：「如果用某種方式來設計，添加特性會簡單得多」。這種情況下我不會因爲自己過去的錯誤而懊惱 — 我用重構來彌補它。之所以這麼做，部分原因是爲了讓未來增加新特性時能夠更輕鬆一些，但最主要的原因還是：我發現這是最快捷的途徑。重構是一個快速流暢的過程，一旦完成重構，新特性的添加就會更快速、更流暢。

修補錯誤時一併重構

除錯過程中運用重構，多半是爲了讓程式碼更具可讀性。當我看著程式碼並努力理解它的時候，我用重構幫助改善自己的理解。我發現以這種程序來處理程式碼，常常能夠幫助我找出臭蟲。你可以這麼想：如果收到一份錯誤報告，這就是需要重構的信號，因爲顯然程式碼還不夠清晰 — 不夠清晰到讓你一目了然發現臭蟲。

復審程式碼時一併重構

很多公司都會做常態性的程式碼復審工作（code reviews），因爲這種活動可以改善開發狀況。這種活動有助於在開發團隊中傳播知識，也有助於讓較有經驗的開發者把知識傳遞給比較欠缺經驗的人，並幫助更多人理解大型軟體系統中的更多部分。程式碼復審工作對於編寫清晰程式碼也很重要。我的程式碼也許對我自己來說很清晰，對他人則不然。這是無法避免的，因爲要讓開發者設身處地爲那些不熟悉自己所做所爲的人設想，實在太困難了。程式碼復審也讓更多人有機會提出有用的建議，畢竟我在一個星期之內能夠想出的好點子很有限。如果能得到別人的幫助，我的生活會舒服得多，所以我總是期待更多復審。

我發現，重構可以幫助我復審別人的程式碼。開始重構前我可以先閱讀程式碼，得到一定程度的理解，並提出一些建議。一旦想到一些點子，我就會考慮是否可以通過重構立即輕鬆地實現它們。如果可以，我就會動手。這樣做了幾次以後，我可以把程式碼看得更清楚，提出更多恰當的建議。我不必想像程式碼「應該是什麼樣」，我可以「看見」它是什麼樣。於是我可以獲得更高層次的認識。如果不進行重構，我永遠無法得到這樣的認識。

重構還可以幫助程式碼復審工作得到更具體的結果。不僅獲得建議，而且其中許多建議能夠立刻實現。最終你將從實踐中得到比以往多得多的成就感。

爲了讓過程正常運轉，你的復審團隊必須保持精練。就我的經驗，最好是一個復審者搭配一個原作者，共同處理這些程式碼。復審者提出修改建議，然後兩人共同判斷這些修改是否能夠透過重構輕鬆實現。果真能夠如此，就一起著手修改。

如果是比較大的設計復審工作，那麼，在一個較大團隊內保留多種觀點通常會更好一些。此時直接展示程式碼往往不是最佳辦法。我喜歡運用 UML 示意圖展現設計，並以 CRC 卡展示軟體情節。換句話說，我會和某個團隊進行設計復審，而和個別（單一）復審者進行程式碼復審。

極限編程（Extreme Programming）[Beck, XP] 中的「搭檔（成對）編程」（Pair Programming）形式，把程式碼復審的積極性發揮到了極致。一旦採用這種形式，所有正式開發任務都由兩名開發者在同一台機器上進行。這樣便在開發過程中形成隨時進行的程式碼復審工作，而重構也就被包含在開發過程內了。

爲什麼重構有用（Why Refactoring Works）

— *Kent Beck*

程式有兩面價值：「今天可以爲你做什麼」和「明天可以爲你做什麼」。大多數時候，我們都只關注自己今天想要程式做什麼。不論是修復錯誤或是添加特性，我們都是爲了讓程式能力更強，讓它在今天更有價值。

但是系統今天（當下）的行爲，只是整個故事的一部分，如果沒有認淸這一點，你無法長期從事編程工作。如果你「爲求完成今天任務」而採取的手法使你不可能在明天完成明天的任務，那麼你還是失敗。但是，你知道自己今天需要什麼，卻不一定知道自己明天需要什麼。也許你可以猜到明天的需求，也許吧，但肯定還有些事情出乎你的意料。

對於今天的工作，我了解得很充分；對於明天的工作，我了解得不夠充分。但如果我純粹只是爲今天工作，明天我將完全無法工作。

重構是一條擺脫束縛的道路。如果你發現昨天的決定已經不適合今天的情況，放心改變這個決定就是，然後你就可以完成今天的工作了。明天，喔，明天回頭看今天的理解也許覺得很幼稚，那時你還可以改變你的理解。

是什麼讓程式如此難以相與？下筆此刻，我想起四個原因，它們是：

- 難以閱讀的程式，難以修改。
- 邏輯重複（duplicated logic）的程式，難以修改。
- 添加新行爲時需修改既有程式碼者，難以修改。
- 帶複雜條件邏輯（complex conditional logic）的程式，難以修改。

因此，我們希望程式 (1) 容易閱讀，(2) 所有邏輯都只在惟一地點指定，(3) 新的改動不會危及現有行爲，(4) 儘可能簡單表達條件邏輯（conditional logic）。

重構是這樣一個過程：它在一個目前可執行的程式上進行，企圖在「不改變程式行爲」的情況下賦予上述美好性質，使我們能夠繼續保持高速開發，從而增加程式的價值。

2.4 怎麼對經理說？

「該怎麼跟經理說重構的事？」這是我最常被問到的問題之一。如果這位經理懂技術，那麼向他介紹重構應該不會很困難。如果這位經理只對品質感興趣，那麼問題就集中到了「品質」上面。此時，在復審過程中使用重構，就是一個不錯的辦法。大量研究結果顯示，「技術復審」是減少錯誤、提高開發速度的一條重要

途徑。隨便找一本關於復審、審查或軟體開發程序的書看看，從中找些最新引證，應該可以讓大多數經理認識復審的價值。然後你就可以把重構當作「將復審意見引入程式碼內」的方法來使用，這很容易。

當然，很多經理嘴巴上說自己「品質驅動」，其實更多是「進度驅動」。這種情況下我會給他們一個較有爭議的建議：不要告訴經理！

這是在搞破壞嗎？我不這樣想。軟體開發者都是專業人士。我們的工作就是儘可能快速創造出高效軟體。我的經驗告訴我，對於快速創造軟體，重構可帶來巨大幫助。如果需要添加新功能，而原本設計卻又使我無法方便地修改，我發現先「進行重構」再「添加新功能」會更快些。如果要修補錯誤，我需得先理解軟體工作方式，而我發現重構是理解軟體的最快方式。受進度驅動的經理要我儘可能快速完事，至於怎麼完成，那就是我的事了。我認爲最快的方式就是重構，所以我就重構。

間接層和重構（Indirection and Refactoring）

— *Kent Beck*

『計算機科學是這樣一門科學：它相信所有問題都可以藉由多一個間接層（indirection）來解決。』— *Dennis DeBruler*

由於軟體工程師對間接層如此醉心，你應該不會驚訝大多數重構都爲程式引入了更多間接層。重構往往把大型物件拆成數個小型物件，把大型函式拆成數個小型函式。

但是，間接層是一柄雙刃劍。每次把一個東西分成兩份，你就需要多管理一個東西。如果某個物件委託（*delegate*）另一物件，後者又委託另一物件，程式會愈加難以閱讀。基於這個觀點，你會希望儘量減少間接層。

別急，夥計！間接層有它的價值。下面就是間接層的某些價值：

- **允許邏輯共享**（To enable sharing of logic）。比如說一個子函式（submethod）在兩個不同的地點被呼叫，或 superclass 中的某個函式被所有 subclasses 共享。
- **分開解釋「意圖」和「實作」**（To explain intention and implementation separately）。你可以選擇每個 class 和函式的名字，這給了你一個解釋自己意圖的機會。class 或函式內部則解釋實現這個意圖的作法。如果 class 和函式內部又以「更小單元的意圖」來編寫，你所寫的程式碼就可以「與其結構中的大部分重要資訊溝通」。
- **將變化加以隔離**（To isolate change）。很可能我在兩個不同地點使用同一物件，其中一個地點我想改變物件行爲，但如果修改了它，我就要冒「同時影響兩處」的風險。爲此我做出一個 subclass，並在需要修改處引用這個 subclass。現在，我可以修

改這個 subclass 而不必承擔「無意中影響另一處」的風險。

- **將條件邏輯加以編碼**（To encode conditional logic）。物件有一種匪夷所思的機制：多型訊息（polymorphic messages），可以靈活彈性而清晰地表達條件邏輯。只要顯式條件邏輯被轉化爲訊息（message[2]）形式，往往便能降低程式碼的重複、增加清晰度、並提高彈性。

這就是重構遊戲：在保持系統現有行爲的前提下，如何才能提高系統的品質或降低其成本，從而使它更有價值？

這個遊戲中最常見的變數就是：你如何看待你自己的程式。找出一個缺乏「間接層利益」之處，在不修改現有行爲的前提下，爲它加入一個間接層。現在你獲得了一個更有價值的程式，因爲它有較高的品質，讓我們在明天（未來）受益。

請將這種方法與「小心翼翼的事前設計」做個比較。推測性設計總是試圖在任何一行程式碼誕生之前就先讓系統擁有所有優秀品質，然後程式員將程式碼塞進這個強健的骨架中就行了。這個過程的問題在於：太容易猜錯。如果運用重構，你就永遠不會面臨全盤錯誤的危險。程式自始至終都能保持一致的行爲，而你又有機會爲程式添加更多價值不菲的品質。

還有一種比較少見的重構遊戲：找出不值得的間接層，並將它拿掉。這種間接層常以中介函式（intermediate methods）形式出現，也許曾經有過貢獻，但芳華已逝。它也可能是個組件，你本來期望在不同地點共享它，或讓它表現出多型性（polymorphism），最終卻只在一處使用之。如果你找到這種「寄生式間接層」，請把它扔掉。如此一來你會獲得一個更有價值的程式，不是因爲它取得了更多（先前所列）的四種優秀品質，而是因爲它以更少的間接層獲得一樣多的優秀品質。

2.5 重構的難題

學習一種可以大幅提高生產力的新技術時，你總是難以察覺其不適用的場合。通常你在一個特定場景中學習它，這個場景往往是個專案。這種情況下你很難看出什麼會造成這種新技術成效不彰或甚至形成危害。十年前，物件技術（object tech.）的情況也是如此。那時如果有人問我「何時不要使用物件」，我很難回答。並非我認爲物件十全十美、沒有侷限性 — 我最反對這種盲目態度，而是儘管我知道它的好處，但確實不知道其侷限性在哪兒。

[2] 譯註：此處的「訊息」（message）是指物件導向古典論述中的意義。在那種場合中，「呼叫某個函式（method）」就是「送出訊息（message）給某個物件（object）」。

現在，重構的處境也是如此。我們知道重構的好處，我們知道重構可以給我們的工作帶來垂手可得的改變。但是我們還沒有獲得足夠的經驗，我們還看不到它的侷限性。

這一小節比我希望的要短。暫且如此吧。隨著更多人學會重構技巧，我們也將對它有更多了解。對你而言這意味：雖然我堅決認爲你應該嘗試一下重構，獲得它所提供的利益，但在此同時，你也應該時時監控其過程，注意尋找重構可能引入的問題。請讓我們知道你所遭遇的問題。隨著對重構的了解日益增多，我們將找出更多解決辦法，並清楚知道哪些問題是真正難以解決的。

資料庫（Databases）

「重構」經常出問題的一個領域就是資料庫。絕大多數商用程式都與它們背後的 database schema（資料庫表格結構）緊密耦合（coupled）在一起，這也是 database schema 如此難以修改的原因之一。另一個原因是資料遷移（migration）。就算你非常小心地將系統分層（layered），將 database schema 和物件模型（object model）間的依賴降至最低，但 database schema 的改變還是讓你不得不遷移所有資料，這可能是件漫長而煩瑣的工作。

在「非物件資料庫」（nonobject databases）中，解決這個問題的辦法之一就是：在物件模型（object model）和資料庫模型（database model）之間插入一個分隔層（separate layer），這就可以隔離兩個模型各自的變化。升級某一模型時無需同時升級另一模型，只需升級上述的分隔層即可。這樣的分隔層會增加系統複雜度，但可以給你很大的靈活度。如果你同時擁有多個資料庫，或如果資料庫模型較爲複雜使你難以控制，那麼即使不進行重構，這分隔層也是很重要的。

你無需一開始就插入分隔層，可以在發現物件模型變得不穩定時再產生它。這樣你就可以爲你的改變找到最好的槓桿效應。

對開發者而言，物件資料庫既有幫助也有妨礙。某些物件導向資料庫提供不同版本的物件之間的自動遷移功能，這減少了資料遷移時的工作量，但還是會損失一定時間。如果各資料庫之間的資料遷移並非自動進行，你就必須自行完成遷移工作，這個工作量可是很大的。這種情況下你必須更加留神 classes 內的資料結構變化。你仍然可以放心將 classes 的行爲轉移過去，但轉移欄位（field）時就必須格外小心。資料尚未被轉移前你就得先運用存取函式（accessors）造成「資料已經轉移」的假象。一旦你確定知道「資料應該在何處」時，就可以一次性地將資料遷

移過去。這時惟一需要修改的只有存取函式（accessors），這也降低了錯誤風險。

修改介面（Changing Interfaces）

關於物件，另一件重要事情是：它們允許你分開修改軟體模組的實作（implementation）和介面（interface）。你可以安全地修改某物件內部而不影響他人，但對於介面要特別謹慎 — 如果介面被修改了，任何事情都有可能發生。

一直對重構帶來困擾的一件事就是：許多重構手法的確會修改介面。像 *Rename Method*（273）這麼簡單的重構手法所做的一切就是修改介面。這對極爲珍貴的封裝概念會帶來什麼影響呢？

如果某個函式的所有呼叫動作都在你的控制之下，那麼即使修改函式名稱也不會有任何問題。哪怕面對一個 public 函式，只要能取得並修改其所有呼叫者，你也可以安心地將這個函式易名。只有當需要修改的介面係被那些「找不到，即使找到也不能修改」的程式碼使用時，介面的修改才會成爲問題。如果情況真是如此，我就會說：這個介面是個「已發佈介面」（published interface）— 比公開介面（public interface）更進一步。介面一旦發佈，你就再也無法僅僅修改呼叫者而能夠安全地修改介面了。你需要一個略爲複雜的程序。

這個想法改變了我們的問題。如今的問題是：該如何面對那些必須修改「已發佈介面」的重構手法？

簡言之，如果重構手法改變了已發佈介面（published interface），你必須同時維護新舊兩個介面，直到你的所有用戶都有時間對這個變化做出反應。幸運的是這不太困難。你通常都有辦法把事情組織好，讓舊介面繼續工作。請儘量這麼做：讓舊介面呼叫新介面。當你要修改某個函式名稱時，請留下舊函式，讓它呼叫新函式。千萬不要拷貝函式實作碼，那會讓你陷入「重複程式碼」（duplicated code）的泥淖中難以自拔。你還應該使用 Java 提供的 deprecation（反對）設施，將舊介面標記爲 "deprecated"。這麼一來你的呼叫者就會注意到它了。

這個過程的一個好例子就是 Java 容器類別（群集類別，collection classes）。Java 2 的新容器取代了原先一些容器。當 Java 2 容器發佈時，JavaSoft 花了很大力氣來爲開發者提供一條順利遷徙之路。

「保留舊介面」的辦法通常可行，但很煩人。起碼在一段時間裡你必須建造（build）並維護一些額外的函式。它們會使介面變得複雜，使介面難以使用。還好我們有另一個選擇：不要發佈（publish）介面。當然我不是說要完全禁止，因爲很明顯

你必得發佈一些介面。如果你正在建造供外部使用的 APIs，像 Sun 所做的那樣，肯定你必得發佈介面。我之所以說儘量不要發佈，是因爲我常常看到一些開發團隊公開了太多介面。我曾經看到一支三人團隊這麼工作：每個人都向另外兩人公開發佈介面。這使他們不得不經常來回維護介面，而其實他們原本可以直接進入程式庫，逕行修改自己管理的那一部分，那會輕鬆許多。過度強調「程式碼擁有權」的團隊常常會犯這種錯誤。發佈介面很有用，但也有代價。所以除非真有必要，別發佈介面。這可能意味需要改變你的程式碼擁有權觀念，讓每個人都可以修改別人的程式碼，以運應介面的改動。以搭檔（成對）編程（Pair Programming）完成這一切通常是個好主意。

不要過早發佈（published）介面。請修改你的程式碼擁有權政策，使重構更順暢。

Java 之中還有一個特別關於「修改介面」的問題：在 throws 子句中增加一個異常。這並不是對署名式（signature）的修改，所以你無法以 *delegation*（委託手法）涵蓋它。但如果用戶程式碼不做出相應修改，編譯器不會讓它通過。這個問題很難解決。你可以爲這個函式選擇一個新名字，讓舊函式呼叫它，並將這個新增的 *checked exception*（可控式異常）轉換成一個 *unchecked exception*（不可控異常）。你也可以拋出一個 *unchecked* 異常，不過這樣你就會失去檢驗能力。如果你那麼做，你可以警告呼叫者：這個 *unchecked* 異常日後會變成一個 *checked* 異常。這樣他們就有時間在自己的程式碼中加上對此異常的處理。出於這個原因，我總是喜歡爲整個 package 定義一個 superclass 異常（就像 `java.sql` 的 `SQLException`），並確保所有 public 函式只在自己的 throws 子句中宣告這個異常。這樣我就可以隨心所欲地定義 subclass 異常，不會影響呼叫者，因爲呼叫者永遠只知道那個更具一般性的 superclass 異常。

難以藉由重構手法完成的設計改動

透過重構，可以排除所有設計錯誤嗎？是否存在某些核心設計決策，無法以重構手法修改？在這個領域裡，我們的統計數據尚不完整。當然某些情況下我們可以很有效地重構，這常常令我們倍感驚訝，但的確也有難以重構的地方。比如說在一個專案中，我們很難（但還是有可能）將「無安全需求（no security requirements）情況下構造起來的系統」重構爲「安全性良好的（good security）系統」。

這種情況下我的辦法就是「先想像重構的情況」。考慮候選設計方案時，我會問自己：將某個設計重構爲另一個設計的難度有多大？如果看上去很簡單，我就不

必太擔心選擇是否得當，於是我就會選最簡單的設計，哪怕它不能涵蓋所有潛在需求也沒關係。但如果預先看不到簡單的重構辦法，我就會在設計上投入更多力氣。不過我發現，這種情況很少出現。

何時不該重構？

有時候你根本不應該重構 — 例如當你應該重新編寫所有程式碼的時候。有時候既有程式碼實在太混亂，重構它還不如重新寫一個來得簡單。做出這種決定很困難，我承認我也沒有什麼好準則可以判斷何時應該放棄重構。

重寫（而非重構）的一個清楚訊號就是：現有程式碼根本不能正常運作。你可能只是試著做點測試，然後就發現程式碼中滿是錯誤，根本無法穩定運作。記住，重構之前，程式碼必須起碼能夠在大部分情況下正常運作。

一個折衷辦法就是：將「大塊頭軟體」重構爲「封裝良好的小型組件」。然後你就可以逐一對組件作出「重構或重建」的決定。這是一個頗具希望的辦法，但我還沒有足夠數據，所以也無法寫出優秀的指導原則。對於一個重要的古老系統，這肯定會是一個很好的方向。

另外，如果專案已近最後期限，你也應該避免重構。在此時機，從重構過程贏得的生產力只有在最後期限過後才能體現出來，而那個時候已經時不我予。Ward Cunningham 對此有一個很好的看法。他把未完成的重構工作形容爲「債務」。很多公司都需要借債來使自己更有效地運轉。但是借債就得付利息，過於複雜的程式碼所造成的「維護和擴展的額外開銷」就是利息。你可以承受一定程度的利息，但如果利息太高你就會被壓垮。把債務管理好是很重要的，你應該隨時藉由重構來償還一部分債務。

如果專案已經非常接近最後期限，你不應該再分心於重構，因爲已經沒有時間了。不過多個專案經驗顯示：重構的確能夠提高生產力。如果最後你沒有足夠時間，通常就表示你其實早該進行重構。

2.6 重構與設計

「重構」肩負一項特別任務：它和設計彼此互補。初學編程的時候，我埋頭就寫程式，渾渾噩噩地進行開發。然而很快我便發現，「事先設計」（upfront design）可以助我節省回頭工的高昂成本。於是我很快加強這種「預先設計」風格。許多

人都把設計看作軟體開發的關鍵環節，而把編程（programming）看作只是機械式的低級勞動。他們認爲設計就像畫工程圖而撰碼就像施工。但是你要知道，軟體和真實器械有著很大的差異。軟體的可塑性更強，而且完全是思想產品。正如 Alistair Cockburn 所說：『有了設計，我可以思考更快，但是其中充滿小漏洞。』

有一種觀點認爲：重構可以成爲「預先設計」的替代品。這意思是你根本不必做任何設計，只管按照最初想法開始編碼，讓程式碼有效運作，然後再將它重構成型。事實上這種辦法真的可行。我的確看過有人這麼做，最後獲得設計良好的軟體。極限編程（Extreme Programming）[Beck, XP] 的支持者極力提倡這種辦法。

儘管如上所言，只運用重構也能收到效果，但這並不是最有效的途徑。是的，即使極限編程（Extreme Programming）愛好者也會進行預先設計。他們會使用 CRC 卡或類似的東西來檢驗各種不同想法，然後才得到第一個可被接受的解決方案，然後才能開始撰碼，然後才能重構。關鍵在於：重構改變了「預先設計」的角色。如果沒有重構，你就必須保證「預先設計」正確無誤，這個壓力太大了。這意味如果將來需要對原始設計做任何修改，代價都將非常高昂。因此你需要把更多時間和精力放在預先設計上，以避免日後修改。

如果你選擇重構，問題的重點就轉變了。你仍然做預先設計，但是不必一定找出正確的解決方案。此刻的你只需要得到一個足夠合理的解決方案就夠了。你很肯定地知道，在實現這個初始解決方案的時候，你對問題的理解也會逐漸加深，你可能會察覺最佳解決方案和你當初設想的有些不同。只要有重構這項武器在手，就不成問題，因爲重構讓日後的修改成本不再高昂。

這種轉變導致一個重要結果：軟體設計朝向簡化前進了一大步。過去未曾運用重構時，我總是力求得到靈活的解決方案。任何一個需求都讓我提心吊膽地猜疑：在系統壽命期間，這個需求會導致怎樣的變化？由於變更設計的代價非常高昂，所以我希望建造一個足夠靈活、足夠強固的解決方案，希望它能承受我所能預見的所有需求變化。問題在於：要建造一個靈活的解決方案，所需的成本難以估算。靈活的解決方案比簡單的解決方案複雜許多，所以最終得到的軟體通常也會更難維護 — 雖然它在我預先設想的方向上的確是更加靈活。就算幸運走在預先設想的方向上，你也必須理解如何修改設計。如果變化只出現在一兩個地方，那不算大問題。然而變化其實可能出現在系統各處。如果在所有可能的變化出現地點都建立起靈活性，整個系統的複雜度和維護難度都會大大提高。當然，如果最後發現所有這些靈活性都毫無必要，這才是最大的失敗。你知道，這其中肯定有些靈

活性的確派不上用場，但你卻無法預測到底是哪些派不上用場。爲了獲得自己想要的靈活性，你不得不加入比實際需要更多的靈活性。

有了重構，你就可以通過一條不同的途徑來應付變化帶來的風險。你仍舊需要思考潛在的變化，仍舊需要考慮靈活的解決方案。但是你不必再逐一實現這些解決方案，而是應該問問自己：『把一個簡單的解決方案重構成這個靈活的方案有多大難度？』如果答案是「相當容易」（大多數時候都如此），那麼你就只需實現目前的簡單方案就行了。

重構可以帶來更簡單的設計，同時又不損失靈活性，這也降低了設計過程的難度，減輕了設計壓力。一旦對重構帶來的簡單性有更多感受，你甚至可以不必再預先思考前述所謂的靈活方案 — 一旦需要它，你總有足夠的信心去重構。是的，當下只管構築可執行的最簡化系統，至於靈活而複雜的設計，唔，多數時候你都不會需要它。

勞而無獲

— *Ron Jeffries*

Chrysler Comprehensive Compensation（克萊斯勒綜合薪資系統）的支付過程太慢了。雖然我們的開發還沒結束，這個問題卻已經開始困擾我們，因爲它已經拖累了測試速度。

Kent Beck、Martin Fowler 和我決定解決這個問題。等待大夥兒會合的時間裡，憑著我對這個系統的全盤了解，我開始推測：到底是什麼讓系統變慢了？我想到數種可能，然後和夥伴們談了幾種可能的修改方案。最後，關於「如何讓這個系統執行更快」，我們提出了一些真正的好點子。

然後，我們拿 Kent 的量測工具度量了系統效率。我一開始所想的可能性竟然全都不是問題肇因。我們發現：系統把一半時間用來創建「日期」實體（instance）。更有趣的是，所有這些實體都有相同的值。

於是我們觀察日期的創建邏輯，發現有機會將它最佳化。日期原本是由字串轉換而生，即使無外部輸入也是如此。之所以使用字串轉換方式，完全是爲了方便鍵盤輸入。好，也許我們可以將它最佳化。

於是我們觀察日期怎樣被這個程式運用。我們發現，很多日期物件都被用來產生「日期區間」實體（instance）。「日期區間」是個物件，由一個起始日期和一個結束日期組成。仔細追蹤下去，我們發現絕大多數日期區間是空的！

處理日期區間時我們遵循這樣一個規則：如果結束日期在起始日期之前，這個日期區間就該是空的。這是一條很好的規則，完全符合這個 class 的需要。採用此一規則後不

久，我們意識到，創建一個「起始日期在結束日期之後」的日期區間，仍然不算是清晰的程式碼，於是我們把這個行爲提煉到一個 ***factory method***（譯註：一個著名的設計範式，見《*Design Patterns*》），由它專門創建「空的日期區間」。

我們做了上述修改，使程式碼更加清晰，卻意外得到了一個驚喜。我們創建一個固定不變的「空日期區間」物件，並讓上述調整後的 ***factory method*** 每次都傳回該物件，而不再每次都創建新物件。這一修改把系統速度提昇了幾乎一倍，足以讓測試速度達到可接受程度。這只花了我們大約五分鐘。

我和團隊成員（Kent 和 Martin 謝絕參加）認真推測過：我們瞭若指掌的這個程式中可能有什麼錯誤？我們甚至憑空做了些改進設計，卻沒有先對系統的真實情況進行量測。

我們完全錯了。除了一場很有趣的交談，我們什麼好事都沒做。

教訓：哪怕你完全了解系統，也請實際量測它的效率，不要臆測。臆測會讓你學到一些東西，但十有八九你是錯的。

2.7 重構與效率/性能（Performance）

譯註：在我的接觸經驗中，performance 一詞被不同的人予以不同的解釋和認知：效率、性能、效能。不同地區（例如臺灣和大陸）的習慣用法亦不相同。本書一遇 performance 我便譯爲效率。efficient 譯爲高效，effective 譯爲有效。

關於重構，有一個常被提出的問題：它對程式的效率將造成怎樣的影響？爲了讓軟體易於理解，你常會作出一些使程式執行變慢的修改。這是個重要的問題。我並不贊成爲了提高設計的純潔性或把希望寄託於更快的硬體身上，而忽略了程式效率。已經有很多軟體因爲速度太慢而被用戶拒絕，日益提高的機器速度亦只不過略微放寬了速度方面的限制而已。但是，換個角度說，雖然重構必然會使軟體執行更慢，但它也使軟體的效率最佳化更易進行。除了對效率有嚴格要求的即時（real time）系統，其他任何情況下「編寫快速軟體」的秘密就是：首先寫出可調（tunable）軟體，然後調整它以求獲得足夠速度。

我看過三種「編寫快速軟體」的方法。其中最嚴格的是「時間預算法」（time budgeting），這通常只用於效率要求極高的即時系統。如果使用這種方法，分解你的設計時就要做好預算，給每個組件預先分配一定資源 — 包括時間和執行軌跡（footprint）。每個組件絕對不能超出自己的預算，就算擁有「可在不同組件之間調度預配時間」的機制也不行。這種方法高度重視效率，對於心律調節器一類的系統是必須的，因爲在這樣的系統中遲來的數據就是錯誤的數據。但對其他類系統（例如我經常開發的企業資訊系統）而言，如此追求高效率就有點過份了。

第二種方法是「持續關切法」（constant attention）。這種方法要求任何程式員在任何時間做任何事時，都要設法保持系統的高效率。這種方式很常見，感覺上很有吸引力，但通常不會起太大作用。任何修改如果是爲了提高效率，通常會使程式難以維護，因而減緩開發速度。如果最終得到的軟體的確更快了，那麼這點損失尙有所值，可惜通常事與願違，因爲效率改善一旦被分散到程式各角落，每次改善都只不過是從「對程式行爲的一個狹隘視角」出發而已。

關於效率，一件很有趣的事情是：如果你對大多數程式進行分析，你會發現它把大半時間都耗費在一小半程式碼身上。如果你一視同仁地最佳化所有程式碼，90% 的最佳化工作都是白費勁兒，因爲被你最佳化的程式碼有許多難得被執行起來。你花時間做最佳化是爲了讓程式執行更快，但如果因爲缺乏對程式的清楚認識而花費時間，那些時間都是被浪費掉了。

第三種效率提昇法係利用上述的 "90%" 統計數據。採用這種方法時，你以一種「良好的分解方式」（well-factored manner）來建造自己的程式，不對效率投以任何關切，直至進入效率最佳化階段 — 那通常是在開發後期。一旦進入該階段，你再按照某個特定程序來調整程式效率。

在效率最佳化階段中，你首先應該以一個量測工具監控程式的執行，讓它告訴你程式中哪些地方大量消耗時間和空間。這樣你就可以找出效率熱點（hot spot）所在的一小段程式碼。然後你應該集中關切這些熱點，並使用前述「持續關切法」中的最佳化手段來最佳化它們。由於你把注意力都集中在熱點上，較少的工作量即可顯現較好的成果。即便如此你還是必須保持謹慎。和重構一樣，你應該小幅度修改。每走一步都需要編譯、測試、再次量測。如果沒能提高效率，就應該撤銷此次修改。你應該繼續這個「發現熱點、去除熱點」的過程，直到獲得客戶滿意的效率爲止。關於這項技術，McConnell [McConnell] 爲我們提供了更多資訊。

一個被良好分解（well-factored）的程式可從兩方面幫助此種最佳化形式。首先，它讓你有比較充裕的時間進行效率調整（performance tuning），因爲有分解良好的程式碼在手，你就能夠更快速地添加功能，也就有更多時間用在效率問題上（準確的量測則保證你把這些時間投資在恰當地點）。其次，面對分解良好的程式，你在進行效率分析時便有較細的粒度（granularity），於是量測工具把你帶入範圍較小的程式段落中，而效率的調整也比較容易些。由於程式碼更加清晰，因此你能夠更好地理解自己的選擇，更清楚哪種調整起關鍵作用。

我發現重構可以幫助我寫出更快的軟體。短程看來，重構的確會使軟體變慢，但它使最佳化階段中的軟體效率調整更容易。最終我還是有賺頭。

2.8 重構起源何處？

我曾經努力想找出重構（refactoring）一詞的真正起源，但最終失敗了。優秀程式員肯定至少會花一些時間來清理自己的程式碼。這麼做是因爲，他們知道簡潔的程式碼比雜亂無章的程式碼更容易修改，而且他們知道自己幾乎無法一開始就寫出簡潔的程式碼。

重構不止如此。本書中我把重構看作整個軟體開發過程的一個關鍵環節。最早認識重構重要性的兩個人是 Ward Cunningham 和 Kent Beck，他們早在 1980s 之前就開始使用 Smalltalk，那是個特別適合重構的環境。Smalltalk 是一個十分動態的環境，你可以很快寫出極具功能的軟體。Smalltalk 的「編譯/連結/執行」週期非常短，因此很容易快速修改程式碼。它是物件導向，所以也能夠提供強大工具，最大限度地將修改的影響隱藏於定義良好的介面背後。Ward 和 Kent 努力發展出一套適合這類環境的軟體開發程序（如今 Kent 把這種風格叫作極限編程 [Beck, XP]）。他們意識到：重構對於提高他們的生產力非常重要。從那時起他們就一直在工作中運用重構技術，在嚴肅而認真的軟體專案中使用它，並不斷精煉這個程序。

Ward 和 Kent 的思想對 Smalltalk 社群產生了極大影響，重構概念也成爲 Smalltalk 文化中的一個重要元素。Smalltalk 社群的另一位領袖是 Ralph Johnson，伊利諾大學烏爾班納分校教授，著名的「四巨頭」[3][Gang of Four] 之一。Ralph 最大的興趣之一就是開發軟體框架（framework）。他揭示了重構對於靈活高效框架的開發幫助。

Bill Opdyke 是 Ralph 的博士班學生，對框架也很感興趣。他看到重構的潛在價值，並看到重構應用於 Smalltalk 之外的其他語言的可能性。他的技術背景是電話交換系統的開發。在這種系統中，大量的複雜情況與時俱增，而且非常難以修改。Bill 的博士研究就是從工具構築者的角度來看待重構。通過研究，Bill 發現：在 C++ framework 開發案中，重構很有用。他也研究了極有必要的「語意保持性（semantics-preserving）重構」及其證明方式，以及如何以工具實現重構。時至今日，Bill 的博士論文 [Opdyke] 仍然是重構領域中最有價值、最豐碩的研究成果。此外他爲本書撰寫了第 13 章。

[3] 譯註：Ralph Johnson 和另外三位先生 Erich Gamma, Richard Helm, John Vlissides 合寫了軟體開發界馳名的《*Design Patterns*》，人稱四巨頭（Gang of Four）。

我還記得 1992 年 OOPSLA 大會上見到 Bill 的情景。我們坐在一間咖啡廳裡，討論當時我正爲保健業務構築的一個概念框架（conceptual framework）中的某些工作。Bill 跟我談起他的研究成果，我還記得自己當時的想法：『有趣，但並非真的那麼重要』。唉，我完全錯了。

John Brant 和 Don Roberts 將重構中的「工具」構想發揚光大，開發了一個名爲「重構瀏覽器」（Refactoring Browser）的 Smalltalk 重構工具。他們撰寫了本書第 14 章，其中對重構工具做了更多介紹。

那麼，我呢？我一直有清理程式碼的傾向，但從來沒有想到這會有那麼重要。後來我和 Kent 一起做了個案子，看到他使用重構手法，也看到重構對生產效率和產品品質帶來的影響。這份體驗讓我相信：重構是一門非常重要的技術。但是，在重構的學習和推廣過程中我遇到了挫折，因爲我拿不出任何一本書給程式員看，也沒有任何一位專家打算寫出這樣一本書。所以，在這些專家的幫助下，我寫下了這本書。

最佳化一個薪資系統

— *Rich Garzaniti*

將 Chrysler Comprehensive Compensation（克萊斯勒綜合薪資系統）交給 GemStone 公司之前，我們用了相當長的時間開發它。開發過程中我們無可避免地發現程式不夠快，於是找了 Jim Haungs — GemSmith 中的一位好手 — 請他幫我們最佳化這個系統。

Jim 先用一點時間讓他的團隊了解系統運作方式，然後以 GemStone 的 ProfMonitor 特性編寫出一個效率量測工具，將它插入我們的功能測試中。這個工具可以顯示系統產生的物件數量，以及這些物件的誕生點。

令我們吃驚的是：創建量最大的物件竟是字串。其中最大的工作量則是反覆產生 12,000-bytes 的字串。這很特別，因爲這字串實在太大了，連 GemStone 慣用的垃圾回收設施都無法處理它。由於它是如此巨大，每當被創建出來，GemStone 都會將它分頁（paging）至磁碟上。也就是說字串的創建竟然用上了 I/O 子系統（譯註：分頁機制會動用 I/O），而每次輸出記錄時都要產生這樣的字串三次!

我們的第一個解決辦法是把一個 12,000-bytes 字串快取（cached）起來，這可解決一大半問題。後來我們又加以修改，將它直接寫入一個 file stream，從而避免產生字串。

解決了「巨大字串」問題後，Jim 的量測工具又發現了一些類似問題，只不過字串稍微小一些：800-bytes、500-bytes…等等，我們也都對它們改用 file stream，於是問題都解決了。

使用這些技術，我們穩步提高了系統效率。開發過程中原本似乎需要 1,000 小時以上才

能完成的薪資計算，實際運作時只花 40 小時。一個月後我們把時間縮短到 18 小時。正式投入運轉時只花 12 小時。經過一年的執行和改善後，全部計算只需 9 小時。

我們的最大改進就是：將程式放在多處理器（multi-processor）計算機上，以多緒（multiple threads）方式執行。最初這個系統並非按照多緒思維來設計，但由於程式碼有良好分解（well factored），所以我們只花三天時間就讓它得以同時執行多個執行緒了。現在，薪資的計算只需 2 小時。

在 Jim 提供工具使我們得以在實際操作中量度系統效率之前，我們也猜測過問題所在。但如果只靠猜測，我們需要很長的時間才能試出真正的解法。真實的量測指出了一個完全不同的方向，並大大加快了我們的進度。

3

程式碼的壞味道

Bad smells in Code, by Kent Beck and Martin Fowler

If it stinks, change it.（如果尿布臭了，就換掉它）
— 語出 *Beck* 奶奶，討論小孩撫養哲學

現在，對於「重構如何運作」，你已經有了相當好的理解。但是知道 How 不代表知道 When。決定何時重構、何時停止，和知道重構機制如何運轉是一樣重要的。

難題來了！解釋「如何刪除一個 *instance* 變數」或「如何產生一個 class hierarchy（階層體系）」很容易，因爲這些都是很簡單的事情。但要解釋「該在什麼時候做這些動作」就沒那麼順理成章了。除了露幾手含混的編程美學（說實話，這就是咱這些顧問常做的事），我還希望讓某些東西更具說服力一些。

去蘇黎士拜訪 Kent Beck 的時候，我正在爲這個微妙的問題大傷腦筋。也許是因爲受到剛出生的女兒的氣味影響吧，他提出「用味道來形容重構時機」。『味道』，他說，『聽起來是不是比含混的美學理論要好多了？』啊，是的。我們看過很多很多程式碼，它們所屬的專案從大獲成功到奄奄一息都有。觀察這些程式碼時，我們學會了從中找尋某些特定結構，這些結構指出（有時甚至就像尖叫呼喊）重構的可能性。（本章主詞換成「我們」，是爲了反映一個事實：Kent 和我共同撰寫本章。你應該可以看出我倆的文筆差異 — 插科打諢的部分是我寫的，其餘都是他的。）

我們並不試圖給你一個「重構爲時晚矣」的精確衡量標準。從我們的經驗看來，沒有任何量度規矩比得上一個見識廣博者的直覺。我們只會告訴你一些跡象，它會指出「這裡有一個可使用重構解決的問題」。你必須培養出自己的判斷力，學會判斷一個 class 內有多少 *instance* 變數算是太大、一個函式內有多少行程式碼才算太長。

如果你無法確定該進行哪一種重構手法，請閱讀本章內容和封底內頁表格來尋找靈感。你可以閱讀本章（或快速瀏覽封底內頁表格）來判斷自己聞到的是什麼味道，然後再看看我們所建議的重構手法能否幫助你。也許這裡所列的「臭味條款」和你所檢測的不盡相符，但願它們能夠爲你指引正確方向。

3.1 Duplicated Code（重複的程式碼）

臭味行列中首當其衝的就是 Duplicated Code。如果你在一個以上的地點看到相同的程式結構，那麼當可肯定：設法將它們合而爲一，程式會變得更好。

最單純的 Duplicated Code 就是「同一個 class 內的兩個函式含有相同算式（expression）」。這時候你需要做的就是採用 *Extract Method*（110）提煉出重複的程式碼，然後讓這兩個地點都呼叫被提煉出來的那一段程式碼。

另一種常見情況就是「兩個互爲兄弟（sibling）的 subclasses 內含相同算式」。要避免這種情況，只需對兩個 classes 都使用 *Extract Method*（110），然後再對被提煉出來的程式碼使用 *Pull Up Method*（332），將它推入 superclass 內。如果程式碼之間只是類似，並非完全相同，那麼就得運用 *Extract Method*（110）將相似部分和差異部分割開，構成單獨一個函式。然後你可能發現或許可以運用 *Form Template Method*（345）獲得一個 ***Template Method*** 設計範式。如果有些函式以不同的演算法做相同的事，你可以擇定其中較清晰的一個，並使用 *Substitute Algorithm*（139）將其他函式的演算法替換掉。

如果兩個毫不相關的 classes 內出現 Duplicated Code，你應該考慮對其中一個使用 *Extract Class*（149），將重複程式碼提煉到一個獨立 class 中，然後在另一個 class 內使用這個新 class。但是，重複程式碼所在的函式也可能的確只應該屬於某個 class，另一個 class 只能呼叫它，抑或這個函式可能屬於第三個 class，而另兩個 classes 應該引用這第三個 class。你必須決定這個函式放在哪兒最合適，並確保它被安置後就不會再在其他任何地方出現。

3.2 Long Method（過長函式）

擁有「短函式」（short methods）的物件會活得比較好、比較長。不熟悉物件導向技術的人，常常覺得物件程式中只有無窮無盡的 *delegation*（委託），根本沒有進行任何計算。和此類程式共同生活數年之後，你才會知道，這些小小函式有多大價值。「間接層」所能帶來的全部利益 — 解釋能力、共享能力、選擇能力 — 都

是由小型函式支援的（請看 p.61 的「間接層和重構」）。

很久以前程式員就已認識到：程式愈長愈難理解。早期的編程語言中，「子程式叫用動作」需要額外開銷，這使得人們不太樂意使用 small method。現代 OO 語言幾乎已經完全免除了行程（process）內的「函式叫用動作額外開銷」。不過程式碼閱讀者還是得多費力氣，因爲他必須經常轉換上下文去看看子程式做了什麼。某些開發環境允許使用者同時看到兩個函式，這可以幫助你省去部分麻煩，但是讓 small method 容易理解的真正關鍵在於一個好名字。如果你能給函式起個好名字，讀者就可以通過名字了解函式的作用，根本不必去看其中寫了些什麼。

最終的效果是：你應該更積極進取地分解函式。我們遵循這樣一條原則：每當感覺需要以註釋來說明點什麼的時候，我們就把需要說明的東西寫進一個獨立函式中，並以其用途（而非實現手法）命名。我們可以對一組或甚至短短一行程式碼做這件事。哪怕替換後的函式呼叫動作比函式本身還長，只要函式名稱能夠解釋其用途，我們也該毫不猶豫地那麼做。關鍵不在於函式的長度，而在於函式「做什麼」和「如何做」之間的語意距離。

百分之九十九的場合裡，要把函式變小，只需使用 *Extract Method*（110）。找到函式中適合集在一起的部分，將它們提煉出來形成一個新函式。

如果函式內有大量的參數和暫時變數，它們會對你的函式提煉形成阻礙。如果你嘗試運用 *Extract Method*（110），最終就會把許多這些參數和暫時變數當做參數，傳遞給被提煉出來的新函式，導致可讀性幾乎沒有任何提昇。啊是的，你可以經常運用 *Replace Temp with Query*（120）來消除這些暫時元素。*Introduce Parameter Object*（295）和 *Preserve Whole Object*（288）則可以將過長的參數列變得更簡潔一些。

如果你已經這麼做了，仍然有太多暫時變數和參數，那就應該使出我們的殺手鐧：*Replace Method with Method Object*（135）。

如何確定該提煉哪一段程式碼呢？一個很好的技巧是：尋找註釋。它們通常是指出「程式碼用途和實現手法間的語意距離」的信號。如果程式碼前方有一行註釋，就是在提醒你：可以將這段程式碼替換成一個函式，而且可以在註釋的基礎上給這個函式命名。就算只有一行程式碼，如果它需要以註釋來說明，那也值得將它提煉到獨立函式去。

條件式和迴圈常常也是提煉的信號。你可以使用 *Decompose Conditional*（238）處理條件式。至於迴圈，你應該將迴圈和其內的程式碼提煉到一個獨立函式中。

3.3 Large Class（過大類別）

如果想利用單一 class 做太多事情，其內往往就會出現太多 *instance* 變數。一旦如此，Duplicated Code 也就接踵而至了。

你可以運用 *Extract Class*（149）將數個變數一起提煉至新 class 內。提煉時應該選擇 class 內彼此相關的變數，將它們放在一起。例如 `"depositAmount"` 和 `"depositCurrency"` 可能應該隸屬同一個 class。通常如果 class 內的數個變數有著相同的字首或字尾，這就意味有機會把它們提煉到某個組件內。如果這個組件適合作爲一個 subclass，你會發現 *Extract Subclass*（330）往往比較簡單。

有時候 class 並非在所有時刻都使用所有 *instance* 變數。果真如此，你或許可以多次使用 *Extract Class*（149）或 *Extract Subclass*（330）。

和「太多 *instance* 變數」一樣，class 內如果有太多程式碼，也是「程式碼重複、混亂、死亡」的絕佳滋生地點。最簡單的解決方案（還記得嗎，我們喜歡簡單的解決方案）是把贅餘的東西消弭於 class 內部。如果有五個「百行函式」，它們之中很多程式碼都相同，那麼或許你可以把它們變成五個「十行函式」和十個提煉出來的「雙行函式」。

和「擁有太多 *instance* 變數」一樣，一個 class 如果擁有太多程式碼，往往也適合使用 *Extract Class*（149）和 *Extract Subclass*（330）。這裡有個有用技巧：先確定客戶端如何使用它們，然後運用 *Extract Interface*（341）爲每一種使用方式提煉出一個介面。這或許可以幫助你看清楚如何分解這個 class。

如果你的 Large Class 是個 GUI class，你可能需要把資料和行爲移到一個獨立的領域物件（domain object）去。你可能需要兩邊各保留一些重複資料，並令這些資料同步（sync.）。*Duplicate Observed Data*（189）告訴你該怎麼做。這種情況下，特別是如果你使用舊式 Abstract Windows Toolkit（AWT）組件，你可以採用這種方式去掉 GUI class 並代以 Swing 組件。

3.4 Long Parameter List（過長參數列）

剛開始學習編程的時候，老師教我們：把函式所需的所有東西都以參數傳遞進去。這可以理解，因爲除此之外就只能選擇全域資料，而全域資料是邪惡的東西。物件技術改變了這一情況，因爲如果你手上沒有你所需要的東西，總可以叫另一個

物件給你。因此，有了物件，你就不必把函式需要的所有東西都以參數傳遞給它了，你只需傳給它足夠的東西、讓函式能從中獲得自己需要的所有東西就行了。函式需要的東西多半可以在函式的宿主類別（host class）中找到。物件導向程式中的函式，其參數列通常比在傳統程式中短得多。

這是好現象，因爲太長的參數列難以理解，太多參數會造成前後不一致、不易使用，而且一旦你需要更多資料，就不得不修改它。如果將物件傳遞給函式，大多數修改都將沒有必要，因爲你很可能只需（在函式內）增加一兩條請求（requests），就能得到更多資料。

如果「向既有物件發出一條請求」就可以取得原本位於參數列上的一份資料，那麼你應該啓動重構準則 *Replace Parameter with Method*（292）。上述的既有物件可能是函式所屬 class 內的一個資料欄（field），也可能是另一個參數。你還可以運用 *Preserve Whole Object*（288）將來自同一物件的一堆資料收集起來，並以該物件替換它們。如果某些資料缺乏合理的物件歸屬，可使用 *Introduce Parameter Object*（295）爲它們製造出一個「參數物件」。

此間存在一個重要的例外。有時候你明顯不希望造成「被叫用之物件」與「較大物件」間的某種依存關係。這時候將資料從物件中拆解出來單獨作爲參數，也很合情合理。但是請注意其所引發的代價。如果參數列太長或變化太頻繁，你就需要重新考慮自己的依存結構（dependency structure）了。

3.5 Divergent Change（發散式變化）

我們希望軟體能夠更容易被修改 — 畢竟軟體再怎麼說本來就該是「軟」的。一旦需要修改，我們希望能夠跳到系統的某一點，只在該處作修改。如果不能做到這點，你就嗅出兩種緊密相關的刺鼻味道中的一種了。

如果某個 class 經常因爲不同的原因在不同的方向上發生變化，Divergent Change 就出現了。當你看著一個 class 說：『呃，如果新加入一個資料庫，我必須修改這三個函式；如果新出現一種金融工具，我必須修改這四個函式』，那麼此時也許將這個物件分成兩個會更好，這麼一來每個物件就可以只因一種變化而需要修改。當然，往往只有在加入新資料庫或新金融工具後，你才能發現這一點。針對某一外界變化的所有相應修改，都只應該發生在單一 class 中，而這個新 class 內的所有內容都應該反應該外界變化。爲此，你應該找出因著某特定原因而造成的所有變化，然後運用 *Extract Class*（149）將它們提煉到另一個 class 中。

3.6 Shotgun Surgery（霰彈式修改）

Shotgun Surgery 類似 Divergent Change，但恰恰相反。如果每遇到某種變化，你都必須在許多不同的 classes 內作出許多小修改以回應之，你所面臨的壞味道就是 Shotgun Surgery。如果需要修改的程式碼散佈四處，你不但很難找到它們，也很容易忘記某個重要的修改。

這種情況下你應該使用 *Move Method*（142）和 *Move Field*（146）把所有需要修改的程式碼放進同一個 class。如果眼下沒有合適的 class 可以安置這些程式碼，就創造一個。通常你可以運用 *Inline Class*（154）把一系列相關行爲放進同一個 class。這可能會造成少量 Divergent Change，但你可以輕易處理它。

Divergent Change 是指「一個 class 受多種變化的影響」，Shotgun Surgery 則是指「一種變化引發多個 classes 相應修改」。這兩種情況下你都會希望整理程式碼，取得「外界變化」與「待改類別」呈現一對一關係的理想境地。

3.7 Feature Envy（依戀情結）

物件技術的全部要點在於：這是一種「將資料和加諸其上的操作行爲包裝在一起」的技術。有一種經典氣味是：函式對某個 class 的興趣高過對自己所處之 host class 的興趣。這種孺慕之情最通常的焦點便是資料。無數次經驗裡，我們看到某個函式爲了計算某值，從另一個物件那兒呼叫幾乎半打的取值函式（getting method）。療法顯而易見：把這個函式移至另一個地點。你應該使用 *Move Method*（142）把它移到它該去的地方。有時候函式中只有一部分受這種依戀之苦，這時候你應該使用 *Extract Method*（110）把這一部分提煉到獨立函式中，再使用 *Move Method*（142）帶它去它的夢中家園。

當然，並非所有情況都這麼簡單。一個函式往往會用上數個 classes 特性，那麼它究竟該被置於何處呢？我們的原則是：判斷哪個 class 擁有最多「被此函式使用」的資料，然後就把這個函式和那些資料擺在一起兒。如果先以 *Extract Method*（110）將這個函式分解爲數個較小函式並分別置放於不同地點，上述步驟也就比較容易完成了。

有數個複雜精巧的範式（patterns）破壞了這個規則。說起這個話題，「四巨頭」[Gang of Four] 的 ***Strategy*** 和 ***Visitor*** 立刻跳入我的腦海，Kent Beck 的 ***Self Delegation*** [Beck] 也在此列。使用這些範式是爲了對抗壞味道 Divergent Change。最根本的原則是：將總是一起變化的東西放在一塊兒。「資料」和「引用這些資料」的行

爲總是一起變化的，但也有例外。如果例外出現，我們就搬移那些行爲，保持「變化只在一地發生」。*Strategy* 和 *Visitor* 使你得以輕鬆修改函式行爲，因爲它們將少量需被覆寫（overridden）的行爲隔離開來 — 當然也付出了「多一層間接性」的代價。

3.8 Data Clumps（資料泥團）

資料項（data items）就像小孩子：喜歡成群結隊地待在一塊兒。你常常可以在很多地方看到相同的三或四筆資料項：兩個 classes 內的相同欄位（field）、許多函式署名式（signature）中的相同參數。這些「總是綁在一起出現的資料」真應該放進屬於它們自己的物件中。首先請找出這些資料的欄位形式（field）出現點，運用 *Extract Class*（149）將它們提煉到一個獨立物件中。然後將注意力轉移到函式署名式（signature）上頭，運用 *Introduce Parameter Object*（295）或 *Preserve Whole Object*（288）爲它減肥。這麼做的直接好處是可以將很多參數列縮短，簡化函式呼叫動作。是的，不必因爲 Data Clumps 只用上新物件的一部分欄位而在意，只要你以新物件取代兩個（或更多）欄位，你就值回票價了。

一個好的評斷辦法是：刪掉眾多資料中的一筆。其他資料有沒有因而失去意義？如果它們不再有意義，這就是個明確信號：你應該爲它們產生一個新物件。

縮短欄位個數和參數個數，當然可以去除一些壞味道，但更重要的是：一旦擁有新物件，你就有機會讓程式散發出一種芳香。得到新物件後，你就可以著手尋找 Feature Envy，這可以幫你指出「可移至新 class」中的種種程式行爲。不必太久，所有 classes 都將在它們的小小社會中充分發揮自己的生產力。

3.9 Primitive Obsession（基本型別偏執）

大多數編程環境都有兩種資料：結構型別（record types）允許你將資料組織成有意義的形式；基本型別（primitive types）則是構成結構型別的積木塊。結構總是會帶來一定的額外開銷。它們有點像資料庫中的表格，或是那些得不償失（只爲做一兩件事而創建，卻付出太大額外開銷）的東西。

物件的一個極具價值的東西是：它們模糊（甚至打破）了橫亙於基本資料和體積較大的 classes 之間的界限。你可以輕鬆編寫出一些與語言內建（基本）型別無異的小型 classes。例如 Java 就以基本型別表示數值，而以 class 表示字串和日期 — 這兩個型別在其他許多編程環境中都以基本型別表現。

物件技術的新手通常不願意在小任務上運用小物件 — 像是結合數值和幣別的 money class、含一個起始值和一個結束值的 range class、電話號碼或郵遞區號（ZIP）等等的特殊 strings。你可以運用 *Replace Data Value with Object*（175）將原本單獨存在的資料值替換爲物件，從而走出傳統的洞窟，進入炙手可熱的物件世界。如果欲替換之資料值是 type code（型別代碼），而它並不影響行爲，你可以運用 *Replace Type Code with Class*（218）將它換掉。如果你有相依於此 *type code* 的條件式，可運用 *Replace Type Code with Subclass*（213）或 *Replace Type Code with State/Strategy*（227）加以處理。

如果你有一組應該總是被放在一起的欄位（fields），可運用 *Extract Class*（149）。如果你在參數列中看到基本型資料，不妨試試 *Introduce Parameter Object*（295）。如果你發現自己正從 array 中挑選資料，可運用 *Replace Array with Object*（186）。

3.10 Switch Statements（`switch` 驚悚現身）

物件導向程式的一個最明顯特徵就是：少用 `switch`（或 `case`）述句。從本質上說，`switch` 述句的問題在於重複（duplication）。你常會發現同樣的 `switch` 述句散佈於不同地點。如果要爲它添加一個新的 `case` 子句，你必須找到所有 `switch` 述句並修改它們。物件導向中的多型（polymorphism）概念可爲此帶來優雅的解決辦法。

大多數時候，一看到 `switch` 述句你就應該考慮以「多型」來替換它。問題是多型該出現在哪兒？`switch` 述句常常根據 type code（型別代碼）進行選擇，你要的是「與該 type code 相關的函式或 class」。所以你應該使用 *Extract Method*（110）將 `switch` 述句提煉到一個獨立函式中，再以 *Move Method*（142）將它搬移到需要多型性的那個 class 裡頭。此時你必須決定是否使用 *Replace Type Code with Subclasses*（223）或 *Replace Type Code with State/Strategy*（227）。一旦這樣完成繼承結構之後，你就可以運用 *Replace Conditional with Polymorphism*（255）了。

如果你只是在單一函式中有些選擇事例，而你並不想改動它們，那麼「多型」就有點殺雞用牛刀了。這種情況下 *Replace Parameter with Explicit Methods*（285）是個不錯的選擇。如果你的選擇條件之一是 null，可以試試 *Introduce Null Object*（260）。

3.11 Parallel Inheritance Hierarchies（平行繼承體系）

Parallel Inheritance Hierarchies 其實是 Shotgun Surgery 的特殊情況。在這種情況下，每當你爲某個 class 增加一個 subclass，必須也爲另一個 class 相應增加一個 subclass。如果你發現某個繼承體系的 class 名稱字首和另一個繼承體系的 class 名稱字首完全相同，便是聞到了這種壞味道。

消除這種重複性的一般策略是：讓一個繼承體系的實體（instances）指涉（參考、引用、*refer to*）另一個繼承體系的實體（instances）。如果再接再厲運用 *Move Method*（142）和 *Move Field*（146），就可以將指涉端（referring class）的繼承體系消弭於無形。

3.12 Lazy Class（冗員類別）

你所創建的每一個 class，都得有人去理解它、維護它，這些工作都是要花錢的。如果一個 class 的所得不值其身價，它就應該消失。專案中經常會出現這樣的情況：某個 class 原本對得起自己的身價，但重構使它身形縮水，不再做那麼多工作；或開發者事前規劃了某些變化，並添加一個 class 來應付這些變化，但變化實際上沒有發生。不論上述哪一種原因，請讓這個 class 莊嚴赴義吧。如果某些 subclass 沒有做滿足夠工作，試試 *Collapse Hierarchy*（344）。對於幾乎沒用的組件，你應該以 *Inline Class*（154）對付它們。

3.13 Speculative Generality（夸夸其談未來性）

這個令我們十分敏感的壞味道，命名者是 Brian Foote。當有人說『噢，我想我們總有一天需要做這事』並因而企圖以各式各樣的掛勾（hooks）和特殊情況來處理一些非必要的事情，這種壞味道就出現了。那麼做的結果往往造成系統更難理解和維護。如果所有裝置都會被用到，那就值得那麼做；如果用不到，就不值得。用不上的裝置只會擋你的路，所以，把它搬開吧。

如果你的某個 abstract class 其實沒有太大作用，請運用 *Collapse Hierarchy*（344）。非必要之 delegation（委託）可運用 *Inline Class*（154）除掉。如果函式的某些參數未被用上，可對它實施 *Remove Parameter*（277）。如果函式名稱帶有多餘的抽象意味，應該對它實施 *Rename Method*（273）讓它現實一些。

如果函式或 class 的惟一使用者是 test cases（測試案例），這就飄出了壞味道 Speculative Generality。如果你發現這樣的函式或 class，請把它們連同其 *test cases*

都刪掉。但如果它們的用途是幫助 *test cases* 檢測正當功能，當然必須刀下留人。

3.14 Temporary Field（令人迷惑的暫時欄位）

有時你會看到這樣的物件：其內某個 *instance* 變數僅爲某種特定情勢而設。這樣的程式碼讓人不易理解，因爲你通常認爲物件在所有時候都需要它的所有變數。在變數未被使用的情況下猜測當初其設置目的，會讓你發瘋。

請使用 *Extract Class*（149）給這個可憐的孤兒創造一個家，然後把所有和這個變數相關的程式碼都放進這個新家。也許你還可以使用 *Introduce Null Object*（260）在「變數不合法」的情況下創建一個 Null 物件，從而避免寫出「條件式程式碼」。

如果 class 中有一個複雜演算法，需要好幾個變數，往往就可能導致壞味道 Temporary Field 的出現。由於實作者不希望傳遞一長串參數（想想爲什麼），所以他把這些參數都放進欄位（fields）中。但是這些欄位只在使用該演算法時才有效，其他情況下只會讓人迷惑。這時候你可以利用 *Extract Class*（149）把這些變數和其相關函式提煉到一個獨立 class 中。提煉後的新物件將是一個 method object [Beck]（譯註：其存在只是爲了提供呼叫函式的途徑，class 本身並無抽象意味）。

3.15 Message Chains（過度耦合的訊息鏈）

如果你看到用戶向一個物件索求（*request*）另一個物件，然後再向後者索求另一個物件，然後再索求另一個物件…這就是 Message Chain。實際程式碼中你看到的可能是一長串 `getThis()`或一長串暫時變數。採行這種方式，意味客戶將與搜尋過程中的航行結構（structure of navigation）緊密耦合。一旦物件間的關係發生任何變化，客戶端就不得不作出相應修改。

這時候你應該使用 *Hide Delegate*（157）。你可以在 Message Chain 的不同位置進行這種重構手法。理論上你可以重構 Message Chain 上的任何一個物件，但這麼做往往會把所有中介物件（intermediate object）都變成 Middle Man。通常更好的選擇是：先觀察 Message Chain 最終得到的物件是用來幹什麼的，看看能否以 *Extract Method*（110）把使用該物件的程式碼提煉到一個獨立函式中，再運用 *Move Method*（142）把這個函式推入 Message Chain。如果這條鏈上的某個物件有多位客戶打算航行此航線的剩餘部分，就加一個函式來做這件事。

有些人把任何函式鏈（method chain。譯註：就是 Message Chain；物件導向領域中所謂「發送訊息」就是「喚起函式」）都視爲壞東西，我們不這樣想。呵呵，我們的冷靜鎮定是出了名的，起碼在這件事情上是這樣。

3.16 Middle Man（中間轉手人）

物件的基本特徵之一就是封裝（encapsulation）— 對外部世界隱藏其內部細節。封裝往往伴隨 delegation（委託）。比如說你問主管是否有時間參加一個會議，他就把這個訊息委託給他的記事簿，然後才能回答你。很好，你沒必要知道這位主管到底使用傳統記事簿或電子記事簿抑或秘書來記錄自己的約會。

但是人們可能過度運用 delegation。你也許會看到某個 class 介面有一半的函式都委託給其他 class，這樣就是過度運用。這時你應該使用 *Remove Middle Man*（160），直接和實責物件打交道。如果這樣「不幹實事」的函式只有少數幾個，可以運用 *Inline Method*（117）把它們 "inlining" 放進呼叫端。如果這些 Middle Man 還有其他行爲，你可以運用 *Replace Delegation with Inheritance*（355）把它變成實責物件的 subclass，這樣你既可以擴展原物件的行爲，又不必負擔那麼多的委託動作。

3.17 Inappropriate Intimacy（狎暱關係）

有時你會看到兩個 classes 過於親密，花費太多時間去探究彼此的 private 成分。如果這發生在兩個「人」之間，我們不必做衛道之士；但對於 classes，我們希望它們嚴守清規。

就像古代戀人一樣，過份狎暱的 classes 必須拆散。你可以採用 *Move Method*（142）和 *Move Field*（146）幫它們劃清界線，從而減少狎暱行徑。你也可以看看是否運用 *Change Bidirectional Association to Unidirectional*（200）讓其中一個 class 對另一個斬斷情思。如果兩個 classes 實在是情投意合，可以運用 *Extract Class*（149）把兩者共同點提煉到一個安全地點，讓它們坦蕩地使用這個新 class。或者也可以嘗試運用 *Hide Delegate*（157）讓另一個 class 來爲它們傳遞相思情。

繼承（inheritance）往往造成過度親密，因爲 subclass 對 superclass 的了解總是超過 superclass 的主觀願望。如果你覺得該讓這個孩子獨自生活了，請運用 *Replace Inheritance with Delegation*（352）讓它離開繼承體系。

3.18 Alternative Classes with Different Interfaces（異曲同工的類別）

如果兩個函式做同一件事，卻有著不同的署名式（signature），請運用 *Rename Method*

（273）根據它們的用途重新命名。但這往往不夠，請反復運用 *Move Method*（142）將某些行爲移入 classes，直到兩者的協定（protocols）一致爲止。如果你必須重複而贅餘地移入程式碼才能完成這些，或許可運用 *Extract Superclass*（336）爲自己贖點罪。

3.19 Incomplete Library Class（不完美的程式庫類別）

復用（reuse）常被視爲物件的終極目的。我們認爲這實在是過度估計了（我們只是使用而已）。但是無可否認，許多編程技術都建立在 library classes（程式庫類別）的基礎上，沒人敢說是不是我們都把排序演算法忘得一乾二淨了。

library classes 構築者沒有未卜先知的能力，我們不能因此責怪他們。畢竟我們自己也幾乎總是在系統快要構築完成的時候才能弄清楚它的設計，所以 library 構築者的任務真的很艱鉅。麻煩的是 library 的形式（form）往往不夠好，往往不可能讓我們修改其中的 classes 使它完成我們希望完成的工作。這是否意味那些經過實踐檢驗的戰術如 *Move Method*（142）等等，如今都派不上用場了？

幸好我們有兩個專門應付這種情況的工具。如果你只想修改 library classes 內的一兩個函式，可以運用 *Introduce Foreign Method*（162）；如果想要添加一大堆額外行爲，就得運用 *Introduce Local Extension*（164）。

3.20 Data Class（純稚的資料類別）

所謂 Data Class 是指：它們擁有一些欄位（fields），以及用於存取（讀寫）這些欄位的函式，除此之外一無長物。這樣的 classes 只是一種「不會說話的資料容器」，它們幾乎一定被其他 classes 過份細瑣地操控著。這些 classes 早期可能擁有 public 欄位，果真如此你應該在別人注意到它們之前，立刻運用 *Encapsulate Field*（206）將它們封裝起來。如果這些 classes 內含容器類的欄位（collection fields），你應該檢查它們是不是得到了恰當的封裝；如果沒有，就運用 *Encapsulate Collection*（208）把它們封裝起來。對於那些不該被其他 classes 修改的欄位，請運用 *Remove Setting Method*（300）。

然後，找出這些「取值/設值」函式（getting and setting methods）被其他 classes 運用的地點。嘗試以 *Move Method*（142）把那些呼叫行爲搬移到 Data Class 來。如果無法搬移整個函式，就運用 *Extract Method*（110）產生一個可被搬移的函式。不久之後你就可以運用 *Hide Method*（303）把這些「取值/設值」函式隱藏起來了。

Data Class 就像小孩子。作爲一個起點很好，但若要讓它們像「成年（成熟）」的物件那樣參與整個系統的工作，它們就必須承擔一定責任。

3.21 Refused Bequest（被拒絕的遺贈）

subclasses 應該繼承 superclass 的函式和資料。但如果它們不想或不需要繼承，又該怎麼辦呢？它們得到所有禮物，卻只從中挑選幾樣來玩！

按傳統說法，這就意味繼承體系設計錯誤。你需要爲這個 subclass 新建一個兄弟（sibling class），再運用 *Push Down Method*（328）和 *Push Down Field*（329）把所有用不到的函式下推給那兄弟。這樣一來 superclass 就只持有所有 subclasses 共享的東西。常常你會聽到這樣的建議：所有 superclasses 都應該是抽象的（abstract）。

既然使用「傳統說法」這個略帶貶義的詞，你就可以猜到，我們不建議你這麼做，起碼不建議你每次都這麼做。我們經常利用 subclassing 手法來復用一些行爲，並發現這可以很好地應用於日常工作。這也是一種壞味道，我們不否認，但氣味通常並不強烈。所以我們說：如果 Refused Bequest 引起困惑和問題，請遵循傳統忠告。但不必認爲你每次都得那麼做。十有八九這種壞味道很淡，不值得理睬。

如果 subclass 復用了 superclass 的行爲（實作），卻又不願意支援 superclass 的介面，Refused Bequest 的壞味道就會變得濃烈。拒絕繼承 superclass 的實作，這一點我們不介意；但如果拒絕繼承 superclass 的介面，我們不以爲然。不過即使你不願意繼承介面，也不要胡亂修改繼承體系，你應該運用 *Replace Inheritance with Delegation*（352）來達到目的。

3.22 Comments（過多的註釋）

別擔心，我們並不是說你不該寫註釋。從嗅覺上說，Comments 不是一種壞味道；事實上它們還是一種香味呢。我們之所以要在這裡提到 Comments，因爲人們常把它當作除臭劑來使用。常常會有這樣的情況：你看到一段程式碼有著長長的註釋，然後發現，這些註釋之所以存在乃是因爲程式碼很糟糕。這種情況的發生次數之多，實在令人吃驚。

Comments 可以帶我們找到本章先前提到的各種壞味道。找到壞味道後，我們首先應該以各種重構手法把壞味道去除。完成之後我們常常會發現：註釋已經變得多餘了，因爲程式碼已經清楚說明了一切。

如果你需要註釋來解釋一塊程式碼做了什麼，試試 *Extract Method*（110）；如果 method 已經提煉出來，但還是需要註釋來解釋其行爲，試試 *Rename Method*（273）；如果你需要註釋說明某些系統的需求規格，試試 *Introduce Assertion*（267）。

當你感覺需要撰寫註釋，請先嘗試重構，試著讓所有註釋都變得多餘。

如果你不知道該做什麼，這才是註釋的良好運用時機。除了用來記述將來的打算之外，註釋還可以用來標記你並無十足把握的區域。你可以在註釋裡寫下自己「爲什麼做某某事」。這類資訊可以幫助將來的修改者，尤其是那些健忘的傢伙。

4

構築測試體系

Building Test

如果你想進行重構（refactoring），首要前提就是擁有一個可靠的測試環境。就算你夠幸運，有一個可以自動進行重構的工具，你還是需要測試。而且短時間內不可能有任何工具可以爲我們自動進行所有可能的重構。

我並不把這視爲缺點。我發現，編寫優良的測試程式，可以極大提高我的編程速度，即使不進行重構也一樣如此。這讓我很吃驚，也違反許多程式員的直覺，所以我有必要解釋一下這個現象。

4.1 自我測試碼（Self-testing Code）的價值

如果認真觀察程式員把最多時間耗在哪裡，你就會發現，編寫程式其實只佔非常小的一部分。有些時間用來決定下一步幹什麼，另一些時間花在設計上面，最多的時間則是用來除錯（debug）。我敢肯定每一位讀者都還記得自己花在除錯上面的無數個小時，無數次通宵達旦。每個程式員都能講出「花一整天（甚至更多）時間只找出一隻小小臭蟲」的故事。修復錯誤通常是比較快的，但找出錯誤卻是噩夢一場。當你修好一個錯誤，總是會有另一個錯誤出現，而且肯定要很久以後才會注意到它。彼時你又要花上大把時間去尋找它。

我走上「自我測試碼」這條路，肇因於 1992 年 OOPSLA 大會上的一次演講。會場上有人（我記得好像是 Dave Thomas）說：『class 應該包含它們自己的測試碼。』這激發了我的靈感，讓我想到一種組織測試的好方法。我這樣解釋它：每個 class 都應該有一個測試函式，並以它來測試自己這個 class。

那時候我還著迷於增量式開發（incremental development），所以我嘗試在結束每次增量時，爲每個 class 添加測試。當時我開發的案子很小，所以我們大約每週增量一次。執行測試變得相當直率，但儘管如此，做這些測試還是很煩人，因爲每

個測試都把結果輸出到主控台（console），而我必須逐一檢查它們。我是個很懶的人，我情願當下努力工作以免除日後的工作。我意識到我其實完全不必自己盯著螢幕檢驗測試所得資訊是否正確，我大可讓電腦來幫我做這件事。我需要做的就是把我所期望的輸出放進測試碼中，然後做一個比較就行了。於是我可以舒服地執行每個 class 的測試函式，如果一切都沒問題，螢幕上就只出現一個 "OK"。現在，這些 classes 都變成「自我測試」了。

確保所有測試都完全自動化，讓它們檢查自己的測試結果。

此後再進行測試就簡單多了，和編譯一樣簡單。於是我開始在每次編譯之後都進行測試。很快我發現自己的生產效率大大提高。我意識到那是因爲我沒有花太多時間去除錯。如果我不小心引入一個可被原測試捕捉到的錯誤，那麼只要我執行測試，它就會向我報告這個錯誤。由於測試本來是可以正常執行的，所以我知道這個錯誤必定是在前一次執行測試後引入。由於我頻繁地進行測試，每次測試都在不久之前，因此我知道錯誤的源頭就是我剛剛寫下的程式碼。而由於我對那段程式碼記憶猶新，份量也很小，所以輕鬆就能找到錯誤。從前需要一小時甚至更多時間才能找到的錯誤，現在最多只需兩分鐘就找到了。之所以能夠擁有如此強大的偵錯能力，不僅僅因爲我構築了 self-testing classes（自我測試類別），也因爲我頻繁地執行它們。

注意到這一點後，我對測試的積極性更高了。我不再等待每次增量結束，只要寫好一點功能，我就立即添加測試。每天我都會添加一些新功能，同時也添加相應的測試。那些日子裡，我很少花一分鐘以上的時間在除錯上面。

一整組（a suite of）測試就是一個強大的臭蟲偵測器，能夠大大縮減搜尋臭蟲所需要的時間。

當然，說服別人也這麼做，並不容易。編寫測試程式，意味要寫很多額外程式碼。除非你確切體驗到這種方法對編程速度的提昇，否則自我測試就顯不出它的意義。很多人根本沒學過如何編寫測試程式，甚至根本沒考慮過測試，這對於編寫自我測試碼也很不利。如果需要手動執行測試，那更是令人煩悶欲嘔；但如果可以自動執行，編寫測試碼就真的很有趣。

實際上，撰寫測試碼的最有用時機是在開始編程之前。當你需要添加特性的時候，先寫相應測試碼。聽起來離經叛道，其實不然。編寫測試碼其實就是在問自己：添加這個功能需要做些什麼。編寫測試碼還能使你把注意力集中於介面而非實作上頭（這永遠是件好事）。預先寫好的測試碼也爲你的工作安上一個明確的結束標誌：一旦測試碼正常執行，工作就可以結束了。

「頻繁進行測試」是極限編程（eXtreme Programming，XP） [Beck, XP] 的重要一環。「極限編程」一詞容易讓人聯想起那些編碼飛快、自由而散漫的駭客（hackers），實際上極限編程者都是十分專注的測試者。他們希望儘可能快速開發軟體，而他們也知道「測試」可協助他們儘可能快速地前進。

爭論至此可休矣。儘管我相信每個人都可以從編寫自我測試碼中受益，但這並不是本書重點。本書談的是重構，而重構需要測試。如果你想重構，你就必須編寫測試碼。本章將教你「以 Java 編寫測試碼」的起步知識。這不是一本專講測試的書，所以我不想講得太仔細。但我發現，少量測試就足以帶來驚人的利益。

和本書其他內容一樣，我以實例來介紹測試手法。開發軟體的時候，我一邊撰寫程式碼，一邊撰寫測試碼。但是當我和他人並肩重構時，往往得面對許多無自我測試的程式碼。所以重構之前我們首先必須把這些程式碼改造爲「自我測試」。

Java 之中的測試慣用手法是 "testing main"，意思是每個 class 都應該有一個用於測試的 `main()`。這是一個合理的習慣（儘管並不那麼值得稱許），但可能不好操控。這種作法的問題是很難輕鬆執行多個測試。另一種作法是：建立一個獨立 class 用於測試，並在一個框架（framework）中執行它，使測試工作更輕鬆。

4.2 JUnit 測試框架（Testing Framework）

譯註：本段將使用英文詞：test-suite（測試套件）、test-case（測試案例）和 test-fixture（測試裝備），期能直接對應圖 4.1 的 JUnit 結構組件，並有助於閱讀 JUnit 文件。

我用的是 JUnit，一個由 Erich Gamma 和 Kent Beck [JUnit] 開發的源碼開放測試框架。這個框架非常簡單，卻可讓你進行測試所需的所有重要事情。本章中我將運用這個測試框架來爲一些 IO classes 開發測試碼。

首先我創建一個 `FileReaderTester` class 來測試檔案讀取器。任何「包含測試碼」的 class 都必須衍生自測試框架所提供的 `TestCase` class。這個框架運用 ***Composite***

範式 [Gang of Four]，允許你將測試碼聚集到 suites（套件）中，如圖 4.1。這些套件可以包含未加工的 test-cases（測試案例），或其他 test-suits（測試套件）。如此一來我就可以輕鬆地將一系列龐大的 test-suits 結合在一起，並自動執行它們。

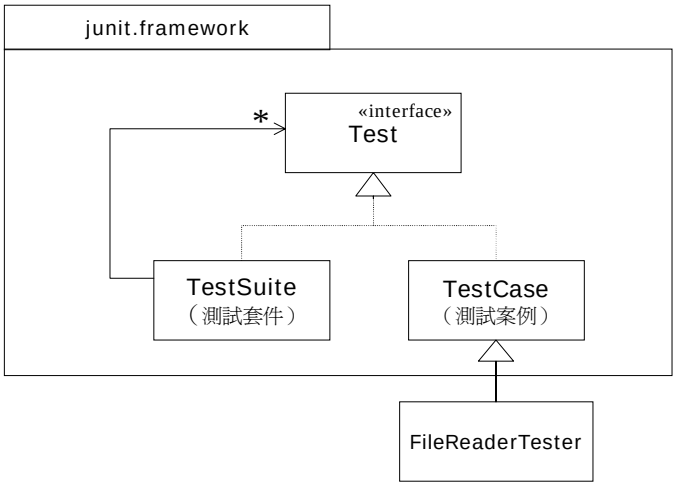


圖 4.1 測試框架的 *Composite* 結構

```
class FileReaderTester extends TestCase {
    public FileReaderTester (String name) {
        super(name);
    }
}
```

這個新建的 class 必須有一個建構式。完成之後我就可以開始添加測試碼了。我的第一件工作是設置 test fixture（測試裝備），那是指「用於測試的物件樣本」。由於我要讀一個檔案，所以先備妥一個測試檔如下：

```
Bradman      99.94    52   80   10   6996     334      29
Pollock      60.97    23   41   4    2256     274      7
Headley      60.83    22   40   4    2190     270*     10
Sutcliffe    60.73    54   84   9    4555     194      16
```

進一步運用這個檔案之前，我得先準備好 test fixture（測試裝備）。`TestCase` class 提供兩個函式專門針對此一用途：`setUp()`用來產生相關物件、`tearDown()`負責刪除它們。在 `TestCase` class 中這兩個函式都只有空殼。大多數時候你不需要操心 test fixture 的拆除（垃圾回收器會扛起責任），但是在這裡，以 `tearDown()` 關閉檔案無疑是明智之舉：

```
class FileReaderTester...
    protected void setUp() {
        try {
            _input = new FileReader("data.txt");
        } catch (FileNotFoundException e) {
            throw new RuntimeException ("unable to open test file");
        }
    }

    protected void tearDown() {
        try {
            _input.close();
        } catch (IOException e) {
            throw new RuntimeException ("error on closing test file");
        }
    }
```

現在我有了適當的 test fixture（測試裝備），可以開始編寫測試碼了。首先要測試的是 `read()`，我要讀取一些字元，然後檢查後續讀取的字元是否正確：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('d' == ch);
}
```

`assert()`扮演自動測試角色。如果 `assert()`的參數值爲 true，一切良好；否則我們就會收到錯誤通知。稍後我會讓你看看測試框架怎麼向使用者報告錯誤訊息。現在我要先介紹如何將測試過程執行起來。

第一步是產生一個 test suite（測試套件）。爲達目的，請設計一個 `suite()`如下：

```
class FileReaderTester...
    public static Test suite() {
        TestSuite suite= new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite;
    }
```

這個測試套件只含一個 test-case（測試案例）物件，那是個 `FileReaderTester` 實體。創建 test-case 物件時，我傳給其建構式一個字串，這正是待測函式的名稱。這會創建出一個物件，用以測試被指定的函式。這個測試係透過 Java 反射機制（reflection）和物件繫結在一起。你可以自由下載 JUnit 源碼，看看它究竟如何做到。至於我，我只把它當作一種魔法。

要將整個測試執行起來，還需要一個獨立的 TestRunner class。TestRunner 有兩個版本，其中一個有漂亮的圖形使用介面（GUI），另一個採用文字介面。我可以在 main 函式中叫用「文字介面」版：

```
class FileReaderTester...
    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
}
```

這段程式碼創建出一個 TestRunner，並要它去測試 FileReaderTester class。當我執行它，我看到：

```
.
Time: 0.110
OK (1 tests)
```

對於每個執行起來的測試，JUnit 都會輸出一個句點，這樣你就可以直觀看到測試進展。它會告訴你整個測試用了多長時間。如果所有測試都沒有出錯，它就會說 "OK"，並告訴你執行了多少筆測試。我可以執行上千筆測試，如果一切良好，我會看到那個 "OK"。對於自我測試碼來說，這個簡單的回應至關重要，沒有它我就不可能經常執行這些測試。有了這個簡單回應，你可以執行一大堆測試然後去吃個午飯（或開個會），回來之後再看看測試結果。

頻繁地執行測試。每次編譯請把測試也考慮進去 — 每天至少執行每個測試一次。

重構過程中，你可以只執行少數幾項測試，它們主要用來檢查當下正在開發或整理的程式碼。是的，你可以只執行少數幾項測試，這樣肯定比較快，否則整個測試會減低你的開發速度，使你開始猶豫是否還要這樣下去。千萬別屈服於這種誘惑，否則你一定會付出代價。

如果測試出錯，會發生什麼事？爲了展示這種情況，我故意放一隻臭蟲進去：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('2' == ch);          // deliberate error
}
```

得到如下結果：

```
.F
Time: 0.220
!!!FAILURES!!!
```

```
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead
test.framework.AssertionFailedError
```

JUnit 警告我測試失敗，並告訴我這項失敗具體發生在哪個測試身上。不過這個錯誤訊息並不特別有用。我可以使用另一種形式的 assert，讓錯誤訊息更清楚些：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals('m',ch);
}
```

你做的絕大多數 asserts 都是對兩個值進行比較，檢驗它們是否相等，所以 JUnit 框架爲你提供 `assertEquals()`。這個函式很簡單：以 `equals()`進行物件比較，以運算子 `==` 進行數值比較 — 我自己常忘記區分它們。這個函式也輸出更具意義的錯誤訊息：

```
.F
Time: 0.170

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead "expected:"m"but was:"d""
```

我應該提一下：編寫測試碼時，我往往一開始先讓它們失敗。面對既有程式碼，要不我就修改它（如果我能碰觸源碼的話），使它測試失敗，要不就在 assertions 中放一個錯誤期望值，造成測試失敗。之所以這麼做，是爲了向自己證明：測試機制的確可以執行，並且的確測試了它該測試的東西（這就是爲什麼上面兩種作法中我比較喜歡修改待測碼的原因）。這可能有些偏執，或許吧，但如果測試碼所測的東西並非你想測的東西，你真的有可能被搞得迷迷糊糊。

除了捕捉「失敗」（failures，也就是 assertions 之結果爲 "false"），JUnit 還可以捕捉「錯誤」（errors，意料外的異常）。如果我關閉 input stream，然後試圖讀取它，就應該得到一個異常（exception）。我可以這樣測試：

```
public void testRead() throws IOException {
    char ch = '&';
    _input.close();
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();    // will throw exception
    assertEquals('m',ch);
}
```

執行上述測試，我得到這樣的結果：

```
.E
Time: 0.110
!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTester.testRead
java.io.IOException: Stream closed
```

區分失敗（failures）和錯誤（errors）是很有用的，因爲它們的出現形式不同，排除的過程也不同。

JUnit 還包含一個很好的圖形使用介面（GUI，見圖 4.2）。如果所有測試都順利通過，視窗下端的進度桿（progress bar）就呈綠色；如果有任何一個測試失敗，進度桿就呈紅色。你可以丟下這個 GUI 不管，整個環境會自動將你在程式碼所做的任何修改連接（links）進來。這是一個非常方便的測試環境。

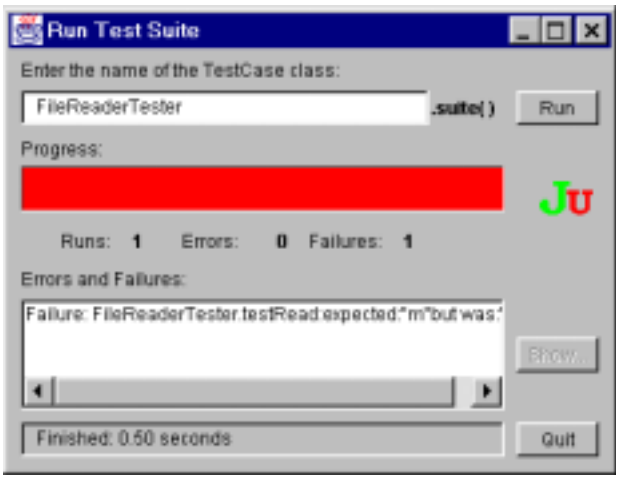


圖 4.2 JUnit 的圖形使用界面

單元測試（Unit Tests）和功能測試（Functional Tests）

JUnit 框架的用途是單元測試，所以我應該講講單元測試和功能測試之間的差異。我一直掛在嘴上的其實是「單元測試」，編寫這些測試的目的是爲了提高身爲一個程式員的生產效率。至於讓品管部門開心，那只是附帶效果而已。單元測試

是高度區域化（localized）的東西，每個 test class 只對單一 package 運作。它能夠測試其他 packages 的介面，除此之外它將假設其他 package 一切正常。

功能測試就完全不同。它們用來保證軟體能夠正常運作。它們只負責向客戶提供品質保證，並不關心程式員的生產力。它們應該由一個喜歡尋找臭蟲的個別團隊來開發。這個團隊應該使用重量級工具和技術來幫助自己開發良好的功能測試。

一般而言，功能測試儘可能把整個系統當作一個黑箱。面對一個 GUI 待測系統，它們透過 GUI 來操作那個系統。面對檔案更新程式或資料庫更新程式，功能測試只觀察特定輸入所導致的資料變化。

一旦功能測試者或終端用戶找到軟體中的一隻臭蟲，要除掉它至少需要做兩件事。當然你必須修改程式碼，才得以排除錯誤，但你還應該添加一個單元測試，讓它揭發這隻臭蟲。事實上，每當收到臭蟲提報（bug report），我都首先編寫一個單元測試，使這隻臭蟲浮現。如果需要縮小臭蟲出沒範圍，或如果出現其他相關失敗（failures），我就會編寫不只一個測試。我使用單元測試來幫助我盯住臭蟲，並確保我的單元測試不會有類似的漏網之…呃…臭蟲。

每當你接獲臭蟲提報（bug report），請先撰寫一個單元測試來揭發這隻臭蟲。

JUnit 框架被設計用來編寫單元測試。功能測試往往以其他工具輔助進行，例如某些擁有 GUI（圖形使用介面）的測試工具，然而通常你還得撰寫一些與你的應用程式息息相關的測試工具，俾能夠比單純使用 GUI scripts（腳本語言）更輕鬆地管理 test cases（測試案例）。你也可以運用 JUnit 來執行功能測試，但這通常不是最有效的形式。當我要進行重構時，我倚賴程式員的好朋友：單元測試。

4.3 添加更多測試

現在，我們應該繼續添加更多測試。我遵循的風格是：觀察 class 該做的所有事情，然後針對任何一項功能的任何一種可能失敗情況，進行測試。這不同於某些程式員提倡的「測試所有 public 函式」。記住，測試應該是一種風險驅動（risk driven）行爲，測試的目的是希望找出現在或未來可能出現的錯誤。所以我不會去測試那些僅僅讀或寫一個欄位的存取函式（accessors），因爲它們太簡單了，不大可能出錯。

這一點很重要，因爲如果你撰寫過多測試，結果往往測試量反而不夠。我常常閱讀許多測試相關書籍，我的反應是：測試需要做那麼多工作，令我退避三舍。這種書起不了預期效果，因爲它讓你覺得測試有大量工作要做。事實上，哪怕只做一點點測試，你也能從中受益。測試的要訣是：測試你最擔心出錯的部分。這樣你就能從測試工作中得到最大利益。

編寫未臻完善的測試並實際執行，好過對完美測試的無盡等待。

現在，我的目光落到了 `read()`。它還應該做些什麼？文件上說，當 input stream 到達檔案尾端，`read()`應該傳回 `-1`（在我看來這並不是個很好的協定，不過我猜這會讓 C 程式員倍感親切）。讓我們來測試一下。我的文字編輯器告訴我，我的測試檔共有 141 個字元，於是我撰寫測試碼如下：

```
public void testReadAtEnd() throws IOException {
  int ch = -1234;
  for (int i = 0; i < 141; i++)
     ch = _input.read();
  assertEquals("read at end", -1, _input.read());
}
```

爲了讓這個測試執行起來，我必須把它添加到 test suit（測試套件）中：

```
public static Test suite() {    // 譯註：原本在 p.93
  TestSuite suite= new TestSuite();
  suite.addTest(new FileReaderTester("testRead"));
  suite.addTest(new FileReaderTester("testReadAtEnd"));
  return suite;
}
```

當 test suit（測試套件）執行起來，它會告訴我它的每個成分 — 也就是這兩個 test cases（測試案例）— 的執行情況。每個案例都會呼叫 `setUp()`，然後執行測試碼，最終呼叫 `tearDown()`。每次測試都呼叫 `setUp()`和 `tearDown()`是很重要的，因爲這樣才能保證測試之間彼此隔離。也就是說我們可以按任意順序執行它們，不會對它們的結果造成任何影響。

老要記住將 test cases 添加到 `suite()`，實在是件痛苦的事。幸運的是 Erich Gamma 和 Kent Beck 和我一樣懶，所以他們提供了一條途徑來避免這種痛苦。`TestSuite` class 有個特殊建構式，接受一個 class 爲參數，創建出來的 test suite 會將該 class

內所有以 "test" 起頭的函式都當作 test cases 包含進來。如果遵循這一命名習慣，就可以把我的 main()改爲這樣：

```
public static void main (String[] args) {     // 譯註：原出現於 p94
  junit.textui.TestRunner.run (new TestSuite(FileReaderTester.class));
}
```

這樣，我寫的每一個測試函式便都被自動添加到 test suite 中。

測試的一項重要技巧就是「尋找邊界條件」。對 read()而言，邊界條件應該是第一個字元、最後一個字元、倒數第二個字元：

```
public void testReadBoundaries()throwsIOException {
    assertEquals("read first char",'B', _input.read());
    int ch;
    for (int i = 1; i <140; i++)
        ch = _input.read();
    assertEquals("read last char",'6',_input.read());
    assertEquals("read at end",-1,_input.read());
}
```

你可以在 assertions 中加入一條訊息。如果測試失敗，這條訊息就會被顯示出來。

考慮可能出錯的邊界條件，把測試火力集中在那兒。

「尋找邊界條件」也包括尋找特殊的、可能導致測試失敗的情況。對於檔案相關測試，空檔案是個不錯的邊界條件：

```
public void testEmptyRead()throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream (empty);
  out.close();
  FileReader in = new FileReader (empty);
  assertEquals (-1, in.read());
}
```

現在我爲這個測試產生一些額外的 test fixture（測試裝備）。如果以後還需要空檔案，我可以把這些程式碼移至 setUp()，從而將「空檔案」加入常規 test fixture。

```
protected void setUp(){
  try {
      _input = new FileReader("data.txt");
      _empty = newEmptyFile();
  } catch(IOException e){
      throw new RuntimeException(e.toString());
  }
}
```

```
private FileReader newEmptyFile() throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream(empty);
  out.close();
  return newFileReader(empty);
}

public void testEmptyRead() throws IOException {
  assertEquals (-1, _empty.read());
}
```

如果讀取檔案末尾之後的位置，會發生什麼事？同樣應該返回-1。現在我再加一個測試來探測這一點：

```
public void testReadBoundaries() throws IOException {
  assertEquals("read first char",'B', _input.read());
  int ch;
  for (int i = 1;i <140; i++)
     ch = _input.read();
  assertEquals("read last char", '6', _input.read());
  assertEquals("read at end",-1,_input.read());
  assertEquals ("readpast end", -1, _input.read());
}
```

注意，我在這裡扮演「程式公敵」的角色。我積極思考如何破壞程式碼。我發現這種思維能夠提高生產力，並且很有趣。它縱容了我心智中比較促狹的那一部分。

測試時，別忘了檢查預期的錯誤是否如期出現。如果你嘗試在 stream 被關閉後再讀取它，就應該得到一個 IOException 異常，這也應該被測試出來：

```
public void testReadAfterClose() throws IOException{
  _input.close();
  try {
     _input.read();
     fail ("no exception for read past end");
  } catch (IOException io) {}
}
```

IOException 之外的任何異常都將以一般方式形成一個錯誤。

當事情被大家認爲應該會出錯時，別忘了檢查彼時是否有異常如預期般地被拋出。

請遵循這些規則，不斷豐富你的測試。對於某些比較複雜的 classes，可能你得花費一些時間來瀏覽其介面，但是在此過程中你可以真正理解這個介面。而且這對於考慮錯誤情況和邊界情況特別有幫助。這是在編寫程式碼的同時（甚至之前）編寫測試碼的另一個好處。

隨著 tester classes 愈來愈多，你可以產生另一個 class，專門用來包含「由其他 tester classes 所形成」的測試套件（test suite）。這很容易做到，因爲一個測試套件本來就可以包含其他測試套件。這樣，你就可以擁有一個「主控的」（master）test class：

```
class MasterTester extends TestCase {
  public static void main (String[] args) {
     junit.textui.TestRunner.run (suite());
  }
  public static Test suite() {
     TestSuite result = new TestSuite();
     result.addTest(new TestSuite(FileReaderTester.class));
     result.addTest(new TestSuite(FileWriterTester.class));
     // and so on...
     return result;
  }
}
```

什麼時候應該停下來？我相信這樣的話你聽過很多次：「任何測試都不能證明一個程式沒有臭蟲」。這是真的，但這不會影響「測試可以提高編程速度」。我曾經見過數種測試規則建議，其目的都是保證你能夠測試所有情況的一切組合。這些東西值得一看，但是別讓它們影響你。當測試數量達到一定程度之後，測試效益就會呈現遞減態勢，而非持續遞增；如果試圖編寫太多測試，你也可能因爲工作量太大而氣餒，最後什麼都寫不成。你應該把測試集中在可能出錯的地方。觀察程式碼，看哪兒變得複雜；觀察函式，思考哪些地方可能出錯。是的，你的測試不可能找出所有臭蟲，但一旦進行重構，你可以更好地理解整個程式，從而找到更多臭蟲。雖然我總是以單獨一個測試套件（test suite）開始重構，但前進途中我總會加入更多測試。

不要因爲「測試無法捕捉所有臭蟲」，就不撰寫測試碼，因爲測試的確可以捕捉到大多數臭蟲。

物件技術有個微妙處：繼承（inheritance）和多型（polymorphism）會讓測試變得比較困難，因爲將有許多種組合需要測試。如果你有三個彼此合作的 abstract classes，每個 abstract class 有三個 subclasses，那麼你總共擁有九個可供選擇的

classes，和 27 種組合。我並不總是試著測試所有可能組合，但我會儘量測試每一個 classes，這可以大大減少各種組合所造成的風險。如果這些 classes 之間彼此有合理的獨立性，我很可能不會嘗試所有組合。是的，我總有可能遺漏些什麼，但我覺得「花合理時間抓出大多數臭蟲」要好過「窮盡一生抓出所有臭蟲」。

測試碼和產品碼（待測碼）之間有個區別：你可以放心地拷貝、編輯測試碼。處理多種組合情況以及面對多個可供選擇的 classes 時，我經常這麼做。首先測試「標準發薪過程」，然後加上「資歷」和「年底前停薪」條件，然後又去掉這兩個條件…等等等。只要在合理的測試裝備（test fixture）上準備好一些簡單的替換樣本，我就能夠很快生成不同的 test case（測試案例），然後就可以利用重構手法分解出真正常用的各種東西。

我希望這一章能夠讓你對於「撰寫測試碼」有一些感覺。關於這個主題，我可以說上很多，但如果那麼做，就有點喧賓奪主了。總而言之，請構築一個良好的臭蟲檢測器（bug detector）並經常執行它；這對任何開發工作都是一個美好的工具，並且是重構的前提。

5

重構名錄

Toward a Catalog of Refactorings

本書 5~12 章構成了一份重構名錄草案（initial catalog of refactorings）。其中所列的重構手法來自我最近數年的心得。這份名錄並非鉅細靡遺，但應該足可爲你提供一個堅實的起點，讓你得以開始自己的重構工作。

5.1 重構的記錄格式（Format of Refactorings）

介紹重構時，我採用一種標準格式。每個重構手法都有如下五個部分：

- 首先是名稱（**name**）。建造一個重構詞彙表，名稱是很重要的。這個名稱也就是我將在本書其他地方使用的名稱。
- 名稱之後是一個簡短概要（**summary**），簡單介紹此一重構手法的適用情景，以及它所做的事情。這部分可以幫助你更快找到你所需要的重構手法。
- 動機（**motivation**），爲你介紹「爲什麼需要這個重構」和「什麼情況下不該使用這個重構」。
- 作法（**mechanics**），簡明扼要地一步一步介紹如何進行此一重構。
- 範例（**examples**），以一個十分簡單的例子說明此重構手法如何運作。

「概要」（**summary**）包括三個部分：(1) 一個簡短文句，介紹這個重構能夠幫助的問題；(2) 一段簡短陳述，介紹你應該做的事；(3) 一幅速寫圖，簡單展現重構前後示例；有時候我展示程式碼，有時候我展示統一建模語言（UML）圖。哪一種形式能更好呈現該重構的本質，我就使用該種形式（本書所有 UML 圖都根據實作觀點（implementation perspective）而畫 [Fowler, UML]。）如果你以前見過這一重構手法，那麼速寫圖能夠讓你迅速了解這一重構的概況；如果你不曾見過這個重構，可能就需要瀏覽整個範例，才能得到較好的認識。

「作法」（**mechanics**）出自我自己的筆記。這些筆記是爲了讓我在一段時間不做某項重構之後還能記得怎麼做。它們也頗爲簡潔，通常不會解釋「爲什麼要這麼做那麼做」。我會在「範例」（**examples**）給出更多解釋。這麼一來「作法」就成了簡短的筆記。如果你知道該使用哪個重構，但記不清具體步驟，可以參考「作法」部分（至少我是這麼使用它們的）；如果你初次使用某個重構，可能「作法」對你還不夠，你還需要閱讀「範例」。

撰寫「作法」的時候，我儘量將重構的每個步驟都寫得簡短。我強調安全的重構方式，所以應該採用非常小的步驟，並且在每個步驟之後進行測試。真正工作時我通常會採用比這裡介紹的「嬰兒學步」稍大些的步驟，然而一旦遇上臭蟲，我就會撤銷上一步，換用比較小的步驟。這些步驟還包含一些特定狀況的參考，所以它們也有檢驗表（checklist）的作用；我自己經常忘掉這些該做的事情。

「範例」（**examples**）像是簡單而有趣的教科書。我使用這些範例是爲了幫助解釋重構的基本要素，最大限度地避免其他枝節，所以我希望你能原諒其中的簡化工作（它們當然不是優秀商用物件設計的適當例子）。不過我敢肯定你一定能在你手上那些更複雜的情況中使用它們。某些十分簡單的重構乾脆沒有範例，因爲我覺得爲它們加上一個範例不會有多大意義。

更明確地說，加上「範例」僅僅是爲了闡釋當時討論的重構手法。通常那些程式碼最終仍有其他問題，但修正那些問題需要用到其他重構手法。某些情況下數個重構經常被一併運用，這時候我會把某個範例拿到另一個重構中繼續使用。大部分時候，一個範例只爲一項重構而設計，這麼做是爲了讓每一項重構手法自給自足（self-contained），因爲這份重構名錄的首要目的還是作爲參考工具。

這些例子不會告訴你「如何設計一個 "employee" 物件或一個 "order" 物件」。這些例子的存在純粹只是爲了說明重構，除此之外別無用途。例如你會發現，我在這些例子中用 `double` 數據來表示貨幣金額。我之所以這樣做，只是爲了讓例子簡單一些，因爲「以什麼形式表示金額」對於重構本身並不重要。在真正的商用軟體中，我強烈建議你不要以 `double` 表現金額。如果真要表示貨幣金額，我會使用 ***Quantity*** 範式 [Fowler, AP]。

撰寫本書之際，商業開發中使用得最多的是 Java 1.1，所以我的大多數例子也以 Java 1.1 寫就，這從我對群集（collections）的使用就可以明顯看出來。本書即將完成之時，Java 2 已經正式發佈。但我不覺得有必要修改所有這些例子，因爲對重構來說，群集（collections）也是次要的。但是有些重構手法，例如 *Encapsulate Collection*（208），在 Java 1.2 中有所不同。這時候我會同時解釋 Java 2 和 Java 1.1。

修改後的程式碼可能被埋沒在未修改的程式碼中，難以一眼看出，所以我使用粗體（**`blodface code`**）突顯修改過的程式碼。但我並沒有對所有修改過的程式碼都使用粗體字，因爲一旦修改過的程式碼太多，全都粗體反而不能突顯重點。

5.2 尋找引用點（Finding References）

很多重構都要求你找到對於某個函式（methods）、某個欄位（fields）或某個 class 的所有引用點（指涉點）。做這件事的時候，記得尋求計算機的幫助。有了計算機的幫助，你可以減少「遺漏某個引用點」的機率，而且通常比人工搜尋更快。

大多數語言都把計算機程式碼當做文字檔來處理，所以最好的幫手就是一個適當的文字搜尋工具。許多編程環境都允許你在一個檔案或者一組檔案中進行文字搜尋，你的搜尋目標的存取控制（access control）會告訴你需要搜尋的檔案範圍。

不要盲目地「搜尋–替換」。你應該檢查每一個引用點，確定它的確指向你想要替換的東西。或許你很擅長運用搜尋手法，但我總是用心去檢查，以確保替換時不出錯。要知道，你可以在不同的 classes 中使用相同函式名稱，也可以在同一個 class 中使用名稱相同但署名（signature）不同的函式，所以「替換」出錯機會是很高的。

在強型（strongly typed）語言中，你可以讓編譯器幫助你捕捉漏網之魚。你往往可以直接刪除舊部分，讓編譯器幫你找出因此而被吊掛起來（dangling）的引用點。這樣做的好處是：編譯器會找到所有被吊掛的引用點。但是這種技巧也存在問題。

首先，如果被刪除的部分在繼承體系（hierarchy）中宣告不止一次，那麼編譯器也會被迷惑。尤其當你處理一個被覆寫（overridden）多次的函式時，情況更是如此。所以如果你在一個繼承體系中工作，請先利用文字搜尋工具，檢查是否有其他 class 宣告了你正在處理的那個函式。

第二個問題是：編譯器可能太慢，從而使你的工作失去效率。如果真是這樣，請先使用文字搜尋工具，最起碼編譯器可以復查你的工作。只有當你想移除某個部

分時，才請你這樣做。常常你會想先觀察這一部分的所有運用情況，然後才決定下一步。這種情況下你必須使用文字搜尋法（而不是倚賴編譯器）。

第三個問題是：編譯器無法找到經由反射機制（reflection）而得到的引用點。這也是我們應該小心使用反射的原因之一。如果系統中使用了反射，你就必須以文字搜尋找出你想找的東西，測試份量也因此加重。有些時候我會建議你只編譯，不測試，因爲編譯器通常會捕捉到可能的錯誤。如果使用反射（reflection），所有這些便利都沒有了，你必須爲許多編譯搭配測試。

某些 Java 開發環境（特別值得一提的是 IBM 的 VisualAge）承受了 Smalltalk 瀏覽器的影響。在這些開發環境中你應該使用功能表選項（menu options）來搜尋引用點，而不是使用文字搜尋工具。因爲這些開發環境並不以文字檔保存程式碼，而是使用一個內建資料庫。只要習慣了這些功能表選項，你會發現它們往往比難用的文字搜尋工具出色得多。

5.3 這些重構準則有多成熟？

任何技術作家都會面對這樣一個問題：該在何時發表自己的想法？發表愈早，人們愈快能夠運用新想法、新觀念。但只要是人，總是不斷在學習。如果過早發表半生不熟的想法，這些思想可能並不完善，甚至可能給那些嘗試採用它們的人帶來麻煩。

重構的基本技巧 — 小步前進、頻繁測試 — 已經得到多年的實踐檢驗，特別是在 Smalltalk 社群中。所以，我敢保證，重構的這些基礎思想是非常可靠的。

本書中的重構準則是我自己使用重構的記錄。是的，我全都用過它們。但是「使用某個重構手法」和「將它濃縮成可在這裡給出之作法步驟」是有區別的。特別是在一些十分特殊的情況下，偶爾你會看見一些問題突然湧現。我並沒有讓很多人進行我所寫下的這些技術步驟以圖發現這一類問題。所以，使用重構的時候，請隨時知道自己在做什麼。記住，就像看著食譜做菜一樣，你必須讓這些重構準則適應你自己的情況。如果你遇上一個有趣的問題，請以電子郵件告訴我，我會試著把你的情況告訴其他人。

關於這些重構手法，另一個需要記住的就是：我是在「單行程」（single-process）軟體這一大前提下考慮並介紹它們的。我很希望看到有人介紹用於並行式（concurrent）和分散式（distributed）程式設計的重構技術。這樣的重構將是完全

不同的。舉個例子，在單行程軟體中，你永遠不必操心多麼頻繁地呼叫某個函式，因爲函式的呼叫成本很低。但在分散式軟體中，函式的往返必須被減至最低限度。在這些特殊編程領域中有著完全不同的重構技術，這已超越本書主題。

許多重構手法，例如 *Replace Type Code with State/Strategy*（227）和 *Form Template Method*（345），都涉及「向系統引入範式（patterns）」。正如 GoF（Gang of Four，四巨頭）的經典著作中所說，『設計範式…爲你的重構行爲提供了目標」。範式和重構之間有著一種與生俱來的關係。範式是你希望到達的目標，重構則是到達之路。本書並沒有提供「助你完成所有知名範式」的重構手法，甚至連 GoF 的 23 個知名範式 [Gang of Four] 都沒有能夠全部涵蓋。這也從某個角度反映出這份名錄的不完整。我希望有一天這個缺陷能夠被填補。

運用重構的時候，請記住：它們僅僅是一個起點。毋庸置疑，你一定可以找出箇中缺陷。我之所以選擇現在發表它們，因爲我相信，儘管它們還不完美，但的確有用。我相信它們能給你一個起點，然後你可以不斷提高自己的重構能力。這正是它們帶給我的。

隨著你用過愈來愈多的重構手法，我希望，你也開始發展屬於你自己的重構手法。但願本書例子能夠激發你的創造力，並給你一個起點，讓你知道從何入手。我很清楚現實存在的重構，比我這裡介紹的還要多得多。如果你真的提出了一些新的重構手法，請給我一封電子郵件。

6

重新組織你的函式

Composing Methods

我的重構手法中，很大一部分是對函式進行整理，使之更恰當地包裝程式碼。幾乎所有時刻，問題都源於 Long Methods（過長函式）。這很討厭，因爲它們往往包含太多資訊，這些資訊又被函式錯綜複雜的邏輯掩蓋，不易鑑別。對付過長函式，一項重要的重構手法就是 *Extract Method*（110），它把一段程式碼從原先函式中提取出來，放進一個單獨函式中。*Inline Method*（117）正好相反：將一個函式呼叫動作替換爲該函式本體。如果在進行多次提煉之後，意識到提煉所得的某些函式並沒有做任何實質事情，或如果需要回溯恢復原先函式，我就需要 *Inline Method*（117）。

Extract Method（110）最大的困難就是處理區域變數，而暫時變數則是其中一個主要的困難源頭。處理一個函式時，我喜歡運用 *Replace Temp with Query*（120）去掉所有可去掉的暫時變數。如果很多地方使用了某個暫時變數，我就會先運用 *Split Temporary Variable*（128）將它變得比較容易替換。

但有時候暫時變數實在太混亂，難以替換。這時候我就需要使用 *Replace Method with Method Object*（135）。它讓我可以分解哪怕最混亂的函式，代價則是引入一個新 class。

參數帶來的問題比暫時變數稍微少一些，前提是你不在函式內賦值給它們。如果你已經這樣做了，就得使用 *Remove Assignments to Parameters*（131）。

函式分解完畢後，我就可以知道如何讓它工作得更好。也許我還會發現演算法可以改進，從而使程式碼更清晰。這時我就使用 *Substitute Algorithm*（139）引入更清晰的演算法。

6.1 Extract Method

你有一段程式碼可以被組織在一起並獨立出來。

將這段程式碼放進一個獨立函式中，並讓函式名稱解釋該函式的用途。

```
void printOwing(double amount) {
   printBanner();

   //print details
   System.out.println ("name:" + _name);
   System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println ("name:" + _name);
   System.out.println ("amount" + amount);
}
```

動機（Motivation）

Extract Method 是我最常用的重構手法之一。當我看見一個過長的函式或者一段需要注釋才能讓人理解用途的程式碼，我就會將這段程式碼放進一個獨立函式中。

有數個原因造成我喜歡簡短而有良好命名的函式。首先，如果每個函式的粒度都很小（finely grained），那麼函式之間彼此復用的機會就更大；其次，這會使高階函式碼讀起來就像一系列注釋；再者，如果函式都是細粒度，那麼函式的覆寫（override）也會更容易些。

的確，如果你習慣看大型函式，恐怕需要一段時間才能適應這種新風格。而且只有當你能給小型函式很好地命名時，它們才能真正起作用，所以你需要在函式名稱下點功夫。人們有時會問我，一個函式多長才算合適？在我看來，長度不是問

題，關鍵在於函式名稱和函式本體之間的語意距離（semantic distance）。如果提煉動作（extracting）可以強化程式碼的清晰度，那就去做，就算函式名稱比提煉出來的程式碼還長也無所謂。

作法（Mechanics）

- 創造一個新函式，根據這個函式的意圖來給它命名（以它「做什麼」來命名，而不是以它「怎樣做」命名）。

  ⇨ 即使你想要提煉（extract）的程式碼非常簡單，例如只是一條訊息或一個函式呼叫，只要新函式的名稱能夠以更好方式昭示程式碼意圖，你也應該提煉它。但如果你想不出一個更有意義的名稱，就別動。

- 將提煉出的程式碼從源函式（source）拷貝到新建的目標函式（target）中。
- 仔細檢查提煉出的程式碼，看看其中是否引用了「作用域（scope）限於源函式」的變數（包括區域變數和源函式參數）。
- 檢查是否有「僅用於被提煉碼」的暫時變數（temporary variables）。如果有，在目標函式中將它們宣告爲暫時變數。
- 檢查被提煉碼，看看是否有任何區域變數（local-scope variables）的值被它改變。如果一個暫時變數值被修改了，看看是否可以將被提煉碼處理爲一個查詢（query），並將結果賦值給相關變數。如果很難這樣做，或如果被修改的變數不止一個，你就不能僅僅將這段程式碼原封不動地提煉出來。你可能需要先使用 *Split Temporary Variable*（128），然後再嘗試提煉。也可以使用 *Replace Temp with Query*（120）將暫時變數消滅掉（請看「範例」中的討論）。
- 將被提煉碼中需要讀取的區域變數，當做參數傳給目標函式。
- 處理完所有區域變數之後，進行編譯。
- 在源函式中，將被提煉碼替換爲「對目標函式的呼叫」。

  ⇨ 如果你將任何暫時變數移到目標函式中，請檢查它們原本的宣告式是否在被提煉碼的外圍。如果是，現在你可以刪除這些宣告式了。

- 編譯，測試。

範例（Examples）：無區域變數（No Local Variables）

在最簡單的情況下，*Extract Method*（110）易如反掌。請看下列函式：

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

我們可以輕鬆提煉出「列印 banner」的程式碼。我只需要剪下、貼上、再插入一個函式呼叫動作就行了：

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}

void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

範例（Examples）：有區域變數（Using Local Variables）

果真這麼簡單，這個重構手法的困難點在哪裡？是的，就在區域變數，包括傳進源函式的參數和源函式所宣告的暫時變數。區域變數的作用域僅限於源函式，所以當我使用 *Extract Method*（110）時，必須花費額外功夫去處理這些變數。某些時候它們甚至可能妨礙我，使我根本無法進行這項重構。

區域變數最簡單的情況是：被提煉碼只是讀取這些變數的值，並不修改它們。這種情況下我可以簡單地將它們當做參數傳給目標函式。所以如果我面對下列函式：

```
void printOwing() {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
   }

   //print details
   System.out.println ("name:" + _name);
   System.out.println ("amount" + outstanding);
}
```

我就可以將「列印詳細資訊」這一部分提煉爲「帶一個參數的函式」：

```
void printOwing() {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
   }

   printDetails(outstanding);
}

void printDetails (double outstanding) {
   System.out.println ("name:" + _name);
   System.out.println ("amount" + outstanding);
}
```

必要的話，你可以用這種手法處理多個區域變數。

如果區域變數是個物件，而被提煉碼呼叫了會對該物件造成修改的函式，也可以如法炮製。你同樣只需將這個物件作爲參數傳遞給目標函式即可。只有在被提煉碼真的對一個區域變數賦值的情況下，你才必須採取其他措施。

範例（Examples）：對區域變數再賦值（Reassigning）

如果被提煉碼對區域變數賦值，問題就變得複雜了。這裡我們只討論暫時變數的問題。如果你發現源函式的參數被賦值，應該馬上使用 *Remove Assignments to Parameters*（131）。

被賦值的暫時變數也分兩種情況。較簡單的情況是：這個變數只在被提煉碼區段中使用。果真如此，你可以將這個暫時變數的宣告式移到被提煉碼中，然後一起提煉出去。另一種情況是：被提煉碼之外的程式碼也使用了這個變數。這又分爲兩種情況：如果這個變數在被提煉碼之後未再被使用，你只需直接在目標函式中修改它就可以了；如果被提煉碼之後的程式碼還使用了這個變數，你就需要讓目標函式返回該變數改變後的值。我以下列程式碼說明這幾種不同情況：

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

現在我把「計算」程式碼提煉出來：

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Enumeration 變數 `e` 只在被提煉碼中用到，所以我可以將它整個搬到新函式中。`double` 變數 `outstanding` 在被提煉碼內外都被用到，所以我必須讓提煉出來的新函式傳回它。編譯測試完成後，我就把回傳值改名，遵循我的一貫命名原則：

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result = each.getAmount();
    }
    return result;
}
```

本例中的 `outstanding` 變數只是很單純地被初始化爲一個明確初值，所以我可以只在新函式中對它初始化。如果程式碼還對這個變數做了其他處理，我就必須將它的值作爲參數傳給目標函式。對於這種變化，最初程式碼可能是這樣：

```
void printOwing(double previousAmount) {
    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
         Order each = (Order) e.nextElement();
         outstanding += each.getAmount();
    }
    printDetails(outstanding);
}
```

提煉後的程式碼可能是這樣：

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
         Order each = (Order) e.nextElement();
         result += each.getAmount();
    }
    return result;
}
```

編譯並測試後，我再將變數 `outstanding` 的初始化過程整理一下：

```
void printOwing(double previousAmount) {
    printBanner();
    double outstanding = getOutstanding(previousAmount * 1.2);
    printDetails(outstanding);
}
```

這時候，你可能會問：『如果需要傳回的變數不止一個，又該怎麼辦呢？』

你有數種選擇。最好的選擇通常是：挑選另一塊程式碼來提煉。我比較喜歡讓每個函式都只傳回一個值，所以我會安排多個函式，用以傳回多個值。如果你使用的語言支援「輸出式參數」（output parameters），你可以使用它們帶回多個回傳值。但我還是儘可能選擇單一回返值。

暫時變數往往爲數衆多，甚至會使提煉工作舉步維艱。這種情況下，我會嘗試先運用 *Replace Temp with Query*（120）減少暫時變數。如果即使這麼做了提煉依舊困難重重，我就會動用 *Replace Method with Method Object*（135），這個重構手法不在乎程式碼中有多少暫時變數，也不在乎你如何使用它們。

6.2 Inline Method

一個函式，其本體（method body）應該與其名稱（method name）同樣清楚易懂。

在函式呼叫點插入函式本體，然後移除該函式。

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

動機（Motivation）

本書經常以簡短的函式表現動作意圖，這樣會使程式碼更清晰易讀。但有時候你會遇到某些函式，其內部程式碼和函式名稱同樣清晰易讀。也可能你重構了該函式，使得其內容和其名稱變得同樣清晰。果真如此，你就應該去掉這個函式，直接使用其中的程式碼。間接性可能帶來幫助，但非必要的間接性總是讓人不舒服。

另一種需要使用 *Inline Method*（117）的情況是：你手上有一群組織不甚合理的函式。你可以將它們都 *inlining* 到一個大型函式中，再從中提煉出組織合理的小型函式。Kent Beck 發現，實施 *Replace Method with Method Object*（135）之前先這麼做，往往可以獲得不錯的效果。你可以把你所要的函式（有著你要的行爲）的所有呼叫對象的函式內容都 *inlining* 到 method object（函式物件）中。比起既要移動一個函式，又要移動它所呼叫的其他所有函式，「將大型函式作爲單一整體來移動」會比較簡單。

如果別人使用了太多間接層，使得系統中的所有函式都似乎只是對另一個函式的簡單委託（delegation），造成我在這些委託動作之間暈頭轉向，那麼我通常都會使用 *Inline Method*（117）。當然，間接層有其價值，但不是所有間接層都有價值。試著使用 inline，我可以找到那些有用的間接層，同時將那些無用的間接層消除掉。

作法（Mechanics）

- ☐ 檢查函式，確定它不具多型性（is not polymorphic）。
  - ⇨ 如果 subclass 繼承了這個函式，就不要將此函式 inline 化，因爲 subclass 無法覆寫（override）一個根本不存在的函式。
- ☐ 找出這個函式的所有被呼叫點。
- ☐ 將這個函式的所有被呼叫點都替換爲函式本體（程式碼）。
- ☐ 編譯，測試。
- ☐ 刪除該函式的定義。

被我這樣一寫，*Inline Method*（117）似乎很簡單。但情況往往並非如此。對於遞迴呼叫、多回返點、inlining 至另一個物件中而該物件並無提供存取函式（accessors）…，每一種情況我都可以寫上好幾頁。我之所以不寫這些特殊情況，原因很簡單：如果你遇到了這樣的複雜情況，那麼就不應該使用這個重構手法。

6.3 Inline Temp

你有一個暫時變數，只被一個簡單運算式賦值一次，而它妨礙了其他重構手法。

將所有對該變數的引用動作，替換爲對它賦值的那個運算式本身。

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

動機（Motivation）

Inline Temp（119）多半是作爲 *Replace Temp with Query*（120）的一部分來使用，所以真正的動機出現在後者那兒。惟一單獨使用 *Inline Temp*（119）的情況是：你發現某個暫時變數被賦予某個函式呼叫的回返值。一般來說，這樣的暫時變數不會有任何危害，你可以放心地把它留在那兒。但如果這個暫時變數妨礙了其他的重構手法 — 例如 *Extract Method*（110），你就應該將它 inline 化。

作法（Mechanics）

- 如果這個暫時變數並未被宣告爲 `final`，那就將它宣告爲 `final`，然後編譯。
  - ⇨ 這可以檢查該暫時變數是否真的只被賦值一次。
- 找到該暫時變數的所有引用點，將它們替換爲「爲暫時變數賦值」之述句中的等號右側運算式。
- 每次修改後，編譯並測試。
- 修改完所有引用點之後，刪除該暫時變數的宣告式和賦值述句。
- 編譯，測試。

6.4 Replace Temp with Query

你的程式以一個暫時變數（temp）保存某一運算式的運算結果。

將這個運算式提煉到一個獨立函式（譯註：所謂查詢式，query）中。將這個暫時變數的所有「被引用點」替換爲「對新函式的呼叫」。新函式可被其他函式使用。

```
   double basePrice = _quantity * _itemPrice;
   if (basePrice > 1000)
      return basePrice * 0.95;
   else
      return basePrice * 0.98;
```

```
   if (basePrice() > 1000)
      return basePrice() * 0.95;
   else
      return basePrice() * 0.98;

...
double basePrice() {
   return _quantity * _itemPrice;
}
```

動機（Motivation）

暫時變數的問題在於：它們是暫時的，而且只能在所屬函式內使用。由於暫時變數只有在所屬函式內才可見，所以它們會驅使你寫出更長的函式，因爲只有這樣你才能存取到想要存取的暫時變數。如果把暫時變數替換爲一個查詢式（query method），那麼同一個 class 中的所有函式都將可以獲得這份資訊。這將帶給你極大幫助，使你能夠爲這個 class 編寫更清晰的程式碼。

Replace Temp with Query（120）往往是你運用 *Extract Method*（110）之前必不可少的一個步驟。區域變數會使程式碼難以被提煉，所以你應該儘可能把它們替換爲查詢式。

這個重構手法較爲直率的情況就是：暫時變數只被賦值一次，或者賦值給暫時變數的運算式不受其他條件影響。其他情況比較棘手，但也有可能發生。你可能需要先運用 *Split Temporary Variable*（128）或 *Separate Query from Modifier*（279）使情況變得簡單一些，然後再替換暫時變數。如果你想替換的暫時變數是用來收

集結果的（例如迴圈中的累加值），你就需要將某些程式邏輯（例如迴圈）拷貝到查詢式（query method）去。

作法（Mechanics）

首先是簡單情況：

- 找出只被賦值一次的暫時變數。
  - ⇨ 如果某個暫時變數被賦值超過一次，考慮使用 *Split Temporary Variable*（128）將它分割成多個變數。
- 將該暫時變數宣告爲 final。
- 編譯。
  - ⇨ 這可確保該暫時變數的確只被賦值一次。
- 將「對該暫時變數賦值」之述句的等號右側部分提煉到一個獨立函式中。
  - ⇨ 首先將函式宣告爲 private。日後你可能會發現有更多 class 需要使用它，彼時你可輕易放鬆對它的保護。
  - ⇨ 確保提煉出來的函式無任何連帶影響（副作用），也就是說該函式並不修改任何物件內容。如果它有連帶影響，就對它進行 *Separate Query from Modifier*（279）。
- 編譯，測試。
- 在該暫時變數身上實施 *Inline Temp*（119）。

我們常常使用暫時變數保存迴圈中的累加資訊。在這種情況下，整個迴圈都可以被提煉爲一個獨立函式，這也使原本的函式可以少掉幾行擾人的迴圈碼。有時候，你可能會用單一迴圈累加好幾個值，就像本書 p.26 的例子那樣。這種情況下你應該針對每個累加值重複一遍迴圈，這樣就可以將所有暫時變數都替換爲查詢式（query）。當然，迴圈應該很簡單，複製這些程式碼時才不會帶來危險。

運用此手法，你可能會擔心效率問題。和其他效率問題一樣，我們現在不管它，因爲它十有八九根本不會造成任何影響。如果效率真的出了問題，你也可以在最佳化時期解決它。如果程式碼組織良好，那麼你往往能夠發現更有效的最佳化方案；如果你沒有進行重構，好的最佳化方案就可能與你失之交臂。如果效率實在太糟糕，要把暫時變數放回去也是很容易的。

範例（Example）

首先，我從一個簡單函式開始：

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

我希望將兩個暫時變數都替換掉。當然，每次一個。

儘管這裡的程式碼十分清楚，我還是先把暫時變數宣告爲 final，檢查它們是否的確只被賦值一次：

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

這麼一來，如果有任何問題，編譯器就會警告我。之所以先做這件事，因爲如果暫時變數不只被賦值一次，我就不該進行這項重構。接下來我開始替換暫時變數，每次一個。首先我把賦值（assignment）動作的右側運算式提煉出來：

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
private int basePrice() {
    return _quantity * _itemPrice;
}
```

編譯並測試，然後開始使用 *Inline Temp*（119）。首先把暫時變數 `basePrice` 的第一個引用點替換掉：

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

編譯、測試、下一個（聽起來像在指揮人們跳鄉村舞蹈一樣）。由於「下一個」已經是 basePrice 的最後一個引用點，所以我把 basePrice 暫時變數的宣告式一併摘除：

```
double getPrice() {
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice() * discountFactor;
}
```

搞定 basePrice 之後，我再以類似辦法提煉出一個 discountFactor()：

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}
private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

你看，如果我沒有把暫時變數 basePrice 替換爲一個查詢式，將多麼難以提煉 discountFactor()！

最終，getPrice()變成了這樣：

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

6.5 Introduce Explaining Variable

你有一個複雜的運算式。

將該複雜運算式（或其中一部分）的結果放進一個暫時變數，以此變數名稱來解釋運算式用途。

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialized() && resize > 0 )
{
   // do something
}
```

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
   // do something
}
```

動機（Motivation）

運算式有可能非常複雜而難以閱讀。這種情況下，暫時變數可以幫助你將運算式分解爲比較容易管理的形式。

在條件邏輯（conditional logic）中，*Introduce Explaining Variable*（124）特別有價值：你可以用這項重構將每個條件子句提煉出來，以一個良好命名的暫時變數來解釋對應條件子句的意義。使用這項重構的另一種情況是，在較長演算法中，可以運用暫時變數來解釋每一步運算的意義。

Introduce Explaining Variable（124）是一個很常見的重構手法，但我得承認，我並不常用它。我幾乎總是儘量使用 *Extract Method*（110）來解釋一段程式碼的意義。畢竟暫時變數只在它所處的那個函式中才有意義，侷限性較大，函式則可以在物件的整個生命中都有用，並且可被其他物件使用。但有時候，當區域變數使 *Extract Method*（110）難以進行時，我就使用 *Introduce Explaining Variable*（124）。

作法（Mechanics）

- 宣告一個 final 暫時變數，將待分解之複雜運算式中的一部分動作的運算結果賦值給它。
- 將運算式中的「運算結果」這一部分，替換爲上述暫時變數。
  - ⇨ 如果被替換的這一部分在程式碼中重複出現，你可以每次一個，逐一替換。
- 編譯，測試。
- 重複上述過程，處理運算式的其他部分。

範例（Examples）

我們從一個簡單計算開始：

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

這段程式碼還算簡單，不過我可以讓它變得更容易理解。首先我發現，底價（base price）等於數量（quantity）乘以單價（item price）。於是我把這一部分計算的結果放進一個暫時變數中：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

稍後也用上了「數量乘以單價」運算結果，所以我同樣將它替換爲 `basePrice` 暫時變數：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(basePrice * 0.1, 100.0);
}
```

然後，我將批發折扣（quantity discount）的計算提煉出來，將結果賦予暫時變數 quantityDiscount：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;
    return basePrice - quantityDiscount +
         Math.min(basePrice * 0.1, 100.0);
}
```

最後，我再把運費（shipping）計算提煉出來，將運算結果賦予暫時變數 shipping。同時我還可以刪掉程式碼中的注釋，因爲現在程式碼已經可以完美表達自己的意義了：

```
double price() {
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;
    final double shipping = Math.min(basePrice * 0.1, 100.0);
    return basePrice - quantityDiscount + shipping;
}
```

運用 Extract Method 處理上述範例

面對上述程式碼，我通常不會以暫時變數來解釋其動作意圖，我更喜歡使用 *Extract Method*（110）。讓我們回到起點：

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

這一次我把底價計算提煉到一個獨立函式中：

```
double price() {
    // price is base price - quantity discount + shipping
    return basePrice() -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(basePrice() * 0.1, 100.0);
}
private double basePrice() {
    return _quantity * _itemPrice;
}
```

我繼續我的提煉，每次提煉出一個新函式。最後得到下列程式碼：

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}
private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}
private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}
private double basePrice() {
    return _quantity * _itemPrice;
}
```

我比較喜歡使用 *Extract Method*（110），因爲同一物件中的任何部分，都可以根據自己的需要去取用這些提煉出來的函式。一開始我會把這些新函式宣告爲 private；如果其他物件也需要它們，我可以輕易釋放這些函式的存取限制。我還發現，*Extract Method*（110）的工作量通常並不比 *Introduce Explaining Variable*（124）來得大。

那麼，應該在什麼時候使用 *Introduce Explaining Variable*（124）呢？答案是：在 *Extract Method*（110）需要花費更大工作量時。如果我要處理的是一個擁有大量區域變數的演算法，那麼使用 *Extract Method*（110）絕非易事。這種情況下我會使用 *Introduce Explaining Variable*（124）幫助我理清程式碼，然後再考慮下一步該怎麼辦。搞清楚程式碼邏輯之後，我總是可以運用 *Replace Temp with Query*（120）把被我引入的那些解釋性暫時變數去掉。況且，如果我最終使用 *Replace Method with Method Object*（135），那麼被我引入的那些解釋性暫時變數也有其價值。

6.6 Split Temporary Variable（剖解暫時變數）

你的程式有某個暫時變數被賦值超過一次，它既不是迴圈變數，也不是一個集用暫時變數（collecting temporary variable）。

針對每次賦值，創造一個獨立的、對應的暫時變數。

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

動機（Motivation）

暫時變數有各種不同用途，其中某些用途會很自然地導致暫時變數被多次賦值。「迴圈變數」和「集用暫時變數」就是兩個典型例子：迴圈變數（loop variable）[Beck] 會隨迴圈的每次執行而改變（例如 `for (int i=0; i<10; i++)`述句中的 `i`）；集用暫時變數（collecting temporary variable）[Beck] 負責將「經由整個函式的運算」而構成的某個值收集起來。

除了這兩種情況，還有很多暫時變數用於保存一段冗長程式碼的運算結果，以便稍後使用。這種暫時變數應該只被賦值一次。如果它們被賦值超過一次，就意味它們在函式中承擔了一個以上的責任。如果暫時變數承擔多個責任，它就應該被替換（剖解）爲多個暫時變數，每個變數只承擔一個責任。同一個暫時變數承擔兩件不同的事情，會令程式碼閱讀者糊塗。

作法（Mechanics）

- 在「待剖解」之暫時變數的宣告式及其第一次被賦值處，修改其名稱。
  - ⇨ 如果稍後之賦值述句是「`i = i +` 某運算式」這種形式，就意味這是個集用暫時變數，那麼就不要剖解它。集用暫時變數的作用通常是累加、字串接合、寫入 stream、或者向群集（collection）添加元素。

- 將新的暫時變數宣告爲 final。
- 以該暫時變數之第二次賦值動作爲界，修改此前對該暫時變數的所有引用點，讓它們引用新的暫時變數。
- 在第二次賦值處，重新宣告原先那個暫時變數。
- 編譯，測試。
- 逐次重複上述過程。每次都在宣告處對暫時變數易名，並修改下次賦值之前的引用點。

範例（Examples）

下面範例中我要計算一個蘇格蘭布丁（haggis）運動的距離。在起點處，靜止的蘇格蘭布丁會受到一個初始力的作用而開始運動。一段時間後，第二個力作用於布丁，讓它再次加速。根據牛頓第二定律，我可以這樣計算布丁運動的距離：

```
double getDistanceTravelled (int time) {
    double result;
    double acc = _primaryForce / _mass;        // 譯註：第一次賦值處
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = acc * _delay;   // 譯註：以下是第二次賦值處
        acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * acc * secondaryTime
                * secondaryTime;
    }
    return result;
}
```

真是個絕佳的醜陋小東西。注意觀察此例中的 `acc` 變數如何被賦值兩次。`acc` 變數有兩個責任：第一是保存第一個力造成的初始加速度；第二是保存兩個力共同造成的加速度。這就是我想要剖解的東西。

首先，我在函式開始處修改這個暫時變數的名稱，並將新的暫時變數宣告爲 final。接下來我把第二次賦值之前對 `acc` 變數的所有引用點，全部改用新的暫時變數。最後，我在第二次賦值處重新宣告 `acc` 變數：

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
```

```
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        double acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * acc * secondaryTime
                * secondaryTime;
    }
    return result;
}
```

新的暫時變數的名稱指出，它只承擔原先 acc 變數的第一個責任。我將它宣告爲 final，確保它只被賦值一次。然後，我在原先 acc 變數第二次被賦值處重新宣告 acc。現在，重新編譯並測試，一切都應該沒有問題。

然後，我繼續處理 acc 暫時變數的第二次賦值。這次我把原先的暫時變數完全刪掉，代之以一個新的暫時變數。新變數的名稱指出，它只承擔原先 acc 變數的第二個責任：

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        final double secondaryAcc = (_primaryForce + _secondaryForce)
        / _mass;
        result += primaryVel * secondaryTime + 0.5 *
        secondaryAcc * secondaryTime * secondaryTime;
    }
    return result;
}
```

現在，這段程式碼肯定可以讓你想起更多其他重構手法。盡情享受吧。（我敢保證，這比吃蘇格蘭布丁强多了 — 你知道他們都在裏面放了些什麼東西嗎？[4]）

[4] 譯注：蘇格蘭布丁（haggis）是一種蘇格蘭菜，把羊心等內臟裝在羊胃裏煮成。由於它被羊胃包成一個球體，因此可以像球一樣踢來踢去，這就是本例的由來。「把羊心裝在羊胃裏煮成…」，呃，有些人難免對這道菜噁心，Martin Fowler 想必是其中之一。

6.7 Remove Assignments to Parameters

你的程式碼對一個參數進行賦值動作。

以一個暫時變數取代該參數的位置。

```
int discount (int inputVal, int quantity, int yearToDate) {
   if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
   int result = inputVal;
   if (inputVal > 50) result -= 2;
```

動機（Motivation）

首先，我要確定大家都清楚「對參數賦值」這個說法的意思。如果你把一個名爲 `foo` 的物件作爲參數傳給某個函式，那麼「對參數賦值」意味改變 `foo`，使它引用（參考、指涉、指向）另一個物件。如果你在「被傳入物件」身上進行什麼操作，那沒問題，我也總是這樣幹。我只針對「`foo` 被改而指向（引用）完全不同的另一個物件」這種情況來討論：

```
void aMethod(Object foo) {
   foo.modifyInSomeWay(); // that's OK
   foo = anotherObject;   // trouble and despair will follow you
```

我之所以不喜歡這樣的作法，因爲它降低了程式碼的清晰度，而且混淆了 *pass by value*（傳值）和 *pass by reference*（傳址）這兩種參數傳遞方式。Java 只採用 *pass by value* 傳遞方式（稍後討論），我們的討論也正是基於這一點。

在 *pass by value* 情況下，對參數的任何修改，都不會對呼叫端造成任何影響。那些用過 *pass by reference* 的人可能會在這一點上犯糊塗。

另一個讓人糊塗的地方是函式本體內。如果你只以參數表示「被傳遞進來的東西」，那麼程式碼會清晰得多，因爲這種用法在所有語言中都表現出相同語意。

在 Java 中，不要對參數賦值；如果你看到手上的程式碼已經這樣做了，請使用 *Remove Assignments to Parameters*（131）。

當然，面對那些使用「輸出式參數」（output parameters）的語言，你不必遵循這條規則。不過在那些語言中我會儘量少用輸出式參數。

作法（Mechanics）

- 建立一個暫時變數，把待處理的參數值賦予它。
- 以「對參數的賦值動作」爲界，將其後所有對此參數的引用點，全部替換爲「對此暫時變數的引用動作」。
- 修改賦值述句，使其改爲對新建之暫時變數賦值。
- 編譯，測試。

  ⇨ 如果程式碼的語意是 *pass by reference*，請在呼叫端檢查呼叫後是否還使用了這個參數。也要檢查有多少個 *pass by reference* 參數「被賦值後又被使用」。請儘量只以 return 方式傳回一個值。如果需要回返的值不只一個，看看可否把需回返的大堆資料變成單一物件，或乾脆爲每個回返值設計對應的一個獨立函式。

範例（Examples）

我從下列這段簡單程式碼開始：

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

以暫時變數取代對參數的賦值動作，得到下列程式碼：

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

還可以爲參數加上關鍵字 `final`，從而强制它遵循「不對參數賦值」這一慣例：

```
int discount (final int inputVal, final int quantity,
              final int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

不過我得承認，我並不經常使用 final 來修飾參數，因爲我發現，對於提高短函式的清晰度，這個辦法並無太大幫助。我通常會在較長的函式中使用它，讓它幫助我檢查參數是否被做了修改。

Java 的 *pass by Value*（傳值）

Java 使用 "*pass by value*"「函式呼叫」方式，這常常造成許多人迷惑。在所有地點，Java 都嚴格採用 *pass by value*，所以下列程式：

```
class Param {
    public static void main(String[] args) {
        int x = 5;
        triple(x);
        System.out.println ("x after triple: " + x);
    }
    private static void triple(int arg) {
        arg = arg * 3;
        System.out.println ("arg in triple: " + arg);
    }
}
```

會產生這樣的輸出：

```
arg in triple: 15
x after triple: 5
```

這段程式碼還不至於讓人糊塗。但如果參數中傳遞的是物件，就可能把人弄迷糊了。如果我在程式中以 Date 物件表示日期，那麼下列程式：

```
class Param {
    public static void main(String[] args) {
        Date d1 = new Date ("1 Apr 98");
        nextDateUpdate(d1);
        System.out.println ("d1 after nextDay: " + d1);

        Date d2 = new Date ("1 Apr 98");
        nextDateReplace(d2);
        System.out.println ("d2 after nextDay: " + d2);
    }
    private static void nextDateUpdate (Date arg) {
        arg.setDate(arg.getDate() + 1);
        System.out.println ("arg in nextDay: " + arg);
    }
    private static void nextDateReplace (Date arg) {
        arg = new Date (arg.getYear(),arg.getMonth(),arg.getDate()+1);
        System.out.println ("arg in nextDay: " + arg);
    }
}
```

產生的輸出是：

```
arg in nextDay: Thu Apr 02 00:00:00 EST 1998
d1 after nextDay: Thu Apr 02 00:00:00 EST 1998
arg in nextDay: Thu Apr 02 00:00:00 EST 1998
d2 after nextDay: Wed Apr 01 00:00:00 EST 1998
```

從本質上說，object reference 是按值傳遞的（*pass by value*）。因此我可以修改參數物件的內部狀態，但對參數物件重新賦值，沒有意義。

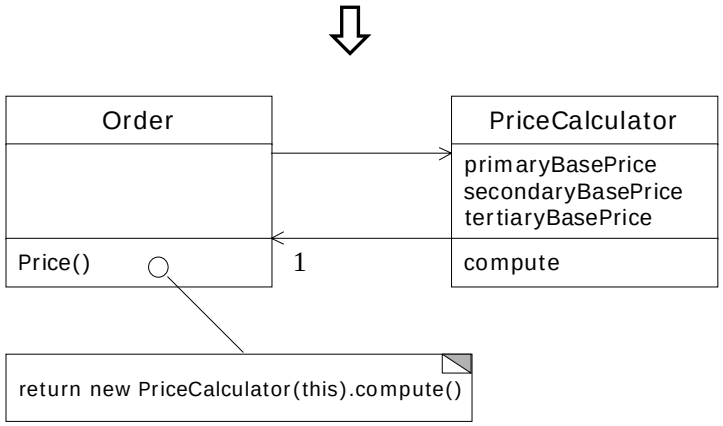
Java 1.1 及其後版本，允許你將參數標示爲 `final`，從而避免函式中對參數賦值。即使某個參數被標示爲 final，你仍然可以修改它所指向的物件。我總是把參數視爲 final，但是我得承認，我很少在參數列（parameter list）中這樣標示它們。

6.8 Replace Method with Method Object

你有一個大型函式，其中對區域變數的使用，使你無法採行 *Extract Method*（110）。

將這個函式放進一個單獨物件中，如此一來區域變數就成了物件內的欄位（field）。然後你可以在同一個物件中將這個大型函式分解爲數個小型函式。

```
class Order...
  double price() {
      double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
      // long computation;
      ...
  }
```



動機（Motivation）

我在本書中不斷向讀者强調小型函式的優美動人。只要將相對獨立的程式碼從大型函式中提煉出來，就可以大大提高程式碼的可讀性。

但是，區域變數的存在會增加函式分解難度。如果一個函式之中區域變數泛濫成災，那麼想分解這個函式是非常困難的。*Replace Temp with Query*（120）可以助你減輕這一負擔，但有時候你會發現根本無法拆解一個需要拆解的函式。這種情況下，你應該把手深深地伸入你的工具箱（好酒沉甕底呢），祭出函式物件（method object）[Beck] 這件法寶。

Replace Method with Method Object（135）會將所有區域變數都變成函式物件（method object）的欄位（field）。然後你就可以對這個新物件使用 *Extract Method*（110）創造出新函式，從而將原本的大型函式拆解變短。

作法（Mechanics）

我厚著臉皮從 Kent Beck [Beck] 那裏偷來了下列作法：

- 建立一個新 class，根據「待被處理之函式」的用途，爲這個 class 命名。
- 在新 class 中建立一個 final 欄位，用以保存原先大型函式所駐物件。我們將這個欄位稱爲「源物件」。同時，針對原（舊）函式的每個暫時變數和每個參數，在新 class 中建立一個個對應的欄位保存之。
- 在新 class 中建立一個建構式（constructor），接收源物件及原函式的所有參數作爲參數。
- 在新 class 中建立一個 `compute()`函式。
- 將原（舊）函式的程式碼拷貝到 `compute()`函式中。如果需要呼叫源物件的任何函式，請以「源物件」欄位喚起。
- 編譯。
- 將舊函式的函式本體替換爲這樣一條述句：「創建上述新 class 的一個新物件，而後呼叫其中的 `compute()`函式」。

現在進行到很有趣的部分了。由於所有區域變數現在都成了欄位，所以你可以任意分解這個大型函式，不必傳遞任何參數。

範例（Examples）

如果要給這一重構手法找個合適例子，需要很長的篇幅。所以我以一個不需要長篇幅（那也就是說可能不十分完美）的例子展示這項重構。請不要問這個函式的邏輯是什麼，這完全是我且戰且走的產品。

```
class Account...
  int gamma (int inputVal, int quantity, int yearToDate) {
      int importantValue1 = (inputVal * quantity) + delta();
      int importantValue2 = (inputVal * yearToDate) + 100;
      if ((yearToDate - importantValue1) > 100)
          importantValue2 -= 20;
      int importantValue3 = importantValue2 * 7;
      // and so on.
      return importantValue3 - 2 * importantValue1;
  }
```

爲了把這個函式變成一個函式物件（method object），我首先需要宣告一個新 class。在此新 class 中我應該提供一個 final 欄位用以保存原先物件（源物件）；對於函式的每一個參數和每一個暫時變數，也以一個個欄位逐一保存。

```
class Gamma...
  private final Account _account;
  private int inputVal;
  private int quantity;
  private int yearToDate;
  private int importantValue1;
  private int importantValue2;
  private int importantValue3;
```

按慣例，我通常會以下劃綫作爲欄位名稱的字首。但爲了保持小步前進，我暫時先保留這些欄位的原名。

接下來，加入一個建構式：

```
Gamma (Account source, int inputValArg, int quantityArg,
       int yearToDateArg) {
    _account = source;
    inputVal = inputValArg;
    quantity = quantityArg;
    yearToDate = yearToDateArg;
}
```

現在可以把原本的函式搬到 `compute()`了。函式中任何呼叫 `Account` class 的地方，我都必須改而使用 `_account` 欄位：

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100)
       importantValue2 -= 20;
    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
```

然後，我修改舊函式，讓它將它的工作轉發（委託，delegate）給剛完成的這個函式物件（method object）：

```
int gamma (int inputVal, int quantity, int yearToDate) {
    return new Gamma(this, inputVal, quantity, yearToDate).compute();
}
```

這就是本項重構的基本原則。它帶來的好處是：現在我可以輕鬆地對 `compute()` 函式採取 *Extract Method*（110），不必擔心引數（argument）傳遞。

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    importantThing();
    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
void importantThing() {
    if ((yearToDate - importantValue1) > 100)
    importantValue2 -= 20;
}
```

6.9 Substitute Algorithm（替換你的演算法）

你想要把某個演算法替換爲另一個更清晰的演算法。

將函式本體（method body）替換爲另一個演算法。

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")) {
            return "Don";
        }
        if (people[i].equals ("John")) {
            return "John";
        }
        if (people[i].equals ("Kent")) {
            return "Kent";
        }
    }
    return "";
}
```

⇩

```
String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[]
                                    {"Don", "John",   "Kent"});
    for (int i=0; i<people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return "";
}
```

動機（Motivation）

我沒試過給猫剝皮，不過我聽說這有好幾種方法，我敢打賭其中某些方法會比另一些簡單。演算法也是如此。如果你發現做一件事可以有更清晰的方式，就應該以較清晰的方式取代複雜方式。「重構」可以把一些複雜東西分解爲較簡單的小塊，但有時你就是必須壯士斷腕，刪掉整個演算法，代之以較簡單的演算法。隨著對問題有了更多理解，你往往會發現，在你的原先作法之外，有更簡單的解決方案，此時你就需要改變原先的演算法。如果你開始使用程式庫，而其中提供的某些功能/特性與你自己的程式碼重複，那麼你也需要改變原先的演算法。

有時候你會想要修改原先的演算法，讓它去做一件與原先動作略有差異的事。這時候你也可以先把原先的演算法替換爲一個較易修改的演算法，這樣後續的修改會輕鬆許多。

使用這項重構手法之前，請先確定自己已經儘可能分解了原先函式。替換一個巨大而複雜的演算法是非常困難的，只有先將它分解爲較簡單的小型函式，然後你才能很有把握地進行演算法替換工作。

作法（Mechanics）

- 準備好你的另一個（替換用）演算法，讓它通過編譯。
- 針對現有測試，執行上述的新演算法。如果結果與原本結果相同，重構結束。
- 如果測試結果不同於原先，在測試和除錯過程中，以舊演算法爲比較參照標準。

  ⇨ 對於每個 test case（測試案例），分別以新舊兩種演算法執行，並觀察兩者結果是否相同。這可以幫助你看到哪一個 test case 出現麻煩，以及出現了怎樣的麻煩。

參考書目

References

[Auer]

Ken. Auer "Reusability through Self-Encapsulation." In *Pattern Languages of Program Design* 1, Coplien J.O. Schmidt.D.C. Reading, Mass.: Addison-Wesley, 1995.

一篇有關於「自我封裝」（self-encapsulation）概念的範式論文（patterns paper）。

[Bäumer and Riehle]

Bäumer, Riehle and Riehle. Dirk "Product Trader." In *Pattern Languages of Program Design 3*, R. Martin F. Buschmann D. Riehle. Reading, Mass.: Addison-Wesley, 1998.

一個範式（patterns），用來靈活創建物件而不需要知道物件隸屬哪個 class。

[Beck]

Kent.Beck *Smalltalk Best Practice Patterns*. Upper Saddle River, N.J.: Prentice Hall, 1997a.

一本適合任何 Smalltalker 的基本書籍，也是一本對任何物件導向開發者很有用的書籍。謠傳有 Java 版本。

[Beck, hanoi]

Kent.Beck "Make it Run, Make it Right: Design Through Refactoring."*The Smalltalk Report, 6*: (1997b): 19-24.

第一本真正領悟「重構過程如何運作」的出版品，也是《*Refactoring*》第一章許多構想的源頭。

[Beck, XP]
Kent.Beck *eXtreme Programming eXplained: Embrace Change. Reading*, Mass.: Addison-Wesley, 2000.

[Fowler, UML]
FowlerM.Scott.K. *UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, Mass.: Addison-Wesley, 2000.

一本簡明扼要的導引，助你了解《*Refactoring*》書中各式各樣的 Unified Modeling Language（UML，統一建模語言）圖片。

[Fowler, AP]
M.Fowler *Analysis Patterns: Reusable Object Models*. Reading, Mass.: Addison-Wesley, 1997.

一本 domain model patterns 專著。包括對 range pattern 的一份討論。

[Gang of Four]
E. Gamma, R. Helm, R. Johnsonand J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

或許是物件導向設計（object-oriented design）領域中最有價值的一本書。現今幾乎任何人都必須語帶智慧地談點 strategy, singleton, 和 chain of responsibility，才敢說自己懂得物件（技術）。

[Jackson, 1993]
Jackson, Michael. *Michael Jackson's Beer Guide*, Mitchell Beazley, 1993.

一本有用的導引，提供大量實踐（實用性）研究。

[Java Spec]
Gosling, James, Bill Joy and Guy Steele. *The Java Language Specification, Second Edition*. Boston, Mass.: Addison-Wesley, 2000.

所有 Java 問題的官方答案。

[JUnit]
Beck, Kent, and Erich Gamma. JUnit Open-Source Testing Framework. Available on the Web (http://www.junit.org).

撰寫 Java 程式的基本應用工具。是個簡單框架（framework），幫助你撰寫、組織、執行單元測試（unit tests）。類似的框架也存在於 Smalltalk 和 C++ 中。

[Lea]

Doug.Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Reading, Mass.: Addison-Wesley, 1997.

編譯器應該阻止任何人實作 `Runnable` 介面 — 如果他沒有讀過這本書。

[McConnell]

Steve. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Redmond, Wash.: Microsoft Press, 1993.

一本對於編程風格和軟體建構的卓越導引。寫於 Java 誕生之前，但幾乎書中的所有忠告都適用於 Java。

[Meyer]

Bertrand. Meyer, *Object Oriented Software Construction*. 2 ed. Upper Saddle River, N.J.: Prentice Hall, 1997.

物件導向設計（object-oriented design）領域中一本很好（也很龐大）的書籍。其中包括對於契約式設計（design by contract）的一份徹底討論。

[Opdyke]

William F. Opdyke, Ph.D. diss., "Refactoring Object-Oriented Frameworks. " University of Illinois at Urbana-Champaign, 1992.

參見 ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z。是關於重構的第一份體面長度的著作。多少帶點教育和工具導向的角度（畢竟這是一篇博士論文），對於想更多了解重構理論的人，是很有價值的讀物。

[Refactoring Browser]

Brant, John, and Don Roberts. Refactoring Browser Tool,

http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser。未來的軟體開發工具。

[Woolf]

Bobby. Woolf, "Null Object." In *Pattern Languages of Program Design 3*, Martin, R. Riehle. D. Buschmann F. Reading, Mass.: Addison-Wesley, 1998.

針對 null object pattern 的一份討論。

原音重現

List of Soundbites

p.7 *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.*

如果你發現自己需要爲程式添加一個特性，而程式碼結構使你無法很方便地那麼做，那就先重構那個程式，使特性的添加比較容易進行，然後再添加特性。

p.8 *Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking.*

重構前，先檢查自己是否有一套可靠的測試機制。這些測試必須有自我檢驗能力。

p.13 *Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.*

重構技術係以微小的步伐修改程式。如果你犯下錯誤，很容易便可發現它。

p.15 *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

任何一個傻瓜都能寫出計算機可以理解的程式碼。惟有寫出人類容易理解的程式碼，才是優秀的程式員。

p.53 ***Refactoring*** *(noun) : a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior of the software.*

重構（名詞）：對軟體內部結構的一種調整，目的是在不改變「軟體之可察行爲」前提下，提高其可理解性，降低其修改成本。

p.54 ***Refactor*** *(verb) : to restructure software by applying a series of refactorings without changing the observable behavior of the software.*

重構（動詞）：使用一系列重構準則（手法），在不改變「軟體之可察行爲」前提下，調整其結構。

p.58 *Three strikes and you refactor.*

事不過三，三則重構。

p.65 *Don't publish interfaces prematurely. Modify your code ownership policies to smooth refactoring.*

不要過早發佈介面。請修改你的程式碼擁有權政策，使重構更順暢。

p.88 *When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.*

當你感覺需要撰寫註釋，請先嘗試重構，試著讓所有註釋都變得多餘。

p.90 *Make sure all tests are fully automatic and that they check their own results.*

確保所有測試都完全自動化，讓它們檢查自己的測試結果。

p.90 *A suite of tests is a powerful bug detector that decapitates the time it takes to find bugs.*

一整組測試就是一個強大的臭蟲偵測器，能夠大大縮減搜尋臭蟲所需要的時間。

p.94 *Run your tests frequently. Localize tests whenever you compile - every test at least every day.*

頻繁地執行測試。每次編譯請把測試也考慮進去 — 每天至少執行每個測試一次。

p.97 *When you get a bug report, start by writing a unit test that exposes the bug.*

每當你接獲臭蟲提報，請先撰寫一個單元測試來揭發這隻臭蟲。

p.98 *It is better to write and run incomplete tests than not to run complete tests.*

編寫未臻完善的測試並實際執行，好過對完美測試的無盡等待。

p.99 *Think of the boundary conditions under which things might go wrong and concentrate your tests there.*

考慮可能出錯的邊界條件，把測試火力集中在那兒。

p.100 *Don't forget to test that exceptions are raised when things are expected to go wrong.*

當事情被大家認爲應該會出錯時，別忘了檢查彼時是否有異常被如期拋出。

p.101 *Don't let the fear that testing can't catch all bugs stop you from writing the tests that will catch most bugs.*

不要因爲「測試無法捕捉所有臭蟲」，就不撰寫測試碼，因爲測試的確可以捕捉到大多數臭蟲。

索引

D

E

F

G

H

I

J

N

O

P

Q

R

S

T

U

壞味道（smell）	常用的重構手法（Common Refactoring）
Alternative Classes with Different Interfaces, p85	*Rename Method* (273), *Move Method* (142)
Comments, p87	*Extract Method* (110), *Introduce Assertion* (267)
Data Class, p86	*Move Method* (142), *Encapsulate Field* (206), *Encapsulate Collection* (208)
Data Clumps, p81	*Extract Class* (149), *Introduce Parameter Object* (295), *Preserve Whole Object* (288)
Divergent Change, p79	*Extract Class* (149)
Duplicated Code, p76	*Extract Method* (110), *Extract Class* (149), *Pull Up Method* (322), *Form Template Method* (345)
Feature Envy, p80	*Move Method* (142), *Move Field* (146), *Extract Method* (110)
Inappropriate Intimacy, p85	*Move Method* (142), *Move Field* (146), *Change Bidirectional Association to Unidirectional* (200), *Replace Inheritance with Delegation* (352), *Hide Delegate* (157)
Incomplete Library Class, p86	*Introduce Foreign Method* (162), *Introduce Local Extension* (164)
Large Class, p78	*Extract Class* (149), *Extract Subclass* (330), *Extract Interface* (341), *Replace Data Value with Object* (175)
Lazy Class, p.83	*Inline Class* (154), *Collapse Hierarchy* (344)
Long Method, p76	*Extract Method* (110), *Replace Temp With Query* (120), *Replace Method with Method Object* (135), *Decompose Conditional* (238)

※譯註：各種壞味道（smell）和各種重構手法（refactorings）之中文譯名見目錄及正文

壞味道（smell）	常用的重構手法（Common Refactoring）
Long Parameter List, p78	*Replace Parameter with Method* (292), *Introduce Parameter Object* (295), *Preserve Whole Object* (288)
Message Chains, p84	*Hide Delegate* (157)
Middle Man, p85	*Remove Middle Man* (160), *Inline Method* (117), *Replace Delegation with Inheritance* (355)
Parallel Inheritance Hierarchies, p83	*Move Method* (142), *Move Field* (146)
Primitive Obsession, p81	*Replace Data Value with Object* (175), *Extract Class* (149), *Introduce Parameter Object* (295), *Replace Array with Object* (186), *Replace Type Code with Class* (218), *Replace Type Code with Subclasses* (223), *Replace Type Code with State/Strategy* (227)
Refused Bequest, p87	*Replace Inheritance with Delegation* (352)
Shotgun Surgery, p80	*Move Method* (142), *Move Field* (146), *Inline Class* (154)
Speculative Generality, p83	*Collapse Hierarchy* (344), *Inline Class* (154), *Remove Parameter* (277), *Rename Method* (273)
Switch Statements, p82	*Replace Conditional with Polymorphism* (255), *Replace Type Code with Subclasses* (223), *Replace Type Code with State/Strategy* (227), *Replace Parameter with Explicit Methods* (285), *Introduce Null Object* (260)
Temporary Field, p84	*Extract Class* (149), *Introduce Null Object* (260)

※譯註：各種壞味道（smell）和各種重構手法（refactorings）之中文譯名見目錄及正文