前言

很久以前，有一位顧問親自拜訪一個開發專案，準備查看一些已經寫好的程式。當這位顧問瀏覽系統核心的類別階層時，發現它相當混亂，裡面的高階類別對於底下的類別該如何運作做了一些假設（即，繼承它的程式需要實體化的假設），但並非所有子類別都適合那些假設，因此高階的假設被大量覆寫，其實那些超類別只要稍做修改可以大幅降低覆寫的需求了。不僅如此，有些超類別的目的沒有被清楚地表達，並且有重覆的行為。在其他地方，有一些子類別負責相同的工作，顯然那些程式碼可以搬到繼承階層的上方。

這位顧問建議專案主管重新審查程式碼並加以清理。但是專案主管意興闌珊，因為程式看起來可以動作，而且他有很大的時間壓力，所以隨口敷衍一下，說他們會再找時間處理那些問題。

顧問也讓負責編寫繼承階層的程式員知道問題，程式員有很大的興趣，也確實看到問題。他們知道有問題不是他們的錯，因為旁觀者清，有些問題需要依靠別人才能發現。那些程式員花了一到兩天整理類別階層，完成工作之後，他們移除類別階層一半的程式，但沒有破壞它的功能。他們對結果非常滿意，發現自己更容易加入新類別，也更容易在系統的其他地方使用那些類別。

但是專案主管不太高興，因為時間非常緊迫，還有許多工作要做。他認為這兩位程式員花了兩天的時間，卻沒有幫幾個月之內就要送出的系統加入任何新功能。原本的舊程式已經可以動作了啊！的確，程式碼的設計更“純粹”且更“簡潔”了，但是專案的目的，是送出可動作的程式，而不是寫出取悅學者的程式。顧問也建議對系統的另一個核心部分做類似的清理，但是這項工作可能

會讓專案延遲一兩週的時間。這些工作的本意都是為了讓程式碼看起來更漂亮，而不是讓程式做原本無法做的事情。

看完這個故事，你有什麼感想？你認為顧問建議做進一步的整理是對的嗎？或者，你認為“如果它可以動作，就不要修復它”這句古老的工程格言是對的？

我必須承認我帶著主觀意識，因為我就是那位顧問。那項專案在六個月之後失敗了，很大程度上是因為程式太複雜而難以除錯，以及難以將性能調整為可接受的程度。

他們邀請 Kent Beck 擔任顧問，重新啟動這項專案，幾乎從頭重寫整個系統。Kent Beck 執行許多不同的策略，其中一項重要的變革就是堅持使用重構來清理程式碼，這個做法改善了團隊的效率。重構在過程中發揮的效果促使我寫出本書的第一版，因為我想要將 Kent 與其他人使用重構來提升軟體品質的經驗傳遞出去。

從那時候開始，重構就成為一種廣受採納的程式設計術語了，原著也大獲好評。但是對一本程式設計書籍來說，八歲是很老的年齡，所以我認為是時候重寫這本書了。我幾乎重寫了書中的每一頁，不過在某種意義上，它的變化沒有那麼大，因為重構的本質是相同的，基本上，大部分的關鍵重構手法都保持不變。但是我希望這本重寫的書可以幫助更多人知道如何有效地進行重構。

什麼是重構？

重構是一種修改軟體系統的程序，這個程序不改變程式碼的外部行為，而是改善它的內部結構。重構是有紀律地清理程式碼，盡量降低 bug 發生機率的方法。基本上，重構就是改善已經寫好的設計。

“改善已經寫好的設計”這句話有點奇怪，因為在軟體開發的歷史中，大多數人都認為應該先完成設計，再開始撰寫程式，但是採取這種做法的話，隨著時

間的過去，程式會不斷被修改，且系統的完整性（根據設計建立的架構）會逐漸褪去，編寫程式的方法會慢慢地從工程方法淪為隨意變動。

重構是相反的做法。我們可以藉由重構取出不良的（甚至混亂的）設計，將它改成良好的結構。重構的每一個步驟都很簡單，甚至簡單得讓人覺得太過份了。在重構時，我會將某個類別的欄位（field）移到另一個類別，從某個方法拉出一些程式碼放入它們自己的方法，或將一些程式碼往類別階層的上方或下方移動，但是累積這些小改變可以徹底改善設計，這個概念正好與軟體的劣化過程完全相反。

重構改變了工作的平衡，我發現重構可讓人們在開發的過程中持續進行設計，而非只在開發初期設計。我可以一邊建構系統，一邊瞭解如何改善設計，這種互動可以讓程式碼隨著開發的進行維持良好的設計。

本書內容

這本書是重構指南，它是為專業程式員寫的。我的目的是告訴你如何以可控制且高效率的方式進行重構，你可以從中學到如何避免在重構時引入 bug，同時有系統地改善結構。

通常一本書都是用簡介來開頭的，原則上我認同這種做法。但是我發現這種做法讓我很難透過廣義的討論或定義來介紹重構，所以我選擇用範例來開場。第 1 章會列出一段小程式，它裡面有一些常見的設計缺陷，我會一邊將它重構成更容易瞭解與修改的程式，一邊告訴你重構的程序以及一些實用的重構手法。如果你想要真正瞭解重構，這是非常重要的一章。

第 2 章介紹重構的一般原則、一些定義，以及重構的理由，我會列舉一些重構的挑戰。在第 3 章，Kent Beck 幫我說明如何尋找程式碼異味，以及如何用重構來清除它們。“測試”在重構的過程中扮演非常重要的角色，第 4 章將說明如何在程式裡面編寫測試。

其餘的部分是本書核心—重構名錄。這不是一份盡善盡美的名錄，但它涵蓋了大多數開發者都有可能用到的重構手法。它們來自我在 1990 年代末期學習重構時寫的筆記，因為我沒辦法記住所有的重構方法，所以到現在還會使用這些筆記。當我想要做某項重構，例如 *Split Phase (181)* 時，就會查詢這份名錄，來瞭解如何以安全、逐步的方式完成它。希望這也是你經常翻閱的部分。

本書以 web 為主

全球資訊網已經對這個社會造成巨大的影響了，特別是影響資訊的取得方式。當初我寫這本書時，軟體開發的知識大部分都是用印刷品來傳遞的，但是現在我大都是從網路上收集資訊。這種改變對我這樣的作者構成一項挑戰：書會不會再也不重要了？以及它們該長怎樣？

我相信書籍仍然有它的用處，但它們必須改變。書本的價值在於它可以用連貫的方式把大量的知識彙集起來，在寫這本書時，我試著納入許多不同的重構手法，並且用一致且整合的方式組織它們。

雖然這種經過整合的抽象文字作品在傳統上是以紙本來呈現的，但未來不一定如此。圖書產業大部分的作品依舊以紙本作為主要的媒介，雖然坊間已經有大量的電子書了，但它們只不過是用電子的形式，來呈現原本的紙本結構。

我試著在這本書採取不同的做法，你可以在本書網路或 web 版看到它的典型版本（canonical form）。購買紙本或 ebook 版本即可使用 web 版本（下面會說明如何在 InformIT 註冊你的產品。）。印刷書的素材來自網站，用適合印刷的方式來編排。印刷書不打算列入網站的所有重構法，因為將來我可能還會在典型 web 版本加入更多手法。電子書也是 web 版本的另一種表現形式，它裡面的重構法也有可能與印刷書不同—畢竟，電子書不會因為我加入更多頁數而變得更重，而且它們比較容易在賣出之後更新。

我不知道你究竟是在網路上閱讀 web 版本、在手機閱讀電子書、閱讀紙本，還是用目前我還不知道的形式閱讀，無論你用哪一種方式來吸收它，我都會盡我所能的讓它成為實用的作品。（註：此 web 版為英文版本）

要閱讀典型 web 版本以及取得更新或修正的內容，請在 InformIT 網站註冊你的 *Refactoring*，*Second Edition* 書本。註冊流程是前往 `informit.com/register` 並登入（如果你還沒有帳號，先建立一個），輸入英文版書籍 ISBN 9780134757599 並按下 Submit，接下來你會看到一個問題，所以手邊一定要有實體書或電子書。成功註冊書籍之後，打開 Account 網頁的“Digital Purchases”標籤，並按下這本書下面的“Launch”連結，即可打開 web 版本。

JavaScript 範例

如同軟體開發領域大多數的技術，在講解概念時，範例程式是很重要的工具。不過，各種語言的重構看起來大同小異。雖然有時特定的語言有特別需要注意的事項，但是重構的核心元素都是相同的。

我選擇以 JavaScript 說明重構手法，因為我認為多數人都看得懂這種語言。但是，請勿認為這些重構手法很難運用在你使用的任何語言上面。我會盡量避免使用 JavaScript 比較複雜的功能，所以你只要大略瞭解 JavaScript 就可以學習這些重構法了。我當然不是為了幫 JavaScript 代言才使用它的。

我在範例中使用 JavaScript 並不代表本書介紹的技術只適合 JavaScript。本書的第一版使用 Java，但仍然有許多從未寫過 Java 的程式員覺得那本書很實用。我曾經用 12 種不同的語言來說明技術的普遍性，但是後來發現這種做法反而會讓讀者一頭霧水。本書是寫給任何語言的程式員看的，除了範例的部分之外，我沒有做出任何關於語言的假設。希望讀者可以吸收普遍性的說明，將它們運用在你使用的任何語言上。事實上，我希望讀者看完 JavaScript 範例之後，能夠將這些範例改成自己使用的語言。

也就是說，除了特定的範例之外，當我談到“類別”、“模組”、“函式”等術語時，指的是普遍的程式設計詞彙，而不是 JavaScript 語言模型的特定詞彙。

使用 JavaScript 作為範例語言，代表我會試著避開不常使用 JavaScript 的程式員比較不熟悉的 JavaScript 風格。這不是一本說明“重構 JavaScript 程式

碼”的書籍，而是碰巧使用 JavaScript 的泛用重構書籍。JavaScript 有許多專屬的重構方法（例如將回呼重構、重構成 promise 或 async/await），但本書不討論它們。

誰該閱讀本書？

本書的目標是專業程式員，也就是靠寫程式來討生活的人。書中的範例與說明有許多需要閱讀與瞭解的程式，範例是用 JavaScript 寫成的，但它的概念適合大多數的語言。我希望程式員有足夠的經驗可以欣賞本書的內容，但不需要具備太多知識。

雖然本書的主要對象是想要學習重構的開發者，但它對已經瞭解重構的人來說也很有價值，因為這本書可以當成教材。我在本書費盡心思地解釋各種重構法的作用，所以資深的開發者可以用這些教材來指導同事。

雖然重構的重點是程式碼，但它對系統的設計有很大的影響。各公司應該讓資深的設計師與架構師瞭解重構的原則，並在專案中運用它們。最好的做法是讓德高望重且經驗豐富的開發者介紹重構，因為這種開發者最瞭解重構背後的原理，也能夠根據特定的工作場合調整原則。當你使用 JavaScript 之外的語言時更是需要如此，因為你必須將我提供的範例改成其他語言。

如果你想要取得最大效益，但又不想讀完整本書籍，可以採取下列的做法。

- **如果你想要瞭解什麼是重構**，看第 1 章—你可以從範例清楚看到整個過程。
- **如果你想要瞭解為何需要重構**，讀前兩章，它們將告訴你什麼是重構，以及為何要重構。
- **如果你想要找出需要重構的地方**，讀第 3 章，它會告訴你需要重構的跡象。

- **如果你想要實際進行重構**，看完前四章，接著跳讀名錄。你可以先大致閱讀全部的名錄來瞭解內容，不需要掌握所有細節。當你真的需要執行重構時，再閱讀有關的重構法並使用它。這份名錄是參考章節，所以你應該不會一次看完它們。

在寫這本書時，為各種重構手法命名是非常重要的工作。術語可以促進溝通，使用術語的話，當一位開發者建議另一位開發者將一些程式碼取出，並放入專屬的函式，或將一些計算式拆成各個階段時，他就知道可以參考 *Extract Function (126)* 與 *Split Phase (181)*。這些術語也可以幫助你選擇自動重構工具。

站在巨人的肩上

在本書的開頭，我必須說，這本書讓我虧欠很多人甚多，他們是在 1990 年代開拓重構領域的前輩們。如果沒有從他們的經驗中得到啟發，我就無法寫出本書的第一版，雖然已經過了很多年，我依然要感謝他們奠定的基礎。在理想的情況下，他們之中的某位才是第一版的作者，但最終只有我有時間與精力做這件事。

Ward Cunningham 與 **Kent Beck** 是最早期的重構倡導者，他們很早就把重構當成開發的基礎，並且修改他們的開發程序來利用重構。必須特別強調的是，我就是跟 Kent 一起工作才看到重構的重要性，進而產生著作本書的靈感。

Ralph Johnson 是伊利諾大學厄巴納 - 香檳分校的團隊領導，以其對物件技術的貢獻而聞名。Ralph 是重構界的大師，他的許多學生都在這個領域做了重大的貢獻。**Bill Opdyke** 的博士論文是第一份關於重構的詳細著作。**John Brant** 與 **Don Roberts** 不僅僅是貢獻著作而已，他們也建立了史上第一個自動化重構工具（Refactoring Browser），可用來重構 Smalltalk 程式。

Chapter 1

重構：第一個範例

我該怎麼開始介紹重構呢？傳統的方式是介紹這個主題的歷史、廣義的原則等等，但是如果有人在會議上這樣做的話，我就會開始昏昏欲睡，我的思緒會開始飄移，久久才聽一下講者在說些什麼，直到他舉例說明為止。

範例讓我回神的原因是它可讓我看到事情的來龍去脈。聽到原則時，我們很容易往籠統的方向思考，因此也就難以理解如何運用技術，但是範例可以釐清很多事情。

所以我要在本書的開頭舉一個重構的例子，我會說明重構的作用，並且讓你瞭解重構的過程，在下一章再採取原則式的介紹。

但是我在選擇範例時遇到兩難的情況，如果我挑選大型的程式，介紹它以及講解重構的過程就會變得太複雜，使得一般的讀者無法看完。（我在原書做過這件事，最後捨棄兩個範例，它們其實不大，但每一個都需要用一百多頁來說明。）但是如果選擇大家都可以理解的小程式，重構看起來就沒有價值。

這是每位介紹程式技術的人都遇過的麻煩，坦白說，下面這段小程式並不值得進行即將展示的重構，但是如果這段程式是大型系統的一部分，重構就很重要了。當你閱讀範例時，請想像它屬於一個大很多的系統。

起點

本書第一版的第一個程式可以印出影音出租店的帳單，但是現在的你可能不知道什麼是影音出租店，與其回答這個問題，我將這個例子換成一個歷史悠久，但仍然流行的東西。

假設有一群可在各種活動演出的演員，顧客通常會指定幾部戲，戲劇公司會根據觀眾的規模以及戲劇的類型向顧客收費。他們目前有兩種戲劇：tragedy（悲劇）和 comedy（喜劇）[譯註 1]。除了表演帳單之外，這間公司也會給顧客“volume credits（批量積點）”，讓他們在將來欣賞表演時享受折扣，這是維持顧客忠誠度的機制。

演員將他們的戲劇資料放在一個簡單的 JSON 檔案裡面，檔案長這樣：

plays.json…

```
{
  "hamlet": {"name":"Hamlet", "type": "tragedy"},
  "as-like": {"name":"As You Like It", "type": "comedy"},
  "othello": {"name":"Othello", "type": "tragedy"}
}
```

發票資料也放在 JSON 檔案裡面：

invoices.json…

```
[
  {
    "customer":"BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience":55
      },
      {
        "playID": "as-like",
```

譯註 1　凡是同時在內文與程式碼中出現的英文單字，第一次出現時皆以 tragedy（悲劇）這種形式表示，此後會直接以原文表示，以方便讀者比對參照。

```
      "audience":35
    },
    {
      "playID": "othello",
      "audience":40
    }
  ]
 }
]
```

這個簡單的函式可以印出帳單：

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency:"USD",
                         minimumFractionDigits:2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
          throw new Error(`unknown type: ${play.type}`);
    }

    // 加入 volume credit
    volumeCredits += Math.max(perf.audience - 30, 0);
    // 每十名喜劇觀眾可獲得額外點數
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // 印出這筆訂單
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
```

```
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

用上面的程式來執行測試資料可產生下列輸出：

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

評論開頭的程式

你覺得這段程式的設計怎樣？首先，我想要說的是，它這個樣子是可以容忍的，因為這段程式很短，沒有任何深層結構需要理解。但是前面提過，我使用小範例是逼不得已的，你必須想像這段程式屬於一個更大規模的程式—或許有幾百行之長。在這種規模之下，就連一個內聯（inline）的函式也有可能難以理解。

既然這段程式可以正常運作，我們豈非只是站在審美的角度，指出這段程式“有夠醜”？畢竟編譯器並不在乎程式碼究竟醜陋還是簡潔啊！但是在修改系統時會有人員參與其中，而人類很在乎這件事。設計不良的系統很難修改，因為你很難找出需要修改什麼，以及釐清你修改的地方將如何與既有的程式互動，來產生你期望的行為。當你難以找出需要修改哪些地方時，就有極大的機會犯錯並引入 bug。

因此，當我要修改有上百行程式碼的程式時，我寧可先將它改成一組函式和其他程式元素，以便瞭解程式的來龍去脈。如果程式的結構不良，我通常會先改良程式結構，再進行所需的變動。

這個例子中，我想要做幾項程式的使用者也希望看到的修改。首先，他們希望句子是用 HTML 印出的，請想想這項變動會產生什麼影響。我看過有人在每一個加入字串的程式周圍，都加上條件式來切換格式，這種做法會讓函式複雜許多。在處理這種情況時，很多人喜歡複製原本的方法，再修改副本，讓它發出 HTML。乍看之下，製作副本不是一件麻煩的工作，但是它會埋下將來的禍患。從今以後，每當我們想要修改收費邏輯時，都必須更改這兩個方法，並且確保它們更改的地方是一致的。如果程式碼永遠不會變動，這種複製貼上的做法就沒有任何問題，但是如果它會存留很長的時間，複製就會埋下禍患。

當你想要在程式中加入一項功能，但是那段程式的結構很不方便時，請先重構程式，讓你可以輕鬆地加入功能後，再加入那項功能。

這讓我想到第二項變動。演員們希望演出更多戲劇，他們希望在戲碼中加入歷史劇、田園劇、田園喜劇、歷史田園劇、悲史劇、悲喜歷史田園劇、場景連貫劇、詩曲無限劇，但是他們還沒有決定確切的劇碼，以及什麼時候可以演出。這項修改會影響演出的收費方式與 volume credit 的計算方式。作為一位資深的開發者，我相信無論他們最終的決定是什麼，在六個月之內他們都會再改變。畢竟，增加功能的需求通常都是蜂擁而至，不會只有一次。

同樣的，為了處理分類與收費規則的變動，我們必須修改 `statement` 方法。但是如果我把 `statement` 的內容複製到 `htmlStatement` 裡面，以後就一定要確保任何修改都是同步的，此外，隨著規則越來越複雜，我會越來越難以確定該在哪裡修改，因此就更容易在修改時出錯。

你要知道，正是因為有這種變動，才會有重構的需求。如果程式的功能是正常的，而且永遠都不需要變動，直接把它晾在那裡絕對沒有問題。雖然改善程式碼仍然是件好事，但除非有人真的想要瞭解它，否則程式碼長怎樣都無關緊要。但是，只要有人需要瞭解程式如何運作，而且程式碼令人難以閱讀，你就要做一些事情了。

重構的第一步

每當我進行重構時，都會先採取同樣的步驟，寫出一套測試程式，用來測試被重構的程式。即使我要做的重構引入 bug 的機會很低，但我是個凡人，仍然有機會犯錯，所以這些測試程式非常重要。程式越大，我的變動越有可能在無意間讓某個東西出錯（在數位時代，軟體是脆弱的象徵）。

因為 `statement` 回傳一個字串，我會建立一些 invoice（發票），在每一個 invoice 裡面指定一些戲劇，產生 statement 字串，接著拿新字串與我確認過的字串做比較。我會用測試框架來安排所有的測試，如此一來，我只要在開發環境輕鬆地按下按鍵就可以執行它們了。這些測試程式只需要花費幾秒鐘的時間執行，你將會看到，我經常執行它們。

測試程式回報結果的方式非常重要。它們要嘛是綠色的，代表所有的字串都與參考字串一致，要嘛是紅色的，會顯示一串失敗訊息，也就是不一致的那幾行字串，因此測試是自檢的（self-checking）。自檢非常重要，因為若非如此，我就要花時間親自比對測試程式產生的值與桌墊上的值，因而降低我的速度。現代的測試框架都有編寫與執行自檢測試所需的功能。

在重構之前，請先準備好堅實的測試程式，測試程式必須是自檢的。

我在進行重構時十分依賴測試，我將它們視為防止犯錯的 bug 偵測機制。藉著寫兩次我想要的東西，一次在原本的程式內，一次在測試程式內，除非兩個地方都有錯，否則任何錯誤都無法騙過偵測機制。藉著反復檢查作品，我可以減少犯錯的機會。雖然編寫測試很花時間，但我最終可以省下許多時間，並且獲得大量的回報，因為我會花更少的時

間在除錯上。測試是重構非常重要的部分，所以我會用完整的一章來討論它（*Building Tests (85)*）[譯註 2]。

分解 statement 函式

每當我重構這種冗長的函式時，都會看看有哪些地方可將整體行為拆成不同的部分。我看到的第一個部分是中間的 switch 陳述式。

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency:"USD",
                         minimumFractionDigits:2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // 加入 volume credit
    volumeCredits += Math.max(perf.audience - 30, 0);
```

譯註 2　作者經常在內文引用重構手法，譯者認為使用原文來稱呼重構手法的好處很多，因此只在詳述各種手法的小節的標題列出該手法的中文，其餘皆用原文。

```
    // 每十名喜劇觀眾可獲得額外點數
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // 印出這筆訂單
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

閱讀這段程式之後，我認為它的功能是計算一場表演的費用，這個結論是透過觀察程式碼得出的。但是如同 Ward Cunningham 所言，這種理解是用大腦得出的，但是大腦很容易忘記事情，我必須將大腦裡面的東西寫成程式碼加以保存，如此一來，當我日後回來閱讀這段程式時，就不需要再次搞懂它，程式自己會告訴我它在做些什麼。

為了將理解的結果放入程式碼，我將那段程式擺到它自己的函式裡面，用它做的事情來幫它命名—例如 `amountFor(aPerformance)`。每當我像這樣把一段程式放到函式裡面時，都會做一系列的動作來將犯錯的機會降到最低，所以我把這個過程寫下來，以方便日後參考，並將它稱為 *Extract Function (126)*。

首先，我要找出將這段程式變成它自己的函式之後，就會離開作用域（scope）的變數。這個例子有三個：`perf`、`play` 與 `thisAmount`。這段取出的程式會使用前兩個變數，但不會修改它們，所以我用參數來傳遞它們。我必須小心地處理會被修改的變數，因為這個例子只有一個，所以我可以回傳它。我也可以將它的初始化放到取出的程式裡面。以上的動作產生這個結果：

function statement…

```
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
  case "tragedy":
    thisAmount = 40000;
    if (perf.audience > 30) {
      thisAmount += 1000 * (perf.audience - 30);
    }
```

```
    break;
  case "comedy":
    thisAmount = 30000;
    if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
    }
    thisAmount += 300 * perf.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return thisAmount;
}
```

程式的斜體標題“*function someName…*”代表接下來的程式碼位於標題所指的函式、檔案或類別範圍內。那個範圍裡面通常還有其他程式碼，因為目前還不會討論它們，所以不加以展示。

現在原始的 `statement` 程式會呼叫這個函式來設定 `thisAmount` 的值：

頂層…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency:"USD",
                         minimumFractionDigits:2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

    // 加入 volume credit
    volumeCredits += Math.max(perf.audience - 30, 0);
    // 每十名喜劇觀眾可獲得額外點數
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // 印出這筆訂單
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

完成修改之後，我會立刻編譯並進行測試，看看有沒有任何損壞。無論重構多麼簡單，在每次重構之後進行測試是很重要的習慣。人很容易犯錯—至少我是如此。在每一次修改之後進行測試，代表當我犯錯時，只要在一次小變動裡面找出錯誤就可以了，因此更容易尋找與修復錯誤。重構程序的重點就是進行小幅度的修改，並且在每次修改之後測試。如果我一次做太多事情，犯錯會讓我進入一個棘手的除錯插曲，可能會花很長的時間。小型的變動可產生緊密的回饋迴圈，它是避免這種麻煩的關鍵。

> 我在這裡使用編譯來代表讓 JavaScript 可被執行的動作。JavaScript 是可直接執行的，或許這個動詞對它來說沒有任何意義，但是在其他語言中，“編譯”可能代表將程式移到一個輸出目錄，並且（或）執行 Babel [babel] 之類的處理程式。

重構就是用小步驟修改程式，讓你可以在犯錯時，輕鬆地找到 *bug* 的位置。

這是 JavaScript，所以我可以把 `amountFor` 放在 `statement` 內。這種做法很方便，因為如此一來，我就不需要將外部函式的作用域內的資料傳給新函式了。在這個例子中，是否採取這種做法沒有任何差異，但可減少一個需要處理的問題。

這個例子的測試通過了，我的下一個步驟是將這次變動上傳到本地版本控制系統。我使用 git 或 mercurial 等版本控制系統來進行私人提交。我會在成功重構之後進行提交，因為如此一來，以後出錯時就可以輕鬆地回到上一個正常的狀態。接著我會將修改結果壓縮（squash）成比較有意義的版本，再將它送到共用的存放區。

Extract Function (126) 是經常被自動化的重構。當我用 Java 寫程式時，都會本能地使用一系列的 IDE 按鍵來執行這項重構。在我寫這本書時，JavaScript 的工具還沒有提供這種強大的重構支援，所以我必須親自處理它。它不難，只是我必須小心地處理這些區域範圍變數。