前言

您是否有曾經讀過某本與軟體相關的書，並覺得作者寫的內容對您來說太過高深了？書中內容是否使用了很多不熟悉的詞彙和過於複雜的概念來表達觀點？是否讓您覺得這本書好像是寫給的無所不知的精英份子而不是給您閱讀的？

本書不會是前述那樣的書籍，這裡的內容很踏實、集中焦點並切中要害。

這本書也不是入門書，不會從最基本的開始講述，不會有讓您厭煩的程式設計和語言的基礎知識。本書內容不會試圖討好您和讓您感覺舒服，因為書中的內容會讓您多動腦，挑戰和激發您的學習欲望。雖然內容有些挑戰性，但不會嚇到您和侮辱您的智商。

「重構（Refactoring ）」這門學科是在不破壞程式碼的情況下，把不好的程式碼轉換為好程式碼。當我們考量到現化文明都依賴於軟體才能繼續存在時，似乎找不到比「重構」更值得的研究課題。

也許您認為這種講法太過誇張，但並不是。請看一看您的周圍，目前身邊有多少個正在執行軟體的處理器呢？手錶、手機、車鑰匙、耳機...等等，在您周邊 30 公尺以內有多少個呢？微波爐、電磁爐子、洗碗機、溫度自動調節器、洗衣機、車子...等等。

如今的社會，沒有軟體就什麼都不能運作了。沒有軟體，您不能買賣任何東西，無法開車或飛去任何地方、不能煮熱狗、不能看電視、也無法打電話給其他人。

這些軟體實際上真的都是好的程式碼嗎？想一想您現在正在使用的系統，這套軟體真的符合乾淨無暇（clean）原則嗎？ 還是像大多數軟體一樣有點糟，急需重構？

本書內容不會介紹以前您可能聽過或讀過的那種枯燥乏味和簡單化的重構理論。這裡談的都是真正的重構實務，在真實專案中所進行的重構，在過去遺留系統中所進行的重構，在我們幾乎每天所面對的各種環境中進行重構。

更重要的是，這本書不會讓您覺得系統沒有經過自動化測試而感到內疚。作者意識到大多數承繼下來的系統都是隨著時間的推移而成長和演變，我們不會那麼幸運能夠擁有這樣的測試套件。

本書制定了一組簡單的規則，讓讀者可以遵循這些規則，可靠地重構複雜、混亂、糾纏、未經測試的系統。學習並遵循這些規則後，您就能真正提高所維護之系統的品質。

但請不要誤解我的意思，本書並不是什麼銀彈和靈丹妙藥。要對舊有、粗糙的、未經測試的程式碼進行重構絕非易事，但在您學習了本書中的規則和範例後，就有機會能攻克長期困擾著您的問題。

所以我建議您仔細閱讀這本書，研究其中的範例，認真思考作者提出的抽象概念和意圖。請取得作者提供的程式碼庫，跟著書中的內容與作者一起重構，隨著書中的內容完成這趟重構旅程。

學習是需要花時間，閱讀本書也許讓您沮喪，而且書中內容不時挑戰您。但您在經歷過後，就會帶著一套技能從另一邊走出來，這些技能會在未來的職業生涯中提供很好的服務。您還學會怎麼區分「好的」程式碼和「不好的」程式碼，以及怎麼讓程式碼變「乾淨無暇」。

—Robert C. Martin (aka Uncle Bob)

序

父親在我很小的時候就教我寫程式，所以從我記事以來就一直在思考結構。我一直都很樂於助人，也是我每天起床的動力。因此，教學對我來說是很自然且有趣的工作。在大學有機會擔任助教的職位時，我立即接受了這項職務。我有過幾次這樣的機緣，但不幸的是我的運氣用完了，整個學期什麼也教不了。

身為創業者，我決定創辦一個學生組織，讓學生之間能互相指教學習。任何人都能參加或發言，主題範圍從業餘個人專案所學到的經驗教訓，到課程沒涵蓋的高階主題都能討論。我相信這樣會讓我有機會指導他人，從結果來看我這樣做是正確的。事實證明，電腦科學家都很膽小，我在連續主持近 60 週之後才有機會開始進行指導的教學工作。在這段時間裡，我學到了很多東西，無論是關於我指導的主題或是關於教學技巧都收獲良多。這樣的互動還催生了一群好奇的人，在那裡我遇到了我最好的朋友。

離開大學之後的一段時間裡，我和其中一位朋友出去旅遊。由於沒有什麼正事可做，所以他問我是否要即興指導交流一下，因為我在學校也做了很多這樣的交流指導，我回答說：「讓我們一起研究看看」。開啟筆電之後我並沒有停下來喘口氣，而是直接輸入了本書 Part 1 部分的主要範例。

當我把手指從鍵盤上拿開時，他驚呆了。他以為那是今天討論所準備的示範，但我有不同的想法。我想教他「重構」。

我的目標是在一小時之後，他能像重構大師一樣編寫設計程式碼。由於重構和程式碼品質是很複雜的主題，很明顯我們不得不做些假設和虛構。所以，我查看了程式碼，並試圖想出一些規則，讓他做正確的事，同時又很容易記住。在這樣的練習過程中，即使其中的程式是假設和虛構，他也對程式碼進行了真正的改進。這樣的成果很令人鼓舞，他的進步如此之快，以至於當晚回到家時，

就寫下了我們所有發生的一切。在職場工作中僱用初階新手時，我重複了這個練習，慢慢收集、建構和提煉本書中的規則和重構模式。

目標：選定的規則和重構模式

> 完美的達成，不是在沒有什麼可以添加，而是在沒有什麼可以去掉的時候。
>
> —Antoine de Saint-Exupéry

世界上有數百種重構模式（refactoring patterns），我只選了 13 種，這樣做的原因是我相信「深刻的理解」比「廣泛的熟悉」更有價值。我還想製作一個完整的、連貫的故事，因為這樣有助於增加視角，使主題更容易在腦中組織起來。相同的論點和做法也可套用到「規則（rules）」上。

> 太陽底下沒什麼新鮮事。
>
> —Book of Ecclesiastes

我並沒有聲稱在本書中有提出很多新奇的內容，但我卻以一種有趣又有用的方式把它們結合起來。許多規則源自 Robert C. Martin 的「Clean Code（Pearson, 2008 年出版）」一書，但經過修改後更易於理解和應用。許多重構模式源自於 Martin Fowler 的「Refactoring（Addison-Wesley Professional, 1999 出版）」一書，但經過調整改編後可用編譯器來處理，而不需要再依賴強大的測試套件。

本書讀者與學習路徑

本書是由風格很不相同的兩個部分所組成。Part 1 的內容針對是個人的學習，為重構觀念奠定堅實的基礎。我所關注的不是全面性，而是易學性。這部分的內容適用於對重構基礎還沒有確實理解的讀者，例如學生、初階新手或自學成才的開發人員。如果您查閱本書隨附的原始程式碼後，覺得「這看起來很容易改進啊」，那麼 Part 1 的內容就不適用於您了。

在 Part 2 的內容中，我把焦點放在上下脈絡（context）和團隊（team）。我選擇了現實世界中關於軟體開發最有價值的課程內容，有些主題較理論性的，例如「與編譯器的協作」和「遵循程式碼中的結構」；有些主題則是較實用性的，例如「愛上刪除程式碼」和「讓不良的程式碼看起來更糟糕」。因此這部分的應用範圍更廣，即使是有經驗的開發者也能從這些章節中學到東西。

因為 Part 1 的章節都使用了一個總體的範例來說明，所以各章節會緊密地連結在一起，應該一章接一章閱讀。但在 Part 2 中，除了少數相互參照引用之外，這些章節大部分都是獨立的。如果您沒有時間從頭到尾閱讀整本書，可以輕鬆單獨選擇 Part 2 中最讓您動心的主題進行閱讀。

關於教學指導這件事

我花了很多時間反思教學指導這件事。傳授知識和技能這件事本身就很有挑戰性，老師得要激發學生的學習動機、信心和反思能力，但學生的腦子卻更喜歡停下來休息和分散注意力，不會想要專心學習。

要應付這樣的腦洞掙扎，得要先激發學生的學習動機。我通常會用一個看起來很簡單的練習吸引學生，當學生發現自己無法解決問題時，好奇心就會被激發了，這就是 Part 1 程式碼的主要作用和目的。書中簡單的一句指示要讀者「改進這個程式碼庫」，但程式碼的品質已經達到一定水準，許多人不知道該如何要繼續下去。

第二階段是讓學生有信心去嘗試動手實作和應用新的知識或技能。我第一次意識到這一點的重要性是在課外學習法文時，當老師想要教我們新的片語時，她會帶著我們走過下列幾個同樣的步驟：

1. 她會要求我們每個人逐字重複那個片語，這個純模仿的步驟會讓我們必須真的說出那個片語一次。
2. 她向我們每個人提出了一個問題。雖然我們並不都能理解問題的意思，但語調讓我們明白這是個問題。由於我們沒有其他的工具，所以又重複了那個片語。這種重複增強了我們的自信，也讓我們對片語的語境上下脈絡有了初步的認知，就在此時，我們的理解就開始了。

3. 她要求我們在對話中使用那個片語。能夠合成出新的東西是教學的目標，這需要理解和信心兩者兼備。

我學到這種方法源自日本茶道和劍道的「**守破離**（**Shuhari**）」概念，這個概念現在越來越受歡迎。它由三個部分組成：「守（Shu）」是模仿，不問為何也不去理解；「破（ha）」是變化，做些稍微新穎的事情；「離（ri）」是創新，去超越已知的範疇。

「守破離」概念貫穿整個 Part 1。我建議一開始先不求甚解直接遵循「規則」，然後在理解其價值後，再加以變化。最後，當您掌握這些技巧，就可以轉進「程式碼異味（code smells）」的領域。以重構模式來說，我會展示如何在真實的程式碼中進行處理，讀者應該跟著做（模仿）。接著會在不同的情境脈絡中展示同樣的重構模式（變化）。最後，我會提供另一個適用於該模式的場景，在讓場景中我會鼓勵讀者自行嘗試（綜合運用）。

讀者可以利用本書來驗證上述的步驟，並使用 Git 標籤來驗證其程式碼。如果您沒有跟著程式碼走，那您可能會覺得有些步驟一直重複，所以我建議您在閱讀 Part 1 時要配合鍵盤輸入，實際動手實作。

關於程式碼

書中含有很多範例程式碼，都是以「Listing 編號」獨立的樣式編排呈現，程式碼是以等寬字型來排版，若是在文字段落的說明中，因為與中文字已有區別，所以只用一般英文來呈現。程式碼也會用語法突顯標示，關鍵字會以粗體標示，這樣能更好地理解程式碼的架構。

在多數情況下，原始程式碼會被重新編排，我們會加上斷行和重新調整縮排，以配合書中有限的版面空間。此外，如果內文中有說明講述程式碼的意義時，Listing 中程式碼的注釋（comment 或譯註解）就不會列出。Listing 中也會有加上箭頭的圖說文字，用來強調程式碼中的某些重點概念。

書中隨附的程式碼可連到 Manning 出版社（https://www.manning.com/books/five-lines-of-code）網站下載，另外也可連到 GitHub 倉庫（https://github.com/thedrlambda/five-lines）下載。

1 重構重構

本章內容

- 理解重構的要素
- 將重構融入您的日常工作中
- 重構安全的重要性
- 簡介 Part 1 的總體範例

眾所周知，「高品質」的程式碼能讓維護成本變得更便宜且錯誤更少，而且還會讓開發人員更快樂。實現高品質的程式碼最常見做法就是透過「重構（refactoring）」。然而，目前教授重構的作法都是以**程式碼異味（code smells）**和**單元測試（unit testing）**為主，這樣對初學者來說門檻較高且不必要。我相信，只要多練習一些簡單的重構模式，任何人都能夠安全地進行程式碼重構。

在軟體開發中，我們把問題放在如圖 1.1 所示的某個位置，用來表示缺乏足夠的技能、文化、工具或這些因素的組合。重構是一項複雜的工作，因此放在中心位置，它需要下列各個組件來配合：

- **技能（skill）**：我們需要具備技能，知道哪些程式碼是不好且需要重構。有經驗的程式設計師能透過對程式碼異味的了解來判斷這一點。但程式碼異味的界限是模糊的（需要判斷和經驗），而且其解釋存有不確定性，因此不容易學習。對於開發新手來說，理解程式碼異味可能更像是第六感，而不是技能。
- **文化（Culture）**：我們需要一種鼓勵花時間進行重構的文化和工作流程。在許多情況下，這種文化是透過測試驅動開發中著名的「**紅燈／綠燈／重構**」循環來達成的。然而，在我看來，測試驅動開發是一門更難的技藝。「紅燈／綠燈／重構」循環也不容易在舊有遺留的程式碼庫中進行。
- **工具（Tools）**：我們需要工具來確保重構是安全的。最常用的方法是透過自動化測試來達成。但如前所述，學習如何進行有效的自動化測試本身就很不容易。

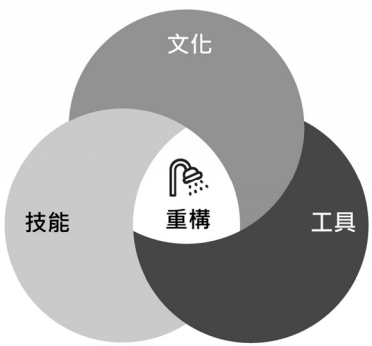

圖 1.1　技能、文化和工具

以下章節會深入探討各個領域，並解說怎麼從更簡單的基礎開始進行重構，而不需要進行測試和抽象程式碼的改善。這種學習重構的方式能快速提升開發新手、學生和程式愛好者的程式品質。技術主管也可用本書中的方法作為引入團隊重構的基礎，就算是不常進行重構的團隊也能夠依照書中方法進行重構。

1.1　什麼是重構？

我會在下一章更詳細地回答「什麼是重構？」這個問題，但在深入探討重構的各種方法之前，先對重構有個直覺的理解是有幫助的。簡單來說，重構是指「改變程式碼而不改變其功能」。讓我們透過一個重構前後的例子來解釋我所談的內容。在這裡，我們會把表示式替換成區域變數。

Listing 1.1　之前

```
01 |   return pow(base, exp / 2) * pow(base, exp / 2);
```

Listing 1.2　之後

```
01 |   let result = pow(base, exp / 2);
02 |   return result * result;
```

要進行重構的理由很多：

- 讓程式碼更快（以前述範例來看）
- 讓程式碼更小
- 讓程式碼更加通用或可重複使用
- 讓程式碼更容易閱讀或維護

最後一個理由很重要且這是判斷是否為好程式碼的核心依據。

> 定義　**好的程式碼**（**Good code**）是指讓人好閱讀看得懂，且易於維護的程式，並且能照其設計執行，得到正確的成果。

由於重構不能改變程式碼原本的功用，因此本書中我們把重點放在讓人好閱讀和易於維護上。我們會在第 2 章中更詳細深入探討重構的這些原因。在本書中，我們只考慮能產生好的程式碼的重構，因此我們所使用的定義如下。

> 定義　**重構**（**Refactoring**）是把程式碼改得更易閱讀和更好維護，但不改變其功能。

我還是必須提醒一下，這裡所談的「重構」類型很大程度是針對物件導向的程式語言。

大多數的人都把程式設計（programming）想成是寫程式（writing code），但對於大多數的程式設計師來說，他們花更多時間在閱讀和試著理解程式碼的用途，而不是在寫程式。這是因為我們在一個複雜的領域中工作，沒有理解就隨意更改程式碼可能會導致災難性的失效結果。

因此，重構的第一個原因純粹是成本效益的考量：程式設計師的時間很寶貴，如果我們讓程式碼庫（codebase）更易讀，就能釋放出更多時間來實作新的功能。第二個原因是，讓程式碼更易於維護，這表示錯誤（bugs）更少且更易修復。第三個原因是，好的程式碼庫更有趣、更讓人愉悅。當我們閱讀程式碼時，需要在腦中建立一個模型，了解程式碼是要做什麼，當需要同時記住很多細節時，就會變得疲累。這就是為什麼從頭開始寫程式會較有趣，而為什麼除錯會讓人覺得很可怕的原因。

1.2　技能：要重構什麼？

了解哪些程式碼需要重構是我們要面對的第一道門檻。通常，重構是要和「**程式碼異味**（**code smells**）」放在一起學習的，這些「異味」是描述程式碼可能有問題的地方。雖然很有用，但也很抽象且難以入門，這需要很多時間和經驗才能對「異味」有感覺。

本書採用了不同的做法，提出易於辨識和套用的「規則」來確定需要進行重構的內容是什麼。這些規則易用且能夠快速學會，但有時候這些規則也可能過於嚴格，有時會去修復一些本來沒有異味的程式碼。然而就算我們遵循這些規則來進行處理後，仍可能產生有異味的程式碼。

如圖 1.2 所示，程式碼異味和規則的重疊不完美。我提出的規則並不能完全涵蓋好程式碼的全部，然而這些規則能讓讀者更快地發展出像高手一樣對好程式

碼的感覺。現在讓我們舉一個例子，說明程式碼異味和書中的規則之間的有何區別。

圖 1.2　程式碼異味和規則

1.2.1　程式碼異味的範例

有個大家都知道的程式碼異味：一個函式應該只做**一件事**。這是個很好的指導方針，但很難釐清「**一件事**」代表的是什麼。再看一下前面的程式碼：這裡是否有異味呢？程式碼的作用是進行除法、乘方的運算，然後再進行乘法運算。這是說程式做了三件事嗎？另一方面，這段程式只返回一個數字且不改變任何狀態，那是否又表示只做了一件事呢？

```
01 |   let result = pow(base, exp / 2);
02 |   return result * result;
```

1.2.2　規則的範例

將前面的程式碼異味與以下規則進行比較（在第 3 章會詳細介紹）：一個方法（method）永遠不應該超過**五行程式碼**。我們一眼就能看出這個方法是否符合該規則，而不會有進一步的問題。這條規則很清晰、簡潔且好記，尤其這也是本書的書名主題。

請記住，書中介紹的規則只是個輔助輪，如之前所討論的，這些規則不能保證在所有情況下都能得到好的程式碼，而且有時候遵循這些規則反而可能是錯的。然而，如果您不知道重構要從哪裡著手，這些規則仍是很有用的提示，可以激勵您把程式重構成好的程式碼。

請留意，所有規則的名稱都使用「**永不**（**never**）」等絕對詞彙來表示，因此很容易記住。但是，詳細的說明中通常會指出例外情況，也就是**不適用**規則的情況，解釋中還說明了規則的目的。在開始學習重構時，我們只需要使用絕對的情況，當這些情況被內化後，就可以開始去了解例外的情況，然後理解其例外的意圖，這樣我們就能成為程式碼專家了。

1.3 什麼時候要進行重構？

> 重構就像洗澡。
>
> —Kent Beck

重構最有效、最省錢的方式是定期進行。如果可以的話，我建議您把重構納入日常的工作中。大多數文獻建議使用「紅燈／綠燈／重構（red-green-refactor）」這套工作流程，但是，正如前面提過的，這種做法會把重構與測試驅動開發綁在一起，而在本書是希望把兩者分開，只專注在「重構」的部分。因此，我建議使用更一般化的「六步驟」工作流程來解決所有程式設計的工作，如圖 1.3 所示：

1. **探索**（**Explore**）。在開始任何程式工作之前，通常不完全確定需要建立什麼。有時候，客戶不知道他們想要我們建構什麼，而需求是以含糊不清的散文形式書寫。而有時候，我們甚至不知道這項任務是否能成為解決方案。因此，我建議從探索和實驗當作起始。快速實作一些東西，然後從客戶那裡進行驗證，確定您開發的東西與客戶需求是一致的。
2. **指定**（**Specify**）。在探索之後，對於需要建構的內容有了更好的理解，應該能明確地指定需求。最好是用自動化測試的方式來驗證結果是否有按照指定的需求正確執行。
3. **實作**（**Implement**）。實作程式碼。
4. **測試**（**Test**）。確定程式碼有通過步驟 2 的需求規格測試。
5. **重構**（**Refactor**）。在交付程式碼之前，請確保程式很容易能讓下一個人（可能是您自己）使用。

6. **交付**（**Deliver**）。有很多種交付程式碼的方式，最常見的做法是透過拉取請求（pull request）或推送（pushing）把程式碼交到特定分支（branch）。最重要的是要確保您的程式碼有送到使用者手中，否則交付就沒有什麼意義了？

圖 1.3　工作流程

因為我們進行的是以規則為基礎的重構，所以工作流程直接且容易進行。圖 1.4 放大了流程中的步驟 5：**重構**。

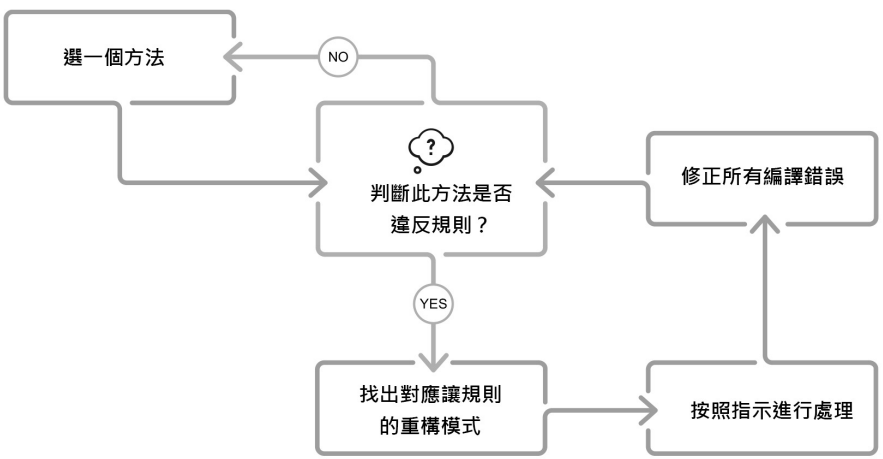

圖 1.4　重構步驟的細部內容

我設計這些規則是為了方便記憶，而且很容易在不需要任何協助的情況下就能發現使用的時機，這表示判斷某個方法是否違反規則也是很簡單的。每條規則都有幾種與之連結的重構模式，這樣就很容易知道怎麼修復問題。重構模式具

有明確的步驟指示和說明，確保您不會意外破壞原本的任何內容。書中的許多重構模式故意使用編譯錯誤來協助讀者不會引入錯誤。一旦經過練習，規則和重構模式就變得容易運用。

1.3.1 在遺留的舊系統中進行重構

就算我們是從一個龐大的遺留舊系統開始，也有聰明的做法可以把重構納入我們的日常工作，而且不必先停下一切然後才去重構整個程式碼庫。您只需要遵循下面這句很棒的名言：

> 先讓程式碼容易理解和修改，將來的修改或新增功能都會更容易。
>
> —Kent Beck

每當我們要實作新功能之前，都會先重構程式碼，這樣就更容易加入新的程式碼了。這就像在烘焙之前先準備好所有材料一樣。

1.3.2 什麼時候不應該重構？

在大多數情況下重構是很棒的處置，但這麼做也會有一些缺點。 重構可能很耗時，尤其是在您不熟悉且不常做的時候。正如前面提到的，程式設計師的時間是很寶貴的。

在以下的三種的程式碼庫中，重構可能不值得進行：

- 您要寫的程式只會執行一次就刪掉的。這種程式在 Extreme Programming 社群中稱之為 **Spike**。
- 程式碼準備退役前處於維護模式。
- 具有嚴格效能要求的程式碼，例如嵌入式系統或遊戲中的高階物理引擎。

除了以上的情況，我建議都要進行重構。

1.4　工具：如何（安全地）重構

我和大家一樣喜歡自動化測試。然而，學會如何有效地測試軟體本身就是一項十分複雜的技能。因此，如果您已經知道怎麼進行自動化測試，請配合書中內容隨時使用，但如果您還不會，那也別擔心。

我們可以把測試想像成：自動化測試對軟體開發就像煞車對汽車一樣重要。汽車裝上煞車是為了能安全地快速行駛。在軟體開發中也是如此，運用自動化測試的目的是要讓我們能在快速開發時更有信心和更具安全感。但在本書的內容中，我們是在學習全新的技能，所以不需要跑得太快。

相反地，我會建議多依賴其他工具的配合而不先運用自動化測試，這些工具的配合有：

- 結構化、逐步詳細的重構模式，就像跟著食譜做菜一樣
- 版本控制
- 編譯器

我相信，如果精心設計重構模式，並且逐步小心地執行，就有可能在不破壞任何東西的情況下進行重構。特別是當整合開發環境（IDE）可以為我們執行重構時，這一點十分重要。

為了彌補本書沒有討論「測試」的事實，我們使用編譯器和型別來捕捉可能犯下的許多常見錯誤。即使如此，我還是建議讀者定期打開您在開發的應用程式，並檢查它是否完全正常。每當我們驗證通過，或者知道編譯器編譯的結果是正常的，就提交一次，當應用程式在某個時刻掛掉了，而我們不知道如何立即修復時，就能輕鬆跳回到上一次它正常運作的時間點。

如果我們處理的是沒有進行自動化測試的真實系統，仍然可以進行重構，但需要從某他方法來獲得信心，例如使用整合開發環境進行重構、手動測試、採取超小型的步驟來進行，或者其他方法。然而，這需要花額外的時間來進行這些動作，如此一來，進行自動化測試反而更具成本效益。

1.5 開始運用工具

正如我前面提過的，書中討論的重構類型需要使用物件導向語言來配合。這是讀者在閱讀和理解本書時所必需知道的重要事實。

寫程式和重構都是需要用到手指頭來實作的技藝。因此，最好是實際動手跟著範例來實驗，在動手練習時能體會更深並享受其樂趣。為了讓您能跟著書中內容來學習和實作，您需要以下的工具，其相關安裝說明請參考附錄。

1.5.1 程式語言：TypeScript

這本書中所有的程式碼範例都是以 TypeScript 編寫的。我選用 TypeScript 的原因有很多，其中最重要的是它看起來和感覺上很像一些常用的程式語言，例如 Java、C#、C++ 和 JavaScript，因此熟悉這些語言的人應該能輕易地閱讀看懂 TypeScript。此外，TypeScript 還提供一種方法，能夠把完全沒有物件導向的程式碼（也就是沒有任何類別的程式碼）轉變成高度物件導向的程式碼。

NOTE 為了能更好地運用書中的版面空間，本書使用了一種避免換行但仍易讀的程式風格。我並不建議您也用這種風格，除非您碰巧也要寫一本含有大量 TypeScript 程式碼的書。這也是為什麼本書中的縮排和大括號有時會與專案程式碼不同的原因。

如果您還不熟悉 TypeScript，書中會在需要補充說明時，用以下這樣的方框來解釋所有要注意的內容。

在 TypeScript …

我們使用 identity（===）來檢查相等性，因為這種表示更像是我們對相等性的期望，而不是雙等號（==）。請思考以下例子：

- `0 == ""` 為 `true`。
- `0 === ""` 為 `false`。

雖然書中的範例是用 TypeScript 寫的，但所有的重構模式和法規都是通用的，適用於任何物件導向的語言。在少數情況下，TypeScript 會對我們有幫助或阻礙還不知道，但這些情況會被明確提出來解說，我們也會討論在其他常見的程式語言中如何處理這些情況。

1.5.2　編輯器：Visual Studio Code

我沒有假設讀者要用某種特定的編輯器，但如果您沒有特別的偏好，我建議您使用 Visual Studio Code。這套編輯器與 TypeScript 很能配合。而且能支援在背景終端機中執行「tsc -w」來進行編譯，這樣我們就不會忘記使用了。

NOTE　Visual Studio Code 與 Visual Studio 是完全不一樣的工具。

1.5.3　版本控制：Git

雖然在閱讀本書時不一定需要使用版本控制來配合，但我仍強烈建議使用，因為版本控制能讓您更輕鬆地復原某些修改，避免迷失和遺憾。

重設回參照的解決方案

任何時候，您都可以用類似下列的命令跳轉回到主要區段開始時的程式碼。

```
git reset --hard section-2.1
```

請小心：您可能會遺失做過的修改。

1.6 總體範例：2D 的拼圖益智遊戲

最後，讓我們討論一下要怎麼教授所有這些精彩的規則和好用的重構模式。本書是圍繞一個主要的總體範例來配合解說，這是個 2D 的拼圖益智遊戲，很像經典遊戲 Boulder Dash（圖 1.5）。

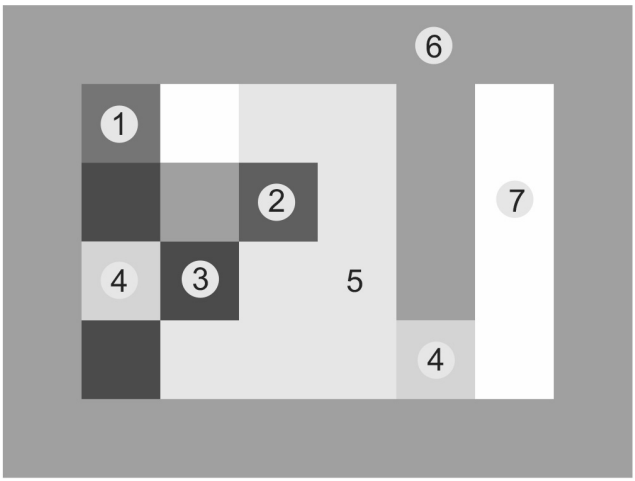

圖 1.5 遊戲的畫面示意

這表示在本書的 Part 1 內容中，我們會有一個重要的程式碼庫（codebase）可以運用。使用同一個範例來解說會節省很多時間，如此一來，我們就不必在每一章重新熟悉另一個新的範例。

這個範例是以真實業界中常用的風格來撰寫的，如果讀者已經學會本書教授的技巧，這個練習實作就不會太難。範例的程式碼已經遵循了 DRY（Don't Repeat Yourself，不要重複自己寫過的東西）和 KISS（Keep It Simple, Stupid，保持簡單）的原則來編寫，即便如此，但也沒有超越 DRY KISS 太多。

這裡選用電腦遊戲為例，是因為當我們進行手動測試時，如果某些東西的行為不正確是很容易被發現：我們對遊戲應該要怎麼動作會有一種直覺的反應。無論如何，遊戲程式還是比查看金融系統的 log 日誌稍微有趣些。

使用者利用鍵盤上的方向鍵來控制玩家的方塊。遊戲目標是把方塊（在圖 1.5 中標示為 2 的方塊）移到右下角。雖然在書中呈現的色階不顯示，但是在遊戲程式中各元素代表的顏色意義標示如下：

1. 紅色方塊代表玩家。
2. 棕色方塊代表箱子。
3. 藍色方塊代表石頭。
4. 黃色方塊是鑰匙或鎖，稍後我們會修正。
5. 綠色調方塊稱為 **flux**，是會**變化**的區塊。
6. 灰色方塊代表牆壁。
7. 白色方塊代表空的（air）。

如果箱子或石頭沒有任何支撐，它就會掉下來。玩家一次只能推動一個石頭或箱子，前提是沒有被阻擋或掉落。箱子與右下角之間的路徑最初會被鎖住，所以玩家必須先取得鑰匙才能打開它。玩家可以在變化的區塊上移動「吃掉」（移除）這些變化的區域。

現在是下載遊戲程式並試玩的絕佳時機：

1. 請在您想要儲存遊戲程式的路徑位置開啟主控台。

   a. 使用 git clone https://github.com/thedrlambda/five-lines 來下載遊戲的原始程式碼。

   b. 使用 tsc -w，每當 TypeScript 發生更改時，就會將其編譯為 JavaScript。

2. 請在瀏覽器中開啟 index.html。

在程式碼中可以修改關卡，所以請隨意更新 map 變數中的陣列來建構屬於自己的地圖，讓您玩得更開心！（請參閱附錄來查看改更的範例）

1. 請在 Visual Studio Code 中開啟範例程式所在的資料夾。
2. 選取「終端機→新增終端」指令。
3. 在終端機中執行 tsc -w 指令。
4. 當 TypeScript 有變更修改時，就會在背景中進行編譯，您現在可以關掉終端機。
5. 每次修改後，稍微等待一下 TypeScript 編譯，然後重新載入瀏覽器即可。

這是您在 Part 1 跟著範例編寫程式時會用到的相同步驟。不過在那之前，我們會在下一章先學習更詳細的重構基礎知識。

1.6.1 熟能生巧：第二個程式碼庫

因為我堅信動手實踐才是正道，所以又做了另一個專案，但沒有提供解答。如果您想挑戰一下，可以在複習重讀時使用這個專案來練習，或者如果您是當老師，可把此專案當作學生的練習題。這個專案是個 2D 動作遊戲，兩個程式碼庫使用相同的樣式和結構，具有相同的元素，在重構時也是採用相同的步驟來完成。雖然這第二個程式碼庫稍微有點進階，但仔細遵循書中講解的規則和重構模式，應該能重構生成預期的結果。若想要取得此專案的程式碼庫，請利用下列網址：https://github.com/thedrlambda/bomb-guy 下載，並採用前述的相同步驟來進行重構練習。

1.7 關於真實世界軟體的提醒

這裡要重申的是，本書的焦點是介紹重構，不是要提供可在所有情況下都能套用到上線程式碼的特定規則。規則的使用是先學習和知曉其名稱，然後再遵循它們。一旦熟悉之後，再去學習其細部描述說明和了解其例外情況。最後利用這些知識來建立對底層程式碼異味（code smell）的理解。過程如圖 1.6 所示。

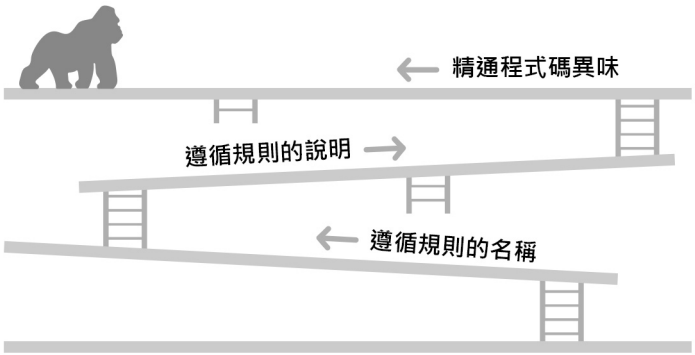

圖 1.6　怎麼使用規則

這也解釋了為什麼我們無法製作自動化重構的程式。（我們也許可以製作外掛 plugin，根據規則來突顯**可能有問題的**程式碼區域。）規則的主要目的是建立對事情的理解。總之：在您更理解熟悉之前請先直接遵循規則。

還有一點要注意，由於我們只專注於學習重構，而且有個安全的環境來練習，所以我們可以不使用自動化測試—但在真實世界的系統中，這可能不是真實的情況。書中的內容之所以這樣做，是因為分開來學習自動化測試和重構會比較容易和單純的。

總結

- 進行重構需要結合技能（skills）、文化（culture）和工具（tools），了解要重構什麼、重構的時機，以及如何進行重構。
- 一般來說，程式碼異味（code smell）是用來說明什麼樣的程式需要重構。但對新手程式設計師來說很難內化，因為程式碼異味是比較模糊的概念。本書提供的學習方式是以具體的規則來替代程式碼異味。這些規則有三個抽象層級：非常具體的名稱、有加上細微差別例外情況的描述，以及最終從它們衍生出來的程式碼異味。
- 我相信自動化測試和重構可以分開來學習，這樣可以進一步降低學習的門檻。這裡不使用自動化測試，而是利用編譯器、版本控制和手動測試等工具和做法來配合。
- 重構的工作流程與測試驅動開發的紅燈／綠燈／重構循環是相關連結的。這表示重構依賴於自動化測試。因此，我建議使用六個步驟的工作流程（探索、明確指定需求、實作、測試、重構、交付）來開發新的程式碼，或在修改程式碼之前進行重構。
- 在本書的 Part 1 內容中，我們會使用 Visual Studio Code、TypeScript 和 Git 來實作轉換 2D 益智拼圖遊戲的原始程式碼。