前言

在開始撰寫《*Software Architecture in Practice*》第四版時，我們問了自己一個問題：軟體架構依然重要嗎？現在有很多參考架構是專為雲端基礎設施、微服務、框架和你可以想像的任何領域和品量屬性而設計的。如今，架構師只要從豐富的工具和基礎架構方案中做出選擇，然後實例化並設置它們，就可以做出一個架構了！這些工具的興起可能讓一些人以為不需要學習軟體架構知識了！

我們從以前到現在都非常確定事實並非如此。我們也許不太客觀，於是，我們訪問了一些同事，他們曾經在醫療保健、汽車、社交媒體、航空、金融、電子商務等領域擔任架構師，他們都是不被教條式偏見支配的族群，他們的意見證實了我們的想法——現在的架構與 20 多年前，當我們撰寫本書第一版時一樣重要。

讓我們來研究一下我們聽到的幾個原因。首先，新需求出現的速度多年來不斷加快，甚至現在仍在持續加速中。由於客戶和商業需求，以及競爭壓力的驅動，當今的架構師面臨不斷增加的功能請求以及需要修復的 bug。如果架構師不在意系統的模組化（而且，微服務不是萬能的），他的系統很快就會變成難以了解、變更、偵錯、修改的瓶頸，並延誤他的工作。

第二，雖然系統的抽象程度不斷提升（我們可以也經常使用許多複雜的服務，且不需要知道它們是如何製作的），但我們要設計的系統的複雜度至少也以同樣的速度不斷提升。這是一場軍備競賽，但架構師不是上風的一方！架構師一直致力於克服複雜性，這種情況在可見的未來還不會消失。

談到抽象程度的提升，model-based systems engineering（MBSE）在過去十年左右已經成為工程領域的一股強大力量。MBSE 正式地運用建模（modeling）來支援系統設計（和其他工作）。International Council on Systems Engineering（INCOSE）將 MBSE 視為整個系統工程學底下的「轉型推動因素（transformational enabler）」之一。所謂的模型

（model）是以圖形、數學或物理來表示可推理的概念或結構。INCOSE 正試著將工程領域從「基於文件」的思維方式移往「基於模型」的思維方式，後者一貫地使用結構模型、行為模型、性能模型和其他模型來建構更好、更快、更便宜的系統。MBSE 本身不在本書的討論範圍，但我們不得不注意到，建模的對象就是架構。那麼，這些模型的作者是誰？答案是：架構師。

第三，資訊系統的飛快發展（以及前所未有的員工流動率），意味著所有人能無法理解現實世界系統中的一切，僅依靠聰明才智和努力工作是不夠的。

第四，儘管現在有很多工具可以將過去需要親力親為的工作自動化（例如，Kubernetes 的協調、部署和管理功能），但我們依然要了解我們所依賴的系統的品質屬性，也要了解將多個系統組合起來產生的品質屬性。大多數的屬性（性能、資訊安全、妥善性…等）都很容易受到「最弱的環節」影響，那些最弱的環節只會在你組合多個系統時出現並反咬你一口。如果沒有人指引如何防禦災難，這個組合極有可能失敗，這個引導者就是架構師，儘管頭銜與工作內容不太一致。

考慮到這些因素，我們認為的確有需要出版這本書。

但真的需要出第四版嗎？（結果應該很明顯）我們同樣得出一個有力的結論，是的！自從上一版出版以來，計算機領域已經有很大的變化。在架構師的日常工作中，有一些以前沒有考慮過的品質屬性越來越重要。隨著軟體持續滲透社會的各個層面，**安全**已經變成許多系統的首要問題，你可以想一下現在的軟體怎麼控制著我們駕駛的車輛。**能源效率**也是十年前的架構師不太考慮、但現在必須關注的品質因素，從耗費大量能源的巨型資料中心，到我們身邊使用電池的小型（甚至微型）行動設備與 IoT（物聯網）設備。此外，因為我們比以往任何時候都更頻繁地利用既有的組件來建構系統，**可整合性**這個品質屬性也越來越吸引我們的注意力。

最後，我們建構系統的方式與十年前不同。如今的系統通常是用雲端的虛擬化資源來建立的，它們需要提供和依靠清楚的介面，而且，它們越來越行動化，這帶來各式各樣的機會和挑戰。所以，我們在這一版加入關於虛擬化、介面、行動性和雲端的章節。

正如你所看到的，我們說服了自己出版這本書，希望我們也說服了你，也希望你會認為這本第四版是值得放在書架上（實體的或電子的）的補充讀物。

PART I 導論

1

何謂軟體架構？

我們被叫來擔任未來的架構師，而不是當它的受害者。

—R. Buckminster Fuller

我們之所以撰寫（就我們而言）或閱讀（對你而言）一本關於軟體架構（且濃縮了許多人的經驗）的書籍，正是因為我們認為

1. 一個可推理的軟體架構對成功地開發軟體系統而言非常重要，而且
2. 關於軟體架構的知識體系足以寫成一本書。

這兩個假設在以前是有待證明的。本書的早期版本曾經試著說服讀者，這兩個假設都是對的，並且在說服讀者之後，提供基本的知識，讓讀者可以自行運用架構的實踐法。如今，這兩個假設已經沒有什麼爭議了，所以這本書會將更多重心放在教導知識上，而不是在說服讀者上。

軟體架構的基本原則在於，每一個軟體系統都是為了滿足組織的商業目標而建構的，而且系統的架構是這些（通常是抽象的）商業目標與最終（具體）的系統之間的橋梁。雖然從抽象的目標到具體的系統之間的途徑有時很複雜，但好消息是，我們可以用「實現商業目標的技術」來設計、分析和記錄軟體架構，並馴服並駕馭複雜性。

所以，本書的目標正是設計、分析和記錄架構。我們也會討論推動這些活動的力量，主要是帶來品質屬性需求的商業目標。

在這一章，我們將從軟體工程的角度來關注軟體架構，也就是說，我們將探討軟體架構可為專案帶來什麼價值。稍後的章節將從商業和組織的角度進行討論。

1.1 軟體架構是什麼？不是什麼？

你可以在網路上查到許多軟體架構的定義，但我們喜歡這個：

> 系統的軟體架構（architecture）是推理（reason）系統所需的一組結構（structure）。這些結構是由軟體元素、它們之間的關係，以及兩者的屬性組成的。

這個定義和提到系統的「早期」、「主要」或「重要」決策的其他定義大異其趣。雖然許多架構決策的確都是在早期決定的，但並非全部如此，尤其是 Agile（敏捷）和螺旋式開發專案裡面的決策。更何況，許多早期的決策並非我們所認為的架構。此外，我們很難判斷一項決策是不是「主要」的，有時這需要時間來證明。因為決定架構是架構師的主要職責之一，我們必須知道架構是由哪些決策組成的。

相較之下，結構很容易識別，它們是協助進行系統設計和分析的強大工具。

所以，我們的結論是：架構是讓人們可以進行推理的結構。

讓我們來看一下這個定義的一些含義。

架構是一組軟體結構

這是我們的定義中的第一個且最明顯的意思。結構其實是一組藉由關係來互相連結的元素。軟體系統是由許多結構組成的，沒有任何一個結構可以聲稱它自己是架構。我們可以將結構分門別類，並且用這些類別來思考架構。架構性結構（architectural structure）可以分成三種實用的類別，它們在你設計、記錄和分析架構時將發揮重要的作用：

1. 組件和連結結構（component-and-connector structure）
2. 模組結構（module structure）
3. 分配結構（allocation structure）

我們將在下一節深入說明這些結構類型。

儘管軟體是由無數的結構組成的，但並非所有結構都是架構性的。例如，將包含「z」的一行原始碼從最短排到最長是一種軟體結構，但這種結構既無聊，也不是架構性的。能夠協助你推理系統和系統屬性的結構才是架構性的，這種推理與關係人重視的系統屬性有關，那些屬性包括系統實現的功能、系統遇到錯誤或別人試圖讓它停止運作時維持有效運行的能力、對系統進行特定更改的容易程度或困難程度、系統回應用戶請求的能力，以及許多其他屬性。我們將在本書花很多篇幅探討架構與這種**品質屬性**之間的關係。

因此，架構性結構既不是固定的，也不受限制。「架構性」取決於你的背景有哪些事情可幫助你推理系統。

架構是一種抽象

由於架構是由結構組成的，而結構是由元素[1]和關係組成的，因此架構是由軟體元素，以及那些元素彼此的關係組成的。這意味著，架構會明確地、刻意地忽略與元素有關的某些資訊，那些資訊對系統的推理而言並不重要。因此，架構是系統的**抽象**，該抽象選擇了某些細節，並掩蓋其他的細節。在現代系統中，元素是透過介面來互動的，且介面將元素的細節分成公用的和私用的部分，架構關注的是公用的部分，元素的私用細節（只和內部實作有關的細節）不是架構性的。這種抽象對駕馭架構的複雜性來說至關重要：我們根本沒辦法不斷處理所有的複雜性，也不想這樣做，我們想要（也需要）讓系統的架構比系統的每一個細節更容易理解許多。就算系統不大，你也不可能將每一個細節記起來，架構就是為了讓你不必記憶細節。

架構 vs. 設計

架構是一種設計，但設計不一定是架構，也就是說，許多設計決策不會被架構限制（畢竟架構是一種抽象），而是由下游設計者甚至實作者決定的。

1 本書不打算區分模組（module）和組件（component），因此用「元素（element）」來代表它們。

每一個軟體系統都有軟體架構

每一個系統都有架構，因為每一個系統都有元素和關係，但是，這不代表任何人都知道架構，也許設計系統的人都已經不在了、文件已經遺失了（或從未製作）、原始碼已經找不到了（或從未交付），我們手邊只有可執行的二進制碼。從這裡可以看出系統的架構和架構的**表象**（*representation*）之間的差異。因為架構可獨立於說明文件或規格存在，所以**架構文件**非常重要，我們將在第 22 章說明。

並非所有架構都是好架構

我們的定義無關一個系統的架構是好是壞。架構可能協助系統實現重要需求，也可能阻礙系統。如果你認為試誤法不是選擇系統架構的最佳手段（試誤法就是隨機選擇一個架構，用它來建構系統，然後進行表面性的修改，期望得到最佳成果），那麼**架構設計**（第 20 章）和**架構評估**（第 21 章）就非常重要了。

架構包含行為

每一個元素的行為都是架構的一部分，因為該行為可以協助你推理系統。元素的行為體現了它們如何彼此互動，以及如何和環境互動，這顯然是我們的架構定義的一部分，而且會影響系統的屬性，例如它的執行期性能。

行為有一些層面不是架構師關注的事項，儘管如此，如果元素的行為會影響整個系統的可接受度，那麼該行為就必須視為系統的架構設計的一部分，而且必須記錄在案。

系統與企業架構

系統架構與企業架構是與軟體架構有關的兩門學問，這兩門學問所關注的事情都比軟體更廣泛，而且可以透過建立限制條件來影響軟體架構，軟體系統和架構師必須遵守那些條件。

系統架構

系統的架構是系統的一種體現，它可以將功能對應到硬體與軟體組件上、將軟體架構對應到硬體架構上，以及描述人類如何與組件互動。也就是說，系統架構涉及硬體、軟體與人類。

例如，架構可影響不同的處理器所負責的功能，也會影響連接那些處理器的網路類型。軟體架構決定功能如何建構，以及在各個處理器裡面的軟體程式如何互動。

因為軟體架構對應到硬體和網路組件，所以你可以用軟體架構的文件來推理性能和可靠性等品質。你可以用系統架構的文件來推理其他的品質，例如電力消耗、重量和物理尺寸。

在設計特定的系統時，系統架構師和軟體架構師經常針對功能的分配進行協商，從而約束軟體架構。

企業架構

企業架構描述了組織流程、資訊流、個人和組織子單元的結構與行為。企業架構不一定包含計算機資訊系統（顯然，在計算機出現之前的組織就已經有符合上述定義的架構了），但如今，如果沒有資訊系統的支援，除非企業規模極小，否則很難想像企業架構如何運作。因此，現代的企業架構關注的是軟體系統如何支援企業的商業流程和目標，裡面通常有一個流程決定企業應支援具有哪些功能的哪些系統。

例如，企業架構會指定讓各種系統用來互動的資料模型。企業架構也會指定企業的系統與外界系統互動的規則。

軟體只是企業架構關注的事項之一。企業架構經常處理的另外兩個問題是決定人們如何使用軟體來執行商業流程，以及定義計算環境。

協助不同系統互相溝通的軟體基礎設施，以及和外部世界進行溝通的軟體基礎設施有時也被視為企業架構的一部分，有時它被視為企業內部的系統之一（無論如何，該基礎設施的架構都是一種軟體架構！），這兩種觀點會讓與基礎設施有關的人有不同的管理結構和影響範圍。

這些學科在本書的討論範圍之內嗎？是的！
（嗯…其實是沒有）

系統與企業提供了製作軟體架構的環境，也對它施加限制。軟體架構必然位於系統與企業之內，是實現組織的商業目標的焦點。「企業與系統架構」和「軟體架構」有很多共同點，它們都可以設計、評估和記錄；它們都是為了回應需求，目的都是為了滿足關係人；它們都是由結構組成的，因此也是由元素和關係組成的；它們都有一系列的模式供各自的架構師使用…等。因此，在某種程度上，這些架構與軟體架構有許多共同點，它們都在本書的討論範圍之內。但是它們和所有的科技學科一樣，也有各自的專業術語和技術，我們將不介紹那些，坊間有大量的其他資源可供參考。

1.2 架構性結構和觀點

因為架構性結構是我們的定義的核心，也是看待軟體架構的核心，所以本節將更詳細地介紹這些概念。第 22 章還會更加深入地研究這些概念，我們將在那一章討論架構記錄。

架構性結構在自然界有對應的結構。例如，神經科醫生、骨科醫生、血液科醫生和皮膚科醫生對人體的結構有不同的觀點，如圖 1.1 所示。眼科醫生、心臟科醫生和足科醫生則專注於特定的子系統。運動治療師和精神科醫師專注於整體行為的不同層面。這些觀點可以用不同的圖片來表達，它們的屬性也大不相同，但它們有固有的關係並且互相連結：它們一起描述了人體的架構。

架構性結構在人類的工作中也有對應的結構。例如，水電工、冷暖氣機安裝工人、屋頂工人和鷹架工人分別關心建築物中的不同結構。你很容易就可以了解各種結構所關心的品質。

軟體也是如此。

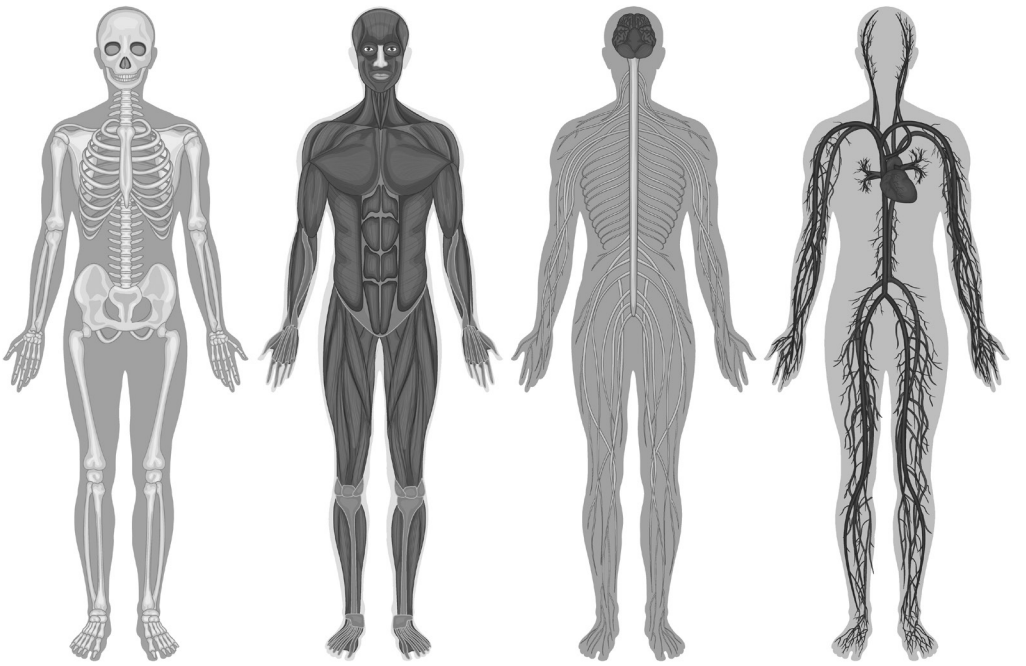

圖 1.1 生理結構圖

三種結構

架構性結構可分為三大類，取決於元素所展現的廣泛性質，以及它們支援的推理類型：

1. **組件與連結（*C&C*）結構**關注元素在執行期為了執行系統的功能而互動的方式。這種結構用一組具備執行期行為的元素（組件）與互動（連結）來描述系統的組成。這種結構中的組件是主要的計算單元，它可能是服務、對等體（peer）、用戶端、伺服器、過濾器，或其他類型的執行期元素。連結是組件之間的通訊媒介，例如 call-return、程序同步運算子（process synchronization operator）、管道（pipe）…等。C&C 結構可協助回答下列問題：

- 什麼是主要的執行組件？它們在執行期如何互動？
- 主要的共享資料儲存體有哪些？
- 系統有哪些部分是複製品？
- 資料如何在系統中傳遞？
- 系統的哪些部分可以平行運行？
- 系統的結構會不會在執行時改變？如果會，如何改變？

廣義而言，這些結構很適合回答與系統的執行期屬性有關的問題，例如性能、資訊安全、妥善性…等。

C&C 結構是最常見的一種結構，但其他兩種結構也很重要，不容忽視。

圖 1.2 以非正式的標記法來描繪一個系統的 C&C 結構，圖例是那些圖案的意思。這個系統有一個共享的儲存體可讓伺服器使用，它也有一個管理組件。操作用戶端的出納員可以和帳號伺服器互動，也可以透過發布 / 訂閱連結來互動。

2. **模組結構**（*module structure*）將系統分為許多實作單位，本書將那些單位稱為**模組**。模組結構展示了系統如何以一組程式碼或資料單元組成，你必須建構或取得裡面的程式碼。模組都有特定的計算職責，程式設計團隊根據它們來分配工作。在任何模組結構中，元素都是某種類型（也許是類別、程式包、階層，或只是功能的劃分，全都是實作單元）的模組。模組就是系統的靜態觀點，模組都被指定了功能職責領域；這些結構不太強調軟體的執行期表現。模組的實件包括程式包（package）、類別與階層（layer）。在模組結構內，模組間的關係有 uses、generalization（或「is-a」），以及「is part of」。圖 1.3 與 1.4 分別為模組元素與關係的範例，它們使用 Unified Modeling Language（UML）標記法。

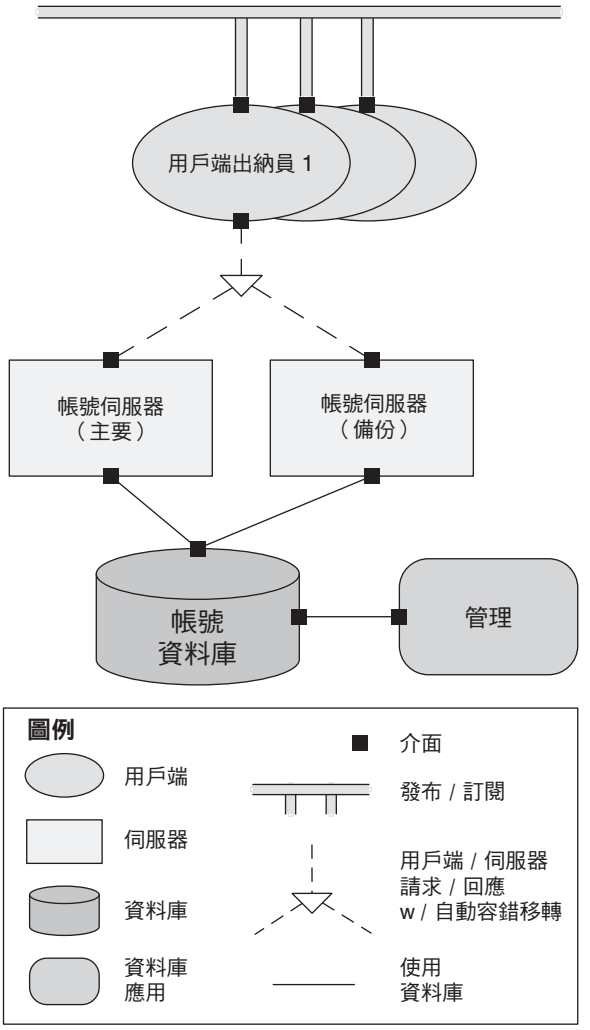

圖 1.2 C&C 結構

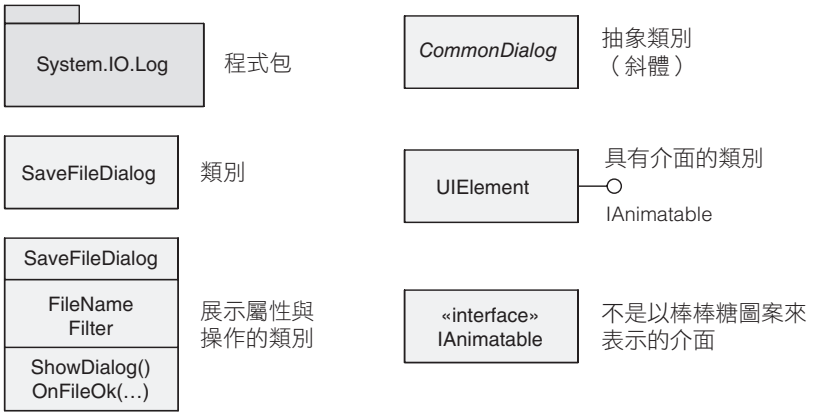

圖 1.3　以 UML 來描繪模組元素

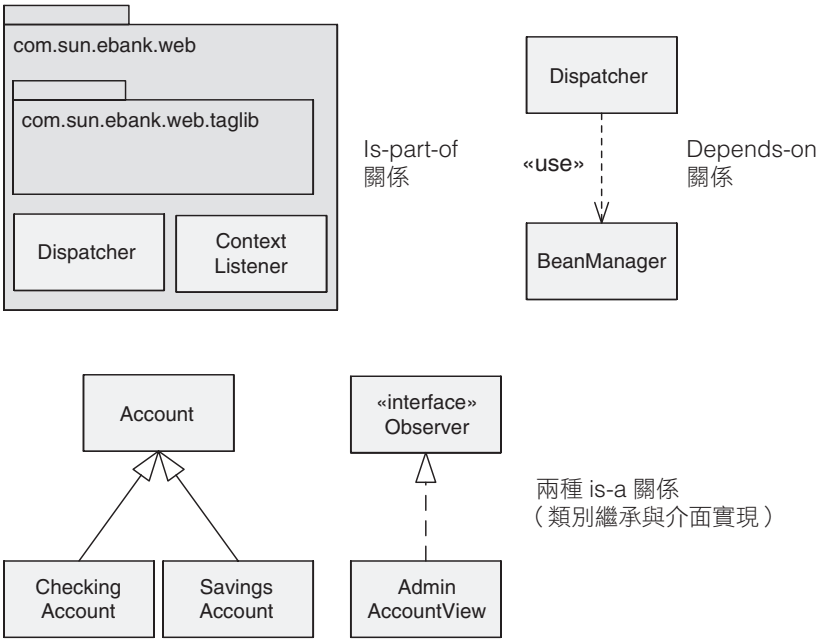

圖 1.4　以 UML 來描繪模組關係

模組結構可回答這類的問題：

- 每一種模組的主要功能是什麼？
- 模組可以使用哪些其他的軟體元素？
- 它使用和依靠哪些其他的軟體？
- 哪些模組與其他模組有 generalization 或 specialization（即繼承）關係？

模組結構可直接傳達此資訊，它們也可以用來回答關於「改變各個模組的職責對系統有何影響」的問題。因此，模組結構是推理系統可修改性的主要工具。

3. **分配結構**可將軟體結構對應至系統的非軟體結構，例如系統的組織，或開發、測試與執行環境。分配結構可回答這類問題：
   - 各個軟體元素是在哪個處理器上執行的？
   - 在開發、測試和系統建構期間，每個元素被存放在哪個目標或檔案內？
   - 各個軟體元素被分配給哪個開發團隊？

實用的模組結構

實用的模組結構有：

- **分解結構**（*decomposition structure*）。其單元是彼此間有「is-a-submodule-of」（…的子模組）關係的模組，這種結構展示了如何將模組反覆地分解成更小的模組，直到容易了解的規模為止。這個結構裡面的模組就是設計的起點，架構師必須列出每一個軟體單元將要做哪些事情，並將每一個項目指派給一個模組，以便進行後續（更詳細）的設計和最終的實作。模組通常有相關的產品（例如介面規格、程式碼和測試計畫）。分解結構在很大程度上決定了系統的可修改性，例如，你的修改是否在一些模組的範圍之內（模組最好很少）？開發專案的組織（organization）通常以這種結構為基礎，包括文件的結構、專案的整合，以及測試計畫。圖 1.5 是分解結構的範例。

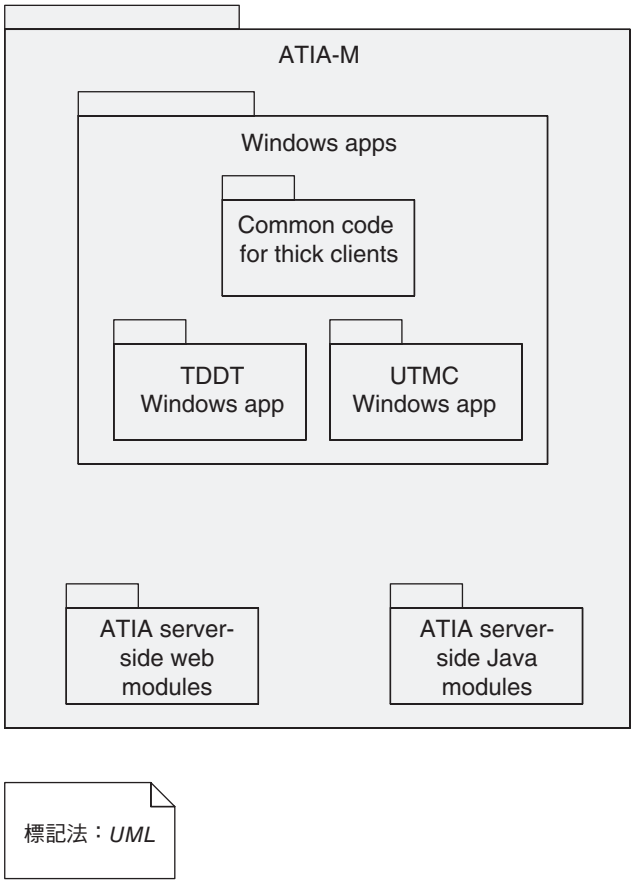

圖 1.5 分解結構

- *uses* 結構（*uses structure*）。在這種很重要但經常被忽略的結構中，單元可能是模組，也可能是類別。單元之間有 *uses* 關係，這是一種特殊的依賴關係。如果一個軟體單元在另一個單元的正確版本（而非 stub）存在的情況下才能正確運行，我們說第一個單元 uses 第二個單元。我們用 uses 結構來設計可以擴展並添加功能的系統，或可從中提取有用的功能子集合的系統。當你可以輕鬆地建立系統的子集合時，你就可以進行遞增開發（incremental development）。這個結構也是衡量社交債務（social debt，即團隊之間的實際溝通量，而不僅僅是應該進行的溝通量）的基

礎，因為它定義了哪些團隊應互相溝通。圖 1.6 是 uses 結構，它展示為了讓 admin.client 模組存在而必須依序（increment）存在的模組。

- **階層結構**（*layer structure*）。在這個結構裡面的模組稱為階層。階層是抽象的「虛擬機器」，它們透過受管理的（managed）介面來提供一組內聚的服務。階層可讓你以受管理的方式來使用其他的階層，在嚴格分層的系統中，一個階層只能使用一個其他階層。這種結構可為系統帶來可移植性，也就是更換底下的虛擬機器。圖 1.7 是 UNIX System V 作業系統的階層結構。

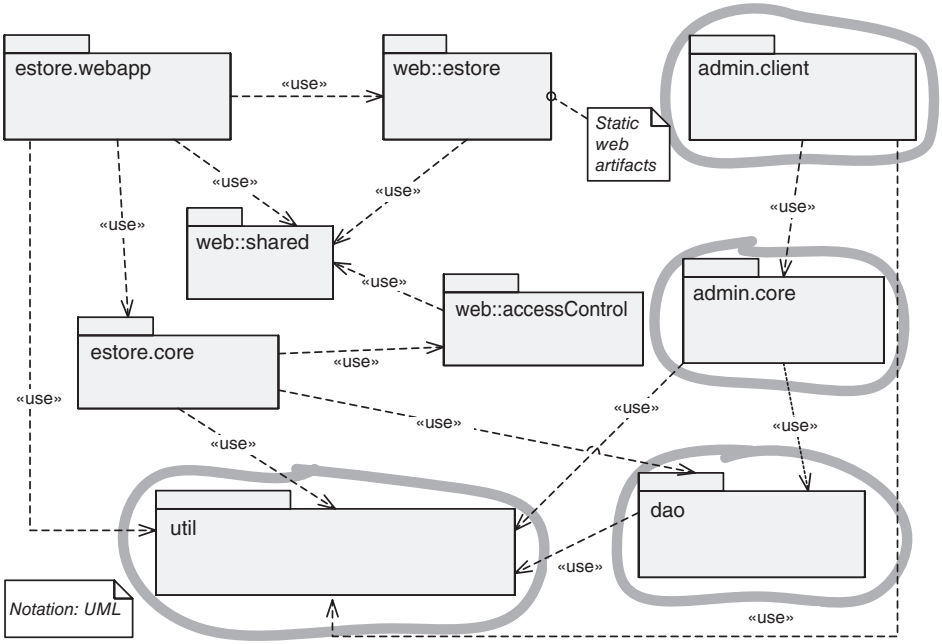

圖 1.6 uses 結構

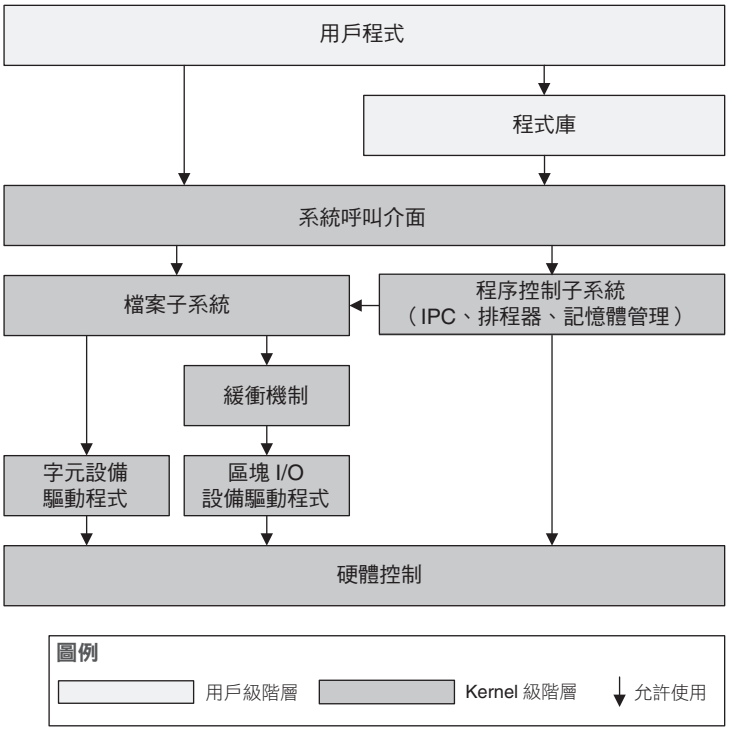

圖 1.7 階層結構

- **類別（抽象化，*generalization*）結構**。在這個結構裡面的模組稱為類別，它們之間有「inherits-from（繼承）」或「is-an-instance-of（為…的實例）」的關係。這種觀點可協助推理一群相似的行為或能力，以及參數化差異（parameterized difference）。類別結構可讓你推理重複使用與遞增功能。如果專案有按照物件導向分析與設計程序來撰寫的文件，它通常就是這種結構。圖 1.8 是取自一個架構專家工具的抽象化結構。

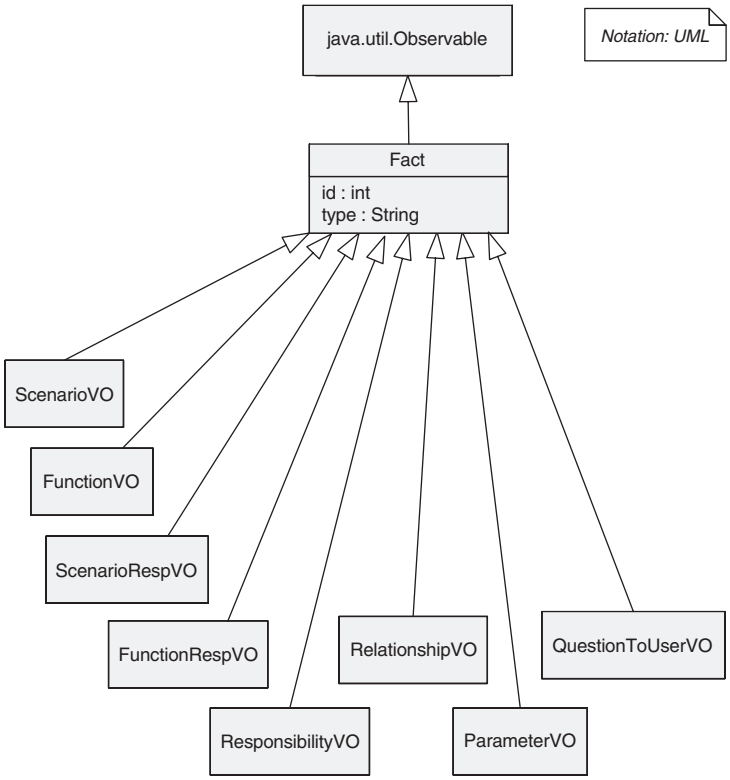

圖 1.8 抽象化結構

- **資料模型**。資料模型描述了資料實體（data entity）和它們之間的關係的靜態資訊結構。例如，在銀行系統中，實體通常有 Account（帳戶）、Customer（客戶）與 Loan（貸款）。帳戶有許多屬性，例如帳號、類型（儲蓄或支票）、狀態和當前餘額。帳戶之間的關係可能說明一位客戶有一或多個帳戶，以及一個帳戶與一或多位客戶有關。圖 1.9 是資料模型範例。

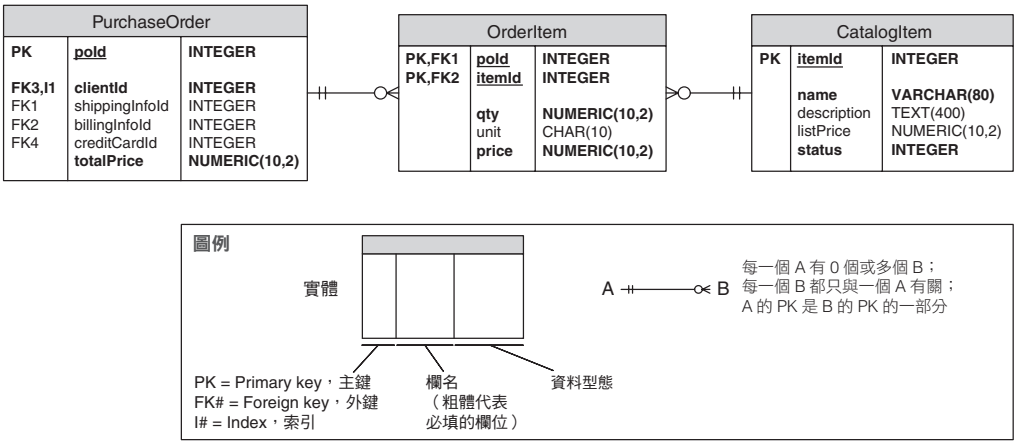

圖 1.9 資料模型

一些實用的 C&C 結構

C&C 結構展示系統的執行期觀點。在這些結構中，剛才介紹的模組都被編譯成可執行的形式。因此，所有的 C&C 結構都與模組結構正交（orthogonal），它們處理的是運行中的系統的動態層面。例如，一個程式碼單元（模組）可能被編譯成一個服務，且該服務在可執行環境中被複製上千次。或者，1,000 個模組可能被編譯和連結在一起，以產生一個執行期的可執行檔（組件）。

在所有的 C&C 結構中的關係都是**附著**，它們展示了組件與連結如何連接在一起（連結本身可能是你熟悉的結構，例如「invokes」）。實用的 C&C 結構有：

- **服務結構**。這裡面的單元是服務，它們透過服務協調機制（例如訊息）來進行交互操作。服務結構是一種重要的結構，可協助系統工程師將獨立開發的組件組合起來。
- **並行（*concurrency*）結構**。這種 C&C 結構可讓架構師釐清平行化的機會，以及可能發生資源爭奪的位置。這種結構的單元是組件，連結是它們的溝通機制。組件都會被放入「邏輯執行緒（logical thread）」。邏輯執行緒是一系列的計算，接下來的設計程序可能會將它們放入一個單獨的實體執行緒。我們會在設計流程的早期使用並行結構來發現與管理與並行執行有關的問題。

實用的分配結構

分配結構定義了 C&C 或模組結構的元素如何對應至非軟體事物，那些事物通常是硬體（可能虛擬化）、團隊和檔案系統。實用的分配結構有：

- **開發結構**。開發結構展示了如何將軟體分配給硬體處理元素和通訊元素。這裡的元素是軟體元素（通常是 C&C 結構中的程序）、硬體實體（處理器）和通訊路徑。在這個結構裡面的關係有「allocated-to」，展示軟體元素位於哪個物理單元，以及動態分配時的「migrates-to」。這個結構可以用來推理性能、資料一致性、資訊安全和妥善性。它在分散式系統中特別重要，是實現易部署性（見第 5 章）的關鍵結構。圖 1.10 是以 UML 繪製的部署結構。

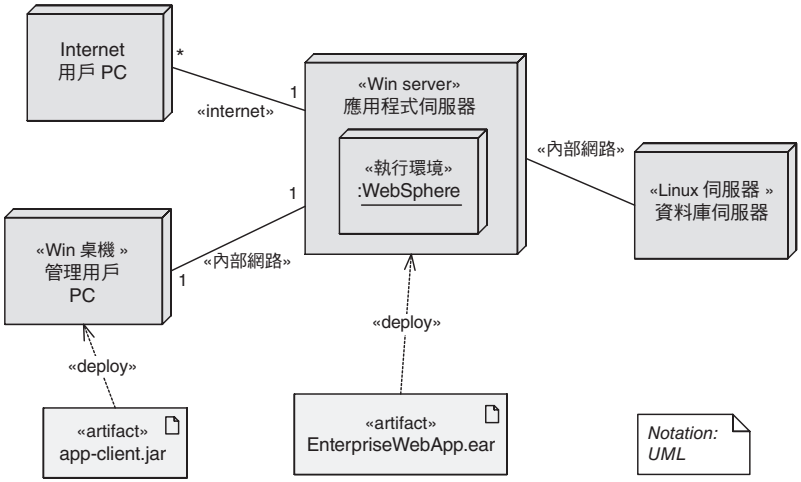

圖 1.10 部署結構

- **實作（*implementation*）結構**。這種結構告訴你如何將軟體元素（通常是模組）對應至系統的開發、整合、測試或組態控制環境中的檔案結構。這種結構在管理開發活動和組建程序時非常重要。
- **工作分配結構**。這種結構將實作模組和整合模組的職責分配給執行這些工作的團隊。在架構中加入工作分配結構可清楚地展示「讓某人進行某項工作」如何影響架構層面，以及如何影響管理層面。架構師可知道各個團隊需要哪種專業技能。例

如，Amazon 讓一個團隊負責一項微服務就是一種工作分配結構的表徵。在大型的研發專案中，有一種很實用的做法是找出具有相同功能的單元，並將它們分配給一個團隊，而不是讓每一個需要那些單元的人實作。這種結構也決定了團隊之間的主要溝通途徑，例如使用定期網路會議、維基、email 名單…等。

表 1.1 是這些結構的摘要，它列出了元素的意義，以及在各個結構內的關係，並說明每一種結構的用途。

結構之間的關係

以上的每一種結構都提供不同的系統觀點與設計把手（design handle），每一種結構本身都有其效用。雖然不同的結構提供不同的系統觀點，但它們不是獨立的，一個結構的元素與另一個結構的元素有關，我們必須了解這些關係。例如，在分解結構裡面的模組可能是某種 C&C 結構的一個組件、一個組件的一部分，或多個組件，反映了它在執行期的另一面。一般來說，不同結構之間有多對多關係。

圖 1.11 簡單地展示兩個結構之間的關係。左圖是一個小型的用戶端 / 伺服器系統的模組分解視圖。這個系統有兩個必須實作的模組：用戶端軟體與伺服器軟體。右圖是同一個系統的 C&C 視圖。在執行期，這個系統會運行十個用戶端，它們會訪問伺服器。因此，這個小系統有兩個模組與十一個組件（與十個連結）。

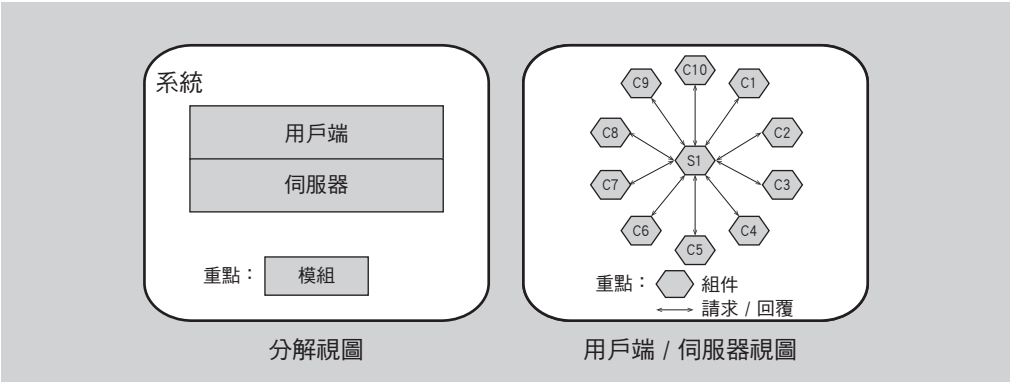

圖 1.11　用戶端 / 伺服器系統的兩個視圖

表 1.1　實用的架構性結構

	軟體結構	元素類型	關係	適用於	影響的品質問題
模組結構	分解	模組	…的子模組	資源分配與專案結構設計和規劃；封裝	可修改性
	uses	模組	uses（即，對象必須正確地存在）	設計子集合與擴展軟體	「可子集合性（subsetability）」、可擴展性
	階層	階層	允許使用…的服務；提供抽象給…	遞增開發；在「虛擬機器」上面實作系統	可移植性、可修改性
	類別	類別、物件	為…的實例；為…的廣義	在物件導向系統中歸納共同點；規劃擴展功能	可修改性、可擴展性
	資料模型	資料實體	{ 一 , 多 }- 對 -{ 一 , 多 }；抽象化；特殊化	為了取得一致性和提升性能，設計全域性的資料結構	可修改性、性能
C&C 結構	服務	服務、服務註冊	附著（透過訊息傳遞）	調度分析；性能分析；強固性分析	可互用性、妥善性、可修改性
	並行	程序、執行緒	附著（透過通訊和同步機制）	找出發生資源爭奪的位置、平行化的機會	性能
分配結構	部署	組件、硬體元素	分配給；遷移至	將軟體元素對應至系統元素	性能、資訊安全、能源、妥善性、易部署性
	實作	模組、檔案結構	儲存於	組態控制，整合，測試活動	部署效率
	工作分配	模組、組織單位	分配給	專案管理，善用專業技能和資源，管理共同點	部署效率

雖然分解結構與用戶端 / 伺服器結構的元素之間的對應關係很明顯，但是這兩種觀點有非常不同的用途。例如，右邊的觀點可以用來進行性能分析、瓶頸預測、網路流量管理，但是這些工作都很難或不可能用左邊的觀點來做（在第 9 章中，我們將學習 map-reduce 模式，它將簡單、相同的功能複本分散在數百個或數千個處理節點之間，也就是讓整個系統使用一個模組，但是讓每個節點使用一個組件）。

有的專案將一種結構當成主要結構，並在可行的情況下，用主要結構來製作其他的結構。它們的主要結構通常是模組分解結構，這是有原因的：它往往可以產生專案結構，因為它反映了開發團隊結構。有些專案的主要結構是一種 C&C 結構，展示了系統的功能或重要的品質屬性如何在執行期實現。

越少越好

並非所有系統都需要考慮許多架構性結構。系統越大，這些結構之間的差異就越明顯，但是小型的系統通常可以使用較少結構，例如只要使用一個 C&C 結構即可，不需要使用每一種 C&C 結構。如果程序只有一個，你可以將程序結構摺疊為一個節點，不需要在設計中明確地展示。如果系統沒有分散（distribution）（也就是說，如果系統是在單一處理器上實作的），它的部署結構就不重要，因此不需要考慮。一般來說，除非設計和記錄一種結構可以帶來正面的投資回報，否則就不需要做，正面的投資回報通常與減少開發或維護成本有關。

該選擇哪些結構？

我們已經簡單地介紹了一些實用的架構性結構了，此外當然還有其他結構。架構師該使用哪些結構？架構師該記錄哪些結構？答案當然不是所有結構。你應該考慮各種結構如何讓你發現和利用系統最重要的品質屬性，然後選擇最能夠幫你提供那些屬性的結構。

架構模式

有的架構元素是為了解決特定的問題而組成的，隨著時間的過去，人們發現這些組合在許多領域中非常有用，於是將它們記錄下來並流傳出去。這些架構元素組合提供了套裝策略來讓你解決某些系統問題，這種組合稱為模式。本書的第二部分將詳細討論架構模式。