依據使用案例的人力評估

John Smith

Rational Software 白皮書

TP 171, 10/99

Rational®
the software development company

目錄

問題

直覺來看，似乎可以依據使用案例模型的性質來形成開發所需要的規模和人力的評估。畢竟，使用案例模型會擷取功能需求，因此，是否不應該有一個相等於功能點的使用案例？這裡有幾個難題：

- 使用案例規格樣式和形式有許多變化，令人很難定義測量值 — 例如，有人可能想要測量使用案例的長度
- 使用案例應該代表外部參與者的系統視圖，因此，500,000 軟體程式碼行 (software lines of code, sloc) 系統的使用案例，與針對 5,000 sloc 子系統而寫的使用案例的***層次***大不相同（Cockburn97 討論層次和目標的概念）。
- 各使用案例有不同的複雜性，既明確寫出，又隱含在必要的實現化中。
- 使用案例應該從參與者的觀點說明行爲，但這可能很複雜，尤其當系統有狀態時（大部分是如此）。因此，要說明此行爲可能需要系統的模型（在完成任何實現化之前）。爲了擷取行爲的本質，這樣做可能會導致太多層次的功能分解和詳細資料。

因此，一定需要某種使用案例實現化才能進行評量嗎？或許對於評量直接來自使用案例的期望過高，而在功能點和使用案例點概念之間畫上平行線也會誤導。功能點計數的計算還是需要系統模型。從使用案例說明衍生功能點需要使用案例表示式的層次統一，而且唯有當實現化開始出現，人們才會對功能點計數有更多信心。Fetcke97 說明使用案例到功能點的對映，但同樣地，使用案例的層次必須適當，對映才有效。其他方法使用類別/物件式測量作爲來源，例如，PRICE Object Points (Minkiewicz96)。

其他工作

有不少的工作量放在說明和形式化使用案例上—Hurlbut97 有一項很好的調查。從使用案例衍生評估測量的工作就比較少。Graham95 和 Graham98 對使用案例有很嚴厲的批評（但我並不完全瞭解爲何他認爲自己的想法與使用案例差距如此之大），並提出「作業 Script」的想法作爲克服使用案例問題的方式，包括其不同的長度和複雜性。Graham 的「原子性的作業 Script」是「作業點」測量值集合的基礎。原子性的作業 Script 的問題在於它非常低階：根據 Graham 的說法，在觀念上，它應該是單一句子，而不能只使用網域專有名詞進一步分解。Graham 的「根作業」包含一或多個原子性的作業 Script，每一個根作業對應到剛好一項系統作業：在起始計劃的類別中。(Graham98)。對我而言，這些根作業似乎很像低階使用案例，而原子性的作業 Script 就像這種使用案例中的步驟。同樣地，層次問題仍然存在。

其他工作是由 Karner (Karner93)、Major (Major98)、Armour 和 Catherwood (Armour96) 與 Thomson (Thomson94) 完成。Karner 文章裡假定有一種方法來計算使用案例點，但同樣假設使用案例是以類別可實現的方式來表達（亦即，在比子系統更細微的層次上）。

因此，我們是否應該避免使用使用案例來評估，而改爲依賴新興的分析和設計實現化？其問題在於它拖延評估的功能，令選擇使用此技術的專案經理非常不悅—提早評估是必要的，而且也必須使用其他方法。基於規劃用途，專案經理最好能夠早點取得評估，然後一個個反覆來***修正***它們，而不是拖延評估及毫無計劃地進行。

本白皮書所述的是一種架構，任何層次的使用案例都可以用來形成人力評估。爲了表明看法，描述了一些簡單的標準結構，以及相關聯的維度和規模，這些都有一些經驗基礎。本白皮書充滿大膽（或赤裸裸）的推測，因爲在缺乏此領域的成果和資料之下，我找不到別的方式進行。我在構想上利用了交互連接系統的系統。

接下來，我很快脫離主題，記下一些讓我走上這一條路的背景想法。

避免功能分解？

功能分解的想法對許多從事軟體開發的人而言，似乎是一項詛咒。而我個人在功能分解方面的經驗（有三千個基本元素在一個超大型資料流程圖中轉換，深度達五或六層，除了基礎架構層次之外，在不考慮架構之下完成），也沒有讓我對它感到樂觀。不過，此案例的問題不只是功能分解，還有等到達到功能基本元素層次才說明流程的概念，這時候規格長度應該不到一頁。

其結果很難理解—更高層次所需要的行爲是如何從這些基本元素的轉換中發生的，難以分辨。此外，功能結構應該如何對映到符合效能和其他品質需求的實體結構，也不明確。因此，我們分解又分解，直到達到我們可以解決問題的層次（基本元素層次）爲止，但基本元素一起工作的結果是否真的達到更高層次的目標，這點並不清楚或可以得到證實，真是自相矛盾。此方法絕不考慮非功能需求。從整體上來看，不只是基礎架構（通訊、作業系統等等），架構應該與分解同時發展，彼此應該互相影響對方。

Bauhaus 所謂的「形式遵循功能」，其內容是什麼？機能主義者的設計方式的確帶來一些好處，但也有一些壞處，例如到處使用平頂式屋頂。如果您只在乎屋頂的功能，而使設計完全屈就於一個遮蔽居住者的屋頂而已，其結果至少在某些地方會令人不滿意。這樣的屋頂很難防水；反而會堆積很多雪。

現在這些問題雖然可以獲得解決***，但其花費高於您選擇不同的設計所需的費用。***因此，雖然是老生常談，形式仍然應該遵循*需求*—不論是功能和非功能的需求全部都是，後者更包含了美學意涵。建築師的問題通常在於非功能需求的表達不夠，多半依賴建築師「想當然爾」的經驗。因此，如果架構完全由功能分解驅動，則功能分解不好—如果向下幾層進行分解，且功能基本元素一對一對映到「模組」—並定義其介面。

諸如這類的考量讓我相信，***在架構工作之前將***使用案例向下分解到一些正常化層次（可經由類別合作來實現）是沒道理的。如果系統頗具規模，當然會發生分解（請參閱 Jacobson97），但分解的準則和工程流程很重要—特定的功能分解不夠理想。

系統考量

系統工程師執行功能分析、分解和配置（在合成設計時）—但功能不是架構的唯一動因—專業工程師團隊將致力於評定替代設計。IEEE Std 1220 是系統工程流程的應用和管理標準，它在 *6.3 功能分析*這一節的 *6.3.1 功能分解*這一小節描述功能分解的使用，以及在 *6.5 合成*這一節描述系統產品解決方案。特定重要的是 *6.5.1 群組和配置功能*與 *6.5.2 實體解決方案替代方案*這兩小節。在 *6.3.1* 這一節，它說執行分解是爲了清楚***瞭解***系統必須完成什麼，***通常一個分解層次已經足夠。***

請注意，功能分解的目的不是要將系統塑型（這是合成的目的），而是要瞭解及傳達系統必須執行的動作 —功能模型是達成此目的之有效方式。在合成時，子功能會配置到解決方案結構中，然後評估解決方案—將所有其他需求列入考量。這種方法和多層次功能分解之間的差異，是在決定下一層的行爲是否需要進一步修正及配置給低階元件之前，您會在每一個層次描述必要的行爲並找出實作它的解決方案。

由此得到一個結論，以幾百個使用案例來描述任一層次的行爲是不必要的作法。足以涵蓋所描述項目—系統、子系統、類別—之行爲的外部使用案例（和相關聯的實務）數目很少也無所謂。我應該以外部使用案例來說明我的看法。以數個子系統組成的一個系統爲例，這些子系統又包含了類別。描述系統行爲及其參與者的使用案例，我稱之爲外部使用案例。子系統也可以有自己的使用案例—這些使用案例是屬於系統內部，但對於子系統卻是外部的。最後用來建構超大型（譬如 1,000,000+ 程式碼行）系統的使用案例總數（包括***外部和內部***）可能高達數百個，因爲此規模的系統將建構成系統的系統，或至少是子系統的系統。

關於結構和規模的假設

使用案例數

在 Rational 娟 oftware 方面，我們一般會建議使用案例數目不要多（譬如 10–50），而且我們觀察到大量使用案例（譬如超過 100）會陷入功能分解，導致使用案例對參與者毫無價值。然而我們也發現，在實際專案中擁有大量使用案例並非全是壞事—它們混合了各個層次—例如，在 Rational 內部電子郵件中，作者引用 Ericsson 的例子：

> Ericsson 是大部分新一代電話交換機的塑造者，規模預估爲 +600 個人年（在尖峰期，更高達 3–400 個開發人員）、200 個使用案例 ***（使用多個使用案例層次，請參閱「交互連接系統的系統」）*** (my italics)

對於有 600+ 個人年的系統（這個規模到底多大？ 1,500,000 C++ 程式碼行？），我猜想使用案例分析會在子系統的上一層停止（也就是說，如果將子系統定義爲 7000–10000 程式碼行），否則這個數字仍然會居高不下。

因此，我還是認爲少數的 *外部*使用案例已足夠。爲了符合我提出的結構和維度，我主張 ***10 個外部使用案例***，每一個有 ***30 個相關聯的實務***[1] 就足以說明行爲[2]。如果在真實範例中，使用案例的數目超過 10 個，而且它們在該層次的確是外部的，則所描述之系統大於相對應的標準形式。我會試著在本白皮書後面，爲這些數目的合理化提供一些支持的論據。

結構階層

提出的結構階層爲：

4 — SystemOfSystems

3 — System

2 — SubsystemGroup

1 — Subsystem

0 — Class

類別和子系統是定義在 UML 中；在 UML 中較大的集合體是子系統（包含子系統）。我以不同方式來命名它們，以方便討論。subsystemGroup 集合體是 CSCI 型規模，從軍事標準得知此專有名詞的人喜歡用 2167 或 498（子系統是 CSC，類別是 CSU）。還記得，對於 Ada 建構應該對映到哪個層次爭論經過 2167 天之後，當一切塵埃落定，Ada 套件通常對映到 CSU。我並非建議系統必須嚴格符合此階層—層次是混合的—但階層可讓我推論規模對每個使用案例人力的影響。

每個層次會有使用案例（雖然可能不是針對個別類別），但可不是一大堆雜七雜八的東西，而是該層次每一個元件（亦即，subsystem、subsystemGroup 等等）的使用案例。[3]我前面主張每一個層次的每一個元件應該有 10 個使用案例。如果使用案例的說明平均 10 頁，這可能產生 100 頁的規格文件長度（加上差不多或少一點的非功能需求的數目）。這是 Stevens98 偏好的數目，接近 Royce98 建議的數字。但爲何要 10 個使用案例？爲了得到此結論，我由下往上推斷，根據我認爲每個子系統的類別數的合理大小、類別大小、作業大小等等。這些收集在一起，與下表的其他假設作爲參考。

作業大小	70 sloc
每個類別的作業數	12
每個子系統的類別數	8

[1] 在 UML1.3，有一個實務是這樣的：**情境：**說明行爲的特定動作序列。實務可用來說明使用案例實例的互動或執行。這裡使用它的第二個含意，即說明使用案例實例的執行。

[2] 請注意，此數字（實務數）是爲了反映使用案例的複雜性—我們不建議開發人員 ***一定***要爲每一個使用案例製作及撰寫 30 個實務—而是建議 30 個實務能夠擷取使用案例大部分重要行爲（即使經由使用案例會有更多路徑也一樣）。

[3] 有些評論家警告，四個層次的使用案例前景堪虞，但是要知道，這只是針對系統的系統，它通常是超大型的系統。在這樣的情況下，有四個層次的使用案例一點也不奇怪，尤其如果是由統包商（針對系統的系統）、轉包商（針對系統）、甚至是子系統的轉包商來完成工作的話，更不足爲奇。

每個 subsystemGroup 的子系統數	8
每個系統的 subsystemGroups 數	8
每個 systemOfSystems 的系統數	8
外部使用案例數（每個系統、子系統等等）	10
每個使用案例的實務數	30
每個使用案例說明的頁數[4]	10

我沒有很多以經驗爲依據的資料—字裡行間倒是有一些片段。Lorentz94 和 Henderson-Sellers96 有一些資料，我也有來自澳洲專案的一些資料，主要是軍用航空領域。無論如何，這個階段的重點是要讓架構大約定位在正確的地方。

階層中的元件大小

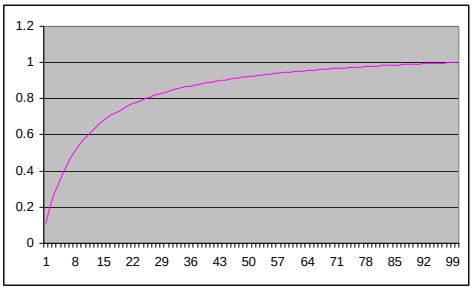
明知道有些人不喜歡這種方式，但我必須承認我在這裡使用了程式碼行。這些是 C++（或同等級的語言）程式碼行，因此它很容易回到功能點而失敗。

儲存器中的類別數和可表達的豐富行爲之間一定有某種關係存在。我選擇 8 個類別/子系統[5]、8 個 subsystem/subsystemGroup、8 個 subsystemGroup/system 等等。爲什麼是 8？

- 它是在 7 以內，加 2 或減 2。
- 因爲每個類別有 850 個 C++ sloc（各有 70 sloc 的 12 個作業），它會產生 ~7000 sloc 的子系統規模—可由一小組人員（假設 3–7 人）在 4–9 個月完成的一個功能/程式碼片段，它應該與範圍 300,000–1,000,000 sloc (RUP99) 的系統反覆長度協調。[6]

因此，可表達 8 個類別的行爲（從外部）的使用案例數是多少？哪些使用案例有凝聚性而曾經並存於子系統中？它不只是使用案例數，也是每一個使用案例決定豐富性的實務數。關於實務/使用案例擴充的準則並不多— Grady Booch 在 Booch98 表示：「從使用案例到實務之間有擴充因素。普通複雜的系統可能有幾十個使用案例擷取其行爲，每一個使用案例可擴充到數十個實務」，Bruce Powel Douglass 在 Douglass99 也說：「許多實務是完整闡述使用案例所必要的…—通常爲十幾個到幾十個」。我選擇了 30 個實務/使用案例—是「數十個」當中較少的，但 Rechtin（在 Rechtin91）說，工程師可以處理 5–10 個互動變數（基於此引數的用途，我將它解譯爲一個協同作業中的 5–10 個類別）和 10–50 個互動作業（我將它們解譯爲實務）。如此解譯之後，多個使用案例成爲此變數空間的多個實例。

因此，這 10 個使用案例，每一個有 30 個實務，總計 300 個實務（到後來會導致 ~300 個測試案例）足以涵蓋 8 個類別的重要行爲。是否有其他跡象顯示這是合理數字？如果適用 Pareto 的 80–20 規則，那麼，類別的 20% 將提供 80% 的功能，同樣地，80% 的功能將由每一個類別 20% 的作業來提供。如果保守一點，假設我們需要 20% 的類別來達到 75% 的功能，並透過此點建構 Pareto 分送（圖 1）。



[4] 本白皮書稍後會針對不同系統類別，對

[5] 我相信這種計數可代表分析—透過設計 數遞增，而作業大小和類別大小會相對減少。

[6] 對於較小型系統（比反覆次數少），子 寸—不過這需要小心控制，有可能需要交付 Stub。

圖 1：柏拉圖式分送

如果我們想要涵蓋 80% 的整體行爲，而 Pareto 規則適用於類別、作業和實務的數目，則我們需要每一個有 93%（0.93^3 是 0.8）的行爲涵蓋率—即每一個需要 50%；4 個類別和 5 個作業（=（12 減去 2 個建構子/解構子）/2）。已建構之節點樹狀結構的不同交叉數，代表 4 個類別 5 個作業的執行模式，每一個可以執行至數千個。我建構了一個，它含有每一個節點最多 3 個鏈結，並假設階層的頂端有 10 個作業（介面作業），而形成有三個層級的樹狀結構。這樣會產生將近 1000 個路徑或實務。因此有 500 個實務會產生 93% 的涵蓋面。如果有 300 個實務（使用相同假設），我們應該可以獲得大約 73% 的涵蓋面。檢查如何刪減樹狀結構來刪除冗餘行爲規格，這暗示著即使更小數字也足夠，視選擇的演算法而定。

處理這個問題的另一個方法是詢問 7000 個 C++ 的 sloc 應該會有多少個測試案例（從實務衍生）。這些測試超出單元測試層次，而根據 Jones91 和 Boeing 777 專案 (Pehrson96) 的一些證據顯示，這個數字至少是安全的，因爲它代表練習。這些來源建議這個數字在 250–280[7] 之間大約正確。在完全不同的層次上，Canadian Automated Air Traffic System (CAATS) 專案使用 200 個系統測試（私人通訊）。

使用案例大小

使用案例到底應該多大？大到足以呈現足夠的詳細資料，來實現所要的行爲—視其內部和外部的複雜性而定，這將與系統類型相關。這裡我們碰到系統內部動作應描述到何種程度的問題。若要從其外部行爲的說明中建置系統，顯然需要輸出與輸入相關。現在，比方說，如果行爲有歷程機密性而且複雜，則對系統內部及其採取的動作沒有某種概念性模型的話，將很難描述它。不過，請注意，這不一定說明系統內部如何建構 —任何滿足非功能需求且符合模型行爲的設計都辦得到。

UML1.3 提供的定義爲：「**使用案例 [類別]：**一系列動作的規格，包括系統（或其他實體）可執行的變式，與系統參與者之間有互動」。對於複雜行爲，可適度地採用此定義來包括內部動作—除非要延遲到實現化爲止—這又拉長了與一般使用者之間的距離。商業規則也應該納入使用案例中，來限制參與者的行爲；例如，在 ATM 系統中，銀行可以規定，不論帳戶餘額多少，單筆交易均不得提領超過 $500。

若以此方式解譯，則事件說明的使用案例流程可能在 2–20 字[8] 頁。在演算上簡單且具有簡單行爲的系統顯然不需要很長的說明。或許我們可以這麼說，簡單商務系統以 2–10 頁爲其特徵，平均爲 5 頁。較複雜的系統（商業和科學）爲 6–15 頁，平均爲 9 頁，而複雜指令和控制項爲 8–20 頁，平均爲 12 頁（這些比率反映出相同大小的系統的人力與系統類型的非線性關係），不過我並沒有資料可以支持這種說法。更具表達性的敘述形式，例如狀態機或活動圖，可能佔較少空間。我們仍傾向於強調文字，因此，我會先略過其他形式—反正也沒有什麼資料可參考。

在組織上與這些大小不同的開發應該將這些啓發所衍生的每個使用案例時數加倍（我建議加入 COCOMO 式成本動因，它是系統分類的觀察平均大小/建議平均大小—簡單的商務系統、更複雜的系統、指令和控制項等等）。

---

[7] 從 Rational 內部評論者的意見來看，這個數字對大部分非關鍵系統而言綽綽有餘；這些系統的每個使用案例少於 30 個實務。如果關於此數字及測試案例數與使用時發現的問題數之間的關係有更多資料的話，會很有意思。

[8] 請注意，這並不是硬性規定的上限；使用案例說明的長度將遵循某種統計分送，出現極端長度的機率較低。

使用案例大小的另一個層面是實務計數；例如，長度僅 5 頁的使用案例可能有容許許多路徑的複雜結構。同樣地，需要評估實務數，此數字與 30（我初次猜測的每個使用案例的實務數）的比例是作爲成本動因使用。

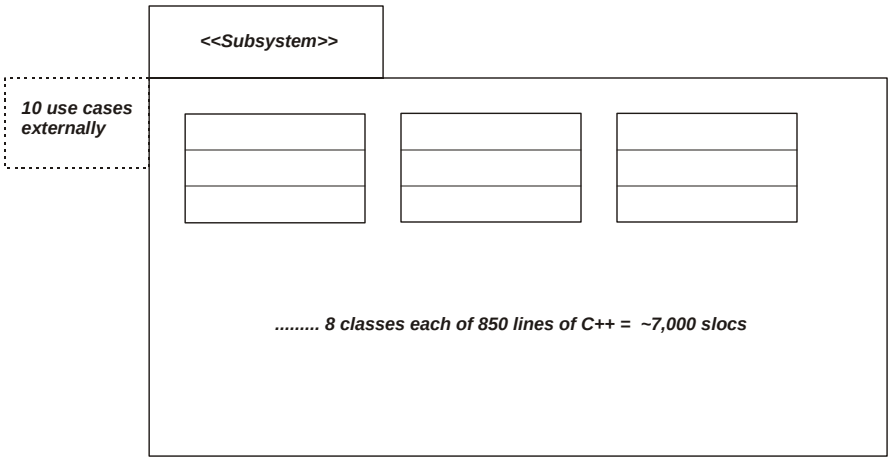
因此，除了增補規格之外，我們主張以 ~100 頁規格爲基礎的使用案例應足以供任何層次的外部規格使用。其範圍是從 20–200 頁（這些限制很模糊）。請注意，雖然最低層次的系統（屬於 subsystemGroup）的總計是 3–15 頁/ksloc（簡單商務系統）—12–30 頁/ksloc（複雜指令和控制項）。這似乎可解釋 Royce98 表格 14-9 之間的矛盾，其構件的頁數很少，但實際專案的觀察（尤其在防禦方面）卻產生一大張紙。這張紙來自不需要寫在書面上的規格層次—Royce 是對的，就像 Vision Statement 所說的，重點應該是表格所指出的順序—大型複雜系統爲 200 頁。

子系統階層

子系統階層的模樣？這裡是我用過的簡單標準形式。請注意，這些是用來實現系統的概念形式。實際系統界限超出這些形式的集合，且每一個形式的外部使用案例的總和就是系統的外部使用案例的總計；因此，實際系統可能有超過 10 個外部使用案例，但上限並非毫無限制，關於這點，我們稍後會提到。請注意，這裡並不建議所有開發作業都要在其說明中使用使用案例的四個層次。較小的系統（<50,000 sloc）可能只使用一兩個。

層次 1

在層次 1，我們有類別在零個或更多個子系統中實現的使用案例：



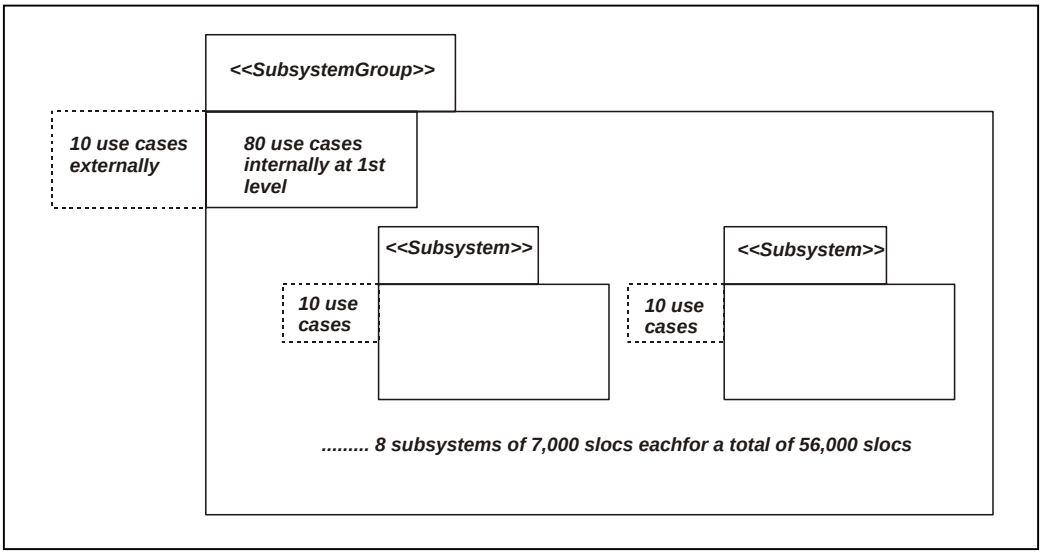
在此層次上評估系統大小的範圍（使用 7 加減 2 的概念）：

- 2 到 9 個類別（不形成子系統）—1700 sloc 到 8000 sloc，或
- 5 個類別的 1 個子系統，總計 4000 sloc，最多
- 7 個類別的 9 個子系統，總計 53,550 sloc。

其使用案例表示爲可由類別實例實現。其範圍爲 2–76 個使用案例。這些是模糊限制，至少上限是—以此方式（以此大小）建置系統的機率（絕不以更高層的形式表達所要的行爲）應在此限制內下降至零。更大的使用案例計數可能表示一些病狀。

層次 2

在下一層，我們有 8 個子系統組成的子系統群組。我認爲這等於防禦專有名詞中的電腦系統配置項目 (CSCI)。在此層次，使用案例是由子系統的合作來實現：



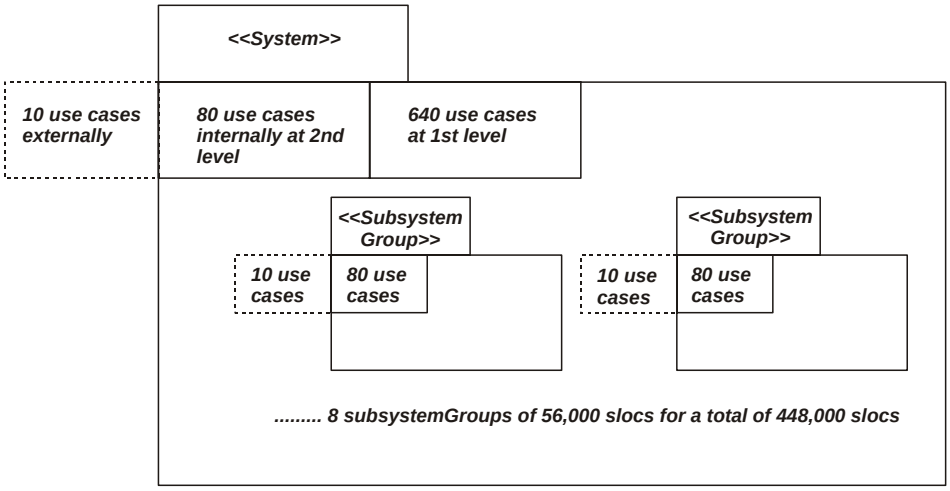
在此層次上評估系統大小的範圍（使用 7 加減 2 的概念）：

- 從 5 個類別的 5 個子系統的 1 個 subsystemGroup，總計 22,000 sloc，到
- 7 個類別的 7 個子系統的 9 個 subsystemGroup，總計 370,000 sloc

其範圍爲 4–66 個外部使用案例。同樣地，這些都是模糊限制。

層次 3

這是下一層，含有屬於子系統群組的系統。在層次 3，使用案例由子系統群組的合作來實現：



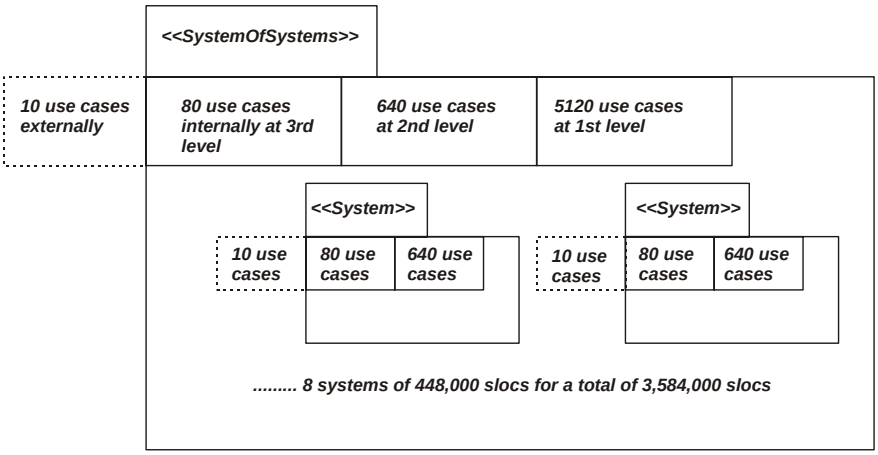
在此層次上評估系統大小的範圍（使用 7 加減 2 的概念）：

- 從 5 個類別的 5 個子系統的 5 個 subsystemGroup 的 1 個系統（總計 110,000 sloc），到
- 7 個 subsystemGroup 的 9 個系統，每一個 subsystemGroup 有 7 個子系統，每一個子系統有 7 個類別，總計 2,600,000 sloc。

其範圍爲 3–58 個外部使用案例。同樣地，這些都是模糊限制。

層次 4

在下一層，我們有系統的系統。在層次 4，使用案例由系統的合作來實現：



在此層次上評估系統大小的範圍（使用 7 加減 2 的概念）：

- 從 5 個類別的 5 個子系統的 5 個 subsystemGroup 的 5 個系統的 1 個系統（總計 540,000 sloc），到
- 7 個系統的系統的 9 個系統，每一個 subsystemGroup 有 7 個子系統，每一個子系統有 7 個類別，總計 18,000,000 sloc。

其範圍爲 2–51 個外部使用案例。同樣地，這些都是模糊限制。我認爲有更大的集合，但我不想理會它們！

每個使用案例的人力

我們在每個層次評估這些名義上的大小的人力，以獲得對每個使用案例人力的一些了解。使用 Estimate Professional 工具[9]（以 COCOMO 2 爲基礎[10] 和 Putnam 的 SLIM[11] 模型），將語言設定爲 C++（其他成本動因設定爲名義上）以及在每一個名義大小點（假設 10 個外部使用案例）計算每一個範例系統類型的人力，產生「表 1」的結果。

大小 (sloc)	簡單商務系統使用案例的工時	科學系統使用案例的工時	複雜指令和控制項系統使用案例的工時
7000 (L1)	55（範圍 40-75）	120（範圍 90-160）	260（範圍 190-350）
56000 (L2)	820（範圍 710-950）	1700（範圍 1500-2000）	3300（範圍 2900-3900）
448000 (L3)	12000	21000	38000
3584000 (L4)	148000	252000	432000

[9] Software Productivity Center Inc, http://www.spc.ca/ 提供 Estimate Professional 工具。

[10] 請參閱 Boehm81 和 http://sunset.usc.edu/COCOMOII/cocomo.html。

[11] 請參閱 Putnam92。

表 1：不同範例類型的每個使用案例人力

表 1 所顯示的層次 1 (L1) 和層次 2 (L2) 的範圍有考慮個別使用案例的複雜性—透過對 COCOMO 程式碼複雜性矩陣的對比加以評估。在 L2，我認爲複雜性的變化將開始歸入系統類型的特性描述中，因此，更高層次的複雜指令和控制系統使用案例，將包含較低層次的複雜性混合。以對數刻度繪製這些即產生「圖 2」。

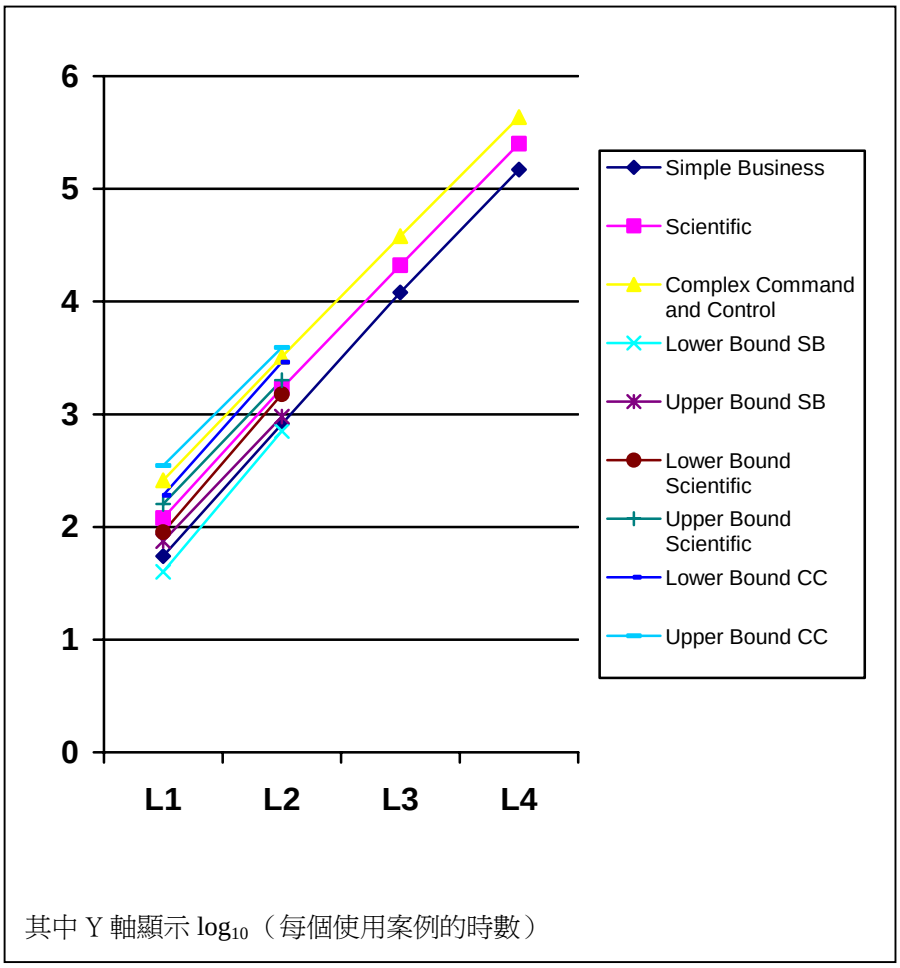


圖 2：按大小的使用案例人力

由此可見，舊的 Objectory 數字 150–350 小時/使用案例 ($10^{2.17}$–$10^{2.54}$) 很適合 L1，也就是說，這些是可以經由類別合作而實現的使用案例—此數字畢竟仍有一些調整空間。然而，在分析期間描述所有專案特性還是不夠—正如某位同僚在電子郵件通訊中所述：那樣太過單調。

人力評估

現在真實的系統大小並不適合這些便利位置，因此，爲了幫助思考該如何描述系統特性，我們可以使用隨之衍生的模糊限制，並如「圖 3」所述地繪製它們。

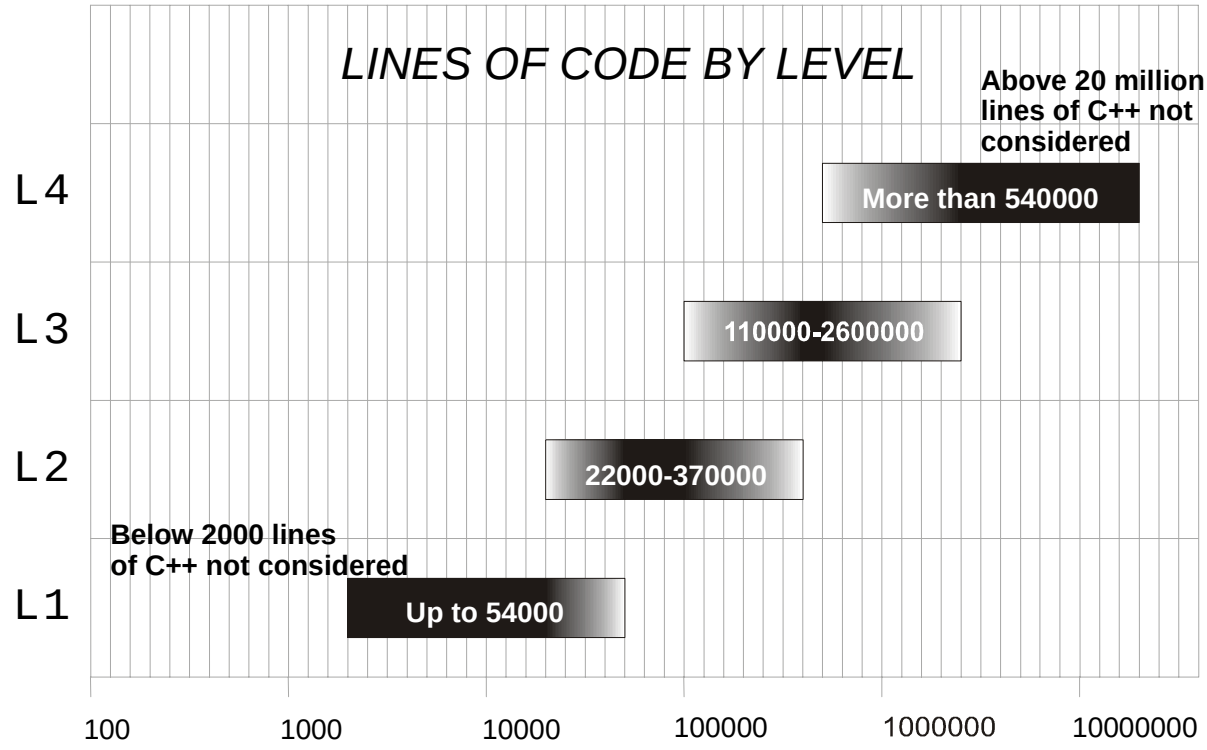


圖 3：每一層的大小頻帶

從「圖 3」我們看到最多 22000 sloc 的系統很可能在層次 1 加以描述，其使用案例計數是介於 2–30 之間。此大小的較高使用案例計數可能表示使用案例的精度太細。

在 22000 和 54000 sloc 之間，有層次 1 和層次 2 使用案例的混合，其使用案例計數介於 4（所有層次 2）和 76（所有層次 1）之間。此圖表試圖顯示這些極端值的機率很低。

在 54000 和 110000 sloc 之間，結構完善的系統可在層次 2 完整描述，其使用案例計數介於 10 到 20 之間；混合結果爲 L1/L2/L3（1–160 個使用案例，其極端值的機率***非常***低）。

在 110000 和 370000 sloc 之間，可能混合了層次 2 和層次 3，其使用案例計數介於 3（所有層次 3）和 66（所有層次 2）之間。

在 370000 和 540000 sloc 之間，如果在層次 3 有完整描述，其使用案例計數將介於 9 到 12 之間；混合結果爲 L2/L3/L4（1–100 個使用案例，其極端值的機率***非常***低）。

在 540000 和 2600000 sloc 之間，可能混合了層次 3 和層次 4，其使用案例計數介於 2（所有層次 4）和 60（所有層次 3）之間。

在 2600000 sloc 以上，層次 4 的使用案例計數應該從 ~8 開始增加。

有多少個使用案例才夠？

這裡有一些有趣的觀察，可支持一些經驗法則。常有的疑問是：多少個使用案例會太多？ 這個問題通常表示在需求擷取期間有多少個使用案例會太多。答案似乎是超過 ~70，即使對於最大型系統，可能指出在設計之前的精度太細。在 5–40 之間剛好，但若沒有考量層次，數字本身是無法用來估計大小和人力的。這是***起始***數字，適合特定層次。如果大型超系統分解成系統，再分解成子系統等等，則可能產生數百個使用案例的計數。如果使用案例一直發展到達到類別層次爲止，則最終計數可能達到數百，甚至數千（假設 140 人年專案是 ~600，或每個使用案例 15 個功能點）。然而，這不會以純使用案例分解的形式發生，且與設計無關。這些使用案例是從 Jacobson97 所描述的流程中發生的—其系統層次的使用案例分割成配置在子系統的行爲，可針對上撰寫低層次使用案例（並以其他子系統作爲參與者）。

人力評估程序

那麼，我們該如何進行評估呢？有一些先決條件：若沒有先瞭解問題領域、***對提出的系統大小及適合評估階段的架構若沒有一點概念，那麼很難做出以使用案例爲基礎的評估。***

可根據專家意見或較正式地由 Wideband Delphi 技術（這是由 Rand 組織在 1948 發明，請參閱 Boehm81 的說明）來完成評估的初步剪接。這可讓評估者將系統放在「圖 3」的其中一個大小頻帶中。此放置暗示使用案例計數的範圍，並指出運算式層次（L1、L1/L2 等等）。評估者必須根據對架構的目前瞭解和網域的詞彙，來決定剛好適合一個層次的使用案例是否不連續地分割，或混合了各個層次（按照事件流程表達的方式）。

經過這些考量，可明顯看出資料可能有問題；比方說，如果 Delphi 估計值爲 600,000 程式碼行（或同等功能點），卻欠缺架構，以致於對系統結構不太瞭解，「圖 3」會建議使用案例計數介於 2（所有層次 4）和 14（所有層次 3）之間。如果使用案例計數實際上是 100，則使用案例可能太早分解，或 Delphi 估計值言之過早。

繼續這個範例：如果實際使用案例計數是 20，且評估者決定這些全都是 L3，而且使用案例長度爲平均 7 頁，系統又是屬於複雜商務類型，則每個使用案例的時數（根據「圖 2」）爲 20,000。這必須乘以 7/9 以說明表面上較低的複雜性（根據使用案例長度）。因此，此方法的總工時爲 20*20000*(7/9) = ~310,000 個人時，或 2050 個人月。根據 Estimate Professional，複雜商務系統的 600,000 行 C++ 程式碼需要 1928 個人月。因此，這個捏造的範例還算一致。

如果實際的使用案例計數是 5，且評估者決定這些使用案例在 L4 分割 1，在層次 3 分割 4，不僅如此，L4 使用案例爲 12 頁，L3 使用案例平均爲 10 頁，因此人力爲 1*250,000*12/9+4*21000*(10/9) = ~2800 人月。這似乎暗示可能需要重看一遍 Delphi 估計值，即使系統的主要組件仍只有在極高層次上才能瞭解，錯誤範圍還是很大。

如果原始 Delphi 估計值爲 100,000 行 C++，「圖 3」表示使用案例應該在 L2，且它們應大約有 18 個。如果實際上有 20 個，就像第一個範例一樣，則應用此方法而不考量實際使用案例層次將產生非常錯誤的結果行（如果 Delphi 估計值錯的很離譜的話）。

因此，評估者必須檢查使用案例是否真的位於建議的摘要層次 (L2)，而且可以經由子系統的合作來實現，且使用案例不是全部都在 L3—雖然 Wideband Delphi 方法並非常常這麼差勁（亦即，當實際值接近 600,000，它卻預測 100,000）。不過，重點是若未建構與使用案例層次一致的一些抽象的或概念的架構，此評估方法將無法大膽進行。對於在此領域經驗豐富的評估者而言，模型是做出層次判斷的心理因素；對於經驗不足的評估者和團隊而言，則最好做一些架構建模，以瞭解使用案例在特定層次實現得如何。

混合表示式使用案例的計數（亦即，混合層次 N 和層次 N+1）應該計算成下界使用案例類型的 $n=8^{(\text{兩個層次之間的極小距離})}$。因此，以 50% L1 和 50% L2 估計的使用案例應該計算成 $8^{0.5} = 3$ L1 使用案例，以獲得整體計數。以 L2 和 L3 之間的 30% 估計的使用案例應該計算成 $8^{0.3}$ L2 使用案例 = 2 L2 使用案例。以 L2 和 L3 之間的 90% 估計的使用案例應該計算成 $8^{0.9} = 7$ L2 使用案例。

表格的大小調整

若要考慮到整體大小，實際上需要進一步調整個別時數/使用數字—在該大小的系統的環境定義中，每個層次上的工時數字是適當的。因此，在「表 1」的 L1 上，在建立 7000 sloc 的系統時，將套用每個使用案例 55 個小時。實際數

字視總系統大小而定，因此，如果要建立的系統假設有 40,000 sloc，而且有 57 個層次 1 使用案例描述它，則簡單商務系統的工時不是 55*57，而是 $(40/7)^{0.11} * 55 = 66$ 小時/使用案例。這是基於大小 (size) 與工時 (effort) 的 COCOMO 2 關係。根據 COCOMO 模型，工時 = A *（大小）$^{1.11}$，其中：

- 大小是 ksloc
- A 將成本動因作爲因素計入
- 專案比例係數是名義上的（指數爲 1.11）

請注意，這些計算可作爲因素計入 Estimate Professional 之類的工具中，以刪除計算負擔；基於完整起見，而在這裡顯示它們。

因此，每個 ksloc 的工時或每個單位等於 A*（大小）$^{1.11}$/大小，這產生 A*（大小）$^{0.11}$，且 S1 大小的工時/單位對 S2 大小的工時/單位的比例是 $(S1/S2)^{0.11}$。

除了 Delphi 估計值之外，系統大小可以從不同層次的使用案例計數粗略計算出來：如果層次 1 有 N1 個使用案例，層次 2 有 N2 個，層次 3 有 N3 個，層次 4 有 N4 個，則總大小爲 [(N1/10)*7 + (N2/10)*56 + (N3/10)*448 + (N4/10)*3584] ksloc。因此，我們可以將總大小除以「表 1」第 1 欄顯示的每一個層次的大小（以 ksloc 爲單位），來計算出「表 1」每一個工時/使用案例數的工時倍數。

- 因此，在層次 1　　$(0.1*N1 + 0.8*N2 + 6.4*N3 + 51.2*N4)^{0.11}$
- 在層次 2　　$(0.0125*N1 + 0.1*N2 + 0.8*N3 + 6.4*N4)^{0.11}$
- 在層次 3　　$(0.00156*N1 + 0.0125*N2 + 0.1*N3 + 0.8*N4)^{0.11}$
- 在層次 4　　$(0.00002*N1 + 0.00156*N2 + 0.0125* N3 + 0.1*N4)^{0.11}$

顯然地，例如在層次 4，層次 1 使用案例的數目與層次 3 或層次 4 的數目相比，作用不大。

摘要

已呈現以使用案例爲基礎的評估的架構。爲了使呈現方式更具體，我們選擇了一些架構參數的值，這些值被認爲誤差不大。和以往一樣，這種推測應該接受現實的測試，並在收集資料時重估參數。架構考量不同系統類別的使用案例層次、大小和複雜性，而不依靠精細的功能分解。若要減輕計算的負擔，可建構工具（例如 Estimate Professional）的前端系統，來提供依據使用案例輸入大小的替代方法。

對於本白皮書若有任何意見和評論，請聯繫 John Smith, jsmith@rational.com。

參照

1. Armour96: Experiences Measuring Object Oriented System Size with Use Cases, F. Armour, B. Catherwood, et al., Proc. ESCOM, Wilmslow, UK, 1996
2. Boehm81: Software Engineering Economics, Barry W. Boehm, Prentice-Hall, 1981
3. Booch98: The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 1998
4. Cockburn97: Structuring Use Cases with Goals, Alistair Cockburn, Journal of Object-Oriented Programming, Sept-Oct 1997 and Nov-Dec 1997
5. Douglass99: Doing Hard Time, Bruce Powel Douglass, Addison Wesley, 1999
6. Fetcke97: Mapping the OO-Jacobson Approach into Function Point Analysis, T. Fetcke, A. Abran, et al., Proc. TOOLS USA 97, Santa Barbara, California, 1997
7. Graham95: Migrating to Object Technology, Ian Graham, Addison-Wesley, 1995
8. Graham98: Requirements Engineering and Rapid Development, Ian Graham, Addison-Wesley, 1998
9. Henderson-Sellers96: Object-Oriented Metrics, Brian Henderson-Sellers, Prentice Hall, 1996
10. Hurlbut97: A Survey of Approaches For Describing and Formalizing Use Cases, Russell R. Hurlbut, Technical Report: XPT-TR-97-03, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf
11. Jacobson97: Software Reuse – Architecture, Process and Organization for Business Success, Ivar Jacobson, Martin Griss, Patrik Jonsson, Addison-Wesley/ACM Press, 1997
12. Jones91: Applied Software Measurement, Capers Jones, McGraw-Hill, 1991
13. Karner93: Use Case Points - Resource Estimation for Objectory Projects, Gustav Karner, Objective Systems SF AB (copyright owned by Rational Software), 1993
14. Lorentz94: Object-Oriented Software Metrics, Mark Lorentz, Jeff Kidd, Prentice Hall, 1994
15. Major98: A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object-Oriented Software Projects, Melissa Major and John D. McGregor, Dept. of Computer Science Technical Report 98-002, Clemson University, 1998
16. Minkiewicz96: Estimating Size for Object-Oriented Software, Arlene F. Minkiewicz, http://www.pricesystems.com/foresight/arlepops.htm, 1996
17. Pehrson96: Software Development for the Boeing 777, Ron J. Pehrson, CrossTalk, January 1996
18. Putnam92: Measures for Excellence, Lawrence H. Putnam, Ware Myers, Yourdon Press, 1992
19. Rechtin91: Systems Architecting, Creating & Building Complex Systems, E. Rechtin, Prentice-Hall, 1991
20. Royce98: Software Project Management, Walker Royce, Addison Wesley, 1998
21. RUP99: Rational Unified Process, Rational Software, 1999
22. Stevens98: Systems Engineering – Coping with Complexity, R. Stevens, P. Brook, et al., Prentice Hall, 1998
23. Thomson94: Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects, N. Thomson, R. Johnson, et al., Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA '94, 1994

Rational
the software development company

兩個總公司：

Rational Software
18880 Homestead Road
Cupertino, CA 95014
電話：(408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
電話：(781) 676-2400

免付費專線：(800) 728-1212
電子郵件：info@rational.com
網址：www.rational.com
國際辦事處：www.rational.com/worldwide