UML 精華第三版

UML Distilled Third Edition

物件模型語言標準簡介

Martin Fowler 原著

趙 光 正 編譯

Cris Kobryn、Grady Booch、

Ivar Jacobson 與 Jim Rumbaugh 序

碁峰資訊股份有限公司 印行

- 想了解什麼是類別圖中最重要的東西嗎？（請參見第 3 章）
- 想一睹新的 UML 2.0 版中，循序圖裡面為流程控制而新增的*互動框*表示法嗎？本書也介紹了許多人愛用的非正式表示法。（請參見第 4 章）
- 想知道 UML 各個版本間有哪些變化嗎？（請參見附錄）
- 想擁有一份 UML 中最常用表示法的快速對照表嗎？（請參見書本內頁）
- 想知道 UML 2.0 版中新加入了哪幾種圖，卻不想費力去K規格書嗎？（請參見第 1 章）

*UML 精華*前兩版讓 30 萬名以上的開發人員受惠。第三版同樣也是 UML 2.0 版與 1.x 版的最佳資訊來源，它引導大家快速、精確地了解 UML 並使用它。

對本書讀者來說，有些人是想要快速跟上 UML 2.0 版的腳步，學習其中的必要內容。其他人則是希望把本書當作手邊方便好用的參考書，快速找到 UML 中最常用的部分。本書作者以簡短、扼要、專注的呈現方式，達到上述兩項要求。

本書介紹 UML 中所有最主要的圖、說明它們的用途，並教導大家在畫或讀這些圖時，一些用得上的基本表示法。這些圖包括類別圖、循序圖、物件圖、套件圖、配置圖、使用案例圖、*狀態機圖*、活動圖、*通訊圖*、*合成結構*、元件圖、互動概圖與*時序圖*。作者並舉了一些簡單易懂的例子，適時說明基本設計邏輯。

如果你跟大多數開發人員一樣，沒時間跟上軟體工程方面的改革創新，那麼 Fowler 的這本新版經典書，正好可以讓你熟悉一些最佳思維。協助大家用最適當的方式，以 UML 來做出有效的物件導向軟體設計結果 － 對專業的軟體設計人士來說，這是一種非常必要的能力。

作者 **Martin Fowler** 是 ThoughWorks 的首席科學家，這家公司從事企業應用程式的開發與移轉工作。作者把物件導向技術應用於企業軟體的開發工作上，已經有十年以上的經驗了。此外，在樣式、UML、重構與敏捷方法論上，他也頗為知名。Martin

跟他的妻子 Cindy，還有一隻不可思議的貓，居住在美國馬薩諸塞州 Melrose 市。個人網頁為http://martinfowler.com。

譯者 趙光正 政治大學資管系碩士斑畢業。他也是 *UML 精華*第二版、*Rational 統一（開發）流程*第二版、*UML 與樣式徹底研究*第二版，以及*使用案例寫作實務*等書的譯者。

目錄

譯序

在翻譯完此書的同時，個人有幸參與邱志良先生所發起的 OOSIG 第一次論壇（http://home.kimo.com.tw/ufjl1572/Chinese/event/event_OOSIG001.htm），並協助舉辦 OOSIG 第二次論壇（http://home.kimo.com.tw/areca_chen.tw/OOSIG/oosig-2.htm）與 OOSIG 第三次論壇（http://140.109.17.201/OOSIG/OOSIG3/OOSIG3.htm），藉由這幾次經驗引發了一些想法，跟大家一起分享。

在軟體開發方面，嫻熟 UML 與架構設計能力的人還是相當少，大家在軟體分析與設計上主要還是藉助不精確的中文與程式碼來溝通。這對國內發展軟體來說是很大的隱憂。如果想要提升大家用 UML 溝通的能力，單單靠個人進修是很難達成的，因為那是很難進行對話的。唯有透過「共同使用」才能讓大家對 UML 這個討論 OO 時所用到的語言，有較好的語言操控能力。至於架構設計方面，因為大家對*設計樣式*（design pattern）的使用還不夠熟練，以致於目前還無法直接以*樣式語言*（pattern language）中的樣式字彙做溝通。由於一個好的架構設計中會牽涉到很多不同的設計樣式，所以如果大家還停留在設計樣式的討論上，將很難以較高的*抽象概念*（abstraction）去釐清架構設計問題。

因此，OOSIG 第三次論壇提出*開放設計*（open design）的想法：利用*開放程式碼*（open source）中的資源，以 UML 淬取出其中的設計方式，而且整個萃取過程模仿開發程式碼精神採用開放設計的方式進行。藉由這樣的過程，磨練大家在 UML 與架構設計方面的能力。

然而，開放設計的目的不僅僅是為了培養大家的設計能力，更希望大家將開放程式碼轉換成開放設計的同時，也能夠吸收裡面所隱含的「產品設計」能力。更進一步將純粹的設計應用在產品上。

重視產品設計的原因何在？跟軟體業的*發展地圖*（development roadmap）有關。

就軟體業的發展來說，國內單一市場並不足以培養出營養均衡的產品，可是絕大部分的軟體公司又沒有足夠的行銷能力與資金，可以將產品推廣到多個市場上面。以硬體業為師，他們也是先在 OEM 市場經營，慢慢轉變成 ODM，之後再由 ODM 轉型成品牌經營，其原因在於：「代工」與品牌經營之間有很明顯的鴻溝在，不容易一舉跨過。既然品牌經營的陳意過高，不容易達成，就現實面來說，軟體業還是只能先靠「代工」賺取足夠的子彈。代工市場可分為「程式設計代工市場」與「產品設計代工市場」。前者向來是印度的天下，近來中國大陸挾充沛、成本低廉的程式設計人員也逐步侵蝕這個市場。由於台灣已進入開發國家之林，人力成本結構在程式設計代工市場方面不具有競爭優勢，所以只能跳過「程式設計代工市場」直接進攻「產品設計代工市場」。任何產業如果不能取得國際市場上一小塊的*市場佔有率*（market share），將很難有所發展。既然要取得「產品設計代工市場」，我們就要想辦法爭取國際市場上的軟體產品設計訂單。不過，國內的軟體產品設計能力，只靠少數的設計人才，將很難取得國際訂單青睞。開放設計一方面可以培養大家在產品設計上的能力，另一方面也有機會讓國際市場了解到台灣具有便宜、熟練的產品設計人才，不啻為一舉兩得的做法。

總而言之，如果沒有品牌經營能力，軟體產品是賣不出去的。台灣因為沒有品牌經營能力，所以想要自己賣軟體產品幾乎是不可能的。退而求其次，我們可以先賺產品設計代工的錢，不過前提是自己要有充沛、成熟的軟體產品設計人才。沒有訂單，設計是不值錢的。所以要爭取國際訂單，就要有便宜、充沛的人才，而這樣的人才，要靠優質的軟體產品設計環境才能培養出來。開放設計就是想透過共享的方式，建立優質的軟體產品設計「環境」。

趙光正

2004/4/16

編輯範例

英文頁碼 ▼ 1

專有名詞 *統一模型語言* (Unified Modeling Language，UML)

重點 **將 UML 視為草稿**

譯註 作者總共提出了三種 UML 的使用差異...

三版序

自古以來，天縱英才的架構師與最有智慧的設計師都了解何謂「儉約之道」。有人把它說得像是一句自相矛盾的話（「少一點就是多一點」），也有人把它說得像是佛家心法（「禪宗的心是初學者的心」），不過裡面所蘊含的智慧都一樣，也不會受到時空所影響，其意義都是：將所有東西減少到只剩下必要的東西為止，讓形體可以很協調地跟功能融合在一起。就建築物來說，不論是金字塔或是澳洲的悉尼歌劇院，都呈現出這種智慧；在電腦方面，不論是范鈕曼電腦架構、UNIX 或 Smalltalk，最棒的架構師與設計師都致力遵循這個眾所皆知、永垂不朽的設計原則。

關於「將論題簡化的思考原則」，我在理解它的價值之後，不論是擔任架構師或是讀書，都一直在尋找這些符合儉約之道的專案或書籍。因為如此，我鼓勵大家最好馬上閱讀本書。

一開始，你可能對我這種講法感到驚訝，畢竟大家很容易把我跟那些*統一模型語言*（Unified Modeling Language、UML）的規格書關聯在一起。因為它們是拿來給工具廠商實作 UML 或給方法論者用的，所以它們被寫得既臭又長。這七年來，我一直擔任一些大型、國際的標準化團隊主席，工作內容包括定出 UML 1.1 版與 UML 2.0 版的規格書，還有兩者間的修定版。經過這段期間，UML 的表達方式與精確度都日趨成熟，不過標準化過程卻讓它無端變得複雜。很遺憾地，標準化過程比較容易得到符合委員會想法的妥協結果，而不易得到具備儉約之美的成果。

個人已經非常熟拈規格書中晦澀難解的細節，對我這樣的 UML 專家來說，可以從 Martin 所粹取出來的 UML 2.0 版精華中學到什麼東西嗎？老實說，我跟你們一樣，都

可以從中學到相當多東西。第一點，Martin 非常熟練地將內容龐大又複雜的模型語言刪到只剩下很實用的一小部分而已，這些內容都是他證實過、有效的實務經驗。第二點，他拒走捷徑，不在原書最後修定版以增頁方式加入新內容。當 UML 這個模型語言不斷變大，Martin 依然保持他的一貫目標，找出「UML 中最有用的一小部分」，而且也只告訴你這麼多東西。他所提到的那一小部分是 UML 中可幫你完成百分之八十工作的百分之二十神秘內容。能捉住 UML 這個令人避之唯恐不及的野獸並馴服它，真是一種無以倫比的成就啊！

讓人印象更深刻的是 Martin 不但達成了上述目標，而且他的寫作風格也非常迷人、口語化。他跟我們分享一些自己的看法與並提到一些軼事，這讓此書讀起來非常有趣，也提醒我們構思系統架構或設計系統這樣的工作需要創造力與生產力兼備。如果我們極力追求儉約心法，那麼應該會發現到用 UML 來畫出專案模型這件事跟初中時大家在寫書法或上畫圖課一樣有趣。UML 既可作為我們觸發創造力的靈感來源，也可以像鐳射光一樣精確規劃出系統藍圖來，然後讓協力廠商競標、依此藍圖建立系統。後者對任何真正的藍圖設計語言來說，都是很嚴峻的考驗。

譯註：「UML 可作為我們觸發創造力的靈感來源」意思是拿 UML 模型作為溝通與理解用，比較接近 Martin 對 UML 的使用態度；「像鐳射光一樣精確規劃出系統藍圖」則是拿 UML 取代文字的程式語言，比較接近 Stephen Mellor 等人所提倡的「可執行的 UML（Executable UML）」。

所以，雖然這是一本很薄的書，不過它的價值卻不容小覷。從本書中，你不但可學到 Martin 建立模型的方式，更可學到他對 UML 2.0 版的詮釋。

我很高興可以跟 Martin 一起工作，協助他從 UML 2.0 版模型語言的特性中挑出本修定版用到的部分，並對本書內容提出修正意見。大家心裡千萬要記得一件事，就是所有現存的語言，不論是自然或人造語言，不是繼續演化下去就是消失。Martin 對新語言特性的選用，以及你跟所有其他實踐者的選用，都是 UML 修正過程中最艱難的部

分。也就是說，你們讓這個模型語言不斷保有活力，並經由市場的自然淘汰，讓它繼續演化下去。

在*以模型驅動的開發方式*（model driven development）還沒有變成主流之前，眼前還是存在著許多挑戰，不過我的意志卻受到這樣的書所鼓舞，因為它很清楚地詮釋出建立 UML 模型時的一些基本概念，並以很實務態度來使用它們。希望你能跟我一樣由此學到許多東西，並運用你的新深刻體驗，改善自己在軟體模型設計方面的實務經驗。

Cris Kobryn
UML 2.0 版規格書提案團隊 U2 Partners 主席
Telelogic 公司首席技術長

一版序

當我們開始精心設計*統一模型語言*（Unified Modeling Language， UML）時，一方面我們希望能夠創造出軟體產業中，最好的、標準的、表達設計理念的方法。同時，也希望它能夠掀開軟體系統模型建立流程的神秘面紗。我們相信一個標準化的模型語言會鼓勵更多的軟體開發者在建立軟體系統之前先建立模型。從 UML 快速且廣泛地被採用來看，開發者確實已經瞭解到，建立模型可以為他們帶來不少的好處。

UML 的誕生就像是一個大型軟體系統模型的建立過程一樣，其過程是經過不斷反覆修改才逐漸產生的。一個標準的誕生包含了許多物件設計者或公司的想法和貢獻。雖然是由我們為 UML 催生的，可是如果沒有其他人的幫助，它也不會成功。真的很感激他們的貢獻。

創造一個標準化的模型語言（modeling language）並且讓它得到大家一致的認同，是一項大挑戰。另一方面，如何用軟體開發的情境，以一個容易被接受的方式，去教育軟體開發者或是把 UML 呈現在他們面前，則又是另一項大挑戰。在這本薄薄的書中，Martin Fowler 克服了重重困難，成功地解說了最新的 UML。

Martin 以一個清晰又友善的風格介紹了 UML 的主要概念。此外，他也很清楚的表達出在軟體開發過程中，UML 所扮演的角色。他以超過 12 年設計軟體和建立模型的經驗，為我們帶來許多在建立模型方面的體認跟知識。

本書已經成功地將 UML 介紹給數千位的軟體開發者了。此外，也讓他們瞭解到，利用這個標準化的模型語言（modeling language）去建構他們的軟體，可以為他們帶來多少好處。

如果您是模型建造者或是開發者，不論您是第一次學習 UML 或是希望很快地了解 UML 在開發過程中所扮演的關鍵角色，我們都誠摯地向您推薦這本書。

Brandy Booch
Ivar Jacobson
James Rumbaugh

自序

在我的生命中，曾遇到許多幸運的事；其中最值得稱道的就是在 1997 年這個對的時間，以對的內容寫出本書第一版。那個時候，以物件導向方式建立模型這個圈子剛從混亂中統一下來，進入由*統一模型語言*（Unified Modeling Language，UML）所主宰的局面。從那時候開始，UML 變成以圖形方式建立軟體模型的一種標準，再也不只是拿它來畫畫物件而已。幸運的是此書竟然變成 UML 方面最暢銷的一本書，狂賣 25 萬本以上。

譯註：modeling 這個字在中文裡面很難找到合適的詞來表達它。有的人把它翻譯成「塑模」，代表「塑造模型」的意思。意思是對了，可是唸起來不是很優雅。像 programming 這個字一樣，大家都知道它代表「程式設計」的意思，可是如果我們把它簡稱為「程設」，就不容易懂；不過，將 programming language 簡稱為「程式語言」，而不用「程式設計語言」，則是一種簡單易懂的說法。因此，個人會視情況選用「模型」或「建立模型」來詮釋 modeling 的意思。例如在 modeling language 中，我就把 modeling 簡稱為「模型」兩字。

沒錯，對我來說這的確是件好事，不過你該因此而購買本書嗎？

我想強調一點：本書內容非常簡潔，所以無法涵蓋 UML 各個層面中的細節。這幾年來，UML 的內容不斷膨脹。我想做的是找出 UML 中最有用的部分，也只告訴你這麼多。雖然厚一點的書可以讓你知道更多細節，不過相對地你也要花更多時間去閱讀它。對你來說，所花費的閱讀時間就是你對一本書所做的最大投資。為了讓本書變薄，我花了很多時間挑出最棒的東西給你，讓你不用花時間來選出這些內容。（遺憾的是，

薄一點的書並不代表它的價格相對便宜；想做出一本有品質的技術書籍，是需要花費相當固定成本的。)

> 譯註：對中文書市場來講，目前還是處於以頁計價的窘境。試想，如果寫程式的人是以程式行數來計價，那麼演算法的價值不就被低估了嗎？

購買此書的理由之一是你想開始學 UML。因為這是一本很薄的書，所以看完之後就可以很快了解 UML 中的必要內容。以此為基礎，你就可以開始看厚一點的書，了解 UML 中的更多細節，例如 UML 的*使用手冊*[Booch、UML user]或*參考手冊*[Rumbaugh、UML Reference]。

本書也可以拿來當作手邊一本方便好用的參考資料，快速查閱 UML 中最常用到的部分。雖然本書無法涵蓋所有內容，不過跟其他大部分的 UML 書籍比起來，它比較輕、也易攜帶。

這也是「有自己想法」的一本書。因為我用物件工作已經有一段很長時間，而且我也對什麼東西有用、什麼沒用有一定想法。

雖然很多人跟我說這是一本很棒的物件入門書，不過當我在撰寫本書時其實並沒有刻意這麼做。如果你想找一本 OO 設計的入門書，個人推薦 Craig Larman 的書[Larman]。

許多對 UML 有興趣的人會用工具來畫 UML 的圖。本書把焦點放在 UML 的標準與常見用法上，而不會詳述各種不同工具所支援的畫 UML 圖功能。雖然 UML 成功地解決了沒有標準表示法之前一片亂轟轟的情況，不過各種工具間還是有一些惱人的差異在，它們分別使用不同的 UML 圖顯示方式或畫法。

本書沒有提到很多跟*以模型驅動的開發架構*(Model Driven Architecture，MDA)有關的事。雖然有許多人把它跟 UML 視為一體，不過許多用 UML 的開發人員卻對 MDA

不感興趣。如果你想多了解 MDA，那麼請先以本書作為了解 UML 概念的起點，然後再看看其他更詳細介紹 MDA 的書。

譯註：對於 MDA，個人推薦 *MDA Explained: The Model Driven Architecture™: Practice and Promise* 一書，三位作者都是 OMG 的 MDA 標準化委員會成員。

雖然本書重點在 UML 本身，不過我還是加入其他對 OO 設計來說很有價值的技術，以作為補充素材（例如 CRC 卡）。對使用物件的人來說，UML 只是成功的一部分原因，個人也認為跟大家介紹一些其他相關技術是很重要的一件事。

對於一本這麼薄的書，想詳細說明 UML 跟原始碼間的關聯是件不可能的事，特別是兩者間並沒有一種標準的對應方式。然而，我還是點出實作 UML 部分片段的一些常見程式寫法。我的程式碼範例是用 Java 與 C#所寫成的，因為我發現這兩種程式語言是最多人看得懂的。不要誤以為我偏好這些程式語言；事實上，我做過許多 Smalltalk 跟 UML 間的對應。

為何對 UML 感到困擾？

我們使用圖形化的設計表示法已經有好一陣子了。對我來說，它們的主要價值在於協助溝通與了解。特別是當你想要省略很多細節時，一張好的圖通常就可協助我們溝通設計上的想法。我們也可以在圖的協助下理解軟體系統或企業流程。當團隊中部份成員想了解某個東西時，圖可以協助他們了解它，並在整個團隊中溝通所理解到的東西。到目前為止，雖然這些圖還無法取代文字化的程式語言，不過它們還是很有用的輔助工具。

許多人相信未來圖形技術將在軟體開發上扮演主導角色。個人對此持較懷疑的態度，不過如果能正確評估出這些表示法能做什麼、不能做什麼，就非常有用。

在這些圖形表示法中，UML 之所以顯得重要，是因為它在 OO 開發族群中被廣泛使用與標準化。現在，UML 不只在 OO 世界成為具主導地位的圖形表示法；在非 OO 圈，也是很風行的一種技術。

本書結構

第一章先簡介一下 UML，說明：它是什麼、對不同人來說它具有的不同意義，以及它的由來。

第二章則提到跟軟體開發流程有關的東西。雖然它跟 UML 是分別獨立的東西，不過個人認為：以了解開發流程的方式來觀察它的使用情境，對像 UML 這樣的東西來說是很重要的。還有一點特別要說明的是，了解反覆式開發方式所扮演的角色是一件很重要的事。對大部分 OO 族群來說，它是開發流程的基本運作方式。

接下來，我將本書剩餘章節依照 UML 中的圖形種類分章說明。第三章與第四章討論 UML 中兩個最有用的部分：*類別圖*（Class Diagram）（核心內容）與*循序圖*（Sequence Diagram）。縱然本書很薄，我相信你還是可以經由我在這些章節中所談論到的技術，由 UML 獲得最大價值。UML 是很大且不斷長大的龐然巨獸，不過你不需要了解 UML 中的所有內容。

> 譯註：順序圖與循序圖是 Sequence Diagram 的兩種常見翻譯。個人雖然覺得順序圖是比較好的翻譯，可是台灣早期使用 UML 的先進們多採用「循序圖」，因此個人也從善如流，在本書中使用「循序圖」一詞。

第五章詳述類別圖中有用但不是那麼必要的部分。第六到八章說明三種有用的圖，它們讓大家從這些圖中看出系統的*結構*究竟為何，包括：*物件圖*（object diagram）、*套件圖*（package diagram）與*配置圖*（deployment diagram）。

譯註：有人將 deployment diagram 翻譯成「部署圖」，個人覺得軍事意味太濃，或許跟台灣早期從事 OO 工作者多從中科院出身有關。

第九到十一章說明三種更有用的*行為*相關技術，包括：*使用案例*（use case）、*狀態圖*（state Diagram）（雖然官方名稱是*狀態機圖*【state machine diagram】，不過大家還是把它叫做狀態圖）與*活動圖*（activity diagram）。第十二章到十七章則簡短說明一些比較不重要的圖。針對這些圖，我只是很快地用範例來解釋它們。

書本內頁中摘要出表示法中最有用的東西。我常常聽見大家說這些內頁是本書中最有價值的部分。你可能發現到閱讀本書某些其他地方時，去參照這些摘要說明是一件很方便的事。

第三版所做的改變

如果你已經擁有本書其他版本，那麼你可能會想第三版跟其他兩版有什麼不同，更重要的是，你是否該買這個新版。

寫這個新版的最主要原因是 UML 2 版的出現。UML 2 版中加入了許多新東西，包括許多新的圖形種類。縱然是跟原先類似的圖，也加入了許多新的表示法，例如循序圖中的*互動框*（interaction frame）。如果你希望知道 UML 2 版發生了什麼變化，卻不想花很多力氣看規格書的話（當然，我不建議你這麼做！），那麼你應該可以從本書看到 UML 2 版的概觀。

我也很高興有機會整個重寫本書中的大部分內容，更新裡面的說明文字與範例。本版還融合了我過去五年來教與用 UML 的經驗。所以這本超薄 UML 書的精神雖然不變，不過大部分內容都是新寫的。

過去這幾年來，我很努力讓本書跟 UML 的最新發展現況保持一致。當 UML 不斷改變時，我也盡己所能讓它跟上腳步。本書是以 2003 年六月相關委員會所通過的 UML 2 版草稿為基礎所寫成的。在這次投票與之後的正式投票間，UML 2 版不太可能會有新的改變，所以個人認為 UML 2 版現在已經夠穩定可以讓本修定版出書了。當然，我也會將最新的更新資訊放在個人網站上（http://martinfowler.com）。

致謝

這些年來，本書的成功實在要歸功於許多人的協助。首先要感謝 Carter Shanklin 與 Kendall Scott。Carter 是 Addison-Wesley 的編輯，是他建議我寫本書的。Kendall Scott 則協助我將本書前兩版整理起來，文字與圖形部分都有。有了他們兩人的協助，我才得以在不可能辦到的短暫時間內完成本書第一版，卻依然符合大家對 Addison-Wesley 的高品質期待。在初期 UML 還不是很穩定的時候，他們也協助我追趕上這些變動。

Jim Odell 是我的良師益友，在我早期的工作中也指導個人許多地方。此外，他也深深地捲入一些技術與個人相關議題，在各持己見的方法論者間調解各種不同看法，讓大家能接受一個共同標準。他對本書的貢獻既深且難以估計，我打賭他對 UML 的貢獻也是如此。

UML 是標準下的產物，不過我個人對那些制定標準的組織過敏。所以為了知道什麼事情正在進行，我需要一個間諜網以隨時追趕上這些委員會的規劃。我的眼線包括 Conrad Bock、Seve Cook、Cris Kobryn、Jim Odell、Guus Ramackers 與 Jim Rumbaugh，如果沒有他們，我的眼光將日益短淺。他們給了我許多有用的提示，也回答了個人許多愚蠢的問題。

Grady Booch、Ivae Jacobson 與 Jim Rumbaugh 是眾所皆知的 OO 界三位巨擘。雖然這些年來個人常開他們玩笑，不過他們還是給了我許多支援，也非常支持本書。不要忘了，我的衝勁通常來自於別人溫暖的贊賞。

評論者是一本書品質好壞的關鍵所在，我從 Carter 那裡學到一點，就是：你不能有太多的評論者。本書前兩版的評論者包括 Simmi Kochhar Bhargava、Grady Booch、Eric Evans、Tom Hadfield、Ivar Jacobson、Ronald E. Jeffries、Joshua Kerievsky、Helen Klein、Jim Odell、Jim Rumbaugh 與 Vivek Salgar。

第三版也有一群很好的評論者，包括：

Conrad Bock	Craig Larman
Andy Carmichael	Steve Mellor
Alistair Cockburn	Jim Odell
Steve Cook	Alan O'Callaghan
Luke Hohmann	Guus Ramackers
Pavel Hruby	Jim Rumbaugh
Jon Kern	Tim Seltzer
Cris Kobryn	

所有這些審查者都花了許多時間來閱讀本書草稿，而且每人至少都發現了一個令人難堪的大錯誤。個人誠致感謝他們。如果本書還存在任何大錯誤的話，那麼個人該負全責。當我發現錯誤時，會把修正內容放在martinfowler.com個人網站書本區的勘誤表上。

負責設計與寫出 UML 規格書的核心團隊為 Don Baisley、Morgan Björkander、Conrad Bock、Steve Cook、Philippe Desfray、Nathan Dykman、Anders Ek、David Frankel、Eran Gery、Øystein Haugen、Sridhar Iyengar、Cris Kobryn、Birger Møller-Pedersen、James Odell、Gunnar Övergaard、Karin Palmkvist、Guus Ramackers、Jim Rumbaugh、

Bran Selic、Thomas Weigert 與 Larry Williams。如果不是他們的努力，我將沒有東西可寫。

Pavel Hruby 開發了一些很棒的 Visio 範本，我用了很多在 UML 的圖上；你可以在 http://phruby.com找到這些範本。

有許多人在網路上跟我接觸，他們對我提出一些個人建議或向我發問，並指出一些錯誤。我無法寫出這些人，在此一併致上最深謝意。

在我最喜歡的 Massachusetts 省 Burlington 市的 SoftPro 技術書籍書店中，我遇到了許多人，也花了許多時間看他們珍藏的 UML 圖，讓我有機會知道大家實際上是如何在使用 UML 的。當然，我也在那裡享受了很棒的咖啡。

Mike Hendrickson 是第三版的組稿編輯。Kim Arney Mulcathy 負責管理整個專案，他也負責版面編排與圖的整理。Addison-Wesley 的 John Fuller 則是產品編輯，而 Evelyn Pyle 與 Rebecca Rider 則協助我們做些編排與校正工作。感謝大家。

當我努力寫書時，感謝 Cindy 陪伴在我的身邊。這段期間，她都會在庭院中種些東西。

父母給了我一個很好的啟蒙教育，讓我的學識得以由此開花結果。

Martin Fowler
Massachusetts 省 Melrose 市
http://martinfowler.com

Chapter 1
簡介

▼ 1

什麼是 UML？

統一模型語言（Unified Modeling Language，UML）代表同一家族的圖形*表示法*（notation），在這些表示法背後有一個共通的*超模型*（meta model）存在。它們可協助我們描述與設計軟體系統，特別是那些用*物件導向*（object-oriented，OO）風格所建造的軟體系統。這個定義有點簡單。事實上，對不同人來說，UML 代表不太一樣的東西。之所以會有這些差異，一半源自於它的歷史，另一半原因是因為大家為了達到有效的軟體工程開發流程，以不同的觀點來用它。因此，本章的主要任務是設定好本書場景，說明大家看待與使用 UML 的不同方式。

在軟體業中，以圖形表示的模型語言已出現好長一陣子了。引發大家使用這些模型語言的背後基本原因是：程式語言無法以夠高的抽象程度，方便我們討論設計相關議題。

雖然*圖形模型語言*（graphical modeling language）已出現好長一陣子了，不過軟體業對它們所扮演的角色還是有很大的爭議在。這些爭議都直接發生在：大家是如何「看待」UML 所扮演角色的。

UML 是相當開放的一種標準，由*物件管理協會*（Object Management Group，OMG）負責管理它，此協會是一個由多家公司所組成的開放性聯合組織。OMG 的成立宗旨

是為了建立支援*互通性*（interoperability）的相關標準，特別是物件導向系統間的互通性。OMG 最廣為人知的事蹟或許是*共通物件請求中介者架構*（Common Object Request Broker Architecture，CORBA）標準。

UML 誕生於 1980 年代末期到 1990 年代初期，許多圖形模型語言的統一。它在 1997 年出現，這時候原本一片亂糟糟的景象總算進入歷史。我跟許多開發人員一樣，都深深感謝這件事情的發生。

UML 的不同用法

深入了解 UML 在軟體開發過程中所扮演的角色，我們發現大家會以不同的方式來使用它，這些差異其實是從其他圖形模型語言延續過來的，也因此導致了一個長久以來難以解決的問題：我們該如何使用 UML 呢？

> 譯註：作者總共提出了三種 UML 的使用差異，第一種差異是分別將 UML 視為草稿、藍圖與程式語言的三種不同用法。第二種差異是以軟體觀點與概念性觀點來看 UML。第三種差異是認為 UML 的本質在於圖或超模型。

為了釐清這個糾結不清的問題，Steve Mellor 跟我兩人分別針對人們使用 UML 的特性提出三種使用模式，分別是把 UML 當成：草稿、藍圖與程式語言來用。就我的「有色」眼光來看，**將 UML 視為草稿**應該是三種用法中最常見的一種用法。採用這種用法時，開發人員會用 UML 來幫他們溝通系統的某些層面。這種用法跟將 UML 視為藍圖的用法一起搭配時，我們可以從*正向工程*（forward-engineering）或*反向工程*（reversee engineering）兩個不同方向來用草稿。如果是採用**正向工程**的話，我們會

在寫程式之前先畫出 UML 圖，而**反向工程**則會由現存程式碼產生 UML 圖，以幫助我們了解程式碼。

譯註：搭配草稿、藍圖兩種用法的步驟為(1)先畫出 UML 草稿、(2)以 CASE 工具用正向工程轉出程式碼大綱、(3)修改程式碼、(4)定期從程式碼以 CASE 工具用反向工程轉出 UML 設計模型。不過，目前的 CASE 工具大部分只能針對系統的靜態結構（例如屬性、方法宣告等）進行正反向工程，對動態的互動情形則支援不一。

將 UML 視為草稿本質上就是在談「選擇性」。以正向使用草稿來看，你會粗略畫出即將要寫程式碼中的某些議題出來，然後用它跟團隊中的一群人討論。使用草稿的目的是為了幫助我們來溝通想法或討論即將要做的一些替代做法。你不會想要跟大家說明所有預計要寫的程式碼，而只會針對一些重要議題跟同事討論一下，或者在開始寫程式之前，先將部份設計方式以視覺方式展現出來。像這樣的會議通常非常短，我們有可能開十分鐘的會以討論未來幾小時的寫程式工作，或者花費一天來討論未來為期 2 個星期的*反覆*（iteration）工作。

以反向使用草稿來看，你可能會用草稿畫出系統中的某個部分是如何運作的。我們不會秀出所有類別，只會秀出有興趣或深入程式碼之前值得一提的部分。

因為將 UML 視為草稿是非正式、變動性也很大的做法，需要大家很快、合力畫出草圖來，所以白板是很常見的一種畫圖工具。我們也常會在紙張上畫出草稿，這時候畫圖的焦點在於溝通而不是完整性。用來畫出草圖的也都是簡易的畫圖工具，而且大家也不會特意遵守 UML 中的每一條嚴謹規則來畫圖。一般書（例如我的其他書）中所畫出來的 UML 圖，大部分都是草稿。它們的焦點都在於選擇性的溝通想法，而不是畫出完整的規格來。

相反地，**將 UML 視為藍圖**則是跟「完整性」有關的做法。如果是採用正向工程的話，那麼我們會先請設計師畫出藍圖，他的工作就是畫出詳細的設計結果，讓程式設計師

依樣畫葫蘆寫程式。設計結果必須夠完整、做出所有的設計決策。程式設計師不太需要思考，就可以用很直覺方式遵循它寫出程式來。當然，設計師有可能跟程式設計師是同一個人，不過一般來說設計師通常是比較資深的開發人員，由他替整個團隊的程式設計師做設計。這種做法其靈感來自於其他類型的工程做法，在這些工程中，我們會先由專業的工程師來畫出工程圖，然後再把這些圖交給營造公司去建造它們。

▼ 3

將 UML 視為藍圖這種做法可以用在「所有」的細部工作上，也可以讓設計師只針對「特定」部份畫出藍圖。常見的做法是請設計師畫出藍圖等級的模型，裡面只包含子系統的介面而已，稍後再由開發人員完成實作時會用到的細節來。

譯註：子系統介面在開發初期不容易定義出來，因為它們大部分都是功能性介面，所以在未設計完整個系統之前，不容易先定義出來。還有就是軟體系統不斷會有設計上的變動（透過重構【refactoring】達成）與需求上的變動（透過變動管理達成）這兩種情形，以致於就算我們在初期把介面定義出來，事實上也不容易穩定下來，這都是此法不易施行的障礙所在。好的設計師了解系統哪些部分容易受到需求影響而不斷修改，而能把系統中不變與易變的兩個部分隔開，因而能控制需求上的變動。不過，設計上的變動就只能靠好的單元測試當作好的防護網，協助我們進行設計上的變動。

如果是採用反向工程的話，藍圖的目的是為了表達出跟程式碼有關的詳細資訊，我們有可能用紙張或在互動式的圖形瀏覽器上秀出藍圖。藍圖裡面可以用圖形方式秀出類別的所有細節，讓開發人員更容易了解。

藍圖比草稿需要更複雜的工具，以便完成這項工作所需要的細節。特製的 CASE（電腦輔助軟體工程）工具就屬於這類用途，不過現在 CASE 這個術語已經被人濫用，而且廠商也避免去用它了。支援正向工程的工具可以讓我們畫圖，把所畫出來的圖放入

儲存庫中以保存資訊。支援反向工程的工具則可以讀取並解讀原始碼，然後產生圖、存入儲存庫中。同時支援正反向工程的工具則稱為***往返式工具***（round-trip tool）。

有些工具直接把原始碼當成儲存庫，把圖當成程式碼的圖形觀點而已。這些工具跟程式設計工作結合地更加緊密，而且通常會直接跟寫程式的編輯器整合在一起。我喜歡把這樣的東西稱為***無往返式工具***（tripless tool）。

藍圖跟草稿間的界線有點模糊，不過個人認為兩者間的差距在於：我們會刻意讓草稿變得不完整，以強調重要資訊；而藍圖則意圖變得無所不包，通常希望寫程式工作可以變成簡單、相當機械化的動作。簡言之，我認為草圖是探索式的，而藍圖則是已決定好的東西。

一旦你花越來越多的功夫在 UML 上，那麼寫程式就會變得更加機械化，很明顯地我們可以把寫程式的工作自動化。事實上，許多 CASE 工具都可以做出某種形式的程式碼產生動作，以自動產生系統的重要部份。自動化程度繼續下去的話，我們最後終將可以在 UML 中詳述系統的所有部份，這時候我們就是**將 UML 視為程式語言**。在這種開發環境下，開發人員所畫出來的圖將可以直接編譯成可執行碼，而 UML 就變成所謂的原始碼了。很明顯地，這種 UML 用法需要很複雜的工具化程度。（此外，從這種用法來看，正反向工程的概念也變得沒有意義，因為這時候 UML 跟原始碼已經是同樣的東西了。）

模型驅動開發架構與可執行 UML

當人們在談論 UML 時，通常也會提到***模型驅動開發架構***（Model Driven Architecture，MDA）[Kleppe et al.]。就本質來說，MDA 是將 UML 視為程式語言的標準用法；這

個標準跟 UML 一樣，都是由 OMG 所掌管。廠商只要做出符合 MDA 的模型製作環境，那麼它所產生的模型也將可用在其他跟 MDA 相容的環境中。

我們在談論 MDA 時通常也會提到 UML，因為 MDA 把 UML 當作基本的模型語言。不過，使用 UML 時當然不一定要用 MDA。

MDA 將開發工作分成兩個主要部分。模型建立者會負責產生***跟平台無關模型***（Platform Independent Model，PIM）。PIM 代表跟任何特定技術無關的 UML 模型。然後工具就可以把 PIM 轉換成***平台特有模型***（Platform Specific Model，PSM）。PSM 代表其目的是想要在某個特定執行環境中執行的系統之模型。其他工具又可以更進一步將 PSM 變成這個平台用的程式碼。PSM 可以是 UML，不過並不一定要是 UML。

所以如果你想用 MDA 建立一套倉儲系統的話，那麼你一開始可以先建立單一個倉儲系統的 PIM。接下來，如果你希望這個倉儲系統可以在 J2EE 與.NET 上執行，那麼你就可以用某些廠商所提供的工具來產生兩個 PSM：每個平台各一個。稍後，再用其他工具進一步產生兩個平台用的程式碼。

如果從 PIM 到 PSM，再從 PSM 變成最終程式碼的過程可以完全自動化，那麼我們就可以將 UML 視為程式語言。如果其中任何步驟要以人工方式完成，那麼我們就是將 UML 視為藍圖。

Steve Mellor 在這類型的工作上一直很活躍，而且最近它被冠上一個術語***可執行 UML***（Executable UML）[Mellor and Balcer]。可執行 UML 跟 MDA 很像，只是使用不一樣的術語而已。同樣地，一開始我們會先產生跟平台無關的模型，它相當於 MDA 的 PIM。然而，下一步是用模型編譯器直接在一個步驟將 UML 模型變成可配置的系統；因此，它不需要產生 PSM。由*編譯器*這個字我們知道這個步驟是完全自動的。

模型編譯器的基礎是那些可再使用的*原型*（archetype）。我們會在**原型**中描述如何將一個可執行 UML 模型轉到特定的程式語言平台上。就倉儲系統這個例子來說，你應

該可以買一個模型編譯器跟兩個原型（J2EE 與.NET 各一）。然後分別用每個原型來跑你的可執行 UML 模型，而得到兩個版本的倉儲系統。

▼ 5

可執行 UML 並沒有用到整個 UML 標準；它認爲 UML 中有許多建構元素都是不需要的，因而沒用到它們。所以，可執行 UML 比整個 UML 還要簡單。

上面這些東西聽起來都不錯，不過這種想法有多真實呢？就我個人觀點來看，至少要考慮到兩個議題。第一個問題是關於工具：是否這些工具成熟到可以完成這個工作。當然，情況會隨時間而變；可確定的是，在我寫本書時，大家還沒有廣泛使用這些工具，而且我也沒有看見有許多工具正在開發中。

另一個更根本的議題跟將 UML 視為程式語言這整個概念有關。就我個人觀點而言，如果 UML 在某個方面較使用其他程式語言更有顯著生產力的話，那麼這麼做是值得的。不過，就我過去曾使用過的各種圖形開發環境來看，我實在很難說服自己這點。縱然它真的更有生產力，我們還是需要有超越臨界量的使用者，才能讓這種做法變成主流。這件事本身就是一個大障礙。跟許多的老 Smalltalk 愛好者一樣，我也認為 Smalltalk 比現在主流的程式語言更有生產力。不過，Smalltalk 現在卻只是一個占有利基市場的程式語言而已，我並沒有看到許多專案在使用它。為了避免落入 Smalltalk 一樣的下場，縱然 UML 是比較好，它還要夠幸運才可成為主流的程式語言。

將 UML 視為程式語言的其中一個有趣問題是：如何畫出行為邏輯的模型來。UML 2 版中提供三種建立行為模型的方法：*互動圖*（interaction diagram）、*狀態圖*（state diagram）與*活動圖*（activity diagram）。每種方法都有人提議將寫程式的做法放進去。如果 UML 真的普及變成一種程式語言，那麼我很想知道究竟是哪一種技術會成功。

看待 UML 的另一種不同觀點在於：大家究竟是以偏概念或建立軟體模型的方式來用它。大部分人都習慣以建立軟體模型的方式來用 UML。採用***軟體觀點***（software

perspective）時，UML 中的各種元素都可直接對應到軟體系統中的元素。如我們可預見的，這種對應關係絕對不會是被強制規定的，不過當我們在用 UML 時，事實上是在討論軟體元素。

如果是採用***概念性觀點***（conceptual perspective）的話，那麼 UML 就是我們對所研究領域的一種描述。這時候，我們不是在討論跟軟體元素有關的東西，而是在建立一套討論某個特定領域的字彙。

對於這些觀點，我們並沒有不可變通的規則存在；如同我們現在所看到的一樣，不同用法間有相當大的空間存在。有些工具會自動把原始碼轉成 UML 的圖，只把 UML 視為是原始碼的另一種觀點而已。這種做法有非常多的軟體觀點存在。另一方面，如果你試著用 UML 的圖跟一大群的會計人員討論，以了解*資產共用池*（asset pool）這個術語的各種意義，那麼骨子裡你是有比較多的概念性觀點存在。

▼ 6

在本書的前一版中，我將軟體觀點細分成*規格觀點*（specification perspective）（*介面觀點*【interface perspective】）與*實作觀點*（implementation perspective）兩種。不過在實務上，我卻發現很難用一條精準的線來區分兩者，所以我覺得不值得這麼大驚小怪地替兩者做出區別。不過，我還是傾向於在圖中強調介面而非實作。

這些不同的 UML 用法會導致大家去爭論 UML 的圖究竟代表什麼意思，還有它們跟世界上其他事物間的關係為何。有一點特別的是，它會影響到 UML 跟原始碼間的關係。有些人所持的觀點是：把 UML 用在產生設計結果，而把程式語言用在實作上，這兩者間是獨立的。其他人則認為「跟程式語言無關的設計結果」是自相矛盾的說法，非常強調這種說法很愚昧。

對 UML 的另一種觀點差異在於：UML 的本質是什麼。就我個人觀點而言，大部分的 UML 使用者，特別是採用草稿用法者，會認為 UML 的本質在於圖。然而，UML 的

創造者卻以第二順位的重要性來看待這些圖；他們認為 UML 的本質在於超模型。圖僅僅是超模型的展現而已。後面的觀點對採用藍圖用法與 UML 程式語言用法的人來說，也是有意義的。

所以無論何時，當你看到跟 UML 有關的東西時，去了解原作者的觀點為何是很重要的一件事。唯有那個時候，你才會真的知道 UML 所引起的那些討厭爭議究竟為何。

討論上述那些差異之後，我需要很清楚地說明自己對它們的立場。幾乎大部分時候，我採用將 UML 視為草稿的用法。個人也發現 UML 草稿在正反向工程，以及概念性與軟體觀點上都是很有用的。

我不是那種會把內容畫得很詳細、對藍圖使用正向工程的人；個人相信我們很難做好它，同時它也會讓開發工作變得很慢。將 UML 視為藍圖時做到畫出子系統介面的程度是很合理的一種做法，不過縱然如此，你還是可以預期到開發人員在實作介面間的互動情形時，很可能會去改變那些介面。至於由反向工程所得到的藍圖，其價值要看工具是如何運作的。如果它被拿來當作程式碼的動態瀏覽器，那麼它就非常有用；如果它只會產生一大堆文件，那麼它只是殺死樹的兇手而已。

個人認為將 UML 視為程式語言是很好的一種想法，不過卻對它是否會被廣泛使用有所存疑。首先，我無法說服自己認為：在大部分寫程式的工作上，圖形比文字更有生產力。縱然真的如此，要讓一種程式語言被廣泛接納也是非常困難的事。

▼ 7

因為上述這些個人偏見，所以本書把比較多的焦點放在草稿用法上。很幸運地，這一點對簡短的入門手冊來說是有幫助的。我無法以跟本書同樣大小的內容來充分發揮 UML 其他兩種用法的價值，不過這種大小的書卻可以有效幫你去看那些採用其他用法的書。所以如果你對 UML 的其他用法有興趣，那麼請將本書當作一本入門書，有需要時再去看其他書。如果你只對草稿用法有興趣，那麼本書將可滿足你的所有需要。

UML 的發展歷程

老實說，我是一位歷史愛好者。當我想看輕鬆的書時，就會拿起一本好的歷史書。不過我知道並不是每個人都對這樣的書有興趣。之所以在這裡談到 UML 的歷史，主要是因為我認為 : 如果不了解它是如何演變的話 , 那麼我們將很難從多方面來了解 UML 的現況。

在 1980 年代，物件剛走出實驗室，邁向真實世界的第一步。某個平台上的 Smalltalk 程式語言已經很穩定、也能用了，而 C++才剛剛誕生而已。此時，大家開始思考物件導向的圖形設計語言。

跟物件導向圖形模型語言有關的重要書籍出現在 1988 年到 1992 年間。當時的領導人物包括 Grady Booch[Booch、OOAD]；Peter Coad[Coad、OOA]、[Coad、OOD]；Ivar Jacobson(Objectory)[Jacobson 、 OOSE] ； Jim Odell[Odell] ； Jim Rumbaugh(OMT)[Rumbaugh、insights]、[Rumbaugh、OMT]；Sally Shlaer 與 Steve Mellor[Shlaer and Mellor 、 data] 、 [Shlaer and Mellor 、 states] ； Rebecca Wirfs-Brock(Responsibility Driven Design)[Wirfs-Brock]。

前面所提到的這些作者都非正式領導一群喜歡他們想法的實踐者。其實，這些方法論之間都非常相似。只是，彼此存在著一些煩人的小差異。同樣的概念可能會有非常不同的表示法，這些差異讓我的客戶感到十分困惑。

在這段令人發暈的期間，標準化的聲音出來了，可是沒有人動手去做這件事。一個 OMG 的工作團隊想試圖進行標準化的工作，可是他們卻收到主要方法論學者的一封公開抗議信。（這讓我想起了一個老掉牙的笑話：方法論者和恐怖主義者間有什麼差異嗎？答案是你只可以跟恐怖主義者談判。）

促成 UML 誕生的第一個重大事件是 Jim Rumbaugh 離開通用電器加入 Grady Booch 所在的 Rational 公司(現在它已屬於 IBM 公司)。大家認為這時候的 Booch/Rumbaugh 聯盟，他們所具有的市場占有率已超過臨界量了。Grady 與 Jim 宣稱方法論的戰爭結束了，他們已經得到勝利。基本上，他們宣稱將以微軟佔領市場的方式來促成標準。一些其他的方法論者因而想成立反 Booch 聯盟。

▼ 8

在 95 年的 OOPSLA 大會上，Grady 和 Jim 第一次提出結合兩人想法的方法論：0.8 版的*統一方法論*(Unified Method)。更重要地，他們宣布 Rational 公司已買下 Objectory 公司，從此以後 Ivar Jacobson 也將加入*統一*團隊。Rational 公司還舉辦了一場慶功宴來慶祝他們的 0.8 版草案問世。(這場宴會的高潮是 Jim Rumbaugh 以歌聲宣布 0.8 版的首次出現；不過大家都希望這是最後一次聽到他的歌聲。)

次年，我們看到比較開放的合併過程出現。OMG，這個過去以來一直擔任旁觀者角色的組織，終於開始扮演主導的角色。Rational 公司一方面必須將 Ivar 的想法融合起來，同時也要花時間跟其他協力廠商溝通。更重要的是，OMG 決定擔任主角。

我們必須了解 OMG 在這個時間點之所以涉入的原因。方法論者，例如一些書的作者，都希望他們受到重視。不過，我認為這些書籍作者的吶喊聲並沒有被 OMG 所聽到。OMG 之所以涉入，其原因是因為他們聽到工具廠商所發出的吶喊聲，大家一致反對這項標準單獨受到 Rational 公司所掌控，因為這麼一來，Ratioanl 工具將取得不公平的競爭優勢。因此，廠商們逼迫 OMG 在維護 CASE 工具互通性的旗幟下做些事情。這面旗子非常重要，因為 OMG 所代表的正是跟互通性有關的事。大家的想法是：創造出 UML 來，讓所有的 CASE 工具都可以自由自在地彼此交換模型。

Mary Loomis 與 Jim Odell 擔任創始工作團隊的共同主席。Odell 很清楚地宣示，為了標準，他準備放棄自己的方法論，不過他不希望產生一個由 Rational 公司強勢推銷得

來的標準。在 1997 年一月，許多組織一起交出方法論標準用的很多提案，藉此達成交換模型的目的。Rational 公司聯合了許多組織，並發行了 UML 1.0 版的文件作為它們的提案，這是第一份叫做統一模型語言的文件。

接下來則是一連串的角力過程，同時很多的提案也彼此合併起來了。OMG 採用最後所得到的 1.1 版作為 OMG 的官方標準。稍後她又做了一些修定版本。1.2 修定版整個都只是在修飾文字而已。1.3 修定版有比較顯著的差異。1.4 修定版新增了許多跟*元件*（component）與*造型輯*（profile）有關的細部概念。1.5 修定版則新增了*動作語意*（action semantic）。

當人們在談論 UML 時，都會讚美 Grady Booch、Ivar Jacobson 與 Jim Rumbaugh 三位創始人。大家喜歡把他們稱為 UML 的三巨頭，不過有些愛開玩笑的人會故意用怪里怪氣的聲音來唸「巨頭」。雖然大家都將 UML 的光環加諸在他們身上，不過個人認為只替他們套上這樣無以倫比的光環，對其他人來說實在有失公允。沒錯，UML 表示法的確是在 Booch/Rumbaugh 的統一方法論中成形的。不過自此以後，大部分的工作都是在 OMG 委員會的領導下所完成的。在後面這些階段，Jim Rumbaugh 是三人當中唯一參與較多的人。個人認為 UML 推動委員會中的這些成員才是 UML 實至名歸的有功人員。

表示法與超模型

目前，UML 中定義了它的表示法與超模型。**表示法**是我們在模型中所看到的圖形部分，它也代表*模型語言*（modeling language）的語法。舉例來說，為了說明*類別圖*（class diagrams）表示法，我們定義了*類別*（class）、*關聯*（association）與*多重性*（multiplicity）等項目與概念來說明它。

當然，這也帶來了一個問題：到底關聯、多重性甚至類別的精確意義是什麼呢？我們可以從一般使用情況得到一個非正式的定義，不過許多人還是希望有一個更嚴謹的定義。

嚴謹的規格與設計語言在正式的方法論中非常普遍。在某些技術中，設計與規格都會以一些述語微積分(predicate calculus)的衍生式來表示。這樣的定義在數學上很嚴謹，而且不會造成模擬兩可的情況。然而，定義的價值是有限度的。我們可以證明一支程式滿足了數學規格，卻不能證明數學規格能夠符合系統的真實需要。

大部份的圖形模型語言都不太嚴謹。他們的表示法看起來很直覺，且沒有正式定義。整體來說，這個情況不會有太大傷害。方法論可能不是很正式，不過很多人卻覺得它們很好用 — 而“ 好用” 就是我們的重點。

然而，方法論者還是試圖找尋方法來改善定義不夠嚴謹的問題，更重要的是還不能破壞好用的特性。其中一個解決方式就是定出**超模型**，並且用圖（通常是類別圖）來定義語言中的一些概念。

圖 1-1 是 UML 超模型中的一小部份，裡面說明*特性*(feature)間的*關係*(relationship)。（舉這個例子是要讓大家對超模型有初步印象，所以這個部份就不多做解釋了。）

對模型表示法的使用者來說，超模型具有多少意義呢？答案大部分要視他們對 UML 的用法而定。採用草稿用法的人通常不會太關心超模型；採用藍圖用法的人應該會多關心一點。至於那些將 UML 視為程式語言的人來說，超模型對他們是相當重要的東西，因為它幫助我們定出語言的抽象語意出來。

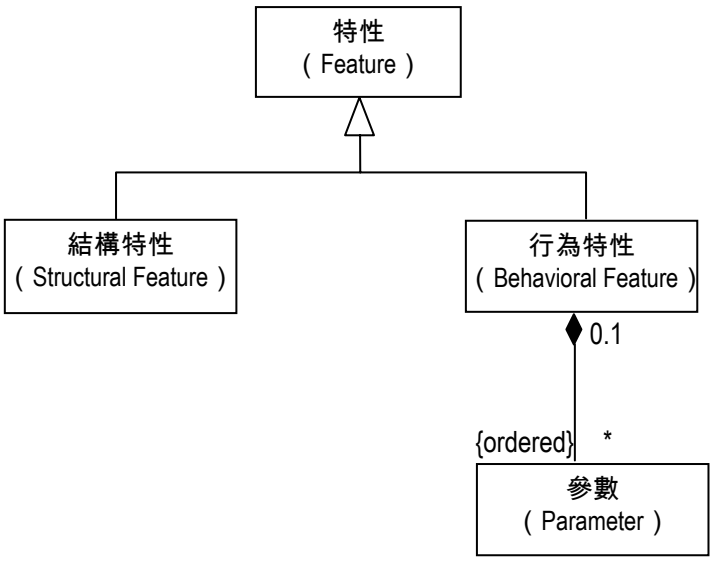


圖 1-1： UML 超模型中的一小部分

▼ 10

大部分持續參與 UML 發展的人，主要都是對超模型感到興趣，因為它對 UML 或程式語言的使用來說是非常重要的。跟表示法相關的議題反而通常被擺在第二順位。如果你想要自行看熟標準相關文件的話，心裡面請謹記這一點。

一旦你深入了解 UML 中更多的細部用法之後，你將知道除了圖形表示法之外，還需要有更多的東西才行。這就是 UML 工具之所以會那麼複雜的原因。

本書並不打算用太嚴謹的方式來說明 UML，我將遵循傳統方法論的方式，讓大家可以用很直覺的方式來瞭解它。如果您希望能夠更嚴謹些，請參閱內容更詳細的書。

▼ 11

UML 中所包含的圖

UML 2 版中描述了 13 種正式的圖形種類，請參閱表 1.1，我們可以依照圖 1.2 中的方式將它們分門別類。雖然這些圖形種類是許多人學習 UML 以及我組織本書的方式，不過那些 UML 的作者們卻不把這些圖視為 UML 的核心部分。因此，這些圖形種類的用法不是那麼硬梆梆的。我們通常可以合法地在某一種圖裡面使用其他種圖所包含的元素。UML 標準會指出某些特定元素典型的用法是將它們放在某些特定的圖形種類中，不過這並不是強制性的規定。

表 1.1 *UML 中的官方圖形種類*

圖形種類	章節	目的	引進此圖的版本
活動圖（activity diagram）	11	程序性或平行性行為	UML 1 版
類別圖（class diagram）	3、5	類別、特性與關係	UML 1 版
通訊圖（communication diagram）	12	物件間的互動情形；焦點在*連結關係*（link）上	UML 1 版中的*合作圖*（collaboration diagram）
元件圖（component diagram）	14	元件的結構與*連接關係*（connection）	UML 1 版
合成結構（composite structure）	13	類別在執行時期的合成情形	UML 2 版新增

配置圖（deployment diagram）	8	將工作成果配置到*節點*（node）上	UML 1 版
互動概圖（interaction overview diagram）	16	混合循序圖與活動圖兩者	UML 2 版新增
物件圖(object diagram）	6	類別實例的組態	UML 1 版中非正式使用
套件圖（package diagram）	7	編譯時期的階層結構	UML 1 版中非正式使用
循序圖（sequence diagram）	4	物件間的互動情形 ; 焦點在訊息的先後順序	UML 1 版
狀態機圖(state machine diagram）	10	說明事件在物件的生命中如何改變狀態	UML 1 版
時序圖(timing diagram）	17	物件間的互動情形 ; 焦點在時序上	UML 2 版新增
使用案例圖（use case diagram）	9	說明使用者如何跟系統互動	UML 1 版

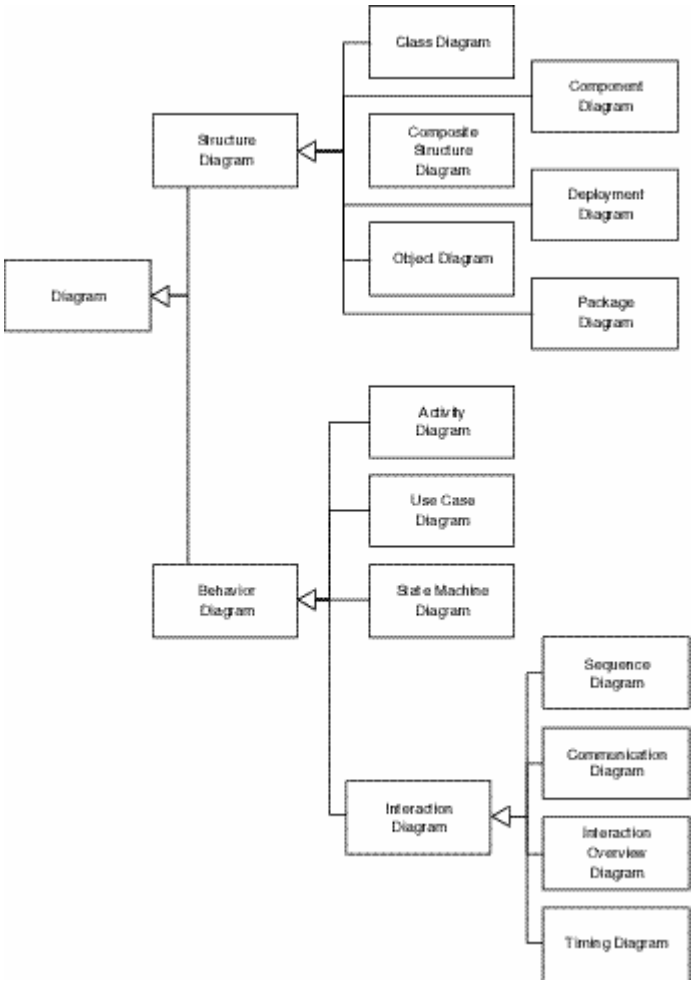


表 1.2 *UML 中圖形種類的分類方式*

▼ 13

什麼才是合法的 UML

剛開始看這個問題時，會以為這是很容易回答的一個問題：所謂合法的 UML 就是在規格書中有定義良好的東西。然而，事實上，它的答案比這個要複雜一些。

在這個問題中，很重要的一個地方是： UML 是否有描述性或強制性的規則存在。具有**強制性規則**的語言會由一份正式規範所掌控，規範裡面陳述什麼東西在語言中是合法的、什麼是非法的，還有就是你在語言中所表達的意義究竟為何。至於擁有**描述性規則**的語言則要靠我們去看實際上大家是如何使用這個語言的，以了解它的規則。程式語言傾向於由一個標準委員會或主導廠商來設定它的強制性規則；另一方面，自然語言，例如英文，則傾向於由慣例來設定那些描述性規則的意義。

因為 UML 是相當精確的一種語言，所以你可能會以為它有一些強制性規則。不過 UML 通常被視為是軟體中跟其他工程學門的藍圖相當的東西，這些藍圖都不會有具備強制性規則的表示法。沒有任何委員會會說結構性工程圖中的哪些符號才是合法的；這些表示法通常跟自然語言一樣，都是約定成俗的。僅僅有一份標準規範並不能解決問題，因為這個專業範疇中的人並不會完全遵守標準規範中所說的每樣東西；你可以問一下法國人跟 Académie Française 有關的事（譯註：個人實在不了解 Académie Française，所以只把這句話直譯出來而已）。除此之外，UML 太複雜了，以致於標準中通常會有許多開放式的解釋。甚至是那些審查此書的 UML 領導者，也對 UML 標準的解釋有不同看法。

對於寫此書的我跟要使用 UML 的你來說，這是很重要的一個議題。如果你希望了解一張 UML 的圖，那麼光知道 UML 標準並不能看到事情全貌，這是你該理解的一件很重要的事。不管是在整個業界，或者是在某個特定專案中，大家真的都會去接受一

些使用上的慣例。因此，雖然 UML 標準是 UML 相關事物的主要資訊來源，不過它卻不是唯一的來源。

我的態度跟大多數人一樣，就是 UML 具有一些描述性規則。UML 標準對 UML 所代表的意義有最大影響力，不過它並非是唯一會影響其涵義的東西。對 UML 2 版來說，我想更是如此，因為裡面引進了一些表示法的使用慣例，這些東西甚至跟 UML 1 版的定義或 UML 的習慣用法相衝突，同時也為 UML 帶來了更多的複雜性。所以，在本書中，當我發現標準跟習慣用法不同時，我都會試圖對兩者做出摘要說明。為了在書中明確指出它們有所不同，我會用**習慣用法**這個字來代表：某種不存在於標準中，不過個人卻認為它被廣泛使用的東西。至於符合標準的東西，我會用**標準**或**規範**這兩個字來標示它。（規範這個字代表標準制定者陳述你必須遵守的一段敘述，遵守它可以讓你的東西符合標準。所以非規範性 UML 就是用來表示某個東西對 UML 標準來說是絕對不合法的。）

▼ 14

當你在看 UML 的圖時，心裡請牢記 UML 的一條通則，那就是：我們可以在一張特定圖中隱藏任何資訊不秀出來。這個隱藏動作可以是一般性的 － 例如隱藏所有屬性，或者是特別指明某些東西 － 例如不要秀出這三個類別。因此，在一張圖中，請不要因為沒有看到某個東西就做出任何推論出來。例如，圖中如果沒有秀出多重性，縱然 UML 的超模型中有預設值（例如屬性的預設值為 1），你也不能因此就推論說它應該具有怎樣的多重性。因此，如果你在圖中沒有看到某個資訊，它不出現的原因有可能是因為它使用了預設值，當然也有可能是因為被刻意隱藏起來。

我曾提過業界有一些常見慣例在，例如把多值的*外部屬性*（property）視為集合。在書中，我會跟大家指出這些預設慣例。

如果你是採用草稿或藍圖用法的人，那麼有件事很重要，就是不要把太多焦點放在得到合法的 UML 上。讓你的系統有好的設計結果是更加重要的事。很明顯地，如果能夠得到既好又合法的東西是件好事，不過我們最好把精力留下來，以便完成好的設計結果，而不要太擔心 UML 中深邃難懂之處。（當然，如果你將 UML 視為程式語言的話，那麼你的東西就必須是合法的，不然你的程式就無法正常執行！）

UML 所代表的含意

跟 UML 有關的議題中最難纏的一個就是：雖然規格書中巨細靡遺地描述了定義良好的 UML 為何，不過裡面卻沒有提到在 UML 超模型的純純世界之外，UML 所代表的意義為何。沒有任何現存的正式定義中有說明我們該如何把 UML 對應到某個特定的程式語言。我們無法看著一張 UML 的圖，就能夠*明確*說出跟它等價的程式碼該長得什麼樣子。然而，你應該可以有大致想法知道程式碼該長得什麼樣子。事實上，這樣已經夠用了。開發團隊對這些東西通常會有他們自己的慣例在，而且你必須熟悉大家在用的這些慣例。

只靠 UML 是不夠的

雖然 UML 中提供了相當多、一整組各式各樣的圖，以幫助我們定出應用程式來，不過我們還是不可能完整列出所有你可能會用到的有用圖形。在許多地方，其他不同的圖都會有幫助，而且如果沒有 UML 的圖適合你的用途，那麼請毫不猶豫地去用不屬於 UML 的圖。

圖 1.3 是一張*畫面流程圖*(screen flow diagram)，裡面秀出使用者介面中會出現的各種畫面，也會秀出這些畫面間該如何切換。我看過、也用過這些畫面流程圖很多年了。不過，我卻只看過非常簡略的定義，說明它們所代表的意義為何；UML 沒有東西跟它一樣，不過我卻發現它是非常有用的一種圖。

表 1.2 中秀出我的另一項摯愛：決策表。決策表是一種可展現出複雜邏輯條件的好方法。你可以用活動圖做同樣的事，不過一旦遇到的不是簡單狀況時，那麼決策表就顯得更加精簡、清楚。此外，現在也已經有許多種形式的決策表問世了。在表 1.2 中，我們把這張表分成上下兩個部分：雙實線上方的是條件，下面的則是結果。每一欄中所秀出的是：我們會由一組特定條件組合得到一組特的結果。

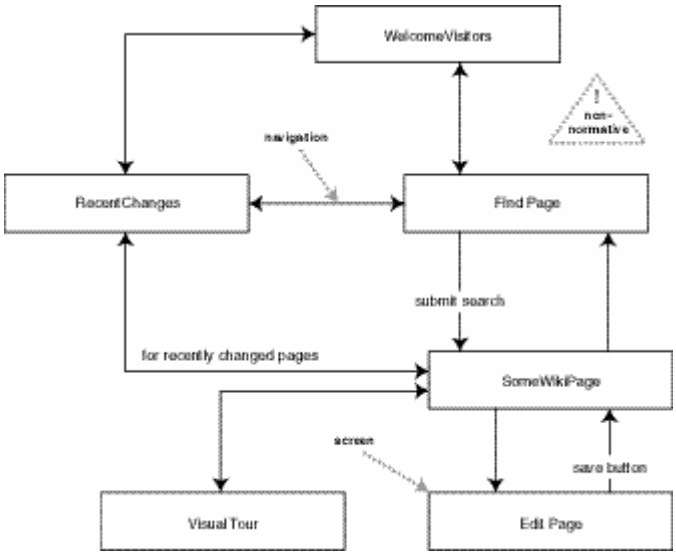


圖 1.3 *描述 wiki（http://c2.com/cgi/wiki）部分系統的非正式畫面流程圖*

表 1.2 *決策表範例*

優待顧客	X	X	Y	Y	N	N
優先訂單	Y	N	Y	N	Y	N
國際訂單	Y	Y	N	N	N	N
費用	$150	$100	$70	$50	$80	$60
通知業務代表	●	●	●			

▼ 16

你可以在各式各樣的書中找到許多種像這樣的東西。如果有看到一些好像適合你專案的技術，那麼請不要猶豫，放心大膽地去試驗這些技術。如果它們運作的不錯，那麼就採用它們吧。如果沒有用的話，就丟掉它們。（當然，對 UML 的圖來說，我們的建議也是如此。）

UML 要從何學起

沒有人，甚至是 UML 的創始人，都無法完全了解或用到 UML 中的所有東西。大多數人只會用到 UML 中的很小部分，而且只用它們就可以行得通了。大家必須找到自己跟同事們適用的那一小部分 UML。

如果你剛開始學 UML，我建議你先把焦點放在類別圖與循序圖的基本型上。它們是最常見的圖形種類，而且就我個人觀點而言，它們也是最有用的。

一旦你抓住它們的竅門，接下來就可以開始去看一些更高等的類別圖表示法，當然也可以看看其他種類的圖形。請去試試看這些圖，並看它們對你來說有多少幫助。如果有些圖看起來對你的工作沒有任何幫助的話，不要怕，請把它們丟掉。

如何獲得更多資訊

本書既不是完整、也不是有決定性的 UML 參考書，當然不用說，它也不是物件導向分析與設計方面的指南。有許多好書裡面寫了太多值得我們學習的想法。當我在討論個別主題時，都會順便把好的參考資料介紹給你，幫助你有更深一層的認識。下面列出一些跟 UML 與物件導向設計相關的一般性書籍。

對於我們所推薦的書，你可能需要檢查一下，看看它們是用哪一版的 UML 所寫成的。在 2003 年六月之前，還沒有已出版的書是用 UML 2.0 版寫成的，這一點也不足為奇，因為標準才剛剛定案而已。我所推薦的都是好書，不過我無法告訴你它們是否或何時會更新成符合 UML 2 版標準。

如果你是物件新手，那麼我推薦你去看自己目前最喜歡的一本入門書：[Larman]。作者在設計方面採用強烈的責任驅動做法，這是值得我們效法的。

如果大家想知道對 UML 來說有最終決定性的講法為何，那麼你應該去看官方標準文件；不過，心裡請記得一件事，就是這些文件是在方法論者的象牙塔中寫成的。如果你想看比較容易消化的標準版本，請看[Rumbaugh，UML Reference]一書。

如果你想多了解物件導向設計方面更詳細的建議，那麼你將可以從[Martin]中學到許多好東西。

我也建議你去讀跟樣式有關的書，這些教材可讓你的功力往上爬。既然方法論的戰爭已經結束了，那麼所有跟分析與設計相關的有趣教材都會在樣式的教材上出現。

Chapter 2
開發流程

▼ 19

就如我曾說過的一樣，UML 是從一大堆物件導向分析與設計（OOAD）的方法論中所誕生出來的。在某種程度範圍內，這些方法論都會在*圖形模型語言*（graphical modeling language）中混合某種開發流程，以說明軟體該如何開發下去。

很有趣的一點是，當 UML 成形時，參與制定工作的人都發現：雖然他們可以認同一種模型語言；不過，他們心裡面有件事情很肯定，那就是他們幾乎無法共同認同某一種開發流程。因此，他們同意把任何在開發流程上希望達成的協議都延後處理，並且把 UML 侷限成一種模型語言而已。

本書的標題是 *UML 精華*，所以我應該可以很安心地把跟開發流程相關的內容排除。然而，如果不去思考模型製作技術該如何用在某種開發流程上，那麼我們將很難相信它還能有什麼用處。換句話說，你用 UML 的方式會受到你所使用的開發流程風格影響。

因此，先討論一下開發流程是一件很重要的事。如此一來，我們將可看到 UML 的使用情境為何。我不準備深入探討任何開發流程細節，只是很簡單希望你有足夠資訊，可以了解它的使用情境，並指出你可在哪裡找到一些更詳盡的資訊。

當你聽到大家在討論 UML 時，通常也會聽到他們談論著 *Rational 統一（開發）流程*（Rational Unified Process，RUP）。RUP 是一種開發流程 － 更嚴謹的說法是，它是一種*開發流程框架*（process framework），你當然可以在 RUP 中用 UML。然而，除了跟它有關係的人同樣也來自 Rational 公司、還有「統一」這個字之外，它跟 UML 間其實並沒有任何特殊關係存在。也就是說，我們可在任何開發流程中使用 UML，RUP 只是其中一種很流行的開發流程而已，稍後會討論到它。

譯註：如果你想快速了解開發流程，那麼最好的方式是先去看 *Process Patterns: Building Large-Scale Systems Using Object Technology*（Scott W. Ambler，1998）一文，它可以讓你快速掌握一般開發流程的大綱。此文中一共提出三種*開發流程樣式*（process pattern），包括(1)*任務流程樣式*（task process pattern），說明要完成某個特定任務時所需採行的詳細步驟為何；(2)*時期流程樣式*（stage process pattern），它由任務流程樣式所組成，我們會反覆執行某個時期中的任務；(3)*階段流程樣式*（phase process pattern），它由時期流程樣式所組成，而且不同的時期流程樣式間通常都會有一些互動情形產生。不同的階段流程樣式會循序發生。

譯註：我們以 RUP 為例，說明該如何把開發流程樣式套用在它身上。RUP 把專案分成四個開發階段，包括*初始階段*（inception）、*詳述階段*（elaboration）、*建構階段*（construction）與*轉換階段*（transition）；每個開發階段裡面會有數次反覆發生；而在每次反覆中，我們會同時有多個*工作科目*（discipline）發生，例如*建立企業模型*（business modeling）、需求、設計等工作科目；每個工作項目裡面又會有詳細的開發活動。如果我們把它跟開發流程樣式對照一下的話，可發現到：四個開發階段相當

於階段流程樣式，工作項目則相當於時期流程樣式，而工作項目中的開發活動又等同於任務流程樣式。唯一有差異的是，RUP 會將每個開發階段細分成數次反覆發生。

反覆式與瀑布式的開發流程

跟開發流程有關的最大爭議點在：*瀑布式*(waterfall) 與*反覆式*(iterative) 開發風格。這兩個術語通常會被大家誤用，特別是「反覆式」開發風格被視為是一種時尚，而「瀑布式」開發風格反被人視為是奇裝異服時更是如此。因此，有許多專案都宣稱他們採用反覆式開發方式，骨子裡卻不折不扣採用瀑布式開發方式。

▼ 20

兩者間的本質差異在於：我們該如何把專案分解成一些比較小的部分。如果你主持一項預計長達一年的專案，那麼幾乎沒有人可以很自在的要求開發團隊離開一年自行工作，等到做好之後再回來。我們需要把專案加以分解，這樣一來大家就可以隨時掌握問題，並追蹤進度。

瀑布式開發風格是根據開發活動來分解專案的。為了撰寫軟體，你需要進行一些特定的開發活動，包括：需求分析、設計、寫程式與測試。因此在這個長達一年的專案裡面，我們就會有長達 2 個月的分析階段，接下來則是 4 個月長的設計階段，再來是 3 個月長的撰寫程式階段，最後才是 3 個月長的測試階段。

反覆式開發風格則是根據不同的功能性子集合將專案分解開來。你可以把一年的專案分解成每次長達 3 個月的*反覆*(iteration)。在第一次反覆中，我們會處理約 1/4 的需求，並且將這 1/4 的需求走完整個軟體生命週期：分析、設計、撰寫程式與測試。在反覆結尾處，我們會有完成 1/4 所需功能性的系統。接下來，我們會進行第二次反覆，它會在第 6 個月結束。這時候，我們會有完成 1/2 所需功能性的系統。

當然，上面對開發流程的說法都有點過分簡化了，不過卻有點出它們在本質上的差異。不過，這樣的描述對這些開發流程來說有點失真。

如果你是採用瀑布式開發方式的話，那麼每個開發階段間都會有某種形式的正式交接動作，不過我們通常也會有機會回到前面的開發階段。寫程式時，有些東西會被提出來，導致我們要重新返回分析與設計開發階段。我們當然不該假設在寫程式工作開始時，所有的設計工作一定都會結束。在稍後開發階段中重新進行分析與設計決策工作是不可避免的情況。然而，這些重返動作都被視為例外情況，大家應該盡可能減少它的發生。

如果你是採用反覆式開發方式的話，那麼在真正的反覆開始之前，我們通常會有某種形式的探索性開發活動。它至少可讓我們對需求取得一些高階觀點：這樣的理解至少足以讓我們把需求分配到接下來會進行的反覆當中。某些高階的設計決策也可能在探索性開發活動中發生。另一方面，雖然我們在每次反覆中都應該要做出產品化、整合後的軟體出來，不過我們通常無法達到這個程度，需要經歷一段穩定期之後，才能去除掉一些最後的程式臭蟲。此外，某些開發活動也會留在最後再做，例如使用者訓練。

你或許無法在每次反覆結尾時把系統變成產品，不過系統還是應該具備產品品質。 然而，我們通常可以每隔一段期間就把系統給產品化；這樣做很好，因為你將可以及早由系統得到評估結果，也可獲得較高品質的回饋。在這種情況下，你通常發現專案中會有多個***發行版本***（release）存在，而這些發行版本則會穿插在一些**反覆**之間。

▼ 21

反覆式開發方式有許多種不同的稱呼：漸進式、螺旋式、演化式與*按摩浴缸式*（jacuzzi spring to mind）等。大家試圖讓它們之間有所差異，不過這些差異既無法得到廣泛認同，也比不上反覆式/瀑布式這個二分法那麼重要。

你可採用混合式的解決方案。[McConnel]一書中描述了一種***分期交付式****生命週期*（staged delivery life cycle），它會先以瀑布式開發風格完成分析與高階的設計工作，然後將寫程式與測試工作分成幾次反覆。以一年的專案為例，它可能會有長達 4 個月的分析與設計工作，然後以 4 次長達 2 個月的反覆來建構出系統。

最近這幾年來的軟體開發流程作家，特別是物件導向社群中的，都不喜歡瀑布式解決方案。眾多不喜歡的理由當中，其中最根本的一個是：在瀑布式開發流程中，我們很難判別專案是否真的沒有脫軌。大家太容易在早期開發階段中就認定專案沒有問題，卻看不到時程延誤情形。我們真正可以判斷出專案沒有脫軌的方式，通常就是產生測試、整合過的軟體。只要我們持續這麼做，那麼當某個東西有偏差時，這種反覆式的開發風格就可以對我們發出較好的警訊。

單單因為上述理由，我就強烈建議專案不該採用純粹的瀑布式解決方案。如果無法採用純粹的反覆式開發技術的話，那麼我們至少也該採用分期交付式的方式來做。

OO 社群長久一來一直偏好反覆式開發方式，而且我們可以放心大膽地說有非常多跟建立 UML 有關的人，至少都偏好某種形式的反覆式開發方式。然而，我所感受到的業界實務經驗中，瀑布式開發方式還是最常見的解決方案。其中一個原因是我所謂的「偽反覆式開發方式」：雖然大家都宣稱正在採用反覆式開發方式，不過實際上卻是按照瀑布式的方法在做。常見的徵狀有：

- 「我們現在正在進行一次分析反覆，接下來會有兩次的設計反覆。...」
- 「這次反覆的程式碼中有許多程式臭蟲，不過我們最後將會把它們清除乾淨。」

譯註：由於反覆式開發方式希望在反覆結束之後，可以產生具備產品品質、測試、整合過的軟體出來。所以如果反覆結束後，程式碼中還有許多程式臭蟲，就不能宣稱反覆結束。另一方面，這點反而像是瀑布式開發流程中撰寫程式開發階段結束，下一步準備進行測試開發階段。

有一件事非常重要，那就是我們要在每次反覆產生測試、整合過的程式碼，而且要盡可能達到產品品質。測試與整合工作是最難以估計的開發活動，所以請不要將這樣的開放式開發活動放在專案要結束時。測試工作應該做到：如果這次反覆原本沒有打算要發行出去，那麼也該做到如果真的要發行的話，也不需要多做一些大量額外的開發工作才行。

採用反覆式開發方式時常會用到的一種開發技術就是***固定時間長度***（time boxing）。它讓每次反覆都有固定長度的時間。如果你發現原本在某次反覆中想要建構的部分無法完全做完的話，那麼你必須決定要在這次反覆中將某些功能性延後處理；而不是將這次反覆的結束日期延後。大部分採用反覆式開發方式的專案都會在整個專案中使用相同的反覆長度；這麼的話，我們就會以有規律性的節奏來產生*建構版本*（build）。

我喜歡這種固定時間長度的做法，因為大家的心態並不容易將系統的功能性延後處理。如果大家能夠經常性的將系統功能延後處理，那麼等到他們面對大的發行版本、需要對延後日期或功能做出明智抉擇時，就會有較好的經驗來處理它。另一方面，在反覆間延後處理某些功能，可有效幫大家學習如何找到真正的需求優先順序。

對反覆式開發方式來說，最常見的考量之一就是重寫程式碼的議題。反覆式開發方式很明白地假設：我們會在專案稍後的反覆中，重寫或刪除掉現存的程式碼。在許多應用領域中（例如製造業），「重做」被視為一種浪費。不過軟體跟製造業不同；因此，重寫現有程式碼要比替原本設計不良的程式碼打補丁要有效率得多了。下面列出技術上的一些實務經驗，它們非常有助於讓重做變得更加有效率。

- ***自動化的回歸測試***（automated regression tests）：當我們正在改變一些東西時，它有助於我們快速偵測到任何程式上的瑕疵。整個 xUnit 家族的*測試框架*（testing framework）對建立自動化的*單元測試*（unit test）特別有價值。從最原始的 JUnit

網站（http://junit.org）開始，現在幾乎已經移植到任何你想像得到的程式語言了（請參見http://www.xprogramming.com/software.htm），對單元測試來說，我們有一個不錯的經驗法則就是：單元測試的程式碼大小應該大約跟你的產品程式碼相當。

- ***重構***（refactoring）：它是以有紀律方式改變現有軟體的一項開發技術[Fowler, refactoring]。重構是將一連串小的、保留行為的轉換動作施行到程式碼庫的工作方式。現在有許多轉換動作已經可以用自動化方式來進行了（請參見http://www.refactoring.com）。

- ***持續整合***（continuous integration）：它能讓整個團隊保持同步，以避免痛苦的整合週期[Fowler and Foemmel]。這種做法的核心是以完全自動化的*建構流程*（build process）為基礎達到的。無論何時，當團隊中有任何成員將程式碼放入程式碼庫時，都會自動啟動此建構流程。我們預期開發人員應該要每天將程式碼放入版本庫控制，所以自動化的建構動作每天會進行好幾次。建構流程中還會執行一大段的自動化回歸測試，以便讓我們捕捉到程式碼中任何不一致的狀況，而能夠很容易就修復它們。

雖然以前大家就都有在用上面所提到的這些技術實務經驗，不過現在因為*終極（程式開發）流程*（Extreme Programming，XP）[Beck]的推廣，而讓它們流行起來。不論我們是採用 XP 或任何其他的*敏捷式開發流程*（agile procss），都可以、也應該去用它們。

預測式或調整式的規劃方式

瀑布式開發方式讓人難過的其中一個原因是：大家想要讓軟體開發變得有可預測性。如果我們沒有辦法很清楚知道寫出某個軟體要花多少錢、要花多久時間才能寫好，這應該就是讓我們感到最沮喪的事了。

預測式的解決方案想在專案初期做一些事，以便可以更加了解稍後會發生的事。依照這種方式，當我們估算專案稍後會發生的事情時，應該可以達到某種合理的精密程度才對。採用***預測式規劃方式***（preditive planning）時，我們可以把專案區分成兩個時期。在第一個時期，我們會先想出一個計畫，這時候計畫是比較難預測的。不過，等到第二個時期，因為計畫已經在實施當中了，所以它就變得比較容易預測。

這不見得是非黑即白、一翻兩瞪眼的事。當專案繼續進行時，它的可預測性會逐漸上升。而且縱然你曾經有過一個預測計畫，事情還是可能不如原先所料。你只能期待穩固可靠的計畫在施行時，實際狀況跟計畫間的差距不會那麼明顯。

然而，有為數不少的人會爭論說：截至目前為止，到底有多少軟體專案是可預測的。這個問題的核心在需求分析。在軟體專案中，增加它的複雜度的其中一個獨特來源就是我們很難了解軟體系統的需求到底為何。絕大多數專案都曾經歷過重大的***需求劇烈變動***（requirement churn）：在專案後期發生需求變動。這些變動會撼動整個預測計畫的基礎。當然，你可以用及早凍結需求、不允許變動的方式來反制這些變動，不過這樣做的風險是你所交出來的系統可能不再符合使用者的需要。

這個問題可能會導致兩種不同反應。第一種方式是把更多的精力放在需求流程上。這樣一來，你可能有辦法獲得更精確的需求，而能減緩需求的劇烈變動程度。

譯註：個人在*醫療資訊系統*（Health Information System，HIS）上的經驗是 － 縱使需求文件寫的再詳細，都無法達到寫程式所需的精確度。請參閱*使用案例寫作實務*（Writing Effective Use Cases）一書中的第 16 章*被漏掉的需求*。還有，什麼時候需求才算寫完呢？相關討論也請參閱同一本書的第 12 章*何時才算寫完使用案例*。

另一種流派則主張需求劇烈變動是不可避免的事。對許多專案來說，我們幾乎不可能讓需求穩定到可使用預測計畫。一方面是因為單憑想像就要知道軟體能做什麼是一件極度困難的事，另一方面則是因為市場環境會造成不可預測的變動。採用這個想法的流派倡導***調整式規劃方式***（adaptive planning），可預測性被他們視為幻影。與其用不切實際的可預測性來欺騙自己，不如面對需求會持續變動這個事實，並且他們所採用的規劃方式會將軟體專案中的變動視為一種常態。他們以控制變動的方式讓專案交出它所能做出來的最佳軟體；我們雖然會去掌控專案，不過，卻不認為它是可預測的。

如果我們談論專案該如何進行，那麼預測式專案跟調整式專案間就會有許多方面的差異浮現出來。如果有人說專案進行的很好是因為它遵照著計畫去走的緣故，那麼他所採用的就是預測式的思維。在調整式的開發環境中，因為計畫始終在變，所以我們沒有辦法說「專案遵照著計畫在走」。這並非意味著調整式專案不做計畫；他們通常會做很多計畫，只不過所做出來的計畫只會被當作是一個基準線，他們用它評估出變動後的結果，而不是拿計畫來預測未來。

▼ 24

採用預測式計畫時，你可以依此做出一份價格固定/範圍固定的合約。這樣的合約會明確說明什麼東西要寫、要花多少錢，還有就是什麼時候要交出軟體。我們不可能用調整式計畫定出這種範圍、價格與時間都固定的內容出來。你可以固定預算與交付時間，可是不能固定要交出的功能性為何。調整式合約假設使用者會跟開發團隊一起合作，經常重新評估要寫出來的功能性究竟為何，而且如果專案進度最後變得太慢的話，就把它結束掉。像這樣，調整式規劃方式的過程可以是價格固定/範圍變動的。

很自然地，我們比較不想採用調整式方案，因為任何人都比較喜歡軟體專案有較好的可預測性。然而，可預測性要看你是不是有一份精確、穩定的需求而定。如果你沒有辦法讓需求變得穩定，那麼預測性計畫就像是建在流砂上的房子一樣，專案脫軌的機會相當高。綜觀上述，我們可以得到兩點建議：

- 除非你有一份精確、穩定的需求，而且有把握它們不會發生顯著變動，要不然請不要做出一份預測式計畫。
- 如果你無法獲得一份精確、穩定的需求，那麼請採用調整式的規劃風格。

預測性與調整性是我們選擇軟體開發生命週期的最初決策。一份調整式計畫絕對需要採取反覆式開發流程來執行它。另一方面，預測式規劃方式採用瀑布式或分期交付式方案時，雖然可以比較容易看到它進行的情況如何，不過它也適可採用反覆式開發流程。

敏捷式開發流程

在過去幾年，大家對*敏捷式軟體開發流程*（agile software process）感到非常有興趣。在*敏捷式*（agile）這個詞底下，其實涵蓋了許多種開發流程，它們都擁有一組共通的價值觀與原則，你可以在敏捷式軟體開發流程的宣言（http://agileManifesto.org）中找到這些價值觀與原則。宣稱遵從敏捷式軟體開發流程的有*終極（程式開發）流程*（Extreme Programming，XP）、Scrum、*系統特性驅動開發方式*（Feature Driven Development, FDD）、*水晶開發流程*（Crystal）、*動態系統開發方法論*（Dynamic Systems Development Method，DSDM）等等。

就我們的討論，敏捷式開發流程本質上具有非常強的調整性在。同時它們也是非常具有人員導向的開發流程。敏捷式解決方案假設專案成功的最重要因素在於專案成員的

素質，以及在人際關係上他們合作的情況究竟如何。至於他們所使用的開發流程或開發工具嚴格來說都只有次一級的效果。

▼ 25

敏捷式方法論傾向於使用時間短暫且長度固定的反覆，通常最多只有一個月長或更短。因為它們不會把太多重心放在文件上，所以敏捷式方案不屑以藍圖方式來使用 UML。它們大部分都會以草稿方式來使用 UML，有些人則倡導將 UML 視為程式語言的做法。

敏捷式開發流程也傾向於有較低的**儀式性**。在專案進行當中，高儀式性或重量級的開發流程會產生很多文件與控制點。敏捷式開發流程則認為儀式性讓專案變得很難變動，其工作方式也不符合有才能成員的本質。因此，敏捷式開發流程通常會展露出**輕量級**的特徵出來。有一點要了解的是，它不具有儀式性是因為調整性與人員導向的結果，儀式性並不是它的基本特質。

Ratioanl 統一（開發）流程

雖然 *Rational 統一（開發）流程*（RUP）跟 UML 無關，可是大家通常會把兩者相提並論。所以我想這裡稍微提一下它應該是值得的。

RUP 雖然被稱為開發流程，不過事實上它是一個開發流程框架，裡面提供跟開發流程有關的一整組字彙與鬆散的結構。當你在使用 RUP 時，你要做的第一件事就是選出自己的***開發案例***（development case）：它是你在專案中將會施行的開發流程。開發案例間的變異很大，所以不要奢望你的開發案例看起來會跟其他開發案例很像。

選出自己的開發案例需要有人事先就非常熟悉 RUP，這個人可以針對特定專案的需要去裁減 RUP。另一方面，我們也可以從越來越多的套裝開發案例中挑一個出來，以此為基礎裁減它。

不論你的開發案例為何，RUP 本質上都是反覆式開發流程。瀑布式開發風格並不符合 RUP 的精神，不過讓人很難過的是，我們經常可以發現到很多專案在執行時，嘴巴說的雖然是用 RUP，骨子裡用的卻是瀑布式風格的開發流程。

所有的 RUP 專案都必須遵循下面四個開發階段。

1. ***初始階段***（inception）：它會對專案做出一個最初的評估結果。一般來說，我們在初始階段中會決定是否要花費足夠的資金來進行詳述階段。

2. ***詳述階段***（elaboration）：它會找出專案的主要使用案例，並且會在幾次反覆中寫出軟體，以便讓系統的架構成形。在詳述階段結束時，你應該對需求有很好的體會，而且會有一個大致上成形、可運行的系統，我們將把它當作後續開發工作的源頭。很特別的一點是，你應該已經找到專案的主要風險所在，也把它們解決掉了才對。

3. ***建構階段***（construction）：它會繼續進行寫軟體的過程，寫出夠發行出去的功能性。

4. ***轉換階段***（transition）：裡面會有各式各樣後期、不需要反覆進行的開發活動，可能會包含將軟體配置到資料中心、使用者訓練等等類似的事項。

在這些開發階段間，有相當多的模糊地帶在，特別是詳述階段與建構階段間更是如此。對某些人來說，他們會在作業模式轉到預測式規劃方式時，從詳述階段轉到建構

階段。不過，對其他人來說，轉換開發階段可能只代表他們對需求有廣泛的了解，而且他們認為可持續到專案其他部分結束的架構，也已經存在了。

有時候，我們會把 RUP 稱做*統一（開發）流程*（Unified Process，UP）。這麼做的原因通常是因為這個組織想要用 RUP 的術語與整體開發風格，不過卻不想用 Ratioanl 軟體公司所授權的產品。你可以把 RUP 視為 Rational 公司以 UP 為基礎所提供的產品，或者乾脆把 RUP 跟 UP 視為相同東西。不管採用哪一種說法，你會發現大家都能夠接受的。

裁減開發流程以適合專案需要

軟體專案跟其他類型的專案有很大不同。軟體開發工作的進行方式會受到許多因素影響，包括：正在建構中的系統、正在使用的技術、開發團隊的大小與地理位置分佈、專案本身所蘊含的風險本質、專案失敗時可能招致的後果、開發團隊的工作風格、組織的文化等等。因此，你應該從來沒有預期過可以找到一個萬用的開發流程來套用在所有專案上。

因此，我們總需要將開發流程加以裁減，以適合專案的特殊環境需要。大家需要做的第一件事就是先看看自己的專案，然後考量一下怎樣的開發流程看起來最合適。結果應該可以得到一小串的開發流程清單供你考慮。

接下來，你應該考慮怎樣的調整動作可以讓這些開發流程符合專案需要。處理這件事時，你必須小心點。除非我們已經開始在用這些開發流程，不然是很難完全體會它們的。在這些情況下，我們最好先以開發流程的現有方式來執行它兩三次反覆，直到你學會它是如何運作時為止，而且這樣做是值得的。接下來，你就可以開始修改這個開發流程。如果一開始你已經蠻熟悉這個開發流程是如何運作的，那麼從一開始就可以

去修改它。請記住，剛開始先少改一點，然後再慢慢增加，這麼做比剛開始改太多，而後又改回來要簡單一點。

樣式

UML 可以告訴我們該如何表達出物件導向設計的結果。相反地，從*樣式*（patterns）所看的是開發流程所得到的結果：它們是可用來作為範例的設計結果。

有許多人會說，專案之所以會有問題，往往是因為專案相關人員缺乏那些有經驗者所熟知的設計方式。樣式中會描述某些事情的一般做法，因為有些人發現某些主題的設計方式經常會重複發生，於是把這樣的設計結果給蒐集起來。他們會針對每個主題來描述它，讓其他人能夠了解樣式，並知道該如何應用它們。

讓我們看看一個例子吧。假設你的桌上型電腦中有一些物件正在某個*處理程序*（process）中執行，它們需要跟其它處理程序中的一些物件溝通。這個處理程序也許是在你的桌上型電腦裡面，也許是在其他地方。你不想讓系統中的物件擔心該如何尋找網路上的其他物件，或執行*遠端程序呼叫*（remote procedure call）。

你能做的就是：在本地端的處理程序中，為遠端物件產生一個*代理者*（proxy）物件。這個代理者物件會有跟遠端物件一樣的*界面*（interface）。你的本地端物件能夠用*處理程序*內部的方式傳*訊息*給代理者物件。無論真實物件存於什麼地方，代理者物件都會負責把訊息傳給*真實物件*（real object）。

代理者樣式是網路或其他地方常使用的一種技術。大家已經有許多經驗知道該如何使用代理者物件、它能帶來什麼優點、它的限制與該如何實作它。不過，跟本

書一樣、與方法論有關的書都不會討論這種知識。我們會討論的是如何畫出代理者樣式，這樣做雖然有幫助，不過跟討論代理者樣式相關經驗的書比起來，後者還是比較有用。

九十年代早期，有些人開始捕捉這些經驗。他們組織了一個對寫出樣式有興趣的社群、一起出資舉辦了一些會議，並且出版了幾本書。

由這個社群所寫出來的樣式相關書中，最有名的是[Ganf of Four]，裡面詳細討論 23 種樣式。如果你希望了解代理者樣式，此書裡面總共花了十幾頁在這個主題上，以詳細說明這些物件該如何一起工作，還有這個樣式的優點與限制、常見的變異情形，以及一些實作上的提示等等。

樣式不僅僅只包含一個模型而已，裡面也包括了採用這種做法的理由究竟為何。我們通常會說樣式代表它對某種設計問題的解決方案。樣式中必須很清楚地點出問題所在、解釋它解決問題的理由，也會說明在什麼環境下，我們可採用或不能採用這個樣式。

▼ 28

樣式之所以很重要，是因為它們是理解一種語言或模型技術後的下一步。樣式可帶給你一系列的解決方案，並告訴你什麼東西可組成一個好的模型，以及你該如何建一個模型。它們用範例來教你這些東西。

當我開始學習設計時，不知道為何凡事都必須從頭開始做起。為何沒有手冊可以指導我該如何做事？樣式社群為大家寫出了這些手冊。

現在已經有許多跟樣式有關的書了，不過它們的品質差異非常大。我最喜歡的書是[Gang of Four]、[POSA1]、[POSA2]、[Core J2EE Patterns]、[Pont]，還有內舉不避親，

跟大家推薦[Fowler, AP]與[Fowler, P of EAA]這兩本書。當然，你也可以參觀一下樣式相關網站：http://www.hillside.net/patterns。

當你開始施行自己的開發流程時，不論你對自己多有自信，隨著專案進行下去，不斷學習開發流程是很有必要的一件事。事實上，反覆式開發方式的最大優點之一就是它支持我們經常進行開發流程的改善工作。

每次反覆結束時，請考慮進行一次***反覆回顧***（iteration retrospective），藉此開發團隊可以聚集在一起，共同思考過去事情是怎麼做的，大家可以怎麼去改善它。如果你的反覆時間為期很短的話，那麼花上兩三個小時就很夠了。有一種進行回顧的不錯方式就是根據下列三類事項，列出一份清單：

1. *要繼續保持下去的*：對於做得不錯的事，你希望以後還是可以確保繼續這麼做下去。

2. *有問題的*：做得不好的部分。

3. *要重新嚐試的*：改變你的開發流程，以求改善之道。

完成第一次反覆之後，你就可以開始每次反覆的回顧工作，這時候你可以審查一下前一次會議中所提出來的這些項目，看看事情是如何變化的。請不要忘記列出要繼續保持下去的事情清單；持續追蹤可行的做法是一件很重要的事。如果你不這麼做的話，那麼我們就可能會喪失對專案的透視感，或許也會沒留意到致勝的實務經驗。

在專案或某個主要的發行版本結束時，你可能考慮做一次比較正式的***專案回顧***（project retrospective），這樣的回顧可能會持續兩三天；相關做法的細節請參見http://www.retrospectives.com/與[Kerth]。讓我最生氣的事情之一就是：組織為何始終無法從他們自己的經驗學到教訓，反而是一次又一次地以所費不貲的錯誤收場。

在開發流程中使用 UML

當大家在使用圖形模型語言時，心裡面通常會以瀑布式開發流程的情境來看待它。瀑布式開發流程在分析、設計與撰寫程式開發階段中，通常會用文件來交接工作。圖形模型通常是這些文件中的主要部分。事實上，從 1970、1980 年代以來，的結構化的方法論中都會談到很多像這樣的分析與設計模型。

不論你是不是採用瀑布式解決方案，我們都依然會做分析、設計、寫程式與測試等開發活動。你可以在具有一個星期長反覆的反覆式專案中，每個星期施行一次微型的瀑布式開發流程。

使用 UML 其實並沒有隱含說一定要製作文件，或者引進一套複雜的 CASE 工具。有許多人只會在會議中用白板畫出 UML 的圖，幫助他們溝通想法。

需求分析

在需求分析開發活動中，我們會試圖了解：針對我們將花費功夫的軟體，它的使用者與顧客究竟想要系統做些什麼事。下面是我們手頭上可用來進行需求分析的一些 UML 現有相關技術：

- *使用案例*（use case），我們會在裡面描述人們是如何跟系統互動的。
- 由概念性觀點所畫出來的*類別圖*（class diagram），它是我們可拿來建構出領域中一組精確字彙的好方法。

- *活動圖*（activity diagram），裡面可秀出組織的工作流程，以了解軟體跟人類活動間是如何互動的。我們可以用活動圖秀出幾個使用案例的背景環境，也可以秀出某個複雜使用案例內部是如何運作的詳細情形。

- *狀態圖*（state diagram），如果某個概念具有一種有趣的生命週期時，它就會變得很有用，裡面會秀出各種狀態，以及改變狀態的事件。

當我們在進行需求分析時，請記住，其中最重要的一件事是跟你的使用者與顧客溝通。一般來說，他們不是軟體開發人員，也對 UML 或其他技術不熟悉。縱然如此，我還是曾經成功地在一些不具有技術相關背景的人身上，使用這些技術。不過，當我們這麼做時，請記得要盡量少用一些表示法。

▼ 30

不論任何時候，只要是有助於溝通，我們心裡面都要有打破那些 UML 規則的準備。在分析時用 UML 的最大風險就是：那些領域專家無法完全理解你所畫的圖。如果了解領域的人無法理解這張圖，那麼它所造成的後果比沒有用更糟；因為它會引起大家對開發團隊的不信任感。

設計

當你在進行設計工作時，你可以在圖中加入更多技術細節。這時候，你可以使用更多的表示法，也可讓表示法更加精確。一些有用的相關技術包括：

- 由軟體觀點所畫出的類別圖，裡面會秀出軟體中的類別，以及它們間是如何相關的。

- 常見情節的*循序圖*（sequence diagram）。有一種很有價值的做法是從使用案例中找出最重要、最有趣的情節，然後用 *CRC 卡*（CRC card）或循序圖畫出軟體中會發生什麼情形。
- 用*套件圖*（package diagram）秀出軟體中大型結構的組織方式。
- 針對有複雜生命歷程的類別，畫出它們的狀態圖。
- 用*配置圖*（deployment diagram）畫出軟體的實體安排情形。

一旦軟體已經寫好，那麼上述這些技術當中，有許多也可以用來記錄文件。如果大家必須要在既有軟體上工作，而且不熟悉這些程式碼，這時候就可以藉此讓大家理解這個軟體。

採用瀑布式生命週期時，你應該在開發階段中畫出這些跟設計相關的開發活動。開發階段結束後的文件中，通常會有這個開發活動合適的 UML 圖。瀑布式開發風格通常隱含我們把 UML 當成藍圖來用。

在反覆式開發風格中，我們可以把 UML 的圖當作藍圖，也可以把它們當作草稿。把它視為藍圖時，我們通常會在負責完成此功能性的反覆之前的那次反覆中，畫出*分析圖*（analysis diagram），而且每次反覆都不是從無到有；相反地，它是修該現有文件中的內容，並在新的反覆中強調有變動的地方。

將 UML 視為藍圖時，設計工作通常會在反覆很早期就做完，而且可能會針對這次反覆要做的部分，一小塊一小塊地完成功能性中的不同地方。這裡重述一次，反覆隱含我們會去修改現有模型，而不會每次建立一個新模型。

以草稿模式來使用 UML 則隱含我們會採用一種更不固定的開發流程。其中一種做法是在反覆一開始時，先花一兩天的時間，粗略畫出這次反覆要用的設計方式。我們也可以在反覆中的任何時間點舉行短暫的設計會議，或者當開發人員開始要處理不小的功能時，舉行為期半小時的快速會議。

採用藍圖模式時，我們預期會在畫出這些圖之後，才進行程式碼的實作工作。改變藍圖中的做法並不是一種常態，所以我們需要由畫藍圖的設計師負責審查被改變的做法。至於草稿則通常被視為設計首稿；在寫程式過程中，如果有人發現草稿不是完全正確，那麼他們將不受到拘束，可逕自改變設計方式。實作者要自行判斷是否這樣的改變需要進行更廣泛的討論，以了解完整的設計脈絡。

我對藍圖做法的其中一個考量就是：自己觀察到要做出正確的藍圖是件非常困難的事，縱然是好的設計師也是如此。我發現自己的設計結果，經歷寫程式的完整過程之後，通常無法不做修改。因此，我始終發現 UML 的草稿雖然有用，卻不認為它們可被視為絕對正確的東西。

不論是採用藍圖或草稿做法，找尋設計上的一些替代方案，這對我們來說都是有意義的事。我們通常最好是在草稿模式下探索這些不同替代方案，因為它可讓我們快速產生這些替代方案，並修改它們。一旦採納其中某種設計方式之後，就可以選擇只畫出草稿，或用藍圖畫出細節出來。

寫文件

一旦你寫好軟體之後，你就可以用 UML 來幫你記錄下來什麼東西已經完成。針對這點，我發現 UML 的圖有助於大家對系統產生一個整體了解。然而，這麼做的時候，我想強調一點，那就是個人並不認為產生整個系統的詳細 UML 圖是有需要的。引述 Ward Cunningham[Cunningham]的一句話：

小心節選出來、寫得很好的一段說明，可取代掉傳統包羅萬象的設計文件。後者中只有幾個單獨地方會引起我們的注意。請強調出這些地方...並忘掉其他東西吧（384 頁）。

我相信詳細文件應該可從程式碼直接產生出來 － 舉例來說，像 JavaDoc 這樣的東西。你所寫出來的額外文件，應該要強調其中的重要概念，並且把這些東西視為讀者深入程式碼細節之前，應該要讀的第一步。我喜歡用 UML 的圖突顯出想要跟大家討論的地方，而且我會以散文形式寫出這些東西、讓它們短得可以在喝一杯咖啡的時間內讀完。我自己喜歡將圖當作草稿來用，裡面強調出系統中最重要的部分。很明顯地，寫文件的人需要決定什麼東西重要、什麼不是，不過寫的人通常會比讀此書的人具備更好知識，足以做出這樣的判斷。

譯註：在 Amazon 網路書店中，有讀者批評 UML 精華第三版在某些表示法的說明不夠詳盡。其實，一本書的內容會反應出作者處理事物的態度。由於作者不希望用本書取代 UML 2 版規格書，所以他認為不是很重要的地方，就只會用簡單的範例帶過。至於他認為比較重要的地方，縱然是一些非正式用法，也是不吝篇幅地做出綜合比較。換句話說，此書是以草稿模式摘錄 UML 2 版中最重要地方的，讀者一方面可藉本書快速瀏覽 UML 2 版中的精華，同時也可接收到作者觀察到：負責制定 UML 2 版的工作團隊「主流」，跟其他實務工作者「非主流」兩派間對 UML 的不同看法。老實說，UML 2 版是在模型驅動開發架構（MDA）下所發展出來的。我們在了解任何事情時，都不該忽略跟它相關的情境究竟為何。MDA 只是主流派對 UML 的看法，因此而制定出來的 UML 2 版，主要也代表他們的聲音。作者想用更中立的態度來看待 UML，所以在本書中引進了非主流派的一些看法，這或許是本書的另一種價值所在。

套件圖中可以秀出很好的系統邏輯地圖來。這種圖可幫我們了解系統的邏輯組成結構，並看出其中的一些*相依性關係*（dependency），以控制它們。配置圖（請參見第 8 章）中則可以秀出高階的實體景象來。在寫文件時期，我們也可以發現到它是很有用的一種圖。

在每個套件中，我希望看到一張類別圖。個人不喜歡秀出每個類別的所有操作。相反地，我只會秀出重要的類別特性（譯註：類別特性包括類別的屬性與方法），以協助我了解裡面有什麼東西。我們可以把這張類別圖當成一份圖形目錄來用。

這張類別圖應該要搭配少量的互動圖，秀出系統中最重要的互動情形來。這裡重述一次，選擇性在這個時候是很重要的一點。請記住，在這種文件中，包羅萬象是具有可理解性的天敵。

如果某個類別擁有複雜的生命週期行為，那麼我會用*狀態機圖*（state machine diagram）來描述它（請參見第 10 章）。當行為很複雜時，我才會這麼做，而且自己也發現到這種情形其實並不會經常發生。

我通常會在文件中放入一些重要的程式碼，這些程式碼是以很易讀的程式風格所寫成的。如果寫到某個特別複雜的演算法，那麼我會考慮多畫一張活動圖（請參見第 11 章）。不過，唯有這張圖比單單只有程式碼更容易讓大家了解時，我才會這麼做。

如果我發現某些概念將來會重複不斷出現，那麼我就會用樣式來捕捉其中的基本概念（請參見前面小節中的說明）。

做文件時最重要的事情之一就是：寫出你不採納的設計替代方案，並說明不採納的原因為何。通常，這是你提供程式碼以外的文件中，最容易被忘掉，卻最有用的部分。

譯註：就譯者的觀點來看，寫文件這個小節中的內容實在非常精采，短短幾句話裡面就針對寫文件這個主題說明了套件圖、配置圖、類別圖、互動圖、狀態機圖與活動圖

這幾種圖之間的相互關聯。老實說，「在開發流程中使用 UML」這個章節裡面所描寫出來的東西，是一種格式鬆散的任務流程樣式。

了解前人所遺留的程式碼

UML 可以用一兩種方式來幫你理解一大堆不熟悉的程式碼。替關鍵事物畫出草圖這種做法可當成一種圖形的筆記機制，它可以在你學習這些程式碼時，幫你記下一些重要資訊。某個套件中關鍵類別的草圖跟類別間的互動情形則可讓你釐清什麼事情會發生。

譯註：作者所描述的這種做法，在譯者的身上確實發生過。由於某項產品的原本開發團隊沒有留下太多設計文件，於是譯者在理解陌生的程式碼時，就試著用 UML 畫出一些圖。事後證明，這些圖在後續討論中，發揮了一些功用。將 UML 當作筆記機制時，畫出「關鍵」的類別圖與互動圖，對程式碼的理解來說的確具有畫龍點睛的效果。

藉由目前市場上的一些工具，你可針對系統中的重要地方，產生詳細的 UML 圖。請不要用這些工具產生大張大張的報表來；相反地，請在你探索程式碼本身時，用它們深入了解程式碼中的一些關鍵地方。這些工具所具備的一種特別棒能力就是可以產生循序圖，秀出多個物件在某個複雜的方法中是如何互動的。

選擇開發流程

個人強烈推薦反覆式開發流程。就如同你在本書之前所看到的一樣：你應該在想成功的專案中使用反覆式開發方式。

或許這麼說有點誇大，不過隨著我的年紀越來越大，個人越來越喜歡採用反覆式開發方式。做得好時，它就是一種必要的技術，你可以用它在早期發現風險，並且在開發過程中獲得比較好的掌控權。這種做法並不等於無管理狀態，不過我還是要很誠實地承認有些人的確是以這種方式來用它。其實，這種做法真的需要很好的規劃，而且它也是很實在的一種做法。基於某種好的理由，每本跟 OO 開發相關的書都會鼓勵大家去用它。

聽到我是敏捷式軟體開發方式的宣言中的一員，你應該不會很驚訝才對，我是敏捷式做法的一位愛好者。自己本身也對終極(程式開發)流程有相當多的正面經驗。當然，你也應該審慎評估它所提到的一些實務經驗。

如何獲得更多資訊

開發流程方面的書已經很多了，而且敏捷式軟體開發方式崛起後也造成許多新書出現。整體而言，我在開發流程方面最喜歡的書是[McConnell]。本書裡面廣泛、實務地提到軟體開發過程中會發生的一些議題，並且列出一長串有用的軟體開發實務經驗來。

在敏捷式軟體開發方式的社群中，[Cockburn, agile]與[Highsmith]兩本書都對它提供了一個很好的概觀。至於以敏捷方式來用 UML 這方面，[Ambler]裡面則提供了很多好的建言。

敏捷式方法論中最流行的一派是終極（程式開發）流程（XP），你可以在 http://xprogramming.com與http://www.extremeprogramming.org中找到許多東西。XP 孕育出許多書，這也是我之前把它稱為輕量級方法論，現在卻稱爲 XP 的原因之一。學習 XP 的起點通常是[Beck]一書。

雖然[Beck and Fowler]一書是針對 XP 而寫的，不過裡面所提到的大部分細節是在談論反覆式專案的規劃方式。此書中的大部分內容在其他跟 XP 相關的書中也會提到，不過如果你只對規劃方面有興趣的話，那麼此書將是一個很好的選擇。

如果你想多了解 Rational 統一（開發）流程的話，個人最喜歡的是[Kruchten]一書。

Chapter 3
類別圖：基本概念

▼ 35

如果有人從暗暗的小巷走向你，跟你說「嗨！能幫我看一張 UML 的圖嗎？」那麼這張圖十之八九就是*類別圖*(class diagrams)。自己受人之託看過的 UML 圖當中，絕大多數都是類別圖。

類別圖不只受到大家的廣泛使用，同時也是我們替概念建模型時，應用範圍最廣的一種圖。每個人都用得到類別圖中的一些基本概念，不過大家卻都很少用得到它的一些高等概念。因此，我把類別圖的討論內容分成兩個部份；基本概念（本章內容）與高等概念（請參閱第 5 章）。

類別圖可用來描述系統中各種物件的*型態*(types), 也可描繪出物件間各式各樣的靜態關係。此外，類別圖中也可以秀出類別的*（類別）性質*(property)與*操作*(operation)，以及可應用到物件間連接方式的一些*限制*(constraint)。在 UML 中，我們用***（類別）特性***（ feature ）來代表類別的（類別）性質與操作這兩種概念。

譯註：property 與 attribute（屬性）這兩個字，有人都把它翻譯成「屬性」。就 UML 的超模型來說，property 等於屬性或關聯。為了讓兩者的有所區別，我們將 property 翻譯成「（類別）」性質。此外，我們也在 feature 的翻譯前面加上「（類別）」以表示它是 feature of class。

圖 3.1 中秀出一張簡單的類別模型，裡面所陳述的東西對處理訂單的人來說並不陌生。圖中的四方形代表類別，方形中又可再細分成三個部分：類別名稱（用粗體標示）、屬性與操作。圖 3.1 中同時也秀出兩種類別間的關係：*關聯*（associations）與*一般化關係*（generalization）。

（類別）性質

（類別）性質代表類別的*結構特性*（structural feature）。一開始，你可以把（類別）性質看成相當於類別中*欄位*（field）的東西。它的真實含意相當複雜，我們稍後才會提到。不過，一開始這麼想是很合理的。

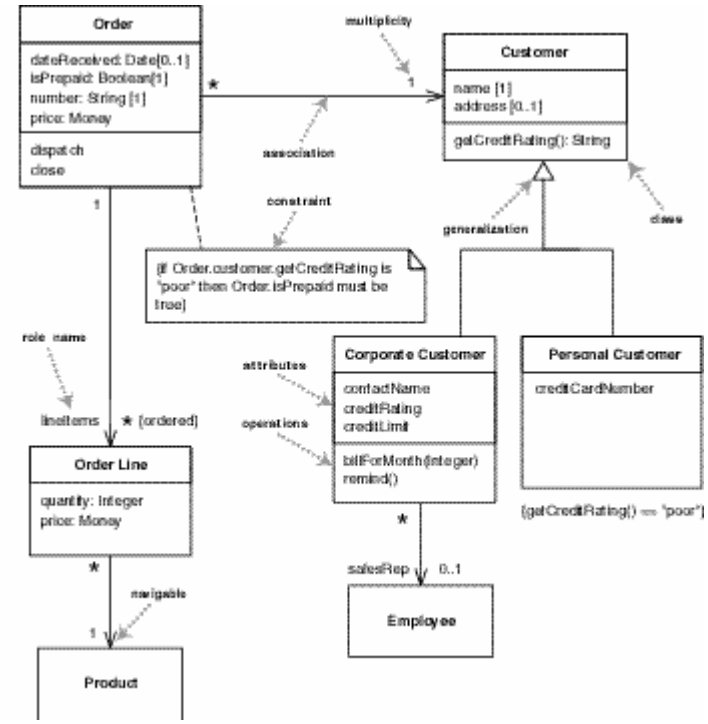


圖 3.1 *一張簡單的類別圖*

譯註：在 UML 2 版中，類別所擁有的（類別）特性可分為結構特性與行為特性（behavioral feature）。結構特性就是上面所說的(類別)性質，而行為特性就是操作。

（類別）性質雖然只是一個概念，不過它卻可以用兩種非常不同的表示法來呈現：屬性與關聯。從圖上看起來，它們雖然非常不同；不過，事實上它們卻代表相同的東西。

屬性

屬性（attributes）表示法用類別四方形中的一行文字來表示（類別）性質。完整語法如下：

可見性 名稱: 型態 多重性 = 預設值 {(類別)性質-字串}
visibility name: type multiplicity = default {property-string}

舉例如下：

```
- name: String [1] = "Untitled" {readOnly}
```

在整個語法當中，只有「名稱」部分是必要的。

- 我們用*可見性*（visibility）標記代表這個屬性是*公開的*（public）（+）或*私有的*（private）（-）；第 5 章中會討論到其他的可見性。
- 屬性的*名稱*大約等同於程式語言中欄位的名稱 － 它說明類別如何參考到這個屬性。
- 屬性的*型態*（type）代表一種限制條件，說明屬性中究竟可以放入哪一種物件。你可以把它視為程式語言中欄位的型態。
- 關於*多重性*（multiplicity），我們稍後再解釋它。
- *預設值*（default value）說明在新產生的物件中，如果產生過程中沒有指定值給這個屬性，那麼我們就會把這個值指定給屬性。
- {（類別）性質-字串}允許你指明這個屬性的一些額外（類別）性質。在上面的例子中，我用{readOnly}代表這個物件的客戶端不能去修改這個（類別）性質。如果沒有這麼寫的話，我們通常會假設這個屬性是可修改的。在本書後面，我會說明一些其他的（類別）性質字串。

譯註：如果屬性是可修改的話，我們會加上{unrestricted}（類別）性質。

關聯

（類別）性質的另一種表示法是關聯。屬性中可秀出來的資訊，差不多也都可以在關聯中秀出來。圖 3.2 與 3.3 分別用兩種不同的表示法來秀出同一個（類別）性質。

關聯用兩個類別中的實線來表示它，箭頭會由來源類別往目標類別畫。（類別）性質的名稱會出現在關聯的目標端，同時還會加上它的多重性。關聯的目標端所連結的類別就是這個（類別）性質的型態。

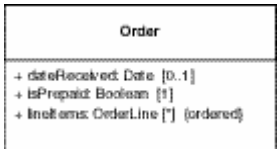


圖 3.2 *以屬性秀出訂單的（類別）性質*

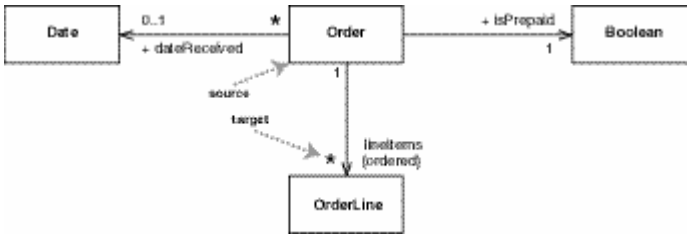


圖 3.3 *以關聯秀出訂單的（類別）性質*

雖然大部分資訊都可以同時用這兩種表示法來表現，不過它們之間還是有些地方不同。其中很特別的一點是：關聯可以在線條兩端同時秀出多重性。

一個東西卻有兩種不同的表示法，所衍生出來的一個明顯問題是：你選擇使用這一種或另一種表示法的原因為何？一般來說，我傾向於把屬性用在小的東西上，例如日期或布林值等，而且它們通常都是*值型態*（value type）的東西。另一方面，我會把關聯

用在有顯著意義的類別上，例如客戶或訂單等。同時，我也傾向比較喜歡用類別四方形畫出圖中有顯著意義的類別，這也會導致我去用關聯；另一方面，我會把圖中比較不重要的東西表示成屬性。做出這種選擇，目的主要是為了要強調出重點，不具有任何更深一層含意。

多重性

（類別）性質的**多重性**代表它可能會有多少個物件存在。你最常可能看到的多重性包括：

- **1**（例如一份訂單只能是某位客戶的。）
- **0..1**（例如企業客戶可能會有一位專門服務他的銷售代表，也可能沒有專屬的。）
- *（例如客戶沒有必要一定要下訂單，而客戶下的訂單也不會有任何上限 － 它可以會有零份或多份訂單。）

比較一般性的說法是，我們可以用一個上限與一個下限來定義多重性，例如我們可以定義凱那斯特紙牌遊戲的玩家有 2..4 的多重性。多重性的下限可以是任何正數或零；上限則可以是任何正數或*（代表沒有上限）。如果上限跟下限一樣，那麼你就可以用一個數字來同時代表上下限；因此，1 等於 1..1。另外，因為 0..*是很常見的多重性，所以我們會把它簡寫成*。

在屬性中，有許多不同的術語都是在說明多重性。

- ***可選擇的***（optional）隱含下限為 0。

- ***強制的***（mandatory）隱含下限為 1 或更多。
- ***單值的***（single-valued）隱含上限為 1。
- ***多值的***（multivalued）隱含上限比 1 更大，通常是*。

如果（類別）性質是多值的，那麼我比較喜歡用複數來命名它。

多重性為多值的元素，我們通常都會預設它會存放在某個*集合*（set）當中，所以如果你跟客戶要他的訂單，那麼訂單出現的順序將是不固定的。如果這些相關訂單的先後順序有意義，那麼你就需要將{ordered}這個（類別）性質字串加到*關聯端*（association end）。如果你希望這些元素可以重複出現，那麼請在關聯端加上{nonunique}（類別）性質字串。（關聯的預設（類別）性質字串為{unordered}與{unique}，一般它們是不會秀出來的，不過如果你想要的話也可以秀出來。）你可能也會看到一些*多重物件導向*（collection-oriented）的名稱，例如{bag}就代表它是無順序性、不具唯一性的。

譯註：有人可能會覺得 UML 中跟集合有關的（類別）性質字串怎麼那麼多，事實上，這都是為了配合程式語言在這方面的需要。以 C++的標準範本庫（Standard Template Library，STL）與 Java 的多重物件框架（Collections Framework）來說，就有許多不同的容器_C_（container class）存在。下面摘要幾個常見的_C_。

STL	Collections Framework	說明
vector	ArrayList 、Vector	同時具有陣列與串列能力，在陣列用途上速度比較快。
list	LinkedList	同時具有陣列與串列能力，在串列用途上速度比較快。
stack	Stack	具有後進先出（LIFO）的特性
queue		具有先進先出（FIFO）的特性
map	TreeMap	儲存一系列、有先後順序的<key, value>對。
set	TreeSet	有先後順序、只存在唯一值的集合，它是 map 的退化型。

UML 1 版允許我們有不連續的多重性，例如 2, 4（它代表 2 或 4，例如小卡車出現前的年代，車子內的座位數）。不連續的多重性其實並不常見，UML 2 版也把它刪掉了。

屬性的預設多重性是[1]。雖然 UML 的超模型中有這條規則在，不過如果圖中某個屬性沒有標示出多重性的話，我們不能因此就假設它的多重性為[1]，因為圖中可以把多重性資訊隱藏起來。因此，如果這個多重性很重要的話，那麼我就比較喜歡明確標示出它的多重性為[1]。

寫程式時對（類別）性質的詮釋

跟 UML 中的其他東西一樣，用程式碼來詮釋（類別）性質的方式不只一種。其中最常見的一種軟體呈現方式就是程式語言的欄位或(類別)性質。所以圖 3.1 中的 Order Line（明細項目）類別應該等同於下面的 Java 程式碼：

```
public class OrderLine…
      private int quantity;
      private Money price;
      private Order order;
      private Product product;
```


至於像 C#這種本來就有（類別）性質的程式語言，它應該等同於：

```
public class OrderLine…
      public int Quantity;
      public Money Price;
      public Order Order;
      public Product Product;
```

請注意，對有支援(類別)性質的程式語言來說，屬性一般就等同於它的公開(類別)性質；對不支援(類別)性質的程式語言來說，屬性等同於它的私有欄位。對於後者，我們可藉*存取子方法*(accessor method)（*讀取用方法*【getting method】與*設定用方法*

【setting method】）將欄位揭露出來。就欄位來說，唯讀屬性不會有設定用方法；而對（類別）性質來說，它將沒有*設定用動作*（setting action）。請注意，如果你沒有指定某個（類別）性質的名稱，我們通常會直接拿目標類別當作它的名稱。

以私有欄位來解讀圖中的（類別）性質是非常以實作為中心的。另一種比較以介面為導向的解讀方式可能會把焦點放在讀取用方法上，而不是底層的資料。就上面的例子來說，Order Line（明細項目）的屬性等同於下列的方法：

```
public class OrderLine…
      private int quantity;
      private Product product;
      public int getQuantity() {
          return quantity;
      }
      public void setQuantity(int quantity) {
          this.quantity = quantity;
      }
      public Money getPrice() {
          return product.getPrice().multiply(quantity);
      }
```

在這個例子中，價格並沒有出現在資料欄位；相反地，它是經由計算所得到的值。不過就使用 Order Line（明細項目）的其他物件所關心的角度來看，讀取用方法所得到的結果跟欄位一樣。使用它的物件並沒有辦法判別出欄位跟算出來的值有什麼差異。像這樣將隱藏資訊起來就是封裝的本質所在。

譯註：透過寫程式時對 property 的詮釋，相信大家一定更了解譯者把 property 翻譯成（類別）性質的用意。

如果某個屬性可能會有多個值，這隱含跟它相關的資料是一個多重物件。所以 Order（訂單）類別應該會指向 Order Line（明細項目）的多重物件。因為它的多重性其先後順序是有意義的，所以這個多重物件也應該要能保持先後順序（例如 Java 中的 List 類別或.NET 中的 IList）。如果這個多重物件其先後順序沒有意義，那麼我們應該嚴格賦予它不具備有意義的順序性，這時候我們可以用*集合*（set）來實作它，不過大部分人

也會用串列來實作不具順序性的屬性。有些人會用*陣列*（array）來實作它，不過因為UML 隱含這樣的屬性沒有上限，所以我總是會用多重物件的資料結構來實作它。

多值的（類別）性質跟單值的會有不相同的介面，以 Java 為例：

▼ 41

```
class Order {
      private Set lineItems = new HashSet();
      public Set getLineItems() {
          return Collections.unmodifiableSet(lineItems);
      }
      public void addLineItem(OrderItem arg) {
          lineItems.add(arg);
      }
      public void removeLineItem(OrderItem arg) {
          lineItems.remove(arg);
      }
```

在大部分狀況下，你不會直接把值指定給多值的（類別）性質；相反地，你會用一些 add 或 remove 方法來更新它。為了控制 Line Items（明細項目）（類別）性質，訂單必須控制多重物件的成員關係；因此，它不該赤裸裸地把多重物件直接傳出去。遇到這種情況，我會用有保護效果的*代理者*（proxy）物件替多重物件提供一個唯讀外衣。你也可以提供不可更新內容的*反覆子*（iterator）或產生複件。使用此多重物件的用戶端可以修改這些成員物件，不過它卻不該直接去更改多重物件本身。

因為多值的屬性已隱含它是多重物件，所以你幾乎不會在類別圖中看到我們畫出多重物件類別。換句話說，只有在非常低階的實作圖中，我們才會看到多重物件的蹤跡。

你應該非常害怕類別中除了多重物件的欄位與相關存取子之外，沒有其他東西。物件導向設計希望你設計出來的物件有辦法行使豐富的行為，所以這些物件不該只是包含其他物件的資料而已。如果你只是不斷地重複呼叫存取子，以取得資料，這意味著某些行為應該要轉到擁有資料的物件本身。

譯註：大家可以參考 UML 與樣式徹底研究一書第 16 章 GRASP：根據責任設計物件中所提到的 GRASP 樣式來分配責任。

經由上面這些例子說明，大家應該可以體會到 UML 跟程式碼間並沒有硬梆梆、快速的對應方式，然而兩者間有很多相似點卻是不爭事實。不過，在專案團隊中，我們可以藉由團隊慣例讓兩者間有較接近的對應方式。

不論（類別）性質在實作時是用欄位或計算得來的值，它都代表這個物件會持續提供的某個東西，所以你不該用（類別）性質來代表一些短暫關係，例如如果某個物件是方法呼叫時所帶的參數，而且也只限在這個方法的互動中使用它，那麼我們在畫圖時，就不該把這個物件畫成（類別）性質。

雙向關聯

我們前面看過的關聯屬於*單向關聯*（unidirectional association）。另一種常見的關聯是*雙向關聯*（bidirectional association），如圖 3.4 所示。

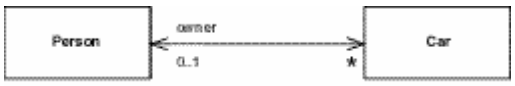


圖 3.4 *雙向關聯*

雙向關聯代表一對（類別）性質，它們從不同方向彼此連結在一起。在圖 3.4 中，Car 類別有（類別）性質 owner:Person[1]，而 Person 類別則有（類別）性質 cars:Car[*]。（請注意，我用【類別】性質所屬型態名稱的複數形替 cars 命名，這是一種常見的做法，不過它不是硬性規定的。）

兩者間反向的連結關係隱含：如果你沿著這兩個（類別）性質找過去，最後應該會回到包含起點的集合。舉例來說，如果我從某個特定的 MG Midget 開始，找到它的擁有者，再回頭看此擁有者所持有的車子集合，就可找到被我當作起點的 Midget 正存在於此集合當中。

除了用（類別）性質替關聯加上標籤之外，有許多人，特別是具有資料模型背景的人，特別喜歡用動詞片語（請參見圖 3.5）替關聯加上標籤，這時候我們可以用完整的句子寫出關係。這種做法是合法的，而且你還可以在關聯上面加上一個箭頭，這麼一來，大家在用動詞片語造句時，就可避免搞不清楚主詞、受詞的困擾。大部分建立物件模型的人都比較喜歡用（類別）性質名稱來替關聯加上標籤，因為這種做法跟責任與操作比較相配。

有些人會以某種方式替每個關聯命名。我則只會在命名有助於增加了解時，才替關聯命名。個人曾看過許多關聯的命名方式都是以「擁有」或「相關於」這樣的動詞片語來命名。

在圖 3.4 中，我們可從關聯兩端的***可瀏覽性箭頭***（navigability arrow）明顯看出關聯的雙向特質。圖 3.5 中則沒有箭頭存在；UML 允許我們用兩種方式中的其中一種來代表雙向關聯。我自己比較喜歡圖 3.4 的雙箭頭做法。當你想清楚表現出雙向關聯時，就可以這麼做。

圖 3.5 *用動詞片語替關聯命名*

在程式語言中實作雙向關聯時，經常會發生一些詭異情況，你必須確認兩邊的（類別）性質有保持同步。如果用 C#的話，我會用像下面這樣的程式碼來實作雙向關聯：

```
class Car…
      public Person Owner {
           get {return _owner;}
           set {
              if (_owner != null) _owner.friendCars().Remove(this);
              _owner = value;
              if (_owner != null) _owner,friendCars().Add(this);
           }
      }
      private Person _owner;
…
      class Person…
           public IList Cars {
              get {return ArrayList.ReadOnly(_cars);}
           }
           public void AddCar(Car arg) {
              arg.Woner = this;
           }
           private IList _cars = new ArrayList();
           internal IList friendCars() {
              //它只會被 Car.Owner 用到
              return _cars;
           }
…
```

其中最重要的地方就是讓關聯的其中一邊控制整個關係 — 如果可能的話，請選擇單值的那一邊。為了讓這種做法可行，從屬端（Person）必須把它原本封裝起來的資料洩漏給主控端知道。結果將導致從屬端產生一個很礙眼的方法（譯註：friendCars），不過如果這種程式語言提供細部的存取控制寫法，那麼這個方法有可能不需要真的存在。在上面的例子中，我用「friend」作為這種方法的命名慣例，表示如果我們是用C++來寫的話，那麼主控端的*設定子*（setter）勢必要是從屬端的*友善類別*（friend）。跟大多數的（類別）性質程式碼一樣，這是非常一成不變的一段程式碼，因此有許多人比較喜歡用程式產生器來產生這些程式。

在概念性模型中，*可瀏覽性*（navigability）並不是一個很重要的議題，所以我不會在任何概念性模型中秀出可瀏覽性箭頭。

操作

操作是類別知道該如何達成的一些動作。很明顯地，操作就是類別中的*方法*（method）。一般來說，我們不會秀出一些僅是處理（類別）性質的操作，因為它們都是可以推論出來的。

▼ 44

在 UML 中，操作的完整語法為：

可見性 名稱（參數串列）：傳回型態 {（類別）性質-字串}
visibility name (parameter-list) : return-type {property-string}

- 可見性可以是公開的（+）或私有的（-）；第 5 章中會討論到其他的可見性。
- 名稱是一個字串。
- 參數串列是操作會用到的一串參數。
- 如果有*傳回型態*（return types）的話，它代表傳回值的型態。
- （類別）性質-字串代表可應用到此操作的一些（類別）性質值（譯註：例如你可用{query}來表示這個操作只會從類別中取出值，不會改變系統狀態）。

參數串列中的參數是用跟屬性類似的方式來表示的。其語法如下：

方向性 名稱：型態 = 預設值
direction name: type = default value

- 名稱、型態與預設值跟屬性中的都一樣。
- *方向性*（direction）代表參數是用來*輸入*（in）、*輸出*（out）或*輸出入*（inout）用的。如果沒有秀出方向性的話，那麼我們會假設它是 in。

舉例來說，操作可能會寫成像下面這樣：

```
+ balanceOn (date: Date): Money。
```

在概念性模型中，操作中不該詳述類別的介面（譯註：詳述介面是採用軟體觀點的模型中的做法）。相反地，我們應該在操作中秀出類別的主要責任來。或許我們可以在操作中寫出匯總 **CRC** 卡片上責任的一些字（請參見第 5 章）。

我經常發現到：把會改變跟不會改變類別中狀態的操作區分開來是非常有益的事。UML 中將 **query**（查詢子）定義成「只會從類別中取出值、不會改變系統狀態」的操作。也就是說，它不會產生副作用。你可以把這種操作加上{query}（類別）性質字串。還有，我自己喜歡將會改變狀態的操作加上 **modifiers**（修改子）（類別）性質字串。

> 譯註：這裡將 query 翻譯成「查詢子」，是想強調它所扮演的角色。這樣一來，constructor（建構子）、desctructor（解構子）、modifier（修改子）等詞，就都有代表「角色」的一致譯詞。

嚴格來說，query 跟 modifiers 間的差異在於：它是否會改到可看到的狀態[Meyer]。可看到的狀態代表外界可察覺到的東西。負責更新快取的操作應該只會改變內部狀態，就外部的觀察者來說，他並不會察覺到這個改變。

我也發現到強調*查詢子*（query）的好處，因為我們可以用任何順序來執行這些查詢工作，可是*修改子*（modifiers）的執行順序就很重要。因此，常見的做法是在寫操作時，避免從修改子傳回值；這樣做的話，我們可確信：傳回值的操作都是在進行查詢動作。

除了查詢子與修改子之外，***讀取用方法***（getting method）與***設定用方法***（setting method）也是我們常見的兩種術語。讀取用方法只會傳回欄位的值（而且只作這麼多）。設定用方法也只會把值寫入欄位中（而且也只作這麼多）。從類別的外部來看，類別的使

用者並無法辨別操作究竟是查詢子還是讀取用方法，也無法辨別它是修改子或設定用方法。判別資訊完全存在於類別內部。

另一個是操作跟方法間的區別。**操作**指的是我們可呼叫物件的某種東西 － 它代表*程序宣告*（procedure declaration），而**方法**則是*程序本體*（procedure body）。這兩種東西在*多型*（polymorphism）存在時，會有很大的差異。例如如果你有一個*超型態*（supertype）與三個*子型態*（subtype），而且每個子型態都會覆寫掉超型態的 getPrice 操作。這時候，我們就有一個操作、四個方法（譯註：一個程序宣告與四個程序本體。）

大家經常會混用*操作*跟*方法*這兩個術語，可是有時候明確指出兩者不同是很有幫助的。

一般化關係

某種生意的個人客戶與企業客戶是**一般化關係**的典型例子。兩者不同，卻也存在許多的相似性。我們可以把相似的地方放在一般性的 Customer（客戶）類別中（它代表*超型態*【supertype】），而 Personal Customer（個人客戶）與 Corporate Customer（企業客戶）則是*子型態*（subtype）。

上述情況會因為我們對模型有不同觀點而有不同解釋。概念上，如果所有 Corporate Customer（企業客戶）的*實例*（Instance）從定義來看都是 Customer（客戶）的實例，那麼我們就會認為 Corporate Customer 是 Customer 的子型態。由此可看出來，Corporate Customer 是一種特別的 Customer。有一個觀念很重要，那就是所有跟 Customer 有關係的東西 — 包括關聯、屬性與操作等，對 Corporate Customer 來說也都是成立的。

從軟體觀點來看，對一般化關係的一種明顯解釋就是*繼承*（inheritance）：Corporate Customer 是 Customer 的*子類別*（subclass）。在主流的 OO 程式語言中，子類別會繼承*超類別*（superclass）的所有（類別）特性，而且還可能會覆寫任何超類別的方法。

如果我們想要有效使用繼承，那麼***可替代性***（substitutability）就是一個很重要的原則。我應該可以將程式碼中所有用到 Customer 的地方都換成 Corporate Customer，而且所有工作都應該要能夠進行得很好。基本上，程式中存在 Customer 的地方，都可以任意把它換成 Customer 的任何子型態。當然，因為多型的緣故，Corporate Customer 中有些命令的執行結果可能會跟其他種類的 Customer 有所不同，可是命令的呼叫者並不需要擔心這些差異。（如果你想了解更多東西的話，那麼請參閱[Martin]一書中的 *Liskov 替代原則*【Liskov Substitution Principle，LSP】。）

雖然繼承是很有力量的一種機制，不過它也帶來很多包袱，不一定能做到可替代性。一個很好的例子就是 Java 初期剛出來的時候，許多人不喜歡內建 Vector 類別的實作方式，而且希望用比較小的幅度來改寫它。然而，當我們想要產生一個可替代 Vector 的類別時，唯一的方式就是產生它的子類別，這也意味著我們必須從 Vector 繼承下來許多不想要的資料與行為。

有許多其他機制可以讓我們產生可替代的類別。因此，許多人喜歡將*子型態化*（subtyping）跟*介面繼承*（interface inheritance）區別開來，也喜歡區別*子類別化*（subclassing）跟*實作繼承*（implementation inheritance）的不同。如果某個類別可以替代掉它的超型態，那麼不論它有沒有用到繼承，都是**子型態**。至於**子類別化**則被視為一般所謂繼承的同義字。

有許多機制讓我們不需用到子類別化，就可以做到子型態化，「實作某種介面」（請參見第 5 章）與許多典型的*設計樣式*（design pattern）[Gang of Four]等都是一些常見的例子。

便條符號與註釋

在圖中，我們用*便條*（note）表示法來代表*註釋*（comment）。便條符號可以自己存在，也可以用虛線把它跟要註解說明的元素連結在一起（請參見圖 3.6）。而且，它可以出現在任何種類的圖形當中。

> 譯註：原本在 UML 1.x 版中，只有 Note 這個術語。在 UML 2 版中，改用 Comment 這個術語，而將它的表示法稱為 Note Symbol。因此，在 UML 1.x 版中，我們將 Note 翻譯成「註釋」；現在，則將 Comment 翻譯成註釋，而將 Note Symbol 翻譯成「便條符號」作為區別。

有時候虛線不是很好處理，因為我們很難知道線條到底在哪裡結尾，所以常見做法是在線條的結尾處畫上非常小的開口圈圈。有時候，我們最好將註釋穿插在圖形元素裡面。這時候，我們可以在說明文字前面加上兩條破折號：--，以表示它是註釋。

圖 3.6 *我們用便條符號替圖中一個或多個元素加上註釋*

相依性

如果我們改變某個元素（**供應端**【supplier】）的定義可能會影響到其他元素（**客戶端**【client】），那麼這兩個元素間就存在著*相依性*（dependency）。對類別來說，相

依性存在的原因有好幾個：某個類別會傳送訊息給另一個類別；某個類別把另一個類別當成一部分資料；某個類別把另一個類別當成方法中的參數等。如果這個類別改變它的介面，那麼任何傳送給這個類別的訊息就可能變得無效。

當電腦系統不斷成長，你必須越來越擔心該如何去控制相依性。如果你沒辦法控制相依性的話，那麼我們對系統所做的任何改變都會造成漣漪效應，讓越來越多東西跟著改變。一旦可能發生的漣漪效應越大，那麼我們就很難去改變任何東西。

譯註：軟體的本質就是「變動」。不管是設計樣式或是終極（程式開發）流程，其目的都是為了配合軟體使用環境的劇烈變化，迅速改變軟體本身。為了避免軟體間的相依性影響到它「可變」的本質，我們通常會用介面將兩個子系統間的實作隔離開來。除此之外，*構面導向程式設計*（aspect-oriented programming，AOP）與*模型驅動開發架構*（model driven architecture，MDA）等都希望我們在設計程式或模型時，可以先根據不同的架構因子分開設計，最後再把它們合併起來，成為最終需要的軟體。

UML 允許我們描繪出各種元素間的相依性。不論何時，當你希望秀出「某個元素中的變動可能會影響到其他元素」時，都可以用相依性來表現它。

圖 3.7 中秀出多層式應用程式中可能會存在的一些相依性。Benefits Windows（津貼視窗）類別 － 使用者介面或***展示類別***（presentation class）－ 會受到 Employee（員工）類別的影響。Employee 代表***領域物件***（domain object），裡面會捕捉系統的必要行為，所謂的必要行為在這個例子中指的就是*企業規則*（business rule）。相依性意味著：如果 Employee 類別改變它的介面，那麼 Benefits Windows 勢必也要跟著改變。

這裡有一件事很重要，那就是相依性只有一個方向 － 從展示類別往領域物件。就此而言，我們知道自己可以自由自在地改變 Benefits Windows，而不會對 Employee 類別或其他領域物件發生任何影響。我發現到：將展示介面跟領域邏輯嚴格分開，並且讓

展示介面相依於領域物件，而不讓領域物件相依於展示介面，是非常值得大家遵循的一個準則。

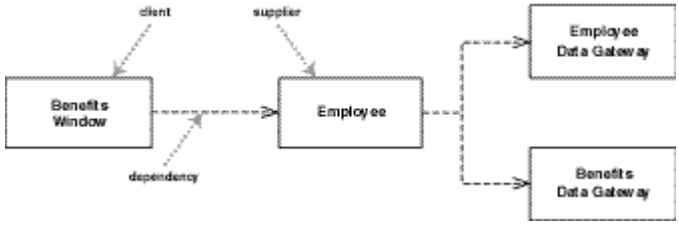


圖 3.7 *相依性的範例*

第二件值得注意的事就是：在這張圖中，Benefits Windows 跟兩個*資料閘道*（data gateway）類別間沒有直接的相依性。如果後者改變的話，那麼 Employee 類別可能會跟著改變。不過如果變動只發生在 Employee 類別的實作，而不在它的介面，那麼變動會就此打住。

UML 中定義了很多種的相依性。每種都有它自己特殊的語意與關鍵字。前面所描述到的*基本相依性*（basic dependency）是我自己發現到最有用，而且通常不用寫出關鍵字的相依性。不過，為了描述出更多細節，我們還是可以替相依性加上合適的關鍵字（請參見表 3.1）。

基本相依性所代表的是不具轉移性的關係. 這裡舉一個具轉移性的關係 － XX 比 OO 有更多的鬍子. 假如 Jim 比 Grady 有更多的鬍子, 而且 Grady 又比 Ivar 有更多的鬍子, 那麼我們可以推論說：Jim 有比 Ivar 更多的鬍子。有些種類的相依性，例如 substitute（替代）是有轉移性的，不過在大部分的情況下，直接與間接的相依性是有很顯著差異的，就像圖 3.7 中的情況一樣。

UML 中有許多關係都隱含著相依性。像在圖 3.1 中，從 Order（訂單）到 Customer（客戶）這個具有可瀏覽性的關聯就意味著 Order 會相依於 Customer。還有，子類別會相依於它的超類別，反之則不然。

表 3.1 *本書精心挑選出來一些有用的相依性關鍵字*

關鍵字	**意義**
<<call（呼叫）>>	來源元素會去呼叫目標元素的某個操作
<<create（產生）>>	來源元素會產生目標元素的實例
<<derive（衍生出）>>	來源元素是由目標元素所衍生出來的
<<instantiate（實例化）>>	來源元素是目標元素的實例。（請注意，如果來源元素是類別，那麼這個類別本身就是 class 類別的實例；換言之，目的類別就是*超模型類別*【metaclass】。）
<<permit（允許）>>	目標元素允許來源元素去存取目標元素中的私有(類別）特性。(譯註：例如在 C++中某個類別的*友善類別*【friend】就可以存取此類別中的私有屬性。)
<<realize（實現）>>	來源元素是目標元素中所定義規格或介面的實作。
<<refine（修飾）>>	修飾代表不同語義等級間的關係；例如，來源元素可能是設計類別，而目標元素則是相對應的分析類別。
<<substitute（替代）>>	來源元素是可替代目標元素的。
<<trace（追蹤）>>	我們用這個相依性來追蹤一些東西，例如從需求到類別

	的相依性，或某個模型中的變動是如何連結到其他模型中變動的。
<<use（使用）>>	來源元素在實作時需要用到目標元素。

▼ 49

一般性原則是盡量降低相依性，特別是相依性會氾濫到系統中的一大片區域時。還有一點要注意的是：我們應該小心避免循環式相依性，因為它會導致循環性的變動。我自己並不會嚴格禁止這種狀況發生。對於關係非常密切的類別，我不會在意它們間具有彼此交互的相依性，不過我會避免較大範圍的循環式相依性，特別是套件間。

試圖秀出類別圖中的所有相依性是一件徒勞無功的事；圖中會有太多的相依性存在，而且它們也常常在變。請根據你想要說明的課題，選擇性地秀出跟課題直接相關的相依性。為了了解與控制相依性，我們最好只在*套件圖*（package diagram）中畫出相依性。

我會畫出類別間相依性的其中一種常見情況是：當我想秀出一些非永久性關係時，例如某個物件被當成參數傳給另一個物件。這時候，你可能會看到我們替相依性加上<<parameter>>、<<local>>或<<global>>等關鍵字。在 UML 1 版的模型中，有人也會把這些關鍵字加在關聯上，這時候它們代表一些非永久性的連結關係，而非（類別）性質。不過，UML 2 版中把這些關鍵字都刪掉了。

我們要查看程式碼才能決定相依性，所以用工具來進行相依性的分析是最理想的做法。用一種工具以反向工程方式畫出相依性是使用這部分 UML 的最有用方式。

限制規則

大部分畫在類別圖中的東西都是一種*限制*（constraints）。圖 3-1 表示 Order（訂單）只會由某位 Customer（客戶）所下。圖上也隱含每個 Line Item（明細項目）都是獨立的：你可以有 40 個棕色的裝飾品、40 個藍色的裝飾品和 40 個紅色的裝飾品，而不是 40 個紅色、藍色加灰色的裝飾品。此外，圖上告訴我們 Corporate Customers（企業客戶）有信用額度，而 Personal Customers（個人客戶）卻沒有。

關聯、屬性與一般化關係的基本結構中已經標示出很多重要的限制了，可是它們仍然無法指出所有的限制。有些限制仍需要額外記錄下來，而類別圖就是記錄限制的好地方。

UML 允許我們用任何東西來描述限制。唯一的規則是：請把限制放在大括號{}內。你可以用自然語言、程式語言或 UML 所提供、正式的*物件限制語言*（Object Constraint Language， OCL）[Warmer 與 Kleppe]等來寫限制（OCL 是以述語微積分為基礎發展出來的）。採用正式的表示法可免除模糊不清的自然語言本身造成誤解的風險。然而，這樣做還是有風險的，因為寫或讀的人對 OCL 的真實含意可能還是會有誤解發生。所以除非讀的人習慣述語微積分這樣的東西，不然我還是建議大家採用自然語言。

▼ 50

有一種選擇性的做法是：替限制命名時，把名稱放在最前面，後面緊接著一個冒號，像這樣{不允許亂倫: 夫妻雙方彼此不能是兄弟姊妹}。

合約式設計

合約式設計（Design by Contract）是由 Bertrand Meyer[Meyer]所發展出來的一種設計技術。此技術是 Eiffel 程式語言的主要語言特性。然而，合約式設計並不是 Eiffel 所特有的，它是可以用在任何程式語言上的一種很有價值的技術。

合約式設計的核心是*假定*（assertions）。**假定**代表永遠應該成立的布林（boolean）陳述，因此它只在程式發生臭蟲時不成立。一般來說，假定只會在程式除錯期間檢查，而不會在產品執行期間檢查。事實上，程式執行時從不認為假定也正被檢查中。

合約式設計中用到三種特定的假定：*事後條件*（post-conditions）、*事先條件*（pre-conditions）和*不變條件*（invariants）。事先條件與事後條件都用在操作上。**事後條件**中描述操作執行後應該得到結果。舉例來說，如果我們定義一個數字的「平方根」操作，那麼它的事後條件就可以寫成 input=result*result，其中輸出為 result，而輸入值為 input。事後條件中通常只說明我們會得到什麼結果，而不說明如何得到這些結果。換言之，它將介面跟實作區分開來。

事先條件中描述操作執行前應該具備的條件。舉例來說，如果我們定義一個數字的「平方根」操作，那麼它的事先條件可能是 input>=0。這樣的事先條件告訴我們：用負數呼叫「平方根」操作會造成錯誤，其結果是不可預期的。

一開始，我們可能會覺得這樣做好像不好，因為我們在呼叫「平方根」操作時需要多做些檢查動作。其實，關鍵點在於誰該負責檢查。

事先條件明白指出檢查動作的責任歸屬於呼叫者。如果沒有明白指出責任歸屬，我們可能會少作了一些檢查動作（因為雙方都假設那是對方的責任），也可能多作了一些檢查（雙方都作了檢查）。多作檢查動作也是一件不好的事，因為它造

成重複的程式碼，而且可能會增加程式複雜度。明白指出責任歸屬可降低複雜度。因為假定通常事先在除錯與測試時就進行檢查，所以呼叫者忘記進行檢查的風險就變得不高了。

譯註：軟體開發流程可分為*紀律導向*（discipline-oriented）或*溝通導向*（comminucation-oriented）的，前者以開發流程為核心，後者以人為核心。所以解決檢查問題，也是有不同做法的。在紀律導向的開發流程中，我們可能會用合約式設計來解決檢查問題，而溝通導向的開發流程就可能是以*集體式程式碼擁有權*（collective code ownership）來解決它。

▼ 51

根據事先條件與事後條件的定義，我們可以嚴格定出***例外情況***（exception）這個術語：如果操作被呼叫時符合事先條件，可是傳回值卻不符合事後條件，那麼這時候就是發生例外情形了。

不變條件（invariants）是跟類別有關的假定。舉例來說，會計類別可能有一個不變條件，裡面描述 *balance == sum(entries.amount())*。對這個類別的所有實例來說，不變條件代表「一定」要成立的條件。「一定」的意思是說：只要物件存在，而且物件的操作可以被呼叫，那麼這時候不變條件中所描述的條件就必須成立。

基本上，不變條件表示這個條件應該存在於類別中所有公開操作的事先條件與事後條件當中。在方法執行期間，不變條件可能會暫時不成立，可是當其它物件可以對這個訊息接收者進行任何動作時，不變條件中所描述的條件就應該還原到成立狀態。

假定對*子類別化*（subclassing）來說具有一種特殊含義。繼承可能帶來的風險之一就是你重新定義的*子類別*（subclasses）操作跟*超類別*（superclasses）的操作不一致。假定可以降低這個風險。類別的不變條件和事後條件一定要適用在所有子類別身上。子類別只可增強不變條件和事後條件，卻不能降低它們。另一方面，事先條件則只可減弱，而不能增強。

前面那段話剛開始看起來會覺得有點怪，可是它對*動態繫結*（dynamic binding）來說是很重要的。因為根據*可替代性原則*（the principle of substitutability），你應該永遠能夠將子類別的物件當成超類別的實例看待。假如子類別增強它的事先條件的話，那麼把原先對超類別所進行的操作呼叫，改成呼叫子類別的，就可能會發生問題（譯註：因為子類別操作的事先條件較嚴格，所以對子類別進行的操作呼叫可能會失敗。）

何時使用類別圖

類別圖是 UML 的主幹，因此你可以會發現隨時都用得到它們。本章只涵蓋一些基本概念，第 5 章有其他的高等概念。

▼ 52

類別圖的最大麻煩在於裡面包含太多東西了，所以它們可能會被過度使用。下面是一些使用上的提示：

- 不要試著用到所有可以使用的表示法。先簡單用本章所提及的一些東西：包括類別、關聯、屬性、一般化關係與限制等。至於第 5 章中所介紹到的東西，只在你覺得有需要時再去用它們。

- 我發現到概念性的類別圖在探索某種企業的領域語言時特別有用。想要這麼做時，你必須努力保持不去討論軟體本身，而且盡可能使用非常簡單的表示法。

- 不要畫出包含所有東西的模型。相反地，把重心放在關鍵地方。持有一些常用的到、時時更新的圖，比起持有很多被人遺忘掉、過時的圖還好。

使用類別圖的最大危險在於你可能會過度把焦點放在結構上，而忽略系統中的行為。因此，當你在畫類別圖以了解軟體本身時，請同時配合某種行為分析技術。當你覺得事情進行的很順利時，這時候你會發現到自己正不斷交替使用各種不同的結構或行為分析技術。

如何獲得更多資訊

所有我在第 1 章中所提到跟 UML 有關的書，都對類別圖有做詳盡的介紹。對比較大的專案來說，相依性的管理是很關鍵的地方。在這方面，個人覺得最棒的書是[Martin]。

Chapter 4
循序圖

▼ 53

我們會在*互動圖*（interaction diagrams）中描述一群物件間的行為，說明它們是如何一起合作的。UML 中定義了好幾種互動圖，其中最常見的一種就是*循序圖*（sequence diagram）。

一般來說，我們會在一張循序圖中畫出某個*情節*（scenario）的相關行為，圖中會秀出這個*使用案例*（use case）裡面可能出現的一些物件，以及在物件間傳送的訊息。

從這裡開始，本章將透過一個簡單情節，做循序圖方面的相關討論。假設我們現在有一份訂單，並且準備要呼叫它的一個命令，算出這份訂單的價格。為了達到這個目的，訂單需要查看它裡面所擁有的一些訂單明細、決定它們的價格，價格決定方式是以訂單明細所中包含產品之定價規則為基礎決定的。對所有訂單明細做完上述動作之後，接下來訂單要算出整體折扣，這時候它是以跟客戶綁在一起的規則為基礎算出整體折扣的。

圖 4.1 中所秀出來的就是這個情節的一種實作方式。循序圖秀出互動情形的方式是讓每個*參與物件*（participant）有一條*生命線*（lifeline），訊息會由頁面上方往下依序執行，而且訊息的先後順序也是沿著頁面往下看的。

對循序圖來說，它的好處是：我幾乎不用解釋這個表示法，大家就能夠了解它。從圖中，我們可以看到訂單實例會送出 getQuantity 與 getProduct 訊息給訂單明細。大家也可以看到訂單如何呼叫自己的方法，而這個方法又送出 getDiscountInfo 訊息給客戶實例。

然而，這張圖並不能很適當地秀出所有東西。對訂單來說，它需要將一系列的 getQuantity、getProduct、getPricingDetails 與 calculateBasePrice 訊息送給每個訂單明細；不過，它卻只需要呼叫 calculateDiscounts 一次。我們無法在這張圖中區別兩者間的差異。稍後，我們將介紹更多表示法，以表現出這種差異。

大部分時候，你可以將互動圖中的**參與物件**視為物件就好，因為在 UML 1 版中，它們真的就是如此。不過，在 UML 2 版中，它們的角色變得更加複雜，而且完整解釋這些角色事實上也超過本書範圍了。所以我在本書中會用**參與物件**這個術語來代表這些角色，不過這個術語在 UML 中並不是一個正式用字。在 UML 1 版中，參與物件代表物件，所以它們的名稱會被加上底線，不過 UML 2 版中在秀出物件時並不需要加上底線，所以我這裡也這麼做。

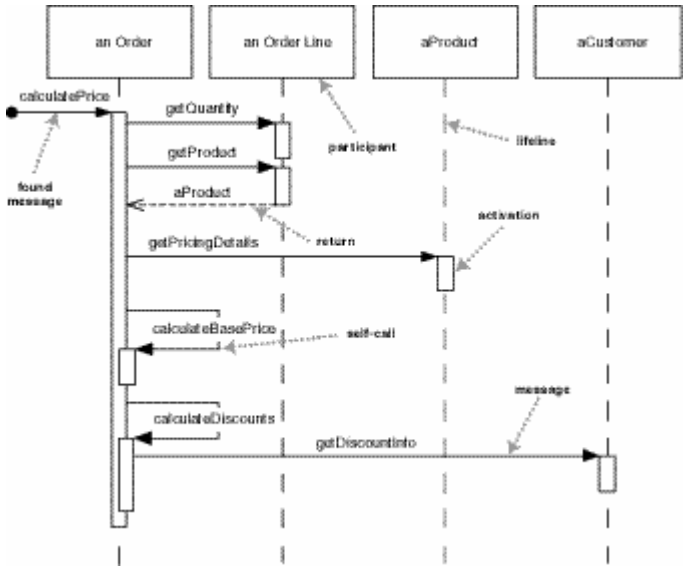


圖 4.1 *採用集中式控制設計風格所畫出來的循序圖*

在本書的圖中，我會用 anOrder 這樣的風格來替參與物件命名。大部分時候，這種做法是很恰當的。它的完整語法是*名稱:類別* name: Class ；其中，名稱與類別都是可以選擇要不要的，不過如果我們有寫出類別，那麼也必須跟著寫出冒號來。（圖 4.4 就是採用這種風格。）

每條生命線上都會有*活化長條*（activation bar），以秀出互動情形中參與物件處於活化狀態的時候。這些活化長條都分別會跟參與物件被呼叫的方法中的其中一個相對應。在 UML 中，活化長條是可秀或不秀的，不過個人發現到秀出它們對釐清行為來說是很有用的。唯一的一個例外情形是：當你在設計會議中討論某種設計方式時，請不要秀出活化長條，因為在白板上畫它是一件很可怕的事。

對圖中的參與物件來說，光是看一些命名方式就可以把它們關聯在一起。從圖中，我們可以看到 getProduct 的呼叫會傳回 aProduct，它跟接下來 getProcingDetails 呼叫時

所傳送的目的地 aProduct 名稱一樣，因此兩者指的是同一個參與物件。請注意，我只有替這個呼叫畫出*傳回箭號*（return）來；之所以這麼做是為了秀出這兩個呼叫間的相關性。有些人會替所有呼叫畫出傳回箭號，不過我只會在這些傳回箭號有助於提供更多訊息時畫出它們。除此之外，畫出它們只會讓畫面變得更加凌亂而已。甚至是這個例子，就算不畫出傳回箭號，看圖的人也不會感到混淆。

譯註：傳回箭號一定是虛線，不過箭頭就有可能是空心、棒狀的（代表它是非同步訊息的傳回箭號），或是實心的（代表它是同步訊息的傳回箭號）。本書作者似乎一律採用空心、棒狀的箭頭。

第一個訊息沒有秀出傳送出這個訊息的參與物件為何，其原因是因為傳送這個訊息的來源不定，所以我們把這樣的訊息稱為***來源不明訊息***（found message）。

譯註 : 除了來源不明訊息之外 , UML 2 版中還有所謂的***目的不明訊息***(lost message)，因為這個訊息的目的地是不定的。當然，箭頭也有可能是空心、棒狀的（代表它是非同步訊息），或是實心的（代表它是同步訊息）。附圖就是一個同步的目的不明訊息。

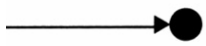


圖 4.2 中所秀出來的是這個情節的另一種實作方式。它解決的問題還是一樣，不過這些參與物件一起合作，以解決問題的方式卻非常不同。Order 要求每個 Order Line 算出它自己的價格。而 Order Line 又更進一步將計算工作轉交給 Product 做；請注意，這時候會傳一個參數給 Product。同樣地，為了算出折扣，Order 也會呼叫 Customer 的某個方法。因為 Customer 在做這件事時需要從 Order 得到一些訊息，所以它又做了一個再次進入 Order 的呼叫（getBaseValue），以取得必要資料。

關於這兩張圖，值得我們注意的第一點就是：循序圖可以很明顯秀出參與物件兩種不同互動情況間的差異。雖然循序圖不利於秀出演算法細節(例如迴圈或條件式行為)，

不過它卻可以很清楚地秀出參與物件間的呼叫情形，因此大家可以很清楚看出參與物件是如何進行相關工作的。

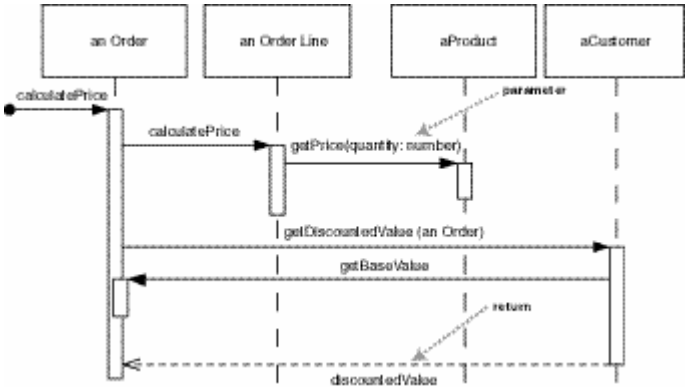


圖 4.2 *採用分散式控制設計風格所畫出來的循序圖*

第二點值得注意的是：我們可以很清楚地看出這兩種互動情形採用不同的設計風格。圖 4.1 中所採用的是***集中式控制***（centralized control）設計風格，它讓某個參與物件負責整個處理過程中的絕大部分工作，其他參與物件只負責提供資料而已。圖 4.2 則採用***分散式控制***（distributed control）設計風格，它將處理過程分散到許多參與物件身上，每個參與物件都負責完成演算法中的一小部分工作。

兩種設計風格各有優缺點。大部分的人，特別是一些物件方面的新手，比較常採用集中式控制的設計風格。許多方面，這種做法是比較簡單的，因為所有的處理過程都會在同一個地方完成；另一方面，如果我們採用分散式控制設計風格的話，追蹤程式如何執行時，就會有一種在物件間繞來繞去的感覺。

排除這種繞來繞去的感受，跟我一樣對物件堅持的人都比較喜歡採用分散式控制的設計風格。為了達到比較好的設計結果，我們主要的設計目標之一就是：盡量將變動可

能發生的影響侷限在一小塊範圍內。資料跟存取此資料的行為通常會跟著一起變動，所以把資料跟使用它的行為放在同一個地方就是物件導向設計的第一號法則。

此外，如果我們採用分散式控制設計風格，那麼往往會有比較好的機會去用到*多型*（polymorphism），而不用寫出條件式邏輯來。例如如果不同類的產品有不同的產品定價演算法，那麼分散式控制機制就可以讓我們用產品的子類別分別處理這些變異點。

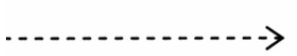
一般來說，OO 設計風格會用到很多小小的物件，而且每個物件裡面也會有許多小小的方法，這種做法讓我們有許多嵌入點可以去覆寫它們或產生不同做法。對習慣寫出又臭又長程序的人來說，對這種設計風格會感到非常困惑；事實上，這種設計風格上的改變正是物件導向**設計典範革命**的核心所在。不過，這也是很難教導之處。真正了解物件導向典範的唯一方式似乎就是在強烈採用分散式設計風格的 OO 環境下工作一陣子。許多人會突然恍然大悟，開始理解到這種設計風格究竟為何。這時候，他們的腦袋將重獲新生，也開始比較容易思考分散式控制的設計風格。

譯註：將一整塊工作分解開來，讓我們有機會在不同變異點應付多變的需求。不過，它要付出的成本也不少。所以，我們對程式設計的概念由簡單結構，慢慢轉向預先產生複雜結構，以應付需求未來可能發生的變動。稍後，又再轉變成：先採用簡單結構，再隨需求逐漸演化成必要的複雜結構。以*測試導向開發方式*（test driven development，TDD）來說，軟體會由簡單的設計開始，隨著*測試案例*（test case）增多，慢慢轉變成比較複雜、合用的設計結果。

參與物件的產生與刪除動作

循序圖中有一些額外的表示法可以秀出參與物件的產生與刪除動作（請參見圖 4.3）。為了產生參與物件，我們會把訊息箭頭直接畫向參與物件的長方形。如果你用建構子來產生物件，那麼這時候訊息名稱就是可有可無的，不過自己不管在什麼情況下都會用「new」來替這個訊息命名。如果參與物件在被產生之後立刻要做些事情（例如查詢命令），那麼你可以在參與物件長方形的下方馬上啟動它的活化長條。

譯註：根據 UML 2 版規格書，產生物件的訊息可以用空心、棒狀的箭頭，加上虛線畫出來，請參見附圖。



我們可以用一個大大的 X 標記來表示某個參與物件會被刪除。如果有某個訊息的箭頭指向 X 標記，代表這個參與物件很明確是由其他參與物件所刪除的；至於生命線末端上的 X 標記則代表參與物件是由本身刪除自己的。

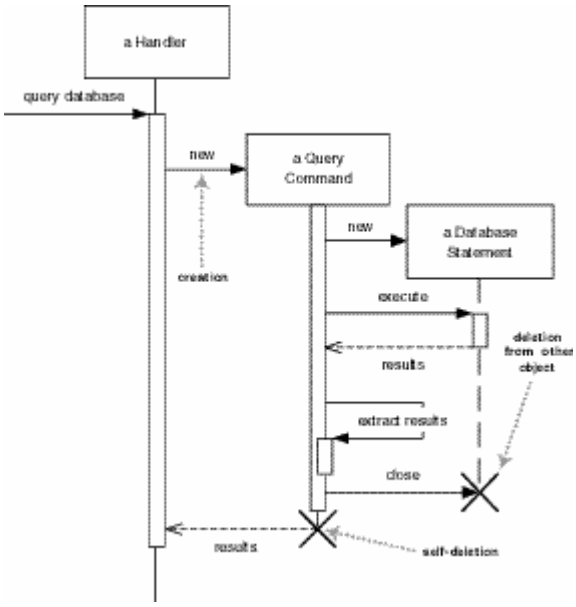
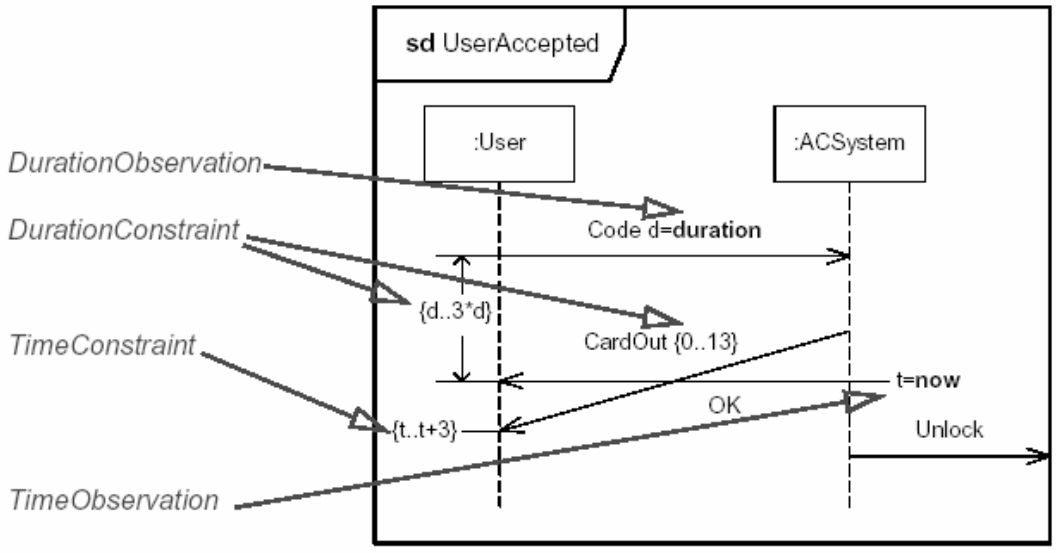


圖 4.3 *參與物件的產生與刪除動作*

在記憶體有垃圾回收機制的環境下（譯註：例如 JVM），我們不會直接刪除物件，不過大家還是可以用 X 標記指出：這時候已不再需要物件，請回收它。對一些結束操作加上 X 標記也是恰當的做法，因為它代表這個物件已經不再是可用的。

譯註：對循序圖來說，我們有時候可能會指定*期間限制*（duration constraint）與*時間限制*（time constraint），以說明訊息在時間上的限制條件。在下圖中，Code 訊息的*期間觀察值*（duration observation）為 d，代表訊息從 User 送到 ACSystem 所需花費的時間等於 d，而期間限制則為{d..3*d}，代表從 Code 訊息被送出來，到收到 OK 訊息為止，我們必須在 3*d 的時間內完成它；另一方面，我們也要求 CardOut 訊息要在 Code 訊息送出後的 13 秒內送到 User。最後，還有一個時間限制就是 CardOut 訊息要在 OK 訊息送到 User 後 3 秒內送達 User，而它的*時間觀察值*（time observation）為 now，代表 OK 訊息送到 User 時的時間。



來源：UML 2.0 Superstructure Final Adopted Specification

迴圈與條件式邏輯等互動框

循序圖中常見的一個議題是如何秀出迴圈與條件式行為。我們要先說明的是：循序圖並不善長做這種事。如果你希望秀出這類型的控制結構，那麼最好是用*活動圖*（activity diagram），甚至是程式碼本身來展現。請將循序圖視為物件如何一起互動的視覺呈現方式，而不要把它當成建立控制邏輯模型的一種工具。

▼ 58

說了這麼多，我們還是開始說明一些相關的表示法。迴圈與條件式邏輯都是用*互動框*（interaction frame）呈現的，它讓我們標示出循序圖中的一小塊來。下面是用*虛擬碼*（pseudocode）所寫出來的一個簡單演算法，而圖 4.4 就是根據它所畫出來的循序圖：

```
prodcedure dispatch
     foreach (lineitem)
```

```
            if (product.value > $10K)
               careful.dispatch
            else
               regular.dispatch
            end if
         end for
         if (needsConfirmation) messager.confirm
   end procedure
```

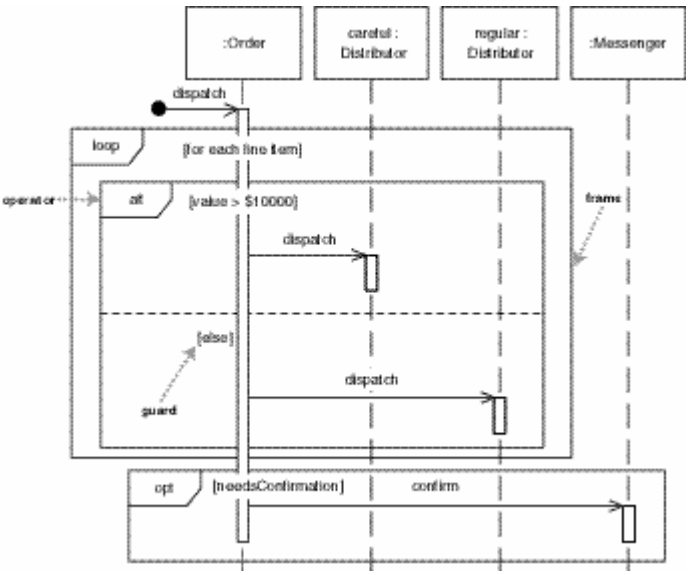
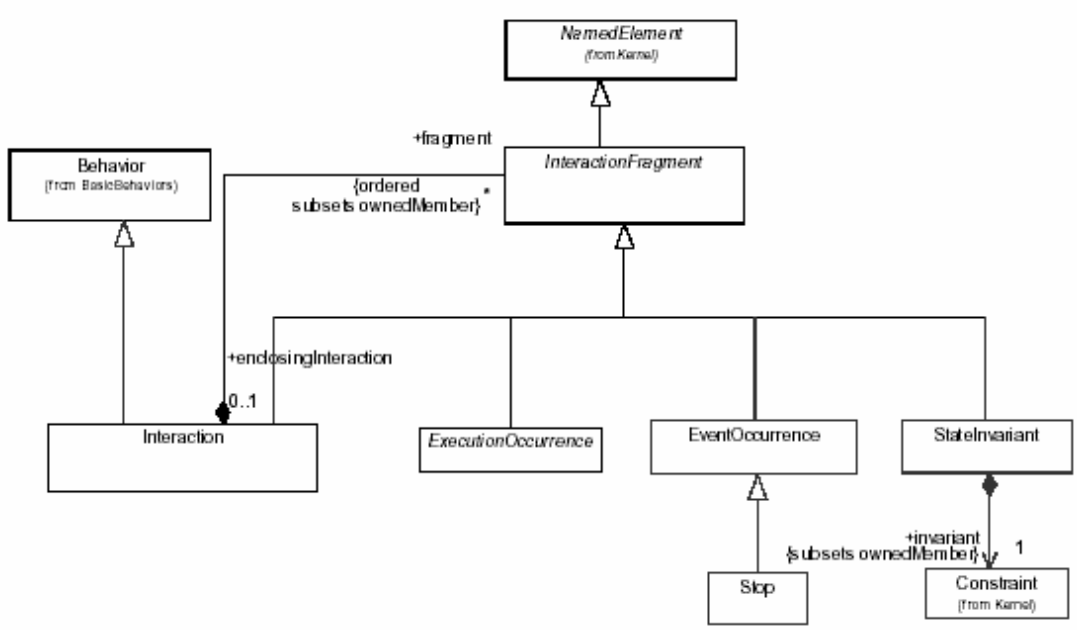


圖 4.4 *互動框*

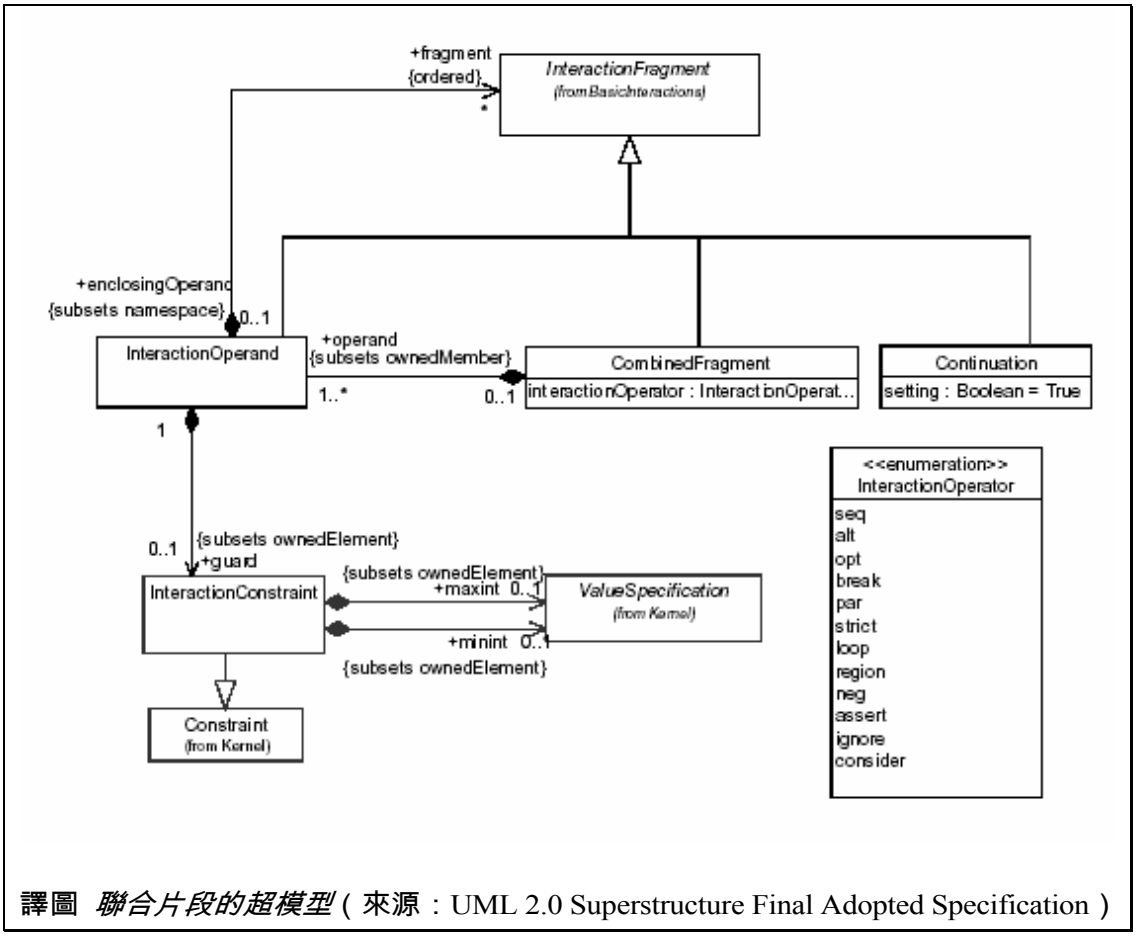
一般來說，互動框是由一小塊的循序圖所構成，裡面包含一個或多個*片段*(fragment)。每個互動框會有一個運算子，而且每個片段都可能會有一個*成立條件*(guard)。(表 4.1 列出互動框的常見運算子。) 為了秀出迴圈，我們使用 loop 運算子，它的運算元只有一個片段，成立條件位置所寫的就是反覆的執行條件。另一方面，對條件式邏輯來說，我們可以使用 alt 運算子，而且每個片段都可以有條件式。只有成立條件為真的那麼片段會執行。如果你只有一個片段的話，那麼可以改用 opt 運算子。

譯註 1：本書作者將任何用框框所圍起來的東西都叫互動框。UML 2 版規格書其實把它們分得更細。參照表 4.1，運算子為 sd 的叫做*互動情形*（interaction），代表某張循序圖。運算子為 ref 的叫做*互動實例*（interaction occurrence），我們如果將互動情形視為一個類別，那麼互動實例就是物件。運算子為 sd 與 ref 以外的叫做*聯合片段*（combined fragment），它會出現在運算子為 sd 的互動框當中。不過，本書作者認為運算子為 sd 的互動框是可畫或不畫的，所以圖 4.4 最外面應該可以畫出運算子為 sd 的一個互動框來。

譯註 2：fragment 的解釋也有點複雜。對互動情形來說，fragment 代表*互動片段*（interaction fragment）；就聯合片段來說，fragment 代表*互動運算元*（interaction operand）。



譯圖 *互動情形的超模型*（來源：UML 2.0 Superstructure Final Adopted Specification）



譯圖 *聯合片段的超模型*（來源：UML 2.0 Superstructure Final Adopted Specification）

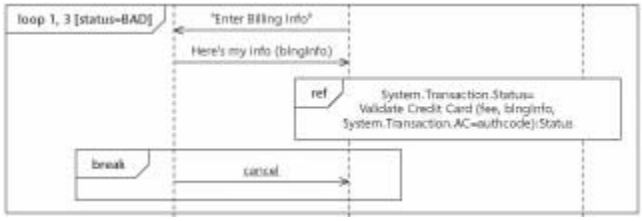
互動框是 UML 2 版才新增的。因此，你可能看到有人用 UML 2 版以前的其他方式來畫這些東西；除此之外，有些人並不喜歡互動框，反而比較喜歡一些老式畫法。圖 4.5 裡面就有一些非正式的畫法。

UML 1 版用*反覆標記*（iteration marker）與*成立條件*（guard）來畫迴圈。所謂的**反覆標記**就是在訊息名稱前面加上*。我們還可以用方括號加上一些說明文字，以作為反覆的執行條件。而**成立條件**就是方括號內的條件式，只有成立條件為真時，訊息才會送出去。雖然 UML 2 版中的循序圖把這些表示法刪掉了，不過通訊圖中這些表示法還是合法的。

反覆標記與成立條件雖然好用，不過它們有一些缺點。首先是成立條件無法說明一整組成立條件間彼此是互斥的，就像圖 4.5 中的情形一樣。其次，它跟反覆標記一樣都只適用於單一個送出去的訊息，如果迴圈或條件區塊在某段活化期間內有多個離開訊息，它們就不是那麼好用。

表 4.1 常用的互動框運算子

運算子	**意義**
alt（多選一）	在多個片段中選擇一個執行它；成立條件為真的那個片段才會被執行（請參見圖 4.4）。 譯註：alt 相當於 if…then…else…endif 或 switch case。
opt（可選用的）	可以選擇要不要執行的一個片段；它的成立條件為真時，這個片段才會被執行。它相當於只有一條*事件實例序列*（trace）的 alt 互動框（請參見圖 4.4）。 譯註 1：在 UML 2 版中 trace 有多種不同的意義；在互動情形中，它代表一條*事件實例序列*（sequence of eventoccurrences），所以我們這裡就把 trace 翻譯成「事件實例序列」。 譯註 2：opt 相當於 if…endif。
par（平行的）	可平行執行的幾個片段。 譯註 1：在 par 中，不同片段間的執行順序不是很重要。不過，同一個片段內的訊息還是要依時間軸先後執行。除了 par 之外，跟執行順序有關的運算子還有 seq 與 strict。seq 代表*弱式順序性*（weak sequencing），它比 par 嚴格點，兩個訊息如果是在

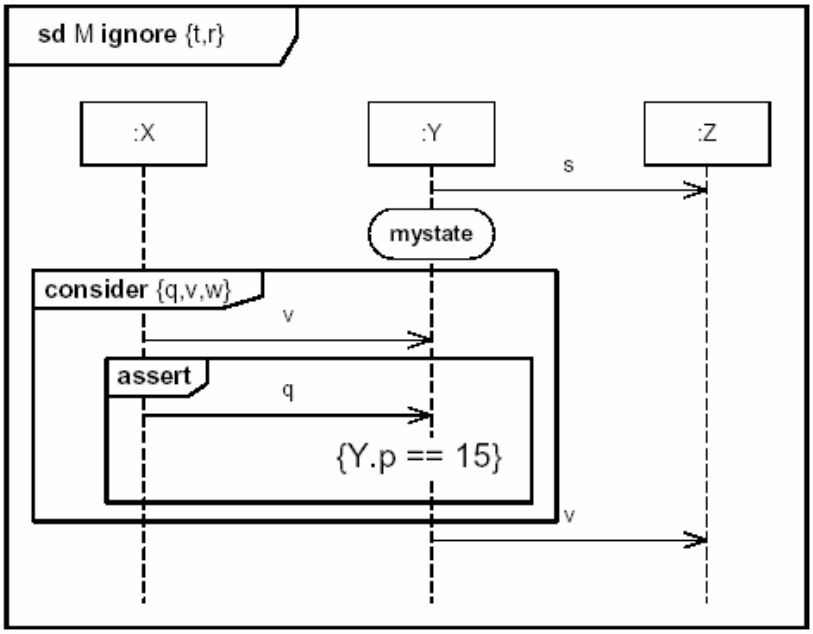
	同一條生命線上，縱然分屬不同片段，還是要依照個別所屬片段的先後順序來執行；strict 代表*強式順序性*（strict sequencing），它又比 seq 更嚴格些，明確規定不同片段的先後執行順序。總而言之，訊息的執行先後順序可從兩個等級來看。在片段內的，一定要依照時間軸的先後順序來執行。至於不同片段間的，就看整個互動框是 par、seq 或 strict 而決定。
loop（迴圈）	這個片段可能會執行多次，它的成立條件是反覆是否要執行的基礎。 譯註 1：根據 UML 2 版規格書，loop 的語法為 loop(minint, maxint) 其中 maxint 要大於或等於 minint。如果 maxint 為*，就代表它是無窮迴圈。如果 minint 與 maxint 都省略的話，代表 minint 為 0，而 maxint 為*。 譯註 2：一般在 loop 中，我們會用 break 運算子以中途跳出迴圈的正常執行過程，請參見下圖中的例子。  來源：UML 2 for Dummies
critical（臨界區域）	在這個片段裡面，每次只能有一個*執行緒*（thread）執行它。
neg（否定）	代表這個片段中所秀出來的是無效的互動情形。

譯註：跟 neg 相對的運算子是 assert（假定），代表這個片段中所秀出來的是唯一有效的互動情形。除此之外，我們可以用 ignore（忽略）運算子說明互動框中會認為這些訊息不重要，所以不會秀出它們；也可以用 consider（重視）運算子說明互動框中認為只有這些訊息是重要的，因此只會秀出它們。兩個運算子的語法為

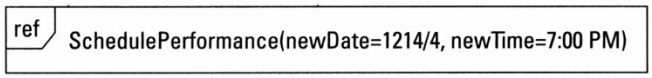
ignore {訊息串列}

consider{訊息串列}

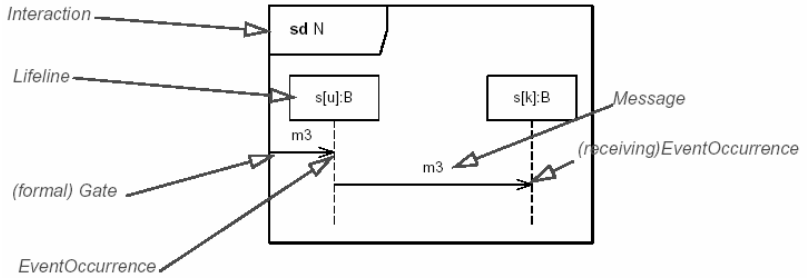
在下圖中，M 互動框中會忽略調 t 與 r 兩個訊息，並在其中一個片段中只重視 q、v 與 w 三個訊息而已。此外，它還預期 v 訊息之後只能出現 q 訊息而已。圖中也秀出*狀態不變條件*（state invariant），它代表一個*事先條件*（pre-condition），說明當 Y 的狀態為 mystate 時，下面的片段才是有效的；我們當然也可以在片段後面秀出狀態不變條件，以代表一個*事後條件*（post-condition），說明前面的片段執行後，這個條件必須成立。狀態不變條件有兩種表示法，圖中所秀出來的是*狀態圖示*（state icon），另一種是在大括號中用運算式來表示它，例如 {Y::mystate}。



來源：UML 2.0 Superstructure Final Adopted Specification

ref（參照）	用來指向另一張圖中所定義的互動情形。我們會讓它的框涵蓋互動情形中有關係的生命線。它可以定義參數與傳回值。 譯註：ref 的語法為 互動情況名稱 (參數串列): 傳回值 interactionname (arguments): return value  來源：UML Bible
sd（循序圖）	如果我們願意的話，可以把整個循序圖框起來變成一個互動框。

譯註：我們用 sd 運算子代表一張循序圖，裡面包含許多生命線與訊息。訊息的開始與結束分別叫做*（傳送端）事件實例*（(sending)EventOccurrence）與*（接收端）事件實例*（(receiving)EventOccurrence）。至於互動框跟外界的介面則稱為*閘門*（gate），如果把互動框視為函數的話，它就是互動框的參數與傳回值，請參見下圖。



來源：UML 2.0 Superstructure Final Adopted Specification

▼ 60

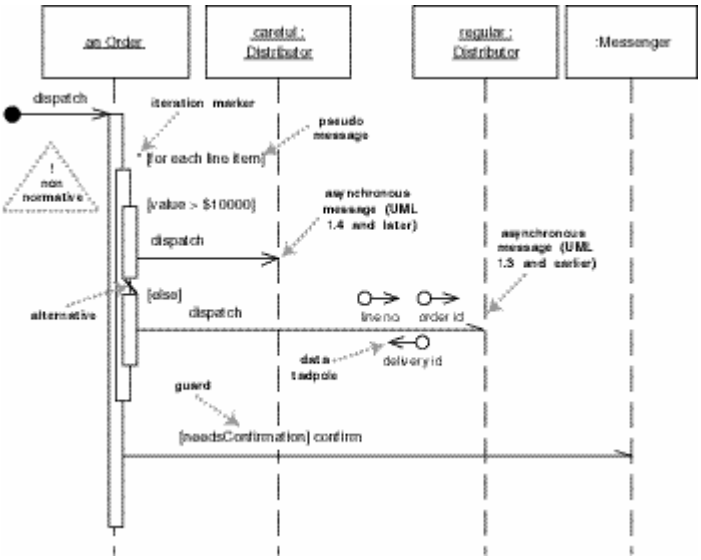


圖 4.5 *舊的控制流程畫法*

為了處理第二個問題，有一種很流行的非正式畫法是用***虛擬訊息***（pseudomessage）來畫它，它讓迴圈的執行條件或成立條件出現在自身呼叫的變種表示法上。在圖 4.5 中，我沒有秀出自身呼叫訊息的箭頭，有些人會把它秀出來。個人認為不秀出來可以讓大家更明確知道它不是真正的自身呼叫。有些人也喜歡將虛擬訊息的活化長條加上灰底。如果有替代性（譯註：多選一）行為的話，我們還可以在活化長條之間加上*替代性標記*（alternative marker）。

雖然我發現到活化長條是一種很有用的表示法，不過對圖 4.5 中的 dispatch 方法來說，它並不能多提供一些東西給我們，因為我們這時候只是送出訊息，接收者的活化長條上並沒有任何其他事情發生。圖 4.5 中將一些簡單呼叫的活化長條去掉是很常見的一種做法。

UML 標準中並沒有可以用來秀出資料傳遞情形的圖形表示法；相反地，它用訊息名稱中的參數與傳回箭頭來秀出所傳遞的資料究竟為何。有許多方法論都會用*資料蝌蚪符號*（data tadpole）來代表資料移動，有許多人喜歡在 UML 中畫出這些符號。

總而言之，循序圖中雖然新增了許多條件式邏輯的表示法，不過個人並不覺得這些表示法會比程式碼或虛擬碼好用。更特別的一點是，個人覺得互動框用起來非常笨拙，也會妨礙我們點出圖裡面的重點，所以個人還是比較喜歡用虛擬訊息的表示法。

同步與非同步呼叫

如果你的警覺性很高，那麼你一定有注意到本章中的一些圖裡面所畫出來的箭頭跟 UML 1 版中的箭頭不同。對 UML 2 版來說，這些小差異是很重要的。因為在 UML 2

版中，填滿的箭頭代表*同步訊息*（synchronous message），而空心、棒狀的箭頭代表*非同步訊息*（asynchronous message）。

如果呼叫者送出一個**同步訊息**，那麼它必須等到訊息完成工作為止，才能繼續執行下去，就像我們在呼叫副程式一樣。如果呼叫者送出一個**非同步訊息**，那麼它不用等到訊息回應，就可以繼續執行下去。我們可以在多執行緒或採用訊息導向中介軟體的應用程式中看到一些非同步呼叫。非同步做法可以讓應用程式有比較好的回應、降低暫時性的停滯情形，不過這樣的應用程式也比較難除錯。

同步與非同步訊息在箭頭上的差異是非常細微的；事實上，兩者間的差異太過細微了。此外，它們也跟 UML 1.4 版的不相容，之前非同步訊息是用半棒狀箭頭，如圖 4.5 所示。

個人認為目前兩種訊息的箭頭差異太小。所以如果你希望強調非同步訊息的話，那麼個人推薦你用 UML 2 版不再使用的半棒狀箭頭，這種畫法可以讓我們的眼睛更容易察覺不同情形。當你在看一張循序圖時，除非你能確定這張圖的作者會刻意畫出不同箭頭來，不然請小心不要假設這些箭頭一定都是同步的。

何時使用循序圖

想知道幾個物件在某個使用案例中的行為時，請使用循序圖。循序圖非常有利於秀出物件間的合作情形，它不適合用來產生物件行為的精確定義（譯註：就是精確寫出訊息的參數與傳回值）。

如果你想瞭解某個物件在不同使用案例間的行為時，可以用*狀態圖*（state diagram）（請參見第 10 章）。如果你想知道橫跨多個使用案例或執行緒的行為時，可以用活動圖（參見第 11 章）。

如果你想很快了解多個彼此替代的互動情形時，最好做法是使用 *CRC 卡*（CRC card），因為它可以避免我們在許多圖上塗塗抹抹。此外，用 CRC 卡會議來了解不同的設計替代方案也是一種很好的做法，至於以後會用到的互動情形，我們可以稍後再用循序圖畫出來。

通訊圖（communication diagram）是另一種有用的互動圖，我們可以用它秀出物件間的一些連結狀況。此外，我們也可以用*時序圖*（timing diagram）秀出一些時間上的限制。

CRC 卡片

在思考一個好的 OO 設計方式時，最有用的技術之一就是去探索物件間的互動情況，因為這麼做時重心都會放在行為，而不會放在資料上面。1980 年代末期由 Ward Cunninghan 所發明的 CRC（類別-責任-合作）圖（請參見圖 4.6）歷經時間考驗之後，已被證明是探索物件間互動行為的最有效方法之一。雖然它不屬於 UML 的一部份，不過對於熟練的物件設計師來說，它是一種很普遍的技術。

要用 CRC 卡時，你跟你的同事們要圍在一張桌子周圍，不斷地模擬各種情節，並且用卡片把情節表演出來。當某些類別處於活化狀態時，就在空中舉起它們，並且彼此傳送卡片，以表示它們之間是何傳送訊息與物件的。這樣的技術很難在書中描繪出來，不過卻很容易用真人示範它，所以學習它的最好方式就是請某位已經施行過這種技術的人帶著大家一起做。

檢查這項商品是否有庫存	Order Line（訂單明細）
決定價格	Customer（顧客）
檢查付款是否有效	
根據送貨地址送貨	

圖 4-6 *CRC 卡片的一個例子*

▼ 63

用 CRC 卡片來思考時，其中很重要的一個部分就是找出***責任***（responsibility）來。我們可以用一個很短的句子來摘要出物件該做的事情來，裡面說明：物件該做的動作、該維護的某種知識或該做出的重大決策等。想法是我們應該可以拿起任何一個類別，用一些責任來摘要說明它。這麼做可以讓你在設計類別時有更清晰的想法。

第二個縮寫 C 代表***合作者***（collaborator），它指的是那些需要跟這個類別一起合作的其它類別。它可以讓你用高階觀點了解類別間的繫結關係。

CRC 卡片的主要優點是它鼓勵開發人員之間做一些模擬討論。當你在揣摩類別如何實作出某個使用案例時，如果一開始就用本章介紹的互動圖，那麼你可能很難畫出物件間的互動情形來。通常你會考慮到很多種不同的替代方案，如果用互動圖來畫出這些替代方案的話，可能要花很長的時間把它們畫出來或擦掉。相反地，用 CRC 卡片時，你只要拿起卡片，然後把它們擺在一起就能夠描述類別間的互動情形了。這樣做可以讓你很快地考慮到各種替代方案。

當你試著這麼做時，通常會先對責任產生一些想法，然後把它們寫到卡片上。去思考類別的責任是一件很重要的事，因為這麼做讓你不把類別當作一堆資料的擁有者而已，同時也助於團隊成員瞭解每個類別的高階行為。一個責任可能會對應到某個操作、屬性或是一些操作與屬性（而且通常是如此）。

個人常見的一種錯誤就是：大家往往會產生一長串的低階責任。這樣做只會讓你覺得失焦。類別的所有責任都應該可以很容易地塞在一張卡片裡面。通常我會懷疑擁有超過三個以上責任的類別。這時候，我會試著問自己是否需要把這個類別加以分割，或是用比較高階的句子來描述責任。

許多人會強調角色扮演的重要性。進行角色扮演時，團隊裡的每個人可能會扮演一個或多個類別的角色。然而，我從未發現 Ward 這麼做，而且不這麼做還是一樣行得通。

許多書都有寫到 CRC 卡片,不過我發現這些書都沒有寫出這個技術的核心所在。由 Kent Beck 所寫，關於 CRC 的一篇最原始論文是[Beck and Cunningham]。為了對 CRC 卡片與設計中的責任有更深了解，個人推薦大家去看[Wirfs-Brock]這本書。

Appendix

UML 各版本間的改變

▼ 151

當本書第一版出現在書架上的時候，UML 的版本是 1.0 版。此時，UML 看起來好像已經穩定了，也通過 OMG 的認可程序。之後，它又歷經了數次的修訂。在附錄中，我會描述 UML 自 1.0 版以來的重大改變，以及這些改變是如何影響本書的。

如果你已經有本書前一版，看完附錄之後，你應該可以很容易就追趕上 UML 的最新改變。本書會隨著 UML 的改變而不斷更新，所以如果你拿到本書的其他後續版本，裡面都會描述 UML 到本書印刷日為止的改變。

UML 的修正版

UML 最早的公開版本是*統一方法論*(Unified Method) 的 0.8 版。它是由 OOPSLA 在 1995 年十月公佈的。這個版本是 Booch 跟 Rumbaugh 努力的結果，當時 Jacobson 還

未加入 Rational 公司。在 1996 年，Rational 公司發佈了 0.9 版跟 0.91 版，裡面就包括了 Jacobson 的一些貢獻。之後，它們被改名成 UML。

在 1997 年一月，Rational 公司跟其他的合作夥伴一起把 1.0 版的 UML 送到 OMG 的分析與設計工作團隊。接著，在 1997 年九月，Rational 公司結合 1.0 版的 UML 跟其它想法送出 1.1 版的 UML 給 OMG 作為標準提案。這個版本在 1997 年底被 OMG 所接納。然而，這裡有一些詭異的地方，OMG 把這個標準視為是 1.0 版的。從此，UML 就有所謂的 OMG 1.0 版跟 Rational 1.1 版。請不要把 OMG 1.0 版跟 Rational 1.0 版搞混了。事實上，每個人都把 OMG 1.0 版稱為 1.1 版標準。（譯註：之後如果沒有特別說明的話，一律以 Rational 的版本為準）

▼ 152

之後，UML 有一些進一步的發展。UML 1.2 版在 1998 年、1.3 版在 1999 年、1.4 版在 2001 年，而 1.5 版則在 2002 年出現。1.x 版間的變化幾乎都是看不見的小改變，只有 UML 1.3 版有一些明顯的改變，特別是*使用案例*（use case）與*活動圖*（activity diagram）。

當 UML 1.x 版不斷修正時，UML 的發展人員已經把眼光放在 UML 的重大修正 － UML 2 版上面了。最初的 RPF（提案需求書）在 2000 年提出，不過一直到 2003 年，UML 2 版才逐漸穩定下來。

UML 之後一定還會繼續發展下去。UML 論壇（http://uml-forum.com）是尋找更多資訊的好地方。我也會在個人網站（http://martinfowler.com）中放一些 UML 相關資訊。

UML 精華一書的改變

隨著 UML 改版，我也試圖讓 ***UML 精華*** (UML Distilled) 一書不斷改版。這也讓我有機會去修正錯誤，寫的更清楚些。

本書第一版是跟隨 UML 改變而修正、變化最劇烈的一段時期，我們會讓每一刷都跟上 UML 標準的最新變化。第 1 刷到第 5 刷都是以 UML 1.0 版為基礎。每刷之間的改變很小。到第 6 刷的時候，我們已經將 UML 1.1 版放進去了。

第 7 刷到第 10 刷則是以 UML 1.2 版為基礎。第 11 刷首次採用 UML 1.3 版。每刷的封面上都會寫出相對應的 UML 版本。

第二版的第 1 刷到第 6 刷是以 UML 1.3 版為基礎的。第 7 刷開始將 UML 1.4 版的小改變考慮進去。

第三版是為了讓本書更新變成 UML 2 版 (請參見表 A.1)。在這個附錄的其它部份，我會摘要說明 UML 從 1.0 到 1.1、1.2 到 1.3 與 1.x 到 2.0 間的改變。這裡不會討論到 UML 的所有改變，只說明本書有提及的部分，或是我覺得很重要的特性。

我會繼續保持本書的精神：討論 UML 的關鍵元素，因為它們會影響到 UML 是如何應用在真實專案中的。不過，這些選擇跟忠告都是我自己的想法。如果我說的跟 UML 官方文件相衝突的話，UML 的官方文件還是大家要遵循的。 (但是請告訴我，讓我能夠修正錯誤。)

感謝讀者告訴我一些重要的錯誤跟遺漏，讓我有機會在重印時更正它們，感謝大家。

表 A.1 UML 精華跟相對應的 UML 修訂版

UML 精華	UML 修訂版
第一版	UML 1.0-1.3 版
第二版	UML 1.3-1.4 版
第三版	UML 2 版之後

UML 從 1.0 版到 1.1 版的改變

型態與實作類別

在本書第一版中，我談到觀點，以及觀點是如何改變人們畫模型與解釋模型的方式，觀點對*類別圖*（class diagrams）的影響最大。UML 現在已經允許類別圖中的所有類別都能夠*特別化*（specialized）成為*型態*（type）或是*實作類別*（implementation class）了。

實作類別相當於軟體開發時寫程式的 class。而**型態**則相當模糊。型態用代表實作限制比較少的*抽象化概念*（abstraction）。它可以是 CORAB 的 types、規格觀點的*類別*（class）或是概念性觀點。如果有需要的話，我們還可以加上*造型*（stereotype），讓它們變得不同。

你可以畫一張特殊的圖，說明所有的類別都遵循某種特殊造型。當我們用某種特殊觀點來畫圖時，就可以這麼做。在實作觀點中，我們會使用實作類別，而在規格觀點跟概念性觀點中，我們應該會用型態。

大家可以用*實現關係*（realization）來表示某個實作類別會實作一個或多個型態。

型態跟*介面*（interface）之間有些不同。介面相當於 Java 或 COM 中的 interface，所以介面中只有*操作*（operation），沒有*屬性*（attribute）。

你可以將單一、靜態*類別化*（classification）用在實作類別上，而將多重、動態類別化用在型態上。（我會這樣假設是因為主要的 OO 程式語言都只能作單一、靜態類別化，假如有一天你所使用的程式語言有支援多重、動態類別化，那麼這個限制就消失了。）

▼ 154

完整和不完整的辨別子限制

在本書前幾刷中，我曾說明：當{complete}*限制*（constraints）用在*一般化關係*（generalization）時，代表*超型態*（supertype）的所有*實例*（instance）都必須是這群*子型態*（subtype）中，某個子型態的實例。UML 1.1 版中則定義{complete}代表全部子型態都已經被畫出來了。兩者意義不全然相同。我發現這個限制會有很多不同解釋，所以請小心處理它。假如你希望表示超型態中的所有實例都必須是這群子型態中某個子型態的實例，那麼我建議你使用其他限制，以免產生混淆。現在我都是用{mandatory}限制。

合成關係

在 UML 1.0 版中，對*單值的*（single-valued）*元件*（component）來說，使用*合成關係*（composition）隱含這個*關聯*（association）是*不可變*（immutability）或*凍結不變*（frozen）的。這個限制現在已經被拿掉了。

不可變限制與凍結不變限制

UML 用{frozen}限制來定義*關聯角色*(association role)是不可變的。根據現在的定義，它似乎不可應用在屬性或類別上。就個人的實際情況來說，我會用 **frozen** 而不用 immutability，而且我也很喜歡把它應用在關聯角色、屬性或類別上。

循序圖上的訊息回覆

在 UML 1.0 版中，*循序圖*(sequence diagrams) 中*訊息回覆*(return) 是用棒形的箭頭表示，而不用實體三角形的箭頭來表示 (請參見本書前幾刷)。這樣的差別很小且不明顯，很容易就沒注意到。UML 1.1 版用虛線的箭號線段來表示訊息回覆，讓它變得很清楚。 (我在 *Analysis Patterns*[Fowler, AP]一書中也是用虛線的訊息回覆，這讓自己覺得很有影響力。)我們還能夠將傳回的東西加以命名，就像 enoughStock : =check() 一樣。

▼ 155

「角色」這個術語的使用

在 UML 1.0 版中，***角色***(role) 這個術語主要是用來表示關聯的某個*方向* (direction) (請參見本書前幾刷)。UML 1.1 版把這樣的用法稱為關聯角色。此外，還有一個***合作角色***(collaboration role)，它代表類別的實例在合作情形中所扮演的角色。UML 1.1 版中提到較多的合作情形，而且看起來「合作角色」好像已經變成「角色」的主要用法了。

UML 從 1.2 版（包括 1.1 版）到 1.3 版（包括 1.5 版）的改變

使用案例

使用案例中有改變的地方是多了新的使用案例關係。UML 1.1 版中有兩個使用案例關係：<<uses>>和<<entends>>，兩者都是*一般化關係*（generalization）的造型。UML 1.3 版中則提供三種關係。

- <<include>>是*相依性*（dependencies）的一個造型。它表示某個使用案例的路徑被包含在另一個使用案例中。一般來說，它發生於幾個使用案例所共享的步驟。被包含的使用案例能夠將共通的行為給分離出來。在 ATM 的例子中，提款跟進行交易都會用到驗證顧客這個步驟。<<includes>>關係取代了原先的<<uses>>關係。

- 使用案例間的一般化關係表示某個使用案例是另一個使用案例的變種。例如，我們可能會有一個使用案例：提款（基本使用案例）和一個獨立的使用案例去處理錢不夠而被拒絕的情況。拒絕的部分可以由*特殊化*（specializing）的提款使用案例處理。（你也可以把它當作提款使用案例中的另一個情節來處理。）一個*特殊化使用案例*（specialized use cases）可能會改變*基本使用案例*（base use cases）裡面的任何部分。

- <<extend>>也是相依性的造型。它比一般化關係提供更多的控制。此時，基本使用案例中會先宣告一些*擴充點*（extension point）。*擴充使用案例*（extending use cases）只能去改變擴充點上的行為。所以假設你在網路上買東西，你可能會有一個買東西的使用案例，它有兩個擴充點：取得送貨資訊與付款資訊。擴充使用案例可針對常客，在這兩個擴充點上加以擴充，對這兩種資訊提供不同的資訊取得方式。

舊的使用案例關係跟新的使用案例關係間有些讓人感到迷惑的地方。許多人將 1.3 版的<<include>>當成舊的<<uses>>來用，所以對大部分的人來說，<<include>>取代了<<uses>>。而且大部分的人把 1.3 版中受控制的<<extend>>加上一般化關係當成 1.1 版的<<extends>>。所以你會認為 1.1 版的<<extends>>被拆成 1.3 版的<<extend>>跟一般化關係。

雖然這樣的解釋可涵蓋個人所看到的大部分用法，不過嚴格來說，這並不是正確的解釋。然而，大部分的人並不會嚴格遵循規格中的用法，所以我也不想在這裡多鑽牛角尖。

活動圖

UML 變成 1.2 版時，*活動圖*（activity diagram）的語意還是有很多不清楚的地方。所以，1.3 版花了很多力氣把語意弄得更清楚些。

就*條件式*（conditional）行為來說，我們可以將菱形用在*分支節點*（branche）跟*合併節點*（merge）的行為上。雖然不是所有的分支節點或合併節點都需要描述條件式行為，不過把菱形都秀出來的話，就可以形成一個共通風格，這樣一來，我們就可以用方括號把條件式行為給括起來了。

同步線（synchronization bar）現在分為***分岔節點***（fork）（將控制權分開）跟***會合節點***（join）（將控制權合在一起）兩種。現在會合節點不一定要有*任意條件*（arbitrary condition）在。此外，我們必須遵守配對規則，保證分岔節點跟會合節點會成對出現。基本上來說，每一個分岔節點都要有相對應的會合節點，由分岔節點開始的*執行緒*（thread）會在相對應的會合節點匯合在一起。我們可以用巢狀方式來使用分岔節點跟

會合節點。圖上有些分岔節點跟會合節點可省略不畫，這個情形是一個分岔節點直接連著另一個分岔節點或一個會合節點直接連著另一個會合節點。

會合節點在所有的*進入執行緒*（incoming thread）都完成時才會被觸發。然而，離開分岔節點的執行緒是可以有條件的。假如條件不成立的話，這個執行緒對會合節點來說就被視為是已完成的了。

多重觸發（multiple trigger）的特性已經不存在了。現在，活動可以是動態、*並行的*（concurrent）（在活動方格內顯示*）。這樣的活動可平行觸發多次。*離開轉換動作*（outgoing transition）要在所有被引發的動作都完成後才能發生。這樣子幾乎等於是多重觸發加上要符合的*同步條件*（synchronization condition）。

這些規則降低了活動圖的彈性，可是這麼做可確保活動圖真的是*狀態機*（state machine）的一個特例。活動圖跟狀態機的關係在 RTF 中有些爭議在。未來 UML 也許可以把活動圖當作完全不同的圖（譯註：這點真的如本書作者所言，UML 2 版中將活動圖跟狀態機圖分開了）。

UML 從 1.3 版到 1.4 版的改變

UML 1.4 版中最顯著的改變是加入了***造型輯***（profile），它允許我們將一整組的擴充情形合在一起，放入一致的集合當中。UML 文件中包含了造型輯的一些例子。有了造型輯，我們就可以用正式方式定出造型，讓模型元素從此有多種造型存在；在 UML 1.3 版中，每個模型元素都只能有一種造型。

UML 中也新增了***工作成果***（artifact）。工作成果是*元件*（component）的實體呈現結果，所以如果 Xerces 是一個元件，那麼在我的硬碟中的所有 Xerces.jar 檔案就都是 Xerces 元件的工作成果。

UML 1.3 版以前（包括 1.3 版），*超模型*（meta-model）中並沒有可處理 Java 中***套件可見性***（package visiability）的東西。現在，我們可以用「~」來表示它。

UML 1.4 版在互動圖中用棒狀箭頭來代表非同步，這是跟前幾版不相容的可怕改變。它讓一些人感到不滿，當然也包括本人。

UML 從 1.4 版到 1.5 版的改變

主要的改變是在 UML 中新加入*動作語意*（action semantic），它是讓 UML 具有程式語言能力的一個必要步驟。這讓大家不用等到完整的 UML 2 版出現就可以先將 UML 當成程式語言來用。

UML 從 1.x 版到 2.0 版的改變

UML 2 版是從以前到現在為止 UML 所做的最大改變。所有東西在這個修正版中都有所改變，而且也對本書造成重大改變。

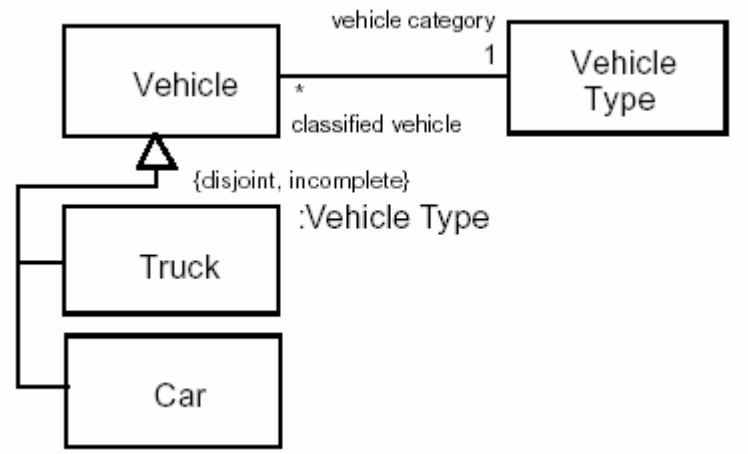
在 UML 2 版中，超模型底層有非常大的改變。雖然這些改變沒有影響到我們在本書中所談到的一些東西，不過對某些團體來說，卻是非常重要的。

UML 2 版中最明顯的改變是它引進了一些新的圖形種類。之前很多人就在使用*物件圖*（object diagram）與*套件圖*（package diagram），不過它們都不是官方、正式的圖形種類；不過，現在它們都是了。UML 2 版將*合作圖*（collaboration diagram）改名成*通訊圖*（communication diagram）。此外，UML 2 版中還引進了一些新的圖形種類，包括：*互動概圖*（interaction overview diagram）、*時序圖*（timing diagram）與*合成結構圖*（composite structure diagram）。

▼ 158

本書故意不提到一些新的改變，例如*狀態機擴充*（state machine extension）、互動圖中的*閘道*（gateway）與類別圖中的*強力型態*（power type）。

譯註 1：強力型態的實例是跟它有關聯之類別的子類別。舉例來說，假設 Truck 與 Car 類別是 Vehicle 類別的*子類別*（sublcass），而且 Vehicle Type 是跟 Vehicle 有關聯的強力型態，那麼 Truck 與 Car 類別都是 Person Type 的實例。



來源：UML 2.0 Superstructure Final Adopted Specification

譯註 2：狀態機擴充代表狀態機是*可一般化的*（generalizable）。在*特殊化狀態機*（specialised state machine）中，我們可以新增或重新定義*一般性狀態機*（general state machine）中的任何東西。

就本小節來說，我只會提到跟本書有關的改變。其中有些改變是跟本書前版中有討論到的東西有關，有些則是本書此版中新出現的東西。因為改變很多，所以下面根據本書章節整理這些改變。

類別圖：基本概念（第 3 章）

屬性跟單向關聯現在都代表*性質*（property）這個概念，它們只是不同的表示法而已。不連續的*多重性*（multiplicity）（例如[2, 4]）現在已經被刪掉了。*凍結不變*（frozen）

性質也被刪掉了。我新增了一份常見的相依性關鍵字清單，其中有很多是 UML 2 版中新增的。<<parameter>>與<<local>>關鍵字也被刪掉了。

循序圖（第 4 章）

循序圖中最大的改變是新增*互動框*（interaction frame）表示法，以處理反覆、條件式或其他的控制行為。現在循序圖已經可以非常完整地表達出*演算法*（algorithm）了，不過個人還是不相信這些圖形表示法會比程式碼更加清楚。用在訊息上舊有的反覆標記（*）與成立條件，在循序圖中已經被刪掉了。*生命線*（lifeline）上方所放的東西也不再是*實例*（instance）；我用*參與物件*（participant）來稱呼它們。UML 1 版中的合作圖也已經被改名成 UML 2 版中的通訊圖。

類別圖：高等概念（第 5 章）

造型現在有更嚴謹的定義。因此，我現在把角括號中的字稱為關鍵字，裡面只有某些才是造型。物件圖中的實例則變成*實例規格*（instance specification）。類別現在可以提供或需要介面。*多重類別化*（multiple classification）用*一般化關係集合*（generalization set）將一般化關係群組起來。元件不再用特有的符號去畫。*主動物件*（active object）也改用雙垂直線，不再用粗線去畫它的框。

▼ 159

狀態機圖（第 10 章）

UML 1 版中將短時間存活的動作跟長時間存活的活動區別開來。UML 2 版中則將動作與活動合稱為「活動」，至於長時間存活的活動特稱為*持續進行活動*（do-activity）。

活動圖（第 11 章）

UML 1 版中將活動圖視為狀態圖的一個特例。UML 2 版則打破它們之間的關係，因此 UML 2 版中的活動圖不用再遵守分岔節點與會合節點要成對的規則。也因為如此，我們最好用*（令）符流*（token flow）來了解活動圖，而不要用狀態轉換動作來了解它。UML 2 版中也因此出現了一大堆的新表示法，包括：*時間信號*（time signal）、*接收信號*（accept signal）、參數、*會合規格*（join specification）、*接腳*（pin）、*（活動）流轉型*（flow transformation）、代表子活動的耙狀符號、*擴張區*（expansion region）與*（活動）流結束*（flow final）等。

有一個簡單但有點可怕的改變是 UML 1 版中將多個進入活動的進入（活動）流視為隱含*合併節點*（merge）在，不過 UML 2 版中卻將此視為隱含會合節點在。因此，個人建議大家在活動圖中明確畫出合併節點或會合節點。

水道（swim line）現在已經可以有多個維度了，因此我們把它改稱為*分割*（partition）。

參考書目

[Ambler]

Scott Ambler, Agile Modeling, Wiley, 2002.

[Beck]

Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000.

李潛瑞(民 91)。「極致軟體製程」。臺北市 : 臺灣培生教育。

[Beck and Fowler]

Kent Beck and Martin Fowler, Planning Extreme Programming, Addison-Wesley, 2000.

王錦裕(民 91)。「規劃極致軟體製程」。臺北市 : 臺灣培生教育。

[Beck and Cunningham]

Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," Proceedings of OOPSLA 89, 24 (10): 1-6. http://c2.com/doc/oopsla89/paper.html

[Booch, OOAD]

Grady Booch, Object-Oriented Analysis and Design with Applications, Second Edition. Addison-Wesley, 1994.

[Booch, UML user]

Grady Booch, Jim Rumbaugh, and Ivar Jacobson, UML User Guide, Addison-Wesley, 1999.
張裕益(民 90)。「UML 使用手冊」。臺北縣汐止市 : 博碩文化。

[Coad, OOA]

Peter Coad and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, 1991.

[Coad, OOD]

Peter Coad and Edward Yourdon, Object-Oriented Design, Yourdon Press, 1991.

[Cockburn, agile]

Alistair Cockburn, Agile Software Development, Addison-Wesley, 2001.

[Cockburn, use cases]

Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2001.
趙光正(民 92)。「使用案例寫作實務 - 寫作指南、祕訣與範本」。臺北市 : 碁峰資訊。

[Constantine and Lockwood]

Larry Constantine and Lucy Lockwood, Software for Use, Addison-Wesley, 2000.

[Cook and Daniels]

Steve Cook and John Daniels, Designing Object Systems: Object-Oriented Modeling with Syntropy, Prentice-Hall, 1994.

[Core J2EE Patterns]

Deepak Alur, John Crupi, and Dan Malks, Core J2EE Patterns, Prentice-Hall, 2001.
鄧文彥、羅文昌(民 91)。「J2EE 核心樣式：實作與設計策略」。臺北市 : 臺灣培生教育。

[Cunningham]

Ward Cunningham, "EPISODES: A Pattern Language of Competitive Development." In Pattern Languages of Program Design 2, Vlissides, Coplien, and Kerth, Addison-Wesley, 1996, pp. 371-388.

[Douglass]

Bruce Powel Douglass, Real-Time UML, Addison-Wesley, 1999.

[Fowler, AP]

Martin Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.

[Fowler, new methodology]

Martin Fowler, "The New Methodology," http://martinfowler.com/articles/newMethodology.html

[Fowler and Foemmel]

Martin Fowler and Matthew Foemmel, "Continuous Integration," http://martinfowler.com/articles/continuousIntegration.html

[Fowler, P of EAA]

Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[Fowler, refactoring]

Martin Fowler, Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999.

侯捷、熊節(民 92)。「重構—改善既有程式的設計」。臺北市 : 碁峰資訊。

[Gang of Four]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

葉秉哲(民 91)。「物件導向設計模式」。臺北市 : 臺灣培生教育。

[Highsmith]

Jim Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002.

[Hohmann]

Luke Hohmann, Beyond Software Architecture, Addison-Wesley, 2003.

[Jacobson, OOSE]

Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[Jacobson, UP]

Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, The Object Advantage: Business Process Reengineering with Object Technology, Addison-Wesley, 1995.

[Kerth]

Norm Kerth, Project Retrospectives, Dorset House, 2001

[Kleppe et al.]

Anneke Kleppe, Jos Warmer, and Wim Bast, MDA Explained, Addison-Wesley, 2003.

[Kruchten]

Philippe Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley, 1999.
趙光正(民 91)。「Rational 統一流程入門」。臺北縣中和市 : 維科。

[Larman]

Craig Larman, Applying UML and Patterns, 2d ed., Prentice-Hall, 2001.
趙光正(民 91)。「UML 與樣式徹底研究」。臺北市 : 臺灣培生教育。

[Martin]

Robert Cecil Martin, The Principles, Patterns, and Practices of Agile Software Development, Prentice-Hall, 2003.

[McConnell]

Steve McConnell, Rapid Development: Taming Wild Software Schedules,
Microsoft Press, 1996.

[Mellor and Balcer]

Steve Mellor and Marc Balcer, Executable UML, Addison-Wesley, 2002.

B-power 工作室(民 92)。「Executable UML 模型驅動架構入門」。臺北市 : 臺灣培生教育。

譯註：此書翻譯品質其差無比，譯者勉強看完一遍之後，就把書丟掉了。

[Meyer]

Bertrand Meyer, Object-Oriented Software Construction. Prentice-Hall, 2000.

[Odell]

James Martin and James J. Odell, Object-Oriented Methods: A Foundation (UML Edition), Prentice Hall, 1998.

[Pont]

Michael Pont, Patterns for Time-Triggered Embedded Systems, Addison-Wesley, 2001.

[POSA1]

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.

[POSA2]

Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, Pattern-Oriented Software Archtecture Volume 2: Patterns for Concurrent and Networked Objects, Wiley, 2000.

[Rumbaugh, insights]

James Rumbaugh, OMT Insights, SIGS Books, 1996.

[Rumbaugh, OMT]

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, Object-Oriented Modeling and Design, Prentice-Hall, 1991.

[Rumbaugh, UML Reference]

James Rumbaugh, Ivar Jacobson, and Grady Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

[Shlaer and Mellor, data]

Sally Shlaer and Stephen J. Mellor, Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1989.

[Shlaer and Mellor, states]

Sally Shlaer and Stephen J. Mellor, Object Lifecycles: Modeling the World in States. Yourdon Press, 1991.

[Warmer and Kleppe]

Jos Warmer and Anneke Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998.

[Wirfs-Brock]

Rebecca Wirfs-Brock and Alan McKean, Object Design: Roles Responsibilities and Collaborations. Prentice-Hall, 2003.

英文索引

B

C

D

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U