活學活用 UML 與樣式第二版

作者 Craig Larman

譯者簡介 趙光正 政治大學資管系碩士斑畢業（kcchao@hotmail.com。）目前從事物件導向顧問工作，曾經以 BEA WebLogic 開發專案，也是 UML 精華第二版、Rational 統一流程第二版與 Java 經典範例第二版等書的譯者。

書背

「在這一版中，Larman 維持他一貫精準、細心的寫作風格。這是一本非常好的書，而且第二版更好。」 －*使用案例最佳實務－寫作指南、秘訣與範本*（Writing Effective Use Cases）與 *Surviving OO Projects* 兩本書的作者：Alistair Cockburn。

「有辦法把東西解釋清楚的人不多，這些人當中還會物件導向分析與設計的人更珍貴了，而 Craig Larman 身兼兩者。」 － Design Patterns 與 Pattern Hatching 兩本書的作者：John Vlissides。

活學活用 UML 與樣式

物件導向分析與設計與統一流程入門

第二版

全世界介紹物件導向分析與設計、反覆式開發方式與 UML 的書當中，銷售量最好的一本書，現在已經完全更新過了！

*活學活用 UML 與樣式*第二版將可幫助任何開發人員或學生精通物件導向分析與設計的核心開發原則與最佳實務經驗，讓大家不只是畫畫 UML 而已，而是真的能把它活用在軟體設計情境中。著名的物件技術與反覆式開發方法先驅 Craig Larman 在本書中用單一、有一致性的個案研究，具體呈現開發過程中的三個反覆。透過這三個反覆，逐步介紹物件導向分析與設計中的關鍵技能，並且點出最必要的開發活動、開發原則與樣式。本書內容涵蓋了：

- 需求與使用案例：發掘需求並記錄它
- 產生領域物件模型：理解領域中「有興趣的物件」、它們的屬性以及它們之間的關係
- 架構：爲了讓應用程式具有最大的彈性、強固性與可維護性，採用分層式架構設計系統
- 必要的物件設計技巧：專精一些關鍵技能，其中包括分配物件責任以及根據一些開發原則設計物件間的合作關係－例如資訊專家樣式、間接性樣式與預防變異樣式
- 設計樣式：用很流行、常用的樣式產生強固的物件與框架－例如策略樣式、工廠樣式、轉接器樣式、觀察者樣式、範本方法樣式與命令樣式
- 反覆式開發方式與「敏捷式統一流程」：用 UP（一種很風行的反覆式流程）中簡單、不可或缺的開發活動與最佳實務經驗組織建立模型與開發系統的流程

這個新版本是經過慎重考慮過的更新版，裡面有新的個案問題，也更新了樣式、使用案例、UP、架構分析等部份。*活學活用 UML 與樣式*第二版對如何思考物件、設計物件做了透徹而實際的介紹。

作者簡介 Craig Larman 是 Valtech 公司在開發流程與方法論方面的主管，

Valtech 是一家居領先地位的國際電子商務顧問團隊。在國際軟體社群中，作者是物件技術、樣式、 UML 、建模型與反覆式開發方式方面廣爲人知的專家。Larman 從 1980 年代開始就協助人們學習物件與反覆式開發方式，他曾親身教過數千位開發人員。此外，他也是 Java 2 Performance and Idiom Guilde 一書的協同作者，並且擁有加拿大不列顛哥倫比亞省 Vancouver 市 Simon Fraser 大學的計算機科學學士、碩士學位。

譯者簡介 趙光正 政治大學資管系碩士斑畢業（kcchao@hotmail.com。）目前從事物件導向顧問工作，曾經以 BEA WebLogic 開發專案，也是 UML 精華第二版、Rational 統一流程第二版與 Java 經典範例第二版等書的譯者。

出版社 培生出版社。

序

程式設計很有趣，不過開發有品質的軟體很難。在好的構想與需求之間或者遠景與務實的軟體產品之間，有許多比程式設計更重要的事。分析與設計（如何解決問題、要寫什麼程式）以及用容易溝通、實作、演化的方式捕捉設計是本書的整個核心，本書將教你這些東西。

統一模型語言（Unified Modeling Language，UML）已經變成畫軟體設計藍圖的語言中被全世界廣泛接受的一種語言。UML 是一種視覺式語言，本書用它傳達設計概念，並且只強調最常用到的 UML 元素，捨棄 UML 中少用的語言特性。

在其它行業中，大家都知道：用手工方式建造複雜的系統時，樣式是很重要的。（在軟體業中，）我們用軟體設計樣式描述設計片段、可再使用的設計概念，並且幫助開發人員學習其它專家的知識。每個樣式都會有一個名稱，裡面包含物件導向技術方面、抽象的啓發、設計規則與最佳實務經驗。任何理智的工程師都不希望用簡單的石版建房子，所以本書提供了許多馬上派得上用場的設計樣式給大家。

然而，在缺乏軟體工程開發流程的情境下，軟體設計讓人感到很無趣也神秘難懂。我很高興 Craig Larman 在本書第二版中，選擇並接受 UP 作爲軟體工程開發流程的情境，並且用相當簡單、低儀式性的方式應用 UP。當他用反覆式、風險驅動（risk-driven）、以架構爲核心（architecture-centric）的開發流程展現個案時，Craig 帶來的忠告就更真實了。透過這種方式，大家可以看到軟體開發過程中會發生的一些動態情況，也可以了解外部力量所扮演的角色。此外，設計開發活動也可以跟其它開發工作串聯在一起，而不再只是系統性轉換或創造性直覺那樣純粹的思考活動而已。Craig 跟我兩人都相信反覆式開發方式的好處，你將在本書完整看到它的好處。

從我看來，本書有最合適的材料。透過本書，你將跟全世界其它數以千計的開發人員一樣，追隨一位偉大、優秀的方法論者、物件導向導師，學習如何用系統化的開發方法進行物件導向分析與設計工作。Craig 用 UP 作爲描述開發方法的情境，並且逐步介紹比較複雜的設計樣式。因爲這樣，當你面對現實的設計挑戰時，會覺得本書非常好用，況且他用的還是最被廣泛接受的表示法。

我很榮幸有機會可以跟這本好書的作者一起共事。讀本書第一版時，個人感到非常受用，也很高興作者請我審查第二版草稿。我們見面討論了好幾次，也用電子郵件交換了許多意見。從 Craig 那裡，我學到許多東西，其中包括我在 UP 方面的工作，學到如何改良 UP、如何在不同組織中使用 UP。我確信你也可以從本書學到很多東西，縱然你很熟悉物件導向分析與設計也是如此。跟我一樣，你將有機會回歸到物件導向分析與設計的基本想法、更新你的記憶，或者從 Craig 的解釋與經驗中更深刻理解物件導向分析與設計。

在反覆式開發流程中，第二次反覆的結果會改進第一次反覆結果的許多地方。同樣地，我認爲本書寫作內容在第二版中也更加成熟。因此縱然你已經擁有本書第一版，第二版對你而言也將非常有用。

快快樂樂讀本書吧！

Philippe Kruchten
美商瑞理會員
美商瑞理軟體公司加拿大分公司
不列顛哥倫比亞省 Vancouver 市

前言

感謝你看本書！這是物件導向分析與設計方面非常務實的一本入門書，裡面同時也提到反覆式開發方式的許多方面。很高興本書第一版被翻譯成很多國語言，成爲全世界都很暢銷的物件導向分析與設計入門書。第二版不只換掉第一版部分內容而已，而是根據第一版的內容重新翻修過。在此，本人由衷感謝第一版的讀者。

透過本書，你將獲得下面幾項好處。

設計強固、易維護的物件系統

第一點，在軟體開發流程中用物件技術已經越來越普遍了，專精物件導向分析與設計是建構強固、易維護物件系統的關鍵之一。

遵從一份學習地圖，完成需求、分析、設計與寫程式的開發工

第二點，如果你剛接觸物件導向分析與設計，我們可以理解你可能會質疑：如何做完物件導向分析與設計這個複雜的工作。本書介紹的 UP 就是定義良好的學習地圖，讓你按部就班從需求變成程式碼。

用 UML 展現分析模型與設計模型

第三點，統一模型語言（Unified Modeling Language，UML）已經變成建模型的標準表示法，精通它對你會非常有幫助。本書用 UML 表示法教你物件導向分析與設計的技能。

用「gang-of-four」與 GRASP 設計樣式改善設計技巧

第四點，用設計樣式作爲溝通字彙，告訴你物件導向設計專家建構系統時會用到的「最佳實務經驗」：實作樣式與解決方案。本書也將教你如何活用設計樣式，其中包含很流行的「gang-of-four」樣式與 GRASP 樣式，後者是物件設計過程中分派（物件）責任的基本原則。學習並活用樣式將加快你專精分析與設計的腳步。

細緻的呈現方式，會讓學習效率更佳

第五點，這些年來，本人曾經訓練並指導過數千人物件導向分析與設計的藝術，以這些經驗爲基礎，希望本書的結構與重點可以提供大家一個細緻、被證實過有效的學習物件導向分析與設計方式，以求最佳的閱讀與學習效果。

透過真實的範例學習（從設計）到程式碼

第六點，本書詳細介紹單一個案研究，真實展現整個物件導向分析與設計過程，還深入探討問題中煩雜的細節，這是一個很真實的範例。

第七點，本書告訴你如何從物件設計工作成果變成 Java 程式碼。

設計分層架構

第八點，本書解釋如何設計出分層架構，並且把圖形使用者介面層跟領域層與技術服務層連結起來。

設計框架

第九點，本書教你如何設計物件導向框架，並且活用它以產生資料庫永續儲存框架。

本書目標

整體而言，本書目標爲：

協助學生與開發人員用可解釋的開發原則與啓發設計法進行物件設計。

研讀並活用本書教你的資訊與技術後，你將能夠更熟練地用（企業）流程與概念

了解問題（譯註：物件導向分析），並且用物件設計出紮實的解決方案（譯註：OOD）。

本書所預設的讀者

本書是物件導向分析與設計的一本入門書，裡面也提到一些需求分析，並且把 UP 當作反覆式開發流程的範例。這不是一本進階的教科書，我們假設讀者是：

- 對物件導向程式設計語言有經驗的開發人員與學生，不過對物件導向分析與設計很陌生或相當陌生。
- 計算機科學或軟體工程課程中研讀物件技術的學生。
- 對物件導向分析與設計有些了解，並且想學習 UML 表示法、活用樣式的人，或者想提升自己分析與設計技能的人。

讀者應先具備的相關知識

如果讀者想用本書增加自己的能力，應該先具備一些必要的相關知識，包括：

- 懂某種物件導向程式設計語言，並且具備寫這類程式的經驗，例如 Java、C#、C++ 或 Smalltalk。
- 懂基本的物件技術概念，例如類別、實例、介面、多型、封裝、介面定義（interfaces）與繼承。

本書並沒有教你基本的物件技術概念。

Java 範例

由於有很多人熟悉 Java ，所以書中的程式碼範例大部分是用 Java 程式語言寫的，也會討論到一些 Java 實作方面的東西。然而，本書所教的概念能夠應用到大部分（縱然不是全部）的物件導向程式設計語言。

本書編排方式

本書整體的組織策略是像軟體開發專案一樣，在「初始階段」（UP 中的一個術語）之後，用三次反覆依序介紹分析與設計的相關主題（請參見圖 P.1。）

1. 在初始階段章節介紹需求分析的基本概念。
2. 反覆 1 介紹基礎的物件導向分析與設計以及如何分派物件責任。
3. 反覆 2 把焦點放在物件設計，特別介紹一些常用的「設計樣式」。
4. 反覆 3 介紹各種主題，例如架構分析與框架設計。

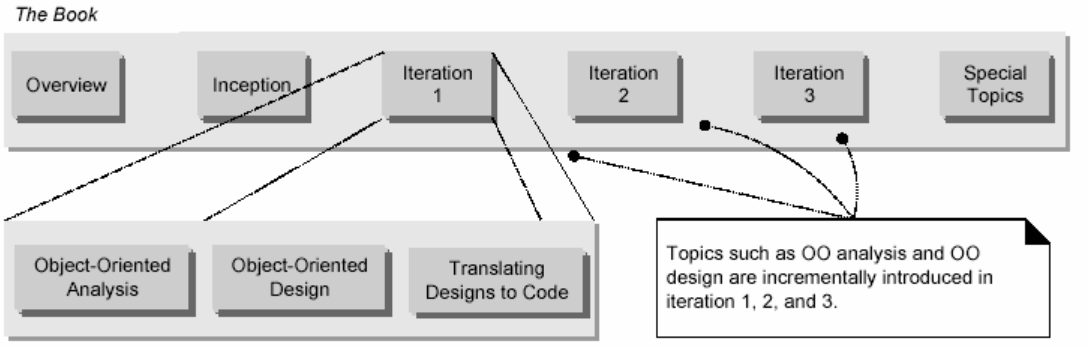


圖 P.1 本書的編排方式跟開發專案一樣

網路相關資源

- 請參閱 www.craiglarman.com 中跟物件技術、樣式與開發流程相關的文章。
- www.phptr.com/larman 中有一些教師用資源。

第二版強化第一版的地方

雖然第二版跟第一版的核心內容相同，不過第二版強化了許多地方，包括：

- 重寫使用案例，以遵循【Cockburn01】一書中常見的使用案例寫作方式。
- 用著名的 UP 作爲示範用的反覆式開發流程，透過這樣的開發流程介紹物件導向分析與設計技能。因此，所有工作成果都根據 UP 中的術語重新命名，例如設計模型。
- 個案研究中的新需求帶領我們完成反覆 3。
- 重寫設計樣式的處理方式。
- 介紹架構分析。
- 介紹預防變異樣式，並且把它視爲 GRASP 樣式其中之一。
- 書中循序圖與合作圖的使用頻率，保持平衡的 50／50 比例。
- 使用最新的 UML 表示法。
- 討論用白板或 UML CASE 工具畫圖時，會面臨的一些實際議題。

致謝

首先，非常感謝本人在 Valtech 公司的友人與同事，這家公司是由一群世界級的物件開發人員與反覆式開發方式專家組成，他們對本書有許多貢獻、支持本書、審查本書內容，其中包括 Chris Tarr、Michel Ezran、Tim Snyder、Curtis Hite、Celso Gonzalez、Pacal Roques、Ken DeLong、Brett Schuchert、Ashley Johnson、Chris Jones、Thomas Liou、Darryl Gebert、Frank Rodorigo、Jean-Yves Hardy 以及許多我不知道名字的人。

感謝 Philippe Kruchten 替本書寫序並審查本書，並且在很多地方幫助本人完成本書。
感謝 John Vlissides 與 Cris Kobryn 爲本書所下的引言，感激不盡。
感謝 Chelsea Systems 公司與 John Gray 提供本書一些 POS 系統需求，這些需求導源於他們用 Java 技術開發完成的 ChelseaStore POS 系統。
感謝 TogetherSoft 公司 Pete Coad 與 Dave Astels 的支援。
也非常感謝本書其它審查人員，其中包括 Steve Adolph、Bruce Anderson、Len Bass、Gray K. Evans、Al Goerner、Luke HohmannEric Lefebvre、David Nunn 與 Robert J. White。
感謝 Prentice-Hall 公司的 Paul Becker，讓本書第一版得以問世，也感謝 Paul Petralia 與 Patti Guerrieri 協助第二版出版。
最後，特別感謝 Graham Glass 的啓迪。

關於作者

Craig Larman 服務於 Valtech 公司，擔任開發流程方面的主管，這是一家國際性的顧問公司，在歐洲、亞洲與北美洲都有分公司，專注於電子商務系統開發、物件技術與 UP 的反覆式開發方式。
從 1980 年代中期開始， Craig 開始協助數千位開發人員應用物件導向程式設計、物件導向分析與物件導向設計，並且協助一些組織進行反覆式開發方式的實務工作。
做不成流浪的街頭音樂家之後，他在 1970 年代用 APL、PL/I 與 CICS 開發系統。在 1980 年代初期，不再幻想當音樂家的他，開始對人工智慧（有一小部分是他自己專屬的）、自然語言處理與知識呈現方式感到興趣，並且用 Lisp 機、Lisp、Prolog 與 Smalltalk 開發知識系統。業餘時間則繼續他的 Changing Requirements 樂隊，擔任不怎麼樣的首席吉他手（這個樂隊本來叫做 Requirements ，不過因爲有成員離開，所以…）。
他擁有加拿大不列顛哥倫比亞省 Vancouver 市 Simon Fraser 大學的計算機科學學士、碩士學位。
你可以透過 clarman@acm.org 與 www.craiglarman.com 跟 Craig 聯繫。

本書編排慣例

這是文中第一次出現的**新術語**。這是文中的*類別*名稱或*方法*名稱。這是某位作者的參考文獻【Bob67】。我們用跟程式語言無關的可見域解析運算子「--」表現類別跟它的方法：*ClassName--methodName*。

產品說明

本書原稿是用 Adobe FrameMaker 編排的。書中所有圖則是用 Microsoft Visio 畫的。文章字體爲 New Century Schoolbook。最後用來印刷的輸出影像是用 AGFA 的 PostScript 驅動程式、Adobe 的 Acrobat Distiller 產生的 PDF 檔。

第一部分簡介

第一章物件導向分析與設計

把寫程式的重點放到樣式上，對寫程式的方式將會有深遠影響。
－Ward Cunningham 與 Ralph Johnson

本章目標

- 比較並區分分析與設計之間的差異。
- 定義何謂物件導向分析與設計。
- 用簡單的例子闡述這些概念。

第一節在物件導向分析與設計中活用 UML 與樣式

系統有好的物件設計這件事有什麼意義呢？本書是教開發人員與學生學習物件導向分析與設計（OOA/D）核心技能的一本工具書。如果你想用物件相關技術與程式語言（例如 Java 、 C++ 、 Smalltalk 與 C# 等等）產生設計良好、強固與可維護的軟體，那麼就必須擁有這些技能。

就物件技術方面來說，俗話說的好：「有槌頭並不代表你就是建築師。」了解物件導向程式語言（例如 Java ）只是必要的，不過這個基礎不足以讓你產生物件系統，因爲知道如何「用物件思考事情」是更重要的事。

這是一本入門書

這是一本物件導向分析與設計的入門書，裡面還介紹了統一模型語言（unified modeling language， UML ）、樣式與 UP。我們不想把本書變成一本進階教科書，因此把焦點放在如何專精基礎概念上，例如如何指派責任到物件身上、常用的 UML 表示法，以及常見的設計樣式等等。而後面的章節則慢慢進階成中級主題，例如框架設計。

應用 UML

本書內容不只是 UML 而已。 UML 是一種標準的圖形表示法。學習表示法對大家很有幫助，不過大家還是要去學習更重要的物件導向概念，特別是如何用物件思考－如何設計物件導向系統。 UML 既不是物件導向分析與設計也不是方法論，它只是表示法而已。努力學習畫出語法正確的 UML 圖或學習使用 UML CASE 工具並沒有多大效果，因爲這樣無法產生好的設計結果出來，或者讓你評

估或改善現有的設計結果。後者屬於更難、更有價值的技能。因此，本書定位爲物件設計的入門書。

然而，進行物件導向分析與設計或畫出「軟體藍圖」時需要用某種語言才能辦到，我們需要用這種語言思考，並且用它跟其他開發人員溝通。因此，本書裡面探討如何把 UML 應用在物件導向分析與設計上，並且在本書介紹常用的 UML 表示法。不過，重點會放在幫大家學習建構物件系統的美學與工程上，而不是教大家這些表示法。

應用樣式與指派責任

我們應該如何分配**責任**到物件的類別上呢？物件之間應該如何互動？哪些類別應該做什麼事？這些都是設計系統時最常發生的關鍵問題。針對某些設計問題，我們可以把大家已經知道、試驗過可行的解決方案用最佳實務原則、啓發與樣式（經過命名的問題－解決方案配套，裡面會編纂示範性的設計原則）表達出來。本書教大家如何應用樣式，讓大家能夠更快速學習、熟練這些基本的物件設計樣式。

單一個案研究

我們用單一個案研究貫穿整本書以介紹物件導向分析與設計，從分析慢慢深入到設計，裡面不但考量一些讓人害怕的細節，也提到並解決一些真實的設計問題。

使用案例與需求分析

物件導向分析與設計（以及所有軟體設計）跟**需求分析**（requirement analysis）這個事先進行、必要的開發活動之間有密切的關係，開發活動中包含寫使用案例這一部分。雖然使用案例並不是物件導向專用的，個案研究一開始還是先介紹這些主題讓大家了解。

反覆式開發流程的範本－UP

從需求到實作，我們需要進行許多開發活動，開發人員或團隊應該如何進行這些開發活動呢？我們需要在某個開發流程的情境中才能討論需求分析與物件導向分析與設計。在本書中，我們採用廣爲人知的**統一流程**（unified process）作爲**反覆式開發流程**（iterative development process）的範本，透過這個開發流程介紹相關主題。雖然我們用特定的開發流程介紹這些主題，不過本書所涵蓋的分析與設計主題，對許多開發方式來說都是一樣的，而且用 UP 學這些主題並不會妨礙大家把這些主題應用到其它方法論上。

整體而言，本書可以幫助學生或開發人員：

- 活用設計原則與樣式以產生比較好的物件設計。
- 以 UP 爲範本，進行分析與設計一般性的開發活動。
- 畫出 UML 表示法中常用的圖。

書中用單一個案研究介紹這些議題。

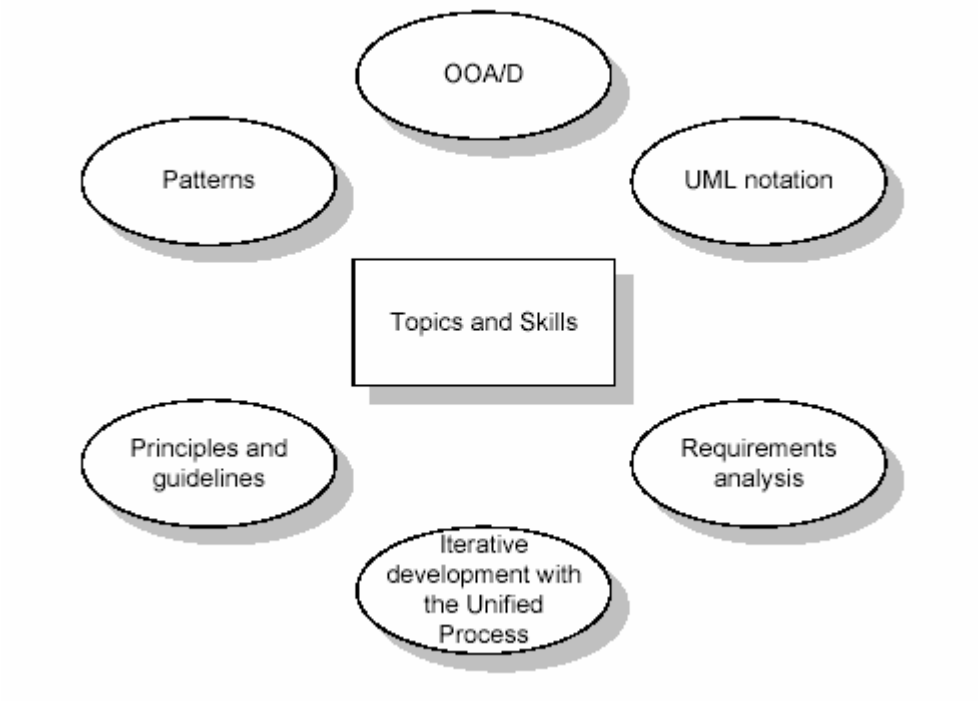


圖 1.1 本書所涵蓋的主題與技能

其它技能也是很重要的

除了需求分析、物件導向分析與設計與物件導向程式設計之外，建構軟體時還需要有大量的技能與步驟。舉例來說，可用性工程（usability engineering）與使用者介面設計對成功的軟體來說都是很重要的；當然資料庫設計也是很重要的。然而，這本入門書把焦點放在物件導向分析與設計上，不敢奢望涵蓋所有軟體開發的相關主題。這只是整個軟體開發大範圍中的一小塊而已。

第二節分配（物件）責任

有許多開發活動與工作成果都可以被拿來介紹物件導向分析與設計，此外也有許多設計原則與設計指引是跟物件導向分析與設計有關的。假設我們只能從本書所討論的主題中挑出最實用的技能－一個像「沙漠綠洲」一樣的技能，我們應該挑哪一個呢？

熟練指派責任到軟體元件上是物件導向分析與設計中很重要、很基礎的一種能力。

爲什麼呢？因爲不論你是畫 UML 圖或寫程式，都需要進行這項開發活動，而且它對軟體元件的強固性、可維護性與再使用性有很大的影響。

當然，物件導向分析與設計中還需要其它必要技能，不過我們在簡介中強調它是因爲：它是成爲物件導向分析與設計專家的一項關鍵技能，非常重要。真正專案

如果採用「急著產生程式碼（rush to code）」的開發流程，那麼開發人員可能沒有機會進行分析或設計開發活動。不過就算如此，分配責任這項開發活動也是不可避免的。

因此，本書所提到的設計步驟會把焦點放在指派責任這個原則上。

本書會介紹並應用九種物件設計與指派責任的基本原則。這些原則都整理在所謂的 GRASP 樣式中，以方便大家學習。

第三節何謂分析與設計？

分析（analysis）把焦點放在*調查*問題與需求上，而不是提供解決方案。舉例來說，如果我們需要電腦化、新的圖書館資訊系統，應該怎麼使用這個系統呢？「分析」是涵義很廣的一個詞，我們最好把它重新定義成*需求分析*（requirement analysis）（需求的調查工作）或*物件分析*（object analysis）（領域模型中物件的調查。）

設計（design）把焦點放在滿足需求的*概念性解決方案*（conceptural solution）而不是實作上。例如資料庫綱目與軟體物件說明。當然，我們最後還是會實作設計。跟分析一樣，設計最好也可以重新定義成物件設計或資料庫設計。

分析與設計可以分別用一句話來表達：*做對的事（分析）*以及*把事做對（設計。）*

第四節何謂物件導向分析與設計？

做**物件導向分析**（object-oriented analysis）時，我們會把焦點放在：找出並描述問題領域的物件－或概念－上。舉例來說，在圖書館資訊系統中，相關概念包括 *Book*、*Library* 與 *Patron*。

做物件導向設計（object-oriented design）時，我們會把焦點放在定義軟體物件，並且還要知道它們之間是如何合作以滿足需求的。舉例來說，在圖書館系統中，*Book* 這個軟體物件可能會有 *title* 屬性、 *getChapter* 方法（請參見圖 1.2。）最後，在實作或進行物件導向程式設計時，我們會實作設計物件，例如用 Java 實作 *Book* 類別。

圖 1.2 物件導向把焦點放在：用物件呈現概念

第五節簡單範例（擲骰子遊戲）

在深入需求分析與物件導向分析與設計的細節之前，本節先利用一個簡單的範例－「擲骰子遊戲」（玩家每次會擲兩粒骰子），針對一些關鍵步驟與圖，介紹需求

分析與物件導向分析與設計的概觀。在這個遊戲中，如果兩個骰子的總合等於七就是玩家贏，否則玩家輸。

譯註：本書的編排方式也符合本節介紹範例的邏輯，第 9 章先介紹使用案例模型、第 10 、 11 、 12 章介紹領域模型、第 15 章介紹互動圖、第 19 章介紹設計類別圖。

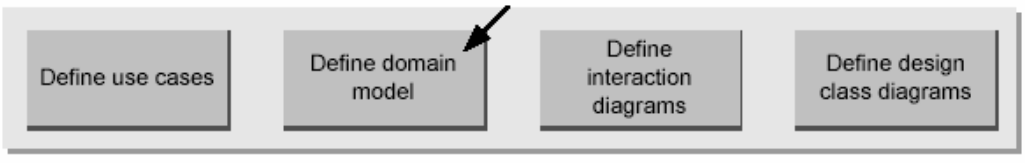
定義使用案例

需求分析中可能會描述相關領域流程，我們把這些描述寫成使用案例。使用案例並不是物件導向專用的工作成果－我們只是把敘事情節寫出來而已。然而，使用案例是需求分析中很普遍、常見的工具，也是 UP 中很重要的一部份。舉例來說，下面是*玩擲骰子遊戲*使用案例的簡式版本。

> 玩擲骰子遊戲：玩家拿起骰子並滾動它們。如果朝上的骰子面點數總合等於七，玩家贏，否則玩家輸。

定義領域模型

物件導向分析就是把物件類別化（classification）以產生領域描述（譯註：一般化【generalization】說明父類別與子類別之間的關係，類別化【classification】則說明類別與實例之間的關係。）我們在分解領域時會找出領域中值得注意的概念、屬性與關聯，然後把它們表達在領域模型中，產生表達領域概念或物件的圖（譯註：本書的編排方式也符合這樣的邏輯，所以第 10 章先介紹領域模型中的概念、第 11 章介紹領域模型中的關聯、第 12 章介紹領域模型中的屬性。）



舉例來說，圖 1.3 中所展現的就是領域模型的一部份。

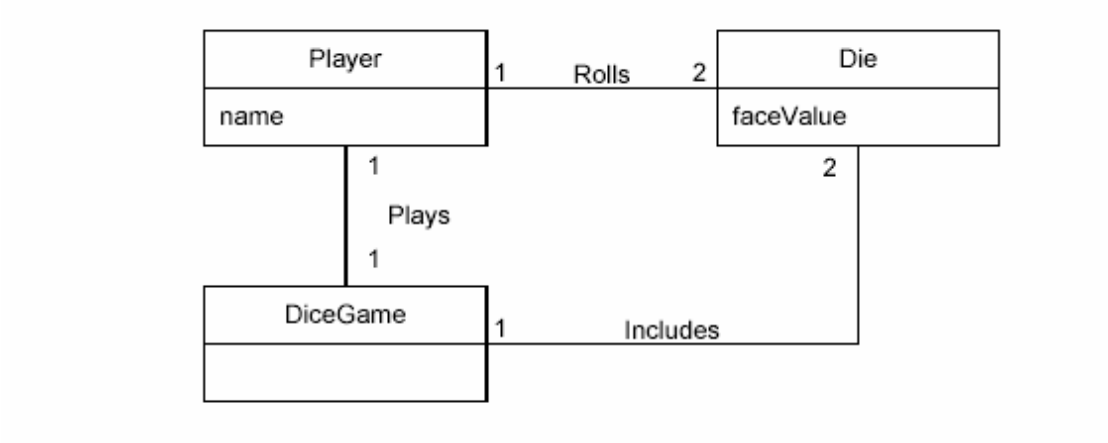


圖 1.3 骰子遊戲中領域模型的一部份

模型中展現一些值得注意的概念 *Player*、*Die* 與 *DiceGame* ，還有相關的關聯與屬性。

請注意領域模型並不是用來描述軟體物件的，我們用它呈現真實世界領域中的一些概念。

定義互動圖

物件導向設計中需要定義軟體物件與它們之間的合作關係。展現合作關係最常用的表示法是互動圖。互動圖中會展現軟體物件之間的訊息流，也就是呼叫哪些方法。

舉例來說，假設我們要實作擲骰子遊戲。圖 1.4 中的互動圖展現玩遊戲的必要步驟，遊戲中會傳訊息給 *DiceGame* 與 *Die* 類別的實例。

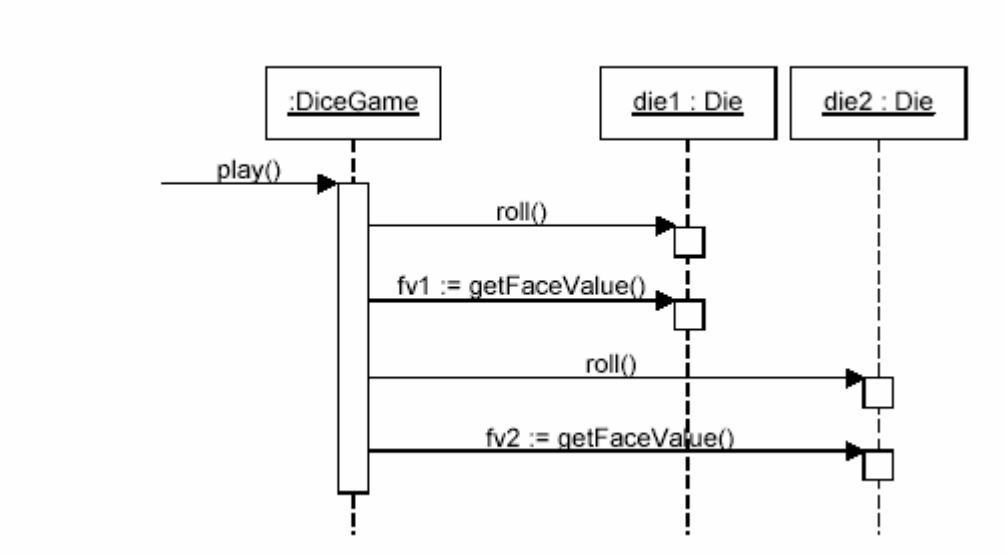
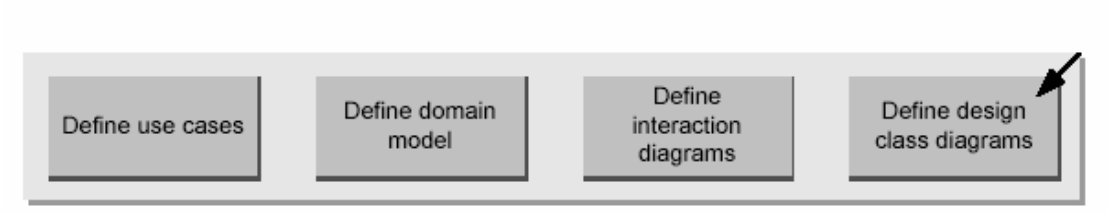


圖 1.4 用*互動圖*展示軟體物件之間的訊息

請注意，雖然在真實世界中是由 *Player* 滾動骰子，然而在軟體設計中，則是由 *DiceGame* 物件負責「滾動」骰子（換言之，傳訊息給 *Die* 物件。）我們通常從真實世界領域中找到軟體物件設計與程式的靈感，不過後者不是真實世界領域的直接對應模型或模擬。

定義設計類別圖

除了用*互動圖*展示物件合作關係的*動態*觀點之外，另外用**設計類別圖**（design class diagram）展示類別定義的*靜態*觀點，對開發工作來說也是很有幫助的。設計類別圖中會展示類別的屬性與方法（譯註：領域模型與設計類別圖之間的差異，除了設計類別圖不是領域模型的直接對應之外，領域模型中的概念只會展示屬性，而設計類別圖中的類別會展示屬性與方法。）



舉例來說，在骰子遊戲中，我們觀察*互動圖*後可以畫出設計類別圖的一部分，如圖 1.5 所示。因爲在*互動圖*中我們會傳 *play* 訊息給 *DiceGame* 物件，所以 *DiceGame* 類別需要有一個 *play* 方法，而 Die 類別中則需要 *roll* 與 *getFaceValue* 方法。

跟領域模型不同，這個圖並非展現真實世界中的概念，裡面所展現的是軟體類別。

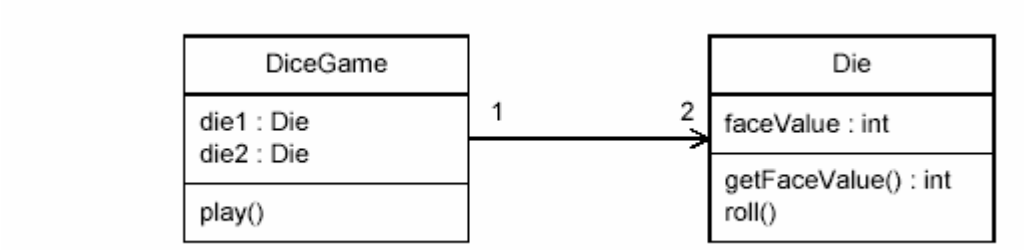


圖 1.5 設計類別圖的一部分

總結

骰子遊戲是很簡單的問題，分析與設計中的重點只有幾個步驟與工作成果。爲了讓本章簡單一點，我們並沒有解釋這裡出現的所有 UML 表示法。在未來的章節中，我們會更進一步探討分析與設計以及相關工作成果的細節。

第六節 UML

引述：

> 統一模型語言（Unified Modeling Language，UML）是用來詳細說明、呈現、建構、記錄軟體系統工作成果的一種語言，它也可以用來建立企業模型以及其它非軟體系統【OMG01】。

現在 UML 已經成爲建立物件導向模型方面實務上、公認的標準畫圖表示法。UML 緣起於 1994 年， Grady Booch 與 Jim Rumbaugh 結合他們兩位受歡迎的方法論－ Booch 與 OMT （Object Modeling Technique）方法論。稍後， Ivar Jacobson（ Objectory 方法論的創始人)加入他們，這三個人被稱爲*三人幫*(three amigos)。此外，還有許多人對 UML 也有很多貢獻，其中最著名的是 Cris Kobryn ，他是之後 UML 修訂版的領導者。

OMG （Object Management Group，一個工業標準的制定團體）在 1997 年正式採納 UML 爲標準，並且不斷致力於改良 OMG 的 UML 新版本。

本書並無法涵蓋整個 UML ，裡面包含龐大的表示法（有人認爲太大了【註】。）書中把焦點放在常用的圖、圖中最常用的畫圖特性，以及 UML 未來版本中可能不變的核心表示法。

【註】: 在 UML 2.0 版所花的功夫中，有一部分就是想簡化或減少表示法。本書只介紹常用、未來簡化版可能存在的 UML 表示法。

爲何本書中看不到介紹 UML 的一些章節？

本書不是一本 UML 表示法的書，而是探討如何在軟體開發過程的整個觀點中，應用 UML 、樣式與反覆式開發流程的一本書。 UML 主要用在物件導向分析與設計，而物件導向分析與設計之前要先做需求分析。因此，前面的章節會先介紹重要的使用案例與需求分析相關主題，稍後的章節中再介紹更多的物件導向分析與 UML 細節。

第七節進階讀物

UML 精華（*UML Distilled*）是可讀性很高、很流行的一本書，裡面彙總了 UML 中必要的表示法，作者爲 Martin Fowler 。

Rational 統一流程入門（*Rational Unified Process — An Introduction*）是介紹 UP（以及調整過的 Rational 統一流程）的一本暢銷書，內容很簡潔，作者爲 Philippe Kruchten 。

如果想知道 UML 1.3 版表示法的詳細內容，The Unified Modeling Language Reference Manual 與 *UML 使用手冊*（The Unified Modeling Language User Guide）這兩本書是很有價值的，作者爲 Booch 、 Jacobson 與 Rumbaugh 。請注意這些教科書並不適合拿來學習如何建立物件模型或做物件導向分析與設計－它們是 UML 表示法的參考手冊。

如果想知道 UML 最新版的現況，可以看在 www.omg.org 上的網路版 *OMG Unified Modeling Lauguage Specification* 。至於 UML 的修正工作以及即將發行版本，也可以在 www.celigent.com/uml 上找到。

軟體樣式方面有許多書，其中最根本、經典的一本書是*設計模式*（Design Patterns），作者爲 Gamma 、 Helm 、 Johnson 與 Vlissides 。對於想研讀物件設計的人來說，這是必讀的一本書。然而，這本書不是一本入門的教科書，最好先具有物件設計與程式設計基礎之後再去讀它。

譯註：*使用案例最佳實務－寫作指南、秘訣與範本*（Writing Effective Use Cases）則是教導使用案例寫作技巧的一本好書，作者爲 Alistair Cockburn 。

第二章反覆式開發方式與 UP

開發人員比任何開發流程都重要。
不過，好的開發人員配合好的開發流程之後比沒有開發流程的優秀開發人員表現更好。
— Grady Booch

本章目標

- 說明接下來幾章的內容與這些章節的安排動機。
- 定義反覆式、可調整的開發流程。
- 定義 UP 中的基本概念。

簡介

反覆式開發方式是一種很有技巧的軟體開發方式，而且它是本書中說明如何做物件導向分析與設計的核心觀念。UP 是用物件導向分析與設計、反覆式開發流程開發專案的一個範例，它的觀念也是本書的編排方式（譯註：請讀者參閱目錄就可以發現，本書的第二部分是初始階段，第四、五、六部分則是詳述階段中的三次反覆。）因此，讀完本章之後，你將了解本書結構的核心概念以及 UP 對本書編排方式的影響。

我們在本章先簡單說明一些觀念想法，至於 UP 與反覆式開發流程實務經驗的進階討論，請參閱第 37 章。

非正式來說，**軟體開發流程**（software development process）中會描述建構軟體、配置軟體以及有可能發生的軟體維護方式。對建構物件導向系統的人來說，**統一流程**（unified process，UP）【JBR99】已經是很流行的一種軟體開發流程。另一方面， **Rational 統一流程**（Rational Unified Process）或者 **RUP** 【Kruchten00】也已經被廣泛採用，它是更詳細的 UP 修正版。

UP 中融合了許多常見、被大家接受的最佳實務經驗，例如反覆式生命週期、風險驅動開發方式等等。它把這些經驗變成有良好文件、一致的說明。因此，本書用 UP 作爲開發流程的範例，並且透過這樣的流程介紹物件導向分析與設計。

本書一開始先介紹 UP，主要有兩個理由：

1. UP 是反覆式開發流程。反覆式開發方式是很有價值的實務經驗，它影響本書介紹物件導向分析與設計的方式，而且本書的寫作過程也真的實踐了它。
2. UP 中的實務經驗提供一個示範性結構，讓我們知道如何進行或學習物件導向分析與設計。

本章只簡單介紹 UP，沒有涵蓋整個 UP。這裡的焦點放在常見的概念與工作成果，以簡單介紹物件導向分析與設計與需求分析。

如果我不重視 UP 呢？

因爲我們需要以某個開發流程爲情境介紹需求分析與物件導向分析與設計，而且 UP（或者 RUP 修訂版）也已經被廣泛採用，所以本書以 UP 作爲開發流程的範例，用它探討需求分析與物件導向分析與設計。此外，UP 裡面也告訴我們常見的開發活動與最佳實務經驗。然而，本書的核心概念－例如使用案例與設計樣式－則跟任何特定的開發流程無關，我們可以把它們應用在許多開發流程上。

第一節UP 中最重要的概念：反覆式開發方式

UP 中提倡幾個最佳實務經驗，其中最重要的一個實務經驗是：反覆式開發方式。如果採用這種開發方式的話，我們會把發開工作組織成一系列時間短暫、固定長度（例如四個禮拜）的迷你專案，稱之爲**反覆**（iteration）。每個反覆的結果都是測試、整合過、可執行的系統。反覆中都包含自己的需求分析、設計、實作與測試開發活動。

反覆式生命週期是以多個反覆中逐漸成長、修正的系統爲基礎，用循環式的回饋與調整爲驅動力，慢慢變成穩定系統。因爲系統會隨著時間、反覆成長，所以這種開發方式也稱爲反覆式、漸增式的開發方式（請參見圖 2.1。）

早期的反覆開發流程概念被稱爲螺旋式開發方式（spiral development）或演化式開發方式（evolutionary development）【Boehm88、Gilb88】。

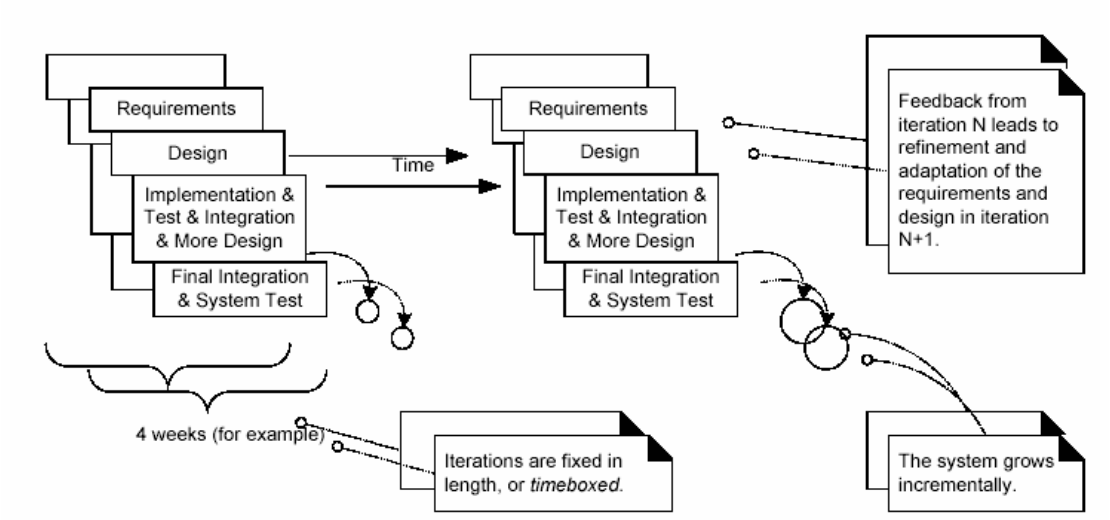


圖 2.1 反覆式、漸增式的開發方式

範例

因爲這裡只是一個範例（而不是建議採用的方法），所以星期一可能會先把時間主要花在分配與澄清這個反覆的工作與需求上，同時由一位同事負責把最後一個反覆的程式碼用反向工程變成 UML 圖（利用 CASE 工具），然後印出或秀出

值得大家注意的圖。星期二，我們用白板進行雙人式設計工作，畫出粗略的 UML 圖，並且用數位相機記錄下來，然後寫一些虛擬碼與設計註解。在剩餘的八天中，我們把時間花在實作、（單元、驗收、可用性）測試、未來的設計工作、整合、每天產生的建構版本、系統測試與加強部分系統穩定性上。其它的開發活動則包括展示系統給關係人看、評估系統，以及未來反覆的規劃工作。

請注意在這個範例中，我們既不會急著寫程式碼，也不會用很長的設計步驟，嘗試在寫程式之前，完成所有設計細節。範例中只會用視覺化方式快速畫出粗略的 UML 圖，完成「一些」屬於事先思考活動的設計結果。如果由兩位開發人員同時進行設計工作的話，或許會花半天或一天的時間。

每個反覆的結果是可執行但不完整的系統，這個系統並無法變成交付客戶的產品。有可能完成幾個反覆之後（例如 10 到 15 個反覆），這個系統還是無法做產品配置。

反覆的輸出結果並不是實驗性或會丟掉的雛型，而且反覆式開發方式也不等於雛型式開發方式。相反地，反覆的輸出結果是達到產品水準、最終系統中的部分系統。

一般來說，雖然每個反覆中都會加入新的需求，並且逐漸擴充成完整系統，不過有些反覆可能把焦點放在改善部分系統效能上，而不是在反覆中擴充新的系統特性。

擁抱改變：回饋與調整

有一本討論反覆式開發方式方面的書，其子標題是*擁抱改變*（embrace change）【Beck00】。這句話讓我們想起反覆式開發方式的一個關鍵屬性：不要嘗試寫出完整、正確、有詳細說明、凍結不變、「簽過名」的固定需求，並且在實作之前就完成整個設計工作，這樣做等於挑戰軟體開發過程中必然發生的變動。另一方面，反覆式開發方式的態度則是擁抱改變，並且在有不可避免、真的必要的驅動力時調整需求（譯註： UP 或 RUP 中雖然強調反覆式開發方式，不過卻沒有針對設計變動提出建議，程式重整【refactoring】概念可以填補這方面的不足。另一方面， XP 中強調的「先寫測試程式【test-first】」概念，可以讓我們及早釐清需求中含糊不清之處。）

這樣的說法並不是認爲反覆式開發方式與 UP 鼓勵不控制、回應式、「巴結特性」的開發流程。在接下來的章節中，我們會探討 UP 如何平衡兩種不同的需要：一方面，系統需要一致、穩定的需求；另一方面，系統需要接受關係人因爲越來越了解自己想要什麼而變動需求的事實。

我們在每個反覆中都需要先選擇需求中的一小部份，然後很快進行設計、實作與測試工作。雖然早期反覆中所得到的需求與設計可能不是最終想要的，不過在所有需求還未定案、整個投機性設計還沒定義出來之前，先用小的開發步驟，很快進行開發工作，有助於快速獲得回饋－從使用者、開發人員與測試（例如負載測

試與可用性測試）中獲得回饋。
這個早期回饋是很有價值的。我們不需要很投機地想做出正確的需求與設計，從真實的系統建構與測試過程所獲得的回饋可以增加我們對系統的實際了解，並且有機會修改或調整對需求或設計的認知。最終使用者也有機會很快看到部分系統，並且說：「對，這就是我希望你們做的，不過我真正想要的有些不同。」【註】這個「對…不過」的過程並不是系統失敗象徵。相反地，這些早期、經常發生、有結構性的「對…不過」循環反而是一種有技巧的做法，它可以不斷發堀並找出關係人真正想要的價值。然而，如前所述，這樣做並不是認同雜亂無章、回應式的開發方式，後者會讓開發人員不斷改變建構系統的方向，而前者則可能採取中庸之道。
【註】：更有可能的說法是：「你不了解我要什麼。」
除了釐清需求之外，有些開發活動像負載測試等等都可以在證明現在所擁有的部分設計與實作是朝正確方向做，或者我們有需要在下個反覆中改變核心架構。在早期就分辨並證實有風險性、關鍵的設計決策比後期才分辨並證實這些決策要好得多，反覆式開發方式正好可以提供這樣的機制。
這樣一來，工作就可以在一連串有結構性的建構－回饋－調整循環中逐漸完成。有件事是我們不會感到意外的，那就是早期反覆應該比後期反覆更偏離系統的「真實需求路徑」（從系統最終的需求與設計來看。）隨著時間過去，系統會逐漸收斂到真實需求路徑，如圖 2.2 所示。

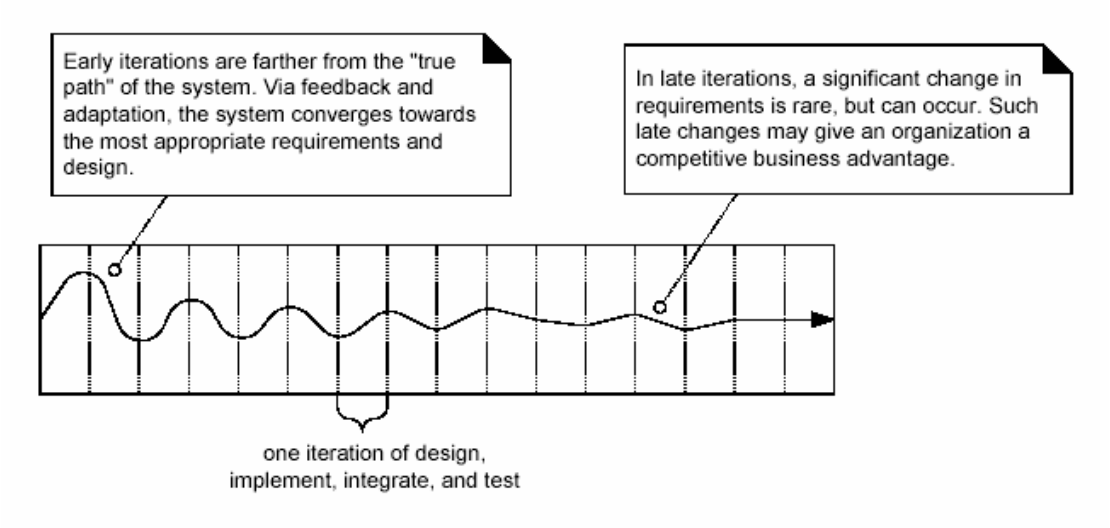


圖 2.2 反覆式的回饋與調整可以引導我們完成使用者想要的系統。
隨著時間過去，需求與設計的不穩定性會逐漸降低。

反覆式開發方式的優點

反覆式開發方式的優點包括：

- 在早期而不是後期開發過程中減緩（技術性、需求、目標、可用性等等）高

風險的危險性

- 早期就可以看到開發工作的進展
- 早期的回饋、使用者約定與調整都可以引導我們完成修正過、比較符合關係人真實需要的系統
- 可管理的複雜性；開發團隊就不會被「分析麻痺症（analysis paralysis）」或非常長、很複雜的開發步驟弄得受不了
- 從反覆中學到經驗，理論上可以隨著一個又一個的反覆，不斷改善開發流程本身。

反覆長度與固定時間長度（timeboxing）

UP（以及有經驗的反覆式開發人員）都建議反覆長度大約在兩個星期到六個星期之間。小的開發步驟、快速回饋與調整都是反覆式開發方式中的核心概念。比較長的反覆會違背反覆式開發方式的主要動機、增加專案風險。如果時間比兩個星期還要短的話，又很難完成足夠的工作量，得不到有意義的生產率與回饋；另一方面，比六個禮拜或八個禮拜長的話，複雜度會比較難接受，而且回饋速度也變慢。太長的反覆會失去反覆式開發方式的優點。短一點是好事。

固定時間長度（timeboxed 或者 fixed in length）是反覆的一個關鍵概念。舉例來說，如果我們敲定下次反覆是四個禮拜長，那麼這個不完整的系統應該在排定的日期內整合、測試完並且要更穩定－我們不鼓勵把日期往後延。如果很難符合截止日期的話，建議從反覆中移除工作或需求，然後把它們放在未來的某個反覆中，而不是延後完成日期。第 37 章裡面彙總了固定時間長度的理由。

大型開發團隊（例如有七百位開發人員）可能需要比六個星期更長的反覆以彌補花在協調與溝通上的時間；不過我們不建議反覆長度比三個月或六個月更長。舉例來說，在 1990 年代被成功替換掉的加拿大航空交通控制系統就是用反覆式生命週期與其它 UP 實務經驗開發的。整個專案包含 150 位程式設計師，並且用六個月長的反覆來組織專案【註】。不過，請注意甚至在這個長達六個月的專案反覆中，擁有 10 或 20 位開發人員的子系統團隊還是可以把他們的工作分解成六個長度一個月的反覆。

【註】：Philippe Kruchten 是當時這個專案的主要架構設計師，他也負責領導 RUP 的開發工作。

六個月長的反覆是針對大型開發團隊的例外情形，不是一種常態。爲了重複進行反覆，UP 中建議一般的反覆期間應該在兩個星期到六個星期之間。

第二節UP 的其它最佳實務經驗與概念

UP 中令人讚賞、實際可行的核心概念是短的、固定時間長度、反覆式、可調整的開發方式。

UP 中另一個不明顯、不過也很重要的概念是使用物件技術，包括物件導向分析與設計、與物件導向程式設計。

此外，UP 中的其它最佳實務經驗與關鍵概念包括：

- 在早期反覆中放進高風險、高價值的議題
- 持續讓使用者參與系統評估、回饋與需求
- 在早期反覆中，建構有一致性的核心架構
- 持續驗證品質；及早測試、經常測試以及用真實資料測試
- 用使用案例（捕捉需求）
- 用 UML ，以視覺方式建立軟體模型
- 小心管理需求
- 實行變動需求管理（change request managment）與配置管理（configuration management）

請參見第 37 章以了解這些實務經驗的細節。

第三節UP 中跟開發階段、時程導向有關的術語

UP 的專案把開發工作與反覆分配在四個主要開發階段中：

1. 初始階段－概略性的專案願景、企業案例、系統範圍與系統價值的評估工作。
2. 詳述階段－修正專案願景、用反覆式開發方式實作核心架構、解析系統高風險的部分、找出大部分的需求與系統範圍、更實際地評估系統價值。
3. 建構階段－用反覆開發方式實作其餘風險比較低、比較簡單的系統元素，並且準備配置工作。
4. 轉換階段－ beta 測試、配置工作。

接下來的章節中會對這幾個開發階段有更完整的定義。

這種開發方式並不是老式的「瀑布式」或循序式生命週期：先定義所有需求，然後做出全部或大部分設計工作。

初始階段並不是需求開發階段；它是評估可行性的開發階段。在這個開發階段中，我們要對系統做足夠的調查工作以做出繼續或終止專案的決策。

同樣地，詳述階段也不是需求或設計開發階段。在這個開發階段中，我們用反覆開發方式實作出核心架構，並且減緩高風險的議題。

圖 2.3 展示 UP 中常見、以時程導向的專用術語。請注意，一個開發週期是由多個反覆組成的，而且每個開發週期結束時，系統都會得到一個達產品品質的發行版本。

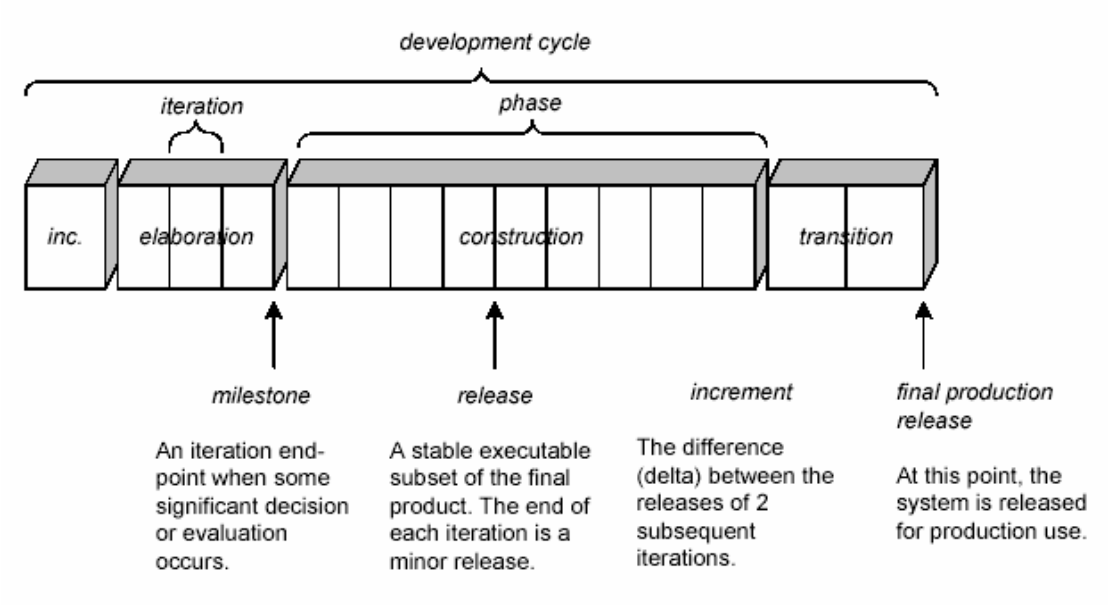


圖 2.3 UP 中以時程導向的專用術語

第四節UP 的工作科目（過去稱爲工作流程）

UP 在工作科目（原本稱爲**工作流程**【workflow】)【註】中描述相關的開發活動。我們先對工作科目做非正式的定義：在某個特定範圍內，工作科目就是一組相關的開發活動（也包括相關的工作成果），例如需求分析中相關的開發活動。在 UP 中，工作成果是代表任何工作成果的通用術語，包括：程式碼、網頁中的圖形、資料庫綱要、文件、圖、模型等等。

【註】：在西元 2001 年，他們用新的術語「工作科目」取代舊的 UP 術語「工作流程」以符合 OMG SPEM 在國際標準化中的努力。不過因爲有許多人已經採用 UP 之前的術語，所以他們還是繼續用工作流程代表工作科目的意義，不過事實上這兩個字的含意有點不同。在 UP 中，現在「工作流程」這個術語已經有新的、有點不同的意義：在某個特定專案中，它代表開發活動的某種特定順序（可能會橫跨數個工作科目）－工作的流程。

UP 中有好幾種工作科目，本書把焦點放在下面三種工作科目的工作成果上：

- **建立企業模型**工作科目－開發單一應用程式時，我們會產生它的領域物件模型。不過，當我們分析大型企業或進行企業流程再造時，還是會產生橫跨整個企業的企業流程動態模型。
- **需求**工作科目－針對某個應用程式所做的需求分析，例如寫出使用案例、找出非功能需求。
- **設計**工作科目－設計的各個構面，包含整體架構、物件、資料庫、網路連結等等。

圖 2.4 中包含更多的 UP 工作科目。

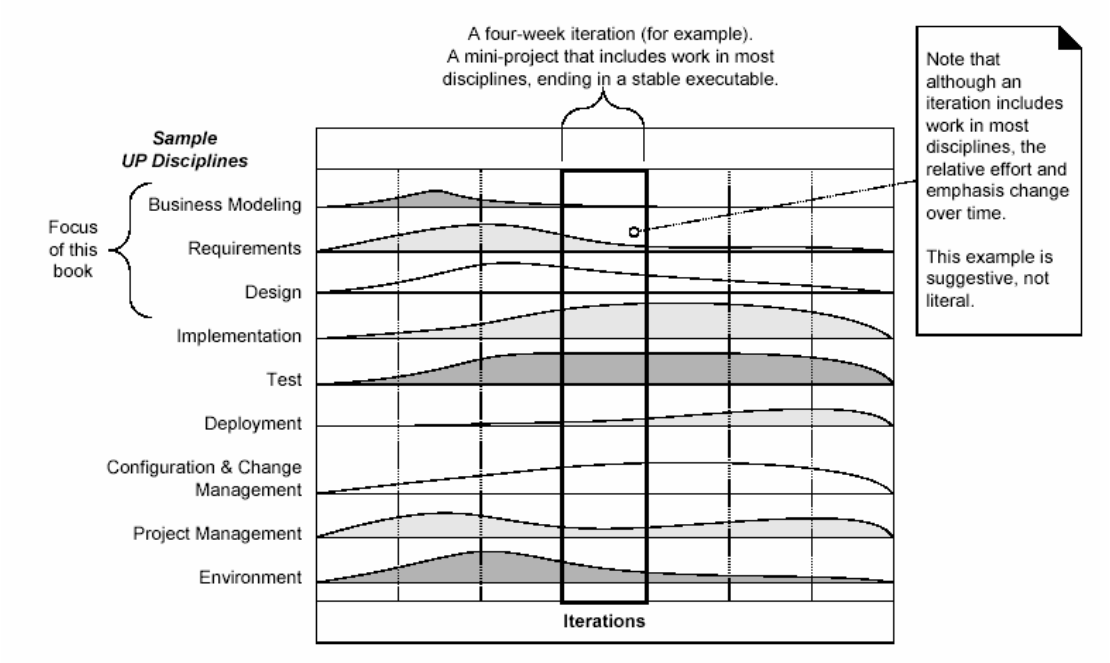


圖 2.4 UP 的工作科目【註】

【註】：從 RUP 產品摘錄出來的圖

在 UP 中，**實作**（implementation）代表寫程式與建構系統，而不是配置系統。**環境**（environment）工作科目代表設定專案用的工具與自定流程－換言之，安裝工具與流程所需的環境。

工作科目與開發階段

如圖 2.4 所示，反覆中的工作會涵蓋幾乎大部分或所有的工作科目。然而，在這些工作科目中所花費的工作量，會隨著時間不同比例也不同。很自然地，早期反覆中會把比較多的注意力放在需求與設計上，之後的反覆則比較少，因爲經由回饋－調整過程，需求與核心設計會逐漸穩定下來。

把這樣的概念用在 UP 的開發階段（初始階段、詳述階段等等）中，圖 2.5 告訴我們在不同開發階段中，各種工作科目所花費的相對工作量也會不同。請注意，圖 2.5 中的比例只是一種建議，不是固定不變的。舉例來說，雖然詳述階段中的反覆還是需要進行一些實作工作，相對來說裡面會有比較高的比例在做需求與設計的開發工作。在建構階段中，我們把比較多的焦點放在實作上，而把比較少的精神放在需求分析。

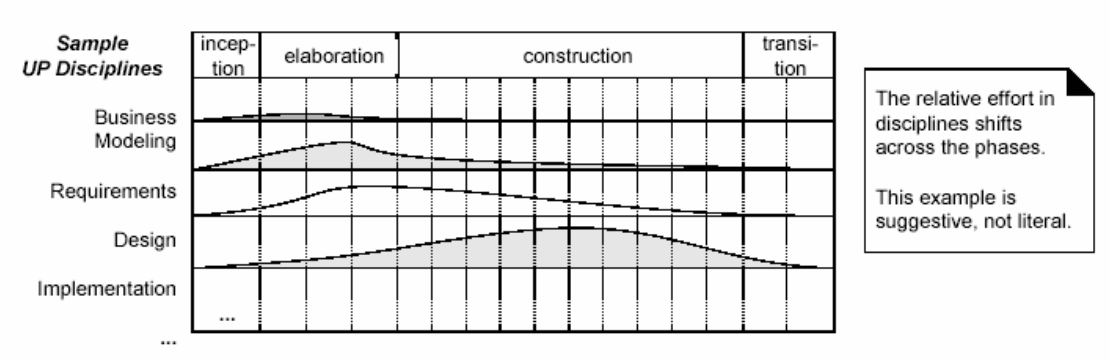


圖 2.5 工作科目與開發階段間的關係

本書結構以及 UP 中的開發階段與工作科目

在開發階段與工作科目方面，我們會把這個個案研究的焦點放在什麼地方？答案是：

我們把這個個案研究的焦點放在初始階段與詳述階段上。因爲本書主要目的是希望應用需求分析、物件導向分析與設計、樣式與 UML ，所以我們把重點放在建立企業模型工作科目、需求工作科目與設計工作科目的一些工作成果上。

本書前面的章節會介紹初始階段中的一些開發活動；後面的章節則探討詳述階段中的幾個反覆。接下來的清單與圖 2.6 則顯示本書跟 UP 中的開發階段相對應的結構。

1. 在初始階段的章節中，我們介紹需求分析基礎。
2. 在詳述階段的反覆 1 中，我們介紹基本的物件導向分析與設計，以及如何把責任分配到物件上。
3. 在詳述階段的反覆 2 中，我們把焦點放在物件設計，特別介紹一些最常用的「設計樣式。」
4. 在詳述階段的反覆 3 中，我們介紹各式各樣主題，例如架構分析與框架設計等等。

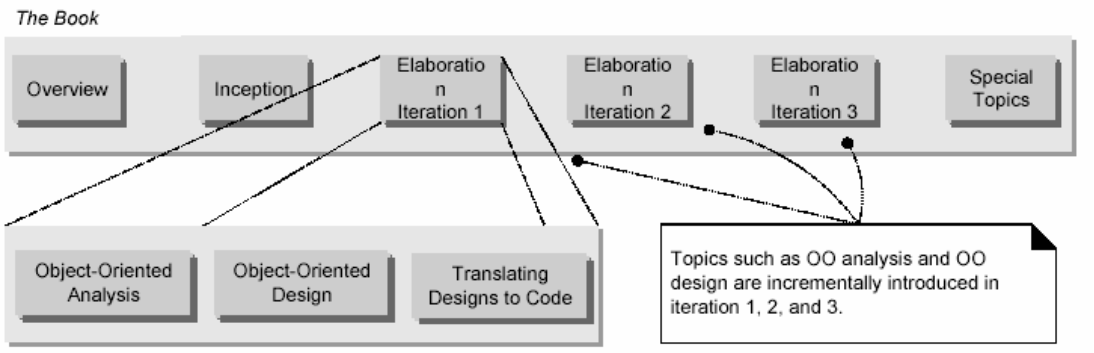


圖 2.6 跟 UP 的開發階段與反覆相對應的本書結構

第五節自訂流程與開發案例

選擇性的工作成果

有些 UP 的實務經驗與原則是不可變的，例如反覆、以風險驅動的開發方式，以及持續驗證系統品質等等。

然而，了解 UP 的關鍵概念後，你可以發現 UP 中所有的開發活動與工作成果（模型、圖與文件等等）都是*選擇性的*（optional）－當然，程式碼不可能是選擇性的！UP 中描述可能用到的工作成果，我們應該把它們當作藥房的藥看待。就像某個人可能無法區別到底服了多少種藥一樣，不過他吃的藥一定要符合疾病需要。同樣地，在 UP 專案中，開發團隊應該針對特定問題與需要選擇專案所需的一小組工作成果。一般來說，我們會把焦點放在有高度實務價值的*一小組*工作成果上。

開發個案（development case）

我們應該把某個專案中需要用到的 UP 工作成果記載在一份短的文件中，稱爲**開發個案**（development case）（環境工作科目中的工作成果）。例如表 2.1 這個開發個案是本書探討「NextGen 專案」時用到的工作成果。

接下來的章節會說明如何產生這些工作成果，其中包含領域模型、使用案例模型與設計模型。

當然，這個個案研究用的工作成果範例對所有專案來說可能不足或不合用。舉例來說，開發機器控制系統時，我們會畫大量的狀態圖。而開發一個網路版的電子商務系統時，則需要把焦點放在使用者介面雛型上。一個「尚未開發過」的新開發專案跟一個系統整合專案所需要的設計工作成果是非常不同的。

工作科目	**工作成果** 反覆→	**初始階段** I1	**詳述階段** E1..En	**建構階段** C1..Cn	**轉換階段** T1..T2
建立企業模型	領域模型		s		
需求	使用案例模型	s	r		
	專案願景	s	r		
	輔助規格書	s	r		
	字彙表	s	r		

設計	設計模型		s	r	
實作	軟體架構文件		s		
	資料模型		s	r	
	實作模型		s	r	r
專案管理	軟體開發計畫	s	r	r	r
測試	測試模型		s	r	
環境	開發案例	s	r		

表 2.1 UP 工作成果的開發個案範例。s—初版；r—修正版

第六節敏捷式統一流程

方法論把開發流程區分爲重量級（heavy） vs. 輕量級（light）以及預想式（predictive） vs. 可調整式（adaptive）。對他們來說，重量級開發流程（heavy process）是帶有輕蔑含意的一個詞，這樣的開發流程可能具有下面的性質：

- 產生許多有官僚氣息的工作成果
- 死板、受控制的
- 詳盡、長期、精細的規劃方式
- 預想的，而不是可調整的

就跟大部分專案一樣，**預想式開發流程**（predictive process）會嘗試用精細、比較長的開發流程展開計劃去規劃並預測開發活動與資源（人員）。預想式開發流程通常採用「瀑布式」或「循序式」生命週期—第一步，先定義所有需求；第二步，定義詳細的設計；第三步，實作系統。相反地，**可調整式開發流程**（adaptive process）認爲變動是不可避免的驅動力，並且鼓勵系統具有彈性、可調整，這樣的開發流程通常採用反覆式生命週期。所謂的**敏捷式開發流程**（agile process）隱含就是輕量級、可調整式的開發流程，以靈活應付變動的需要。

雖然 UP 的發明者並沒有打算把它當成重量級或輕量級的開發流程，不過在大家的印象中，UP 裡面有許多選擇性的開發活動與工作成果，這樣的印象可能讓大家覺得 UP 是重量級的。事實上，UP 裡面採用敏捷式開發流程的精神，它是所謂的**敏捷式統一流程**（agile UP）。下面是 UP 採用敏捷式精神的一些例子：

- 建議選用小部份的 UP 開發活動與工作成果。有些專案可能需要其它東西，不過通常來說，選用的東西少一點。
- 因爲 UP 是反覆式的，所以在實作之前，需求與設計還沒有完成。這些未完成的需求與設計會因爲回饋，在一連串的反覆中浮現出來。
- 不準備涵蓋整個專案的*詳細*計劃。我們會有一個高階計劃（稱爲**階段計劃【**phase plan**】**）裡面估算專案截止日期與主要的專案里程碑，不過裡面不會有專案里程碑的細部開發步驟。而詳細的計劃（稱爲**反覆計劃【**iteration

plan】）只會針對一個反覆做比較詳細的規劃，並且在一次又一次的反覆中，調整詳細規劃方式。請參見第 36 章裡面針對反覆式專案規劃方式所做的一些說明與解釋。

本書的個案研究把焦點放在少數的工作成果上，並且採用敏捷式統一流程的精神—反覆式開發方式。

第七節循序式、「瀑布式」的生命週期

跟 UP 的反覆式生命週期不同，舊式的生命週期是循序式、線性或「瀑布式」生命週期【Royce70】。常見的做法是定義出類似下面的開發步驟：

1. 釐清、記錄並確認完整、固定不變的需求。
2. 根據這些需求設計系統。
3. 根據設計結果實作。

在 MIT Sloan Management Review 裡面一篇長達兩年的研究中指出：成功軟體專案有四種常見成功因素，其中第一種成功因素就是反覆式開發流程而不是瀑布式開發流程【MacCormack01】【註】。

【註】：其它成功因素包括： (2).每天至少把新的程式碼整合成完整的建構版本一次，並且（經由測試）快速回饋設計上的變動； (3).一個曾經包裝過多種產品的團隊；以及 (4).及早把焦點放在建構並驗證有一致性的架構。這四種成功因素中的三種因素都是 UP 裡面明確描述的實務經驗。

我們在第 37 章中對循序式開發流程的問題有簡短描述，還說明了反覆式開發方式是如何減緩這些問題的。

第八節如何察覺自己還不是很了解 UP

如果你還不是很了解該如何採用 UP 以及 UP 所隱含、符合敏捷式開發精神的反覆式開發方式，下面是一些徵兆。

- 你認爲 初始階段 = 需求 、 詳述階段 = 設計 、 建構階段 = 實作 （換言之，把瀑布式生命週期強加在 UP 上。）
- 你認爲詳述階段的目的就是完整、小心地定義模型，之後會在建構階段把這些模型轉成程式碼。
- 你嘗試在開始進行設計與實作之前先定義出大部分的需求。
- 你嘗試在開始進行實作前，先定義出大部分的設計；你嘗試在反覆寫程式與測試程式之前，先定義出完整、確認過的架構。
- 在開始進行程式設計前，花「很長的時間」做需求與設計的開發工作。
- 你相信合適的反覆長度期間長達四個月，而不是四個星期（除了多達數百位開發人員的專案之外。）
- 你認爲畫 UML 圖與進行設計開發活動時，應該完整、精確、很詳盡地定

義出設計與模型，而寫程式只是把它們轉成程式碼的簡單機制而已。

- 你認爲採用 UP 就是盡量進行裡面所定義的許多開發活動、產生許多文件，而把 UP 視爲有許多開發步驟要遵守的正式、過分講究的開發流程。
- 你嘗試規劃專案開始到結束的詳細專案計劃；你嘗試很投機地預測所有反覆，以及每個反覆中會發生的事。
- 你希望在詳述階段完成之前，就完成可信任的計劃與評價。

第九節進階讀物

有一本針對 UP 與修定過的 RUP 所寫、可讀性很高的入門書 *Rational 統一流程*（The Rational Unified Process），作者是 Philippe Kruchten ，他是 RUP 的首席架構設計師。

最原始的 UP 說明文件可以在 The Unified Software Development Process 這本書找到，作者是 Jacobson 、 Booch 與 Rumbaugh 。這是一本很值得研究的書，不過建議大家先去讀 Kruchten 的入門書，因爲後者比較小本、也比較精簡，而且裡面介紹原始 UP 更新修正過的 RUP。

Rational Software 公司銷售線上網路版的 RUP 文件產品，裡面提供廣泛的 RUP 工作成果與開發活動讀物，也提供大部分工作成果的範本。第 37 章有 RUP 的簡短討論。施行 UP 專案的組織可以把這個產品的產品說明教練與書籍當作學習資源，不過有些人發現 RUP 也是一種很有用的學習工具、開發流程工具。

在由 Ambler 與 Constantine 所主編的一系列叢書中也有說明 UP 的開發活動（例如 The Unified Process : Elaboration Phase 【Ambler00】。）這些書裡面的文章都是這幾年來發表在 *Software Development* 雜誌上的文章，Ambler 與 Constantine 把它們用 UP 的開發階段與開發活動分類加以分門別類。請注意，雖然這些文章一開始不是專爲 UP 寫的，不過裡面提供一些很有用的建議。此外，也請注意一下，在這個系列中有一點錯誤：它們把 UP 的詳述階段當成產生丟棄式雛型的一個開發階段，以減少程式設計或設計時需注意的事項。這種說法不是很正確的；我們應該在詳述階段產生達到產品品質的設計結果與程式碼（雖然此時只產生部分系統而已。）Ambler 發現到這個錯誤，並且在稍後同系列的書中修正這項錯誤【註】。

【註】：private communication，作者 Ambler 。

至於其它敏捷式方法論，我們推薦終極流程（extreme programming，XP）（譯註：Extreme Programming 亦有人翻譯成終極程式設計）這一系列的書【Beck00、BF00、JAH00】，例如 Extreme Programming Explained 。XP 中大部分實務經驗（例如先寫測試程式【test-first】與反覆式開發方式）都跟 UP 實務經驗相通或完全一樣，我也鼓勵大家把這些 XP 實務經驗應用在 UP 專案中。請注意，短的、有時間限制、反覆式、可調整的開發方式並不是 XP 發明的（而且也不是只有 XP 宣稱這些實務經驗），UP 與其它反覆式方法論已經採用這個實務經驗多年。UP

與 XP 之間還是有兩點差異是值得注意的（除了這兩點之外還是有其它差異存在）： (1).UP 中建議用遞增方式撰寫使用案例與非功能需求文件（ XP 則否）；(2). UP 中建議在反覆開始、還沒寫主要程式之前，（花半天或一天的時間）畫出許多設計圖，而 XP 的領導者則建議花一點時間就好（例如 30 分鐘。）

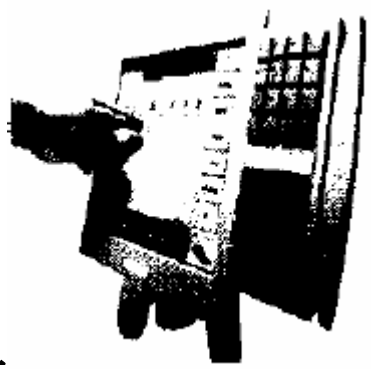
Highsmith 在 Adaptive Software Development 【Highsmith00】一書中，對可調整式開發方式的價值，做了一些說明。

第三章個案研究：NextGen POS 系統

很少東西可以比好的例子更讓人接受。
— Mark Twain

簡介

本章簡單描述本書所採用的個案研究。如果你了解這個領域，大可以跳過本章。事實上，我們之所以選擇這個個案研究，主要是因爲大家都很熟悉這個領域，而且裡面有許多有趣的設計問題與架構問題，讓我們可以專心討論如何進行分析與設計，而不用花太多時間在問題與領域上。



NextGen POS 系統

本書採用的個案研究是 NextGen 的點銷售（POS）系統。這個表面上看起來很直覺的問題領域，稍後我們可以發現裡面其實有非常有趣的需求與設計問題要解決。此外，這是一個很真實的問題；組織真的用物件技術寫 POS 系統。

POS 系統是一個電腦化的應用程式，我們會用它記錄銷售並處理付款；這個系統通常用在零售店。系統包括硬體元件（例如電腦與條碼掃描器），以及執行系統的軟體。它跟其它服務應用程式之間會有許多介面存在，例如由協力廠商所提供的稅金計算與庫存控制。我們都必須對這些系統做相關容錯處理；換言之，縱然暫時無法取得遠端服務（例如庫存系統），系統仍然要能夠處理銷售與現金付款（這樣一來生意才不會中斷。）

慢慢地， POS 系統必須支援各種不同的客戶端終端設備與介面。其中包括輕便型（thin）、以網頁瀏覽器爲基礎的終端設備、以標準個人電腦附上像 Java Swing 的圖形使用者介面、觸控式螢幕輸入設備、無線 PDA 等等。

此外，我們準備建立一個商業型的 POS 系統。爲了賣給不同客戶，系統必須可以處理不同的企業規則需要。每位客戶可能想要在系統使用情節中的某些特定點

執行一組專用邏輯，例如當我們開始一新的銷售，或者當我們加入新的產品項目。因此，我們需要用一種機制提供這種彈性與客製化。
應用反覆式開發策略，我們將一路從需求、物件導向分析、物件導向設計一直到實作爲止。

第一節構成系統架構的分層結構　以及個案研究焦點

典型的物件導向資訊系統設計時會分成幾個分層結構或子系統（請參見圖 3.1。）下面不是分層結構的完整清單，不過大家可以參考：

- **使用者介面**—圖形介面；視窗。
- **應用程式邏輯與領域物件**—軟體物件代表領域中的概念（舉例來說，有個叫做 *Sale* 的軟體類別）以滿足應用程式需求。
- **技術性服務**—提供技術性服務的一般目的物件與子系統，例如跟資料庫之間的介面或記錄錯誤的服務。這些服務通常跟應用程式無關，並且可以在不同的系統中重新使用。

一般來說物件導向分析與設計跟建立應用程式邏輯層與技術性服務層最相關。

NextGen 個案研究主要把焦點放在問題領域物件、分配責任到物件以滿足應用程式需求上。也應用物件導向設計產生技術性服務層，作爲資料庫跟應用程式邏輯層之間的介面。
用這種方式設計系統的話，使用者介面層的責任會非常少，這樣的使用者介面層稱爲*輕便型*客戶端。視窗裡面*不會*包含執行或處理應用程式邏輯的程式碼。相反地，使用者介面層會把工作請求轉送到其它層做。

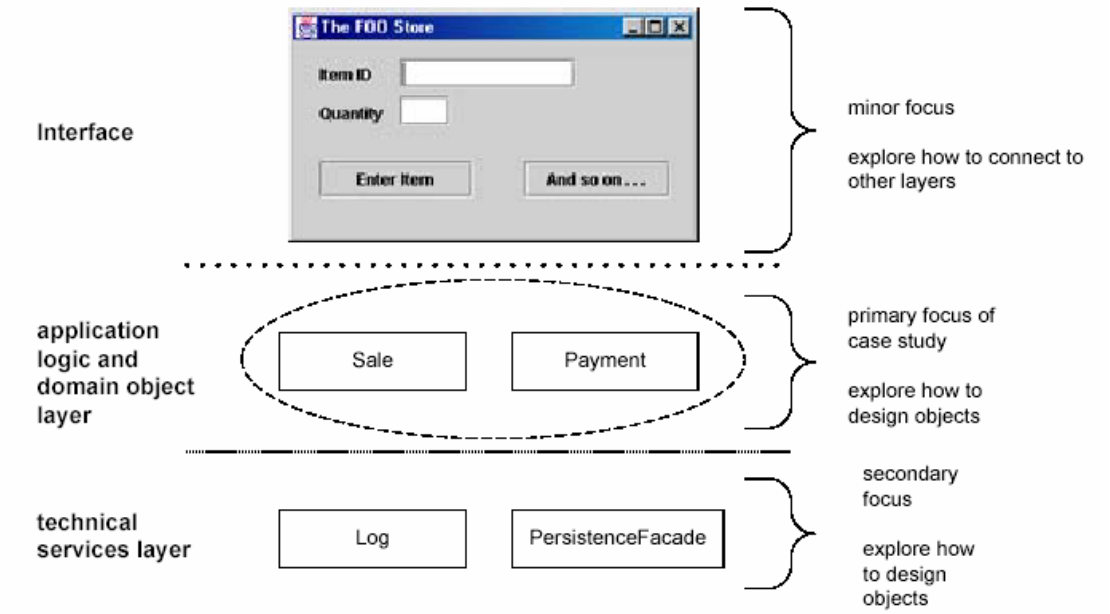


圖 3.1 物件導向系統中的分層結構與物件範例以及個案研究焦點

第二節本書策略：反覆式學習與開發方式

本書的編排方式遵循反覆式開發方式策略。我們把物件導向分析與設計分成多次反覆應用在 NextGen POS 系統上；第一個反覆是爲了一些核心功能。稍後的反覆則擴充系統的功能性（請參見圖 3.2。）爲了跟反覆式開發方式步調一致，我們在呈現分析與設計的相關主題、 UML 表示法與樣式時也用反覆、漸增的方式介紹它們。在第一個反覆中，我們呈現分析與設計的核心主題與表示法。第二個反覆中則延伸到新的概念、 UML 表示法與樣式。第三個反覆也是如此。

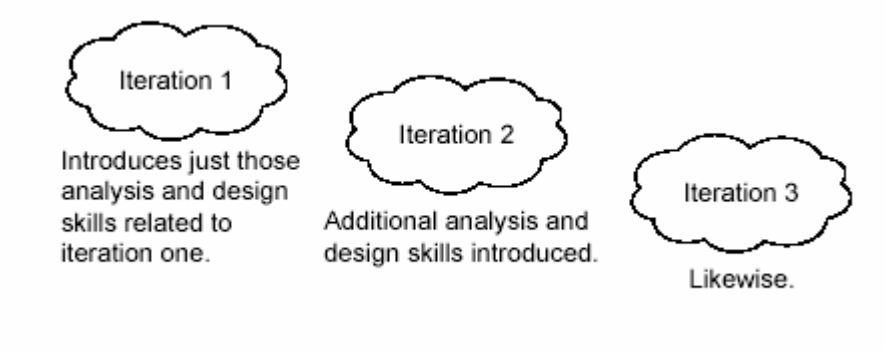


圖 3.2 按照反覆安排的學習路徑

第二部分初始階段

第四章初始階段

Le mieux est l'ennemi du bien（完美是現況之敵）。

— Voltaire

本章目標

- 定義初始階段。
- 說明第二部分其它章節的編排動機。

簡介

本章定義專案中的初始階段。如果開發流程的概念不是你最先想學的，或者你一開始想要把焦點放在這個開發階段中主要、實務的開發活動—建立使用案例模型—上，那麼你可以先跳過本章。

專案裡面都需要一個短的初始開發步驟以探討下面這些問題：

- 這個專案願景與企業案例爲何？
- 可行性？
- 買系統且／或自行建構？
- 粗略估算成本：專案要花 $ 10K-100K 美金或百萬美金？
- 我們應該繼續進行專案或終止專案？

爲了定義專案願景並獲得系統規模大小的估計值（這個估計值是不可信賴的），我們必須稍微了解需求。然而，初始階段這個開發步驟的目的並不是要定出所有需求，或者產生可信任的估計值或專案計劃。爲了避免過分簡化，這裡的概念是對需求做足夠的探討以形成潛在新系統的整體目的與可行性的合理選項，並且決定專案是否值得做更深入的探討（深入探討是詳述階段的目的）。

因此，對大部分專案來說，初始階段應該比其它開發階段短一點，例如一個禮拜或幾個禮拜長。事實上，如果初始階段的時間比一個禮拜長，那麼會模糊掉初始階段的焦點：初始階段的目的是爲了決定專案是否值得做真正的調查工作（真正的調查工作是在詳述階段做的），而不是做真正的調查工作。

用一個句子解釋初始階段：

建立產品範圍、專案願景與企業案例。

用一個句子說明初始階段中要解決的主要問題：

關係人是否對專案願景有基本認同，而且專案是否值得做真正的調查工作？

第一節初始階段：類推式的想法

在石油行業中，當他們考慮是否要開發一座油田時，會採取下面的步驟：

1. 決定是否有足夠的證據或企業案例可以證明探討性的探勘是合理的。
2. 如果有的話，做一些估計與探討性的探勘工作。
3. 提供油田規模與估計值的資訊。
4. 更進一步的開發步驟…

初始階段就像上面這樣的類推過程一樣。在步驟一，我們無法預測油田裡面會有多少儲油，或者萃取石油要花費的成本或工作量。現在談這些還言之過早—我們還沒有足夠的資訊。雖然不用花調查成本或工作量就可以回答「多少」或「何時」的問題是很好的，不過在石油這個行業中，我們都知道這是不實際的答案。

在 UP 的術語中，真正進行調查的開發步驟叫做詳述階段。初期進行的初始階段近似可行性研究，以決定是否值得做探討性探勘投資。經過調查之後（詳述階段），我們才有資料與深入的了解做出稍微可相信的評估值與計劃。因此，在反覆式開發方式與 UP 中，初始階段的計劃與估計值並不值得信賴。它只讓我們知道要花費的工作量大小，並且幫助我們決定是否繼續進行專案或終止專案。

第二節初始階段的時間可能很短

初始階段的意圖是針對專案目標建立某些初始、共同的專案願景，決定專案的可行性，並且決定詳述階段中某些正式調查工作是否值得做。如果我們已經先知道專案一定會進行，而且專案的可行性也很清楚（或許開發團隊之前已經做過類似的專案），那麼初始階段將特別短。這樣的初始階段可能包含第一次需求討論會、規劃第一次反覆，然後就很快轉到詳述階段。

第三節初始階段中需要開始做哪些工作成果？

表 4.1 中列出常見的初始階段（或早期詳述階段）工作成果，並且指出這些工作成果所專注的議題。接下來的章節將詳細討論部分議題，特別是使用案例模型。其中跟反覆式開發方式有關的一個關鍵點是：我們在這個開發階段中只會完成部分內容，然後在接下來的反覆中修正內容，甚至如果這些工作成果沒有真正實務價值的話，就不需要產生。此外，因爲現在是在初始階段，所以不需要太多的調查工作與工作成果內容。

舉例來說，使用案例模型（我們會在後面的章節中說明）裡面可能只先列出大部分預期使用案例與參與者的*名稱*就好，而且可能只描述 10% 使用案例的細節—針對系統範圍、目的與風險，完成粗略、高階的專案願景就好。請注意，我們可能會在初始階段寫一些程式、產生驗證概念用的雛型、經由（典型的）使用者介面導向雛型釐清一些需求，並且針對關鍵、會「阻礙專案前進【show stopper】」的技術性問題寫程式實驗一下。

工作成果【註】	說明
專案願景與企業案例	描述高階目標與限制、企業案例，並且

	提供管理用摘要。
使用案例模型	描述功能需求與相關的非功能需求。
輔助規格書	描述其它需求。
字彙表	關鍵領域術語。
風險清單與風險管理計劃	描述企業、技術性風險、資源風險、時程風險，以及減緩或回應這些風險的想法。
雛型與概念驗證	釐清專案願景，並且證實技術性概念是否可行。
反覆計劃	描述第一個詳述階段的反覆要做什麼。
開發階段計劃與軟體開發計劃	粗略猜想詳述階段的期間長度以及要花費的工作量。 開發工具、人力、教育訓練以及其它資源。
開發案例	針對這個專案，描述自定的 UP 開發步驟與工作成果。在 UP 中，我們一定要針對專案自定一個開發流程。

表 4.1 初始階段工作成果的範例

【註】：在初始階段中我們只會完成這些工作成果的部分內容，然後在接下來的反覆中反覆修正它們。這些工作成果的名稱都是 UP 中的官方名稱。

不是要製作很多文件嗎？

還記得工作成果都是可選擇性製作的。只有當工作成果真的可以增加專案價值時，我們才會製作這些工作成果，而且如果無法證實它們的價值就不需要製作這些工作成果。

工作成果的重點不是在文件或圖本身，而是思考、分析事物，以及在進行開發活動之前做好事先準備（然後是記錄事物以避免重新創造或不斷重述事物。）就像艾森豪將軍說過的：「我發現爲戰爭而準備的計劃通常沒用，不過規劃過程卻是不可或缺的。」【Nixon90、BF00】

用數位、網路—讓大家可以從專案網站上看到—方式記錄工作成果，不要用紙張記錄。

請注意，爲前一個專案所寫的 UP 工作成果可以在下個專案中使用。通常不同專案中的風險、專案管理、測試與環境工作成果內容會很類似。所有 UP 專案都將（或應該）用同樣方式、名稱（風險清單、開發案例等等）編排這些工作成果。這樣才比較容易跟新的 UP 銜接，或者從之前的專案中找到可重新使用的工作成果。

第四節如何察覺自己還不是很了解初始階段

- 大部分專案的初始階段會比「幾個」星期還長。
- 嘗試在初始階段中定義大部分的需求。
- 認爲估計值或計劃是可信賴的。
- 在初始階段中定義架構；而不是在詳述階段中用反覆方式完成架構。
- 你相信合理的工作順序應該是：(1).定義需求；(2).設計架構 ；(3).實作。
- 不需要企業案例或專案願景的工作成果。
- 沒有辨別出大部分使用案例與參與者的名稱。
- 寫出所有使用案例細節。
- 不寫任何使用案例細節，而不是：寫出 10-20% 使用案例的細節，讓大家對問題範圍有實際的了解。

第五章了解需求

快速、便宜、好：任選兩個。
— 無名氏

本章目標

- 定義 FURPS+ 模型。
- UP 工作成果跟各種需求間的關聯。

簡介

並不是所有需求都是一樣重要的。本章介紹 FURPS+ 中的需求分類方式。
需求代表系統—或者更廣泛的說法：專案—應該符合的能力與條件【JBR99】。
需求工作中的最大挑戰是用清楚的形式，跟客戶與開發團隊成員討論，以找出、溝通並記住（通常是記錄）哪些是真的需要的東西。
UP 裡面提倡一些最佳實務經驗，其中一個經驗就是*管理需求*。我們並不是要在專案的第一個開發階段，用瀑布式方法定義完整與穩定的需求，而是（在需求會有不可避免的動變以及關係人需求不明確的情況下）用「系統化方式找出、記錄、組織並追蹤系統的變動需求」【RUP】；簡言之，有技巧地管理需求而不是讓需求的管理變得很鬆散。請注意*變動*這個字；UP 接受需求會變動這個事實，並且把它當成專案的基本驅動力。*找到*（需求）也是另一個重要的字；換言之，就是利用一些技術（例如寫出使用案例與需求討論會）有技巧地找出需求。
如圖 5.1 所示，在一個針對有問題專案所做的研究中發現： 37% 的失敗因素都跟需求相關問題有關，其中產生需求議題佔整個問題的最大比率【Standish94】。因此，專精於需求管理是很重要的事。瀑布式開發方式對這個結果的回應方式是試著讓需求更完美、讓需求更穩定，並且在設計與實作之前，固定需求不變。不過歷史告訴我們，這樣做的結果註定失敗。另一方面，反覆式開發方式的回應方式則是接受變動，並且把回饋當成發掘需求的驅動力。

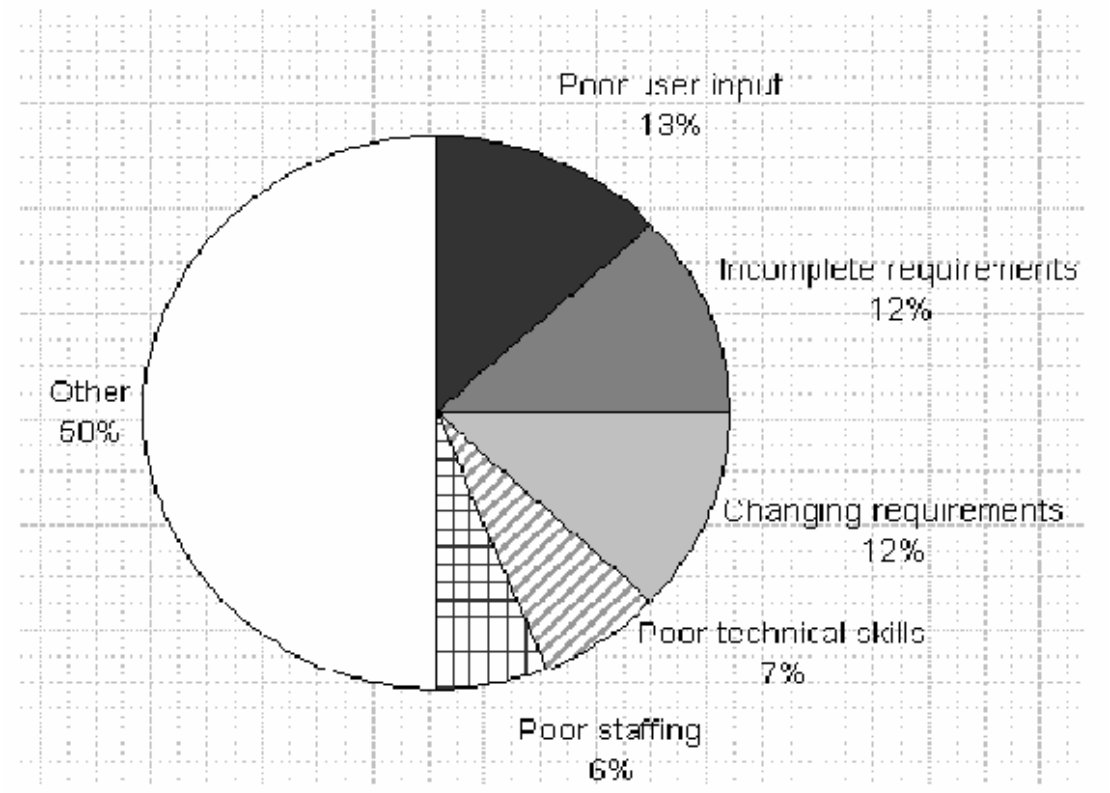


圖 5.1 有問題專案的失敗因素

第二節需求的種類

UP 中根據 FURPS+ 模型做需求分類【Grady92】，這個方便記憶的縮寫字涵義如下：【註】

【註】：有些書與標準組織（例如 ISO 9126 ，它跟 FURPS+ 的清單很類似）已經發表好幾種需求分類與品質屬性體系。此外，軟體工程協會（Software Engineering Institute，SEI）也制定好幾種需求分類體系；我們可以在 UP 專案中採用任何一種體系。

- **功能性**—系統特性、能力與安全性。
- **可用性**—人性因素、線上說明、文件。
- **可靠性**—錯誤發生頻率、從錯誤復原的能力、可預測性。
- **效能**—回應時間、產能、準確性、可取得性、資源使用率。
- **可支援性**—可調整性、可維護性、國際化程度、易配置性。

FURPS+ 中的「+」代表一些補充、次要的因素，例如：

- **實作**—資源的限制、語言與工具、硬體等等…
- **介面**—跟外部系統之間的介面所造成的限制。
- **操作性**—操作系統的相關設定以管理系統。
- **包裝**
- **合法性**—授權等等。

FURPS+ 的分類方式（或者叫做分類綱目）可以作爲檢查需求涵蓋率的檢核單，以降低沒有考慮到系統某個重要面向的風險。

在這些需求中，有些需求可以合稱爲**品質屬性**（quality attribute）、**品質需求**（quality requirement）或系統的「xx 性。」其中包括可用性、可靠性、效能與可支援性。一般而言，我們通常會把需求區分成**功能**（行爲的）或**非功能**（功能以外的所有東西。）有人不喜歡這種一般性的分類方式【BCK98】，不過這種分類方式是很常見的。

我們在使用案例模型中討論並記錄功能需求，使用案例模型是下個章節的主題，而且在專案願景工作成果中記錄系統特性清單。其它需求則可以記錄在相關使用案例或輔助規格書工作成果中。專案願景工作成果中會摘要說明高階需求，然後在其它工作成果中再詳細說明這些需求。字彙表中則記錄並澄清需求中會用到的術語。UP 中的字彙表也涵蓋**資料字典**（data dictionary）概念，裡面記錄跟資料相關的需求，例如驗證規則、可接受值等等。雛型是澄清哪些是想要或可能要的一種機制。

稍後討論架構分析時，我們就可以了解品質需求對系統架構有很大的影響力。舉例來說，高效能、高可靠性需求將影響我們選擇的軟體與硬體元件，以及它們的配置方式。此外，有可能因爲基本需求會經常改變，所以系統要能夠容易調整，這樣的需求也會影響到軟體的基本設計方式。

第三節進階讀物

跟需求與使用案例有關的參考資料可以在後面的章節中找到。使用案例導向的需求教科書，例如*使用案例最佳實務—寫作指南、秘訣與範本*（Writing Effective Use Cases）【Cockburn01】是研讀需求的最好起點，我們不建議大家先去看太一般性的（而且通常是傳統的）需求教科書。

在軟體工程知識團體（Software Engineering Body of Knowledge ， **SWEBOK**）這個網絡下有許多人廣泛討論需求—他們也討論各式各樣的軟體工程主題，你可以在 www.swebok.org 找到相關資料。

SEI （ www.sei.cmu.edu ）裡面有許多提案是跟品質需求有關的。 ISO 9126 、IEEE Std 830 與 IEEE Std 1061 都是跟需求與品質屬性相關的標準，可以在許多網站上找到這些標準。

閱讀一般性的需求書籍，甚至是那些號稱涵蓋使用案例、反覆式開發方式的書籍或 UP 中的需求資料時都需要注意下面幾點：

1. 大部分的寫法都偏向瀑布式開發方式，它們在進行設計與實作之前，都會有很明顯或完整的預先式（up-front）需求定義。這些資料對需求都有很深刻的理解，而且都是有廣泛價值、深入、很有用、跟方法論無關的。不過，我們要釐清的是：這些資料並沒有用反覆式開發方式的正確觀點呈現出來。主要原因是因爲這些作者的主要背景經歷都是瀑布式專案，他們在進行設計之

前，都會先修正需求、小心並完整定出確定的需求。這些提到反覆式開發方式的書，大多只是粗淺提到反覆式開發方式，有的甚至只是把「反覆式」教材加入書中以說明現代軟體工程的趨勢。因此，我們閱讀需求的書籍與文章時應該很小心；有些人的概念甚至被影響到，而嘗試在初始開發階段中小心定義出所有需求，這樣的想法跟反覆式開發流程並不一致。

2. 有許多一般性的需求書籍也號稱內容裡面有包含使用案例，事實上也只是初淺介紹使用案例，或者誤解以使用案例驅動需求的真正含意。其原因可能是因爲這些作者的主要背景經歷都是傳統的需求方法論，而且他們還嘗試把使用案例附加到之前的方法論上，事實上卻沒有真正感受到由 Jacobson 與 UP 所修正的使用案例核心概念：(1).把使用案例當成需求解決方案的核心—取代其它需求文件成爲需求的核心文件；(2).使用案例會涵蓋並驅動需求工作，它不是不重要或中等重要的，也不是傳統需求文件或解決方案的附屬性技術而已。

總而言之，一般性的需求書籍可以對技術、需求蒐集議題提供有用忠告，其內容也都是由熟練有實務經驗的工作者寫的，不過他們通常在瀑布式開發流程背景下提出忠告，而且對使用案例沒有很深入的理解。各式各樣對開發流程的忠告裡面都隱含「試著先定出大部分需求，接下來進行設計與實作」，這樣的忠告跟反覆式開發方式與 UP 的概念並不一致。

第六章使用案例模型：寫出某種情境下的需求

心想事成的第一個必要步驟是：決定你要什麼。

— Ben Stein

本章目標

- 找出並寫下使用案例。
- 說明使用案例跟使用者目標、基本企業流程之間的關聯。
- 簡單說明如何使用簡式、非正式或完整（使用案例）格式。
- 說明使用案例工作跟反覆式開發方式之間的關聯。

簡介

第一次閱讀本書時，本章對你來是非常有價值的，因爲現在使用案例已經是大家廣泛用來發掘並記錄需求的一種機制（特別是功能需求）；使用案例會影響專案很多地方，包含物件導向分析與設計。了解並產生使用案例是很有價值的事。寫使用案例—描寫系統使用方式的敘事情節—是一種可以讓我們了解並描述需求的很棒技術。本章探討關鍵的使用案例概念，並且用 NextGen 應用程式爲例子解說使用案例。

UP 在需求工作科目中定義了使用案例模型。基本上來說，使用案例模型裡面會包含所有的使用案例；它是系統功能性與環境的模型。

第一節目標與敘事情節

顧客與終端使用者都會有自己的目標（在 UP 中也可以稱爲*需要*【need】），而且希望電腦系統能幫他們達成目標，目標從記錄銷售到預估未來油井出油量都有可能。有許多方式可以捕捉這些目標與系統需求；簡單、大家熟悉的方式會是比較好的方法，因爲這樣一來大家—特別是顧客與終端使用者—都可以很容易捕捉到需求、說出自己的需求定義並評估現有需求，降低弄錯目標的風險。

使用案例這個機制可以讓需求保持在很簡單的狀態，並且所有關係人也都可以很容易瞭解使用案例。非正式來說，使用案例是利用系統達成目標的敘事情節。下面是*簡式*使用案例的一個範例：

> **處理銷售**：顧客帶著要買的東西到櫃檯結帳。收銀員用點銷售系統記錄顧客要買的每項商品。每記錄一個商品時，系統會不斷算出總價並秀出商品

明細。顧客輸入付款資訊之後，系統會驗證付款資訊並把它記錄下來。系統更新庫存資料。顧客從系統收到一份收據，帶著商品離開。

使用案例通常需要寫的比上面的例子還要詳盡。本質上，使用案例裡面總是寫出系統滿足不同關係人目標的敘事情節，我們用使用案例發掘並記錄功能需求；換言之，使用案例就是使用（系統）的個案（cases of use）【註】。記錄詳細情節的使用案例其概念跟簡式使用案例沒有什麼不同，不過我們很難找出或決定前者裡面應該寫些什麼，而且既詳細又有一致性的使用案例是很難寫出來的。

【註】：這個術語是從原本的瑞典字「usage case」直接翻譯成 use case 的。

使用案例是一種簡單的概念，不過一些有創造力、有想法的人往往在使用案例身上加上太多複雜的東西（雖然這些次要議題還是蠻有價值的），讓原本簡單的概念反而變得不清楚。大部分時候，建立使用案例模型的新手（或認真、資深的分析師）有可能因爲過份注意一些次要議題（例如使用案例圖、使用案例間的關係與使用案例套件、選擇性的屬性等等），而忽略一定要寫的敘事情節。總而言之，使用案例機制的威力在於它的複雜度與正式性是能夠視需要調整的。

第二節背景

用使用案例描述功能需求這樣的概念首先是由 Ivar Jacobson 【Jacobson92】在 1986 年所提倡的，他同時也是 UML 與 UP 的主要貢獻者。Jacobson 的使用案例概念是最初的想法，也受到廣泛重視；簡單又有用是使用案例的主要優點。此外，許多人也對使用案例有貢獻，其中最有影響力、影響範圍最廣泛、有一致性的是 Alistair Cockburn 在一本很流行的教科書*使用案例最佳實務－寫作指南、秘訣與範本*（Writing Effective Use Cases）【Cockburn01】中所定義的：何謂使用案例（或使用案例應該爲何）以及如何寫使用案例。（這本書的基礎是 Cockburn 自 1992 年以來早期的工作與文章）這些早年的工作是他稍後工作的基礎，前後工作具有相當地一致性。

第三節使用案例與（對參與者而言）有價值的概念

首先，我們先給大家一些非正式的定義：**參與者**（actor）是具有行爲能力的某種東西，例如一個人（我們用角色看待人）、電腦系統或組織；例如收銀員。

情節（scenario）代表（討論中的）系統與參與者之間一連串特定動作與互動情形；也有人稱之爲**使用案例實例**（use case instance）。它是使用系統過程中一段特定的敘事情節，或者是貫穿使用案例的一條路徑（譯註：使用案例的某些步驟會有替代情節，所以貫穿使用案例的路徑可能不只一條。）例如順利用現金購買

商品的情節或因爲信用卡交易失敗而無法完成購買商品的情節。

我們也非正式定義一下**使用案例**（use case）：使用案例代表參與者用系統達到目標的一組相關成功情節與失敗情節。舉例來說，下面是用非正式格式使用案例所寫的一個範例，裡面包含一些替代情節：

處理退貨

主要成功情節：顧客帶著要退還的商品到結帳櫃檯前面。收銀員用 POS 系統記錄每個要退還的商品…

替代情節：

如果信用授權被拒絕的話，告知顧客並要求他用其它付款方式退款。

如果系統中無法找到這項商品的識別碼，告知收銀員並建議他採用手動方式輸入識別碼（或許識別碼已經被改過。）

如果系統偵測到跟外部稅金計算系統之間的通訊中斷，…

RUP 中有另一種類似的使用案例定義：

使用案例中包含一組使用案例實例，其中每個實例代表系統執行的一連串動作，這些動作對特定參與者來說會產生有價值、看得到的結果【RUP】。

其中的「產生有價值、看得到的結果」看似簡單卻是很重要的一句話，因爲這句話強調系統行爲應該把重點放在提供使用者價值上。

對使用案例工作的關鍵態度就是要把焦點放在這個問題：「如何用系統提供使用者看得到的價值或滿足使用者目標？」而不是像送到洗衣店的清單一樣，把系統需求當成系統特性或功能清單。

或許提供使用者看得見的價值這種想法很淺顯易懂，不過軟體業裡面到處都是被丟棄一旁的失敗專案，這些專案都無法做出人們真正需要的東西。用系統特性與功能清單捕捉需求只會造成負面結果，因爲這種做法並不會鼓勵關係人在考量需求時，把需求放在比較大的系統使用情節中，這樣的情節才可以達成某種有價值、看得見的結果或某種目標（譯註：作者試圖讓大家了解系統特性與功能跟使用案例之間的差異；事實上，新手往往把使用案例跟系統特性與功能畫上等號。一般來說，系統特性比較適合記錄比使用案例更高階的目標，而功能又太瑣碎）相反地，使用案例用目標導向的情境取代系統特性與功能。這也是我們在本章標題加上「情境」的原因。

這也是 Jacobson 嘗試在使用案例概念中傳達的關鍵概念：把捕捉需求工作的焦點放在系統如何增加使用者價值、滿足使用者目標上。

第四節使用案例與功能需求

使用案例等於需求；裡面主要是功能需求，也就是系統應該做的事。在 FURPS+ 的需求分類中，特別強調「F」（功能或行爲的需求），不過我們也可以在使用案例中捕捉其它需求類型，特別是當這些需求類型跟使用案例之間存在很強烈的關

係時。在 UP 中－包括大部分現代方法論，使用案例是它們推薦用來發掘並定義需求的核心機制。使用案例代表承諾或合約，裡面定義系統應該如何運作。
爲了更明確一點：使用案例*等於*需求（雖然使用案例無法涵蓋所有需求。）有些人把需求當作「系統應該做的…」功能或系統特性清單。事實上不是如此，使用案例的關鍵概念（通常）是降低詳細、舊式系統特性清單的重要性與使用程度，而不是只用使用案例記錄功能需求。稍後章節會說明更多關於這一點的細節。
使用案例不是圖，它只是文字型的文件，建立使用案例模型時的主要工作也是在寫說明文字而不是畫圖。然而， UML 中定義了使用案例圖，圖中可以展現使用案例與參與者的名稱與關係。

第五節使用案例的型態與格式

黑箱型使用案例與系統責任

黑箱型使用案例（black-box use case）是最常見也是大家推薦使用的一種使用案例；這種使用案例不描述系統內部工作、元件或設計，只描述系統應該負擔的*責任*。責任是物件導向思考方式中很常見、一致的主題－軟體元素負擔責任，並且跟負有其它責任的軟體元素一起合作。
譯註：考量黑箱型使用案例與白箱型使用案例（white-box use case）時，還要注意 (1).系統邊界（system boundary）爲何、(2). 外部參與者（external actor） vs. 內部參與者（internal actor）與 (3). 使用案例層級（use case level）。當我們建立企業使用案例模型時，使用案例層級爲彙總目標層級（summary goal），一般稱爲企業使用案例（business use case）。我們用黑箱型企業使用案例【black-box business use case】描述顧客跟企業之間的*互*動情形、白箱型企業使用案例【white-box business use case】描述企業工作人員（business worker）之間的*互*動情形。外部參與者爲顧客，內部參與者爲企業工作人員，而系統邊界則是整個企業。
如果我們用黑箱型使用案例定義系統責任的話，就有可能明確指出*哪些*是系統必須要做的事（代表功能需求）而不用決定*如何*做這些事（代表設計。）事實上，「分析」與 「設計」意義上的差異通常也被歸納成「做什麼」與「如何做」。而且這樣的區別也是好的軟體開發中很重要的一個主題：在需求分析時避免做出「如何做」的決策，並且把系統外部行爲當成黑箱，明確描述它。稍後進行設計時，再產生符合規格的解決方案。

黑箱風格	不採用黑箱風格
系統記錄銷售。	系統把銷售寫入資料庫中…或者(甚至用更差的寫法)： 系統產生銷售用的 SQL INSERT 述

	句。

正式程度不同的使用案例格式

使用案例的寫法有很多種不同格式，你可以根據需要使用它們。除了黑箱型與白箱型兩種*可見性*不同的使用案例型態之外，使用案例的寫法還可以依*正式*程度不同區分：

- **簡式**（brief）－只有一個段落、精簡的摘要，通常只寫出主要成功情節。前面的*處理銷售*範例就是用簡式寫法。
- **非正式的**（casual）－用非正式、一段一段的文字記錄使用案例。在不同段落中描述各種情節。之前的*處理退貨*範例就是非正式的寫法。
- **正式的**（full dressed）－最詳盡的格式。詳細寫出所有步驟與變異之處，並且也寫出其它支援性小節，例如事先條件（precondition）與成功保證（success guarantee）。

下面範例是 NextGen 個案研究中用正式寫法的一個範例。

第六節一個正式的使用案例範例：處理銷售

正式的使用案例裡面會寫出很詳盡的細節，也有結構性；寫這些內容是爲了對目標、工作與需求有更深入的了解。在 NextGen POS 個案研究中，應該由系統分析師、主題專家（subject master）與開發人員一起合作，在早期的需求討論會中寫出這些使用案例。

由 usecases.org 所提供的格式

現在我們可以找到許多正式的使用案例範本。其中，最被廣泛使用、共享的格式是由 www.usecases.org 所提供的範本。下面的範例就是用這種風格寫出來的。請注意，這個範例是本書詳細使用案例的主要個案研究範例；我們可以在這個範例中看見許多常見的使用案例組成元素與相關議題。

使用案例 UC1：處理銷售

主要參與者：收銀員

關係人與利益：

－收銀員：希望準確、快速的輸入方式，也不會發生付款錯誤的情形，因爲收銀機抽屜如果短缺金額的話，要從他們的薪水裡扣。

－銷售員：希望銷售佣金能（隨銷售）馬上更新。

－顧客：希望買到商品、省時省力的快速服務。希望購買後保證能退貨。

－公司：希望準確記錄交易，並且滿足顧客喜好。希望記錄付款授權服務（Payment Authorization Service）的付款應收帳款。希望在無法獲得伺服器元件（例如遠端

信用驗證）的情況下，也能容錯、繼續捕捉銷售紀錄。希望自動、快速更新會計與庫存紀錄。

－政府稅務單位：希望收到每筆銷售稅金。賦稅單位可能有多個，例如國家級、州級、郡級。

－付款授權服務：希望用正確格式與協定收到數位授權請求。希望顧客對商店的應付帳款是很精確的。

事先條件：確認收銀員身分，並授權給此收銀員。

成功保證（事後條件）：儲存銷售。正確算出稅金。更新會計與庫存紀錄。記錄佣金。產生收據。記錄付款授權認可。

主要成功情節（基本流程）：

1. 顧客帶著要買的商品與／或服務到 POS 的結帳櫃檯前面。
2. 收銀員啓動一筆新的銷售。
3. 收銀員輸入商品識別碼。
4. 系統記錄銷售明細，並且顯示商品說明文字、價格與累計購買總金額。根據一組計價規則計算價格。

收銀員重複步驟 3-4 直到完成所有商品爲止。

5. 系統秀出包含稅金的總金額。
6. 收銀員告知顧客總金額，並且要求顧客付款。
7. 顧客付款，並且由系統處理付款。
8. 系統記錄完成的銷售，並且送出銷售與付款資訊到外部會計系統（爲了會計與佣金）與庫存系統（爲了更新庫存。）
9. 系統秀出收據。
10. 顧客帶著商品與收據（如果有的話）離開。

擴充情節（或替代流程）：

*a. 在任何時間點，當系統失效時：

爲了支援系統復原能力並修正會計資料，確認交易中所有容易受影響的狀態與事件，不論是在情節的哪個步驟都能復原。

1. 收銀員重新啓動系統、登入系統，並且要求還原成先前狀態。
2. 系統重新還原成之前的狀態。

2a. 系統偵測到阻礙復原的非正常狀態：

1. 系統告知阻礙復原的錯誤、從錯誤中還原，並且進入未改資料時的狀態。
2. 收銀員啓動一筆新的銷售。

3a. 無效的識別碼：

1. 系統告知錯誤，並且拒絕輸入商品。

3b.同種類的商品有多個，而且追蹤每個商品的唯一識別碼並不是很重要的事（例如五個有包裝過、同商品種類的蔬菜堡）：

1. 收銀員輸入商品種類的識別碼與數量。

3-6a: 顧客要求收銀員移除某個原先想購買的商品：

1. 收銀員輸入銷售中要移除的商品識別碼。
2. 系統更新現在的總銷售額。

3-6b. 顧客告訴銷售員要取消銷售：

1. 收銀員取消這一筆銷售。

3-6c. 收銀員暫停這筆銷售：

1. 系統把銷售記錄下來，而且之後可以從任何 POS 終端設備中取回這筆銷售紀錄。

4a. 不希望採用系統所產生的某項商品價格（例如顧客抱怨說某個東西應該是比較低的價格）：

1. 收銀員輸入超額部分金額。
2. 系統秀出新的價格。

5a. 系統偵測到跟外部稅金計算系統服務之間的通訊錯誤：

1. 系統重新啓動 POS 節點上的服務，然後繼續。
    1a. 系統偵測到服務並沒有被重新啓動。
        1. 系統告知錯誤。
        2. 收銀員可能手動計算稅金並輸入稅金，或者取消這筆銷售。

5b. 顧客說他們具有折扣身分（例如雇員、具有特別優惠身份的顧客）：

1. 收銀員對系統下折扣請求。
2. 收銀員輸入顧客識別資料。
3. 系統根據折扣規則，呈現總折扣額。

5c. 顧客說他們的個人帳戶中還有餘額，把這些餘額用在銷售中：

1. 收銀員發出使用餘額請求。
2. 收銀員輸入顧客識別資料。
3. 系統把餘額用在銷售中，直到價格 = 0 ，並且減少餘額。

6a. 顧客說他們想用現金付，不過現金不足。

1a. 顧客用其它付款方式。

1b. 顧客告訴收銀員取消這筆銷售。收銀員取消系統中的這筆銷售。

7a. 用現金付款：

1. 收銀員輸入顧客付的現金。
2. 系統呈現結餘金額，並打開現金抽屜。
3. 收銀員放進顧客付的現金，然後拿結餘金額給顧客。
4. 系統記錄現金付款金額。

7b. 用信用卡付款：

1. 顧客輸入他們的信用卡帳號資訊。
2. 系統送出付款授權請求到外部付款授權服務系統，要求付款認可。
    2a. 系統偵測到跟外部系統間的通訊發生錯誤：
        1. 系統告知收銀員錯誤。

2. 收銀員要求顧客用其它付款方式。

3. 系統收到付款認可，並且告知收銀員付款已經被認可過。

3a. 系統收到拒絕付款訊息：

1. 系統告知收銀員付款已經被拒絕。

2. 收銀員要求顧客用其它付款方式。

4. 系統記錄信用付款資訊，其中包括付款認可資訊。

5. 系統呈現信用付款簽名輸入機制。

6. 收銀員要求顧客做信用付款簽名。顧客輸入簽名。

7c. 用支票付款…

7d. 用賒帳付款…

7e. 顧客出示折價卷：

1. 在處理付款之前，收銀員記錄每個折價卷，而且系統把價格減掉適當金額。爲了處理會計問題，系統會記錄用過的折價卷。

1a. 所輸入的折價卷並不能用在任何購買的商品上：

1. 系統告知收銀員錯誤。

9a. 有產品回扣存在：

1. 系統呈現回扣型式以及回扣商品的回扣收據。

9b. 顧客要求贈品收據（不秀出任何價格）：

1. 收銀員要求贈品收據，然後系統秀出收據。

特殊需求：

－大型、平板監視器上的觸控式螢幕使用者介面。必須在 1 公尺外也能看得到上面的字。

－要能夠在 30 秒內回應 90% 的信用授權。

－由於某種未知原因造成存取遠端服務（例如庫存系統）失效時，希望系統有好一點的復原能力。

－所顯示的文字是國際化語言。

－能夠在步驟 3 到 7 之間嵌入企業規則。

－…

技術性或資料變異清單：

3a. （如果有條碼的話）用鍵盤或條碼雷射掃描器輸入商品識別碼。

3b. 商品識別碼可以是 UPC 、EAN 、JAN 或 SKU 其中任何一種編碼方式。

7a. 用讀卡機或鍵盤輸入信用帳戶資訊。

7b. 在紙收據上做信用付款簽名。不過在兩年之內，我們預期有許多顧客將希望用數位方式簽名。

發生頻率：應該幾乎會連續不斷發生。

開放議題：

－稅法有哪些地方會不同？

－探討遠端服務復原的議題。

－不同企業會有哪些客製化需求？
－收銀員登出系統時要帶著他們自己的現金抽屜嗎？
－顧客可以直接操作讀卡機，或者由收銀員操作？

這個使用案例只是展示用的，裡面並沒有列出全部內容（雖然這個使用案例是從真實 POS 系統需求來的。）然而，裡面有足夠的細節與複雜性可以讓我們真正感受到正式的使用案例裡面其實是可以記錄許多需求細節的。這個範例將是許多使用案例問題的模型。

另一種兩欄式的格式

有些人比較喜歡用兩欄式或對話式的格式，這樣的格式強調參與者與系統之間存在互動關係的事實。這種格式首先是由 Rebecca Wirfs-Brock 在【Wirfs-Brock93】中提出來的，而且 Constantine 與 Lockwood 兩人提倡這種格式對可用性分析與工程【CL99】有幫助。下面把同樣的情節用兩欄式格式表現出來。

使用案例 UC1：處理銷售

主要參與者：…

…跟之前一樣…

主要成功情節：

開發活動動作（或意圖）	系統回應
1. 顧客帶著要買的商品與／或服務到 POS 的結帳櫃檯前面。	
2. 收銀員啓動一筆新的銷售。	
3. 收銀員輸入商品識別碼。	4. 記錄銷售明細，並且顯示商品說明文字、價格與當時的總金額。根據一組計價規則計算價格。
收銀員重複步驟 3-4 直到完成所有商品。	5. 系統秀出包含稅金的總金額。
6. 收銀員告知顧客總金額，並且要求顧客付款。	
7. 顧客付款。	8. 處理付款。 9. 系統記錄完成的銷售，並且送出銷售與付款的資訊到外部會計系統（爲了會計與佣金）與庫存系統（爲了更新庫存。）系統秀出收據。
...	…

哪種格式是最好的？

沒有所謂最好的使用案例格式；有些人偏愛單欄式的，有些人則用兩欄式的。格式中的小節也都是可以隨需要自行增減；小節的命名方式也是可改變的。這些東西並不是特別重要；最重要的東西是用某種格式寫出主要成功情節與擴充情節的細節。【Cockburn1】裡面摘錄了許多種有用的格式。

個人實務經驗

這只是個人實務經驗而已，並不是一個建議。幾年以前，我用兩欄式的格式寫使用案例，因爲這種對話方式在視覺上有很清楚的區隔效果。然而，後來我轉用單欄式的格式寫使用案例，因爲這樣的格式比較簡潔、也比較容易格式化，而且從視覺上區隔對話其實價值不高，比不上單欄式的其它優點。我發現如果把每個參與者與系統回應分別寫在不同步驟，那麼用視覺區隔對話的最簡單方式就是：找出不同的參與者（顧客、系統等等。）（譯註：也就是說，只要分別強調參與者與系統回應就可以有區隔效果。）

第七節解釋使用案例中的各個小節

使用案例中的第一個組成元素

有許多小節可以當作使用案例中的第一個組成元素。請在使用案例一開始、主要成功情節之前擺很重要、大家需要閱讀的組成元素，並且把一些額外「有標題的」內容放到使用案例的後面部分。

重點：關係人與利益清單

當你之後再看這份清單時，會發現到它的重要性與實用價值遠比你剛開始看到它時高出許多。清單裡面會建議並限制系統該做的一些事。這裡引述一段話：

> 【系統】的運作情形跟關係人之間有合約關係存在，使用案例裡面會詳細說明這個合約的行爲部分…我們用代表合約中行爲的使用案例捕捉滿足關係人利益的*所有*相關行爲，而且使用案例裡面*也只*紀錄相關行爲【Cockburn01】。

這段話回答了一個問題：使用案例裡面應該有哪些東西？答案是：滿足關係人利益的東西。此外，在寫使用案例其它部分之前先列出關係人與利益，可以提醒我們哪些是系統應該做的詳細責任。舉例來說，如果一開始我們沒有列出銷售員這個關係人與利益，那麼我應該寫出處理銷售員佣金的責任嗎？有可能最後我會寫出這個責任，不過這代表我們在第一次需求會議中漏掉了這個東西。關係人利益

這個觀點提供我們完整、有條不紊的程序，以發掘並記錄所有需要的行爲。

關係人與利益：
－收銀員：希望準確、快速的輸入方式，也不會發生付款錯誤情形，因爲收銀機抽屜裡面如果短缺金額的話，要從他們的薪水裡扣。
－銷售員：希望銷售佣金能（隨銷售）馬上更新。
－…

事先條件與成功保證（事後條件）

事先條件（precondition）中陳述使用案例開始之前，*永遠必須*成立的東西。事先條件並*不是*在使用案例中才測試是否成立的，它們是使用案例開始之前就假設會成立的一些條件。在一般情形下，事先條件隱含其它使用案例的情節已經順利結束，例如登入或比較一般性的講法「已經識別收銀員身分，並且也已經授權給他。」請注意，雖然有些條件是一定要成立的，不過有可能寫下來其實並沒有什麼實務價值，例如「系統必須有電。」我們可以在事先條件這個小節中說明一些值得注意的假設，這些假設是寫使用案例的人認爲閱讀這個使用案例的人應該注意的事。

事先條件：確認收銀員身分，並且授權給此收銀員。
成功保證（事後條件）：儲存銷售。正確算出稅金。更新會計與庫存紀錄。記錄佣金。產生收據。…

主要成功情節與步驟（或基本流程）

這樣的情節被稱爲「快樂路徑（happy path）」或白話點「基本流程【basic flow】。」裡面描述滿足關係人利益的典型成功路徑。請注意，裡面通常*不*包含任何條件情況或分支情況。雖然加入分支情況沒有錯也沒有不合法，不過沒有分支或任何條件時更容易解讀使用案例，擴充時也更有一致性。我們可以把所有條件式處理情況放到擴充情節小節中。

建議
把所有條件式、分支敘述放到擴充情節小節中。

情節中會記錄一些步驟，這些步驟可以分成三類：
1. 參與者之間的互動情形【註】。
【註】: 請注意，當討論中的系統也扮演一個參與者角色跟其它系統一起合作時，這個系統本身也應該視爲一個參與者（譯註：所以這裡所指的參與者之間的互動情形其實就是參與者與系統之間的互動情形。）
2. （通常由系統完成的）驗證動作。
3. 系統所造成的狀態改變（舉例來說，記錄或修改某個東西。）

使用案例中的第一個步驟有時候不屬於上面三種分類之一，有可能是啓動情節的

觸發事件。

有一種常見的習慣寫法是：參與者的名稱一定要大寫以方便我們認出參與者。此外，也請注意範例中用來表示重複步驟的習慣寫法。

主要成功情節（基本流程）：

1. 顧客帶著要買的商品與／或服務到 POS 的結帳櫃檯前面。
2. 收銀員啓動一筆新的銷售。
3. 收銀員輸入商品識別碼。
4. …

收銀員重複步驟 3-4 直到完成所有商品爲止。

1. …

擴充情節（或替代流程）

擴充情節（extension）非常重要。它們代表主要成功流程以外的其它情節或分支情形，其中有成功的情形也有失敗的情形。請注意觀察在正式的使用案例範例中，擴充情節這個小節比主要成功情節小節要長、複雜得多；這種情形是常常會發生的，也是我們可以預期的。擴充情節也被稱爲「替代流程（alternative flow）」（譯註：或替代方案【alternative】。）

在內容完整的使用案例中，快樂路徑與擴充情節中的情節，兩者合起來應該可以滿足關係人「幾乎」所有利益。不過，這種說法有一點限制在，因爲有些利益可能比較適合當成非功能需求放在輔助規格書中，而不適合放在使用案例中。

擴充情節中的情節都是從主要成功情節分支出來的情形，因此也是用跟主要成功情節有關的表示法呈現。舉例來說，在主要成功情節的步驟 3 可能會遇到無效的商品識別碼，原因可能是不正確的輸入結果或系統未知的商品識別碼。這個擴充情節被標示成「3a」；情節中先說明發生條件，然後是系統回應方式。步驟 3 的另一個擴充情節則被標示成「3b」，依此類推。

擴充情節（或替代流程）：

3a. 無效的識別碼：

1. 系統告知錯誤並拒絕輸入商品。

3b. 同種類商品有多個，而且追蹤每個商品的唯一識別碼不是很重要的事（例如五個有包裝過、同商品種類的蔬菜堡）：

1. 收銀員輸入商品種類的識別碼與數量。

擴充情節分成兩個部分：發生條件與處理情形。

指引：在發生條件中寫出系統或參與者能夠*偵測到的*某種事物。爲了讓你更清楚些，這裡舉兩個相對照的例子：

5a. 系統偵測到跟外部稅金計算系統服務通訊中斷的錯誤。

5b. 外部稅金計算系統無法正常工作。

我們比較喜歡前面的寫法，因爲裡面所描述的是系統能偵測到的情形；後面的寫

法只是讓大家參考對照用。

我們可以把擴充情節的步驟摘要成一個步驟，或者用一連串步驟組成，就像下面這個例子一樣。此外，這個例子也讓我們知道發生條件可以涵蓋某個範圍內的幾個步驟：

3-6c. 收銀員暫停這筆銷售：

1. 系統把銷售記錄下來，之後就可以從任何 POS 終端設備中取回這筆銷售紀錄。

擴充情節的結尾之處都預設分支情節會回到主要成功情節，不過擴充情節也有可能不回到主要成功情節。

有時候，某個特別的擴充情節可能非常複雜，就像「用信用卡付款」擴充情節一樣。這時候，我們可能會想要把這個擴充情節變成一個獨立的使用案例（譯註：在*使用案例最佳實務－寫作指南、秘訣與範本*【Writing Effective Use Cases】一書中則建議用一般化關係【generalize】，我們可以先產生一個子使用案例「進行交易」，再把步驟 7a、7b、7c、7d、7e 變成這個子使用案例的特殊化使用案例。）這個擴充情節範例也示範失敗情形的擴充情節。

7b. 用信用卡付款：

1. 顧客輸入他們的信用卡帳號資訊。
2. 系統送出付款授權請求到外部付款授權服務系統，要求付款認可。
    2a. 系統偵測到跟外部系統間的通訊發生錯誤：
        1. 系統告知收銀員錯誤。
        2. 收銀員要求顧客用其它付款方式。
3. …

如果我們想要描述任何時候（或大部分步驟）都可能發生的擴充情節條件，我們可以用 *a 、 *b 等標籤。

*a. 在任何時間點，當系統失效時：

爲了支援系統復原能力並修正會計資料，確認交易中所有容易受影響的狀態與事件，不論是在情節中的哪個步驟都能夠復原。

1. 收銀員重新啓動系統、登入系統，並且要求還原成先前狀態。
2. 系統重新還原成之前的狀態。

特殊需求

如果某個非功能需求、品質屬性或限制條件跟某個使用案例特別有關係時，可以把它記錄在這個使用案例中。可能的品質需求包括效能、可靠性與可用性，或者被指定或可能發生的設計限制（通常是輸入／輸出裝置。）

特殊需求：

－大型、平板監視器上的觸控式螢幕使用者介面。必須在 1 公尺外也能看得到上面的文字。

－要能夠在 30 秒內回應 90% 的信用授權。
－由於某種未知原因存取遠端服務（例如庫存系統）失效時，我們希望系統有好一點的復原能力。
－所顯示的文字是國際化語言。
－能夠在步驟 3 到 7 之間嵌入企業規則。

把這些特殊需求記錄在使用案例中是 UP 的標準建議，當我們寫使用案例初版時，這裡是很合理的地方。然而，許多有實務經驗的人發現：爲了管理內容、易於理解與可讀性，最後把所有非功能需求合併到輔助規格書是很有用的，因爲進行架構分析時，通常應該整體看待這些需求。

技術性或資料變異清單

使用案例中通常會有技術性變異存在，說明*如何*用不同方式做某些東西，而不是該做些什麼，這些不同做法都應該記錄在使用案例中。常見的情形是：關係人提到輸入或輸出技術方面的技術性限制。舉例來說，關係人可能說：「 POS 系統必須用讀卡機與鍵盤支援信用帳號的輸入工作。」請注意，這些例子都是早期的設計決策或限制；一般來說，熟練的開發人員會避免做出不成熟的設計決策，不過有時候，某些設計決策是很明顯的或無法避免的，特別是那些跟輸入／輸出技術相關的設計決策。

有時候，我們也需要了解資料綱目上的變異，例如商品識別碼採用 UPC 或 EAN 條碼編碼方式。

我們用這份清單記錄這類型的變異情形。用這種方式記錄某個特殊步驟的資料變異也是很有幫助的。

技術性或資料變異清單：

3a. （如果有條碼的話）用鍵盤或條碼雷射掃描器輸入商品識別碼。
3b. 商品識別碼可以是 UPC 、EAN 、JAN 或 SKU 其中任何一種編碼方式。
7a. 用讀卡機或鍵盤輸入信用帳戶資訊。
7b. 在紙收據上做信用付款簽名。不過在兩年之內，我們預期有許多顧客將希望用數位方式簽名。

建議

本小節中*不*應該用多個步驟表達不同情況下的各種行爲。如果有需要的話，把它放在擴充情節小節中。

第八節使用案例的目標與範圍

應該如何發掘使用案例 呢？我們通常無法確信某個使用案例是否有效（或者更實際的說法：有用的。）我們可以把工作依照不同粗細程度加以群組，其中有些

使用案例可能只有一個或幾個小步驟，有些也可能包含企業層級的活動。
使用案例裡面應該要表達怎樣的層級與什麼範圍呢？
在接下來的小節中，我們詳細探討基本企業流程與目標的簡單概念，並且把這樣的概念當成框架，找出應用程式中的使用案例。

基本企業流程用的使用案例

下面哪個使用案例是有效的？

- 洽談跟供應商之間的合約
- 處理退貨
- 登入

有一點可能會有爭議，那就是：依據系統邊界、參與者與目標不同，它們都是屬於*不同層級*的使用案例。我們介紹基本企業流程之後會告訴大家這些候選使用案例的評估結果（譯註：作者會說明它們分別是屬於哪些使用案例層級的。）
「什麼是有效的使用案例？」這個問題太一般化了。對 POS 個案研究來說，比較相關的問題是：什麼層級可以有效表達應用程式需求分析中的使用案例？

指引：符合 EBP 的使用案例

對一個電腦應用程式 需求分析來說，我們應該把使用案例的焦點放在**基本企業流程**（elementary business process ，EBP）上。

EBP 是企業流程工程領域【註】所用的一個術語，定義如下：

【註】：**使用者工作**（user task）跟 EBP 很像，雖然在可用性工程（usability engineering）中，前者的意義比較不嚴謹。

> 爲了回應某個企業事件，一個人在某個地方、某個時間點所執行的工作。這個事件會增加可衡量的企業價值，並且讓資料處於一致狀態。例如，核准信用或爲訂單計價【很抱歉，我們無法找到這段話的起源。】

後面這個問題可能有點像在咬文嚼字：如果需要由兩個人一起執行工作，那麼這個使用案例就不是 EBP 嗎？可能不會，不過當你這麼想的時候，你對前面這段定義的感覺幾乎快要正確了。使用案例並不是像「刪除一項商品」或「列印文件」這樣的單一小步驟。主要成功情節中可能包含五到十個步驟。使用案例可能不用花好幾天或好幾段時間才能寫出來。它需要的時間有可能從幾分鐘到一個小時那麼長。就像 UP 的定義一樣，使用案例的焦點在於增加企業看得見、可衡量的價值，並且其結果是系統與資料都會處於穩定與一致的狀態。
常見的使用案例錯誤就是定義太多（譯註：目標）層級太低的使用案例；換言之，它可能只是 EBP 中的一個步驟、子功能或子工作。

違反 EBP 指引的合理情形

雖然應用程式的「基本」使用案例應該要滿足 EBP 指引，不過有時候產生一些

獨立的「子」使用案例，以代表基本使用案例中的子工作或一些步驟也是很有幫助的。我們可以擁有一些無法通過 EBP 測試的使用案例；大部分這樣的使用案例都是（譯註：目標）層級太低的使用案例。這個指引只適用應用程式 需求分析中有舉足輕重的使用案例；換言之，我們應該把焦點放在這樣的層級，寫出這種使用案例的內容。

舉例來說，像「用信用方式付款」這樣的子工作或擴充情節可能會重複出現在幾個基本使用案例中。因此我們會想把它分離出來變成使用案例（這樣的使用案例並無法不能滿足 EBP 指引），並且把它連到這幾個基本使用案例（譯註：包含關係【include】），以避免複製這些文字。

第 25 章會探討使用案例關係的一些議題。

使用案例與目標

參與者會有目標（或需要），並且用應用程式滿足這些目標。因此，我們把 EBP 層級的使用案例稱爲符合**使用者目標層級**（user goal）的使用案例，強調這樣的使用案例是爲了滿足系統使用者或主要參與者的目標（譯註：Cockburn 把使用案例層級【use case level】分爲彙總目標層級【summary goal】、使用者目標層級【user goal】與子功能層級【subfunction】。）

並且這樣的使用案例會導致我們採用下面的程序：

1. 找出使用者目標。
2. 定義每個使用者目標的使用案例。

對建立使用案例模型的人來說，這裡所強調的重點不太一樣。現在不是問「什麼是使用案例？」，而是問：「什麼是你的目標？」事實上，符合使用者目標的使用案例其名稱會反映出使用者目標的名稱，以強調使用者目標的觀點－目標：捕捉或處理一筆銷售；使用案例：*處理銷售*。

請注意，因爲使用者目標與使用案例之間會有一致性，所以 EBP 指引同樣可以用來決定目標或使用案例是否屬於合適的層級。

因此，這裡有一個關鍵概念是關於調查使用者目標 vs. 調查使用案例的：

想像一下，假設我們現在在一場需求討論會中。我們應該互問對方：

- 「你要做什麼？」（這個問題大致上是使用案例導向的問題），或者
- 「你的目標是什麼？」

回答第一個問題很像是要反映現有的解決方案與程序，也包括它們之間的糾葛關係。

回答第二個問題很像是用新視野找出新的、改良過的解決方案，特別是當我們在探討比較高階的目標層級時（「什麼是這個目標的目標？」）。此時，我們會把焦點放在增加企業價值，並且了解關係人想從正在討論的系統中真是想獲得的東西。

範例：應用 EBP 指引

假設當系統分析師負責發掘 NextGen 系統的需求時，你（收銀員）則負責探討使用者目標。兩人之間的對話可能像下面這樣：在一場需求研討會中：

系統分析師：「當你使用 POS 系統時，哪些是你的目標？」

收銀員：「有一個，我希望登入時能快一點。此外，系統要能夠捕捉銷售。」

系統分析師：「哪些高階目標會是登入的動機？」

收銀員：「我想讓系統識別我的身分，這樣一來，系統就能夠知道我是否有權限用系統捕捉銷售工作或其它工作。」

系統分析師：「有比這些目標層級更高的目標嗎？」

收銀員：「預防偷竊、竄改資料，顯示公司私有資訊。」

請注意，分析師的策略是搜尋目標層級以找到符合 EBP 原則、比較高階的使用者目標，察覺使用者行爲背後的真正意圖，並且了解這些目標的情境。

「預防偷竊…」是比使用者目標層級更高階的目標層級；我們可以稱爲企業目標層級（enterprise goal），這樣的目標不符合 EBP。因此，雖然這個目標可以激發我們思考問題與解決方案（例如刪掉整個 POS 系統與收銀員），不過現在我們還是把它放在一旁。

把目標層級降低到「識別並驗證身分」似乎比較接近使用者目標層級。不過這樣的目標層級符合 EBP 層級嗎？這樣的目標層級並沒有增加可得見或可衡量的企業價值。如果 CEO 問說：「你今天做了些什麼？」而你回答說「我登入系統 20 次！」相信他沒辦法想像你今天在做什麼。因此，它屬於次要目標層級（secondary goal），代表你是在進行某些有用的服務，不過它不屬於 EBP 或使用者目標層級（譯註：這裡把使用案例層級分爲企業目標層級【enterprise goal】、使用者目標層級【user goal】與次要目標層級【secondary goal】或子功能目標層級【subfunction goal】， Cockburn 用不同的名稱但含意一樣：彙總目標層級【summary goal】、使用者目標層級【user goal】與子功能層級【subfunction】。）

在另一個例子中，某些商店裡面會有稱爲「存入現金（cashing in）」的流程，此時收銀員會把他自己的現金抽屜放入終端設備中、登入、然後告訴系統抽屜中有多少現金；*存入現金*是 EBP 層級（或使用者目標層級）的使用案例；而登入步驟則不屬於 EBP 層級的使用案例，它是支援存入現金目標的子功能目標。

子功能目標與使用案例

「識別並驗證身分」（或者「登入」）不屬於使用者目標層級，這樣的使用案例是比較低的目標層級，稱爲**子功能目標層級**（subfunction goal）－支援使用者目標層級的子目標層級。我們只有在某些時候才會寫出這些子功能目標層級的使用案例，不過使用案例專家往往發現當他們在評估並改善（通常是簡化）一組使用案

例時，這些使用案例通常是問題所在。
寫出子功能目標層級的使用案例並非不合法，不過這樣的使用案例通常對我們沒有什麼幫助，因爲這種做法會增加使用案例模型的複雜度；造成系統中存在數百個子功能目標層級（或子功能層級）的使用案例。
譯註：寫出子功能目標層級的使用案例其最大好處大概就是少複製一些文字，但是這種做法會造成一些問題，包括：(1). 使用案例數目太多難以管理；(2). 難理解子功能目標層級使用案例的價值與 (3).開發人員不容易用這樣的使用案例進行分析、設計與實作。
很重要的一點：使用案例的數量與粗細程度會影響我們理解、維護與管理需求時所需花費的時間與複雜度。
寫出子功能目標層級使用案例最常見、有效的動機是：當你發現多個使用者目標層級使用案例中會重複這個子功能目標層級使用案例時，或者它是這些使用者目標層級使用案例的事先條件。
因此，我們還是有可能會寫出*授權使用者*這個子功能目標層級的使用案例。

目標與使用案例是可合成的

一般來說，目標是可以合成的，從企業層級目標（「獲利」）到許多支援性、中等程度目標（用應用程式「捕捉銷售」），再到支援應用程式的子功能目標（「檢查輸入是否有效」。）
很類似地，使用案例也可以分成不同層級以滿足這些目標，而且比較低階的使用案例可以合成變成比較高階的使用案例。

由於有層級不同的目標與使用案例，所以我們在尋找適合的應用程式層級使用案例時，常常會產生困擾。 EBP 指引可以讓我們濾掉大多數低階的使用案例。

第九節找到主要參與者以及跟主要參與者有關的目標與使用案例

使用案例是爲了滿足主要參與者的使用者目標而存在的。因此，定義使用案例的基本流程如下：

1. 選擇系統邊界。這是一個軟體應用程式還是需要整合硬體與軟體的系統？使用者是一個人還是整個組織？
2. 找出主要參與者—這些使用者會透過系統所提供的服務滿足他們的目標。
3. 針對每個主要參與者，找出他的使用者目標。把這些使用者目標合成變成滿足 EBP 指引比較高階的使用者目標層級。
4. 定義滿足使用者目標的使用案例；根據這些目標替使用案例命名。通常使用

者目標層級的使用案例跟使用者目標之間是一對一的，不過有一種會例外情形，我們稍後解釋這個情形。

步驟 1 ：選擇系統邊界

在這個個案研究中， POS 系統是我們要設計的系統；系統以外的所有東西都是在系統邊界外面。收銀員、付款授權服務等等都是在系統邊界外面。
如果系統邊界定義的不清楚，我們可以轉而哪些是定義系統邊界以外的東西—外部的主要參與者與支援性參與者—反向定義出設計中系統的邊界。一旦我們找到外部參與者，邊界就很清楚了。舉例來說，付款授權服務責任是在系統邊界內嗎？不是，因爲這個責任是由外部的付款授權服務參與者負責的。

步驟 2 、 3：找到主要參與者與目標

先找到主要參與者，再找到使用者目標，用這種線性方式找出需求，過程不是很自然；相反地，我們可以在需求討論會中用腦力激盪方式，產生混合主要參與者與使用者目標的需求。有時候，目標可以讓我們找到參與者，反之亦可能。
指引：進行腦力激盪時先找到主要參與者，因爲它會構成未來調查工作的框架。

請記住我們的問題是要找到參與者與目標

除了很明顯的主要參與者與使用者目標之預，試著問自己後面這些問題可以幫我們找出遺漏掉的參與者與使用者目標：

誰負責啓動或停止系統？

誰負責系統管理？

誰負責管理使用者與安全性？

「時間」是一個參與者嗎？如果系統會對時間事件回應那麼時間就是參與者。

如果系統當掉的話，會有監督的執行程序去重新啓動系統嗎？

誰負責評估系統活動或效能？

如何處理軟體更新工作？推式或拉式的更新方式？

誰負責評估系統記錄？這些系統記錄是從遠端讀取的嗎？

主要參與者與支援性參與者

回想一下，主要參與者會經由系統所提供的服務滿足他們的使用者目標。他們要求系統幫助他們。相對而言，支援性參與者則提供服務給設計中的系統。我們現

在暫時把焦點放在主要參與者而不是支援性參與者上。
也請回想一下，系統可能存在於其它系統當中，因此主要參與者也可能是其它電腦系統，例如「監視狗」軟體處理程序。

建議
如果發現主要參與者裡面沒有外部電腦系統，請注意：這種情況是很可疑的。

參與者－目標清單

回想一下在參與者—目標清單中的主要參與者與使用者目標。在 UP 的工作成果中，我們會把這份清單放在專案願景工作成果的其中一個小節（下一章會說明這個工作成果。）
舉例來說：

參與者	目標		參與者	目標
收銀員	處理銷售 處理出租 處理退貨 存入現金 取出現金 …		系統管理員	新增使用者 修改使用者 刪除使用者 管理安全性 管理系統表格 …
經理	啓動 關閉系統 …		銷售活動系統	分析銷售與績效資料
…	…		…	…

銷售活動系統是一個遠端應用程式，它經常透過網路跟每個 POS 節點要求銷售資料。

規劃專案時會用到的維度

事實上，這份清單裡面會有額外的欄位來記錄優先順序、工作量與風險；第 36 章中會簡單介紹這份清單。

很混亂的實際情形

這份清單看起來好像很整齊，不過實際上產生這份清單的過程卻不是這回事。我們需要在需求討論會中進行很多腦力激盪、反覆推敲之後才能產生這份清單。想一想之前在討論「登入」使用者目標時應用 EBP 規則的情形。系統分析師會深入了解需求，並且提昇低階的登入機制目標層級（收銀員可能想像使用 GUI 對話框的情形）到比較高階的「識別身分並授權給我。」然而，分析師稍後理解到這樣的目標並無法通過 EBP 指引，而不把它視爲使用者目標。當然，實情可能有點不同，因爲有經驗的分析師會從過去經驗或研究中得到一些啓發，例如「使

用者授權很少符合 EBP 指引」，而能夠很快去掉這樣的目標。

根據系統邊界決定主要參與者與使用者目標

為什麼在*處理銷售*使用案例中，主要參與者是收銀員而不是顧客呢？為何顧客沒有出現在參與者—目標清單中？

答案會因設計中系統的系統邊界不同而有差異，如圖 6.1 所示。如果把企業或結帳服務視為整個服務、目標是取得商品或服務，然後離開，那麼顧客*就是*主要參與者。然而，如果只從 POS 系統（它是這個個案研究中所選定的系統邊界）的觀點來看，系統則是為了滿足收銀員（與商店）的目標處理顧客銷售的。

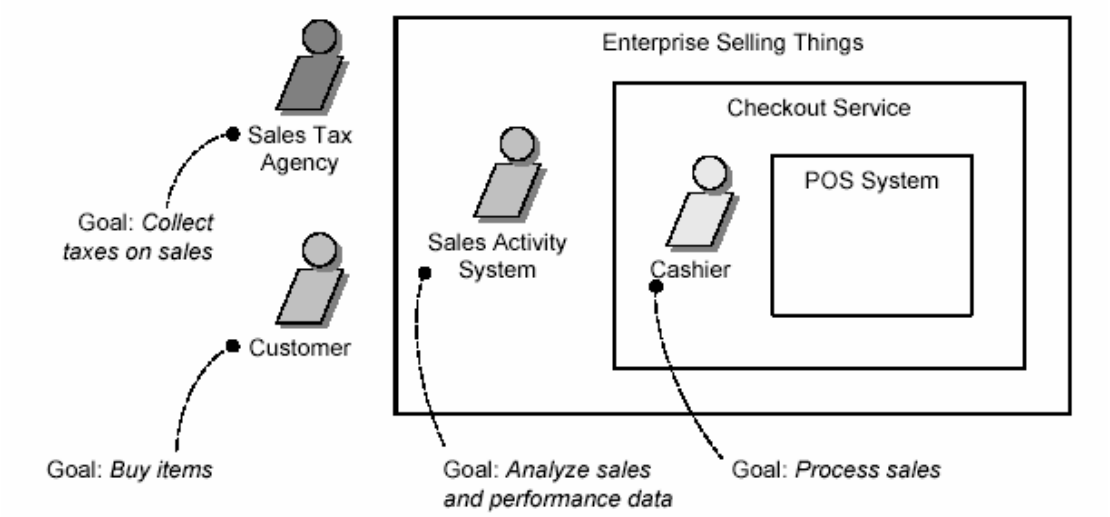


圖 6.1 不同系統邊界下的主要參與者與目標

由事件分析中找到參與者與目標

另一種方式可以協助我們找到參與者、目標與使用案例，那就是：找出外部事件。哪些是外部事件、它們是從哪裡來的、為何有那些外部事件存在？通常，同一組事件會屬於同一個 EBP 層級的目標或使用案例。舉例來說：

外部事件	從哪個參與者來的	目標
輸入銷售商品的明細	收銀員	處理一筆銷售
輸入付款	收銀員或顧客	處理一筆銷售
…		

步驟 4 ：定義使用案例

一般來說，我們會針對每個使用者目標定義一個相對應的 EBP 層級使用案例。並且把這個使用案例跟使用者目標用類似名稱命名—例如目標：處理一筆銷售；使用案例：處理銷售。

此外，使用案例名稱請用動詞起頭。

在「每個使用者目標會對應一個使用案例」這個慣例中，有一個常發生的例外情形就是把個別的 CRUD（產生、讀取、更新與刪除）目標整合成一個 CRUD 使用案例，習慣性稱呼爲*管理＜X＞*。舉例來說，像「編輯使用者」、「刪除使用者」等等都可以用*管理使用者*使用案例替代。

「定義使用案例」時會花費的工作量，程度上可能會有些不同，從幾分鐘或簡單記錄名稱到花幾個星期寫出正式的版本。本章稍後的 UP 開發流程小節會把這個工作—何時與花多少時間寫使用案例—放在反覆式開發方式與 UP 的情境下討論。

第十節恭喜：現在你已經寫出一些使用案例了，

不過它們還不是很完整

溝通與參與的需要

NextGen POS 開發團隊在一連串短暫的開發反覆中，透過多次的需求討論會寫出使用案例，並且根據回饋不斷增加、修正或調整這些使用案例。主題專家、收銀員與程式設計師都很主動參與使用案例的寫作過程。在收銀員、其它使用者或開發人員之間沒有任何中介人員存在；他們都直接溝通。

這樣做很好，不過還不夠好。寫好需求規格書之後，我們會存有需求正確的幻象；不過事實上不是如此。我敢保證使用案例與其它需求依然還不是很正確。這些需求缺乏關鍵資訊，裡面也會有一些錯誤描述。解決之道不是用「瀑布式」開發流程的態度，嘗試並很努力地想要一開始就記錄完美、完整的需求，當然我們還是會在時間允許的情形下盡力做到最好。不過需求總是嫌不夠。

我們需要採用另一種不同方式。大部分來說，我們會採用反覆式開發方式，不過還會加入其它想法：*持續不斷地進行人員溝通（ongoing personal communication）*。開發人員與某些了解領域、能做需求決策的人，每天持續、密切地參與與溝通。當程式設計員有問題時，可以花幾秒走到某人旁邊，馬上釐清問題。舉例來說， XP 實務經驗【Beck00】裡面就有一個絕佳的建議：*使用者*

全職在專案開發場所協助專案開發。

第十一節用精練、不含使用者介面的風格寫出使用案例

新的與改良過的（解決方案）！指紋辨識的例子

調查並詢問目標而不問工作或流程的做法，可以鼓勵我們把焦點放在需求本質上—找出需求的真正意圖。舉例來說，在需求討論會中，收銀員可能會說他的目標之一是「登入。」此時，收銀員可能是想像一個 GUI 畫面、對話盒、使用者 ID 與密碼。登入本身只是達到目標的一種機制而不是一種目標。不斷往目標階層的上層調查上去（問說「什麼是這個目標的目標」），系統分析師可以找到跟機制無關的目標：「識別身分並授權給我」或甚至更高層的目標：「預防偷竊…」
這樣的發掘過程可以開闊專案願景、形成新的、改良過的解決方案。舉例來說，配上生物測定讀取機（通常用來做指紋辨識）的鍵盤與滑鼠已經很普遍也不貴。如果目標是「識別並授權」，爲何不用鍵盤上的生物測定讀取機，讓過程更簡單、更快？不過合適的答案還要配合一些可用性分析工作，例如了解標準使用者的概況。他們手指上是否總是沾滿了油？或者他們有手指嗎？

本質性寫作風格

在許多使用案例指引中，這個概念已經被彙總成「拿掉對使用者介面所做的說明；把焦點放在意圖上」【Cockburn01】。這個概念的動機與表示法可以在 Larry Constantine 說明如何產生比較好的使用者介面（UI）與進行可用性工程中找到比較完整的討論【Constantine94】。當我們省略 UI 細節，並且把焦點放在使用者的真正意圖上時，把這種做法被 Constantine 稱爲**本質性**寫作風格【註】。
【註】：本質性（essential）這個字是從 Essential System Analysis 【MP84】中的「essential models」來的。
當我們採用**本質性**寫作風格時，描述事情的層次會達到使用者的*意圖*與系統的*責任*，而不是它們的實際動作。裡面避免提到技術與機制上的細節，特別要跟 UI 無關。

用本質性風格寫使用案例；省略掉使用者介面，並且把焦點放在參與者的意圖上。

本章之前的所有使用案例範例（例如處理銷售）都是用本質性風格寫的。
請注意，字典中把目標定義成意圖的同義字【MP89】，正好說明 Constantine 的本質性風格概念與本章之前所強調的目標導向觀點之間的關聯性。事實上，本質性風格中所寫出說明參與者意圖的步驟，有許多可以歸類成子功能*目標*。

兩個互相對照的例子

本質性風格

假設*管理使用者*使用案例中需要識別與授權。由 Constantine 所提倡的本質性風格是用兩欄式格式寫的，不過我們也可以用單欄式格式來寫。

…	
參與者意圖	系統責任
1. 管理員識別自己身分。	2. 身分授權。
3. …	

單欄式格式如下所示：

…
1. 管理員識別自己身分。
2. 系統做身分授權。
3. …

針對這些意圖與責任，可採用的設計解決方案範圍會很廣：生物測試讀取機、圖形化使用者介面（GUI）等等。

具體風格—避免在早期需求工作中用這種風格

相對地，還有另一種**具體使用案例**（concrete use case）風格。用這種風格時，使用者介面方面的決策會放在使用案例的說明文字中。這些說明文字裡面甚至還會有視窗畫面雛型、討論視窗瀏覽方式、 GUI 視窗中小元件操作方式等等。舉例來說：

…
1. 管理員在對話盒中輸入 ID 與密碼（請參見圖 3 。）
2. 系統授權給管理員。
3. 系統顯示「編輯使用者」的視窗畫面（請參見圖 4。）
4. …

這些具體的使用案例對稍後完整或詳細的 GUI 設計工作可能很有用，不過在早期需求分析中寫這些東西可能不是很恰當。在早期需求工作中，我們最好「省略使用者介面—把焦點放在意圖上。」

第十二節參與者

參與者代表任何具有行爲的東西，當討論中系統（system under discussion，SuD）呼叫其它系統所提供的服務時，它也算是參與者【註】。主要參與者與支援性參與者都會出現在使用案例說明文字的動作步驟中。參與者不只可能是由人扮演角色，也有可能是組織、軟體與機器。在 SuD 中可能會有三種外部參與者：

【註】：這是修正、改良過的另一種參與者定義，舊定義包括 UML 與 UP【Cockburn97】早期版本中的定義。之前的舊定義並不一致，它把 SuD 排除在參與者之外，縱然 SuD 呼叫其它系統的服務時也一樣。所有實體（包括 SuD）都可能扮演多重*角色*（role）。

- **主要參與者**— SuD 的服務可以滿足這種參與者的使用者目標。例如收銀員。
  - 爲何要找出這種參與者呢？是爲了找到使用者目標，這些目標會驅動使用案例。
- **支援性參與者**—提供服務（例如資訊）給 SuD 。自動化的付款授權服務就是一個例子。通常支援性參與者是電腦系統，不過也有可能是組織或個人。
  - 爲何要找出這種參與者呢？是爲了釐清外部介面與跟外部之間的協定。
- **檯面下的參與者**—使用案例的行爲會對檯面下參與者產生利益，不過這些參與者不是主要參與者或支援性參與者，例如政府稅金機構。
  - 爲何要找出這種參與者呢？是爲了確認已經找到所有關係人利益、也都滿足這些利益了。檯面下參與者的利益有時候很難找到或很容易沒注意到，除非我們先找出並明確地替這些參與者命名。

第十三節使用案例圖

UML 中有提供使用案例圖的表示法以展示使用案例與參與者的名稱，以及它們之間的關係（請參見圖 6.2。）

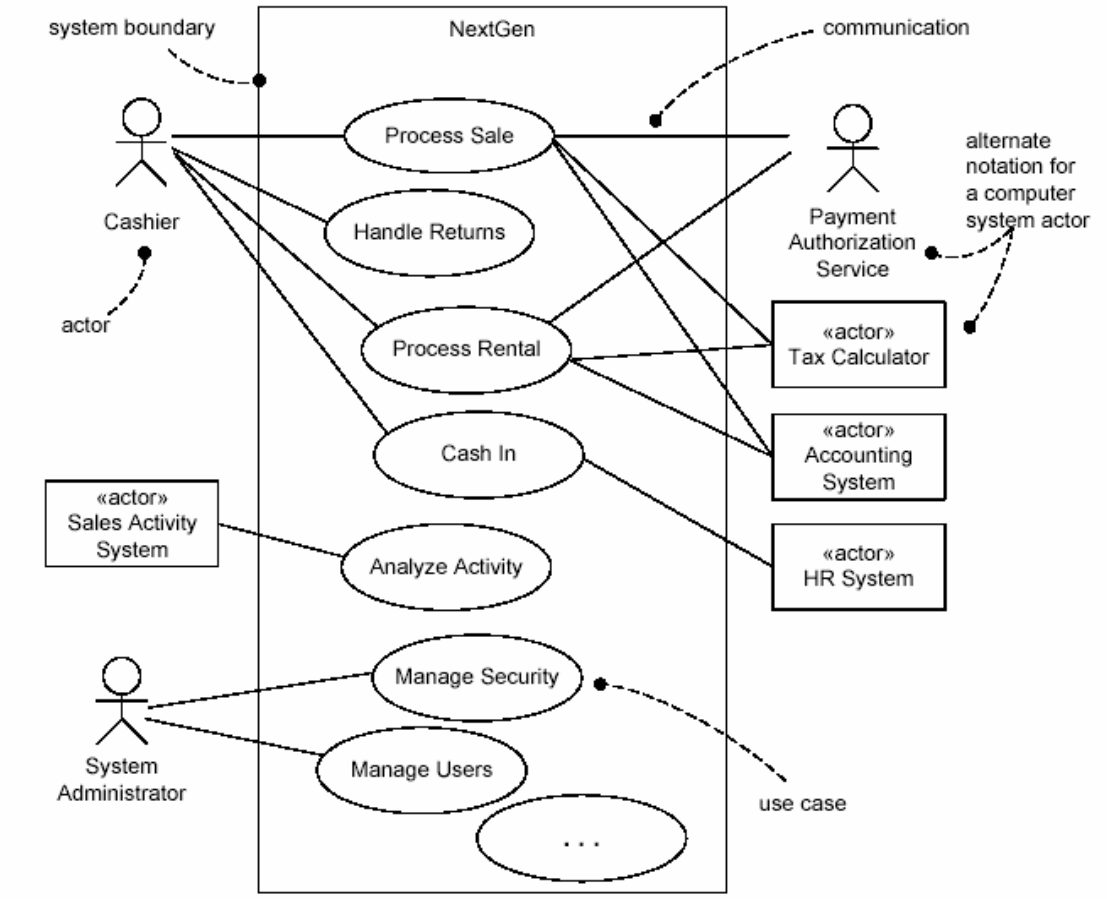


圖 6.2 使用案例概圖的其中一部分

畫使用案例圖與找出使用案例之間的關係是使用案例工作中的次要工作。因爲使用案例是由文字構成的文件，所以進行使用案例工作就是要寫出這些說明文字。

建立使用案例模型的新手（或學生）常發生的一種情況就是：把精神放在使用案例圖與使用案例之間的關係，而不是寫說明文字。世界級的使用案例專家，例如 Anderson 、 Fowler 、 Cockburn 等等都不是很重視使用案例圖與使用案例之間的關係，而把焦點放在寫說明文字上。把這樣的建言記在心理，簡單的使用案例圖反而可以提供系統一個簡潔、用視覺化方式呈現的概圖，以展示相關的外部參與者以及它們使用系統的方式。

建議

畫一個簡單的使用案例圖，並且把它跟參與者—目標清單關聯在一起。

使用案例圖是描繪系統情境很好的一種概觀圖；換言之，它是很好的**概圖**（context diagram），可以顯示系統邊界、什麼在系統外面、如何使用系統。我們可以把它當成一種溝通工具，裡面彙總系統與參與者的行爲。圖 6.2 是 NextGen 系統的部分使用案例概圖。

畫使用案例圖時的一些建議

圖 6.3 裡面有一些畫圖的建議。請注意在參與者方塊中有加上 （（actor）） 符

號。這樣的符號稱爲 UML 的**造型**（stereotype）；我們可以用這種機制根據某種方式把某些元素分類。造型的名稱會用雙角括號框起來—這個括號是*單*一字元（而不是「((」與「))」），最常被用在法式印刷體中當引號用。

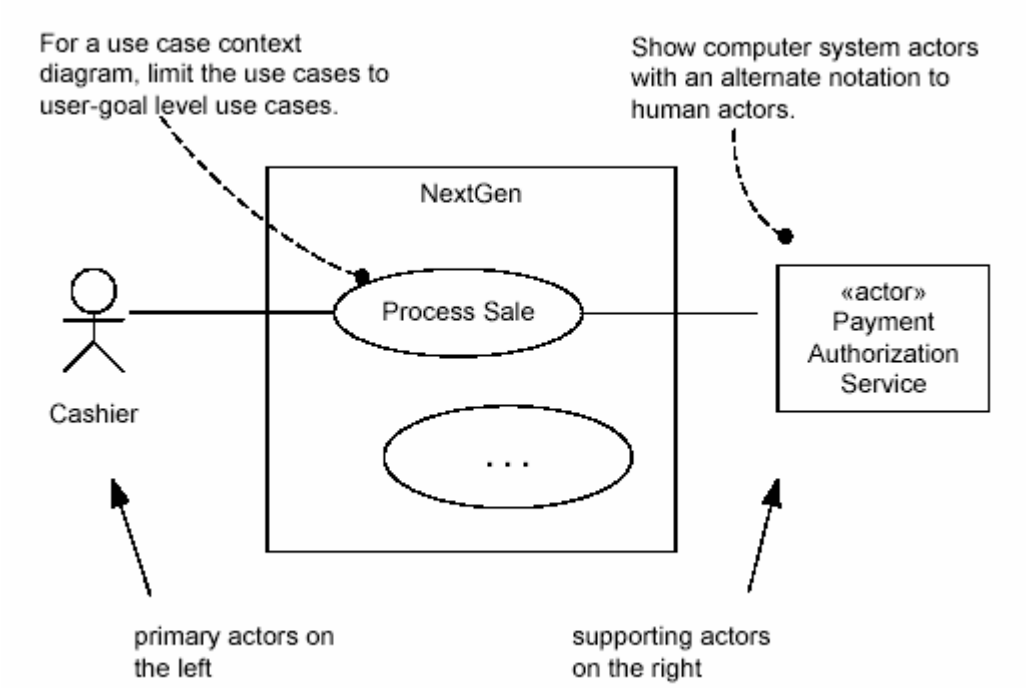


圖 6.3 對表示法的一些建議

爲了釐清不同概念，有些人偏好用其它表示法以強調外部電腦系統，如圖 6.4 所示。

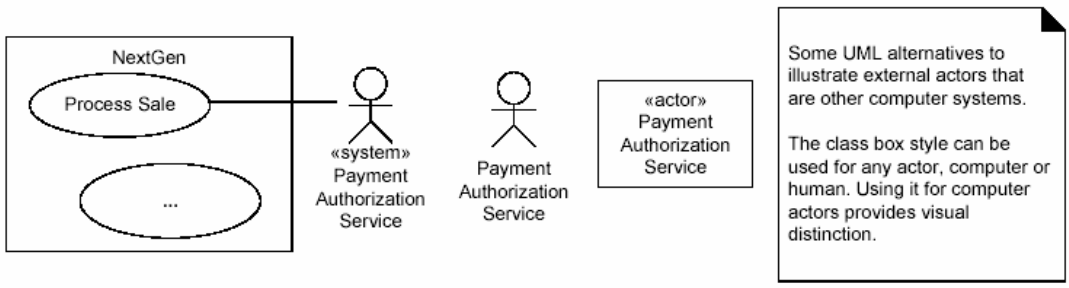


圖 6.4 參與者的其它種表示法

注意不要畫太多的圖

我們再強調一次，進行使用案例的工作中最重要的事情是要寫出說明文字，而不是使用案例圖或是使用案例之間的關係。如果有一個組織花許多小時（或者更糟的情況，花好幾天）畫使用案例圖並討論使用案例之間的關係，而不是把焦點放在寫出說明文字，那麼這些工作量就被花在不對的地方。

第十四節根據情境寫成的需求（使用案例）與低階的系統特性清單

就像 Use Cases: Requirements in Context 【GK00】這本書的書名所隱含的一樣，使用案例概念的關鍵動機是在目標與使用系統的情境下考慮並組織需求。這樣做是件好事—可以加強內聚力與理解性。然而，使用案例不是唯一必要的工作成果。某些非功能需求、領域規則與情境，以及其它很難找到地方放的元素都很適合放在輔助規格書中，我們會在下一章中解說。

使用案例後面所隱含的一個概念是用使用案例取代詳細、低階的系統特性清單（這份清單在傳統需求方法論中是很常見的），不過會有一些例外。這份清單很可能會看起來像下面一樣，用功能領域群組起來：

ID	系統特性
FEAT1.9	系統應該能夠讓我們輸入商品識別碼。
…	…
FEAT2.4	系統應該把信用付款記錄到會計應收系統中。

像這樣的低階系統特性清單是有些用處的。然而，完整的清單不只會有半頁；最大的可能性是數十頁到數百頁。這樣的結果會導致某些缺點，而使用案例則可以彌補它的不足，包括：

- 很長、詳細的功能清單無法把需求用一致的情境考慮；慢慢地，不同的功能與系統特性會像無關連的「雜貨清單」一樣越來越長。相對地，使用案例則把需求放在敘事情節與使用系統目標情境中考慮。
- 如果同時使用使用案例與詳細的系統特性清單，就會重複工作。反而要做更多工作、有更多工作要寫、要看，也會有更多的一致性問題與同步問題要考慮。

建議

努力試著用使用案例取代詳細、低階的系統特性清單。

高階系統特性清單是可接受的

用精簡、高階的系統特性清單彙總系統功能性是很常見、很有用的方式，我們通常把它放在專案願景文件中的系統特性小節。相對於 100 頁、低階、詳細的系統特性，我們的系統特性清單通常只有數十個項目。這份清單對系統功能性提供很簡潔的摘要，而且跟使用案例觀點無關。例如：

系統特性摘要

- 捕捉銷售
- 付款授權（信用、借貸或支票）
- 使用者、安全性、程式碼與常數表格等等的系統管理
- 當外部元件失效時，自動、離線的銷售處理
- 以業界標準爲基礎、使用協力廠商系統，包含庫存、會計、人力資源、稅金計算器與付款授權服務的即時交易
- 在處理事情的情節中，某些固定、常見的地方可以定義並客製化、加入「可嵌入」的企業規則
- …

我們會在下一章探討這份系統特性清單。

什麼時候用詳細的系統清單是適當的？

有時候使用案例並不是真的很適用；某些應用程式需要用系統特性驅動的觀點。舉例來說，應用程式伺服器、資料庫產品與其它中介系統或後台系統主要需要用系統特性來考慮或演化系統（例如「我們需要在下個發行版本支援 XML」。）使用案例本質上並不適合這類應用程式或者需要隨著市場力量而改版的系統。

第十五節使用案例並不是物件導向專用的

使用案例裡面並沒有什麼東西跟物件導向有關的；寫使用案例並不是在做物件導向分析。雖然這不是缺點，不過還是我們需要澄清這一點。事實上，使用案例是一種可以廣泛應用的需求分析工具，我們可以把它應用在非物件導向專案中，這點增加使用案例的可用性，讓它成爲一種捕捉需求的方法。然而，就像我們曾經討論過的，使用案例是典型物件導向分析與設計開發活動中很關鍵的資訊輸入來源。

第十六節UP 中的使用案例

使用案例在 UP 中是不可或缺、居核心地位的，它鼓勵我們採用**使用案例驅動的開發方式**（use-case driven development）。這隱含：

- 我們主要把需求記錄在使用案例中（使用案例模型）；其它需求技術（例如功能清單），則是次要的記錄需求工具。
- 使用案例是反覆式規劃方式中很重要的一部份。反覆裡面的工作「一部分」會由某些使用案例情節或整個使用案例定義。而且使用案例是評估值的關鍵資訊來源。
- **使用案例實現**（use case realization）會驅動設計工作。換言之，開發團隊會設計物件與子系統內的合作情形以執行或實現使用案例。

■ 使用案例通常會影響到使用者手冊的編排方式。

UP 中把系統使用案例與企業使用案例分開看。**系統使用案例**（system use case）是本章所討論的，例如*處理銷售*。我們在需求工作科目中產生系統使用案例，它屬於使用案例模型中的一部份。

企業使用案例（business use case）通常比較少寫。如果有的話，我們通常是在建立企業模型工作科目中產生企業使用案例，並且把它當成大規模企業做流程重整的其中一部份工作或者幫我們了解企業中新系統的情境。它們把企業中一連串動作當成一整塊東西描述以滿足**企業參與者**（business actor）（企業環境中的參與者，例如顧客或供應商）的目標。舉例來說，在餐廳中，*服務餐點*是其中一個企業使用案例。

橫跨反覆的使用案例與需求規格

本節重述 UP 與反覆式開發方式中的一個關鍵概念：需求規格所需花費的時間與工作量可能會橫跨數個反覆。表 6.1 只是一個範例（不是固定不變的方式），裡面說明 UP 如何發展需求的策略。

請注意，開始系統建構跟完全掌握需求之間有一段很長的時間差，當技術團隊開始建構系統產品核心時，可能只有 10% 的需求是有寫出細節的。事實上，一直到接近詳述階段第一個反覆結束之前，我們都可能缺乏有一致性的需求。

這是反覆式開發方式與瀑布式開發流程之間最主要的差異點：很早就已經開始具有產品品質的系統核心開發工作，時間遠遠在我們知道所有需求之前。

工作科目	工作成果	需求所花工作量的多少與相關說明				
		初始階段 1 星期	詳述階段 1 4 星期	詳述階段 2 4 星期	詳述階段 3 3 星期	詳述階段 4 3 星期
需求	使用案例模型	2 天的需求討論會。找出大部分使用案例的名稱，並且用一個短的段落彙總它們。只有 10% 有寫出細節。	接近這個反覆結尾時，會進行 2 天的需求討論會。從實作工作中對需求有更深入的了解並獲得回饋。然後完成 30 % 使用案例	接近這個反覆結尾時，會進行 2 天的需求討論會。從實作工作中對需求有更深入的了解並獲得回饋。然後完成 50 % 使用案例	跟前面的反覆一樣，完成 70 % 使用案例的細節。	跟前面的反覆一樣，完成 80-90 % 使用案例的細節。只有一部份使用案例會在詳述階段完成，其餘部分在建構階段完成。

			的細節。	的細節。		
設計	設計模型	無	針對一小部份有高風險、對架構有顯著影響的需求進行設計工作。	重複前面工作。	重複前面工作。	重複前面工作。現在高風險、對架構有顯著影響的設計部分應該已經很穩定了。
實作	實作模型（程式碼等等）	無	實作這些使用案例	重複進行實作工作，約完成最終系統的 5％。	重複進行實作工作，約完成最終系統的 10％。	重複進行實作工作，約完成最終系統的 15％。
專案管理	軟體開發計劃	對總工作量有非常含糊的估計值。	開始慢慢成形估計值。	更好一點…	更好一點…	現在可以很合理提出整個專案的長度、主要開發里程碑、工作量與成本的估計值。

表 6.1 我們在橫跨早期數個反覆的需求上可能花費的工作量；這並不是固定不變的。

請觀察表 6.1 中接近詳述階段第一個反覆的地方，此時會進行第二次需求討論會，以討論出約 30% 的使用案例細節。這種緩慢漸進的需求分析方式會從建好一些核心軟體的經驗回饋中學到一些東西。可能的回饋包括使用者的評量、測試以及對「系統該有什麼東西知道得更多。」換言之，快速建立軟體的動作讓需要澄清的假設與問題浮現出來。

產生 UP 工作成果的時間點

表 6.2 中展現一些 UP 工作成果，裡面還包括這些工作成果初版與修正版的時程。使用案例模型是在初始階段開始的，此時可能只有 10% 的使用案例已經寫出細節。隨著詳述階段的反覆進行，我們會慢慢寫出大部分的使用案例細節。所以詳述階段結尾時，我們會寫出大部分的使用案例細節與其它需求（在輔助規格書中），以這些東西爲基礎，我們可以很真實估算出到專案結束所需花費工作量

的估計值。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	領域模型		s		
需求	**使用案例模型(SSD)**	s	r		
	專案願景	s	r		
	輔助規格書	s	r		
	字彙表	s	r		
設計	設計模型		s	R	
實作	軟體架構文件		s		
	資料模型		s	R	
	實作模型		s	R	r
專案管理	軟體開發計畫	s	r	R	r
測試	測試模型		s	R	
環境	開發案例	s	r		

表 6.2 UP 中可能所需要用到的工作成果與產生這些工作成果的時間點。 s — 初版；r — 修正版

初始階段中的使用案例

接下來的討論是延續表 6.1 中的一些資訊。

在初始階段中，並不是所有使用案例都會有正式格式的細節。相反地，我們假設在早期只花 2 天需求討論會對 NextGen 進行調查工作。第一天，一開始時，我們會先花時間找出目標與關係人，並且很投機地預測什麼東西會在專案範圍之內、什麼不會。我們會寫出參與者—目標—使用案例關係表並且用電腦投影機展現出來。開始畫使用案例概圖。幾個小時之後，大概會找出 20 個使用者目標（而且也會找出相同的使用者目標層級使用案例），其中包括*處理銷售*、*處理退貨*等等。大部分我們有興趣、複雜、或有風險性的使用案例都會用簡式的格式寫下來。平均每個使用案例要花兩分鐘寫完。開發團隊開始構思系統功能性的高階概圖。之後，約有 10% 到 20% 代表核心複雜功能或某些方面特別有風險性的使用案例會用正式格式寫出來；然後開發團隊調查一小部份、有興趣的使用案例，以對專案的大小、複雜度與隱藏陷阱有更深入、更多了解。或許就是這兩個使用案例：*處理銷售*、*處理退貨*。

我們用整合了文字處理器的需求管理工具寫出需求，並且當開發團隊一起合作進行分析與寫需求時，用投影機顯示出來。爲了發掘更細微的（或可能最花成本的）功能需求與關鍵非功能需求—或系統品質（例如可靠性與產能）—，我們寫出這些使用案例的*關係人與利益*清單。

分析的目標不是要完整寫出所有使用案例，而是想花幾個小時對專案有更深入了解。

專案贊助者需要決定專案是否值得進行大量的調查工作（換言之，就是進行詳述階段。）初始階段的工作不是要進行調查工作，而是要對專案範圍、風險、工作量、技術可行性與企業案例有低準確度的了解，以決定是否要往前走、如果要做的話從哪個部分開始做、或者終止專案。

NextGen 專案的初始階段大概會進行五天。我們在這個星期中會進行爲期 2 天的需求討論會、簡短的使用案例分析與其它調查工作，最後決定這個系統要進行詳述階段。

詳述階段中的使用案例

接下來的討論是延續表 6.1 中的一些資訊。

這個開發階段中包含很多有時間長度限制的反覆（例如四個反覆）。 在這些反覆中，我們慢慢建構系統中有風險性、高價值或顯著影響到架構的部分，並且找出、澄清「大部分」需求。從實際寫程式所得到的回饋會影響並增加開發團隊對系統的了解，需求是不斷慢慢調整出來的。每個反覆中或許會有一個爲期 2 天長的需求討論會，而反覆則長達四個星期。然而，我們不會在討論會中調查所有使用案例。使用案例是有先後順序的；早期的討論會中會把焦點放在一部份最重要的使用案例上。

接下來每次時間不長的討論會讓我們有機會調整並修正核心需求願景，在早期反覆中它還不是很穩定，直到後面一點的反覆中才會穩定下來。因此，我們會不斷穿插做需求發掘與建構部分軟體的工作。

在每次需求討論會中，我們會修正使用者目標與使用案例清單，寫出更多的使用案例或用正式格式重寫使用案例。在詳述階段結尾時，我們會寫出 「80-90%」的使用案例細節。在 POS 系統 20 個使用者目標層級的使用案例中， 15 個或更多個最複雜、風險性最高的使用案例會被調查過、用正式格式寫出來或重寫過。

請注意，詳述階段中會進行部分系統的程式設計工作。在詳述階段結尾時，NextGen 開發團隊不但會對使用案例的定義有更深入了解，還會有可執行的軟體。

建構階段中的使用案例

建構階段中包括幾個有時間長度限制的反覆（例如 20 個長達 2 個星期的反

覆。）一旦我們在詳述階段中處理好有風險性、核心不穩定的議題之後，我們會在建構階段中把焦點放在完成系統上。這時候可能還需要寫出一些次要使用案例的細節，也有可能還需要進行需求討論會，不過跟詳述階段比起來，工作量已經少很多了。在這個開發階段中，大部分核心功能與非功能需求經過不斷調整後，應該已經穩定下來。這不代表需求不會改變或我們已經完成整個調查工作，不過變動幅度應該變小得多。

第十七節個案研究：NextGen 系統初始階段中的使用案例

跟我們在前面一節中說的一樣， 初始階段中不會用正式格式寫出所有使用案例的細節。在目前開發階段中，這個個案研究的使用案例模型應該包含下面內容：

正式的	非正式的	簡式
處理銷售 處理退貨	處理出租 分析銷售活動 管理安全性	取出現金 存入現金 管理使用者 管理系統表格

第十八節進階讀物

最受歡迎的一本使用案例指引是*使用案例最佳實務－寫作指南、秘訣與範本*（Writing Effective Use Cases）【Cockburn01】【註】，它已經被翻譯成多國語言。因爲這本書是最多人看過、被大家遵循的一本使用案例書，所以我們建議把它當成使用案例的主要參考書。本章對使用案例所做的簡介主要是以這本書的內容爲基礎，而且跟裡面的內容一致。建議：不要因爲這本書作者用不同圖示代表不同使用案例層級，而且書中很早就強調使用案例層級與使用案例分類法，就遲遲不看這本書。這些圖示都是可選用的，也不是那麼重要。而且雖然對使用案例不熟悉的人來說，一開始就看使用案例層級與目標的相關討論會分散注意力，不過對使用使用案例一段時間的人來說，大家都很讚賞這種區分使用案例層級與範圍的概念，認爲這是使用案例中很關鍵、實務的議題，因爲當我們建立使用案例模型時，對使用案例層級與範圍的不了解是造成模型變得很複雜的原因。

【註】：請注意， Cockburn 這個名字跟 slow burn 這個字押韻，代表他不是產量很大的作者。

「Structuring Use Cases with Goals」【Cockburn97】是使用案例方面被最廣泛引用的一篇論文，你可以在 www.usecases.org 中找到。

Use Cases: Requirements in Context 【GK00】是另一本很有用的教科書。裡面強

調一個很重要的觀點—如同書名所說的—使用案例不只是另一種捕捉需求的工作成果，它是驅動需求工作與需求資訊的核心主軸。
另一本值得讀的書是 Applying Use Cases: A Practical Guilde 【SW98】，作者是很有經驗的使用案例老師與實踐者，書中告訴我們如何在反覆式生命週期中應用使用案例。

第十九節UP 中跟使用案例有關的工作成果與流程情境

如圖 6.5 所示，使用案例會影響到許多 UP 中的工作成果。

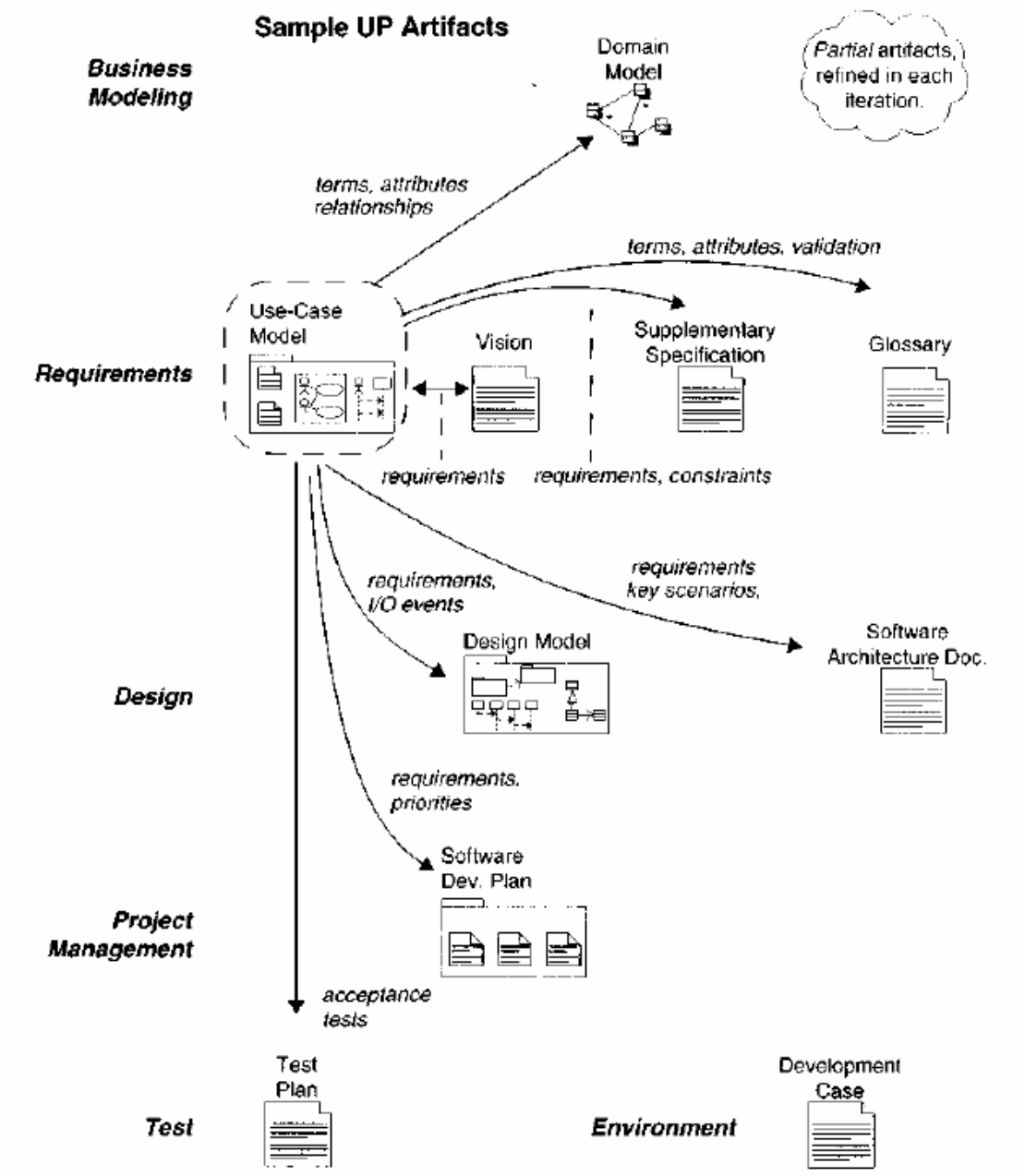


圖 6.5 UP 中可能會被使用案例影響到的工作成果。

在 UP 中，使用案例工作是需求工作科目中的一個開發活動，我們應該在需求討論會中開始這項工作。圖 6.5 提供進行這項工作的可能時間點與所需會議空間。

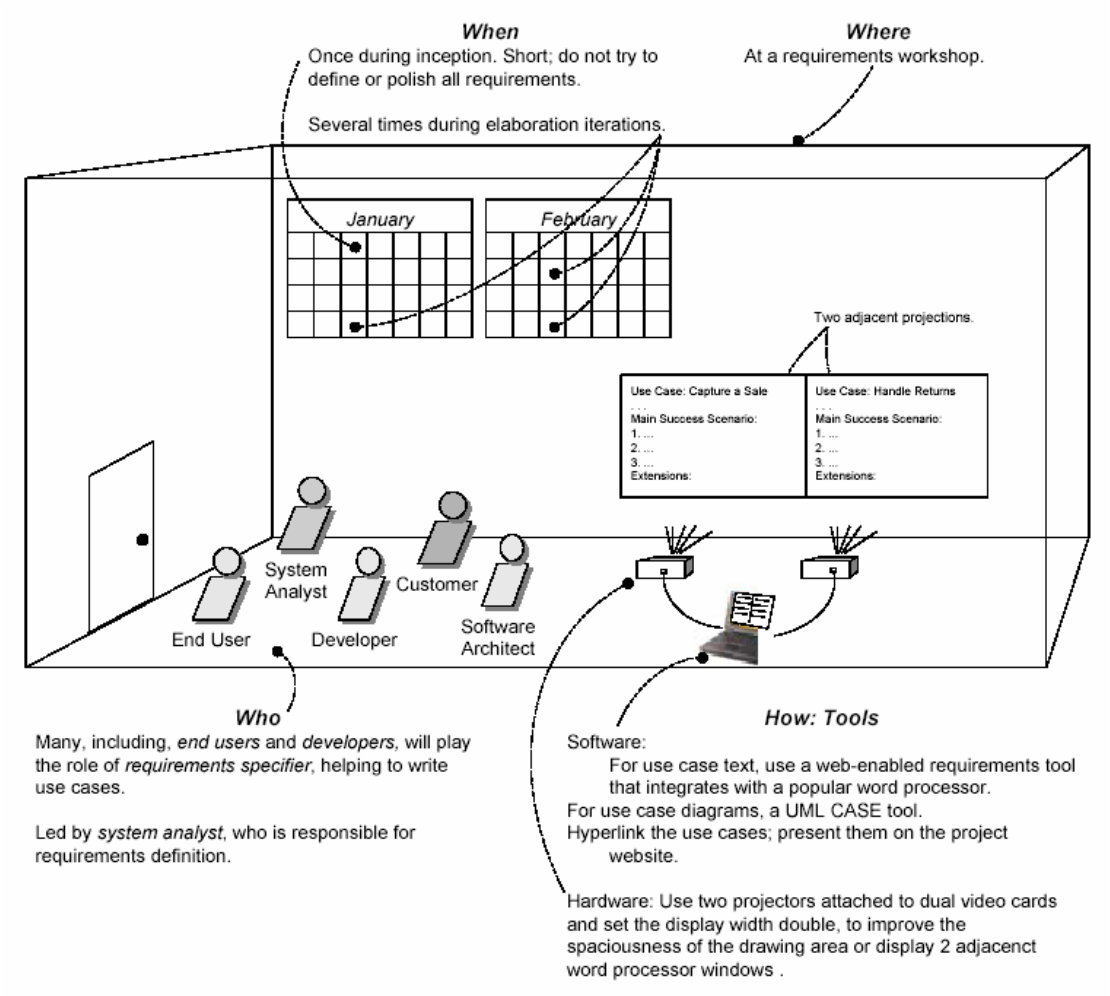


圖 6.6 開發流程與情境的設定

第七章找出其它需求

當理想失敗時，閒言閒語隨之而來。
— Johann Wolfgang von Goethe

本章目標

- 寫出輔助規格書（supplementary specification）、字彙表與專案願景。
- 比較並區分系統特性（system feature）與使用案例的不同。
- 建立專案願景與其它工作成果間的關係，以及專案願景與反覆式開發方式間的關係。
- 定義品質屬性。

簡介

只寫出使用案例是不夠的。我們需要找出其它需求，例如使用說明（documentation）、包裝方式（packaging）、可支援性、授權方式等等。這些需求都會被放在**輔助規格書**（supplementary specification）中。
字彙表（glossary）中則放一些術語與相關定義；它也可以扮演資料字典的角色。
專案願景（vision）中則彙總專案的「願景。」我們可以利用它很簡潔地告訴大家(1).爲何提出專案、(2).要解決的問題是什麼、(3).誰是關係人、(4).關係人需要什麼、(5).被提出來的解決方案爲何。
引述：

> 專案願景中利用要開發產品、關係人的關鍵需要與系統特性的詳細描述定義出關係人的觀點。裡面包含想像核心需求大綱，作爲更詳細技術性需求的合約基礎【RUP】。

第一節NextGen POS 範例

後面這個範例的目的不是想完整呈現專案願景、字彙表或輔助規格書，因爲這些工作成果中有某些小節—雖然對專案很有用—跟我們的學習目標沒有關係【註】。本書的目標是物件設計、使用案例需求分析與物件導向分析中的核心技能，而不是 POS 相關的問題或專案願景描述。因此，我們只會簡單觸及某幾個節以連結前後相關工作、強調值得注意的議題、讓大家對內容有感覺以求討論地更快。
【註】: 慢慢擴大範圍不只是需求中會遇到的問題，也是寫需求時會遇到的問題。

第二節NextGen 範例：輔助規格書（部分）

輔助規格書

修訂歷史

版本	日期	說明	作者
初始階段草稿	2031 年 1 月 10 日	第一版草稿。 主要是爲了在詳述階段中修正用。	Craig Laman

簡介

這份文件存放 NextGen POS 需求中使用案例以外的所有其它需求。

功能性

（橫跨許多使用案例的常見功能性）

系統記錄與錯誤的處理方式

記錄所有錯誤到永續儲存體中。

可嵌入式企業規則

在某幾個使用案例（後面會定義出來）的情節中，我們可以在某個點或事件中，用任意一組規則客製化系統的功能性。

安全性

系統所有的使用行爲都需要經過使用者授權。

可用性

人性因素

顧客希望 POS 能用大型螢幕顯示，因此：

- 要能夠在一公尺外看到文字。
- 避免使用色盲無法辨別的顏色。

快速、很容易、避免錯誤發生的處理方式對銷售處理來說是最重要的，因爲顧客只想快點離開結帳櫃檯，如果很慢的話，顧客對於在這裡買東西（或對銷售員）會留下不好的印象。

收銀員通常只會看著顧客或商品，而不看電腦螢幕。因此，除了顯示文字之外，我們還要用聲音發出訊號與警告訊息。

可靠性

可復原性

如果無法使用外部服務（付款授權、會計系統等等）時，試著用本地端的解決方案解決問題（例如先儲存起來再傳送出去）以完成一筆銷售。這裡需要做更多的分析工作…

效能

如同人性因素所提到的，買東西的人希望很快完成銷售處理過程。其中一個潛在

的效能瓶頸是外部付款授權。我們的目標是希望 90% 的情況下，可以在一分鐘內完成授權。

可支援性

可調整性

NextGen POS 系統的不同顧客在處理銷售時，需要有自己獨特的企業規則與處理方式。因此，在情節中很多事先定義好的地方（例如當一筆新的銷售被啓動時、當我們新增一筆商品項目時），要能夠使用嵌入式企業規則。

可配置性

不同顧客會希望他們的 POS 系統有不同的網路配置方式，例如厚重型（thick）vs. 輕便型（thin）客戶端、兩層式（two-tire） vs. 多層式（n-tire）實體層等等。此外，顧客可能希望修改系統的配置方式，以符合他們不斷變動的企業與效能需要。因此，系統必須有某方面的可配置性以符合這些需要。這個部分需要做更多的分析工作以發掘更多細節、了解複雜度、達成這些目標所需的工作量。

實作限制

NextGen 的領導階層堅持要用 Java 技術的解決方案，認爲這種技術除了容易開發之外，還會改善長期的系統移植方式與可支援性。

要購買的元件

- 稅金計算器。它必須針對不同州的需要支援可嵌入式計算器。

免費、開放原始碼的元件

一般來說，我們推薦在這個專案中使用免費、以 Java 技術開發的開放原始碼元件。

雖然還不是很確定最後可能採用的設計方式與可能選擇的元件，我們仍然建議下面一些候選元件：

- 用 JLog logging 框架
- …

介面

值得注意的硬體與硬體介面

- 觸控式螢幕顯示器（對作業系統來說，它還是標準螢幕，而觸控時的手勢則變成滑鼠事件）
- 條碼雷射掃描器（通常會接上一個特殊鍵盤，而且掃描出來的結果也會當成按鍵動作）
- 列印收據的印表機
- 信用卡＼借貸卡讀取機（系統的第一版不提供）

軟體介面

針對大部分外部協力廠商系統（稅金計算器、會計、庫存等等），需要能嵌入各式各樣的系統或各種介面。

領域（企業）規則

ID	規則	可變性	來源

規則 1	信用付款方式所需的簽名。	我們始終需要購買者的「簽名」，不過最多再過兩年，顧客會希望用電子簽名裝置取得簽名，我們也預期在五年內，顧客會希望用美國法律現在已經承認的新的、有唯一性的數位碼「簽名。」	差不多是所有信用授權公司的政策
規則 2	賦稅規則。銷售需要加上稅金。請參考政府目前的規定細節。	變動性高。各級政府的稅法每年都會變動。	法律
規則 3	取消信用付款時只能付錢到購買者的信用帳戶，而不能付現金。	變動性低。	信用授權公司的政策
規則 4	購買者的折扣規則。例如： 雇員— 20% 折扣。 偏好顧客— 10% 折扣。 老顧客— 15% 折扣。	變動性高。每個零售商會使用不同的規則。	零售商的政策
規則 5	銷售（交易層級）的折扣規則。用在加稅前的總價。例子： 總價超過 $100 USD 時，超過的部分有 10% 折扣。 每個星期一有 5% 折扣。 今天早上 10 點到下午 3 點之前，所有銷售有 10% 折	變動性高。每個零售商會用不同的規則，他們可能會每天或每小時變動一次。	零售商的政策

	扣。 今天早上 9 點到 10 點之間，豆腐有 50% 折扣。		
規則 6	產品（明細項層級）折扣規則。例如：這個星期拖引機有 10% 折扣。 買 2 個蔬菜堡，1 個免費。	變動性高。每個零售商會用不同的規則，他們可能會每天或每小時變動一次。	零售商的政策

法律議題

如果可以解決某些開放原始碼元件的授權方式，讓我們銷售包含開放原始碼的產品，我們就推薦採用這些元件。

有興趣領域中的相關資訊

定價方式

除了在領域規則小節描述到的定價規則，請注意每項產品都會有一個*原價*，而且可能會有永久性的減價。如果有永久性的減價，那麼（折扣前的）產品價格就是這個減價。因爲會計與稅金的緣故，所以縱然有永久性減價，組織還是會維護原價。

信用付款與借貸付款的處理方式

付款授權服務批准一筆電子或借貸付款後，他們就必須負責付款給銷售者，而不是購買者要負責。因此，針對每筆付款，銷售者需要在他們的應收帳款裡面記錄授權服務還沒付的錢有多少。通常每天晚上授權服務將進行電子轉帳，把每天應該付給銷售者的錢扣掉交易費用後轉到他們的帳戶。

銷售稅金

銷售稅金的計算非常複雜，而且經常需要變動以回應各級政府的法律變動。因此，我們建議把稅金的計算交給協力廠商計算器軟體（可以選擇好幾個服務。）稅金可能要付給市政府、區政府、州政府與聯邦政府。有些商品可能沒有限制、對所有人來說都是免稅的，或者根據收據人（購買者或收取者）的身分決定是否免稅（例如農夫或小孩。）

商品識別碼：UPC、EAN、SKU、條碼與條碼讀取機

NextGen POS 系統需要支援各種商品識別碼綱目。 UPCs （Universal Product Codes）、EANs（European Article Numbering）與 SKUs（Stock Keepng Units）是三種常見的產品識別碼系統。Japanese Article Numbers （JANs）是 EAN 的另一種版本。

SKUs 則是完全由零售商任意定義的識別碼。

然而， UPCs 與 EANs 已經有標準元件。

www.adams1.com/pub/russadam/upccode.html 有這些標準元件的簡介。也可以到

www.uc-council.org 與 www.ean-int.org 獲得資料。

第三節評論：輔助規格書

我們會在**輔助規格書**（supplementary specification）中放進使用案例或字彙表中不容易描述的其它需求、資訊與限制，其中包含涵蓋整個系統的「URPS+」品質屬性或需求。請注意，這些需求之前可能（而且應該）先寫到使用案例中的*特殊需求*小節，然後稍後再整合到輔助規格書中。輔助規格書裡面的組成元素應該包括：

- FURPS+ 需求—功能性（functionality）、可用性（usability）、可靠性（reliabiltiy）、效能（performance）與可支援性（supportability）。
- 報告
- 硬體與軟體方面的限制（例如作業系統或網路系統等等）
- 開發限制（例如開發流程或開發工具）
- 其它設計與實作上的限制
- 國際化考量（單位、語言…）
- 授權方式與其它法律上的考量
- 包裝方式
- （技術的、安全的、品質的）標準
- 實體環境考量（例如溫度或震動）
- 操作上的考量（例如如何處理錯誤，或多久做一次備份？）
- 領域或企業規格
- 有興趣領域中的相關資訊（例如什麼是整個信用付款處理週期的處理方式？）

限制（constraint）並不代表行爲，而是設計或專案中其它種類的限制。它們也是需求的一部份，不過我們通常把它們稱爲「限制」是強調它們在限制方面的影響性。舉例來說：

一定要用 Oracle（我們跟他們有授權協定。）

一定要用 Linux（成本比較低。）

建議

（在不成熟的詳述階段做的）早期設計決策與限制幾乎都是不好的主意，所以我們應該懷疑並質疑它們，特別是在詳述階段，我們這時候還沒有小心分析這些決策與限制。有些限制會被強加上去是因爲某些無法避免的理由，例如法律限制或要呼叫的現存外部系統介面。

品質屬性

有些需求被稱爲系統的**品質屬性**（quality attribute）【BCK98】（或 xx「性」。）

其中包括可用性、可靠性等等。請注意，這些指的都是系統的屬性，並不一定是高品質的（品質在英文中已經被過度使用了。）舉例來說，如果產品不需要長期使用，可支援性的品質有可能是低的。

品質屬性有兩種型態：

1. 執行時可觀察的（功能性、可用性、可靠性、效能…）
2. 執行時無法觀察的（可支援性、可測試性…）

我們在使用案例中詳細說明功能性，也會在使用案例中說明跟使用案例相關的其它品質屬性（例如*處理銷售*使用案例的效能品質。）

其它涵蓋整個系統的 FURPS+ 品質屬性則放在輔助規格書中。

一般來說，功能性是一種有效的品質屬性，「品質屬性」這個術語通常隱含「功能性以外的系統品質。」我們這裡也是同樣的用法。不過，這不代表品質屬性完全等於非功能需求，後者代表除了功能性之外的*所有東西*，是一個意義更寬廣的術語（例如包裝方式、授權方式。）

當我們扮演「架構分析師」角色時，我們會特別對涵蓋整個系統的品質屬性有興趣（而輔助規格書則是記錄它們的地方）—就像我們將在第 32 章介紹的一樣—架構分析與架構設計中大部分工作是在功能需求情境下找出並解決品質屬性。舉例來說，假設 NextGen 系統的其中一個品質屬性是：遠端服務無法正常工作時的容錯能力。從架構觀點來看，這個品質屬性會影響到大範圍的設計決策。

品質屬性之間會有交互關係，也需要做一些取捨。如同在 POS 這個簡單的範例中，「具有可靠性（容錯）」與「容易測試」在某些方面是衝突的，因爲分散式系統中有許多小地方很可能會失效。

DO（企業）規則

領域規則【Ross97、GK00】中會規定領域或企業可能的操作方式。它們不是任何應用程式的需求，雖然應用程式的需求通常會受到領域規則影響。公司政策、實體法律與政府法律是常見的領域規則。

我們通常把這些東西稱爲**企業規則**（business rule），這是最常見的一種類領域規則，不過前者的含意比較狹窄，因爲有些軟體應用程式是針對非企業問題開發的，例如天氣模擬系統或軍事後勤系統。天氣模擬系統中有些是會被「領域規則」影響到的應用程式需求，這些需求跟實體法律與相關法律有關。

找出並記錄這些影響需求的領域規則通常很有用。一般來說，我們會把這些領域規則實現在使用案例中，因爲這些規則可以澄清不完整或含糊不清的使用案例內容。舉例來說，在 POS 系統中，如果有人問說是否*處理銷售*使用案例中應該寫出沒有簽名時的信用付款替代方案，企業規則（RULE1）就可以很清楚地告訴我們，信用授權公司是否允許這樣的信用付款方式。

警告

規則並不是應用程式需求。也不要把系統特性記成規則。規則中描述領域工作時

的限制與行爲，而不是應用程式的。

有興趣領域中的相關資訊

主題專家所寫出跟新軟體系統（銷售與會計、跟地底下油＼水＼天然氣相關的地球科學知識等等）相關的一些領域說明（或提供的 URLs）是很有價值的，因爲它們可以提供開發團隊一個適當的情境，讓他們對新軟體系統有更深入了解。這些說明裡面會告訴我們一些重要文獻或專家、公式、法條或其它參考資料。舉例來說，NextGen 的開發團隊必須對 UPC 與 EAN 編碼綱目、條碼符號學有某種程度了解。

第四節NextGen 範例：遠景（部分）

專案願景

修訂歷史

版本	日期	說明	作者
初始階段草稿	2031 年 1 月 10 日	第一版草稿。 主要是爲了在詳述階段中修正用。	Craig Laman

簡介

這裡的分析是爲了說明用而虛構的

我們想像下一代具有容錯能力的點銷售（POS）應用程式— NextGen POS ，它要能夠支援各種客戶的企業規則、多種終端設備與使用者介面機制，以及跟多種協力廠商的支援性系統整合在一起。

競爭位置

企業機會

現存的 POS 產品無法根據顧客企業調整，使用各種企業規則與網路設計方式（例如輕便型或厚重型； 2 、 3 或 4 層式架構。）此外，現有的產品無法隨著終端設備與企業發展而擴充。也沒有產品可以動態根據外部系統失效的情況，自動選擇連線或離線模式運作。也沒有產品可以很容易地跟許多協力廠商系統整合在一起。沒有產品可以連上新的終端技術設備，例如移動式 PDAs。在這些沒有彈性的地方，市場並沒有被滿足，需要符合這些彈性的 POS 滿足這個市場。

問題描述

傳統的 POS 系統缺乏彈性、沒有容錯能力，也很難跟協力廠商系統整合。上述這些缺點會造成一些問題：及時銷售處理時發生外部系統失效的問題；改良流程後，新流程跟現有軟體不合；缺乏衡量與規劃時所需的準確、及時會計與庫存資

料。收銀員、店經理、系統管理員與公司管理都會受到影響。

產品競爭位置描述

—用簡潔方式彙總說明：系統是爲誰而做的、系統的突出特性、系統跟競爭對手之間的差異。

替代方案與競爭對手...

關係人描述

了解誰是主角，以及他們的問題所在

市場人口統計資料...

關係人（非使用者）摘要...

使用者摘要...

關係人的關鍵、高階目標與問題

整合參與者與目標清單與使用案例中關係人的利益

由主題專家與其它關係人開爲期一天的需求討論會，並且在幾個零售商店進行問卷調查之後，找到下面的關鍵目標與問題：

高階目標	優先順序	問題與考量點	現有解決方案
快速、強固、整合過的銷售處理方式	高	負載增加時，處理速度會降低。 元件失效時，系統喪失處理銷售的能力。因爲跟現有的會計、庫存、人力資源系統之間沒有整合過，所以這些系統中的資訊都過時了。導致我們很難做衡量與規劃的工作。 無法根據獨特企業需求客製化企業規則。 新的終端設備或使用者介面型態很難連到系統上（例如移動式 PDAs。）	
…	…	…	…

使用者層級的目標...

這裡的資料或摘要可能是在建立使用案例模型時產生的參與者—目標清單

系統需要滿足這些使用者（與外部系統）的目標：

- 收銀員：處理銷售、處理退貨、存入現金、取出現金
- 系統管理員：管理使用者、管理安全性、管理系統表格
- 經理：啓動系統、關機
- 銷售活動系統：分析銷售資料
- …

使用者環境...

產品概觀

從使用案例圖所摘要來的概圖其格式、詳細程度可有不同，不過裡面都需要顯示出跟系統相關的外部參與者

產品前景

NextGen POS 系統未來通常是裝在店裡面的；如果使用移動式終端設備的話，我們會用商店網路在店面的裡面或外面等鄰近商店地點使用。系統會提供服務給使用者，並且跟其它系統一起合作，如圖 專案願景-1 所示。

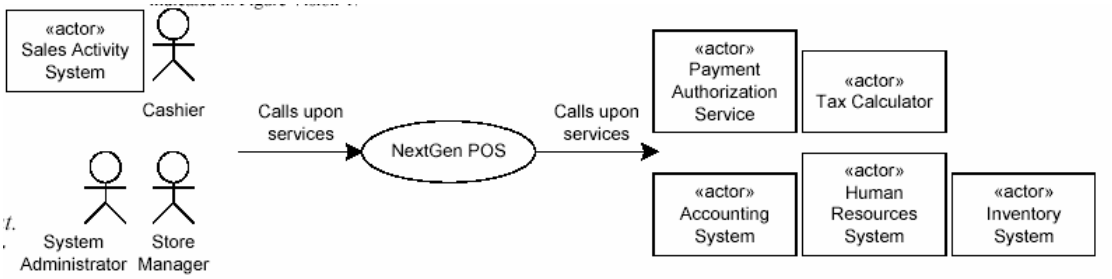


圖 專案願景-1 NextGen POS 系統的概圖

利益摘要

有支援的系統特性	關係人的利益
在功能性方面，系統將提供銷售組織所需要的所有常見服務，包括捕捉銷售、付款授權、退貨處理等等。	自動化、快速的點銷售服務。
自動偵測遠端服務失效，並且針對無效的服務，馬上轉到本地端、離線的處理方式。	當外部元件失效時，能夠繼續處理銷售。
在銷售處理情節中，有許多地方都可以用可嵌入式企業規則。	彈性的企業邏輯配置方式。
跟協力廠商系統之間，根據業界標準協定處理即時交易。	及時、精確的銷售、會計與庫存資訊，以支援衡量與規劃的工作。
…	…

跟參與者一目標清單很像，不過裡面用比使用案例還高階的方式，把目標、利益與解決方案放在一起

裡面還彙總了產品品質的差異與價值

假設與相關性...

成本與定價...

授權與安裝...

系統特性摘要

如下面所討論的，系統特性是格式簡潔的功能性摘要

- 捕捉銷售
- （信用、借貸或支票的）付款授權
- 使用者、安全性、程式碼、限制表格等等的系統管理工作
- 當外部元件失效時，自動離線的銷售處理方式
- 根據業界標準、跟協力廠商系統之間進行即時交易，外部系統包括庫存、會計、人力資源、稅金計算與付款授權服務
- 在處理情節的某些固定地方，我們可以定義並執行客製化、「可嵌入」的企業規則
- …

其它需求與限制

包含設計限制、可用性、可靠性、效能、可支援性、使用說明、包裝方式等等。請參見輔助規格書與使用案例。

第五節評論：專案遠景

我們解決同樣的問題嗎？是對的問題嗎？

問題描述

在初始階段早期需求工作中，關係人需要一起合作以定義出簡潔的問題描述，這樣可以避免關係人試著解決類似但有點差異的問題，而且我們也可以很快找出問題描述。有時候，我們定義問題描述時，可以把不同成員想達成目標中有本質上差異的地方顯示出來。

我們不用散文方式陳述問題描述，下面的表格是 RUP 所提供的問題描述範本：

問題爲	…
會影響哪些東西	…
某些東西所受的影響	…
成功的解決方案應該爲	…

關係人的關鍵高階目標與問題

表格中彙總高階的目標與問題，而不是工作階層的使用案例，並且會揭露某個使用案例或涵蓋數個使用案例的非功能目標與品質目標，例如：

- 我們需要具有容錯能力的銷售處理方式。
- 我們需要能客製化的企業規則。

什麼是根本的問題與目標？

關係人表達他們的目標時，常常把他們所想像的解決方案講出來，例如：「我們需要全職的程式設計師，當我們想改變企業規則時可以加以客製化。」有時候關係人講的解決方案是可以理解的，因爲他們非常了解問題領域與可能有的選擇。不過有時候關係人的解決方案並不是最適當的，或者沒有解決底層的主要問題根源。

因此，系統分析師需要調查問題與目標鏈－就像前一章討論使用案例與目標一樣－了解根本問題、問題的相對重要性與影響性，以決定優先順序，並且花最多精神、用最好的解決方案來解決問題。

協助群組產生概念的方法論

雖然有點超出討論範圍，不過我們製作一些開發活動時這裡所談到的特別有用，例如定義高階問題與找出目標這類型創造性、調查性的群組工作。有幾種協助群組技術可以幫大家發掘根本問題與目標、產生概念、訂出優先順序，包括：心靈地圖方式（mind mapping）、魚骨圖（fishbone diagram）；柏拉圖（pareto diagram）、腦力激盪（brainstorming）、多次投票權選舉（multi-voting）、陣營選舉（dot voting）、名目團體程序（nominal group process）、腦力書寫（想到就寫）（brainwriting）、分類群組（affinity grouping）。請上網查看這些方法。我喜歡在同一個討論會中同時用多種技術，從不同角度發掘常見問題。

系統特性—功能需求

使用案例並不是我們表達功能需求的唯一方式，原因如下：

- 功能需求太詳細了。關係人通常希望一份紀錄最有價值功能的簡短摘要。
- 那麼簡單列出使用案例名稱（處理銷售、處理退貨等等）來彙總功能性如何？第一點、這份清單還是太長。此外，名稱可能無法表現出關係人真正有興趣的功能性；換言之，只列出名稱這樣的粗細程度可能會隱藏值得注意的功能。舉例來說，自動付款授權功能性的說明是在*處理銷售*使用案例裡面。只讀使用案例名稱清單還是無法知道系統是否會做付款授權（譯註：這裡說明沒有細節也是不行的）。此外，（爲了簡短起見）有人可能想把一組使用案例組合成一個系統特性，例如*包含使用者、安全性、程式碼、常數表格等等的系統管理*。
- 有些值得注意的功能性本質上就適合用簡短句子表達，無法把它們對應到使用案例名稱或基本企業流程層級目標。這樣的功能性可能會橫跨幾個使用案例或者跟使用案例的構面垂直。舉例來說，在 NextGen 系統的第一次需求討論會中，有人可能說「系統應該可以跟現存的協力廠商會計、庫存與稅金計算系統一起合作、進行交易。」這個功能性描述並無法用某個特定使用案例代表，不過卻可以用系統特性以很適當、精簡的方式表達、紀錄與溝通。
  - 針對最後一點來說，有些應用程式需要用系統特性說明主要功能性，使用案例反而不是那麼合適。對中間層產品（例如應用程式伺服器）來說，大家不是那麼想用使用案例。假設開發團隊正在考慮下一個發行版本。討論需求時，人們（例如行銷人員）可能會說：「下一版需要支援 EJB 2.0 entity bean。」這樣的需求主要是用系統特性清單表達的而不是用使用案例。

因此，表達系統功能的另一種方式或輔助方式就是用**系統特性**（system feature）說明情境，系統特性是彙總系統功能的高階、精簡描述。更正式來說，在 UP 裡面，系統特性是「由系統提供、直接滿足關係人需要的外部、看得見的（系統）

服務」【Kruchten00】。

系統特性代表系統能做的東西。它們應該要能夠用這樣的句子表達： *系統應該能做＜系統特性 X＞*

舉例來說：

系統應該能做付款授權

回想一下，我們把專案願景當成開發者與企業之間正式或非正式的合約。系統特性就是我們可以拿來彙總合約中所記載系統該*做*的事。系統特性很簡潔，我們可以用它彌補使用案例的不足。

系統特性跟很多非功能需求或限制不同，例如：「*系統必須在 Linux 上跑、必須是 24/7 無休運作，也必須有觸控式螢幕介面。*」請注意，這些說明都不能用前面的句子「*系統應該能做＜系統特性 X＞*」表達。

有時候，「外部、看得見的（系統）服務…」很難讓我們決定某個東西是不是系統特性。例如，下面句子所描述的是不是系統特性：

系統應該可以跟現存的協力廠商會計、庫存與稅金計算系統合作進行交易。

裡面是描述一種行爲，而且可能值得關係人注意，不過合作本身並不一定是外界看得見的東西，要看你的時間片段、從多近的角度看、從哪裡看而定。姑且先把它當成系統特性吧－我們通常不需要爲太細的分類問題傷腦筋。

最後，請注意大部分系統特性都可以在使用案例的說明文字中找到詳細的描述。

表示法與組織方式

首先、最重要的一點，系統特性必須是簡短、高階的描述。我們應該能夠很快地閱讀系統特性清單。

雖然我們通常可以用「*系統應該能做...*」表達系統特性，不過沒有必要說我們一定要遵從「*系統應該能做...*」或其它表示法。

下面是一個多系統專案的高階特性範例， POS 只是其中一個組成元素而已：

主要系統特性包括：

- *POS 服務*
- *庫存管理*
- *以網頁爲基礎的購物方式*
- *...*

我們通常會用兩層式的系統特性來組織它們。在專案願景中用超過兩層以上的系統特性可能會有太多細節；在專案願景中的系統特性是爲了彙總功能性，而不是把它們分解成一長串太細的組成元素。下面是具有合理細節的系統特性：

主要系統特性包括：

- *POS 服務*
  - *捕捉銷售*
  - *付款授權*
  - *...*

- *庫存管理*
  - *自動重新補貨*
  - ...

有時候，第二層的系統特性相當於使用案例名稱（或使用者層級目標），不過不一定如此；系統特性是另一種彙總功能性的方式。然而，大部分系統特性都可以在使用案例中找到細節說明。

專案願景中應該放多少系統特性呢？

建議

少於 50 個系統特性的專案願景是我們想要的。如果包含更多的系統特性，考慮把它們群組起來或者把它們抽象化。

專案願景中的其它需求

在專案願景裡面，我們利用系統特性彙總描述在使用案例中的詳細功能需求。同樣地，專案願景裡面也可以彙總使用案例中特殊需求小節中或輔助規格書（SS）中的其它需求。然而，這樣做可能有造成不必要重複的風險。例如在 RUP 產品所提供的範本中，專案願景與輔助規格書裡面都針對其它需求（例如可用性、可靠性、效能等等）提供相同或類似的小節。像這樣的重複，不可避免地會造成維護上的惡夢。此外，在專案願景與輔助規格書中，相仿的小節（例如效能）之間爲了夠相似，它們的明細程度也需要差不多；換言之，其它需求描述的明細程度如果分別是「必要的」與「詳細的」，那麼兩者之間就會比較接近。

建議

針對其它需求，避免專案願景與輔助規格書（SS）－甚至包括使用案例－中的描述盡量避免重複。我們最好只把它們記錄在輔助規格書或使用案例中（如果使用案例中可以詳細說明的話。）然後在專案願景中，再引導讀者參考輔助規格書或使用案例中的其它需求小節。

上面這種做法可以降低寫文件的複雜度，不過這跟標準 RUP 範本中的寫文件方式有一點細微差異。如果你喜好採用標準範本的話，也是可以的。

專案願景、系統特性或使用案例—先寫什麼？

堅持某些工作成果的寫作順序對事情並沒有什麼幫助。而同時產生不同的需求工作成果反而可以產生縱效，你可能在寫某一份需求工作成果時，澄清另一份需求中不清楚的地方。然而，我們還是建議用下面順序寫這些工作成果：

1. 先寫出專案願景很簡短的第一版草稿。

2. 找出使用者目標與支援性使用案例。
3. 寫出一些使用案例，並且開始寫輔助規格書。
4. 修正專案願景、從前面寫出的工作成果彙總出資訊。

第六節NextGen 範例：字彙表（中的一部分）

字彙表

修訂歷史

版本	日期	說明	作者
初始階段草稿	2031 年 1 月 10 日	第一版草稿。 主要是爲了在詳述階段中修正用。	Craig Laman

定義

術語	定義與相關資訊	別名
銷售商品項目	銷售的產品或服務	
付款授權	由外部付款授權服務負責認可，它們保證會付款給銷售者。	
付款授權請求	由一些元素組成，用電子方式傳送給授權服務。組成元素包括：商店 ID 、顧客帳戶號碼、金額與時間戳記。	
UPC	用來識別產品的 12 位數編碼。通常用條碼標示在產品上。細節請參見 http://www.uc-council.org 。	Universal Product Code
…	…	…

第七節評論：字彙表（資料字典）

最簡單的**字彙表**（glossary）格式是由值得注意的術語加上定義所組而成的清單。，我們常常會很驚訝地發現不同關係人使用某個術語（通常是技術性的或針對某個領域的）時，有一些小差異存在，我們必須釐清這些差異以降低溝通、不明確需求的問題。

建議

早一點開始建字彙表。我記得有一次跟模擬專家一起工作時，發現在團體成員中對「cell」這個字的意義竟然不是那麼明確、大家對它有不同解釋。

我們不是要記下所有可能用到的術語，不過需要記下一些意義不清楚、含糊，或者需要某種程度詳細說明的術語，例如某些格式的資訊或驗證規則。

把字彙表當成資料字典

在 UP 中，字彙表也扮演**資料字典**（data dictionary）的角色，資料字典裡面會記錄關於資料的資料－換言之，**超資料**（metadata）。在初始階段，字彙表應該只是記錄術語與相關說明的簡單文件而已。在詳述階段，我們把它擴充成資料字典。術語的屬性可能有：

- 別名
- 說明
- 格式（形態、長度、單位）
- 跟其它元素之間的關係
- 資料值的範圍
- 驗證規則

請注意，在字彙表中值的範圍與驗證規則會隱含系統行爲需求。

單位

正如 Martin Fowler 在 *Analysis Patterns* 【Fowler96】中所強調的，我們必須考慮應用程式中所用到的單位（幣別、度量等等），特別當軟體應用程式到達國際化階段時。舉例來說，我們希望能把 NextGen 系統賣給不同國家的許多顧客，此時*價格*就不只是一個數字而已（譯註：錢的單位也很重要。）爲了表達各種不同幣別，我們要使用 *Money* 或 *Currency* 單位。

合成術語

字彙表裡面不止可以記錄一些意義上不可分割的術語，例如「產品價格。」裡面也可以包含一些合成元素（譯註：合成術語），例如「銷售」（裡面可能包含其它元素，例如日期與地點），而且我們可能會用一些匿稱說明使用案例中參與者之間所傳輸的一整組資料。例如在*處理銷售*使用案例中，我們考慮像下面這樣的句子：

系統傳送付款授權請求到外部付款授權服務，要求付款認可。

「付款授權請求」是一組資料的暱稱，我們需要在字彙表中解釋它。

第八節可信賴的規格：你不懷疑嗎？

看到寫好的需求時，我們可能會相信自己已經了解、也定義好需求了，並且可以（在早期）用它做出可信賴的專案評估值與專案計畫。對不寫軟體的開發人員來說，這個幻象更真實；反而程式設計師可能從過去慘痛經驗知道這些東西是不可信賴的。這是本章一開始所引述歌德的那段話其中一部份動機（譯註：這句話是「當理想失敗時，閒言閒語隨之而來。」如果我們把寫好的需求當成理想，需求變動時就會招致很多人的怨言。）

當我們所建構的軟體已經通過使用者與關係人的驗收測試，也符合這些人的真正目標，這才是最真實的結果（通常要等到他們看到、評估、使用軟體後，真正目標才會浮現出來。）

寫出專案願景與輔助規格書是很有價值的，這樣做可以釐清第一版需求：想要哪些東西、使用這個產品的動機與記錄主要概念。不過它們不是－包含其它所有需求工作成果－可信賴的規格。只有寫出程式碼、測試它、得到回饋、繼續跟使用者與顧客密切合作、調整系統，才能真正命中目標。

上面的說法不是要你放棄分析與思考、直接產生程式碼，而是建議你把寫好的需求看得輕一點，並且要跟使用者持續－事實上應該每天－密切配合。

第九節專案網站中的線上工作成果

因爲這是一本書，這些範例與之前的使用案例看起來都是靜態、用紙張記錄的感覺。然而，這些東西都應該是數位化的工作成果、用線上方式記錄在專案專屬網站上。而且除了不是平面、靜態的文件外，它們可能還會有超連結，或者記錄在某種工具裡面，而不是記錄在文字處理器或試算表中。例如字彙表可能是記錄在資料庫的資料表格。

第十節初始階段中是否不需要用到太多 UML

初始階段的目的只是要蒐集足夠資訊以建立共同專案願景、決定繼續開發下去的可行性，以及專案是否值得繼續做詳述階段調查工作。在這種情形下，我們只會用到簡單的 UML 使用案例圖，而不會畫出太多圖。在初始階段，我們把比較多的重點放在了解基本系統範圍與 10% 的需求上，用文字格式表達它們。事實上，畫 UML 圖的事情大部分會出現在下個開發階段－詳述階段。

第十一節UP 中的其它需求工作成果

跟前一章使用案例一樣，表 7.1 裡面彙總了專案可能用到的工作成果與製作時

間點。所有需求工作成果都是在初始階段開始的，而且主要工作會持續進行、貫穿整個詳述階段。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	領域模型		s		
需求	**使用案例模型（SSD）**	s	r		
	專案願景	s	r		
	輔助規格書	s	r		
	字彙表	s	r		
設計	設計模型		s	R	
實作	軟體架構文件		s		
	資料模型		s	R	
	實作模型		s	R	r
專案管理	軟體開發計畫	s	r	R	r
測試	測試模型		s	R	
環境	開發案例	s	r		

表 7.1 UP 可能所需要的工作成果與產生這些工作成果的時間點。 s — 初版；r — 修正版

初始階段

我們不會在初始階段完成這些需求工作成果。事實上，只是開始寫這些工作成果而已。

關係人需要知道專案是否值得繼續進行調查工作；我們在詳述階段才做真正的調查工作而不是在初始階段。在初始階段，專案願景中會用一張表格彙總專案概念，以幫助關係人決定是否值得繼續下去、從哪裡開始調查工作。

因爲大部分需求工作都是在詳述階段做的，我們在初始階段中只會寫出一點輔助規格書、強調值得注意的品質屬性（例如當外部服務失效時， NextGen POS 系統可以復原）以揭露主要的風險與挑戰。

初始階段中的需求討論會是這些工作成果的主要資料來源，討論會中會明白考量到系統主題，而使用案例分析中則間接考量到系統主題。討論會中不會馬上寫出草稿或可閱讀的工作成果，不過之後會由系統分析師寫出來。

詳述階段

在詳述階段，根據慢慢建構部分系統時得到的回饋、系統調整，我們會在這幾個反覆的需求討論會中，修正「願景」與專案願景工作成果。

經過不斷進行的需求調查工作與反覆式開發工作，其它需求會越來越清楚，再記錄在輔助規格書中。輔助規格書中的品質屬性（例如可靠性）是形成核心架構的關鍵驅動力，我們會在詳述階段設計並寫出系統核心架構來。這些品質屬性同時也是關鍵風險因子，會影響我們決定哪些東西要在早期反覆中進行。舉例來說，我們會在詳述階段探索 NextGen POS 的品質需求：當外部元件失效時，客戶端的可復原性。

大部分開發團隊會在詳述階段找尋並詳細說明字彙表。

在詳述階段結尾時，我們應該會有使用案例、輔助規格書與專案願景，而且在合理情況下，準備交付的系統特性與其它需求應該穩定了。然而，輔助規格書與專案願景不是「簽名後」就固定不動的固定規格；可調整的－而不是僵硬的－規格是反覆式開發方式與 UP 的核心價值所在。

對「簽完名、固定不變」的一些觀念釐清：在詳述階段結尾時，跟關係人之間對於專案剩下要做的東西會有一個共識，而且對需求與時程做下承諾（可能用合約）是完全合理的。在某些時候（UP 中是在詳述階段結束時），我們需要對「做什麼、做多少、何時做」有一個值得信賴的概念。根據這個想法，我們對需求做出正式承諾是很正常、也是可預期的。此外，我們也需要有變動控制流程（UP 中的最佳實務經驗之一），這樣才能正式考量並核准需求變動，而不是雜亂無章、不去控制需求變動。

另一方面，下面的概念是對「簽完名、固定不變」所做的說明中所隱含的一些想法：

- 在反覆式開發方式與 UP 中，我們知道無論花多少努力在需求規格上，某些變動是不可避免的，也是可接受的。這個變動可能是對系統做最新、投機性的改良，讓系統擁有者取得競爭優勢，或者是因爲對系統有更深了解而做的變動。
- 讓關係人持續參與以評估系統、提供回饋與引導專案往他們真正要的方向走是反覆式開發方式的核心價值。相反地，要關係人簽下固定不變的需求後洗手不管，等待完成產品是沒有任何好處的，因爲這樣做通常得不到他們真正想要的東西。

建構階段

相對地，主要的需求－功能需求與其它需求－應該是穩定而不是固定不變的，它會在一些小變動之後穩定下來。因此，在這個開發階段中輔助規格書與專案願景

不太可能會有太多變動。

第十二節進階讀物

專案願景與輔助規格書這類文件不是新的發明。有許多專案用過、許多需求方面的書也都有說明過。不過，大部分這類書都隱含瀑布式態度，目標希望在一開始還沒進行-設計與實作之前，就先寫出有細節、很正確、保證過的需求。在這種情況下，雖然它們可以對文件可能應該有的小節或內容提供很好的建議，不過傳統式的說明其實沒有什麼幫助。

軟體架構方面的書都會討論需求分析中應用程式的品質屬性，因爲這些品質需求對架構設計有很大的影響力。其中一本書是 *Software Architecture in Practice* 【Beck98】。

在 *The Business Rule Book* 【Ross97】中對企業規則有詳細的介紹。這本書對企業規則的相關理論既廣又深，也有完整考量，不過書中所用的方法論並沒有跟其它現代需求技術（例如使用案例或反覆式開發方式）做很好的連結。

第十三節UP 中跟初始階段有關的工作成果與流程情境

圖 7.1 中展現對專案願景、輔助規格書與字彙表有影響的工作成果、或被影響的工作成果。

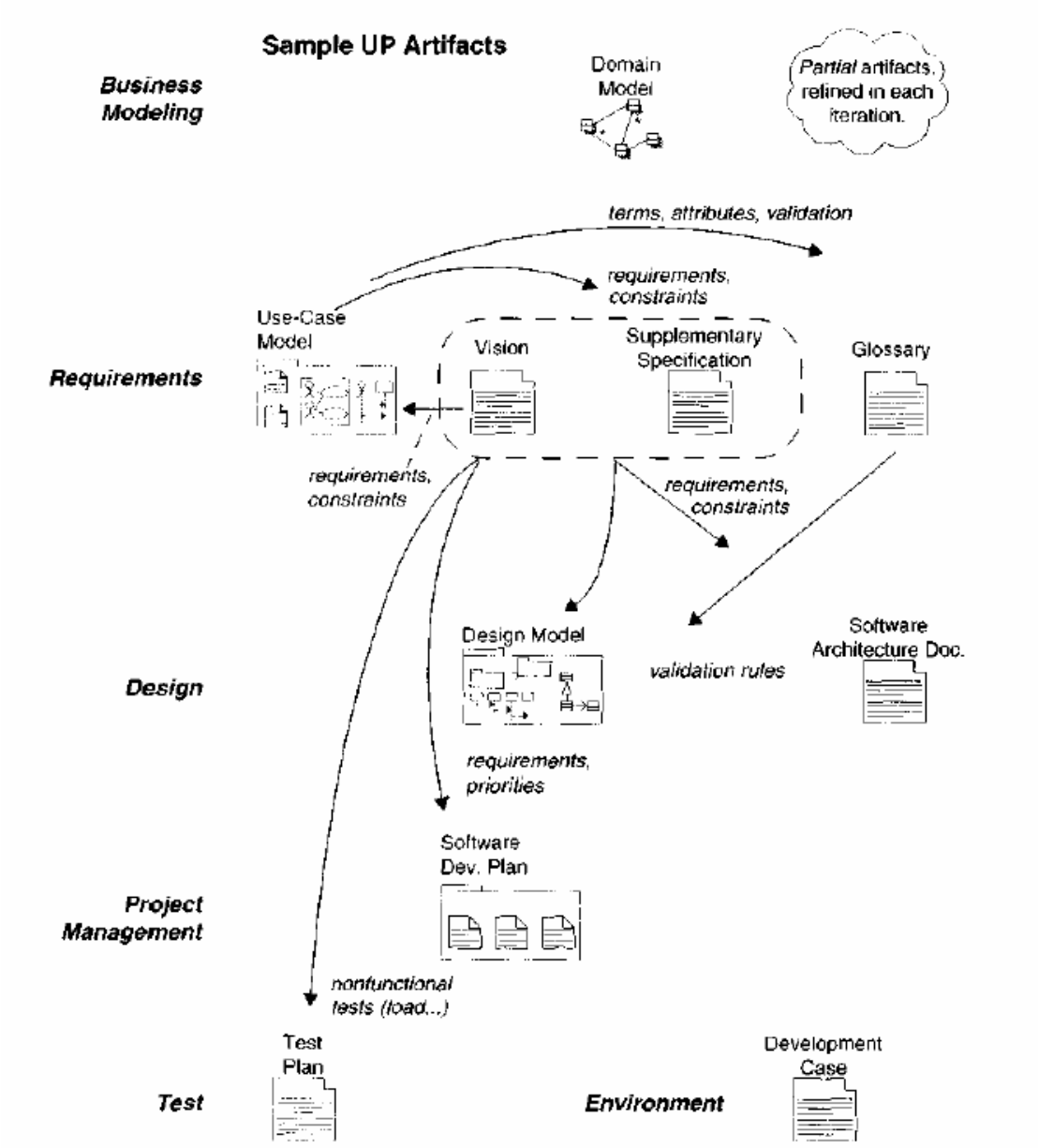


圖 7.1 UP 中工作成果的可能的交互影響

在 UP 中，專案願景與輔助規格書工作屬於需求工作科目中的開發活動，可能會在需求討論會中隨著使用案例分析工作開始。圖 7.2 針對這項工作提出一些時

間與空間上的建議。

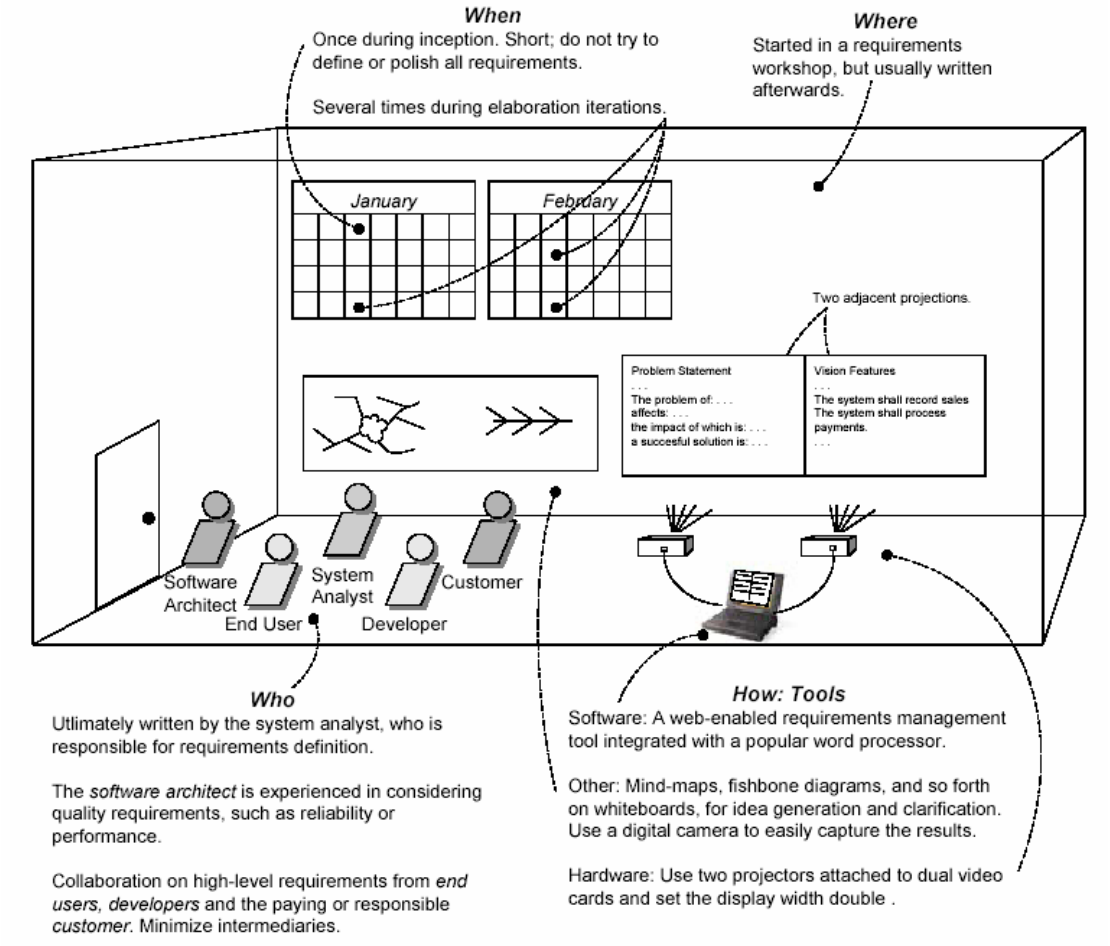


圖 7.2 開發流程與情境的設定

第八章從初始階段到詳述階段

硬的、脆的東西容易斷，柔軟的東西才容易保留下來。
— Tao Te Ching

本章目標

- 定義詳述階段開發步驟。
- 說明第三部分的編排動機。

簡介

在詳述階段最初一系列的反覆中，我們會：

- 發掘大部分需求，並且讓這些需求穩定下來
- 減緩或去除主要風險
- 實作核心架構的組成元素，並且驗證核心架構是可行的

很少有架構是沒有風險的－舉例來說，如果我們建構一個類似開發團隊之前成功開發過的網站，而且用同樣工具、有類似需求－那麼在早期反覆中，不會有什麼值得特別注意的。這時候，我們可能在早期反覆中實作有關鍵性但不影響架構的系統特性或使用案例。

本書在詳述階段中把重點放在介紹物件導向分析與設計、應用 UML、樣式與架構上。

第一節檢核點：初始階段中做了什麼事？

NextGen POS 專案中的初始階段開發步驟可能只花一個星期。所產生的工作成果內容可能很簡短、不完整。這個開發階段進行地很快，調查工作的份量也很少。初始階段並不是專案的需求開發階段，而是一個比較短的開發步驟，讓我們決定基本的可行性、風險與系統範圍，並且決定專案是否值得做詳述階段中更正式的調查工作。我們並沒有進行初始階段中所有合理的開發活動；調查工作的重點放在需求導向的工作成果上。初始階段可能會有的開發活動與工作成果包括：

- 爲期很短的需求討論會
- 找出大部分參與者、目標與使用案例並加以命名
- 大部分使用案例都用簡式的格式寫；10-20% 的使用案例則用正式的格式寫出細節以增加我們對系統範圍與複雜度的了解
- 找出大部分有影響性、風險的品質需求
- 寫出第一版的專案願景與輔助規格書
- 風險清單

- 舉例來說，領導階層真的希望 18 個月後在漢堡所舉辦的 POSWorld 展覽中展示系統。不過還沒有做更深入調查工作之前，展示系統所需花費的工作量還沒有辦法估計。

- 技術性、驗證概念用的雛型與其它調查工作，以探討特殊需求的技術可行性（「Java Swing 在觸控式螢幕顯示器上可以正常工作嗎？」）
- 使用者介面導向的雛型，以釐清功能需求的願景
- 元件該買／建構／再使用的建議，這些建議會在詳述階段中修正
  - 舉例來說，建議購買稅金計算套件
- 高階候選架構與建議的元件
  - 這不是詳細的架構說明，而且這個架構也不是最後結果或一定正確的。相反地，這只是一份簡短的推測，我們用它當作詳述階段調查工作的起點。舉例來說，「Java 的客戶端應用程式、沒有應用程式伺服器、用 Oracle 資料庫等等。」在詳述階段，我們會證明這樣的架構是否可行，或者發現它是不好的想法並拒絕這個架構。
- 第一個反覆的計畫
- 候選開發工具清單

第二節進入詳述階段

詳述階段中的最初幾個反覆是開發團隊真正進行正式調查工作、實作核心架構、釐清大部分需求、處理高風險設計議題的時候。在 UP 中，風險裡面包含企業價值。因此，早期工作包括實作真正重要的情節，而不是特別有技術風險的情節。詳述階段通常由兩個到四個反覆組成；除非開發團隊真的很大，每個反覆的時間長度建議在兩個到六個星期之間。每個反覆都是固定時間長度的，也就是說反覆的結束日期是固定的；如果開發團隊無法在日期內完成，就把需求放回未來工作清單中，這樣一來反覆的時間才能穩定，而能在確定時間測試發行版本。

詳述階段不是設計開發階段，或者我們在這個開發階段中先把模型完全開發完以準備建構階段中的實作工作－這樣做就是把瀑布式概念強加在反覆式開發方式與 UP 上。

在這個開發階段，我們也不是要產生會丟棄的雛型；相反地，程式碼與設計結果都是達到產品品質最終產品的其中一部分。UP 的某些說明可能有會產生誤解的術語「**架構雛型**（architectural prototype）」。它代表部分系統，而不是代表這個雛型是會丟棄的實驗品；在 UP 中，它代表最終產品中的一部分。我們更常叫它爲**可執行架構**（executable architecture）或**架構基準**（architectural baseline）。

用一句話說明詳述階段
建構核心架構、解決高風險元素、定義出大部分需求，並且估計整個專案時程與所需資源。

跟詳述階段有關的一些關鍵概念與最佳實務經驗包括：

- 進行時間很短、長度固定、以風險驅動的反覆
- 及早開始寫程式
- 調整設計、實作，並且測試架構中核心、有風險的部分
- 及早測試、經常測試、用真正資料測試
- 根據測試者、使用者與開發人員的回饋調整系統
- 透過在每個反覆中舉辦的需求討論會，寫出大部分使用案例與其它需求的細節

詳述階段中什麼東西影響架構最大？

早期反覆中必須建構並驗證核心架構。對 NextGen POS 專案來說－事實上、大部分的專案也都是如此－裡面要包含：

- 利用「寬且淺」的設計與實作方式；或者如 Grady Booch 所稱呼的「用有接縫方式設計。」
  - 換言之，找出分離的處理程序、分層結構、套件與子系統，以及它們的高階責任與介面。實作部份模組，以連接它們並釐清彼此之間的介面。模組裡面可能大部分都是「空殼的」程式碼。
- 修正模組內本地端與遠端的介面（其中包含最詳細的參數與傳回值。）
  - 舉例來說，物件的介面會把協力廠商會計系統的存取方式封裝起來。
  - 第一版介面很少是好的。及早試著進行壓力測試、「撕裂」並修正一些介面，這些介面又助於稍後多個開發團隊的平行開發工作，他們需要穩定的介面。
- 整合現有元件。
  - 舉例來說，稅金計算器。
- 實作簡化的端對端情節以加強橫跨許多主要元件的設計、實作與測試。
  - 舉例來說，*處理銷售*的主要成功情節中會用到的信用付款擴充情節。

詳述階段中的測試是很重要的，它可以讓我們獲得回饋、調整系統、驗證核心架構是穩固的。 NextGen 專案的早期測試包括：

- *處理銷售*中使用者介面的可用性測試。
- 測試遠端服務（例如信用授權服務）失效時的復原能力。
- 測試遠端服務（例如遠端稅金計算器）的高負載能力。

第三節規劃下次反覆

規劃專案並管理專案是很重要、範圍很大的主題。這裡介紹一些關鍵概念，第 36 章中會有更詳細的介紹。

根據風險、涵蓋率與重要性去組織需求與反覆。

- **風險**包含技術複雜度與其它因子，例如工作量或可用性的不確定性。

- **涵蓋率**隱含我們在早期反覆中，至少要碰觸系統所有主要部分－或許採用「廣且淺」、橫跨許多元件的實作方式。
- **重要性**跟高企業價值的功能有關。

這些準則是用來替反覆分級的。我們實作時會先區分使用案例或使用案例情節的等級。此外，某些需求會表達成不跟特定使用案例相關的高階系統特性，例如登入服務。這些系統特性也都要分級。

因爲新需求與對系統的新了解會影響分級，所以雖然我們在反覆 1 之前就會分級過，不過在反覆 2 之前還會重新分級，如此進行下去。換言之，計畫是可以調整的，而不是在專案一開始時就投機決定、固定不變。

通常根據某些小團體合作分級技術，我們會產生有點模糊的需求群組結果。例如：

等級	需求 （使用案例或系統特性）	說明
高	處理銷售 登入 …	對所有分級準則來說，都會把它分到最高等級。 普遍的。稍後很難加到系統中的。 …
中	維護使用者 …	會影響到安全性子領域…
低	…	…

根據這個分級方式，我們可以看到某些對架構有顯著影響的*處理銷售*使用案例關鍵情節會在早期反覆中探討。這份清單並沒有完；我們還要探討其它需求。此外，在每個反覆中，沒有明確寫出來或有明確寫出來的*啓動*使用案例也要能順利工作以滿足初始化動作的需要。

如果從 UP 來看，我們要把規劃資訊放到某些工作成果中：

- 下個反覆所選出來的需求會簡單列在**反覆計畫**（iteration plan）中。這份計畫不會包含所有反覆，它只是包含下個反覆的計畫而已。
- 如果反覆計畫中的簡短描述不足，我們可以把反覆要做的工作或需求用更詳細的方式寫在另一份**變動需求**（change request）中，並且把它交給負責的夥伴。
- 整體需求的分級方式會記錄在**軟體開發計畫**（software development plan）中。

第四節反覆 1 的需求與焦點：基本的物件導向分析與設計技能

在這個個案研究中，詳述階段的反覆 1 會把重點放在基礎、常見的物件導向分析與設計技術，以建構物件系統，例如指派責任到物件身上。當然，許多其它技能與開發步驟－例如資料庫設計、可用性工程與使用者介面設計－都是建構軟體所需的技能，不過它們超出物件導向分析與設計與 UP 入門的範圍。

反覆 1 需求

NextGen POS 應用程式的第一個反覆需求包括：

- 實作*處理銷售*使用案例中關鍵情節：輸入商品項目並收到現金付款。
- 實作*啓動系統*使用案例以支援所有反覆的初始化動作需要。
- 不處理一些比較花俏或複雜的部分，只處理快樂路徑情節，而且只做這部分的設計與實作。
- 暫時先不跟外部系統合作，例如稅金計算器或產品資料庫。
- 先不採用複雜的定價規則。

支援性使用者介面的設計與實作也會先做，不過也只先做部分。
接下來的反覆會以這個爲基礎逐漸擴充它。

橫跨數個反覆、逐漸開發完成的同一個使用案例

請注意，在反覆 1 中並不會完成*處理銷售*使用案例中的所有需求。通常每個反覆可能會實作不同情節或系統特性，某個使用案例要經過幾個反覆才會逐漸完成系統中的所有功能性（請參見圖 8.1。）另一方面，短的、簡單的情節才能夠在一個反覆中完成。

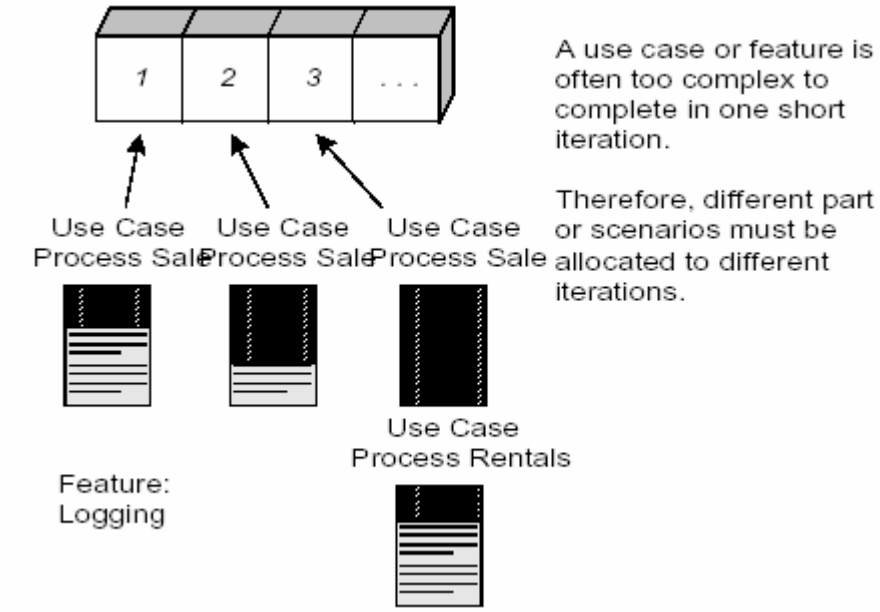


圖 8.1 某個使用案例的實作可能會橫跨數個反覆。

第五節詳述階段中應該開始做哪些工作成果

表 8.1 中列出詳述階段可能會開始製作的工作成果，並且說明它們所強調的議題。後面的章節中會更詳細說明相關細節，特別是領域模型與設計模型。簡單來說，表裡面拿掉從初始階段開始的工作成果（我們已經列在第 4 章了）；這裡只介紹最可能在詳述階段開始製作的工作成果。請注意，我們無法在一個反覆中完成這些工作成果；相反地，會在一連串的反覆中修正它們。

工作成果	說明
領域模型	這裡用視覺方式呈現領域概念;它跟領域實體的靜態資訊模型很像。
設計模型	用來描述邏輯設計的一些圖。其中包含軟體類別圖、物件互動圖、套件圖等等。
軟體架構文件	摘要關鍵架構議題與相關設計解決方案的學習輔助工具。裡面彙總了系統中重要的設計概念與動機。
資料模型	其中包含資料庫綱目、物件跟非物件呈現方式的對應策略。
測試模型	說明要測試什麼、怎麼測試。
實作模型	這是真正的實作－包含原始檔、可執行檔、資料庫等等。
使用案例記事板（use case storyboard）、使用者介面雛型	使用者介面說明、瀏覽路徑、可用性模型等等。

表 8.1 詳述階段中可能開始製作的工作成果，其中不包括從初始階段開始的工作成果。

第六節如何察覺自己還不是很了解詳述階段

- 大部分專案中，詳述階段會比「幾個月」還長。
- 裡面只有一個反覆（而且問題定義良好、沒有可能發生的例外需求。）
- 大部分需求在詳述階段之前就已經定義好。
- 並沒有探討有風險的組成元素與核心架構。
- 詳述階段結束之後不會有*可執行的*工作成果；不會寫出任何達到產品等級的程式碼。
- 主要把它視爲一個需求開發階段，然後在之後的建構階段中實作。
- 試著在寫程式之前做出完整、小心設計過的結果。
- 最少的回饋與系統調整；使用者不會持續參與評估系統並回饋給開發人員知道。
- 沒有很早就開始進行、用真正資料進行的測試。
- 在寫程式之前，很投機地固定架構不變。
- 把它視爲寫程式以驗證概念的一個開發步驟，而不是寫出達到產品品質的核心、可執行的架構。
- 沒有舉行多次短暫的需求討論會以便根據之前與現在反覆中的回饋調整系統並修正需求。

如果專案具有上面某種徵兆，那麼代表你們還沒有了解詳述階段。

第三部分詳述階段中的反覆 1

第九章使用案例模型：畫出系統循序圖

從理論來看，理論與實務是沒有差別的，不過事實上卻有差別。

— Jan L.A. van de Snepscheut

本章目標

- 找出系統事件（system event）。
- 產生使用案例的系統循序圖（system sequence diagram）

轉移到反覆 1

現在 NextGen POS 專案準備進入第一個真正的開發反覆。我們已經在初始階段進行一些需求工作以決定專案是否值得做真正的調查工作。我們已經完成第一個反覆的規劃工作，而且也決定在這個反覆開發*處理銷售*（沒有跟遠端一起合作完成工作）使用案例中簡單、只跟現金有關的成功情節，目標是現在開始進行「範圍廣、深度淺」的設計與實作開發工作，以碰觸新系統中大部分跟主架構有關的組成元素。第一個反覆裡面有許多工作都會跟建立開發用環境（例如開發工具、開發人員、開發流程與設定）有關；本書省略了這些部分。

這裡把注意力放在使用案例與建立領域模型的分析工作上。開始進行反覆 1 的設計工作之前，先調查一下領域問題是很有幫助的。調查中有一部分工作是釐清跟系統有關的輸入與輸出系統事件，我們將用 UML 的循序圖畫出這些事件。

簡介

系統循序圖是可以很快、很容易產生的工作成果，裡面展示跟討論中系統有關的輸入與輸出事件。 UML 中的循序圖表示法可以展現從外部參與者到系統的一些事件。

第一節系統行爲

在開始進行邏輯設計，設計軟體應用程式如何工作之前，把系統當「黑箱」研究並定義它的行爲是很有幫助的。**系統行爲**（system behavior）裡面描述系統該做些*什麼*，而不解釋它是如何辦到的。系統循序圖是描述系統行爲的其中一種工作成果。其它工作成果包括使用案例與系統合約（稍後會討論到。）

第二節系統循序圖

使用案例中描述外部參與者如何跟我們想要產生的軟體系統互動。在互動過程中，參與者會對系統產生事件，而且通常會要求執行某種操作以回應事件。舉例來說，當收銀員輸入一個商品 ID 時，收銀員會要求 POS 系統記錄商品的銷售情形。這個請求事件會啓動系統中的一個操作。

我們有必要把外部參與者要求系統做的操作加以區隔並展現出來，因爲這些事件是我們了解系統行爲的重要東西。

系統循序圖（system sequence diagram，SSD）是系統的一張概圖，裡面會展現使用案例中某個特別情節、外部參與者所產生的事件、事件的發生順序以及系統之間的事件。所有事件都用黑箱方式描述；圖中會強調跨越系統邊界，從參與者到系統的事件。

系統循序圖裡面應該只畫出使用案例的主要成功情節與常發生或複雜的替代情節。

UML 中並沒有特別針對「系統」的循序圖定義特別的表示法，只是用循序圖畫出系統循序圖而已。加上「系統」兩個字只是強調我們畫循序圖時把系統當成黑箱來看。稍後，我們將會把循序圖用在其它情境－用循序圖展示設計結果，說明軟體物件之間是如何互動以達成工作的。

第三節系統循序圖的範例

針對使用案例中某個特殊的事件流，系統循序圖中會展現跟系統有直接互動關係的外部參與者、系統（它被當成一個黑箱）以及參與者產生的系統事件（請參見圖 9.1。）時間先後是由上而下看的，而且事件先後順序也會遵循它們在使用案例中的順序。

系統事件可能會帶參數。

下面這個例子是*處理銷售*使用案例的主要成功情節。我們可以看到收銀員會產生 *makeNewSale* 、*enterItem* 、 *endSale* 與 *makePayment* 系統事件。

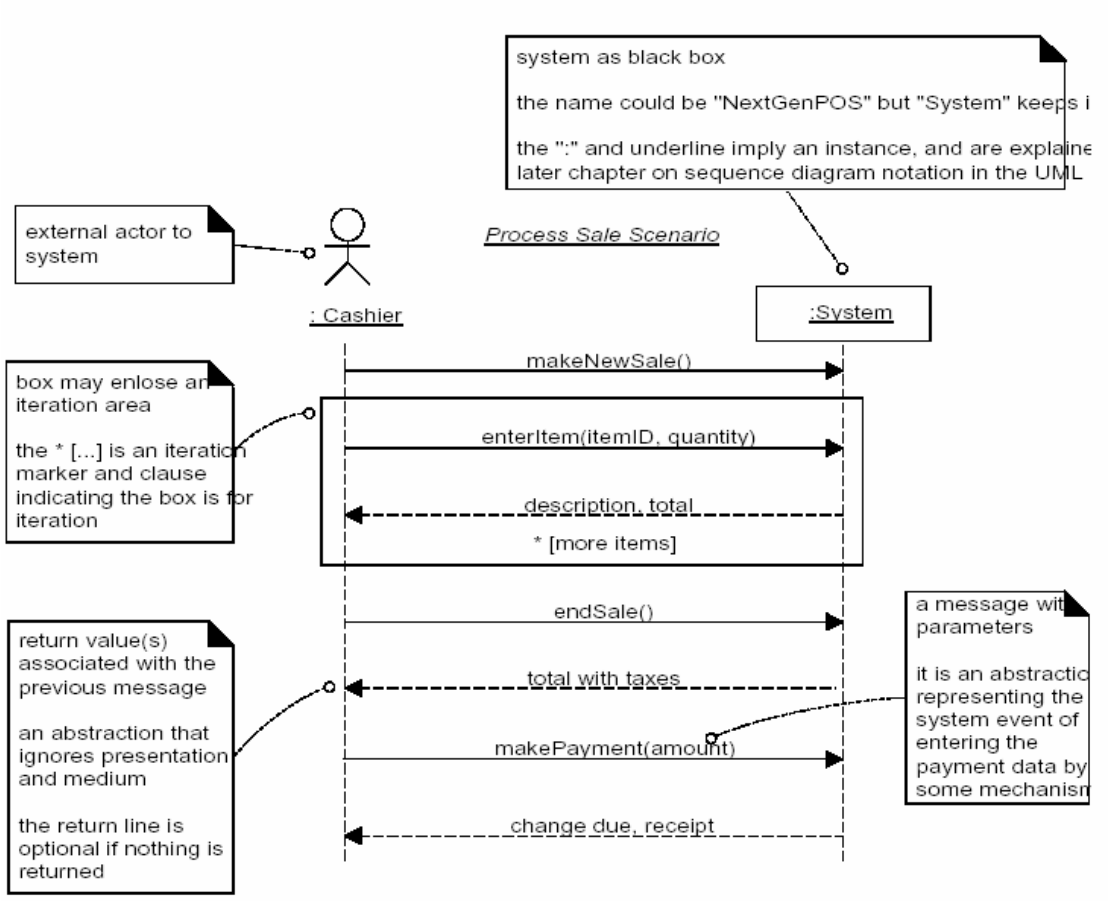


圖 9.1 *處理銷售*情節的 SSD。

第四節系統之間的系統循序圖

SSD 也可以展現系統之間的合作情形，例如 NextGen POS 跟外部信用付款授權者之間的合作情形。然而，稍後反覆中才會介紹這個個案研究，目前這個反覆並不打算涵蓋跟遠端系統合作的情形。

第五節系統循序圖與使用案例

SSD 用來展示使用案例情節中的系統事件，我們可以根據使用案例產生 SSD（請參見圖 9.2。）

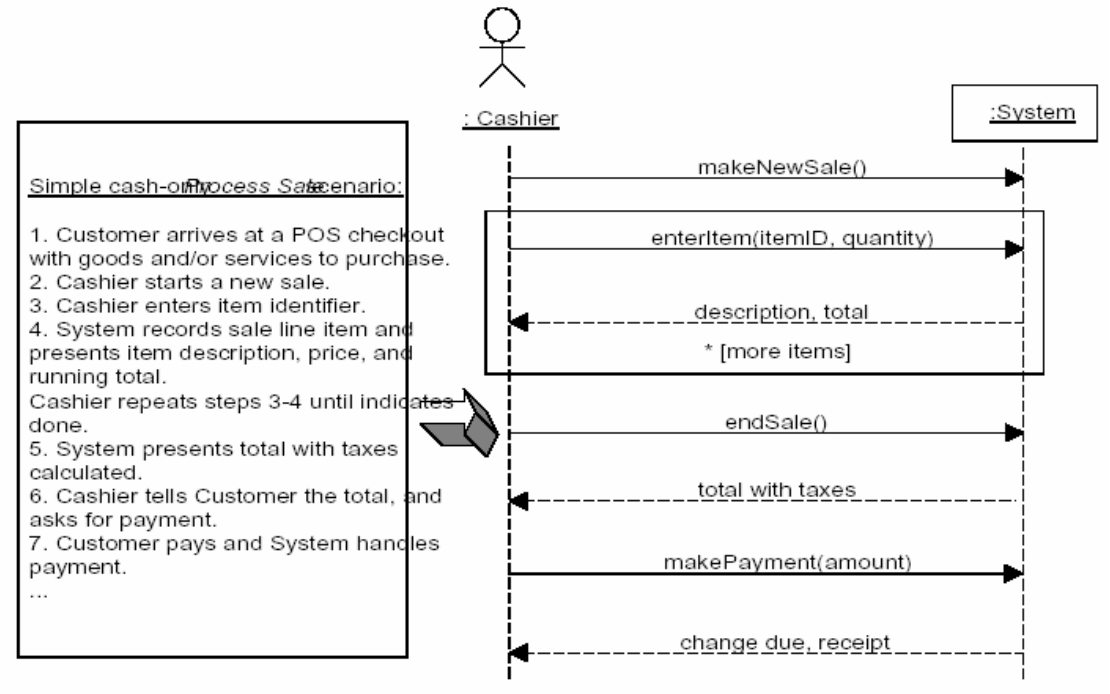


圖 9.2 從使用案例中衍生出來的 SSD

第六節系統事件與系統邊界

爲了找出系統事件，就像前一章說明使用案例時一樣，我們必須清楚定出系統邊界在哪裡。爲了達到軟體開發目的，我們通常把系統邊界放在軟體（也有可能是硬體）系統本身；在這個情境下，系統事件是直接驅動軟體的外部事件（請參見圖 9.3。）

我們現在試著找出*處理銷售*使用案例的系統事件。首先，我們必須決定跟軟體系統直接*互*動的參與者。在這個只跟現金有關的情節中，跟收銀員*互*動的顧客並不會直接跟 POS 系統*互*動－只有收銀員才會跟 POS 系統直接*互*動。因此，顧客不是系統事件的產生者；收銀員才是。

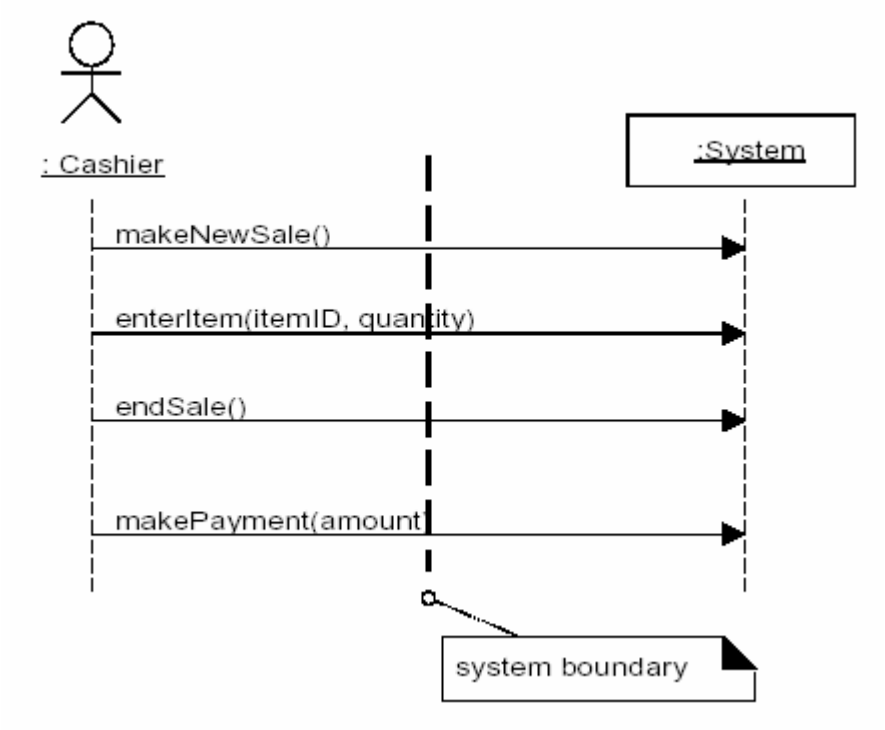


圖 9.3 定義系統邊界

第七節替系統事件與系統操作命名

我們應該表達出系統事件（以及跟它們相關的系統操作）的意圖，不要只表達出輸入時的媒體或介面視窗小元件。

同時，在系統事件的名稱前面加上動詞（例如加入…、輸入…、終止…、產生…等等）（如圖 9.4 所示）可以讓系統事件的目的更清晰，因爲這樣的命名方式可以讓我們知道這些事件是命令導向的。

因此「輸入商品（enterItem）」比「掃描（scan ）」（也就是雷射掃描）還好，因爲關於設計時所選擇的捕捉系統事件用介面，前者一方面既抓住了操作的意圖，也維持了介面的抽象性與不確定性。

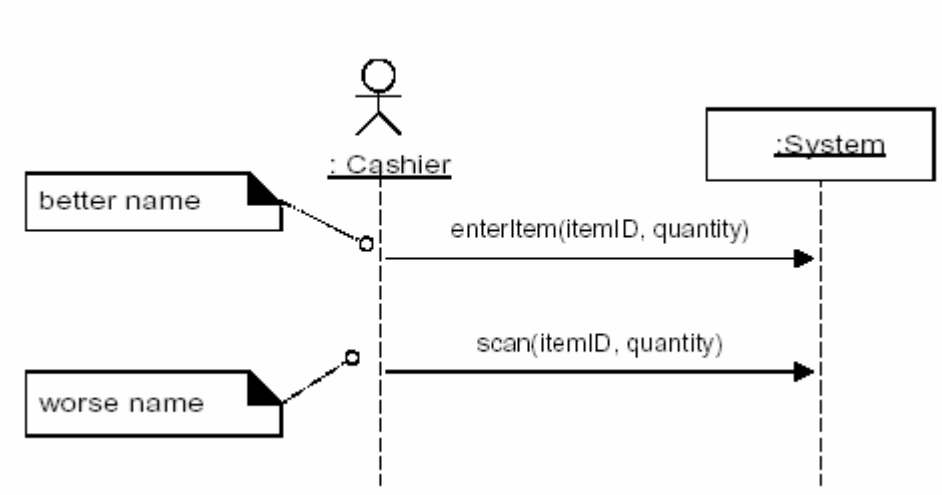


圖 9.4 在抽象層次替事件與操作命名

第八節秀出使用案例中的一些說明文字

有時候我們會想要在情節中秀出使用案例說明文字的片段，以釐清或強化外部參與者與系統的兩個觀點（請參見圖 9.5。）說明文字提供我們一些細節與情境，而圖則用視覺方式展現互動情形。

第九節系統循序圖與字彙表

在 SSD 中秀出來的字（操作、參數與傳回資料）都很精簡。我們可能需要稍微解釋，才能在進行設計工作時清楚知道傳什麼東西進去、什麼東西出去。如果在使用案例中沒有明確說明這些東西，那麼我們就應該在字彙表中加以說明。
然而，跟往常一樣，產生非程式碼（專案核心）工作成果時，請保持懷疑的態度。有時候字彙表中的資料應該真的會有某種使用意義，或者我們會在裡面做一些決策，而不是只把字彙表當成價值低、沒有必要的工作成果。

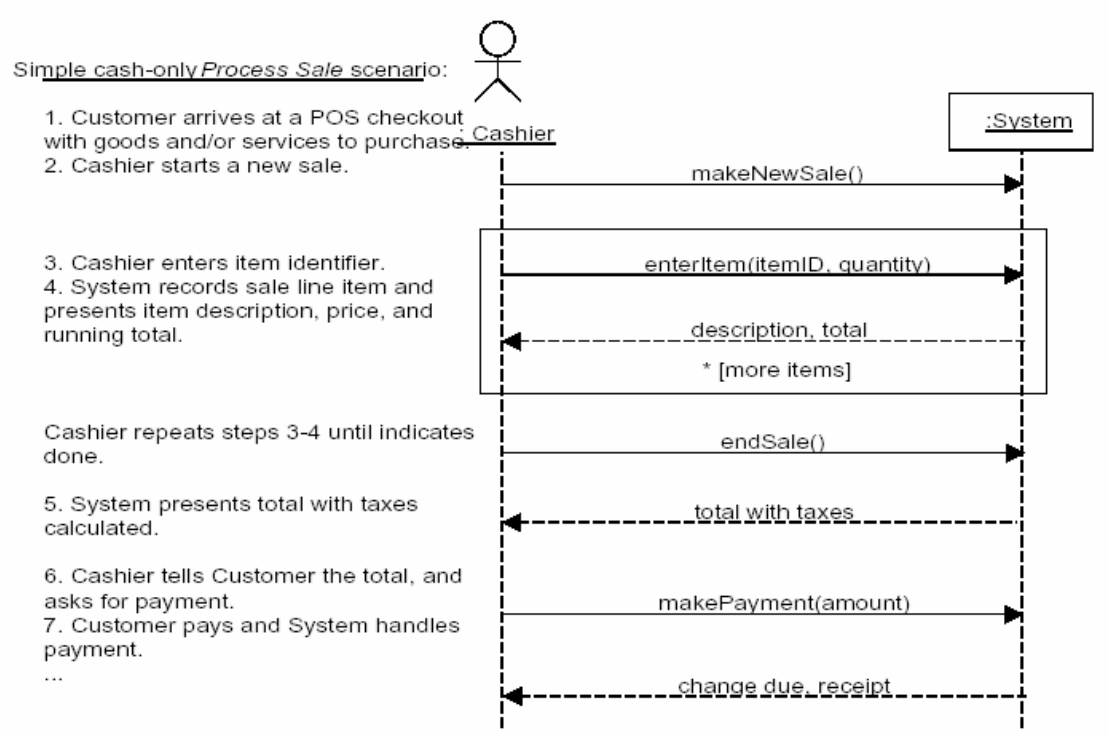


圖 9.5 加上使用案例說明文字的 SSD

第十節UP 中的系統循序圖

SSD 是使用案例模型其中一部份－它用視覺方式展現使用案例中所隱含的互動情形。雖然 UP 的創造者知道並了解這種圖的用途，不過原始 UP 說明中並沒

有明確提到 SSD 。 SSD 是 UP 或 RUP 文件中沒有提到、好用的分析或設計工作成果之一。

開發階段

初始階段－ 初始階段中通常不會用到 SSD 。

詳述階段－大部分的 SSD 都是在詳述階段產生的，因爲 SSD 可以讓我們 (1).找出系統事件細節以釐清哪些主要操作是設計系統時該處理的、(2).寫出系統操作合約（第 13 章會討論到），以及 (3).算出可能有用的支援性估算值（例如用尚未調整過的功能點做出巨觀的估算值【macroestimation】與 COCOMO II。）請注意，我們不需要產生所有使用案例、所有情節的 SSD －至少不要同時做這件事。相反地，我們只會針對目前反覆選出一些情節，然後畫出它們的 SSD 。我們應該只花幾分鐘或半個小時產生這些 SSD 。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	領域模型		s		
需求	**使用案例模型(SSD)**	s	r		
	專案願景	s	r		
	輔助規格書	s	r		
	字彙表	s	r		
設計	設計模型		s	r	
實作	軟體架構文件		s		
	資料模型		s	r	
	實作模型		s	r	r
專案管理	軟體開發計畫	s	r	r	r
測試	測試模型		s	r	
環境	開發案例	s	r		

表 9.1UP 中可能用到的工作成果以及產生這些工作成果的時間點。s － 初版；r － 修正版

第十一節進階讀物

過去幾十年來，已經有各式各樣圖把系統視爲黑箱、展現輸入／輸出事件；例如通訊系統中的電話流圖（call-flow diagram）。在物件導向方法論中，採用 Fusion 方法論【Coleman+94】的人特別流行用這種做法，他們提供 SSD 、系統操作跟其它分析與設計工作成果之間詳細、可能存在的關係。

第十二節UP 中跟使用案例模型相關的工作成果

請參見圖 9.6 ，了解 SSD 跟其它工作成果之間可能存在的關係。

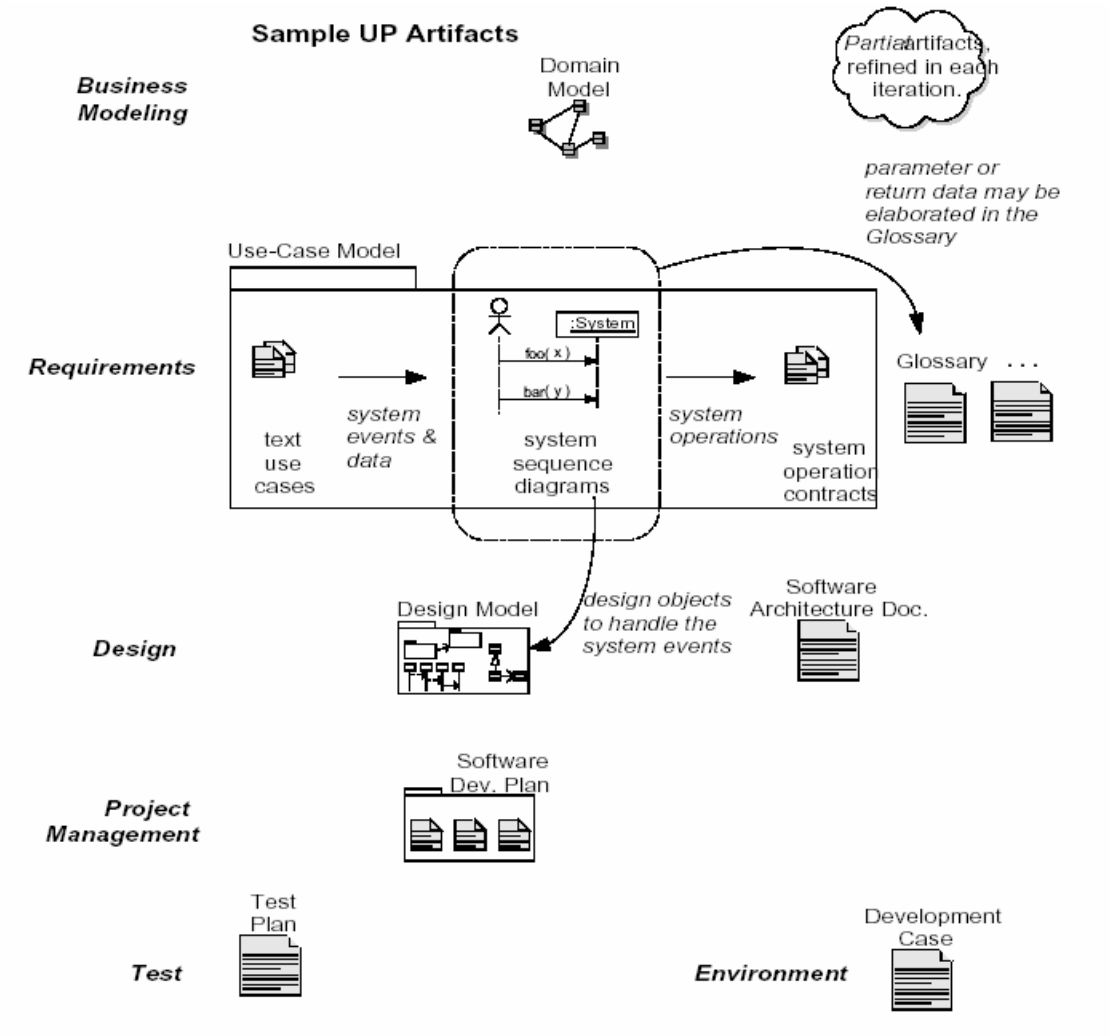


圖 9.6 UP 中工作成果之間可能存在的相互影響關係。

第十章領域模型：用視覺方式呈現概念

空有理論是行不通的，真的去做才能成功。
— 一句來源不詳的管理格言

本章目標

- 找出跟目前反覆需求有關的概念性類別。
- 產生初版的領域模型。
- 區分正確與錯誤的屬性。
- 有必要時（在領域模型中）加入*規格型*概念性類別。
- 比較並比對概念性觀點與實作觀點之間的差異。

簡介

很多人用領域模型作爲設計軟體物件時的靈感來源，而且它也是本書後面討論到的許多工作成果的必要資訊來源。因此，如果你對建立領域模型不熟的話，本章對你非常重要。

領域模型會展現問題領域中（對建模型者）有意義的概念性類別；它是我們進行物件導向分析【註】時要產生的最重要工作成果。本章中將探討建立領域模型的入門技能。後面兩章則是建立領域模型技能的延伸－增加屬性與關聯。

【註】使用案例是需求分析中很重要的工作成果，不過它不是特別針對物件導向的。它的重點放在領域中的流程觀點。

找出很多概念性物件或類別是物件導向分析的重心，而且這裡所花費的功夫在進行設計與實作工作時可以得到回饋。

找出概念性類別是調查問題領域的一部分工作。UML 裡面有展示領域模型用的類別圖表示法。

關鍵概念

領域模型代表真實世界中的概念性類別，而不是軟體元件。它*不是*描述軟體類別或軟體物件責任的一些圖。

第一節領域模型

典型的物件導向分析或調查工作是把有興趣領域分解成我們所熟知的獨立概念性類別或物件（譯註：領域相關知識是一連串糾葛在一起的觀念，我們需要先找

出概念：概念後來會演變成類別，再釐清它們之間關係：用關聯、繼承、可瀏覽性、多重性等等描述關係。）領域模型（domain model）用視覺方式呈現有興趣領域中的概念性類別或真實世界中的物件【MO95、Fowler96】。也有人把它們稱爲概念性模型（conceptual model）（這是本書第一版所用的術語）、領域物件模型（domain object model）或分析物件模型（analysis object model）【註】。

【註】：它們也跟概念性實體關係模型有關，後者能夠顯示出純粹的概念性領域觀點，不過大部分人把概念性實體關係模型重新解釋成資料庫設計的資料模型。領域模型並不是資料模型。

UP 中有定義領域模型【註】，並且把它當成建立企業模型工作科目時製作的工作成果。

【註】： 當某個模型是 UP 中有定義的正式模型時，（譯註：原文版中）我們會把用大寫表現這個術語以強調它是 UP 中的正式模型，例如我們用大寫「Domain Model」跟大家所知道的概念「domain model」做區別。

當我們用 UML 表示法呈現領域模型時，會畫出一些沒有操作的**類別圖**（class diagram）（譯註：當我們在畫領域模型時，裡面的類別都是概念性類別，這樣的類別可能會有屬性，而不會有操作。）圖中可能會顯示出：

- 領域類別或概念性類別
- 概念性類別之間的關聯
- 概念性類別的屬性

舉例來說，圖 10.1 顯示的是領域模型的一部份。從圖中可以發現：(1). *Payment* 與 *Sale* 概念性類別都是這個領域裡面很重要的概念，(2). 兩者之間的關係也值得注意，還有 (3). *Sale* 會有 date 與 time 屬性。目前我們先不說明其它細節，因爲現在這些細節並不是很重要（譯註：這些細節包括角色名稱【role name】、多重性【multiplicity】、可瀏覽性【navigability】。）

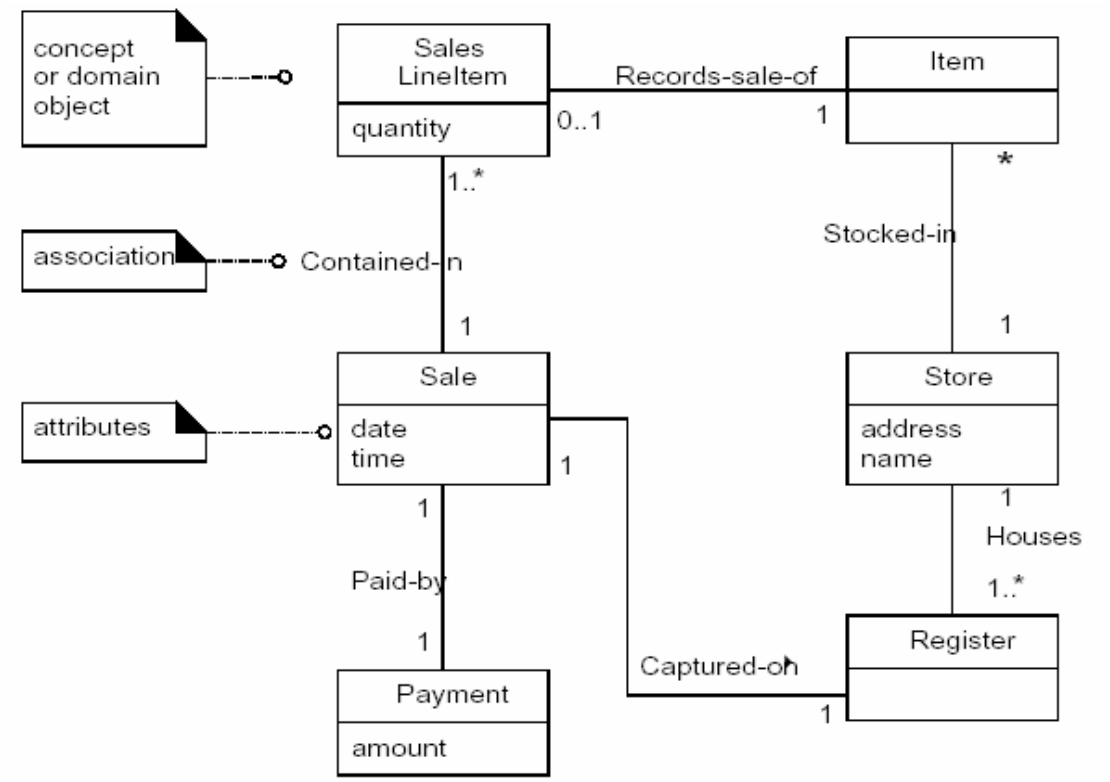


圖 10.1 領域模型的一部份—用視覺方式展現的資料字典。在關聯端點上的數字代表多重性，我們會在下一章說明。

關鍵概念：領域模型－用視覺方式呈現

抽象概念字典

請看一下圖 10.1 。裡面用視覺方式呈現領域中一些術語或概念性類別。同時圖中也展現這些概念性類別的*抽象概念*（abtraction），因爲裡面包含了許多溝通時會用到的東西，例如登錄人員、銷售等等。模型中只會顯示出部分觀點或抽象概念，忽略（建模型者）沒興趣的細節。

圖中（用 UML 表示法）展現的資訊也可以用文字或說明紀錄在字彙表或其它地方。不過用視覺方式呈現的語言更容易讓我們瞭解不同組成元素與其關係—「用視覺瞭解概念」應該是人類的優點。

因此，領域模型讓我們用視覺方式呈現(1).值得注意的抽象概念、(2).領域字彙表與(3).包含領域資訊內容的字典。

領域模型不是軟體元件模型

如圖 10.2 所示，領域模型用視覺方式呈現我們有興趣的真實世界領域，而不是

呈現像 Java 或 C++ 類別的軟體元件或軟體物件責任。因此，下面元素不適合放到領域模型中：

- 軟體工作成果（例如視窗或資料庫），除非模型要描述的領域是這樣的軟體概念，例如軟體使用者介面模型。
- 責任或方法【註】。

【註】：在建物件模型時，我們通常會談論到跟軟體元件相關的責任，而且方法也是純粹的軟體概念。不過，領域模型中要描述的是真實世界概念而不是軟體元件。*在設計*工作中考量物件責任是很重要的；不過責任不是領域模型的一部份。在領域模型呈現責任的合理情況是：當責任代表人類工作人員的角色（例如收銀員）時，而且建模型者也希望記錄這些人類工作人員的責任。

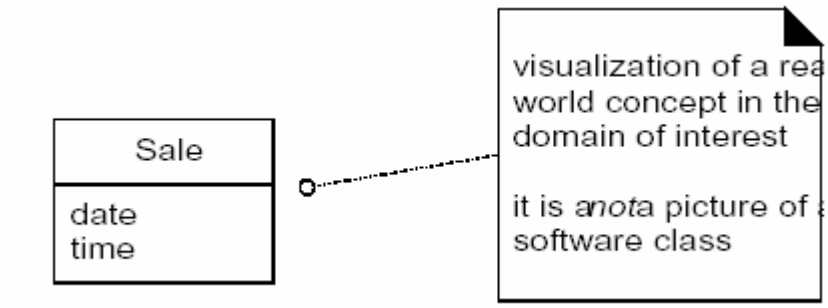


圖 10.2 我們用領域模型展示真實世界中的概念性類別，而不是軟體類別。

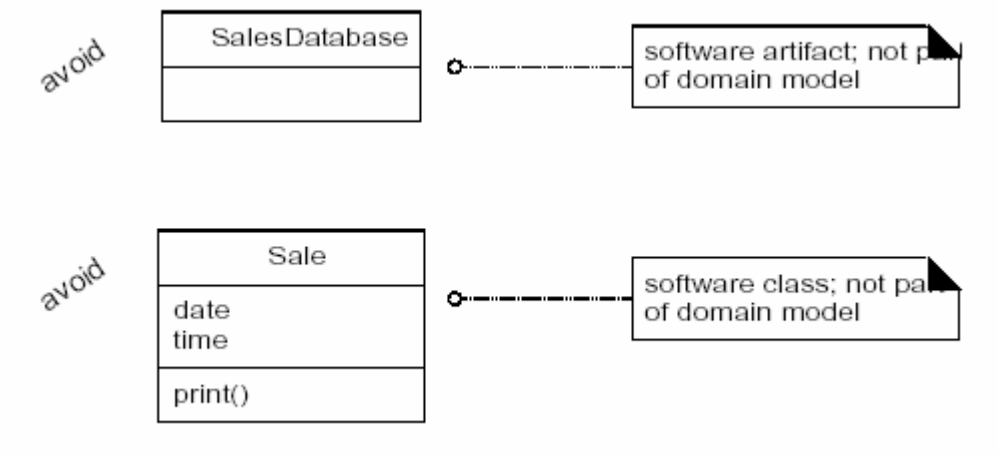


圖 10.3 領域模型中不需要展現軟體工作成果或軟體類別。

概念性類別

領域模型展示領域中的概念性類別或術語。非正式來說，概念性類別可能是一個想法、東西或物件。正式一點的說法則是：概念性類別可以從它的符號、內涵與外顯結果來考量【MO95】（請參見圖 10.4。）

- **符號**（symbol）—代表概念性類別的字或圖片。
- **內涵**（intension）—概念性類別的定義。

■ **外顯結果**（extension）—可應用概念性類別的地方。

舉例來說，考量「買東西的交易」這個事件會用到的概念性類別。我可能會把這個概念性類別用 *Sale* 符號命名。 Sale 的內涵則可能是「代表買東西交易的事件，它會有日期與時間。」 Sale 的外顯結果則是所有的銷售實例；換言之，所有的銷售結果。

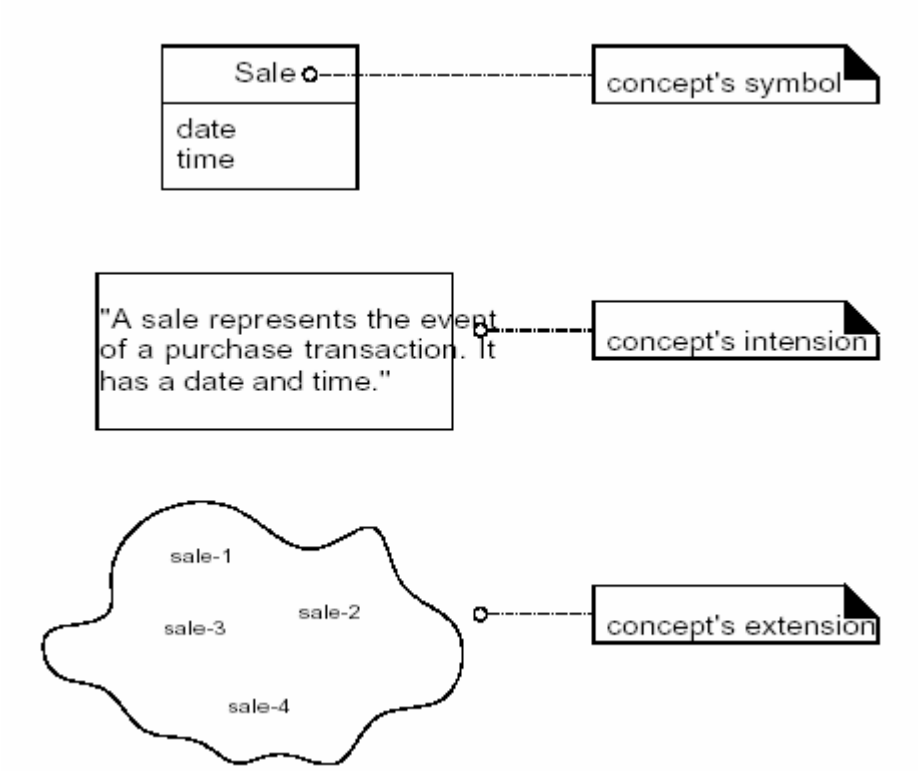


圖 10.4 概念性類別會有符號、內涵與外顯結果。

當我們建立領域模型時，裡面通常放的是概念性類別的符號與內涵觀點，因爲它們是我們實務上最有興趣的東西。

領域模型與分解

軟體要解決的問題可能很複雜；分解—切割並加以解決（divide-and-conquer）—是處理複雜性時常見的策略之一，它把問題空間切割成可理解的單位大小。在結構化分析（structured analysis）中，我們所分解的維度是流程或功能。然而，在物件導向分析中，分解的維度基本上是領域中的東西或實體。

物件導向與結構化分析最核心的差異是：物件導向是根據概念性類別（物件）切割而不是根據功能切割領域的。

因此，主要的分析工作是要找出問題領域中的不同概念，並且把領域模型中的結果記錄成文件。

銷售領域中的概念性類別

舉例來說，在商店銷售的真實世界領域中包括 *Store* 、 *Register* 與 *Sale* 概念性類別。因此，我們的領域模型中包括 *Store* 、 *Register* 與 *Sale*，如圖 10.5 所示。

圖 10.5 商店領域中領域模型的一部份。

第二節找出概念性類別

我們的目標是在有興趣領域（sales）中，產生有興趣或有意義概念性類別的領域模型。在這裡，我們有興趣的概念是跟使用案例*處理銷售*相關的概念。
在反覆式開發方式中，我們可以在詳述階段的幾個反覆中慢慢建構出領域模型。在每個反覆裡面，我們會把領域模型限制在考量過的之前情節與目前情節中。而不是建構一個「大爆炸」模型，在早期嘗試捕捉出所有概念性類別與其關係。例如我們在這個反覆中只考慮簡化、跟現金有關的*處理銷售*情節，產生領域模型的一部份。這個領域模型只會反映出考量中的情節，不會包含更多情節。
因此，核心工作是找出跟設計中情節相關的概念性類別。
下面指引可以幫我們找出概念性類別：

過分簡化原本有許多細部概念性類別的領域模型比詳細列出這些細部概念性類別更好。

不要認爲有比較少概念性類別的領域模型一定比較好；有時候，放進比較多的概念性類別似乎會比較切實際。
剛開始找出概念性類別時遺漏了一些概念性類別，而在考量屬性與關聯時才發掘到其它概念性類別是很正常的事。如果找到新的概念性類別，請把它們加到領域模型中。
不要因爲需求中沒有明顯說明我們有必要記錄某些概念性類別資訊，就把這樣的概念性類別排除在外（這是進行關聯式資料庫設計、建立資料庫模型時常見的準則，不過卻不適用於建立領域模型上），或者因爲概念性類別沒有屬性就把它排出在領域模型之外。
沒有屬性的概念性類別或是在領域中只扮演行爲角色而非資訊角色的概念性類別都是可接受的。

找出概念性類別的策略

我們會在下面的小節介紹找出概念性類別的兩種技術：

1. 利用概念性類別分類清單。
2. 找出名詞片語。

此外，另一種建立領域模型的好方法是使用**分析樣式**（analysis pattern），這是由專家所建立現存、不完整的領域模型，你可以參考一些已出版的資源，例如 Analysis Patterns 【Fowler96】與 Data Model Patterns 【Hay96】。

利用概念性類別分類清單

在開始建立領域模型之前，我們可以先建立候選概念性類別清單。表 10.1 中包含許多值得注意、常見的概念性類別分類，不過表中並沒有說明這些分類重要性的先後順序。裡面的範例是從商店與航空訂位領域中找出來的。

概念性類別分類	範例
實體或看得到的物件	Register Airplane
某些東西的規格、設計方式或說明文字	ProductSpecification FlightDescription
地點	Store Airport
交易	Sale、Payment Reservation
交易明細	SalesLineItem
人的角色	Cashier Pilot
某些東西的容器	Store、Bin Airplane
容器中的東西	Item Passenger
系統外的其它電腦系統或機電系統	CreditPaymentAuthorizationSystem AirTrafficControl
抽象名詞概念	Hunger Acrophobia
組織	SalesDepartment ObjectAirline
事件	Sale、Payment、Meeting、Flight、Crash、Landing
流程（通常*不會*用概念呈現流程，不過還是有可能用概念呈現流程的）	SellingProduct BookingASeat
規則與政策	RefundPolicy

	CancellationPolicy
目錄	ProductCatalog PartsCatalog
理財、工作、合約、法律相關事物的紀錄	Receipt、Ledger、EmploymentContract、MaintenanceLog
理財工具	LineOfCredit Stock
手冊、文件、參考論文、書籍	DailyPriceChangeList RepairManual

表 10.1 概念性類別分類清單

用概念性類別跟名詞片語之間的關連性

尋找概念性類別

我們建議（因爲這種技術很簡單）的另一種有用技術是【Abbot83】中的語言分析（Linguistic analysis）技術：在領域的文字說明中找出名詞與名詞片語，並且把它們當成候選概念性類別或屬性。

應用這個技術時要很小心；想要用機械方式做名詞與類別之間的對應是不可能的，自然語言中的字含意通常是很模糊的。

然而，這是我們的另一個靈感來源。從分析而來的正式格式使用案例，對我們來說是最好的說明文字。例如我們可以用*處理銷售*使用案例中目前考慮到的情節作爲尋找概念性類別的靈感來源。

主要成功情節（基本流程）：

1. **顧客**帶著要買的**商品**與／或**服務**到 **POS** 的**結帳櫃檯**前面。
2. **收銀員**啓動一筆新的**銷售**。
3. **收銀員**輸入**商品識別碼**。
4. 系統記錄**銷售明細**，並且顯示**商品說明文字**、**價格**與累計購買總金額。根據一組計價規則計算價格。

收銀員重複步驟 3-4 直到完成所有商品爲止。

5. 系統秀出包含**稅金**的總金額。
6. 收銀員告知顧客總金額，並且要求顧客**付款**。
7. 顧客付款，並且由系統處理付款。
8. 系統記錄完成的**銷售**，並且送出銷售與付款資訊到外部**會計**系統（爲了會計與**佣金**）與**庫存**系統（爲了更新庫存。）
9. 系統秀出**收據**。
10. 顧客帶著商品與收據（如果有的話）離開。

擴充情節（或替代流程）：

…

7a. 用現金付款：

1. 收銀員輸入**顧客付的現金**。
2. 系統呈現**結餘金額**，並打開**現金抽屜**。
3. 收銀員放進顧客付的現金，然後拿結餘金額給顧客。
4. 系統記錄現金付款金額。

領域模型會用視覺方式呈現值得注意的領域概念與術語。我們要從哪裡找到這些術語呢？從使用案例啊！因此，我們可以把從使用案例中找到的名詞片語當作領域概念與術語，它們是很豐富的靈感來源。

有些找到的名詞片語可以作爲候選概念性類別，有些在目前反覆中則可能要先忽略（例如「會計」與「佣金」），而且有些可能只是概念性類別的屬性而已。請參考下一節與下一章中對屬性的一些建議，它們教我們如何區分屬性與概念性類別的不同。

這種方法的弱點是：自然語言的不準確性，不同名詞片語可能代表相同概念性類別或屬性。我們建議大家把這個技術跟*概念性類別分類清單*技術一起使用。

第三節找出銷售領域中的候選概念性類別

利用概念性類別分類清單與名詞片語分析技術，我們產生了這個領域中的候選概念性類別清單。這份清單只包含目前考量中的需求—*處理銷售*使用案例中簡化過的情節。

Register	*ProductSpecification*
Item	*SalesLineItem*
Store	*Cashier*
Sale	*Customer*
Payment	*Manager*
ProductCatalog	

其實並沒有所謂的「目前」清單。它有一點像是建模型者認爲值得列入考慮的抽象概念、領域字彙表的集合。不過，只要按照這些找出概念性類別的策略，不同的建模型者應該會產生類似的清單。

報告型的物件－要把收據放到模型中嗎？

收據是銷售與付款的紀錄，它也是領域中相當重要的概念性類別，我們應該把它放到模型中嗎？

我們需要考量幾個因素：

- 收據是銷售的一種報告。一般來說，我們通常不會在領域模型中秀出其他資

訊的報告，因爲這些資訊是從其它資訊來源來的；報告中複製其它地方可以找到的資訊。這是我們把收據排除在領域模型外的一個原因。

- 就企業規則來說，收據扮演一種特殊角色：持收據的人通常可以拿它退還之前買的東西。這是我們把它放在模型中的原因。
  因爲目前反覆中不考慮退貨問題，所以我們現在把 *Receipt* 排除在模型外。等到*處理退貨*使用案例的反覆時，再把它放到模型中。

第四節領域模型指引

如何產生領域模型

我們可以用下面的開發步驟產生領域模型：

1. 用概念性類別分類清單與找出名詞片語技術，列出跟目前討論中需求相關的候選概念性類別。
2. 把這些候選概念性類別放到領域模型中。
3. 把必要的關聯加入領域模型，記錄一些一定要記住的關係(後面章節會討論。)
4. 把必要的屬性加入領域模型中以滿足資訊需求（後面章節會討論。）

另一種輔助性的有效方法是學習並複製分析樣式，稍後章節中會討論到。

命名或建模型時的策略：地圖繪製師

地圖繪製師策略可以同時應用在地圖與領域模型中。
用製圖師或地圖繪製師工作時的精神產生領域模型：

- 使用這個地區現有的名稱。
- 排除不相關的一些特性。
- 不要加入不存在的東西。

領域模型就像描述領域中事物或概念的一種地圖。這種想法強調領域模型的分析角色，而且隱含下面一些建議：

- 地圖繪製師會使用這個地區現有的名稱—他們不會改變地圖上的城市名稱。就領域模型來說，這代表*當我們替概念性類別與屬性命名時，會使用領域模型中的字彙表*。例如如果我們正在開發圖書館模型，把顧客稱爲「借閱者」或「贊助者」都是圖書館館員常用的術語。
- 如果地圖繪製師認爲某些東西跟畫地圖的目的無關，就不會把它們畫在地圖上；舉例來說，我們不需要把地形與人口畫在地圖上。同樣地，領域模型中也會把問題領域中跟需求無關的概念性類別排除。例如我們不會把 *Pen* 與 *PaperBag* 放在領域模型中（就目前需求而言），因爲這些東西現在並沒有扮演很明顯的角色。

- 地圖繪製師不會加入不存在的東西，例如不存在的山。同樣地，領域模型中也會把不存在於討論問題領域中的東西排除在領域模型之外。

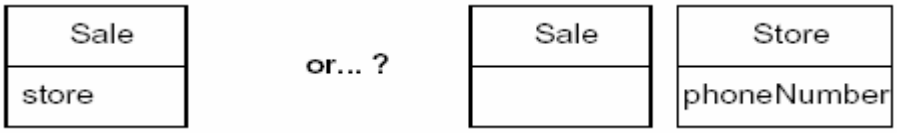
也有人把這個原則稱爲*使用領域字彙表*策略【Coad95】。

尋找概念性類別時常見的錯誤

當我們產生領域模型時，最常見的錯誤可能是把應該視爲概念的東西當成屬性看。有一個經驗法可以預防這種錯誤：

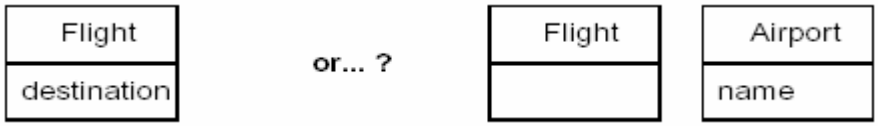
如果我們不認爲某些概念性類別 X 是真實世界中的數字或說明文字，那麼 X 可能就是概念性類別而不是屬性。

舉例來說， *store* 應該是 *Sale* 的屬性或是獨立的概念性類別 *Store* 呢？



在真實世界中，商店不能視爲一個數字或說明文字—這個術語很可能是一個合法的實體、組織與擁有空間的某種東西。因此， *Store* 應該是一個概念。

另一個例子，想一下航空訂位領域。我們應該把 *destination* 當成 *Flight* 的屬性，或者把它當成獨立的概念性類別 *Airport* ？



在真實世界中，目的地機場並不是一個數字或說明文字—它是一個擁有空間的龐然大物。因此， Airport 應該是一個概念。

如果不確定的話，還是先把它當成獨立概念。領域模型中很少出現屬性。

第五節區別相似的概念性類別－Register vs. 「POST」

POST 代表點銷售終端機。在電腦世界中，終端機可以是系統中任何終端點設備，例如客戶端 PC 、無線網路 PDA 等等。在很久以前、還沒有 POST 系統時，商店裡面都會有一本 *register* （現金登記簿）—這本簿記裡面會紀錄銷售與付款記錄。最後，它演變成自動化機器：「現金收銀機。」今天， *POST* 系統已經可以滿足現金登記簿的角色（請參見圖 10.6。）

現金登記簿是紀錄銷售與付款的東西，不過 POST 也可以滿足這個需要。然而，術語 *register* 比較抽象，跟 *POST* 比起來也比較跟實作無關。所以，在領域模

型中，我們應該用符號 *Register* 代替 *POST* 嗎？

首先，根據經驗法則，領域模型並沒有絕對的對或錯，只有比較有用或沒有用而已；它是我們溝通用的工具。

根據地圖繪製師原則，「*POST*」是領域中熟悉的術語，所以從熟悉與溝通的觀點來看，這是一個好的字。如果爲了產生代表抽象概念、跟實作無關的模型，*Register* 是比較有好也比較有用的術語【註】。因此 *Register* 可能是比較好的概念性類別，它同時代表紀錄銷售地點與各種終端機（例如 POST）的抽象概念。

【註】：請注意，以前 *register* 只是紀錄銷售的一種可能實作方式。隨著時間過去，這個術語已經有比較一般性的意義。

這兩種選擇都各自有優點：我們在這個個案研究中選用 *Register* 其實是有點武斷的，對關係人來說 *POST* 應該會比較容易瞭解。

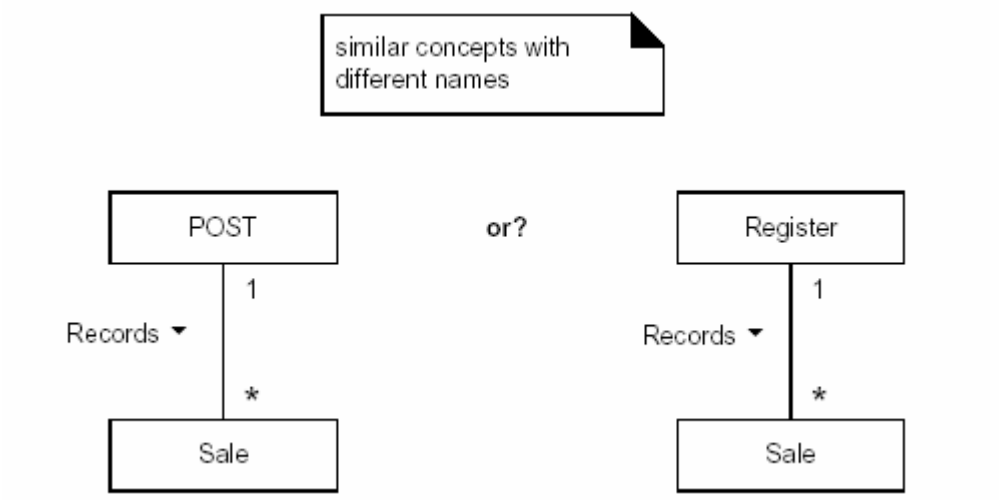


圖 10.6 POST 與 register 這兩個術語是很類似的概念性類別

第六節產生虛擬世界模型

有些軟體系統所針對的領域跟真實領域與企業領域之間幾乎沒有關係；通訊業的軟體就是其中一個例子。不過，我們還是可以針對這些領域產生領域模型，只過需要從熟悉的設計結果中回復到非常高的抽象概念程度。

舉例來說， Message 、 Connection 、 Port 、 Dialog 、 Route 與 Protocal 都是跟通訊交換機相關的候選概念性類別。

第七節規格型或描述型的概念性類別

接下來的討論，你剛開始看起來會認爲是很少見、非常特殊的議題。然而，最後大家會知道許多領域模型中其實都需要規格型的概念性類別。因此，我們這裡特別強調一下。

先做一些假設：

- *Item* 實例代表商店中的一個實體商品項目；有時候，它甚至可能是一個序列號。
- 每個 *Item* 都會有說明文字、價格與商品項目辨別碼，我們會把這些東西跟 *Item* 紀錄在一起。
- 在商店工作的人都可能健忘。
- 當我們賣出一個實體商品項目時，會從「軟體世界」中刪除相對應的 *Item* 軟體實例。

有了這些假設之後，接下來的情節中會發生什麼事？

很流行、新的蔬菜漢堡— ObjectBurger 的需求量很強勁。當商店中的蔬菜漢堡銷售一空時，電腦記憶體中所有的 *ObjectHurger Item* 實例也都會被刪除。

現在，問題的核心是：如果有人問「ObjectBurger 要賣多少錢？」將沒有人可以回答這個問題，因爲它們的價格（存放在記憶體中）都跟庫存實例連接在一起。因爲實例已經被賣完，所以所有的價格也都被刪掉了。

請再注意一下，這個模型的軟體如果是用這種方式實作的，我們將會有重複的資料，造成空間不足，因爲同樣產品的說明文字、價格與商品項目辨別碼會不斷在 *Item* 的實例中重複。

何時需要規格型或說明型的概念性類別

前面所展現的問題讓我們知道紀錄其它東西規格或說明文字的物件概念是需要存在的。爲了解決 *Item* 問題，我們需要有像 *ProductSpecification*（或 *ItemSpecification* 、 *ProductDescription* 等等）的概念性類別紀錄關於各個商品項目的資訊。 *ProductSpecification* 並不等於 *Item* ，它代表商品項目的說明資訊。請注意，縱然我們已經賣完所有庫存商品項目，跟它們相對應的 *Item* 軟體實例也都會被刪除， *ProductSpecification* 還是會存在。

說明型物件或規格型物件會跟它們所描述的東西產生很強的關聯性。在領域模型中，我們通常會說 *XSpecification 是用來描述 X 的*（請參見圖 10.7。）

在銷售與產品領域中，我們常常需要用到規格型概念性類別。製造業中也常常會用它描述製造出來的東西，以便跟製造出來的東西區隔開來。時間與空間都有可能是我們使用規格型概念性類別的動機，因爲這些說明文字常常會重複；這不是少見的建立模型概念。

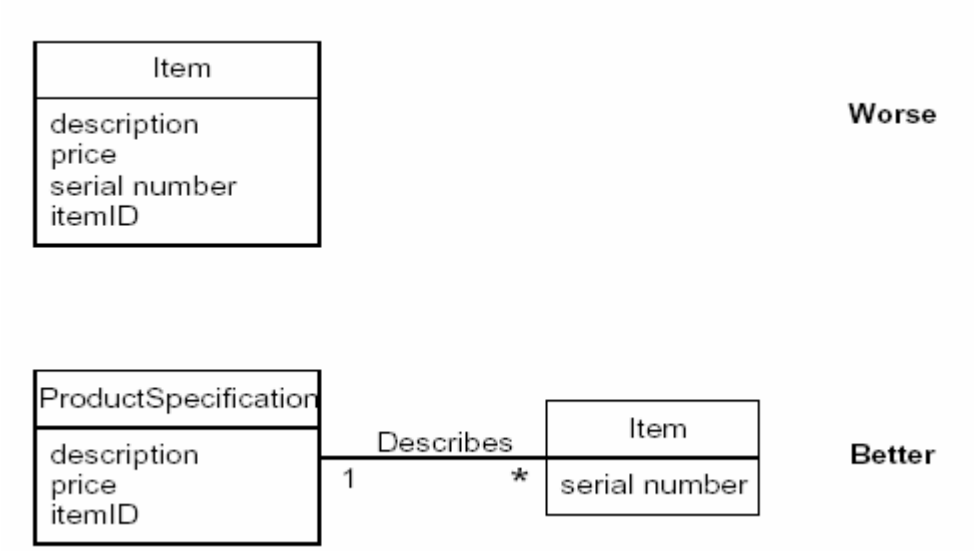


圖 10.7 跟其它東西有關的規格或說明文字。「*」代表「多個」的多重性。它說明一個 *ProductSpecification* 可能會描述多個（*） *Item* 。

何時需要規格型概念性類別？

接下來的指引建議我們何時使用規格型概念性類別：

當我們遇到下面情況時，會使用規格型或說明型概念性類別（例如 *ProductSpecification* ）：

- 某個商品項目或服務需要一段說明文字，它跟目前現存的任何商品項目或服務實例之間是無關的。
- 刪除它們所描述的實例（例如 *Item*）時會失去我們需要維護的一些資訊，原因是我們要刪除的東西跟描述它們的資訊之間有不正確的關聯。
- 它可以減少多餘資訊或重複資訊。

其它規格型概念性類別的例子

我們現在考慮另一個例子，假設有一家航空公司的班機發生嚴重墜機事件，並且假設六個月內的所有班機都要被取消掉以完成調查工作。還假設：當班機被取消時，相對應的 *Flight* 軟體物件也要從電腦記憶體中刪掉。因此，發生墜機事件後，所有的 Flight 軟體物件都會被刪掉。

如果我們只在 Flight 軟體實例中紀錄班機要飛的機場，裡面代表某個特定日期與時間的特定班機，當 Flight 軟體物件都被刪掉後，就沒有記錄有記載班機的航程。

爲了解決這個問題，不論是否有排特定班機，我們都需要用 FlightDescription（FlightSpecification）紀錄班機與航程。

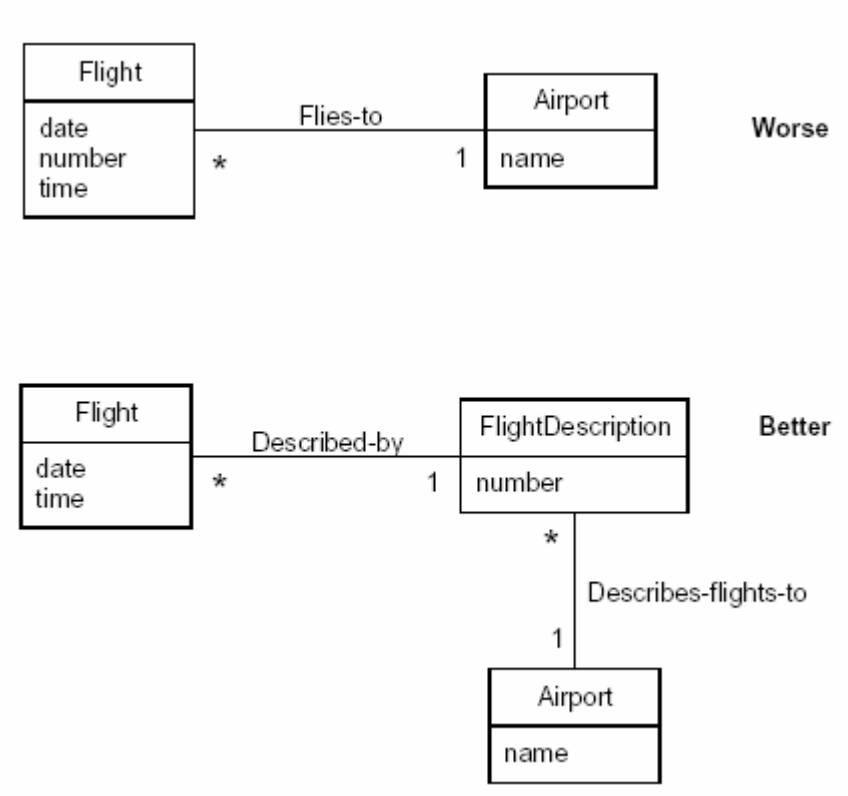


圖 10.8 紀錄其它東西用的規格型概念性類別

服務的說明

請注意之前的例子是用來紀錄跟服務有關的說明（班機）而不是貨物（例如蔬菜堡。）我們常常需要對服務或服務計劃做說明。

再舉另一個例子，行動電話公司銷售一些套件例如「銅級」、「金級」套件等等。我們需要把套件說明(一種服務計劃，裡面描述每分鐘費率、無線網際網路內容、成本等等）跟真正賣出去的套件（例如在 2002 年 1 月 1 日以每月 $55 賣金級套件給 Craig Larman）區分開來。在銷售之前，行銷人員需要先定義並紀錄這個服務計劃或 MobileCommunicationPackageDescription 。

第八節 UML 表示法、模型與不同的開發方法：

多重觀點

UP 中定義叫做領域模型的東西，並且用 UML 表示法展示這個模型。然而，在 UML 正式說明文件中並沒有一個叫做「領域模型」的術語。從這裡，我們可以知道很重要的一點：

UML 中只簡單描述原始的畫圖型態，例如類別圖與循序圖。裡面並沒有附上某

種方法論或建模型時的觀點。然而，開發流程（例如 UP）則會把原始 UML 應用在方法論者所定義的模型情境上（譯註：爲了應用原始 UML，我們通常會用造型【stereotype】在原始 UML 表示法上加上新的語意，例如企業使用案例就是加上造型的使用案例，特別用在企業模型中的使用案例。此外，也可以把原始 UML 表示法視爲官方語言，而加上造型後的表示法則視爲方言。）

舉例來說，原始 UML 畫類別的表示法可以代表領域概念性類別（領域模型）、軟體類別、關聯資料庫資料庫表格等等。

因此，不要把基本 UML 圖形表示法跟視覺化方法論者所定義的各種模型混淆在一起（請參見圖 10.9。）這一點不只適用在 UML 的類別圖，也可以適用在大部分的 UML 表示法上。

UML 的循序圖另一個例子，我們也是可以在不同模型中用不同方式解釋原始 UML 畫圖的表示法。原始循序圖是用來展示軟體物件間的訊息（例如 UP 中的設計模型），也可以用來展現真實世界中人們與當事人之間的互動情況（例如 UP 中的企業物件模型。）

Syntropy 物件導向方法論【CD94】中特別強調這樣的觀點，並且 Martin Fowler 在 UML 精華（UML distilled）【FS00】中也重複強調這樣的觀點。換言之，我們可以把同樣的畫圖表示法用在三種不同觀點與（譯註：各種方法論定義的）模型型態上：

1. **本質性或概念性觀點**（essential or conceptual perspective）—我們用這些圖描述真實世界或有興趣領域中的東西。
2. **規格觀點**（specification perspective）—我們用這些圖（跟本質性模型中的表示法相同）描述軟體抽象概念或元件的規格或介面，不過這裡並沒有承諾要用哪種特定實作方式（例如我們並沒有指定類別是用 C# 或 Java 實作的。）
3. **實作觀點**（implementation perspective）—我們用這些圖（跟本質性模型中的表示法相同）描述軟體實作方式是用某種特定技術與程式語言實作的（例如 Java。）

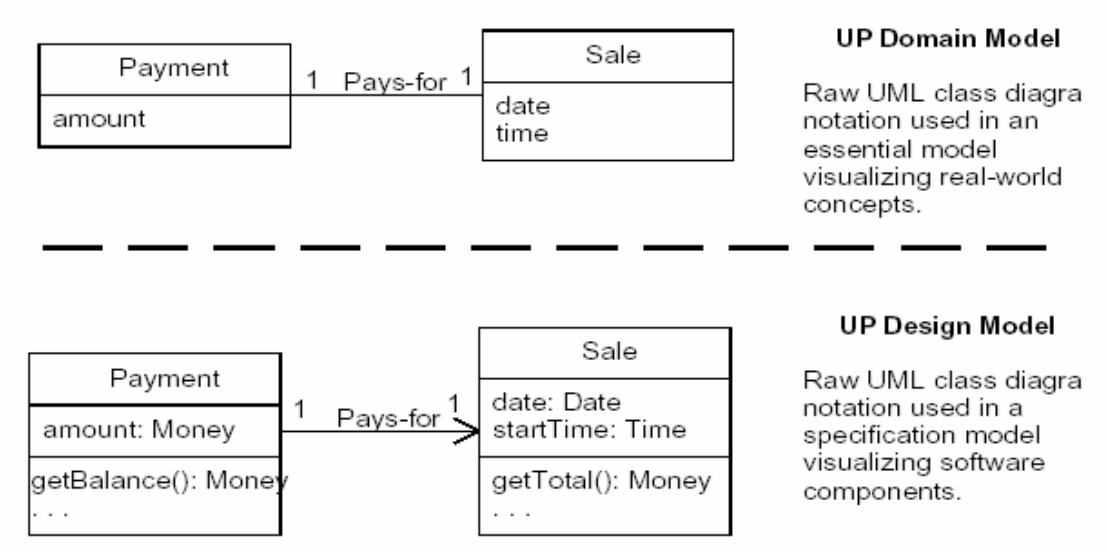


圖 10.9 把原始 UML 表示法應用在不同觀點與開發流程或方法論所定義的模型上。

被不斷附加意義的術語： UML vs. 方法論

在原始的 UML 中，圖 10.9 所顯示的方形稱爲類別（class）。不過請注意，在 UML 中這個術語涵蓋很多種現象—實體東西、軟體東西、事件等等【註】。開發流程或方法論會附加其它術語在 UML 上面。舉例來說，在 UP 裡面，當我們把 UML 的方形畫在領域模型時，這個方形就被稱爲領域概念（domain concept）或概念性類別（conceptual class）；領域模型提供概念性觀點。在 UP 中，當 UML 方形被畫在設計模型時，它們的正式稱呼則是設計類別（design class）；建模型者在描述設計模型時，可能是使用規格觀點或實作觀點。

【註】：UML 的類別是更爲一般性的 UML 模型元素：**有行爲者**（classifier）—具有結構特性與／或行爲的某種東西，例如類別、參與者、介面與使用案例—的特例。

不論定義如何，我們的底線是把分析師看到的真實世界概念觀點（例如一筆銷售，它屬於概念性觀點）跟軟體設計師看到的軟體元件（例如 *Sale* 軟體類別，它屬於規格觀點或實作觀點）區隔開來，這樣做是很有幫助的（譯註：簡單來說就是分析跟設計的區別，就譯者的個人經驗來說，兩者之間的差距還是很大。一般來說，我們會在分析時產生領域模型，而在設計時產生設計模型，不過有必要的話，還可以在領域模型與設計模型之間產生分析模型。領域模型中會存在企業參與者【business actor】、企業工作者【business worker】與企業實體【business entity】；分析模型中會有由企業參與者或企業工作者演變而來的參與者，以及由

企業實體演變而來的邊界類別【boundary class】與控制類別【control class】與實體【entity】；設計模型中則只包含參與者與類別。）
因爲 UML 展現兩種不同觀點時的表示法與術語很類似，所以建模型者要時時把自己採用的觀點牢記在心。

爲了讓不同觀點更清楚，本書採用下面一些跟類別相關的術語，這些術語跟 UML 與 UP 的稱呼都一致：

- **概念性類別**—真實世界的概念或東西。採用概念性觀點或本質性觀點。UP 中的領域模型會包含概念性類別。
- **軟體類別**—採用規格觀點或實作觀點，代表軟體元件的類別，跟開發流程或方法論無關。
- **設計類別**—UP 設計模型中的一個組成元素。它是軟體類別的同義字，不過爲了某種理由，我強調說明它是設計模型中的類別。UP 中允許設計類別可以根據建模型者的需要採用規格觀點或實作觀點。
- **實作類別**—用物件導向程式語言實作的類別，例如 Java 。
- **類別**—在 UML 中，這個一般性的術語既可以代表真實世界中的事物（概念性類別），也可以代表軟體事物（軟體類別。）

第九節減少觀點呈現差距

請看一下圖 10.10 。爲什麼有些書與教育者討論物件設計時通常只討論反映出領域字彙表名稱的軟體類別？爲何把軟體類別的名稱叫做 *Sale* 呢？還有 *Sale* 要做什麼事呢？
簡單來說，用領域字彙表裡面的名稱（*Sale*）命名可以讓我們很容易瞭解軟體類別，也可以提供我們線索了解 *Sale* 軟體類別程式碼裡面會做什麼事。我們會對問題領域（例如商店銷售商品）產生一個心智模型或領域模型。在真實世界中，我們知道銷售會有日期。因此，如果我們產生 *Sale* 的 Java 類別，並且真實銷售與日期的責任指派給它，那麼 Java 類別 *Sale* 裡面應該就跟真實領域的心智模型或領域模型相對應；換言之，這個類別導源於領域的直覺知識。

領域模型讓我們用視覺方式呈現領域字彙表與概念字典，當我們設計軟體時，可以把它當作替某些東西命名時的靈感來源。

這一點跟**觀點呈現差距**（representation gap）有關，差距發生的原因在於領域的心智模型與軟體中的呈現方式不同。

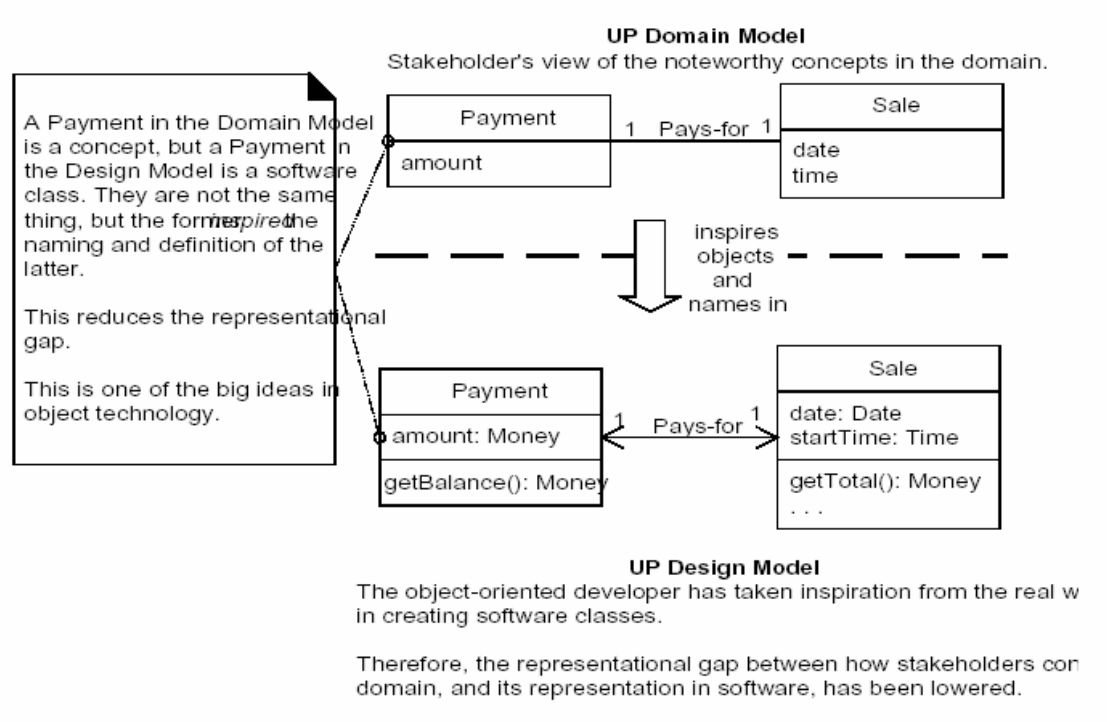


圖 10.10 當我們進行物件設計與寫程式時，通常會從真實世界領域中尋找軟體類別的命名方式與資訊。

最極端的一種方式就是寫 NextGen POS 應用程式程式時，直接用原始二進位碼呼叫處理器的指令集。這種做法所造成的觀點呈現差距是很大的，而且軟體發生巨幅的觀點呈現差距時，其真實成本是很難估計的，因爲這樣的成本很難理解，也很難跟問題領域關聯在一起。從另一個方式來看，物件技術讓我們把程式碼聚集在一塊變成類別，類別可以反映出我們所認知的領域。舉例來說，在真實世界中，我們認知到有一塊東西（或事件）叫做 *Sale* ，所以在軟體世界裡，我們也有一個軟體類別稱爲 *Sale* 。把領域字彙表跟軟體字彙表之間用一對一方式對應，並且用同樣方式把程式碼聚集在一起，可以降低觀點呈現差距。這種做法加快我們對現存程式碼的理解（因爲程式碼用我們預期的方式運作，而我們了解領域知識），而且可以用很「自然」的方式擴充程式碼，讓程式碼跟領域之間用類似方式對應或讓程式把導源於領域的直覺知識。簡單來說，軟體模型讓我們回想起概念性模型或心智模型，並且用可預測的方式運作。

觀點呈現差距小的軟體模型有實際上的好處。雖然很難明確說出這個好處爲何，不過大部分軟體工程師都知道這個講法是真的。事實上，有一個證據存在，就是 Java 混亂器會改變 Java 類別與方法名稱讓它們難以理解、不符合領域直覺知識，這樣一來雖然沒有辦法改變控制與資料結構，二位元碼反向工程所得到的原始碼也難以理解。

當然，物件技術還有一種價值，就是它讓我們設計出精鍊、耦合力低的系統，這樣的系統很容易擴充變大，本書後面其它部分會探討這個部份。比較小的觀點呈現差距是有用的，不過物件很容易改變、擴充、管理、隱藏複雜度則是主要的優

點。

第十節範例：NextGen POS 系統中的領域模型

我們可以用一開始的領域模型、以圖形方式呈現出 NextGen POS 領域中的概念性類別清單。

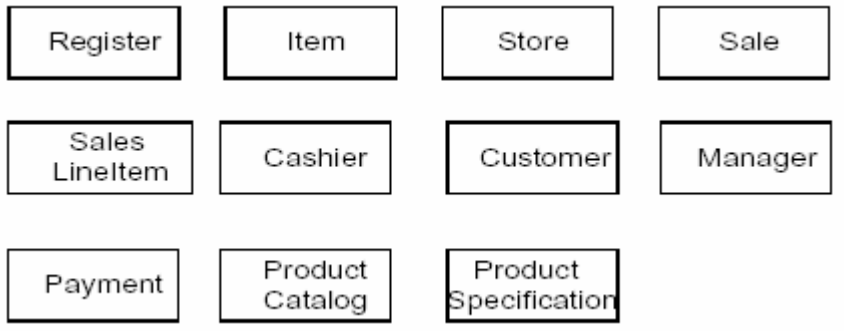


圖 10.11 一開始的領域模型

後面章節會討論這個領域模型中屬性與關聯方面的考量。

第十一節UP 中的領域模型

如同表 10.2 的範例所建議的，我們通常在詳述階段開始領域模型並完成它。

初始階段

我們並不會很想在初始階段中開始製作領域模型，因爲初始階段的目的不是要進行正式調查工作，而是要決定專案是否值得在詳述階段中做更深入的調查工作。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	**領域模型**		S		
需求	使用案例模型(SSD)	s	R		
	專案願景	s	R		
	輔助規格書	s	R		
	字彙表	s	R		
設計	設計模型		S	r	
實作	軟體架構文件		S		

	資料模型		s	r	
	實作模型		s	r	r
專案管理	軟體開發計畫	s	r	r	r
測試	測試模型		s	r	
環境	開發案例	s	r		

表 10.2 可能用到的 UP 工作成果以及產生它們的時間點。s — 初版；r — 修正版

詳述階段

領域模型主要是在詳述階段的反覆中製作的，這時候我們需要了解最值得注意的概念，並且在設計時把某些概念對應到軟體類別上。

雖然有些諷刺的是我們需要花好多頁才能解釋如何建立領域物件模型，不過對有經驗的開發人員來說，在每個反覆中開發（部分或漸增式）領域模型時，往往只需要花幾個小時。如果使用事先定義好的分析樣式，更可以縮短建立領域模型的時間。

UP 中的企業物件模型 vs. 領域模型

UP 中的領域模型是 UP 中比較少見的企業物件模型（BOM）的一種正式變異。UP 中的 BOM —不要跟其他人或其它方法論所定義的 BOM 搞混，在這裡同樣的語有不同的意義—是描述整個企業的企業模型。不論我們在開發哪一種軟體應用程式，進行企業流程工程或企業流程重整時都可以用到它。引述：

> 【UP 中的 BOM 】可以視爲描述企業工作者與企業實體之間關係以及如何合作達成企業目標的抽象概念【RUP】。

BOM 可以呈現在幾個不同的圖中（類別圖、活動圖與循序圖），說明整個企業是如何運作的。如果我們正在進行涵蓋整個企業範圍的企業流程工程，那麼 BOM 就非常有用，不過如果我們只是要產生單一軟體應用程式，那麼就很少會產生 BOM 。

因此，UP 中定義更常製作的領域模型，以作爲 BOM 的次要工作成果或規格。引述：

> 你可以選擇性的發展「不完整」的企業物件模型，把焦點放在解釋領域中重要「東西」或產品上…我們通常把這樣的東西稱爲領域模型【RUP】。

第十二節進階讀物

Odell 的 *Object-Oriented Methods: A Foundation* 裡面對建立概念性領域模型有

很紮實的介紹。 Cook 與 Daniel 的 *Designing Object Systems* 也是很有用的一本書。

Fowler 的 *Analysis Patterns* 則提供領域模型中很有價值的樣式，而且這也是我們極力推薦的一本好書。另一本好書是描述領域模型中樣式、由 Hay 撰寫的 *Data Model Patterns: Conventions of Thought*。如果建立資料模型專家能了解純粹概念性模型與資料庫綱目之間的差異，那麼他的忠言對我們建立領域模型來說很有幫助。

Java Modeling in Color with UML 【CDL99】這本書對建立領域模型的建言比書名看起來還要多。作者在書中找出常見樣式的型態與關聯；至於顏色方面則是以視覺方式替型態分類的一種工具，例如 descriptions （藍色）、roles（黃色）與 moment-intervals（粉紅色。）顏色是幫助我們區別樣式的工具。

從 Abbot 開始，現在的語言分析技術已經變成比較複雜的物件導向分析技術了。一般來說，我們把它稱爲建立自然語言模型技術。請參見【Moreno97】，它是一個很好的例子。

第十三節UP 中跟領域模型相關的工作成果

圖 10.12 所展示的是跟領域模型相關、受到影響的工作成果。

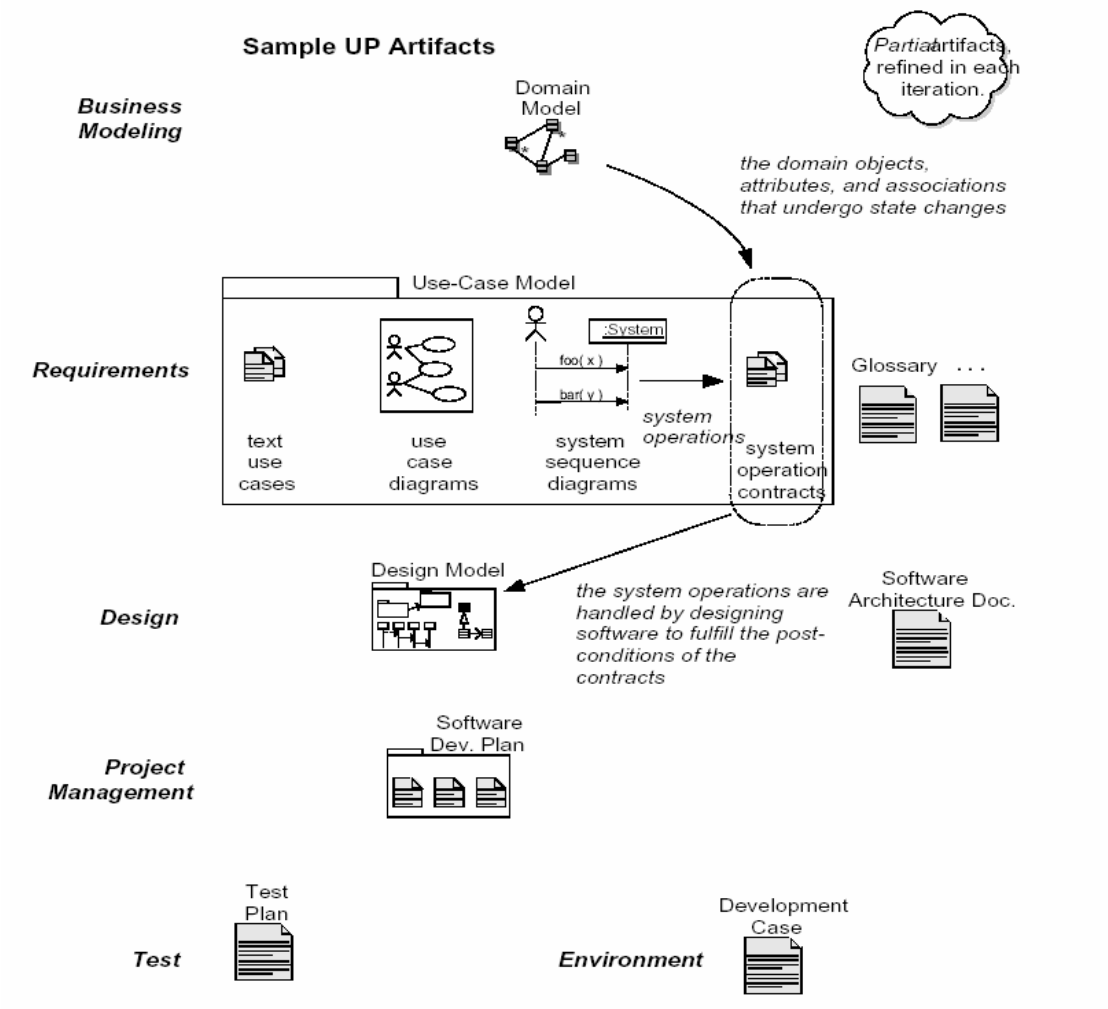


圖 10.12 UP 中跟領域模型相關、受到影響的工作成果。

第十一章領域模型：加入關聯

本章目標

- 找出領域模型中的一些關聯。
- 區別「必須知道」的關聯與「瞭解專用」的關聯。

簡介

爲了滿足目前發展中情節的資訊需求而找出概念性類別的關聯是一件很有用的事，有助於我們了解領域模型。本章裡面探索如何找出合適的關聯，並且針對 NextGen 個案研究，在它的領域模型中加入關聯。

第一節關聯

關聯（association）代表型態（或者更正式一點，這些型態的實例）之間的關係以指出一些有意義、有興趣的連結（請參見圖 11.1。）
在 UML 中，關聯被定義成「兩個或多個有行爲者（classifier）之間、具有語意的關係，這個關係跟這些有行爲者實例的連結有關。」

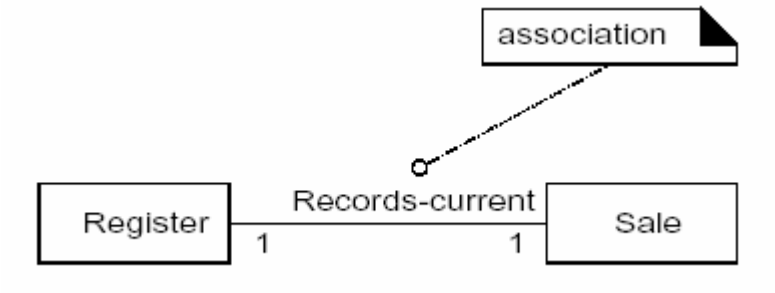


圖 11.1 關聯

選出有用關聯的準則

值得注意的關聯通常是我們根據情境需要，保存一段時間—可能從數微秒到幾年—的關係相關知識。換言之，我們需要記憶哪些物件之間的關係呢？例如我們需要記憶 *SalesLineItem* 實例與 *Sale* 實例之間的關聯嗎？絕對需要，不然我們將無法重建一筆銷售、列印收據或計算銷售總金額。
請考慮在領域模型中加入下面這樣的關聯：

- 關聯是需要保存一段時間的關係知識（「必須知道」的關聯。）
- 利用常見關聯清單找到的關聯。

爲了做一個對照，我們想問：你需要紀錄目前 *Sale* 與 *Manager* 之間的關係嗎？不需要，因爲需求中不需要紀錄這類型的任何關係。雖然秀出 *Sale* 與 *Manager* 之間的關係並沒有錯，不過在現在的需求情境中，這個關係沒有強制性或用處。這是很重要的一點。在有 n 個不同概念性類別的領域模型中，最多可能會有 n(n-1) 個關聯存在。圖中存在太多線條將增加「視覺上的困擾」，讓圖變得更難理解。因此，我們應該用本章所建議的準則，盡量減少圖中的關聯線條。

第二節 UML 中的關聯表示法

關聯是用類別之間的線條來呈現的，線條上面還會有關聯名稱。本質上，關聯是雙向的，也就是說，邏輯上我們可以從任一類別的實例瀏覽到另一個類別的實例。這裡的瀏覽純粹是抽象的；不代表軟體實體之間會有連結存在。

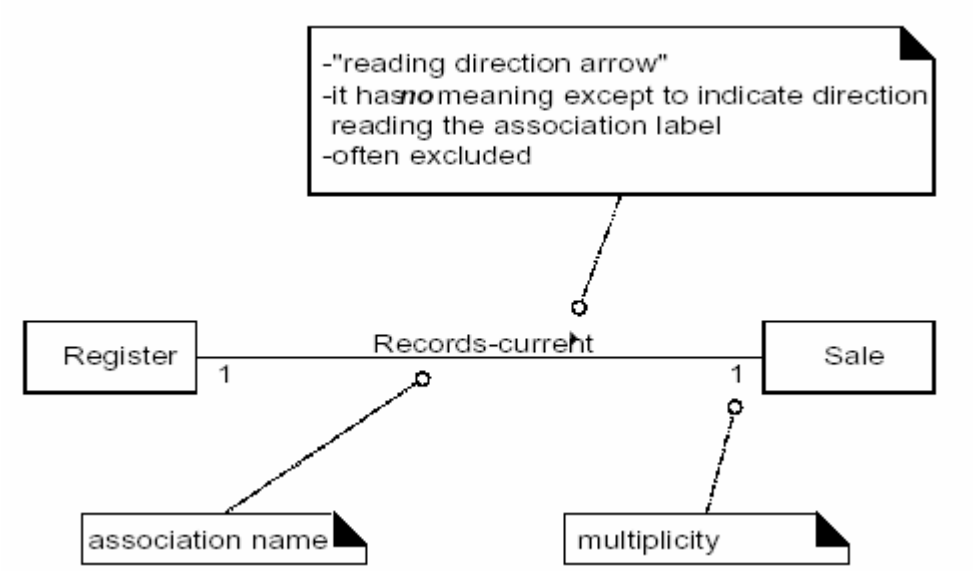


圖 11.2 UML 中代表關聯的表示法

關聯的端點上可能會有多重性以指明類別實例之間的關係數。

一個可選用的「閱讀方向箭頭（reading direction arrow）」讓我們知道閱讀關聯名稱時的方向；它不代表可見性或可瀏覽性方向。

如果沒有讀取方向的話，習慣上我們會從左而右、從上而下閱讀關聯，不過 UML 中沒有定出這樣的規則（請參見圖 11.2。）

閱讀方向箭頭對模型來說沒有什麼意義；它只是方便我們閱讀圖而已。

第三節找出關聯－常見關聯清單

我們一開始可以利用表 11.1 中的清單找出關聯、把關聯加入圖中。

清單中包含常見、值得注意的一些關聯分類。範例則是從商店領域與航空訂位領域中找出來的。

分類	範例
在實體上，A 是 B 的一部份	Draw—Register（或者更明確一點 POST） Wing—Airplane
在邏輯上，A 是 B 的一部份	SalesLineItem—Sale FlightLeg—FlightRoute
在實體上，A 被包含在 B 裡面	Register—Store、Item—Shelf Passenger—Airplane
在邏輯上，A 被包含在 B 裡面	ItemDescription—Catalog Flight—FlightSchedule
A 是 B 的說明文字	ItemDescription—Item FlightDescription—Flight
A 是交易或報告 B 的明細	SalesLineItem—Sale MaintenanceJob—MaintenanceLog
A 被 B 知道／紀錄／報告／捕捉起來	Sale—Register Reservation—FlightManifest
A 是 B 的成員之一	Cashier—Store Pilot—Airline
在組織上， A 是 B 的次級單位	Department—Store Maintenance—Airline
A 會用到或管理 B	Cashier—Register Pilot—Airplane
A 會跟 B 溝通	Customer—Cashier ReservationAgent—Passenger
A 跟交易 B 有關係存在	Customer—Payment Passenger—Ticket
A 是跟交易 B 有關係的另外交易	Payment—Sale Reservation—Cancellation
A 在 B 的鄰近位置	SalesLineItem—SalesLineItem City—City
A 被 B 所擁有	Register—Store Plane—Airline
A 是跟 B 有關的事件	Sale—Customer、Sale—Store Departure—Flight

表 11.1 常見關聯清單

優先順序高的關聯

下面是一些優先順序高的關聯分類，我們通常會把它們放在領域模型中：

- 在實體上或邏輯上，A 是 B 的一部份。
- 在實體上或邏輯上，A 被包含在 B 裡面。
- A 被 B 紀錄起來。

第四節關聯指引

- 把重點放在關係知識需要保存一段時間的關聯上（「必須知道」的關聯。）
- 找出*概念性類別*比找出關聯還要重要。
- 太多的關聯反而會混淆領域模型，其實並沒有多展現出什麼東西。找出這些關聯需要花很多時間，請尋找有邊際效益的關聯。
- 避免秀出多餘或可以從其它關聯推導出來的關聯。

第五節角色

我們把關聯的端點稱爲**角色**（role）。下面是角色可能具備的屬性：

- 名稱
- 多重性（multiplicity）表示式
- 可瀏覽性（navigability）

我們接下來討論多重性，其它兩種特性也會在後面章節中討論。

多重性

多重性（multiplicity）定義類別 A 會有多少實例跟類別 B 的一個實例關聯在一起（請參見圖 11.3。）

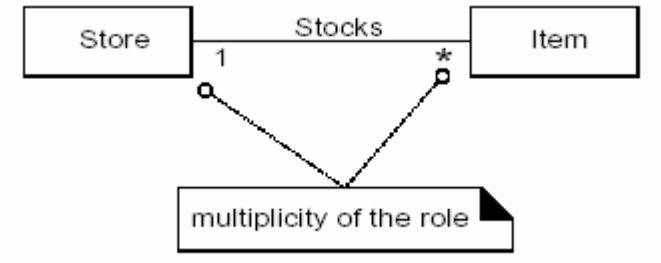


圖 11.3 關聯上的多重性

舉例來說， *Store* 的單一實例可以關聯到「多個」(零個或多個，用 * 表示） *Item* 實例上。

圖 11.4 是一些多重性表示式的範例。

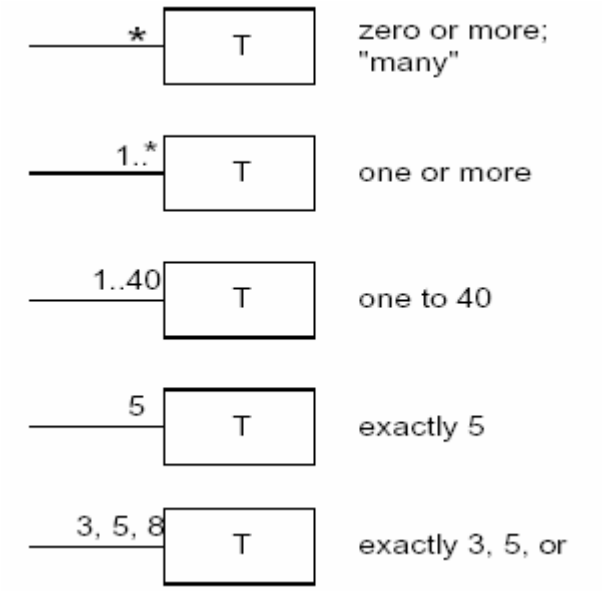


圖 11.4 多重性的一些可能值

多重性的值告訴我們有多少個實例（在某個特定時候，而不是一段時間內）是可以關聯到另外的實例上。例如在一段時間內，一台二手車可能被賣回二手車交易商很多次，不過在某個特定時候，這輛車只會被一家交易商*庫存起來*（Stocked-by），而不會被多家交易商*庫存起來*。同樣地，許多國家都制定一夫一妻制法律。在某個特定時候，一個人只能跟另外一個人*結婚*（Married-to），經過一段時間後，他們則可能跟多個人結婚過。

多重性的值會隨著建模型者與軟體開發人員有興趣的地方而異，因爲多重性可以告訴我們領域限制，而且這個限制會反映在軟體上。請參見圖 11.5 裡面的範例與說明文字。

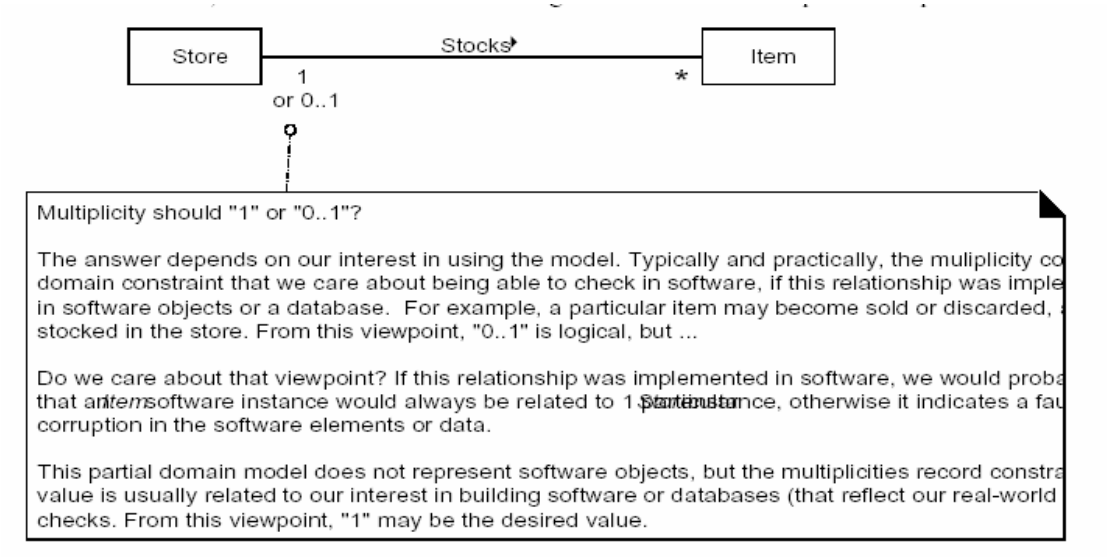


圖 11.5 多重性是跟情境有關的

Rumbaugh 舉了另一個跟 *Person* 與 *Company* 有關的 *Works-for* 關聯範例【Rumbaugh91】。一個 *Person* 的實例是否替一個或多個 *Company* 實例工作是

要看模型情境而定的；賦稅單位會對*多個*實例有興趣；協會則可能只對*一個*實例有興趣。在某個實作中，多重性的選擇通常要看我們是爲誰建軟體而決定出有效的多重性。

第六節關聯應該要描述地多詳細？

關聯很重要，不過當我們建立領域模型時常常陷入一個陷阱：就是花太多時間在發掘關聯上。

下面的概念是很有關鍵性的：

> 找到概念性類別比找到關聯還重要。當我們產生領域模型時，應該把大部分時間花在找出概念性類別而不是關聯上。

第七節替關聯命名

> 根據*型態名稱—動詞片語—型態名稱*的格式替關聯命名，其中動詞片語會產生模型情境下可讀、有意義的結果。

關聯名稱應該要用大寫字母開始，因爲關聯代表實例之間的一種有行爲者：繫結（link）；在 UML 中，有行爲者應該要用大寫字母開頭。有兩種常見到、意義相當的複合式關聯名稱的合法格式是：

- *Paid-by*
- *PaidBy*

在圖 11.6 中，閱讀關聯名稱時的預設方向是從左到右、從上到下。這不是 UML 中的預設規則，不過是常見的一種慣例。

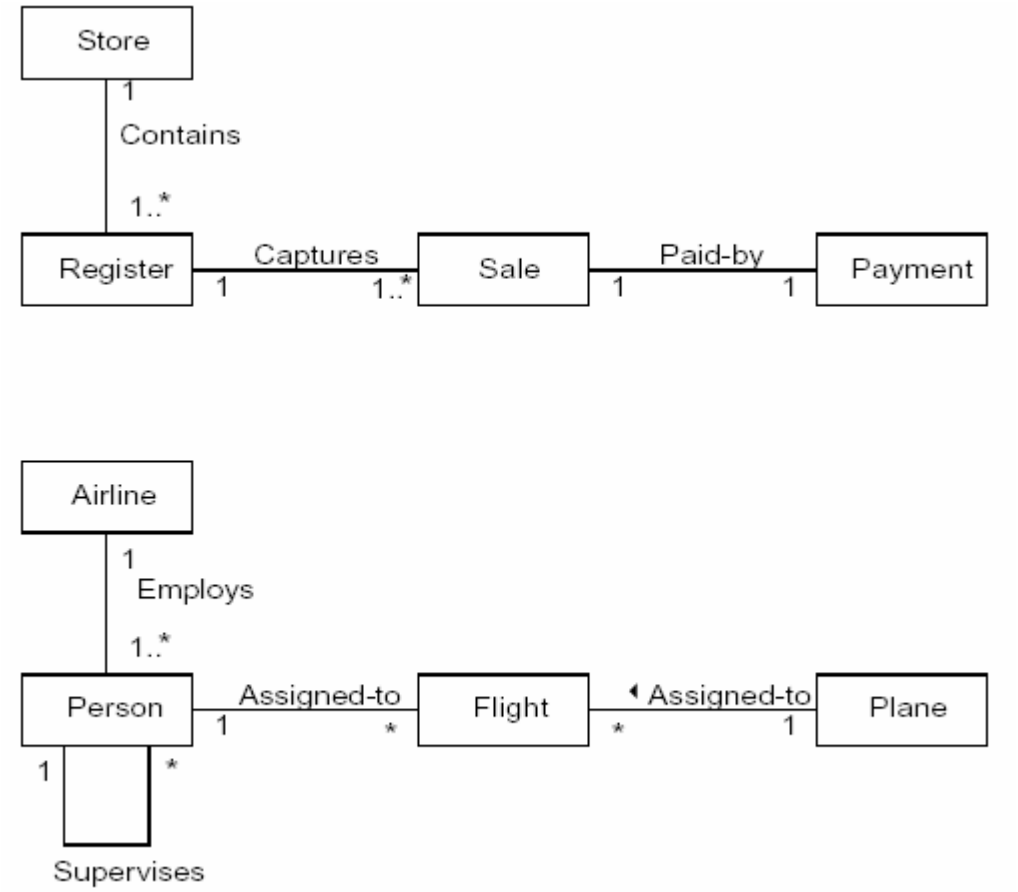


圖 11.6 關聯名稱

第八節兩個型態之間的多重關聯

兩個型態之間可能會存在多重關聯；這是常發生的情形。我們的 POS 個案研究裡面沒有這種特別的例子，不過在航空領域裡面 Flight （或者更精確一點 FlightLeg ）跟 Airport 之間的關係則是一個這樣的例子（請參考圖 11.7）；其中的 flying-to （飛到）與 flying-from（飛自於）關聯分別代表不同關係，我們應該分別顯示出來。

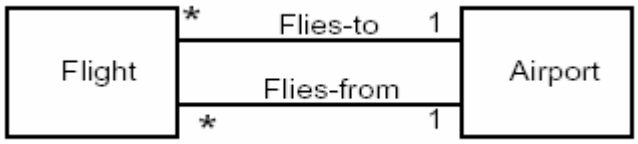


圖 11.7 多重關聯

第九節關聯與實作

在建立領域模型時，關聯*不是*在說明軟體解決方案中的資料流、實例變數或物件連結爲何；它純粹描述從概念來看—真實世界中—有意義的關係。按實際情況來

說，許多關係通常都是在軟體中被實作成瀏覽路徑與可見性（分別在設計模型與資料模型中），不過領域模型中的概念性觀點並不需要呈現這些關聯的實作方式。當我們產生領域模型時，可能會定義出實作時不存在的關聯。相反地，我們可能會發現有些需要實作的關聯，建立領域模型時沒有被找出來。發生這些情況時，我們可以更新領域模型、反映出這些新發現。

建議

之前的一些調查模型（例如領域模型）應該隨著實作過程中對系統的越來越多瞭解（例如新關聯）而更新嗎？除非將來會用到這個模型，不然不要擔心這個問題。如果之前的調查用模型只是暫時性的工作成果（通常是這種情況），而且只是爲了提供後面開發步驟的靈感，之後就沒有什麼意義，爲什麼要更新它呢？除非有明確的證據知道未來會用這個工作成果，不然應該不用製作或更新這些文件或模型。

稍後，我們將討論用物件導向程式語言實作關聯的方式（最常見的方式就是用一個屬性去參考被關聯類別的實例），不過我們現在最好還是先把它們當成純粹的概念性表示式，而不是資料庫或軟體解決方案的說明。跟往常一樣，延後考量因素讓我們可以只做純粹的「分析」調查工作以免除額外的資訊與決策，並且讓稍後的設計結果有最大的彈性。

第十節 NextGen POS 系統領域模型中的關聯

現在可以在 POS 領域模型中加入關聯了。我們應該把需求（例如使用案例）中建議應該要有、隱含要記憶，或者問題領域認知中強烈建議的關聯加入領域模型中。探討新需求時，我們應該重新審查並考量之前介紹過的常見關聯分類。一般來說，它們是許多需要紀錄的關聯。

商店中不可忽略的關係

接下來的關聯範例是我們必須知道的關聯。以目前考量中的使用案例爲基礎找出這些關聯。

Register Records Sale	爲了知道目前銷售、產生總銷售金額、列印收據。
Sale Paid-by Payment	爲了知道這筆銷售是否已經付過錢、把付款金額與銷售總金額關聯在一起、列印出收據。
ProductCatalog Records ProductSpecifications	爲了根據給定的 itemID 找到 ProductSpecification 。

應用關聯分類檢核單

接下來，我們會以之前找出來的關聯型態、目前考量中的使用案例需求爲基礎來應用這份檢核單。

分類	範例
在實體上，A 是 B 的一部份	Register—CashDrawer
在邏輯上，A 是 B 的一部份	SalesLineItem—Sale
在實體上，A 被包含在 B 裡面	Register—Store Item—Store
在邏輯上，A 被包含在 B 裡面	ProductSpecification—ProductCatalog ProductCatalog—Store
A 是 B 的一種說明	ProductSpecification—Item
A 是交易或報告 B 的明細	SalesLineItem—Sale
A 被 B 知道／紀錄／報告／捕捉起來	（完整的）Sales—Store （目前的）Sale—Register
A 是 B 的成員之一	Cashier—Store
在組織上， A 是 B 的次級單位	沒有合適的例子
A 會用到或管理 B	Cashier—Register Manager—Register Manager—Cashier，不過可能不適用。
A 會跟 B 溝通	Customer—Cashier
A 跟交易 B 有關係	Customer—Payment Cashier—Payment
A 是跟交易 B 有關係的另外一個交易	Payment—Sale
A 在 B 的鄰近位置	SalesLineItem—SalesLineItem
A 被 B 所擁有	Register—Store

第十一節NextGen Pos 系統中的領域模型

圖 11.8 中的領域模型是 POS 應用程式的候選概念性類別與關聯。這些應用程式主要是從候選關聯檢核單來的。

需要把必須知道的關聯保存下來嗎？

圖 11.8 所秀出來的關聯中大部分都是機械式的從關聯檢核單找到。然而，我們希望用更嚴格的方式決定是否要把關聯放到領域模型。因爲領域模型被視爲溝通工具，所以我們不希望領域模型中充斥非必要性的關聯或無法展現我們對領域瞭

解的關聯。太多非必要性的關聯只會模糊領域模型焦點而無法讓領域模型變得更清晰。

如同之前所建議的一樣，我們會根據下面準則決定某些關聯是否需要秀出來：

- 把重點放在關係知識需要保存一段時間的關聯上（「必須知道」的關聯。）
- 避免秀出多餘或可從其它關聯推導出來的關聯。

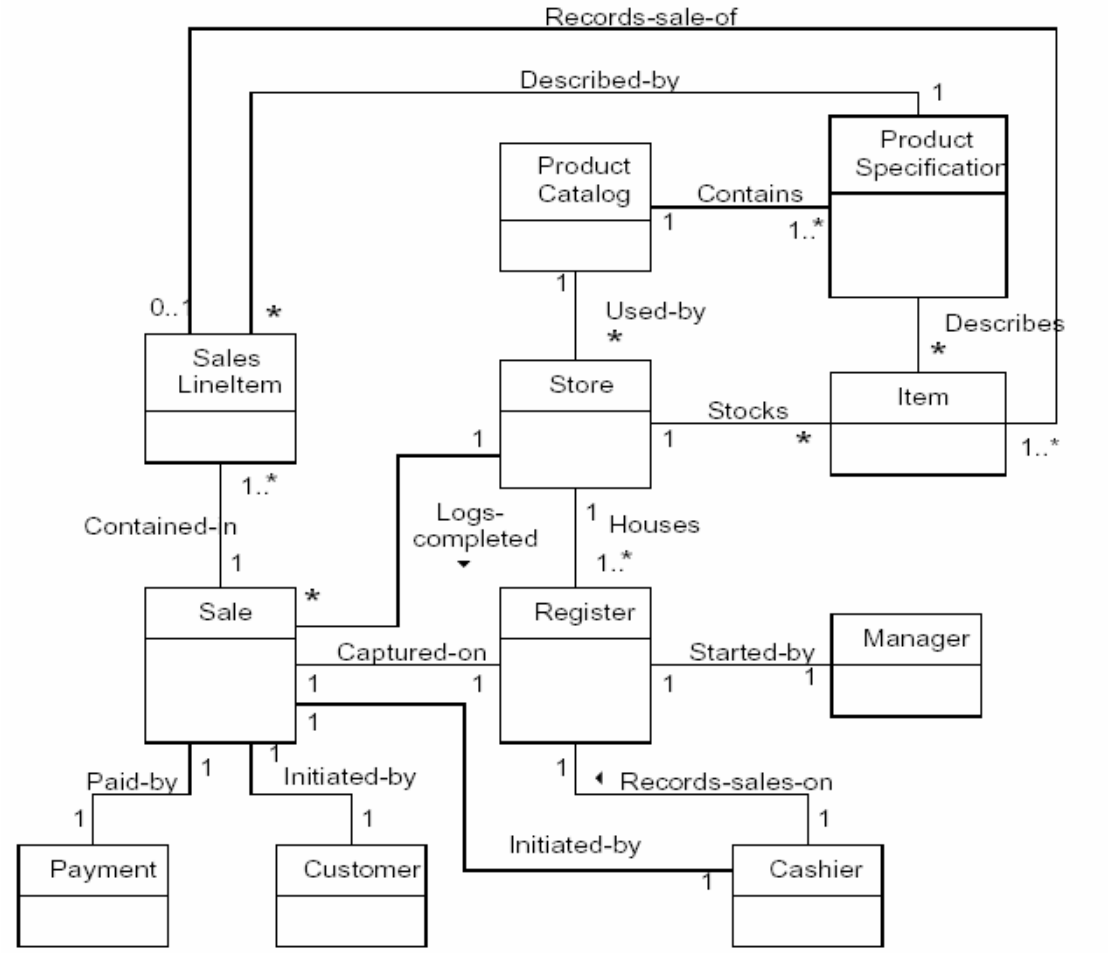


圖 11.8 領域模型的一部份

根據這個忠告，目前所秀出來的關聯並非都是必要性的。考慮下面這些關聯：

關聯	討論
Sale Entered-by Cashier	需求中沒有指出必須知道或紀錄目前的收銀員爲誰。此外，如果 *Register Used-by Cashier* 關聯已經存在的話，我們也可以從它推導出這個關聯。
Register Used-by Cashier	需求中沒有指出必須知道或紀錄目前的收銀員爲誰。
Register Started-by Manager	需求中沒有指出必須知道或紀錄負責啓動 *Register* 的管理員爲誰。
Sale Initiated-by Customer	需求中沒有指出必須知道或紀錄負責開

	始一筆銷售的目前顧客爲誰。
Store Stocks Item	需求中沒有指出必須知道或維護的庫存資訊爲何。
SalesLineItem Records-sale-of Item	需求中沒有指出必須知道或維護的庫存資訊爲何。

請注意，要證明某個關聯是否爲必須知道的關聯是根據需求決定的。很明顯地，當需求改變時—例如收據上需要秀出收銀員的 ID —是否需要記住某個關係的情況也會跟著改變。

根據上面的分析結果，我們可以刪掉有問題的關聯。

必須知道的關聯 vs. 瞭解專用的關聯

維護關聯時只保留必須知道的關聯可以讓我們產生問題領域所需要的最小「資訊模型」—問題領域會被限制在目前討論的需求中。然而，用這個解決方案所產生的模型將無法傳達我們或其他人對領域的完整瞭解。

除了把領域模型視爲必須知道事物的資訊模型之外，我們也可以把領域模型當作溝通工具，嘗試用這個工具讓別人瞭解並溝通重要概念與概念間的關係。從這個觀點來看，刪掉一些不符合嚴格的「必須知道」準則的關聯而產生的模型可能無法溝通關鍵概念與關係。

舉例來說，在 POS 應用程式中：雖然根據嚴格的「必須知道」準則，我們不需要紀錄 *Sale Initiated-by Customer* ，不過少了這個關聯之後，我們將無法了解領域的一個重要概念—顧客會產生一筆銷售。

就關聯方面來說，好的模型介於最小的「必須知道」模型與展示所有想像得到關係模型之間。證明某些關係價值的基本準則爲：它是否滿足「必須知道」需求，或者有助於溝通，可以讓我們瞭解問題領域中的重要概念。

把重點放在必須知道的關聯上，不過模型中還是需要放入只可增加問題領域關鍵概念的瞭解專用關聯。

第十二章領域模型：加入屬性

我們無法在後面開發階段中區別程式臭蟲是屬於哪個系統特性的。

— Rich Kulawiec

本章目標

- 找出領域模型中的屬性。
- 區別正確的屬性、不正確的屬性。

簡介

針對目前開發中情節的資訊需求，找出能滿足其需要的概念性類別屬性是很有幫助的。本章探討如何找到適合的屬性，並且把屬性加入 NextGen 領域模型中。

第一節屬性

屬性（attribute）代表物件中邏輯上要存在的資料值。

請把需求中建議或隱含要記住的資訊，當成屬性放到領域模型中。

例如因爲某種理由，管理人員希望知道日期與時間，所以在收據（銷售資訊的一種報告形式）中通常會有日期與時間。因此， *Sale* 概念性類別中應該要有 *date* 與 *time* 屬性。

第二節 UML 中屬性的表示法

屬性是顯示在類別方塊圖形中的第二個區隔（請參見圖 12.1。）我們可以選擇要不要秀出屬性的型態。

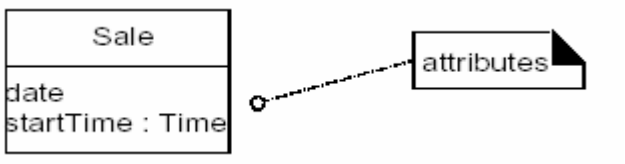


圖 12.1 類別與屬性

第三節有效的屬性型態

有些東西不應該用屬性表示，而應該用關聯表現。本節探討一些有效的屬性。

讓屬性保持簡單

很直覺地，大部分簡單屬性型態通常都是被視爲基本資料型態的東西，例如數字。屬性的型態通常不該是複雜的領域概念，例如 *Sale* 或 *Airport* 。例如我們不希望有像圖 12.2 裡面 *Cashier* 類別中的 *currentRegister* 屬性，因爲它不是簡單的屬性型態（例如 *Number* 或 *String*。）表達 *Cashier* 會用到 *Register* 的最好方式是用關聯而不是屬性。

領域模型中的屬性最好應該是**簡單屬性**或**簡單資料型態**。 最常見的屬性資料型態包含： Boolean 、 Date 、 Number、 String （Text）與 Time 。 其它常見的型態包含：Address 、 Color 、 Geometrics（Point 、 Rectangle）、 Phone Number 、 Social Security Number 、 Universal Product Code （UPC）、 SKU 、 ZIP 或 postal codes 、 列舉型態。

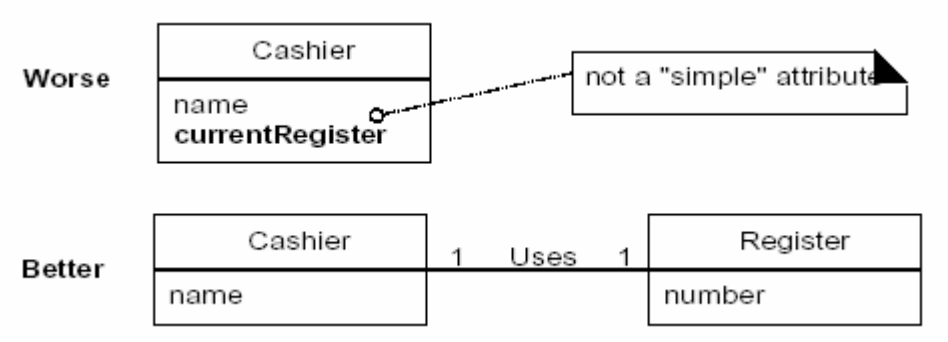


圖 12.2 應該用關聯而不是屬性。

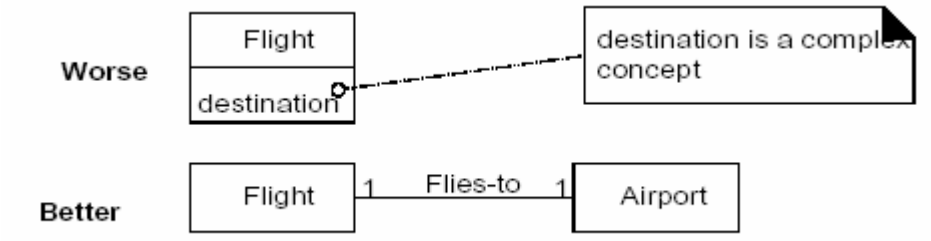


圖 12.3 避免把複雜領域概念當作屬性；用關聯。

重複前面的例子，有一個常見的疑惑是建模型時把複雜領域概念當成屬性。例如航程的目的機場不是字串而已，它是複雜的東西，會佔據好幾千公尺的空間。因此， *Flight* 跟 *Airport* 之間應該用關聯而不是屬性，如圖 12.3 所示。

跟概念性類別之間的關係用關聯而不要用屬性。

概念性觀點 vs. 實作觀點：程式碼中的屬

性是如何實作的？

領域模型中的屬性應該是簡單資料型態，這個限制並沒有說 C++ 或 Java 屬性（資料成員、實例欄位）只能是簡單、基本的資料型態。領域模型把焦點放在純粹問題領域的概念性敘述，而不是放在軟體元件上。

稍後進行設計與實作工作時，我們會看到領域模型中物件之間的關聯通常會被實作成參考到其它複雜軟體物件的屬性。然而，這也只是實作關聯的其中一種可能設計解決方案而已，建立領域模型時，不要太快下這些決策。

資料型態

屬性通常應該是資料型態（data type）。資料型態是 UML 中的一個術語，它隱含代表一組資料值，（在我們的模型或系統情境中）這些值的獨立個體是沒有意義的【RJB99】。例如區別下面東西（通常）是沒有意義的：

- 區別*數字* 5 的不同實例。
- 區別*字串*「cat」的不同實例。
- 區別相同*電話號碼*的不同實例。
- 區別相同*住址*的不同實例。

另一方面，區別兩個「Jill Smith」的不同 Person 實例則*是*有意義的，因爲這兩個實例分別代表有同樣名稱的不同個體。

從軟體來看，只有在很少時候我們才會比較*數字*、*字串*、*電話號碼*或*地址*的實例記憶體位址；我們只有比較值才有意義。另一方面，對 *Person* 實例的記憶體位址比較則是可以理解的，因爲縱然它們的屬性值一樣，所代表的個體也是不同的，因爲它們的單一個體是有意義、很重要的。

因此，所有基本型態（數字、字串等等）都是 UML 的資料型態，不過並非所有資料型態都是基本的。例如電話號碼就不是基本資料型態。

也有人把這些資料型態值稱爲**值物件**（value object）。

資料型態的表示法有些地方會很詭異。根據經驗法則，我們要忠於「簡單」屬性型態的基本測試：如果可以把它視爲單純的數字、字串、布林值、日期或時間等等，就把它變成屬性；否則的話，把它視爲獨立的概念性類別。

如有有疑慮的話，把某個東西定義成獨立的概念性類別而不要把它當成屬性。

第四節非基本資料型態型類別

我們可以把屬性的型態在領域模型中用非基本類別表達。例如 POS 系統中有一個商品識別碼，一般來說我們只會把它視爲數字。應該把它視爲非基本類別嗎？請根據這裡的指引決定：

如果遇到下面的情況，我們可以把一開始視爲基本資料型態的東西（例如數字或字串）視爲非基本類別：

- 它由幾個不同區段組成。

電話號碼、人的名字

- 它通常會跟操作有關，例如解析或驗證操作。

社會安全號碼

- 它的裡面有其它屬性。

促銷價格應該有開始（有效）日期與截止日期

- 它是帶有單位的數量。

付款金額需要有現金單位

- 它是具有某些特性、一個或多個型態的抽象概念。

在銷售領域中商品識別碼代表某些型態（例如 Universal Product Code 【UPC】或 European Article Number【EAN】）一般化後的結果。

把這樣的指引應用在 POS 領域模型之後，我們可以得到下面的分析結果：

- 商品識別碼是各種常見編碼綱目的抽象概念，例如 UPC-A 、 UPC-E 與整個 EAN 綱目家族。這些數字性編碼綱目裡面會區分成不同部分，分別用來識別製造商、產品、國家（EAN）與驗證用的檢查加總位元。因爲它滿足上面許多指引的條件，所以領域模型中應該要有非基本類別 *ItemID* 類別。
- 價格與總價屬性應該分別是非基本類別 *Quantity* 或 *Money* ，因爲它們都具有現金單位。
- 地址屬性應該是非基本類別 *Address* ，因爲裡面有不同的區段存在。

類別 *ItemID* 、 *Address* 與 *Quantity* 都是資料型態（實例的單獨個體是沒有意義的），不過因爲它們具有一些性質，所有我們應該把它們視爲獨立的類別。

資料型態型類別應該秀在哪裡？

我們應該在領域模型中把 *ItemID* 用獨立概念性類別秀出來嗎？要不要秀出來主要是根據你在圖中想強調的重點爲何來決定的。因爲 *ItemID* 是*資料型態*（實例的單獨個體並不重要），所以我們可以把它秀在類別方塊圖形的屬性區隔中，如圖 12.4 所示。不過因爲它是非基本類別，有自己的屬性與關聯，所以我們也可能把它當概念性類別一樣用獨立方塊圖形秀出來。其實沒有所謂的正確答案；它是看我們把領域模型當成怎樣的溝通工具，以及這個概念在領域中的重要性來決定。

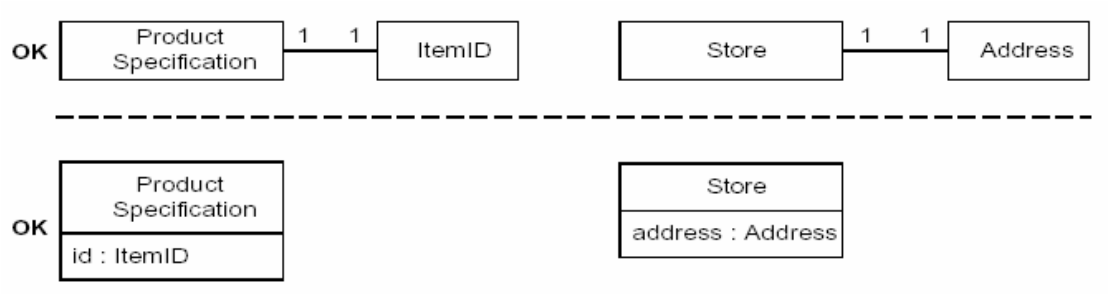


圖 12.4 如果屬性類別是一種資料型態，我們可能會把它秀在類別方塊圖形的屬性區隔內。

第五節設計偷跑：不要把外來鍵當成類別中的屬性

我們不應該在領域模型中利用屬性關聯到其它概念性類別。最常違反這個原則的情況就是加入**外來鍵型屬性**（foreign key attribute），我們常在關連式資料庫設計中用這樣的欄位關聯兩個型態。例如在圖 12.5 中我們不希望 *Cashier* 類別有一個 *currentRegisterNumber* 屬性，以便從 *Cashier* 關聯到 *Register* 物件上。最好的方式是用關聯說明 *Cashier* 會使用 *Register* ，而不是用屬性。再強調一次，用關聯而不要用屬性建立兩個型態之間的關係。

我們可以用很多種方式連到其它物件上—外來鍵是其中一種—我們最好等到設計時才決定如何實作這個關係以避免設計偷跑（design creep）的狀況。

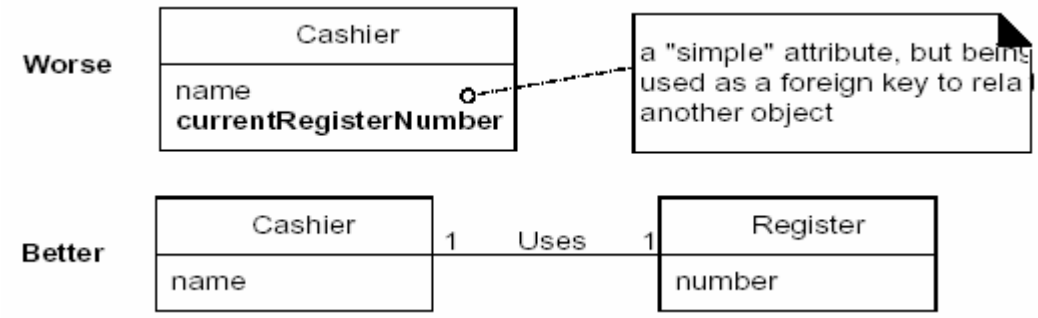


圖 12.5 不要把屬性當外來鍵用。

第六節描述模型中屬性的數量與單位

大部分數字型的數量都不該只用單純數量表示。想一下價格或向量問題。這些都是跟單位有關聯的數量，而且爲了方便溝通，我們通常也需要知道單位。 因爲 NextGen POS 軟體是針對國際市場設計的，所以系統裡面的價格需要支援多種

幣別。在一般情況下，我們會把這樣的數量跟*單位*關聯在一起【Fowler96】。因爲這些數量都是資料型態（實例的單獨個體並不重要），因此把它們秀在類別方塊圖形的屬性區隔是可接受的（請參見圖 12.6。）我們也常秀出數量特殊化的結果。例如 *Money* 就是帶有幣別單位的數量。 *Weight* 則是帶有公斤或英鎊單位的數量。

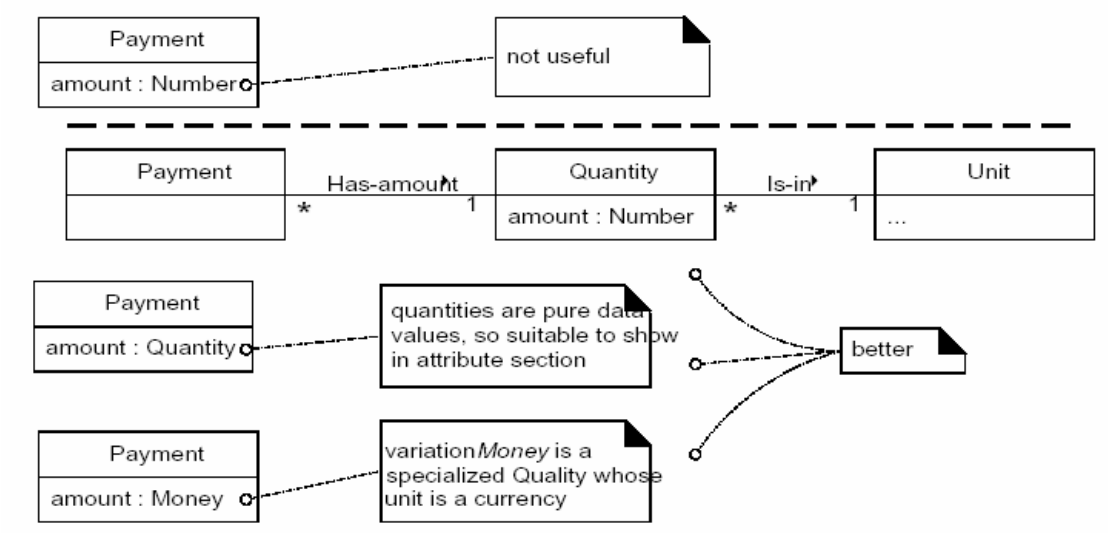


圖 12.6 建立跟單位相關的模型。

第七節NextGen 系統領域模型中的屬性

我們根據目前反覆中的需求— *處理銷售*情節 —選擇屬性。

Payment	amount（金額）—爲了決定付款金額是否足夠、算出要找的零錢，我們必須捕捉金額（有時候也稱爲付款金額。）
ProductSpecification	description（說明文字）—爲了在螢幕或收據上秀出說明文字。 id（識別碼）—爲了找到 *ProductSpecification* （產品規格），我們會輸入商品識別碼，再把這個商品識別碼跟 *ProductSpecification* 的 id 關聯在一起。 price（價格）—爲了計算銷售總金額、商品明細項目價格。
Sale	date（日期）、time（時間）—收據是銷售的紙張報告，裡面通常會秀出銷售的日期與時間。
SalesLineItem	quantity（數量）—當某種銷售明細項目

	的數量超過一個時，需要輸入並記錄銷售數量（例如有 *5* 包豆腐。）
Store	address（地址）、name（名稱）—收據中需要商店的名稱與住址。

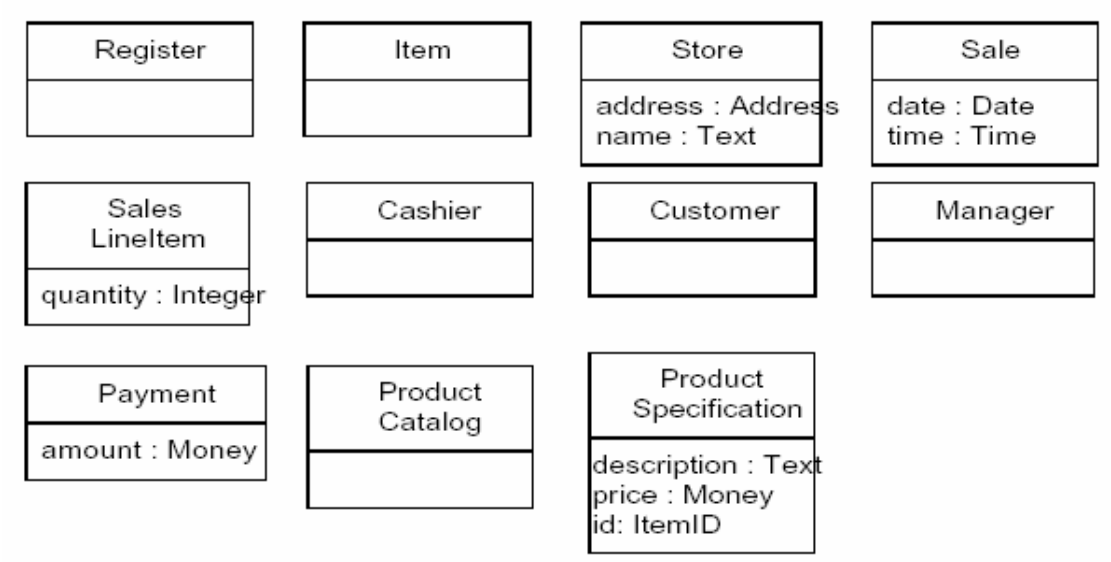


圖 12.7 需要在領域模型中秀出來的屬性。

第八節在 SalesLineItem 與 Item 之間關聯的多重性

收銀員有可能會收到一組類似的商品項目（例如六包豆腐），這時候他只要先輸一次 itemID ，再輸入數量就可以了（例如 6。）因此，一個 *SalesLineItem* 有可能會跟多個某種商品項目的實例關聯在一起。

由收銀員輸入的數量可以紀錄成 *SalesLineItem* 的屬性（圖 12.8。）然而，這個數量也可以從關係的實際多重性算出來，所以它是一種**導出屬性**（derived attribute）—可以從其它資訊導出的屬性。在 UML 中，導出屬性用「/」符號標示。

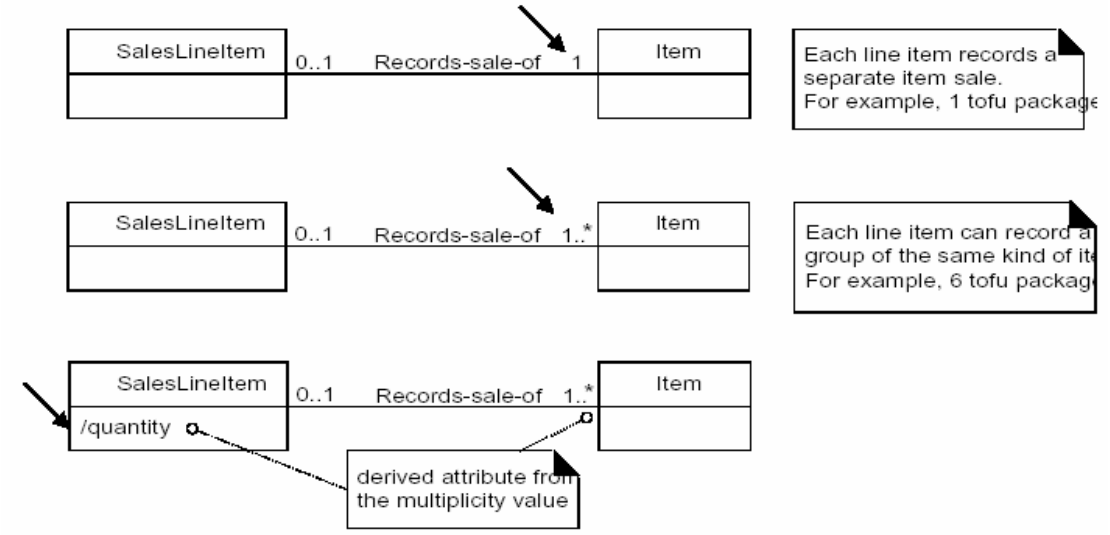


圖 12.8 紀錄商品明細項目中所銷售的商品數量。

第九節領域模型總結

總結之前所找到的概念性類別、關聯與屬性，可以畫出圖 12.9 。

對 POS 應用程式的領域來說，這是相當有用的領域模型。事實上，並沒有所謂「對」的模型存在。所有模型都是我們嘗試理解領域而得到的近似模型而已，而且模型是用來幫人們瞭解領域的—瞭解領域中的概念、術語與關係。

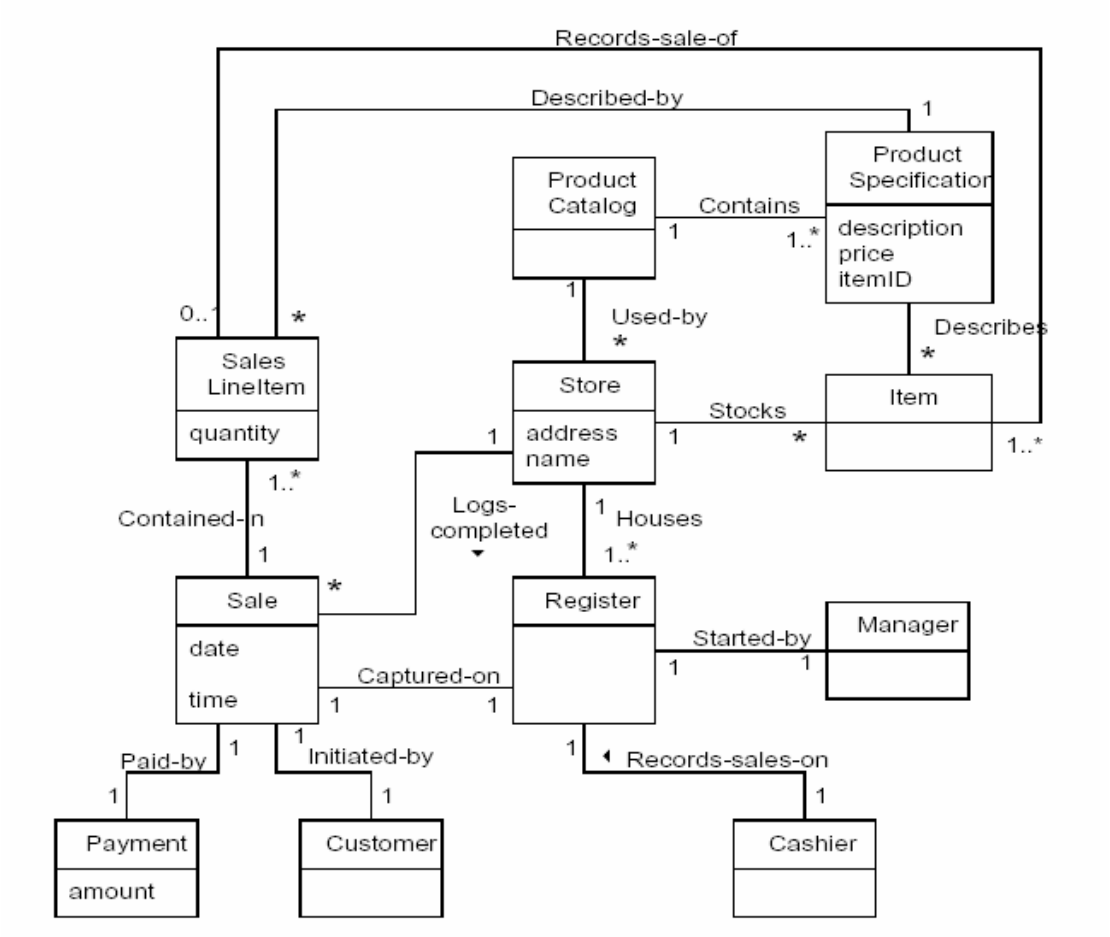


圖 12.9 領域模型的一部份

第十三章使用案例模型：加入操作合約細節

任何口頭上的承諾都比不上白紙黑字。
— Samuel Goldwyn

本章目標

- 產生系統操作（system operation）合約。

簡介

我們可以用操作合約定義系統行爲；它們用領域物件的狀態變化描述系統操作執行後的外顯結果。本章探討如何使用操作合約。

第一節合約

使用案例是 UP 中描述系統行爲的主要機制，用它描述系統行爲其實已經足夠了。然而，有時候更詳細的系統行爲說明是有價值的。合約用系統行爲執行後的領域模型狀態變化描述系統行爲細節。

系統合約與系統介面

合約是用來定義**系統操作**（system operation）的—這種操作把系統視爲黑盒子，裡面是系統爲了處理系統事件而提供的系統公開介面。我們可以由系統事件找到系統操作，如圖 13.1 所示。

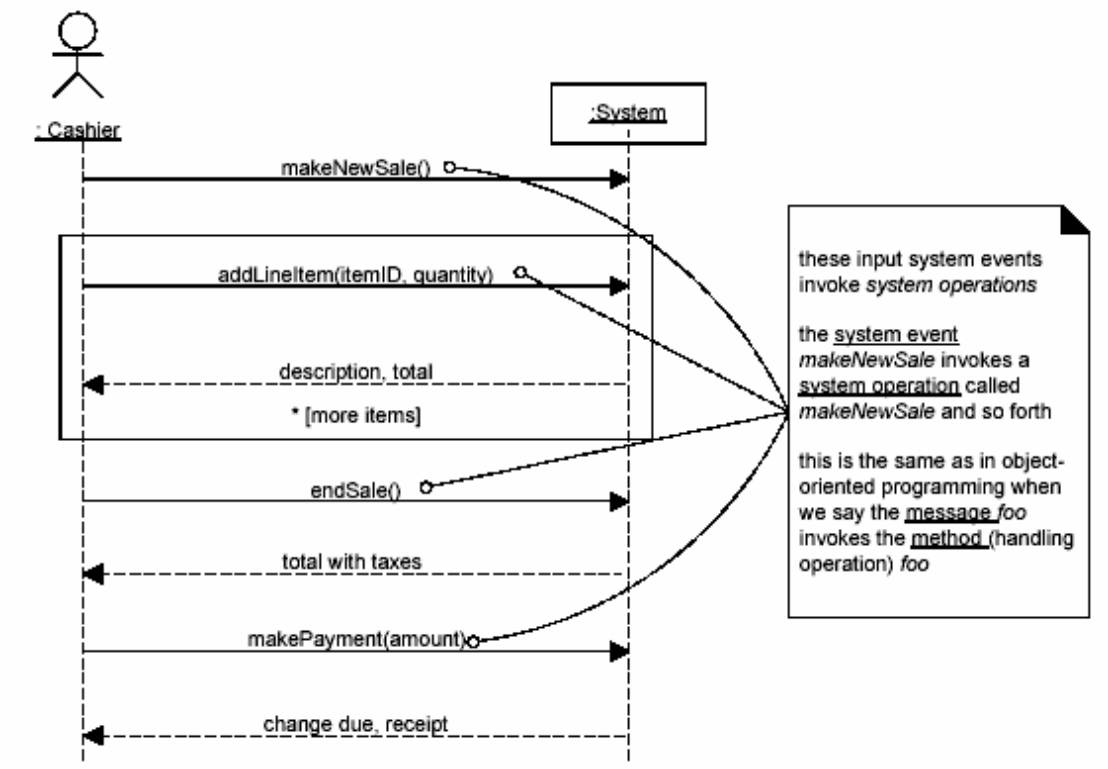


圖 13.1 系統操作會處理輸入系統事件。

橫跨所有使用案例的一整組系統操作代表公開系統介面，這時候我們是把系統當成單獨的元件或類別看待。在 UML 中，我們可以用類別代表視爲一體的系統。

第二節合約範例：enterItem

在說明寫合約的理由之前，我們先介紹一個值得注意的例子。後面是 *enterItem* 系統操作的合約內容。

合約 CO2: enterItem

操作：	enterItem(itemID : ItemID, quantity : integer)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—已經產生 SalesLineItem 的一個實例 sli（實例的產生。） —把 sli 跟目前 Sale 關聯在一起（關聯的形成。） —把 sli.quantity 修改變成 quantity（屬性的修改。） —根據 itemID ，把 sli 跟 ProductSpecification 關聯在一起（關聯的形成。）

第三節合約中的各個小節

下面是合約中各個小節的說明。

操作：	操作的名稱與參數
交互參考：	（選用）這個操作所存在的使用案例
事先條件：	執行操作之前，值得注意的一些*假設*，其中包括系統或物件的狀態。操作邏輯中不會測試這些假設，並且假設它們是真的，它們是閱讀合約者應該要知道的有意義假設。
事後條件：	—完成操作後的領域模型物件狀態。接下來的章節會討論其中的細節。

第四節事後條件

請注意在 *enterItem* 範例中，每個事後條件都會有一種分類，例如*實例的產生*或*關聯的形成*。這是一個關鍵點：

> 事後條件裡面會描述領域模型中物件狀態的改變。領域模型的狀態改變包括實例的產生、關聯的產生與中斷、屬性的修改。

事後條件不是操作執行時的動作；相反地，它們是操作完成後—*在瀰漫的煙霧消失後*，領域模型中物件一定要成立的結果。

彙總來說，事後條件會屬於下面分類之一：

- 實例的產生與刪除。
- 屬性的修改。
- 關聯（更準確的說法是 UML 中的繫結【*link*】）的形成與中斷。

舉一個關聯中斷的例子，考慮刪除商品明細項目的操作。我們應該把它的事後條件念作「把選到的 SalesLineItem 跟 Sale 之間的關聯中斷掉。」在其它領域中，當貸款已經被付清或有人取消他們的會員身分時，關聯就會中斷掉。

刪除實例的事後條件很少見，因爲在真實世界中一般人通常不會明確刪除某個東西。然而，還是可以找到一個例子：在許多國家中，如果有人宣佈破產，而且已經過了七年或十年，根據法律所有破產計錄都必須刪除掉。請注意，這裡所提到的是概念性觀點而不是實作關點，所以這裡不是要釋放掉電腦中被軟體物件佔據的記憶體。

最重要的一個性質是：它是宣告性、狀態改變導向而不是動作導性的，因爲事後條件裡面只會宣稱狀態或外顯結果，而不是執行動作或解決方案設計的說明。

事後條件是跟領域模型有關的

事後條件用領域模型中物件表達出來。事後條件裡面說明會產生怎樣的實例？—這些實例是從領域模型來的；要形成怎樣的關聯？—這些關聯也是從領域模型來的；就像這樣。

事後條件的好處：它是分析後的細節

因爲合約是用宣告式、狀態變化的風格表達，所以它是很好的需求分析工具，裡面描述系統操作所需要的狀態變化（用領域模型物件 表達）而不描述要如何達成這些東西。換言之，我們稍後才考慮軟體的設計與解決方案，現在只要把分析焦點放在發生什麼而不是如何達成。此外，點銷售中會提供細部的細節，並且明確宣告操作必須達成的外顯結果。

我們也可能在使用案例中用這種程度表達細節，不過通常這樣的使用案例不是我們要的，因爲這樣一來使用案例會變得有點囉唆、太詳細了。

考慮這樣的事後條件：

事後條件：	—已經產生 SalesLineItem 的一個實例 sli （實例的產生。） —把 sli 跟目前 Sale 關聯在一起（關聯的形成。） —把 sli.quantity 修改變成 quantity（屬性的修改。） —根據 itemID ，把 sli 跟 ProductSpecification 關聯在一起（關聯的形成。）

裡面沒有說明 *SalesLineItem* 的實例是怎麼產生的、或怎樣跟 *Sale* 關聯在一起。事後條件有點像寫了一些碎紙片，再用訂書機把它們釘在一起、用 Java 技術產生軟體物件然後把它們連接成系統或者把記錄插入關聯式資料庫中。

事後條件的精神所在：舞台與舞台上的幕

用過去式表達事後條件是爲了強調它們是宣告過去發生的狀態變化。例如：

■ （比較好的寫法）**已經**產生一個 *SalesLineItem*。

而不是用這樣寫的

■ （比較差的寫法）產生一個 *SalesLineItem* 。

用下面的畫面想像一下事後條件：

系統與它的物件都在劇場舞台上。

1. 執行操作之前，先替舞台照一張相。
2. 落下舞台上的幕，並且執行系統操作（背後有鏗鏘聲、尖叫聲等等。）
3. 升起舞台上的幕，再照第二張相。
4. 比較之前與之後的照片，並且用事後條件表達舞台狀態的變化（已經產生一個 *SalesLineItem* 等等。）

如果寫出（操作）合約，事後條件應該要寫得多完整？

首先，我們可能不需要寫出合約。這個問題會在稍後章節中討論。不過假設我們只想產生一些合約。在需求開發工作中，我們不可能（甚至不需要）完整與詳細的系統操作事後條件。瞭解合約是不完整的之後，我們只把這些寫出來的合約視爲剛開始的最好猜測而已。在早期產生這些合約—縱然不完整—也一定比到設計開發工作時再調查還好，因爲那個時候開發人員的重點是設計出解決方案，而不是調查要做*什麼*東西。

我們會在設計開發工作中發掘一些比較細微的細節—有時候甚至是比較粗略的細節。這樣做不一定是件壞事；如果把需求分析弄得太長，我們所花費功夫的回報率就會降低。在設計開發工作中，有些發現會很自然地增加，我們會把這些後來獲得的資訊當成稍後反覆的需求。這是反覆式開發方式的優點之一：從前面反覆得到的發現可以加強接下來反覆的調查與分析工作。

第五節討論：enterItem 的事後條件

我們會在這個小節仔細分析 *enterItem* 系統操作的事後條件動機。

實例的產生與刪除

輸入某個商品項目的 *itemID* 與數量之後，我們應該產生怎樣的新物件呢？ *SalesLineItem* 。因此：

- 我們會產生一個 *SaleLineItem* 實例 sli（實例的產生。）

請注意實例的命名方式。這個名稱可以讓我們很簡單就在其它事後條件敘述中參考這個新的實例。

屬性的修改

收銀員輸入商品項目的 *itemID* 與數量之後，我們應該修改這個新的或現存物件的什麼屬性呢？ *SaleLineItem* 的 *quantity* 應該要跟 *quantity* 參數一樣。因此：

- *sli.quantity* 會成爲 *quantity* 裡面的值（屬性的修改。）

關聯的形成與中斷

收銀員輸入商品項目的 *itemID* 與數量之後，我們應該形成或中斷新的或現存物件之間的關聯？新的 *SaleLineItem* 應該跟 *Sale* 建立起關係，也要跟 *ProductSpecification* 建立起關係。因此：

- *sli* 會跟目前 *Sale* 建立關聯（關聯的形成。）
- 根據 *itemID* ， *sli* 會跟相符的 *ProductSpecification* 建立關聯（關聯的形成。）

請注意，這裡是非正式的寫法：它會跟 *itemID* 符合參數的特定 *ProductSpecification* 建立關係。我們可以用更炫的正式語言，例如物件限制語言（Object Constraint Language，OCL）來寫。建議：讓事後條件保持清楚、簡單。

第六節寫合約時可能會需要更新領域模型

產生合約時，我們通常會需要在領域模型中紀錄新的概念性類別、屬性或關聯。不要被之前定義出來的領域模型限制住；當你思考操作合約而有新的發現時，就可以加強現有的領域模型。

第七節什麼時候合約對我們有幫助？合約 vs. 使用案例

使用案例是專案的主要需求寶庫。它們可以提供設計時需要知道的大部分或所有細節，這時候，合約並沒有特別有幫助的地方。然而，有的時候想在使用案例中捕捉一些必要狀態變化的細節與複雜度是很可怕的。

舉例來說，考慮航空訂位系統中的系統操作 *addNewReservation* 。如果要考慮到所有有變化、新產生或有關聯的領域物件，複雜度將非常高。我們可以在使用案例中寫出跟這個系統操作有關的所有細部細節，不過這樣一來使用案例將變得非常複雜（例如注意所有物件中會改變的屬性。）

請觀察一下，合約中的事後條件格式可以提供並鼓勵我們用非常精確、解析得非常細、嚴格的語言寫出最完整的細節。

如果只以使用案例爲基礎，搭配主題專家的合作，開發人員就可以瞭解要做什麼，那麼我們就沒有必要寫出合約。

然而，如果高複雜度與高精確度是有價值時，就可以把合約當成另一種需求工具。事實上，我們通常不會想寫出合約，所以如果開發團隊想寫出每個使用案例中每個系統操作的合約，有可能是使用案例寫得很差、跟主題專家後續的合作或接觸

不夠，或者開發團隊寫太多不必要的文件了。
NextGen POS 個案研究中寫出的合約比可能需要的情況更多，其目的是爲了教學用。在實務上，這些合約中紀錄的大部分細節都可以很明顯地從使用案例說明文字中找到。不過，「明顯」是很難拿捏的一個概念。

第八節指引：合約

產生合約時可以參考下面的建議：

產生合約時：
1. 從 SSD 中找出系統操作。
2. 針對複雜、結果可能有細微差異或使用案例寫得不清楚的系統操作，產生合約。
3. 根據下面分類來描述事後條件：

- 實例的產生與刪除
- 屬性的修改
- 關聯的形成與中斷

寫合約時的建議

- 用宣告式、被動式、過去式的時態格式（被…）寫出事後條件，以強調說我們是在宣告狀態的變化而不是設計如何達成這些變化。例如 ：
  - （比較好的寫法）已經產生一個 *SalesLineItem* 。
  - （比較差的寫法）產生一個 *SalesLineItem* 。
- 要記得在現存或新的物件之間定義關聯的形成。例如當 *enterItem* 操作發生時，只產生一個新的 *SalesLineItem* 實例是不夠的，我們還要把新產生的實例關聯到 *Sale* 上；因此：
  - *SalesLineItem* 會被關聯到 *Sale* 上（關聯的形成。）

寫合約時最常犯的錯誤

最常犯的錯誤是忘了把關聯的形成加到合約中。特別是當我們產生一個新的實例時，這時候非常有可能需要建立跟好幾個物件之間的關聯。請不要忘記！

第九節NextGen POS 系統中的合約範例

處理銷售使用案例中的系統操作

合約 CO1: makeNewSale

操作：	makeNewSale()
交互參考：	使用案例：處理銷售
事先條件：	無
事後條件：	—已經產生一個 Sale 實例 s （實例的產生。） —把 s 跟 Register 關聯在一起（關聯的形成。） —把 s 的屬性初始化（屬性的修改。）

合約 CO2: enterItem

操作：	enterItem(itemID : ItemID, quantity : integer)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—已經產生 SalesLineItem 的一個實例 sli （實例的產生。） —把 sli 跟目前 Sale 關聯在一起（關聯的形成。） —把 sli.quantity 修改變成 quantity（屬性的修改。） —根據 itemID ，把 sli 跟 ProductSpecification 關聯在一起（關聯的形成。）

合約 CO3: endSale

操作：	endSale()
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—把 Sale.isComplete 的值設成 true（屬性的修改。）

合約 CO4: makePayment

操作：	makePayment(amount: Money)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—已經產生 Payment 的一個實例 p （實例的產生。） — p.amountTendered 的值設成 amount （屬性的修改。） —把 p 跟目前 Sale 關聯在一起（關聯的形成。）

—把目前 Sale 跟 Store 關聯在一起（關聯的形成；把目前 Sale 加到以完成銷售的歷史記錄中。）

第十節領域模型的變化

我們從合約中可能發現到有一份資料還沒有出現在領域模型：紀錄銷售的商品項目已經輸入完成。我們會在 *endSale* 中修改這個資料，而且這個資料也有助於稍後 *makePayment* 操作的設計中用它測試銷售是否已經完成，以便在銷售還未完成之前禁止付款動作。

呈現這個資訊的一種方式是在 *Sale* 裡面加入 *isCompelete* 屬性，這個屬性的資料型態是 boolean ：

設計時還可以用其它方式。其中一種技術稱爲**狀態樣式**（state pattern），我們會在第 34 章討論。另一種方式是使用「session」物件，我們會追蹤 session 的狀態，不允許操作沒有按照順序執行；稍後也會介紹這種方式。

第十一節合約、操作與 UML

UML 中的合約：操作規格

UML 中有正式定義**操作**（operation）。引述：

操作是我們呼叫物件執行，以轉變或查詢物件狀態的一種規格【RJB99】。

例如在 UML 裡面介面的組成元素都是操作。操作是抽象概念而不是實作。另一方面，（UML 中的）**方法**則是操作的實作。

UML 的操作包含**用法**（signature）（名稱與參數）、**操作規格**（operation specification）（用來描述執行操作後所產生的效果；換言之，事後條件。）UML 中操作規格的格式是很彈性，它不是用本章所秀的合約格式。然而，UML 中的文件有類似合約風格的例子，裡面包含事先條件與事後條件，這也是最廣爲人知的正式操作規格。

結論：UML 中有定義操作規格，我們可以像合約風格一樣用事先條件與事後條件明確說明操作規格。就像本章所強調的一樣，UML 的操作規格裡面*不要*秀出演算法或解決方案，只要秀出操作會產生的狀態變化或效果就好了。

除了用合約標明整個*系統*的公開操作（系統操作），我們可以把合約用在任何粗細程度的操作上：子系統、抽象類別等等的公開操作（或介面。）本章所討論的操作隸屬於 *System* 類別。UML 的操作都必須隸屬於類別。此外，在 UML 中，「子系統」也被視爲一種類別（同時也被視爲套件。）UML 裡面把整個「系統」視爲最上層的子系統，也把它視爲帶有公開操作與規格的 *System* 類別（事實上，系統用任何名稱都可以。）

用 OCL 表達操作合約

跟 UML 相關的一種正式語言叫做物件限制語言（Object Constraint Language，OCL）【WK99】，我們可以用它表達模組中的限制。你可以用 OCL 取代本章所使用的非正式自然語言；UML 允許我們用任何格式寫出操作規格。

建議
除非有強制性、實務上的理由需要人們學習並使用物件限制語言，請讓東西簡單點，並且使用自然語言。

OCL 對如何標示操作的事先條件與事後條件有正式格式，下面是範例片段：

System::makeNewSale()

pre : （statements in OCL）

post : …

更多的 OCL 細節已經超出這本入門書的範圍。

合約式設計中的合約

UML 操作規格中所使用的事先條件與事後條件是 Bertrand Meyer 推廣多年的想法；這種想法後來演變成正式的設計解決方案，稱爲**合約式設計**（design by contract）【Meyer97（1989 年第一版）】，事實上在 1960 年代已經有正式規格語言出現。在合約式設計中，我們也會寫出細部類別的操作合約，不只是系統或子系統的公開操作而已。

此外，合約式設計提倡合約裡面要有*不變條件*（invarant）小節，這樣才是完整的合約規格。不變條件中定義執行操作之前與之後不變的狀態。爲了簡單起見，本章範例並沒有使用不變條件小節。

程式語言對合約的支援

有些程式語言（例如 Eiffel）對不變條件、事先條件與事後條件有最好的支援。Java 中也有一些事先編譯器（pre-processor）提供類似支援。

第十二節UP 中的操作合約

大家都知道事先條件與事後條件合約是 UML 中明確說明操作的風格。不過 UML 中的操作可能有許多層級，從 *System* 到細部類別都有可能，例如 *Sale* 。雖然原始的 RUP 或 UP 說明文件中沒有特別強調，不過 *System* 層級的操作規格合約是使用案例模型中的一部份；我們有跟 RUP 作者確認過的確是如此【註】。
【註】：透過私人通訊。

開發階段

初始階段—我們在詳述階段不會想寫出合約，它們太瑣碎了。
詳述階段—如果會用到合約的話，大部分的合約都是在詳述階段寫的，這時候大部分的使用案例已經被寫好了。我們只需要寫出最複雜、有細微差異的系統操作。

工作成果之間的關係

圖 13.2 與圖 13.3 中是在不同層級下，合約與其它工作成果之間的關係。

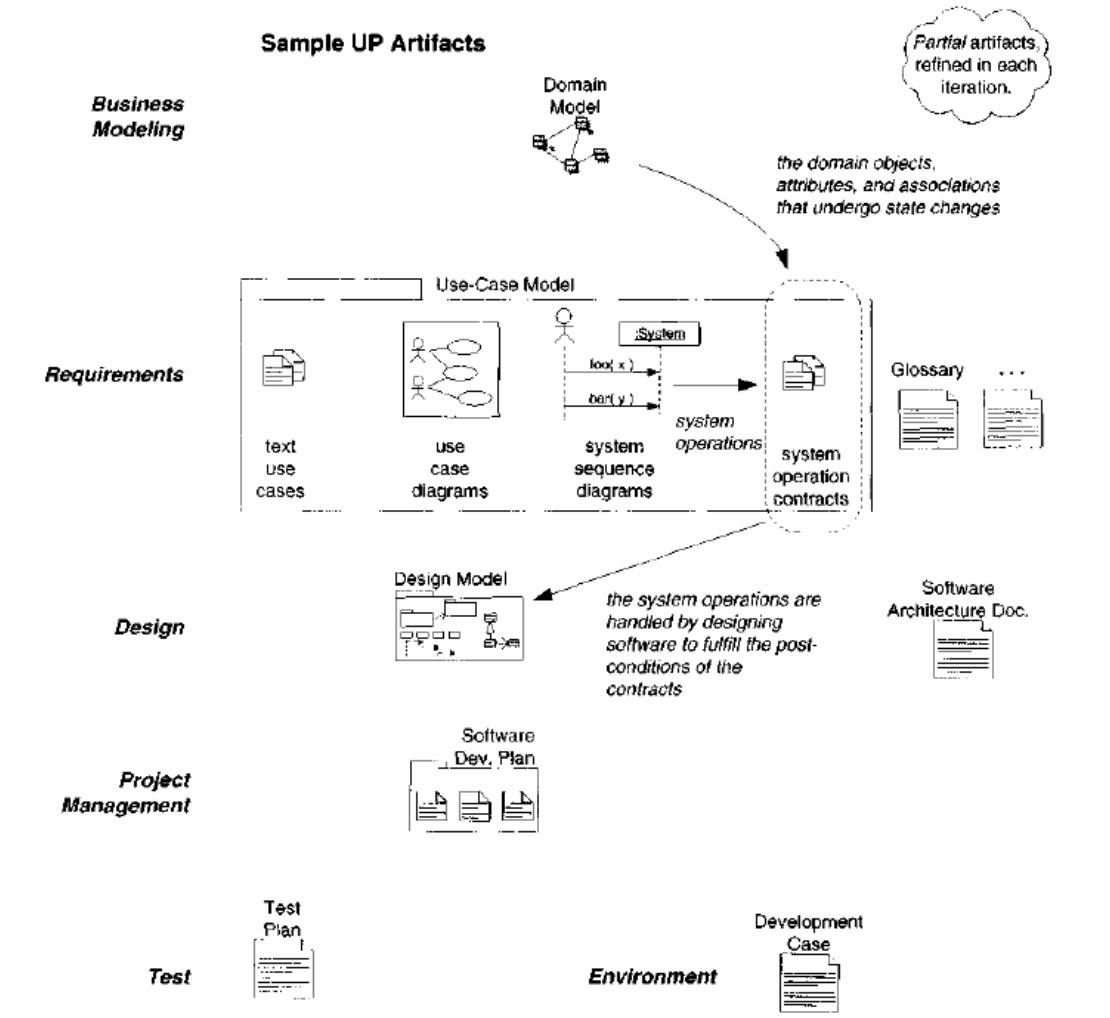


圖 13.2 UP 中工作成果之間可能發生的相互影響。

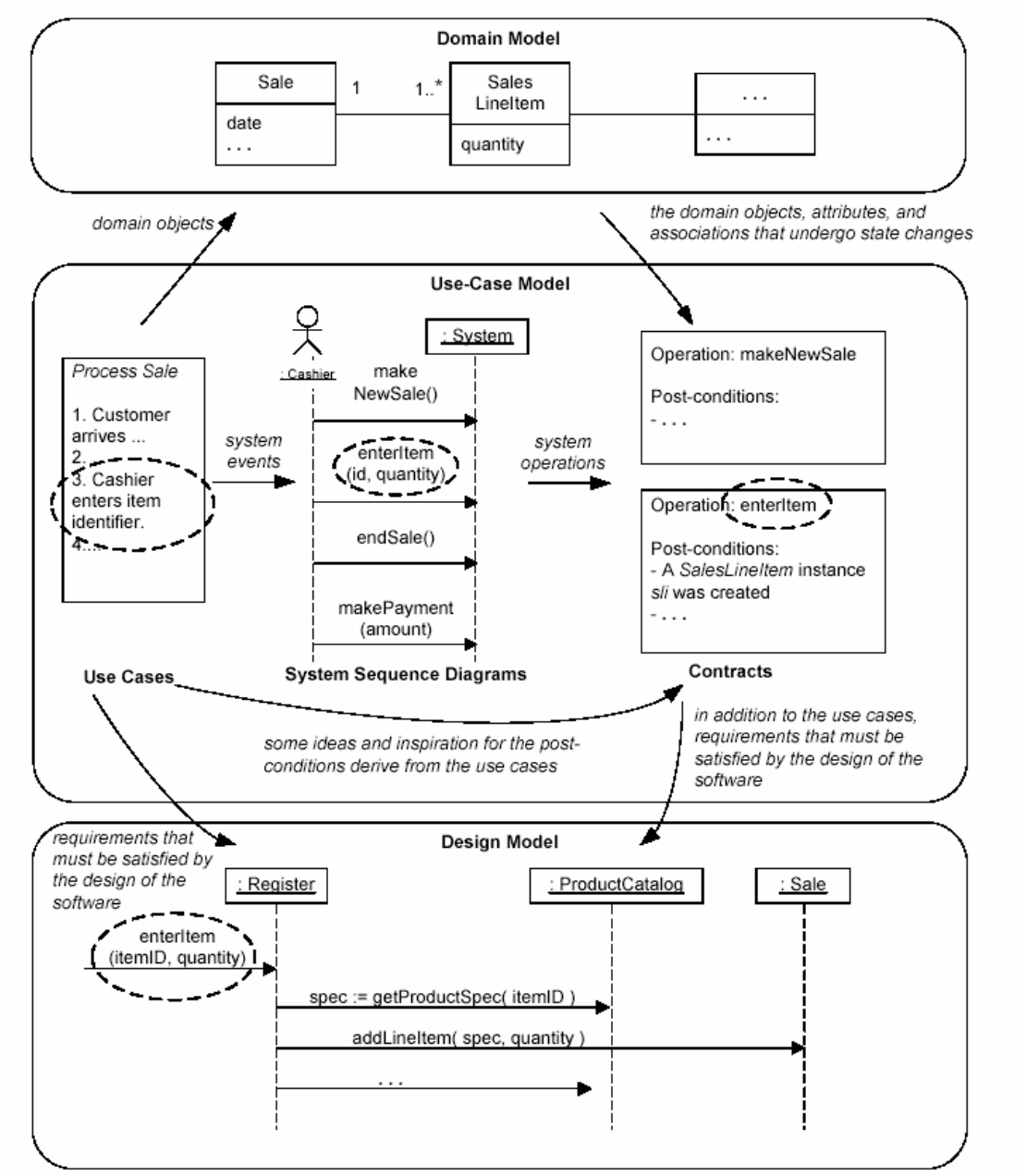


圖 13.3 合約跟其它工作成果之間的關係。

第十三節進階讀物

操作合約是從正式規格領域來的一種概念，從 1960 年代開始就有人開始用這種概念，並且概念也一直不斷修正中，例如 Vienna Development Method（VDM）【BJ78】；在 VDM 與其它正式規格語言中你可以找到許多豐富的文獻。

Bertrand Meyer 把事先條件與事後條件放入 Eiffel 程式語言中，讓更多人知道正式規格與合約的存在；你可以在他的書 *Object-Oriented Sooftware Construction* 中找到許多細節。他負責**合約式設計**概念。

在 UML 中，我們可以用更強固的物件限制語言（OCL）明確說明操作合約，針

對這一點， Warmer 與 Kleppe 的 *The Object Constraint Language: Precise Modeling with UML* 是值得一看的好書。

第十四章在這個反覆中把需求變成設計

本章目標

- 說明設計開發活動的動機。
- 比較物件設計技能與 UML 表示法知識的重要性。

簡介

到目前爲止，這個個案研究把焦點放在需求、概念與跟系統相關操作的調查工作上。根據 UP 指引，大約有 10% 的需求會在初始階段中找到，然後在詳述階段的第一個反覆中開始進行更深入的調查工作。接下來的章節會把焦點轉移到用軟體物件合作關係設計出這個反覆的解決方案。

第一節做對的事、把事做好，不斷重複這樣的過程

需求與物件導向分析把焦點放在學習*做對的事*；換言之，瞭解 NextGen POS 、相關規則與限制的一些外顯目標。另一方面，接下來的設計工作則強調*把事做好*；換言之，很熟練地設計出一個解決方案以滿足這個反覆中的需求。

在反覆式開發方式中，每個反覆都會發生主要焦點從需求轉移到設計與實作的過程。因此，很自然、也很健康地，我們會在*早期*反覆的設計與實作工作中發現並改變一些需求。這些發現一方面可以釐清目前反覆的設計工作目的，另一方面可以修正我們對未來反覆需求的瞭解。經過這些早期詳述階段反覆之後，發現需求的情況越來越少，需求會穩定下來 ，所以在詳述階段結尾時，大約 80% 的需求都已經有詳細定義也可信賴了。

第二節應該花幾個星期做這些事嗎？不，不用花這麼多時間

經過前面幾章的詳細討論，那些建立模型的工作看起來好像要花幾個星期才能做完。其實不用。當你已經熟悉使用案例的寫作、建立領域模型等等，真正做這些工作所需花費的時間只要花*幾*天就好了。

不過，這不代表專案只開始幾天而已，因爲我們還需要進行許多開發活動，例如驗證概念的程式設計、找資源（人、軟體等等）、規劃、設定環境等等，需要花幾個星期準備這些工作。

第三節接下來繼續進行的物件設計工作

在物件設計中，我們會發展出以物件導向典範爲基礎的邏輯解決方案。解決方案的核心是產生***互動圖***（interaction diagram），展示物件如何合作滿足需求。

在畫出—或同時間—*互*動圖之後，我們會畫出（設計）**類別圖**（class diagram）。類別圖裡面彙總軟體中要實作的軟體類別（與介面）定義。

就 UP 來說，這些工作成果都是設計模型（design model）的一部份。

事實上，我們應該同時產生*互*動圖與類別圖，這樣做能夠有縱效產生。不過個個案研究中會循序介紹它們，以簡化並釐清相關概念。

物件設計技能的重要性 vs. UML 表示法相關知識的重要性

接下來的章節會探討如何產生*互*動圖、類別圖、設計模型等工作成果，更精確的說法是：產生這些工作成果的物件設計技能。知道如何用物件思考並設計系統是很重要的事，而且這跟知道 UML 的畫圖表示法不太一樣，前者更加重要。不過，利用標準、以視覺方式呈現的語言畫圖是很棒的事，所以我們會用必要的 UML 表示法畫出設計工作所要呈現的東西。

接下來要探討的兩個工作成果（*互*動圖與類別圖）中，如果從開發好的設計結果方面來看，*互*動圖是最重要的工作成果，畫*互*動圖時需要最多的創造性。*互*動圖的產生需要應用指派**責任**原則、**設計原則與樣式** （design principle and pattern）。因此，接下來的章節會把焦點放在這些物件設計的原則與樣式上。

物件設計技能 vs. UML 表示法技能

畫 UML 互動圖時會做出物件設計的一些決策。

物件設計技能是真槍實彈的東西，它不是畫 UML 圖的技能。

基本物件設計所需的相關知識包括：

- 指派責任原則
- 設計樣式

第十五章互動圖的表示法

貓比狗還精明，不過你不可能讓八隻貓在雪地拉雪橇。

— Jeff Valdez

本章目標

- 學習 UML 中基本的互動圖（循序圖與合作圖）表示法。

簡介

我們在接下來幾個章節中探討物件設計。互動圖是展現設計用的主要圖。因此，建議大家至少要抓住本章範例中的精華，繼續讀更後面的內容之前先熟悉（譯註：UML）表示法。

UML 中用**互動圖**（interaction diagram）展現物件之間的訊息互動情形。本章先簡介相關表示法，後面幾章則把焦點放在 NextGen POS 個案研究情境，讓你學習、進行物件設計。

閱讀接在本章後面的幾章以瞭解一些設計指引

本章中先簡介表示法。爲了產生設計良好的物件，我們必須先了解設計原則。熟悉互動圖的表示法之後，請研究接下來的章節以瞭解這些（設計）原則，並且把原則應用在畫互動圖的工作上是件很重要的事。

第一節循序圖與合作圖

互動圖這個術語代表 UML 中兩種特殊化的圖；這兩種圖可以表達類似的訊息互動情形（譯註：有些 CASE 工具甚至可以幫你互轉兩種圖）：

- 合作圖
- 循序圖

本書中兩種圖都有用到，以強調它們的選擇彈性。

合作圖（collaboration diagram）用平面圖或網路格式展示物件互動情形，在這種圖中物件可以放在圖的任何地方，如圖 15.1 所示。

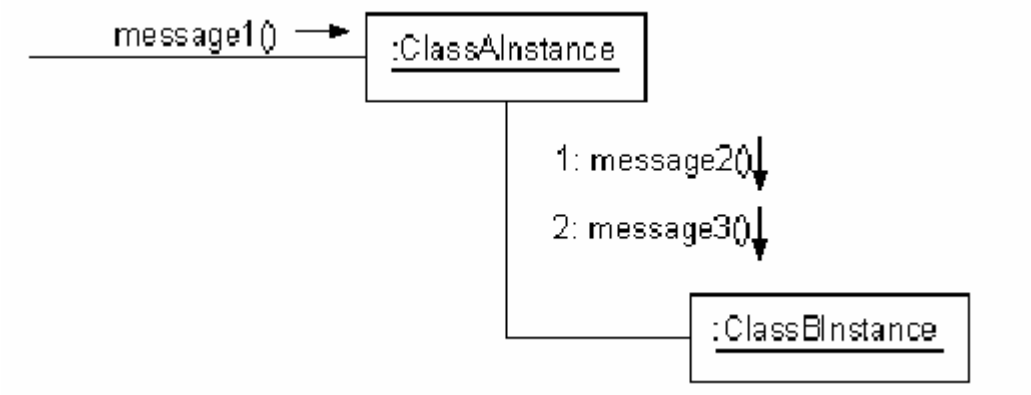


圖 15.1 合作圖

循序圖（sequence diagram）用柵欄格式展示物件互動情形，在這種圖中新的物件會加在圖的右方，如圖 15.2 所示。

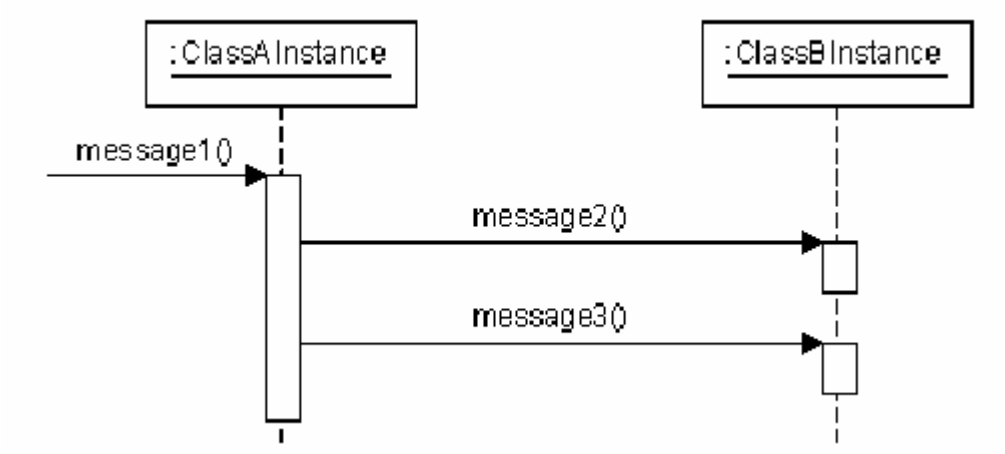


圖 15.2 循序圖

這兩種圖分別都有強點與弱點。當我們想把圖印在頁面窄的紙上時，合作圖可以把新的物件放在頁面的垂直方向（譯註：合作圖的強點）；而循序圖中的新增物件則必須放在右方，這是一種限制。不過，合作圖的範例很難看清楚訊息的先後順序（譯註：合作圖的弱點）。

當我們用 CASE 工具對原始碼做反向工程產生互動圖時，大部分的人比較偏好用循序圖，因爲循序圖可以很清楚地展現訊息的先後順序（譯註：其實這很直覺，程式碼中本來就沒有空間相關訊息，又怎麼可能會產生好的合作圖圖面配置呢？）

圖形種類	強點	弱點
循序圖	清楚顯示出訊息的先後順序或時間先後 簡單的表示法	一定要從右方加入新的物件；會用掉水平空間
合作圖	比較節省空間—可以很彈性地在兩個空間維度任意加入物件	很難看出訊息的先後順序 比較複雜的表示法

	比較容易展示複雜的分支條件、反覆與並行行爲	

譯註：不管用循序圖或合作圖，只要我們想在一張圖上表現太多資訊，結果都會面臨空間不足的窘況。就以循序圖來說，水平方向放置物件，垂直方向放置訊息互動情形。如果這張圖上互動的物件太多，水平空間自然不足；如果圖上的訊息互動情形太多、太細，又換成垂直空間不足。其實，最大的問題在於：如何用一張圖表現出「適量」的內容。在水平方面，如果這張圖的重點在參與者與系統的互動，就不要畫出太多系統內部的物件；如果重點在分層結構中的領域層，就不要放太多 UI 層的物件在圖上（例如又是 jsp 、又是 HTML）。在垂直方面，如果這張圖所涵蓋的使用案例情節很長，就不要畫得太詳細，如果想畫詳細一點，所涵蓋的情節就短一點。

第二節合作圖範例： makePayment

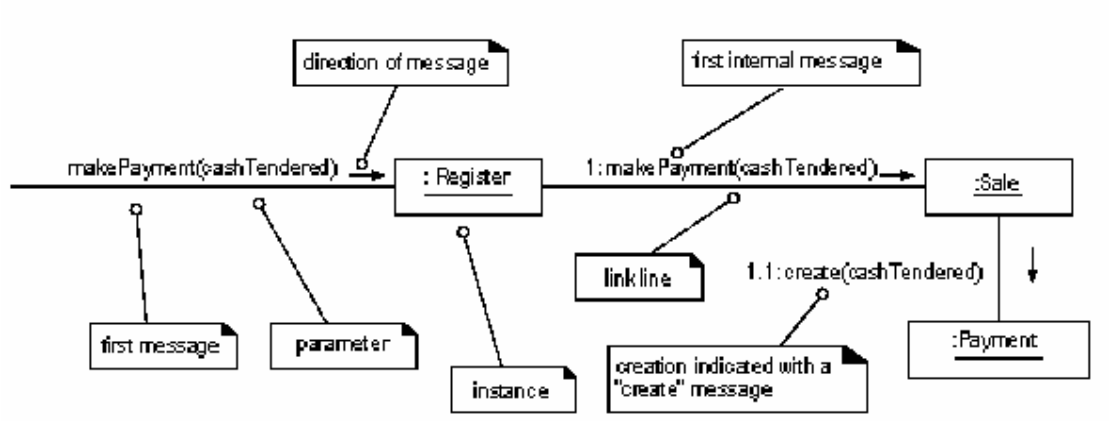


圖 15.3 合作圖。

圖 15.3 中的合作圖可以用下面方式解讀：

1. 送 *makePayment* 訊息到 *Register* 實例。圖中沒有標示出發出訊息者。
2. *Register* 實例送 *makePayment* 訊息到 *Sale* 實例。
3. *Sale* 實例負責產生 *Payment* 實例。

第三節循序圖範例： makePayment

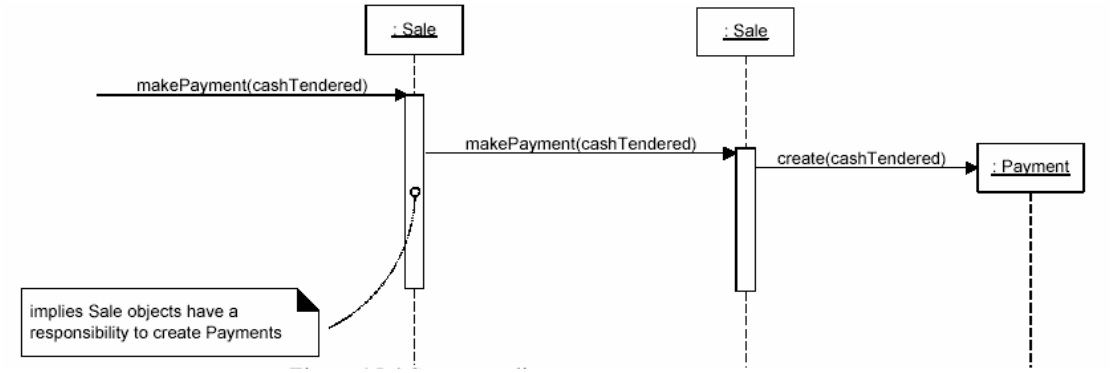


圖 15.4 循序圖。

圖 15.4 所顯示的循序圖，其內涵跟之前的合作圖一樣。

第四節*互*動圖是很有價值的

使用物件技術的專案中一個常見的問題是：沒有正確認知用*互*動圖進行物件設計的價值所在（譯註：指派責任到物件。）另一個相關的問題則是畫*互*動圖時不夠明確，例如送給物件的訊息事實上可能需要更進一步的詳細設計；舉例來說，只送一個 *runSimulation* 訊息給 *Simulation* 物件，而沒有繼續畫出更詳細的設計結果，好像認爲只要有良好命名的訊息，就可以很神奇地完成設計結果（譯註：被指派的責任太大塊。）

我們需要花不少時間與工作量產生*互*動圖，因爲產生物件設計細節的思考過程需要花很多的時間與工作量。例如如果反覆的固定時間長度爲兩個星期，在還沒有寫程式之前，我們可能要在反覆剛開始時，花大約半天或一天時間產生這些*互*動圖（同時也會畫出類別圖。）沒錯，圖中所展示出來的設計結果可能不完整、也很投機，寫程式時需要做些修改，不過它提供我們一個思考過、有一致性、共同的起點，作爲寫程式時的靈感來源（譯註：如果要雞蛋裡面挑骨頭，還是要先有雞蛋是嗎？）

建議

產生*互*動圖時，最好兩個人一起進行，不要單獨畫*互*動圖。兩人一起合作可以彼此修正設計結果，彼此快速學習（譯註：三個臭皮匠，勝過一個諸葛亮。兩人式開發其實是作者從 XP 中擷取的最佳實務經驗）

設計技能（包括樣式、實作樣式與設計原則）主要是在這個開發步驟中需要用到的。相對來說，使用案例、領域模型或其它工作成果的製作比指派責任、產生設計良好的*互*動圖還容易。因爲一大堆設計原則的差異很細微（譯註：有些情況會同時符合好幾個設計原則的條件，而且有些設計原則的結果不同，這時候我們就不容易釐清到底要用哪個設計原則了），而且設計良好的*互*動圖，其「自由度」

比大部分物件導向分析與設計的工作成果還要大（譯註：自由度大的意思就是：建模型者可採用的設計方式有很多種選擇，每種選擇各自有優缺點，這樣一來，自由度自然就大的。）

在物件導向分析與設計中，產生互動圖（換言之，決定物件設計細節）是非常有創造性的開發步驟。

我們可以應用編纂起來的樣式、設計原則與實作樣式改善設計的品質（譯註：學習設計樣式時如果按圖索驥，就像是填鴨式教育，題目做得越多，分數越高，本書所介紹的 GRASP 樣式則比較像啓發式教學，訓練你的思考過程。成長時只有營養均衡，發育才會良好。）

成功建構互動圖時所需的設計原則可以用方法論的方式加以編纂、解釋與應用。這種瞭解並使用設計原則的解決方案是以**樣式**（pattern）—結構化的指引與原則—爲基礎的。因此，介紹互動圖語法之後，接下來（在後面的章節中）我們會把重點放在設計樣式以及如何把設計樣式用在互動圖中（譯註：如果分析與設計之間有一段鴻溝的話，設計樣式就是一座橋。）

第五節常見的互動圖表示法

展現類別與實例

UML 用一種簡單、有一致性的方式區分**實例**（instance） vs. 有行爲能力者（classifier）（請參見圖 15.5。）

- 不論是哪種 UML 元素（類別、參與者等等），實例跟型態的表示法都用同樣的圖形符號，不過實例的字串會加上底線。

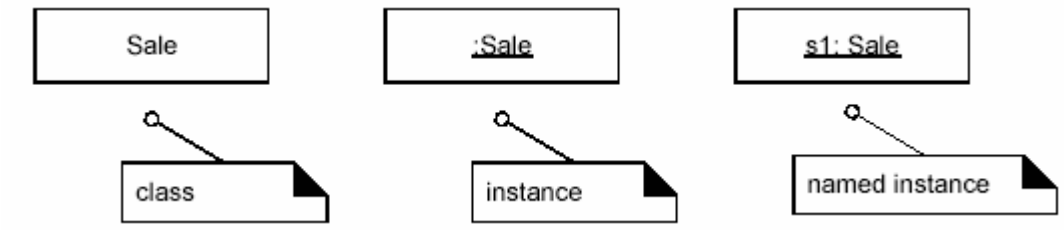


圖 15.5 類別與實例。

因此，爲了在互動圖中秀出類別的實例，我們會用標準的類別方塊圖形符號，並且在實例名稱加上底線。

我們可以用名稱標示唯一的實例。如果沒有用實例名稱的話，就用「:」加上類別名稱（譯註：請注意，這時候雖然是用類別名稱，不過只要加上底線之後整個方塊圖形就代表實例，只是這個實例沒有名稱而已；如果這個方塊圖形代表類別，一方面不需要加上底線，另一方面沒有「:」。）

基本訊息表示式的語法

UML 中有訊息表示式的標準語法：

RETURN := MESSAGE(PARAMETER : PARAMETERTYPE) : RETURNTYPE

如果很明顯不需要型態資訊或型態資訊不重要的話，就不要秀出型態資訊。例如：

spec := getProductSpect(id)

spec := getProductSpect(id:ItemID)

spec := getProductSpect(id:ItemID) : ProductSpecification

第六節基本的合作圖表示法

繫結

繫結（link）代表兩個物件之間的連結路徑；我們用它標示物件之間存在某種形式的瀏覽性與可見性（請參閱圖 15.6。）更正式的說法是：繫結是關聯的實例（譯註：這種說法是根據 UML 的超模型【meta-model】，把關聯視爲類別、繫結視爲實例。）例如在 *Register* 與 *Sale* 之間存在一個繫結—或瀏覽路徑，繫結上面可能會有訊息流的存在，例如 *makePayment* 訊息。

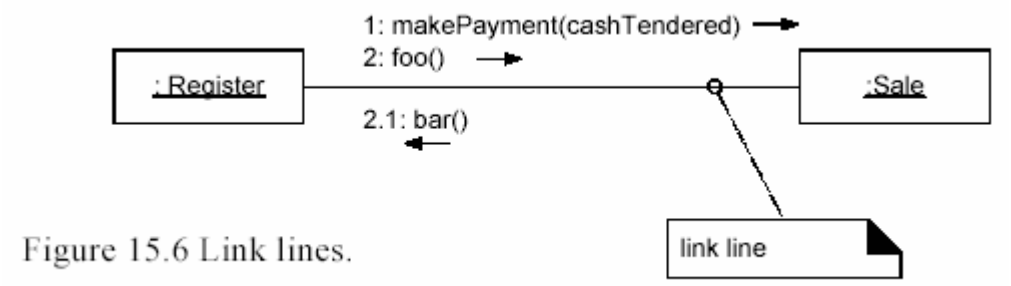


Figure 15.6 Link lines.

圖 15.6 代表繫結的線條。

請注意，在同一條繫結上，可能會有好幾個訊息或不同方向的訊息流過（譯註：如果合作圖是一張地圖的畫，物件就是不同的城市、繫結代表連接城市之間的道路，而訊息就是在道路上的車輛。）

訊息

物件之間的每個訊息都會用訊息表示式與小的箭頭標示訊息方向。繫結上可能會有許多個訊息流過（請參見圖 15.7。）我們可以針對目前的控制執行緒，依序排列把序號秀在訊息上。

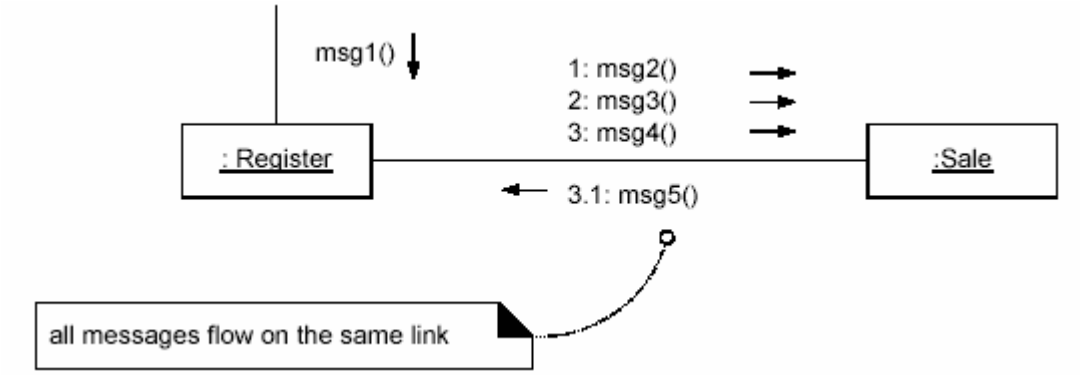


圖 15.7 一些訊息。

傳回「自身」或「this」的訊息

訊息可以傳回到物件本身（請參見圖 15.8。）我們用連回物件自身的繫結表示，繫結上的訊息會流回物件。

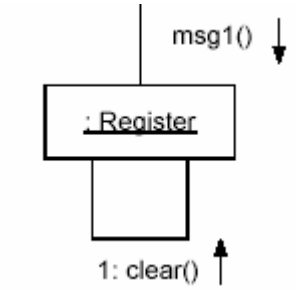


圖 15.8 傳到「this」的訊息。

實例的產生

我們可以用任何訊息產生實例，不過在 UML 中有個慣例，就是用叫做 *create* 的訊息產生實例。如果用其它名稱的話（意義上可能變得沒有那麼明顯），那麼我們可能要用 UML 的另一種語言特性：造型（例如 *((creat))*）爲訊息加上註解（譯註：造型是進一步分類後的結果，如果訊息是一般性的，那麼 *create* 訊息就是一種特殊型態的訊息；在意義上，等於(1).一般性訊息的含意加上(2).產生實例的含意。）

create 訊息可能會帶參數，以設定實例的初始值。例如在 Java 裡面呼叫帶參數的建構子。

此外，我們也可以在實例方塊圖形上選擇性加上 UML 屬性 *{new}* 以強調這是新產生的實例。

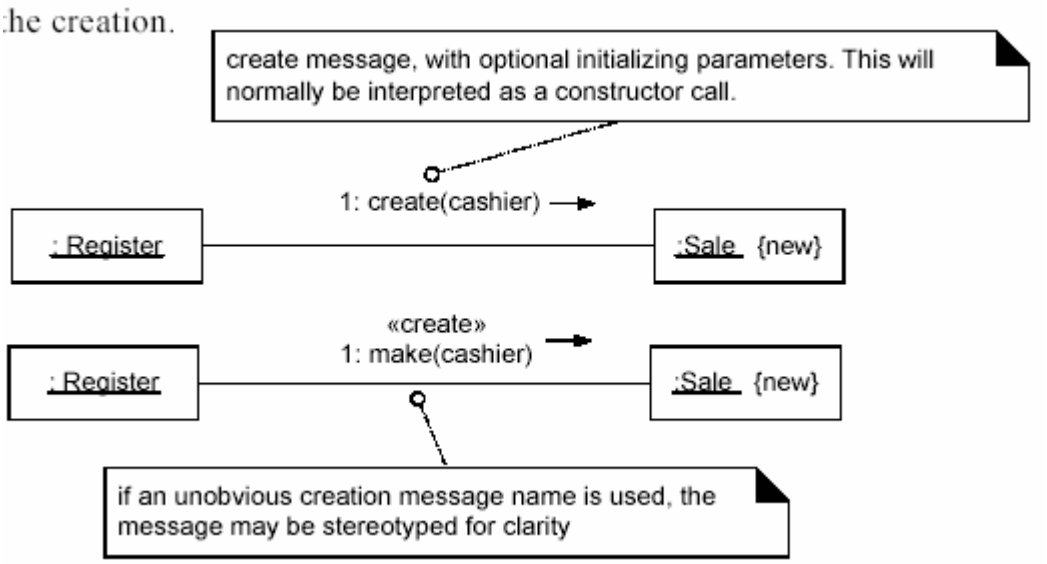


圖 15.9 實例的產生。

訊息的編號順序

我們可以用**序號**（sequence number）說明訊息順序，如圖 15.10 所示。圖中的編號方式爲：

1. 第一個訊息沒有編號。因此， *msg1()* 前面沒有數字。
2. 接下來訊息的編號順序與巢狀結構則是用合法的編號方式，遇到巢狀訊息時編號就會加一層數字。我們可以從進入訊息的號碼與離開訊息的號碼看出巢狀結構。

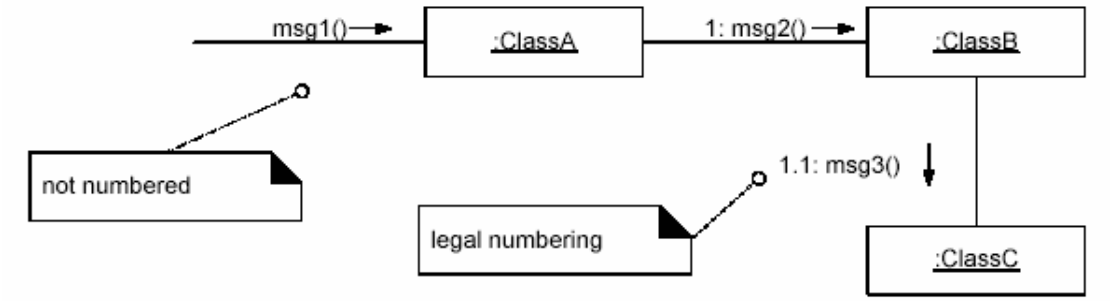


圖 15.10 序號編號方式

圖 15.11 是更複雜的例子。

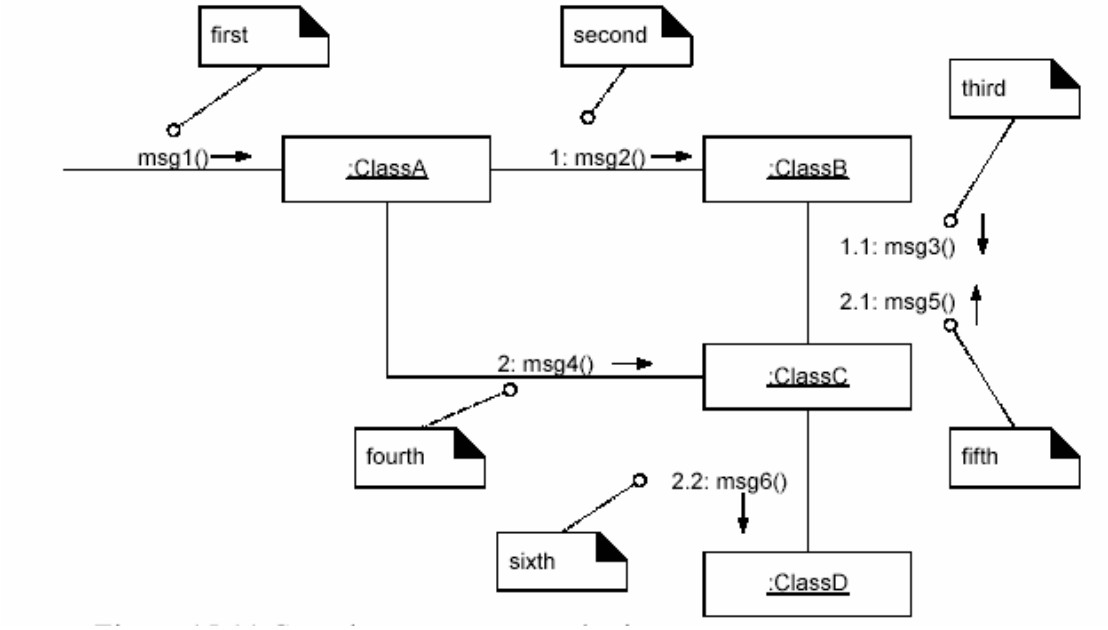


圖 15.11 複雜的序號編號方式

條件式訊息

我們在序號的後面用方括號條件子句顯示條件式訊息（圖 15.12），這種表現方式跟反覆子句很像。條件成立時，我們才會送出訊息。

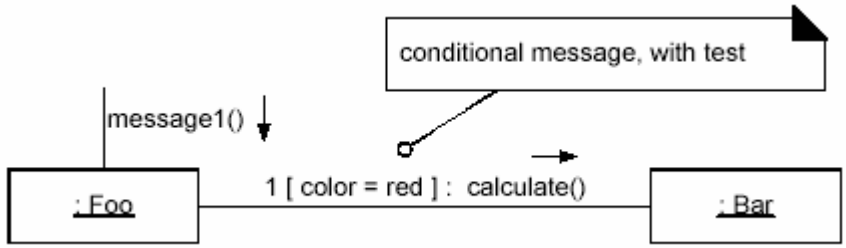


圖 15.12 條件式訊息

彼此互斥的條件式路徑

圖 15.13 所展示的序號代表彼此互斥的條件式路徑。

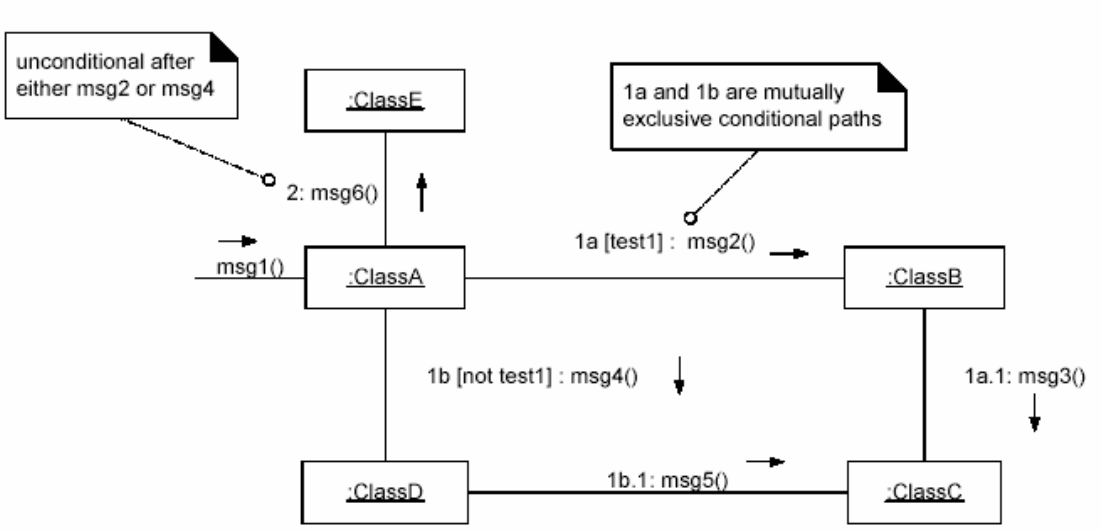


圖 15.13 彼此互斥的訊息

在這個情況，我們需要用條件式路徑字母修改序列表示式（sequence expression）：按照慣例 a 是第一個字母。圖 15.13 說明 *msg1* 之後會執行 *1a* 或 *1b* 。這兩個訊息的序號都是 1 ，因爲它們都是第一個內部訊息。

請注意，接下來的巢狀訊息依然符合離開它們的訊息編號方式。因此，*1b.1* 是 *1b* 的巢狀訊息。

反覆或迴圈

圖 15.14 所示的是反覆的表示法。如果對建模型者來說，反覆子句的細節不是那麼重要，就可以用簡單的「*」代表反覆。

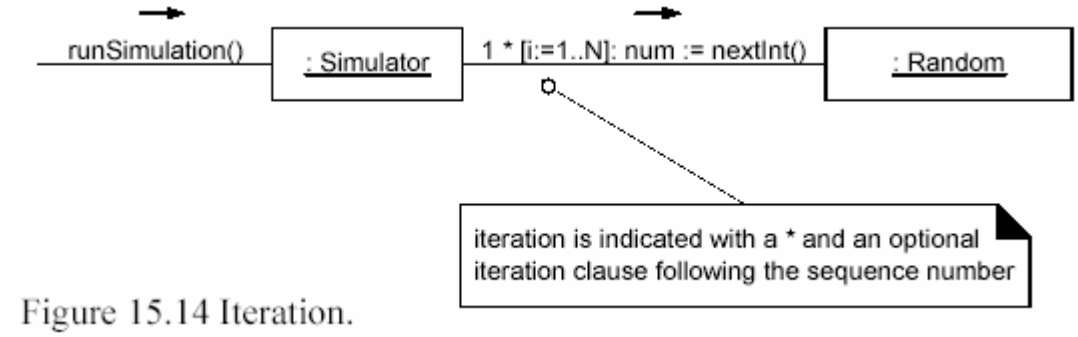


Figure 15.14 Iteration.

圖 15.14 反覆。

用反覆處理多重物件

用反覆處理多重物件（collection 或 multiobject）（例如 list 或 map）中的所有成員是常見的演算法，此時我們會送訊息給多重物件中的每個成員。通常我們最後都會用某種反覆子（iterator）物件實作多重物件（譯註：多重物件代表介面），

例如 java.util.Iterator 的實作或 C++ 標準函式庫中的反覆子。在 UML 中，**多重物件**（multiobject）代表一組實例— collection 。在合作圖中，我們可以用像圖 15.15 的方式代表多重物件。

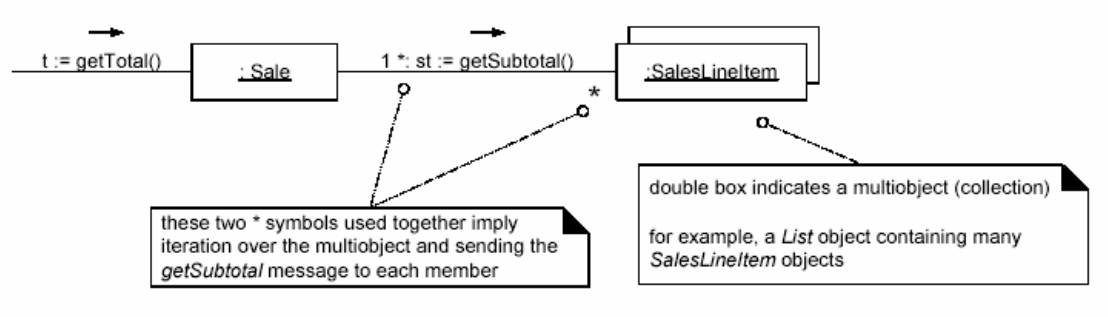


圖 15.15 用反覆處理多重物件。

在繫結端點上的「*」多重性標示是說明訊息是送給多重物件中的每個成員，而不是重複送訊息到多重物件本身。

傳到類別型物件的訊息

訊息也可以送到類別而不是送到實例以呼叫類別方法或**靜態方法**（static method）。這時候，訊息所送到的類別方塊圖形名稱不會加上底線，以表示訊息是傳給類別而不是實例的（請參見圖 15.16。）

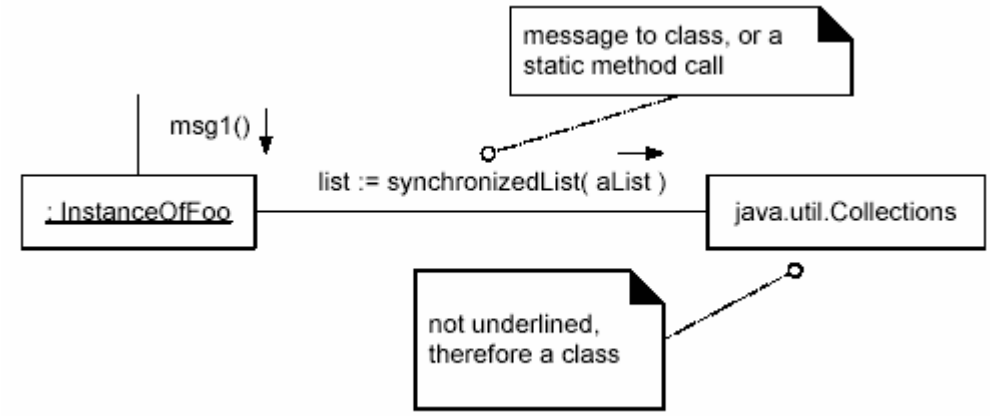


圖 15.16 傳到類別型物件的訊息（靜態方法呼叫）

因此，把你想要的實例名稱加上底線是很重要的事，這樣一來才有一致性，否則就無法正確辨識出訊息是傳到實例或類別的。

第七節基本的循序圖表示法

繫結

跟合作圖不同，循序圖中不會顯示繫結（譯註：不論是循序圖或合作圖，它們都需要用適當方式組織在圖上流竄的訊息，循序圖是用時間先後順序組織訊息，而合作圖則是用繫結加上編號組織訊息。）

訊息

物件之間的每個訊息都是用物件之間、有箭頭的線條加上訊息表示式形成的（請參見圖 15.17。）訊息發生時間的先後順序則是依照圖由上而下排列。

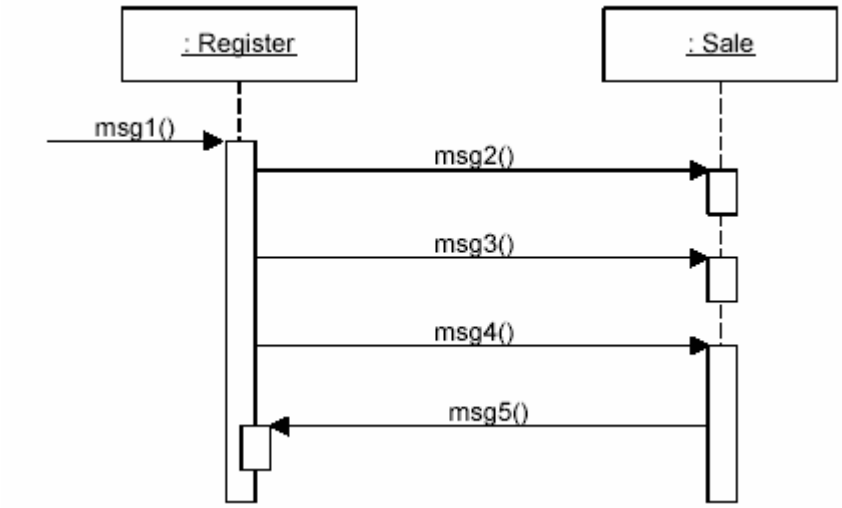


圖 15.17 訊息與用活化長條形標示的控制權焦點

控制權焦點與活化長條形

如圖 15.17 所示，循序圖中也可以用**活化長條形**（activation box）秀出控制權焦點（換言之，在標準、會轉移控制權的【blocking】呼叫中，操作會被放在呼叫堆疊【call stack】中。）這個長條形是依建模型者需要選用的，不過 UML 實踐者通常會用它。

展示（呼叫後的）傳回訊息

我們也可以選擇要不要秀出循序圖中訊息的傳回訊息（return），它在活化長條形

端用的是開放式箭頭，而且線條是虛線（請參見圖 15.18。）許多 UML 實踐者不會畫出傳回訊息。有些人則爲了說明從訊息傳回的東西（如果有的話）而畫出傳回訊息。

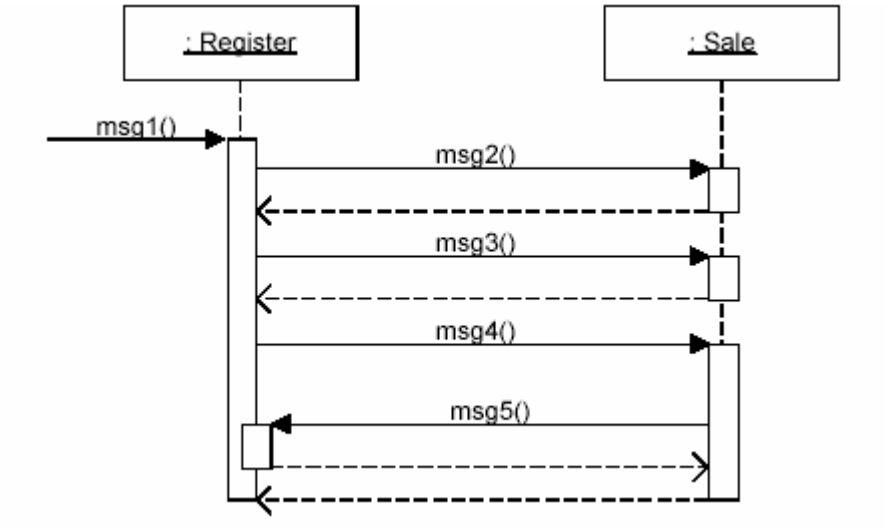


圖 15.18 秀出傳回訊息。

傳回「自身」或「this」的訊息

我們可以用巢狀的活化長條形展示傳回物件本身的訊息（請參見圖 15.19。）

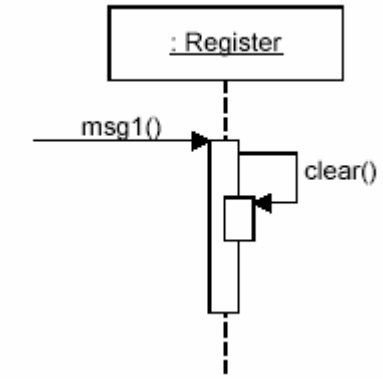


圖 15.19 傳回「this」的訊息。

實例的產生

圖 15.20 秀出訊息會產生物件時的表示法。

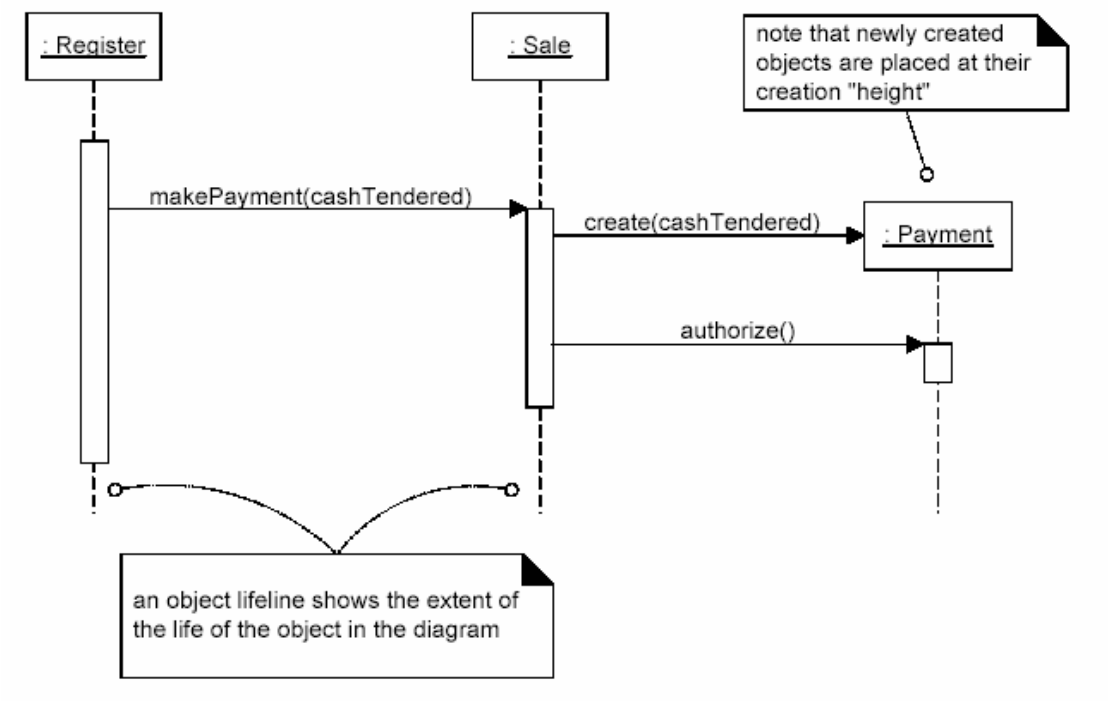


圖 15.20 實例的產生與物件生命線。

物件生命線與物件的消失

圖 15.20 中也秀出**物件生命線**（object lifeline）—每個物件底下的垂直虛線。我們用這條虛線說明圖中物件的生命範圍。在某些情況下，我們想要明確秀出物件的消失（例如在沒有提供記憶體垃圾蒐集服務的 C++ 中）；UML 的生命線表示法可以讓我們展現出物件的消失（如圖 15.21。）

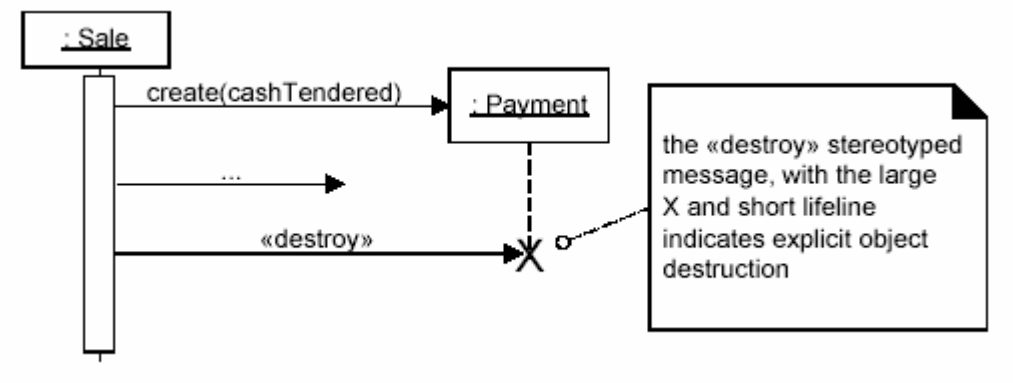


圖 15.21 物件的消失。

條件式訊息

圖 15.22 所展示的是條件式訊息。

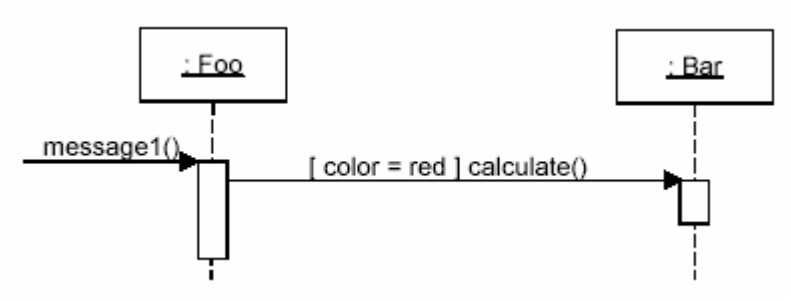


圖 15.22 條件式訊息。

彼此互斥的條件式路徑

圖 15.23 所展示的是從共同點發出、有角度的訊息。

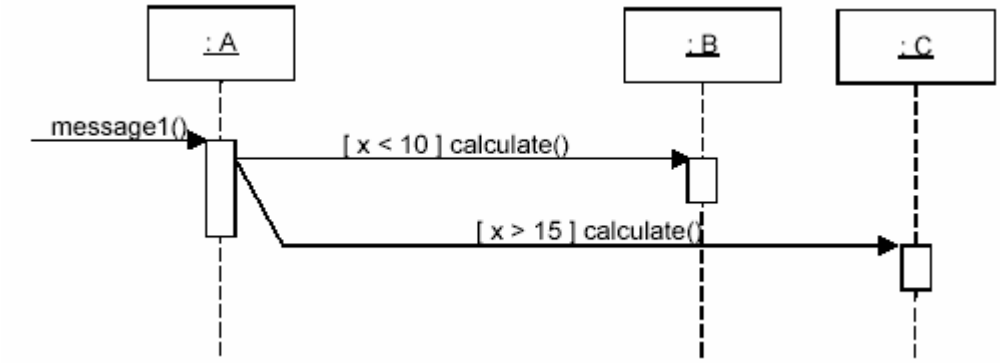


圖 15.23 彼此互斥的條件式訊息。

涵蓋單一訊息的反覆

圖 15.24 是涵蓋單一訊息的反覆。

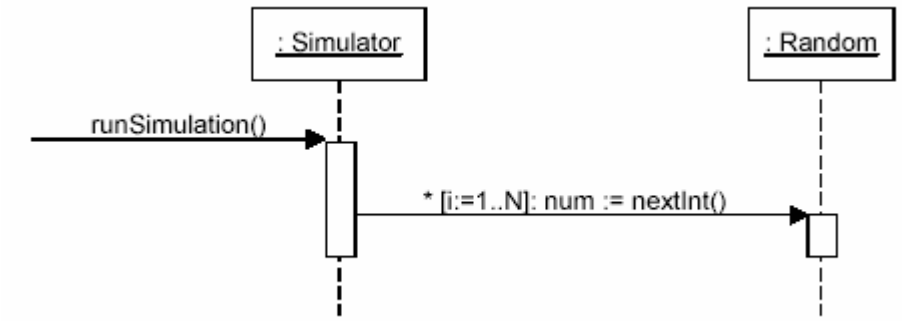


圖 15.24 涵蓋單一訊息的反覆。

涵蓋一組訊息的反覆

圖 15.25 則是涵蓋一組訊息的反覆。

用反覆處理多重物件

在循序圖中，我們用圖 15.26 的方式顯示出用反覆處理多重物件（collection 或 multiobject）的情形。
在合作圖中，我們是在（靠近多重物件的）角色端點上用「*」多重性標示出這個訊息會送給多重物件中的每個成員，而不是重複送給多重物件本身。然而，UML 在循序圖中並沒有任何方式可以表現這個狀況（譯註：其實可以在訊息的箭頭端加上「*」多重性，不過這不是 UML 的規定。）

傳到類別型物件的訊息

跟合作圖一樣，我們可以在類別方法或靜態方法所呼叫的有行爲者（classifier）名稱上加上底線，說明這是類別物件而不是實例（請參見圖 15.27。）

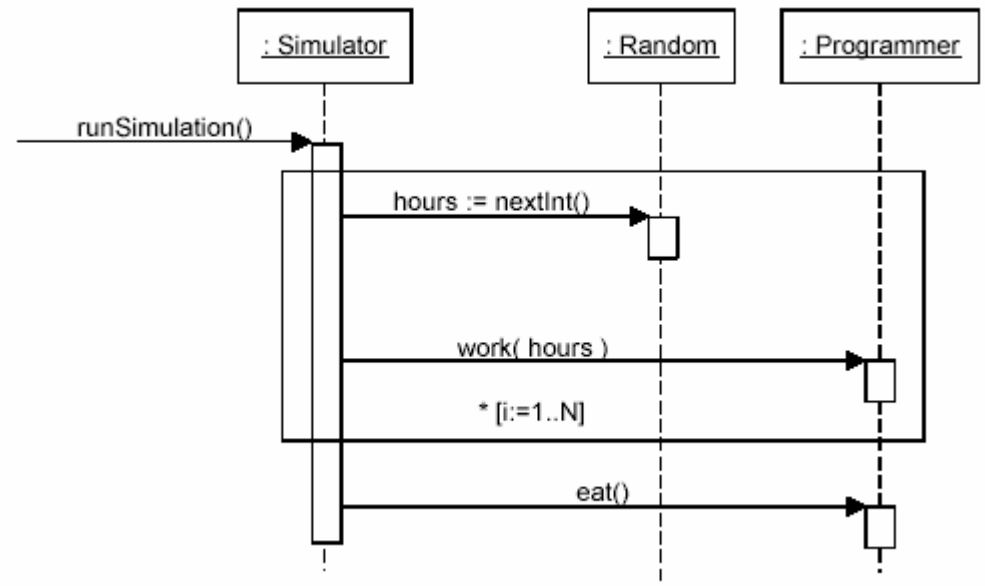


圖 15.25 涵蓋一組訊息的反覆。

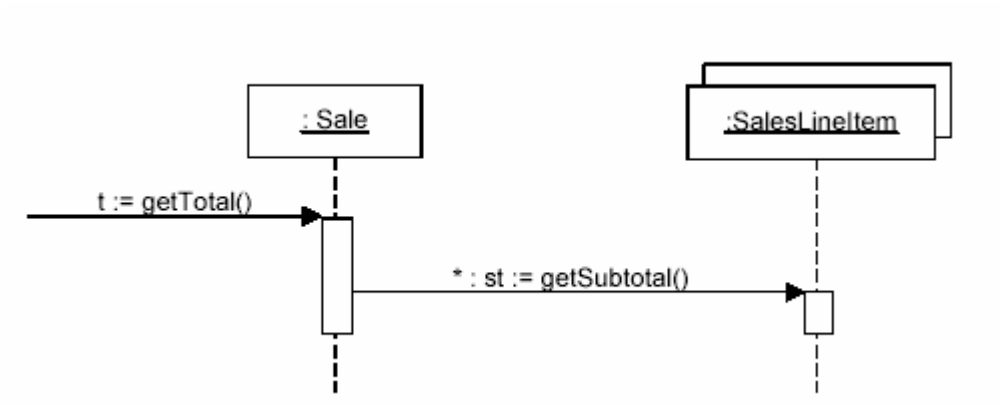


圖 15.26 用反覆處理多重物件。

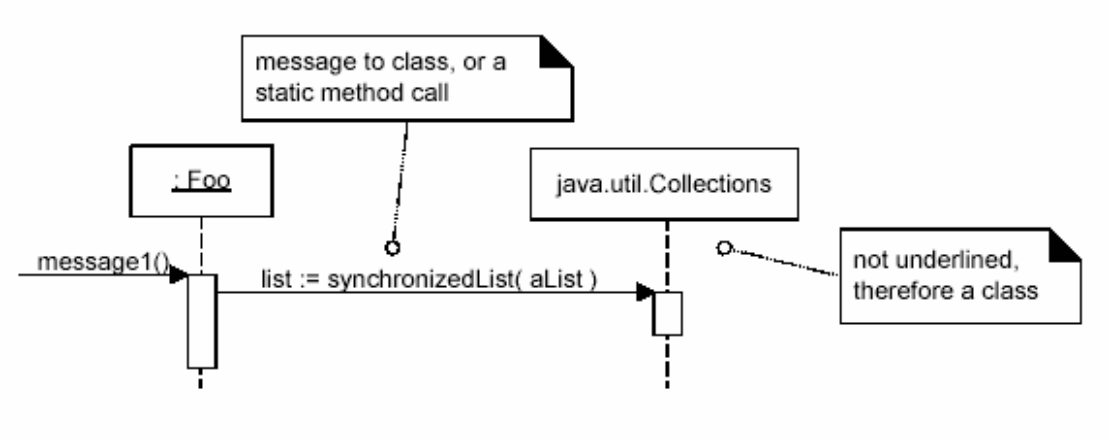


圖 15.27 呼叫類別方法或靜態方法。

第十六章GRASP：根據責任設計物件

大部分專家都同意：意外是最可能摧毀世界的真兇。我們這些電腦專家則是最可能造成意外的人。

— Nathaniel Borenstein

本章目標

- 定義樣式。
- 學習如何活用 GRASP 中的五個樣式。

簡介

物件設計過程跟下面所說的過程大同小異：

> 找出需求、產生領域模型之後，接下來的工作就是把方法加到軟體類別中，並且爲了完成需求，定義物件之間的訊息傳送方式。

像上面這麼簡單的建議對大家來說其實幫助不大，因爲這些步驟裡面存在許多更深層的設計原則與設計議題。決定某個訊息應該放在哪裡、物件之間如何互動，都是很重要卻很繁瑣的事。本書會小心解釋這些步驟，說明大家畫 UML 圖、寫程式時可以用得上的概念。

物件設計是開發物件導向系統的關鍵步驟－整個開發過程的核心工作不是像畫畫領域模型圖、套件圖等事項，而是物件設計工作。

把 GRASP 當成學習基本物件設計、很有條理的學習法

我們說明 GRASP 基本物件設計方式中的詳細設計原則與過程，並且教你把它們應用在物件設計方法論上，這些方法論通常不會談到這些神奇之處，省略一些讓人混淆不清之處。

GRASP 樣式是一種學習輔助工具，它可以幫助你了解物件設計本質，並且用方法論、合理、可解釋的方式應用在設計過程中。GRASP 樣式裡面所介紹並使用的設計原則都跟指派責任的樣式有關。

第一節責任與（實作責任的）方法

UML 中把**責任**（responsibility）定義成「有行爲者（classifier）的合約或契約」【OMG01】。物件的契約責任反映在物件行爲上。基本上來說，這些責任可區分成下面兩種：

- 知道事情的責任（knowing）
- 辦事的責任（doing）

物件的辦事責任包括：

- 自行完成的事，例如產生物件（譯註：造物主物件）或者做一個計算工作
- 對其它物件做初始化動作
- 控制並協調其它物件的活動（譯註：控制者物件）

物件知道事情的責任包括：

- 知道私有、封裝起來的資料爲何（譯註：私有屬性、受保護屬性）
- 知道相關物件爲何（譯註：跟其它物件之間的關聯）
- 知道可推導出來或算出來的東西（譯註：導出屬性）

我們在物件設計過程中把責任指派給物件的類別。舉例來說，我可能宣稱「 *Sale* 必須負責產生 *SalesLineItems* 」（辦事的責任），或者「 Sale 必須知道自己的總金額爲何」（知道事情的責任。）知道事情責任的歸屬通常可以從領域模型中推理出來，因爲這些責任會呈現在屬性與關聯上。

把責任放到類別與方法上這件事會因爲責任的大小粗細程度而有所不同。例如「提供關連式資料庫的存取方式」的責任可能會跟數十個類別、數百個方法有關，我們需要把這些類別包裝成子系統。另一方面，「產生 *Sale* 」的責任可能只跟一個或幾個方法有關。

責任不等於方法，不過我們會在方法的實作中完成被賦予的責任。我們可能用單獨執行的方法實作責任，這個方法也可能會跟其它方法合作完成責任。舉例來說，*Sale* 裡面可能會定義一個或數個知道總金額的方法，例如 *getTotal* 。爲了完成這個責任， *Sale* 可能需要跟其它物件一起合作，例如傳送 *getSubtotal* 訊息給每一個 *SalesLineItem* 物件，詢問它們的小計金額。

第二節責任與互動圖

本章的目的在於協助大家有系統地把基本設計原則應用在指派物件的責任上。寫程式時經常需要做這件工作。在 UML 的工作成果中，考慮責任（由方法實作）歸屬問題的時機點通常是產生互動圖時（互動圖是統一流程【UP】設計模型的部份工作成果），我們已經在前一章解釋過互動圖的基本表示法。

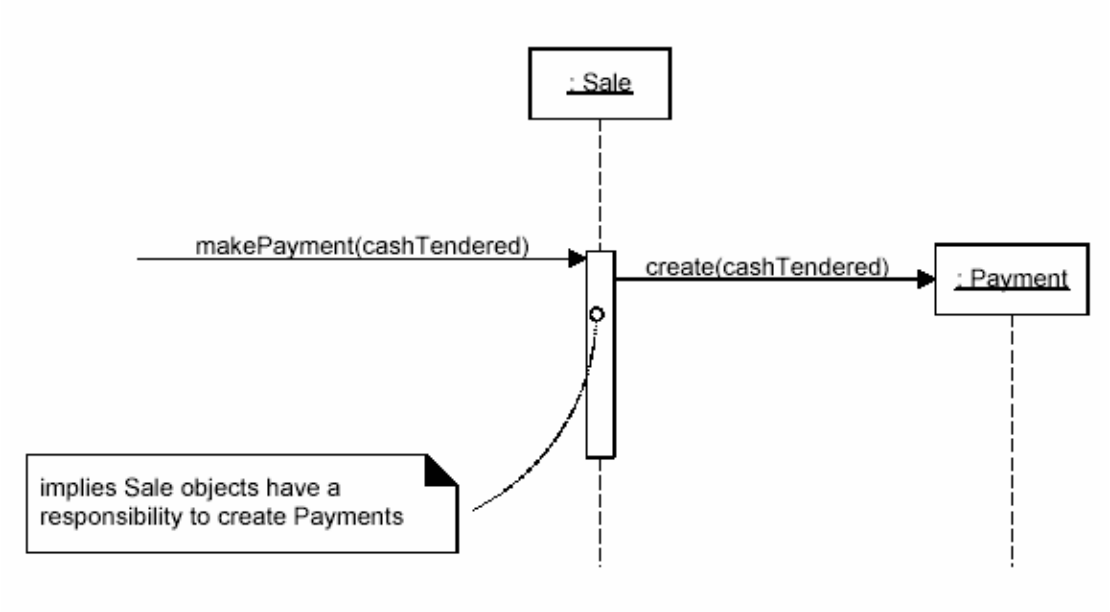


圖 16.1 責任與方法之間的關連性

我們可以從圖 16.1 中看出：產生 *Payments* 是 *Sale* 被賦予的一個責任，我們會用 *makePayment* 訊息呼叫它，而由 *Sale* 中相對應的 *makePayment* 方法負責處理這個訊息。此外，爲了完成這個責任， *Sale* 需要跟 *SalesLineItem* 合作，產生 *SalesLineItem* 物件、呼叫它的建構子（constructor）。

總而言之，畫互動圖的過程中需要做一些決策，看看責任應該被指派到哪個物件身上。做這些決策後的結果是：訊息分別被傳送到不同物件的類別身上。本章裡面教你如何把責任指派到類別上，強調一些基本的設計原則－ GRASP 樣式 。而指派責任時所做的選擇最後會反映在互動圖上。

第三節樣式

有經驗的物件導向開發人員（以及其它軟體開發人員）都會自行保有一份清單，裡面記載一般性的設計原則與慣用的解決方案，並且利用這份清單完成軟體開發工作。如果用結構化的格式描述這些設計原則與慣用方案的問題與解決方案，並且給它一個名稱，那麼我們就可以稱它爲**樣式**（pattern）。舉例來說，下面是一個樣式的範例：

樣式名稱	資訊專家樣式
解決方案	把責任指派給擁有相關資訊的類別。
樣式所解決的問題	指派物件責任的基本設計原則是什麼？

在物件技術中，樣式代表某個問題與相關解決方案的說明與名稱，而且新情境依然可以應用相同解決方案。在理想狀況下，樣式中還會提供一些忠告，教我們如何把樣式應用在各種情境下、考慮影響設計的各種力量與如何做取捨【註】。許多樣式裡面都會提到一些指導原則，教我們遇到特定類型時，如何指派物件責任。

【註】：樣式的正式概念起源於 Christopher Alexander【AIS77】的 the (building) architectural patterns。軟體的樣式則是 Beck 在 1980 年代所提倡的，他先察覺到 Alexander 在架構方面的樣式工作成果，稍後他和 Ward Cunningham【BC87、Beck94】一起開發軟體樣式。

簡單來說，樣式就是一組命名過的問題／解決方案，我們可以把它們應用在新情境下，樣式裡面也提供一些忠告，教我們如何在新環境活用樣式，並且討論一些設計上的取捨。

「某人的樣式就是另一個人的設計基礎材料（primitive building block）」這句技術方面的格言說明樣式這種東西是一種很模糊的概念【GHJV94】。這種處理樣式的方式跳過樣式名稱的議題，把焦點放在務實面，用樣式風格替有用的軟體工程原則命名、呈現、學習，並且把它記錄下來。

樣式是會重複出現的

如果有一個新樣式真的是在描述新概念，那麼裡面可能會有矛盾、不完美之處需要修飾。我們用這個特別的字「樣式」代表會不斷重複發生的東西。樣式的重點不在於表達出新設計概念。樣式的重點反而在於寫下現存、錘鍊百鍊過的知識、慣用解決方案與設計原則，被修飾的越多、用的越廣泛，就代表樣式越好。

因此，稍後介紹的 GRASP 樣式 並不是新概念，我們只是把一些被廣泛用到的基本設計原則寫下來。物件專家把這些 GRASP 樣式概念（而不是名稱）視爲非常基本的東西，也非常熟悉它們。這才是重點！

樣式必須有名稱

在理想狀況下，所有樣式都會有建議名稱。替樣式、技術或設計原則命名有下面的好處：

- 有助於把概念集結成塊、具體化，讓我們易於了解與記憶。
- 有助於溝通。

替樣式這樣複雜的概念命名正好可以說明抽象化的力量－抽象化就是刪去細節，把複雜形式簡化的過程。因此，GRASP 樣式都有一個簡潔名稱，例如資訊專家樣式、造物主樣式、預防變異樣式等等。

替樣式命名有助於改善溝通

一旦樣式有了名稱之後，我們就可以用簡單的名稱跟其他人討論複雜的設計原則或設計概念。請看看下面，兩個軟體設計師之間的一段討論，他們用常見的樣式字彙（造物主樣式、工廠樣式等等）決定設計方式：

Fred：「你認爲我們應該把產生 *SaleLineItem* 的責任擺在哪裡？我認爲應該放在

工廠（物件）上。」
Wilma：「根據造物主樣式，我認爲 *Sale* 比較合適。」
Fred：「沒錯！我同意你的看法。」
常見、易於了解的名稱可以協助我們彼此溝通整個設計慣用解決方案與設計原則，並且讓我們用更抽象的概念探討問題。

第四節GRASP 樣式：分配責任時常會用的設計原則

下面彙總前面的一些說明：

- 熟練指派責任是物件設計中最重要的事。
- 指派責任通常發生在產生互動圖時，當然也會發生在寫程式時。
- 樣式是成對、命名過的問題／解決方案，裡面編纂很好的忠言與設計原則，這些東西通常跟指派責任有關。

問題：什麼是 GRASP 樣式？
答案：GRASP 樣式裡面用樣式的形式描述基本物件設計原則與指派責任原則。

瞭解這些樣式，並且在產生互動圖時應用這些樣式是很重要的事，因爲物件技術方面的軟體開發新手需要盡快專精這些基本設計原則；這些設計原則是我們如何設計出系統的基石。
GRASP 是 **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns【註】的縮寫。我們用這樣的名稱以強調：*心領神會*這些設計原則對成功設計出物件導向軟體的重要性。
【註】：從技術層面來說，我們應該寫成「GRAS Patterns」而不是「GRASP Patterns」，不過後者的讀音比較好聽。

如何應用 GRASP 樣式

後面幾個小節中分別介紹 GRASP 樣式的前五個樣式：

- 資訊專家樣式（information expert）
- 造物主樣式（creator）
- 高內聚力樣式（high cohesion）
- 低耦合力樣式（low coupling）
- 控制者樣式（controller）

稍後章節中會再介紹其它樣式，不過這五個樣式非常值得你熟練它們，因爲它們所針對的都是非常基本、常見的問題與基本設計議題。
請研讀接下來的樣式，並且注意它們是如何被應用在互動圖範例中，以及如何應

用在產生互動圖過程中。一開始，先熟練資訊專家樣式、造物主樣式、高內聚力樣式、低耦合力樣式與控制者樣式，稍後再學習其它樣式。

第五節 UML 中類別圖的表示法

UML 中展示軟體類別的類別方塊圖形通常包含三個區隔；我們在第三個區隔中展示類別的方法，請參見圖 16.2。

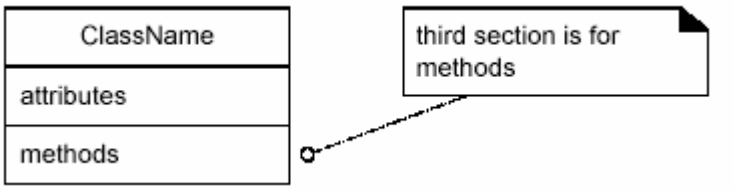


圖 16.2 在軟體類別中展現方法名稱。

我們會在下一章說明類別圖表示法的細節。下面對樣式的討論中可能偶而會出現這種類別方塊圖形。

第六節資訊專家（或專家）樣式

解決方案

把責任分派給資訊專家（information expert），它擁有必要資訊，可以完成被賦予的責任。

問題

指派物件責任的通則爲何？

設計模型裡面可能會有幾百甚至幾千個軟體類別，而且在應用程式裡面可能需要完成幾百甚至幾千個責任。進行物件設計以定義物件之間互動情況時，我們同時也會決定要指派責任給哪個軟體類別。如果我們能夠很恰當地指派責任，系統將很容易理解、維護或擴充，並且在未來應用程式中，有更多機會可以重新使用（現有的）元件。

範例

在 NextGen POS 應用程式中，有些類別需要知道銷售總金額。

開始指派責任之前，先清楚描述責任。

根據這個建議，我們重新描述責任成：

誰有責任需要知道一筆銷售的總金額？

根據資訊專家樣式，我們試圖尋找哪些物件的類別擁有決定總金額的必要資訊。接下來是關鍵問題：分析哪些類別擁有必要資訊時，要看領域模型還是設計模型？領域模型中的類別代表真實世界、問題領域中的概念性類別，而設計模型中的類別則代表軟體類別。

答案：

1. 如果設計模型中有相關類別，就先看設計模型。
2. 否則，請看領域模型，並且嘗試用（或擴充）這個概念性類別產生相對應的設計類別。

舉例來說，假設我們才剛開始進行設計工作，完全沒有或只有很少的設計模型。因此我們需要從領域模型中找尋資訊專家，或許我們找到的是真實世界中的 *Sale* 概念性類別。接下來，我們在設計模型中新增相似的軟體類別，同樣命名爲 *Sale*，然後把知道總金額的責任指派給它，並且把這樣的方法命名爲 *getTotal*。這種做法可以得到很小的*觀點呈現差距*（representational gap），因爲物件的軟體設計方式跟真實領域中的概念組織方式類似。

爲了詳細了解這個例子中的細節，請看圖 16.3 中的部分領域模型。

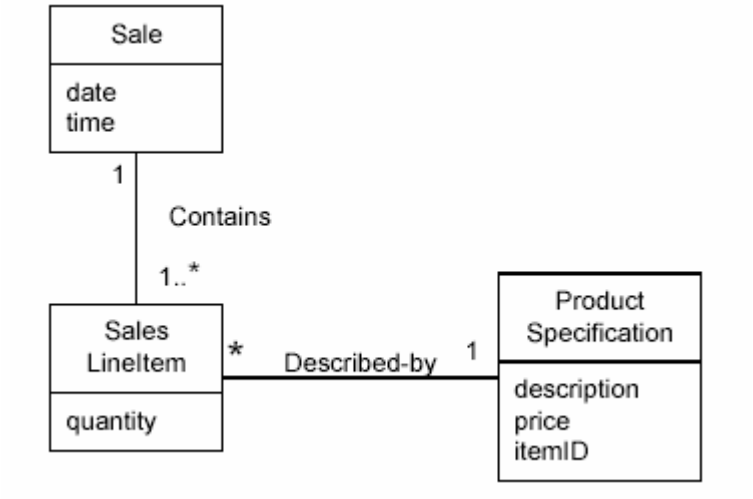


圖 16.3 跟 *Sale* 有關的一些關聯

決定總金額時需要哪些必要資訊呢？我們需要知道跟這筆銷售相關的所有 *SalesLineItem* 實例以及它們的小計金額。因爲 *Sale* 實例中包含這些 *SalesLineItem* ，因此根據資訊專家樣式，*Sale* 類別的實例適合負起知道總金額的責任，它是這項工作的資訊專家。

前面已經提過，畫互動圖時，我們通常會遇到責任歸屬問題。想像現在我們正準備畫圖，把責任分派給物件。圖 16.4 裡面部分的互動圖與類別圖，秀出我們可能下的一些決定。

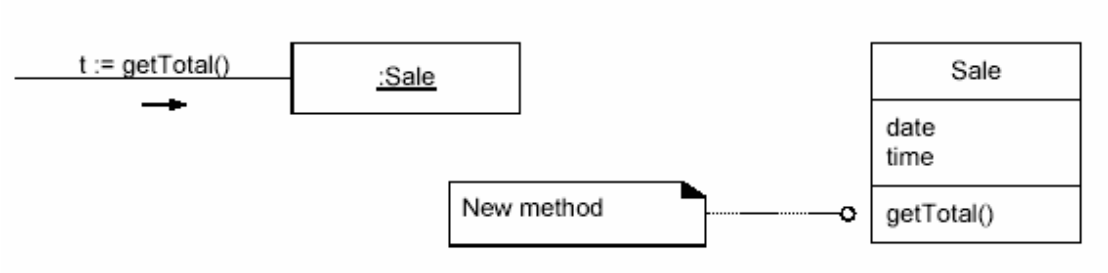


圖 16.4 部分的互動圖與類別圖

事情還沒有結束。決定商明細項目的小計金額時，哪些是必須資訊？

SalesLineItem.quantity 與 *ProductSpecification.price* 都是必要資訊。由於 *SalesLineItem* 知道自己的數量與相關的 *ProductSpecification* ，因此根據資訊專家樣式，應該由 *SalesLineItem* 決定小計金額，它是這項工作的資訊專家。

從互動圖看來，代表 *Sale* 需要發出 *getSubtotal* 訊息到每個 *SalesLineItem* ，並且加總所有結果，圖 16.5 顯示設計結果。

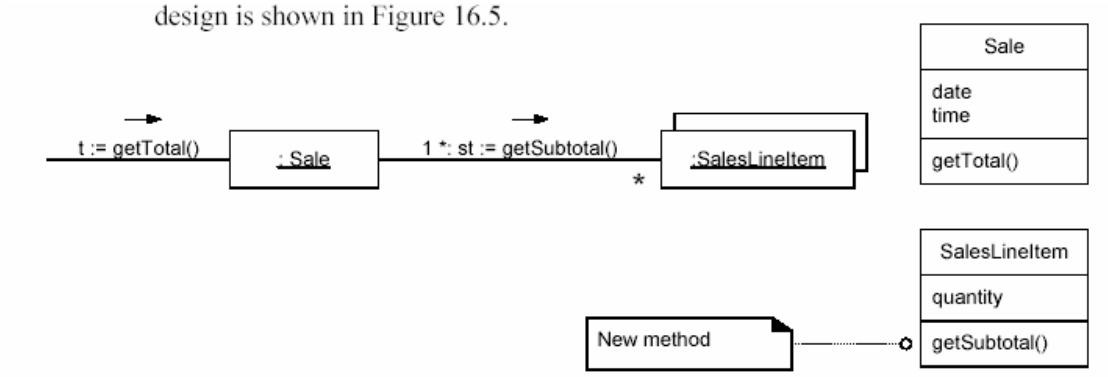


圖 16.5 計算 *Sale* 的銷售總金額

爲了完成知道並回答小計金額的責任，*SalesLineItem* 需要知道產品價格。

ProductSpecification 是回答產品價格的資訊專家，因此 *SalesLineItem* 會送訊息給 *ProductSpecification* ，詢問它的價格。

設計結果請參閱圖 16.6。

總而言之，爲了滿足知道並回答銷售總金額的責任，我們分別把三個責任指派給三種設計類別，如後所示。

設計類別	責任
Sale	知道銷售總金額
SalesLineItem	知道銷售明細項目的小計金額
ProductSpecification	知道產品價格

我們畫互動圖時會考慮並決定這些責任要指派給那個物件。在類別圖中，類別的方法部分會顯示出它所擁有的所有方法。

資訊專家樣式是我們指派責任的設計原則之一，這個樣式把責任指派給擁有資訊的物件，由這個物件負責。

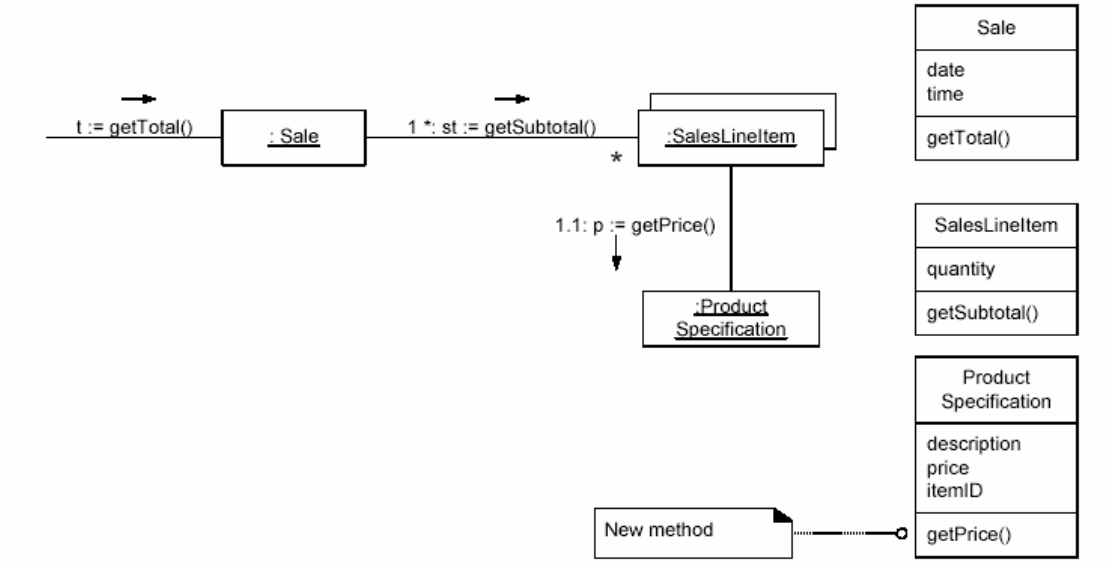


圖 16.6 計算 Sale 的銷售總金額。

討論

指派責任時常會用到資訊專家樣式，它是物件設計過程中持續用到的一項基本原則。專家樣式並不是一種難以理解的概念或怪想法，它是一種常見的「直覺式」設計方式，把事情交給擁有相關資訊的物件做。

請注意，爲了完成一項責任，有可能經常需要其它類別的物件提供資訊。換言之，一項工作除了主要資訊專家之外，還可能會跟許多「區域性」資訊專家合作。就算出銷售總金額這個問題來說，總共需要三種類別的物件合作。

當資訊分散在不同物件身上時，這些物件需要透過訊息互動一起分擔工作。

專家樣式的設計結果通常爲：負責操作的軟體物件通常是真實世界中無生命事物， Peter Coad 把這種設計方式稱爲「自己動手做（do it myself）」策略【Coad95】。舉例來說，在真實世界中，如果不使用機電工具，銷售無法自行告訴你總金額，因爲它是無生命事物，需要靠某人才能算出銷售總金額。不過，在物件導向軟體世界中，所有軟體物件都是「活的」，它們可以負擔責任做事。基本上，軟體物件會負責做跟它們所知道資訊有關的事。我把它稱爲物件設計中的「動畫」原則，因爲在動畫中，所有東西都像是有生命的。

就像物件技術中的許多事物一樣，我們也可以在真實世界中找到跟資訊專家樣式相對應的東西。在真實世界中，我們通常把責任交給擁有必要資訊的人處理。例如，在公司裡面，誰負責產生損益表？有權限存取必要資訊的人必須負責產生損益表，這個人有可能是財務長。就像軟體物件之間的合作關係一樣，（財務）資訊分散在各處，負責的人需要跟許多人互動。因此公司財務長可能需要跟會計要借貸報表。

例外情形

有時候根據專家樣式所得到的解決方案可能不是我們想要的，最常發生的問題是：耦合力與內聚力的問題（本章後面會討論到這些設計原則。）

舉例來說，誰應該負責把 *Sale* 儲存到資料庫中呢？顯而易見，大部分資訊都放在 Sale 物件中，因此根據專家樣式觀點，應該由 *Sale* 類別負責。依照這樣的邏輯推論下去，最後每個類別都會擁有儲存自己到資料庫中的服務。這種做法會引起內聚力、耦合力與重複問題。舉例來說， *Sale* 類別中必須自己處理資料庫相關邏輯，例如跟 SQL、JDBC （Java Database Connectivity）相關的程式邏輯。這時候，類別將不再純粹只有「進行一筆銷售」的應用程式邏輯，它必須負起其它責任，降低類別中的內聚力。此外，這個類別也必須跟其它子系統中的技術性資料庫服務產生耦合力，例如 JDBC 服務，而不是跟領域層中其它軟體物件產生耦合力，結果引起耦合力問題。此外，許多永續類別中擁有許多相似、重複的資料庫邏輯。

上面這些問題都違反了基本架構設計原則：設計時分離系統不同的主要考量因素。換句話說就是：把應用程式邏輯保持在一個地方（例如領域軟體物件中）、資料庫邏輯保持在另一個地方（例如一個分離的永續服務子系統），而不是把不同的系統考量因素通通混在同一個合成關係中【註】。

【註】：請參閱第 32 章中對分離（系統）考量因素的討論。

假設分離（系統）主要考量因素可以改善設計中的耦合力與內聚力。那麼根據專家樣式，我們應該把資料庫服務責任交給 *Sale* 類別，不過，其它設計原則（通常是內聚力與耦合力）告訴我們這是一種不好的設計方式。

優點

- 專家樣式可以維持資訊封裝，因爲物件用自己的資訊完成工作。通常這樣做可以達成低耦合力（低耦合力也是一種 GRASP 樣式，後面章節會討論到），讓系統更強固、易於維護。
- 行爲分散到擁有必要資訊的類別身上，可能產生更多具內聚力的「輕量級」類別定義，這些類別將易於理解與維護。專家樣式通常可以達到高內聚力（稍後會討論到這個樣式。）

相關樣式或相關設計原則

- 低耦合力樣式
- 高內聚力樣式

其它別名或相似樣式

「把責任與資料放在一起（Place responsibilities with data）」、「讓知道（資訊）的人辦事（That which knows, does）」、「自己動手做（Do It Myself）」、「把服務跟服務用到的屬性放在一起（Put Services with the Attributes They Work On）」。

第七節造物主樣式

解決方案

如果下面任何一個情況成立的話，把產生類別 A 實例的責任交給類別 B：

- B 中*聚合（aggregate）* A 的物件。
- B *包含（contain）* A 的物件。
- B 負責*紀錄（record）* A 的物件實例。
- B *密切用到（closely use）* A 的物件。
- B 中*有*產生 A 時會用到的*初始化資料（have initializing data）*（因此 B 是產生 A 的專家。）

B 是 A 物件的*造物主（creator）*。

如果負責產生 A 物件的選擇有好幾個的話，我們傾向於選擇*聚合（aggregate）*或*包含（contain）*類別 A 的類別 B。

問題

誰應該負責產生某些類別的新實例？

產生物件是物件導向系統中最常見的活動之一。因此，如果有通則可以指派產生物件的責任將非常有幫助。如果責任分配妥當的話，設計結果將具有低耦合力、設計邏輯也會更清晰、也可以增加封裝性與再使用性（reusability）。

範例

在 POS 應用程式中，誰應該負責產生 *SalesLineItem* 的實例呢？根據造物主樣式，我們應該尋找*聚合（aggregate）*或*包含（contain）SalesLineItem* 實例的類別。請看一下圖 16.7 中的部分領域模型。

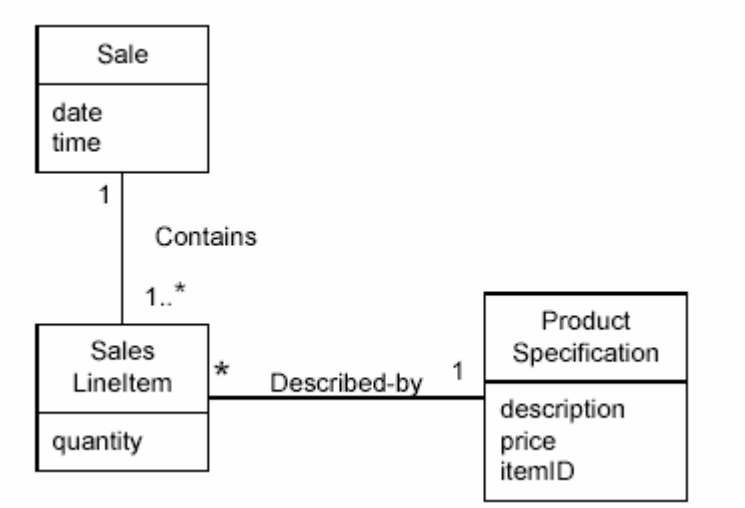


圖 16.7 領域模型的一部分

因爲 *Sale* 裡面*包含（contain）*（事實上是*聚合 aggregate*）許多 *SalesLineItem* 物件，所以造物主樣式建議我們：*Sale* 是產生 *SalesLineItem* 實例的好候選類別，可以把責任加在 *Sale* 身上。

設計的物件互動情形如圖 16.8 所示。

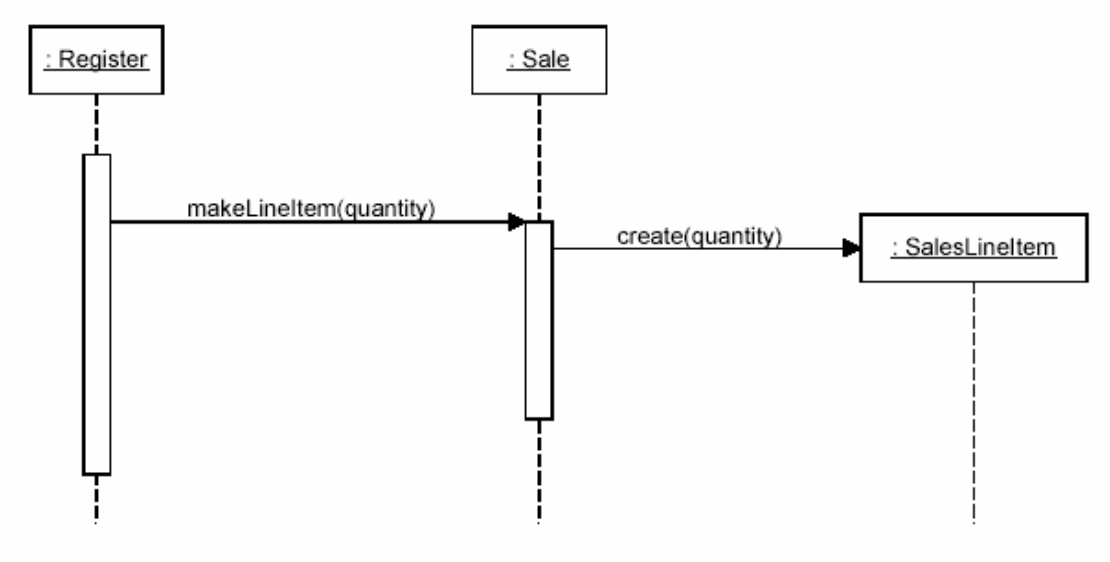


圖 16.8 產生 *SalesLineItem*。

指派責任給 *Sale* 的結果是我們需要在 *Sale* 上新增 *makeLineItem* 方法。

重複一次，我們會在畫互動圖時考慮並指派責任。類別圖中各個類別的方法部分是指派責任後的結果。換句話說，責任的指派是用方法實現（定義）的。

討論

造物主樣式引導我們指派跟產生物件有關的責任，這是一種常見的責任。它的基本內涵是：想找出因爲某種原因而跟被產生物件連結在一起的造物主（物件）。這樣的造物主（物件）有助於產生低耦合力的設計結果。

聚合體*聚合（aggregate）*零件、容器*包含（contain）*內容以及紀錄者*紀錄（record）*被紀錄者等情況都是類別圖中常見的關係。造物主樣式建議我們：封裝其它東西的容器或紀錄者類別是產生被包含者或被紀錄者的很好候選類別。當然，這只是一項指導原則。

請注意，造物主樣式中會考量聚合關係這個概念。我們會在第 27 章討論聚合關係，這裡先給一個簡短定義：聚合關係代表兩個東西之間有很堅固的整體－部分關係（whole-part）或組合體－零件關係（assembly-part），例如身上有（*聚合 aggregate* ）腿或者段落中有（*聚合 aggregate* ）句子。

有時候，我們可以找尋產生類別時會用到的初始化資料，以找到我們要的造物主物件。事實上，這也是專家樣式的一個例子。初始化資料是在產生物件時，透過某個初始化方法（initializing method）傳送給被產生物件的，例如帶參數的 Java 建構子。舉例來說，假設產生 *Payment* 實例時，需要用 *Sale* 的總金額做初始化動作。因爲 *Sale* 知道總金額，所以它是 *Payment* 的候選造物主。

例外情形

通常，產生物件的動作複雜度很高，例如爲了效能理由用回收的實例、以外顯屬性（property）的值爲條件從一家族相似的類別中產生實例等等。遇到這樣的情況，你可以把產生物件的動作委託給協助類別（helper class）：*Factory*【GHJV95】，而不要用造物主樣式所建議的類別。我們會在第 23 章討論工廠樣式（Factory）。

優點

- 造物主樣式的設計結果將具有低耦合力（稍後會介紹），而低耦合力隱含比較低的維護依賴關係，也會有比較高的再使用機會。耦合力之所以不增加是因爲造物主可能早就看得見被產生者，這個既存的關聯是我們選擇前者成爲造物主的原因。

相關樣式或相關設計原則

- 低耦合力樣式
- 工廠樣式
- 整體－部分樣式【BMRSS96】中所定義的聚集物件支持合成關係封裝。

第八節低耦合力樣式

解決方案

指派責任後不會提高耦合力。

問題

如何讓相依性（dependency）小、變動的影響不大，並且增加再使用？
耦合力代表一個元素連到其它元素的關係強弱、一個元素對其它元素的了解，或者一個元素依賴其它元素的程度。一個有低（或者弱）耦合力的元素並不會依賴太多其它元素。雖然是否「太多」還要看當時環境決定，不過我們還是檢查得出來。（檢查一個元素跟其它元素間的相依性時，）要檢查的元素包括類別、子系統、系統等等。
一個類別如果跟許多其它類別之間有高（或者強）耦合力，這樣的類別可能不是我們要的，因爲它可能導致下面問題：

- 相關類別變動時，會造成這個類別連帶變動。
- 很難單獨了解這個類別。
- 很難再使用這個類別，因爲它需要依賴其它額外現存類別。

範例

請看看 NextGen 個案中的部分類別圖：

假設我們需要產生 *Payment* 實例，並且把它跟 *Sale* 關連在一起。哪個類別應該負責產生這個實例？因爲 *Register* 負責「*紀錄（record）*」真實世界領域中的 *Payment*，造物主樣式建議我們把 *Register* 當成產生 *Payment* 實例候選類別之一，然後再由 *Register* 送出 *addPayment* 訊息給 *Sale*，並且把新產生的 *Payment* 實例當成參數傳過去。圖 16.9 反映出這種設計的部分互動圖。

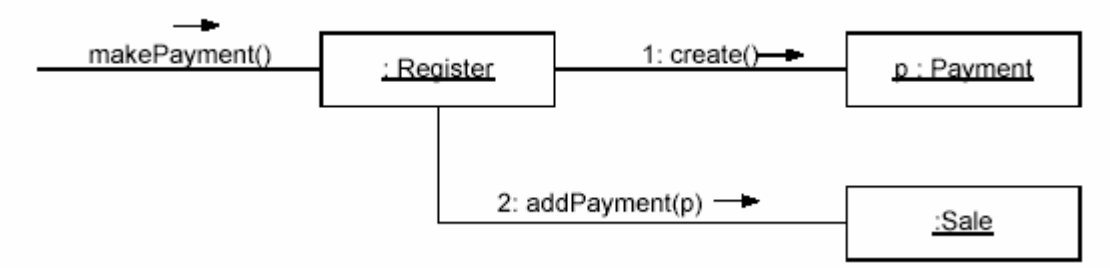


圖 16.9 由 Register 負責產生 Payment 實例。

這種指派責任的方式會產生 *Register* 與 *Payment* 之間的耦合力，*Register* 必須了解 *Payment* 類別。

UML 表示法：請注意，我們把 *Payment* 實例明確命名爲 *p*，因此我們可以在訊息 2 中把它當參數傳過去。

另一種產生 *Payment* 實例的方式是把 *Payment* 與 *Sale* 關連起來，如圖 16.10 所示。

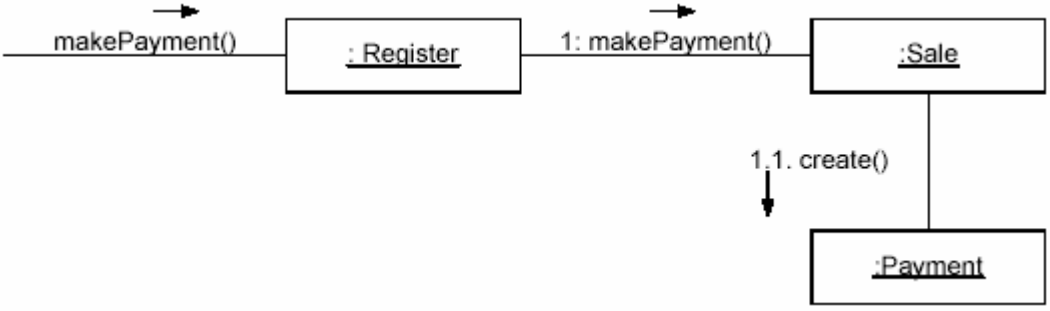


圖 16.10 由 *Sale* 負責產生 *Payment* 。

哪一種責任指派方式可以具有低耦合力呢？在這兩個情況中，我們都假設 *Sale* 與 *Payment* 之間最後都會產生耦合力，*Sale* 必須了解 *Payment*。在設計 1 中，由 *Register* 負責產生 *Payment* ，這會增加兩者之間的耦合力。在設計 2 中，由 *Sale* 負責產生 *Payment* ，此時並沒有不會增加耦合力。單純從耦合力角度來看，我們傾向選擇設計 2 ，因爲它維持整體的低耦合力。在這裡所舉的例子中，兩個樣式－低耦合力樣式與造物主樣式－可能分別建議不同的設計方式。

在實務上，考量耦合力水準時不能不考量其它設計原則，例如專家樣式與高內聚力。然而，耦合力仍然是改善設計結果、需要考量的架構因子之一。

討論

在做任何設計決策時，我們都應該時時把低耦合力設計原則放在心中，它是我們應該時時考量的基本設計目標。同時，它也是我們的**評估原則**（evaluative principle），設計人員在評估設計決策時應該考量它。

在 C++、Java 與 C# 等物件導向程式語言中， TypeX 對 TypeY 有耦合力的常見形式包括：

- TypeX 中有一個屬性（資料成員或實例變數）參考到 TypeY 實例或者 TypeY 本身。
- TypeX 物件會呼叫 TypeY 物件所提供的服務。
- TypeX 中有一個方法參考 TypeY 實例或 TypeY 本身，其中的可能情況包括：擁有型態 TypeY 的參數或區域變數，或者訊息傳回值是 TypeY 實例。
- TypeX 是 TypeY 的直接或間接子類別。
- TypeY 是一個介面，而 TypeX 則負責實作介面。

低耦合力樣式強調指派責任時不要把耦合力水準提高到高耦合力可能的不良影響。

低耦合力能夠讓類別的設計方式更加獨立，以降低變動的影響。考量低耦合力時，還是需要考慮其它樣式，例如專家樣式與高內聚力樣式。不過，指派責任時低耦合力是影響決策的必要設計原則之一。

子類別跟它的超類別之間會發生很強烈的耦合力。從一個超類別衍生子類別時需要仔細考慮，因爲這種關係是一種很強的耦合力。舉例來說，假設物件需要永續儲存在關聯式資料庫或物件式資料庫中。產生一個抽象超類別 *PersistentObject* ，並且讓其它物件從它衍生下來是常見的設計方式。這種子類別化（subclassing）的缺點是它把領域物件跟特定技術服務產生很高的耦合力，並且混合不同架構考量因素。優點是類別會自動繼承永續行爲。

沒有絕對的衡量方式可以讓我們知道什麼時候耦合力太高。重要的是開發人員可以衡量目前耦合力的高低，並且評價增加耦合力時是否會造成問題。如果類別的本質具有一般性，而且有很高的再使用可能性，那麼它應該有非常低的耦合力。

低耦合力樣式的一個極端情況是類別間不存在任何耦合力。我們不希望得到這樣的結果，因爲物件技術隱含系統是透過訊息溝通、相連在一起的物件。如果過份強調低耦合力樣式，可能得到很差的設計結果，因爲它導致一些內聚力差、過份膨脹、複雜的動態物件，它們都是自行完成所有工作，而且許多非常被動、完全沒有耦合力的物件，最後都會變成資料倉庫。產生物件導向系統時，類別之間有中等的耦合力是很正常、必要的，這時候工作將由互相連結的物件合力完成。

例外情形

對穩定、普遍的元素產生高耦合力不會有什麼問題。舉例來說，一個 Java J2EE 的應用程式跟 Java 程式庫（java.util 等等）之間產生耦合力是很安全的，因爲這些程式庫既穩定又普遍。

選對戰場

高耦合力本身並不是問題，跟某些不穩定的元素產生高耦合力才是問題，例如不穩定的介面、實作或不一定存在的元素。

這一點很重要：系統設計師應該增加彈性、封裝細節與實作內容，並且系統許多地方的設計方式都要有低耦合力。不過，我們不應該花時間減少「防範未然」或不具有真實動機地方的耦合力。
系統設計師應該選擇要減少耦合力與封裝細節的地方，把焦點放在真正有高度不穩定性或者會高度演化的地方。例如在 NextGen 專案中，我們已知系統會跟不同協力廠商的稅金計算器配合（有不同介面）。因此，在這個會有變異的地方加強低耦合力設計有實際上的需要。

優點

- 不受其它合成關係變動的影響
- 簡單、易於個別了解
- 方便再使用

相關背景知識

耦合力與內聚力（稍後會說明）兩者都是基本設計原則，所有軟體開發人員都應該體會並應用這些設計原則。在 1960 年代，Larry Constantine 大力倡導耦合力與內聚力，把它們視爲關鍵設計原則【Constantine68, CMS74】，並且認爲我們應該找出、討論它們。他是 1970 年代結構化設計（structured design）的創始者，現在則是可用性工程的提倡者【CL99】。

相關樣式或相關設計原則

- 預防變異樣式

第九節高內聚力樣式

解決方案

指派責任後仍然維持高內聚力。

問題

如何把複雜度保持在可管理的範圍內？
從物件設計來看，**內聚力**（或者更明確點：功能的內聚力）可以用來評估某個元素所負擔的責任相關性是否高、焦點是否集中。如果這個元素的責任有很高相關

性，並且沒有做太多工作，那麼它就具有高內聚力。需要評估內聚力的元素包括：類別、子系統等等。

內聚力低的類別通常會做許多不相關事物或做太多工作。這樣的類別不是我們想要的類別，因爲它可能導致下面問題：

- 很難理解
- 不容易再使用
- 很難維護
- 需要小心處理它，因爲它可能不斷受到其它類別變動的影響

具有低內聚力的類別通常代表很大塊的抽象概念，或者擁有一些應該委派給其它物件的責任。

範例

我們用跟低耦合力樣式相同的例子說明高內聚力樣式。

假設我們現在需要產生（現金的）*Payment* 實例，並且把它 *Sale* 關連在一起。哪個類別應該負起這個責任呢？因爲在真實世界中， *Register* 負責紀錄 *Payment*，所以造物主樣式建議我們由 *Register* 負責產生 *Payment* ，所以接下來 *Register* 實例會送 *addPayment* 訊息給 *Sale*，把新的 *Payment* 當參數送過去。請參見圖 16.11。

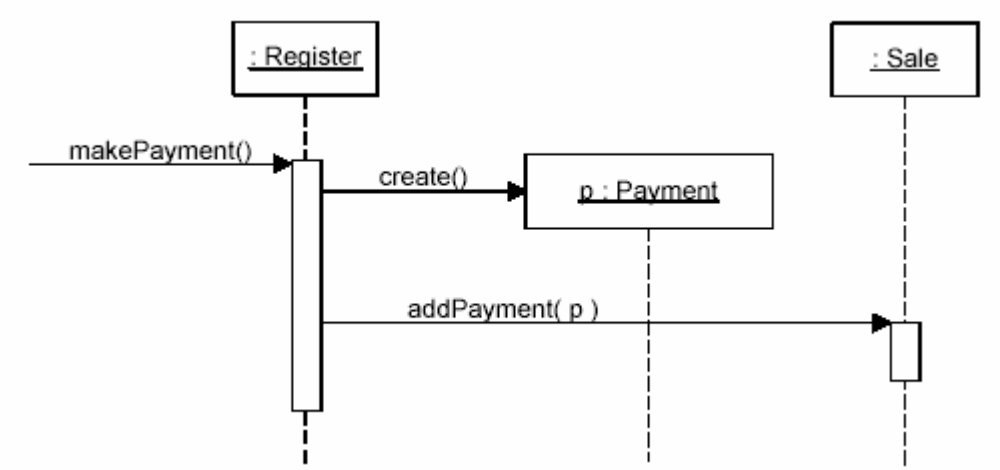


圖 16.11 Register 負責產生 Payment。

這種指派責任的方式把付款責任放在 *Register* 身上，因此 *Register* 需要完成 *makePayment* 系統操作。

從這個單獨例子看來，這種做法是可接受的。不過，如果我們繼續由 *Register* 負責處理越來越多該系統操作相關的一些或大部份工作，那麼 *Register* 的責任就會明顯增加，變得缺乏內聚力。

想像一下 *Register* 負責五十個系統操作時的情形。如果 *Register* 真的負責這些工作，那麼它就會變成過份膨脹、缺乏內聚力的物件。不是產生 *Payment* 的單一工作讓 *Register* 變成缺乏內聚力，而是更大的責任分派範圍讓它變成低內聚

力。
更重要的是：從物件設計師應該具備的開發技能來看，不論最後的設計結果爲何，開發人員至少都應該知道內聚力的影響。
在圖 16.12 中，我們把產生付款物件的責任委託給 *Sale* 做，這樣 *Register* 可以維持高內聚力。
第二種設計方式同時具有高內聚力與低耦合力，這種做法才是我們想要的。

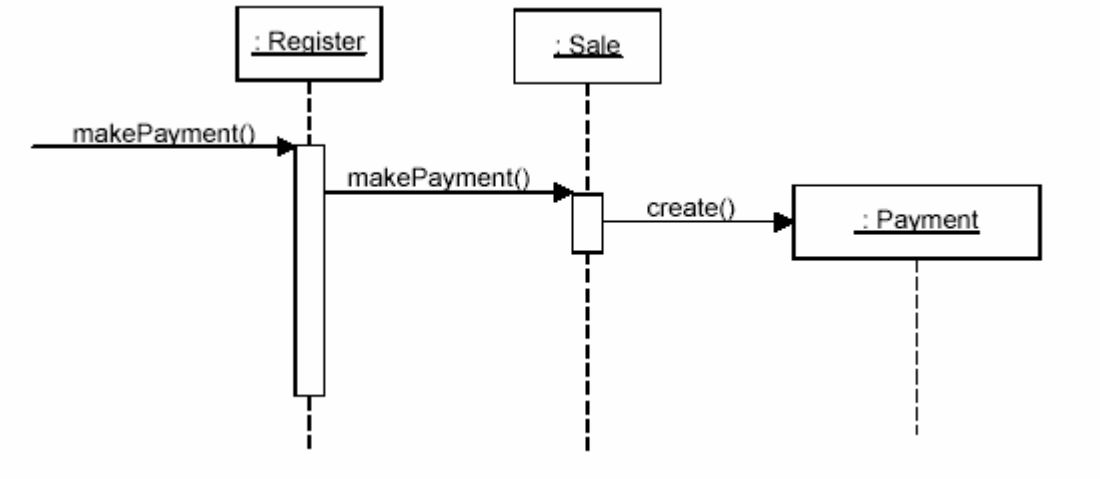


圖 16.12 由 Sale 負責產生 Payment。

事實上，我們不能單獨考量內聚力水準，而不顧其它責任或設計原則，例如專家樣式與低耦合力樣式。

討論

就像低耦合力樣式一樣，當我們進行所有設計決策時，都應該把高內聚力樣式放在心裡，它是必須持續考量的基本設計目的。當系統設計師需要評估設計決策時，它是設計師評估的設計原則之一。
Grady Booth 認爲：當某個元件中的元素能夠「一起合作產生 D 範圍定義良好的一些行爲」，那麼它就具有高的功能內聚力【Booch94】。
下面這些情節中分別展示程度不同的功能內聚力：
1. 很低的內聚力－類別 單獨負責許多各種不同功能範圍的東西。
假設有一個 *RDB-RPCInterface* 類別存在，它完全負責跟關連式資料庫之間的互動，並且處理所有遠端程序呼叫。這是兩個範圍非常不同的功能，而且每個功能範圍都需要許多程式碼。這些責任應該分散到跟 RDB 相關的家族類別以及跟 RPC 相關的其它家族類別。
2. 低內聚力－類別負責某個功能範圍內的複雜工作。
假設有一個 *RDBInterface* 類別，它完全負責跟關連式資料庫之間的互動。這個類別的方法彼此都有相關，不過類別中有太多方法、程式碼存在，類別中可能會有數百個方法。這個類別應該把責任分散到一家族輕量級的類別身上，由這些類別共同完成存取關連式資料庫工作。

3. 高內聚力－類別負責某個功能範圍內的適當責任，並且跟其它類別合作完成工作。

假設有一個 *RDBInterface* 類別，它只承擔跟關連式資料庫之間互動的部份責任，並且跟十幾個其它類別合作以存取關連式資料庫，完成讀取與儲存物件的工作。

譯註：單從作者對低內聚力與高內聚力的文字說明，我們很難感受到它們之間的差異，大家對照第 34 章的例子就可以比較容易了解。其中 *PersistenceFacade* 類別承擔某些物件跟關連式資料庫之間的互動責任，由 *PersistenceFacade* 負責從關連式資料庫中讀出或寫入某些物件資料。如果採用低內聚力做法，*PersistenceFacade* 就必須「單獨」負責所有跟關連式資料庫之間的互動，而高內聚力的做法就是第 34 章所提出的永續框架，由 *PersistenceFacade* 跟 *Persistence* 套件中的 *IMapper*、 *AbstractPersistenceMapper* 、 *RDBMapper* 以及 NextGen *Persistence* 套件中的 *ProductSpecificationRDBMapper* 「共同」負責跟關連式資料庫之間的互動，而 *PersistenceFacade* 只承擔「部份」責任。

4. 中等內聚力－由輕量級的類別負責概念相關、不同範圍內的某些責任，這些範圍中的責任並不是完全相關的。

假設有一個 *Company* 類別，它須要知道(a).它的職員與(b).它的財務資訊。這兩個責任範圍並非很相關，不過邏輯上它們都跟公司概念有關。此外，它所需的公開方法總數與程式碼也不會太多。

從經驗法則來看，有高內聚力的類別只會有數量相當少的方法，而且這些方法之間的功能性相關係也很高，也不會有太多工作。它會跟其它物件合作，共同完成比較大的工作。

具有高內聚力的類別之所以比較好是因爲它比較容易維護、了解與再使用。類別中的操作功能性相關程度越高，操作數目就越少，也越容易維護或增強功能。具有範圍切割良好、高度相關的功能性可以增加再使用機會。

高內聚力樣式跟物件技術中的許多事物一樣，都可以在真實世界找到相比擬的東西。我們常常可以發現：如果某人承擔太多不相關責任－特別是一些該委派給其他人的責任－那麼這個人的效率一定很差。這種現象經常發生在一些不懂得授權的經理人身上。這些人深受低內聚力之苦，他們真的是身心俱疲。

另一個經典設計原則：模組化設計

耦合力與內聚力都是軟體設計中存在很久的設計原則，用物件設計系統並不代表我們應該忘記一些良好基礎。另一個跟耦合力與內聚力關連性很強的設計原則是：模組化設計，這裡引述 Booch 的話：

> 模組化代表系統可以被分解成一些具有高內聚力、低耦合力的模組【Booch94】。

我們鼓勵模組化設計方式，讓設計出來的方法與類別具有高內聚力。在物件這個層級，模組化代表我們把目的清楚、唯一目的、考量因素有相關性的方法設計到

同樣類別中。

內聚力與耦合力；陰與陽

內聚力很差通常也會造成耦合力很差，反之亦然。我把內聚力與耦合力之間的關係稱爲軟體工程中的陰與陽，它們會彼此影響對方。例如，假如有一個 GUI 視窗小元件類別除了代表並畫出 GUI Widget 之外，還會把資料儲存到資料庫中、呼叫遠端物件服務等等。這樣的類別不只缺乏內聚力，也會跟許多不同元素之間產生耦合力。

例外情形

在某些少數情況下，我們也可以接受低內聚力。

其中一個情況是：爲了讓某人單純維護一組責任或程式碼，而把它們放到單一類別或元件中，這種群組方式可能會造成維護困難。不過，我們假設在一個應用程式中，因爲其它設計原則的緣故，內崁式 SQL 述句會分布在十個類別中，例如十個「資料庫對應者」類別。雖然專案中可能會有數十位物件導向程式設計師，你可能只有一兩個 SQL 專家知道如何好好定義並維護 SQL，大部份程式設計師並沒有很好的 SQL 技能。甚至 SQL 專家有可能不是合適的物件導向程式設計師。在這個情況下，軟體架構設計師可能決定把所有 SQL 述句放到同一個 *RDBOperations* 類別，方便 SQL 專家在同一個地方處理 SQL。

另一個元件有低內聚力的情況是分散式系統中的伺服端物件。因爲遠端物件與遠端之間的通訊有經常性與效能上的考量因素，有時候我們比較傾向於產生少一點、大一點、內聚力比較低的伺服端物件，由它爲多個操作提供單一介面。這種做法稱爲**粗部遠端介面**（coarse-grained remote interface）樣式。在這種樣式中，遠端操作做的事比較有整體性，方便在一次遠端操作呼叫（remote operation call）中做比較多的事，這樣才不會因爲在網路上做太多次遠端呼叫而影響到效能。舉一個簡單的例子，原本遠端物件需要三個細部操作（fine-grained operation）：*setName* 、 *setSalary* 與 *setHireDate* ，現在則用一個遠端操作 *setData* 代替，一次傳整組資料過去，結果會有比較少的遠端呼叫、比較好的效能。

優點

- 提高設計的清晰度與可理解性。
- 簡化維護與加強功能的開發工作。
- 通常也會具有低耦合力。
- 一小撮高度相關的功能性可以增加再使用，因爲我們可以在非常特殊的情況

下使用內聚力高的類別。

第十節控制者樣式

解決方案

把接收或處理系統事件訊息的責任指派到代表下面概念的類別上：

- 代表整個系統、裝置或子系統（表層介面型控制者。）
- 代表系統事件所屬的使用案例情節，我們通常把這樣的類別命名爲 （使用案例名稱）Coordinator 或者 （使用案例名稱）Session （使用案例型控制者或者 Session 型控制者。）
  - 對某個使用案例情節中的所有系統事件用同樣的控制者。
  - 非正式定義：Session 是（系統）跟參與者之間的對話實例（a instance of a conversation with an actor。） Session 的時間長度不定，不過通常都是根據使用案例決定 Session 範圍的。

概念延伸：請注意「視窗」、「applet」、「視窗小元件」、「view」與「document」類別都不是可能的控制者。這些類別不應該完成系統事件所賦予的責任，它們主要目的在接收事件，再委託給控制者做。

問題

誰應該負責處理系統輸入事件（input system event）？

輸入時的**系統事件**（system event）代表外部參與者所產生的事件。它們通常跟**系統操作**（system operation）有關－這種操作負責回應系統事件，就像訊息與方法之間的關係一樣。

舉例來說，當收銀員按下 POS 終端機上「完成銷售」按鈕時，就產生代表「這筆銷售已經完成」的系統事件。同樣地，當作家按下文字處理機上的「拼字檢查」按鈕時，就產生代表「執行拼字檢查」的系統事件。

控制者代表接收或處理系統事件的非使用者介面物件。控制者定義系統操作的方法。（譯註：方法是操作的本體）

範例

在 NextGen 應用程式中，會存在許多系統操作，如圖 16.13 所示。圖中把系統本身當成一個類別或元件（在 UML 中這種做法是合法的。）

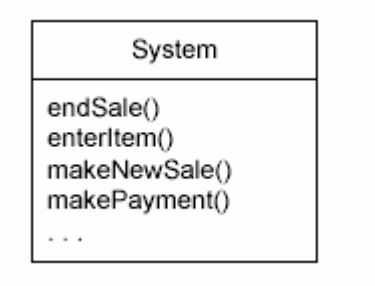


圖 16.13 跟系統事件有關的系統操作。

在分析期間，我們可能會把系統操作指定給 *System* 類別，以表示它們是系統操作。然而，這不代表在設計期間，我們會有一個 *System* 軟體類別負責完成這些系統操作。在設計期間，我們會把系統操作指定給控制者類別（譯註：可能不只一個控制者類別。）請參見圖 16.14。

誰應該是 *enterItem* 與 *endSale* 兩個系統事件的控制者呢？

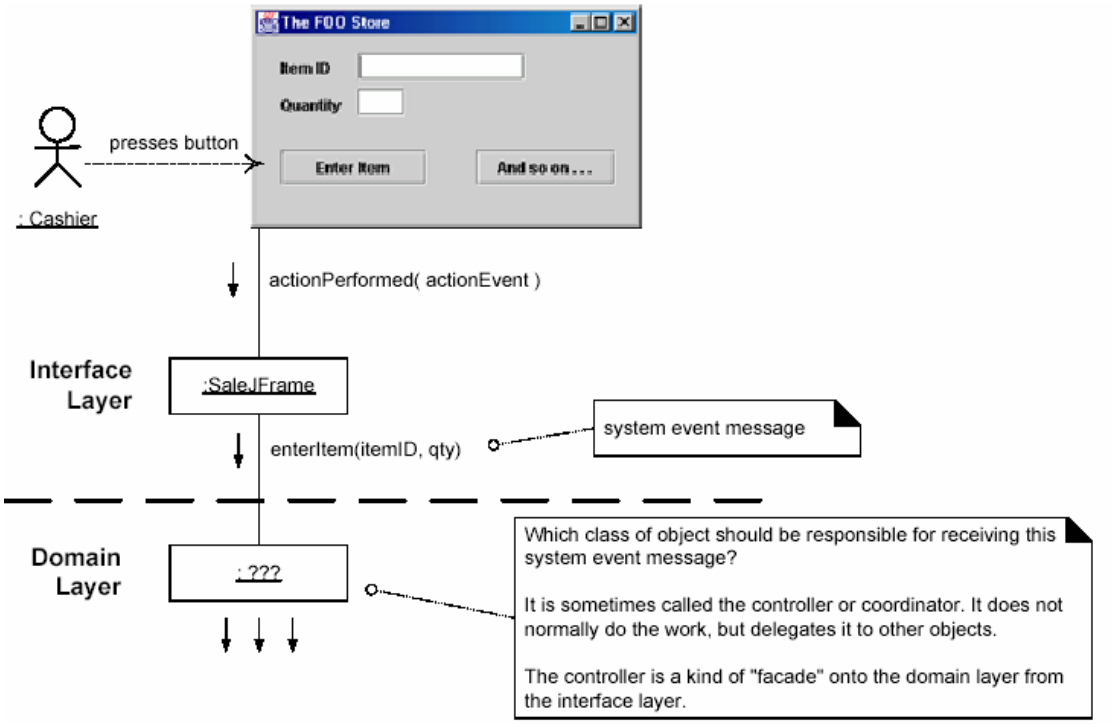


圖 16.14 誰是 enterIem 的控制者？

根據控制者樣式，我們可能有下面幾種選擇：

代表整個「系統」、「裝置」或者「子系統」	*Register*、*POSSystem*
在一個使用案例情節中，代表所有系統事件的接收者或處理者	*ProcessSaleHandler*、*ProcessSaleSession*

用圖 16.15 中的互動圖來看就很清楚。

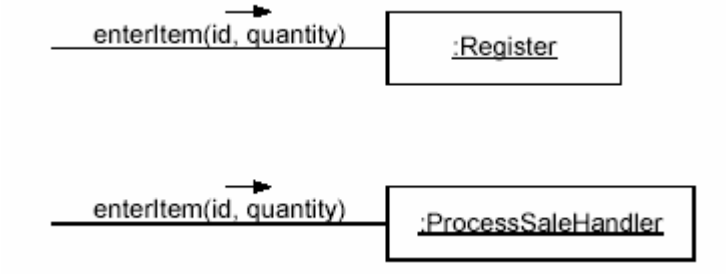


圖 16.15 可能選擇的控制者。

在這些可能的類別中，哪個類別是最合適的控制者可能會受到後面章節所討論的考量因素影響。

我們會把分析時期找到的系統操作指派到一個或數個設計時期的控制者類別上，例如 *Register*，如圖 16.16 所示。

討論

系統負責接收外部輸入事件，人們通常利用 GUI 操作系統。其它輸入媒介還包括一些外部訊息，例如負責處理電話的通訊交換機或處理控制系統中的感應器訊號。

不論是哪種情況，如果我們採用物件設計方式，就必須選擇一些事件處理者。控制者樣式提供選擇事件處理者時的指引，幫助我們做出合適的選擇。在圖 16.14 中，控制者是從領域層到介面層的一種表層介面類別。

我們通常用單一控制者類別處理某個使用案例中的所有系統事件，也有可能會在這個控制者中維護跟使用案例有關的狀態資訊，這樣的資訊可以幫助我們找出不合法的系統事件順序（例如在 *endSale* 操作之前發生 *makePayment* 操作。）不同的使用案例可能會有不同控制者。

一般來說，控制者應該把需要完成的工作*委託*（delegate）給其它物件做，它則負責協調並控制整個活動，不要自己做太多事情。

請參閱後面針對詳述階段所做的「議題與解決方案。」

第一種控制者是一種表層介面型控制者，它可能代表整個系統、裝置或子系統。概念是選擇一些類別名稱，這些名稱具有位於應用程式中其它分層結構上面、表面或表層介面的含意，而且它代表從使用者介面層到其它層的主要系統服務點。它可能是整個實體單位的抽象概念，例如 *Register*【註】、*TelecommSwitch*、*Phone* 或者 *Robot* 等等；也可能是代表整個軟體系統的類別，例如 *POSSystem*，或者設計師所選代表整個系統或子系統的其它概念，例如代表遊戲軟體的 *ChessGame* 。

【註】：實體 POS 裝置有許多不同術語稱呼，其中包含收銀機、銷售點終端機（POST）等等。隨著時間過去，「收銀機」既代表實體裝置，也代表收銀機銷售與付款的邏輯抽象概念。

如果沒有「太多」系統事件的話，或者不可能把系統事件訊息從使用者介面轉送到替代性控制者（例如在訊息處理系統中），此時可以用表層介面型控制者處理所有系統事件。

如果我們選擇採用使用案例型控制者，那麼每個使用案例都會有不同控制者存在。請注意控制者物件並不是領域物件，它是支援系統用的人造產物（在 GRASP 樣式中，我們把這種做法稱爲純虛構物樣式。）舉例來說，如果 NextGen 應用程式中有*處理銷售*與*處理退貨*兩個使用案例，那麼可能會有像 *ProcessSaleHandler*、 *HandleReturnsHandler* 類別存在。

什麼時候我們應該採用使用案例型控制者呢？我們可以想想看把責任放在表層介面型控制者中是不是會讓設計結果具有低內聚力或高耦合力。一般來說，表層介面型控制者會因爲擔負過多的責任而變得臃腫不堪。如果不同處理流程中有許多系統事件存在，這時候採用使用案例型控制者是一種好的選擇，這種做法讓責任分散到不同類別中，類別的責任才不會失去控制。此外，我們也能夠知道或理解目前情節的狀態。

在 UP 與 Jacoboson 舊的 Objectory 開發方法【Jacobson92】中，有提到一個（選擇性的）概念：邊界類別（boundary class）、控制類別（control class）與實體類別（entity class）。**邊界物件**（boundary object）代表介面的抽象概念，**實體物件**（entity objects）代表跟應用程式無關的（而且通常是永續儲存的）領域軟體物件，而且**控制物件**（control object）則代表控制者樣式中的使用案例處理者。

控制者樣式中最重要的概念延伸是介面物件（例如視窗物件或視窗小元件【widget】）或展現層（presentation layer）不應該負責系統事件工作。換言之，系統操作應該由應用程式邏輯層（或領域層）中的物件處理，而不是由系統介面層處理。

控制者物件通常是客戶端物件（例如具有 Java Swing GUI 的應用程式），它會在某個流程中擔任使用者介面角色。當使用者介面是瀏覽器上的網頁客戶端時，控制者物件的功能可能部分會在伺服端軟體。後面處理系統事件的例子中會用到許多常見樣式，這些樣式會受到所選擇的伺服端、技術性框架（例如 Java servlets）影響。不論如何，大家常用一種實作樣式（idiom）：產生伺服端的使用案例型控制者，這個控制者有可能是某個使用案例的 servlet 或者某個使用案例的 session bean（它是一種企業 JavaBeans【Enterprise JavaBeans，EJB】。）這個伺服端的 session 物件代表系統跟外界參與者互動的某一段「期間。」

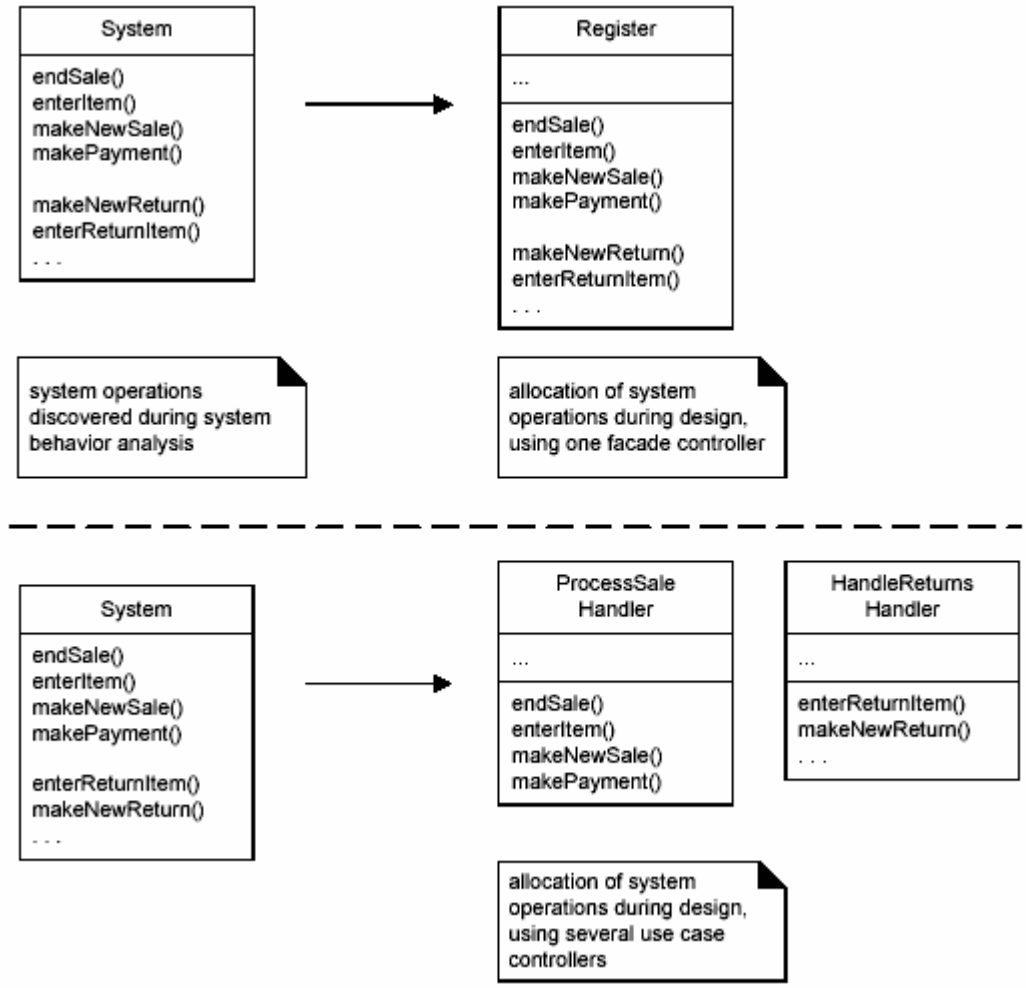


圖 16.16 系統操作的責任分配。

如果使用者介面層不是網頁伺服端（例如它是 Swing 或 Windows GUI），而且應用程式會呼叫遠端服務，控制者樣式還是會常用的樣式。使用者介面負責把請求傳給本地客戶端的控制者，然後這個控制者可能會再把全部或部分請求的處理責任交給遠端服務做。這種設計方式會降低使用者介面跟遠端服務之間的耦合力，而且我們可以很容易藉由這個間接的客戶端控制者，在本地端或遠端提供服務。

總而言之，控制者負責接收從使用者介面層傳來的服務請求，並且協調如何完成這些請求，而且通常是靠委派責任給其它物件達成的。

優點

- *增加再使用與可插入式介面（pluggable interface）的可能性*－如果採用這種做法，我們可以確定應用邏輯不會在介面層處理。從技術層面來看，控制者被賦予的責任是可以由介面物件處理的，不過這種設計方式隱含完成應用邏輯所需的相關程式碼或邏輯被分散在介面或視窗物件上。把介面當成控制者（interface-as-controller）的設計方式會降低未來應用程式中重新使用這些邏

輯的機會，因爲邏輯跟特定介面（例如視窗型物件）緊緊綁在一起，而其它應用程式很難使用這些介面。另一方面，把系統操作責任委派給控制者可以增加未來應用程式中重新使用邏輯的可能性。而且因爲應用邏輯沒有跟介面層綁在一起，我們還可以更換介面。

- *存放使用案例狀態的合理位置*－有時候我們需要確認系統操作是否按照合理順序發生或者理解某個使用案例中的活動現狀與操作。舉例來說，我們應該保證除非發生 *endSale* 操作，否則不會發生 *makePayment* 操作。在這種情況下，我們需要把狀態資訊放在某些地方，控制者就是合理的選擇，特別是整個使用案例中共用一個控制者的情況（我們特別推薦這種情形。）

一些議題與相關解決方案

肥胖的控制者

如果缺乏良好設計，控制者類別的內聚力可能很低－缺乏焦點，而且會處理太多責任範圍；這種類別稱爲**肥胖的控制者**（bloated controller）。肥胖徵兆包含：

- 系統中只有*唯一*控制者類別，它負責接收所有系統事件，而且系統中存在許多系統事件，這種情形通常發生在表層介面型控制者。
- 控制者自行完成系統事件所需的許多工作，沒有把這些工作委託給其它類別做。這種情形通常會違反資訊專家樣式與高內聚力樣式。
- 控制者裡面有許多屬性存在，裡面保有系統或領域中的重要資訊，沒有把資訊分散到其它物件身上，或者有可能系統中到處都是重複資訊。

肥胖的控制者問題，有下面幾種治療方法：

1. 增加更多的控制者－系統中不應該只存在一個控制者。請使用使用案例型控制者，而不要使用表層介面型控制者。舉例來說，在航空訂位系統中，應用程式可能會有許多系統事件。爲了處理這些系統事件，我們可能會有下面的控制者：

使用案例型控制者
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

2. 設計控制者時，每個系統操作所需責任主要都委託給其它物件做。

不要在介面層處理系統事件

再重述一次：控制者樣式的一個重要推論就是介面物件（例如視窗物件）與介面層不應該負責處理系統事件。舉例來說，如果你是用 Java 設計系統，那麼用

JFrame 顯示訊息時， *JFrame* 不應該負責處理系統事件。

假設 NextGen 應用程式中有一個視窗負責顯示銷售資訊、捕捉收銀員操作。在使用控制者樣式的情形下，圖 16.17 展示 *JFrame* 、控制者以及（簡化的） POS 系統中部分物件之間的合理關係。

請注意 *SaleJFrame* 類別－介面層的部分－這裡把 *enterItem* 訊息傳給 *Register* 物件，而不自行處理處理操作或決定如何處理它，視窗物件僅僅負責委託訊息給其它層做。

假設我們使用控制者樣式，把系統操作的責任放在應用層或領域層物件上，而不把責任放在介面層，以增加重新使用類別的可能性。如果是由介面層物件（例如 *SaleJFrame* ）負責處理系統操作，而且這個系統操作代表一部分企業流程，那麼這種設計方式會把企業處理邏輯放在介面物件上（例如視窗物件），這種物件再使用的可能性很低，因爲它跟特定介面與應用程式有很高的耦合力。

結論是：圖 16.18 中的設計結果不是我們想要的。

把（處理）系統操作的責任放在領域物件的控制者上，就可以很容易在未來應用程式中重複使用這些支援企業流程的程式邏輯。這樣做法可以讓我們很容易拿掉原先介面層，使用不同介面框架或技術，甚至在離線、「批次」模式下執行系統。

訊息處理系統與命令樣式

有些應用程式是處理訊息（message-handling）的系統或伺服器，它們會接收由其它處理程序（process）來的請求。電信通訊交換機就是一個常見例子。在這樣的系統中，介面與控制者的設計方式會有點不同，其它章節會介紹詳細細節。簡單來說，常見的設計方式是利用命令樣式【GHJV95】與第 34 章介紹的命令處理者樣式【BMRSS96】。

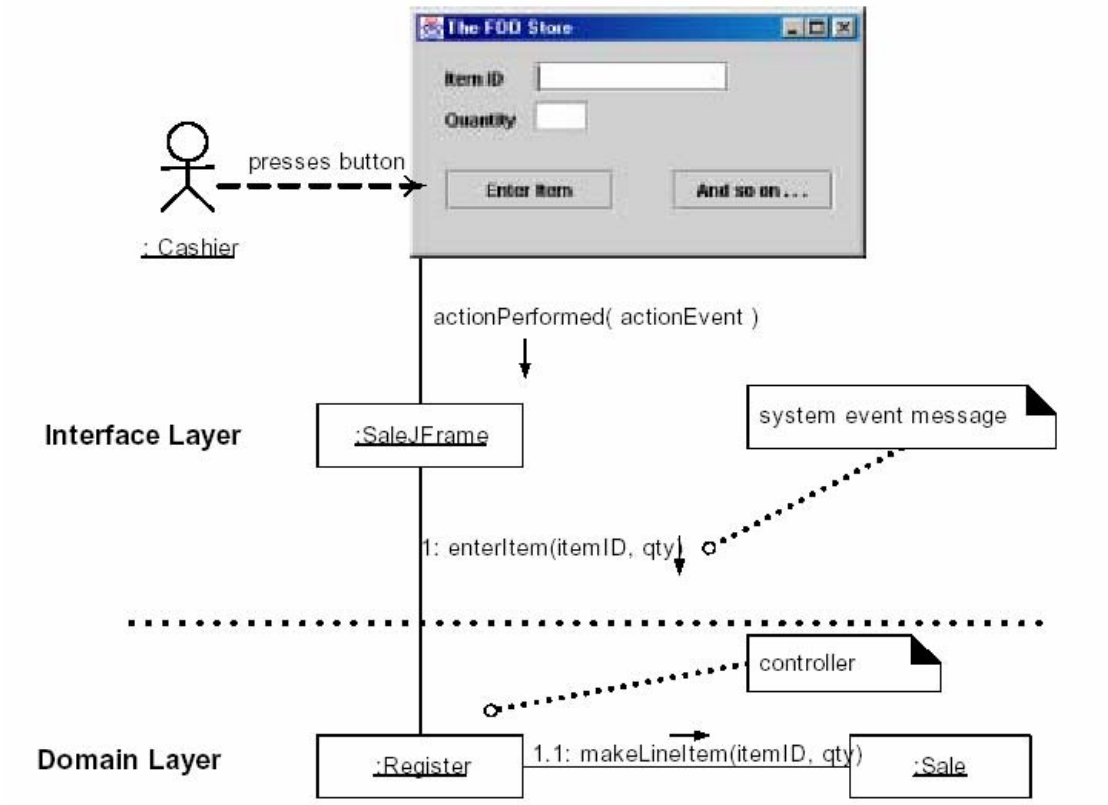


圖 16.17 我們想要的介面層與領域層之間的耦合方式。

相關樣式

- 命令樣式－在訊息處理系統中，我們可能用個別 Command 物件【GHJV95】代表並處理每個訊息。
- 表層介面樣式－表層介面型控制者就是一種表層介面樣式【GHJV95】。
- 分層樣式－這是一種 POSA 樣式【BMRSS96】。把領域邏輯放在領域層而不放在展現層是分層樣式概念的一部分。
- 純虛構物樣式－這是另一種 GRASP 樣式。純虛構物是由設計師任意產生的，軟體類別名稱不是從領域模型想到的。使用案例型控制者就是一種純虛構物樣式。

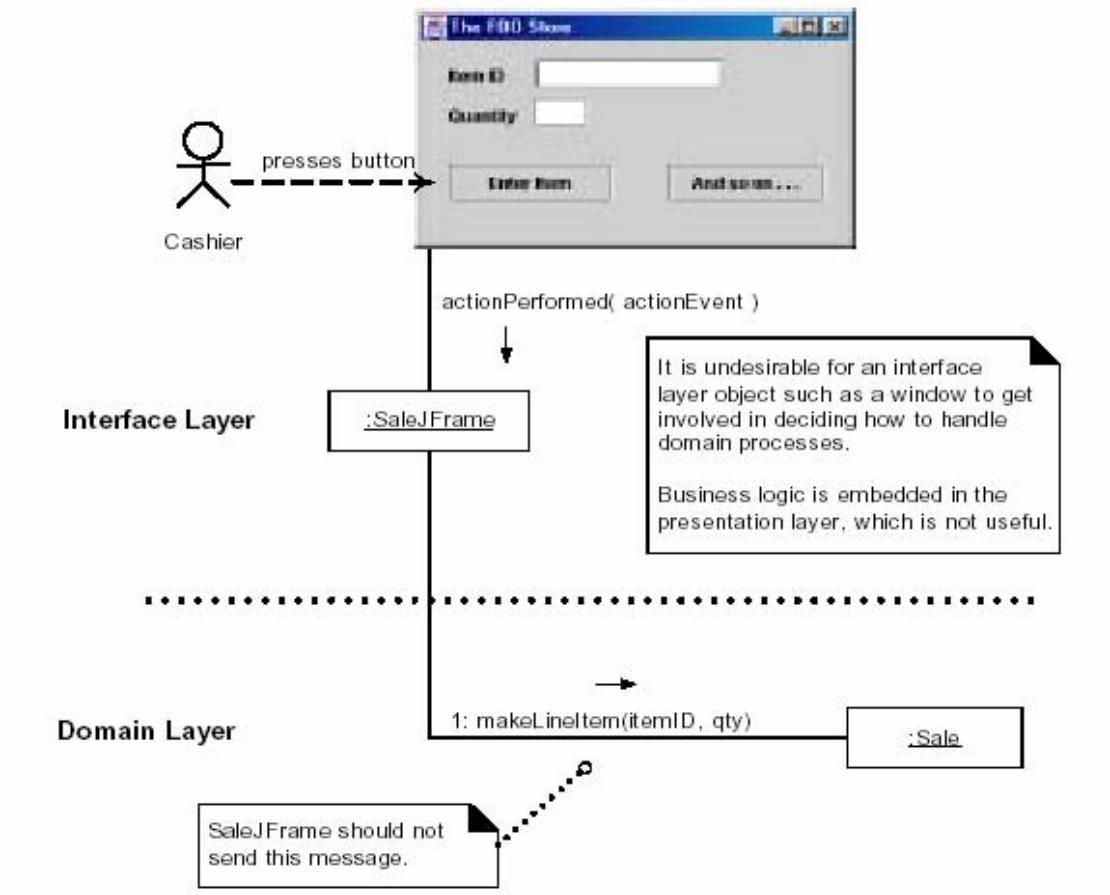


圖 16.18 在介面層與領域層之間，我們不想要的耦合力。

第十一節物件設計與 CRC 卡

雖然 UML 中沒有正式定義 CRC 卡（class-responsibility-collaborator cards）【BC89】，不過我們通常會建議大家用 CRC 卡指派責任，並且在卡中它跟其它物件之間的合作關係。這是由 Kent Beck 與 Ward Cunningham 率先提倡的概念。他們主要是鼓勵物件設計師用更抽象的指派責任與合作關係思考物件，也鼓勵大家使用樣式。

CRC 卡是一種索引卡片，每個類別用一張卡片，裡面簡單寫出類別的責任，並且說明合作物件清單以完成所賦予責任。這些卡片通常是在一小群人的會議中發展出來的。當我們用 CRC 卡考慮設計結果時，可以應用 GRASP 樣式。

CRC 卡是紀錄責任指派結果與合作關係的一種設計方式。這種記錄方式可以加強我們使用互動圖與類別圖。真實價值不在於卡片或圖上，而是指派責任時的一些考量因素。

第十二節進階讀物

負責式物件合作或**責任驅動設計**（responsibility driven design）的指標性使用者

中，最有影響力的是在波蘭 Tektronix 進行的 Smalltalk 物件相關工作，著名的人包括 Kent Beck 、 Ward Cunningham 、 Rebecca Wirfs-Brock 等等。*Designing Object-Oriented Software*【WWW90】是最具指標性的一本教科書，到今天來說，裡面的觀念還是相當有參考性。

其它兩本建議的教科書裡面則強調基本物件設計原則，分別是 Riel 寫的 *Object-Oriented Design Heuristics* 與 Coad 寫的 *Object Models*。

第十七章設計模型：用 GRASP 樣式完成使用案例實現

你需要有好的想像力才能發明東西，而且有可能會發明一大堆垃圾。
— Thomas Edison

本章目標

- 設計使用案例實現。
- 應用 GRASP 樣式以指派責任到類別身上。
- 利用 UML 的互動圖表示法展示物件設計。

簡介

本章探討如何用物件之間的責任合作產生設計結果。請特別注意如何應用 GRASP 樣式以開發出設計良好的解決方案。請注意像 GRASP 樣式這樣的東西或名稱不是最重要的；它們是一種學習工具，而且有助於我們交談、用方法論的方式進行物件設計。

本章以 NextGen POS 爲範例，利用這些設計原則說明物件導向設計師如何指派責任、建立物件之間的互動關係 — 這是物件導向開發過程中的核心技能。

請注意：

> 指派責任與設計（物件之間的）合作關係是設計開發工作中非常重要、富有創造力的開發步驟，不論你是畫圖或寫程式都一樣。

本章內容會介紹的很詳細；裡面嘗試詳細展現出物件設計中的決策 — 責任的指派與物件互動的選擇 — 不是「一種魔術」或不合理的推理過程，我們可以用合理方式解釋並學習它們。

第一節使用案例實現

引述：「使用案例實現裡面描述如何用設計模型中的物件互動情形實現特定使用案例」【RUP】。更精確一點，系統設計師將描述某個使用案例中一個或多個情節的設計結果；針對每個設計結果所做的描述都可稱爲使用案例實現。使用案例實現是 UP 中的術語或概念，我們用它連結以使用案例表達出來的需求與滿足需求的物件設計。

我們通常會用 UML 的互動圖表達使用案例實現。而且就像我們在前一章所探討的一樣，有些物件設計的原則與樣式（例如資訊專家樣式與低耦合力）可以應用

在設計開發工作中。

請參見圖 17.20 （在本章結尾處），裡面展示 UP 中一些工作成果之間的關係：

- 使用案例裡面可以找到可能的系統事件，我們在系統循序圖中明確秀出系統事件。
- 以領域物件的變化情形說明系統事件的詳細效果，我們可以依需要把它們寫在系統操作合約上。
- 系統事件代表初始互動圖的訊息，我們用互動圖展示物件之間是如何互動以滿足需求—使用案例實現。
- 互動圖裡面描述軟體物件之間訊息互動的情形，有時候這些軟體物件的名稱是從領域模型中的概念性類別名稱得到靈感的，此外軟體物件中也有可能包含其它物件的類別。

第二節使用案例實現相關工作成果說明

互動圖與使用案例實現

在目前反覆中，我們考慮下列的情節與系統事件，例如：

- *處理銷售*：*makeNewSale*、*enterItem*、*endSale*、*makePayment*

如果我們用合作圖展示使用案例圖，那麼我們會用不同的合作圖圖展現每個系統事件訊息的處理方式。例如（圖 17.1）：

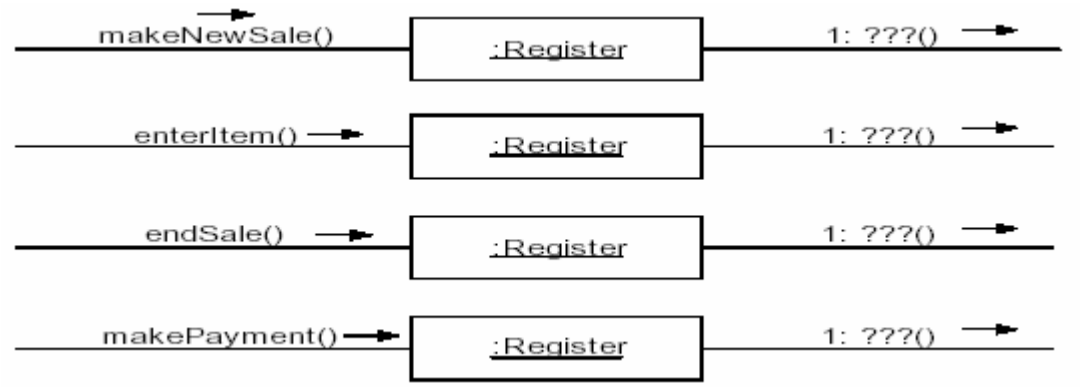


圖 17.1 合作圖與系統事件訊息的處理方式。

另一方面，如果用循序圖的話，我們就能夠把所有系統事件訊息放在同一張圖中，如圖 17.2 所示。

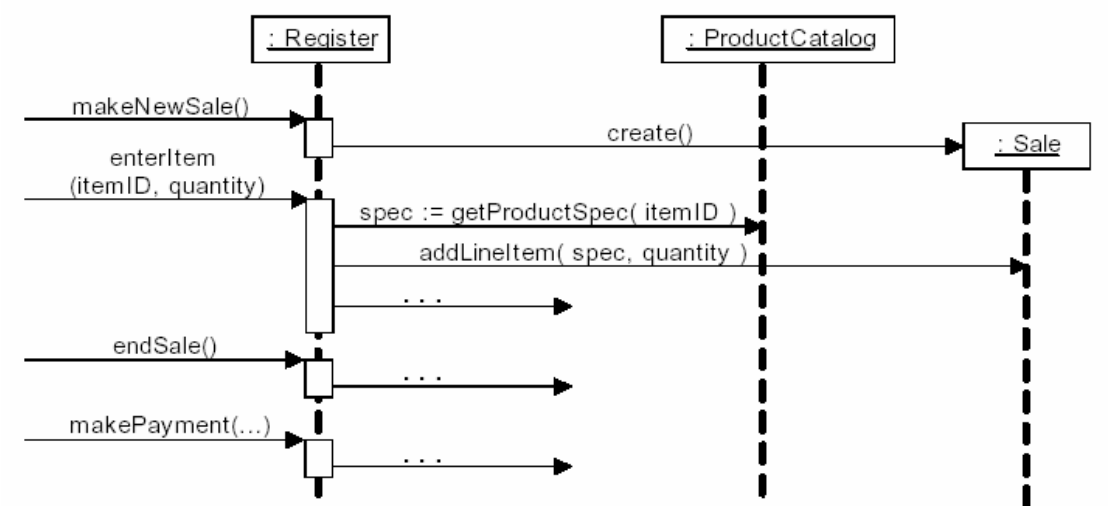


圖 17.2 循序圖與系統事件訊息的處理方式

然而，如果只用一張循序圖，這樣的循序圖太複雜也太長了。對互動圖來說，我們可以針對每個系統事件訊息畫出一張循序圖，請參見圖 17.3。

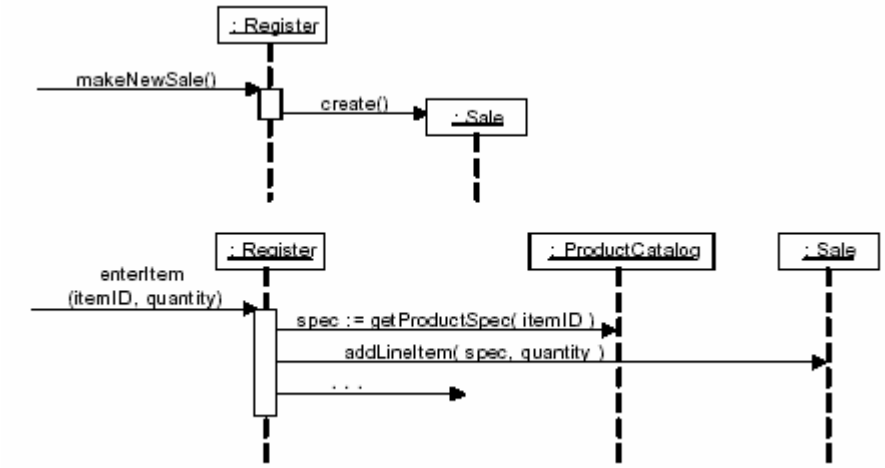


圖 17.3 幾張循序圖與系統事件訊息處理方式。

合約與使用案例實現

再重複一次，我們可以從使用案例的說明文字中直接設計出使用案例實現。此外，針對某些系統操作，我們也可以寫出合約，裡面會描述更多細節。例如：

合約 CO2: enterItem

操作：	enterItem(itemID : ItemID, quantity : integer)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—產生 SalesLineItem 的一個實例 sli（實例的產生。） —…

爲了跟使用案例的說明文字連結在一起，針對每個合約，我們會寫出事後條件以

說明狀態變化，並且設計滿足需求的訊息互動情形。舉例來說，我們用部分 *enterItem* 系統操作爲藍本，畫出圖 17.4 中的部分互動圖，裡面滿足產生 *SalesLineItem* 實例的狀態變化。

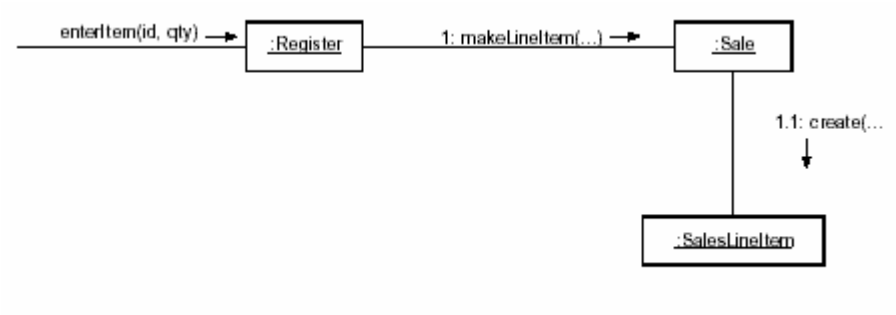


圖 17.4 部份互動圖。

警告：需求是不完美的

我們要時時牢記在心：之前所寫的使用案例與合約只是對想達成事物的一種猜測。軟體開發歷史告訴我們一項不變的發現就是：需求是不完美的，它會改變。這不是在爲沒有做好需求工作找藉口或忽略需求工作，而是體認到我們需要顧客與主題專家的持續參與，並且需要他們審查不斷增加的系統行爲、持續回饋。反覆式開發方式的其中一種優點是它很自然就支援我們在設計與實作開發工作中不斷發現的新分析與設計結果。反覆式開發方式的精神就是在需求分析時捕捉「合理的」資訊明細程度，然後在設計與實作時再補捉更多細節。

領域模型與使用案例實現

我們會在互動圖中畫出某些軟體物件之間訊息互動情形，其靈感來自於領域模型，例如 *Sale* 概念性類別與 *Sale* 設計類別。當我們用 GRASP 樣式選擇適當地方放置責任時，主要依靠領域模型中的資訊。如同前面所提過的一樣，現存的領域模型可能不完美；我們可以預期到裡面會有一些錯誤或遺漏。你可能會發現之前遺漏的新概念、捨去之前找到的概念。同樣情況也會發生在關聯與屬性上。

概念性類別與設計類別

回想一下，UP 中的領域模型所展現的不是軟體類別，不過我們可以用領域模型爲靈感，找出設計模型中的軟體類別名稱。當我們畫互動圖或寫程式替一些設計類別命名時可能會查看領域模型，這種設計方式可以降低軟體設計跟軟體相關真實領域概念之間的觀點呈現差距（請參見圖 17.5。）

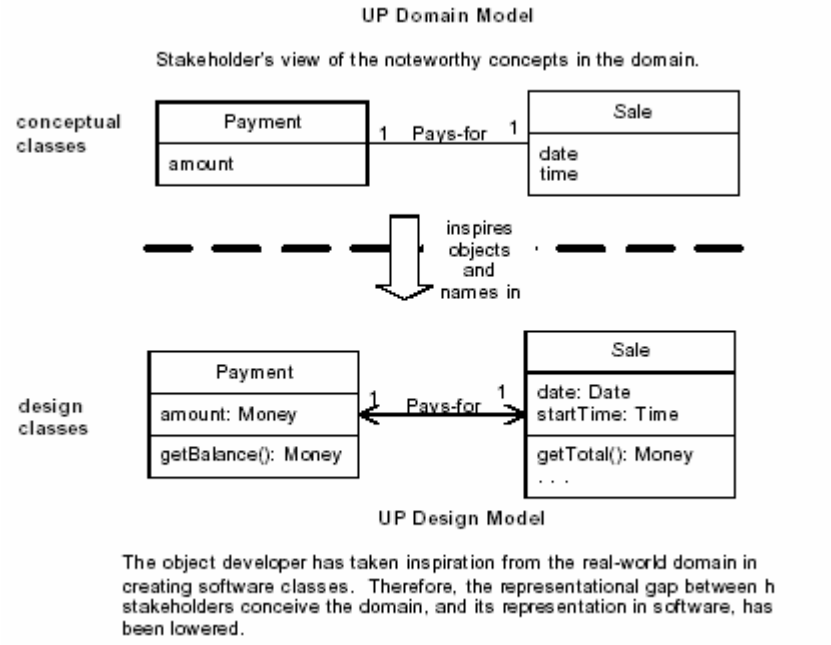


圖 17.5 用概念性類別替設計類別命名以降低觀點呈現差距。

那麼設計模型中的設計類別其名稱應該用領域模型中的類別名稱嗎？不一定；我們可能在設計開發工作中發現之前領域分析時遺漏的新概念性類別，而且也可能產生名稱或目的跟領域模型完全無關的軟體類別。

第三節Next Gen 系統中，針對這個反覆所做的使用案例實現

接下來幾個小節中會探索以 GRASP 樣式爲基礎、設計使用案例實現時可能會發生的選擇與決策。我們會詳細說明細節，嘗試展現設計良好的互動圖背後不會有「一隻看不見的手」；這些互動圖的產生是以合理的設計原則爲基礎。

就表示法來說，針對每個系統事件訊息所做的物件設計可以分別秀在不同圖中，展現不同設計議題的焦點。然而，我們也可以把它們群組在一張循序圖中。

第四節物件設計：makeNewSale

當顧客帶著要買的東西到櫃檯前面，收銀員就會要求開始進行一筆新銷售，這時候會發生 *makeNewSale* 系統操作。使用案例中的內容可能已經夠我們決定要做什麼事，不過在這個個案研究中，爲了解釋與完整性，我們已經寫出所有系統事件合約了。

合約 CO1: makeNewSale

操作：	makeNewSale()

交互參考：	使用案例：處理銷售
事先條件：	無
事後條件：	—產生一個 Sale 實例 s（實例的產生。） —把 s 跟 Register 關聯在一起（關聯的形成。） —把 s 的屬性初始化（屬性的修改。）

選擇控制者類別

我們面臨的第一個設計決策是要選擇這個系統操作訊息 *enterItem* 的控制者。根據控制者樣式，我們有下面幾種可能選擇：

代表整個「系統」、裝置或子系統　　*Register*、*POSSystem*

代表某個使用案例情節中所有系統事件 *ProcessSaleHandler*、*ProcessSaleSession* 的接收者或處理者

如果系統中只有幾個系統操作，而且表層介面型控制者（façade controller）不會負擔太多責任（換言之，如果它的責任不會沒有一致性），選擇像 *Register* 這樣的表層介面型控制者就可以了。當我們有許多系統操作，而且爲了區別責任、產生輕量級控制者類別、讓個別控制者類別有焦點存在（換言之，有一致性），這時候我們可以選擇使用案例型控制者（use case controller）。在這個例子中，*Register* 可以滿足我們的需要，因爲系統中只有一些系統操作而已。

> *Register* 代表設計模型中的軟體類別。它不是真正的實體收銀機，不過我們用這個軟體抽象概念名稱以降低領域概念與軟體之間的觀點呈現差距。

因此，圖 17,6 中的互動圖會先有送到 *Register* 軟體物件的 *makeNewSale* 訊息。

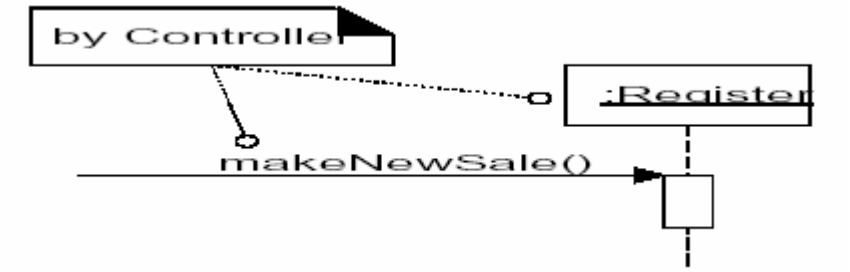


圖 17.6 應用 GRASP 中的控制者樣式。

產生一筆新銷售

現在我們必須產生一個軟體 *Sale* 物件，而且 GRASP 中的造物主樣式建議我們把產生類別的責任指派給聚合、包含或紀錄被產生物件的類別。

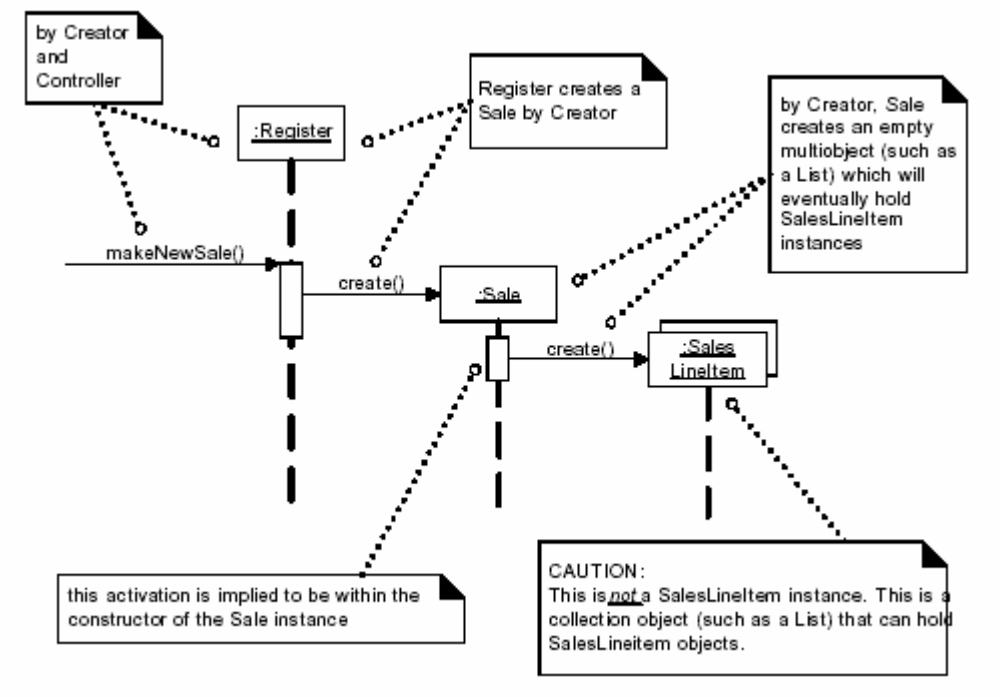


圖 17.7 產生 *Sale* 與 *SalesLineItem* 多重物件。

分析領域模型之後，可以發現 *Register* 也可視爲銷售記錄；事實上，多年以來在企業中，「收銀機」這個字已經代表紀錄會計交易（例如銷售）的東西。

因此，*Register* 是產生 *Sale* 的合理候選類別。而且如果由 *Register* 產生 *Sale* 的話，我們也可以把兩者關聯在一起，那麼在未來的交易流程操作中，*Register* 中就會有 *Sale* 實例的參考存在。

除了上述工作之外，當我們產生 *Sale* 時，也必須同時產生一個空的多重物件（容器，例如 Java 的 List）以紀錄所有未來要加入的 *SalesLineItem* 實例。*Sale* 裡面會包含並維護這個多重物件，根據造物主樣式，這一點也隱含 *Sale* 是產生這個多重物件的好候選類別。

因此，*Register* 將負責產生 *Sale*，而 *Sale* 則負責產生空的多重物件，請參見互動圖。

圖 17.7 的互動圖展示上述設計結果。

結論

這裡的設計並不難，不過重點是如何用控制者樣式與造物主樣式小心解釋設計結果，展現出設計細節是可根據設計原則與樣式（例如 GRASP）用很合理、方法論的方式決定的。

第五節物件設計：enterItem

當收銀員輸入想買東西的 *itemID* 與數量（選擇性的）時，會發生 *enterItem* 系統

操作。下面是完整的合約：

合約 CO2: enterItem

操作：	enterItem(itemID : ItemID, quantity : integer)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—產生 SalesLineItem 的一個實例 sli（實例的產生。） —把 sli 跟目前 Sale 關聯在一起（關聯的形成。） —把 sli.quantity 修改變成 quantity（屬性的修改。） —根據 itemID，把 sli 跟 ProductSpecification 關聯在一起（關聯的形成。）

我們會畫出滿足 *enterItem* 事後條件的互動圖，這時候可以用 GRASP 樣式幫我們做出設計決策。

選擇控制者類別

我們面臨的第一個決策跟 *enterItem* 系統操作的責任有關。根據控制者樣式，跟 *makeNewSale* 一樣，我們繼續把 *Register* 當作控制者。

顯示商品項目的說明文字與價格

根據**資料模型－顯示分離原則**（model-view separation），非 GUI 物件不應該跟輸出工作相關。因此，雖然在使用案例中有說到要在操作執行後顯示出商品的說明文字與價格，不過現在的設計結果中會先忽略這一點。

在這個例子中，跟顯示資訊有關的責任是所需資訊爲何，這些資訊現在都已經是已知的。

產生一筆新的 SaleLineItem

enterItem 的合約事後條件中有標示出 *SalesLineItem* 的產生、初始化動作與關聯。分析領域模型之後，我們知道 *Sale* 中會包含 *SalesLineItem* 物件。根據領域而來的靈感，我們的軟體 *Sale* 中也會包含軟體 *SaleLineItem*。因此，根據造物主樣式，軟體 *Sale* 是產生 *SalesLineItem* 的合適候選類別。

當我們把新產生的 *SalesLineItem* 實例儲存在 *Sale* 的商品項目多重物件中，會把 *Sale* 與 *SalesLineItem* 關聯在一起。事後條件中有說到當我們產生新的 *SalesLineItem* 時需要紀錄數量；因此，*Register* 必須傳送數量給 *Sale*，然後 *Sale* 再把數量當參數傳給 *create* 訊息（在 Java 中，我們實作時會用參數呼叫建構子。）因此，根據造物主樣式，我們會傳 *makeLineItem* 訊息給 *Sale*，由它負責產生 *SalesLineItem*。*Sale* 會產生 *SalesLineItem*，然後把新的實例傳給永續多重物件。

makeLineItem 訊息中會有一個 *quantity* 參數，所以 *SalesLineItem* 可以把它紀錄下來，類似的 *ProductSpecification* 也一樣，我們會紀錄跟 itemID 相符合的 *ProductSpecification*。

找到 ProductSpecification

SalesLineItem 中需要跟所輸入 itemID 相符合的 *ProductSpecification* 之間建立關聯。換言之，我們必須根據 *itemID* 讀取相符合的 *ProductSpecification*。
考慮*如何*達成查詢 *ProductSpecification* 目的之前，我們需要先考慮*誰*應該負責查詢 *ProductSpecification*。因此，第一步是：

開始指派責任之前先把責任陳述清楚。

我們重新把問題陳述爲：

知道跟 *itemID* 相符合的 *ProductSpecification* 爲何是誰該負的責任？

這個問題既不是產生實例的問題，也不是選擇系統事件控制者的問題。這裡是我們在設計過程中第一次應用資訊專家樣式的地方。
在許多情況下，資訊專家樣式是我們最先採用的設計原則。資訊專家樣式建議擁有滿足責任所需資訊的物件應該負責。那麼誰應該知道所有 *ProductSpecification* 呢？
分析領域模型之後，我們知道在邏輯上，*ProductCatalog* 包含所有 *ProductSpecification*。我們再一次從領域中尋找軟體的設計靈感，因此設計軟體類別時也用類似的組織方式：由軟體 *ProductCatalog* 包含所有軟體 *ProductSpecification*。
做了上述決定之後，接下來根據資訊專家樣式，*ProductCatalog* 是負責查詢 *ProductSpecification* 的很好候選類別，因爲它知道所有 *ProductSpecification* 物件。這個需求很可能會由一個叫做 *getSpecification*【註】的方法實作。
【註】：存取型方法的命名方式當然要遵從各種程式語言的慣例。Java 中總是用 object.getFoo()的形式、C++則比較傾向於 object.foo()，而 C#則用 object.Foo（如果這個方法是存取公開屬性的方法呼叫或直接存取方法，就會像 Eiffel 與 Ada 一樣，把方法隱藏起來。）這裡的例子是用 Java 風格命名的。

對 ProductCatalog 的可見性

誰應該送 *getSpecification* 訊息給 *ProductCatalog*，跟它要求 *ProductSpecification* 呢？
我們可以很合理地假設：在初始的*啓動*使用案例中會產生 *Register* 與 *ProductCatalog* 的實例，而且在 *Register* 物件與 *ProductCatalog* 物件之間存在永久連結。有了上述假設之後（我們必須把這些事情紀錄成清單，以確定當我們開始設計初始化動作時都有設計到它們），可能會由 *Register* 負責送 *getSpecification*

訊息給 *ProductCatalog*。

這裡隱含物件設計中的另一個概念：可見性（visibility）。**可見性**代表一個物件可以「看到」或參考另一個物件的能力。

如果一個物件要送訊息給另一個物件，那麼它必須對後者有可見性（譯註：這句話只有 90 % 正確，如果系統裡面有傳送訊息機制存在就不需要有可見性。）

因爲我們將假設 *Register* 對 *ProductCatalog* 有永久性的連結 — 或參考，前者應該對後者有可見性，因此可以傳訊息給後者，例如 *getSpecification* 訊息。

後面的章節對可見性問題有更深入的探討。

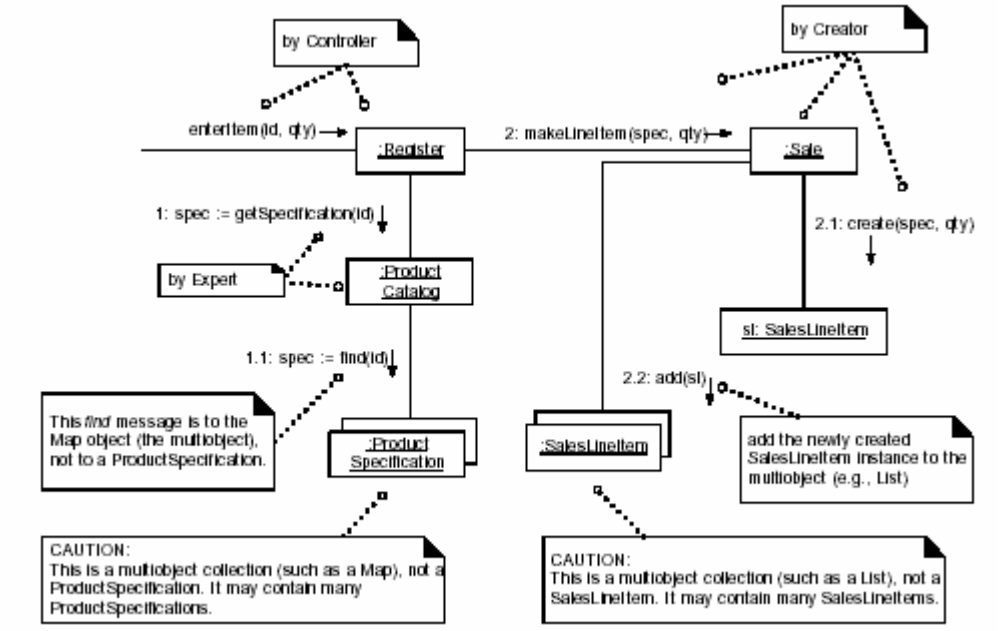


圖 17.8 enterItem 的互動圖。

從資料庫讀取 ProductSpecification

在最後一版的 Next Gen POS 應用程式中，不希望所有的 *ProductSpecification* 都存在於實際記憶體中。它們最有可能儲存於關連式資料庫或物件式資料庫中，然後根據需要讀取部分 *ProductSpecification*；有些 *ProductSpecification* 可能因爲效能或容錯原因而存在於客戶端處理程序中的快取。然而，我們先不考量存取資料庫的相關議題以簡化設計。現在先假設所有 *ProductSpecification* 都存在於記憶體中。

第 34 章會探討永續物件存取資料庫的主題，這是影響選用技術的大主題，可能採用的技術包括 J2EE、.NET 等等。

enterItem 的物件設計

經過上述討論之後，圖 17.8 中的互動圖反映出上述決策，這些決策都是跟責任

的指派與物件之間如何互動的相關決策。請注意，這樣的設計結果是以 GRASP 樣式爲基礎，經過深思熟慮得到的；物件互動的設計與責任的指派都需要經過深思熟慮。

傳到多重物件的訊息

請注意，傳到多重物件的訊息，在 UML 中是解釋成傳給多重物件本身的訊息，而不是傳給多重物件內成員的隱含式廣播行爲。對通用多重物件操作（例如 *find* 與 *add*）來說，在這樣的解釋方式下意義特別清楚。

例如在 *enterItem* 互動圖中：

- 送給 *ProductSpecification* 多重物件的 *find* 訊息會送給由多重物件（例如 Java Map）所代表的多重資料結構（collection data structure）一次。
  - 這個跟程式語言無關、通用的 *find* 訊息將在程式設計中轉換成特定程式語言與函式庫。它或許可能是 Java 中的 *Map.get*。我們可以在圖中用訊息 *get*；也可以用訊息 *find* 以說明這個設計圖可能需要再對應到不同的程式語言與函式庫。
- 送給 *SalesLineItem* 多重物件的 *add* 訊息會在多重物件（例如 Java List）所代表的多重資料結構中加入一個元素。

第六節物件設計：endSale

當收銀員按下按鈕表示銷售已經完成時，就會發生 *endSale* 系統操作。下面是這個系統操作的合約：

合約 CO3: endSale

操作：	endSale()
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—把 Sale.isComplete 的值設成 true（屬性的修改。）

選擇控制者類別

我們面臨的第一個決策是處理系統操作訊息 *endSale* 的責任。根據 GRASP 樣式中的控制者樣式，跟 *enterItem* 一樣，我們繼續用 *Register* 作爲控制者。

設定 Sale.isComplete 屬性

合約中的事後條件中說明：

■ 把 *Sale.isComplete* 設定成 *true*（屬性的修改。）

跟以前一樣，除非遇到控制者問題或產生實例問題（這個例子不是），否則我們都先考慮資訊專家樣式。

誰應該負責把 *Sale* 的 *isComplete* 屬性設定成 *true*？

根據資訊專家樣式，應該由 *Sale* 自己負責，因爲它擁有並維護 *isComplete* 屬性。因此，*Register* 將傳 *becomeComplete* 訊息給 *Sale*，讓它把 *isComplete* 屬性設定成 *true*。

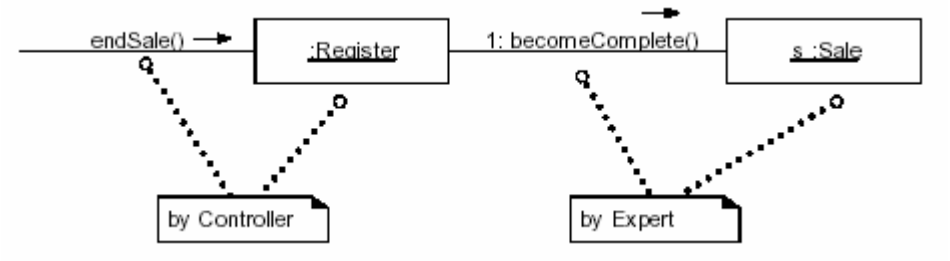


圖 17.9 完成商品項目的輸入工作。

用 UML 表示法秀出限制、註解與演算法

圖 17.9 中有秀出 *becomeComplete* 訊息，不過沒有繼續說明 *becomeComplete* 方法裡面所發生的細節（雖然一般會認爲這個例子不是很重要。）有時候在 UML 裡面，我們會用使用案例說明文字描述方法的演算法或說明一些合約明細。

爲了上述需要，UML 中提供**限制**（contraint）與**註解**（note）。UML 的限制是語意上有意義的一些資訊，我們會把限制附加在模型組成元素上。UML 限制是用{}括起來的文字；例如{x）20}。我們可以在限制中放入任何資訊或使用正式語言，而且如果有人想這麼做的話，UML 裡面還特別提供了 **OCL**（物件限制語言 object contraint language）。

UML 的註解是用來說明一些非語意上的影響，例如日期的產生或作者。

註解總是秀在註解方塊圖形中（有狗耳朵的文字方塊圖形。）

我們可以用加上大括號的簡單文字代表限制，這種方式適合短的說明文字。然而，有長說明文字的限制可能還是需要放在「註解方塊圖形」中，這時候註解方塊圖形事實上是放限制而不是註解。

在圖 17.10 中我們用了兩種風格。請注意，在限制的簡單風格中（放在大括號而不是在方塊圖形中）只放一定會成立的說明文字（這是限制的邏輯典型意義。）另一方面，方塊圖形中的「限制」所寫的則是 Java 方法實作。在 UML 中，這兩種限制的展示風格都是合法的。

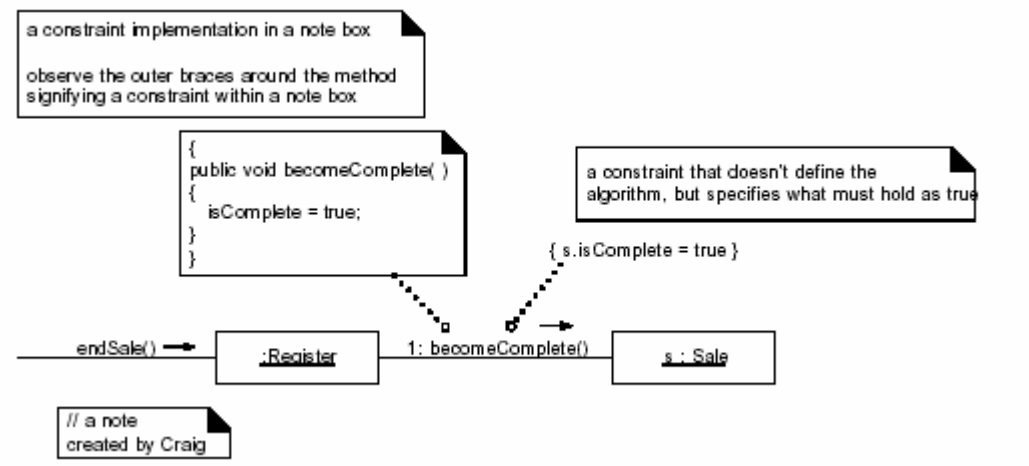


圖 17.10 限制與註解。

算出 Sale 總金額

*考慮處理銷售*使用案例中的片段：

主要成功情節（基本流程）：

1. **顧客**帶著要買的**商品**與／或**服務**到 **POS** 的**結帳櫃檯**前面。
2. **收銀員**啓動一筆新的**銷售**。
3. **收銀員**輸入**商品識別碼**。
4. 系統記錄**銷售明細**，並且顯示**商品說明文字**、**價格**與累計購買總金額。根據一組計價規則計算價格。

收銀員重複步驟 3-4 直到完成所有商品爲止。

5. 系統秀出包含**稅金**的總金額。

步驟 5 中會秀出總金額。根據資料模型－顯示分離原則，我們不會考量銷售總金額是如何顯示的，只確定我們是否知道總金額。請注意，現在並沒有設計類別知道銷售總金額，所以我們需要設計滿足這個需求的物件互動。

跟以前一樣，除非遇到控制者問題或產生實例問題（這個例子不是），否則我們都先考慮資訊專家樣式。

我們可以很明顯看出來 *Sale* 自己應該要知道總金額，不過爲了讓資訊專家樣式的推理過程更清楚一點 — 我們用下面例子做一些分析。

1. 陳述責任：
   - 誰應該知道銷售總金額？
2. 彙總所需資訊：
   - 銷售總金額是所有銷售商品細項的小計金額。
   - 銷售商品細項的小計金額:=商品細項的數量*產品價格
3. 列出滿足責任所需資訊與知道資訊的類別。

銷售總金額所需資訊	資訊專家樣式
ProductSpecification.price	*ProductSpecification*

SalesLineItem.quantity	*SalesLineItem*
在目前 Sale 中的所有 SalesLineItem	*Sale*

詳細的分析過程如下：

- 誰應該負責算出 *Sale* 總金額？根據資訊專家樣式，應該是由 *Sale* 自己負責，因爲它知道所有 *SalesLineItem* 實例，我們應該把它們的小計金額加總起來以算出銷售總金額。因此，*Sale* 應該知道它的總金額，用 *getTotal* 方法實作出來。
- 爲了讓 *Sale* 算出它的總金額，它需要知道每個 *SalesLineItem* 的小計金額。根據資訊專家樣式，應該是 *SalesLineItem* 自己要知道小計金額，因爲它知道數量與相關聯的 *ProductSpecification*。因此，*SalesLineItem* 有責任要知道它的小計金額，用 *getSubtotal* 方法實作出來。
- 爲了讓 *SalesLineItem* 算出小計金額，它需要知道 *ProductSpecification* 的價格。誰應該負責提供 *ProductSpecification* 的價格？根據資訊專家樣式，應該是 *ProductSpecification* 自己要知道價格，因爲它把價格封裝成屬性。因此，*ProductSpecification* 有責任要知道它的價格，用 *getPrice* 方法實作出來。

雖然上述所分析的例子不是很重要，而且在設計實務上並不會要求像這裡所呈現、讓人覺得有點困擾的明細程度，不過同樣的推理策略（應用資訊專家樣式）則可以用在比較難處理的情況。你將發現一旦學會這些設計原則之後，就能夠很快在心裡進行這樣的推理過程。

Sale—getTotal 的設計結果

經過上面討論之後，我們現在可能想要畫出一張互動圖，以展現出當我們送 *getTotal* 訊息給 *Sale* 時所發生的事情。圖中的第一個訊息是 *getTotal*，不過請注意 *getTotal* 訊息並不是系統事件。

我們因此可以得到下面的觀察結果：

並不是所有互動圖都是以系統事件訊息初始的；它們可以用系統設計師想要的任何訊息初始，秀出互動情形。

圖 17.11 就是我們畫出來的互動圖。首先，我們會先送 *getTotal* 訊息給 *Sale* 實例，然後 *Sale* 會送 *getSubtotal* 訊息給所有相關的 *SalesLineItem* 實例。最後 *SalesLineItem* 再送 *getPrice* 訊息給相關聯的 *ProductSpecification*。

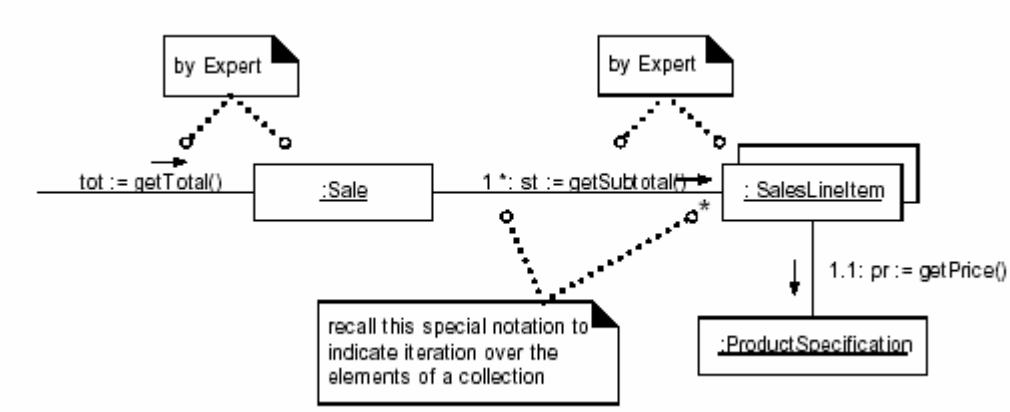


圖 17.11 Sale—getTotal 的互動圖。

因爲訊息上（通常）不會展示數學式，所以計算的細節可能用演算法或限制附加在定義計算結果的圖上。

誰應該送 *getTotal* 訊息給 *Sale* 呢？最可能的情況是使用者介面層的物件，例如 Java JFrame。

請注意圖 17.12 中用演算法註解與限制說明 *getTotal* 與 *getSubtotal* 的細節。

第七節物件設計：makePayment

當收銀員輸入現金付款金額時，就會發生 *makePayment* 系統操作。

合約 CO4: makePayment

操作：	makePayment(amount: Money)
交互參考：	使用案例：處理銷售
事先條件：	有一筆進行中的銷售。
事後條件：	—產生 Payment 的一個實例 p（實例的產生。） —p.amountTendered 的值設成 amount（屬性的修改。） —把 p 跟目前 Sale 關聯在一起（關聯的形成。） —把目前 Sale 跟 Store 關聯在一起（關聯的形成；把目前 Sale 加到以完成銷售的歷史記錄中。）

我們會設計出滿足 *makePayment* 事後條件的物件互動情形。

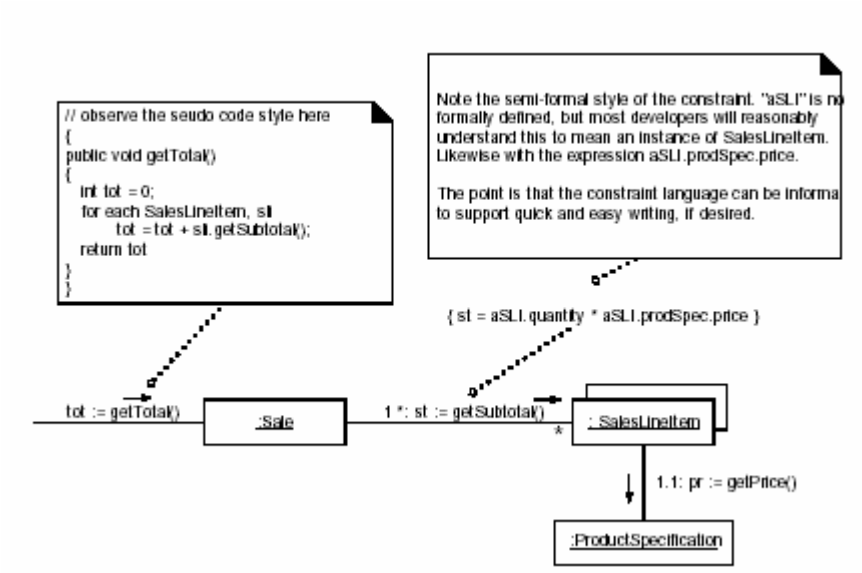


圖 17.12 演算法的註解與限制。

產生一筆 Payment

合約中的一個事後條件陳述：

- 產生 *Payment* 的一個實例 p（實例的產生。）

這是產生實例的責任，所以我們應該應用 GRASP 樣式中的造物主樣式。

誰會紀錄、聚合、最常使用或包含 *Payment* 呢？我們從某些敘述中可以知道：邏輯上是由 *Register* 負責紀錄 *Payment* 的，因爲在真實領域中，「收銀機」會紀錄會計資訊，所以我們把 *Register* 當成候選類別以達到降低軟體設計中觀點呈現差距的目的。此外，*Sale* 軟體也是很合理的候選類別，因爲它會經常使用 *Payment*。

另一種找到產生實例的方式是應用資訊專家樣式，找出誰是初始化資料的資訊專家 — 在這個例子中初始化資料是付款金額。*Register* 是收到系統操作 *makePayment* 訊息的控制者，所以 *Register* 一開始就會有這個付款金額。因此，*Register* 再度成爲候選類別。

彙總來說，我們有兩個候選類別：

- *Register*
- *Sale*

現在，我們面臨一個關鍵設計概念：

當我們有不同的設計選擇時，請進一步查看不同替代方案所隱含的內聚力與耦合力，以及替代方案面臨未來可能演變的壓力。選出內聚力與耦合力最好以及在未來最可能變動壓力下，最穩定的替代方案。

用高內聚力與低耦合力 GRASP 樣式考慮這些選擇所隱含的內聚力與耦合力。如果我們選擇由 *Sale* 產生 *Payment*，那麼 *Register* 的工作（或責任）就變得比較輕鬆 — 這樣可以得到比較簡單的 *Register* 定義。此外，*Register* 也不需要知道 *Payment* 實例的存在，因爲它是透過 *Sale* 紀錄 *Payment* 的 — 這樣可以降低

Register 的耦合力。設計結果如圖 17.13 所示。

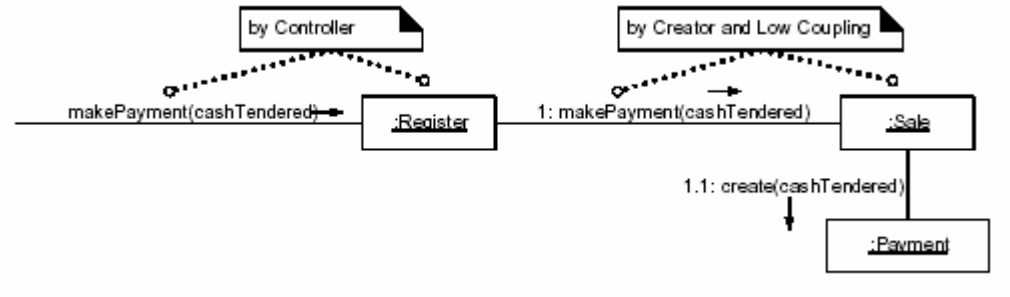


圖 17.13 Register—makePayment 的互動圖。

這個互動圖滿足合約中的事後條件：產生 *Payment*、把它跟 *Sale* 關聯起來，以及設定 *amountTendered* 的值。

紀錄一筆 Sale

一旦完成銷售之後，需求告訴我們應該把銷售放在歷史記錄中。跟以前一樣，除非遇到控制者問題或產生實例問題（這個例子不是），否則我們都先考慮資訊專家樣式。這裡的責任應該陳述爲：

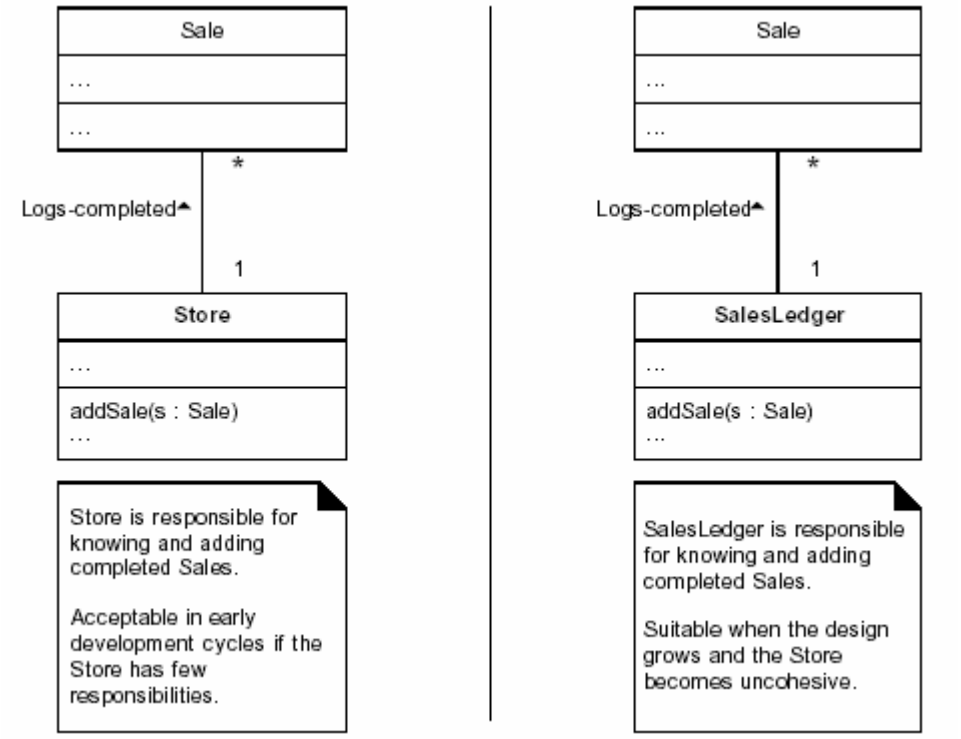


圖 17.14 誰應該負責知道所有紀錄起來的銷售，並且進行紀錄工作？

爲了降低軟體設計中觀點呈現差距（跟我們的領域概念相符合），而且銷售記錄也跟商店財務有關，所以 *Store* 是知道所有被紀錄銷售的合理候選類別。另一個替代方案是典型的會計概念，例如 *SalesLedger*。當設計量越來越大、*Store* 越來越不一致時，*SalesLedger* 就變得比較有意義（請參見圖 17.14。）

請注意，合約的事後條件中也有把 *Sale* 關連到 *Store*。如果我們要使用 *SalesLedger* 的話，事後條件所陳述的就不是我們真正設計的方式。或許之前我們沒有想過用 *SalesLedger*，不過我們現在可能會想這麼做，用它替代 *Store*。如果最後做出這樣的決定，那麼在理想狀況下，還要把 *SalesLedger* 加入領域模型中，因爲它也是真實世界領域中的概念名稱。在設計開發工作中出現這樣的發現與變動是可預期的。

在這個例子中，我們最後還是堅持維持原本計劃用的 *Store*（請參見圖 17.15。）

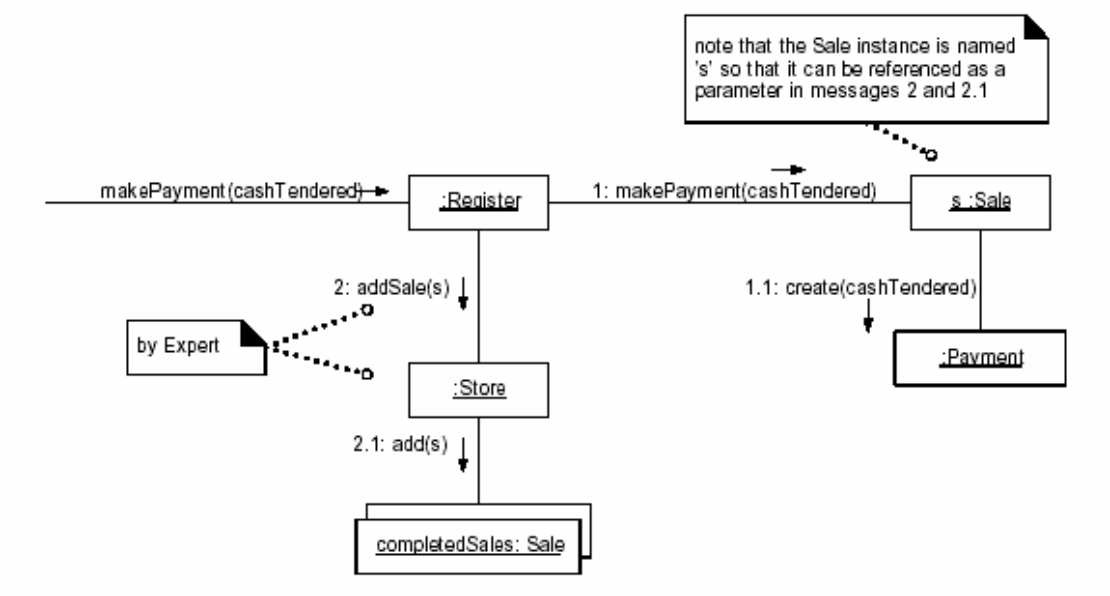


圖 17.15 紀錄已完成的銷售。

算出餘額

*處理銷售*使用案例中隱含收據與螢幕上要秀出、從付款得到的餘額。

根據資料模型－顯示分離原則，我們不關心餘額是如何顯示或列印出來的，不過我們需要確定是否知道餘額。請注意，現在沒有類別知道餘額，所以我們要設計滿足這個需求的物件互動情形。

跟以前一樣，除非遇到控制者問題或產生實例問題（這個例子不是），否則我們都先考慮資訊專家樣式。這裡的責任應該陳述爲：

誰應該負責知道餘額？

爲了算出餘額，我們需要知道銷售總金額與現金付款金額。因此，*Sale* 與 *Payment* 都是解決這個問題的部分專家。

如果主要是由 *Payment* 負責知道餘額的話，那麼它需要擁有對 *Sale* 的可見性，以便跟 *Sale* 詢問銷售總金額。因爲目前 *Payment* 看不見 *Sale*，所以這種做法會增加整體設計的耦合力—這樣一來就不符合低耦合力樣式。

另一方面，如果主要是由 *Sale* 負責知道餘額的話，那麼它需要擁有對 *Payment* 的可見性，以便跟 *Payment* 詢問付款金額。因爲 *Sale* 已經擁有對 *Payment* 的可

見性 — *Sale* 是 *Payment* 的造物主，所以這種做法不會增加整體設計的耦合力，因此這是我們比較想要的設計方式。

圖 17.16 中的互動圖提供餘額的解決方案。

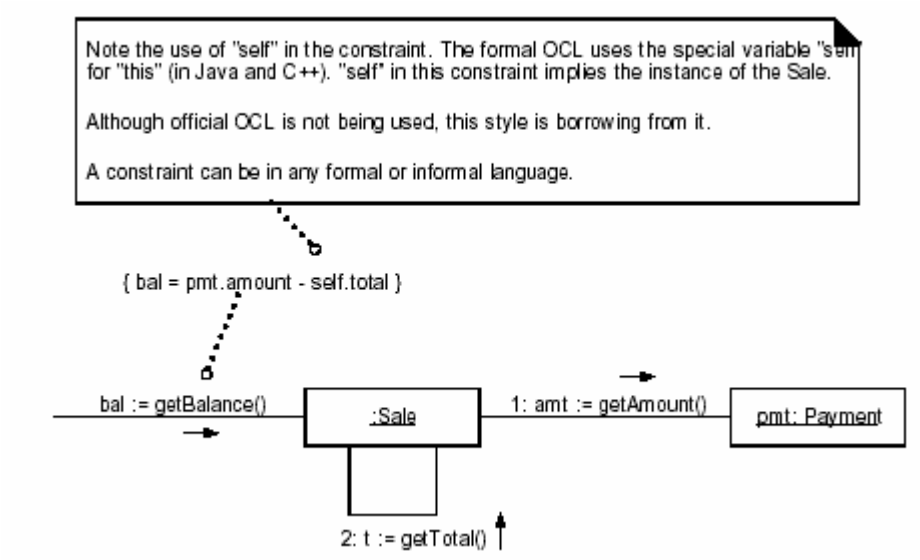


圖 17.16 Sale—getBalance 互動圖。

第八節物件設計：**startUp**

何時開始進行 **startUp** 設計

大部分時候（有可能不是所有情況），系統中都會有一個*啓動*使用案例與啓動應用程式的初始化系統操作。雖然 *startUp* 是最早開始執行的系統操作，不過我們最好在考量其它系統操作之後，再開始發展這個互動圖。這樣我們才可以確認考慮初始化開發活動之前，已經找到稍後系統操作執行時所需的一些資訊（譯註：還記得我們需要有一個初始化動作清單嗎？這樣一來我們在考量一些系統操作時，可以隨時紀錄初始化動作。等到我們真的進行*啓動*使用案例設計工作時，就可以參考這份清單了。）

把初始化動作的設計工作放在最後做。

應用程式是如何啓動的

從抽象層面來看，*startUp* 操作代表應用程式被啓動後所執行的初始化階段。爲了瞭解如何設計出這個操作的互動圖，我們需要先了解初始化動作的情境：應用程式如何啓動與初始化會受到程式語言與作業系統的影響。

在所有的情況下，常見實作樣式的做法最終都會產生**初始領域物件**（initial domain object），它也是應用程式中第一個被產生的軟體「領域」物件。

術語小筆記：我們在後面的章節中將會說明如何用邏輯上的分層結構來組織應用程式，以分離應用程式中的主要考量因素。可能會有的分層結構包括 UI 層（針對 UI 方面的考量因素）與「領域」層（針對領域邏輯方面的考量因素。）構成設計模型中領域層的軟體類別其名稱大多都是由領域字彙表而來的，我們會在領域層中放應用程式邏輯。

一旦產生初始領域物件之後，我們就會由它負責產生直接性子代【direct child】領域物件（譯註：所謂直接性子代就是在這個類別中直接產生實例的那些類別，它們之間所存在的不是一般化關係【generalization】，而是相依性關係【dependency】。）舉例來說，如果我們選擇 *Store* 作爲初始領域物件，那麼它就必須負責 *Register* 物件的產生工作。

至於產生初始領域物件的地方則依照所使用物件技術不同而異。例如在 Java 應用程式中，我們可能會在 *main* 方法中產生它，或者把這個工作委託給 *factory* 物件。

```
public class Main
{

public static void main( String[] args )
{
    // Store 是我們的初始領域物件。
    // Store 會負責產生一些其它領域物件。

    Store store = new Store();
    Register register = store.getRegister();
    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    …
}
}
```

startUp 系統操作的一些詮釋

從之前的討論中，我們可以發現 *startup* 系統操作是跟程式語言無關的抽象概念。在設計時，產生初始領域物件的地方可能不同、初始領域物件也不一定會取得處理程序的控制權。如果應用程式有 GUI 存在的話，初始領域物件通常不會取得控制權；否則，它通常會取得控制權。

我們在 *startUp* 系統操作的互動圖中展示：初始問題領域物件被產生時會發生什麼事，也可以選擇性展示它是否取得控制權。裡面不包括 GUI 層物件在之前或之後發生的任何活動（如果有的話。）

因此，我們可以把 *startUp* 系統操作重新詮釋成：

1. 在一張互動圖中，送出 *create* 訊息以產生初始領域物件。
2. （選擇性的）如果初始物件有取得處理程序控制權的話，那麼就在第二張互動圖中，送出 *run* 訊息（或意義相等的訊息）給初始物件。

POS 應用程式中的 startUp 操作

當經理開啓 POS 系統的電源、載入軟體後，就會發生 *startUp* 系統操作。這裡假設初始領域物件不會取得處理程序的控制權；我們產生初始領域物件之後，控制權還是保留在 UI 層上（例如 Java JFrame。）因此，*startUp* 系統操作的互動圖可以重新解釋成：單純送出 *create* 訊息，以產生初始物件。

選擇初始領域物件

初始領域物件應該是那個類別呢？
當我們在選擇初始領域物件時要選擇在領域物件容器階層（containment hierarchy）或聚合階層（aggregation hierarchy）中接近或位於頂端的類別。這個類別可能是表層介面型控制者（例如 *Register*）或某個被認爲會包含全部或大部分其它物件的物件（例如 *Store*。）
這些不同替代方案的選擇可能會受到高內聚力樣式與低耦合力樣式的考量因素影響。在這個應用程式中，我們選擇 *Store* 作爲初始物件。

永續物件：ProductSpecification

ProductSpecification 實例會存在於永續儲存媒體中，例如關連式資料庫或物件式資料庫。當我們執行 *startUp* 系統操作時，如果這樣的物件只有一些，我們可能會把它們載入電腦的直接記憶體中。然而，如果實例的數目很多，通通載入它們會消耗太多記憶體或時間。另一種可能解決方案 — 大部分情況下都是這種做法 — 則是根據需要，當我們需要個別實例時再把它們載入記憶體中。
如果我們使用物件式資料庫，那麼根據實際需要、動態從資料庫中載入物件的設計方式就很簡單，不過如果是使用關連式資料庫的話，就有點困難。我們現在先不考慮這個問題，並且做一個簡化問題的假設：我們可以「很神奇地」在記憶體中產生所有 *ProductSpecification* 物件。
第 34 章中會探討永續物件的問題，並且介紹把永續物件載入記憶體的一種方式。

Store—create()的設計結果

這裡產生物件與初始化動作都是導源於之前其它設計工作的需要，例如處理 *enterItem* 的設計工作等等。根據之前的那些互動圖，我們可以找到下面的初始化

工作：

- *ProductCatalog* 需要跟 *ProductSpecification* 之間建立關聯。
- *Store* 需要跟 *ProductCatalog* 之間建立關聯。
- *Store* 需要跟 *Register* 之間建立關聯。
- *Register* 需要跟 *Register* 之間建立關聯。

圖 17.17 中是整個設計結果。根據造物主樣式，我們選擇由 *Store* 負責產生 *ProductCatalog* 與 *Register*。同樣地，我們也選擇由 *ProductCatalog* 負責產生 *ProductSpecification*。請記住，這裡產生 *ProductSpecification* 的方式只是一個暫時性解決方案。在最後設計結果中，會在需要時才把這些 *ProductSpecification* 從資料庫中具體化成物件。

UML 表示法：請注意，我們用代表重複動作的一小段表示式標示所有 *ProductSpecification* 實例的產生與加入容器動作，這個動作用序號後面的「*」表示。

模型與真實世界領域之間有一個有趣差異是：軟體 *Store* 物件只會產生*一個* *Register* 物件。真實的商店裡面卻可能放置*多個*真正的收銀機或 POS 終端機。然而，我們現在是在設計軟體，而不是要反映真實生活。在目前需求中，我們的軟體 *Store* 只需要唯一的軟體 *Register* 實例。

領域模型中概念性類別之間的多重性與設計模型中設計類別的多重性不一定相同。

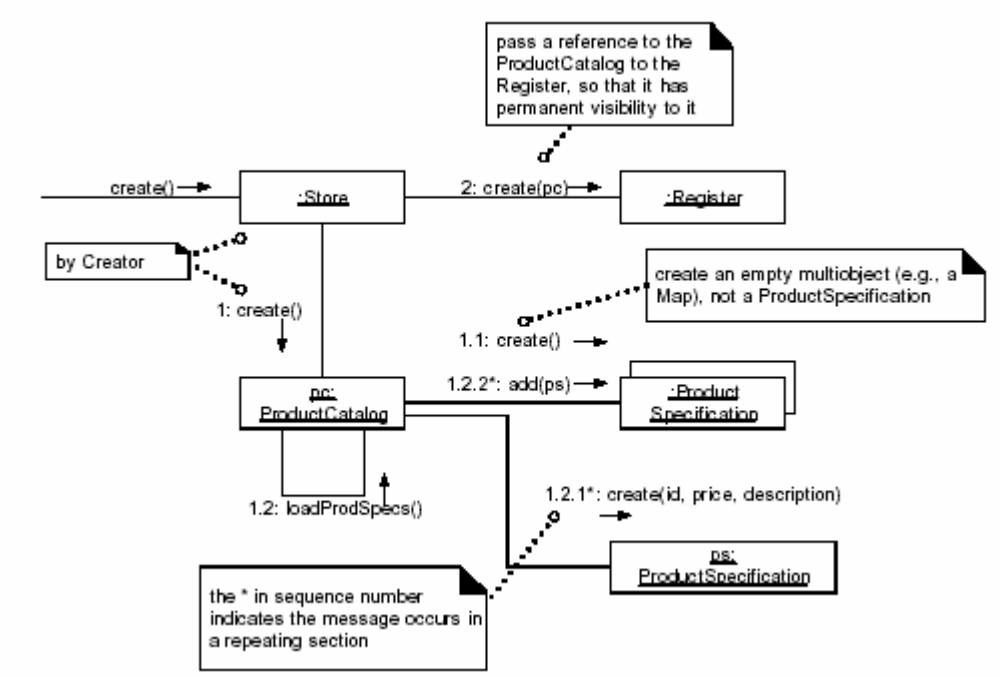


圖 17.17 初始領域物件的產生與後續產生的物件。

第九節從使用者介面層到領域層

我們之前已經簡單討論過：應用程式可以被組織成邏輯性的分層結構，以分離應用程式的不同主要考量因素，例如 UI 層（從 UI 方面考量因素）與「領域」層（從領域邏輯方面考量因素。）

爲了讓 UI 層的物件擁有對領域層物件的可見性，常見的設計方式爲：

- 由一段初始化副程式（例如 Java *main* 方法）同時產生 UI 物件與領域物件，並且把領域物件傳給 UI。
- UI 物件透過已知資源（例如負責產生領域物件的 factory 物件）讀取領域物件。

譯註：這兩種方式都無法解決分散式系統 UI 層與領域層之間的問題，此時 UI 層物件通常要透過目錄服務（directory service）或中介者（broker）取得領域層物件的遠端物件或遠端介面（remote interface），並且透過遠端方法呼叫（remote method invocation）傳送訊息。

下面是第一種解決方案的程式碼範例片段：

```
public class Main
{
public static void main ( String[] args )
{
    Store store = new Store();
    Register register = store.getRegister();
    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    …
}
}
```

一旦 UI 物件建立跟 *Register* 實例之間的連接（設計結果中的表層介面型控制者），它就可以把系統事件訊息轉送給 *Register*，例如 *enterItem* 與 *endSale* 訊息等等（請參見圖 17.18。）

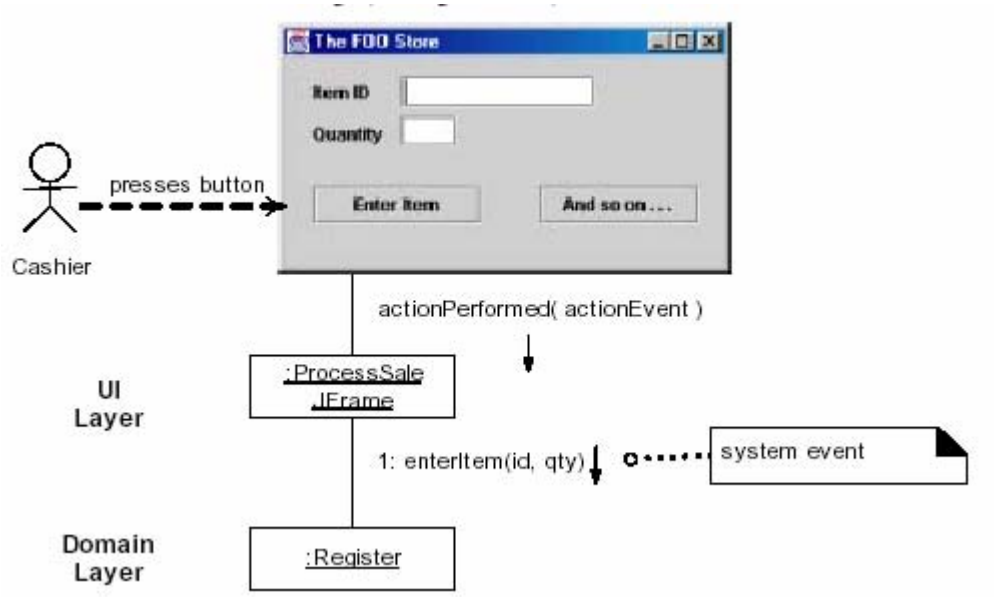


圖 17.18 建立 UI 層跟領域層之間的連接。

在 *enterItem* 訊息這個例子中，每次輸入商品項目後，視窗需要秀出目前的總金額。下面是幾種可能的解決方案：

- 替 *Register* 加上 *getTotal* 方法。UI 送 *getTotal* 方法給 *Register*，然後 *Register* 再轉送給 *Sale*。這樣做的優點是維持 UI 層跟領域層之間的低耦合力—UI 層只需要知道 *Register* 物件。不過我們必須增加 *Register* 物件的介面，這樣一來會降低它的內聚力。
- UI 會要求取得目前 *Sale* 物件的參考，當它需要銷售總金額時（或者任何其它跟銷售有關的資訊），就直接送訊息給 *Sale*。這樣的設計方式會增加 UI 層與領域層之間的耦合力。然而，就像我們在 GRASP 的低耦合力樣式中討論的，高耦合力本身並不是問題；然而，如果某個東西跟不穩定東西之間有耦合力的話則會形成問題（譯註：不穩定意謂著需要修改程式，很容易形成程式臭蟲。）假設 *Sale* 是穩定的物件，它屬於設計整體中的一部分 — 這樣的假設非常合理。那麼，跟 *Sale* 之間形成耦合力就不是問題。

如圖 17.19 所示，裡面的設計遵從第二種解決方案。

請注意，在這些圖中 Java 視窗（屬於 UI 層的一部份）並不負責處理應用程式邏輯。它把工作請求（系統操作）經由 *Register* 轉送到領域層。這一點導致下面的設計原則：

（使用者）介面層與領域層的責任

UI 層不應該包含任何領域邏輯責任。裡面應該只負責使用者介面工作，例如更新視窗小元件。

UI 層應該把所有設計導向的工作請求轉送到領域層，由它負責處理這些請求。

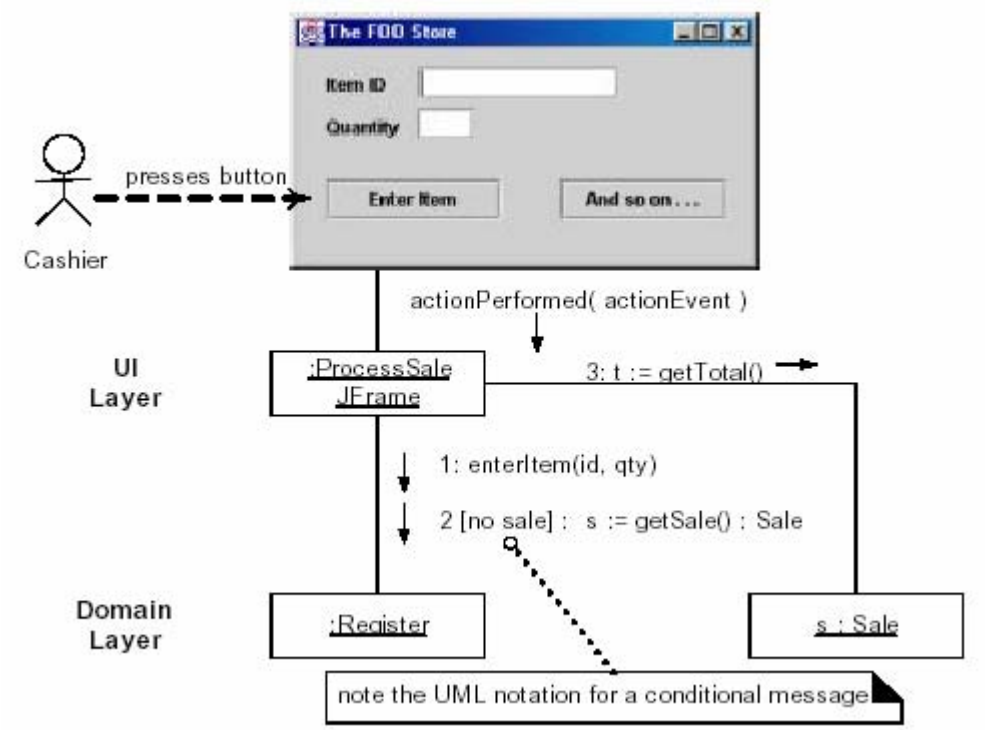


圖 17.19 連接 UI 層與領域層

第十節UP 中的使用案例實現

使用案例實現是 UP 中設計模型的一部份。本章把焦點放在畫互動圖上，不過一般來說我們建議同時畫出類別圖。類別圖會在第 19 章詳加說明。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	**領域模型**		S		
需求	使用案例模型(SSD)	s	R		
	專案願景	s	R		
	輔助規格書	s	R		
	字彙表	s	R		
設計	**設計模型**		S	r	
實作	軟體架構文件		S		
	資料模型		s	r	
	實作模型		s	r	r
專案管理	軟體開發	s	r	r	r

	計畫				
測試	測試模型		s	r	
環境	開發案例	s	r		

表 17.1 可能用到的 UP 工作成果以及產生它們的時間點。s － 初版；r － 修正版

開發階段

初始階段—我們通常會等到詳述階段才開始製作設計模型與使用案例實現，因爲在初始階段詳細的設計決策還不成熟。

詳述階段—在這個開發階段，我們必須針對顯著影響到架構或有風險的情節做出使用案例實現。然而，我們不必替所有情節畫出 UML 圖，而且也不必畫出完整、細部細節。這裡的概念是只針對關鍵使用案例實現畫出互動圖，獲得一些前瞻性想法，並且探索替代方案，也就是把焦點放在主要設計決策上。

建構階段—我們針對其它設計問題畫出使用案例實現。

UP 中的工作成果與開發流程情境

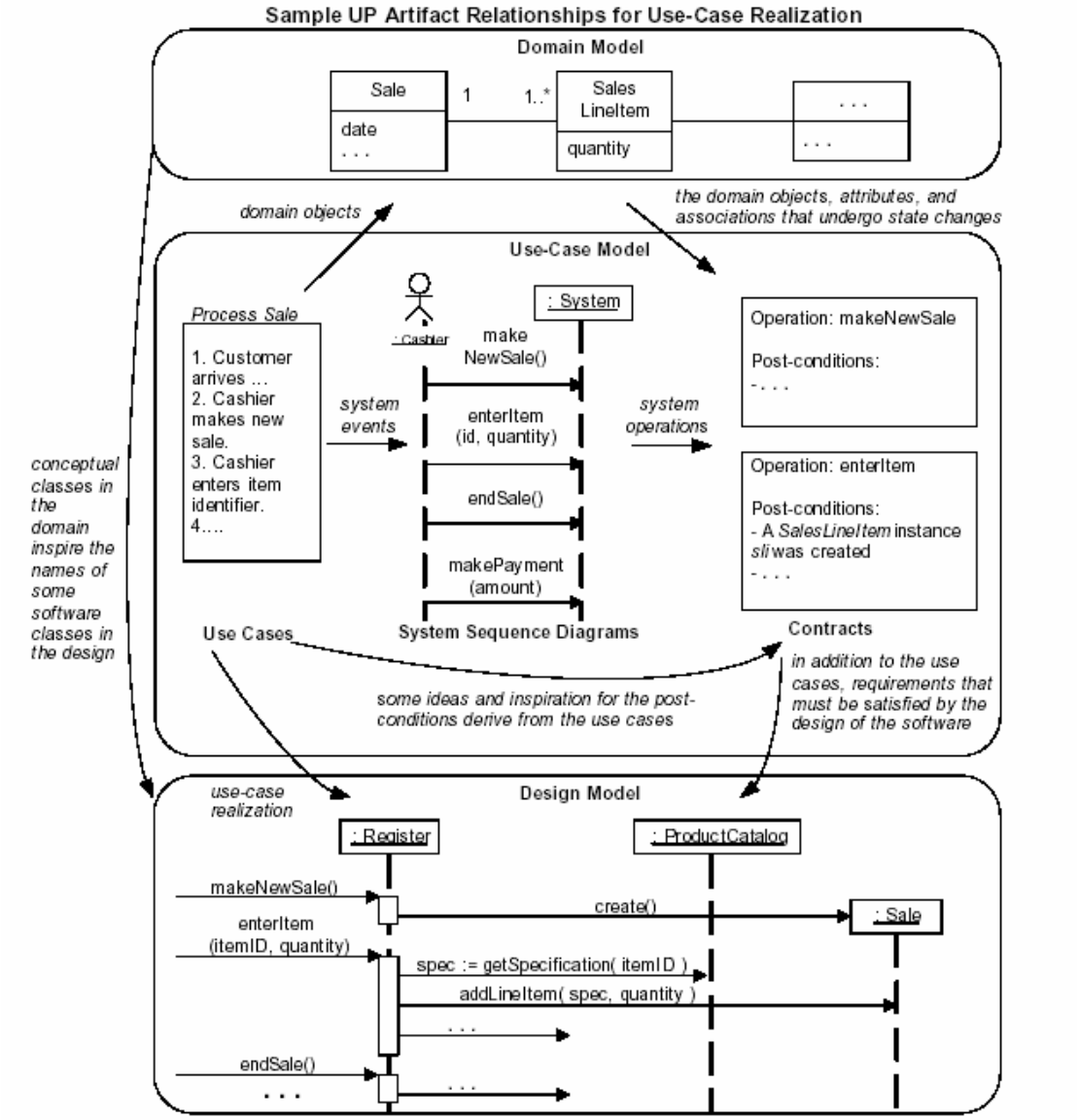


圖 17.20 UP 中工作成果之間可能發生的相互影響。

在 UP 中，使用案例實現的製作工作屬於設計開發活動。圖 17.21 提供我們進行這項工作時在時間與空間上的建議。

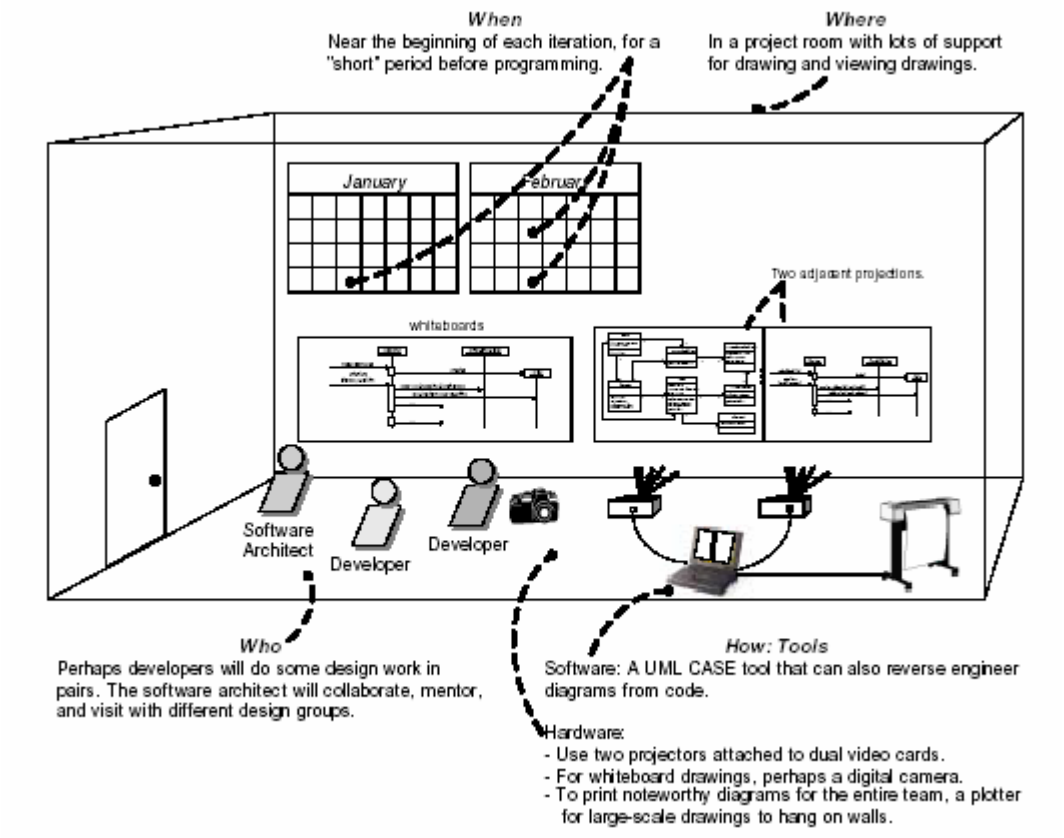


圖 17.21 開發流程與情境的設定

第十一節總結

設計物件互動情形與指派責任是物件設計的核心工作。這些設計決策對物件軟體系統的可擴充性、清晰度、可維護性，以及元件可重複使用程度與品質有很重大影響。有些設計原則有助於做出指派責任決策；GRASP 樣式中彙總一些最具一般性、物件導向設計師最常見的設計原則。

第十八章設計模型：決定可見性

數學家是幫我們從咖啡中找出定理的人。
— Paul Erdos

本章目標

- 找出四種不同可見性。
- 設計可見性。
- 用 UML 表示法展現各種可見性。

簡介

所謂的可見性就是一個物件可以看到或參考另一個物件的能力。本章探索跟可見性相關的一些設計議題。

第一節物件之間的可見性

針對系統事件（*enterItem* 等等）所做的設計結果中必須展現物件之間的訊息。當傳送物件傳送訊息給接收物件時，傳送物件必須看得見接收物件 — 傳送物件必須擁有接收物件的某種參考或指標。
舉例來說，當 *Register* 可以傳 *getSpecification* 訊息到 *ProductCatalog* 時，隱含 *Register* 實例是看得見 *ProductCatalog* 的，請參見圖 18.1。

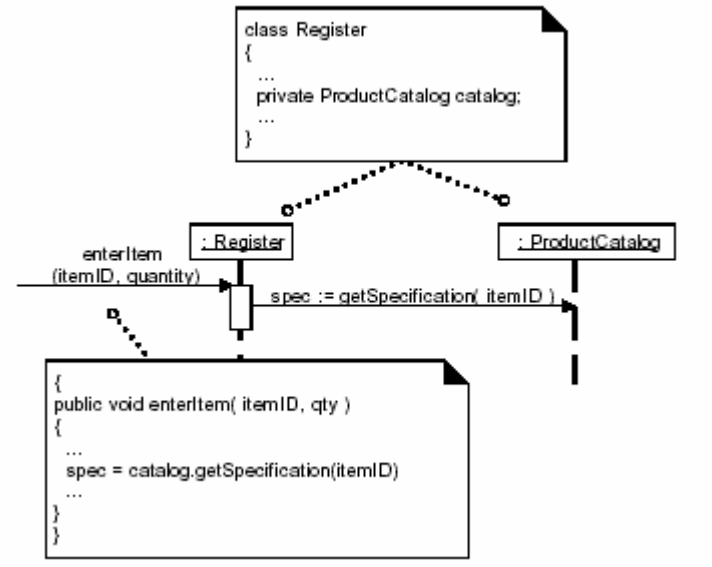


圖 18.1 我們需要從 *Register* 到 *ProductCatalog* 的可見性【註】。
【註】：在這個程式碼範例與之後的範例中，都會簡化程式讓它看起來更簡潔、

清晰。
當我們用物件的互動情形描述設計結果時，必須確認有足夠的可見性可以支援訊息互動。

第二節可見性

在一般用法中，**可見性**（visilibty）是一個物件可以「看到」或參考另一個物件的能力。更一般的說法是它跟可視範圍的議題有關：某個資源（例如一個實例）是否存在於另一個資源的可視範圍之中？有四種方式可以達成物件 A 到物件 B 的可見性。

- 屬性可見性 — B 是 A 的屬性。
- 參數可見性 — B 是 A 中某個方法的參數。
- 區域可見性 — B 是 A 中某個方法的（非參數型）區域物件。
- 全域可見性 — 用某種方式讓全域都看得到 B。

考量可見性的動機爲：

爲了讓物件 A 傳送訊息給物件 B，B 必須被 A 看見。

舉例來說，當我們產生互動圖，在互動圖中 *Register* 實例會傳送訊息給 *ProductCatalog* 實例，這時候 *Register* 必須對 *ProductCatalog* 有可見性。典型的可見性解決方案是在 *Register* 中放一個屬性，作爲連到 *ProductCatalog* 實例的參考。

屬性可見性

當 B 是 A 的屬性時，就會存在從 A 到 B 的**屬性可見性**（attribute visibility）。這是一個比較永久性的可見性，因爲當 A 與 B 存在時，這個可見性都會存在。在物件導向系統中，這是很常見的可見性形式。
爲了說明這種可見性，我們用 Java 定義了一個 *Register* 類別，在 *Register* 實例中可有可以看到 *ProductCatalog* 的可見性，因爲 *ProductCatalog* 是 *Register* 的一個屬性（Java 實例變數。）

```
public class Register
{
…
private ProductCatalog catalog;
…
}
```

圖 18.2 中需要屬性可見性，因爲在 *enterItem* 圖中，*Register* 需要傳 *getSpecification* 訊息給 *ProductCatalog*：

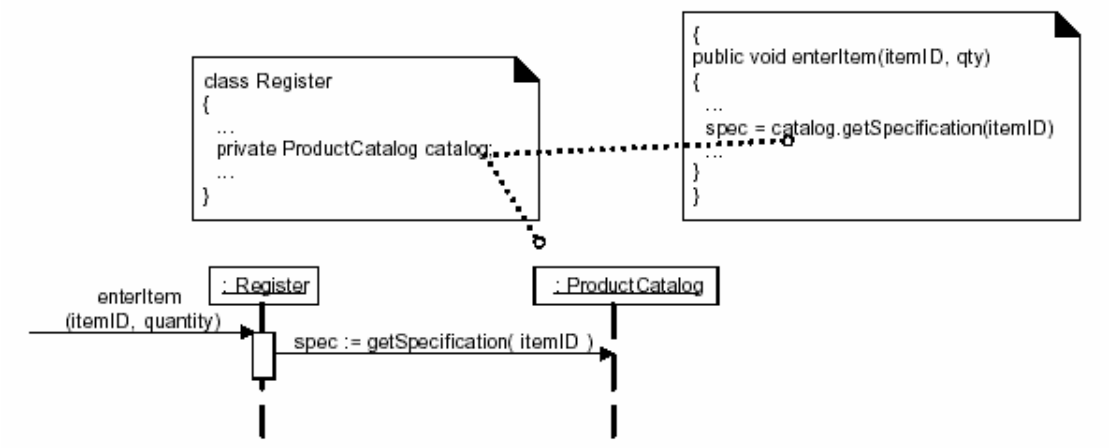


圖 18.2 屬性可見性。

參數可見性

當 B 被當成參數傳到 A 的方法時，從 A 到 B 就存在**參數可見性**（parameter visibility）。這個可見性是比較暫時性的可見性，因爲它的可見範圍只在方法之內。它是除了屬性可見性之外，物件導向系統中第二常見的可見性。

請參見圖 18.3 ，裡面展現參數可見性。當 *makeLineItem* 訊息被送到 *Sale* 實例時，會把 *ProductSpecification* 實例當參數傳過去。在 *makeLineItem* 方法的範圍內，*Sale* 擁有 *ProductSpecification* 的參數可見性。

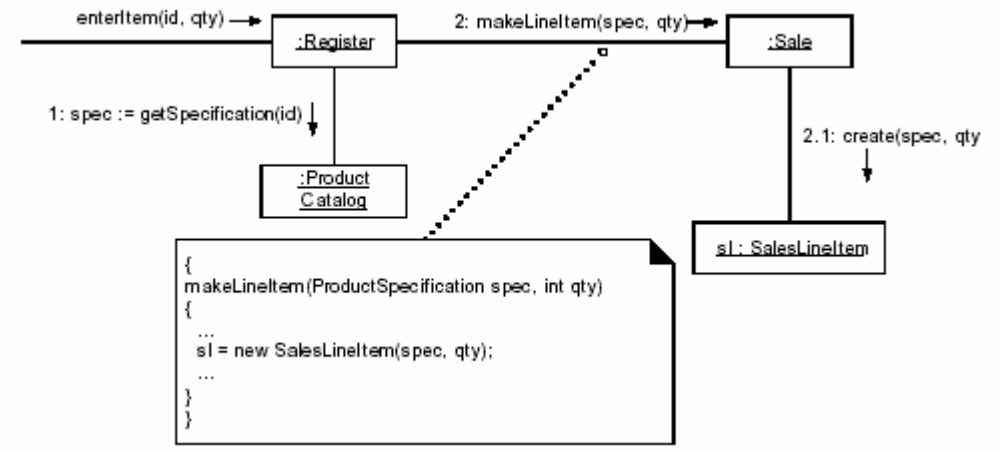


圖 18.3 參數可見性。

我們常常會把參數可見性轉變成屬性可見性。例如當 *Sale* 產生新的 *SaleLineItem* 時，會把 *ProductSpecification* 傳給初始化方法（在 C++或 Java 中，初始化方法稱爲**建構子**【constructor】。）在初始化方法中，參數會被指定到某個屬性上，以建立屬性可見性（圖 18.4。）

區域可見性

當 B 是 A 的方法中的一個區域物件，那麼從 A 到 B 就存在區域可見性。它也是比較暫時性的可見性，因爲它的可見範圍只在方法之內。在物件導向系統中，它是繼參數可見性之後，第三常見的可見性形式。

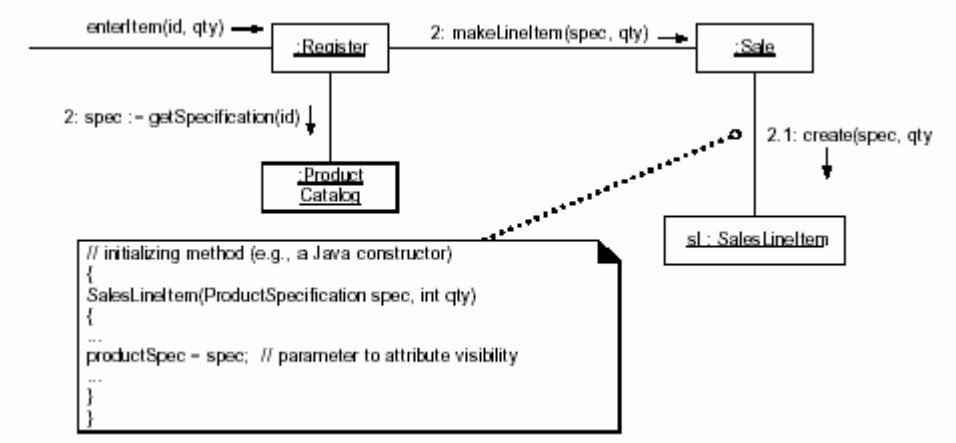


圖 18.4 把參數可見性轉變成屬性可見性。

有兩種方式可以達成區域可見性：

- 產生一個新的區域實例，並且把它指定給區域變數。
- 呼叫某個方法後，把傳回的物件指定給區域變數。

跟參數可見性一樣，我們會常常把區域可見性轉變成屬性可見性。

我們可以從圖 18.5 類別 *Register* 的 *enterItem* 方法中看到用第二種方法達成區域可見性的例子，圖中會把傳回的物件指定給區域變數。

跟第二種方法有點小差異的作法是沒有把傳回的物件指定給區域變數，而直接呼叫它的方法，這個物件的存在就沒有那麼明顯。

```
// 呼叫 getFoo 方法所傳回來的 foo 物件，
// 這裡對它有隱性的區域可見性存在。
anObject.getFoo().doBar();
```

全域可見性

如果 B 對 A 來說是全域的話，那麼從 A 到 B 就存在**全域可見性**(global visibility)。這是一個比較永久性的可見性，因爲當 A 與 B 存在時，這個可見性都會存在。在物件導向系統中，這是最不常見的可見性形式。

達成全域可見性的方法之一是把實例指定給全域變數，某些程式語言中可以這樣作（例如 C++），其它則否（例如 Java。）

達成全域可見性的最好方式是使用**唯一物件樣式**（Singleton）【GHJV95】，我們會在後面的章節中討論。

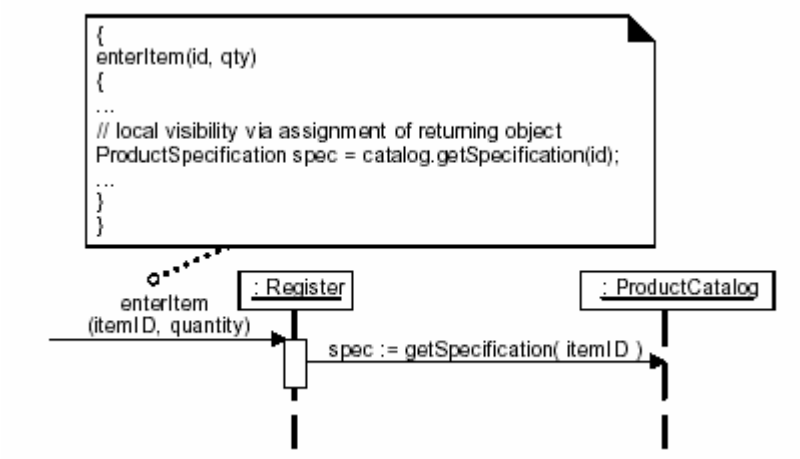


圖 18.5 區域可見性。

第三節在 UML 中展現可見性

UML 的合作圖中有秀出某種可見性的表示法（請[參見圖 18.6。）不過這些修飾性的表示法都是選擇性使用的，而且我們通常很少用它們；只有特別想釐清某個東西時才會用它。

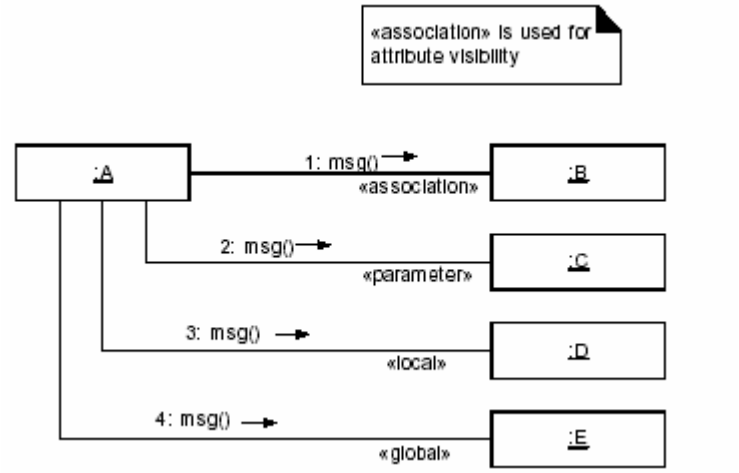


圖 18.6 可見性用的實作造型。

第十九章設計模型：產生設計類別圖

反覆、遞迴、憑直覺推測都是天性。
— 無名氏

本章目標

- 產生設計類別圖。
- 找出類別、方法與關聯，並且把它們畫在設計類別圖中。

簡介

當我們完成 NextGen POS 應用程式目前反覆中使用案例實現所需的互動圖後，應該可以找到軟體解決方案中的軟體類別（介面）規格，並且爲規格加上設計細節，例如方法。
UML 的類別圖中有可表現設計細節的表示法；本章會探討這些細節，並且產生 DCDs。

第一節何時該產生設計類別圖

雖然我們先介紹互動圖的製作，然後再介紹 DCDs，不過事實上，我們會同時製作這兩種工作成果。有可能會在早期開發時期、還沒畫出互動圖之前，我們就會先應用責任指派樣式很快地粗略寫出許多類別、方法名稱與關係。有可能、也需要在畫出一些互動圖圖之後，再去更新 DCDs，然後隨著互動圖越畫越多，更新更多的 DCDs。
跟 CRC 卡比起來，類別圖是紀錄責任與合作情形的另一種選擇，它也是比 CRC 卡更圖形化的表示法。

第二節設計類別圖的範例

圖 19.1 的 DCD 中展現 *Register* 與 *Sale* 類別的部分軟體定義。

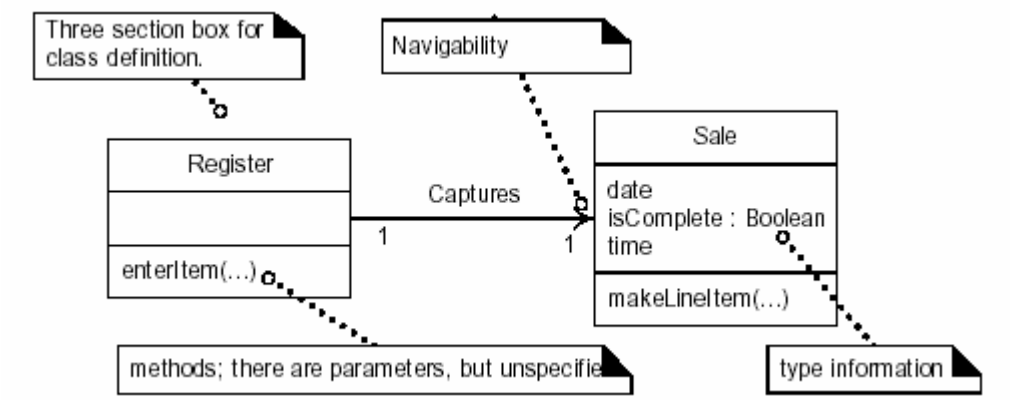


圖 19.1 設計類別圖範例。

除了基本關聯與屬性之外，設計類別圖中也可以展現出每個類別的方法、屬性型態資訊與物件之間屬性可見性與瀏覽性等等。

第三節設計類別圖與 UP 中（設計類別圖）的相關術語

設計類別圖（design class diagram，DCD）會展現應用程式中軟體類別與介面的規格（例如 Java 介面。）典型資訊包括：

- 類別、關聯與屬性
- 介面以及介面中的操作與常數
- 方法
- 屬性型態資訊
- 可瀏覽性
- 相依性

跟領域模型中的概念性類別不同，DCDs 中的設計類別會顯示出軟體類別的定義而非真實世界概念。

UP 中並沒有特別定義稱爲「設計類別圖」的工作成果。不過，UP 中有定義設計模型，裡面包含一些圖：互動圖、套件圖與類別圖。UP 設計模型中的類別圖會包含所謂的 UP「設計類別。」因此，我們常常把「設計類別圖」簡稱爲「設計模型中的類別圖。」

第四節領域模型中的類別 vs. 設計模型中的類別

再重複一遍，UP 的領域模型中，*Sale* 並不是軟體定義；它是真實世界概念的抽象概念，前者可能代表我們有興趣的一段文字。另一方面，DCDs 則是針對軟體應用程式，把類別定義表達成軟體元件。在這些圖中，*Sale* 代表軟體類別（請參見圖 19.2。）

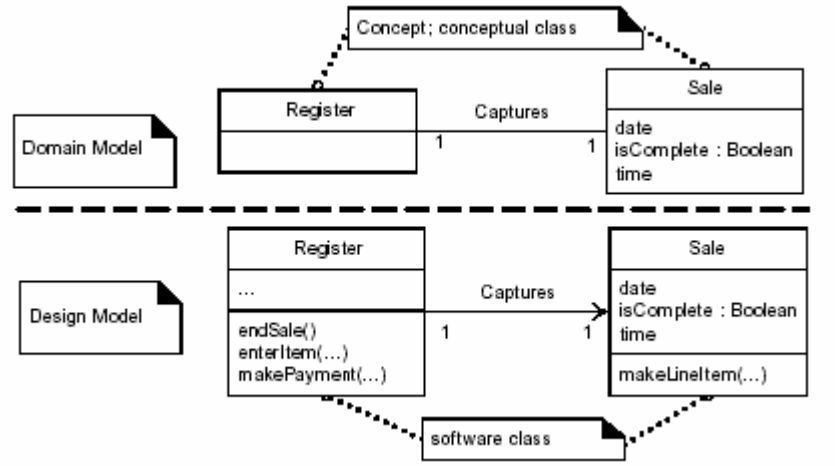


圖 19.2 領域模型中的類別 vs. 設計模型中的類別。

第五節產生 NextGen POS 系統的設計類別圖

找出軟體類別，並且把它們展現在設計類別圖中

產生 DCDs（解決方案中的部分工作成果）的第一步是要找出軟體解決方案中的類別。我們可以檢視所有的互動圖，並且列出裡面提到的類別以找出這些類別。POS 應用程式裡面的設計類別包括：

Register	Sale
ProductCatalog	ProductSpecification
Store	SalesLineItem
Payment	

第二步則是畫出這些類別的類別圖，並且把之前在領域模型找到的屬性放入圖中（請參見圖 19.3。）

請注意，領域模型中的有些概念（例如 *Cashier*）可能不會出現在設計結果中。在目前反覆中，我們有可能還不需要把它們放入軟體中。然而，在稍後的反覆中，

當我們考量新需求與使用案例時，可能才會把它們放入設計結果中。例如，當我們實作安全性與登入系統需求時，可能才會放入叫做 *Cashier* 的軟體類別。

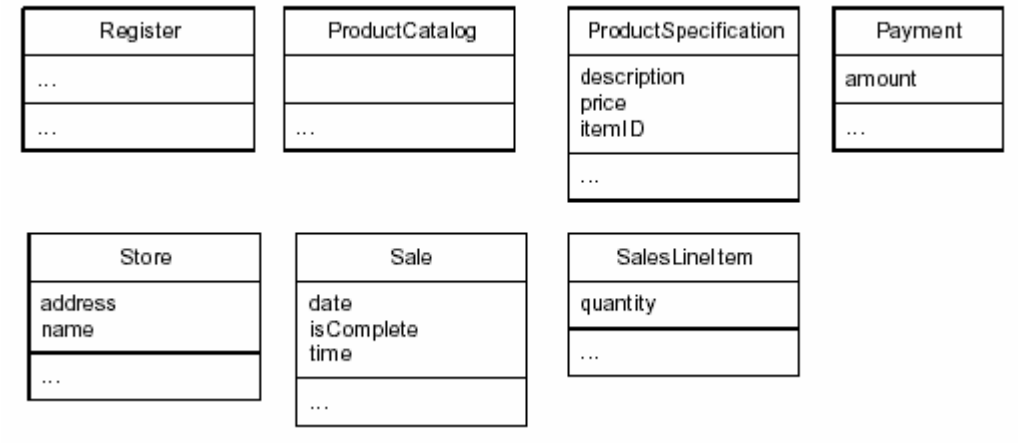


圖 19.3 應用程式中的軟體類別。

加入方法名稱

分析互動圖可以幫我們找出每個類別的方法。例如如果我們傳 *makeLineItem* 訊息給類別 *Sale* 的實例，那麼類別 *Sale* 裡面就會有一個 *makeLineItem* 方法（請參見圖 19.4。）

一般來說，在所有互動圖中被傳送到類別 X 的所有訊息將是類別 X 中的大部分方法。

我們檢查 POS 應用程式的互動圖後會產生圖 19.5 中的方法配置結果。

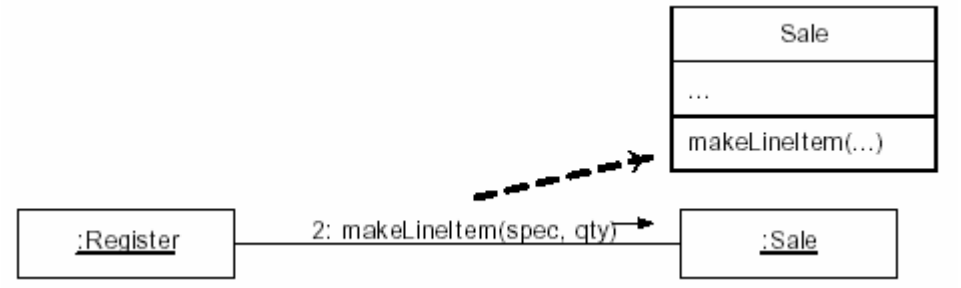


圖 19.4 從互動圖得到的方法名稱。

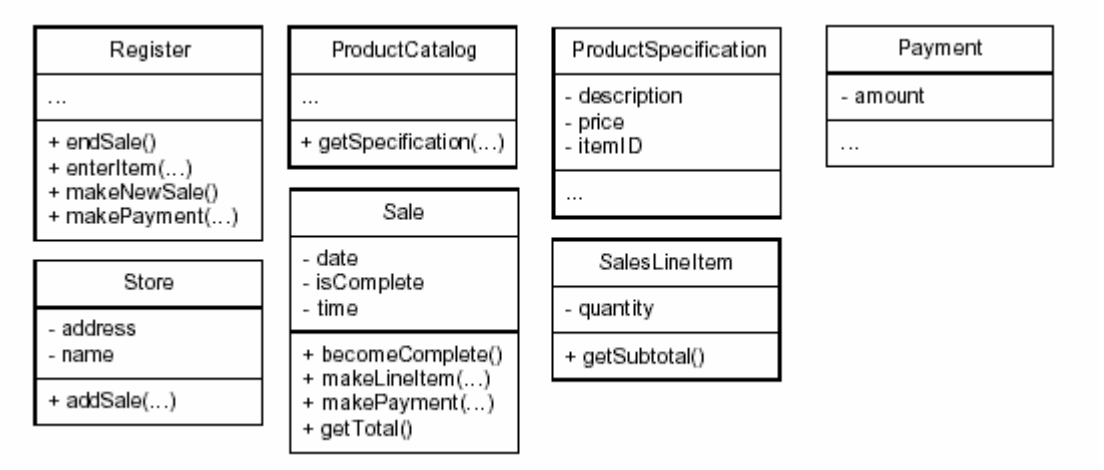


圖 19.5 應用程式中的方法。

跟方法名稱相關的一些議題

我們必須考慮下面跟方法名稱相關的一些議題：

- 對 *create* 訊息的詮釋
- 存取型方法的敘述
- 詮釋傳到多重物件訊息
- 跟程式語言有關的程式語法

方法名稱 — create

create 訊息是 UML 中跟程式語言無關、標示實例化與初始化動作的一種訊息。當我們把這個設計結果轉換成物件導向程式語言時，必須用一些實例化與初始化的實作樣式表現。在 C++、Java 或 Smalltalk 中，並沒有程式語言規定的 *create* 方法存在，它隱含呼叫建構子之後，會自動配置記憶體或用 *new* 操作配置被釋放掉儲存記憶體。

因爲這個訊息有不同的解釋方式，而且初始化動作又是非常常見的開發活動，所以我們在 DCD 中常常省略跟 *create* 相關的方法與建構子。

方法名稱 — 存取型方法

存取型方法（accessing method）會讀取（讀取子方法）或設定（設定子方法）屬性。在某些程式語言中（例如 Java），把所有屬性宣告成私有的（以加強封裝），並且讓每個屬性都有讀取子【accrssor】或設定子【mutator】是常見的實作樣式。我們通常不會在圖中畫出這些方法，因爲畫出來的價值遠低於所造成的困擾；每 n 個分析就會產生 2n 個我們沒興趣的方法。例如 *ProductSpecification* 的 *getPrice*（或 *price*）方法雖然存在，不過我們不會秀出它，因爲這是一個簡單的讀取子

方法。

方法名稱 — 多重物件

傳到多重物件的訊息會把解釋成傳到容器／多重物件本身的訊息。例如圖 19.6 中傳到多重物件的*find* 訊息被解釋成傳到容器／多重物件（例如 Java 的 *Map*、C++的 *map* 或 Smalltalk 的 *Dictionary*）本身的訊息。

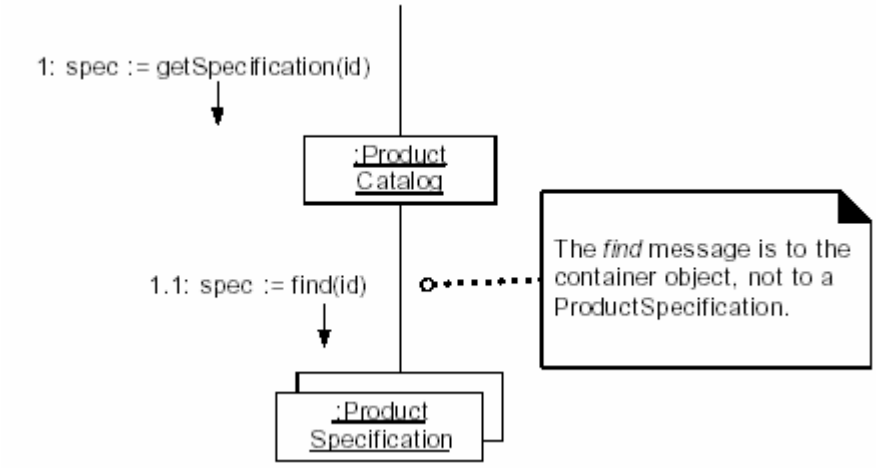


圖 19.6 傳到多重物件的訊息。

因此，*find* 訊息不屬於 *ProductSpecification* 類別；相反地，在 *ProductSpecification* 類別中加入 *find* 訊息是不正確的。

這些容器／多重物件介面或類別（例如 *java.util.Map* 介面）通常是程式庫中事先定義好的元素，而且在 DCD 中也不會明確秀出這些類別，因爲帶來的資訊不多卻造成干擾。

方法名稱 — 跟程式語言相關的程式語法

某些程式語言（例如 Smalltalk）中表達方法的語法可能跟 UML 基本格式：*方法名稱(參數串列)*不同，這時候縱然計畫用來實作的程式語言有不同程式語法，我們還是建議採用 UML 基本格式。在理想狀況下，我們會在產生程式碼時，把 UML 基本格式轉換成實作程式語言的語法，而不是在產生類別圖時就採用實作程式語言的語法。然而，UML 還是允許我們對方法規格用不同的語法。

加入更多型態資訊

屬性、方法參數與方法傳回值的型態都是可選擇性秀出來的。決定要不要秀出某些資訊這個問題時，我們必須考量下面情境：

DCD 的製作是依照閱讀它的人而定的。

- 如果產生 DCD 是爲了用 CASE 工具產生程式碼，那麼就需要完整、詳盡的

細節資訊。

- 如果產生 DCD 是爲了給軟體開發人員看，那麼太詳盡、完整的細節反而會造成困擾，價值也不高。

舉例來說，我們需要秀出所有參數與參數型態資訊嗎？這個問題的答案要看你期待的觀眾，以決定是否需要更多明顯資訊。

圖 19.7 中的設計類別圖就包含比較多的型態資訊。

加入關聯與可瀏覽性

關聯的端點都是一個角色，而且在 DCDs 中，我們可以用可瀏覽性箭頭修飾角色。**可瀏覽性**（navigability）是角色的外顯屬性，用它說明我們可以從來源物件單向瀏覽到目的物件。可瀏覽性隱含可見性 — 通常是屬性可見性（請參見圖 19.8。）

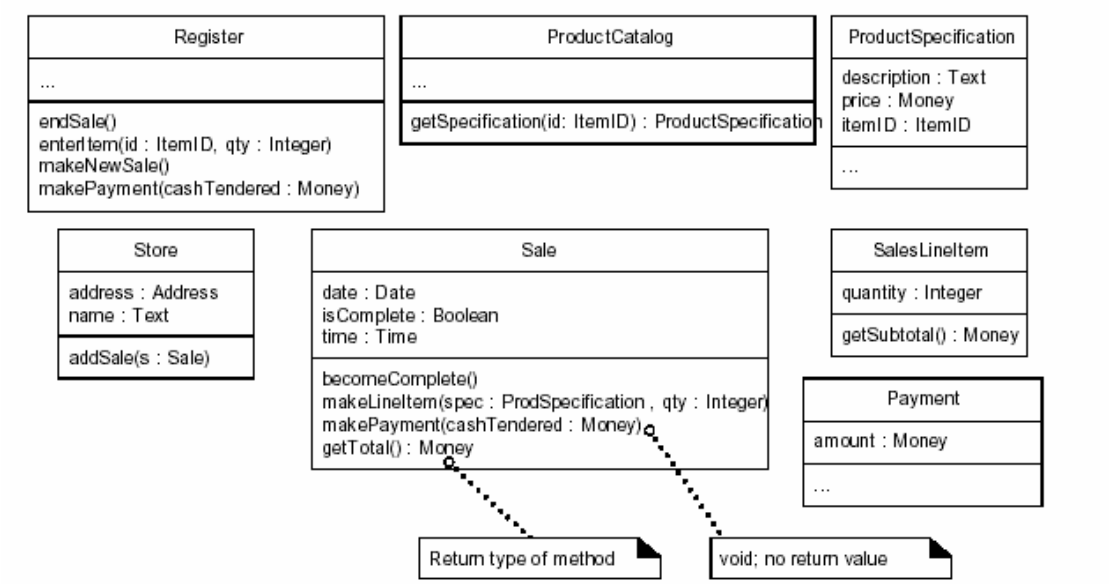


圖 19.7 加入型態資訊

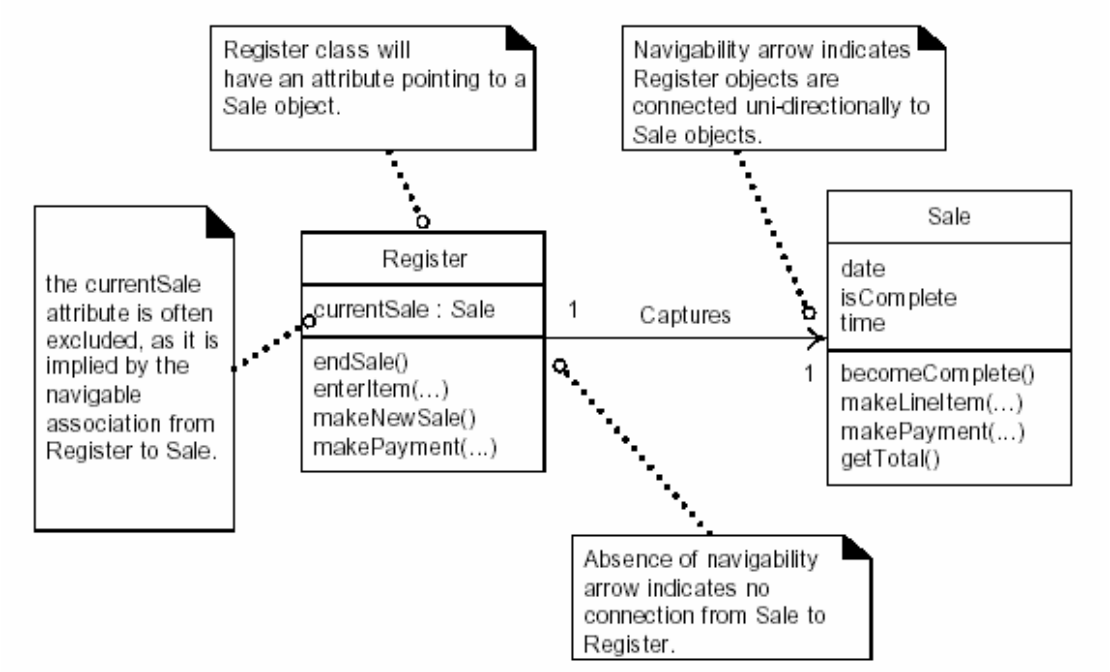


圖 19.8 秀出可瀏覽性或屬性可見性。

我們通常會把具有可瀏覽性箭頭的關聯解釋成從來源物件到目的物件的屬性可見性。當我們用物件導向程式語言實作可瀏覽性時，通常會把它轉變成：在來源類別中用一個屬性參考到目的類別的實例。例如 *Register* 類別可能會定義參考 *Sale* 實例的一個屬性。

（縱然不是全部情況）在大部分情況下，DCDs 中的關聯都會用畫出必要的可瀏覽性箭頭。

在 DCD 中，我們會用很強烈的「軟體導向、一定要知道」的準則選擇要不要關聯 — 在互動圖中需要存在哪個關聯以滿足可見性與後續記憶需要？這樣的關聯跟領域模型的關聯是不同的，後者可能只是想增進問題領域的了解。這又是設計模型與領域模型目標上的差異：一個是分析性的，而另一個則是爲了描述軟體元件。

我們會把類別之間的必要可見性與關聯標示在互動圖中。發生下面情況時，我們建議在從 A 到 B 的關聯上加上可瀏覽性箭頭：

- A 傳送訊息給 B。
- A 產生實例 B。
- A 需要維護連到 B 的參考。

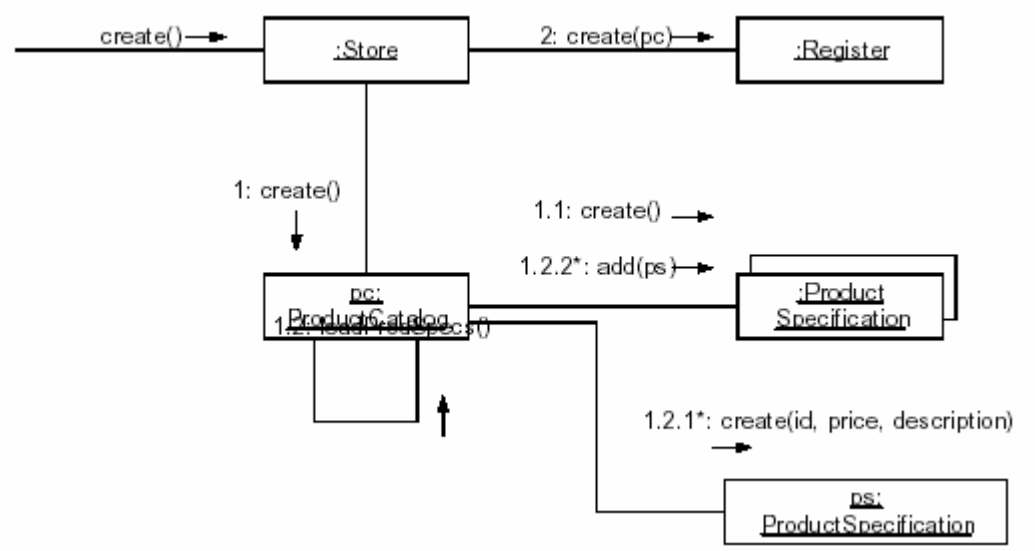


圖 19.9 互動圖中找到的可瀏覽性。

舉例來說，圖 19.9 中的互動圖是由傳送到 *Store* 的 *create* 訊息啓動的，它是範圍更廣的互動圖中的一部分，我們從圖中可以看出 *Store* 應該持續跟它所產生的 *Register* 與 *ProductCatalog* 實例建立連結。同樣地，*ProductCatalog* 也需要繼續跟它所產生的 *ProductSpecification* 建立連結。事實上，負責產生其它物件的物件通常需要繼續連到被產生的物件上。因此，在類別圖中就會相對存在關聯，以代表隱含要存在的連結。

根據上面對關聯與可瀏覽性的準則，分析 NextGen POS 應用程式的所有互動圖之後，我們可以產生圖 19.10 的類別圖，裡面畫出必要關聯（這裡爲了更清楚看出關聯，所以並沒有畫出詳盡的型態資訊。）

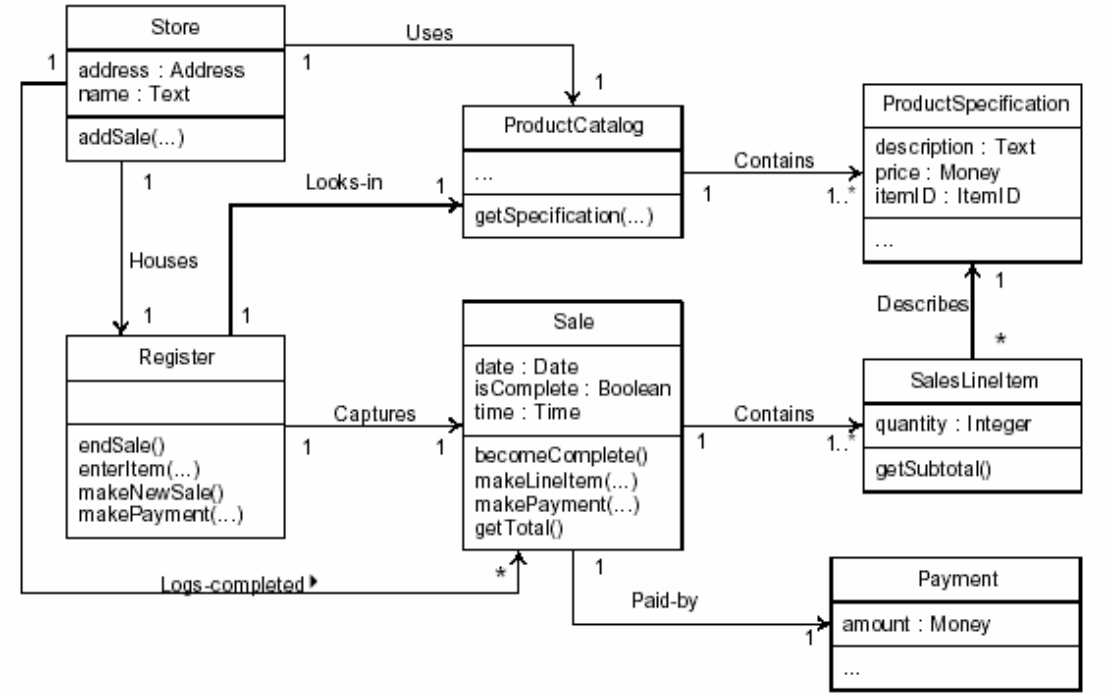


圖 19.10 加上可瀏覽性的關聯。

請注意，這裡的關聯跟領域模型類別圖中的關聯並不會完全一樣。例如，在領域模型中，*Register* 與 *ProductCatalog* 之間並沒有 *Looks-in* 關聯存在，之前我們並沒有發現這是重要、持續存在的關係。直到畫互動圖時，我們才發現軟體 *Register* 必須保持跟軟體 *ProductCatalog* 之間的連結以查詢 *ProductSpecification*。

加入相依性關係

UML 用一般的**相依性關係**（dependency relationship）標示：某個元素（例如類別、使用案例等等）擁有另一個元素的相關知識的情形。我們用虛線段的箭號代表相依性關係。在類別圖中，我們用相依性關係描述類別之間的非屬性可見性；換言之，參數可見性、全域可見性或區域可見性。另一方面，屬性可見性則用標準、帶可瀏覽性箭頭的關聯展現。例如 *Register* 軟體物件傳送訊息給 *ProductCatalog* 型態的物件之後會收到 *ProductSpecification* 型態的物件。因此，*Register* 對 *ProductSpecification* 有區域可見性。此外，*Sale* 會從 *makeLineItem* 訊息中收到 *ProductSpecification* 參數，它對 *ProductSpecification* 有參數可見性。非屬性可見性用虛線段的箭號代表相依性關係（請參見圖 19.11。）圖中畫相依性關係的箭號不是直線是曲線，只是爲了畫圖方便而已。

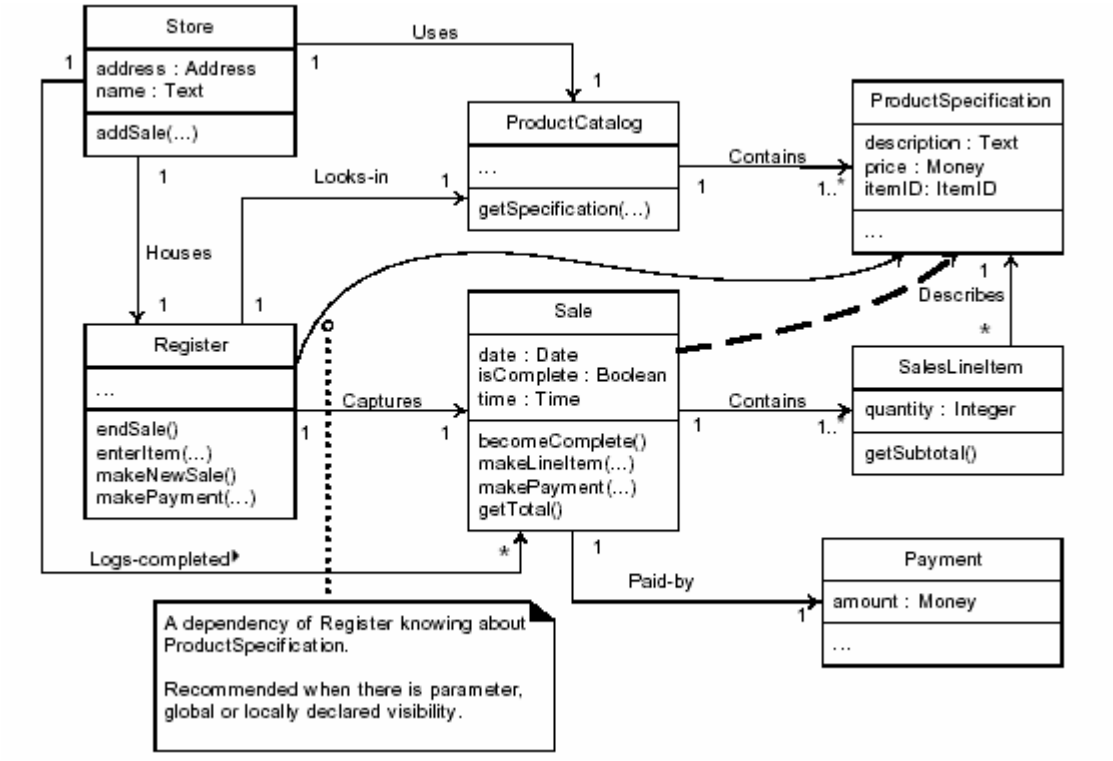


圖 19.11 代表非屬性可見性的相依性關係。

第六節用來描述成員細節的表示法

UML 中提供很豐富的表示法描述類別與介面成員的語言特性，例如可見性、初

始值等等。如圖 19.12 所示。

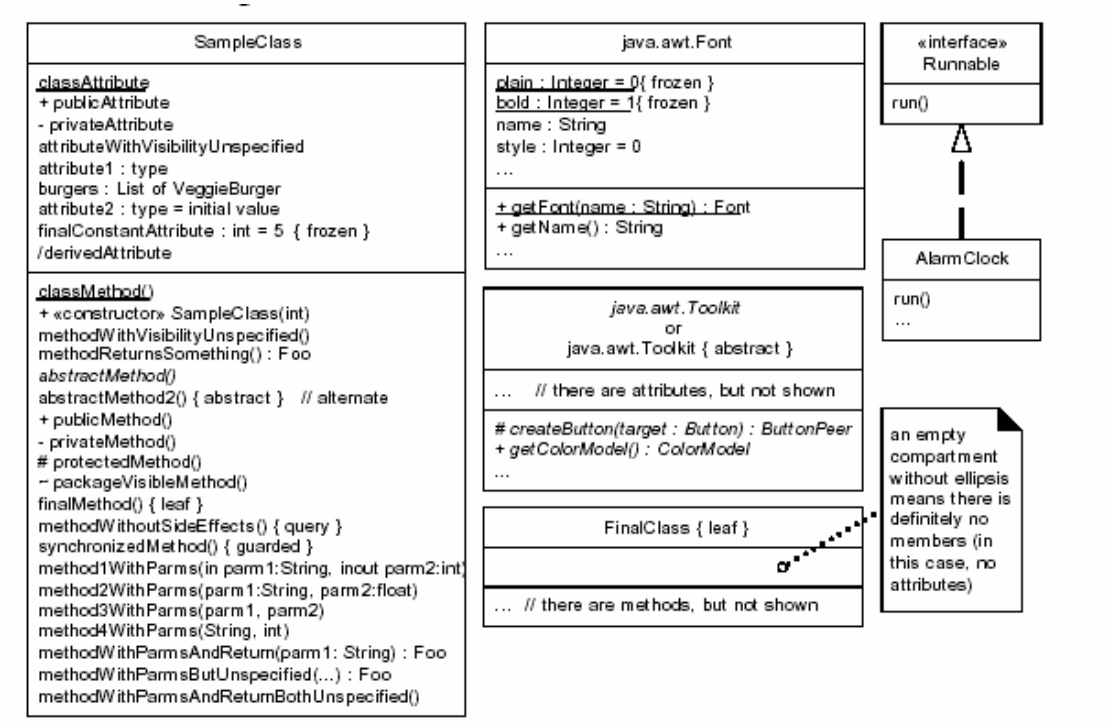


圖 19.12 UML 類別圖中用來標示成員細節的表示法。

UML 的預設可見性爲何？

如果圖中沒有用明確的可見性標示屬性或方法，預設值爲何？答案：沒有預設值。如果沒有秀出可見性的話，在 UML 中就是「沒有指定。」然而，常見慣例會假設沒有標示代表：屬性是私有的，而方法是公開的。

NestGen POS 目前反覆的設計類別圖中（請參見圖 19.13）並沒有畫出很多有興趣成員的細節；所有屬性都是私有的，而方法是公開的。

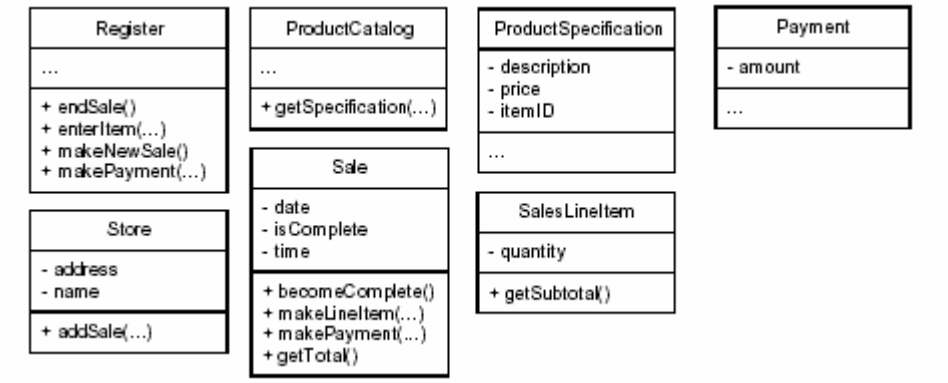


圖 19.13 POS 類別圖中成員的細節。

設計類別圖（與互動圖）中方法本體所用

的表示法

我們可以用圖 19.14 的方式在 DCD 與互動圖中呈現方法本體。

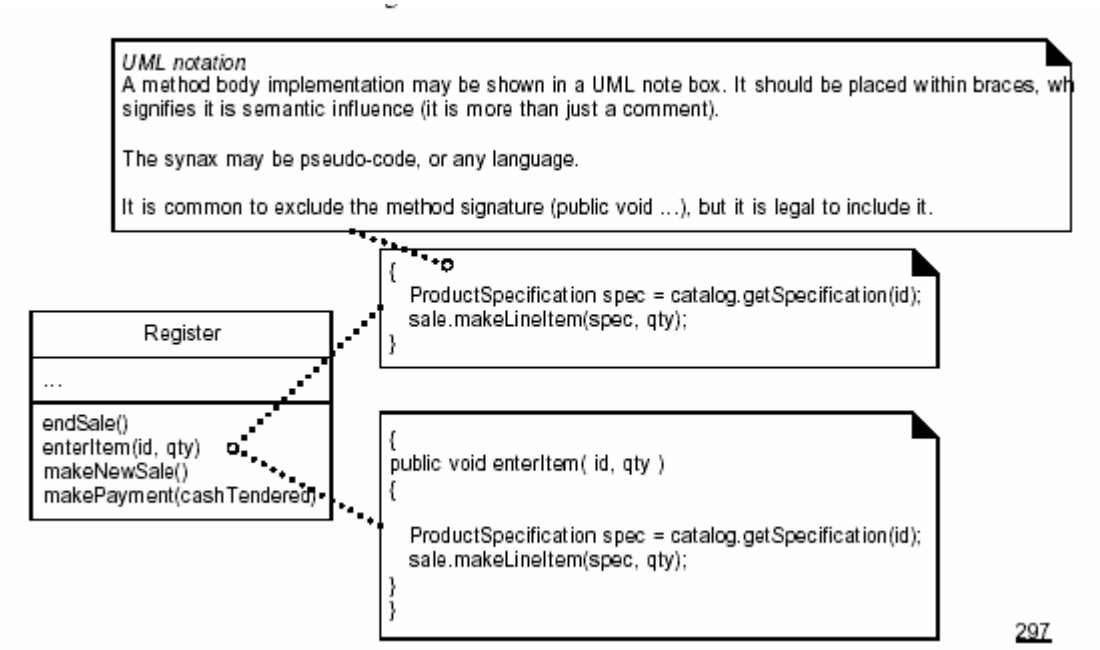


圖 19.14 方法本體的表示法。

第七節設計類別圖、畫設計類別圖與 CASE 工具

CASE 工具可以從原始程式碼中反向工程產生 DCDs。在第 35 章中會簡短討論開發流程情境與畫 DCDs 的一些實務經驗。

第八節UP 中的設計類別圖

DCDs 是使用案例實現的一部分，因此它也是 UP 中設計模型的一部分。

工作科目	工作成果 反覆→	初始階段 I1	詳述階段 E1..En	建構階段 C1..Cn	轉換階段 T1..T2
建立企業模型	**領域模型**		S		
需求	使用案例模型(SSD)	s	R		
	專案願景	s	R		

	輔助規格書	s	R		
	字彙表	s	R		
設計	**設計模型**		S	r	
實作	軟體架構文件		S		
	資料模型		s	r	
	實作模型		s	r	r
專案管理	軟體開發計畫	s	r	r	r
測試	測試模型		s	r	
環境	開發案例	s	r		

表 19.1 可能用到的 UP 工作成果以及產生它們的時間點。s － 初版；r － 修正版

開發階段

初始階段—我們通常會等到詳述階段才開始製作設計模型與使用案例實現，因爲在初始階段詳細的設計決策還不成熟。

詳述階段—在這個開發階段，我們會同時製作 DCDs 與使用案例實現；而且也只會先產生對架構影響性最大設計類別的 DCDs。

請注意，CASE 工具可以從原始程式碼中反向工程產生 DCDs。我們建議經常從原始程式碼中反向產生 DCDs，把系統的靜態結構用視覺方式呈現出來。

建構階段—我們會持續從原始程式碼中產生 DCDs，把系統的靜態結構用視覺方式呈現出來。

第九節UP 中跟設計類別圖相關的工作成果

圖 19.15 中是對 DCDs 發生顯著影響的相關工作成果。

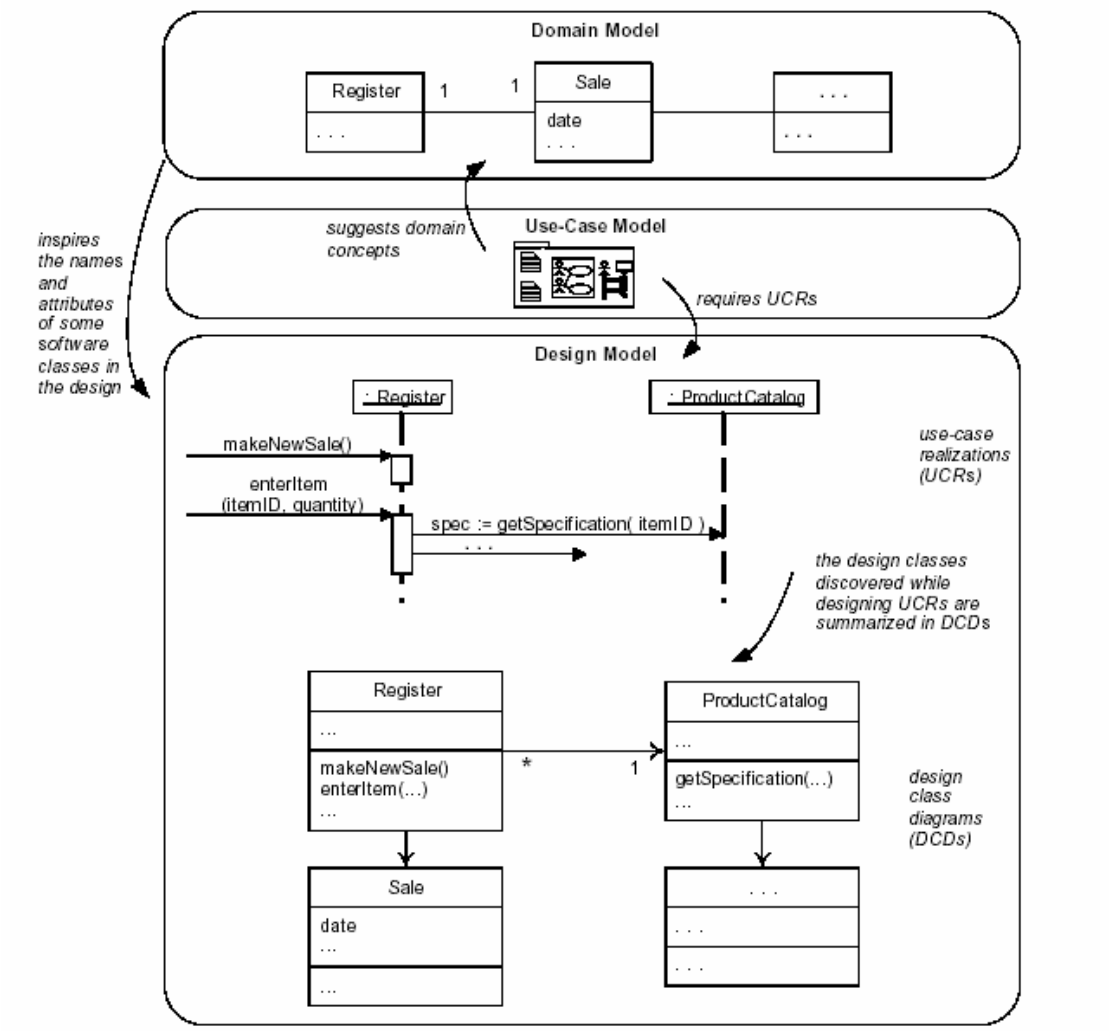


圖 19.15 UP 中工作成果之間可能發生的相互影響。

第二十章實作模型：把設計變成程式碼

程式碼中會有程式臭蟲存在；我必須證明程式是對的，只是測試過是不夠的。

— Donald Knuth

本章目標

- 把設計工作成果變成用物件導向程式語言寫成的程式碼。

簡介

完成 NextGen 應用程式目前反覆中所需的互動圖與 DCDs 之後，我們就有足夠細節可以產生領域層物件程式碼。

在設計開發工作所產生的 UML 工作成果 — 互動圖與 DCDs — 可以變成程式碼的輸入資訊。

UP 定義的實作模型中包含一些實作工作成果，例如原始程式碼、資料庫定義、JSP/XML/HTML 網頁等等。因此，本章所產生的程式碼屬於實作模型的一部分。

程式語言範例

我們用 Java 作爲寫範例時的程式語言，因爲這種程式語言已經被廣泛使用，大家也很熟悉它。然而，我們並沒有暗示一定要用 Java 或特別強調要用 Java；有許多程式語言例如 C#、Visual Basic、C++、Smalltalk、Python 等等都可以符合這些物件設計原則，也能寫出符合這個個案研究用的程式碼。

第一節程式設計與開發流程

之前的設計開發工作並非暗示說寫程式過程中不會產生雛型或進行設計工作；現在的開發工具提供我們很好的開發環境，可以很快探索不同的解決方案，而且邊設計變寫程式的做法通常是（很）有價值的。

然而，有些開發人員發現在寫程式之前，先思考一下、用視覺方式建立模型會很有幫助，這種做法特別適合某些習慣用視覺方式思考或用視覺化語言畫圖的人。

建議

對於一個長達兩星期的反覆來說，可以考慮在反覆一開始時，至少花半天時間用視覺方式做一些設計開發工作，然後再繼續寫程式。我們可以用一些簡單的「工

具」很快畫出圖，例如白板加上數位相機。你也可以用 UML 電腦輔助軟體工程（CASE）工具，這樣做也是很快速、簡單、便利與好用（譯註：不論是白板、CASE 工具還是輕量級方法論所提倡的兩人式開發，加強開發人員之間的溝通才是重要的課題。）

用物件導向程式語言 — 例如 Java 或 C# — 寫程式並不是物件導向分析與設計的一部分；程式是我們最後想要的東西。在 UP 設計模型中所產生的工作成果可以提供我們產生程式碼時的必要資訊。

在使用物件導向分析與設計與物件導向程式的情形下結合 UP，其優點是提供從需求到程式碼兩個端點之間的開發原則。我們可以用可追溯、有用的方式連結之前產生的工作成果與稍後產生的工作成果。這條開發之路不一定很平順，或者可以很簡單、機械式的一步一步做下去 — 其中存在太多變數。不過有開發原則存在至少可以讓我們嘗試與討論。

實作時發生的創新與變動

我們會在設計開發工作中做一些決策，也會有一些創新。在接下來的討論中，我們可以看到 — 在這個範例裡面 — 產生程式碼是比較機械式的轉換過程。

然而，一般來說，寫程式的開發工作不僅僅是沒什麼意義的產生程式碼開發步驟 — 它反而是相當有意義的。在真實情況下，我們設計時所產生的結果是不完整的初步結果；等到寫程式與測試時，會做很多變動，也會發現到詳細問題。

如果設計工作成果做的很好，那麼它就可以成爲具有彈性的核心基礎，寫程式遇到新問題時，用良好、強固的方式擴充它。因此，我們預期並規劃寫程式時會對設計工作成果做一些變動。

程式碼的變動與反覆式開發流程

反覆式、漸增式開發流程的優點之一就是前個反覆的結果可以回饋到下個反覆的開始時期（請參見圖 20.1。）因此，後續的分析與設計結果可以一直不斷修正並加強之前實作的結果。舉例來說，當我們從反覆 N 的設計得到反覆 N 的程式碼（由設計得到程式碼是必然的），以反覆 N 的實作爲基礎的最後設計結果可以作爲反覆 N+1 的分析與設計模型輸入來源。

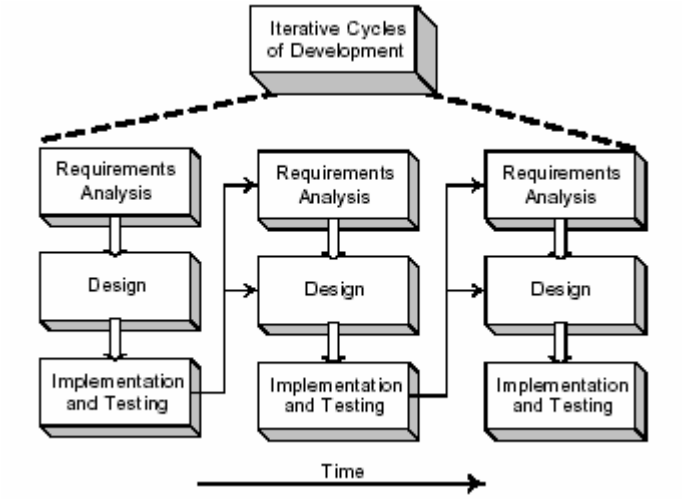


圖 20.1 某個反覆的實作會影響稍後的設計。

在每個反覆中初期會進行一個開發活動以便讓設計圖與實作同步；反覆 N 初期得到的圖跟反覆 N 最後程式碼是無法相符的，因此我們在擴充新設計結果之前必須讓兩者一致。

程式碼的變動、CASE 工具與反向工程

我們會想在設計時用半自動方式更新圖，反映出程式稍後所做的變動。在理想狀況下，我們可以用 CASE 工具讀取原始程式碼以自動更新圖，例如更新套件、類別與循序圖。這跟**反向工程**（reverse-engineering）（從原始碼產生圖的開發活動）有關。

第二節把設計變成程式碼

我們用物件導向程式語言進行實作時需要寫出下列的程式碼：

- 類別與介面的定義
- 方法的定義

下面的小節會討論如何用 Java 爲例子產生這些定義。

第三節從設計類別圖產生類別定義

我們至少會在 DCDs 中描述類別的類別名稱或介面名稱、超類別、方法的用法與簡單屬性。這些資訊已經夠我們在物件導向程式語言中產生類別基本定義。稍後的討論會探討介面與命名空間（或套件）等額外資訊。

定義出類別的方法與簡單屬性

以 *SalesLineItem* 爲例，我們可以用很直覺的方式從 DCD 中對應出用 Java 定義的基本屬性定義（簡單 Java 實例欄位）與方法的用法，請參見圖 20.2。

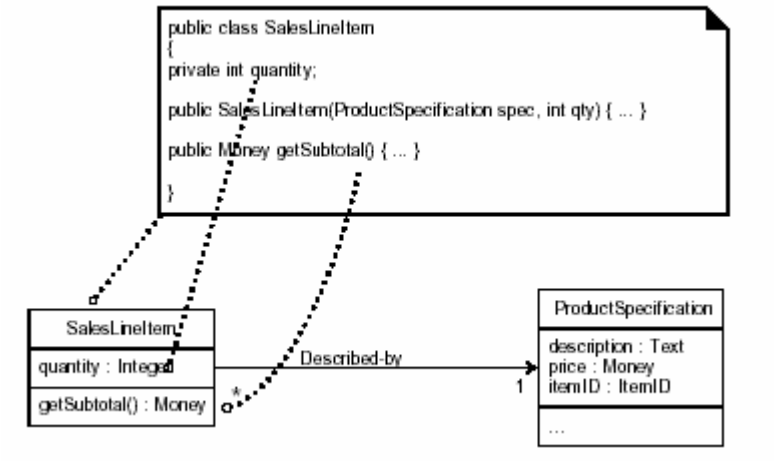


圖 20.2 用 Java 定義的 SalesLineItem。

請注意我們在原始碼中加入的 Java 建構子 *SalesLineItem(...)*。它是由 *enterItem* 互動圖傳給 *SalesLineItem* 的 *create*(spec,qty)訊息轉變而來的。換言之，Java 中的建構子需要加上 spec、qty 參數。我們通常不會在類別圖中宣告 *create* 方法，因爲每個類別中都會有這個方法，而且隨著目標程式語言不同，會有不同的解釋方式。

加入參考屬性

參考屬性（reference attribute）代表這個屬性會參考其它複雜物件，它的型態不像字串、數字等等基本型態。

類別圖中的關聯與可瀏覽性可以告訴我們類別中需要哪些參考屬性。

舉例來說，*SalesLineItem* 中會有連到 *ProductSpecification* 的關聯，前者對後者具有可瀏覽性。我們通常會把這樣的關聯解釋成類別 *SalesLineItem* 中的參考屬性，由它參考到 *ProductSpecification* 實例（請參見圖 20.3。）

在 Java 中，這代表我們會有一個參考到 *ProductSpecification* 的實例欄位。

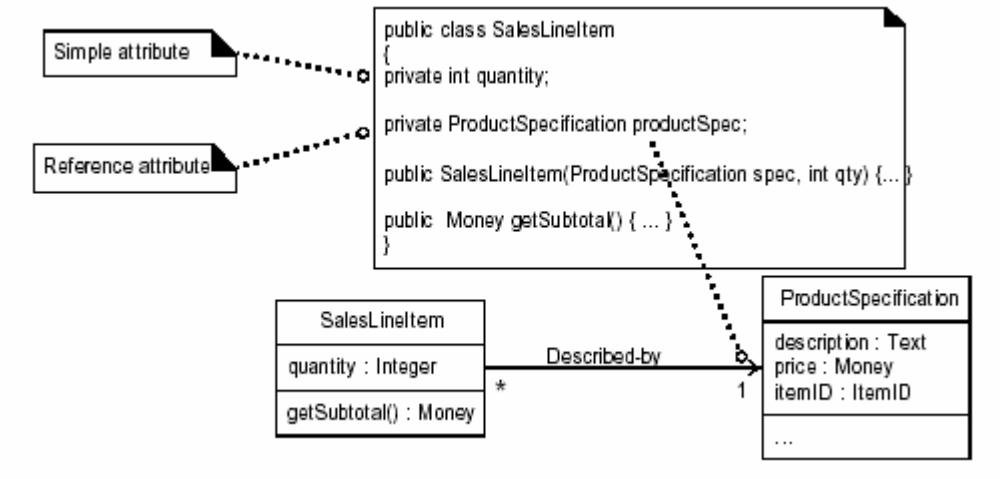


圖 20.3 加入參考屬性。

請注意，類別的參考屬性在類別圖通常是隱含不明確宣告的。

例如雖然我們在 *SalesLineItem* 的 Java 定義中加入指向 *ProductSpecification* 的實例欄位，在類別方塊圖形的屬性部分不會明確宣告這個欄位。圖中會有*建議性的*屬性可見性存在 — 用關聯與可瀏覽性標示出來，我們會在寫程式碼的開發階段把它轉變成明確的屬性定義。

參考屬性與角色名稱

接下來我們會探討靜態結構圖中角色名稱的概念。關聯的每個端點都是一個角色。簡單來說，所謂的**角色名稱**（role name）是用來識別角色的，而且我們通常可以用角色名稱提供跟角色本質有關的一些語意情境。

如果我們在類別圖中定義角色名稱，那麼就會在寫程式時，用這個角色名稱宣告參考屬性名稱，如圖 20.4 所示。

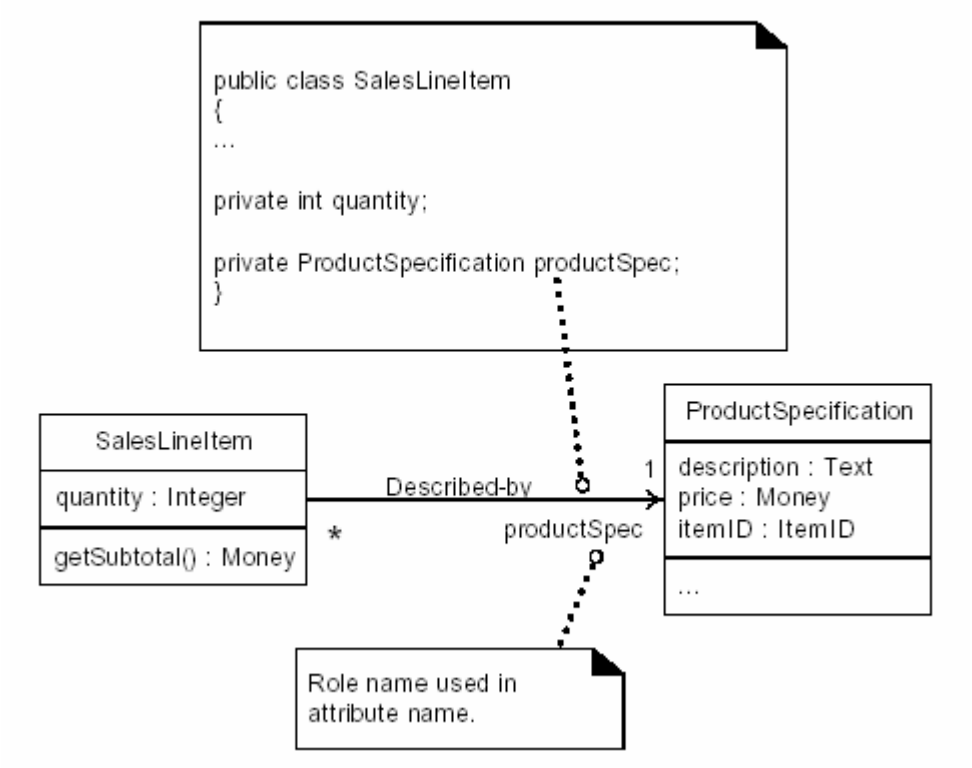


圖 20.4 角色名稱可能被用來當成實例變數名稱。

（不同程式語言的）屬性對應方式

從圖 20.5 中可以看出， *Sale* 類別在設計時期的屬性，在某些情況下需要考慮不同程式語言的對應方式。圖 20.5 中展現問題與解決方案。

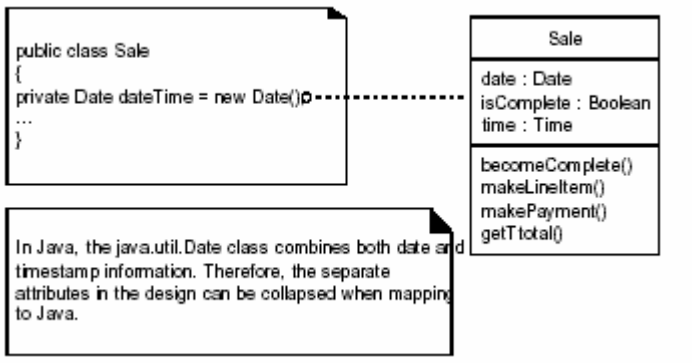


圖 20.5 用 Java 定義出日期與時間屬性。

第四節從互動圖找到方法

當我們畫互動圖時，回應某個方法的呼叫時會送出一些訊息。這些依序出現的訊息可以轉變成一連串用方法定義寫成的述句。在圖 20.6 中，*enterItem* 互動圖就用一連串的訊息展現 Java 定義的 *enterItem* 方法。

在這個例子中，我們以 *Register* 爲例。圖 20.7 是它的 Java 定義。

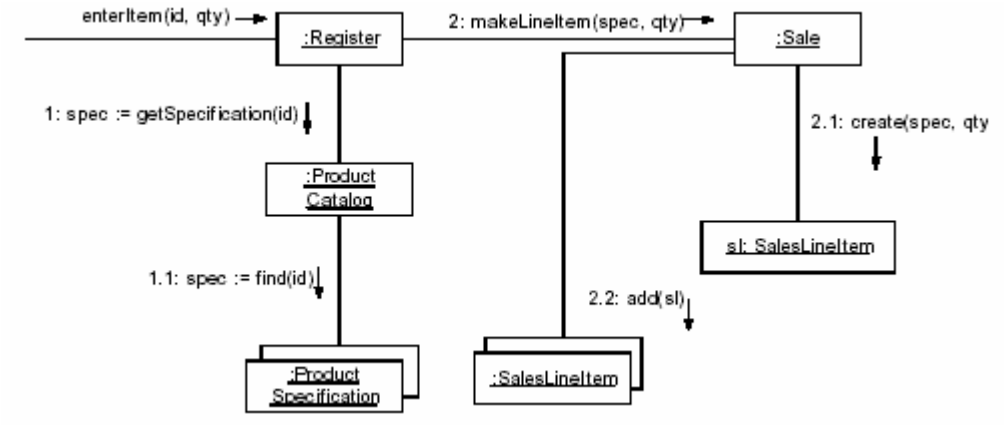


圖 20.6 enterItem 互動圖。

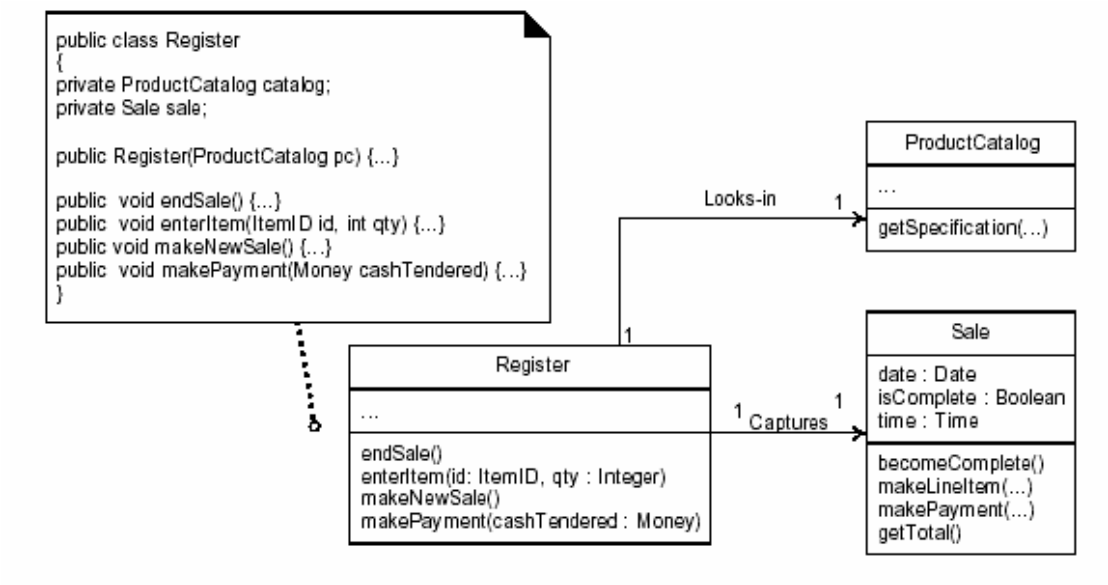


圖 20.7 Register 類別。

Register – enterItem 方法

enterItem 訊息會被傳給 *Register* 實例；因此，類別 *Register* 中會定義出 *enterItem* 方法。

public void enterItem(ItemID itemID, int qty)

訊息 1：傳送 *getSpecification* 訊息給 *ProductCatalog* 以讀取 *ProductSpecification*。

productSpecification spec = catalog.getSpecification(itemID);

訊息 2：傳送 *makeLineItem* 訊息給 *Sale*。

sale.makeLineItem(spec, qty);

總而言之，互動圖中每個依序出現的訊息都可轉變成用 Java 方法寫成的述句。

圖 20.8 中秀出完整的 *enterItem* 方法定義以及相對應的互動圖。

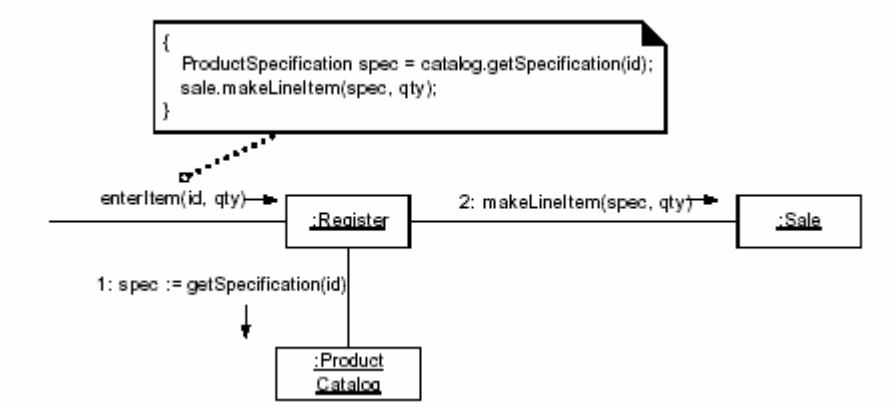


圖 20.8 enterItem 方法。

第五節程式碼中的容器類別／多重物件類別

我們常常需要讓某個物件可以看得見一群其它物件；類別圖中多重性的值可以告訴我們是否有這樣的需要存在。例如我們可以從圖 20.9 中看出 *Sale* 必須看得到一群 *SalesLineItem* 實例。

在物件導向程式語言中，我們實作這些關係時通常會靠中介的容器或多重物件達成，我們會在關聯多重性爲 1 的類別中定義一個參考屬性，由它指向容器／多重物件實例，裡面存放關聯多重性爲多個的類別。

舉例來說，Java 函式庫中有一些多重物件類別，例如 *ArrayList* 與 *HashMap* 分別會實作 *List* 與 *Map* 介面。利用 *ArrayList*，*Sale* 就可以定義一個屬性，裡面維護存放一群 *SalesLineItem* 實例的有序串列。

多重物件的選擇當然會受到需求的影響；以主鍵爲基礎的查詢方式就需要用 *Map*，而會不斷長大的有序串列就需要用 *List* 等等。

第六節處理異常情況與錯誤情況

到目前爲止，我們在尋找解決方案的開發過程中一直忽略異常情況的處理方式。因爲我們之前一直把焦點放在指派責任與物件設計的基本問題上。然而，在應用程式的開發過程中，進行設計與實作開發工作時考慮異常狀況處理方式是很明智的做法。

簡單來說，在 UML 中我們在互動圖中用非同步訊息展現例外情況。第 33 章會有更深入的討論。

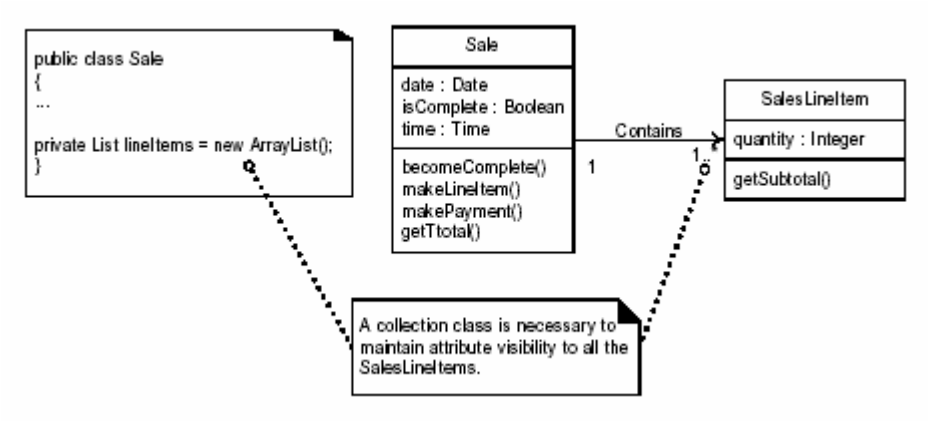


圖 20.9 加入多重物件。

第七節定義 Sale 類別的 makeLineItem 方法

我們也可以爲類別 *Sale* 的 *makeLineItem* 方法畫出合作圖，然後看著它寫出方法定義。圖 20.10 所展示的是整個 *enterItem* 互動圖的其中一部分，裡面也秀出 Java 方法定義。

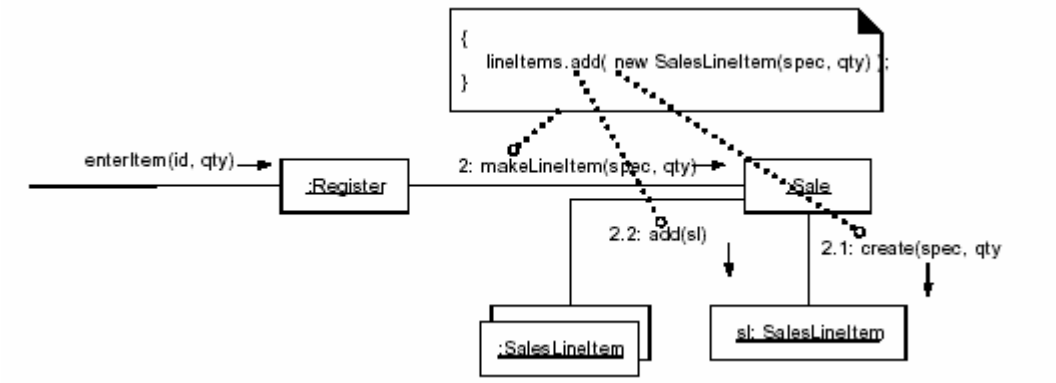


圖 20.10 Sale—makeLineItem 方法。

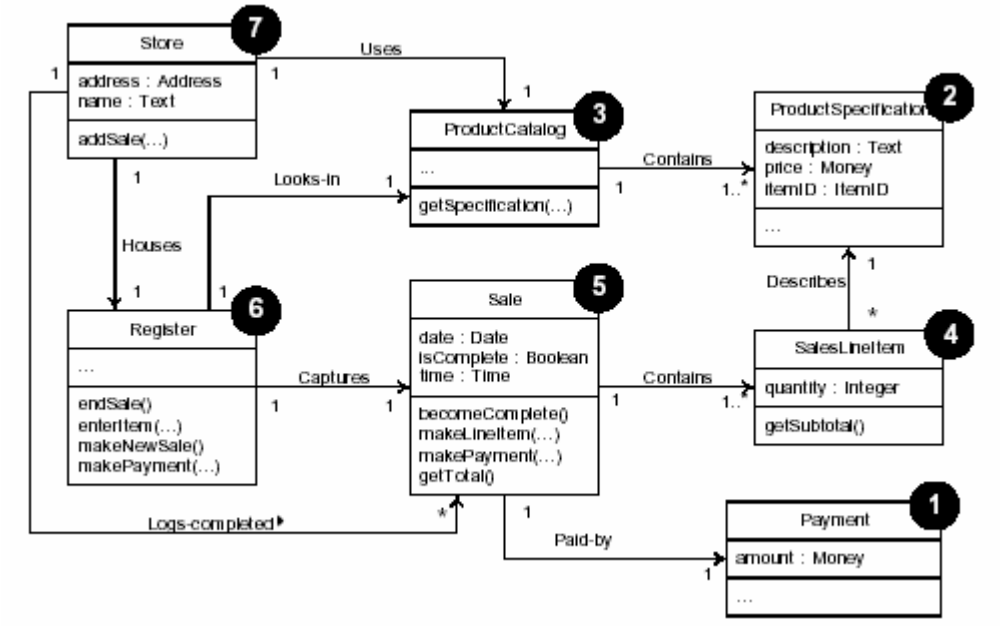


圖 20.11 實作並測試類別的可能順序。

第八節實作順序

我們實作類別時（在理想狀況下，類別都會經過完整的單元測試）應該先實作耦合力最少的類別，最後才實作耦合力最多的類別（如圖 20.11 所示。）例如我們應該最先實作的是 *Payment* 或 *ProductSpecification*；接下來實作跟前面有關的類別 — *ProductCatalog* 或 *SalesLineItem*。

第九節先寫測試程式的程式設計方式

終極流程（extreme programming，XP）中提倡一種很棒的實務經驗，我們也可以把它用在 UP 中（事實上大部分的 XP 實務經驗都可以應用在 UP 中）就是：**先寫測試程式的程式設計方式**（test-first programming）。在這個實務經驗中，我們在寫程式碼之前要先寫出單元測試的程式碼，而且開發人員要寫出*所有*產品程式碼（production code）的單元測試程式碼。這個實務經驗的基本模式是先寫出一些測試程式碼，然後再寫出一些產品程式碼，後者通過測試之後，再寫出更多的測試程式碼，依此類推（譯註：這種方式跟譯者原先寫程式的習慣不同，不過後來往往發現沒有經過完整測試的程式碼，所需花費的除錯時間往往比寫程式時間更多。）

優點包括：

- **確保單元測試一定會被寫出來** — 如果稍後才思考單元測試的話，人性（至少是程式設計師的本性）總是會避免寫出單元測試。
- **程式設計師的滿足感** — 如果開發人員寫出產品程式碼、用非正式方式除

錯，稍後再思考加入單元測試，那麼開發人員的滿足感不會那麼高。然而，如果先寫出測試程式碼的話，當我們產生產品程式碼時，就可以修正程式碼以通過測試，這樣會有完成某個東西 — 通過測試 — 的滿足感。我們不能忽略開發人員的心理層面 — 寫程式是由人類花精神做的事。

- **釐清介面與行爲** — 通常在寫出程式之前，我們不會完全知道確實的公開介面與行爲。如果先寫出單元測試的話，我們就可以更清楚類別的設計方式。
- **可證實的驗證方式** — 很明顯地，有數百、數千個單元測試，對產品程式碼的正確性可以產生某種有意義的驗證效果。
- **對改變程式碼的信心** — 如果採用先寫測試程式的程式設計方式，我們將會有數百、數千個單元測試，而且每個產品類別也都會有相對應的單元測試類別。當開發人員需要改變現有的程式碼時 — 可能是他們自己寫的或別人寫的 — 可以執行一個單元測試輯（unit test suite）以了解程式變動有是否會造成錯誤。

有一個很流行、簡單、免費的 Java 單元測試框架：JUnit（www.junit.org）。假設我們用 JUnit 與先寫測試程式的方式開發 *Sale* 類別。在寫 *Sale* 的程式之前，我們會在 *SaleTest* 類別中先寫出單元測試用方法：

1. 建立一筆新銷售。
2. 把銷售明細項目加入銷售中。
3. 算出銷售總金額，並且用預期值驗證它。

例如：

```
public class SaleTest extends TestCase {
   // ...

  public void testTotal() {
    // set up the test
    Money total = new Money( 7.5 );
    Money price = new Money( 2.5 );
    ItemID id = new ItemID( 1 );
    ProductSpecification spec;
    spec = new ProductSpecification( id, price, "product 1" );
    Sale sale = new Sale();

    // add the items
    sale.makeLineItem( spec, 1 );
    sale.makeLineItem( spec, 2 );

    // verify the total is 7.5


  assertEquals( sale.getTotal(), total); } }
```

只有寫出 *SaleTest* 類別之後，我們才會寫出可通過測試的 *Sale* 類別。然而，我們不用一下子寫出全部的單元測試用方法。開發人員只要先寫出一個單元測試用方法，然後寫出滿足測試的產品程式碼，再寫出另一個單元測試用方法等等。

第十節總結：把設計變成程式碼

把 DCDs 變成類別定義、互動圖變成方法的轉換過程是很直覺的。雖然在寫程式時還是有很大的空間要做決策、變動設計與探索需求，不過寫程式之前還是要先

考慮一些影響範圍比較大的設計概念。

第十一節第一版程式

本節用 Java 展現我們在這個反覆中的領域物件層程式的解決方案。這裡產生的大部分程式都是根據之前所討論把設計變成程式碼的原則，從設計開發工作的設計類別圖與互動圖來的。

列出這些程式碼的主要重點是展現我們可以從設計工作成果轉變成程式碼基礎。這裡的程式碼只是一個簡單的例子；它不是一個強固、開發完整的 Java 程式，裡面沒有包含同步、例外情況處理等等。

Class Payment

```
public class Payment {
   private Money amount;
   public Payment( Money cashTendered ){ amount = cashTendered; }
   public Money getAmount() { return amount; } }
```

Class ProductCatalog

```
public class ProductCatalog {
   private Map productSpecifications = new HashMap();

    public ProductCatalog() {
        // sample data
        ItemID id1 = new ItemID( 100 );
        ItemID id2 = new ItemID( 200 );
        Money price = new Money( 3 );

        ProductSpecification ps;
        ps = new ProductSpecification( id1, price, "product 1" );
        productSpecifications.put( id1, ps );
    ps = new ProductSpecification( id2, price, "product 2" );
    ProductSpecifications.put( id2, ps ); }

    public ProductSpecification getSpecification( ItemID id ) {
        return (ProductSpecification)productSpecifications.get( id );
    }
 }
```

Class Register

```
public class Register {
   private ProductCatalog catalog;
   private Sale sale;

    public Register( ProductCatalog catalog ) {
        this.catalog = catalog; }

    public void endSale() {
        sale.becomeComplete();
     }

    public void enterItem( ItemID id, int quantity ) {
        ProductSpecification spec = catalog.getSpecification( id );

        sale.makeLineItem( spec, quantity ); }

    public void makeNewSale() {
        sale = new Sale(); }

    public void makePayment( Money cashTendered ) {
        sale.makePayment( cashTendered ); }

}
```

Class ProductSpecification

```
public class ProductSpecification {
   private ItemID id;
   private Money price;
   private String description;

   public ProductSpecification
       ( ItemID id. Money price. String description ) {
      this.id = id;
      this.price = price;
      this.description = description; }

   public ItemID getItemlDO { return id;}

   public Money getPrice() { return price; }

   public String getDescription() { return description; }
}
```

Class Sale

```
public class Sale
{
   private List lineltems = new ArrayListO;
   private Date date = new Date();
   private boolean isComplete = false;
   private Payment payment;

   public Money getBalanceO {
      return payment.getAmount().minus( getTotal() ); }

   public void becomeComplete() { isComplete = true; }

   public boolean isComplete() { return isComplete; }

   public void makeLineltem
       ( ProductSpecification spec, int quantity ) {
      lineltems.add( new SalesLineltem( spec, quantity ) ); }

   public Money getTotal()
   {
      Money total = new MoneyO;
      Iterator i = lineltems.iterator( ) ;
      while ( i.hasNextO )
      {
       SalesLineltem sli = (SalesLineltem) i.nextO;
       total.add( sli.getSubtotal() );
      }
      return total; }


   public void makePayment( Money cashTendered )
   {
   payment = new Payment( cashTendered ); } }
```

Class SalesLineItem

```
public class SalesLineltem {
   private int quantity;
   private ProductSpecification productSpec;

   public SalesLineltem (ProductSpecification spec, int quantity )
   {
      this.productSpec = spec;
      this.quantity = quantity; }

   public Money getSubtotal() {
      return productSpec.getPrice().times( quantity );
   } }
```

Class Store

```
public class Store
{
   private ProductCatalog catalog = new ProductCatalog();
   private Register register = new Register( catalog );

   public Register getRegister() { return register; } }
```

字彙表

英文	中文	解釋
abstract class	抽象類別	這種類別只能做爲其它類別的超類別；不可能會有抽象類別的物件存在，我們只可能產生它的子類別實例。
abstraction	抽象化或抽象概念	把焦點放在類似事物的必要性質或共通動作上（譯註：抽象化。）此外，它也可以代表某個事物抽象化之後所得到的必要特徵（譯註：抽象概念。）
active object	主動物件	擁有控制執行緒的物件。
aggregation	聚合關係	關聯的外顯屬性，它代表整體 — 零件關係，通常也代表生命時期中的包含關係。
analysis	分析	對領域所做的調查工作，結果是領域靜態與動態特徵的一些模型。它把重點放在了解「要做什麼」而不是「如何做」。
architecture	架構	非正式來說，它是對系統的組織方式、架構設計動機與結構的一些說明。架構可以從許多不同層面來觀察它跟所開發軟體系統之間的關係，從實體的硬體架構到應用程式框架的邏輯架構都有可能。
assocaition	關聯	描述兩個類別的物件之間所存在的一整組相關繫結（link）。
attribute	（內部）屬性	類別中命名過的特徵或性質。
class	類別	在 UML 中，「我們用它描述一組物件，這些物件會有共同的屬性、操作、方法、關係與行爲」【RJB99】。我們可能用它代表軟體元素（譯註：軟體類別）或概念性元素（譯註：概念性類別。）
class hierarchy	類別階層	我們用它描述類別之間的一些繼承關係。
class method	類別（生存空間）方法	定義類別本身的行爲，而不是類別實例的行爲。
classification	類別化	類別化定義類別跟它的實例之間的關

		係。類別化這個對應關係可以讓我們知道類別的外顯概念（譯註：知道類別的實例爲何。）
collaboration	合作（關係）	兩個或多個物件之間透過主從關係（譯註：一個物件負責發出訊息，另一個物件負責接收訊息）互動以提供服務。
composition	合成關係	定義類別的實例是由其它物件構成的。
concept	概念	某些概念或事物的分類結果。在本書中，我們把它用在真實世界的事物上而不是軟體實體上。概念的內涵意義是它的屬性、操作與語意，外顯意義則是一組物件實例，這些物件都是這個概念的成員之一（譯註：可以把概念視爲一個集合，而物件視爲屬於這個集合的元素。）我們通常把「概念」當成領域物件的同義字。
concrete class	具體類別	可產生實例的類別。
constraint	限制	對某種元素的約束或條件。
constructor	建構子	在 C++或 Java 中，當我們產生某個類別的實例時，會呼叫這個特殊的方法。建構子中通常會執行一些初始化動作。
container class	容器類別	我們設計這種類別，讓它能夠同時容納並操作一組物件。
contract	合約	我們用它定義操作或方法的責任或事後條件（譯註：事先條件則是呼叫操作者的責任。）也可以用它描述跟某個介面相關的一組條件。
coupling	耦合力	它代表某些元素（例如類別、套件與子系統）之間的相依性，產生相依性的原因通常都是因爲元素之間有合作關係。
delegation	委託	這個概念說明物件回應某個訊息時會對另一個物件發出訊息。第一個物件把責任委託給第二個物件。
derivation	衍生（類別）過程	參考一個已存在類別、加入新屬性與方法以產生新類別的過程。這個已存在類別是超類別；新的類別則是子類別或衍生類別。
design	設計	用分析結果產生實作系統時會用到的規

		格。裡面描述系統是如何運作的。
domain	領域	一個正式的範圍，裡面定義我們有興趣的特定主題或範圍。
encapsulation	封裝	我們用這種機制來隱藏某個元素（例如物件或子系統）裡面的資料、內部結構與實作細節。跟物件之間的所有互動都是透過操作的公開介面達成的。
event	事件	出現值得注意的事。
extension	外顯概念	代表某個概念的一組物件，這些物件是這個概念的例子或實例。
framework	框架	一組互相合作的抽象類別或具體類別，我們以它們爲範本解決一組相關問題。針對應用程式特定行爲，我們通常會已子類別化擴充（譯註：以子類別化方式擴充的框架稱爲白箱式框架，如果應用程式的特定行爲是透過實作某些介面的元件達成，就是黑箱式框架。後者通常比前者成熟。）
generalization	一般化關係	找出概念之間的共通性、定義超類別（一般化概念）與子類別（特殊化概念）關係的開發活動。這是產生概念分類系統的一種方式，產生之後通常會以類別階層展現一般化關係。概念性子類別必須符合概念性超類別的內涵概念與外顯概念。
inheritance	繼承	物件導向程式設計語言的一種特性，根據這種特性，我們可以從比較一般性的超類別中特殊化變成新的類別。子類別會自動擁有從超類別來的屬性與方法定義。
instance	（類別）實例	屬於某個類別的個別成員。在 UML 中，我們把它叫做物件。
instance method	實例方法	生存空間爲實例的方法。要呼叫實例方法時必須傳送訊息給實例。
instance variable	實例變數	實例的屬性，就像我們在 Java 與 Smalltalk 中的用法一樣。
instantiation	實例化	由類別產生實例的過程。
intension	內涵（概念）	某個概念的定義，包括屬性、操作等等。

interface	介面	一組公開的操作用法。
link	繫結	兩個物件之間的連結；關聯的實例（譯註：關聯代表一種類別而繫結代表這個類別的實例。）
message	訊息	物件之間溝通的機制；通常代表執行某個方法的請求。
metamodel	超模型	定義其它模型的模型。UML 的超模型裡面就會定義 UML 的元素型態（例如有行爲能力者【classifier】。）
method	（物件或類別的）方法	在 UML 中，類別某個操作的特定實作或演算法。非正式來說，它代表回應訊息時執行的軟體程序。
model	模型	描述某個主題範圍內的靜態與／或動態特徵。我們會透過一些觀點（通常是圖或文字）來描繪這些特徵。
multiplicity	多重性	在某個關聯中，某一個端點被允許參與的物件數目。
object	物件	在 UML 中，它代表某個類別的實例，裡面會封裝狀態與行爲。非正式來說，它是某種東西的實例。
object identity	物件識別字	透過這個特性，我們可以讓某個物件的存在跟物件裡面的任何資料無關。
object-oriented analysis	物件導向分析	用領域概念（例如概念性類別、關聯與狀態變動）呈現問題領域或系統的調查結果。
object-oriented design	物件導向設計	用軟體物件（例如類別、屬性、方法與合作關係）呈現邏輯軟體解決方案中的規格。
object-oriented programming language	物件導向程式語言	支援封裝、繼承與多型等概念的程式語言。
OID	物件識別碼	=Object Identifier。
operation	操作	在 UML 中，操作代表「物件被呼叫的規格，執行時會轉換物件內部狀態或查詢物件屬性」【RJB99】。每個操作都會有一種用法，用法中標示操作名稱與參數，而且要經由訊息呼叫。方法則是操作的實作，裡面寫出特定的演算法。

pattern	樣式	樣式代表命名過的問題、相關解決方案以及使用解決方案時如何才能把解決方案應用在新環境中。
persistence	永續（儲存）	持續儲存物件狀態。
persistent object	永續物件	物件可以在產生它的執行程序或執行緒中存活（譯註：一般來說，我們會認爲產生永續物件的執行程序或執行緒消失後，會把它儲存在次要儲存體中而不會消失。）永續物件會一直存在直到它被明確刪除爲止。
polymorphic operation	多型操作	由兩個或多個不同類別所實作的相同操作。
postcondition	事後條件	操作執行完後必須成立的限制。
precondition	事先條件	要求操作在執行前必須成立的限制。
private	私有（屬性、方法）	一種生存空間機制，我們用它限制對類別成員的存取動作，讓其它物件看不到它們。通常會把所有屬性設成私有的，也會把某些方法設成私有的。
public	公開（屬性、方法）	一種生存空間機制，我們用它讓其它物件可以存取到這些成員。通常會把某些方法設成公開的，而不會把屬性設成公開的，因爲這樣做違反封裝性。
pure data values	純資料值	對這些資料值來說，區分個別的唯一個體是無意義的，例如數字、布林值與字串等等。
qualified association	限定關聯	在這種關聯上，加上限定者的值之後，原本看起來相同的成員會被視爲不同成員（譯註：例如在銷售與銷售明細之間的限定關聯上，從銷售往銷售明細看時，原本是無法區分出不同銷售明細的，不過加上限定者 — 產品代碼 — 之後，我們就可以找出屬於不同產品代碼的銷售明細；不過，我們又不能以產品代碼當做銷售明細的主鍵，因爲不同銷售中可能會有相同產品代碼的銷售明細。）
receiver	（訊息）接收者	被傳送訊息的物件。
recursive	遞迴式關聯	這種關聯的來源類別與目的類別都是同

association		一個物件類別。
responsibility	（物件）責任	由某個元素（例如類別或子系統）所提供、已知或會做的某種或一整組服務；責任中可能包含某個元素的一個或多個目的或應盡義務。
role	（關聯某一端點的）角色	它代表關聯上命名過的一個端點，名稱是用來標示端點目的。
state	狀態	物件在不同事件之間的情況。
state transition	狀態轉換	物件的狀態變動；由事件所引發的某種東西。
subclass	子類別	其它類別（超類別）的特殊化結果。子類別會繼承超類別的屬性與方法。
subtype	子型態	概念性子類別。其它型態（超型態）的特殊化結果，它必須符合超型態的內涵意義與外顯意義。
superclass	超類別	其它類別會從這個類別繼承屬性與方法。
supertype	超型態	概念性超類別。在一般化 — 特殊化關係中，代表比較一般性的型態；擁有子型態的物件。
transition	轉換	如果發生事件且符合條件情況時，橫跨在不同狀態之間的關係。
visibility	可見性	可以看見某個物件或參考某個物件的能力。