# 序

領域驅動設計（domain-driven design）提供了一組協作方法的實踐，用於從業務觀點——即領域（domain）來建立軟體，以及作為您目標的領域問題。它最初是由 Eric Evans 於 2003 年所創造，並出版了在 DDD 社群中廣為人知的「藍皮書」，書名是《*Domain-Driven Design: Tackling Complexity in the Heart of Software*》。

雖然解決複雜度並提供通往清晰的道路是領域驅動設計的目標，但仍有許多很棒的想法甚至可以應用在較不複雜的軟體專案。DDD 提醒我們，軟體開發人員並不是唯一參與建立軟體的人，軟體是為了領域專家（domain experts）而建立的，他們為正被解決的問題帶來關鍵性的理解。當我們首先應用「戰略設計」（strategic design）來理解業務問題（又稱領域）並將問題分解為更小、可解決、相互關聯的問題時，我們會在整個創造的階段建立合夥關係（partnership）。與領域專家的合作也促使我們使用領域的語言進行交流，而不是強迫業務方面的人學習軟體的技術語言。

基於 DDD 專案的第二個階段是「戰術設計」（tactical design），我們將戰略設計的發現轉變為軟體的架構和實作。同樣的，DDD 為了組織這些領域並避免更大程度的複雜度提供了引導和模式。即使領域專家查看軟體團隊建立的程式碼，領域專家也將識別出他們領域的語言，戰術設計使得與領域專家的合夥關係延續下去。

自從「藍皮書」出版後的多年來，不僅許多組織受益於這些想法，而且一個由經驗豐富的 DDD 從業者所組成的社群也獲得發展。DDD 的協作本質促使了這個社群分享他們的經驗和觀點，並且創建工具來幫助團隊擁抱這些想法並從中受益。在 2019 年探討 DDD 的主題演講中，Eric Evans 鼓勵社群繼續發展 DDD——不僅是它的實踐，而且要尋找方法來更有效地分享其想法。

而這讓我明白為什麼我如此熱愛領域驅動設計學習手冊。透過 Vlad 的會議演講和其他著作，我已經是他的粉絲了。他作為一位從事於一些相當複雜的專案，並一直慷慨分享這些知識的 DDD 從業者，他擁有許多得來不易的經驗。在這本書中，他用獨特的方式述說 DDD 的「故事」（不是它的歷史，而是它的概念），為學習提供了很好的觀點。雖然這本書是針對新手的，但我作為一位撰文並談論 DDD、長期的 DDD 從業者，我發現我仍從他的觀點中學到很多。在這本書出版之前，我就迫不急待地在我 Pluralsight 上的 DDD 基礎課程中參考他的書，而且已經在與委託人的對話中分享了一些這種觀點。

入門 DDD 時可能會令人困惑，正如我們使用 DDD 來降低專案的複雜度一樣，Vlad 以降低主題本身複雜度的方式來呈現 DDD。他所做的不僅僅是解釋 DDD 的原理，這本書的後半部分享了一些從 DDD 發展而來的重要實踐，例如事件風暴（EventStorming）解決了發展業務重點或組織的問題，以及這可能會如何影響軟體，還討論了如何保持 DDD 和微服務（mircoservices）的一致，以及您如何能把它和許多知名的軟體模式做整合。我認為領域驅動設計學習手冊對於新手來說是一本很好的 DDD 入門書，對於有經驗的從業者來說也是一本非常值得一讀的書。

*—Julie Lerman*
*軟體教練、O'Reilly 作者，*
*以及 Serial DDD 倡導者*

# 前言

我清楚記得我開始第一份真正的軟體工程工作的那一天，我欣喜若狂卻又感到害怕。在我高中時期為當地企業解決軟體問題之後，我便渴望成為一名「真正的程式設計師」並為該國最大的外包公司之一撰寫一些程式。

我在那裡的第一天，我的新同事告訴我要做什麼。在設定好公司電子郵件並瀏覽了時間追蹤系統之後，我們終於來到了有趣的東西：公司的程式撰寫風格和標準。有人告訴我「在這裡，我們總是撰寫精心設計的程式碼並使用分層架構（layered architecture）」，我們逐一瀏覽了三個層——資料存取層（data access layer）、業務邏輯層（business logic layer）和展示層（presentation layer）——的定義，然後討論了解決這些層需求的技術和框架。當時，公認的資料儲存解決方案是 Microsoft SQL Server 2000，它是使用 ADO.NET 在資料存取層做整合的，展示層因用於桌面應用程式的 WinForms 或是用於 web 的 ASP.NET WebForms 而很棒，我們在這兩層上花了相當長的時間，所以當業務邏輯層沒有得到任何關注時，我感到很困惑：

「但業務邏輯層呢？」

「那是很直接的，這裡就是你實作業務邏輯的地方。」

「但什麼是業務邏輯呢？」

「哦，業務邏輯就是為了實作需求，你所需要的所有迴圈和『if-else』陳述。」

那天，我開始了我的旅程以了解業務邏輯到底是什麼，以及應該如何在設計良好的程式碼中實作它。我花了三年多的時間才終於找到答案。

答案就在 Eric Evans 的開創性著作《*Domain-Driven Design: Tackling Complexity in the Heart of Software*》中。結果證明了我沒有錯，業務邏輯確實很重要：它是軟體的核心！然而不幸的是，我又花了三年的時間才理解 Eric 所分享的智慧。這本書非常先進，英語是我的第三語言這一個事實並沒有幫助。

不過，最終一切都明朗了，我接受了領域驅動設計（domain-driven design，DDD）的方法論，我明白了 DDD 的原則和模式、建模和實作業務邏輯的複雜度，以及如何應對我正建立的軟體核心複雜度。儘管有阻礙，但這絕對是值得的，踏入領域驅動設計對我來說是一次改變生涯的經歷。

## 為何我會寫這本書

在過去十年裡，我向不同公司的同事介紹了領域驅動設計、進行了實體課程，並講授了線上課程。從教學的角度不僅幫助我深化知識，還讓我完善了我說明領域驅動設計原則和模式的方式。

正如經常發生的那樣，教比學更具有挑戰性。我是 Eliyahu M. Goldratt（*https://oreil.ly/ZZdXf*）工作和教學的超級粉絲。Eliyahu 曾經說過，即使是最複雜的系統，如果從正確的角度來看，本質上也是簡單的。在我教 DDD 的這些年裡，我一直在尋找一種能夠揭示領域驅動設計本質上簡單的方法模型。

這本書是我努力的結果。它的目標是使領域驅動設計大眾化；使它更容易理解也更易於使用。我相信 DDD 的方法絕對是無價的，尤其是在設計現代軟體系統時。這本書將為您提供足夠的工具，讓您開始在日常工作中應用領域驅動設計。

## 誰應該讀這本書

我相信領域驅動設計原則和模式的知識對各級軟體工程師都有用：初級（junior）、資深（senior）、主任（staff）和首席（principal）。DDD 不僅提供了用於建模和有效實作軟體的工具和技術，它還闡明了軟體工程中一個經常被忽視的面向：上下文（context）。具備系統的業務問題知識之後，您將更有效地選擇合適的解決方案，一個不是設計不足（under-engineered）或過度設計（over-engineered），而是可以解決業務需求和目標的解決方案。

領域驅動設計對於軟體架構師（software architects）來說更為重要，對於有抱負的軟體架構師來說更是如此。它的戰術設計決策工具將幫助您將大型系統分解為元件（components）——服務、微服務（microservices）或子系統（subsystems）——並設計如何讓元件互相整合以形成一套系統。

最後，在本書中，我們不僅會討論如何設計軟體，還會討論如何隨著業務上下文的變化共同發展設計。軟體工程這一關鍵面向將幫助您隨著時間推移保持系統設計的「形態」，並防止它退化成一個大泥球（big ball of mud）。

## 本書導覽

本書分為四個部分：戰略設計（strategic design）、戰術設計（tactical design）、實踐中的 DDD，以及 DDD 與其他方法和模式的關係。在第一部分，我們介紹用來制定大規模軟體設計決策的工具和技術。在第二部分，我們關注程式碼：實作系統業務邏輯的不同方式。第三部分討論了在實際專案中應用 DDD 的技術和策略。第四部分繼續討論領域驅動設計，但這次是在其他方法和模式的背景之下。

以下是每章內容之簡短摘要：

- 第 1 章建立軟體工程專案的背景：業務領域、它的目標，以及軟體打算如何去支持它們。
- 第 2 章介紹「統一語言」（ubiquitous language）的概念：用於有效溝通和知識分享的領域驅動設計實踐。
- 第 3 章討論如何應對業務領域的複雜度，並設計系統的高階（high-level）架構元件：限界上下文（bounded contexts）。
- 第 4 章探討在限界上下文之間組織溝通和整合的不同模式。
- 第 5 章開始討論業務邏輯的實作模式，包含兩個解決簡單業務邏輯案例的模式。
- 第 6 章進一步從簡單到複雜的業務邏輯，並介紹了解決其複雜度的領域模型（domain model）模式。
- 第 7 章增加時間的角度，並介紹了一種更進階的方式來建模和實作業務邏輯：事件源領域模型（event-sourced domain model）。
- 第 8 章將重點轉移到更高的層次，並描述了建構元件的三種架構模式。
- 第 9 章提供編排系統元件工作所需要的模式。

- 第 10 章將前面章節中討論過的模式串聯在一起，形成一些簡單的經驗法則，簡化了制定設計決策的流程。
- 第 11 章從時間的角度探討軟體設計，以及它應該如何在其壽命中改變和發展。
- 第 12 章介紹事件風暴（EventStorming）：一個用於有效分享知識、建立共同的理解和設計軟體的低技術研討會。
- 第 13 章介紹在將領域驅動設計導入棕地（brownfield）專案時可能面臨的困難。
- 第 14 章討論微服務的架構風格和領域驅動設計之間的關係：它們的差異和互補之處。
- 第 15 章在事件驅動（event-driven）架構的背景下探討領域驅動的設計模式和工具。
- 第 16 章的討論從營運系統（operational systems）轉移到分析資料管理系統（analytical data management systems），並討論領域驅動設計和資料網格（data mesh）架構之間的相互作用。

為了加強學習，所有章節都會以一些練習題作為結束，一些問題使用虛構的公司「WolfDesk」來展示領域驅動設計的各個面向。請閱讀以下 WolfDesk 的描述，並在回答相關練習題時回到這裡。

## 示範領域：WolfDesk

WolfDesk 提供服務台工單（help desk tickets）管理系統作為其服務。如果您的初創（start-up）企業需要為您的客戶提供客服，借助 WolfDesk 的解決方案，您可以立即啟動並運行。

WolfDesk 使用和其競爭對手不同的支付模式，它不對每位使用者收取費用，而是讓租戶根據需要設定任意數量的使用者，再根據每次收費期間所開啟的客服工單數量來對租戶收費。沒有最低費用，若超過每月工單的特定門檻，則有總額的自動折扣：月開 500 張以上 10%、月開 750 張以上 20%、月開 1,000 張以上 30%。

為了防止租戶濫用這個商業模式，WolfDesk 工單生命週期的計算方法會確保暫時不用的工單自動關閉，並鼓勵客戶在需要進一步客服時打開新的工單。此外，WolfDesk 實作了一個詐欺偵測（fraud detection）系統，它可以分析訊息並偵測出在同一張工單中討論了無關主題的情況。

為了幫助租戶簡化客服相關的工作，WolfDesk 實作了「客服自動導引」功能。自動導引會分析新的工單，並嘗試從租戶的工單歷史記錄中自動找到相對應的解決方案，這個功能使得工單的壽命更加縮短，鼓勵客戶開啟新的工單以解決更多問題。

WolfDesk 整合了所有安全標準和措施來驗證並授權租戶的使用者，還允許租戶使用他們現有的使用者管理系統來設定單一登入（single sign-on，SSO）。

管理介面讓租戶能設定工單類別的可能值，以及支援的租戶產品列表。為了能夠只在租戶客服人員的工作時間內將新的工單發送給他們，WolfDesk 允許輸入每位客服人員的工作時間。

由於 WolfDesk 提供的服務不收取最低費用，因此它必須最佳化其基礎設施，以最大限度地降低新租戶的入門成本。為此，WolfDesk 利用了無伺服器運算（serverless computing），讓它能根據有效工單上的操作來彈性地擴展其計算資源。

## 本書編排慣例

本書使用以下的編排慣例：

斜體字（*Italic*）

代表新的術語、URLs、email 地址、檔名和副檔名。中文用楷體表示。

定寬字（`Constant width`）

用來列出程式，以及在內文引用的程式元素，例如：變數或函式名稱、資料庫、資料型態、環境變數、陳述式及關鍵字。

這個圖示代表一般註解。

## 使用範例程式

補充材料（範例程式、練習等）可以從 *https://learning-ddd.com* 下載。

書中介紹的所有範例程式都是用 C# 語言實作的。通常，您在章節中看到的範例程式是展示所討論概念的節錄。

第一章

# 分析業務領域

如果您和我一樣，喜愛撰寫程式碼：解決複雜的問題、提出優雅的解決方案、透過精心設計規則、結構和行為來建構全新的世界，我相信這就是您在領域驅動設計（DDD）中會有興趣的地方：您想讓您的技術變得更好。但是，本章與撰寫程式碼無關，在本章中，您將了解公司是如何運作的：它們存在的原因、追求的目標，以及實現目標的戰略。

當我在我的領域驅動設計課程中講授這些內容時，實際上許多學生會問：「我們需要了解這些內容嗎？我們是在撰寫軟體，又不是在經營企業。」對於他們的問題，答案是大聲肯定的「是」。要設計和建立有效的解決方案，您必須了解問題，在我們的上下文（context）中，即是我們必須建立的軟體系統。要理解問題，您必須了解它存在的環境——組織的業務戰略，以及它透過建立軟體來獲得什麼價值。

在本章中，您將學習用於分析公司業務領域及其結構的領域驅動設計工具：其核心（core）、支持（supporting）和通用子領域（generic subdomains）。這份教材是設計軟體的基礎。在其餘的章節中，您將了解這些概念影響軟體設計的不同方式。

## 什麼是業務領域？

業務領域定義了公司的主要活動區域。一般來說，這是公司為客戶提供的服務。例如：

- FedEx 提供快遞服務。
- 星巴克以咖啡聞名。
- Walmart 是最廣為人知的零售店之一。

一家公司可以運營多個業務領域。例如，Amazon 同時提供零售和雲端運算（cloud computing）服務。Uber 是一家共乘公司，還提供送餐和共享單車服務。

重要的是要注意，公司可能會經常更改其業務領域。一個典型的例子是 Nokia，多年來，它在木材加工、橡膠製造、電信和行動通訊等領域開展業務。

# 什麼是子領域？

為了達成公司業務領域的目的和目標，它必須在多個子領域（*subdomains*）中運營。子領域是業務活動的細粒度（fine-grained）區域，公司的所有子領域構成了它的業務領域：它為客戶提供的服務。實行單一的子領域不足以讓公司取得成功；它只是整體系統中的一個建構區塊，子領域必須彼此互動以達成公司在其業務領域中的目的。例如，星巴克最聞名的也許是它的咖啡，但建立一個成功的咖啡連鎖店需要的不僅僅是懂得如何製作出好咖啡，還必須在有效的地點購買或租用房地產、雇用人員，以及管理財務等活動，這些子領域都不會單獨成為一家盈利的公司，所有這些都是公司能夠在其業務領域競爭所必需的。

## 子領域的類型

正如軟體系統包含各種架構元件——資料庫、前端應用程式、後端服務等——子領域具有不同的戰略 / 業務價值。領域驅動設計區分為三種類型的子領域：核心（core）、通用（generic）、支持（supporting），讓我們從公司戰略的觀點來看看它們有何不同。

### 核心子領域

核心子領域是公司與競爭對手的不同之處，這可能涉及發明新產品或服務，或透過最佳化現有流程來降低成本。

讓我們以 Uber 為例，最初，該公司提供了一種新穎的交通方式：共乘。隨著競爭對手的追趕，Uber 找到了最佳化和發展其核心業務的方法：例如透過配對朝同方向前進的乘客來降低成本。

Uber 的核心子領域影響它的盈虧，核心子領域是公司如何和它的競爭對手區別開來的方式，這是公司為客戶提供更好服務和 / 或最大化其盈利能力的戰略。為了維持競爭優勢，核心子領域涉及發明、智慧最佳化、商業技能知識（know-how），或是其他智慧財產。

想想看另一個例子：Google 搜尋的排名（ranking）演算法。在撰寫本文時，Google 的廣告平台占其大部分的利潤，儘管如此，Google Ads 不是一個子領域，而是一個由子領域組

成的獨立業務領域，包含其雲端運算服務（Google Cloud Platform）、生產力和協作工具（Google Workspaces），以及 Google 的母公司 Alphabet 所營運的其他領域。那麼 Google 搜尋及其排名演算法呢？雖然搜索引擎不是付費服務，但它是 Google Ads 最大的展示平台，它提供出色搜尋結果的能力是帶動流量的原因，所以它是 Ads 平台的重要元件。提供欠佳的搜索結果（不論是由於演算法中的錯誤、或是競爭對手提供了更好的搜尋服務）將損害到廣告業務的收入，所以對 Google 來說，排名演算法是一個核心子領域。

**複雜度。**易於實行的核心子領域只能提供短暫的競爭優勢。因此，核心子領域自然是複雜的。繼續 Uber 的例子，該公司不僅透過共乘創造了一個新的市場空間，而且透過針對性地利用科技，顛覆了已有數十年之久的龐大計程車行業架構。透過了解其業務領域，Uber 能夠設計出一種更可靠、更透明的交通方式。公司的核心業務應該要有很高的進入門檻（entry barriers）；這應該讓競爭對手很難複製或模仿公司的解決方案。

**競爭優勢的來源。**重要的是要注意，核心子領域不一定是技術性的，並非所有業務問題都透過演算法或其他技術方案解決，一家公司的競爭優勢可以來自各種來源。

例如，想想看一家在線上銷售其產品的珠寶製造商，網路商店很重要，但它不是核心子領域，珠寶設計才是。該公司可以使用既有、現成的線上商店搜尋引擎，但不能外包其珠寶設計，設計是客戶購買珠寶製造商產品並記得品牌的原因。

另一個較複雜的例子，想像一家專門從事手動詐欺偵測（fraud detection）的公司，該公司培訓其分析師檢查有問題的文件並標記可能的詐欺案件，您正在建立分析師使用的軟體系統，它是核心子領域嗎？不是，核心子領域是分析師正在做的工作。您正在建立的系統與詐欺分析無關，它只是顯示文件並追蹤分析師的評論。

### 核心子領域 vs. 核心領域

核心子領域（core subdomains）也稱為核心領域（core domains）。例如：在最初的領域驅動設計書籍中，Eric Evans 交替使用「核心子領域」和「核心領域」。儘管「核心領域」的術語經常被使用，但出於許多原因，我更喜歡使用「核心子領域」。首先，它是一個子領域，我傾向避免和業務領域混淆。其次，正如您將在第 11 章中學習到的，子領域隨著時間推移而演化並改變它們類型的情況並不少見。例如：核心子領域可以變成通用子領域。因此，說「通用子領域已經演化為核心子領域」比說「通用子領域已經演化為核心領域」更直接。

## 通用子領域

通用子領域（*Generic subdomains*）是所有公司都以相同方式執行的業務活動。就像核心子領域一樣，通用子領域通常很複雜而且難以實行，但通用子領域卻不會為公司提供任何競爭優勢。這裡不需要創新或最佳化：經過實戰考驗的實踐已經廣為可用，所有公司都在使用它們。

例如，大多數系統需要對其用戶進行身份驗證和授權，與其發明專有的身份驗證機制，不如使用現有的解決方案更為合理。這種解決方案可能更加可靠和安全，因為它已經被許多其他具有相同需求的公司測試過了。

回到珠寶製造商在線上銷售產品的例子，珠寶設計是一個核心子領域，但線上商店是一個通用子領域，使用與其競爭對手相同的線上零售平台——相同的通用解決方案——不會影響珠寶製造商的競爭優勢。

## 支持子領域

顧名思義，支持子領域（*supporting subdomains*）支持公司的業務。然而，與核心子領域相反，支持子領域並沒有提供任何的競爭優勢。

例如，想想看一家線上廣告公司，其核心子領域包括配對廣告和訪客、最佳化廣告效果，以及最大限度地降低廣告空間的成本。然而，為了在這些領域取得成功，公司需要分類它的創意素材。公司把它的實際創意素材像是橫幅（banners）和登陸頁面（landing pages）儲存並編入索引的方式不會影響利潤，在這個領域沒有什麼可以發明或最佳化的。從另一個角度來看，創意目錄對於實行公司的廣告管理和服務系統而言是不可或缺的。這使得素材編目的解決方案成為公司的支持子領域之一。

支持子領域的顯著特徵是解決方案的業務邏輯複雜度。支持子領域很簡單，它們的業務邏輯主要類似於資料輸入螢幕和 ETL（提取（extract）、轉換（transform）、載入（load））的操作；也就是所謂的 CRUD（新增（create）、讀取（read）、更新（update）、刪除（delete））介面，這些活動區域不會為公司提供任何的競爭優勢，因此不需要很高的進入門檻。

## 比較子領域

現在我們對這三種類型的業務子領域有了更深入的了解，讓我們從其他角度來探討它們的差異吧，看看它們如何影響戰略性的軟體設計決策。

## 競爭優勢

只有核心子領域才能為公司提供競爭優勢，核心子領域是公司將自己與競爭對手區別開來的戰略。

根據定義，通用子領域不能成為任何競爭優勢的來源。這些是通用解決方案——公司與其競爭對手使用的相同解決方案。

支持子領域的進入門檻低，也不能提供競爭優勢。通常，公司不會介意競爭對手複製其支持子領域——這不會影響它在行業中的競爭力。相反的，從戰略上說，公司希望其支持子領域是通用的、現成的解決方案，從而消除了設計和建立實踐的需要。您將在第 11 章中詳細了解支持子領域轉變為通用子領域的這類案例，以及其他可能的排列變化。附錄 A 將概述這類情境的真實案例研究。

公司能夠解決的問題越複雜，它可以提供的商業價值就越大。複雜的問題不僅限於向消費者提供服務，一個複雜的問題可以像是使業務更有效率和最佳化，例如提供和競爭對手相同水平的服務，但營運成本較低，這也是一種競爭優勢。

## 複雜度

從更為技術的角度來看，辨別組織的子領域很重要，因為不同類型的子領域具有不同等級的複雜度，在設計軟體時，我們必須選擇適合業務需求複雜度的工具和技術。因此，辨別子領域對於設計完善的軟體解決方案而言至關重要。

支持子領域的業務邏輯很簡單，這些是基本的 ETL 操作和 CRUD 介面，業務邏輯顯而易見。它通常不會超出驗證輸入，或把資料從一種結構轉換為另一種結構的範圍之外。

通用子領域要更複雜得多，對於其他人為何投入時間和精力來解決這些問題，應該有其充分的理由。這些解決方案既不簡單也不普通，想想看例如加密演算法或身份驗證機制。

從知識可用性的角度來看，通用子領域是「已知的未知（known unknowns）」，這些是您知道自己不知道的事情。除此之外，這些知識很容易獲得，您可以使用業界認可的最佳實踐（industry-accepted best practices），或是如果有需要，聘請專門從事該領域的顧問來幫忙設計客製化的解決方案。

核心子領域很複雜，它們應該讓競爭對手盡可能地難以複製——公司盈利的能力取決於它。這就是為什麼在戰略上，公司指望將複雜的問題作為其核心子領域來解決。

區分核心子領域和支持子領域有時候可能具有挑戰性。複雜度是一個有用的指導原則，詢問正在討論中的子領域是否可以變成副業，有人會自己付錢嗎？如果會，這就是一個核心子領域。類似的推理也適用於區分支持子領域和通用子領域：用您自己的實踐會比整合外部的實踐更簡單、更便宜嗎？如果是，這就是一個支持子領域。

從更為技術的角度來看，重要的是識別複雜度會影響到軟體設計的核心子領域，正如我們前面所討論的，核心子領域不一定與軟體相關。另一個有用的指導原則來識別與軟體相關的核心子領域是評估您必須用程式碼建模和實作的業務邏輯複雜度。業務邏輯是否類似於用於資料輸入的 CRUD 介面，還是您必須實作由複雜業務規則和固定規則（invariants）編排出的複雜演算法或業務流程？前者是支持子領域的標識，後者則是典型的核心子領域。

圖 1-1 中的圖表代表三種類型的子領域在業務差異性和業務邏輯複雜度的相互作用。支持子領域和通用子領域之間的交集是一個灰色區域：它可以走任何一條路。如果支持子領域的功能存在通用解決方案，則生成的子領域類型取決於整合的通用解決方案是否比從頭開始實作功能更簡單和 / 或更便宜。

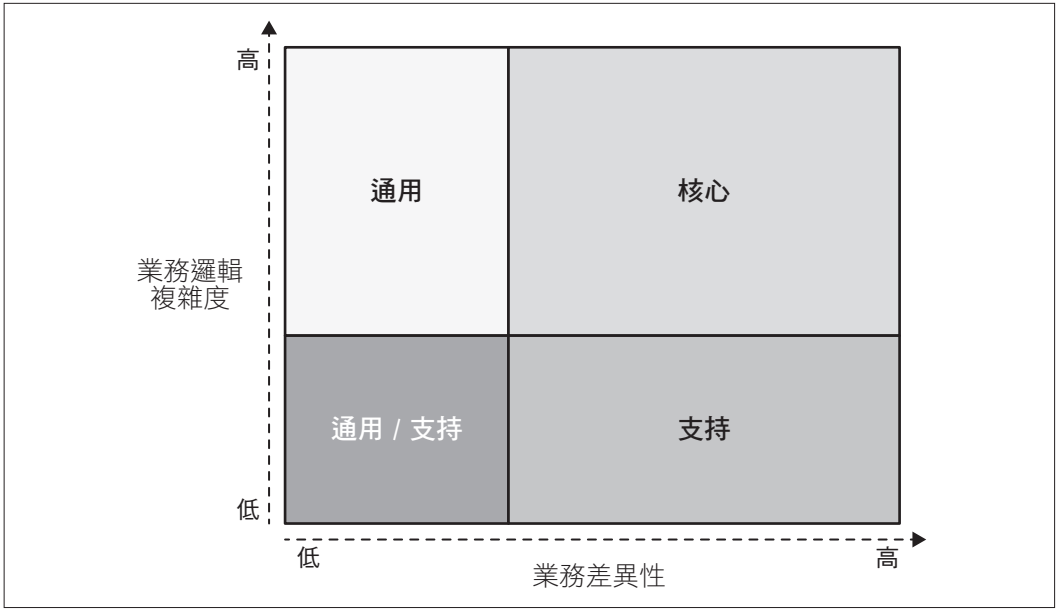


圖 1-1　三種類型子領域的業務差異性和業務複雜度

## 不穩定性

如前所述，核心子領域可以經常更改。如果一個問題可以在第一次嘗試就解決，它可能不是一個好的競爭優勢——競爭對手很快就會趕上。因此，核心子領域的解決方案應運而生，必須嘗試、改進和最佳化不同的實踐，此外，核心子領域的工作永遠不會完成，公司要不斷創新和發展核心子領域。這些更動以增加新功能或最佳化既有功能的形式出現，不論哪種方式，其核心子領域的不斷發展，對於公司能持續領先競爭對手來說都是不可或缺的。

和核心子領域相反，支持子領域不會經常更動。它們不會為公司提供任何競爭優勢，因此和投入相同的精力在核心子領域上相比，支持子領域的發展提供了微不足道的商業價值。

儘管有既存的解決方案，但通用子領域會隨著時間而更動，這些改變出現的形式可以是安全補丁、錯誤修復，或是對於通用問題的全新解決方案。

## 解決方案的戰略

核心子領域為公司提供了與業內其他參與者競爭的能力，這是一項關鍵業務的責任，但這是否意味著支持和通用子領域並不重要？當然不是。公司需要所有子領域才能在其業務領域中運行。子領域就像基礎的建構區塊：拿走一個，整個結構可能就會倒塌。儘管如此，我們可以利用不同類型子領域的固有特性來選擇實行的戰略，以最有效的方式實行每種類型的子領域。

核心子領域必須在內部實行，它們不能被購買或採用；這會破壞競爭優勢的概念，因為公司的競爭對手也可以這樣做。

外包核心子領域的實行也是不明智的。這是一項戰略性投資，在核心子領域上走捷徑不僅在短期內是有風險的，而且從長遠來看可能會產生致命的後果：例如不能支持公司目標的不可維護程式碼庫。應該指派組織中最熟練的人才從事其核心子領域的工作，此外，在內部實行核心子領域使公司能夠更快地做出改變和發展解決方案，進而在更短的時間內建立競爭優勢。

由於核心子領域的需求預計會經常不斷地變化，因此解決方案必須可維護且易於發展。因此，核心子領域需要最先進工程技術的實踐。

由於通用子領域很難，但已經解決問題了，因此購買現成的產品或採用開源解決方案，比投入時間和精力在內部實行通用子領域更具成本效益。

避免在內部實行支持子領域是合理的，因為它缺乏競爭優勢，但和通用子領域不同的是，它沒有現成的解決方案可以使用，所以公司別無選擇，只能自己實行支持子領域。儘管如此，業務邏輯的簡單性和更改的頻率很低，因此很容易走捷徑。

支持子領域不需要精細的設計模式或其他先進的工程技術，一個快速的應用程式開發框架將足以實作業務邏輯，而不會引入意外的複雜度。

從人事的角度來看，支持子領域不需要高度熟練的技術能力，並提供了一個很好的機會來培養有潛力的人才，省去您團隊中經驗豐富的工程師，他們熟練於解決核心子領域的複雜挑戰。最後，業務邏輯的簡單性使支持子領域成為外包的良好候選。

表 1-1 總結了三種類型子領域的不同面向。

表 1-1　三種子領域的區別

| 子領域類型 | 競爭優勢 | 複雜度 | 不穩定性 | 實行 | 問題 |
|---|---|---|---|---|---|
| 核心 | 是 | 高 | 高 | 內部 | 引人入勝 |
| 通用 | 否 | 高 | 低 | 購買 / 採用 | 已經解決 |
| 支持 | 否 | 低 | 低 | 內部 / 外包 | 顯而易見 |

## 辨別子領域邊界

正如您已經看到的，在建立軟體解決方案時，辨別子領域及其類型可以極大地幫助您制定出不同的設計決策，在後面的章節中，您將學習到更多利用子領域來簡化軟體設計過程的方法。但是，我們實際上要如何辨別子領域及其邊界呢？

子領域及其類型由公司的業務戰略定義：其業務領域以及它如何區別自己以和同一領域的其他公司競爭。在絕大多數的軟體專案中，子領域以某種方式「已經在那裡了」。然而，這並不表示辨別它們的邊界總是簡單又直接的。如果您向 CEO 索取他們公司子領域的列表，您可能會收到茫然的眼神，他們不知道這個概念。因此，您必須自己進行領域分析以辨別並分類運作中的子領域。

公司的部門和其他組織單位是一個好的起點。例如，線上零售店可能包括倉儲、客戶服務、揀貨、運輸、品質控管和通路管理等部門。然而，這些是相對粗粒度（coarse-grained）的活動領域。以客服部門為例，可以合理地假設它是一個支持，甚至是一個通用的子領域，因為這個功能經常外包給第三方供應商。但是，這些資訊足以讓我們做出合理的軟體設計決策嗎？

## 精煉子領域

粗粒度的子領域是一個很好的起點，但魔鬼藏在細節裡，我們必須確保不會遺漏了隱藏在錯綜複雜業務功能中的重要資訊。

讓我們回到客服部門的例子，如果我們調查它的內部運作，我們會發現一個典型的客戶服務部門是由更細粒度（finer-grained）的元件（components）組成，例如服務台（help desk）系統、輪班管理和排班、電話系統等。當這些活動被視為單獨的子領域時，它們可以是不同的類型：雖然服務台和電話系統是通用子領域，但輪班管理是支持子領域。而公司可能會開發其獨創的演算法，將事件派送到曾在類似個案中獲得成功的客服人員。

派送演算法需要分析傳入的個案並識別與過去經驗中的相似之處——這兩項都是不普通的任務。由於派送演算法讓公司提供比競爭對手更好的客戶體驗，因此派送演算法是一個核心子領域。這個例子如圖 1-2 所示。

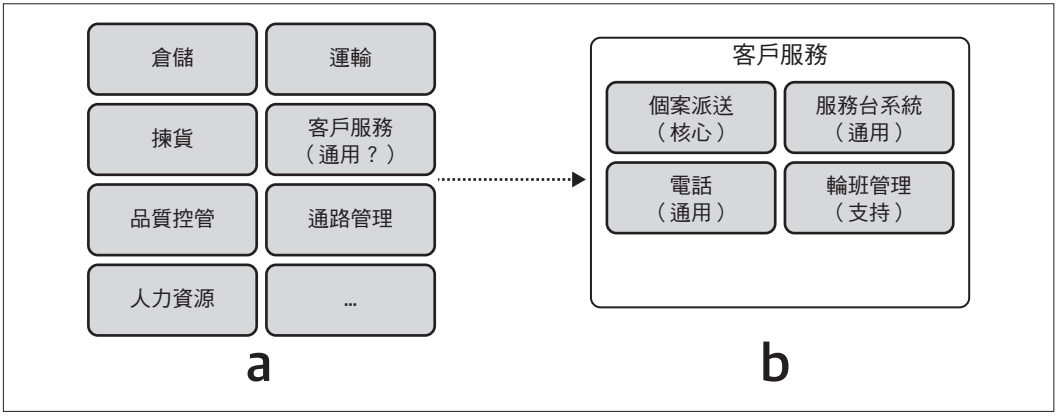


圖 1-2　分析疑似通用子領域的內部運作，以找到更細粒度的核心子領域、支持子領域和兩個通用子領域

從另一方面來看，我們不能無限深入下去，在越來越低的粒度級別上尋找洞察。您應該在什麼時候停下來呢？

## 把子領域視為連貫的使用案例

從技術的角度來看，子領域類似於一組相互關聯、連貫的使用案例。這樣的使用案例集通常涉及相同的角色、業務實體，並且它們都操作一組密切相關的資料。

考量圖 1-3 所示的信用卡付款閘道（payment gateway）使用案例圖。使用案例與他們正在使用的資料和涉及的角色緊密結合。因此，所有用使用案例都形成了信用卡付款的子領域。

我們可以使用「把子領域視為一組連貫的使用案例」的定義當作何時停止尋找更細粒度子領域的指導原則，這些是子領域最精確的邊界。

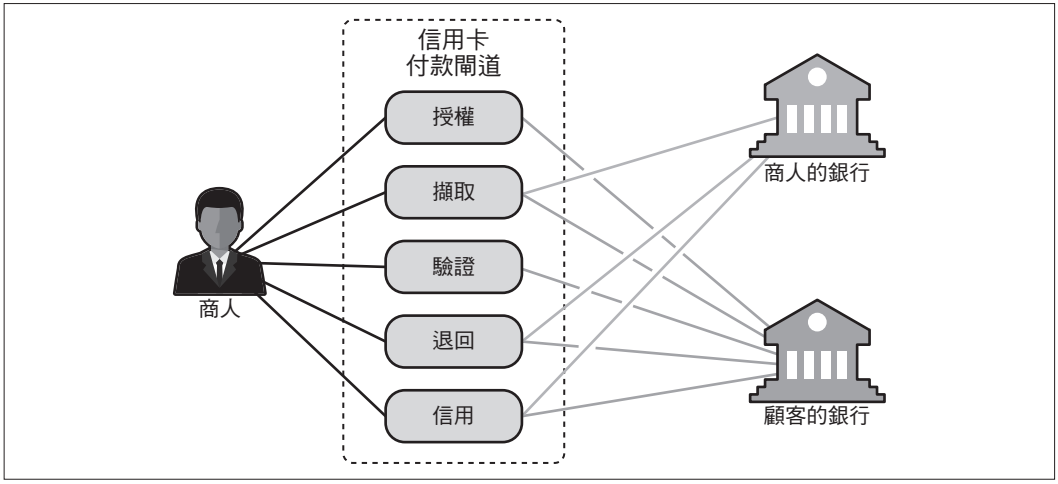


圖 1-3　信用卡支付子領域的使用案例圖

您是否應該持續努力去辨別這種高度聚焦的子領域邊界？對於核心子領域而言，這絕對是必要的。核心子領域是最為重要、不穩定且複雜的。我們必須盡可能地精煉它們，因為這將讓我們提取所有通用和支持功能，並把精力投入到更加聚焦的功能上。

對於支持和通用子領域，精煉可以稍微放鬆些。如果進一步深入下去並沒有揭露任何能幫助您制定出軟體設計決策的新洞察，那麼它可能是一個停下來的好地方。例如，當所有更細粒度的子領域和原始子領域的類型相同時，就會發生這種情況。

考量圖 1-4 中的範例。服務台系統子領域的進一步精煉不太有用，因為它不會揭示任何戰略性資訊，並且將使用粗粒度的現成工具作為解決方案。

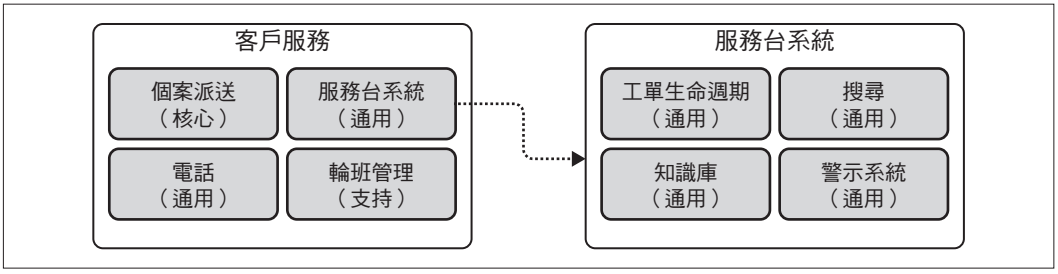


圖 1-4　精煉服務台系統子領域，揭露通用的內部元件

辨別子領域時要考慮的另一個重要問題是：我們是否需要所有子領域。

### 專注於本質

子領域是一種改善軟體設計決策過程的工具，所有組織都可能擁有相當多，卻與軟體無關的業務功能來推動它們的競爭優勢，在本章前我們討論的珠寶製造商就是一個例子。

在尋找子領域時，重要的是要識別和軟體無關的業務功能，承認它們的存在，並專注於您正著手的軟體系統相關的業務面向。

## 領域分析範例

來看看我們如何在實踐中應用子領域的概念，並將其用於制定許多戰略設計的決策上。我將描述兩個虛構的公司：Gigmaster 和 BusVNext。作為練習，在閱讀時分析公司的業務領域，嘗試為每家公司辨別三種類型的子領域。請記住，在現實生活中，一些業務需求是隱性的。

免責聲明：當然，我們無法透過閱讀如此簡短的描述來識別每個業務領域所涉及的所有子領域。換句話說，只要訓練您能辨別並分類出可用的子領域就夠了。

### Gigmaster

Gigmaster 是一家門票販售和經銷公司，它的行動應用程式分析使用者的音樂庫、串流服務帳戶、社群媒體的個人資料，以找出使用者有興趣參與的鄰近表演。

Gigmaster 的使用者對於他們的隱私有所意識，因此，所有使用者的個人資訊都是加密的。此外，為了確保使用者罪惡的享樂在任何情況下都不會洩露出去，該公司的推薦演算法僅在匿名資料上運作。

為了改進應用程式的推薦，一個新的模組被實作出來了。它允許使用者記錄他們曾出席過的演出，即使門票不是透過 Gigmaster 購買的。

## 業務領域和子領域

Gigmaster 的業務領域是門票銷售，這就是它為客戶提供的服務。

**核心子領域。**Gigmaster 的主要競爭優勢是它的推薦引擎，該公司還非常重視使用者的隱私，而且僅在匿名資料上運作。最後，雖然沒有明確提及，但我們可以推論行動應用程式的使用者經驗也相當重要。因此，Gigmaster 的核心子領域是：

- 推薦引擎
- 資料匿名化
- 行動應用程式

**通用子領域。**我們可以識別並推論出以下通用子領域：

- 加密，用於加密所有資料
- 會計，因為公司從事銷售業務
- 結帳，向客戶收費
- 身份驗證和授權，用於識別它的使用者

**支持子領域。**最後，以下是支持子領域，這裡的業務邏輯很簡單，類似於 ETL 流程或 CRUD 介面：

- 和音樂串流服務整合
- 和社群網路整合
- 出席演出的模組

## 設計決策

知道運作中的子領域和它們類型之間的差異後，我們已經可以制定出幾個戰略設計的決策：

- 推薦引擎、資料匿名化和行動應用程式必須在內部實作，並使用最先進的工程工具和技術，這些模組將最常更動。
- 現成或開源的解決方案應該用於資料加密、會計、結帳、認證。
- 與串流服務、社交網路的整合，以及出席演出的模組可以外包。

# BusVNext

BusVNext 是一家大眾運輸公司，它旨在為客戶提供舒適的巴士服務，就像坐計程車一樣，該公司在各大城市中管理巴士車隊。

BusVNext 客戶可以透過行動應用程式訂車，在預定的出發時間，附近的巴士路線將即時調整，以便在指定的出發時間接送客戶。

該公司的主要挑戰是實作路線選擇（routing）演算法。它的需求是「旅行推銷員問題（travelling salesman problem）」（*https://oreil.ly/LLHij*）的變形，路線選擇邏輯不斷地調整和最佳化。例如，統計數據顯示取消乘車的主要原因是等待巴士抵達的時間過長。因此，該公司調整了路線選擇演算法以優先考慮快速接客，即使這代表延遲下車。為了進一步最佳化路線選擇，BusVNext 與第三方供應商整合交通狀況和即時警報。

BusVNext 會不定時發布特別折扣，以吸引新客戶並平衡高峰和離峰時段的乘車需求。

## 業務領域和子領域

BusVNext 為其客戶提供最佳化的巴士乘車服務，業務領域是大眾運輸。

**核心子領域。**BusVNext 的主要競爭優勢是它的路線選擇演算法，它在解決複雜問題（「旅行推銷員」（traveling salesman））的同時，優先考慮不同的業務目標：例如減少接客時間，即使它會增加整體行程的長度。

我們還看到乘車資料被不斷地分析，以獲取對客戶行為的新洞察。這些洞察讓公司能透過最佳化路線選擇算法來增加利潤。最後，BusVNext 為其客戶及司機提供的應用程式必須容易使用，而且提供便利的使用者介面。

管理車隊並非易事，巴士可能會遇到技術問題或需要維護，忽視這些可能會導致財務損失和服務水準下降。

因此，BusVNext 的核心子領域是：

- 路線選擇
- 分析
- 行動應用程式的使用者經驗
- 車隊管理

**通用子領域。**路線選擇演算法還使用第三方公司提供的交通數據和警報——通用子領域。此外，BusVNext 接受來自客戶的付款，因此它必須實作會計和結帳功能，BusVNext 的通用子領域是：

- 交通狀況
- 會計
- 計費
- 授權

**支持子領域。**管理促銷和折扣的模組支持公司的核心業務。儘管如此，它本身並不是核心子領域。它的管理介面類似用於管理有效優惠券代碼的簡單 CRUD 介面。因此，這是一個典型的支持子領域。

## 設計決策

知道運作中的子領域及它們類型之間的差異，我們已經可以做出一些戰略設計的決策：

- 路線選擇演算法、資料分析、車隊管理，以及應用程式的易用性必須在內部實作，並使用最精密的技術工具和模式。
- 促銷管理模組的實作可以外包。
- 辨別交通狀況、授權使用者，以及管理財務記錄和交易可以交給外部服務的供應商。