前言

我從未計劃寫關於文件製作的書，也不覺得這個主題值得寫一本書。

很久以前我有個雄偉的夢想，打算製作能了解程式設計時的設計決策的工具。多年間我花很多閒暇時間嘗試建立這樣一個框架，最後發現非常難建立一個適合所有人的框架。但我還是在嘗試了對這個專案有任何幫助的想法。

2013 年我在 Øredev 談重構規格，最後提到我過去嘗試過的一些想法，意外的是我收到很多關於活文件想法的熱情回饋。此時我認識到有需要更好的製作文件方法。然後我又做了幾次同樣的演講，持續有關於文件製作、改善方法、實時自動化製作的回饋。

活文件（*living documentation*）一詞出自 Gojko Adzic 的 *Specification by Example* 一書，是實例規格的許多好處之一。但活文件這個名字不只適用於規格而已。

我對活文件有許多想法可以分享。我列出曾經嘗試過以及相關的東西。更多想法來自其他人——實際認識以及只在 Twitter 認識的人。隨著想法的累積，我決定寫一本書。相較於提供現成的框架，我認為書更能幫助你建立快速且自定的方案來製作你自己的活文件。

本書主題

活文件的想法出自於 *Specification by Example* 一書，描述寫文件的行為範例可提升為自動化測試。你知道測試失敗時文件就不再與程式碼一致，而你可以很快的修改。這個想法顯示出能夠製作有效且不會在寫完就過期的文件。我們還可以讓這個想法更進一步。

這本書擴展 Gojko 關於活文件的想法，讓文件從專案的業務目標、業務領域知識、架構與設計、程序、部署等各方面隨著程式碼演進。

這本書結合理論與實務，包含圖表與範例。你會認識到如何以良好的製作物以及合理的自動化投資，於會更新且成本最小的文件製作。

你會發現無需在可用軟體與大量文件間做選擇！

本書讀者

這本書主要是為了開發者或不怕原始碼控制系統中的程式碼的人。它以程式碼為中心，適合開發者、程式設計架構設計師、懂程式的資深角色。它也以修改原始碼並提交給原始碼控制系統的軟體開發者的觀點，討論業務分析與經理人等其他利益關係人的需求。

這本書不討論使用者文件的製作。寫使用者文件需要技術性寫作等特定技巧，而這絕對不是本書的主題。

如何閱讀本書

這本書的主題是活文件，以相關主題模式展開。每個模式的內容可獨立閱讀，但建議同時閱讀相關模式以充分認識各個模式的適用背景。本書網站有展示模式關係圖。

本書內容安排從管理知識問題開始，然後是 BDD 的啟示、一些初步理論、不同步調的知識變化與相對應的文件製作技術。接下來的內容專注於架構與舊系統上的應用，以及如何在你的環境中引進活文件。

建議從第 1 章開始，並在開始討論一般實務技術的第 5 章到第 9 章前先掌握第 3 章與第 4 章的重要概念。然後以第 10 章轉換觀點。第 11 章到第 15 章討論特定主題並提供額外範例。

有些讀者喜歡從第一頁讀到最後一頁；但也可以掃過、細讀、或隨意翻閱。

本書內容

第 1 章“重新思考文件”從第一原理檢視文件製作，提供後續章節的基礎。

第 2 章“行為驅動開發即為實例規格”說明 BDD 如何啟發活文件，但 BDD 本身並非本書的主題。

第 3 章“知識利用”與第 4 章“知識增強”為其他實踐奠基，特別是討論擷取知識並補漏增強。

第 5 章“有效整理展示：識別權威知識”展示如何透過整理與展示將知識轉換成有用的東西，並接受認同知識的持續變化。

第 6 章“文件自動化”將知識隨著改變的節奏化為文件與圖表。

第 7 章“執行即為文件”擴展前一章，討論如何運用執行時才能取得的資訊。

第 8 章“可重構文件製作”以程式碼為中心並專注於以開發工具輔助更新文件。

第 9 章“穩定的文件”討論不會改變而無需活文件技術的知識與這種知識的文件製作方法。

第 10 章“避免傳統文件”採取更激進的觀點，專注於文件紀錄的替代方案。

透過設計改善文件製作後，第 11 章“超越文件：活設計”採取另一種觀點：專注於文件製作如何幫助你改善設計本身。

第 12 章“活架構文件”將活文件應用於軟體架構並討論特定技術。

第 13 章“新環境導入活文件”指導如何對你的環境引進活文件，主要是人際挑戰。

由於我們身旁都是舊系統，第 14 章“製作舊應用程式的文件”以處理舊系統挑戰的特定模式終結。

附贈的第 15 章“額外收錄：醒目的文件”提出讓推動活文件更有效的實務建議。

Chapter 1

重新思考文件

忘掉文件。相反的，專注於軟體開發的速度。你想要更快的交付軟體。不只是讓現在加快，還要長期持續維持速度。不只是讓你加快，還要讓整個團隊或公司加快。

讓軟體開發更快不只涉及高生產力程式語言與框架、更好的工具、更高水準的技能，但業界在這些方面做出越多進步，我們就必須越檢查其他瓶頸。

除了利用科技外，寫軟體更多是關於基於知識的決策。沒有足夠的知識時，你必須透過實驗學習並與他人合作發現新知識。這需要時間，同時也表示知識的代價與價值很高。變快在於需要新知識時學得更快或快速發現以前的有用知識。讓我們用一個小故事說明。

活文件傳奇

故事是這樣的。有個開發新應用程式的軟體專案是公司資訊系統的一部分。你是此專案的開發者。你的任務是新增一種老客戶折扣。

為什麼要這個功能？

你與行銷團隊的 Franck 與測試者 Lisa 開會討論新功能、提問、找使用案例。Lisa 問：“為什麼要這個功能？”。Franck 解釋是因為要獎勵老客戶以遊戲化方式提升回購率並推薦維基百科的條目。Lisa 在筆記寫下討論要點與主要場景。

討論進行的很快，因為大家面對面溝通。還有，案例很容易理解且之前不清楚的地方也釐清了。全部清楚後，各自回到自己的座位。這次輪到 Lisa 寫紀錄並發給所有人（上一次輪到 Franck）。現在你可以開始寫程式。

你之前的工作程序不是這樣。團隊間透過難以閱讀、含糊的文件溝通。你笑了。你很快的將第一個場景寫成自動化測試、看著它失敗、開始寫讓它通過的程式碼。

你很高興的覺得你寶貴的時間沒有浪費而是花在重要的事情上。

接下來就不再需要這個草圖

當天下午，同事 Georges 與 Esther 問了一些必須做的設計決策。你們在白板前開會並快速的評估所有選項。此時不需要動用 UML [1]，只需要畫一些方塊與箭頭。你想要確保每個人都懂了。幾分鐘後選出一個解決方案。計劃是在訊息系統中使用兩種不同的主題；這麼做是因為必須分離訂單與出貨請求。

Esther 用手機將白板拍下來以防被擦掉，但她知道半天後就會實作出來，然後可以安全的從手機中刪除照片。一小時後，她在提交新訊息主題時加註原因為“分離訂單與出貨請求”。

隔天，前一天請假的 Dragos 注意到新程式碼並產生疑問。他執行 `git blame` 並立即得到答案。

抱歉，我們沒有行銷文件！

一週後，新來的行銷經理 Michelle 取代了 Franck。Michelle 比 Franck 更注重顧客回購率。她想要知道應用程式中已經做了哪些顧客回購率有關的功能，所以她查了行銷文件並很驚訝什麼都沒有寫。

1 統一模型語言，Unified Modeling Language: http://www.uml.org/

她叫道：“不可能！”。但你馬上打開驗收測試紀錄給她看。她搜尋“顧客回購率”並檢視結果：

```
為了增加顧客回購率
身為一個行銷人
我想要提供忠實顧客折扣

    情境：忠實顧客下次購買時折抵 10 元
    ...

    情境：忠實顧客上一週購買 3 次
    ...
```

搜尋結果顯示很多給忠實顧客的特殊折扣情境。Michelle 笑了。她甚至不用查行銷文件也會得到這個知識。這些情境的正確程度超過她的預期。

Michelle 問：“能不能歐元也做這種折扣？”。你回答：“我不太熟悉外幣部分，但我們可以試試看 “。你從 IDE [2] 改變測試的幣別並再次執行測試。失敗，因此你知道需要改程式碼才能處理外幣。Michelle 馬上得到答案。她覺得你的團隊跟她以前遇到的不太一樣。

你一直用這個詞，但它不是這個意思

次日 Michelle 有另一個問題：購買與訂單有什麼不同？

她通常只會要求開發者檢查程式碼並說明差別。但團隊預見到這個問題，專案網站已經列出詞彙表。她問“詞彙表有更新嗎？”。你答“有，每一次建置都自動更新”。她很驚訝。為什麼沒有每個人都這麼做？你簡短的回答：“程式碼必須與業務領域一致才行”，但其實你很想引用 Eric Evans 的 *Domain-Driven Design* [3] 一書的論述。

2 整合開發環境（integrated development environment，IDE）。

3 Evans, Eric. Domain-*Driven Design: Tackling Complexity in the Heart of Software.* Hoboken:Addison-Wesley Professional, 2003.

Michelle 從詞彙表中找到之前沒有人發現的命名問題並提出正確名稱的建議。但事情不是這樣做的。修改詞彙表名稱首先要改程式碼。將類別重新命名並執行建置，然後詞彙表也跟著改好了。每個人都滿意，而你今天又新學到電子商務的東西。

看大局就知道哪裡出錯

現在你想要消滅兩個模組間有害的相依性，但你不熟悉全部的程式，因此你問熟悉這一塊的 Esther 要相依圖。她說：“我會從程式碼產生相依圖。我一直想要這個。這需要一小時，但跑完以後就可以用”。

Esther 知道有幾個開源函式庫可以擷取類別或套件的相依性，她很快的設定好 Graphviz 來自動產生圖表。幾個小時後，她的小工具產生出相依圖。你拿到你要的東西，你很滿意。然後她花了半小時將這個工具整合進建置。

有趣的是 Esther 第一次檢視此圖表時，她注意到一個問題：某兩個模組間不應該相依。比較記憶中的概觀與系統實際跑出來的圖就很容易找出設計缺陷。

這個設計缺陷在下一個迭代中改正，而相依圖在下一個建置中自動更新。圖變得更乾淨了。

活文件的未來是現在

這個故事不是未來。它已經發生，就是現在，已經出現好幾年。引用科幻作家 William Gibson 的話：“未來已至，只是還未普及”。

工具已經到了。技術已經到了。人們早就開始這麼做了，只是還未成主流。可惜了這麼好的軟體開發想法。

接下來的內容會討論上述方法以及其他方法，你會學到如何在專案中應用它們。

傳統文件製作的問題

> 文件是程式設計的瀉藥——經理人認為它對程式設計師很好，但程式設計師討厭它！
>
> ——*Gerald Weinberg*，《*Psychology of Computer Programming*》

文件製作是很悶的主題。我不知道你怎麼想，但我的經驗顯示文件製作是沮喪的重大源頭。

嘗試閱讀文件時總是找不到我要的資訊。就算找到，通常也過時或有錯，無法信任。

為他人製作文件很無聊，我情願寫程式。但事情不一定得這樣。

很多時候我看過、用過、聽過更好的文件製作方法。我嘗試過很多方法。我蒐集了很多故事，你會在這本書中看到一些。

有比較好的方法，但需要對文件製作採取不一樣的心態。具備這種心態與相應的技術就能讓文件製作與寫程式一樣有趣。

製作文件通常不酷

聽到寫文件你會想到什麼？下面是幾個可能的答案：

- 無聊。
- 寫很多文字。
- 試著在移動 Microsoft Word 中的圖片時維持心平氣和。
- 身為一個開發者，我喜歡展現行動的動態、可執行的東西。對我來說，製作文件像是一潭死水。

- 它應該要有用，但通常只導致誤會。
- 寫文件是無聊的苦差事，我不如去寫程式（見圖 1.1）！

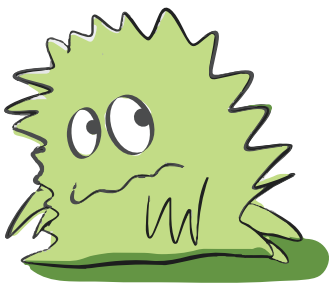

圖 1.1 天啊…我比較想寫程式！

文件需要很多時間撰寫與維護，它很快就會過時、最好的文件也不完整、完全沒有樂趣可言。製作文件令人頭疼。給你看這麼悶的主題我也覺得抱歉。

文件製作的缺陷

> 如同劣酒，紙本文件很快老化且讓你頭疼。
>
> ——*@gojkoadzic* 的推文

傳統文件有許多缺陷與反模式。反模式的意思是很糟糕且應該避免的復發問題。

下面是一些最常見的文件缺陷與反模式。你的專案出現幾個？

活動分離

就算是宣稱敏捷的軟體開發專案，建置、寫程式、測試、寫文件通常是分離的活動，如圖 1.2 所示。

活動分離引發很多浪費與失去機會。基本上，各個活動都在操作相同的知識，但形式不同且製作物也不同（或許有一些重複）。此外，“相同”的知識會在過程中演進而導致不一致。

圖 1.2 軟體開發專案中的活動分離

抄錄

等到寫文件時，團隊成員選取一些完成的知識元素，並以符合受眾預期的格式抄錄。基本上，這表示以另一種文件寫出程式碼已經做過的事情，像是印刷術發明之前的抄寫員（見圖 1.3）。

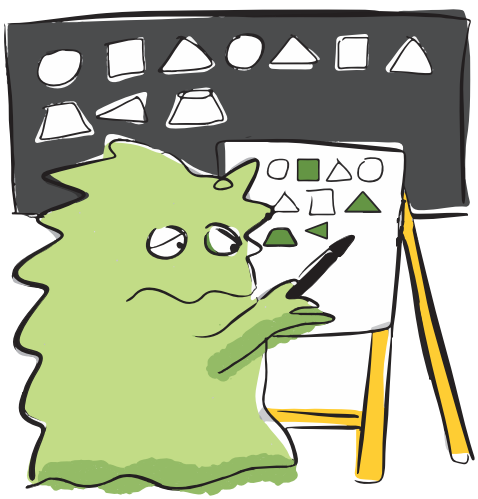

圖 1.3 抄錄

重複的知識

抄錄只產生重複的知識。最終得到的還是原始材料（通常是程式碼）與各種形式的複製品。不幸的是，修改某個製作物（例如程式碼）很難記得要跟著改其他文件。結果文件很快的過時，文件最終因不完整而不可信。這種文件有什麼用？

浪費時間

經理人想要可給使用者與團隊新人看的文件。但開發者討厭寫文件。它較寫程式或任務自動化無聊。死文字很快的過時且因為不能執行而讓開發者覺得特別無聊。開發者寫文件時會希望做點真正有意義的事情。矛盾的是他們使用第三方的軟體時又特別希望有更多的文件。

寫技術文件是一份工作，但所需的知識通常來自開發者，而且通常也只是抄錄而已。這很悶且消耗很多寶貴的時間（見圖 1.4）。

圖 1.4 文件很花時間

想到什麼寫什麼

由於寫文件很無聊且不得不寫，通常是隨便寫寫而沒有深思熟慮。結果就是當時想到什麼寫什麼（見圖 1.5），這對任何人都沒有幫助。

圖 1.5 想到什麼就寫什麼的文件不一定有用

美化圖表

反模式經常出現在喜歡使用 CASE 工具的人的身上。這些工具不是用來打草稿的，它是用來製作各種大圖表佈局與檢驗模型設計。這些工作很花時間。雖然這些工具有自動佈局的功能，但就算是簡單的圖也會花太多時間。

執迷於記號法

UML 越來越不流行，但從 1997 年成為標準以後就是所有大大小小軟體的通用記號法，無論是否合適。之後沒有其他記號法這麼流行，很多團隊不管是否適合都還在用 UML 製作文件。你只知道 UML 時，所有東西看起來都是它的標準圖表。

沒有記號法

事實上，執迷記號法的反方越來越常見。許多人完全忽略 UML，用別人看不懂的自定記號法畫圖表，並隨意混合如建置相依性、資料流程、部署考量等部分。

資訊墳場

知識死在企業知識管理解決方案。以下列項目來說：

- 企業 wiki
- SharePoint
- Microsoft Office 文件
- 共用資料夾
- 搜尋功能不佳的記錄系統與 wiki

這些文件製作方法通常都因很難找到正確資訊或很難更新而失敗。它們偏好唯寫或只寫一次的文件製作方式。

在一次 Twitter 對話中，Tim Ottinger（@tottinge）問到：

> 產品類別：“文件墳場”——是否所有文件管理、wiki、SharePoint、共用資料夾都完了？

James R. Holmes（@James_R_Holmes）回答：

> 有個笑話是你說“在 intranet 裡面”時，對方的反應會是“你是不是叫我去吃 ___?”。
>
> （注意：不雅字眼已經拿掉；你知道是什麼意思）

誤導

文件未能嚴格更新時會誤導，如圖 1.6 所示。雖然看起來有用，其實是錯的。這種文件也許讀起來很有趣，但需要額外花大量時間分辨什麼還是對的與什麼已經不對了。

現在還有別的更重要的事情

寫文件需要很多時間，維護甚至需要更多時間。有時間壓力的人經常會略過文件工作或隨便做。

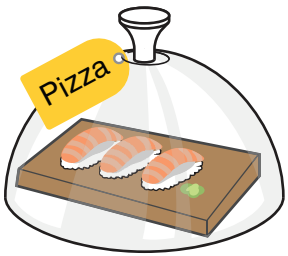

圖 1.6 會誤導的文件有毒

敏捷宣言與文件製作

敏捷宣言是由一群軟體從業者於 2001 年寫下，其中列出的價值觀包括：

- 個人與互動重於流程與工具
- 可用的軟體重於詳盡的文件
- 與客戶合作重於合約協商
- 回應變化重於遵循計劃

第二條“可用的軟體重於詳盡的文件”經常被誤解。許多人認為它完全摒棄文件。事實上，敏捷宣言並沒有說“不要製作文件”。它只是偏好而已。宣言的作者表示：“我們擁抱文件，但不為從不維護與罕用的文件浪費紙張”[4]。然而，隨著敏捷成為大公司的主流，誤解還是存在且許多人忽略文件製作。

4 Martin Fowler 與 Jim Highsmith，http://agilemanifesto.org/history.html

但我注意到最近的缺少文件是許多客戶與同事的麻煩的來源，而這些麻煩越來越嚴重。我很驚訝的是，我於 2013 年在瑞典 Öredev 的研討會首次提出活文件後，看到對文件製作主題很大的需求。

是時候開始文件 2.0

傳統文件製作有缺陷，但我們現在有更好的認識。從 1990 年代末，清潔程式、測試驅動開發（TDD）、行為驅動開發（BDD）、領域驅動設計（DDD）、持續交付等實踐越來越受歡迎。這些實踐改變了我們交付軟體的方法。

TDD 規範測試先行。DDD 識別業務領域的程式碼與模型設計，打破傳統的模型與程式碼分離。一個結果是我們預期程式碼能說出領域的完整故事。BDD 透過工具支援借用業務語言並讓它更白話。持續交付展示幾年前還看起來很離譜（以非事件的方式進行一天多次交付）但若遵循建議做法則實際上是能做到的，而且是很好的想法。

另一個有趣的事情與時間有關：雖然文學程式設計或 HyperCard 等舊想法未成為主流，但它們還是慢慢的產生影響，特別是在帶進舊想法的 F# 與 Clojure 等新程式設計語言社群中。

現在我們至少可以期待一種實用、保持更新、低成本、有趣的文件製作方法。我們知道傳統文件製作方法的所有問題，也看到必須滿足的需求。這本書討論以更有效率的方式滿足需求的方法，但先讓我們研究文件製作到底是什麼。

文件關乎知識

軟體開發關乎知識與根據知識做決定然後建立更多知識。要解決的問題、做出的決策、決策的方式、決策的根據、考慮過的替代方案都是知識。

你可能沒這麼想過，但打出來的每一行程式語言指令都是個決策。決策有大有小，但不論大小都是決策。在軟體開發中，設計階段之後就沒有昂貴的建構階段：建構（執行編譯器）太便宜，只有（有時沒完沒了的）設計階段才有成本。

軟體設計能持續很長時間，長到足以忘記前面的決策與其脈絡、人們離開而將知識帶走、新加入缺少知識的人。知識是軟體開發等設計活動的核心。

此等設計活動因很多原因而需要團隊。團隊合作意味著一起做決策或根據他人的知識做決策。

軟體開發獨特處在於設計不僅涉及人還涉及機器。電腦是其中一環，許多決策與執行的電腦有關。這通常透過稱為*原始碼*的文件完成。知識與決策以電腦能理解的程式設計語言傳入。

然而，寫出讓電腦懂的原始碼不難，沒有經驗的開發者也能辦到。困難的部分是讓其他人也能懂以讓他們做得更好更快。

野心越大，使知識的累積能超過個人大腦處理能力的文件就越重要。我們的大腦與記憶能力不足時，需要寫作、輸出、軟體等技術的輔助以記得與組織更大量的知識。

知識的起源

知識從何而來？知識主要來自於*對話*。我們透過與其他人對話產生知識。這在結對程式設計、開會、喝咖啡、打電話、群組討論、郵件討論等一起工作時發生，例如 BDD 的規格研究活動（specification workshop）與敏捷的三人行（three amigos）。

但軟體開發者也與機器對談，我們稱此為*實驗*。我們以某種程式設計語言告訴機器一些事情，然後機器執行並回覆我們一些事情：測試成功或失敗、UI 如預期動作、結果不如預期等，然後我們從中獲取新知識，例如 TDD、顯現設計（emerging design）、精實創業（Lean Startup）等實驗。

知識也來自對背景的觀察。你在公司上班，從觀察他人的交談、行為、情緒中學到很多東西，例如領域沉浸（domain immersion）、告示牆痴迷（obsession walls）、資訊輻射器（information radiator）、精益創業的“走出建築物”。

知識來自觀察人以及在可觀察的背景下以機器進行實驗。

知識如何演進？

有些知識長期穩定，有些知識變化的很快。

任何一種文件都必須考慮維護成本並要盡可能接近零。穩定的知識可採用傳統方法，但寫文件並隨著變化而更新對經常變化的知識來說不可行。

軟體產業的加速度效應使我們想要能非常快的讓軟體演進。這種速度讓我們來不及一頁一頁的寫文件，但我們還是需要文件的功能。

為什麼需要知識

建構軟體時，我們研究問題、做決策、然後依據所學做調整：

- 我們要解決什麼問題？大家最好從現在開始搞清楚。
- 我們真正要解決什麼問題（在發現一開始搞錯時嘗試回答這一題）？
- 我們分不清訂單與出貨，但最終明白它們不一樣。以後不應該再搞混。
- 我們嘗試過新的資料庫，但它不符合需求——有三個原因。如果需求不變就不用再試了。
- 我們決定將購物車模組與付款模組分離，因為我們注意到改變其中一個與另一個沒有關係。兩者不應該耦合。
- 我們意外發現這個功能沒有用，因此計劃下個月刪除。但我們可能會忘記為什麼刪除，如果不改程式碼則它永遠會是個謎。

遺失現有軟體在過去發展出的知識時，我們只好重新製作，因為我們不知道以前有什麼。我們也不知道功能與元件的關係，因為我們不知道原來是怎樣，而某個功能的程式碼也四散在各種元件中。

最好是有知識能夠回答下列常見問題：

- 這個問題要改哪裡？
- 這個功能要放在哪裡？
- 原作者會改哪裡？
- 刪除這一行看起來沒用的程式安全嗎？
- 我想要改參數，會有什麼影響？
- 是否只能靠逆向工程來理解它是如何運作的？
- 是否只能靠逐行讀程式碼才能知道目前的業務規則？
- 客戶要求新功能時要如何知道是否已經寫好了？
- 我們已經盡可能把程式改好了，但是否對它的認識還不夠完整？
- 如何快速找到處理特定功能的程式段？

缺少知識會造成兩項成本：

- **浪費時間**：這個時間可以用來改善其他部分。
- **次佳決策**：決策還能更好，或者長期來看更便宜。

這兩項成本會隨著時間複合：花時間找遺失知識就佔用做出更好的決策的時間，然後次佳決策會讓我們的日子更辛苦，直到我們不得不另起爐灶為止。

聽起來讓知識能夠以對開發任務有幫助的方式存取是個好主意。

程式設計是建立與傳遞理論

Peter Naur 在 1985 年於他著名的“Programming as Theory Building”論文中完美的闡述關於集體合作程式設計的真相：重點不在於告訴電腦要做什麼，而是與其他開發者分享透過耐心的學習、實驗、對話、深度反思而產生的理論（“心智模型”）。用他自己的話說：

正確的程式設計應該是對問題具有某種洞察、理論的程式設計師所進行的活動。此建議是相對於程式設計應該是程式與其他文件的製作這種更常見的看法[5]。

問題在於此理論的大部分是看不見的。程式碼只是冰山一角，更多的是開發者的心智中的理論的成果而非理論本身。Peter Naur 認為此理論包含三個主要知識領域：

- 程式碼與其表示的世界的對應關係：具有程式理論的程式設計師能夠說明解決方案與它處理的問題的關係。
- 程式的原理：具有程式理論的程式設計師能夠說明程式每個部分是什麼；換句話說，程式設計師能夠以某種道理支持實際程式。
- 擴展或改進程式的潛力：具有程式理論的程式設計師能夠有建設性的回應任何修改程式的要求，以透過新方法支持要處理的問題。

我們隨著時間學到能讓人們傳遞理論的技術。清潔程式碼與 Eric Evans 的領域驅動設計，鼓勵程式設計師找出以程式文字表達腦中理論的方法。舉例來說，DDD 的統一術語（ubiquitous language）連結世界語言與程式語言，幫助解決問題的對應。我希望未來的程式設計語言能認識到不只有表現程式碼行為的需求，還有產生程式碼的更大的程式設計師的心智模型。

還有實際上嘗試包裹理論的模式與模式語言。我們知道越多模式就越能納入看不見的理論、讓它明顯並擴展。模式在其作用的描述中體現了選擇它們的基本原理的關鍵要素，它們有時暗示應該如何擴展。它們可能暗示了該程式的潛力；舉例來說，策略模式在於以新增策略來擴展。

但隨著我們逐漸加深認識，我們還要處理更大的挑戰，因此挫敗感依然存在。 我相信 Naur 於 1985 年發表的論述在接下來的幾十年中仍然有效：

5 Peter Naur, “Programming as Theory Building,” Microprocessing and Microprogramming, Volume 15, Issue 5, 1985, pp. 253–261.

> 對一個要理解現有程式的理論的新程式設計師來說，有機會熟悉程式文字與其他文件是不夠的[6]。
> 我們絕對不會完整的解決知識傳遞問題，但可以接受這個事實並與之共存。理論在程式設計師腦中形成的心智模型無法完全分享給未參與建立過程的人。
> 結論似乎是必然的：特定類型的大型程式、持續適應、修改、矯正錯誤，取決於一群緊密且持續相互聯繫的程式設計師所擁有的某種知識。
> 值得注意的是經常一起工作的固定團隊不會有太多的理論傳遞問題。

文件製作關乎轉移知識

文件製作（*documentation*）一詞有很多意義：寫文件、Microsoft Word 或 PowerPoint 文件、根據公司範本寫的文件、印出來的文件、網站或 wiki 上長篇大論的無聊文字等。但這些意義將我們限制在過往的做法上並排除許多較新較有效率的做法。

本書對文件製作採用更廣義的定義：

> 轉移有價值的知識給現在與未來其他人的程序。

文件製作有個邏輯。它關乎在時空中轉移知識給他人，技術工作者稱此為*持續存在*（*persistence*）或*儲存體*（*storage*）。我們對文件製作的定義大體上像是貨物的運輸與倉儲，而此貨物是知識。

6 Peter Naur, “Programming as Theory Building,” Microprocessing and Microprogramming, Volume 15, Issue 5, 1985, pp. 253–261.

在人群間轉移知識實際上是在大腦間轉移知識（見圖 1.7）。從一個大腦到另一個大腦，重點在於轉換或擴散（例如傳播給一大群受眾）。從現在的大腦傳遞給未來的大腦，重點在於知識持續存在，這關乎記憶體。

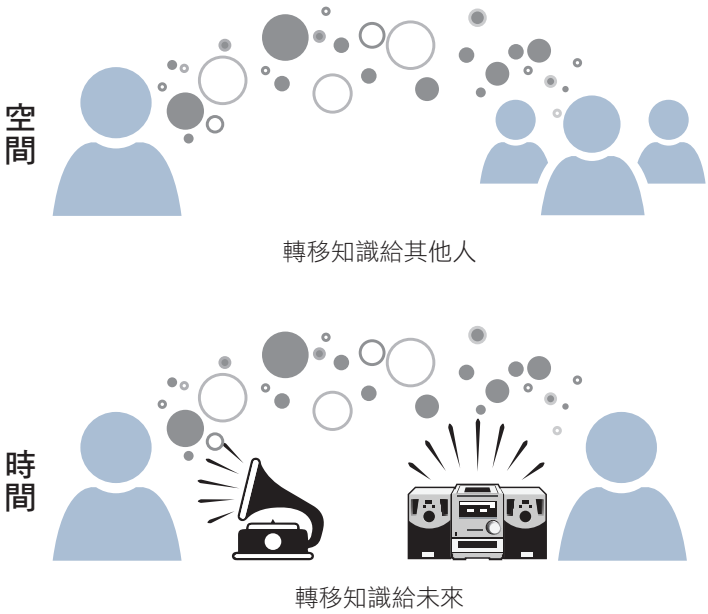

圖 1.7 文件製作關乎轉移與儲存知識

你知道嗎？

開發者的半衰期是 3.1 年，而程式碼是 13 年[7]。文件製作必須解決這個不匹配。

從一個技術工作者的大腦轉移知識到另一個技術工作者的大腦，在於讓知識可存取。另一個讓知識可存取的狀況是讓知識可有效的搜尋。

還有其他狀況，像是將知識轉換成特定文件格式以利匯編（因為你就是得這麼做）。

7 Rob Smallshire, Sixty North blog, http://sixty-orth.com/blog/predictive-models-of-development-teams-and-the-systems-they-build

專注於重點

作為一種轉移有價值知識的方法，文件製作有許多形式：寫文件、面對面交談、程式碼、社交工具上的活動、或不需要時什麼都不做。

我們可以透過文件製作的定義說明一些重要原則：

- 知識是值得記錄的長期利益。
- 知識是值得記錄的大量人群的利益。
- 有價值或重要的知識也必須記錄。

另一方面，你無需在乎不屬於上述項目的知識記錄。為它花時間會是浪費。

知識的重點在於價值。無需花時間轉移沒價值的知識給很多人。常識若只對一個人有用或事後才有意義，則無需轉移或保存。

> **預設不記錄文件**
>
> 除非有合理必要的原因，否則記錄知識的投入是一種浪費。不要為了沒有記錄無需記錄的東西難過。

從轉移與保存知識以及初期應該如何管理文件的觀點來重新思考什麼是文件製作，接下來說明活文件的中心思想與核心原則。

活文件的核心原則

活文件（*living documentation*）一詞因 Gojko Adzic 的 *Specification by Example* 一書而聞名。Adzic 這麼描述採用 BDD 的好處：情境為規格建立且測試也因記錄業務行為而實用。文件因為測試自動化而在測試全部通過時更新。

活文件對軟體開發專案的各方面都有相同的好處：業務行為、當然還包括業務領域、專案願景與業務推動因素、設計與架構、舊策略、程式設計指引、部署、基礎設施。

活文件有四項原則（見圖 1.8）：

- **可靠性**：活文件在任何時間均正確且與交付的軟體一致。
- **低投入**：活文件減少文件製作工作量，包括修改、刪除、新增。它只需要最少的投入——且只需一次。
- **合作**：活文件鼓勵所有參與者交談與分享知識。
- **洞察力**：活文件透過吸引各方面的注意，來提供回饋的機會與鼓勵更深的思考。它幫助反映工作狀況與做成更好的決策。

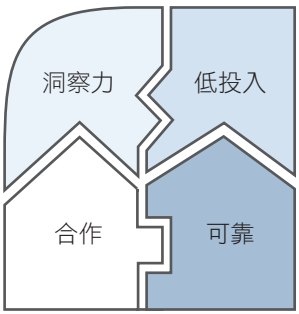

圖 1.8 活文件的原則

活文件也為開發者與其他團隊成員帶來樂趣。他們可在專注於工作的同時獲得活文件。

下一節簡短描述在活文件的四項原則指導下產出最大利益。接下來與後續三章內容會詳述這些原則。

可靠性

文件必須值得信任才有用；換句話說，它必須 100% 可靠。由於人從來都不可靠，我們需要輔助可靠性的紀律與工具。

可靠的文件製作依賴下列想法：

- **利用可用的知識**：大部分的知識已經呈現在專案製作物中，只是需要利用、加強、編輯以供文件製作。
- **正確性機制**：需要正確性機制以確保知識同步。

低投入

活文件在經常變化的環境下必須低投入才可行與可持續；你可以透過下列想法達成：

- **簡單化**：很明顯的，如果沒有需要聲明的東西，則文件是最好的。
- **標準重於自定方案**：標準應該是人盡皆知的，如果沒有則參考外部標準（例如你最喜歡的書、作者、或維基百科）。
- **長青內容**：總是有東西不變或不經常變，這種材料的維護成本很低。
- **重構知識**：有些東西在有變化時無需人力投入。這是因為能自動傳播相關變化的重構工具或知識，本來就在其中並隨著變化。
- **內部文件**：一個東西的額外知識最好跟著它，或盡可能靠近。

合作

活文件必須如下合作：

- **對話重於正規文件**：沒有事情比面對面互動交談更能有效的交換知識。不要不好意思記錄所有討論。雖然我通常偏好對話，但有些知識在長期間對許多人有用。要注意想法隨著時間沉澱的過程以決定什麼知識值得保存記錄。
- **可存取的知識**：活文件實踐中的知識，經常在原始碼控制系統的技術製作物中宣告，這讓非技術人員很難存取。因此，你應該提供工具讓所有受眾毫不費力的存取知識。
- **共同負責**：原始碼控制系統中的知識並非完全由開發者掌管負責。開發者不掌管文件；他們只是負責技術性的處理。

洞察力

上述原則很有用，但實現活文件的全部潛力必須具有洞察力：

- **清楚的決策**：如果不清楚你在做什麼，製作活文件時就會立即露出馬腳。這種反應能鼓勵你明白你的決定，而使得你在做什麼很容易說明。清楚的決策經常會提升工作品質。
- **內含學習**：你想要寫出能讓同事從交互過程中，學習到設計、業務領域、系統其他方面的程式碼與其他技術製作物。
- **事實核查**：活文件幫助顯露系統的實際狀況（舉例來說，“我以為實作不會這麼亂”與“我以為鬍子刮乾淨了，但鏡子顯示出並非如此”）。接受現實狀況與想像中不同也可以幫助改善。

接下來深入說明這些原則，而接下來的幾個章節會擴展至成功施行活文件的相關模式與實踐。但首先要說明啟發活文件的螞蟻，與其他社會性昆蟲的合作與知識交換方式。