01-01 精實的由來

「精實」（Lean）這個詞彙是 John Krafcik 在1988年於他的一篇文章《Triumph of the Lean Production System》[1]裡所首先提出來的，但他所稱的精實製造（Lean production），指的是製造業的精實理論，而軟體界的精實（Lean）則稱為精實軟體開發（Lean Software Development），它源自於 Mary Poppendieck 和 Tom Poppendieck 在 2003 年的著作《Lean Software Development：An Agile Toolkit》，書中闡述了精實軟體開發的七大原則，屬於敏捷開發的成員之一。

敏捷軟體開發（Agile software development）是從 1990 年代開始逐漸取代傳統開發法的一些新型軟體開發方法，是一種應對快速變化需求的軟體開發能力。相對於傳統開發方法，敏捷軟體開發最大的差異在採用「迭代式」的開發模式，而不是一次定江山的「瀑布式」開發模式，以及接受客戶對需求合理的變更（讓客戶對需求做出不同優先等級的區分，並盡力去滿足它）。

敏捷（Agile）一詞來自於 2001 年初，敏捷方法發起者和實踐者在美國猶他州雪鳥滑雪聖地的一次聚會，有 17 位當代軟體代表人物共同簽署了敏捷宣言，並成立了敏捷聯盟。但在此之前，早在 1991 年麻省理工學院發表的一篇《The machine that changed the world（改變世界的機器）》研究報告中，就已經把日本 Toyota 公司的「豐田式生產系統（TPS）」歸納成為一套精實生產（Lean Production）方法。

1. 這裡是依據 Mary Poppendieck 夫婦的名著《Lean Software Development：An Agile Toolkit》對「精實軟體開發」的定義，The term was first coined by John Krafcik in his 1988 article, Triumph of the Lean Production System。

嚴格來說，精實（Lean）比敏捷（Agile）要早誕生許多年，但現在擁戴精實的人士也已經加入了敏捷聯盟的陣營，雖然他們依然遵循著精實精神的七大原則而不是敏捷的四大宣言和 12 項原則，但實質上他們都共同擁護敏捷式的開發方法及精實精神，二者並無牴觸。

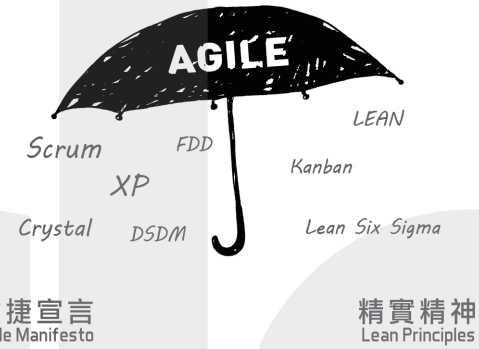

圖1-1 敏捷傘下的兩大陣營

01-02 精實軟體開發

「精實軟體開發」並沒有具體的開發方法或步驟，而是一堆原則，原因是它認為沒有所謂的最佳實踐。「原則」具有較廣泛的普遍性，能指導對某一學科的思考和領悟，而「實踐」則是為執行原則而採取的實際措施，需要針對某一領域進行調整，尤其必須考慮到具體實施的環境。精實軟體開發是由軟體開發領導者，例如：軟體開發部經理、專案經理和技術領導者，而不是一般程式開發人員所創設的思想工具。

因為精實軟體開發沒有具體的實行方法，這會讓你覺得它只是一些原則、教條，執行起來應該是最簡單的、影響不大，即便做錯了也是無害的那

種。如果你這麼想的話就錯了，因為「原則」所影響的是企業的文化層面，比起單純的開發方法影響要巨大多了。

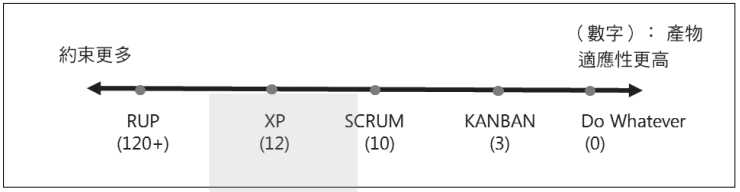

圖1-2 依照規範的多寡由左至右排列各種開發方法[2]

依上圖的區分，右邊第二位隸屬於精實開發體系下的**看板方法**（Kanban），是距離**胡作非為**（Do Whatever，也就是完全沒有規範）最接近的一個敏捷開發方法。越往右側的開發方法就表示規範越少，我們稱為輕量級（light weight）的軟體開發方法，越往左邊的開發方法則是規範越多，相對於輕量級的開發方法有較多的約束，我們稱為重量級（heavy weight）的開發方法，例如：RUP（Rational Unified Process，統一軟體開發過程）。

本章的主旨在闡述，如何將精實的精神由原則轉換為適用於特定環境下的敏捷實作。說得更精確些，就是針對七大原則作實務的詮釋，目標是**看板系統**，尤其是那些依靠經驗法則才能換來的經驗知識[3]。

2. 這張圖是在描述軟體開發方法的規範多寡，雖然圖上的幾種規範都不承認自己是一個開發方法，例如：Scrum 稱自己只是一個開發的框架，而 Kanban 則稱自己是流程控制的方法，但我們還是統稱為軟體開發方法。
3. 「經驗知識」指的是人透過經歷該事件後所獲得的知識，而我們從小接收由學校教育所灌輸的知識，則常常被稱為「套裝知識」。

圖1-3 在Curation Web的時代裡，充斥著各式各樣的物件

在沒有使用過之前，你實在很難去判斷是不是用錯了元件了？

01-03 精實軟體開發七大原則

以下為精實軟體開發的七大原則：

1. 消除浪費（Eliminate waste）
2. 增強學習（Amplify learning）
3. 盡量延遲決策（Decide as late as possible）
4. 盡快交付（Deliver as fast as possible）
5. 授權團隊（Empower the team）
6. 嵌入完整性（Build integrity in）
7. 著眼整體（See the whole）

乍看之下，你可能覺得這些原則感覺上像口號一樣，真的有用嗎？讓我告訴你，當你在繪製**看板**時（也就是將你的工作流程放到**看板**上成為一個或

多個垂直欄位時），你所依據的便是對這幾條原則的了解程度。如果你覺得很陌生的話，下次改變看板外觀時，一邊看著這些條列一邊思索是否需要改善哪裡？改的原因是甚麼？想達成哪一條原則？多練習幾次就熟了。記得一次只改善一個地方就好，這樣比較容易看出是哪個異動所造成的結果，這一點跟改 bug 是一樣的，一次同時修改好幾個地方，就搞不清楚誰才是元兇了！

1-3-1 消除浪費（Eliminate waste）

何謂浪費？只要是對客戶或產品沒有提升任何價值的行為，基本上就是一種浪費！

Bug 是第一大浪費

程式開發人員最大的浪費，便是在開發作業時製造一大堆 bug，然後再費盡心力把它解決掉。有趣的是，解決這些 bug 之後還能獲得相當的充實感！反倒是很少人會回過頭來檢討，這些 bug 實際上都是自己造成的。會有這些 bug 產生，其實是軟體的複雜性所造成的，是我們把程式寫複雜了。而如何減少 bug 呢？就是想辦法把程式寫簡單一點，只有很簡單的程式，bug 才會相對減少。如果程式複雜了，最後便只能靠測試來守住品質了，這一點也間接地說明，開發和測試實際上是一體兩面，開發者必須負起減少 bug 的第一線任務，因為它才是最大的浪費。

現在的程式開發作業太複雜了

開發軟體最重要的是知道要做甚麼，然後去做，就這樣了！

但經過歲月不斷的累積之後，我們把這個過程變複雜了。是那些有智慧的學者不斷地把經驗奉獻出來，針對各種不同問題提出解答（設計模式便是這樣誕生的），智者怕我們重蹈覆轍便想辦法把經驗累積下來，原意是為了照顧後進，結果卻是把開發作業越弄越複雜（HTML 的演進史就是這個縮影）。十年前的軟體開發專案，經過十年後規格並沒有太大改變，但我們卻可以把它弄得越來越有學問，看起來越專業，專業到必須有相當的學習過程才足以開發十年前就能做到的事！程式在執行的速度上變快了，但是在品質這一點上卻始終沒有太大的提升，原因是我們把自己弄複雜了，我們一再地把開發程序跟門檻弄高了，而複雜所帶來的最大罪惡便是浪費，所以消除浪費便成了近代工程師要學習的第一要務。

「簡單」是對付 bug 的有效法則

想要減少 bug，就是把程式弄簡單些讓自己隨時都能看得懂。開發軟體時，bug 總是自動在過程中被隱含進來。通常，一知半解進行程式撰寫是製造 bug 的最大元兇，這種 bug 最難以被偵錯出來，再來則是邏輯思維被打斷也是在程式撰寫中很容易產生 bug 的時候。所以在進行工作拆解時，最重要的是「簡單化」，簡單是減少 bug 的最佳處方。話雖如此，但很多時候軟體開發就是這麼複雜，該如何是好呢？

「**錯誤的估算**」便是一個簡單不下來的原因。千萬不要在沒有做適度的拆解問題（工作項目）下進行時程的預估，因為那完全是在猜猜看！猜是人類最糟糕的預估了，最少也要有參考依據（有參考依據可以讓預估準確許多，例如：找到可以做比較的案例），但是必須經過拆解成為較小的工作單元，參考才足以成立。所以在減少浪費的前提下，「先拆解再簡單化」是開工之前（或是進行工時預估前）的必備動作，正確的拆解可以避開那些不必要的複雜性干擾。

專案經理（PM）習慣向開發人員要求預估需要多少開發時間，想藉助工程師每個人的預估，然後一一加總起來，以得到團隊的整體開發時程（當然再加上一點自己習慣性的保險時間）。這是一種將專案分解成多個區塊，然後針對各個區塊進行預估的作法。這樣所估出來的時程乍看之下是會比較準確些，但是卻忽略了工程師本身所估算的數據本來就是基於一種猜測得來的數值，根本上就已經不準確了。所以，雖然已經經過拆解，但這是基於工程師個人的猜測而來，當然就沒有太大的意義。

甚麼樣的估算才會比較準確呢？老實說，只有進行一段時間，有更深一層的了解後再來估算自然會準確許多。這種較準確的估算通常發生在專案進行五分之一到三分之一之間，這是一件耐人尋味的事，此時工程師對於專案的把握度就可以大幅提升，這個時候的預估就可以接近「承諾」了。

工程師的承諾與預估

專案開始時工程師無從參考比較，此時的工時估算應該只能說是猜測；一旦專案開始進行後，隨著對專案的了解增加，通常工程師在開發作業進行到五分之一到三分之一之間，由於對任務越來越熟悉了，自然就可以做比較有把握的預估了，這個時候的估算就可以稱之為「承諾」了。

寫程式想要減少 bug，專注（Focus）是最大良方。這裡討論的專注是指「短時間」的專注，而不是廢寢忘食、長時間瘋狂在做某一件事的專注。短時間指的是 25 分鐘的專心作業，這一點請參考**蕃茄工作法**[4]。

4. **蕃茄工作法**是簡單易行的時間管理方法。蕃茄工作法是弗朗西斯科・西里洛於 1992 年創立的一種相對於 GTD（Getting Things Done，一種行為管理的方法）更微觀的時間管理方法。在蕃茄工作法一個個短短的 25 分鐘內，收穫的不僅僅是效率，還會有意想不到的成就感。

如何識別浪費？

豐田生產系統的策劃人之一新鄉重夫（Shigeo Shingo），他首創製造業的 7 種浪費類型，而 Mary & Tom Poppendieck 則將它轉換成軟體業的七種浪費類型，對照如下：

	製造業7種浪費	軟體業7種浪費
1	庫存	半成品、部分完成的工作，Partially Done Work
2	額外過程	額外過程，Extra Processes
3	生產過剩	多餘功能，Extra Features
4	運輸	任務調換，Task Switching
5	等待	等待，Waiting
6	移動	移動，Motion
7	缺陷	缺陷，Defects

判別是否浪費十分重要，它是你避免浪費的基礎。接下來的說明雖然看起來冗長，但請務必讀完，每個項目的最後會括弧說明相對於**看板方法**的運用手法，譬如你不知道該如何新增看板的垂直欄位或調整 WIP（半成品）值，即可參考以下的幾條原則做依據。

■ 部分完成的工作

「半成品」的英文是 Work-In-Process（WIP），雖然翻譯成「在製品」看起來比較貼切，但我偏好採用「半成品」這個字眼。所謂「部分完成的工作」，它是一種賭博，一個隨時可能會失效的功能，因為它有可能還沒上場就被換掉了（原因是你提前做了）。另外是太早做可靠度就差一些，雖然你可能用單元測試來保證它的輸出入值，但在還沒有經過整

合之前，沒有人可以保證它是好的，所以以投資報酬率而言，它是最不理想的半成品。在團隊開發流程中，嘗試把半成品控制在最小範圍內是最理想的狀態，也是減少浪費的好方法。（對**看板方法**而言，可以用來限制WIP值，產生盈餘時間[5]或是看到瓶頸所在，這一點可以透過**累積流程圖**來進行觀察）

■ **額外過程**

文書工作就是一種最有爭議的額外過程，它會消耗資源、降低反應速度、還會隱藏風險；但相對的它可以讓客戶簽字表態、或是取得變更許可、或是便於追蹤。幾乎所有的軟體交接都需要文件，但是基本上它是不會產生增值的，所以也是一種浪費，但若是基於需求上的工作，則可以把它視為是一種增值了。請注意，好文件應該保持簡潔、具有概念性、便於做交接。（**盈餘時間**是最適合用來進行文件的撰寫作業了，工程師要學會交接文件給自己）

■ **多餘功能**

有句古老的名言：「當你新增功能時你也同時新增了 bug」，意思是盡量減少不必要的功能！一般的程式設計師總以為突發奇想的功能是一個不錯的想法，為了未雨綢繆就自作主張把它加進來了，老實告訴你這是最不好的做法，記得，只有在有必要的時候才新增功能，任何一段不需要的程式碼都是一種浪費，千萬要懂得抵抗自以為能夠有先知卓見這種未雨綢繆能力的誘惑。（額外的程式碼將會佔據半成品的數量限制，透過**累積流程圖**可以很清楚地界定它的影響）

5. 盈餘時間是**看板方法**中非常重要的一個觀念，也是我做顧問時必用的手法之一，下一章我們就會談到。

■ **任務調換**

多工是一種浪費[6]，給員工同時間分配多種工作是專案產生浪費的一個根源。軟體開發人員每次在轉換工作時都會浪費大量的調換時間，因為他必須調整思路以便投入新的任務流程。當然，若是你同時參與多個開發團隊的話，自然會造成更多的停頓，從而引起更多的任務調換而浪費更多時間。（限制 WIP 值就是為了減少這種浪費）

■ **等待**

在團隊進行協同開發時最浪費的可能就是等待了。有時候等待上級或協力廠商的回音，例如：等待 email 的回函或通知，這種必然會存在、無法改變的過程，就是一種浪費。（這種必然的等待可以在**看板**上增加一個垂直欄位特別來處理它，如此可以讓我們更明確知道現在的狀態在哪裡了？）

■ **移動**

當開發人員遭遇無法立刻處理的問題、需要他人協助時，他需要移動多少距離才能找到問題的答案？是否立即就能得到解答？還是需要繼續移動到其他房間詢問其他人的協助呢？這就是一種浪費。當然人不是唯一會移動的，各類文件也會移動，尤其是測試文件的流動程序，也可能造成龐大的浪費。（這種事務型的開銷，在**看板**上面能夠表現得越明顯，就越容易被考慮進去解決方案）

■ **缺陷**

缺陷也常常被翻譯成漏洞，是指在軟體執行中因為程式本身有錯誤而造成的功能不正常、當機、資料遺失、非正常中斷等現象。缺陷（bug）

6. 多工是不好的事（Multitasking is evil），可能是專案開發的頭號殺手。

所造成的影響，取決於它被發現的時間和它的大小，越早找到缺陷越好，如果能在需求撰寫階段就被測試人員發覺，是最好不過的時機了。（文件審核有時候可以盡早找到缺陷）

能夠盡早處理掉問題絕對比在客戶端才發現好上許多，因此就有人發明，從一開始寫程式就開始做測試的方法，稱為「測試開發法（Test Driven Development，簡稱TDD）」，就是希望能盡早找到缺陷並即刻修復的撰寫程式手段。類似的方法（ATDD、BDD等）大部分也都有它的價值，也都在市場上佔有一定的比重，但是軟體界尚未找到一種足以被大部分程式設計人員完全接受的開發法。所以借助頻繁的整合及發佈是一種較容易受到客戶肯定的行為，也是避免重大浪費的必要小浪費。

軟體開發是一門藝術還是一門科學呢？由程式誕生的方式到我們除錯解決缺陷的方式，藝術的成分還真是佔據蠻大的比例，因此軟體開發是一門工藝是目前較被接受的一種說法。也就是說，我們很難完全不倚靠經驗來開發軟體，豐田式的生產方式對軟體界而言仍是一種夢想，而我們也只能在類似的精實原則上盡量去學習了。

看到浪費

浪費只要被看見了就容易改善。透過識別哪些行為是浪費，它們是如何造成浪費，可以讓我們更容易看到真正的問題所在，也就能「消除浪費」。看板方法能讓問題成為大家都知道、都看得見的事，如此便可以透過改善的方式（行為）來避免不必要的浪費，這是精實軟體開發法的第一個原則。不浪費本來就很合理，但在運用上，由於軟體開發有別於製造業，因此受爭議之處特別多，宜視環境時時調整。下圖是一個遊戲開發公司的開發流程圖，把大的開發步驟以及這些步驟所花的工作時間用時間軸畫出

來，很容易就可以看出對開發作業沒有增值的部分，也就是浪費的地方。

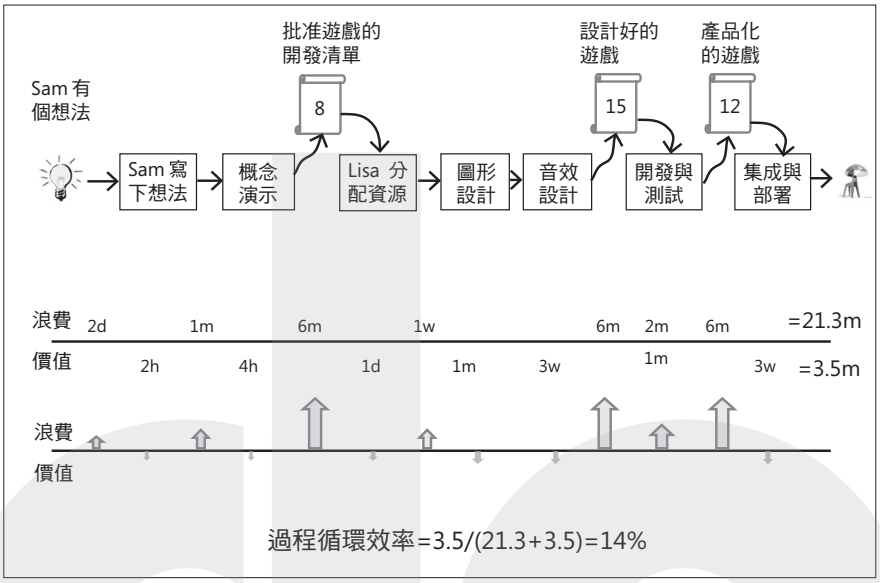

圖1-4 遊戲開發作業的流程圖

圖下方有二條時間軸表示浪費及增加價值的時間數值。上面一條是明確的時間值，下面一條則是指標式的示意圖（圖中用了二種不同的圖示，目的在闡述數字大小代表的只是一個約略值，在某些時候用比例來區分可能更合適些）。整個流程的統計結果顯示在右側，浪費的時間為 21.3m（月），而真正拿來增值的工作是 3.5m（月），二個數字相除以後得到產出率為 14%。一旦透過分析讓人們看見浪費的根源時，通常你就會開始產生一種想要改善的衝動，這便是視覺化後的一種催化效益。

軟體是知識型的開發活動，對於浪費的界定要比製造業複雜許多，所以請先判斷它是「直接的浪費」或是「間接的浪費」。所謂「直接的浪費」指的是沒有其他效益的活動，它純粹就是浪費工時，我們很直覺地可以判斷的浪費行為。「間接的浪費」是除了產生活動還有其他效益的活動，例

如：增加團隊溝通協作的活動，我們便可以稱它為間接浪費的工時。因此在我們考慮消除浪費的時候，間接浪費應該被排在最後，屬於非必要時無需消除的範圍，因為一旦你消除了間接的浪費活動，也就失去了它相對所產生的效益了。

1-3-2 增強學習（Amplify learning）

軟體開發是一種學習的過程。也就是說，開發人員學得越快越好，所撰寫的程式才可能越正確，對客戶也越有利。因此程式設計人員從一開始就要下定決心把事情學好，然後再運用學會的專業知識來輔助我們撰寫正確的程式。

有趣的是，開發過程也是一種發現的過程，我們經常在創作的過程中有了全新的體驗，所以撰寫程式它不全然是一種學習。很多時候，經驗可以幫助我們學得更快更好，但創作則不只要依靠經驗，專注力可能是最不可或缺的。

科學方法尤其適用在解決複雜的問題

科學方法是透過觀察、建立假設、設計實驗、進行實驗、然後得到結果。有趣的是，如果你的假設越正確，你就不會學到太多東西。當失敗率達到 50% 時，你會得到最多的訊息，也就是學到最多。

工程師撰寫程式時不也是如此？如果一次就做對了，可能完全沒學到任何東西，只是在工作而已。傳統的開發方式正是要求大家透過審慎的態度一次做對，而敏捷開發則是鼓勵透過嘗試、測試、修正的短週期來開發程式，自然你會學到最多。

寫文章時常要修改個幾回才能成章，何不讓寫程式也如此

其實寫文章比起寫程式要難上許多，但有一個最大的差別，文章的 bug（錯字）很容易發現，但程式的缺陷卻很難找到。就價值觀而言，我們可以很容易地以程式的應用範圍來衡量其價值，但相對的寫文章就很難評價了。二者在抽象的程度上很不相同，一個是越明確我們認為越好，另一個則是越抽象反而讓人覺得越是受用無窮。

最小的成本產生最多的知識

既然程式開發是一個學習的過程，為了得到好的成果，學習善用最小的成本獲得最大的學習效果，應該是程式設計人員不可缺少的技能。例如：如果測試成本很高，就多花些時間仔細思考、審慎檢查後再動手，如果實驗的成本很低，那它就是最有效的方法了！

短暫的學習週期是最高效的學習過程

一種高效能的學習方法名為週期性的重構，也就是一邊開發系統的同時也在進行改善設計方案的動作，這是我們產生知識的最好途徑之一。衍生式的設計方式又稱為浮現式設計（Emergent Design），它是架構設計師在採用敏捷式開發法時所遭遇到的最大困難（因為不能做太多的預先設計，必須有問題做對應時才能做相對的設計），既然我們不能一口氣就把架構設計完畢，那只有透過堆疊的方式，讓問題來引導架構的途徑。但千萬別孤獨的讓問題來引導架構，因為設計模式正是為了解答那些重複出現的問題所誕生的最佳解答。（其實是沒有最佳解答的，說穿了應該只是最佳參考罷了，因為沒有銀子彈，所以你必須在研究清楚環境之後自己來，這也正是所謂的高效學習的過程。）

測試是最好的回饋

傳統一次性通過的開發方式（single-pass model），是假設一開始便可以把開發的細節想清楚，針對每一個需求都一視同仁一樣重要，按部就班把所有的需求都做完，所以也就不會有太多的回饋，也失去了反覆修正的機會。傳統開發反而害怕回饋所帶來的學習會破壞了原先預定的計畫。

就是這種思維造成學習被延後，直到最終測試的時候才出現，只可惜為時已晚，只能期待下一次的專案能夠完全一樣，這個學習的所得就能派上用場了。所以按照道理應該增加回饋，因為回饋是處理軟體開發上遭遇問題時最有效的方法之一。以下是增加回饋的幾點原則：

- 即時回饋，在寫完一段程式碼後立刻進行測試。
- 運用單元測試來核對程式碼，而不是用文件來記錄程式邏輯的細節。
- 透過向客戶的展示收集回饋以及變更的需求。

團隊同步學習

短的迭代循環是團隊共同學習的最佳方式。同步是團隊開發非常重要的一個步調，尤其當有一些項目出現測試失敗後，所遺留下來待解決的 bug 經常會讓共同開發的同仁產生不同步的紊亂感覺，彼此之間誤以為需要等待才能繼續下去，造成學習受到中斷的浪費。而迭代讓大家有一個共同的起點，能夠促使團隊不斷的誕生新的學習機會，因此維持同步與採用短週期的迭代是團隊學習的基本需求。

善用共同開發工具

一定要善用數位資訊。無論是微軟的 TFS 或是 Open Source 的 Git，在 Web

的運用上都具有絕佳的協同合作特性，這一點讓程式碼不再是一個人的私人收藏，而是屬於團隊共同擁有的資產，大家都能透過訊息相互學習。透過這些好的數位工具，不論是 code review 或是程式碼的 check in/out 都成為公開的行為，這樣的同步性可以造成更有效率的集體學習行為。

拜行動裝置普及之賜，訊息傳遞總是超乎想像的快捷，對於工作流程上的事件前置時間（Lead Time）有了相當的改善，使得運用流程來提升團隊效能成了這一代最重要的課題，因此能否善用數位學習也變成分辨優劣的關鍵因素之一。

愉快的心情是改善學習環境的重要因素，對於個人是如此，對於團隊更見效益！沒有比在心情好的時候更能充分吸收知識的了，保持愉悅是一種成功學習的祕訣，所以在每日站立會議時進行鼓舞士氣、提升團隊和諧的行為，對一天的工作絕對有它的提升效益，你應該嘗試看看！相對的，心情低落是學習的一大障礙，管理者應該立刻給予關注才是。

1-3-3 盡量延遲決策（Decide as late as possible）

對流程而言：

等到真正需要做改變的時候再做決策，
提前的變更只會增加無形的成本。

對人而言：

等到要做決策所需的資訊較充分了，
再來做判斷會比較正確。

適當等待是作出好的決策時不可缺少的行為

敏捷開發為了處理需求不斷變更的前提下，**鼓勵把決策的下達延到最後可以容忍的時間點再做決定**。例如，印表機廠商因出貨國家的電源種類不同所產生不同的庫存種類而無法善用庫存而傷透腦筋，如果採用對生產線最便利的方式，便是依據運往地點做直接配備該出口地點的使用電源。但如果採用決策的下達延到最後的概念，就是等貨到了該國度之後再由銷售門市依客戶的選擇把電源線加上去，就可以避開這個複雜的出貨問題了。

對個人而言，**延遲決策是為了避免在早期資訊還不夠清楚的狀況下，就迅速做出決定，或是進行評估作業，這會是一種浪費，因為可能會有很多東西之後還要修改或是重做**。

平行開發（Concurrent Software Development）

為了同步作業我們可能需要等待，而等待所換來的則是「順序性」的同步作業。作業原本就可以採用非同步、並行的方式完成，我們之所以沒有採用是因為太複雜了，就為了讓工作可以在同步與非同步之間做切換，僅僅在處理那種複雜機制所損失掉的力量可能就會大過並行工作所能取得的收穫！但那是多年前的說法了，多核心的 CPU 創造了新的並行開發程式語法，而簡易的語法造就了更快速的運算效能。

敏捷開發採用短週期迭代的方式來降低風險，就如同非同步的語法也僅僅是在小範圍的相同理論上運作一般，雖然多工礙事（Multitasking is evil），但在受控制的小範圍內實施卻是利多於弊的。因此決策的時間往往需要參考到並行作業的結果，所以盡量將決策延遲到訊息較明確的狀態再下達是較高效的一種做法。

最後負責時刻（The last Responsible Moment，LRM）

最後負責時刻並不等於盡量延遲決策。這是豐田精神轉換到軟體開發上較有爭議的一個地方，「盡量延遲決策」在針對生產線上的決策關鍵時間有相當正面的意義，並無爭議，但對應到「最後負責時刻」這個並不是那麼正面的用詞，就讓人十分迷惑。

最後負責時刻是指，當你再不做出「決策時，不做決策的成本就要高於做出決策的成本時，就稱之為「最後負責時刻」。

明顯的，最後負責時刻是針對在作業上可以獲得較高的靈活性。這個詞彙最初是由精實建築協會（www.leanstruction.org）所提出，並經常在敏捷開發上被引用來處理尚不明確的需求，建議決策宜延遲到最後一刻，也就是狀態較明朗的時候再做決策。延遲所獲得的時間，則讓決策者有較高的靈活性可以拿來處理其他事情。

談決策：先深入（Depth-First）或是先廣度（Breadth-First）

在處理人工智慧的程式裡頭，你總是會遇到這種應該先深入問題的探討呢，還是先廣泛搜集更多類似問題再逐一深入探討的決策性問題。舉例子來說，假設我們要尋找一把放在大樓某個房間裡頭的手槍，只知道它被放在一棟四層樓高、每層有 5 個房間的建築裡，試問你要怎麼尋找它？是先把一層樓都找完之後再換樓層呢？還是找一個房間後就換一個樓層？你要如何來設計這個邏輯決策呢？

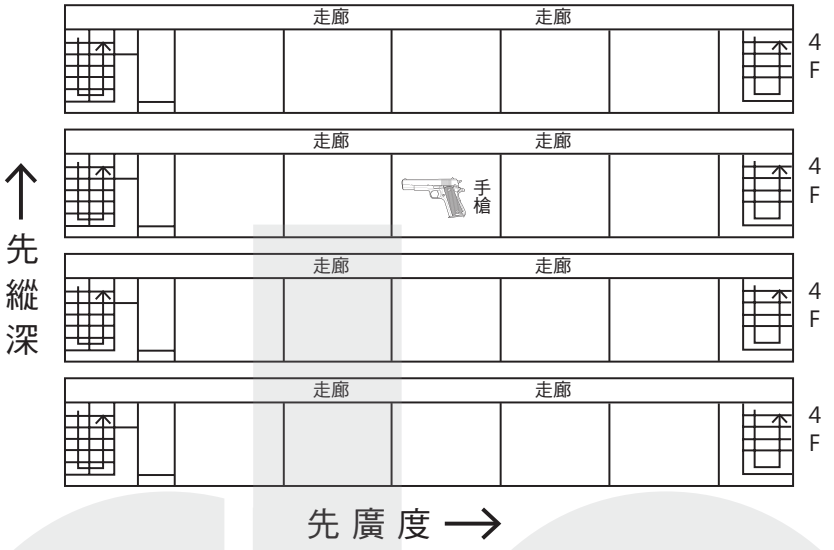

圖1-5 處理人工智慧的程式思考

先深入（Depth-First）的做法就好比將一個樓層都找完再換另一個樓層，它的優點是簡單，你可以很早就下定決策，接著照著做就是了，它可以迅速降低問題的複雜性。缺點是它會過早縮小各種可能性的研究，例如先探討每個樓層會有幾個樓梯？每隔幾個房間就有一個樓梯之類的解題參考。

廣度優先（Breadth-First）則會延遲決策，當搜尋到有多個樓梯在樓層與樓層之間時，就有機會再擬定新的尋找策略。初學者通常會採用**先縱深**的方式，等到熟悉後有經驗後再做其他嘗試。**先縱深**或是**先廣度**這二種方法沒有絕對的對或錯，經過學習後的再進化才是重點。但不論採用哪一種策略，先搜集相關的專業知識都是不可缺少的（參考上面找槍的例子，設計者通常都會留下一些蛛絲馬跡，讓有經驗的搜尋者做判斷的依據，目的當然是讓使用者能夠迅速透過學習來累積經驗，期待下一次會有更好的表現）。

直覺式的決策

最後我們來談一下相對於延遲決策的快速決策：**直覺式的決策**，這是一種類似「急診室的分類法」，先做簡易分類後立刻做決策（括弧中是處理等候的時間）：

- 第一級：**復甦急救**（須立即急救；例如車禍大量出血、意識不清）
- 第二級：**危急**（10 分鐘；例如車禍出血，但生命穩定）
- 第三級：**緊急**（30 分鐘；例如輕度呼吸窘迫、呼吸困難）
- 第四級：**次緊急**（60 分鐘）
- 第五級：**非緊急**（120 分鐘）

進入急診室後，請向醫護人員詢問您屬於第幾級傷病？需要等候多久？

通常救護人員、消防人員很少做決策，他們直覺地依據守則（通常是經驗）走，而非理性決策過程！理性決策會先做分解問題、移除上下文、應用分析技術、進行討論等步驟，這種理性的過程會在有意無意間忽視人們的經驗，而經驗正是敏捷開發最有利的參考。如果能依靠理性分析指出甚麼地方存在不一致，甚麼地方忽略了關鍵因素，固然很好，但他遠遠不如直覺來得有用，因為理性分析移除了上下文之間的關係，所以不太可能觀察到嚴重的失誤，這就是經驗最有價值的地方。**培養經驗最有效的方式就是：透過閱讀把別人的經驗變成自己的知識。**