Chapter 3

XP的團隊實務

團隊實務及其替代方案的探索。

以下要探討的是“團隊”實務，因為若是將這些做法獨自施行，將沒有任何作用：如果只有一個程式師做到經常整合或堅守編程標準，好處也是很有限的。

- 程式所有權：當程式需要需要改變時，誰有權改它？
- 整合：團隊如何確保、何時確保每個人的程式兜在一塊時可以運作？
- 加班工作：當時間『燒光』時，你會怎麼做？
- 工作環境：團隊的位置應該怎麼坐？
- 更版時程：更版的頻率為何？
- 編程標準：程式應該怎麼寫？

程式所有權

當程式需要改變時，誰有權（或必須）改它？

沒人要改（“孤兒型”）

在某些團隊裡，沒有人擁有程式碼，對於那些不開放原始碼、由 third party 所維護的系統而言，“沒有人”意謂著“人不在這兒。”另外還有一些程式的原始碼是觸碰不到的：原始程式幾年前就不見了，或者，系統脆弱、複雜到無法改了它而不出事。

當程式碼沒人管時，其它開發者會避開它、或將之視為一個黑箱。視為黑箱這種態度是很傷的，進出這個黑箱的資料都必須符合某種固定的格式；開發者也必須在兩種思維模式之間轉來轉去（使用 screen-scraping 技術[1]）；一些你原本認為應該是可以做的也都不能做了；資料也可能根本沒有做正規化，最後，還會有資料一致性的問題，因為系統內外都存有一份資料。

這種“程式沒人管”（或沒人想管）的狀況必須極力避免。

最後碰的那個人要改（“貼標籤”（Tag）或“大風吹遊戲”（Musical Chairs））

你的系統可能相當穩定，但維護上仍然很痛苦，這種狀況下，通常組織內部文化就會形成一種慣例：最後碰的人要負責接下來發生的問題（而且令人洩氣的，你很難逃得掉這種慣例），不然就是把問題丟給菜鳥（通常是以“反正你就給我弄好”的命令為之）。

看來，這種維護的模式相當常見。

一個蘿蔔一個坑（“一夫一妻制”）

這種模式在新開發的專案中很常見：每個程式都歸某個“作者”管轄，一直到另一位作者接手為止。

但重點是，每段程式碼都有專人控管的結果，就是如果你想改變那段程式碼，就必須跟那個人協商，雖然，他可能會讓你改，但卻保留准許你可以改成什麼樣子的權力。

有時，這種擁有權會失效，因為負責的人離職了，那部分的程式碼也就不怎麼有在（變）動了。之後，那段程式碼就會一直都沒有真正的主人，直到某人需要完成某件事，團隊才會再指定一位負責人管它。

這種做法的好處是，有個清楚的方法可以決定哪些人負責哪些事，而且可以讓一個人在某個特殊的領域發展出他自己的專長。

然而，也有若干缺點，如果需要改變時，負責人找不著，團隊開發的速度就會被拖慢下來了（有些團隊藉由緊急支援的方式來減輕這種衝擊）。有時候，負責的人不同意你提出的修改計畫，或許根本完全就反對呢！這時，你可能就要採取迂迴設計（design around）的方式，或把這個黑箱以某種方式包裝起來，儘管這種包裝對系統有害。

還有一種問題，管的人可能遇到困難卻隱瞞事實（在他缺乏技術這件事被注意之前，程式碼可能就已經有很嚴重的問題了）。

這種程式碼只歸一個專人控管的做法，可能會需要把介面早早就固定下來，因為要讓開發者“有所進展”，但介面此時卻尚未被大家真正了解。重整也會變得更難進行，因為介面都已經公開，所以需要更多的協同合作方能為之，這時不免會產生『政治上的壓力』，迫使我們不去變動這些程式碼（即使它們的確需要變動）。

前後期所有權（Sequential Owners）

（“ 租賃的所有權” 或“ 分期式的一夫一妻制” ）

有些組織的做法，則是一直都有專人負責的制度，但這個『專人』，是視工作看時期而決定的，並非永遠都是同一個人。

這種做法的好處是多樣性，缺點則是人們可能就沒有機會培養出足夠的技術讓自己很有效率。但至少還是有專人負責，所以每個人都會知道要找誰。不過，最大的問題是，程式碼就像是租來的所有權一樣，目前的“ 擁有者” 實上是“ 房客” ，他們知道這一切都是暫時的，所以比較不會去顧慮這些程式碼的長期價值（特別是如果他們可以在程式碼變得很糟之前“ 退租” 的話）。

分層擁有權（“ 部落制度” ）

還有些組織，把他們的軟體組織得層次分明，甚至連應用軟體也分層級，然後以此為擁有權的基礎。同層次（部落）的每一個人，都可以依自身的需求任意修改軟體，但必須告訴大家他做了些什麼。但其它部落的人卻不敢跳進來（你可以把這種分層想成是使用者介面、企業邏輯、資料庫等三層，或應用軟體甲、應用軟體乙、資料庫）。

這種方法克服了“ 一夫一妻制” 的某些瑕疵：“ bus number” 會比較高、以及不良的程式碼沒有機會藏匿在分組團隊裡等等（“ bus number” 指的是一個人數，如果那麼多的人『同時出事』[被...巴士撞到]，專案就會失敗）。

分組團隊可以發展出非常強烈的團隊精神，這種團隊精神會進一步的提升其生產力。

然而，層與層之間的介面卻會提升到一種近乎“金科玉律”般的狀態，使得它們更加難以撼動。對於想要改善系統總體設計的重整動作而言，這是一種妨礙。

同一組人的溝通往往非常順暢，但跨組之間卻更為困難。在“單一管理者”模式下要說服一個人的改變動作，現在變成是“我們兩人的組員開個會討論討論如何；下星期大家都有空嗎？”這勢必拖慢整個團隊的速度。

集體所有權

（程式共有，每個人均可以動到任何一部分的程式碼）

在程式共有的模式下，整個團隊擁有整個程式碼，任何人都可以依需要改變任何一部分。

這種做法的主要好處是阻礙最少，事情可以快速完成。團隊重整的能力會改善，因為介面可以隨時依需求而改變。如果團隊願意常常重整的話，這種模式可以讓他們非常有『戰力』(下一小節描述 XP 如何試圖保有這種模式的好處，而又不致成為這種模式風險下的犧牲品)。

當然，程式共有也有一些缺點：

- 有些人會以自己的程式為傲，而不希望別人去碰那些程式。
- 你會冒上“共同的悲劇”（tragedy of the commons[2]）這種風險：“每個人都負責”最後可能變成“沒有人要負責。”
- 跟“大風吹遊戲”一樣，但更加劇，風險也許會惡化到沒有人發展出真正的專業，沒有人在乎長期的價值。
- 別人的程式碼讀起來很難，要真正用上也不容易，最後你得到的，可能是一團亂的程式風格跟演算法。
- 由於要存取共同的程式碼，每個人被別人“拖累”的機會將增加。

XP採取集體所有權的做法

XP 能體認到上述這些風險，但還是採行了程式共有的做法，它的一些實務試圖把這些風險減輕：

- 以自己的程式碼為傲：XP 不怎麼在意這種事，不過大概會把這種自豪拉高到團隊的層次。
- 共同的悲劇：搭檔編程加上百分之百過的單元測試程式，有助於確保沒有任何人可以偷偷的“污染”程式碼；搭檔的不斷重組，則有助於讓所有程式碼透明化；重整可以清理掉任何已然發生的問題。

- 不夠專業：搭檔編程把知識散佈到整個團隊；開放的工作空間能讓人們在其它人遇到瓶頸或擔憂做不出來時言無不盡；簡單的設計可以讓我們不需要太"艱深"的專業；最後，測試程式可以確保我們做出的功能不致偏離原先的需求。

- 別人的程式碼：搭檔編程所產生的共享文化、以及編程標準，均有助於減少這類問題。

- 被別人"拖累"：持續整合確保個人寫的程式碼不久就會融入大家所的程式中；測試程式確保所有的迴歸測試(regression test[3])動作都會被有效執行。

我發現，一般還算支持XP的人，往往不容易接受程式共有這招的確有用的看法，如果對你而言，這可能是個問題，那就要看一下你的整合動作做得夠不夠頻繁（下一小節會談到）、搭檔編程有沒有落實（下一章會談到）。

整合

一個團隊何時可以拍胸脯保證所有的程式碼都可以一起運作無誤？又如何做到這種保證？

出貨前整合

有些我曾共事過的團隊，通常都讓開發人員在自己的一塊小天地中埋頭工作，必要時再從別人那邊尋求一些協助，在系統出貨之前，他們

會嘗試“凍結程式碼”（但通常，那時程式碼才會因為外在需求不斷改變而開始“溶解”呢）：每個人確定他們的程式碼都已經融入(check in)系統了，任何因為介面更改所引起的程式相衝也都解決了（特別是很明顯會讓編譯動作不過的那些程式衝突）(這有些類似煉金的過程：一開始有很多動作，但隨著溫度降低，一切均歸於一種低能量的狀態)。

這種做法的最大問題，是它讓人們走了徹底錯誤的方向，因為沒有任何事情可以迫使人們做上數日或數週的測試（甚至整合）。最後，整合動作會變成非常花工，因為每個人都必須要下海解決所有問題。

每日建構（Daily Builds）

比上述做法更進一步的是每日建構，每天晚上系統都會整個被編譯一次，並進行“冒煙測試”（smoke test）。

開發人員現在的格言變成了“不要讓建構不過。”在把程式放進系統之前，他們應該針對自己的程式先做一次整合測試。

這樣一來，應該就不會有什麼意外才是（我曾經待過的一個採行此做法的團隊，那位專案經理非常積極的檢查大家有沒有先做自己的整合測試）。

團隊會逐步演化出不同的機制來處理程式不過的問題，我曾經聽過一個團隊，他們說“ 誰讓建構不過，誰就必須在次日一早八點以前進辦公室，把最新的建構檢查一次(一直做到有下一個人讓它不過為止)。”

另一個我曾待過的團隊，靠的是同儕之間的壓力，公佈誰讓建構不過，最後，團隊請了一位組態經理人（configuration manager），他的任務就是在每天一個固定的時間找出任何編譯或整合上的問題，然後通知相關的程式師（們）將其修正[4]。

持續整合

XP 所採行的持續整合不是真的“ 時時刻刻” 都在整合，但一天下來的確會做上幾回，每當一組搭檔完成了一段程式，他們就會整合至大家的程式中。通常會有一台機器專門做整合的工作，先到的程式碼先做。

XP 的整合會由測試替它背書，程式師自己的單元測試必須全過，整合時，他們帶著寫好的程式碼到一台目前測試全過的機器中去跑。如果測試全過，他們的這段程式碼才算完成，如果不過也很簡單，因為距上次不過只有一件事變了：就是系統剛加入了他們的程式碼。這時，他們必須修正程式碼(也許在其它人的協助之下)，或是把這個改變『取消』，亦即丟掉那些不過的程式碼。系統不允許這些程式碼進入，以免『退化』。

持續整合在 XP 中是可行的，因為團隊有一種“ 患難與共” 的文化；因為測試會幫它背書；也因為 XP 利用重整達到了更為簡單的系統設計方式。

加班工作

當時間『燒光』時，你會怎麼做？

加班

對許多團隊而言，加班是他們對時程危機的第一個反應，一開始是工時變長，然後被要求週末也要來，有些更是離譜到要求強制加班的程度。

這種方法可能對生產力適得其反，工作有可能變成一段“ 死亡進行曲” （death march，Yourdon 語，1999[5]），人們發現自己太累而無法正確思考，因而導致生產力的低落，只有『機械性的寫程式』而沒有任何『心智性的思考』，不然就是家庭生活瀕臨毀滅。

四十工時

XP 要推的是一種不同的觀點，它主張每週四十工時較為合宜（或四十上下）。當然，XP 團隊體認到或許偶爾需要比較長的工時，但如果長到連續兩週都要加班，那就表示有其它問題了。

所謂“40”是什麼意思呢？不是說一定剛好 40，對有些人來說可能是 35，另一些人則是 45...等等，40 指的是我們必須有個限制。

Bob Martin 談到了一種概念：“燃燒 8 小時。”意思是說，你可以拼死拼活做 8 個鐘頭，但超過 8 小時則無論如何不再恰當了。

XP 式的團隊希望把生產力一直維持在一定的水準，如果團隊無法做到他們在某個週期所承諾的全部，他們就會把功能縮減一些，而不是進入『加班』的狀態。XP 的團隊使用“昨日天候法則”（Yesterday's Weather Rule）：下一個開發週期能夠完成的工作量預估，跟這個週期的一樣（就如同昨天的天氣足以當作今天天氣的指標，上一個開發週期的速度也足堪這次週期的借鏡）。這種法則有助於團隊認清他自己在生產力上的真正水準。而加班會干擾這種自我發現的過程，如果團隊正進行一個為期兩週的開發週期，而要完成該週期所承諾的事項需要加班方能竟其功，那非常有可能，下一週期也會要加班。

我的朋友 Steve Metsker 說道：“專家應該會願意每週花上（工作以外的）五個鐘頭來提昇自己。”有的 XP 團隊甚至願意每週保留半天當作他們的“遊戲時間”（在那段時間內，團隊成員可以進行諸如學習新程式語言這類的事），這段不受約束的時間相對於主要的工作時間而言

是划算的。在 Kent Beck 跟 Martin Fowler 所著的《*Planning Extreme Programming*》一書中（2000 年），Beck 有一段很棒的故事，談到一個從“我們時間不夠”轉變成“是我們事情太多”思維的團隊。他指出這種轉變非常具有開創性（empowering）：你拿時間是沒輒的，但你可以在工作量的調整上做些努力。

工作場所

團隊在地理位置上應該如何組織？

地理位置有所分隔的狀況（也一併探討遠距工作）

在地理上分隔的團隊位於不同的地區，也意謂著他們面臨溝通上的困難，可能是語言不同（最慘的狀況）或時區不同，即使時區表面上相同，但工作時間也可能不一樣。我曾經待過一個團隊，一部份的人幾乎都是早上 8:00 到 9:00 之間進辦公室，另一部份人的時間則大約是 10:00 到 10:30 左右，午餐跟下班時間也不同，這兩群人共同的時間是早上 10:30 到 11:30、以及下午 1:30 到 4:00。

剛開始，即使表面看起來成本很低（像是“我們可以雇用兩地最好的人才”、“我們打算在另一地開始，因為那裡的人力比較便宜”等

等）, 但在這種地理上區隔很遠的狀況中，溝通問題反而會讓整個專案成本變得更高。

大部分的團隊偶爾還是必須要『實際面對面共事』一陣，所以，除了溝通成本，還有『旅遊』成本（travel cost）, 包含那些被派去『旅遊』人員心中的『怨氣成本』。據我所知，有個團隊的成員遍及三大洲，這些人感覺到，至少每季都要『聚在同一間屋子裡』, 不然專案很可能就此失敗。

遠距工作更是讓這種地理區隔的問題雪上加霜，因為它會產生更多的溝通問題和更多的孤立感。如果工作可以被分割成只需要最小程度的溝通，遠距工作也許會是個受歡迎的方案。但我了解的狀況是，僅僅有少數人曾經在這種狀況下撐過一年多一點[6]。

（一人或兩人的）辦公室

對於開發者而言，辦公室提供了最大程度的隱私性和靜謐性，在《*Peopleware*》（1987 年）這本書中，作者 DeMarco 和 Lister 的報告指出，有自己辦公室的人，在跟程式碼搏鬥時表現得最好，他們把這種現象歸因於程式師想達到行雲流水狀態的這種『需要』。但辦公室很明顯是最昂貴的抉擇，因為看來很少公司會願意為開發人員提供辦公室的（我發現雙人的辦公室要比小隔間來得更有生產力）。

小隔間

在程式師的眼中，這種型式的『辦公室』大概是他們最不喜歡的，但對於把他們丟在那兒的經理人而言，這卻是他們最喜歡的。我曾待過的一個公司似乎還想把這種隔間弄得更小：十年前，這種隔間的標準尺寸還有 100 平方呎（公司總裁的也是這個大小），但今時今日（在另一間更小的公司），卻只變成 54 平方呎了（而且我的塊頭還比那時更大了呢！）。我目前的隔間大小，還不夠讓兩個人『並桌』而坐，就算真的可以，那電腦就只能擺在角落了。

還有，隔間大概只會變得更小，而絕不可能更大。我有一位朋友，他們公司偶爾還會在週末來個“大風吹”，你在週末把東西打包整理好，下星期一進辦公室時，座位還是維持不變，只是空間再縮小 6 吋。因為他們認為每個隔間貢獻 6 吋，像足球場大小的房間就可以再擠出更多小隔間。

隔間無法容許太多的隱私或寧靜，而且對溝通也會多所干擾（我隨便在什麼地方都可以看到 4 到 8 呎不等的高牆，但大都是『上空』，而且一定都沒有門）。

開放空間

XP 表明要一個開放空間，在一個中央區之外，通常還有些小小的私人空間環繞於外。最快的機器放在公共區，為搭檔編程用的。XP 希望團隊裡的程式師，能夠專注在他們要解決的問題上，同時還能聽到隔壁

的對話，以便在幫得上忙時跳進去幫忙。但若沒有了 XP，同樣是開放空間，卻會退化成像是“ 臨時拘留所” （bullpen）一般，每個人都被塞進同一塊區域，完全喪失了私人空間。而且由於你跟其它人不一定在做同一個專案、或處於同一個階段（phase），所以溝通的步調就會彼此衝突，這是我經歷過最糟的辦公室型態。

剛開始採行 XP 的團隊，通常對開放空間的導入也是亦步亦趨，他們的做法是把一些多餘的空間拿來擺放搭檔編程的那些電腦，或把一排的隔間充作公共區。但值得注意的是，有些地方的隔間牆並非是冷冰冰的牆，它們裡面是有電線的，移動這些『牆』可能會有危險。

圖 3.1 顯示了一個擁有四個程式師的 XP 團隊可能用採行的空間模式，注意到有兩台機器做搭檔編程用[7]，一台做整合用。圖下方則是私人辦公區，團隊中最好的機器往往位於共用區，稍遜的就放在私人區，供 email 或上網之用。

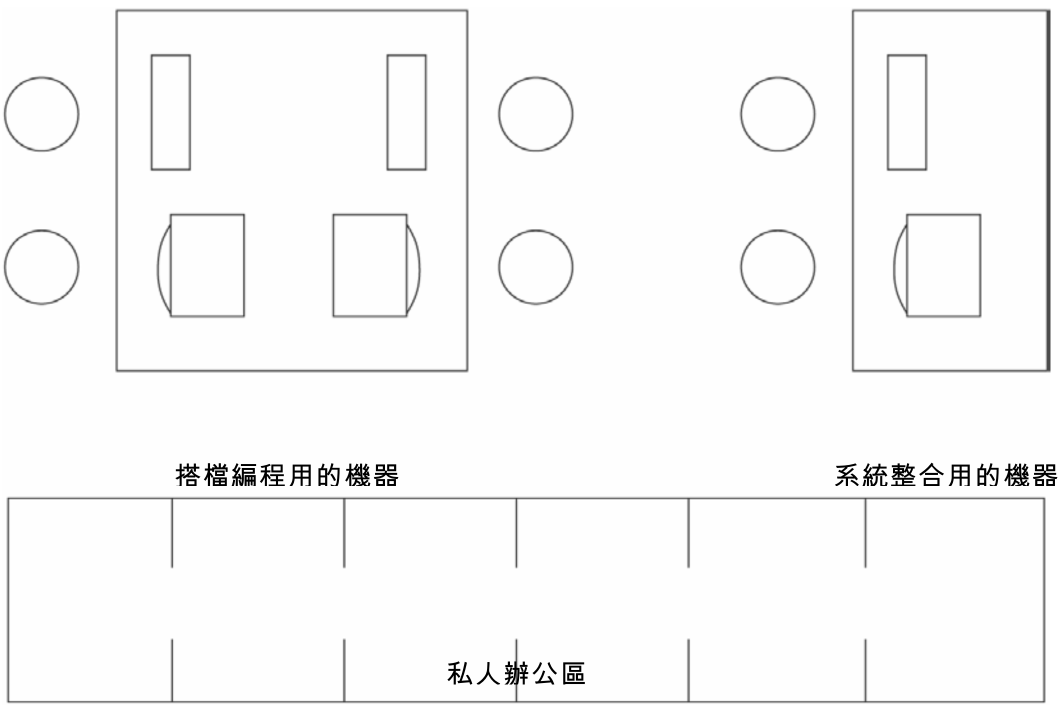

圖 3.1：開放空間的範例

更版時程：頻繁改版

團隊應該多常更新他們的產品？XP 主張頻繁改版，且每次改版幅度儘可能小，這麼做是在學習，學習中可以獲得的回饋（特別是從實際使用經驗中的回饋）愈多愈好。

經年累月都不做改版的專案會累積大量風險：科技會變、企業大環境會變、團隊組成也會變，小幅度的改版可以降低這些風險。

改版幅度是應該小，但也應該改得有意義：夠得上一個功能的改版才應該納入。第一次的版本至少要初具主要元件或子系統的功能。

在系統開發過程中未能達到小幅改版的目標，是我曾犯過的最嚴重錯誤。當我專注在這件事情上時，我發現這個幅度甚至可以比我預期的更小。

所以我決定，每個專案一開始，我都會問這個問題：“ 這樣的介面可以嗎？” (『即使手沒空，用你的鼻子跟前額來按按鈕都沒問題』的介面---參見圖 3.2）

答案通常是“ 可以，如果你知道參數的話。” 如此一來，我們一開始就可以不考慮使用者介面（GUI）的問題。如果這樣的介面還不夠，那也可以讓我們專心去了解系統最起碼的功能應該是什麼（我也發現，抗拒這種『行遠自邇』做法的，可能是開發人員本身，因為他們能看到系統的『遠景』，所以極不願意從那樣的高度走下來）。

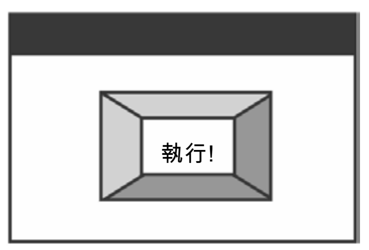

圖 3.2：最簡單的介面

在第一期『XP 集訓營』中，Michael Hill 談到他做出一個『零功能版本』（Zero Feature Release，ZFR；唸做“ ziffer” ）的系統，那是一個徹徹底底什麼也不做的版本，但在這第一版中，架構跟部署策略卻都齊備了。要讓系統一開始就能『上場打仗』並非易事，不斷更新通常比較容易一些。

編程標準

編程標準可以改善人們閱讀彼此程式碼的能力，此外，在 XP 中，編程標準還能支撐起重整：外觀一致的程式碼可以讓人們在重整時更有信心。

對於團隊建立編程標準這件事，有好幾種做法：

- 根本沒有標準，不妙。
- 程式師自己決定，然後他自己廣為使用它(“如此一來他就可以知道自己加進的程式碼在哪一段”)，還是不妙。
- 程式師自己決定自己程式碼的風格，但要修改一段既有程式碼時，該程式碼的風格必須維持原樣，這是我能接受的最低限度。我曾經跟很多團隊共事過，他們都想要有一個編程標準，但最後都演變成這種模式，因為沒有強制性的壓力。採行此種做法，至少你在閱讀個別的程式模組上沒有問題。不過絕對會對重整造成干擾，因為，當你把程式碼在模組間搬來搬去時，必須連同程式碼的格式也一併『重整』。
- 團隊有個大家都遵循的標準，對 XP 而言這是最理想的狀況。團隊級的編程標準能支撐起 XP 的眾多實務，像是程式共有、搭檔編程、程式重整、持續整合等[8]。

我曾看過兩種團隊用以決定其編程標準的方法，第一種的團隊在專案一開始就來一場大辯論，激烈的爭論空格跟括弧的擺法，但最終塵埃落定到一份共同的風格。第二種的團隊根本不『吵架』，但也未能建立

起團隊的標準，他們最後的結局是“ 每個人都有自己的編程標準。”對於這種團隊，我不確定自己能否帶領他們前進。

那我們應該用什麼樣的程式碼風格呢？在我最近參與的一個專案中，他們在定義這個風格的時候（當時用的是 Java 語言），有一份六頁的 Sun 官方版 Java 編程慣例，也包括 JavaBeans 的命名慣例。我當時建立了一頁的編程標準，但說實話，如今我只想試著跟 Sun 的標準一致：四個空格的縮排、開頭的大括弧跟前面的程式碼放在同一行、以及 JavaBeans 的命名標準等。我相信，只要團隊奉行程式共有、搭檔編程、以及最初原則上同意的一份編程標準，他們就終將會有一份合理可行的編程標準。

總結

我們討論了幾個 XP 的團隊實務，以及一些可能的替代方案：

程式所有權：當程式需要需要改變時，誰有權改它？XP 主張“ 程式共有” ，非 XP 團隊或許會認為這種做法有問題。

整合：團隊如何確保、何時確保每個人寫的程式兜在一塊可以運作？XP 主張“ 持續整合” ，非 XP 團隊應該儘可能努力達成持續整合。

加班工作：當時間『燒光』時，你會怎麼做？XP 主張“ 四十工時” ，非 XP 團隊也應該努力達成這個目標。

工作環境：團隊的位置應該怎麼坐？XP 主張"開放空間"，如果是一個非 XP 的團隊，我偏好『辦公室』那種，但仍留意團隊的溝通管道。

更版時程：更版的頻率為何？XP 主張"頻繁改版"，如果是一個非 XP 的團隊，我認為這仍不失為一個有用的目標---儘管會比較難做到。

編程標準：程式應該怎麼寫？XP 主張"必須要有一個團隊級的共同標準"，如果是一個非 XP 的團隊，我認為團隊有個標準仍屬恰當。

資源

- Beck, Kent. 2000. Extreme Programming Explained. Boston: Addison-Wesley.
- Beck, Kent, and Martin Fowler. 2000. Planning Extreme Programming.Boston: Addison-Wesley.
- Cockburn, Alistair. 1998. Surviving Object-Oriented Projects, Reading, MA: Addison-Wesley.
- Discusses various models; recommends "Owner per Deliverable." Coplien, Jim. 2001. "Code Ownership." 網址：http://www1.belllabs.com/user/cope/Patterns/Process/section18.html.
- DeMarco, Tom, and Timothy Lister. 1987. Peopleware: Productive Projects and Teams. New York: Dorset House.

■ Jeffries, Ron. 2001. “Code Ownership.” 網址：
http://www.xprogramming.com/Practices/PracOwnership.html.

■ Sun. 2001. “Sun’s Java Coding Conventions.” 網址：
http://java.sun.com/docs/codeconv/index.html.

■ Sun. 2001. “JavaBeans Conventions.” 網址：
http://java.sun.com/beans/docs/beans.101.pdf.

■ Sun. 2001. “JavaDoc Conventions.” 網址：
http://java.sun.com/products/jdk/javadoc/writingdoccomments/.

■ Yourdon, Edward. 1999. Death March: The Complete Software Developer’s Guide to Surviving “Mission Impossible” Projects. Englewood Cliffs, NJ: Prentice-Hall.

譯註

1. 一種轉換程式，可以在像 IBM 3270 那種『主機--終端機』舊式系統和新式的 Windows 使用者介面之間做輸出入的轉換動作，亦即，把從 Windows GUI 得到的輸入轉換成 IBM 3270 式的輸入，然後將處理結果以 Windows GUI 式的輸出呈現在使用者面

前。這種程式的目的，是要讓舊型系統的資料仍舊可以被繼續使用。

2. 請參見《第五項修練》。

3. 每當系統中新加入一些程式碼時，之前測試過的那些程式碼又會變得『岌岌可危』，因此，必須把以前的測試跟新的測試全部都跑過一遍，才能確保所有的程式都沒問題，這種『溯及既往』的動作就稱為『迴歸測試』。

4. 通常是最後一位把其程式碼放進系統的那位程式師或那組搭檔。

5. 這也是 Yourdon 那本著作本身的書名。

6. 由天下出版社翻譯的《資訊科技的商業價值》，裡面有一篇提到電子通勤在執行面上的困難度，這種上班方式的成本其實比我們想像中來得高。

7. 因為搭檔編程之故，四個程式師只需兩台機器寫程式。

8. 關於 XP 眾多實務之間的彼此支撐狀況，請參閱《*Extreme Programming Explained*》圖四。