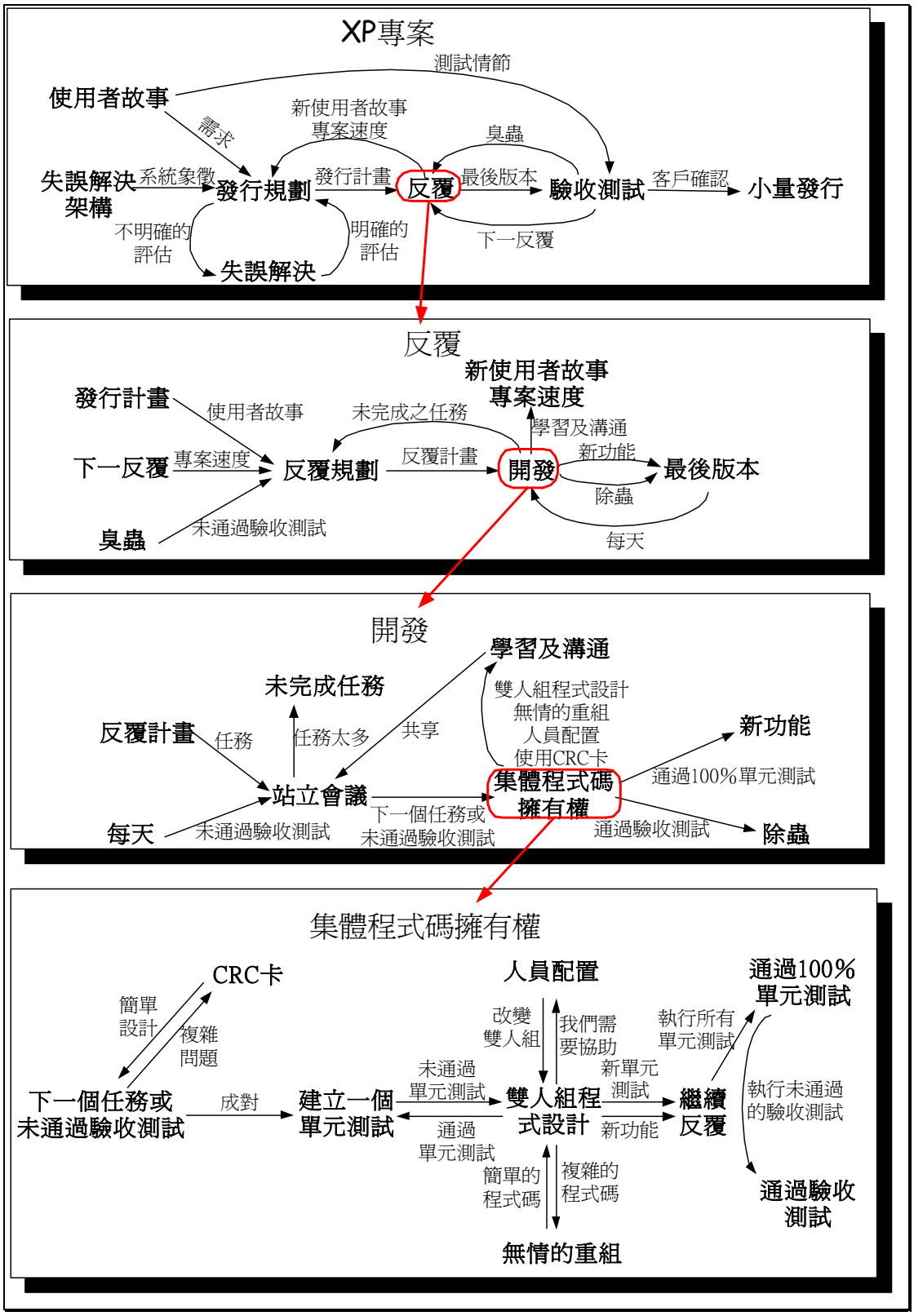
終極製程 Extreme Programming

本文的目標是介紹 Extreme Programming (XP)的概觀。



譯者：Areca Chen

Dot.Space 點空間 Extreme Programming :A gentle introduction.
[原網頁]

[下載整個網頁]

[原文發展的過程]可以看到最新的相關資訊。

首先我們以一個簡單的問題『甚麼是 XP』開始。就如您將看到的；XP 是一個嚴謹且有規律的軟體開發方式(或軟體發展指導原則)。接著我們可能想知道『何時使用 XP』。由於**需求的動態改變**所產生的專案危機最適合使用 XP，這類專案更能從 XP 體驗到更大的成就及生產力。但我們是否還需要『其他的軟體方法論(methodology)』，事實上我們是需要的。XP 是一種全新的方式，XP 的成功來自於強調客戶滿意及促進團隊合作(team work)。這是如何達成的?XP 最令人驚艷的是其簡單的規則及實務作法。它們乍看起來似乎是笨拙的並且可能是幼稚的，但使用後立刻接受其改變。客戶樂於參與軟體的開發過程且開發人員不顧一切的主動貢獻其經驗。**這些規則及實務作法必須互相支援**，XP 圖顯示他們共同工作以組成一個開發的方法論。沒有生產力的活動得以改善並降低成本及失敗。我想嘗試 XP 但我如何開始？只要在你現有的方法論中額外增加一點點並且立即嘗試，馬上在你的專案中得到許多利益。你在哪裡可以獲得更多的資訊？這裡有參考資料、書籍、網頁當然還有許多人。有許多人及專案正在尋找參與嘗試 XP。到底有哪些其他的專案已經學習相關的 XP？這裡有些重要的課程需要學習。

- XP2001 XP 討論會在義大利, May 21-23.
- XP Universe July 23-25 in Raleigh, North Carolina.

甚麼是 XP？(What is XP)

XP 事實上是一種嚴謹且有規律的軟體開發方式，其發展已有 5 年的歷史，它已在一些對成本重視的公司中被實證，如 Bayerische Landesbank, Credit Swiss Life, DaimlerChrysler, First Union National Bank, Ford Motor Company 及 UBS。XP 的成功是由於其強調的客戶滿意度，其方法論被設計在需要時用來實現你的客戶所需要的軟體。XP 授權您的開發人員自信的回應客戶需求的改變；既使是在軟體生命週期的後期。

這個方法論同時強調團隊合作。管理人員、客戶及開發人員都是貢獻於開發高品質軟體團隊中的成員。 XP 的實作非常簡單；甚至在團隊開發都能有效的運用。 XP 改善軟體專案的四個基本方式：溝通、簡化、回饋及勇氣。 XP 的程式設計師與其客戶及程式設計師同僚溝通。他們保持其設計簡單且清晰。他們一開始便不斷從測試軟體獲的回饋，他們儘可能提早交付系統給客戶並且立即依據其建議改善軟體。基於這個基礎 XP 的程式設計師能夠勇於回應持續改變中的需求及技術。

譯者：Areca Chen

XP 是不同的，他就像一堆拼圖，有許多小的單元。個別的單元是沒有意義的，但組合起來便是一幅完整的圖片。這是與傳統軟體開發方法重大的不同並且引領我們改變程式設計的方式。

何時使用 XP(When should XP be Used)

XP 被建立在回應需求持續改變的問體領域，你的客戶可能對於系統應該是甚麼樣子沒有確定的概念，你的系統功能可能隨時會改變。在許多軟體環境動態改變的需求是唯一的常態。這是 XP 可以成功而其他方法沒辦法的地方。XP 同時也設定處理專案風險的問題，如果您的客戶需要新的系統而其資料非常特殊此時會有高風險產生，如果這個系統對你的團隊是一個新的挑戰則風險更高。如果這個系統對於整個軟體工業是一個新的挑戰則風險是高於以往任何情況。XP 的作法可以降低風險並提升成功的可能性。

XP 所建立的是小群組的程式設計團隊；約 2 至 10 人。你的程式設計師可以是普通的程度；使用 XP 你的程式設計師無須是博士。但**你無法將 XP 使用在大型團隊的專案上**。我們應該注意到的是動態需求的專案或高風險的專案你可以發現小團隊的 XP 程式設計師是比大型團隊更有成效。

XP 需要一個延伸的(extended)開發團隊，**XP 團隊包括不只是開發人員；還包括管理人員及客戶**，他們手攜手共同工作。詢問問題、協商界定範圍及進度時程，及建立功能測試需求，而非只有開發人員投入生產軟體。

XP 另一個要求的特性是可測試性(testability)。你必須能夠建立自動化的單元測試及功能測試。但是某些領域無法符合這個要求，你可能懷疑有多少是不符合的。你需要在某些領域中應用一些巧妙的測試技巧，你可能需要改變你的系統設計以方便測試。記住；無論如何一定要建立測試的方法。

最後一件事是生產力。XP 專案相較於在相同環境中的其他專案開發方式無疑是有較高的生產力。但這不僅僅是 XP 方法論的目標，**真正的目標是交付的軟體是客戶真正需要的軟體**。如果這是你專案中最重要的那麼正是你採用 XP 的時候了。

其他的軟體方法論(Do We Need Yet Another Methodology)

當哈柏望遠鏡(Hubble Telescope)開始展望太空；它有一個鏡片設計錯誤，並不是直到望遠鏡完成部署交付客戶時這個問題才發現，一個衛星的客戶不能直到衛星發射到太空才測試它，但爲甚麼軟體是要這樣？如果你走入一個電子實驗室並看到工作台上的是舊的設備你會如何？你會保留那個巨大陳舊的真空管電視只因爲它有好的供電器？你會保留一個舊的調幅收音機只因爲它可用於無線電產生器？你不會的；因爲你的工作台實在是太凌亂而無法做事情。但這正是我們如何對待我們的軟體。

跨越 Saginaw River 的 Zilwaukee 大橋急需要建造，這座橋樑從兩岸同時建造並在中點會合，但兩邊在中間會合時高度相差 3 呎。當你的專案也是從兩端開始建構；是否直到兩者的終點否則你無法整合你的專案；但爲甚麼我們要如此對待我們的軟體？

XP 被設計來回應這類的問題。XP 是基於觀察哪些使得電腦程式快速或變慢的原因。XP 是新的重要方法論；原因有二：首先且是最重要的；它是一個軟體開發實務的重新調查(re-examination)並逐漸成爲標準操作程序。其次它是許多新出現的輕量級軟體方法論(lightweight software methodologies)其中之一；其目的是建立來降低軟體成本。XP 提前一步定義簡單並愉快的程序。

簡單的規則及實務作法(The Rules and Practices of XP)

- 規劃階段
  - 撰寫使用者的故事(User stories are written.)
  - 進行發行規劃以建立進度時程(Release planning creates the schedule.)
  - 提高小版本的發行頻率（小階段發行）(Make frequent small releases)
  - 評估專案的速度(The Project Velocity is measured)

- 將專案開發工作切割成許多的反覆(The project is divided into iterations)
- 以反覆規劃啓動每一個反覆開發(Iteration planning starts each iteration)
- 配置人員(Move people around)
- 每一天的站立會議(A stand-up meeting starts each day)
- 當 XP 無法遵守確定它(Fix XP when it breaks)

- 設計階段
  - 簡化(Simplicity)
  - 選擇一個系統象徵(Choose a system metaphor)
  - 設計會議時使用 CRC 卡(Use CRC cards for design sessions)
  - 建立失誤解決方案以降低風險(Create spike solutions to reduce risk)
  - 功能不要太早加入(No functionality is added early)
  - 隨時隨地盡可能重整(Refactor whenever and wherever possible)
- **程式碼編輯：**
  - 客戶是隨時可得的(The customer is always available)
  - 程式碼必須有一致的撰寫標準(Code must be written to agreed standards)
  - 首先撰寫單元測試程式(Code the unit test first)
  - 所有程式碼產品都是雙人組設計(All production code is pair programmed)
  - 一次只有一對雙人組整合程式碼(Only one pair integrates code at a time)
  - 隨時整合(Integrate often)
  - 使用集體程式碼擁有權(Use collective code ownership)
  - 直到最後才最佳化 (Leave optimization till last)

- 決不加班(No overtime)

- **測試階段：**
  - 所有的程式碼必須有單元測試(All code must have unit tests)
  - 所有程式碼在發行前必須通過單元測試(All code must pass all unit tests before it can be released)
  - 發現臭蟲立即建立測試(When a bug is found tests are created)
  - 隨時實施驗收測試並公佈成績(Acceptance tests are run often and the score is published)

如何開始(How do I start this XP thing)

開始使用 XP 最佳的方式是應用於新的專案。從蒐集使用者的故事及針對可能是風險的事務實施失誤解決方案開始，只要幾週的時間就可以完成這件事。接著發表規劃會議的進度時程。邀請客戶、開發人員及管理人員建立一個大家都認同的進度時程。以反覆規劃會議開始你的反覆規劃。那麼現在你已開始了。

一般專案只有在發生問題時才會想要尋找新的方法論；如 XP，此時最佳開始使用 XP 的方式是好好檢視您現有的軟體方法論並且指出哪些讓你慢下來的原因。先用 XP 解決這些問題。

例如；如果你發現你的開發程序方法中 25%你的需求說明是完全無用時；那麼召集你的客戶及相關人員開始撰寫使用者的故事。

如果你有一個常常改變需求而導致你頻繁的修改進度表的長期問題，那麼在每一個小的反覆中嘗試一個比較簡單且容易的發行規劃會議(但你先要有使用者的故事)。你的程式開發任務可以嘗試一個反覆風格的開發(iterative style of development)與及時風格的規劃(just in time style of planning)。

如果你最大的問題是產品中有許多臭蟲，那麼嘗試自動驗收測試。 使用這種測試適合可逆性(regression)（譯註：測試新功能時也要做之前已經做過的測試）及確認(validation)測試。

如果你最大的問題是整合性的臭蟲(integration bugs)那麼嘗試自動單元測試。在任何新的程式碼被存入程式碼庫(code repository)之前這個測試需要 100%通過。

如果有一兩個開發人員擁有的系統核心類別並且需要改變而遇到瓶頸，那麼嘗試集體程式碼擁有權(你同時需要單元測試)。讓每一個人在需要時可以改變核心類別。

你可以持續這種方式直到沒有任何問題存在。接著人員盡可能加入其他的作法。一開始你加入的作法看起來很容易，表示你已花費少量額外的精力即已解決一個大問題。第二個加入的作法也是看起來很容易，但是介於使用少數的 XP 規則與使用所有的 XP 規則間的某些點需要堅持以讓 XP 成爲有用的技術。你的問題將會被解決同時你的專案得以控制。此時放棄新的方法論並回到你已熟悉並舒服的方式看起來可能是不錯的主意，但是持續下去最終將會獲得回報。你的開發團隊將會變的比你預期的更有效率。在某些點你會發現 XP 規則不再像是規則了。規則中有一些協同作用並不是那麼容易了解除非你已完全浸淫於其中。

在使用雙人組的設計(pair programming)這種攀登高峰的感覺尤其真實，但這種技術的回報是非常大的。同時單元測試需要花時間去收集，但單元測試是 XP 中其他許多作法的基礎所以其回報也是非常大的。

XP 專案並不是沉靜的；總是有人在談論者問題及如何解決這個問題。人們來來往往向別人詢問問題與夥伴交換程式設計的心得。人們自然的會談以解決問題同時散播。要促成這種反覆的會議，提供一個會議空間並設定工作空間以讓兩人可以容易的共同合作。整個工作空間可以是開放空間以促成團隊溝通。

學習相關的 XP(What We Have Learned About XP)

- 發行規劃(Release Planning)
  - 團隊擁有進度時程(The Team owns the schedule)
- 簡化(Simplicity)
  - 簡化就是容易維護(Simplicity is easier to maintain)
  - 你逐漸不需要它(You aren't going to need it)
- 系統象徵(System Metaphor)
  - 系統象徵可以簡化設計(A metaphor can simplify the design)
- 雙人組設計(Pair Programming)
  - 整體大於部分(he whole is greater than the parts)
  - 一些經驗法則(some rules of thumb)
  - 控制莽撞的程式碼(Rein in the Cowboy Coders)
  - 雙人組可以減少猶豫不決(Pairing reduces indecision)
  - 不犯錯，雙人組是辛苦的工作(Make no mistake, pairing is hard work)
  - 雙人組的實驗性證據(Experimental evidence for pairing)
  - 程式碼檢查可能是有害的(Code reviews considered hurtful)

- **隨時整合(Integrate Often)**
  - XP 與資料庫(XP and Databases.)
  - 整合的間隔可以縮小到每秒(Integration can be reduced to seconds.)
- **最後才最佳化(Optimize Last)**
  - 並不如你所想像的那麼遲(It may not be as slow as you think)
- **單元測試(Unit Tests)**
  - 值得投資(Well worth the investment)
  - 可以節省我們的時間(Could have saved us some time)
  - 一開始便撰寫測試程式使得程式碼變的可測試(Testing first makes the code testable)
- **驗收測試(Acceptance Test)**
  - 它們讓我們有穩定感(They give a feeling of stability)
  - 建構一個工具維護它們(Create a tool to maintain them)

如果你曾使用 XP 或有元件實務經驗(component practice)告訴我們你學到甚麼！如果你有一個故事關於節省你的時間請寄給我們。如果是有關難以實踐的也寄給我們。請撰寫關於 XP 方法的學習課程並傳到點空間

我們改變程式設計的方式(A Change in the Way We Program)

軟體是一種工程；簡單而且精緻的軟體其價值不如複雜並且難以維護的軟體。這是真的嗎？事實上 XP 的概念並不認爲這是真的。

一個普通的專案較硬體耗費約 20 倍的人力，這表示一個專案如果每年花費 200 萬元在程式設計師而花在電腦設備每年只要 10 萬元。如果讓我們說我們是精明的程式設計師而且我們設法使用巧妙的程式技巧節省 20%硬體成本。這會讓程式碼難以了解及維護，但是我們每年節省 20%或 2 萬元，節省成效卓著。相對的如果我們撰寫程式時讓程式碼容易了解並具延展性，我們預期可以節省至少 10%人力成本，也就是 20 萬元不是節省的更多。這的確是你的顧客非常重視的。

另一個對於客戶重要的議題是臭蟲。XP 強調的不只是測試而且是好的測試。測試是自動的(automated)並且提供程式設計師及客戶一個安全網。測試是在撰寫程式碼之前就建立，在程式撰寫中及程式撰寫完成都是如此。只要發現臭

蟲新的測試便加入。一個安全網緊緊的被建立，臭蟲無法穿透兩次，這也是客戶非成重視的。

另一個客戶會重視的是程式設計對於需求改變的態度。XP 讓我們能夠擁抱改變。相同的一個客戶往往希望有機會讓系統在發行之前看到它是可用的。XP 在早期便取得客戶的回饋，因爲此時還有時間去改變功能或者改善使用者的接受度。你的客戶一定會注意到這些。

大部分想要採用 XP 的人是在對軟體建構的重新評估(re-evaluation)後，程式碼品質的重要性是無可替代的，就因爲我們的客戶無法看到我們的程式碼；不代表我們不會因我們投入諸多心血而感到驕傲。

Jeanine De Auzman
Ford Motor Company

輕量級軟體方法論(What is a Lightweight Methodology)

軟體方法論是一組規則及實務作法用於建構電腦程式。重量級(heavyweight)方法論有許多規則、實務作法及文件。它需要訓練及時間以便可以正確的遵循。輕量級(lightweight)方法論只有少數的規則及實務作法這兩者都很容易遵循。

在 1960 年代末期到 1970 代初期對於程式設計師設計軟體時一般所使用的方法是只要他們認爲可以的方式都可以。許多程式設計師非常傑出他們所設計的程式非常複雜以致一般人很難了解。那時候程式沒有臭蟲簡直是奇蹟。讓電腦可用是值得思考的問題而不是使它像到西部探險一樣。

在 1968 年 Edsger Dijkstra 寫信給 CACM 內容指出 *GOTO* 指令是有害的，軟體工程的中心思想因之誕生。此時我們相信較大型且精練的方法論可以幫助我們建構軟體時保持一定的品質及可預期的成本。難以駕馭的牛仔程式碼得以控制。

在 1980 年代程式設計師有一段美好的時光，我們建構軟體只有少許的規則及實務作法，早期我們的軟體對於品質沒有優勢。好像如果只要我們建立足夠的規則涵蓋這些問題那麼我們便能建構完善的軟體並且能夠準時交付出。於是我們便加入更多的規則及實務作法以涵蓋所有潛在的問題。

在 21 世紀的現在我們發現這些規則難以依循，因爲複雜的程序、未完整了解及以某些抽象的標記撰寫的大量文件都失去了控制。嘗試提出一個較大型及較好的方法論就像加州的淘金熱，來到西部的每一個人所得的只有失望。

我們建構軟體以幫助我們建構軟體。但很快的失去了控制並且大無畏的 CASE 工具產生了。這些工具原本是用來幫助我們遵循規則；本身卻難以使用。程式設計師發現需要抄近路並跳過重要的作法以便維持進度。沒有人真正遵循這些讓我們綁手綁腳沉重的方法論。此時牛仔又回來了。

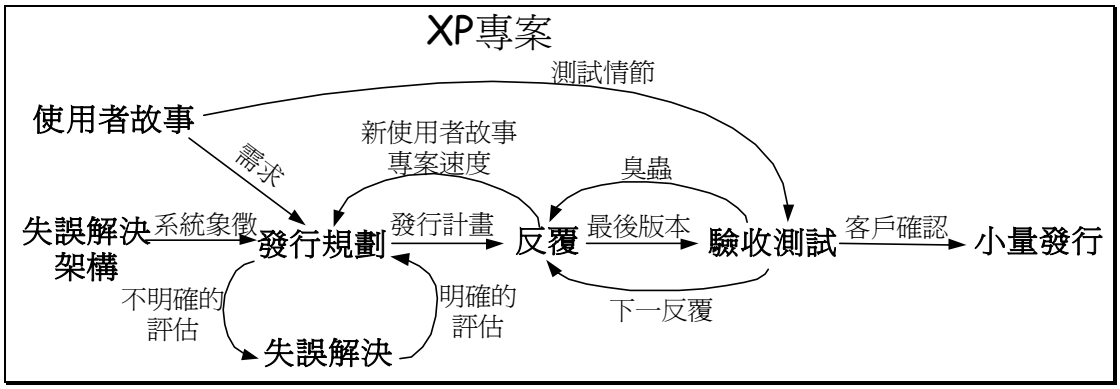
當程式設計師忽略他們方法論的規則，他們直覺的躲開了重量級方法論並回到早期輕量方法論簡單的時光，只需要少數的規則就行了。

但別忘了我們前面所學的，我們可以保留幫助我們建構高品質軟體的規則並且拋棄那些妨礙進度的規則。我們可以簡化太過複雜而難以正確遵循的規則。

我們並不想回到早期沒有規則牛仔式的程式設計。相反的讓我們在使用剛剛好的規則以保持我們軟體的可信賴度及合理的價格。取代牛仔型的程式設計師我們有軟體司法官(sheriffs)參與在團隊中；快速的劃定界線；提供少量的規則及實務作法，這些都是輕量級、簡明並且有效的。

XP 是許多輕量級方法論其中之一。XP 只有少數的規則及適當數量的實務作法，都是容易遵守的。XP 是一個明確且簡明的環境，這個環境的開發是經由觀察那些何者讓軟體開發快速及何者讓軟體開發緩慢的因素。它是一個環境，其中程式設計師的感覺是自由的建構及產出但仍維持組織及焦點。

使用者的故事(User Stories)



使用者故事的目的與使用者案例(use cases)類似但不完全相同。它是用來建立發行規劃會議的時間估算。它也用來取代大量的需求文件。使用者故事是由客戶撰寫有關系統需要執行的工作。它類似習慣用法的情節(usage scenarios)；

此外它不是侷限於描述使用者介面。它的格式是三段式(three sentences)的文字，由客戶撰寫；以客戶的術語而非技術語法(techno-syntax)

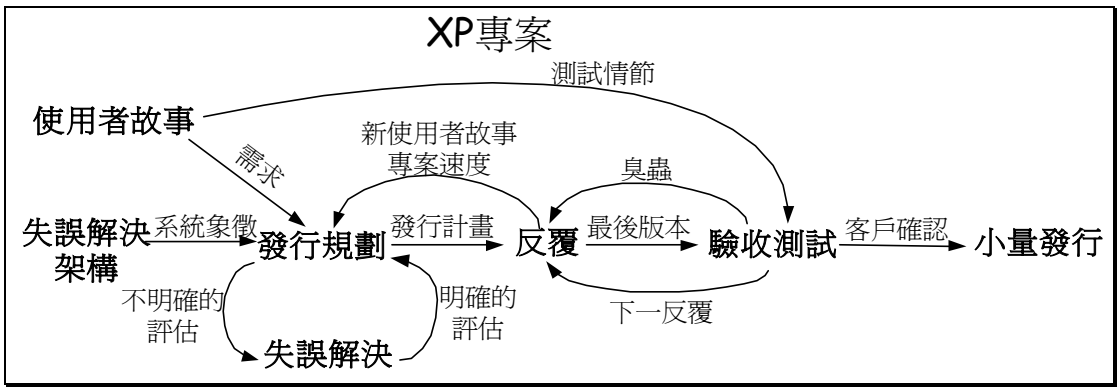
使用者故事同時用於推動建立驗收測試。一或多個自動驗收測試必須建立以便確認使用者故事已被正確的實作。

使用者故事最大的誤解之一是它與傳統的需求說明的差異有多大。最大的不同是明確的程度。使用者故事只需提供足夠的細節以降低有關實作這個故事所要耗用時間合理的風險。當開始實作這個故事；開發人員將拜訪客戶並且面對面的接受詳細的需求描述。

開發人員實作這個故事需要花費的時間，每一個故事以 1、2 或 3 週來預測『理想的開發時間(ideal development time)』。理想開發時間要多長是依據實作這個故事成爲程式碼；在沒有其他事務干擾；沒有其他額外的分配工作；及你已完整了解要做些甚麼。超過 3 週表示你必須將這個故事再進一步分解。不足一週表示你已處於太細節的部分，可以結合其它的故事。在發行規劃時約 80 加或減 20 個使用者故事是建立發行規劃的適當數量。

另一個使用者故事與需求文件間的差異是聚焦於使用者需要。你應該盡量避免描述技術上的細節、資料庫配置及演算法。你應該嘗試讓使用者故事聚焦於使用者的需要及利益而不是說明使用者頁面的配置。

發行規劃(release planning)



發行規劃會議是用於建立發行計畫(release plan)， 發行計畫配置整個專案。因此發行計畫用於每一個單獨的反覆開發建立反覆計畫。

由技術人員執行技術決策而由業務人員執行業務決策是很重要的。發行規劃有一組規則允許每一個人參與專案並執行個別的決策。這些規則定義一個方法以便協調每一個人都可以投入的進度時程。

發行規劃會議的本質是讓開發團隊預估每一個使用者故事理想的程式設計週數。你可預測的理想週數是實作這個故事時你完全沒有其他的事要作，沒有其他相關的事也沒有額外的事，但這些額外的事並不包括測試。客戶接著決定哪些故事是最重要的或哪些是有高優先權要完成的。

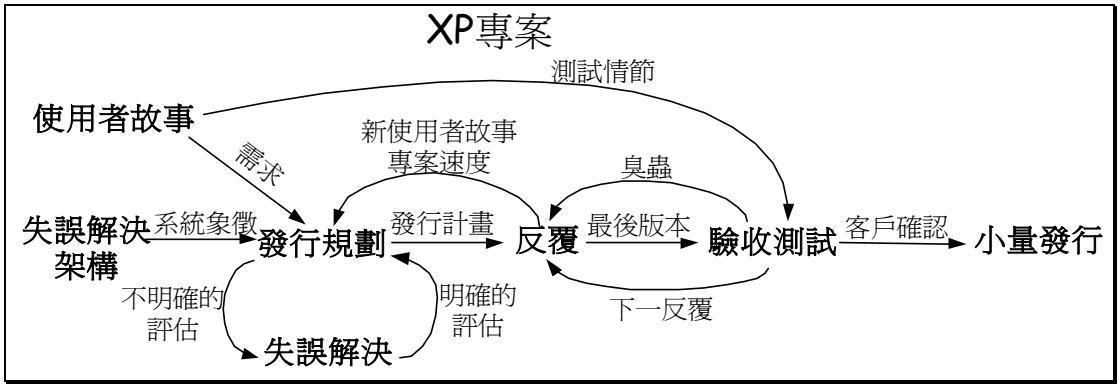
使用者故事列印或寫在卡片上。開發人員及客戶在一張大桌子上共同移動這些卡片以建立一組在第一次發行或下一次發行要實作的故事。一個可用並可測試的及了解商業意義的系統需要及早實現。

你可能以時間或以範圍來規劃，專案速度是用以決定在給定的日期(時間)內有多少個故事可以實作，或一組故事要花多少時間來完成(範圍)。當以時間來規劃；把反覆的數量乘於專案速度決定有多少個使用者故事可以完成。當以範圍規劃；把預期的使用者故事的總週數除於專案速度決定在發行完成需要多少反覆。個別的反覆在每一個反覆開始前才規劃；而且不能在未準備開始前不要預先規劃。發行規劃會議被稱爲規劃遊戲(planning game)而且在波特蘭樣式寶庫(Portland Pattern Repository)中可以找到規則。

當最後的發行計畫建立後而可能想要改變對於使用者故事評估的行爲可能不受管理機制所歡迎。此時你不該這麼做，評估是有根據的而且在反覆規劃會議中將會是需要的。 現在錯估在未來會造成問題。與其協商一個可接受的發行計畫，不如等到開發人員、客戶及管理人員共同協商而獲得發行計畫的共同認同。

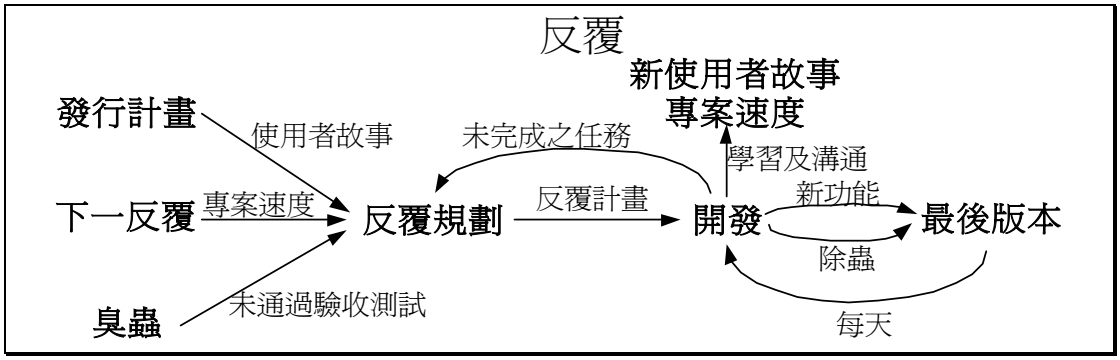
發行計畫的基本哲學是一個專案可以以四個變數量化：範圍、資源、時間及品質。『範圍』是有多少要去完成的。『資源』是有多少人員可用。『時間』是專案或發行將於何時完成。『品質』是這個軟體將有多好或其被測試的有多好。管理人員只能選擇這 4 個變數中的 3 個來支配，而開發人員則支配剩下的一個變數。要注意的是降低品質會有對其他三者無法預期的衝擊。基本上你只有三個變數可以變動；以便讓開發人員緩衝客戶的要求以便讓專案立即降低一次使用的人力。

提高小版本的發行頻率(Make frequent small releases)



開發團隊需要隨時發行系統的反覆版本給客戶。發行規劃會議是用於找尋功能的小單元；這些小單元的功能是具有良好的商業意識；而且可以在專案早期發行到客戶的環境中。這是立即獲得有價值的回饋的關鍵；並且衝擊系統的開發。你愈慢介紹系統重要的外貌特色給使用者那麼你就只有更少的時間來修改它。

專案的速度(project velocity)



專案速度(或只是速度)是衡量你專案中的工作可以多快速的完成。負載因子(load factor)是用來衡量的一個變數。專案速度似乎是比較容易衡量使用。如果那是有益的，使用負載因子作爲評估專案速度的初始預測，但此時量測速度及使用它便互換。

量測專案速度你只要計算有多少使用者故事，或者在反覆中有多少程式設計任務要完成。總計預期已收到的這些故事或任務，就這麼簡單。

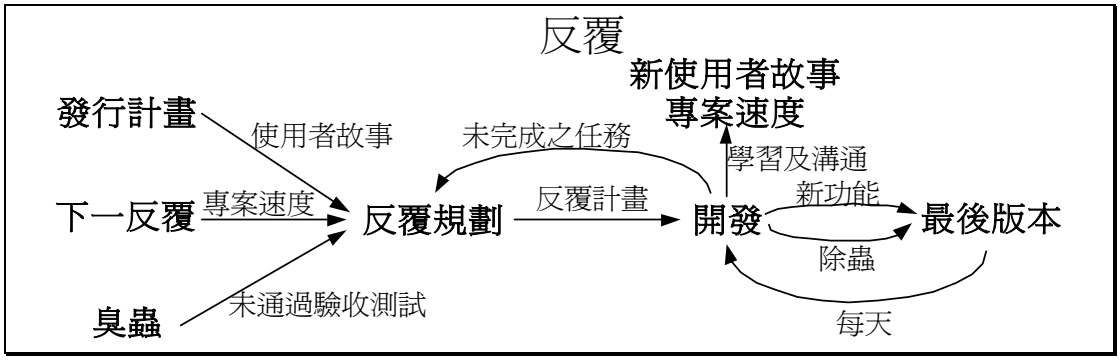
在發行規劃會議中；已完成的故事的專案速度可以用以評估有多少故事將被完成。在反覆規劃會議中開發人員允許簽註執行相同程式設計任務的評估天數；此天數就是之前反覆評估的專案速度。

這個簡單的機制允許開發人員在一個技術困難的反覆之後恢復(recover)或清掉。你的專案因允許開發人員當其工作提早完成並且尚未清掉之前詢問客戶其他的故事而得以提升速度。

別擔心把專案區速度以反覆的長度或開發人員的數量切割，這個數量並不是用於比較兩個專案的生產力的好方法；因爲每一個專案團隊各有其不同的偏好來評估故事及任務，有些評估較高有些較低。這在長期是沒有關係的。追蹤每一個反覆中完成的任務總數量是保持專案在一個平穩(on an even keel)的關鍵。

專案速度的些微增減是可能的。但如果你的專案速度差距達一個反覆時你應該使用一個發行會議重新評估及重新協商發行計畫。當這個系統已進入產品階段期望此專案速度再次改變則是由於維護任務的需要。

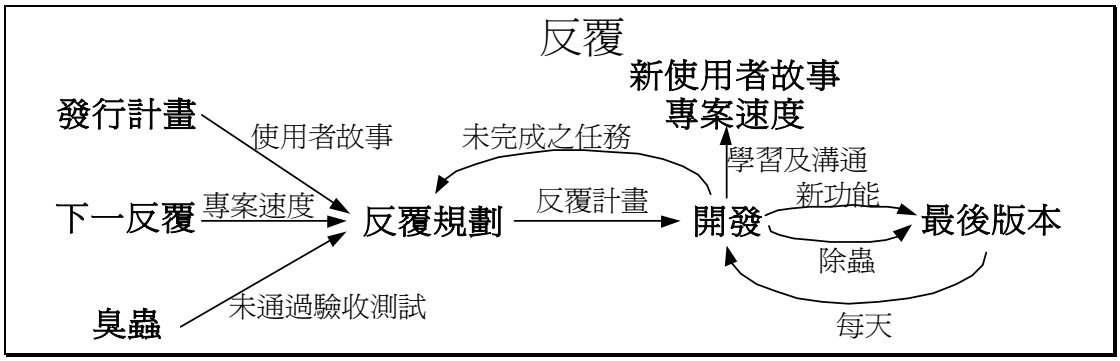
反覆的開發(Iterative Development)



反覆開發使得開發成程序增加許多靈活度(agility)。切割你的開發進度成約一打的反覆；每一反覆 1 到 3 週的長度。

不要讓你的程式設計任務快過進度，相反的讓一個反覆規劃會議在每一個反覆之前召開以規劃要作甚麼事。考慮到將來(look ahead)並嘗試實作任何不是進度時程安排在這個反覆中的事也是違反規則的。當它是發行計畫中最重要的事時就會有充足的時間實作這些功能。

反覆規劃(iteration planning)



在每一個反覆之前召開一個反覆規劃會議以訂定這個反覆的程式設計任務計畫。每一個反覆是 1 到 3 週的長度。使用者故事由客戶自發行計畫中依據何者是客戶認爲是優先的順序選出。無法通過驗收測試需要修改的也被選出。

使用者故事及未通過測試的程式被轉換成程式設計任務。任務係以卡片寫下，這些卡片將成爲反覆的明細規劃。

每一個任務約有 1 到 3 個理想程式設計天數。理想的程式設計天數是在沒有其他雜務下你要完成這個任務的天數。任務的天數低於 1 天者可以聚集在一起，超過 3 天者需要進一步分解。

開發人員簽署這個任務並且評估他們的任務要花多少時間完成。開發人員接受任務並且是評估要花多少時間完成的人是非常重要的。

專案速度是用以決定反覆是否超過預定時程。總計任務的理想程式設計天數，這個時間不能超過從先前反覆的專案速度，如果這個反覆的使用者故事多過於客戶必須選擇的使用者故事就要延遲到下一個反覆(snow plowing)。

如果這個反覆的使用者故事太少；則加入其他的故事是可以接受的。在任務天數(反覆規劃)中的速度可以覆蓋故事週數(發行規劃)中的速度；因爲天數是比較精準。

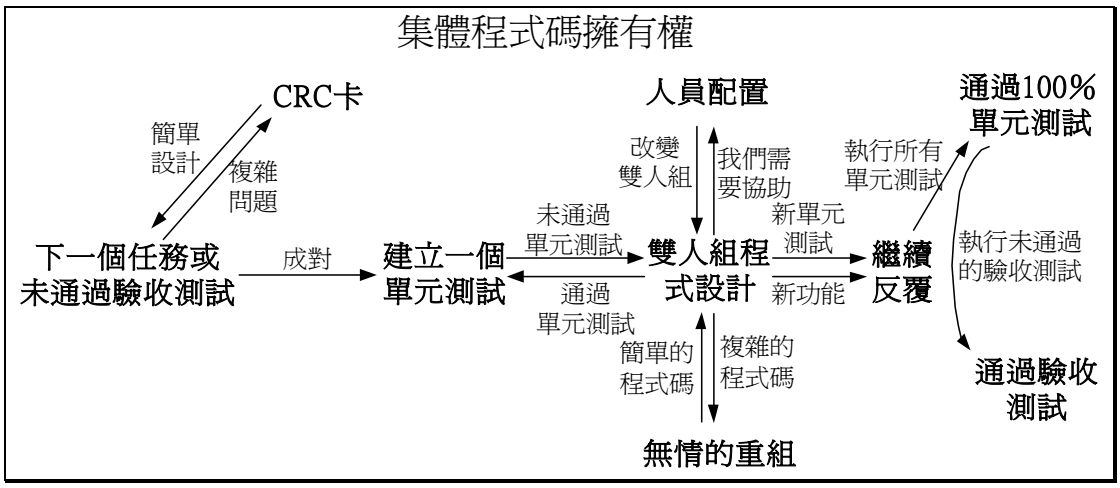
如果看到使用者故事被延遲(snow plowed)往往是令人擔憂的，別驚慌，記住單元測試及重整(refactoring)的重要性。在這方面的問題會拖慢你的速度。避免不按進度增加任何功能，只增加你今天所需要的，增加額外的任何事只會拖慢你的速度。

不要冒險更改你的預計的任務及故事。規劃程序完全依賴一致評估的真實性；捏造它會產生更多問題。

隨時注視著你的專案速度及延遲。你可能需要每 3 到 5 個反覆重新評估所有的故事及重新協商發行規劃；這是正常的。只要你常常優先實作最有價值的故事；你將永遠是盡可能爲你的客戶及管理人員來做。

一個反覆開發風格可以使你的開發程序更爲靈活。試著剛好即時的規劃；即不要超前目前反覆規劃特殊程式設計任務。

配置人員(Move people around)



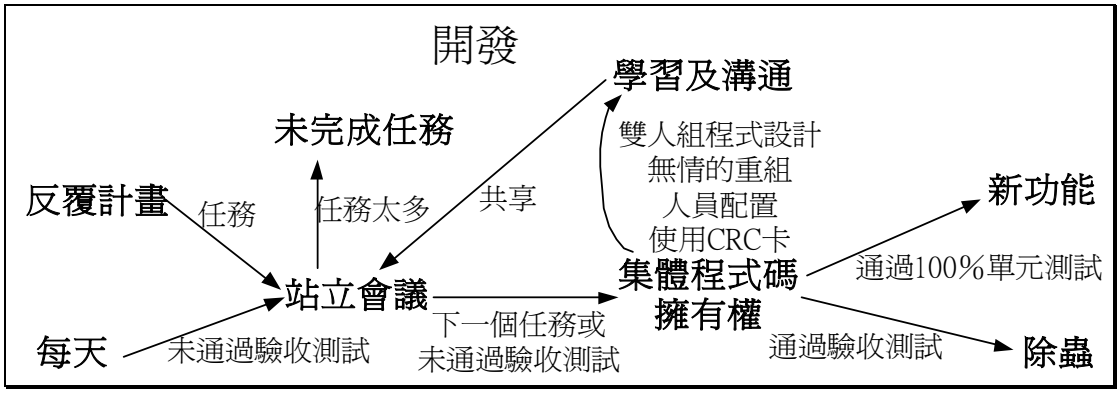
配置人員以避免重要的知識漏失及程式設計的瓶頸。如果在你的團隊中只有一個人可以在給定的領域中工作而這個人離職或者你有數個這部分的工作等待處理；你會發現你的專案的進度變的很緩慢。

交叉訓練(Cross training)往往是公司避免成爲知識的孤島重要的考量，知識有時是非常容易流失的。以程式爲基礎配置人員結合雙人組程式設計(pair programming)可以爲你達成交叉訓練的目的。不要讓單獨一個人了解所有部份的程式碼；應該是讓團隊中的所有人知道每一部分的大多數程式碼。

如果每一個人充分了解他們工作的系統相關的每一部分那麼這個團對是比較有彈性的團隊。若只有少數人過分負荷而其他的團隊的成員則沒甚麼事幹，團隊整體不可能有生產力。任何數量的開發人員可以被指定系統的最熱門部分，這類型的彈性的負荷平衡就是管理人員的夢想成真。

只要鼓勵每一個人嘗試在一個系統新的部分工作；至少在每一個反覆的某一部分。雙人組程式設計可以實現這個想法而不會流失生產力並確保思考的持續性。雙人組中的一個人可以被替換另一個人如果需要可以繼續與新的同伴進行工作。

每一天站立會議(Daily Stand Up Meeting)



在傳統專案會議中多數的與會人員並沒有貢獻，而只是聽聽結論而已。一大堆的開發人員的時間只是浪費在得到瑣碎的溝通。讓許多人參與每一個會議會從專案耗用資源同時變成進度的夢魘。

整體團隊的溝通是站立會議(stand up meeting)的目的。每天早上的站立會議是用以溝通問題、解決方式及促進團隊焦點。每一個人圍成圓圈站立以避免長時間的討論。縮短會議時間是比較有效率的；其中每一個人必須參與；而不只是少數的開發人員。

當你有了每日站立會議；其他的任何會議的參與人員可以依據其實際上的需要並有所貢獻者才參與。如此避免安排多數的會議變爲可能。只有有限的參與人員多數的會議可以在電腦前面自然的發生；此時程式碼可以瀏覽而其概念可以實驗。

每日站立會議不像其他的會議浪費人們的時間，可以取代許多其他會議讓整個網絡節省許多時間。

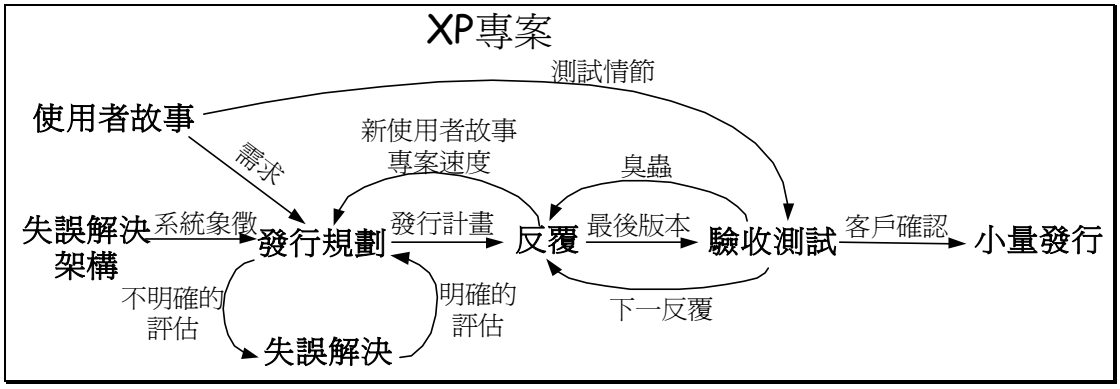
當 XP 無法遵守確定它(Fix XP When It Breaks)

當 XP 的規則無法遵守一定要確定其程序。我們不說『如果』因爲我們已經知道你將需要爲你的特殊的專案作某些改變。依據 XP 的規則開始，但不要猶豫改變行不通的部分。這並不代表團隊可以隨心所欲。規則必須遵守除非團隊已改變規則。你所有的開發人員必須明確的知道每一個人對於其他人的期望，讓一組規則設定是達成這些期望的唯一方式。舉行會議討論有關哪些可以執行哪些則否；並設計實際應用 XP 的方式。

簡化是關鍵(Simplicity is the Key)

簡單的設計總是較複雜的耗費較少的時間完成。所以總是做簡單且比較可能執行的事。如果你發現某部分是很複雜請以較簡單的方式取代。在你浪費許多時間在處理複雜的事情前；以簡單的程式碼取代複雜的程式碼總是比較快速而且便宜的。盡可能保持事情的簡單化就是決不在排入進度前增加功能。盡管如此；保持事情的簡化是一件困難的工作。

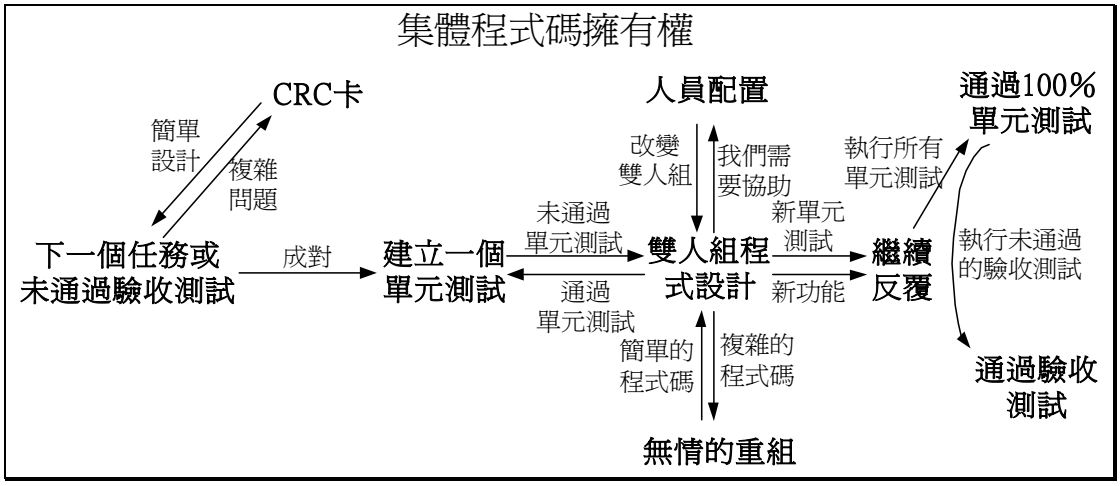
選擇一個系統象徵(Choose a System Metaphor)



選擇一個系統象徵以保持團隊在相同頁中命名類別及方法以使之一致性。你如何命名你的物件對於了解系統的整體設計及程式碼再使用都是非常重要的。能夠臆測哪些事務可能的命名方式；如果哪已是存在的並且是正確的；確實可以節省許多時間。爲你的物件選擇一個名稱系統這樣每一個人都能有關便無須特別說明，以努力贏得有關系統的知識。

例如；Chrysler 薪資系統是建構成一個生產線，在福特汽車的銷售員被建構成一個材料單(bill of materials)。有一種象徵稱之爲天真的象徵(naive metaphor)；那是以你領域本身爲基礎的。但不要選擇天真的象徵除非它是非常的簡單。

CRC 卡(CRC Cards)



在團隊中使用『類別(class)』、『責任(responsibilities)』及『合作(collaboration)』(CRC)卡設計系統。CRC 卡最的的價值是允許人們脫離程序性的思考模式並完全趨向於物件技術。CRC 卡允許整個專案團隊貢獻於設計。有更多的人可以幫助設計系統可以產生更多的好的概念組合。

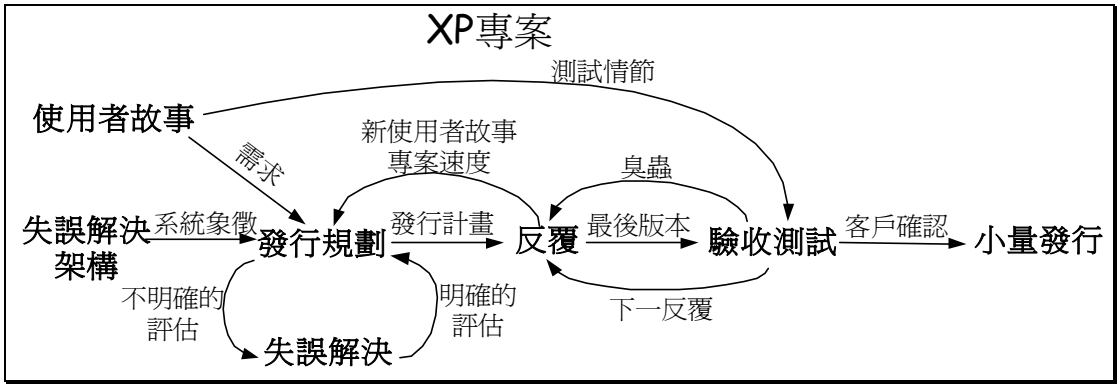
個別的 CRC 卡用以代表物件。物件的類別可以寫在卡片的上端，責任則列在卡片的左側合作的類別則列在每一個責任的右側。我們說『可以寫(can be written)』因爲只要 CRC 會議是活躍的；參與者全程只需要幾張有類別名稱的卡而無須每有一張卡都是填滿的。有一個簡短的範例是有關咖啡製造的問題。

CRC 會議進行的方式是有人模擬系統說出有關哪一個物件傳送訊息給其他的物件。逐次進行這個程序；則弱點及問題很容易的被揭露。在這模擬設計過程中設計替代方案可以被快速的探測中提出。

如果你發現太多人討論並立即移動卡片那麼只要限定只能站立及移動，當某一方坐下另一個才能站立。這個會議的工作失去控制；往往是發生在當一個艱難的問題終於解決而團隊變的煩躁時。

對於 CRC 卡最大的批評是缺乏撰寫設計(written design)。這是一般不需要讓 CRC 卡使得設計看起來很明顯。是否應該要有更固定的紀錄方式，一張卡片代表一個類別可能被寫滿並被保留變成一份文件。一個設計；一旦想像如果已建構完成並執行，讓它與一個人共處一段時間。

建立失誤解決方案(Create spike solutions)



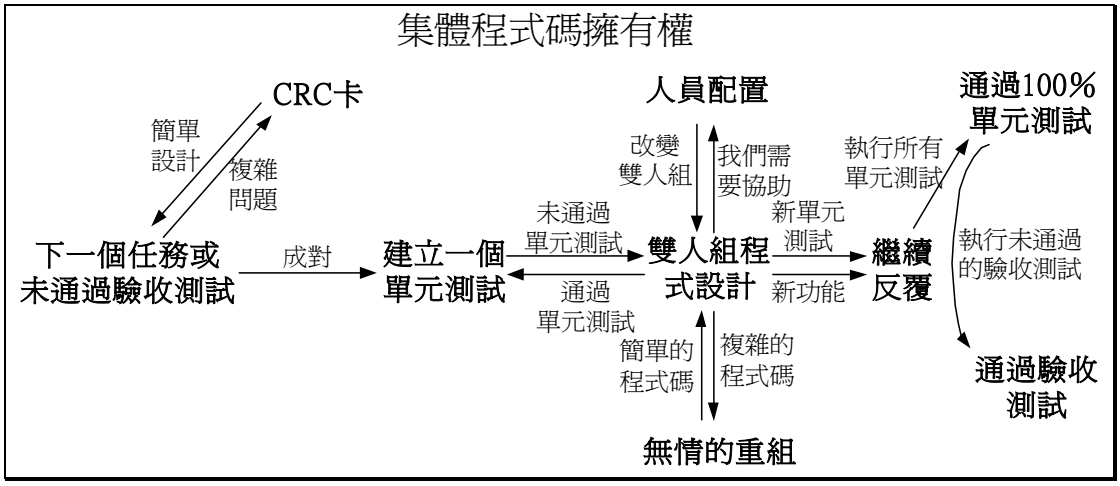
建立失誤的解決方案以便理解困難技術或設計問題的解答。失誤的解決方案是一個非常簡單的程式以發覺潛在的解決方法。建立一個系統只是處理在檢驗下的問題並忽略其他的考量。多數的失誤並不是值得保留的，所以期待把它丟開。目標是降低技術風險或增加一個使用者故事評估的可信賴度。

當一個技術性困難可能導致阻礙系統的開發；投入一對開發人員在這個問題一兩週以減低潛在的危機。

功能不要太早加入(Never Add Functionality Early)

以你預測隨後會使用到的額外的人員保持系統整齊(uncluttered)。只有10%得額外人員會被使用到，所以浪費 90%你的時間。我們都希望即時增加新的功能因爲我們可以實際的看到如何增加或者因爲它讓系統更好。立即加入新功能似乎看起來更快速。但我們需要不斷持續的提醒我們自己我們不是真正的需要這樣。額外的功能總會拖慢我們並浪費我們的資源。注視著未來的需求及額外的彈性，只集中心力於進度所安排今天應該做的事。

無情的重組(Refactor Mercilessly)



我們電腦程式設計師持續我們的軟體設計；直到他們變的難以使用之後。我們不斷的使用再使用不再容易維護程式碼，因爲這些程式碼仍以某種方式在工作而我們害怕去修改它。但那是否耗費成本而能有效去如此做？XP 的立場並不是如此。 當我們移除多餘、剔除未使用的功能及更新陳舊的設計我們是在重整。重整存在於整個專案生命週期中可以節省時間及提升品質。

無情的重整以保持設計簡單像你之前所做的並且能避免無需的混亂及複雜度。保持你的程式碼清楚簡潔而能容易了解、維護及擴充。確認所有事務表達一次而且只有一次。最終它能花費較少的時間生產一個系統；所以是值得推薦的。

重整有點類似禪宗。一開始很困難認爲你必須能夠放開去執行完美的設計，你必須想像並接受這個經由重整的設計；那對你而言猶如僥倖發現。你必須了解在過去你想像的設計是一個好的指引，但現在已是陳舊的事務了。

毛蟲是一種完美的設計可以吃掉大片的樹葉但它找不到配偶，它必須重整成蝴蝶才能搜尋天空找尋同類。讓你的概念開放到底系統應該是怎樣或不是怎樣並嘗試去看新設計就如你之前想像的。

客戶是隨時可得的(The Customer is Always Available)

XP 中比較稀有的需求之一是客戶是隨手可得的。客戶不只是幫助開發團隊同時是團隊的成員之一。XP 專案的所有階段皆需要與客戶在旁溝通；最好是面

對面。最好分派一或多個客戶給開發團隊。縱使如此；似乎是讓客戶綁死很長的一段時間而且客戶的部門可能偏向於派遣一個新手充當專家。你需要的是真正的專家。

使用者故事是由客戶經由開發人員的幫助所撰寫，以允許評估時間及指定先後順序。客戶協助確認大部分系統所需的功能已由使用者故事所涵蓋。

在發行規劃會議中客戶需要協商使用者故事的選擇以便放到每一個發行進度中。發行的適當時間可能也需要協商。客戶必須作決定；而這個決定影響其企業目標。發行規劃會議用於定義小的發行增量(incremental releases)以便讓功能及早發行給客戶，以便讓客戶及早測試系統並立即回饋給開發人員。

由於使用者故事並沒有細節所以開發人員需要與客戶討論以獲的詳盡的細節以便完成程式設計的任務。專案的任何各重要部份需要由客戶專任的人員確認。

客戶也需要協助功能測試。測試資料也需要建立並且其相對結果需要計算及驗證。功能測試驗證系統是否已完善並可以發行爲產品。有可能系統發行在即卻發現無法通過所有的功能測試，此時客戶需要檢視測試結果並允許系統繼續發行爲產品或駁回。

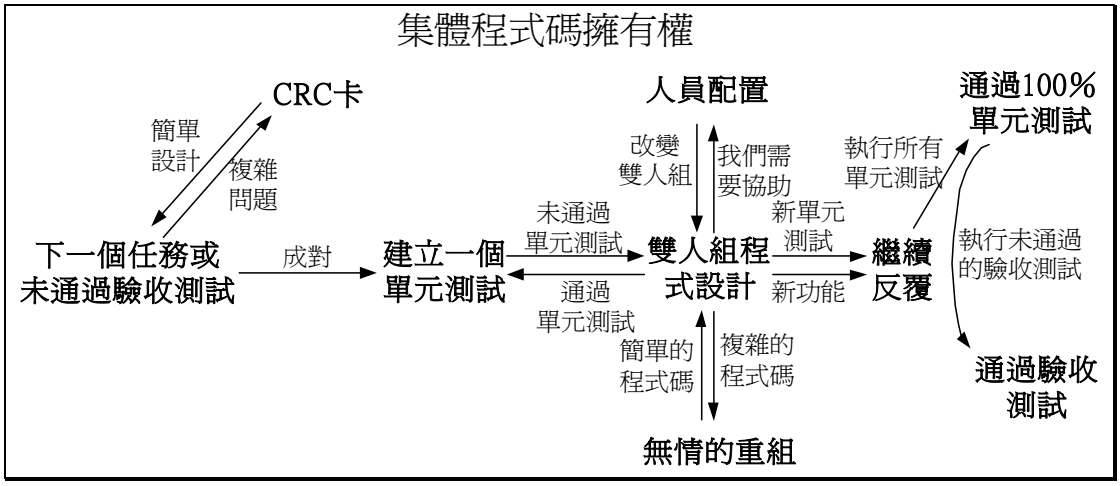
乍看之下客戶似乎須要花費許多時間在這裡；但我們應該了解客戶的時間在一開始停止請求詳細的需求說明之後便可以降低；及在交付的是一個可以真正實現客戶需求的系統時可以節省更多時間。

當有多個客戶都是部分時間參與可能產生問題。每一領域的專家常常相互爭辯，這是正常的。解決這個問題需要要求所有的客戶都能同時參與臨時的團隊會議以便統合意見差異。

程式碼編寫標準(Coding Standards)

程式碼必須依據共同協議的程式碼格式標準撰寫。程式碼標準維持程式碼一致性並方便整個團隊閱讀及重整。Smalltalk 專案可以使用 Smalltalk Best Practice Patterns 作爲程式碼標準。

首先撰寫單元測試程式 (Code the Unit Test First)



當你在設計程式碼之前先建立你的測試；你會發現你可以快速並容易的建立你的程式碼。建立單元測試及建立某些程式碼可以通過測試的時間相加約等於馬上撰寫程式的時間。但如果你已有單元測試你無須再建立可以節省你後續的時間。

建立單元測試幫助開發人員真正的考慮哪些是需要做的。需求在測試時明確的確定。此時對撰寫成可執行程式碼格式的規格便不會有所誤解。

你工作時同時需要立即回饋，當一個開發人員已完成所有需要的功能往往不夠明確。一些不知不覺的部分(Scope creep)可能發生並擴增以及錯誤狀況需要考慮。如果我們先建立我們的單元測試那麼我們便知道我們何時可以完成，因爲單元測試一直在跑。

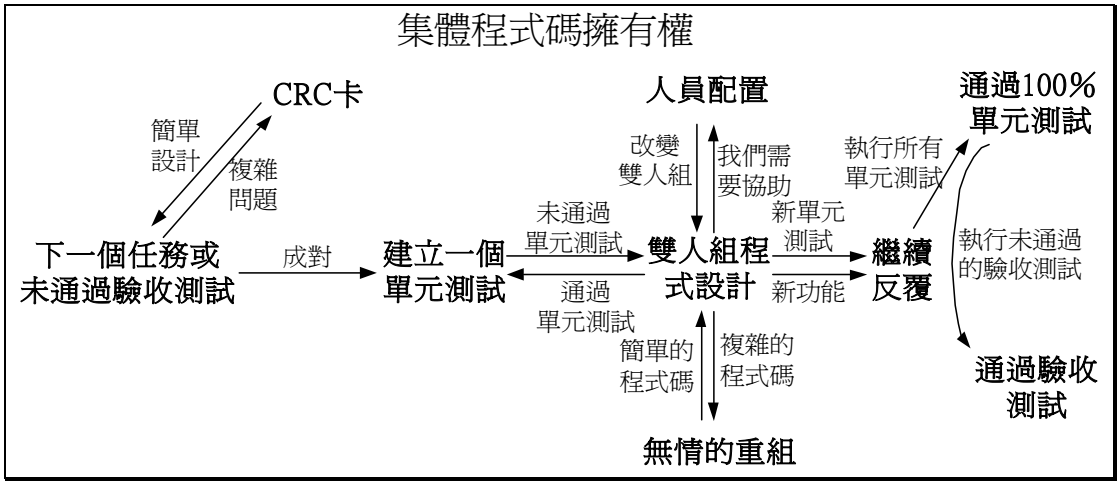
單元測試對於系統設計也有幫助。有些軟體系統很難做單元測試。這些系統一般是先建立程式碼然後再測試，且往往是由兩個完全不同的團隊來做。若建立測試在先你的設計將受想要通過客戶測試所有的值所影響。

先開發軟體單元測試是一種節奏。你建立一個測試來定義手上的問題某些小的觀點，那麼你建立簡單的程式碼便可以通過測試。接著你建立第二個測試。現在你可以增加程式碼並通過新的測試，但別再進行下去，除非你再有第三個測試。如此持續下去直到你沒有任何需要的測試。咖啡製造的問題顯示一個範例那是以 Java 所撰寫的。

你建構的程式碼需要簡單並且簡潔的；只實作你要的特色。其他的開發人員可以經由瀏覽測試來看看如何使用新的程式碼。輸入的結果若是未定義的將從系列測試中完全排除。

譯者：Areca Chen

雙人組的設計(Pair Programming)



包含於一個產品發行的所有程式碼是由兩個人共同於單一之台電腦中設計。雙人組設計可以提升軟體品質而沒有交付時間的壓力。那是直接的計算，但 2 個人在一台電腦工作所增加的功能將與 2 人分開工作時一樣多；除了兩人共同工作可以得到更高的品質。由於品質的提升將可在專案後續帶來更大的節省。

雙人組設計的最佳方式是兩人在螢幕前坐在一起，移動鍵盤及滑鼠，一個人打字並思考欲建構關於方法的策略，另一個人思考這個方法配合類別的策略。使用雙人組設計需要時間去習慣；所以別擔心如果你一開始覺得很笨拙。

循序式整合的程式碼(Sequential Integration)

在沒有控制程式碼整合下；開發人員測試他們的程式碼及整合相信都是適當的。但是由於程式碼的平行整合(parallel integration)模式需要在未曾共同測試之前有一個程式碼結合(combination)，會有許多整合問題發生而未能發覺。

若沒有明確的切割最終版本後續會有問題產生。這表示不只是程式碼還必須包括驗證程式碼的正確性的單元測試序列(unit test suite)。如果你無法擁有一個完整、正確及一致性序列測試你將追逐並不存在的臭蟲而對於存在的臭蟲毫無所知。

有些專案嘗試讓開發人員擁有特殊的類別，類別的擁有者便確保每一個類別的程式碼適當的被整合並發行。這樣可以減少問題但類別間的相依性可能還是很糟。這並未解決所有的問題。

可能的方法是指定一個整合的人或小組，整合多個開發人員的程式碼使得以控制。而且超過一週一次才整合一個小組多人的資源實在是太大了。在這個環境下開發人員處理舊的版本可能將錯誤的版本再整合到程式碼庫中。

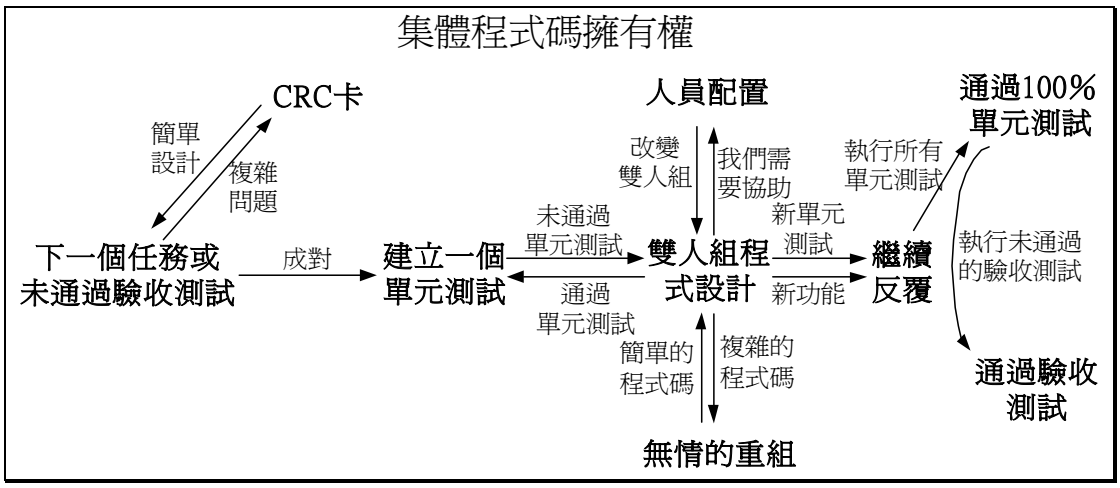
這種解決方式沒有深入到問題根源，你希望開發人員能夠平行處理；勇敢的改變系統需要的每一部份，但你也希望其結果不要把錯誤整合進去。就像一打蒸氣火車頭同時朝向轉轍器，這樣是會有問題的。與其限制開發是循序的(sequential)或者需要複雜的整合程序我們再想想這個問題。我們的火車頭可以全部進入轉轍器而不會相撞只要他們輪流進入就可以。我們也是需要這樣處理程式碼整合。

嚴格要求開發人員本身循序(或單一工作線)整合並結合集體程式碼擁有權是這個問題的簡單解決方式法。所有的程式碼輪流發行到原始程式碼儲存庫(source code safe)。這就是只有一個開發雙人組隨時整合、測試並發行改變的部分到程式碼儲存庫。單一工作線整合允許一個較後的版本仍被連貫一致的識別。

這並不表示你無法在你的工作站隨時整合你本身的改變與最後的版本。你只是無法發行你的改變到團隊而無需排隊等待。

某些鎖定的機制需要的，最簡單的方式使用實體的標記(token)；這個標記是在所有開發人員中傳遞。如果開發團隊是位於同一地點使用單一台電腦來管理這個工作。整合及發行程式碼以縮短持有鎖定標記的時間而使的獲取鎖定標記的時間減少。

隨時整合(Integrate Often)

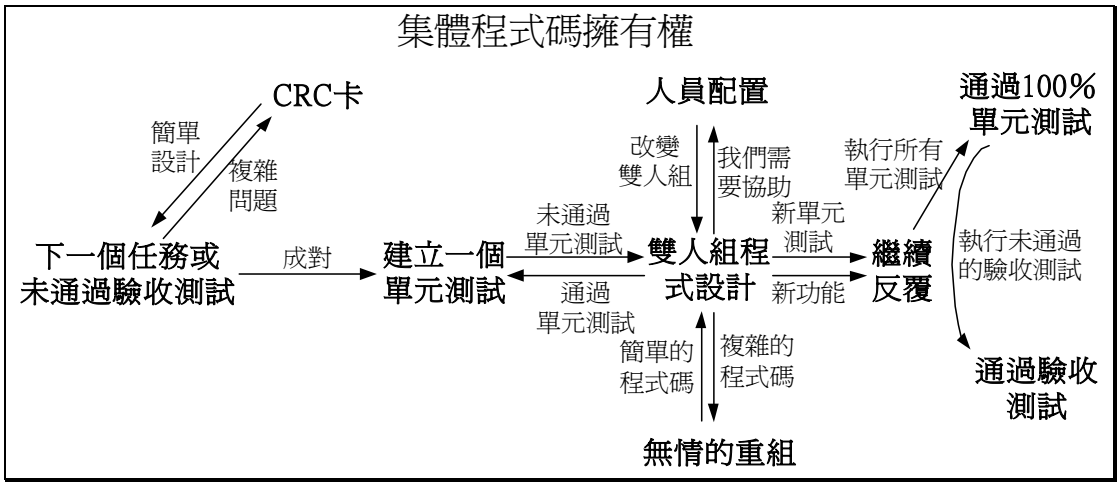


開發人員應該盡可能每幾個小時整合並發行程式碼到程式庫中。在任何情況下決不要緊握著改變超過一天。持續整合往往可以避免偏離或使開發的努力支離

破碎；尤其是開發人員沒有與其他每一個人溝通有關哪些可以再使用或共享。每一個人需要使用最終的版本工作。荒廢(obsolete)的程式碼盡量不要修改以免造成整合的困難。

持續整合幾乎可以早期避免或偵查到相容性的問題。整合是一種「現在支付或以後再支付」類型的活動。也就是說如果你在整個專案中持續少數量的整合；而不至於直到幾週後才在期限的壓力下匆忙的整合。總是在系統的最終版本下工作是最好的。

集體程式碼擁有權(Collective Code Ownership)



集體程式碼擁有權促使所有人在系統的所有部分貢獻新的概念。每一個開發人員可以改變任一條線上的程式碼如增加功能、除蟲或者重整。沒有人會在改變產生瓶頸。

這種作法一開始很難理解。很難想像整個團隊可以負責系統的架構，而沒有單一的主導架構設計師(chief architect)保有某些熱情的想像力似乎是不可能進行的。

但是詢問主導架構設計師一個問題並得到的回答卻是很明顯的錯誤是很平常的。那並不是你的領導設計師的錯。任何一個複雜的系統無法由一個人的記憶來掌握。其他的程式設計師辛苦的改變系統對於架構設計師的觀點是沒有利益的。不管你了解它與否你的架構已分散在你的團隊中了。如果整個團隊都已有某些責任於架構設計，難道他們不應該接受這個權責嗎？

這是每一個開發人員爲其開發的程式碼建立單元測試的工作方式。所有的程式碼的發行到程式碼庫都包含單元測試。程式碼被加入；臭蟲已被修復；而且舊有功能若被改變將由自動測試涵蓋，現在你真正可以依賴系列測試作爲你程式碼庫的看門狗。任何程式碼發行之前必須 100%通過整個測試序列。

譯者：Areca Chen

只要任何人可以改變任何類別的任何方法並發行到程式碼庫是需要的，當結合頻繁的整合開發人員很難注意到一個類別已被擴充或修補。

在實務上集體程式碼擁有權是實際上比由單一個人負責監視特定類別較可信賴的。尤其是當一個人可能隨時離開專案時。

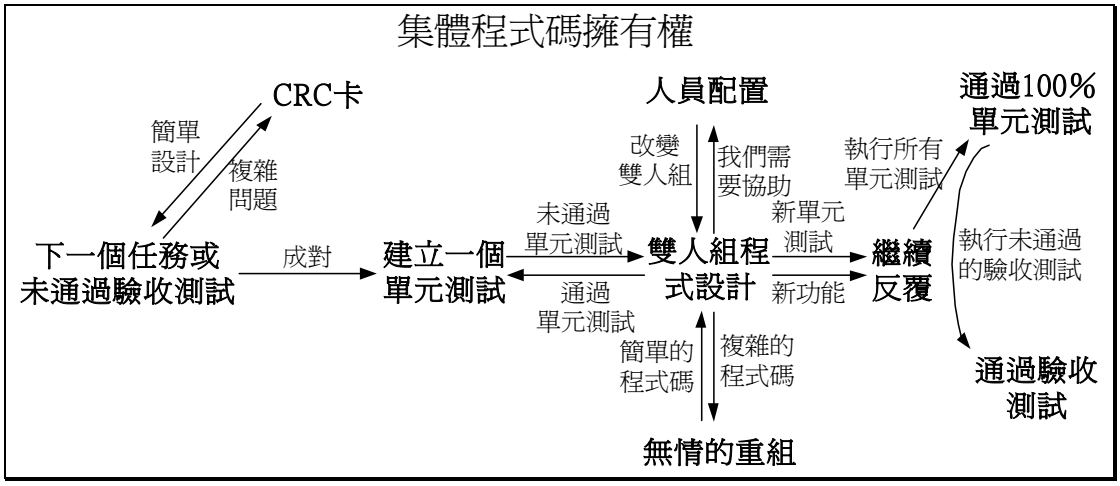
最後才最佳化(Optimize Last)

直到最終才最佳化。決不要嘗試猜測什麼是系統的瓶頸，而是量測它。讓它可以工作；正確的工作；快速的工作。

決不加班(No Overtime)

加班會使得團隊的精神及動力喪失。需要加班來即時完成工作最終並無所得。你應該使用一個發行規劃會議改變專案的範圍及時程，當專案已延遲時採用增加人力資源的方式來解決也不是好的方法。

單元測試(Unit tests)



單元測試是 XP 的基石。但 XP 的單元測試風格略有不同。首先你必須建立或下載一個單元測試框架以便建立自動單元測試系列。其次你應測試系統中所有的類別。一般瑣碎的 *Get* 即 *Set* 方法可以忽略。最後你應該在撰寫程式碼之前嘗試建立你的測試。

單元測試應該與其測試的程式碼一並發行到程式碼庫中。沒有測試的程式碼不得發行。如果發現有一個單元測試被忽略應即時建立此單元測試。

譯者：Areca Chen

致力於單元測試所耗費的時間對於期限所造成的壓力是最大的阻力。但在專案的生命週期中一個自動測試可以解省你許多耗費在找尋及預防臭蟲的時間成本。愈難撰寫的測試你愈需要；因爲愈可以解省你的成本。自動單元測試所獲得的回報會大於建構它的成本。

其他一般性的誤解是單元測試可以在專案的最後三週才來撰寫。不幸的是既使沒有單元測試開發人員還是會拖延並吃掉最後這三週。既使時間是足夠的好的單元測試系列是需要開發時間的。發現所有問題是需要耗費時間的。爲了要有一個完整的單元測試系列；當你需要你就要立即建立。

單元測試使集體程式碼擁有權變爲可能。當你建立單元測試你可以避免你的功能受到意外的傷害。在發行前要求所有的程式碼通過所單元測試可以確保所有的功能都是正常的。如果所有的類別由單元測試監控程式碼擁有權專屬於誰並不是必要的。

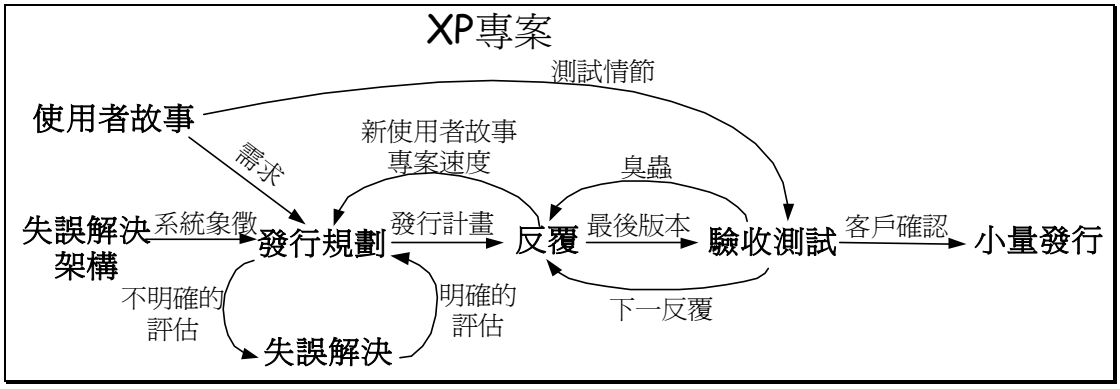
單元測試也讓重整變爲可能。在所有的小改變情況下單元測試都能驗證；結構的改變並不會造成功能的改變。

建立一個單一全體的(universal)單元測試系列以便確認(validation)及復原(regression) 測試使得頻繁的整合變爲可能。如果執行你本身最後版本的測試系列；快速的整合現有的任何改變是可能的。當一個測試失敗表示你的最後版本與團隊的最後版本並不相容。每幾小時修復小的問題所花費的時間低於在期限終了前修復大的問題。有了自動單元測試便可以快速合併一組改變成最後發行版本。

一般增加新功能需要改變單元測試以反映新的功能。雖然那可能在程式碼及測試中帶入臭蟲；但在實務上並不多見。偶而發生測試是錯的但程式碼是正確的，在除蟲的過程中將會顯現並修復。希望在撰寫程式碼之前建立與程式碼獨立的測試，可以建立檢驗與平衡並且大大改善在第一次就得到正確的機會。

單元測試提供一個復原(regression)測試及確認測試的安全網因此你可以有效的重整及整合。就像在馬戲團中所說的決不在沒有安全網下工作。撰寫程式碼前先建立單元測試是在凝聚需求(solidifying the requirements)、改善開發焦點及避免遲緩交件(creeping elegance)有所幫助。

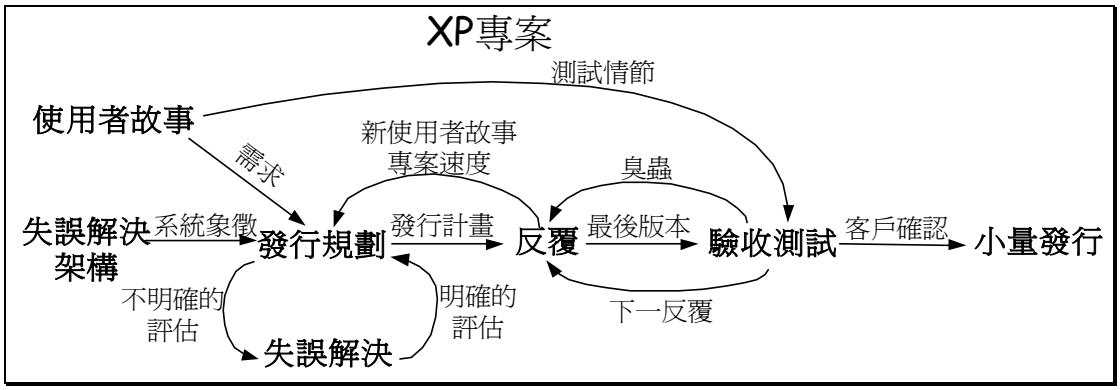
發現臭蟲(When a Bug is Found)



當發現臭蟲建立測試以監視臭蟲被修護。產品中的臭蟲需要一個驗收測試來監視。開始除蟲前先建立驗收測試可以幫助客戶簡明的定義問題及與程式設計師溝通問題。程式設計師若沒有通過測試便可以在問題修護當中聚焦於他們的重點及所知的部分。

若驗收測試失敗，開發人員可以建立一個單元測試從程式碼特定的觀點顯示其不足之處。單元測試失敗當臭蟲修復後可以立即回饋給開發人員。當單元測試100%通過後；再執行之前失敗的驗收測試以確認臭蟲已修護。

驗收測試(Acceptance tests)



驗收測試是由使用者故事建立的。在一個反覆當中從反覆規劃會議中選擇的使用者故事將轉換成驗收測試。客戶指定特定的情節(scenarios)用來測試使用者故事是否正確的實作。一個故事可以有一或多個驗收測試來確保其功能正常運作。

驗收測試是黑箱(black box)系統測試。每一個驗收測試需要一些期望的系統產出結果。客戶負責查證驗收測試的正確性並檢視測試評分以決定哪一個未通過的測試最優先。驗收測試也用於在產品發行之前的復原測試。

一個使用者故事在未通過驗收測試之前並不代表完成。這表示新的驗收測試必須建立，此時每一個反覆或開發團隊的進度報告是 0。

品質保證(Quality assurance)(QA)是 XP 程序最基本的部分。在某些專案 QA 是由其他小組來管理，而在其他的 QA 將整合到開發團隊本身當中。在這兩中情況下 XP 的開發需要與 QA 有更密切的關係。

驗收測試應該自動的以便可以隨時執行。驗收測試的評分在整個團隊中公佈。對於團隊而言他們需要在每一個反覆中負責安排進度並修復任何無法通過測試的部分。

驗收測試的名稱隨著功能測試而改變。最好是能反映其目的，以保證客戶的需求可以符合而且系統可以接受。

團隊擁有進度時程(The Team Owns the Schedule)

一個正確的團隊可以達成的目標超過一個人可以想像的。這些團隊也就是高學習能力的組織，他們經由團隊擁有計劃擁有權而提昇有效的專案管理實務，同時他們希望軟體開發專案以兩部分來管理：一個專案可以做得到的進度時程(發行計劃)及一個程式設計(工程)的反覆規劃任務。

這兩部分並不一樣，大部分的專案管理人員無法讓工程團隊的工作符合工程規劃的節拍，相反的他們指派工作並強迫團隊使用一個專案計劃而其焦點只是可以交付給客戶。

事實上，我們都知道這個錯誤的作法並不符合好的物件思考，更不用說應用可再使用的框架、測試、重整、XP 等等。因此我們成功的使用了終極進度協商(extreme schedule negotiation) [也就是發行規劃]。

簡化設計就是容易維護(A Simple Design is Easier to Maintain)

我們很難再重建福特的 VCAPS 系統。其原始系統的資料庫是不再支援的 GemStone。也沒有辦法直接更新因爲這個系統太複雜了。

譯者：Areca Chen

我們建構所謂的 Replicator 將資料從舊系統移到新系統中。建構它我們耗費了好幾個月。但 Replicator 有很大的彈性而且是以框架來建構。

當新系統實作完成我們往往需要改變 Replicator。但它太複雜以致要改變非常困難也讓我們進度緩慢。我們需要一個人全職來維護它。當任何團隊成員要改變 Replicator 他們必須等待 Replicator 負責人有空去做。

唯一可做的就是集合團隊成員共同設計比較好的解決方案。當我們執行了許多想法之後得到了一些比較好的概念。我們執行失誤解決方案以評估何者是最佳的。

於是我們在建立新的 Replicator 時撰寫自動單元測試；因此我們盡可能可以重整複雜的部分。當我們完成新的設計幾週後並且我們有約 1/6 的程式碼。

新的系統並沒有一個框架的基礎，它只有做需要做的。我們發現它更容易維護並且只要我們需要更容易增加我們所需要的。我們發現我們也能集體擁有。現在所有人都可以快速的改變他們所需要的。

Don Wells
eXPerience Software

你逐漸不需要它(You Aren't Going to Need It)

在我們目前專案的初始需求及設計會議，我們持續有人想要加入許多“未來”的需求到第一階段。我們的答案往往是「我們會寫下卡片這樣我們不會忘了它，但我們會等到真正需要的時候才會放入設當中。」

這個策略挽救了我們大量的不幸，尤其是考慮系統的基礎建構區塊(fundamental building blocks)之一。我們設計一個模式用於定義可能的產品提供的組合(combinations)。

隱藏的困難是；約六個月的時間後一個新的共同模式(corporate model)就要發行。我們得到許多需求試著要去預測而且塑造不可避免的共同方向。我們抗拒，並且相反的塑造系統的產品定義以反映目前企業實務，而且只有相關於我們專案的那些部分。我們保持設計盡可能的簡單而且仍能滿足所有的專案需求。我們實踐「你並不需要它」的理念。

六個月之後共同的模式持續到次年。同時我們的專案可以說是完全的更新而且不同的模式以擴充需求(expanded requirements)爲基礎。

如果我們已經嘗試在共同模式猜測我們需要幾個月的時間重新遷移到新的模式。因爲我們保持我們的設計非常簡單，我們可以簡單的發展我們的模式到新的模式。我們預測那只要一兩週的調整，很明顯的有實質上的節省。

系統象徵可以簡化設計(A System Metaphor can Simplify the Design)

當我們改寫福特的 VCAPS 系統，我們想要選擇一個系統的象徵但發現那是困難的。我們從許多來源得到資料；這些料看來似乎沒有共通性(commonality)。

當我們發現所有的資料可以一致的表示成材料單(bill of material)；因而有所突破。所以我們選擇『材料單』作爲我們的系統象徵。在舊系統往往很困難找到正確的資料。在新系統很明顯的它將可以。

從上到下我們的新模式有相同的象徵。當我們一路走下去使用新的象徵我們可以衡量其一致性並節省我們 95%的資料庫大小。同時現在我們只有一種也是唯一的一種存取資料的方式；不管它從哪裡來或它如何顯示。

這種一致性也簡化我們的演算法並且減少處理時間。我們希望平均約 15 分鐘的處理時間變成『立即回覆』。

Don Wells
eXPerience Software

整體大於部分(The Whole is Greater Than the Parts)

一年以前我們應用雙人組程式設計並且發現雙人組改善設計品質及實作而對於生產力沒有犧牲。事實上；依據我們經驗，整體的生展力證明大於個別的部分。(我們唯一美中不足的部分是有些人無法共同工作，而且有些個人太個人主義以致使得雙人組設計無法順利推行。)

Mark Bradac
Lucent Technologies

雙人組設計的經驗法則(Some Pair Programming Rules of Thumb)

我們從福特的 VCAPS 專案中建構雙人組程式設計中學到一些基礎規則：

1. 決不將兩個新手雙人組放在一起(總是一個老手帶一個新手)。
2. 當一組人員選擇分開工作但負共同的責任，這並不是真正的雙人組程式設計。
3. 若兩人無法在相同的螢幕中觀察發生的事情這並不是真正的雙人組程式設計。
4. 所有人都是雙人組的；決不允許單獨存在。
5. 每個人必須信賴其他所有人；而建立團隊中彼此的信賴需要花費時間的。

我想大多數的人都曾經在雙人組程式設計中失敗過(但最終是成功的)必須學習這些基本規則。

Jeanine De Luzman
Ford Motor Company

雙人組設計控制莽撞的程式碼(Pair Programming Reins in the Cowboy Coders)

XP 不只是在不好的環境中差的程式設計實務的法典編纂(codification)。關鍵點是限制(閱讀改善)；雙人組設計以 JANGIT(只增加新的垃圾而忽略測試 Just Add New Garbage Ignoring Tests)的用心加諸於莽撞程式設計師(cowboy programmers)的效果 。即使你偏向於在你的團隊中使用莽撞的程式設計格調，將他們以雙人組方式配置都包含一個有經驗的終極程式設計師實質上保證程式撰寫的正確性而且需要的測試也絕對不會少的。

即使兩位莽撞的程式設計師放在一起，其結果仍然較之任一個單獨所寫的爲佳。XP 提供每對程式設計師一組檢測及平衡；因爲沒有任何兩個人所想的是一致的。當結合 XP 的其他規則其結果是排除差的程式設計實務及差的環境。

雙人組也排除由於不高明的程式設計師所設計的不良程式碼及文件的問題。

我所說明的重點是經由個人的經驗，我不認爲雙人組設計可以幫助我改善我的程式碼並增加速度。所以我完全是懷疑的，就如你可能說的；我的觀察已有改變一點點。

Tom Kubit
eXPerience Software

雙人組設計可以減少猶豫不決(Pair Programming Reduces Indecision)

XP 最大的效力之一是雙人組程式設計。我們發現雙人組成員之間的互動更多於個別雙人組之間的總和。即使最有經驗的程式設計師發現他們產出的設計及程式碼較他們個別所產生的層次還高。我們在 VCAPS 專案中發現他們在雙人組時是比較不會猶豫不決；一些意見丟來丟去；而解決空間迅速集中在於相關的優缺點的討論。

Kevin Bradtke
eXPerience Software

不犯錯，雙人組是辛苦的工作(Pair Programming is Hard Work)

我曾經在我的目前位置實驗雙人組程式設計。我很幸運的緊鄰著程式設計師工作而且我們契合的非常良好，而且他們也非常瘋狂的讓我嘗試一些 XP 的實務。我們曾經使用 50%的時間於雙人組程式設計，其結果非常令人鼓舞。我的觀察是：

1. 從某些角度而言那是辛苦的工作。因爲我的夥伴要求我證明所有不明確或不愉快的事情。

2. 對我而言當時那不像是更有生產力－－當你一直在溝通結果並沒有加快速度，尤其是在劇烈的爭辯時。但我們還是持續下去，我們發現我們寫下一大堆卡片我們都了解並喜歡的。

3. 有時我們兩個會有衝動，我們並沒有衝動的決定，所以我認爲哪不是問題。

4. 漫無目標的夥伴可以控制干擾並持續撰寫程式，這是大的生產力優勢。

5. 當我爭論獲勝，感覺上全然沒問題。當我輸了是因爲我的意見不是最好的，我也感到滿意，因爲我知道我的夥伴剛剛幫助我避免了不良的程式碼設計。

6. 有時我喜歡分歧(fork)，由一個人做網頁搜尋或者撰寫一個快速測試程式，但我們一般合作太久。我們兩人都不希望在程式碼還有疑問時讓另一個離開。

我們的團隊領導注意到我們雙人組的工作，並在下一個專案中正式成爲雙人組。我無法想像有更好的，我們是雙人組的工作。

Wayne Conrad

雙人組的實驗性證據(Pair Programming Experiment)

我們在猶他大學執行一個實驗其中有 42 個高級程式設計師。其中 14 人單獨工作。其餘的雙人組工作。所有的學生完成相同的指定工作。

雙人組程式設計明確的最多不會超過兩次。在第一次指定工作(我稱之爲成行指定"jelling-assignment")；雙人組較單獨組多花費 60%的程式設計師時間。再第二次指定工作，他們習慣了雙人組程式設計的方式，雙人組較單獨組多花費了 20%的總時間。再第三次指定工作，雙人組較單獨組多花費 10%的時間；也就是說如果單獨組花費 10 小時則雙人組共同工作 5 小時 15 分。

在所有的狀況下，雙人組多通過 15%後開發測試(post-development test)的狀況。

同時超過 90%的程式設計師更喜歡程式設計並且他們在雙人組工作時覺得更舒服。

這個實驗的後測試(posttest)結果，所有的學生個別的工作完成一個指定工作。其中有一個學生談到要回到單獨程式設計時說：「沒有我的夥伴，我好像失去半邊的腦袋」。

Laurie Williams
University of Utah

程式碼檢查可能是有害的(Code Reviews Considered Hurtful)

在你投入的程式碼建立之後；對於你的程式碼往往很容易變得情緒化。一個正式的團隊的檢查程序會產生壓力的狀況而且在開發人員及檢查人員之間產生情緒的反應。

並沒有好的方式由許多人去檢查某人的程式碼並建議修改而不會導致對個人的評論。這種感覺就像從各個方面被攻擊；而且哪裡需要改變以滿足檢查人員並不是很清楚。

我曾經目擊一個開發人員在檢查的程序中感到個人的受迫害。我曾是檢查團隊的一個成員而程式碼是由一個沒有經驗的開發人員所撰寫。只經過兩次檢查；這個開發人員要求管理人員不要再有任何檢查。

我們向她保證這不是個人的攻擊，但她對於專案的熱情已經消失了。她覺得完全失去信心。我們都想要去幫助她，但沒有人能夠跟她坐在一起並指導她。我擔心檢查的工作幾乎是爲她做的因爲我能夠感覺壓力升高，而且認知的時間已經浪費了。

雙人組設計改變這種環境；經由評論及競爭達成學習及合作。程式設計夥伴必須對所有其他人說明他們所做的，一個是老師一個是學生，使得角色反轉。學生被鼓勵參與新意見或對於舊意見新的扭轉，以獲得整體的信心。

藉由介於兩個開發人員之間的討論以取代上司的簡短指示，沒有任何評論是隱藏在建議後面，只有相互的發現及意見一致。其最終程式碼往往是比較好的；因爲哪是經過兩對眼睛看過的。在完成一天的工作時你會覺得充滿成就感而非敵意。

譯者：Areca Chen

Tom Kubit
Interface Systems

XP 與資料庫(XP and Databases)

再 VCAPS 專案中我們面對 XP 與一個大資料庫的問題。我們的資料庫是物件導向，但一個關聯資料庫可以以相同的方式控制。記住如果你實作使用者故事時依據客戶的評價首先建立你的資料庫表格及正規化將會比較穩定快速。

這個關鍵是由 Kent Beck 所建議，假設資料庫是容易改變的。關聯資料庫的建構是讓它很有彈性。Kent 也建議當有些東西是非常困難的盡量多發時間總是好的。在工作中得到好的作法時那就不再是困難了。習慣你的資料庫時常變遷(migration)，你就不再容易犯錯。

VCAPS 解決方案被區分成一個金級、一個銀級及許多銅級的資料庫版本。金級是類似產品的資料庫。銀級是金及資料庫的變遷版本。每一個開發人員有一個銅級的資料庫；是銀級資料庫的變遷版本。

銅級資料庫在開發人員的程式碼發行時變成銀及資料庫。當產品資料庫已變遷完成時變成金級資料庫。

其中最重要的是每一個人都可以很容易的取得金級或銀級資料庫的拷貝快速的當作銅級資料庫來使用；而且我們持續追蹤資料庫改變的部分。

設定執行稿(scripts)把資料庫當作檔案複製是非常有用的。你需要一個人去將銅級的資料庫升級爲銀級資料庫，而另一個人將銀級的目前的版本回存到區域的主機。

每一個資料庫都有相同的使用者 ID 及密碼。開發人員及資料庫溝通可以使用相同的 ID 及密碼在任何的資料庫。

每一個開發雙人組發行新的程式碼到程式碼庫並且同時升級其銅級資料庫爲銀級(在通過 100%單元測試後)。此時新的銀級資料庫與目前發行的程式碼是完全的同步。這是很重要並且需要訓練去維護。

在任何時候即時的金級資料庫與產品發行相配以發行產品，在任何時候銀級資料庫與大部分的目前開發發行相配以便 100%執行單元測試。

開發人員可以隨時整合因爲複製銀級資料庫並且同時檢查目前發行的程式碼是很方便。單元測試可以 100%執行。開發人員加入他們的改變；執行任何的資料庫變遷；並且等到單元測試通過 100%時再次整合所有的程式碼及資料庫。這種方法非常快速只需花費少數幾分鐘。

譯者：Areca Chen

爲支援測試我們有建構資料庫的程式碼及事先定義的一組測試資料。某些資料是由測試產生；但提供資料庫協助的範例。我們發現每週建立新的金級資料庫及變遷爲銀級資料庫以避免不可避免的資料毀壞(corruption)並確保我們的變遷部分正確是有用的。

我們偶而在變遷時發生問題。我們在持續追蹤改變的部分並不是做的如我們應該做的那麼好。在接近終點時我們結束了使用資料庫維護物件(DB maintenance object)中的方法來公式化(formulate)我們的變遷。這使得產品變遷更可信及無誤(non-event)。

Don Wells

整合的間隔可以縮小到每秒(Code Integration Can be Reduced to Seconds)

VCAPS 的成功可以歸功於持續整合的作法。[持續整合是一種朝向整合的態勢；也就是盡可能的隨時整合]我們發現一些成功的關鍵要素：

1. 單元測試是絕對必要有的。除非通過 100%的單元測試否則決不可發行。如果他們不這麼做；你絕不能再做任何改變。

2. 每一個人對所有的程式碼都有集體程式碼擁有權。亦即任何時候程式設計師接觸程式碼必須改善、重整及簡化。

3. 必須有一個發行中心(release station)。任何人不得自其個人工作站執行發行動作。

隨時整合!!整合期間愈長，你會愈痛苦。持續整合(猶如一個口號)在我們的環境(Visual Works Smalltalk and GemStone Smalltalk)中被執行是每分鐘的重要事情；一般而言是每秒。開發人員在實務上每天整合許多次。如果整合你的改變期間愈長；則發行的單位工作太大。程式設計雙人組要爲其構築的整合困境；或天堂；負責，他們會喜歡哪一個.....

Jeanine De Luzman
Ford Motor Company

最後才最佳化因爲那並不如你所想像的那麼遲

(Optimize Last Because it May Not be as Slow as You Think)

在福特的 VCAPS 專案中我們需要加入某些新的功能。以其他某些系統中的程式碼爲基礎；在新的部分執行 8 小時即使在我們勉強增加一點點執行效率也將至少執行 16 小時。我們增加新的功能盡可能的簡單。我們忽略速度的考量有助於程式碼簡潔及可維護的考量。我們讓系統可以執行、讓它正確再考提昇速度。

當我們完成程式碼我們再星期五下午啓動並預期會有幾天的執行時間；因爲我們還沒實作最佳化。但是我們在一小時之後得到結果；我們嚇到了。經由持續保持我們的設計簡單及可了解性；我們的管理可以避免許多「小聰明卻是大笨蛋(penny wise, pound foolish)」的最佳化類型而讓其他部分的程式碼變成犧牲品。

此時我們可以考慮最佳化，但是因爲它已經較系統其他部分更快速我們無須再做任何最佳化。

Don Wells
eXPerience Software

單元測試是值得投資(Unit Tests Are Worth the Investment)

一年以前我們偶而發現 XP 網頁而且應用當時支持的兩個觀念：雙人組設計及自動單元測試。一開始我們計劃嘗試自動單元測試。我要報告的是單元測試的努力獲得巨大的利益。我們實作 Kent Beck 的 Simple Smalltalk Testing：使用樣式的方法(Patterns approach)並實作一個『殺手(killer)』使用者介面以組織、初始化測試及收集結果。初始的努力及學習曲線非常值得投資時間。我們在品質上有戲劇性的提昇同時在整個測試、尋找及修復的間隔上降低許多。

Mark Bradac
Lucent Technologies

單元測試可以節省我們的時間(Unit Tests Could

Have Saved Us Time)

在福特 VCAPS 專案期間我們導入自動單元測試。它變成將單元測試包含入任何的新功能或維護，但有一個案例是一個管理人員抱怨如果事事測試將無法於時限內交件。在這個情況下特殊的管理(dispensation)是被允許的。在這些程式碼發行後花費約 32 開發小時以搜尋一個問題更別說數不清的客戶延遲時間。我們發現開發團隊的錯誤是由於其工作採用舊版本。由於我們自己的緣故單元測試在問題發生之後來承擔這個責任。而讓新程式碼適應現有的單元測試祇花費一個小時而已。

Don Wells
eXPerience Software

一開始便測試使得程式碼變的可測試(Testing First Makes the Code Testable)

Massimo Arnoldi 與我爲一個保險公司的預定支付部門工作。需求及程式碼很難掌握因爲所有的特殊狀況：如果支付太多怎麼辦；如果支付太慢怎麼辦；如果一次支付兩個月怎麼辦等等。

我們花費半天的時間嘗試爲這種情況提出測試及程式碼而不成功。每次我們讓一個測試可以工作，我們發現需要其他三個。

在吃過一頓飯後，我們決定解決一個簡單的問題，我們是否能夠一次測試及撰寫所有決定是否預定支付的程式碼？如果所有的都符合我們可以預定，否則我們必須詢問人們的介入。我們做了一些測試而有一個物件通過，使用的時間是 15 分鐘。

其次；我們能正確的預定一個支付？在第一次測試正確的通過，我們可以認爲來源帳戶(source account)已有正確的平衡。測試及程式碼撰寫是很瑣碎：一個測試、一個物件、一個方法，結果花費 5 分鐘。

如果我們未事先撰寫測試程式碼我們從不建構兩個物件。第二個物件將非常複雜如果我們無法依賴第一個通過測試。以測試爲首的結果：乾淨的設計、正確的行爲、簡單的測試及簡單的程式碼。

譯者：Areca Chen

我現在嘗試應用這個策略在我碰到的所有困難問題上。「什麼是細微的部分；這些細微部分的組合我是否有信心確保所有細微的部分都能工作？」我並不總是發現細微的部分都是正確的。有時幾個月或幾年，在這段期間我有許多測試我的不是最理想(less-than-optimal)的設計，當看到它醜陋面目的背後，我需要將所有的原始碼重新改版。

Kent Beck
First Class Software

驗收測試不只是排除臭蟲它們增加穩定感

(Acceptance Tests Don't Just Eliminate Bugs

They Add a Feeling of Stability)

當 XP 首次導入到福特的 VCAPS 專案時並沒有自動驗收測試(automated acceptance tests)，它花費了一些時間去加入這個範圍(coverage)。只要需要改變時我們爲所有的新舊功能加入測試。約一年後我們估計由 40%有測試涵蓋而問題統計單也降低 40%。我們不認爲這是偶然。

但客戶有不同的紀錄結果。在它們抵達產品環境之前的除蟲只有非常少數的緊急產品發行。在以往；由於臭蟲的關係需要立即修復而連續好幾天一天發行好幾次產品是非常平常的事。驗收測試改善系統的品質到一個程度；其中一個產品發行真正是再次的發行(re-released)因爲一個緊急修復是需要的。

客戶經驗了這個有高度穩定的系統，他們對於我們及系統有更高的信賴度。客戶同時注意到較少的發行次數往往由於快速及污濁的修復(dirty fixes)而導致大量落入假性的臭蟲(spurious bugs)。

Don Wells
eXPerience Software

驗收測試需要簡單的更新(Acceptance Tests Need to be Easy to Update)

在福特的專案中；於單元測試及驗收測試中使用單元測試框架(unit test framework)。當時這看起來似乎是最簡單的事情可以工作的。但爲了使用更一般性單元測試框架我們必須支援程式碼大量的資料，亦即；我們的資料庫建構成：＝(指定)的指令。其結果是需要一個人熟悉驗收測試資料來更新或者加入新的資料並測試。

我注意到那是很困難的。但不管如何還是持續。這是我的錯誤；我需要建立一個工具來維護我們的測試。一個由客戶設計特殊領域的工具可以幫助我們及客戶建立驗收測試。我的理由是沒有足夠的時間建立它。我們幾乎使用更多的時間很辛苦的建立測試資料。

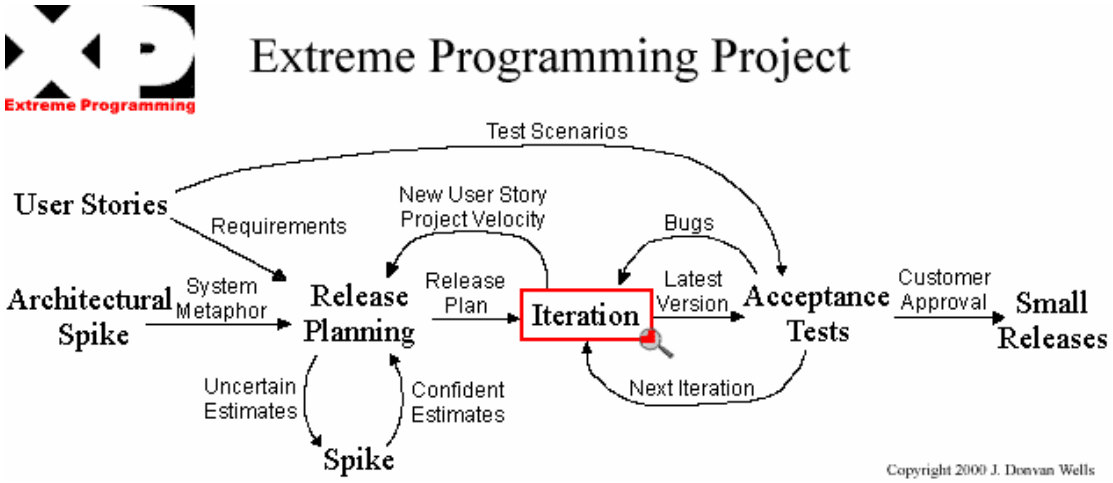
幾年後 VCAPS 已進入其生命週期的維護部分。驗收測試仍然用於尋找整合的臭蟲，但沒有增加新的測試。維護工程師所發現的是幾乎沒辦法增加新的驗收測試並且對於那些沒有設計工具讓所有資料容易控制的現有部分難以維護。

對專案我所做過最好的事是事先建立工具。回顧過去使用工具是多麼簡單的事。建立工具可以在專案生命週期的所有階段幫助我們並節省我們的時間。

Don Wells

eXPerience Software

發行計畫(release plan)



譯者：Areca Chen

使用者故事撰寫完成之後你可以使用發行規劃會議建立發行計劃。發行計劃嚴謹的指定每一個系統發行中哪一個使用者故事要實作及這些發行的日期。發行計劃在反覆規劃會議中給定一組使用者故事給客戶去選擇在下一反覆中要實作的部分。這些被選出的故事便轉換成個別的程式撰寫任務；在這個反覆中實作以達成故事的目的。

故事也在這個反覆中轉換成驗收測試。這些驗收測試在反覆中執行；並在後續的反覆中當故事已被正確的達成時作爲確認並持續正確的工作。

當專案速度在一些反覆中改變或在某任何情況下有許多反覆超前；安排一個發行規劃會議與你的客戶討論並建立一個新的發行計劃。

發行計劃通稱爲『委託進度表』(commitment schedule)，這個名稱被改變以便更準確的描述它的目的並且與反覆規劃更一致性。

負載因子(Load Factor)

負載因子是在專案速度變的更普遍之前一個專案如何被追蹤。負載因子等於完成一個由開發人員切割任務並估計的理想執行天數的實際日曆天。亦即；試想一個任務將花費你一天的時間；而且是你可以全神灌注於這個任務別無旁務的情況下，現在描繪你本身在真實世界嘗試要去完成它，實際要花費的總天數就是負載因子。

負載因子從 2 到 5 是正常的。如果你要推測負載因子以便開始；你需要考慮人的經驗及所使用的技術。2 是最樂觀的(optimistic)，3 最典型的(typical)，而 4 及 5 是當專案使用的是不熟悉的技術的情況。Ron Jeffries 建議在新專案中使用 3 作爲初始的推測。

在初始推測之後你必須在整個專案中持續衡量及追蹤負載因子是否更好的狀態。

負載因子不能用以比較兩個專案，每一個專案及團隊都是不一樣的而且有不同的原因型成不同的負載因子。

如果負載因子變動過大時使用發行規劃會議來從新評估及協商發行計劃。當系統已經推出成爲產品由於維護的任務；負載因子可以再改變。

專用發行電腦(Dedicated Release Computer)

如果開發團隊是都在同一個地方;使用一部單一電腦專用於持續的發行可以讓工作做得更好。這部電腦就像是一個實體的標記(token)控制著發行。也可以讓開發人員看到系統目前的最終狀況。開發人員有一個問題就是程式碼最終版本的仲裁，這部電腦可以讓開發人員看看誰已發行、在什麼時候發行。當這部發行電腦已被佔用所有其他的改變不得發行以確保穩定性。

最後的組合單元測試系列可以在發行前執行。因爲只使用單一電腦；系列測試可以確保是最新的版本。如果單元測試通過 100%；則改變的部分可以發行。如果無法通過測試則改變的部分需要除蟲或倒退到開發人員的工作站上除蟲。

單元測試框架(Unit Test Framework)

一般人有一個誤解認爲單元測試框架是測試工具(testing tools)，其實它是開發工具就像是你的編輯器及編譯器。不要在專案最後階段才使用這個功能強大的工具，務必在整個專案過程中使用它。你的單元測試框架可以幫助你格式化需求、澄清你的架構、編寫程式碼、除蟲、整合程式碼、發行、最佳化當然還有測試。

單元測試框架並不難直接建立(from scratch)但大部分語言已經有建立並可以自 XProgramming.com 下載。

線上看執行結果

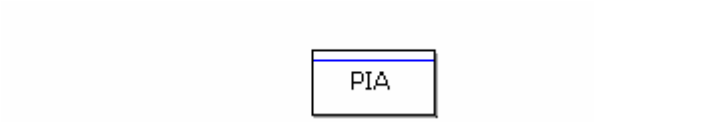
範例

設計一個咖啡製造的模擬(Design a Simulator for the Coffee Maker)

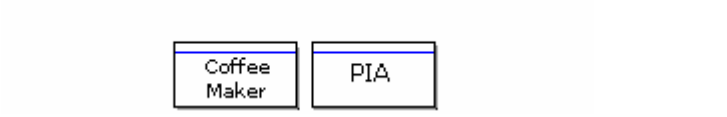
在 XP 中實現設計有三種方法：就是 CRC 卡、重整及雙人組設計。CRC 卡可以視爲設計的策略層次，雙人組設計是技術層次，而重整包含兩者。

爲了讓我與團隊工作愉快，所以讓我們從 CRC 卡開始吧。

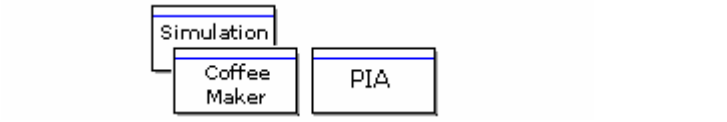
從我們的硬體介面開始。我們知道我們擁有可程式化介面接合器(programmable interface adapter (PIA))。我們可以以一張卡片代表一個物件就是 PIA。



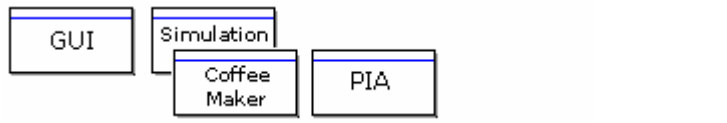
現在讓我們加上咖啡製造機的卡片假如它是一個單一物件。它可能是也可能不是；但直到現在我們是在設計一個模擬器(simulator)。把它放在 PIA 右側因爲它要以它爲介面。



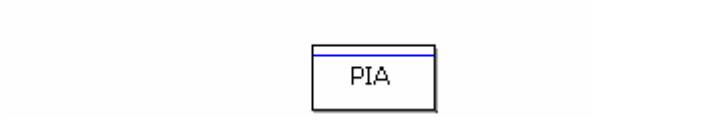
接下來我們需要一個模擬器物件。這是一個物件循環模擬時間並且使得 PIA 在如果被一個真實硬體打開電源時作反應。我想模擬器要持有咖啡製造及機卡片，我將這張卡片放在咖啡製造機左後方。每當 PIA 改變時模擬器會通知咖啡製造機，咖啡製造機可以隨之反應。



使用者介面(GUI)將完成我們的設計。它與模擬器互動顯示咖啡製造機的狀態並接受使用者輸入像「煮咖啡」的按鈕。但團隊並不確認這個設計。模擬器與 PIA 互動並且需要知道咖啡製造機卡片的內涵；咖啡製造機同時也與 PIA 互動。如果讓 PIA 本身是模擬器難道不是更好嗎？這樣縮短許多介面。

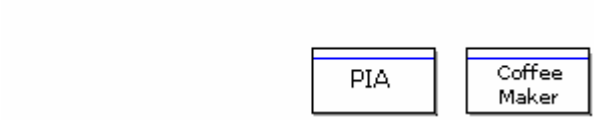


讓我們重新來過。使用 CRC 卡的好處是我們可以隨時清除桌面並且不會浪費太多時間在繪製圖表在每一次設計。

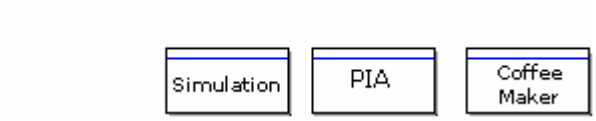


我們回到只有 PIA，這是我們必須有的。接著我們增加咖啡製造機物件，其介面是 PIA。此時我們可以說 PIA 執行模擬，但經過一番討論團隊並不喜歡這

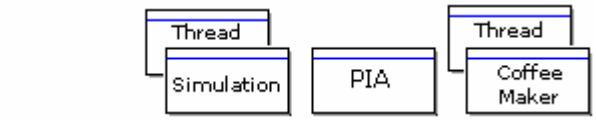
樣。我們希望 PIA 介面簡單化及一般化，比較接近代表硬體。增加模擬的部分無法達到那樣。



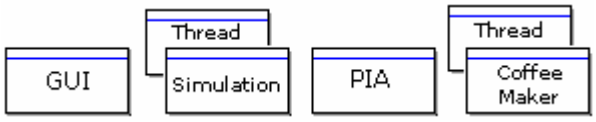
所以我們還是放入模擬器，但模擬器不擁有咖啡製造機，我們只能說模擬器只是 PIA 的介面。模擬器並不了解咖啡製造機工作的內部知識。我們都可以接受這是一個比較好的模擬。



爲了讓這個能夠工作我們需要咖啡製造機及模擬器分別的執行緒(separate threads) 。這個增加複雜的程度，但去除交換部分的兩個複雜度。我們降低循環複雜度(net complexity)因爲我們無須提供一個咖啡製造機介面給模擬器及某些其他的介面於真實硬體執行咖啡製造機。從另一方面 Java Thread 也沒有那麼複雜。同時我們可以正確的測試咖啡製造機當它在真實硬體執行時。我們同意這是比較好的既使它是多執行緒的。



最後我們加入我們的 GUI 到這個模擬的介面。這似乎是一個好的設計可以開始了。記住；我們可以在任何時候發現難以實作時隨時改變我們的想法。我們依賴的是 XP 的強大優勢；所以我們不需要事先設計我們的所有類別的明細。現在我需要想一個系統象徵以符合我的設計。



測試PIA類別(Test the PIA Class Into Existence)

程式設計的第一小段是 PIA 類別。我們知道我們需要模擬硬體同時我們知道看起來需要怎樣。它有三個方法，一個是輸出入，一個讀取，一個寫入 PIA。所以先建立 PIA。

在 XP 我們建立程式碼的方式是以測試開始。

很幸運的我們已建立一個單元測試框架。所以讓它來爲我們首次測試建立一些程式碼。此處我們需要的是能夠讀取 PIA 並且撰寫程式讓輸出位元總是爲 0。輸入位元在讀取時是不受影響。

```
package simulator.r1.unittest;

import unittest.framework.*;
import simulator.r1.*;

public class TestReadFromPIA extends Test
    {public void runTest()
        {PIA.register = 0x0F0F;
        PIA.setInputs(0x00FF);
        should(PIA.read() == 0x000F, "Outputs should always be zero");};}
```

要能夠編譯並執行這個測試我們需要爲 PIA 建立一個根類別(stub class)。我們只是建立方法而不必麻煩的撰寫程式碼。

```
package simulator.r1;

public class PIA
    {public static int register;

    public static int read()
        {return 0x0000;}

    public static void setInputs(int aShort)
        {};}
```

接著我們可以爲我們的第一次測試建立一個 TestSuite。

```
package simulator.r1.unittest;

import unittest.framework.*;

public class SimulatorTests extends TestSuite
    {public SimulatorTests()
        {tests = new Test[1];
        tests[0] = new TestReadFromPIA();};}
```

現在我們可以執行這個測試。當然它沒有通過。但我們希望執行它以便確認。有時這個測試會通過，表示我們的程式碼已經執行我們所需要的或我們的測

試並未測試我們所需要的。讓我們回到 PIA 類別並且撰寫一些程式碼讓測試可以工作。我們加入一些程式碼繼續。

```
package simulator.r2;

class PIA
    {public static int register = 0;
    public static int inputBits = 0;

    public static int read()
        {return register & inputBits;}

    public static void setInputs(int aBitMask)
        {inputBits = aBitMask;};}
```

讓我們嘗試這個單元測試。它執行的結果就如我們所預期的。我們的程式碼仍然不錯並且簡單乾淨所以無須重整，所以讓我們加入第二個測試。我們需要測試寫入 PIA。

線上看執行結果

PIA 類別的第二個測試(A Second Test for the PIA Class)

接這我們可以爲寫入 PIA 登錄建立一個單元測試。在這個案例中我們可以確認只有輸出位元被更改。

```
package simulator.r3.unittest;

import unittest.framework.*;
import simulator.r3.*;

class TestWriteToPIA extends Test
    {public void runTest()
        {PIA.register = 0x00FF;
        PIA.setInputs(0x0F0F);
        PIA.write(0x3333);
        should(PIA.register == 0x303F, "Write never changes inputs");};}
```

我們同時需要將這個測試放入我們的測試序列。

譯者：Areca Chen

```
package simulator.r3.unittest;

import unittest.framework.*;

public class SimulatorTests extends TestSuite
    {public SimulatorTests()
        {tests = new Test[2];
        tests[0] = new TestReadFromPIA();
        tests[1] = new TestWriteToPIA();};}
```

如果我們終止(stub out)我們的寫入方法我們可以編譯並執行我們的新測試以確認它是失敗的。現在讓我們建立寫入方法的程式碼。

```
package simulator.r3;

public class PIA
    {public static int register = 0;
    public static int inputBits = 0;

    public static int read()
        {return register & inputBits;}

    public static void write(int theNewOuputs)
        {register = read() | (theNewOuputs & ~inputBits);}

    public static void setInputs(int aBitMask)
        {inputBits = aBitMask;};}
```

再次執行這個測試。這次如預期的通過了。事情進行的不錯。但我們不確認我們喜歡這些程式碼看起來的方式，讓我們做些重整的工作。

線上看執行結果

字彙表

Iteration：反覆、循環；表示在程式建構過程中依據專案的部分需求所做的分析、設計、實作及測試四個步驟的生命週期稱爲一個循環，整個系統的建構部分是由許多循環組成。

Developers：開發人員；在 XP 中主要爲程式設計師。

test suite：序列測試；所有測試單元的集合。

附錄：

由趙光正先生提供相關名詞翻譯的說明。藍色部分是趙先生的建議，綠色及粉紅色部分是譯者的的說明。

聽起來"終極製程"還蠻不錯的,不過,可能有人會以爲是 Extreme Process

Areca: 我想如果 XP 真的能在國內被接受後，應該會有一個大家共同所能接受的名稱。因爲畢竟 XP 談的主要也是軟體發展的 Process。其實『編程』這個譯名應該是大陸用的其實也不錯；往後兩岸交流頻繁有些名詞會流通的這點也可以考慮翻譯的方式，畢竟電腦這個領域一直在發明新名詞不是嗎？

Extreme Programming = 終極程式設計 or 終極程式 ,例如 Programming Language = 程式語言

最早 XP 我翻譯的就是〔終極程式設計〕；只是後來看到 SOS747 翻成製程，其涵蓋範圍也比較符合 XP 的精神；也就是說 XP 不只是撰寫程式而已還包括系統設計及測試等部分，單純〔程式設計〕。

統一字彙主要是在統一名詞,而且是專有名詞,這樣讀者才不會一頭霧水,

通常會跟上下文(Context)有關係的大多是形容詞

ARECA:一般這種情況我會特別附上原文以供比對；畢竟有時專有名詞也可能有其他的意義，但基本上我還是非常支持統一譯名。所以既使是統一也可能列出各種譯法，這個需要依靠『字彙表』或『譯註』來補救。

雖說統一翻譯字彙是一條捷徑，對翻譯而言幫助非常大，但絕非百分百，既使字典也會把所有得解釋羅列以供比對其前後文才可看出單字實際上的意義。不過使用字彙表對譯者及讀者相信都有幫助，但每篇章仍應對單字特殊的解釋提供說明。

我對 XP 不熟,能否解釋一下雙人組程式設計

ARECA：所謂雙人組程式設計是『雙人組設計的最佳方式是兩人在螢幕前坐在一起，移動鍵盤及滑鼠，一個人打字並思考欲建構關於方法的策略，另一個人思考這個方法配合類別的策略。』這是 XP 驚人之舉。

Pair Programming = 成對式程式設計

這也是最初的譯法；只是後來簡化成〔成對程式設計〕或〔成對設計〕。這對 XP 而言是比較新的做法；至目前爲止也是比較專用的。所以應可統一譯法但需使用譯註說明，所以我必較偏愛〔成對設計〕當作一個 專有名詞因其似乎是比較口語化便於溝通；同時應不致於混淆。不過最後發現使用『雙人組程式設計』是最貼切的。

譯者：Areca Chen

Extreme Programming :A gentle introduction.

第一次翻這個字的時候,我也很想翻成循環,

不過讀者可能會誤以爲是 Cyclic,

而且要當做循環,必須有數個狀態,依序轉換狀態

反覆的意思則是重複

例如有一個狀態機器(State Machine)狀態爲 A, B, C, D

如果執行順序依序爲 A->B->C->D->A->….

那麼 A,B,C,D 就是一個循環

而 A->B*->C->D 中 B 的部分就是反覆

ARECA：沒錯反覆也有幾種狀態，下圖（節錄自 XP 教主 Kent Beck:Embracing Change with Extreme Programming）可以證明。反覆還是可以分成『分析』、『設計』、『開發』及『測試』(下文是訪談 Robert C. Martin,的節錄 Robert C. Martin,是 President of ObjectMentor, Object Guru, Editor of C++ Report and author of 'Designing Object-Oriented C++ Applications using the Booch Method' on his recent endorsement of XP.)。依趙兄的說法 B 的部分是反覆，也很難說 B 細分下去不能在包含幾個狀態。這只是依據我們的『觀點』定位；也就是我們分析的細緻程度而定。所以基本上還是照字面 Iterative 翻成反覆而 cyclic 才翻成循環。

[Mark] Ralph Johnson is quoted as saying (this is hearsay) that the XP process is analysis...test...code...design... would you say this is an accurate description of XP?

[Bob] No. It is, however, an adequate description of one development episode in XP. A development episode might last an hour. An XP project is filled with thousands of little micro iterations. Each of which contains a component of analysis, test, code, design. The ordering is significant. Understanding comes first (i.e. analysis); which is mixed with some design as well. Then we write test cases that describe that understanding. The act of writing tests means that we must also have a design for the code in our minds. Then we write code that passes the test cases; and of course there is an element of design
involved with writing the code. Finally we refactor the code to make it as simple and clean as possible. So there is an aspect of design in all four of the steps.

譯者：Areca Chen

Extreme Programming :A gentle introduction.

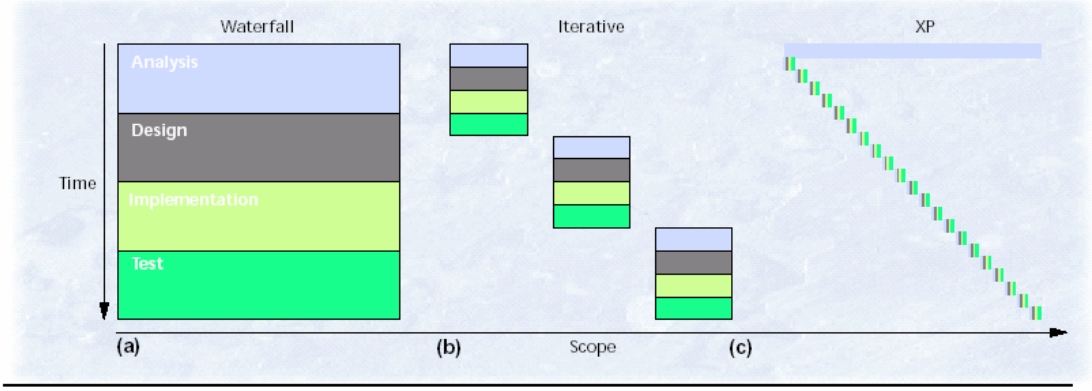


Figure 1. The evolution of the Waterfall Model (a) and its long development cycles (analysis, design, implementation, test) to the shorter, iterative development cycles within, for example, the Spiral Model (b) to Extreme Programming's (c) blending of all these activities, a little at a time, throughout the entire software development process.

Iterative Development = 反覆式開發

Iterative 這個單字譯爲〔反覆〕蠻拗口的，翻成〔循環〕似乎是比較好，但觀之業界好像都翻成〔反覆〕，只好從善。〔反覆式開發〕基本上較〔反覆的開發〕用字較雅且較貼進原文（名詞對名詞），而後者開發有點動詞對味道。我想這是我近來的翻譯風格希望偏向口語化以及以自己的口吻來寫而非照字翻字，所以兩者會有一些差異。

Team Work = 團隊合作

這就比較不像照字翻字應該也是比較好的方式。

Testability 是一種非功能需求(Non-Functional Requirement)

Areca：所謂『非功能性』的定義可否請趙兄指教。本句原文是『Another requirement is testability』，不過『要求(?)』與『需求(request)』是否有所區別？因爲所謂『不明確』本身就很難定義，尤其是在 XP 中所有需求幾乎隨時都可以推翻的，因爲 XP 的目的就是回應需求改變頻繁的環境。

我猜你翻譯成"要求"的英文是 Request,這個字有時代表使用者提出來,比較不明確的需求

所以我可能譯爲:

XP 的另一種需求(Request)是滿足可測試性(Testability)

Testability = 可測試性

這個應看前後文原文是

〔XP 另一個要求是可測試的(testability)〕

若用可測試性應翻成

〔XP 要求的另一個特性是可測試性〕

說法不同用字也不同。

Stability 也是一種非功能需求(Non-Functional Requirement)

所以如果不用固定的翻法,讀者可能混淆,

這句話我會翻成:

譯者：Areca Chen

Extreme Programming :A gentle introduction.

驗收測試不只能夠減少程式臭蟲,它們還能增加程式的穩定性

Areca:這段中有一句『The customers experienced this as a system with a far greater feeling of stability』這句話說明『stability』是人（客戶）的感覺，所以我會翻成『穩定感』就是這個原因，所以我還是強調前後文的重要性。

Stability = 穩定性

原文：也是應參考前後文

〔可接受測試不只是排除臭蟲它們讓我們感到穩定感(Acceptance Tests Don't Just Eliminate Bugs They Add a Feeling of Stability)〕

或許更好的方式是：〔可接受測試不只是排除臭蟲它們讓我們增加穩定感〕

Release = 發行(動詞), 發行版本(名詞)

這個建議不錯。

會把 Functionality 翻成功能"性",主要是想把 Functionality 跟 Function 有所區隔

一般在談論需求(Requirement)的時候,會用而不會用 Function,

所以兩者的區隔是有必要的

Areca：Functionality 跟 Function 有所區隔我非常贊成；只不過 Functionality 翻成『功能性』可能需要斟酌；因爲加上『性』有點像是形容詞（如『功能性需求』變成形容『需求』），而 Functionality 是一個名詞，可能需要創造一個相對應的名詞會比較恰當。Functionality 在字典的翻譯是『官能、機能；功能』，使用機能可能可以區別但是總覺得怪怪的。我想我再找找相關文章瞭解實際上其所指爲何再決定如何翻這個字。也希望大家集思廣益提供意見。

這句,我可能會譯成:

不要太早加入(系統的)功能性(需求).

Functionality = 功能性, 與 Function = 功能 區別

原文：

〔功能不要太早加入(Never Add Functionality Early)〕

對照前後文〔功能性〕可能不是很恰當。

"解構"興起於文學結構之再現，每一主體(subject)面對相同客體(object)時會有他們自己本身的解讀方式。

所以原本是用解構來翻譯 Reconstruction,我想解構應該也蠻適合 Refactoring 這個字

而 Decompose 應該只有分解的意思

Areca：在 XP Refactor 表示再不改變功能並能通過原有的所有測試前提下，修改程式碼使之簡化易懂。所以應包括解構及組合兩個程序。基本上『解構』讓人容易誤會只有分解而沒有組合，所以『重組』可能會比較貼切一些。所以我的意思是 Refactor = (decompose)+(analysis)+(compose)，其中(decompose)+(analysis)表示『解析』。

譯者：Areca Chen

Refactor = 解構

Refactor 這個字在字典找不到，不過一般不只是解構(decompose)還包括組合(compose)，可以說是重組還差不多。有關麼一部份 http://www.refactoring.com 有進一步的說明可供參考。譯者後續將進一步介紹。

Acceptance Test = 驗收測試

這個不錯。

Schedule = 時程

這個也視前後文考慮。

一般生產管理最常用到這個字,所以可以參考他們的翻法

Bill of Material = 物品清單

〔材料清單〕是(BABYLON)字典唯一的翻譯，也是直譯。

Coding = 寫程式

〔撰寫程式〕或〔編寫程式〕應該都可以。

我原本還以爲 Sequential Integration 跟 Sequential Development 有關係

看來 Sequential Integration 跟 Iterative Development 比較類似

前者有一個整合階段,而後者則不斷持續進行整合工作

Areca：我想這是我的誤解，『循序式整合』是正確的。因爲在 XP 中每個人都擁有所有程式碼的修改權，也就是平行式的處理程式碼，但若是每個人修改後就發行容易造成最終發行版本的不一致導致混亂。因此需要有一個機制控制這種情況就是『循序式整合』。也就是說修改可以任意爲之，但整合到最新發行就必須管制以免造成程式碼版本混亂。

Sequential Integration = 循序式整合

這是標題的部分〔連續整合的程式碼(Sequential Integration)〕我只是想讓讀者更了解內容罷了。其實翻成〔循序式整合〕意義上與內文似乎難以了解實際內涵。

Quality Assurance = 品保

翻成〔品質保證〕也沒什麼問題。

Review 這個字在專案管理中常常用到,所以最好有一個固定的名詞

Areca：依據字典的 revier 是『再』檢查，所以『審查』是比較正確的譯法。

Code Review = 程式碼審查

譯者：Areca Chen

〔檢查〕是比較深入而〔審查〕是比較表面；一內文而言似乎應爲〔檢查〕比較貼切。

如果是名詞的話,可以直接套用

殺手級 xxx

例如殺手級軟體

Areca：『xx 級』是形容詞，可能難以意會是一個『軟體名稱』。

Killer = 殺手級

依前後文此應是名詞，故用〔殺手〕。

負載因子是一般的翻法,所以...

Areca：因爲我也沒有特別的依據，我想還是從善如流吧。

Load Factor = 負載因子

〔因子〕與〔因素〕是相同的。

〔負荷〕或〔負載〕應是〔負荷〕比較貼切。因其是以所需天數來計算。

因爲是形容詞,所以陳兄講的沒錯

Typical = 標準的

字典｛典型的,有代表性的，特有的,獨特的;表現特徵的｝

依照前後文應爲〔典型的〕或〔一般性的〕

按照意義翻成

測試單元輯 或 測試單元集合 如何

Areca：因爲我想強調『系列』這一詞。因爲單元測試是每一反覆之前就要實作這個反覆的單元測試，完成後必須執行之前所有『一系列』的單元測試。朱兄曾說『系列』一詞有『順序』的關係意涵，的確 XP 的這些測試是沒有規定順序的（至少目前我還沒看到有此一說），只是我不覺得『系列』是有順序的關係；雖然他是一串的但並不代表有人爲的特定安排順序頂多是依據完成的先後順序罷了。其實趙兄的譯法也非常正確，但是個人體會可能不一致，畢竟這還是人性的問題。翻譯本身就不是完整的創作，但還是保留一點點創作的意味在吧。

Test suite = 測試套餐 or 測試包, 例如我們常見到一些上網包

我的意思是〔test suite：序列測試；所有測試單元的集合。〕意義上應是相同的。

2001/8/16 譯稿

2001/8/22 校稿

於高雄永安 ARECA CHEN