物件開發程序系列...XP 頭對頭

XP:抄近路的詭計

Doug Rosenberg 及 Kendall Scott 對於 XP 的質疑

來源：  We Know the Object ...

Pdf 版本(793KB)

Zipped 版本 (599KB)

介紹

終極製程(Extreme Programming (XP))提供一些有用的概念。我們相信強調測試及他們的指引原理是有重大的價值；這些原理在軟體開發程序的發展上非常有用。但是不管如何；曖昧的主張(dubious claims)、明顯的缺乏可達成性(scalability)、動聽但是問題重重的口號及極端的(Xtremist)姿態幾乎掩蓋了良好的本質。

Doug 曾經在 OTUG(Object Technology User Group)中有過廣泛的討論。其中有些是相當有用的，有些則是愚蠢的，有些仍有問題亟待回答。Kendall 花了相當長的時間詳讀 Wiki 網站上的文章。下面大多數的探討 XP 的結構及程序(ritual)是依據這些交談及研究。

曖昧的主張

Kent Beck 及其主要的追隨者曾經定位 XP 是一種『輕量級(lightweight)』程序；這個開發程序可以產出較高品質；以及在一個更及時性的基礎上；然後由更細節的程序所組成。（XP 人(XPers)偏向歸因這些所使用的術語如大設計(Big Design)<譯註 1>，我們在這整篇文章中也使用這個術語。）然而在這個『程序』的核心有一個相當大的漏洞：分析基本上以被拋出(tossed)。

讓我們看看一些主張；XP 追隨者把跳過大設計的分析中『笨重(cumbersome)』的成分做了合理化。

每週改變的需求

依據 XP 人的說法，處理那些變動的需求在需求的前置(upfront)分析至少有兩個主要的缺點：

- 大設計尋求控制改變的機制，經由早期獲得及一般性的一次解決所有的需求及架構。實際上多數的大設計方法有詳細的結構承擔必然發生的外部變動，但這些機制往往似乎是或多或少在程序中屬於次要的。
- 大設計視任何不確定情況是一種危機。如果有一些事情是我們不知道的，我們無法推理它們。

這些由 Doug 所接收到記錄的言詞，經與 OTUG 聯繫，其答覆是：「你說前置(up-front)『 擷取需求(capture requirements)』，但坦白的說真的嚇到我了，因爲我從幾哩外聞到『瀑布式』。」

甚至，XP 陣營的感覺是客戶對於其在專案中相要的只有一點點的概念。有一個信徒甚至宣稱「一開始客戶不可能瞭解他們真正所想要的。」甚至當客戶當初指出他們的需求，他們在幾週之後堅持改變這些需求，而有時是每天的改變。因此，在跳入程式撰寫之前嘗試凍結需求是沒有意義的。

我們同意需求分析及擷取是充滿困難及不確定性。但 Doug 曾經在許多業態中做過多個專案，其中他的客戶或多或少控制避免如此的優柔寡斷。不，在你開始構建你的系統之前你無需擷取所有個別的需求，但我們堅持某層次的前置分析可以解省你一路上許多時間。

有時在開始撰寫程式碼之前要求客戶精鍊(refine)他們所瞭解的並指出他們想要的是有用的，客戶希望從開發團對中獲得一些訓練(discipline)；而同時要求客戶提出一些訓練也是全然合理的。如果客戶真的對他們所想要的沒有任何概念；當他們看到一些可執行的東西；雛形（以實際的程式碼建立的）是一個很好的技術可以幫助客戶獲得清晰的瞭解他們所想要的。那是與客戶合作的一種良好分析的工作同時可以幫助客戶精鍊他們所瞭解的。

確實有這種情況；當開始撰寫程式之後客戶的需求改變了。我們假涉有任何的可能性，這些改變在下一構建循環中獲得控制。若那是不可能的時候，你應當盡你最大的能力去做。許多 XP 技術在「盡你最大的能力去做」大概是相當的有價值的。

有人問 Daug「如果你後來發現客戶的需求已被曲解而修正需要從設計的基礎改變你會怎麼辦？」真正的問題是，這種情況發生的頻率有多高，而我們需要多辛苦事先避免產生這種情況？而這正是嘲笑『分析』程序爲其本身付出太多時間。我們作分析以避免這種情況的產生。

使用案例(use case)太複雜

XP 建議你以『使用者故事』擷取需求。這是相對使用案例；XP 人一般的感覺是使用案例只能在詳盡的細節架構使用者的需求，在長長的格式中有太繁重的東西要去填入。引用 Wiki 網頁誠實的傳達 XP 群體對於使用案例感覺：「我的目標是維持介於企業(business)與開發之間策略權力(political power)的平衡。我曾看過的使用案例的應用太過複雜且正式(formal)以致於商業一方不想去接觸使用案例。這導致開發一方詢問所有的問題並寫下來所有的回答並且承擔所有結果的責任。商業一方簡化成只坐在桌子的另一邊並指指點點的。」

這是我們看這件事的方式。

一個使用案例是一系列的活動；其中一個活動者(actor)在一個系統中執行以達成特定的目標。關於其定義並沒有特別的複雜，或太過正式。

使用案例是最有效說明使用者的觀點；其中是以動詞片語表達活動。這表示使用者需要有一個清晰的瞭解這一系列的活動是他期望依循的，那確實意味著客戶是主動的參與使用案例的細節。如果所瞭解的不是這些，那麼使用案例是太複雜需要精鍊--你知道，就像程式碼迫切需要重整(refactoring)。

在我們的經驗中，『分析癱瘓(paralysis)』在關連到使用案例塑模(modeling)至少可能有三種發生的理由：

- 團隊花費幾週的時間構建精心製作精緻的使用案例模式而無法據以設計。
- 團隊空轉於憂心是否使用任何或所有的 UML 構詞(constructs)『include』、『extends』及/或『uses』。
- 團隊浪費時間在長及複雜的使用案例樣版（有趣的是，我們似乎或多或少同意 XP 人這方面的觀點。）。

XP 信徒顯然指熟悉導致分析癱瘓的一類使用案例。我們說如果你以使用案例連接(in conjunction with)頻繁的雛形法推導(derive)，並且/或從舊有(legacy)文件（特別的，使用者手冊）挖掘使用案例，而你仍高度全時間聚焦於你的客戶，你非常有機會在你開始做之前指出你所需要做的，同時結果將是顯著的一路安全的走下去。

修復的成本曲線是平緩的

Kent Beck 曾經宣稱改變程式碼中的臭蟲的成本與改變設計臭蟲的成本一樣的，因爲程式碼是設計。我們在後面會說明這個理論的基礎上的愚蠢，但同時我想要指出由於 Xtremists 從圖形吹噓其分析，那是 導因爲果(circular reasoning)。

其推論如下。如果我們在撰寫程式之前並未作分析及設計，那麼沒人可以斷言在我們開始撰寫程式設計之前修復錯誤是比較便宜的--因爲沒有「在我們開始撰寫程式之前」這件事情。迅速的，修復的成本曲線從對數變(logarithmic)成線性(linear)。

讓我們稍微重新敘述：

我們可以是軟體開發爲一持續系列的撰寫程式、測試程式碼、拆解(ripping)程式碼、重新測試程式碼、重新撰寫程式碼、重新測試程式碼如此不斷的循環。 這些都是相同的動作，加上一些分析及設計交雜持續進行。因此，程式碼變動的成本是線性的與你何時發現需要改變沒有關係。這類的導因爲果的聲音對我們而言就像...嗯，就像有人在推銷一些東西。

直接的比較這些循環論法(circular arguments)是我們的經驗；我們的經驗中有許多錯誤癱瘓專案的方式是「我認爲那是紅色的（或可能是津貼），而你認爲是藍的（或可能是扣項）。」此種不明確說明的假設進入程式碼並且被證明其本身是潛伏的臭蟲。當這些不明確的假設進入架構本身，我們主要看到的是詛咒並且重寫（及重測試）的動作。

我們並不相信任何宣稱任何事可以使得這項工作免費的。拆解並重寫絕不可能是免費的。如果拆解並重寫是『常態』的，也不可能是免費或便宜的。可能使得曲線異乎尋常的平坦，但仍然不視線性的。我仍未看到任何證明 Barry Boehm 所做的研究是虛假的，他的研究顯示當你從分析移動到發行產品之間修正一個錯誤的成本是以指數遞增。重複宣稱成本曲線是不可思議的平緩並不能成爲事實。

明顯缺乏可達成性(Scalability)

讓我們看看爲什麼 XP 相關一個專案時似乎不像可以達成並且成功；其中這些條件似乎尚未出現。

Pair Programming

在理想的 XP 專案當中，開發者是成對的工作，而且也不是靜態的成對：依據哪些事要做而誰知到怎麼做，人們可能變換伙伴一天好幾次。這個想法；當然；是立基於格言「兩個頭腦比一個好(two heads are better than one)」。就其本身而論，嘗試去遵守這個原理是非常恐怖的。

但當你無法在一段有意義的時間內讓你所有的開發者同在一個地方；在一個『牛欄』內（假設有一個可用的地方）；會發生甚麼事？事實上世上有許多開發者並不是真正喜歡這類的堅強的溝通及社交技能；而這些在 XP 架構下是很明確需要的；面對這個事實你要怎麼辦？當每一個人爲每一件事情爭功時，你如何掌控任何人的應負的責任？

如我們在這本書所描述的，最好的方式包含在高階及初階人員之間實作一組檢驗及平衡，例如；我們建議比較沒有經驗的人審查系列的圖形；這些開發者使用重要的 OO 專家產品。雙人組程式設計顯示出這個想法到一定程度，但雙人組程式設計同時假設一對一配對總是可行的；而這個簡單的說在許多背景中並不是實際可行的。

嗯--可能如果那是稱爲雙人組分析、設計及程式設計，我們將會比較滿意...。

關於雙人組程式設計並沒有任何事是天生邪惡的--還是可能獲得一些利益--**但雙人組程式設計對我們而言看起來是意圖補償分析的缺口；因此強迫兩個人檢視他們所撰寫的每一行程式碼**。這類似強迫幼稚園小朋友走向餐廳時兩兩牽手因爲你無法信任他們不會毫無目的的隨意漫遊。

以另一個方式來看：我們兩個一對下棋天生的每一盤棋都一定輸給 Garry Kasparov，既不是伙伴系統也不是『詳盡的測試』可以補償前置規劃的缺口。畢竟，Garry 非常小心的規劃他的策略。

索引卡

XP 一族倡議使用類別-責任-合作卡(Class-Responsibility-Collaboration (CRC))。目前；這些卡被重視並廣泛的使用--但只限於相當小的規模。CRC 卡的設計確實不是用來擷取特別的細節，而像我們做研究論文在我們發掘內在的知識之前使用都是不錯的，它們是保持記錄的佼佼者。而你如何期望在有一段距離在開發者之間傳遞索引卡？

我們客戶中之一有位主任建築師花了許多時間與其他的程式設計師工作。事實上，他花了太多的時間以致於他沒有足夠的時間給他的新材料他需要做的，這使得他相當願意參與獲得一些他的設計文件。很不幸的，這家公司的開發者位於兩岸，因此整個「每一個人在一個大房間中共享索引卡」的想法是有點距離的。使用 E-mail 傳遞 模組對他們而言將是較好的方式。同時，我們相信客戶在下一版本的產品中使用塑模做更好的工作會是相當成功的， 同時透過提供更視覺化於架構及設計他們將能夠明顯的降低他們的延遲次數。

我們看過非常少的開發成果而沒有一些頗爲嚴肅的議題需要除裡。尤其是快速成長的公司；其中新專案的大小較之剛開始時更是快速的成長。我們建議使用一個好的視覺化塑模工具把開發的工作放在顯著強壯的基石上而不是做一串蘇散四處漂浮的索引卡。

缺乏責任性

XP 擁戶者堅持大設計趨於只是讓人們；理論上；有特別的知識及技巧以執行確定的任務（你知道；這個團隊必須包括類別庫及一個架構設計師及一個技術撰寫者等等）。於此，XP 說「每一個人做每一件事，」以直接回應對於『大設計佔便宜的事實是有些個別的人比別人較有資格做某些設計決策』的認知。但現在我們會相信開發者將都會立即變的熱衷於必須真的以讓新人可以獨立瞭解的方式寫下東西嗎？（同時；不管如何；有好的設計者執行設計有甚麼錯嗎？）

事實上，如我們將在在本文後面探討的，XP 擁護者宣稱適當的重整(refactoring)程式碼是或多或少自我文件化(self-documenting)，而所有其他的文件形式太不可信賴而值得存在。這是他們對於『口頭的傳統(oral tradition)』太過於聚焦的部分合理化，同時他們認爲保持東西是如此重度的依賴讓所有的關鍵資訊被維護在同一個地方。畢竟；程式碼外部及所有這些索引卡；並沒有變成任何文件上的東西，對不對？但那只是沒有務實的思考這是將爲任何專案工作；而有更多的開發者聚在一個房間之內。不錯；溝通管道的數量在人們加入專案時以指數方式增加，同時；我們在本書中倡議一個相當簡易(minimalist)的方式，但 XP 的方式只是太過簡易而無法在所有的專案中可用。

一個 XP 替代方案：『聰明製程(Smart Programmers(SP))』

一般而言，多數我們的客戶對於這種沒有大小限制(scale)的解決方式沒有太多的興趣。不管如何；如果小規模(small-scale)解決方式是你所想要的，這裡有一個 Doug 曾經經歷過的；這是本文中的一個目的我們稱之爲『SP』。

如果你是靠自己的力量經營一家沒有太多可冒險的資本的公司 SP 是非常有用的。它的基礎理論是所有程式設計師中 5%（或更少）做 95%（或更多）的工作，而且只有只有這些（5%）程式設計師值得聘請。

SP 的本質如下：

- 只需聘請前面 5%執行的人是你個別認識的，或你信任的前 5%的人員。
- 讓一個主任架構師(chief architect)非常小心的規劃軟體工作單元架構；如此單元（即 UML 包裹）之間有極小的耦合性(coupling)。
- 讓主任架構師比對 SP 成員的特殊專長及經驗。
- 讓所有的 SP 程式設計師在家工作，在家中他們不會受干擾，除非他們明確的要求在一個共通的工廠中工作。
- 限制狀況報告爲半頁，並透過 E-mail 每週傳送一次。
- 讓你的 SP 成員當他們認爲他們已完成時導入漸進式建立測試。
- 限制(restrict)SP 開發者在需要的時候只能透過 E-mail 或電話溝通。**絕不要**把所有的程式設計師同時帶進一個房間內；除了在整合測試時，以及只有在他們的程式碼是完全的含入。

現在，你可以以這種方式建立非常大量的程式而只使用非常小的團隊。Doug 的經驗指出生產力水準更高過於 XP 所舉出的克萊斯勒薪資專案。

此外；生產力將比 XP 所產生的更高。不管如何，嘗試讓主任架構師跳過分析及前置規劃，或做了一個壞的聘請的決定，你會在你手中產生一個災難。我們絕不會麻煩的使之形式化或甚至命名，SP 技術因爲其沒有大小限制及沒有錯誤的限度。就我們來看像 XP 可能也如此裝腔作勢。

<譯註 2>

響亮但問題重重的口號

如果你想要的是一個立即讓你難忘強而有力的口號，XP 正是指引你一條路。畢竟，就如 XP 的訓練現在已經很清楚，「軟體也是...很難花費時間在沒有意義的事情上，」這類的主題很容易變成一種口號。讓我們看看一些 XP 人呼喊整體重新整合相關的關鍵慣用語(phrase)。

只有四件關於軟體重要的事情

這是最大的一個，XP 存在的理由。這四件事是甚麼？撰寫程式、測試、傾聽及設計（從前的重整(refactoring)）。

喔，同時別忘掉：「這裡所有的都是關於軟體，任何人告訴你的與此不同都只是在推銷東西。」

就如我們之前指出的，『分析』並沒有使得時間縮減。（這可能是 Beck 在 OTUG 有些事情與某件事情：「我以真正的名稱稱呼分析：說謊。」）更重要的，既使，博學多聞的人知道關於軟體恰巧有七件重要的事。（我們不想在此分享，但在我們的書中可以找到，如果先前你不是很有知識）

儘管；嚴肅的如此熱烈的宣稱軟體開發只有四種元素是有意義的；是極端的超越現實，甚至還有些事情稱之爲 XP。

原始程式碼就是設計

我們承認從這個得到相當大的刺激(kick)，這不是最不重要的因爲我們曾寫個一個笑話「使用案例(use case)不是需求」；而在我們的書中有類似的。在 Wiki 的首頁上中的評論像「當你重整，你是在進行細節的設計」，這就像我們將在另一本書中做的那至少有一個章節說明針對駁斥這類的廢話。

重整是一種極爲迷人的主題，我們希望在 Martin Fowler 的書中證實。我們確認那將提供我們許多有用的技術以改善我們的程式碼。但設計是設計，而撰寫程式是撰寫程式，把他們視爲相同的一件事是愚蠢的。UML 之神（好吧，三個其中之二，不管如何）基本上忽略初步的設計是很不好的，現在我們必須處理 Xtremist 狂熱的扭曲細節設計的概念而超越所有的認知。

每次我們看到爭論我們喜歡總結如「改變程式碼是那麼的便宜；不管我們是否從設計做起。」我們都會感到胃痛。這正是我們要嘗試指出是甚麼導致的種感覺及去嘮叨這件事。

在 XP，『甚麼(what)』描述是從使用者故事擷取。我們關心的是專案團隊如何處理跨越『甚麼』及『如何(how)』之間的差距。可見的理由之一是在 XP 中漸進式發展(increments)是如此的微小，對每一個漸進式發展階段，介於『甚麼』及『如何』之間的差距不僅必須跨越設計層次；那是很難滿足的；同時包括程式碼層次。這表示設計錯誤及程式碼錯誤將被混雜(intermixed)。

不可避免的時間將花費在修復程式碼錯誤以便讓程式可以執行及測試，而測試將暴露設計錯誤。設計一經改變，所有花費在修復程式碼錯誤及測試程式修復部分的精神及時間都將浪費，因爲程式碼現在要被詛咒(ripped out)並取代。在新的程式碼當中也將有新的程式碼錯誤，而前面的步驟還會再重複。這看起來還是 OK，因爲程式設計師在程式設計中『獲得樂趣』。

使用哪種方式並不是說無法達成--但對我們而言，哪似乎要完成工作需要浪費許多精力及時間。

如果我們沒有嘗試盡我們的可能做最好的事，**所有的**時間（而這應包括我們撰寫程式碼**之前**），我們都是馬馬虎虎。如果我們認爲馬馬虎虎的工作是 OK，只要我們快速的工作（「喔，沒關係，我們隨後可以修改」），我們得到的是一堆垃圾。軟體不是與其它的努力(endeavors)在這個觀點上有不可思議的差異，不管你有多好的重整瀏覽器。

在你開始撰寫程式碼之前只要證實設計是正確的，你對於專案審查會有最佳的概念，最好的是在初步層次(preliminary level)（以便確保沒有人朝向錯誤的方向快速前進）及細節層次。然後你才開始撰寫程式並曾測試獲得回饋。

做可能達成的最簡單的事情

這個口號說你應該簡化、簡化、簡化。 避免傾向與把你不真正需要的材料構築入你的程式總是一件好事。但是，我們關心的是那很可能是盡力鼓勵開發者把 Scotch 影帶、泡泡糖、橡皮筋雜七雜八的東西隨意放置在程式碼中；這種程式是由爲充分小心的花時間定義一個堅固的架構所逐漸形成的。

我們最近有一個經驗；其中我們經由反轉工程(reverse engineering)的工具吸收一些『已完成(as-built)』的 Java 程式碼並且發現有些方法其名稱向 bugFix37。很不幸的，任何創造這個方法的人是做可以達成的最簡單的事，那只是相當字面上的意義。（我們只能從它的搭擋程式設計師聽到迴響，歡興鼓舞的說「嘿，我們正期待做可以達成的最簡單的事，不是嗎？」）按照紀錄，這個有問題的專案不是以 XP 所建立的，但那是以沒有正式的分析及設計所構築的。

XP 其它的指導原理並沒有幫助如：如果你認爲你可能並不需要的，你並不需要。這類事情有助於鼓勵這類與世隔絕(insular)的相法；這種想法導致軟體並沒有解決任何位於外界人認爲值得解決的問題。我們引用 Ron Jeffries 的話來加強這個概念：「一個開發者真正需要開始認真工作是直到她已開發她自己的內在品質的意識(sense)，我不期望我的符合你的；我真正希望我們與他們有足夠的接觸；我們可以與他們討論並使用我們不同的意識選擇一個解決今日滿足我們兩者內在的尺寸的問題。」悲哀的，在 XP 除了程式碼每件事都是可視的，我們剩餘的期望只是品質保證。

我們還認爲一個系統的品質最好是經由耦合性（愈疏鬆愈好）、內聚力（愈緊密愈好）及完整性的衡量來反應，加上思考像每一類別加權(weighted)方法及子類別投入的數量作爲好的衡量。參閱我們的書第 5 及 8 章關於至方面的作法(metrics)的更進一步資訊。

Xtremist 故做姿態

從另一方面而言，XP 人喜歡廣泛的爭辯關於是否『勇氣(courage)』或『進取心(aggressiveness)』是比較正確的說明就處理這些麻煩(pesky)的付錢的人而言他們所處的位置。

從另一方面而言，XP 人傾向迷失在新鮮的空氣及說一些事如：「XP 大師(Master)瞭解的如此深刻以致於他們無法被瞭解」

XP 的花言巧語讓我們覺得那是結合相當高程度的傲慢並且試圖模糊使得變得很奇怪。

準備、射擊、瞄準

就如統一開發程序(Unified Process)描述的四個階段（反覆(iteration)、精細製作(elaboration)、建構(construction)及轉換(transition)）；這些在在專案的每一個反覆中執行，XP 說的是「故事(stories)、測試(testing)、撰寫程式碼(coding)及設計(design)」。強烈的與大設計比較；統一開發程序符號表示法(symbolizes)，XP 方法論提供『微量漸進式發展(nanoincrement)』，它的基礎是不超過一週的漸進式開發，甚至是短到一天。更快、更快、更快！

我們不反對找到軟體開發程序更快速的方法--只要這種方式產生的結果不是「準備、射擊、瞄準」。<譯註 3>

由於拒絕花任何一些有意義的時間指出在開始『生產』之前應做甚麼，並且堅持在一個進行中的基礎下『以小的方式(in the small)』做大的工作(great work)；從某種角度當煙幕消散時將神奇的產生整個系統被矯正，XP 人似乎認爲他們已找到銀色子彈(silver bullet)<譯註 4>以反擊大設計的假設：偏向於在細節部分陷入泥沼。

這是一個很明顯的短視(short-sighted)方式。我們可以以分析及設計到一個非常可接受的層次降低失敗的風險而無須把身體連同洗澡水拋棄。早期進入程式碼撰寫並不代表沒有風險；這個風險只是不同的本質（在某些情況可能更難以捉摸）。我們應該嘗試最小化專案整體的風險層次，而不是部分的風險。

當 Doug 七年前開始教授 OOAD，最通常的失敗點是在 訓練研討會(training workshop)；總是發生在從需求層次（使用案例）系統行爲觀點移動到一個細節設計層次（循序圖/合作圖）行爲觀點所做的努力。 嘗試跳過這個階段--在這本書中我們稱之爲跳過『甚麼-如何(what-how)』的差距--從一個純需求(pure requirements)觀點到一個細節設計觀點是異常的困難。人們總是發現的是經由增加動態模式（堅實的分析(robustness analysis)）的『初始設計(preliminary design)』觀點，他們持續能夠讓團隊通過這個轉換點。

我們仍然能夠平衡漸進式開發的利益（可能使用一些較大的漸進式程序），嚴格的測試及其它可能被證明有用的技術，如雙人組程式設計。以這種方式，我們保證我們不止將會總是擁有一個執行的方案(functioning program)，同時我們可以讓最好的開始是『一開始就朝正確的方向』，同時減少對以這種勞力密集(labor-intensive)方式重做及重測試的需要。

文件是無用的

一個真正的 Xtremist 相信程式碼不只是設計同時是文件。『無情的重整(Merciless refactoring)』（另一個 XP 用語）將一直一直一直產生完全乾淨的程式碼；既使是最沒有經驗的開發者也能夠達成。一個信徒杜撰了一個名詞『極端的清澈(Extreme Clarity)』，其中他定義爲「一個程式的屬性就像程式碼展示每一個人

需要知道的系統所有東西；立即上手；一目了然；非常的準確(accurate to six places)」

我們沒有強辯這不是一個我們想要的結果。而就非程式碼(non-code)文件而言，維持與想像中的快速撰寫程式碼機制同步的可能性不是真的嗎？而口頭的溝通比在多數專案中寫下來的作用好不是真的嗎？總而言之，「所有的文件要被懷疑是過期及主觀的偏見」這些聲音對我們而言就像一個無理的對文件的恐懼。

事實上，在一個開發成果的背景，夠格的技術撰寫者知道一些相關程式碼的東西將通常能夠趕上開發者及分析者並且維持到目前爲止。同時，有效的口頭溝通是關鍵性的重點，但不能免除把資料寫下來--爲外部的人、爲新人、爲未來的參考者、爲某些目前可能不是很明顯的原因。就主觀的偏見而言，嗯，所有的寫下的都有某種的偏見，但軟體文件設想是傳達是甚麼(what is)，而不是可能是甚麼(what could be)，同時如果那不如此做，不是媒體的錯誤。

除了口頭溝通任何事物的缺乏表示撰寫程式碼的人必須回答問題。

依據歷史觀點，在近半世紀被無數次用作爲『工作保障(job security)』的機制。在太多的案例中，這些人把文件放在他們的腦袋中（或者是團對把文件放在他們集體的腦袋中）在維護程式設計師出現時已是過去式了。

這裡有一個 Doug 的故事

「我有一個程式設計師爲我工作，負責我們的資料流圖編輯。他的進度常常延遲。我們開會討論這種情況。他說--我引用他的話而我記得非常清楚：『你不能讓我更快，你不能讓別人來幫助我，你不可以解雇我』他很有信心的說，因爲所有的設計文件都在它的腦袋中。我非常震驚於他的說法，第二天我解雇了他。這是我做的決定中最好的一個。我以一個人取代他，這個人所寫的程式碼非常好，而且隨時把他所做的記入文件中。」

不管如何，爲後人記錄圖表並**不是**十分重要的事。**檢視**(reviewing)圖表--並據以設計--在撰寫程式碼之前**是**十分重要的事。如果你能讓團隊中的高級程式設計師檢視（並修正）系列的圖表中將要撰寫程式碼的情節(scenarios)，同時你在撰寫程式碼之前已查核(verify)架構（靜態模式）可以支援所有的情節，你將有一個比較輕鬆的撰寫程式及測試的時光，同時你將使得後續要做的拆開程式碼、重新撰寫程式碼及重新測試的工作降至最低。

而這裡有從 Beck 所說的：「多數的人表達他們的恐懼；就是把 CRC 卡轉換成資料庫（Notes 或 Access）、Excel，或某些該死的專案管理程式。」因此我們現在看到 XP 人把客戶視爲不確定（考慮他們的需求時）並且恐懼（就持續追蹤

將要發生的事）。我們期待 XP 信徒(XPites)證實他們的『懷疑』如此 XP 可以假設擁有傳說中的 FUD(恐懼(Fear),不確定(Uncertainty),懷疑(Doubt))三位一體，如以往對微軟的觀感一般。

詢問程式碼

我們很高興的承認程式設計還是有某一程度的『藝術』的成分加上『科學』的元素這個觀點；這個觀點是我們更想要聚焦的。但有些 Xtremists 的說法傾向於讓像我們這類非信仰的人覺得 XP 或多或少有個人崇拜的現象。舉例來說：

- 「重組系統結構(Restructure the system)的基礎是系統告訴你的系統想要被如何結構化。」

- 「程式碼要簡化。」
- 「系統駕馭(ridding)我更甚於我駕馭系統。」
- 「程式碼**異味(smell)**是你程式碼當中某部分變糟的暗示(hint)。使用這個異味追蹤問題。」
- 「詢問程式碼。」

這個概念；很顯然；是只要程式碼實現，便假設有一個神秘的溝通力量，而建立者義不容辭的傾聽；小心的聽；這個聲音...你會非常想睡...啊，抱歉。

這些說法其根本是就 Kent Beck 而言所有的內容就是程式碼，整個程式碼，除了程式碼以外沒有別的。設計代表『幻想(visions)』，分析代表『幻想的幻想(visions of visions)』，而方法論就是整體代表『幻想的幻想的幻想(visions of visions of visions)』 ，以及「當...幻想成真只會惡化問題』。只要開發者的思考是簡化的--另一種說法--只要他們能夠只寫他們的程式碼而無須所有愚笨的前置材料--他們將「沒有壓力的工作、沒有恐懼的工作並喜愛他們的任務及他們自己。」

有人要來一杯 Kool-Aid 嗎？

結論

XP 人喜歡抱怨是如何的沒有其他的事。例如，當 Doug 在 OTUG 開始指出前置分析是多麼好的事情，他得到不只一封的電子郵件詢問「你寫多少的程式碼，無論如何？」這意涵他寫的程式不夠多；在 Xtremists 的眼中不是夠格的『真正程式設計師』； 擁有像水一樣多的概念卻說他不懂得 UML 的『擴充(extends)』建構式(construct)（我們的書的讀者，忠實的 OTUG 人，已經到這個胡說八道的故事），但這不是重點。重點是 XP 的擁護者有一種齷齪的傾向羞辱我們是比較喜歡不終極(less extreme)；但仍然高度的可運轉(workable)；盡量建立有品

質的軟體--而那是太差的，因爲有一些好的要素(stuff)在其中。

可能有一天我們將看到一個更合理及平衡的開發程序形式。於此之際，我們同意一位在 Wiki 的仁兄所言「XP 最引人注意的特性是它嘗試駕馭(ride)；而不是壓制(quelling)在任何真正的程式設計師內心中深沈的要求，那就是要求開闢一條大道(urge to hack)。

於此之際，我想要以一個問題結束這篇文章。

假如你決定在你的專案中使用 XP，而在六個月之後，你覺得這種方式並未符合你的期望。而沒有寫下任何東西--沒有架構或設計文件及在開發者腦袋中沒有專案的任何知識。**你會怎麼辦？**

Doug Rosenburg：dougr@iconixsw.com

Kendall Scott：onSoftDocWiz@aol.com

<譯註 1>大設計亦即前置設計(upfont) ，相對於 XP 的其他開發程序；這些其他的開發程序在開始撰寫程式碼之前將所有專案架構做完整的設計被 XP 追隨者稱之爲大設計或前置設計。

<譯註 2>SP 應是作者故意創造用以反諷 XP 的一個名詞。

<譯註 3>對於這點 Kent Beck 及 Martin Fowler 在『Planning eXtreme Programming』一書中第三章有所說明。

<譯註 4>銀色子彈亦即中文的萬靈丹 。

翻譯 Areca Chen