RUP® 和 XP 的比較

John Smith

Rational Software 白皮書

TP 167, 5/01

Rational®
the software development company

目錄

簡介

本白皮書比較 Rational Unified Process® (RUP) 與 Extreme Programming (XP)，前者是一種流程架構，由 Rational® Software 公司歷經數年精雕細琢而成，目前廣泛使用於各種大小規模的軟體專案，後者是一種軟體開發方法，越來越多人認爲它是在變更需求的環境中建置小型系統的有效方法。

此比較是以各自的本質和樣式引導來作爲流程說明，及其基礎結構、假設和呈現方式。本白皮書也檢查每一個的可能目標及其使用結果。

大部分流程有一些共同元素，以便進行對稱式比較。它們需要由角色執行的作業序列或群組（通常由個人或團隊扮演角色），來產生構件或工作產品，其中一部分或全部會交付到客戶手中。大部分流程也確認流程實例會有一個含開始和結束的時間維度，及一個代表重要活動（或活動叢集）完成的中間里程碑，還有相關聯構件的製作。於是，本白皮書探索下列流程維度並使用它們來進行比較：

- **時間和人力配置** — 討論如何隨時間安排每一個流程及如何比較人員配置。
- **構件** — 比較工作產品；在以 XP 和 RUP 爲基礎的專案過程中產生的那些*東西*。
- **作業** — 討論每一個流程說明應產生其構件的方式。
- **規範** — 比較 XP 和 RUP 描述軟體工程之重要區域的方式。
- **角色** — 探索 RUP 中的角色（執行作業）和在 XP 中較接近團隊內位置的角色之間的差異。

本白皮書的動機是要闡明 RUP 和 XP 在流程範圍中的相對位置，以瞭解彼此如何*互惠*，及澄清 XP 是 *重量級* RUP 的 *輕量級*替代方案的看法。

撰寫本白皮書所用的 XP 資訊的主要來源是 Addison-Wesley XP Series 中的三本書：

- Extreme Programming Explained [Beck00]
- Extreme Programming Installed [Jeffries01]
- Planning Extreme Programming [Beck01]

這些書獲選的原因，是因爲它們是 XP 資訊最明顯的儲存庫，許多感興趣的讀者或可能的 XP 使用者都是以它們作爲入門書。有規劃其他書，但在撰寫本白皮書時它們還未上市。RUP 資訊的來源是 Rational Software Corporation 的 RUP 產品本身。

本白皮書不是要作爲 XP 或 RUP 的教學指導，因此，對於這些方法已稍有瞭解或可以取得某些參考資料的那些人閱讀起來會比較輕鬆；例如，以 XP 爲例是 [Beck00]，以 RUP 爲例是 [Kruchten00]。

時間和人力配置

RUP 處理專案（來開發軟體）、其有效期限分割成*初始階段*、*詳述*、*建構*和*轉換*等階段。每一個階段進一步分割成反覆，每一個反覆需要數個建置。

對於 RUP 的大部分專案而言，反覆的持續時間是介於 2 週到 6 個月之間[1]，而專案生命期限的反覆數將介於 3 和 9 之間。因此，在這些預設值中，RUP 涵蓋的專案範圍是在 6 週到 54 個月的這段持續時間。

XP 的反覆大約爲 2 週的持續時間。XP 的版本爲 2 個月（或更久一點）；它們是由客戶定義後發行給客戶。在持續時間內，XP 的版本類似 RUP 的反覆 — 至少是在詳述或建構期間[2] ，針對適合 XP 的專案；亦即，針對最大團隊規模爲 10 人（假設）的那些專案[3]，以建立的架構基準爲基礎者。[4]

爲了說明這一點，我們假設 XP 最大團隊規模是 10 人，去發掘適合此團隊的最大規模專案，然後看看 RUP 如何處理不超過此規模的專案。如果我們使用 COCOMO II[5] 作爲預估模型，它預測 10 人團隊通常與 15 個月的持續時間上限的專案是相關聯的。持續超過 15 個月的專案通常會運用 10 人以上的團隊。如果您已準備要提供更久的排程給他們，則您可以建置只有 10 人的較大型系統，不過，排程是很重要的，如果您可以有效率地執行，那麼您可以規劃新增更多人員。此範例專案可從頭產生大約 40,000 到 50,000 程式碼行 (slocs)（800-1000 功能點[6] 以 Java）撰寫。

根據模型，這是您要迅速完成的具有 XP 規模團隊的最大型專案。此外，或許還有其他原因可解釋爲何小團隊無法有效率地建置大型系統；例如，經過一段時間之後對於已建置的程式碼不再熟悉。（大型團隊可並列進行的工作，小型團隊必須逐一進行。因此，到了整合測試和除錯的時候，小型團隊必須再度造訪它頗早以前撰寫的程式碼）。

如果我們看看不超過此規模的一組專案，會發現 XP 的版本和 RUP 的反覆之間有合理的對映。下列範例說明它們在 RUP 內是如何規劃的。數字只是一種指標，但適用於這種規模的專案。它們包括除了程式碼之外可交付的所有必要 RUP 構件的準備工作所花的人力和時間，例如使用手冊、安裝和操作資訊等等。

XP 可修正專案的範例

範例 1 說明包含 5,000 行新 Java 程式碼的超小型專案，它需要大約 12 個人月，歷時 7 個月左右。[7]

---

[1] 在電子商業開發中預期的反覆持續時間可能比較短 — 很可能在 2 到 6 週的範圍內。

[2] 在初始階段的反覆通常較短。並請注意，RUP 的階段模型可接受範圍很廣的形狀，因此，在長期轉換階段，您可能面對一連串反覆，每一個反覆累積到交付客戶的軟體中，而且是以兩個月爲間隔，但進入轉換表示架構已完全穩定，且所預期的變更大部分是適合的、完成的且已矯正。

[3] 請參閱 [Beck00]，它說您可能無法執行一個有 20 位程式設計師的 XP 專案，但 10 位是絕對沒問題的。

[4] XP 聚焦於交付客戶直接的商業價值，主要是功能。在新增功能時，XP 很少直接考量架構，而是讓它從一系列軟體重構中浮現出來。如果解決方案的架構尚未建立完善，那麼隨著功能的新增，使先前的假設（和特定的解決方案）失效，並產生即使區域重構也無法改善的問題，其所帶來的風險將導致一連串嚴重的毀損。

[5] COCOMO II 是原本由 Dr. Barry Boehm 開發的典型 COCOMO 軟體成本預估模型的重做模型。它已校準過，適用於 2,000 slocs（程式碼行）的小專案。請參閱 [Boehm00]。

[6] 功能點是軟體大小的程式碼獨立測量，藉由限定對使用者提供的功能來達到目的，它完全是以邏輯設計和功能規格爲基礎。此定義來自 International Function Point Users Group 網站：http://www.ifpug.org/。

[7] 這看起來似乎太久了，但請注意，它涵蓋的是整個生命週期；從本來沒有人員且沒有定義需求的起點（只有構想的起源），到專案被接受和結束。它也是 COCOMO II 模型的輸出，允許更多排程壓縮，但會帶來增加成本和風險的不利後果。然而，根據表格的排程，可以在專案開始之後三個半月（在第一個建構反覆結束時）儘早使用有效的功能。

範例 1

	初始階段	詳述	建構	轉換
人員	1	1.5	2	2
持續週數	3	6	18	3
反覆數（及每一個反覆的持續週數）	1 (3)	1 (6)	3 (6)	1 (3)

範例 2 是關於一個有 10,000 行新 Java 程式碼的小專案，需要大約 27 個人月，歷時 8 個月左右。

範例 2

	初始階段	詳述	建構	轉換
人員	1.5	2.5	4	3
持續週數	4	7	20	4
反覆數（及每一個反覆的持續週數）	1 (4)	1 (7)	3 (7)	1 (4)

範例 3 說明有 40,000 行新 Java 程式碼的中型專案，需要大約 115 個人月，歷時 15 個月左右。

範例 3

	初始階段	詳述	建構	轉換
人員	3	5	10	8
持續週數	6	16	36	6
反覆數（及每一個反覆的持續週數）	1 (6)	2 (8)	4 (9)	1 (6)

在這些範圍內（涵蓋相當寬廣的開發範圍），RUP 反覆與 XP 版在目的和持續時間上都緊密對映。

大型系統開發不適合 XP

在超大型開發中—RUP 可以處理，但 XP 無法處理 —RUP 反覆會更久一點。範例 4 顯示有 1,500,000 行新 Java 程式碼的專案，需要 4,600 個人月，歷時 45 個月以上。[8]

範例 4

	初始階段	詳述	建構	轉換
人員	35	70	140	100
持續週數	20	50	100	20
反覆數（及每一個反覆的持續週數）	2 (10)	2 (25)	3 (33)	2 (10)

[8] 您可能認爲自己絕不會以此方式描繪專案特性；您主張此規模的專案應該分割成許多較小的專案（XP 可修正每一個小專案）。當然，此規模的專案會建構成子系統的系統（或對超大型專案而言，建構成系統的系統），但當這些子系統或系統需要整合到單一產品時，您需要考量架構，特別是關於軟體聚集與產生它們的團隊之間的介面。目前形式的 XP 不處理這些問題。

顯然在這個範例中，人員不是組織成單一完整團隊—專案是由數個團隊組成，每一個團隊負責處理子系統，子系統內再包含子系統，如此一來在較低層時，會在類似規模上規劃一個 XP 可修正規模的專案。然而，此專案是要代表整合系統，因此，在最上層需要規劃，才能在超出 XP 支配範圍的規模（33 週）上進行反覆。因爲此規模的整合專案的規劃慣性，所以，這些反覆是使用此持續時間。RUP 的 within-iteration planning（反覆內規劃）是透過整合建置計劃來完成（可能在系統和子系統層次上），而這些計劃將建構在較接近 XP 版本的規模上（針對此超大型系統），以及在最低層接近 XP 反覆的規模上（需要大約 2 週），尤其是在後期的詳述和建構階段。

因此，回到小型系統的範例，RUP 的反覆大約等於 XP 的版本，且 RUP 的建置大約等於 XP 的反覆。

RUP 的階段對映到 XP 的什麼？

至少第一眼看來，RUP 階段是對映到 XP 週期中未深入描述的層面。XP 的建構或至少描述看起來符合包含反覆的持續版本（與商業週期一致）。大家都知道，有某種*專案*必須引導才能存在（請參閱 [Beck01] 中的「專案範圍設定」），而此時建構的*大計劃*將分割成版本，再分割成反覆。[Beck01] 說大計劃幫助我們做這樣的決定：在專案中投資並不笨。

因此，在版本和反覆週期開始之前，XP 有一段時間的專案工作是要決定它適不適合以及花費多少成本，還有，若要這麼做，必須建構必要功能的一些粗略構想。這就是 RUP 所稱的初始階段。在您對 XP 的印象中，覺得這應該很快就可以完成。在 RUP，我們卻說，越謹慎小心越好—視內在風險而定。即使在 XP，您也會讀到有關此時需要完成的下列各項動作的文字（由一小群人員）：

- 一些需求誘導（寫一些「頭條新聞」）
- 一些規劃
- 與客戶的一些協商（設定期望）
- 在任何商業限制下進行

顯而易見，從冷啓動到有共識（這就是 RUP 所謂的「頭條新聞」）和企業案例（證明專案合理的預算和投資報酬 (ROI)）到軟體開發計劃的第一次刪減，可能需要幾天的時間。

就像在 XP 一樣，在 RUP，當您確定自己有不錯的成功機率之後，您在專案開始時同樣需要進行一大堆探索和規劃。如先前的範例 2 所示，RUP 規劃建議在初始階段要投資（在人力上）大約 $12,000 美元，以證明大約 $250,000 的支出是合法的。這視不同情況而異：在某些熟悉的問題領域中，您不需要做這麼多；例如，也許您不是採用冷啓動的方式。在另一些情況下，在不熟悉的領域上要重新開始，您需要花費更多，因爲它需要某種程度的研究和開發。

在其對應的初始階段之後，XP 會直接進入第一個版本的規劃。RUP 說詳述階段是跟在初始階段之後，而在詳述反覆中，解決方案的架構定型。在明顯的對照下，XP 建議版本規劃要功能導向，使所有版本都能爲客戶提供商業價值。XP 對於它所謂的基礎架構規劃的不屑態度是相當明確的；它認爲只要足夠支援該版本已選取的功能即可。以後可依需要新增更多基礎架構，如果因爲以後新增的功能使得較早期的基礎架構決策變無效而中斷系統，則程式碼會重構，使它可以運作。[9] 其想法是不要花費很多時間來建構一些對客戶沒有價值的東西。

相較之下，關於 RUP 有一些重點如下：

- ***RUP 的架構概念不只是基礎架構***

  引述 RUP 的說法，「軟體系統的架構（在某一時間點上）是系統透過介面互動的重要元件的組織或結構，內含由連續小元件和介面組成的元件。」

  因此，架構是從一組特定觀點，在適當摘要層次上處理整個解決方案—而不只是基礎架構。

[9] 但請注意，重構只導致區域改善：重構不解決整體架構。如果整體架構能夠解決錯誤問題，則重構不會產生正確的解決方案。唯一解決方案是大幅變更，或乾脆重新開始。這有明顯的預算和排程風險。

- ***RUP 這種可執行架構的概念爲客戶帶來商業價值。***

  它可以提早示範解決方案，滿足客戶非功能需求（以及流程中的一些主要功能需求）。

  以此方式降低風險—而非功能需求通常帶來風險—本身對客戶就具有很大的商業價值。

爲什麼我們相信需要以此方式描述流程？因爲 RUP 就是要趕走風險，而且我們相信 XP 方法無法修正的問題和系統有類別之分，讓架構能夠從重構程式碼浮現檯面—通常是更大更複雜的架構。風險之一，是在重構中進行區域變更（其必要性範圍將受到限制）時，將產生區域最佳化，而累積造成非常次最佳化解決方案。這不只是我們的主張：Kent Beck 本身在 [Beck00] 的 *Four Values* 這一章的 *Courage* 標題之下，也談到很多。問題在於 XP 對架構上重大變更的靈感起源幾乎隻字未提—只說它需要勇氣[10] 來套用它們！

有些變更很難重構。例如，要使單一使用者的單機系統變成多使用者的多重處理器系統，您必須重新設計很多東西，但所產生的系統仍然有許多限制。

RUP 對架構的強調就是爲了處理起始方法錯誤而需要完全重新考慮的那些情況。小型系統的風險較小—重構等於是系統的大掃除，而起始架構不正確的可能性並不高。

另請注意，在幾乎察覺不到風險的地方（因爲規模、新技術、不熟悉、複雜度、效能的其他迫切需要、可靠性、安全性等等），若解決方案可以一目了然的現有架構來提供，則 RUP 詳述階段（即處理架構問題的階段）可能非常短（而且本身會比較注重重要功能需求的誘導、雕琢和示範）。也就是說，RUP 生命週期會倒退到更接近 XP 生命週期。當問題可以由 XP 順利解決時，RUP 開發案例也會產生同樣輕量型的結果（雖然不完全相同）。

RUP 的建構階段相等於 XP 的版本系列，但有這樣的資格限定：如果您遵循 RUP，則必須提供每一個建構反覆的結果給客戶，以便與 XP 具有相同效果。ㄧ XP 實際上不等於 RUP 的轉換階段，因爲每一個 XP 版本都已掌握在客戶手中。在 XP 專案結束時，有一個 [Beck00] 所謂的專案結束（請參閱稍後的標題 *XP 生命週期*之下的資訊），會發生一些相同專案結案活動。

XP 的階段對映到 RUP 的什麼階段？

XP 的階段（適用於 XP 的版本和反覆）爲探索、確定和操縱。在版次上，這些階段與 RUP 專案管理規範中的特定作業集合（RUP 中的活動）一致。這些爲「規劃下一個反覆」（對映到探索和確定）及「監督和控制專案」（對映到操縱）。下列章節對此有圖解的說明。

XP 階段

在 [Beck00]，有一章是關於理想的 XP 專案的生命週期，第一層標題是**探索**、**規劃**、**第一個版本的反覆**、**生產化**、**維護**和**結束**。這些看起來是最上層階段，但結果並不十分正確。XP 專案將由 *起始*探索階段設定範圍，當專案結束時則由「結束」設定範圍。但主力是在版本，而且*每一個版本都有探索階段*。因此，包圍專案（並導致第一個版本）的探索階段是一種特殊案例，因爲要通過一個起始閘道（請參閱 [Beck01] 中的*專案範圍設定*），而是否繼續的決策就是在這裡產生的。XP 版本和反覆都有三個階段—探索、確定和操縱。在第一個版本之後，「維護」實際上是以 XP 專案的本質爲特色。

XP 生命週期

整體而言，有 7 個版本並超過 15 個月的專案，其 XP 生命週期類似「圖 1」。

[10] 勇氣是一種美德，這點沒有人會反對—而且有「好人」才擁有此美德的含意。然而，即使最勇敢的人有時候也需要有組織的工作方式和可解決棘手問題的架構。

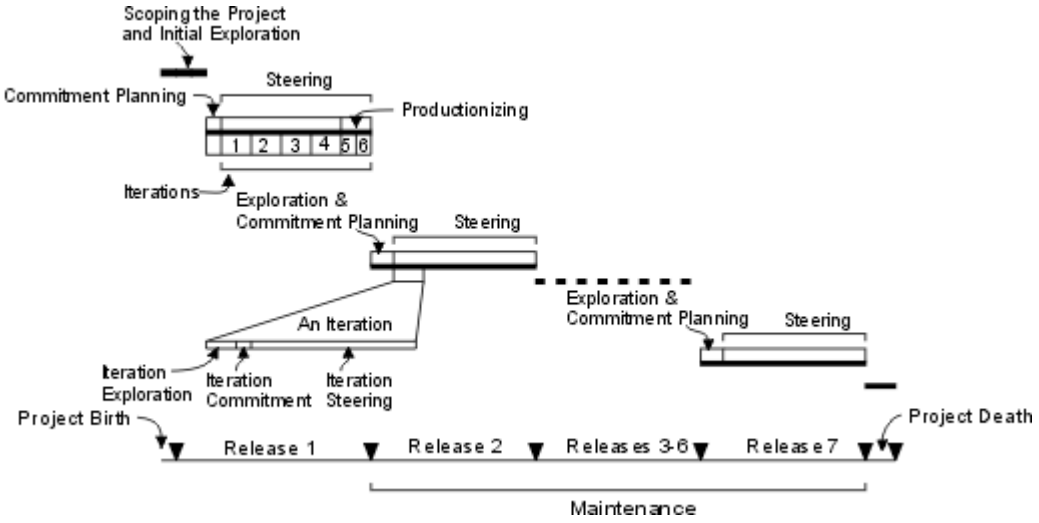

圖 1

第一個版本（三個月）之後的每一個版本，其持續時間大約爲兩個月，每一個反覆大約兩週，除了在版本的後面階段（即 [Beck00] 所謂的「生產化」），屆時反覆速度會加快。

從規劃度量觀點來看，這合理嗎？在這個設計過的範例中，我們試圖遵循 XP 規定的準則—即版本持續時間應該爲兩個月左右，第一個版本介於二到六個月之間—而且我們已將團隊規模維持在 10 人以下。如果此專案類似前面圖解的範例 3，它總共將提供大約 40,000 行 Java 程式碼行或 800 個功能點（如果我們接受 COCOMO II 模型中的轉換因素的話）。如果這在兩個版本之間平均劃分，則每一個版本大約提供 115 個功能點。

有疑問的部分是第一個版本：從起跑（假設已指派一名成員，未擷取需求，未完成範圍設定），到三個月內交付客戶具有正式作業品質的東西，將是一挑戰。即使我們允許高生產力，假設是 12 個功能點/人月（依據 The David Consulting Group 提供的產業資料，請參閱 http://davidconsultinggroup.com/indata.htm），也無法以任意高比率來提高人員配置—我們這裡所處理的問題範圍（115 個功能點）很小，無法分割成更小的片段，因此，大型團隊有可能攻擊它。

然而，如果我們假設可以完成它，則需要大約四人的平均團隊規模，且專案將以一個七人團隊來結束第一個版本。如果在專案持續時間多增加一名成員到團隊中，則專案剩餘時間內，我們會在含有八人團隊的 XP 標準模式（維護）下—即 XP 最適合的規模。當交付的規模增加時，生產力當然會下降一些，而縮短的版本間隔（兩個月）將抵銷增加的人員數目，因此，每一個版本所交付的額外功能加權與第一個版本大約相同。整體專案人力變成 108 個人月—比先前顯示的範例 3 少一點。

預設的 RUP 生命週期

相對地，類似規模的 RUP 專案的*預設*生命週期如下：

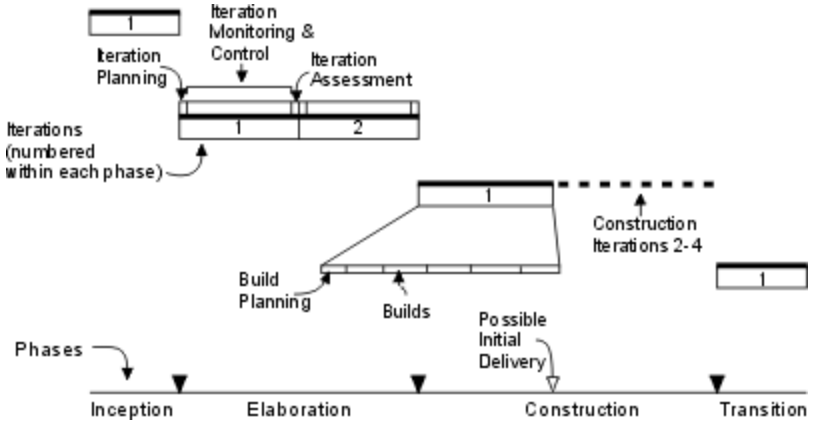

圖 2

同樣地，此計劃是以範例 3 爲基礎。RUP 反覆會在每一個階段內編號。使用預設 RUP 生命週期，很可能可以 *交付* 接近正式作業品質給客戶的最早機會是在第一次建構反覆結束時，這是發生在專案七個月以後。然而，此時 RUP 可交付大約總功能的四分之一或 200 個功能點（相對於三個月之後的 XP 115 個功能點）。在正常情況下，第一次交付客戶是發生在建構結束時，在此情況下，是在專案的十三個月以後，屆時所有功能實際上都應該有才對。這不表示客戶第一次看到該產品；有了 RUP，客戶可獲邀參加反覆評估，並可看到在測試下發展的產品。

爲什麼 RUP 與 XP 之間有這種差異存在？這是因爲在這個預設案例中，已假設此解決方案的架構風險夠大（例如因爲前所未有或全新技術，或「無功能」需求特別麻煩），足以保證在嘗試建置客戶需要的功能之前，可以靠詳述階段中的兩個反覆穩定它。這表示在詳述階段結束時，只有探索（及可能實作）架構重要需求。

XP 不會這麼做。它繼續在每一個版本交付一系列垂直完成的解決方案，並假設兩個版本之間的架構不會中斷，或重構可修復發生的毀損。我們認爲這樣會將 XP 可感應的應用程式限制爲可建置在已知功能的現有架構上的系統類別。

實際 RUP 生命週期

對於這種系統，實際 RUP 生命週期看起來不太一樣：詳述階段會比較短，而初始階段也可能比較短。如果我們把一次詳述反覆與建構反覆交換，可使第一個建構反覆提前幾個月結束，變成五個月而不是七個月。

這樣可能交付比預設範例中的 200 個功能點少一點（因爲在早期建構反覆中工作的人員數量少一點），但這樣可達到 XP 排程的靈活度—有足夠放寬措施來大量降低風險。最大限度，我們可以想像一個以 RUP 爲基礎的方法，它是在小型系統（假設有像 XP 範例中的 115 個功能點）起始交付時建立的，後面接著 RUP 的一些完整發展週期（完整初始階段、詳述、建構和轉換順序），以類似 XP 的維護模式來交付系統的其餘部分。

同樣地，在大約六個月左右，起始交付可能在 XP 建議的時間範圍外部限制，但 RUP 會認可並要求您針對似乎被 XP 掩蓋住的內容（例如版本的部署）加以規劃（以及配置時間和人力）。

我們可以從這些觀察中推斷出對 XP 適合的專案本質的其他限制。XP 明確地要求客戶和開發團隊之間要有非常密切的關係，要求客戶全天候在現場，並要求客戶寫稿及定義版本。在 XP，客戶實際上是團隊中的一個角色：扮演此角色的個人或群組不一定屬於個別認購公司，但他們有權代表認購該軟體的人發言。實質上，他們有這些需求。

比起對外界客戶正式簽約的開發，此種關係更常出現在機構內的開發。在這樣的環境中，兩個月的版本週期也更能夠被人接受。每兩個月跨公司部署新版本（具有新增特性，而不只是問題修正），可能帶來令人無法接受的後勤額外負荷。

有了這幾種放寬措施（小團隊、已建立的架構、低形式部署的機構內開發），一或多個適當量身訂作的 RUP 週期就會有類似 XP 開發的效果。起始版本持續時間可能比 XP 最佳效果久一點，但它的風險較低。

若反其道而行（大規模團隊、前所未有的架構、對交付和部署有限制的正式簽約的開發），就很難看出 XP 如何調整比例—而且老實說，本白皮書的來源並未宣稱它可以這麼做。在另一反面，RUP 是設計來應付這類專案所帶來的正式手續和形式。或許 XP 式方法（即使用像雙人程式設計和重構之類的 XP 技術）可以內嵌到較大型 RUP 專案*之內*，但必須等到架構穩定才行。

構件

RUP 說明超過 100 個構件，以致於有人批評它對小專案加諸令人無法忍受的形式負荷。這種看法犯了四個大錯：

- 專案不必產生所有構件：選取和量身訂作構件是流程必要的一部分。RUP 提供有關何者可省略和何者可量身訂作的準則。
- 構件（在 RUP，爲構件*說明*）是規格實體 —它指定作業要產生的資訊，而爲了有計劃地進行，它可說明構件的摘要*形式*。像模型、模型元素或文件（大約有一半的 RUP 構件歸類爲文件）之類的構件，其 RUP 特性化並不是爲了要默示特定*實現化*。

  可使用目的建置工具（例如 Rational Rose）來擷取及呈現 UML 模型，或在另一方面，可使用簡單圖形工具產生圖表，或甚至作爲白板上的草圖—這仍然算是 RUP 術語中的構件（不過白板會有消失不見的風險！）

  「文件」這個詞彙有歷史涵意，仍然會使人們把文件想成是具體的書面作業產品，具有一組特定（且必要）的段落標頭—全部都要填滿文字（也可以有圖表）才能成爲完整文件。但這只是 RUP 文件的一種實現化。

  RUP 文件是由文字和圖表組成的資訊集合。爲了有所幫助，RUP 指定資訊內容，而方便且有計劃的做法是指向一個範本，它列舉及說明要考量的項目和問題。當然，可以直接而正式地使用這些範本，使文件形式的構件具體化（一般含意）；亦即電子或書面。由於知道有許多專案想要這麼做，所以我們選擇以此方式，有點直接地呈現一組可使用的範本。但這麼做不是必要的—並非範本的一切內容都與特定專案有關—且 RUP 讓專案能爲構件自由選擇適當的實現化（和交付機制）。重點是它包含的資訊。
- 我們覺得流程工程師和專案經理應該要確認流程所有可能的工作產品，以便自覺地做出省略或量身訂作它們的決策，並充分瞭解省略的後果。我們相信以此方式建立完整而明確的構件清單有助於客觀地配置流程。它也支援較缺乏經驗的專案經理，讓他們能夠注意到經驗老道的專案經理「表示肯定」但未提到而被新手忽略掉的事情。擁有定義完善的文件集也可幫助客戶和專案經理提早同意要交付的內容以及交付的形式，以避免引起在專案結束時常見的令人不悅的爭論。
- RUP 說明數個複合構件，並繼續說明其元件爲構件。這樣可以對有用的作業輸入和輸出提出精確的說明。在其說明中，RUP 已在複合元件（例如設計模型）停止，並直接將類別當作設計模型的一部分而不是獨立構件來參考。確認這些爲構件可以準確說明其用途，又能遵守流程 Meta 模型，但代價是會增加整體構件計數。

XP 似乎免於遭到構件繁重的批評，但這過於單純化，原因有兩個：

- XP *已經量身訂作*爲適合某一種開發，能處理某種規模的 RUP 規範子集，因而可能需要較少的構件。
- XP 強調使用者情景和程式碼的重要性，但在說明流程時會順便提起其他工作產品，因此構件計數不像第一次看見時那麼小。

以 XP 方法而論，在我們爲了做此比較而使用的來源資料的三本書中（因爲它們是流程的一流參考書），「構件」和「工作產品」這兩個詞彙並未出現在索引中，而這或許這不足爲奇。然而，要遍覽文字並選出構件參考一點也不難。以下是一些範例：

- 報導
- 限制式
- 任務
- 技術作業
- 驗收測試
- 軟體—程式碼

- 版本
- 隱喻
- 設計—CRC、UML 草圖
- 設計文件—在專案結束時產生
- 程式碼撰寫標準
- 單元測試
- 工作區（開發和其他機能）
- 版本計劃
- 反覆計劃
- 會議報告和記錄
- 整體計劃—預算
- 進度報告
- 報導評論
- 作業評論
- 問題（和相關聯的資料）
- 來自交談的其他文件
- 支援的文件
- 測試資料
- 測試架構工具
- 程式碼管理工具
- 測試結果
- Spikes（解決方案）
- 作業工作時間記錄
- 度量資料
    - 資源
    - 範圍
    - 品質
    - 時間
- 其他度量
- 追蹤結果

我們大約有 30 個構件，其中有些也是複合的。此清單並未詳盡，只作爲指標。XP 書籍不花費太多時間說明其中大部分，在寫書的層次上，我們並不期望它們這麼做。然而，專案瞭解它們之後，會需要更詳細的內容和形式。專案當然可以很快達成此目的，但需要扣掉實際工作的時間；反之，RUP 則提供直接指引，故可節點專案時間。

爲什麼 XP 不需要所有 RUP 構件？

原因之一是 XP 沒有 RUP 的範圍。這是有意圖的：XP 是關於符合商業需求的程式設計。商業需求如何發生—以及它如何建模、擷取和推論—這並非 XP 的主要考量。本身是客戶的 XP 團隊成員，向 XP 開發團隊報導需求的精華；他們也是商業價值的仲裁者。報導是如何以該形式表達出來，此一神奇力量並不在 XP 的考量範圍之內。因此，比方說，RUP 在其商業模型規範中說明的內容已超出 XP 的範圍（RUP 的商業模型有 ~14 個構件）。XP 向客戶說明有

一般版本的流程。這些版本的部署底層機制不是開發的重點，因此，RUP 部署規範大大超出 XP 的範圍（RUP 的部署有 ~9 個構件）。因此，對於 XP 可負責的小專案，您預期可在量身訂作 RUP 時省略 ~23 個構件。

另一個原因，是 XP 主張需求和設計的擷取可以簡單完成：需求以使用者情景擷取（有時候必須分割），然後分解成作業—它們在本質上就是設計—記住系統的隱喻。這對所有系統都行得通嗎？當然不是。對某些系統行得通嗎？當然，爲了對 XP 公平起見，它並未聲稱可以處理所有系統。而且，或許 XP 作者在做這些聲明時只是說說而已。

更大更複雜系統的預期行爲若無一些有計劃有步驟的方法（例如使用案例），恐怕很難連貫起來。不僅無法根據客戶和開發人員之間的交談來一貫地闡述複雜的使用者報導，人類的記憶也不可靠。而且，開發在大型系統中*實現*複雜行爲的結構，受惠於塑造及協調該系統架構的各種摘要視圖的能力。RUP 在「需求」（~14 個構件）和「分析與設計」規範（~17 個構件）中說明的構件，可讓它對付這些系統的變化、大小和複雜性。

在 RUP 的小專案導覽圖中，構件的計數（針對「需求」和「分析與設計」）減少爲 7 個，其中有些減少是參考複合構件而達到的。這不是耍花招；它處理構件的方式與 XP 一模一樣。例如，由於它在小專案中實現的可能方式，RUP 對設計模型的說法如下：

> 「「設計模型」原預期爲透過一系列集體檢討的階段作業而逐漸形成，其中開發人員將使用 CRC 卡和手繪圖表來探索及擷取設計。只有開發人員覺得有用時，才要維護「設計模型」。它與實作並不一致，但可存檔供作參考。」

最後，RUP 允許專案管理在必要時有更多的形式和繁文縟節，某些合約的安排上會是如此。同樣地，許多 RUP 的專案管理構件（在專案管理規範中有 15 個）是複合構件的一部分，爲了因應專案形式的要求，只需要以個別文件實現之。例如，在 RUP 中，軟體開發計劃「包含」像風險管理計劃和產品驗收計劃之類的構件。在較小較不正式的專案中，可以只在軟體開發計劃中使用一兩個段落來處理這些計劃所提出的問題。在 RUP 的小專案導覽圖中，專案管理構件數減少爲 6 個。

比較小專案的構件

因此，當您爲小專案量身訂作 RUP 及量身訂作構件需求時，整體上會發生什麼事？當您查看 RUP 中的小專案導覽圖及計算構件時，數目爲 30（如果您省略一些部署，則爲 26）—構件並沒有那麼多！XP 含糊其詞的內容，RUP 會明確描述，這可讓您決定什麼是必要的，並提供如何選擇的指引。現在，兩邊的精度不同，但重點是在 RUP 中，原本 XP 可輕鬆處理的類型的小專案的構件計數與 XP 的順序是相同的。

XP 構件與 RUP 構件的粗略對映

XP	RUP 小專案導覽圖
報導 來自交談的其他文件	願景 名詞解釋 使用案例模型
限制式	增補規格
驗收測試 單元測試 測試資料 測試結果	測試模型
軟體—程式碼	實作模型
版本	產品 版本注意事項
隱喻	軟體架構文件
設計—CRC、UML 草圖 任務 技術作業 設計文件—在專案結束時產生 支援的文件	設計模型
程式碼撰寫標準	設計準則 程式設計準則
工作區（開發和其他機能） 測試架構工具	 工具
版本計劃 報導評論 作業評論 反覆計劃	軟體開發計劃 反覆計劃
整體計劃—預算	商業案例 風險清單 產品驗收計劃
進度報告 作業工作時間記錄 度量資料：資源、範圍、品質、時間 其他度量 追蹤結果 會議報告和記錄	狀態評量
問題和相關聯的資料	變更要求
程式碼管理工具	配置管理計劃 專案儲存庫

	工作區
Spike	原型
	開發案例 專案特定範本

準則

RUP 提供與構件相關聯的準則，這些準則實際上就是關於構件的詳細資訊；包括它的意義、表示法、與其他構件的關係、內容和使用。這些準則將涵蓋數百個印出的實際頁面，看起來很嚇人，但您只要閱讀您需要的那些構件的必要資訊即可。

XP 書籍也包含作法和構件的很多準則，但不嘗試（或需要）嚴格建立關係模型。不過，關於 XP 的文獻總數也很可觀。目前所寫的有三本書共 600 頁，還有即將問世的兩本書，又會增加大約 700 頁左右。還有一本關於重構的書 [Fowler99]，大約 400 多頁。

除此之外，有幾個網站也有關於 XP 的報導。不過其涵蓋面可能重疊—從不同觀點檢驗 XP 並加入經驗庫中。實際上，這點明了 RUP 和 XP 之間的差異： RUP 是產品；XP 不是。XP 沒有單一來源，不過書籍是不錯的入門工具。最快「學會」XP 的方式是透過其背後思想領導者所提供的商業培訓課程。[11]

[11] RUP 有時候是描寫成「專用的」，但任何人只要遵守版權和授權合約，都可以購買 RUP、利用它的構想、新增內容、刪除與其組織或專案無關的部分等等。以書面形式表明的 XP 也是有智慧財產權的，由作者和出版商所擁有；唯一的差異在於您可以從其來源完全理解它們到什麼樣的程度。 RUP 是產品，力求完整；目前的 XP 書籍則需要一些補充（可透過密集培訓或諮詢）。

任務

RUP 正式定義「作業」一詞爲角色所執行的工作，它使用及轉換輸入構件，來產生新的及變更的輸出構件。RUP 接著繼續列舉這些作業，並根據「規範」或主要「重點區」（如 RUP 所定義），在專案內將它們分類。這些規範如下：

- 商業模型
- 要件
- 分析與設計
- 實作
- 測試
- 部署
- 配置與變更管理
- 專案管理
- 環境

作業透過它們產生及耗用的構件而與時間相關：當作業的輸入可用時（而且是在適當的成熟狀態下），在邏輯上作業就可以開始了。這表示如果構件狀態允許，生產者-消費者作業配對在時間上可以重疊；它們不需要嚴格地按照順序。

RUP 中的作業是要使產生構件的智慧流程減少不透明化—對於如何產生提供一些有力的指引。作業也可以用協助專案經理進行規劃。從生命週期方面來描述，交織在 RUP 中的構件和作業是「最佳作法」：軟體工程準則已證明能夠產生高品質軟體來建置到可預測的排程和預算上。

透過其作業及其相關聯的構件，RUP 可支援及實現這些最佳作法—它們是貫穿整個 RUP 的主題。請注意，XP 也使用「作法」的概念，但誠如我們所見到的，它與 RUP 最佳作法的概念並不完全一致。

XP（請參閱 [Beck00] 第 9 章）呈現吸引人的簡易軟體開發視圖，內含四項基本作業，將根據一些支援的作法來啓用及建構它們：

- 程式碼撰寫
- 測試
- 接聽
- 設計

實際上，XP 的作業在範圍上較接近 RUP 的規範，而較不接近 RUP 的作業，而在 XP 專案發生的事（除了程式碼撰寫、測試、接聽和設計以外），有很多是來自其作法的詳述和應用。

XP 有沒有 RUP 作業的同等項？

有的，但 XP 的「作業」並未正式加以識別或描述：例如，在 [Jeffries01] 的第 4 章*使用者報導*中，您會發現它的標題是 “*Define requirements with stories, written on cards*” ，然後整章的內容混合了流程說明和關於有哪些使用者報導及如何（由誰）產生它們的指引。接下來，這些書籍以大標題（它們在 [Jeffries01] 中是混合的）描述 XP—有些是以構件爲焦點，有些是以活動爲焦點）；其「完成之物」和「產生之物」將以不同程度的指示和細節加以描述。

RUP 的表面指示來自它的完整性及其有計劃有步驟的作業處理上更大的形式，還有它們的輸入和輸出。XP 也有指示，但試圖保持「輕量型」，省略了形式和詳細資料。在專案中實作 XP 時，必須加上 RUP 呈現的詳細資料。缺乏具體性既不是優點也不是缺點，但不應該將簡化與 XP 缺乏詳細資訊加以混淆。在某個時間點，專案的人必須知道該做什麼，屆時會需要詳細資料。

規範

RUP 的規範是作業（及相關聯概念）的集合，會產生一組特定的構件，代表軟體開發中的一些重要層面或考量。此用法與字典中將規範定義爲知識或教導的分支非常吻合。

如前所述，RUP 的規範是指商業模型、需求、分析與設計、實作、測試、部署、配置與變更管理、專案管理和環境。這不涵蓋組織或企業在雇用人員開發、部署、操作、支援、販售、行銷或處理大量軟體的系統時可能執行的每一個層面。例如，RUP 目前不涵蓋系統工程。它也不涵蓋某些國際軟體流程標準的所有需求，例如 ISO 15504，它是從軟體獲取與人力資源管理的相關層面上加以描繪。這是自由選擇的：這些其他重要層面已超出 RUP 的工程焦點。

XP 明確地進一步自我限制：它包括四個基本活動 —程式碼撰寫、測試、接聽和設計（如前所述，與 RUP 規範更加吻合）—使用一組作法來執行，這些作法需要執行其他活動來對映到 RUP 中的一些其他規範。從 [Beck00] 逐項引用的 XP 作法如下：

- 「*The Planning Game*—結合商業優先順序和技術評估，儘快決定下一版的範圍。當現實追上計劃時，要更新計劃。
- *小型版本*—使簡易系統快速進入正式作業，然後在極短的週期內發行新版本。
- *隱喻*—以關於整個系統如何運作的一個簡單報導來引導所有開發團隊。
- *簡易設計*—不論什麼時間，系統應該設計成越簡單越好。一發現過於複雜，就馬上消除。
- *測試*—程式設計師不斷撰寫單元測試，它們必須完美無瑕地執行，開發作業才能夠繼續下去。客戶撰寫測試，則是爲了證明特性已經完成。
- *重構*—程式設計師重組系統，但不變更系統移除重複、增進通訊、簡化或增加彈性的行爲。
- *雙人程式設計*—所有正式作業程式碼是由兩位程式設計師在同一部機器上撰寫。
- *群體擁有權*—任何人都可以隨時隨地在系統中變更任何程式碼。
- *連續整合*—每次完成作業即整合和建置系統，一天多次。
- *40 小時週*—通常一週工作不超過 40 小時。第二週絕不連續加班工作。
- *現場客戶*—團隊中包括一位真正的使用者，可全天候回答問題。
- *程式碼撰寫標準*—程式設計師依照規則撰寫所有程式碼，規則強調程式碼溝通順暢。」

關於*簡易設計*主題，有一點值得注意：在有經驗的旁觀者眼中，「過於複雜」這個詞彙有一點主觀，很難去定義它。

因作法的結果而執行的活動，例如，*The Planning Game* 主要對映到 RUP 的專案管理規範。然而，有些主題已超出 XP 的範圍；例如，商業模型。需求*誘導*大部分已超出 XP 的範圍—由客戶（與團隊一起在現場）定義及提供需求（以報導的形式）。所發行軟體的部署超出 XP 的範圍。由於所處理的開發範圍和類型，所以 XP *非常*輕率地處理 RUP 詳述涵蓋的環境與配置和變更管理規範中的問題。

在 RUP 中使用 XP 的作法

在 XP 和 RUP 重疊的規範中，XP 所描述的一些作法可以運用在 RUP 中（有些已經運用了）；例如：

- *雙人程式設計：*XP 要求正式作業程式碼由雙人一組的程式設計師在同一個工作站上製作，並主張這對於程式碼品質有益，而一旦獲得技巧，將更有趣。RUP 不在精細層次上說明程式碼正式作業的機制，但當然可以在 RUP 式流程中使用雙人程式設計。關於此作法和*先測試設計和重構*（如下所示）的資訊，現在由 RUP 以白皮書形式提供。顯然，在 RUP 中使用此作法並不是一項需求，我們也不認爲有需要規定它。

  對產業級有利的證據薄弱而多軼聞；[Nosek98] 和 [Williams00] 所做的研究已比較過雙人組和個人（單獨工作），但出色的團隊不是以這樣的方式工作。在團隊環境中，其開放溝通的文化允許個人隨時向同僚（和小組負責人或主管）發問並依照定義的流程工作，我們大膽猜測雙人程式設計的好處（從總生命週期成本的效果來看）將更難以分辨。在運作良好的團隊中，很自然大家會一起討論和解決問題，而不是被迫的。

  [Beck00] 對人員和流程的看法有點悲觀，它表示：雙人程式設計的另一個強大特性是有些作法沒有它行不通。在壓力之下，人會恢復原來的做法。他們會省略撰寫測試程式碼。他們會延後重構 。這些原本都是會導致更好結果的自然作法”—爲什麼沒有人要這麼做？

  理想流程的實例化是不打擾人的；建議必須在 “micro” 層級上實施它會令人不悅。然而，在某些狀況下，雙人組工作顯然有好處，因爲彼此可以*互*相幫忙；例如：

  - 在團隊形成的初期，大家開始認識對方
  - 在對某些新技術不熟悉的團隊中
  - 在經驗老手和新手*互*相混合的團隊中
- *先測試設計和重構：*這些是不錯的技巧，可套用在 RUP 的實作規範中。尤其是先測試設計，是在詳細層次上釐清需求的絕佳方式。
- *現場客戶：*從效率提高方面來看，許多 RUP 作業透過客戶到現場作爲團隊成員之一，而獲益良多。讓客戶以此方式進場，可減少對許多中間交付項（尤其是文件）的需求。

  XP 不願意說除了程式碼以外都需要擷取，並強調交談是比較好的溝通媒介。這有賴持續性和熟悉性才能成功。當系統必須變革時，即使是小型系統，也必須產生其他內容。

  XP 最後的確允許這麼做，但卻是事後的想法；例如，在專案結束時設計文件。事實上，XP 並不禁止製作文件或其他構件（只是以自動方式執行），更確切地說，您只應該製作自己真正需要和使用的東西。RUP 也一樣，但如果持續性和熟悉性不理想時，它仍然會列出及說明您*可能*需要的內容。
- *程式碼撰寫標準：*RUP 有一個構件（程式設計準則）幾乎一律被視爲必要的（大部分專案風險設定檔是量身訂作的主要驅動者，會讓它變成如此）。
- *連續整合：*RUP 透過在子系統和系統層次（在反覆內）的建置來支援它；已做過單元測試的元件是在新興的系統環境定義中整合和測試。

不調整比例的 XP 作法

然而，有些作法不調整比例（XP 也不主張它們可以這麼做），所以我們是在 RUP 的這個附帶條件之下使用它們；例如：

- *群體擁有群：*讓小團隊中熟悉所有程式碼的成員去負責小系統或大系統中的子系統很有幫助。您是否要讓所有團隊成員有隨地變更的同等權力，視系統或子系統的本質或複雜性而定。讓目前處理程式區段的個人（或雙人組）寫修正程式通常更快（而且更安全）。

  對即使最佳的程式碼的熟悉度也會隨時間逝去而快速消失不見，尤其如果它的演算法很複雜的話。

- *重新建構：*在大系統中，經常重構並不能替代架構的缺乏。[Beck00] 說：XP 的設計策略類似爬坡演算法。您得到簡單的設計後，讓它變得複雜一點，變得簡單一點，再變得更複雜一點。爬坡演算法的問題在於達到區域最佳解時，沒有小幅變更可以改進情況，但大幅變更倒是可以。”

  在 RUP 中，架構提供「廣泛」的檢視和存取權，使大型複雜的系統也容易處理。

- *隱喻：*對於較大型的複雜系統，隱喻架構是不夠的。RUP 爲架構提供更豐富的敘述，而不只是像 [Beck00] 輕描淡寫的「大型機器和連線」而已。
- *快速版本：*客戶可接受及部署新版本的速度視許多因素而定；其中之一通常是系統的大小，它通常與商業影響相互關聯。兩個月的週期對於系統的某些類別來說可能太短—部署的底層機制可能禁止它。

有些作法乍看之下顯然很健全，而且在 RUP 中可能可以使用，但普遍套用時則需要一些詳述和注意：

- *簡單設計：*XP 非常功能導向：使用者報導經選取後分解成作業，然後實作。根據 [Beck00] 的說法，軟體的正確設計需具備下列條件：“

  1. 執行過所有測試
  2. 沒有重複邏輯…
  3. 向程式設計師陳述每一項重要意圖
  4. 具有最少的類別和方法。”

  任何現在不需要的東西，XP 認爲不應該加入。在 RUP 中處理一種叫作非功能的需求類別時有一個問題，與區域最佳化問題有點關聯。這些需求爲客戶帶來商業價值，但更難以報導形式來表達—有些 XP 所稱呼的限制即屬此類。

  RUP 不主張以任何推測的方式來進行非必要的設計，但它主張設計時腦中要有一個架構模型；該架構模型爲符合非功能需求的關鍵之一。

  因此 RUP 與 XP 吻合：「簡單設計」應該執行過所有測試，附帶條件是要包括示範該軟體符合非功能需求的測試。同樣地，唯有當系統大小和複雜性增加、該架構無任何先例或非功能需求很麻煩時，這才會隱約成爲大問題。例如，資料轉置的需求（爲了在異質分散式環境中操作）似乎使程式碼太過複雜，但它仍然是一項廣域需求。

- *40 小時週：*就像在 XP 一樣，RUP 也建議加班工作不應該成爲習慣。XP 不建議強迫 40 小時限制，因爲瞭解到工作時間有不同的耐受性。只是爲了看到完事的滿足感而讓軟體工程師長時間工作卻無額外報酬，主管也不一定要加以制止，此乃眾所周知之事。

  主管不應該利用它或強加它—主管應該要每次收集*實際*工作時數的度量（即使無報酬也一樣）。如果任何人的工作時數記錄有一段很長時間居高不下，則當然應該介入調查。但這些是在發生它們的特定情況下，在瞭解團隊其餘成員的考量之後，主管和員工之間要解決的問題。四十小時只是一個指引（卻很重要）。

角色

在 RUP，作業是由角色執行。[12] 角色對特定構件也負有責任—負責的角色通常建立構件並確定其他角色所做的變更（如果允許這些變更的話）不會破壞構件。RUP 中的角色可由個人或一組人執行。個人或群組同樣可以執行數個角色。角色不一定要對映到組織中的單一職位或「位置」；角色與組織單位的對映也可以是多對多關係。

RUP 角色

RUP 定義共 30 個角色：與規範之間沒有確切的對映，因爲角色—例如軟體架構師—不一定限制在一個規範，而是粗略地（將角色放在它主要隸屬之處）：

- 商業模型有三個角色
- 需要有五個角色
- 分析與設計有六個角色
- 實作有三個角色
- 測試有兩個角色
- 部署有四個角色
- 配置與變更管理有兩個角色
- 專案管理有兩個角色
- 環境有三個角色

爲什麼 RUP 有這麼多角色？

RUP 中的角色是用來分割作業及仔細區別要執行該角色所需的技巧和能力，這有助於指引執行角色人員的選擇。當角色的重要性變更時，部門等級也有助於識別新的組織職位。例如，在小型非正式專案上，專案經理的角色和配置管理者（RUP 角色）可由同一人執行；在大型正式軍用航空專案中，配置管理者的工作相當專業及繁重，無法授權交由小團隊執行。RUP 以此方式說明角色，來幫助其對映。

XP 角色

[Beck00] 識別 XP 適用的七個角色，接著說明角色的責任、要執行該角色的人所需要的技巧和特點。這些角色如下：

- 程式設計師
- 客戶
- 測試者
- 追蹤者
- 輔導員
- 顧問
- 大老闆

在其他一些 XP 書籍中，在詳述角色執行之作業的地方，有對這些角色的參考。

[12] 更清楚（而明確）地說，作業是由*扮演*角色的個人或團隊執行.

XP 角色和 RUP 角色數量的差異很容易解釋：

- XP 不涵蓋所有 RUP 規範。
- XP 角色實際上比 RUP 角色更接近職位；例如，XP 的程式設計師實際上執行多個 RUP 角色—即執行者、整合者和程式碼複查者的那些角色—而這些只需要稍微不同的能力。

當 RUP 角色對映到小專案時（如 RUP 小專案導覽圖所示），所對映的 XP 式角色（即職位）的數量大幅減少 30。在小專案導覽圖中，職位數爲 5 個，如下表所示。

RUP 角色對映到 ABC 公司的小專案

ABC 公司工作職稱	RUP 角色
專案管理者	專案管理者 流程工程師 部署管理者 需求檢閱人員 架構檢閱人員
ABC 公司主管	專案檢閱人員 關係人 (Stakeholder) 需求檢閱人員
首席程式設計師	系統分析師 需求指定者 使用者介面設計師 軟體架構師 設計檢閱人員 流程工程師 工具專家 配置管理者 變更控制管理者 *以及勉強與*程式設計師相同的角色
程式設計師	設計師 執行者 程式碼複查者 整合者 測試設計師 測試者
行政助理	*負責：* • *維護「小專案」網站* • *在規劃或排程活動中協助*專案經理*角色* • *在控制構件變更時協助*變更控制管理者*角色* • *必要時也爲其他角色提供協助*

結論

XP 其實與 RUP 並不相同。RUP 是一種流程架構，其中的特定流程可以配置及建立實例。RUP *必須*配置，而這實際上是 RUP 本身所定義的一個必要步驟。嚴格說來，您應該比較含有 XP 的 RUP 量身訂作版本，與已量身訂作爲 XP 明確建立的專案判別式（以及可推論的那些判別式）的 RUP。這種量身訂作的 RUP 流程可容納一些 XP 作法（例如雙人程式設計和先測試設計和重構），但與 XP 不完全相同，因爲它確認架構的重要性、摘要（在建模中）、風險及其不同的時間結構（階段、反覆）。

RUP 將允許建構流程來容納在規模或種類上超出 XP 範圍的專案。RUP 是重量型，只因爲它是可以輕量或重量的流程系列的完整*說明*—在構件、交付項、形式、規定、儀式或「重量」的任何其他量度中—也可以在實作中。XP 當然是輕量型，因爲它刻意在（輕量型）流程的實作上指引，但 XP 的說明（至少在書籍中）亦不詳盡。因此，在 XP 實作中，有需要探索、創造或快速定義的東西。所以，與 RUP 相較，XP 在敘述性資料中也是輕量型。

這有可能會變更，事實上，它可能已與另外兩本書籍的發佈資訊交換，其中之一的 [Succi01] 有 512 頁。然而，照目前情況看來，這兩個方法之間的正式通過人力的設定檔並不相同。RUP 在培訓需求和流程量身訂作上直接更換許多人力。組織也很可能量身訂作 RUP，以應用到全組織的特定類型和大小的專案上，並在數個專案中使用結果。有了 XP，會有一些必要的直接培訓，但其餘正式通過的人力將散佈到專案中，因爲它會詳述並擷取所有那些會變成使 XP 可以運作所需要的東西。XP 未明顯激發「組織記憶」的擷取，使採用它的組織（如果它沒有保存其流程經驗的話）易流於人員接替。

將 RUP 標示爲重量型，以及將 XP 標示爲 XP 輕量型，卻無進一步的資格限定，這會對兩者造成傷害，因爲它使兩者的定義和用途混淆不清。而且，如果是以輕蔑的方式進行，只是毫無意義的故作姿態罷了。這些是「重量型」或「輕量型」流程的實作，當狀況需要時，它們應該是重量型或輕量型。

XP 不是開放式的，一切都要遵守規範 —它聚焦在特定軟體開發層面和交付價值的方式，且關於其達成方式有非常嚴格的規定。

RUP 的涵蓋面更廣更深，這說明了它的明顯「規模」。然而，在流程的微觀層次上，RUP 有時候會允許並提供同等有效的替代方案（XP 則不然）；例如，雙人程式設計的作法。這並不是對 XP 的批評；只是 XP（顧名思義）如何聚焦的說明而已。

參照

[Beck00] *Extreme Programming Explained,* Kent Beck, Addison-Wesley, 2000

[Beck01] *Planning Extreme Programming,* Kent Beck, Martin Fowler, Addison-Wesley, 2001

[Boehm00] *Software Cost Estimation with COCOMO II,* Barry W. Boehm et al, Prentice Hall PTR, 2000

[Fowler99] *Refactoring: Improving the Design of Existing Code,* Martin Fowler et al, Addison-Wesley, 1999

[Jeffries01] *Extreme Programming Installed,* Ron Jeffries, Ann Anderson, Chet Hendrickson, Addison-Wesley, 2001

[Kruchten00] *The Rational Unified Process, An Introduction, Second Edition,* Philippe Kruchten, Addison-Wesley, 2000

[Martin01] *Extreme Programming in Practice,* Robert C. Martin, James W. Newkirk, Addison-Wesley, 2001（尚未發行）

[Nosek98] *The Case for Collaborative Programming,* John T. Nosek, *Comm. ACM,* Vol. 41, No. 3, 1998, pp. 105-108

[Succi01] *Extreme Programming Examined,* Giancarlo Succi, Michele Marchesi, Addison-Wesley, 2001（尚未發行）

[Williams00] *Strengthening the Case for Pair Programming,* Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries, IEEE Software, Vol. 17, No. 4, 2000, pp. 19-25

Rational
the software development company

兩個總公司：

Rational Software
18880 Homestead Road
Cupertino, CA 95014
電話：(408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
電話：(781) 676-2400

免付費專線：(800) 728-1212
電子郵件：info@rational.com
網址：www.rational.com
國際辦事處：www.rational.com/worldwide