RUP® と XP の比較

John Smith

Rational Software ホワイト・ペーパー

TP 167, 5/01

Rational®
the software development company

目次

概要

このホワイト・ペーパーでは、Rational Unified Process® (RUP) とエクストリーム・プログラミング (XP) を比較します。RUP は、Rational® Software 社が長い年月をかけて開発したプロセス・フレームワークです。小規模なプロジェクトから大規模開発まで、さまざまなソフトウェア・プロジェクトで幅広く使用されています。XP は、要求が変化する環境で比較的小規模なシステムを構築する場合に効果的な方法として、注目を集めているソフトウェア開発手法です。

比較する内容は、それぞれのプロセスを表す性質とスタイル、基礎的な構造、前提事項、表現です。また、それぞれの手法の対象と使用結果についても検証します。

多くのプロセスには、システマティックな比較が可能な共通要素があります。プロセスには、一連のタスクあるいはタスク・グループが必要です。これらは、ロール (通常は、個人またはチームに割り当てられます) によって実行され、成果物やワーク・プロダクトが作成されます。これらの一部またはすべてが顧客に納品されます。多くのプロセスでは、プロセスのインスタンスに開始と終了の決められた期間が設定され、重要なアクティビティー (または一連のアクティビティー) の完了と関連する成果物の生成を示す中間マイルストーンがあります。したがって、ここではプロセスの次のような側面を調査し、これらを使用して比較を行います。

- **時間と作業量の配分** — 各プロセスが時間軸に沿ってどのように配置されるか、スタッフの作業量がどのように配分されるかを比較します。
- **成果物** — ワーク・プロダクト、つまり XP と RUP に基づくプロジェクトで *作成された結果* について比較します。
- **タスク** — それぞれのプロセスで成果物を作成する方法について説明します。
- **作業分野** — XP と RUP で、ソフトウェア・エンジニアリングに関係のある主な領域がどのように分けられているかを比較します。
- **ロール** — RUP における (タスクを実行する) ロールと、XP のロール (チーム内で言う「立場」に近い) について、相違を検証します。

このホワイト・ペーパーの意図は、プロセス全体における RUP と XP の相対的な位置を明らかにして、これらがお互いに協調するものであることを確認し、「重量級の RUP よりも、軽量級の XP の方が望ましい」というような考え方を払拭することにあります。

このホワイト・ペーパーを作成するにあたり、XP に関する主な情報源として Addison-Wesley 社刊行の以下の 3 冊の XP 関連書籍を使用しました。

- Extreme Programming Explained [Beck00] (邦訳:「XP エクストリーム・プログラミング入門」)
- Extreme Programming Installed [Jeffries01] (邦訳:「XP エクストリーム・プログラミング導入編」)
- Planning Extreme Programming [Beck01] (邦訳:「XP エクストリーム・プログラミング実行計画」)

これらの書籍は XP に関する情報についての主導的な資料であり、多くの読者や XP の潜在的なユーザーがその手始めに利用しています。ほかの書籍も探しましたが、このホワイト・ペーパーの執筆時点では発行されていませんでした。RUP の情報源は、Rational の RUP 製品自身です。

このホワイト・ペーパーは、XP または RUP のチュートリアルではありません。これらの手法について、既にいくらか知識のある開発者や、参考文献 (例えば、XP については [Beck00]、RUP については [Kruchten00]) を読んだことのある開発者であれば、簡単に読み進めることができるでしょう。

時間と作業量の配分

RUP は (ソフトウェアを開発するための) プロジェクトを扱い、その存続期間は「方向づけ」、「推敲」、「作成」、「移行」というフェーズに分割されます。さらに各フェーズは数回の反復に分割され、各反復では数回のビルドが必要となります。

RUP のほとんどのプロジェクトでは、反復の期間は 2 週間から 6 カ月[1]の間です。また、プロジェクト全体での反復の回数は 3 回から 9 回です。したがって、こうした既定の設定では、RUP の扱うプロジェクトは 6 週間から 54 カ月の範囲になります。

XP での反復の期間は、およそ 2 週間です。XP のリリースは 2 カ月です (またはもう少し長くなります)。これは顧客によって定義され、顧客に対してリリースされます。期間に関して、XP のリリースは、RUP の (少なくとも推敲または作成[2] における) 反復に似ています。ただし、確立されたアーキテクチャー・ベースライン[3]に基づく、XP に適したプロジェクト、すなわちチーム規模が最大 10 人[4]のようなプロジェクトの場合です。

これを示すために、XP の最大のチーム要員を 10 人と仮定します。このチームに適切な最大のプロジェクトを検証し、RUP がこの規模のプロジェクトや、より小規模なプロジェクトをどのように処理するかを確認します。通常、評価モデルとして COCOMO II[5] を使用する場合、10 人のチームが携わるプロジェクトの最大期間は 15 カ月であると予測されます。15 カ月以上続くプロジェクトは、通常、10 人以上のチームが担当します。より長いスケジュールを与えることができるなら、10 人の開発者でもさらに大規模なシステムを作成できます。ただし、通常はスケジュールが重要になるので、可能な場合は、効果的にスタッフを追加する計画を立てます。このプロジェクト例では、およそ 40,000 から 50,000 のソース・コード行 (sloc) をスクラッチから作成します (Java では 800 から 1000 のファンクション・ポイント[6]となります) 。

このモデルに従うと、効率的に開発を進める場合は、これが XP サイズのチームで行うプロジェクトの最大規模になります。また、小さいチームが大規模なシステムを効率的に構築できない理由はほかにもあります。例えば、既に作成したコードに関する精通度が、時間が経過すると共に失われることです。(大きいチームで並行して実行できる作業も、小さいチームでは逐次的に行う必要があります。したがって、統合テストやデバッグの段階になると、小さいチームではかなり以前に作成したコードを見直す必要が発生します。

この規模より小さなプロジェクトに注目すると、XP のリリースと RUP の反復には適切な対応が見られます。次の例は、RUP でこれらがどのように計画されるかを示しています。数値は一例ですが、これらの規模のプロジェクトでは妥当な値です。これらの値には、ユーザー・ガイドやインストールと操作に関する情報など、コードと共に納品する RUP 成果物すべてを準備するための作業量と時間が含まれています。

---

[1] e-ビジネスの開発で予想される反復の期間はこれよりも短く、多くの場合は 2 週間から 6 週間の範囲になります。

[2] 通常、方向づけにおける反復はさらに短くなります。また、RUP のフェーズ・モデルはさまざまな形態に対応するので、長い移行フェーズで反復のシーケンスを構想して、2 カ月間隔で顧客にソフトウェアを納品できます。ただし、移行フェーズに移るということは、アーキテクチャーが完全に安定し、予期される変更はほとんど適応可能で、完了を示す調整的なものであるということです。

[3] XP は、顧客への直接的なビジネス価値の提供がその主な機能として注目されています。XP ではアーキテクチャーに対する直接的な考慮はほとんど行われず、機能が追加される際に、ソフトウェアのリファクタリングの間で明らかになります。ソリューションのアーキテクチャーがまだ十分に構築されていない場合は、前提事項 (および随時ソリューション) を無効にし、ローカル・リファクタリングを行えない問題を起こすような機能が追加されると、重大な破損が生じる危険があります。

[4] 参考資料 [Beck00] には、プログラマー 20 人では XP プロジェクトを実行できない可能性が高いが、10 人ならば「間違いなく実行できる」とあります。

[5] COCOMO II は、Barry Boehm 博士によって開発された COCOMO ソフトウェア・コスト見積もりモデルを改善したものです。2,000 SLOC 以下のプロジェクトに対して調整されています。参考資料 [Boehm00] を参照してください。

[6] ファンクション・ポイントは、論理設計と機能仕様だけに基づいた、ソース・コードに依存しないソフトウェア・サイズの測定方法です。この方法では、ユーザーに提供する機能の量が測定されます。この定義は、International Function Point Users Group の Web サイト http://www.ifpug.org/ から引用しました。

XP に適切なプロジェクトの例

例 1 は、5,000 行の新規 Java コードを作成する、非常に小規模なプロジェクトを示しています。7 カ月の期間に、およそ 12 人月が必要です[7]。

例 1

	方向づけ	推敲	作成	移行
スタッフ	1	1.5	2	2
期間 (週)	3	6	18	3
繰り返し回数 (括弧内は、各反復の期間 (週単位) を表します)	1 (3)	1 (6)	3 (6)	1 (3)

例 2 では、10,000 行の新規 Java コードを作成する、小規模なプロジェクトを示します。8 カ月の期間に、およそ 27 人月が必要です。

例 2

	方向づけ	推敲	作成	移行
スタッフ	1.5	2.5	4	3
期間 (週)	4	7	20	4
繰り返し回数 (括弧内は、各反復の期間 (週単位) を表します)	1 (4)	1 (7)	3 (7)	1 (4)

例 3 では、40,000 行の新規 Java コードを作成する、中規模なプロジェクトを示します。15 カ月の期間に、およそ 115 人月が必要です。

例 3

	方向づけ	推敲	作成	移行
スタッフ	3	5	10	8
期間 (週)	6	16	36	6
繰り返し回数 (括弧内は、各反復の期間 (週単位) を表します)	1 (6)	2 (8)	4 (9)	1 (6)

これらの範囲 (かなり広範囲な開発も含まれます) では、RUP の反復と XP のリリースは、目的と期間の両方において非常に類似しています。

XP では適切とは言えない大規模なシステム開発

XP ではなく、RUP で扱うような非常に大規模な開発では、RUP の反復期間はより長くなります。例 4 では、1,500,000 行の新規 Java コードを作成するプロジェクトを示します。45 カ月の期間に、およそ 4,600 人月が必要です[8]。

---

[7] この期間は長すぎると思われるかもしれませんが、スタッフの編成も要求の定義も行われていない (アイデアが出ているだけの状態) 開始状態から、プロジェクトの受け入れと終了までの、ライフ・サイクル全体が扱われている点に注目してください。これも、COCOMO II モデルの出力です。スケジュールをさらに圧縮できますが、コストとリスクは増加します。ただし、表に示すスケジュールに従うと、有効な機能を備えた状態を、プロジェクトの開始後 3 カ月半 (初回の作成反復の終了時) で利用できるようになります。

[8] このサイズのプロジェクトでは、このような設定を行わずに、より小さい (それぞれが XP に適するような) 複数のプロジェクトに分割するべきだと考えることもできます。もちろん、このサイズのプロジェクトは、複数のシステムやサブシステム (非常に大規模なプロジェクトでは、システムのシステム) で構成されますが、これらのサブシステムやシステムは 1 つの製品に統合する必要があります。したがって、ソフトウェアとそれを生成したチーム間で、アーキテクチャーや特にインターフェースについて考慮する必要があります。現在の XP では、これらの問題は扱われません。

例 4

	方向づけ	推敲	作成	移行
スタッフ	35	70	140	100
期間 (週)	20	50	100	20
繰り返し回数 (括弧内は、各反復の期間 (週単位) を表します)	2 (10)	2 (25)	3 (33)	2 (10)

明らかに、この例ではスタッフは 1 つのチームとして編成されません。プロジェクトは、サブシステムを担当する複数のチームで構成されます。これらのサブシステムはさらにサブシステムで構成され、より低いレベルの、XP に適した規模のプロジェクトと同じ規模で計画することもできます。ただし、このプロジェクトは統合されたシステムを表すことを目的としています。したがって、XP の要件を大きく超える規模 (33 週) での反復を、トップレベルで計画する必要があります。これらの反復の期間は、統合されたこの規模のプロジェクトの計画から惰性的に設定されたものです。RUP の反復内の計画は (システムとサブシステム・レベルで存在する) 統合ビルド計画を通して行われ、これらは (このような、非常に大規模なシステムの場合) XP のリリースに近い規模で作成されます。特に、後の推敲と作成フェーズでは、XP の反復に近い最も低いレベル (およそ 2 週間) で作成されます。

したがって、小規模なシステムの例と比較すると、RUP の反復と XP のリリースについてはほぼ同じですが、RUP のビルドと XP の反復に関しては少し異なります。

RUP のフェーズに対応する XP の要素

一見した限り、RUP のフェーズは、十分に定義されていない XP のサイクルの解釈に対応します。XP は、反復を含む (ビジネス・サイクルに調和した) 定期的なリリースの作成に適合するように構築されています (少なくとも、そう説明されています) 。認識としては、成立させる必要のある「プロジェクト」のようなものがあり (参考資料 [Beck01] の「Scoping a Project」を参照)、この時点で作成された「大きな計画」はリリースに分割され、続いて反復に分割されます。[Beck01] では、大きな計画の役割は、プロジェクトに投資することが愚かでないことを判断しやすくすることであると説明されています。

したがって XP では、リリースと反復のサイクルが開始される前に、プロジェクトに関して、実行可能であるかどうか、コスト、実行方法、作成する機能に必要な大まかなアイデアを決定する作業のための時間があります。これは、RUP で方向づけフェーズと呼んでいる期間です。XP ではこれを素早く行うべきだという印象があります。RUP では、このフェーズは内在するリスクに応じてできるだけ慎重に行われます。XP でも、ここで (少人数によって) 実行されなければならない、次のような項目があります。

- 要求の誘導 (「大きなストーリー」の作成)
- 計画
- 顧客との話し合い (予測の設定)
- ビジネス制約の分解

何もない状態から始めて、合意した開発構想 (RUP で「大きなストーリー」と呼ばれるもの) と開発企画書 (プロジェクトを正当化する予算と投資収益) を作成し、最初のソフトウェア開発計画書を作成するのに、数日かかることは簡単にわかります。

XP の場合と同様、RUP では、成功の可能性が十分にあることを確認した後、最初にプロジェクトを進めるための十分な調査と計画を行います。前の例 2 の場合、RUP の計画では、方向づけにおける約 12,000 ドルの (作業量に関する) 投資によって、およそ 250,000 ドルの出費を検証することを提案しています。 値は場合によって変化します。何度も扱われた問題領域ではそれほど多くの投資は必要ないでしょうし、おそらく何もない状態から開始することもないでしょう。一方、ある程度の研究や開発が必要な未知の分野では、最初から多くの費用がかかります。

方向づけに相当する作業の後、XP では最初のリリースの計画に進みます。RUP では、方向づけの後に推敲フェーズが続きます。 推敲の反復で、ソリューションのアーキテクチャーが安定します。対照的に XP では、すべてのリリースで顧客

にビジネス価値を納品できるように、リリース計画を機能優先で行うように提案されています。XP では、インフラストラクチャー計画というものはまったく重要視されていません。現在のリリースで選択された機能をサポートすることで十分であると考えられています。後で必要に応じてインフラストラクチャーが追加され、追加した機能によって以前のインフラストラクチャーの決定が無効になり、システムが破綻した場合は、機能するようにコードがリファクタリングされます[9]。この考え方は、顧客に価値として認められないものを作るために、時間を浪費しないということです。

これに対して、RUP では次の 2 点のように考えます。

- ***RUP のアーキテクチャーの概念は、単なるインフラストラクチャーではありません。***

  次の文は、RUP からの引用です。「(ある時点での) ソフトウェア・システムのアーキテクチャーは、システムの主要なコンポーネントの組織または構造です。その主要なコンポーネントはインターフェースを通じて下位のコンポーネントとインターフェースと相互作用します。このような関係が順々に下位の階層に及びます。

  したがって、アーキテクチャーは、適切なレベルの抽象化において、特定の視点からインフラストラクチャーだけなくソリューション全体を扱います。

- ***RUP の実行可能アーキテクチャーの概念は、顧客にビジネスの価値を提供します。***

  顧客の機能外要求 (また、プロセスにおける主要な機能要求) を満足するソリューションを、早い段階で示すことができます。

  このようにリスクを削減することは (多くの場合、機能外要求にはリスクがあります)、それ自体が顧客に対する大きなビジネス価値となります。

なぜ、プロセスをこのような方法で説明する必要があるのでしょうか。RUP はリスクを排除することがすべてであり、コードのリファクタリングからアーキテクチャーが浮かび上がってくるという XP 手法では、対処できない (一般的には、より大規模でより複雑な) 問題やシステムのクラスがあると考えているからです。1 つには、リファクタリングで局所的な変更を行う (必然的に開発範囲は制限される) ときに、局所的な最適化によって、全体として最適でないソリューションが生成されるというリスクがあります。これは、Kent Beck 氏が参考資料 [Beck00] の「Four Values」の「Courage」で主張していることです。難しい点は、XP ではアーキテクチャー上重要な変更がどのように発想されるかについてはほとんど触れず、それらを適用するには勇気[10]が必要な場合があるとしか説明されていない点です。

また、リファクタリングが困難な変更もあります。例えば、シングル・ユーザー、シングル・マシンのシステムをマルチ・ユーザー、マルチ・プロセッサー・システムに変更するには、多くの部分を再設計する必要があり、それでも完成したシステムには多くの制約が生じます。

RUP がアーキテクチャーを強調しているのは、最初の手法が間違っていて、徹底的な再考が必要となるような場合に対処するためです。小規模なシステムでは、この点はほとんどリスクになりません。リファクタリングでシステム全体を整理でき、最初のアーキテクチャーが間違っていたということはめったにありません。

認められる (規模、新しい技術、未知であること、複雑さ、パフォーマンスに関する別の緊急性、信頼性、安全性などによる) リスクはほとんどなく、ソリューションではよく理解された既存のフレームワークを使用できるため、RUP の (アーキテクチャーに関する問題を処理する) 推敲フェーズは非常に短くなります (重要な機能要求の顕在化、洗練化、説明により関係する場合もあります) 。つまり、RUP のライフ・サイクルは崩壊し、XP のライフ・サイクルに近くなります。問題を XP で適切に処理できた場合、RUP の開発個別定義書は (まったく同じではありませんが) 「軽量級」になります。

---

[9] リファクタリングの結果は局所的な改善でしかありません。リファクタリングでは全体のフレームワークは扱いません。フレームワーク全体で問題を解決している場合、リファクタリングでは正しいソリューションは生成されません。唯一のソリューションは、根本的な変更を行うか、最初からやり直すことです。この場合、予算とスケジュールにリスクがあることは明らかです。

[10] 勇気が美点であり、おそらく勇気ある人は「優れた人物」であることについては議論の余地はないでしょう。ただし、最も勇敢な人物でも、組織的な作業や、難しい問題を解決できるフレームワークを必要とする場合があります。

RUP の作成フェーズは、XP の一連のリリースと同じです。RUP に従う場合は、各作成フェーズの反復における結果を顧客に納品する必要があり、XP と同じ効果があります。XP では、各リリースが既に顧客に納品されているため、RUP の移行フェーズに相当するものはありません。XP プロジェクトの最後には、参考資料 [Beck00] でプロジェクトの終わりと呼ばれている作業があり (後出の「XP のライフ・サイクル」を参照)、ここで同じようなプロジェクトの終了アクティビティーが行われます。

XP のフェーズに対応する RUP の要素

XP のフェーズ (XP のリリースと反復の両方に適用される) は、探索、コミット、舵取りです。リリース・レベルでは、これらは RUP のプロジェクト管理の作業分野における、特定のタスクの集合 (RUP でのアクティビティー) に対応します。つまり、次の反復計画 (探索とコミットに対応)、プロジェクトの監視と管理 (舵取りに対応) です。これについて、次のセクションで説明します。

XP のフェーズ

参考資料 [Beck00] の第 21 章では、理想的な XP プロジェクトのライフ・サイクルが説明されています。この章は、**「探索 (exploration) 」**、**「計画 (planning) 」**、**「最初のリリースへの反復 (iterations to first release) 」**、**「稼働への移行 (productionizing) 」**、**「メンテナンス (maintenance) 」**、**「終わり (death) 」**という見出しで構成されています。これらはトップレベルのフェーズを表しますが、正確ではありません。XP プロジェクトは、最初の探索フェーズで境界を定められ、プロジェクトが終了したときに「終わり」ます。しかし、主にリズムを取るのはリリースであり、各リリースに探索フェーズがあります。したがって、プロジェクトを一括する (そして、最初のリリースに導く) 探索フェーズは特殊なケースであり、続行するかどうかを決定する、通過すべき最初の関門 (参考資料 [Beck01] の「Scoping a Project」を参照) を含んでいます。XP のリリースと反復には、どちらにも探索、コミット、舵取りの 3 つのフェーズがあります。「メンテナンス」は最初のリリースの後、実際に XP プロジェクトの性質を決定します。

XP のライフ・サイクル

15 カ月の期間で 7 回のリリースを行うプロジェクトでは、XP のライフ・サイクルは全体として図 1 のようになります。

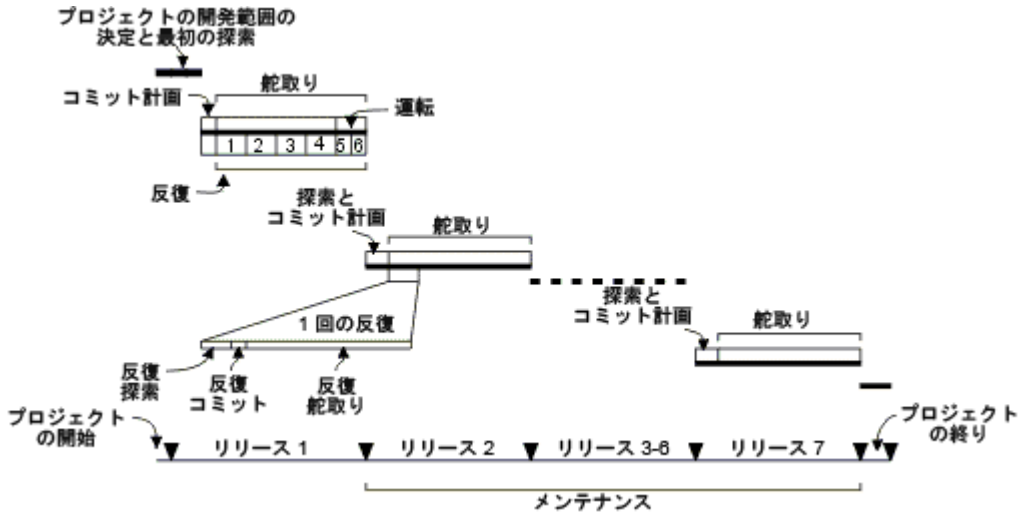

図 1

2 回目以降のリリースは、およそ 2 カ月で行います (1 回目は 3 カ月) 。また、各反復は 2 週間で行いますが、反復の速度が加速する後の方のリリース段階 (参考資料 [Beck00] では「稼働への移行 (productionizing) 」と呼ばれている) では異なります。

計画のメトリックスから考えて、これは意味があるでしょうか。この例では、XP で規定されたガイドラインに沿って開発を進めます。つまり、リリースの期間を約 2 カ月に設定し、最初のリリースは 2 ～ 6 カ月の間に行います。また、チームのサイズは 10 人以下になるようにします。このプロジェクトが前に示した例 3 と同様である場合は、合計で 40,000 行の Java コードまたは 800 のファンクション・ポイント (COCOMO II モデルの変換係数を使用した場合) が納品されます。これをすべてのリリースで均等に分割すると、各リリースで納品されるファンクション・ポイントはおよそ 115 になります。

この場合、最初のリリースは信用できません。何もない (割り当てられたメンバーは 1 人、要求が獲得されていない、範囲の設定も行われていない) 状態から開始して、3 カ月で製品品質のものを顧客に納品するのは困難です。12 ファンクション・ポイント/人月くらいの高い生産性 (David Consulting Group の工業データに基づきます。http://davidconsultinggroup.com/indata.htm を参照) があるとしても、ぜいたくなスタッフを自由に編成することは不可能です。ここで扱う問題領域 (115 ファンクション・ポイント) は非常に小さく、大規模なチームで担当できるように小さく分割することはできません。

ただし、これを可能であると仮定した場合は、平均して 4 人規模のチームが必要で、プロジェクトはリリース 1 を 7 人スタッフのチームで終了します。プロジェクトの期間中にスタッフ・メンバーをもう 1 人追加すると、プロジェクトの残りの期間は、XP のノーマル・モード (メンテナンス) で、8 人のチームで処理できます。これは XP でちょうどよい規模です。納品のサイズが大きくなると、生産性はほぼ間違いなく低下します。また、リリース間隔 (2 カ月) を短くすると、メンバーの増員分が相殺されます。各リリースでは、追加する機能が最初のリリースとほぼ同じ量になるように納品を行います。プロジェクト全体の作業量は 108 人月となり、例 3 の場合よりもわずかに小さくなります。

既定の RUP ライフ・サイクル

一方、同じ規模の RUP プロジェクトにおける既定のライフ・サイクルは次のようになります。

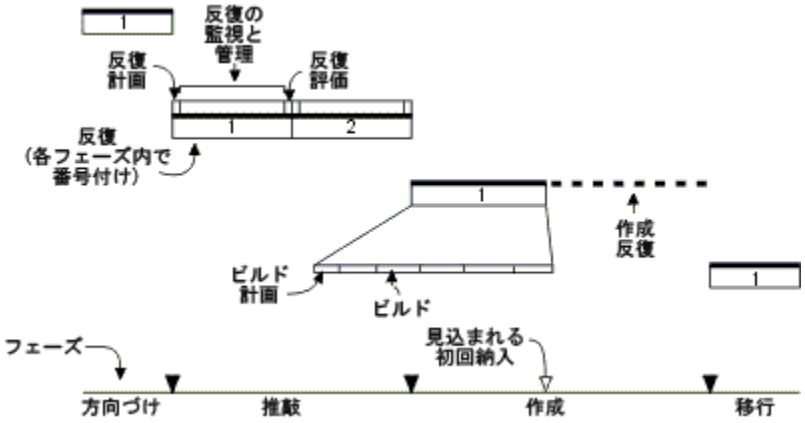

図 2

この場合も、計画は例 3 に基づきます。RUP の反復には、各フェーズ内で番号が付けられます。既定の RUP ライフ・サイクルを使用すると、最も早く製品品質に近いものを顧客に納品できるのは、おそらく作成フェーズにおける最初の反復の終わりです。これは、プロジェクトを開始してから 7 カ月後です。ただし、この時点で、すべての機能の 4 分の 1、つまり 200 ファンクション・ポイントを納品できます (XP の場合は、3 カ月後に 115 ファンクション・ポイントです) 。通常、顧客への最初の納品は作成フェーズの終わりに行われます。この場合はプロジェクトを開始してから 13 カ月目となり、この時点で実質的にすべての機能が提示されます。顧客がここで製品を初めて目にするという意味ではありません。 RUP では、顧客は反復の評価を求められ、テスト対象にある開発中の製品を確認することができます。

RUP と XP にはなぜこの相違があるのでしょうか。この既定のケースでは、ソリューションのアーキテクチャーに (例えば、前例がなかったり、新しい技術であったり、機能外要求が面倒であるなどの理由で) 多くのリスクがあり、顧客が要求する

機能を作成する前に、推敲で 2 回の反復を行ってこれを安定させるという前提事項があるためです。これは、推敲の終わりまでに、アーキテクチャー上の重要な要件だけは探索済み (そして、おそらく実装済み) となることを意味します。

XP ではこれを行いません。XP では、各リリースで完全なソリューションを納品し続けます。 仮定として、リリース間でアーキテクチャーが破綻することなく、発生する障害はすべてリファクタリングで修復できると考えられています。これにより、XP を適用できる範囲が、既知の機能を備えた既存のアーキテクチャー上に作成できるシステムに制限されると考えられます。

実際の RUP ライフ・サイクル

これらのシステムでは、実際の RUP ライフ・サイクルは少し違っています。推敲フェーズはより短く、おそらく方向づけフェーズもさらに短くなります。推敲の 1 つの反復を作成の反復と入れ替えると、最初の作成の反復が 2 カ月繰り上がり、7 カ月を 5 カ月にできます。

納品は既定の例の 200 ファンクション・ポイントよりも少なくなりますが (最初の作成の反復で作業するスタッフの人数は少なくなる)、XP のスケジュールの軽快さに近くなり、リラックスしてリスクを大きく軽減することができます。極端に言えば、RUP ベースの手法は、小規模システム (例えば、XP の例の 115 ファンクション・ポイント) の最初の納品近くで確立され、RUP の多くの進化サイクル (方向づけ、推敲、作成、移行シーケンス) を経て、XP に似たメンテナンス・モードでシステムの残りを納品すると考えることができます。

また 6 カ月の辺りでは、最初の納品が XP で推奨される開発期間の上限になりそうですが、RUP では、XP で完了済みとされているリリースの配置などについて計画 (そして時間と作業量を配分) するように要求されます。

これらの観察結果から、XP 向きのプロジェクトの特性について別の制限を推論できます。XP では、顧客と開発チームの間に親密な関係を築くこと、顧客がフルタイムでオンサイトに常駐すること、顧客がストーリーを作成してリリースを定義することが、明示的に要求されます。XP では、顧客は実際にチームにおいてロールを持ちます。ロールを演じる人物またはグループは、別の企業に所属するかどうかに関係なく、ソフトウェアを開発しているメンバーの誰にでも意見を述べる権利を持ちます。本質的に、彼らには要求があります。

このような関係は、外部クライアントと正式に契約した開発よりも、社内で行われる開発でよく見られます。これらの環境では、2 カ月のリリース・サイクルがより受け入れやすいでしょう。企業間で 2 カ月ごとに新しい (障害の修正だけでなく、新しい基本要件も備えた) リリースを行うと、物流費が受け入れられない可能性が高くなります。

これらの緩和 (小さいチーム、既に確立されたアーキテクチャー、形式的でない社内スタイルでの開発) により、適切に調整された RUP サイクルから XP 開発と同様の効果が得られます。最初のリリース期間は XP の最適な場合よりも少し長くなりますが、リスクは小さくなります。

逆の場合 (大きなチーム、前例のないアーキテクチャー、納入と配置に制約のある正式に契約された開発)、XP をどのように調整できるかを考えるのは困難です。また参考資料でも、その可能性について説明されていないので、ここで説明することはできません。一方、RUP は、このようなプロジェクトに伴う手続きや形式性にうまく対処できるように設計されています。おそらく XP のような手法 (ペア・プログラミングやリファクタリングなどの XP 技術を使用する手法) は、一度アーキテクチャーが安定すれば、より大規模な RUP プロジェクトに埋め込むことができます。

成果物

RUP で説明されている成果物は 100 以上になります。このことは、小規模なプロジェクトに対して過度の官僚的オーバーヘッドを強要しているという批判の原因にもなっています。この考え方は、次の 4 つの点で正しくありません。

- プロジェクトで、すべての成果物を生成する必要はありません。成果物を選択してカスタマイズすることは、プロセスにおいて必須な作業です。RUP は何を省略し、何をカスタマイズできるかについてのガイドラインを提供します。
- 成果物 (RUP では、成果物の「*説明*」) は仕様の実体です。つまり、タスクで生成される情報を示しています。そして、これを体系的に行うために、成果物の抽象的な*形式*を説明している場合もあります。RUP の成果物がモデル、モデル要素、文書として記述されているのは (RUP 成果物のおよそ半分は文書に分類される)、特定の「実現」を示すためではありません。

  UML モデルは、専用のツール (Rational Rose など) を使用して獲得し、表現できます。また、簡単なグラフィック・ツールを使用して作成された図や、ホワイトボード上でのスケッチとしても表現できます。これらも、RUP 用語での成果物として数えられます (ホワイトボードでは消えてしまうというリスクもありますが)。

  「文書」という用語の歴史的な性質から、これを段落ごとに見出しを付けた紙ベースの独立したワーク・プロダクトで、完成させるにはすべてのページをテキスト (そして、おそらくは図) で埋める必要があると考えてしまう場合もあります。しかし、これは RUP 文書を実現したものの 1 つでしかありません。

  RUP における文書とは、テキストと図で構成された情報のセットです。情報を利用しやすいように、RUP ではこの情報の形式を定めており、そのための便利で体系的な方法として、考慮すべき項目や問題を列挙し説明するテンプレートを提示しています。もちろん、これらのテンプレートを直接使用したり、その形式を使用したりして、(一般的な意味での) 文書形式、つまり電子的あるいは紙の成果物を実現することが可能です。多くのプロジェクトでこのような方法が望まれることがわかったので、ほぼそのまま使用できるテンプレートのセットを用意することにしました。しかし、これは必須ではありません。テンプレートに用意されているすべてが、特定のプロジェクトに関係あるとはかぎりません。RUP では、成果物の適切な実現方法 (と納入メカニズム) を選択できるように、プロジェクトに自由度が与えられています。重要なことは、そこに含まれる情報です。
- プロセスのすべての見込まれるワーク・プロダクトは、プロセス・エンジニアやプロジェクト管理者によって検討できるよう、明確に特定される必要があると考えます。これにより、それらを除外するかカスタマイズするかの決定を、その結果を十分に理解した上で意識的に行うことができます。このような方法で成果物リストを完成させ、明確にすることで、プロセスの構成に客観性が得られることと思います。また、経験を積んだ管理者が当然としていること (述べられていないが、初心者の見落としがちなこと) に注目させることで、経験の少ないプロジェクト管理者をサポートします。明確な文書セットがあれば、納品する内容や形式について、顧客とプロジェクト管理者が早期に同意する手助けになり、プロジェクトの中止を決定するという望ましくない議論を避けることができます。
- RUP では、複合物である成果物がいくつか説明されており、そのコンポーネントも成果物として示されています。これにより、必要な場合に、タスクの入出力に関してきめの細かい説明が得られます。説明に関しては、設計モデルなどの構成要素で終了したり、クラスを成果物として独立させずに、設計モデルの一部とすることもできます。これらを成果物として識別することにより、その使用に関する正確な説明が得られます。同時に、全体の成果物の数が増加する代わりに、プロセス・メタモデルに従うことになります。

XP には、大量の成果物が生じるという批判はありませんが、これは次の 2 つの理由で単純化されているためです。

- XP は、特定の種類の開発に対して既にカスタマイズされています。RUP の作業分野のサブセットを特定の規模で扱うため、要求される成果物の量は少なくなると予測されます。
- XP は、ユーザー・ストーリーとコードの重要性を強調しています。ほかの作業成果については、プロセスの説明で簡単に述べられているだけです。したがって、成果物の数は一見するよりも多くなります。

XP の手法を考えると、ここで使用している 3 冊の (プロセスに関する最新の) 参考資料の索引に、「成果物」や「作業成果」という用語が示されていないのは驚くことではないでしょう。ただし、テキスト全体から、成果物を表す用語を見つけ出すのは難しくありません。次にその例を示します。

- ストーリー
- 制約
- タスク
- 技術的なタスク
- 受け入れテスト
- ソフトウェア — コード
- リリース
- メタファ
- 設計 — CRC、UML スケッチ
- 設計文書 — プロジェクトの終わりに生成
- コーディング規約
- 単体テスト
- ワークスペース (開発とその他の設備)
- リリース計画
- 反復計画
- ミーティングでのレポートとメモ
- 全体計画 — 予算
- 進捗レポート
- ストーリーの見積もり
- タスクの見積もり
- 障害 (と関連するデータ)
- 会話からの追加文書
- サポート文書
- テスト・データ
- フレームワーク・テスト・ツール
- コード管理ツール
- テスト結果
- スパイク (ソリューション)
- タスクの作業時間記録
- メトリックス・データ
    - リソース
    - 開発範囲
    - 品質
    - 時間
- ほかのメトリックス
- 追跡結果

30 の成果物があり、いくつかは複合の成果物です。また、このリストに挙がっていないものもあります。XP の書籍では、これらについて多くは説明されていません。また、書籍のレベルで説明されることはないと予想されます。ただし、プロジェクトでこれらを実現する必要が生じる場合、より詳しい内容や形式が必要になります。確かにプロジェクトの進行中にこれらの作業を行うことも可能ですが、実際の作業から時間を割くことになります。一方、RUP では先行的な指示が出されているため、プロジェクトの時間を消費することはありません。

XP ですべての RUP 成果物を必要としない理由

理由の 1 つは、XP が RUP ほどの開発範囲を持たないことです。XP では、ビジネス・ニーズがどのように発生したか、それをどのようにモデリング、獲得、説明するかについては注目しません。顧客である XP チーム・メンバーは、XP 開発チームに対して、抽出した要求をストーリーとして示します。開発チームは、ビジネス価値の判定者でもあります。ストーリーを、どのような方法を使ってその形式で表現 (または表現可能に) したかは、XP では重要ではありません。例えば、RUP がビジネス・モデリングの作業分野で説明している内容は、XP の開発範囲にはありません (RUP のビジネス・モデリングには、最大で 14 の成果物があります)。XP では、顧客に定期的なリリースを行うプロセスが示されています。これらのリリースを配置する方法は開発時に考慮されないため、RUP の配置の作業分野 (最大 9 の成果物) のほとんどは XP の開発範囲に入りません。したがって、XP に適した小さいプロジェクトでは、RUP をカスタマイズする際に最大で 23 の成果物を省略することになります。

別の理由は、XP では要求と設計の獲得が簡単にできると主張されていることです。要求はユーザー・ストーリー (分割が必要な場合もある) として把握され、タスクに分解されます。これらは本質的に、システムのメタファを想定して行った設計です。このような考え方は、すべてのシステムで可能でしょうか。明らかに無理です。では、一部のシステムでは可能でしょうか。確かに可能です。 また、XP 自身でも、この考え方ですべてのシステムを処理できるとは主張していません。おそらく XP の作成者たちは、これらの説明を冗談で行ったのでしょう。

大規模でより複雑なシステムに要求される振る舞いは、ユースケースなどの体系的な手法を使用しなければ、表現が非常に困難になります。また、顧客と開発者の間で交わされた会話や、本質的に間違いを起こす人間の記憶に頼って、複雑なユーザー・ストーリーを矛盾なく作り上げるのは不可能です。また、大規模なシステムの複雑な振る舞いを「実現する」構造の開発は、そのシステムのアーキテクチャーのさまざまな抽象ビューを作成して推測する能力によって支援されます。RUP が要求の作業分野で説明している成果物 (最大で 14) と、分析/設計の作業分野で説明している成果物 (最大 17) は、これらのシステムの多様性、規模、複雑さに対応できます。

RUP の小規模プロジェクト向け手順では、成果物 (要求と分析/設計における成果物) の数は 7 つに削減されています。このうちのいくつかは、単に複合の成果物を指しているだけです。これはごまかしではなく、XP と同じ方法で成果物を扱っています。例えば、小規模なプロジェクトで実現しやすい方法であるため、RUP では設計モデルが次のように説明されています。

> 「設計モデルは、ブレーンストーミング・セッションを繰り返すことで展開すると考えられます。このセッションでは、開発者は CRC カードと手書きの図を使用して、設計を調査して獲得します。設計モデルは、開発者が有効であると考えるときにだけメンテナンスされます。実装との一貫性が維持されませんが、参考用に資料として保存されます。」

最後に、必要な場合や契約で申し合わせがある場合、RUP ではプロジェクト管理がより形式的 (かつ「儀礼的」) になることが認められます。RUP のプロジェクト管理の成果物 (プロジェクト管理の作業分野では 15 の成果物がある) の多くは、複合成果物を構成する部品です。 これらは、プロジェクトが形式的な場合にだけ、独立した文書として実現する必要があります。例えば、RUP のソフトウェア開発計画書には、リスク管理計画書や製品検収計画書などの成果物が含まれています。小規模で形式的でないプロジェクトでは、これらの計画書で扱われる内容を、ソフトウェア開発計画書で 1 段落から 2 段落にすることもできます。RUP による小規模プロジェクトの手順では、プロジェクト管理の成果物は 6 つになります。

小規模プロジェクトでの成果物の比較

小規模プロジェクト用に RUP を構成し、成果物の要求を調整すると、どうなるでしょうか。RUP の小規模プロジェクトの手順を確認すると、成果物の数は 30 (配置を除外すると 26) になります。 それほど多くの成果物ではありません。RUP で

は、XP であいまいにされている部分が明確に記述されています。また、何が必要かを決定でき、選択を行う方法についてのガイドラインが提供されます。詳細は異なりますが、XP で楽に処理できる小規模プロジェクトに対する RUP の成果物数は、XP と同じ程度になります。

XP 成果物と RUP 成果物の対応

XP	RUP による小規模プロジェクトの手順
ストーリー 会話からの追加文書	開発構想書 用語集 ユースケース・モデル
制約	補足仕様書
受け入れテスト 単体テスト テスト・データ テスト結果	テスト・モデル
ソフトウェア — コード	実装モデル
リリース	製品 リリース・ノート
メタファ	ソフトウェア・アーキテクチャー説明書
設計 — CRC、UML スケッチ タスク 技術的なタスク 設計文書 — プロジェクトの終わりに生成 サポート文書	設計モデル
コーディング規約	設計ガイドライン プログラミング・ガイドライン
ワークスペース (開発とその他の設備) フレームワーク・テスト・ツール	 ツール
リリース計画 ストーリーの見積もり タスクの見積もり 反復計画	ソフトウェア開発計画書 反復計画書
全体計画 — 予算	開発企画書 リスク・リスト 製品受け取り計画書
進捗レポート タスクの作業時間記録 メトリックス・データ (リソース、開発範囲、品質、時間) ほかのメトリックス 追跡結果 ミーティングでのレポートとメモ	ステータス評価書
障害と関連データ	変更依頼
コード管理ツール	構成管理計画書 プロジェクト・リポジトリー ワークスペース
スパイク	プロトタイプ
	開発個別定義書 プロジェクト固有のテンプレート

ガイドライン

成果物に関連して、RUP では作業ガイドラインが提供されます。本質的に、これらは成果物に関する詳細な情報 (別の説明、ほかの成果物との関係、内容、使用方法) です。これらのガイドラインは印刷すると数百ページにもなりますが、実際には、必要な成果物に関する部分だけを読むだけでかまいません。

XP の参考資料にも、実践方法と成果物の両方に関する詳しいガイダンスが説明されていますが、関係を正確にモデリングする試み (または必要性) は記されていません。ただし、XP に関する資料は薄いものではありません。現在利用できる 3 冊の参考資料は合計で約 600 ページの量があり、これから出版される別の 2 冊の書籍で、これに 700 ページ程度が追加されます。参考資料 [Fowler99] には、リファクタリングに関する 400 ページ以上の記述があります。

さらに、XP はいくつかの Web サイトでも取り上げられています。説明されている内容は、さまざまな観点からの XP の検証と体験集の追加など、部分的に重複しているようです。これらは RUP と XP の別な相違点を示しています。つまり、RUP は製品であり、XP は製品でないということです。入門として書籍を選ぶのはよい方法ですが、XP の情報は別の方法でも得られます。XP を最も早く身に付ける方法は、商業的なトレーニングを思想的リーダーから受けることです[11]。

[11] RUP は「商標」として説明されることがありますが、誰でもこれを購入して、説明されているアイデアを使用できます。また、著作権とライセンス契約に違反しない限り、アイデアを追加したり、開発組織やプロジェクトに関係のない部分を削除することもできます。書籍として記述された XP に関する説明も、著者と出版社が所有する知的所有物です。唯一の違いは、それぞれの情報から十分に理解できる範囲です。製品である RUP は完全であることを意図していますが、現在の XP の書籍には (集中的なトレーニングやコンサルティングによる) いくらかの補足が必要です

タスク

RUP では、「タスク」という用語が、入力成果物を使用して変換し、新規または変更された出力成果物を生成する、ロールによって実行される作業として正式に定義されています。RUP では、これらのタスクを列挙し、プロジェクト内の「作業分野」または主要な「関係領域」に従って分類しています。これらの作業分野を次に示します。

- ビジネス・モデリング
- 要求
- 分析/設計
- 実装
- テスト
- 配置
- 構成と変更管理
- プロジェクト管理
- 環境

タスクは、生成または消費する成果物を通して、時間に関連付けられます。つまり、タスクは入力が利用できるようになると (また、適切な開発状態になると) 論理的に開始できます。このことは、成果物の状態で許容される場合、生産/消費タスクのペアは時間的に重複可能であり、厳密に逐次的に実行される必要がないことを意味します。

RUP のタスクの目的は、成果物を生産する知的なプロセスの不明瞭な部分をなくすことです。つまり、生成方法に関する詳細なガイダンスを提供することです。タスクは、プロジェクト管理者の計画にも使用できます。ライフ・サイクル、成果物、タスクに関して説明されているように、RUP を通じて生成されたものは「最善の実践原則」です。スケジュールと予算を予測可能にして、品質の高いソフトウェアを作成するための、ソフトウェア開発の原則です。

RUP は、タスクとそれに関連する成果物を通じて、これらの最善の実践原則をサポートして実現します。これが、RUP を通じた実行におけるテーマです。XP でも「実践原則」という概念が使用されますが、見たところ、RUP の最善の実践原則との正確な対応はありません。

XP では、ソフトウェア開発に関する魅力あるシンプルな考え方が提供されています (参考資料 [Beck00] の第 9 章を参照)。補助的な実践原則に従って有効化され構築される、次の 4 つの基本的なタスクが示されています。

- コーディング
- テスト
- リスニング
- 設計

XP のタスクの開発範囲は、実際には RUP のタスクよりも作業分野に似ています。XP プロジェクトで行われる (コーディング、テスト、リスニング、設計以外の) タスクのほとんどは、その実践原則の詳細化と適用から生じます。

RUP のタスクに相当する XP の要素

RUP のタスクに相当する XP の要素は存在します。ただし、XP の「タスク」は正式には特定あるいは説明されていません。例えば、参考資料 [Jeffries01] の「Chapter 4 User Stories」では、「要求をストーリーに定義して、カードに記述します」と説明されています。 この章では、プロセスの説明と、どのようなユーザー・ストーリーがあり、それをどのように (そして誰が) 生成するかについてのガイダンスが示されています。そして、この章の各項目では (成果物に注目する項目と、アクティビティーに注目している項目が混在している)、「行うこと」と「作成するもの」が、規定と詳細の程度を変化させながら説明されています。

RUP の明らかな規定は、タスクの体系的な取り扱いとその入出力が完全で、非常に形式性が高いことに基づきます。XP でも規定は不足していませんが、おそらく「軽量」であろうとするために、形式性や詳細さが省略されています。プロジェクトで XP を実装する場合は、RUP で示されている詳細事項を追加する必要があります。特異性の不足は利点でも欠点でもありませんが、XP において詳細な情報が不足することと、シンプルであることを混同すべきではありません。ある時点で、プロジェクトのメンバーは何を行うかを確認する必要があり、その時点で詳細情報が必要になります。

作業分野

RUP における作業分野とは、特定の成果物セットを生成するタスク (と関連する概念) の集合です。これは、ソフトウェア開発における重要な側面や問題を表しています。この用語の使用法は、「作業分野 (discipline) 」の辞書的な定義、つまり学問分野という意味にうまく対応しています。

前述したように、RUP の作業分野には、ビジネス・モデリング、要求、分析/設計、実装、テスト、配置、構成と変更管理、プロジェクト管理、環境があります。これは、開発、配置、運用、サポート、販売、マーケティング、大規模なソフトウェア・システムに関するその他の処理を行うメンバーを編成するときに、組織またはビジネスで行われるすべての側面を扱っているわけではありません。例えば、現在のところ、RUP ではシステム・エンジニアリングを扱っていません。また、ISO 15504 などの国際的なソフトウェア・プロセス規格 (例えば、ソフトウェアの取得や人的リソースの管理に関する側面を示す) の要求も扱いません。これは RUP の仕様です。こうした側面は重要ですが、RUP のエンジニアリング対象からは外れています。

XP は、さらに自身を明示的に制限しています。XP にはコーディング、テスト、リスニング、設計 (これらは、RUP の作業分野に近いものとして説明した) という、基本となる 4 つのアクティビティーがあり、これらは実践原則のセットを使用して実行されます。実践原則を実行するには、RUP のその他の作業分野に対応する、別のアクティビティーを実行する必要があります。XP の実践原則を参考資料 [Beck00] から引用します。

- 計画ゲーム — ビジネスの優先度と技術的な見積もりから、次のリリースの開発範囲を早急に決定します。現実と計画が食い違ったら、計画を更新します。
- 短期リリース — シンプルなシステムを素早く生産に投入し、その後、非常に短いサイクルで新しいバージョンをリリースします。
- メタファ — システム全体がどのように動作するかを示すシンプルなたとえ話を共有することで、すべての開発を導きます。
- シンプルな設計 — システムはどの時点でも可能な限りシンプルに設計します。不必要に複雑な部分は、見つかった時点で取り除きます。
- テスト — プログラマーは、継続的に単体テストを作成します。開発を続行するには、これを完全に実行する必要があります。顧客は、機能が完成したことを確認するテストを作成します。
- リファクタリング — プログラマーは重複の削除、コミュニケーションの改善、単純化、柔軟性の追加を行い、システムの動作を変更することなく、これを再構築します。
- ペア・プログラミング — すべての製品コードは、2 人のプログラマーが 1 台のコンピューターで作成します。
- 共同所有権 — すべてのメンバーは、いつでもシステムのどこにあるコードでも変更できます。
- 継続的な統合 — タスクが完了するごとに、1 日に何度でもシステムの統合とビルドを繰り返します。
- 週 40 時間労働 — 原則として、週に 40 時間を超えては働きません。2 週続けて労働時間が超過しないようにします。
- オンサイト顧客 — 実際のユーザーをチームに加えて、いつでも質問の回答が得られるようにします。
- コーディング標準 — プログラマーはコードを作成するときに、コードを通じてコミュニケーションを強調する規則に従います。

「シンプルな設計」について注意することは、「余分な複雑さ」は主観的であり、定義するのが困難であることです。これらは経験を積んだ開発者の視点で考えられています。

例えば、「計画ゲーム」の結果として実行されるアクティビティーは、主に RUP のプロジェクト管理の作業分野に対応します。ただし、ビジネス・モデリングなどのいくつかの項目は、XP の開発範囲を超えています。要求を導き出す作業は、多くの場合、XP の開発範囲ではありません。 要求は、顧客 (オンサイト顧客) が (ストーリーの形で) 定義して提供します。リ

リースしたソフトウェアの配置も、XP の開発範囲外です。XP では、扱う開発の規模や種類によって、RUP で詳しく扱われる「環境」と「構成と変更管理」の作業分野の問題を、非常に簡単に扱うことができます。

RUP における XP 実践原則の使用法

XP と RUP で重複する作業分野では、XP で説明した実践原則のいくつかを RUP にも適用できます (一部は既に適用されています) 。例として以下のものをあげることができます。

- *ペア・プログラミング。*XP では、製品コードの作成は、1 台のワークステーションで 2 人のプログラマーが作業する必要があります。これはコード品質に関して有利な方法であり、ペア・プログラミングのスキルを身に付ければより楽しい作業になると説明されています。RUP では、コード生成の方法について、このようなきめ細かいレベルでは説明されていません。また、RUP に基づくプロセスでも、ペア・プログラミングを使用することは明らかに可能です。この実践原則に関する情報や、テスト先行設計とリファクタリングに関する情報 (以下を参照) は、RUP のホワイト・ペーパーとして利用できるようになっています。ただし、RUP でこの実践原則を使用することは必要条件ではありません。また、この実践原則を要求する必要があるとも思いません。

  工業的な規模では、ペア・プログラミングの利点を証明する資料は不十分で、逸話的です。参考資料[Nosek98] と [Williams00] の研究で、ペアと個人 (単独での作業) について比較されていますが、優秀なチームはこのように作業していません。チーム環境では、開放的コミュニケーションの精神によって、個人は同僚 (およびチーム・リーダーや管理者) に自由に質問したり、定義されたプロセスに従って作業します。このような場合、ペア・プログラミングの (総ライフ・サイクル・コストに対する効果に関して) 利点を認めるのは難しいと推測されます。うまく機能しているチームでは、メンバーは強制されることなく自然に集まって議論し、問題を解決します。

  参考資料 [Beck00] では、次のように、人やプロセスが批判的に見られています。「ペア・プログラミングの別の効果的な特徴は、一部の実践原則はペア・プログラミングがなければ機能しないということです。ストレスを受けると、人は元の習慣に戻ってしまいます。テストの作成を省略し、リファクタリングは延期されます。」これらの作業は、よりよい結果を導く自然な実践原則と考えられます。どうして実行しないことがあるのでしょう。

  よいプロセスを例示することは意味のあることですが、「微細な」レベルで強制されなければならないという提案は、適切ではありません。ただし、状況によっては、ペアを組むそれぞれが相手を助けることができ、ペアで作業することに明らかな利点がある場合があります。例えば、以下のような場合です。

  - チームが編成されたばかりで、メンバーが互いのことをまだよく知らないとき
  - 新しい技術について経験の浅いチーム
  - 経験を積んだスタッフと初心者が混在するチーム

- *テスト先行設計とリファクタリング。*これらは、RUP の実装の作業分野に適用できる優れた技術です。特に、テスト先行設計は、詳細なレベルで要求を明らかにするための優秀な方法です。
- *オンサイト顧客。*顧客をチームのメンバーとしてオンサイトに配置することは、RUP のタスクの多くにおいて、効率の面で大きな利点となります。顧客をスタッフの一員とすることで、中間的な納入物 (特に文書) の多くを削減することができます。

  XP は、コード以外のものを獲得することを提案していません。コミュニケーションの手段としては、会話が強調されています。この場合、成功は継続性と精通の度合いに依存します。したがって、システムを移行する必要がある場合は、たとえ小さなシステムでも、別のものを作成する必要があります。

  XP では最終的に、これをプロジェクトの終わりで生成する設計文書など、後から追加する作業結果として認めています。実際に XP では、文書やその他の成果物の作成は禁止されていません (説明されていないだけです)。本当に必要なものや使用するものだけを作成するように示されています。RUP も、この点については同意しています。ただし、継続性や精通の度合いが理想的でない場合に必要となるものについても説明しています。

- *コーディング標準。*RUP には、ほとんど常に必須であるとされる成果物 (プログラミング・ガイドライン) があります (これは、カスタマイズを行う主な原因である、多くのプロジェクト・リスク・プロファイルによる)。

- *継続的な統合。*RUP はこれを (1 反復内の) サブシステムとシステム・レベルのビルドでサポートします。単体テストを受けたコンポーネントは、新しいシステム・コンテキストで統合されテストされます。

適用の難しい XP の実践原則

一部の実践原則には適応性はありません (XP でも主張されていません)。これを前提に、RUP での使用を考えてみます。例として以下のものをあげることができます。

- *共同所有権。*小規模なシステムや大規模なシステムのサブシステムを担当する小さなチームのメンバーを、すべてのコードに習熟させるのは有益です。すべてのチーム・メンバーに、どこでも変更できる権限を与えるかどうかは、システムまたはサブシステムの性質と複雑さによります。一般的には、コード・セグメントを現在作業している個人 (またはペア) に修正させる方が早く (かつ安全) です。

  特にアルゴリズムが複雑である場合は、最適なコードに対する精通の度合いも、時間の経過と共に失われます。

- *リファクタリング。*大規模なシステムでは、頻繁にリファクタリングを行っても、アーキテクチャーの不足を補うことにはなりません。参考資料 [Beck00] では、次のように説明されています。XP の設計戦略は、丘を登る方法 (アルゴリズム) に似ています。シンプルな設計を行い、これを少し複雑にします。今度はこれを少しシンプルにし、さらにまた少し複雑にします。このような丘を登るアルゴリズムの問題点は、局所解に達することです。 この場合、小さな変更では状況を改善することはできず、大きな変更が必要となります。

  RUP では、「大きな丘」に対する展望と取り組み方がアーキテクチャーによって提供され、大規模で複雑なシステムを制御しやすくなります。

- *メタファ。*より大規模で複雑なシステムの場合、メタファによるアーキテクチャーは十分ではありません。RUP では、アーキテクチャーに対してより高品質で説明的なフレームワークが提供されます。これは、参考資料 [Beck00] で説明されている「大げさな箱と大げさな接続線でまとめたもの」ではありません。
- *短期リリース。*顧客が新しいリリースを受け入れて配置できるペースは、多くの要因で変化します。通常、その 1 つにシステムのサイズがありますが、これはビジネスへの影響に関連付けられます。システムによっては、2 カ月のサイクルでは短すぎる場合があります。配置のロジスティクスにより、禁止されることがあります。

一見して RUP で利用できる、またはその可能性があるいくつかの実践原則は、多くの場合、適用する際に少しの推敲や注意が必要です。

- *シンプルな設計。*XP は非常に機能的に駆動されます。ユーザー・ストーリーが選択され、タスクに分割され、そして実装されます。参考資料 [Beck00] では、次のように説明されています。「正しいソフトウェア設計は、次のようなものなります。

  1. すべてのテストをパスする。
  2. 重複したロジックがない。(略)
  3. プログラマーにとって重要な意図がすべて述べられている。
  4. クラスとメソッドの数が最小である。

  XP では、今必要ないものを追加するという作業は、有効であると考えられていません。ここに、RUP で機能外と呼ばれている要求に関して、局所解の場合と同様の問題があります。これらの要求は、顧客にビジネス価値をもたらしますが、ストーリーとして表現するのは困難です。XP で制約と呼ばれているものは、このカテゴリーに分類されます。

  RUP は、要求されている以上のものを推測して設計することを提唱していませんが、アーキテクチャー・モデルを考慮して設計することは主張しています。アーキテクチャー・モデルは、機能外要求を満たすための 1 つの鍵です。

  したがって、RUP の考え方は、次の点に関して XP と同じです。つまり、「シンプルな設計」ではすべてのテストを実行するべきであり、ソフトウェアが機能外要求を満たしていることを示すテストもこれに含まれます。システムの

サイズや複雑さが増加する場合、アーキテクチャーが新しい場合、機能外要求が複雑な場合にだけ、これが主な問題として浮かび上がってきます。例えば、データを (種類の異なる分散環境で操作するために) 整理する必要が生じると、コードは過度に複雑になると考えられますが、これは全体を通しての必要性です。

- *40 時間労働。*XP と同様に、RUP では超過勤務が慢性的になることを避けるように提案されています。XP は、正確に 40 時間であることを提案しているわけではありません。作業時間に対する耐久度に個人差があることは認められています。ソフトウェア・エンジニアは、完全なものを完成させるという満足感のために、追加報酬なしで長時間働くことで有名ですが、管理者は必ずしもそれを止める必要はありません。

  管理者がすべきでないのは、それを利用したり、強制することです。また管理者は、たとえ報酬の対象にならないとしても、常に実際の作業時間を評価するためのメトリックスを収集すべきです。誰かの作業時間記録が制限を超過している場合は、間違いなく調査すべきです。ただし、特定の状況では、チームのほかのメンバーとの関係を認識して、管理者と個人の間で解決すべき問題もあります。40 時間というのはガイドでしかありません (ただし、有力なガイドです) 。

ロール

RUP では、タスクはロールによって実行されます。[12] また、ロールには特定の成果物に対する責任があります。通常、責任のあるロールは成果物を作成し、ほかのロールによって行われた変更が (変更が認められている場合)、成果物を決して破壊しないようにします。RUP におけるロールは、個人またはグループによって実行されます。同時に、個人またはグループは複数のロールを実行します。1 つのロールを、組織中の 1 か所に割り当てる必要はありません。複数のロールを、組織的な単位に多対多で割り当てる場合もあります。

RUP のロール

RUP では、全部で 30 のロールが定義されています。ロール (例えば、ソフトウェア・アーキテクト) は必ずしも 1 つの作業分野に限定されないため、作業分野との正確な対応はありません。ロールの数を主に属する作業分野ごとに数えると、次のようになります。

- ビジネス・モデリング - 3
- 要求 - 5
- 分析/設計 - 6
- 実装 - 3
- テスト - 2
- 配置 - 4
- 構成と変更管理 - 2
- プロジェクト管理 - 2
- 環境 - 3

RUP に多くのロールがある理由

RUP のロールは、タスクを分割するために使用されます。また、ロールを実行するために必要なスキルと能力をうまく判断するために使用されます。これにより、ロールを実行するスタッフを選択しやすくなります。分割のレベルにより、ロールの重要性が変更されたときに新しい組織配置を確認しやすくなります。例えば、小規模で形式性の低いプロジェクトでは、プロジェクト管理者と構成管理者 (RUP のロール) は、同じ人物が兼任するとうまくいきます。また、大規模で形式性の高い軍事/航空宇宙プロジェクトでは、構成管理者の作業はきわめて特殊化され、小規模なチームで担当者を設ける必要があるほど複雑です。このようにロールを説明することで、ロールの割り当てが容易になります。

XP のロール

参考資料 [Beck00] では、XP の 7 つのロールが示されています。また、ロールの責任、スキル、これらのロールを実行する人物に必要な特性についても説明されています。これらのロールを次に示します。

- プログラマー
- 顧客
- テスト担当者
- トラッカー
- コーチ
- コンサルタント
- 上司

[12] より正確には、タスクはロールを*実行する* 個人またはチームによって実行されます。

これらのロールは、XP の書籍の、ロールが実行するタスクに関する項目で説明されています。

XP と RUP のロールの相違は、簡単に説明できます。

- XP では、RUP の作業分野のすべてを扱っていません。
- XP のロールは、実際には RUP のロールよりも立場に近いものです。例えば、XP のプログラマーは、実際には複数の RUP のロール (実装担当者、統合担当者、コード・レビュー担当者) を実行します。そして、これらのロールでは必要な権限が少しずつ異なります。

RUP のロールを小さなプロジェクトに (RUP の小規模プロジェクトの手順に示すように) 対応させると、対応付けられる XP のようなロール (つまり、立場) の数は、30 以上減ります。小規模プロジェクトの手順では、立場の数は 5 つです。次の表を参照してください。

ABC 社の小規模プロジェクトに対応する RUP のロール

ABC 社における役職	RUP のロール
プロジェクト管理者	プロジェクト管理者 プロセス・エンジニア 配置管理者 要求レビュー担当者 アーキテクチャー・レビュー担当者
ABC 社管理職	プロジェクト・レビュー担当者 利害関係者 要求レビュー担当者
チーフ・プログラマー	システム分析者 ユースケース定義者 ユーザー・インターフェース設計者 ソフトウェア・アーキテクト 設計レビュー担当者 プロセス・エンジニア ツール・スペシャリスト 構成管理者 変更管理責任者 より小規模な場合は、プログラマーと同じロール
プログラマー	設計者 実装担当者 コード・レビュー担当者 統合担当者 テスト設計者 テスト担当者
管理アシスタント	*担当業務* • *「小規模プロジェクト」の Web サイト保守* • 計画またはスケジュール作成のアクティビティーにおける、プロジェクト管理者ロールの補助 • 成果物に対する変更を管理する際の、変更管理責任者ロールの補助 • *必要に応じた、ほかのロールの支援*

結論

XP と RUP の考え方は、実際には同じではありません。RUP は、特定のプロセスを構成して、インスタンス化できるプロセス・フレームワークです。RUP は構成する必要があり、実際に RUP 自体に定義されている必須の手順です。厳密に言えば、カスタマイズされた RUP と XP を比較すべきです。つまり、XP が明示的に確立する (そして推論可能な)、開発プロジェクトに影響する要因に対してカスタマイズされた RUP を使用するべきです。このようなカスタマイズされた RUP プロセスは、いくつかの XP の実践原則 (ペア・プログラミング、テスト先行設計、リファクタリングなど) に適応しますが、XP と同じではありません。アーキテクチャー、抽象化 (モデリングにおける)、リスクの重要性に対する認識や、時間に関する構造 (フェーズ、反復) が異なります。

RUP では、プロセスを作成して、規模や種類が XP の開発範囲から外れているプロジェクトに適用することができます。RUP は、プロセス群の完全な「説明」であるという点においてだけ重量級です。実装においては、成果物、納入物、正式性、規定、儀式性、「級」に対するその他の基準に関して、必要に応じて軽量級と重量級のどちらにもなれます。XP は、意図的に (軽量な) プロセスの実装に注目している点で、確かに軽量級です。 ただし、XP の説明も (少なくとも参考資料として使用した書籍では) 十分に仕上げられていません。XP の実装では、その場で発見、調査、定義しなければならないことがあります。したがって、RUP との比較において、XP は資料の面でも軽量級です。

この状況は変化しそうです。実際に、2 冊の書籍により既に変化が見られ、このうちの 1 冊 [Succi01] には 512 ページに及ぶ説明があります。ただし、現状では、これらの手法を採用するにあたって必要な労力については、両者の間に相違があります。RUP では、先行的な労力のほとんどを、トレーニングの要求とプロセスのカスタマイズの両方に使用します。組織は、特定の種類と規模のプロジェクトにおける組織全体での適用に対して RUP をカスタマイズし、その結果を複数のプロジェクトで使用します。XP の場合は、先行的なトレーニングもいくらか必要ですが、採用に伴う労力の残りはプロジェクト全体に広がります。つまり、XP を機能させるために必要であると判明した補助的なものすべてに労力が広がり獲得されます。XP では、「コーポレート・メモリー」の獲得は明確に示されていません。組織は (プロセスの経験を記録しておかなければ) スタッフの再編に対して脆弱になります。

無条件に RUP を重量級、XP を軽量級と分類することは、それぞれの持つ本来の性質や意図されていることがあいまいになり、どちらも損失になります。いずれかを軽視するような姿勢には、まったく意味がありません。「重量級」または「軽量級」であるのはプロセスとしての実装であり、必要としている環境に応じたサイズであるべきです。

XP は自由な形式ではなく、すべてのことは規律正しく行われます。ソフトウェア開発の特定の側面や価値を納品する方法に限定的に注目し、これを実行する方法を強く規定しています。

RUP は、より幅広く、深い適用範囲を持ちます。 これは、明白な「規模」を示しています。ただし、プロセスを微小なレベルで考えると、XP とは逆に、RUP は同じように有効な選択肢 (ペア・プログラミングの実践原則など) を許容し提案する場合があります。このホワイト・ペーパーは、XP を批判する目的で作成したものではありません。XP の名前が示すように、XP が注目する内容をどのように制限するかを説明しているにすぎません。

参考資料

[Beck00] "*Extreme Programming Explained*", Kent Beck, Addison-Wesley, 2000 (邦訳:「XP エクストリーム・プログラミング入門」長瀬 嘉秀、永田 渉、飯塚 麻理香訳、ピアソン・エデュケーション)

[Beck01] "*Planning Extreme Programming*", Kent Beck, Martin Fowler, Addison-Wesley, 2001 (邦訳:「XP エクストリーム・プログラミング実行計画」長瀬 嘉秀、飯塚 麻理香訳、ピアソン・エデュケーション)

[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 (邦訳:「リファクタリング ープログラムの体質改善テクニック」児玉 公信、友野 晶夫、平澤 章、梅澤 真史訳、ピアソン・エデュケーション)

[Jeffries01] "*Extreme Programming Installed*", Ron Jeffries, Ann Anderson, Chet Hendrickson, Addison-Wesley, 2001 (邦訳:「XP エクストリーム・プログラミング導入編」平鍋 健児、高嶋 優子、藤本 聖訳、ピアソン・エデュケーション)

[Kruchten00] "*The Rational Unified Process, An Introduction, Second Edition*", Philippe Kruchten, Addison-Wesley, 2000 (邦訳:「ラショナル統一プロセス入門 第 2 版」藤井 拓監訳、ピアソン・エデュケーション)

[Martin01] "*Extreme Programming in Practice*", Robert C. Martin, James W. Newkirk, Addison-Wesley, 2001 (邦訳:「XP エクストリーム・プログラミング実践記編」比嘉 康雄、平鍋 健児、高嶋 優子訳、ピアソン・エデュケーション)

[Nosek98] "*The Case for Collaborative Programming*", John T. Nosek, *Comm. ACM,* Vol. 41, No. 3, 1998, 105 ～ 108 ページ

[Succi01] "*Extreme Programming Examined*", Giancarlo Succi, Michele Marchesi, Addison-Wesley, 2001 (邦訳:「XP エクストリーム・プログラミング検証編」小野 剛、石川 真之、細川 馨訳、ピアソン・エデュケーション)

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

Rational®
the software development company

Dual Headquarters:

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

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

Toll-free: (800) 728-1212
E-mail:info@rational.com
Web:www.rational.com
International Locations:www.rational.com/worldwide