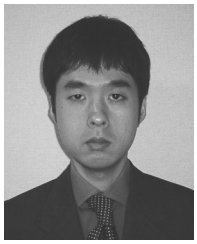
技術レポート

繰り返し型開発プロセス概説
ーRUP、XP を中心にー



金融システム第七事業部　今井雅之

１．はじめに

現在のシステム開発の分野において、RUP（Rational Unified Process）、および XP（eXtreme Programming）の２つを代表とした、繰り返し型開発プロセスが注目されている。その背景としてはビジネス環境、技術要素ともに変革が激しく、システム開発における不安定な要因が増加していることがあげられる。そのような状況においてリスクの早期発見による削減、および変更への柔軟性を実現するためには、繰り返し型開発プロセスを適用することが必須だろう。

そこで本稿では、繰り返し型開発の概要と代表的なプロセスである、RUP と XP について紹介する。

２．繰り返し型開発の特徴

2.1　従来型開発プロセス（ウォーターフォール型）

従来のウォーターフォール型プロセスでは、１つの工程が完了してから次の工程へ進む手順をとる。典型的な例では、要件定義、設計、製作、テストの順に抽象度の高い工程から始めて、段階的に詳細化して実装を行なうことになる（図１参照）。

これにより、プロジェクト全体で歩調を合わせて同じ工程を行なうことになるため、ステータスを把握することが容易であり、要員計画を立てやすくなる。

その反面で、下流工程で問題が発見されて上流工程への手戻りが発生した場合には、工程を順番に完了していくという前提が崩れてしまい、プロジェクト全体に影響が及んでしまう可能性がある。ウォーターフォール型プロセスでは、手戻りは極力減らすことが必要であり、要件定義を完了した時点で「仕様凍結」を行ない、設計以降の工程にとりかかることがポイントになる。

しかし、「凍結」したはずの仕様であってもリリースまでのビジネス環境の変化により、変更や追加を余儀なくされることが多い。特に現在では、日進月歩で新たな情報技術が登場し、先進の技術や多様な要素を組み合わせてシステムを構築することが求められる。また、ユーザーとの仕様に対する認識の相違がプログラムを動かした段階で明らかになったり、設計で想定したメカニズムがうまく実装できなかったりといった、下流工程になってから顕在化するリスクも数多く存在する。このことから、ウォーターフォール型プロセスの最大のメリットである、技術リスクの早期発見と削減を発揮できるチャンスがほとんどないまま、下流工程を迎えることになってしまうケースが多くみられる。

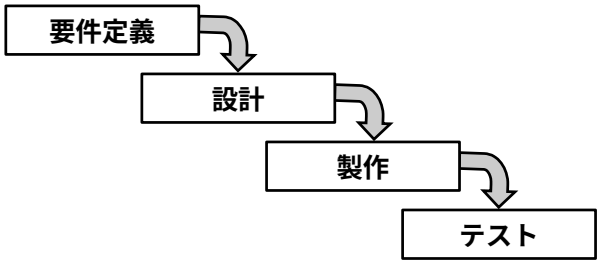


図１　ウォーターフォール型プロセス

2.2　繰り返し型開発

一方、繰り返し型開発プロセスという考え方がある。繰り返し型開発はシステムを小さな機能単位に分割し、テストと構築を繰り返しながら機能を１つずつ開発して、徐々にシステム全体を構築していく方法である（図２参照）。

リスクの高い部分から先に構築することでシステム開発

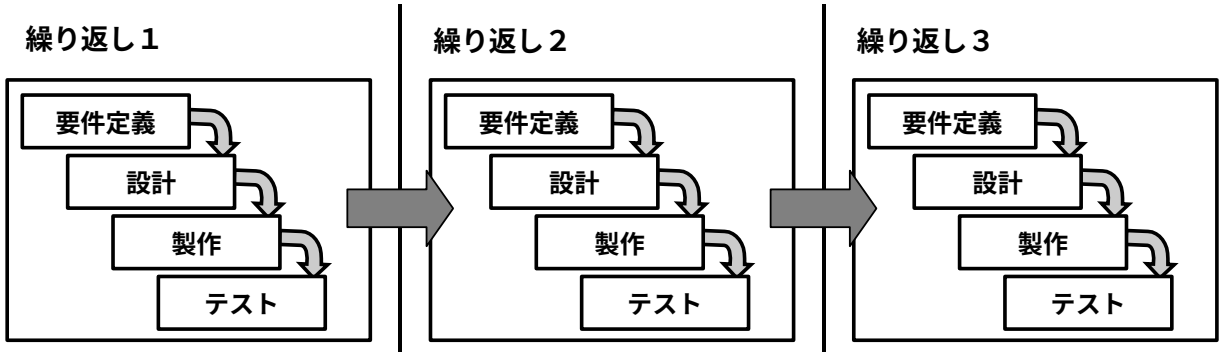


図２　繰り返し型開発プロセス

全体のリスクを削減でき、もし手戻りが発生した場合にも影響範囲を小さくすることができる。

繰り返し型開発は、一般的に以下のような特徴をもつ。

⑴　小さな開発単位

繰り返し型開発を行なうためには、システムを小さな機能単位に分割する必要がある。その各機能単位が開発の基準となり、要件定義からテストまでのすべての開発工程は、この機能単位ごとの小さな範囲に絞って行なうことができる。また、繰り返し型開発の対象とする機能単位は、各開発工程の優先順位に応じて決定することができる。

⑵　ミニプロジェクトの繰り返し

１つの繰り返し型開発の内訳は、要件定義からテストまでを行なうウォーターフォール型プロセスをもつミニプロジェクトとして構成される。したがって、開発者はプロジェクト初期のミニプロジェクトにおいて、繰り返し型開発プロセスのひととおりの手順を経験することになる。このため、設計ドキュメントの形式、標準化、実装上の問題、テスト方法など、プロジェクトに関わるさまざまな問題を早期に認識することが可能である。

そして１つの繰り返し型開発の終了は、システムの部分的なリリースを意味する。たとえ機能的には不充分であっても、実際に動作するシステムの一部が完成したことで、ユーザーはプロジェクトの進捗状況の評価や仕様の確認を具体的に行なうことが可能となる。

一方で、開発者も繰り返し型開発ごとのリリースにより、技術的な確信と達成感を得ることができる。また、繰り返し型開発は短いサイクル、かつ一定のリズムを持った作業であるため、確度の高いスケジューリングを可能にし、長期間の開発工程における中だるみ状態を防ぐ効果も期待できる。

また、開発途中で仕様の追加、変更があった場合には、次の繰り返し型開発で行なわれる要件定義に取り込むことが可能となる。実際には開発スケジュールやリリース時期との兼ね合いで難しいこともあるだろう。しかし、システム仕様の見直しを行なう機会を開発プロジェクトの要所ごとに設けられることは、大いに意義がある。

⑶　アーキテクチャ主導

アーキテクチャはシステムの構造を意味する。本来のシステム構築は、安定したアーキテクチャの構想を実現した上で実施する必要がある。しかし、Web アプリケーションのように技術革新が早く、比較的経験も少ない上にシビアな性能が要求される分野では、実際に作って動かすまでアーキテクチャに確信を持てない場合が多い。そして、このような場合にこそ、繰り返し型開発プロセスの適用が最も大きな効果を発揮するのである。具体的には、開発プロジェクトの初期の段階で、アーキテクチャを決定する重要な機能を対象として繰り返し型開発を行って検証を行ない、アーキテクチャを確定することが可能となる。そして後続の繰り返し型開発では、確定したアーキテクチャにしたがって安定した開発が行なえるため、アーキテクチャに関するリスクを初期の段階で大幅に削減することができる。

なお、プロジェクト初期の段階で実際に動作する機能を作成して検証を行なうという意味では、繰り返し型開発はプロトタイピング手法と共通する部分がある。しかし、プロトタイピングでは作り捨ての機能試作品的な意味合いが強いのに対し、繰り返し型開発では実際にリリースして運用可能なものを作成する。繰り返し型開発は、より確度の高い検証を目指すという点が異なっており、プロトタイピングの発展形ともいうことができる。

３．RUP

3.1　概要

RUP は Objectory 法[*1]をベースに、いくつかの方法論

＊１）Objectory 法：Ivar Jacobson によって提唱されたソフトウェアの工業的開発のための体系的プロセス。ユースケースを中心とした設計に基づき、実際のシステム使用方法の理解に主眼をおいたオブジェクト指向分析／設計へのアプローチ方法。

および実践経験、およびUML（Unified Modeling Language）[*2]記法を取り入れて作られた開発プロセスであり、Rational Software社から製品としてリリースされている。RUPでは各種ワークフロー、ワーカーの役割、成果物などが詳細に規定されており、利用者はプロジェクトの状況に応じて必要なものを取捨選択し、カスタマイズして適用することができる。

3.2 基本概念

⑴ ユースケース駆動

RUPでは、システムが実現する機能をユースケース[*3]として分割し、開発プロセスのすべてにわたった基本単位として利用する。そのため、分析、設計、実装、テストの各工程の成果物をユースケースごとにトレースすることができ、ユーザー要求が確実に実現されていることの確認が可能となる。

⑵ アーキテクチャ中心

RUPでは、安定したアーキテクチャを実現することを重視し、繰り返し型開発の初期段階で検証を行い、アーキテクチャ・ベースラインを作成する。

なお、アーキテクチャはシステムの機能構成、オブジェクトの構造、ソースファイルの構成、ハードウェアの配置など複数の視点で捉える必要がある。RUPでは、アーキテクチャを「４＋１ビュー」として定義している（図３参照）。

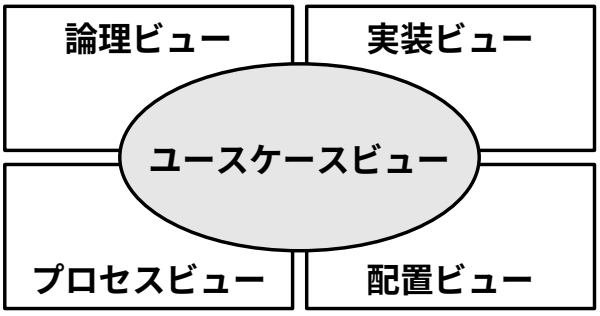


図3　4＋1ビュー

⑶ 反復的でインクリメンタル

RUPでは、ユースケース単位に分割した機能を、１カ月～数カ月間のミニプロジェクトの繰り返しにより、少しずつ完成させていく。ただし、無計画に繰り返し型開発を行なうと現場の混乱を招き、開発効率を悪化させることになるため、プロジェクト全体を反復ワークフローで進行する。

3.3 反復ワークフローの４つのフェーズ

反復ワークフローでは、プロジェクトを４つのフェーズに分けて進行し、各フェーズでは１回～数回の繰り返し型開発が行なわれる。繰り返し型開発では、要求定義、分析／設計、実装、テストという基本フローから構成するミニプロジェクトが実行される。各フェーズについて、以下に説明する。

⑴ 方向付けフェーズ

方向付けフェーズでは、プロジェクトの目標や範囲を決定する。ユースケースのリストアップやアーキテクチャ候補の作成を行なうことで、システムの全体像と今後進むべき方向を決めていく。

⑵ 推敲フェーズ

推敲フェーズでは、アーキテクチャ・ベースラインを確立することを目標とし、さらにユースケースの大部分の詳細化を行なって次の作成フェーズに備える。リストアップされたユースケースの内で重要なものを対象として実装を行ない、アーキテクチャを検証する。よほど単純なアーキテクチャでない限りは、万全を期すために、このフェーズで２回以上の繰り返し型開発を行なうことが望ましい。

⑶ 作成フェーズ

推敲フェーズで決定したアーキテクチャに基づいて、本格的にユースケースの実装を行ない量産体制に入る。作成フェーズは、プロジェクトの規模に応じて何回か繰り返して実行され、それぞれの終了時点でシステムがベータリリースされる。ベータリリースとは、初期運用が可能なシステムのことである。

⑷ 移行フェーズ

ベータリリースとなったシステムをユーザー環境にインストールし、受入れテストを行なう。テストからのフィードバックにより問題点を修正し、受入れテストの合格をもってプロジェクトを終了する。

４．XP

4.1 概要

XPはクライスラーＣ３プロジェクト[*4]での経験などを

＊２）『SOFTECHS』Vol. 23, No. 2（2000年11月発行）掲載の「UMLによるモデリング－オブジェクト指向開発における分析・設計方法－」（先端技術研究室山田喜彦）を参照。

＊３）ユースケース：システムの機能を、そこに関わるユーザーや他システムなど外部アクターの視点から捉え、シナリオなどの形で記述したもの。

＊４）クライスラーＣ３プロジェクト：ケント・ベック氏らがコンサルタントとして従事したクライスラー社の賃金支払システム構築プロジェクト。一度失敗し、すべてのコードを破棄した状態からXPを用いることでプロジェクトを成功へと導いた。

元に、ケント・ベック氏らにより確立された開発プロセスである。現在では、多くのコミュニティによって議論が行なわれており、日本でもメーリングリストなどの活動が盛り上がりつつある。

XP は、RUP とは対照的な軽量級プロセスであり、ワークフローや成果物の詳細な規定がない代わりに「12のプラクティス」という実践すべき項目があげられている。そして eXtreme Programming という名前のとおり、開発作業の中でも、特にプログラミングに重点が置かれている。先を見越した設計ドキュメントを書くよりも、まずはプログラムを作り、必要になった際に修正する、という変化に対する柔軟さを保つことが特徴となっている。

また、コミュニケーションの重要性や職場の施設にも言及するなど、プロジェクト・メンバーを単なる「リソース」ではなく、「人間」として捉える姿勢が貫かれており、これが XP 人気の大きな要因と考えられる。

4.2 変化を抱擁せよ

XP では、システム仕様の変更に対して柔軟に対応できることを重視する。つまり、情報システムを取り巻くビジネス環境や情報技術の変化が激しい中で、始めから正しいゴールを見定めて進んで行くのではなく、常に自分のいる位置を確認し、細かな軌道修正を繰り返しながらゴールを目指す、という考え方である。

従来型の開発方法では、プロジェクト全体に関わる重要な判断は上流工程で行なわれ、それが正しいことを前提として下流工程へ進む手順がとられた。そのため、下流工程に進んでから仕様変更が発生すると、完了したはずの上流工程への手戻りが発生してしまい、多大なコストがかかってしまった。つまり、ソフトウェアの変更コストは、プロジェクト開始から時間が経つほど増大する傾向にあった。

しかし、時間が経過してもソフトウェアの変更コストが一定ならば、プロジェクト初期の段階ですべての判断を行なう必要はなく、状況の変化が起こった時点で開発の軌道修正を行なうことが可能となる。XP は、この夢のような話を実現にしたのだ。

4.3 XP の特徴

XP は以下の３つの要因によって、ソフトウェア開発のどの段階で仕様変更が発生した場合でも、柔軟かつ一定のコストで対応できる。

⑴ シンプルな設計

プログラムは、ほんとうに必要な機能だけを盛り込んだシンプルな設計とし、それ以外の機能は必要になった時点で追加すればよい。設計段階で将来的に使う可能性があるから、という理由で不確定な機能を盛り込んでおいても、プログラムが複雑になるばかりか状況が変化すれば不要になってしまい、無駄になる可能性がある。また複雑なプログラムは変更にかかるコストも増大してしまう。

⑵ 自動化されたテスト

ひとたび完成したプログラムに修正を加えた場合、追加した機能だけでなく、既存の機能に影響がないことまで再テストする必要がある。これは人間系で実行すると、たいへんな手間がかかる。そこで、自動化テストを導入することで、変更による再テストの手間とコストを減らすことができる。

⑶ 設計を常に改良する

シンプルな設計によって作られたプログラムでも、変更を繰り返していくと重複する部分や、他のプログラムとの整合性を保つために改良すべき部分などがでてくる。蓄積した改良点を一斉に修正を行なうのは面倒で退屈な作業である。そこで、「設計を改良する」という意識を常に持ってプログラミングを行ない、設計を改良すべきという「におい」を感じた時点で、すぐに修正を行なうようにする。これにより特別なコストや時間をかけずに正しい設計を保つことができる。

4.4 12のプラクティス

次に XP の根幹をなす12のプラクティスをとりあげる。これらは特に出色したアイデアというわけではなく、個々は「わかってはいるが守ることが難しい」といったものが多い。しかし、XP ではこれらを包括してプログラミングに適用することで大きな相乗効果を狙っている。

⑴ 計画ゲーム

XP では、ユーザーの要求をユーザー・ストーリーとして捉える。これは RUP でのユースケースと同様の位置付けで、開発作業の基本単位となるものである。しかし、ユースケースのように詳細な要求定義を行なうのではなく、ストーリー・カードと呼ばれる１枚のカードにユーザー要求の概要を記述するものである。

開発者は、どのストーリー・カードを次のリリースで実現すべきかを決めて、見積りと詳細なスケジューリングを行なう。リリース範囲は小規模を前提とし、まずは大まかな計画を立案して開発を開始し、必要ならば計画の見直しと改良を行なっていく。基本的には、計画中で最もリスクのある部分を先にスケジューリングしてリスクの削減を行なう。これは RUP の推敲フェーズにおいて、重要なユースケースを実装してリスクを減らす、という考え方とも一致している。

⑵ 小規模リリース

リリース範囲を小規模とすることでリリース計画の立案が容易になる。設計も、そのリリースに必要なものだけを行なえばよいため、シンプルに考えることが可能となる。さらに、計画が失敗してしまった場合にも影響を最小限に

抑えることができる。典型的なサイクルは３週間とされている。

⑶ メタファ（比喩）

メタファはシステムを他のものに例えて表現したもので、アーキテクチャ説明書に相当する。筆者は具体的なメタファのイメージをあまり把握できていないが、よほどシンプルなシステムでない限りは、メタファのみでアーキテクチャの理解を共有することは難しいように思える。

⑷ シンプルな設計

先に述べた、変更に強いプログラムを実現するために必要な要因の１つである。この考えは非常によく理解できるが、実践するのはなかなか難しい。特にオブジェクト指向を採用した場合、再利用や拡張性の名目で、抽象クラスの追加やテンプレートメソッドのような「拡張点」を作成しがちである。しかし、本当に必要か、他人が容易に理解できるか、などをよく検討してみる必要があるだろう。

⑸ テスティング

テスティングはプラクティスの中でも最も重要なものであると同時に、XP ではないプロジェクトに対しても非常に有用な考え方を提供している。

テスティングにおいて重要なのは、テストを自動化することである。従来のようにテスト仕様書を書いてテストデータを作成し、実行して結果を検証するという手順を人間が行なうには多大な手間がかかる。そして、プログラムに何か変更があるたびに既存部分のテストをやり直すことは、現実的には不可能である。そこで、ツールを利用してテストを自動化し、すべてのテストを、いつでも繰り返して行なえる仕組みを実現する。そして、変更があった場合には回帰テストを行ない、常にすべてのテストをパスできるよう、プログラムの状態を保っておく。このテストツールについては、Java 用の JUnit をはじめとして、さまざまな環境をターゲットにしたものが作成されており、本誌の別記事[*5]にも詳しく紹介されている。

また、コードを書く時には、必ずテストコードをペアとして書くようにする。テストが重要であることは理解していても、ひととおりコードを書いた後で、改めてテストケースを作成するのは退屈な作業であり、ともすれば手を抜きがちになってしまう。そこで少しコードを書くたびに必ずテストを行ない、パスすることを確認してからまたコードを書く、というサイクルを守ることにする。そうすれば、コードが完成したと同時に必要なテストも完了することになり、プログラムを変更した場合には、それに対するテストを追加するだけでよくなる。

⑹ リファクタリング

XP では、先を見越した計画的なプログラム設計は行なわない。しかし、常に良い設計であるかどうかを考慮し、改良する必要があるときには躊躇なくプログラムを修正することができる。これは自動化テストによる回帰テストが実現され、デグレードの心配をせずプログラミングを行なえるからこそのメリットである。

⑺ ペアプログラミング

プログラマは２人ペアとなり、１台のマシンに向かってコーディング作業を行なう。これは XP の概念の中で最もインパクトがあるとともに、XP の導入に二の足を踏ませる要因にもなっているようだ。

たしかに１人で行なうべきタスクを２人で行なえば、生産性が低下して現実的ではないと考えるのは当然である。しかし、「傍目八目」という言葉もあるように、コーディング中に後ろで傍観していた人がバグの原因を発見する、などという経験を持つ人も案外多いだろう。つまり、１人がタイピングしてコードを書く作業をしている間に、もう１人が広い視点でコードを検証し、品質を高めていくことを想定している。

またコーディング作業中は、個別に黙々と作業を続けることが多くなりがちであるが、ペアでの作業により相互のコミュニケーションも必然的に活発になる。そして、自分の不得意な分野の作業は、スキルを持つ人とペアを組むことにより OJT（On the Job Training）的な効果を得られ、チーム全体のスキル向上も図ることができる。

そして、ネガティブではあるが重要な効果としては、お互いの作業を監視する、ということがあげられる。XP のプラクティス、例えばテスティングやコーディング標準などは多大な手間を要するため、スケジュールが逼迫すると省かれてしまう可能性がある。ペアプログラミングを行うことによって、お互いの「手抜き」を抑止する効果が期待できる。

⑻ 共同所有権

コードは特定の個人が所有するのではなく、全員がすべてのコードを修正する権利を持つ。これはリファクタリングとも関連するが、プログラムの修正が必要な場合は、作成したプログラマに依頼することなく、誰でもすぐに修正することを可能とする。また、自分の担当以外のコードを扱うことで、システム全体に関する知識も得ることができる。

⑼ 継続的インテグレーション

プログラムを修正した場合には、すぐに結合環境にリリースしてテストを行なう。これにより、いっぺんにすべてのプログラムを統合したときに膨大な不具合が発現することを防ぎ、原因の特定を容易にする。

⑽ 週40時間

プログラミングに携わる時間を週40時間以内とする。こ

＊５）『SOFTECHS』本号 p.95～p.101「繰り返し型開発におけるテスト技法－JUnit について－」（先端技術研究室田中佳美）を参照。

れは誰しも賛成するだろうが、実現はなかなか難しいプラクティスといえるだろう。恒常的に長時間の残業をしていては良いアイデアも浮かばないし、そもそも計画自体が間違っているということになる。また、ペアプログラミングでは通常よりも高い緊張状態が続くため、本来はこのプラクティスもあわせて導入すべきであろう。

⑾ オンサイト顧客

システムを実際に利用するユーザーがプロジェクトに常駐する。XPでは、当初は大まかな計画でスタートし、オンサイト顧客に対する質問と議論行いながらゴールへの軌道修正を図っていく。またオンサイト顧客はユーザー・ストーリー、機能テスト（受入れテスト）の作成も行なうことになる。

これも実現できればかなり有効であるが、エンドユーザーを仕事から引き離してプロジェクトに参加させるのは、かなり困難だと思われる。現実的にはメールでのやり取りによる代用などで妥協せざるを得ないだろう。

⑿ コーディング標準

コーディング標準は、どこのプロジェクトでも当然必要なものである。コードの共同所有権やペアプログラミング、リファクタリングといった共同作業が必要なプラクティスを実践するために、XPでは特に重要視されている。

５．繰り返し型開発プロセスの適用

繰り返し型開発プロセスに関するRUP製品やXPの書籍を購入したからといって、そのまま無条件にプロジェクトに適用できるわけではない。プロジェクトの規模、アーキテクチャ、メンバーのスキルなどの要因を考慮して、適用するプロセスを決定する必要があるだろう。

筆者の見たところ、XPはRUPに比べて規約も少なく軽量級プロセスではあるが、むしろ導入のための壁は厚いように感じる。12のプラクティスの内で、ペアプログラミングは心理的に抵抗を感じる人が多いと思われるし、週40時間やオンサイト顧客などは、顧客を含めてプロジェクト周辺の人々の意識まで変える必要があるだろう。

しかし、XPのプラクティスのいくつかは、XPを離れても有効に利用することができる。たとえばテスティング、リファクタリング、継続的インテグレーションなどは、プロジェクトの規模に関わらず高品質を実現するための仕掛けとして利用可能であるし、ペアプログラミングをOJTの手法として取り入れることも有効だろう。

その一方、RUPは基本となるワークフローや成果物が示されており、内容的にも既存の方法論の集大成となっているため、導入への抵抗は比較的少ないだろう。しかし、それだけに内容が膨大であるため、必要に応じて取捨選択をする必要がある。

ドキュメントに関しても、XPではペアプログラミングやコードの共同所有権などでメンバー間のコミュニケーションを図り、極力ドキュメントを削減するのに対して、RUPではUMLを用いた数々の成果物が示されている。このように、RUPとXPはまさに対照的な考え方を持つが、その優劣を論じるだけではなく、状況に応じてそれぞれの利点を取り入れていく姿勢が大切であろう。一般的には、RUPのワークフローを基本としつつ、XPのプラクティスからテスティング、リファクタリングなどを取り入れ、シンプルな設計、週40時間をスローガンとして掲げておく、といった融合案が現実的なのではないだろうか。

筆者が携わっているプロジェクトでは、RUPをベースにプロセスを定義して開発を進めているが、各ワークフローの位置付けや成果物の定義など、現実に適用するためには試行錯誤が必要な部分も多い。そしてRUPやXPが過去の成功例＝ベストプラクティスから生まれたように、我々も実践を重ねることで、本当に有効なプロセス適用のパターンを模索していきたい。

〈参考文献〉

１．I. ヤコブソン，G. ブーチ，J. ランボー：UMLによる統一ソフトウェア開発プロセス，翔泳社，1999

２．K. ベック：XPエクストリーム・プログラミング入門，ピアソンエデュケーション，2000

３．“ Extreme Programming Roadmap ”：
http : //c2.com/cgi/wiki?ExtremeProgrammingRoadmap

４．Don Wells “ A gentle introduction ”：
http : //www.extremeprogramming.org/

５．eXtreme programming FAQ ：
http : //objectclub.esm.co.jp/eXtremeProgramming/