ウォーターフォール開発から反復的開発へ ープロジェクト管理者にとっての移行への挑戦

Philippe Kruchten

Rational Software ホワイト・ペーパー

TP 173, 5/00

Rational®
the software development company

目次

概要

Rational Unified Process (RUP) では、反復アプローチ (スパイラル・アプローチとも呼ばれる) のソフトウェア開発ライフ・サイクルが提唱されています。この方法は、多くの点でウォーターフォール方式より優れていることが、何度も証明されています。ただし、反復的ライフ・サイクルが備える多くの利点を、労力もなく得られるとは考えないでください。反復型開発は魔法の杖ではなく、ソフトウェア開発におけるすべての潜在的な問題と困難を一振りで解決してはくれません。反復的にしただけで、プロジェクトのセットアップ、計画、管理が容易になるわけではありません。実際、プロジェクト管理者はいっそう困難な作業を抱え込むことになります。 特に初めての反復的プロジェクトやそのプロジェクトの初期の反復ではリスクが大きく、早々に失敗してしまう危険性があります。このホワイト・ペーパーでは、プロジェクト管理者の観点から反復型開発の課題についていくつか説明します。また、Rational としてのコンサルティングの経験やスタッフからの試行錯誤の報告からわかった、プロジェクト管理者が陥りやすい共通の「罠」、つまり問題点についても触れます。

反復型開発

従来のソフトウェア開発プロセスは、次の図で示すようなウォーターフォール・ライフ・サイクルに従って行われます。図 1 に示すように、この方式での開発は、要求分析から、設計、コーディングと単体テスト、サブシステム・テスト、そしてシステム・テストへと、前のフェーズの結果に対する限られたフィードバックを受けて、直線的に進行します。

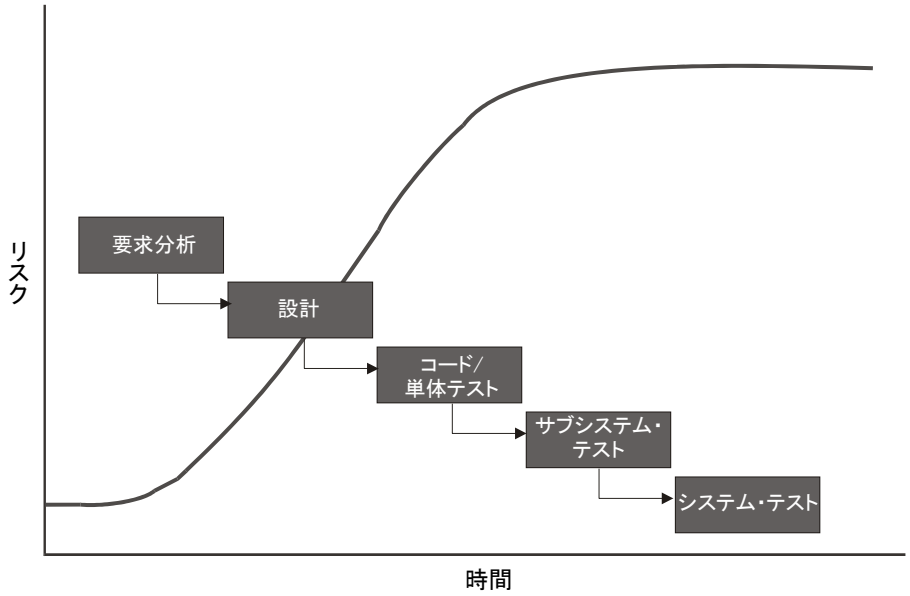

図 1:ウォーターフォール開発のプロセス

この方式の根本的な問題は、リスクが先送りにされ、前のフェーズでの誤りを修正するためにコストがかかることです。初期の設計には重要な要求に関する欠陥が含まれることが多く、さらに、設計の欠陥が後になって発見されると、コストが超過してプロジェクトが中止に追い込まれることもよくあります。ウォーターフォール方式では、有効な対処を実行するには遅すぎる段階になって、プロジェクトに対する真のリスクがわかる傾向があります。

ウォーターフォール方式に代わる方式として、図 2 に示すような反復と追加に基づくプロセスがあります。

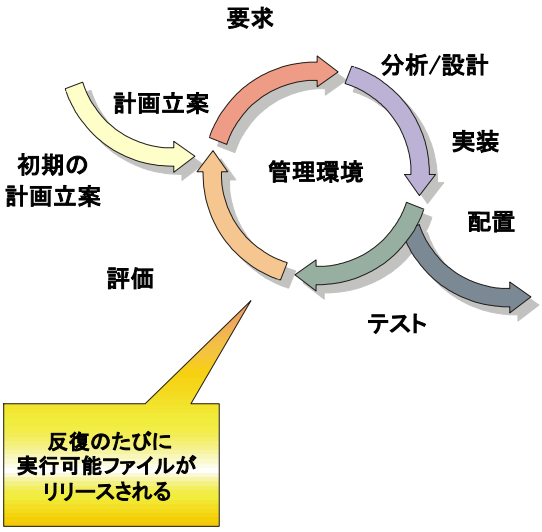

図 2:反復型開発のアプローチ

この方式は、Barry Boehm 氏が考案したスパイラル・モデル (「参考文献」を参照) が基になっており、プロジェクトに対するリスクの識別がライフ・サイクルの初期に行われます。この段階であれば、手遅れになる前に有効な方法でリスクに取り組んで対処することができます。この方法は、発見、考案、実装を連続して行う方法の 1 つであり、開発チームは、反復ごとに予測可能かつ反復可能な方法でプロジェクトの成果物を完成することを求められます。

長所:反復型開発のメリット

従来のウォーターフォール・プロセスと比較して、反復プロセスには以下のような多くの利点があります。

1. 致命的な誤解がライフ・サイクルの早期に明らかになるので、それに対処することができます。
2. ユーザーからのフィードバックが得られやすいので、システムに対する真の要求を引き出すことができます。
3. 開発チームは、プロジェクトにとって最も重大な問題に集中しなければならなくなるので、プロジェクトの真のリスクから注意をそらすような問題に煩わされなくなります。
4. 継続的なテストの反復によって、プロジェクトのステータスを客観的に評価できます。
5. 要求、設計、実装の間の不整合を、早期に発見できます。
6. チーム、特にテスト・チームの作業量が、ライフ・サイクル全体を通してより均等に分配されるようになります。
7. チームは以前の反復の教訓を活かすことができるので、継続的にプロセスを改善できます。
8. プロジェクト内の利害関係者は、ライフ・サイクルの全期間を通して、プロジェクトのステータスを表す具体的な物証を得ることができます。

リスクの軽減

一般に、リスクを発見して対処する唯一の機会は統合フェーズなので、反復プロセスでは早い段階でリスクを軽減できます。また、初期の反復を何回か実行すれば、すべてのプロセス・コンポーネントが実施され、ツール、市販ソフトウェア、人的なスキルなど、プロジェクトのさまざまな側面を経験できます。リスクだと思われていたものがリスクでないとわかることもあれば、新たに予想外のリスクが発見されることもあります。

プロジェクトが何らかの理由で失敗するのであれば、多くの時間、労力、費用を費やす前に、できる限り早く失敗させるべきです。現実から逃避して長時間を無駄にするのではなく、リスクに対処する必要があります。間違った製品を製造するなどのさまざまなリスクの中でも、特に次の 2 種類のリスクの早期軽減には、反復的開発プロセスが役立ちます。

- 統合のリスク
- アーキテクチャーに関するリスク

反復プロセスを採用すると、何回かの反復を経てエラーが修正されるので、より堅牢なアーキテクチャーができあがります。不具合は、早期の反復で方向づけフェーズが終わるときに検出されます。パフォーマンスのボトルネックは、納品の直前ではなく、まだ対処の可能な時期に発見されます。

統合は、ライフ・サイクルの最後に 1 回だけ発生する「ビッグ・バン」ではなく、要素が段階的に統合されていきます。実際、推奨される反復アプローチでは、ほとんど連続して統合が行われます。プロジェクトの最後に全労力の 40% を費やして行われていた不確実で苦労の多い長時間の作業は、6 ～ 9 回の小規模な統合に分割され、はるかに少ない数の要素を統合することから始まるようになります。

変更への対応

以下のように、何種類かの変更を考えることができます。

- 要求の変更

  反復プロセスでは、要求の変更を考慮する必要があります。要求は変更されるのが普通である、というのが正しい考え方です。要求が変更されたり「要求が揺れ動く」ことは、常にプロジェクトのトラブルの最大の原因であり、納品の遅れ、スケジュールの狂い、顧客の不満、開発者の欲求不満などをもたらしてきました。しかし、製品の初期バージョンをユーザー (またはユーザーの代表) に公開することで、製品をより業務に適したものにすることができます。

- 戦術的な変更

  反復プロセスでは、既存製品との競合など、製品に対する戦術的な変更を行う手段が管理者に提供されます。機能を削った製品を早い段階でリリースすることによって競争相手の動きに対抗したり、特定の技術に別のベンダーを採用したりすることができます。また、反復の内容を再編成して、供給業者による解決が必要となるような統合の問題を軽減することもできます。

- 技術的な変更

  範囲は限定されますが、反復アプローチを使えば技術的な変更にも対応できます。推敲フェーズの過程であれば技術的な変更に対応できますが、作成フェーズと移行フェーズでは、本質的なリスクがあるためこの種の変更は避ける必要があります。

作業と並行した学習

反復アプローチの利点は、開発者が開発を行いながら学習することができ、さまざまな能力や専門性をライフ・サイクル全体を通してよりいっそう発揮できることです。例えば、テスト担当者は早くからテストを行い、テクニカル・ライターは早くから文書を作成します。 一方、反復的ではない開発では、これらの担当者は繰り返し計画を立てながら、作業の開始を待たなければなりません。評価レビューの早い時期から、トレーニングや (たいていは外部からの) 追加支援の必要性が出てきます。

プロセス自体も、作業を進めながら改善して洗練することができます。反復の最後に行われる評価では、製品やスケジュールの面からプロジェクトの状況が調べられ、組織やプロセスで変更が必要な点が分析されて、次の反復に活かされます。

再利用の機会の増加

反復プロセスでは、最初からすべての共通部分を明確化するのではなく、部分的な設計または実装を通して容易に共通部分を識別できるので、プロジェクトの要素の再利用が促進されます。再利用可能な部分を識別して開発する作業は、簡単ではありません。初期の反復で設計レビューを行うことにより、設計者は予想外の潜在的な再利用を識別し、以降の反復の中で共通のコードを開発して充実させることができます。推敲フェーズの反復の中で、共通の問題に対する共通のソリューションが発見され、システム全体に適用するパターンとアーキテクチャー・メカニズムが明確になります。

全体的な品質の向上

反復プロセスで開発された製品は、従来の順次的なプロセスで開発された製品よりも全体的に品質が優れています。システムが繰り返しテストされて、そのたびにテストの品質は向上していきます。要求が洗練され、ユーザーの実際のニーズとより密接に関連付けられるようになります。そして、納品の時点では、システムは既に長時間稼働していることになります。

短所: 予想外の悪影響と陥りやすい問題点

反復的開発を行っても、作業が軽減され、スケジュールが短縮されるとは限りません。反復型開発の主なメリットは、結果とスケジュールをより正確に予想できることです。反復型開発では、要求だけでなく設計や実装も時間をかけて発展させるので、エンド・ユーザーの真のニーズを満たす高品質の製品ができあがります。

実際には、反復型開発ではより多くの計画立案が必要になるので、プロジェクト管理者の負担はかえって重くなる可能性があります。全体的な計画を作成し、さらに各反復に対する詳細な計画立案を行う必要があります。また、問題、ソリューション、計画のトレードオフを、常に調整する必要もあります。アーキテクチャーに関する計画立案作業も増えて、しかも早い段階から発生します。成果物 (計画、文書、モデル、コード) の修正、レビュー、承認を、リビジョンごとに繰り返し行う必要があります。戦術や範囲を変更すると、絶え間ない計画の見直しを余儀なくされます。したがって、反復ごとにチーム編成の変更がいくらか必要になります。

問題点:終了時までの詳細すぎる計画立案

スケジュールやリソースの全体的な概要を評価するために行う場合を除き、通常は、開始から終了までの詳細な計画を作成するのは無駄なことです。最初の反復が終わる前に、このような計画は役に立たなくなるでしょう。アーキテクチャーが固定され、要求が確実に理解されるまでは (だいたいは LCA (ライフ・サイクル・アーキテクチャー) マイルストーンの時点) 、実際的な計画を立てることはできません。

したがって、計画立案の対象となるアクティビティー、成果物、反復に関してわかっていることに応じた正確さで、計画を立てるようにします。短期間の計画は、詳細できめの細かいものにします。期間の長い計画は、おおまかな形式のままにします。

知識や情報が不足している管理者が「包括的な全体計画」を立てようとするかもしれませんが、その圧力には抵抗する必要があります。管理者を教育して、反復の計画立案の概念と、はるか将来まで詳細に予測しようとする労力が無駄であることを説明します。たとえとして、ニューヨークからロサンゼルスまでの自動車旅行が役に立ちます。全体のルートを計画するかもしれませんが、必要なのは、都市部を出て旅の最初の行程に乗るまでの詳細な経路だけです。カリフォルニアへの到着は言うまでもなく、カンザスを通り抜ける詳細な経路も計画する必要はありません。 カンザス州内の道路が工事中であれば、別の経路を探さなければなりません。

再作業に対する事前の認識

ウォーターフォール方式では、最終的なテストと統合の際に重大なバグが発見されると、最後の最後になって計画外の面倒な再作業が大量に発生する場合があります。さらに悪いことに、欠陥の主な原因が設計のエラーであるとわかり、実装において回避策を講じてその影響を軽減しようとすると、さらに欠陥が発生することがあります。

反復アプローチでは、再作業が発生し、特に最初は再作業量が多いということは事前にわかっています。初期のアーキテクチャーのプロトタイプで問題が見つかったら、その再作業を行う必要があります。また、実行可能なプロトタイプを作成するにはスタブと足場を用意する必要があり、それを後でさらに充実した堅牢な実装に置き換えます。健全な反復的プロジェクトでは、スクラップまたは再作業の割合は急速に減ります。 アーキテクチャーが安定し、困難な問題が解決されるにつれて、変更の範囲は狭くなります。

問題点:集束しないプロジェクト

反復型開発は、反復ごとにすべてをスクラップにするという意味ではありません。スクラップと再作業の量は、反復を重ねるに従って減っていくはずで、特に LCA マイルストーンでアーキテクチャーがベースライン化された後では少なくならなければなりません。開発者は、よりよい技法の導入や再作業の実行などの金メッキのために、反復型開発を利用しようとすることがよくあります。プロジェクト管理者は、問題がなく十分によい状態の要素については再作業を認めないように、用心する必要があります。また、開発チームの規模が大きくなり、メンバーの異動があると、新しいメンバーが入ってくることがあります。新しいメンバーは、作業の進め方に自分なりの考えを持っていることがよくあります。同様に、顧客 (またはマーケティングや製品管理などのプロジェクトの代表者) が、反復型開発での自由度を悪用して変更を加えたり、際限なく要求の変更や追加をしたりしようとする場合があります。この場合の影響を「要求の揺れ」と呼ぶことがあります。プロジェクト管理者は、厳格にトレードオフを判断し、優先順位を調整する必要があります。LCA マイルストーンの前後で要求がベースライン化されると、スケジュールと予算が調整し直されない限り、すべての変更のコストに制限が設けられます。何かを加えるということは、何かを取り除くことを意味します。

そして、「完璧を求めることは害である (Perfect is the enemy of good.)」(フランス語では、「Le mieux est l'ennemi du bien」) ということを忘れないでください。

問題点:明確な目標を決めずに開始する

反復型開発は、永久に続くファジーな開発という意味ではありません。チームを忙しくしておくだけのために、または明確な目標が突然浮上してくることを期待して、設計やコーディングを開始しないでください。明確な目標を定義してそれを文書化し、すべての関係者の同意を得る必要があります。 その後、目標を練り上げて発展させ、再び同意を得ます。反復型開発のよい点は、設計、コーディング、統合、テスト、評価を始める前に、すべての要求を決定しておく必要がないことです。

問題点:自分の成功に囚われる

注意を要するリスクは、プロジェクトの終了が近くなって、消費者のコインが突然裏返ったときに発生します。つまり、何も納品できないだろうと確信していたユーザーが、チームは本当にやり遂げるかもしれないと確信し始めたときです。好材料は、プロジェクトに対する外部の認識が切り替わったことです。しかし、月曜日に一部が納品されて満足していたユーザーも、火曜日には、すべてが納品されることはないのではないかと心配し始めます。これは悪材料です。最初のベータ版と 2 番目のベータ版の間のいずれかの時点で、開発者は、ユーザーが最初のリリースに追加しようと考えている機能のことで自分の頭がいっぱいになっていることに気付きます。突然、これらが大きな問題になります。受け入れ可能な最低限の機能を提供することに腐心していたプロジェクト管理者も、今では、最後まで残ったすべての要求が最初の納品で「必須」であるかのように考える状況に陥ります。コインが裏返ったときのように、すべての未決項目が優先度「A」の状態に昇格したかのように考えてしまいます。実際には行うべき事項の数は変わらず、その実行にかけられる時間も同じです。外部の認識が変わっても、優先順位を考えることは依然として非常に重要です。

いざというときにプロジェクト管理者が気後れして要求をすべて受け入れ始めると、プロジェクトを再びスケジュール上の危機に陥れることになります。このときこそプロジェクト管理者は毅然とした態度を取り、新しい要求を受け入ないようにします。新しい要求と引き換えに何かを取り除くだけでも、この段階に来てリスクが拡大する可能性があります。用心しなければ、成功を目前にして敗北してしまうことになります。

ソフトウェアを最優先する

ウォーターフォール・アプローチで重点が置かれることは、「仕様」(つまり問題空間の記述) とそれを適正、完全、かつ精緻なものにして、承認を受けることです。反復プロセスでは、開発するソフトウェアが何より優先されます。ソフトウェア・アーキテクチャー (つまりソリューション空間の記述) で、ライフ・サイクル初期の決定事項を運用する必要があります。顧客は仕様を買うのではありません。主に注目しているのはソフトウェア製品であり、仕様とソフトウェアは並行して発展します。「ソフトウェア最優先」が重視されているということは、さまざまなチームに影響を及ぼします。例えば、テスト担当者は、これまではテスト開始に必要な完全で安定した仕様と、多くの事前情報を受け取っていたかもしれません。一方、反復型開発では、まだ発展途中の仕様と要求を基にして、直ちに作業を始める必要があります。

問題点:管理成果物の過剰な重視

次のようなことを言う管理者がいます。「私はプロジェクト管理者なので、できる限り最善の管理成果物を作成することに重点的に取り組む必要があり、それこそがすべてに対する鍵である。」これは正しくありません。優れた管理も重要ですが、プロジェクト管理者は、最終製品が最良であることを保証する必要があります。プロジェクト管理とは、失敗はしても可能な限り最善の管理を行ったことを示して、自分自身を擁護することではありません。同様に、過去に下手な要求管理のためにひどい目にあったことがあり、可能な限り最善の仕様を作成することを重視するのかもしれません。しかし、対応する製品にバグがあり、遅く、不安定で、壊れやすいのでは、仕様を最善にしても何の役にも立ちません。

困難な問題の早期解決

ウォーターフォール・アプローチでは、困難な問題、リスクのある事柄、そして実際にはわからないことの多くは、計画立案プロセスでは右側に押しやられて、差し迫ったシステム統合アクティビティーになってから解決策が取られます。その結果、プロジェクトの前半は比較的平穏に経過し、この間、問題は、多くの利害関係者 (テスト担当者など) 、ハードウェア・プラットフォーム、実際のユーザー、実際の環境とは関係のない紙の上で扱われます。そして突然、プロジェクトは統合の地獄に陥り、あらゆるものがバラバラになってしまいます。反復型開発では、計画立案は主としてリスクと未知の事項に基づいているので、開始時から耐久性を持っています。困難、危機的で、多くの場合レベルが低い一部の技術的な問題は、後回しにするのではなく、直ちに対処する必要があります。つまり、かつて誰かが言っていたように、反復型開発では長い間 (自分自身または世の中に対して) 嘘をついていることはできません。失敗が運命付けられているソフトウェア・プロジェクトは、反復アプローチの早い時期にその運命と出会う必要があります。

例えるなら、大学のコースで、学期の前半は比較的基本的な概念を教授が教えているようなものです。そのようなコースは、最低限の努力を払えば中間テストでよい成績を得られる簡単なクラスという印象を与えます。そして、学期が終わりに近づくと急に加速します。教授は、期末テストの直前になってすべての高度なテーマを採り上げます。この段階でもっともよくあるシナリオが、クラスの大部分がプレッシャーに押しつぶされ、期末テストは惨憺たる結果になる、というものです。驚くべきことに、ほかの点では聡明な教授が、毎年クラスごとに繰り返されるこの惨事に面食らうのです。より賢明な方法としては、コースの前半に力を注ぐようにして、困難な課題をいくつか含めて、授業内容の 60% を中間テストまでに取り組むようにします。この例を反復プロジェクトの管理に対応させると、プロジェクトの始めのほうで、問題にならないことの解決やつまらない作業の遂行に貴重な時間を費やさないようにするということになります。開始時の技術的な失敗として最も一般的な原因は、「簡単な事柄にすべての時間を費やしてしまった」ことです。

問題点:問題から目をそむける

次のようなことを言ってしまいがちです。「これはデリケートな問題なので、時間をかけて考える必要があります。後で解決することにしましょう。そうすれば、これについて考える時間がもう少しできます。」こうなると、プロジェクトで着手されるのはすべて簡単な作業になり、困難な問題にはあまり注意が払われなくなります。解決が必要な段階になると、解決と決定が性急に行われるか、またはプロジェクトの計画が狂うことになります。これとは反対のことを行う必要があります。困難な問題には直ちに対処します。私はときどきこう言います。「プロジェクトが何らかの理由で失敗することになるのなら、すべての時間と費用を費やす前に、できる限り早く失敗させるべきだ。

問題点:新しいリスクについて忘れる

方向づけにおいてリスク分析を実施し、それを使用して計画を立てます。しかし、その後は、プロジェクトで後から発生するリスクについては忘れてしまいます。そして、後でそのようなリスクのために傷を負うことになります。リスクは、(毎月でなければ) 週ごとにでも再評価を続ける必要があります。最初に作成したリスクのリストは、あくまでも仮のものです。チームが (ソフトウェア優先で) 具体的な開発を開始して初めて、ほかの多くのリスクが発見されます。

別のライフ・サイクル・モデルによる衝突

反復型プロジェクトの管理者は、反復型開発を採用していない (または反復型開発の性質を理解していても) 管理職、顧客、請負業者などのほかのグループと自分の環境との間で、衝突が発生するのを頻繁に目にします。このようなグループは、主要なマイルストーンにおいて、完成して固定された成果物が出来上がることを期待しており、何回かに分けて要求をレビューすることを嫌がります。再作業に憤慨し、不完全なアーキテクチャー・プロトタイプの目的や価値を理解できません。彼らは反復のことを、目的もなく手探りし、技術をもてあそび、仕様が固定される前にコードを作成し、使い捨てのコードをテストしているものでしかないと考えています。

少なくとも、意図と計画をはっきりと目に見えるようにする必要があります。反復アプローチが、頭の中と、チームが共用するいくつかのホワイトボードの上にしかないのでは、いつかはトラブルに見舞われます。

プロジェクト管理者は、外部の攻撃と力関係からチームを守り、外部によってチームが混乱したり意気消沈したりすることがないようにしなければなりません。管理者は盾になる必要があります。「舵をしっかりと握っている」ためには、プロジェクト管理者は、プロジェクト管理者は、外部のコミュニティーとの信頼関係を築く必要があります。そのためには、可視性および「計画の追跡」がやはり重要であり、特に多くの人の目には異例のものに見える「計画」についてそのことが言えます。事実、実際の場ではさらに重要となることなのです。

問題点:異なるグループがそれぞれ固有のスケジュールで作業する

すべてのグループ (またはチーム、下請け) を同じフェーズと反復計画に従って作業させる方が、問題がなく簡単です。多くの場合、プロジェクト管理者は、個々のチームのスケジュールを微調整して時間を最適化し、結果としてチームごとに固有の反復スケジュールができあがります。これが実行されると、すべてのメリットはいずれ失われて、チームはより遅いグループに同期を取らされることになります。可能な限り、全員の進行速度が同じになるようにする必要があります。

問題点:方向づけの途中での価格決定

多くのプロジェクトは、方向づけの途中の早すぎる時期に、請負開発の価格決定に追い込まれます。反復的開発では、すべての関係者にとって最良の価格決定のタイミングは、LCA マイルストーン (推敲フェーズの終わり)です。このときに特別な方法はありません。利害関係者の調整と教育を行い、反復的開発の利点を示して、最終的には 2 段階の価格決定プロセスを行います。

進捗の評価が異なる

進捗を評価する出来高システムは多様です。 成果物は完成も固定もされていませんが、複数のインクリメントで再作業が行われるためです。成果物が出来高システムである値を持ち、作成した最初の反復でその値がマイナスになっている場合は、進捗の評価は楽観的すぎます。2 ～ 3 回の反復の後で成果物が安定したときにだけマイナスになる場合は、進捗の測定はひどく悲観的になります。そのため、このような方法で進捗を監視するときは、成果物を分解する必要があります。 例えば、初期文書 (40%)、第 1 版 (25%)、第 2 版 (20%)、最終文書 (15%) というように分けます。各文書に値を割り当てます。このようにすれば、各要素を完成させなくても、出来高システムを使用できます。

代わりの方法としては、反復自体に出来高を設定し、評価基準から値を測定します。この場合は、ステータス評価書で報告される中間の追跡ポイント (通常は月ごと) は、反復計画に基づいて設定されます。この場合、さまざまなユースケースやテスト・ケースなどの完成を追跡するので、従来の要求仕様や設計仕様などよりもきめ細かく成果物を追跡する必要があります。

Walker Royce 氏は次のように言っています。「プロジェクト管理者はテキストのページ数やコードの行数を数えるよりも、要求、設計、コードに対する変更の測定とモニターにもっと集中すべきです。」(下記の『参考文献』を参照してください)。また、Joe Marasco 氏は次のように付け加えています。「変更だけでなく繰り返しにも注目する必要があります。何回も変更されて最初と同じ状態に戻るものは、問題が深刻である徴候です。」

肯定的な面として、賢明なプロジェクト管理者は、早い時期に実際的なソフトウェアを動作させて、早い段階である程度信用できるものにすることができます。何百ものチェック・ボックスがある用紙に印を付けてレビューするより、意味のある方法で進捗を示すことができます。また、エンジニアは、「それがどのように動作するはずかを示す」より「それがどのように動作するかを示す」方を好みます。まずデモンストレーションを行ってから文書化します。

反復の回数、期間、内容の決定

まずすべきことは何でしょうか。反復型開発を初めて利用する管理者は、反復の内容を決定するのに苦労することがよくあります。まず、この計画立案は、技術的なリスクとプログラム上のリスク、開発中のシステムの基本要件または特性の重要性に基づいて行われます (RUP では、反復の回数と期間を決定するためのガイドラインが示されています)。基準もライフ・サイクルを通して発展します。計画立案は、作成では、特定の基本要件またはサブシステムの完成と関連付けられます。移行では、問題の解決および頑強性とパフォーマンスの向上と関連付けられます。

問題点:最初の反復に詰め込みすぎる

困難な問題に最初に取り組まないという問題点については既に触れました。一方で、これとは反対の方向に進みすぎることも失敗につながります。最初の 1 つまたはいくつかの反復で、あまりにも多くの問題に対処し、あまりにも広い範囲をカバーしようとする傾向があります。これではほかの要素を認知することができなくなります。チームを組織 (訓練) して、新しい技術を学習し、新しいツールを獲得することも必要です。また、問題領域が多くの開発者にとって初めて見るものであることがよくあります。その結果、往々にして、最初の反復で深刻な超過が発生し、反復アプローチ全体の信頼が失われる可能性があります。または、反復が途中で中断され、何も動かない状態で終了が宣言されます。ここからは基本的に何も教訓は得られず、反復型開発のメリットがほとんど失われた状態での「勝利」宣言です。

確実でない場合、または危機的な状態に直面した場合は、何らかの方法で規模を小さくします (問題、ソリューション、チームに適用します)。完全さはライフ・サイクル後半での考慮事項であることを思い出してください。初期のライフ・サイクルにおいて管理者が考慮することは「適切な不完全さ」です。最初の反復での目標が多すぎる場合は、2 つの反復に分割した後、最初に実現を試みる目標の優先順位付けを厳格に行います。

プロジェクトの初期には、より簡単で控えめな目標にする方がよいでしょう。これは簡単なことではありません。プロセスの初期に確実な結果を得ることは、士気の向上に役立ちます。最初のマイルストーンで失敗したプロジェクトの多くは、復元できません。後でどれほど努力しても、運命を変えることはできません。初期のマイルストーンを失敗しないように計画を立てる必要があります。

問題点:過剰な反復

まず、プロジェクトにおいて、毎日または毎週の作業と反復を混同しないでください。反復の計画立案、監視、評価には一定のオーバーヘッドがあるので、このアプローチに不慣れな組織では、最初のプロジェクトにおいて短期間での反復を試したりしないでください。反復の期間には、組織の規模、地理的分散の程度、関係する独立した組織の数も考慮する必要があります。「6 ± 3」の経験則をもう一度参照してください。

問題点:反復のオーバーラップ

もう 1 つの非常によく陥る問題点は、反復のオーバーラップを多くしすぎることです。GANTT 図で開始する場合、現在の反復の終わり 5 分の 1 位になった時点で次の反復の計画を開始し、アクティビティーをある程度オーバーラップさせる (つまり、現在の反復を終了してそこから学習する前に、次の反復の詳細分析、設計、コーディングを開始する) と、一見よいように思えますが、この方法では問題が生じます。一部のメンバーは、現在の反復に対するフォローアップや貢献が確実に完了できず、問題を迅速に解決できない場合があります。 または、次の反復になって初めてフィードバックを考慮す

ることになります。ソフトウェアの一部は、先に進んだ作業をサポートできなくなります。現在の反復に関係のない作業を実行するために一部のマンパワーを割くことはできますが、これは例外的なこととして最低限に留める必要があります。この問題は、組織のメンバーのほんの限られた一部のスキルによって、または非常に硬直した組織において頻繁に発生します。たとえば、Joe はアナリストで、彼ができること、またはしたいことはこれだけだとします。彼は、設計、実装、テストには加わりたくない、というような場合です。もう 1 つの否定的な例として、大規模な指示管制プロジェクトの反復が大きくオーバーラップしていて、基本的に、ある時点ではすべての反復が並行して実施されるものとします。 このような状況では、スタッフ全体をすべての反復に分散させる必要があり、前に行われた反復から後の反復に教訓がフィードバックされることも望めません。

よくある非生産的な反復パターンについては、図 3 を参照してください。

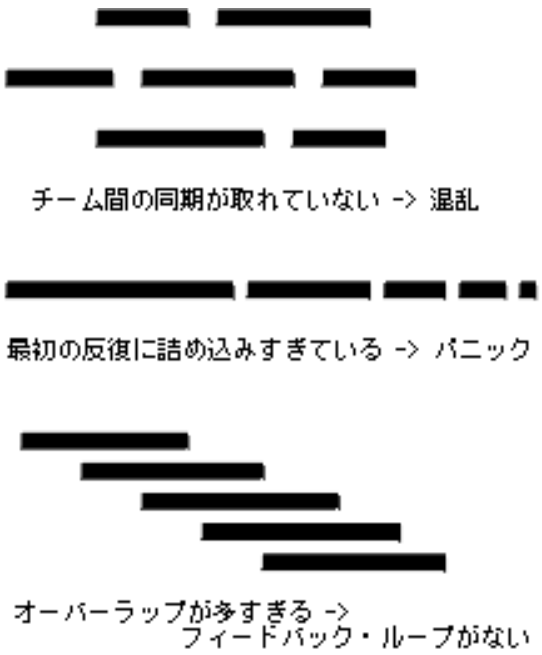

図 3:危険な反復パターン

優れたプロジェクト管理者と優れたアーキテクト

ソフトウェア・プロジェクトが成功するには、優れたプロジェクト管理者と優れたアーキテクトの両方が必要です。最善の管理と反復型開発が行われても、優れたアーキテクチャーがなければ製品開発は成功しません。逆に、素晴らしいアーキテクチャーであっても、プロジェクトの管理がうまくいかないと失敗してしまいます。したがって、これはバランスの問題であり、プロジェクト管理だけに重点を置いても成功につながりません。プロジェクト管理者は、アーキテクチャーを無視することはできません。初期の反復における決定事項の 20% に、アーキテクチャーの専門家と対象分野の専門家が携わります。

問題点:プロジェクト管理者とアーキテクトの兼務

同じ要員がプロジェクト管理者とアーキテクトの両方を担当しても機能できるのは、小規模なプロジェクト (5 ～ 10 人) においてのみです。大規模なプロジェクトでは、1 人でプロジェクト管理者とアーキテクトの両方のロールを担当すると、ほとんどの場合、そのプロジェクトは管理が適切に行われないか、またはアーキテクチャーに問題が出てきます。第 1 に、これらのロールに必要なスキルは異なります。第 2 に、これらのロールを兼務しては、フルタイムで仕事をしても足りません。したがって、プロジェクト管理者とアーキテクトは、毎日調整し、話し合い、歩み寄る必要があります。これらのロールは、映画のディレクターとプロデューサーの関係に似ています。どちらの仕事も同じ目標を目指しますが、事業の異なる側面についての責務を負います。同じ人間が両方のロールを担当したのでは、プロジェクトはまず成功しません。

結論

ここまで読んでくると、意気消沈しているのではないでしょうか。問題や陥りやすい失敗が多数待ち構えています。反復型開発の計画と実行がそれほど難しいことだとしたら、そのことで悩む必要はありません。嬉しいことに、すべての問題に系統立てて対処する秘訣と技法があります。 高品質のソフトウェア製品を確実に実現できれば、面倒な作業をしてでもはるかに大きなものが得られます。重要なテーマとしては、次のようなものがあります。「リスクを積極的に攻撃せよ。 そうしないとリスクに攻撃される。」 (「参考文献」で示されている Tom Gilb 氏の著書から引用。) ソフトウェアを優先します。スクラップと再作業があることを認識します。共同で作業するプロジェクト管理者とアーキテクトを選びます。反復型開発のメリットを活用します。

ウォーターフォール・モデルを利用すると、管理者にとっては状況が容易になりますが、エンジニアリング・チームには困難になります。反復型開発は、ソフトウェア・エンジニアの作業方法との整合性でははるかに優れていますが、管理面での複雑さという代価があります。ほとんどのチームでエンジニアと管理者の比が 5 対 1 (またはそれ以上) であるとするなら、これは大きな取り引きです。

反復型開発は、初めて利用するときには従来のアプローチより手が掛かりますが、長期的に見ればメリットが大きいことは確実です。要領をつかんでしまえば、自分が有能な管理者になったことに気付き、より大きく複雑なプロジェクトを従来より簡単に管理できることがわかるでしょう。チーム全体が反復型開発を理解し、反復的な考え方を身に着ければ、この方式は従来のアプローチよりはるかに適合性に優れています。

メモ:このホワイト・ペーパーを執筆するにあたって、John Smith 氏、Dean Leffingwell 氏、Joe Marasco 氏、Walker Royce 氏が、反復的プロジェクトの管理に関する経験を分け与えて、協力してくれました。この執筆の一部は、同僚の Gerhard Versteegen 氏が執筆したソフトウェア開発に関する新しい本の第 6 章からの抜粋です (「参考文献」を参照)。

著者について

Philippe Kruchten 氏は 1987 年に Rational に入社し、現在は ブリティッシュ・コロンビア州バンクーバーに本拠を置く Rational Fellow です。 以前はプロセス・ビジネス・ユニットのディレクター兼ゼネラル・マネージャーであり、Rational Unified Process の開発を率いていました。ソフトウェア・アーキテクチャーと設計に注目するだけでなく、ソフトウェア・エンジニアリングの実践原則と開発プロセスにも強く興味を抱いています。機械工学の学位とフランスの機関から与えられたコンピューター・サイエンスの博士号を持っています。

参考文献

1. "*Rational Unified Process 2000*", Rational Software, Cupertino, Ca., 2000
2. Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", *Computer*, May 1988, IEEE, pp.61-72
3. Tom Gilb, "*Principles of Software Engineering Management*", Addison-Wesley, 1988
4. Philippe Kruchten, "*The Rational Unified Process—An Introduction*", Addison Wesley Longman, 1999 (邦訳: 「ラショナル統一プロセス入門」日本ラショナルソフトウェア訳、藤井拓 監訳)
5. Walker Royce, "*Software Project Management—A Unified Approach*", Addison Wesley Longman, 1999 (邦訳: 「ソフトウェア・プロジェクト管理 - 21 世紀に向けた統一アプローチ」日本ラショナルソフトウェア 監訳)
6. Gerhard Versteegen, "*Projektmanagement mit dem Rational Unified Process*", Springer-Velag, Berlin, 2000.

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