XP の真髄

Java プロジェクトにおいてより大きな成功を収める方法

Roy Miller
Independent consultant

2001年 3月 01日

Christopher Collins
Senior Software Developer
RoleModel Software, Inc

Java 言語を使用したオブジェクト指向プログラミングの人気が非常に高まってきています。オブジェクト指向プログラミングは、ソフトウェア開発にある程度の革命をもたらしました。しかし最近の調査では、ソフトウェア・プロジェクトの 2 分の 1 は遅延し、また 3 分の 1 は予算オーバーをしているという結果がでています。問題はテクノロジーではありません。問題は、ソフトウェアを開発する方法にあるのです。Java 言語などのオブジェクト指向言語の能力および柔軟性は、一般に「ライトウェイト」または「アジャイル」と呼ばれる方法と組み合わされて、非常に興味深いソリューションを提供しています。アジャイル方法の中でもっともよく知られているのが、エクストリーム・プログラミングまたは XP と呼ばれるものですが、多くの人はこのプログラミング方法がどういうものなのかよく分かっていません。XP を Java プロジェクトで使用すると、プロジェクトが成功する確率は劇的に高まります。この記事では、XP の概要となぜこの方法が重要なのかを、うわさや誇長は一切含まずに説明します。

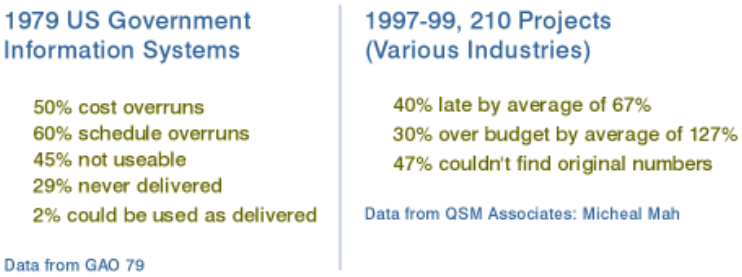
ここ 10 年程の間、さまざまな企業の CEO は、常に収益を向上させていかなければならないという大変なプレッシャーに直面してきました。そして、企業の CEO は人員削減、アウトソーシング、リエンジニアリング、エンタープライズ・リソース・プラニング (ERP)、その他の方法でその課題に対応してきました。このように非効率的な部分に取り組んでいくことで、S&P 500 に挙げられている多くの企業が 1990 年台後半の何年かの間は、2 桁の成長率維持することができました。しかし、これからはそうは行きません。

Gary Hamel は、著書 Leading the Revolution (参考文献を参照してください) で、「複数の指標が、従来のビジネス・モデルには、切り取る贅肉がほとんど残されていないということを示している。収益の成長を引続き維持していくには、私たちは何か別の方法を探さなくてはならない。」と主張しています。彼は、企業が成長を続けていくための唯一の方法は、根本的な革新であると提唱します。私たちは、これは特にソフトウェア開発の分野に当てはまると考えています。

ビジネスに関する問題

標準的なソフトウェア開発の採用を使用している場合、たとえ Java プラットフォームで開発を行っているとしても期待通りに行かない場合があります。図 1 のとおり、最近行った調査ではプロジェクトの 2 分の 1 は遅延し、3 分の 1 が予算をオーバーしていることが示されています。この結果は、政府の会計局が 1979 年に行った調査の結果よりもわずかに改善されているだけです。

図 1. ソフトウェア・プロジェクトの成功と失敗に関する過去と現在



これらの数値を大きく向上させるには、根本的な革新をソフトウェア開発の方法に取り入れることが必要です。現在の方法論は、2 つの主な要因によって左右されています。

- 失敗への恐れ
- ソフトウェアの本質に対する誤解

失敗を計画する人はいません。皮肉なのは、失敗を最小限にするために作成された方法は、失敗に陥りやすいということです。ソフトウェアについての間違った理解は、問題の原因となります。「恐れ」はまさにその兆候です。既存の方法論は、優秀な人々によって作成されたもので巧くまとめられています。しかし、彼らはソフトウェアの「ソフト」の部分を忘れていました。ソフトウェアの作成は、橋を組み立てるようなものだと思っていたのです。そのため彼らは、橋のような「ハード」なものに巧く適用できるさまざまなエンジニアリングの規則の中から、最高の方法を借りてきました。その結果、「大規模設計 (BDUF)」 精神に基づく無責任な開発が実践され、また役に立たない軟弱なソフトウェアが出来上がりました。

1 つのソリューション: アジャイル方式

最近、いわゆる「ヘビーウェイト」方法論から、クリスタル・メソッド、アダプティブ・ソフトウェア開発、そして (現在もっとも人気のある) XP といった「ライトウェイト」または「アジャイル」と呼ばれる方法論に動向が変わってきました。これらのプロセスはすべて、ソフトウェアを開発するためには複数の開発者が協力して作業しているという現実を取り入れています。成功するソフトウェア・プロセスは、開発者の長所を最大化し、かつ短所を最小化するようなものでなくてはいけません。長所と短所は必ず存在します。私たちは、プロジェクトに携わる開発者に影響を与えるすべての補足的効果を取り込んでいるという点で XP の内容が最も優れていると考えています。

XP は、ソフトウェア開発のプロセスに根本的な革命を起こすための、この 10 年間で唯一最大の好機です。ピープルウェアの著者、Tom DeMarco はこう言っています。「XP は今日の我々の分野にとって最も重要な動向です。私は、過去の世代にとって SEI とその CMM (Capability Maturity

Model) が不可欠であったのと同じように、XP が現在の世代にとって重要なのものになるだろうと予想しています。」

XP は、ソフトウェア開発者の仕事"コードの作成"が、最高のものになるために、コアとなる価値とプラクティス (実践術) のセットを規定しています。XP は、ほとんどの「ヘビーウェイト」プロセスが持つ不要な成果物を取り除きます。不要な成果物とは、開発スタッフをスローダウンさせたり疲れさせたりすることで目標から遠ざけるもののことで、たとえばガントチャート、状況レポート、および何巻にも及ぶ要件資料などがあります。「エクストリーム・プログラミング」と呼ばれるものは、本格的な開発プロセスとして、企業のマネージメントに売り込むのは難しいかも知れません。しかしソフトウェア・ビジネスに従事している企業であるならば、その名称はさておき、XP が提供する競争力の強化という利点をマネージメント側に理解してもらうことは必要であると私たちは認識しています。

Kent Beck は、著書エクストリーム・プログラミング入門 (参考文献を参照) で XP のコアとなる価値について概説しています。私たちはそれを以下のように要約してみました。

- **コミュニケーション** プロジェクトにおける諸問題の原因は、誰かがある時点で重要なことを誰にも連絡していなかったことに起因することがよくあります。XP では、コミュニケーションを取らないということはほとんど不可能です。
- **簡潔さ** XP は、プロセスおよびコード作成に関して、常に可能な限り簡潔な方法を取ることを提案しています。Beck はこう言っています。「XP は賭けと同じです。今日は結局使わないかも知れない複雑なことをするよりも、今日単純なことをした方がよい、ということに賭けるのです。」
- **フィードバック** カスタマー、チーム内、また実際のエンド・ユーザーからの早期の具体的なフィード・バックを得ることにより、開発者は作業の方向確認がし易くなります。フィードバックによって、無駄骨を折らずに計画どおりに進めることができます。
- **勇気** 勇気は、上記の 3 つの価値の中に含まれています。これらは互いに支え合っています。作業の途中で返ってくる具体的なフィードバックの方が、最初にすべてを把握しようとするよりも適切であると信じるには勇気が要ります。自らの無知を露呈してしまうかも知れないようなことを、チームの他のメンバーに相談することにも勇気が要ります。また、システムを単純な状態に維持して、明日の決定を明日まで据え置くのも勇気が必要なことです。そして、単純なシステム、知識を広げるための絶えざるコミュニケーション、および、方針を決めるためのフィードバックなしでは、勇気をもつことは困難です。

XP のプラクティスは、これらの価値を開発者として日々なすべき作業に言い換えたものです。これには、新しいことは何もありません。XP のプラクティスは、ここ数年業界で「もっとも優れたプラクティス」であると認識されてきました。事実、「エクストリーム (極端な)」という言葉は、以下の 2 つの事柄に由来しています。

- XP は証明済みの業界最高のプラクティスを選び、それぞれの調節バーを 10 まで引き上げました。
- XP はこれらのプラクティスを、単なる各パーツの「和」以上の内容を製作するできるような方法で組み合わせます。

それは、どのようなものでしょうか。コードをレビュー (再検討) するのは良いことなので、コードを作成する際には常に対となる作業として行ってください。テストも良いことなので、

コードを作成する前にはまずテスト・コードを書くようにし常にテストを行ってください。ドキュメンテーションはコードとはほとんど同期しないので、これは必要最低限のみ行い、あとは明確に書かれたコードおよびテストを信頼するようにします。XP は、開発者がいつも適切な作業を行うことを保証するものではありませんが、XP を使用して適切な作業を行うことは可能です。XP は、「エクストリーム (極端)」なプラクティスが、互いにサポートし合い、スピードと効率性を大幅に改善できるような方法で、これらを組み合わせます。

XP の 12 のプラクティス

XP の 12 のプラクティス (図 2 参照) は、XP を規則として定義しています。「XP する」とはどういうことかをよく理解するために各プラクティスを詳しく見ていきましょう。

図 2. XP の 12 のプラクティス

The Planning Game
Pair programming
Testing
Refractoring
Simple design
Collective code ownership
Continuous integration
On-site customer
Small releases
40-hour week
Coding standards
System metaphor

計画ゲーム(The Planning Game)

XP は、ハッキングを美化したものであって、カウボーイ集団が規律を持たずに、システムを作っているにすぎないと批判する人々もいます。それは違います。XP は、始めの時点からすべてを知ることはできないということを認識している、数少ない方法論の 1 つです。顧客と開発者の双方が、プロジェクトが進むに連れてさまざまなことを学んでいきます。将来このような変化を奨励しかつ取り込んでいく方法論のみが、有効になるでしょう。旧方法論は変化を無視します。XP は耳を傾けます。XP は、Kent Beck による造語「計画ゲーム (planning game)」という概念に基づいて耳を傾けます。

このプラクティスを支える主な考え方は、短時間で大まかな計画を立て、あらゆることが明確になるに従い、それをさらに精密化していくことです。この計画ゲームが作り出すものは以下のとおりです。1) インデックス・カードの束。各カードには、カスタマー・ストーリーが書かれていて、これによってプロジェクトの反復が行われます。2) 次回の、またはさらにその次のリリースの大まかな計画。これについては、Kent Beck と Martin Fowler の共著 Planning Extreme Programming (参考文献を参照) に説明があります。このスタイルを成功させるための決定的な要因は、ビジネス上の意思決定は顧客に委ね、技術的な決定は開発チームが行えるようにすることです。これが実現しなければ、プロセス全体が失敗に終わってしまいます。

開発チームが決定することは以下のとおりです。

- 1 つのストーリーを開発するのにかかる時間の見積もり
- さまざまな技術オプションを使用する際のコストの推定
- チーム編成
- 各ストーリーの「リスク」
- 1 つの反復工程における、ストーリー開発の順序 (リスクの高いものから始めることにより、リスクを軽減できます)

顧客が決定することは以下のとおりです。

- 目標範囲 (あるリリースに対するストーリー設定と、各反復工程に対するストーリー設定)
- リリース日
- 優先順位 (ビジネスにおける有用性に基づき、どの機能を先に開発するか)

計画は頻繁に行います。それによって、顧客および開発者の双方にとって、新しいことを学びながら計画を調整していく機会が頻繁に得られます。

ペア・プログラミング(Pair programming)

XP では、すべての実稼働用コードはペアになった二人の開発者の組によって作成されます。この方法は、非効率的に聞こえるかも知れません。Martin Fowler は、「『ペア プログラミングは、生産性を低下させるものだ』と言われる度に、私は『プログラミングで最も時間を食うのがタイピングだった場合には、そのとおりでしょうね。』と答えています。」と語っています。実際、ペア プログラミングは、経費の面でも他の面においても多くの利点を提供しています。

- すべての設計に関する決定に最低でも 2 つの頭脳が関わっている
- システムのすべての部分について、最低 2 人の開発者が熟知している
- その 2 人の両方がテストあるいは他のタスクを怠る可能性は低い
- ペアを変えることによって、チーム内で知識を広めることができる
- コードは常に、最低 1 人によってレビューされることになる

リサーチによっても、ペアでプログラミングを行うことが、単独で行うよりも効率的であることが示されています(詳細は、Alistair Cockburn と Laurie Williams の共著、The Costs and Benefits of Pair Programming を参照してください)。

テスティング(Testing)

XP には以下の 2 つのタイプのテストがあります。

1. 単体テスト
2. 受け入れテスト

開発者は、コードを作成する際に単体テストを作成します。顧客は、ストーリーを定義した後に、受け入れテストを作成します。開発者は単体テストによって、そのシステムがある任意の時点で「動作する」かどうかを確認することができます。受け入れテストは、チームに対しシステムがユーザーの希望どおりに動作するかどうかを知らせます。

チームが、Java などのオブジェクト指向言語を使用していると想定してください。開発者は、うまく動作しないかもしれないすべてのメソッドに対して、メソッドのコード作成する前に、単体テスト・ケースを作成します。それから、テストにパスするためにちょうど必要なだけのコードを作成します。開発者はこれを少し変わったやり方だと思うこともあるでしょう。これを行う意味は簡単です。テスト・ケースを先に書くことにより以下が得られます。:

- 可能な限り完璧なテストのセット
- 正しく機能する最も単純なコード

- コードの目的に関する明確なビジョン

開発者は、すべての単体テストにパスするまでは、ソース・コード・リポジトリーにコードをチェックインすることはできません。単体テストによって、開発者はコードが正常に機能していると確信することができます。またこのような証跡を残すことにより、他の開発者がオリジナルの開発者の目的を理解することができます(事実、これこそ私たちが今までにみた最高のドキュメンテーションです)。単体テストはまた、開発者にコードのリファクタリング (改善のための再構成) を行う勇気も与えます。なぜなら、テストが失敗すればすぐに何かが失敗していることが分かるからです。単体テストは自動化されている必要があり、かつ「パス」か「失敗」か、結果を明確に示すものでなければなりません。xUnit フレームワーク (参考文献を参照)では、この両方とさらに多くのことが行えます。そのためほとんどの XP チームは、xUnit フレームワークを使用しています。

顧客は、各ストーリーに必ずそれを検証するための受け入れテストが作成されていることを確認する責任があります。顧客自身がテストを書くことも可能です。自分の企業から誰か (たとえば、QA (品質管理) 担当者かビジネス・アナリスト) を起用してテストを作成することも、その 2 つの方法を組み合わせることもできます。これらのテストによって、顧客はシステムが意図したとおりの機能を備え、かつそれらが正しく機能しているかどうかを知ることができます。理想的なのは、開発の各反復工程が終了する前に、顧客がその反復におけるストーリーの受け入れテストを用意することです。受け入れテストは自動化し、頻繁に実行して、開発者が新しい機能をインプリメントする際に既存の機能を壊すことがないようにする必要があります。一般に、顧客が受け入れテストを作成する際には、開発チームのヘルプを必要とすることになるでしょう。あるプロジェクトで、私たちは、再利用可能な自動受け入れテスト・フレームワークを開発しました。これを使用すれば、顧客は簡単なエディターに入力データと予期出力データを入力することができます。フレームワークは、入力データを XML ファイルに変換し、そのファイル内でテストを実行し、「パス」または「失敗」をそれぞれのテストに対して出力します。これは顧客に大変気に入られています。

必ずしも、常にすべての受け入れテストがパスする必要はありません。大切なのは、受け入れテストによって顧客がプロジェクトの進行具合を評価できるようにすることなのです。さらに受け入れテストによって、顧客は、ある製品がリリース可能かどうかを豊富な判断材料を持った上で決定することができます。

リファクタリング(Refactoring)

リファクタリングは、機能を変更することなく、コードを改善する技法です。XP チームは、容赦なくリファクタリングを行います。

開発者がリファクタリングを行うための主な機会は 2 回あります。それは、機能をインプリメントする前と後です。開発者は、既存のコードを変更することによって、新しい機能のインプリメンテーションが容易になるかどうかを判別します。今作成し終わったコードについて、これ以上簡潔にする方法がないか調べます。たとえば、抽象化できるような箇所が見つかったら、実際のインプリメンテーションから重複しているコードを除去するためにリファクタリングします。

XP は、正しく機能するもっとも簡潔なコードを書くように指示しますが、同時に、それにより学んでいくことにもなると言っています。リファクタリングは、テストをパスしつつ、学んだことをコードに反映していくことを促します。これによって、コードをクリーンに保つことができま

す。つまり、コードの寿命が長くなり、将来の開発者にもたらす問題が減り、また彼らを正しい方向に導くことができるのです。

シンプル・デザイン(Simple design)

XP を非難する人たちは、このプロセスが設計を怠っていると主張しています。これは正しくありません。問題は、「ヘビーウェイト」的方法は、最初の時点でほとんど意味のない設計タスクを行うように指示していることです。これではまるで、地平線の静止写真を撮って、じっと動かずにいて、目的地に辿り着くための地図を書こうとするようなものです。XP は、変更がないという幻想に基づき最初の時点で設計を一度に行なうようなことは、してはいけないと主張します。XP は、設計は非常に重要であり、絶えず行っていくものだと考えています。私たちは、日々発生する現実を反映するために設計を変更しつつ、どの時点においても常に正常に機能する最も簡潔な設計を使用するように努めています。

正常に機能する最も簡潔な設計とは何か。それは、以下のような設計です (このリストを提供してくれた Kent Beck に感謝します)。

- すべてのテストを実行する
- 重複するコードを含まない
- すべてのコードに対し、プログラマーの意図を明確に述べている
- 可能な限りの少ないクラスおよびメソッドしか含まない

簡潔な設計を要求してもそれは、すべての設計が小さかったり平凡であったりすることを言っているのではありません。それは、可能な限り簡潔でかつ正常に機能するものであればいいのです。使用されない余分な機能は含まないようにしてください。私たちはこのような機能を、YAGNI、つまり「将来必要に『ならない』もの (You Aren't Going to Need It)」と呼んでいます。どうか、この YAGNI にあなたの成功のチャンスを潰されないようにしてください。

コードの共同所有権(Collective code ownership)

チームの誰もがコードを改善するために変更を加える権限を持つ必要があります。全員がすべてのコードを所有する、すなわち全員がコードに対し責任を持つことになります。この技法では、開発者は、個人のコード所有者を気にすることなくコードを変更することができます。全員が責任を持つという事実により、コードが誰にも所有されずに混乱状態が起こることもありません。

全員がすべてのコードを所有すると言うことと、誰もコードを所有していないと言うこととは同じではありません。誰もコードを所有していなければ、開発者は勝手気ままにコードを破壊しても責任を負いません。XP ではこう言います。「壊したら自分で直せ (You break it, you fix it)」私たちは、必ずすべてのインテグレーションの前と後に単体テストを実行します。もし、何かを壊した場合、それがコードのどの部分であっても、修正するのは壊した人の責任です。これは、極端な規律を必要とします。おそらく、このことが「エクストリーム」という名前を持つもう一つの理由でしょう。

継続的インテグレーション(Continuous integration)

頻繁にコードのインテグレーションを行うことにより、インテグレーションの悪夢を避けることができます。XP チームは、システムを実行するためのすべての単体テストが終わった後に、一日数回コードをインテグレートします。

従来の方法における作業は、「コードをたくさん作成し、大規模のインテグレーションを行い、その後問題の修正に非常に長い時間をかける」という傾向があります。このようなおかしなやり方は、確実にプロジェクトをスローダウンさせます。大規模なインテグレーションは、チームに多くの問題を一度にまとめて提出します。しかも、これらの問題には考えられる原因が数え切れない程あります。

インテグレーションを頻繁に行うと、特定のインテグレーションの障害の原因はより明白になります(テストはすでに行われているので、原因はそれ以降のものにあるはずです)。この方法で行けば、問題に遭遇した際、原因の可能性をかなり限定することができます。修正もより簡単で短時間で行うことができ、それによりチームは最高のスピードで仕事を進めることができます。

オンサイト顧客(On-site customer)

最適に機能するために、XP チームは、顧客にオンサイトでストーリーを明確化し、重要なビジネス上の意思決定をしてもらう必要があります。開発者だけでこれを行うことは許されていません。顧客にその場で参加してもらうことで、顧客の決定を待たなければならない場合に生じるボトルネックを排除することができます。

XP は、ストーリー・カードだけが開発者が必要なコードを作成するための指示であると言い張るつもりはありません。ストーリーは、後で顧客と開発者が詳細を具体化するために使用する決め事のようなものです。ここでの狙いは、静的な文書にすべての要件を書くよりも、対面のうえでコミュニケーションを取る方が誤解を最小限に抑えるとこができるということです。

私たちは、顧客にオンサイトで参加してもらうのが考えられる最も良い状況だと理解していますが、これだけが成功する唯一のシナリオではありません。重要なのは、顧客が、ビジネスの有用性に基づいていつでもチームに対して質問に答えたり、指示を与えたりすることが可能でなければならないということです。顧客がフルタイムでチームとオンサイトにいなくても、このようなことができるのであれば、それは素晴らしいことです。ただ、実際にチームと同席している方が簡単なので、この方法がお勧めです。

小規模リリース(Small releases)

リリースは可能な限り小規模なものにし、その一方で高い価値を持ちビジネスにおける有用性を備えている必要があります。

システムは妥当だと判断されれば、すぐにリリースしてください。これによって、少しでも早くその価値を顧客に提供することができます(今日の金は、明日の金より価値があるということを覚えておいてください)。さらに、小規模なリリースによって、開発者は何が顧客のニーズを満たしていて、何がそうでないのかに関して具体的なフィードバックを得ることもできます。そして、チームはこのようにして得たその教訓を次のリリース用の計画に含めることができます。

週 40 時間(40-hour week)

Kent Beck はこうありたいと言っています。「毎朝、フレッシュでやる気にあふれ、毎晩、疲れて満足している」週 40 時間労働ではこれが実現できます。40 という数字は重要ではありません。重要なのは、その信条です。油を長時間燃やすとその火力は衰えます。疲れた開発者は通常よりも多くのミスを犯します。その結果、長期的には「ノーマル」なスケジュールに沿って行うよりも作業が遅くなってしまいます。

たとえ、開発者がもっと長い時間有効に作業することができたとしても、そうする必要はありません。結局、彼らは、仕事にうんざりして仕事をやめてしまいます。でなければ、仕事とは関係のない問題に悩まされ、それがパフォーマンスに影響を与えることになります。開発者の生活を邪魔すると、後でしっぺ返しが来ます。残業は、プロジェクトの問題の解決策ではありません。事実、残業はさらに大きな問題の兆候なのです。死の行進が見たければ、開発者を絞り上げることです。

コーディング標準(Coding standards)

コーディング標準を持っていると以下ことをしていることになります:

- 重要でないことに関するくだらない議論によってチームの気が散ることがなくなり、最大スピードで作業を行うことができる。
- 他者のプラクティスをうまく取り入れる。

コーディング標準がないと、コードのリファクタリングが難しくなりますし、必要なときにペアを変更したり、迅速に作業を進めていくことも難しくなります。目標は、チームの中で誰がどのコードを書いたのか区別がつかなくなることです。チームとして標準に合意し、あとはそれに従うのみです。目標は、徹底的な規則のリストを作ることではなく、コードが明確に情報を伝えることができるようにするための指針を提供することなのです。コーディング標準はまず単純なものから始め、その後チームの経験に基づいて徐々に発展させていけばよいのです。最初の時点で、これに大量に時間を費やさないようにしてください。正しく機能する最も単純な標準を作成して、作業を開始してください。

システム・メタファー(System metaphor)

アーキテクチャーは、何のためにあるのでしょう。それは、システムのさまざまなコンポーネントとそれらがどう相互作用しているのかを示す図面、つまり開発者に新しいコンポーネントがどこに適合するのかを示す地図のようなものです。

XP のシステム・メタファーは、ほとんどの方法論でアーキテクチャーと呼ばれているものに類似しています。メタファーは、既存のシステムが機能する方法、新しいパーツが適合する場所、およびそれらがどのような形式を取る必要があるのかについて説明するために使用できる、矛盾のない図面をチームに提供します。

全員にシステムがどのように組み合わさっているのかを理解させることが重要なのであって、美しいメタファーを作ることが重要なのではありません。よいメタファーが思い浮かばないこともあります。思いついたときは、最高の気分になります。

プラクティスは相互作用する

全体は、個々の和よりも大きくなります。個々のプラクティスを、あるいはその小さなサブセットを実行し、それらを使わない場合に比べれば大きな利点を得ることはできます。しかし、利点を最大に活かせるのは、すべてのプラクティスを実行した場合のみです。それは、プラクティスのパワーは相互作用によって得られるものだからです。

最初は、ベンチマーク・テストとして、本を参考にしながら XP を行ってください。一度、プラクティスがどのように相互作用するかを理解すれば、自分のコンテキストでそれを適用するための

必要な知識が身につきます。「XP する」ことが目標ではないということを忘れないでください。これは目標のための手段です。目標は、優れたソフトウェアを迅速に開発することです。もし、あなたのプロセスが突然変異して、XP をしているとは言えないような状況になったとして、それでも競争相手が吹っ飛んでしまう程の結果が出せたとしたら、あなたは成功したのです。

なぜ XP が重要なのか

率直に言って、使用するのが XP (または他のアジャイル方法論) であることは重要ではありません。肝心なのはそれが生み出す結果です。XP のようなアジャイル方法が、より優れたソフトウェアを迅速にかつあまり苦しまずに作成するために役立てば、これは検討に値します。

この記事の冒頭で述べた、恐るべきいくつかの数字を思い出してください。私たちは、これらの問題を改善できる可能性が一番高いのは、XP によるオブジェクト指向ソフトウェアの開発であると信じています。今のところ、この私たちの確信は経験によって確認されています。

著者について

Roy Miller

Roy W. Miller は最初に Andersen Consulting (現在の Accenture) にて、10 年以上技術コンサルタント、ソフトウェア開発者、および指南役を務め、ノースキャロライナの RoleModel Software 社にてほぼ 3 年を過ごしました(ここでは彼は、エクストリームプログラミング (XP) を使用した Java 言語アプリケーションの構築に注力しました)。現在、彼は独立コンサルタントであり指南役です。彼は、XP を含む重要なメソッドおよび機動的なメソッドを使用しており、Addison-Wesley XP Series (Extreme Programming Applied: Playing to Win) 中で本を共同執筆しました。彼の最も最近の本である Managing Software for Growth: Without Fear, Control, and the Manufacturing Mindset は、どのように、複雑な技術がソフトウェア開発を支援することができ、そして他の IT マネージャー達が、プログラマをコントロールしたり枯らすことなく、彼らのチームが実際の人々が使用して楽しいような素晴らしいソフトウェアを作成することの、支援方法を理解するかを述べています。

Christopher Collins

Christopher T. Collins は RoleModel Software, Inc. のシニア・ソフトウェア・デベロッパーです。RoleModel で Chris は、2 年近くかかった XP プロジェクトに参加し、新しい Motorola 携帯電話用プラットフォームのための組み込み Java アプリケーションを開発し、Sun の J2ME プラットフォームで実行される JUnit を移植しました。RoleModel 社に入る前は、5 年間いくつかの組織のためにさまざまな言語を使用してソフトウェアを開発していました。最近では、米国国防省のためのアプリケーションを手がけています。Chris は、複数の異なる方法論を使用、および適応させた経験があり、RUP と XP を含むアジャイルとヘビーウェイトの両方の経験もあります。Chris は、West Florida 大学でコンピューター・サイエンスとソフトウェア・エンジニアリングの理学修士号を取得しました。現在は、North Carolina 州立大学の、Java プログラミング・コースで教鞭をとっています。また Duke 大学では XP についての招待講演者として活躍しており、XP2001 ではプロセス・アダプテーションについての論文を発表する予定です。