解説 1

アジャイル開発のスケールアップに向けて

ー大規模・分散プロジェクトでもアジャイル開発のメリットを得るためにー

市場の変化に迅速に対応して「モノづくり」を行うアジャイル・プロセスを用いたソフトウェア開発（以下、アジャイル開発）に対する期待が、米国のみならず、日本国内でも高まってきています。

小規模で同一地点にいる開発チームにアジャイル開発を適用するのは比較的容易であり、これまでも多数の実績が出ています。しかし、適用規模が大きい、地理的に分散している、アプリケーションが複雑である、などの状況に応じて、アジャイル開発の適用の難易度は増し、工夫が必要となっています。

本記事では、アジャイル開発をスケールアップさせる上での課題、工夫のポイントを整理し、実際に製品開発の事例を取り上げて具体例を用いて解説します。

Article 1

Scaling Up Agile Development

- To Take Advantage of Agile Development Even in Large-scale Distributed Projects -

Expectations for software development using Agile methodology, where a team can rapidly respond to the requirement changes in the market, have increased recently both in the US and Japan.

The adoption of agile methodology to a small and collocated team is relatively easier and more successful. However, if a project team is bigger, more distributed and/or the application is more complicated, the adoption of the agile methodology is more challenging and needs more effort.

In this article, the authors describe issues and practices for scaling software agility. They also clarify those points by elaborating on the real cases of a software development project in IBM.

❶ アジャイル開発への期待度の高まり

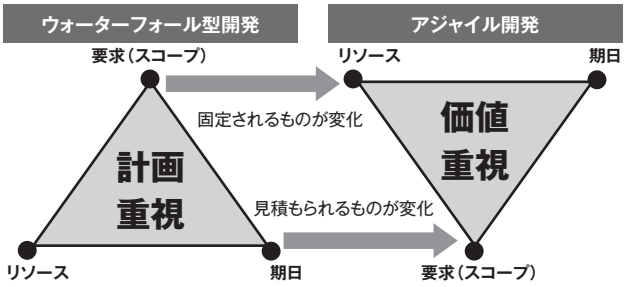
あらゆる産業において市場の競争が激しくなっており、経営戦略の寿命自体が短期化してきています。市場ニーズへ俊敏に対応していくためにも、差別化のための源泉となるソフトウェア開発の能力がますます重要になってきています。そのような状況で、市場の変化に迅速に対応して「モノづくり」を行うアジャイル開発に対する期待が、米国のみならず、日本国内でも高まってきています。

米国では、アジャイル開発がメインストリームになりつつあるという統計や見解も出てきています［1］。日本でも2010年に入りIPA（情報処理推進機構）が「非ウォーターフォール型開発に関する調査」を発表するなど、非常に注目を集めています［2］。

IBMでは、すでに製品開発部門、社内アプリケーション開発部門においてアジャイル開発を全社的に推進しており、開発リード・タイムの削減、顧客満足度の向上、コスト削減などにおいて大きな成果を上げています（本誌4ページ以下: マネジメント最前線①参照）。

❷ アジャイル開発の特徴

ここではアジャイル開発の特徴を見てみましょう。まず、アジャイル開発の方法論には、さまざまなものがあり（代表的なもので、XP［3］、Scrum［4］など）、それぞれの方法論がプラクティスと呼ばれる技法を持っています。そのプラクティスの中には、方法論に共通のものもありますが、独自のものも存在しています。2001年には、代表的な方法論者が集まり、アジャイル・アライアンスを組織し、アジャイル・マニフェストが発表されました［5］。このマニフェストでは、「プ



アジャイル開発では、要求（スコープ）を固定せずに、管理する。

図1. アジャイル開発のパラダイムシフト［6］

ロセスやツールよりも個人と対話を」「包括的なドキュメントよりも動くソフトウェアを」「契約交渉よりも顧客との協調を」「計画に従うことよりも変化への対応を」の重要性が説かれ、アジャイル方法論に共通の基盤と理解を与えるのに貢献しました。

アジャイル開発は、初期の XP に代表されるような、ペア・プログラミングやテスト駆動型開発などのプログラマー主体のプラクティスのイメージが強いですが、その本質はビジネス視点の価値を重視した考え方が根底にあります。**図 1** の左側にあるように、ウォーターフォール型開発においては、

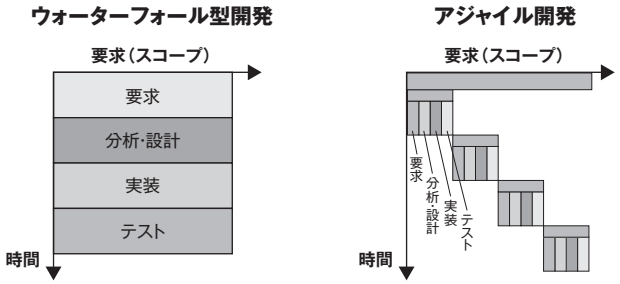


図2. ウォーターフォール型開発とアジャイル開発のプロセスの比較
出典:[7]より一部改変

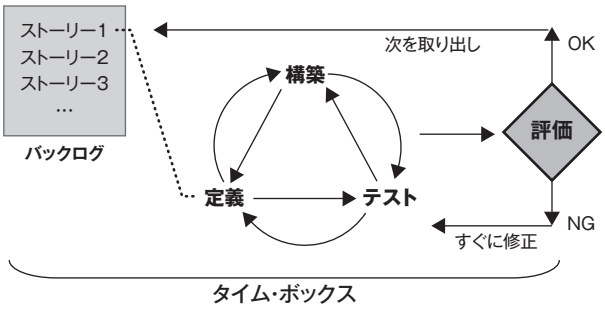


図3. タイム・ボックスで動くソフトウェアを作成[6]

プロセス	ウォーターフォール型開発	反復型開発	アジャイル開発
成功の測定	計画への適合	⟷	変化への対応 動くコード
マネジメント文化	命令と制御	⟷	リーダーシップ 協業的
要求と設計	大きくて前払い	⟷	継続的・発現的 ジャスト・イン・タイム
コーディングと実装	全機能を同時にコーディング 後でテスト	⟷	コーディングとユニット・テスト 順番に納品
テストと品質保証	大きく、計画に基づく遅くテスト	⟷	継続的・同時発生 早期にテスト
計画とスケジューリング	PERT・詳細・範囲固定 時間と工数を見積もり	⟷	2レベル計画・期日固定 範囲を見積もり

図4. ウォーターフォール型開発とアジャイル開発の比較[6]

一般的に、まず要求（スコープ）を固定させてリソースと期日を見積もる、というやり方をします。右側のアジャイル開発では、この三角形が逆向きになっているのに注目です。まずリソースと期日を固定させた上で、その前提条件の中で、どう作れば一番価値の高いシステムが作れるか、という考え方をするのです。こうすることで、限られた資源の中で、何から順番に作ればいいか、何を省けばいいか、という思考に至ります。

この考え方を踏まえて、アジャイル開発の各方法論に共通するポイントを、以下に簡単に説明してみましょう。まず反復型かつ漸進型の開発ライフサイクルが特徴です。

図 2 の左図のように、ウォーターフォール型開発では、要求定義、分析・設計、実装、テストといった各フェーズを順番に完成させて、戻ることなく開発する、という流れを取り、できるだけ手戻りを起こさないことを重視するライフサイクルです。しかし、ウォーターフォール型開発には、プロジェクト初期で要求が完全に定義できない、要求が後に変更してしまう、実装後にシステム全体を統合した際のリスクが後送りされてしまう、などのリスクが指摘されています。

それに対し、**図 2** の右図に示すアジャイル開発では、プロジェクト初期に、要求全体のスコープは抑えますが、詳細まで完全に定義をするのは後回しにします。2 ～ 6 週間に固定された期間（反復と呼ぶ）を繰り返し、毎反復で、動くソフトウェアを作成することを重視します。この際に、ビジネス、アーキテクチャー、リスクの視点から優先順位を決め、その順位の高い要求から作っていきます。こうすることで、変化する要求を受け入れるタイミングを確保し、統合リスクを削減し、価値重視のソフトウェア開発を行う、などのメリットがあります。

さらに、個々の反復中を、**図 3** を用いて見てみましょう。アジャイル開発では、バックログという概念がありますが、こちらは、ユーザー視点の小さな要求の断片（ストーリーと呼ぶ）を優先順位付けして並べたものです。開発チームは 10 人以下の小規模チームで、チームとして要求定義、構築、テストができるスキルを兼ね備えたチームで開発します。この開発チームが、バックログから優先順位の高いストーリーを取り出し、動くソフトウェアを作り出し、評価をして問題がなければ次のストーリーに取り掛かります。小さなロットで、追加的に要求を付け足していくイメージです。こうすることで、タイム・ボックス（固定された期間のこと）の中で、できるところまでストーリーを片付けていき、最終日が来た段階で、できたところまで（しかし結果的に一番価値の高いものとなっている）のソフトウェアを出荷することができます。最後に、参考までに、ウォーターフォール型開発とアジャイル開発を比較した表を**図 4** に示します。

❸ アジャイル開発のスケールアップ

これまで、アジャイル開発の特徴を述べてきましたが、小規模で同一地点にいる開発チームにアジャイル開発を適用するのは、比較的容易でこれまでもたくさん実績が出ています。しかし、適用規模が大きい、地理的に分散している、アプリケーションが複雑である、などさまざまな状況に応じて、アジャイル開発の適用の難易度は増し、工夫が必要となっています（**図 5**）。

まず、代表的なアジャイル・プラクティスのうち、実質的に大規模開発に不向きなものがあります。例えば、「同一地点での開発」「顧客もチームの一部にする」といったプラクティスはプロジェクトによっては実現が困難でしょう。また、既存の開発スタイルに慣れている組織にとっては、「新しい」やり方を導入することが難しいでしょう。特に、契約体系、既存の資産の存在、役割別の縦割り組織など、アジャイル開発の典型的な弊害となります。

IBM Rational® においては、Disciplined Agile（もしくは Agility@Scale）というイニシアチブの下に、アジャイル開発のスケールアップに積極的に取り組み、ベスト・プラクティスやサポートするツールを提供しています［8］。

ここで、アジャイル開発のスケールアップをより理解するために、まずは、規模に関係なく、アジャイル開発において重要かつ共通なプラクティスについて見てみましょう。

- 「定義・構築・テスト」コンポーネント・チーム
- 反復型開発
- 頻繁な小規模リリース
- 2 レベル計画作りと追跡
- コンカレント・テスティング
- 継続的インテグレーション
- 継続的な考察と適応

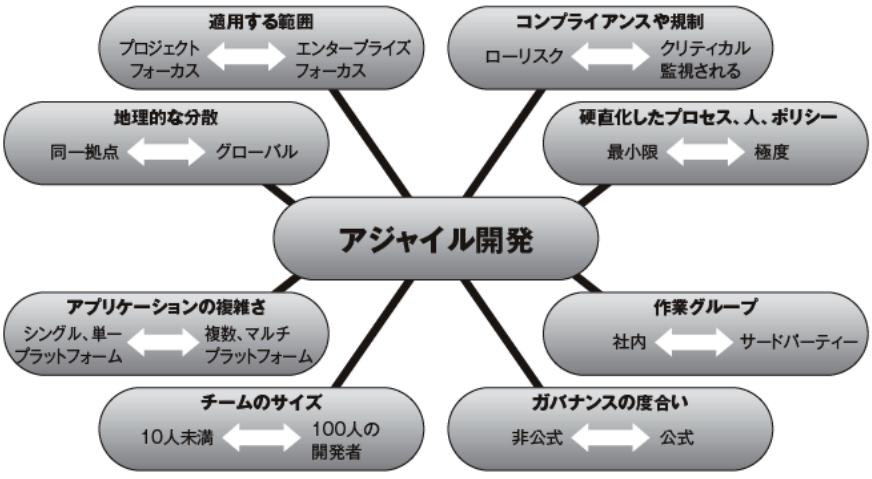


図5. スケールアップする際の考慮点

これらは、小規模な開発でも重要かつ共通なプラクティスといえます。さらに、ある程度規模のある開発においては、上記以外にも、特に下記のポイントが重要になってきます。

- 意図的にアーキテクチャーを作成する
- ジャスト・イン・タイムの要求定義
- 分散した開発チームの運営

次の章では、実際に実開発事例を用いて、規模に関係なく適用できるプラクティス、そして、大規模な開発で必要な考慮点について解説を加えていきます。

❹ IBM におけるアジャイル開発適用例

4.1 IBM Rational Team Concert™ 開発事例の紹介

ここ数年の間に IBM でのソフトウェア製品開発でも、アジャイル開発を適用することがごく当たり前のことになってきています。多くの開発は数十人から百人規模のものであり、また複数の地理的に離れた開発拠点からマネージャーや開発者が参加して開発を行っています。つまり、大規模かつグローバルな開発に対してアジャイル開発を実践していることになります。ここでは実際にアジャイル開発を適用して製品を数回リリースしている開発プロジェクト Jazz™ の中から、IBM Rational Team Concert（以下、チームコンサート）の開発を例に、前述したプラクティスがどのように行われているかをご紹介します。

4.2 規模に関係なく適用できるプラクティス

4.2.1「定義・構築・テスト」コンポーネント・チーム

アジャイルにおいては 10 人規模の小規模なチームが開発の責任を持ちます。責任とはチームの運営、コンポーネントのデザイン、実装（コーディング）、テストを含みます。

チームコンサートの開発チームは複数の小規模な開発チームによって構成されています（**図 6**）。各チームはおおむね 10 人程度、多くても 20 人は超えない規模に収まるように構成されます。Project Management Committee（以下、PMC）というチームをまたがるチームも構成され、プロジェクト全体のスケジュールや課題の調整などを行います。

運営はチームに任せられますが、多くのチームでは一般的なアジャイル開発のプラクティスを実行しています。例えば、コーディングが始まるとデイリー・スクラム・ミーティングを行い、

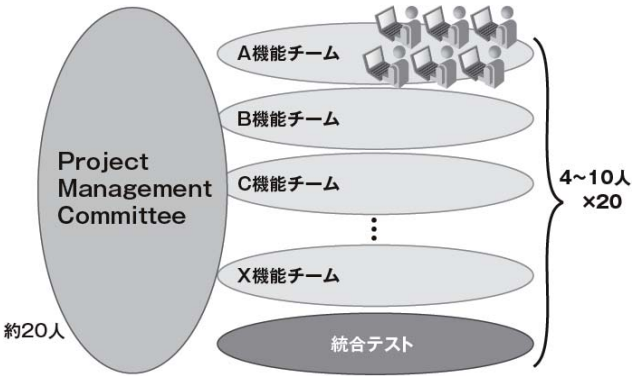


図6. 開発プロジェクトのチーム構成

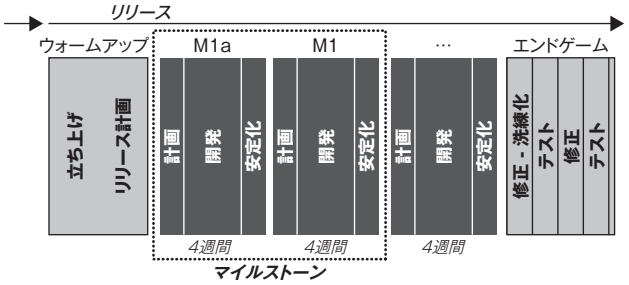


図7. リリース、マイルストーン、反復

毎日決まった時間に全員でステータスの更新と、問題を報告します。そこで挙げられた問題は担当者レベルで別途会議などを開いて解決します。

4.2.2 反復型開発&頻繁な小規模リリース

2章で述べたように、アジャイル開発では複数回の反復を繰り返し、反復のたびに動くプログラムをリリースしてステーク・ホルダーからのフィードバックを得ることが行われます。

チームコンサートの開発でも、全チームに共通する製品のリリース・スケジュール、反復のスケジュールなどは初期フェーズで決め、その後は4週間程度の反復を繰り返して開発を進めます（**図7**）。反復の前半でその反復で実装するものを決め、反復の後半では新規機能の実装は基本的に行わず、プログラムの安定化のためのテストとデバッグを全員で行います。反復終了時、その反復で実装した機能のデモをステーク・ホルダーに対して行い、フィードバックを得ます。反復が終了すると、動くプログラムが公開され、顧客も含めて広くフィードバックを得ることができます。

4.2.3 2つのレベルでの計画作りと追跡

アジャイル開発では、粗いリリース計画と、詳細な反復計画を用いて、計画作りとトラッキングを行います。

チームコンサートの開発の場合、プラニングのレベルは、リリース計画レベルと、反復計画レベルの大きく2つのレベルがあります（**図8**）。リリース計画は、製品リリースに含むべき大きなテーマや機能を、反復が始まる前に決めます。リリース計画で導出された計画項目に基づいて、各反復の最初に、その反復で実行するタスクを決めるプランニング・セッションを行い、誰が何を行うのかを全員で決めます。すべてのタスクはデイリー・スクラム・ミーティングやツールを介してチーム・メンバーで常に共有され、チーム・メンバーはチームで行われていることを常に把握することが可能です。

4.2.4 コンカレント・テスティング

アジャイル開発では、「すべてのコードはテストされたコード」として品質を担保します。JUnit のようなフレームワークを用いて、常にコードをテストすることが推奨されます。

チームコンサートの開発では、チームにコードを提供する前に、まず個人ビルドを行って、そこでテストを行います。さらに統合ビルドにコードを提供する前にチームの環境でビルドを行い、その成果物に対してテストを実行してコードの安定性を保持します。

4.2.5 継続的インテグレーション

継続的インテグレーションとは、正常に動くビルドを少なくとも1日1回作り出す考え方です。毎日統合ビルドを行うことによって、プログラム間の不整合を早期に発見できるなど、コードの安定化に役立てます。これを実現するには、ソース統合、ビルド自動化、テスト自動化が必要となります。

チームコンサートの開発では、毎日最低1回の統合ビルドが行われます。ビルドでは、コンパイル、パッケージング、自動テストが実行されます。コンパイル・エラーやテスト・エラーがタイムリーに報告されます。

エラーが報告されると、担当者は即座にそのエラーを修正することが求められます。特に反復の後半になると、すべてのエラーを修正してビルドが正しく行われるまで1日に何度も統合ビルドを行うこともあります。

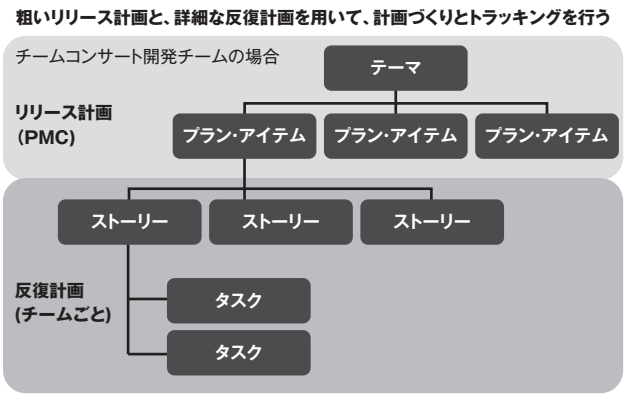


図8. リリースレベルと反復レベルの計画

4.2.6 継続的な考察と適応

アジャイル開発では、「ふりかえり」（レトロ・スペクティブ）を行います。うまくいったこと、いかなかったことなどをメンバー全員で話し合い、やり方自体に継続的な改善を行います。

チームコンサートの開発では、ふりかえりはチームごとに行われますが、チームで行われたふりかえりは必要に応じて PMC に報告され、プロジェクト全体の改善につなげていきます。

4.3 大規模な開発で必要な考慮点

4.3.1 意図的なアーキテクチャー設計

大規模なプロジェクトになると、複数のチームである程度独立して開発が進められるような大きな機能分割レベルでのアーキテクチャーが求められます。

チームコンサートの場合には、プロジェクトの開始時に少数のアーキテクトによる、初期アーキテクチャーの組み立てが行われました。機能分割によってコンポーネント化が行われると、コンポーネントに対応してチームが構成され、4.2 で述べたようにチームがコンポーネントの開発に責任を持って取り組むのです。複数のコンポーネントに影響がある課題を解決するのに、チームをまたがるチームとしての PMC が存在します。PMC には各チームからの代表者とプロジェクト全体のリーダーが集まり、全体にかかわるデザインや、複数のチームにまたがるインターフェースの調整などを行います。PMC はあくまで全体の調整役であり、個々のチームが自分たちのルールで自律的に開発を行います。

4.3.2 ジャスト・イン・タイムの要求定義

4.2.3 で述べたように、リリース計画と反復計画の大きく 2 つのレベルの計画が行われますが、大規模な開発になると、リリースをまたがって、プロジェクト全体に影響があるようなビジョンや、ロードマップなどを持つことも重要だと考えられます（**図 9**）。ビジョン規模になれば、それを実現するには時間がかかります。詳細な設計までしても、時間がたてば陳腐化してしまう可能性もあります。そのため、詳細な要求定義やコードにかかわるような設計は反復時に行います。

チームコンサートの場合にも、各リリースとは別にチームコンサートという製品そのもののビジョンを持ち、それを元にリリースごとに何を提供するかを考えていきますが、詳細設計などは反復の中で行っています。

4.3.3 分散した開発チームの運営

開発規模が大きくなれば、地理的に離れた開発者が参加することも多いでしょう。

チームコンサートの開発チームも世界中に散らばっています。1つのチーム内に複数の国から参加していることも珍しくありません。そのような構成で最も難しいのはメンバー間のコミュニケーションです。例えばデイリー・スクラム・ミーティングは電話会議で行いますが、時差の問題などはメンバー間で調整を行うしかありません。このように分散した開発チームが、同期を取りながらプロジェクトとして成果を出していくためには、コミュニケーションをサポートするツールの存在が欠かせません。次章では、このようなアジャイル開発を支える代表的なツールである Jazz プロジェクトのチームコンサートを解説します。

❺ ツールによるサポートの重要性

5.1 Jazz とは?

Jazz とは、一言で表すと、「ソフトウェアの開発チームの生産性向上」を助けるためのプロジェクトです［9］。大きな成功を収めた Eclipse プロジェクトの次のステージとして、IBM が進めているプロジェクトです。Jazz プロジェクトは、ソフトウェア開発というチーム作業をサポートするために、ソフトウェア開発の開発環境はどうあるべきかというビジョンを定義し、そのためのオープンなアーキテクチャーを策定し、そのアーキテクチャー上に Jazz 製品群を開発しています。Jazz が大事にしているのは、人々がソフトウェア開発においてどのように協調して働くべきか、すなわち、いかにコラボレーションし、生産性を向上させ、透明性を確保してソフトウェア開発を行うかという観点です。

そういった開発スタイルを実現するため

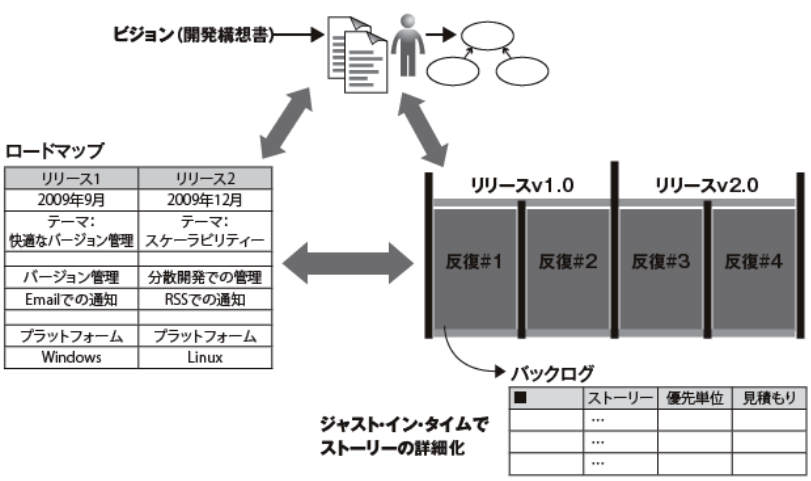


図9. ビジョン、ロードマップとジャスト・イン・タイムのバックログ詳細化

には、開発チームをサポートするどのような環境が必要でしょうか? 特に、昨今は、ソフトウェア開発は大規模化し、世界中に分散した開発チームが協業して、ソフトウェアを開発しています。大規模で、世界中に分散した開発チームであっても、協業できるような、理想的な開発プラットフォームが求められているのです。

Jazz プロジェクトのゴールは、理想的なチーム開発環境を提供することにより、開発チームをより協調的、革新的なものとし、価値のあるソフトウェアを開発するための手助けをすることです。では早速、Jazz プロジェクトを成り立たせる 3 つの要素を見ていきましょう。

① Jazz アーキテクチャー : 開発ライフサイクル全体の情報統合を可能に
② Jazz 製品群 : 理想的な開発環境を実現
③ Jazz コミュニティー : オープンでコミュニティー・ベースの商用開発

5.2 Jazz アーキテクチャー : 開発ライフサイクル全体の情報統合を可能に

ソフトウェア開発はチーム・スポーツに似ています。ソフトウェア開発は、一人で開発することは滅多にありません。ある程度規模のあるソフトウェアを作るためには、チームで開発する必要があります。そして、チームで開発するには、情報を共有したり、仕事を分担したりするために、さまざまなツールを利用することになります。典型的なものとして、バージョン管理を行うもの（CVS、Subversion など）や障害管理を行うもの（Bugzilla、Mantis など）があります。しかし現状では、それぞれのツールが独自サーバーを持ち、独自のユーザー管理、独自の UI、データ構造、アクセス方法を持っていて、それぞれ、導入、管理を行う必要があります。これではとても、チーム開発を支える環境が整っているとはいえないでしょう。こういった現状を打開し、理想的なチーム環境を提供する土台となるのが Jazz アーキテクチャーです［10］。サーバー環境も含めて、オープンで拡張可能なアーキテクチャーを提供し、統合された快適なチーム開発環境を実現します。

また、Jazz アーキテクチャーにおける外部インターフェースを、OSLC（Open Services for Lifecycle Collaboration）という外部団体で標準化する活動もしています［11］。こうすることで、他社ベンダー間のツールも相互に連携できるようになり、ユーザーにとっては、各エリアで最適なツールを選択し連携して使っていくことも近い将来に可能になってくるでしょう。

5.3 Jazz 製品群 : 理想的な開発環境を狙って

では、実際のところ、Jazz の製品とはどのようなものなのでしょうか? まず、Jazz 製品第一弾である、チームコンサートを見てみましょう［9］［12］。

チームコンサートは、チームが自律的にコラボレーティブに働ける理想的環境を目指し、チーム開発に必要なサーバー製品を1つにまとめて、改めて最適な構造に作り直した製品といえます。チームコンサートによって、人はソフトウェアを通してお客様に価値を届けるというソフトウェア開発の本来の仕事に集中することができるようになります。

まずは、人がやらなくてもよいような仕事を自動化することで肩代わりしてくれます。チームコンサートを利用することで、開発に関係する情報はすべて一元化され、レポート作成や、成果物間のひも付けや、ビルド作業などの労働集約的な作業を自動化してくれます。

次に、チームコンサートは人が間違ったことをしないようにガイドしてくれます。開発の規模が大きくなれば、一定の規律が求められます。例えば、コンパイル・エラーのあるコードをリポジトリーに入れないとか、ソースを変更した際には、どの機能拡張依頼や障害に基づいて変更したのかを適切に管理することなどです。これらの規律は、ソフトウェアの品質や開発効率の向上には欠かせませんが、チーム・メンバーに徹底させることは大変なことです。チームコンサートでは、ツールがそれらを肩代わりし、人がルールを守らなければならないという心理的な負担から解放し、創造的な仕事に集中させてくれます。

最後に、チームコンサートは情報の見える化によって、開発にかかわる人の待ち時間を削減してくれます。チームコンサートでは、情報が一元管理され、かつ情報の関連がリアルタイムで更新され、必要な情報が必要な人にリアルタイムで伝達されるようになっています。イベントのフィード、ディスカッションやチャット機能を内蔵しており、チームでのコミュニケーションをサポートしてくれます。

チームコンサートを利用した人の話を聞いてみると、もう昔の環境には戻れないという声をよく聞くのも無理のないことかもしれません。

5.4 Jazz コミュニティー : オープンでコミュニティー・ベースの商用開発

Jazz はこれまでとまったく異なる Open Commercial Development と呼ばれる新しい開発スタイルを取り入れています。それは、商用製品の開発にもかかわらず、オープンソース・コミュニティーで行われているようなコミュニティーを主体とした開発スタイルなのです。端的にいいますと、オープンソース・コミュニティーがやるように、ソースコード、開

発への参加者、マイルストーン、現状の障害までもjazz.net［9］と呼ばれるWebサイト上で公開しているのです。ユーザーは障害登録、機能拡張の要望を直接登録することができます。開発チームは直接、要望に回答します。こうすることで、早期におけるユーザー・フィードバックの取得を可能にし、パートナー様がJazzプラットフォーム上で自社製品を作成するための情報をより取得しやすくしています。商用の製品開発に、コミュニティーの手法を取り入れているところが、革新的なところだといえるでしょう。

❻ 今後の課題

これまで、アジャイル開発をスケールアップするためのエッセンスと実際の製品開発プロジェクトでの事例における適用方法、そして、アジャイル開発をサポートする開発ツールについて述べてきました。アジャイル開発のスケールアップは、まだまだ発展途上のものですが、プラクティスや開発ツールはかなり進化してきていることがお分かりいただけたでしょうか。

特に、アジャイル開発のスケールアップが発展してきた米国と比べると、米国ではユーザー企業が開発者を多く抱え企業内アプリケーションを内製化するのが一般的であるのに対して、日本ではユーザー企業での内製比率が低く、受託開発が多いという特徴があります。そのため、アジャイル開発を適用する際には、開発体制、契約、評価手法などにおいて、さらなる工夫をするべく多くの日本企業が取り組んでいます。

［参考文献］

［1］ Has Agile Peaked?, http://www.drdobbs.com/architecture-and-design/207600615
［2］ 非ウォーターフォール型開発に関する調査：（独）情報処理推進機構　ソフトウェア・エンジニアリング・センター , (2010).
［3］ XP, http://www.extremeprogramming.org/
［4］ Scrum Alliance, http://www.scrumalliance.org/
［5］ Agile manifest, http://agilemanifesto.org/
［6］ Dean Leffingwell, アジャイル開発の本質とスケールアップ , 翔泳社 , (2010).
［7］ 平鍋健児 , ソフトウェア工学の分岐点における、アジャイルの役割 , http://www.slideshare.net/hiranabe/software-engineering-and-role-of-agile
［8］ Scott Ambler, Agility@Scale / Disciplined Agile, IBM Rational, (2009). https://www.ibm.com/developerworks/mydeveloperworks/blogs/ambler/entry/disciplined_agile_delivery?lang=ja
［9］ Jazz.net, http://jazz.net/
［10］ Jazz Integration Architecture, https://jazz.net/projects/DevelopmentItem.jsp?href=content/project/plans/jia-overview/index.html
［11］ Open Services for Lifecycle Collaboration, http://open-services.net
［12］ Rational Team Concert wiki, https://www.ibm.com/developerworks/wikis/display/rtcjj

日本アイ・ビー・エム株式会社
ソフトウェア事業ラショナル事業部
クライアントテクニカルプロフェッショナル
IBM ソフトウェア・ライフサイクル・エバンジェリスト

玉川 憲 **Ken Tamagawa**

［プロフィール］
2000年にIBM Research Tokyo入所。ウェアラブル・コンピューターの研究開発に従事。2003年よりIBM Rational事業部に所属。RUP・要求管理・オブジェクト指向分析設計のコンサルティング実施。2006年より米国在住。アジャイル方法論を用いたソフトウェア開発に従事。2009年から現職。

日本アイ・ビー・エム株式会社
ソフトウェア開発研究所
シニア・テクニカル・スタッフ・メンバー

若尾 正樹 **Masaki Wakao**

［プロフィール］
1990年、日本IBM入社。ホームページ・ビルダー®、Rational Application Developer、Rational ClearQuest®などの製品開発を経て、2009年よりJazzベースの製品開発に従事している。

日本アイ・ビー・エム株式会社
ソフトウェア事業ラショナル事業部
クライアントテクニカルプロフェッショナル
シニア IT スペシャリスト

和田 洋 **Hiroshi Wada**

［プロフィール］
1988年、日本IBM入社。入社後メインフレームの生産技術エンジニア。1993年にSEに転身し製造業、金融業のお客様のシステムを構築。2001年よりソフトウェア事業WebSphere®事業部の技術サポートを経て、2003年よりIBM Rational事業部に異動（現職）。システムを構築する立場から、システムを構築する人たちを助ける立場に移り、いろいろなお客様に接する。