
Photo by Wada, Takashi

アジャイル
ソフトウェア開発マネジメント

Embrace Autonomy and Autonomation

shaping tomorrow with you

社会とお客様の豊かな未来のために

FUJITSU

リリース

初版　　2020 年 7 月

第 2 版　2020 年 9 月 8 日　　“5.3　在庫はリードタイム”を追加しました。

第 3 版　2020 年 9 月 28 日　　“第 6 章　アジャイル開発の士気管理”を追加しました。

第 4 版　2021 年 8 月 5 日　　“第 7 章　「アジャイルなら Git」の神話”を追加しました。

はじめに

アジャイルソフトウェア開発では、有用な機能を少しずつつくりながらリリースします。リリース頻度を高めソフトウェアの利用価値を高めるとともに、利用者からのフィードバックを頻繁に得るためです。

しかし、少しずつつくるということは、仕事の大きさは小さくなるものの、仕事の数は増えるということです。仕事の数が増えると、ただ一人の管理者がソフトウェア開発者たちに指示し、管理・監督することは非常に困難になります。このためアジャイルソフトウェア開発では、ソフトウェア開発者たちが自ら考え、判断し、行動するマネジメント、すなわち自律管理が必要となります。

本書は、アジャイルソフトウェア開発者向けのマネジメント Tips です。マニュアルやメソッドでもなければ、フレームワークでもありませんが、いくつかのマネジメント領域について自律管理的な観点からの検証、考察にもとづき記載されています。

本書をお読みいただくことで、自律管理の考え方に沿ったアジャイルソフトウェア開発マネジメントを実現いただければ幸甚です。

目次

第1章 アジャイル開発とは

変化の激しいビジネス環境の中で、ソフトウェアに対する要求の変化も激しさを増しています。要求の変化に追従するとともに新しいビジネス変化を生み出すために、より迅速なソフトウェアの提供が求められています。迅速にソフトウェアを開発する技法として、アジャイル開発が近年ますます注目されてきました。

1.1 アジャイル開発の概要とウォーターフォール開発との対比

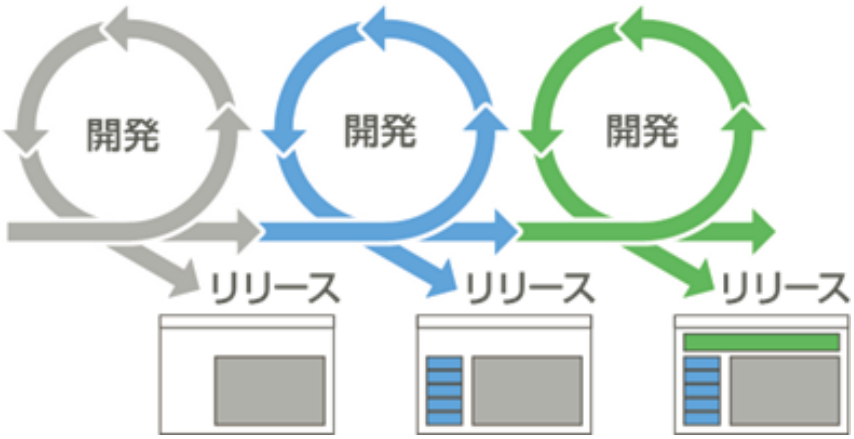
アジャイル開発プロセスの概要を述べます。また、アジャイル開発とウォーターフォール開発の特徴を比較します。

i. アジャイル開発とは

アジャイル開発には、スクラム、リーン開発、XP など様々な手法や方法論があり、プロセスも様々です。ここでは各開発方法論に共通したアジャイル開発プロセスの概要を示します。

反復増加型開発

アジャイル開発は、1 週間から 1 か月の反復期間を設け、その反復ごとに機能の追加を継続する「反復増加型」の開発プロセスによって実現されます。



図：反復増加型開発

計画

反復期間内にどの機能を開発対象とするかを計画します。開発対象の機能ごとに、優先順位づけと規模見積りを実施します。

開発

機能ごとに設計、実装、テストを行います。開発者は、テスト専任、実装専任などの役割分担はなく、全ての役割を担当します。

リリース

受入テストをパスした機能をリリースします。

開発の終了

さらなる機能エンハンスのために開発を継続するか、あるいは機能十分とみて開発を終了するかが、ソフトウェアを利用する顧客や製品責任者などビジネスサイドによって判断されます。

アジャイル開発の特長

アジャイル開発の特長、メリットをご紹介します。

より速いスピードでの提供 ～ 正味現在価値の視点

短サイクルの反復増加型開発により要望の高い機能が反復ごとにリリースされます。要望から提供までのスピードが速いため、利用したい機能をタイムリーに利用できます。

要求の変化に柔軟な対応が可能 ～ 変動対応性の視点

反復ごとに開発対象の機能を決定します。確定していない要望や変更の可能性が高い機能は、次回以降の反復で開発するかどうかが決定されるため、要求の変化に柔軟に対応できます。また、不必要な機能をつくることもありません。

品質と顧客満足度の向上 ～ リスク早期検知の視点

機能を一つ一つ順に利用可能な状態まで完成させます。要望の誤解、技術的な実現可能性や品質の検証が早期に実施されるため、故障も少なく満足度の高いソフトウェアが提供されます。

開発者やチームの成長を促進 ～ 学習の視点

機能を一つ一つ開発することにより設計、実装、テストのサイクルを繰り返します。また実装専任やテスト専任などの役割分担がありません。多様な作業を何度も体験することができるため“多能工化”、“技術力向上”、“士気の向上”など「開発者の成長」を促すとともに、“プロセスの改善”、“マネジメント力の向上”など「チームの成長」を促します。

アジャイル開発がもたらすもの

アジャイル開発は、機敏に迅速にソフトウェアを提供できる反復増加型の開発です。

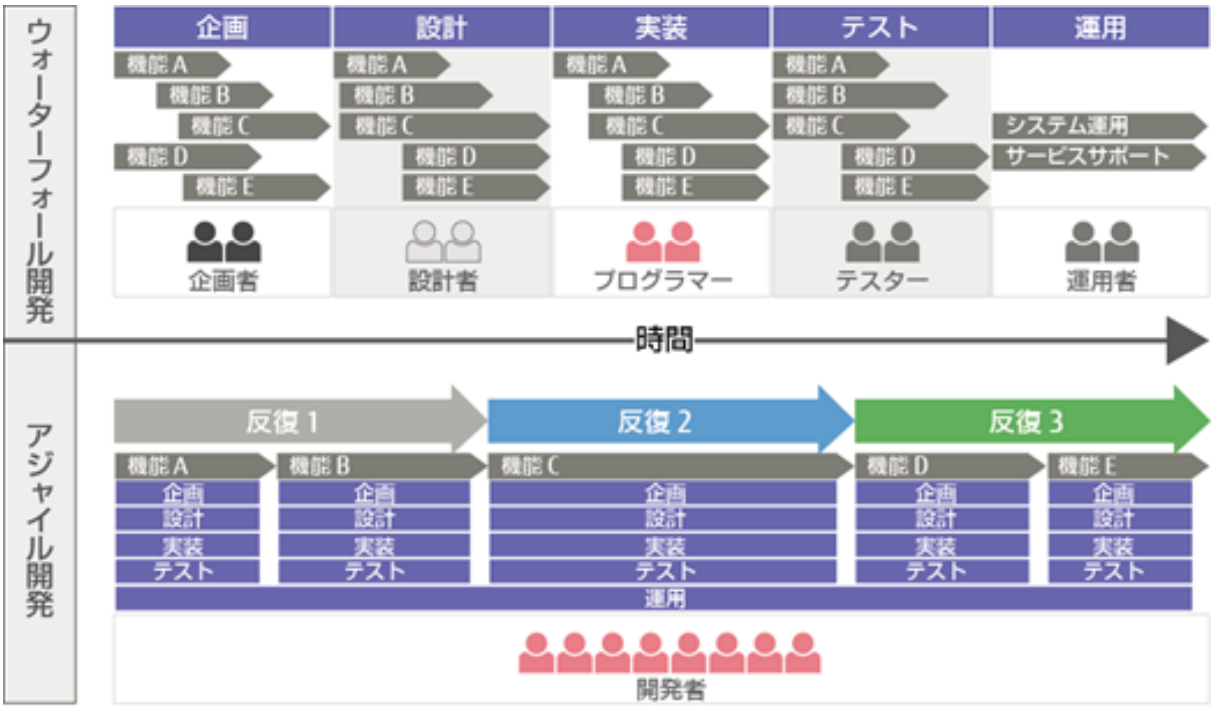
反復期間ごとに現実に動作し役に立つソフトウェアを提供することで、開発サイドとビジネスサイドが「現物のソフトウェア」を介してコミュニケーションすることが可能となります。

共に現物を触れることで、開発サイドは即座にフィードバックを受けることができます。ビジネスサイドは現物に（文字どおり）触発され、新たな要求やアイデアが湧き立ちます。

アジャイル開発は、要求の変化へ追従するだけではなく、新しい変化を自らもたらします。

ii. ウォーターフォール開発とアジャイル開発

ここからは伝統的なウォーターフォール開発とアジャイル開発の特徴を比較します。



図：ウォーターフォール開発とアジャイル開発

ウォーターフォール開発 ～ 工程に分割した開発

ウォーターフォール開発は、企画／設計／実装／テストなどの工程に分割して開発を進めます。

- 開発対象の機能を初期に確定させます。

- 各工程は工程専任の技術者が担当します。
- 前工程で作成されたドキュメントが、後工程への情報伝達の手段です。
- 全工程の完了後にソフトウェアが利用可能になります。

工程を分割することによって、以下のメリットがあります。

- 開発対象を初期に確定させるため、開発スケジュールの計画立案や見積りが容易になります。
- 工程ごとの担当開発者は専任技術者であり多種多様な技術を必要としないため、教育や採用が容易です。
- 工程ごとにエビデンスが残されることや工程完了確認がなされることで、作成請負契約による外部調達が容易です。

アジャイル開発 ～ 反復を繰り返す開発

アジャイル開発は、期間で区切られた反復を繰り返しながら開発を進めます。

- 強く要求される機能から順番に開発します。
- 開発対象の機能は反復ごとに決定します。
- それぞれの開発者は設計、実装、テストなどすべての開発作業を担当します。
- 実際に動作するソフトウェアを反復ごとに提供します。

反復を繰り返すことによって、以下のメリットがあります。

- 反復ごとに開発／提供するため、より速いスピードでの提供が可能です。
- 反復ごとに開発対象の機能を決定するため、要求の変化に柔軟に対応することが可能です。
- 反復ごとに機能の出来栄えを確かめられるため、品質や顧客満足度が向上します。
- すべての開発作業を何度も繰り返すため、開発者の成長やプロセスの改善およびチームの成長を促します。

特徴の比較

ウォーターフォール開発とアジャイル開発の特徴を各観点で比較します。

表：ウォーターフォール開発とアジャイル開発の特徴

比較の観点	ウォーターフォール開発	アジャイル開発
ビジネスサイド／顧客の関与	・開発開始前に要求を決定 ・開発終了後に受入の判定	・開発期間中に要求の変更や追加 ・開発期間中に受入の判定
要求変更への対応	・開発前の要求確定が原則	・開発期間中の変更を受入れる（反復ごとに開発対象を決定するため）
提供スピード	・全工程の完了後に提供	・速い（反復ごとに提供）
ドキュメント	・各工程の生産物として必須・後工程への唯一の情報伝達手段 ・工程完了のエビデンスとして必須	・適切な情報伝達手段であれば作成（保守用途、リリースノートなど）
テスト	・テスト工程でのみ実施	・実装やビルドごと頻繁にテスト
開発者	・工程ごとの専任担当者	・すべての開発作業を担当する多能工

1.2 スクラムとエクストリームプログラミング（XP）

i. スクラムとは

スクラムとは、反復増加型ソフトウェア開発チームに適用するプロジェクト管理フレームワークです。

スクラムの由来

1986 年、論文「The New New Product Development Game」が野中郁次郎氏と竹内弘高氏によって発表されました。日本の製造業における新製品開発や組織の形態が、ラグビー、スクラムのようであると例えられています。

1993 年、ジェフ・サザーランド氏らが反復増加型の開発を実践する中で、この論文に触発されて名付けられたのがソフトウェア開発「スクラム」の最初です。

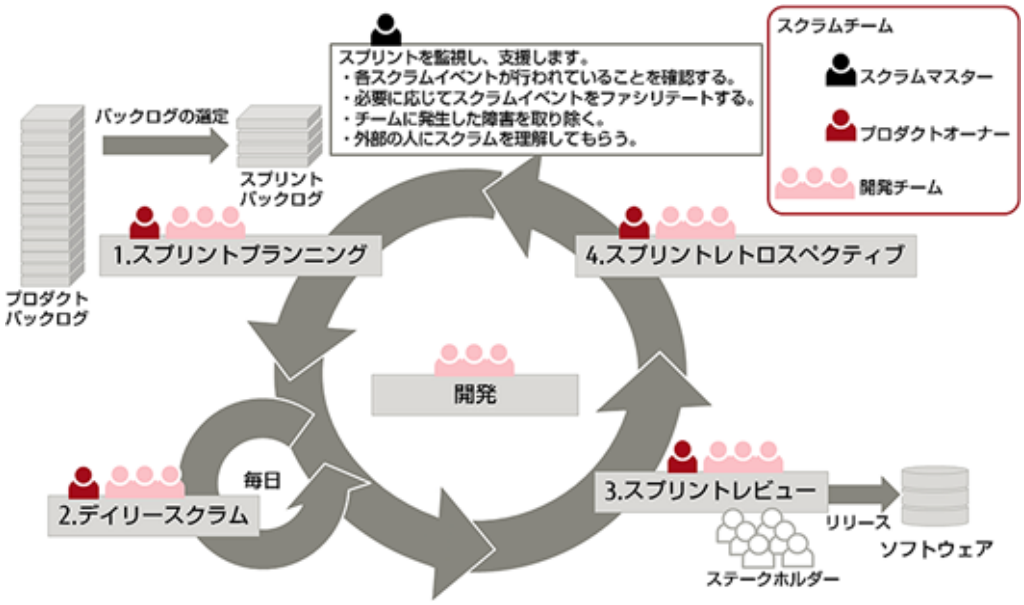
スクラムの特徴

スクラムの特徴を以下に示します。

- 反復増加型のソフトウェア開発プロジェクトを管理するためのフレームワークです。
- 開発技術を必要とする活動はありません。
- 反復期間内に開発する機能は、反復ごとに決定されます。
- 反復期間中、チームは外部（顧客やビジネスサイド）からの干渉を全く受けません。
- プロジェクト管理の権限はチームに委譲され、開発手法や利用技術もチームが決定します。

スクラムのプロセス概要

スクラムにおける開発プロセスの大まかな流れを以下に示します。



図：スクラムのプロセス概要

スクラムのプロセスは、スプリントと呼ばれる反復期間を繰り返すことで増加的に機能を開発します。スプリントでは、以下に挙げるスクラムイベントが実施されます。

1. スプリントプランニング：

プロダクトオーナーと開発チームは、全体のプロダクトバックログ（実装予定機能の一覧）の中からスプリント（反復期間）で実装する機能を決定します。 選定された機能一覧をスプリントバックログといいます。

2. デイリースクラム：

決まった時刻に開発チームは短時間のミーティングを実施します。 開発の進行状況や課題などを共有し一日の仕事の進め方を決定します。

3. スプリントレビュー：

スプリントの最後にステークホルダーに対して開発した機能をデモンストレーションします。ステークホルダーからフィードバックを受けます。 レビューの結果が OK の機能はリリースされますが、NG の場合には次回以降のスプリントにリリースは持ち越されます。

4. スプリントレトロスペクティブ：

スプリントレビューの終了後、スクラムチームは今回のスプリントのふりかえりを実施します。レトロスペクティブとは「ふりかえり」のことです。次のスプリントの改善計画を行います。

スプリントの期間は 1 週間から 1 か月が設定されることが一般的です。

スクラムにおける役割

スクラムチームは以下の 3 つの役割で構成されます。

プロダクトオーナー（PO）：

- プロダクトオーナーはソフトウェアの価値を最大にすることに責任を持ちます。
- プロダクトバックログを優先順位順に並べ（並び替え）ます。
- 開発チームに対して次に開発する機能を示し、開発の動機づけを行います。

スクラムマスター（SM）

- スクラムチームの価値を最大にする役割があります。
- 開発チームがスクラムイベントを理解し、スプリントを円滑に回すことに責任を持ちます。
- プロダクトオーナーに対して開発チームの現状を説明し理解を得ます。
- ステークホルダーに開発プロセスの実態や狙いを説明し理解を得ます。
- スプリントバックログの開発に寄与しない活動がないか監督し改善に寄与します。

開発チーム

- スプリントごとにソフトウェアを提供することに責任を持ちます。
- 開発チームは実際に動作するソフトウェアを開発します。

  （リリースの可否はスプリントレビューで判断されます。リリース判断可能な状態にすることに責任を持ちます）

- 開発プロセスや開発手法の新たな採用や、改善の方法を決定します。
- 開発チームの人数は 7±2 人が望ましいとされています。

スクラムによる効果

- スクラムの導入は、ソフトウェア開発プロジェクトに以下の効果をもたらします。
- スプリントごとに動くソフトウェアを顧客に提供し、フィードバックを受けられます。
- 顧客が提示した順番に機能を提供できます。

- 反復増加型のアジャイル開発に適した自律的なチームを早期に形成できます。

外部からの干渉を受けない権限移譲されたチームが、共有された目的達成のために新たなブレークスルーやイノベーションを生みます。

～小話～　スクラムの特徴を語るこんな小話があります。

書籍「アジャイルソフトウェア開発スクラム」から

ニワトリとブタがいた。
　ニワトリは「さあ、レストランでもやろうよ」と言った。
　ブタはよく考えてから「レストランの名前は何にしようか」と言った。
　ニワトリ は考えた。「ハムーエッグさ」
　ブタは言った。「僕は止めておくよ。君は産むだけだけど、僕は切り刻まれるんだよ」

ニワトリは出資はするものの再生産可能な卵を提供するといい、対してブタは死を伴う肉の提供を拒否しています。つまり、痛みを伴う者たちが自分たちの行動を決定すべきであることを伝えています。

プロジェクトの権限を委譲され、外部からの干渉は受けない開発チームは、自分たちで開発手法や利用技術を決定します。他の誰も決めてくれません。

期限のある反復期間の中で「誰が何をどのようにつくるか」を自分たちで決定しなければならない状態に（半ば強制的に）置かれたチームは、自律管理や自己組織化が高速に進行します。また、技術的なブレークスルーやイノベーションを生み出す動機ともなり得ます。

スクラムの効果は、短期の自己組織化とイノベーションを生む環境をつくりだすことです。

ii. エクストリームプログラミング（XP:eXtreme Programing）とは

エクストリームプログラミングとは、開発やマネジメントの経験則を適用の指針とともにまとめたものです。

XP の由来

ケント・ベックとウォード・カニンガムは、「パタン・ランゲージ（pattern language）」と呼ばれる建築理論に触発され、ソフトウェア開発の成功事例を「パターン」として集める活動を始めました。

その後、ベックは自らが参画したクライスラー社の開発プロジェクトで、集めたパターン群を実践して調整や改善を重ね、1999 年にエクストリームプログラミングとして提唱されました。

ソフトウェアの開発現場における経験則を「極端（＝エクストリーム：extreme）」なまでに実践するという発想から命名されました。

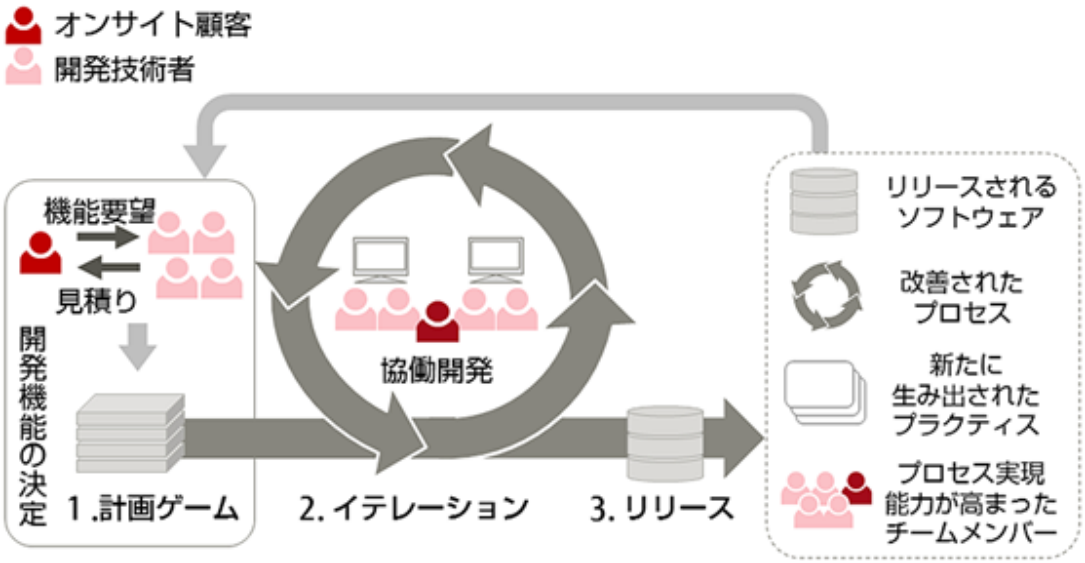
XP の特徴

XP の特徴を以下に示します。

- 開発に関わるすべての人が互いに尊重し、働きかけ、協力し合うことが重視されています。
- 状況の変化を歓迎し、受け入れ、それにすばやく追従することができます。
- 顧客はチームメンバーの一員です。オンサイト顧客と呼ばれチームと常に同席し、技術者と協働してソフトウェアを開発します。
- 開発プロセスはチームにより新たにつくられ、よりよいものに改良され、合わなくなったものは捨て去られます。
- 開発技術を必要とする活動が多くあります。

XP のプロセス概要

XP における開発プロセスの大まかな流れを以下に示します。



図：XP のプロセス概要

XP のプロセスは、イテレーションと呼ばれる反復期間を繰り返すことで増加的に機能を開発します。イテレーションおよびその前後に、以下に挙げる活動が実施されます。

1. 計画ゲーム：

イテレーションの期間内に実装する機能を決定します。
開発技術者は機能の規模を見積り、オンサイト顧客は機能の優先順位を提示します。
両者のやりとりによって開発機能が決定されます。

2. イテレーション：

オンサイト顧客と開発技術者が協働して開発を行います。
オンサイト顧客は機能の受入条件を作成します。
オンサイト顧客は開発技術者と協働して受入テストを作成、実施します。
オンサイト顧客と開発技術者は常に同席し緊密なコミュニケーションのもとで、機能の詳細／技術上の制約／ビジネスサイドの目的を明らかにします。

開発技術者は機能の開発と同時にテストを作成実施し、ソフトウェアが正常に動作する状態を維持します。動作するソフトウェアを共にみて触れることで、オンサイト顧客と開発技術者は齟齬のないコミュニケーションを図ります。

3. リリース：

イテレーションの終了時に、オンサイト顧客の受入テストが完了している機能をリリースします。

イテレーションの期間は 1 週間から 2 週間が設定されることが一般的です。

XP のプラクティス

XP では、ソフトウェア開発における問題発生のリスクを低減する具体的かつ経験則的な活動を「プラクティス」と呼んでいます。プラクティスの多くはチームや組織のマネジメントを目的としていますが、実行に当たっては開発技術が必要なプラクティスがほとんどです。

ここでは、代表的なプラクティスをいくつか紹介します。

全員同席

十分な広さのスペースに全員が集まって開発を行います。オンサイト顧客も常に同席します。

技術上の問題の発見や解決までの時間を短縮するとともに、ビジネスサイドの要望や詳細な仕様も即座に確認できるため誤解を極めて短時間で検知できます。

計画ゲーム

オンサイト顧客と開発技術者が優先順位と技術的な見積りをもとに、イテレーションで開発する機能を決めます。（見積りには実装方法の検討つまり設計行為が必要です）

ビジネスサイドからのリリースタイミングの制約と、チームが開発可能な規模の折り合いをつけることが目的です。 ビジネス戦略の実現に、開発チームが無理なく継続的に貢献することが可能となります。

ストーリー

開発する機能を、顧客から見てどう動くのかを簡潔な文章で表現します。

機能単位での増加型開発を容易にします。ストーリーには受入条件を併記するため、過剰な実装を防ぐと共に品質の向上が図れます。

ペアプログラミング

1 台の PC を 2 人ペアで使い開発します。調査、設計、実装、テストなどの開発作業をペアで一緒に実行します。

問題の発見、解決策の実行が即座に実施されるため、機能提供の時間が短縮されるとともに品質が向上します。個人の知見や知識が伝搬され、開発技術者のスキル、チームの開発能力、ソフトウェアの品質が向上します。

継続的インテグレーション

開発中のソフトウェアのビルド、テストを自動的にかつ即座に実行します。

「いつでも出荷できる状態」を継続的に保つことを目指し、「テストされていない時間」を最小化することで問題の発生をすぐに検知し、すばやい対応が行えます。

コードの共同所有

チームのメンバーはすべてのコードを参照し改修することができます。コンポーネントごとの「担当者」は存在しません。

個々の開発技術者が広範囲のコードを扱うことで、個々の開発技術者の知識範囲が拡大します。担当者不在による開発遅延の防止に効果があります。

XP における役割

XP チームではメンバーの役割は固定化されず、状況に応じて全員が様々な役割を担当します。以下のような役割があります。

コーチ

ストーリーの開発には直接参加しません。技術的な、例えばアーキテクチャやテスト戦略などに課題が発生した場合には、遊撃的に解決にあたります。
開発の停滞を防ぐとともに、開発技術者に対して具体的に支援することでチームやメンバーの能力向上に寄与します。

トラッカー

完了状況と残量つまり進行状況を常に追跡（トラッキング）して把握します。
チームの負荷状況やイテレーション内での完了見込みをチームに知らせます。

顧客

顧客はチームの一員です。オンサイト顧客と呼ばれ開発技術者と常に同席します。

ビジネス的な視点に基づき、チームが開発する機能の意思決定を行います。
開発する機能とその受入条件を明示し優先順位を付けます。
受入テストを開発技術者とともに作成し実行します。
ビジネスサイドの目的や仕様詳細を開発技術者に伝えます。

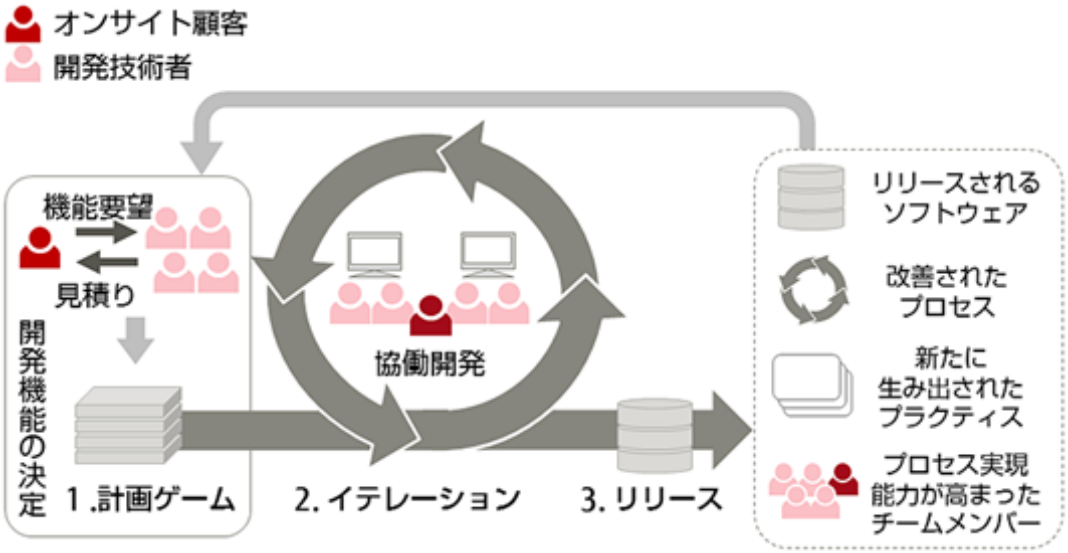
XP による効果

XP はソフトウェア開発プロジェクトに以下の効果をもたらします。

- 計画ゲームによって、ビジネスサイドからの要求とチームの負荷の折り合いをつけるため、
  - ビジネス戦略の実現に、開発チームが無理なく継続的に貢献することが可能となります。
- 常時同席しているオンサイト顧客が、動作するソフトウェアを直接確認するため、
  - 確認待ち時間が最小化され、機能の開発完了までの時間が短縮されます。
  - すばやく詳細なフィードバックによって、使い勝手の良いソフトウェアとなります。
- 品質リスクへの対応を目的の一つとするプラクティスが充実しているため、
  - 問題の早期発見、解決の迅速化、手戻りの最小化、ソフトウェアの品質向上効果があります。
    （ペアプログラミング、継続的インテグレーション、コードの共同所有 etc.）
- 知識の伝搬を促進するプラクティスが多いため、

- 技術知識や業務知識がチーム全体に伝搬され、開発技術者の多能工化とスキル向上、チームの開発能力向上、ソフトウェアの品質向上に効果があります。
  （計画ゲーム、ストーリー、ペアプログラミング、コードの共同所有 etc.）

- 開発技術者の多能工化とチームの開発能力向上によって、
  - より少ない開発技術者によって、以前と同じ負荷をこなすことが可能となります。
  - 高い技術レベルを有する技術者が、他の開発プロジェクトに移動してもインパクトが最小化されます。
  - 技術者の移動でチームが少人数になった場合には、他チームと合併しメンバー数を増やすことで多くの知識を共有する機会を復活させることができます。



図：XP のプロセス概要（再掲）

再掲した図“XP のプロセス概要”を見ると、イテレーションのアウトプットは、リリースされるソフトウェアだけではありません。

- 改善されたプロセス
- 新たに生み出されたプラクティス
- プロセスを実現能力が高まったチームメンバー

これらもイテレーションのアウトプットです。アウトプットは次のイテレーションのインプットとなります。

ソフトウェアが反復増加型で開発され成長し続けるのと同様に、プロセス（Process）もプラクティス（Practice）もチーム（People）も成長し続けます。

1.3 スクラムと XP の調和

アジャイル開発プロセスの事例を紹介します。

アジャイル開発の代表的な方法論である「スクラム」と「エクストリームプログラミング（XP：eXtreme Programming）」の特徴を活かし、調和させた開発プロセスの事例です。

スクラムと XP の特徴

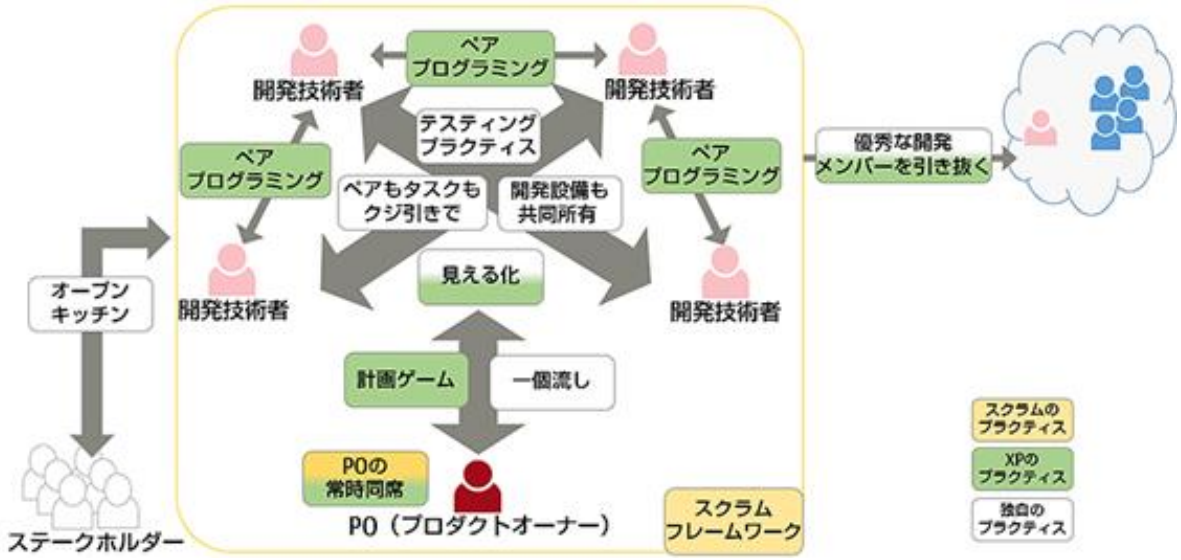
スクラムと XP の特徴について、簡単にまとめます。

表：スクラムと XP の特徴

	スクラム	XP
実態	反復型開発をうまく回すための仕組み （フレームワーク）	よいものをつくるための知恵の集まり （パタン・ランゲージ）
チームとビジネスサイドの関係	スプリント期間中は没交渉 ・スプリントの前に要求確定 ・スプリント後に確認レビュー	全期間に渡って深く関与 ・要求の決定、変更、抹消 ・受入テストの作成実施 ・開発技術者と常に同席
プロセスの特徴	フレームワークの実行に開発技術は不要 ・外部干渉を遮断したブレークスルー指向 ・ブレークスルーを利用したプロダクトアウト指向	プラクティスの実行に開発技術が必要 ・プラクティス自体はマネジメント指向 ・顧客との緊密な関係はマーケットイン指向
適した領域	ブレークスルーが必要な新製品の開発	継続的な成長を望む寿命が長い組織
チームに適した人材	能力の高い開発技術者たち	ふつうの開発技術者たち 能力が高ければ高くてもよい
人材とチームの能力	早期のチームビルディングが可能 個々の高い能力がチームによって活かせる	プラクティスのほとんどは人材／チームの育成が目的 人材／チームの成長は長期継続的

スクラムと XP の調和

スクラムと XP のそれぞれの特徴を活かしたアジャイル開発プロセスの事例を、具体的なプラクティスをもとに紹介します。



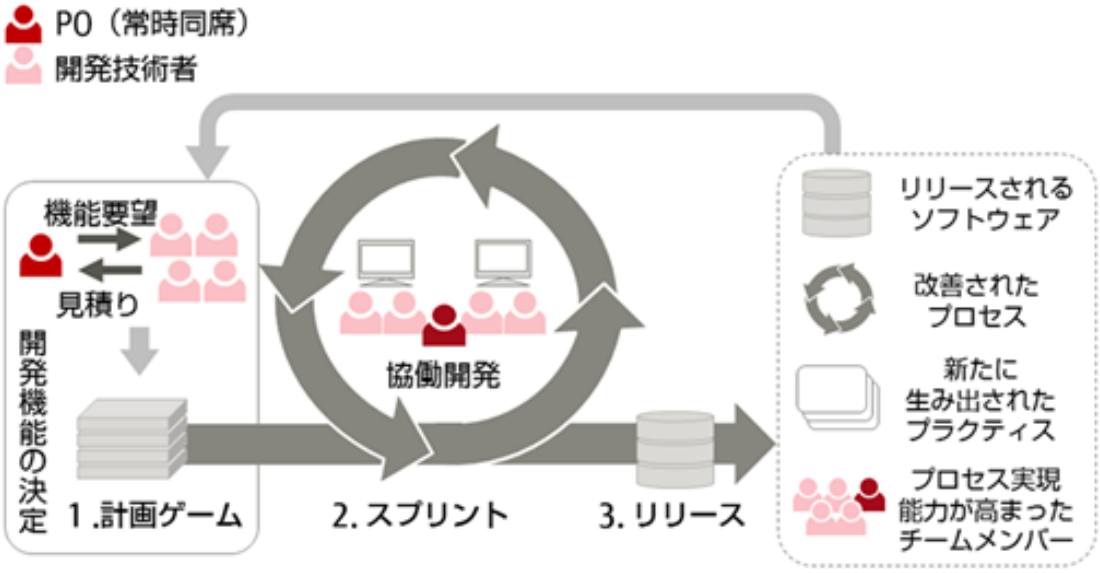
図：スクラムと XP の調和

スクラムフレームワーク

プロセスの骨格にはスクラムフレームワークを利用します。これは、スクラムの「早期のチームビルディングが可能」という特徴を利用するためです。

スクラムフレームワークは、スクラムガイドブックに明確に規定されており、チームが反復増加型の開発サイクルを短期間で修得することが可能です。また、スクラム自体の知名度が高く、フレームワークの導入に対して抵抗感が少ないことも有利なポイントの一つです。

なお、本事例では専任のスクラムマスターは配置しません。スクラムマスターの役割は、リーダーと開発技術者たち自身が担います。



図：スクラムと XP の融合プロセス

PO（プロダクトオーナー）の常時同席

本事例では、PO と名乗りつつ、実態は XP の「オンサイト顧客」です。

PO はチームの一員です。開発技術者と常に同席します。
PO は機能の受入条件を作成します。また、開発技術者と協働して受入テストを作成、実施します。
PO は受入テストをスプリントの終了を待つことなく実施します。
PO はビジネス的な視点に基づき、チームが開発する機能の意思決定を行います。

実態は XP のオンサイト顧客なのですが、スクラムフレームワークの PO という言葉が広く浸透されているため、その名称を利用します。

PO が開発技術者と常に同席し緊密なコミュニケーションを図ることで、機能の詳細／技術上の制約／ビジネスサイドの目的を明らかにします。
常時同席している PO が、動作するソフトウェアを直接確認するため、確認待ち時間が最小化され、機能の開発完了までの時間が短縮されます。

PO からのすばやく詳細なフィードバックによって、使い勝手の良いソフトウェアとなります。

計画ゲーム

PO が同席し開発チーム全員で行います。
ビジネスサイドからのリリースタイミングの制約と、チームが開発可能な規模の折り合いをつけることが目的です。

開発技術者は機能の規模を見積り、PO は機能の優先順位を提示します。
開発技術者と PO が技術的な見積りと優先順位をもとに、スプリントで開発する機能を決めます。（見積りには実装方法の検討つまり設計行為が必要です）

ビジネス戦略の実現に、開発チームが無理なく継続的に貢献することが可能となります。

ペアプログラミング

開発はすべてペアプログラミングです。プログラミング以外の開発作業もすべてペアワークです。

ペアプログラミングの本質は、単なる技術の伝達や知識の共有ではありません。

ペア間での対話によって自らに新たなナレッジを生み、そのナレッジがほかのメンバーに伝搬されることで、さらなるナレッジを創造する機会が生まれます。相互作用を重ね、ナレッジのスパイラルアップを狙います。

開発設備も共同所有

ソースコードはもちろん共同所有します。それだけではありません。開発設備も共同所有します。机や椅子、ディスプレイも共同所有です。

一日の仕事が終了したら、毎日毎日所定の位置に格納します。デスクトップ PC も配線やキーボードを外して収納します。
整理、整頓、清掃、清潔を維持、習慣化します。

ドキュメント、ファイルの管理方法はもちろん、構成管理（例えば Git のブランチ）や要求管理（例えば Redmine のチケット記事）、ソースコードやテストコードのリファクタリング、開発プロセス、管理プロセス、成果物も整理、整頓、清掃、清潔の維持が習慣化します。

これは全て、品質のためです。ソフトウェアの品質を維持向上させる行動の習慣化のために、共同所有し、毎日毎日綺麗に片付けるのです。

ペアもタスクもクジ引きで

ペアプログラミングの相方は毎朝クジを引いて決めます。ペアが担当するタスクも毎日くじを引いて決めます。ランダムに決まるペア。ランダムに決まる担当タスク。

公平とか不公平とかではありません。

ランダムに決まると得手不得手が発覚します。開発技術者たちそれぞれが持つ技術や知識、持っていない技術や知識が、否応なく表面化するのです。表面化したそのときが、学習の機会です。

より多くの学習の機会を得て、開発技術者の技術や知識、チームの能力向上を図ること、それが狙いです。

テスティングプラクティス

- 一行テスト

機能が記載されたシナリオやユーザーマニュアルなどドキュメントの記載通りにシステムが動作するか、ドキュメントの記載一行ずつを全てテストします。

開発技術者が利用者の仮面を被り、本当に使えるのか？使い勝手はよいのか？使うと嬉しいのか？を感じながらテストすることで、品質向上のポイントを見つけることが、一行テストの狙いです。

- 探索的テスト

テストにかける時間を決めた上で、テスト対象を設定し、開発メンバーがユーザー利用の観点で、目標とした不具合検出数を目指してシステムの動作テストを行います。

検出した不具合を診て、その傾向やインパクトの大きさを検証します。ソフトウェア構造に起因する不具合、処理性能にかかわるような不具合が検出されていれば、異なる発生条件、異なる不具合現象が発生する可能性が高いと判断されます。

より効果的な品質向上のポイントを探すのが探索的テストの狙いです。

見える化

単なる可視化やグラフ化を見える化と呼ぶことがあるようですが、ここに述べる見える化はそうではありません。リアルタイムな異常検知と迅速な対応のためのプラクティスが見える化です。

いつもと違う状態、問題が発生する可能性が高まる状態をあぶり出すことを狙っています。

「アナログなかんばん」や「バーンダウンチャート」をプロジェクトルームに貼り出します。チーム全員が常に見える状況は必須です。メールで送られてくるスプレッドシートや、バグトラッキングツールでは見える化は実現しません。

貼られている状態、貼り変えようとしているチームメンバー、それ自体が見える化の対象です。

プロジェクトの進行状況が一目でわかるようすること、正常状態と非正常（異常）状態の判別が誰にでも付くことで、リアルタイムな異常検知と迅速な対応が可能になります。

オープンキッチン

チームの一員である PO はもちろん、その他お客さまを含むステークホルダーにも開発過程のすべてを公開しています。アナログツールによる進行状況の共有だけでなく、内部検出した不具合、規模や工数見積りの予定と実績、開発者間のチャットやブログまでも（生々しい不満も愚痴も悪口も）公開しています。エビデンスではなく、プロセスそのものがお客様の目の前で進行するその様は、さながら「オープンキッチン」です。

オープンキッチンのメリットは以下の 2 点です。

- つくったソフトウェアもそのつくり方も常に公開されていることで、チームに、そして開発技術者それぞれに「見られている」感覚が生まれます。その感覚は適度な緊張感と規律を生みます。決して手を抜かないクラフトマンシップと、さらなる品質向上をめざしたチャレンジの組織文化・風土を醸成します。
- 常に公開されていること、頻繁なインタラクションが発生することなど、ソフトウェアやそのプロセスを介した濃密なコミュニケーションが相互理解を進行させます。チームにはお客様の趣味嗜好への洞察が、お客様には開発チームの技量への期待が生まれ、オープンキッチンが共創の場へと進化します。

優秀な開発メンバーを引き抜く

常に誰かが居なくなるかもしれない。当初からそう宣告します。

要らない人を追放する訳では決してありません。優秀な開発技術者が、よりチャレンジングでエキサイティングなプロジェクトで活躍するためです。

であれば、

残されたチームの能力が決して低下しないようなプラクティスの利用が促進されます。
「ペアプログラミング」、「ペアもタスクもクジ引き」などのプラクティスにより、1 人の異動がチームにインパクトを与えなくなります。
いま所属しているチームで活躍すれば、よりやりがいのあるチームに異動できます。
いま所属するチームで活躍する動機の一つとなります。

優秀な開発技術者がいなくなれば、残された開発技術者に活躍のチャンスが生まれます。
優秀な開発技術者がチームを去ることが、今後活躍する動機の一つとなります。

優秀な開発技術者の流動性を高めることが、いまのチームにも、異動先のチームにも、異動する開発技術者にも、残留する開発技術者にもよい影響を与えます。

（「優秀ではない人」が「重要ではない仕事」の担当に異動させられる。そんなことが習慣化されてしまえば…、誰が幸せになるというのでしょうか…）

一個流し

複数の機能を同時に着手すると、完了までに時間がかかります。
一つの機能に集中して開発すると、完了までの時間がより短くなります。

工程で分割して、複数の機能の開発を同時に進行させるウォーターフォール開発は完了までの時間がかかります。
また、アジャイル開発であっても複数の機能それぞれを担当割りしてしまうと、完了までの時間がかかります。

一つの機能の開発に集中して開発することを「一個流し」と呼びます。

少々大きな機能であっても設計や実装方法の工夫で、複数の開発技術者が寄ってたかって開発することで、一個流しは実現できます。

開発期間、すなわち提供までの時間を短縮することができるのです。

本章冒頭に、こう記しました。

変化の激しいビジネス環境の中で、ソフトウェアに対する要求の変化も激しさを増しています。要求の変化に追従するとともに新しいビジネス変化を生み出すために、より迅速なソフトウェアの提供が求められています。迅速にソフトウェアを開発する技法として、アジャイル開発が近年ますます注目されてきました。

変化に追従し、新しい変化を生み出すためには、よりスピードのある開発が必要です。

一個流しは、「環境の変化に追従」し「自ら新しい変化を生み出す」ためのプラクティスです。

最後に

XP の提唱者の一人である Kent Beck らの著書「Extreme Programming Explained」のサブタイトルは、「Embrace Change」です。「Embrace Change」を「変化ヲ抱擁セヨ」と翻訳する"Agile practitioner and book translator"がいます。XP に影響を与えたトヨタ生産方式で知られる大野耐一の著書「トヨタ生産方式」のサブタイトルは「脱規模の経営をめざして」です。また、トヨタ生産方式を起源とする「リー

ン開発」の提唱者である Poppendieck の著書「Implementing Lean Software Development」のサブタイトルは、「From Concept to Cash」です。

「From Concept to Cash」は、発想や要望が発生してから、実体化し、商品となり、価値を認めたお客様が購入し、お金を生むまでの時間のことです。「脱規模の経営をめざして」は、規模の経済からスピードの経済への変革を意味します。スピードを高めることが環境の変化に追従するための最大要因であり、変化ヲ抱擁スル態度こそが自ら新しい変化を生み出す源泉だと考えられます。

第2章 アジャイル開発の品質管理技法

2.1 開発プロセスの特徴

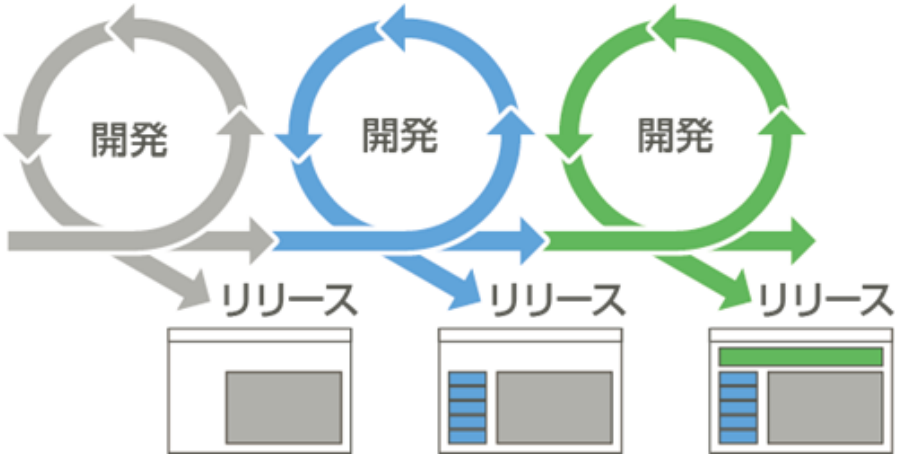
"新技術の登場"や"市場の要求"が激しく変化する現在、ICT は情報システム部門だけのものではなくなってきています。

「技術や市場への機敏な反応が高収益やビジネス成功の条件である」と判断した事業部門が自ら ICT を企画、開発、運用する状況が進行しているためです。

i. アジャイル開発の特長

アジャイル開発の品質管理技法を考えるには、アジャイル開発の特長や（品質管理技法が確立されているとされている）ウォーターフォール開発との違いを踏まえておく必要があります。

まずは、アジャイル開発の特長について見てみましょう。



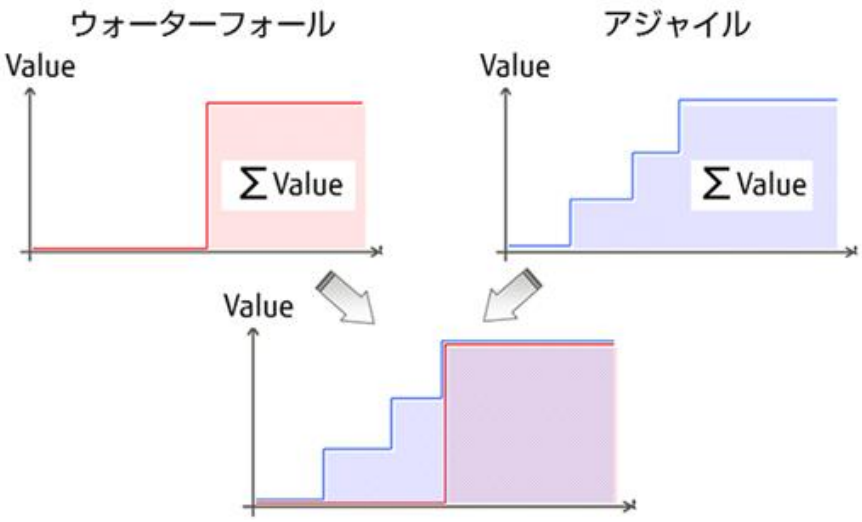
図：反復増加型開発

テクノロジーコラム「第 1 章 アジャイル開発とは」でご紹介したように、反復増加型開発であるアジャイル開発には以下の特長があります。

- より速いスピードでの提供　～ 正味現在価値の視点
- 要求の変化に柔軟な対応が可能　～ 変動対応性の視点
- 品質と顧客満足度の向上　～ リスク早期検知の視点
- 開発者やチームの成長を促進　～ 学習の視点

これらの特長は、規模の経済からスピードの経済への変革、すなわち、変化を受入れるためのリードタイム短縮を目指すことによって得られるものです。

より速いスピードでの提供 ～ 正味現在価値の視点

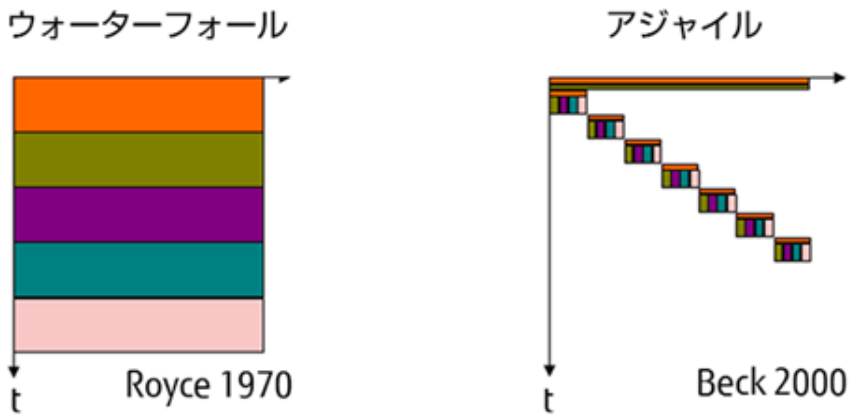


短サイクルの反復増加型開発によって、要望の高い機能が反復ごとにリリースされます。要望から提供のスピードが速いため、利用したい機能をタイムリーに利用できます。提供ごとに機能を利用できるため価値の総和が最大化されます。

ウォーターフォール開発では機能が利用可能になるのは開発終了時ですが、アジャイル開発では開発中であっても段階的に機能を利用できます。このためアジャイル開発は、価値（機能の利用による便益の享受）の総量を最大化できます。

ウォーターフォール開発とアジャイル開発で、全ての機能を開発する工数が同じであれば、これは生産性の向上を意味します。

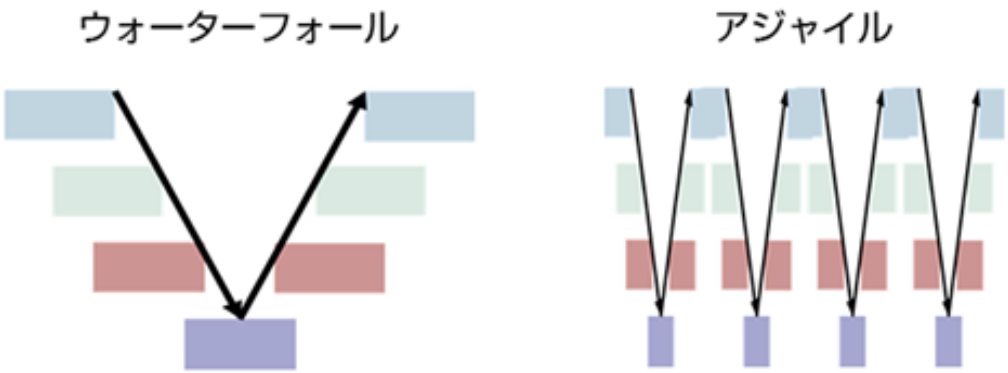
要求の変化に柔軟な対応が可能 ～ 変動対応性の視点



反復ごとに開発対象の機能を決定します。

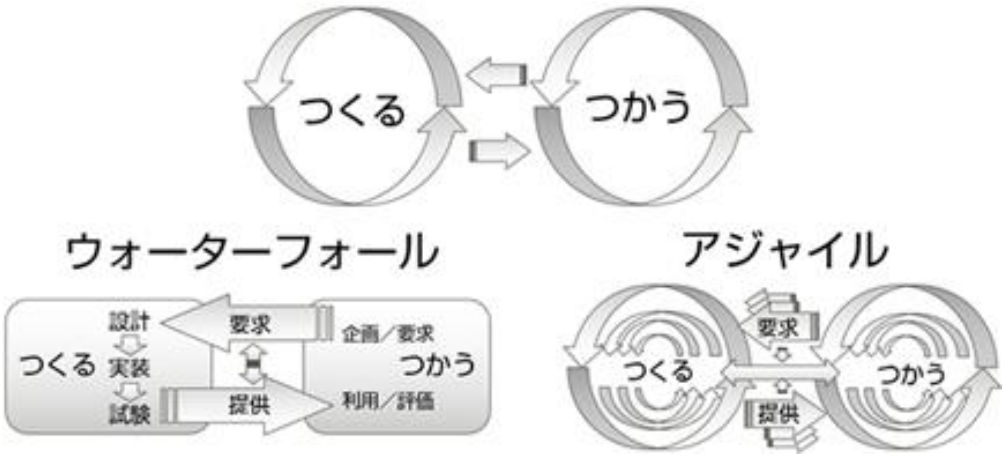
確定していない要望や変更の可能性が高い機能は、次回以降の反復で開発するかどうかが決定されるため、要求の変化に柔軟に対応できます。また、不必要な機能をつくることもありません。

品質と顧客満足度の向上 ～ リスク早期検知の視点



機能を一つ一つ順に利用可能な状態まで完成させます。要望の誤解、技術的な実現可能性や品質の検証が早期に実施されるため、故障も少なく満足度の高いソフトウェアが提供されます。

開発者やチームの成長を促進 ～ 学習の視点

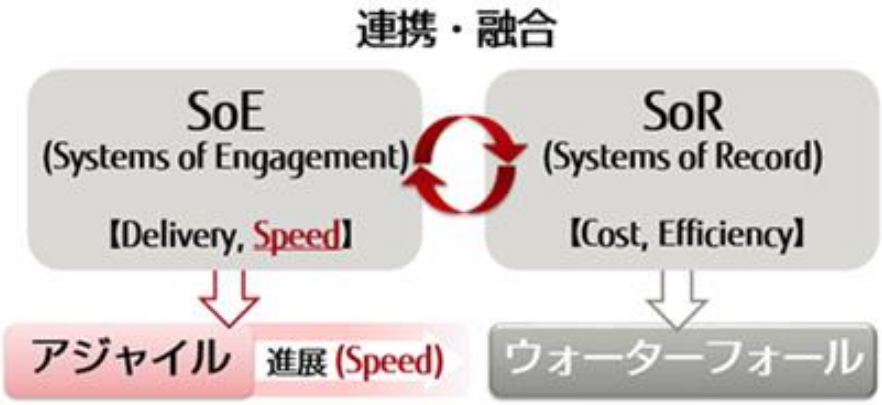


機能を一つ一つ開発することにより設計、実装、テストのサイクルを繰り返します。また実装専任やテスト専任などの役割分担がありません。多様な作業を何度も体験することができるため“多能工化”、“技術力向上”、“士気の向上”など「開発者の成長」を促すとともに、“プロセスの改善”、“マネジメント力の向上”など「チームの成長、学習」を促します。また、学習の効果は開発者つまりつくり手だけにとどまりません。新しいつくり方のほかに新しいつかい方を見つける機会も増加します。そして、つくり手つかい手相互のインタラクションの増加を促します。

ii. SoE 領域から SoR 領域へと広がるアジャイル開発

アジャイル開発は前述の特長によるメリット、その魅力によって広範に展開を続けてきました。

スピードが要求される SoE 領域からはじまり、その SoE との連携融合が進む SoR 領域にも、SoE のスピードに追従する必要からアジャイル開発が展開しつつあります。

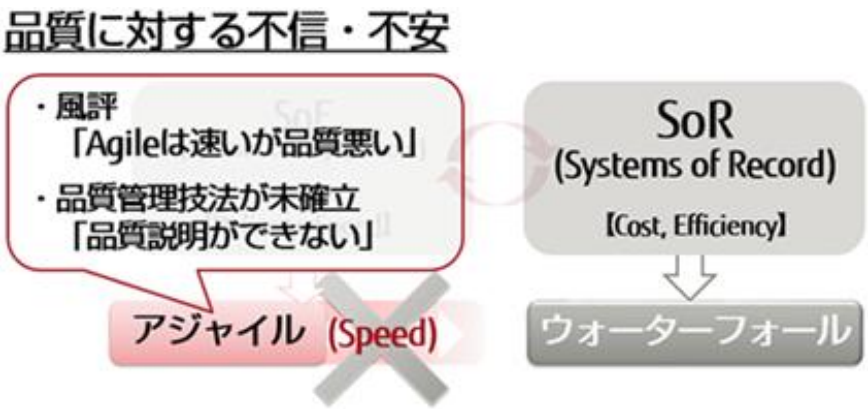


アジャイル開発の品質

しかし、アジャイル開発の展開が進みつつある中で、アジャイル開発の品質について不信や不安もあるようです。

- アジャイルは速いが品質は悪い
- 品質説明が出来ない

などの評判が品質不安を誘い、特に、可用性や高信頼性が強く求められる SoR 領域への進展は遅れています。



発生しつつあるビジネス上の問題

品質不安からアジャイルの進展が遅れてしまうと、冒頭に述べたメリットを享受できなくなります。そしてその結果、競合他社との競争優位性を得られなくなってしまうのです。

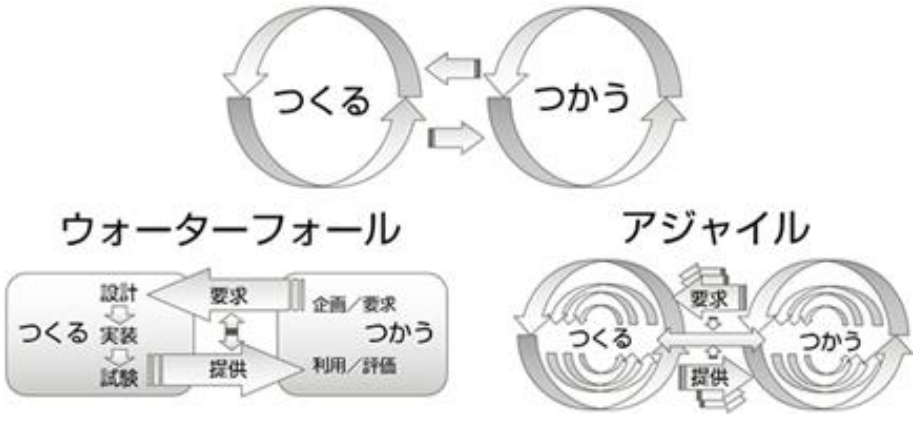
このような評判や不安を引き起こしてしまっているのはなぜでしょうか？
アジャイル開発の品質はどのように管理されているのでしょうか？

iii. 品質管理

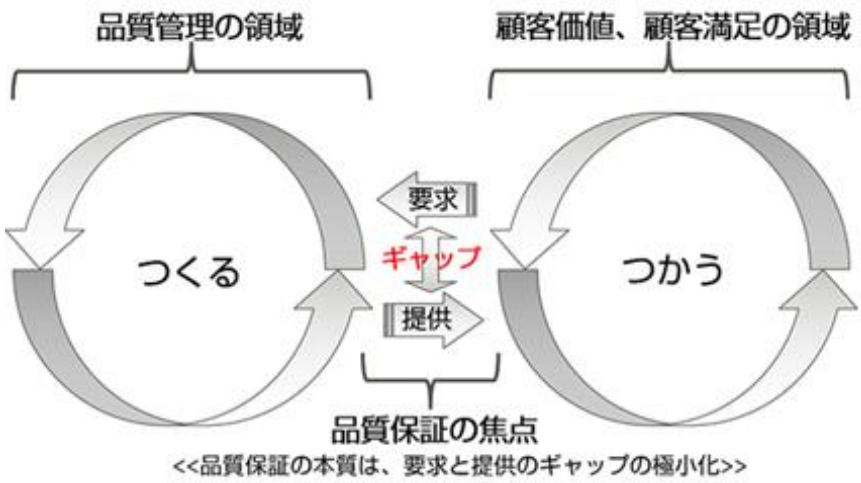
では、品質管理とはどのようなものかを見てみましょう。あわせて品質保証も。

学習とインタラクション

下の図は、さきほど提示したアジャイルの特質の１つ「学習」です。



反復増加型開発によるつくり手とつかい手のインタラクションは、品質を高めるきっかけです。



品質管理と品質保証、さらには品質マネジメントと、ISO に明確な定義はあるようですが、定義が広かったり被ったりしている日本流の概念も古くから存在します。

品質管理と品質保証

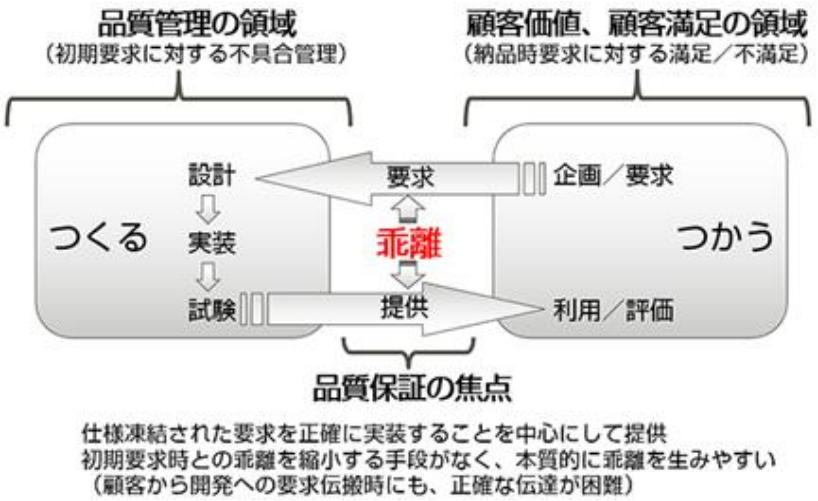
ここでは、品質管理と品質保証を分けて考えます。

- 品質管理は、出来栄えをみながらつくり方を微調整すること、それを繰り返すこと。
- 品質保証は、お客様や市場の求める品質を会社として請け合うこと。

（品質保証にはバグが無いこと以外に、求められているものを提供するということも含みます）

品質管理（QC：Quality Control）	品質保証（QA：Quality Assurance）
ソフトウェアを目標の品質にするための諸々の活動のこと 以下の活動を反復的に実施することによって実現される。 1. ソフトウェアの開発（および改修） 2. ソフトウェア開発メトリクスの測定 3. 試験等による品質特性の測定 ソフトウェア開発プロセスの微調整（1.へフィードバック）	お客様の求める品質を会社として請け合うための諸々の活動のこと 以下の活動を反復的に実施することによって実現される。 1. 目標の品質（＝お客様が求めるであろう品質）を設定 2. 品質管理をともなったソフトウェア開発 3. 品質保証部門の検証により、お客様、市場への提供可否判断 ソフトウェアをお客様、市場へ提供、反応を採取（1.へフィードバック）

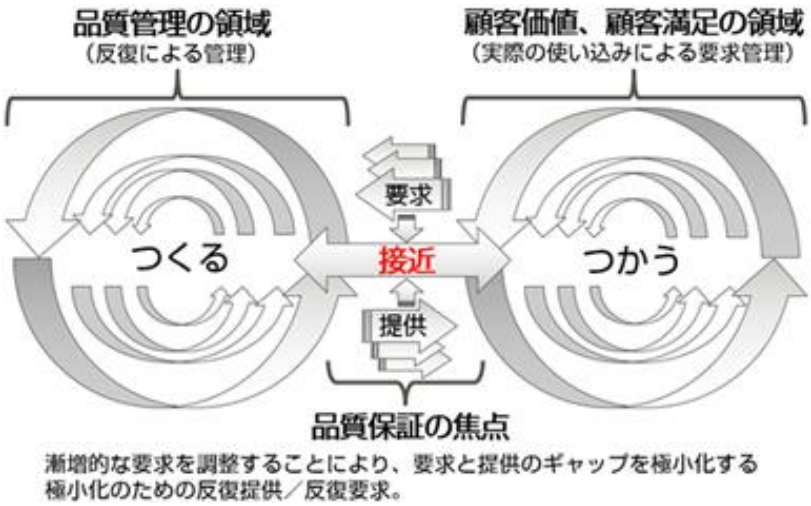
ウォーターフォール開発の品質管理と品質保証



ウォーターフォール開発の場合は、凍結された要求の正確な実装・提供を狙うので、初期要求との乖離を縮小する手段がなく、本質的に乖離を生みやすいといえます。顧客から開発への要求伝達時にも、正確な伝達が困難です。また、仕様凍結されているので、品質保証の活動チャンスが少ないといえます。

結果的に、ウォーターフォール開発の品質保証は品質管理の領域とほぼ同一になってしまいそうです。

アジャイル開発の品質管理と品質保証



アジャイル開発の場合には、少なくとも反復ごとには要求の調整が可能です。このため要求の提供のギャップを最小化するためのチャンスが幾度かあります。また、最小化のための反復である。ともいえそうです。

2.2 統計的品質管理の応用

i. 品質管理技法

ここでは、「アジャイル開発の品質管理技法」について述べます。

統計的品質管理

ソフトウェア開発に適用される品質管理技法の基本は、統計的品質管理です。ウォーターフォール開発にも伝統的に適用されているばかりか、ハードウェアの開発、生産にも適用されています。

ここからは、アジャイル開発の品質管理を統計的品質管理の観点から見てみましょう。

統計と統計的品質管理

統計的品質管理とは、統計的方法を用いて品質管理や工程改善を推進することです。

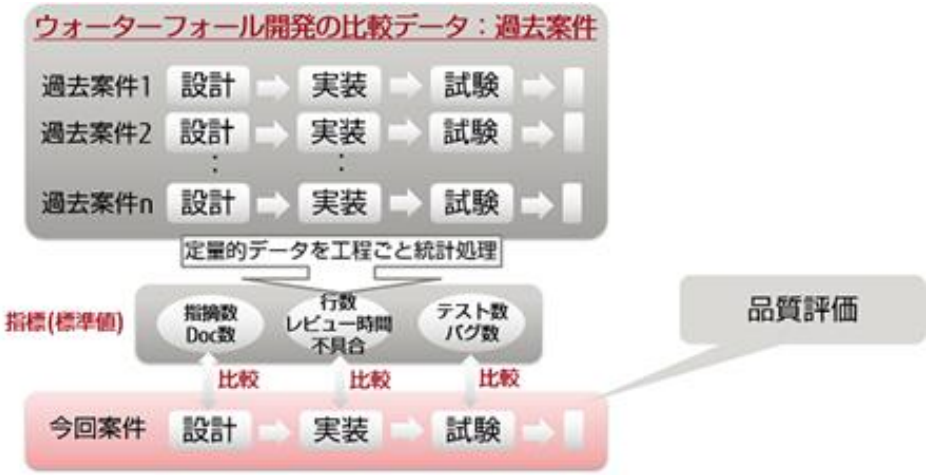
そして、統計とは

1. まず定量的なデータを採取
2. データを指標化
3. 指標から定性的傾向を検知
4. 傾向から実態の糸口を見つける

というように定量的なデータから定性的な傾向を見出すことです。

ウォーターフォール開発の統計的品質管理

伝統的なウォーターフォール開発では、過去の案件の定量的データを工程ごとに統計処理し、標準的な指標を得ます。そして、それらの指標の標準値と対象となる案件の指標を比較して品質を評価します。



しかしこれらの指標は品質をあらわしているといえるのでしょうか？

ドキュメントの枚数やコードの行数、レビュー時間は品質そのものではなさそうです。 では一体なぜ、品質評価にこれらの指標を用いているのでしょう？

品質評価には代用特性を利用

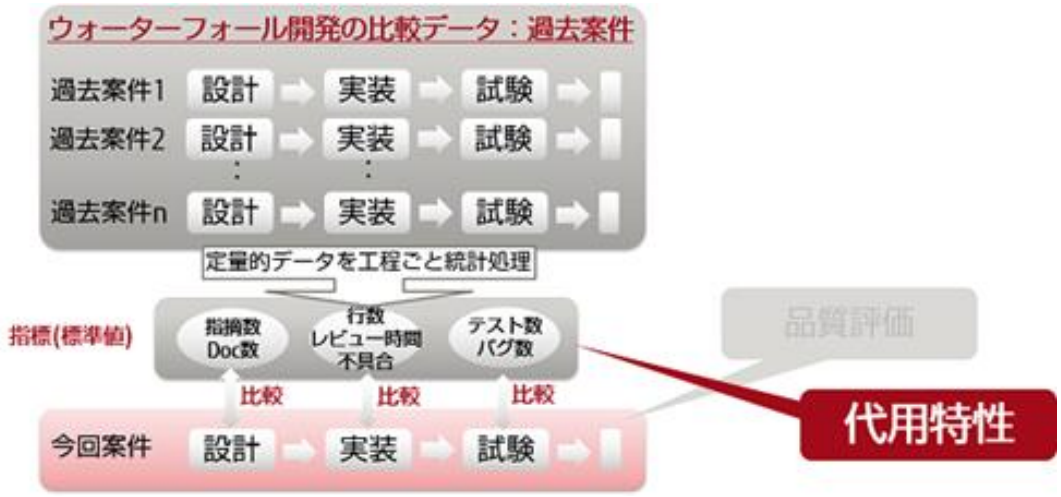
代用特性という考え方があります。

> 代用特性：ある特性を直接測定できない場合に、対象の特性と連動する他の特性を観測することで代替する特性

測りづらいもの、たとえば温度がそうでしょう。

温度計は、温度を測っているのではなく、アルコールの体積を測っています。温度と体積の連動性を代用特性として利用しています。

ウォーターフォール開発の品質管理で用いた指標も実は代用特性であり、品質と連動性の高い指標を利用しているのです。



アジャイル開発にも統計的品質管理は適用できるのか？

ソフトウェアの指標に関し、興味深い発言があります。

日本ソフトウェアシンポジウム 2017 の招待講演で奈良隆正氏は、こう語っています。

> ソフトウェアメトリクスは代用特性であり、開発における測定はコアコンピタンスにも拘わらず、慣習によって意味を理解せず計測していないか？

アジャイル開発については、こんなことがいえそうです。

> アジャイル開発におけるソフトウェアメトリクスも代用特性であろう。しかし、ウォーターフォール開発とは異なる開発プロセスのアジャイル開発においても、慣習によってウォーターフォール開発と同様の計測・評価をしていないか？

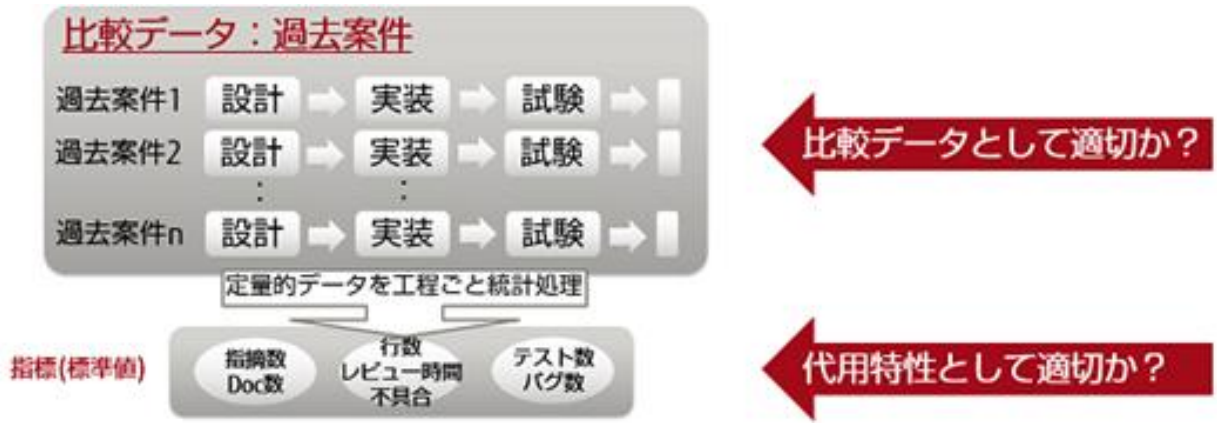
ウォーターフォール開発とアジャイル開発、その開発プロセスを比較してみます。

	利用技術	プロセス	反復性
ウォーターフォール開発	限定された技術 ・OS/言語/Platform	標準化が進んでいる ・成功事例の横展開 ・フレームワーク化	なし
アジャイル開発	多様な技術 ・新技術の登場に追従	多様性が拡大している ・採用技術に依存（設計/試験） ・各プロジェクト固有	あり

ウォーターフォール開発は利用技術もプロセスも、複数のプロジェクトを通してわりと似ていると言えそうです。しかし、アジャイル開発では、プロジェクトごとに技術もプロセスも多様ではないでしょうか？

異なる開発プロセスに同一の指標を利用するのはやはり不自然です。アジャイル開発に品質管理技法を適用する場合には、指標の意味に注意を払う必要がありそうです。

注意を払うべき点は以下の二点でしょう。



アジャイル品質管理技法の適切な利用

アジャイル開発を SoR 領域など、より広範囲に展開するためには、アジャイル開発への品質管理技法の適切な適用が必要です。そのためには、前述の二点「比較データの適切性」と「代用特性の適切性」を考慮した、アジャイル品質管理技法が必要となるのです。

ii. アジャイル開発のための品質管理技法

ここからは以下の二点

- 「比較データの適切性」
- 「代用特性の適切性」

これらを考慮した、アジャイル品質管理技法について述べます。

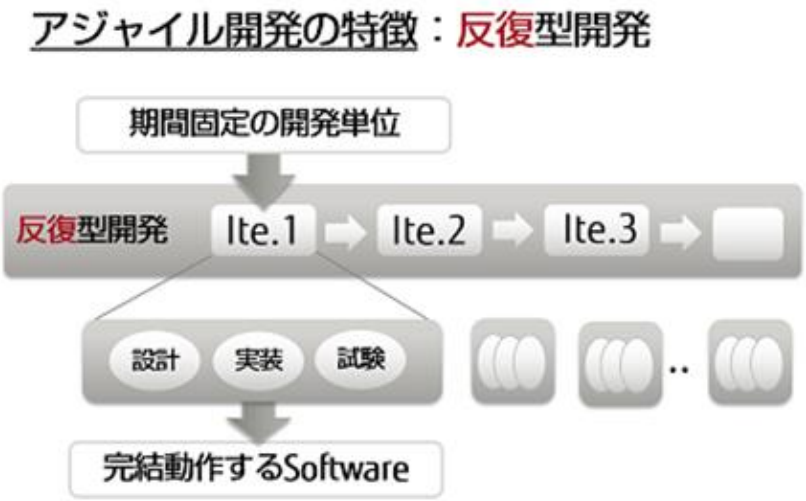
比較データの適切性

比較データとは比較の相手になるものです。

アジャイル開発プロジェクトは、他のプロジェクトとの類似性が少ないことをすでに述べました。つまり比較の相手がいないということになります。

そこで、似ている何かを見つけるために、反復性に着目します。

	反復性
ウォーターフォール開発	なし
アジャイル開発	あり



アジャイル開発はイテレーション（あるいはスプリント）と呼ばれる反復期間の中で、設計・実装・試験が同時進行し、反復期間の終わりには完結動作するソフトウェアがアウトプットされます。

つまり、反復開発です。

反復期間ごとのプロセスにはそれぞれ強い類似性があります。

類似性項目	内容
開発メンバー	ほぼ同一メンバーによる継続的開発
開発プロセス	改善による変化はあるが、同一プロジェクトであるためほぼ同一プロセス
利用技術	期間中の新技術導入はあるが、同一プロジェクトであるためほぼ同一技術

メンバー、プロセス、技術はプロジェクト期間をつうじてとてもよく似ています。

適切な比較データ

> アジャイル開発に適切な比較データは、重ねた反復である

といえそうです。つまり、最近の自分たちと比較することです。

代用特性の適切性

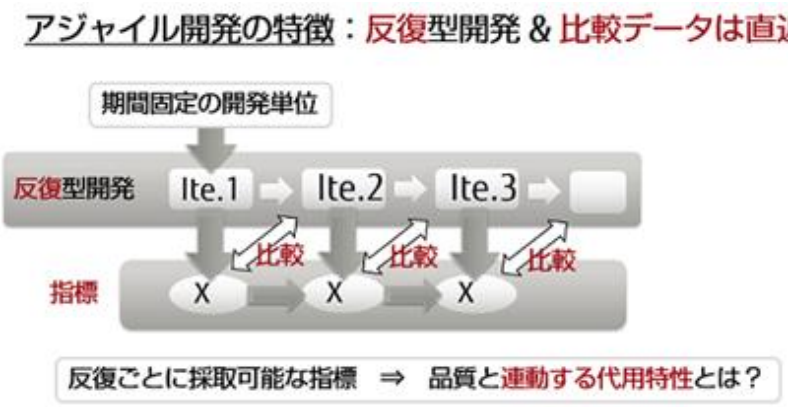
代用特性とは比較の物差しになるものです。

代用特性についても、やはり反復性に着目します。

	反復性
ウォーターフォール開発	なし
アジャイル開発	あり

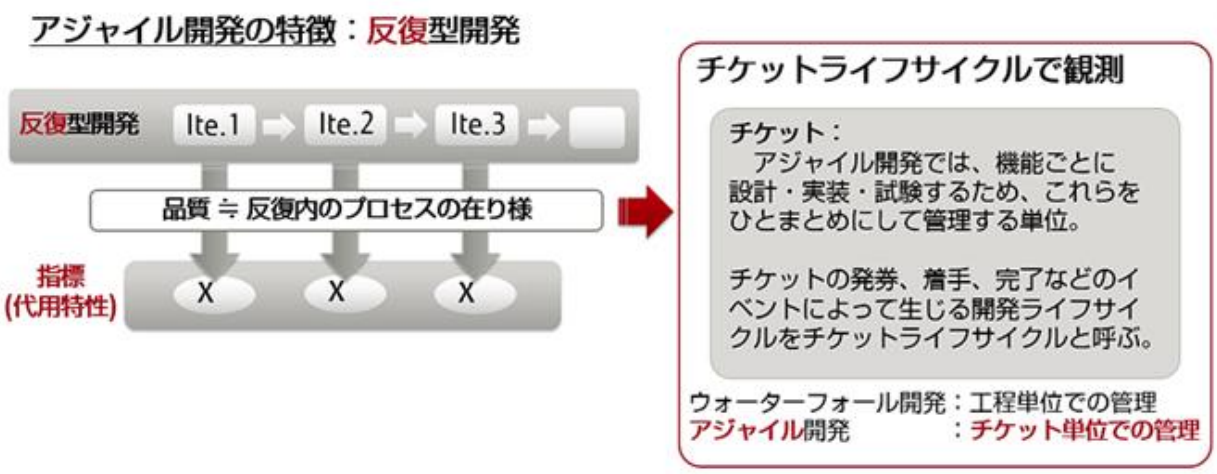
アジャイル開発は反復開発ですから、少なくとも反復期間ごとに品質データを採取可能です。つまりメトリクスが採取可能です。

この指標を反復ごとに、あるいは過去の反復すべてと比較することが可能です。



完結動作するソフトウェアを開発するのがアジャイル開発プロセスです。

そのプロセスがソフトウェアをアウトプットするので、ソフトウェアの品質はプロセスの在り様、反復内のプロセスの在り様と連動していそうです。



プロセスの在り様は、チケットライフサイクルで観測が可能です。

アジャイル開発では機能ごと 1 つずつ開発するため、多くの場合に管理の単位にチケットを使います。チケットの発券、着手、完了などのイベントで生じるライフサイクルをここでは、チケットライフサイクルと呼びます。

このチケットライフサイクルに着目すると、

- リードタイム
- 着手待ち時間
- 発券タイミング

などが指標として想定できます。割込み、障害や、新機能の要求発生などのイベントがプロセスの変化や乱れとして、チケットライフサイクルに現れるためです。

指標	変化の要因
チケットリードタイム	割込み事象による優先順位下落、待ちによる遅延
チケット着手待ち時間	障害発生による発券即着手
チケット発券タイミング	反復開始時の計画的発券と、反復期間中の随時発券のバランス （健全な改善事項の検出による随時発券か、不用意な障害の検出による随時発券か）

アジャイル開発に適切な代用特性は、そのプロセスの特徴に起因するチケットライフサイクルだといえます。

アジャイル開発の品質管理に必用なもの

アジャイル開発の統計的品質管理に必用なものは、ウォーターフォール開発で利用されてきた比較データや代用特性ではありません。アジャイル開発では、

比較データ：自プロジェクトの重ねた反復データ
代用特性：チケットライフサイクル

これらの比較データや代用特性の利用が必要です。

2.3 アジャイル品質管理技法の検証と考察

i. 品質管理技法の検証

ここでは、ここまで述べた品質管理技法を具体的に検証します。

対象プロジェクト

対象プロジェクトは SoR 領域をアジャイル開発するプロジェクトで、メンバー数は 10 名前後です。

SoR 領域と SoE 領域はデータ連携があり、SoE のプロジェクトからは新たな要求が随時発生します。

	特長
メンバー数	10 名前後
チーム齢	1 年以上
開発プロセス	SoR、SoE ともにアジャイル開発
要求変化	SoE 側から要求変更頻発

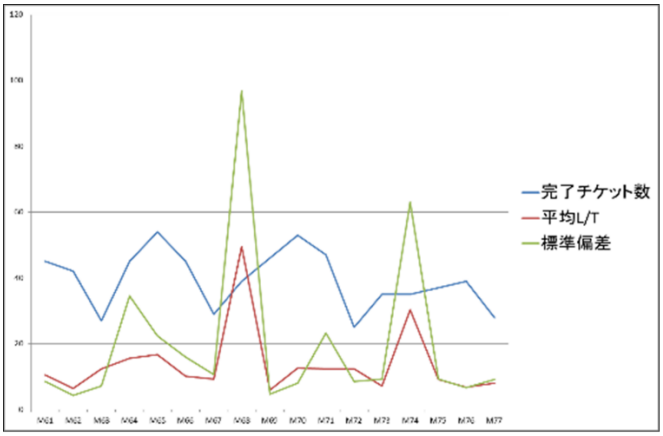
ii. 検証結果

対象プロジェクトが重ねた反復のデータから、チケットライフサイクルと以下 3 点との連動性を観測しました。

リードタイムと手戻り 障害発生とチケット着手待ち時間 品質とチケット発券時期

リードタイムと手戻り

1 点目は、手戻りとリードタイム（L/T）です。



完了チケット：
概ね一定数で推移
平均L/T、標準偏差：
ピーク2か所
（チケット粒度：均一）
↓
長L/Tチケットの存在
↓
当該チケット精査
↓
多数の手戻り履歴

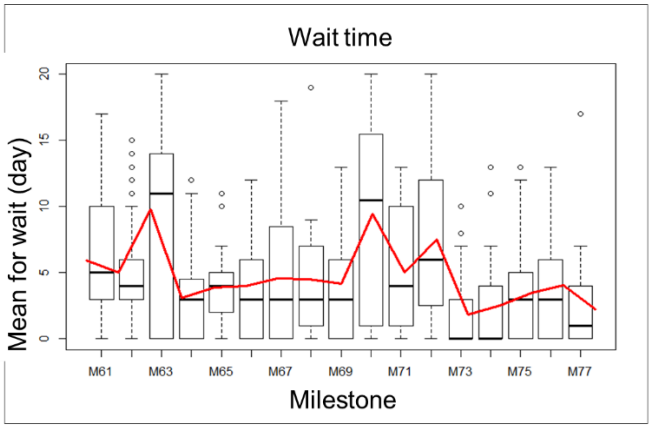
この折れ線グラフ、横軸はアジャイル開発における反復期間です。

青線は完了チケット数を示しています。また、赤線は平均リードタイムで、緑線はそのリードタイムの標準偏差を示しています。完了チケット数（青）は、概ね一定数で推移していますが、平均リードタイム（赤）とそのバラツキ（緑）は、ピークが二か所検出できます。このピークは長いリードタイムのチケットが存在することを意味します。

長いリードタイムのチケットの内容をみると、多数の手戻り履歴を確認することが出来ました。

障害発生とチケット着手待ち時間

2 点目は、障害発生と着手待ち時間です。



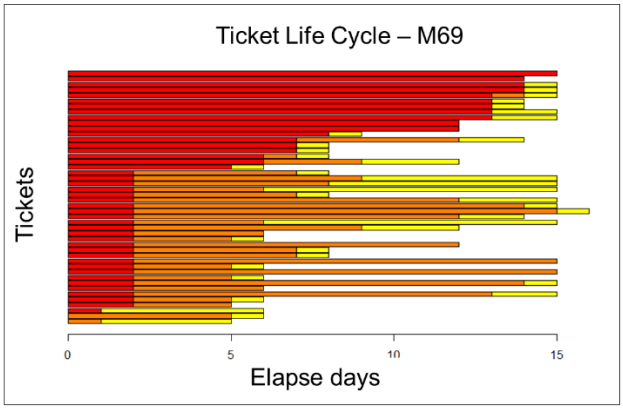
折れ線グラフ：
平均待ち時間
箱ひげ図内の太線：
中央値（底にべた付き）
↓
待ち時間＝0のチケット多数
↓
即日発券、即日着手
↓
当該チケット精査
↓
障害/考慮漏れ発生

横軸はアジャイル開発における反復期間です。各反復のチケット着手待ち時間を、箱ひげ図で表してみました。

赤の折れ線は平均待ち時間です。箱ひげ図内の太線は中央値です。開発期間の後半（つまりグラフの右側）で、中央値が底にベタ付きしています。ベタ付き、つまり多くのチケットの着手待ち時間がゼロです。これはチケットの即日発券、即日着手を意味します。

着手待ち時間がゼロのチケットの内容をみると、障害／考慮漏れを起因とするものでした。

チケット着手待ち時間についてもう少し観察するために、チケットライフサイクルの積上げ図（下図）を見てみます。横軸は経過日数、縦に各チケットを積み上げています。



健全なイテレーション

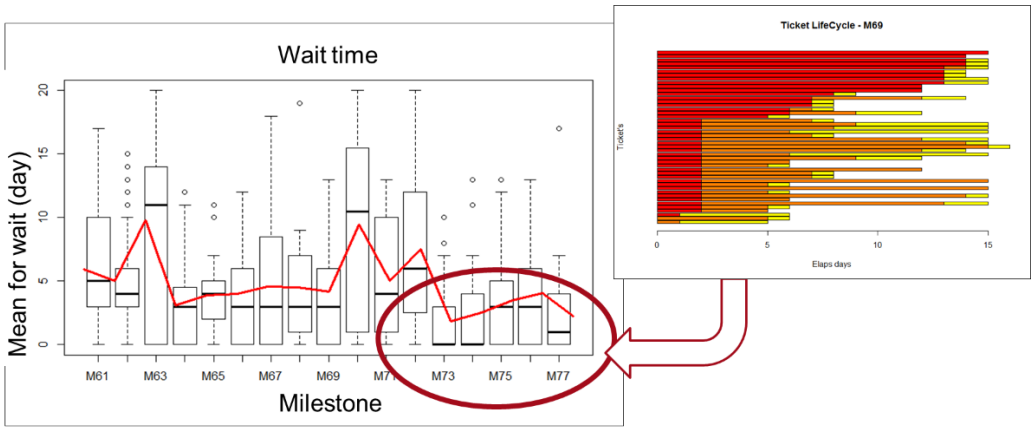
赤：未発券
橙：着手待ち
黄：実行中

計画発券と随時発券の均衡

随時発券は…
健全な発見か？改善か？
それとも
障害か？考慮漏れか？

着手タイミングは赤と橙の境目です。

下方のチケットは赤と橙の境目すなわち着手タイミングが縦に揃っており、同時に計画的に発券されていることがうかがえます。一方、上方のチケットは境目が乱れています。乱れはチケットの随時発券着手を示しています。



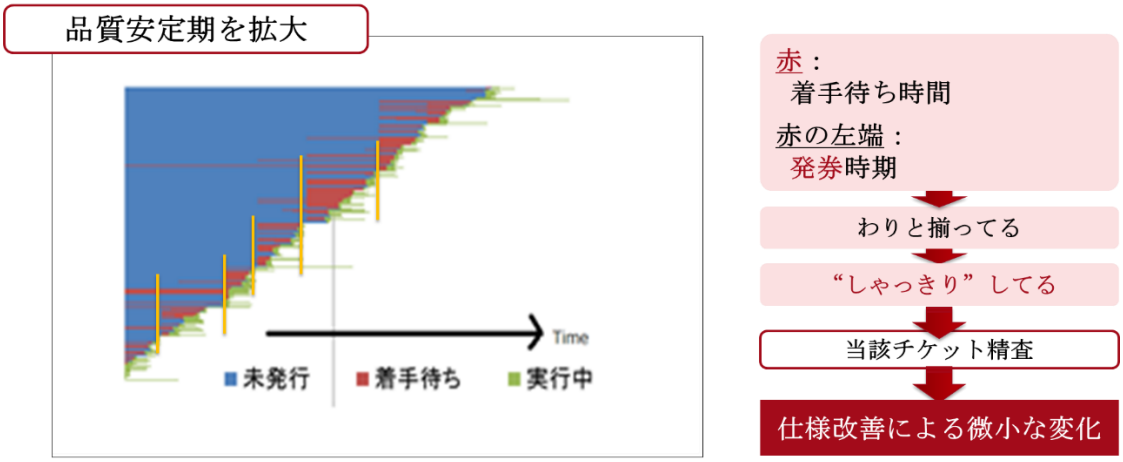
この箱ひげ図の中央値が底にベタ付きしているのは、即日発券、即日着手という、予期しないチケット発券が頻発していることを示しています。つまり、障害発生によるプロセスの乱れを示しているといえます。

品質とチケット発券時期

最後、3 点目は品質安定とチケット発券時期です。

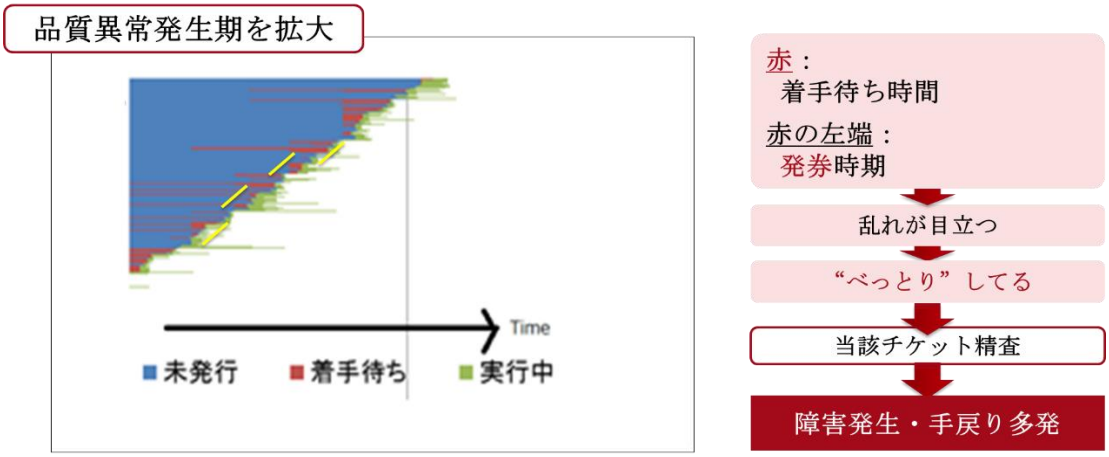
チケットライフサイクル図を長期間積上げてみました。

品質安定期を拡大して見てみます。赤の左端、赤と青の境目がチケットタイミングです。



品質安定期には、発券タイミングは比較的揃っており「しゃっきり」とした形状が観察できます。この時期のチケットを観察すると、仕様改善による微小な変更、微小な随時発券／即日着手があるのみでした。

対して、品質異常期を見てみると、



発券タイミングに乱れが目立ち「べっとり」とした形状が観察できます。この時期のチケットを観察すると、障害発生や手戻りが多発していました。

チケットライフサイクルの連動性

チケットライフサイクルと（つまり開発プロセスの在り様、あるいはソフトウェアのつくり方と）、手戻りや障害発生といった品質の連動性を 3 点観測することができました。定量的なデータから、品質の変動をうかがい知ることができた。といえます。

iii. 検証に対する疑問

しかし、品質をみるにあたり下の疑問があるでしょう。

- しゃっきり？べっとり？
- 定量的評価なのか？
- 個別に精査するのか？

オノマトペ

擬音語や擬態語といったオノマトペを、品質管理にふつうに使う人たちが多くいるようです。人の生命にかかわる、ハードウェア・ソフトウェアの仕様として使われています。

計測された数値ではなく、視覚や聴覚など人の感覚を言葉で表現して正常時と異常時の違いを判別しているのです。

定量的な評価

この項の検証では、グラフを目視しての判断を実施しました。 しかし、先ほどこう述べました。

統計的品質管理とは、統計的方法を用いて品質管理や工程改善を推進することです。そして、統計とは

1. まず定量的なデータを採取
2. データを指標化
3. 指標から定性的傾向を検知
4. 傾向から実態の糸口を見つける

というように定量的なデータから定性的な傾向を見出すことです。

定量的なデータは、実態を探る糸口を見つけることしかできないのです。

指標値は再点検の目安でしかありません。指標からは、なにか想定外の起きている可能性を検知することしかできないのです。判断するための指標では決してないのです。

統計的品質管理とはそういうものです。ウォーターフォールもハードウェアもそうです。指標値（品質メトリクス）が、既定範囲に収まっているかどうかで品質の良し悪しを判断することはできません。

それぞれのプロジェクト・チームが、それぞれの技術、それぞれのプロセスで開発するアジャイル開発は、指標も、プロセスも、人それぞれ、チームそれぞれであることが、自然です。

iv. 考察

なぜチケットライフサイクルと品質の連動性が観測できたのか？以下の 3 点が必要条件だと考えます。

- 比較データの正確性
- データ取得の容易性
- データの可視化

比較データの正確性

1 つ目は、比較データの正確性です。これに必要なことは、

- ライフサイクルイベントが明確に規定されていること
- 機能設計能力と試験（ソフトウェアテスティング）能力によって適切なチケット粒度が保てること

です。

データ取得の容易性

2 つ目は、データ取得が容易であることです。

- チケット管理ツールによって、チケットステータスの変更イベントすなわちライフサイクルデータを容易に採取できること
- ステータス変更の入力が開発サイクルの一環であること、つまり、データ入力自体が開発プロセスに内包／ビルトインされていること

データの可視化

最後 3 点目は可視化です。

- グラフ化によって、第三者からも確認が容易であること
- オノマトペなどによって、直感的認識が共有しやすいこと

です。

v. 残された問題

ここまで述べた品質管理技法が適用できないチーム・プロジェクトが存在します。

チケット粒度

チケット粒度が粗く不揃いなチーム・プロジェクトには適用できません。その解決には、適切な粒度を保つための

- 機能設計能力
- 試験（ソフトウェアテスティング）能力

が必要です。

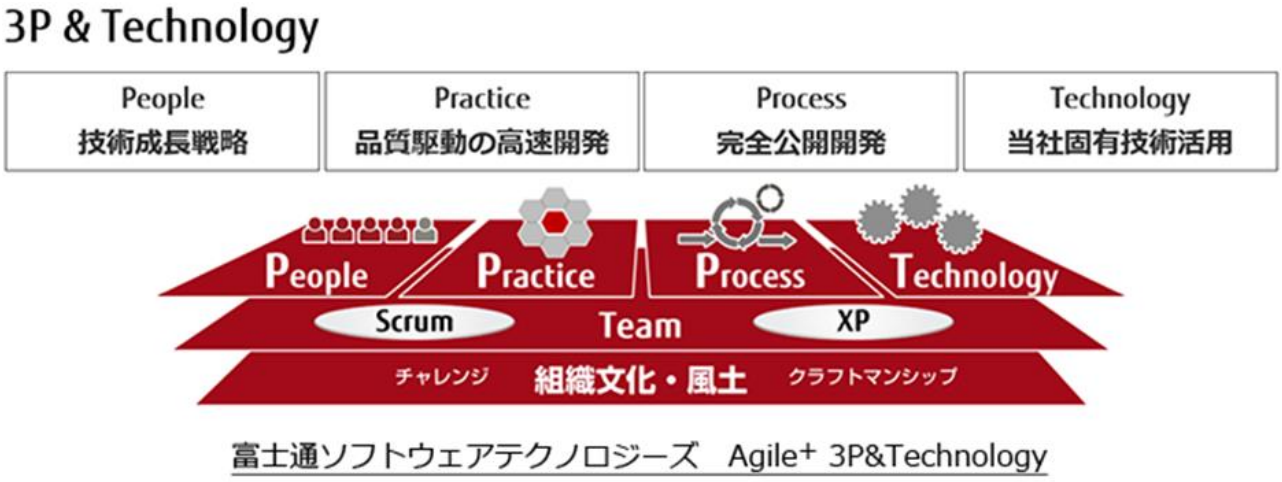
正確な入力

データの正確な入力もこの技法の適用には必要です。

- 真摯で誠実な開発者
- クラフトマンシップに溢れたリーダー
- 真偽を見極められるマネージャー

といった正直さと正直さを維持し続ける人の態度が必要です。

検証対象プロジェクトで連動性が観測できたのは、組織の根底にある「組織文化や風土」が原因ではないか？組織文化や風土が、彼ら開発者やリーダー、マネージャー資質を醸したのではないかと考えます。



残された問題を解決するには、

- 品質管理技法の適用可能なチケット粒度を保てる設計／試験能力
- 正直な入力をふつうにできる組織文化や風土

の 2 点が、必要です。

vi. 品質説明の要諦

アジャイル開発の品質説明の勘所はいったいどこにあるのか？それは、以下の 2 点に尽きます。

- 品質と連動する代用特性を見つける。
- 代用特性をウォッチしていつもと違う変化を捉える。

さきほど、ソフトウェアメトリクスは代用特性であると述べました。しかし、品質と連動する代用特性はプロジェクトごとに異なります。

品質と連動性の高そうなメトリクスにあたりをつけて観測し、代用特性であることを検知すること。
そして、その代用特性を継続的にウォッチしていつもとは違う変化を捉えることが重要です。

代用特性に変化が見られた場合には、当該時期のチケットやソースコードを注意深く観測し、変化の原因を見極め、適切に対応することが必要です。

「品質と連動する代用特性をウォッチして変化を検知し対応する」

これは、アジャイル開発に限らず、ウォーターフォール開発でも同様です。アジャイル開発とウォーターフォール開発の違いは、比較データや代用特性がそれぞれ異なるだけです。ウォーターフォール開発での

品質説明となにか異なる点があるでしょうか？もし異なると感じたのであれば、それは、「慣習によって意味を忘れてしまっている」 のではないでしょうか？

第3章 大規模アジャイル開発の可能性

3.1 ウォーターフォールは大規模向きだったのか？

アジャイル開発の普及にともなって、SoE 領域から SoR 領域へとアジャイル開発は進展しています。さらには、大規模開発の領域にもアジャイル開発は進展しつつあります。まずここではウォーターフォール開発のスケーラビリティーについて、アジャイル開発のスケーラビリティーと比較しながら考察します。

i. 規模とリスクの因果関係

ウォーターフォール開発には規模に向き合い、多くのミッションクリティカルなシステムを構築してきた実績があります。しかし、大規模になれば開発対象となるシステム／ソフトウェアの複雑度が高まります。

- いかにつくるか？　～　構造、機能の複雑性
- なにをつくるか？　～　要求、要件の複雑性

大きくはこれら 2 つの複雑性が、大規模化のリスクでしょう。さて、規模とリスクはどんな関係なのでしょうか？

規模とリスクの関係

規模とリスクの関係について、以下の 2 つが挙げられるでしょう。

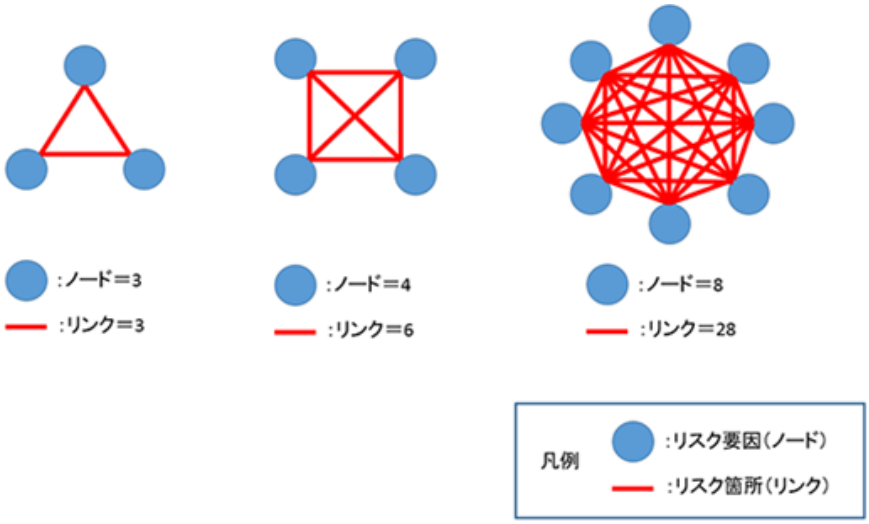
- 規模増加にともなってリスクは増大する。
- リスクを増大させる要因はいくつかある。

「機能、構造の複雑性」、「要求、要件の複雑性」から発生するリスクが増大する要因としては、以下事柄が考えられます。

- ソフトウェアの構造、機能の依存関係の増加
- ソフトウェアコンポーネント数 ・ 機能数 など
- コミュニケーションパスの増加
- 人の数の増加 ・ team の数の増加 など

このようなリスクの要因（因子数）が増えると、実際のリスクはどのように増加するのでしょうか？

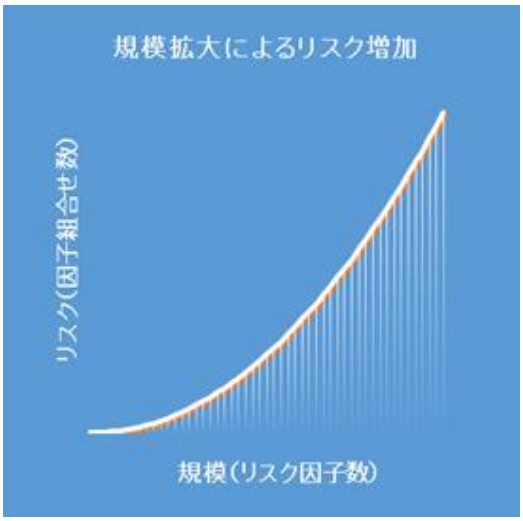
要因をノード、リスク発生個所をリンクと見立てたモデルが下図です。



因子組合せ数 = n×(n-1)/2　　(n:因子数)

リスク発生個所数（つまり因子組合せ数）は、因子数の 2 乗にほぼ比例して増加していくことがわかります。

このモデルをもとに規模とリスクの関係を示したグラフです。

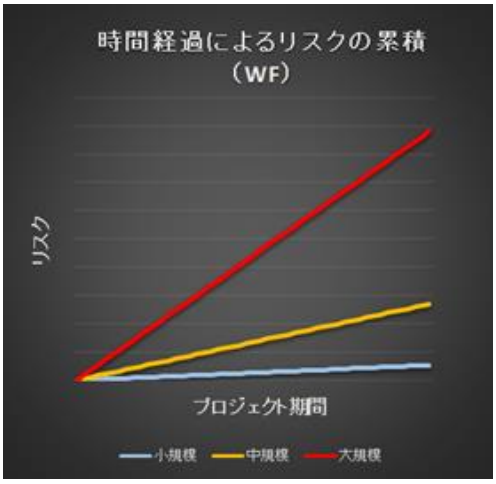


ii. ウォーターフォール開発の規模とリスク

まず、ウォーターフォール開発における、規模とリスクの関係をみてみましょう。

時間経過によるリスクの累積～ウォーターフォール開発

時間経過にともなうリスクの累積をグラフ化してみました。規模ごとに小規模（青線）、中規模（黄線）、大規模（赤線）で示しています。



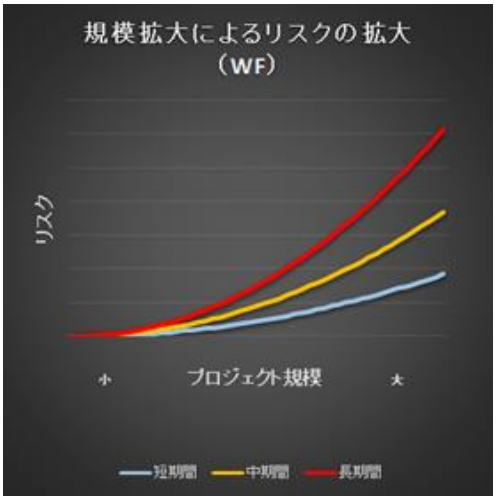
ウォーターフォール開発プロセスでは、リスクの顕在化時期が下流工程に偏在しているためプロジェクト期間を通じてリスク検知されることなく時間経過に従って累積します。（モデル単純化のため累積が最小であろう線形増加としています）

また、規模が大きくなるとともにリスク因子数も増加するため、グラフの傾きは激しくなります。

大規模化によるリスクの拡大～ウォーターフォール開発

視点を変えてみましょう。

規模の増加にともなうリスクの拡大をグラフ化してみました。開発期間ごとに短期間（青線）、中期間（黄線）、長期間（赤線）で示しています。



規模が大きくなるにつれてリスクは拡大します。冒頭述べたとおりです。規模の 2 乗に比例して大きくなります。また長期間になればなるほど、累積されるリスクは大きくなります。

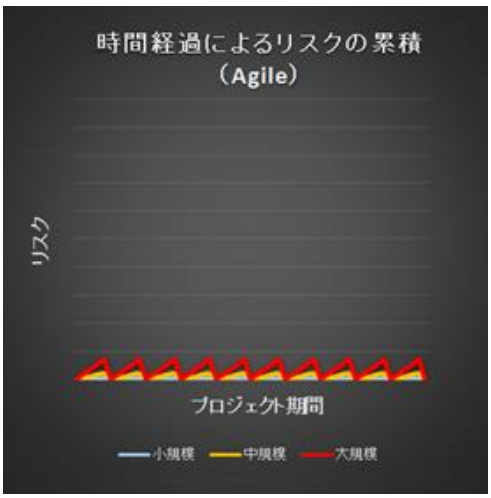
ウォーターフォール開発プロセスでは、リスクの顕在化時期が下流工程に偏在しているためです。

iii. アジャイル開発のリスク

アジャイル開発は反復型開発です。細かく期間を区切って開発します。ここでは、アジャイル開発における、規模とリスクの関係をみてみましょう。

時間経過によるリスクの累積～アジャイル開発

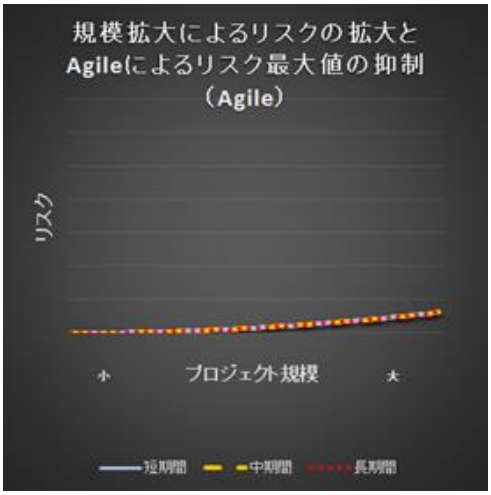
ウォーターフォール開発と同じく、アジャイル開発のリスクの累積をグラフ化してみました。規模ごとに小規模（青線）、中規模（黄線）、大規模（赤線）で示しています。



反復終了時には役に立つソフトウェア（working software）がアウトプットされるアジャイル開発ですから、少なくとも反復ごとにリスクは顕在化し、改修によってリスクはリセットされます。このため大規模であってもリスクの累積は極めて抑制されます。

大規模化によるリスクの拡大～アジャイル開発

アジャイル開発についても、規模の増加にともなうリスクの拡大をグラフ化してみました。開発期間ごとに短期間（青線）、中期間（黄線）、長期間（赤線）で示しています。

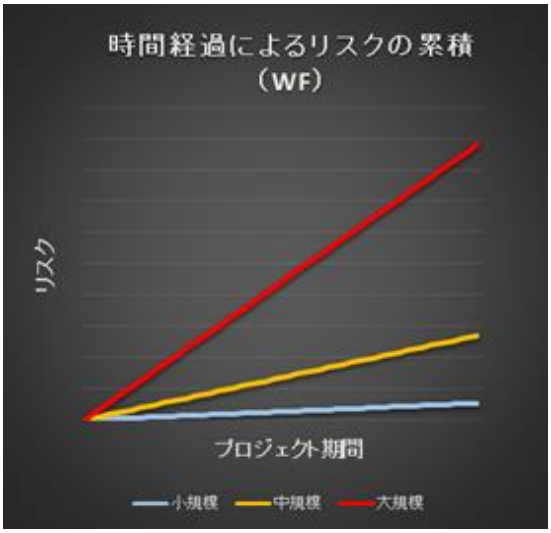


規模が拡大された場合であっても、リスクは極めて抑制されます。規模に関わらず、リスクは少なくとも反復期間ごとには顕在化されるため、極めて低減されたリスクだといえます。

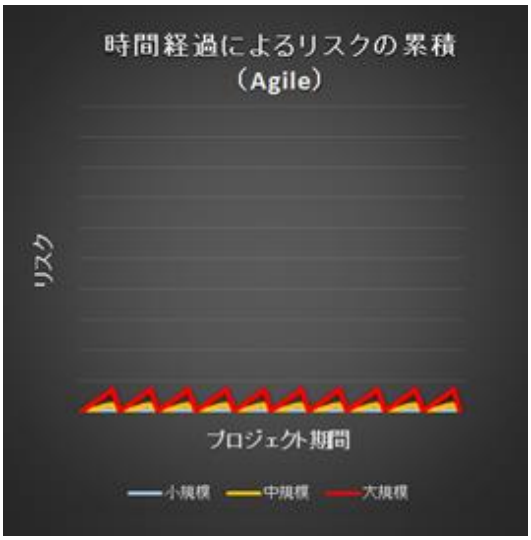
iv. 比較してみましょう

時間経過によるリスクの累積を比較

ウォーターフォール開発



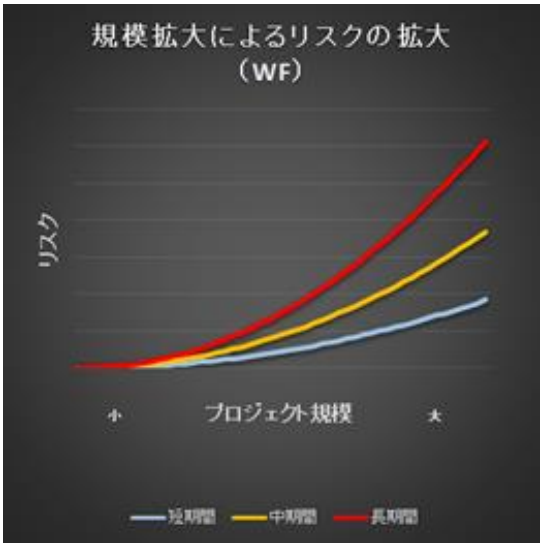
アジャイル開発



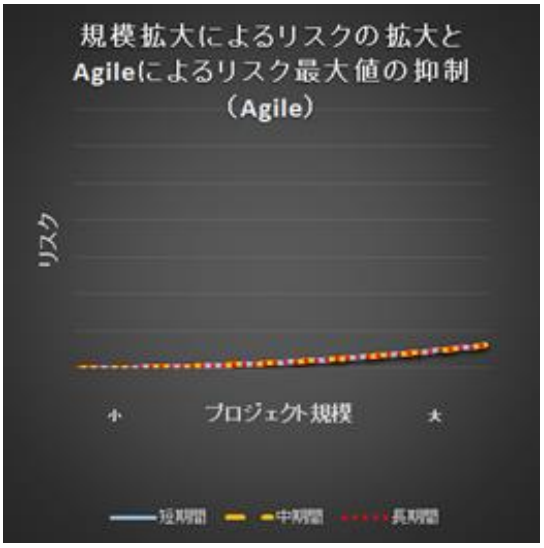
ウォーターフォール開発はプロジェクト期間内に累積され続けるのに対して、アジャイル開発では、極めて低いレベルで推移しています。

大規模化によるリスクの拡大を比較

ウォーターフォール開発



アジャイル開発



ウォーターフォール開発では大規模になるほど、リスクは規模の 2 乗に比例して大きくなります。対して、アジャイル開発では極めて低いレベルに抑制されます。

どちらがスケール容易か？

ウォーターフォール開発とアジャイル開発を比較すると、

- アジャイル開発は（ウォーターフォール開発よりも）スケール容易
- ウォーターフォール開発は（アジャイル開発よりも）スケール困難

と言えそうです。

v. ウォーターフォール開発はスケールしない？

IPA 独立行政法人 情報処理推進機構の発行する「ソフトウェア開発データ白書 2018-1019」を参照してみました。

大規模化による生産性の低下

白書の「図表 8-2-4 FP 規模別 FP 生産性の基本統計量（新規開発）」をみると、FP（ファンクションポイント）規模が 400FP を超えると、規模が拡大するにつれて生産性が低下していく傾向が見えます。また、「図表 8-2-3 FP 規模別 FP 生産性（新規開発）箱ひげ図」では、グラフ（箱ひげ図）化されており、この傾向をより直感的に認識できるのではないでしょうか。

規模拡大による生産性の低下は、本稿の冒頭に述べた、

- ソフトウェアの構造、機能の依存関係の増加
- ソフトウェアコンポーネント数 ・ 機能数

に起因しているであろうと考えられます。

要員数増加による生産性の低下

これも白書を見てみました。「図表 8-4-18 月当たり要員数別の FP 生産性の基本統計量（新規開発）」によれば、月当たりの要員数が増えるにつれて生産性が低下していく傾向が見えます。 また、「図表 8-4-17 月当たりの要員数別の FP 生産量（新規開発）箱ひげ図」では、著しく低下していく傾向が見て取れます。

要員数の増加による生産性の低下は、これも本稿の冒頭に述べた、

- コミュニケーションパスの増加
- 人の数の増加 ・ team の数の増加

に起因しているであろうと考えられます。

なぜウォーターフォール開発で大規模開発を成功させてきたのか？

データからも大規模化による生産性低下が認められるなか、なぜウォーターフォール開発が大規模開発を成功させてきたのでしょうか？

成功したと思っていただけでしょうか、いやそんなはずはありません。おそらく、

> 生産性低下はするものの、低下の度合いを経験的に見積れていた

ということではないでしょうか。多くの大規模開発を経験する中で、そのリスクを工数と期間の見積りに反映することが出来たということではないでしょうか。

3.2 アジャイル開発は大規模に耐えられるのか？

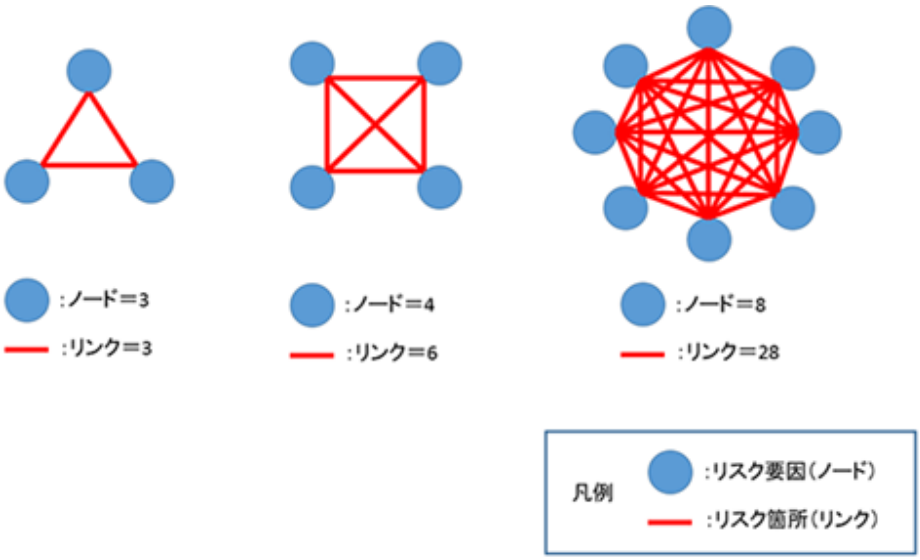
大規模開発に向いていると思われているウォーターフォール開発も、実は大規模化するにつれてリスクが高まると述べました。ここでは、アジャイル開発は「大規模化に耐えうるエンタープライズアジャイル」として成立するのか？について考察します。

i. アジャイル開発チームの適性規模

スクラムガイドによれば開発チームの適正規模は、9 名以下つまり 1 桁だと記載されています。しかし、10 名を軽く超えていても上手くやっているアジャイル開発チームはよく見かけます。（上手くいっているチームほど 10 名を軽く超えている印象を受けます）

果たして、10 名程度の人数で、個々のメンバーへの調整がチーム活動を阻害するほどのコストになるでしょうか？

さきほど、ノードとリンクの関係を下図のように示しました。

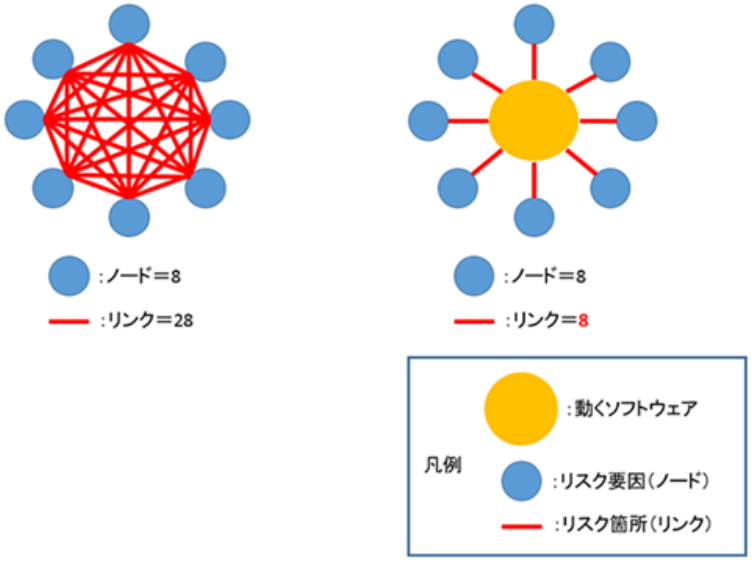


アジャイル開発の場合はどうでしょう？アジャイル開発チームの場合、多くは

- 常時結合（継続的統合：Continuous Integration）を実施
- コードの共同所有

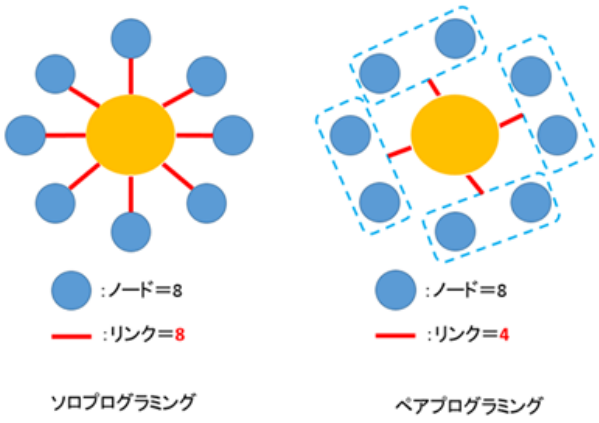
を実施しているようです。

であれば、下図の右側のようなコミュニケーションパスとなるのではないでしょうか？



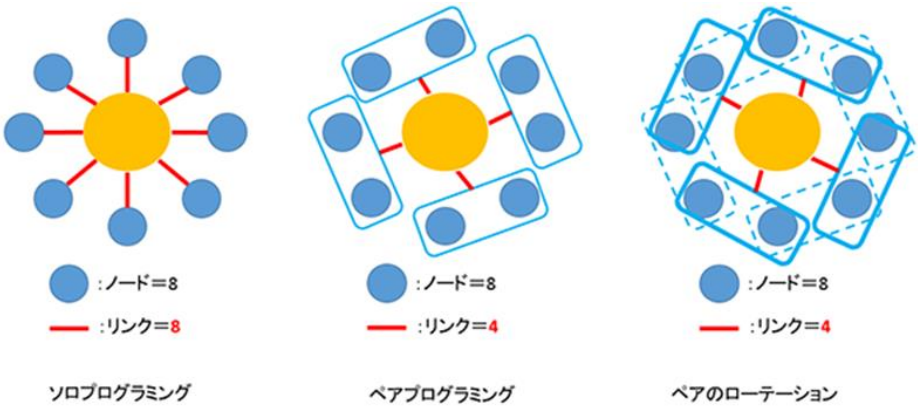
共同所有され、しかも常時結合された動くソフトウェアそのものを媒体にしたコミュニケーションであれば、コミュニケーションパスは人数の 2 乗に比例したりしないのではないでしょうか。

また、ペアプログラミングを実施しているのであれば、コミュニケーションパスは下図のようになるでしょう。



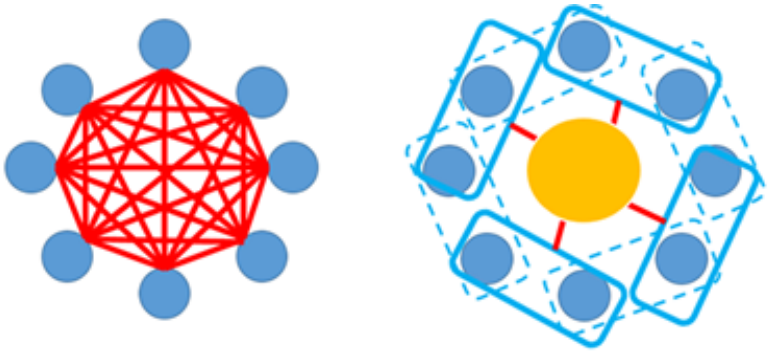
ペアプログラミングとは（大げさに言えば）脳を共有してプログラミングすることです。上図のように、コミュニケーションパスはソロプログラミングに較べて半減します。

さらに、ペアのローテーションを頻繁に実施するとなればどうでしょうか。



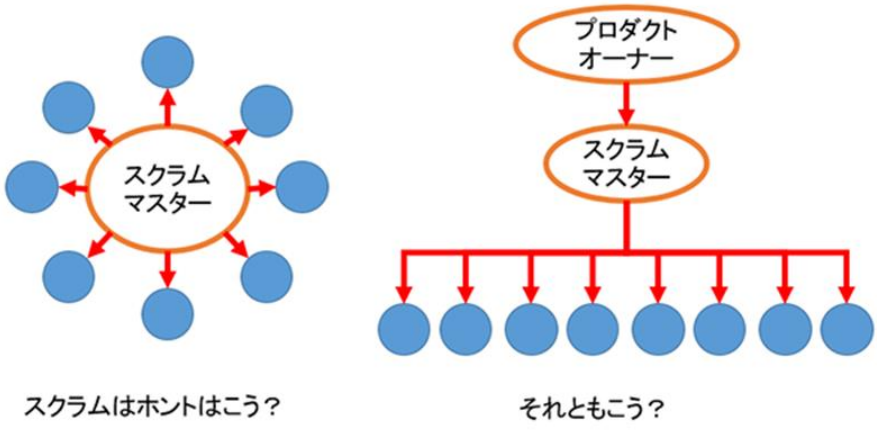
個々のメンバーへの調整がチーム活動を阻害する人数は、決して 1 桁超えた程度ではなさそうです。

アジャイル開発であれば、コミュニケーションのありようは下図の左から右へと変化し、結果的にコミュニケーションパスはかなり減少しそうです。



つまり、アジャイル開発はスケールしやすいといえそうです。

ではなぜ、メンバー数 1 桁が神話のように信じられているのでしょう。ひょっとすると、下図のように誤解されているのではないでしょうか？



ii. 複数のアジャイルチームで構成される大規模アジャイル開発

1 つのアジャイル開発チームがスケールされるとはいえ、さすがに 100 名を超えるメンバーの場合は直感的にも運営が困難になりそうです。

そこで、大規模開発に対応するためのアジャイルフレームワークがいくつか提唱されています。LeSS、Nexus™、Scrum@Scale™、SAFe®などといった大規模アジャイルフレームワークがそれです。単一チームでは成しえない、より大きな規模の開発に向き合うためのフレームワークです。

LeSS

LeSS（Large-Scale Scrum）は、著書「初めてのアジャイル開発」などで知られるクレーグ・ラーマンらによって提唱された大規模アジャイルフレームワークです。規模によって 2 種類の Less が用意されており、いずれも複数のスクラムチームによって構成されます。小規模は"Less（basic Less）"、大規模は"LeSS Huge"と呼ばれます。Less Huge は、8 チーム以上（1,000～2,000 名程度まで）の規模であるとされています。

Scrum@Scale™

Scrum@Scale™はスクラムの提唱者の 1 人であるジェフ・サザーランド氏によって提唱された、複数スクラムチームを一貫して機能させるための大規模アジャイルフレームワークです。Scrum@Scale®ガイドによれば、「（複数の）スクラムチームだけから構成されるために、理解は容易である」とされています。

Nexus™

Nexus™はスクラムの提唱者の 1 人であるケン・シュウェーバー氏によって提唱された、3～9 のスクラムチームと Nexus 統合チームによって構成される大規模アジャイルフレームワークです。

SAFe®

SAFe®は、ディーン・レフィンゲル氏によって提唱された大規模アジャイルフレームワークです。スクラムチームを開発組織の構成単位としており、PPM（Program Portfolio Management）が特徴的です。

スポティファイモデル

スポティファイとは、スポティファイ・テクノロジー社によって提供されている音楽配信サービスであり、その開発モデルがスポティファイモデルと呼ばれることがあります。開発プロセスのみならず、クロスファンクション、技術経営、組織学習など組織構造を含めた大規模アジャイルモデルです。

iii. 大規模アジャイルが解決すべき課題

大規模アジャイル開発が目指すものは、単一チームのアジャイル開発と同様であり、以下の特長

- より速いスピードでの提供 ～ 正味現在価値の視点
- 要求の変化に柔軟な対応が可能 ～ 変動対応性の視点
- 品質と顧客満足度の向上 ～ リスク早期検知の視点
- 開発者やチームの成長を促進 ～ 学習の視点

これらを実現することです。

これらの特長は、規模の経済からスピードの経済への変革、すなわち、変化を受入れるためのリードタイム短縮を目指すことによって得られるものです。

iv. 大規模アジャイルフレームワーク

多くの大規模アジャイルフレームワークの目指すものを端的に言ってしまえば、

> 複数のアジャイル開発チームの成果物を、いかにすばやくインテグレーションするか

つまり、大規模アジャイル開発においても、「唯一のコードベース」を保ちつつ、いかにリードタイムを短縮できるかということに尽きるのではないでしょうか。

個々のアジャイル開発チームによって十分に短縮されたリードタイムを、大規模アジャイルでも短いまま維持するための「常時結合（継続的統合）の戦略」が大規模アジャイル開発の意義だと考えられます。

v. コストに対するアプローチ

最初に全てが計画されているウォーターフォール開発では、品質の確保とともにコストダウンによる利益の確保が重要な課題です。価格が決定されているため、利益確保の方法がコストダウンに依らなければならないためです。

さて、大規模アジャイル開発はどのようにコストに対してアプローチするのでしょうか？アジャイル開発は、XP（eXtreme Programming：エクストリームプログラミング）のプラクティスである計画ゲームに代表されるように、**ビジネスとテクノロジの調和**を目指しています。

であれば、大規模アジャイル開発も、ウォーターフォール開発のようにコストに対するアプローチも必要ではないでしょうか。

しかし、アジャイル開発はウォーターフォール開発のようなコストダウン戦略を単純に取り込むことはできません。最初に全てが計画されている訳ではないからです。コストを統制すると同時に、アジャイル開発の特長である自律によるスピードも維持する。そんな戦略が必要そうです。

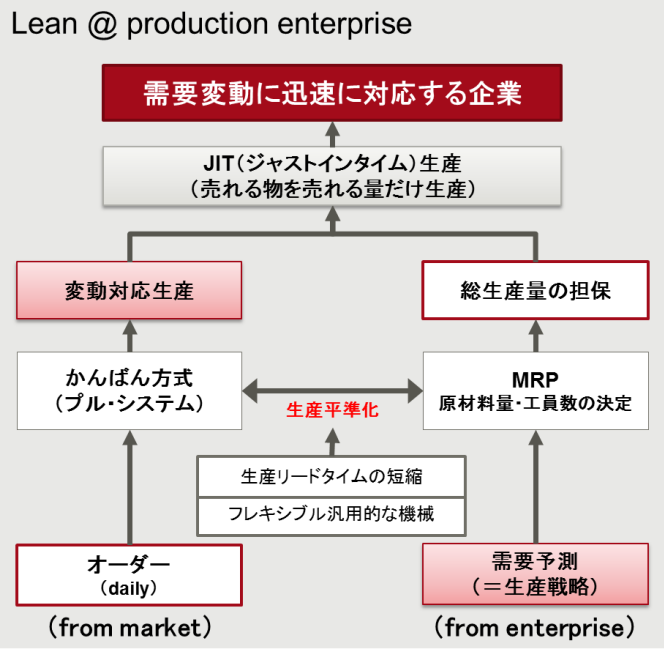
- 統制と自律の調和
- コストとスピードの調和

これは何かに似ています…

DX なエンタープライズアジャイルと JIT

門田安弘氏の著作「トヨタプロダクションシステム―その理論と体系」によれば、

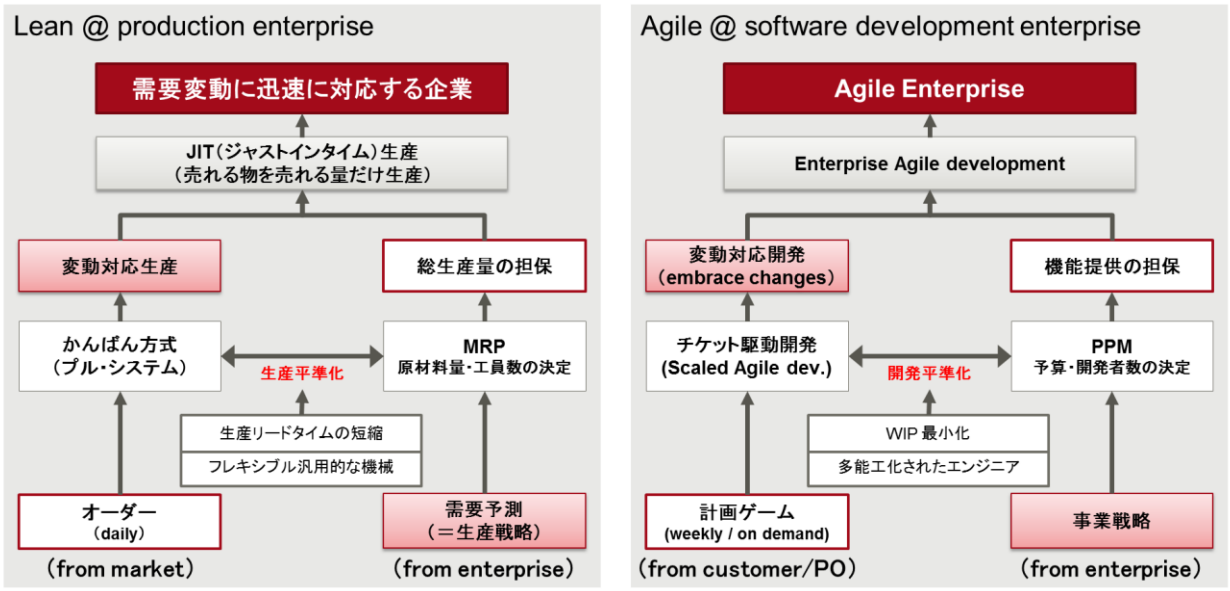
JIT（Just In Time）は、かんばん方式と MRP（Materials Requirements Planning：資材所要量計画）の調和によって成立されているようです。要約して下図に示します。



参考：門田安弘著、書籍『トヨタプロダクションシステム～その理論と体系』

変動対応生産と総生産量の担保のために、プルシステムであるかんばん方式と原材料量や工員数を決定する MRP が併存し、生産平準化を実現します。

大規模アジャイル開発、いわゆるエンタープライズアジャイルも類似性が認められるのではないでしょうか？



参考：門田安弘著、書籍『トヨタプロダクションシステム～その理論と体系』

変動対応開発と機能提供の担保のために、プルシステムであるチケット駆動開発と予算や開発者数を決定する PPM（Program Portfolio Management）が併存し、開発平準化を実現する。

これらによって、

- 統制と自律の調和
- コストとスピードの調和

これらの調和を図れそうです。

PPM に対応した大規模に対応したアジャイルフレームワークとしては、先に述べた SAFe®があります。一見、他のフレームワークよりも分がありそうです。

しかし、PPM に対応しているからといって安直に SAFe®を採用するにはリスクもありそうです。マネジメントのオーバーヘッドが、必要以上に大きくなることがあり得るからです。ウォーターフォール開発では、設計／実装／テストに関わることのないメンバー（非開発要員）が多くいます。大規模なウォーターフォール開発になれ親しんだ SIer が非開発要員を、SAFe®のマネジメント層に異動担務させることも大いに想定されます。

雇用対策的なこの異動担務が、大規模アジャイル開発を官僚的な組織にさせてしまえば、自律性は失われ開発者はただの歯車のようになり、重鈍な塊になるリスクがあることを忘れてはならないのではないでしょうか？

3.3 大規模はアジャイル開発の味方

i. コミュニケーションはリスク？

ここまで、人員規模の増加によるコミュニケーションの増加は、すなわちリスクの増加であると述べました。また、アジャイル開発では、動くソフトウェアをコミュニケーションの中心としていること、さらにはペアプログラミングとペアローテーションによって、このリスクの増加を小さくしていると述べました。

コミュニケーションの増加は、確かにリスクを増加させます。しかしそれだけでしょうか？

アジャイル開発方法論の 1 つであるエクストリームプログラミング（XP：eXtreme programming）では、

- コミュニケーション（Communication）
- シンプリシティ(Simplicity)
- フィードバック（Feedback）
- 勇気（Courage）
- リスペクト（Respect）

これらが 5 つの価値とされており、コミュニケーションはその筆頭に上げられています。であれば、コミュニケーションには、リスクを生んだり、リスクを増加させたりする以外に、なにか効能がありそうです。

ii. 質の高いコミュニケーションとは？

『質の高いコミュニケーションとはなんでしょう？』こう問うと、たいていの場合次のような答えが返ってきます。

- 正確であること、誤りなく伝えられること。
- 効率的であること、安価であること。
- すばやく伝わること、速度が速いこと。

どうやら情報伝達に関する QCD に注目しているようです。

Q（品質：正確であること、誤りなく伝えられること）については、「狩野モデル」（注釈 1）でいうところの『当たり前品質』を対象としているようですし、C（コスト：効率的であること、安価であること）や D（すばやく伝わること、速度が速いこと）については、『一元的品質』を対象にしているようです。

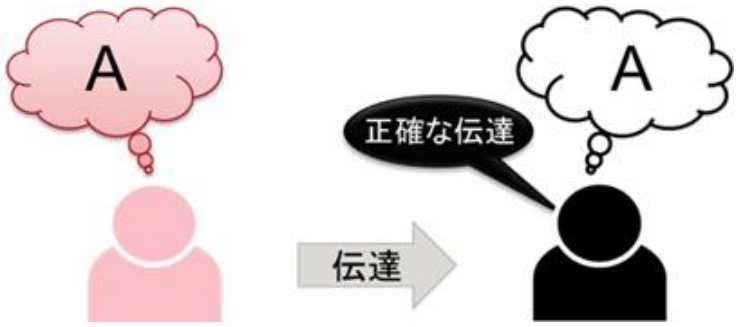
コミュニケーションの質は、『当たり前品質』や『一元的品質』にとどまるのでしょうか？コミュニケーションに、『魅力品質』はないのでしょうか？

【質って？】
たとえば、豆腐の質ってなんでしょう？

腐っていないこと？異物混入していないこと？　～　当たり前品質
美味しいこと？　～　魅力品質

じゃあ、コミュニケーションの質ってなんですか？

コミュニケーションの質が正確であることであれば、下図のように表されるでしょう。

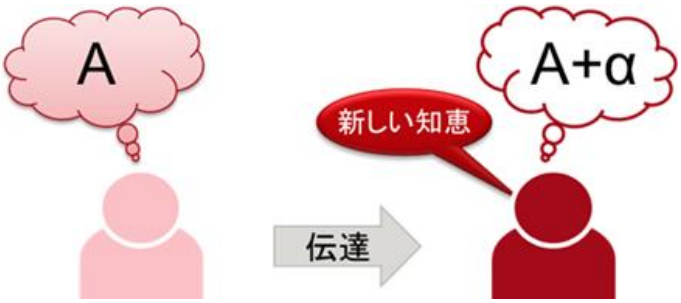


『間違いのないことが質の高いコミュニケーションである』もしそうなら（それだけであれば）エクストリームプログラミング（XP：eXtreme programming）の価値の筆頭に挙げられないように思われます。

コミュニケーションが発生するとき、一体なにが起きているのでしょうか？コミュニケーションの魅力品質とは、一体なんなのでしょうか？

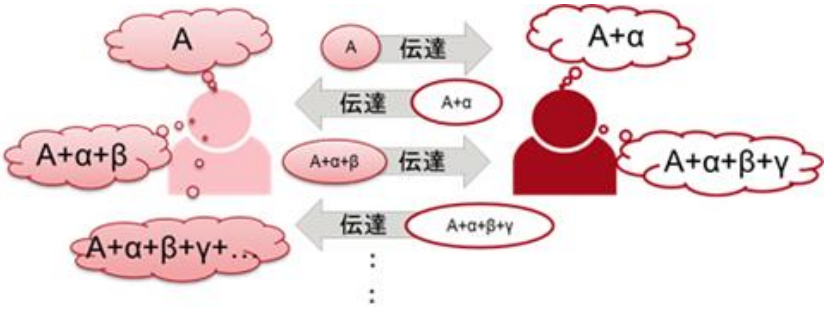
iii. 知恵の集積

コミュニケーションが発生するとき、受け手は伝達された情報を受け取るでしょう。受け取られた情報に、それまで自身の中に積み重ねてきた経験や知恵が反応し、新しい知恵が生まれるのではないでしょうか？

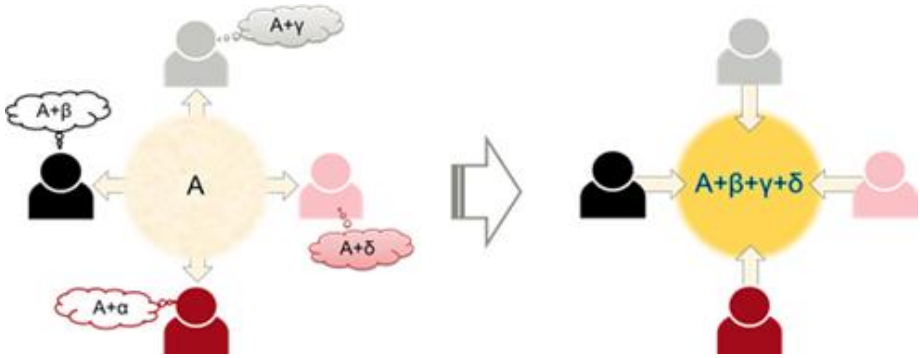


A を伝えられたら、受け手の中で A は A+αに増幅するということです。

これが双方向のコミュニケーションで発生するとどうなるでしょう。



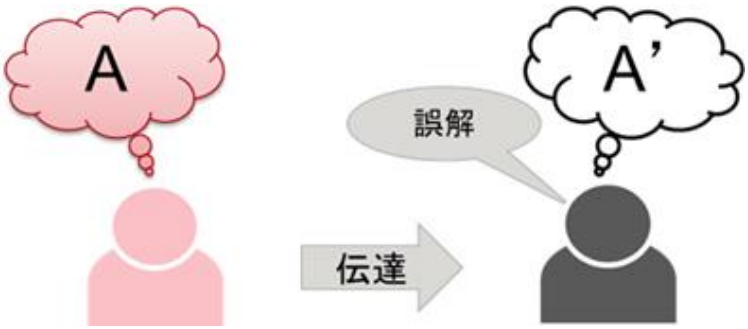
知恵を重ねる。重ねて返す。さらに重ねる。重なる。チームであれば、何人分もの知恵がさらに、一気に重なることでしょう。



知恵はコミュニケーションによって増幅、集積されるのです。たんなる既存の知恵の共有ではなく、集積され続けるのです。これこそが、コミュニケーションの魅力品質です。

iv. チーム／組織の進化

情報が正しく伝達されないこともあるでしょう。



受け手がそれまで自身の中に積み重ねてきた経験や知恵のために、受け手に正しく伝わらない場合があるかも知れない、エラーが発生するかもしれないということです。正確な伝達が価値であれば、それは損失でしかないでしょう。

しかし、この現象を生物の遺伝／進化になぞらえてみるとどうでしょうか。

- DNA／遺伝子の複製の誤りによって突然変異が発生する。
- 適さないものは淘汰され、適したものが生存しやすくなる。

伝達エラーは、より優れた知恵が生まれる機会でもあるということです。自然淘汰による適者生存は、すなわち組織能力の進化であり、競争優位性を獲得することに他ならないのではないでしょうか？

v. 規模はアジャイルの味方

コミュニケーションによって、『知恵の集積』や『チームの進化』を促すものは、おそらく規模と時間でしょう。

- 規模：人の数
- 時間：チームの寿命

大規模化によるコミュニケーションパス数の増加が、知恵の集積機会の増加につながり、チーム寿命の長期化が、進化（と淘汰）の機会増加につながる。と言えそうです

> 10 名を軽く超えていても上手くやっているアジャイル開発チームはよく見かけます。
> （上手くいっているチームほど 10 名を軽く超えている印象を受けます）

と、さきほど述べました。

上手くいっているチームは、コミュニケーションによって生まれる新たな知恵や、チームの進化を上手く利用していると思われます。

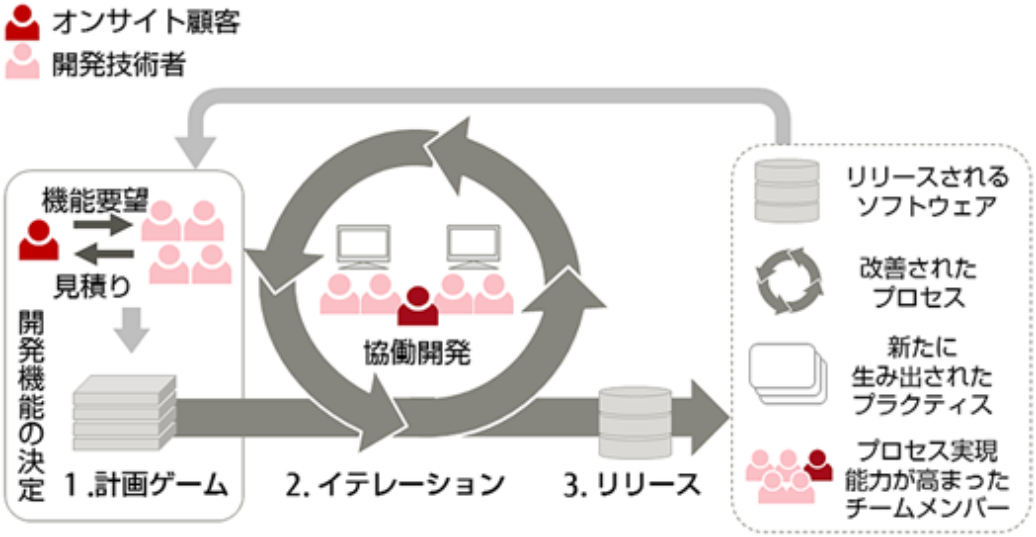
ウォーターフォール開発では、コミュニケーションを通じた知恵の創造を期待していません。予め決定された情報が正確に伝達されることだけを期待されています。そのため大規模化による要員の増加はコストやリスクにしかなりません。

アジャイル開発では、コミュニケーションパス（あるいはコミュニケーションの場）で、知恵の創造と集積を強く期待されています。

ペアプログラミング、ペアローテーションを実践しているアジャイル開発チームのコミュニケーションパス

（上図の青線および青枠がコミュニケーションの発生する場）

下図は、アジャイル開発プロセスの概要です。



アウトプットは、リリースされるソフトウェアだけではなく、

- 改善されたプロセス
- 新たに生み出されたプラクティス
- プロセスの実現能力が高まったチームメンバー

これらもアウトプットであると述べました。

プロセス（Process）もプラクティス（Practice）もチーム（People）も成長し続けます。成長に必要な知恵を創造し集積するための重要な要素が、アジャイル開発におけるコミュニケーションなのです。つまり、コミュニケーションの魅力品質を高めることで、アジャイル開発は規模を味方にできるのです。

規模はアジャイル開発の味方です。

第4章 アジャイル開発の原価管理

4.1 フロントローディング

ウォーターフォール開発が主流の時代にあっては、ソフトウェア開発の原価管理といえば、効率化、調達費用削減、手戻り防止などを主たる施策とするとともに、見積りの精度を高めることを狙ってきました。

しかしアジャイル開発が主流となるとともに、「アジャイル開発とは異なるプロセスであるウォーターフォール開発の原価管理方法をそのままアジャイル開発に適用できないのではないか？」という疑問も発生してきました。

ここではソフトウェア開発の原価管理について考察します。

i. ハードウェア製造業の原価管理

アジャイル開発、ウォーターフォール開発といったソフトウェア開発の原価管理の前に、まずはハードウェア製造業の原価管理についてみてみましょう。

原価管理の三本柱

原価管理というと、原価低減（＝コストダウン）をイメージすることが多いと思われますが、ハードウェア製造業では、以下の 3 つを原価管理の 3 本柱と呼ぶことがあります。

- 原価維持
- 原価改善
- 原価企画

原価維持

設定した目標原価を達成するための活動です。プロセスだけでなく、原材料費の維持も対象とし、その変動要因である購買ルートや為替の変動リスクへの対応もその範疇です。

原価改善

原価低減ともよばれます。改善によって過去の目標原価を上回るコストダウンを狙った活動のことです。原材料費だけではなく、労務費や流通コストなども対象とします。

原価企画

原価設計とも呼ばれます。企画設計段階いわば源流から「コスト」のみならず「納期」や「品質」もつくりこむ活動です。以下のような手法が代表的です。

- VE/VA ：VE（Value Engineering：価値工学）、VA（Value Analysis：価値分析）

  より安価に、同じ効果を発揮する原材料や方法を採用することで、同一機能に対するコストを削減する方法です。

- QFD ：QFD（Quality Function Deployment：品質機能展開）

  顧客要求と求められる品質特性との関連を明らかにし、品質特性を実現する機能、構造に展開することで製品仕様を設計する方法です。機能、構造ごとに目標原価を割り当てることで、顧客要求に基づいた品質と原価を企画設計段階でつくりこむことが可能になります。

- コンカレントエンジニアリング

  異なる複数の部門が同時期に活動することで、品質向上と開発期間短縮を実現する方法です。

  自動車開発であれば、スタイリング、開発部門、生産技術部門など異なる複数部門が同時に活動することで、例えば

  > スタイリングが空力性能に及ぼす影響を開発部門が検知する。
  > 成型加工時の寸法安定性上の問題を生産技術部門が検知する。

  というように、問題を早期に検知します。これによって、後々発生するはずだった修正コストを（その発生要因もあわせて）解消するとともに、開発期間の短縮が可能となります。

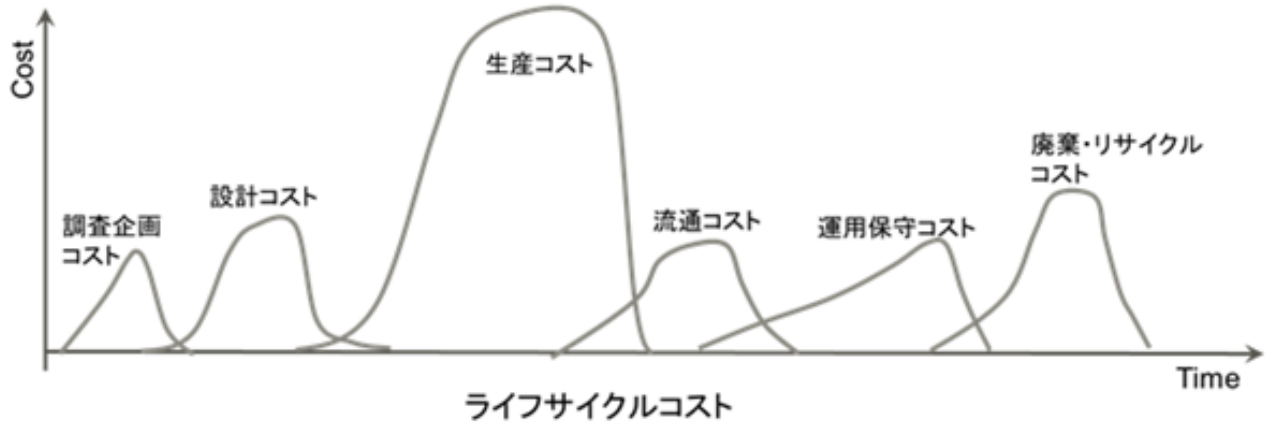
  原価維持や原価改善は、主に生産段階以降の原価管理を対象としていますが、原価企画では、製品のライフサイクル全般に関わる原価管理を対象としています。

  なぜなら、ある工程における原価の発生要因はその工程よりも前の工程で決定されるためです。

ライフサイクルコスト

ライフサイクルコストという考え方があります。

製品やサービスの提供にかかるコストすべて、つまり設計・生産だけではなく、企画段階から、流通、廃棄に至る製品ライフサイクルにかかる全てのコストのことです。

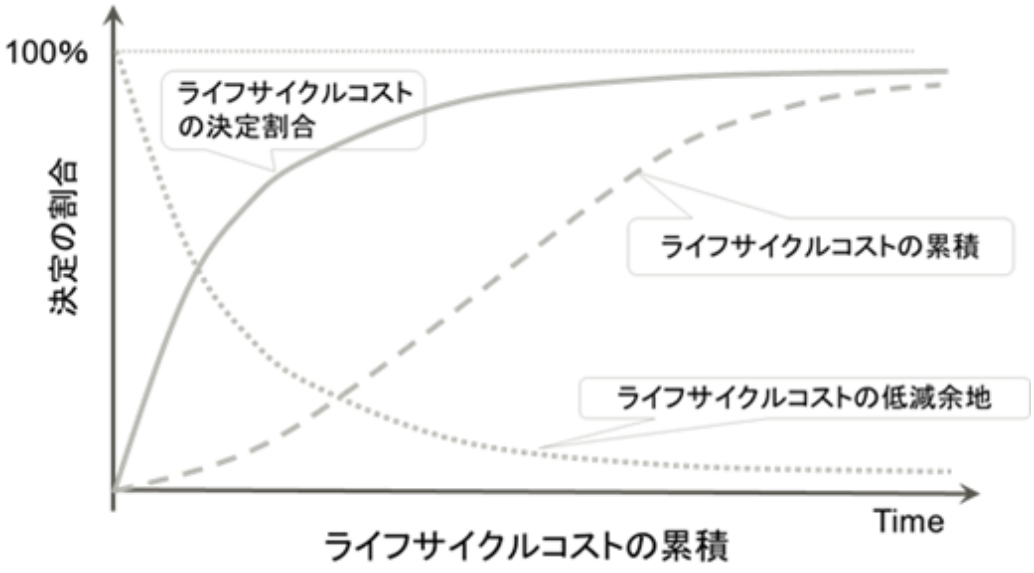


- つくりやすいのか？
- 運びやすいのか？
- 捨てやすいのか？

これらは、企画設計段階で既に決定されています。

原価の発生要因、つくる上で、運ぶ上で、捨てる上でなど製品ライフサイクル全般に渡ってコストを発生させる要因のことをコストドライバーと呼びます。

先ほど例に挙げた成型加工時の寸法安定性は、生産時の歩留まりを悪化させコストを上昇させます。寸法安定性は設計段階でつくりこまれるコストドライバーであり、生産工程でより多くのコストを発生させる要因です。



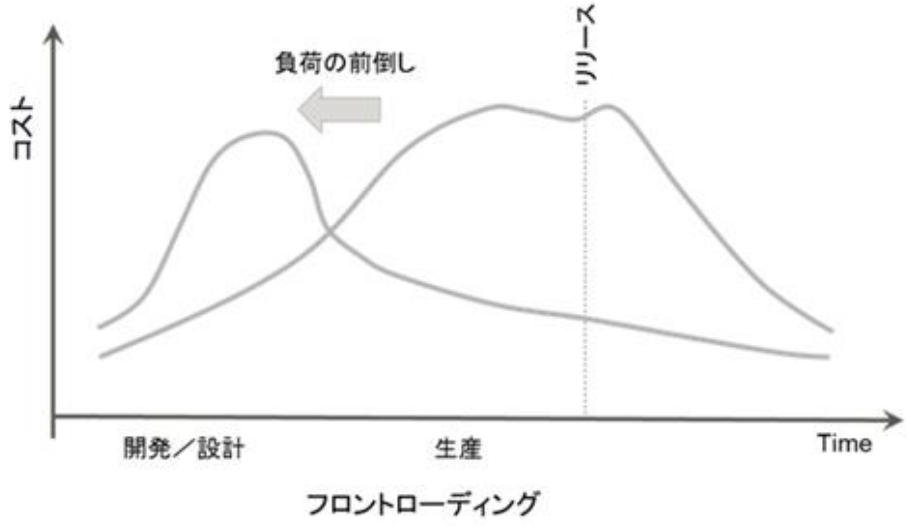
原価企画はライフサイクルコストの低減余地が大きい（＝ライフサイクルコストが未決定な）早期の段階で、品質をつくりこむと同時にコストドライバーを見極め対策することで、ライフサイクルコストの最小化を狙います。

フロントローディング

フロントローディングとは文字どおり負荷の前倒しのことです。

原価企画では、製品の企画開発段階で不具合やコストドライバーを検知し、削減抑制することで品質をつくりこみ、ライフサイクルコストを下げ、製品リリースの時間を短縮することを狙います。ライフサイクルコストの低減余地が大きい、すなわちライフサイクルコストの決定がまだあまりなされていない時期での活動です。

製品ライフサイクルの初期段階つまりフロントに負荷がかかることから、フロントローディングと呼ばれます。



フロントローディング

ライフサイクルコストの決定要因であるコストドライバーは、製品の企画開発段階でその大部分が決定されます。各技術や各工程を考慮し、多面的な視点に基づいたコンカレントエンジニアリングが必要であり、コンカレントエンジニアリングをより効果的に機能させるために、クロスファンクショナル（機能横断）チームやマトリクス組織が必要となります。そして最も重要なことは実際に試すということです。試作を重ね、多面的な視点に基づいて各分野の専門技術者たちが、同時に実際に本物で検証するということです。

もっとも、量産開始までの時間制限もあれば、ハードウェアであるため実際に試作するコスト（例えば金型製作にかかる多大な費用）の制限もあります。そのため効率的な検証のためにシミュレーションによる疑似的な検証も実施します。

フロントローディングは手法と考えられがちですが、現象と考えた方が妥当かもしれません。負荷の前倒し自体は手法ではなく、品質のつくりこみやコストドライバーの削減に必要な原価企画の活動によって負荷がフロントに移動する現象であるためです。

ii. ソフトウェア開発の原価管理

さて、ソフトウェア開発ではどのような原価管理を実施しているのでしょうか。

最初に全てが計画されているウォーターフォール開発では、原価改善（原価低減）が重要な課題です。とくに作成請負契約によって受託した SI ベンダーにとっては、売上価格が決定されているため、原価低減すなわちコストダウンによってのみ利益の確保が可能だからです。

以下の 3 つは、おもだったコストダウンの施策です。

- 効率化 ～ プロセスやツール
- 調達費用削減 ～ 外部調達、単能工の利用
- 手戻り防止 ～ レビュー強化、チェックリストの利用

効率化

プロセスやツールによる効率化が代表的でしょう。

プロセスの標準化／横展開、統合開発環境の利用やフレームワークの利用、インテグレーション／テストの自動化などによる効率化が代表的です。

調達費用削減

工程ごとの専門技術者（いわゆる単能工）が担当することで、教育コストを低減することが可能です。また、廉価な外部調達（外製化や派遣技術者の利用）によるコストダウンも代表的な施策です。

手戻り防止

工程ごとのレビュー強化や、レビューの際にチェックリストの利用による手戻り防止が代表的な施策です。

上記の内、「調達費用削減」と「手戻り防止」の 2 つは、工程ごとにドキュメントを作成し次工程への入力とするウォーターフォール開発ならではのコストダウン施策です。

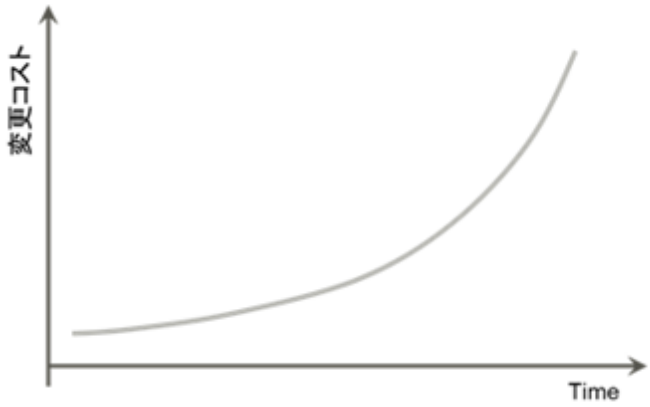
手戻り防止

アラン・M・デイビスの著作『ソフトウェア開発 201 の鉄則』によれば、

> もし、要求仕様書に誤りがあれば、見つけるのが後になればなるほどとんでもなく高くつく。
> もし要求仕様書の誤りが設計まで残っていたら、それを見つけて修正するのに５倍のコストがかかる。
> もしコーディングまで残っていたら、１０倍かかる。
> もしテスティングまで残っていたら、２０倍かかる。
> もし納入時点まで残っていたら、２００倍かかる。

と、あります。

上流でつくり込まれた不具合は、後工程になればなるほど修正・変更コストが急激に上昇するということです。



ウォーターフォール開発のフロントローディング

ウォーターフォール開発では、『フロントローディングによって手戻りを防止する』との主張をよくみかけます。

- 入念にレビューする。
- 静的解析を強化する。
- 単体テストをしっかりやる。

などの方法です。

最初に全てを計画し、計画どおり直線的に工程を進めるウォーターフォール開発にとっては、確かにこれらの方法である程度の手戻りを防ぐことはできるでしょう。しかしフロントローディングというには、大切なものが欠けていそうです。そう、最も重要な、実際に試すということができないのです。

実際に試すことができるのは、インテグレーションしたその後です。ハードウェア、OS、ネットワーク、データベース等のミドルウェア、外製パッケージソフトウェア、内製のソフトウェアコンポーネントなど、システムの構成要素すべてを統合し「本物を対象とした」End to End なテストでなければ品質をみることはできません。

構成要素それぞれについて、品質、機能、性能などを検証されていたとしても、実際に統合してみるまでは実際に試していることにはなりません。相互作用やボトルネックも判明しなければ、コストドライバーの特定も叶わないのです。

ウォーターフォール開発では原価企画的な活動は難しそうです。原価維持や原価改善を中心とした原価管理が限界であり、現実的であるといえそうです。

4.2 工程とコストドライバー

ここからは、アジャイル開発の原価管理について探っていきましょう。特に、アジャイル開発において原価企画的な発想は可能なのか、フロントローディングは機能するのか、これらを中心に考察します。

i. それぞれの工程

原価、特にライフサイクルコストを考える上で工程の概念が欠かせません。ソフトウェア開発の工程、特にウォーターフォール開発では、

- 企画
- 設計
- プログラミング（製造）
- テスト

といった、作業の順番や段階を示すことが多いでしょう。工程から工程へと順次アウトプットが受け渡されていく段取りです。

しかしこれらの段取りをソフトウェア開発上の工程とするには、少々疑問があります。

ある工程が完了しそのアウトプットを、次の工程にインプットする。もし、その工程が完了しているのであれば、アウトプットの品質は保証されていなければなりません。つまり、ある工程におけるアウトプットの品質が検証できなければならないということです。検証可能でなければ工程ではないといえるでしょう。

検証可能ということは使って試せるということです。使って試せなければ工程とはいい難いのではないでしょうか。

例えば「設計」。設計を終えた段階ではその設計が正しいことを試す術がありません。ソフトウェアはインテグレーションを実施したその後でなければ本物で試すことが出来ないのです。実際に使って試せなければ検証可能とは言えません。

アウトプットつまり次の工程へのインプットが検証できないのですから、工程とはいい難いのです。

そのため、企画、設計、プログラミング、テストなどを、ソフトウェア開発の工程と考えるのは少々無理がありそうです。ハードウェアの開発工程は、試作品を文字どおり「作り」そして「試して」いますから工程であるといえるでしょう。

ハードウェアの工程

開発と製造が分離

ハードウェアは開発工程と製造工程が分離されています。開発工程では試作品と製品設計情報が、製造工程では使用に供するための製品が、それぞれアウトプットされます。

開発と製造が分離されたハードウェアにおいては、

- 製造時に発生するコストがライフサイクルコストを支配する。
- 製造時に発生するコストドライバーは設計段階においてつくり込まれる。

という特質を利用してフロントローディングを実現しています。

開発段階で既にコストドライバーがつくりこまれているため、製造工程では原価改善は微々たる効果しかあげない。それゆえ企画設計段階のいわば源流から「コスト」のみならず「納期」や「品質」もつくりこむ活動がフロントローディングです。

また、ハードウェアでは開発と製造が分離されているからこそコンカレントエンジニアリングやクロスファンクショナルチームによって、（さっさと試すための）フロントローディングを実現したといえるでしょう。

ハードウェアは、開発と製造が分離されていることをうまく利用しています。

ソフトウェアの工程

設計と製造が同時進行

ハードウェアの場合には開発と製造が分離されています。一方、ソフトウェア開発、特にアジャイル開発では設計と製造（実装）が同時に進行しています。

一般に製造と呼ばれるプログラミングは、詳細な論理構造を明確にする設計行為でもあり、既に決定された設計情報に基づいただけの単なる製造とはいえません。 つまりソフトウェアでは設計工程と製造工程が同時一体として進行しているのです。決して分離してはいません。

ソフトウェア開発の工程とは実際に試せる単位、検証可能な単位でなくてはならず、それは顧客にとっての価値が発生する「1 機能のつくりこみ（設計/コード/テストの一連の作業のまとまり）の単位」、「反復の単位」あるいは供用の単位である「リリース」である。と考えるのが妥当ではないでしょうか。

ソフトハードの違い

ソフトウェアとハードウェアでは、特徴にいくつかの違いがあります。

	ソフトウェア	ハードウェア
工程	設計と製造が同時一体として進行しています。	開発と製造が分離されています。
テスト	本物でテストが出来ます。破壊試験すら本物で全品テスト可能です。	全品破壊試験することはできません。
変更容易性（コスト）	ハードウェアに較べて（ソフトですから）変更が容易（低コスト）です。 （容易かつ低コストでなければソフトウェアで実現する必要はありません）	ソフトウェアに較べて変更が（高コスト）困難です。 （変更に高いコストが必要なものの中では金型が代表的です）

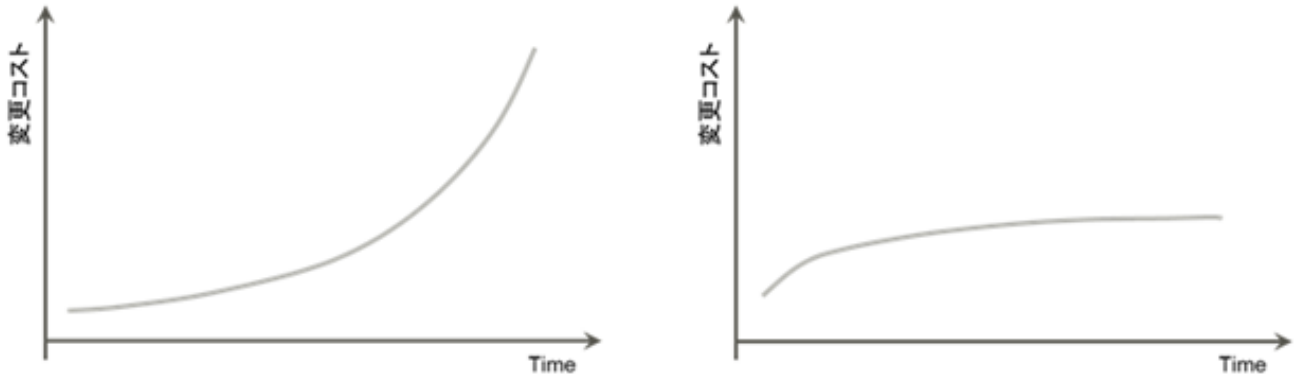
この差を越えて、あるいはこの差を利用して、ハードウェアにおける原価管理の知見をソフトウェア開発へと持ち込むことが必要でしょう。

ii. ソフトウェア開発のコスト

ソフトウェア開発のコストについて、アラン・M・デイビスは、自身著作『ソフトウェア開発 201 の鉄則』でこう述べています。

もし、要求仕様書に誤りがあれば、見つけるのが後になればなるほどとんでもなく高くつく。
もし要求仕様書の誤りが設計まで残っていたら、それを見つけて修正するのに５倍のコストがかかる。
もしコーディングまで残っていたら、１０倍かかる。
もしテスティングまで残っていたら、２０倍かかる。
もし納入時点まで残っていたら、２００倍かかる。

次の左側の図のように、時間が経つにつれて右肩上がりになるということです



左側の図のようにならないよう、時間が経過しても平らに（上図の右側のように）する。つまり、時間の経過にかかわらず変更コストを一定にする作戦をとれれば、アラン・M・デイビスの主張する現象を回避することができるでしょう。

iii. アジャイル開発の工程

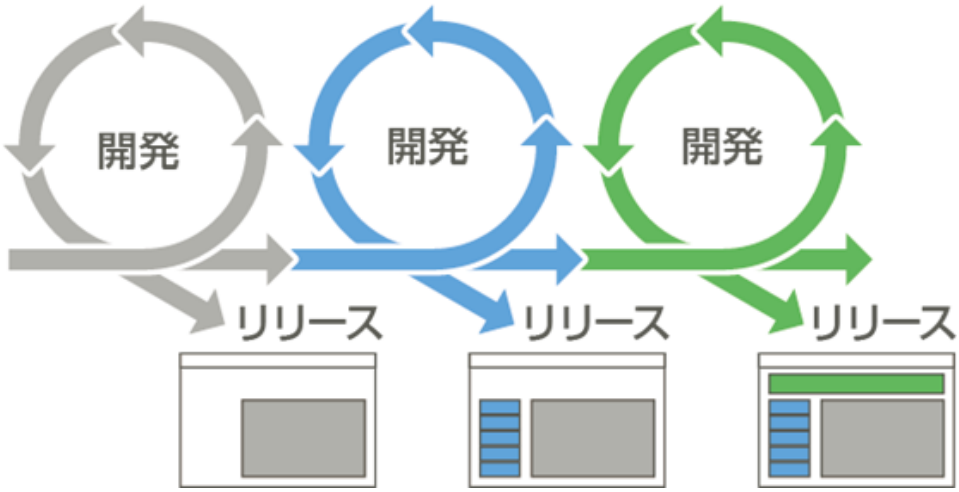
ハードウェアの場合、開発工程の後工程は製造工程でした。では、アジャイル開発の場合はどうでしょうか？

アジャイル開発は、顧客が利用可能な単位である「ストーリー」ごとに設計、コード、テストを実施する増加的（インクリメンタル）な開発です。また、1～2 週間のイテレーション（またはスプリント）と呼ばれる期間を 1 単位として反復的（イテレーティブ）な性質もあります。さらには、1～複数のイテレーションごとに実際に顧客へのリリースを実施します。

> ソフトウェア開発の工程とは実際に試せる単位、検証可能な単位でなくてはならず、それは顧客にとっての価値が発生する「1 機能単位のつくりこみ手順（設計/コード/テストの一連の手順）」、あるいは供用の単位である「リリース」である。と考えるのが妥当ではないでしょうか。

と述べました。

であれば、アジャイル開発の場合、開発工程の後工程は開発工程である。といえそうです。



- ハードウェア：開発工程の後工程は製造工程
- アジャイル：開発工程の後工程は開発工程

といえるでしょう。

アジャイル開発のフロントローディング

アジャイル開発の工程をストーリー、イテレーションあるいはリリースなど機能単位で完結すると捉えることで、フロントローディングの考え方が応用できそうです。つまり工程のより上流で、不具合やコストドライバーを検知し、削減抑制することで品質をつくりこみ、ライフサイクルコストを下げ、製品リリースの時間を短縮することを狙えそうだということです。

それには、アジャイル開発のコストドライバーを明確にする必要があります。

iv. アジャイル開発のコストドライバー

コストドライバーとはコストの発生要因のことです。さらに、アジャイル開発では、開発工程の後工程は開発工程でした。開発工程の前工程も開発工程です。「前工程である開発工程」がつくりこむ「後工程である開発工程」におけるコストドライバーは以下の 2 点であるといえるでしょう。

- 改修の難しさ
- 試験の難しさ

開発工程の後工程は開発工程。つまり改修に改修を重ね、改修のたびに試験を繰り返すアジャイル開発では、改修そのものの難しさ（容易さ）と試験の難しさ（容易さ）がもっとも大きなコストです。

しかも改修の対象は（不具合・故障の改修であっても、機能の追加・更新であっても）予想できません。事前にすべてを予見し計画することは不可能なのです。

品質特性

ISO/IEC 25010 では、ソフトウェアの品質特性をいくつかに分類しています。そのうち「保守性」はソフトウェアの改修・機能更新を有効に効率的にできる度合いであるとされており、保守性もさらにいくつかの副特性に分類されています。上に述べた「改修の難しさ』や「試験の難しさ」は、保守性の副特性である「修正性」や「試験性」に該当するでしょう。 修正性とは品質を低下させることなく安定的に修正できる度合いのことであり、試験性とは OK/NG が判定容易な試験を有効的に効率的に実行できる度合いのことです。

表：品質特性～システム／ソフトウェア製品品質

品質特性	機能的合成	性能効率性	互換性	使用性	信頼性	セキュリティ	保守性	移植性
副特性	機能安全性	時間効率性	共存性	適切度認識性	成熟性	機密性	モジュール性	移植性
	機能正確性	資源効率性	相互運用性	習得性	可用性	インテグリティ	再利用性	適応性
	機能適切性	容量満足性		運用操作性	障害許容性（耐故障性）	否認防止性	解析性	設置性
				ユーザエラー防止性	回復性	責任追跡性	修正性	置換性
				ユーザインタフェイス快美性		真正性	試験性	
				アクセシビリティ				

（日本工業規格 JIS X25010：2013(ISO/IEC 25010：2011)を参照して作成）

また、修正性や試験性とともに「解析性」や「モジュール性」も保守性の副特性として挙げられています。解析性とは欠陥故障の原因のわかりやすさや改修箇所の特定のしやすさの度合いであり、モジュール性とはソフトウェアを構成する要素が他の要素に影響を与えないように分離されている度合いのことです。よって、解析性やモジュール性は、修正性や試験性に大きな影響を与える要因であるといってよいでしょう。

- 修正性や試験性は、解析性やモジュール性およびソフトウェア構造に大きく影響される。
- 解析性やモジュール性もソフトウェア構造に大きく影響される。

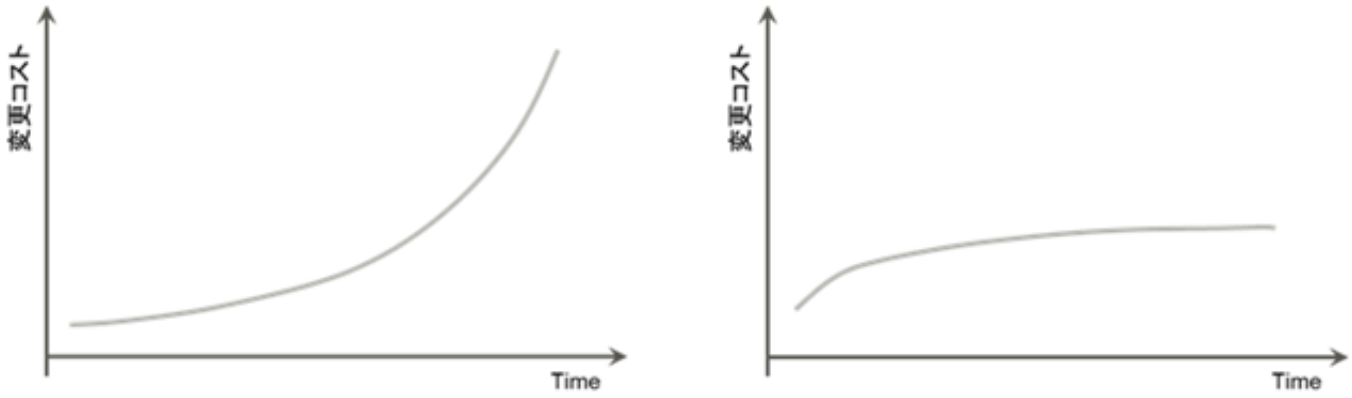
また、解析性やモジュール性はソフトウェアの構造に起因する特性であり、修正性や試験性も同じくソフトウェアの構造に起因する特性であるといえます。

EoC と EoT

コストドライバーとして挙げた「改修の難しさ」や「試験の難しさ」は、修正性や試験性のネガティブ表現であり、それぞれ「EoC（Ease Of Changing：修正容易性）」、「EoT：（Ease Of Testing）試験容易性」と呼ばれることがあります。

アジャイル開発においては「改修の難しさ」や「試験の難しさ」がコストドライバーであり、EoC と EoT を高めることが、アジャイル開発における原価管理の要諦であり、フロントローディングであるといえるでしょう。

もっとも、フロントローディング（負荷の前倒し）とはいえ、アジャイル開発では初期に負荷が偏るのではなく、下図の右側のように平坦な負荷状態になります。



とはいえ、アラン・M・デイビスの主張する「後期に負荷が偏る現象（上図の左側）」と比較すると、相対的にはフロントローディングされています。

4.3 品質とコスト

ここまで、アジャイル開発においては「改修の難しさ」や「試験の難しさ」がコストドライバーであり、EoC と EoT を高めることが、アジャイル開発における原価管理の要諦であり、フロントローディングであると述べました。

ここでは、アジャイル開発ではどのように EoC や EoT を維持向上させているのかをみていきます。

i. コストドライバーの制圧

ウォーターフォール開発におけるコストドライバーの制圧方針は、「計画どおりにやる」、「間違えないようにやる」ことでした。コストドライバーは設計時に決定済みであるため、分業しやすい設計や経験済みのノウハウ（フレームワーク、チェックリストなど）の横展開、あるいは専業分業体制によって、コストドライバーを制圧してきたのです。

これに対してアジャイル開発では、開発工程の後工程が開発工程であるため、コストドライバーを常につくり込み続けるリスクがありました。このため、コストドライバーそのものを最小化する作戦をとる必要があります。変更しやすい（EoC）設計、試験しやすい（EoT）設計でコストドライバーを制圧する必要があるということです。

それはすなわち EoC と EoT、ISO では修正性と試験性、そして解析性とモジュール性を向上させるように、ソフトウェア構造を維持改善し続けるということです。

アジャイルプラクティスで制圧

アジャイル開発では、コストドライバーをいくつかのアジャイルプラクティスで制圧しています。

アジャイルプラクティス、具体的には XP（eXtreme Programming：エクストリームプログラミング）のいくつかのプラクティスです。XP のプラクティスのうち、『開発のプラクティス』と呼ばれるプラクティスを見てみましょう。

ペアプログラミング

1 台の PC を使い、2 人で一緒にプログラミングします。

実装だけではなく、調査、設計、実装、テストなどの開発作業をペアで一緒に実行します。問題の発見、解決策の実行が即座に実施されるため、機能提供の時間が短縮されるとともに品質が向上します。テストプログラムも同時に作成実行するため、よりテストしやすく、わかりやすい実装と設計が 2 人の視点で磨かれていきます。

リファクタリング

外部から見た動作を変えずに、ソフトウェア構造を改善します。自動化されたテスト（あるいは、手動で実施すべきテストが明らかになっていること）が必須です。テストとは仕様そのものであり、仕様を変化させずに、わかりやすく、テストしやすく、他の構成要素に影響を与えない構造へと継続的に改善させます。

テスト駆動開発

まずテストプログラム（＝仕様）をつくってから実装する開発です仕様とは単体レベルから機能レベルまでの仕様を含みます。つくるものを明確に絞って順に実装し、デグレードなく機能と品質を積み重ねます。そしてその過程にはリファクタリングが必須であり、ソフトウェアの設計を継続的に進化させます。

継続的インテグレーション：CI（Continues Integration：常時結合）

開発中のソフトウェアを自動的にかつ即座にビルド、テストします。「いつでも出荷できる状態」を継続的に保つことが目的です。「統合、テストされていない時間」を最小化することで問題の発生をすぐに検知し意図せぬ不具合の混入を最小化します。

YAGNI

そんなものは必要にならない。You Are not Going to Need It. あるいは、 You Ain't Gonna Need It.の略です。いま要るものだけを必要最小限に実装します。要るかもしれない（というだけ）ものをつくりこめば、設計や実装の複雑度は増し、テストも必要となり、同時着手の仕事（WIP：Work In Progress）も増え、統合のリスクも増します。またリードタイムも長くなるため、不具合の検知も遅れ改修のコストが増加します。 YAGNI はこれら設計や実装の複雑度、統合のリスク、回収コストのリスクを最小化します。

このように、これらのプラクティスはソフトウェアの保守性を成す修正性（修正容易性、EoC：Ease Of Changing）や試験性（試験容易性、EoT：Ease Of Testing）を直接的に高めるとともに、修正性や試験性に寄与する解析性やモジュール性も高めています。

表：品質特性～システム／ソフトウェア製品品質

品質特性	機能的合成	性能効率性	互換性	使用性	信頼性	セキュリティ	保守性	移植性
副特性	機能安全性	時間効率性	共存性	適切度認識性	成熟性	機密性	モジュール性	移植性
	機能正確性	資源効率性	相互運用性	習得性	可用性	インテグリティ	再利用性	適応性
	機能適切性	容量満足性		運用操作性	障害許容性（耐故障性）	否認防止性	解析性	設置性
				ユーザエラー防止性	回復性	責任追跡性	修正性	置換性
				ユーザインタフェイス快美性		真正性	試験性	
				アクセシビリティ				

（日本工業規格 JIS X25010：2013(ISO/IEC 25010：2011)を参照して作成）

XP のプラクティスはコストドライバーを制圧しているといってよいでしょう。

マネジメントプラクティス

ここまで見てきた XP のプラクティスは、技術的プラクティスであるといわれているようです。しかしそうでしょうか？

図：Planning and feedback loops in extreme programming
（Wikipedia "Extreme programming "https://en.wikipedia.org/wiki/Extreme_programming より引用）

上図を眺めてみると、そうではないように思われてなりません。Planning と Feedback のループ、しかも大小さまざまな相似的なループはマネジメントサイクルを表していると判断せざるを得ません。

- XP のプラクティスはマネジメントプラクティスである。
- ただし技術がなければ実践できないマネジメントプラクティスである。

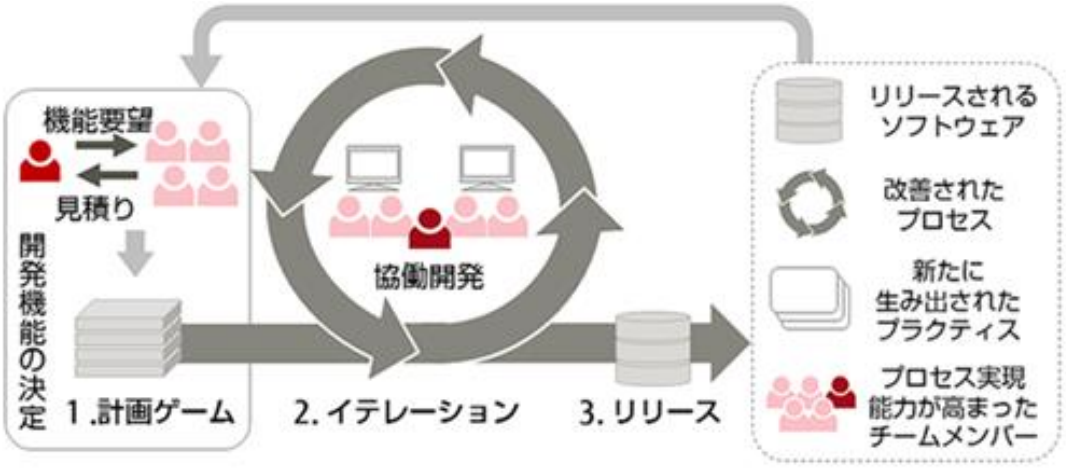
と、いえるでしょう。

しかも秒単位から月単位までの相似ループは、品質、コスト、デリバリー、スコープを同時にマネジメントするサイクルであると見てとれます。コストだけを切り離してマネジメントしている訳ではありません。アジャイル開発（特に XP）では、WIP を絞ってデリバリーを短縮し、インクリメンタル＆イテレーティブなプロセスによって品質を積み上げると同時に、コスト（コストドライバー）を制圧する作戦をとっているといえます。

チームの能力と開発者のスキル

ここまで、アジャイル開発ではプラクティスによってコストドライバーを制圧すると述べました。しかし、プラクティスは制圧の機会を確保しているにすぎません。機会を利用して EoC や EoT を向上させているのは開発者であり、開発チームに他なりません。

次の図に、アジャイル開発プロセスの概要を示します



開発プロセスのアウトプットは、ソフトウェアだけではなく「改善されたプロセス」、「新たに生み出されたプラクティス」そして能力が高まったチームメンバー（開発者たち）であることを示しています。この図に示したイテレーションサイクルだけではなく、図 『Planning and feedback loops in extreme programming』で示した（秒単位から月単位まで）大小さまざまなサイクルの中で、個人の知見や知識が伝搬され、開発技術者のスキル、チームの開発能力が向上します。コストドライバーの制圧だけではなく、制圧能力自体も高めているのです。

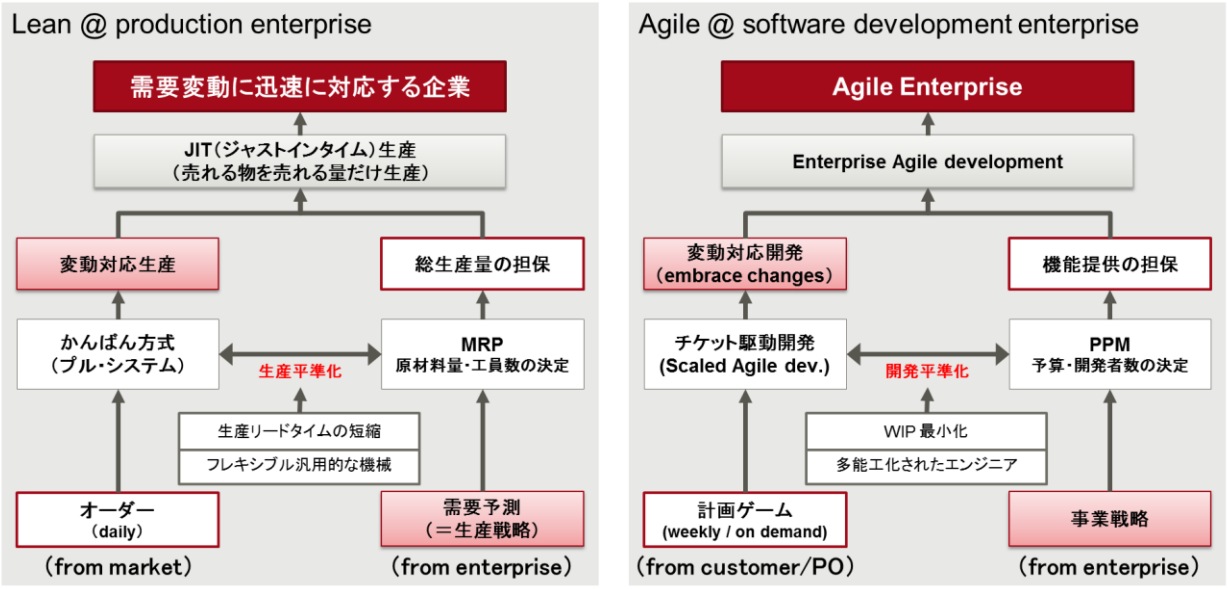
ii. エンタープライズアジャイルの原価管理

3.2 「アジャイル開発は大規模に耐えられるのか？」で、

> 変動対応開発と機能提供の担保のために、プルシステムであるチケット駆動開発と予算や開発者数を決定する PPM（Program Portfolio Management）が併存し、開発平準化を実現する。これによって「統制と自律の調和」、「コストとスピードの調和」を図れる

と述べました。「統制と自律の調和」とは「トップダウンとボトムアップの調和」であり、「コストとスピードの調和」は、「規模とスピードの調和」です。これはつまり、ウォーターフォールとアジャイルの調和に他なりません。

「統制と自律の調和」、「コストとスピードの調和」という課題をハードウェア製造業では既に気づき解決していました。JIT（Just In Time）です。



参考：門田安弘著、書籍『トヨタプロダクションシステム～その理論と体系』

アジャイル開発にあっても、エンタープライズアジャイルの進展による大規模化が進めば「なにをどれだけつくるか？」の変動がコストドライバーになります。これを制圧するための解決策が PPM になり得そうです。

iii. 最後に

ウォーターフォール開発に慣れ親しんだ方々の中には、「アジャイル開発では、リファクタリングやペアプログラミングなど、コスト増を招くだけで機能性を高めることもない習慣を推奨している」と、眉をひそめる方もまだまだいらっしゃいます。原価低減による利益確保を習慣としてきた方々にとっては当然の発想なのかもしれません。

確かに、リファクタリングやペアプログラミング、それにテスト駆動開発も付加価値や機能性を高めることは無いでしょう。しかし、これらはコストのみを高める活動ではなく、コストドライバーを制圧する活動です。制圧能力を高める活動なのです。

実は、ハードウェア製造業にもリファクタリングに似た活動があります。

5S と呼ばれる活動がそれです。製造工程おもに生産現場で活動するのが 5S（整理：Seiri、整頓：Seiton、清掃：Seisou、清潔：Seiketsu、躾：Shitsuke）です。要るものと要らないものを分けて要らないものを捨てる、あるべき場所にあるべきものを格納する、いつでも使えるようにする、誰が見てもわかりやすい状態を維持する、ルールや規律を習慣づける。これらの活動のことです。

製造業の方々は概ね 5S の活動を大切にされています。これらの活動はそこで働く人々の意欲や能力を高め、原価低減に寄与します。しかし、既に開発工程で決定済みのコストドライバーを製造工程で制圧することは叶いません。

原価企画やフロントローディングなど、ハードウェア製造業によって培われた知見を持ち込むにあたり、開発工程でつくり込まれ得るコストドライバーをアジャイルプラクティスによって制圧することは、『ソフトウェアであること』そして『ソフトウェアは開発と製造が同時進行すること』をうまく利用した原価管理であるといえるでしょう。

第5章 アジャイル開発の進捗管理

5.1 バーンダウンチャート

ウォーターフォール開発とアジャイル開発とでは工程の捉え方が大きく異なります。異なるプロセス、異なる工程。であれば、アジャイル開発では異なる進捗管理が必要になりそうです。

i. 進捗管理とは

進捗管理。ソフトウェア開発では日常的によく使われることばです。その定義を調べてみようと Web 検索してみました。しかし、「進捗管理とは○○である」といった定義を見つけることができませんでした。どのように進捗管理を実施するのか？つまり How についての記述は数多くありましたが、What や Why にあたる定義がなかなか見つかりません。

いくつかみた Web 検索結果をあわせてみると、

> 進捗管理とは、計画をまもるために、予定と実績とのズレや進み遅れを管理すること。遅れたらリスケジューリングして遅れの原因に対処すること

と、判断できます。であれば、

- Why：計画をまもるため
- What：予定と実績のズレや進み遅れを管理すること
- How：遅れたらリスケジューリングして遅れの原因に対処すること

と、いえるでしょう。

進捗率

進捗は進捗率（単位はパーセント）でみることが多く、分母と分子は、それぞれ「全体の量」と「終わった量」で示します。

> 進捗率（%）＝終わった量／全体の量

全体に対する完了の度合いが進捗率です。全体の量と完了した量がわかれば、進捗率は出せるということです。

全体の量と完了した量

アジャイル開発では、開発する全体の量は決まっていません。しかしイテレーションあるいはスプリントと呼ばれる反復期間でどれだけつくるか、つまり反復期間内の開発量は、反復の冒頭に計画ゲーム（あるいはスプリントプランニング）と呼ばれるミーティングで計画します。

ii. 計画

開発者全員を含む関係者によるミーティング（計画ゲームあるいはスプリントプランニング）で、反復期間内につくる機能の量や優先順位を計画します。

- 反復期間：開発期間を通じて固定
- 個々の機能の量：それぞれの機能をつくるのにどれくらいかかりそうか？
- 受容能力（Capacity）：全部でどれくらいつくれそうか？
- 優先順位：どういう順番でつくるか？

反復期間

反復期間（イテレーションあるいはスプリント）の長さは、概ね 2 週間以下の期間とすることが多いです。根拠としては、

- 計画が容易：なんとか想像できるのは、向こう 2 週間までの分までがやっと
- ふりかえりが容易：なんとか思い出せるのは、2 週間以内に起きたことがやっと
- リスクを最小化：変動に対するリスク、検証・品質に対するリスクを小さくできる

これらが上げられます。しかし「ハイリスクである」、「頻繁なリリースが必要である」あるいは「開発プロセスをより頻繁に調整する必要がある」など、より短いサイクルが必要な場合には、2 週間未満（例えば 1 週間）とすることもあります。

また、反復期間を固定にするのは以下の特質によるものです。

- 反復増加型のアジャイル開発では機能ごとに品質検証の完了が必要である
- 反復期間内に費やすコストは反復期間の長さに依存する

これらの特質から、反復期間（デリバリータイム）を固定にすることで、品質とコストとデリバリー（QCD）を一定にすることが可能となります。

個々の機能の量と受容能力

下記の視点で量を見積ります。

- 開発する機能それぞれ、一つ一つの大きさ（＝規模、量）はどれくらいか？
- 反復期間内にどれくらいの量をつくれそうか？

優先順位

機能ごとにつくる優先順位をつけます。優先度ではありません。一列に並ぶように順番をつけます。優先順位に従ってつくるためです。同時につくるとリードタイムが長くなるため、一つの機能に集中してつくりきります。

分担して別の機能をつくったり、開発対象の機能を切り替えたりしながらつくると、完了までの期間が延びるため、一つずつつくります。一つずつつくることを製造業ではよく「一個流し」といいます。

iii. アジャイル開発の進捗

冒頭、進捗の管理は進捗率でみることが多いと述べました。

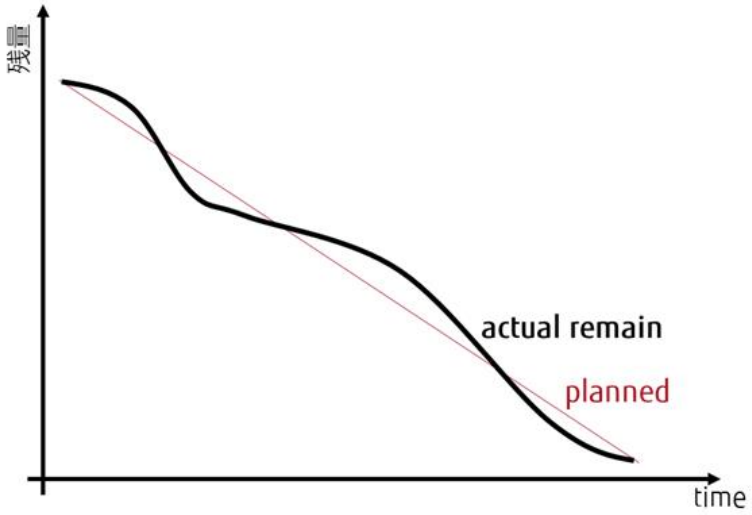
進捗率（%）＝終わった量／全体の量

しかし、進捗率で進捗を管理するアジャイル開発チームを見かけたことはほとんどありません。そもそも、全体の量が決まっていないアジャイル開発では進捗率を計上できないでしょう。

全体の量を決定することはできませんが、反復期間の冒頭には、スコープ（どの機能をつくるか、どれだけの機能をつくるか）を計画します。そして反復期間は固定されています。反復期間内には、バーンダウンチャートを利用して機能の量を追跡することが多いでしょう。

バーンダウンチャート

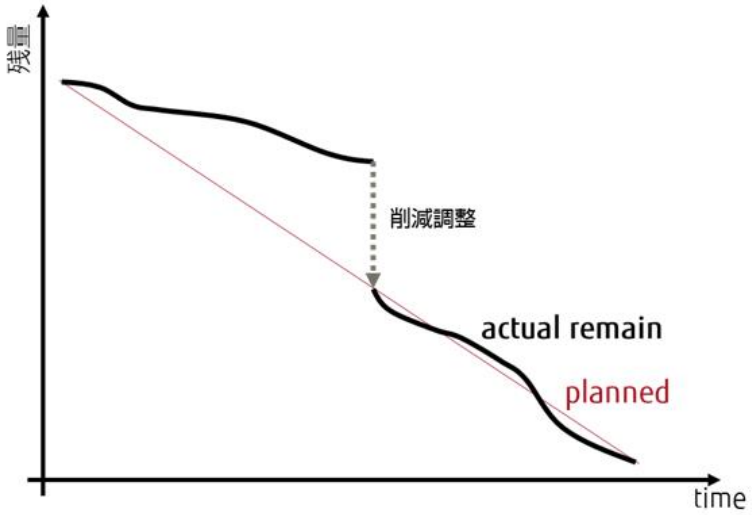
下の図は、バーンダウンチャートの例です。



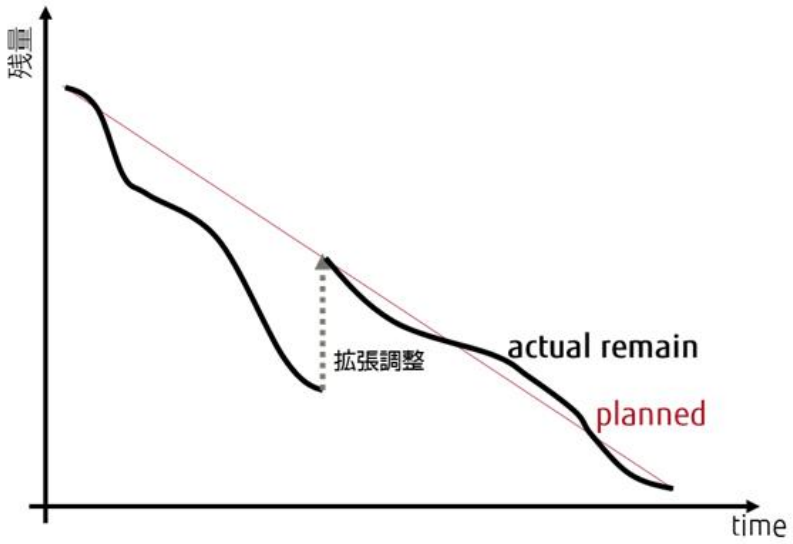
バーンダウンチャートの横軸は反復期間内の経過時間、縦軸は反復期間内につくるつもりの残機能量です。一日 1 回あるいは複数回、その時点での残機能量をプロットします。

進捗管理しない？

多くのアジャイル開発では、計画したスコープが固定された反復期間内に収まりそうにない場合には、機能量を削減します。



また、計画したスコープが反復期間の終了時期よりも早く完了しそうな場合には、機能量を追加します。



進捗管理とは、計画をまもるために、予定と実績とのズレや進み遅れを管理すること。遅れたらリスケして遅れの原因に対策すること

と、述べましたが、計画をまもるためにバーンダウンチャートが利用されている訳ではないようです。

多くのアジャイル開発では、

- 「どれだけつくるのか」の計画自体を変える　～　スコープは可変
- 「いつまでにつくるのか」の計画はまもる　～　反復期間は固定

という具合に、反復期間は固定されているものの、反復期間内に計画したスコープを達成できそうになかったらスコープを減らしてしまいます。計画が必ずしも遵守される訳ではないのです。また、進捗を測る単位である進捗率は、

進捗率（%）＝終わった量／全体の量

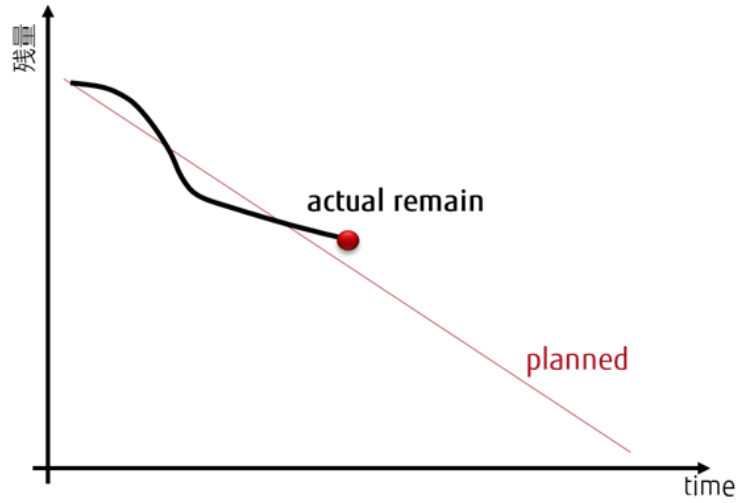
であり、スコープが可変である場合には意味を持ちそうにありません。ということは、アジャイル開発では、進捗管理を実施しないのでしょうか？進捗管理を実施しないのであれば、何を管理しているのでしょうか？

5.2 進捗管理と進行管理

ここからは、バーンダウンチャートを利用して何を管理しているのか？についてみていきましょう。

i. バーンダウンチャートでわかること

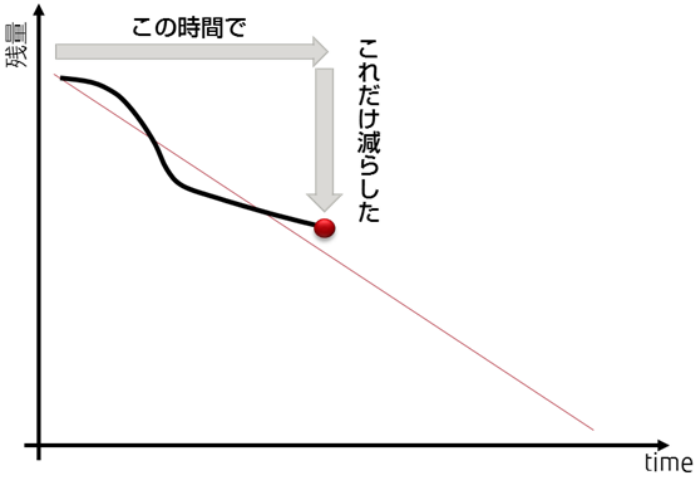
バーンダウンチャートはある時点における残量をプロットしただけの単純な図です。



これがなんの役に立つのでしょうか？いったいなにを管理（マネージメント）しているのでしょう？ここで、視点を変えてみましょう。「バーンダウンチャート、右から見るか？左から見るか？」

過去から時を追う

左から右へと時系列に観察するとどうでしょうか。

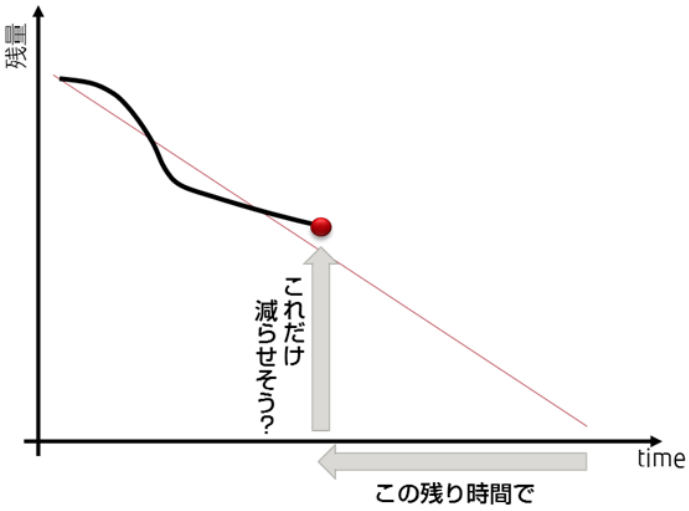


経過した時間で完了させた機能量がわかります。それをどう役にたてるのでしょうか？

未来の予想に使用することはできないでしょう。そもそもアジャイル開発は「予想なんかできない」ということが前提ではなかったでしょうか。それとも「これだけ消化できた」と誰かに報告するためでしょうか。「あちら側」と「こちら側」の間に壁がありそうです。

未来から時を遡ろう

右から左へと観察するとどうでしょうか。



- 残り時間はどれだけあるか
- つくりたい機能はどれくらい残っているか？

この 2 つがわかります。これらは残り時間で残りの機能をつくれそうかどうかの判断材料に使えそうです。

予想ではありません。「ここまでやったこと」と「ここからやること」では、対象の機能が異なりますから、予想する訳ではありません。しかし、反復期間の終了時点に「着地できるかどうか」、「着地させるためにはなにが課題か」を想起するには役立ちそうです。

左から観察しがちではありますが、開発者を含めた関係者たちはバーンダウンチャートを右から見たかったのではないでしょうか。

ii. 進行管理と進捗管理

進捗管理ということばを調べていく中で、進行管理ということばが使われている場面をいくつかみました。しかし進行管理ということばも、進捗管理と同様にその定義を見つけることができませんでした。

進行管理ということばは、行政機関でよく使われているようです。市区町村から国の省庁レベルまで、計画の遂行と目標の達成度合の両方を開示する例が多く、使用されている文脈からは、「目標を達成するために、計画を変更し続けること」と判断できます。 目標を達成することなく計画を順守して失敗することを回避したいのは、公共団体も営利組織も同じでしょう。また、出版業界でも進行管理ということばが使用されているようです。動かせない期限が決定されているなかで定期刊行を繰り返すことに、アジャイル開発の反復との類似性を感じざるを得ません。

進行管理

多くのアジャイル開発では、

- 「どれだけつくるのか」の計画自体を変える　～　スコープは可変
- 「いつまでにつくるのか」の計画はまもる　～　反復期間は固定

という具合に、反復期間は固定されているものの、反復期間内に計画したスコープを達成できそうになかったらスコープを減らしてしまいます。計画が必ずしも遵守される訳ではありません。

対してウォーターフォール開発では

> 進捗管理とは、計画をまもるために、予定と実績とのズレや進み遅れを管理すること。遅れたらリスケジューリングして遅れの原因に対処すること

計画をまもることを是とするのではなく、「目標を達成するために、計画を変更し続けること」 を是とするアジャイル開発でバーンダウンチャートを利用するのは、この進行管理を実現するためのように考えられます。

ウォーターフォール開発の進捗管理とアジャイル開発の進行管理を表にして比較してみます。

	ウォーターフォール開発の進捗管理	アジャイル開発の進行管理
品質	計画をまもる	計画をまもる
納期	計画をまもる	絶対に計画をまもる
コスト	増減する（残業、増員、休日出勤）	ほぼ一定（すこし残業することもある）
スコープ	計画をまもる	つくる機能の量や種類を変える

バーンダウンチャートを参照して、反復期間の残りを見つつ、スコープを調整しながら、固定した反復期間にソフトウェアを出荷可能な状態にするための管理 — 進行管理のためのツールとして、バーンダウンチャートは適しているといえます。

iii. 進捗管理

「進捗管理」、英語では "progress management" というようです。

"progress"という語をいくつかの Web 辞書で調べてみたところ、進歩、発達、発展、前進、前進、進展、向上、成長、発育、上達といった意味を表す名詞あるいは動詞のようです。「進捗」と掲載されていたのは一部の辞書であり、しかも掲載は下位でした。

また、辞書にあたっていく過程で、"proceed"という類義語を見つけることができました。英英辞典（オンライン版ロングマン現代英英辞典）で 2 つの語をそれぞれみてみると、

progress

> the process of getting better at doing something, or getting closer to finishing or achieving something

proceed

> to continue to do something that has already been planned or started

とあります。

- "progress" は、なにかをうまくできるようになる、あるいはなにかが完成・達成することに近づくプロセス
- "proceed" は、すでに計画または開始されていることを継続して行うこと

と訳せるでしょう。

であれば、ウォーターフォール開発の進捗管理は、ほんとうに progress management なのでしょうか？ progress よりは proceed をみているように思われてなりません。

さきほど、「アジャイル開発の進行管理」と述べましたが、どうやら"progress management"を実現しているのは、「アジャイル開発の進行管理」だといってよいでしょう。

iv. Progress Management

バーンダウンチャートの縦軸に、見積り時間（工数）、タスクの見積り時間（工数）あるいは相対見積りによるユーザーストーリーポイントを利用されている事例をしばしば見かけます。

Principles behind the Agile Manifesto （アジャイル宣言の背後にある原則）をみると 7 番目の原則には

Working software is the primary measure of progress.
（動くソフトウェアこそが進捗の最も重要な尺度です。）

とあります。"Working software" は「動くソフトウェア」と訳されていますが、「役に立つソフトウェア」といった意味合いであるでしょう。であれば、時間（工数）あるいは相対見積りであるにせよ、開発者の手間暇を縦軸にとることに疑問を感じざるを得ません。

「ユーザーストーリーの数」を縦軸にすることで、バーンダウンチャートの本来的な役割を果たすのではないでしょうか。ストーリーをより小さなサイズにすること。あるいは（顧客価値の発生単位という意味で）「受入テストの数」を縦軸にとることで、意味のあるバーンダウンチャートが実現され、より効果的な"progress management"が可能となるでしょう。

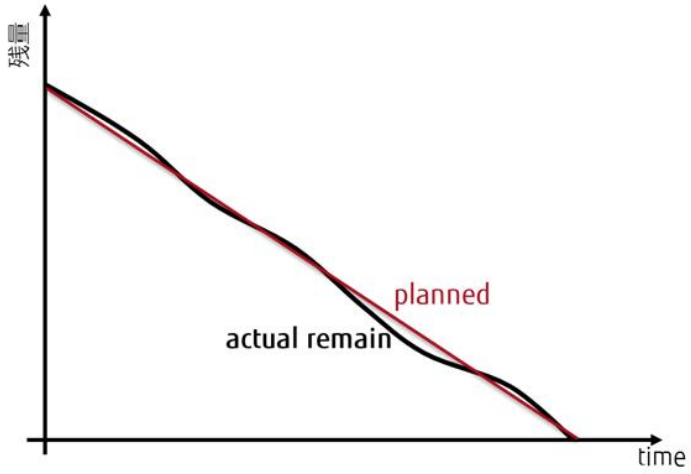
5.3 在庫はリードタイム

ここでは、バーンダウンチャートの形状から観測される事象について紹介します。あわせて、その原因と対策について考察します。

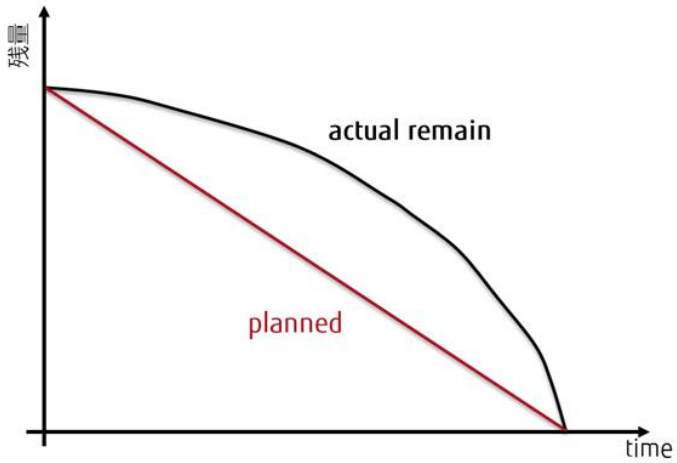
i. 何故か上に凸

バーンダウンチャートは、縦軸に機能残量を、横軸に経過時間をプロットする単純なグラフです。

下の図のように、反復期間の最初に計画した機能残量の減少を、理想線とよばれる（図中では赤）線で示します。また、機能残量の減少を実績線としてプロット（図中では黒い線で表示）します。



上の図は、計画どおりに開発が進んでいったバーンダウンチャートといえるでしょう。しかし実際のバーンダウンチャートをいくつかみてみると、下図のように実績線が上に凸の形状を示していることがしばしばあります。



機能残量が反復期間の終了間際に急激に減少するこの現象については、以下のような言葉で説明されることが多いようです。

学生症候群

夏休みの宿題や試験対策など「期限までまだ時間があるから…」と、なかなか着手しない学生のような状態のこと。

パーキンソンの法則

使える資源（時間やお金）を使い切ってしまうまで使ってしまうこと。計画よりはやく完成していても見積りどおりの時間まで完成したと言わなくなること。

新たな問題の発見

必要な作業の不足や、見込みではうまく動作するはずだった設計など、計画では予期していなかった新たな問題が発見されること。

学生症候群によって着手は遅延する。あるいは、パーキンソンの法則によって前倒しには進まない。しかし、予期せぬ新たな問題の発見のために完成には見積りを超える時間がかかる。つまりバーンダウンチャートの実績線は理想線を必ず上回る。しかし、期限間際で頑張ることによって、反復期間の最後に帳尻を合わせる。

と、説明されていることをよく見かけます。

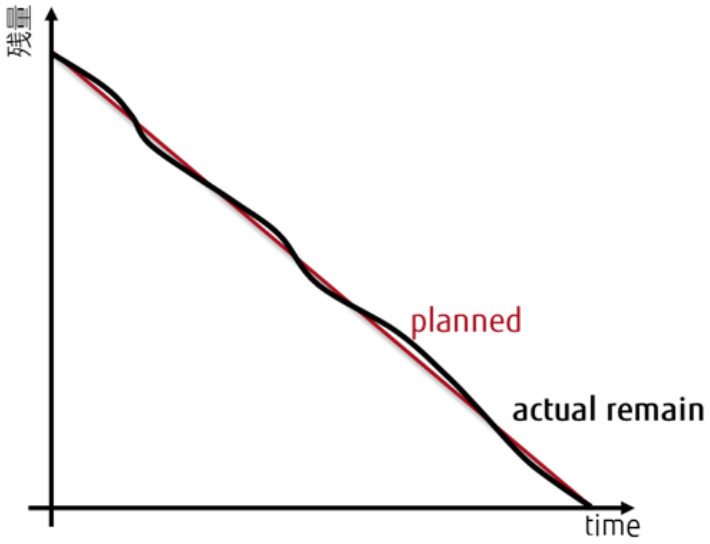
本当にそうなのでしょうか？実際そういう事例があるのかもしれません。

しかし、そうではない事例も実は存在します。そうではない事例とは、学生症候群もパーキンソンの法則も関係なく、新たな問題の発見もない。にもかかわらずバーンダウンチャートが上に凸の形状を示す事例です。

ii. From Concept To Cash

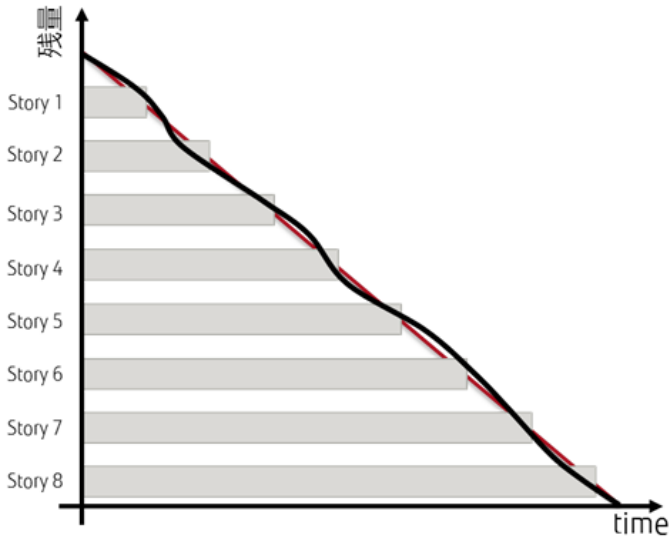
「From Concept To Cash」は、リーンソフトウェア開発で言及されることの多い考え方です。「概念が浮かんでからお金がいただけるまで」そのリードタイム（経過時間）を重視します。バーンダウンチャートの形状からは、このリードタイムを推測することが可能です。

まずは、理想線と実績線がほぼ一致するバーンダウンチャートを見てみましょう。



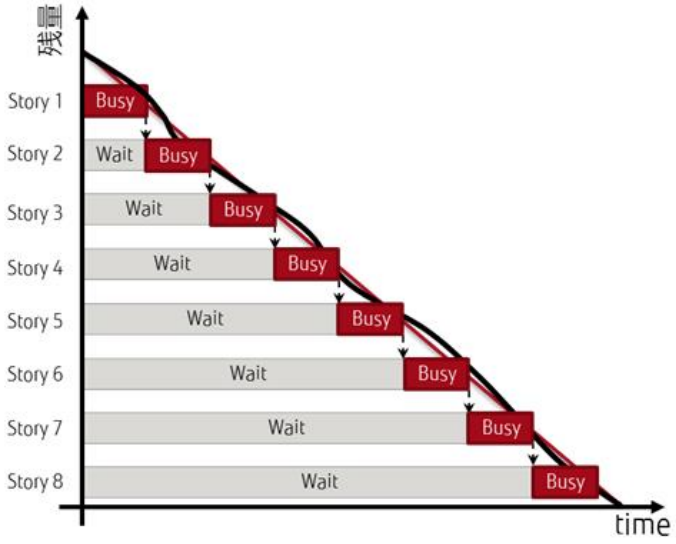
実績線つまり機能残量は、時間の経過にともなってほぼ線形に下降しています。

縦軸にプロットした機能残量が、どのように減少しているのかをバーンダウンチャート内に図示してみます。



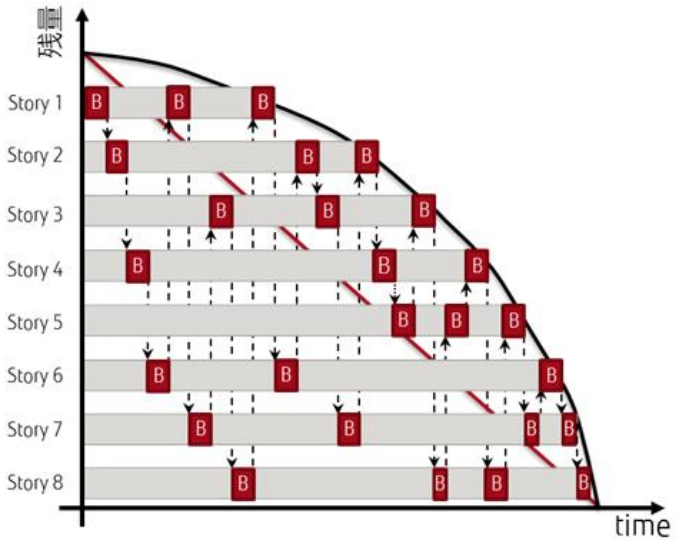
反復期間の冒頭に開発予定であった（図中に Story と表示した）8 つの機能は、順次完了しています。ある時点の機能残量は、その時点に完成していない機能の数（チャート内の機能生存を表すバーの本数）です。

さらに、生存状態の機能（つまり未完了の機能）の様子を詳しく示してみましょう。



各機能を示すバーをグレーと赤で塗り分けました。グレーは（作業中ではない）Wait 状態を、赤は(作業中である）Busy 状態を示します。各機能は同時に開発される訳ではなく、一個流し（同時着手することなく一個ずつつくる方法）で開発されていることがわかります。

では次に「学生症候群」も「パーキンソンの法則」も関係しない。「新たな問題の発見」も無い。にもかかわらずバーンダウンチャートが上に凸の形状を示す場合についてみてみましょう。



各機能の生存状態を見てみると、Wait 状態と Busy 状態がまだらになっていることがわかります。つまり一個流しで開発されていないことがわかります。

さらによく見てみると、Busy 状態の時間の合計は、理想線と実績線が一致する場合と同じであるということもわかります。「学生症候群もパーキンソンの法則も関係なく、新たな問題の発見もない」ということ

は、各機能を完了させるためにかかる時間は変わらないということですから、時間の合計値が同じになります。

時間の合計値、より正確には工数（人時：ManHour）は両者ともに同じです。しかし、リードタイムつまり各機能が完了するまでの経過時間には相当な開きがあります。これは Wait 状態の時間によるものです。

一個流しでない場合には、Wait 状態によってリードタイムが長期化するため、バーンダウンチャートは上に凸の形状を示すのです。

iii. 在庫はリードタイム

バーンダウンチャートから観測できる「ある時点の機能残量」は、その時点に完成していない機能の数（チャート内の機能生存を表すバーの本数）です。製造業でいう「在庫（中間在庫、あるいは仕掛品）」の数といえるでしょう。ソフトウェア開発では WIP（Work In Progress あるいは Work In Process）と呼ばれます。

WIP が大きければ大きいほどリードタイムは長くなり、「一個流し」の場合、つまり「WIP=1」の場合にリードタイムは最短になります。これは、WIP（=在庫）を制御することでリードタイムを制御できることを意味します。

しかし、一個流しにしたからといって Velocity（反復期間あたりに完成する機能の数）が向上するわけではありません。各機能のリードタイムが短縮されるだけです。また、コストダウンされる訳でもありません。完成する機能の数が同じであり、稼働工数も同じだからです。

	WIP=1（一個流し）	**WIP=大**
リードタイム	最短	WIP の増加につれて長期化
コスト	WIP で変化しない	
Velocity	WIP で変化しない	

それでは、理想線と実績線がほぼ同一である（=各機能のリードタイムが最短である）ことのメリットとはどのようなものなのでしょうか？

iv. リードタイム短縮のメリット

リードタイム短縮のメリットはアジャイル開発のメリットそのものです。

開発者やチームの成長を促進 ～ 学習の視点

複数人が開発作業を共にするチームで一個流しを実現するためには、全員が同時に一つの機能の開発にあたる必要があります。機能一つずつの設計、実装、テストに対する知見を共有し開発者やチームの成長を促します。また、全員が同時に作業できるシンプルな設計へと洗練されます。これは品質の向上も促します。

品質と顧客満足度の向上 ～ リスク早期検知の視点

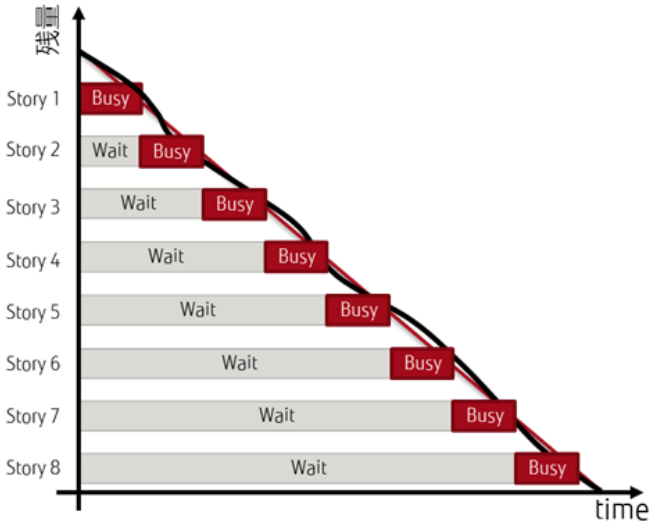
一個流しは、機能を一つ一つ順に利用可能な状態まで完成させます。要望の誤解、技術的な実現可能性や品質の検証が早期に実施されるため、故障も少なく満足度の高いソフトウェアが提供されます。

要求の変化に柔軟な対応が可能 ～ 変動対応性の視点

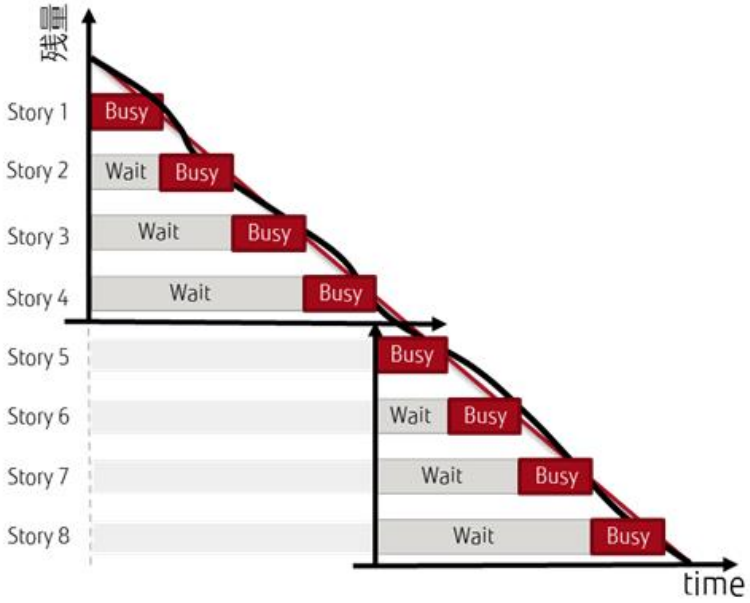
着手中の機能は一つ（一個流し）だけです。要求の変更が発生しても変更によるインパクトは最小化されます。このため要求の変化に柔軟に対応できます。

より速いスピードでの提供 ～ 正味現在価値の視点

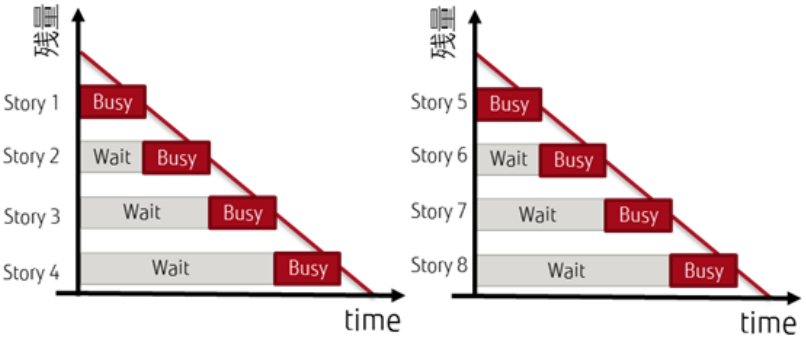
理想線と実績線がほぼ同じであるということは、反復期間を短縮できるということです。



上図のようであれば、次の図のように反復期間を半分にできます。



つまり、2 回の反復に分割し、反復回数を倍増できるということです。



反復期間が半分になり反復回数が倍になれば、要望の高い機能がより短いサイクルでリリースされます。要望から提供までのスピードが速いため、利用したい機能をタイムリーに利用できます。

このように一個流しによるリードタイムの短縮（WIP=1）は、アジャイル開発のメリットを最大化するものに他なりません。

v. 持つべき視点

規模の経済のメリットを追求するウォーターフォール開発に対して、スピードの経済のメリットを追求するのがアジャイル開発です。スピードすなわちリードタイムの最小化によって、組織、プロセスや品質、顧客満足、財務状況が向上されます。

バーンダウンチャートは、縦軸に機能残量を、横軸に経過時間をプロットするだけの単純なグラフです。しかし上に凸のバーンダウンチャートを「最後にがんばったから期限に間に合った」とだけ見るのは少しもったいない話です。その背後にある WIP やリードタイムの構成への視点を持つことで、よりバーンダウンチャートが有益なものとなります。この視点は、アジャイル開発者であれば当然持って然るべきといえるでしょう。

第6章 アジャイル開発の士気管理

6.1 ニコニコカレンダー

情報システムは、「ハードウェア」と「ソフトウェア」そして「ピープルウェア」ともいえる開発チーム（開発者たち）によって、かたちづくられています。チームの士気がピープルウェアのパフォーマンスを左右する大きな要因であることは自明でしょう。

ここでは、生産性や品質に大きな影響を与える「チームのムード」や「メンバーの気持ち」といったチームの士気を見える化するツール「ニコニコカレンダー」を事例とともに紹介します。

i. ニコニコカレンダーとは

下のサンプルを見てください。

	1(水)	2(木)	3(金)	4(土)	5(日)	6(月)	7(火)	
鈴木								
佐藤								
田中								

ニコニコカレンダー

名前（行）と日付（列）の表形式のカレンダーにフェースマークを記しています。3 種類のフェースマークには、それぞれ意味があります。

マーク	意味
	いい感じ、ハッピー
	普通、ぼちぼち
	やな感じ、どんより、憂鬱

フェースマークの意味

メンバーの気持ちをフェースマークにしてカレンダーに貼り付ける。それがニコニコカレンダーです。

壁に貼る

ニコニコカレンダーは、チームのムードやメンバーの気持ちを見える化するツールです。効果的な見える化のために、メンバーからいつも見える壁に貼りましょう。

大きさは、A4 ないし A3 程度の大きさが適当でしょう。

- 行にはメンバー
- 列には日付を書きましょう。

簡単に表（カレンダー）をつくったら、壁に貼りましょう。

気持ちを貼る

カレンダーには気持ちをフェースマークで貼りつけます。これにはシールがおすすめです。

- 100 円ショップで売っている
- 丸いシール
- 8mm～16mm

素材のシール

気持ちの表し方はいくつかあります。

マーク	意味
	いい感じ、ハッピー
	普通、ぼちぼち
	やな感じ、どんより、憂鬱

気持ちの表し方 1

こういう風にシールに顔を書く方法もあれば

マーク	意味
	いい感じ、ハッピー
	普通、ぼちぼち
	やな感じ、どんより、憂鬱

気持ちの表し方 2

こういう感じで交通信号のように表す方法もあります。

特にルールがある訳ではありません。チームの雰囲気に合うやり方でよいでしょう。

フェースマーク

あらかじめ用意したフェースマーク

シールに顔を書くにあたっても、

- 各メンバーがカレンダーに貼るたびに顔を書く
- あらかじめまとめて、顔を書いておく

など、方法は一つではないのですが、その都度、各メンバーが自分で書くことをお勧めします。

ニコニコマークを書いていると自身の表情もニコニコして来ます。不思議なことですが、多くの方がそう言います。また、顔がニコニコしてくると、気持ちまでニコニコしてくるから不思議です。

日々繰り返す

顔入りシールは自身の分身です。顔入りシールは、**その日の帰り際に各メンバーが自分で貼る**のが良いでしょう。「その日、一日なにがあったか？」、「なにがうれしかったか？」、「残念なことはなかったか？」など、**自分の中での小さなふりかえり**をしてみましょう。そしてシールに自分の気持ちを託しましょう。

毎日シールを貼り続けます。しかしそれだけではありません。夕方、帰宅するときにシールは貼られます。貼られたシールは、メンバーの気持ちです。チームメンバーみんなで見ましょう。翌朝の朝会でみんなで見ることも良いでしょう。しかし多くのひとのまえで、ことさらに指摘することを嫌うひともいます。あくまで慎重に、しかし注意深くみんなの気持ちに注意しましょう。

“みる”といってもいろいろあります。

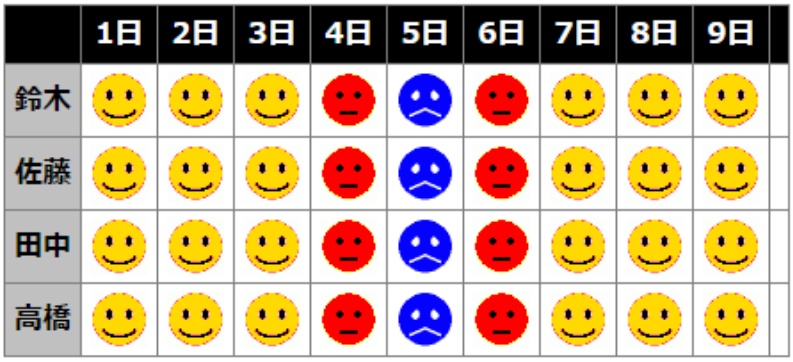
- 見る
- 視る
- 観る
- 診る
- 看る

単に見るだけでなく、注意深く、中身までみて、判断し、診断し、**看る**ところまで心を配りましょう。

ii. 事例

いくつかの事例を紹介します。

チームの変化



同期した変化

メンバーの気持ちが同期しているようです。たとえブルーな日があったとしてもチームのムードとしては自然な変化といえるでしょう。

	1日	2日	3日	4日	5日	6日	7日	8日	9日
鈴木									
佐藤									
田中									
高橋									

1 人だけがブルー

1 人だけが毎日ブルー。しかし他のメンバーは毎日ハッピーだと感じています。あきらかに何か問題があるといえるのではないでしょうか？

	1日	2日	3日	4日	5日	6日	7日	8日	9日
鈴木									
佐藤									
田中									
高橋									

みんな毎日ふつう

毎日みんなふつうのまま変化がない。本当のことでしょうか？健全なのでしょうか？

独りでふりかえり

ニコニコカレンダーは独りでもできます。下の写真はその例です。

独りでニコニコカレンダー

この例ではシールではなくマーカーとペンで卓上カレンダーに記入しています。他にみてくれるひとがいなくても、日々のふりかえり習慣として有効でしょう。

労働時間

ニコニコカレンダーに毎日の労働時間情報を併せて記載する事例もあります。

残業時間を併記

残業時間が少ないのにブルーな気分が続いている…。そんな様子を見かけたときには「メンタル的になにかある？」、「なにかのサイン？」と認識して声かけをするようにしていた。

と、このチームのメンバーは語っています。

6.2 プラクティスは実践知

ニコニコカレンダーは著名なアジャイリストが発明したプラクティスではなく、ふつうの人たちがつくり出したプラクティスです。ここでは士気管理プラクティスとしてのニコニコカレンダーについて考察します。

i. ソフトウェアメトリクスとして

2005 年、当社における「ソフトウェア開発の見える化」という流れ、つまり「測定し、可視化し、改善する」という方針のもとで、製造業の視点である生産管理指標（QCD,IP,SM）に着目し、ソフトウェア開発の見える化を図ったことが、ニコニコカレンダー誕生のきっかけのひとつです。

表 ソフトウェアメトリクスとしての QCD,IP,SM

項目	意味・測定方法
Q（Quality:品質） C（Cost:原価） D（Delivery:納期）	いわゆる QCD です。 これらは直接にあるいは代用特性を利用して測定されます。
I（Inventory：在庫） P（Productivity：生産性）	ソフトウェア開発では、バーンダウンチャートで見える化が実現されています。 縦軸が在庫。傾きが生産性と考えてよいでしょう。
S（Safety：安全性） M（Morale：士気）	**精神的な健康**、目標の達成や能力向上の**意欲**のことです。 測定は困難です。

測定困難な SM（Safety, Morale）、"安全性"と"士気"について見える化を試みたのが、ニコニコカレンダーであるといえるでしょう。

ii. 見える化

ここまで「見える化」ということばを度々使いました。日本語として少々奇妙なことばのように感じますが、元々は製造業に由来しているようです。

異常の露出

見える化で最も重要なことは"異常"を見えるようにすることです。この"異常"ということばも製造業にかかわる人たちの中では少し特殊な意味で使われています。

> 異常とは、“不良ではないが、正常ではない状態”、“正常と不良の間にある状態”

例えば、

- 機械は異音を発しているが、不良品を出すことなく動作し続けている。
- 目の下にクマを浮かべているが、バグをつくりこむことなく開発し続けている。

「決して正常ではない状態」、「不良に至りそうなサイン」を指すことばです。

いきなり不良が発生する状態は、決して好ましくありません。ましてや SM（Safety, Morale）における不良とは、人の安全にかかわることです。

「正常ではない状態」や「不良に至りそうなサイン」を見逃さないために、「日々少しずつ変化を見よう」、「傾向を見よう」ということなのです。

これを異常管理といいます。

フィードバック

いくら異常を監視していても、なにも対処しなければ管理・マネジメントしていることにはなりません。フィードバックがあることが重要なのです。異常管理の原則といえます。

ニコニコカレンダーの場合、そのフィードバックとはいったいなんでしょう？リーダーやメンバーがケアする。手伝う。仕事をシェアする。いろいろあるでしょう。単に話をきくだけでも大きな効果があるでしょう。（看るではなくても）見ていることが伝わるだけでも、そのストレスは軽減されるようです。

フィードバックがなければ、嫌になってしまいます。絶対嫌になってしまいます。そしてやらなくなってしまいます。

しかし、ニコニコカレンダーは継続性が高いプラクティスであることが、多くの現場から伝えられてきます。それは、リーダーやメンバー以外からの重要なフィードバックがあるからです。

擬人性

重要なフィードバックは自分自身から発せられます。毎日自分自身でシールに顔を書きニコニコカレンダーに貼る。その際、小さなふりかえりを実施しているためです。

カレンダーに貼ったシールの顔と見つめあいながら、一日をふりかえり「よかったこと」、「よくなかったこと」、「ああすればよかった」、「今度はこうしよう」という思いが巡るのです。

顔があることでシールはある意味、擬人性を帯びるのではないでしょうか？

感情を露わにしない習慣のなかで過ごしてきた開発者たちが、小さなふりかえりをおこないつつ、チームの仲間へもサインを送る。そんな毎日の繰返しが、精神的浄化作用を生んでいるのではないでしょうか。毎日浄化。少しずつ浄化するプラクティスがニコニコカレンダーです。

メトリクスからプラクティスへ

このように、SM（Safety, Morale）のメトリクス、あるいはメトリクス測定のツールだったニコニコカレンダーは、測定の営みの中で、日々を"ふりかえり"、"貼る"、そして"みる"といった一連の行動そのものが、SM（Safety, Morale）そのものを高めるプラクティスへと変遷していったのです。

iii. プラクティス

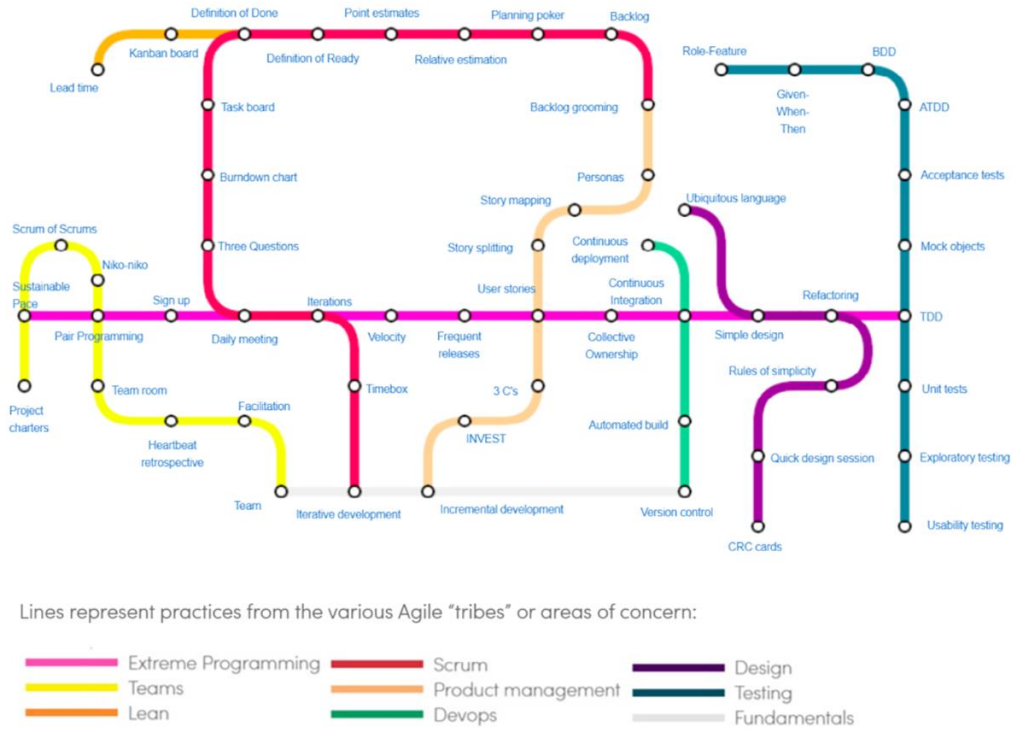
アジャイル開発の主要な方法論の 1 つであるエクストリームプログラミング（XP：eXtreme Programming）は、価値（Value）、原則（Principle）、プラクティス（Practice）の 3 階層で記述されています。しかし、なぜか"プラクティス"を日本語に翻訳された例を見かけることはほとんどありません。（エクストリーム・プログラミング入門の初版では、「実践」と訳されていましたが、2 版からはプラクティスと訳されています）

しかし、「プラクティス」ということばはどうもピンと来ない気がします。

Niko-Niko

ニコニコカレンダーは「This Isn’t Kindergarten」などいくつかの批判はあるものの、多くの言語で紹介され論文からの参照も少なからずあります。

また、"Niko-niko Calendar"としてアジャイルアライアンスの Web サイトにアジャイルプラクティスとして掲載されています。



Subway Map to Agile Practices（アジャイルアライアンスの Web サイトから引用）
（https://www.agilealliance.org/agile101/subway-map-to-agile-practices/）

アジャイルアライアンスのアジャイル用語集やプラクティスタイムラインにも掲載されており、スタンダードなプラクティスの 1 つと認識されているようです。

経験則

「ニコニコカレンダー」は「安全性や士気を高めるプラクティス」であることは、おそらく確かです。チームのムードをみてフィードバックし、安全性や士気を高める施策をとっているからです。勘定して帳面につけるだけの管理ではありません。アウトプットからプロセスを調整するフィードバックが存在する本来的な管理（マネジメント）プラクティスです。

そしてそれは、著名なアジャイリストが発明したプラクティスではありません。ふつうの人たちがつくり出したものであり、つくり出した人たちの役に立つものでした。

エクストリームプログラミングに詳しいある友人はこう云いました。

> ニコニコカレンダーのいいところっていうのは「プラクティスって自分たちでつくっていいんだ」ってことを示したことじゃない？”

プラクティスの中身も、プラクティスのつくり方も、自分たちが、自分たちをみて、自分たちで判断して、自分たちのやり方を変える。自律的につくるのがプラクティスです。

士気管理プラクティス「ニコニコカレンダー」そのものの効用よりも、「ニコニコカレンダー」が自分たちでつくったプラクティスであるが故に、安全性や士気を高めていったのかもしれません。であれば、士気管理を狙っていないプラクティスであっても安全性や士気は十分に高められそうです。

自分たちでつくったプロセスの中で、自分たちのプラクティスを活かすことこそが、士気を高めパフォーマンスを向上させる源泉であるということを、スクラムでは「**自己組織化**」として、エクストリームプログラミング（XP:eXtreme Programming）では「**実践**（初版訳）」として語られているのではないでしょうか。

さきほど、「『プラクティス』ってことばはどうもピンと来ない気がします」と述べましたが、書籍「ソフトウェア・テストの技法 第 2 版」で、"practice"ということばは"**経験則**"と訳されています。"経験則"、これが最もしっくりくる訳語であると思われてなりません。

第7章 「アジャイルなら Git」の神話

i. 背景

ここ数年、職場でも Git を使うことがわりと当たり前になってきました。筆者が個人的に Git を使い始めた頃、こんな複雑なバージョン管理は職場で普及しないだろうと思っていました。何より Subversion では積極的に使う気になれなかったブランチを誰もが作ってマージしたりするとは想像できませんでした。

それが今ではどうでしょう。アジャイル開発関連のプレゼンテーションを見てみると、そのほとんどが Git を利用しているようです。Subversion を利用している事例はほとんど見かけません。アジャイルなら Git を利用することがデファクトスタンダードのように見えます。

そんなある日、社内で先輩に言われたことがあります。

「Git はフォークやブランチによって派生を促す傾向が強く、イノベーションを生む OSS に向いている。一方で分断を助長し、アジャイルが志向する常時結合を阻害する側面もある。」

最初に聞いたとき、なんとなく分かるも、釈然としないところもありました。**ブランチを手軽に作って機能の開発に集中できることこそ Git の強み**であり、Subversion などほかのバージョン管理システム（VCS）と差別化される特徴だと思っていたからです。

もちろん常時結合（CI: Continuous Integration）の意味や目的は、唯一のコードベースを動作する状態で維持し、そのコードベースを基点にして全ての開発者が共有し、進化させることにあります。

ですから、Git の便利な機能は、その使い方に気を付けないと逆にアジャイルな開発からは遠ざかってしまいそうです。機能の開発に集中できるということは、言い換えればほかの開発者やブランチのことを無視した、ひとりよがりな開発になるかもしれません。

Git の強みである「手軽さ」や「軽快なブランチ機能」を活かしながら、唯一のコードベースへの常時結合を志向する「よりアジャイルな開発」を実現するにはどうしたら良いのでしょうか。

ii. トピックブランチの功罪

Git で一般的に使われるトピックブランチの功罪について改めて整理してみます。

なお、**トピックブランチ**[1]とは「一つのテーマ（トピック）に集中して、その上では一つのテーマを解決する以外の仕事はしないようなブランチ」のことです。一方、「トピックブランチを分岐する元になり、また、完成したトピックブランチでの変更を併合するブランチ」は**統合ブランチ**[1]と呼びます。一般には master や main ブランチを統合ブランチに当てます。

開発者たちにとっての利点

前述したとおり、機能の開発に集中できることがブランチの利点でしょう。

- 統合ブランチからの影響を避けられる

  トピックブランチで開発を進めている間に統合ブランチが進んだ（追加のコミットがあった）としても、別のブランチですから影響を受けることなく開発を継続することができます。

- 統合ブランチへの影響を避けられる

  レビューやテストが不十分なままコミットしても、それが統合ブランチでなければ統合ブランチへの影響を避けることができます。

- 開発途中のコードをローカル以外に残せる

  機能の開発がすぐに終わりそうにないとき、ブランチを分けていなければ完成するまでローカルディレクトリーに変更を残し続けなければなりません。

  しかし、ブランチを分けてコミット（プッシュ）しておけば、もし翌日、ローカルの環境が壊れたり、急用（病気とか含めて）で続けられないときにも引き継げるので安心です。

- コミットを使ってレビューできる

  GitHub や Bitbucket のプルリクエスト、GitLab のマージリクエストを使ってコードの変更差分を確認しながらレビューをすることができます。差分の中にレビューのコメントを残すことも簡単です。

常時結合を志向するアジャイル開発にとっての弊害

[1] 濱野純「入門 Git」7.6 より引用。

トピックブランチを使って複数の開発者がそれぞれ集中して開発できることは、一個流し[2]の阻害、すなわち WIP[3]の増加の要因にもなり、様々な弊害を生みます。

- 統合ブランチとの乖離によるコンフリクト

  複数人で開発していれば、ブランチで分かれて開発している間に統合ブランチも変化します。その期間が長くなるほど、コンフリクトが発生しやすくなります。コンフリクトしたらそれをマージするときに解消しなければなりません。

- 常時結合の阻害

  トピックブランチ上で CI ツール等を使って自動テストをしていたとしても、そのときの統合ブランチの変更が反映されていないわけですから、Continuous Integration、つまり継続的に統合できている状態とは言えません。

- レビュー機会の減少

  それぞれの開発者はそれぞれのトピックブランチを使って開発しているわけですから、完成前でもドラフトのプルリクエストを作って頻繁にプッシュするなど工夫と意識付けをしないと、レビューがマージの直前だけになってしまいそうです。

iii. トランクベースとトピックブランチの開発の比較

Git のトピックブランチを利用した開発がそれほど普及していなかった頃、つまり Subversion を使っていた頃はトランクベースでの開発が一般的でした。Subversion のブランチはその機構上、動作が重く、機能的にも決して使い勝手が良いものとはいえなかったからでしょう[4]。また、Subversion は分散リポジトリーではないため、Git ほどにはブランチを利用する旨味が少なかったからでもあるでしょう。

ここで、Subversion でよく用いられるトランクベース開発と Git でよく用いられるトピックブランチベースの開発、それぞれにおける特徴を整理してみます。

---

[2] 「一個流し」については「1.3 スクラムと XP の調和」を参照。

[3] Work In Progress (または Work In Process) の略。「5.3 在庫はリードタイム」も参照。

[4] 「Subversion だとブランチを作るのが恐ろしい。ソースファイルが多くなると、ブランチの切り替えの時間が長くなるからだ(何分もかかる)。」(Robert C. Martin「Clean Agile 基本に立ち戻れ」第 6 章より)

	トランクベースで開発	トピックブランチによる開発
タスクやストーリーへの集中	統合ブランチの変更に常に対処する必要がある	統合ブランチや他のブランチの変更を気にせず集中できる
プロジェクトへの参入障壁	高い（統合ブランチにコミットできる権限が必要）	低い（フォーク/クローンしたリポジトリーがあれば良い）
プルリクエスト機能	使えない	使える
WIP	コミットを短時間でトランクに反映するため少なく抑えやすい	コミットをブランチで保持するため増えやすい
タスクやストーリーの大きさ	コミットを短時間でトランクに反映できるように小さくしようとする力が働く	ブランチで隔離できるため小さくする動機が起きにくい
レビュー	統合された状態で継続的にレビューできる	マージ前にレビューできる（特に開発者のスキルに懸念がある場合に有用）
継続的インテグレーション	コミットする度に実施	ブランチをマージしたときのみ実施（マージ前のブランチをプッシュして CI ツールでテストを実行することは可）

開発者にとってトピックブランチを使った開発、つまり Git の軽快なブランチ機能を利用した開発は、ほかの開発者やほかの機能を気にすることなく心地よいものです。

一方、アジャイルという観点で見ると、唯一のコードベースを維持し、全ての開発者がそれをもとに開発できることが理想です。それを実現するためには WIP を最小化し常時結合することが必要になります。

やはり、Git の便利で軽快なブランチ機能と、常時結合を志向するアジャイル開発の間に何らかの調和が必要そうです。

iv. トピックブランチによる開発

Git が普及したのは GitHub の影響が大きいでしょう（筆者もそうでした）。

GitHub が OSS のフォークを簡単なものにし、プルリクエストなどを使ったソーシャルコーディングへと発展しました。従来、メーリングリスト等で行っていたであろうパッチの説明やレビューをプルリクエストは容易なものにしました。

ソーシャルコーディングの開発スタイルは、OSS でなくても便利に使うことができます。

Git ならタスクやユーザーストーリー毎にブランチを分けることも簡単です。一つのユーザーストーリーが完成するまでの間、ほかの開発者のコミットからは少し距離を置いて開発に集中することができます。コーディングが終わったらマージ前のブランチを同僚にレビューしてもらい、テストがパスすることを確認してから統合ブランチにマージすれば良いのです。そして、このマージまでのプロセスに GitHub などが持つプルリクエストの機能がそのまま使えます。

このソーシャルコーディング由来の開発スタイルが普及するとともに、トランクベースの開発スタイルが減っていったと考えられます。

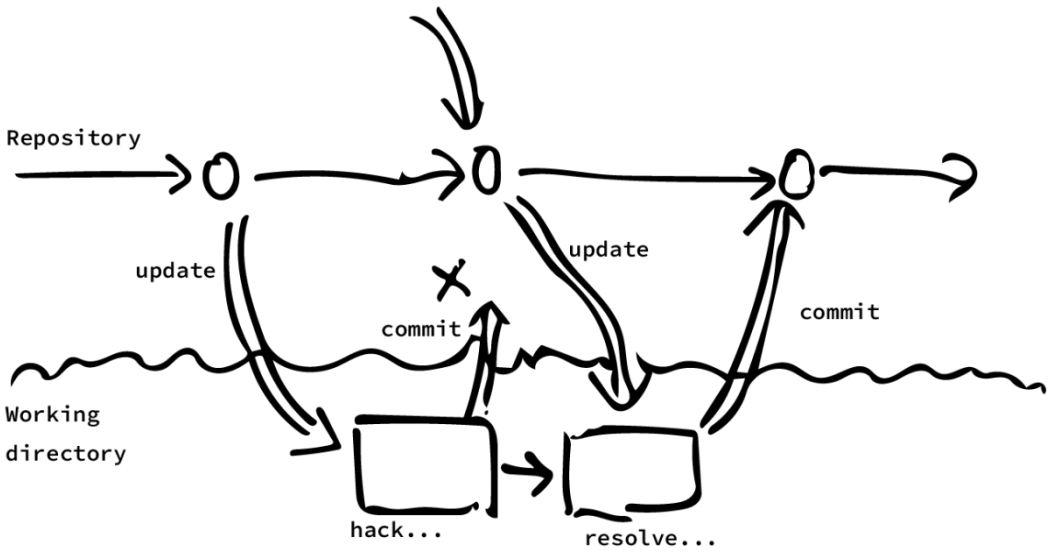
また一方で、トピックブランチによる開発はその扱いに注意しないと、WIP の増大や、ブランチが長くなって多くのコンフリクトを誘発することになります。

v. トランクベースでの開発

Subversion の場合

Subversion は集中型リポジトリーですので、コミットがすなわちチームで共有しているリポジトリーへの反映を意味していました。

このとき、コミットする前にもし別の人がコミットしていた場合、svn update によって先に行われたコミットを**作業ディレクトリー**にマージ（場合によってはコンフリクトの解消も）しなければなりません。



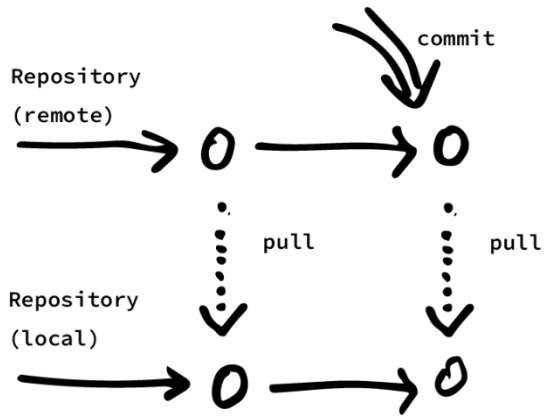
このコミット前の**一連のプロセスによって常時結合するよう制約していた**ことになります。

Git の場合

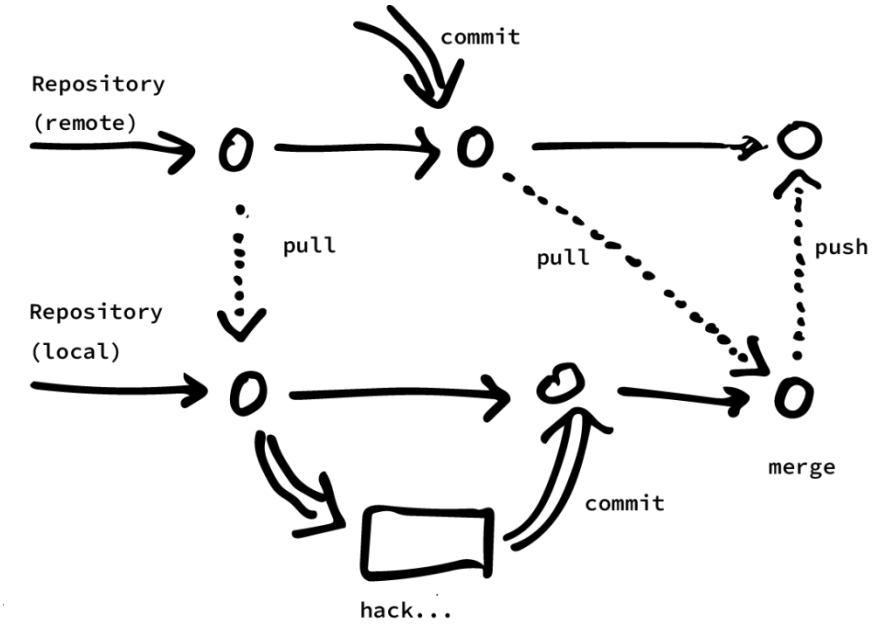
さて、Git でトピックブランチを作らなかったらトランクベースの開発となるでしょうか。

Git では、リモートリポジトリー（ブランチ）の変更をローカルに反映するのに git pull を使います。

通常は、ローカルリポジトリーでリモートに未反映のコミットがなければ、リモートの変更がローカルリポジトリーにそのまま反映されます（fast-forward マージ）。



一方、ローカルリポジトリーでリモートに未反映のコミットがあれば、それぞれのコミットはそのまま残され、更にそれぞれのコミットをマージまでできます。



つまり、細かくコミットの分岐や統合が発生することはあるものの、設定やオプションの使い方に気を付ければ、**トランクベースで開発を進めることは Git でも実現できそうです。**

Subversion と比べて安全な Git

Subversion ではコミット前の作業ディレクトリーとトランクの最新のコミットを常にマージしながら開発しています。

もし、マージに失敗して最初からやり直したくなったとしても、変更したファイルは作業ディレクトリーのみにしかないため元に戻せなくなる危険性があります。

Git ではローカルにコミットした後、リモートの最新のコミットとマージしながら開発します。

このため、マージに失敗しても変更前のファイルはローカルにコミットしていますから、そこから元に戻して最初からやり直すことができます。

常時結合を志向したトランクベースの開発であっても、Subversion より安全で、しかも軽快で使い勝手の良い Git の特徴を活かすのが良さそうです。

vi. Git で WIP の最小化を目指す

Git を利用し、Git のトピックブランチを利用する。その際、トピックブランチを短く小さくなるように工夫したらどうでしょう。ブランチを簡単に作ることもできれば手軽に捨てられるのも Git ならではです[5]。トピックブランチに対応する**タスクやストーリーをテスト可能な最小単位にまで小さくする**ことによって、WIP を 1 に固定したトランクベースと言わないまでもアジャイルの志向する WIP を削減し、継続的インテグレーションを活かした開発に近付けることができるのではないでしょうか。

もちろん、「最小単位にまで小さくする」とはいっても実現の難易度は決して低くはありません。難易度はソフトウェアの構造に起因しますし、その構造は開発者の能力に依存するからです。しかし、WIP の最小化にむけて、より良い構造とより良い設計を目指す開発者たちの行動が、開発者たちの能力向上とソフトウェア構造の改善に繋がるでしょう。

vii. むすび

これを書くにあたり、入門 Git - 秀和システムを読み返しました。少し古い本で、コマンドのデフォルトの動作など今とは変わっているところもありますが、ツールの基礎からその思想をふりかえるのは改めて良い学びになりました。

---

[5]「コードをすばやく捨てるために Git を使うなんて、開発者のリーナス・トーバルズも信じられないと思っているだろう。」(Robert C. Martin「Clean Agile 基本に立ち戻れ」第 6 章より)

GitHub などのコードホスティングもどんどん使い易くなっていますが、機能の使い方に目を奪われて、自分達のやりたかったことと合っているのか考えることが減ったかも知れません。

Git の成功したブランチモデルとして 2010 年に git-flow が公開されましたが、これを提唱した Vincent Driessen 氏はその 10 年後、ブランチの扱い方に万能薬はなく、どのようなやり方が適しているかは自ら考えなければならないと注意を付け加えています。

迷ったときこそ、慣習にとらわれず、基本に立ち戻って自ら考え直す良い機会になります。

参考文献

第１章 アジャイル開発とは

１.１ アジャイル開発の概要とウォーターフォール開発との対比

藤本聖, 平鍋健児. “ - XP は私達に何をもたらすのか". オブラブ. 2002. http://objectclub.jp/community/XP-jp/xp_relate/xp_present#sec5, (参照 2018-8-24).

古垣幸一他. "トヨタ生産方式の導入によるソフトウェア 開発プロセスの革新". 雑誌 FUJITSU. 2005, Vol.56, No.6, p.605-613. ISSN 0016-2515.

Jim Highsmith. “アジャイルソフトウェア開発エコシステム". 株式会社テクノロジックアート訳, 長瀬嘉秀監訳, 今野睦監訳, 株式会社ピアソン・エデュケーション, 2003, 404p, ISBN 978-4894717374.

落水浩一郎, "WATERFALL Model 再考 ソフトウェア開発管理とソフトウェア開発方法論の融合について", SEAMAIL. 2006, Vol.14, No.9-10. 岸田孝一編. ソフトウェア技術者協会.

小椋俊秀, "ウォーターフォールモデルの起源に関する考察 ウォーターフォールに関する誤解を解く", 商学討究. 2013, 第 64 巻, 1 号, 小樽商科大学, p.105-135. ISSN 0474-8638.

Winston Walker Royce, "Managing the Development of Large Software Systems: Concepts and Techniques", Proc. of IEEE WESCON, p.1-9, 1970 (Proc. of 9th ICSE, p328-338, 1987, ISBN 978-0897912167 に再掲).

佐藤善昭. "ケーススタディ 富士通ソフトウェアテクノロジーズ"アジャイルとトヨタ生産方式が融合した俊敏な開発プロセス"". PM magazine. 2005, vol.003, p.48-50, ISBN 978-4798107707.

１.２ スクラムとエクストリームプログラミング（XP）

Kent Beck, Cynthia Andres. “Extreme Programming Explained: Embrace Change". 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

株式会社永和システムマネジメント. “ - eXtreme Programming - Embrace Change". オブラブ. http://objectclub.jp/community/XP-jp/xp_relate/essential-ec, (参照 2018-8-10).

江渡浩一郎. “パターン、Wiki、XP－時を超えた創造の原則". 技術評論社, 2009, 224p, ISBN 978-4774138978.

Jim Highsmith. “アジャイルソフトウェア開発エコシステム". 株式会社テクノロジックアート訳, 長瀬嘉秀監訳, 今野睦監訳, 株式会社ピアソン・エデュケーション, 2003, 404p, ISBN 978-4894717374.

独立行政法人情報処理推進機構, “アジャイル型開発におけるプラクティス活用リファレンスガイド". IPA 独立行政法人 情報処理推進機構. 2013. https://www.ipa.go.jp/files/000026850.doc, (参照 2018-8-10).

角谷信太郎. “デブサミ 2009:パネルディスカッション：テストを行うこと、テストを続けること". 角谷 HTML 化計画. 2009. https://kakutani.com/20090213.html#p01, (参照 2018-8-10).

角谷信太郎. “アジャイル開発者の習慣－
acts_as_agile". gihyo.jp. http://gihyo.jp/dev/serial/01/agile, (参照 2018-8-10).

規世やよい. “XP 再入門". WEB+DB PRESS. 2012, Vol.72, p.125-146, ISBN 978-4774153957.

Glenford J. Myers 他. “ソフトウェア・テストの技法". 松尾 正信訳, 株式会社近代科学社, 第 2 版, 2006, 221p, ISBN 978-4764903296.

佐藤善昭. “ケーススタディ 富士通ソフトウェアテクノロジーズ"アジャイルとトヨタ生産方式が融合した俊敏な開発プロセス"". PM magazine. 2005, vol.003, p.48-50, ISBN 978-4798107707.

Ken Schwaber, Jeff Sutherland. “The Scrum Guide". Scrum Guides. 2017. https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf, (参照 2018-08-24).

Ken Schwaber, Mike Beedle. “アジャイルソフトウェア開発スクラム". スクラム・エバンジェリスト・グループ訳, 株式会社テクノロジックアート編, 長瀬嘉秀監訳, 今野睦監訳, 株式会社ピアソン・エデュケーション, 2003, 183p, ISBN 978-4894715899.

竹内弘高, 野中郁次郎. “The New New Product Development Game". Harvard Business Review. 1986. ISSN 0017-8012. https://hbr.org/1986/01/the-new-new-product-development-game, (参照 2018-8-16).

1.3 スクラムと XP の調和

Kent Beck. “Extreme Programming Explained: Embrace Change". Addison-Wesley Professional, 1999, 224p, ISBN 978-0201616415.

Kent Beck, Cynthia Andres. “Extreme Programming Explained: Embrace Change". 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

株式会社永和システムマネジメント. “ - eXtreme Programming - Embrace Change". オブラブ.http://objectclub.jp/community/XP-jp/xp_relate/essential-ec, (参照 2018-8-10).

神部知明. Agile japan2010 "受託開発におけるアジャイルの取り組み事例". SlideShare.https://www.slideshare.net/slicks/agile-japan2010, (参照 2018-8-22).

大野耐一. “トヨタ生産方式――脱規模の経営をめざして". ダイヤモンド社, 1978, 232p, ISBN 978-4478460016.

Mary Poppendieck, Tom Poppendieck. “Implementing Lean Software Development: From Concept to Cash". Pearson Education, 2006, 304p, ISBN 978-0321437389.

Ken Schwaber, Jeff Sutherland. “The Scrum Guide". Scrum Guides. 2017. https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf, (参照 2018-08-24).

第２章 アジャイル開発の品質管理技法

２.１ 開発プロセスの特徴

（参考文献なし）

２.２ 統計的品質管理の応用

奈良 隆正. “ソフトウェアの品質保証の本質 技法の変遷から学ぶ"日本ソフトウェアテストシンポジウム JaSST2017 招待講演 http://jasst.jp/symposium/jasst17tokyo/pdf/A7.pdf (p.57)

一般財団法人日本科学技術連盟. “統計的品質管理とは". 日科技連. https://www.juse.or.jp/statistical/about/

一般財団法人日本規格協会内品質管理検定センター. “品質管理検定(QC 検定)4 級の手引き". 品質管理検定. https://webdesk.jsa.or.jp/pdf/qc/md_4611.pdf

２.３ アジャイル品質管理技法の検証と考察

（参考文献なし）

第３章 大規模アジャイル開発の可能性

３.１ ウォーターフォールは大規模向きだったのか？

Manifesto for Agile Software Development https://agilemanifesto.org/iso/en/manifesto.html

IPA 独立行政法人 情報処理推進機構 "ソフトウェア産業の実態把握に関する調査" 調査報書 (p.78,79)https://www.ipa.go.jp/files/000026799.pdf

IPA 独立行政法人 情報処理推進機構 "ソフトウェア開発データ白書 2018-1019" https://www.ipa.go.jp/files/000069381.pdf

３.２ アジャイル開発は大規模に耐えられるのか？

The LeSS Company B.V. "LeSS（Large-Scale Scrum）". https://less.works/. (2020-01-10 参照)

門田安弘. “トヨタプロダクションシステム―その理論と体系". ダイヤモンド社, 2006/02, ISBN 978-4478460085.

SAFe® Scaled Agile https://www.scaledagileframework.com/ (2020-02-25 参照)

Scrum@Scale. LLC."Scrum@Scale™". https://www.scrumatscale.com/. (2020-01-10 参照)

Scrum@Scalele™,Spotify のスケーリングアジャイル - 部隊、分隊、支部やギルドと共に歩む . https://www.scrumatscale.com/(2020-03-02 参照)

Scrum.org™. The Nexus™ Guide. https://www.scrum.org/resources/nexus-guide. (2020-01-10 参照)

Ken Schwaber and Jeff Sutherland. "The Scrum Guide™", https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf(2020-02-25 参照)

３.３ 大規模はアジャイル開発の味方

Kent Beck, Cynthia Andres. “Extreme Programming Explained: Embrace Change". 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

第４章 アジャイル開発の原価管理

４.１ フロントローディング

Ken Auer(原著),Roy Miller(原著),平鍋 健児(翻訳),遠藤 真奈美 (翻訳),高嶋 優子 (翻訳),山田 禎一 (翻訳)."XP エクストリームプログラミング適用編―ビジネスで勝つための XP".ピアソン・エデュケーション. 2002/08, ISBN-10:4894715554

ケント・ベック,"XP エクストリームプログラミング入門―ソフトウェア開発の究極の手法". ピアソン・エデュケーション. 2000/12, ISBN-10:489471275X

Alan M. Davis (原著), 松原 友夫 (翻訳)."ソフトウェア開発 201 の鉄則".1996-3-22. 日経 BP.ISBN-10:4822290026

平鍋健児. An Agile Way "ソフトウェア開発とフロントローディング". https://blogs.itmedia.co.jp/hiranabe/2008/01/post-3a7f.html

ジェームズ・M・モーガン,ジェフリー・K・ライカー "トヨタ製品開発システム" .日経 BP. 2007/2, ISBN-10:4822245705

小川正樹."実践原価企画―環境経営に対応した理想ライフサイクルコストの追求".税務経理協会.2001/07. ISBN-10:4419037849

吉田栄介."原価企画能力―持続的競争優位をもたらす" .中央経済社.2003/02.ISBN-10:4502219800

４.２ 工程とコストドライバー

Ken Auer(原著),Roy Miller(原著),平鍋 健児(翻訳),遠藤 真奈美 (翻訳),高嶋 優子 (翻訳),山田 禎一 (翻訳).「XP エクストリームプログラミング適用編―ビジネスで勝つための XP」.ピアソン・エデュケーション. 2002/08, ISBN-10:4894715554

ケント・ベック,"XP エクストリームプログラミング入門―ソフトウェア開発の究極の手法". ピアソン・エデュケーション. 2000/12, ISBN-10:489471275X

Alan M. Davis (原著), 松原 友夫 (翻訳)."ソフトウェア開発 201 の鉄則".1996-3-22. 日経 BP.ISBN-10:4822290026

永和システムマネジメント.平鍋健児."XP：EXtreme Programming：ソフトウェア開発プロセスの新潮流 -前編：XP 概要とその周辺-".2002-04-15.情報処理 43 巻 4 号.書誌レコード ID AN00116625

平鍋健児. An Agile Way "ソフトウェア開発とフロントローディング". https://blogs.itmedia.co.jp/hiranabe/2008/01/post-3a7f.html

日本工業規格. "システム及びソフトウェア製品の品質要求及び評価(SQuaRE)－システム及びソフトウェア品質モデル"JIS X25010：2013(ISO/IEC 25010：2011)

Mary Poppendieck, Tom Poppendieck. “Implementing Lean Software Development: From Concept to Cash”. Pearson Education, 2006, 304p, ISBN 978-0321437389.

和田卓人."質とスピード(2020 春版)". https://speakerdeck.com/twada/quality-and-speed-2020-spring-edition

４.３ 品質とコスト

Kent Beck. “Extreme Programming Explained: Embrace Change". Addison-Wesley Professional, 1999, 224p, ISBN 978-0201616415.

Kent Beck, Cynthia Andres. “Extreme Programming Explained: Embrace Change". 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

Roy Miller. “"XP の真髄"に立ち戻る 第 1 回 XP の誇大宣伝を詳しく検討する". IBM Developer Works. https://www.ibm.com/developerworks/jp/java/library/j-xp0813/ (配信 2002-08-01).

Roy Miller, Christopher Collins(共著). “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 - XP は私達に何をもたらすのか". IBM Developer Works. https://www.ibm.com/developerworks/jp/java/library/j-xp/ (配信 2001-03-01).

門田安弘. “トヨタプロダクションシステム―その理論と体系". ダイヤモンド社, 2006/02, ISBN 978-4478460085.

日本工業規格"システム及びソフトウェア製品の品質要求及び評価(SQuaRE)－システム及びソフトウェア品質モデル"JIS X25010：2013(ISO/IEC 25010：2011)

Wikipedia "Extreme Programming" https://en.wikipedia.org/wiki/Extreme_programming

第５章 アジャイル開発の進捗管理

５.１ バーンダウンチャート

Alistair Cockburn. "アジャイルソフトウェア開発" .株式会社テクノロジックアート訳, 長瀬 嘉秀(監訳), 今野 睦(監訳). ピアソン・エデュケーション. (2002/8/30). 400p. ISBN 978-4894715790

Alistair Cockburn. "Crystal Clear: A Human-Powered Methodology for Small Teams" . Addison-Wesley Professional; 1 版 (2004/10/19). 336p. ISBN 978-0201699470

５.２ 進捗管理と進行管理

Principles behind the Agile Manifesto https://agilemanifesto.org/iso/en/principles.html.

アジャイル宣言の背後にある原則 https://agilemanifesto.org/iso/ja/principles.html.

Kent Beck. "Extreme Programming Explained: Embrace Change". Addison-Wesley Professional, 1999, 224p, ISBN 978-0201616415.

Kent Beck, Cynthia Andres. "Extreme Programming Explained: Embrace Change". 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

Kent Beck, Martin Fowler. "XP エクストリームプログラミング実行計画". 長瀬 嘉秀(翻訳), 飯塚 麻理香(翻訳). ピアソン・エデュケーション. (2001/4/1), 142p. ISBN 978-4894713413

Mike Cohn, "User Stories Applied: For Agile Software Development". Addison-Wesley Professional; 1 版 (2004/3/1). 304p. ISBN 978-0321205681

Ron Jeffries, Ann Anderson, Chet Hendrickson. "XP エクストリームプログラミング導入編 - XP 実践の手引き". 平鍋 健児(翻訳), 高嶋 優子(翻訳), 藤本 聖(翻訳). ピアソン・エデュケーション. (2001/8/10). 348p. ISBN 978-4894714915

中谷宇吉郎. "北海道開発に消えた八百億円". 青空文庫. https://www.aozora.gr.jp/cards/001569/files/57275_61600.html

５.３ 在庫はリードタイム

Mary Poppendieck, Tom Poppendieck. “Implementing Lean Software Development: From Concept to Cash”, Pearson Education, 2006, 304p, ISBN 978-0321437389.

第６章 アジャイル開発の士気管理

６.１ ニコニコカレンダー

Agile alliance, "Niko-niko Calendar", https://www.agilealliance.org/glossary/nikoniko/

トム・デマルコ, ティモシー・リスター, "ピープルウエア 第 2 版 ― ヤル気こそプロジェクト成功の鍵", 松原友夫(訳), 山浦恒央(訳), 日経 BP, 2001, 328p, ISBN 978-4822281106

坂田晶紀, "ニコニコカレンダー", 2005,https://sites.google.com/view/niko-niko-calendar/home/ja

6.2 プラクティスは実践知

Agile alliance, "Niko-niko Calendar", https://www.agilealliance.org/glossary/nikoniko/

Agile alliance, "Agile Practices Timeline", https://www.agilealliance.org/agile101/practices-timeline/

Agile alliance, "Subway Map to Agile Practices", https://www.agilealliance.org/agile101/subway-map-to-agile-practices/

Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley Professional, 1999, 224p, ISBN 978-0201616415

Kent Beck, Cynthia Andres, "Extreme Programming Explained: Embrace Change", 2nd ed., Addison-Wesley Professional, 2004, 224p, ISBN 978-0321278654.

ケントベック, "XP エクストリーム・プログラミング入門―ソフトウェア開発の究極の手法", 長瀬嘉秀(訳), 飯塚麻理香(訳), 永田渉 (訳), ピアソンエデュケーション, 2000, 190p, ISBN 978-4894712751

ケントベック, "XP エクストリーム・プログラミング入門―変化を受け入れる", 長瀬嘉秀(訳), テクノロジックアート(訳), ピアソンエデュケーション, 2005, 189p, ISBN 978-4894716858

ケントベック, シンシアアンドレス, "エクストリームプログラミング", 角征典(訳), オーム社, 第 2 版, 2015, 181p, ISBN 978-4274217623

江渡浩一郎, "パターン、Wiki、XP－時を超えた創造の原則", 技術評論社, 2009, 224p, ISBN 978-4774138978

extremeprogramming mailing list, "Re: "Niko-Niko" Calendar, feedback of team mood", https://groups.io/g/extremeprogramming/topic/39160290#116354

Nicole Forsgren Ph.D., Jez Humble, Gene Kim, "Lean と DevOps の科学 テクノロジーの戦略的活用が組織変革を加速する", 武舎広幸(訳), 武舎るみ (訳), インプレス, 2018, 320p, ISBN 978-4295004905

Ken Schwaber, Jeff Sutherland, "The Scrum Guide", Scrum Guides. 2017. https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf

Glenford J. Myers, Todd M. Thomas, Tom Badgett, Corey Sandler, "ソフトウェア・テストの技法 第2版", 長尾真(訳), 松尾正信(訳), 近代科学社, 2006, 221p, ISBN 978-4764903296

Glenford J. Myers, Corey Sandler, Tom Badgett, Todd M. Thomas, "The Art of Software Testing", Wiley, 2版, 2004, 256p, ISBN 978-0471469124

野中郁次郎, 竹内弘高, "The New New Product Development Game", Harvard Business Review. 1986. ISSN 0017-8012. https://hbr.org/1986/01/the-new-new-product-development-game

坂田晶紀, "ニコニコカレンダー", 2005, https://sites.google.com/view/niko-niko-calendar/home/ja

Sabrina Son, "Why a Niko-Niko Calendar Kills Workplace Morale", https://www.tinypulse.com/blog/sk-niko-niko-calendar-workplace-morale

第7章 「アジャイルなら Git」の神話

濱野純. “入門 Git”. 秀和システム, 2009, 106p, ISBN978-4798023809.

Robert C. Martin. “Clean Agile 基本に立ち戻れ”. 角征典訳, 角谷信太郎訳, アスキードワンゴ, 2020, 142p, ISBN978-4048930741.

Vincent Driessen. “A successful Git branching model » nvie.com”. https://nvie.com/posts/a-successful-git-branching-model/,（参照 2021-5-13）.

執筆（50 音順）

坂田 晶紀（さかた あきのり） *富士通株式会社*

ソフトウェア開発者、中小企業診断士（Registered Management Consultant）
メインフレーム OS、主にシステム資源最適化プログラムの開発に従事。その後、WEB アプリケーション開発業務に携わり、2002 年からアジャイル開発を実践。アジャイルソフトウェア開発の品質管理、開発管理、組織マネジメントを中心に、約 1,000 の職場をコンサルティング。

松村 直哉（まつむら なおや） *富士通株式会社*

CSM（Certified ScrumMaster）、SPC（SAFe Program Consultant）
車載メーター搭載のソフトウェア開発、大規模プロジェクト向け工程管理・課題管理システムの開発に従事。アジャイル開発を修得しつつ、アジャイル人材育成・研修サービスを開発・運営。現在は、社内のアジャイル開発チームの育成、およびお客様企業におけるアジャイル開発の内製化をサポート。

三木 真由子（みつき まゆこ） *富士通株式会社*

CSM（Certified ScrumMaster）
サービスロボット共通通信プロトコルの策定、空間をデジタル化する UI システムの PoC 開発に従事。
アジャイル開発を修得しつつ、アジャイル人材育成・研修サービスを開発・運営。
現在は、お客様企業におけるアジャイル開発を支援するソリューションの企画を担当。

米原 健策（よねはら けんさく） *富士通株式会社*

ソフトウェア開発者、SPC （SAFe Program Consultant）
アプリケーションプログラマとして幅広い業種、多くのプラットフォーム上で動作するソフトウェアを開発。現在はアジャイルコーチとして、内製化を目指すお客様企業に自己組織化支援および技術的支援を実施。

和田 隆司（わだ たかし） *富士通株式会社*

ソフトウェア開発者、SPC（SAFe Program Consultant）、CSM（Certified ScrumMaster）
2006 年に業務掛け持ちの改善のために取り組み始めた「かんばん」、「リーン開発」をきっかけとし、Web アプリケーション開発業務を中心にアジャイル開発を実践。
加えて現在は、新規ビジネス検証部門をはじめ、様々なチームの立ち上げにおけるアジャイル開発の技術支援にも従事している。

レビュー（50 音順）

石元 達也（いしもと たつや） *富士通株式会社*

大塚 憲（おおつか あきら） *富士通株式会社*

神部 知明（かんべ ともあき） *富士通株式会社*

俵谷 健太郎（ひょうたに けんたろう） *富士通株式会社*

村松 正浩（むらまつ まさひろ） *富士通株式会社*

編集

淡島 政紀（あわしま まさのり） *富士通株式会社*

富士通株式会社

ジャパン・グローバルゲートウェイ

お問い合わせ

https://fujitsu.com/jp/services/agile/