リリース頻度を向上したアジャイル開発への段階的移行に関する一考察

A Study on the Gradual Transition to Agile Development with Increased Release Frequency

研 究 員：上野彩子（個人）
山口 信二郎（エヌ・ティ・ティ・コミュニケーションズ株式会社）
主　査 ：喜多 義弘 （長崎県立大学）
副 主 査：上田 和樹（日本ナレッジ株式会社）

研究概要
プロダクトの顧客価値向上を図るため，既存の開発手法からアジャイル開発へ切り替える企業が増えている．しかし，アジャイル開発はウォーターフォール開発とは顧客との接点の持ち方や開発の進め方が大きく違うため実践が難しく，誤った取り組みをしてしまう企業は多い．高いリリース頻度とフィードバックに基づき，プロダクトの価値を高める取り組みを実践する為には，組織や社内ルールなどの文化慣習の変革，チームメンバーの技術力向上，およびアジャイル開発への理解の促進を推進する必要があるが，急激な変革を推進することは大きな困難が伴う．
本論文では，ウォーターフォール開発に慣れたチームがアジャイル開発に移行する際に障壁となる要因を分析する．また，ウォーターフォール開発に慣れたチームが陥りがちな，誤ったアジャイルパターンを定義し，理想的なアジャイルと比較を行うことで問題点を明確化する．その上で，誤ったアジャイルパターンから理想的なアジャイルに移行する際の障壁を緩和し，誤ったアジャイルパターンから脱却する為の中間解を提案する．

Abstract An increasing number of companies are switching from existing development methods to agile development to improve the customer value of their products. However, agile development is difficult to implement because it differs greatly from waterfall development in the way it interacts with customers and the way development proceeds, and many companies are taking the wrong approach. To implement initiatives to increase the value of products based on high release frequency and feedback, it is necessary to change cultural practices such as the organization and internal rules, improve the technical skills of team members, and promote understanding of agile development, but promoting rapid change is very difficult.
In this paper, we analyze the causes of agile development for teams accustomed to waterfall development. We also define the incorrect agile pattern that teams accustomed to waterfall development tend to fall into, and clarify the problems by comparing them with the ideal agile development. Then, we propose an intermediate solution to ease the barriers in moving from the incorrect agile to the ideal agile, and to break free from the incorrect agile pattern.

1 はじめに

クラウドサービスの普及やユーザーニーズの多様化に伴い，売り切りのプロダクトを開発するときよりも，ユーザーの深い理解とユーザー体験の設計が重要視されるようになってきている[1]．ソフトウェア開発手法も，これを実現するための手法を選択する必要がある．
ウォーターフォール開発では，要件定義，設計，および実装を経て，プロダクトのリリースまでに数ヶ月から数年を要することもあるが，プロダクトがリリースできるころには，実装された要件はすでに社会が求めるものではなくなっていることも多い．
そのため，プロダクトを素早くエンドユーザーに届けフィードバックを得ることで，エンドユーザーが求めるプロダクトに近づけていく．その手法の一つとしてアジャイル開発[2]や DevOps の考え方[3]が知られており，それらの実践が実際に企業のビジネス競争力を高めていることが知られている[4]．具体的に，プロダクトのリリース頻度が高い企業のほうがパフォーマンスがよいという定量データがある[4]．このような社会的背景から，日本企業でも競争力を高めるためにアジャイル開発に取り組む企業が増えている．

しかし，ウォーターフォール開発に慣れている企業や組織でアジャイル開発を推進することは大きな困難が伴う．ウォーターフォール開発とアジャイル開発では顧客価値向上に関する取り組み方が異なっているため，理解や実践が非常に難しいことが原因の一つとして考えられる．事例としては，計画ミーティングやふりかえりなどのプラクティスの導入が目的になりリリース頻度があがらない，リリースまでのリードタイムを短縮するにもかかわらずプロダクトの開発スコープが変わらないためリソース不足から品質の低下を招く，などが挙げられる．

本論文では，そのような組織がアジャイルに移行する際に障壁となる要因を整理する．また，そのような組織が陥りがちな誤ったアジャイルパターンを定義し，理想的なアジャイルと比較を行う事で，主にリリース頻度とフィードバックの観点から問題点を明確化する．その上で，誤ったアジャイルパターンから理想的なアジャイルに移行する際の障壁を緩和し，誤ったアジャイルパターンから脱却する為の中間解を提案する．

2 アジャイル開発導入の障壁となる要因

アジャイルの導入を試みたものの，理想的なアジャイル開発が実践できていない企業が存在すると考えられる． PMI の調査［7］によると，

- 約 20%の回答者がアジャイル内でウォーターフォールのプロセスを導入している
- ユニットテスト自動化の導入率が約 20%にとどまる
- イテレーションの期間が 1 ヶ月を超えるか，期間を明確に定めていない

など，理想的なアジャイルのベストプラクティスを実践出来ていないケースがある．このようなチームでは高いリリース頻度の達成が難しく，早いフィードバックに基づいたビジネスニーズの理解および追従が難しいと考えられる［4,8］．

このような状態に陥る主な原因を以下の様に分類した．なお，本論文におけるアジャイル開発は，スクラム[9]に限定する．

(1) 組織や社内ルールに起因する問題
(2) 個人の技術不足
(3) アジャイルのコンセプトの理解不足

各要因について詳細に取り上げる．

2.1 組織や社内ルールに起因する問題

2.1.1 承認行為によるリードタイムの延伸

これまでの組織構造や社内ルールなどがアジャイル開発に適しておらず，アジャイル開発の導入や実践が難しいという問題が考えられる．

例えば，評価会，出荷判定会議を実施しないとリリースできず，その場に提出するテストのエビデンスが必要であるため，会議の前に必ずテストを実施するといった事例がある．出荷判定会議のようなチェックポイントが存在すると，承認が得られない限りリリースできないため，結果としてリリースまでのリードタイムを短縮できない．

2.1.2 契約形態に依存する問題

アジャイル開発ではチームのメンバーを固定して開発を進めることが理想とされている［1］が，組織によっては社内リソースでの固定チームをもっていないことがある．その場合，他社開発ベンダーに作業を委託することになる．ベンダーに委託する利点としては，専門性の高い人材を集められること，社内リソースの一部を他社開発ベンダーにアウトソースすることで，社内の人件費における固定費の比率を下げられること，などが挙げられる．一方で欠点としては，開発スコープの変更に対応したり，チーム間のゴールの共有やコミュニケーションを密に取ったりしながら進めていくアプローチが契約形態によっては難しい場合がある[5]．また，1つのチームとして機能しなければならないにも関わらず，自社と他社ベンダーでサイロ化しやすい傾向があるため，チームの中で知見および技術の共有や底上げが難しい問題が発生することがある．

仮に社内リソースでの固定チームを持っていたとしても，日本は IT 投資がアメリカと比較して盛んではなく，IT 人材の育成や開発チームの維持を固定費の負担増と捉えているところから技術力の高いチームが育たないという指摘もある[1]．

2.1.3 組織の構造に依存する問題

組織織構造が職能別に分かれており，一つの開発チームとして動けない場合，リリース頻度の低下や，品質の低下を招く可能性がある．たとえば，デプロイメントを行うチームが開発チームの外にあり，開発チームがデプロイメントの権限を持っていない場合，リリース頻度は明らかに低下する．QA チームが開発チームの外にあり，開発が完了次第まとまったテストのフェーズを設けてテストを実施する，といったケースも同様である．

この問題は開発チームに必要な権限が委譲されていないという点において，2.1.1 の承認行為とも密接に関連している．それだけではなく，担当者がプロダクトに必ずしも精通していないため，開発チームに同様のスキルを持ったメンバーが参画していれば指摘できたような仕様の漏れや手順のミスに気づかず，結果として不具合を引き起こす可能性もある．

2.2 個人の技術不足

アジャイル開発においてリリース頻度を高めていくためには，開発チームが正しく要求を理解すること，およびテストで不具合を見つけて修正するのではなく，開発中から品質の高い成果物作成していくことが重要である．これを実現するための代表的な開発手法として，テスト駆動開発（Test-Driven Development, TDD）がよく知られている．TDD は最初にテストコードを書き，テストが通るようにプロダクトコードを書いていくことから，要求仕様を正しくテストコードに表現することができれば非常に品質の高いプロダクトコードを生成することができる．TDD のほかにも，変更容易性の高いアーキテクチャーの選定，必要なテスト環境をすぐに構築できるインフラ技術，および変更による不具合がないことを素早く確かめるためのテスト自動化は，リリース頻度を高めるために必要不可欠である．

ところが，ウォーターフォール開発に慣れたデベロッパーは，こういった手法を知らなかったり，実践した経験がなかったりする場合がある．個人がこのような技術を活用できない場合，手戻りが発生したり，品質が低下したりすることで，リリース頻度が下がる一因となると考えられる．

2.3 アジャイルのコンセプトの理解不足

2.3.1 ユーザーストーリーに対する理解不足

ウォーターフォール開発におけるデベロッパー視点での機能の分割整理に慣れている場合，ユーザーストーリーをウォーターフォール開発の際と同様に機能のまとまりとして捉える傾向がある．理想的には「デベロッパー視点で分割した 1 つの機能」ではなく，「1 つのユーザー体験として成立する最小単位」としてユーザーストーリーを作成し，開発の粒度を細かくすることでリリース頻度をあげ，素早くフィードバックを得ることを目指すべきである．ところが，「デベロッパー視点で分割した 1 つの機能」としてユーザーストーリーを作成した場合，1 つのユーザーストーリーの開発が完了したとしてもユーザー体験を満足するプロダクトができないため，リリース頻度を高くすることができない．

2.3.2 製造業をベースとする品質保証の考え方とアジャイルのコンセプトとの相違

製造業をベースとした品質保証を実施している企業は，アジャイル開発における品質の考え方が馴染まない可能性がある[1]．そうした製造業をベースとした企業の開発プロジェクトでは，リリース後にプロダクトに変更を加えることが難しいため，変更を前提としない設計に基づき厳しい品質管理をクリアした．そのため素早くリリースしようとしてもプロセスの軽量化が難しかったり，プロセスの軽量化により品質低下への懸念が出てきたりとある．スプリント中にチーム内で行ったテストだけではプロダクトの品質が心配であるため最後にシステムテストのような工程を設けていることもある．テスト工程がボトルネックとなり，ウォーターフォール開発と同様，リリースまでのリードタイムを短縮できない．

2.3.3 アジャイルテストの理解不足

アジャイル開発におけるテストについて提唱されている考え方がウォーターフォール開発のテストの考え方と異なっており，理解や実践の妨げになっている可能性もある．例えば，アジャイル開発におけるテストのベストプラクティスを示した「アジャイルテストの 4 象限」という考え方がある[6]．これは，各ユーザーストーリーに対し適切なテスト活動を実施するために，テストタイプを「ビジネス面/技術面」「製品を批評する/チームを支援する」という軸で分類したものである．ここで重要なことは，テスト活動を工程（テストレベ

ル）別に分けていないことである．これは，各ユーザーストーリーに対し適切なテストタイプとその順序をチームが決定することを目的とするためである．ところが，ウォーターフォール開発に慣れている組織では，テストレベルからテスト活動の実施タイミングを定義することがあるため，この考え方は時間軸を定義していない点で異なっており，理解や実践が難しいと考えられる．

3 陥りがちな誤ったアジャイルパターン

前章で論じた例に限らず，本来のアジャイルの目的を達成するのが困難になっているアジャイルを「似非アジャイル」と定義する．似非アジャイルは様々なパターンが考えられるが，本論文では 1 つの代表例として，付録表 1 で表される，ウォーターフォールをベースとしたパターンを定義する．

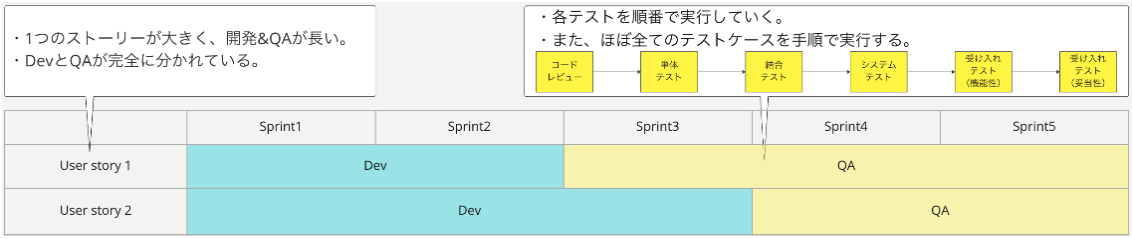

図 1 似非アジャイルの全体像

図 2 理想的なアジャイルの全体像

付録表 1 似非アジャイルと理想的なアジャイルの比較 は，似非アジャイルと理想的なアジャイルの比較である．各項目には分類名を付与しており，項目の概要や目的を表現している．付録表 3 比較表の各項目と，対応する問題・分類の対応表 では，付録表 1 の各項目が，2 章で述べたどの原因に対応しているかを記載している．また，原因に対応していない項目については分類名を記載している．

似非アジャイルは開発フェーズと QA フェーズが明確に分かれており，テストフェーズに入るまでテスト実施を行わない．また，テストフェーズでは各テストレベルを順に実施する．一方，理想的なアジャイルではアジャイルテストの 4 章限の中から必要なテストを選び，開発と同時に行うことで，バグの発見タイミングを早期化している．バグの発見の遅れはバグ修正コストを指数関数的に増加させる[8]ため，似非アジャイルはバグ修正に時間を要することが考えられる．

システムアーキテクチャーに関して，似非アジャイルは小さなユーザーストーリーを並行して開発する事を前提としておらず，複数コンポーネントが絡み合った設計となりやすい．その場合開発領域が区分けしづらいため，複数人開発時にはコード管理システムでコンフリクトが発生しやすい．また，一部のコンポーネントに変更があると，それを利用する他のコンポーネントのテストも全て変更する事になる．保守に関しても，バグの箇所の特定が難しく，コード変更に伴う影響範囲が大きくなりやすいので修正がしづらいといった事態に繋がる．一方，理想的なアジャ

イルでは小さなユーザーストーリーの並行開発を意識し，適切にコンポーネントが独立しているため，開発，テスト，デプロイ，保守，運用がやりやすい．ユーザーストーリーのサイズに関して，似非アジャイルでは複数の機能を盛り込んだストーリーを作るため，サイズが大きくなる傾向がある．これにより，理想的なアジャイルと比較して 1 つのストーリー開発を完了させるのに時間を要する．

付録表 1 で挙げているその他の項目の内容も似非アジャイルにおける開発・QA スピード低下に影響し，1 つのストーリーの完成に必要なスプリント数が増大していくと考えられる．結果的に，”チームのパフォーマンス計測の指標”分類に挙げているように，リリース頻度が遅くなるという事態に繋がる．図 2 のように，2 スプリントを終えた時点で理想的なアジャイルでは 6 個のユーザーストーリーが完了しているのに対して，図 1 のように，似非アジャイルでは完了しているユーザーストーリーがない．これは，早いリリースに基づいたフィードバックをもらう頻度および機会の減少を意味する．

また，アジャイルを始める段階で全てのストーリーを作成することや，開発チームを維持しようとしないことなど，フィードバックを貰うことを前提とせずに開発を進行するため，本来のアジャイルの目的を果たすことが出来ない．

4 提案手法

これまで論じてきた通り，似非アジャイルと理想的なアジャイルは大きな差がある．似非アジャイルを行っているチームが理想的なアジャイルを目指したいとしても，短期間で理想状態を目指すことは難しい．

そこで，似非アジャイルから段階的に理想的なアジャイルへ近づくためのアジャイル手法として，「中間解アジャイル」を付録表 2 似非アジャイルと中間解アジャイル、理想的なアジャイルの比較のように定義し，提案する．

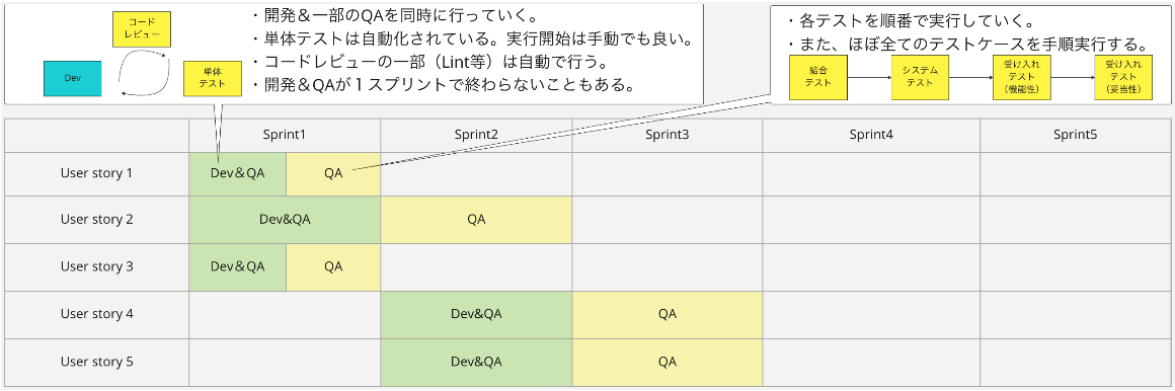

図 3 中間解アジャイルの全体像

中間解では，理想的なアジャイルを目指す上でハードルとなる，組織の意思決定プロセス，社内ルールや他社ベンダーとの契約に起因する項目を出来るだけ回避しつつ，アジャイルの目的である，リリース頻度を上げるための取り組みと，顧客のビジネスニーズに追従するための取り組みを定義している．一部の項目については最初から理想的なアジャイルを目指すことが難しいため，似非アジャイルと同じ内容になっているが，多くの項目が似非アジャイルよりも改善されている．

“単体テスト実施タイミングと頻度”では，中間解は開発と同時に単体テストを何度も実行する．これにより，前章で述べたように，指数関数的に増大するバグ修正コストを低く抑えることで開発スピードの向上に繋がる．

また，単体テストは自動化しているため，変更開発の際のリグレッションテストが高速化される．これは継続的な開発を前提としているアジャイルにおいて大きなアドバンテージとなる．一方，自動単体テストを実施していなかったチームのデベロッパー全員が自動単体テストを導入するのはハードルが高い．そこで”QA 担当者の役割”において，QA 担当者はデベロッパーを育成し，また，テスト環境を整備することで，単体テストをデベロッパーが実施できるようにする役

割と定義している．なお，そのためには QA 担当者自身が単体テストのコーディングスキルが必要である．

“結合テスト実施タイミングと頻度”では，中間解はコンポーネントの IF 結合仕様を満たすように Mock & Stub を利用したテストを開発と同時に何度も実施する．これにより，単体テストと同様に，開発スピードの向上が期待される．Mock & Stub だけでなく，実際のコンポーネントとの結合試験を開発と同時に実施するのが理想ではあるが，“開発・テスト環境”で述べているように，各のローカル環境にシステムの各コンポーネントをデプロイするのは難しいと考えられる．その理由として，対象となる組織またはチームの成熟度が低い場合，各コンポーネントをコンテナ技術などで仮想化し，全コンポーネントを商用と同一のバージョンでローカル環境に構築する事の技術的な難易度が高いことが挙げられる．また，有償コンポーネントのライセンスのコストの問題も挙げられる．コストは組織の問題であるため，それを回避するためにも Mock & Stub による QA を実施するのが良いと考えられる．その上で，実際の結合対象コンポーネントとの結合テストは，開発フェーズが終わった後に専用の環境で実施する．

“システムアーキテクチャー”では，コンポーネントの独立性を比較的高くすべきとしているが，これは複数のデベロッパーが並行してそれぞれ小さなユーザーストーリーを開発していく際に重要となる．逆に，ユーザーストーリーのコードをマージした際にコンフリクトが発生しないようなコンポーネント独立性を目指すと，良いアーキテクチャーに繋がると考えられる．

“ユーザーストーリー作成プロセス”では，フィードバックを基にしてユーザーストーリーを作るという方法を組み込んでおり，アジャイルチームが継続的なプロダクト改善を行うことを理想としている．しかし，対象となる組織またはチームの成熟度が低い場合，完全なフィードバックサイクルを組むことは難しい．そこで中間解では，フィードバックサイクルを組むことよりも，フィードバックを促しながら継続的なプロダクト改善を行うことを目指す．これにより，従来よりも市場のニーズに追従したプロダクト開発に繋がることが期待できる．

また，ユーザーストーリー作成にはデベロッパーも出来るだけ参加するようにする．トップダウンで決められたユーザーストーリーを開発する場合と異なり，デベロッパー自身もユーザーストーリーの作成に関わることで，プロダクトオーナーだけでなく，デベロッパーもプロダクトのビジネス価値を意識し，チームとしてのアウトプットを意識するようになる．それにより，ユーザーストーリーの優先度付けにおいてはチーム内での議論やコミュニケーションが活発化し，プロダクトオーナー1 人が優先度付けするよりも，優れた判断ができることが見込まれる．また，デベロッパーがユーザーストーリーの作成段階から参画することで，技術的観点から無理な仕様が無いかを早期に議論したり，受け入れ試験の観点からユーザーストーリーの完成の定義の精緻化を行ったりすることで，ユーザーストーリーの品質が向上することも考えられる．更に，デベロッパーがユーザーストーリーの内容をよく理解する事で，ユーザーストーリーの認識齟齬による，受け入れテストの手戻りの発生を防ぐ効果も期待される．

5 考察

5.1 中間解を用いることで得られると考えられる効果

ウォーターフォール開発をベースとしている似非アジャイルから理想的なアジャイル開発へ移行するための中間解を提示してきた．この中間解を目指すことで，似非アジャイルに比べてリリース頻度が上がり，市場のニーズを反映した開発が実現できる可能性を示した．ウォーターフォール開発に馴染みのある会社や組織が移行するうえで導入障壁が高くならないよう、既存組織のルールや慣習などを大きく変えなくても実現しやすい内容にしている．

中間解では，特にリリース頻度を重要視している．スクラムガイドではリリース頻度についての言及はない[9]．また実際のプロダクト開発ではプロダクトのインパクトやユーザー影響を考慮して，数スプリントのインクリメントをまとめてリリースをすることもある．しかし，リリース頻度の高さとハイパフォーマンスな組織であることに正の相関関係があるため，本論文ではスプリント終了後に毎回必ずリリースを行うことを推奨することとしている．

中間解アジャイルは理想的なアジャイルに比べて各項目でまだ差異があり，結果としてリリース頻度は理想的なアジャイルよりは低くなることが予想されるが，似非アジャイルよりは高まると考えられる．また，フィードバックを前提とした，ビジネス価値のある開発をするための仕組みが組み込まれており，似非アジャイルに比べて市場のニーズに適合した成果物が出来ると考えら

れる．

5.2 中間解の活用方法

中間解がうまく浸透しているかを確認する指標として，”ユーザーストーリーの完成にかかる期間”や”リリース頻度”の目標を立てる．これが達成できていない場合は，ユーザーストーリーのサイズが肥大していることが考えられるし，テスト活動や開発品質に問題が発生してリリース頻度が下がっていることも考えられる．指標を管理することで継続的にチームのアジャイル手法のどこに問題があるかを考えるきっかけとなる．

5.3 制約事項と今後の展望

本論文では中間解から理想的なアジャイル開発への移行については論じておらず，この点は今後の課題である．特に，理想的なアジャイル開発へ移行するためには組織の意思決定プロセスや社内ルール，契約などの制約に対応したほかの指標を取り入れることが必要であると考えられる．

しかし一方で，中間解を実践しビジネスニーズを捉えた開発へ段階的に移行することは，経営層に対しても良いアピールとなり，中間解では対応していない制約に対しても会社として改善を検討する可能性を高める効果も期待される．

6 おわりに

本論文では，ウォーターフォール開発に慣れたチームがアジャイルに移行するために，徐々に理想的なアジャイル開発に近づけていくための一つの中間解を提案した．中間解を目指すことにより，リリース頻度を徐々に高めながら市場からの早いフィードバックを取り入れる開発ができる可能性について考察した．

開発・テスト技術および開発プロセス改善の観点だけでなく，組織の意思決定プロセスや社内ルールなどの観点からアジャイル移行を考察する必要があり，この点は今後の課題である．

参考文献

1. 及川卓也，“ソフトウェア・ファースト”，日経 BP，2019.
2. Kent Beck et al.，アジャイルソフトウェア開発宣言，https://agilemanifesto.org/iso/ja/manifesto.html
3. Jennifer Davis, Ryn Daniels, “Effective DevOps”, O’Relly, 2018.
4. Nicole Forsgren, 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―”, impress, 2018.
5. 独立行政法人情報処理推進機構 社会基盤センター，“アジャイル開発版 情報システム・モデル取引・契約書 ユーザ／ベンダ間の緊密な協働によるシステム開発で，DX を推進”，https://www.ipa.go.jp/ikc/reports/20200331_1.html.
6. Janet Gregory (著), Lisa Crispin (著), “実践アジャイルテスト テスターとアジャイルチームのための実践ガイド”, 翔泳社, 2009.
7. PMI 日本支部アジャイル研究会 , 2021 年度「アジャイルプロジェクトの実態」に関するアンケート，https://www.pmi-japan.org/topics/PMI_Japan_Chapter_Agile_Survey_2021.pdf.
8. Caper Jones, “Applied Software Measurement: Global Analysis of Productivity and Quality”, O’Relly, 2008.
9. Ken Schwaber & Jeff Sutherland, スクラムガイド，https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Japanese.pdf.