【JUAS】基幹系システムアジャイル適用研究会

2022/4/13

（はじめに）研究会立上げの背景

DX時代の到来が叫ばれる中、日本企業ではなかなか進まないことが問題視されている。また、技術革新も目覚しく、システム開発PJそのものも多様化してきている。それは、これまでの古き良き日本が発展するのに適していた日本式SIerモデルによるシステム開発の限界が近づいているのかもしれない。

そういった中、ひとつの解として「アジャイル開発」が考えられる。昨今では、アジャイル開発もかなり浸透してきているようであり、日本でも実用例が増えてきている。がしかし、大規模／基幹系での適用事例はあまり見られない。日本式SIerモデルの限界を問題視するなら、大規模／基幹系を無視できないと考える。

このような背景により、本研究会を立ち上げるに至った。基幹系システムにアジャイルを適用するにはどうしたらいいか、注意する点は何か、どのように適用することが望ましいか、、等について考察する。

目次

１．基幹系システムとは？

２．アジャイルを適用するメリットとハードル

３．ハードルを越えるために　～マネジメント～

４．ハードルを越えるために　～組織～

５．アジャイルに適したプロジェクト

１－１　基幹系システムの定義

まず初めに、基幹系システムについて定義しておく。
- その会社のビジネスの根幹となる業務を担う重要なシステム
- システムが停止した場合、影響が広範囲にわたり、その企業の活動に大きな支障をきたすもの
  ⇒ **高い可用性、堅牢性、安定性（品質）が求められる**。
- 企業経営に必要な重要かつ機密性の高いデータを管理している。
  ⇒ **高いセキュリティが求められる**。

中でも、今回の研究テーマとしては、以下の表の中の「根幹業務システム」を想定する。

種類		独自要件	要件変更	PKG適合	特徴
会計系・人事系	財務会計、予実管理	小	小	○	・独自要件、要件変更は比較的少ない（会計系よりも人事系がやや多い） ・法令・法制と密接に結びつくため高い品質と安定性が必須要件となる ⇒法律と結びつくため下段と比べて独自色は薄く、パッケージを利用するケースが多い
	人事管理、労務管理、厚生管理	中	中	○	
根幹業務システム（業態により異なる）	製造業、建設業 例）販売管理・在庫管理 生産管理・工事管理	大	大	△	・企業ごとの独自要件が多い ・業務に結びついているため高い品質と安定性およびレスポンスの速さなどが求められる ・独自ロジックなどが多く、継承性に難がある ⇒企業の競争力の源泉となるシステム
	金融業、サービス業 例）勘定系、保険販売、契約管理等				

１－２　基幹系システムが置かれているシチュエーション

基幹系システムは、その会社の根幹業務を担うものであることから、多大な投資を継続して行っており、簡単に変更や代替ができるものではない。
多かれ少なかれ以下のような状況に陥ることが多いと思われ、それが**新しいものへの取り組みの難しさ、代替システムへの変更の難しさ、機動力の無さ**、、等々につながっていると考える。

- **一度完成されたシステムは長期に渡ってそのまま使われ続ける**ことが多い。
  （数年あるいは10年以上稼働しているシステムが多い）
- **開発はWF**が主流。開発期間は様々であるが、半年～**２年かける開発PRJも**いまだ健在。
  - 開発関係者（ビジネスサイド、システム開発担当（ベンダ含む））にアジャイル経験者は少ない。
  - 開発期間が長いため、プロジェクト期間中における要件変更はそれなりにある。
  - 利用開始後の改良が行われることは少なく、**最初から十分な品質・完成度を要求**される。
- 昨今の事業環境の変化により、品質をトレードオフにすることなく、プロジェクト期間短縮などが要求されている。
- システム構造として、メンテを繰り返す過程ですでに疎結合にはできておらず、**各機能は密結合**に近い。
- 新しいテクノロジーを加えにくく、自動化できる部分は限られていることが多い。

１．基幹系システムとは？

２．アジャイルを適用するメリットとハードル

３．ハードルを越えるために　～マネジメント～

４．ハードルを越えるために　～組織～

５．アジャイルに適したプロジェクト

2－1　アジャイルを基幹系に適用するメリット

基幹系システムにアジャイルを適用するメリットは確かにある。WFと比べ、以下のような点について、メリットを享受できると考えられる。（詳細な分析は、別紙３参照）

	アジャイル開発のメリット	基幹系での享受（WF比）	
1	仕様バグの低減	○	・WFでは後半のユーザーテストで顕在化する要件バグが、ユーザーとのコミュニケーション、上流工程からの参画により早い段階で摘出可能
2	ドキュメントの簡素化、質の向上	○	・コミュニケーションやソースコードで理解することに重点を置くことで、結果的にドキュメントが簡素化 ・コミュニケーションの積み重ねによる質の向上
3	プロダクトの価値の向上（優先順位の最新化、要件変更に迅速に対応）	○	・開発期間が長い基幹系の場合は、要件の陳腐化が少なくなる（その時点で優先度の高いものを開発可能）
4	開発生産性の向上	△	・同じメンバーが継続的に開発に携わることで、学習効果により開発生産性は段階的に向上 ・WFに比べてアイドリング状態となる期間が少ない
5	サービスインまでの期間短縮 （優先度付けにより短期間でリリース）	△	・基幹系は密結合状態となっているシステムが多く、リリースに向けて相応のテストが必要となるケースが多いため、短縮効果を得るにはテスト自動化などの施策を並行して実施する必要がある。 一方で、独立した機能を改修するケースや疎結合化されたシステムに適用する場合は、短縮メリットを享受しやすい。
6	対面でのコミュニケーションによりコミュニケーションコストが低減	△	・要件定義などの手戻りコスト、リスクの減少 ・ウォーターフォールの場合に発生しがちなチーム間連携でのミス等のリスクを下げられる（ただし、規模が大きい大人数のプロジェクトでは難しい部分も）

○：基幹系でも概ねメリットを享受できる　△：基幹系でもケースによってはメリットを享受できる（効果があまり出ないケースもある）
×：基幹系の場合、WFとあまり差は出ない

2－2　アジャイルを基幹系に適用するときのハードル

一方で、アジャイルを適用するには乗り越えるべきハードルも相応にある。特に基幹系の場合は、以下の点についてハードルになることが多い。
次のスライドから、この乗り越えるべきハードルについて考察していく。

	アジャイル適用時のハードル	基幹系での影響（WF比）		
1	品質管理の難しさ	▲	・短期でのサービスインにこだわった場合で、テスト不足がある場合に顕在化 ⇒テスト期間の確保及びテスト実施方法の工夫、さらにはメンバーがアジャイル開発における開発の仕方や品質確保の考え方をきちんと共有することが必要	**マネジメントの考察**
2	コスト管理・スケジュール管理の難しさ	▲	・要件変更が多い場合はコストやスケジュールへの影響が生じる	
3	維持に向けた引継ぎの難しさ	▲	・ドキュメントの不足により、仕様の背景や経緯の引継ぎが難しくなる ⇒ドキュメントの工夫により、一定の解消は可能	
4	チームビルディングの難しさ（多人数での開発に適していない）	×	・コアとなる人材の不足（特に規模が大きな場合） ・基幹系のベンダはアジャイル経験が不足	**組織の考察**

※　×：基幹系開発の場合、ハードルが高い　▲：ハードルではあるが対策により解消可能　〇：あまり差はない

１．基幹系システムとは？

２．アジャイルを適用するメリットとハードル

３．ハードルを越えるために　～マネジメント～

４．ハードルを越えるために　～組織～

５．アジャイルに適したプロジェクト

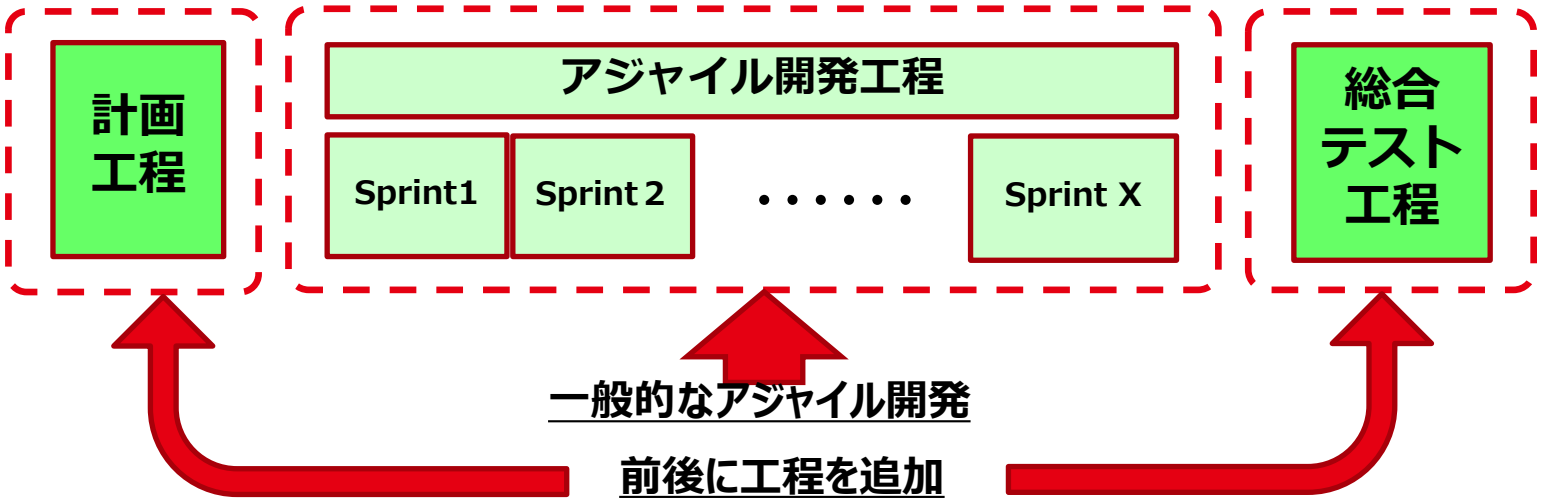
３－１　ハードルを越えるためのマネジメント　基本的な考え方

アジャイルのプロジェクトは難しいと言われるが、基幹系に適用する場合には、通常のプロジェクト以上に難しいと考える。基幹系に適用するためには品質の確保など、クリアすべきハードルがある。QCDのバランスを考えたプロジェクトの計画、マネジメント方法について考察する。

※右下のようなシチュエーションを想定

●基幹系システムと密接に関連するサブシステムの開発のため、一般的なアジャイルほどの自由度で開発することは難しく、**事前にある程度の要件や計画を検討する**ことが望ましいと考える。
●基幹系システムへの影響を考えると、次々とリリースを繰り返すようなことも難しいことから、アジャイル開発後に、**全体を通した総合テストを実施**することが望ましいと考える。
●逆に、できるだけアジャイルの利点を失くさないためにも、アジャイル開発工程は一般的なアジャイルに。

➡ **アジャイル開発工程の前後に工程を加えた形**が望ましいのではないか。



＜想定されるシチュエーション＞

●高品質が求められるが、競争力向上のため、独自性とスピードも求められてきている。
●そういった背景の中、**新商品**（新サービス、新製品、新事業、新機能・・）**を、別途サブシステムとして構築**することとなった。
●これまでの考え方では、商品開発も既存の基幹系の中で構築してきたところであるが、今回、別途サブシステム化し、**必要な部分のみ既存基幹系と連携**することとした。
●新商品の仕様は、完全には固めることができない。そのため、この**サブシステムの開発をアジャイルで**行いたい。
●連携部分の開発として、**既存の基幹系にもある程度の改修**が必要となる。

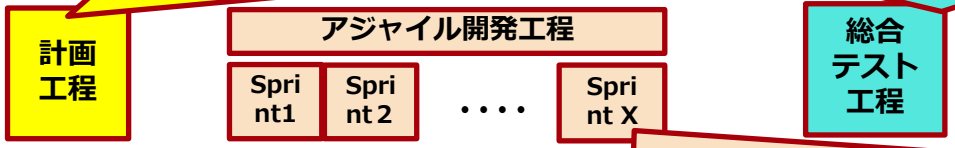
3－2　各工程のポイント

＜計画工程のポイント＞

- 開発工程がぶれすぎないように要件の概要を決めるが、一般的な**WFの要件定義工程レベルには定めない**。
- 全体のデザイン、プロジェクト計画を策定する。
  （制約・前提、基幹システム側のマイルストン、デリバリ計画、、）
- 要件変更を想定し、ある程度の**バッファSprintを設けておく**のが望ましい。
- **基幹系との連携部分は、優先度を上げて開発**するよう計画すべき。
  （早期のSprintで開発）
  - ※　基幹系システム側のデッドライン等を決めておくのが望ましい。
  - ※　接続先とのHUBになるような周辺システムで吸収するという方法もあり。
- 基幹系との構築範囲、接続方式、テストスコープ、テスト開始基準を明確化することが望ましい。
- 基盤の構築スケジュール（開発環境、本番環境など）も決めておくことが望ましい。

＜総合テスト工程のポイント＞

- 基幹系と連動したテストを実施し、全体的な整合性確認含め、基幹系に耐えうる品質を確保。
- 非機能系、接続テスト系は、アジャイル開発工程で先行しておくと、なお望ましい。
- 稼働後の運用を意識した確認を実施しておくことが望ましい。（変更管理プロセス、インシデント対応プロセス、など）
- 発見されるバグの性質も、WFの総合テストのそれとの違いを意識する。
  - －　WFと比べ、要件認識齟齬、要件漏れ等は少ない傾向。
  - －　単体、結合のすり抜け、デグレが多くなりがちだが、開発工程でのテスト自動化次第。



※その他、詳細のポイントは別紙４参照。

＜アジャイル開発工程のポイント＞

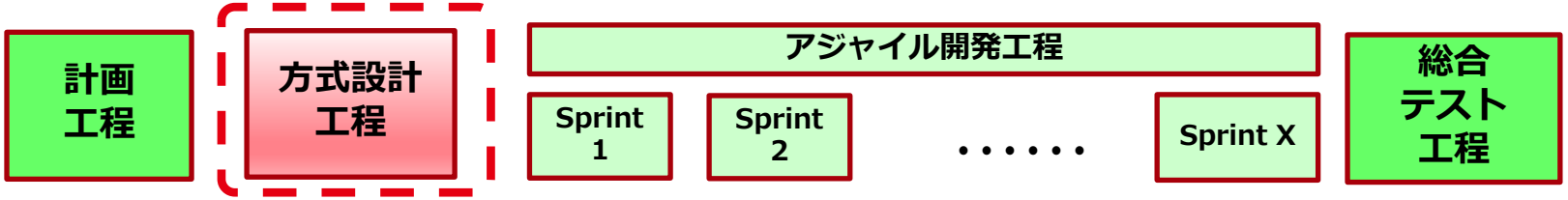
＝　基本的に**一般的なアジャイルをイメージ**

- テストは自動化（どんどん追加）。並行してドキュメントも作成。
- **この工程が終わる段階で、①詳細化した要件、②ドキュメント、③実装、の整合性を取りつつ完成**させることが望ましい。
- アジャイル開発工程までは自由に要件変更可能とするも、アジャイル開発工程完了時に要件はFIXし、それ以降は通常の変更管理。
- 基幹系システムとの接続仕様に絡むところは、先行Sprintで開発（要件FIX）することが望ましい。
- バーンダウンチャート等を用い、開発の進捗状況や生産性を常に測っておくことが望ましい。中でも、基幹系との連携部分にかかる機能は、バックログの段階から区別しておくことが望ましい。（変更要求があった際に影響を意識する）
- 基幹系も意識した総合テストのシナリオについて、この工程で完成しておくことが望ましい。

３－３　応用パターン①

基幹系にアジャイルを適用する際に、品質やマネジメントを意識した基本的な考え方と、その際のポイントを工程ごとに示したが、各システムやプロジェクトにおいて、いろんな形が考えられる。基本的な考え方は先の通りであるが、さらなる応用パターンとしていくつか考えられる。ここでは、以下の3パターンについて考える。

（１）サブシステムがより重要な場合、大規模な場合
（２）基幹系本体の改修が少ない場合
（３）サブシステムの重要性が劣後できる場合、小規模な場合



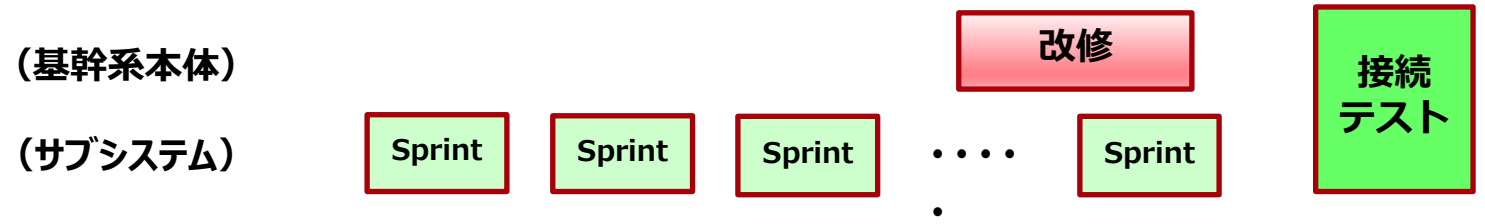
（１）サブシステムがより重要な場合や、大規模になる場合　＝アジャイル開発の前に方式設計を実施

サブシステム自身がかなり重要な場合や、かなり大規模な場合は、全体処理方式の設計のように、システム全体を設計する工程を独立して切り出し、計画工程とアジャイル開発工程の間に実施することも考えられる（これまで述べてきた計画工程で実施する想定のものを、より詳細に実施する形）。また、大規模となる場合には、基盤の設計が必要になる場合もあり、そのためにも方式設計を実施すべきということも考えられる。

基盤の構築はＷＦで実施することも多いと思われるため、構築スケジュールがアジャイル開発工程の制約とならないように注意しないといけない。（早期に構築するためにも、クラウド等を有効活用することも考えられる。）

この場合、アジャイルのリスクをより低減することが可能になると思われるが、逆に、よりWFにも近づくため、**アジャイルのメリットのひとつであるスピード感が落ちる**ことを意識する必要がある。

3－4　応用パターン②



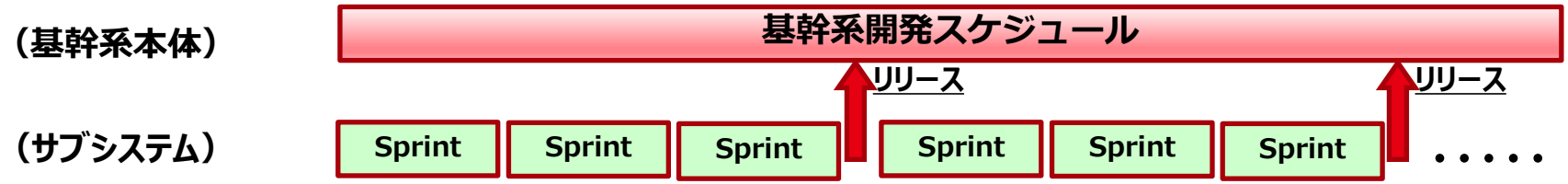
（２）基幹系本体の改修が少ない場合　＝　冒頭の計画工程を設けない

－　サブシステムに切り出したため、基幹系本体にはほとんど手が入らない場合

－　サブシステムは軽く開発するため、トータル的な開発期間が短縮

考え方は同じように事前に概要と計画を定めること、アジャイル開発完了後に接続テストを実施することは変わりないが、アジャイル開発前に計画工程を設けなくとも、**開発を走りながら計画できるというメリット**。

－　基幹系の改修時期と接続仕様確定タイミング、テスト可能時期を意識しておく



（３）サブシステムの重要性が劣後できる場合、小規模な場合　＝　軽く早くサブシステムの開発を回す

－　サブシステムは早く軽く開発するも、基幹系の開発もしっかり実施する場合

－　基幹系が開発する間に、サブシステム側は何度もリリースが可能

✓　基幹系開発と比べて何度もリリースができるため、**基幹系との接続仕様に関するところからリリースするように計画**するのが望ましい。
（接続仕様部分は基幹系の設計段階で確定し、その後に他の機能を構築していく）

✓　**リリース前の最終Sprintは、テストがメインのSprint**にすることが望ましい。
（基幹系に対する提供版の品質を上げるため）

✓　**基幹系システムへの提供時の受入基準（テストシナリオ）を、適切に決めておく**ことが望ましい。

－　受入基準検討や、各Sprintの振り返り等の重要なMtgに、基幹系担当者も入れる。

３－５　ドキュメントについての考察

アジャイルはどこまでドキュメントを作成すべきか、通常の場合でも議論になることも多い。基幹系への適用に必要と考えられるドキュメントについて考察する。

まずは、ドキュメントを3種類に分類して考える。

①**計画書関連**：プロジェクト計画書やテスト計画書など、開発プロジェクトで必要なもの
②**運用関連**：運用手順書など、稼働後のシステム運用に必要となるもの
③**設計書関連**：各種設計書など、構築するシステムそのものの元となるもの

＜①計画書関連＞

- 基本的に、**プロジェクト毎に必要なもの**を作るべき。（アジャイルかどうかに関わらない）
- プロジェクトメンバーが同じ認識を持ち、理解を一致させるために作成する。

＜②運用関連＞

- 基本的に、**会社ごとに決められたルールに合わせて**作る。（個社ごとに違うと想定）

＜③設計書関連＞

- **全体を俯瞰するもの**（システム概観、機能配置等）、基幹系との**接続仕様書**は作ることが望ましい。
- **開発者が構築時に整理したもの、認識共有に使ったもの**（ホワイトボード等）を残しておくことが望ましい。
- いずれにしても、何を目的としたドキュメントかを整理できていることが必要。

１．基幹系システムとは？

２．アジャイルを適用するメリットとハードル

３．ハードルを越えるために　～マネジメント～

４．ハードルを越えるために　～組織～

５．アジャイルに適したプロジェクト

４－１　組織に関する考察

アジャイルを採用する場合の大きなハードルの一つとして、組織が問題になることがある。中でも、今回は大きく二つの観点で考察する。

- 開発チームとしての組織　・・・　スクラムチームの組成
- 企業全体としての組織　・・・　アジャイルマインドセットの作り方

※組織の前提は別紙５参照。

NO	テーマ	備考
1	スクラムチームの組成	• POをどの部門から出すのか • POをビジネス部門から出す場合の負荷 • システム側の負荷や求められるレベル　etc.
2	アジャイルマインドセットの作り方	• ウォーターフォールの開発手順に慣れている人に対して、アジャイル開発・思考をどう定着させるか • 部門が跨ることにより、アジャイルの認識にずれが生じてしまう可能性 • コミットラインを達成することが目的となり、見積りにバッファを詰み、結果として成功しづらくなる恐れ

４－２　スクラムチーム組成の考察

スクラムチームについては、POと開発チームの作り方によって、大きくは以下の4パターンが考えられる。それぞれにメリットとデメリットがあり、**パターンの最適解はなく、各社の状況に応じて以下のパターンから選択**することを想定。

	PO	開発チーム	メリット	デメリット
1	**ビジネス部門**	**情シス部門／開発部門**	• 既存システムの体制と親和性が高く、連携しやすい • POとステークホルダーの関係性が構築できているため、要件の決定等が早くできる	・基幹系システムの組織として、縦割り組織になりがちのため、他部門の役割に踏み込まない ・非機能要件等のシステム技術面で、開発チームがPOをかなりサポートする必要ある ・POを担うビジネス部門の負担が高く、PO人材の育成も難しい ・開発チーム内で、情シス部門と開発部門の責任範囲に対する考え方の違いや、力関係等から、開発スピード低下等を招く恐れ
2	**情シス部門**	**開発部門**	• 既存の組織構造と類似しているので、組織や体制を変更するためのハードルが低い • 技術的な課題への理解が得られやすい	• ビジネス部門との距離が遠く、ビジネス部門の協力を得られない ⇒社内で「システム部門か勝手にやってるだけ」となりそう ・ビジネス部門（ステークホルダー）は言いたい放題 ・POとステークホルダーとの関係性が希薄で要件調整が難航 ・真のPOがチーム外に存在することとなる可能性
3	**ビジネス部門／情シス部門**	**開発部門**	既存組織の役割範囲内でスクラムチームを組成できそう	POの意思決定を統一できるかが課題
4	**開発部門**	**開発部門**	• スクラムチーム内での力関係が発生しづらいため、チームとしては健全に運営できる • コミュケーション、チームビルディングがしやすい	• ステークホルダーをまとめきれなかったり、優先順位を付けられない可能性あり ⇒実質的な意思決定権がない ・開発部門が正しくビジネス要求をとらえることが難しい可能性あり

4－3　アジャイルマインドのための施策

アジャイルを成功させるためには、開発チームだけでなく、その会社全体がアジャイルを正しく理解して進める必要がある。アジャイルのマインドセットについて、ステークホルダーの理解が不十分だと、失敗リスクが高まることとなる。
基幹系システムはステークホルダーが多くなる傾向があるため、**如何にステークホルダーのアジャイルマインドを浸透させるか**が、ひいては企業全体のマインドの変化となり、アジャイルプロジェクトの成功につながる。

持ってほしいマインド（ToBe）	現状（AsIs）	改善の対策（例）	さらなる施策
プロダクトの価値の最大化を優先する	与えられたタスクのみ実施 一度決めたことは必要性を再評価しない。	トップダウンで顧客価値にフォーカスしたKPIを設定する	アジャイル開発を支援するための中立的・客観的な組織の立ち上げ（第三者組織）
チームとして活動する/プロセスを共有する	決められたプロセス、役割を全うする。組織・役割の縦割りも上記を助長。	組織の構造変化が発生してしまう可能性を許容する	
		ビジネス／開発に偏った立場ではなく、フラットなメンバー（第三者）が客観的に判断するような体制をつくる	
チームを継続的に改善し続ける	ルールの縛りが強く、変更のハードルが高い。改善活動の価値の理解に乏しい。	改善活動を定量化し、ビジネス側（ステークホルダー）のメリットを伝える	
ドキュメントより対話を重視	経営層への報告、ビジネスとの課題検討等、ドキュメントを都度必要とする。	対話によるコミュニケーションの時間を確保してもらう	

１．基幹系システムとは？

２．アジャイルを適用するメリットとハードル

３．ハードルを越えるために　～マネジメント～

４．ハードルを越えるために　～組織～

５．アジャイルに適したプロジェクト

5 – 1　プロジェクトの分類

基幹系システムのうち、アジャイル開発の適したプロジェクトについて考察する。基幹系システムにおける開発プロジェクトを以下の3グループ8形態に分類し、先述のアジャイル適用時のメリットの出方や、ハードルの高低について考察を行う。

グループ	分類	プロジェクト規模	要件の確定度合い	結合度合い
新規開発	① 業務要件から新規にシステムを構築	大規模	低い	（システムによる）
リビルド/リライト	② リビルド：現行の業務仕様をもとに再構築	大規模～中規模	高い（仕様を見直す部分は低い）	（システム・改修内容による）
	マイクロサービス化等の疎結合対応			低い（マイクロサービス化等により疎結合化）
	③ リライト：現行仕様を変えずにプログラムを別言語に変換		高い	高い（既存プログラムを踏襲した場合）
機能改修/追加	④サブシステムの新規構築/改修（基幹とは切り離されているが重要度は基幹同等）	小規模	低い	低い（独立性が高い）
	⑤サブシステム（周辺システム）の新規構築/改修（基幹部分に比べ重要度が劣後する周辺システム）	小規模	低い	低い（独立性が高い）
	⑥ 基幹部分の機能追加・改修	小規模	低い	高い（既存部分と密結合/接続多のケースが多い）
	⑦ 基幹部分の機能追加・改修（法令改正対応など要件が決まっている開発）	小規模	高い	高い（既存部分と密結合/接続多のケースが多い）

5－2　新規開発の場合

・新規開発の場合、要件の確定度合いが低く、アジャイルのアプローチが最大限発揮でき、常にビジネスサイドとすり合わせしながら、優先順位の高いものから必要なものだけに絞りながら進めていくことで、**プロダクトの価値向上が最大限期待**できる。

・一定規模以上のプロジェクトとなることが多く、開発期間も長くなることから、イテレーションによる学習効果が増え、且つウォーターフォールに比べアイドリングによる工数ロスが少なくなる。同じチームメンバーで長い期間での開発となるため、イテレーション回数も増え、**生産性向上のメリットも最大化**できる。

・基幹系の新規開発の場合、プロジェクトが大規模となることが多いため、**立ち上げ時のチームビルディングや、メンバーの教育に関する部分が一番のハードル**となる。特に、開発者を外部から調達するケースの場合、基幹系の領域はアジャイル経験者が多くないため、早めに調整・準備を始めることが必要となる。また、プロジェクト期間が長くなるため、ユーザー部門と、それなりの人数に長い期間参画してもらうよう事前調整することが必要となる。着手時点で人数が揃わない場合の対応としては、期間を延ばし段階的にリリースする、スコープを絞るなどの調整を行い、経験を積んだところでアジャイルの適用範囲を広げていくというやり方も考えられる。

	開発生産性の向上	仕様バグの低減	サービスインまでの期間短縮	プロダクトの価値向上	チームビルディング
新規開発	期間が長いため学習効果大	コミュニケーションにより低減	一定の結合テストが必要なため、テスト効率化のための整備が必要となる。疎結合化されたシステムであれば比較的早期に実現も	**要件の確定度が低いため効果大**。特に優先順位をコミットし、機能単位でリリースできる場合は効果が期待できる	関係部署や人数が多くなるため、 円滑にプロジェクトを立ち上げ、回していくための工夫が必要

5 – 3　リビルド･リライトの場合

・既存システムのリビルドの場合、要件定義の骨格は既存ドキュメントをベースとすることが多いため、アジャイルの効果が出る範囲は新規開発より狭くなるが、改修部分について積極的にアジャイルの手法を適用することで、**「仕様バグ低減」、「プロダクト価値向上」の点に関しては同様の効果**が期待できる。

・特にリビルドの中でもマイクロサービス化などの疎結合対応を行う場合は、機能ごとの独立性が増し改修時の影響範囲が狭まるため、テストを効率的に実施することで、「**サービスインまでの期間短縮**」につなげることができる。マイクロサービス化でアジャイルをセットで導入するケースが多くみられるのはそうした理由による。

・リビルドの場合、プロジェクト規模は新規開発ほどではないが一定以上となるため、チームビルディングはこのケースでもハードルとなるが、仕様検討（要件検討）を行う箇所が限られるため、ユーザ部門の関係者が比較的少なくて済むという点がある。また、プロジェクトの期間もそれなりに長くなることから、繰り返しによる学習効果（生産性向上効果）も期待できる。

・ツール等を使ったリライト（言語書き換え）プロジェクトの場合は、特段要件には変更が入らず、工程を区切って変換作業、確認テストを集中的に実施するウォーターフォールのほうが効率的となるケースが多い。そのため、**リライトの場合はアジャイルを適用するメリットはあまり見当たらない**。

	開発生産性の向上	仕様バグの低減	サービスインまでの期間短縮	プロダクトの価値向上	チームビルディング
リビルド	改修規模/範囲によるが、学習効果により生産性は向上	仕様変更部分については低減	一定の結合テストが必要なため、テスト効率化のための仕組みが整備されているかによる。	仕様変更の範囲による（リライトのように少ない場合効果は限定的）	新規開発に比べるといくらか容易
リビルド（マイクロサービス化）	同上	同上	機能毎のリリースが可能となるため短縮可能	同上	同上

５－４　機能追加/改修の場合

・機能追加/改修の場合、改修部分について積極的にアジャイルの手法を適用することで、**「仕様バグ低減」、「プロダクト価値向上」の点に関しては同様の効果**が期待できる。

・サブシステムという形で既存部分から一定程度独立した機能の開発の場合、既存部分への影響範囲、テスト範囲が狭くなるため、テストを効率的に実施していくことで、「**サービスインまでの期間短縮**」効果が得られやすくなる、また、案件次第ではあるがプロジェクト規模が小規模件であれば、新規開発やリビルドに比べチームビルディングも進めやすいという面もある。基幹系システムでそうしたサブシステムの開発がある場合、特に基幹部分よりも重要度が劣るような周辺機能の開発の場合、チャレンジしやすく、アジャイル適用の効果も上げやすい案件と思われる。

・基幹部分を含む改修の場合は、影響範囲・接続先が増えるケースが多いため、すぐには期間短縮効果を上げられないことも多いが、その場合も、自動テストの整備を並行して行ったり、経験を積むことによる学習効果で、徐々に短縮効果を期待できる。

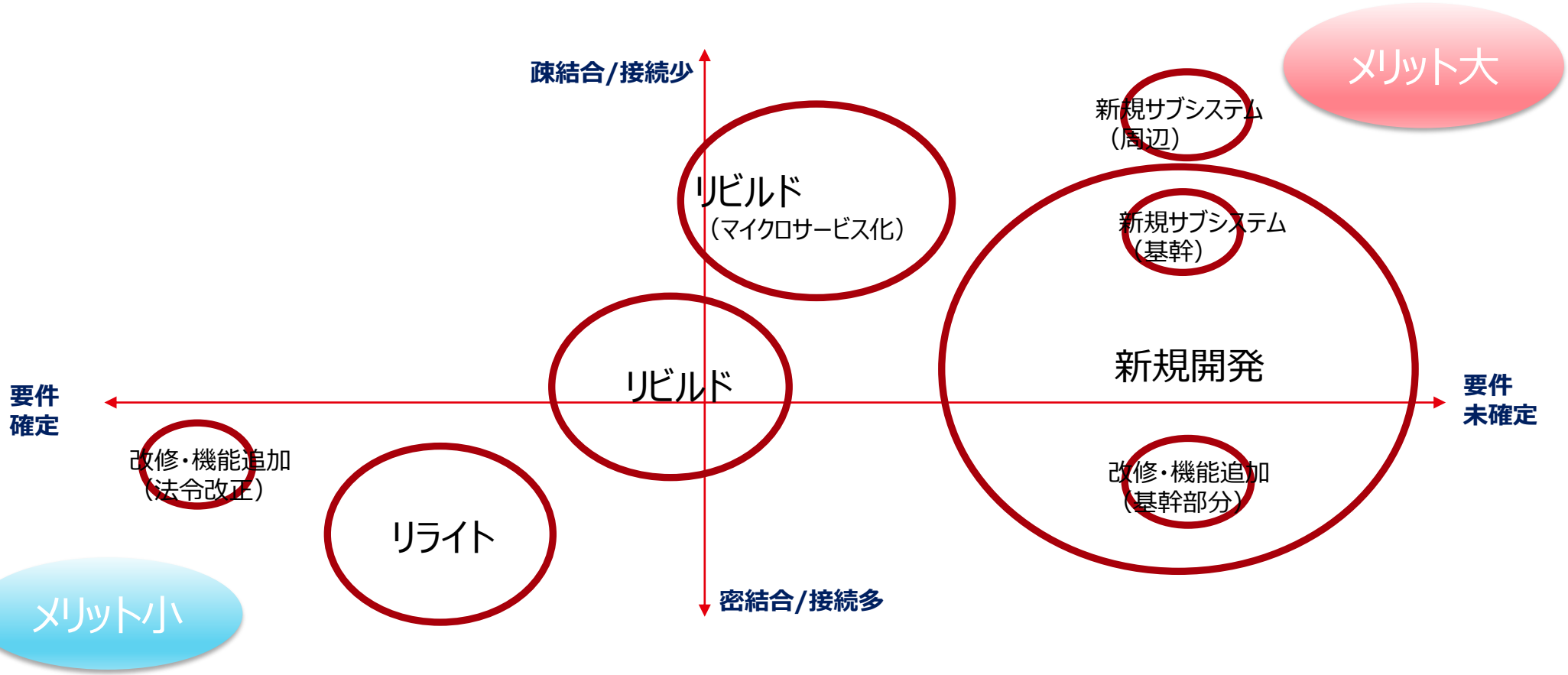
・あらかじめ要件が固まっている場合は、プロダクトの価値向上やサービスインまでの期間に関してウォーターフォールとの差異がないことから、**機能改修のうち法令改正への対応などの場合はアジャイルを適用するメリットはあまりない**ことが多い。よって、改修内容を吟味したうえでよりベターな手法を取ることが望ましい。

	開発生産性の向上	仕様バグの低減	サービスインまでの期間短縮	プロダクトの価値向上	チームビルディング
サブシステムによる新規機能構築	改修規模/範囲によるが、学習効果により生産性は向上	コミュニケーションにより低減	サブシステム毎のリリースが可能な場合は、効果が期待できる。	優先順位をコミットしリリースしていくことで効果を上げることが可能	小規模かつ新規要件のため既存メンバーにとらわれずチームの組成が可能
基幹部分の改修	改修規模/範囲によるが、学習効果により生産性は向上	コミュニケーションにより低減	影響範囲が広い場合は、テスト自動化等を整備する必要があり、効果が出るまで時間がかかる	上記同様に改修範囲で効果。ただし、法令改正対応などは特段WFとの差異なし	小規模な案件が多いためチーム組成のハードルは高くない。

5－5　メリットを出しやすい基幹系プロジェクトは？

下の図は基幹系の開発プロジェクトを、要件の確定度（横軸）結合・接続の多さ（縦軸）で示したもの。プロジェクトやシステムの特性による部分もあるが、一般的に**より右上に位置し、規模がコンパクトな（＝丸が小さい）プロジェクトが、相対的にアジャイルのメリットを出しやすく適用のハードルが低いプロジェクト**と考えられる。

基幹系は品質第一という面があるため、ユーザー企業・ベンダーとも従来側の開発を変えずに、アジャイルの採用が進んでいない状況が多く見られるが、右上のプロジェクトからまずはチャレンジし、経験を積みながら適用範囲を拡大していくことが、基幹系にアジャイルを浸透させていくためのよいアプローチではないかと思われる。



（さいごに）なぜ、基幹系にアジャイルを活用するのか？

経産省も推進するDXでは、アジャイルを推奨している。

※ここでいう「アジャイル」とは、内製化、一体開発、スモールスタート、トライ＆エラー等の取り組みを総称したように使っている

しかしながら、日本ではアジャイルはなかなか定着しない。
各社、実施事例、成功事例も増えてはいるが、定着というには不十分。

→ **それは、アジャイルが、日本の古くからのモデル（SIerモデル）と対極にあるからではないか。**

※SIerモデル：大手ベンダへの一括請負による丸投げ。実際に開発するのは末端の作業者だが、二重/三重の請負によるコスト大

成功事例の各社が実施しているのは、基幹系とは切り離した独立新規システムのみ。その部分での実例は増えてきているも、基幹系には手を付けない。（つけられない）

※基幹系システムは、上記SIerモデルの象徴的なもの。

→ **これが、いくら実例が増えても、なかなか定着感がない理由ではないか。**

基幹系システムへの適用ができないと、日本には根付かないのではないか。
基幹系システムへの活用を考えることが、アジャイルが定着する唯一の方法ではないか。

⇒ **継続して活動を続けていきたい！**

研究会参加企業・幹事団（2022年3月時点、50音順）

アイエックス・ナレッジ㈱
旭化成㈱
アフラック生命保険㈱
コベルコシステム㈱
㈱ジェーシービー
㈱J-POWERビジネスサービス
㈱証券保管振替機構
スミセイ情報システム㈱
住友生命保険相互会社
住友電工情報システム㈱
大和ハウス工業㈱
東京電力ホールディングス㈱
東京海上日動システムズ㈱
㈱東証システムサービス
ＴＯＴＯ㈱
日清食品ホールディングス㈱
ニッセイ情報テクノロジー㈱
日販テクシード㈱
㈱日本取引所グループ
㈱ノーリツ
パナソニック㈱
㈱村田製作所

（22社、25名）

●研究会幹事団(敬称略)

部会長	山森 一頼	㈱日本取引所グループ
副部会長	水田 耕太郎	㈱日本取引所グループ
副部会長	中林 達哉	スミセイ情報システム㈱
副部会長	山田 晋	大和ハウス工業㈱

(appendix)

（別紙１）アジャイル開発の定義　特徴

(1)『計画→設計→実装→テスト』といった開発工程を、機能単位の小さいサイクルで繰り返す手法

(2)　優先度付けによりサービスインまでの期間短縮が可能
　アジャイル開発の場合、優先度の高い重要な機能から着手し、リリースしてからブラッシュアップしていくことが可能。つまり、サービスインまでの期間を短縮することができ、ビジネス のスタートを早めることができる。

(3)　仕様変更に強く、プロダクトの価値を最大化することに重点を置いた手法
「変化を受け入れる（要件の変更は開発期間中でもよい）」という前提で進めるので仕様変更に強く、プロダクトの価値を最大化することに重点を置いた開発手法

(4)　対面でのコミュニケーションを基本に業務とＩＴが一体となって推進
　開発において、基本的には不要なドキュメントは作成しない

（別紙２）アジャイル開発の定義　構成要素（プラクティス）

- イテレーション（Sprint）・・優先度の高い要件から順に要件定義～機能の確認までを実施し、反復的に工程を進める１サイクル単位。イテレーションの期間は一般的に1～2週間程度で、イテレーション毎に機能をリリースする。期間の設定は開発チームによって変化する
- ユーザ部門との協働体制の構築・・要件定義と設計の工程の区分けを無くし、要件の検討とシステムの設計を同時並行的に実施する。システム部門も要件定義プロセスに入り込むことで、システム部門とユーザー部門が協働で要件確定し、案件を推進していくための体制を構築する。
- Virtual One Room（Teams・Zoomなどを活用した環境整備）・・一定の期間内・時間は、チーム全員が顧客満足度の高いモノづくりに専念するための環境を整備する。
- リリース計画・・「いつまでにどの機能をリリースできるか」というプロジェクト全体を管理するためのリリース計画で、プロジェクトのゴール、イテレーションの長さ、ユーザーストーリーの優先順位を決定する。
- Daily Scrum・・ユーザー部門含めて、計画・実行内容・振り返り内容を日次で確認し、モノ作りを推進する。
- ドキュメント削減・・要件の合意プロセスやレビュープロセスにおける部門間、メンバー間での書面のやり取りの工数を削減する。
- ペア作業（ペアプログラミング）・・主に品質向上とスキル継承を目的として、ドライバーとナビゲーターと呼ばれる２人の開発担当者が共同してプログラミング等の作業を行う。
- ユーザーストーリー・・アジャイル開発において「要件」の代わりに用いられる概念。「ユーザーが実現したいこと」「ユーザーにとって価値があること」（意図・要求）を簡潔にまとめた文章のこと
- プランニングポーカー・・作業の重みを相対的に評価する簡易的な見積手法。
- タスクかんばん・・１つの付箋紙につき１つのタスクを書き出し、壁やホワイトボードに貼り、チーム内のタスクの状況を可視化する手法。
- KPT法・・チームの改善サイクルをまわすための手法。改善点だけでなく、良かった点についても話し合う。
- バーンダウンチャート・・タスクの消化状況を可視化する手法。
- 朝会/夕会・・毎日の始業/終業時に行う、短いMTGのこと。進捗、課題の共有を行う。

（別紙3）基幹系に適用する場合のメリット・ハードルの考察　1/5

(1)『計画→設計→実装→テスト』を繰り返す手法

（メリット）開発生産性の向上

アジャイル開発は、『計画→設計→実装→テスト』を、機能単位の小さいサイクルで繰り返す手法であるが、繰り返による学習効果による生産性向上については、基幹系システムでもアジャイル開発採用のメリットとして享受できると考えられる。例えば、要件調整を繰り返すことにより、ユーザーと調整するコツがつかめたり、開発を繰り返すことで、似た部分は実装スピードが上がり、品質も向上すること等が期待できる。これらの効果は、同じ人間が継続してアジャイルプロジェクトに参加する場合に特に発揮できるものである。

また、ウォーターフォールの場合、設計・製造着手は、すべての要件・設計が確定し、体制整備ができてから一斉に取り掛かることが多く、特に上流工程では、要件・仕様が固まっても、未確定な要件や一部の設計が仕掛中の場合に、次工程へ進めない状態（アイドリング状態）が発生するが、アジャイルの場合、開発単位を細かくし、要件（仕様）が確定したものだけでスピーディに次工程の設計、製造作業に移れるため、アイドリング状態が少なくなるという利点もある。特に、規模の大きな基幹系システムの場合、アイドリング状態が発生する期間・範囲が広くなる傾向にあるため、この効果は大きなものとなる。

（2）（メリット）仕様バグの低減（動くプログラムで確認できる）

（ハードル）品質管理の難しさ（ユーザテストでOKとなってもレアケースで不具合潜在していたら見逃す可能性あり）

アジャイル開発の場合、動くプログラムで確認できるため、仕様バグが低減するというメリットが挙げられる。一方で、短期間でリリースする場合等は特に、テスト期間が短く、テスト実施範囲が狭くなる傾向があるため、バグを見逃しやすいというのがデメリットとして挙げられる。基幹系システムでアジャイルを採用する場合は、リリース前に非改修個所も含めて通しでの機能確認、ノンデグ確認を行うケースが多いが、相応のテスト期間を取り、バグを潰しこむことで、こうした品質面のリスクは下げられると考えられる。

（別紙3）基幹系に適用する場合のメリット・ハードルの考察　2/5

（３）開発において、基本的には不要なドキュメントは作成しない

（メリット）ドキュメントを簡素化できる

（ハードル）維持に向けた引継ぎの難しさ

ウォーターフォールでは、要件定義・設計工程と製造工程ではメンバーが変わることが多く、その場合の拠り所はドキュメントになるが、ドキュメントの読み違いでバグが発生することが多々ある。アジャイルの場合、ドキュメントに頼らず、コミュニケーションやソースコードで理解することに重点を置くことで、結果的にドキュメントが簡素化される。アジャイルの場合、同一プロジェクトにおいてメンバーは初めから終わりまで継続して関与するため、明文化されてないルールが、チームのナレッジとして開発期間中に蓄積されていくが、開発後半に作成する維持向けのドキュメントも、ナレッジが整理された状態で作成されるため、ドキュメントが成熟化されたものになるというのが、アジャイルの場合のドキュメントの優位点となる。

開発期間中の様々なコミュニケーションの結果として、ポイントが絞られた状態で出来上がったドキュメントこそが、維持フェーズにおいてもよいドキュメントということが言えるだろうし、ドキュメントがコミュニケーションによりポイントが絞られ、成熟化されていくことがアジャイルにおけるドキュメントの強みとも言える。そうしたドキュメントの考え方、ポイントの絞り方をメンバーで共有し、強みを生かしたドキュメント整備を行うことにより、基幹系システムなど規模の大きなプロジェクトで洗練されたドキュメント整備が可能となる。

ただし、アジャイルの場合、ドキュメントの簡素化により、維持フェーズで設計の背景やポイント等の引継ぎがうまくできないということが起こりやすいのも事実であり、ドキュメント以外の部分を含めてドキュメントに頼らない工夫や対処が必要となる。例えば、背景や経緯があまり必要ないプログラム間をつなぐ仕様書等については、直接ソースを見るという整理にしてドキュメントを削減する代わりに、メンバー全員が、いつでも必要な時にソースを見れるよう、見るための環境、ツール類を整備しておく、などが考えられる。

（別紙3）基幹系に適用する場合のメリット・ハードルの考察　3/5

（４）　優先度付けによりサービスインまでの期間短縮が可能

（メリット）　・サービスインまでの期間が短縮

（ハードル）　・品質管理の難しさ

アジャイルの場合、優先度の高い重要な機能から着手できるため、素早くリリースしてからブラッシュアップしていくことが可能である。つまり、サービスインまでの期間を短縮することができ、ビジネスのスタートを早めることができるという特徴点がある。

基幹系システムでも、アジャイルを採用するシステムがすでに疎結合状態となっており、機能分割ができている場合は、そうした効果を出すことも可能だが、規模が大きく蜜結合状態となっている場合、単機能の確認だけでなく、一連の機能をつなぎ合わせて動作確認を行う必要があるため、短期でサービスインという点に関しては、なかなか効果を見出すことは難しい。無理にサービスインを早めようとして、テストが不十分なままリリースした場合、見逃したバグ等により基幹系システムが担う重要業務に影響が出てしまう可能性もあるため、アジャイルを適用した場合のリリースサイクルや、リリースまでのテスト期間、テスト実施範囲については、改修内容や品質面の考慮を十分行ったうえで決定する必要がある。

基幹系システムでサービスインまでの期間短縮などアジャイルの特徴点として言われる効果を出したい場合は、マイクロサービス化などの疎結合化と組み合わせて導入するというのが、改修の手間はその分かかることになるが、正攻法であり近道であるかもしれない。

（別紙3）基幹系に適用する場合のメリット・ハードルの考察　4/5

(5) 仕様変更に強く、プロダクトの価値を最大化することに重点を置いた開発手法

（メリット）　・成果物の価値が向上、要求の変更に対して迅速に対応

（デメリット）・当初計画にコミットできない（計画の管理が困難、計画変更の妥当性の評価が困難）

開発当初に決めた要件で、もっといい案があった場合に開発期間中に柔軟に変更可能なことが、アジャイルのメリットである。要件変更が入った場合に、コストや納期に影響が出るリスクが高まるというデメリットもあるが、その時点でユーザーが一番必要なものを、必要な形で作れる（≒プロダクトの価値を最大化できる）という内容面のメリットの方が大きいといえる。アジャイルの場合、当初計画からコストやスケジュールが変動しやすいというリスクに関しては、関係者を含め状況の共有が必要となる。

アジャイル開発を行う際に、主目的に照らして何から開発を始めるかは非常に重要であり、メンバー内でアジャイルのメリット・目的がどれだけ理解・共有されているかがカギとなる。基幹系のような規模の大きなプロジェクトの場合、開発者だけでなくユーザー部門、ベンダを含め、参加している人全員がアジャイルのメリット・目的をきちんと理解していないと、この「プロダクトの最大化のメリット」は得られにくい。それらを関係者が理解したうえで、開発順位やチーム構成、テストの仕方については、関係者一緒になって考えてもらう必要があり、そのためにもメリット・目的を理解するという最初の準備作業は非常に重要となる。

一方で、アジャイルの場合、仕様変更の際に、当初要件として挙がっていた機能を削るやり方がどれだけ受け入れられるかが課題となることが多い。当初の計画（コストやスケジュール）を守るために、開発の最初の段階で、途中優先度の高いものが出てきた場合は、優先度が低いものは削るということをあらかじめユーザ部門と合意し、実際の変更の際も伝えていく必要がある。そうした考えをユーザ部門と合意し、常に優先度の高いものを選択して作り続けることで、価値を最大化することにつながっていく。また、アジャイルの場合、いくらでも要件変更できるというミスリードを防ぐために、変更を受ける期限については、最初に握っておくのがベターである。

（別紙3）基幹系に適用する場合のメリット・ハードルの考察　5/5

(6) 対面でのコミュニケーションを基本に業務とＩＴが一体となって推進

（メリット）・ユーザとの密なコミュニケーションにより仕様バグが低減、一体感でメンバーのモチベーション向上、

・幅広工程を担うためメンバーのスキルアップ

（ハードル）・チームビルディングの難しさ

ITとユーザ部門が一体となって密に連携しながら開発できるため、要件定義などの手戻りのコスト、リスクを減らせるというメリットは、基幹系システムを含めプロジェクトの規模に限らず享受できると思われる。一方で、規模が大きくなると関係者が多くなるため、体制の組み方や役割分担の整理が難しいところである。機能ごとに小さなチームが複数ある体制を組むことができればよいが、既存の基幹系システムの開発体制をすぐにアジャイルのメリットが引き出せる体制に移行できるかというと、なかなか難しいところがある。また、アジャイルを内製で行う場合とベンダで開発する場合とでは、役割分担など整理の仕方は異なる。ベンダに開発をお願いする場合は、ユーザー側は要件の提示者として入っていき、開発やチーム間の横連携は従来通りベンダ側にお願いすることになる。ウォーターフォールで今までやってきた役割分担をベースに、どこをどう変えるかをあらかじめ関係者で整理・合意し、開発の中で役割を少しオーバーラップしながら進めていくというのが、一定規模以上の開発においてアジャイルを導入する場合の一つのやり方となる。

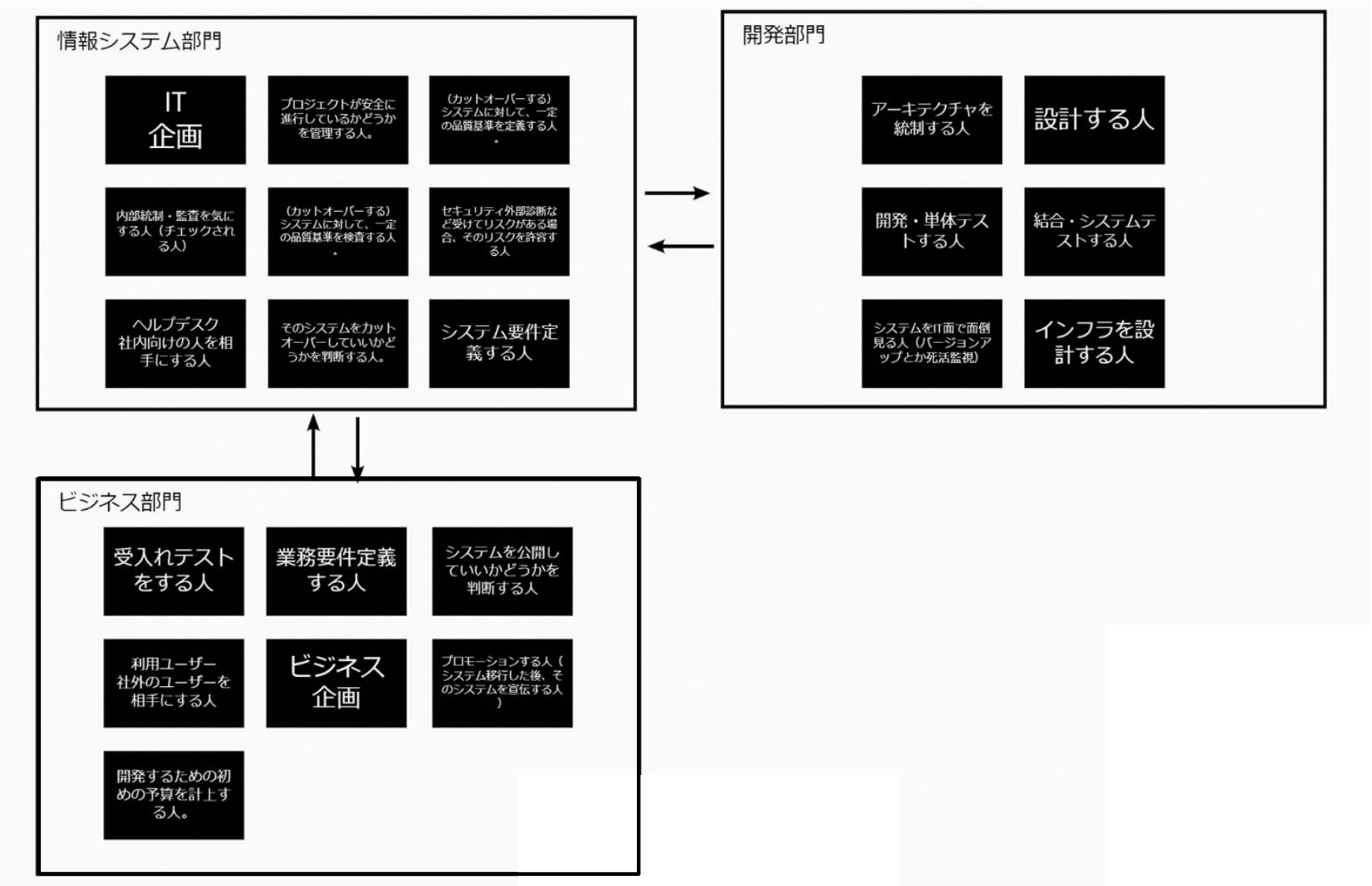
アジャイル開発の場合、スクラムを組むのが一般的だが、その場合少人数（５～8人くらい）が対面でコミュニケーションするやり方が基本となる。ウォーターフォールの場合、工程縦割りで、設計担当者がドキュメントを使って製造チームに伝達する形になるが、アジャイルの場合は基本プロジェクトの初期から通しで参加するメンバーが多いため、コミュニケーションコストを下げることができ、また、ウォーターフォールで発生しがちなチーム間連携でのミスのリスクを下げられるのがメリットとなる。ただし、基幹系システムの開発の場合、それなりの開発規模となり、関係者も大人数となるケースが多いため、アジャイル特有のコミュニケーションコストの削減がどこまで発揮できるかは微妙な部分もある。そのため、チームの人数を固定して、そのチームでできる範囲で段階的に開発していくというのが、このメリットを継続的に出すための一つの形となる。基幹系の場合、全面刷新のようなケースで適用するよりも、機能毎に区切って新規開発やリビルドを行うケース、あるいは一部のサブシステムで適用する方が、コミュニケーションミス等のデメリットを抑え、メリットを見出しやすいと思われる。

（別紙４）ハードルを越えるためのマネジメント　その他のポイント

PJごとに状況は異なるものの、さらに注意すべきポイントの具体例を以下に示す。

	工程	注意すべきポイント	備考
1	計画工程	アジャイル開発や利用技術、基幹系業務・システムについて、メンバーのスキルマップを共有することが望ましい	
2		プロダクトバックログの構築の際、WF部分との機能範囲の明確化、接続仕様の明確化をしておくことが望ましい	
3		基幹系チームとのコミュケーションルート、会議体を決めておくことが望ましい	
4	開発工程	バックログの棚卸しの際、接続仕様に絡むものやハイブリッド相手からの変更要求については、特に注意して優先度や影響を確認することが望ましい	
5		ベロシティの安定状況、課題解決までに要した時間、要員のスキル・モチベーション、接続先とのテスト計画・準備状況などを測っておくことが望ましい	
6		Sprint期間中の品質を維持するため、ユーザーストーリーの受入れ基準、Sprintの完了基準について、業務仕様のみならず、非機能仕様（レスポンス、スループット、エラー時挙動等）も加えることが望ましい	
7		実装内容からドキュメントを作成することがある場合（運用・操作手順書等）には、実装内容の不備や改善事項の抽出を意識しながら作成できると品質向上につながる	
8		システムテスト、ユーザーテストの実施要領はサブ、メイン部分の連携を考慮し、関係者へのレビューを総合テスト前までに完了させることが望ましい	
9	総合テスト工程	エラー原因の分布状況や推移を確認し、基幹系システムの稼働状況や移行等の支障になるものから重点的・優先的に対応を講ずることが望ましい	
10		サービス開始に向けて、基幹系システムや接続先、利用者も含めて実サービスを想定したリハーサルを実施することが望ましい	
11		稼働後の運用や追加開発で見直しが必要になるプロセスを意識して確認することで、稼働後の運用品質を確認することが望ましい	
12		総合テストで発見したバグをバックログとして管理する場合には、基幹系システム側のリリース要否・影響を確認し、リリース前・後どちらで対応するのかの切り分けを含めて、バックログの優先順位を管理できていることが望ましい	

（別紙５）組織のペルソナ


情報システム部門
IT
企画
プロジェクトが安全に進行しているかどうかを管理する人。
（カットオーバーする）システムに対して、一定の品質基準を定義する人。
内部統制・監査を気にする人（チェックされる人）
（カットオーバーする）システムに対して、一定の品質基準を検査する人。
セキュリティ外部診断など受けてリスクがある場合、そのリスクを許容する人
ヘルプデスク
社内向けの人を相手にする人
そのシステムをカットオーバーしていいかどうかを判断する人。
システム要件定義する人
開発部門
アーキテクチャを統制する人
設計する人
開発・単体テストする人
結合・システムテストする人
システムをIT面で面倒見る人（バージョンアップとか死活監視）
インフラを設計する人
ビジネス部門
受入れテストをする人
業務要件定義する人
システムを公開していいかどうかを判断する人
利用ユーザー
社外のユーザーを相手にする人
ビジネス
企画
プロモーションする人（システム移行した後、そのシステムを宣伝する人）
開発するための初めの予算を計上する人。

（別紙６）スクラムチームのパターン毎のデメリット対策　1/4

NO	PO	開発チーム	備考
1	ビジネス部門	情シス部門／開発部門	

メリット	
• 既存システムの体制と親和性が高く、連携しやすい • POとステークホルダーの関係性が構築できているため、要件の決定等が早くできる	
デメリット	対策
• 基幹系システムの組織として、システムが多く、縦割り組織になりがちのため、既存システム（WF）の役割を引きずり、各部門の役割に踏み込まない	• スクラムの目的やメリットを全員が理解することで協力体制を作る
• 開発チームがPOをかなりサポートする必要ある（非機能要件等のシステム技術面で）	• 開発チームやスクラムマスターが率先してPBLを準備して価値をPOに説明する • 開発チームがサポートする内容を明確にする
• POを担うビジネス部門の負担が高く、PO人材の育成も難しい	• POを専任にする • 中長期的には、研修を行いPOやPOをサポートできる人材を育成する
• 開発チーム内で、情シス部門と開発部門の責任範囲に対する考え方の違いや、力関係等から、開発スピード低下等を招く恐れ	• 情シス部門からの人選に気を付ける（既存システムの関係者は参画させない等） • 情シス部門と開発部門の責任範囲を明確にできない場合は、ステークホルダーにする

（別紙６）スクラムチームのパターン毎のデメリット対策　2/4

NO	PO	開発チーム	備考
2	情シス部門	開発部門	ビジネス部門は、 ステークホルダーとして参画

メリット	
• 既存の組織構造と類似しているので、組織や体制を変更するためのコストやハードルが低い • 開発チームに整理された情報が伝達されやすい、技術的な課題への理解が得られやすい	
デメリット	対策
• 基幹系システムの組織として、ビジネスとシステムとの距離が遠く、ビジネス部門の協力を得られない ⇒社内で「システム部門か勝手にアジャイルやってる　だけ」となりそう	• システム部門主体のアジャイル開発にならないように、開発部門・情シス部門に越境できるようなマインドセットを育てる • チームビルディングの教育
• ビジネス部門（ステークホルダー）は言いたい放題	• 必須機能の洗い出しとタスクの優先順位付け
• POとステークホルダーとの関係性が希薄で要件調整が難航しそう	• 現行組織の責務を引き回さないような運営づくり • ステークホルダーにアジャイルの教育を行う
• プロダクトの価値を判断する人（真のPO）　がチーム外に存在することになり、優先度の判断を誤りそう	• POへの権限委譲 （その前提条件として、POには社内調整スキルや高い信頼関係等が求められる）

（別紙６）スクラムチームのパターン毎のデメリット対策　3/4

NO	PO	開発チーム	備考
3	ビジネス部門／情シス部門	開発部門	

メリット	
• 既存組織の役割範囲内でスクラムチームを組成できそう	
デメリット	**対策**
• POの意思決定を統一できるかが課題	• 1チーム内に複数のPOが存在する場合は、チーフPOを立て、各POのコントロールや最終意思決定を行う。（チーフPOに対して権限移譲） • 意思決定プロセスを可視化し、PO同士が「いつ、誰が、どのような根拠・理由で、どのように判断したのか」を分かるようにする • 1プロダクトに対して複数のスクラムチームが組成され、ビジネス部門からのPO、情シス部門からのPOが各チームに設定される場合は、チーフPOを立てた上でPO間で連携する仕組みを作る

（別紙６）スクラムチームのパターン毎のデメリット対策　4/4

NO	PO	開発チーム	備考
4	開発部門	開発部門	ビジネス部門と情シス部門は、 ステークホルダーとして参画

メリット	
• スクラムチーム内での力関係が発生しづらいため、チームとしては健全に運営できる • POと開発チーム間でのコミュケーション、チームビルディングがしやすい	
デメリット	対策
• POのスキルに依存する。 ステークホルダーをまとめきれなかったり、 優先順位を付けられない可能性あり ⇒実質的な意思決定権がない	• ある程度ステークホルダーと対等にやり取りできる人材をPOにアサインする • PO向けの研修やPOが集まる共有会等を定期的に社内で実施する • ステアリングコミッティを密度高く実施する
• 開発部門が正しくビジネス要求をとらえることが難しい可能性あり	• POがビジネス部門に常駐する等、入り込む工夫が必要
• ビジネスの要求が開発部門に伝わりづらい	• スプリントレビューに必ずビジネス部門のステークホルダーに参加してもらい、開発チームに直接フィードバックできる場を作る