2009 情財第 0507 号

非ウォーターフォール型開発に関する調査

調査報告書

平成 22 年 3 月 30 日

独立行政法人　情報処理推進機構
ソフトウェア・エンジニアリング・センター

調査報告書目次

表 目 次

図 目 次

１ 背景と目的

１．１ 背景

ウォーターフォール型でない、アジャイル的な開発など非ウォーターフォール型開発は、Web アプリケーションなどを中心に広がり、ニーズへの迅速な対応、開発者技術者のモチベーション向上等に成果を挙げている。一方、適用にあたり、品質・信頼性の確保の問題、情報不足や相互の理解不足、契約面での課題、開発管理上の課題、人材育成等への対応が整備されていない、との指摘もあり、適用範囲は限られている。これらの課題への対応を検討し、非ウォーターフォール型開発の各種開発手法とそれらが効果的に適用される分野ごとのマップ作りやガイドラインの提供等が求められている。

情報システムには、高信頼性が求められる基幹システムと、市場へいち早く提供していくことに価値がある製品があると考えられるが、特に、開発形態が多様化している後者において、どのように対応していくべきか、欧米との競争力確保も視野に入れつつ検討していく必要がある。

１．２ 目的

本調査では、それぞれのソフトウェア開発方法の適用領域及び開発を成功に導く要因を明確にするための調査を実施し、ソフトウェア・エンジニアリング上の課題を抽出し明らかにするとともに、わが国のソフトウェア開発力の向上に資することを目的とする。なお、本調査では、エンタプライズ系ソフトウェアを対象とする。

２ 調査項目

２．１ 非ウォーターフォール型開発手法の適用分野別マップ作りと各種手法の体系化のための調査

本調査項目においては、非ウォーターフォール型に含まれる開発手法を整理するとともに、現実にどのような領域においてそれらが活用されているのかを把握するため、以下に示す 4 つの項目について調査を行った。

(1) 非ウォーターフォール型開発手法の分類と整理

初めに、ソフトウェアの開発型の変遷を振り返り、本調査において対象とする非ウォーターフォール型開発に含む開発型を定義する。本調査では、次に示す 4 つの開発型を調査対象とする。

- プロトタイピング開発
- 計画駆動型開発
- アジャイル型開発
- パッケージ利用開発

次に、上記 4 つの開発型について定義するとともに、それぞれに含まれる具体的な開発手法を整理する。つまり、たとえば、アジャイル開発であれば、eXtreme Programming(XP)やスクラムなどといった粒度で手法を整理することである。

(2) 非ウォーターフォール型開発の特徴調査

(1)にて整理した調査対象とする開発手法ごとに、それを実践するためのプラクティスや原則を整理し、それぞれの特徴について分析する。

(3) 文献・Web およびヒアリングによる事例調査

非ウォーターフォール型の開発を実践している事例に関する情報を、文献・Web およびヒアリングによって収集し、どのようなソフトウェアを開発した事例であるか(事例概要)、どのようなプロジェクトであるか (プロジェクト基本情報)、ならびにその事例が持つ特に着目すべき点（特徴）を整理する。

なお、本調査においては、研究会委員、研究会委員から紹介いただいた企業、および文献調査によって収集した事例を実施した企業に対してヒアリングを行った。

(4) 非ウォーターフォール型開発の適用分野別マップ試作

非ウォーターフォール型の開発がどのような分野において実践されているのかを把握するため、(3)にて示した事例を別途定義する座標系にマップするとともに、各事例のミクロな特徴を把握するため、各々の実践内容をレーダチャートにマップする。

２．２ 各種手法の利害得失の調査

(1) 各手法のタスク（プラクティス）の影響評価

前節にて示すとした各適用領域について、非ウォーターフォール型開発手法を用いることの利害得失について分析する。

⑵ 各手法の代表的なプラクティスの影響評価

各開発手法に含まれる代表的なプラクティスや原則が、ソフトウェア開発におけるさまざまな観点で、どのような影響を与えうるかを分析する。

２．３ 非ウォーターフォール型開発の品質・信頼性及び開発管理についての課題整理と提言

本調査項目では、ウォーターフォール型開発に比べて問題点と指摘されている品質・信頼性及び開発管理について、実態を調査・分析するとともに課題を整理する。

⑴ 非ウォーターフォール型開発の品質・信頼性及び開発管理の課題整理

ここでは、非ウォーターフォール型開発におけるプラクティスをそのまま適用した場合に､品質･信頼性及び開発管理においてそれぞれ課題と成り得る事柄を整理するとともに、非ウォーターフォール型開発における品質・信頼性及び開発管理についての不安や課題およびプラクティスを実践するために必要となる前提条件を抽出する。

⑵ 非ウォーターフォール型開発の品質・信頼性及び開発管理の提言

⑴にて抽出された課題や前提条件を克服するための工夫を実践事例より抽出し、信頼性をはじめとする信頼確保に向けた提言を行う。

２．４ 非ウォーターフォール型開発への顧客参画に関する現状調査とあり方への提言

本調査項目では、非ウォーターフォール型開発への顧客の関わり方の実態を把握するとともに、より望ましい顧客参画のあり方とその実現に向けての提言を行う。

⑴ 非ウォーターフォール型開発における顧客参画について

非ウォーターフォール型開発において顧客が参加するプラクティスを抽出するとともに、顧客のより望ましい参画を実現するための前提条件を抽出する。

⑵ 文献・Web およびヒアリング調査による顧客参画の実態把握

事例に基づき、顧客がいかに非ウォーターフォール型開発へ関与しているかの実態を把握する。

⑶ 非ウォーターフォール型開発における顧客参画への提言

非ウォーターフォール型開発に求められる顧客参画のあり方と事例調査により明らかとなった顧客参画の実態とのギャップが、いかにすれば埋めうるかを提言するとともに、実際に課題を克服するために行われている取組みを抽出する。

２．５ 契約方式上の課題整理

本調査項目では、非ウォーターフォール型開発において、現状の契約形態での課題となりうる諸点を明らかにし、それらを解決するために今後どのような点について検討を深めていく必要があるのか整理する。

(1) 非ウォーターフォール型開発における契約方式上の課題整理

非ウォーターフォール型開発手法を適用した開発を実施する場合に、従来型の開発手法による開発が前提となって定められた契約方式では十分に対応しきれない点を抽出する。

(2) 非ウォーターフォール型開発における今後の契約のあり方

(1)にて抽出した契約上の課題を解決するために、どのような契約形態が有効かについて、既に提案されている形態等に基づいて、妥当性を検討するとともに、今後の契約のあり方について整理する。

(3) 契約方式上の課題解決の方法

利害関係者の明記、イテレーションの考え方、完成責任の考え方等、非ウォーターフォール型ソフトウェア開発業務に関する契約方式に関する解決すべき課題について検討を行うとともに、継続的な議論を必要とする項目について提示する。

３ 非ウォーターフォール型開発手法の適用分野別マップ作りと各種手法の体系化のための調査

３．１ 非ウォーターフォール型開発手法の分類と整理

(1) 開発方法論の変遷

1970 年に W.W.ロイスが提唱したソフトウェアのライフサイクルプロセスの概念が源流といわれるウォーターフォール型の開発方法は、開発のライフサイクルが要件定義、設計、構築、テストといったいくつかのフェーズに分割され、各フェーズとも前フェーズの成果物をインプットとして実施されるものである。この方法は現在でも非常に幅広く活用されている一方、いくつかの問題点が指摘されている。たとえば、要件定義段階で漏れなく要件を定義することが難しい、テストが開発の最終盤に位置するためシステムの不具合の発見が非常に遅くなる、あるいは、比較的長期間にわたる開発であっても、ひとたび確定した要件は開発が完了するまで変更しないためリリース時にはビジネスニーズを満たさない場合があるなどである。

このような問題点に対応するために、1980 年代に入るとプロトタイピング型の開発方法が提案された。これは、システムに対する要件を具体化するために、短期かつ簡易にプロトタイプを作成し、それをシステムユーザが利用して得た結果を開発にフィードバックするものである。このようにすれば、要件を誤解したり必要な要件を漏らしたりする可能性を低減することができる。

また、1988 年には、B.W.ベームによりソフトウェア開発における活動とリスクを最小限に制御するためのリスク管理と結合したスパイラル型開発が提唱された。これは、計画策定、目標・代替案・制約の決定、代替案とリスクの評価、および開発とテストの実施からなる一連のサイクルを繰り返しながら、運用コンセプトを固め、そこから要件を導き出し、それに基づいて設計および構築を行い、開発されたものをテストするという各段階を渦巻状に上っていく開発方法である。

そして、1990 年代に入ると、より積極的に要件の変更を受け入れる開発方法として、反復型の開発方法がいくつも提唱された。中でも、人に焦点を当てつつビジネスの流動性にさらに俊敏に適応することを目指したいくつかの手法が、アジャイル型の開発方法論として提唱されている。

本調査では、このようなソフトウェア開発方法論の変遷を踏まえ、計画駆動型、プロトタイプ型、アジャイル型に加え、パッケージソフトウェアの適用をベースにした開発方法論であるパッケージ利用型の 4 つの開発型を対象として実施する。

(2) 本調査における調査範囲

「実践ソフトウェアエンジニアリング　ソフトウェアプロフェッショナルのための基礎知識（ロジャー S.プレスマン著、2005 年）」および「ソフトウェア工学　理論と実践（シャリ・ローレンス・プリーガー著、2001 年）」、「アジャイルと規律（バリーベーム、リチャードターナー著、2004 年）等を基に、ソフトウェアを開発するための開発型について定義する。

表 1　ソフトウェア開発型

開発型	定義
計画駆動型開発	開発プロセスを明確にし、要求定義／構築というパラダイムで開発する一連の手法を指す。
プロトタイピング開発	プロトタイプを作成し顧客の評価を受け、開発するソフトウェアの要求事項を具体的に検討する。この一連の手順を繰り返し、開発者が顧客のニーズを理解し取捨選択する一連の手法を指す。 理想的には、プロトタイプは要求事項を明らかにするためだけのメカニズムであるが、「最初のシステム」として役立てることも、破棄して本当のソフトウェアを開発することも可能な開発手法である。
アジャイル型開発	一般的な反復型開発と比較し、より俊敏にビジネス環境の変化に適応できるよう短期間でのイテレーションを繰り返す一連の開発手法を指す。
パッケージ利用開発	典型的には ERP、CRM 等のパッケージソフトウェアを利用し、パラメータの調整や一部機能の追加、変更のようなカスタマイズを行う一連の開発手法を指す。

また、上記の各開発型を実践するための具体的な開発手法が提唱されている。本調査では、各開発型に含まれる開発手法を以下のとおり整理した。表内において、本調査が参照した情報源を各脚注に示している。

表 2　各開発型に含まれる開発手法

開発型	開発手法
計画駆動開発	ウォーターフォール型開発 スパイラル開発 インクリメンタル開発・イテラティブ開発 Rational Unified Process　（RUP）[1] Enterprise Unified Process　（EUP）[2] SWAT2.0
プロトタイピング開発	Rapid Application Development（RAD）[3] Operational Prototyping（OP）[4]
アジャイル型開発	Lean Development[5]

[1] Craig Larman, *Agile and Iterative Development: A Manager's Guide*, (Addison-Wesley Professional, 2003) [訳: 児高慎治郎ら, 初めてのアジャイル開発　~スクラム、XP、UP、Evo で学ぶ反復型開発の進め方~, (日経 BP 社, 2004)]

[2] 株式会社オージス総研, EUP のベストプラクティス, http://www.ogis-ri.co.jp/otc/swec/process/eup-res/eup/essays/bestPractices.html

[3] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, (McGraw Hill Higher Education, 2004) [訳: 西康晴ら, 実践ソフトウェアエンジニアリング-ソフトウェアプロフェッショナルのための基本知識-, (日科技連出版社, 2005)]

[4] Davis, A.M., *Operational prototyping: a new development approach*, IEEE Software, 9(5), 1992

[5] Mary Poppendieck and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*, (Addison-Wesley Professional, 2003)

開発型	開発手法
	スクラム[1] Crystal Clear[6] Extreme Programming （XP）[1] Feature Driven Development （FDD）[7] Dynamic Systems Development Method （DSDM）[8] Adaptive Software Development （ASD）[9] Evolutionary Project Management（Evo）[10] セル生産[11] ユニケージ
パッケージ利用開発	フィットアンドギャップ分析

本調査では、表 2 に示した開発手法のうち、計画駆動型開発に含まれるウォーターフォール型開発を除くものを非ウォーターフォール型開発手法と呼ぶこととし、それらの手法を調査対象とする。

なお、本調査では、ここに示した開発手法レベルでの適用実態や有効性等よりも、主として各手法に含まれるもう一段細かい粒度での活動(プラクティス)や原則などに着目し、分析を進めるものとする。そのため、次節では各手法において実施されるプラクティスや原則などの性質に着目し、分類する。

３．２ 非ウォーターフォール型開発の特徴調査

３．２．１ 手法を構成するプラクティスの整理

３．１(2) にて整理した各開発手法には、それを実践するためのプラクティスや原則がそれぞれ示されている。そこで、非ウォーターフォール型開発手法がどのような特徴を有しているのかを概観するため、プラクティスや原則を要件、設計、実装、テスト／検証、プロジェクト管理、構成管理／変更管理等のソフトウェア開発における側面に分類する。

表 3 に、各種開発手法で提唱されているプラクティスを次のとおり分類している。本調査における分析対象とするものである。

- 要求
- 設計
- 実装
- テスト／検証
- プロジェクト管理
- 顧客との関係
- その他

---

[6] Alistair Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*, (Addison-Wesley Professional, 2004)

[7] Prentice Halltephen R Palmer, John Mac Felsing, *A Practical Guide to Feature-Driven Development*, (Prentice Hall, 2002)

[8] DSDM CONSORTIUM, *DSDM Consortium - Enabling Business Agility*, http://www.dsdm.org/

[9] James A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, (Dorset House, 1999) [訳: 山岸 耕二ら, 適応型ソフトウエア開発-変化とスピードに挑むプロジェクトマネージメント, (翔泳社, 2003)]

[10] Tom Gilb, *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, (Butterworth-Heinemann, 2005)

[11] 松本吉弘, ソフトウェア・セル生産方式の概要, http://www5d.biglobe.ne.jp/~y-h-m/Cell-based%20Software%20Production%20Overview.pdf

表 3　プラクティス整理表

要求		設計		実装		テスト／検証		プロジェクト管理																顧客との関係		その他	
								統合管理		スコープ管理		スケジュール管理		コスト管理		品質管理		組織・要員管理		コミュニケーション管理		リスク管理		外注管理					
Evo-2	重点要求トップ10を定義する	XP-4	システムのメタファ	XP-8	頻繁なリファクタリング	XP-6	テスト駆動開発	XP-1	リリース計画ゲーム	XP-2	イテレーション計画ゲーム	XP-3	小規模で頻繁なリリース			Scrum-10	1日以内の障害除去	XP-13	チーム全体が一緒に	Evo-18	クライアント駆動型計画	DSDM-32	リスク管理			Evo-1	利害関係者を見つける	Evo-19	価値のあることは何でも
Evo-3	機能仕様を定義する	XP-5	シンプルデザイン	XP-9	ペアプログラミング	XP-7	受け入れテスト	Scrum-1	ゲーム前計画	Scrum-2	スプリント計画	Scrum-3	原則30日（カレンダー上）のイテレーション			Evo-14	早期のインスペクションによる仕様品質のコントロール	Scrum-13	共通の部屋	C.Clear-1	短くてリッチなコミュニケーションパスがある	EUP-10	リスクを管理する			Evo-23	ユーザの権限	ASD-9	プロジェクトの事後評価
Evo-4	パフォーマンス仕様を定義する	Evo-10	設計仕様を定義する	XP-10	共同所有権	Scrum-15	スプリントレビュー	Scrum-6	スクラムミーティング	Scrum-7	イテレーションに追加してはならない	XP-12	持続可能なペース			FDD-5	インスペクション	Scrum-11	ニワトリとブタ	C.Clear-13	浸透したコミュニケーション					ASD-5	開発チームへの権限の付与	C.Clear-15	集中
Evo-5	明確で（可能なら）測定可能な仕様を定義する	Evo-11	影響評価	XP-14	コーディング規約	DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待され	Scrum-8	スクラムマスターのファイアウォール	ASD-1	ミッションを明らかにする	Scrum-9	1時間以内の判断			DSDM-35	品質管理	Scrum-12	7人のチーム	C.Clear-22	情報発信					DSDM-5	積極的なユーザー関与が絶対に必要である	C.Clear-23	方法論の改善
Evo-6	Planguageで仕様を記述する	Evo-12	設計アイデアがどう要求を満たすかを記述する	FDD-2	feature毎の開発	Lean-2	品質を作りこむ	Evo-7	進化型プロジェクトマネジメント	ASD-2	ミッションを文書化する	Scrum-4	スプリントバックロググラフの作成			Lean-3	知識を作り出す	Scrum-5	自律的な組織化チーム							DSDM-13	すべての利害関係者間の協力が不可欠で	C.Clear-29	プロセスの簡易版
DSDM-2	MoSCoW	Evo-13	テストを測定基準を要求仕様に記述する	FDD-3	クラス毎のオーナーシップ	ASD-7	顧客のフォーカスグループレビュー	Evo-8	進化型出荷	ASD-3	ミッションが表す価値を共有する	Evo-21	頻繁な出荷			ASD-4	結果に焦点を合わせる	Evo-24	測定と褒賞									EUP-8	開発後のことを考慮する
DSDM-3	プロトタイピング	Evo-16	オープンエンドのアーキテクチャ	ASD-6	JAD開発	SWT-5	ユーザ動作確認	Evo-9	出荷したソリューションの影響を測定す	RUP-2	要求管理	DSDM-1	タイムボックス			ASD-8	ソフトウェアインスペクション	FDD-4	feature チーム										
DSDM-11	要件は高いレベルで基準化される	Evo-17	安全係数	RUP-3	コンポーネント・アーキテクチャの使用	EUP-5	継続的に品質を検証する	Evo-20	迅速な学習	C.Clear-19	360度の探索	DSDM-7	頻繁な引き渡しが鍵である			RUP-5	品質の継続的検証	DSDM-6	チームに引き渡しの権限が与えられなければならない										
DSDM-36	実現可能性調査	Evo-22	問題の分割	C.Clear-5	作業成果物の所有者を決める			FDD-1	ドメイン・オブジェクト・モデリング	SWT-2	タイムボックス計画	DSDM-8	受け入れの主たる条件は現在のビジネスニーズを満たす機能の引き渡し			SWT-6	ピックアップレビュー	Cell-5	セルの構築と独立性										
DSDM-37	ビジネス調査	Cell-2	DSMを用いたアーキテクチャ変換	C.Clear-30	Side-by-Sideプログラミング			FDD-8	目に見える進捗と成果			DSDM-9	反復的で漸増的な引き渡しが不可欠である					Cell-6	専門技術者の育成										
Cell-4	ビジネスとソフトウェア構築の直結	RUP-4	ビジュアル・モデリング					DSDM-4	ワークショップ			C.Clear-2	1～3ヶ月以内に出荷する					Cell-7	セルのスキル割り当て										
C.Clear-4	プロジェクトの要諦を把握する	C.Clear-20	実行可能なスケルトン					DSDM-31	プロジェクト計画			C.Clear-11	頻繁なリリース					Cell-8	セル内の平等責任										
C.Clear-16	エキスパートユーザとのコミュニケーショ	C.Clear-21	インクリメンタルなアーキテクチャの再設計					DSDM-33	DSDMプロジェクトの計測			C.Clear-27	毎日のスタンドアップミーティング					C.Clear-3	真のユーザをプロジェクトに巻き込む										
SWT-4	要件確認会	C.Clear-28	アジャイルインタラクション設					DSDM-34	見積もり			C.Clear-31	バーンダウンチャート					C.Clear-14	個人の安全										
EUP-2	要求を管理する	EUP-3	実証されたアーキテクチャ					DSDM-38	機能モデルイテレーション			SWT-1	タイムボックス					EUP-7	協調的な開発										
		EUP-4	モデリング					DSDM-39	設計・構築モデルイテレーショ			SWT-3	80%ルール																
								Cell-1	シェフモードとコックモードの分離			EUP-9	動くソフトウェアを定期的に納品する																
								Cell-3	資産の活用																				
								ASD-5	適応型サイクルの導入																				
								RUP-1	反復型開発																				
								C.Clear-12	反省と改善																				
								C.Clear-24	反省ワークショップ																				
								C.Clear-25	集中的な計画																				
								C.Clear-26	デルファイ見積もり																				
								EUP-1	反復的に開発する																				
								XP-11	継続的インテグレーション																				
								Scrum-14	日次ビルド																				
								Evo-15	仕様の関連																				
								FDD-6	構成管理																				
								FDD-7	定期ビルド																				
								DSDM-10	プロジェクトライフサイクル中に行われた変更はすべて元に																				
								RUP-6	変更管理																				
								C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションnおける技術的な環境																				
								EUP-6	変更を管理す																				

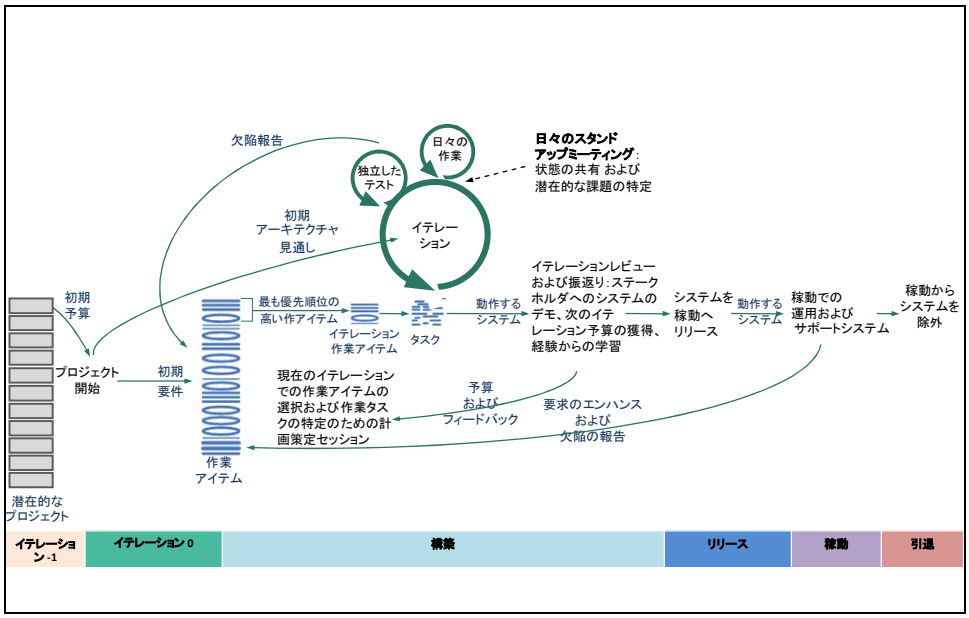
（備考）各プラクティスの説明は、付録を参照。

表中で用いている略号が表す開発手法は、それぞれ次のとおり。

ASD：Adaptive Software Development　**Cell**：セル生産　**C.Clear**：Crystal Clear　**Evo**：EVOlutionary project management　**EUP**：Enterprise Unified Process　**DSDM**：Dynamic Systems Development Method　**FDD**：Feature Driven Development　**OP**：Operational Prototyping　**Lean**：Lean Development　**RAD**：Rapid Application Development　**RUP**：Rational Unified Process　**Scrum**：スクラム　**SWT**：SWAT2.0　**Uni**：ユニケージ　**XP**：eXtreme Programming

３．２．２ 非ウォーターフォール型開発手法を適用した場合の開発プロセス

一般にアジャイル型の開発手法は、適合的に生起する部分要求に対するソリューションを、決められたタイムボックス制約およびコスト制約のもとでコード化して、テストし、すでに運用中の先行システムに継続的に統合して運用することを、繰り返しながら、全要求を満たすシステムを構築する。



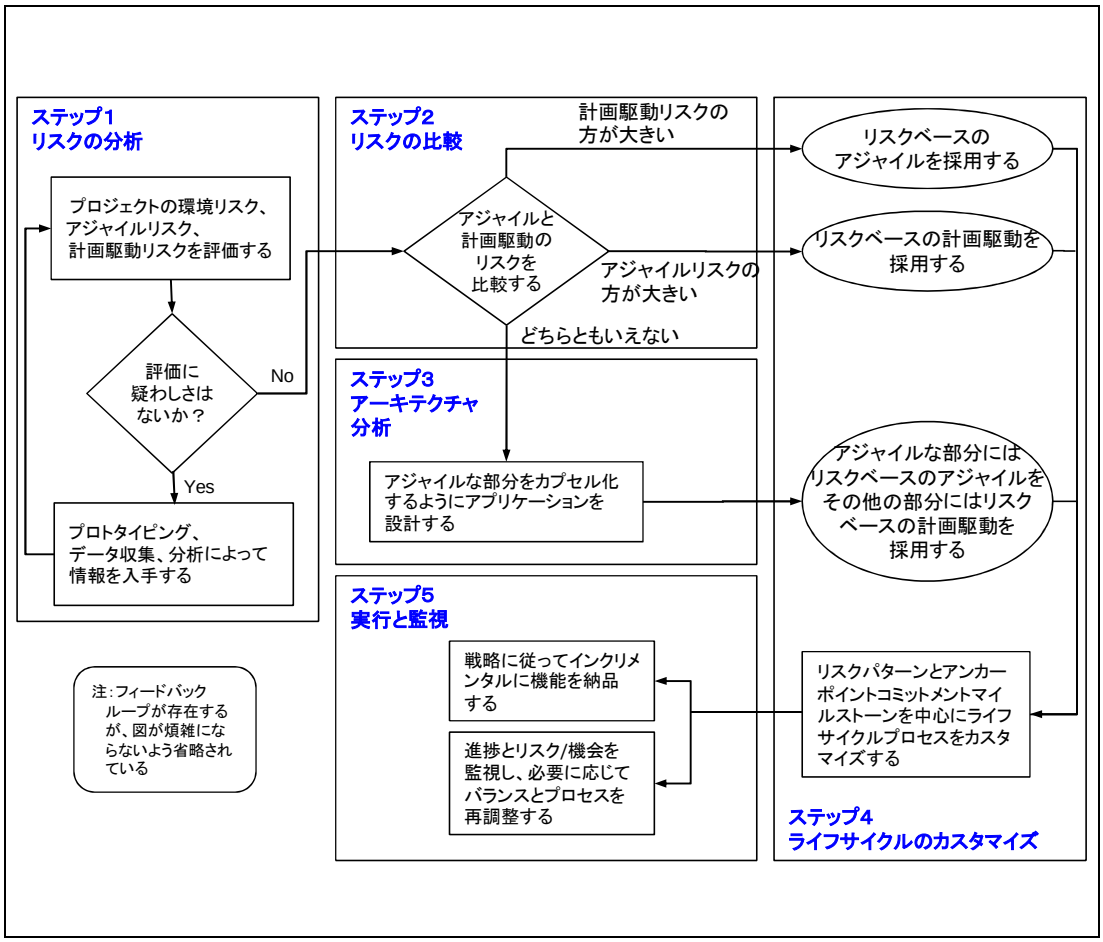
[出典] Ambysoft, *The Agile System Development Life Cycle (SDLC)*, http://www.ambysoft.com/essays/agileLifecycle.html をもとに MRI 作成

図 1　アジャイル型開発手法を適用した典型的な開発ライフサイクル

しかし、実際の開発現場では、「全くのアジャイル型、そうでなければ全くの計画駆動型」というような「黒でなければ白」という考え方に基づく手法選択より、何らかの比率で両手法を組み合わせる方が現実的である。そこで、アジャイル型開発手法と計画駆動型開発手法を組合せる手法の一例として、プロジェクトにおけるリスクに着目した方法を示す。

(1) プロジェクト始動後、初期の段階で、プロジェクトの進行過程上に、計画的にアンカーポイント・マイルストーン（以下、チェックポイントと略称する）を設定する。
(2) チェックポイントに到達すると、つぎのチェックポイント至るまでのプロセスに関するアジャイルリスクと計画駆動リスクを分析し、評価する【ステップ 1】。
(3) 評価の結果に従って、つぎのチェックポイントまでを、アジャイル型にするか、計画駆動型にするかを決定する【ステップ 2】。両リスクが拮抗して判断できない場合には、アジャイル部分をカプセル化して、ソフトウェアアーキテクチャを同定する【ステップ 3】。
(4) ステップ 2 の判定に基づいて、アジャイル型プロセスか、計画駆動型プロセスかのどちらかを実行する。ステップ 3 に進んだ場合には、カプセル化した部分はアジャイル型プロセス、それ以外の部分は計画駆動型プロセスを実行する。実行後、全ライフサイクルを見直して、チェックポイントを再調整する【ステップ 4】。
(5) ステップ 4 で生成した成果物をテスティングし、必要であれば、リリースし、先行シ

ステムに継続的に統合する。次のチェックポイントから、再びステップ１に戻り、同じ手順を繰り返す。



[出典] Richard Turner, *Using CMMI to Balance Agile and Plan-driven Methods*, CMMI Technology Conference, 2003 をもとにMRI作成

図 2　アジャイル型開発手法と計画駆動型開発手法の組み合わせ方

３．２．３ パッケージ開発の特徴

本項では、非ウォーターフォール型開発のうち、パッケージ利用開発に関する特徴についてまとめる。

なお、パッケージ利用開発以外については、３．３項以降で多くの事例等に基づいて詳細に特徴を示す。

(1) パッケージ利用開発の概要

パッケージ利用開発のうち代表的なのはいわゆる ERP（Enterprize Resource Planning）パッケージと呼ばれる企業全体の経営資源を有効に活用することを目的に統合的に管理し、経営の効率化を図るためのパッケージを活用して、情報システムを構築するものである。

ERP パッケージとしては、ドイツ SAP 社の R/3、Oracle 社の Oracle Applications などがある。基本的に、ビジネスでの標準的なプロセスをあらかじめパッケージの標準機能として用意し、様々なビジネスコンテキストに対応するべく、パラメータ等でプロセスの

並びや実現機能を調整したり、また、新たな機能をカスタマイズとして作成し、組み入れることを可能としている。

標準機能が活用できればできるほど、効率的に、短期間で情報システムを構築することができるが、実現したい業務プロセス・機能の洗い出しとパッケージがあらかじめ用意している標準機能との対比のプロセスが必要となる。実現したい業務プロセス・機能の洗い出しは、他の開発でも重要な要件定義と同じである。対比プロセスは、フィットアンドギャップ分析と呼ばれ、パッケージ利用開発における特徴的なプロセスである。

(2) パッケージ利用開発の具体的な特徴

パッケージ利用開発における特徴は、要件定義における上記のプロセスが特徴的であるが、実際の開発における着目点等の具体的な内容を示すために、非ウォーターフォール型の代表的な方法であるアジャイル的なアプローチとの対比を通して説明を行う。

以下では、カナダの鉄道会社であるカナディアン・パシフィック（CP）での社内会計システムの更改の事例で ERP パッケージである SAP を活用した報告に基づいて、特徴をまとめる。

このシステム更改は、引合いから契約、注文・支払いまでの経理プロセスの効率化と現状のシステムを SAP で更改することであった。目的は、使用性を上げて会計部門の生産性の向上と営業への注力を可能とすることである。

そして、効率的に情報システム開発を実現するために、パッケージ開発とともに、アジャイル的な取組みを加味することが試みられたものである。

(a) パッケージ開発とアジャイル開発への期待

前提としては、現状の機能より使いづらいものや足りないことは許されない、つまり、SAP で実現できないとの理由で機能の優先順位を落とすことはない状況で開発が開始されたものである。実際には、機能にギャップがあったが、SAP が標準に持っている機能は、SAP のコードからもカスタムのプログラムからも呼び出せるようになっている。そのため、SAP の言語 ABAP を使って、既に出来上がっている機能を利用して現場のニーズに合ったフロントエンドを作成することとなったものである。

その時の期待は、次のとおりである。

- 完全に動く標準化されたスタンドアローンシステムであることで、動かしながら様々な機能の追加や動きのインテグレーションをしていくことができると期待。
- また、アジャイル的な活動を助けるものとして、Recorded Test ツール、eCATT と xUnit テストフレームワーク、ABAPUnit が準備されていることもアジャイル的な取組みをサポートすると期待。

(b) パッケージ開発とアジャイル開発との対比

一方、現実には、以下に示すような SAP（ERP）開発の特性（文化を含む）とアジャイル的な取組みとの間で相性の課題はあり、個別に解決を試みられている。以下に、パッケージ開発の特徴を、対比を通してまとめる。

- パッケージ開発の特徴として、プロダクトが先にあることの影響
  - SAP はある特定のプロセスを中心にサポートするものとなっている。基本的には、様々なヴァリエーションをパラメータ設定で変えることで、

組織のビジネスにあったITシステムとするという作りになっている。

- このプロダクトが先にあるという思想は、ユーザ側の要求に合わせるというアジャイルとややなじめない。これに関しては、SAPが"User Exits"と呼ばれるメカニズムを用意しており、既存のアルゴリズムを上書きすることができるようになっている。

➢ SAP利用開発における専門化された役割

- SAPはデータ駆動であり、構成を自由に変えられる（Configurableと表現される）。データは、トランザクションデータ、マスターデータ、コンフィグレーション･データ（SAPの動きを制御）、ABAPコード自体もデータとして格納され、実行時に呼び出される。
- これらの機能を使いこなすには、充分な経験と知識が必要であり、Functional Analystと呼ばれる要求とSAPの標準機能とのギャップを分析し、SAPでの実現方法を検討するメンバとカスタム部分をABAPで作成するABAPプログラマの確保が基本である。しかしながら、Functional Analystの役割として、全体を見ているというよりは、個別の機能の実現を見ており、ソリューション全般を把握するという立場ではないと見られる。ABAPプログラマはさらに規定された要求をカスタムで作成するという役割が明解である。
- アジャイル開発等において、ジェネラリスト的な役割を果たすことが求められることと比較して、特徴的といえる。

➢ 協調のコミュニケーションの文化の課題

- 協調と対面のコミュニケーションはアジャイルで重要だが、SAPコミュニティではやや難しいものがあったと報告されている。伝統的にSAP利用の開発ではドキュメント駆動である。コンフィグレーションならコンフィグレーション、コード開発ならコード開発でやり取りがドキュメントでなされ、分断されている。また、ユーザの言葉よりもSAP専門用語が飛び交う。
- 本事例では、アジャイル的に開発を進めるための工夫として、ペアプロや、ストーリごとにコンフィグレーションと開発者が横に並ぶといったことがなされている。

➢ サーバーベースの開発

- SAPはサーバーベースの開発であり、すべてのものは一元管理される。
- SAPの「利益」の一つは、完全に統合されたシステムであること。Reworkをしないですべてを見通せるということ。一方、アジャイルではある程度、柔軟に対応することが基本であり、例えば、全ての機能の可能性を残しつつ作成することを行う。これは、統合システムとしてカチッと決まっていることとやや相反する考え方である。

➢ パッケージ開発とアジャイルとの組合せ

- 結果的に、いわゆるパッケージ開発にとってトラディショナルな部分は残して独立に実施することで、アジャイルの活動と組合せて実施が続けられている。
- 実際、アジャイル的な活動の組合せ例として、次のものが示されている。

    - イテレーション計画ミーティングで始まり、レトロスペクティブで終わる。
    - 人手でユーザストーリごとにストーリテストを作成し、ABAP Unit で単体テストを書いている。
    - スモークテストの自動化
- なお、最も大きな課題は、文化の問題が結論として示されている。SAP とアジャイルでは、開発に対するマインドセットが違う部分があるとされ、特に、ドキュメント駆動的なところが最もアジャイル・フレンドリでないとされている。

３．３　文献・Web およびヒアリングによる事例調査

３．３．１　非ウォーターフォール型開発手法の適用事例

本調査では、非ウォーターフォール型開発手法を適用して実施された開発事例について、文献・Web およびヒアリングによる調査を実施した。その結果、収集された事例の特徴を把握しやすくするため、以下に示す 2 つのカテゴリに分類した。

- カテゴリ 1
  社内の業務・生産効率化、対話・知識流通、教育、実験などを目的とし（顧客が社内部門）、かつ継続的インテグレーション環境が、実行系フレームワーク・プラットフォームから分離され、独自に設定されているもの。
- カテゴリ 2
  カテゴリ 1 に含まれないもの。例えば、次のようなシステムの開発事例。
  - 外販目的のサービス、パッケージ、アプリケーションシステム
  - 社内向けでも、サーバなどに実装されている実行系フレームワーク・プラットフォームに直結されているもの

また、各事例に登場する利害関係者の関係性や関与の仕方によって、それぞれ図 3 に示す 5 つのタイプに分類した。

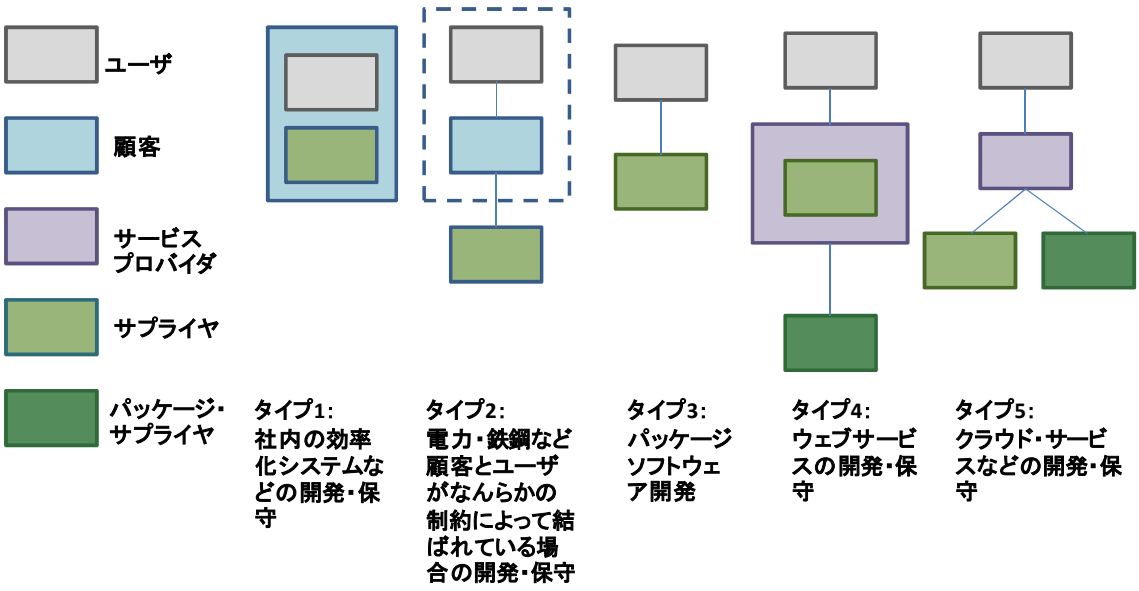


図 3　利害関係者の関係に着目した事例の分類方法

表 4　利害関係者の定義

利害関係者	定義
ユーザ	開発されたシステムの利用者
顧客	システム開発の発注者
サービスプロバイダ	開発されたシステムによって実現されるサービスの提供者
サプライヤ	システムの開発者
パッケージサプライヤ	システム開発に利用するパッケージソフトウェアの提供者

ここでは、本調査において収集された各事例をカテゴリ別に列挙し、その概要について説明する。

(1) カテゴリ 1 に属する事例

(a) 小売業における業務システム開発事例（利害関係者関与：タイプ 1） 【A 社】

■ 事例概要

小売業を営む顧客のマーチャンダイジングシステム開発を出発点に、これまでにシステム全体で 160 メニュー、約 8 万ステップの開発。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	開発当初の目的は、既存システムの完全リプレイス（システムの Web 化対応）。
ライフサイクルモデル	• ユーザヒアリング 初めに顧客業務を理解するため、ユーザヒアリングを実施する。この段階で、現行システムを理解し開発すべきシステムの姿を描く。 • データ設計 次に、データ配置の設計および技術的な検証を実施する。 • 構築 プログラム開発（1 次）を行った後、機能アップのヒアリングを実施し、プログラム開発（2 次）を行う。 ［1 次開発および 2 次開発は 1～2 週間、テストは 1 週間程度である。1 次開発と 2 次開発の間はおよそ 1 ヶ月間あり、ユーザは開発されたソフトの使用感を実感してバージョンアップ要求を出す。］
チーム編成	• ベテラン開発者 1 名と新人 5 名 • 顧客側も同数程度のプロジェクトメンバを配置 ユーザと開発者の人数が半々になるように、ユーザ側にもプロジェクトチームを立ち上げる。これは、業務とシステムは一体であり、システムはユーザと開発者が共同で作るという意識を持って開発を進めるための取組みである。(A 社が開発を引き受ける際の 1 つの条件)
プロジェクト期間	6 ヶ月
プロジェクト初期における要件の確定度合い	データベースのレイアウトやアルゴリズムなどについての仕様はなく、外部仕様レベルでの基本要件が固まっていた状態であり、まったく新たに要件を構築していくということではない。

契約形態	請負契約ではなく、顧客業務（顧客による開発）の支援という形態をとり、成果物責任は顧客側が持つ形態をとった。ただし、実態としては、何々の実現を支援するという形式になるため、開発するシステムの難易度および量によって契約額が固まる。

■ 特徴

- ユニケージ手法という独自の開発手法によってシステムを構築している。ユニケージ手法とは、プログラムは全てシェルスクリプトによって構成するとともに、データは全てテキストファイルにて管理するという考え方に基づく開発手法である。
- システムを、販売管理や会計管理のようなシステムに対する要求があまり変化しない領域（実行系）と、販売計画や在庫管理のような日々の業務遂行によりシステムに対する要求が目まぐるしく変わりうる領域（情報系）とに分割して開発している。前者は他社による従来型の開発手法によって構築され、後者をユニケージ手法によって構築している。
- 要求の確認は文書に頼るのではなく、ユーザとの会話によるコミュニケーションと実際に動作するプログラムを用いて行うという考え方を徹底している。
- 要求は実際にシステムを使用するユーザから汲みあげるが、顧客にとって重要な要求のみを抽出するため、システム化するかどうかは会議によって決定する。
- プロジェクトの初期段階でユーザに対してプログラミングの教育を実施することで、システムリリース後の保守・運用をユーザ自身で行えるようにする。これは、ユニケージ手法がシェルスクリプトとテキストファイルという極めてシンプルな構成によって成り立っていることが大きい。

(b) ソーシャルネットワーキングサービス（SNS）システム開発事例（利害関係者関与：タイプ 1→4） 【B 社】

■ 事例概要

社内において技術情報を共有するためのコミュニケーション基盤として、SNS を開発。社内での実利用を経て、現在では SaaS によるサービス提供も行う。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	経営的には社内における技術情報の共有の実現ニーズがあり、技術者の中には技術情報を提供するための場に対するニーズがあったため、両者のニーズを同時にみたす仕組みの構築を志向。 社内システム開発段階においては、開発スピードとフィージビリティを優先した。素早く確認しながらの開発。

ライフサイクルモデル	社内開発時は随時リリース。SaaS 事業での開発時には、2 週間のイテレーションを繰り返し、1 ヶ月に 1 度リリースする。 ・2005 年 11 月から開発開始。 ・2008 年 5 月ごろにオープンソース化。 ・2008 年 11 月から SaaS ビジネス開始。
チーム編成	PM1 名、開発者 2 名。
プロジェクト期間	社内開発の最初のリリースまでは 2 ヵ月
プロジェクト初期における要件の確定度合い	技術情報を共有するということが決まっていただけで、システムに対する要件は全く固まっていなかった。
契約形態	SaaS の場合は、月々の利用料のみ。サービスのカスタマイズに応じるプレミアムユーザとは、準委任（コンサルティング契約のような形態）契約である。アジャイル型の開発の場合、従来型契約ではうまくいかないだろう。

■ 特徴

- 社内にて技術情報を共有するというテーマしか固まっていない段階から開発を始めなければならなかったため、開発者がイメージしたコンセプトをプログラミングし、完成したものを早期にリリースしてユーザからのフィードバックを受けてシステムとして成長させていった。
- 上がってきた要望に対して、要望数の多さやユーザに対するインパクトの大きさなどを基準として優先順位付けし、優先順位の高いものから順に実装した。
- PM が、各メンバが毎日どれだけのことができたのかを把握することにより、進捗を把握した。
- 開発開始段階でのメンバには、今回利用した開発言語に対する経験が全くなかったため、開発を進めながら学習していった。
- 途中段階から開発に加わったメンバには、ペアプログラミングを通じて技術および情報移転を図った。
- SaaS 型のビジネスに発展させたこともあり、単に要求されたものを作ることができるというのではなく、何を作るべきかの提案ができることが重要となっている。

(c) サプライチェーンマネジメントシステム開発事例（利害関係者関与：タイプ 1）【C 社】

■ 事例概要

本事例は、個別事業を対象としたものではなく、本事例を実施した企業における標準的な取り組みを説明するものとなっている。

■　プロジェクト基本情報

項目	内容
優先した IT 戦略	低コストや要求の変化への対応を優先した。
ライフサイクルモデル	キックオフ→2～4 週間のイテレーションの繰り返し→完成形。リリースした後も変更を続けて行く。 （通常はイテレーション 2 週間×3 回。2 ヶ月程度でのリリースが多い。規模が大きいと 4 週間×4 回である。）
チーム編成	3～5 人。アプリケーションマネージャ（アカウントエグゼクティブ）が 1 名。ファシリティーズ（インフラレベルの統括）が 1 名。両方に紐づく形でアーキテクト（2～4 名）
プロジェクト期間	2 ヶ月程度の事例が多い。
プロジェクト初期における要件の確定度合い	分析・設計から入ることが多い。要件を決めてもイテレーションの最中に変わることもある。我々からの提案もある。
契約形態	サービスの利用料金がビジネスの基本単位である。スクラッチ開発とほぼ同様のレベルでフルカスタマイズできる ASP サービスであるが、契約自体は賃貸借契約に準ずる ASP 契約である。初期開発費用はもらわず、月額の利用料で回収する。 料金の指標は、アクセス数や帳票の数、外部接続の数といくつかあるが、主要なものはテーブル数である。 サービスの利用は最短で 2 ヶ月で止めることも可能。最初にフル装備で構築したが、実際に利用されない機能があった場合、その機能(テーブル)を削ることで月額料金を下げることが可能。 今まで失敗プロジェクトは 1 件もない。リリースしたが 1 年後にサービスを停止することになった、ということは実績としてある。政治的要因によりユーザのシステムに対するニーズが実際よりも過大評価され、その評価に基づきプロジェクトを開始してしまったことに原因があったものと分析している。

■　特徴

- ユーザからシステムの開発費は取らず、開発したシステムによって提供されるサービスの利用料を受け取るというビジネスモデルである。
- 解決すべき課題を合意し、プロジェクトステートメントを定める。これに従っ

て目的達成のために解決策を両者で模索することをプロジェクト推進の基本スタンスとする。

- イテレーションにおいてユーザが細かい操作性にこだわり、議論が発散してしまうケースも存在するが、その場合一旦持ち帰り、我々が最適と想定するインタフェースを後日改めてプロトタイプとして実際に操作可能な形で提示することで、議論を収束させると共に納得感・信頼感を醸成することができている。
- 開発者には非常に高いスキルが求められるため、徹底した社内研修を行い、スキルアップを図っている。例えば、業務知識の獲得および開発スキルの向上のため、ウェブアプリケーションによる財務会計システムの個人ベースでの構築を研修として実施している。

(d) 研修運営システム開発事例（利害関係者関与：タイプ 1） 【D 社】

■ 事例概要

RUP をベースとした独自の反復型開発プロセスにより、システム開発を行った。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	要求の変化への対応、使用性を優先した。
ライフサイクルモデル	基本構想（1 ヶ月・2 人月） 方向付け（2 ヶ月・7 人月） 推敲（3 ヶ月・30 人月 反復 3 回） 構築（4 ヶ月・55 人月 反復なし） 移行（1 ヶ月・10 人月）
チーム編成	推敲まではアーキテクチャレイヤー別（Web、BL、EIS 等）でチーム構成 構築以降はサブシステム別に構成変更した
プロジェクト期間	10 ヶ月
プロジェクト初期における要件の確定度合い	業務プロセス未整備
契約形態	基本構想から方向付けフェーズまでは準委任契約 遂行フェーズから移行フェーズまでは請負契約

■ 特徴

- 反復開発の経験がない開発者がほとんどであり、プロジェクト初期段階で、プロセス説明に時間を費やした。
- 要求変更を含む要求管理は、一括して PM 指揮下でのコントロールとした。
- 外注要員を、アーキテクチャを構築するハイスキルな集団と、開発ガイドに沿ってアプリケーション開発を行う集団に階層化した。
- ユーザのユースケースやモデルに対する理解が深まらなかったため、画面ベースのプロトタイプ作成を行うことで、要求仕様を固めた。

(e) 開発案件管理 Web アプリケーション開発事例（利害関係者関与：タイプ 1） 【E 社】

■ 事例概要

顧客システムの保守案件を管理する Web アプリケーションソフトウェア開発。このシステムのユーザは、別のシステムの保守要員および顧客の情報システム部門である。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	特になし
ライフサイクルモデル	（不明）
チーム編成	入社 3 か月の若手 3 人。ベテラン 1 人。 特定のメンバが決まった画面を担当するのではなく、作業負荷に応じて柔軟に対応する。 メンバの増員・入れ替えは無い。
プロジェクト期間	3 ヶ月
プロジェクト初期における要件の確定度合い	非常に低い
契約形態	新人育成を目的に実施されたプロジェクトであり、契約関係はない。

■ 特徴

- 入社 3 か月の若手メンバ 3 人とベテラン 1 人のチーム構成であった。
- 開発メンバも積極的にマネジメントに参加し、自律的にチームを運営した。
- メンバが 1 プロジェクトを通じて経験・成長することが、1 イテレーションに集約されているため、イテレーションを繰り返すほど人・ツール・品質も成長した。
- 顧客システムの開発・保守を実施するサービスマネージャーを顧客プロキシに設定した。プロジェクトマネジメントの経験も豊富であり、1 回／週の計画ゲーム・ふりかえりに参画した。
- メンバの間のコミュニケーションを効率的に行い、円滑な進捗を図るために、ファシリテータという役を設けた。ファシリテータ役を担うメンバは日替わりとし、朝会、計画ゲームを主宰する。
- 計画ゲームの段階で、タスクかんばんによって要員の負荷を平準化した。
- 学習時間をリスク分として見込み、作業時間と分けて見積もった。

(f) 製造業向けプロトタイプシステム開発事例（利害関係者関与：タイプ 1） 【F 社】

■ 事例概要

アジャイル型開発未経験者のみによる実践事例。XP のプラクティスをベースとし、製造業向けのプロトタイプシステムを開発。

■　プロジェクト基本情報

項目	内容
優先した IT 戦略	納期より機能スコープを優先（イテレーションが終了すると、すべての顧客要求が実装されていないものの、優先順位が高い機能は実装され、動作可能なソフトウェアが出力される）
ライフサイクルモデル	期間は通常、1 週間～1 ヶ月。イテレーションの期間は、初回と同様の期間が設定されることが多い。理由は、同一の期間を繰り返すことで、チーム内に期間内の開発リズムを体得させる効果をねらっているため。 タスクの実施から受入テストによるストーリの完了が、イテレーション期間内で毎日繰り返される。 本事例は、初めにトライアルとして 5 日間の第 0 イテレーションと 2 週間×3 回のイテレーションを設定。
チーム編成	カッコ内は、それぞれソフトウェア開発経験とオブジェクト指向開発経験年数。 X 氏：トラッカー；お客様プロキシ（9 年、8 年） Y 氏：マネージャ・コーチ（5 年、3 年） Z 氏：コーチ（3 年、2 年） A 氏：開発者（3 年、1 年） B 氏：開発者（3 年、1 年） C 氏：開発者（0 年、0 年）新人 D 氏：開発者（2 年、1 年） E 氏：開発者（4 年、1 年） F 氏：開発者（14 年、7 年） E 氏と F 氏は途中参加。 非ウォーターフォール型開発はみな未経験。
プロジェクト期間	約 2 ヶ月
プロジェクト初期における要件の確定度合い	（不明）
契約形態	（不明）

■　特徴

- プロセス間のコラボレーションは、「後工程引取り」のコンセプト（顧客の要求する機能の完了条件となる受入れテストによって実装が駆動される）で実施。
- 明確になっていた要求は全体の 50%程度。開発工程期間中に散発的に到着。途中、消滅したものも多い。
- 開発側に「お客様プロキシ」を設置（顧客要求を代表するチーム内の代理人。開発メンバ内で、より顧客の立場を理解しているメンバが当該役割を担う）。
- 「ジャストインタイム」のコンセプトで、顧客によって優先順位付けされた機能を順次実装（YAGNI：You Aren't Going To Need It）

- XP のプラクティスである「ミラー」で、目で見る管理を体現。ストーリカード、タスクカード、バーンダウンチャート（バックログ数のグラフ）を壁に貼り、進捗状態を開発者がリアルタイムに見られるように工夫。
- 作業者が複数の作業スキルを備える「多能工化」と作業者ごとの作業時間をならす「平準化」を原則とする。
- 顧客要求は、ストーリと呼ばれる単機能に整理。作成されたストーリは、計画ゲームでタスクの単位に分割。タスクは開発者から見た実装単位。
- 計画ゲームで分割されたタスクは毎朝実施されるスタンドアップミーティングで、開発者に割り当て。管理者によるトップダウンではなく、開発者自らの志願によることが多い。
- 開発では、ペアプログラミングを実施。ペアは毎日交代することが望ましい。ペア交代の連鎖により、知識の伝播を促進し、学習効果を高めることが目的。
- プログラムコードと同時に、テストプログラムコードを作成（コード作成とと同時にバグ摘出･修正)。ストーリを構成するタスクがすべて消化されたとき、ストーリの完了テストを実施。
- 受入テストは、顧客又はお客様プロキシが実施。
- 自動化された回帰テスト（最長 1 日でバグは検出される)。
- 保守性を高めるため、リファクタリングを実施。

(2) カテゴリ 2 に属する事例

(g) 携帯ソーシャルゲーム開発事例（利害関係者関与：タイプ 4） 【G 社】

■ 事例概要

携帯端末向けのソーシャルゲームサービスを提供するために必要となるシステムを自社にて開発。これまで多くの Web サービスをアジャイル型の開発手法によって開発してきた企業によるもの。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	スピード、要求の変化への対応、利用率を優先
ライフサイクルモデル	企画～プロトタイプ～α版開発（1.5 ヶ月）->社内グループインタビュー～α版改修（0.5 ヶ月）->社外グループインタビュー～β版開発～クローズドβ公開（0.5 ヶ月）->β版改修～全展開（0.5 ヶ月） イテレーション期間：1～2 週間 サービスイン後も継続的に Day 単位～Week 単位で改版を継続。
チーム編成	企画 1 名 エンジニア 1 名 ※どちらも自走できるレベル。 後半、業務系開発にベンダ 1 名追加
プロジェクト期間	3 ヶ月～継続中

プロジェクト初期における要件の確定度合い	ゼロベースからの検討を要した
契約形態	社内開発であるため、契約関係はない。

■ 特徴

- 社内では、ウォータフォール型開発が主体となるものと非ウォータフォール型開発が主体となるものの 2 種類のシステムを開発し活用している。
- 適用する開発手法の区別は、社内規定に従って判定されるというものではなく、案件ごとの特性や制約などによって、自然と適した方が選択されるイメージに近い。傾向としては、利害関係者が多く、より高い信頼性、正確性が重要となるような開発の場合にウォータフォール型が選択されることが多く、より早期にリリースすることが重要な開発の場合に非ウォータフォール型が選択されることが多い。
- 非ウォータフォール型開発における成功体験をウォータフォール型開発へ移転し、非ウォータフォール型開発の長所をウォータフォール型開発へも取り込むようにしている。
- 非ウォータフォール型開発を支援するため、共通利用可能なソースコード群からなる独自のフレームワークを構築している。フレームワークへのニーズには、メンテナンスの際に対応する。
- 個別案件の実施中にフレームワーク自身の変更、修正は行わず、現在あるフレームワークを用いてできることを実施するというスタンスで開発を行っていく。
- 開発チームには、かならず 1 名以上、自律的に開発を推進し、課題に対処できるスキルを持ったメンバをアサインする。
- 技術的な経験が十分ではない若手でも、問題を察知する能力に長けているエンジニアには積極的に開発を任せるようにしている。
- エンジニアにも企画を提案することを求めている。そのため、エンジニアには、自身が開発したシステムの運用も任せている。

(h) 携帯端末向けブログシステム開発事例（利害関係者関与：タイプ 4） 【H 社】

■ 事例概要

既存のブログシステムに対する性能面の問題から、全面的に更改を行った事例。開発するシステムに対する基本的な要件は固まっているものの、新たに追加していく機能に対する要件は全く未知の状態であった。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	スピード、要求や市場の変化への対応力、信頼性、利用率、モバイル／ウェブクライアント重視、OSS 重視
ライフサイクルモデル	1 イテレーションを 1 週間と設定し、既存サービスの再構築を 4 ヶ月（14 人月）で行い、β サービスリリー

	ス後はサービス終了までの 15 ヶ月間、サービスとしての新機能・運用ツールやバグフィックスのリリースを毎週行った。
チーム編成	当初開発は 3 名体制だったが、予定していた人員の補強が出来ずにマネージャが開発に入り、4 名体制で開発を行った。その後、内製化を行うために 1 名増員を行い計 5 名で開発を行った。また、企画会議には開発者・運用担当（3 名）・企画担当（1 名）・開発マネージャも参加して、全員でサービスを作っていった。
プロジェクト期間	1 年 7 ヶ月
プロジェクト初期における要件の確定度合い	既存のαサービスのリプレイス部分の基本的な機能の要件はほぼ固まっていたが、新機能に関しては、企画段階の物が多く要件としては全く固まっていなかった。
契約形態	月ごとの請負契約

■ 特徴

- 週次で開発マネージャを含めて計画ゲームを行い、次回のリリース計画を作成した。タスクカードはかんばんを用いて運用した。
- テスト駆動開発による継続的インテグレーションを実施。運用上の不具合等は全て ITS に登録することで管理を行い、開発マネージャの修正確認後 Fix とするような運用を行った。リリースサイクルが 1 週間と短いためステージング環境で 1 週間テストを行った後にリリースを行うようにした。
- Flash チームと Server チームを分割した。後発で増員されたメンバにはペアプログラミングを行い知識の共有を行った。
- 当初は客先に常駐することにより全員でチームだという環境を作り、チームの方向性が一致してからはリモートで Skype を用いてコミュニケーションを図った。また、週次で定例ミーティングを開催し、週に 1 度以上は全員が顔を合わせる機会を設けた。
- 週次で振り返りを行い、各自が抱えている不安やリスク等を洗い出し、対応計画を立てた。
- 開発初期から継続的インテグレーションを利用した。また、毎週リリースを行う上で、品質を確保するためにステージング環境にリリース対象をデプロイした後、1 週間かけてテストを実施してからリリースを行った。

(i) パッケージソフトウェア開発事例（利害関係者関与：タイプ 2）　【H 社】

■ 事例概要

ワークフローを管理するためのパッケージソフトウェアの基盤となるエンジン部分を開発。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	要求の変化への対応、信頼性、利用率
ライフサイクルモデル	1 イテレーションは 1 週間。最初の 1 ヶ月でごく基礎的な機能を実装し、α リリース。その後は、1~3 ヶ月の感覚で β リリースを 1 年 7 ヶ月続けたのち、正式リリース。その後は 3~6 ヶ月単位でメンテンスリリースを続けている。
チーム編成	プロダクトオーナ 1 名、オーナ支援 1 名、開発チーム 4 名。開発チームは当初はベテラン 4 名だったが、開始 4 ヶ月目以降は若手と順次入れ替えた。
プロジェクト期間	2 年(最初の製品正式リリースまで)
プロジェクト初期における要件の確定度合い	概要レベルで数ページのドキュメントが存在した程度
契約形態	四半期単位での準委任契約

■ 特徴

- 初期チームはベテラン 4 名で構成した。3 ヶ月目以降、徐々に若手とベテランとを入れ替えた。手法については完全に OJT で対応した。製品正式版リリース時点では初期メンバは 1 人もいなかったが文化は継承されていた。
- プロジェクト開始時点で合宿を実施し、プロダクトビジョンボックスを作成した。進行中は週次の計画ゲームと振り返りによって、プロジェクトの進むべき方向を調整した。
- ストーリカードとイテレーションのベロシティを元に、リリース計画とイテレーション計画を実施した。ベロシティのトラッキングにはバーンダウンチャートを用いた。
- ストーリカードを元にタスクカードを作成し、日々の進捗をスタンドアップミーティングとバーンダウンチャートによって追跡した。

(j) 共通認証システム開発事例（利害関係者関与：タイプ 1） 【I 社】

■ 事例概要

公共団体向けの共通認証システム等の開発事例である。システムの性質上、セキュリティが重視される。また、アーキテクチャとして SOA 及び OSS の採用が重視されたものである。2 ヶ月程度のイテレーションを 8 回、1 年半の期間である。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	セキュリティ SOA／OSS 重視
ライフサイクルモデル	イテレーション回数：8 回 イテレーション期間：2 ヶ月程度 イテレーション終了・開始時に、イテレーション報告、

	成果物のリリース、次イテレーションの計画を実施
チーム編成	5 人チーム リーダ：1 名 サブリーダ：1 名 開発者：3 名（若手 2 名、入社 2 年目 1 名）
プロジェクト期間	16 ヶ月
プロジェクト初期における要件の確定度合い	大雑把な要件は決まっていた
契約形態	請負契約

■ 特徴

- 実装するスコープは決定していたため、実装範囲、イテレーション回数などをプロジェクト計画として設定。スコープの確定後は、イテレーション単位の実施装範囲を計画して遂行。2 週間単位の振り返りを行い改善。
- ポストイットをタスクカードとして ToDo ボードに張り出し。計画ゲームにてタスクを決定、所要時間見積り、実績の測定、バーンダウンチャートで進捗を見える化。
- タスクごとの理想見積りを行い、実績から求めた係数をかけて工数を算定。
- 要件定義時には、UML（ユースケース図、アクティビティ図など）を利用した要件、業務フローの整理。
- 設計時には、UML（アクティビティ図、クラス図、シーケンス図など）を利用した設計。
- ビジネスロジック層はテスト駆動型開発で実施。
- 継続的インテグレーションを実施、単体テストの自動化、カバレッジの計測。テストで問題があった場合には、開発者メーリングリストに通知される。全機能が完成したあとは、ウォーターフォール型と同様に結合テスト以降を実施。
- 開発者はプロジェクトルームに集結。
- リスク管理票を作成。
- 開発マネージャには、特に教育は行っていない（実践しながらのトレーニング）。
- プロセス間のコラボレーション用に Wiki 等を活用。

(k) プロジェクト管理システム開発事例（利害関係者関与：タイプ 1） 【D 社】

■ 事例概要

1 週間単位のイテレーションを実施した。結果的にプロジェクト生産性が約 1.5 倍に向上したため、余力分を品質と保守性の向上のための作業に回すことが可能になった。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	スピード、要求の変化への対応を優先した。
ライフサイクルモデル	要件定義（1 ヶ月・2 人月）と並行して基本設計（1

	ヵ月・1 人月）⇒【基礎開発（1 ヶ月・3 人月）】⇒【本格開発（2 ヶ月・8 人月）】⇒改善対応（半年・1 人月）【】内がイテレーション期間。 各イテレーションは 1 週間に設定し、週はじめ開発対象の最終イメージ確認（紙芝居とデモ画面）、週おわりに最終成果で反応を見て適宜週末に要求の変化へ追随。
チーム編成	要件定義、開発、テストの 3 チーム。要件定義は独立チームで成果を開発リーダへ共有、基本設計～イテレート開発はリーダが率先して進めて若手を使い、テストは客観視する別チーム。
プロジェクト期間	4 か月
プロジェクト初期における要件の確定度合い	システム化対象の業務が未整理で要件定義（要求開発）から実施。
契約形態	請負契約

■ 特徴

- 要件定義、開発、テストの 3 チームの編成で実施した。要件定義は独立チームで成果を開発リーダへ共有するようにした。基本設計～イテレーションはリーダが率先して進めて若手を使うようにした。テストは客観視する別チームとして組織した。
- システム化される業務像と、完成システム最終イメージの具体化をスピーディに顧客側プロジェクトマネージャと開発リーダの間で、効率的かつ効果的に進めた。この結果、想定外の手戻りがなく、結果的にプロジェクト生産性が約 1.5 倍に向上した。
- １週間イテレーションの作業予実をもとに見積精度を向上していき、イテレーション開発を進めたことで顧客へ生産性を明示して期待させる形で進めた。

(l) アプリケーションプラットフォーム開発事例（利害関係者関与：タイプ 1） 【D 社】

■ 事例概要

ドメインの知識と従来の開発手法を良く知る顧客企業の開発メンバと D 社のメンバが組むことで、相互に補完しあう形でのアプリケーションプラットフォームの開発を実施した。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	標準化の推進、品質の安定、生産性向上
ライフサイクルモデル	要件定義(1 ヶ月・7.5 人月)→ 基本設計(2 ヶ月・15 人月)→ 実装(詳細設計や結合テストを含む、6 ヶ月・45 人月)

	→ パイロットシステムの開発(3 ヶ月・15 人月) 基本設計以降は、2 週間単位でイテレーション。実装工程以降は、イテレーションごとに統合、結合テストを実施
チーム編成	全国の開発拠点から選抜されたプロパーのプロジェクトメンバが 3 名、D 社のコンサルタントが 5 名でチームを編成。
プロジェクト期間	1 年
プロジェクト初期における要件の確定度合い	業務要件は決まっていた。システム要件はプロジェクト初期段階で定義。
契約形態	準委任契約

■ 特徴

- イテレーションごとに継続的な統合を実施したため、モジュール間のインタフェースの問題を早期に発見することができた。
- イテレーションが終わるたびに、そのイテレーションでの成果を統合した。開発が完了しなかったものは、次イテレーションに持ち越した。
- 開発プロジェクトマネージャを含む開発者全員は、プロジェクトルームに常駐した。
- 顧客要求は開発プロジェクトマネージャが窓口となり、開発チームが要求変更に振り回されないようにコントロールした。
- 継続的な統合により、「動くもの」を早期からシステムオーナに見てもらい、ギャップを埋めるようにした。

(m) 教務 Web システム開発事例（利害関係者関与：タイプ 1） 【E 社】

■ 事例概要

大学の基幹業務の一部である「教務システム」の開発を行った。学校行事に沿った各データ登録とバッチ処理から成る。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	入学生登録・履修登録・成績登録・進級卒業判定など、短期間で大量データ入力・処理する必要がある ・処理パターンの単純化 ・適度な自動化 ・大量印刷の分散化 を基本戦略とした
ライフサイクルモデル	（不明）
チーム編成	社内　開発リーダ：1 名、開発者：2 名 パートナー　開発者：3 名

プロジェクト期間	14 ヶ月
プロジェクト初期における要件の確定度合い	非常に低い 既存システムが参考になる程度
契約形態	一括作成請負

■ 特徴

- ポストイットを用いて簡易かつスピーディに設計を行った。
- 定期的にソースコードを見直し、内部構造の改善を行った。
- 技術課題・プロセス課題を貼り出して見える化し、開発者全員で解決策を検討・実施した。
- イテレーションごとに、要件の変化と技術課題を考慮し、作業タスクを調整した。
- プロセス・作業タスク・体調や精神管理についての情報共有を確実に行うため、毎日の朝回、月ごとの振り返り、ニコニコカレンダーの活用を実施した。

(n) 教育機関向け統合業務パッケージ開発事例（利害関係者関与：タイプ 3） 【E 社】

■ 事例概要

以前リリースされた大学向け統合業務パッケージのバージョンアップ版を開発。時代のニーズに合わせて機能を随時追加していく必要があるパッケージについては、アジャイル適用の効果が高いと考え、アジャイル型手法を適用した。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	既存の枠組みにとらわれない顧客視点のパッケージ
ライフサイクルモデル	全開発期間をイテレーション（短い期間）に分ける。各イテレーション内で設計、実装、テストを行う。いくつかのイテレーションを合わせてリリースと呼ぶ。1 リリースの中の初回イテレーションでは、リリース計画および既存障害修正期間とし、リリースに含める要件を決定する。最終イテレーションでは、テスト・リリース期間とし、リリースモジュールの作成およびシステムテストを行う。
チーム編成	1 つの顧客プロキシチームおよび複数の開発チームから構成される。開発チームは、各チーム 3～7 名、最大 4 チームで構成。
プロジェクト期間	41 ヶ月
プロジェクト初期における要件の確定度合い	低い（要件はイテレーション毎に追加が可能）
契約形態	社内パッケージ製品の開発であり、顧客との直接契約はない

■ 特徴

- プロジェクト方針は、「楽しく仕事をする」、「とにかくやってみる」であった
- 現在のタスクのほかにも、少しでも気になることがあれば、すぐに紙に記述して部屋の壁に張り付けるようにした。
- バーンダウンチャートを使用したカイゼンサイクルを実施した。見える化を行うだけではなく、全員で異常に気づき、対策を考え、実施するというサイクルを回した。
- ステークホルダとの調整は最小限に抑え、少人数の体制で意志決定していく方針を取った。
- 期間と投資上限をあらかじめ限定した。ビジネスの見立てがついた時点で、このプロセスの適用判断を行い、ビジネス検討期間も含めて期間、投資上限を決定した。
- レビューのためのドキュメントを極力減らし、開発者が必要なものだけをドキュメント化した。

(o) 検索エンジン開発事例（利害関係者関与：タイプ 4） 【J 社】

■ 事例概要

ショッピングサイトで商品、店舗検索をするためのエンジン開発の事例である。①検索性能の向上、②検索結果表示の戦略性向上、③店舗登録情報の早期反映が開発での方針である。主なシステムオーナは、市場事業になるが、開発部内からの改善、提案もあるため、自分自身がオーナになることもある。プロジェクトによって多少メンバの入れ替わりはあるが、比較的メンバは固定している。発案からリリースまで、2～8 週間と短期間である。

(p) システム管理ミドルウェア開発事例（利害関係者関与：タイプ 2） 【K 社】

■ 事例概要

システム管理のためのミドルウェアの GUI 部分の開発に関する事例。別企業にて開発した現行バージョンのバージョンアップを実施。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	(1)使用性(ユーザエクスペリエンス) (2)要求の変化への対応 (3)スピード
ライフサイクルモデル	予備検討(3 ヶ月)→開発(1 ヶ月×3 回)→品向フェーズ(統合テスト対応)(1 ヶ月)
チーム編成	十数名を 1 チーム。チーム内にサブリーダクラスを 3 名配置。
プロジェクト期間	7 ヶ月
プロジェクト初期における要件の確定度合い	中程度。但し、GUI 部分のため後工程で仕様変更の発生する可能性が高い。

契約形態	請負契約

■ 特徴

- 開発期間全体およびイテレーションごとにスコープを設定した。原則として次イテレーションへの持ち越しは極力回避する方針とした。
- イテレーションごとにスプリントバックログを元にスケジュール管理を実施。また、発注元開発との統合スケジュールについては入念に意識合わせを実施してスケジュール調整を実施した。
- プロジェクト開始時に全イテレーションにおける開発項目および開発規模の大まかな見積りを実施し、各イテレーションの実行可能性をチェックした。以後、各イテレーションの開始時に見積りを精査した。
- 本開発対象の GUI を統合する部分は、従来型の開発スタイルで実施していたため、従来プロセスの品質プロセス、品質指標を適用することが求められた。最終的に、個々のイテレーションではなく、イテレーション全体を対象としてバグ検出密度等の品質指標を適用することとした。
- 進捗管理についても、チーム内での管理にはバーンダウンチャートを用いたが、従来型の開発チームへの報告を行う際は、バーンダウンチャートをサマライズし、ガントチャートと整合性が取れるものを提供するようにした。

(q) 株式取引のための Web アプリケーション開発事例（利害関係者関与：タイプ 4）【L 社】

■ 事例概要

セルと称する小集団枠組によって、株取引のための Web アプリケーションシステムの開発を行った。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	1. IT 投資の回収を早めるために早期にビジネスを開始したい。 2. 新企画のビジネスであるため機能性や使用性等についてシステム利用者の反応を見ながら構築したい 3. 情報の漏えい等がないように安全に運用したい 4. 情報システム部門の多忙な負担をできるだけ軽減したい 5. 数年間は利用し続けたい（システムライフサイクル） 6. 品質要求については、保守性を最優先し、使いやすさ、即応性能を適切にする。
ライフサイクルモデル	X 証券会社は、非機能要件は定義できるが、どのような機能を実装してよいか、ほとんど確定できていないところから L 社に発注を行っている。X 証券会社と

	しては、注文を走らせながら、収益の大きな機能を模索し、できるだけ早く収益性の高い機能から実装して市場に提供したい。まず、対象となるシステムのスコープや品質、それを拡張・改善するために必要なイテレーション回数と並行稼働セル数を予測して、契約した。個々の機能に対するストーリは、リクエストオーダとして、随時発行されている。リクエストオーダとは、目的システムのユーザからX証券会社が随時受ける様々な要望であり、その視点や粒度は一定ではない。初期契約が締結された後、対象となるシステムのスコープや品質に適応可能なアーキクチャを設計し、母体となるシステムを約1ヶ月かけて構築する。母体が完成した後、機能の実現・実装を対象とするイテレーションを開始した。 バックログにある（工数を見積もられた）リクエストオーダの中からイテレーション（2 週間以内に構築できるリクエストオーダ集合）内に入れるべきリクエストオーダを選択する。生産企画者は、リクエストオーダを、その内容に適したセルにそれぞれ割りつける。セルはリクエストオーダに対する実現・構築を行い、2週間ごとにイテレーションとして納品する。プロトタイプや半製品のようなものは一切納品しない。1 つのセルは同時に1つのリクエストオーダしか受け付けない。X 証券会社は、納品されたシステムを検証し、ビジネスの現場へ投入できるものを検収し、実装する。以上の手順を契約期間の満了まで反復する。
チーム編成	世間一般のチームという概念に換えて、セルと称する小集団枠組を採用した。セルは複数存在し、それぞれ同じまたは異なる適性をもち、個々のセルの活動は、セル仕様によってマネージされ、セル屋台と称するツールセットによって支援される。セル仕様には、そのセルが必要とするスキルが定義されているだけで、セルに専従の人を張りつけるのではなく、動的に必要な人を張り付けるようにしている。セル仕様には、前提条件、完結条件、そのなかで必要となる制約、ガイドなどが定義される。
プロジェクト期間	23ヶ月
プロジェクト初期における要件の確定度合い	非常に低い
契約形態	X証券会社（顧客）とL社（ベンダ）の間で請負契約を締結している。X 証券が提供するサービスのユー

	ザ（証券取引者）間の契約は別に存在する。X と L 間契約は、対象となるシステムのスコープや品質基準とイテレーション回数と各イテレーションで並行稼働させるセル数を基準にする「逐次反復委任契約（仮称）」を理想とするが、まだ試行中であり、あるべき契約形態には到達していない。しかし、前向きに進んでいる。

■ 特徴

- イテレーション単位とバックログを基にしてスケジュール管理を行う。イテレーションではタイムボックスが固定することを絶対として進捗を管理する。そのため残業はない。バーンダウンチャートを利用し、作業進捗の管理ではなく、バックログの残数に対応するためのチーム作業バッファを管理する。WBS を利用したガントチャートによる進捗管理では対応できない。
- ソフトウェア・セルを採用。セル屋台をオン・オフライン上に作成し、コミュニケーションを図る。15 分間の朝会とイテレーションごとに回顧（振返り）を実施することでもコミュニケーションを図る。
- 2 週間という短期間のサイクルを繰り返すことで、顧客とのリスクは共有しやすくなった。また、バーンダウンチャートやかんばんを利用することでリスクを可視化し、課題の解決につなげた。また、チームワークやペアワークを基本とすることで、個人に依存するリスクを軽減した。
- イテレーション毎に必要な設計、プログラミングを実施する。すべての作業は、自動化もしくはペアワークを基本とする。
- 開発フェーズはアーキテクチャを決定し、システム母体を構築する 1 ヶ月だけととらえ、契約期間中の多くはメンテナンスであると考える。そのため、メンテナンスについて改めて考えるようなことはしない。

(r) プラント監視制御用計算機システム開発事例（利害関係者関与：タイプ 1） 【M 社】

■ 事例概要

プラント監視制御用計算機システムの開発事例である。プラントを監視制御する標準的なソフトウェアプラットフォームを提供するものであり、過去に開発されたプラットフォームソフトウェアを異なるプラットフォームへ移植し、維持する。現在使用中の顧客システムのリプレイスの為、アプリケーションソフトウェア再利用による信頼性が重視される。移植のための開発に 1～2 年を要し、移植後は約 10～15 年は使用し続けられる予定。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	現在使用中の顧客システムのリプレイスの為、アプリケーションソフトウェア再利用による信頼性を重視。
ライフサイクルモデル	過去に開発されたプラットフォームソフトウェアを異

	なるプラットフォームへ移植し、維持する。移植のための開発に 1～2 年を要する。移植後は約 10～15 年は使用し続ける。
チーム編成	システム設計チームは顧客要求に従ってデータ部をカスタマイズし開発部門はプラットフォームを開発維持供給する。 プロジェクト管理者層にはベテランメンバと技術専門者を入れ、仕様や品質に漏れが無いように体制を工夫。
プロジェクト期間	2 年
プロジェクト初期における要件の確定度合い	システム要件の実現にあたっては試行錯誤の繰返しであったと推定される。
契約形態	（不明）

■ 特徴

- 規模は、200 万から 300 万ステップ。
- ビジネスの新規性については、現在は確立しているが開発初期は先駆けであったもの。
- 技術の新規性についても、開発初期は全く新しい考え方であったが現在は確立した技術を採用。
- 開発計画書により開発目的、項目、スケジュール、資金、体制を定義した。
- 中日程、小日程により開発項目全てのタイムスケジュールを策定し、週ピッチで進捗を管理した。
- 開発開始から終了までのコストを見積もって予算化し、消化計画を策定した。
- 全ての開発ソフトウェアに対し、単体テスト、組合わせテスト計画を策定しテストを実施した。
- テストに関しては、試験仕様書、試験方案を策定し、機能仕様、性能、信頼性など多岐に渡ったシステム試験を実施。
- 週報やウォークスルーでメンバ全員が情報を共有。
- リスク部分については先行的にプロトタイピングにより評価し解決。
- スキルマップにより評価し最適な JOB アサインを実施。
- 仕様は把握しているので標準仕様を策定し社内メンバで決定。
- 前プラットフォームのアプリケーション機能仕様書を流用、またプログラムをマイグレーション。
- 保守としては、開発したソフトウェアは倉庫へ保存し、変更管理とアップグレード化している。

(s) 生産管理システム開発事例（利害関係者関与：タイプ 1） 【N 社】

■ 事例概要

新工場にて使用する生産管理システムの構築に関する事例。システム要件の細部は、新工場の建設に併せて固まるものであるため、要件未定ながら開発を進めなければならない状況であった。

■　プロジェクト基本情報

項目	内容
優先した IT 戦略	スピード、要求や市場の変化への対応力 ・新工場の建設と並行してシステム開発を進め、かつ竣工と同時にシステムが稼働できなければならなかったため、要件が固めきれないうえに納期を厳守しなければならなかった。そのため、仕様変更を受け入れながら短期間で開発できる手法が必要であった。 ・全社的な統合システムを一気に導入するのはさすがに無理があるから、まずは営業部門が取ってくるインゴットの受注から出荷までの部分を一元管理することが可能なシステムを作る。 ・これまで遅れを取っていた IT 活用による業務改革の先駆けになる（旧態依然としたやり方を変えていくのだという意識を皆に持ってもらうことも狙い）。
ライフサイクルモデル	予備検討（3 ヶ月） ↓ 構想具体化・実現性検証（6 ヶ月） ↓ 【 基礎開発（3 ヶ月） ↓ 本格開発（6 ヶ月） 】 ↓ 総合テスト（2 ヵ月） ↓ 改善対応（5 ヵ月） 【】内がイテレーション期間
チーム編成	設計チーム：6 人 実装チーム：8 人 （その他管理者）
プロジェクト期間	2 年
プロジェクト初期における要件の確定度合い	生産の現場では、システム化対象のプロセスの整理さえなされていなかった
契約形態	準委任契約

■　特徴

- 実装時には利用部門のキーパーソンに開発プロジェクトの部屋に詰めてもらい、業務の細かい仕様で不明な点はその場で解決できるようにした
- 開発メンバはベテランと若手の半々であった。アジャイル開発に向かないメンバを途中で 2 人交代させた

- 9 回のイテレーションのうち、前半から中盤にかけては時間を多く要しそうな重要な機能を組み入れ、後半では比較的軽い機能を多く入れることで、開発メンバにかかる作業負担が軽くなっていくようなペース配分にした
- 新しい要件を追加した場合は、優先度の低い要件を削るようにすることで、全体の負荷調整を行った

(t) Web メディア開発事例（利害関係者関与：タイプ 4）　【O 社】

■ 事例概要

完全内製ではないユーザ企業が、自社システムのビジネス価値を高めるために、アジャイル開発の考え方を体系化し、自分たちの現場に適用するなど、積極的にアジャイル的な取組みを組織的に推進している事例である。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	自社システムのビジネス価値向上を目的として、WEB メディア開発のスピードアップ。そのための解決策として、アジャイル開発の考え方を現場に浸透。
ライフサイクルモデル	次の順番で進む。 ・①ビジネス検討、②要件定義（フェーズ 1、フェーズ 2（サイクル 1、2））、③設計・製造・動作確認（サイクル 1、2）、④ユーザ動作確認、⑤テスト ・サイクル 1 で正常系を開発し、サイクル 2 で異常系を開発。
チーム編成	アプリ開発チーム、プロジェクト推進、プロジェクト基盤、共通アプリ基盤
プロジェクト期間	3 ヶ月
プロジェクト初期における要件の確定度合い	要件定義（PH1）終了段階で、各画面の基本要件について概ね洗い出されている。ワイヤーフレーム（ペーパプロトタイピング）が終わっていないと設計を開始しないとの方針が背景にある。
契約形態	準委任契約

■ 特徴

- アジャイル的に開発を進める手法を体系化し、推進部隊が、各プロジェクトに数人入り、その考え方を普及している。また、手法を活用して成功したプロジェクト例を他の事業部にも広めたりなどの動機づけを実施。
- 企画部門（ユーザ部門）、システム担当（システム部門）、開発者の 3 者が密なコミュニケーションを取りながら、開発をコントロールすることを重視。最初、毎日要件確認会を実施していたが、企画担当者の負荷が高すぎ、現在は 1 週間に 1 回になっている。ただし、半日以上の時間の確保をコミット。
- 決定・コミュニケーションの遅れが、最も開発現場を遅らせる。ドキュメント

ベースのやり取りは重たいため、企画部門、システム担当、開発者が対面で要求、仕様を伝達・確認するように改善。
    - 要件確認会には、企画担当者が出席するとともに、確認内容に関係する開発者は全員出席。
    - 要件確認会では、80%ルールに基づいて判断。
    - 要件確認会では、随時意思決定。
- 約 1500FP の規模を 3 ヶ月で開発。
- 2 種類（タスクベースと時間ベース）のバーンダウンチャートで進捗状況を把握。特に、Web サイトは似ている機能が多いため記録としてデータを蓄積しており、例えば会員機能であれば 40 画面必要だといった見積りを行っている。
- 品質確保の観点からは、新しいメンバが対応する場合は外部レビューを実施、サンプル的にソースコードレビューの実施、セキュリティや性能（負荷、レスポンス等）のチェックは専門チームが担当している。
- 情報システム部門の社員の責任が急激に高まっている。
- 過去に 2 チームで開発をした時にチーム間のアジャイル的な連携がうまくいかなかった経験に基づき、アプリ開発チームが複数チーム存在する場合、各チームの関係が疎になるように設計
- 保守性向上のために、相互レビューの推奨

(u) アジャイル型開発の支援環境開発事例（利害関係者関与：タイプ 3） 【P 社】

■ 事例概要

アジャイル型開発の支援環境を、当該支援環境のβ版を用いて、開発を行った事例である。10 人程度の開発チームが複数存在し、各チームリーダもメンバであるプロジェクト全体の会議体が全体のラフなプランを策定し、それぞれのチームが担当した部分（コンポーネント等）を反復的に実現している。なお、開発チームは国内だけでなく、海外にも分散している。リリース間隔は、約 1 年である。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	開発環境のユーザに対して、オープンソースと同じようなスタイルで、何を実装するかを決める段階からコミュニティにアイディアを公開し、コミュニティの人からフィードバックを得ている。また、商品としての成熟度が上がる前からソースコードをダウンロードできるようにして機能拡張のリクエストを収集。コミュニティからのフィードバックが、より製品の価値を高めるという考え方に基づいている。
ライフサイクルモデル	イテレーション 2 回が 1 サイクル。前半、後半がそれぞれ 4 週間くらい。
チーム編成	API またはコンポーネントごとにチームを編成。PMC と呼ばれる全体を統括する会議体で設定。

プロジェクト期間	4 年間続いているプロジェクトであり、現在も進行中。
プロジェクト初期における要件の確定度合い	上記のとおりコミュニティからの要望に基づくので、ある程度の要件は固まる。テーマと呼ばれる大枠での方向性（10 個）は変更しない。ただし、テーマ内の中小程度のものは変化する。
契約形態	社内開発

■ 特徴

- 開発に関わるメンバが 100 人を超え、10 近くのチームが海外にも分散して開発した事例。
- 開発対象が、アジャイル型開発のための開発環境であり、プロジェクトで、自分自身のβ版を適用し、実践したもの。
- 「テーマ」「フィーチャ」「ストーリ」でスコープを設定。ストーリは月間の集計が見えるとなどの動かすイメージ。フィーチャは、それが営業支援なのかなどのカテゴリ。1 年後のリリースのテーマは、例えばスケーラビリティを上げるためにはどうすればいいかを考えると、管理系でこういうオペレーションができればいい等のイメージ。
- マイルストーンファースト（最初にどれくらいの位置にマイルストーンを置くかを決めてからプロジェクト開始）とともに、アダプティブ・プランニング（計画駆動の目標も状況に応じて変えていく）を実践し、比較的計画に基づいて開発を進めつつ、状況に応じて計画を適応させることを行っている。
- 計画文書（作業項目を作る）があり、各メンバの作業は明確になっている。全体を統括する会議体はハイレベルなビューでチームごとの進捗状況が確認できる。ただし、この仕組みの大きな目的は上からの管理ではなく、チーム間で状況を把握するためのものとして機能させている。
- 複数チームで開発するので、コンポーネント（伝統的なアーキテクチャ上のコンポーネントではなく、フィーチャという単位できられたコンポーネント群）を疎結合に設計することが重要。アーキテクトが設計。
- 分散開発なので、電話会議・チャットを利用。特に海外とはチャットが言語の壁を下げるので有効。
- スクラムマスターはファシリテータ能力・チームの雰囲気作りが重要。

(v) 共通 EDI 開発事例（利害関係者関与：タイプ 5） 【Q 社】

■ 事例概要

業界内で共通的に活用可能な EDI 基盤を開発した事例である。要件提供者が多数存在するため、システム構成をモジュール化し、要件の固まったモジュールから順に開発。

■ プロジェクト基本情報

項目	内容
優先した IT 戦略	・業界標準システムの構築

	・共通画面・共通操作が可能なシステムの構築 ・企業間のシステム連携が可能なシステムの構築 ・基本機能は無償となるシステムの構築 ・中小企業用の標準システムの構築 ・短期開発を可能にするためのシステムのモジュール構造化 ・ハード・ソフトの省資源化（グリーン IT 化） （・最終的には SaaS 化）
ライフサイクルモデル	・初期プロトタイプの開発後、ユーザレビューと機能拡張を繰り返す ・EDI 基盤のコアの部分をはじめに固め、サブを順次追加 ・2 週間単位のイテレーション
チーム編成	15 名（リーダ：1 名、仕様担当：5 名、設計担当 3 名、開発担当：6 名）
プロジェクト期間	初期プロトタイプの開発に 3 ヶ月
プロジェクト初期における要件の確定度合い	多業種にわたるシステムであるため、開発当初は仕様が全く分からない、かつ固められない状態であった。
契約形態	準委任契約

■ 特徴

- 自社および協力会社のメンバにはアジャイル型開発の経験者がいなかったため、アジャイル経験が豊富なコンサルタントの支援を受けながら開発を推進。
- 開発開始当初は、新しい開発スタイルに上手く適用できずに様々な問題が発生したが、問題を解決しながら開発を進めていくうちに慣れ、最終的には当初見込みの半分程度の期間でプロトタイプが開発できた。
- 業務に絶対に必要な帳票項目だけを取り上げて機能を開発し、ユーザとすり合わせながら最終的な項目を固めた。
- ユーザからの変更要求は、それが実現されなければ業務が実施できないものに限るなどして、無制限にスコープが変更、拡大するのを制御した。
- ユーザに対してこまめに開発の状況を伝えることにより、なぜ要件が受け入れられないのかの理解も得るようにした。
- モジュールごとに開発担当者を決めることをせず、メンバ全員で全てのものを作ることとした。その結果、全員が何でも把握しているため、全ての仕様を文書化しなくとも運用で困ることはなかった。
- メンバ全員で全てのものを作る方針であったため、初心者が熟練者のソースに数多く触れることとなりスキルの向上につながった。

３．３．２ 非ウォーターフォール型開発手法に関するその他の取り組み

我が国においても、非ウォーターフォール型開発手法の普及促進や、適用領域の拡大等を目指した様々な取り組みが行われている。その代表的な例を列挙する。

表 5　非ウォーターフォール型開発に関する国内における取り組み

名称	活動	
Agile Japan	概要	Agile Japan は、アジャイルを軸に現場改革を進める、ビジネスマインドを持つ人の交流サイト。毎年 4 月に開催される同名のイベントの告知の他、ビジネスや改善の気づきを得るための情報提供（ニューインテリジェンス）、アジャイル開発に関する生の声の紹介（リレーコラム）、およびアジャイル開発の参考となる書籍の紹介（読書案内）などを行っている。 また、Agile Japan（イベント）は、世界最大のアジャイルコミュニティである Agile Alliance の協賛を得て行われている。
	URL	http://www.agilejapan.org/
アジャイルプロセス協議会	概要	2003 年 2 月にアジャイルプロセスの普及や情報交換などを目的として設立された。技術交流会や WG 形式による研究などを行っている。 現在、見積・契約 WG、アジャイルマインド勉強会、アジャイル・プロジェクト・マネジメント WG、西日本アジャイルプロセス研究会、知働化研究会等のワーキンググループによる活動が行われている。
	URL	http://www.agileprocess.jp/
日本 XP ユーザグループ	概要	2001 年 3 月に XP に関する情報交換や交流の場として設立。毎年 9 月には XP 祭りと題するイベントを開催。
	URL	https://r31.youroom.in/

３．４　非ウォーターフォール型開発の適用分野別マップ試作

３．４．１　適用分野別マップの試作

次に、非ウォーターフォール型の開発手法がどのような領域に対して適用されているのかを把握するため、３.３に示した各事例を直交する３つの軸によって形成される座標系にマップする。

ここでは、座標軸として次の３つの軸を設定する。

- **水平軸（不確実性）**

  エンタプライズから見た視点として、変化志向性が大きいか安定志向性が大きいかによって分類する。これは、事例を分類する際に採用したカテゴリ１（変化志向性大）、およびカテゴリ２（安定志向性大）に相当する分類である。ここで、不確実性を表す属性は、次のとおりに定義する。

属性	スコア				
	1	3	5	7	10
市場の不確実性	社会インフラを形成する基幹システムであるため、不確実な部分は逐次段階的であっても確実化してから実装する必要がある	目標市場に関する既成概念のマイナーチェンジで対応することが可能	目標市場の最初の推測の修正を繰り返しながら開発する必要がある	かなり不確実な要素が存在する市場	未知かつ未試験の新市場
技術の不確実性	既存アーキテクチャの拡張で解決する	構築方法が分かっている	構築方法がはっきりとは分からないが、既成技術、既成アーキテクチャを適用できる	既成技術、既成アーキテクチャでは解決できない部分が含まれる	新規技術、新規アーキテクチャ
プロジェクトの期間	1-3 ヶ月	6 ヶ月前後	12 ヶ月前後	18 ヶ月またはそれ以上	24 ヶ月またはそれ以上

依存関係／スコープの柔軟性	十分に定義されたスコープにおける責務遂行が義務付けられている	スコープが厳密に定義されていなくても、責務は義務付けられている	スコープと責務にはある程度の柔軟性が認められる	スコープには高い柔軟性があるので、義務遂行に関する契約上の配慮が必要	二者間で協議のうえ、最も適切な契約を選択する

不確実性の度合いは、属性ごとのスコア（x_i）を決定のうえ次の式に従って算出する。

$$\text{不確実性} = 2^{\Sigma \log_{10} x_i} \quad \cdots (1)$$

すなわち、不確実性の度合いが低い事例はつぎのようなものである。

(1) 定められた業務の実行および管理を、安定的に支援するもの
(2) ネットワークによって連携する分散組織間の協調を安定的に支援するもの

また、不確実性の度合いが高い事例は、つぎのようなものである。

(1) 「事業体自身のアジャイル化」を狙い、事業環境から得られる実時間情報と密接に連携して、企業戦略、業務執行計画およびガバナンスを支援するもの
(2) 事業体内の識見、知識、経験を相互に流通し、事業体のなかの生産性向上、品質向上、知識・スキル向上を主な目的にするもの

- **垂直軸（高速適応性）**

開発・実現方法から見た視点として、早期に価値を実現することを段階的に目指す傾向が強いか価値の実現速度以上に安全かつ安心な価値の実現を目指す傾向が強いかによって分類する。これは、継続的統合環境上での開発であるか従来型フレームワークに基づく開発であるかに相当する分類である。運用（または市場投入）によって早期価値実現を段階的に目指すものを上寄りに、早期というより安全かつ安心な価値実現により重点をおくものを下寄りに配置する。ここで、高速適応性を表す属性は、次のとおりに定義する。

属性	スコア		
	1	5	10
リリース密度（リリース回数／プロジェクト期間（月））	年 1 回	月 1 回	月 2 回以上
CI 環境の商用フレームワークからの独立度	商用フレームワークと密接に連携	商用ではないがある種のフレームワークと連携	独立した環境
ソフトウェア実装フレームワークの複雑度	クラウドや複雑・大規模な商用ソフトウェアフレームワークおよびプラットフォーム	標準的なマルチティレイヤ・ソフトウェアフレームワーク／マルティ・プラットフォーム	独自のシンプルな実装環境

高速適応性の度合いは、属性ごとのスコア（y_i）を決定のうえ次の式に従って算出する。

$$\text{高速適応性} = 2^{\sum \log_{10} y_i} \quad \cdots (2)$$

すなわち、高速適応性の度合いが高い事例には、つぎの特徴がある。

(1) 複数のリポジトリから構成され、CI を目的とした軽量な環境を用いて、開発の終わったイテレーション成果物から逐次システムに統合し、運用可能にするもの

(2) 商用ソフトウェアフレームワーク（ソフトウェア・サブシステムをレイヤに分けて配置し、アプリケーションを実行するために必要なサブシステム統合を行うために必要なセキュリティ・運用管理・通信管理機能を合わせ具えた枠組み、たとえば .NET または JavaEE）のなかのひとつのレイヤに対するイテレーションおよび CI を目的とするもの：たとえば、ウェブアプリケーションやワークフロー・アプリケーションだけを対象とするもの

また、高速適応性の度合いが低い事例には、つぎの特徴がある。

(1) 自社の社内開発環境上において、イテレーションおよび逐次 CI を繰り返し、システム全体 V&V および顧客（ユーザ）満足度が計画値を越えた段階で、一挙にリリースし、運用に入れるように計画されたもの

(2) 運用に入ったシステムに対して、ソフトウェアの全ライフサイクルに亘って、継続してイテレーションおよび CI を繰り返し、ソフトウェアの成

長、成熟、衰退、破棄までの構成管理を行うよう計画されたもの

- **第 3 軸（複雑性）**

情報システムを構築する技術、さらには情報システムを構築するプロジェクトに与えられる制約条件等、プロジェクトの複雑度が高いかどうかによって分類する。ここで、複雑性を表す属性は、次のとおりに定義する。

属性	スコア				
	1	3	5	7	10
チームの大きさ(人)	1	5	15	40	100
ミッションクリティカル	投機的	少数のユーザ	確立された市場	多数のユーザを持つミッションクリティカルシステム	重大な障害に耐えられるセーフティクリティカルシステム
チームの場所	同じ部屋	同じ建物	車で移動可能な範囲	時差 2 時間以下のタイムゾーン内	世界中に複数のサイト
チームの能力	熟練者による確立されたチーム	熟練者による新しいチーム	1 人の熟練者に対して限定的な経験しか持たないメンバ数を適正化	1 人の熟練者に対して限定的な経験しか持たないメンバ数をコントロール	1 人の熟練者に対して限定的な経験しか持たないメンバ数が大きくなる
ドメイン知識のギャップ	チームメンバは熟練者同様ドメインについて知っている	チームメンバはドメインについてかなり知っている	チームメンバはドメインについての支援を必要とする	チームメンバはドメインに触れたことがある	ドメインについて知らないチームメンバを入れざるを得ない

依存関係	依存関係はない	限定的かつ／もしくは十分に分離されている	中程度の依存関係	重大な依存関係	周辺システムに対する統合・完全化が必要

複雑性の度合いは、属性ごとのスコア（z_i）を決定のうえ次の式に従って算出する。

$$\text{複雑性} = 2^{\sum \log_{10} z_i} \quad \cdots (3)$$

- **マップ**

「不確実性」と「高速適応性」を、直交する 2 座標にそれぞれ割り付け、開発・保守手法ドメインをこの 2 座標によって形成される 4 つの象限に分割し、x 座標を式(1)によって算出される値、y 座標を式(2)によって算出される値として事例を各象限にマッピングする。これによって、まず、事例のマクロな特徴を把握できるようにした。

また、プロジェクトの複雑性を考慮するため、複雑性の高い事例と低い事例を別平面にマッピングすることとした。

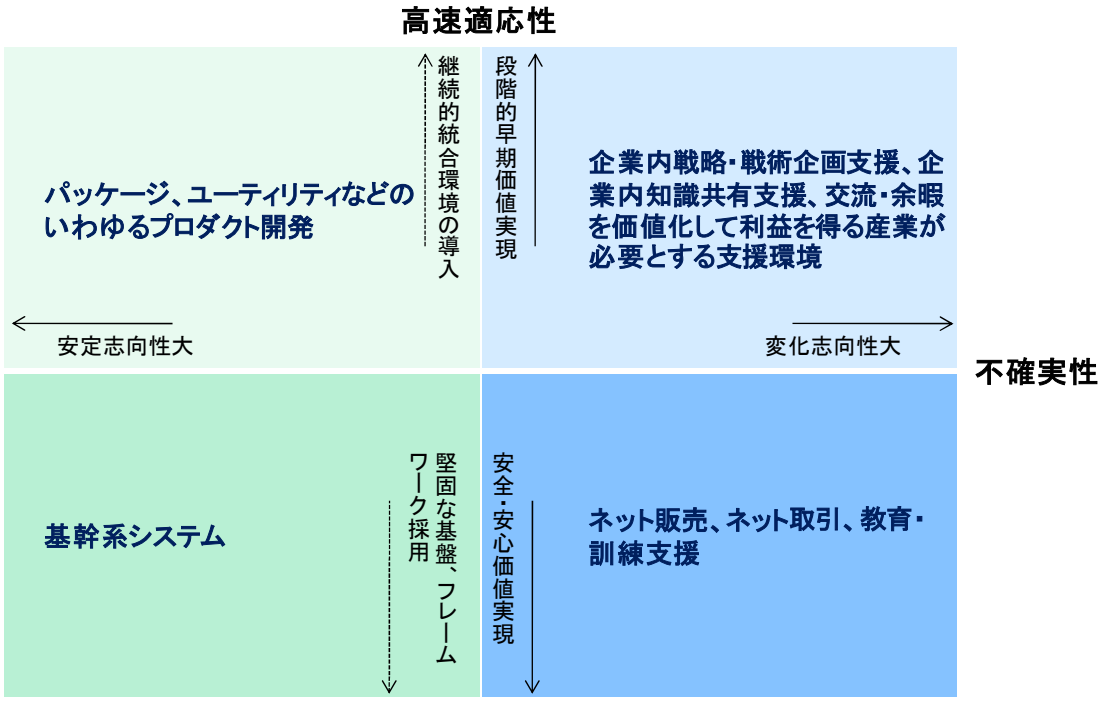


図 4　マップの定義

各象限に含まれる事例の特徴を、イテレーションの適用範囲の観点から分析する。

アジャイル型開発手法と計画駆動型開発手法の主な違いは、前者におけるイテレーションは、適合的（＝成り行きに合わせる）に行われるのに対して、後者におけるイテレーションは、計画的（予測的）に行われることにある（イテレーションという語に対する意味づけは、両手法でそれぞれ異なるが、ここでは単に反復される作業単位と考える）。イテレーションの適用範囲は、つぎの 3 種類に分類される。

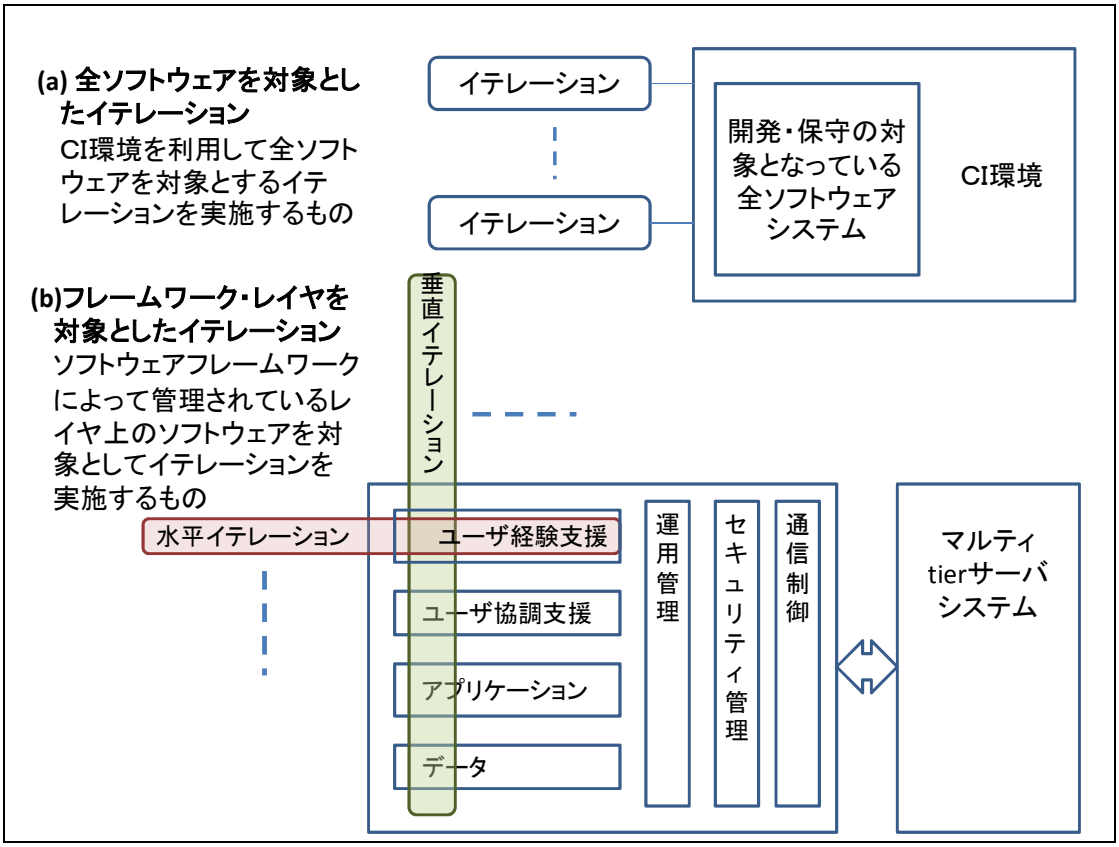


図 5　イテレーションの適用範囲

(a) 全ソフトウェアを対象としたイテレーション

CI 環境を利用しながら、各イテレーション成果物を、イテレーション完成時に、先行システムに統合し、運用に入れる。

第 1 象限にマッピングされた事例において、採用されている。

(b)フレームワーク・レイヤを対象としたイテレーション

(1) 特定のフレームワーク・レイヤに対して適用する水平イテレーション

ソフトウェアフレームワークは、ソフトウェア部品集合をレイヤに分けて管理し、実行時に、セキュリティ管理、運用管理、通信制御ユティリティによって部品を統合し、マルティ・ティア（multi-tier）サーバシステムと連携してユーザ・リクエストに応えるように構築されたソフトウェアの枠組みである。ここでいう水平イテレーションとは、特定のレイヤに管理されているソフトウェア部品集合だけを対象として実施するイテレーションのことである。第 2 象限にマッピングされている事例では、水平イテレーションの適用が見られた。

(2) 複数のフレームワーク・レイヤに対する垂直イテレーション

異なるレイヤに管理されている関連するソフトウェア部品集合の組に対して適用するイテレーションを、垂直イテレーションとよぶ。第 3 象限にマッピングされた事例では、垂直イテレーションの適用が見られる。大規模システムでは、機能ごと、たとえば、発券処理、改札処理、プリペードカード処理などごとにイテレーションを行うような場合が考えられる。このようなイテレーションでは、ひとつのイテレーションのなかに、複数のレイヤを含む必要がある。このようなイテレーションを、垂直イテレーションとよぶ。

以上のような性質を持つ座標平面に、本調査において収集した事例をマップした結果を示す。

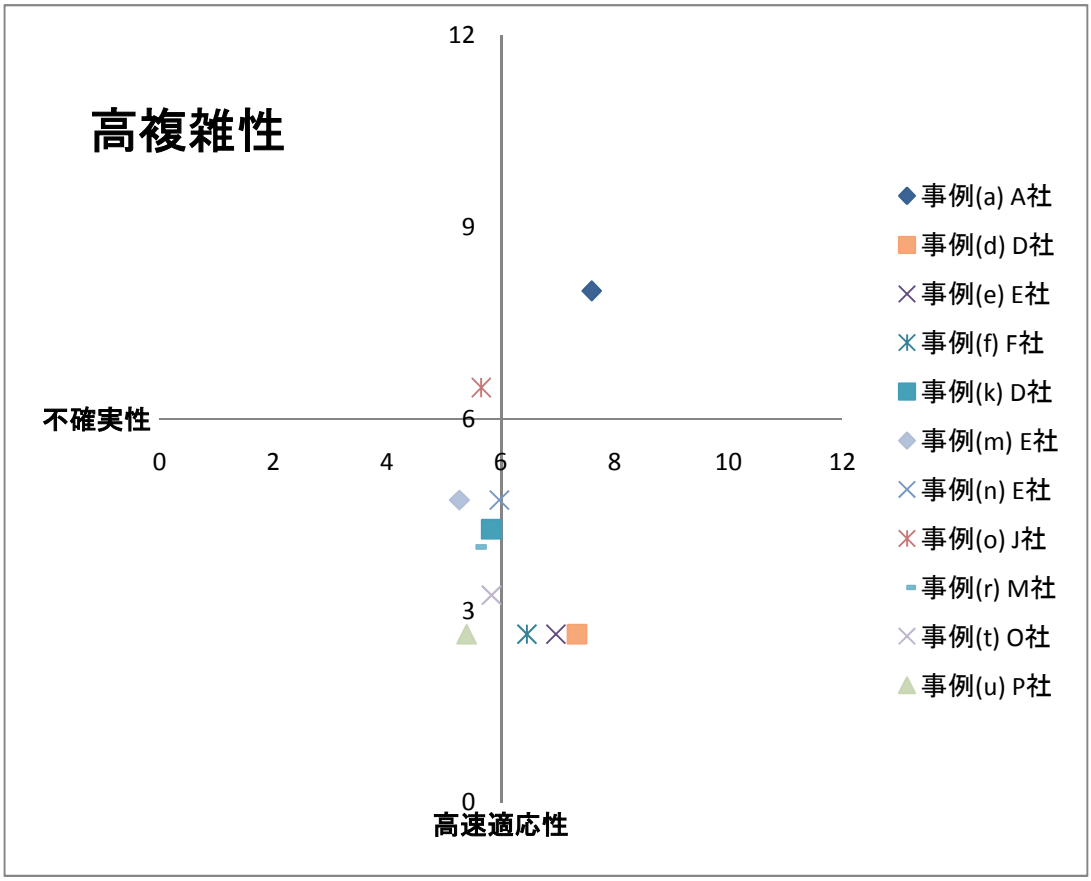


図 6　事例の分類結果（高複雑性事例の分類）

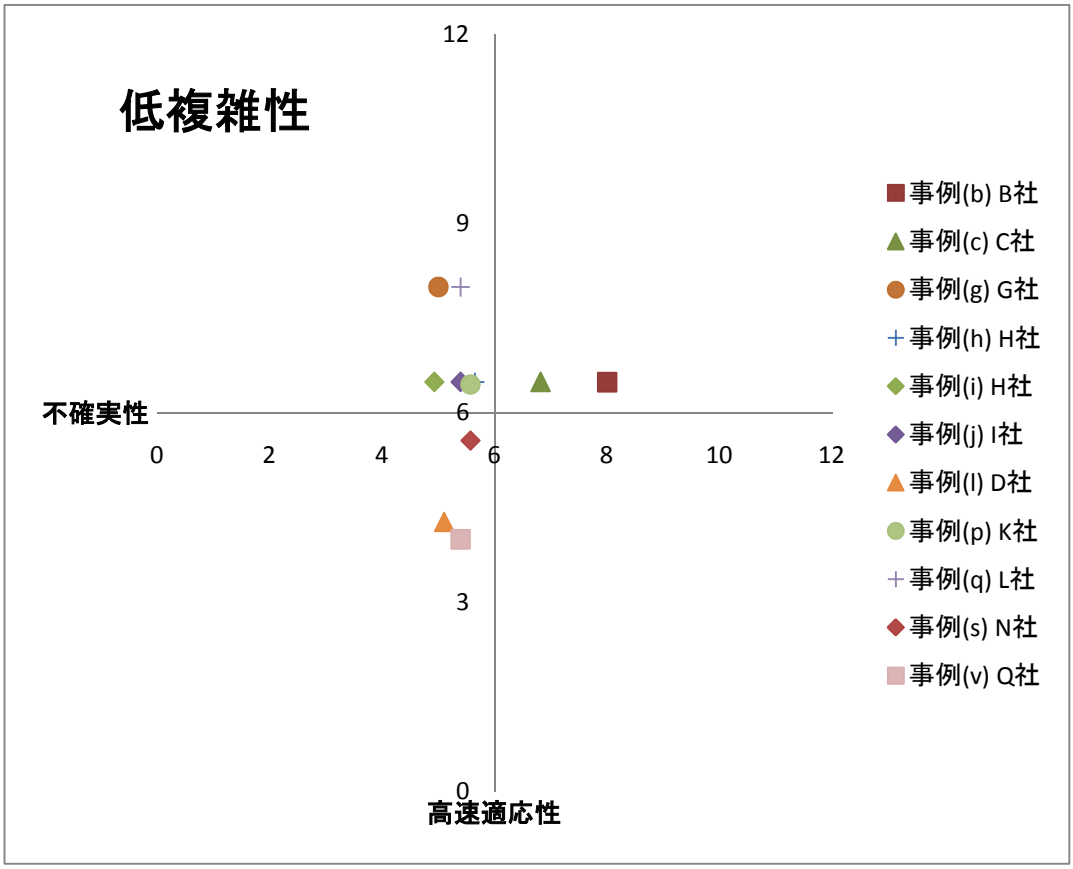


図 7　事例の分類結果（低複雑性事例の分類）

３．４．２ 非ウォーターフォール型開発の適用分野を整理するためのレーダチャート

(1) レーダチャートの作成目的

３.３では、非ウォーターフォール型開発手法が適用されたシステム開発の事例について説明した。しかし、非ウォーターフォール型開発手法がどのような領域に適用されているのかを把握するためには、各事例についてもっと俯瞰的に把握できるようにする必要があると考えられる。そのため、各事例の特徴をより明確に把握するのに有効な軸をいくつか設定し、レーダチャートを作成することとした。

(2) レーダチャートを構成する軸について

本調査では、各事例の特徴を表現するために、次に示す８つの軸を設定した。

表 6　レーダチャートの軸の定義

軸	軸定義	度数定義
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	次に示す特徴を備えた人材がプロジェクトチーム内に占める割合。 トレーニングを受ければ、手法の手順のうち手続き的な部分を遂行することができる（たとえば、単純なメソッドのコーディング、簡単なリファクタリング、コーディング標準や構成管理手順への準拠、テストの実行など）。 経験を積めばより上位レベルのスキルのいくつかをマスターすることができる。	0：不明 1：20%未満 2：20%～40%未満 3：40%～60%未満 4：60%～80%未満 5：80%以上
チームにおける「プロジェクトをマネジメントできる人」の割合	次に示す特徴を備えた人材がプロジェクトチーム内に占める割合。 先例のある新しい状況に適合するために手法をカスタマイズすることができる、もしくは、先例のない新しい状況に適合するために（ルールを破ってでも）手法を改訂することができる	0：不明 1：20%未満 2：20%～40%未満 3：40%～60%未満 4：60%～80%未満 5：80%以上
Time-to-market の時間的制約の厳しさ	仕様策定後、製品を市場に投入するまでの期間	0：不明 1：3 ヶ月以上 2：2 ヶ月～3 ヶ月未満 3：1 ヶ月～2 ヶ月未満 4：2 週間～1 ヶ月未満 5：2 週間未満
組織文化	カオスにおける繁栄を好むメンバの占める割合 カオスにおける繁栄とは 高い自由度を持つことによって、快適で権限を与えられていると感じる文化で繁栄する	0：不明 1：20%未満 2：20%～40%未満 3：40%～60%未満 4：60%～80%未満 5：80%以上

軸	軸定義	度数定義
システムの重要度の高さ	システムに不具合が生じた場合の影響の大きさ	0：不明 1.25：自社内の影響 2.50：一部の利用者の被害 3.75：社会的混乱が大きい 5：人命への影響、甚大な社会的損失が予想される
要求された稼働率の高さ	システムに要求された稼働率（運転時間内においてどれくらいの割合で稼動しているか）	0：不明 1：20%未満 2：20%～40%未満 3：40%～60%未満 4：60%～80%未満 5：80%以上
プロジェクト期間の短さ	プロジェクトの開始から終了までの期間	0：不明 1：1 年以上 2：6 ヶ月～1 年未満 3：3 ヶ月～6 ヶ月未満 4：1 ヶ月～3 ヶ月未満 5：1 ヶ月未満
プロジェクト初期における要件確定度合いの低さ	最初の設計に入る段階において要件が決まっていた度合い	0：不明 1.25：ほぼ固まっていた（非ウォーターフォール型開発でなくとも対応可能な程度） 2.5:ある程度の要件は固まっていた 3.75：基本的な要件しか決まっていなかった 5：ほとんど決まっていなかった（どんなビジネスを始めたい、システムで何を管理したいというイメージがある程度）
ビジネスの新規性の豊かさ	開発するシステムを活用して実施するビジネス自身の新規性の度合い	0：不明 1.25：ビジネス情報が世の中にもあり、自社内（自社内で行った検証等）にもある 2.5:ビジネスは世の中にあるが、自社内にはない。 3.75：ビジネス情報は自社内にはあるが、世の中には少数しかない。 5：ビジネス情報が自社内にも、世の中にもない。

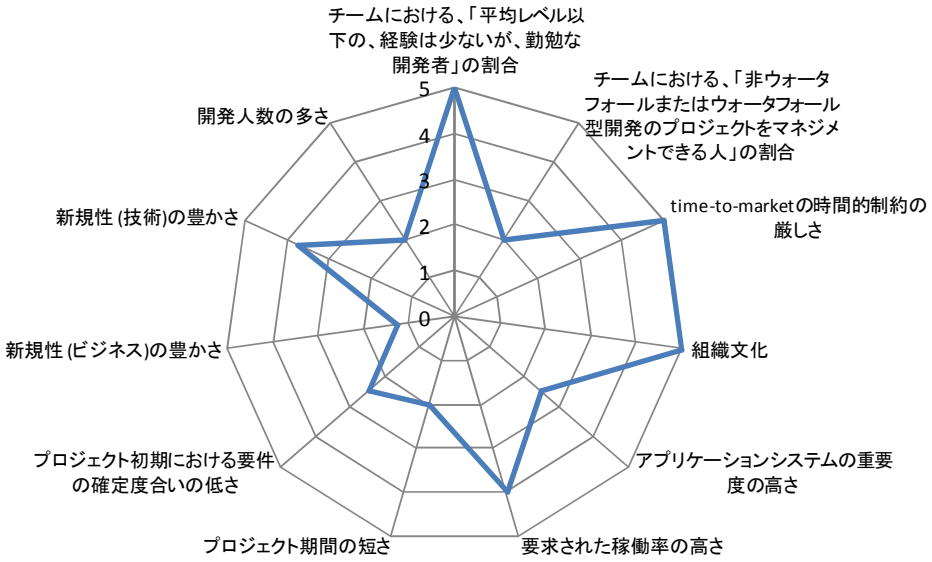
軸	軸定義	度数定義
採用技術の新規性の豊かさ	開発時に利用した技術の新規性や技術に対する経験の度合い	0：不明 1.25：技術情報が世の中にもあり、自社内（自社内で行った検証等）にもある 2.5:技術情報は世の中にあるが、自社内にはない。 3.75：技術情報は自社内にはあるが、世の中には少数しかない。 5：技術情報が自社内にも、世の中にもない。
開発人数の多さ	開発に携わった開発者の人数	0：不明 1：5 人未満 2：5 人～10 人未満 3：10 人～15 人未満 4：15 人～20 人未満 5：20 人以上

３．４．３ 事例に対するレーダチャートの適用結果

レーダチャートを、３.３にて説明した各事例に対して当てはめた結果を示す。

(1) カテゴリ 1 に属する事例

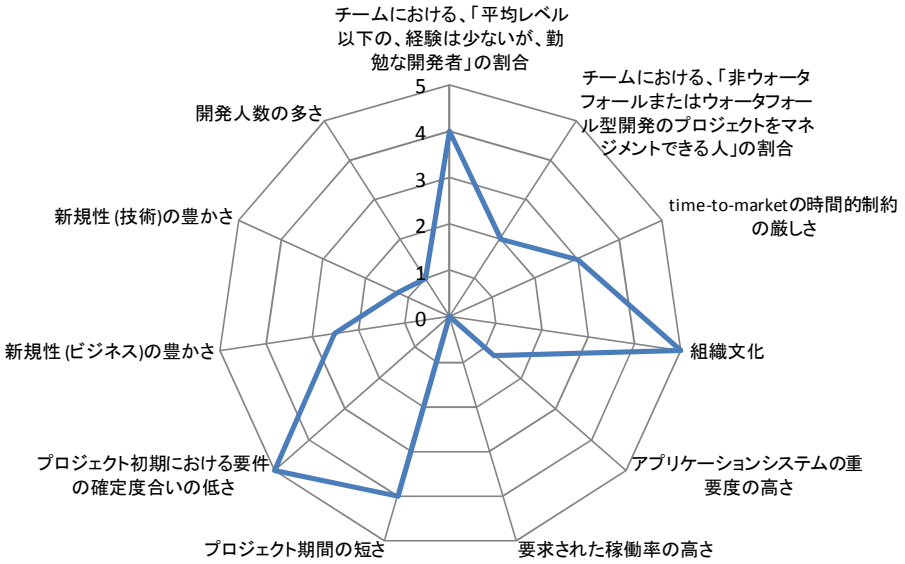
(a) 小売業における業務システム開発事例【A 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	80%	5
チームにおける「プロジェクトをマネジメントできる人」の割合	20%	2
Time-to-market の時間的制約の厳しさ	1 週間で開発し、1 週間で手直ししてリリースするサイクル	5
組織文化	カオスにおける繁栄を非常に好む風土	5
システムの重要度の高さ	システムの不具合が顧客のお客様へも及ぶ可能性がある	2.5
要求された稼働率の高さ	店舗営業時間中（6 時～22 時）は常時稼働	4
プロジェクト期間の短さ	6 ヶ月	2

プロジェクト初期における要件確定度合いの低さ	データベースのレイアウトやアルゴリズムなどについての仕様はなく、外部仕様レベルでの基本要件が固まっていたため、新たに要件を構築していくということはなかった	2.5
ビジネスの新規性の豊かさ	特に新規性はない	1.25
採用技術の新規性の豊かさ	独自の開発手法にこだわった開発であるが、その基礎となっている技術は非常に一般的	3.75
開発人数の多さ	6 人	2

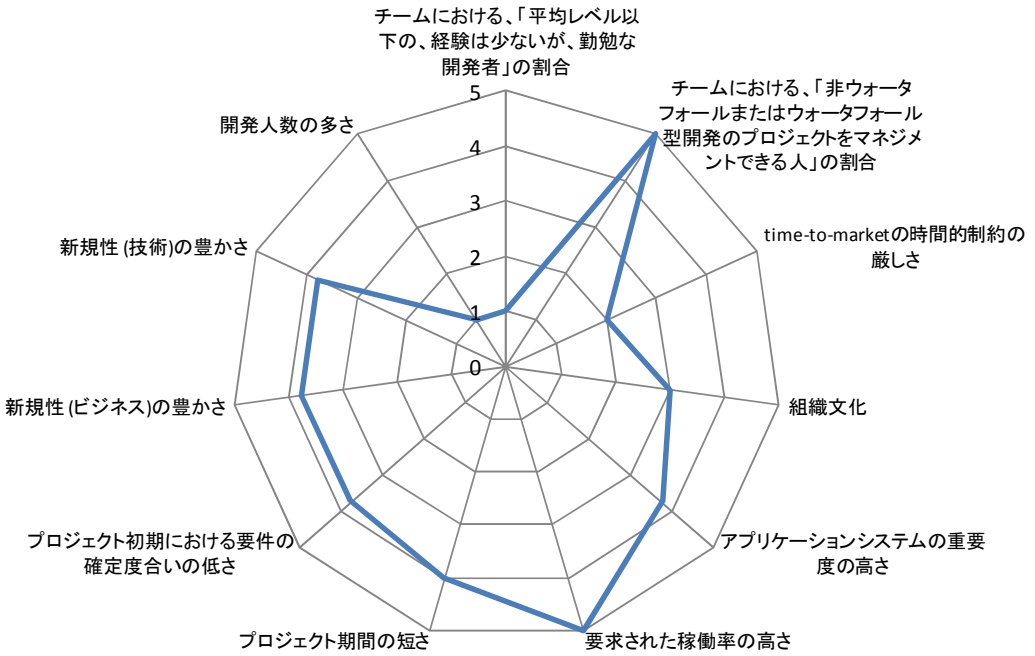
(b)　ソーシャルネットワーキングサービス（SNS）システム開発事例【B 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	66%	4
チームにおける「プロジェクトをマネジメントできる人」の割合	33%	2

Time-to-market の時間的制約の厳しさ	現在は 1 ヶ月に 1 度リリースする（イテレーションは 2 週間単位が基本）	3
組織文化	カオスにおける繁栄を非常に好む風土	5
システムの重要度の高さ	システムの不具合の影響は利用者の中にとどまる	1.25
要求された稼働率の高さ	不明	0
プロジェクト期間の短さ	社内開発時の初期リリースまでの期間は 2 ヶ月	4
プロジェクト初期における要件確定度合いの低さ	社内の技術情報を共有するという目的のみが定まった状態からスタート	5
ビジネスの新規性の豊かさ	SNS 自身に特に新規性はないが、SIer が開発したシステムをオープンソース化することは珍しい	2.5
採用技術の新規性の豊かさ	新規性は高くない	1.25
開発人数の多さ	3 人	1

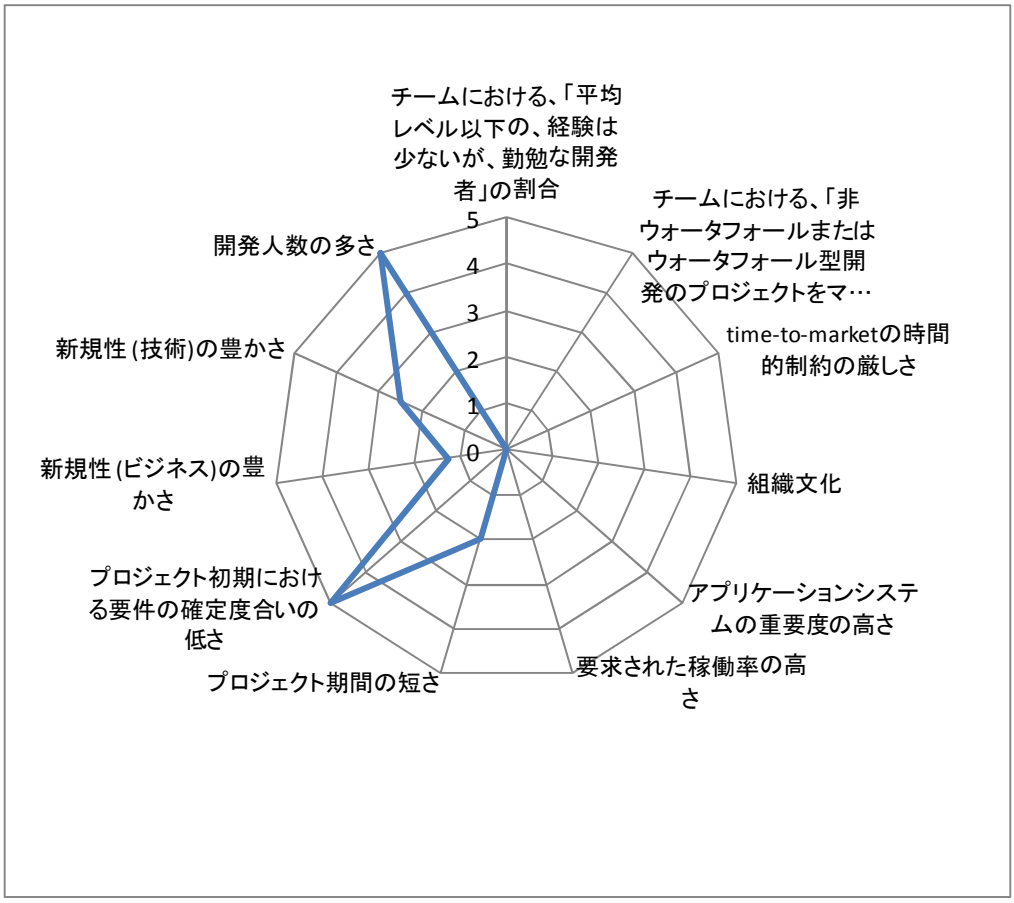
(c)　サプライチェーンマネジメントシステム開発事例【C 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	0%	1
チームにおける「プロジェクトをマネジメントできる人」の割合	100%	5
Time-to-market の時間的制約の厳しさ	2 ヶ月程度の事例が多い。	2
組織文化	－	3
システムの重要度の高さ	金融機関向けのシステムもある	3.75
要求された稼働率の高さ	およそ 100%。契約時点で SLA を締結し、月間で 30 分以上停止したら返金を行うようになっている。返金額は停止時間に依存する。	5

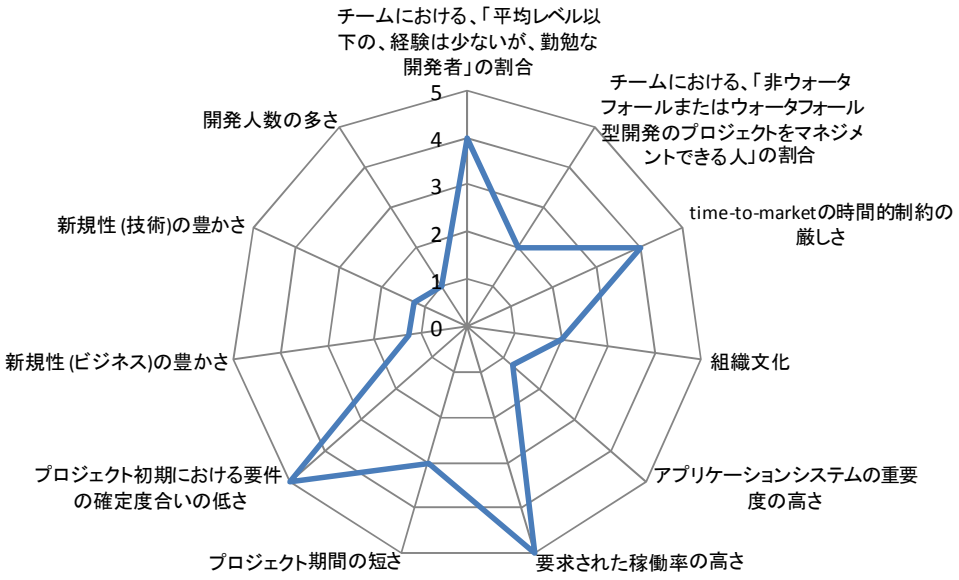
プロジェクト期間の短さ	2 か月程度の事例が多い。	4
プロジェクト初期における要件確定度合いの低さ	分析・設計から入ることが多い。要件を決めてもイテレーションの最中に変わることもある。我々からの提案もある。	3.75
ビジネスの新規性の豊かさ	既存 ASP 乃至はパッケージで満たすことのできないお客様のコアコンピタンスに係わるニーズを満たすためにスクラッチからアプリケーションを構成するケースが多い。	3.75
採用技術の新規性の豊かさ	独自技術を中核として開発を実施。開発者は、技術を十分に理解している。	3.75
開発人数の多さ	3-5 人と決まっている。コミュニケーションのオーバーヘッドが少なく、かつ、ある程度ロバストな人数である。	1

(d)　研修運営システム開発事例【D 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	（不明）	0
チームにおける「プロジェクトをマネジメントできる人」の割合	（不明）	0
Time-to-market の時間的制約の厳しさ	（不明）	0
組織文化	（不明）	0
システムの重要度の高さ	（不明）	0
要求された稼働率の高さ	（不明）	0
プロジェクト期間の短さ	10 ヶ月	2
プロジェクト初期における要件確定度合いの低さ	業務プロセス未整備	5
ビジネスの新規性の豊かさ	既存業務の改善	1.25
採用技術の新規性の豊かさ	既存技術の適用（一部フレームワークで新規性あり）	2.5
開発人数の多さ	ピーク時 20 名程度	5

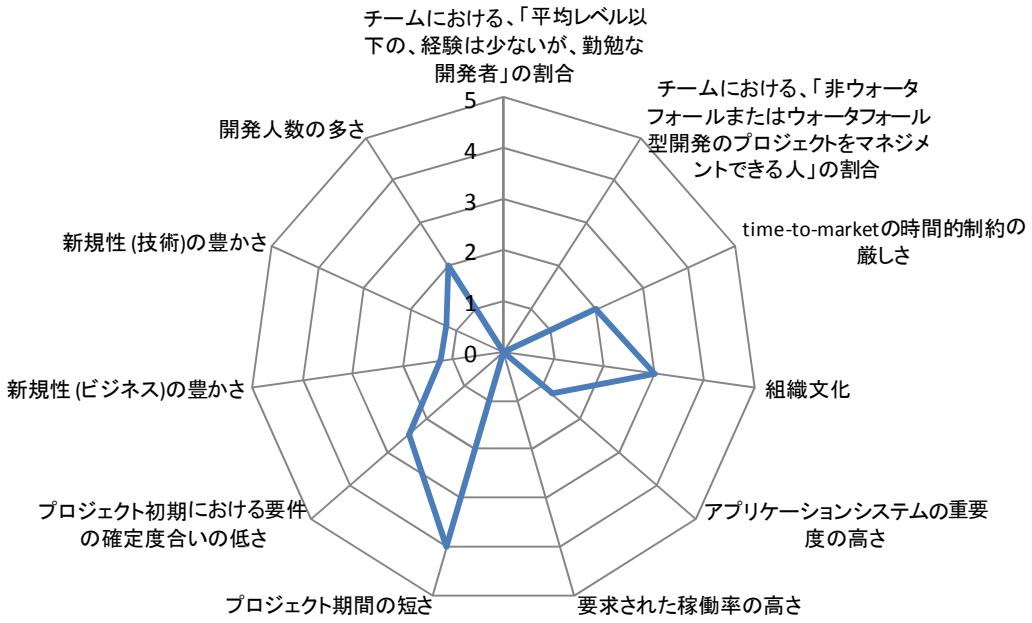
(e)　開発案件管理 Web アプリケーション開発事例【E 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	75%	4
チームにおける「プロジェクトをマネジメントできる人」の割合	25%	2
Time-to-market の時間的制約の厳しさ	特になし	4
組織文化	入社直後の新人がメンバであるが、先輩に対して従順かつ素直に受け入れる姿勢。 また、プロジェクトマネージャは面倒見がよく、指示・命令よりも動機付けとコーチング	2

	によってメンバを動かすリーダ。	
システムの重要度の高さ	自社内の一部の影響	1.25
要求された稼働率の高さ	ー	5
プロジェクト期間の短さ	3 か月	3
プロジェクト初期における要件確定度合いの低さ	非常に低い	5
ビジネスの新規性の豊かさ	特になし	1.25
採用技術の新規性の豊かさ	特になし	1.25
開発人数の多さ	3 人	1

(f)　製造業向けプロトタイプシステム開発事例【F 社】

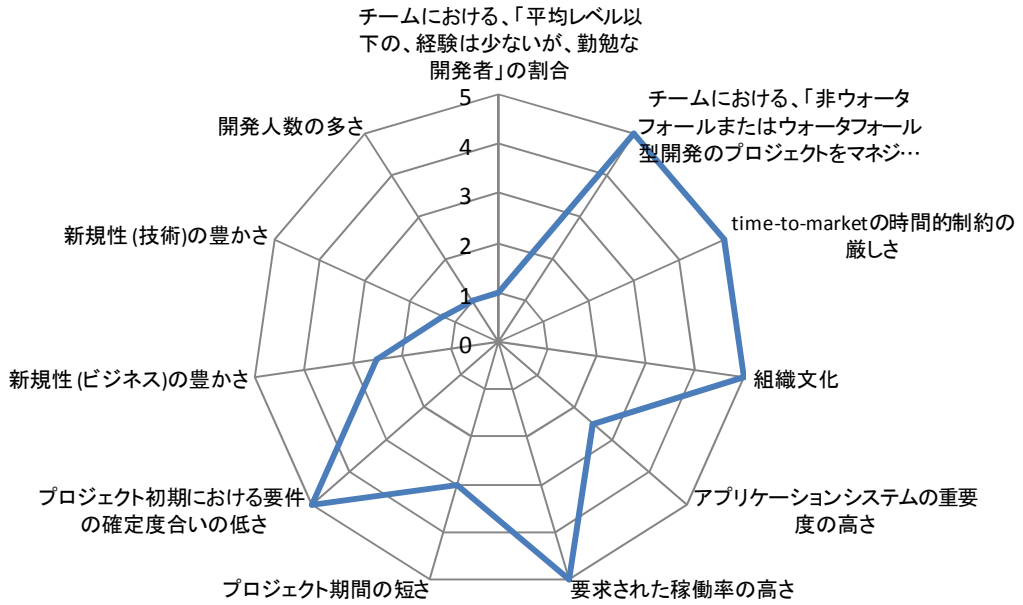


軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉	（不明）	0

な開発者」の割合		
チームにおける「プロジェクトをマネジメントできる人」の割合	（不明）	0
Time-to-market の時間的制約の厳しさ	2 ヶ月	2
組織文化	自律的に行動し、自ら率先する文化	3
システムの重要度の高さ	プロトタイプ（あまり重要ではない）	1.25
要求された稼働率の高さ	（不明）	0
プロジェクト期間の短さ	約 2 ヶ月	4
プロジェクト初期における要件確定度合いの低さ	明確になっていた要求は全体の 50%程度。開発工程期間中に散発的に到着。途中、消滅したものも多い。	2.5
ビジネスの新規性の豊かさ	同一システムを前回も開発	1.25
採用技術の新規性の豊かさ	製造業向けプロトタイプシステム開発	1.25
開発人数の多さ	9 名	2

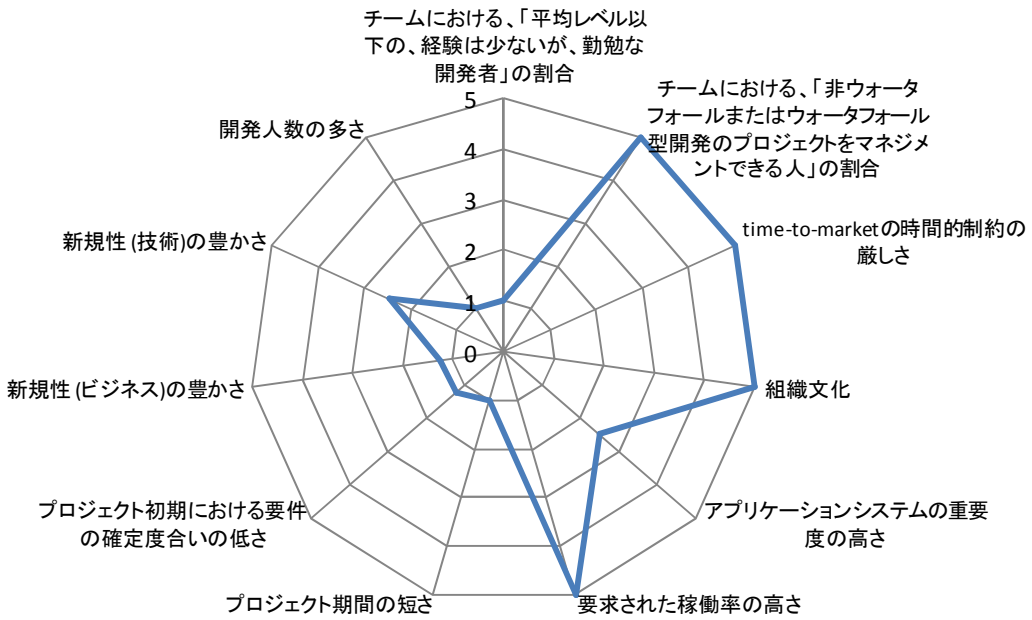
(2) カテゴリ 2 に属する事例

(g) 携帯ソーシャルゲーム開発事例【G 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	0%	1
チームにおける「プロジェクトをマネジメントできる人」の割合	100% ただし、多くの手法をマスターしたエンジニアではない。	5
Time-to-market の時間的制約の厳しさ	随時	5
組織文化	100%	5
システムの重要度の高さ	一部の利用者の被害	2.5
要求された稼働率の高さ	99.50%	5
プロジェクト期間の短さ	3 ヶ月～継続中	3
プロジェクト初期における要件確定度合いの低さ	0 ベース	5
ビジネスの新規性の豊かさ	社内では新規	2.5
採用技術の新規性の豊かさ	Flash 生成エンジンの一部以外既存	1.25
開発人数の多さ	2 人	1

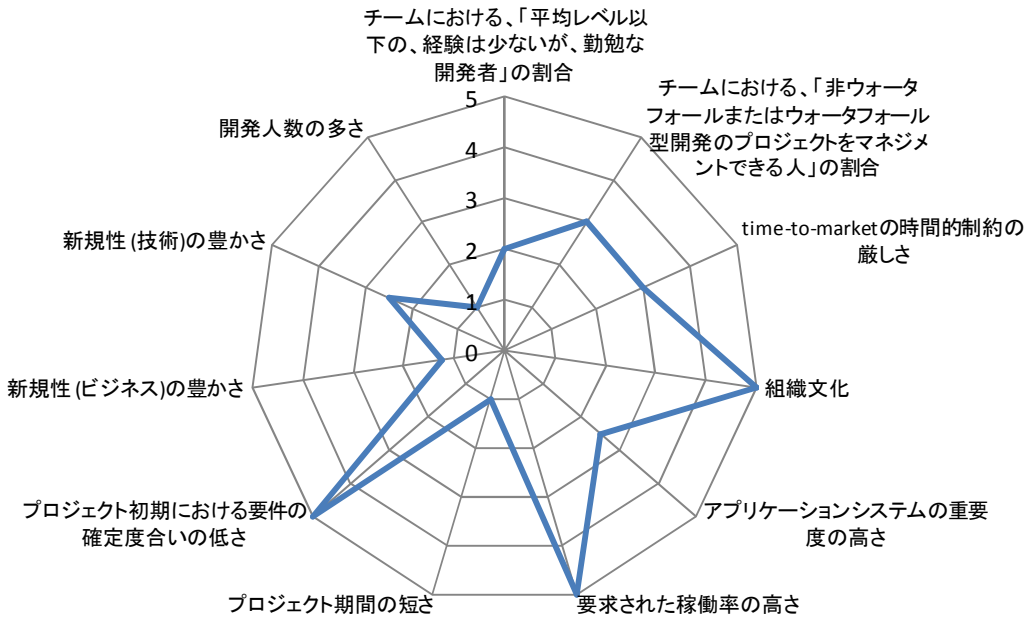
(h)　携帯端末向けブログシステム開発事例【H 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	0%	1
チームにおける「プロジェクトをマネジメントできる人」の割合	80%	5
Time-to-market の時間的制約の厳しさ	最初のリリースまでは 4 ヶ月 1 週間のイテレーション後テストを実施する。 ただし、イテレーションやテスト期間は 企画・機能の大きさによって変化。	5
組織文化	高い自由度がありカオスの状態を好む文化であったので、90%程度	5
システムの重要度の高さ	一部の利用者の被害	2.5
要求された稼働率の高さ	99%	5
プロジェクト期間の短さ	1 年 7 ヶ月	1
プロジェクト初期における要	既存のαサービスのリプレイス部分の基本的	1.25

件確定度合いの低さ	な機能の要件はほぼ固まっていたが、新機能に関しては、企画段階の物が多く要件としては全く固まっていなかった。	
ビジネスの新規性の豊かさ	Flash を全面に使ったゲーム性のあるブログサービス	1.25
採用技術の新規性の豊かさ	通信プロトコルは初採用の技術だった。	2.5
開発人数の多さ	3-5 人	1

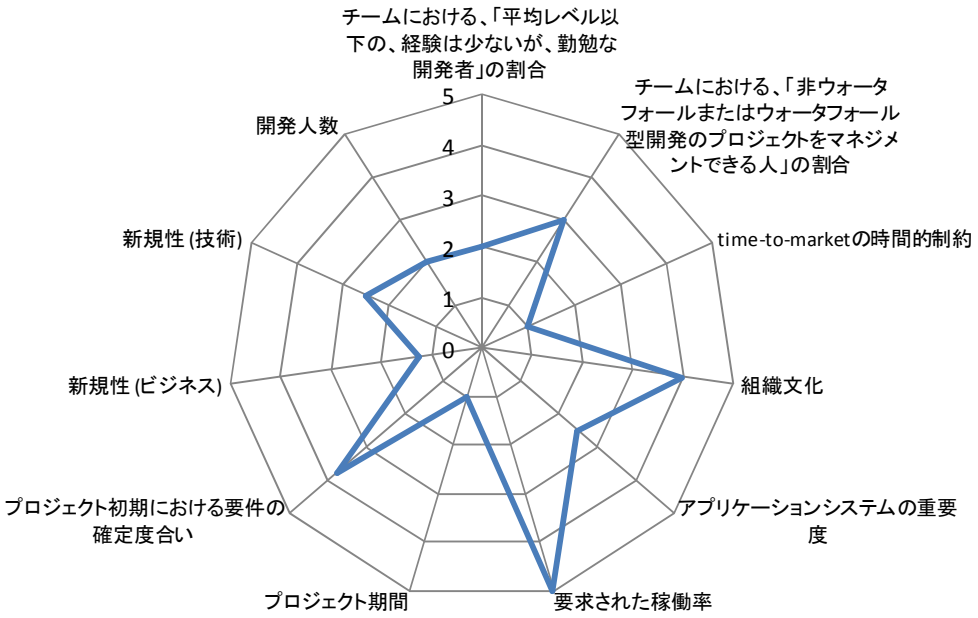
(i) パッケージソフトウェア開発事例【H 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	25%	2
チームにおける「プロジェクトをマネジメントできる人」の割合	50%	3
Time-to-market の時間的制約の厳しさ	1 ヶ月ー3 ヶ月に 1 回はリリースされた	3

組織文化	80%	5
システムの重要度の高さ	一部の利用者の被害	2.5
要求された稼働率の高さ	99%	5
プロジェクト期間の短さ	2 年(最初の製品正式リリースまで)	1
プロジェクト初期における要件確定度合いの低さ	概要レベルで数ページのドキュメントが存在した程度	5
ビジネスの新規性の豊かさ	別プラットフォーム(C++)の EJB 環境へのポーティングがベース	1.25
採用技術の新規性の豊かさ	EJB での本格利用は初めて	2.5
開発人数の多さ	4 人	1

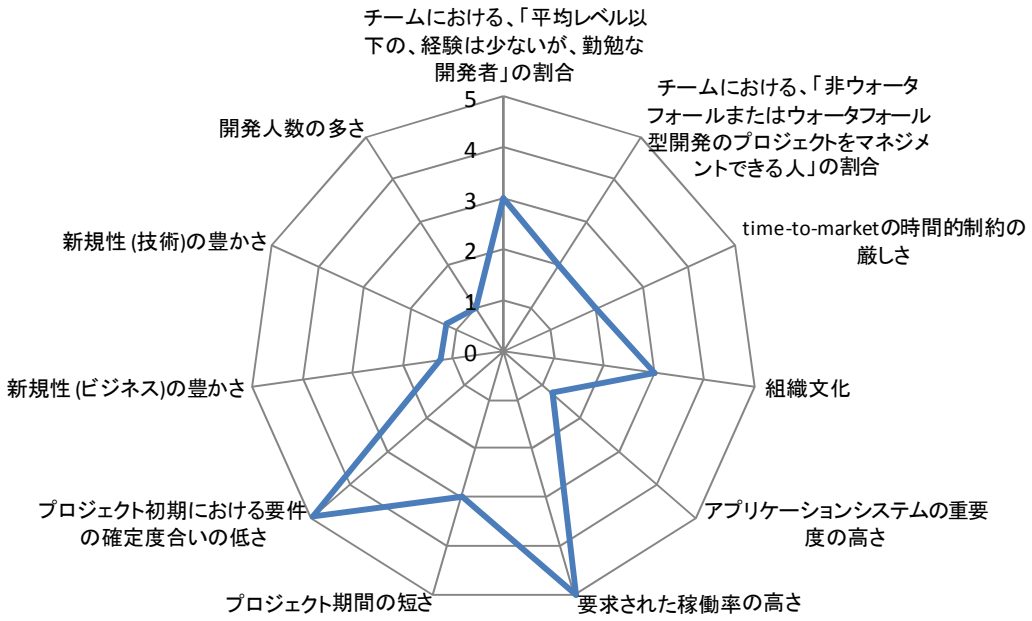
(j)　共通認証システム開発事例【I 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	20%	2

チームにおける「プロジェクトをマネジメントできる人」の割合	40%	3
Time-to-market の時間的制約の厳しさ	16 ヶ月（SI 案件）	1
組織文化	60%	4
システムの重要度の高さ	一部の利用者の被害	2.5
要求された稼働率の高さ	99%	5
プロジェクト期間の短さ	16 ヶ月	1
プロジェクト初期における要件確定度合いの低さ	大雑把な要件は決まっていた	3.75
ビジネスの新規性の豊かさ	複数システムの共通認証基盤となるもの	1.25
採用技術の新規性の豊かさ	SOA などの採用	2.5
開発人数の多さ	5 人	2

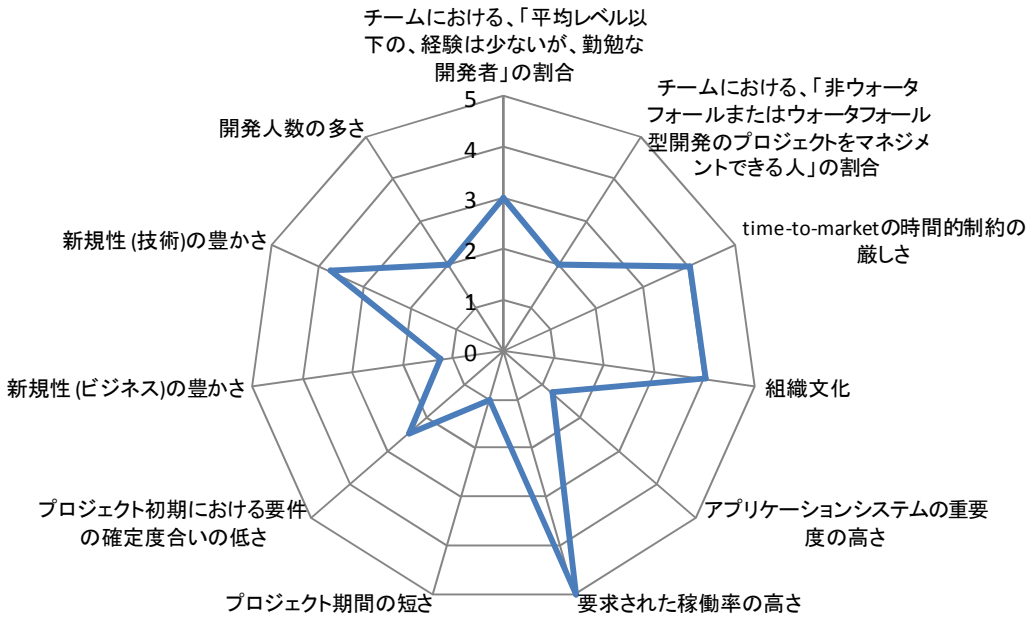
(k)　プロジェクト管理システム開発事例【D 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	50%	3
チームにおける「プロジェクトをマネジメントできる人」の割合	50%	2
Time-to-market の時間的制約の厳しさ	要件定義後 2 か月でリリース	2
組織文化	顧客は秩序における繁栄を好む文化であるが、ユーザ裁量が必要な対象システムであったため 40%程度。	3

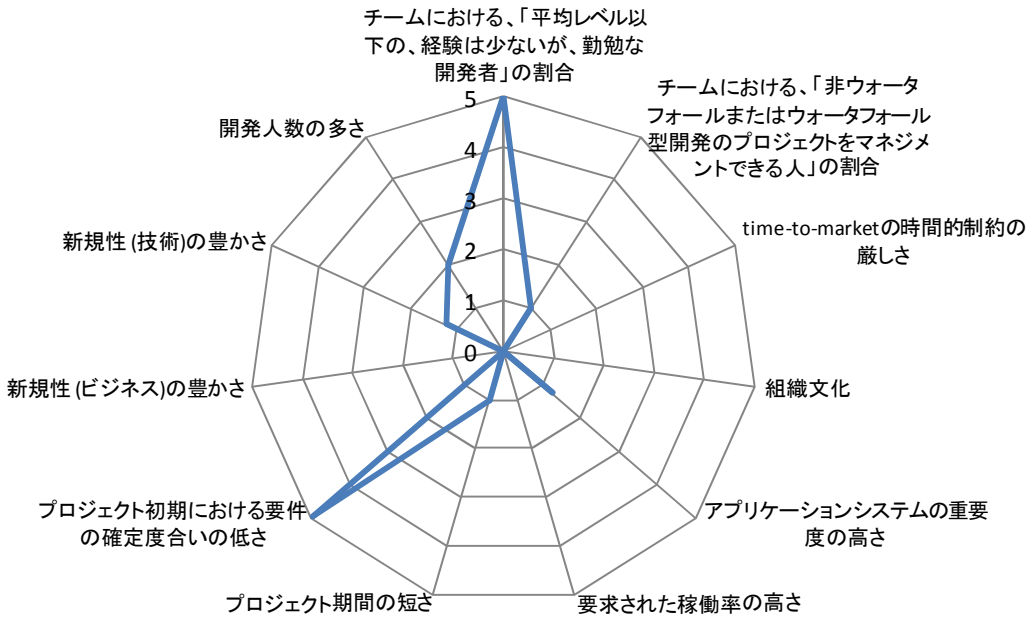
システムの重要度の高さ	被害は社内に留まる	1.25
要求された稼働率の高さ	90%	5
プロジェクト期間の短さ	4 ヶ月	3
プロジェクト初期における要件確定度合いの低さ	システム化対象の業務が未整理で要件定義（要求開発）から実施。	5
ビジネスの新規性の豊かさ	新規性なし（システム有無に関わらず必要な企業活動）	1.25
採用技術の新規性の豊かさ	基本 JSTL でシステム化するのはリーダが熟練していたが、他のメンバは初めて。	1.25
開発人数の多さ	プログラムは 3 人。主要部はリーダが開発して、他はテストなど。	1

(l)　アプリケーションプラットフォーム開発事例【D 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	50%	3
チームにおける「プロジェクトをマネジメントできる人」の割合	25%	2
Time-to-market の時間的制約の厳しさ	当初 8 ヶ月を想定。 実際には 12 ヶ月後に市場投入	4
組織文化	0.7	4
システムの重要度の高さ	被害は社内に留まる	1.25
要求された稼働率の高さ	99%	5
プロジェクト期間の短さ	1 年	1
プロジェクト初期における要件確定度合いの低さ	業務要件は決まっていた。システム要件はプロジェクト初期段階で定義	2.5
ビジネスの新規性の豊かさ	新規性はなし。従来ビジネスの開発プラットフォーム刷新	1.25
採用技術の新規性の豊かさ	Spring.NET は D 社としては他システムでの利用実績はあったが、一般的にはあまり使われていなかった。	3.75
開発人数の多さ	8 人	2

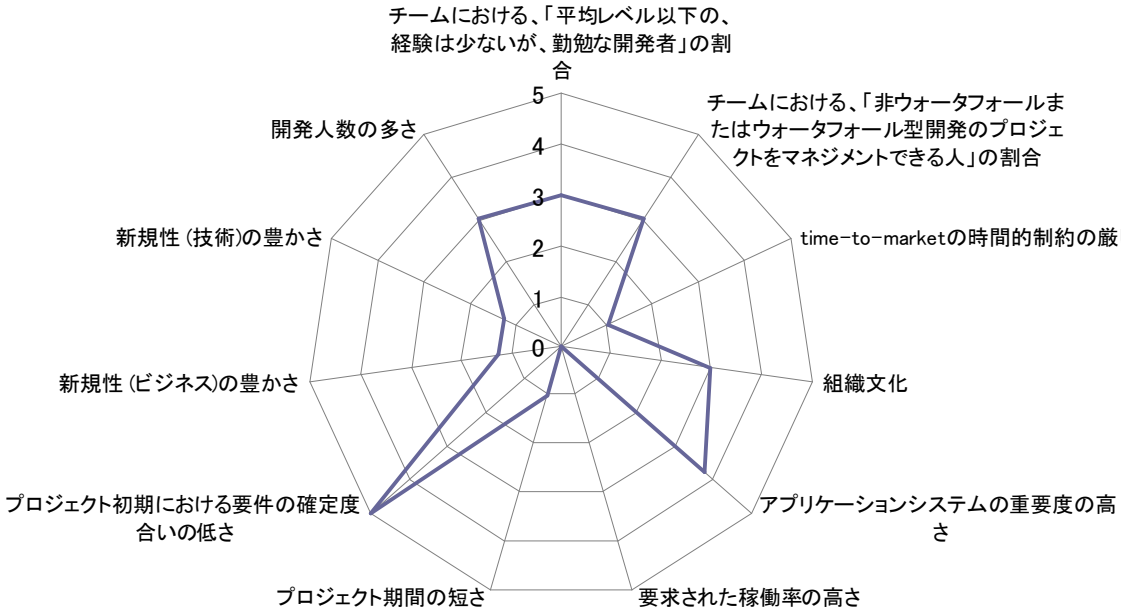
（m） 教務 Web システム開発事例【E 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	83%	5
チームにおける「プロジェクトをマネジメントできる人」の割合	17%	1
Time-to-market の時間的制約の厳しさ	不明	0
組織文化	不明	0
システムの重要度の高さ	重要 大量処理を伴うためシステム利用が必須	1.25

要求された稼働率の高さ	不明	0
プロジェクト期間の短さ	14 ヶ月	1
プロジェクト初期における要件確定度合いの低さ	非常に低い 既存システムが参考になる程度	5
ビジネスの新規性の豊かさ	不明	0
採用技術の新規性の豊かさ	オープンソースフレームワーク	1.25
開発人数の多さ	6 名	2

(n)　教育機関向け統合業務パッケージ開発事例【E 社】



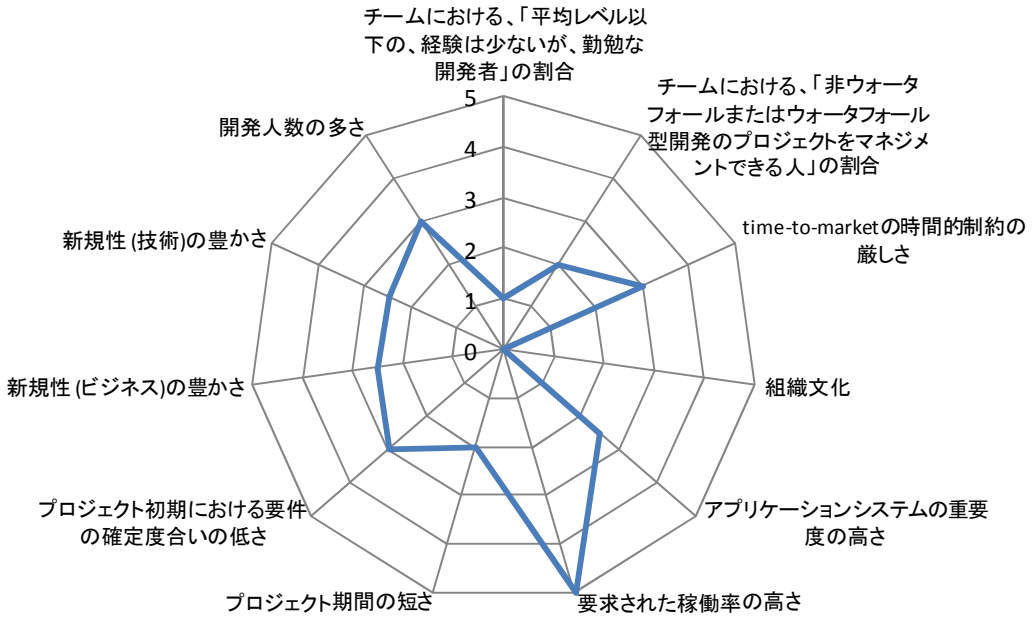
軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	50%	3
チームにおける「プロジェクトをマネジメントできる人」の割合	50%	3

Time-to-market の時間的制約の厳しさ	約半年～２年の開発期間	1
組織文化	・現場主義に基づく充実した職場環境 ・「とにかくやってみる」という風土 ・１人作業ではなく、複数での共同プレー	3
システムの重要度の高さ	レベル 2.5（：一部利用者の被害）～レベル 3.75（:社会的混乱が大きい）－基幹系は職員だけだが、情報系は職員に加え学生や保護者が含まれる	3.75
要求された稼働率の高さ	不明	0
プロジェクト期間の短さ	41 か月	1
プロジェクト初期における要件確定度合いの低さ	低い（要件はイテレーション毎に追加が可能）	5
ビジネスの新規性の豊かさ	特になし	1.25
採用技術の新規性の豊かさ	特になし	1.25
開発人数の多さ	当初 8 人～最大 24 人	2

(o)　検索エンジン開発事例【J 社】

※確認取り次第掲載予定。

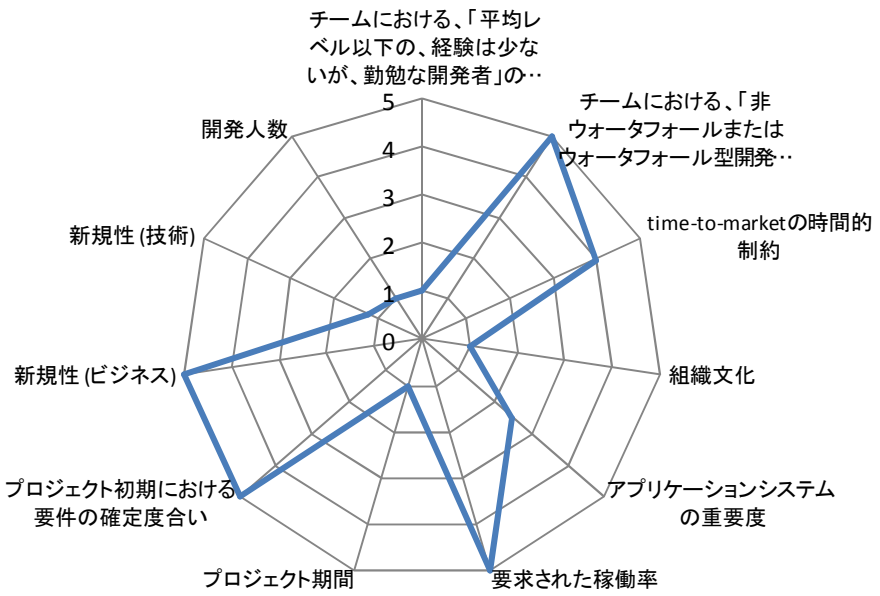
(p)　システム管理ミドルウェア開発事例【K 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	0%	1
チームにおける「プロジェクトをマネジメントできる人」の割合	20%	2
Time-to-market の時間的制約の厳しさ	・海外営業拠点での操作性評価に使用するタイミングに関する制約 ・K 社開発部分との統合テストに使用するタイミングに関する制約 ・製品β版、リリース版の提供時期必達に対する制約(海外営業拠点からの指示)	3
組織文化	不明	0
システムの重要度の高さ	SMB のユーザを想定し、SE レスでインスト	2.5

	ールから運用、問題解決までできるようにすることを目標としている。	
要求された稼働率の高さ	要求はされていない	5
プロジェクト期間の短さ	7 ヶ月	2
プロジェクト初期における要件確定度合いの低さ	中程度。但し、GUI 部分のため後工程で仕様変更の発生する可能性が高い。	3
ビジネスの新規性の豊かさ	新規ビジネス	2.5
採用技術の新規性の豊かさ	あり(Flex)	2.5
開発人数の多さ	十数名	3

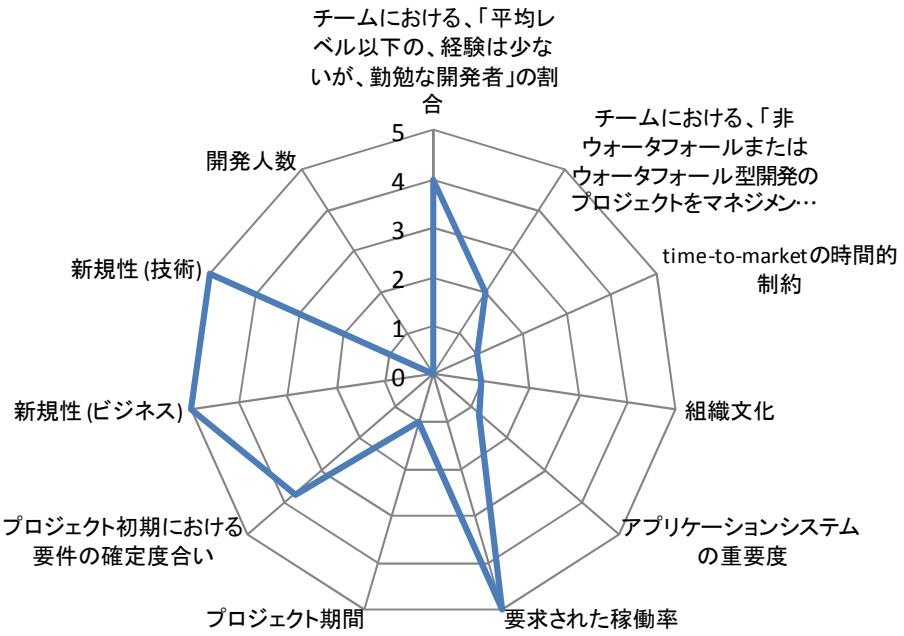
(q)　株式取引のための Web アプリケーション開発事例【L 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	0%	1
チームにおける「プロジェクトをマネジメントできる人」	100%	5

の割合		
Time-to-market の時間的制約の厳しさ	バックログに蓄積されたリクエストオーダ（新規または拡張・改善要求）の 2 週間分をまとめて取り上げ、開発、拡張、または改変構築を行い、その成果物プロダクトを 2 週間ごとに納品し、検収を受ける。納品されたプロダクトは、発注元である証券会社内部のプロモーションプロセスを経て実装され、運用に供される。	4
組織文化	「2 週間ごとに､蓄積されたリクエストオーダを処理して納品する」というディシプリンが組織的に浸透している。前に納入し、運用されているプロダクトの拡張・改善の比率がきわめて高いので、保守が容易になるように構築しなければならないという躾が浸透している。	1
システムの重要度の高さ	現在は、レベル 2.5（：一部利用者の被害）にとどまるが、将来は、レベル 3.75（:社会的混乱が大きい）――ネット証券という社会インフラのひとつとして利用される。	2.5
要求された稼働率の高さ	営業日の営業時間内では、100%の稼働率が求められる。	5
プロジェクト期間の短さ	23 か月	1
プロジェクト初期における要件確定度合いの低さ	非常に低い	5
ビジネスの新規性の豊かさ	新企画のビジネス	5
採用技術の新規性の豊かさ	特になし	1.25
開発人数の多さ	3～5 人	1

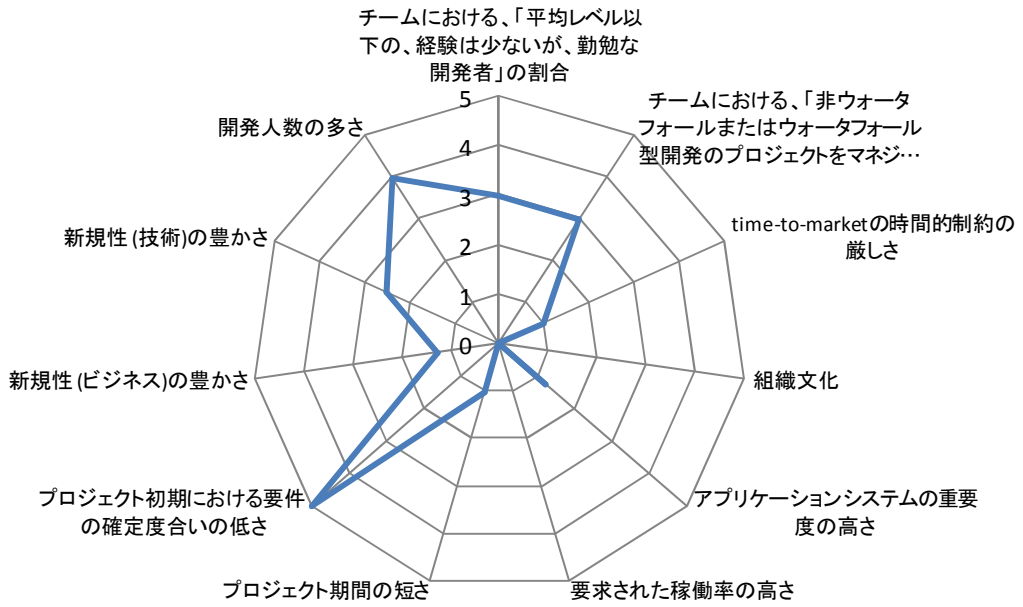
(r)　プラント監視制御用計算機システム開発事例【M 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	70%	4
チームにおける「プロジェクトをマネジメントできる人」の割合	20%	2
Time-to-market の時間的制約の厳しさ	12 ヵ月に 1 回リリース	1
組織文化	どちらかと言うと秩序による繁栄を好むため 10%	1
システムの重要度の高さ	被害は社内にとどまる	1.25
要求された稼働率の高さ	99%以上	5
プロジェクト期間の短さ	2 年	1
プロジェクト初期における要件確定度合いの低さ	システム要件の実現にあたっては試行錯誤の繰り返しであったと推定される	3.75

ビジネスの新規性の豊かさ	現在は確立しているが開発初期は先駆けであった	5
採用技術の新規性の豊かさ	開発初期は全く新しい考え方であったが現在は確立した技術である	5
開発人数の多さ	（不明）	0

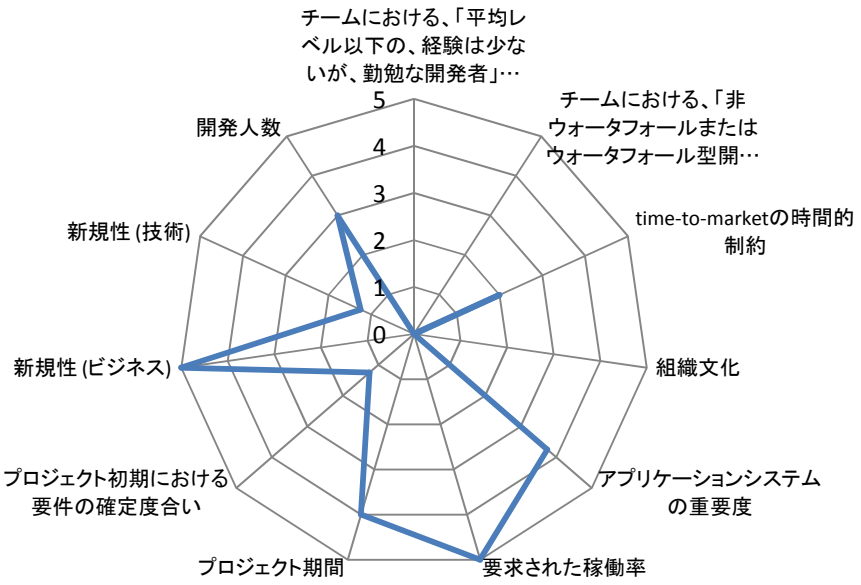
(s)　生産管理システム開発事例【N 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	不明（ただし、アジャイル型の開発経験者はいない）	3
チームにおける「プロジェクトをマネジメントできる人」の割合	不明（ただし、若手とベテランの割合は半々）	3
Time-to-market の時間的制約の厳しさ	3 ヵ月に 1 回リリース	1
組織文化	不明	0
システムの重要度の高さ	クライアント端末は 70 台あり、3 箇所の工場	1.25

	に設置されている。	
要求された稼働率の高さ	不明	0
プロジェクト期間の短さ	2 年	1
プロジェクト初期における要件確定度合いの低さ	生産の現場では、システム化対象のプロセスの整理さえなされていなかった	5
ビジネスの新規性の豊かさ	不明	1.25
採用技術の新規性の豊かさ	アジャイル型の開発は初めての経験であった。	2.5
開発人数の多さ	設計チーム：６人 実装チーム：８人 （その他管理者）	4

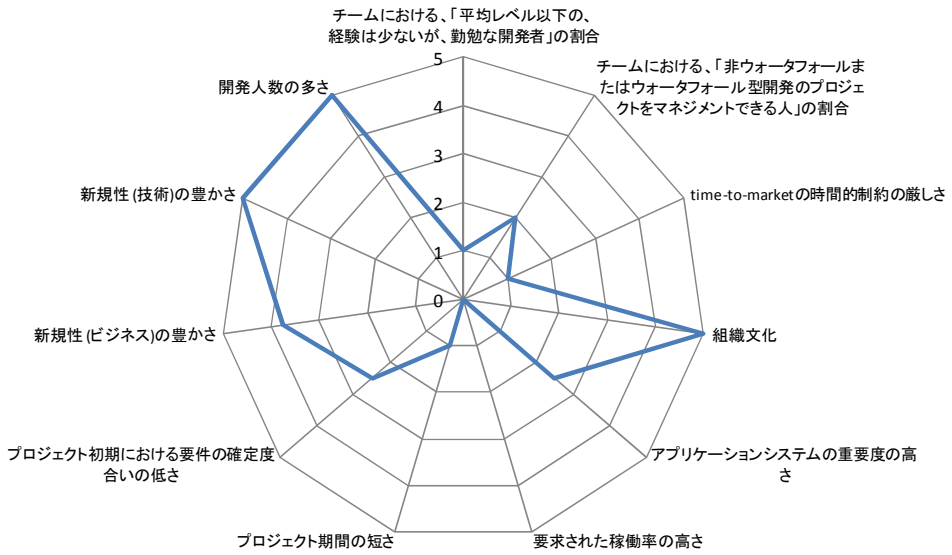
(t)　Web メディア開発事例【O 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	不明	0
チームにおける「プロジェク	不明	0

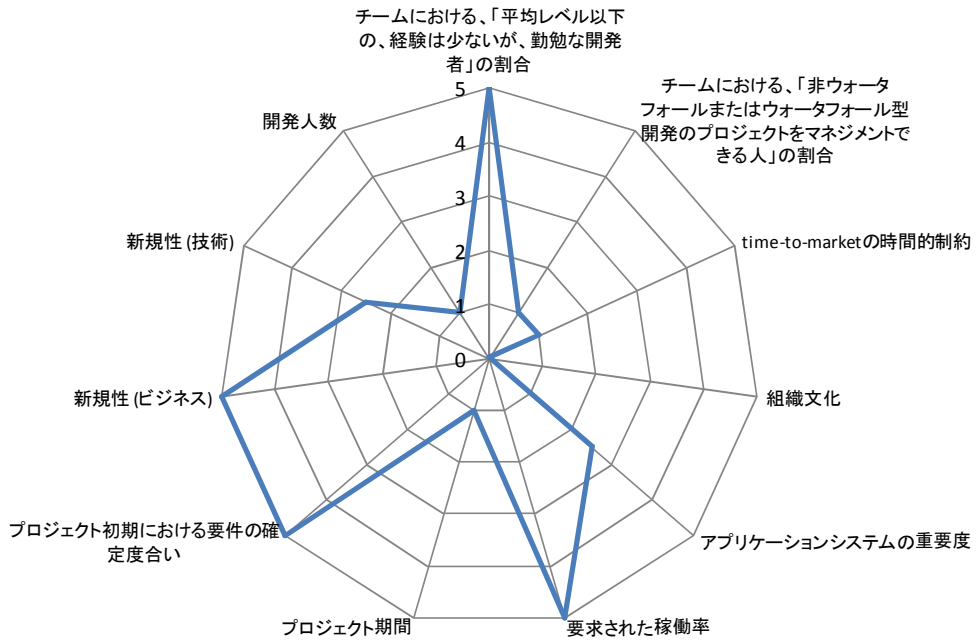
トをマネジメントできる人」の割合		
Time-to-market の時間的制約の厳しさ	3 ヶ月	2
組織文化	不明	0
システムの重要度の高さ	社外にユーザ（社外にも影響）	3.75
要求された稼働率の高さ	Web サイトとして 80%以上は確実と思われる。	5
プロジェクト期間の短さ	3 ヶ月	4
プロジェクト初期における要件確定度合いの低さ	要件定義の終了段階で、各画面の基本要件について概ね洗い出されている	1.25
ビジネスの新規性の豊かさ	常に新しいメディアやサービスへの挑戦。スタートアップの新事業、営業体制がまだできていないところが多い。	5
採用技術の新規性の豊かさ	特になし	1.25
開発人数の多さ	10 数名	3

(u)　アジャイル型開発の支援環境開発事例【P 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	10%	1
チームにおける「プロジェクトをマネジメントできる人」の割合	25%	2
Time-to-market の時間的制約の厳しさ	リリースまでは 1 年間。4 週間を 2 サイクルまとめたものをイテレーションとし、それを複数回実施する。	1
組織文化	90% （開発者としては自由にやりたいと思っている人が多い。コードを管理されていることを望んでいる人はあまりいないだろう。）	5
システムの重要度の高さ	一部の利用者の被害	2.5
要求された稼働率の高さ	（質問外）	0
プロジェクト期間の短さ	継続的に 4 年間行っているプロジェクト。1 年に一回のリリース（バージョンアップ）。	1
プロジェクト初期における要件確定度合いの低さ	コミュニティからの要望に基づくので、ある程度の要件は固まる。テーマと呼ばれる大枠での方向性（10 個）は変更しない。ただし、テーマ内の中小程度のものは変化する。	2.5
ビジネスの新規性の豊かさ	自社内の経験をベースに開発を行った。	3.75
採用技術の新規性の豊かさ	① REST アーキテクチャの全面採用 ② 並行してソフトウェア開発情報の標準的モデルの規定とその実装（OSLC イニシアティブ）	5
開発人数の多さ	多い。百数十人程度	5

(v)　共通 EDI 開発事例【Q 社】



軸	実績値	度数
チームにおける「平均レベル以下の、経験は少ないが勤勉な開発者」の割合	100%（アジャイル経験者は無し）	5
チームにおける「プロジェクトをマネジメントできる人」の割合	0%（アジャイル経験者は無し）	1
Time-to-market の時間的制約の厳しさ	5 ヵ月	5
組織文化	不明	0
システムの重要度の高さ	不具合の影響が複数社に及ぶ	2.5
要求された稼働率の高さ	1 日最大 24 回の納入に対応	5
プロジェクト期間の短さ	11 ヶ月（システム全体での開発期間）	1
プロジェクト初期における要件確定度合いの低さ	多業種にわたるシステムであるため、開発当初は仕様が全く分からない、かつ固められな	5

	い状態であった。	
ビジネスの新規性の豊かさ	全く新規の取組み	5
採用技術の新規性の豊かさ	アジャイル型の開発は初めての経験であった。	2.5
開発人数の多さ	15 名（リーダ：1 名、仕様担当：5 名、設計担当 3 名、開発担当：6 名）	1

４　各種手法の利害得失の調査

非ウォーターフォール型開発手法を適用して開発を実施する場合に生ずる、開発が成功裏に進む蓋然性を高める要因および、期待される成果を得ることを危うくする要因について、適用分野およびプラクティスごとに整理する。

４．１　適用分野別の利害得失の分析

３.４.１にて示した非ウォーターフォール型開発手法の ４ つの適用分野それぞれについて、収集したソフトウェア開発における選好因子と危険因子について整理する。また、プロジェクトの複雑性を考慮するため、複雑性の高い事例と低い事例を別平面にマッピングする。

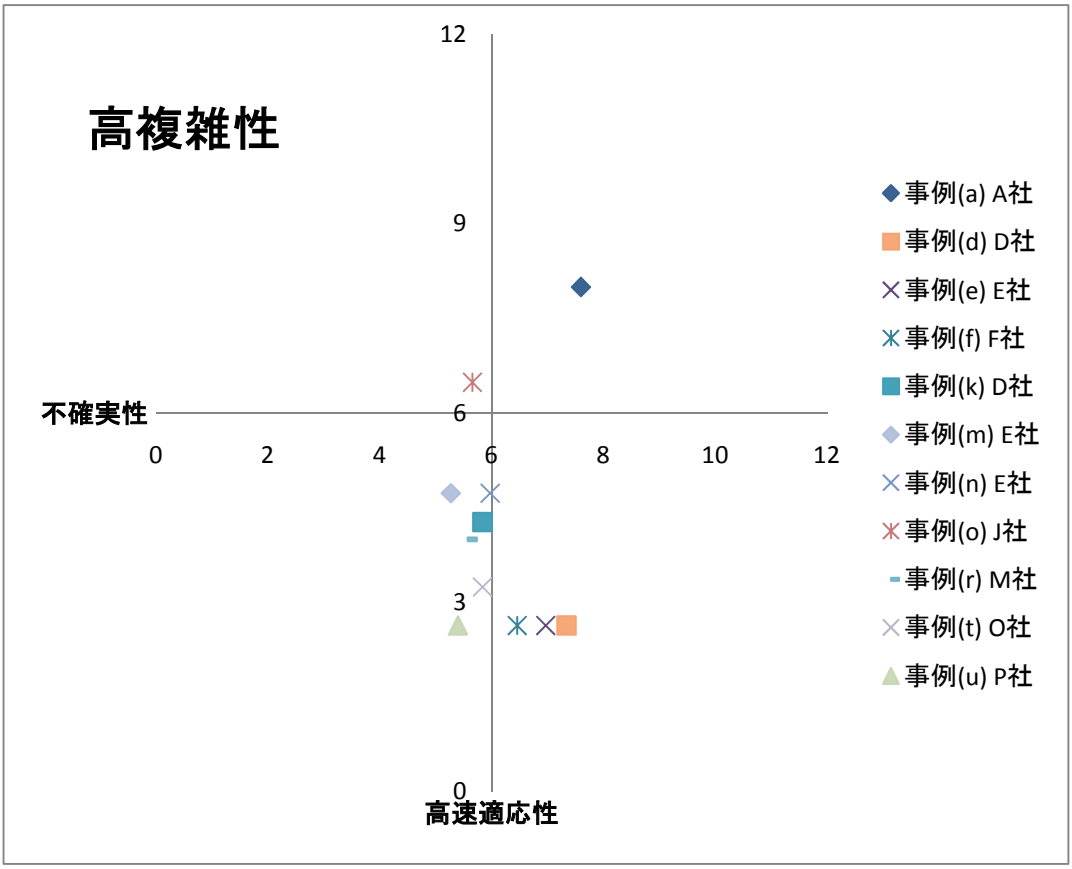


図 ８　事例の分類結果（高複雑性事例の分類）【再掲】

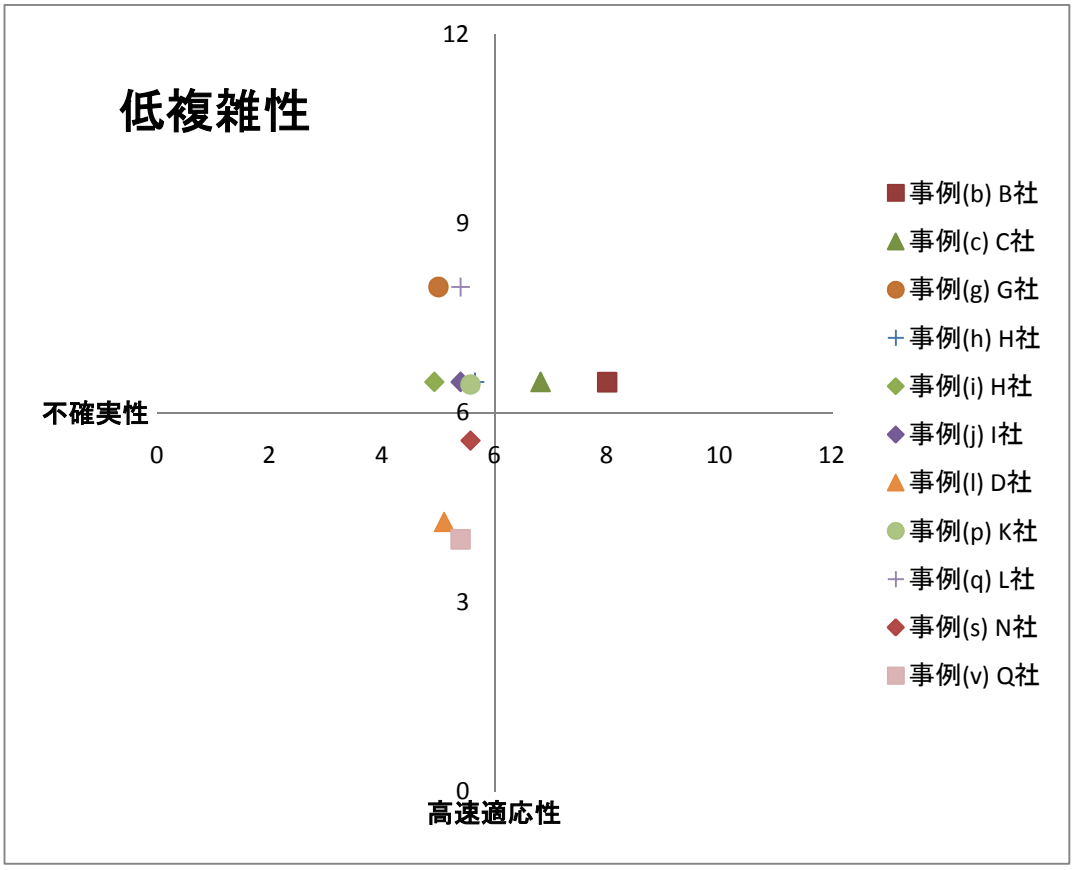


図 9　事例の分類結果（低複雑性事例の分類）【再掲】

高速適応性ならびに不確実性の特徴で分類した際に共通して現れる選好因子及び危険因子を記述する。

(1) 第 1 象限に関する選好因子および危険因子

■ 選好因子

➢ プロジェクトの進行過程上に、あらかじめ計画的にチェックポイントを設定し、変化する成り行きに沿ってコードを開発・テストしなければならない部分と、計画的に設計・構築する部分を識別・分離し、両者の成果物を統合してから、つぎの過程に進み、全ソリューションコードの統合および検証・妥当性確認が承認されてからリリースすることが許されていること

■ 危険因子

➢ 上記チェックポイントをどこに設ければよいかについての方法論が確立されていないこと

(2) 第 2 象限に関する選好因子および危険因子

■ 選好因子

- 成り行きに沿って生起する部分要求に順応して、定められた時間枠（タイムボックス）のなかで、コードを開発・テストし、先行システムに継続統合することが許されていること
- 上記の手順（イテレーション）の反復を許すことによって、先行イテレーションの成果物の改善が許されていること
- 軽量な CI 環境の支援下で、time-to-market を短縮できること

■ 危険因子

- 適合的に生起する部分要求が占める、全要求に対する位置づけが明確にされない（あらかじめ実装アーキテクチャを決めて、イテレーション成果物の割り付け先が決められる場合はよい）場合に、リリースされた全システムの最適化が中途半端に終わること
- 個々のイテレーション成果物を市場、または運用に投入することによってもたらされる価値を計量・評価する方法が確立されていないこと
- イテレーションで与えられる要求（問題）を、コード化するために必要な技術者資質およびスキルを評価する基準が確立されていないこと

(3) 第 3 象限に関する選好因子および危険因子

■ 選好因子

- 成り行きに沿って生起する部分要求に順応して、定められた時間枠（タイムボックス）のなかで、コードを開発・テストし、先行システムに継続統合することが許されていること
- 上記の手順（イテレーション）の反復を許すことによって、先行イテレーションの成果物の改善が許されていること

■ 危険因子

- イテレーションで与えられる要求（問題）を、コード化するために必要な技術者資質およびスキルを評価する基準が確立されていないこと

(4) 第 4 象限に関する選好因子および危険因子

■ 選好因子

- プロジェクトの進行過程上に、あらかじめ計画的にチェックポイントを設定し、変化する成り行きに沿ってコードを開発・テストしなければならない部分と、計画的に設計・構築する部分を識別・分離し、両者の成果物を統合してから、つぎの過程に進み、全ソリューションコードの統合および検証・妥当性確認が承認されてからリリースすることが許されていること

■ 危険因子

- 上記チェックポイントをどこに設ければよいかについての方法論が確立されていないこと
- 上記チェックポイントにおいて、アジャイル駆動プロセスを選択すべきか、計画駆動プロセスを選択すべきか、を判定するための方法論が確立されていないこと
- 進捗管理、コスト管理が複雑化すること

４．２ 各手法の代表的なプラクティスの影響評価

本項では、各手法に含まれる代表的なプラクティスや原則が、ソフトウェア開発における顧客ニーズへの対応、品質・信頼性、生産性、コスト、効率性（効果、性能）、規模、非機能要件への対応、あるいは保守の観点で、どのような影響を与えうるかを分析する。

表 7　プラクティスの影響分析

プラクティス名	利害得失
リリース計画ゲーム	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
	○：ユーザにとって必要なものを、最も効果的なタイミングでリリースするための計画が策定されるため、ユーザにとってのソフトウェア価値が最大化される
	○：開発途中に要求を変更することができる
イテレーション計画ゲーム	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○：顧客と積極的に話し合うことによって顧客から潜在的な要求

	を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
	○：ユーザにとって必要なものを、最も効果的なタイミングでリリースするための計画が策定されるため、ユーザにとってのソフトウェア価値が最大化される
	○：開発途中に要求を変更することができる
小規模で頻繁なリリース	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○: ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：ユーザにとっては、実際に動くものをベースに品質、進捗等を確認することができるため、状況の把握がしやすい
	○：細かい単位でユーザと成果物について合意を図れるため、安心して開発を進めていくことができる
	×：短い期間で定期的にリリースを重ねなければならないため、開発者にとっては負担が大きい
システムのメタファ	○： あらゆるデータ・条件においても正しく動作するかどうかを行うレビューのしやすさが向上するため、信頼性の副特性である成熟性の向上につながる
	○： レビューのしやすさが向上するため、機能性の副特性である

	正確性の向上につながる
シンプルデザイン	○： 必要なものだけを開発するため、コストを低く抑えることができる
	○： 必要なものだけを開発するため、開発スピードを向上させることができる
	○: シンプルでわかりやすい設計になり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： あらゆるデータ・条件においても正しく動作するかどうかを行うレビューのしやすさが向上するため、信頼性の副特性である成熟性の向上につながる
	○： レビューのしやすさが向上するため、機能性の副特性である正確性の向上につながる
	○： 無駄を排除することにより、要求される機能が正確に動作する可能性が高まるため、機能性の副特性である正確性の向上につながる
	○：必要最小限のものを分かりやすく設計するため、構築もスムーズに進められる
	○：なぜそのような設計になっているのか、設計者以外にも理解しやすいため、保守しやすい
	×：必要最小限のものしか設計しないため、将来的な変化に弱い場合がある
テスト駆動開発	○： あらゆるデータ・条件を想定した単体テストが通ったコードのみが存在することになるため、信頼性の副特性である成熟性の向上につながる
	○： 単体テストが通ったコードのみが存在することになるため、機能性の副特性である正確性の向上につながる
	○：テストの漏れを防ぐことができる
	○：プログラムや設計の問題を早期に発見できる
	×：変化が早いとテストが破たんする恐れがある
受け入れテスト	○： 統合テストが自動化されていることにより、ソフトウェアの変更があってもテストに要する労力を少なく抑えることができるため、保守性の副特性である試験性の向上につながる
	○： 顧客に受け入れテストを作成してもらうことにより、顧客の要求を正しく反映した製品になっているかどうかの確認ができる

	ため、機能性の副特性である合目的性の向上につながる
頻繁なリファクタリング	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○： あらゆるデータ・条件においても正しく動作するかどうかを行うレビューのしやすさが向上するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
	○： レビューのしやすさが向上するため、機能性の副特性である正確性の向上につながる
	○： 無駄を排除することにより、要求される機能が正確に動作する可能性が高まるため、機能性の副特性である正確性の向上につながる
	○：ソースコードが洗練されるため、性能が改善される
	○：また、可読性が高まり、修正や保守がしやすくなる
	×：リファクタリングそのものの手間がかかる
ペアプログラミング	○： ベテラン開発者と若い開発者がペアを組むことにより、若い開発者の教育につながる
	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
	○：ピアレビューを実施しながらの実装であるため、ソースコードの品質が高まる
	○：ペア間での知識移転が発生する（知識共有／人材育成効果）
	×：2 人で 1 人分の実装しか行えない

共同所有権	○: 他人が書いたコードも自由に修正できるため，確認をとりあう必要がなく，開発がスムーズに進む
	×: 他人に修正されたコードを，元々書いた人が理解しようとする時間がかかる（「コーディング規約」が実施されていれば，コードの理解は容易であるため，あまり問題にならない．また，「チーム全体が一緒に」が実施されていれば，質問があればすぐに聞くことができるので，あまり問題にならない）
継続的インテグレーション	○： 統合テストが自動化されていることにより、ソフトウェアの変更があってもテストに要する労力を少なく抑えることができるため、保守性の副特性である試験性の向上につながる
	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
	○: 結合・ビルド・テストが自動化されている
	×: システム規模が膨大だとテストに時間がかかる
	×：ライフサイクルの短いシステムの場合、CI 環境を構築するコストがメリットを上回る場合がある
持続可能なペース	○： 開発者の体調を良い状態で維持することができる
	○：開発を進めるのに適したリズムが生まれ、生産性が高まる
チーム全体が一緒に	○： チーム内で綿密なコミュニケーションを取ることができる
	○: 疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる
	○：適時、適切に要件の確認、知識の移転、共有が図れる
	×：プロジェクトルームの設置コストがかかる
コーディング規約	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： あらゆるデータ・条件においても正しく動作するかどうかを行うレビューのしやすさが向上するため、信頼性の副特性である成熟性の向上につながる
	○： レビューのしやすさが向上するため、機能性の副特性である

	正確性の向上につながる
	○：ソースコードの品質のばらつきがおさえられる
	○：他人が書いたコードの理解が容易になる
	×：コーディング規約を定めるためのコストがかかる
	×：コーディング規約を習得するためのコストがかかる
ゲーム前計画	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
スプリント計画	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
原則 30 日（カレンダー上)のイテレーション	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
自律的な組織化チ	○： 開発者が作業に集中できる環境を作り出すことができる

ーム	
スクラムミーティング	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
イテレーションに追加してはならない	○： 開発者が作業に集中できる環境を作り出すことができる
スクラムマスターのファイアウォール	○： 開発者が作業に集中できる環境を作り出すことができる
1 時間以内の判断	○： 判断に 100%の正確さを求めないことにより、迅速な意思決定を行うことができるため、開発スピードを向上させることができる
1 日以内の障害除去	○： あらゆるデータ・条件において、不具合を発見したらそれを迅速に修正するため、信頼性の副特性である成熟性の向上につながる
	○： 不具合を発見したらそれを迅速に修正するため、機能性の副特性である正確性の向上につながる
ニワトリとブタ	○： 開発者が作業に集中できる環境を作り出すことができる
共通の部屋	○： チーム内で綿密なコミュニケーションを取ることができる
	○: 疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる
日次ビルド	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
スプリントレビュー	○： 振り返りを実施することにより、これまでのイテレーションを反省し、以降のイテレーションに反省を生かすことができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性で

	ある理解性、習得性、運用性及び魅力性の向上につながる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
ドメイン・オブジェクト・モデリング	○： 問題領域の把握を行い、それを元に実現すべき機能の発見を行うことができるため、機能性の副特性である合目的性の向上につながる
feature 毎の開発	○：シンプルでわかりやすい設計になり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： 独立した構造を持つソフトウェアになり、一部の修正が他の部分へ与える影響を小さくすることができるため、保守性の副特性である安定性の向上につながる
	○： 細かい単位で開発を進めることにより、あらゆるデータ・条件においても不具合が入り込みづらい開発が可能となるため、信頼性の副特性である成熟性の向上につながる
	○： 細かい単位で開発を進めることにより、不具合が入り込みづらい開発が可能となるため、機能性の副特性である正確性の向上につながる
	○： 独立した構造を持つソフトウェアになり、移植によって一部を修正する必要が生じても、その修正が他の部分へ与える影響を小さくすることができるため、移植性の副特性である設置性の向上につながる

クラスの責任者	○： 機能ごとに責任者を割り当てることにより、あらゆるデータ・条件においても機能が正確に動作することを実現する責任が明確になるため、信頼性の副特性である成熟性の向上につながる
	○： 機能ごとに責任者を割り当てることにより、正確に機能を実現する責任が明確になるため、機能性の副特性である正確性の向上につながる
feature チーム	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
定期ビルド	○： 統合テストが自動化されていることにより、ソフトウェアの変更があってもテストに要する労力を少なく抑えることができるため、保守性の副特性である試験性の向上につながる
	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
無駄をなくす	○： 必要なものだけを開発するため、コストを低く抑えることができる
	○： 必要なものだけを開発するため、開発スピードを向上させることができる
	○： あらゆるデータ・条件においても正しく動作するかどうかを行うレビューのしやすさが向上するため、信頼性の副特性である成熟性の向上につながる
	○： レビューのしやすさが向上するため、機能性の副特性である正確性の向上につながる
	○： 無駄を排除することにより、要求される機能が正確に動作する可能性が高まるため、機能性の副特性である正確性の向上につながる
品質を作りこむ	○： あらゆるデータ・条件において、不具合を発見したらそれを迅速に修正するため、信頼性の副特性である成熟性の向上につながる

	○： 不具合を発見したらそれを迅速に修正するため、機能性の副特性である正確性の向上につながる
知識を作り出す(顧客からのフィードバック)	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
知識を作り出す(テストからのフィードバック)	○： 統合テストが自動化されていることにより、ソフトウェアの変更があってもテストに要する労力を少なく抑えることができるため、保守性の副特性である試験性の向上につながる
	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
知識を作り出す(追加容易なアーキテクチャ)	○： ソフトウェアに新規機能を追加しやすい構造になるため、保守性の副特性である変更性の向上につながる
速く提供する	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
短くてリッチなコミュニケーションパスがある	○： チーム内で綿密なコミュニケーションを取ることができる
	○: 疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる
1~3 か月以内に出	○： 早い段階でリリースすることにより、顧客に価値を早く提供

荷する	することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
作業成果物の所有者を決める	○： 機能ごとに責任者を割り当てることにより、あらゆるデータ・条件においても機能が正確に動作することを実現する責任が明確になるため、信頼性の副特性である成熟性の向上につながる
	○： 機能ごとに責任者を割り当てることにより、正確に機能を実現する責任が明確になるため、機能性の副特性である正確性の向上につながる
頻繁なリリース	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から

	潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
反省と改善	○： 振り返りを実施することにより、これまでのイテレーションを反省し、以降のイテレーションに反省を生かすことができる
浸透したコミュニケーション	○： チーム内で綿密なコミュニケーションを取ることができる
個人の安全	○： 開発者が作業に集中できる環境を作り出すことができる
集中	○： 開発者が作業に集中できる環境を作り出すことができる
エキスパートユーザとのコミュニケーション	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○: ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境	○： 統合テストが自動化されていることにより、ソフトウェアの変更があってもテストに要する労力を少なく抑えることができるため、保守性の副特性である試験性の向上につながる
	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
360 度の探索	○： 問題領域の把握を行い、それを元に実現すべき機能の発見を行うことができるため、機能性の副特性である合目的性の向上に

	つながる
早期の成功体験	○： 開発者が自信を持つことができる
反省ワークショップ	○： 振り返りを実施することにより、これまでのイテレーションを反省し、以降のイテレーションに反省を生かすことができる
集中的な計画	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
アジャイルインタラクション設計	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
プロセスの簡易版	○： 開発者に新しい開発プロセスを慣れさせることができる
Side-by-side プログラミング	○： ベテラン開発者と若い開発者がペアを組むことにより、若い開発者の教育につながる
	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
反復型開発	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる

	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
要求管理	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
コンポーネント・アーキテクチャーの使用	○： 信頼性の高いコンポーネントを再利用することにより、信頼性の向上につながる
	○： 機能性が高いコンポーネントを再利用することにより、機能性の向上につながる
品質の継続的検証	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
プロトタイプ専門家の派遣	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○: 顧客が製品をどのように利用しているかを専門家が観察することにより、顧客が使いやすい製品になるにはどうすればいいかを把握することができるため、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○: 顧客が製品をどのように利用しているかを専門家が観察することにより、顧客が望む処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる

	○：顧客が製品をどのように利用しているかを専門家が観察することにより、顧客の隠れた要求を引き出すことが可能になるため、機能性の副特性である合目的性の向上につながる
タイムボックス	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
MoSCoW	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
プロトタイピング	○： 製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ワークショップ	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
積極的なユーザ関与が絶対に必要である。	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる

頻繁な引き渡しが鍵である。	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
反復的で漸増的な引き渡しが不可欠である。	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
プロジェクトライフサイクル全体で統合されたテスト	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる

が期待される。	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
すべての利害関係者間の協力が不可欠である。	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
プロジェクト計画	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
リスク管理	○: あらゆるデータ・条件における欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、信頼性の副特性である成熟性の向上につながる
	○: 欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、機能性の副特性である正確性の向上につながる
DSDM プロジェクトの計測	○: あらゆるデータ・条件における欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、信頼性の副特性である成熟性の向上につながる
	○: 欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、機能性の副特性である正確性の向上につながる
品質管理	○: 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○: 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○: 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○: 頻繁に製品の動作を確認してもらうことによって、顧客から

	潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
	○：レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
実現可能性調査	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ビジネス調査	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
機能モデルイテレーション	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○：製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○：製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○：製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○：製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
設計・構築モデルイテレーション	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○：製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる

	○： 製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
重点要求トップ 10 を定義する	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○： 実現すべき機能に順位づけを行うことにより、顧客自身が重要な機能を把握することを促進するため、機能性の副特性である合目的性の向上につながる
パフォーマンス仕様を定義する	○： 効率性に関する要求を明確に定義し、把握することができるため、効率性の向上につながる
明確で（可能なら）測定可能な仕様を定義する	○： 明確な要求仕様を記述することにより、実現すべき機能における曖昧さを排除することができるため、機能性の副特性である合目的性の向上につながる
Planguage で仕様を記述する	○： 明確な要求仕様を記述することにより、実現すべき機能における曖昧さを排除することができるため、機能性の副特性である合目的性の向上につながる
進化型プロジェクトマネジメント	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要

	求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
進化型出荷	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○: ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
設計仕様を定義する	○： 明確な要求仕様を記述することにより、実現すべき機能における曖昧さを排除することができるため、機能性の副特性である合目的性の向上につながる
影響評価	○： 効率性に関する要求を明確に定義し、把握することができるため、効率性の向上につながる
設計アイディアがどう要求を満たすかを記述する	○： 頻繁にテストをすることにより各機能があらゆるデータ・条件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○: 頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
テストの測定基準を要求仕様に記述する	○： 効率性に関する要求を明確に定義し、把握することができるため、効率性の向上につながる

早期のインスペクションによる仕様品質のコントロール	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○： 効率性に関する要求を明確に定義し、把握することができるため、効率性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
クライアント駆動型計画	○: 疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる
頻繁な出荷	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○: ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
問題の分割	○: シンプルでわかりやすい設計になり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： 細かい単位で開発を進めることにより、あらゆるデータ・条件においても不具合が入り込みづらい開発が可能となるため、信頼性の副特性である成熟性の向上につながる
	○： 細かい単位で開発を進めることにより、不具合が入り込みづらい開発が可能となるため、機能性の副特性である正確性の向上につながる

測定と褒賞	○： 正しい評価方法を用いてチームを褒賞することにより、チームメンバのモチベーションを向上させることができる
シェフモードとコックモードの分離	○： ソフトウェアの変更に対応しやすい構造になるため、保守性の副特性である変更性の向上につながる
	○： 独立した構造を持つソフトウェアになり、一部の修正が他の部分へ与える影響を小さくすることができるため、保守性の副特性である安定性の向上につながる
	○： 独立した構造を持つソフトウェアになり、移植によって一部を修正する必要が生じても、その修正が他の部分へ与える影響を小さくすることができるため、移植性の副特性である設置性の向上につながる
	○： ソフトウェアの変更に対応しやすい構造になり、移植によって一部を修正する必要が生じても、比較的容易に修正を行うことができるため、移植性の副特性である設置性の向上につながる
DSM を用いたアーキテクチャ変換	○： ソフトウェアの変更に対応しやすい構造になるため、保守性の副特性である変更性の向上につながる
	○： 独立した構造を持つソフトウェアになり、一部の修正が他の部分へ与える影響を小さくすることができるため、保守性の副特性である安定性の向上につながる
	○： 独立した構造を持つソフトウェアになり、移植によって一部を修正する必要が生じても、その修正が他の部分へ与える影響を小さくすることができるため、移植性の副特性である設置性の向上につながる
	○： ソフトウェアの変更に対応しやすい構造になり、移植によって一部を修正する必要が生じても、比較的容易に修正を行うことができるため、移植性の副特性である設置性の向上につながる
ビジネスとソフトウェア構築の直結	○： 製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる

	○： 製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
セルの構築と独立性	○： 細かい単位で開発を進めることにより、あらゆるデータ・条件においても不具合が入り込みづらい開発が可能となるため、信頼性の副特性である成熟性の向上につながる
	○： 細かい単位で開発を進めることにより、不具合が入り込みづらい開発が可能となるため、機能性の副特性である正確性の向上につながる
専門技術者の育成	○： 専門技術者を育成することにより、今後の開発に生かすことができる
	○： あらゆるデータ・条件における欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、信頼性の副特性である成熟性の向上につながる
	○： 専門技術者を担当させることにより、不具合が入り込む可能性を減少させるため、機能性の副特性である正確性の向上につながる
セル内の平等責任	○： チーム内で綿密なコミュニケーションを取ることができる
タイムボックス	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
80%ルール	○： 判断に 100%の正確さを求めないことにより、迅速な意思決定を行うことができるため、開発スピードを向上させることができる
要件確認会	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から

	潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ユーザ動作確認	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ピックアップレビュー	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○：レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
JAD 開発	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
顧客のフォーカスグループレビュー	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる

	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ソフトウェアインスペクション	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
プロジェクトの事後評価	○： 振り返りを実施することにより、これまでのイテレーションを反省し、以降のイテレーションに反省を生かすことができる
反復的に開発する	○: ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能から順番に提供することができる
	○： 優先順位が高い機能から開発を行うため、重要な機能ほど予算内で開発できる可能性が高くなる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
要求を管理する	○： 要求を厳密に管理することによって、顧客の要求漏れを防ぐ可能性が高まるため、機能性の副特性である合目的性の向上につながる
モデリング	○： 問題領域の把握を行い、それを元に実現すべき機能の発見を行うことができるため、機能性の副特性である合目的性の向上につながる
継続的に品質を検	○： 頻繁にテストをすることにより各機能があらゆるデータ・条

証する	件においても正しい計算や処理を行うことを確認するため、信頼性の副特性である成熟性の向上につながる
	○：頻繁にテストをすることにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
変更を管理する	○： 要求を厳密に管理することによって、顧客の要求漏れを防ぐ可能性が高まるため、機能性の副特性である合目的性の向上につながる
協調的な開発	○： チーム内で綿密なコミュニケーションを取ることができる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○：顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
	○：疑問点をすぐに顧客に確認することによって正しい要求の把握が可能となるため、機能性の副特性である合目的性の向上につながる
	○：レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
開発後のことを考慮する	○： 社内の標準にあった開発を行っているため、保守性の副特性である標準適合性の向上につながる
動くソフトウェアを定期的に納品する	○： 早い段階でリリースすることにより、顧客に価値を早く提供することができる
	○：ユーザに対して動くソフトウェアを頻繁に見せる必要があるため、開発者の緊張感を維持することができる
	○： 頻繁に製品の動作を確認してもらい、顧客が使いやすい製品になるための要求を出してもらうことにより、使用性の副特性である理解性、習得性、運用性及び魅力性の向上につながる
	○： 頻繁に製品の動作を確認してもらい、処理時間などの要求を把握することができるため、効率性の副特性である時間効率性の向上につながる
	○： 頻繁に製品の動作を確認してもらうことによって、顧客の要求を確認することができるため、機能性の副特性である合目的性の向上につながる

	○： 頻繁に製品の動作を確認してもらうことによって、顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
リスクを管理する	○： あらゆるデータ・条件における欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、信頼性の副特性である成熟性の向上につながる
	○： 欠陥の把握を行うことで、欠陥を見逃す可能性を減少させるため、機能性の副特性である正確性の向上につながる
ユーザ一体型開発	○: 顧客と積極的に話し合うことによって顧客から潜在的な要求を抽出することを促進するため、機能性の副特性である合目的性の向上につながる
ペアプログラミング	○： ベテラン開発者と若い開発者がペアを組むことにより、若い開発者の教育につながる
	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
	○： レビューにより各機能があらゆるデータ・条件においても正しく動作することを確認するため、信頼性の副特性である成熟性の向上につながる
	○: レビューにより各機能が正しい計算や処理を行うことを確認するため、機能性の副特性である正確性の向上につながる
コミュニケーション技術	○： チーム内で綿密なコミュニケーションを取ることができる
非分業の原則	○： 作業を分業しないことにより、チームワークと分業を体感することができる
作り捨てとコピーライト	○： 機能ごとに責任者を割り当てることにより、あらゆるデータ・条件においても機能が正確に動作することを実現する責任が明確になるため、信頼性の副特性である成熟性の向上につながる
	○： 機能ごとに責任者を割り当てることにより、正確に機能を実現する責任が明確になるため、機能性の副特性である正確性の向上につながる
ワンプログラム・ワンフロー	○： シンプルでわかりやすいコードになり、保守する部分を識別し、特定するまでの労力が少なくて済むため、保守性の副特性である解析性の向上につながる
モデリング	○： 問題領域の把握を行い、それを元に実現すべき機能の発見を

	行うことができるため、機能性の副特性である合目的性の向上につながる

５ 非ウォーターフォール型開発の品質・信頼性及び開発管理についての課題整理と提言

ウォーターフォール型開発に比べて問題点と指摘されている品質・信頼性及び開発管理について、実態を調査・分析するとともに課題を整理する。

５．１ 非ウォーターフォール型開発の品質・信頼性及び開発管理の課題整理と分析

ここでは、非ウォーターフォール型開発におけるプラクティスをそのまま適用した場合に、品質・信頼性及び開発管理においてそれぞれ課題と成り得る事柄を整理する。

まず品質・信頼性及び開発管理に関わるプラクティスの抽出・整理を行う。ここで品質・信頼性及び開発管理の観点において、非ウォーターフォールで用意されているプラクティスが十分であるかどうかを確認する。

次に、世間一般に言われている、非ウォーターフォール型開発における品質・信頼性及び開発管理についての不安や課題をインターネットなどの文献から抽出・整理する。

そしてこれらの整理から、非ウォーターフォール型開発において品質・信頼性及び開発管理にどのような課題があるかを分析する。

最後に、品質・信頼性及び開発管理についての提言を行う。

５．１．１ 品質・信頼性

非ウォーターフォール型開発において、品質・信頼性向上につながるプラクティスを抽出する。これにより、非ウォーターフォール型開発が、どのように品質・信頼性を確保しているかを把握することができる。

その上で、これらのプラクティスを適用するための前提条件を導出する。たとえば、ユーザからのフィードバックを求めるプラクティスを適用する場合は、ユーザの積極的な関与が前提となる。ユーザの積極的な関与が望めない場合は、このプラクティスの適用は難しい。また、継続的インテグレーションというプラクティスを実施するためには、通常それを支援するツールが必要であると考えられる。ツールを用意できない場合は、このプラクティスを実施することは困難になる。

(1) 品質・信頼性に関わるプラクティスの抽出

非ウォーターフォール型開発において、品質・信頼性向上につながるプラクティスを抽出したものを以下に示す。ここでは、各手法におけるプラクティスの差異も併せて確認するため、各手法のプラクティス ID を付加している。

品質特性として、ISO/IEC9126 に定義されている、機能性・信頼性・使用性・効率

性・保守性・移植性の側面で整理する。さらに、違う側面における整理の仕方として、迅速な不具合の除去・テスト・レビュー・品質指標・ユーザにおける品質管理の側面でも整理を行う。

(a)　品質特性によるプラクティスの整理

① 機能性

表 8 からわかるとおり、非ウォーターフォール型開発では、顧客と頻繁にやりとりをすることによって、顧客から真の要求を導出するためのプラクティスが多い。これらは、機能性の副特性である合目的性の向上につながる。

また、テストやレビューを実施することにより、各機能が正しい計算や処理を行うことを保証している。これらは、機能性の副特性である、正確性の向上につながる。

表 8　機能性向上に関わるプラクティス

プラクティス ID	プラクティス名
XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
XP-3	小規模で頻繁なリリース
XP-4	システムのメタファ
XP-5	シンプルデザイン
XP-6	テスト駆動開発
XP-7	受け入れテスト
XP-8	頻繁なリファクタリング
XP-9	ペアプログラミング
XP-11	継続的インテグレーション
XP-13	チーム全体が一緒に
XP-14	コーディング規約
Scrum-1	ゲーム前計画
Scrum-2	スプリント計画
Scrum-3	原則 30 日（カレンダー上)のイテレーション
Scrum-6	スクラムミーティング
Scrum-10	1 日以内の障害除去
Scrum-13	共通の部屋
Scrum-14	日次ビルド
Scrum-15	スプリントレビュー

FDD-1	ドメイン・オブジェクト・モデリング
FDD-2	feature 毎の開発
FDD-3	クラスの責任者
FDD-4	feature チーム
FDD-7	定期ビルド
Lean-1	無駄をなくす
Lean-2	品質を作りこむ
Lean-3	知識を作り出す（顧客からのフィードバック）
Lean-4	知識を作り出す（テストからのフィードバック）
C.Clear-1	短くてリッチなコミュニケーションパスがある
C.Clear-2	1~3 か月以内に出荷する
C.Clear-5	作業成果物の所有者を決める
C.Clear-11	頻繁なリリース
C.Clear-16	エキスパートユーザとのコミュニケーション
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
C.Clear-18	360 度の探索
C.Clear-25	集中的な計画
C.Clear-28	アジャイルインタラクション設計
C.Clear-30	Side-by-side プログラミング
RUP-1	反復型開発
RUP-2	要求管理
RUP-3	コンポーネント・アーキテクチャーの使用
RUP-5	品質の継続的検証
OP-2	プロトタイプ専門家の派遣
DSDM-2	MoSCoW
DSDM-3	プロトタイピング
DSDM-4	ワークショップ
DSDM-5	積極的なユーザ関与が絶対に必要である。
DSDM-7	頻繁な引き渡しが鍵である。
DSDM-9	反復的で漸増的な引き渡しが不可欠である。
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待される。
DSDM-13	すべての利害関係者間の協力が不可欠である。

DSDM-14	プロジェクト計画
DSDM-15	リスク管理
DSDM-16	DSDM プロジェクトの計測
DSDM-18	品質管理
DSDM-19	実現可能性調査
DSDM-20	ビジネス調査
DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-2	重点要求トップ 10 を定義する
Evo-5	明確で（可能なら）測定可能な仕様を定義する
Evo-6	Planguage で仕様を記述する
Evo-7	進化型プロジェクトマネジメント
Evo-8	進化型出荷
Evo-10	設計仕様を定義する
Evo-12	設計アイディアがどう要求を満たすかを記述する
Evo-14	早期のインスペクションによる仕様品質のコントロール
Evo-18	クライアント駆動型計画
Evo-21	頻繁な出荷
Evo-22	問題の分割
Cell-4	ビジネスとソフトウェア構築の直結
Cell-5	セルの構築と独立性
Cell-6	専門技術者の育成
SWT-4	要件確認会
SWT-5	ユーザ動作確認
SWT-6	ピックアップレビュー
ASD-6	JAD 開発
ASD-7	顧客のフォーカスグループレビュー
ASD-8	ソフトウェアインスペクション
EUP-1	反復的に開発する
EUP-2	要求を管理する
EUP-4	モデリング
EUP-5	継続的に品質を検証する
EUP-6	変更を管理する
EUP-7	協調的な開発

EUP-9	動くソフトウェアを定期的に納品する
EUP-10	リスクを管理する
Uni-1	ユーザ一体型開発
Uni-2	ペアプログラミング
Uni-5	作り捨てとコピーライト
RAD-3	モデリング

② 信頼性

表 9 からわかるとおり、非ウォーターフォール型開発においては、頻繁にテストやレビューを実施することを推奨している。これらは、信頼性の副特性である成熟性や障害許容性の向上につながる。

表 9　信頼性向上に関わるプラクティス

プラクティス ID	プラクティス名
XP-4	システムのメタファ
XP-5	シンプルデザイン
XP-6	テスト駆動開発
XP-8	頻繁なリファクタリング
XP-9	ペアプログラミング
XP-11	継続的インテグレーション
XP-14	コーディング規約
Scrum-6	スクラムミーティング
Scrum-10	1 日以内の障害除去
Scrum-14	日次ビルド
Scrum-15	スプリントレビュー
FDD-2	feature 毎の開発
FDD-3	クラスの責任者
FDD-4	feature チーム
FDD-7	定期ビルド
Lean-1	無駄をなくす
Lean-2	品質を作りこむ
Lean-4	知識を作り出す（テストからのフィードバック）
C.Clear-5	作業成果物の所有者を決める
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境

C.Clear-28	アジャイルインタラクション設計
C.Clear-30	Side-by-side プログラミング
RUP-3	コンポーネント・アーキテクチャーの使用
RUP-5	品質の継続的検証
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待される。
DSDM-15	リスク管理
DSDM-16	DSDM プロジェクトの計測
DSDM-18	品質管理
Evo-12	設計アイディアがどう要求を満たすかを記述する
Evo-14	早期のインスペクションによる仕様品質のコントロール
Evo-22	問題の分割
Cell-5	セルの構築と独立性
Cell-6	専門技術者の育成
SWT-6	ピックアップレビュー
ASD-8	ソフトウェアインスペクション
EUP-5	継続的に品質を検証する
EUP-7	協調的な開発
EUP-10	リスクを管理する
Uni-2	ペアプログラミング
Uni-5	作り捨てとコピーライト

③ 使用性

表 10 からわかるとおり、非ウォーターフォール型開発では、顧客と頻繁にやりとりをすることによって、顧客が使いやすい機能を実現するためのプラクティスが多い。これらは、使用性の副特性である理解性の向上につながる。

表 10　使用性向上に関わるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
Scrum-3	原則 30 日（カレンダー上)のイテレーション
Scrum-15	スプリントレビュー
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース

C.Clear-16	エキスパートユーザとのコミュニケーション
RUP-1	反復型開発
OP-2	プロトタイプ専門家の派遣
DSDM-3	プロトタイピング
DSDM-7	頻繁な引き渡しが鍵である。
DSDM-9	反復的で漸増的な引き渡しが不可欠である。
DSDM-18	品質管理
DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-7	進化型プロジェクトマネジメント
Evo-8	進化型出荷
Evo-21	頻繁な出荷
Cell-4	ビジネスとソフトウェア構築の直結
SWT-4	要件確認会
SWT-5	ユーザ動作確認
ASD-7	顧客のフォーカスグループレビュー
EUP-1	反復的に開発する
EUP-9	動くソフトウェアを定期的に納品する

④ 効率性

非ウォーターフォール型開発における表 11 に示したプラクティスの多くは、ユーザに対して、実際にソフトウェアを利用してもらうことを求めている。これにより、効率性の副特性である時間効率性に関する要求を導き出すことが可能となる。

表 11　効率性向上に関わるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
Scrum-3	原則 30 日（カレンダー上)のイテレーション
Scrum-15	スプリントレビュー
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-16	エキスパートユーザとのコミュニケーション
RUP-1	反復型開発

OP-2	プロトタイプ専門家の派遣
DSDM-3	プロトタイピング
DSDM-7	頻繁な引き渡しが鍵である。
DSDM-9	反復的で漸増的な引き渡しが不可欠である。
DSDM-18	品質管理
DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-4	パフォーマンス仕様を定義する
Evo-7	進化型プロジェクトマネジメント
Evo-8	進化型出荷
Evo-11	影響評価
Evo-13	テストの測定基準を要求仕様に記述する
Evo-14	早期のインスペクションによる仕様品質のコントロール
Evo-21	頻繁な出荷
Cell-4	ビジネスとソフトウェア構築の直結
SWT-4	要件確認会
SWT-5	ユーザ動作確認
ASD-7	顧客のフォーカスグループレビュー
EUP-1	反復的に開発する
EUP-9	動くソフトウェアを定期的に納品する

⑤ 保守性

表 12 からわかるとおり、非ウォーターフォール型開発では、開発過程の各局面で必要な機能のみを実装することを推奨している。つまり、無駄な機能がなく、シンプルな設計を行うことを方針としている。これらは、保守性の副特性である解析性の向上につながる。

また、シンプルな設計や、新規機能が追加しやすいアーキテクチャは、一般にソフトウェアの変更に柔軟に対応可能である。これらは保守性の副特性である変更性の向上につながる。

表 12　保守性向上に関わるプラクティス

プラクティス ID	プラクティス名
XP-5	シンプルデザイン
XP-7	受け入れテスト

XP-8	頻繁なリファクタリング
XP-9	ペアプログラミング
XP-11	継続的インテグレーション
XP-14	コーディング規約
FDD-2	feature 毎の開発
FDD-7	定期ビルド
Lean-4	知識を作り出す（テストからのフィードバック）
Lean-6	知識を作り出す（追加容易なアーキテクチャ）
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
C.Clear-30	Side-by-side プログラミング
RUP-3	コンポーネント・アーキテクチャーの使用
Evo-16	オープンエンドのアーキテクチャ
Evo-22	問題の分割
Cell-2	DSM を用いたアーキテクチャ変換
EUP-8	開発後のことを考慮する
Uni-2	ペアプログラミング
Uni-6	ワンプログラム・ワンフロー

⑥ 移植性

表 13 からわかるとおり、非ウォーターフォール型開発においては、ソフトウェアの変更に対応しやすい構造にすることが強く要求されている。したがって、移植によって一部を修正する必要が生じても、比較的容易に修正することができる。これは移植性の副特性である設置性の向上につながる。

表 13　移植性向上に関わるプラクティス

プラクティス ID	プラクティス名
FDD-2	feature 毎の開発
RUP-3	コンポーネント・アーキテクチャーの使用
Evo-16	オープンエンドのアーキテクチャ
Cell-2	DSM を用いたアーキテクチャ変換

(b)　品質管理の側面によるプラクティスの整理

① 不具合の除去に関する特徴

表 14 からわかるとおり、非ウォーターフォール型開発においては、不具合が潜在している期間をできる限り短くすることを目的としているプラクティスが多い。基本的に迅速な不具合の除去の実現を目指している。

表 14　不具合の除去に関わるプラクティス

プラクティス ID	プラクティス名
XP-9	ペアプログラミング
XP-11	継続的インテグレーション
Scrum-10	1 日以内の障害除去
Scrum-14	日次ビルド
FDD-7	定期ビルド
Lean-2	品質を作りこむ
Lean-4	知識を作り出す（テストからのフィードバック）
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待される
Uni-2	ペアプログラミング

② テストに関する特徴

表 15 からわかるとおり、非ウォーターフォール型開発においては、頻繁にテストを実施することを要求するプラクティスが多い。

表 15　テストに関わるプラクティス

プラクティス ID	プラクティス名
XP-6	テスト駆動開発
XP-7	受け入れテスト
XP-11	継続的インテグレーション
Scrum-14	日次ビルド
Lean-4	知識を作り出す（テストからのフィードバック）
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
RUP-5	品質の継続的検証
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待され

	る
DSDM-18	品質管理
Evo-13	テストの測定基準を要求仕様に記述する

③ レビューに関する特徴

表 16 からわかるとおり、非ウォーターフォール型開発において、頻繁なレビューの繰り返しをプラクティスとしている例が多い。レビューには、開発者同士で行うものと、顧客を含めたレビューに分けられるが、非ウォーターフォール型開発においては特に、顧客も含めたレビューを行うことを重視している。

表 16　レビューに関わるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
XP-9	ペアプログラミング
Scrum-15	スプリントレビュー
FDD-5	インスペクション
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-30	Side-by-side プログラミング
DSDM-7	頻繁な引き渡しが鍵である
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-18	品質管理
Evo-8	進化型出荷
Evo-14	早期のインスペクションによる仕様品質のコントロール
ASD-8	ソフトウェアインスペクション
Uni-2	ペアプログラミング

④ 品質指標に関する特徴

表 17 は、非ウォーターフォール型開発において、品質指標に関わる観点を明示的に示しているプラクティスを取り出したものである。表から、明示的に示しているプラクティスの数は少ないということがわかる。しかし、実際に、非ウォーターフォール型開発を行っている事例において、何らかの品質指標を用いていたり、テストカバレッジを大きくしたりする努力が行われている。

表 17　品質指標に関わるプラクティス

プラクティス ID	プラクティス名
DSDM-16	DSDM プロジェクトの計測
DSDM-18	品質管理
Evo-13	テストの測定基準を要求仕様に記述する

⑤ ユーザにおける品質管理に関する特徴

表 18 からわかるとおり、非ウォーターフォール型開発においては、ユーザにおける品質管理を実現するプラクティスが多く存在する。ユーザは、実際に動くソフトウェアを見て利用することにより、機能性や使用性などの品質特性を確認することが可能である。基本的に状況が判断しづらいといわれるソフトウェア開発に対して、実際に動くもので状況を確認することを目指していると考えられる。

表 18　ユーザにおける品質管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
XP-7	受け入れテスト
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
OP-1	進化型プロトタイプの作成
OP-2	プロトタイプ専門家の派遣
DSDM-3	プロトタイピング
DSDM-7	頻繁な引き渡しが鍵である
DSDM-8	受け入れの主たる条件は現在のビジネスニーズを満たす機能の引き渡しである
DSDM-9	反復的で漸増的な引き渡しが不可欠である
Evo-8	進化型出荷
Evo-9	出荷したソリューションの影響を測定する
Evo-21	頻繁な出荷
SWT-4	要件確認会
SWT-5	ユーザ動作確認
ASD-7	顧客のフォーカスグループレビュー
EUP-1	反復的に開発する

EUP-9	動くソフトウェアを定期的に納品する

非ウォーターフォール型開発手法それぞれを見ると、重視している品質特性に偏りが見られることもある。だが、非ウォーターフォール型開発手法の全体で見れば、各品質特性を全て網羅している。

(2) 品質・信頼性に関わるプラクティス実施のための前提条件

品質・信頼性に関わるプラクティスの整理により、改めて以下の前提条件が必要であると考えられる。

(a) 積極的なユーザの関与

機能性や効率性、使用性の向上につながるプラクティスの多くは、継続的に、ユーザからの積極的なフィードバックを求めている。そのため、ユーザが積極的に関われないような状況である場合は、これらのプラクティスを実施することは困難である。このような状況はたとえば、ユーザが複数のプロジェクトを兼任している場合や、遠隔地にいる場合などが考えられる。

(b) テストツールの存在

機能性や信頼性の向上につながるプラクティスの多くは、継続的なテストの実施を求めているものが多い。人手によって頻繁にテストを実施することは負担が大きいため、テストツールの存在が前提となる。したがって、テストツールを用意できないような状況である場合は、これらのプラクティスを実施することは困難と考えられる。

(c) テストに係るコストの低減

上記と同様、機能性や信頼性の向上につながるプラクティスの多くは、継続的なテストの実施を求めているものが多い。したがって、テストを行う際に計算時間や費用面において非常に高いコストがかかる場合は、これらのプラクティスを実施することは困難である。

(d) 変更が容易なアーキテクチャの存在

機能性や効率性、使用性の向上につながるプラクティスの多くは、継続的に、ユーザからの要件変更を受け入れている。したがって、変更が容易なアーキテクチャの存在が必須であり、存在しなければ作成する必要がある。

(e) スキルを持った技術者の存在

以上の前提を実現するには、そのためのスキルを持った開発技術者の存在が必要である。開発メンバの全員が同等のスキルをもつ必要はないが、技術的・実施上のリーダまたはファシリテータが必要である。

(3) 品質・信頼性に関わる課題の仮説

Web などの文献から、非ウォーターフォール型開発において品質・信頼性に課題があると指摘される場合に、一般的にどのような認識が示されているかについて抽出・整理を行う。

(a) チーム規模の限界

開発に関与する人数が増えると、開発を適切に進めていくことが困難になるとされている。実際、開発者が 10 数名を越える規模の開発では、システムを複数のサブシステムに分割し、それぞれ開発を担当するチームを分けたという事例がある。しかし、サブシステム間の結合が疎でなかったために、開発が上手く進まなかったという例もある。

(b) 仕様判断の揺らぎ

ウォーターフォール型開発では一貫した仕様書があり、判断根拠が明確である一方、非ウォーターフォール型開発では仕様に曖昧な部分が残ることがあるため、開発者が誤った理解のまま開発を進めてしまい、後で手戻りが発生する恐れがあるとされている。

(c) 曖昧な要件受入れによるテスト要因不足

ウォーターフォール型開発では、要件の抽出時にその利用方法や環境などの要件ついても併せて規定・抽出することにより、必要十分なテスト項目の洗い出しが可能である一方、非ウォーターフォール型開発においては、曖昧な要件を受け入れるため、利用方法や環境などの要件についてのテスト条件が漏れる可能性が比較的高いとされている。

５．１．２ 開発管理

非ウォーターフォール型開発において、開発管理に関わるプラクティスを抽出する。これにより、非ウォーターフォール型開発が、どのように開発管理を行っているかを把握することができる。

その上で、これらのプラクティスを適用するための前提条件を導出する。たとえば、一つの部屋でチーム全員が作業することを求めるプラクティスについては、チーム人数が大きくなると適用が困難である。

(1) 開発管理に関わるプラクティスの抽出

非ウォーターフォール型開発において、開発管理に関わるプラクティスを抽出したものを以下に示す。ここでは PMBOK の分類に従い、統合管理、スコープ管理、スケジュール管理、コスト管理、品質管理、組織・要因管理、コミュニケーション管理、リスク管理、外注管理の観点で整理を行う。

非ウォーターフォール型開発手法それぞれを見ると、重視している開発管理の分類に偏りが見られることもある。だが、非ウォーターフォール全体で見れば、必要な開発管理をほぼ網羅している。ただし、外注管理に触れたプラクティスは少ない。

(a) 統合管理に関わるプラクティスの整理

プロジェクトを全体として統合的に捉えるプラクティスとして計画関連のものがあり、また、途中段階での達成状況を確認するプラクティスがある。これらは実際に代表的なプラクティスである。統合管理に対して効果があると考えられる。

表 19 統合管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
XP-3	小規模で頻繁なリリース
XP-7	受け入れテスト
XP-11	継続的インテグレーション
Scrum-1	ゲーム前計画
Scrum-2	スプリント計画
Scrum-6	スクラムミーティング
Scrum-8	スクラムマスターのファイアウォール
Scrum-14	日次ビルド
Scrum-15	スプリントレビュー

FDD-1	ドメイン・オブジェクト・モデリング
FDD-2	feature 毎の開発
FDD-6	構成管理
FDD-7	定期ビルド
FDD-8	結果の申告と可視化
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-12	反省と改善
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
C.Clear-18	360 度の探索
C.Clear-22	情報発信
C.Clear-23	方法論の改善
C.Clear-24	反省ワークショップ
C.Clear-25	集中的な計画
C.Clear-26	デルファイ見積もり
C.Clear-28	アジャイルインタラクション設計
RUP-1	反復型開発
RUP-2	要求管理
RUP-6	変更管理
DSDM-2	MoSCoW
DSDM-3	プロトタイピング
DSDM-4	ワークショップ
DSDM-7	頻繁な引き渡しが鍵である
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-10	プロジェクトライフサイクル中に行われた変更はすべて元に戻せる
DSDM-11	要件は高いレベルで基準化される
DSDM-14	プロジェクト計画
DSDM-16	DSDM プロジェクトの計測
DSDM-17	見積もり
DSDM-19	実現可能性調査
DSDM-20	ビジネス調査

DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-2	重点要求トップ 10 を定義する
Evo-7	進化型プロジェクトマネジメント
Evo-8	進化型出荷
Evo-9	出荷したソリューションの影響を測定する
Evo-10	設計仕様を定義する
Evo-11	影響評価
Evo-15	仕様の関連
Evo-20	迅速な学習
Evo-22	問題の分割
Cell-1	シェフモードとコックモードの分離
Cell-3	資産の活用
SWT-4	要件確認会
ASD-1	ミッションを明らかにする
ASD-2	ミッションを文書化する
ASD-5	適応型サイクルの導入
ASD-7	顧客のフォーカスグループレビュー
ASD-9	プロジェクトの事後評価
EUP-1	反復的に開発する
EUP-2	要求を管理する
EUP-6	変更を管理する
EUP-9	動くソフトウェアを定期的に納品する
Uni-7	ドキュメンテーション
RAD-2	計画立案
RAD-5	展開

(b)　スコープ管理に関わるプラクティスの整理

優先順位を設定するプラクティスは、複数のアジャイル手法で共通的に利用されているものがある。また、具体的に確認する手段として、反復的な確認やプロトタイピングといったプラクティスが代表的なものとしてある。スコープ管理に対して効果があると考えられる。

表 20　スコープ管理に関わるプラクティス

プラクティス ID	プラクティス名

XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
XP-3	小規模で頻繁なリリース
XP-7	受け入れテスト
Scrum-1	ゲーム前計画
Scrum-2	スプリント計画
Scrum-7	イテレーションに追加してはならない
Scrum-15	スプリントレビュー
FDD-1	ドメイン・オブジェクト・モデリング
Lean-3	知識を作り出す（顧客からのフィードバック）
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-18	360 度の探索
C.Clear-19	早期の成功体験
C.Clear-25	集中的な計画
C.Clear-27	毎日のスタンドアップミーティング
RUP-1	反復型開発
RUP-2	要求管理
OP-2	プロトタイプ専門家の派遣
DSDM-2	MoSCoW
DSDM-3	プロトタイピング
DSDM-4	ワークショップ
DSDM-7	頻繁な引き渡しが鍵である
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-14	プロジェクト計画
DSDM-19	実現可能性調査
DSDM-20	ビジネス調査
DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-3	機能仕様を定義する
Evo-4	パフォーマンス仕様を定義する
Evo-5	明確で（可能なら）測定可能な仕様を定義する
Evo-6	Planguage で仕様を記述する
Evo-8	進化型出荷

SWT-2	タイムボックス計画
ASD-1	ミッションを明らかにする
ASD-2	ミッションを文書化する
ASD-3	ミッションが表す価値を共有する
EUP-9	動くソフトウェアを定期的に納品する
RAD-2	計画立案
RAD-5	展開

(c)　スケジュール管理に関わるプラクティスの整理

統合管理でも示したとおり、計画関連のプラクティスは充実している。また、具体的な手法として、バーンチャートといった実際に使われている事例が多いものがある。スケジュール管理に対して効果があると考えられる。

表 21　スケジュール管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
XP-3	小規模で頻繁なリリース
XP-7	受け入れテスト
XP-12	持続可能なペース
Scrum-1	ゲーム前計画
Scrum-2	スプリント計画
Scrum-3	原則 30 日（カレンダー上)のイテレーション
Scrum-4	スプリントバックロググラフの作成
Scrum-9	1 時間以内の判断
Scrum-15	スプリントレビュー
FDD-8	結果の申告と可視化
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-18	360 度の探索
C.Clear-25	集中的な計画
C.Clear-26	デルファイ見積もり
C.Clear-27	毎日のスタンドアップミーティング
C.Clear-31	バーンチャート

RUP-1	反復型開発
DSDM-1	タイムボックス
DSDM-2	MoSCoW
DSDM-4	ワークショップ
DSDM-7	頻繁な引き渡しが鍵である
DSDM-8	受け入れの主たる条件は現在のビジネスニーズを満たす機能の引き渡しである
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-14	プロジェクト計画
DSDM-16	DSDM プロジェクトの計測
DSDM-17	見積もり
Evo-8	進化型出荷
Evo-21	頻繁な出荷
SWT-1	タイムボックス
SWT-2	タイムボックス計画
SWT-3	80%ルール
EUP-9	動くソフトウェアを定期的に納品する
RAD-2	計画立案
RAD-5	展開

(d)　コスト管理に関わるプラクティスの整理

優先順位付けを行うプラクティスは、コストや期間との調整のときの意思決定の材料として活用可能である。コスト管理に対して効果があると考えられる。

表 22　コスト管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
Scrum-2	スプリント計画
Scrum-15	スプリントレビュー
C.Clear-18	360 度の探索
RUP-1	反復型開発
DSDM-14	プロジェクト計画
DSDM-16	DSDM プロジェクトの計測

DSDM-17	見積もり
DSDM-19	実現可能性調査
DSDM-20	ビジネス調査
RAD-2	計画立案

(e)　品質管理に関わるプラクティスの整理

既に前項５.１.１項で述べたとおりである。

表 23　品質管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
XP-7	受け入れテスト
XP-11	継続的インテグレーション
Scrum-6	スクラムミーティング
Scrum-10	1 日以内の障害除去
Scrum-14	日次ビルド
Scrum-15	スプリントレビュー
FDD-5	インスペクション
FDD-7	定期ビルド
FDD-8	結果の申告と可視化
Lean-2	品質を作りこむ
Lean-3	知識を作り出す（顧客からのフィードバック）
Lean-4	知識を作り出す（テストからのフィードバック）
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境
C.Clear-27	毎日のスタンドアップミーティング
RUP-5	品質の継続的検証
DSDM-4	ワークショップ
DSDM-7	頻繁な引き渡しが鍵である
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待される
DSDM-14	プロジェクト計画
DSDM-18	品質管理

DSDM-22	設計・構築モデルイテレーション
Evo-14	早期のインスペクションによる仕様品質のコントロール
SWT-5	ユーザ動作確認
SWT-6	ピックアップレビュー
ASD-4	結果に焦点を合わせる
ASD-8	ソフトウェアインスペクション
EUP-5	継続的に品質を検証する
EUP-9	動くソフトウェアを定期的に納品する

(f)　組織・要因管理に関わるプラクティスの整理

チームのメンバに対して、モラールを継続的に維持するためのプラクティスは比較的多い。これは、アジャイル的な手法が、メンバの精神・内面的な面を重要視していることが背景と考えられる。組織・要因管理に対して効果があると考えられる。

表 24　組織・要因管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-12	持続可能なペース
XP-13	チーム全体が一緒に
Scrum-5	自律的な組織化チーム
Scrum-11	ニワトリとブタ
Scrum-12	7 人のチーム
Scrum-13	共通の部屋
FDD-4	feature チーム
Lean-5	知識を作り出す（適切な決定）
Lean-9	人を尊重する
C.Clear-3	真のユーザをプロジェクトに巻き込む
C.Clear-14	個人の安全
C.Clear-15	集中
C.Clear-18	360 度の探索
C.Clear-19	早期の成功体験
C.Clear-29	プロセスの簡易版
DSDM-6	チームに引き渡しの権限が与えられなければならない
DSDM-14	プロジェクト計画
Evo-1	利害関係者を見つける

Evo-23	ユーザの権限
Evo-24	測定と褒賞
Cell-3	資産の活用
Cell-5	セルの構築と独立性
Cell-6	専門技術者の育成
Cell-7	セルのスキル割り当て
Cell-8	セル内の平等責任
EUP-7	協調的な開発
Uni-1	ユーザ一体型開発
Uni-4	非分業の原則

(g)　コミュニケーション管理に関わるプラクティスの整理

コミュニケーションもアジャイル的な手法が重要視するものである。コミュニケーション管理に対して効果があると考えられる。

表 25　コミュニケーション管理に関わるプラクティス

プラクティス ID	プラクティス名
XP-13	チーム全体が一緒に
Scrum-13	共通の部屋
C.Clear-1	短くてリッチなコミュニケーションパスがある
C.Clear-3	真のユーザをプロジェクトに巻き込む
C.Clear-13	浸透したコミュニケーション
C.Clear-16	エキスパートユーザとのコミュニケーション
C.Clear-18	360 度の探索
C.Clear-22	情報発信
DSDM-5	積極的なユーザ関与が絶対に必要である
DSDM-13	すべての利害関係者間の協力が不可欠である
DSDM-14	プロジェクト計画
Evo-18	クライアント駆動型計画
Uni-3	コミュニケーション技術

(h)　リスク管理に関わるプラクティスの整理

アジャイル的な手法のプラクティスの特徴として、状況をできるだけ見えるようにする見える化のためのものが存在する。また、頻繁なリリースなど確認を求める機会を増やすことで大きなリスクの顕在化を避けることができる。

表 26　リスク管理に関わるプラクティス

プラクティス ID	プラクティス名
Scrum-1	ゲーム前計画
Scrum-6	スクラムミーティング
Scrum-15	スプリントレビュー
FDD-8	結果の申告と可視化
C.Clear-2	1~3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-18	360 度の探索
C.Clear-25	集中的な計画
C.Clear-27	毎日のスタンドアップミーティング
DSDM-4	ワークショップ
DSDM-7	頻繁な引き渡しが鍵である
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-14	プロジェクト計画
DSDM-15	リスク管理
DSDM-16	DSDM プロジェクトの計測
DSDM-19	実現可能性調査
DSDM-20	ビジネス調査
DSDM-21	機能モデルイテレーション
EUP-9	動くソフトウェアを定期的に納品する
EUP-10	リスクを管理する
RAD-2	計画立案

(i)　外注管理に関わるプラクティスの整理

これは、本項の最初にも記載したが、外注管理に直接つながるプラクティスは少ない。

表 27　外注管理に関わるプラクティス

プラクティス ID	プラクティス名
DSDM-14	プロジェクト計画

(2) 開発管理に関わるプラクティス実施のための前提条件

開発管理に関わるプラクティスを整理した結果、以下の前提条件が必要であると考えられる。

(a) 従来の開発管理手法を用いないことを納得する

たとえば、非ウォーターフォール型開発においては、WBS を利用した管理を行わないことが多い。したがって、WBS による管理に慣れている開発者が、アジャイルにおける管理に根拠なく不安を抱く可能性がある。

(b) チーム規模が大きすぎないこと

非ウォーターフォール型開発においては、コミュニケーション管理の一つとして、チーム全員が一つの部屋で作業することを推奨するプラクティスが存在する。だがチーム規模が大きくなりすぎると、このプラクティスは実現困難になる。この場合は、他のプラクティスを利用して、コミュニケーション管理を充実させる必要がある。

(3) 開発管理に関わる課題の仮説

Web などの文献から、非ウォーターフォール型開発において開発管理にどのような課題があると人々に認識されているかについて抽出・整理を行う。

(a) ドキュメント不足による保守の困難さ

非ウォーターフォール型開発の中でもアジャイル型開発では、完全なドキュメントよりも動くソフトウェアを重視する。したがってドキュメント不足に陥り、運用担当者がソフトウェアの中身を理解できない恐れがあるとされている。

(b) 要件の変化に対するためのコスト超過への不安

非ウォーターフォール型開発では、顧客の変化する要件に迅速に対応することが求められる。顧客の要求を絶えず聞き入れることによって、途中まで開発を行っていたアーキテクチャでは対応できなくなる可能性が生じるとされている。

(c) 分散開発の困難さ

非ウォーターフォール方開発においては、チーム全員が綿密にコミュニケーションを取ることが要求される。したがって、チームが地理的に分散している場合、非ウォーターフォール型開発が行えないのではないかという不安が生じる。特に海外の拠点も含めた開発を行う場合は、時差や言葉の壁によるコミュニケーションの難しさが生じるとされている。

５．２ 非ウォーターフォール型開発の品質・信頼性及び開発管理の提言

５．１．１(1) 及び５．１．２(1) において示したとおり、非ウォーターフォール型開発におけるプラクティスを適切に利用することにより、ウォーターフォール型開発と比較して特に品質・信頼性や開発管理における課題があるということにはならない。ただし、５．１．１(2) 及び５．１．２(2) に示したとおり、非ウォーターフォール型開発におけるプラクティスを実施できる前提条件が存在するということに注意を払う必要がある。

ただし、この前提条件とは、プラクティスをそのまま利用する際における前提条件であり、何らかの工夫を施せば前提条件を取り除ける可能性の高いものである。本節では、実際に工夫を行っている事例を整理する。

また、５．１．１(3) 及び５．１．２(3) に示したように、非ウォーターフォール型開発においては、品質・信頼性や開発管理において課題があるとされる項目が存在する。実際に非ウォーターフォール型開発を行った事例から、これらの課題が存在するかどうかの確認も行う。

(1) 事例における実践例

まず、事例調査において、品質・信頼性のためにどのようなことを実践したのかを抽出し、表 28 に示す。また、開発管理に関しては抽出した結果を表 29 に示す。また、特に、既存のプラクティスをそのまま適用するのではなく、工夫が見られるものの例は５．２(2) に示す。

表 28　事例調査における品質・信頼性確保の実態

分類	プラクティス例
テスト	テスト駆動開発
	単体テストの自動化
	継続的統合
	カバレッジを満たすものをあらかじめ作成
	自動化されたテストを毎日行う
	ステージング環境での重点的なテスト実施
	受け入れテスト
	受け入れテストケースについてオーナがレビュー
	受け入れテストは顧客又は顧客プロキシが実施
	コーディング段階から実データを使用してテスト
	外部ペネトレーションテスト
	機能テストや統合テスト

	セキュリティや性能（負荷、レスポンス等）のチェックは、専門チームが担当
	テスト専門のチームを作る
レビュー	ベテランが若手の設計をレビュー
	ベテラン開発者によるソースコードのレビュー
	インフォーマルな設計セッションを開く
	ソースコードレビュー（サンプリング）を実施する
	新しいメンバが対応する場合は、特に外部レビューを実施する
	相互レビューの推奨
リファクタリング	積極的なリファクタリングを行う
頻繁なリリース	リリースを繰り返し、品質を要求に近づける

表 29　事例調査における開発管理の実態

分類	実践例
統合管理	仕様変更の際は顧客が起票
	イテレーション単位で管理
	絵コンテ、機能大枠、開発スケジュール大枠といったディスカッション用資料程度
	合宿を実施し、プロダクトビジョンボックスを作成
	進行中は週次の計画ゲームとふりかえりによって、プロジェクトの進むべき方向を調整
	イテレーション単位で「ソフトウェアかんばん」を使った計画ゲームを実施
	開発開始時にプロセス、作成するドキュメント、品質方針、リリースタイミング等を発注元と打合せしながら策定
	イテレーションが終わる度に、そのイテレーションでの成果を統合。開発が完了しなかったものは、次イテレーションに持ち越す
	前半にユースケースで重要な幹の部分を合意し、画面や帳票といった枝葉は後半で詰めるような計画とする（負荷配分の調整）
	Team Concert を用いることにより、計画文書（作業項目）と、各人がやるべきことの関係を可視化する
	概算 FP を算出し、過去の実績と照らし合わせて必要な工数、期間を算出
スコープ管理	プロジェクト開始から完結に至る全イテレーションと全バックロ

	グを管理スコープとする
	エクセルでアクティビティを洗い出して工程を管理
	個人ベースではタスク一覧を作成
	ストーリーカードとイテレーションのベロシティをもとに、リリース計画とイテレーション計画を実施
	ベロシティのトラッキングにはバーンダウンチャートを用いる
	リリースまでのマイルストーンを作成し、週次で行われる計画ゲーム時にマイルストーンの見直しを行う
	業務フローを紙とポストイットによるシミュレーションの繰り返しで明らかにし、成果物の範囲を決定
	成果物の機能(ユーザ視点)をストーリかんばんとし、作業範囲の管理をバーンダウンチャートで行う
	開発期間全体およびイテレーション毎にスコープを設定。原則として次イテレーションへの持ち越しを極力回避
	イテレーション開始時の計画ゲームにて、当イテレーションでの開発対象を決定
	初期にスコープを確定、その後はイテレーション単位の実装範囲を計画して遂行
	案件管理表にて優先度を決めスコープを決定する
	「ジャストインタイム」のコンセプトで、顧客によって優先順位付けされた機能を順次実装（YAGNI : You Aren't Going To Need It)
	間違いの修正は優先度をあげて対応するが、大きな要件の追加の際にはユーザと相談
	要件定義（PH1）終了段階で、必要となる機能を画面ごとにあらかた確定
スケジュール管理	プランニングポーカーの利用
	実際にかかった時間をフィードバックし、見積もりの精度を上げる
	リリース期間が短い場合（管理しなくても問題ない場合）は、スケジュール管理を行わない
	締切りが近づくに従って生産性が上がる締切り効果を生かす
	WBS の利用禁止（人間は機械ではないため、WBS を作成して線表を引いてもコンスタントに力を出せない。最後だけ生産性が高

	いことを無視した線表では無意味)
	イテレーション単位とバックログを基にしてスケジュール管理を行う
	イテレーションではタイムボックスが固定することを絶対として進捗を管理
	バーンダウンチャートを利用し、作業進捗の管理ではなく、バックログの残数に対応するためのチーム作業バッファを管理
	期間内開発というコントロールのみを実施
	ストーリーカードを元にタスクカードを作成し、日々の進捗をスタンドアップミーティングとバーンダウンチャートによって追跡
	週次で開発マネージャを含めて計画ゲームを行い、次回のリリース計画を作成した。タスクカードは「かんばん」を用いて運用
	イテレーション単位とソフトウェアかんばん(ストーリかんばん、タスクかんばん）を基にしてスケジュール管理
	バーンダウンチャートを利用し、作業進捗の管理ではなく、タスクかんばんの残数に対応する管理を行う
	イテレーション毎にスプリントバックログを元にスケジュール管理を実施
	発注元開発との統合スケジュールについては入念に意識あわせを実施してスケジュール調整を実施
	XPlanner を使用して、タスクごとの進捗を管理
	ポストイットをタスクカードとして ToDo ボードに張り出した
	計画ゲームにてタスクを決定、所要時間見積り、実績の測定、バーンダウンチャートで進捗を見える化
	独自のスケジュール管理表を使って管理
	進捗管理（実績管理）は行うが、見積もりは行わない。この機能をいつまでに作る、ということは決定しない
	WBS、課題管理表、日次の To Do リストを作成
	毎朝、開発状況の確認ミーティングを実施
	XP のプラクティスである「ミラー」により目で見える管理を体現。ストーリーカード、タスクカード、バーンダウンチャート（バックログ数のグラフ）を壁に貼り、進捗状態を開発者がリアルタイムに見られるようする
	毎日、立ったままでの朝会から 1 日をスタートさせる

	各自は、自分のタスクを申告してから業務に取りかかる
	マイルストーンファースト（最初にどれくらいの位置にマイルストーンを置くかを決めてから始める）
	アダプティブ・プランニング（計画駆動の目標も状況に応じて変えていく）
	タイムボックスによる時間管理
	日ごとの残作業時間数を PL が管理
	朝会、振り返りの実施
コスト管理	イテレーション単位で原価管理を行い、月単位で集計
	テストケースを複数回実施
	イテレーション単位のベロシティのトラッキング
	イテレーション毎に、チームが費やせる時間と、タスクの総投入時間の差を見て、投入するタスクの調整を行う。タスクの投入時間は、チーム全員で見積もる
	プロジェクト開始時に全イテレーションにおける開発項目および開発規模の大まかな見積りを実施し、各イテレーションの実行可能性をチェック。以後、各イテレーションの開始時に見積りを精査
	プロジェクトメンバ数は固定なので、コストは期間によってのみ決まる
	当初の計画よりもスケジュールの遅延が発生したが、開発プロジェクトマネージャとシステムオーナで協議し、追加予算を獲得
	タスクごとの理想見積りを行い、実績から求めた係数をかけて工数を算定
	開発生産性強化グループより週次で提示
品質管理	障害管理
	変更管理
	運用上の不具合を全て ITS に登録
	セキュリティや性能（負荷、レスポンス等）のチェックは、専門チームが担当
組織・要員管理	ソフトウェア・セル単位でのコントロール
	数人のため座席（寝床も）隣接
	ペアプログラミング
	後発で増員されたメンバにはペアプロを行い知識の共有を行う

	各作業者の作業状況を日次(朝会等)で把握。遅延の傾向が見える場合には、サブリーダ主導で作業調整を実施(作業者の変更等)
	イテレーション会議において、開発者自身の見積もりによってタスクのコミットを実施。負荷(稼動)の平準化を図る
	ドメインの知識や従来の開発手法をよく知るプロパーと、新プラットフォームのアーキテクチャをよく知るメンバが組んで、ペアプログラミングを実践
	2 週間単位の振り返りを行い改善
	作業者が複数の作業スキルを備える「多能工化」と作業者ごとの作業時間をならす「平準化」を原則
コミュニケーション管理	ソフトウェア・セルを採用。セル屋台をオン・オフライン上に作成
	１５分間の朝会とイテレーション毎に回顧（振り返り）を実施
	開発チームは同じ場所に常駐
	オンサイト顧客
	当初は客先に常駐することにより全員でチームだという環境を作り、チームの方向性が一致してからはリモートで Skype を用いてコミュニケーションを図る
	週次で定例 MTG を開催し、週に 1 度以上は全員が顔を合わせる機会を設ける
	朝会と毎日ふりかえり、イテレーション毎にふりかえりを実施することでもコミュニケーションを図る
	ニコニコカレンダーによって、メンバの体調やメンタル面の異常に気づくことができる
	全作業者を同じフロアの 1 箇所に集めて作業実施
	毎朝の朝会実施と、イテレーション毎の振り返り(KPT)を実施
	開発プロジェクトマネージャを含む開発者全員は、プロジェクトルームに常駐
	開発者はプロジェクトルームに集結
	毎朝、開発プロジェクトマネージャが開発者一人ずつに対して、ミーティングを行う
	開発プロジェクトマネージャと開発者は同じ部屋で作業を行う
	顧客との打ち合わせは 2 週間に一度行う
	定期的（1-2 週間に一度）に進捗会議を実施する

	ペアを毎日変える
	文書よりも対話での情報共有を重視し、ユーザは開発者と離れずにプロジェクトルームに詰めて一体となった密接なチームを形成
	各自の席は壁沿いに壁を向くように配置し、プロジェクトルームの中央にミーティング用の机を置く
	セル内およびセル間におけるコミュニケーションを目視できるカードにしてソフトウェア看板に貼り付ける
リスク管理	バーンダウンや「かんばん」を利用することでリスクを可視化
	チームワークやペアワークを基本とすることで、個人に依存するリスクを軽減
	日次のスタンドアップミーティング、週次のふりかえりとイテレーション計画、1ヶ月3ヶ月単位のリリース計画でリスク項目について議論
	週次でふりかえりをおこない、各自が抱えている不安やリスク等を洗い出し、対応計画を立てる
	２週間という短期間のサイクルを繰り返し、顧客プロキシにも計画ゲームに参画することで、変更リスクを早期に取り除く
	バーンダウンやかんばんといった貼り物による見える化で、課題の解決につなげる
	開発初期段階でモックアップ等の動くプログラムをベースに仕様を策定することで、後工程での仕様変更発生のリスクを低減
	スプリント途中での飛び込みでの要件追加等に対する対応策として、「バッファ」を見積もってプロダクトバックログに追加
	リスク駆動の方針に則り、難しい要求、技術的ブレイクスルーが必要なものを先に着手
	リスク管理票を作成。定期的な見直しと対策の実施
外注管理	契約単位で継続するかどうかを判断

(2) 事例における工夫例

特に、既存のプラクティスをそのまま適用するのではなく、工夫が見られるものの例を以下に示す。

表 30　品質・信頼性における工夫例

分類	実践例
テスト	ステージング環境にリリース対象をデプロイ後 1 週間かけてテストを実施し、リリース
	デイリーテストにしてカバー率を重視
	NUnit を利用したテストの自動化
	コーディング段階から実データを使用してテスト
リファクタリング	TDD による大量のテストケースに担保された積極的なリファクタリング
	テストツールによって抽出された問題の多いと考えられるソースコードを、月 1 回のペースでメンバ全員でリファクタリングすることにより、理想的なソースについての共通認識を醸成させる
レビュー	ドメインの知識や従来の開発手法をよく知るプロパーと、新プラットフォームのアーキテクチャをよく知るメンバが組んで、ペアプログラミングを実践

表 31　開発管理における工夫例

分類	実践例
統合管理	セルは、セキュリティが厳格に管理されている仮想環境によって支援するようにし、アクセスを許諾されたものは、セルの観察によって、イテレーションを統合管理
	合宿を実施し、プロダクトビジョンボックスを作成する
	プロジェクトの開始時にある程度プロジェクト計画はしっかりと立案する
	安心して進められるだけの見通しと、タスクへ落とせる概要像・コンセプトなどヒント作りと、最終形イメージの具体的な合意形成ルールの作業連鎖
	熱あるインタビュー(質問でなく誘導)で本来要求を効率的に導出する
	赤書レビュー
	記録できるものは全て記録する（たとえば、ブラウザの画面遷移を全て記録しておく。構成管理のメッセージと突き合わせることで、精度高く、顧客が主張しようとしていることがわかる。不具合の対処の 8－9 割は、問題の特定に費やされることが普通である。全ての記録を

	取るのはデータ量が多くなるが、それでもメリットが大きい）
	Team Concert のソースコード管理システムを利用する（サーバのほかに「リポジトリワークスペース」という“サーバ上の個人用お砂箱”を作っている。見た目はクライアントのワークスペースであるが、リポジトリとクライアントの間にレイヤを追加することで、離れた人に対して実験的なコードを共有領域を崩すことなく共有することが可能となる。後はチャットなどで差分を添付して相手に送れば、マージできる）
	サイクル 1 で正常系を開発し、サイクル 2 で異常系を開発
	要件定義（PH1）終了段階での要件の固まり具合をみて、開発を実施するかどうか判断
	複数チームを作成し、階層的なスクラムを組む
	複数チームで開発するので、コンポーネントを疎結合に設計する（アーキテクトが設計）
スコープ管理	基本的に全体計画立案時に海外営業拠点と合意したスコープに対してドロップすることは容認されないため、極力、スプリント毎に設定したスコープを守るようにする
	顧客が具体的にイメージできる、そのとき開発チームが提供できる最良なものを使った。ほとんどは実際動くプロトタイプを使う
	「テーマ」「フィーチャ」「ストーリ」でスコープを定める（ストーリは月間の集計が見えるとなどの動かすイメージ。フィーチャはそれが営業支援なのかとかのカテゴリ。一年後のリリースのテーマは、スケーラビリティを上げるためにはどうすればいいかを考えると、管理系でこういうオペレーションができればいいなどを決める。テーマはのろしであり、フィーチャやストーリはそれに基づいて細かくしたものである。テーマは初期に決定し、変更されないレベルで決定する）
	ユーザからの要求変更は、それがなければ業務が実施できないものに限定（要求が実現される範囲には限界があることを理解してもらう）
スケジュール管理	バーンダウンチャートに上限線や目標線を引く
	独自のスケジュール管理表を使って管理
	イテレーション開発すると見通しの中で最終成果が仕上がるため、顧客と開発チーム間で、スケジュール管理というキーワード自体が空気のように意識しなくなる
	締切りが近づくに従って生産性が上がる締切り効果を生かす

	2 種類（タスクベースと時間ベース）のバーンダウンチャート進捗状況把握
	モデル定義や FP 法での要件定義の段階において、どういうチームビルディングになって、というシミュレーションを実施する（Web サイトは似ている機能が多いため記録として取っている。たとえば会員機能であれば 40 画面必要だと、作業工程はこの程度だと記録が残る）
	20%の規模・期間のバッファとして設ける
コスト管理	開発開始時に全スプリントのおおまかな計画を先に立案しておく
	開発生産性強化グループより週次で提示
	工程及び反復ごとに出来高（進捗）を把握し、消費コスト（社員、社外）を捉えて予実を管理
	当初の計画よりもスケジュールの遅延が発生した際には、開発プロジェクトマネージャとシステムオーナで協議し、追加予算を獲得
品質管理	運用上の不具合等は全て ITS に登録することで管理を行い、開発マネージャの修正確認後 Fix とするような運用を行う
	レビュー回数、指摘数の管理及び、テスト項目数、不具合数の管理
	イテレーション開発で統合できる範囲の結合テストまで実施する（全体的に実際業務に合わせたデータで統合テストするのは最後に合わせて実施して仕上げる）
	1 つのクラスを複数の開発者によって開発させることにより、実装とレビューが繰り返されるようにルール化
組織・要員管理	朝会時に、チーム全体のタスク管理表を見ながら報告を実施
	ハイスキルなアーキテクトの早期アサインと、若手開発者のリード
	方式面で弱い若手が多く、真似る形でやらせて方向変換が必要なとき追加で教える。技能に劣る人員をテストに回して、世話のかからない人員のみ開発へ残す
	イテレーション会議において、開発者自身の見積もりによってタスクのコミットを実施。負荷(稼動)の平準化を図る
	ドメインの知識や従来の開発手法をよく知るプロパーと、新プラットフォームのアーキテクチャをよく知るメンバとが組んで、ペアプログラミングを実践
	ソフトウェア・セル単位でのコントロール
	遅延の傾向が見える場合には、サブリーダ主導で作業調整を実施(作

	業者の変更等)
	作業者が複数の作業スキルを備える「多能工化」と作業者ごとの作業時間をならす「平準化」を原則
	小さなチームへの権限の委任（より権限が強いステークホルダから、良く分かっていないという認識を持たれた場合、実質的には委任という状態にならず他のステークホルダの介入を許してしまうため、チームメンバは業務についてよく理解する必要がある）
	アジャイル型の開発に馴染めない者を交代させる
コミュニケーション管理	当初は客先に常駐することにより全員でチームだという環境を作り、チームの方向性が一致してからはリモートで Skype を用いてコミュニケーションを図る
	朝会時に、チーム全体のタスク管理表を見ながら報告を実施
	顧客が開発に安心して任せられるよう、フェーズ毎にメリハリつけて頻繁かつコミュニケーション距離を近づける工夫
	ソフトウェア・セルを採用。セル屋台をオン・オフライン上に作成
	分散開発なので、電話会議・チャットを利用。特に海外とはチャットが言語の壁を下げるので有効
	バックログはソフトウェア・セルのものとし、バックログの解決を個人に依存しないようにする
リスク管理	バッファとして見積もっている時間もプロダクトバックログに含めて管理する
	プロジェクトリスクに関しては PM、顧客で洗い出し、重要リスクのみステアリングコミッティで監視
	リーダ負荷を分散するためチームが集中力を確保できるだけの指示をリスクと対処指示の形で示す
外注管理	アーキテクチャを構築するハイスキルな集団と、開発ガイドに添ってアプリ開発する集団に外注要員のプロスペクトを階層化

(3) 信頼性をはじめとする品質確保及び開発管理に関する提言

非ウォーターフォール型の開発において、信頼性向上、品質確保及び開発管理に関するプラクティスは存在しており、非ウォーターフォール型の開発におけるこれらの問題の所在が手法にあるとは考えられない。品質・信頼性に関わる課題の仮説を５．１．１(3) 及び５．１．２(3) において示したが、これらの課題について対応できる可能

性のあるプラクティスを５．１．１(1) 及び５．１．２(1) から、また、取り組みの事例を５．２(1) 及び５．２(2) から見ることができる。以下において、課題とその対応策を述べる。課題についての詳細は、５．１．１(3) 及び５．１．２(3) を参照のこと。

(a) チーム規模の限界

チーム規模の増加に対しては、表 31 にあるように、複数のチームを構成し、階層的なスクラムチームを組むことによって対応した事例が見られる。二段目のスクラムチームについては、一段目の各スクラムチームの代表者によって構成される。

一方、チームを分割するため、サブシステム間の結合が疎でなかったために、開発が上手く進まなかったという課題も存在する。この課題に対しては、表 31 にあるように、サブシステムを疎結合に設計するという実践事例が見られる。また、サブシステムを疎結合に設計するためのプラクティスとして、「Cell-2：DSM を用いたアーキテクチャ変換」というプラクティスが存在する。DSM は Dependency Structure Matrix の略語であり、設計依存性を可視化する手法である。

(b) 仕様判断の揺らぎ

非ウォーターフォール型開発においては、開発者は少しでも仕様に不明な点が存在する場合は、すぐに顧客に内容を確認することが求められる。そのため、顧客とのコミュニケーションパスの必要性を訴えるプラクティスが数多く存在する（「XP-13：チーム全体が一緒に」や「C.Clear-1：短くてリッチなコミュニケーションパスがある」など)。また、顧客との綿密なコミュニケーションを取れない開発者をプロジェクトから外した事例も見られる。したがって、仕様書のみを見て判断するのではなく、すぐに顧客に聞ける能力や環境が重要となる。

(c) 曖昧な要件受入れによるテスト要因不足

当初は要件が曖昧であったためにテストの網羅性を確保できなかったとしても、要件が確定した時点で、その要件に関わるテストを作成することができる。非ウォーターフォール型開発においては、多くのプラクティスが継続的インテグレーションによる自動テストを要求しており（「XP-11：継続的インテグレーション」や「Scrum-14: 日時ビルド」、「C.Clear-17：自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境」)、テスト項目が増えた際においても、ソフトウェア全体に対して迅速にテストを実施することが可能である。また、リリース直前に再度、網羅的なテストを実施する事例も見られる。

(d)　ドキュメント不足による保守の困難さ

非ウォーターフォール型開発では、完全なドキュメントよりも動くソフトウェアを重視しているが、ドキュメントを作成してはいけないというわけではない。必要なドキュメントであれば必ず作成する必要がある。また、保守担当者にも開発に携わらせることにより、ソフトウェアの中身をよく理解させる取り組みの事例も見られる。

また、ソフトウェアは「XP-5：シンプルデザイン」や「Lean-1：無駄をなくす」のプラクティスが示すように、シンプルで理解しやすい設計の下で開発されることが求められている。したがって、ソフトウェアの中身を十分知らない技術者がソースコードを見た際においても、比較的容易に理解できるようになっているはずである。

(e)　要件の変化に対するためのコスト超過への不安

非ウォーターフォール型開発においては顧客の要求の変化に迅速に対応することが利点として挙げられるが、全ての要求の変化を受け入れるわけではない。たとえば、「DSDM-11：要件は高いレベルで基準化されている」というプラクティスは、細かい部分を後で決めることを許容しながらも、システムの目的とスコープを高いレベルにおいては固定して同意することを求めている。また、複数の非ウォーターフォール型開発手法が採用している「タイムボックス」というプラクティスでは、新しいタスクを受け入れる場合は、優先度が低いほかのタスクを排除することを要求する。

事例においても、要件の抽象度が高い順に「テーマ」「フィーチャ」「ストーリ」でスコープを定め、「テーマ」については基本的には変更しないことを求めている事例が見られた。また、ユーザからの要求変更は、それがなければ業務が実施できないものに限定するという事例もある。

このように、非ウォーターフォール型開発はユーザの要求には可能な限り対応していくものであるが、新しい要件を受け入れた場合は、優先度の低い他の要件を排除することにより、コストやスケジュールが予定を超過することを防ぐことが必要である。

だが課題として、ウォーターフォール型開発に慣れた顧客に対しては、このような取り組みが理解されない恐れがある。プロジェクト開始前の段階において、要求の変化は受け入れるが、予算やスケジュールに変更が無い場合は、全ての要求を受け入れるわけではないということを理解してもらう必要がある。

(f) 分散開発の困難さ

分散開発を行うと、チームメンバ同士が顔を合わせる機会が減少することにより、円滑なコミュニケーションの実施が難しくなる。だが、表 31 にあるように、電話会議・チャットを利用することにより、コミュニケーションを図った事例が見られる。また、特に海外とはチャットが言語の壁を下げるので有効であることが事例からわかった。分散開発を行う場合は、ウォーターフォール型開発・非ウォーターフォール型開発に関わらず開発は難しくなるが、特に非ウォーターフォール型開発においては、このような工夫が必要だと思われる。

上記からわかるとおり、非ウォーターフォール型開発に対する課題の仮説それぞれについて、非ウォーターフォール型開発のプラクティス・及び開発事例から、その対応策を提示することができる。

一方、信頼性向上、品質確保及び開発管理につながるプラクティスにおいて、５．１．１(2) 及び５．１．２(2) に示したとおり、その効果を十分に発揮する、または、失敗しないための条件を満たす必要があることも事実である。このような場合は、一部のプラクティスをそのまま利用することは困難である。したがって状況に合ったプラクティスを取捨選択することが重要である。

６ 非ウォーターフォール型開発への顧客参画に関する現状調査とあり方への提言

非ウォーターフォール型開発においては、ウォーターフォール型開発以上に顧客とベンダとの協議、顧客企業が積極的に参画する環境作りによって非ウォーターフォール型開発の実効性を高めるといわれている。

６．１ 非ウォーターフォール型開発における顧客参画について

(1) 顧客参画に関わるプラクティスの抽出

非ウォーターフォール型開発において顧客が参加するタスクやプラクティスを抽出し、以下に示す。

これらの表からわかるとおり、各開発手法において、顧客が関わるプラクティスの数は多い。このことからも、非ウォーターフォール型開発手法は、顧客との関係を重視しているということがわかる。

(a) 計画時に行われるプラクティスの整理

方針として明確にユーザの関与を求めるものをはじめ、プロジェクトの全体を把握し、顧客と共有するための計画関連のプラクティスがある。また、これらは顧客の優先順位等を把握することで、必要なものから実現するなど実際的でかつ効率的なプラクティスである。計画時に顧客の参画を確保するために効果があると考えられる。

表 32 計画時に行われるプラクティス

プラクティス ID	プラクティス名
XP-1	リリース計画ゲーム
XP-2	イテレーション計画ゲーム
Scrum-1	ゲーム前計画
Scrum-2	スプリント計画
Scrum-13	共通の部屋
RUP-2	要求管理
DSDM-5	積極的なユーザ関与が絶対に必要である
DSDM-19	実現可能性調査
Evo-18	クライアント駆動型計画

(b)　開発時に行われるプラクティスの整理

開発時では、顧客の関与を確保するために具体的な方法を示したプラクティスがある。場所をはじめ、ステークホルダの見極め、コミュニケーションから、開発でユーザの関与を前提とした開発方法などのプラクティスが示されている。開発時に、顧客の参画を確保するために効果があると考えられる。

表 33　開発時に行われるプラクティス

プラクティス ID	プラクティス名
XP-13	チーム全体が一緒に
Scrum-13	共通の部屋
C.Clear-1	短くてリッチなコミュニケーションパスがある
C.Clear-3	真のユーザをプロジェクトに巻き込む
C.Clear-16	エキスパートユーザとのコミュニケーション
DSDM-5	積極的なユーザ関与が絶対に必要である
DSDM-13	すべての利害関係者間の協力が不可欠である
Evo-18	クライアント駆動型計画
Evo-23	ユーザの権限
SWT-3	80%ルール
EUP-7	協調的な開発
Uni-1	ユーザ一体型開発

(c)　顧客からのフィードバックを求めるプラクティスの整理

実際に動くものを見せることを基本的な方針として、具体的な方法を中心に、様々なプラクティスが示されている。プロトタイプをはじめ、実際に動作の確認を反復的に行うものが多い。また、基本方針としてできるだけ早く構築しできるだけ早く確認を求めることが伺える。顧客からのフィードバックを確実に得て、状況や内容を共有することに効果があると考えられる。

表 34　顧客からのフィードバックを求めるプラクティス

プラクティス ID	プラクティス名
XP-3	小規模で頻繁なリリース
Scrum-3	原則 30 日（カレンダー上)のイテレーション
Scrum-15	スプリントレビュー
Lean-3	知識を作り出す（顧客からのフィードバック）

Lean-8	速く提供する
C.Clear-2	1～3 か月以内に出荷する
C.Clear-11	頻繁なリリース
C.Clear-28	アジャイルインタラクション設計
OP-1	進化型プロトタイプの作成
OP-2	プロトタイプ専門家の派遣
DSDM-3	プロトタイピング
DSDM-5	積極的なユーザ関与が絶対に必要である
DSDM-7	頻繁な引き渡しが鍵である
DSDM-8	受け入れの主たる条件は現在のビジネスニーズを満たす機能の引き渡しである
DSDM-9	反復的で漸増的な引き渡しが不可欠である
DSDM-21	機能モデルイテレーション
DSDM-22	設計・構築モデルイテレーション
Evo-7	進化型プロジェクトマネジメント
Evo-8	進化型出荷
Evo-9	出荷したソリューションの影響を測定する
Evo-21	頻繁な出荷
SWT-4	要件確認会
SWT-5	ユーザ動作確認
ASD-7	顧客のフォーカスグループレビュー
ASD-9	プロジェクトの事後評価
EUP-1	反復的に開発する
EUP-9	動くソフトウェアを定期的に納品する

(2) 顧客参画に関わるプラクティス実施のための前提条件

顧客参画に関わるプラクティスを整理した結果、以下の前提条件が必要であると考えられる。

(a) 顧客の高いスキルと権限

優先順位をミーティング中にその場で判断できる必要がある。いったん持ち帰っての判断などは、非ウォーターフォール型のメリットであるリズムとスピードを損なう結果につながってしまう。

(b) 顧客のプロジェクトへの専任度合いの高さ

顧客からの継続的なフィードバックを求めるプラクティスが多い。顧客がすぐに反応できないと開発が止まってしまう。

(c)　顧客の知識

タイミングの良い判断のためには、業務知識が必要である。

(d)　開発者の高いスキル

機能を実現するまでに必要な期間やコストをミーティング中にその場で判断できるためには、開発における幅広い知識が必要となる。

６．２　文献・Web およびヒアリング調査による顧客参画の実態把握

(1) 顧客の関わり度合い

事例調査から把握される、プロジェクトへの顧客の関わり度合いの実態例を次に示す。この表からわかるように、顧客は週に 1 から 2 回のコミットが求められていることが多い。また、打ち合わせに毎日出席することを求められている事例もあった（ただし、担当者の負担が大きく他の業務への影響が出ることから後に週に 1 回に変更されている）。

表 35　プロジェクトへの顧客の関わり度合い

事例
顧客側に 2 週間に一回、必ず受け入れ検収を実施できる体制が必要
2 週間に 1ー3 度、オーナが来社しオンサイト顧客を実施
週次で開発マネージャを含めて計画ゲームを行い、次回のリリース計画を作成した
週に 1 回、プロジェクトの進捗状況を開発プロジェクトマネージャが発注元の開発マネージャに報告実施。また発注元の開発マネージャ同席の元、適宜、プロダクトマネージャと電話会議にて打合せ実施
発注者と開発者は毎週 2 回の打合せで週次イテレーション開発
1 回／週の計画ゲーム・ふりかえりに参画する
企画担当者は毎日要件確認会に出席する（負担が大きすぎたため、後に週に 1 回に変更。少なくとも 1 週間に半日はコミットできることを要求）

(2) 顧客の役割・権限・知識・スキル・理解

事例調査において、顧客にどのような役割、権限、知識、スキル、理解が求められているかについて抽出したものを、表 36 に示す。

基本的にリリースされた結果に対して確認（検収）を行うこととともに、速やかに決定するための権限及び決断力を有しているとの例が示されている。

知識に関して、非ウォーターフォール型の知識がある場合、ユーザ側から非ウォーターフォール的な実践に関する提案があったりする場合もあるが、基本的には必ずしも非ウォーターフォールについて知識が必要であるとは限らないと思われる。一方、業務知識は必要である。これがないと決断が難しい。

必要なスキルとしては、コミュニケーション能力、すなわち、考えていることの表現及び関係者の意見に対する理解能力が重要とされている。これは、確認・決断・実装の短期サイクルのリズム・スピードを維持するために必要であると考えられる。

表 36　事例調査から抽出した顧客の役割・権限・知識・スキル・理解

分類	実践例
役割	リリースされたプロトタイプを使用し、フィードバックする
	2 週間に一度、必ず受け入れ検収を実施する
権限	ビジネス執行企画者は、ビジネス上の優先順位を決定でき、速やかに決断する力を有する
	機能に関する決定権を持っている必要がある
知識	非ウォーターフォールについての知識は豊富。実践は初めて
	技術的な知識はほぼゼロ
	業務知識が十分にあり、要件を決めることができる
スキル	リスペクトがあるという前提があれば、顧客のスキルは高いものは要求されない
	情報システム責任者は、関連システムを含む全体を見通せ、各部門との調整に長けている
	コミュニケーションスキルが高く、ビジネス要件にも踏み込め、開発の経験もある
	コミュニケーションスキルや技術スキルに優れている
	顧客参加はなし。顧客プロキシはシステム開発・保守およびプロジェクトマネジメントの経験が豊富
理解	生データを提出すること
	開発者に対しリスペクトすること
	非ウォーターフォール型開発に理解を示すこと
	80%ルールを理解し、随時、意思決定できること

６．３ 非ウォーターフォール型開発における顧客参画への提言

６.１において、顧客参画に関するプラクティスを整理し、６.２において、顧客参画の実態を挙げた。これらを踏まえ、以下の提言を行う。

・　ウォーターフォール型開発と比較して、非ウォーターフォール型の開発においては、ユーザの役割として、頻繁にリリースされるソフトウェアを使用し、フィードバックを行うことが基本的に重要であるがわかる。また、機能に関する決定権を持っている必要があり、技術的知識や技術的スキルは求められないが、業務知識について熟知している必要があると考えられる。

・　顧客参画が非ウォーターフォール型開発における基本条件であることは明らかであるが、一方、顧客の負担（作業・責任）はかなり大きなものとなり、顧客側に常に理想的な体制がとれるとは限らない。この辺りの顧客側の理解及び理解に基づき体制を整えることが成功の鍵である。だが顧客に参画してもらえるような体制を整えることができない場合に、「顧客プロキシ」を立てるという事例が複数見られた。顧客プロキシとは、顧客の業務をよく知る開発者を顧客の代わりに見立てて、プロジェクトに参加してもらうことである。顧客が担当するプラクティスを顧客プロキシが代理人となって実施することになる。

・　非ウォーターフォール型開発手法においては、開発スピードを向上させるために顧客に対して迅速な決断を求める事例が見られる。そのため、顧客は迅速な判断ができるような高いスキルが求められることになり、顧客が非ウォーターフォール型開発を採用する障壁となる可能性がある。また、顧客が判断を行うために長い時間を要する場合、非ウォーターフォール型開発がうまく進まない恐れがある。逆の場合も同様であり、非ウォーターフォール型開発においては、開発者に対しても迅速な判断が求められる。そのため開発者が迅速な判断を行えない場合は開発が滞る恐れがある。これらの課題に対応するために、仕様の完全さの尺度として 80%ルールを利用する事例がある。これは、特にウォーターフォール型の開発に慣れている場合に、仕様を完全に決めきれないために次のステップに進めないなど、非ウォーターフォール的のメリットを生かすために必要な活動・判断が取れないといった場合の方針として活用されているものである。

　ただし、80%ルールを実践する場合は、その内容を顧客にしっかりと理解してもらう必要がある。たとえば、一度開発を行うと決めた内容について、難易度が高いことが後に判明したとする。この場合に、顧客側が開発者側に対して、以前決めた通りの開発を行うことを強く要求すると、これ以降の判断の場において、開発者側が時間をかけて慎重に判断をせざるを得なくなってしまう。このような状況に陥ると、早い開発スピードを実現することが困難になる。

・　顧客が自ら運用・改修ができるように、顧客を教育するという事例も見られた。将来のコスト削減等のために顧客が教育されることを望む場合は、顧客が積極的にプロジェクトに関わり、システムについて深く理解することになる。したがって、

顧客のプロジェクト参画の度合いが高くなるため、プロジェクトを容易に進めやすくなる。このようなことを実践するためには、開発の熟練者ではない顧客にとっても理解が容易なシステムアーキテクチャであることが要求される。

- 5.2(3) (e) に示したように、非ウォーターフォール型開発においては顧客の要求の変化に迅速に対応することが利点として挙げられるが、開発者側は顧客の全ての要求を受け入れるわけではない。要件を一つ追加する、あるいは変化させる場合は、その作業量に相当する優先度の低い要件を削除する必要があることを、顧客は認識する必要がある。また、要件を削除しないのであれば、予算の追加及びスケジュールの見直しが必要となることを認識する必要がある。

- 非ウォーターフォール型開発においては、表 34 に示すとおり、顧客からのフィードバックを要求するプラクティスが多く存在する。したがって、顧客がフィードバックを行いやすい環境作りをすることが必要となる。顧客に対してドキュメントのみを用いて説明するのではなく、実際に動くソフトウェアを見せて説明する（「EUP-9：動くソフトウェアを定期的に納品」や「Lean-3：知識を作り出す（顧客からのフィードバック)）などの工夫が求められる。

7 契約方式上の課題整理

7．1 非ウォーターフォール型開発における契約方式上の課題整理

ソフトウェア開発の契約では、請負契約、準委任契約、システム・エンジニアリング・サービス契約の 3 種類の契約が一般的に行われている。

表 37 ソフトウェア開発における現状の契約形態

契約の種類	内容
請負契約	請負契約とは、ベンダ企業側が成果物の完成を請負い、ユーザ企業側が成果物に対する報酬の支払いを約束する契約形態である。
準委任契約	準委任契約とは、業務を委託する契約であり、ベンダ企業側の責任は、業務を実施することにあり、成果物に対する完成責任を負わない。
システム・エンジニアリング・サービス契約	業務委託契約の一形態であるが、エンジニアの能力そのものを契約の対象とし、ユーザ企業からベンダ企業への支払いは単価×時間で行われる。

また、ソフトウェア開発業務に関する契約とは別に、ソフトウェア開発要員の利用に関わる契約として労働者派遣契約がある。請負契約および準委任契約と労働者派遣契約の比較を表 38 に示す。ここに示したように、労働者派遣契約は要員派遣を目的としたものであるため、当該要員への指揮命令権は派遣先が持っている（請負契約および準委任契約の場合、委託側企業内で作業を実施していたとしても、委託側企業に指揮命令権がないことに注意する必要がある）。

表 38 請負契約・準委任契約と労働者派遣契約の比較

	請負契約・準委任契約	労働者派遣契約
契約の対象	ソフトウェアの開発作業	開発要員の提供・受入れ
作業場所	契約内容による	原則として委託側企業内
実施作業	契約書で指定される仕様書によって規定される	実施作業の規定はなく、派遣先での指示に従う。
受託企業作業者への指示	作業責任者への指示のみ可 （作業責任者以外に指示を与えることが常態化していると「偽装派遣」とみなされる）	可能

一方、非ウォーターフォール型開発を実施するプロジェクトの代表的な特徴として、以下の点を挙げることができる。

- 要件が未確定である
- 成果物が不明確である
- 性能と品質が不明確である
- 反復型開発を実施する場合、何回反復すれば開発が完了するか事前に見積るのが難しい（開発工数の見積りが難しい、工期が不明確である）

・ ユーザとベンダが連携して開発を実施する必要がある
・ ユーザとベンダのコミュニケーションを密にして開発を行う必要がある

これらの点を踏まえて、非ウォーターフォール型開発で、請負契約、および準委任契約を利用するときの課題をまとめたのが、次表である。なお、次表で【受】、【発】となっているのは、それぞれ、受注側あるいは発注側から見た問題点（リスク）を示している。

表 39　非ウォーターフォール型開発における請負契約と準委任契約の問題点

非ウォーターフォール型開発の特徴	請負契約	準委任契約
要件が未確定	【受】成果物の要件が決まっていないため、契約金額の範囲内で業務が完了できない場合がある。	【発】要件の確定に時間がかかると、その分委託費用が増加する場合がある。 成果物に不具合が発生した場合に、製造上の瑕疵に相当する内容であっても、受注者側に責任を取らせることができない。
成果物が不明確	【受】契約時に成果物を規定できないため、納入すべき成果物としての妥当性を主張できない。	【発】同上
性能と品質が不明確	【受】契約時に合意した内容がないため、発注側の要請により、工期が際限なく延び、契約金額の範囲内で業務が完了できない場合がある。 【発】工期延長に伴い、システムの稼動時期が遅れるリスクがある。	【発】工期の延長は、委託費用の増加につながるため、予算超過となる場合がある。
開発工数の見積りが難しい	【受】過小見積りにより、トラブルが発生しなかったとしても適正な対価が得られない場合がある。 【発】過大見積りにより、発生工数に比べて過大な契約額となる場合がある。	ー
工期が不明確	【受】契約期間が決まらないため、開発メンバのアサインができない。 【発】上記に加えて、システム稼動時期が想定できないリスクがある。	【発】委託費の額が想定できないため、予算超過になる場合がある。

非ウォーターフォール型開発の特徴	請負契約	準委任契約
ユーザとベンダの連携が必要	【受】【発】発注側と受注側の責任分解点を明確にできないため、開発したシステムに不具合が発生したときの責任の所在が不明確になるリスクがある。 【発】発注側メンバから受注側メンバへの指示が労働者派遣法に抵触する可能性がある。	【発】発注側メンバから受注側メンバへの指示が労働者派遣法に抵触する可能性がある。
ユーザとベンダのコミュニケーションが必要	【発】発注側メンバから受注側メンバへの指示が労働者派遣法に抵触する可能性がある。	同上

この表に示されているように、非ウォーターフォール型開発に、現行の請負契約あるいは準委任契約を適用しようとすると、種々のリスクが存在している。特に、請負契約では、受注側のリスクが高く、準委任契約では発注側のリスクが高くなっている点に注意する必要がある。したがって、非ウォーターフォール型開発を対象として、これらのリスクを緩和した上で発注側と受注側が対等な立場で締結できる契約が望まれている。

次表は、3.3で紹介した各事例における契約方式の扱いを示したものであるが、請負契約と順委任契約の件数がほぼ同数となっている。また、社内開発での事例も3件あり、契約時の問題を避けながら、非ウォーターフォール型開発を実践している例として考えられる。

表 40　調査事例における契約方式

事例	契約方式
ブログシステム	請負契約（毎月更新）
共通認証システム	請負契約
プロジェクト管理システム	請負契約
教務 Web システム	請負契約
システム管理ミドルウェア	請負契約
株式取引のための Web アプリケーション	請負契約
研修運営システム	準委任契約（推敲フェーズまで） ＋請負契約（作成フェーズ以降）
パッケージソフトウェア	準委任契約（四半期単位）
アプリケーションプラットフォーム	準委任契約
生産管理システム	準委任契約
小売業における業務システム	準委任契約

事例	契約方式
ソーシャルネットワーキングサービス（SNS）システム	準委任契約
Web メディア開発	準委任契約
共通 EDI 開発	準委任契約
検索エンジン	派遣契約
サプライチェーンマネジメントシステム	サービスの利用料金がビジネスの基本単位となる、ASP 契約
携帯ソーシャルゲーム	社内開発のため契約なし
教育機関向け統合業務パッケージ	社内開発のため契約なし
開発案件管理 Web アプリケーション	社内開発のため契約なし
アジャイル型開発の支援環境開発	社内開発のため契約なし
製造業向けプロトタイプシステム	不明
プラント監視制御用計算機システム	不明

７．２ 非ウォーターフォール型開発における今後の契約のあり方

ここでは、７.１で説明したような状況の下、非ウォーターフォール型開発に適した契約形態とはどのようなものであるか、海外での動向をベースに検討を行う。

７．２．１ Peter Stevens の分析

米国のスクラムトレーナの Peter Stevens が Scrum とアジャイル開発プロジェクトで利用することを前提として 10 種類の契約形態を比較している[12]ので、その概要を以下に示す。

Peter Stevens が比較を行った契約形態は、次の 10 種類である。

- Sprint Contract（スプリント契約）
- Fixed Price/Fixed Scope （固定価格／固定スコープ）
- Time and Materials （タイム・アンド・マテリアル）
- Time and Materials with Fixed Scope and Cost Ceiling （タイム・アンド・マテリアル 固定スコープとコスト上限付）
- Time and Materials with Variable Scope and Cost Ceiling （タイム・アンド・マテリアル 変動スコープとコスト上限付）
- Phased Development （フェーズ開発）
- Bonus/Penalty Clauses （ボーナス／ペナルティ条項）
- Fixed Profit （固定利益）
- Money for Nothing , Changes for Free （早期中止、変更無料）
- Joint Ventures （ジョイントベンチャ）

次表に、それぞれの契約形態の概要とその特徴をまとめる。

[12] http://agilesoftwaredevelopment.com/blog/peterstev/10-agile-contracts

表 41　契約形態の概要と特徴

契約の種類	内容
スプリント契約 Sprint Contract	【構造】 反復を１回実施するためにプロダクトオーナと開発チームの間で交わす合意であり、正式の契約ではない。 【スコープ】 開発チームは、その反復の終了までに、合意した機能について納品する。 プロダクトオーナは反復が終了するまで、スコープを変更しない。 【リスク】 スコープの変更に関して、各反復ごとに確認を行う。
固定価格／固定スコープ Fixed Price/Fixed Scope 【定額請負契約】	【構造】 合意した納品物について、納品を行う。定額契約であるため、作業量が増加すると利益は減少する。 【スコープ】 スコープの変更は、開発プロジェクトの遂行に関して高くつくので、制限されている。 【リスク】 見積りに誤りがあると、サプライヤ側にリスクがある。 【顧客とサプライヤの関係】 基本的に互いに無関心 顧客はより以上作業してくれることを望み、サプライヤはより少ない作業を行うことを望んでいる。
実費精算契約 Time and Material 【準委任契約】	【構造】 サプライヤは作業量に応じた請求を行う。変更に関するリスクは、顧客側が負っている。 【スコープ】 金額の上限は決まっていないため、顧客はある時点で、プロジェクトの終了を検討するようになる。 【リスク】 顧客がすべてのリスクを負っている。サプライヤには、コスト削減へのインセンティブがない。 【顧客とサプライヤの関係】 互いに無関心 サプライヤは仕事が続く限り、満足する。

契約の種類	内容
実費精算契約（固定スコープ、コスト上限付） **Time and Material with Fixed Scope and Cost Ceiling**	【構造】 上限に達するまでは、実費精算契約と同様に、サプライヤは作業量に応じた請求を行うが、コストが上限に達すると、支払いが受けられなくなる。 顧客側からすると、開発が早期に終了すれば、固定価格／固定スコープ型よりもコストを抑えることができる。 【スコープ】 固定価格／固定スコープと同様、スコープの変更は制限されている。 【リスク】 上限に達するまでは、開発コストが抑えられ、かつサプライヤもかかった経費を請求できるので、顧客視点では、双方にとって最適な契約と映る。 上限まで達すると定額契約となる。 【顧客とサプライヤの関係】 従属関係 サプライヤ視点では、コストを抑制した契約に映る。 上限金額以下でプロジェクトを終了させようというインセンティブをサプライヤは持たない。
実費精算契約（変動スコープとコスト上限付） **Time and Material with Cost Ceiling**	【構造】 実費精算契約と同じ。 コストの上限を設けておくことにより、顧客側のリスクが軽減される。 【スコープ】 固定価格／固定スコープに同じ。 【リスク】 顧客にとって、必要とするビジネス価値を得る前に、予算を使い切ってしまうリスクがある。 【顧客とサプライヤの関係】 協力関係 予算の制限とスコープの変更を組み合わせることにより、予算の範囲内で望ましい結果を得ようと、顧客とサプライヤの双方が同じ視点を持つ。

契約の種類	内容
フェーズ開発 Phased Development	【構造】 ４半期に１回行われるリリースに対して開発費用を提供する。リリースが成功すれば、次フェーズの開発費用が承認される。 【スコープ】 スコープ変更に関しては、モデルの中では明確に定義されていない。リリースは実際にはタイムボックスとなっていて、次の 4 半期には新たなリリースがあるということがわかると、タイムボックスの達成のためのある機能のリリース延期は受け入れやすくなる。 【リスク】 4 半期に 1 回支払われる開発コストの価値に、顧客のリスクは軽減される。 【顧客とサプライヤの関係】 協力関係 次回の資金提供が認められるように、顧客とサプライヤの双方に、4 半期に 1 回のリリースが成功させようというインセンティブが働く。
ボーナス／ペナルティ条項 Bonus/Penalty Clauses	【構造】 サプライヤはプロジェクトが早期に完了したらボーナスを受け取り、遅延したらペナルティを支払う。 【スコープ】 納期に影響を与えるため、スコープの変更の受け入れは困難である。 【リスク】 顧客側に開発を早期に終わらせるインセンティブが働かないと、時間をかけたにもかかわらず、貧弱な結果しか得られなくなる。 【顧客とサプライヤの関係】 構築可能な協力関係 ただし、合意した納期までに開発しているソフトウェアを、真に必要としていないことが判明した場合は、無関心状態に変わる可能性がある。

契約の種類	内容
固定利益 Fixed Profit	【構造】 目標納期前にプロジェクトが完了した場合は、それまでに発生経費に比例する形で利益を上乗せした支払いを行う。目標納期後に完了した場合は、発生経費に定額の利益を上乗せした支払いを行う。 【スコープ】 スコープは固定である。 【リスク】 リスクは共有している。 早期にプロジェクトが終了した場合、顧客側の支払いは少ないが、サプライヤも利益を確保できる。サプライヤ側から見ると、目標納期までにプロジェクトが完了すると利益率が高くなるが、目標納期を超えると、利益率は低くなる。 【顧客とサプライヤの関係】 協力関係 早期に終了すると、顧客側は経費が抑えられ、サプライヤ側は利益率が高くなるため、双方に、早期終了のインセンティブが働く。
早期中止、変更無料 Money for Nothing , Changes for Free	【構造】 実装しない機能があるようなアジャイル型ソフトウェア開発を対象とした契約である。 反復が終了した段階では、開発対象機能は、実装されたか、まだ着手されていないかのどちらかである。業務の基本は実費精算契約で遂行されていくが、反復的に機能を実現していくうちに、ある段階でそれ以上の機能は実現しなくても、ビジネス上の価値として十分なものに達したと判断できるレベルになる。顧客はこの段階でプロジェクトを中止し、サプライヤに対するキャンセル料として、そのままプロジェクトを継続したときに得られる利益の相当する額を支払う。 【スコープ】 スコープは変更可能である。計画されたが実装されなかった機能については、同規模の別の開発プロジェクトとして実施する。 【リスク】 リスクは共有している。 顧客側、サプライヤ側の双方とも、プロジェクトを早期に終了させることに関心を持っている。

契約の種類	内容
ジョイントベンチャ Joint Ventures	【構造】 顧客側、サプライヤ側の相応が、共同出資の形でプロダクトに投資する。 開発フェーズに両者の間で金銭の移動が生じるのではなく、そのソフトウェアを利用することで得られる利益を共有する。 【スコープ】 パートナーシップのニーズにあわせて定義する。 【リスク】 意思決定が長引くことと、チーム間で競争が生まれることの 2 点がリスクである。

この Peter Stevens の分析をまとめると、次のようになる。

- 契約のベースとしては、実費精算（Time and Material）契約を推奨している。
- 実費精算契約は、プロジェクトが長引くと顧客のコストが肥大化するため、顧客側とサプライヤ側の双方に、プロジェクトの早期終了に関するインセンティブが働く枠組みを導入する必要がある。
- 最終的に、顧客とサプライヤの双方が、win-win の関係を築けることが必要である。

７．２．２ デンマークの事例[13]

デンマークのコンサルティング企業である BestBrains 社では、非ウォーターフォール型の開発プロジェクト分野を対象とした新しい契約方式を試行している。この契約方式は、以下の特徴を持つ対象としている開発プロジェクトを対象としている。

- サプライヤが反復的にソリューションを開発し、詳細について段階的に解決できるもの
- プロジェクトの全体を通してハイレベルの開発品質を保つもの
- 協調的な作業と、プロジェクト全体に有利に働くソリューションの提供の努力に関するインセンティブを顧客とサプライヤに与えるもの
- 適正な規模の機能性によりプロジェクトを早期に終了させるインセンティブを顧客とサプライヤに与えるもの

BestBrains 社が用意した新しい契約では、以下の項目を定義している。

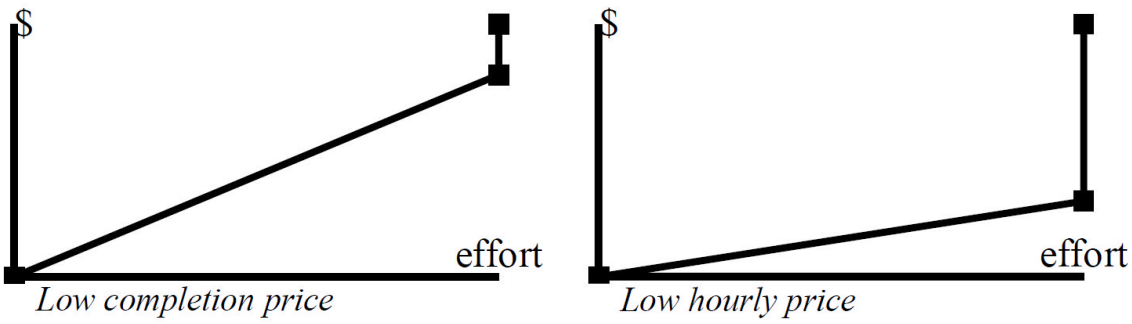
- ある種のビジョンステートメントを数段落で粗く記述したスコープ
- 純粋の Time and Material 契約における通常の費用から比べると 10～50%に相当する時間単価
- 定額の支払を行うためのマイルストーンの集合。それぞれのマイルストーンでは、その時点で顧客がソフトウェアのデプロイを実際に実施することが、完了基準となる。
- アジャイルプロセスに従った開発プロセス
- プロジェクト全体と個々のマイルストーンに関する暗黙的な時間フレーム

この契約では、プロジェクト費用に関して、時間単価のほかに完成時費用を採用してい

[13] Lars Thorup, Bent Jensen, "Collaborative Agile Contracts," agile, pp.195-200, 2009 Agile Conference, 2009

る。時間単価と完成時費用は、プロジェクト規模に従って算定される。具体的には、全体費用を時間単価分と完成時費用分に配分するため、プロジェクト費用総額を時間単価に割り当てると、Time and Material 契約になり、逆に費用総額を完成時費用に割り当てるとFixed-Price 契約となる。プロジェクト費用総額から完成時費用を差し引いた額を、プロジェクトの見積り工数で割って、時間単価を算出することになるため、通常の Time and Material 契約と比較して 10～50%の時間単価が実現できることになる。

完成時費用を低くした場合と時間単価を低くした場合の費用発生の仕方をグラフで示したものが、次図である。



完成時費用を低く設定した場合　　時間単価を低く設定した場合

（出典）Lars Thorup, Bent Jensen, "Collaborative Agile Contracts"

図 10　BestBrain 図社の契約におけるコストモデル

BestBrains 社は、この契約を使って、次の 2 つの実プロジェクトを実施した。

- 事例 1
  対象企業：小さなイベント事務所
  開発システム：対話型のプラットフォーム
  規模：3 人×3 週間
- 事例 2
  対象企業：大規模なエネルギー関連企業
  開発システム：電力プラントの登録システム
  規模：3 人×9 ヶ月

事例 1 では、完成時費用をプロジェクト総額の 10%として実施したため、納期を守ろうとするインセンティブが低くなったので、完成時費用の割合をもう少し高くすべきであるとの分析が行われている。

一方、事例 2 では、時間単価を低く設定したところ、顧客は、納期を守ることよりも低い単価で多くの機能を実現することを求めてきたため、納期が 20%遅れてしまい、BestBrains 社の利益を下げることにつながったということである。この結果から、時間単価は、通常の Time and Material 契約の 50%程度に設定することで、顧客は速い time-to-market を目指すことになるとの分析を行っている。

７．３ 契約方式上の課題解決の方法

７.２では海外の事例を見てきたが、法制度上の違いもあって海外で成功している考え方をそのまま国内に持ち込むことができない部分もある。ここでは、上述の事例を参考にしながら、７.１で提示した課題解決に向けた方向性を示すこととする。

７．３．１ 契約書の位置づけ

契約書が必要となる局面は、発注側と受注側の間でトラブルが発生し、その解決のための拠り所を契約書に求める場合である。

したがって、発注側と受注側で win-win の関係が築けていて、開発作業の実施内容に双方が満足していれば、契約書の文面が請負契約と準委任契約のいずれであっても、問題となることはない。

ただし、いったんトラブルが生じてしまうと、その責任を発注側、受注側のいずれかが負う必要性が発生してしまう。このため、トラブルが発生しないとは言い切れない以上、トラブル発生時のリスクをヘッジする必要があり、契約書の文面で発注側と受注側の役割を明確にしておくことが求められることになる。

７．３．２ 契約書に記載すべき内容

非ウォーターフォール型開発の特徴は本章の冒頭で説明したが、そのベースとなっているのは、明確でない要求を具体的な内容に組み上げていくために、反復的に開発を実施するということである。

７.２で示した海外の事例が非ウォーターフォール型開発を扱う契約には Time and Marterial 契約が向いていること示しており、また国内のこれまでの商習慣上、成果が明確に規定できない場合には準委任契約を利用することになっていることを考えると、非ウォーターフォール型開発を対象とした契約のベースには、準委任契約（Time and Marterial 契約）を利用することが望ましい。

次に、準委任契約をベースとしたときに、契約条項として盛り込むべき内容を以下に挙げる。

- 利害関係者とその役割
  非ウォーターフォール開発は、発注側と受注側の協調作業があってはじめて成功するものであるので、契約書（仕様書を含む）の中で、当該開発作業の利害関係者を明記するとともに、それぞれの役割を明記する必要がある。特に、利害関係者間のコミュニケーションの方法や、実施時期、参加者等に関するコミュニケーション計画関わる事項については、契約段階に明らかにしておく必要がある。
- 反復に関する考え方
  上述のように、反復は非ウォーターフォール型開発のベースであるので、その実施内容を含めて事前に明らかにしておく必要がある。具体的には、1 回の反復期間およびその実施内容、反復における発注側の役割、などが挙げられる。また、可能であれば、反復の終了条件を、期間面、機能面（たとえば、予定開発機能に対する実装率）、予算面（累積開発費用）といった観点から明らかにしておくことも望ましい。
- 完成責任の考え方
  開発作業を発注側と受注側が協調して実施する以上、これまでの準委任契約のような、発注側がすべての開発責任を負うという考え方はそぐわないことになる。反復

への発注側の積極的な関与など、発注側が実施すべき責任を適切に果たすことを条件にして、準委任契約であっても受注側にも開発責任を負わせることが必要である。

・ インセンティブの付与
海外の事例では、プロジェクトの早期終了を実現するために、発注側、受注側双方が目標を共有し、その上で目標達成に向けたインセンティブを持たせる必要性が強調されていた。インセンティブ付きのソフトウェア開発契約は国内ではなじみが薄いが、どのような形で実現可能かを含め、検討を行う必要がある。

以上、契約条項として盛り込むべきことを検討してきたが、国内では実プロジェクトへの非ウォーターフォール型開発の適用経験そのものが少なく、実施上の問題としてどのようなものが発生するかも明確になっていない状況である。

このため、非ウォーターフォール型開発を対象とした契約形態にしても短期的に検討して終わるのではなく、実務を通じて、適宜、評価・見直しを、産官共同で継続的に実施していく必要がある。

８ まとめ

８．１ 非ウォーターフォール型開発に関する国際的な動き

本調査では、非ウォーターフォール型手法を形成するプラクティスや原則に着目し、その実践がもたらす正負の効果について、様々な角度から調査を行った。我が国における非ウォーターフォール型開発の適用実態を把握するため、20 を超える事例の収集、分析を行った。

ここでは、参考までに海外で実施されたアンケート調査結果例の概要を示し、主要な事項について、わが国の事例の傾向との対比を行う。なお、当該アンケート調査結果のまとめは付録 2.に掲載する。

(1) アジャイル型開発に対する期待と気掛かり

研究会の中でも議論があったとおり、海外では非ウォーターフォール型開発、特に、アジャイル型開発は開発企業のステータス的な側面も持っており、既にこれまでのアジャイルの次は何か、といった議論がなされているという状況であるといわれている。その意味で、研究会の中でわが国の状況は周回遅れとも表現されることも少なからずあった。

そのような海外の状況を見てみると、まずアジャイル型開発に対する期待については、わが国においても期待されている「Time-to-Market の加速」と「要件等の変化（特に、優先順位）へ対応」することが目的としては高い。また、気掛かりについても、「事前計画の欠如」、「管理コントロールの欠如」、「文書の欠如」、「予測性の欠如」が 35%から 40%の回答で見られる。これは、一般に言われているアジャイル手法に対する気掛かりと一致しており、一般にそう思われている様子が分かる。このあたりの見方については、特に海外とわが国において齟齬はないと見てよいと思われる。

(2) アジャイル型開発の手法

一方、実践している手法についてみると、活用している手法は、約半数がスクラムと他の手法を圧倒している。続いて、スクラムと XP の組合せで、XP 単体での活用が続く。ここまでで、ほぼ 8 割はカバーされている。スクラムは、スクラム･マスターなどの人材育成などの制度面での整備を行っており、これらの効果が上がっていると見ることができる。このあたりの傾向は、日本では海外等で提示された手法を自らのビジネス環境や開発環境に合わせて工夫を凝らすという対応が見られ、スクラムや XP という手法全体として捉えられるよりは、それらを換骨奪胎して活用している事例が多く、やや対応に違いがあると考えている。

(3) アジャイル型開発のプラクティス

実際のプラクティスを見てみると、8 割を超えて活用されているプラクティスが、「反復型計画」である。続いて 7 割を超えているのが、「ユニットテスト」、「デイリー･スタンダップ」、「リリース計画」である。

6 割を超えているのが、「継続的インテグレーション」、「自動ビルド」、「ハーンダウンチャート」である。ここまでで、およそ、計画及び進捗の確認のためのプラクティスとともに、構築及びテストの基本的なプラクティスが見えている。

5 割以上で見ると、49%の「テスト駆動型開発」を含めて、「リファクタリング」、「レトロスペクティブ」、「コーディング作法」「スピード」とやはり、進捗と繰り返しの中での次

への対策､繰り返しでもデグレを起こさせないための手法が活用されている様子が伺える。プラクティスレベルでは、海外で活用比率が高いものは、わが国の事例でも採用されている場合が多く、(2)で述べた傾向があることが分かる。

８．２ わが国における非ウォーターフォール型開発について

事例調査を通じて、非ウォーターフォール型開発に関する課題を、抽象的にではなく開発現場に密着した具体論として見てきたが､その結果、８.１項で示すような傾向とともに、今後、非ウォーターフォール型開発をさらに普及させていくために押さえるべきいくつかの観点を明らかにすることができた。

(1) ビジネス等のコンテキストに応じた開発方法の選択

第一に、ウォーターフォール型開発と非ウォーターフォール型開発を二元論的にとらえ、どちらがより優れているかという議論はもはや意味をなさず、開発するソフトウェアの特性やプロジェクトに与えられる制約などを踏まえ、妥当な開発手法を定めた結果として、ウォーターフォール型開発色が強い場合もあれば、その逆の場合もあるという状況と考えられる。

今回調査した各事例においても、それぞれの取組みに至った背景として、開発手法を導入することが目的ではなく、ビジネスまたは開発を取り巻く環境に応じて適切な開発プロセス・手法を探っていった結果、それぞれの事例のような取組になったことが基本であった。

例えば、ステークホルダが多く、確認を取りながら開発を進める必要がある場合は、ウォーターフォール型となる傾向にあり、反対に期間が短く少人数で行う開発は非ウォーターフォール型となる傾向が見られるように、プロジェクトを取り巻く環境に応じて適切な取組み方が決められている。

別の例としては、新規ビジネスを小さく始め様子を確認しながら拡大していく場合は、アジャイル的なプロセス・手法で実現することを志向する傾向にあり、マーケットを押さえるために大々的に事業を展開する場合は、様々なことを決めてからウォーターフォール的に一気に開発することを志向する傾向になるなど、同じ企業内でもプロジェクトや組織の特性に応じて採られる開発手法が変わってくることが指摘されている。

(2) プラクティスの活用

非ウォーターフォール型の様々な手法において提唱されているプラクティスは多数あるが、各事例で適切なものが選ばれ、ただ単に定義に従って実践されるだけでなく、自らの状況やニーズにあった工夫を行ったうえで実践されていた。これは、それぞれのプロジェクト・組織（企業）で、自らの開発にあった方法を非ウォーターフォール型の開発プラクティスを参考にして利用するのが良いということを示唆するものと考えられる。

(3) 非ウォーターフォール型開発の手法の適用上の留意点

非ウォーターフォール型開発の手法については、プロジェクトマネジメント上の難しさ、品質･信頼性確保の課題などが指摘されることがある。これらは手法自体の課題ではなく、その使い方（効果を出すための前提条件を考慮せずに導入するなど）の問題であったり、従来型の開発手法を適用した場合と同等の成果をあげるために何をどのように実施すれば

よいか十分に把握できていないことからくる不安であったり、新しいパラダイムを従来型の開発方法論に当てはめて理解しようとするために起こる誤解、などがその背景にあるものと考えられる。

例えば、非ウォーターフォール型の開発では顧客の関与は極めて重要なポイントであるが、その関与が担保されないまま非ウォーターフォール型の開発を進めて成功させるのは難しい。また、開発のリズムとスピードが非ウォーターフォール型のメリットであるが、実際にチームメンバの資質やスキル等がある程度そろわないと、崩壊に瀕してしまう可能性が高い。

ウォーターフォール型の開発であっても非ウォーターフォール型の開発であっても手法ありきではなく、プラクティスの意図や、なぜそのようなプラクティスが提唱されているのかといった背景についても理解を深めなければ、期待する効果を得るための真の条件や正しい取り組み方は見えてこない。

社会あるいはビジネスの変化やソフトウェア開発技術の進化などにより、ソフトウェア開発を取り巻くパラダイムが大きく移り変わっている。その新しいパラダイムに適応していくために、これまで獲得してきた知見と非ウォーターフォール型開発という新しい発想とをうまくブレンドして活用すれば、1 つ 1 つのプロジェクトが成功裏に推進されるようになるのではないだろうか。

それこそが、技術者が生き生きと働ける環境づくりの出発点であり、我が国の産業競争力の向上の源泉となるものであると考えている。

付録編

1 プラクティス一覧

プラクティスID	プラクティス名	プラクティス概要
XP-1	リリース計画ゲーム	顧客にとってソフトウェアの価値が最大になるよう、次の運用リリースのスコープを定義。顧客が必要な機能を説明するためにストーリーカードを書き、開発者がその作業時間を見積もる。それから、顧客は次のリリースでどのカードを実装するのかを選択する。
XP-2	イテレーション計画ゲーム	顧客はイテレーションで実装するストーリーカードを選択する。そのストーリーカードごとに、プログラマがストーリを実現するためのタスクリストを（カードまたはホワイトボード上で）作成する。
XP-3	小規模で頻繁なリリース	可能なら1〜3週間ごとにリリース。顧客はリリースごとにフィードバック。
XP-4	システムのメタファ	設計をうまく伝えるために、システム全体や各サブシステムを覚えやすいメタファで表現し、主要なアーキテクチャのテーマを描写。
XP-5	シンプルデザイン	将来の変更の可能性を推測して設計しない。今すぐ必要でない汎用のコンポーネントを作成しない。コードの重複をなくし、クラスやメソッドのが比較的少なく抑えられた、簡単に理解できる設計をすべき。
XP-6	テスト駆動開発	テスト対象のコードよりも先にプログラマが単体テストを書く。
XP-7	受け入れテスト	全ての機能は自動化された受け入れ（機能）テストに組み入れる。受け入れテストは顧客と協力して作成する。顧客はテストに使える内容の受け入れ基準定義書を作成する。
XP-8	頻繁なリファクタリング	全てのテストが通ることを確認しながら、きめ細かいコードや大規模な設計要素を単純化する。
XP-9	ペアプログラミング	アプリケーションコードは必ず、２人のプログラマが１台のコンピュータに向かって作成する。定期的に交代しながら交互のコーディングをする。ペアは、タスクが変われば頻繁に組みなおされる。見ている側の人は、リアルタイムコードレビューを行っていることになり、おそらくは入力している人よりも幅広い考えが出来る（テストなどについて考えられる）。メンバが相互に学習すること。もっと規律を持ってプラクティスを遵守し、だらだらせずにもっと多くの時間を実際のプログラミングに割くよう仲間同士でプレッシャーをかけ合うこと。リアルタイムのコードレビューによって欠陥を削

		除すること。そして、ペアの一人が行き詰まったときでも先に進み続けられる根気と洞察力を持つこと。
XP-10	共同所有権	チーム全体が共同で全てのコードに責任を持つ。変更要求を必要としないため、開発速度が向上する。単体テスト・受け入れテストが存在し、継続したインテグレーションによって、コードに問題が起きたときにはそれがわかる。また、共通のコーディング規約が守られていると、どのコードも似たようなものになっている。
XP-11	継続的インテグレーション	チェックインされたコードは全て、継続的に結合テストが繰り返される。
XP-12	持続可能なペース	時間外労働無しを勧める。
XP-13	チーム全体が一緒に	顧客にプロジェクト専任者を出してもらい、開発者と共通のプロジェクトルームに常駐してもらう。
XP-14	コーディング規約	コードの共同所有、頻繁なリファクタリング、ペアプログラミングのパートナーの頻繁な組み換えを行うためには、全員が同じコーディングスタイルに従う必要がある。
Scrum-1	ゲーム前計画	ゲーム前計画の期間中、利害関係者はだれでも、機能、ユースケース、機能拡張、欠陥などの一覧を作成し、「プロダクトバックログ」に記録することが出来る。プロダクトバックログの所有者として一人のプロダクトオーナが任命され、要求はプロダクトオーナを通じて取り決められる。
Scrum-2	スプリント計画	各イテレーション（スプリント）を開始する前に、２つのミーティングを続けて開催する。最初のミーティングでは、利害関係者があつまって、プロダクトバックログとリリースバックログの精度を上げ、優先順位を付け直し、今回のイテレーションの目標を決定する。優先順位は、通常最も高いビジネス上の価値やリスクによって決定される。２つ目のミーティグでは、スクラムチームとプロダクトオーナが集まって要求をどう実現するか熟考し、目標を達成するためのタスクを列挙した「スプリントバックログ」を作成する。イテレーションの作業が進むにつれ、スプリントバックログは更新される。
Scrum-3	原則 30 日（カレンダー上)のイテレーション	作業は原則 30 日（カレンダー上）のイテレーションに分割。それぞれをスプリントと呼ぶ。
Scrum-4	スプリントバックロググラフの作	スプリントバックログのタスクの推定残り作業時間をグラフにまとめたもの（バーンダウンチャート)。毎日スクラムミーティングまでに、このグラフの更新版を壁に貼り出すことが

	成	推奨されている。
Scrum-5	自律的な組織化チーム	イテレーションの期間中、経営陣やスクラムマスターは、イテレーションの目標をどのようにして達成するか、問題をどのようにして解決するか、作業順序をどう計画するかについて、(判断を依頼されたり、報告された障害を取り除く必要がない限りは) チームに口出しをしない。
Scrum-6	スクラムミーティング	毎日、同じ時間、同じ場所で、チームメンバは輪になってミーティングを開き、特定の同じ質問をして、各チームメンバがそれぞれ答える。立ったまま行う。15～20 分程度で終わらせる。ホワイトボードの前で行い、報告されたタスクや障害はすべて書き留める。書き留めた障害をスクラムマスターが消すのは、その障害が取り除かれた場合だけである。その場にいないメンバが参加できるように、スピーカーで会話できる電話機を使う (参加は必須である)。 質問は次のとおり。 1.前回のスクラム以降、何を行ったか？ 2.今から次回のスクラムまでに何を行うか？ 3.イテレーションの目的を達成する上での障害は何か？ 4.スプリントバックログに追加するタスクはあるか(新しい要求ではなく、忘れられていたタスク) 5.チームメンバから刺激を得て (技術的なこと、業務知識などで) 何か新しく学んだことはあったか？
Scrum-7	イテレーションに追加してはならない	イテレーション期間中、経営陣はチームや個人に対して作業を追加してはならない。中断されずに集中できるようにする。万が一何かを追加しなければならない場合は、代わりに他の作業をなくすのが理想である。
Scrum-8	スクラムマスターのファイアウォール	スクラムマスターは、チームが外部からの作業依頼によって作業を中断されないよう気を配る。中断された場合には、その原因を取り除いたり、政治的な問題や外部のマネジメントの問題に対応したりする。
Scrum-9	1 時間以内の判断	スクラムミーティングで障害が報告され、スクラムマスターの判断が必要な場合には、即時に、あるいは 1 時間以内に判断を下すのが理想である。「悪い判断でもないよりはましだし、後で破棄することもできる」という価値が奨励されている。

Scrum-10	1 日以内の障害除去	スクラムミーティングで報告された障害は、次のミーティグの前に取り除かれるのが理想である。
Scrum-11	ニワトリとブタ	スクラムミーティングの間はスクラムチーム（ブタ）だけが発言することが出来る。そのほかの人（ニワトリ）は、出席できるが黙っていなければならない。CEO も同様である。
Scrum-12	7 人のチーム	1 チームのメンバは最大 7 人にすることが推奨されている。大規模プロジェクトでは複数のチームを編成することになる。
Scrum-13	共通の部屋	チームは共通のプロジェクトルームで一緒に作業するのが理想である。
Scrum-14	日次ビルド	少なくとも 1 日 1 回、プロジェクトのチェックインされたコードをすべて統合し、回帰テストを行う。
Scrum-15	スプリントレビュー	各イテレーションの最後には、スクラムマスター主催でレビューミーティングを行う（最大 4 時間まで)。チーム、プロダクトオーナ、その他の利害関係者が参加する。このレビューでは製品のデモを実施する。レビューの目的の一つは、システムの機能、設計、長所/短所、チームの作業、今後の問題が置きそうな箇所について、利害関係者に知らせることである。
FDD-1	ドメイン・オブジェクト・モデリング	問題領域における重要なオブジェクトを記述するクラス図によって構成される。新しい機能や能力をシステムに追加することが容易になる。
FDD-2	feature 毎の開発	ドメインオブジェクトモデルの中でクラスを特定できたら、それをそれぞれ設計し、実装を行う。ほとんどの feature が数時間から数日以内で実装されるように十分小さく分割する。
FDD-3	クラスの責任者	誰（人またはロール）がクラスの内容に責任を持つかを明らかにする。
FDD-4	feature チーム	熟練した開発者たちを各チームリーダに設定し、それぞれに複数の feature を割り当てる。
FDD-5	インスペクション	設計とコードを高品質に保つため、インスペクションを実施する。
FDD-6	構成管理	完成した feature のソースコードを特定することができ、クラスに対して行った変更履歴を保管できる機能を持つ構成管理システムが必要である。
FDD-7	定期ビルド	完成した feature の全てのソースコードと、システムが依存するライブラリやコンポーネントを含めて、定期ビルドを実施する。
FDD-8	結果の申告と可視化	プロジェクトの現状を正しく認識し、開発チームが新しい機能を追加するのにどのくらい時間がかかるかを知ることによって、チームリーダや管理者が、プロジェクトを正しく進めるための情報を得ることができる。
Lean-1	無駄をなく	無駄な機能を作らない。

	す	
Lean-2	品質を作りこむ	バグトラッキングシステムに欠陥情報を入力するようなことはしない。テスト駆動型開発や継続的統合によって、正しいコードしか存在しない状態にする。
Lean-3	知識を作り出す(顧客からのフィードバック)	最小の機能セットを早期に顧客にリリースし、評価とフィードバックを得る。
Lean-4	知識を作り出す(テストからのフィードバック)	毎日ビルドし、統合テストからすばやいフィードバックを得る。
Lean-5	知識を作り出す(適切な決定)	経験と直感を備えたチームやリーダに適切な決定を下させる。
Lean-6	知識を作り出す(追加容易なアーキテクチャ)	新規機能が追加しやすいモジュールアーキテクチャにする。
Lean-7	決定を遅らせる	手遅れにならずに決定を下せる最後のチャンスまで取っておく。
Lean-8	速く提供する	今しなければならないことだけして、速く提供する。
Lean-9	人を尊重する	人が頭を使い、自らそれを見つけ出すことのできる、自律管理能力を備えた組織を作る。
Lean-10	全体を最適化する(顧客ニーズを満たす)	顧客のニーズを満たすための注文を受けた時点から、ソフトウェアが導入され、ニーズが解決されるまでのバリューストリーム全体を最適化する。
Lean-11	全体を最適化する(インセンティブを備えた契約)	全員が確実に全体の最適化を重視するようになるインセンティブを備えた契約や外注契約や部門間の協力合意を結ぶ。
C.Clear-1	短くてリッチなコミュニケーションパスがある	全員が一つに部屋にいることが望ましい。
C.Clear-2	1~3 か月以内に出荷す	ドキュメント整備のマイルストーンではなく、コードのマイルストーンを管理する。

	る	
C.Clear-3	真のユーザをプロジェクトに巻き込む	ユーザは開発者のスクリーンスケッチやバリデーションの手助けをする。
C.Clear-4	プロジェクトの要諦を把握する	何らかの要求記述フォーマットを利用し、何らかの記述言語を用いてシステムデザインを把握する。
C.Clear-5	作業成果物の所有者を決める	所有者は個人かサブチームかチーム全員かもしれないが、しっかり決めておくことが必要である。
C.Clear-11	頻繁なリリース	実行可能であり、テストされたコードを数か月ごとに真のユーザに引き渡す。
C.Clear-12	反省と改善	チームが一緒に、何がうまくいって何がうまくいっていないかをリストアップし、何をすればより良くなるかを議論し、次のイテレーションに生かす。
C.Clear-13	浸透したコミュニケーション	関連した情報をチームメンバに徐々に浸透させる。これはチームメンバが同じ部屋で作業することによって実現可能である。
C.Clear-14	個人の安全	報復を恐れることなく、問題点を指摘できるような環境を整える。
C.Clear-15	集中	まず何をすべきかを知り、それを実施するための時間と心の平穏を持つ。何をすべきかは、ゴールの方向性や優先順位についての議論から知ることができる。時間と心の平穏は、人々が作業に没頭できる環境を整えることによって得られる。
C.Clear-16	エキスパートユーザとのコミュニケーション	頻繁なリリース、成果物の品質に対する早いフィードバック、決定された設計についての早いフィードバック、要求の更新を行うことによって実現される。
C.Clear-17	自動テスト、構成管理、頻繁なインテグレーションにおける技術的な環境	自動化されたテストは必須ではないが、開発効率を向上させる。構成管理により、開発者が別々に、かつ、共同で開発を行うことができる。一日に複数回インテグレーションすることが望ましい。それが無理なら 1 日に 1 回、最低でも 2 日 1 回はインテグレーションを行うべきである。これら 3 つを併せた、テストを含めた継続的なインテグレーションを行うことが望ましい。
C.Clear-18	360 度の探索	新しいプロジェクトを開始する際、数日から最大 2 週間かけて下記の点について考えなければならない。 ・ビジネス価値 ・要求 ・ドメインモデル

		・技術計画 ・プロジェクト計画 ・チーム構成 ・プロセスや方法論
C.Clear-19	早期の成功体験	とりあえず一つ動くものを作成する。最初に小さな勝利を味わうことで、チームメンバが自信を持つようになる。
C.Clear-20	実行可能なスケルトン	小さな機能を持つシステムを作る。これは最終的なアーキテクチャを用いる必要はないが、おもなアーキテクチャコンポーネントとの関連を明らかにしておく必要がある。
C.Clear-21	インクリメンタルなアーキテクチャの再設計	システムアーキテクチャは、「実行可能なスケルトン」から徐々に進化していく必要がある。進化は、技術やビジネス要件が時間とともに変化するに従って行われる。
C.Clear-22	情報発信	メンバが作業中や歩いているときに見える場所に情報を置く。
C.Clear-23	方法論の改善	より良い経験についての情報を収集する。まずプロジェクトのインタビューを行い、それに基づいてワークショップを開催する。
C.Clear-24	反省ワークショップ	各リリース後、何がうまくいって、何を改善すべきで、次のイテレーションにおいて何をすべきかのワークショップを開催する。
C.Clear-25	集中的な計画	速くて協調的なプロジェクト計画を行う。たとえばXPの計画ゲームを利用できる。
C.Clear-26	デルファイ見積もり	デルファイ法を用いてプロジェクト全体の見積もりを行う。
C.Clear-27	毎日のスタンドアップミーティング	チームの情報共有を早く効率的に行う。
C.Clear-28	アジャイルインタラクション設計	ワークショップの開催、UIの抽出、ユーザビリティのインスペクション、QAテストとシステムのパーソナライズ化によってアジャイルインタラクション設計を実施する。
C.Clear-29	プロセスの簡易版	新しい開発プロセスを、長期のプロジェクトでいきなりそのまま適用することは難しい。まずは短い時間の中でプロセスを適用し、メンバに慣れさせる。
C.Clear-30	Side-by-side プログラミング	ペアプログラミングに代わるものである。メンバはお互いのディスプレイが見える程度に近い場所に座り、別々に作業を行う。ときどき、メンバは隣の席の人のプログラミングを見ることが可能であり、必要に応じて共同作業を実施する。
C.Clear-31	バーンチャート	バーンチャートを用いてプロジェクトの進行を可視化し、管理する。

RUP-1	反復型開発	2 週間から 6 週間のタイムボックス化されたイテレーションによって開発する。
RUP-2	要求管理	洗練された方法で「発見」「整理」「追跡」することで要求を管理する。事前に大規模な分析を行うのではなく、反復的、インクリメンタルに要求を発見・改良する。たとえば、開発の初期に 1 日というタイムボックスを設定した短い要求ワークショップを何回か（イテレーションごとに 1 回ずつ）行う等。ツールを使って、リスク、優先度といった属性や、他の要求との依存関係をもとに要求を整理し、分析しやすくする。ツールを使うことで、要求の状態を追跡でき、何が終了し、何が作業中かなどを知ることができる。
RUP-3	コンポーネント・アーキテクチャーの使用	ハイリスクまたは重要な要素から開発すること、中核アーキテクチャを初期のイテレーションで構築すること、規模の大小を問わず既存コンポーネントを再利用して新規のコードや欠陥を減らそうと努力する。
RUP-4	ビジュアル・モデリング	プログラミングを始める前に、少しでも視覚的なモデリングを行う必要がある。1 時間ホワイトボードに向かって絵を描く等。UML に完全に準拠する必要はない。
RUP-5	品質の継続的検証	早い段階から頻繁にテストを行う。実際には、イテレーションごとに全てのソフトウェアを統合しテストする（単体テスト、システムテスト、負荷テスト)。また、定期的にミーティングを行い、さまざまな作業について評価することで、使いやすさや、コード以外の成果物（要求など）の品質、プロセス自体についても早い段階で検証すべき。
RUP-6	変更管理	規律のある構成管理およびバージョン管理、変更依頼の手順、各イテレーションの最後にリリースをベースライン化するなどによって、変更を管理する。
OP-1	進化型プロトタイプの作成	要求が確定している部分についての進化型プロトタイプを作成する。
OP-2	プロトタイプ専門家の派遣	ユーザにプロトタイプを送り、また、プロトタイプの専門家を派遣。プロトタイプの専門家がソフトウェアを利用するユーザを観察。ユーザが問題に直面したり、新たな要求や機能を思いついたら、プロトタイプの専門家がそれを記録。これにより、ユーザが問題を記録する必要がなくなり、業務に専念できる。
OP-3	使い捨て型プロトタイプの作成	ユーザの利用が終わると、プロトタイプの専門家はベースのプロトタイプを改良し、使い捨て型のプロトタイプを作成する。ユーザはその新たなソフトウェアを使って評価する。変更が効果的でない場合は、プロトタイプの専門家はそれを削除する。

OP-4	新たな進化型プロトタイプの作成	ユーザが変更を気に入った場合、プロトタイプの専門家は機能改善要求を書いて開発チームに送る。開発チームはそれに基づき、新たな進化型プロトタイプを作成。
DSDM-1	タイムボックス	実施すること： ・プロジェクト終了日を固定し、その日までにすべてのビジネス要求が実現される ・プロジェクト内の各インクリメントの最終日を固定し、優先度の高いビジネス要求や技術的要求をその日までに満たす ・フェーズレベルのアクティビティの最終日を固定し、このプロジェクトのための目的を定義する ・ワークショップ、ミーティング、レビューの終了時刻を固定し、参加者はあらかじめ定義された優先度の高い目的のために働く 重要事項： ・Must Have は 60%のエフォート、Should Have は 20%のエフォート、Could Have は 20%のエフォートで実現できるようにタイムボックスを固定する（Must Have を 100%に設定すると、見積もりが誤っていたときに問題が発生する）。
DSDM-2	MoSCoW	要件に優先度を付ける方法。 必須（Must have）：これがなければシステムが成り立たないような要件 必要（Shuold have）：重要な要件 要望（Could have）：できればあった方がいい要件 次回（Want to have）：次回実現したい要件
DSDM-3	プロトタイピング	4 種類のプロトタイプ。 ビジネス・プロトタイプ：システムの評価 ユーザビリティ・プロトタイプ：UI の確認 性能/容量プロトタイプ：性能や容量の制約を満たすことの確認 機能/設計プロトタイプ：オプションの評価
DSDM-4	ワークショップ	さまざまな利害関係者を集め、話し合いの場を持たせる。
DSDM-5	積極的なユーザ関与が絶対に必要である。	DSDM はユーザを中心とした手法である。もしユーザが開発のライフサイクル中にあまり関与できないなら、意志決定の反映が遅れ、開発に遅延が発生するだろう。
DSDM-6	チームに引き渡しの権	DSDM チームは開発者とユーザから成り立つ。要求が洗練・変更されることに対応するため、彼らは意思決定を行う権限

	限が与えられなければならない。	を持たなくてはならない。
DSDM-7	頻繁な引き渡しが鍵である。	製品ベース手法がアクティビティベース手法よりも柔軟である。DSDM チームの役割で重要なものは、同意した時間間隔において製品を引き渡すことである。要求される製品をより早く実現することが可能となる。引き渡しまでの期間を短く設定することにより、正しい製品をより効率よく作成することが可能となる。
DSDM-8	受け入れの主たる条件は現在のビジネスニーズを満たす機能の引き渡しである。	DSDM において重要なことは、必要な時に必要な機能を提供することである。このアプローチが受け入れられれば、コンピュータシステムを後でより厳密に開発することができる。
DSDM-9	反復的で漸増的な引き渡しが不可欠である。	DSDM はインクリメンタルにシステムを開発する。したがって、開発者はユーザから常にフィードバックを得ることができる。また、部分的に完成した製品は、ビジネスニーズを即座にある程度満たすことができる。
DSDM-10	プロジェクトライフサイクル中に行われた変更はすべて元に戻せる。	全ての製品（ドキュメント、ソフトウェア、テストコードなど）の進化を管理するため、全ての製品の変更を保存しておかなければならない。
DSDM-11	要件は高いレベルで基準化される。	細かい部分を後で決めることを許容しながらも、システムの目的とスコープを高いレベルにおいては固定して同意することが必要である。高いレベルにおいて決定したスコープを後で変更する場合は、通常 escalation（DSDM で定められている意志決定の変更プロセス）を要求する。
DSDM-12	プロジェクトライフサイクル全体で統合されたテストが期待される。	テストは分離したアクティビティではない。システム開発がビジネスの方向性として正しいことを保証するためだけではなく、技術的にも正しいことが保証されるように、システムがインクリメンタルに開発されるに従って、開発者とユーザによってテストとレビューが繰り返される。DSDM の初期段階では、テストはビジネスニーズと優先順位に対するバリデーションを中心に行われる。プロジェクト後期では、システム全体の機能が正しいかどうかを検証する。
DSDM-13	すべての利害関係者間	利害関係者はビジネススタッフと開発者のみではなく、サービス提供者やリソース管理者なども含める。

	の協力が不可欠である。	
DSDM-14	プロジェクト計画	各フェーズにおいて計画を立てる。Feasibility Study フェーズでは Outline Plan、Business Study フェーズでは Development Plan、また、各タイムボックスではタイムボックスプランを立て、Funcational Model Iteration フェーズでは Implementation Plan を立てる。
DSDM-15	リスク管理	リスクログに基づいてリスクを管理する。
DSDM-16	DSDM プロジェクトの計測	下記の目的のために計測する： ・将来何が起こるかを予測するための基準を確立する ・プロセスが成功して機能していることを証明する ・問題を明らかにするために、プロセス自体を調査する まず計測するものを決定する必要がある。プロジェクトごとに異なるだろう。 計測するものの例として下記のものが挙げられる： ・サイズ（LOC、ビジネス機能の数と複雑度、入力と出力の数と複雑度など） ・成果（人月ではなく、成果物とタイムボックスで計測する） ・欠陥（重要度とタイプ別に記録する。重要度は高中低の三段階がよく用いられる。タイプとしては、たとえば、機能・データ・内部インタフェース・外部インタフェース・非機能、が挙げられる。）
DSDM-17	見積もり	下記の目的のために見積もる： ・コストと利益を評価することにより、プロジェクトの意義を査定する ・プロジェクトの計画、スケジューリング、管理のために利用する
DSDM-18	品質管理	品質を保証するために奨励されていること： ・ワークショップの開催 ・ユーザの継続的な関与 ・レビューの実施 ・テストの実施 ・構成管理 その他考慮すべきこと： ・ユーザに提供される成果物と、品質に関連した活動を特定する ・各成果物の品質をどのようにチェックすべきかを特定する（レビューやテストなど）

		・品質チェックをいつ行うか、それが必須か任意か、ある成果物の全てをチェックすべきなのかサンプルで良いのか、開発中にチェックすべきか終了時のみで良いのかを特定する ・各成果物について誰がチェックするのか、欠陥が発見された際に誰が責任を持つのかを特定する ・品質チェックのために用いる指標を特定する ・どの標準を成果物に適用すべきかを特定する（コーディング基準や UI スタイル標準など）
DSDM-19	実現可能性調査	実施すること： ・ビジネス要件を技術的に実現可能かどうかを確認する ・DSDM を適用するのが良いかどうかを確認する ・技術的ソリューションの候補を洗い出す ・一番大まかな時間とコストの見積もりを行う 関係者： ・ビジネス分析者、エンドユーザ、技術者、ユーザ側の上級管理者 成果物： ・実現可能性確認報告書、実現可能性確認プロトタイプ（オプション）、計画概要書
DSDM-20	ビジネス調査	実施すること： ・システムがサポートするビジネス・プロセスの範囲を確定する ・これからの開発の概要をどのようなプロトタイプを作るかによって確定する ・プロトタイピングにおけるユーザの代表を確定する ・要件に優先順位を付ける ・DSDM の適用の可否について再確認する ・今後の開発の技術的な基盤をもっと明確にする ・非機能的な要件を確定する 成果物： ・ビジネス分野定義書 ・優先度の付いた要件の一覧 ・システムアーキテクチャ定義書 ・プロトタイプ概要計画書
DSDM-21	機能モデルイテレーション	実施すること： ・実際に動作するプロトタイプと静的モデルを含む機能モデルを用いて，要求されている機能性をデモする ・このプロトタイプではデモされなかった非機能的要件を記

		録する. 成果物： ・機能モデル、機能プロトタイプ、非機能的要件の一覧、機能モデルのレビュー記録、実装戦略、開発リスク分析報告書
DSDM-22	設計･構築モデルイテレーション	実施すること： ･機能プロトタイプを洗練して，非機能的要件を満たすようにすること ・ユーザ要件を満たすデモができるようにアプリケーションを作ること 成果物： ・設計プロトタイプ、設計プロトタイプのレビュー記録、テストされたシステム、テスト記録
Evo-1	利害関係者を見つける	内部/外部の利害関係者、好意ある/敵対する利害関係者、システムのライフサイクル全体を通した利害関係者を見つける。
Evo-2	重点要求トップ 10 を定義する	早い段階で、概要レベルの要求を（Planguage で）定義する。Planguage は、システムの品質特性を明確に評価可能な形で定義するための記述方法。
Evo-3	機能仕様を定義する	少なくとも次のイテレーションで実現する機能については明確に定義する。必須ではないが、Planguage を使って機能要求仕様を記述しても良い。
Evo-4	パフォーマンス仕様を定義する	システムパフォーマンスとは、システムがどの程度うまく動くか、どんな利益があるか、環境にどう影響するか、などである。パフォーマンス仕様をインクリメンタルに記述し、改良していく。 パフォーマンス属性は機能に付随する。具体的には次に 3 種類に分類される。 ・品質 - どれだけうまく動くか（信頼性、使いやすさなど） ・作業能力 ・リソース節約
Evo-5	明確で(可能なら)測定可能な仕様を定義する	仕様を記述するときは、誤解や曖昧さをできるだけ排除できるやり方や言語を使う。要求の詳細が少なすぎたり多すぎたりしないようバランスを取ることが推奨される。次の短いイテレーションで実装する主な仕様は、明確で詳細に記述する必要がある。
Evo-6	Planguage で仕様を記述する	Planguage は、Evo で要求および設計の仕様を記述するための構造化言語。必須ではないが、使うよう勧められている。
Evo-7	進化型プロ	・利害関係者に対して進化型出荷を行い、実際に使ってもら

	ジェクトマネジメント	ってフィードバックを得る。 ・小刻みのイテレーション（理想的には隔週、あるいはプロジェクト全体の予算および期間の 2-5%）。 ・単位コストあたりの品質要求が最も高いステップに、最も高い優先順位をつける。 ・既存システムを最初の基準として使うとよい。 ・フィードバックをもとに今後の計画と要求を修正する。適応型で計画を立て、仕様を進化させる。 ・システム全体としてのアプローチ。役立つことは何でもする。 ・早い段階で結果を出すことを重視する。
Evo-8	進化型出荷	早い段階で部分的なソリューションを出荷して実際に使ってもらう。出荷が行われる頻度は、一般的に週に 1 度であり、厳密には、期間および予算の 2-5%ごとである。
Evo-9	出荷したソリューションの影響を測定する	出荷したソリューションの結果を記録、分析し、今後の計画の進め方を示す。
Evo-10	設計仕様を定義する	Planguage を使って設計仕様を記述し、要求仕様と共にインクリメンタルに進化。
Evo-11	影響評価	設計アイディアによる効果が、コストやパフォーマンス要求（品質、作業能力、リソース制約）に見合っているかどうかを、数値的に分析・比較する。
Evo-12	設計アイディアがどう要求を満たすかを記述する	設計により要求が実現される理由と実現割合を明確に定義する。
Evo-13	テスト測定基準を要求仕様に記述する	パフォーマンス分析時に、そのパフォーマンス属性に対する計器をパフォーマンス要求仕様中に定義。
Evo-14	早期のインスペクションによる仕様品質のコントロール	ルール集に記述された内容や「チェック担当者」が照らし合わせるチェックリストに反していないかを探す。 ・読み手にとってあいまいなものなく明確でなければならない。 ・パフォーマンス要求およびコスト要求には、測定基準を指定して概念を定義しなければならない。 ・インスペクションには 2-5 人のチェック担当者が参加する。 ・インスペクションは仕様中の数ページをサンプリングして行う。ドキュメント全体をチェックするのではない。

		・チェック担当者が作成者に対して修正方法を助言することはない。問題点を注意するだけである。
Evo-15	仕様の関連	Planguage 仕様テンプレートには、要求の追跡可能性のサポートに関連するセクションが設けられている。
Evo-16	オープンエンドのアーキテクチャ	将来変更される可能性が高いと予想される部分に柔軟性を持たせるための施策を導入。施策とはたとえば、インタフェースと実装を分離して実装の変更がインタフェースに影響を与えないようにする、等。
Evo-17	安全係数	設計の影響見積りを算出するときは、定義された安全係数を推定の影響値に掛けるべきである。デフォルトの安全係数は 2 である。
Evo-18	クライアント駆動型計画	次にどのステップを実施すればよか判断できなければ、主要な利害関係者に尋ねるべき。
Evo-19	価値のあることは何でも	利害関係者のために何ができるか、に集中する。
Evo-20	迅速な学習	現実的な測定を行い、迅速に学習する。
Evo-21	頻繁な出荷	利害関係者に対して、早い段階から頻繁にステップごとに本当に価値のあるものを提供する。
Evo-22	問題の分割	複雑なシステムに対して謙虚になる。単純化し、一度に少しずつ問題に対処する。
Evo-23	ユーザの権限	最終的なユーザに権限を委譲する。手法にこだわったり変に官僚的になったりせず、最終結果に焦点を合わせる。
Evo-24	測定と褒賞	測定可能な結果、つまり利害関係者にとっての費用対効果のフローをベースにチームを賞賛し褒賞を与える。
Cell-1	シェフモードとコックモードの分離	シェフモードではメニュープロダクトの開発を行い、投資する。シェフモードで開発したプロダクトを初出荷後、システムアーキテクチャを管理しやすい構造に変換してプロダクトライン化し、コックモードへ移行する。
Cell-2	DSM を用いたアーキテクチャ変換	DSM により設計依存性を可視化し、いくつかのインテグラルアーキテクチャを容認したモジュラアーキテクチャへ変換する。
Cell-3	資産の活用	プロダクト、プロセス、人の領域知識・スキルを系列化して、組織的に再利用する。
Cell-4	ビジネスとソフトウェア構築の直結	ビジネスモデルから直接コードを自動生成し、ビジネスイノベーションを計画する段階で、即時プロトタイピングを行い、ビジネスとプログラムの間を直結してシステム化する。
Cell-5	セルの構築と独立性	一つのセルの中では、与えられたモデルに関して、原則として、分析、設計実装、テスト、運用までの責務を負う。内部

		仕様や内部実装の変更は自由。セル間のインタフェースの変更は公式に行う。
Cell-6	専門技術者の育成	ある特定のセルライン（セルで必要となる知識・スキルおよびセル仕様を分類し、類型化したもの）を特定の技術者に担当させることによって技術者の専門性を育て、結果としてプロジェクトの生産性および品質を高めることができる。
Cell-7	セルのスキル割り当て	セルには必要とされるスキルを定義する。そのスキルを持った人材が動的に配置される。
Cell-8	セル内の平等責任	セルには責任者を置かない。セルのメンバは均等に責任を負い、均等に評価されるものとする。セル内部では頻繁な情報交換が求められる。
SWT-1	タイムボックス	1 週間を単位とする小さなサイクルを繰り返す。 フェーズの終わりには、バッファ用のタイムボックスを設ける。 バーンダウンチャートと組み合わせることにより、開発の進み具合を把握する。
SWT-2	タイムボックス計画	ビジネス検討終了段階で、タイムボックスの繰り返し回数や各タイムボックスで対応する大まかなタスクを計画する。
SWT-3	80%ルール	実装スピードより意思決定スピードの方が、開発全体のスピードへ及ぼす影響が大きい。 そのため、判断に必要な情報を完全にそろえて判断するのではなく、80%の精度で判断して前に進む。 判断に誤りがあった場合、企画担当、システム担当、開発担当の三者が連携してフォローする。
SWT-4	要件確認会	動作するソフトウェアを見ながら、要件の実現方法や追加・修正等を判断する。 定期的（毎週）に実施する。
SWT-5	ユーザ動作確認	企画担当者およびシステム担当者は、各タイムボックスの中で、開発されたソフトウェアの動作確認を随時実施する。
SWT-6	ピックアップレビュー	繰り返しが 1、2 度終了した段階で、ソースコードをいくつかピックアップしレビューを実施することにより、品質のバラつきをなくす。
ASD-1	ミッションを明らかにする	ミッションを構成する内容を理解する。
ASD-2	ミッション	ミッションを具体的な文章として表現し、文書化する。

	を文書化する	
ASD-3	ミッションが表す価値を共有する	チームメンバ内でミッションについての共通の価値観を醸成する。
ASD-4	結果に焦点を合わせる	結果を定義し、結果を測定する品質基準を定義し、方向に関する共通の理解を作り上げる。さらに、結果を創発的に進化させる。
ASD-5	適応型サイクルの導入	適応型サイクルの特徴 ・ミッション駆動 ・コンポーネントベース ・イテレーション型 ・タイムボックス ・リスク駆動での変化の受入れ
ASD-6	JAD 開発	短期間で高品質の成果物を作成するためのクライアントの意思決定担当者と IT スタッフからなる組織的で効率を高める。
ASD-7	顧客のフォーカスグループレビュー	製品/システムそのものに対して、顧客からレビューを受けることにより、顧客の変更要求を明らかにする。
ASD-8	ソフトウェアインスペクション	インスペクションによって欠陥を発見するとともに、チームとしてのコラボレーション能力を高める。
ASD-9	プロジェクトの事後評価	実施したプロジェクトの結果を振り返り、その内容を次のプロジェクトに活かすために学習する。
EUP-1	反復的に開発する	反復による開発を行うことで、プロジェクトは優先順位をもとにリスクに対処することができる。また、反復は期間が固定され、具体的な目標が決められているため、進捗を絶えず測定することができる。反復それぞれの最後にはプロジェクトの進み具合が知らされるため、利害関係者は、動くコードの実際の進捗をもとに、プロジェクトの残りの部分に対する現実的な予想を立てることができる。
EUP-2	要求を管理する	利害関係者のニーズを満たすシステムを納品するための鍵となるのは、システムに対する要求を明らかにし、管理することである。そのためには、要求を収集・文書化・保守し、変更を系統的に組み込み、場合によっては要求から設計へと追跡する必要がある。要求管理のプロセスの中には、非常にきっちりと定義されていて、たいていはかなりの工数や費用がかかるけれども、決定事項について正確かつ詳細な文書を作成できる、といったものもあれば、インデックスカードを使

		ったXPの計画ゲームのようにシンプルなものもある。この 2 つは両極端であるが、その間に位置するものもあります。RUP はプロジェクトの厳密なニーズに合わせて仕立てることが可能である。
EUP-3	実証されたアーキテクチャ	構築対象のシステムに適したアーキテクチャを洗い出し、それからプロトタイプを作って実証する。
EUP-4	モデリング	モデリングをすることで、概念を描いてじっくり考えることができ、それを他の人と共有することができる。モデリングは、高性能な GUI ベースのソフトウェアアプリケーションを使って行うこともできれば、単にホワイトボードに絵を描いて行うこともできる。
EUP-5	継続的に品質を検証する	品質を保証するとは、ソフトウェアが要求を満たしていると確認するためにテストすることだけではない。利害関係者と一緒に、要求や設計やモックアップ（mockup）をレビューすることも、継続的な品質の検証に含まれる。早い段階でテストをして不具合を見つける方が、最後の段階で広範囲のテストをするより、ずっと効率的である。
EUP-6	変更を管理する	ソフトウェア開発において変更は当然起きることである。プロジェクトを円滑に進め、競合に対して優位に立てる可能性のある変更を利用するには、変更を予期し、適切に対処しなければならない。変更が起きると、文書、モデル、計画、テスト、コードなど、さまざまな成果物に影響が及びかねない。アジャイル主義者が言うように、影響を最低限に抑えるには、できるだけ身軽な旅をするとよいであろう。
EUP-7	協調的な開発	システムは人がチームとして開発するものであり、参加する人が効率的に協力しなければプロジェクト全体が失敗する危険がある。アジャイルソフトウェア開発コミュニティでは、協調的な開発をするための手法を明らかにし、促進するために、大きな努力をしてきている。主な手法には、ペアプログラミング（2 人のメンバが一緒にコードを作成する）、他の人と一緒にモデリングする（複数の人が別々に作業をするより、一緒にモデリングをした方が効果的だという考え方を促進する）、利害関係者の積極的な参加（プロジェクトの利害関係者は、タイミングよく情報の提供や決断を行ったり、開発作業自体に参加するべきだという考え方を促進する）などがある。
EUP-8	開発後のことを考慮する	システムは、構築して稼動状態に導入するだけでなく、導入した後で運用やサポートをする必要がある。優れた開発者はこれを分かっていて、運用やサポートの専門家のニーズが満たされるよう、協調して仕事を進める必要があることを理解

		している。さらに、システムが組織全体の中にうまく納まること、つまり、システムが全社共通の標準を反映していることを確認する必要があることを十分に理解している。そのため、彼らの作るシステムは、既存のアーキテクチャやインフラストラクチャを利用し、決められた全社的なガイダンス（標準や指針）に沿ったものになる。
EUP-9	動くソフトウェアを定期的に納品する	ソフトウェア開発プロジェクトが成功したかどうかを判断する第 1 の基準は、ユーザのニーズを満たす動くソフトウェアの納品であるべきである。効果的に開発を行うには、動くソフトウェアを少しずつ納品すること、残っている機能の中でもっとも優先順位の高いものを反復ごとに納品することである。
EUP-10	リスクを管理する	効果的なプロジェクトチームは、リスクを明らかにし、発生したものを管理しようと努力する。必要に応じて、リスクを完全に軽減する場合もあれば、考え得る影響を少なくする場合もある。
Uni-1	ユーザ一体型開発	業務とシステムは表裏一体であり、業務を理解している人はシステムを語り、コンピュータを理解するように努め、開発者は、企業の言葉を身につけ、業務とシステムを理解するように努める。
Uni-2	ペアプログラミング	先輩と後輩、修行型で仕事を進める。
Uni-3	コミュニケーション技術	報告は、簡潔に、指示者にまっすぐに、会えば必ず、結論→理由→経緯の順に実施。連絡は誰に知らせるべきかを考え、10 分考えて分からないことは必ず人に聞く。最後に、確認では自分の言葉で言いなおし、相手の了解を得る。
Uni-4	非分業の原則	自分ですべてできるセル開発方式。メンバ全員が自律的に仕事をして、チームワークと分業を体感。自分でやり失敗することで学ぶ。
Uni-5	作り捨てとコピーライト	自分が作ったものへのこだわりを捨てる一方、自分が作ったものの責任は取る（諦観と責任）。人の書いたプログラムをコピーしたら、すべての責任はコピーした人にある（新コピーライト）。
Uni-6	ワンプログラム・ワンフロー	データはファイルでわけ、IF やフラグ、区分を使わないようにする。インクルード処理を禁止し、各スクリプト内にすべて記述する。
Uni-7	ドキュメンテーション	①ネットワークやハードウェア全体図は必須。②タイミングチャート、データ配置図、バッチ処理概要等は必要に応じて書く。③システム全体を理解するための資料は必須ではない。④その仕様書さえ見ればプログラムが書けるような「詳細仕様書」は不要。

RAD-1	コミュニケーション	構造化手法やプロトタイピングを活用し、ビジネス問題とソフトウェアで扱う情報の特徴を理解する。
RAD-2	計画立案	複数のソフトウェアチームが異なるシステム機能に対して並行して作業を行う計画を立案。最も重要なプラクティス。
RAD-3	モデリング	３つの主要な側面（①ビジネスモデリング、②データモデリング、③プロセスモデリング）からなり、RAD の構築アクティビティの土台となる設計表現を確立。
RAD-4	構築	既存のソフトウェアコンポーネント、及び自動コード生成アプリケーションの利用を重視。
RAD-5	展開	必要に応じて次回以降の反復の基礎を確立。

２ 海外統計データ

ここでは、VERSIONONE によるアジャイル開発に関する動向調査（第 3 回、2008 年度版）から見える傾向を以下に示す。

この調査の回答者は何らかの形でアジャイルに関わっていると見られる 2319 件（総回答数は 80 カ国から 3061 件）の回答に基づいたものである。2008 年の 6 月から 7 月にかけて実施されたものである。2008 年度が第 3 回であったが、回答者のうち 70%弱は、2006 年及び 2007 年の同じ調査には回答していない新規の回答者である。

(1) アジャイル手法の導入理由

アジャイル手法の導入の理由としては、いわゆる「Time-to-Market の加速」と「要件等の変化（特に、優先順位）へ対応」することが目的としては高い。

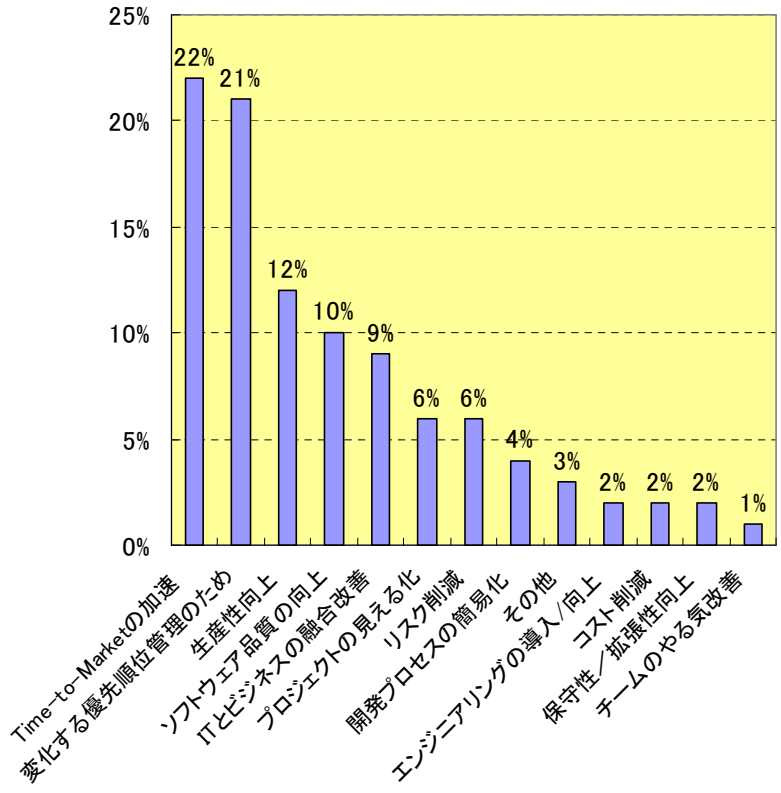


図 11　アジャイル手法の導入理由

(2) アジャイル手法の導入に当たっての気掛かりの理由

一方、導入に当たっての気掛かりとしては、「事前計画の欠如」、「管理コントロールの欠如」、「文書の欠如」、「予測性の欠如」が 35%から 40%の回答で見られる。

これは、一般に言われているアジャイル手法に対する気掛かりと一致しており、一般にそう思われている様子が分かる。

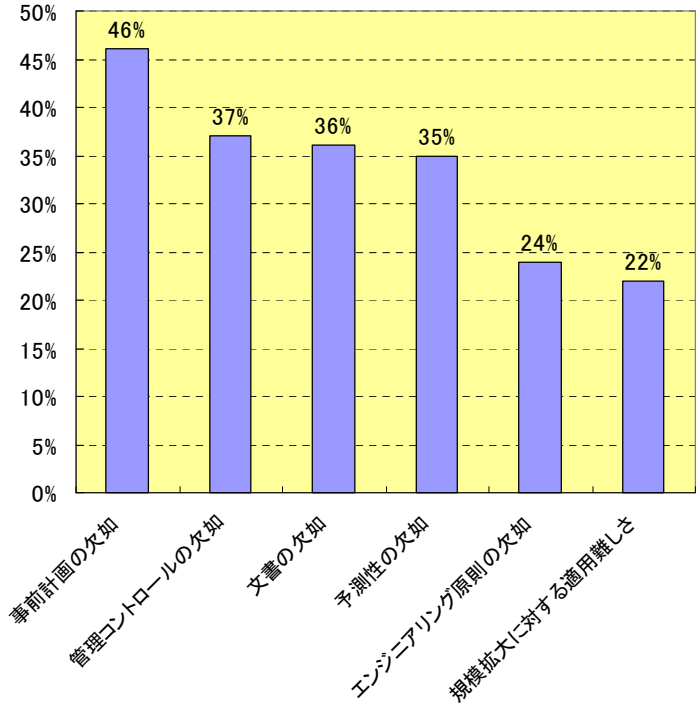


図 12　気掛かりの理由

(3) アジャイル手法の推進者

アジャイル手法の推進者、特に最初に導入等を提唱した役職としては、開発部門の長が 4 分の 1 くらいで、あとはプロジェクトマネージャ、開発マネージャ、CIO/CTO がほぼ同じ割合となっている。

基本的に開発の責任者が中心となって、導入を進めている様子が伺える。

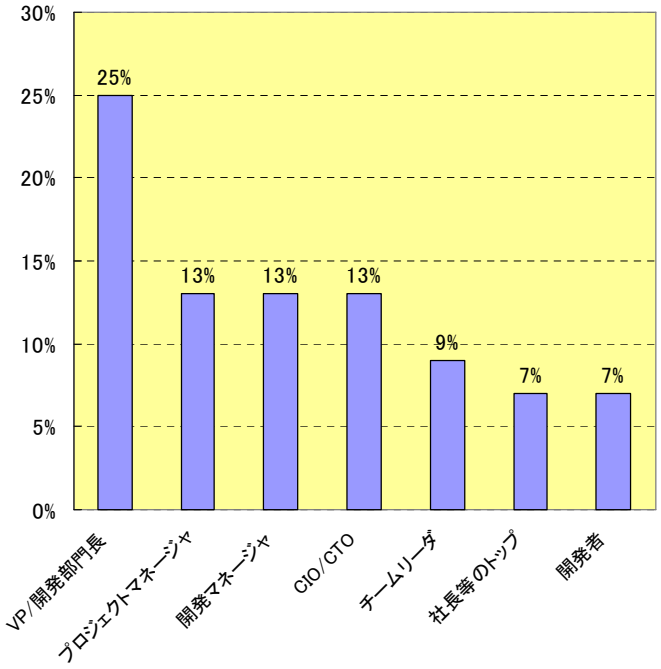


図 13　推進者

(4) 活用しているアジャイル手法

活用している手法は、約半数が SCRUM と他の手法を圧倒している。続いて、SCRUM と XP の組合せで、XP 単体での活用が続く。ここまでで、ほぼ 8 割はカバーされている。SCRUM は、スクラム･マスターなどの人材育成などの制度面での整備を行っており、これらの効果が上がっていると見ることができる。

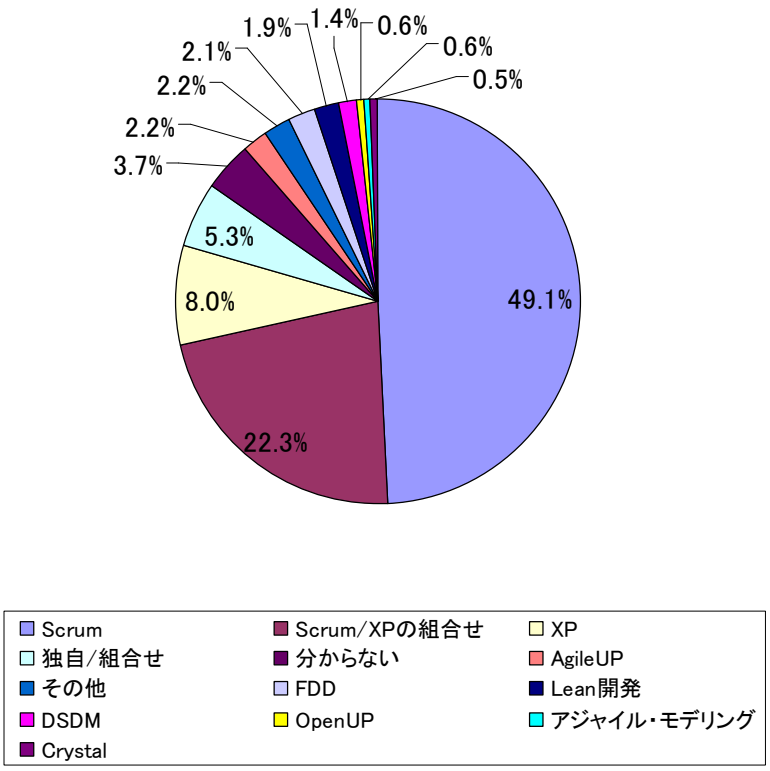


図 14　アジャイル手法の種類

(5) アジャイル手法の更なる活用における障壁

アジャイル手法を更に拡大していく上での障壁として、変化への対応が上位を占めている。特に、組織文化の変化を実現できるか否かが最も高い障壁となっている。2 番目は、一般的な変化に対する抵抗感である。アジャイル的な手法を採用することは、大きな変化をもたらすものであるとの様子が伺える。また、変化が必要であることがわかる。

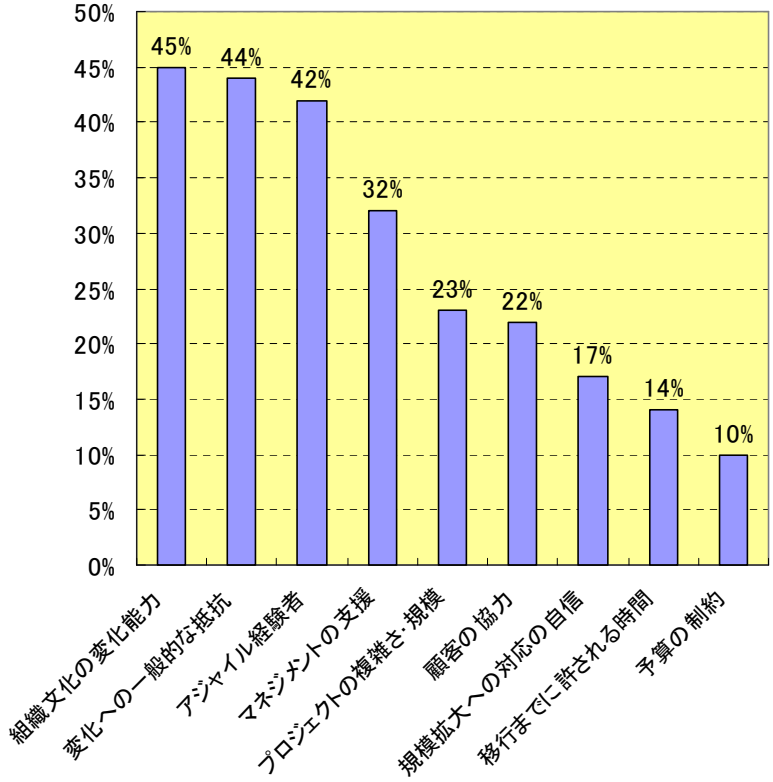


図 15　アジャイル手法の導入拡大の障壁

(6) アジャイル型プロジェクトの失敗理由

アジャイル型プロジェクトの失敗理由としては、企業の哲学（方針）と文化とマッチしなかったときに起こりやすいと考えられている。前号の組織文化がトップになっているのと軌を一にしている。また、手法への不慣れもほぼ同じ程度に理由として上げられている。既に指摘したとおり、アジャイル的なアプローチでの成功のためには、様々な前提条件やプラクティスを実践する必要があり、その内容を把握しており、かつ、慣れていないとなかなか効果を出すには難しい様子が伺える。

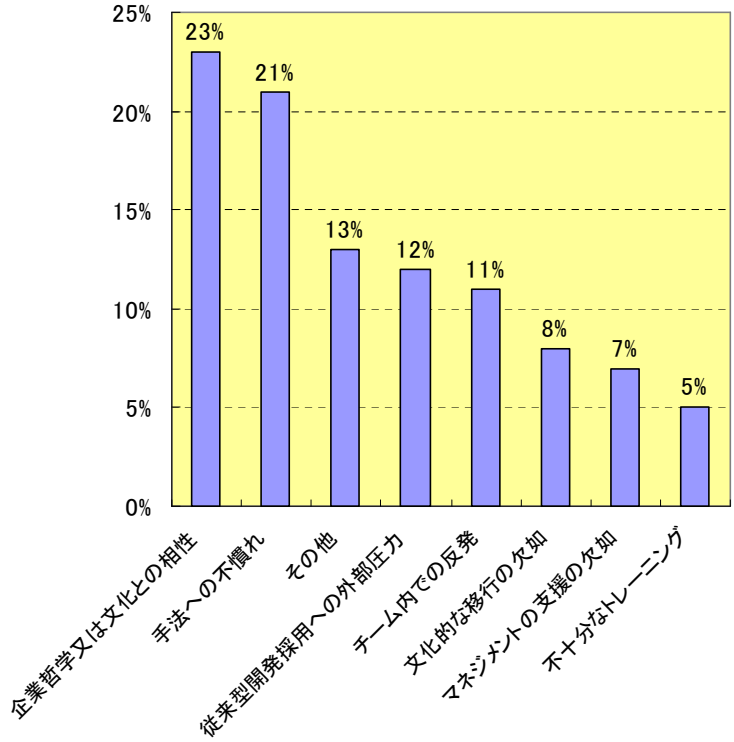


図 16　アジャイル型プロジェクトの失敗理由

(7) 活用されているプラクティス

8 割を超えて活用されているプラクティスが、「反復型計画」である。続いて 7 割を超えているのが、「ユニットテスト」、「デイリー・スタンダップ」、「リリース計画」である。

6 割を超えているのが、「継続的インテグレーション」、「自動ビルド」、「ハーンダウンチャート」である。ここまでで、およそ、計画及び進捗の確認のためのプラクティスとともに、構築及びテストの基本的なプラクティスが見えている。5 割以上で見ると、49%の「テスト駆動型開発」を含めて、「リファクタリング」、「レトロスペクティブ」、「コーディング作法」「スピード」とやはり、進捗と繰り返しの中での次への対策、繰り返しでもデグレを起こさせないための手法が活用されている様子が伺える。

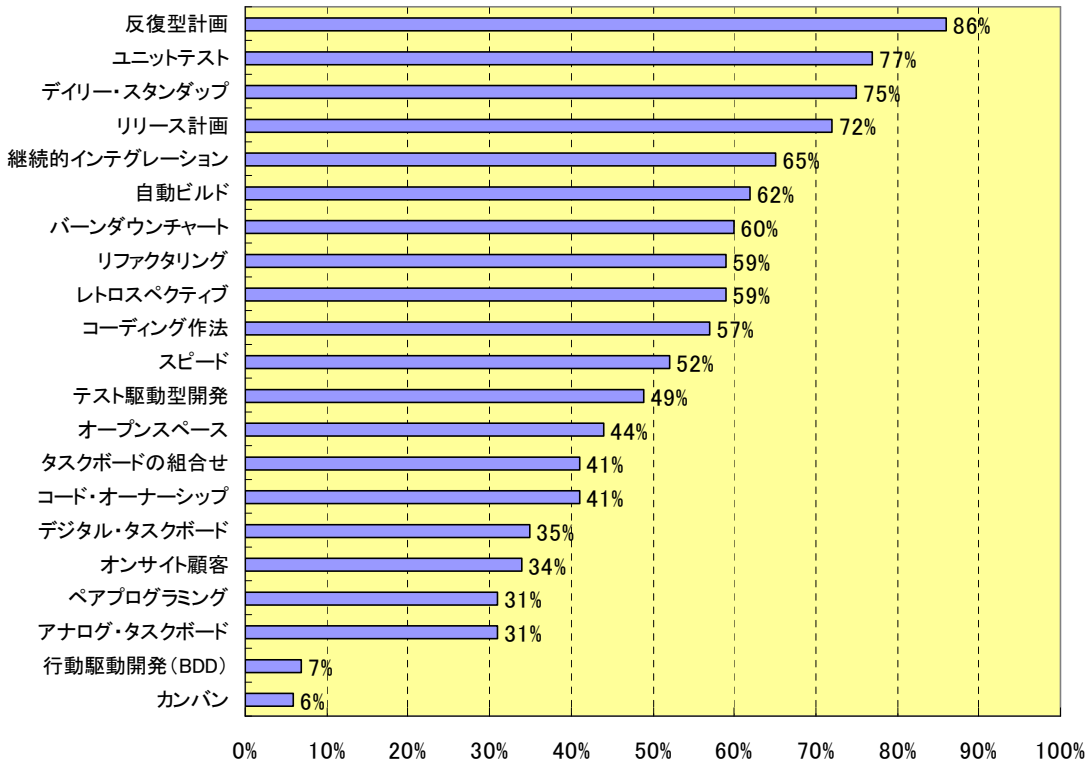


図 17　活用されているプラクティス

(8) アジャイル手法の導入効果

生産性、Time-to-Market の加速、ソフトウェア不具合の削減で、10%以上の効果があるとする回答が 50%を超えている。Time-to-Market は、導入理由として最も高いものであり、実際の効果も上がっていることが分かる。一方、生産性向上や品質向上（不具合の削減）は、一般にはややアジャイル的な開発では期待されないとの意見が聞かれるものであるが、実際には効果があるとの見解が強いと見てもよさそうである。

なお、コスト削減に対する効果は、3 割程度となっており、他の 3 つほどではない。生産性向上のコスト削減効果は、繰り返し開発の中で効果が吸収されていると見られる（逆の見方をすると、本来コスト増になるところを生産性向上で抑えていると見ることができる）。

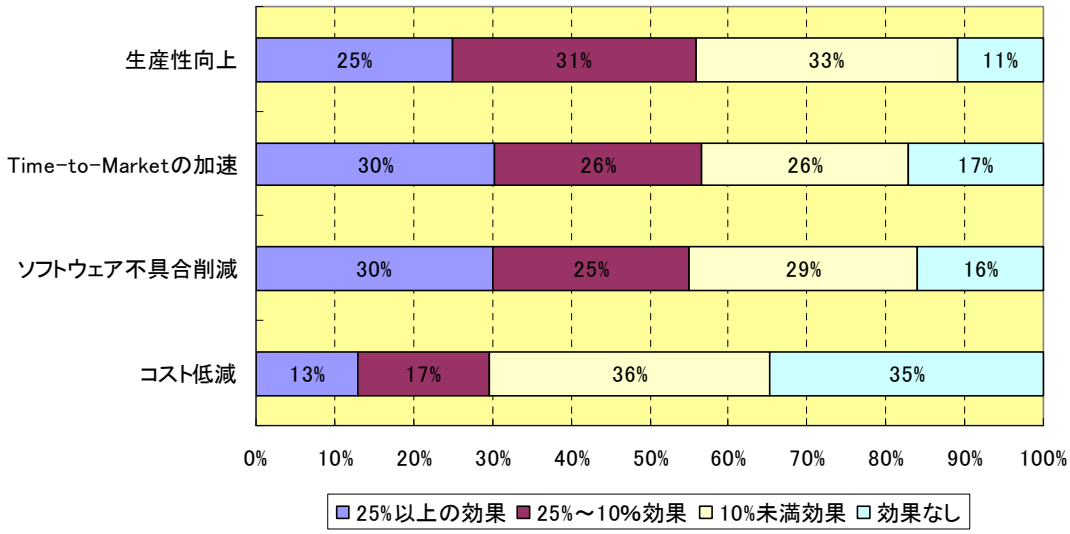


図 18　アジャイル手法の導入効果

以上

３ 非ウォーターフォール型開発に関する研究会活動記録

３．１ 委員構成

非ウォーターフォール型開発に関する研究会は、次の委員から構成された。

氏名		所属
松本　吉弘	座長	財団法人・京都高度技術研究所 顧問
稲村　直穂子	委員	株式会社ディー・エヌ・エー　システム統括本部本部長
大槻　繁	委員	株式会社一 副社長
合田　治彦	委員	富士通株式会社 システム生産技術本部長代理
田澤　久	委員	楽天株式会社 開発部開発生産性強化グループ グループマネージャー
羽生田　栄一	委員	株式会社豆蔵 取締役
平鍋　健児	委員	株式会社永和システムマネジメント 副社長、 株式会社チェンジビジョン 代表取締役
広瀬　敏久	委員	日本電気株式会社 主席技術主幹
前川　徹	委員	サイバー大学　IT 総合学部 教授
馬嶋　宏	委員	株式会社日立製作所 ソフトウェア事業部企画本部統括部長
松島　桂樹	委員	武蔵大学 経済学部 教授
南　悦郎	委員	新日鉄ソリューションズ株式会社 技術本部システム研究開発センター所長

３．２ 研究会開催記録

非ウォーターフォール型開発に関する研究会は次のとおり全 5 回開催された。

開催回数	開催日程
第 1 回研究会	11 月 26 日
第 2 回研究会	12 月 22 日
第 3 回研究会	1 月 19 日
第 4 回研究会	2 月 16 日
第 5 回研究会	3 月 5 日

なお、研究会の活動内容は、研究会報告書を参照のこと。