アジャイル型開発における
プラクティス活用事例調査

調査報告書

～ガイド編～

平成 25 年 3 月 19 日

独立行政法人情報処理推進機構

技術本部 ソフトウェア・エンジニアリング・センター

はじめに

独立行政法人情報処理推進機構 技術本部 ソフトウェア・エンジニアリング・センターでは、「日本のソフトウェア産業の競争力を強化すること」、「エンジニア 1 人ひとりが生き生きと働ける環境を作ること」を目指すべきゴールとして、「日本国内における非ウォーターフォール型開発の普及促進」と「わが国のソフトウェア産業の特性に照らした非ウォーターフォール型開発の課題解決」を目標に掲げ、平成 21 年度から調査検討に取り組んで参りました。その中で、開発手法の実践的な解説書と実例やテクニック等の工夫を取りまとめた Tips 集を、広く一般に公開することか求められていることが分かりました。本事業では、アジャイル型開発におけるプラクティス活用の事例調査を実施し、調査結果を報告書として取りまとめました。

報告書は以下の 2 部構成となっており、本報告書は「Ⅱ部 ガイド編」にあたります。

Ⅰ部 調査編

対象読者を今後の調査業務にあたる人物に設定して、調査全体の概要（調査の指針や手順）を簡潔にまとめました。

Ⅱ部 ガイド編

対象読者を「アジャイル開発に関心を持ち始め、実際に自らで試してみようと思っている」初心者層に設定し、事例調査から得られた知見を元に、利用者の利用目的や課題解決等に役立つようプラクティスのレファレンスをまとめました。

本調査事業は、「アジャイル型開発におけるプラクティス活用事例調査事業」として、株式会社永和システムマネジメントに委託、実施致しました。

アジャイル型開発におけるプラクティス活用事例調査

【調査報告書　ガイド】

独立行政法人情報処理推進機構

目次

1. 目的

非ウォーターフォール型開発の代表的な手法であるアジャイル型開発を実際に適用した国内外のプロジェクトにおける「プラクティス」（チーム運営からプログラミングまでの幅広い手法、活動など）の具体的な活用事例を幅広く収集し、開発対象ソフトウェアの特徴や開発組織・プロジェクトの状況（以下「コンテキスト」とする）の違いの観点で分類・整理した上で、ソフトウェア開発の現場で利用できるレファレンスのためのガイドとして取りまとめた。

海外書籍の翻訳は既にいくつか発行されている。本ガイドは、日本の現場での実践事例を広く紹介する（特に、プロジェクト独自の工夫や日本国内でアジャイル型開発を実践する際に留意しなければならない点を紹介する）ことで、国内でのアジャイル型開発のより広い普及を目指す。

メインターゲットの読者層を「アジャイル型開発に関心を持ち始め、実際に自分で試してみようと思っている」初心者に置いている。

「アジャイル型開発におけるプラクティス活用事例調査 ～調査編～」の内容に基づいて調査した結果から得られた知見を元に複数ある調査対象事例の共通部分をパターン・ランゲージの記述方法を参考としてまとめ、読者の利便性を高めた。

2. 使用方法

本ガイドでは、アジャイル型開発を実践している企業への調査結果として得られた知見を元に、パターン・ランゲージの記述方法[1]を参考にしてプラクティスをまとめている。[Meszaros et al 1996]

パターン・ランゲージの記述方法を採用した理由は次の通りである。

（1）読者がプラクティスの説明を見る際には「何をするか」に着目してしまう。そのため、本来は「解決したい問題」があり「期待する効果」を求めて適用されるべきものが、深く考えずにプラクティスを使うことが目的になりがちである。

（2）プラクティスは使用する状況にも左右される。プラクティスを使うべき状況でない時に使ってしまうことで、本来の効果が発揮できないどころか逆効果にもなってしまうこともある。

このような点を考慮して、単なる解決策の説明に留まらないプラクティス解説にするよう留意した。

2.1. プラクティス解説の見方

プラクティス名
　プラクティスの日本語名／英語名
別名
　プラクティスの別名
要約
　プラクティスの概要説明
状況
　解決すべき問題が発生する状況
問題
　特定の状況下で発生する問題
フォース
　問題解決策を選択する上で、かぎとなる考慮点・制約事項
解決策
　プラクティスそのものの説明。具体的な工夫も解説

[1] http://hillside.net/index.php/a-pattern-language-for-pattern-writing

留意点

以下を解説：

- プラクティスを適用する上でコンテキストに応じた留意点
- プラクティスをそのまま適用できない場合の代替策
- うまくいかない場合の注意点

効果

プラクティスを活用した時に得られる効果

利用例

調査先企業への調査から得られたプラクティスの利用例

関連プラクティス

関連性の高いプラクティス

参考文献

プラクティスが紹介されている文献

2.2. **本ガイドの使い方**

興味のあるカテゴリのプラクティスを知る

本ガイドでは、一般的なカテゴリでプラクティスを大きく 3 つに分類している。興味のあるカテゴリのプラクティスを順番に読み進めることにより、プラクティスの利用例から自分たちの組織やプロジェクトに合わせた活用方法が理解できるようにした。

「3.プラクティス解説」を参照のこと。

プラクティス適用の際に発生している問題への対応策を探す

アジャイル型開発を実践している企業に調査を行った中で頻出したプラクティス適用時の「よくある問題」と、その問題に対しての効果的なプラクティスや対応策を探し出せるようにしている。ここで言う「よくある問題」とは、一般的に言われているものではなく、調査結果から導きだしたものである。

「3.5.プラクティス適用時のよくある問題と対応策」を参照のこと。

特定の事例で活用されているプラクティスを探す

本ガイドでは活用事例を紹介している。自分たちの組織やプロジェクトの状況（コンテキスト）に近い事例を探し出し、その事例の中でどのようなプラクティスが使われているか、そして、それらのプラクティスにどのような工夫が加えられているかを理解できるようにした。

「4.活用事例」を参照のこと。

読者の組織、プロジェクトの状況に対してのお勧めプラクティスを探す

本ガイドでは調査結果を元に、特定の状況に対してのお勧めプラクティス群を、9 つのケースに分類してまとめている。あくまでもスタート地点に過ぎないが、自分たちの組織、プロジェクトの状況（コンテキスト）に似たケースを参考に、お薦めのプラクティス群を参考にしてほしい。

「5.活用のポイント」 を参照のこと。

プラクティス群活用に際しての留意点（アンチパターン等）を知る

プラクティスが上手くいかないケースも状況（コンテキスト）によってはかなりの確率で存在する。また、プラクティス自体は上手くいっているが、そこからさらに新しく問題を引き起こす、という場面もある。そういった事例を留意点（アンチパターン）として取り上げた。

「3.プラクティス解説」の各プラクティスの留意点を参照してほしい。

参考文献を探す

各プラクティスには、元となる詳細な参考文献が存在する。利便性を考え、特定のプラクティスについてのみ解決してある参考文献は、各プラクティスの紹介ページ末に列挙した。

複数のプラクティスを扱っている参考文献については、巻末の参考文献に列挙した。読者の必要に応じて参照されたい。

3. プラクティス解説

本章では、プラクティスを以下の 3 つのカテゴリに分けて紹介する。

1. プロセス・プロダクト
　アジャイル型開発における反復漸進開発のプロセスを構成するプラクティス群及びプロダクトの価値の向上、判断を支援するプラクティス群。

2. 技術・ツール
　ソフトウェアシステムの設計、開発、保守を支援するプラクティス群。効率を高めるツールなども含む。

3. チーム運営・組織・チーム環境
　アジャイルチームの運営、コミュニケーション、組織展開といった部分について役立つプラクティス群。

プラクティスの一覧を下記に示す。

カテゴリ	サブカテゴリ	日本語名	英語名
プロセス・プロダクト	プロセス	リリース計画ミーティング	Release planning to release product increments
		イテレーション計画ミーティング	The Planning Game
		イテレーション	Iteration
		プランニングポーカー	Planning poker to estimate tasks during sprint planning
		ベロシティ計測	The project velocity is measured
		日次ミーティング	Short daily meeting to resolve current issues
		ふりかえり	end-of-iteration retrospectives / print retrospective to learn from previous sprint
		かんばん	Kanban
		スプリントレビュー	Sprint review meeting to present completed work
		タスクボード（タスクカード)	Task board (Task card)
		バーンダウンチャート	Burn down chart to monitor sprint progress
		柔軟なプロセス	Flexible process
	プロダクト	ユーザーストーリー	User stories are written
		スプリントバックログ	Mutual commitment to sprint backlog between product owner and team
		インセプションデッキ	Inception Deck
		プロダクトバックログ（優先順位付け）	Priorities (product backlog) maintained by a dedicated role (product owner)
	フィードバック	迅速なフィードバック	Rapid feedback

表 1 プラクティスとカテゴリ一覧(1)

カテゴリ	サブカテゴリ	日本語名	英語名
技術・ツール	設計開発	ペアプログラミング	All production code is pair programmed
		自動化された回帰テスト	Automated regression test
		テスト駆動開発	Code the unit test first
		ユニットテストの自動化	Unit Test Automation
		受入テスト	Acceptance tests are run often and the score is published
		システムメタファ	System metaphor
		スパイク・ソリューション	Create spike solutions to reduce risk
		リファクタリング	Refactor whenever and wherever possible / Constant refactoring
		シンプルデザイン	Simplicity in design
		逐次の統合	Only one pair integrates code at a time
		継続的インテグレーション	Continuous Integration / Integrate often
		集団によるオーナーシップ	Use collective ownership
		コーディング規約	Code written to agreed standards
	障害対応	バグ時の再現テスト	When a bug is found, tests are created
	利用ツール	紙・手書きツール	Paper, Handwriting

表 2 プラクティスとカテゴリ一覧(2)

カテゴリ	サブカテゴリ	日本語名	英語名
チーム運営・組織・チーム環境	人	顧客プロキシ	Customer Proxy
		オンサイト顧客	The customer is always available / Constant user interaction
		プロダクトオーナー	Product Owner
		ファシリテータ（スクラムマスター）	Development process and practices facilitated by a dedicated role (Scrum master)
		アジャイルコーチ	Agile Coach
		自己組織化チーム	Team members volunteer for tasks (self-organizing team)
		ニコニコカレンダー	Niko-niko Calendar / Smiley Calendar
	進め方	持続可能なペース	Set a sustainable pace
	組織導入	組織に合わせたアジャイルスタイル	Right agile style for their organization
	ファシリティ、ワークスペース	共通の部屋	Give the team a dedicated open work space
		チーム全体が一つに	One whole team
		人材のローテーション	Move people around
		インテグレーション専用マシン	Set up a dedicated integration computer

表 3 プラクティスとカテゴリ一覧(3)

3.1. プロセス・プロダクト

3.1.1. リリース計画ミーティング / Release planning to release product increments

“リリース計画はプロジェクトの道しるべ”

別名: なし

要約

いつどの機能をリリースするかについての計画を立てるミーティングを開く。その結果、チームとプロダクトの関係者が共通の目標を持つことができる。

状況

一定期間のサイクルで繰り返し（イテレーション）開発したいと考えている。

プロダクトとして何を用意するべきかをリストアップしたプロダクトバックログが準備されており、内容を開発者とプロダクトオーナーの間で合意している。

問題

いつどのような価値をユーザーに届けるのかという目標がなければ、ただイテレーションを繰り返しているだけになってしまう。

リリース予定の機能に対する優先順位付けを行わなければ、ユーザーにとって最も価値のある機能からリリースすることができない。

プロダクトを提供する組織として、リリースによるユーザーからのフィードバックを踏まえたプロダクト戦略を立てることができない。

フォース

◆プロジェクト早期にリリース計画を立てたいが、詳細なタスクまでを洗い出して計画するには、タスクへの分割やタスクの実施順序の決定、担当者のサインアップに至るほどの詳細な計画を立てることができない。

解決策

プロダクトのリリース計画ミーティングを開発チームとプロダクトオーナーが協力して開く。

リリース計画は以下の段取りで進める。

プロジェクトのミッション（インセプションデッキの「われわれはなぜここにいるのか」）を確認する。また、スケジュール、スコープ、予算を確認し、プロジェクトの制約条件とステークホルダーの期待を明らかにしておく。

プロダクトバックログのうち、次のリリースに含めるユーザーストーリーの見積りを行う。イテレーションの長さを決める。多くのチームが 1 週間から 4 週間の間で設定している。プロダクトに関して不明なところが多い場合は、フィードバックをこまめに得られるようにするためにイテレーションの期間を短くする工夫を取る。

チームのベロシティを見積もる。見積りの方法については、関連プラクティスとしてベロシティ計測を参照されたい。

ユーザーストーリーの優先順位を付け、開発対象とするユーザーストーリーを選択する。選択したユーザーストーリーの開発規模とベロシティから、必要なイテレーション数を算出しリリース日を決める。逆に、リリース日が先に決まっている場合は、選択するユーザーストーリーがリリース日までのイテレーションに収まるかを確認する。収まらない場合は、スコープの調整を検討する。

ベロシティの変動があるため、適宜リリース計画の見直しを行うこと。見直しを行うタイミングは、イテレーション計画ミーティングが適している。

留意点

リリース計画ミーティングでは、タスクの分割や実施順番の決定、担当者のアサインまでは行わない。

リリース計画の段階では、まだユーザーストーリーの理解が進んでいない場合が多い。タスクに関する計画は、イテレーション計画ミーティングで行う。

効果

◆リリース計画によっていつどの機能を提供するかという、日付入りの目標を持つことができる。
◆利用価値の高い機能を優先してリリースすることによって、早期に投資の回収が行える。
◆プロダクトのリリース計画を踏まえた組織の戦略を立てることができる。

利用例

リリース計画ミーティングは、全体では75%の事例に適用されていた。システム種別、規模、契約のいずれの切り口別でもそれほど相違はないが、手法別で見ると、スクラムでは67%であるのに対して、XP では70～80%の適用率であった。

A 社事例(1)では、四半期に１回、ビジョンとマイルストーンを関係者へ提示している。リリース計画はビジネスの成否を評価し検討する重要な場となっていた。

B 社事例(3)では、計画段階でのコンセプト決めや、ユーザーストーリーマッピング（ユーザーの行動に即してユーザーストーリーを洗い出す手法）などに時間を割くようにしていた。最初はあえてリリース期日を決めず、まずは一定の期間でコンセプトを元に開発を行い、後から段階的にスコープや期日を明確にする方法を取っている。

H 社事例(14)では、プロジェクトの初期段階でプロダクトオーナーがリリース計画を仮決めしている。

K 社事例(20)では、全工程の３分の１をバッファとしており、開発期間の半分を過ぎても進捗が遅れているようであれば、リリース期日から逆算、リリース期日優先で線表を引くようにしている。その結果、あるリリースでは、必要な機能が入らないと判明した時点で顧客との調整を行っている。無理に機能を入れこんだがためにトラブルに発展した過去の体験を、顧客もチームも経験として活かすことができた。最終的には品質を優先する開発を行っている。

L 社事例(23)では、ユーザーストーリーマッピングによって、リリース計画の可視化を行った。

関連プラクティス

インセプションデッキ
プロダクトバックログ
ベロシティ計測
イテレーション（スプリント）
イテレーション計画ミーティング
プランニングポーカー
ユーザーストーリー
バーンダウンチャート
ユーザーストーリーマッピング

3.1.2. イテレーション計画ミーティング / Iteration planning meeting

“リリース計画で遠くを眺め、イテレーション計画で近くを見る”

別名：計画ゲーム、スプリント計画ミーティング、反復型計画

要約

イテレーションを始める際に、イテレーションで達成すべきゴールと、それを実現する作業を洗い出そう。その結果、チームが開発を進めるために必要かつ具体的な計画を立てることができる。

状況

チームはプロダクトのリリース計画を立てている。リリースまでの期間、開発を一定期間のサイクル（イテレーション）で繰り返し行いたい。

プロダクトとして何を用意するべきかをプロダクトバックログで管理している。プロダクトバックログに含まれるユーザーストーリーは、開発チームとプロダクトオーナーの間で合意している。

問題

リリース計画は中長期的な目標のため、それだけではイテレーションで何をどのように開発すれば良いかわからない。

イテレーションでの開発を進めるためには、より具体的な計画が必要である。

フォース

- ◆ユーザーストーリーは、リリース計画を作るには粒度が細かいが、実際に開発者がイテレーションの中で作業するには粒度が大きい。
- ◆ユーザーストーリーの見積り単位には、ストーリーポイントと呼ばれる作業規模を表現する独自の単位がよく使われるが、イテレーション内の作業は時間で見積りたい。
- ◆プロダクトバックログの内容は、顧客（ビジネスの問題やニーズを把握している人）の視点からの「何を作ってほしいか」に関する記述はされているが、開発者視点の「どう実現するか」については記述されていない。

解決策

イテレーションで開発するユーザーストーリーと、その完了までに必要なタスクを洗い出し、計画を立てるミーティングを開く。

イテレーション計画ミーティングは大きく 2 つの目的がある。一つは、そのイテレーションで何をするのか、プロダクトバックログからユーザーストーリーを選択することである。もう一つは、選択したユーザーストーリーを完了するために必要なタスクを洗い出し、その作業時間を見積もる。つまり、イテレーションで達成できる作業の計画を立てることにある。

タスクの洗い出しはチーム全員で実施する。やらなければいけないユーザーストーリーは、個人に割当てられるのではなく、チーム全員で責任を持つようにする。そのため、チームとして何を実現しなければいけないか、そのためにはどんな作業が必要かを全員で考えなければならない。

チームが選択するユーザーストーリーの見積り合計はイテレーションの範囲に収まる必要がある。イテレーションの範囲で収まるかどうかはベロシティ計測の結果を参考にする。もし収まらなそうな場合はユーザーストーリーを適度な大きさに分割する。

開発チームとプロダクトオーナーでイテレーションのゴールを決める。イテレーションのゴールは達成すべきミッションであり、チームにとっての指針になる。

選択したユーザーストーリーを実現するためのタスクを洗い出す。具体的な作業に落とし込むことによって必要な時間を見積もることができる。

ミーティングの時間は、イテレーション期間の 5%程度の時間とする~~を使う~~。

タスクの見積りにはプランニングポーカーが利用できる。

イテレーション計画の日々の状況確認は、日次ミーティングで行う。

留意点

開発依頼者は、チームを信じて計画を任せるべきである。

動機付けられたチームは、見積りが大きく外れるようであれば、自らその原因を分析

し、次の見積りに活かすはずである。ただし、そのためにはチームにとって達成すべきゴールが明確になっている必要がある。

プロダクトオーナーがユーザーストーリーの選択の際に技術的な観点を見過ごさないよう、開発チームは技術的な問題に関して、プロダクトオーナーに分かりやすく説明するようにする。

イテレーション計画ミーティングの時間が意図せず長時間に及んでしまう場合は、やるべきユーザーストーリーの詳細化が充分でないことや、完了条件が明確になっていないことが考えられる。

効果

- ◆イテレーションで達成できるユーザーストーリーと、それを完了するために必要なタスクと作業時間を見積もることができる。
- ◆チームで作業を洗い出すため、そのイテレーションで何をしなければいけないかを全員が共有できる。

利用例

イテレーション計画は、アジャイル型開発の根幹を成すため、システム種別、規模、契約など、どのような特性のプロジェクトであっても 90%以上の高適用率であった。逆に利用していない事例ではマイルストーンを決めるが、そこまで反復して進めるか否かは現場に任せていた。(I 社事例(15))

B 社事例(2)では、イテレーション計画ミーティングのファシリテータを持ち回りで担当することで、負担が 1 人に集中するのを避け、経験を共有することができた。
また、タスク出しはユーザーストーリー毎に分担して行った後、タスクの過不足をチーム全員で洗い出すことで計画にかかる時間を短縮していた。

C 社事例(4)では、ユーザーストーリーの選択に際して、このイテレーションの中で「確実に実現したいもの」「可能なら実現したいもの」「早く終わったら実現したいもの」という優先度を設定した。

G 社事例(9)では、イテレーション計画ミーティングの中でペーパープロトタイピングといわれる手法を用い、モデルや画面イメージを紙で作ることによって設計意図をわかりやすくして UI デザインの共有と受け入れ条件の確認を行っていた。そのため、計画にはかなりの時間を要したが、見積りの前提が具体的になったため、結果的に見積り作業時間の削減につながった。

K 社事例(20)では、当初は顧客と共にイテレーション計画ミーティングを行っていたが、時間短縮のため、開発チームが計画の素案を作り、計画レビューという形でミーティングを行うようにした。

関連プラクティス

プロダクトバックログ
スプリントバックログ
ベロシティ計測
イテレーション
ユーザーストーリー
日次ミーティング
スプリントレビュー
バーンダウンチャート

3.1.3. **イテレーション** / Iteration

“小さなプロジェクトの積み重ねで遠くへ行ける”

別名: タイムボックス、スプリント、反復

要約

プロジェクトで起こる変化（プロジェクトを開始する前の期待や想定とのギャップ）が大きく予想できない時には、一定期間を何度も繰り返して開発を進める。その結果、チームはプロダクト開発に必要な知識を学びながら、変化に適応していくことができる。

状況

新しいプロダクトの開発を始めようとしている。

問題

新しいプロダクトの開発では、プロダクト及びプロジェクトについての知識が不足している。

プロダクトについての知識とは、そのプロダクトがどんな機能を備えておくべきか、どうあるべきかについての理解である。

プロジェクトについての知識とは、プロダクトを開発するために必要な技術及びリスクは何か、集まったチームがどのように開発を進めていくべきかについての理解である。

フォース

◆チームが最初の計画を立てる段階でプロダクトやプロジェクトについての必要な知識をすべて備えていることは現実的に望めない。

解決策

1週間から4週間の一定期間で区切られたイテレーションのもとで開発を行う。

チームは、最初に計画を立てた段階からプロジェクトを進める過程において、プロダクトがどうあるべきか、またそれをどのように開発するべきかを学び続けている。直前のイテレーションで得た知識を踏まえて、新たな状況に適応していく。

イテレーションは「タイムボックス」である。タイムボックスとは時間を延長不可の箱と考え、その箱の大きさに合うだけの内容を入れるという比喩である。

イテレーションの長さは、チームに必要なフィードバックの間隔で決める。チームの習熟度が低い、または不確実性の高いプロジェクトの場合は、イテレーションの長さも短くする（1週間など）。逆に、不確実性がそれほど高くなく、チームも成熟している場合は、イテレーションは長くても構わない。

イテレーションの長さによって、その中で実現されるユーザーストーリーなど開発項目の粒度は変わる。1週間では、大きな項目は実現できないため、細かく分割する必要がある。その結果、小さく作りあげていく姿勢が身につきやすい。逆に1ヶ月だと、1週間の場合よりも大きなユーザーストーリーの計画も可能になる。

イテレーションは一つの小さなプロジェクトである。この中で必要な作業（計画、分析、設計、実装、テスト）をすべて行うが、当然、1回のイテレーションですべての要求を満たすことは容易ではないため、何度かイテレーションを繰り返す中で、少しずつ開発を進めていく。

留意点

イテレーションは長さが4週間を超えると、チームが学習し適応するためのサイクルが長くなり、フィードバックが遅れることでリスクが増大する恐れがあるため、長くなりすぎることのないよう注意する。

計画通りにいかないからといって、イテレーション開始後に期間を延ばしてはいけない。イテレーションはタイムボックスであることを意識して、作業量を減らすこと。但し、イテレーションの切れ目であれば、期間の変更を行ってもよい。

効果

◆チームはプロダクトやプロジェクトについて獲得した知識や反省点を次のイテレーションに活かして改善することができる。その結果、プロダクトがステークホルダーの期待に近づく。
◆イテレーションによる開発は、開発のリスクが影響する範囲を1イテレーションに納めることができる。それは、イテレーション毎に計画ミーティングやイテレーションの成果

に対するレビューを行うためである。

利用例

イテレーション計画はアジャイル型開発の根幹を成すため、システム種別、規模、契約など、どのような特性のプロジェクトであっても90%以上の高適用率であった。逆に利用していない事例ではマイルストーンを決めるが、そこまでは反復して進めるか否かは現場に任せていた。(I 社事例(15))

A 社事例(1)では、イテレーションを１週間としている。イテレーション中の要求変更も内容と状況によっては受け入れている。

B 社事例(2)では、イテレーションを１週間としている。当初は２週間としていたが、１週間に変更することで企画から開発の流れが速まり、開発がスムーズになった。

G 社事例(9)では、当初１ヶ月はチームの学習フィードバックサイクルを重視して１週間にしていたが、１週間だと祝日がある場合は実働４日となり、うち１日はスプリントレビューとイテレーション計画ミーティングになってしまうため状況に応じて1～2 週間サイクルに変更しながら行なった。

G 社事例(10)では、顧客とのトライアル期間(アジャイル型開発の進め方を実感してもらう期間)はイテレーションを１週間とし、トライアル期間終了後は２週間に変更した。

E 社事例(6)では、イテレーションを１週間とすることで見通しを立てやすく、ゴールを身近に感じられることから、チームメンバーが動きやすくなった。

L 社事例(21)では、イテレーションを１ヵ月としていた。プロダクトオーナーと別ロケーションでの開発であったため、１ヵ月はちょうど良い期間設定であった。

L 社事例(22)では、イテレーションを１週間としていたが、期間内で目に見える成果を出すことができ、また１週間単位で柔軟に計画の変更を行うことができた。

関連プラクティス

スプリントバックログ
ベロシティ計測
イテレーション計画ミーティング
日次ミーティング
スプリントレビュー
ふりかえり

3.1.4. **プランニングポーカー** / Planning poker to estimate tasks during sprint planning

“チームで低コストに効率良く見積もる”

別名：見積りポーカー

要約

開発規模の見積り精度が個人のスキルに依存する、非作業者が見積っている場合は、チーム全員で素早く見積もっていこう。その結果、全員の知識や経験を活かした見積り結果を得ることができる。

状況

開発期間の初期ほど、チームメンバーの開発対象に関する知識にばらつきがあるため、見積りを行うのが難しい。よって、効率良くチームの見解をまとめる必要がある。

イテレーション計画ミーティングにて、開発対象の見積りを行い、計画を立てようとしている。

問題

開発の初期ほど開発対象に対する知識が不足している。

見積りに対するチームメンバーの見解が別れることが多くなるため、チームメンバーそれぞれの経験や知識を引き出して、確度の高い見積りを行わなければならない。

また、見積りに時間をかければ見積りの確度が上がるというものではないため、適度な時間で効率良く見積もる必要がある。

フォース

◆見積りをチームでただ議論しているだけでは、時間がかかりすぎてしまう。
◆見積りに時間をかければ見積りの確度が上がるというわけではない。

解決策

見積りを表す数字が記入されたカードを用いて、チームで対話をしながら見積りを行う。

プランニングポーカーはリリース計画ミーティングやイテレーション計画ミーティングの時に利用する。

手順は以下の通りである。

1. 見積りのベースラインを決める。
   基準となる開発対象を選び、その見積りを仮置きする（3 日、3 ポイントなど）。見積りの単位はチームによって様々だが、ユーザーストーリーならば理想日（開発作業に必要な時間のうち周辺的な、開発に直接関係のない作業時間を差し引いた正味の作業時間）やストーリーポイント（ユーザーストーリーを実現するための作業規模を任意の数字に置き換えたもの。時間とは異なる）を使う。タスクの見積りであれば、作業工数（時間）単位で見積るのが一般的である。
2. 対象を一つ選び、見積もる。
   チームメンバーは、これまでの知識や経験を元にして対象を見積り、数字が記入されたカードを選び全員で提示する。
   見積りは 1.で決めた基準に対して相対的にどのくらいになるかという相対規模での見積りを行う。これは、規模の絶対値を正確に見積もることは難しいが、基準に対する相対値ならば、ある程度確度高く見積もることができるためである。
3. カードを一斉に出して、見解を述べる。
   出されたカードの中で、一番大きな数字と、一番小さな数字を出したメンバーから見解を述べてもらい、それらに対して議論を行う。このとき、議論にかける時間をあらかじめ決めておき時間を過ぎても議論がまとまらない場合には、そのまま議論を継続するかどうかの判断を行うようにする。
4. 再度見積りし、カードを出す。
   議論を踏まえて、各自カードを出し直す。
   見積りに対する見解がまとまらなければ、２.に戻りこれを繰り返す。

効果

◆チームの持っている知識や経験をいかした、

確度の高い見積りができる。
◆見積り根拠を互いに述べ合うことで、開発対象や見積りに対する理解が深まる。
◆ゲーム感覚で活発な対話を引き出し、チームの相互理解を深める。

留意点

見積り見解に対する議論は、時間を決める、見積りサイクルの回数を制限するなどして長引かせないようにする。

プランニングポーカーにおいては、互いの専門知識を活かし、見積りの確度を上げるためにも、プログラマだけでなく、データベースエンジニア、デザイナ、テスター、アナリストなど開発関係者全員に参加を促すようにする。

プランニングポーカーに用いる数字は、見積り確度を落とさないためにも、あまり大きすぎる数値を使わないこと。

適度な時間で効率よく見積もることが目的であるため、カードに記入される数字はフィボナッチ数列(1,2,3,5,8,13…)を用いることが多い。

プランニングポーカーに基づく見積り結果は、通常はチーム毎に基準が異なるため、他チームとの比較には使えない。

プランニングポーカーでは開発規模を見積もるため、個人のスキル差によって生じる作業工数のバラツキを考慮する必要はない。

必ずしもカードにこだわる必要はなく、指などで代用することもできる。(見積りじゃんけんと呼ぶ)

見積り単位としてストーリーポイントが許容できない場合は、理想日を見積り単位としてもよい。

利用例

プランニングポーカーは、全体として 64%の適用率であり、特に契約の受託開発で適用率が低かった。しかし B2C サービスでは半数、社内システムでは 80%以上の高い適用率となった。プランニングポーカー自体は契約に依存せずに利用が可能である。

B 社事例(3)では、ユーザーストーリーとそれに対するタスクの両方でプランニングポーカーによる見積りを行っていたが、開発が進み、ユーザーストーリーの見積り確度が高まるにつれ、タスクの見積りは行わなくなっていった。

C 社事例(4)では、要求を管理するチームが、該当イテレーションに入るユーザーストーリーを精査するためにプランニングポーカーを行っていた。

E 社事例(6)では、タスクやユーザーストーリーを見積もる際に必ず使用していた。意見がなかなか出せないメンバーでもカードを出すことで、意思表明がしやすくなった。

G 社事例(9)では、早く楽しく見積もるための工夫として、「せり」のスタイルで見積りを出す工夫を行った(最も大きい見積りがチームの見積りとして採用される)。その結果、チームの雰囲気が良くなった一方で、「せり」によって見積りが過大になることを危惧する声もあった。

L 社事例(23)では、プロジェクトの開始時のみカードを使ったプランニングポーカーを行っており、慣れてくるとカードにこだわらず指で代用するなど、より簡易的に行う工夫を実施している。

関連プラクティス

リリース計画ミーティング
イテレーション計画ミーティング
ベロシティ計測

3.1.5. **ベロシティ計測**/ The project velocity is measured

“ベロシティで着地点を見積もる”

別名: なし

要約

今後のチームの作業量を予測したいならば、チームのイテレーションあたりの開発量を計測し続ける。その結果、ある期間までの開発量の予測や、現在の開発規模についての着地予測を見積もることができる。

状況

ユーザーストーリーの見積りを行っている。

開発を一定期間のサイクル（イテレーション）で繰り返し行っている。

問題

プロダクトの開発規模が見積もれたとしても、チームのイテレーションあたりの開発量がわからなければ、リリース日を見積もることができない。

主要な開発テーマが、それぞれいつリリースできるか計画を立てるためには、1回のイテレーションでどの程度開発ができるのか想定できなければならない。

フォース

- ◆リリース計画のためにはチームのベロシティが必要だが、初めてのチームにはベロシティの実績がない。
- ◆作業を開始して初めて分かることが多く、作業をする前の予測は、あまり役に立たない。
- ◆開発を反復的に進めていく上でチームは開発に慣れてくる

解決策

イテレーション終了時に、チームが完了した全ユーザーストーリーのストーリーポイント数を計測する。その実績を元にチームのイテレーションあたりの開発可能な作業量を算出し、将来の予測に利用し、リリース日を見積もる。

「実計測に基づいた一定の時間内における作業量」を**ベロシティ**と呼ぶ。つまりチームが1回のイテレーションで完了させたユーザーストーリーのストーリーポイントの合計値である。

リリースに必要なユーザーストーリーの見積りを合計すれば、リリースまでに必要な機能の規模を見積もることができる。これをチームのベロシティで割ることによって、必要なスプリントの回数を算出することができる。結果、スプリントの回数からリリース日を見積もることができる。

規模の単位には、ストーリーポイントの他に、理想日を使うこともできる。

ベロシティは、直近のいくつかのイテレーションの計測を元に算出するとよい。ベロシティの算出方法は幅を持たせる、係数を用いるなどの手法がある。［コーン 2009］

ベロシティは、チームの過去の実測値を用いるが、チームがはじめて結成された場合や、メンバーが大きく入れ替わった時、または採用する技術が大きく変わる場合などは、過去の実績値をそのまま用いても有効ではない可能性が高い。

ベロシティに過去の実測値が使えない場合、実際に1度イテレーションを実施してみて、その結果を元に今後のベロシティとして利用してもよい。通常ベロシティがある程度安定してくるのは、数回のイテレーションを過ぎてからである。

イテレーションを実施できない状況下で、ベロシティを算出しなければならない場合は、各メンバーのイテレーションあたりの作業可能時間を予測する。一方、ユーザーストーリーをタスクに分割し、タスクの作業時間を見積もることで、イテレーションに入るタスク全体の規模を予想できる。それらの結果から、イテレーションで実施するユーザーストーリー全体の規模を見積もることができ、1イテレーションあたりのストーリーポイントの合計値を見積もることができる。これが想定されるチームのベロシティとなる。

ベロシティが、イテレーションのタイムボックスの箱の大きさとなる。ベロシティを元にイテレーション計画ミーティング、リリース計画ミーティングが実施される。

留意点

ベロシティは、開発を進める過程で変動するため、イテレーションを進めていく中でベロシティの計測幅を見直す必要がある。これに合わせて、リリース計画の見直しも検討する必要がある。

途中でイテレーションの長さを変更した場合は、ベロシティもその変化を考慮して調整しなければならない。

効果

◆チームのベロシティを元にリリース計画を立てることができる。
◆状況に応じたリリース計画を立て続けることができる。

利用例

ベロシティ計測は、B2C サービス、社内システム共に 60～80%の割合で利用されていた。

G 社事例(9)では、当初は 1 週間サイクルで開発を行っていたが、途中で 1 ～2 週間サイクルを適時選択していたため、その都度ベロシティを補正していた。

H 社事例(11)では、複数チームで構成されるプロジェクトであったため、複数チームを見ているプロダクトオーナーが理解しやすいようにベロシティで計測するユーザーストーリーの見積り単位をチームで横断的に揃える必要があった。

K 社事例(20)では、2～3 年間同じチームで開発をしているため、ベロシティが安定し、見積り確度が高くなった。

L 社事例(21)では、顧客や開発者がファンクションポイントに馴染んでいたため、ストーリーポイントの代わりにファンクションポイントで予実管理を行っている。

関連プラクティス

プロダクトバックログ
リリース計画ミーティング
イテレーション計画ミーティング
イテレーション
プランニングポーカー
ユーザーストーリー

3.1.6. **日次ミーティング**/ Short daily meeting to resolve current issues

“現在の状況を共有するための短いミーティング”

別名:
朝会、朝礼、デイリースクラム、スタンドアップミーティング

要約

毎日、時間を決めて短い時間で関係者が顔を合わせる。その結果、チーム全体が日々の必要な情報を共有できるようになる。

状況

チームには、プロジェクトのタスクをこなす時間が不足しがちである。状況や情報の共有のために取れる時間はほとんどない。

問題

情報の共有遅れが問題を大きくする

情報共有の時間が取れないまま、状況認識と問題対処への判断が遅れると、問題が大きくなり、より深刻な状況を招いてしまう。

フォース

◆関係者が多忙なため、情報共有のための時間が取れない
◆情報共有の間隔が空いてしまうと、情報量が増え、共有に必要な時間が余分にかかってしまう。

解決策

チームのメンバー全員が集まって必要な情報を短い時間で毎日共有する。

短い時間（15 分が目安）で済むように、必要な情報に絞って共有する。

情報共有の間隔が長くなるほど、共有に要する時間が必要となるため、毎日短いミーティングを設けるようにする。

共有する情報としては、(1) 昨日行った作業、(2) 本日行う予定の作業、(3)困っている点や問題点をチームメンバーがそれぞれ報告する。

時間が長くなることを防ぐために、全員が立ったまま行うのが望ましい。

必ずしも朝の時間帯にこだわらず、関係者が集まりやすい時間帯に開催する（例えば、終業近い時間帯に開催する=夕会）。情報共有の頻度をもっと頻繁にしたい場合は 1 日に複数回実施することも検討する。

問題の解決方法まで議論する時間を含めるとミーティングが長時間に及ぶため、日次ミーティング自体は情報の共有に留め、問題解決は別途会議体を設けることが望ましい。

状況を共有するためには、チームの状況が可視化された情報（タスクボード、かんばん、バーンダウンチャート）のある環境でのミーティングが最も理想的であるといえる。

留意点

複数チームで実施する場合には、段階的に開催する。段階には大きく以下の 3 通りのパターンがある。

(1) 個別→全体
(2) 全体→個別
(3) 個別→全体→個別

(1) の場合は、個々のチームの状況説明の後に、代表者のみで集まる。そのため全体としての状況の把握はしやすいが、チームをまたいだ周知事項などの場合には伝達が行き届かない懸念もある。
(2) の場合は、(1) とは逆パターンとなる。
(3) に関してはいずれのパターンにも対応しているが時間がかかるのが難点である。

異なるチームのメンバーを集めて全体の日次ミーティングを実施しても、参加者に価値のない特定チームの報告・情報だけだとうまくいかない。参加者に意味のある情報のみ共有すること。

効果

◆情報共有の漏れを防ぎ、問題の把握と対処を適切なタイミングで行うことができる。
◆朝の日次ミーティングはチームの１日のリズムを整える時間となる。

利用例

日次ミーティングはすべての事例で利用されていることから、アジャイル型開発の導入を検討する際においては様々な状況で利用必須のプラクティスであることがわかった。

B 社事例(2)では、日次ミーティングの長時間化という問題の対策として、所用時間をタイマーで計測し、ホワイトボードに記録して可視化することにより、朝会の所要時間が短くなっていった。
また、プロダクトオーナーも含めた通常のミーティングである朝会に加えて、技術的な細かい課題を共有するための技術者ミーティングを夕方に実施し、翌日の朝会でプロダクトオーナーと共有すべき事項を意識合わせしていた。(プロダクトオーナーが多忙のため朝会での情報共有を重視していることから)

C 社事例(4)では、複数のチームから構成される中規模プロジェクトであり全員が集まることは困難であった。そのため最初にチーム毎のキーマンおよび要件チームで全体の課題を取り上げる日次ミーティングを行い、その後、各チームに内容を伝達した上でチーム毎に行う通常の日次ミーティングを実施していた。

E 社事例(6)では、長時間化を避けるために、日次ミーティング内では、報告内容の問題解決は行わずに一旦ミーティングを終了させ、その後に改めて問題解決のためミーティングを設けるルールを決めていた。

G 社事例(9)では、遠隔地にいるメンバーも日次ミーティングに参加できるように Web 会議システムを利用していた。またスマートフォンをスピーカーフォンとして利用して遠隔地と日次ミーティングも行っていた。

J 社事例(17)では、メンバー同士が持っている情報を頻繁に共有する必要があったため、1 日 3 回（朝、昼、夕）10 分程度のミーティングを実施した。
その結果、問題を報告/解決するためのリズムが開発メンバー全員に浸透し、短期間で問題提起と解決が可能になった。

L 社事例(22)では、日次ミーティングで毎日顔を合わせることでメンバー間の情報共有をすることで問題を１人で何日も抱えることがなくなった。

関連プラクティス

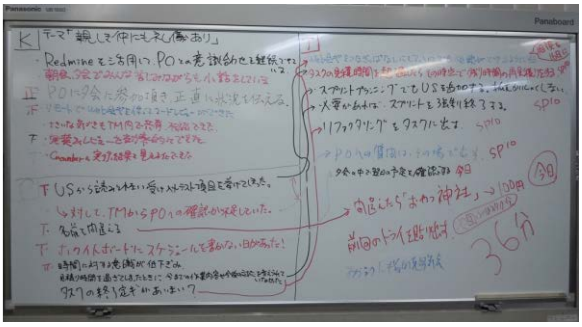
タスクボード（タスクカード）
かんばん
バーンダウンチャート
チーム全体が一つに

参考文献

平鍋健児・天野勝 (2005) 「プロジェクトファシリテーション 実践編 朝会ガイド」
http://objectclub.jp/download/files/pf/MorningMeetingGuide.pdf

3.1.7. **ふりかえり** / sprint retrospective to learn from previous sprint

“改善の螺旋階段を登りチームが反復毎に成長する”



別名:
レトロスペクティブ、リフレクション、
内省、反省会

要約

最初から開発プロジェクトに最適な手法がわからないのであれば、イテレーション毎にふりかえり、学んだことを共有してより最適なやり方を編み出す。その結果チームの現状に最適な開発プロセスを作りあげていくことができる。

状況

イテレーション毎に、チームは動くソフトウェアとして成果を出そうとしている。イテレーションを繰り返して、チームはソフトウェアを開発していく。

アジャイル型開発は要件定義からテスト、配備までの幅広い工程を頻繁に繰り返し何度も行うため、一度失敗しても、またその工程を繰り返す機会がある。

問題

開発チームにとって、最初から最適な開発プロセスを実践することは困難である。

仮に最初にうまくいきそうな開発プロセス、ルールを定めたとしても、それがベストなものである保証はどこにもない。

フォース

◆イテレーションの単位で開発に区切りがある。
◆イテレーションでの開発はうまくいくこともあるが、うまくいかないこともある。
◆教科書通りの手法でうまくいかせようとしても、現実とのギャップによりうまくいかない。
◆最初にルールを定めてしまうと、最適な方法を模索するよりも、そのルールに従うことが目的になりがちである。

解決策

イテレーション内で実施したことを、最後にチームでふりかえり、開発プロセス、コミュニケーション、その他様々な活動をよりよくする改善案を考え実施する機会を設けること。

XP では“内省”をチームで実施することを原則としている。スクラムでは、スプリントレトロスペクティブとして、スプリントの最後にスプリントレビューとは別に、チームが開発プロセスを改善することをプラクティスとして定義している。

イテレーションの最後に、チーム全員が集まり、その期間の中で起きた出来事をふりかえる。A)開発プロセス、B)チーム内のコミュニケーション、C)プロダクトの品質など様々な点を検討材料とし、次のイテレーションに向けての改善点をチームで考え、実施することに合意する。実際に次のイテレーションで、改善案を実施して効果を評価する。単なる問題点を列挙するのではなく、具体的な改善案の実施を積み上げていくことが重要である。

日本国内では、ふりかえりの手法として KPT（Keep, Problem, Try の頭文字）が広く用いられている。[天野 2006]
KPT とは、今後も続けたいこと（Keep）、うまくいかなかった改善したいこと（Problem）、次に試したいこと（Try）という観点でふりかえり、チームで改善案を抽出して実施する手法である。『アジャイル レトロスペクティブズ』にはふりかえりの様々な手法が紹介されている。[ダービー et al. 2007]

他にも、トヨタ生産方式が発祥の「なぜなぜ5回」（発生した問題の原因を「なぜ問題が起きたのか」という問いを5回繰り返すことで考え、根本原因を導きだす手法）のように、根本原因を追求して対策案を考える手法も併用して実施される。

ふりかえりでは、シングルループ学習（「どう

すれば私たちがやっていることをもっとうまくやれるのか？」を問う方法）だけでなく、ダブルループ学習（「なぜ私たちはこれがやるべきことだと考えているのか？」を問う方法）によって、依拠している前提、価値観、思考、仮説自体をも検証して改善していくことが必要である。[2]

複数チームで構成されるプロジェクトでは、ふりかえりをチーム単位で行う他に、全チームメンバーや複数チーム混成メンバーで実施することで、よりプロジェクト全体としてのふりかえりの効果を高めることができる。

ふりかえりはイテレーション毎に実施するだけでなく、リリース毎や、マイルストーン毎のように、より長い期間を対象に実施するのも効果的である。

ふりかえりの進行（ファシリテーション）は、ファシリテータ役を立てて行う。特定のファシリテータ役が進行を一手に引き受けるのではなく、プロジェクトチームのメンバーが持ち回りで進行を行っていくことで、メンバー全員がふりかえりの進行に責任を持つようになり、うまくいく場合が多い。

ふりかえりの経験者がまったくいない場合はアジャイルコーチやファシリテータを招聘して、進行役を依頼するとスムーズに進行され、ポイントも学ぶことができる。

アジャイル型開発を組織的に導入している場合は、ふりかえりでの各チームの改善を元に、組織に合わせたアジャイルスタイルに進化を進めていく。

留意点

◆ふりかえりの参加者は「参加者全員がベストを尽くしたこと」を疑ってはならない。(ノーム・カースの最優先事項) [Kerth 2001]
◆メンバーや問題点を糾弾する場にしてしまうと、改善すべき点を積極的に話し合えない場になってしまう。
◆チームのメンバー同士が、充分な信頼関係が醸成できていない場合には、ふりかえりの場を設けても、意見が出ないことが多い。
◆ふりかえりがシングルループ学習のみの場合は、そもそも前提として考えているやり方について疑問を投げ掛けにくいため、表面的な改善に留まってしまい、根本的な解決に辿りつかない。
◆問題の根本原因の分析に時間をかけ過ぎて、実際の改善行動まで辿り着かないと、参加者のモチベーションが低下する。
◆開発のスピードを重視している現場では、ふりかえりが軽視される傾向がある。
◆改善案を出しても、実際に実行可能なレベルの具体的なアクションになっていないと実施されない。（「XXX を意識する」、「しっかり XXX をする」では実行可能なレベルになっていない）
◆複数のチームで連携しているプロジェクトの場合は、チーム毎にふりかえりを実施しているとチームの外部への不満、責任転嫁になってしまうことがある。
◆「新しく何かをする、ルールを決める」だけでなく、積極的に「今は行っているが、必要のないことを止める、減らす」ことを考える。そうしないと「やらねばならないこと」が増え続けるだけである。

効果

◆イテレーション毎にチームの開発プロセスや規律を改善できる。
◆早期の比較的小さい失敗から学ぶことができ、大きな失敗をしにくくなる。
◆チームで学習する結果、個々のメンバーの成長が早くなる。

利用例

ふりかえりは、ほぼすべての事例で利用されていた。しかしその内容や効果は事例によってバラつきがあり、チームにとって不可欠になっている場合もあれば、あまり定着しないという場合もあった。

B 社事例(2)では、KPT をベースにふりかえりを実施していた。マンネリ化に陥いった場合は、その場で独自の方法で即興のふりかえりを実施していた。明るい雰囲気になるように、チームの中で笑いが出るような話題を率先して取り上げていた。

C 社事例(4)では、プロジェクトが複数のチームで構成されており、チーム毎にふりかえりを実施していたが、結果、自分達で何か行動できる改善策を実施するのではなく、チームの外部への不満を発散するだけの会になってしまった。

[2] http://www.infoq.com/jp/news/2012/05/double_loop

これを防ぐために、チーム混成、あるいは全体でふりかえりを実施するようにした。
また、イテレーション単位のふりかえりだけでなく、3ヶ月単位での、より大規模なふりかえりを実施していた。

D社事例(5)では、ふりかえりの時間が当初は2時間だったものを効率化して1時間に減らした。手法には「なぜなぜ5回」を用いた。

E社事例(6)では、ふりかえりの重要度は大きかった。ふりかえりのやり方自体も見直しながら、チーム独自のやり方を編み出していった。また、ふりかえりがメンバー同士を互いに認め合う場にもなっていた。

G社事例(9)では、ふりかえりの習慣がなく当初は戸惑っていた。最初はスピード感がなかったが、徐々に解消されていった。さらに、ふりかえりにテーマ設定をするという工夫をした。常に楽しんで取り組む気持ちを持って取り組んでいた。

I社事例(15)では、ふりかえりの推奨はしているが、チームに定着はしていない。特に、新規プロジェクトの場合はやったほうがよいと認識されているが、それほど実施はされていなかった。ふりかえりを実施しなかったからといって、すぐに開発に支障が出るわけではないため、他のプラクティスに比べると、どうしても実施の優先度が低くなりがちである（重要度が高いことは認識されている）。

M社事例(25)では、当初はKPTを用いてふりかえりを行っていたが、ファシリテータの技量にふりかえりの質が依存してしまう、また声の大きいメンバーに影響を受けてしまうことに気づいた。そのため、意見を集めるやり方として、[ダービー et al. 2007]で紹介されている555（Triple Nickels）という手法を用いて、発言ではなく紙によかった点と改善点を書き、全員に回す方法を取っていた。改善点は投票の結果、優先度の高いもののみ実施するようにしている。

関連プラクティス

ファシリテータ(スクラムマスター)
アジャイルコーチ
組織に合わせたアジャイルスタイル

参考文献

[Kerth 2001] Kerth, N. (2001). Project Retrospectives: A Handbook for Team Reviews, Dorset House

[天野 2006] 天野勝 (2006) 「プロジェクトファシリテーション 実践編 ふりかえりガイド」 http://www.objectclub.jp/download/files/pf/RetrospectiveMeetingGuide.pdf

懸田剛・天野勝 (2011) 「Retrospective Patterns」 http://patterns-wg.fuka.info.waseda.ac.jp/asianplop/proceedings2011/asianplop2011_submission_17.pdf

3.1.8. **かんばん** / Kanban

“計画をせずとも価値の流れを実現する”

別名：Kanban，フィーチャパイプライン

要約

プロジェクトを取り巻く状況が不安定で計画を頻繁に変更せざるを得ない場合には、同時に開発する機能の数を制限して流量コントロールする。その結果、機能を流れるように提供することができ変化に対応しやすい。

状況

システムは既にリリース済みで運用を開始している、あるいはリリース前ではあるが、要件の変更が頻繁に発生する。

アジャイル型開発で採用されるイテレーション期間は、1～4 週間の期間を固定して、その間でチームが動作するソフトウェアを実現する。チームは一つのイテレーション期間が終了する度に顧客にソフトウェアを届けることができる。

特に新規開発のプロジェクトは、複数のイテレーションをまとめてリリース計画を作る。顧客の手元に欲しい機能が届くまでに数ヶ月以上の時間を要する場合が多い。

問題

事前に計画を立てたいが、いつ、どのくらいの要件が発生するのかが予測できない。

特に、運用中のシステムでは、障害や機能追加などが不定期に発生する。そのためイテレーション単位で計画しようと思っても、イテレーションの開始時点では何をしなければいけないのかが見えていないことが多い。

フォース

- ◆運用中はできるだけ早く変更を適用したい。特に障害の改修はすぐにでも適用したい。
- ◆プロダクトオーナーが欲しい機能の要求を出してから、利用者の手元に届くまでの時間（リードタイム）をできるだけ短くしたい。
- ◆

解決策

一定期間で実施する作業を計画するのではなく、重要な要件から順に実施する。その際、同時並行で作業する数を制限し、作業の流れが機能単位で一つずつ完了するように作業を進めること。

作業の進行管理は、壁に貼られた**かんばんボード**と呼ばれるボード上に、**かんばん**(実現する要件、機能を表現する)を置き、移動することで行う。
見た目はタスクボードと似ているが、大きく違う点としては、作業状態を示す各**レーン**に書かれた数字である。

この数字は **WIP**（Work In Progress：仕掛かり）と呼ばれる同時に実施できる並列作業数を意味し、レーン毎にこの数を制限する。作業はかんばんボード上で「見える化」される。チームはボードの上で、後工程の作業が WIP 以下になったら、次の工程に作業を渡すことができるようになる。

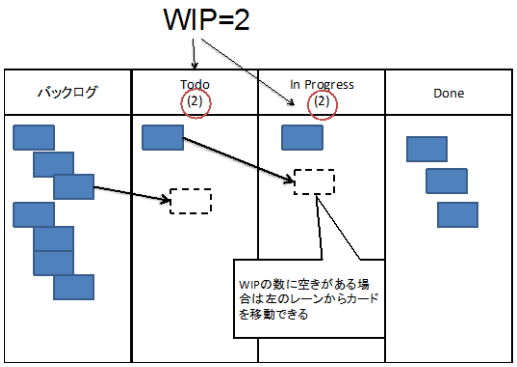


図 1 WIP に余裕があり移動できる

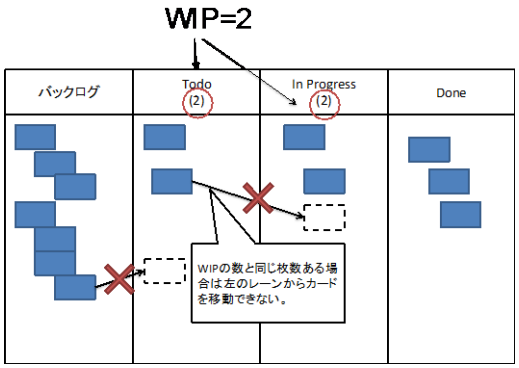


図 2 WIP の制約のため移動できない

レーンの作業の数が WIP 以下であれば、図 1 のように隣のレーンの作業を WIP に達するまでレーン上に追加することができる。逆にレーン上の作業数が WIP に達している場合は、図 2 のように新しい作業をレー

ンに追加することはできない。あるレーンで作業が滞留していたら、その滞留を取り除いて作業がよどみなく流れるようにする。

要件の実現にどれくらい時間がかかりそうか、という見積りは特に必要とされない。その代わりに、**リードタイム**（かんばんボード上に置かれたかんばんが完了するまでの時間）と、**スループット**（一定時間内に完了したかんばんの数）を測定して改善し続ける。

かんばんは、イテレーション毎の計画に依存しないため、事前に計画する必要がない。随時発生する要件や障害を貼り、WIP を守って作業を進めていき、リードタイムとスループットを計測していく。

もしも、問題がある場合は、かんばん上で停滞していることが見える化されるため、チームがその問題を取り除くことに集中する。そのため、自然とボード上をカードが流れていくようになる。

かんばんを用いて、機能がよどみなく実現される開発の進め方を**フィーチャパイプライン**とも呼ぶ。

かんばんは、一つのチーム内で閉じる必要はなく、例えばレーンを工程毎のチームに分けて、設計チーム→開発チーム→テストチームという流れで分担して進めていくこともできる。ただしこの場合も、各工程の WIP を守るようにして、滞っている工程のボトルネックを全員で取り除いたり、各工程毎の人員数を調整して、ボトルネックを解消させるように全体で最適化していく必要がある。

レーン毎のボトルネックの除去は、日次ミーティング、あるいはもっと頻繁なタイミングで行う。

顧客がかんばんボードに追加要件をいつでも貼り出すことができるようにオンサイト顧客を検討するとよい。

かんばんボードはふりかえりによって改善していくのがよい。

留意点

◆WIP の制約は、チーム状況によって調整する必要がある。厳しすぎるとパフォーマンスが上がらず、緩すぎるとカードがレーン（特定の工程）にたまってしまいボトルネックが発生しやすい。
◆WIP の制約を守らないと、ただの視覚化されただけのボードになってしまう。

効果

◆あらかじめ計画を立てなくても効率的に開発を進めることできる
◆自然に障害を取り除くようになり作業の流れを最適化しやすい
◆作業の流れを最適化しようとすることで協働作業が促進される。

利用例

かんばんは、適用率は 10%程度と低く、特に B2C サービスでの利用に限られていた。しかし、一部では運用、保守作業で使われている事例が見受けられた。

H 社事例(11)の場合は、開発作業はイテレーションで開発していたが、保守作業はかんばんを用いて管理していた。

K 社事例(20)では、運用案件はかんばんを顧客と共有して運用しており、日常的に定着し、必要不可欠となっている。

L 社事例(21)では、アーキテクチャチームだけが、かんばんを利用していた。アーキテクチャチームでは、他のチームから共通部品への要望が不定期に発生し、しかも素早く対応する必要があるため、かんばんボードが役に立った。

M 社事例(25)では、障害対応や運用保守系でかんばんを利用している。事前に計画ができる領域ではスクラムでスプリント計画を立てるが、見積りや計画ができない領域はかんばんを使っている。

関連プラクティス

オンサイト顧客
ふりかえり
日次ミーチィング

参考文献

Anderson, D. (2010). Kanban. Blue Hole Press

Kniberg, H. Skarin, M. 大田緑・近藤寛喜 (2010). 『かんばんとスクラム 両者のよさを最大限引き出す.』InfoQ

3.1.9. スプリントレビュー / Sprint review meeting to present completed work

“スプリントレビューでチームの次の航路を決定する”

別名: デモ

要約

イテレーションの終わりに完了したものを関係者にデモをする。その結果、チームは次のイテレーションで何をするべきかフィードバックを得る。

状況

開発を一定期間のサイクル（イテレーション）で繰り返し行っている。

スプリントバックログによって、イテレーションで開発すべき項目、実施すべき作業を管理している。

開発しているプロダクトのフィードバックを求めることができるステークホルダーが存在している。

問題

イテレーションの成果を関係者で確認する機会がなければ、現在の状況を判断し、次のイテレーションの適切な計画を立てることができない。

現在のイテレーションの結果を踏まえることで、次のイテレーションでやるべきことが計画できる。

開発しているプロダクトへのフィードバックがなければ、正しいものを開発しているのかどうかの判断をすることができないため、リリース間際になって、作ったものがプロダクトオーナーやユーザーの想定と違ったということが起こりうる。

フォース

- ◆イテレーションにつき１回のレビューでは、やり方を工夫しなければ、膨大な時間がかかってしまう。
- ◆プロダクトをレビューする関係者が集まれる機会は限られている。
- ◆ステークホルダーは今回のスプリントで何ができたのかを知りたがっている。

解決策

イテレーションの終わりに、完了したものをステークホルダーにデモをし、フィードバックを得るためのレビューを開催する。

スプリントレビューは、イテレーションにつき１回開催する。スクラムマスターがその実施に責任を持ち、ステークホルダーを集める。

イテレーションの長さに応じてスプリントレビューの時間を決める。１イテレーションが１か月の場合は４時間、２週間の場合は２時間程度に収まるように段取りをする。

スクラムマスターが、当該イテレーションの目標と実際の成果を比較説明する。

開発チームが、当該イテレーションで開発した機能を実際に動かしながら説明を行う。関係者からのプロダクトに対するフィードバックを集める。

プロダクトオーナーはレビューの結果を踏まえて、プロダクトバックログの内容を変更し、優先順位付けする。

レビュー参加者全員で、次に何をするべきか議論を行い、次のイテレーション計画ミーティングにインプットする。

留意点

スプリントレビューの準備に時間をかけ過ぎないこと。

イテレーション計画ミーティングで決めたゴールとその結果は、できている/できていないに関わらず、状況と理由を共有しておくこと。

ステークホルダーに情報を提供するための機会であるため、進捗の遅れなどがあった場合にも、事実を覆い隠してはいけない。

効果

- ◆イテレーションの成果をステークホルダー全員で共有することができる。
- ◆レビューの結果を元に必要に応じてプロダクトバックログを変更し、次のイテレーションでやるべきことを検討することができる。

利用例

スプリントレビューは、スクラムで定義されているプラクティスであるため、XP を採用している事例ではまったく適用されていなかった。

しかしフィードバックをもらうという観点では、スクラムではなくとも、全体の 60～70%程度の適用率であった。プロダクトオーナーを立てて開発を進める際には、適用したほうがよい。事業企画者が共通の部屋にいるような状況では、実現したユーザーストーリーのデモを随時行い確認してもらうため、スプリントレビューの適用率が下がると推測される。

A 社事例(1)では、スプリントレビューをイテレーションの最後に実施するのではなく、ユーザーストーリーが完成する度にステークホルダーにデモしていた。

B 社事例(2)では、朝会でステークホルダーにデモをしていた。そのため、スプリントレビューは対象機能の完了を確認するのみとし、デモは行わないことにした。

B 社事例(3)では、「完了」の定義を作成していなかったため、確認があいまいとなってしまった。

C 社事例(4)では、スプリントレビューに先立って、完了した機能をプロダクトオーナーに動かしてもらい、フィードバックを Excel にまとめるようにした。
スプリントレビューでは、当初ステークホルダー、チームメンバー全員が出席していたが、時間短縮のため、開発チームの代表と要求を取りまとめるチームのメンバーが出席する形を取った。

顧客にデモを実施するための、「デモルーム」を用意してもらった。この部屋では、プロダクトオーナーがいつでも動くソフトウェアを試すことができるようになっていた。

D 社事例(5)では、スプリントレビューをプロダクトオーナーと毎日実施するようにしていた。プロダクトオーナーが関わっているビジネス側のステークホルダーとは、プロダクトオーナーがコミュニケーションを取り、成果を伝えるようにしていた。

G 社事例(9)では、デモに時間がかかり過ぎていたため、受入テストツールを使ったテストシナリオを事前にスプリントレビュー参加者に共有することで、デモの実施時間を短縮することができた。

L 社事例(21)では、イテレーション毎にデモを中心にしたレビューを実施していた。システムのユーザーが実際のソフトウェアを見て動かすことで、改善要望を 100 個近く拾い上げることができた。

関連プラクティス

プロダクトバックログ
イテレーション計画ミーティング
イテレーション
ふりかえり

3.1.10. タスクボード(タスクカード) / Task board(Task card)

“タスクボードが情報発信の場となる”

別名: スクラムボード

要約

チームのタスク状況を可視化するために、タスクカードを用意し、状態を管理するタスクボードにはりつける。その結果、だれがどのタスクを担当しているか、どこかに異常が起きていないかなどをチーム全員が把握できるようになる。

状況

チームでタスクを洗い出し、タスクベースで作業を行っている。

問題

今、チームの抱えるタスクが、どれに取り掛かり中で、どのような状態になっているかわからない。

だれも作業をせずに、いつまでも着手されないまま残っているタスクがあることにさえ気づかない。

どのタスクが終わったのかわからない。

特定のだれかがタスクを抱えすぎていたとしても、検知できない。

フォース

- ◆タスクを分担すると、開発者は自分のタスクに集中してしまい、周囲への関与や協力が薄くなってしまう。
- ◆タスクをソフトウェア上で管理してしまうと、 全体の状況を俯瞰して見ることが難しい。

解決策

タスクを付箋などの紙に書出し(タスクカード)、ToDo/Doing/Done の状態で区画(レーン)を分けたボードにはりつけ、チームが見える場所に配置する。

ToDo はやるべきタスク、Doing はやっているタスク、Done は終了したタスクである。タスクボードには左から右に ToDo/Doing/Done とレーン（列）を分けて並べることが多い。

ToDo/Done はそれぞれ一つのレーンとし、Doing のレーンをメンバーごとに分けることで、だれがどのタスクに仕掛り中なのか見えるようになる。

保留を意味する Pending レーンや、ユーザーストーリーのレーンの追加など、レーンの使い方は現場によって様々な工夫をするのが良い。

日次ミーティングはタスクボードの前で実施して、チームで共有する時間を設ける。動かないタスクや、だれかにタスクが偏っているなどの異常を検知し、チームで対策を検討する。

プロジェクトの状況によって、メンバーが同じ場所にいない場合には効果が現れにくいが、オフショアにおいても、そのサイト毎にタスクボードを利用すれば可能である。

タスクボードは紙・手書きツールを使って、チームの創意工夫を発揮するのがよい。付箋の色、形、大きさ、場所、ペンの色、太さなど、自分達の状況に合ったタスクボードにカスタマイズしていくのが望ましい。ふりかえりのタイミ

ングなどでより良い改善のアイデアを出していく。

タスクボードは、共通の部屋で利用することでより効果を発揮できる。チームの誰もがタスクボードを見ることで状況が一目でわかるようになる。

タスクボードは「今どうなっているか」を示すことはできるが、「今後どうなるか」という傾向、時系列変化を表現することはできない。バーンダウンチャートと併用して、チームの今とこれからを見える化するのが望ましい。

留意点

タスクボードで管理をしたとしても、タスクボードを常に更新しなければ状況の見える化にはならない。

タスクの状態が変化したらメンバーが各自でタスクボードの更新を行うこと。

都度更新するのが煩雑な場合は、日次ミーティングの前や1日の終わりなど1日の中でイベントにひもづけて更新をするルールをチームで決めても良い。

タスクボードは機能を実現するためのタスク(作業)の状況を見える化することで行動を誘発する。一方、かんばんは実現する機能そのものの状況を見える化し、かつ流量コントロールを行うことでボトルネックを明白にする。

効果

◆タスクの見える化によって、チームメンバーの異常を検知できるようになる。その結果、チームで相互協力しやすい。
◆タスクの実施漏れを防ぐことができる。
◆チームの状況が表現されているため、それを見ながら日次ミーティングを実施しやすい。

利用例

タスクボードは、社内システム開発(69%)よりも、B2C サービス(94%)でより多く使われていた。中大規模よりも(75%)、小規模(90%)でより多く使われていた。またXP採用事例では100%の適用率であった。B2C、小規模案件では積極的に利用していきたいプラクティスである。

B 社事例(2)では、付箋を活用したタスクボードを活用しており、アナログによる運用が非常に好評だった。一度、デジタルツールによる管理も試みたが、一覧で全体が見えない、ログインが面倒であるという理由で使用を取りやめた。

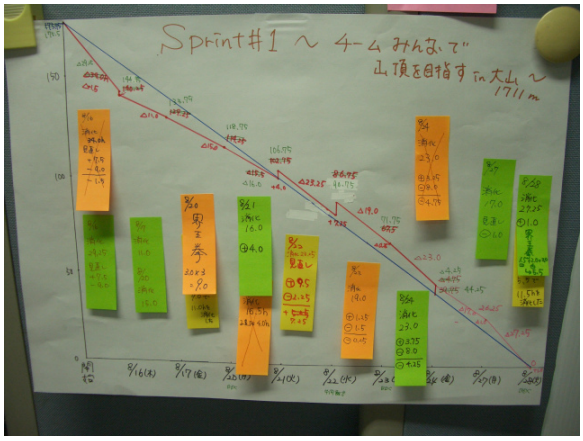
J 社事例(17)では、ホワイトボードを使ってタスクボードを運用していた。ホワイトボードならば、タスクボードのフォーマットを手軽に変更できるためである。

関連プラクティス

チーム全体が一つに
イテレーション計画ミーティング
日次ミーティング
紙・手書きツール
ふりかえり
共通の部屋
バーンダウンチャート

3.1.11. **バーンダウンチャート** / Burn down chart to monitor sprint progress

“残作業量の変遷から開発状況を一目瞭然にする”



別名: なし

要約

リリースに向けた進捗やイテレーションの状況を把握するために、チームの残作業量を可視化したバーンダウンチャートを書く。その結果、状況に対する分析や適切なアクションを取ることができるようになる。

状況

リリースやイテレーションの計画を立てており、各イテレーションや日々の実績作業量を計測することができる。

プロダクト全体とイテレーション毎の予定作業量が見積もられている。

問題

イテレーションをこなしているだけでは、プロジェクト全体が順調に進んでいるのか危険な状態にあるのかがわからない。

イテレーションが進むにつれて、当初の計画ではわかっていなかった作業が発生したり、新しい要求が発見されたりすることがある。これらのように増えていく作業を含めて、状況を分析しなければ、状況に対する打ち手が取れない。

作業が増える、作業が予定通り終わらないなど、イテレーション毎に状況が変わるため、結果的にリリース日を遵守できるのかを予測することができない。

作業の消化が予定に対して追いつかない場合、スコープを調整してリリース日に間に合わせるべきか、リリース日を調整してスコープを維持するべきかについては、プロジェクト全体を俯瞰してどれくらい遅れているのか、もしくは、スコープをどれくらい削る必要があるのかという分析が必要となる。

予定作業量に対する実績の乖離がどの程度深刻なのかは、目の前のイテレーションだけで考えていても将来の見積りができない。
プロジェクトの状況は、プロダクトオーナーやプロジェクトマネージャだけではなく、チームで把握しておく必要がある。

フォース

◆予定作業量と実績の数字だけを見ていても、プロジェクトの状況をすぐに明確に把握することはできない。

解決策

残作業量を対象期間で計測したグラフを作り、チームで共有して、状況の分析と対策を行う。イテレーション、リリースと違う時間軸で、別々に残作業量を計測して状況を明らかにする。

リリースバーンダウンチャートの場合、縦軸にはリリースまでに必要な作業量（必要な作業時間やストーリーポイントを用いる）、横軸にはリリースまでの期間（イテレーションを単位とする）を取る。各イテレーションにおいて残作業を計測し、チャート上にプロットしていく。

イテレーションバーンダウンチャートの場合、縦軸には当該イテレーションで予定した作業量、横軸には作業日を取る。1イテレーション内での作業量の変遷を確認する。

作業開始時の最大作業量から、目標となるリリース日もしくはイテレーション終了日までを直線で結び、理想的な作業進行を表す理想線を書いておく。実績をプロットしていく際に、この理想線より上となる場合は作業が遅れている、理想線より下となる場合は作業が進んでいる、と判断することができる。

作業の遅れが続いた場合、実績のラインは下方へ収束せず、危険な状況にあることを兆候とし

て捉えることができる。

作業が増えた場合は、実績のラインが右肩上がりになる場合もある（バーンアップ）。作業の進捗が計画に対して追いついていないことが一目瞭然でわかる。

バーンダウンチャートは、イテレーションや作業日の終わりに実績を記載する。

バーンダウンチャートからプロジェクトの状況をだれもがすぐに明確に把握できるように壁に張り出すなどして、チーム内で可視化する。

バーンダウンチャート上の残作業量と残りの日数やイテレーション数を元に状況を分析しアクションを取るかどうかを決める。

チャートは表計算ソフトなどを用いて計算、出力することもできるが、可能であればチームメンバーが手書きで更新するとよい。そうすることでチーム全員が今の状況をより実感を持って認識することができる。

チャートは、折れ線グラフの他、棒グラフの形状を取ることもできる（バーンダウン棒グラフ）[コーン 2009]。折れ線同様に、縦軸に残作業量を取るが、作業が増えた場合、棒グラフの下を延ばすように書く。このようにすると、作業の削減と追加を明確に分けて記載することが可能であり、チームのベロシティとプロジェクトスコープの変化を分けて管理することができる。

バーンダウンチャートの類例として、機能の完了状況の可視化に重点を置くパーキングロットチャート[コーン 2009]という方法もある。パーキングロットチャートは、四角形の中に、機能とそれを構成するユーザーストーリーの数、ストーリーポイントの合計、完了したストーリーポイントの割合を記載する。この四角形を機能の数だけ用意するものである。機能単位で完了状況を把握するには優れた道具である。

スプリントバーンダウンチャートはタスクボードと併用することで、「今の状況」と「今後の傾向」の視点を得ることができるためアクションを取りやすい。

留意点

◆実績のチャートへのプロットを怠るとバーンダウンチャートは信頼できるものではなくなり、用を成さない。

◆バーンダウンチャートをチームの評価に使わないこと。問題の把握と、対策を取るための道具として使う。

効果

◆バーンダウンチャートによって、プロジェクトが順調に進捗しているのか、もしくは危険な状況にあるのかなど、チームのだれもが状況をすぐに明確に把握することができるようになる。
◆状況を正しく把握することによって、適切な対策をタイムリーに取ることができるようになる。
◆バーンダウン棒グラフによって、ベロシティが想定より上がらないために進捗状況が遅れているのか、またはベロシティ以上に追加作業が起きているのか、把握することが可能であり、それに応じた対策を取ることができる。

利用例

バーンダウンチャートは、全体の 87%の事例で適用されていた。適用率を調べてみると、システム種別の切り口では、社内システムでの適用率が 100%であった（B2C は 78%）。また規模の切り口では、中大規模でも適用率が 100%（小規模は 83%）、契約別の切り口では、受託開発での適用率が 100%であり（自社開発では 75%）、手法別での切り口では、スクラムでの適用率は 100%であった。（XP では 80%）。

全般的に適用率が高いプラクティスではあるが、より時間管理が厳しい業務に取り組む場合、あるいはスクラムを導入する際には最優先で検討を考慮するプラクティスであることがわかった。

A 社事例(1)の場合は、チャートの更新を怠ったため風化してしまった。

B 社事例(2)では、イテレーションバーンダウンチャートを使っていた。夕会で実績の記録を行い、朝会でプロダクトオーナーを含めてチーム全員で確認するようにした。プロット作業を持ち回りにすることで、チャートの更新を維持することができた。

C 社事例(4)では、イテレーションバーンダウンチャートを使っていた。チャート更新の担当者を役割として配置した。更新は朝会よりも前に行うようにして、朝会で状況を共有していた。また、表計算ソフトでもタスクの作業時間を管

理しており、表計算ソフトで集計した結果を壁に貼り出したアナログのチャートにプロットしていた。

E 社事例(6)では、イテレーションバーンダウンチャートにチームメンバーの気持ちやチームのルールを記載していた。これによって、バーンダウンチャートがコミュニケーションスペースになっていた。
Redmine[3]などのソフトウェアツールと異なり、目につくところに貼り出すことで自然とメンバーの視野に入り、チームメンバーのやる気につながっていた。

G 社事例(9)では、プロダクトオーナーがリリースバーンダウン棒グラフを書き、要求の追加と削除の状況まで把握していた。

G 社事例(9)(10)ではイテレーションバーンダウンチャートにイテレーションのテーマと、各日に何があったかを書出しておいた。これにより、バーンダウンチャートを見てイテレーションで何が起きていたかのふりかえりを容易にしていた。

I 社事例(15)ではリリース日までのリリースバーンダウンチャートを使って、日々の状況を反映させて残時間ベースで管理をしていた。

J 社事例(17)(18)では、イテレーションバーンダウンチャートを写真で撮って、共有サーバー上にアップロードし、遠隔地にいるプロダクトオーナーと毎日共有していた。

関連プラクティス

リリース計画ミーティング
イテレーション計画ミーティング
ベロシティ計測
スプリントレビュー
タスクボード(タスクカード)

[3] オープンソースのチケット管理システム
http://www.redmine.org/

3.1.12. **柔軟なプロセス** / Flexible process

“状況や目的にあったプロセスを考えよう”

別名：なし

要約

現時点でのプロセスが最善とは限らない。そのため、自らプロセスを改善していく。

状況

アジャイル型開発を現場に導入したことで、環境および内部の状況が刻々と変わっている。

開発には、様々な業務領域（ドメイン）があり、チームメンバーの状況も異なる。同じドメインであったとしても、アジャイル型開発のスタイルやプロセスは現場によって異なる。例えば、同じコードとサービスであっても、サービスのリリース前か後かによってスタイルが変わってくる。リリース前はイテレーション単位でのフィードバックを想定していても、リリース後であれば、イテレーション単位とは関係なく、利用者からの指摘や社会的な事象などへの対応が必要となる。このように、刻々と変わる状況によって適切なアジャイルのプロセスは違う。

問題

アジャイル型開発をしているが、しっくりこない。組織の状況と目的に対して適切な状態になっていない。

フォース

◆組織や状況、目的に合わせた開発スタイルにしたいが、適切なプロセスがわからない。
◆書籍などの文献を調査し、文献に記載されている方法のまま実施してみたが、経営や市場、ステークホルダーなどが有している問題状況に合わない。

解決策

ふりかえりを利用し、プロセスを柔軟に変更する。正解を探すことよりも、常に改善し続けることが大切である。

ふりかえりの KPT （Keep Problem Try）であれば、Try で検討されたことのうち、実施の可能性が高く、効果が期待できることから試してみる。

ファシリテータ（スクラムマスター）や、社内・社外のアジャイルコーチ、他のチーム、または関係者の意見を仰ぐ。

ふりかえりの Try に上がったことのうち試したことは、次回のふりかえりで評価するとよい。経験を積んだ後は、状況を意識し、トラブル発生時のモードやリリース前のモードなどを状況に応じて使い分けられるようになる。

組織に合わせたアジャイルスタイルを目指そう。

留意点

◆あまりに柔軟に変更すると、その変化に追従できないメンバーが出てくるため、チームで相談しながら進めること。特に「声が大きい人」は、他の人の意見に謙虚に耳を傾けるようにする。
◆検討されたすべての修正案を改善するのではなく、できるところから変更すること。
◆本来の目的や、変更する理由を問い続けること。

効果

◆組織が有する問題解決に適した納得の高いプロセスに成長してくる。

利用例

柔軟なプロセスは、全体の 58%の事例で適用されていた。契約別の切り口で見ると、自社開発(71.%)と比べて、受託開発(36.%)での適用率が低かった。ふりかえりの適用率が 90%を越えていたのに対して、柔軟なプロセスの適用率が低いということは、ふりかえりの中で、プロセスの変更について検討していなかったことが推測される。

B 社事例(2)では、ふりかえりを活用し、プロセスを含めてどんどん変えていた。また、スクラムマスターやアジャイルコーチがいなかったため、プロセスをチームメンバーらが自らで作っていった。合わなかったら即変更し、実際に効果があるかどうかわからないものは「とりあえず試す」ことができた。

D 社事例(5)では、うまくいっていない問題点をふりかえるようにしている。ふりかえりの Try

を参考にプロセスを変えた。

H 社事例(12)では、複数のチームがあり、それぞれに特色があるため、チーム毎にプロセスをカスタマイズすることを許容した。

H 社事例(14)では、オフショア先のプロセスを改善するため、オンラインミーティング、画面共有などを用いている。

L 社事例(21)では、開発対象領域（ドメイン）によってプロセスを柔軟に変更した。例えば、テストを先に設計したり、事前の内部設計を手厚くしたり、様々であった。

M 社事例(25)では、柔軟に変更することが推奨されているとは言え、複数のチームが勝手にプロセスを変更すると、中にはスクラムの根幹を揺るがしかねない変更をするチームが出てきかねないため、各チームの好き勝手にはできないようにしている。（例：紙・手書きツールをデジタルツールには戻さない。スクラムのフレームワークを超えて改善するところは止める。メンバーは朝10時には会社に来るようにする。ユニットテストは書く。コードレビューをする。）
これらの行動規範（Working Agreement）を定期的に評価し、プロセスをかなり柔軟に変更している。

関連プラクティス

ふりかえり
ファシリテータ（スクラムマスター)
アジャイルコーチ
組織に合わせたアジャイルスタイル

3.1.13. **ユーザーストーリー** / User stories are written

“すべてがストーリーから始まる”

別名: ストーリーカード

要約

ソフトウェアで実現したいことを、顧客の価値を明確にして簡潔に表現し書き出す。その結果、開発者とプロダクトオーナーの会話を促進することができる。

状況

ソフトウェアで実現したいことを語ることができるプロダクトオーナーや顧客が存在する。

問題

要求を明確に伝えようと、ドキュメントを詳細に書いたとしても、ドキュメントを通してのコミュニケーションでは、憶測や誤解を招きやすい。

要求が変更された時にドキュメントが更新されなければ、ドキュメントは誤った要求を伝え続けることになる。

ドキュメントを作り込めば作り込むほど、要求が変わった時のドキュメントの変更箇所が多くなり、ドキュメントの維持に多大なコストを要する。
また、膨大なドキュメントをベースに計画を立てるとなると時間もコストも多分に必要となる。

ソースコードレベルの詳細をドキュメントに記述している場合は、変更が発生する度に、ソースコードとドキュメントを二重に変更しなければならず、変更コストが増大する。

フォース

◆要求をある時点で詳細にしても、開発が進む中で、まだ着手していない要求に変更が起きる可能性がある。その場合、要求を詳細化するために使ったコストがムダになる。
◆要求を機能で定義すると、顧客にとってなぜそれが必要で、どのような価値があるのか見失ってしまう。その結果、必要のない機能を作り込んでしまう可能性が増える。

解決策

要求は、顧客の価値を簡潔に表現するようにしよう。システムをが必要な人と、作る人の会話を促進させるために、「だれのために」「何をしたいのか」「なぜそうしたいのか」という構成で書き出すようにする。

ユーザーストーリーのテンプレートでは以下のものが有名である。[ラスマッソン 2011]
　<ユーザー／顧客>として
　<XXX を達成>をしたい
　なぜなら<理由>だからだ。
顧客がソフトウェアを必要とする「理由」は、顧客にとっての価値を表すことなる。

ユーザーストーリーを書出す時点では、それほど詳細化できない場合もある。粒度の粗いユーザーストーリーはエピック（叙事詩）と呼び、粗いままで記載しておき、必要になった時点で詳細化する。[コーン 2009]

よいユーザーストーリーの特徴として以下の項目があげられる。 [ラスマッソン 2011]

◆ユーザーストーリーはお互いに「独立している（Independent）」。独立していることで、ストーリーのトレードオフが可能となる。
◆ユーザーストーリーは顧客と「交渉可能である（Negotiable）」。要求の実現手段を検討することができる。
◆ユーザーストーリーは顧客にとって「価値がある（Valuable）」。
◆ユーザーストーリーは「見積り可能である（Estimatable）」。
◆ユーザーストーリーは「テストできる（Testable）」。テストが書けなければ、ストーリーを何で持って完了とみなすか判断ができない。

これらの頭文字を取って、**INVEST** と称する。

また、ユーザーストーリーを表す言葉として、3C がある。

◆Card ストーリーはカードに書く
◆Conversation ストーリーは会話の約束
◆Confirmation 受入テストによってストーリーが実装されたか確認する。

これも頭文字を取ったものである。

ユーザーストーリーを書くのはプロダクトオーナーの役割とされるが、開発チームが要求を

聞きながら、書き出しても良い。その際は、関係者が集り、ユーザーストーリーを収集するワークショップを開催する。

ユーザーストーリーには満足条件が必要である。何を持って、ユーザーストーリーが意図通り実装されているかを確認する必要がある。この満足条件が明らかでないと、開発者はゴールが定まらず作業に迷ってしまう。満足条件は顧客価値の表現とは別に、ユーザーストーリーが見積られるまでに記述されている必要がある。

ユーザーストーリーは、プロダクトバックログの項目として使われることが多い。

ユーザーストーリーを書く前にインセプションデッキによってプロダクトの目的やビジョンを共有し理解しておくことが望ましい。

プロダクトオーナーと開発者が一緒にユーザーストーリーを記述する際には、情報カードのような紙・手書きツールを用いると、開発者全員で一斉に書きだしたり、机の上に並べて全体像を把握しやすくなるため便利である。

ユーザーストーリーの満足条件を元に受入テストを作成する。

ユーザーストーリーを使って全体像を俯瞰したい場合や、スコープ調整をしたい場合には、ユーザーストーリーマッピングを利用するとよい。

留意点

◆ユーザーストーリーに記述することは「だれのために」「何をしたいのか」「なぜそうしたいのか」という必要最低限の内容である。実際に開発するためには、この内容をベースにして、顧客と会話をしてより詳しく要求を理解しなければならない。
◆開発者がユーザーストーリーに着手する時点で、規模が大きくなりすぎてはいけない。開発前に必ずイテレーション内で実現可能な大きさに分割する必要がある。
◆ユーザーストーリーを分割する際には、必ず顧客価値の単位で分割する。ソフトウェア開発の単位(画面、データーベースアクセス、など)で分割してはならない。[ラスマッソン 2011]

効果

◆ユーザーストーリーとしてまとめることで、重厚なドキュメントを作成して進めるのに比べて、要求の全体をつかみ、理解するのに必要とするコストや時間を抑えることができる。
◆ユーザーストーリーは必要になった時に開発者がプロダクトオーナーと会話し、内容を詳細に聞く。その結果、ドキュメントを作りすぎるムダが生じることがない。

利用例

ユーザーストーリーは、全体で 71%の事例で適用されていた。契約別の切り口では、自社開発での適用率が 82%であったのに対して、受託開発では 55%であった。顧客側からユーザーストーリーの利用を拒否された事例もあり、利用することの価値を理解してもらった上での適用が重要である。

B 社事例(3)ではプロダクトの方向性をチームで理解しながらユーザーストーリーを書き出している。書き出すのは、プロダクトオーナーではなく、開発チームメンバーが行っている。

G 社事例(9)では、プロダクトオーナーがユーザーストーリーを書いて開発者に提示した。ただし、プロダクトオーナーと開発者が会話を進める中でユーザーストーリーの分割の必要が生じたり、別のものに書き直したりしなければならない場合は、全員で書き直していた。満足条件はプロダクトオーナーが付箋に書出し開発者に提示していた。

G 社事例(10)では、プロダクトオーナーにユーザーストーリーの書き方を教えて、ユーザーストーリーを書いてもらうようにした。当初は書きぶりが曖昧だったので、修正しながら進めていった。
ユーザーストーリーを半日～1 日かけて収集した上で、ユーザーストーリーマッピングを行っていた。

K 社事例(20)では、ユーザーストーリーが抽象的過ぎたため、当初はプロダクトオーナーに受け入れられなかったが、機能や仕様のレベルまで詳細にすることで、会話ができるようになった。

関連プラクティス

プロダクトバックログ
インセプションデッキ

受入テスト
ユーザーストーリーマッピング

3.1.14. スプリントバックログ / Mutual commitment to sprint backlog between product owner and team

“スプリントバックログで開発を駆動する”

別名：なし

要約

イテレーションで何をするべきか、スプリントバックログにリストアップする。イテレーションの残タスクを追跡することによって、進捗を把握することができる。

状況

開発を一定期間のサイクル（イテレーション）で繰り返し行っている。

プロダクトバックログでプロダクトについて開発すべき項目を管理している。

問題

そのイテレーションでどのプロダクトバックログアイテム（3.1.16 参照）を開発するのか、どのような手順で実現していくのか、必要な作業は何かが明らかになっていない。

イテレーションの期間中に実施するタスク内容とそれに必要な時間が管理できていなければ、開発の進捗状況がわからない。

イテレーションの中で実施するタスクを管理していなければ、どれだけの作業を終えればイテレーションのゴールが達成されるのか誰にもわからない。

フォース

◆プロダクトバックログのままでは、項目を開発完了にするために必要なタスクが見えない。
◆あまりに早くプロダクトバックログを詳細化してしまうと、変更があった場合にムダになってしまうかもしれない。

解決策

プロダクトバックログからイテレーションのゴールに必要な項目を選択し、その項目を実現するための作業を洗い出し、見積り、作業を管理しよう。

スプリント計画ミーティングで決めたイテレーションのゴールを達成するのに必要なタスクを洗い出し、スプリントバックログで管理を行う。

プロダクトバックログで管理している項目を開発するために必要なタスクに分解する。

スプリントバックログのそれぞれのタスクを完了するために必要な時間をチームで見積もる。

タスクの見積りが難しい場合は、スパイク・ソリューションを行う。見積り対象のタスクは、スパイク前は大雑把な見積りをしておく。スパイク実施後に、タスクの見積りを更新する。

タスクの見積りにはプランニングポーカーを利用することができる。

タスクのサイズは、1 人の開発者が 1 日で完了できるサイズ以下にしておく。これ以上のサイズになる場合はタスクの分割を検討する。

スプリントバックログへのタスクの追加や削除、見積りとその更新はチームの責任で行う。スプリントバックログの内容の変更は日次ミーティングで共有を行う。

スプリントバックログは、タスクボードとして書出しておきチームで共有する、あるいはチケット管理システムや表計算ソフトを利用する。

日次ミーティングで、スプリントバックログの合計残作業時間を追跡する。このとき、バーンダウンチャートを使うことで見える化できる。

留意点

スプリントバックログは開発チームの責任で管理すること。プロダクトオーナーや他のステークホルダーが勝手にタスクを追加したり、変更したりしてはいけない。イテレーションのゴールを達成するという開発チームのコミットメントを維持できなくなるからである。

効果

◆チームが、イテレーションで何を開発するべきか、タスクとして何が必要かを明確に理解

できるようになる。
◆タスクの残作業時間を計測するため、イテレーションの進捗を管理することができる。

利用例

スプリントバックログは、全体の 77%の事例で適用されていた。利用していないと回答した事例も、実際にはタスクボードに必要な作業を洗い出して、見積り、管理しているところが多かった。

A 社事例(1)では、スプリントバックログの管理を最初は付箋で行っていたが、マネージャの出張が多く、共有しにくかったため電子化するに至った。

B 社事例(2)では、プロダクトオーナーとメンバーの代表者（交代制）が事前に話をして、タスクのざっくりした見積りと優先順位を決めておく。その後、チームメンバー全員で集まって話し合うようにした。管理は Redmine を使っている。タスクの見積りは行っていない。

D 社事例(5)では、アナログのタスクボードでスプリントバックログを管理していた。デジタルなツールは使っていない。

G 社事例(9)では、スプリントバックログは付箋に書出してタスクボードに貼り出していた。
G 社事例(10)では、タスクに書出しタスクボードに貼り出しておいたが、Redmine にも転記していた。

H 社事例(14)では、スプリントバックログは開発チームで管理しており、プロダクトオーナーには見せていない。プロダクトバックログに上がっているユーザーストーリーのレベルで開発チームとプロダクトオーナーが合意している。

K 社事例(20)では、スプリントバックログの代わりに WBS（Work Breakdown Structure）を使ってタスクの管理をしている。
新しく発生したタスクを追加した場合、ステークホルダー全員で共有するようにしていた。

関連プラクティス

プロダクトバックログ
イテレーション計画ミーティング（スプリント計画ミーティング)
日次ミーティング（デイリースクラム)
イテレーション（スプリント)
スプリントレビュー
バーンダウンチャート
タスクボード
プランニングポーカー

3.1.15. **インセプションデッキ** / Inception Deck

"10 の質問でプロジェクトの方向を明らかにする"

別名なし

要約

プロダクトの目的や方向性が曖昧のまま開発を進めている場合は、プロダクトの目的や方向性を明らかにした 10 の質問に回答しチームで共有する。その結果、チームはプロダクトの目的、ビジョン、方向性を理解して開発が進めやすくなる。

状況

プロジェクト開始直後は、プロジェクトのゴールやビジョンについて、関係者それぞれで思い描いていることが異なっている場合がある。この状況で開発を始めたとしても、出来上がった成果物の認識が大きくズレることになり、プロジェクトが破たんすることもある。プロジェクトを始める前に、関係者全員で、認識を合わせる必要がある。

問題

プロジェクトについての認識がステークホルダーの間でそろっていない。

ステークホルダーが揃っていないところで合意した事項をプロジェクトを進める上での前提に据えると、認識がズレたまま隣、結果的に目的にかなう成果物が出来上がらない。

フォース

プロジェクトが進んだところで、前提を問う質問をしても手遅れになっていることが多い。

解決策

関係者全員でプロジェクトについての 10 の質問と課題をベースに話し合い、互いの期待や認識を合わせる。

以下が 10 の質問である。[ラスマッソン 2011]

1. われわれはなぜここにいるのか
2. エレベータピッチを作る
3. パッケージデザインを作る
4. やらないことリストを作る
5. 「ご近所さん」を探せ
6. 解決案を描く
7. 夜も眠れなくなるような問題は何だろう？
8. 期間を見極める
9. 何をあきらめるのかをはっきりさせる
10. 何がどれだけ必要なのか

効果

◆プロジェクトの目的や背景についての方向性がステークホルダーの間で合致し、チームが状況に応じた適切な判断を下せるようになる。
◆ステークホルダーにプロジェクトについての情報を提供することになり、ステークホルダーがプロジェクトを続けるかどうかの判断を下せるようになる。

留意点

プロジェクトのステークホルダーをできる限り集めること。インセプションデッキの質問に答えられるメンバーが集まらなければ、認識合わせと正しい判断ができない。

プロジェクトの状況や方針に大きな変更が発生した場合は、その都度インセプションデッキを改訂すること。

インセプションデッキを作りあげたとしても、それを常日頃から意識しておかないと、単に作って終わりということになりがちである。掲示しておく、定期的に見直す、などの工夫が必要だ。

10 の質問は、プロジェクトにあわせて用意する。

利用例

インセプションデッキは、全体で 21 %の事例で適用されていたが、受託開発では事例が皆無であった。日本に本プラクティスが紹介されたのが 2011 年と比較的新しいことが要因であると推測される。

B 社事例(22)では、10 の質問のうち前半 5 つの質問を作成している。プロジェクト立ち上げ時の意識合わせに効果的であった。

H 社事例(11)では、主に 10 の質問のうち前半 5 つの質問の部分を用いていた。

K 社事例(20)では、インセプションデッキを実際にまだ使ってはいなかったが、「われわれは

なぜここにいるのか？」という問いかけが重要であると感じている。「何のために」「何ができるか」を明らかにしてこないプロジェクトはこれまで経験上うまくいっていなかったため、これから導入したいと述べていた。

しかしながら、一方ではあまり有効でなかった、という意見もある。

A 社事例(1)では、インセプションデッキを書いてはみたが、書いただけて終わってしまい効果を得ることができなかった。

D 社事例(5)では、インセプションデッキを何度か使ってみたが、うまくいったことがなかった。最終的には、自社サービスであるため、プロジェクト憲章のようなものは必要ないという結論に至った。

関連プラクティス

チーム全体が一つに

3.1.16. **プロダクトバックログ**/ Priorities (product backlog) maintained by a dedicated role (product owner)

“プロダクトが存在する限り、プロダクトバックログは不滅である。”

別名：マスターストーリーリスト

要約

プロダクトとして何を作るべきかをプロダクトバックログで管理し、その優先順位をチームとの会話を踏まえ、プロダクトオーナーに判断してもらう。その結果、チームは何から開発を行えばよいか明確になる。

状況

要件に責任を持つプロダクトオーナーや顧客が役割として存在する。

開発を一定期間のサイクル（イテレーション）で繰り返し行っており、要求の優先順位や、新しい要求などの変化が激しい。

問題

プロダクトを開発する上で、何から作業に取り掛かればよいかわからない。

プロダクトに関する知識が不足している場合に陥りやすい状況である。

変化が激しいと、どんどん優先順位が変わる。

全体の精緻な計画を立てたとしても、すぐに変更になるため、計画を変更していくコストがかかってしまう。

フォース

◆プロダクトに対する要求の優先順位を付けたいが、様々なステークホルダーからの多様な要求により、優先順位付けが困難である。

解決策

プロダクトに必要な項目や作業を、リスト化、及び順序付けをして管理して、優先順位の高いものから作業を行っていく。これをプロダクトバックログ(PBL)と呼ぶ。

プロダクトバックログは、要求、要望、修正など、その時点でプロダクトに必要だとわかっていることはすべてリストアップする。プロダクトバックログに含まれている項目のことを、プロダクトバックログアイテム(PBI)と呼ぶ。

プロダクトを取り巻く状況によって、プロダクトバックログの内容も常に変わる。必要な項目は増え、不要になった項目はリストから外す。この作業をプロダクトバックログの手入れ(グルーミング)と呼び、プロダクトオーナーは定常的に手入れを実施する必要がある。また開発者と共にワークショップ形式で定期的に実施するとよい。[ピヒラー 2012]。

プロダクトバックログの内容（項目と優先順位）はプロダクトオーナーが責任を持つ。

優先順位付けは、項目の金銭価値（その機能によってどれだけの収益があげられるか）や開発コスト、開発による獲得知識（開発によって何を学ぶか）、リスクを勘案して行う。

価値とリスクの組み合わせから優先順位付けを決める。高価値で高リスクなものは早めに取り掛かったほうがよく、次いで、高価値で低リスク、低価値で低リスクという順番で開発を行うことが望ましい。低価値、高リスクは開発するべきではない。

優先順位を付ける際には、狩野モデル（プロダクトの品質を検討するための枠組みの一つ)を利用することもできる。[狩野 et al. 1984][コーン 2009]

狩野モデルでは、その機能が「存在してあたり前」の必須なのか、「あればあるほど良い」ものなのか、「魅力的な、わくわくする」ものなのかで評価し、順番を付ける。

チームはプロダクトバックログの１番上の項目から開発を始める。優先順位の高いものから、内容を詳細かつ明確にする。

プロダクトバックログに責任を持つプロダクトオーナーはただ１人とする。チームが、拠り所にする基準を一つにすることで、開発すべき内容や、優先順位を誤らないためである。

留意点

◆プロダクトバックログの優先順位付けは、開発チームとプロダクトオーナーが協力して

行うこと。技術的なリスクなどから、チームからの提案や意見を優先順位付けの参考にするべきである。

効果

◆開発チームが、何をどれから開発するべきか明確に理解できるようになる。
◆プロダクトにとって何が重要なのかが順序づけされ理解しやすい。
◆変化に対して適応するのが容易である。

利用例

プロダクトバックログは、全体の 71%の事例で適用されていた。特に中大規模では 90%以上が適用していたが、受託開発での適用は約 50%に滞っていた。

B 社事例(2)では、プロダクトバックログの順序について、最終的な判断はプロダクトオーナーが行っていたが、管理はチームメンバーが代行していた。イテレーション毎に、メンバーがプロダクトバックログを整理し、イテレーション計画ミーティングの前日にプロダクトオーナーと意識合わせをして、ミーティングの場で、全員で優先順位含め内容を合意するというやり方を取っていた。優先順位の基準は、インセプションデッキで作った際の、プロダクトのビジョンを用いた。

C 社事例(4)では、最低限なくてはならない必須機能と、あれば良いという機能を明確に分けて、プロダクトバックログで管理した。最初は、欲しいものから項目を洗い出し、プロジェクトの途中からは、予算から判断して必要な機能に絞るようにした。

G 社事例(9)(10)では、プロダクトバックログアイテムの管理に、ユーザーストーリーマップ（ユーザーの行動に即してユーザーストーリーを洗い出して整理した図）を主として活用している。プロダクトオーナーとはユーザーストーリーマップを使って、コミュニケーションを取っていた。

K 社事例(20)では、プロダクトオーナーに優先順位を付けてもらおうとしたが、「すべてが揃わないとリリースできない」との言及があったため一元的な順序を付けることができなかった。その代り「後回しにできる」という項目を洗い出して判断してもらい、スコープを調整することができた。

関連プラクティス

プロダクトオーナー
スプリントバックログ
リリース計画ミーティング
イテレーション（スプリント）
スプリントレビュー
ユーザーストーリーマッピング

参考文献

[狩野 et al. 1984] 狩野紀昭、瀬楽信彦、高橋文夫 、辻新一 (1984) 「魅力品質と当たり前当り前品質」『品質』 14(2)

3.1.17. **迅速なフィードバック**/ Rapid feedback

“適切なフィードバックを得よう。”

別名: なし

要約

アウトプットが適切かどうかわかりにくい時は、迅速なフィードバックを心掛ける。その結果、現状把握と軌道修正がしやすくなる。

状況

サービスやプロダクト、ソースコードやドキュメント、価値や情報などの種別を問わず、何らかのアウトプットしている、もしくは、アウトプットを心掛けている。

個人および組織のアウトプットには様々な種類がある。日次ミーティングでは、状況や持っている問題などの情報をアウトプットする。プログラマは、ソースコードをアウトプットする。小売業者は、サービスやプロダクトをマーケットに提供する。

開発チームはアウトプットに対して、プロダクトオーナーやユーザーからフィードバックを得ている。フィードバックを得ることで、次のアウトプットへ影響を与えることができている。例えば、デイリーミーティングで報告された問題に対して、何らかの対策を検討し、試してみる、などである。

問題

ソースコードから仕様やユーザーストーリー、プロダクトやサービスに至るまで、あらゆるアウトプットについてフィードバックがないと効果や価値があるのかわからない。

フィードバックがあっても、遅すぎると活かすことができない。

フォース

◆フィードバックを得ているが、修正するための工数をさけない。
◆フィードバックを得ているが、時間経過での変化に気がつかない。

解決策

迅速なフィードバックが得られるように、評価と修正が軽量に実施できるような仕組みを作る。

コミュニケーションについてのフィードバックや、製品やサービスとそれを関連するコンセプトからソースコードに対するフィードバックなど、様々な種類のフィードバックに関するプラクティスがある。

例えば、アジャイル型開発には、コミュニケーションに関する次のようなプラクティスがある。

◆リリース計画ミーティング
◆イテレーション計画ミーティング
◆スプリントレビュー
◆ふりかえり
◆日次ミーティング
◆共通の部屋などによる日常的な会話

日次ミーティングで得られるフィードバックの内容や量が充分ではない場合は、1日に3回のミーティングを行うようにする。

コードが適切に書かれているかを迅速に確認したい場合は、テストの自動化や継続的インテグレーション環境の構築が、迅速なフィードバックをサポートする。

プロダクトオーナーやステークホルダーとのコミュニケーションによるフィードバックを増やしたいのであれば、障害や課題レポートや、進捗状況をオンラインで公開しておくことによって、状況把握が速やかになる。

製品やサービスの構成について、顧客の反応がわからないのであれば、一部の機能ができた段階で、限られたマーケットにプロダクトやサービスを実際に出してみて、早めにフィードバックを得てみる。

コミュニケーションに溝や断絶を感じた場合は、ここで述べたように様々な種類のフィードバックに着目し、試してみる。

留意点

◆価値や評価は、それぞれの文化やコンテキストに基づいていることに留意すべきである。
◆組織は、ある程度の規模になると適切なフィ

ードバックを得にくくなる。
◆フィードバックの場を用意したものの、単に”声の大きな”人の発表の場になっていることがある。例えば、全員で付箋に書くなど、小さな声も尊重するような方法を選ぶようにする。
◆フィードバックを与える側と得る側の立場や認識の違いにより、意図したものとは違う伝わり方をすることがある。
◆日次ミーティングや、一定期間のサイクル（イテレーション）では、フィードバックが充分ではない場合には、回数を増やすなどの検討をする。
◆フィードバックの量が多すぎると、本来の開発やビジネスの進捗に影響を及ぼすため、短時間に終わらせたり、頻度を減らしたりすることも検討する。
◆批判的なフィードバックをすると萎縮して意見が出なくなることがあるが、肯定的なフィードバックであれば、様々な意見が発展するきっかけになる。特に企画段階では、肯定的な姿勢でさらに意見を引き出すよう心掛ける。逆に、プロダクトとしての出荷時には、5回「なぜですか?」と問うなど批判的なフィードバックを心掛ける。

効果

◆プロダクトオーナーや顧客など情報を知りたい人が安心することができる。
◆日次ミーティングを1日に3回行うなど回数を増やすことによって、状況の把握に役立つようになった。

利用例

迅速なフィードバックについては、全体の73%の事例で適用されていたが、社内システムや、受託開発においては50%程度と適用率が低かった。開発を反復的に実施していても、最終的な利用者からのフィードバックが迅速ではなかったためであると推測される。

B社事例(2)では、共通の部屋やオンサイト顧客などのプラクティスで迅速なフィードバックを得られるようにし、開発チームとプロダクトオーナーのコミュニケーションを密にしたところ、緊急対応などの無理な要求が発生して、チームが要求の対応に追われてしまい、結果的に開発チームが集中できない状況になってしまった。

C社事例(4)とH社事例(14)、L社事例(21)では、スプリントレビュー時にフィードバックを積極的に得て、次のイテレーション計画ミーティングのインプットにしている。

D社事例(5)では、スプリントレビューは1時間以内としている。午前はスプリントレビューとふりかえりに、午後は次のスプリントのイテレーション計画ミーティングに充てている。

F社事例(8)では、カナリア環境と呼ぶ実行環境を用意している。希望ユーザーに機能を本実装する前のプロトタイプを利用してもらい、その機能についての要望を収集しフィードバックを得ることができる。

G社事例(10)では、スプリントレビューにおけるデモンストレーションを含むプレゼンテーション準備にレビュー前日から工数を割いており、そのために開発時間を削り残業をして準備をしていた。レビューの時間よりも開発自体に時間を割いて欲しいとの思いから、レビュー当日に準備の時間を確保して準備の負担を軽減するよう顧客から依頼があった。

J社事例(17)では、発注元との連携はうまくいっているが、実際のエンドユーザからのフィードバックがうまくいっていない。

J社事例(19)では、SNSコミュニティで発注元と情報を共有している。

関連プラクティス

日次ミーティング
ふりかえり
スプリントレビュー
自動化された回帰テスト
継続的インテグレーション

3.2. 技術・ツール

3.2.1. ペアプログラミング / Pair Programming

“知識共有、品質向上、集中力持続、達成感の増幅を実現”

別名：ペアワーク、ペアリング

要約

チームの中で知識やコードの共有ができていない場合は、ペアを組んで作業をする。その結果、開発チームの中で業務知識やコードについての知識が共有でき、品質や作業効率も向上できる。

状況

個人でできることは限られているため、開発メンバーはチーム一丸となって仕事に取り組んでいる。

問題

知識をより素早くチーム全体に広げたい。

チームで一丸となって目標に取り組んでいる時に、実際に開発しているプロダクトコードの詳細をじっくりと読む機会がないため、自分以外の人の作業内容を理解するのは困難である。

個人のスキルにより作業の質にばらつきがある。

また開発者自身で決めた開発ルールがどうしても徹底されない。

フォース

◆ある作業について深く理解するためには、結果だけでなく経緯を知る必要がある
◆話を聞くことと、その内容を充分に理解することは別である
◆チームのパフォーマンスは個人の能力の総和以上であることが望ましい

解決策

プロダクトコードをペアで開発する。

プログラミングだけでなく、重要な作業もペアで実施するとよい。

ペア作業はドライバー（キーボードの前に座り主導権を握る役割）と、ナビゲーター（ドライバーの作業に注意を払い、かつ先の仕事の戦略を考える役割）に別れて、一つの画面に２人で向きあって実施する。

ドライバーは手を動かしながら作業に集中しているが、ナビゲーターは一歩引いた観点で作業を俯瞰して、ミスをいち早く発見し、共に問題解決に取り組む。

ペアプログラミングは、「常に誰かに監視されている」状態ではなく「常に誰かと一緒に問題解決に取り組む」状態である。

ペアは頻繁に役割を交代することで、互いのスキルを高めることができる。

チームが行っている作業をできるだけ全員で共有しておく。難問もペアを交代することで解決できる場合があるため、ペアの組合せを１日に何度も交代するのが望ましい。１日に数回ペアを交代する場合は、時間を決めて強制的に交代するのがよい。

ペアで作業するかしないかを状況によって区別するのが現実的である。しかし、ペアで作業していないと、品質や情報共有の質が下ってしまうことを念頭に入れておくこと。

ペアプログラミングは１人分の仕事を２人で行うのではなく、仕事を２人でやることによって、１人前の仕事をより効率的に、手戻り少なく進める方法である。

留意点

ペアプログラミングは非常に頭を使うため、1

日 6 時間程度が限度であることが多い。

ペアの間にスキルギャップが大きい場合、師匠と弟子のような一方的な上下関係ができてしまうとうまくいかないことがある。教育目的で実施する際にも、互いにフィードバックできるように、ドライバーとナビゲーターを交代しながら作業をすること。

ペアを組む個人が、特定のツール、キーボードなどにこだわってしまうとうまくいかない。

作業環境がペア作業に適さない場合（席が非常に狭い、いすを移動させることが困難、物理的に離れているなど）は実現が難しい。

抵抗する開発者にペアプログラミングをむりやり強制してもうまくいかない。まずはお試し期間を設けて実施してもらい、その上で採択の判断をする。

ペアプログラミングが実施できるようになっていれば集団によるオーナーシップが促進される。

効果

◆作業の結果についての質が高まりやすい
◆難しい問題が早く解決しやすくなる
◆作業やコードの内容についての理解がチームに広まる
◆規律を守りやすくなる
◆開発が楽しくなる

利用例

ペアプログラミングは、全体の約 50%の事例で適用されていた。受託開発では 60%の適用率だったのに対して、自社開発では 40%未満と適用率が低かった。また手法別では、スクラムでは 30%程度だったのに対して、XP では適用率が 90%に達していた。

B 社事例(2)では、必要に応じて各個人でペアを探してペアプログラミングを実施していた。ただし、難しい部分は必ずペアで作業を行うようにしていた。

個々人の成長を促すために、あえて 1 人で設計・コーディングしたものを、後でレビューするということも並行して実施していた。1 人で実施→レビューというプロセスだと時間はかかるが、個々人のスキルアップも重要ということで、ペアプログラミングに固執しないで行っていた。

うまくいかなかった例としては、ペアプログラミングに入る前に、対象についてよく知っている人が 1 人で事前調査をしていたことがあった。これだと時間はとられるしペア同士の協調作業にはなっていなかった。事前調査は行わずに、調査もペアで行ったほうがよい。

C 社事例(4)では、ペアプログラミングをスキルトランスファの目的で部分的に実施していた。特にプロジェクトに入ってきたばかりの人や、スキルレベルの低い人を中心にペアを組み合わせて行った。

E 社事例(6)では、すべての開発をペアプログラミングで実施していた。ペア決めのために「ペアプロスロット」を制作して、当日そのスロットで抽選してペアの組合せを決めていた。
そのため、自然と開発者全員がソースコード全体を見ることになり、コードは全員の所有物であるという意識が生まれた。

L 社事例(21)では、一部でペアログラミングを実施していた。主に教育的な意図でベテランとビギナーの組合せで行い、ビギナーがプログラミングしたコードの品質の向上に効果があった。

関連プラクティス

集団によるオーナーシップ

3.2.2. **自動化された回帰テスト** / Automated regression test

“修正がある度に全テストを実施できる”

別名：リグレッションテスト

要約

システムを修正する度に回帰テストを手動で実施しなければいけないなら、回帰テストを自動化すること。その結果、回帰テストにかかる工数が大幅に削減でき、何度も実施できるようになる。

状況

何回もイテレーションを繰り返して実現した機能が積み重なっている。新しく機能を追加する際には、必ず既存の機能が正しく動作しているかを確認するために、それまでに実施してきたテスト群を回帰的に実施する必要がある。

問題

テストは何度も繰り返し実施しなければならないため、ミスなく、短時間で終えられるようにする。

イテレーションの度に増えていく回帰テストを何度も実行する必要がある。回帰テストを手動で実施していると、イテレーションの度にテストが増えていき、テストにかかる時間をとられる。

テストに時間がかかってしまうと、繰り返し実行するコストが膨大になる。そのため、いずれはテストが実行されなくなる可能性もある。

フォース

◆回帰テストは機能を追加する度に実施したいが、その度にテスト工数がかかる。
◆回帰テストは、短期的視点よりも長期的視点で考えないといけない。

解決策

様々なテストを自動化して回帰的に実行できるようにする。

ユニットテストだけでなく、ユーザー機能テスト、GUI からのテスト、など可能なものは自動化すること。

回帰テストのメリットは、短期的な視点で見た場合よりも、長期的視点で見た場合のほうが圧倒的に大きい。短期的視点で見ると、回帰テストは手動による実行で賄えると考えてしまいがちであるが、時間がたてばたつほど、自動化するための障害（自動化にかかる工数、自動化しにくい設計）が増えていくため、初期投資と

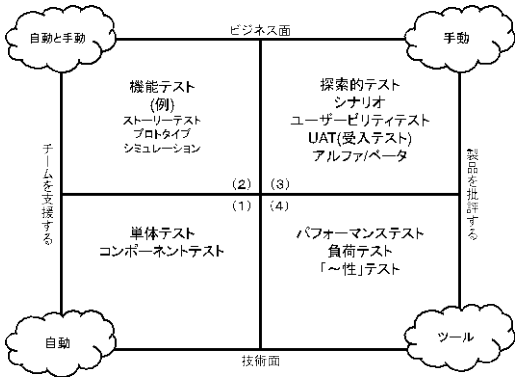


図 3 アジャイルテストの四象限

して早期から自動化に取り組むのが望ましい。

図 3 アジャイルテストの四象限[クリスピン et al. 2009] の通り、自動回帰テストの対象となるのは主に（1）、（2）の領域である。（1）の領域はユニットテストの自動化を実施することで対応可能になり、（2）の領域はユニットテストとは別に、機能テストの自動化が必要となる。

ユニットテストの自動化や、受入テストの自動化ができると、回帰テストの価値が高まる。

自動化された回帰テストは、継続的インテグレーションにより頻繁に実施されるのが望ましい。

発見された障害は、バグ時の再現テストを作成した後に修正するのが望ましい。

留意点

◆回帰テストの自動化を後回しにしてしまうと、機能の実現が最優先となり、優先順位が下がってしまうことが多い。
◆GUI 部分テストの自動化はUI の見た目の変更によってテストが失敗する頻度が多い。そのためテストを修正するコストがかかる。

◆自動化できない領域のテストも実施すること。自動化は「壊れていないこと」を確認する目的でしかない。

効果

◆回帰テストの実施時間を大幅短縮できる。
◆既存の機能の不具合を早期発見できる。

利用例

自動化された回帰テストは、全体では65%の事例に適用されていた。手法別で見ると、スクラムでは40%程度であった適用率が、XPになると90%であった。

B社事例(2)では、元々テストを自動化する習慣がなく後回しになっていたが、開発の後半になり実施することにした。もっと早くやるべきだったが、他の優先事項のために後回しになってしまった。そのため、自動化が充分に実現できず、次回のプロジェクトでは最初から回帰テストを自動化することになった。

B社事例(3)では、プロジェクトの最初から自動回帰テストを実現する環境を構築しており、非常に効果があった。

D社事例(5)では、ユニットテストの自動化は実施していないが、Selenium[4]によるGUIベースのテストを自動化して回帰テストとして実施した。この結果バグ率が低くなった。

J社事例(17)では、元々回帰テストは自動化されてはいなかった。しかし修正があった場合に手動で回帰テストを実施するのは効率が悪く、途中から自動化を進めた。しかし、すべての観点を自動化された回帰テストで網羅するには至っていない。

L社事例(21)では、改善要望を取り込むためには、自動化された回帰テストは必須であった。

関連プラクティス

ユニットテストの自動化
受入テスト
バグ時の再現テスト
継続的インテグレーション

[4] Webブラウザ経由の自動テストツール
http://seleniumhq.org/

3.2.3. テスト駆動開発 / Test Driven Development

“テストが開発を駆動する”

別名: なし

要約

自動化されたテストコードを書き実行させながらプロダクトコードを育てていく。その結果、バグが混入してしまうことを防ぎ、プロダクトコードの変更容易性が向上し、開発者は自信を持って開発することができる。

状況

ユニットテストの自動化によって、開発メンバーはプロダクトコードを修正した時にテストを実行して壊れていないことを確認できる安心感を体験できている。

新規にソースコードを作成する際にも、テストを実行しながら、コードを壊さずに安心して開発がしたい。

ユニットテストコードには、クラスやメソッドがどう振る舞えばよいかを記述する。

問題

新しくプロダクトコードを書いている最中も、今書いているコードが正しい仕様に基づいているのか、バグを組み込んでいないか不安である。

プロダクトコード→テストコードの順番で新規で開発している場合、最初のプロダクトコードを書いている際にはテストコードは存在しない。そのため、テストコードを実行しながら、意図する仕様に基づいて動作するコードを書けているか、これまで書いてきたものを壊していないかを確認しながら進めることが難しい。

フォース

◆設計段階でテスト容易性を考慮した設計にしたいが、実際にテストコードを書くまでは、その設計が妥当かどうかわからない。
◆一度実行させたプロダクトコードに対してテストコードを書くことは開発者に対しての動機づけが弱い。
◆開発者はユニットテストが配備されるまで、プロダクトコードの正しい振る舞いかがわからず不安である
◆短期的視点では、テストコードを書くことは追加コストとして見なされてしまう。

解決策

プロダクトコードと並行してテストコードを書き、小まめに実行して結果を見ながら開発すること。できるだけ小さい単位で、プロダクトコードとテストコードの両方を少しずつ作り実行していきながら、ソフトウェアを育てていくこと。

テスト駆動開発(TDD)は以下のプロセスを何度も反復して実施していく。このプロセスをTDDマントラ[5]と呼ぶ。

1. テストコードと、テストの実行が可能な最小限のプロダクトコードを書き、テストを実行して失敗を確認する
2. テストが成功する最小限のプロダクトコードを書き、テストを実行して成功を確認する
3. テストが成功する状態を維持しながら、より整理されたコードにリファクタリングする（重複の除去、メソッドへの切り出し、など)
4. リファクタリング後にテストを実行し、成功していれば次の実装に移る→1 へ

テストコードを先に書く（テストファースト）ことによって、テストコードはプロダクトコードをどのように実装すれば良いのかという判断基準（テストが成功するように実装することがゴール）になり、小さなゴールを達成していきながら、プロダクトコードを作りあげていく。

結果として、期待する動作をするプロダクトコードと、その振る舞いを検証するテストコードが出来上がっていくことになる。

常にテストコードを成功させていきながら開発を進めていくことができる。

[5] テスティングフレームワークが表示する成功(緑)と失敗(赤)を元に「レッド→グリーン→リファクター」を繰り返すサイクルのこと

未経験者、初学者がTDDを実践する場合は、ペアプログラミングを推奨する。最初は開発速度が遅く感じるかもしれないが、ペアで実施していくことで学習速度が向上する。

テスト駆動開発によって、テストコードが配備されながら開発される。そのため、テストを継続的に実施するために継続的インテグレーションで早期の問題発見に役立つ。

常にテストコードが存在するため、安心して他人が書いたコードを修正することができる。そのため集団的オーナーシップを促進させる。

TDDに慣れていない場合、ついテストコードを忘れてプロダクトコードに集中してしまうことがある。ペアプログラミングによってTDDのリズムを意識しながら実施するとよい。

留意点

TDDを「最初にユニットテストケースをすべてテストコードとして記述してから、プロダクトコードを作成する」と誤解している場合がある。このようなプロセスは厳密にはTDDとは言えず、短く頻繁なフィードバックを得ることができない。

TDDはユニットテストの自動化、テストファースト、リファクタリングの知識が必要とされ未経験者だけで実践するのは困難であるため、TDDの経験者の支援が望ましい。

アジャイル型開発で変更を受け入れていく中で、ソフトウェアの設計も進化していく必要がある（進化的設計）。例えばメソッドのシグニチャ、クラス構造などがその対象である。進化的設計が許されない環境（事前設計を厳守すべき、など）でもTDDの採用は可能だが、効果が半減する。

テストコードもプロダクトコードも意図を明確に表現し、開発者にとって読みやすいコード（=リーダブルコード）を書くべきである。

TDDには、テストが快適に実行できることが不可欠である。ユニットテストの実行に時間がかかる環境（マシンが遅い、データベースが遅い、など）では、TDDマントラが素早く行えずにTDDのメリットを享受しづらい。

TDDを実践する際には、計画時点でTDDを行って開発する工数を考慮しておかないといけない。もしTDDを実践したことがない状態で見積りをする場合は、TDDで行った上でベロシティを計測し再度作業工数を見積る必要がある。

効果

- ◆常にテストコードをパスさせながら開発をするため、テストコードの範囲内においてはバグが混入せずに品質が向上する
- ◆テストコードとプロダクトコードを並行に作っていくため、ユニットテストコードが自然と揃っていく。
- ◆常に自動テストが実行できる環境で開発することになり、開発者が安心してプロダクトコードを修正することができる。
- ◆レッド→グリーン→リファクタのリズムで開発を行っていくと、テストコードを書くこと自体が開発者にとって負荷ではなく楽しみになる。
- ◆オブジェクト指向型言語を利用している場合は、プロダクトコードが以下のようなテスト容易性の高い設計[小井土 2003]になる。
  - ♦ クラス、メソッドの独立度が高い
  - ♦ メソッドが一つの機能を提供する
  - ♦ メソッドが結果を提供する
  - ♦ 特定の環境に依存しない

利用例

テスト駆動開発は、全体で59%が適用していた。そのうち「適用している」と答えた事例が10件だったのに対して、「部分的に適用している」という事例が9件であった。切り口毎にそれほど差異は見られなかったが、「必要と感じてはいるが、充分に適用できない」という回答が多く見受けられた。

A社事例(1)の場合は、技術プラクティスを特に重視していたため、TDDを充分に行っていた。

B社事例(2)の場合は、ユニットテストの自動化は必須としていたが、TDDで必ずしも開発を行っているわけではなかった。TDDで開発するかどうかが個人の意識にゆだねられており、その変革が難しかったが、ペアで作業することで、ペアがお互いに声を掛け合ってTDDを意識するようになった。

B社事例(3)の場合は、TDDで開発をすることでテストコードを書く楽しさを開発者が感じるようになった。効果については厳密には測れてはいないが、開発の際の安心感が非常に高いと感じていた。

C 社事例(4)の場合は、TDD を実施することが自己目的化することを恐れたため、あえて TDD を導入しなかった。

G 社事例(9)の場合は、当初は開発者がユニットテストの自動化及び TDD の両方を経験したことがなかった。チームで勉強会を開き、ペアで作業することで書き方を学んでいった。

J 社事例(16)の場合は、TDD を実施したいが、実際はできていなかった。その代りに詳細な仕様書を先に書くことがテストの代わりになっていると回答していた。

J 社事例(18)でも同様に「やってみたいが、実践できていない」という回答であった。

関連プラクティス

ユニットテストの自動化
リファクタリング
継続的インテグレーション
集団的オーナーシップ
ペアプログラミング

参考文献

小井土亮 (2003)
「テストパターンの有効性について」
http://www.jasst.jp/archives/jasst04/pdf/B1bp.pdf

ベック, K. 長瀬嘉秀監訳 (2003) 『テスト駆動開発入門』 ピアソン・エデュケーション

Freeman, S. Pryce, N. 和智右桂・高木正弘訳 (2012) 『実践テスト駆動開発 テストに導かれてオブジェクト指向ソフトウェアを育てる』 翔泳社

3.2.4. ユニットテストの自動化/ Unit Test Automation

“自動化されたユニットテストはシステムのセーフティネットである”

別名: デベロッパーテスティング

要約

動作しているソースコードに手を入れることに不安がある場合は、ユニットテストを自動化する。その結果、修正をしても、ユニットテストコードを実行して結果を見ることで、元のコードが壊れていないかを確認することができ、万が一不具合があった場合でもすぐにわかる。

状況

アジャイル型開発では、イテレーションの度に動作するソフトウェアを提供しなければいけない。そのためにはイテレーションの中で何度もユニットテストを実施する必要がある。

問題

ユニットテストを手動で実施するには、多くの問題がある。

一番の問題は、時間がかかり過ぎるということだ。ソフトウェアが大きくなるにつれて、テストにかかる時間が増えていく。つまりイテレーション毎にテストにかかる時間が増えていくということである。ユニットテストにかかる時間が増えるといことは、ユニットテスト以外に使える時間が減っていくということを意味する。

第二に、ユニットテストを手動で実施しようとすると、テスト手順書に従ってテストを実施することになる。人が行うテストは間違えやすく、さらには納期目前などのプレッシャーの高い場面では、間違える、あるいは手を抜く誘惑にかられることもある。

第三に、動作するソフトウェアを提供し続けるためには、既に作ったものを元に、機能を追加し続けていく必要がある。しかし、動作しているソフトウェアに手を入れることで、新しい不具合を混入してしまう可能性がある。そのため不具合が混入されていないこと、既存の機能に影響がないことを確かめるため、ユニットテストをその保証に使おうとすると、第一の問題として取り上げた「時間がかかる」という問題があるために、意図した目的にユニットテストを使うことができない。

フォース

- ◆変更のためソースコードに手を入れたいが、元の機能が壊れていないことが確認できない。
- ◆イテレーションの度に既存の機能に対するユニットテストを実施したいが、イテレーションを経る毎にテスト工数が増える一方である。

解決策

ユニットテストを自動化し、開発者がいつでも実施して結果がわかるようにしておくこと。

ユニットテストの自動化をサポートするフレームワークは、プログラミング言語毎に用意されていることが多い。(Java はJUnit、C#はNUnit、Ruby はTest::Unit、RSpec、など)。それらのフレームワークを用いてユニットテストで検証したい項目を、動作するテストケースとして記述して、実行可能にする。

自動化されたユニットテストを実行することで、いつでもユニットテストの結果を把握することができる。

さらには自動化されたユニットテストセット（テストスィートとも呼ばれる)は、回帰テストとして常に実行することができる。

後で変更をする際にも、テストを実行しながら修正をしていくことで、不具合の混入をいち早く検知して対応することができる。

ユニットテストの自動化の手順は、大きく分けて、最初にユニットテストを書いてから、プロダクトコードの実装に入る**テストファースト**と、既に動作しているプロダクトコードに対してユニットテストを自動化する**テストラスト**の二種類があり、状況に応じて使い分ける必要がある。

ユニットテストを開発者自身が書くことについて、テストの品質について疑問を持つ場合は、品質保証部門の担当とテスト設計をすることでユニットテストの品質を高めることができる。

開発中からユニットテストを自動化する際には、開発プロセスとしてテスト駆動開発を検討するのが望ましい。

作成したユニットテストセットを継続的に実施するために、継続的インテグレーションを検討するのが望ましい。

留意点

テスティングフレームワークを使いこなすスキルも重要であるため、実際に利用する前に学習が不可欠である。

テストラストでのみユニットテストを自動化しようとすると、ユニットテストが書きにくい設計になっていることが多い。テストを書きやすい設計にするためには、テストファーストを心掛けたほうがよい。

ユニットテストが整備されていない既存のソフトウェア（レガシーコードと呼ぶ）を扱う場合は、ユニットテストの自動化のコストが飛躍的に高くなる。この場合は、段階的に設計を見直し、少しずつユニットテストの自動化を整備していく必要がある。

単純に自動テストを追加していくだけでは、テストコード自体の保守工数が増加してしまうリスクがある。テストコード自体も開発者が読みやすく保守しやすいシンプルな設計に改善していかないといけない。

ユニットテストを自動化しても実施に実行コストや不確実性があるケース（データベースへの接続、ネットワーク通信）の場合は、テストの実行に時間がかかり過ぎる、テスト結果が状況次第で変わってしまうことがある。ユニットテストのレベルでは、テストダブル[6]（テスト対象が依存しているコンポーネントを置き換える代用品)を用いてテストを実施するのがよい。

1回限りしか実施しないテストは自動化する必要はない。

テスト結果に失敗が含まれていた場合、その状態を改修せずに放置しておくと、テストが失敗する状態が常態化し、テストがメンテナンスされなくなる。テスト結果が失敗になる場合は直ちに改修し、常に100%の成功を保つようにしなければならない。

ユニットテストでプロダクトコードが仕様に合っているかどうかは検証できるが、仕様そのものの妥当性は検証できない。

効果

◆ユニットテストの実施時間が大幅に短縮され、頻繁に実施可能となる。
◆テスト実施のミスを削減できる
◆メンテナンス期間が長いシステムであればあるほど投資効果が高い

利用例

ユニットテストの自動化は、全体で80%以上の適用率であった。切り口別でも差異はなく、まんべんなく適用されていた。

B社事例(2)では、初心者ばかりのチームであるにもかかわらず、高い品質のソフトウェアを作りあげることができた。一方、ユニットテストの自動化について手を抜きがちなので、しつこく言い続ける必要があった。

B社事例(3)では、当初から自動化を行っていた。そのためコードを変更した時の安心感を持って修正できている。

G社事例(10)では、元々のシステムが利用していたユニットテスティングフレームワーク（RSpec）[7]を検討していたが、そのシステムは同時に受入テスト自動化フレームワーク（Cucumber）[8]も利用していた。工数的に両方の自動化に対応できなかったため、受入テストの自動化フレームワークのみ採用し、ユニットテストの自動化は実施しなかった。
受入テストの自動化によって、ユニットテストフレームワークで記述していたテストがカバーする範囲をほぼ置き換えることができた。

---

[6] http://ja.wikipedia.org/wiki/テストダブル

[7] Rubyで広く用いられているユニットテストフレームワーク。BDD(振舞い駆動開発)フレームワークとも呼ばれる。http://rspec.info/

[8] Ruby上で動作する自然言語で書いた振る舞いをテストとして実行可能にするフレームワーク http://cukes.info/

K 社事例(22)の場合は、ユニットテストの自動化の価値は認めるものの、テストコードを書くコストは無視できないことがわかった。

M 社事例(25)の場合は、最初はユニットテストの自動化は行っておらず、途中から自動化に取り組みはじめた。そのため、リファクタリングを実施しながら、タイミングを見て自動化を進めている。

関連プラクティス

テスト駆動開発
継続的インテグレーション

3.2.5. **受入テスト** /Acceptance tests are run often and the score is published

"顧客の受入テストがストーリーのゴールとなる"

別名: 顧客テスト、機能テスト、ストーリーテスト

要約

ユーザーストーリーが完了したかを判定するために、プロダクトオーナーと合意した受入テストを作成する。その結果、ユーザーストーリーの完了条件が明確になり開発チームとプロダクトオーナー間の認識の齟齬がなくなる。

状況

開発者は、プロダクトオーナーとの間でイテレーションの計画を作る中で、開発するユーザーストーリーについて様々な会話をする。どのような仕様か、機能を通じて何を実現したいのか、機能の実装に必要な様々なビジネスルールなどを聞き出す必要がある。

開発者はユーザーストーリーの内容を知り、見積り、計画にコミットしなければならない。

問題

ユーザーストーリーには顧客が受け入れることができる条件が不可欠である。

ユーザーストーリー毎に受け入れ条件を明らかにしなければ、仕様の明確化や見積りをすることはできない。また、何を作ればいいかの合意形成もできない。

ユーザーストーリーの具体的な受け入れ条件を明確にしないまま開発を進めると、最終的にプロダクトオーナーが欲しかったものと、開発者が作ったもののギャップが大きくなる。

フォース

◆プロダクトオーナーは必要な機能に対するイメージはあるが、具体的には把握していない場合もある。
◆ユーザーストーリーが正しく実現されているかどうかのテストを、ソースコードを変更する度に実施しなければならない。

解決策

プロダクトオーナーと機能について対話をして受け入れ条件やビジネスルールを明確にし、それを元に受入テストを作成して実施すること。

受入テストは、イテレーション毎に実現されるユーザーストーリーに対するテストであり、ユーザーストーリーのゴールでもある。

プロダクトオーナーが機能に対して具体的なイメージを持っていない場合は、対話やペーパープロトタイピングなどで、具体的なイメージを引き出しながら洗い出す。

プロダクトオーナーが受入テストを書く場合は、開発者が支援をしながら作成する。反対に、開発者が受入テストを書く場合は、プロダクトオーナーが理解しやすい形式で作成する。

開発者とプロダクトオーナーや顧客が一緒に仕様や受け入れ項目を洗い出すワークショップを開催するとよい。そのタイミングはプロダクトバックログの見直しのタイミングがよい。

作成された受入テストが成功しないうちは、ユーザーストーリーは完了にならないため開発者はまず受入テストを作成することを最初の作業としなければならない。

受入テストは可能な限り自動化するのがよい。自動化にあたっては、GUI レベルから自動化するのか、GUI のバックグラウンドにある API レベルから自動化するのかの戦略を決定する必要がある。

受入テストが自動化された後は、自動化された回帰テストとして、継続的インテグレーションを行うのが望ましい。

留意点

◆受入テストを手動のままにしておくと、イテレーションが進むにつれてテスト工数が増えるため実施に時間がかかり、繰り返し実行することが困難になる。
◆受入テストは特定のユーザーストーリーに対してのテストであるため、複数のユーザーストーリーにまたがるテストは別途考慮が必要である。
◆受入テストは成功している状態を維持しな

ければならないため、失敗を見つけたら即刻修正する必要がある。

効果

◆ユーザーストーリー毎の完了を明確にできる。
◆自動化することでユーザーストーリーの振る舞いが正常かどうかをいつでも確認できる。

利用例

受入テストは、全体で 44%の事例で適用されていた。そのうち自動化を実現している事例はさらに少数であった。

A 社事例(1)では、受入テストの自動化は実施しておらず、プロダクトオーナーと正常系のエンド・ツー・エンドテストを手動で行っていた。

C 社事例(4)では、プロダクトオーナーに受入テストを依頼していたが、受入テストの内容について詰めきれていなかった。そのため、プロダクトオーナーからテスト観点が不明であるとの指摘があった。
テストシナリオを起こしてのテストは実施できておらず、アドホックに行っていた。受入テストが 100%実施できていたかというと、そうではなかった。
開発チームで受入テストを実施した時には、そのテスト結果を公開したが、プロダクトオーナーが受入テストを実施した場合は、受入テストが成功したか失敗したかの結果ではなく、プロダクトオーナーが気になった点のみが報告されていた。

F 社事例(8)では、元々が自動化されたテストが全くない環境で、最初はユニットテストの自動化を試みたがうまくいかなかった。そこで受入テストの自動化に注力した。よく使われる機能から投資対効果を評価して受入テストの自動化を実施した。その結果、受入テスト自動実行することによりバグ修正後に既存の動作を壊していないことがすぐ確認できるため、1 日以内にリリースできるようになった。

G 社事例(10)では、受入テストツールの Cucumber を用いた。日本語を用いてテストシナリオを書き、それを元に受入テストの自動化を行った。
テストシナリオを開発チームとプロダクトオーナーの間で合意しているため受入テストの成功がユーザーストーリー完了の条件になっていた。
受入テストは何度も実施していたが、プロダクトオーナーにテスト結果を公開することはしていなかった。プロダクトオーナーが信頼して開発チームに任せていたためである。

K 社事例(20)では、プロダクトオーナーからテスト作業はできないとの申し入れがあった。テストシナリオを確認することは問題ないが、膨大な量のテストは実施してはもらえないため、開発チームが受入テストを実施した。
Redmine に仕様、受け入れ条件や制約を記述している。遠隔地にいるプロダクトオーナーがその内容を確認したり、印刷したりすることができるようにした。

関連プラクティス

ユーザーストーリー
自動化された回帰テスト
継続的インテグレーション

参考文献

Adzic, G. (2009). Bridging the Communication Gap: Specification by example and agile acceptance testing, neuri

Adzic, G. (2011). Specification by Example: How Successful Teams Deliver the Right Software, Manning Publications

3.2.6. **システムメタファ** / System Metaphor

“喩えが共通の語彙と、新たな発想を生みだす”

別名:なし

要約

チームがより直感的にわかる共通の語彙が必要ならば、だれもがわかるメタファを使ってシステムを表現する。その結果、システムに対しての共通認識を簡単に構築でき、内部設計からユーザーエクスペリエンスまでの広い範囲で、新たな解決策や、見過していた問題を発見することができる。

状況

システムについて開発者やステークホルダー間で共通認識を持ちたいと考えている。

問題

システムの設計は、抽象的な思考だけで行われることが多い。システムを理解するために、開発者とユーザーの双方に通用する共通の用語が存在しない。

開発者は、レイヤ、フィルタ、コレクション、といった抽象的な概念を用いてシステムを設計する。しかしユーザーにとっては、上記概念は全くの抽象的概念であり、システムの理解の役には立たない。

より直感的に理解ができる見方が欲しい。

フォース

◆既存の問題領域の概念（ドメインモデル）を使って設計しているが、思考がドメインモデルに縛られすぎていて飛躍しない。

解決策

システムの問題領域を、だれでもわかるメタファ（隠喩、例え）を使って表現して、開発者や関係者の間で利用すること。システムの設計をメタファを使って行ない、メタファから導かれる新たな概念を積極的に利用すること。

システムメタファは、ソフトウェア設計において、昔から利用されているが、あまりに浸透し過ぎてしまっていて気づかなくなっているこ
とが多い（ファイアウォール、レイヤ、メール、フォルダ、ノートブックなど）。

[ベック 2000] や [エヴァンス 2011]でシステムメタファは明示的にプラクティスとして紹介されて再度人々に認知されている。

[ウェイク 2002]によれば、システムメタファは以下の目的のために利用する。

◆共通のビジョン
- ♦ 関係者間でシステムの動作について共通の認識を持つ

◆共通の語い
- ♦ オブジェクトなどに名前を提供して専門用語を生み出す。

◆新たな発想の生成
- ♦ メタファを用いることで新たな概念を生成することができる。

◆アーキテクチャ
- ♦ メタファによってオブジェクト間のインターフェイスが決まる

また、開発者が利用する内部の設計だけでなく、ユーザーの目にとまるプロダクト全体に、ユーザーのよく知る概念をメタファとして用いることで、ユーザーエクスペリエンスを向上させることができる。既に馴染みのある「書類、フォルダ、ゴミ箱」（オフィスの机のメタファ）や、ショッピングカートなどはそのよい例である。

[ベック 2000]で紹介された例は、ユーザーインターフェイスではなく、システムの概念モデルとしてメタファを利用していた。

システムメタファを適用する上でもシンプルデザインを心掛ける必要がある。

システムメタファを新たに適用しなければならない、あるいは不要になった時はリファクタリングの必要がある。

システムメタファを用いて、システムについての認識を合わせることでチーム全体が一つに近づくことができる。

留意点

メタファはシステムを完全に表現できないため、すべてを一つのシステムメタファで統一して表現しようとしてもうまくいかない。

しっくりこないメタファは逆に人々を混乱させるため、適切なメタファに変更する必要があ

る。
メタファを使えばシステムをユーザーが使いやすくなるとは限らない。適切なメタファであるかどうかは検証が必要である。

共通理解が充分あり、特に新しい発想などが必要ない場合は、メタファは不要かもしれない。

[エヴァンス 2011]においては、システムメタファの利用を推奨していながらも、その利用には注意する必要があると述べている。メタファで生まれた新しい発想をユビキタス言語（あるドメインにおける共通語彙）に加えていくことを主張している。

深くシステムメタファがシステムに組み込まれている場合は、メタファなしにシステムについての話はできなくなる。

効果

◆ステークホルダーの間でシステムについての共通認識が向上する。
◆メタファを通じてシステムへの新しい発想を得ることができる 。
◆設計に一貫性が生じる。

利用例

本調査の国内事例 26 事例の中にはメタファを利用している事例が存在しなかったため、海外の事例に目を向けることとした。

米 I 社はアジャイル型開発に関する eLearning のコンテンツを Web サイト上で提供していた。それまでのコンテンツの集大成を「グレイテストヒッツ」と呼び Web サイト上で提供していたが、当初はシンプルな Web サイトのデザインであった。

Web サイトのデザインをユーザビリティの専門家に評価してもらったところ、10 点満点中 2 点であった。そのため、ユーザビリティの改善を実施して、サイト全体を本のメタファを使って構築し直した。サイトのデザインやユーザーインターフェイスだけでなく、システムのドメインオブジェクトも Book, Chapter, Page というように変更された。よいシステムメタファがもたらす「メタファを元に、何が必要かを発見できる」状況であった。

その後、e-Learning のユーザーから「プレイリストが欲しい」というフィードバックをもらった。プレイリストには、「本」ではなく「音楽」のメタファが適切であると考えた。データベースも、ドメインオブジェクトも、ユーザーインターフェイスも「本」のメタファに縛られていたが、実現したいことをより表現できると考え「音楽」メタファを変更することに決めた。

アルバム、ボックスセット、プレイリストという音楽の概念を使ってシステムを再構築した。

「音楽」のメタファは、概念を表現するだけでなく、さらに深くプロダクトに入りこんで新しいサービスも生み出した。

音楽の世界で、生徒の演奏を批評するクリニックの概念を用いて、プログラミングのパフォーマンスを批評するサービスを生み出した。プログラマのパフォーマンスは、節（Passage）、音符（Note）、和音（Chord）という音楽の概念で表現されている。

他方、プロダクトが徹頭徹尾音楽メタファで構成されているため、逆に音楽メタファを使わないで表現することは困難になっている。

システムメタファというだけでなく、プロダクトメタファとして隅々で利用されている事例であった。

関連プラクティス

シンプルデザイン
リファクタリング
チーム全体が一つに

参考文献

[ウェイク 2002] ウェイク, W.W.W.　長瀬嘉秀・紺野睦監訳 (2002)『XP エクストリーム・プログラミングアドベンチャー』 ピアソン・エデュケーション

[エヴァンス 2011] エヴァンス, E.エヴァンス, E. 今関剛監訳 (2011)『エリック・エヴァンスのドメイン駆動設計』 翔泳社

3.2.7. スパイク・ソリューション / Spike Solution

“リスクに対して くさびを打ち岸壁を登る”

別名：実験、曳光弾

要約

技術的に不明な点がある場合は、動くコードを書いて実験をして技術的な学習をする。その結果、技術的不明点について実践的な学びを得ることができる。

状況

開発を進める上で、技術的な情報が不足している場合がある。
例えば、あるメソッドが特定の条件でどのような例外を発生させるかわからない、あるいは技術的な実現方法が不明確な部分がある。

開発メンバーのだれもが確実な答えは持っていない。

問題

情報が足りない中で推測を元に開発を進めることはリスクである。

開発を進めていく上で、様々な場面で情報が不足していることがある。例えば、使用しているライブラリのメソッドが特定のケースでどのような例外を発生させるかわかっていない場合を考えてみる。この場合、開発者が自分の推測で「多分、この例外が発生するはずだから、XXX の例外処理を実施しよう」と開発を進めてしまうと、実際に発生する例外が開発者の想定と異なった場合、期待する結果にならず、不具合の原因になる。

また、そもそも技術的に実現方法が見えない場合もある。そのような状態で計画を立てても、到底思うようにいかない。

フォース

- ◆事前に技術的な疑問点をすべて解消しておきたいが、どれくらいの期間が必要かを予測するのが難しい
- ◆技術的な疑問点は実際に実装する直前になって発覚することが多い

解決策

技術的に疑問点がある時は、実際に動くコードで実験して学習しよう。

技術的な疑問には、実装上の疑問と設計上の疑問の 2 種類がある。いずれも動作するコードで実験した後に、得た情報を踏まえて以降の作業を決定する。

できる限り「独立したプログラム」を使って実験を行い、結果を検証した後はコードを捨ててしまってよい。実験コードは使い捨てを前提とするが、後でプロダクトコードの実装の参考にするために、専用のディレクトリを作成してドキュメント扱いで格納しておくのがよい。
（spikes/など）

実験をテストコードの中で行うのも有効だ。その場合もテストケースの中に、実験のコードを直接記述する。

実験をプロダクトコード上で実施しなければならない場合は、あらかじめ実験前のコードをチェックインしておき、いつでも実験を廃棄できるようにしておくこと。

実験は最小限のコードに抑える。着目すべきは実験の結果、どのように動作するかである。値もハードコードし、コードの読みやすさもそれほど考える必要はない。

イテレーションの最中で疑問が生じた場合は、必要な時に実験を行う。イテレーション計画ミーティングの中で実験が必要だとわかった場合には、その分の作業も考慮して見積もる。

もし大掛かりな実験が必要であると判断した場合は、実験を行うユーザーストーリーを別に切り出して、作業を見積もるとよい。

実験した結果を元にシンプルデザインで、テスト駆動開発を用いてプロダクトコードを開発するのが望ましい。

留意点

実験で使ったコードを汎用的に再利用しようとすると、「動けばいい」だけのコードに引きずられてしまう。捨ててもよいコードとして実験を行い、プロダクトコード上で実装する際には、テスト駆動開発を使って書き直すこと。

実験は事前に全て実施しておきたいという欲求に駆られ、調査フェーズなどを設けがちである。しかし、変化が激しい場合は、シンプルデザインと同様に、事前に調査する範囲を決定しようとせず、調査する必要が生じた時点で実験を行うことでムダな実験を防ぐことができる。

効果

◆動作するソフトウェアで実験することで、推測に基づく開発を抑制でき、効率が高まる。
◆プロダクトコード上で試行錯誤せずに、お試しの環境で実験を終えることができる。

利用例

スパイク・ソリューションは、全体の 48%で適用されていた。切り口別に見ると、自社開発では 60%の適用率であるのに対し、受託開発では 27%であった。スパイク・ソリューションのように創発的に発見される作業を組み込みにくいためと推測される。また、使い捨てにしない例も見受けられた。

B 社事例(2)では、必要に応じてスパイク・ソリューションを実施していた。特に開発序盤が多かった。

C 社事例(4)では、技術的に難易度が高い領域があったため、プロトタイピングで技術的検証を行った。

H 社事例(14)では、初期段階でプロトタイプを作成し、それを改修してプロダクトを製品化した。厳密に言うとスパイク・ソリューションとは異なる。

L 社事例(23)、(24)では、実現や導入の可能性について、事前調査のフェーズを設けて対応した。厳密に言うとスパイク・ソリューションとは異なる。

関連プラクティス

イテレーション計画ミーティング
ユーザーストーリー
シンプルデザイン
テスト駆動開発

3.2.8. **リファクタリング**/ Constant refactoring

“リファクタリングで、既存コードの設計を改善しよう”

別名なし

要約

コードがわかりにくい、複雑である、汚いと感じたら、コードの振る舞いは変えずに内部の設計を改善する。その結果、コードの見通しがよくなり、将来的な設計へのリスクを軽減することができる。

状況

ウォーターフォール型開発においては、あらかじめ全体像を設計し、その設計に基づいてコードを書くことが多い。設計や要求の変更がない時は、正常に動作しているコードを修正することは比較的少なかった。

一方、アジャイル型開発においては、イテレーション毎にフィードバックを得るため、変更の頻度が高い。

オブジェクト指向プログラミングが台頭してきたこともあり、情報隠蔽やインターフェイスを用いて、機能を変更せずにコードを変更することが容易になっている。

問題

変更や修正を繰り返している中でコードがカオス状態になり、バグが入り込む余地ができてしまう。

また、動作はするが、よくない設計、ビジネス変化により、もはや不要ではあるが存在する設計判断、不要なコメント、巨大なメソッド、のようなものを負債に喩えて**技術的負債**と呼ぶ。

技術的負債を返済しないまま開発を進めていくと、負債の利子は増え続け、最終的には高いメンテナンスコストになってしまう。最悪の場合はシステムをメンテ不能に陥れる。

フォース

ソースコードを修正するにはリスクが伴い、既に正常に動いているソースコードを変更する場合には新たな問題を引き起こす可能性がある。

リファクタリングを実施しなくても、とりあえずプロダクトコードは動作する。

解決策

外部から見た時の振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変化させる。

リファクタリングのタイミングは、イテレーションの後半や、毎日の決まった時間を取って定期的に行う。また、コードレビューや、ふりかえりなどをトリガーに自然発生的に行うこともある。ふりかえりで気づいたことはタスクとして追加する。

リファクタリングは、堅実なテストの存在が欠かせない条件である。内部構造を変更したとしても、振る舞いに影響が出ないことをテストコードによって確認できるからである。

リファクタリングでは、重複したコード、長過ぎるメソッド、巨大なクラス、多すぎる引数などから、わかりにくい/不要なコメント記述に至るまで、問題と感じる箇所に対して行う。

リファクタリングを行う際には、あらかじめ作業の中で工数を見積もっておくことが重要である。大きなリファクタリングが必要になる場合は、新しい機能の追加量、リファクタリングを実施しない場合の潜在的リスク、対応工数、などを含めてプロダクトオーナーと共に検討しなければならない。

ふりかえりで、リファクタリングについて話そう。ユニットテストの自動化、シンプルデザインも、リファクタリング実施の前提になる。

リファクタリングを行う前と後で自動化された回帰テストや、受入テストのテスト結果が変わらないことを確認しなければならない。

効果

◆アーキテクチャも含め、より設計改善を行うことができる。
◆メンテナンスしやすいコードを維持することができる。
◆リファクタリングは実施しているが、内部構造の改善につながっているかどうかは定かではない場合もある。

留意点

リファクタリングをしなくてもプロダクトコードは動作する。そのため、ついうっかり対応し忘れてしまうことが多い。タスクボード上で明示しておく、「完了」の定義に含めておく、対応する時間を決めておく、ペアプログラミング等で忘れないようにする、といった工夫をするのが望ましい。

バグの発生は、ある特定の場所に集中することが多い。そのため、ポイントを絞り、そのポイントを徹底的にリファクタリングすると効果が高い。同様に、全コードに対してテストコードが書かれていない場合、その絞られたポイントに対してのみテストを書くだけでも効果がある。

リファクタリングは開発者が必要性を理解しやすいが、プロダクトオーナーや顧客にその価値をうまく伝えなければならない。顧客からすると「新しい機能を追加しないで、既存の機能をいじっているだけ」と捉えられてしまう。開発側からビジネス的な影響も含めての説明が必要だ。

リファクタリングは工数の確保が大切である。あらかじめリファクタリングに必要な工数を見積もっておこう。

利用例

リファクタリングは、全体で 69%の事例で適用されていた。契約別で見ると、自社開発では 78%の適用率であったのに対して、受託開発では 54%の適用率であった。これはリファクタリングの工数確保が難しいことが原因と推測される。

A 社事例(1)では、リファクタリングは特に意識することなく日常的に行っていた。「サボらずテストを書く」ことと「リファクタリング」を大事にした。

B 社事例(2)では、アーキテクチャも含め、ソースコードをどんどん変更していた。リファクタリングを非常に重要視しており、チーム全体で推奨していた。

B 社事例(3)では、リファクタリングをルール化していないものの、問題になりそうな箇所があれば、ふりかえりの中でチームからリファクタリングの指摘が出ていた。

C 社事例(4)では、イテレーションの間、気づいた時にリファクタリングを行っていたが、100%きれいなコードにはできていなかった。プロジェクトの規模が大きいため、どんどん生じるコードの制御が効かず、リファクタリングしたほうがよいところについてすべて対応できていなかった。

E 社事例(6)では、小さな単位のリファクタリングはイテレーションの開発の中で工数を確保して実施できていたが、特に大きなリファクタリングについては、事前の計画に含まれておらず実施にあたり調整が生じた。

I 社事例(15)では、リファクタリングの実施時期が不明だったので、静的解析ツールを使用し、ルールを定義してスコアが低ければリファクタリングするようにした結果、タイミングがつかめた。

K 社事例(20)では、以前、顧客がソースコードのメンテナンス性に課題を感じていたため、顧客のほうから、工数を 15%上乗せしてリファクタリングに充てるように助言された。その結果、リファクタリングを前提にした開発が実現できた。
リファクタリングは忘れがちなので、タスクボードのレーンとして表現し、忘れずに実施することができた。

L 社事例(21)では、イテレーションの後半に集中的にリファクタリングを実施して、改善要望を取り込みやすいコードを維持することに効果があった。

L 社事例(22)では、リファクタリングが後回しになることを防ぐために、毎日決まった時間に強制的にリファクタリングの時間を取ることで対応した。

他方、H 社事例(14)では、リファクタリングを実施しているが、内部構造の改善につながっているかどうかは定かでないという疑問を持っている意見もあった。

関連プラクティス

ふりかえり
自動化された回帰テスト
ユニットテストの自動化
シンプルデザイン
受入テスト

参考文献

ファウラー,M. 児玉公信・友野晶夫・平澤章・梅澤真史訳（2000）『リファクタリング プログラミングの体質改善テクニック』

3.2.9. **シンプルデザイン** / Simple Design

“ムダのないデザインが価値を生む”

別名: YAGNI

要約

将来の変更を予測して設計を複雑化するのではなく、今必要な要件を実現するための最小限の設計にすること。その結果、修正や変更に柔軟に対応できる設計になる。

状況

要求の変更に耐えられるように、柔軟性のある設計にしたい。

問題

変更を予測して事前に設計を行うことは難しい。

将来の変更を予測して設計すると、不要なものまで含めてしまい、結果的に複雑化してしまう。設計が複雑なソフトウェアは変更に対して柔軟ではない。

デザインパターンのような再利用を促進する設計カタログもあるが、前提として「新しい要求や、既存の要求に対する変更を前もって知る」ことが前提となっている。しかし、そもそも変更を予測することは困難である。

フォース

- ◆将来の変更に対応するために、設計に柔軟性を盛り込むことはできるが、高いコストがかかる。
- ◆設計に柔軟性を盛り込むコストを払うためには、要件として正当化される必要がある。
- ◆既存のソフトウェアの設計を途中で柔軟に変更したい時には、全体にかかわってくる要件ほど変更コストは高くつく。
- ◆変更や非機能要件などをフレームワーク に任せることによって担保したいが、そもそもフレームワークの想定内に収まるかは未知数である。

解決策

設計はその時点の要求で必要とされる範囲に留めておき、できるだけシンプルに実現すること。

シンプルさとは、ムダな設計の複雑さを取り除くことである。シンプルな設計の原則として、以下のようなものがある。

YAGNI（You Aren't Gonna Need It）原則

推測に基づいて設計するのではなく、今まさに実現しようとしている機能、要件に対して必要なことのみ設計し実装する。同様に既に使われていないソースコードは取り除き、今必要なものだけにする。

今は必要ないが、将来必要になりそうだとわかっている場合も、その機能が実際に必要となる時まで手を入れないようにする。計画が変更になってしまうと、その設計はムダになるため、あくまでも、設計する対象はユーザーが確実に必要とするものだけにする。

DRY（Don't Repeat Yourself）原則

重複を避け、ただ一度だけ表現すること。Once and Only Once（一度、ただ一度だけ）でも言われる設計ガイドライン。単純にプロダクトコードの重複をなくすだけでなく、重要なコンセプトを明確にして一度で表現する。
例えば、通貨を decimal データ型で表現するのではなく、Currency クラスを作って表現することで、システム内に散らばる処理を一箇所に格納することができる。

今は必要ないが、後で検討するには複雑でコストが高くなりそうな問題（分散処理、永続化、国際化、セキュリティ、など）についても、事前に設計して実装する欲求を抑えてシンプルに実現するのが望ましい。

プロダクトオーナーが、リスクを踏まえて特定の機能を実現する要求の優先順位を上げることができるのであれば、顧客価値を実現しつつリスクも解決できる。ただし優先順位を付ける決定権はプロダクトオーナー側にある。

例えば、現在は日本語しか対応していないプロダクトを、将来的に多言語化に対応させたいが、今は中国語対応したいと考える場合、「多言語化に対応する汎用的なメカニズムを作る」という形で多言語化の仕組みだけ組み込むのではなく、「中国語に対応する」という要求を実現する中で、多言語化に対応した仕組みを実現しするのが望ましい。

また、既存の設計をリファクタリングする時に、顧客が求めていない機能まで実装するのは不適切であるが、将来のリスクを軽減する方向に向けることは可能である。例えば、リスクを軽減するためにDRY原則を適用して、将来、手を加える可能性が高い処理を特定のクラスに集めることはできる。
例えば、国際化に必要であると思われる通貨を扱う変換処理が複数のクラスに散らばっているのであれば、それをCurrencyクラスの一つのメソッドに集約することで、機能は追加せずに設計をDRYにして改善し、将来のリスクを軽減することができる。

シンプルデザインを保っていくためには、ユニットテストの自動化やリファクタリングが不可欠となる。

留意点

◆シンプルデザインを単純に「クラスの個数を少なくする」、「メソッドの数を減らす」というルールに置き換えるとうまくいかない。
◆シンプルデザインは言い方を変えると、必要な時に必要なだけ設計を変更し改善していくことを意味する。ゆえに、常に意識していないとすぐに複雑化してしまう。
◆外部システムを利用する場合には、中間層を介して利用するようにし、外部システムが変更された場合の影響を局所化する。

効果

◆シンプルな設計を保っていることで、予期せぬ変更に対応しやすくなる。

利用例

シンプルデザインは、全体の60%で適用されていた。契約別の切り口で見ると、自社開発では70%以上の適用率であるのに対して、受託開発では40%であった。リファクタリングが適用しづらい影響が出ているのが理由と推測される。

A社事例(0)では、事前に利用ユーザー数を予測していたが、実際には予想をはるかに上回っていた。したがって、ある程度はアーキテクチャに余裕を持たせていたが、変更せざるを得ない状態になった。

D社事例(5)では、「汎用的に」「スケールする」「時間があれば」という言葉を思考停止ワードとして、設計に関する議論の際に注意していた。

J社事例(18)では、シンプルデザインを充分に行うことができていなかった。その結果、ソースコードが複雑になりテストコードが書きづらくなってしまった。

L社事例(21)では、アーキテクチャチームを構成し、事前に非機能要求を実装していたが、シンプルな設計を意識してムダな作り込みをしないようにしていた。

関連プラクティス

ユニットテストの自動化
リファクタリング

3.2.10. **逐次の統合** / Only one pair integrates code at a time

“一歩ずつ一歩ずつ”

別名: なし

要約

複数の修正箇所があるならば、一つずつインクリメンタルに修正してその都度統合する。その結果、問題の特定がしやすくなる。

状況

開発者は構成管理リポジトリを用いて、既存のコードに修正を加え、複数人で開発を進めている。

問題

統合する際に、複数の修正が含まれていると、どこが問題であるかわからなくなる。

ビックバン統合と言われるように多くの結合を行うと複雑さが高まり、結合の困難さが増す。場合によっては、統合が失敗することがある。

その際、統合時に複数の変更セットが含まれていると、失敗の原因を特定することが難しくなる。

問題がある変更セットがリポジトリに格納されていると、そのコードを使用して更なる問題を引き起こす可能性が高い。

フォース

◆開発者は並行して開発を進めたい。
◆リポジトリには問題のあるコードは含めたくない。
◆問題のある状態を最小限に防ぎたい。

解決策

インクリメンタルに少しずつ修正しよう。統合に含めるのは一度に一つの変更セットだけにしよう。

一度に一つの変更セットを統合することを心掛けていれば、もし何か問題が起こった場合に、原因は統合に含めた変更セットであることが容易にわかる。

開発者は、一つの意味ある単位の変更セット（テストコードが含まれている)を構成管理ツールにコミットする。その度に、統合を実施して、コミットした変更セットが正しく動作しているか、既存のコードを壊していないかを確認する。

問題がなければそのまま先に進み、もし問題が発生した場合は、該当する変更セットを取り消す、あるいはすぐに問題に対応する必要がある。

継続的インテグレーションツールを使用している場合は、統合タイミングをコミット毎にしておくことで、変更セット単位の統合が可能になる。

ユニットテストの自動化や、継続的インテグレーションは必須となるだろう。またテスト駆動開発を実施することで、逐次の統合が行いやすくなる。

開発者が共通の部屋で作業していれば、互いの変更についての情報共有がしやすく、問題切り分けを行いやすい。

インテグレーション専用マシンがあれば、開発者が手動で統合を行ってもよい。そうすると確実に逐次の統合を実現できる。

留意点

開発者が並行して変更セットをコミットしている場合は、逐次の統合ではコミット頻度に追いつかない可能性もある。

テストコードが存在しないと、逐次の統合で既存のコードを壊していないかの確認が取れない。

効果

◆統合時の問題対応を素早く行うことができる
◆構成管理リポジトリに問題のあるコードを混入させる確率が激減する

利用例

逐次の統合は、全体の 40%の事例が適用していた。規模別の切り口では、小規模では 48%適用

していたのに対して、中大規模では17%に低下している。開発人数が多くなるほど、逐次の統合が困難になることが推測される。

特に特徴的な利用方法はなかった。

関連プラクティス

継続的インテグレーション
ユニットテストの自動化
テスト駆動開発
共通の部屋
インテグレーション専用マシン

参考文献

Wells, D. (2009) Sequential Integration, http://www.extremeprogramming.org/rules/sequential.html

3.2.11. **継続的インテグレーション** / Continuous Integration/Integrate often

“小さく続けても、大きな成果を”

別名：常時結合、CI

要約

インテグレーション（システムのビルド、テストの実行）を自動化し、継続的に行うことによって、コードだけでなく動作環境を含めた確認を行うことができる。

状況

開発者は、各自の開発マシン上で開発を進めている。統合テスト環境や、本番環境は別に存在する。

開発完了したソフトウェアを本番環境で動作させるためには、ビルドだけでなく設定ファイルの編集やファイルの移動など細かな作業が必要である。

問題

開発環境では動作していても、環境が変わったり、他人の変更と一緒にすると動作しない場合がある。

たとえ開発者の環境で自動化されたテストがすべて成功していても、統合環境や本番環境で動作するとは限らない。また他人の変更によって動作しなくなることも多い。

プロダクトを常に利用可能な状況にしておくことで、適切で迅速なフィードバックを得たい。

ソースコードがインテグレーションされておらず動かない状態では、計画が適切であるかどうかを知ることができない。例えば、現状を把握せずに次のイテレーションに向けての計画を行うことは、現状からの乖離が起こりやすく危険である。

チームにおける「完了」の状態と、プロダクトを出荷できる状態に違いがある。

トラブルやインシデントは、ソースコードの間違いだけでなく、実行環境の設定や構成などの間違いなどによって発生している。出荷できる状態でのプロダクトの確認が大切である。

フォース

- ◆設定や構築、テストを適切に行いたいが、その機会は限られている。
- ◆出荷可能な最新の動作環境での動作確認をしたいが、その状態に持っていくことは時間やコストがかかる。

解決策

インテグレーションマシンを用意し、自動でインテグレーションを継続的に行おう。

コードを数時間もしくはコミットされる毎にインテグレーションする。
ビルドとテスト、およびリリースするための環境を最新にしておく。

インテグレーションは、リポジトリから最新のファイルを取り出し、ビルドし、ファイルの設定や構成を行い、テストをすることなどを含む。それらを自動的に行うJenkins[9]などのツールを使うと容易に実現できる。

インテグレーションの結果、回帰テストが実行され、その結果が失敗していたら直ちに修正する。継続的インテグレーションを実施することで、バグを早期に発見して対処することができる。

ビルド結果は、他の環境でも動作するようにすること。

複数のチームに関係する時は、専門のインテグレーションチームをアサインすることもある。

ユニットテストの自動化や、自動化された回帰テストが不可欠となる。迅速なフィードバックを実現する。

集団によるオーナーシップ、インテグレーション専用マシンは、継続的インテグレーションの前提となる。

インテグレーションの結果、バグを発見した後は、バグ時の再現テストを心掛けよう。

[9] http://jenkins-ci.org/

留意点

◆チーム内だけでなく、運用チームやインフラチームに影響が出ないか検討しておくこと。
◆継続的インテグレーションを初めて導入する時は、ツールなどの調査をすること。本調査時は、Jenkins を使う事例が多かった。
◆1 時間おき、コミット毎、1 日に数回、イテレーション毎のようにインテグレーションのタイミングはプロジェクトで決める。
◆ビルドやテストの実行に時間がかかる場合は、頻繁にインテグレーションできない。遅いテストを切り分けて、別のサイクルで実行できるように検討する。また遅い箇所も継続的に改善してできるだけ早くなるようにする。
◆ビルドやテストに時間がかかるようであれば、サブシステムに分割して、自動テストを実行することも検討する。
◆継続インテグレーションので失敗したテストが存在する場合は、すぐに対処しておかなければテストが失敗することが定常化してしまい、コードの問題があることを見過ごしてしまう。常にすべてのテストが成功する状態を保つこと。

効果

◆継続的インテグレーションを行うことで、継続的に「出荷できる状態」にできる。
◆新しく追加されたコードが問題既存のバグを引き起こしていても、早期に検知できる。
◆問題に早期に対処することで継続的に出荷可能な状態を保つことができる。
◆動作環境も含め、継続的インテグレーションできるため、自動テストやアジャイル型開発の基盤の一つとなる。

利用例

継続的インテグレーションは、全体の 78%の事例で適用されていた。システム種別の切り口では、B2C サービスでは 88%の適用率であったが、社内システムでは 69%であった。継続的に利用されるシステムであればあるほど、継続的インテグレーション環境を整備しておくことが望ましい。

B 社事例(2)は、モバイル端末用アプリケーションの開発で、継続的インテグレーションはしていなかったものの、統合開発環境 Eclipse で自動実行していた。継続的インテグレーションツールである Jenkins で複雑度のメトリクスを計測している。

C 社事例(4)では、継続的インテグレーションツールをプロジェクト開始段階で導入し、1 時間毎にビルドを行うようにした。ビルドの状況を全員にメールした。ビルドが失敗（Fail）した時は、ビルドが成功（Success）するよう即時に対応している。ビルド開始のトリガーは時間であり、コミット単位ではない。

G 社事例(9)では、継続的インテグレーションツールは使っておらず、ステージング環境への自動配置（デプロイ）を実施した。継続的インテグレーションではなく、スプリントのレビュー前に自動配置した。

G 社事例(10)では、継続的インテグレーションツールの Jenkins でテストは実行していたが、デプロイまでは実施できていない。

J 社事例(16)では、継続的インテグレーションツールの Jenkins を、世間的な流行に従って使っている。

J 社事例(18)では、コミット毎と 1 日に 2 回、継続的インテグレーションツールでテストを流している。

M 社事例(25)では、継続的インテグレーションツールである Jenkins を導入しようとしているところである。

関連プラクティス

ユニットテストの自動化
自動化された回帰テスト
集団によるオーナーシップ
インテグレーション専用マシン
バグ時の再現テスト

3.2.12. 集団によるオーナーシップ / Use collective ownership

“コードはチーム全員のもの”

別名: 共同所有

要約

特定の人しか知らないことがないように、ソースコードや業務に関する知識を共同所有する。その結果、変化や問題に強いチームになる。

状況

チームで開発を行っているが、ソースコードや業務に関する知識などは特定の人のみが把握している状況である。

問題

ソースコードや業務に関する知識が属人化されると、他の人がそのタスクを実施することができずに、作業を平準化することができず特定の人に作業が集中してしまう。

トラブル発生時も、担当者がいないと対応ができない。属人化の問題は、なにより「学びの視点」や「学びの機会」が減り、成長が偏ってしまうことである。

その結果、その担当者の退職・病欠などの場合には対応できない状況になってしまう。

トラックナンバー（プロジェクトメンバーのうち何人がトラックにはねられたらプロジェクトが立ち行かなくなるかを示す人数）は、特定の人に知識が集中しているかどうかの指標になる。

フォース

◆属人性を排除したいが、開発スピードも落としたくない。
◆自分の開発したソースコードやドキュメントに誇りがあり、公開したくない。

解決策

リポジトリの内容は、担当者だけが変更するのではなく、チームのだれもが把握し変更できるようにしておくこと。

バージョン管理をして、ソースコードやドキュメントを共通リポジトリに入れる。その時の公開範囲の変更があった際には関係者に告知する。

また、単に関係者から見えるようにしただけでは、属人性は排除できない。ペアプログラミングやレビュー、インスペクションなどは、集団によるオーナーシップを促進する。

だれが見ても理解が容易になるように、コーディング規約でチームのルールを明らかにして、だれでも理解容易にしておくのがよい。

人材のローテーションによって、積極的に広く様々なコードに触れる機会を増やす。

集団によるオーナーシップを進め、自己組織化チームに一歩近づける。

留意点

◆ソースコードだけでなく、ドキュメントを共同所有すると、より効果的である。
◆ペアプログラミングやソースコードレビュー、インスペクションをして、チームメンバー全員がすべてのソースコード・ドキュメントに触れられる状態にしていないとうまくいかない。
◆集団によるオーナーシップで情報を得た側は、情報を提供した人に対して感謝と尊重の意を伝えるようにすると、より共同所有が進む。
◆業務が多忙な時ではなく、比較的落ち着いている時に共有を進めるとよい。

効果

◆ペアプログラミングやソースコードレビューを通じて知識が共同所有されるようになり、さらにはチームの知恵が蓄積される。
◆属人性が排除され、トラブルや変化に強いチームになる。

利用例

集団によるオーナーシップは、全体で83%の事例に適用されていた。規模の切り口では、小規模が87%に対し、中大規模は67%とやや低かった。規模が大きくなると、集団によるオーナーシップが難しくなる傾向が推測される。

A社事例(0)では、一部のみ実施、もしくは実施できていない状態である。アーキテクチャは疎

結合にしているため、それぞれの担当領域を分けており、効率性を重視している。
四半期で若干の人の異動があるものの、システム規模が大きく、複数のシステムを多数のエンジニアで見ているため、自然と担当が決まってきている。ただし、明確に担当を決めすぎると様々な弊害を生み出すので、2～3 人くらいのグループで共有するようにしている。しかし、まれに特定の人しか知らない状況に陥ることがある。
担当領域を分けているため、結局は担当者に任せることになるが、緊急の障害発生時にこれだと困るため、複数人が担当できるように作業分担を工夫している。
不具合を含む部分をリリースした場合は、「切り戻す」手順はだれでもできるようにし、安全にできるようにしている。

A 社事例(1)では、Web 上の共有バージョン管理システムを用いて、コードの共同所有を行っている。コミット権は事業部内のメンバーに限っているが、他の事業部の人にも閲覧可能にし、変更要求（Pull Request）を送ってもらうようにしている。実際のところ、頻繁に Pull Request やソースコードレビューがあるわけではないが、たまに「チームを超えた所有」が発生する。

B 社事例(2)では、他人が作ったソースコードでも自由に編集している。特定の人しか触れない部分は基本的にない。

C 社事例(4)では、プログラマによっては自分の作業範囲を限定したいという気持ちが働き、ソースコードの共同所有に否定的であった。ソースコードの共同所有の範囲も、チームの単位で閉じてしまう。

D 社事例(5)では、Subversion を使って、コードの共同所有を行っている。

H 社事例(12)では、他プラットフォームのコードに対して責任を持つ運用にはせず、自分たちのプラットフォームのコードについてはチーム内の全員が責任を持つ運用にしている。

L 社事例(21)では、初期コーディング時の主担当は割当てたが、バグ修正やエンハンスはだれでも行えるようにした。

M 社事例(25)では、Web 上の共有バージョン管理システムを使用し、他のチームのレビューもできるようにしている。
当初は、ソースコードがレビューされているかどうか未確認の場合にも変更を取り込んでいたが、品質の担保のため、２人以上のレビューがないと変更を取り込めないようにした。
複数チームにまたがるブランチのマージはそれぞれのチームから代表の人を出して行っている。

関連プラクティス

ペアプログラミング
コーディング規約
人材のローテーション
自己組織化チーム

3.2.13. **コーディング規約**/ Code written to agreed standards

“合意形成の練習をしよう”

別名：コーディング標準

要約

チームでの開発を始める時、メンバーで合意形成をしながらコーディング規約を決める。その結果、チームの合意形成の練習になり、コードの統一性が増して可読性が高まる。

状況

より良いプロダクトを作ることを目的として、知識や経験が異なるメンバーから成るチームでソフトウェア開発を始めている。

問題

チームで開発を行うためには、合意形成が大切になる。

合意形成をしながら、コーディング規約を統一したい。

変数名（クラス名、メソッド名）の命名規則が明確になっておらず、ソースコードに統一性がない。

チームで開発する上で、最低限、だれが見ても理解できるコードにしたい。

フォース

◆合意形成の練習をしたいが、最初は対象とする成果物がない。
◆しっかりとした暗黙的な知識の共有を行いたいが、学習のスピードより人員の入れ替わりスピードのほうが早い。

解決策

受け入れられる必要最小限のコーディング規約を作る。ことのき、完全性ではなく、意見の一致を重視するようにする。

最初のイテレーションで、コーディング規約についての話題を上げる。ただし、時間をかけずに簡潔であるようにする。

まず、使用している言語、また環境における標準や慣例を調べ、それに従うかを検討する。

チームの状況にふさわしいコーディング規約を決める。特に大規模開発であったり、チームのメンバーが定常的に入れ替わったりする現場であれば、しっかりとしたコーディング規約を決める必要がある。逆に、小規模かつ人員が入れ替わらないチームであれば、暗黙的な知識の共有で充分である。

コーディング規約に従っているかチェックするツールや開発環境が利用可能かを検討する。

個人のスタイルを尊重する、という規約を定めてもよい。また、ふりかえり時や問題が発生した時のみ、最小限のコーディング規約を決めるようにすることもできる。

コーディング規約は、ふりかえりのタイミングで見直していくのがよい。チームの成長と共に、コーディング規約も進化していく。

コーディング規約がどうしても守れない場合は、ペアプログラミングで作業してみると規約を守りやすくなる。

コーディング規約でチームの理解しやすいコードになれば集団によるオーナーシップが促進される。

留意点

◆コーディング規約を作るための時間が必要となるため、チーム状況を鑑み、「最小限とする」「問題があったことのみ作成する」など、臨機応変に対応することがポイントとなる。
◆コーディング規約に従わない人がいたら、まず理由を聞き、チームとしてよりよい仕事をするために、何ができるのかを皆で考えるようにする。プロフェッショナルとして多様性を尊重し合えるような現場を目指すこと。
◆コーディング規約は「守ること」が目的ではない。何のためにその規約が必要なのかを常に考えておく。

効果

◆ツールでコーディング規約を強制することで、統一的なプロダクトコードになる。
◆ソースコードの可読性が高まる。
◆チーム内の合意形成と、コミットメントに従わないメンバーに対する許容についての練

習になる。

利用例

コーディング規約は、全体の 71%の事例で適用されていた。切り口別でも大きな違いは見られなかった。

A 社事例(0)では、コーディング規約は重視されず、暗黙的なルールを尊重している。またコードレビューは徹底して行っている。
コーディング規約として、「インデントがタブかスペースかは気にするな」という規約があり、また暗黙的ルールとして、「変数の使い回し早めよう」「ユニットテスト可能な設計にしよう」「状態を持ちすぎないように」という暗黙的ルールがある。暗黙的ルールは、コードレビューでメンバーに浸透させている。

A 社事例(1)では、コーディング規約は「Ruby っぽく書けばよい」としている。チームとしては定めてはいないが、全社として Ruby のコーディングの慣習に従って書いている。

B 社事例(2)では、最初にコーディング規約を作ったが守られていなかった。そのため、統合開発環境（IDE）のコードフォーマッタに定義して、警告が出たらそれを直すようにするという運用にした。
コーディング規約の更新頻度が高いため、Wiki に書いている。

C社事例(4)では、公開されているJavaルールのコーディング標準[10]を用いている。統合開発環境Eclipseのコーディング規約をチェックできるツールCheckStyle でチェックしているが、完全ではない。

D 社事例(5)では、最初からコーディング規約を作らず、随時問題が発生する度に作るようにし、徐々に増やしている。

F 社事例(8)では、チームや対象領域が固定されているため、コーディング規約の更新はあまり発生しない。

G 社事例(9)では、漸進的に見直しながら運用している。あまり厳密にしなくてもよいがルールは決めたほうがいいという考えのもと進めていた。規約はイテレーションを繰り返しながら見直していった。
コーディング規約は Wiki で共有している。同一ロケーションで開発しているため Wiki である必要はないが、最新の情報を共有するのには役に立った。
Ruby や Rails はテストコードについてのルール（ケーススタディ）をまとめた。その結果、メンバーによる品質のバラつきがなくなった。

G 社事例(10)では、事例(9)の時に Wiki でまとめた規約を再利用した。

J 社事例(16)では、統合開発環境 Eclipse で CheckStyle を導入した。

L 社事例(21)では、フレームワークの規約は書籍や Web 等に書かれているため、あえてコーディング規約としてまとめていない。
可読性向上のため、コードスタイルを統一するための緩い規約のみを設定している。

関連プラクティス

ふりかえり
ペアプログラミング
集団によるオーナーシップ

[10] http://objectclub.jp/community/codingstandard/

3.2.14. **バグ時の再現テスト** / When a bug is found, tests are created

“バグは新しいテストケース作成の引き金”

別名：なし

要約

バグを発見した後は、修正だけでなく再現テストを書いてから修正する。その結果、修正モレや再発を防ぐことができる。

状況

予期せぬバグ（不具合）が見つかり、直ちに修正しなければならない。

問題

不具合が発見されたら、その原因をつきとめて修正し、再発を防がなければならない。

不具合を修正するためには、原因を特定しなければならない。原因を特定して修正した後には、実際に不具合が修正されたかどうかの確認が必要になる。

再現手順が明らかになっていたとしても、修正後の確認以降も再発していないことをチェックしなければならない。

フォース

◆不具合を修正する前に、開発者の環境で実際に再現するかの確認が必要である。
◆不具合を修正した後に、再現手順で不具合が発生しないことを確認したい。

解決策

不具合を修正する前に、まず不具合を再現させる自動テストケースを作成すること。また不具合の再現をテストで確認した後、修正してテストが成功するのを確認すること。

不具合の発生手順をテストケースにする場合には、その内容によって、ユニットレベル、インテグレーションレベル、受入テストレベルのいずれかでテストが失敗する自動テストケースを作成する。このテストを不具合修正前に実行すると、当然テストは失敗するはずである。

不具合の修正は、このテストケースを成功させるように行う。

この修正プロセスは、テスト駆動開発で開発を進めるプロセスと等価になる。

不具合の再現手順が不明確な場合は、現象から原因を分析した後に、不具合が発生する手順をテストケースとして作成する。

不具合が発見されたら、できるだけ早く修正する。不具合が発見されてから、修正されるまでの時間が長くなるほど、関連する問題を引き起こしてしまう。

不具合を修正した後には、その発生原因の根本原因分析を行い、似たような問題を発生させないための設計改善や、不具合を組み込まないためのプロセス改善などにつなげること。日次ミーティングや、ふりかえりでテーマを取り上げる。

不具合がだれでも修正できるように、集団によるオーナーシップを実現しておく。

バグを早期に検知するために、ユニットテストの自動化、継続的インテグレーションは不可欠である。

留意点

◆ユニットテストをはじめとするテストの自動化に取り組んでいないと、バグの再現テストを繰り返し実行することは難しい。
◆該当のバグを起さないことは確認できるが、副作用（今まで正常に動作していた機能が使えなくなる）がないことを保証することはできない。対応策として自動化された回帰テストを検討するのがよい。

効果

◆不具合に対応したことを保証できる。
◆テスト駆動開発と同じ手順で不具合に対応できる。
◆不具合に対応した後にリファクタリングする際の助けになる 。

利用例

バグ時の再現テストは、全体の 42%で適用され

ていた。部分的にのみ適用しているケースが多く見られた。

B 社事例(2)の場合は、バグの修正時に再現テストを対応している場合と、そうでない場合が混在していた。その違いの理由について回答はなかったが、コードの担当がなんとなく決まってしまう状況の中では、担当者が再現テストを書けるほど熟知している場合と、そこまで熟知していない場合で対応が変わったのではないかと推測する。

C 社事例(4)では、リリースした後は実施していたが、リリース前の開発中は厳格にできていなかった。

関連プラクティス

日次ミーティング
ふりかえり
集団によるオーナーシップ
継続的インテグレーション
ユニットテストの自動化
自動化された回帰テスト

3.2.15. **紙・手書きツール**/ Paper, Handwriting

“字を書いたり、絵を描いたり、貼ったり、剥したり、自由自在”

別名：紙、付箋、情報カード、ホワイトボード

要約

チームの中で何かを議論したり、共有したりする場合には、付箋やホワイトボードを充分に活用しよう。

状況

アジャイル型開発において、チームは原則として同じ場所に集まって仕事をしている。対面でのコミュニケーションが重視されており、直接の対話が行われる。

チームの中では、Wiki システムやチケット管理システム、また表計算ソフトといった様々な情報共有のためのシステムやツールが使われている。

問題

同じ場所で作業をするチームの間で、情報共有やアイデア出しのためには、どんなツールを用いればよいのだろうか？

フォース

- ◆ソフトウェアシステムによる情報共有は情報の整理が便利であるが、「見に行く」手間があるため、即座のフィードバックは得にくい。
- ◆ソフトウェアシステム上で、ブレインストーミングなどの同時に発言をまとめようとすると、キーボードを手にしている人しか入力することができない。

解決策

付箋、情報カード、模造紙、ホワイトボードを使って、チームの間で共有すべき情報を、全員で同時に書き出す。チームが常に目にする壁やホワイトボードに貼り出しておき情報を共有する。

付箋を用いる場面は、企画から開発、レビュー、ふりかえり、運用に至るまで幅広い。例えば、企画段階でも、画面遷移と、その画面の簡単なスケッチを付箋で表すことによって、その場にいる人たちの認識合わせに寄与する。

付箋の使い方は様々である。かんばんやタスクボードを表す方法、タスクボードで用いる方法、そのほかマイルストーンや連絡事項の掲載など、枚挙にいとまがない。

付箋は一瞬で共有でき、変更や修正を行いやすいなどの特性を持つため、イテレーションで行うタスク、休暇取得などの短期情報の管理など、短期的で流動的な状況を把握する場面で用いられることが多い。一方、長期的なプロジェクトの状態やリリース計画、WBS などの管理は、電子媒体で外部組織と共有するほうがよい。上記の特性から、付箋はオンサイト顧客など、対面でのコミュニケーションを強調するアジャイル型開発に適応しやすいツールである。

情報カードは、付箋とは違い粘着力はないが、紙が厚いため、ユーザーストーリーのように長期的に貼り出す情報を書き出しておくのに向いている。

またはテーブルの上で、重ねる、並べる、並び換えをする、といった用途に向いている。
軽量なオブジェクト指向のモデリング手法である CRC カードでは、情報カードにクラス、責務、協調クラスを記述して、机や壁の上に並べてモデリングを行う。[ベリン 2002]

情報カードを貼り出す場合は、別途再剥離可能な粘着剤やマスキングテープを使用する必要がある。

模造紙は、直接何かを描く、あるいは、付箋などを貼る台紙として使われる。また壁に直接何かを貼るのではなく、模造紙に貼ったものを壁に貼ることによって、模造紙を壁から剥すことで、貼り出した掲示物を持ち運び可能にすることもできる。

ホワイトボードは、何度も書き直しできる便利なツールである。ミーティング時に 1 人あるいは複数人で同時に絵、文を描きながら議論をすることで、描いた内容がそのままミーティングの議事のログになる。また軽量な設計セッションをホワイトボードに描きながら行うことで、設計作業と同時に、関係者間での意識共有が促進できる。

ホワイトボードは、手軽に描けるツールであるが、描いた内容を永続化しておきたい、あるいは長期間掲示しておきたい場合には、写真あるいはコピーをして別途保存する。ホワイトボードは常に消して、新たに描くことができる状態にしておくのが望ましい。

共通の部屋は、紙・手書きツールを有効に用いる前提となる。

ふりかえりや、リリース計画ミーティング、イテレーション計画ミーティングでは付箋やホワイトボードは広く使われている。

かんばん、タスクボード(タスクカード)は、ソフトウェアシステムで使う前に、まず紙・手書きツールで実現してみると、自分達なりの工夫がしやすい。

CRC カードやホワイトボードを用いて、議論しながら設計を行うことでシンプルデザインの助けになる。

バーンダウンチャートは表計算ソフトで計算させてグラフを作成することもできるが、あえて紙を貼りチームで計算しながら線を書き込むことで状況の共有を促進させることができる。

留意点

付箋などを貼り出す場所を確保することが重要になる。部屋を該当チームや関係者のみでプロジェクトルームとして使用できる場合は、壁を全面使うことができる。

壁がキャビネットなどに占有され貼り出す場所がない場合は、他部署と調整の上、可能ならばそれらの備品を移動し場所を確保する。

壁がない場合は、可動式のホワイトボードや直立する支持体のイーゼルなどを用いて、付箋を貼る場所を確保することもできる。

外部の人が覗くことができるような物理的セキュリティの問題がある場合は、壁面や可動式ホワイトボードなどを活用できないため、A4 サイズの折りたたみフォルダに付箋を貼って、小さいながらも場所を確保する。

付箋などの内容を、編集可能なデータ化したい場合には、別途手入力などの手間がかかるため、データとして再入力する価値が本当にあるかを事前に見極めること。

コンプライアンスの都合で、オフィスにだれでも見えるような情報を貼り出してはいけない現場も存在する。

効果

◆チームは、現在の状況をいつでも見ることができる。様々な変更や修正を素早く他の人に伝え説明でき、誤解や行き違いの発生を防ぐ。
◆付箋を使わない場合は、管理ツール上の情報と現状との差異が発生していたが、付箋を使うことによって、最新の状況が表現できる。
◆紙・手書きツールは柔軟であるため、現場の創意工夫を活かしやすい。
◆ただし、分散拠点で利用しようとするのは難しい。

利用例

紙・手書きツールは、全体の 90%の事例で適用されていた。手法別の切り口では、XP の採用事例において 100%利用していた。事例の中にはデジタル→アナログの移行例も、アナログ→デジタルの移行例も存在した。

A 社事例(1)では、Redmine のチケットに気づかないことがあるため、直近のチケットは付箋にして、貼り出している。

C 社事例(4)では、ふりかえりで付箋を使用している。タスクカードは情報カードに決まったフォーマットを印刷しているため、付箋は使用していない。

G 社事例(9)では、イテレーション計画ミーティング、タスクボード、ふりかえり、など様々な場面で付箋を用いていた。ホワイトボードは大きいものと小さいものを併用しており、設計やミーティングなどの様々な場面で利用していた。ユーザーストーリーについては、表計算ソフトに記入したものを印刷してラミネートした状態で開発者に提示していたため~~が~~、その場で修正することが難しかった。

G 社事例(10)では、模造紙にタスクボードやユーザーストーリーマップを貼り出して、別拠点にいるプロダクトオーナーに会う際には必ず持参していた。

D 社事例(5)では、特殊コーティングされた何度でも書いて消せるホワイトボードの特性を持つ紙を用いて、修正や再利用ができるようにしている。

H 社事例(12)では、チームや内容によって色分けした。色を見ればそれがどういった

内容のものなのかある程度わかる様になった。

M 社事例(25)では、付箋の消費量が多いため、磁石付きのプレートにホワイトボードマーカーで何度でも記載するようにし、再利用できるようにした。

関連プラクティス

共通の部屋
ふりかえり
リリース計画ミーティング
イテレーション計画ミーティング
かんばん
タスクボード(タスクカード)
バーンダウンチャート
シンプルデザイン

参考文献

[ベリン 2002] ベリン, D. サッチマン, S. (2002)『実践 CRC カード ロールプレイとブレインストーミングによる大規模システム開発手法』 ピアソン・エデュケーション

3.3. チーム運営・組織・チーム環境

3.3.1. 顧客プロキシ/ Customer Proxy

“顧客になりかわりチームに顧客の声を届ける”

別名：なし

要約

顧客がチームの側にいないならば、顧客のように考える顧客プロキシを置く。その結果、チームは、顧客プロキシから顧客の抱える問題やニーズについて聞くことができる。

状況

顧客が開発以外の仕事に時間を取られており、開発チームに対しての時間を割くことができない、または新しいプロダクトのため、顧客がまだ存在していない。

企業文化などにより、開発者と顧客が隔たれており、コミュニケーションが取りづらくなっている。

問題

顧客の声を聞けていないために、顧客を無視した設計、開発を行ってしまう。結果、顧客の期待するプロダクトが出来上がらない。

顧客の声に基づく設計ではなく、チームが独自に考えた設計に合うシステムの挙動を仮定してしまう。

顧客とのコミュニケーションが不充分なため、顧客の抱える問題やニーズをチームが捉えることができない。

フォース

◆顧客は忙しいため、自分自身が抱える日常業務に追われ、システム開発に向ける時間を充分に取ることができない。

解決策

顧客の代わりに顧客のように考え、開発者とコミュニケーションを取るロールをチーム内に置く。

ビジネスアナリストやプロジェクトマネージャなど、顧客との距離が近いロールに顧客プロキシを務めてもらう。

チームは顧客プロキシを本当の顧客のように扱う。プロダクトに対するアイデアや要求を顧客プロキシから聞く。

顧客プロキシは顧客の立場に立って、開発チームとコミュニケーションを取り、ユーザーストーリーを書いたり、プロダクトバックログのメンテナンスも行う。

顧客プロキシは顧客に変わって、受入テストを行うこともある。

理想は、開発者自身が、顧客プロキシも務めることである。

留意点

◆顧客プロキシが顧客の抱える問題やニーズについて充分に理解していなければならない。
◆顧客プロキシは顧客の代理でしかないため真の顧客の声を汲み取り続ける必要がある。
◆真の顧客と対話できないプロダクトの場合、あるいは顧客からの権限委譲が可能な場合はプロダクトオーナーを検討する。
◆顧客組織に開発者が駐留することも考えられるが、成果が目に見えて分かりやすい目前の短期的、局所的な問題への対処に追われる可能性がある。また、特定の部署への配置は、部署横断的な問題の発見と対処を難しくする。

効果

◆顧客が開発に充分に時間が取れなくても、チームは顧客の問題やニーズを把握し、期待するプロダクトを開発することができる。

利用例

顧客プロキシは、全体の 42%の事例で適用されていた。契約の切り口で見ると、顧客という概念が存在しにくい自社開発では 29%、受託開発では 59%と、周囲の状況によってはっきりと適用率が異なっていた。

C 社事例(4)の場合は、要件チームという要求を定義するチームを結成して顧客プロキシとしていたが、顧客から権限の移譲は行われておらず、あくまで顧客のサポートという位置づけであった。

D 社事例(5)では、「サービスプロデューサー」という役割を置いており、チームとビ

ジネス部門との間に立ち、プロキシとなっている。

K 社事例(20)では、開発チームが顧客を完全に巻き込み一緒に開発を進めたいと考えていることに対して、顧客は自分の業務範囲外に踏み込みすぎと感じている。
開発チームに顧客プロキシを実施するだけではうまくいかない。優先順位を付けてくれるプロダクトオーナーより、一緒に「作っていく」人を巻き込みたい。
開発チームと顧客プロキシの間で信頼関係ができていれば、仮に実際の顧客が顧客プロキシとは異なる決断をした場合でも、顧客の意向に添えるように前向きな気持ちになれる。

L 社事例(21)では､それぞれの業務担当者を選任し、顧客プロキシになってもらった。

M 社事例(25)では、エンドユーザにインタビューやテストを依頼している。

関連プラクティス

プロダクトオーナー
オンサイト顧客

参考文献

Coplien, J. Harrison, N. (2004). Organizational Patterns of Agile Software Development, Peason Prentice Hall

3.3.2. **オンサイト顧客** / The customer is always available/Constant user interaction

“いつでも顧客と相談することで
ムダな機能を作らない”

別名：全員同席

要約

ムダなソフトウェア機能が作られてしまうことを防ぐため、開発者と顧客が常に会話ができる環境で仕事をしよう。その結果、顧客と開発者のコミュニケーション量が増えて、顧客が望むソフトウェアが作られやすくなる。

状況

アジャイル型開発は、顧客のビジネスに価値のあるソフトウェアを開発し提供することが第一の目的である。開発者は、顧客の要求がどれだけビジネスに価値があるのか、何のために必要なのかを知り、開発を進めていく。

その最中に様々な要因により変わっていく要求や詳細な仕様について開発者と顧客が密にやりとりする必要がある。

問題

文書やオンライン上のやりとりではコミュニケーションは充分とは言えない。

両者の間の誤解、コミュニケーションミスによって、使うことのないムダな機能、推測に基づく誤った仕様の機能を作ってしまうリスクは常に存在している。2002 年のStandish Report Studyによると、よく使われる機能は20%しかないという調査結果もある[11]。

また、短期間に動作するソフトウェアを提供していくためには、開発者と顧客の間で疑問点をできるだけ早期に解決して開発速度を向上させたい。

フォース

◆文書やメールでコミュニケーションを取ることは可能だが、文書だけで全てを伝えようとすると、誤解や行き違いなどが発生しやすい。
◆電話でのコミュニケーションは会話による情報交換が可能だが、1 対 1 のコミュニケーションになってしまう。また、タイミングを計るのが困難なことが多い。
◆開発者と顧客が会う機会が少ないと、一度にまとめて質問事項をやりとりせざるを得ない。

解決策

顧客と開発者が同一の場所で作業できる環境を整えて、対面でのコミュニケーションを充分に取れるようにすること。

顧客が開発者と常に同席して仕事ができるようにしておくことが理想であるが、常に同一の場所で、直接コミュニケーションができない場合は、できるだけリアルタイムにコミュニケーションが取れるようにツールを用いるか、対面のコミュニケーションを実施する頻度をできるだけ増やすこと。

顧客が開発者と場所を共にして作業をすることが難しい場合は、訪問頻度を増やして対応をするのがよい。

顧客と開発者が同一の建物ではあるがフロアや部屋が異なる場合には、席替えによってできる限り両者間の距離を近付ける努力をする。

顧客が開発メンバーと同一拠点で一緒に作業することができるならばチーム全体が一つに近づくことができる。

同一拠点での作業については共通の部屋をデザインする。

留意点

◆開発者とコミュニケーションを取る顧客に充分な権限が与えられていない場合は、その場で意思決定ができなかったり、後で権限のある人物に決定を覆されてしまったりする恐れがある。
◆開発者と顧客を近付けることで、互いの発言が直接聞こえてしまう。その悪影響の例として、顧客が開発者への不満を口にした場合に開発者のモチベーションが低下してしまう恐れがある。単に席を近づけるだけでなく、両者が信頼関係を築きながら作業ができるようにする必要がある。

[11] http://www.featuredrivendevelopment.com/node/614

◆どうしても顧客が忙しい場合は、タイミングを決めて頻度を増やして直接話す時間を設けるようにする。タイミングとしては日次ミーティングのタイミングなどがよい。
◆遠隔地間など、どうしても同一拠点で同席できない場合は、電話会議システムやオンラインミーティングシステムなどを用いて随時会話ができるようにする。オフショアなど海外拠点とやりとりをする場合は、タイムゾーンによる時差を考慮しなければならない。

効果

◆顧客は開発しているソフトウェアをいつでも見ることができる。ビジネス上の変更を素早く開発者に伝え説明でき、開発者との誤解や行き違いの発生を防ぐ。
◆開発者は疑問点を相談し、仕様上の意志決定を顧客に素早く依頼できる。作ったもののフィードバックを得られやすくなり、ムダな機能が作り込まれることを防ぐ。
◆同じ場所で働くことで、開発者と顧客の間に信頼関係が生まれ、小さな相談も気軽にできるようになる。

利用例

オンサイト顧客は、全体の 56%の事例で適用されていた。規模別の切り口では、小規模が 50%の適用率に対し、中大規模は 75%であった。契約別の切り口では、一見実現の可能性が高そうな自社開発での適用率が 53%、対して受託開発では 59%の適用率であった。アジャイル型開発を採用する以上、顧客側も開発とのコミュニケーションに責任を持って関っていた部分が大きいと推測される。

C 社事例(4)では、チャットシステムを用いて、常に顧客とコミュニケーションが取れる状態にしていた。また週に 2～3 回の頻度で対面でのコミュニケーションをとる時間を作った。

K 社事例(20)では、顧客は常に同席はできないが、週 3 日、顧客が開発ルームに訪れた。最初は開発者が顧客との距離感をとりあぐねていて、何を聞けばよいかがわからなかったが、プロジェクトが危機的状況に陥った時に、顧客がオフィスに常駐するようになってから、状況の共有、優先順位付けの依頼、仕様についての相談など、開発者が顧客と何を話すべきか、ということがわかるようになった。
その一方、開発者とコミュニケーションを取っていた人物は充分な権限がない状態であったため、後で決定が覆される場面も何度かあった。しかし開発者と顧客との間で信頼関係が醸成されていたため耐えることができた、と述べられている。

H 社事例(12)では、チームの隣の席にプロダクトオーナーの席が来るように席替えを依頼して、常にコミュニケーションが円滑になる様に工夫した。

J 社事例(17)では、平均週 2 回、顧客と対面での課題解決、スケジュール確認を実施した。コミュニケーションの方法として対面以外でも、電話、チャット、プロジェクト管理ソフト（Redmine）を活用し、常に開発者と顧客の間でコミュニケーションが円滑に進むように留意した。

B 社事例(2)では、プロダクトオーナーが非常に多忙な状況でありオンサイト顧客を実施するのが難しかった。顧客は日次ミーティングには常に出席するようにして、開発者とのコミュニケーションを取っていた。

関連プラクティス

チーム全体が一つに
共通の部屋

3.3.3. **プロダクトオーナー** / Product Owner

“プロダクトへの責任を持ち、ビジョンと情熱でチームを牽引する”

別名：なし

要約

要件の優先順位や仕様がなかなか決まらない場合は、プロダクトの結果に責任を持ち優先順位や仕様を決める役割を設ける。その結果、プロダクトについての決定を素早く行うことができる。

状況

開発するシステム、プロダクトの周囲には、顧客や様々なステークホルダーがいる。様々な人々がシステムに対して意見を持っている。

業務システムであれば、システムの企画者、利用者、そして予算を握っている経営者がいる。B2C サービスであれば、上記の役割以外に、営業、マーケット部門なども含まれる。それぞれの要望をプロダクトにうまく反映したい。

要件は状況が変わる度に見直し、優先順位や仕様を変えていかなければならない。また、開発期間を通じて開発者とコミュニケーションを取りながら、要件のメンテナンスを行っていく必要がある。

問題

「コックが多すぎるとスープがうまくできない」[12]

システムについて意見を持つ人が複数いること自体は一般的だが、システムに対する要望が多方面から出ても収集がつかなくなる。システムによって解決したい問題、顧客にどのような価値を届けるかによって、意見を収束させなければならない。

システムの価値を最大化するために、価値の高い要求から順に優先順位付けしなければならないが、責任者が複数存在すると、その擦り合わせに時間がかかることが多い。そのため優先順位を付けるのに時間がかかる。

要件についての意思決定が遅れると、開発が先に進まない。

フォース

- ◆顧客に責任者になってもらいたいが、顧客が時間を割くことは難しい。
- ◆新規でプロダクトを開発する際に、複数の人が意見を言って、要望がまとまらない。

解決策

プロダクトの成果に責任を持つ役割を決めて、1 人の個人に割当てること。その人物に要件の優先順位付けや、仕様の決定を行う権限を持たせ、頻繁に開発者と対話しながらプロダクトの価値を最大化するために作業を行うこと。

プロダクトオーナー（Product Owner；以下、PO）は、プロダクトの問題領域のエキスパートであることが望ましい。この役割はプロダクトのビジョンを策定し、ROI（Return Of Investment：投資対効果）を最大化するために要件の適切な優先順位付けを実施する。

また PO は専任であることが望ましい。要件の取りまとめ、優先順位付け、詳細化、仕様化、様々なステークホルダーとの調整、開発者とのコミュニケーションなど作業は多くある。PO の作業が滞ってしまうと、開発速度に多大な影響を与える。

状況の変化、実際に動作するソフトウェアをレビューすることによって、新たな要件の追加や変更が発生するため、素早く要件の順序付けや、順番の入れ替えを行わなければいけない。

要件の仕様、完了条件についても、PO が要件を開発者に引き渡す前に決めていく必要がある。開発が始まった後も、開発中の細かい仕様の疑問点を解消するために、PO は開発者と密にコミュニケーションを取る必要がある。

PO は、スクラムにおいて定義された役割であるが、現在ではスクラムを越えてアジャイル型開発全体を通じて必要な役割として認知されている。

PO が遠隔地や開発現場と離れた場所にいる場合はオンサイト顧客を検討するのがよい。

[12] “Too many cooks apoil the broth” 日本では「船頭多くして船山に登る」

スクラムではPOとは別に、開発プロセスがうまく回っていることを支援するスクラムマスターを立て、開発プロセスの改善を促進する。スクラムマスターはPOの支援をする役割でもある。

POは開発者が自己組織化チームとして動けるように、明確な目標とビジョンを示す必要がある。ビジョンを提示するためにインセプションデッキを用いるとよい。

POはイテレーション計画ミーティング(スプリント計画ミーティング)の前までに、充分に詳細化されたプロダクトバックログを提示しなければならない。

留意点

◆POが開発者と頻繁に会うことができないと、役割として成り立たない。複数プロジェクトを兼任している場合などは、特定のプロジェクトに専任化するのが望ましい。
◆POを任命された人物に権限が委譲されていないと、意思決定の速度が遅れる、決定が覆されるなどの問題が発生するためうまくいかない。
◆POは開発者とだけでなく、プロダクトを取り巻く様々なステークホルダーとコミュニケーションを取り調整していく必要があるため多忙になりがちである。
◆複数のチームに別れて開発しており、各チームにPOを配置する場合、1チームのPOを複数人で構成する場合は、PO同士の合意形成を充分に行なっておかないと、PO毎に意見が異なり開発チームが混乱してしまう。
◆POがイテレーション計画ミーティングに先立って要件を詳細な粒度に分割し、完了条件を明確にしていないと、開発者は開発をスタートすることができない。事前に次のイテレーションで実現したい要件の具体化、詳細化を行うミーティングを開発者の協力を得て実施する必要がある。

効果

◆プロダクトのコンセプトに沿う形で要件をまとめ、素早く意思決定することができる。
◆開発者と密にコミュニケーションを取ることで誤解が生じにくい

利用例

プロダクトオーナーは、全体の69%で適用されていた。特に契約別での自社開発では82%の事例で適用していた。受託開発でも50%の適用率であり、顧客の存在があってもプロダクトオーナーという役割に意味がある証拠と言える。

B社の事例(2)においては、管理職にPOを任せていたが、開発者と会う機会が少なくうまく回っていなかった。POを担当者に変更してからうまく回るようになった。このことから、POは実際にシステム開発に時間を割ける人でないと効果がないことを痛感した。

C社事例(4)の場合は、最初は担当をPOとして進めていた。後になってPOと顧客企業の社長の意見が異なることが発覚した。そのためPOが即時に判断できない状況においては、顧客起業の社長を含める経営陣が即座に判断するような体制に移行した。

G社事例(9)の場合は、POは情報システム部門の担当人物であった。実際の利用者は工場の現場の作業者であり、POと現場の責任者の間で調整した結果を開発者に提示していた。

G社事例(10)の場合は、POは特定の製造ラインのエキスパートであったが、それゆえに全体最適化の視点に欠けているという指摘が顧客の管理部門からあった。全体最適化を行う部門からすると、POが決めた機能は特定の製造ラインの部分最適化になっていると思われていた。このことから、全体を見ているIT責任者と密にコミュニケーションを取っていれば部分最適にならないよう方向修正ができたのではないかと回想していた。

K社事例(20)の場合は、POの役割を顧客に求めるのは無理であると感じていた。開発現場に立ち寄り開発者とのコミュニケーションを取っていた人物には権限がなく、POは部長職であった。さらに、社長権限により決定が覆されることがあった。権限はないが、開発者と密にコミュニケーションを取る顧客と一緒に働くことで信頼関係を築いていけば、後で決定を覆されても、開発者が我慢できると話していた。

M社事例(25)の場合は、不特定多数の利用者に提供するサービスを開発していた。そのため利用者には直接会うことはできなかった。ここでは、PO役のビジネス企画者だけでなく、開発者自身も提供サービスについて考えるという姿勢が徹底していた。

関連プラクティス

オンサイト顧客
ファシリテータ(スクラムマスター)

自己組織化チーム
インセプションデッキ
イテレーション計画ミーティング

参考文献

Galen,R. (2009). SCRUM Product Ownershiop: Balancing Value From the Inside Out, RGCG, LLC

3.3.4. ファシリテータ(スクラムマスター) / Development process and practices facilitated by a dedicated role (Scrum master)

“潤滑油の役割でなめらかな開発を！”

別名：なし

要約

開発チームの内外に調整が必要であれば、ファシリテータの役割を設置する。その結果、開発が進みやすくなる。

状況

アジャイル型開発を行っており、開発チーム内部の調整や外部との調整が日常的に発生している。

アジャイル型開発では、イテレーションや日次ミーティングのように守るべき開発のリズムや人間関係性の調整が、より着目されている。

問題

自らの状況を客観的に見ることができない組織では、問題を認識し、改善や運用が難しい。

顧客やステークホルダーからのプレッシャーに対して、適切なバランスを保つことが難しい。場合に応じて、関係悪化や組織硬直などの問題が発生する。

フォース

◆開発者もプロダクトオーナーもそれぞれの意図があり調整したいが、適切なバランスが難しい。
◆開発プロセスの維持や改善をしたいが、第三者目線で把握することが困難である。

解決策

開発プロセスやプラクティスをファシリテートする明確な役割を設けよう。

スクラムにおいては、スクラムのフレームワークを維持するためのスクラムマスターを決める。
ファシリテータは、関係者との調整や、チームの活気を維持し、適切なペースで仕事ができるように取り計らう。
いくつかの事例では、チーム全員が持ち回りでファシリテータを経験し、全員でファシリテートできるようにしている。また、第三者の視点を大切にし、個別の役割を尊重している事例もあった。

また、スクラムマスターはスクラムの中でも重要な役割である。以下の異なる役割を演じる必要がある。

◆触媒
- ♦ 成功のために協調作業を促進する

◆完了マスター
- ♦ チームに自分達の「完了」の定義を守らせる

◆牧羊犬
- ♦ チームがプロセスを脱線しないようにする

◆鏡の剣士
- ♦ チームに自分達の改善すべき点を気づかせる

◆コーチ
- ♦ 改善に挑戦させてアドバイスを送る

◆チアリーダー
- ♦ チームを勇気づけ、ポジティブフィードバックを送る

◆ファイアウォール
- ♦ チームを外部からの妨害から守る

◆プロダクトオーナートレーナー
- ♦ プロダクトオーナーがよりよくなるために支援する

ファシリテータ（スクラムマスター）は、チームを一歩引いて全体的に見ることが必要である。普段の仕事場での観察が重要だ。

チームが成熟するまでは、ファシリテータが率先して、様々なミーティングや活動の進行や、準備を行う。しかしチームがファシリテータ(スクラムマスター)に依存するようになってはいけない。

チームのメンバーが、次のファシリテータの役割をこなせるようになるために、後進を育てていくことを常に意識しておく。

当初は、日次ミーティング、イテレーション計画ミーティング、イテレーション計画ミーティング、ふりかえり、スプリントレビューのような会議体の進行を務めてチームにそのやりかたを伝えること。

スクラムマスターは、チームのパフォーマンス

を妨げる障害(インペディメント)を率先して発見し、チームに伝えて改善を促す(鏡の剣士)。柔軟なプロセスや組織に合わせたアジャイルスタイルを先導する。

スクラムマスターはプロダクトオーナーの状況を観察して積極的に支援する。

留意点

◆スクラムを採用している場合は、認定スクラムマスター研修で役割や内容を理解するとよい。
◆ファシリテータは、プロダクトの作成に直接的に関係しないため、余計なコストとして見なされることがある。
◆プロセスの遵守とプロダクトの価値のバランスを取るため、スクラムマスターとプロダクトオーナーは兼任してはいけない。

効果

◆開発チームの活気を持続することができる。
◆プロダクトオーナーやチームの外部のステークホルダーとの調整が行える。

利用例

ファシリテータ（スクラムマスター）は、全体の 73%の事例で適用されていた。システム種別の切り口では、B2C サービスよりも社内システムが、契約別の切り口では、受託開発よりも自社開発が 20%以上も多く適用していた。

A 社事例(1)では、最初のうち、ファシリテータは必要ないと思っていたが、ミーティングの時に同僚がファシリテータとして入った際に、話しやすくなるという効果を実感した。

B 社事例(2)では、特にスクラムマスターという役割を置かず、全員がスクラムマスターとしてファシリテートしている。以前はファシリテータ（スクラムマスター）を持ち回りでやっていたが、全員にファシリテータ（スクラムマスター）の意識が浸透すると、特別にスクラムマスターの役割を設けなくても、各人の判断で必要に応じてファシリテートできるようになった。
スクラムマスターは手を出してはいけない、決めてはいけないというルールがあるが、スクラムマスターになった人も、自ら開発したいという希望があったためスクラムマスターを設置するのを止めた。
全員がファシリテータをできるようになり、属人性を排除できた。ファシリテータの苦労を全員が体験し自律的な行動に効果があった。

C 社事例(4)では、3 人で管理チームを構成し、計画ゲームやふりかえりの進行、プロセスがまわっていることの監視などをファシリテートしていた。
開発チームとプロダクトオーナーの調整、顧客が開発現場に来た（オンサイトの）時の段取り、プロジェクト進行の障害を取り除くなど、プロジェクトが回るようにした。

F 社事例(8)では、明確なスクラムマスターはいない。最初は必要であったが、スクラムマスターの立場を全員で理解して実施するようになった。

G 社事例(9)では、最初、アジャイルコーチを設置し、アジャイルコーチがファシリテータの代わりをやった。その後、開発者が認定スクラムマスターの資格を取得し、「開発者視点」ではなく「顧客視点」が加わった。

G 社事例(10)では、当初はスクラムマスターを設置していたが、その後顧客との関係もよく、開発メンバーも慣れてきたので明確なスクラムマスターは立てていない。しかしながら、スクラムマスターがいなかったため、プロダクトオーナーに助言できていなかったことに後で気づいた。アジャイル型開発を初めて経験した顧客が、スクラムマスターの立ち位置が有意義であると評価していた。

H 社事例(14)では、プロダクトオーナーは日本におり、開発チームとスクラムマスターは海外（オフショア先）に滞在している。

K 社事例(20)では、スクラムマスターはシステムの中身に踏み込んで、コードを見たり、プロダクトオーナーの代理のようなことをやったりしたが、順調な様子ではなかった。
開発に直接関わらないスクラムマスターは、内部のことがすぐにわからなくなってしまう（アンチパターン）。
ふりかえりで、スクラムマスターがボトルネックであることをよく指摘される。しかしながら、スクラムマスターがいないと、チームが暴走をしがちになるので、チームに活気や安定を届けていることに寄与している。

L 社事例(22)では、まずはスクラムマスターを試してみることを優先し、やってみて違和感のあること、腑に落ちないことは皆で議論して改善していくことをコミットメントして進めた。

M 社事例(25)では、今までは、エンジニア兼スクラムマスターだった。しかしながら、兼任すると、1. 近視眼的になるので改善が進まない 2. 他のメンバーが受け身になる、という問題が発生したため、スクラムマスターに専任するようにした。
スクラムマスターは複数のチームを担当し、適正のある人のみが担当するようにしている。

関連プラクティス

リリース計画ミーティング
イテレーション計画ミーティング
ふりかえり
スプリントレビュー
組織に合わせたアジャイルスタイル

参考文献

ScrumPLoP, ScrumMaster, https://sites.google.com/a/scrumplop.org/published-patterns/very-old-patterns/scrummaster

3.3.5. **アジャイルコーチ** / Agile Coach

“チームは先人に連れられてアジャイルの門を開く”

別名：なし

要約

もしチームが初めてアジャイル型開発に挑戦しようとしているなら、アジャイルコーチに導入を手伝ってもらうべきである。その結果、より確実にチームがアジャイルへの変化を体験できる。

状況

チームはアジャイル型開発に取り組んだ経験が皆無であるため、アジャイルへの適応を最小限の痛みに抑えて、大きな失敗を回避したいと考えている。

問題

アジャイル型開発は教科書通りに実施しても、簡単にはうまくいかない。

アジャイル型開発は、様々なプラクティスと相互関係、その裏にある原則、価値に基づく姿勢により構成されている。

プラクティスや原則には、従来のやり方とは矛盾するものもある。例えば「リーダーの指示を待つのではなく、メンバーが自ら考えて行動する」などは、根本となる考え方が異なり、違った行動が求められる。

そのため、アジャイル型開発の採用初期は、行っていることが正しい方向に向かっているのか、そうでないのかを、チームメンバー自身では判断しづらい。

フォース

- ◆書籍などではプラクティスを一度に紹介しているが、どの順番で適用していったらよいかがわからない。
- ◆人は新しいやり方の効果に不安があると、古いやり方に戻しがちである。
- ◆プラクティスの背後にある価値観、姿勢は本だけで学ぶことは難しい。
- ◆自分達だけで実践して学びを得たいが、プロジェクトに時間的猶予がない。

解決策

アジャイル型開発の経験者を探して、コーチとして招聘し、チームに対してのアジャイルの導入や改善を手伝ってもらう。

コーチは単にアジャイル型開発の経験者というだけでなく、チームに経験を伝えることができ、実際にメンバーをファシリテートしながら、様々なプラクティスを実践できる必要がある。

チームが手本を元に自分達で実施できるようになった後は、助言を与え、問題解決のためのヒント、更なる改善のための問い掛けを行う。

コーチはチームが自分達で様々なプラクティスを実践できるようにし、自分達自身で改善していける高いパフォーマンスのチームとして自立させることが目標である。

アジャイルコーチは、組織の中にいる経験者を呼ぶこともあれば（社内コーチ）、アジャイルコーチを生業としている社外コーチを招聘する場合もある。

組織内にアジャイル型開発を広く展開していきたい場合には、社内コーチを育てて、各プロジェクトの発足時に参画して支援することもある。

コーチの形態には、フルタイムでチームに参加してコーチする場合や、週に 1～3 回アジャイルのミーティングを中心に参加する場合がある。

コーチは問題解決をチームに代わって行うのではなく、問題解決に立ち向かう考え方、方法を教え伝え、チームが間違った道に進みそうになるのを気づかせる役割である。問題解決はチーム自身が行うように促す。

コーチは「何をするか（Do Agile）」をチームに教え伝えるだけでなく、「どうあるか（Be Agile）」を、日々の言動、姿勢からチームに示す存在でもある。

コーチには得手/不得手がある場合もある。そのためコーチは 1 人とは限らず、得意分野の異なる複数人のコーチに支援してもらうこともある。

アジャイルコーチに参画してもらった後は、日次ミーティング、イテレーション計画ミーティング、ふりかえりのファシリテータを最初に依頼するとよい。これらはチームのリズムを作り、改善を促すからだ。

特に開発技術系のコーチを依頼する場合は、

ペアプログラミングを一緒に行ってもらうことで、実際の開発のリズムや細かな作法を学ぶことができる。

組織に合わせたアジャイルスタイルを確立したい場合はアジャイルコーチに相談するとよい。

留意点

◆コーチを招聘する際には、チームとの相性を知るためにも最初に講演やミーティングのファシリテートをしてもらうのがよい。
◆チームがコーチに問題解決を期待するのは、コーチに対する認識がズレている証拠である。
◆スクラムマスターのスキルが高い場合は、スクラムマスターがアジャイルコーチの役割を兼ねる場合もある

効果

◆チームがアジャイルのプラクティスを最短で学ぶことができる。
◆プラクティスだけでなく背景にある考え方を学ぶことができる。
◆チームが短期間で自立できるようになる。

利用例

アジャイルコーチは、全体の 48%の事例で適用されていた。システム種別の切り口では、B2C サービスでは 34%であったのに対し、社内システムでは 63%が適用していた。

B 社事例(2)の場合は、社内の経験者がアジャイルコーチとしてチームの面倒を見ていた。時期によって、プロジェクトには所属せずに、複数のチームのコーチをする時期と、プロジェクトに所属して、プログラマの仕事をする時期を切り替えていた。これには、アジャイルコーチが現場感を失わないという効果がある。

E 社事例(6)では、社外コンサルタントに教育や導入の支援を依頼してアジャイル型開発を開始していた。

G 社事例(9)の場合は、プロジェクト発足当初から社外アジャイルコーチに参画してもらい、基礎教育を実施し、ミーティングなどでのファシリテートを行いながらチームにやり方を伝えていた。プラクティスの導入についてもチームの状態を見て適切なプラクティスを紹介していた。

K 社事例(20)の場合は、プロジェクトが大炎上した後に、アジャイルコーチを外部から招聘してチームを見てもらった。主にスクラムを中心としたプロセスやチームの状態を見るコーチと、TDD や品質面で支援してもらうコーチという役割の違う 2 人のコーチを招いた。

M 社事例(25)の場合は、スクラムマスター経験者が外部の研修を受けた上で、プロダクトオーナーやスクラムマスターの先生役（コーチ）として支援した。また、社内コーチの指導のために、社外のコーチを招聘した。

関連プラクティス

イテレーション計画ミーティング
日次ミーティング
ふりかえり
ペアプログラミング
組織に合わせたアジャイルスタイル

参考文献

Adkins,L. (2010). Coaching Agile Teams, Addison-Wesley Professional

3.3.6. **自己組織化チーム** / Team members volunteer for tasks (self-organizing team)

“自分たちの力で、自分たちの仕事をしよう”

別名：なし

要約

チームメンバーが指示されて動くのではなく、各人の判断で動けるようにする。その結果、透明性の高い強い組織になる。

状況

チームで仕事をするようになっている。

従来型の開発では、プロジェクト・マネージャ（PM）が、WBS などを用いて、モレやダブりがないようにタスクに分割し、担当者にアサインする。メンバーは、排他的に指示された仕事の領域をこなすことに専念する。

問題

複雑な領域の詳細までPMが把握することは困難で工数がかかる。

領域は複雑に絡み合っていることが多く、分離することは困難な状況が発生しやすい。
管理作業が PM に集中し、ボトルネックになりやすい。

フォース

◆全体を把握したいが、１人１人へのヒアリングは工数がかかり、最新の状況が見えにくい。

解決策

タスクを主体的に志願するボランティアのように行う自己組織化されたチームを作る。

自己組織化は、それぞれのメンバーがパターンを認識しながら、自らの判断で動く。そのため、権限委譲や大まかな枠組や方針について認識を得られる。

自己組織化チームで、最も大切なことは信頼である。信頼関係を築くには、成果を出すことが一番の近道である。

自己組織化チームを実現するには、チームの構成員がどの仕事にも自律的に取り組めるように多能工化を促進するとよい。

自己組織的に行動するためには、チームとしての目標、プロダクトとしての目的が明確になっている必要がある。インセプションデッキを用いて、チームの目的を明らかにするとよい。

指示されるのではなく、日次ミーティングやプロダクトバックログや、スプリントバックログなど、アジャイル型開発のプラクティスを通して、自らの判断でタスクをこなす。

ふりかえりを継続的に行い、自ら改善する。

留意点

◆既存の PM は、助言のつもりが指示や命令になり、自己組織化を阻害することがある。まずは、チームを尊重すること。
◆経営者やプロダクトオーナーとの関係が大切になるため、ファシリテータ（スクラムマスター）の任命も検討すること。
◆メンバーの流動性が高い場合、その文化の維持と伝達に気を付ける必要がある。
◆様々な組織から人が集められたチームの場合は、チームとしてではなく所属組織としての振る舞いを選択することがある。

効果

◆自ら判断しながら自律的にチームで仕事ができるようになる。
◆PM の管理をするための工数が減り、常に最新の状況が可視化され透明性が増す。

利用例

自己組織化チームは、全体の 80%の事例に適用されていた。規模別の切り口では、小規模では 88%であったのに対して、中大規模では 58%と低下している。「自己組織化が難しかった」という事例もあり、多くの人員を必要とする場合は、自己組織化を促進するための策が必要である。

C 社事例(4)では、60 名（うち、社員 20 名、パートナー40 名）のプロジェクトにおいては、元請会社とパートナー会社の間に契約関係があり、自己組織化よりも契約のほうが優先されたため、自己組織化するのが難しかった。

D 社事例(5)では、最初は社内のアジャイルコーチがファシリテータ（スクラムマスター）を兼任していたが、一定期間を過ぎた後は、開発メンバーに引き継ぎ、チームが自己組織的に動けるように促した。

E 社事例(6)では、チームメンバーそれぞれがタスクを実施することに貪欲だったため、最終的にはとても自律したチームになった。

J 社事例(17)では、タスクを実施する目的をメンバーに常に意識させ、作業順番もメンバーが考えられる状況まで情報共有を行い、自己組織化されたチームになった。

L 社事例(21)では、日次ミーティングで共有した課題に対し、タスクフォース的な対策チームを組むなど、自発的に得意不得意を補い合っていた。

L 社事例(22)では、規模が大きく、複数のチームで開発を行っていたが、プラクティスの実施方法を細部まで規定せず、各チームにプラクティスの運用を任せていた。

M 社事例(25)では、チームが専門性の高いメンバーで構成されており、上からの指示だけでなく、メンバーが主体的に動いている。

関連プラクティス

インセプションデッキ
ふりかえり
ファシリテータ（スクラムマスター）
チーム全体が一つに

3.3.7. ニコニコカレンダー / Niko-niko Calendar (Smiley Calendar)

“チームのムードやメンバーの気持ちを見える化する”

別名: なし

要約

カレンダーに毎日その日の気持ちを表すシールを貼るようにする。その結果、チームのムードやメンバーの気持ちが見えるようになる。

状況

同じ場所で、チームで仕事をしているが、お互いの状態を伝えるような機会がない。

問題が表面化することなく、プロジェクトが滞りなく進んでいるように見える。

問題

メンバーに起きている異常に気付くことができない。

メンバーが正常ではない状態に陥っていたとしても、互いの状態を伝える機会がなければそれを把握し、対処することはできない。

問題が大きくなる前に、異常を察知し、メンバーへのケアをチームで行いたい。

フォース

◆人によっては、性格的に自分の気持ちを積極的に他人に伝えることができない。

解決策

1 日の仕事が終わって帰宅する際、カレンダーにその日の気持ちを表すシールをチームメンバー全員に貼ってもらう。

シールは、丸い三色のものを用意する。それぞれの色に対して、良い感じ、普通、嫌な感じなどの意味を決めておく。何色にどの感情を意味付けるかはチームで決めれば良い。

シールにはサインペンなどで、その感情を表す表情を書いておくとわかりやすい。

その日をふりかえって気持ちを表すために、シールを貼るタイミングは帰り際が良い。

カレンダーは、A3 や A4 の紙を使って、いつ、どのメンバーがどんな感情だったかわかるように、表形式にすることが多い。縦軸をメンバー、横軸を日付とする。

カレンダーは、壁などに貼ってだれもが見えるようにしておく。

シールを選ぶ、一言コメントを加える、などのチーム独自の一工夫を付け加えることで、より楽しく運用できる。

留意点

◆ニコニコカレンダーを定期的に見ること。また状態に対する対処を行わなければ、カレンダーはただシールを貼るだけで形骸化してしまい、やがてだれも使わなくなってしまう。

効果

◆「自分の気持ちを表現してよい」という場を作り、チームの絆を深める
◆気付きにくいメンバーの状態を、チームお互いで把握できるようになる。
◆メンバーの状態に対して、適切なタイミングで対処を行うことができる。

利用例

ニコニコカレンダーは、全体の 17%の事例で適用されていた。B2C よりも社内システム、小規模よりも中大規模、自社開発よりも受託開発の事例で多く適用されていた。チームの関係性をより深めたい場合に取り組むとよい。

E 社事例(6)の場合は、Redmine のプラグインでニコニコカレンダーを運用していた。ここからチーム内のコミュニケーションが生まれることが多々起きていた。

K 社事例(20)では、ニコニコカレンダーに頼らずとも、自分の状態を他人に伝えられるチームになっていたため、必要がなかった。

L 社事例(22)では、自由に気持ちを出してもらうため、だれが何を貼ったか追求しないようにした。

関連プラクティス

チーム全体が一つに

参考文献

坂田晶紀 (2005) 「ニコニコカレンダー」
http://www.geocities.jp/nikonikocalendar/index_ja.html

Agile Alliance (n.p) Niko-niko calendar
http://guide.agilealliance.org/guide/nikoniko.html

3.3.8. **持続可能なペース** / Set a sustainable pace

“健康で活気のある職場で、効果のある仕事を！”

別名：ゆとり、活気のある仕事

要約

高い効果や生産性を維持できるよう持続可能なペースを保つ。その結果、健康的で活気のある職場になる。

状況

プロジェクトや日常業務など様々な仕事に取り組んでいる。

長時間労働が暗黙的であるにせよ、強制され、高いプレッシャーを与えられる職場もある。その高いプレッシャーが、メンバーの疲弊を招く場合もある。
様々な要因から強度の標準化や人員削減が進み、それによって効率化がされている。効率化が進みすぎると、ゆとりがなくなる。

問題

無理をしてでも、高い効果や生産性を維持したい。

仕事の効率化を行った結果、様々な弊害をもたらし、変化への対応が困難になった。
不適切な業務時間や拘束時間によって疲弊してしまう。

フォース

- ◆成果を出したいが、燃え尽き疲れてしまっている。
- ◆仕事の効率化を行いたいが、効率化したことが様々な弊害をもたらし、変化への対応が困難になった。

解決策

持続可能なペースを守る。病気の時は休息を取る。健康の維持が第一である。

タイムカードや勤務管理システム、ニコニコカレンダーなどの機会を用いて、業務時間が長過ぎていないか、ストレスがたまっていないかを調査する。
規定時間以上に仕事や残業をするメンバーについては、本当に必要なものなのかどうかを確認し、また自ら行動することで持続可能なペースを維持しよう。ふりかえりを通じて、適切な状況かを問いかけよう。
自己組織化チームを目指し、仕事を分配する平準化などチームでできることをしよう。重要ではないタスクをあらかじめ計画に入れることによって、バッファにする。
恒常的に長時間労働や高いストレス状態が続いている場合は、プロダクトオーナーと共にタスクの優先順位付けの強化をしよう。
さらに、状況が困難であれば、人事関連の相談窓口などの適切な部署に事実を伝えるようにしよう。増員やチーム分割、その他の対応ができるかもしれない。
ただし、リリース直前や事象発生時のような時は、集中して行うなどの柔軟さも必要である。

留意点

プロダクトオーナーや顧客など、関係者からの要請との折り合いを付ける必要がある。しかし、危機的な状況であれば、あくまで短期間においては長時間労働などを行うこともありうるが、だらだら長時間拘束することは回避する。
プロダクトオーナーも多忙であることが多いため、プロダクトオーナーのチーム化などの対応が必要な場合もある。
プロダクトオーナーや顧客などとの信頼関係の構築および維持が、このプラクティスを成功に導くキーポイントとなる。そのための成果を事前に示そう。

効果

- ◆メリハリのある仕事のリズムを刻めるようになる。
- ◆一気に作りあげて、その後が続かないのではなく、継続的に安定的な効果を提供することができる。

利用例

持続可能なペースは、全体の 88%の事例で適用されており、切り口別でも大きな違いは見られなかった。

A 社事例(1)では、持続可能なペースは気にしているが、締め切りやマイルストーンに間に合わせる必要がある時には土日も作業する。自社プロダクトであるため、やらされ感はなく、メンバーのモチベーションは高い。

B 社事例(2) (3)では、週 40 時間を厳密に守っている訳ではないが、残業が増えると、ふりかえりの議題に上げる。プロダクトオーナーから無理強いされたり、プレッシャーをかけられたりすることはない。そのような関係性を作るために、対面の機会を増やすなどの取り組みをしている。

C 社事例(4)では、リリース直前の２ヶ月は忙しく、持続可能なペースにはなっていなかった。それ以外の期間は、持続可能なペースを意識していた。

D 社事例(5)では、風邪や体調不良の時、出社禁止を権限のある立場の人から指示し徹底したところ、本プラクティスが徹底して実行されるようになった。

G 社事例(10)では、チームの実力が顧客の期待に達していないことを身を持って感じたため、顧客の期待に応えようと、持続可能ではないペースの残業をして対応していた。

J 社事例(16)では、１日６時間程度の稼働を想定した計画を立てている。

関連プラクティス

ニコニコカレンダー
ふりかえり

3.3.9. 組織にあわせたアジャイルスタイル / Right agile style for their organization

"状況や目的に合ったアジャイルスタイルにしよう"

別名: なし

要約

その時のプロセスが最善とは限らない。組織に合わせた開発スタイルを模索する。

状況

アジャイル型開発を導入したい環境および組織の状況はそれぞれ異なる。

開発には、様々な業務領域（ドメイン）やチーム人員などが存在する。同じドメインであったとしても、現場の状況によっても異なる。

同じコードベースとサービスであっても、状況によって求められる行動が変わる。例えば、リリース前は、イテレーション単位でのフィードバックで充分だったものが、リリース後であれば、利用者からの指摘や社会的な事象などへの対応がリアルタイムで必要となる。

問題

アジャイル型開発をしているが、しっくりこない。組織の状況と目的に対して適切な状態になっていない。

例えば以下のような問題だ。

a) 時間より品質を優先する企画や調査段階において、イテレーション計画ミーティングの時点では妥当な計画が困難になる場合がある。問い合わせや指摘事項を受け付け日々の変更要求が発生する運用保守段階や、販売状況等ビジネス状況に応じて日々のタスクが決まるなど不確定要素が多い場合がある。その結果、イテレーション計画ゲームで綿密にタスク計画を建てても、その計画どおりにいかず、意味のない計画になってしまう。
b) 発注元の顧客や、一緒に仕事をしているパートナーがウォーターフォール型開発を前提にしているため、アジャイル型開発との親和性が低く、場合によっては拒絶される。もしくは、初めてアジャイル型開発を行うため顧客やチームメンバーには拒絶反応を示す。
c) スクラムで、プロダクトオーナーがボトルネックになる。プロダクトオーナーと開発チームが対立する価値観を持っている状況で、ビジネス企画のオリジナリティが損なわれる。
d) 全体の規模が大きく、要件定義やテスト、アーキテクチャ設計などを、開発サイクルのイテレーションを走らせながら実施することは困難である。

フォース

◆特定の手法を用いたいが、そのままでは自分の状況にはマッチしない。
◆どの状況でも最適なソリューションは存在しない。

解決策

組織の状況に合わせてスタイルを柔軟に変える。正解はないため、自分たちにあったスタイルを常に改善し続ける。

全体の解決策：組織が持つ制約事項によって、スタイルが変わってくる。分散した拠点で開発する時は、コミュニケーションが疎になりやすいので、顧客プロキシなどのプラクティスを用いて、知識の連携を強める。タスクの見積りが難しい研究段階の場合は、イテレーションを繰り返しながら見積りの確度を上げていく。
このように組織の状況にあったプラクティスやプロセスを選択して利用すること。

全社横断など複数のチームで統一したスタイルにしたい場合は、パイロットプロジェクトを用いる。経験を通じて課題を明確にし、パイロット的なアジャイルスタイルを記述し、そのスタイルを経験したメンバーや、複数チームを見るファシリテータ（スクラムマスター）を配置するなど横展開する手法がある。

a) 企画・調査段階や運用保守段階は、かんばんなどのプラクティスを用いる。
b) アジャイル型開発であることを隠蔽し、ウォーターフォール型開発やプロジェクト管理手法で使われる WBS などに変換する。
c) プロダクトオーナーを増員し、複数人のプロダクトオーナーを構成する。もしくは、プロダクトオーナーの役割をなくし、開発チームと一体になってその責務を行う。
d) 要件定義とテストをウォーターフォール型開発とし、具体的な開発をアジャイル型開発

とする。

そもそも、アジャイル型開発を何のために適用するのか、今の状況に対して適切かどうかも見極める。

留意点

◆同じ組織での別チームであっても、人材や状況が違う。主に他のチームとのインターフェイスになるところや、コアのスタイルのみを定義することが望ましい。多様性を認め、できる限り自己組織化されたチームを目指すようにする。
◆チームの成熟度合いによってカスタマイズの方針が変わる。例えば、アジャイル型開発の初心者であれば、カスタマイズせずに一度は文献に記載されているプラクティスをそのまま実施するのがお薦めである。
◆現状を受けいれて適応したやり方を実施するのと、現状にある課題を見ないふりをするのは異なる。まずできるところから着手していき、徐々に改善する領域を増やしていくとよい。最初はチーム内からはじめ、徐々にその影響範囲を広げていくとよい。

効果

◆環境や目的に応じたアジャイル型開発のスタイルに変容していくことができる。

利用例

組織に合わせたアジャイルスタイルは、全体の79%の事例で適用されていた。特にシステム種別の切り口で、社内システムでは56%であったのに対して、B2C サービスでは86%であった。

A 社事例(0)では、会社の方針を経営層やビジネスの担当者が一方的に決めず、「〜についてどう思う？」と開発者に問いかける文化になっている。開発者も、ビジネスやユーザーの観点を持ち、総合的に話しあって決めていく。開発がビジネスに対してフィードバックするよう組織に合わせた。

C 社事例(4)と F 社事例(8)では、中大規模開発で、かつウォーターフォール型開発との組み合わせに適応するアジャイルスタイルにした。例えば前半にアーキテクチャを構築し、最後はテストとその修正を行うようにした。ウォーターフォール型開発を行っているチームとは、インターフェイスを決定した上で、疎結合に開発を進めた。

E 社事例(6)では、アジャイルの経験はなかったため、要件定義とテストはウォーターフォール型開発で行い、設計開発は反復的に実施した。
E 社事例(7)では、業務要件定義を行った後に、画面設計〜結合テストまでを反復的に実施した（ウォータースクラムフォール）。

D 社事例(5)では、スクラムを採用して開発をしていたが、XP のプラクティスが有用なことに気づいて XP の面を強化している。

F 社事例(8)では、イテレーション計画ミーティングで計画することが困難なほどタスクの内容が変わってしまうため、プロセスを固定化した「かんばん」として捉え、制約理論(TOC)のバッファコントロールを取り入れている。

H 社事例(11)では、組織に合わせることが必ずしも解決にはならないため、習熟するまでプラクティスを変更しないようにした。

H 社事例(12)では、組織に合わせるために、スクラムでは許されていないスプリント中のバックログの追加変更などをカスタマイズした。

H 社事例(14)では、オフショアに対応してオンラインミーティングや画面共有を行うなど組織の状態に合わせた。

L 社事例(21)では、複数あるチームに対して一律でアジャイルスタイルを決めておらず、各チームの特性に合わせて必要なプラクティスを適用した。

L 社事例(21)と事例(23)では、顧客とパートナーはウォーターフォール型開発であり、それぞれのチームと連携する必要があるため、作業計画や状況を WBS に変換して顧客とパートナーに渡し、ウォーターフォール型開発で進めているように見せた。

M 社事例(25)では、開発プロセスの異なる複数の組織を横断したプロジェクトにおいては、「アジャイル型開発を採用」と全面に押し出すのではなく、会議体や責任範囲の明確化を行うためにアジャイルのプラクティスを採用していた。

関連プラクティス

柔軟なプロセス
ふりかえり

3.3.10. **共通の部屋** / Give the team a dedicated open work space

“全感覚を使ってコミュニケーションを円滑に”

別名：全員同室

要約

意思疎通がうまくいかず、活気がない場合は、共通の部屋を作る。その結果、コミュニケーションは改善する。

状況

チームやステークホルダーが協調しながら開発をしたいと考えている。チームメンバーがそれぞれ、パーティションや、机の並びや、部屋で区切られている。
地理的に分散して開発を行っている。

問題

チームや関係者のコミュニケーションがうまくいっていない。活気がない。

プロダクトオーナーと開発チームのコミュニケーションに問題が発生する。ちょっとした質問をしたい時に、作業場所が遠隔地であったり、パーティションで区切られていたりすると回答がすぐに得られない可能性が高い。

コミュニケーションは、文字以外でも多くの情報を伝える。例えば、「あ」と一言に言っても、「あー（納得）」や「あ？（怒り）」、「あ（れ）？（疑問）」など様々な意味がある。また、顔の表情や体勢なども様々な情報を持っている。しかしながら、同じ部屋にいないとそれらを伝えることが難しい。

また情報を共有する際にも、すぐに目に飛び込む掲示やポスターを使うことができない。

フォース

◆コミュニケーションを活性化するために、オンラインチャット・システムや会議システムを導入したが、必要なコミュニケーションの多くを伝えることはできない。
◆コミュニケーションツールでは、 ちょっと声を掛けたり、様子を伺ったり、相手の繁忙を見て声を掛けることができない。

解決策

充分な広さの場所を用意し、チームメンバーやステークホルダーが仕事をする。

充分に広いオープンスペースや、専用のプロジェクトルームを用意し、そこで開発チームやプロダクトオーナー、デザイナなどの関係する人が作業を行う。

部屋の周囲の壁には、様々な情報を貼り出して共有しておくと大変有用である。

ミーティングなども、スペースを作っておけば、わざわざ会議室を予約するまでもない。

共通の部屋によって、チーム全体が一つになることが促進される。またオンサイト顧客が実現できると尚よい。

壁には紙・手書きツールを使った、かんばんやタスクボード(タスクカード)、バーンダウンチャート、インセプションデッキを貼っておいて、常にチーム全員が気に掛けるようにすることができる。

留意点

◆同じプロダクトやサービスに関係する人を近い席にするなどの配慮が必要になる。
◆コミュニケーションが活発になる反面、問い合わせなどのコミュニケーションが多すぎて、集中して作業ができなくなる可能性があるため、お試しで実施し、効果を計測すること。
◆物理的に不可能な時は、オンラインチャット・システムや遠隔テレビ会議システムを使用することは、完全ではないにしろ、多少の改善は望める。日次ミーティングの時だけでなく常時接続することによって、簡単な問い合わせなども円滑になりやすい。
◆プロジェクトルームを作る場合、チーム以外の人が出入りしにくい閉鎖的な状況にならないように注意する。

効果

◆チームに活気が出て、コミュニケーションが円滑になる。
◆お互いの信頼関係を醸成する土壌となる。

利用例

共通の部屋は、全体の 69%の事例で適用されていた。分散拠点以外の事例では、積極的に検討すべきプラクティスである。

A 社事例(1)では、オープンスペースがなく、プロジェクトルームが作れなかったが、関係者は近くの席にした。別チームの人も近いので、相談しやすい利点もあった。

C 社事例(4)では、会議室をプロジェクトルームにした。外部からの雑音を排除できた。

D 社事例(5)では、背中合わせで座れるようにし、円滑なコミュニケーションと、集中した作業の両立を実現した。

G 社事例(9)では、パーティションで仕切られた専用の開発室を設置した。壁を使える部屋が欲しかったので、プロジェクトルームを作った。その結果、雑音を排除し、チームが集中でき、「全体を一つに」のプラクティスをサポートできた。

壁に付箋を貼ってすぐに情報共有できる環境を作ることができた。
ただし、負の面として、社内への波及効果を考えると、閉ざされすぎていて、関係者以外は入りづらかったという意見があった。社内にアジャイル型開発の事例として見てもらいたかったが、それができにくい状況を作ってしまった。

関連プラクティス

チーム全体が一つに
オンサイト顧客
紙・手書きツール
かんばん
タスクボード (タスクカード)
バーンダウンチャート
インセプションデッキ

3.3.11. **チーム全体が一つに**/ One whole team

“合い言葉は、「機雷がなんだ、全速前進！」”

別名：なし

要約

チーム全員が一つの目標に向かうような取り組みを行う。その結果、チームは素早く正確にコミュニケーションができ、生き生きとした仕事ができるようになる。

状況

なぜ、この仕事をしているのかチームメンバーそれぞれで把握している内容が異なる。あるいは、プロジェクトのゴールをだれも知らない。

チームの仕事場がばらばらで、全員同時にコミュニケーションを取ることができない。

チームの一体感がなく、共に一つの仕事に取り組んでいる感覚がない。

自分の与えられた仕事をこなすことに集中していて、チームとしての成果について考える機会がない。

問題

チームが一つになっていなければ、互いの作業、学習をサポートすることはなく、結果として成果が上がらない。

一つの仕事をチームではなく、個々人が別々に取り組んでおり、相互のコミュニケーションがなく、プロダクトの理解に誤りがあっても修正されることがない。また、個々の仕事の間で整合性が取れない。

互いの作業をサポートする雰囲気がチームになければ、それぞれが自分のことをのみ考え、結果として、プロダクトの完成が遅れたり、品質が低いものになったりする可能性がある。

フォース

◆チームメンバーから仕事のゴールが見えにくい場合、チームが自然と一つになることは難しい。
◆役割が明確に分かれすぎていると、役割毎に意識が分断されてしまいがちである。

解決策

仕事の目的や目標を明確にする。

われわれはなぜここにいるのか？を明確にする。チームに期待されていることとは何かをメンバーそれぞれが把握する。

遠くにある最終的な目標とは別に、直近のイテレーションにおける目標もチームで共有する。当該イテレーションで求められる振る舞いを取れるようにする。

メンバーがチームの一員であるという感覚を持つために、チーム全体への期待とは別に、目標に対するメンバーそれぞれのミッションを明確にする。

インセプションデッキや、プロダクトバックログを全員が深く理解することで、チーム化が促進される。

仕事場に全員同席する。

チーム全体が入れるだけの充分な広さの仕事場を確保し、共に仕事に取り組む。

コミュニケーションがすぐに取れるように、メンバーが座る場所を一箇所に集める。振り向いたら、チームメンバーがいるような状況を作る。

ペアで作業を行うようにする。例えばペアプログラミングなどである。

チーム内の役割で偏った座り方をするのではなく、プログラマの側にビジネスアナリストが座るなど、多様な組み合わせのコミュニケーションが生まれるようにする。

チームが役割を越えて一つになることで、自己組織化チームに進化していく。

留意点

◆コミュニケーションが活発になることで、雑音も増え、気が散るなどの影響が起きる場合もある。

効果

◆チームメンバーそれぞれが仕事の目標を捉えることで、ゴールに向かった振る舞いを取ることができる。
◆行き詰ったら、他のメンバーに助けを求められるようになる。
◆コミュニケーションが効果的になる。推

測するのを止めて、お互いに質問や相談をするようになる。アイデアが伝達されやすくなる。
◆チームに一体感と互いへの敬意が生まれる。仕事が楽しくなる。

利用例

チーム全体が一つには、全体の 86%の事例で適用されていた。特にチームが断片化されがちな中大規模事例で 100%適用されていたのが興味深い。チームがバラバラになりそうな状況では積極的に適用すべきである。

C 社事例(4)の場合は、合宿を開き、チームのクレドを作ったが、合意形成は困難であった。

G 社事例(9)では、プロジェクトの開始時にチームビルディングのためのワークショップを行った。プロジェクト開始時やイテレーション毎で、スローガンを掲げて取り組むようにし、掲げたスローガンを壁に貼っていつでもだれでも見えるようにしていた。

H 社事例(12)では、インセプションデッキを貼り出し、今どこに向かっているのか常に見える工夫を取った。

J 社事例(18)では、タスクを実施する目的をチームメンバー全員で常に意識し、作業の段取りもチームメンバー自身が考えられる状況になるまで、情報共有を行っていた。

J 社事例(19)では、イテレーションの目標をホワイトボードに書き、チーム全員が認識できるようにした。

関連プラクティス

インセプションデッキ
プロダクトバックログ
ペアプログラミング
自己組織化チーム

3.3.12. **人材のローテーション** / Move people around

“どんな仕事でもこなせるようになろう”

別名：多能工、

要約

属人化しているならば様々な役割が経験できるようにする。その結果、変化やインシデントに強い組織になる。

状況

チームや組織、ステークホルダーには、様々な強みを持った人がいる。データベース設計、オブジェクト指向プログラミング、アーキテクチャ、業務ドメイン知識などである。

チームメンバーには、プロダクトやサービスの知識や経験に偏りがある。メンバーによっては、あるサブシステムや特定の機能については詳しいが、その他については知らないということがある。

チームは、数ヶ月に渡る比較的長期的な仕事をしている。

規模が大きなプロジェクトで複数のチームで構成されている。

問題

チームや組織内で専門知識や業務知識などの偏りが発生し、属人化している。

属人化している領域が単一障害点になりやすい。例えば、ある専門知識がある人の退職や休暇によって、その担当分野の仕事が止まってしまう。

特定の人しか知らない領域のタスクについては、他の人をアサインできない。

フォース

◆属人性を排除したいが、ある特定の人に知識や経験が集中してしまう。
◆与えらえた同じような仕事をこなしていると、その仕事の専門化になりがちである。
◆いきなり経験のない新しい仕事に取り組んでもすぐにできるようにはならない。

解決策

仕事の役割を変更し、どんなタスクもこなせるようになる。

チームや組織には、様々な強みを持った人がいる。その個人が持っている強みをチームのものとする。

業務知識や専門知識などの対象は最初から限定せず、ふりかえりなどで発見された領域をローテーションの候補となる。効果が高く現実的なところなど優先順位を高いところから実施する。

教師役もしくは前任者が全体像を伝えるようにする。全体像の中での実際のタスクの位置を理解することにより、作業のズレが少なくなる。

さらに、ペアプログラミングのようなペア作業や、対象のレビューを実施する。その際、その具体的な作業や行動の背景にある理由（「なぜ」）や、条件（「いつ」「なにを」「どのように」）などを伝える。

ローテーション後は、前任者をレビュアやペアプログラミングのナビゲーターとして指名することにより、ヌケやモレなどを最小限に押さえるようにする。

留意点

◆プロジェクトや業務で、比較的余裕がある人や時期を選ぶとよい。
◆ローテーションを行う時期を事前に決めておく。定期的に行っても良い。
◆業務領域や専門領域、また前提となる古典的な知識、社会的な潮流など、様々な観点における知識や経験がポイントとなるため、日常的に学び、習うような環境を構築するとよい。
◆このプラクティスは、人的な影響が大きいため、メンバーの相性や特性、実施時期や方法を充分に吟味してから行うこと。
◆このプラクティスは、比較的長期的に継続する組織やプロジェクトに向いている。
◆短期的には工数（コスト）が増えている。

効果

◆ファシリテータやスクラムマスターなどの役割についても、ローテーションすることによって、その役割についての共通理解が広がる。
◆属人性が減少し、変化やインシデントに

対する耐性が強い組織になる。
◆共通理解が広がると、ふりかえりやレビューなどで、より成熟した解決策を選択し、実施することができる。

利用例

人材のローテーションは、全体の 42%の事例で適用されていた。B2C サービス(38%)よりも社内システム(63%)で、小規模(28%)よりも、中大規模(92%)で、より適用されていた。長期視点で考えるべきシステムの場合は積極的に適用したい。

B 社事例(2)では、半年でメンバーを入れ替えている。メンバーを変えた結果、チームの雰囲気が大きく変わる。

C 社事例(4)では、複数の開発チームに別れていたが、それぞれのチーム間での知識の共有が必要であるとメンバーが認識し、自主的にローテーションを提案して定期的にメンバーの入れ替えを実施した。

D 社事例(5)および E 社事例(7)では、全員がファシリテータ（スクラムマスター）になるようにした。特に向かない人については、長めにその役割をアサインし、周りもサポートした。その結果、全員がファシリテートできるようになり、特定のファシリテータ（スクラムマスター）が不要になった。

E 社事例(7)では、チーム内のローテーションは行うが、チーム外からの人材ローテーションは行わないようにし、サービスやプロダクトについての知識や経験がある状態を保つようにした。

G 社事例(9)では、経験がない対象をあえて選択し、ローテーションした。技術的背景が違う人が揃ったので、教育的なペアプログラミングを実施し、知識の共有を促進する取り組みをしていた。

H 社事例(12)では、命令や指示としてローテーションするのではなく、メンバーから志願する形で実施した。

L 社事例(22)では、目先の効率よりも、チーム力の向上を目指すことを始めに宣言しローテーションすることによって、一時的に効率が落ちることをチームメンバーやプロダクトオーナーなどに納得してもらいやすくなった。

M 社事例(25)では、チーム内で人材のローテーションを実施はしているものの、技術的な傾向（得意/不得意）もあり難しいことも多い。

関連プラクティス

ペアプログラミング

3.3.13. インテグレーション専用マシン/ Set up a dedicated integration computer

“統合マシンは、アジャイル開発を加速する”

別名

要約

継続的インテグレーションを実施するためには、本番環境に近いインテグレーション専用のマシンや環境を用意する。その結果、インテグレーションを頻繁に実施でき、本番環境との差異を最小限に抑えることができる。

状況

アジャイル型開発においては、顧客やプロダクトオーナーなどからのフィードバックを得て、方向性を修正しながら開発を進める。イテレーション期間の間に、ソースコードを逐次統合し、頻繁にビルドとテストを繰り返す必要があり、継続的インテグレーションが望まれている。

ビジネス特性や開発特性によっては、デプロイメントまで自動化し、継続的デリバリ（継続的インテグレーションだけでなく、ソフトウェアの本番環境へのデプロイメントまでも自動に行われる仕組み）が行われる。

インテグレーションのフィードバックサイクルを頻繁にしたいため、高速なインテグレーション環境が欲しい。

問題

インテグレーション環境が、共用だったり、遅かったり、本番環境と異なっていては、インテグレーションの価値が下がってしまう。

例えばサーバーを別プロジェクトと共用していて、その上でイテグレーションを実施しようとしても、実際の本番環境と異なっていたり、速度が望めないことが多い。

インテグレーションに時間がかかると、その分問題の検知が遅れることになる。

フォース

◆開発が進むにつれ、ソースコードやテストケースが増える。
◆ソースコードやテストケースが増加すると、それを実行するためにコンピュータのリソースを消費し、時間がかかるようになる。

解決策

専用の統合マシンを用意する。統合マシンでは、できるだけ本番環境に近い形で、ビルドやテスト、動作を高速に確認できるようにする。

開発現場の状況や方針によって、開発環境は異なるが、インテグレーション専用マシンと共にソースコードを保持するリポジトリ環境、プロダクトオーナーや顧客にデモを見せるデモ環境、ビジネス企画を試すカナリア環境、本番に類似したリリース候補（RC）環境、実際の顧客やエンドユーザが操作する本番環境などを用意する。

実際のエンドユーザや顧客が操作する環境は、本番環境や商用環境と呼ばれる。本番環境では、顧客が満足するような応答速度などのパフォーマンスや、個人情報を含む機密情報保護などセキュリティの非機能要件を満たしている。

開発およびデリバリにおいて、オペレーティングシステムや、データベースやミドルウェアなどの種類やバージョン、設定が異なることで問題や課題が発生することが多い。そのためインテグレーション専用マシンでは、最新ソースコードを用い、環境をできる限り本番環境と同じにすることによって、本番環境で発生しうる問題を早期に発見し対応できるようにする。複数の動作環境がある場合はそれぞれの環境を構築する。

クラウド上に仮想環境を構築するならば、メモリなどのリソースをできるだけ多く割当てて高速化すること。物理環境を構築する場合は、できるだけスペックを高くしておくことでインテグレーションの速度を高めフィードバックサイクルを頻繁にすることができる。

継続的インテグレーションを行うと、毎日、もしくは、ソースコードが最新になった時に逐次、ビルドとそれに関連するテストが実行されるようにする。継続的デリバリの場合は、必要な確認や承認作業の後、デプロイメントまで自動に行う。

インテグレーション専用マシンが用意できていれば、逐次の統合を検討する。

効果

本番環境に近い環境で、高速にインテグレーションを実現することができる。

インテグレーション専用マシンを社内の利用者に公開することにより、社内での開発の透明性が上がる。

継続的インテグレーションを実施する環境が整備できる。

留意点

ネットワーク・セグメントや物理的な配置は、その開発環境が属するポリシーや状況に依存する。一般的に、本番環境と、インテグレーション専用マシンは、別のネットワーク・セグメントや物理的な配置にすることが望ましい。

利用例

インテグレーション専用マシンは、全体の83%の事例で適用されていた。特に社内システムが 56%だったのに対し、B2C サービスでは 94%の適用率であった。継続的インテグレーションを検討する際には一緒に適用したい。

B 社事例(1)では、関連会社のクラウドサービスを用いて、Jenkins 実行環境を設定している。

C 社事例(4)では、継続的インテグレーション環境だけでなく、デモ、ステージング、本番環境を用意している。

G 社事例(9)と(10)では、クラウドサービスを用いて、本番デプロイされる前に試すことができるステージングサーバを用意している。

F 社事例(8)では、インテグレーション環境以外にもカナリア環境と呼ぶ実行環境を用意している。希望ユーザーに機能を本実装する前のプロトタイプを利用してもらい、その機能についての要望を収集しフィードバックを得ることができる。

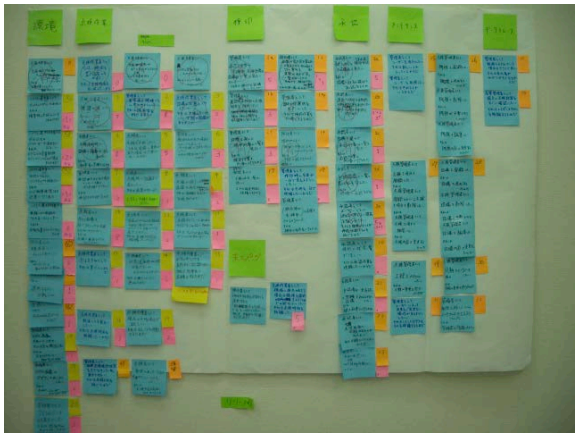
関連プラクティス

継続的インテグレーション
逐次の統合

3.4. **発見されたプラクティス**

事例調査の中で、各事例の現場毎に様々な工夫をこらしていた。
その中でも 3 つ以上の事例で見受けられた工夫の一部を、調査対象ののプラクティスとは別に「発見されたプラクティス」として紹介する。

3.4.1. ユーザーストーリーマッピング / User Story Mapping



“プロダクトの地図を作って航海しよう”

別名: なし

要約

プロダクトの全体像が見えない、何が重要なのかがわからない場合は、ユーザーストーリーを利用者の時間軸と、優先順位の 2 つの軸でマッピングしよう。その結果プロダクトの全体を俯瞰でき、計画づくりに役立つ。

状況

プロダクトオーナーは、どのようなプロダクトを作りたいかのイメージを持っており、開発者に伝えるための様々な資料も作りあげている。

これからユーザーストーリーを用いて、プロダクトバックログを作っていきたいと考えている。

開発者たちは、まだプロダクトの全体像や、具体的なイメージが把握できていない。

問題

ユーザーストーリーやプロダクトバックログだけでは、プロダクトの全体像は把握しづらい。全体像を把握した上でスコープを決めないとプロダクトの価値提供がうまくいかない。

ユーザーストーリーは顧客価値を与える機能を表現する。プロダクトバックログは、ユーザーストーリーも含めた、プロダクトに必要な要件を優先順位付けされたリストとして格納している。

しかし、ユーザーストーリーが複数あっても、それらがプロダクトの全体のどこに価値を与えるのか、全体のどの辺りを今のスプリントの対象にしているのかが理解しづらい。

開発者として、プロダクトバックログのリストを見せられたとしても、それらが最終的にどのようなプロダクトになるかを想像するのは難しい。

フォース

- ◆プロダクトバックログ以外の資料でプロダクトの全体像を表現することは可能だが、プロダクトバックログに細分化した後に全体像とマッピングすることは難しい。
- ◆プロダクトオーナーがいくら資料を作りプロダクトについて説明しても、開発者は内容を噛み砕いて理解するのに時間がかる。

解決策

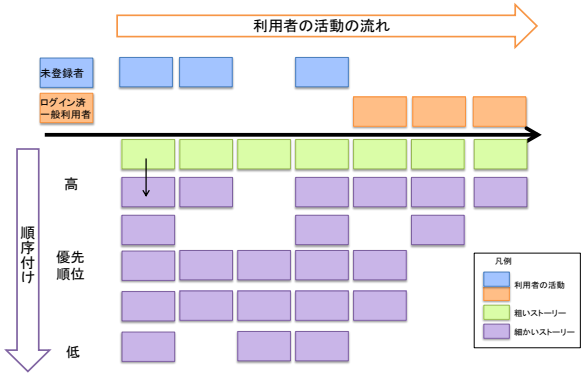
ユーザーストーリーを洗い出す時に、チーム全員で、利用者の時間軸を横軸に、優先順序軸を縦軸に置き、ユーザーストーリーを配置していく。

ユーザーストーリーマッピングは Jeff Patton 氏によって考案されたユーザーストーリーを用いた全体像を把握するテクニックである。最もシンプルな手順は次の通りになる。

(1) プロダクトの利用者が行う活動を洗い出し、時間軸で左→右に並べていく。
(2) 活動に対応するユーザーストーリーを書き出し、活動の下に配置する。
(3) 一つの活動に対して複数のユーザーストーリーが存在する場合は、優先順位で上から下に、大事なものから並び変えていく。

出来上がった、時間と優先順位の 2 軸のマップをユーザーストーリーマップ（図 4）と呼び、マップを作りあげていく作業をユーザーストーリーマッピングと呼ぶ。

マッピングは、プロダクトオーナーと開発者全員で実施することが望ましい。なぜなら、ユーザーストーリーを書いていきながらプロダク



- 時間と優先順序の二軸で構成される

図 4 ユーザーストーリーマップ

トの全体像を作りあげていくことで、プロダクトに対しての理解が深まるためである。

またマッピングは、紙・手書きツールを使って実施するのが望ましい。壁一面に付箋を使ってチームでユーザーストーリーを書き出し、マッピングし、共有していくことで、常に俯瞰できる全体と、着目している部分の両方を意識することができるからだ。

マッピングを行うタイミングとしては、まずは開発が開始される前に行うが、一度マップを作ったら更新しないのではなく、ユーザーストーリーの内容は刻々と変わわわっていくため、変化に追従しながら更新していく必要がある。

ユーザーストーリーマップを元に、リリースに含めるスコープ調整や優先順位づけなどを行なってリリース計画ミーティング、イテレーション計画ミーティングにつなげることができる。

ユーザーストーリーマップ上で順序が決まれば、プロダクトバックログ上での一元的な順序が決まる。

ユーザーストーリーマッピングによってプロダクトの全体像を共有することで、チーム全体が一つになるのを促進させる。

留意点

ユーザーストーリーマップは巨大なマップになってしまう場合が多い。貼り出しておく壁が存在しない場合は、模造紙などを用いて移動可能にしておく。

ユーザーストーリーマップを表計算ソフトなどで書き写すことは、格納スペースを節約するという意味では適しているが、全員が常に全体像を意識するという意味では逆効果である。できれば全体像をアナログデータのままで貼り出しておける工夫を考えたい。ただし、分散拠点間でマップを同時に共有したい場合は、アナログでの利用は困難のため、何らかでデジタルデータ化する必要があるだろう。

あらかじめプロダクトオーナーがユーザーストーリーマッピングを行ってマップを作り、それを開発と共有する方法もあるが、マッピング自体のプロセスも含めてチームで共体験することが望ましい。

プロダクトバックログにはユーザーストーリーマップの内容は全て含まれるが、プロダクトバックログの内容すべてがユーザーストーリーマップ上に含まれるわけではない。ゆえにプロダクトバックログは依然として必要である。

ユーザーストーリーマップは共通の部屋で利用することで、効果が飛躍的に高まる。

ユーザーストーリーマッピングの前段階として、インセプションデッキでプロダクトのビジョン、目的、エレベータピッチといった本質的な価値を共有しておくのが望ましい。

効果

- ◆関係者間でプロダクト全体像の理解を促進する
- ◆全体が把握できることで、最小スコープが決めやすくなる
- ◆プロダクトバックログの優先順位を付けやすくなる

利用例

B 社事例(2)では、ユーザーストーリーマッピングを進めていく中で、開発者が主体的にプロダクトについて考え、意見を出していっていた。

B社事例(3)では、プロダクトオーナーと共にチームがユーザーストーリーマッピングやリーンキャンバス[13]を使ってプロダクトの方向性を深く理解していた。そのため、プロダクトオー

[13] 「Running Lean -実践リーンスタートアップ」で提唱されるプロダクトのビジネスモデルの表現形式

ナーの代りに、新たなユーザーストーリーを開発者自身で書くことができていた。

G 社事例(9)では、アジャイルコーチの勧めでユーザーストーリーマッピングを使って、プロダクトの全体像を可視化した。全体像を俯瞰しながら、リリースやスプリント毎の優先順位、実現範囲などをプロダクトオーナーと開発者が共有しながらプロジェクトを進行していた。

G 社事例(10)では、事例(9)と同様の動機でユーザーストーリーマップを利用していた。プロダクトオーナーが別拠点にいたため、模造紙上にマップを作成して、携帯して持ち運べるように工夫していた。ユーザーストーリーマップによって、ユーザーの全体像がわかり、顧客との会話の原点とすることができていた。顧客と一緒に半日ぐらいの時間を費やしてマップを作成していた。

H 社事例(12)では、リリースバーンダウンチャート、プロダクトバックログ、などと共にユーザーストーリーマップによって、進捗や提供機能の全体像を可視化していた。スクラムやXPの中心的プラクティスに留まらずに新しいプラクティスにチャレンジしていた。

L 社事例(24)では、プロダクトが全体として何を実現するのかを意識するために、チーム全員でユーザーストーリーを書き出し、ユーザーストーリーマップ風に並べて共有していた。

関連プラクティス

リリース計画ミーティング
イテレーション計画ミーティング
ユーザーストーリー
プロダクトオーナー
プロダクトバックログ
共通の部屋
インセプションデッキ
チーム全体が一つに

参考文献

Jeff Patton, (2008). User Story Mapping, http://www.agileproductdesign.com/presentations/user_story_mapping/index.html

Takeshi Kakeda, (2010)「アジャイルな地図づくり」 http://www.slideshare.net/kkd/user-story-mapping-for-agile-team

[アウリャ 2012] アウリャ,A. 角征典訳 (2012)『Running Lean――実践リーンスタートアップ』 オライリージャパン

3.4.2. 「完了」の定義 / Definition of DONE

“チームの「完了」を明らかにする”

別名:完全 Done

状況

アジャイルチームは、イテレーションのゴールを明確にして、チーム一丸となってゴールの達成に向けて自己組織的に仕事を行う。

問題

チームの中で「完了」についての認識が揃っているだろうか?

ある開発者が、「ユーザーストーリーが完了しました」と報告した。ユーザーストーリーは何を持って完了したと言えるのだろうか? 「ユーザーストーリーの実装に必要なタスクがすべて終わる」ことが完了だろうか?「受入テストに成功をする」こと、または 「リファクタリング済みの上で、受入テストに成功をした」ことが完了なのか?

チームによっても、人によって完了の意味はそれぞれ違う。その違いが顕在化するのは、いつも作業の後である。フィードバックではなく「手戻り」作業が発生してしまう。

「完了」は様々な場面に偏在する。タスク、ユーザーストーリー、イテレーション、リリース、バグの改修、などである。すべての場面で「完了」についての認識が揃っていることは少ない。

プロダクトオーナーや顧客の期待する「完了」と、開発者の考える「完了」がズレていると、両者の信頼関係に悪影響が生じる。

フォース

◆人の「やって当たり前」は、他人の「当たり前」とは異なる。
◆「何をするか」には着目するが、「どうしたら終わりか」についてはあいまいになりがちである。

解決策

チームの中で作業の「完了」についての共通見解について合意して定義しておくこと。定義は開発者だけでなく、プロダクトオーナーや顧客との間で合意しておくこと。

「完了」の定義で最も大事なのは、プロダクトバックログの項目（ユーザーストーリーによって実現され動作するソフトウェアの増分)の「完了」の定義である。

チームは「完了する」ために何をしなければいけないかを列挙して、顧客、顧客プロキシ、プロダクトオーナー、をはじめとする関係者間で合意しておく。合意だけでなく、可能な限り「完了」の定義を全員が常に認識できるようにしておくのがよい。

「完了」の定義は、そのまま作業のチェックリストになり、すべてのチェックがついていないと「完了」にならない。

「完了」の定義は、チームで改善していき、チームの成長と共に、より多くの定義を一度の完了で達成できるようにしていく。
始めは、「ユーザーストーリーの受け入れ条件を満足している」段階が完了だとしても、チームの成長と共に、「該当ストーリーについて気になるところのリファクタリングが完了している」という定義が追加されていくかもしれない。

「完了」の定義を改善し、拡張していくことで、完了の中でより多くの意味を持たせるようにしていくことが可能である。

「完了」の定義は、ふりかえりの中で見直し改善していくことが必要だ。

「完了」の定義をすべて満たしているかどうかでベロシティ計測を行う。すべて満たしていないものはベロシティに含めてはならない。

効果

◆「完了」の定義を揃えておくことで、作業モレを防ぎ、確実にチームとしてのアウトプットを出すことができる。
◆開発者と事業側の間で完了についての齟齬を防ぎ透明性を実現することができる。

留意点

「完了」の定義は、開発者のスキルに依存する。スキルが低い開発者に、高いレベルの「完了」の定義を設定すると、なかなか「完了」に至らないことが予想される。

そのような場合は、「完了」の定義をチームの

パフォーマンスレベルに応じた定義にしておく必要があるかもしれない。

後で「完了」の定義を拡張した場合には、それまでの低いレベルの「完了」した作業を再確認する必要があるだろう。(テストの自動化が定義に含まれていないが、後に加わった場合は、それまでのテストの自動化がない部分にも対応しないといけない)

ユーザーストーリーを確実に完了させるためには、対象となるユーザーストーリーをより小さくしておく必要がある。大きなストーリーだと、「完了」の定義をすべて満たす前にイテレーションの終りが来てしまうかもしれない。そのため、小さく分けて着実に「完了」していくのが望ましい。

利用例

B 社事例(2)では、「完了」の定義をどう作ってよいか明確ではなかったため、完了を 4 つのステップに分け (プロダクトコードのコミット時、タスク、ユーザーストーリー、スプリント)それぞれに完了の条件を定義していった。また、この4つのステップをプレゼンテーションソフトのテンプレートとして、新しく「完了」の定義を作る際に役立てている。

B 社事例(3)では、「完了」の定義を作成していなかったため、スプリントレビューで「完了」の確認があいまいになってしまっていたと回想していた。

G 社事例(9)では、アジャイルコーチの勧めにより「完了」の定義をチームで作りチームの成長と共に少しずつ拡張させていった。定義を拡張していくことでチーム内での完了の認識のずれを防ぎ確実に進めることができた。また最初は定義に含まれていなかったユニットテストの自動化、受入テストの自動化なども含めていくことで完了のレベルを高めていった。

M 社事例(25)では、「完了」の定義を用いて開発を行っていた。ふりかえりのタイミングで、「完了」の定義を更新していた。最初は定義もどんどん変わっていった。

関連プラクティス

ユーザーストーリー
ふりかえり
ベロシティ計測
プロダクトオーナー

顧客プロシキ

3.4.3. **楽しい工夫** / “Fun” is important

“自分たちで自分たちのために工夫しよう”

別名:なし

要約

様々なところに工夫できる点はたくさんある。感じた違和感や発見されたボトルネックをみんなで楽しんで工夫する。その結果、チームが一体になり、問題の取り組む障壁が軽減する。

状況

職場環境がギスギスした雰囲気であったり、プレッシャーが高い状況であったり、良くない状態である。

問題

普段の仕事の中で、ちょっとした違和感を見つけたり、心理的なストレスの解消法を模索したり、工夫できるところを見つけたりしているが、なかなか改善に結びつかない。

例えば、プランニングポーカーを実施している時に、見積りに対する確実性はその都度変わってくる。これは確実だと感じて見積もる時もあれば、不確定要素が多く大きなズレがあると感じながら見積もることもある。そのような付加的な情報は、見積りを決める上で重要なのにも関わらず、伝えることが難しい。

例えば、会議で沈黙となり、形式的な議論に終始し、意義が薄くなってしまう。ファシリテータやリーダーが活性化を計ろうとしたところで、なかなか思い通りに進まない。

開発標準やコンプライアンスなどのルールや規約、プラクティスやノウハウ、慣習に基づいて進めているが、それらの規範の実施は時間が進むにつれ、規範を強化する方向へフィードバックがかかることが多いため、息苦しさを感じる人もいる。ゆとりがなくなり、効果的な成果に結びつくことが難しい状況になりやすい。

フォース

◆トラブルに対処する一番の方法は、そのトラブルにまっすぐ対応することではあるが、その状況のプレッシャーから、気疲れしてしまう

◆心理的障壁は、些細なことだと感じた場合でも、プロジェクト推進や業務に大きな影響を及ぼすことが多い。

解決策

自分たちで工夫を楽しもう。また、その楽しい工夫を受け入れよう。単に楽しむだけでなく、「楽しく」かつ「役に立つ」工夫を発見しよう。

自分たちが感じている違和感を表す言葉を探してみよう。他のチームメンバーも同じことを考えている場合も多い。

大きなリリースやトラブル解消の後、落ち着いた時に工夫を試す。仲間内だけに通じる共通の話題や趣味に根ざした工夫がよい。周囲から見るとふざけているように見えるかもしれないが、頭ごなしに否定するのではなく、周囲の人は様子を見守る。

工夫を実施する側も、興味を持たない人は参加させず、影響を与えないような配慮が必要である。多様な意見を尊重し、実施することが大切である。強制することは、ハラスメントやいじめに結びつくこともある。

これらの工夫はふりかえりの機会を用いて、アイデアを出して、いろいろ試みる。

組織にあったアジャイルスタイル、柔軟なプロセスなどのプラクティスは参考になる。

留意点

気軽にコミュニケーションが取りやすい環境を作るため、雑談などの目的を持たない会話をしよう。何気ない会話から課題が浮き彫りになったり、問題解決の糸口が見つかったりすることが多い。

複雑さや困難に打ち勝つためには、ルールに従うだけでは難しい場合がある。状況をよく把握し、いろいろ試す環境を用意する。

辛い仕事や単調な仕事こそ楽しむ工夫をしよう。

このような取り組みを批判し、止めさせることは簡単だ。しかし、試したいと思ったことを試せる環境が、次のイノベーションに結びつく。

効果

◆開発が楽しくなる、という意見が多い。
◆小さいけれども、効率の良い取り組みを自分たちで考え、変化や状況にあった環境に取り組むことができる。
◆チームの一体感が増す。

利用例

B 社事例(2)では、チームメンバーとファシリテータ（スクラムマスター）を兼任し、チームメンバーが持ち回りで実施している。持ち回りにすると機械的で退屈になる上、自己申告制にするにも人間関係の調整などからファシリテータ（スクラムマスター）を継続的に決めることが難しくなってきたためである。

ファシリテータはじゃんけんで決めるようにし、負けた人が罰ゲーム的に押し付けられるのではなく、勝った人が担当するようにした。また、朝会のファシリテータ（スクラムマスター）は毎日交代するようにし、ちょうど 5 人だったので 1 週間で一巡するようにじゃんけんで順番を決めていた。全員が本気でじゃんけんをすることで非常に盛り上がり、チームの雰囲気作りにとても役に立った（漢気じゃんけん）。

B 社事例(3)では、バグが発生した場合に、バグという言葉から受けるネガティブなイメージで開発者はストレスフルな状況であった。
そこでチームはバグというネガティブな表現ではなく、「ラーメンの具」を追加するというポジティブなイメージでバグの記録を付けるようにした。
バグの量が増えると、「大量のバグに対応する」のではなく「大盛りの具が乗ったラーメンを食べる」と捉えることができ、楽しく前向きに対応することができようになった。その管理表が大盛りのラーメンのように見えることから、管理表を大盛りで有名なラーメン店の愛称で呼びチームの文化になった。

G 社事例(9)では、開発現場に会話がなく、聞きたいことがあっても話しかけづらい雰囲気があった。そこで、開発ルームに常時置いているお菓子用の貯金箱を用意し、つまらない冗談を言った人は、100 円をお菓子購入用に罰金として貯金箱に支払うようなゲームにしたところ、現場を活性化を促すことができた（おやつ神社）。

G 社事例(9)では、プランニングポーカーの番号を出す時に、有名漫画の決めシーンをそれぞれ表現することによって、その見積に対する思いを同時に表現するようにした。

関連プラクティス

ふりかえり
柔軟なプロセス
組織にあったアジャイルスタイル

参考文献

懸田剛, (2007) 「レゴブロックを使った欠陥の見える化 バグレゴによる試行」, JaSST'07 Tokyo, http://www.jasst.jp/archives/jasst07e/pdf/A4-1.pdf

懸田剛 (2008)「バグレゴその後 レゴブロックから学んだもの」, JaSST'08 Tokyo, http://www.jasst.jp/archives/jasst08e/pdf/B2-2.pdf

3.4.4. 組織構造のバウンダリをゆるめる / Slacking boundary of organizational structure

“組織構造が生み出すバウンダリを意識しよう”

別名:

要約

組織やプロセスの構造が生み出すボトルネックがあれば、バウンダリをゆるめることによって、発生している問題を当事者である「我々の」問題として解決しやすい構造を作る。その結果、根底の問題を解決できるだろう。

状況

プロダクトには、様々な組織が関わっている。チームもひとつの組織であるし、その周囲にも様々な組織がある。

組織は多くの役割で構成されている。人の行動は、組織構造や役割によって規定される部分が大きい。

問題

組織やプロセスの構造が生み出すボトルネックがある。

プロダクトオーナーは、バックログの維持など、開発チームを含むステークホルダーとの調整や、継続的に発生する割り込みなどの状況で忙殺され、相対的にボトルネックになりやすい。

あるチームや役割の周辺には、別の役割との関係性がボトルネックになる場合がある。例えば、規模や業種、方針に従って、その業務に詳しく経験のあるドメインエキスパート、画像のデザイン部品を作るデザイナ、アプリケーションを維持するインフラストラクチャーやプラットフォームを開発維持する部門、品質保証部門や監査部門などなどの関係する役割や部門がある。その役割や部門は、専門性を高めることや、第三者機関のように評価や相互監視など調整機構を働かせるなどのメリットがある。しかし、役割を細分化することによって、分割された仕事の間での融通しにくくなってしまう。その結果、全体から考えると、役割の分割によってボトルネックを生み出す場合がある。

チーム内であっても、専門性が高く属人化していると同様に、そこにフローが集中してしまい、ボトルネックが発生してしまう。

このようにバウンダリを分けることによって得られる事柄があると共に、構造的にコミュニケーションやプロセスのボトルネックが生じてしまいやすくなる。

フォース

- ◆バウンダリを明確に切り分けようと思っても、複雑で現実的ではないが、その周辺こそ創造性を発揮しやすいところである。
- ◆コンテキストによって、それぞれの持つ意見が正しいことが説明できる。しかし、明確なバウンダリ内にいると他のコンテキストを理解しにくくなる。専門性を活かしたいが、原理主義的に自分の専門を持つことにより、他の状況を配慮せず深い対立を生み出すことがある。
- ◆専門知識の交換には時間が必要であり、異なった部門にいると、コミュニケーションの時間が限られる。

解決策

組織的な構造が生み出すバウンダリをゆるめよう。お互いに役割を越えてチームとして助けあおう。

どこにボトルネックが発生しているか仕事の流れを観察し、価値の流れを図示するバリューストリームマップやシステムを構成する要素の因果関係を表す因果ループ図などを用いて書いてみる。その中でボトルネックを発生させているところを特定し、解決策を考える。

このボトルネックの解消を目的とするアジャイル型開発のプラクティスがいくつも存在する。チーム全体を一つに、オンサイト顧客、顧客プロキシ、一つの場所に集まる共通の部屋、集団によるオーナーシップなどの実施を検討しよう。

プロダクトオーナーがボトルネックになる場合、ふたつの方針がある。一つは、プロダクトオーナーの人数を増やし、チームにすることである。もう一つは、プロダクトオーナーと開発チームを同じチームとし、その役割を意識せず同じチームとして問題に取り組むことである。

一つのチームにするだけでも、コミュニケーションの回数と密度は向上する。さらにプロダク

トオーナーを特定の役割とせず、チーム全員でプロダクトオーナーの責務も行うことによって、さらに役割や立場のバウンダリはゆるまる。重大なトラブルの発生時など責任を取る必要があるときも全員で対応する。

組織構造のバウンダリをゆるめるためには、属人性の排除や学習の機会が重要である。技術的な側面だけでなく経営、マーケティング、経理など幅広い視点に基づいた学習が有効なことが多い。教育目的を兼ねたペアプログラミングならぬペアプロダクトオーナーや、ピア・レビューなどの機会を用いて、属人化を解消しよう。

これらのボトルネックは、誰も漠然とは気づいているが言い出しにくいため、ふりかえりの時に話題にあげるのが良い。

ファシリテータ(スクラムマスター)は、ボトルネックに気づきやすい。負荷の高い人を支援するだけでなく、このままでいいのか、改善する方法はないのかをチームに問い掛けてみる。

チームが一体となって行動するためには、共通の目的、目標の理解が欠かせない。インセプションデッキ、ユーザーストーリーマッピングを用いて、目的や全体像をチーム全員でよく理解しておく。

留意点

既存のプロセスや組織構造の良いところを見極めながら行うとよい。

構造を変更させた結果、また新たな視点に基づいた問題に行き当たることがある。適切なフィードバックをしながら検討する。

第三者機関などの必要性については、経営者や必要な各部門などと調整する必要がある。

全体のスループットが一時的に低下するため、長期的な成果との見極めが難しい。

効果

◆役割のボトルネックが解消しチームのパフォーマンスが向上する。
◆別の観点が入ることによって多様性が高まり、さらに創造性が高まることが期待できる。

利用例

B 社事例(2)では、スクラムを採用しており、プロダクトオーナーが非常に多忙のため、ボトルネックになっていた。開発チームはプロダクトビジョンの段階からプロダクトオーナーと共に関わっていることもあり、プロダクトオーナーの業務をほぼ開発チームが行った。最終決定はプロダクトオーナーがしているが優先順位付けを開発チームに任せたところ、ボトルネックは解消した。

D 社事例(5)では、スクラムにおいてプロダクトオーナーと開発チームにわかれている状況であったが、プロダクトオーナーは、ビジネス企画がつまらなくなり、ボトルネックになることもあった。そのため、プロダクトオーナーも開発について知り、開発チームもプロダクト企画に関わり、より一体になることを目指した。

I 社事例(15)では、自社開発においてビジネス企画と開発チームが別の組織で、ビジネス企画がボトルネックになっている。ビジネス企画は、開発メンバーとの調整だけでなく、ビジネス企画を進める上で対外的な調整も必要であり負荷が高かった。そこで、開発チームとビジネス企画の人を１つのチームに集めて機能集約チームを結成したところ意思の疎通がよくなり、ビジネスが推進できた。

関連プラクティス

チーム全体を一つに
オンサイト顧客
顧客プロキシ
プロダクトオーナー
共通の部屋
集団によるオーナーシップ
ふりかえり
ファシリテータ(スクラムマスター)
インセプションデッキ
ユーザーストーリーマッピング

3.5. プラクティス適用時のよくある問題と対応策

本節では、事例の中で、プラクティスを適用する上で、よく見受けられた問題と、問題が引き起こすであろう結末、問題についての解決策をまとめる。

「よくある問題」は、今回の事例調査で、3 事例以上で発見されたものを指している。問題は事例が多かったもの順に紹介する。

問題に対応した解決策が調査した事例にある場合は、問題として取り上げていなかった事例も含め解決策を提示している。問題に対して、解決策が調査事例からは存在しなかったものについては、その旨記述して解決へのヒントを提示した。

3.5.1. プロダクトオーナーがボトルネック

問題

アジャイル型開発でボトルネックになりやすいのが、プロダクトオーナー、顧客、顧客プロキシなどのプロダクトバックログ供給者である。本調査では 6 事例が、プロダクトオーナーのボトルネックを問題に取り上げていた。

ボトルネックには大きくわけて、(1)準備作業が滞る、(2)決定が遅れる、の２種類があった。

(1) 準備作業が滞るボトルネック

以下の 4 事例で発見された。

B 社事例(2)では、プロダクトオーナーが多忙なため、ほとんど開発者と話せる時間がなかった。

C 社事例(4)では、プロダクトオーナー役を任されていた顧客の担当者が、通常の業務との兼任であったため多忙となり、要件の詳細化や、ユーザーストーリーの詰めができていなかった。

D 社事例(5)では、プロダクトーオーナーが 1 名、開発者が 6 名というチームで自社サービスを開発していたが、開発チームの開発速度が向上した結果、プロダクトオーナーの準備が間に合わなくなりボトルネックになってしまった。

H 社事例(13)では、オフショア開発を行っており、プロダクトオーナーだけが国内にいた。プロダクトオーナー1 名、開発メンバー10 名で構成されており、開発メンバーのスキルは申し分なかったが、プロダクトオーナーが他の仕事も兼任していたため、開発チームに渡すプロダクトバックログの準備に必要な時間を割けなかった。

(2) 決定が遅れるボトルネック

以下の 2 事例で発見された。

G 社事例(10)では、プロダクトオーナー役は現場の責任者が行っており、決断も素早かったが、予算を持っているわけではなかったため、決裁者に伺いを立てる必要性について常に気にかけなければならなかった。

J 社事例(15)では、スマートデバイスのアプリケーションを開発しているが、開発できたものの確認を営業にお願いしており、営業のチェックを終えないと公開ができなかった。そのため、営業の確認が遅れると、その分公開が遅れていた。

予期される結末

問題（1）の予想される結末は、プロダクトオーナーがボトルネックの状態が続くことで、開発速度にブレーキがかかる。そのため、いくら開発者に余力があってもプロダクトバックログが準備されていなければ、開発者の仕事が進まない。

問題（2）の予想される結末は、決定が遅れることだけでなく、決定を後で覆されてしまい、作業のムダを作る、あるいは開発速度に大きなブレーキをかけてしまう。

解決策

プロダクトオーナー役のボトルネック解消には、状況に応じて次の４通りのパターンが見受けられた。

1. プロダクトオーナー役にサポーターをつけることで解決する（C 社事例(4)）
2. プロダクトオーナー役の仕事の一部を開発者が代行することで解決する（B 社事例(2)、D 社事例(5)）
3. プロダクトオーナーに権限を委譲することで解決する（J 社事例(15)）
4. プロダクトオーナーを権限のある人に変更する（C 社事例(4)）

1. サポーターをつける

C 社事例(4)の場合は、開発チームから数名が顧客の会社に出向し、プロダクトオーナーのサポートチームとして専任で支援にあたることでボトルネックを解消した。

2. プロダクトオーナーの仕事を開発者が手伝う

B 社事例(2)の場合は、プロダクトオーナーのほとんどの仕事を開発メンバーが代行していた。優先順位づけも、ビジョンを物差しにして開発メンバーで考えていた。最終決定はプロダクトオーナー役に伺いをたてるが、かなりの部分で受け入れてもらっており、手戻りがわずかだった。

D 社事例(5)の場合は、プロダクトオーナーがそれまで行っていた詳細化の作業を開発チーム側に頼み、チーム一体で要件をメンテナンスすることでボトルネックを解消した。

3. プロダクトオーナーに権限を委譲する

J 社事例(15)では、これまで「クライアント確認」と呼ばれる営業確認を毎回行っていたが、その都度確認のための時間がかかっていた。そのため、ビジネス担当者に権限委譲を行い、クライアント確認を不要にして意思決定を素早くすることでボトルネックを解消した。

4. プロダクトオーナーを権限ある人に変更する

C 社事例(4)の場合は、担当者が担当レベルでは判断しかねる状況においては、顧客企業社長を含める経営陣が、即座に判断するような体制に移行した。厳密な意味でのプロダクトオーナーの変更ではないが、プロダクトオーナーがフロントの担当に加えて経営陣を巻き込むことで、経営に影響があるレベルの意思決定を即座に実施できる体制になった。

G 社事例(10)の場合は、実際に問題は発生はしなかったが、プロダクトオーナーが、毎回予算権限のある人物に承認依頼の返事を待つのではなく、組織の最終意思決定者がプロダクトオーナーになることで開発が素早く進むのではないかと考えていた。

新たな課題

これらの解決策には、以下のような新たな問題を引き起す可能性があることも考慮しておきたい。

1. サポーターに頼ってしまい、プロダクトオーナーが事実上機能しない
2. 意思決定プロセスを明確にしておかないと、プロダクトオーナー視点と、開発者視点のコンフリクトのためプロダクトの方向性に影響がでる
3. 権限を委譲されても、本人に責任を全うする気持ちがないと効果が発揮できない
4. 権限者に役割を変更しても忙しくプロダクトオーナーとしての役目を果すことができない

詳細はプロダクトオーナー、顧客プロキシ、オンサイト顧客も参考にされたい。

3.5.2. 分散拠点で円滑なコミュニケーションがとれない

問題

分散拠点で開発していくには、同一拠点と比べて特に拠点間のコミュニケーションの円滑化に注意する必要がある。本調査では 4 事例が分拠点での開発にまつわる問題を挙げていた。

G 社事例(9)では、プロダクトオーナーが別拠点におり、さらに開発期間中に開発者 1 名が別拠点に移動した。そのため従来実施してきた同一拠点での日次ミーティングを、分散拠点に対応させなければならなかった。

J 社事例(16)では、開発メンバーは同一拠点だが、デザインチームと顧客がそれぞれ別拠点にあり、三拠点間のコミュニケーションを円滑にしたかった。

H 社事例(13)では、開発をオフショア先に依頼しており、そことのコミュニケーションを円滑にする必要があった。

L 社事例(24)はプロダクトオーナーは日本に、開発メンバーは中国にいた。

分散拠点間でのコミュニケーションでは、コミュニケーションの即時性や応答性、情報量の制約だけでなく、信頼関係構築の手段（一緒に食事をする、仕事以外のプライベートな情報をやりとりする、など）も考慮しなければならない。

信頼関係が築けていないと、不信感や不満を持つことで、問題の早期開示を妨げる原因になり、問題の発見が遅れるリスクがある。

予期される結末

分散拠点で円滑なコミュニケーションが実現できない場合に起り得る問題は「コミュニケーション齟齬」である。意図や意味が思うように伝わらず、その結果、作業にズレが生じて、期待する成果がでてこない。

単にコミュニケーションのズレだけではなく、拠点間の信頼関係の醸成にも影響が出てくる。思うように意志を伝えられない、あるいは互いの状況がわからなくなると、物事を悪い方に想像してしまいがちである。その結果、相手を疑いはじめ、信頼関係が崩れていく。そうなるとアジャイル型開発で重要な、関係者同士のオープンなコミュニケーションが崩れ、隠蔽、報告の遅れなどの状況が生まれてしまう。

解決策

分散拠点で円滑なコミュニケーションを実施するために、大きく 3 つの解決策を採用していた。

ネット電話会議システム

大半の事例では、リアルタイムでのコミュニケーション手段として、ネット電話会議システムを用いていた。アジャイル型開発のリズムとなる日次ミーティング、イテレーション計画ミーティング、スプリントレビュー、ふりかえり、だけでなく、日常的なコミュニケーションのためにも利用されていた。

G 社事例(9)では、Web 会議システムを用いて日次ミーティングを実施していたが、予約が必要であったため、スマートフォンのスピーカーフォン機能を使うこともあった。日次ミーティングの実施中には、人数が多い拠点側がスマートフォンを持ちながら話すことで、少人数側にも声がきちんと届くように工夫していた。

L 社事例(24)では、ネット電話会議システムを用いて遠隔地間での日次ミーティングをはじめとするコミュニケーションを円滑化するためのプラクティスを実施した。

プライベートも含む情報共有

J 社事例(16)は、J 社と顧客先で共有している SNS を用いて、業務的な内容もプライベートな内容も含めて関係者間のコミュニケーションを図っていた。

同一拠点から分散へ

いきなり分散拠点で開発を始めず、同一拠点でチーム開発を経験した後に分散する工夫もされていた。

H 社事例(13)では、分散拠点で開発する前に、まず同一拠点で 1～2 週間開発するようにした

（段階的分散）。 この結果、最初に顔を見ながら一緒に開発したことで、その後のコミュニケーションがとりやすくなった。

[IPA 2012]では、中大規模(30 人以上)の分散拠点開発において「同一拠点から分散へ」という工夫が報告されていた。本調査では「同一拠点から分散へ」の事例として１件のみ報告されたが、分散拠点開発を実施する際には、コミュニケーションツールの利用の前に、始めは同一拠点での開発を行い、徐々に分散拠点に移行することが望ましい。

3.5.3. テストの自動化が後回しになってしまう

問題

本調査では３事例がこの問題を取り上げていた。

ユニットテストの自動化や、自動化された回帰テストは、変更に適応するアジャイル型開発では重要な位置付けである。

しかし実際の現場においては、プロダクトを継続的に開発していく上で重要になるのはわかってはいるが、プロジェクト開始時点からテストの自動化を始めなければいけないという動機づけが弱いという意見が上がっていた。（I 社事例(15)）

また、とりあえずテストの自動化なしに開発を進めていくが、テストの自動化を後回しにしてしまうと、機能を追加することに対して、テストの自動化は「必要な作業」ではなく「できればやりたい作業」という位置付けになる。そのためテストの自動化は短期的には負荷となる。

単純な計算では、テストの手動実行のコストと、テストの自動化にかかるコスト+テスト自動化による軽減コストとのバランスで決定される。実際には、新規機能追加に対しての圧力、未来の手動テストの予測実行コスト、未来のテスト自動化による予想削減コストを考慮した上で決定していくことになる。

長期的視点のプロダクトの品質を担保するテストの自動化と、短期的視点でリリースまでに実現したい機能とを比較した場合に、動いているコードのテストの自動化よりも、機能の実現を優先されてしまうという意見があった。（E 社事例(6)）

少しずつテスト自動化を整備しようとしてはいるが、プロダクトコードが優先されて、ついつい自動実行のためのテストコードを書くのが後回しになってしまうという意見もあった。（B 社事例(2)）

解決のポイントとなるのは、以下のフォースをどう考慮するかである。

◆工数を後から捻出することができない
◆ついテストを書くのを忘れてしまいがちである
◆テストの自動化に関するコストと、自動化によるメリットを比較した場合、プロジェクトの直近のリリースには費用対効果がそれほど得られないと判断される
◆テストの自動化に掛る工数がはっきりしない
◆設計が複雑で自動化しづらくなっている
◆テストを自動化するスキルが足りない

予期される結末

テストの自動化を後回しにしておくことで起り得る問題は、テスト実行がイテレーション内に収まらなくなる点である。

頻繁に変更していくプロダクトコードの品質を担保するために、変更された度に手動で回帰テストを実施していこうとするが、イテレーションを繰り返す毎にテストケースおよび手動テストの実行工数も増大する。この傾向が進むと、イテレーションの範囲内ではすべての回帰テストが実施できなくなる。

このような場合は、イテレーションとは別途テストフェーズのような期間を設けて回帰テストを実施する、イテレーション期間を長くする、部分的なテストの実行で対応する、という対症療法が考えられるが、いずれもアジャイル型開発の機敏さを損ねてしまう。

更には、「動くコードには手をつけない」という状態も起こり得る。自動テストがないプロダクトコードに対して開発者が修正してバグを混入させることを恐れ、できるだけ手をつけないようになる。
こうなると、変更を入れづらいプロダクトコードになり、アジャイル型開発の求める「変化に適応する」ことができなくなる。

解決策

テストの自動化を後回しにしないため、あるいは後回しになってしまった場合の解決策として前述の３事例以外も含めて以下の解決策が発見された。

ペアプログラミングでヌケ防止

B 社事例(2)では、ペアプログラミングを実施することで、つい後回しになりがちなユニットテストの自動化を、ペア同士の声かけによって防ぐことができていた。

自動化範囲の切り分け

E 社事例(7)では、テストを自動化をする部分と、

そうでない部分を切り分けて対応した。テストの自動化が複雑になりそうな部分は手動で、そうでない部分は自動化をするという形で対応した。

途中から自動化に取り組む場合は、現実的に効果の高そうな箇所をまず見極めて、現実的な範囲を絞って対応する。(F 社事例(8))

受入テストのみ(あるいは中心)自動化

また自動化する範囲において、ユニットテストではなく、受入テストを中心に自動化している事例もあった。

D 社事例(5)においては、ユニットテストは自動化せずに、シナリオ形式の受入テストをツールにより自動化した。

F 社事例(8)においては、元々ユニットテストがないプロダクトコードにユニットテストを自動化して整備しようとしたが、コストが膨大にかかることがわかったため、受入テストに絞って自動化を行うことで品質の担保を実現した。

G 社事例(10)では、ユニットテストと受入テストを両方自動化するための学習コストを減らしたかった。そのため、受入テストツールを用いて、ユニットテストの範囲と受入テストの範囲を自動化した。この場合すべてのユニットテストの範囲を自動化できたわけではなかったが、ツールの学習コストを考えると妥当であったと判断している。

まとめ

テストの自動化は、後から工数を捻出しようとしても調整は難しい。可能であれば、当初から導入検討しておき、「完了」の定義に列挙し、開発工数に含めた上での見積を行うことが望ましい。

テストの自動化は短期的なメリット（開発者の安心）以上に、長期的なメリット（変更の際にバグを検知できる）が大きい。そのため開発側が導入を検討したい場合は、テストの自動化を含めるメリット、含めないリスクを説明した上で可能な範囲で工数についての了解をとる必要がある。

詳しくは、ユニットテストの自動化、自動化された回帰テスト、テスト駆動開発を参考されたい。

3.5.4. リファクタリングができない

問題

本調査では 3 事例がこの問題を取り上げた。

契約種別の切り口で見ると、3 事例のうち、自社開発が 1 例（E 社事例(6)）、受託開発が 2 例（C 社事例(4)、K 社事例(20)）であった。

C 社事例(4)では、イテレーションの度に機能を実現しなければならず、顧客からのフィードバック、バグの対応などやることがどんどん増えていった。そのため、「動いているプロダクトコードの設計を、将来的なリスクを考慮して改善する」という特徴を持つリファクタリングの作業は、機能へのフィードバックやバグ対応と比較して優先順位が下げられてしまい 100%実施するまでには至らなかった。

E 社事例(6)では、リファクタリングという行為自体が計画に入っていなかった。小さな単位のリファクタリングはイテレーション中の工数で補うことができたが、大きなリファクタリングは工数を捻出できず、調整の必要がでてきた。

予期される結末

リファクタリングができない問題を放置しておくことで起こり得る問題は、頻繁に変更されていくコードベースのエントロピーの増大、つまり複雑さの混入に対して、対応ができないことだ。その結果、どんどんコードは複雑化、理解容易性が低下する。

開発者も当初の開発者以外の人間に変わる可能性がある。すると「そこで何を実施しているか」がわかりずらく、修正するのが困難なコードベースに変わっていく。

最終的には、テストの自動化と同様に、だれも触れない、変更できないコードになり、アジャイル型開発の特徴である「変化に適応する」ことができなくなる。

解決策

3 事例のうち、問題を解決してリファクタリングを充分に実施した事例は一つであった。

K 社事例(20)では、顧客企業が、リファクタリングを実施しないまま開発を進めたことによりプロダクトコードが複雑になりメンテナンスが困難になった過去の経験を持っていた。そのため、K 社に対して全体工数の 15%をリファクタリングの工数として見積りを出して欲しいと K 社に依頼していたため、工数を確保して実施することができた。

顧客やプロダクトオーナーからすれば、リファクタリングは「新しい機能を追加しないで、既存のコードを修正しているだけの手戻り作業」に見えてしまいがちである。K 社事例(20)の顧客企業のように過去に失敗をしていないと、その価値は伝わりにくい。

開発者は、リファクタリングを行わないことの潜在リスクを顧客に誠意を持ってあらかじめ伝えておくのが望ましい。リファクタリングは、手戻り作業というよりも、将来のメンテナンスコストを低減させるために定期的に実施する小規模メンテナンス作業であることを伝える必要があるだろう。

またリファクタリングの工数を事前にモレなく計画することはできない。そのため計画に工数として考慮するだけでなく、「完了」の定義としてリファクタリングを必須にしてして確実に実施していく必要があるだろう。

イテレーションの後半をリファクタリング実施にあてる(L 社事例(21))、リファクタリングを強制的に時間を設けて実施する(L 社事例(22))、リファクタリングを集中的に行うイテレーションを設ける(M 社事例(25))、といった工夫を用いてリファクタリングを開発作業の一貫として組込むとよいだろう。

詳しくはリファクタリングを参考にされたい。

3.5.5. 外から進行状況がわかりづらい

問題

アジャイル型開発では、チーム、関係者間での中で様々な情報共有が行われている必要がある。同一フロアにいるチーム内では状況が共有できているが、外部から見た時に進行状況がわかりづらい、という問題が 3 事例で取り上げられた。

F 社事例(8)では、プロダクトオーナーや企画側から見た時に、開発チームの内部状況が不明瞭であった。

G 社事例(10)では、プロダクトオーナーが別の拠点におり、見えづらい状況だった。

H 社事例(11)では、現在の進行状況がわかりづらい状況であった。

わかりづらい状況としては

1. 拠点が別のためわかりにくい
   (G 社事例(10))
2. 現場の状況がわからない
   (F 社事例(8))
3. 俯瞰的な状況がわからない
   (H 社事例(11))

という 3 つのケースが見受けられた。

状況がわかりづらいとは、透明性が損われているということである。「透明性」はスクラムを始めとして、アジャイル型開発における重要なポイントで、開発の状況が、隠し事なく常にわかっている状態である。いくら開発者の中だけで透明性を高めたところで、例えばプロダクトオーナーに見えないと、チームとしての透明性が確保できていないことになる。

透明性が足りないと、プロダクトオーナーは開発者がどんな問題に悩んでいて、どういう状態なのかが理解できなくなり不安になる。そのため、開発者に対して、わかりやすい状況報告を依頼するようになるかもしれない。すると開発者は本来しなくてもよい作業に手をとられてしまいパフォーマンスを発揮できない。

予期される結末

外から進行状況がわかりづらい状況を放置しておくことで起こり得る問題は、プロダクトオーナーがタイムリーに適切な判断が下せなくなることだ。

プロダクトオーナーが状況を把握するために、状況を知るための資料を必要となり、そのための工数を開発から捻出しなくてはいけなくなってくる。

調整は早めに行わないとならないが、状況が見えないと、様々なビジネス上の判断を素早く行うことができない。

状況が見えない状態で進んだ結果として、芳しくない成果を手にしてしまうと、プロダクトオーナーや顧客は「開発者が手を抜いている、無駄なことをしているのではないか」といった推測をしてしまい、開発者との溝ができてしまう。

解決策

F 社事例(8)では、SNS やチケットシステムを使い、とにかく徹底的に透明性を確保した。すべての情報をオープンにして、開発者の悩みまで共有対症とした。それにおって内部状況を理解する材料を揃えた。

G 社事例(10)では、スプリントの中間に、見通しをプロダクトオーナーに説明するイベントを設けて直接対面で状況を説明するようにしていた。

H 社事例(11)では、後述のリリースバーンダウンチャートを用いて、リリースに向けての見通しを一目でわかるようにした。

H 社事例(11)で用いているリリースバーンダウンチャートは、リリースまでの期間を反復回数で分割し、残りのストーリーポイントをプロットしていくチャートである。反復単位で用いるイテレーション（スプリント）バーンダウンチャートは、反復内のタスクの作業量をプロットするが、リリースバーンダウンチャートは期間も、単位も異なる。

拠点が別の場合は、開発の状況をマクロにもミクロにも可視化した上で、拠点間で共有できるようツールを配備する必要があるだろう。

詳細は、バーンダウンチャート、ベロシティ計測、共通の部屋を参考にされたい。

3.5.6. 真の顧客からのインプットやフィードバックがない

問題

アジャイル型開発の主要な動機である「フィードバックを受け入れる」に対して、開発チームと直接やりとりするプロダクトオーナーや顧客プロキシではなく、本当にシステムを利用する人（=真の顧客）からのインプットやフィードバックがもらえない、という問題を取り上げた事例が３事例あった。

F 社事例(8)では、要件定義とテストフェーズはウォーターフォールで実施し、開発フェーズをスクラムで進めた。ユーザーのシステム部門とは合意をとりながら進めてはいるが、最終的に利用者が触れるのはリリース後になってしまう。そのため、それまでの３ヶ月の間はフィードバックが貰えていない。（フィードバックサイクルが長すぎる）

G 社事例(9)では、タブレット PC 上で動作する工場点検システムを作っていた。プロダクトオーナー役は情報システム部門の担当であり、プロダクトオーナーが、工場の現場の意見をとりまとめて要求を開発者に提示していた。しかし開発者は、実際に使用される現場、つまり工場がどのような場所で、点検作業が具体的にどういうものかを全く把握できていなかった。

J 社事例(17)では、発注元と、真の顧客が別の組織であった。そのため発注元との合意はとれていたが、真の顧客からのフィードバックは貰えておらず、利用状況もわからないことが多々あった。

予期される結末

真の顧客のフィードバックがないことで起こり得る問題は、期待したものとの相違が後で発覚することだ。

真の顧客からのフィードバックが後になればなるほど、使い物にならないシステムを提供してしまうリスクが高まるが、これは致命的である。

解決策

F 社事例(8)では、カナリアリリースとよばれるβユーザー向けリリースを行うことで、感度の高い一部のユーザーからのフィードバックを受け付けることで対応した。

G 社事例(9)では、開発者が実際にシステムが利用される工場の現場を訪問し、工場見学を実施した。
実際の利用者に、現場で開発中のシステムを触ってもらいながら質疑応答を行った。開発者は、利用者が使っている様子を観察する、あるいは開発者自身が作業を体験することで、業務の流れ、様々な環境的制約を発見することができた。その結果、利用者から出る要望だけでなく、利用者が口には出さない現場環境におけるフィードバックを反映することができた。

J 社事例(17)は解決策がまだ見つかっていなかった。

F 社事例(8)、J 社事例(17)の抱える問題は、真の顧客の巻き込みが足りていない場合に発生する。

できるだけ早期から真の顧客を巻き込むことが重要だ。例えばスプリントレビューに呼ぶ、デモイベントを開催する、などの工夫が必要であろう。それでも真の顧客に来てもらえないのであれば、チームが出向いてデモをして感想をもらうとよい。

イテレーションの度にフィードバックをもらうのが難しいのであれば、いくつかのイテレーションをまとめて見てもらう機会を設ける、また、自由な時に触れてフィードバックをもらえる環境を作る(C 社事例(4))といった工夫も必要だろう。

また利用者に早期に触れてもらうために、機能スコープを最小限にしてリリース、あるいは真の顧客に触れてもらえるまでの期間を短くしてフィードバックを貰う頻度を増やすことを検討する。そのためにユーザーストーリーマッピングを用いると、最小限のスコープを調整することがおこないやすい。

4. 活用事例

本章では、調査の対象となった全 26 事例について、個々の事例プロフィールを紹介する。また、個々の事例が活用したプラクティスのうち、特徴的なプラクティス、うまくいかなかったプラクティスについて解説する。

各プラクティスや独自の工夫と課題についての詳細は、「3 プラクティス解説」を参照してほしい。

事例毎のプラクティスの適用については、「付録 2 事例とプラクティスの対応」にまとめてある。読者の環境に似たコンテキストの事例がある場合には「付録 2 事例とプラクティスの対応」を使ってその事例で活用しているプラクティスを参照されたい。

4.1. **事例**(0): A **社**

事例プロフィール

A 社事例(1)と同様に広告配信システムの開発であるが、広告の集め方が異なる点と、高速かつスケーラブルなサーバクラスタ、無停止で安全な保守を実現すべく、関与人数が多くなっている。

自分達に必要なことを行っているだけで、「アジャイル型開発を実施している」という認識はなく、プラクティスとして必要なものを掻い摘まんで実施している状況であった。

プロジェクトではなく自社サービスとして開発を行っているため、「自分達のサービス」という意識が強い。
Web サービス開発ではあるが、早期リリースよりも安定したシステムの提供を重視しており、保守性を重んじている。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	広告配信システム
プロジェクト工数	20 人月(初期リリースまで) 以降も継続中
プロジェクト期間	6 ヶ月（初期リリースまで） 以降も継続中（2 年経過）
プロジェクト関与者人数	開発チームは 8 名（開発以外を含めると 20 名）
開発言語	Java, PHP, Perl
開発拠点	同一拠点
開発手法	Scrum+XP
成功度	4：まだ開発途中だが、うまくいきそうな見込みはある

表 4　A 社事例(0) プロパティ

使用プラクティス群

技術プラクティスを中心に、手法にとらわれず、その時に必要なものを採用している。

特徴的なプラクティス

- 日次ミーティング
- タスクをチケットとして登録し、それを見ながら実施。夕方開催している。
- 組織に合わせたアジャイルスタイル
- マネージャ（予算責任)とディレクター（要件責任)と開発の三者の役割分担、必要なプラクティスのみ実施
- コマンド化されたデプロイ手順を準備。Jenkins などのツールに依存させない

うまくいかなかったプラクティス

- リリース計画ミーティング
- 3 ヶ月の大きな目標は共有しているが、その先は変わっていくので厳密にはしていない。
- ペアプログラミング
- 開発者によりエディタ、キーボードの好みが違いペアプログラミングしづらかった。
- 受入テスト
- 検収環境で確認し OK を貰う程度。受入テストを自動化しようと試みたがツールが不安定でうまくいかなかった。
- 集団によるオーナーシップ
- システム規模が大きくエンジニアも多いためなんとなく担当が決まっていため、集団によるオーナーシップを実現するに至らなかった。

契約形態との関係

自社サービスのため契約は発生せず

チーム編成・チームメンバ研修との関係

メンバーはすべて社内で調達した。特に研修は実施しておらず、自主的な学習に任せている。

組織的に、技術研修の会場提供を積極的に実施しており、誘致した研修に社員が参加することで新しい技術の習得などを促進している。

また、社内でのフォーマル/インフォーマルな技術勉強会が活発に行われている。参加率はフォーマルな勉強会は 6 割くらい、インフォーマルな勉強会は 2 割くらいの人が参加している。

組織の人事考課の評価項目に、技術スキルへの取組が含まれている。仕事で作ったものを評価面談時にデモをしてエンジニアから評価される。

技術力を高める仕組みを組織的に整えている。

使用ツール

構成管理	git
IDE / エディタ	Eclipse、商用 IDE、Emacs
ビルド	Make, Maven
CI	Jenkins
チケット管理	商用統合開発管理システム
その他	-

表 5　A 社事例(0)の使用ツール

4.2. 事例(1): A 社

事例プロフィール

広告配信のためのプラットフォーム開発プロジェクトの事例である。開発メンバーは1名でセールス担当と事業マネージャの3名で進めている。事業の発案や予算コントロールはマネージャが行い、セールスは広告代理店との折衝やお金の調整を行っている。サービスのビジョンを3人で話し合いながら進めていった。開発者も広告代理店に同行し、できるだけ一緒に動いている。

「計画通り作る」のではなく「どうビジネスを始めていけるか」というビジネスゴールを最重視して開発を進めている。

組織的にエンジニアの技術スキルを重視しており、技術スキルの取組がエンジニアの人事考課に評価項目に入っていたり、エンジニア同士のインフォーマルな勉強会の頻繁な開催が実施されるなど、組織的に技術力の向上を支援している。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	広告配信システム
プロジェクト工数	-
プロジェクト期間	-
プロジェクト関与者人数	開発1名、セールス1名、事業マネージャ1名
開発言語	Web 部分は Ruby、広告部分は Flash (ActionScript)
開発拠点	同一拠点
開発手法	Scrum+XP
成功度	-

表 6　A 社事例(1) プロパティ

使用プラクティス群

スクラムやXPをベースにしているものの、手法に囚われずに必要なプラクティスを選択している。特に技術的なプラクティスを重要視して実践していた。

プロセスに関するプラクティスも実施はしているが、技術的プラクティスだけで実施が手一杯であり、実施の優先度も下がっているため、プロセスに関して深く実践したり改善したりはできていなかった。

特徴的なプラクティス

- イテレーション計画ミーティング
- タスクレベルの詳細な計画は作らずユーザーストーリーレベルの見積による計画をしていた。
- プランニングポーカー
- ストーリーサイズは倍か半分かぐらいのざっくりとした規模で見積した。
- スプリントバックログ
- 最初は付箋で実施していたが、マネージャの出張が多いためチケットシステムを使うようにしたが一度頓挫した。現在は再度チケット管理システムを利用しはじめている。
- ユニットテストの自動化
- サーバ側の自動化と、ローカル側の実行の両方を実施している。
- ユニットテストが常に成功であることを維持するので精一杯だった。
- リファクタリング
- 非常に重視しており、日常的に、手を抜かずに実施していた。
- 集団によるオーナーシップ
- プロジェクト内外にソースコードを閲覧可能にしており、プロジェクト外部の人からレビューや修正パッチが送られてくることもある。
- コーディング規約
- 「Ruby らしいコードを書くように」とだけしている。

うまくいかなかったプラクティス

- リリース計画ミーティング
- 計画通りに行うことではなく、ビジネスとして始めていけるかをゴールに設定していた。そのためリリース計画を重視していない。

◆ タイムボックス
◆ 厳密に実施していない。そのためイテレーション中に割り込みがあっても作業を受け付けるようにしている。
◆ ふりかえり
◆ 実施結果をふりかえるだけで、問題や改善の深堀はしていない。
◆ ファシリテータ
◆ 結果的には実施していない。ただ意見として挙がったのは、最初はファシリテータの存在、その価値を理解していなかったが、実際に開発を進めていく上で、後になってミーティングの進行などでファシリテータの有り難さに気づいた。
◆ バーンダウンチャート
◆ 利用していたが、更新を怠ってから段々風化して利用をやめた。
◆ スプリントレビュー
◆ 事業マネージャやセールス担当には、機能ができた時点で随時レビューしてもらっているため、定期的なイベントとしての「レビュー」タイミングは不要であった。
◆ インセプションデッキ
◆ 書いてはみたが、そこで終わってしまった。
◆ 共通の部屋
◆ 各自の席は近くだが、チーム毎に集まって座っているわけではない。
◆ 持続可能なペース
◆ 持続可能なペースを尊重しながらも必要な場合は休日出勤も厭わない。
◆ 受入テスト
◆ 受け入れ項目は「広告がでているか」というレベルで手動確認に留まる。

契約形態との関係

自社サービスのため開発についての契約は存在しない

チーム編成・チームメンバ研修との関係

開発者は１人のため、特に研修は行われてはいない。組織としてインフォーマルに技術者同士の交流を持ち、技術的スキルを向上させる機会が多かった。

使用ツール

構成管理	git
IDE / エディタ	NetBeans
ビルド	Make, Maven
CI	Jenkins、Guard[14]
チケット管理	付箋 商用統合開発管理システム
その他	インテグレーション環境をクラウドサービス上に仮想環境を構築しその上で実現

表 7　A 社事例(1)の使用ツール

[14] https://github.com/guard/guard

4.3. 事例(2): B 社

自社サービスをアジャイルで開発する事例である。案件に応じて開発手法を選択するのではなく、アジャイル型開発ができる案件を選んでいる。

事業部としてアジャイル型開発に取り組んでいる。アジャイル型開発は、成果物を見ながらフィードバックしていくため、経営層にも注目を浴びている。アジャイル型開発を社内で目立たせるために、受託開発ではなく自主開発を選んでいる。
当事例は、スマートフォン向けのメールクライアントを提供する。チームでサービスを検討しながら実施している。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	スマートフォン用クライアント
プロジェクト工数	24 人月
プロジェクト期間	6 ヶ月
プロジェクト関与者人数	4 人
開発言語	Java
開発拠点	同一拠点で同一フロア
開発手法	Scrum+XP
成功度	4: かなりの部分でうまくいった

表 8　B 社事例(2) プロパティ

使用プラクティス群

アジャイル型開発にフォーカスし、スクラム ＋XP で実施している。

教科書通りのプラクティスの実施というよりも、自分達の現場の状況に合うように、独自の工夫をしているものが多かった。

またチームはプロダクトオーナーはいるものの、開発者がかなりの部分でプロダクトオーナー的な責務を支援しており、一般的なスクラムよりも、プロダクトオーナーと開発者の境界はゆるくなっている。

特徴的なプラクティス

- イテレーション計画ミーティング
- プロダクトオーナーが多忙なため、チームメンバーが交代でプロダクトオーナーの責務をサポートしながら実施している。ユーザーストーリーやペルソナ／シナリオ法、インセプションデッキなどを試している。優先順位付けもチーム内で行っている。
- イテレーション
- 最初は二週間であったが、ビジネス的にスピード感が足りず一週間に変更した。その結果サービス企画へのフィードバックも迅速にできた。
- イテレーション計画ミーティング
- タスクを洗い出す際には、全員でタスクを出すのではなく、ストーリーごとに人が分かれ、分担してタスクを洗い出し、最後に集まって過不足タスクをチェックすことで時間短縮した。
- プロダクトバックログ（優先順位付け）
- プロダクトオーナーが多忙なため、チームメンバーが交代でプロダクトオーナーの責務をサポートしながら実施している。優先順位付けも開発者が行っている。
- アジャイルコーチ
- 社内のアジャイルコーチを中心として、各チームのスクラムマスターが集まって「スクラムマスター道場」と呼ばれる勉強会を開催している。
- 日次ミーティング
- 朝だけでなく、夕方も実施している。朝はプロダクトオーナーも含めてチーム全員で実施し、夕方は開発者のみ集まり問技術的問題点を解決するために実施している。
- ふりかえり
- KPT を中心に実施している。活気を維持するために、笑いが起こるような話題を意図的に含めていた。
- バーンダウンチャート
- メンバーがもちまわりで記録して日次ミーティング（朝）で確認している。

◆ タスクボード（タスクカード）
◆ 付箋を使ってのタスクボードがチームに好評であった。一度ソフトウェアシステムで管理してみたが、全体が見えない、ログインが面倒、といった理由によりすぐに付箋に戻した。
◆ 柔軟なプロセス
◆ ふりかえりを活用し、積極的に変更した。
◆ インセプションデッキ
◆ 最初の 5 項目（プロジェクトの概要、背景、目的と妥当性、要求事項、環境）のみ実施した。
◆ プロダクトオーナー
◆ プロダクトオーナーが多忙なため最初はうまく機能しなかった。余裕のある担当者に引き継がれると、うまく回るようになった。実際に動ける人でないと機能しないことを痛感した。
◆ 人材のローテーション
◆ 半年単位でチームメンバーを入れ替えて新しい視点を導入していた。
◆ 迅速なフィードバック
◆ 社内でユーザテストを実施し、ユーザ・エクスペリエンスを改善した。
◆ 持続可能なペース
◆ プロダクトオーナーとの関係性作りを大切にし、お互いの状況を共有している。
◆ 自動化された回帰テスト
◆ プロジェクト後期に実施し、効果が高かった。
◆ スパイク・ソリューション
◆ プロジェクト開始時に良く実施した。
◆ リファクタリング
◆ アーキテクチャも含めて常に実施していた。リファクタリングをチーム全体で推奨していた。
◆ 継続的インテグレーション
◆ ツールは使っていないが、統合開発環境が自動でビルドしていたため代わりになった。
◆ 集団によるオーナーシップ
◆ 他人が作った部分でも積極的に変更し、特定の人しか編集できないという状況は作っていない。

うまくいかなかったプラクティス

◆ リリース計画ミーティング
◆ 一度計画を作ったが、リリースの日程が無期限延期状態になってしまい、マイルストーンが決められなくなり、半ば放置状態になってしまい使われなくなった。
◆ ペアプログラミング
◆ 当初は、強制的にペアを作り実施していたが、うまくいかなくなった。スキル差が大きいと、スキル不足な人が積極的にならない傾向があり、結果品質向上に寄与しなかった。現在はレビューに移行している。
◆ 共通の部屋
◆ 希望したが、実施できなかった。
◆ ファシリテータ（スクラムマスター）
◆ 最初はスクラムマスターを立てていたが、メンバー内で持ち回るようにして、現在は特定の個人を立てることはやめた。チーム全員でファシリテートしている。
◆ テスト駆動開発
◆ ユニットテストの自動化は必須にしていた。啓蒙しているが、実施できていないことがあった。ペアプログラミングを行うと実施度合いは高くなった。チームのメンバーの意識をテスト駆動開発を実施するように変えるのが大変だった。

契約形態との関係

自社開発のため開発についての契約は存在しない

チーム編成・チームメンバ研修との関係

全員が開発をし、ファシリテータ（スクラムマスター）はじゃんけんで決めていた。

業務時間内で毎週二時間の勉強会を実施している。ここで他チームとの交流が多く生まれていた。

自分たちのチーム編成や持続可能なペースについても議論してきた。チームで話し合いをする際には、グループでの対話的な手法を用いていた。

使用ツール

構成管理	-
IDE / エディタ	-

ビルド	-
CI	-
チケット管理	-
その他	-

表 9　B 社事例(2)の使用ツール

4.4. 事例(3): B 社

当事例は、事例(2)を参考にしながら実施しているため、重複部分が多い。スマートフォン向け画像コンテンツの販売サービスを開発している。
自社サービスをアジャイルで開発する事例である。案件に応じて開発手法を選択するのではなく、アジャイル型開発ができる案件を選んでいる。アジャイル型開発は、成果物を見ながらフィードバックしていくため、経営層にも注目を浴びている。アジャイル型開発を社内で目立たせるために、受託開発ではなく自主サービスを選んでいる。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	スマートフォン向けサーバおよびクライアント
プロジェクト工数	12 人月
プロジェクト期間	3 ヶ月
プロジェクト関与者人数	4 人
開発言語	Java + JavaScript
開発拠点	同一拠点で同一フロア
開発手法	Scrum+XP
成功度	4: かなりの部分でうまくいった

表 10　B 社事例(3) プロパティ

使用プラクティス群

アジャイル型開発にフォーカスし、スクラム ＋XP で実施している。
定期的に評価環境へデプロイを行い、迅速なフィードバックを得るようにしている。

特徴的なプラクティス

- イテレーション計画ミーティング
- ユーザーストーリーをタスクに分割する際は、ストーリごとに分担して行い、後で集まって過不足タスクを洗い出すことで時間短縮した。
- プロダクトバックログ（優先順位付け）
- リーンキャンバスやユーザーストーリーを丁寧にプロダクトオーナーと作り、その後はチームがメインで行っている。
- アジャイルコーチ
- 社内のアジャイルコーチを中心に読書会を開催し、各チームが集まっている。
- 日次ミーティング
- 朝会だけでなく、夕会も実施している。
- ペアプログラミング
- 難しい部分のみペアプログラミングしている。
- ふりかえり
- KPT を中心に実施した。活気を維持するために Keep に必ず笑えるネタを入れた。
- バーンダウンチャート
- もちまわりでプロットし、日次ミーティング（朝会）で確認した。
- 自動化された回帰テスト
- プロジェクト開始時から実施し、効果が高かった。
- テスト駆動開発
- テスト駆動開発が苦手な人と重点的にペアプログラミングをしたところ、効果を上げ始めた。
- インテグレーション専用マシン
- 自社で提供しているクラウドサービス上にサーバを作っているため、容易に準備できる。チームのメンバー全員がサーバ立ち上げから運用までをできるように教育している。

うまくいかなかったプラクティス

- リリース計画ミーティング
- プロジェクト開始からミーティングを行っているものの具体的に落とし込めていない。期日もスコープも明確ではないため、とりあえず、動くものを作ってみようと話している。
- ユニットテストの自動化
- サーバサイドの JavaScript で出来るところのみ実施している。レビューでフォ

ローしている。
- 逐次の統合
- Web 上のバージョン管理システムでは、ビルドを実施している。しかし、テストは自動化されず、ローカル開発環境のみで実施していた。

契約形態との関係

自社開発のため開発についての契約は存在しない

チーム編成・チームメンバ研修との関係

全員が開発をし、ファシリテータ（スクラムマスター）はストローによるくじ引きで決めていた。
業務時間内で毎週二時間の勉強会を実施しており、自分たちのチーム編成や持続可能なペースについても議論してきた。

使用ツール

構成管理	-
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	-
その他	-

表 11　B 社事例(3)の使用ツール

4.5. **事例**(4) : C **社**

事例プロフィール

C 社の顧客であるサービスプロバイダーが、それまでの自社提供サービスの開発を特定のベンダーに委任していたが、反面、自社の思い通りにならなかった。

そのため C 社にサービスリプレイス開発を依頼した。単純なサービス移行ではなく、新しい要件も加えながら開発したいとの要望を聞き、C 社から顧客にアジャイル型開発を提案して開始した。

リプレイスといいながらも、元のサービスはブラックボックスのままで、暗中模索の中、要件を聞き出し顧客と一体となりながら開発を進めていった。要件が固められない部分のみアジャイル型開発を行い、要件が明らかな部分(インフラ回り、メールエンジン開発)についてはウォーターフォール型開発を実施した。国内 3 拠点での分散開発事例でもあった。

プロジェクト中盤に大幅なスコープの見直しが発生したが、アジャイル型開発のおかげで、サービスのコアに注力する計画に変更しリリースに漕ぎ着けた。顧客プロキシチームを設けることで顧客側のボトルネックを解消した。

事例タイプ	サービス開発／中規模プロジェクト
会社タイプ	中小 SIer・ソフトハウス
システムタイプ	メール配信サービス
プロジェクト工数	500 人月
プロジェクト期間	2 年
プロジェクト関与者人数	59 名(開発者 32 名、その他 27 名) WF 部分は 5 名程度
開発言語	Java
開発拠点	東京、地方を含めた 3 拠点
開発手法	XP+WF
成功度	3 （うまくいった所もあるが、うまくいかなかった所もある）

表 12　C 社事例(4) プロパティ

使用プラクティス群

ふりかえりを軸として、進むにつれて様々な改善を行なっていること、また大規模特有のチーム構成やスキル伝達などに工夫をこらしているのも特徴的である。

他方、大規模ゆえに開発者の人数が増えてしまったため、自己組織的なチームの構築や、規模が増えた事によるコードの品質確保について、充分ではなかった、という意見もあった。

特徴的なプラクティス

- 日次ミーティング
- 一日二回のミーティングを段階的に実施していた（キーマンによるミーティング→チームごと)
- ふりかえり
- イテレーション単位と、四半期単位の実施をしていた。チーム混成で実施することでチームに閉じた改善を防いだ。
- スプリントレビュー
- デモだけでなく顧客にじっくり触ってもらうために専用の操作スペースを用意した。
- プロダクトバックログ（優先順位付け）
- リプレイスではあるが、既存機能についても「本当に継続する価値があるか？」を判断して機能を吟味した。
- 「最低限ほしい」と「オプション」を明確化した。
- コストと収益のバランスで順位付けをした。
- プロダクトオーナー
- 担当者が判断できない状況においては、顧客社長を含める経営陣が即座に判断するような体制に移行した。

◆ 顧客プロキシ
◆ 開発企業が、顧客サポートのため出向して対応した。
◆ ファシリテータ
◆ 管理チームがファシリテートして開発とプロダクトオーナーの間を取り持つようにした。
◆ チーム全体が一つに
◆ プロジェクトが中規模だったため、なかなか意識統一ができておらず、プロジェクトのクレドを作成して心をひとつにしょうと試みた。
◆ ユニットテストの自動化
◆ 自動化されたユニットテストがないとメンテナンスの際にプロダクトコードを変更することができなくなるため、重要視して実施していた。
◆ スパイク・ソリューション
◆ プロトタイプを作って検証を行った。
◆ 技術的難易度が高くハイリスクな部分はウォーターフォールで開発した。

うまくいかなかったプラクティス

◆ 自己組織化チーム
◆ 社員 20 名、全体で 60 名の大人数を自己組織化チームにするのは困難であった。
◆ 迅速なフィードバック
◆ プロダクトオーナーから開発者個人に「今日中にやってほしい」と頼まれるなど無理な要求が発生していた。
◆ 受入テスト
◆ 顧客と受入テストについて詰めきれず「何を確認すればよいのか」整理できていなかった
◆ 集団によるオーナーシップ
◆ チームの単位で所有権が閉じてしまっておりうまくいかなかった。

契約形態との関係

◆ 四半期毎に契約更新をした。
◆ 顧客とは準委任契約（時間精算無し)で行ったが、開発パートナーとは準委任契約（時間精算あり)での契約にした。

チーム編成・チームメンバ研修との関係

チームメンバは漸進的に追加されていったため、参画の都度にアジャイル教育を実施し、ペアプログラミング、人材のローテーションなどを用いて知識の伝播を積極的に行っていた。

使用ツール

構成管理	-
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	タスクボード 表計算ソフト Mantis
その他	遠隔地とオンライン通話システムのチャット機能を活用

表 13　C 社事例(4)の使用ツール

4.6. 事例(5): D 社

事例プロフィール

SNS のサービス開発における事例。アジャイル型開発経験者が皆無の状態で、社内アジャイルトレーナーが支援しながら開発を進めた。
開発者は若手が多くスキルが低いため、エンジニアリングの基本を教えながらの開発にならざるを得なかった。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	SNS
プロジェクト工数	18 人月
プロジェクト期間	3 ヶ月
プロジェクト関与者人数	6 人
開発言語	Java、PHP、Ruby
開発拠点	同一拠点、同一フロア、別の島
開発手法	XP
成功度	開発はかなり成功。ビジネス価値へはこれからだが、だいぶうまくいってきた

表 14　D 社事例(5) プロパティ

使用プラクティス群

メンバーが若手中心であり、元々はスクラムを中心にアジャイル型開発を導入していたが、エンジニアリング面が弱いとうまくいかないことがわかり、XP に代表される開発プラクティスを中心に導入する方向性に切り替えた。

他方、XP で代表されるテスト駆動開発などは取り上げず、ユニットテストの自動化ではなく、受入テストをシナリオベースで自動化して品質を担保しているのが特徴的である。
また、社内における人材教育の側面も強い事例であった。

特徴的なプラクティス

- イテレーション計画ミーティング
- 4 週分の計画ををユーザーストーリーレベルで作成する。
- イテレーション
- 教育という観点があるので一週間程度の短さでないと、問題が長く滞留しそうで不安である。
- 日次ミーティング
- 通院や早く帰ることを隠さずに表明している。
- ふりかえり
- 「5 つのなぜ」を最初は 2 時間で実施していたが、その後 1 時間になり今は 30 分で実施できるようになった。楽しく行えている。
- 柔軟なプロセス
- うまくいっていないところを中心に変えている。
- スプリントバックログ
- チケットシステムはうまくいかなかったので付箋で管理するように変更した。
- 顧客プロキシ / プロダクトオーナー
- サービスプロデューサーという役割の人物が顧客プロキシ/プロダクトオーナーを担っていた。この役割は通常プロダクトオーナーなどに求められるビジネス、交渉能力、推進能力、要件を落とせる人物像に加えて、開発知識も必要であった。
- ファシリテータ（スクラムマスター）
- 期間限定でトレーナーが実施するが、後はチームに委譲する。ファシリテータ希望者を募っていた。
- アジャイルコーチ
- 社内のアジャイルトレーナー（コーチ）が XP のプラクティスを最初は教えるが、半年でフェードアウトするようにしている。
- 持続可能なペース
- 無理はせず、1 日 8 時間。
- 自動化された回帰テスト
- Selenium を用いてバグ率が減った。
- 受入テスト

- 1 日 1 回実施、所要時間は 30 分程度。
- スパイク・ソリューション
- 「ちょっと作ってみる」というのが増えてきた。
- リファクタリング
- ユニットテストが十分に整備されている状況ではないが、ソースコードのブラッシュアップを定期的に行っている。
- シンプルデザイン
- 「汎用的に」「スケールを考えて」「時間があったら」を思考停止ワードとして注意している。
- コーディング規約
- 困らないことは作らない。問題があったら初めて作る方針である。
- 紙・手書きツール
- 会社に付箋や消せる紙、マスキングテープなどの備品を大量に用意している。

うまくいかなかったプラクティス

- バーンダウンチャート
- タスクの時間を見積っていないので使えていない。バグ数をバーンダウンチャートで管理している。
- スプリントレビュー
- プロダクトオーナーが近くに同席しており、すぐに確認してもらえる環境のためイテレーション単位では不要である。
- インセプションデッキ
- 試してちあが成功したことがない。プロジェクトではないのでプロジェクト憲章は必要ない。
- ニコニコカレンダー
- 以前は使用していたが、日次ミーティングでメンバーの状況がわかるので使用中止した。
- テスト駆動開発 / ユニットテストの自動化
- ユニットテストがなく実施していない。代わりに受入テストとしてシナリオベースの画面テストを自動化している。

契約形態との関係

自社開発のため開発についての契約は存在しない

チーム編成・チームメンバ研修との関係

アジャイル型開発のトレーニングは社内アジャイルトレーナーが実施。ふりかりのタイミングでも勉強会を実施している。

使用ツール

構成管理	-
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	付箋 商用チケット管理システム
その他	Wiki: 商用 Wiki システム 受入テスト: Selenium Web Driver

表 15　D 社事例(5)の使用ツール

4.7. **事例**(6) : E **社**

事例プロフィール

損害保険会社が社内業務の申請、依頼で利用するワークフローアプリケーションの開発プロジェクトの事例である。開発メンバーは８名で、アジャイル型開発の導入、実践のために社外コンサルタントが１名、週１回程度で参画していた。
最初に「システム構想書」を１ヶ月かけてじっくりと描き上げて、あとは構想書で検討したアーキテクチャに沿って、イテレーション開発を実施した。
明確なプロダクトオーナーは置かず、チームがプロダクトオーナーも兼ねていた。方向性は組織内でのレビューと週次の進捗報告にて確認した。この週次の報告の他にも、担当役員から報告を求められた時に都度説明、動いているソフトウェアを見てもらいながら説明を行っていた。

事例タイプ	社内システム開発／初めてのアジャイル
会社タイプ	ユーザー企業のシステム子会社
システムタイプ	社内業務の申請、承認ワークフロー
プロジェクト工数	15 人月
プロジェクト期間	10 ヵ月
プロジェクト関与者人数	チーム８名 社外コンサルタント１名 ※メンバーは他のプロジェクトとの掛け持ちであった。
開発言語	C#
開発拠点	同一拠点
開発手法	Scrum
成功度	仕組み部分がうまく機能しており、かなりの部分でうまくいった。

表 16　E 社事例(6) プロパティ

使用プラクティス群

スクラムをベースとして開発を行っているものの、調査対象のプラクティスをほぼ全般利用している。これは、開発を担当した部署が、社内の技術調査を担う部門であったため、アジャイル型開発の実験的な試行、評価のミッションを持っていたことが影響している。

特徴的なプラクティス

- イテレーション
- イテレーション期間に限らず、タイムボックスを絶対に守るという方針をチームで共有していた。ツールで時間を測りながら打ちあわせを行うなど徹底していた。
- プランニングポーカー
- タスクやストーリーを見積もるときは必ず使用していた。意見が出せないメンバーでもポーカーならば必然的にカードを出さなければならなくなるため意見を聞くことができた。
- 日次ミーティング
- タイムボックスを決めて必ず実施していた。
- ふりかえり
- KPT を基本として、お互いを認め合う場として開催していた。
- バーンダウンチャート
- コミュニケーションスペースとなっており、進捗状況を確認するほかにも、メンバーの気持ちやチームのルールを書き込むスペースとして役立っていた。

うまくいかなかったプラクティス

- リファクタリング
- 小規模なリファクタリングはイテレーションの中で実施できたが、大きなリファクタリングは計画上織り込み辛く、計画見直しの必要が生じた。

契約形態との関係

自社開発のため開発についての契約は存在しない。

チーム編成・チームメンバ研修との関係

チーム内でアジャイル型開発をリードしたメンバーが認定スクラムマスターを取得している。

使用ツール

構成管理	Subversion
IDE / エディタ	商用 IDE(C#)
ビルド	-
CI	-
チケット管理	Redmine
その他	-

表 17　E 社事例(6)の使用ツール

4.8. **事例**(7): E **社**

事例プロフィール

損害保険会社がパッケージソフトウェアを活用して構築した基幹業務システムの事例である。基幹業務システムとあって、開発期間は長期にわたっており、最初のフェーズを終えるまで 4 年を要している（プロジェクトは現在も継続している)。

ウォーターフォールとアジャイルを組み合わせたハイブリット型の開発プロセスを採用。業務要件定義を実施した後に、画面インターフェイスの定義から結合テストまでを反復型開発した。

複数チームの構成で、各チーム内にアーキテクトを設置、チームを跨いだ設計課題をアーキテクト間で解決するようにしていた。また、中国にオフショアの開発チームが存在した。
ステークホルダーには、各組織階層における会議を定期的に開催して、状況を報告していた。

事例タイプ	社内システム開発／中規模プロジェクト
会社タイプ	ユーザー企業のシステム子会社
システムタイプ	事務処理支援を行う基幹業務システム
プロジェクト工数	-
プロジェクト期間	4 年
プロジェクト関与者人数	-
開発言語	Java, COBOL
開発拠点	国内同一拠点 オフショアは中国大連
開発手法	Scrum+WF
成功度	まずまずまずの評価

表 18　E 社事例(7) プロパティ

使用プラクティス群

スクラム+WF のハイブリット型の開発であったため、プロセス系のプラクティスは比較的多く利用されている。一方で、設計開発系のプラクティスについての利用が少ない状況となっている。

特徴的なプラクティス

- プランニングポーカー
- 実施チームの見積り精度は高かった。

うまくいかなかったプラクティス

- かんばん
- 更新が追いつかなくなり、形骸化した。
- アジャイルコーチ
- 現状の組織文化を踏まえずに、いきなりアジャイルにはできなかった。

契約形態との関係

- 準委任契約で実施
- E 社は受託側(一次請け)

チーム編成・チームメンバ研修との関係

社外コンサルタントによる研修を取り入れていた。研修を受ければ、アジャイル開発手法を理解し遂行できる下地のあるメンバーを人選した。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	商用統合開発環境
CI	-

チケット管理	商用統合開発環境
その他	自動テストツール:商用ツール

表 19　E 社事例(7)の使用ツール

4.9. 事例(8) : F 社

事例プロフィール

工期短縮を目指してプロジェクト管理をするための社内システムであり、想定最大ユーザ数は 6 万人で、実際のユーザはその一割弱である。当初は、ウォーターフォール型開発で行っており、優先順位やスコープについて、ユーザ部門と合意形成をせずに、開発部門の都合で決めていた。

その後、サブシステムの結合テスト時に問題が発生し、プロジェクトが一旦休止した。その後、アジャイル型開発として再開発し、バックログを作り、ユーザ部門と合意形成をしながら進めた事例である。現在、既に本番リリースは終了し、その後のエンハンスおよび保守開発の状態である。

事例タイプ	社内システム開発／中規模プロジェクト
会社タイプ	大手 SIer
システムタイプ	社内システム
プロジェクト工数	-
プロジェクト期間	初回リリースまで 3 ヶ月、トータルで 2 年。
プロジェクト関与者人数	最大 30 名
開発言語	C#
開発拠点	東京およびインド
開発手法	Scrum+TOC
成功度	-

表 20　F 社事例(8) プロパティ

使用プラクティス群

開発手法を選択する際に、いったんはアジャイル型開発も含め特定の開発手法を忘れ、ビジネスや開発の背景、及び特性を見極め、何を期待するのかをよく検討してから選択を行った。例えば、スクラムを用いてスプリントをまわしているが、エンドユーザがそれを確かめられるのは 3 ヶ月後であることもあり、メリットを享受できない状況もある。開発の特性によっては、ウォーターフォール型開発がふさわしい場合もある。

現在の開発は基本的にスクラムであるが、制約理論（TOC）と組み合わせている。プロセスを固定しているに近い状況である。制約理論を用いて、プロセスの流れをスムーズにするため、ボトルネック工程とボトルネックのリソースを可視化している。

スクラムを用いるメリットとポイントは透明性の確保である。テストのカバレッジ、バグ対応、機能実施、及びテスト実施状況などをオープンにし、品質も分析、またレポートを報告し、経営層やマネジメント層の理解を得ている。
受入テストの自動化を行い、リリースまでのデリバリ時間を短くするなどの工夫して、バグがあっても一日以内にリリースできるようにした。

要求の妥当性を確かめるための独自の環境を用意し、要求を出した担当者や、新しい機能にいち早く導入することを許容する関係者（アーリーアダプター）が確認する。例えば、要求が実現した場合の画像イメージだけを貼付けるなど、工数をかけずに確認環境を用意することによって、その要求の妥当性を事前に確認する。

特徴的なプラクティス

- イテレーション
- 3回の1ヶ月スプリントでリリースするリズムであった。
- プロダクトバックログ(優先順位付け)
- 徹底的に話し合って、合意形成するようにした。
- オンサイト顧客
- 分散開発拠点のインドに、プロダクトについて詳しい人員を常駐させた。
- 受入テスト
- 受入テストを自動化し、その状況を社内に公開している。
- 組織にあわせたアジャイルスタイル
- 制約理論(TOC)など複数の手法を取り

入れている。

うまくいかなかったプラクティス

- ユニットテストの自動化
- 当初、ユニットテストは存在しなかった。どの機能やコードが使われているかを測定し、変更が多い箇所から順番にユニットテストを自動化していき、最終的にはカバレッジ率が 60%まで拡大した。しかし、それ以上のユニットテストの自動化はあきらめ、リファクタリングと一緒に行うようになった。
- ユーザーストーリー
- アジャイル型開発でよく使われているユーザーストーリーに対しては、要求収集についてのテクニックの一つであり、常に効果があがるわけではないという見解を持っていた。そのため、ユースケースや他の手法とあわせて実施している。
- 人材のローテーション
- 知識や経験を保持するために、他のプロジェクトに人を異動させないようマネジメント層に具申した。

契約形態との関係

- 自社開発のため、開発についての契約は存在しない
- オフショア先とは準委任契約

チーム編成・チームメンバ研修との関係

最大の時には日本に 10 名、インドに 20 名の合計 30 名である。

使用ツール

構成管理	商用統合開発管理ツール
IDE / エディタ	商用 IDE for .Net
ビルド	商用統合開発管理ツール
CI	商用統合開発管理ツール
チケット管理	商用統合開発管理ツール
その他	-

表 21　F 社事例(8)の使用ツール

4.10. 事例(9): G 社

事例プロフィール

Ruby の特徴を活かした開発を実証することを目的に、県が公募し採択された実証事業プロジェクト。エンドユーザは県内の製造業メーカーで、工場内の日常点検をシステム化してコスト削減が目的であった。開発会社は Ruby の特徴を充分に活かした開発ができておらず、またエンドユーザも従来の内製システムに課題を抱えており、両者が Ruby、アジャイル型開発、SIer と協働開発にチャレンジした。実証事業のため、エンドユーザと開発会社間では契約は結ばれていなかった。

アジャイル型開発について初の試みであったため、アジャイルコーチを招聘し、チームビルディングから、アジャイルプラクティスの段階的導入を支援してもらっていた。開発メンバーの半数が Ruby での開発経験がない状態で開始したが、最終的には技術的プラクティスを駆使できるまでメンバーが成長していった。

初期開発は同一拠点で行い、途中から分散拠点で開発を進めた。
実際のシステムが使われる工場に見学に行き、現場の点検員の行動を見ながらシステムに対するフィードバックを得ていた。

事例タイプ	社内システム／初めてのアジャイル
会社タイプ	中小 SIer・ソフトハウス
システムタイプ	工場の日常点検システム
プロジェクト工数	20 人月
プロジェクト期間	5 ヶ月
プロジェクト関与者人数	5 名(開発 4 名、PO1 名)
開発言語	Ruby、Ruby on Rails
開発拠点	初期:同一拠点 後半:分散拠点(開発会社とエンドユーザ会社)
開発手法	Scrum+XP
成功度	4(実証事業としての評価は高い。ただしエンドユーザから「もっとできると思ったのに」と言われたこともあり)

表 22 G 社事例(9) プロパティ

使用プラクティス群

スクラムを基本として、技術プラクティスを段階的に導入していった。チームが状況に合わせて、ふりかえりのタイミングなどで少しづつプラクティスを増やしていった。継続的インテグレーションはできてはいなかったが、テストの自動化はユニットテスト、受入テスト共に実践していた。

特徴的なプラクティス

- イテレーション
- 反復漸進型は必須であった。最初は一週間であったが休日だと 4 日になってしまうため、適時一週間か二週間のどちらかに決めていた。「完了」の定義が反復毎に成長していった。
- イテレーション計画ミーティング
- ペーパープロトタイプで受け入れ条件の確認時間を短縮していた。
- プランニングポーカー
- 素早く楽しく見積る工夫。”せり”スタイル。
- 日次ミーティング
- Web 会議システムで遠隔地のメンバーが日次ミーティングに参加した。
- ふりかえり
- KPT をベースに実施。テーマ設定を重視した。
- タスクボード
- プロジェクトルームの壁に貼り出しユーザーストーリーとタスクを見える化。

◆ スプリントレビュー
◆ Cucumber でシナリオを事前共有し時間短縮。
◆ アジャイルコーチ
◆ 社外アジャイルコーチが当初は教育及びミーティングのファシリテータを行った。プラクティス導入についてアドバイスした。
◆ チーム全体が一つに
◆ プロジェクト開始時に開発者、顧客を含めてプロジェクトについての期待と目標を書出して、関係者全員が期待と目標を共有して開始できるようにした。
◆ 共通の部屋
◆ パーティションで区切られた開発室を設置した。周囲の壁を利用して情報を貼り出した。
◆ 受入テスト
◆ 途中から Cucumber を用いてシステムの振る舞いを自然言語で記述し自動化。

うまくいかなかったプラクティス

◆ 継続的インテグレーション
◆ デプロイはレビュー前だけで継続的ではなかった。CI ツールは使う余裕がなかった。

契約形態との関係

◆ 県の実証事業のため、エンドユーザとの契約はなかった。
◆ 費用は県が持ち、ソースコードと報告書を納品。

チーム編成・チームメンバ研修との関係

◆ 開発メンバー4 名のうち、1 名は Ruby では初めての開発、1 名は本格的な開発が初めてだった。
◆ 開発メンバー及びプロダクトオーナーはアジャイル基礎教育とワークショップを受講した。
◆ 同じ開発環境上で、プロジェクト開始後の一週間、有名 SNS サービスのクローンを開発し、アジャイル型開発のリズムと環境に慣れた。

使用ツール

構成管理	Subversion(SaaS)
IDE / エディタ	NetBeans
ビルド	Rake[15]
CI	-
チケット管理	Redmine(SaaS) タスクボード(付箋、模造紙)
その他	ユニットテスト: Rspec 受入テスト: Cucumber 自動配備: Capistrano[16]

表 23　G 社事例(9)の使用ツール

[15] Ruby によるビルドの自動化ツール
http://rake.rubyforge.org/
[16] 自動配備ツール
https://github.com/capistrano/capistrano

4.11. 事例(10) : G 社

事例プロフィール

事例(9)で開発した点検システムをベースにして機能拡張をした設備点検システム。実証事業の成果を元に、アジャイル型開発を進めるにあたって県から 50%の補助がでた。契約としては顧客と請負契約を結んだ。製造現場の業務改善を目指したものであり、日常からの改善活動と同様に不確実性の高い（継続的に改善される）ソフトウェアの開発であったため、アジャイル型開発が適当だと考えた。
一週間のお試し期間を設けて、顧客にアジャイル型開発を体感してもらった後、本格的に開発を進めた。また最初に工場見学をしてシステムが利用される現場を見た上で、プロダクトバックログに落とした。
終盤、顧客の期待に応えようとしすぎて残業が増え高負荷な状態になってしまった。
プロダクトオーナーとしてエンドユーザーを配置したため、判断が早かった。また開発も直接エンドユーザーと接することでモチベーションが高まった。

事例タイプ	社内システム開発
会社タイプ	中小 SIer・ソフトハウス
システムタイプ	設備点検システム
プロジェクト工数	7.5 人月
プロジェクト期間	2.5 ヶ月
プロジェクト関与者人数	3 名
開発言語	Ruby
開発拠点	同一拠点、同一フロア
開発手法	Scrum+XP
成功度	4（顧客の期待にこたえようと持続可能でない作業量をこなしてしまった）

表 24 G 社事例(10) プロパティ

使用プラクティス群

G 社事例(9)のメンバーが、新しくチームを作って開発したため、多くの部分を継承しているが、テストの自動化や、顧客との関係の中で更に進化していた。

現場の責任者に直接プロダクトオーナー役を担ってもらったため、顧客とのやりとりに関してのプラクティスに特徴があった。

特徴的なプラクティス

- ユーザーストーリー
- 顧客にユーザーストーリーの書き方を教育してから書いてもらった。
- プロダクトバックログ
- ユーザーストーリーマップを利用した。
- プロダクトオーナー
- プロダクトオーナーは現場責任者。
- ユニットテストの自動化
- 二つのツールを同時に覚えるコストを掛けられなかったため、ユニットテストツールの RSpec ではなく受入テストツールの Cucumber に絞って使用した。Cucumber で完全ではないがユニットテストの領域の自動化をカバーできた。
- 継続的インテグレーション
- Jenkins を利用してテストを実行。デプロイまではやっていない。

うまくいかなかったプラクティス

- プロダクトオーナー
- 現場責任者がプロダクトオーナーになったが、現場の最適化の視点で会社の全体最適化の視点が抜けていた。
- ファシリテーター（スクラムマスター）
- 顧客との関係もよかったためスクラムマスターを立てる必要がないと判断したが、補助してくれる人がいなくなりプロダクトオーナーが困っていた。
- 受入テスト
- シナリオで受け入れ基準を合意。Cucumber で受入テストを自動化したが結果を顧客には公開していなかった。

契約形態との関係

- 請負契約であった。
- 費用の 50%を県が助成した。

チーム編成・チームメンバ研修との関係

メンバーは３人であったが、事例(9)プロジェクトの経験者がいたため、新規メンバーに初期段階で簡単な教育のみを実施した。

プロダクトオーナー役の顧客にはユーザーストーリーの書き方を教育した。

使用ツール

構成管理	Subversion(SaaS)
IDE / エディタ	NetBeans
ビルド	-
CI	Jenkins
チケット管理	Redmine(SaaS) タスクボード(付箋、模造紙)
その他	ユニット/受入テストフレームワーク: Cucumber 自動配備: Capistrano

表 25　G 社事例(10)の使用ツール

4.12. 事例(11): H 社

事例プロフィール

音楽コンテンツ配信（販売と試聴）サービス開発の事例である。開発メンバーは 1 チーム当たり平均 6 名､全体で 4～6 チームが動くプロジェクトであった。ビジネスおよびシステム全体についての責任は事業本部長が担い、個々のサブシステムについてはそれぞれの企画部署、開発プロジェクトとしての責任はシステム開発部長が担う体制であった。
大きな技術的課題に対しては、チーム横断的なミーティングを実施して、問題解決に当たった。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	音楽コンテンツ配信サービス用システム
プロジェクト工数	工数としてはなく、1000 ストーリーの開発を行った
プロジェクト期間	1 年
プロジェクト関与者人数	4～6 チーム 1 チーム平均 6 名
開発言語	Java, C#, Objective-C
開発拠点	国内同一拠点 一部、メンバーは中国
開発手法	Scrum
成功度	新規、小規模な機能は良いが、影響範囲の広い規模の大きな機能については、メンテナンスが難しいわかりづらいプロダクトコードを改善しづらかった。

表 26　H 社事例(11) プロパティ

使用プラクティス群

スクラム+XP をベースとして開発を行っており、プロセス系のプラクティスは数多く利用している。一方で、設計開発系のプラクティスについては、部分適用やカスタマイズして利用する傾向となっている。

特徴的なプラクティス

- ベロシティ計測
- 通常､ベロシティのモノサシとなる見積り単位のストーリーポイントはチーム間で比較するものではない。しかし本事例は複数チームで構成されるプロジェクトのため、ベロシティについては 1 ポイント=1 日という単位で統一して、プロダクトオーナーが複数チームを跨いで理解できる単位にした。
- オンサイト顧客
- オフショアメンバーとはオンラインのツールを利用してコミュニケーションを取った。

うまくいかなかったプラクティス

- 特になし

契約形態との関係

- H 社が発注元となり､オフショアチームと準委任契約を締結した。
- 国内協業は、派遣契約。

チーム編成・チームメンバ研修との関係

チームは、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メンバーで大半が構成されていた。
社外コーチを一定時期（半年）開発チームに配置した。

使用ツール

構成管理	-
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	-
その他	-

表 27　H 社事例(11)の使用ツール

4.13. 事例(12): H 社

事例プロフィール

エンターテイメント系サービスの開発プロジェクトの事例である。開発メンバーは総勢 24 名。内訳は、iOS アプリ開発チームが 7 名（内 3 名がオフショアメンバー)、Android アプリ開発チーム 6 名、Web 担当チームが 2 名（2 名ともオフショアメンバー)、共通アプリケーションチームが 2 名 (2 名ともオフショアメンバー)、プロダクトオーナーが 5 名、スクラムマスターが 2 名。プロダクトオーナーの 1 人が事業責任を負っていた。

初期段階でインセプションデッキを使い、ビジョンやプロジェクトの認識を合わせた。マルチプラットフォーム対応のアプリケーションの為、プロダクトオーナーと開発チームをプラットフォーム毎に分け、並行開発を行えるようにした。週次でミーティングを開催し、プロダクトに関するチーム間のズレを予防した。

また、プロダクトオーナーも同フロアにおり、毎日日次ミーティングにも参加してもらい、スプリント計画、スプリントレビュー、ふりかえりにも参加してもらうことで、プロダクトの方向性、状況を逐次確認、共有するようにしていた。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	エンターテイメント
プロジェクト工数	-
プロジェクト期間	8 ヵ月
プロジェクト関与者人数	チーム 24 名
開発言語	Java, C#, Objective-C
開発拠点	国内同一拠点 一部メンバーは中国
開発手法	Scrum
成功度	かなりの部分でうまくいった。短い周期（スプリント期間が 1～2 週間）で PDCA サイクルが回るため、課題や改善点が早期に抽出され、対処を行えた。

表 28　H 社事例(12) プロパティ

使用プラクティス群

スクラム+XP をベースとして開発を行っている。テスト系のプラクティスも含めて、全般的にプラクティスを広く利用している。リリース計画ミーティングやイテレーション計画ミーティングなどの利用はなかった。

特徴的なプラクティス

- インセプションデッキ
- 10 の質問では足りない部分を補うために、「スマートフォンのプラットフォームのバージョン」、「下位互換性」などの質問を追加して利用した。
- オンサイト顧客
- 開発メンバーの隣に、プロダクトオーナーの席が来る様に席配置を行い、コミュニケーションがより円滑になる様に工夫した。
- 組織に合わせたアジャイルスタイル
- スクラムでは許されていないスプリント中の割り込みも自社に合った形の為にカスタマイズした。
- コードの共同所有
- プラットフォーム毎に担当を分け、担当のソースコードについては複数名が共同で責任を持つ運用にしている。

うまくいかなかったプラクティス

- 特になし。

契約形態との関係

- オフショアチームと準委任契約を締結

した。H 社が発注元。

チーム編成・チームメンバ研修との関係

チームは、トレーニングを受ければ、手法の手順のうち自由裁量の部分を遂行することができるメンバーで大半が構成されていた。

使用ツール

構成管理	-
IDE / エディタ	Eclipse 商用 IDE(ObjectiveC、C#)
ビルド	-
CI	Jenkins
チケット管理	-
その他	-

表 29　H 社事例(12)の使用ツール

4.14. 事例(13): H 社

事例プロフィール

マーケティングデータ分析システムの開発プロジェクトの事例である。開発メンバーは 11 名。うち、10 名がオフショア開発メンバーであり、プロダクトオーナー1 名が日本にいた。
プロジェクトは、最初の 1 ヵ月でプロダクトの実現性検証を行った。その後、プロダクトバックログを作成し、プロジェクト計画の立案、プロトタイプの開発をこちらも 1 ヵ月で行った。その後のスプリント開発は、1 スプリント 2 週間で実施している。
オフショア側とは日次ミーティングも含めて、リリース計画ミーティング、イテレーション計画ミーティング、スプリントレビュー、ふりかえりといったスクラムで主に定義されているイベントでコミュニケーションを図っている。

事例タイプ	社内システム開発
会社タイプ	サービス事業者
システムタイプ	マーケティングデータ分析システム
プロジェクト工数	50 人月
プロジェクト期間	5 ヵ月
プロジェクト関与者人数	1 チーム 11 名
開発言語	Java
開発拠点	国内と海外の分散拠点。プロダクトオーナーは国内、開発チームは中国
開発手法	Scrum
成功度	プロジェクト継続中のため未回答

表 30　H 社事例(13) プロパティ

使用プラクティス群

スクラム+XP をベースとして開発を行っているため、プロセス系のプラクティスをよく利用している。設計開発系のプラクティスに関しては検討段階のものが多いが、継続的インテグレーションやユニットテストの自動化など、品質を維持するためのプラクティスは活用している。

特徴的なプラクティス

- 特徴的なプラクティスの活用は見受けられなかった。
-

うまくいかなかったプラクティス

- 特になし（未評価や検討中のものが多い)。

契約形態との関係

- オフショアチームと準委任契約を締結した。H 社が発注元。

チーム編成・チームメンバ研修との関係

オフショア先には、社内のコーチを配置した。

使用ツール

構成管理	-
IDE / エディタ	-
ビルド	-
CI	Jenkins
チケット管理	-
その他	-

表 31　H 社事例(13)の使用ツール

4.15. 事例(14): H 社

事例プロフィール

スマートフォン向け健康情報サイトの開発プロジェクトの事例である。開発メンバーは 9 名。最終的な責任は事業部長が持つが、権限を委譲された事業部門のメンバーがプロダクトオーナーを担当した。また、プロダクトオーナーは国内、開発チームは中国というオフショア開発を行った。技術面とプロセス面での支援を目的としてアーキテクトとスクラムマスター経験者を日本側に配置した。
初期のスケジュールはプロダクトオーナーが決定・提示していたが、スプリント開始後はプロダクトオーナーと開発チームが実績を踏まえて相談する形で調整を行い、開発を進めた。

事例タイプ	自社サービス開発
会社タイプ	サービス事業者
システムタイプ	スマートフォン向け健康情報サイト
プロジェクト工数	約 35 人月
プロジェクト期間	2 年
プロジェクト関与者人数	1 チーム・9 名
開発言語	C#
開発拠点	国内同一拠点 開発チームは中国
開発手法	Scrum
成功度	社外ステークホルダとの調整コストが高く、プロジェクト進行に影響した。 チームの技術力が不足しており、期待する品質を得られていない。 プロダクトオーナーの経験が浅く、進捗管理、要求伝達等に改善課題が残った。

表 32 H 社事例(14) プロパティ

使用プラクティス群

スクラム+XP をベースとして開発を行っているため、プロセス系、設計開発系、いずれのプラクティス群もほぼすべて利用している。一方で、自動化されたテスト、テスト駆動開発など、テストに関するプラクティスは利用していないものもある。

特徴的なプラクティス

- リリース計画ミーティング
- イテレーション計画ミーティングに含めて運用。
- スプリントバックログ
- スプリントバックログはチームで管理しており、プロダクトオーナーがコミットしていない。該当スプリントで開発対象とするプロダクトバックログアイテムについては両者でコミットしている
- 組織に合わせたアジャイルスタイル
- オフショアに対応してオンラインミーティング、画面共有などの工夫を行った。
- スパイク・ソリューション
- 初期段階でプロトタイプを作成。
- シンプルデザイン
- ホワイトボードによる設計セッション、設計レビューを実施

うまくいかなかったプラクティス

- ユニットテストの自動化
- 全コードに対してのユニットテストの自動化は、テストコード実装工数に見合った品質向上メリットが得られないと感じ実施しなかった。

契約形態との関係

- H 社が発注元となり、オフショアチームと準委任契約を締結。

チーム編成・チームメンバ研修との関係

オフショアの開発チームがアジャイル型開発未経験であったため、プロジェクト立ち

上げ段階でアジャイル型開発に必要な技術とプロセスに関する教育を実施した。

使用ツール

構成管理	Subversion
IDE / エディタ	商用 IDE(C#)
ビルド	-
CI	-
チケット管理	-
その他	ユニットテスティングツール: NUnit

表 33　H 社事例(14)の使用ツール

4.16. 事例(15): I 社

事例プロフィール

会社組織が分社化したため、全体でのガバナンスを効かせていた状況から、各事業部毎に IT 戦略の個別最適に変更になった。
以前は社内で定義した独自アジャイル型開発プロセスを採用していたが、現在はそのプロセスをベースにしつつも、自由度を高めて現場が工夫しやすいようにしている。

以前は開発側の効率化だけで短納期を目指そうとしていたが、事業側を巻き込んで、ビジネス企画から開発までを短くしないと本当の最適化にならないことがわかった。

Android、iOS、Windows8 などのスマートデバイスに対して、部門として 50 以上のアプリを提供している。競合他社との競争が激化しており、特にスピードが求められる状況で、QCD とのトレードオフでの低下をどう抑えるかに苦心している事例。
組織デザインと、開発基盤の組合せで、短納期、高品質の実現を試みている。

事例タイプ	自社サービス開発／中規模(組織展開)
会社タイプ	サービス事業者
システムタイプ	マルチプラットフォームのスマートデバイスアプリ
プロジェクト工数	まちまち
プロジェクト期間	1～3 ヶ月
プロジェクト関与者人数	部門全体で 60 名、開発は数名程度
開発言語	Java, Objective-C, HTML5/JavaScript
開発拠点	同一拠点、同一フロア、ワンロケーション
開発手法	Scrum
成功度	5（スマートデバイスアプリ開発においては適合度が高かった）

表 34 I 社事例(15) プロパティ

使用プラクティス群

「開発スピードの向上、高品質化」を目的に、特定の手法に囚われずに必要なプラクティスを実践している。

フィードバックに対応するために、反復型で実践しているというよりも、短期間でアプリをリリースし、継続してメンテナンスするためのプラクティスが中心になっている。逆にフィードバックの機会を増やし漸進的に開発していくという観点は少ない。

また共通の部屋、権限委譲、共通ゴールの設定、スキルローテーション、個人の意識改革などを積極的に行い、組織的なボトルネックを最小限に抑える仕組みを作っている。

特徴的なプラクティス

- バーンダウンチャート
- リリース日までの残日数と残作業を日々追跡していた。
- オンサイト顧客/プロダクトオーナー
- １人の企画者に決定権を委譲している。
- 共通の部屋
- 担当するアプリケーション単位で島を作っていた。
- 気軽に集まって会議ができるスペースを設けた。
- 人材のローテーション
- 担当している Web サイトの知識、スキル（iOS、Android）といった観点で人材をローテーションして知識を広めていった。
- 組織に合わせたアジャイルスタイル
- タイムボックスは明確にしていなかった。
- 持続可能なペース
- プロジェクト自体が継続性を求めていく。メンバーにアンケートやヒアリングを実施している。
- 受入テスト

◆ QA チームがテストシナリオを実行していた。
◆ リファクタリング
◆ Sonar[17]を使い解析し、スコアが低いものをリファクタリングした。
◆ 集団的なオーナーシップ
◆ ソースをチーム横断で見られるようにしている。
◆ 就業時間外にコミットがあった場合は、問題が起こっていないか確認するようにした。

うまくいかなかったプラクティス

◆ イテレーション計画ミーティング、イテレーション
◆ リリースまでの残日数で管理しているためイテレーションを強制していない。席が近いため会議体は設けていない。
◆ ユニットテストの自動化
◆ 機種依存のバグの方が多く、ユニットテストの自動化では担保できない。
◆ ふりかえり
◆ 絶対ルールとしていたが、定着しなかった。イテレーションを強制していないため区切りがつかない、開発速度を重視しているため開発を進めることが優先されたためと推測される。

契約形態との関係

◆ 自社開発のため開発についての契約は存在しない。
◆ 開発パートナーとは準委任契約であった。

チーム編成・チームメンバ研修との関係

スキルの高いエンジニアを基盤チームに配置して、各プロジェクトのエンジニアのサポートを行った。

使用ツール

構成管理	Subversion,StatsSVN
IDE / エディタ	-
ビルド	Maven
CI	Jenkins
チケット管理	Redmine
その他	バーンダウンチャート: 表計算ソフト テスト: TestFlight(iOS)[18] QA 管理: Sonar

表 35 Ｉ社事例(15)の使用ツール

[17] コード品質管理ツール
http://www.sonarsource.org/

[18] スマートフォンのフリーテストサービス
https://testflightapp.com/

4.17. 事例(16) : J 社

J 社の親会社がサービスオーナーであり、親会社からの受託開発である。
マルチプラットフォーム・アーキテクチャを採用し、HTML5 によるスマートデバイス向けアプリケーションを開発している。

事例タイプ	サービス開発
会社タイプ	メーカーのシステム子会社
システムタイプ	ソフトウェアプロダクト
プロジェクト工数	-
プロジェクト期間	６ヶ月
プロジェクト関与者人数	1 チーム 10 人で 2 チーム
開発言語	Java, JavaScript
開発拠点	開発者は同一拠点（デザインチームは別拠点）
開発手法	XP
成功度	開発中のため評価できず

表 36　J 社事例(16) プロパティ

使用プラクティス群

コミュニケーションに重点を置いているため、日次ミーティングや紙・手書きツールのほか、自社で販売している社内・組織向けソーシャル・ネットワーキング・サービス(SNS)を用いて、チャット機能と共に徹底的なコミュニケーションを実現している。メンバーのブログ、議事録、検討結果は掲載され、発注元にも公開している。

リリース手順書や仕様書は、チケット管理システムおよび構成管理システムを使って管理し、発注元が確認を行っている。

業務終了後のテーブルにディスプレイを含めた物品を全く置かないようにするなど、開発現場は統制されている状態である。一方で、タスクボードや日次ミーティングの工夫をチームごとに行うなど、創造性の発揮を尊重している。
各イテレーションは２週間に設定し、１ヶ月ごとに顧客の反応を見ることで要求の変化に対応している。

特徴的なプラクティス

- イテレーション計画ミーティング
- １日かけて丁寧に実施している。
- プランニングポーカー
- 消せるカードに書いている。見積ができずわからないときは「100」を出すようにしている。
- 日次ミーティング
- １日 2 回 10 分程度のミーティングを実施している。問題を報告・解決するリズムが開発メンバーに浸透している。
- ふりかえり
- KPT を週に一度実施している。
- オンサイト顧客
- 発注元の担当者がひんぱんに常駐している。
- ペアプログラミング
- メンバー間のスキル差を吸収している。ペアの入れ替えをするが、設計時はペアの組み合わせを固定にすることが多い。
- コーディング規約
- 統合開発環境にツールを入れて、チェックしている。
- 持続可能なペース
- 1 日 6 時間の作業を想定している。
- チーム全体が一つに
- イテレーションごとのマイルストーン目標を明確にしている。
- 人材のローテーション
- 状況によってメンバーを異動させる。

うまくいかなかったプラクティス

- 自動化された回帰テスト
- 途中から開始したが、網羅できていない。
- テスト駆動開発
- 実施はしていない。テスト仕様書を詳細なレベルで記述することで代替していると考えている。
-

契約形態との関係

請負契約である。

チーム編成・チームメンバ研修との関係

発注元が顧客（プロダクトオーナー）で、10名のチームが2チームで開発を行っている。デザインチームは、別の地点にいる。経験の違いのあるペアプログラミングを実施して、教育としている。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	Ant
CI	Jenkins
チケット管理	Redmine
その他	社内・組織向け SNS/IRC

表 37　J 社事例(16)の使用ツール

4.18. 事例(17) : J 社

エンドユーザ向けサービスを開発しているグループ会社が、プロジェクト実施の際に利用するクラウド基盤システムの Web API を提供している。システムオーナーは親会社である発注元である。

ライフサイクルモデルは、ウォーターフォール型開発にアジャイル型開発を組み込んだ。データ構造、インターフェイス設計（1.5 ヶ月・4.2 人月）→ 実現性検証/基礎開発（1 ヶ月・3.3 人月）→【本格開発（2 ヶ月・8.9 人月)】→ 総合テスト（1.5 ヶ月・2.7 人月）→ 改善対応（1 ヶ月・1.1 人月）と進め、【】内がイテレーション開発とした。短期間でのリリースを繰り返すことにより、顧客要求と乖離の少ないシステム提供を行うことができた。開発者全員が開発目的を共有することができ、ムダな作業を排除しながら進めることができた。

事例タイプ	サービス開発
会社タイプ	メーカーのシステム子会社
システムタイプ	ソフトウェアプロダクト
プロジェクト工数	20.2 人月
プロジェクト期間	6 ヶ月
プロジェクト関与者人数	1 チーム 5 人
開発言語	Java, JavaScript
開発拠点	同一拠点、同一フロア
開発手法	XP
成功度	4: かなりの部分でうまくいった

表 38 J 社事例(17) プロパティ

使用プラクティス群

コミュニケーションを強化おり、チーム内では、対面のミーティングも 1 日 3 回の日次ミーティング、イテレーションごとの計画会議とふりかえりを行っている。顧客とは週に 2 回対面ミーティングを実施し、対面以外に電話、チャット、プロジェクト管理ソフトを使い分けた。充分なコミュニケーションがとれるように心がけた。

共通する原則である「時間を節約せよ」「さぼることを考えろ」「作業場所をきれいにせよ」「時間を使わず頭を使え」を守れるような問いかけをしている。

特徴的なプラクティス

- ◆ 日次ミーティング
- ◆ 一日 3 回（朝、昼、夕)10 分程度のミーティングを実施し、問題を報告/解決するためのリズムが開発メンバー全員に浸透して短期での問題提起ができている。
- ◆ ふりかえり
- ◆ イテレーション毎にふりかえりを実施し、実施ごとに必ずトライする項目を決定して実行した。
- ◆ バーンダウンチャート
- ◆ プロジェクト管理ソフト(Redmine)を使って毎日顧客に公開している。
- ◆ タスクボード（タスクカード)
- ◆ ホワイトボードを常にメンバーが参照できる位置に設置。一定期間でホワイトボードに記述する内容/構成の見直しを行い、現在の状況が他チームにも見やすい状況を作っている。
- ◆ オンサイト顧客
- ◆ 平均週 2 回の対面での課題解決、スケジュール確認を実施した。コミュニケーションの方法として対面以外で、電話、チャット、プロジェクト管理ソフト(Redmine)を活用した。
- ◆ 自己組織化チーム
- ◆ タスクを実施する目的をメンバーが常に意識し、作業順番も考えられるよう情報共有を行った。
- ◆ テスト駆動開発
- ◆ Web API を提供するため、テスト駆動開発としてユニットテストを自動化して、イテレーションごとにテストを実施した。
- ◆ ユニットテストの自動化
- ◆ ユニットテストは完全自動化し、Jenkins から実行可能としてリリース前に毎回ユニットテストを実施している。

うまくいかなかったプラクティス

- 迅速なフィードバック
- 発注元との連携はうまくいっているが、その先のユーザーからのフィードバックがうまくいっていない。

契約形態との関係

- 請負契約である。

チーム編成・チームメンバ研修との関係

発注元が顧客（プロダクトオーナー）で、5名のチームで開発を行っている。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	Redmine
その他	ユニットテスティングツール: JUnit 社内・組織向け SNS

表 39 J 社事例(17)の使用ツール

4.19. 事例(18) : J 社

エンドユーザ向けサービスを開発しているプロジェクトが利用するクラウド基盤システムの Web API を開発している。システムオーナーは発注元、開発プロジェクトマネージャはシニアマネージャが担当している。
独自アーキテクチャを採用し、発注元と合同でプロトタイピングを実施した上で合意した。
ライフサイクルモデルは、インターフェイス設計→本格開発→結合テストをイテレーション開発している。イテレーションは 2 週間に設定し、イテレーションごとにリリースを実施している。

事例タイプ	サービス開発
会社タイプ	メーカーのシステム子会社
システムタイプ	ソフトウェアプロダクト
プロジェクト工数	-
プロジェクト期間	9 ヶ月
プロジェクト関与者人数	1 チーム 10 人
開発言語	Java, JavaScript
開発拠点	同一拠点、同一フロア
開発手法	XP
成功度	4:かなりの部分でうまくいった

表 40　J 社事例(18) プロパティ

使用プラクティス群

ユニットテストを全て自動化したうえで日次／イテレーション毎にテストを実施した。
自動テスト以外に、テストケースに留まらず、自由に探索的なテストを実施し、イレギュラーパターンでのバグ検出率の向上を図った。
継続的インテグレーションを実施。
ユニットテスト、結合テストを完全自動化して機能修正によるレベルダウンが発生しないようにし、品質管理を行った。

特徴的なプラクティス

- イテレーション
- タイムボックスは崩さないよう意識した。
- プランニングポーカー
- プロジェクト開始時点のイテレーション計画でプランニングポーカーを実施。単位は時間である。
- 日次ミーティング
- 一日 2 回（朝、夕）10 分程度のミーティングを実施。問題を報告・解決するためのリズムが開発メンバー全員に浸透し、短期での問題提起ができている。
- タスクボード（タスクカード）
- タスクボードを常にメンバーが参照できる位置に設置している。一定期間でタスクボードに記述する内容/構成の見直しを行い、現在の状況が他チームにも見やすい状況を作っている。スプリント毎にタスクボードの配置を見直す。
- バーンダウンチャート
- バーンダウンチャートを毎日顧客に公開している。
- オンサイト顧客
- 平均週 1 回の対面での課題解決、スケジュール確認を実施した。近くにいるために、口頭でコミットメントを済ませたことが、あとあとトラブルになることがあった。
- ユニットテストの自動化
- ユニットテストは書きづらいため行わず、エンドツーエンドテストで代替している。
- 継続的インテグレーション
- コミット毎と 1 日に 2 回流している。
- 自己組織化チーム
- タスクを実施する目的をメンバーが常に意識し、計画ゲームのときに作業順番も考えられるよう情報共有を行った。

うまくいかなかったプラクティス

- イテレーション計画ミーティング
- 意味のある計画を立てられるよう、発注元と一緒に計画ゲームを行ったが、QCD の考え方が合わないなどの理由で一緒に

やりづらい面もあった。

- ベロシティ計測
- 計測は実施しているが集計していない。
- 自己組織化チーム
- メンバー個人個人の特性によることが多くチーム全体としての評価が不明である。
- テスト駆動開発
- メンバーとの意識の乖離があり、実施できていない。
- シンプルデザイン
- 設計が複雑になって、テストが書きづらくなっている。

契約形態との関係

- 請負契約である。

チーム編成・チームメンバ研修との関係

発注元が顧客（プロダクトオーナー）で、10 名のチームで開発を行っている。
週 1 回プロジェクト内での勉強会を実施している。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	Maven
CI	Jenkins
チケット管理	Redmine
その他	ユニットテスティングツール: JUnit, , JerseyTestFramework 社内・組織向け SNS

表 41　J 社事例(18)の使用ツール

4.20. 事例(19) : J 社

J 社の親会社がサービスオーナーであり、親会社からの受託開発である。
アジャイル型開発を採用した理由は、これまで IT を適用したことがない業務の IT 化であったため、実際に動くものを短いサイクルで顧客提供し、要件を確認しながら進める必要があったからである。１ヶ月毎に顧客要望機能をリリースし、顧客要件を固めていった。これが顧客要件を最も確認しやすい開発方法である。

事例タイプ	サービス開発
会社タイプ	メーカーのシステム子会社
システムタイプ	ソフトウェアプロダクト
プロジェクト工数	200 人月、現在も継続
プロジェクト期間	2 年、現在も継続
プロジェクト関与者人数	1 チーム 10 人
開発言語	Flex (ActionScript), HTML5 (JavaScript), Java
開発拠点	同一拠点、同一フロア
開発手法	XP
成功度	3: 課題はあるが、うまくいっている

表 42 J 社事例(19) プロパティ

使用プラクティス群

コミュニケーションに重点を置いているため、日次ミーティングや紙・手書きツールのほか、自社で販売している社内・組織向けソーシャル・ネットワーキング・サービス(SNS)を用いている。仕様書も Redmine の Wiki に掲載し､顧客も閲覧できるようにしている。
課題がある実装について、継続的にリファクタリングを実施している。

特徴的なプラクティス

- イテレーション
- １週間だとオーバーヘッドが大きいため、2 週間イテレーションにしている。
- 日次ミーティング
- 朝会、夕会を実施している。朝会は作業予定、夕会は作業結果を話し合う。
- ふりかえり
- 週に一度、チームメンバ全員で KPT によるふりかえりを実施し課題と対応方針を決める。
- タスクボード（タスクカード）
- タスクボードを作業フロアに設置し､メンバ全員がすぐに確認できるようにしている。
- ペアプログラミング
- 技術の高いメンバと低いメンバをペアリングし、技術継承を行えるようにした。ペアは１週間で交代させ、機能実装担当が固定化しないよう注意した。
- 迅速なフィードバック
- SNS コミュニティで情報を共有している。
- チーム全体が一つに
- 現在のイテレーションの目標をホワイトボードに書き、チーム全員で認識している。

うまくいかなかったプラクティス

- ユニットテストの自動化
- テストを自動化できていない。
- 持続可能なペース
- リリース前は作業が増える傾向にある。飛び込みで作業が入る。

契約形態との関係

- 請負契約である。

チーム編成・チームメンバ研修との関係

発注元が顧客（プロダクトオーナー）で、10 名のチームで開発を行っている。
ペアプログラミングによる技術継承を行った。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	Ant
CI	Hudson
チケット管理	Redmine
その他	ユニットテスティングツール: JUnit 社内・組織向け SNS

表 43　J 社事例(19)の使用ツール

4.21. 事例(20): K 社

事例プロフィール

EC サイト及びコンテンツ管理システム(CMS)、受発注、在庫、売上、仕入管理を行うバックオフィスシステムの開発事例である。開発メンバーは 7 名。システム利用企業がプロジェクトマネジメントを担い、K 社側は、開発、保守、データセンター提供を担当した。
プロトタイプを早期に提供するも、ステークホルダーとの調整に失敗し工期後半で大量の仕様齟齬が発生した。リリース後は 2 年弱運用改修を行なっている。この運用保守フェーズにおいて、1 週間 1 スプリントのスクラムと、かんばん併用の開発を行なっている。

事例タイプ	サービス開発
会社タイプ	中小 SIer・ソフトハウス
システムタイプ	EC サイト及びバックオフィスシステム
プロジェクト工数	初期構築: 約 80 人月 運用改修: 約 200 人月
プロジェクト期間	初期構築: 10 ヵ月 運用改修: 2 年
プロジェクト関与者人数	専任 7 名 非専任 2～4 名
開発言語	PHP
開発拠点	開発チームと顧客は別拠点
開発手法	Scrum
成功度	成功はアジャイルコーチと理解ある顧客によるところが大きい。逆に、チームの現状を把握しないままプロジェクトに入り、開発をチーム任せにしていたところが失敗点だった。

表 44　K 社事例(20) プロパティ

使用プラクティス群

全般的によくプラクティスを利用しており、特にスクラムのプラクティスを活用している。長期にわたる運用保守開発を行うため、チームを固定して人材ローテションは適用していない。また、設計開発系のプラクティスでは、ペアプログラミングや全コードのユニットテストなどは行っていない。

特徴的なプラクティス

- イテレーション計画ミーティング
- スクラムのプラクティスの中でもっとも時間を割いている。
- プランニングポーカー
- ストーリーや機能はタスク分割を行いながら、相対時間の見積りをプランニングポーカーで実施している。
- 日次ミーティング
- 朝会の他、昼会と夕会も毎日実施している。
- かんばん
- 運用案件はかんばん無しに回すことができない。お客様とかんばんを共有している。
- リファクタリング
- お客様に開発期間の 15%を充てるように言われている。
- アジャイルコーチ
- スクラムのコーチと、テストのコーチを両名招聘して支援してもらっていた。

うまくいかなかったプラクティス

- ニコニコカレンダー
  すぐに使わなくなった。使わなくても見ていれば状態はわかった。メンバーも空気読まずに言う人ばかりであった。

契約形態との関係

一括請負契約。K 社は受注先。

チーム編成・チームメンバ研修との関係

チームは、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できるメンバーと、手順内で自由裁量の部分を遂行できるメンバーで構成されていた。
開発期間中、外部のコーチを 1 週間に最大 4 日という頻度で招聘し、コーチングにあたってもらった。

使用ツール

構成管理	-
IDE / エディタ	-
ビルド	-
CI	Jenkins
チケット管理	Redmine
その他	自動テストツール: Selenium

表 45　K 社事例(20)の使用ツール

4.22. 事例(21): L 社

事例プロフィール

就職情報支援システムの開発事例である。開発メンバーはピーク時に 18 名。プロダクトのオーナーは顧客システム部門の開発担当者が担い、各ユーザー業務部よりユーザー代表者を選出した。L 社にて要件定義～移行導入まで実施。
アーキテクチャチームが先行して、既存フレームワークをもとに非機能要求を実装。その後もアーキテクチャチームは継続して、アプリケーションチームからの要望をかんばんにて管理し、改善要望に対応した。
プロジェクト全体の開発プロセスは、UP をベースとしており、作成フェーズをスクラムによる反復型開発というハイブリット型のプロセスを取っている。具体的には、最初の 2 ヵ月を推敲フェーズとして、その後の作成フェーズを反復型開発で行った。5 ヵ月の作成フェーズを経て、移行フェーズを 2 ヵ月かけて実施した。

事例タイプ	社内システム開発
会社タイプ	大手 SIer
システムタイプ	就職情報支援システム
プロジェクト工数	96 人月
プロジェクト期間	9 ヵ月
プロジェクト関与者人数	10～18 人
開発言語	Java
開発拠点	顧客とは別サイトで開発。開発チームは自社の専用プロジェクトルームに全員着席。
開発手法	Scrum+UP
成功度	期待を上回る結果。短納期を実現させた高い生産性（JUAS の標準工期を 30%短縮）。ウォーターフォールでは成し遂げられない漸進的なシステム改善。

表 46　L 社事例(21) プロパティ

使用プラクティス群

プロセス系のプラクティスの活用が多い。一方で人に関するプラクティスや、設計開発プラクティスでテスト駆動開発など、利用していないプラクティスが散見される。ただし、ペアプログラミング、ユニットテストの自動化、全コードのユニットテスト、リファクタリング、継続的インテグレーションなどエンジニアリングの要となるプラクティスの活用度は高い。

特徴的なプラクティス

- 日次ミーティング
- 朝にチーム単位のミーティング、昼に全チームでミーティングを実施。問題の早期発見、解決方法の素早い横展開に非常に有効。
- バーンダウンチャート
- 毎日、各チームのバーンダウンチャートをプロジェクトルームに貼り付ける。進捗の見える化、即時共有で、問題の早期発見にも効果を発揮。
- スプリントレビュー
- スプリント毎にデモンストレーションを中心にしたレビューを実施。ユーザーが実際のシステムを見ることで、改善要望を 100 個近く拾い上げることができた。
- 自己組織化チーム
- 日次ミーティングで共有した課題に対し、タスクフォース的な対策チームを組むなど、自発的に得意不得意を補填し合っていた。

うまくいかなかったプラクティス

- 特になし。

契約形態との関係

◆　要件定義～移行導入まで一括請負契約。
L 社は受注先（一次請け）。

チーム編成・チームメンバ研修との関係

チームは、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できるメンバーで大半が構成されていた。
主要メンバーが PostgreSQL の社外講習、Seasar2 の社内講習を受講。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	Ant
CI	-
チケット管理	-
その他	ユニットテスティングツール: S2Unit 開発フレームワーク: Seasar2+自社フレームワーク

表 47　L 社事例(21)の使用ツール

4.23. 事例(22): L 社

事例プロフィール

顧客が全社で利用する基幹システムの構築プロジェクトの事例である。1次フェーズの完了まで2年を要し、その後も開発は継続している。開発メンバーは3チームで総勢60名。システムオーナーはお客様親会社の経営委員会。プロジェクトマネジメントは、親会社、システム子会社よりそれぞれ選出し、2人体制でプロジェクトの運営に当たっている。
6ヶ月を一つの期とし、期ごとの目標を決めてプロジェクトを運営。最後の期はシステムテスト、ユーザーテスト、リリース準備のみを実施した。

事例タイプ	社内システム開発／大規模プロジェクト
会社タイプ	大手 SIer
システムタイプ	基幹システム（営業フロント、取引・資産、請求・入金、財務など）
プロジェクト工数	-
プロジェクト期間	1次フェーズまで2年。その後も継続中。
プロジェクト関与者人数	400名。
開発言語	Java
開発拠点	プロジェクト全体が同一拠点内で作業している。
開発手法	Scrum+WF
成功度	上手くいったサブシステムと、上手くいかなかったサブシステムがある。

表 48　L 社事例(22) プロパティ

使用プラクティス群

大規模なプロジェクトでのスクラム適用。プロセス系のプラクティスの利用がある一方、設計開発系に関しては、あまり活用には至ってない。
オンサイト顧客、自己組織化チーム、ニコニコカレンダーなど、人に関するプラクティスを比較的活用しているのは、多くの開発者が関わるプロジェクトならではの傾向といえる。

特徴的なプラクティス

- プランニングポーカー
- メンバー全員でプランニングポーカーを実施した。メンバー間の知識が共有されるため、メンバー間のスキルの差が序所に埋まっていった。
- ふりかえり
- 実施することが目的とならないように、司会を持ちまわり、チーム編成を変更するなど工夫を実施した。
- 日次ミーティング
- 毎日、朝会を実施。短時間で終了させたいため、課題の共有までを行う。詳細な議論は別で実施する。
- かんばん
- かんばんに記入するタスクの粒度を1日で終了する程度の大きさに統一。
- オンサイト顧客
- 顧客と同じスペースにて作業を実施した。
- 自己組織化チーム
- 個々のプラクティスにおいて、参加メンバーへ権限委譲することによって自主性を引き出す。
- 人材ローテーション
- 目先の効率よりも、チーム力の向上を目的に、ローテーションを実施。最初にこの目的を明示することで、一時的に効率が落ちることをメンバー、プロダクトオーナーに納得してもらいやすくなった。
- リファクタリング
- 毎日の作業の中で、強制的にリファクタリングの時間を取った。

うまくいかなかったプラクティス

- 特になし。

契約形態との関係

◆ 準委任契約。L 社は受注先。

チーム編成・チームメンバ研修との関係

特になし。

使用ツール

構成管理	-
IDE / エディタ	-
ビルド	Maven
CI	Jenkins
チケット管理	-
その他	-

表 49　L 社事例(22)の使用ツール

4.24. 事例(23): L 社

事例プロフィール

クラウドサービス連携基盤フレームワークを研究開発した事例である。月単位で研究開発依頼先とミーティングを実施して、進捗を報告した。
研究開発のため不確定要素が多く、プロジェクトの初期では明確的なゴールが見えなかった。対策として、アジャイル型開発で検証しながらゴールを明確にするプロセスを取った。このため、発注側とはウォーターフォール型の開発を行っているものとし、内部では反復型のアジャイル型開発を行うというねじれた形での開発プロジェクトとなっているのは特徴的である。
アジャイル型開発は、プロダクトオーナーがチームの一員になり、同席することでコミュニケーションを向上すること、また個人のスキルをチームの力でカバーすることが、成功のキーポイントと評価している。

事例タイプ	技術評価
会社タイプ	大手 SIer
システムタイプ	クラウドサービス連携基盤フレームワーク
プロジェクト工数	40 人月
プロジェクト期間	8 ヵ月
プロジェクト関与者人数	-
開発言語	Java
開発拠点	同一拠点内
開発手法	Scrum+WF
成功度	-

表 50　L 社事例(23) プロパティ

使用プラクティス群

対象プラクティスを、ほぼ全般に渡って利用している。他の事例では、プロセス系に偏り、設計開発系のプラクティスの活用がまだこれからという状況が多い中、本事例の活用プラクティスの多さは際立っている。部分適用も含むが、テスト駆動開発をはじめとしたテスト系のプラクティスを実施しているのは特徴的である。

特徴的なプラクティス

独自の工夫は見受けられないが、数多くのプラクティスを活用している。

うまくいかなかったプラクティス

- 特になし。

契約形態との関係

パートナー企業と一括請負契約。

チーム編成・チームメンバ研修との関係

チームメンバーの多くは、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可能であった。メンバー全員がスクラム入門のトレーニングを受けた。また、チームにアジャイルコーチを配置した。

使用ツール

構成管理	Subversion
IDE / エディタ	Eclipse
ビルド	-
CI	Jenkins
チケット管理	Redmine

その他	ユニットテスティング ツール: JUnit

表 51　L 社事例(23)の使用ツール

4.25. 事例(24): L 社

事例プロフィール

社内プロダクトの UML モデリングツール（パッケージ製品)の開発事例である。本プロジェクトでは、アーキテクチャを重視し、オブジェクト指向での設計を採用していた。アジャイル型開発を導入したのは、曖昧な仕様、品質、またチームのモチベーションの向上を期待したためである。

プロダクトオーナーが国内、開発チームがオフショアという体制であったため、プロダクトオーナーの開発への関与度合いを高め、開発上の課題に一緒になって取り組んだ。

事例タイプ	自社プロダクト開発
会社タイプ	大手 SIer
システムタイプ	UML モデリングツール
プロジェクト工数	30 人月
プロジェクト期間	3 ヵ月
プロジェクト関与者人数	12 名
開発言語	C#
開発拠点	プロダクトオーナーは大阪、開発チームは中国上海。
開発手法	Scrum
成功度	プロダクトオーナーの協力とチームの努力は重要だが、長い時間をかけて築いた信頼関係は成功の大きな要因であった。

表 52　L 社事例(24) プロパティ

使用プラクティス群

事例(22)同様に、対象プラクティスをほぼ全般に渡って利用している。開発はスクラムで遂行しており、プランニングポーカーやかんばんなどの活用はなかった。設計開発系のプラクティスの利用が多いことも特徴的である。

特徴的なプラクティス

- ユニットテストの自動化
- プロジェクトの途中からユニットテストを作成し始めた。一度に全てを作成するのは難しかったため、変更の部分からテストケースの追加を行った。
- スパイク・ソリューション
- 実現の可能性や導入の可能性について、事前調査のフェーズを設けた

うまくいかなかったプラクティス

- 特になし。

契約形態との関係

- パートナー企業と一括請負契約。

チーム編成・チームメンバ研修との関係

チームメンバーの多くは、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できるメンバーであった。チーム全員がスクラム入門・実践、TDD 入門のトレーニングを受けた。また、チームにアジャイルコーチを配置した。

使用ツール

構成管理	-
IDE / エディタ	商用 IDE
ビルド	-
CI	Jenkins
チケット管理	商用統合開発管理ツール

その他	ユニットテスティングツール: NUnit

表 53　L 社事例(24)の使用ツール

4.26. 事例(25) : M 社

事例プロフィール

ソーシャル・ネットワーキング・サービス(SNS)を自社サービスで提供している。
組織横断の方法が必要だったのでスクラムを用いている。採用に置いては、アジャイル型開発を全面に採用するというよりも、会議体や責任範囲の明確化のために、プラクティスレベルでアジャイルプラクティスを採用していた。
ビジネスのミッションで組織を分割している。プラットフォーム開発のミッションと重要業績評価指標(KPI)は、ユーザがアクティブであるかどうかである。

事例タイプ	自社サービス開発／大規模（組織展開）
会社タイプ	サービス事業者
システムタイプ	ソーシャルゲーム
プロジェクト工数	-
プロジェクト期間	-
プロジェクト関与者人数	200 人
開発言語	Perl
開発拠点	同一拠点
開発手法	Scrum
成功度	5:うまくいった

表 54　M 社事例(25) プロパティ

使用プラクティス群

スプリントは 2 週間に設定している。プラットフォーム開発と運用は小さな機能改善が多いので、スプリントの中でいくつもの案件をこなしている。大きなユーザ向けサービスは、規模も大きく複数のスプリントを分けている。

当事例は、企画フェーズ、開発フェーズ、保守運用フェーズ、それぞれの状況にあわせたプロセスを組み合わせる例があった。企画フェーズでは見積りが困難なため、かんばんを用いる。方針が固まり見通しができると、開発フェーズではイテレーション計画ゲームを実施するなど、スクラムで進める。保守運用フェーズになると、かんばんで変更要求を受け付ける。

特徴的なプラクティス

- ふりかえり
- KPT は、全員の意見が出てこないため、『アジャイルレトロスペクティブズ』で紹介されている 555（トリプルニッケル）という手法を用いている。[ダービー et al. 2007]
- かんばん
- スクラムは、プロダクトの開発初期のように計画を中心にして変化を許容するフェーズには向いているが、そもそも見積や計画できない業務については、かんばんでやっている。
- 柔軟なプロセス
- スクラムの規範を超えてしまうようなプロセスの変更は認めないものの、かなり柔軟なプロセスとしている。
- スプリントバックログ
- スプリントバックログは、チケット管理システムを止めてアナログでホワイトボードに貼っている。
- チケット管理システムを使っていない理由としては、1) ユーザーストーリーやタスクの優先順位を変更するのに手間がかかる、2) タスクに分解した状態だと一画面に入らないため、朝会の非効率さと、チームに対する非効率さがあった、3) スクラムマスターや外にいる人が理解しづらいなどの理由があげられた。
- 顧客プロキシ
- プロダクトオーナーがエンドユーザーの代表者にインタビューを行うことがある。
- プロダクトオーナー
- プロダクトオーナーのみがプロダクト（サービス）を考えているのではなく、開発スタッフもまた同様である。
- ファシリテータ（スクラムマスター）
- 適性を持つ専任のスクラムマスターに

し、複数のチームを持つ。開発者が兼任すると、1）近視眼的になるので改善が進まない、 2）リーダーとして振る舞ってしまう、というマイナス面がある。
- 自己組織化チーム
- 属人性の強い人がいるチームの方が自己組織化はできる。
- スパイク・ソリューション
- 最初の数スプリントを技術検証フェーズとしている。
- リファクタリング
- 特定のスプリントで集中的にリファクタリングしている。
- 集団によるオーナーシップ
- プロジェクト内外に公開したソースコードに対して、コメントや修正案を受けつけるオープンな開発方式を実施している。２人以上のレビューがないとマージできない。
- 共通の部屋
- オープンスペースがある。
- ミーティングの際は、会議室に可動式のホワイトボードを持ち込むようにしている。

うまくいかなかったプラクティス

- バーンダウンチャート
- 手間がかかるため、抵抗されることが多い。
- 人材のローテーション
- 比較的ひんぱんにやっている。属人性の強いチームのローテーションは、技術的な傾向もあり難しいことも多い。
- 持続可能なペース
- 自社サービスであるため、直接自社の利益や評価に直結することから、無理をして仕事を進めることが多い。

契約形態との関係

自社開発のため開発についての契約は存在しない。

チーム編成・チームメンバ研修との関係

1 チームが 5~6 名で構成されており、その中に企画者や開発者が含まれている。

使用ツール

構成管理	-
IDE / エディタ	-
ビルド	-
CI	-
チケット管理	商用チケット管理ツール
その他	-

表 55　M 社事例(25)の使用ツール

5. 活用のポイント

本章では、事例調査を元に、以下の９つの特性を持つプロジェクトに有効なプラクティスの活用について述べる。

読者の現場の特性と比較して、プラクティスの適用のヒントにされたい。

1. 短納期、開発期間が短い
2. スコープの変動が激しい
3. 求められる品質が高い
4. コスト要求が厳しい
5. チームメンバーのスキルが未成熟
6. チームにとって初めての技術領域や業務知識を扱う
7. 初めてチームを組むメンバーが多い
8. オフショアなど分散開発を行う
9. 初めてアジャイル型開発に取り組む

5.1. 短納期、開発期間が短い

開発対象のボリュームに比して開発期間が短い場合、チームの開発速度を計測し、そのスピード感で、予定している開発量が期限内に完了するのか、常に点検する必要がある。

期日までに完了しない可能性が高まった場合、ステークホルダーとその後の打ち手について検討を始めなければならない。

活用するプラクティスは、ベロシティ計測と、バーンダウンチャートが挙げられる。ベロシティについては、関係者であるプロダクトオーナーが理解できる基準で計測する必要がある（H 社事例（11））。また、バーンダウンチャートは、関係者と定期的に共有する機会を設けることが活用のポイントである（B 社事例(2)、J 社事例(17)(18)）。

5.2. スコープの変動が激しい

開発中に要求の変更が頻繁に発生するプロジェクトでは、チームが扱う要求が全体として何があり、現在のイテレーションでどの要求を開発することになっているか、要求の状態が管理でき、柔軟に優先順位を変えられるプラクティスが求められる。

この場合、活用するプラクティスは、プロダクトに関する要求全体を管理するプロダクトバックログと、現在のイテレーションで開発対象となる要求を管理するスプリントバックログである。

プロダクトバックログは、イテレーション毎に整理を行い、チーム全員で優先順位と内容を合意すると良い（B 社事例(2)）。なお、プロダクトバックログの優先順位に責任を持つ、プロダクトオーナーを設置することも、プロダクトバックログとセットで活用するべきプラクティスである。

プロダクトオーナーは特定の領域の専門知識を有したエキスパートが務めることがあるが、重要なのは、適切な要求の優先順位付けである。プロダクトオーナーには業務や全社的に全体最適となる判断が求められる（G 社事例(10)）。

5.3. 求められる品質が高い

品質要求が高いプロジェクトでは、品質を保ち続けるプラクティスを活用するべきである。テストに関するプラクティス、具体的には、自動化された回帰テスト、ユニットテストの自動化などである。

自動化に関しては、プロジェクトの初期段階で、実施の有無、実施のための取り決め、また使用ツールを検討しておく必要がある。これを後回しにすると、必ず機能開発が優先され、自動化にたどりつかない（B 社事例(2)）。なお、自動化の仕組みに関しては、継続的インテグレーションが用いられることが多い。プロダクトコードの変更時に、既存の機能に影響を与えていないか、フィードバックを得る手段として活用したい。

5.4. コスト要求が厳しい

必要のないものを作るムダをなくし、必要なものをより素早く提供することが ROI（費用対効果）の向上につながり、コスト要求に応えることができる。そのためには、的確に顧客の要求を把握し、認識の相違をなくす必要があるため、プロダクトバックログ（優先順位付け）を活用する。

また、顧客とのコミュニケーションを密にすることで認識の相違を無くすことを目的にオンサイト顧客を利用する。

さらに、要求に対する顧客の意図を的確に把握、開発機能が意図通りになっているかの検証のために、受入テストを活用することができる。オンサイト顧客には、優先順位や仕様の確認をその場で行い、迅速に方針を決められるというメリットがある（K 社事例(20)）。

万が一、顧客とチームの間で認識がズレていた場合、方向をすぐに修正する必要がある。迅速なフィードバックを活用し、フィードバックを得る機会を設けておくと効果的である（C 社事例(4)、L 社事例(21)）。

具体的には、イテレーションの終了時に、完了したものを関係者にデモをする、スプリントレビューがあげられる。システムの利用者に実際に見て、動かしてもらうことで、多くのフィードバックが得られる（L 社事例(21)）。

5.5. チームメンバーのスキルが未成熟

常に技術的に熟達したメンバーでチームが構成できるわけではない。むしろ、スキル的に未成熟なメンバーが成長していく機会として、プロジェクトを計画するべきである。

イテレーション計画ミーティングで、全員の知恵を絞って作業レベルの計画を作っていくことで、作業の段取りや考慮すべきポイントを学んでいくことができる。

ペアプログラミングは、ベテランとメンバーが一緒に仕事をすることで、技術的な指導を行うのに適したプラクティスである（C 社事例(4)）。

また、実施した作業や行動から、次の改善点を考えるふりかえりも、メンバーの成長の機会として捉えることができる。ふりかえりについては、ふりかえりのやり方自体も見直しながらチームに適したやり方を模索すると良い（E 社事例(6)）。

チームがアジャイル型開発の技術スキルに未熟な場合は、アジャイルコーチにトレーニングしてもらう、アドバイスをうけるのがよい(B 社事例(2),(3)、G 社事例(9)、M 社事例(25))。

5.6. チームにとって初めての技術領域や業務知識を扱う

技術の進化が速いシステム開発では、プロジェクトを始める際に、必ずしもチームがプロダクトに必要な技術領域の経験を保有しているわけではない。また、プロダクトの背景にある業界の知識や、要求の理解と実装に必要な業務知識の獲得は、必ずといって良いほど課題となる。

これらに対し、活用できるプラクティスとして、スパイク・ソリューションとシステムメタファを挙げることができる。スパイク・ソリューションを適用することは、リスクとなりそうな技術課題について、プロジェクトの初期段階で実験的に小さく試しておくことであり、チームとプロジェクトを後々助けることに繋がる（C 社事例(4)）。また、システムメタファは、開発者にとって、なじみの薄い業務知識を理解する手段として有効と考えられる。顧客が話す言葉については、ユーザーストーリーを開発するために行う会話の中で、逐次確認と理解をしながら進めていくことが望ましい。

5.7. 初めてチームを組むメンバーが多い

初めてチームを組むメンバーが多い場合、お互いのことが充分に理解できていない状態からプロジェクトを始めることとなり、チームがばらばらになりやすい。チームが向かう方向を明確にすることと、チームビルディングのためのプラクティスの活用を検討するべきである。

チームのミッションを明らかにするプラクティスとして、インセプションデッキが挙げられる。インセプションデッキの作成を通じて、プロジェクトの目的や目標が明らかとなる（B 社事例(1)）。チーム全体が一つになるための工夫としては、共通の部屋に、全員が同席することも挙げられる。

また、一つの作業をペアで行う、ペアプログラミングも有効と考えられる。また、チームメンバーの状況を把握しづらいようであれば、ニコニコカレンダーをチームに持ち込むことで、メンバーの感情や状況を見える化する工夫を取ると良い（E 社事例(6)）。

5.8. オフショアなど分散開発を行う

プロダクトオーナーと開発チームが別の拠点でそれぞれの作業を行う場合、その間のコミュニケーションは課題となりやすい。認識の相違が見当違いな開発を生み出し、プロジェクトに与える影響は大きなものとなる。このような状況下では、オンラインでのコミュニケーション手段を検討し、頻繁にコミュニケーションが取れるように準備するべきである。

日次ミーティングは、離れた者同士が毎日顔を合わせる機会として、ぜひ活用するべきである（G 社事例(9)）。また、顧客と開発チームのやりとりがどうしても限られてしまう場合は、顧客に成り代わる顧客プロキシの設置を検討すると良い。分散した環境下でも、迅速なフィードバックが得られる工夫を取らなければならない。

5.9. 初めてアジャイル型開発に取り組む

事例調査では、初めてアジャイル型開発に取り組むケースが数多く見受けられた。

社内にアジャイル型開発に取り組んだ経験のある人がいる場合はその人に、いない場合は社外からアジャイルコーチを頼んで導入の手伝いをしてもらうのがよい。初めて取り組む場合は、イテレーション期間を短かくした上で、ふりかえりの中で改善点をチームで考え実行していくことが不可欠となる。

それぞれの事例でどのようなプラクティスに取り組んでいたかは、ガイド編の活用事例を参照されたい。

初めてプラクティスを適用する際には、本書を参考として、どのような問題に対し、何を狙いとして活用するのか充分に検討されたい。組織にあわせたアジャイルスタイルというプラクティスが示すとおり、どのようなコンテキストでも通用する万能のプラクティスは存在しない。必ず、各自の現場の状況に応じて、工夫するよう注意をして欲しい。

6. 参考文献

[ベック 2000]
ベック, B. 長瀬嘉秀監訳（2000）『XP エクストリーム・プログラミング入門 ソフトウェア開発の究極の手法』 ピアソン・エデュケーション

[ベック et al 2005]
ベック, B. アンドレアス,S. 長瀬嘉秀監訳（2005）『XP エクストリーム・プログラミング入門 変化を受け入れる第 2 版』 ピアソン・エデュケーション

[ジェームス 2009]
ジェームス,S. 木下史彦・平鍋健児監訳（2009）『アート・オブ・アジャイルデベロップメント 組織を成功に導くエクストリームプログラミング』 オライリージャパン

[クリスピン et al 2009]
クリスピン,L. ジャネット.G. 榊原彰監訳（2009）『実践アジャイルテスト テスターとアジャイルチームのための実践ガイド』 翔泳社

[Elssamadisy 2008]
Elssamadisy, A. (2008). Agile Adoption Patterns: A Roadmap to Organizational Success, Addison-Wesley

[シュエイバー et al 2003]
Schwaber,K. Beedle,M. 長瀬嘉秀・今野睦監訳（2003）『アジャイルソフトウェア開発スクラム』 ピアソン・エデュケーション

[Schwaber et al 2011]
Schwaber,K. Sutherland,J. 角征典訳（2011）「スクラムガイド スクラム完全ガイド : ゲームのルール」
http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum%20Guide%20-%20JA.pdf

[コーン 2009]
コーン,M. 安井力・角谷信太郎訳（2009）『アジャイルな見積りと計画づくり』 毎日コミュニケーションズ

[ラスマッソン 2011]
ラスマッソン,J. 西村直人・角谷信太郎監訳（2011）『アジャイルサムライ 達人開発者への道』 オーム社

[ピヒラー 2012]
ピヒラー,R. 江端一将訳（2012）『スクラムを活用したアジャイルなプロダクト管理』 ピアソン

[ダービー et al 2007]
ダービー,E. ラーセン,D. 角征典訳（2007）『アジャイルレトロスペクティブズ 強いチームを育てる「ふりかえり」の手引き』 オーム社

[Meszaros et al 1996]
Gerard Meszaros, Jim Doble (1996) 「A Pattern Language for Pattern Writing」
http://hillside.net/index.php/a-pattern-language-for-pattern-writing

[IPA 2012]
独立行政法人情報処理推進機構（2012）『非ウォーターフォール型開発の 普及要因と適用領域の拡大に関する調査 国内の中規模及び大規模開発プロジェクトへの適用事例調査 調査報告書』独立行政法人情報処理推進機構 http://sec.ipa.go.jp/reports/20120328/20120328_1.pdf

7. 索引

本報告書で取りあげたプラクティスとプラクティスの別名の索引を示す。

C

K

Y

あ

い

う

え

お

か

き

け

こ

し

す

せ

そ

た

ち

て

な

に

は

ふ

へ

ほ

ま

ゆ

り

れ

付録1. 発見された工夫と課題

組織や扱うサービス、製品によって、アジャイル型開発の状況は異なる。また、たとえ同じ開発チームであっても、ビジネスの企画段階であるか、もしくはリリース後の保守運用段階であるかなど、フェーズによっても異なってくる。アジャイル型開発のプラクティスは、状況に合わせて柔軟に変更しながら活用していくことが重要である。

今回の調査では、主に一般に確立されているプラクティスを対象とした。また、調査対象のプラクティスの適用状況だけではなく、
a) 関係者の状況に合わせてプラクティスを独自に修正しているか
b) 調査対象以外のプラクティスを活用しているか
の二点についてもヒアリングした。その結果、プラクティスとして完成には至らなかったり、既存のプラクティスに類似していたりする例も中には見られたが、数多くの工夫や課題に関する情報が得られた。

ここでは、アジャイル型開発の実践を通して、その振り返りの中から独自に発展させ、あるいは新たに作り出しながらプラクティスを適用していくヒントとなることを期待し、それらを付録として掲載する。3 事例以上で見つけられた工夫については、3.4 に記載している。

なお、本付録では、以下の点に注意していただきたい。
・空欄のある不完全なプラクティスが含まれている。
・それぞれの工夫や課題には類似したプラクティスが含まれている。
・解決策の記載がないプラクティスは課題である。下線を引いた名前で示した。

1.1 プロセス・プロダクト

1.1.1 イテレーション

名前	粗い見積
状況	計画する上での大体の期間、規模が知りたい。スコープや時間が制御可能である。
問題	見積に時間がかかり過ぎる。
フォース	細かく見積しても、精度が一定以上は上がらない。
解決策	見積はストーリーに対して規模をざっくり粗く行う。
結果	リリース時期を見極めるだけの見積が得られる。
課題	
事例	1

1.1.2 イテレーション計画ミーティング

名前	見積のためのタスク分割
状況	イテレーション計画でタスクを見積っている。ストーリーは見積ってはいない。 プランニングポーカーは使っていない。
問題	タスクはどのくらいの粒度で見積れがよいのか？
フォース	継続して一つのタスクを実施する必要がある。
解決策	タスクの粒度を 2〜5 時間の粒度と制約をかけた。
結果	詳細な単位で、粒度が均一化してタスクを分解して見積ることができた。
課題	一つのタスクを無理矢理分割するため、「完了」の定義に影響を及ぼすことがある。
事例	4

名前	スプリント計画に時間がかかる
状況	スプリントをはじめる準備が足りていない。
問題	イテレーション計画ミーティングに時間がかかりすぎる。

フォース	
解決策	なし
結果	
課題	
事例	14

名前	ペーパープロトタイピング
状況	スプリント計画ミーティングで、プロダクトオーナーが仕様をチームに説明している。
問題	ミーティングに時間がかかってしまう。
フォース	プロダクトオーナーは事前に説明資料を作っておく時間がない。 開発者は詳細なタスクを見積るために、具体的な仕様が欲しい。
解決策	スプリント計画ミーティング時に付箋やコピー用紙でペーパープロトタイプを作り、開発者に仕様を説明し共有した。
結果	ミーティングの時間が短縮化できた。
課題	
事例	9

名前	イテレーション計画に時間がかかる
状況	スプリントをはじめる準備をしている。
問題	イテレーション計画ミーティングに時間がかかりすぎる。
フォース	
解決策	なし
結果	
課題	
事例	

1.1.3 イテレーション

名前	ポモドーロ・タイマーで集中
状況	スプリントの中で効率時間を使いたい。
問題	すぐに予定している時間を超過してしまう。
フォース	時間を決めてはいるが、意識できていない。
解決策	各自の作業を、パスタの茹で時間を計るポモドーロ・タイマーを用いて 25 分間集中して作業をし、タイムマネジメントするようにした。
結果	時間を厳密に意識するようになった。
課題	個々で時間を計測していたので、他のメンバーが割り込みづらかった。
事例	9

名前	メンバーを表彰
状況	イテレーションが終わった。ふりかえりを実施しようとしている。
問題	皆の頑張りに対して報いたい。
フォース	
解決策	イテレーション中で頑張った人を表彰する。
結果	
課題	
事例	6

名前	スプリント#0 不足
状況	開発チームは、開発環境が整うより前にスプリントをスタートしている。
問題	スプリントが始まっているのに、開発環境が整備されていない。
フォース	スタートと同時に開発をスタートさせたいが、環境を揃える工数を事前に捻出できていない。
解決策	なし
結果	
課題	本来はスプリント#0 を設けなければいけない。
事例	10

1.1.4 ベロシティ計測

名前	上がらないベロシティ
状況	はじめてのアジャイル型開発である。開発経験が少ないメンバーが半分であった。
問題	チームのベロシティがでなかった。

フォース	時間が限られている。
解決策	なし
結果	
課題	
事例	9

1.1.5 日次ミーティング

名前	スマートフォンで日次ミーティング
状況	全開発メンバーが同一拠点で開発を行っていたが、メンバーの１人が自社に戻ることになり同一拠点での開発ができなくなった。
問題	離れた拠点間でも、これまでのように日次ミーティングを実施したい。
フォース	無償の電話会議システムを使いたいがセキュリティ上利用できない。許可された市販ビデオ会議システムを使いたいが、予約が柔軟にできない。
解決策	スマートフォンのスピーカーフォンを利用て、相手に話しが聞こえやすいように工夫した。
結果	分散拠点間でも朝会が実現できるようになった。
課題	
事例	9

名前	フォーマルなテレビ会議
状況	全開発メンバーが同一拠点で開発を行っていたが、メンバーの１人が自社に戻ることになり同一拠点での開発ができなくなった。
問題	離れた拠点間でも、これまでのように日次ミーティングを実施したい。
フォース	無償の電話会議システムを使いたいがセキュリティ上利用できない。
解決策	テレビ会議システムを用いて、同じ PC の画面やお互いの表情を見ながらミーティングが実施できるようになった。
結果	分散拠点間でも朝会が実現できるようになった。
課題	システムの利用に事前予約が必要であり、柔軟性に欠ける。
事例	9

名前	段階的日次ミーティング
状況	複数チームが連携しているプロジェクトである。 朝会は個別チーム毎に行っている。
問題	複数チームの情報をどう共有させればよいだろうか。
フォース	チーム横断で共有するべき情報がある。
解決策	チーム毎の朝会の前に、チーム代表者が集まる朝会を実施する。全体周知の情報はそこで伝える。
結果	全体で共有する情報を、各チームに伝達できるようになった。
課題	
事例	4

名前	チーム横断日次ミーティング
状況	様々なチームがそれぞれサービスを開発している。
問題	横断的に発生している問題やノウハウを共有したい。
フォース	全員が集まってミーティングすると、時間がかかってしまう。
解決策	毎日夕方に、エンジニアが集って各チームの状況を共有する。
結果	様々なアドバイスを仰げる。問題を共有できる。
課題	
事例	1

名前	夕方に日次ミーティング
状況	フレックスタイム制で個人ごとに出勤時間が異なる。エンジニアは朝遅い人が多い。
問題	全員集まらないと日次ミーティングの意味がない。
フォース	朝に集まることは、各々の勤務条件によって現実的ではない。
解決策	朝ではなく夕方に日次ミーティングを実施する。
結果	日次ミーティングに参加しやすくなった。
課題	
事例	1

名前	分散拠点の日次ミーティング
状況	開発はオフショア先に出している。プロダクトオーナーはシステム開発以外の日常業務に追われている。
問題	拠点が違うので容易に日次ミーティングができない。
フォース	
解決策	夕方に拠点内ミーティング、翌日にプロダクトオーナー、ブリッジ、オフショアでの合同ミーティングを実施。
結果	
課題	
事例	13

名前	時間がかかる日次ミーティング
状況	関与する人間が多い（20 名）。
問題	時間が長くなりすぎる（60 分）。
フォース	
解決策	なし
結果	
課題	
事例	1

1.1.6 ふりかえり

名前	混成チームでふりかえり
状況	開発チームでふりかえりを実施していた。ふりかえりの初心者が多かった。プロジェクトは複数チームで構成されている。
問題	他チームへの悪口が増えてしまい改善が行われない。
フォース	自分達で自身の仕事の進め方を改善するという、ふりかえりの趣旨が理解されていない。
解決策	参加メンバーの一部を他チームとメンバーと入れ替えたり、複数チーム合同でふりかえりをやることにした。
結果	手本となるチームの振る舞いや行動を学び、改善が行われるようになった。
課題	
事例	4

名前	インフォーマルなふりかえり
状況	ふりかえりを実施しても、なかなか本音で話すことは困難である。
問題	フォーマルな場面だと、なかなか本音を話せない。
フォース	緊張している雰囲気の中では話しにくい。
解決策	定期的にインフォーマルに集まれる場所を用意し、ざっくばらんにビジネス状況の話をしてふりかえる。
結果	
課題	
事例	1

名前	褒めるふりかえり
状況	ふりかえりをおこなっている。
問題	発言者に対し批判的な意見を言う人がいると、自分の発言も批判されるのではないかと思い、様々な意見が言いにくく、発言しにくい雰囲気になってしまう。
フォース	
解決策	ふりかえりの中でメンバー同士を褒め合った。
結果	
課題	
事例	7

名前	定着しないふりかえり
状況	各チームにふりかえりをルールとして提示している。
問題	ふりかえりが定着しない。
フォース	他のチームからふりかえりのメリットを聞いているが実施していない。
解決策	なし
結果	
課題	ふりかえりが定着しない理由は不明である。
事例	15

名前	着手できない大きな課題
状況	ふりかえりで課題について考えている。
問題	複数チームに関わる問題や、品質などの大きな問題には着手できなかった。
フォース	
解決策	なし
結果	
課題	
事例	11

1.1.7 かんばん

名前	スクラムとかんばん
状況	スクラム開発を行っている。
問題	イテレーション計画ミーティングで計画することが困難なほど、タスクの内容が変わってしまう。
フォース	プロダクトオーナーですら予測することが困難なほど未確定要素が多い。
解決策	プロセスを固定化した「かんばん」としてとらえ、制約理論(TOC)のバッファコントロールを取り入れている。
結果	
課題	
事例	8

1.1.8 スプリントレビュー

名前	シナリオ事前共有
状況	スプリントレビューでプロダクトオーナーにストーリーのデモを行い、内容を確認してもらう。
問題	スプリントレビューに時間がかかっている。
フォース	事前に作成したシナリオをプロダクトオーナーは確認したいが、確認に時間がかかる。
解決策	レビュー前にプロダクトオーナーにシナリオを事前に見てもらう。
結果	スプリントレビューの時間が短縮された。
課題	
事例	9

名前	マルチスクリーンでレビュー
状況	スプリントレビューの時にアジェンダを見ながら議事を書き進行している。プロダクトの動きをプロダクトオーナーに見てもらう必要がある。
問題	プロジェクタが一つなので、アジェンダとプロダクトの画面を切り替えないといけない。
フォース	見たい内容が二つあるが、画面は一つである。
解決策	レビューの際にはプロジェクタを二つ用意して、アジェンダ/議事用の画面と、プロダクト用の画面に分けて使った。
結果	切り替えの手間がなくなり、レビュー進行の効率化が上がった。
課題	
事例	9

1.1.9 タスクボード（タスクカード）

名前	半日以上のタスク追加禁止
状況	リリースまでの期間が短い。
問題	タスクの粒度が大きいとズレも大きい。
フォース	
解決策	タスクは半日以内に完了するくらい大きさに分割する。。
結果	より精緻な計画がつくれる。
課題	
事例	20

名前	タスクボード形骸化
状況	
問題	タスクボードの更新が追いつかずに形骸化
フォース	
解決策	なし
結果	

課題	
事例	6

名前	ステークホルダーのタスク忘れ
状況	開発チームだけがタスクボードを使っている。
問題	ステークホルダーがタスクを実施しない。
フォース	そのステークホルダーは、タスクボードを見ていないため、そのタスクを認識しない、もしくは忘れてしまう。
解決策	なし
結果	
課題	
事例	20

1.1.10 バーンダウンチャート

名前	終わらない計画の見える化
状況	スコープ調整がこれ以上できない。
問題	このままでは機能を期間内に開発完了させることができない。
フォース	
解決策	リリースバーンダウンチャートを使って終わらない作業を可視化する。
結果	数ヶ月で現実的なラインにもっていけた。
課題	
事例	11

名前	書き込みバーンダウンチャート
状況	バーンダウンチャートを利用している。
問題	バーンダウンチャートだけでは、対話が充実しない。
フォース	
解決策	バーンダウンチャートに様々な情報を書き込んでいった。
結果	コミュニケーションスペースになった。
課題	
事例	7

1.1.11 柔軟なプロセス

名前	プロセスの不足
状況	まずは技術面の向上で質の高いソフトウェアを作ることが大事である。
問題	現在のプロセスが適切かどうかを検討し改善出来る状態ではない。
フォース	
解決策	なし
結果	
課題	
事例	1

1.1.12 ユーザーストーリー

名前	ユーザーストーリーマップの共有
状況	
問題	ゴールを意識できていない。
フォース	
解決策	皆でストーリーを書き、ストーリーマップにした。
結果	
課題	
事例	23

名前	ユーザーストーリー以外での要件
状況	要件をユーザーに伝えたい。
問題	ユーザーストーリーだけでは伝わらないことがある。
フォース	
解決策	ユーザーの要件を、ユーザーストーリーだけでなく他の方法（例:UML など)も選択して記述した。
結果	
課題	
事例	8

名前	要求と画面のマトリクス
状況	ユーザーストーリーを使っている。
問題	要求の文章だけだと、どのような画面になるのか見当がつきにくい。
フォース	関係者には画面の名前を使って説明すると理解しやすい。
解決策	ストーリー（要求）と機能（画面）のマトリクスを作成した。
結果	ストーリーがどの画面に関連しているかがわかり理解が深まった。
課題	
事例	6

名前	ストーリーが大きすぎる
状況	2 週間のスプリントである。
問題	ストーリーが 2 週間では大きすぎるため、実現できない。
フォース	
解決策	なし
結果	
課題	
事例	24

1.1.13 プロダクトバックログ（優先順位付け）

名前	隠蔽スクラム
状況	顧客とパートナーはウォーターフォール型開発を行っている。自社ではスクラムを実施している。
問題	顧客、パートナーと自社での開発のやり方が異なる。
フォース	顧客とパートナーはウォーターフォール型開発に経験があり、スクラムの経験はなかった。
解決策	顧客、パートナーにタスクや進捗を提示する際、WBS に変換したものを渡す。
結果	関係を維持して開発することができた。
課題	
事例	21

名前	絵コンテでの説明
状況	女性向けサイトを作っているが、関係者全員が男性である。
問題	バックログアイテムの背景や理由がわからなかった。
フォース	
解決策	バックログアイテムの内容を絵コンテで説明した。
結果	理解が進んだ。
課題	
事例	12

名前	WBS で関連性の把握
状況	スクラムを用いてアジャイル型開発を行い、バックログで管理している。
問題	バックログアイテム間の関連性がよくわからない。
フォース	WBS を作成すると、作成された WBS 通りに進めるように暗黙の圧力が発生してしまう。
解決策	WBS を作り、バックログアイテムの内容を関係者間で共有し、WBS を修正しながら運用する。
結果	全バックログアイテムが一覧化され、関係者と共有できた。
課題	
事例	20

1.1.14 迅速なフィードバック

名前	中間報告
状況	受託開発で顧客と開発者が別拠点であった。Redmine をつかって拠点間で状況を共有していた。スプリントの最中で、計画が実行できない状況になってしまった。
問題	プロダクトオーナーにはスプリントの最中の状況が見えにくい。
フォース	開発者は遅れていると、ついもっと頑張ろうとしてしまい連絡が遅れてしまう。
解決策	見通しについて中間報告するイベントを設けた。直接対面で話をして、スプリントの計画が達成できそうになければ、頭を下げストーリーの出し入れをした。

結果	連絡が遅れることなく両者が状況の共有をすることができた。
課題	
事例	10

名前	フィードバック後のコミットメント
状況	顧客からフィードバックをもらっている。
問題	行わなければならないタスクが増えフィードバックに対応することができていない。
フォース	フィードバックはもらうが、「どれをどこまでやるか」が明確でなかった。 フィードバックを、どこで実現するかの計画がなかった。
解決策	フィードバックに優先順位を付けて実施するようにした。
結果	顧客のフィードバックを受け入れられるようになった。
課題	
事例	4

1.1.15 持続可能なペース

名前	必須機能の計画バッファ
状況	イテレーションを繰り返し実施することで、以前のイテレーションでの未対応事項に新しく発生した障害が上乗せされ、計画が滞ってくる。
問題	対応しなければならない仕事が多すぎて、すべてに対応するのが困難になっている。
フォース	イテレーションの中で使える時間は限られているが、仕事がたくさんある。
解決策	「確実にやりたいもの」「可能ならやりたいもの」「早く終ったらできるもの」を明確に区別してイテレーション計画をたてる。 「確実にやりたいもの」がギリギリ間に合うような計画を作らない。 「確実にやりたいもの」は 6 割くらいにしておき、余裕を持つ。
結果	持続可能で価値のある計画を立てることができた。早く終れば他のチームの手伝いもできた。
課題	
事例	4

名前	持続可能なペースの個別ヒアリング
状況	従業員は残業をしがちであるため、就業規則違反および健康上の問題が発生しやすい。
問題	残業をすると業務効率が下がり、品質が落ちる傾向があるため、なるべく就業時間内に行えるように努力し、プロジェクトとして品質と継続性を高めたい。
フォース	
解決策	メンバーひとりひとりに、持続可能なペースで開発を進められているかどうかをヒアリングし、残業をしないようお願いしていた。
結果	持続可能なペースでの開発の度合いがわかるようになった。
課題	
事例	15

名前	深夜コミットチェック
状況	仕事が忙しく、深夜や休日まで仕事をしている。
問題	無理をすると健康を害するため、結果持続的ペースから逸脱する。
フォース	
解決策	土曜や深夜に仕事をしていることが分かる仕組みがある。無理をしている人に声がけするようにしている。
結果	開発者が燃え尽きる前に状況を改善できる。
課題	
事例	15

名前	バッファのあるリリース計画
状況	
問題	リリース計画が順調にいかない。

フォース	
解決策	リリースまでの期間の1/3はバッファにしている。半分を過ぎて順調でない場合は、リリース日から逆算してリリース優先の線表を作る。
結果	
課題	
事例	20

名前	頑張りすぎるメンバー
状況	自社サービス開発で熱心なメンバーが多い。
問題	残業や長時間労働で頑張りすぎてしまう。
フォース	計画通りに仕事を終わらそうと思うと残業時間の増加が発生する。自らが決めた規律（Working agreement）を忠実に守る人が多い。自社サービスであるため、計画通りに終わらせないと、自らのビジネスや自社の評価に直結するため、がんばってやってしまう。
解決策	なし
結果	
課題	
事例	25

名前	飛び込みタスク
状況	リリース間近には飛び込みタスクが増える。
問題	タスクの総量も増え、リリースに間に合わないというリスクが増える。
フォース	
解決策	なし
結果	
課題	
事例	19

1.2 技術・ツール

1.2.1 ペアプログラミング

名前	準備過多なペアプログラミング
状況	ペアプログラミングを実施しようとしている。2人の間には大きなスキル差がある。
問題	ペアプログラミングの前に、スキルのある人がスキル不足の人向けに下準備をして時間がかかってしまった。
フォース	スキル不足の人に背景を説明しなければならないが、時間がかかる。
解決策	コードの共同所有を促進して理解を深めた。
結果	
課題	
事例	2

名前	ペアプログラミングでテストを書く
状況	ユニットテストの自動化ができていない。
問題	ユニットテストの自動化が後回しになり、結局未実施のままになりがちである。
フォース	必要性は認識しているが、先にテストが必要という意識になっていない。
解決策	テストを書くことが得意な人が教育目的でペアプログラミングを実施。
結果	チーム全員でテストが書けるようになった。
課題	
事例	2

名前	ペアプロスロット
状況	ペアプログラミングを行っている。
問題	ペアのマッチングは難しく、ペアリングの相手を選択することが難しい。
フォース	
解決策	スロットを回してペアを決めていた。
結果	恣意的ではなく偶発的にペアが組めるようになった。

課題	
事例	7

名前	技術不足を補う
状況	
問題	技術力が不足している。
フォース	
解決策	ペアプログラミングやホワイトボードを使った設計やモデリングを実施した。
結果	技術がある人が、技術不足の人へ伝達しながら、全体的な技術力が向上した。
課題	
事例	23

1.2.2 自動化された回帰テスト

名前	リスクベースドテスト
状況	受入テストやユニットテストを行っている。
問題	テストを網羅的に行うと時間がかかる。
フォース	すべての自動化を行いたいが、順序を決めたい。
解決策	プロダクト（サービス）の利用状況や利用に伴うリスクを調査し、利用状況やリスクが高い箇所を重点的にテストした。
結果	
課題	
事例	8

1.2.3 テスト駆動開発

名前	漸進的なテスト導入
状況	アジャイル型開発の導入は初めてであるため、開発メンバーは様々なツールの使用経験もない。
問題	早くアジャイル型開発のフルセットを利用したい。
フォース	アジャイル型開発でやるべきことはたくさんあるが、最初から全てを実践できない。
解決策	段階的にテストに関するプロセスを拡張していった。最初はテスト駆動もない状態で、途中からテスト駆動開発(TDD)を導入し、次にテストフレームワーク (Cucumber)を用いた受入テスト駆動開発(ATDD)を導入していった。そうすることで徐々にテストが「完了」する定義を拡張していった。
結果	チームの習熟度向上に伴い、アジャイルチームとしてやるべき事が増えた。ソフトウェアの品質を担保できる状態に成長できた。
課題	プロジェクト初期段階に作ったソフトウェアについてはテストが存在しないので、後で追加が必要である。
事例	9

名前	ペアでテスト駆動開発
状況	
問題	品質を上げたい。
フォース	
解決策	ペアでテスト駆動開発(TDD)を実践した。
結果	
課題	
事例	24

名前	テスト駆動開発(TDD)の工数がとれない
状況	短納期である。スコープは削れない。
問題	TDD に充分とりくむ工数がとれなかった。
フォース	
解決策	なし
結果	
課題	
事例	11

1.2.4 受入テスト

名前	受入テストの自動化

状況	受入テストを行っている
問題	受入テストの期間が必要になり、リリース（デリバリー）が短期間でできない。
フォース	受入テストは必ず実施する。
解決策	受入テスト導入と運用についてのコスト評価を行い、コストダウンにつながる場合は受入テストの自動化を計った。
結果	バグ対応などがあっても、一日以内にリリースできるような状況になった。
課題	
事例	8

名前	早期のシステムテストケース
状況	
問題	要求の実装漏れが早い段階から検出できない。
フォース	
解決策	早期からシステムテストケースを書いた。
結果	要求の漏れを早い段階から検出できた。
課題	
事例	24

名前	顧客が受入テスト拒否
状況	顧客は開発現場に頻繁にきてくれる。
問題	顧客がテストのための時間を取れない。
フォース	
解決策	
結果	デモで期待通りかだけ確認する。複雑なケースはテストしない。
課題	
事例	20

名前	受入テストに非協力な顧客
状況	顧客にストーリーの確認をしてもらいたい。
問題	顧客との間で何を確認すればいいのかの合意がとれていない。
フォース	顧客は要求を出すが、何をもってOKとするかまでは考えていない。
解決策	なし
結果	
課題	
事例	4

名前	機種依存のバグ
状況	ユニットテストは実施している。
問題	機種毎起因するバグが発生する。
フォース	機種毎のテストは自動化しづらい。
解決策	なし
結果	
課題	
事例	15

名前	Android 端末の多様性
状況	アプリは Android に対応する
問題	テストしなければならない端末種類が多く課題が満載である。
フォース	Android は端末単位で微妙に異なる。
解決策	なし
結果	
課題	
事例	15

名前	発注者と QCD のズレ
状況	受託開発をしている。
問題	発注元と QCD（品質・コスト・デリバー）の考え方が合わない。
フォース	
解決策	なし
結果	
課題	
事例	18

1.2.5 ユニットテストの自動化

名前	テスティングフレームワークの統合
状況	テスティングフレームワークである RSpec と Cucumber の両方を用いて、Web 開発プラットフォームである Rails で開発している。ベースになっている元のシステムがある。
問題	新しいメンバーに最短でテストの自動化を会得してもらいたい。
フォース	メンバーは両方のツールとも経験がないので慣れてほしいが、両方を学ぶのはコストがかかる。 既存のシステムの振る舞いを壊さないようにユニットテストのカバー率を拡張したい。
解決策	テスティングフレームワークを Cucumber に一本化し、ユニットテストと受入テストを自動化にした。
結果	学習コストを最小限に抑えることができた。 1 つのツールだけのテストデータを作るだけで良くなり、作業コストも削減できた。
課題	元々RSpec で書いていたテストを Cucumber で置き換える必要がでてきた。 Cucumber ではできないケースのユニットテストがあった。
事例	10

名前	ユニットテストの集中と選択（あきらめ）
状況	自動化されたユニットテストなしで開発を行っている。
問題	既にテストがないコードがある。
フォース	すべてのコードにテストをつけることは困難である。
解決策	フューチャやコードの利用状況や変更状況の統計に基づき、効果的な部分からテストコードを記述した。 逆に、コストに見合わない部分は、テストコードの記述をあきらめた。
結果	網羅率は 60%であるが、バランスの取れたユニットテストが行われている。
課題	
事例	8

名前	部分的ユニットテストの自動化
状況	ユニットテストを自動化している。
問題	自動化が進まない。
フォース	テストケースの量が多く、中には複雑なものもある。
解決策	全てを自動化することはやめて、部分的にした。複雑なものは手動で実施するようにした。
結果	
課題	
事例	6

名前	エンドツーエンドテストでカバー
状況	ユニットテストを書いている。
問題	テストが書きづらい。
フォース	
解決策	外部インターフェイスや端末からの操作を想定したエンドツーエンドテストのカバレッジでカバーしている。
結果	
課題	
事例	18

名前	ユニットテスト自動化へ後ろ向き
状況	部分的にはユニットテストは自動化できている。
問題	プロジェクト立ち上げ時にユニットテスト自動化の重要性が実感できないため、自動化を推進できない。
フォース	継続していく上で重要になるが、立ち上げ時はピンとこない。後で実施しようと思っても、まとまった時間やコストが必要になる。
解決策	なし
結果	
課題	
事例	15

名前	テストが後回し

状況	やらなければいけないことがたくさんある。
問題	ユニットテストの自動化が後回しになり、未実施のままになりがちである。
フォース	必要性は認識しているが、先にテストが必要という意識になっていない。
解決策	なし
結果	
課題	
事例	2

名前	ユニットテストの自動化断念
状況	テストの自動化は実施したい。
問題	時間がなくて自動化まで手が回らなかった。
フォース	
解決策	なし
結果	
課題	
事例	7

名前	テストが書きづらい
状況	ユニットテストを書きたい。
問題	設計が複雑になってテストが書きづらくなっている。
フォース	
解決策	なし
結果	
課題	
事例	18

1.2.6 リファクタリング

名前	リファクタリングレーン
状況	タスクボードを使っている。
問題	リファクタリングをつい忘れてしまいがちである。
フォース	リファクタリングの工数は見積ってはいる。
解決策	リファクタリングをタスクボードのレーンとして表現する。
結果	忘れずにリファクタリングできる。
課題	
事例	20

名前	日常的なリファクタリング
状況	リファクタリングを重視している。
問題	リファクタリングの習慣が根付いていない。
フォース	リファクタリングを中断すると、リズムが崩れ、継続されない。
解決策	リズムを持って息をするように、1つのモジュールのリファクタリングを少しずつに分けて何度も実施した。
結果	小さなリファクタリングが積み重なって、ソースコードを常に良い状態を保てている。
課題	
事例	1

名前	ツールによるリファクタリングチェック
状況	静的解析ツールを使用している。
問題	リファクタリングを実施の判断がつきにくい。
フォース	
解決策	解析ツールにルールを定義して、スコアが低ければリファクタリングするようにした。
結果	リファクタリングの必要なタイミングがはっきりするようになった。
課題	
事例	15

名前	リファクタリングの工数確保
状況	リファクタリングをせず、ソースコードの品質が悪いことで、以前、顧客を困らせた経験があった。
問題	見積の際にリファクタリングの工数を確保していない。
フォース	

解決策	顧客からリファクタリングの工数として 15%多く見積もるように言われた。
結果	リファクタリングを前提にした開発が実現できる。
課題	
事例	20

名前	リファクタリングの断念
状況	要件を元に計画を作っている。リファクタリングを実施したい。
問題	リファクタリングが計画に入っていないため、実施する時間をとれない。
フォース	機能開発を優先せざるを得ない。
解決策	なし
結果	
課題	
事例	7

1.2.7 シンプルデザイン

名前	クロスプラットフォームフレームワークの利用
状況	ソフトウェアは iOS、Android をターゲットにしている。
問題	複数のプラットフォームに対応する際には別々に開発しないといけない。
フォース	それぞれのプラットフォームでは互換性が皆無である。
解決策	クロスプラットフォーム対応のフレームワークを用いて開発する。
結果	
課題	
事例	15

名前	疎結合なアーキテクチャ
状況	対象とするソフトウェア規模が大きい。
問題	インテグレーションの問題を減らしたい。
フォース	インテグレーションについて詳細に検討するが、複雑だと要素間の調整事項が多くなる。
解決策	疎結合なアーキテクチャにする。レイヤー構造などいくつかのアーキテクチャのパターンを適用する。
結果	
課題	
事例	1

名前	非機能要件標準の策定
状況	複数のチームがあり、それぞれのチームで非機能要件を事前に洗い出している。開発には共通フレームワークを用いている。
問題	複数チームが個別に非機能要件を策定すると、セキュリティやパフォーマンスなどの非機能要件がチームごとにバラバラになった。場合によってはヌケモレが発生した。
フォース	
解決策	部門など 1 つの組織で共通の開発標準を定め、共通フレームワークに実装することによって、複数チームを通じ全体として非機能要件を担保する。
結果	個別チームで考慮すべきことが減り、チーム間での違いが減った。またヌケモレがなくなった。
課題	
事例	15

名前	クロスプラットフォームの開発標準
状況	スマートフォンは、異なったプラットフォームを持つ複数の機種があり、それぞれで正常に動作するアプリケーションを開発している。
問題	開発プラットフォーム毎に標準が異なるため、それぞれに個別に開発することは効率的ではない。
フォース	
解決策	クロスプラットフォームでの開発標準を策定して展開する。
結果	ツールを開発プラットフォーム間で横展開でき生産性が向上した。
課題	
事例	15

名前	クリーンコード保持困難
状況	イテレーション毎にソフトウェアを開発しているため、イテレーションの度にコードが新しく増えていく。
問題	イテレーション内でコードを見直すことができていない。
フォース	イテレーションの度にストーリーを開発しなければならないが、コードを見直さないと将来的にリスクが増えていく。
解決策	なし
結果	
課題	
事例	4

1.2.8 集団によるオーナーシップ

名前	外部からの変更要求
状況	ソースコードをWeb上のリポジトリに置いてある。
問題	チーム以外からもソースコードやドキュメントの変更要求がない。
フォース	コードの閲覧は可能である。folk（元のソースコードを分岐して編集したもの）は作れる。 コミット権限を与えることはできない。
解決策	外部からの変更要求を受け付けるようにした。
結果	
課題	
事例	1

名前	チームを越えたオーナーシップ
状況	リポジトリにソースコードを置いてある。 他のチームの業務領域と関連している。
問題	チーム以外が関わっているサービスや、プロダクトとの関係性を保って開発を進めたい。
フォース	同時に開発するには、チームおよび開発範囲が大きくなってしまう。
解決策	チームメンバー以外もソースコードを閲覧、変更できるようにしておく。
結果	より多くの目でのフィードバックが可能になる。
課題	
事例	1

名前	ソースコードのプロジェクト横断共有
状況	プラットフォームを提供する基盤チームがいる。
問題	問題が発生したときのプラットフォームの問題か、そのプラットフォーム上で動いているアプリケーションソフトウェアの問題かの切り分けが難しい。
フォース	切り分けようとしても、他のチームのソースコードがわからない。
解決策	プラットフォームのソースコードをプロジェクト横断で見られるようにした。
結果	基盤チームが横断的に修正することができる。
課題	
事例	15

名前	ソーシャルコーディング
状況	Web を通じて共有されたコードの分散リポジトリを使っている。
問題	コードの品質が高まらない。
フォース	
解決策	pull request を利用して、修正する人と修正をコードベースに反映する人の両方の目で確認するようにした。
結果	1 コミットあたり 2 人以上の確認が入るため、質の高い修正のみをコードベースに反映することができる。
課題	
事例	25

1.2.9 コーディング規約

名前	プラットフォーム横断コーディングルール

状況	プラットフォームを提供する基盤チームがいる。
問題	プラットフォームの適切な使い方を知りたい。
フォース	人の出入りが多く、技術習得コストが高くつく。
解決策	共通プラットフォーム横断でコーディング規約を策定した。
結果	
課題	
事例	15

名前	プラットフォーム横断コーディングルール
状況	プラットフォームを提供する基盤チームがいる。
問題	プラットフォームの適切な使い方を知りたい。
フォース	人の出入りが多く、技術習得コストが高くつく。
解決策	共通プラットフォーム横断でコーディング規約を策定した。
結果	
課題	
事例	15

名前	コードフォーマッタ
状況	コーディング規約をつくった
問題	チームがコーディング規約を守れない。
フォース	
解決策	ソースコードを整形するコードフォーマッタに規約を定義して、それに従いソースコードを修正するようにした。
結果	コーディング規約を守れるようになった。
課題	
事例	2

1.2.10 紙・手書きツール

名前	紙・手書きツールへ移行
状況	チケット管理システムを使ってスプリントバックログを管理している。
問題	改善を促すことができない。
フォース	チケット管理システムだと変更する手間がかかる。画面が小さくて朝会の時に見えづらい。スクラムマスターから見ても内容が把握できない。
解決策	紙・手書きツールを壁に貼り出すようにした。
結果	改善を促すことができた。
課題	
事例	25

名前	紙・手書きツール面倒
状況	紙や手書きツールでタスクボードを使っていた。
問題	字を手で書くのが面倒になった。
フォース	
解決策	なし
結果	
課題	
事例	23

1.3 チーム運営・組織・チーム環境

1.3.1 顧客プロキシ

名前	ブリッジ・プロダクトオーナー
状況	オフショア開発をしている
問題	オフショア先の開発チームでは、プロダクトやサービスの理解が進まない。
フォース	テレビ会議では、プロダクトやサービスの勘所がわからない。プロダクトオーナーと開発チームのコミュニケーションを密に取ると、細かい複数の作業が断続的に発生する状態になり、コストがかさむようになった。

解決策	プロダクトオーナーを、オンサイトが補佐した。経験を積んだ後、オフショアでブリッジ・プロダクトオーナーになってもらった。本来のプロダクトオーナーは役割分担を決めて後は任せた。
結果	プロダクトオーナーとして、業務の負担を減らしつつ、オフショア先との開発を進められるようになった。
課題	
事例	8

1.3.2 オンサイト顧客

名前	覆される決定
状況	オンサイト顧客として週に数回顧客役が来てくれる。
問題	顧客役が決めた優先順位がひっくりかえることがある。
フォース	権限のあるプロダクトオーナーは別にいる。
解決策	なし
結果	
課題	
事例	20

1.3.3 プロダクトオーナー

名前	現場見学
状況	工場で使われる端末のシステムを開発している。
問題	開発者は端末上の動きはチェックしているが、実際の利用現場でどのように使われているかはわからない。
フォース	システムはハンドヘルド端末として現場で使われる。開発者は、現場での業務内容までは把握していない。
解決策	開発メンバーが実際に端末が使われる工場に出向き、現場でどのように使われるかのレクチャーを受けた。
結果	顧客の環境を踏まえた、想像ではないシステムへの提案ができるようになった。
課題	
事例	9

名前	ペルソナ法の活用
状況	開発チームは現場を見学してきた。
問題	システムの要件を考える際に、現場で本当に有用なのかがイメージしづらい。
フォース	システム化対象業務に詳しいドメインエキスパートは多忙で、プロジェクトに割ける時間がない
解決策	現場の利用者の仮想人格（ペルソナ)を設定し、その人物の視点で要件を検討した。
結果	システムについての要件を検討、提案できるようになった。
課題	
事例	9

名前	ドメインエキスパートプロダクトオーナー
状況	チームはプロダクトオーナー役を設置する必要がある。
問題	現場の問題をリアルに把握した上で要件を出してほしい。
フォース	プロダクトオーナーが現場を把握できていなければ、現場の人間に要件を聞きださなければならなくなる。
解決策	真の顧客にプロダクトオーナー役を演じてもらう。
結果	現場の問題に直結した要件を収集でき、決断も早かった。
課題	担当している現場の個別最適になってしまい、全社的な視点が欠如してしまった。
事例	10

名前	顧客への教育
状況	顧客にプロダクトオーナー役を任せたいと考えている。
問題	顧客はプロダクトオーナーが何をアウトプットすべきなのか具体的に知らない。
フォース	

解決策	顧客に必要な作業（ユーザーストーリーやバックログの維持）を教育する。
結果	顧客がプロダクトオーナーとして役割を担えるようになった。
課題	
事例	10

名前	顧客（プロダクトオーナー)サポーター
状況	顧客は 3 人いたが、100%開発に割当てられているわけではなく本来の業務ももっている。階層関係もあって１人の人に作業が集中していた。
問題	顧客役として要件の詳細化や、受入テスト、ストーリーの詰めができていなかった。
フォース	顧客側から支援の人間は追加できない。
解決策	開発側から、顧客側に１人出向させて、顧客役の作業を支援するようにした。
結果	仕事が回るようになった。
課題	
事例	4

名前	経営陣の巻き込み
状況	プロダクトオーナーは一担当者である。システムが経営に影響を及ぼすことがある。
問題	権限を持つ人物とプロダクトオーナーの意見の相違が後で発覚した。
フォース	現場レベルで決定をして開発を進めても、後で経営的判断で覆されてしまう恐れがある。
解決策	経営的判断が必要でプロダクトオーナーに判断できない場合には、顧客社長を含めた経営陣が責任もって即座に意思決定する体制にした。
結果	決定が覆されることがなくなった。
課題	権限を持つ人物は忙しい。
事例	4

名前	ディレクターへの権限委譲
状況	ディレクター（企画者）は、別組織のステークホルダーに決裁をとるようにしていた。
問題	決裁と調整の時間がかかりベロシティがあがらない。
フォース	
解決策	ディレクター（企画者）がステークホルダーに対し、決裁をとらずに決定できるよう調整した。利用者によるダウンロード数や成約率などのデータに基づいて開発する機能を決定をしていくようにした。
結果	決定速度が素早くなった。
課題	
事例	15

名前	開発者がプロダクトオーナー代行
状況	スクラムを実践している。
問題	プロダクトオーナーが非常に多忙である。
フォース	開発チームはプロダクトビジョンの段階からプロダクトオーナーと共に関わっている
解決策	プロダクトオーナーの仕事をほとんど開発チームが行っている。最終決定はプロダクトオーナーが行うが、優先順位も含めて開発に任されている。
結果	プロダクトオーナーのボトルネックはなくなった。
課題	
事例	2

名前	社内アドバイザー
状況	開発メンバーよりも営業の方が顧客のことをよく知っている。営業もアジャイル型開発についての知識が充分にある。
問題	価値ある仕事だけをしたい。
フォース	「プロダクトとして価値があるか」という観点で考えることが重要だが、開発者は「モノ作り」に考えが偏りがちである。
解決策	営業がアドバイザーとなって、ストーリーやプロセスに対して客観的な意見をくれた。
結果	開発メンバーに様々な気づきがあった。
課題	

事例	9

名前	開発チームがプロダクトオーナー
状況	自社開発でプロダクトオーナーが存在しない。
問題	ゴールを決めてくれる人がいない。
フォース	
解決策	チームが自分でゴールを決めるようにした。
結果	利用者からフィードバックをもらうことができた。
課題	
事例	7

名前	開発兼任のプロダクトオーナー
状況	研究開発プロジェクトである。
問題	ストーリーを書くのも技術力が求められる。
フォース	
解決策	開発メンバーがプロダクトオーナーも兼任した。
結果	
課題	
事例	23

名前	決裁者不在
状況	プロダクトオーナーやアドバイザーは予算をもっていない。何をしても役員、社長の決済が必要になる。
問題	プロダクトオーナーだけでプロダクトについての決断を素早く行うことができにくい可能性がある。
フォース	プロダクトオーナーが予算をとりたいが、決裁者に説明してお伺いをたてないといけない。
解決策	なし
結果	
課題	社長がアジャイルを推進している方がよいと感じている。
事例	10

名前	プロダクトオーナーがボトルネック
状況	開発メンバーはスキルが高く開発で困ることはない。
問題	プロダクトオーナーが日常業務にかなり時間をとられている。
フォース	
解決策	なし
結果	
課題	
事例	13

1.3.4 ファシリテータ（スクラムマスター）

名前	スクラムマスター専任化
状況	スクラムマスターを開発と兼任している。
問題	短期的な視点での改善が多くなり、中長期的な視点での改善が進まない。
フォース	
解決策	スクラムマスターを専任にすることで、複数チームを見られるようになった。
結果	
課題	
事例	25

名前	開発の状況を知らないスクラムマスター
状況	スクラムマスターの開発経験が乏しいため、をしていない。あまりコードを見ていない。
問題	スクラムマスターの言うことに説得力がない。
フォース	スクラムマスターはチームが具体的に何を開発しているのか知らない。
解決策	なし
結果	
課題	
事例	20

1.3.5 アジャイルコーチ

名前	オフショアスクラムでのコーチ
状況	オフショアでスクラムを使ったアジャイル型開発を行っている。
問題	ロケーションが離れているため、アジャイル型開発の方法を適切に伝えることができない。
フォース	日本とオフショア先に物理的な距離がある。
解決策	コーチが国内のプロダクトオーナーにアジャイル型開発のトレーニングを行い、その後2人で中国へ渡り、アジャイル導入研修を実施した。
結果	今後もプロジェクトに携わるプロダクトオーナーを教育することで、プロダクトオーナーとオフショア先の開発チームでスクラムを継続できた。
課題	
事例	21

1.3.6 ニコニコカレンダー

名前	負荷やストレスの見える化
状況	メンバーはニコニコカレンダーに日々の気持ちを書いている。
問題	メンバーは負荷やストレスの見える化に興味がなく、継続されなかった。
フォース	元気のないメンバーに対し、メンタルヘルス相談に行かせるかどうかも判断がつかない。
解決策	なし
結果	
課題	プロジェクト内メンバーの気持ちが見える化されていない。
事例	8

1.3.7 組織に合わせたアジャイルスタイル

名前	アジャイル型開発の採用
状況	ウォーターフォール型開発で開発している。
問題	結合テストで大量にバグが発生し、要件定義など前工程からの見直しをしなければならなくなったため、プロジェクトが停止した。
フォース	結合テストを成功させるために綿密な計画を立てているが、計画通りに進まない。
解決策	アジャイル型開発で最初からやり直した。 大きな規模の結合をせず、小さな単位で確認しながら統合している。
結果	当初の定義された要件のうち、6割程度は実現する必要がないことが判明した。
課題	
事例	8

名前	アジャイル（スクラム）型開発をやめる
状況	スクラムで継続的な開発をしている。決定権を持つ強力な外部要因によりスプリントの内部的な変更が多い。
問題	計画ゲームで時間を要するなどコストがかさみ、スプリントを維持することが困難である。
フォース	ユーザからの一つ一つの要求が小さい。
解決策	スクラムをやめ、小さなウォーターフォールにする。
結果	
課題	
事例	11

名前	トライアル期間
状況	お客様はアジャイル型開発での契約は初めてである。
問題	従来と異なるアジャイルの進め方に納得した上で契約したい。
フォース	アジャイル型開発についての説明はしているが、実際に始まってみないと気づかないことがある。
解決策	無料でトライアルを実施した。どういう進め方になるか見通した上で契約するようにした。
結果	契約後、クレームなどの問題発生を防ぐことができた。
課題	

事例	10

名前	かんばんとスクラム
状況	サービス提供を目指している、もしくは既にサービス提供している。
問題	スクラムは、見積ができる状態の開発に向いているが、企画段階や、運用保守などは見積が困難であり、スクラムのフレームワークがボトルネックになっている。
フォース	計画と見積に基づいたタスクを考えているが、変化が激しく意味のないものになってしまう。
解決策	企画や保守運用ではかんばんを用いる。ボトルネックをが発生したときは、そのボトルネックを排除する。開発はスクラムを用いる。その結果、全体としては「かんばん→スクラム→かんばん」となった。
結果	
課題	
事例	25

名前	スクラムウォーターフォール
状況	全面的なシステムリニューアルのタイミングであった。アジャイルの経験はなかった。アジャイルコーチが参画した。
問題	段階的にアジャイルを導入したかった。
フォース	いきなり 100%のアジャイル型開発を行うと、習慣や常識が違うため反発する人が多くなり、妥当性を示せなくなる。
解決策	要件定義とテストはウォーターフォール型開発で行い、設計開発は反復的に実施した。
結果	反発も少なく妥当なスタイルから始められることができた。
課題	
事例	7

名前	WBS 変換
状況	アジャイル型ではない開発を行っているチームと連携する必要がある。
問題	アジャイル型開発を行っているチームと行っていないチームで進捗を調整しないといけない。
フォース	WBS はウォーターフォール型開発チームでよく使われている。アジャイルでは用いられない。
解決策	作業計画や状況を WBS に変換してウォーターフォール型開発チームに渡した。
結果	
課題	
事例	23

名前	アジャイルと言わない
状況	複数組織を横断しているプロジェクトであり、アジャイル型開発を採用すると効果がありそうな状況である。
問題	アジャイルという言葉に抵抗感があり、アジャイル型開発の導入が困難である。
フォース	アジャイルプラクティスの効果が高そうだが、アジャイルという言葉に抵抗感を持つ人が存在する。
解決策	アジャイル型開発を全面的に採用せず、プラクティスの一部を採用した。
結果	抵抗なくプラクティスを採用できた。
課題	
事例	25

名前	スクラムをやめる
状況	スクラム開発をして、プロダクトオーナーと開発チームが分離している。
問題	プロダクトオーナーが何からの理由でビジネスの企画を放棄する。
フォース	
解決策	開発とビジネス企画を同じチームにするなど一体化する。 XP が近いので、XP の面を強化する。
結果	
課題	
事例	5

名前	テストチーム
状況	開発者が期限に追われ、目の前の機能の実装だけに集中している。
問題	テストについて考える余裕がない。
フォース	
解決策	テストチームを構成して、開発とは別に実施している。
結果	
課題	
事例	15

名前	専門化したチーム
状況	オフショア先に開発を依頼している。
問題	オフショア先の担当がそれぞれ専門化してしまったため、その専門以外の作業を依頼できなくなった。
フォース	
解決策	
結果	
課題	
事例	13

1.3.8 共通の部屋

名前	オフショアメンバーがオンサイト
状況	オフショアで分散開発をしたい。
問題	オフショア先では、進め方や背景などを理解しにくい。
フォース	
解決策	オフショア先の一部のメンバーに来てもらい開発した。ローテーションした。
結果	オフショア先に戻った後も円滑に開発できた。
課題	
事例	6

名前	島で会議スペース
状況	チームは一つの「島」スタイルの座席配置で仕事をしている。
問題	会議室の予約がとりづらいため、頻繁に会議を開けない。
フォース	会議室を増築する予算も時間もない。
解決策	座席で作られた「島」に、大きなディスプレイを置き、壁にホワイトボードを貼って、気軽に会議ができるようにした。
結果	会議室に移動しなくても会議ができるようになった。
課題	
事例	15

名前	最初は同一フロア
状況	これからオフショア先に開発を出そうとしている。
問題	意思疎通が分散拠点では難しい。
フォース	
解決策	最初の 1～2 週間は同一拠点で開発する。
結果	信頼関係を築けて分散開発のコミュニケーションに役立った。
課題	
事例	13

1.3.9 チーム全体が一つに

名前	クレド作成
状況	プロジェクトに大勢の人が関わっている。
問題	大勢の人達を一つのチームに仕立てるのが難しい。
フォース	1 人 1 人が違った想いを持って、プロジェクトに参加している。
解決策	プロジェクトの信条や行動規範を記述したクレドを作った。
結果	全員で意識をあわせて実施することができた。
課題	
事例	4

名前	チームの約束
状況	チームで自らの仕事のやり方を改善をしている。
問題	チームで決めたことを守れない。

フォース	個人任せにしていると、ついつい忘れてしまうことがある。
解決策	「チームの約束」という形で、自分達で決めたことを書出して壁に貼り出しておく。
結果	チームとして行動規範が守れるようになった。
課題	
事例	25

名前	開発者の問題共有
状況	部門内の開発者は別々のプロジェクトチームに所属しており、普段はプロジェクトメンバーとの交流が主である。
問題	各自の問題を共有する場がない。
フォース	同一フロアで作業している。
解決策	プロジェクト横断で開発者同士のミーティングを開き、互いの迷っている点、抱えている問題を共有する。
結果	問題や悩みが解決、また抱え込みが少なくなった。意識改革ができてきた。
課題	
事例	15

名前	企画者の問題共有
状況	部門内の企画者は別々のプロジェクトチームに所属しており、普段は同じプロジェクトのメンバーとの交流が主である。
問題	各自の問題を共有する場がない。
フォース	
解決策	プロジェクト横断で企画者同士のミーティングを開き互いの迷っている点、抱えている問題を共有する。
結果	問題や悩みが解決、また抱え込みが少なくなった。意識改革ができてきた。
課題	
事例	15

1.3.10 人材のローテーション

名前	メンバー固定
状況	オフショアも含めた中長期（数年スパン）のプロジェクトである。
問題	開発チームのメンバーが変わると、新たに教育する必要がある。
フォース	メンバーは減ることがあっても増えることがないプロジェクト状況である。
解決策	開発チームのメンバーの入れ替えが起こらないように手配した。
結果	開発や業務に関するノウハウが成熟した。
課題	
事例	8

名前	持ち回りのスクラムマスター
状況	オフショアも含めた中長期（数年スパン）のプロジェクトである。
問題	スクラムマスターの役割を理解していない。
フォース	
解決策	チーム・メンバーがお互いにスクラムマスターについての教育・サポートをしながら、スクラムマスターの役割を持ち回りで付与した。 スクラムマスターの役割に慣れていないチーム・メンバーは、慣れるまでの期間従事した。
結果	最終的には、全員がスクラムマスターの役割を理解した。その結果、自己組織化され、固定のスクラムマスターは不要になった。
課題	
事例	8

名前	チームメンバーは固定
状況	メンバーが既にチームとして開発を経験している。
問題	チームを組んで新しいアプリを開発する際には、最初は開発速度がでない。
フォース	

解決策	別のアプリ開発を立ち上げる際には、できるだけ既存のチームで開発する。
結果	チームワークが最初からできており開発の立ち上がりが速い。
課題	
事例	15

名前	係のローテーション
状況	チームにはメインの開発以外にも色々な仕事がある。(例:バーンダウンチャートのプロット)
問題	開発以外の仕事も、誰かがやらなければならない。
フォース	リーダーが指示することはない。
解決策	XXX 係を任命して、一週間単位で回すようにした。
結果	全員が責任をもって開発以外の仕事をできるようになった。
課題	
事例	2

名前	担当が決まってしまうコード
状況	関与する開発者が多い。
問題	特定の人しか知らない部分が増え、障害対応などで困る。
フォース	なんとなく担当が決まってしまう。
解決策	なし
結果	
課題	
事例	1

1.3.11 インテグレーション専用マシン

名前	カナリア環境
状況	改善要求を受け付けている。
問題	新たな要件・要望の価値や効果が不明である。
フォース	要件・要望を持っている本人にとっても、その価値がわからない。
解決策	工数を掛けない工夫をしつつテスト環境（カナリア環境）を用意し、そこでアーリーアダプタ（新しい機能にいち早く導入することを許容する関係者）や要件提供者によるプロトタイプを体験してもらう。
結果	実装の必要性や改善方法などを、提案者とプロダクトオーナー側でコンセンサスが取れた。実装する必要がない要求なども明らかになった。
課題	
事例	8

名前	運用の自動化
状況	運用を実施している状況で、ミスが許されない状況である。
問題	詳細な手順書を用意しても、ミスが起こる可能性はある。
フォース	
解決策	手順を文書ではなくスクリプトで自動化する。
結果	ミスをせずに運用されるようになった。
課題	
事例	1

名前	共通インテグレーション環境
状況	様々なアプリケーションを開発している。
問題	リリースの度にインテグレーション環境を構築するのはコストがかかる。
フォース	
解決策	クラウド上にインテグレーション環境を構築して利用し、他の部門からも構築できるようなキットを整えた。
結果	インテグレーション環境をゼロから作る必要がなくなった。
課題	
事例	15

名前	チェックシート

状況	本番リリースや受入テスト、スプリントレビューの準備に、何らかの確認や手作業が発生している。 ファシリテータを全員で回したい。 チームはふりかえりで改善策を考えている。
問題	チェックをすることや、また手順や工程を忘れることがある。 新しい決め事や、ファシリテータに任せている手順を忘れてしまう。
フォース	自動化を目指しているが 100%は難しい。 ネットワークが分離された環境などの確認を忘れてしまう。
解決策	手順を忘れないようにするためチェックシートを作り、朝会等で定期的にチェックする。
結果	
課題	
事例	2

1.4 そのほかのプラクティス

1.4.1 ユーザーストーリーマッピング

名前	ユーザーストーリーマッピング
状況	プロダクトオーナーはスコープを決める必要がある。
問題	何が必要で、何が不要かを判断したい。
フォース	プロダクトを提供する上での必要最低限の機能を把握したい。
解決策	ユーザーストーリーマッピングをプロダクトオーナーと開発者が実施した。
結果	全体像を見ながらプロダクトオーナートと会話することができた。 優先順位をつけやすくなった。
課題	壁一面に貼り出されるため持ち出すことができない。
事例	9

名前	ユーザーストーリーマッピング
状況	プロダクトオーナーはリリースやスプリント内で何を実現するかのスコープを決める必要がある。 開発メンバーはプロダクトの本質を踏まえた上でプロダクトオーナーにフィードバックをしたい。
問題	スコープ管理をうまく実施したい。
フォース	大事なものが何なのかを他と比較しながら見極めなければならない。 プロダクトの全体像について開発メンバーがきちんと理解できているかは疑問である。
解決策	ユーザーストーリーマッピングを実施した。
結果	プロダクトの全体像を見ながら必要なストーリーを選択してスコープを決められるようになった。
課題	紙に書いて壁に貼りだしたユーザーストーリーマップは、別拠点からは参照できない。
事例	10

名前	モバイルユーザーストーリーマップ
状況	ユーザーストーリーマップの導入によってスコープマネジメントが行いやすくなった。 顧客の会社に赴いて計画づくりを実施した。
問題	壁にユーザーストーリーを直接貼りだしてしまうとユーザーストーマップは持ち出せない。
フォース	ユーザーストーリーマップは便利だが巨大になってしまう。表計算ソフトなどで電子化してしまうと、全員で俯瞰してみることができない。 顧客の会社と、開発拠点の両方でマップを閲覧したい。
解決策	模造紙の上にストーリーマップを貼り出し持ち運び可能にした。
結果	どこにでもストーリーマップを持ち込めるようになったことで、計画づくりをはじめとしたミーティングを顧客と一緒に実施できるようになった。
課題	
事例	10

1.4.2 「完了」の定義

名前	「完了」定義のテンプレート
状況	チーム内でスクラムで提唱されている「完了」の定義を作ろうとしている。
問題	何をもって「完了」とするかの定義を、どう作っていいかよくわからない。
フォース	
解決策	「完了」を４つのステップに分け（コミット、タスク、ストーリー、スプリント)、それぞれに完了の条件を定義していった。
結果	どのチームでも「完了」の定義を作れるようになった。
課題	
事例	2

名前	「完了」の定義
状況	スクラムでチーム開発をしている。
問題	作業の「完了」の定義がチームの中でバラバラになる。
フォース	
解決策	「完了」の定義を決めて、ふりかえりの度に拡張していった。
結果	チームの中で「完了」が共通化された。齟齬がなくなった。
課題	
事例	25

1.4.3 楽しい工夫

名前	おやつ神社
状況	開発ルームでは、メンバーはよくお菓子を食べたり、ギャグを言ったりしている。
問題	メンバーが楽しみながら開発するにはどうすればよいか。
フォース	
解決策	ギャグがつまらなかった人は、お菓子購入用に罰金として100円を貯金箱に支払うようにした。
結果	お菓子を購入する資金が溜まった。
課題	つまらないギャクは減らずに、むしろ増えた。
事例	9

名前	漢気じゃんけん
状況	チームメンバーとファシリテータ（スクラムマスター）を兼任し、チームメンバーが持ち回りで実施している。最初は自己申告制で実施していたが、面倒になってきた。
問題	ファシリテータ（スクラムマスター）の選任が難しい。もっと簡潔に当番を決めたい。
フォース	指名や立候補での選任は、人間関係上難しい。 持ち回りにすると単調になる。
解決策	ファシリテータは、じゃんけんで負けた人が罰ゲーム的に担うのではなく、勝った人に任せるようにしていた。また、朝会のファシリテータは毎日交代するようにし、ちょうど5人だったため1週間で1巡するようにじゃんけんで順番を決めていた。
結果	非常に盛り上がり、チームの雰囲気作りにとても役に立った。
課題	
事例	2

名前	バグや課題を食いつぶせ
状況	障害や課題、事象などを受け付ける。
問題	障害や課題、事象などに積極的に取り組めない。
フォース	精神的プレッシャーが高まり、行き詰まってしまう。
解決策	バグ管理表や課題管理表を某ラーメン店メニューのチョモランマやマシマシなどの隠語を使い、障害の深刻度を管理している。障害管理というとプレッシャーが高まり、チームの雰囲気が悪くなるため、それを軽減したかった。
結果	これにより、チームが課題に対して前向きに対処できるようになった。
課題	
事例	2

名前	ラーメンメニュー（バグ管理表・課題管理表）
状況	バグやトラブルなどの不具合や事象が発生している。
問題	バグや不具合の状態を知りたい。
フォース	バグやトラブルなどの言葉にはネガティブな印象があり、プレッシャーがかかる。
解決策	「バグ」の深刻度をラーメンの具の分量に喩えることによって、バグや不具合管理を楽しく行った。
結果	バグやトラブルに対して、積極的に対応できるようになった。
課題	
事例	2

1.4.4 勉強会

名前	朝の勉強会
状況	チームには開発スキルが不足しているため、より開発を効率化するために習得が必要なスキルやツールがまだまだある。
問題	チームとして早急なスキルアップが必要である。
フォース	各人個別に学習する時間や余裕がない。
解決策	朝の 30 分を勉強会の時間に充て、チームで学ぶ時間にした。
結果	開発立ち上がり時の、まだスキルが低い段階で、必要なことを学べた。
課題	ネタがなくなり、下火になった。
事例	9

名前	朝稽古
状況	毎朝 30 分、朝の勉強会を実施していた。
問題	忙しい時は勉強会に充てる時間も惜しい。
フォース	忙しい時でも日々のスキルアップを行いたい。
解決策	毎朝 30 分をプロジェクトをよりよくするための活動を自由にできる（勉強をしてもよいし、仕事をしてもよい）時間にした。
結果	時間の使い方の自由度が高くなった。
課題	
事例	10

名前	社内勉強会の開催
状況	技術について経験や知識がバラバラである。
問題	技術スキルを向上させたい。
フォース	技術スキルを高めたいが、費用や時間の面で社外講習への参加が難しい。
解決策	業務時間内において頻繁に社内で勉強会を実施する。
結果	
課題	
事例	1

名前	社内への勉強会の招聘
状況	社外に継続的に開催している勉強会や読書会がある。
問題	技術スキルを向上させたい。
フォース	技術スキルを高めたいが、社内にはその領域の専門家がいない。
解決策	社外コミュニティの勉強会を招聘して、そこに社員が参加する。
結果	
課題	
事例	1

名前	横断的勉強会
状況	複数のチームそれぞれに向上心がある。
問題	技術や知識が複数チームで共有されていない。
フォース	一つのチームで勉強をすると、メンバーやテーマがいつも同じになり煮詰まってしまう。
解決策	週 1 回、2 時間の勉強会を実施している。ワークショップをメインに行っている。「他のチームのタスクボード見学」、「5 分間で発表するライトニングトーク大会」「チーム編成」「テスト」などさまざまなテーマについて学ぶ。

結果	最初は1チームで勉強会を実施していたが、徐々に他のチームのメンバーも参加するようになってきた。 共通の知識を得ることができて、コミュニケーションが容易になり、アイデアの幅も広がった。
課題	人数が増えす過ぎたため、会場が押さえられなくなってしまった。
事例	2

1.4.5 組織構造のバウンダリをゆるめる

名前	一体のチーム
状況	同一組織内にプロダクトオーナーと開発チームがいるが、両者の部署が分かれている状況である。
問題	プロダクトオーナーがボトルネックになっている。
フォース	開発チームの開発スピードと、プロダクトオーナーのバックログの生成と管理のスピードがかみ合っていない。
解決策	プロダクトオーナーと開発チームの歩幅を合わせる。
結果	
課題	
事例	5

名前	組織から改善
状況	開発チームの努力だけで短納期や高品質を実現しようとしている。
問題	短納期や高品質の実現が難しい。
フォース	開発チームの努力だけでは速度は向上できない。
解決策	組織デザインの見直し（部門横断、同一拠点）と、開発基盤の両面からアプローチする。
結果	短納期、高品質が実現できた。
課題	
事例	15

名前	機能集約チーム
状況	自社開発においてビジネス企画と開発チームが別の組織になっている。
問題	ビジネス企画側がボトルネックになっている。
フォース	ビジネス企画側は、開発メンバーに限らず、ビジネス企画を進める上での対外的な調整も必要であるため負荷が大きい。
解決策	開発とビジネス企画の人を１つのチームに集めて機能集約チームを結成する。
結果	
課題	
事例	15

名前	会議体を作らない
状況	チーム内の企画と開発メンバーが同じ島にいるため、いつでも話し合える。
問題	複数の人が集まって合議する会議体を作り時間を設けると、逆に融通がきかない。
フォース	
解決策	会議体を設けずに、その場で必要な時にミーティングを開く。
結果	効率的に時間を使えるようになった。
課題	
事例	15

名前	フラットなマネジメント構造
状況	短納期・高品質が求められている。
問題	ビジネスの企画から開発、リリースまでの全体の時間を短縮したい。
フォース	システム開発のプロセス改善のみで短納期を目指しても限界がある。
解決策	プロジェクト責任者が階層ではないフラットなマネジメント体制を作り、プロジェクトを牽引する。
結果	素早く判断できる。
課題	
事例	15

名前	開発がわかる企画者
状況	これまで企画者は開発のことを感知していなかった。

問題	企画者と開発者が同じ目標を持ち、一緒になって仕事をしないとスピードが上がらない。
フォース	成果が出なければ人事評価や処遇が悪くなる。
解決策	企画者と開発者のコラボレーションの成果を社内評価の対象とした。
結果	企画者が開発者と真剣に関わる姿勢になった。
課題	
事例	15

1.4.6 真の顧客

名前	テストチームも開発に参加
状況	テストを専門に担当するチームが存在する。
問題	仕様を理解したい。
フォース	
解決策	テスターも開発チームに関わった。
結果	
課題	
事例	24

名前	現場見学
状況	これからチームで開発を始めようとしている。
問題	開発者はシステムがどこで使われるかの把握していない。
フォース	システムは持ち運び可能な端末に導入して現場で使われる。開発者は業務内容をあまり理解していない。
解決策	開発者は、開発が始まる前にシステムが使われる現場を見学しておく。
結果	作ろうとしているシステムが、何のために、どのように使われるのかのイメージが持てるようになり、現場に即した提案を開発ができるようになった。
課題	
事例	10

名前	詰所（顧客が自由にシステムを触れる場所）
状況	できたシステムを顧客に公開して実際に触れてもらい、フィードバックを促す。
問題	顧客はじっくりシステムに触れたいが機会がない。
フォース	セキュリティ上の問題があるため公開できない。
解決策	顧客がシステムを自由に触れられるスペースを設けて、そこで自由に触ってもらい、質問があれば開発者に直接聞けるようにした。
結果	顧客がシステムを十分使った上で、フィードバックを得ることができた。
課題	
事例	4

名前	真のユーザとの乖離
状況	スプリントを回しているけれども、エンドユーザが触るのはリリース後である。
問題	ユーザのシステム部門とのコンセンサスは取れているが、エンドユーザが触れるのはリリース後であるため、エンドユーザとの齟齬が発生している。
フォース	
解決策	なし
結果	
課題	
事例	8

名前	全体最適化の欠如
状況	顧客は全社的に業務を効率化したい。
問題	ドメインエキスパートは自身の専門分野に目が向きがちなため、全社的な視点が不足する。
フォース	
解決策	なし
結果	
課題	
事例	10

名前	真の顧客不在
状況	発注元と真の顧客が別部門、または別組織である。
問題	発注元とはコミュニケーションと合意がとれているが、真の顧客とはとれていない。そのため、真の顧客がどのようにシステムを利用するか把握できない。
フォース	
解決策	なし
結果	
課題	
事例	17

1.4.7 プロダクトと、その品質

名前	クオリティを犠牲にしない
状況	顧客とは信頼関係が築けている。顧客から想定以上のクオリティを期待されている。
問題	計画期間内に、予定していた機能ができそうにない。
フォース	期間内に納めようとすると、クオリティを妥協しなければならない。
解決策	クオリティを妥協して間に合わせても、結局手戻りコストがかかり負担になるため、正直に説明して開発範囲を縮小するか期間を延長する。
結果	品質を犠牲にすることなく開発ができる。
課題	
事例	20

名前	シナリオカバレッジ
状況	時間と予算はあるので、更なる品質の向上を目指している。
問題	もっと品質を向上させたい。
フォース	
解決策	ユニットテストのカバレッジよりも、ユーザの操作を想定するシナリオを丁寧に作り、シナリオでシステムの動作を確認するようにした。
結果	使う側の視点に立った品質向上ができた。
課題	
事例	20

名前	プロダクトの煮詰り感
状況	プロダクトの規模が大きくなってきた。
問題	新しいものを生み出すときのわくわくする感覚がない。
フォース	
解決策	なし
結果	
課題	
事例	11

1.4.8 外から進行状況

名前	透明性の徹底
状況	スクラム開発を開始した。
問題	プロダクトオーナーや企画側からは、開発チームの内部状態が把握できない。
フォース	スクラムをもっとも適切に進めるためには、開発状況の透明性確保が大切である。
解決策	リリース計画や品質の管理状況、バーンダウンチャート、バックログなどを徹底的にオープンにした。
結果	経営者をはじめ、関係者の信頼を得ることができた。
課題	
事例	8

1.4.9 納品と調達

名前	分割検収
状況	実際のリリースまではまだ時間がある。イテレーション単位で開発を進めている。
問題	検収を最後にまとめて行うと、想定外の結果になるリスクがある。
フォース	契約は、準委任契約である。
解決策	3 ヶ月毎に分割検収を行い、顧客の了承をもらった。

結果	最後に検収が不合格になるリスクを回避できた。
課題	顧客が時間を取れず受入テストができない。 顧客からフィードバックされても、タスク消化が滞っているため対応できず、顧客がフィードバックへの意欲を減退させてしまう。
事例	4

名前	五月雨納品
状況	デザイン提供業者やコンテンツ提供業者と協力してソフトウェア開発を行っている。
問題	デザイン提供業者やコンテンツ提供業者と協力して仕事を行いたい。
フォース	デザイン提供業者やコンテンツ提供業者は、手戻りが原則なく一括納品の概念があるウォーターフォール型開発に馴染んでいる。例えば、デザインやコンテンツが先行して確定すると変更が困難になってしまう。
解決策	小さい機能の完成する度に順次提供してもらう五月雨式の納品に変更してもらった。検収も、フェーズにわけることにした。
結果	アジャイル型ソフトウェア開発との親和性が向上した。
課題	デザイン提供業者やコンテンツ提供業者に対して、アジャイル型開発の説明をする必要がある。
事例	11

1.4.10 教育・人事

名前	チームを育てる順番と、伝えて行く順番
状況	新人採用が増えている。
問題	新人にアジャイル型開発入門を言葉で説明しても理解されづらい。 アジャイル型開発について書かれている書籍を読めば充分である。 「～とは論」を最初に話してはいけない。
フォース	目前の事に必死になり、「なぜ」を考えることができない状況というのは、エンジニアリング（技術）が不足している場合に陥りがちなため、まず技術を教える必要がある。 アジャイル型開発において大切なポイントやマインドはふりかえりのなかで伝えて行く。
解決策	チームを育てる順番と、伝えて行く順番を変える。 チームを育てる順番：エンジニアリング→改善→価値 伝えて行く順番：価値→原則→プラクティス
結果	
課題	
事例	5

名前	技術スキルで人事評価
状況	技術を駆使したビジネスを展開している。
問題	技術職にあった尺度で人事評価をしたい。
フォース	技術職以外の人事評価尺度をそのまま技術職の評価に適用することはできない。
解決策	仕事で作った成果物の品質をエンジニアが判定して評価する。
結果	技術スキル向上の意欲が増加する。
課題	
事例	1

名前	KPI のデイリー共有
状況	チームはアプリケーションの評価指標(KPI)を定めている。
問題	KPI の状況が不明である。
フォース	
解決策	KPI の状況をチームにメールで毎日送信する。目標を達成したらクス玉を割る。チームメンバーに読んでもらえるようにメールの文面を工夫している。
結果	チームが KPI の状況に気を配るようになり、チームの共通ゴールへの意識が高まった。
課題	

事例	15

名前	学びたいものマップ
状況	チーム内の開発スキルが不足している。開発の効率を上げるためには、習得すべきスキルやツールがある。
問題	何から学ぶべきかの判断がつかない。
フォース	学びたいものが多いが、一度にはできない。
解決策	学びたいものを列挙して、分野事にマップを制作し、その中から必要なものを順に勉強会のテーマとした。
結果	何から学べばよいのかの優先順位が明白になった。
課題	
事例	9

名前	ディレクター、デザイナの勉強不足
状況	エンジニアと一緒に仕事をしている。
問題	エンジニアほど勉強会は実施されていない。
フォース	
解決策	なし
結果	
課題	
事例	1

名前	エンジニアのとりあい
状況	部署全体でアプリ開発にとり組んでいる。
問題	ディレクター間で、優秀なエンジニアの争奪がおきる。
フォース	
解決策	なし
結果	
課題	
事例	15

名前	部署横断の壁
状況	部門ではアジャイル型開発を行っている。
問題	部署横断的に実施する方法がない。
フォース	
解決策	なし
結果	
課題	
事例	25

1.4.11 コミュニケーションの工夫

名前	オンライン・チャットによるコミュニケーション
状況	顧客のオフィスは離れた場所になるが、週に 2～3 日は開発ルームを訪れる。
問題	物理的な距離があるため、確認事項がある場合にもすぐに聞くことができない。
フォース	コミュニケーション頻度を上げたいが、顧客もしくは開発メンバーが間の場所に常駐することは困難である。
解決策	チャットシステムを使って顧客とリアルタイムに連絡がとれるようにした。
結果	頻繁にコミュニケーションをとれるようになった。
課題	顧客の作業時間を奪ってしまう。
事例	4

名前	SNS（ソーシャル・ネットワーキング・サービス）大活用
状況	離れた場所にいる多くのメンバーが開発やデザインに関わっている。
問題	多くのメンバーでのコミュニケーションが難しい。
フォース	全員の状況を把握しようとすると、時間が大幅にかかってしまう。
解決策	プロジェクトメンバー限定の SNS を活用する。 仕様書を Wiki ページ（編集できるウェブページ）に記載する。 ブログを書いて、課題や要求の受付状況を記載する。
結果	

課題	
事例	16

名前	プレゼンタイマー
状況	ミーティングを実施している。
問題	時間超過を避けたい。
フォース	時間のことを指摘されると責められている気がする。
解決策	プレゼンタイマーを使って時間を管理していた。
結果	時間を管理しながら、安心して会議ができるようになった。
課題	
事例	7

名前	言葉を助けるスライド
状況	オフショア先に開発を依頼している。オフショア先のメンバは、日本語が得意ではない。
問題	話し出すと議論が迷子になる。
フォース	
解決策	スプリントレビューの時にスライドを使用する。
結果	話の整理ができ議事録にもなった。議論途中で迷わなくなった。
課題	
事例	13

1.4.12 企画・仕様

名前	ペーパープロトタイピング
状況	開発初期の段階で、デザイナやコンテンツ提供者を含めた企画を行っている。
問題	プロトタイプの作成に時間がかかる。
フォース	プレゼンテーションツールでプロトタイプを作成すると、ツールの操作に時間を取られスピードが低下する。
解決策	テーブルに模造紙を貼って、付箋を用いたペーパープロトタイピングを実施した。画面遷移もわかりやすくなった。
結果	全員がすぐに理解し、デザイン上の合意形成が楽になった。
課題	
事例	11

名前	仕様書とプログラムの齟齬
状況	仕様書を作って、実装していた。
問題	仕様書通りにプログラムが実装されているか妥当性を確認したところ、仕様書は大幅に修正されていることに気づいた。
フォース	
解決策	仕様書はあくまで仮説にすぎないことを学んだので、仕様書作りに時間を費やすことをやめた。 あらかじめ仕様書を作る代わりに、小さく開発して顧客に確認をとりながら進めることにした。
結果	
課題	
事例	5

付録2. 事例とプラクティスの対応

調査した事例とプラクティスの対応を示す。
（○：適用している、△：部分的に適用している、×：適用していない）

No.	プラクティス	A社		B社		C社	D社	E社		F社	G社	
		(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	リリース計画ミーティング	○	△	△	△	○	○	○	○	○	○	○
2	イテレーション計画ミーティング	△	○	○	○	○	○	○	○	○	○	○
3	イテレーション	○	○	○	○	○	○	○	○	○	○	○
4	プランニングポーカー	×	△	○	○	×	○	○	○	○	○	○
5	ベロシティ計測	×	×	○	○	○	○	○	○	○	○	○
6	日次ミーティング	○	○	○	○	○	○	○	○	○	○	○
7	ふりかえり	△	△	○	○	○	○	○	○	○	○	○
8	かんばん	×	×	×	×	×	×	×	×	×	×	×
9	スプリントレビュー	○	×	○	△	○	×	△	△	○	○	○
10	タスクボード（タスクカード）	○	○	○	○	○	○	△	△	○	○	○
11	バーンダウンチャート	×	×	○	△	○	×	○	○	○	○	○
12	柔軟なプロセス	○	×	○	○	○	○	△	△	△	○	○
13	ユーザーストーリー	×	△	○	○	○	○	○	○	○	○	○
14	スプリントバックログ	○	○	○	○	○	○	△	△	○	○	○
15	インセプションデッキ	△	△	△	×	×	△	×	×	○	×	×
16	プロダクトバックログ（優先順位付け）	○	○	×	△	○	○	△	△	○	○	△
17	迅速なフィードバック	○	○	○	○	○	○	×	×	○	○	△
18	ペアプログラミング	×	△	△	△	○	△	○	○	△	△	×
19	自動化された回帰テスト	△	○	△	○	○	○	×	×	○	○	○
20	テスト駆動開発	○	○	△	○	○	×	△	△	○	○	○
21	ユニットテストの自動化	○	○	○	○	○	×	△	△	○	○	○
22	受入テスト	×	×	×	×	△	○	×	×	○	○	△
23	システムメタファ	×	×	×	×	×	×	×	×	×	×	×
24	スパイク・ソリューション	○	△	○	○	○	○	×	×	○	○	△
25	リファクタリング	○	○	○	○	○	○	△	△	△	○	△
26	シンプルデザイン	○	×	○	○	×	○	△	△	△	○	○
27	逐次の統合	○	×	△	×	×	×	○	○	×	×	×
28	継続的インテグレーション	○	○	○	○	○	○	△	△	○	△	○
29	集団によるオーナーシップ	△	○	○	○	△	○	○	○	○	○	○
30	コーディング規約	×	○	○	×	○	○	△	△	△	○	△
31	バグ時の再現テスト	○	×	△	△	△	○	△	△	○	○	○
32	紙・手書きツール	△	○	○	○	○	○	△	△	○	○	○
33	顧客プロキシ	△	×	×	×	○	○	×	×	○	△	△
34	オンサイト顧客	×	×	△	○	○	×	×	×	△	△	○
35	プロダクトオーナー	△	×	○	○	○	○	×	×	○	○	○
36	ファシリテータ（スクラムマスター）	×	△	△	○	△	○	○	○	○	○	△
37	アジャイルコーチ	×	×	△	×	×	○	○	○	○	○	△
38	自己組織化チーム	○	×	○	○	×	○	○	○	○	○	○
39	ニコニコカレンダー	×	×	×	×	△	×	○	○	△	×	×
40	持続可能なペース	○	○	○	○	○	○	△	△	○	○	○
41	組織にあわせたアジャイルスタイル	○	○	×	○	○	○	×	×	○	○	○
42	共通の部屋	△	△	×	×	○	○	○	○	×	○	○
43	チーム全体が一つに	△	×	○	○	○	○	○	○	○	○	○
44	人材のローテーション	×	×	○	○	○	×	○	○	○	○	×
45	インテグレーション専用マシン	○	○	○	○	○	○	×	×	○	○	△

No.	プラクティス	H社				I社	J社				K社
		(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)
1	リリース計画ミーティング	△	×	△	△	○	○	○	○	○	△
2	イテレーション計画ミーティング	○	○	○	○	×	○	○	○	○	○
3	イテレーション	○	○	○	○	×	○	○	○	×	○
4	プランニングポーカー	○	○	×	○	×	○	△	△	×	△
5	ベロシティ計測	△	○	○	○	×	△	△	△	×	○
6	日次ミーティング	○	○	○	○	○	○	○	○	○	○
7	ふりかえり	○	○	○	○	○	○	○	○	○	○
8	かんばん	△	×	×	×	×	×	×	×	×	△
9	スプリントレビュー	○	○	○	○	×	×	×	×	×	○
10	タスクボード（タスクカード）	○	○	×	○	×	○	○	○	○	○
11	バーンダウンチャート	○	○	○	○	○	○	○	○	○	○
12	柔軟なプロセス	○	○	×	○	○	×	×	×	×	△
13	ユーザーストーリー	○	○	○	○	×	×	×	×	○	△
14	スプリントバックログ	○	○	○	○	×	×	×	×	○	△
15	インセプションデッキ	△	△	×	○	×	×	×	×	×	×
16	プロダクトバックログ（優先順位付け）	△	○	○	○	○	△	×	×	×	○
17	迅速なフィードバック	×	○	○	○	○	△	△	△	△	○
18	ペアプログラミング	△	△	×	×	×	○	○	○	○	×
19	自動化された回帰テスト	△	△	○	×	△	○	○	○	△	△
20	テスト駆動開発	△	△	×	×	△	○	○	○	×	△
21	ユニットテストの自動化	△	○	○	○	△	○	○	○	△	△
22	受入テスト	△	○	×	○	○	×	×	×	○	△
23	システムメタファ	×	×	×	×	×	×	×	×	×	×
24	スパイク・ソリューション	×	○	×	○	×	×	×	×	×	△
25	リファクタリング	△	○	×	○	○	×	×	×	△	○
26	シンプルデザイン	△	○	×	○	○	△	△	△	×	×
27	逐次の統合	×	○	○	×	×	○	○	○	×	×
28	継続的インテグレーション	△	○	○	×	○	○	○	○	○	○
29	集団によるオーナーシップ	△	△	○	○	○	○	○	○	△	○
30	コーディング規約	△	○	○	×	○	○	△	△	○	○
31	バグ時の再現テスト	△	△	×	×	×	×	×	×	×	△
32	紙・手書きツール	○	○	○	○	△	○	○	○	○	○
33	顧客プロキシ	×	×	×	×	×	△	×	×	○	○
34	オンサイト顧客	△	○	○	○	○	△	○	○	×	○
35	プロダクトオーナー	○	○	○	○	○	△	×	×	○	△
36	ファシリテータ（スクラムマスター）	○	○	○	○	△	×	×	×	○	○
37	アジャイルコーチ	×	×	△	○	×	×	×	×	○	○
38	自己組織化チーム	△	○	×	○	×	○	○	○	○	○
39	ニコニコカレンダー	×	×	×	×	×	×	×	×	×	△
40	持続可能なペース	○	○	○	○	○	○	○	○	△	○
41	組織にあわせたアジャイルスタイル	○	○	△	○	○	○	○	○	×	○
42	共通の部屋	△	○	△	○	○	○	○	○	×	○
43	チーム全体が一つに	△	○	×	○	○	○	○	○	○	○
44	人材のローテーション	×	△	×	×	○	○	○	○	○	×
45	インテグレーション専用マシン	×	○	○	○	○	○	○	○	○	○

No.	プラクティス	L社				M社
		(21)	(22)	(23)	(24)	(25)
1	リリース計画ミーティング	×	×	○	○	○
2	イテレーション計画ミーティング	○	△	○	○	○
3	イテレーション	○	○	○	○	○
4	プランニングポーカー	×	○	△	×	○
5	ベロシティ計測	△	×	○	○	○
6	日次ミーティング	○	○	○	○	○
7	ふりかえり	△	○	○	○	○
8	かんばん	×	×	×	△	△
9	スプリントレビュー	○	○	○	○	○
10	タスクボード（タスクカード）	△	○	○	○	○
11	バーンダウンチャート	○	○	○	○	○
12	柔軟なプロセス	○	×	×	×	○
13	ユーザーストーリー	△	×	○	○	○
14	スプリントバックログ	△	○	○	○	○
15	インセプションデッキ	×	×	×	×	△
16	プロダクトバックログ（優先順位付け）	△	○	○	○	○
17	迅速なフィードバック	△	×	○	○	○
18	ペアプログラミング	△	×	△	△	△
19	自動化された回帰テスト	○	×	△	△	△
20	テスト駆動開発	×	×	△	△	×
21	ユニットテストの自動化	○	○	○	○	△
22	受入テスト	×	×	○	○	△
23	システムメタファ	×	×	×	×	×
24	スパイク・ソリューション	×	×	○	△	△
25	リファクタリング	△	○	○	○	△
26	シンプルデザイン	△	×	○	○	△
27	逐次の統合	×	×	○	○	×
28	継続的インテグレーション	○	×	△	△	△
29	集団によるオーナーシップ	△	×	○	○	△
30	コーディング規約	△	○	○	○	△
31	バグ時の再現テスト	△	×	△	△	△
32	紙・手書きツール	△	○	○	○	○
33	顧客プロキシ	△	○	○	○	△
34	オンサイト顧客	×	○	×	×	○
35	プロダクトオーナー	△	×	○	○	○
36	ファシリテータ（スクラムマスター）	△	○	○	○	○
37	アジャイルコーチ	×	×	○	○	○
38	自己組織化チーム	○	○	○	○	△
39	ニコニコカレンダー	×	○	×	×	×
40	持続可能なペース	×	○	○	○	△
41	組織にあわせたアジャイルスタイル	○	×	○	○	○
42	共通の部屋	○	×	○	×	○
43	チーム全体が一つに	△	○	○	○	○
44	人材のローテーション	×	○	×	×	△
45	インテグレーション専用マシン	○	×	○	○	○