Your team will be much faster.

エッセンシャル版

Making Agile Development Work

あなたのチームは
まだまだもっと速くなる.

大規模アジャイル開発の真髄

成功するアジャイル開発

大垣伸悟

【エッセンシャル版】成功するアジャイル開発

大垣伸悟：著

この本は縦書きでレイアウトされています。また、ご覧になる機種により、表示の差が認められることがあります。

本書は「成功するアジャイル開発」の1〜3章を抜粋したエッセンシャル版です。
完全版として「成功するアジャイル開発」に全10章が収録されていますので、そちらをお求めください。

まえがき

私がなぜアジャイル開発について説明する本を書こうと思ったのか。

その理由は、アジャイル開発という手法は非常に誤解の多く、ただ導入しただけではすぐにうまくいかない現状があることについての問題意識からです。

「今回のプロジェクトから一気にウォーターフォールモデルから脱却するぞ！」「これからはアジャイル開発だ！」と勢いよく始めたものの、徐々にその思いは失速し、プロジェクト思ったように進まずどんどんと問題が立ち現れては累積し、ついには「このプロジェクトの特性を考えるとウォーターフォールモデルの方が適切である」と言って、いつの間にやらウォーターフォールモデルに逆戻りしていく、という現場も少なくないのではないでしょうか。

こうした問題が起こる原因のうちの一つは、アジャイル開発で最も目立つ特徴のうちの一つ、反復的な開発（イテレーティブな開発）に注目しすぎるあまり、「反復さえしていれ

ばアジャイルである」という思い込みから、「とりあえず反復していればＯＫ」と考え、プロジェクト全体の期間を何回かに分けて反復しているようなスケジュールを引き、アジャイルをやっているつもりになってしまう。つまりはアジャイルとはいったい何なのかについての本質的な理解が不足している状態でありながら、しかし理解したつもりになってしまっている、といった錯覚現象のもたらす弊害があります。

そしてアジャイル開発というものを熟知してくると、アジャイルなプロジェクト、アジャイルなチーム、アジャイルな組織、アジャイルな人材、アジャイルな意思決定、アジャイルなコミュニケーション、といったような「アジャイルな○○」といったキーワードでもアジャイルを説明できるようになります。

しかし本当のところ、アジャイルの本質とは一体何なのでしょうか。

私はソフトウェア開発をうまく成功させたいというニーズに対して、コンサルティング・コーチングを行って、プロジェクトを支援してきました。そしてその中でたくさんのことが分かってきました。

その分かったことを本書を通じて皆さんにお届けすることで、皆さんのプロジェクトが成功するための助けになればと思います。もし皆さんがたった今アジャイル開発に取り組んでいて、すでに何か悩まれていたり、更に上手くやれるのではないかと疑問に思っていたりするのであれば、本書が役立つかもしれません。

ぜひ本書で得られた知識を現場で活かしていただければと思います。

成功するアジャイル開発　目次

まえがき……4

第一章：ウォーターフォールモデルとアジャイルモデルの違いについて……10

なぜいまだに日本ではウォーターフォールモデルが採用され続けるのか……11

ウォーターフォールとアジャイルの違いは「予測に対する態度」……21

どのようにして全体性を損なわないようにすればいいのか……30

システム思考……33

リスク管理……74

モデリング……86

認知科学（社会心理学）……89

第一章のまとめ……98

第二章：アジャイルとは何か――99
アジャイルのメリット――100
アジャイルの世界観――104
スクラム――105
アジャイルへの誤解から生まれる新たな問題――135
第二章のまとめ――141

第三章：アジャイルコーチ・スクラムマスターとは一体何者か――143
チームファーストの精神――144
アジャイルコーチ・スクラムマスターに必要な能力――147
心理的安全性の重要性――156
チームを信頼する――163
速さをもって解決する――165

見たくないものを見せて聞きたくないことを聞かせる――166
アジャイルコーチ・スクラムマスターを目指す人へ――170
第三章のまとめ――173

あとがき――175
巻末付録：スクラムを用いたプロジェクトでの具体的な流れ――182
参考文献――193
著者について――199
書籍情報――201

第一章：ウォーターフォールモデルとアジャイルモデルの違いについて

なぜいまだに日本では
ウォーターフォールモデルが採用され続けるのか

「正しいただ一つの計画」の誤謬とその利点

今日本のソフトウェア開発の現場ではウォーターフォールモデルは人気があるとは言えないまでもいまだに多くのプロジェクトで採用されている開発手法です。特に大規模な開発に採用されることが多く、成功確率のとても低い開発手法でありながら、「正しく計画して、その計画通り正しく作れば必ず成功する」といったような思い込みから採用される状況が続いています。大規模になればなるほど、計画を立てる難易度は上がっていき、そしてその立てた計画通り実行する難易度もさらに高まります。

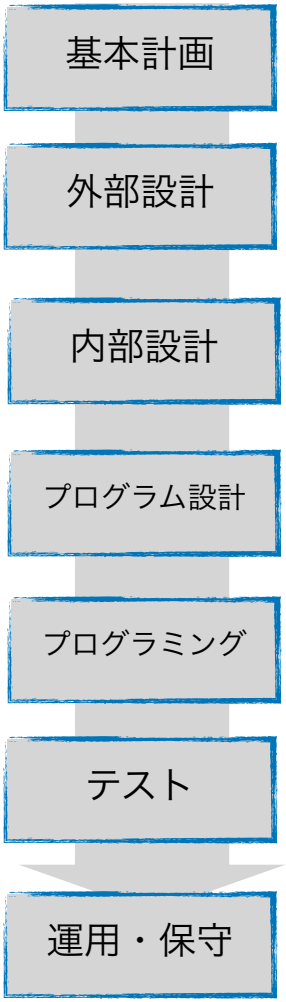
結果的に後から振り返ってみると、現実で起こった様々な出来事が事前の計画に盛り込まれていないということに誰もが気が付きます。そのように計画に盛り込まれていない状況では

計画通りに実行しようとしても現実と計画の間にギャップがある限り計画通りに実行することはできなくなってきます。正しい計画を立てればよかったのでしょうか。いえ、問いをもう少し正確にするとすれば、「どの程度にまで正しい計画を立てればよかったのでしょうか？」大規模になればなるほど、ほんの少しの事前の予測のずれが、プロジェクトの後半になっていくほど影響が連鎖して軌道修正ができなくなるほどは現実と計画が乖離してきます。2年越しのプロジェクトの細部で何が起こるかまで事前に全てを見通せる人などいるでしょうか。ソフトウェア工学の世界ではプロジェクト管理のあり方として、いかに正しく予測をして、それらを詳細に計画に盛り込むか、というテクニックというものについて研究し、それらのテクニックを洗練させようと努力をしてきました。それらのテクニックについて考えること自体は学ぶことも多くあります。しかしそれらの多くのテクニックを学んだところで、プロジェクトの成功確率は大して上がらないのです。

ではなぜ未だにこのウォーターフォールモデルが採用され続けるという現実があるのでしょうか？

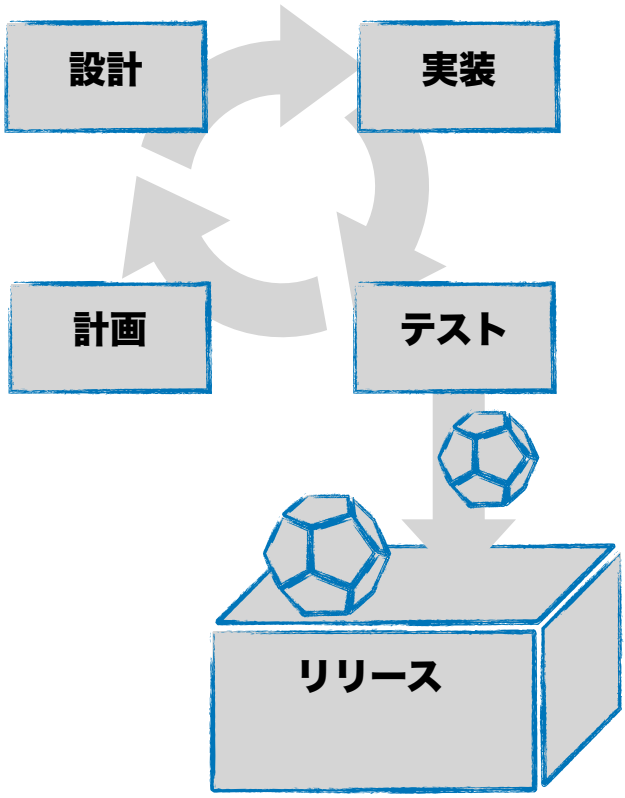
ウォーターフォールモデルの流れ

上から下に滝が流れるように次工程に流れていく
原則前工程に戻ることを認めない



アジャイルモデルの流れ

小さく速く作りユーザーに製品を届け、反復しながら機能追加を行っていく



日本のＩＴ産業の構造的な問題

日本のＩＴ産業というのは、親会社が大企業のソフトウェア部門的な会社が牽引してきました。これらの会社はもともとは親会社の中でソフトウェアを作る部門が、自社のために作るだけではなく他からもソフトウェア開発の仕事を受注するようになるために子会社として独立し、それによってコスト部門から利益部門への転換を図った結果でもあります。ソフトウェアを作るというのはそのほとんどが人件費です。自社のために専属で開発をし続けるソフトウェア開発部門というのは大きなコストを抱える部門になります。そこで他からもソフトウェア開発の仕事を請け負うことで人件費以上の売上を立て、利益を上げる組織に変わろうとしていったのです。

日本国内には親会社の名前がついた比較的大きなシステム開発会社、ソフトウェア開発会社が多くあります。そうなってくると何かのシステムを構築する、という一つの区切りで、仕事が発注されることになります。親会社が大企業ともなると上場していたりしますから、年間でどのぐらいの予算を使うのか、あらかじめ計画を立てて公表する必要があります。そ

してシステムを構築するために必要な諸々を、発注をする最初のタイミングで決める必要があります。これらのシステムを構築するのにかかる費用、期間を決める必要があるのです。

大抵、大規模なシステムの構築の場合、事前に調査をし、要件をまとめあげて、どのくらいの規模になるのかを概算で見積もります。そしてここで発注金額が決まると、後はその発注金額内で収まるように開発者が招集され開発スケジュールが決まります。

ここまでくるともうお分かりかもしれませんが、ウォーターフォールモデルはこうした日本のＩＴ産業の構造の中でビジネスをしている上では、発注がしやすい仕組みなのです。長期間に渡って行われるプロジェクトの必要な費用を最初に決めてしまうのですから、発注を受けたソフトウェア開発側は予めまとまった開発資金が手に入ることになりますし、発注をする親会社側は、何度も何度もソフトウェア開発費用を修正して公表し直すという手間が省けます。

私もこうした場面に居合わせたことがありますが、まだ誰が開発するのか、実際の開発者が決まっていない状況なのにも関わらず、発注金額が決まってしまったことで自ずとその他

の要素である、開発の成果物である製品の機能や開発したシステムが使えるようになるまでの納入日・稼働する日、などがなぜか決まってしまい、そのまま進んでいくという雰囲気になっていきます。

概算で見積もった規模が元となり、合意が形成されていくのです。受注をした時は楽観的ですが、実際に開発プロジェクトが始まって計画通りにいかないということがわかり軌道修正も困難となると、とても悲観的な雰囲気になってしまいます。

ウォーターフォールモデルは始めるのは簡単で終わるのが難しい

実際にプロジェクトも後半に差し掛かってくると納期までに開発が終わらないということが分かると、親会社にそのことを伝え追加の費用の請求をします。プロジェクトが延長されるわけです。そして追加の人員も増員されていきます。プロジェクトの規模がさらに大きくなります。さらに複雑になってきます。開発効率は著しく落ちていきます。そしてまた追加の

請求をする、ということを何度かやり、実装する機能を最終的に削ったりしながら、なんとかプロジェクトを終わらせます。予算オーバーをしたら、おかわりをするわけです。

なのでこうした追加の請求、おかわりができる状況にある子会社ソフトウェア会社はウォーターフォールモデルでもなんとかやっていけるわけです。もしおかわりを認めなければシステムは完成せず、今まで投じた資金が泡となって消えてしまいます。となると、おかわりに応じないという選択肢はないと言っても過言ではないです。

これが親会社とソフトウェア子会社の間だけの問題ならいいのですが、開発するのに投じたコストに見合っただけの価値のあるシステムが出来上がらないとなると、そのシステムを使う親会社は競争力を失います。そして何より開発現場の開発者達は疲弊していきます。それでも また次のシステムを構築する際には、同じような事が繰り返されるわけです。

ウォーターフォールモデルで作られたシステムに大した価値がないという意味ではなく、システムが出す価値以上に開発コストがかかりすぎているということです。

それに対してアジャイルモデルは、始めるのが難しく終わるのは簡単という特徴があります。

始めるのが難しいという理由ですが、アジャイルモデルではシステム全体の規模や、最終的に必要となる機能などを洗い出してそれらを必要となる費用の見積もりに使うという事はしません。将来的にどのようなシステムになるのかというビジョンとして用いることはあっても、実際に開発してみてユーザーが使える状態にならないとシステム自体の価値は分からないし、それを作るのにいったいどれぐらいの時間がかかるのか、というのは実際に開発してみないとわからない、というスタンスです。とても現実的です。

開発するシステムが同じでも、開発する開発者が違うだけで開発にかかる期間も違えば、開発者の人件費も変わってきます。となると最初に合意を形成するのが難しく、ウォーターフォールモデルでの受注に慣れている人達からすると、何も決めずに始めようとする、という風に見えるのです。

何も決めてないので始められない、どう始めていいかわからない、となるわけです。ですので、最初からいきなり大規模なプロジェクトにしてしまうのではなく、最終的には大規模なプロジェクトになったとしても、スタートはスモールで始める、資金も少額で始める、開発

する人員も少人数で始める、という形を取ることによって、スタートしやすくする必要があります。

ウォーターフォールとアジャイルの違いは「予測に対する態度」

ウォーターフォールモデルの予測は「天気予報型」

ウォーターフォールモデルでは予測というのは最初でありかつ絶対的なもので、予測を基にして計画を立てます。予測に対して確実に結果を得られるように入念に練り上げた計画をつくります。そしてそれをそのまま計画通りに実行するということを考えます。

各工程は次のステップにパスし続けますので、基本的に前行程に戻ることは原則的に認めていません。ただ実際には前行程に戻るということも妥協としてあり得ますので、あくまでもウォーターフォールモデルが手法として想定している流れとして、一方向しか認めていないという意味です。これがウォーターフォールモデルの基本スタンスとなります。

この場合の予測はかなり特殊な予測です。私はこれをまるで天気予報のようなので、『天気予報型予測』と呼んでいます。予測をすることによって得られた何かしらの情報を、どのように使ってとしても、結果に影響を与えることはないというスタンスです。

例えば、天気予報で明日の天気が雨と予測をして、誰もが一斉に傘を持って出かけたとします。街では誰もが傘を持っている状態ですが、それによって天気が雨以外に変わるかというと変わりません。みんなが傘を持ったとしても天気は晴れません。天気は変更されません。天気自体は影響を受けないのです。これが天気予報型です。

予測することが結果に対して影響することはない、と考えるのが天気予報型です。

アジャイルモデルの予測は『株価予想型』

それに対して、アジャイルモデルにおける予測は『株価予想型』です。

例えば有名なアナリストがA社の分析をして、影響力のあるメディアで「A社の株は上がるでしょう」という予測を発表したとします。

するとそれを観た人たちが、一斉にA社の株を買った場合、実際にA社の株は上がってしまいます。予測をしてその予測で得られた情報を用いたことによって結果が影響を受けてしまう、それがこの『株価予想型』になります。

ソフトウェアビジネスにおいてソフトウェアを作って、それをユーザーが使い始めたときにユーザーの業務が変化します。それはソフトウェアが価値をもたらした結果であって、逆に言えばソフトウェアを導入したことによってユーザーの業務が一切変わらないとしたら、それはそのソフトウェアに価値はないということですから、ソフトウェアに価値がある場合は必ずユーザーの業務は効率化されたり良い方向に向かうはずです。変化をするわけです。するとその変化がユーザのビジネス環境、あるいは業務を行っている、より大きな意味でのシステムは変更されます。それによって別の箇所に歪みが発生したり、あるいは新たなビジネスニーズが生まれたりします。

するとそれによって当初考えていた予測が外れるということが起こります。あるいは当初の予測が変更を加えられることになります。私たちが生きているこの世界では、予測というの

は結果に対して影響を与えている場合がほとんどである、ということを理解する必要があります。

それがアジャイルの基本スタンスです。

ウォーターフォールモデルでは予測からスタートしていきましたが、アジャイルの場合は予測からスタートはしません。

予測はしますが予測を完全に練り切った上で行動しようとするのではなく、まずは実行します。実行によって結果が生まれますから、ユーザーに対する価値が生まれたのか、あるいは業務に変更が入ったのか、その価値を観察します。そしてその観察によって得られた情報から学習します。

この『学習』によって得られた情報をもとに『予測』を立てます。その『予測』から『計画』を立てます。そしてまた『実行』して、というループを繰り返すんですけれども、『学習』によって『予測』をした場合、もともとの当初の立てていた『予測』とその後の2度目の

ループで学習したことによって得られた『予測』というのは変わってきますから、変更を加えます。ここで変化を取り入れることになります。

実際の予測がどのようだったかに拘らず、結果を重視して「実際にどうなのか」、「現実の世界はどうなのか」に対してフィットさせようというのが、アジャイルの考え方です。

一年後の株価を予想して何かを行動に起こそうとせずに今の株価と過去の株価の実績を参照するように、今この瞬間の現実はどうなのか、というのを最重要視して、結果あるいは現実と実績などから開発する計画を組み立てていく、というのがアジャイルになります。アジャイルでの予測というのは幅があるものになります。最低から最高までの範囲があるものがアジャイルにおける予測の結果です。そして時間が流れていくと、徐々にその幅は狭まっていきます。予測に幅があるのであれば、計画にも幅があります。アジャイルでは計画を立てないと思っている人もいますが、たった一つの固定された計画を立てるのではなく、幅のある計画を用意するようにします。たった一つの計画では役に立たないので、もしたったひとつの固定された計画しか用意しないのであれば、それはもはや計画が無いのとあまり変わ

らない、というスタンスです。むしろたったひとつの固定された計画があることの弊害の方が大きいので、そんなものは捨ててしまえとまで言い切ってしまう人もいるくらいです。

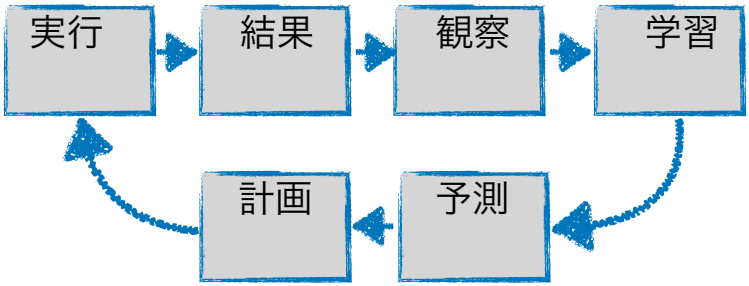
この予測に対する態度・基本スタンスについて、計画駆動型とアジャイルでは大きく異なります。

ソフトウェアビジネスの現実は、株価予想型の方がその現実に近いのですが、計画駆動型は天気予報型の予測であるにも関わらず、そうではないソフトウェアビジネスにそれを適用しようとしてしまっているという点でも、色んなところに歪みを生じさせることになります。

なのでウォーターフォールモデルが、単にうまくいかない、失敗する、納期に間に合わない、予算内に収まらない、品質が担保できない、ビジネス要件の変更に対処できない、というこれらの問題の多くは実はこの予測に対する態度、天気予報型を用いてしまっている、というところにあります。

このアジャイルではこの開発フローだけで対処できるかというと、実はこのアジャイルにも欠点があります。

計画駆動：ウォーターフォールモデル



アジャイルモデル

株価予想型

それは何かと言うと『全体性を損ね易い』ということです。

全体性を把握することの重要性を意識する必要がある

特にスクラムでは「スプリントという期間に区切って小さく早く作りましょう」「ユーザーがすぐ使えるものを提供しましょう」という基本スタンスですので、作るものは「なるべく小さく早く作る」、そして「ユーザーのフィードバックを早く受ける」、それを第一優先度においてどんどん作り始めていくわけですが、システムというのは全体性が重要なので、部分としては完成して、機能が提供できてユーザー使えるものができたとしても、それらを組み合わせていく過程で、あるいは組み合わせた結果、システム全体としては当初期待していた挙動とまったく違う挙動になってしまう場合があります。

これはシステムというもの自体の特性でもあります。ある特定の条件下においてはうまく挙動するが、少し特殊な状況あるいは当初想定していなかったシチュエーションでは全く想定していなかった挙動になってしまう、ということが往々にしてあります。

それがシステムの特性でありシステムとしての価値でもあるのですが、それらをこのアジャイルでは見失いやすいのです。そのため、アジャイル開発をより良いものにするためにはさらに工夫が必要があります。全体性をいかに損なわないかという視点が重要になってきます。

どのようにして全体性を損なわないようにすればいいのか

全体性を捉え続けるためのテクニック

開発が進んでいき、システムの規模が大きくなっていくと、把握しなくてはいけない範囲も広がり、そして様々な情報が付帯されていきます。システムの細部まで正確に把握しようとすること自体は良いことではありますが、現実的に考えると人間の脳が把握できる情報量というものには限りがありますので、細部を完全に把握し切ろうとするスタイルでは、ある一定規模を超えた時に、急激に疲弊していくという事態を招くことになります。そうならないために重厚なドキュメントを作成する、という対処法を用いるプロジェクトもあったりしますが、その場合も同様の現象が起こります。

全体を把握するために作られたドキュメントが増えて過ぎてしまい、結局は全てのドキュメントを把握している人が存在しない、という現象を生み出し、ドキュメント自体が形骸化していしまうことにも繋がります。ですので、全体性の把握に関しては、細部まで詳細に把握

したい、という欲求を抑えて、いかに抽象化し細部をうまく省略するか、がキモになります。

そしてこのうまく抽象化をし省略するためのテクニックをマスターするために、今から紹介するいくつかの領域から知識を得て、理解を深めていくのが有効です。

実際に私はアジャイル開発に取り組むチームにコーチングする際に、次の四つの領域の知識について紹介をし、それを実際の開発現場でうまく使えるようにレクチャーすることにしています。

- システム思考
- リスク管理
- モデリング
- 認知科学（社会心理学）

これがそれぞれの領域に関して後ほど解説をしていきたいと思いますが、どれもが専門的で高度な分野ですから、それぞれから掻い摘んで美味しい部分だけでもいいので、つまみ食

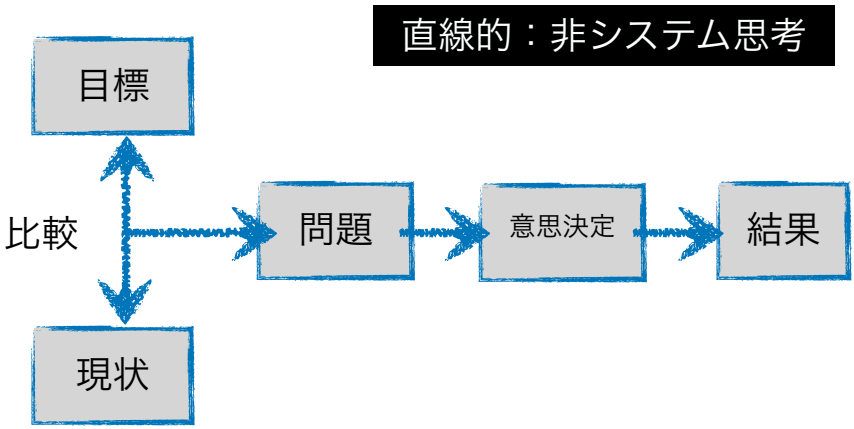
いするだけでもいいと思います。実際にそれらをどのように応用すれば、上手く使えるツールとして役立たせることができるのか、という手本を見せたりしながら、そのプロジェクトの状況をうまく活用して、具体的な事例を示しながらそのテクニックを提供したりもします。

システム思考

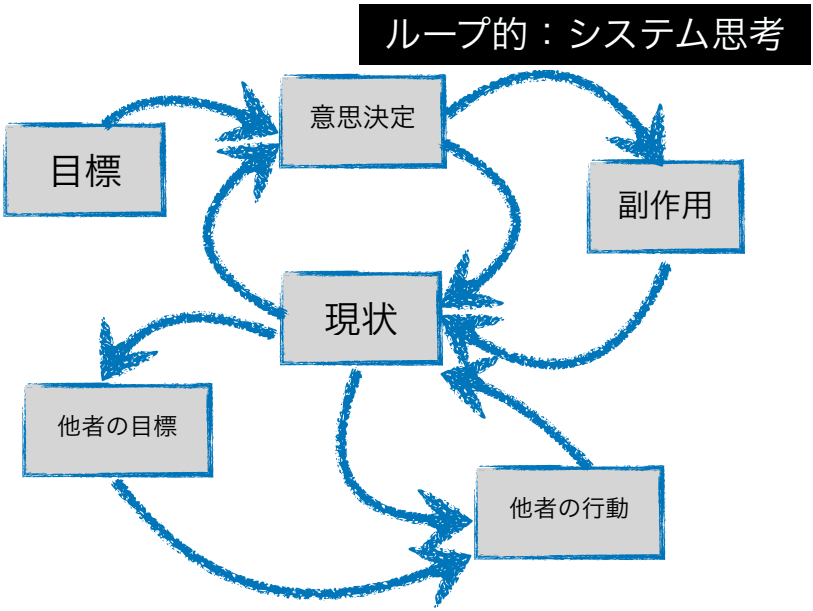
システム思考は、システムダイナミクスやビジネスダイナミクスなどと言われたりもしますが、今から六十年以上も前にMIT（マサチューセッツ工科大学）で創始されて以来、生理学から財政政策、組織変革から気候変動まで幅広い課題で応用され成果を上げています。システム思考とはさまざまな要素の複雑なつながりをシステムとして捉えて、構造の全体像を俯瞰してその複雑な挙動を理解して、システムそのものの改善を図るものの見方です。

アジャイル開発において、それぞれのイベントや手法、ルールなどを融合させて一貫性のある体系を構築するためにもシステム思考は重要です。新しい技術や手法が次から次に出ては消えていくこのソフトウェア開発の世界において、バラバラの目新しい戦略や組織論、興味を引く華々しいアーキテクチャの最新の流行に踊らされてばかりで、有機的な結びつきを得られないという問題を回避することができます。システム的な視点がないとそれらがどのように相互に関連し合っているのかに目を向けようとする意欲も起こりません。

出来事に着目している世界観での解決のアプローチ



フィードバックに着目している世界観での解決のアプローチ



システム思考は「全体は部分の総和に勝る」ということを絶えず私たちに思い出させてくれます。システム思考なしにビジョンを持つと、未来についてバラ色の絵を描くことになってしまうでしょう。多くの組織が、ビジョンが重要であると啓蒙され、「ビジョンを描くという流行」に流されていきましたが、高い目標を設定するだけではその運命を好転させることはできないということを痛感してきました。非システム的な思考が支配的になると、口では「我々はビジョンを実現できる」と言うかもしれませんが、目の前の現実は「他の誰かが作り出した状況だ」と思い込むようになり、いわば自己擁護に走り自己欺瞞的になっていくのです。

複雑なシステムというものは、外部からそのシステムの一部分だけに干渉しようとすると、ほぼ確実に他の全く別の部分で予期せぬ破壊的な現象を引き起こすリスクがあるという特徴があります。システム全体を理解せずして介入をすると、その介入の意図に反して、思わぬ挙動をシステムから見せつけられることになります。

私たちはよく期待している結果に対して、違った結果が出てしまうことを、副作用と呼んだりします。そしてシステムには副作用がつきものであるかのように受け止めます。私たちが行動を起こすと様々な作用が生じます。予測されている作用や有益だと思われる作用をシステムの価値だと考えたりもします。実際には副作用があるのではなく複数の作用があるだけであり、予想していなかった作用や、システムそのものに悪影響を与えたりする作用を副作用と呼んでいるに過ぎないのです。通常、副作用と呼んでいるものは、単に私たちがシステムに対する理解の限界を表しているに過ぎないのです。

システム思考が重要なのは、複雑なシステムというものは、原因と結果が時間的にも空間的にも遠く離れた状態が作られてしまうことによって引き起こされる作用の連鎖に対する理解をもたらすことにあります。

多くの場合、私たちがシステムの挙動に驚かされてしまうのは、原因と結果が常に近くにあると思い込んでしまうことにあります。複雑なシステムというのは、原因が発生してから何らかの結果が出るまでの間に、時間的な遅れを伴ったり、作用が増幅されたり、元に戻ろ

うと抵抗したりします。そうしたシステムの特性を理解しておくと、あらゆるものがシステムであると考えられるようになり、開発の対象物であるソフトウェアプロダクトやサービスデザインだけでなく、実は組織や社会というものもある種のシステムなのであると捉えられることができるようになります。

アジャイル開発というメソッドそのものが、システム思考との親和性が高く、開発プロセスそのものもシステムであると考えられるようになることによって、見せかけのアジャイル開発から脱却し、本物のアジャイル開発を実現できるようになるでしょう。ウォーターフォール開発では、このシステム思考と親和させることが難しく、システム思考ができるようになってくると、ウォーターフォール開発ではなぜ成功確率が低いのか、という問いに対しても、答えられるようになるでしょう。

システムの構造を理解せずにむやみに介入をして挙動を変えようとすると、システムは抵抗します。私たちは一般的に、「在庫が増えすぎている」とか、「今月は人件費がかかりすぎている」などといった問題があった場合に、すぐ近くの原因を探ろうとします。全ての出

来事に原因があり、その原因をまたさらに別の原因の結果だと考えてしまいます。こうした出来事に着目した世界観は、問題を解決する際にも出来事に着目したアプローチを生み出してしまいます。まず現在の状況と目標と比較します。その目標と現在の状況とのギャップを問題だと定義します。

例えば、目標が「今月リリースする機能は５０ポイント分」だったとします。現在の状況は３５ポイントでした。問題は「目標よりも１５ポイント低かった」となります。そうすると開発者を残業させまくってでもリリースできる機能を増やそうとするかもしれませんし、開発者の数をチームの人数を３０%増やすかもしれません。あるいは他に別の解決策が思いつくかもしれません。その中で一番有効だと思われる選択肢を選んでそれを実行することで期待する結果を得ようとします。実際にそれで目標に到達するかもしれません。それで問題解決したかのように思えます。しかしシステムはその解決策に反応します。開発者残業させまくれば開発者は疲弊していき 生産性は下がり、翌月にリリースできる機能は減ることになります。あれは開発者の数を増やしたことで、引き継ぎや情報共有に時間がかかったり、コミュニケーションコストが増大したことによって、システムの全体を把握すること

もなく、むやみやたらと手当たり次第に多くの機能をつけただけの製品がリリースされてしまい、ユーザーから多くのクレームを受けるという問題を生む恐れもあります。

このような直線的な考え方が、「今日の解決策が明日の問題を生む」ということを引き起こします。こうしたシステムの抵抗が起きるのは、システム内で起こる様々なフィードバックを正しく理解できていないためであって、私達が良かれと思ってシステムに介入したその行為がシステムの状態を変更するため、バランスを取り戻すためにシステム反応するのです。そうした私たちのシステムに対する介入が副作用を引き起こすということが起こります。

私はこのシステム思考を実際に体験してもらうためにビールゲームというものを実施することがあります。自分たちの意思決定の結果がシステム全体に対してどのように思わぬ挙動を引き起こしてしまうのか、私たちの意思決定がいかに近視眼的に行われているのか、について実感してもらいます。複数のチームを作って競い合うようにゲームをしますので、参加者は皆各々でゲームを楽しみながらも、自分たちの意思とは真逆の動きをしていくシステムに驚かされ、焦りながら、システムの特性というものを学習していきます。ゲームに勝とうと

してシステムに積極的に介入して調整しようとしますが、なぜそのような挙動をしているかの理解がないうちは、介入すればするほどシステムの挙動は望ましくない方向に暴走していくことを経験できます。

それぞれが実は「自分を正しいことをしている、間違っているのは他のチームメンバーだ」と思い始めるようになりますが、ゲームが終わった後それぞれに感想を聞いてみると実は全員がそのように思っていて、中には「この人たちはバカなんじゃないか」とまで思っていたメンバーもいたりするほどです。メンバーそれぞれの意見を聞くと、互いに他のメンバーを責めることができないということがわかり、実はチームメンバーに問題があるのではなく、システムそのものに問題があったのだということに気づくことになります。このゲームで学べることは、システムというものは介入をした結果となる作用が遅れて増幅されて出てくるということです。この時間的遅れというものが、私たちの意思決定を狂わせるのです。

これはシステム自体が抱えている潜在的な問題でもあるのですが、同時に私たち人間の持つ認知能力の限界でもあると言えます。私たちは時間的・空間的な範囲をそれほど広く持って

意思決定できないのです。原因と結果が、時間的にも空間的にも遠く離れていた場合、そこに因果関係を見出すことは難しくなります。特にシステムの場合、介入とその作用に時間的な遅れが本来は存在するのですが、介入に対する作用が繰り返し現れるので、私たちはたった今行った介入とその直後に現れた作用に因果関係があると取り違えてしまいます。実際に今みた作用はそのずっと前に起こした介入が原因になっているのですが、連続して介入を行っていると作用も連続して現れることになり、どの介入がどの作用に影響を与えているのかが正しく把握できず、つい直近の介入が直後の作用に影響を及ぼしていると取り違えてしまうことになります。私たちの認知能力では、因果関係をとても狭い範囲でしか捉えられないのです。時間的に近いところ、空間的に近いところでしか、因果関係を見いだすことができないのです。そのためシステム思考を持つことは、こうした問題を改善できる助けとなります。

システム思考で重要なのはそのシステムの構造を解明することです。なぜ構造への理解が重要かと言うと、それを持ってしか挙動パターンそのものを変えるレベルで、挙動の根底にある原因に対処することができないからです。構造が挙動を生み出します。すなわちシステム

思考とは構造的な思考法と思います。挙動の構造的な原因を見つける能力を磨く思考法です。

システム思考を実践して活かすために有益ないくつかのツールを紹介いたします。パターン、構造、挙動をあぶり出し、表現するためのツールです。これらのツールでは、捉える時間軸を広げ伸ばしたり、全体的な流れやパターンを把握したり、前提を問い直したりする効果があります。

システム思考を実践するのに役立つモデル群

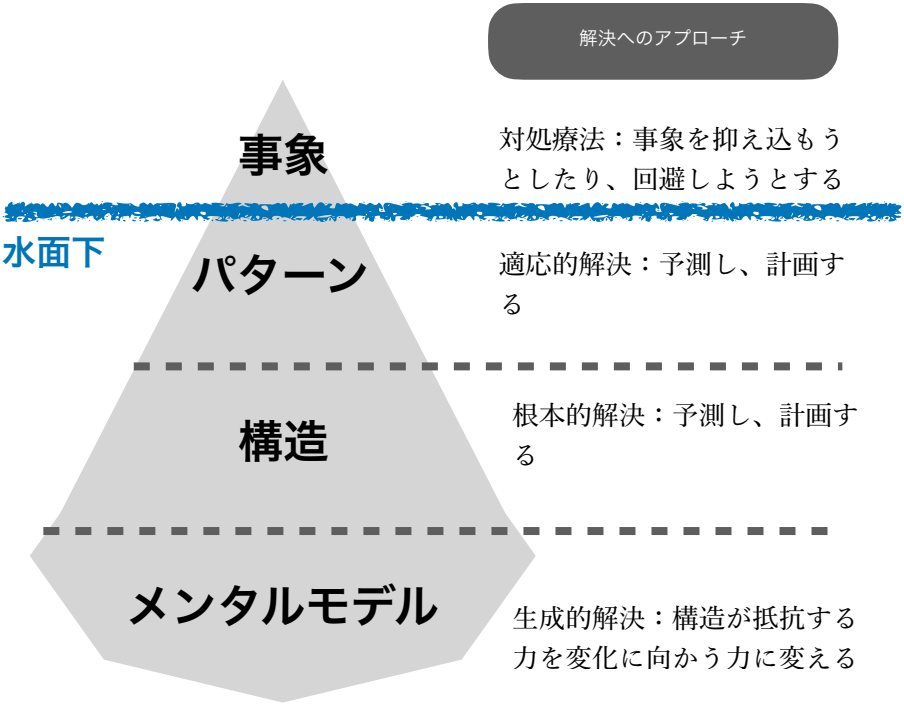
- 氷山モデル
- ループ図
- ストック／フロー図
- システム原型

ではそれぞれのモデルについて説明してみたいと思います。

氷山モデル

氷山モデルは、根底に潜む因果律を考えるための表現方法です。この氷山モデルでは、物事を見る際に、水面上に表出した氷山の一角をみて判断するのではなく、水面下に隠れた部分を意識し考えることを示唆してくれます。

氷山モデル


解決へのアプローチ
事象
対処療法：事象を抑え込もうとしたり、回避しようとする
水面下
パターン
適応的解決：予測し、計画する
構造
根本的解決：予測し、計画する
メンタルモデル
生成的解決：構造が抵抗する力を変化に向かう力に変える

「事象」は私たちが認識することのできる出来事や情報であり、その「事象」はその下にある「パターン」による挙動の結果であり、その「パターン」はその下にある「構造」で決まります。そしてその「構造」自体も、その下にある関連する人たちの「メンタルモデル」が自ずと生み出していたりします。この氷山モデルを用いる場合のメリットは、表出している事象に議論が集中して、根本的な解決について意識が向かない状況にある場合に、メンバーの視野を広げ、見えていない水面下の状況を想像させることにあります。

具体的なシチュエーションを例に考えてみましょう。

リリースしたアプリケーションにてバグが出て、ユーザーからクレームがきたとします（「事象」）。これ自体に対応を行おうとすると、謝罪してバグを直すことを約束する（「対処療法」）となります。

そしてこの「事象」の下にある「パターン」は、ある特定の機能のバグの発生率が高いということがわかりました。これに対応するとなると、バグの発生率の高い特定の機能のテストをより多く時間をかけて実施する（「適応的解決」）ということになります。

さらに「パターン」の下の「構造」について考えてみましょう。ある特定の機能のバグが多い理由は、テストケースに漏れが多く、質の高いテストケースが考えられていないこと、そしてそのテストケースが考えられていない理由は、開発中に機能要件が何度も変更されており、その要求に対してテストケースが対応しきれていないということが分かりました。機能要件が変更されたことがリアルタイムに共有されないコミュニケーションの構造になっていたのです。要件を管理するチームとテストケースを作成するチームが別になっており、このチーム間のコミュニケーションが週に1回のミーティングでしか情報共有を行っておらず、1週間の間に変更が入るとそれがテストケースを作るチームに知らされるのは最大で1週間後という遅れが発生してしまう構造なのでした。ここに対策を講じるとすると、要件管理をするチームとテストケースを作成するチームを分けずに一つのチームにする、あるいは、リアルタイムに情報が共有されるようにチケット管理システムなどでの要件変更の管理をチー

ム横断で行う、あるいはテストケース作成に要件を管理しているチームもレビュワーとして加わる、などが考えられます（「根本的解決」）。

ここまでで、表出した問題は解決されそうですが、更にその下の「メンタルモデル」にまでアプローチを深めますと、要件定義を管理するチームとテストケースを作成するチームが対立しがちな心理状態であったことに着目します。要件を管理するチームは、要件の中での言及漏れ、要件間の矛盾点、定義文章の曖昧さ、などについてテストケースを作るチームから質問を受けると、「重箱の隅をつつくように粗探しされている」と感じていることが分かりました。テストケース作成をするチームは、「テストケースに漏れがあって怒られるのは自分たちなので、事細かに確認したいだけだ」と考えていることも分かりました。そのため、要件定義を管理するチームは、要件が確実に確定となるまでのギリギリまで情報を共有せずに、確実に確定といえるものだけをテストケース作成チームに共有していました。そのため、情報の共有が遅れ、そしてそこから質問が始まるので、時間がかかってしまい、テストケース作成に十分な時間がかけられていないことが起こっていました。この場合の対応は、要件を管理するチームの評価が「顧客にどれだけの『価値』をもたらしたか」となっていた

部分の『価値』という箇所の対象の中に、『仕様バグを出した場合の被害・損失』を含めるようにし、『仕様バグの発生確率』をモニターして『仕様バグの被害 × 仕様バグ発生確率』を『仕様バグ混入によるリスク』として、『価値』の計算の中に含めるようにします。そしてテストケース作成チームと協力することが要件定期を管理しているチームの評価を上げるためには必要である、というメンタルモデルに変更します（「生成的解決」）。そしてどのようにすれば互いに協力できるのか、を考えるように仕向けることで、情報共有の仕組みを変えることに着手します。

バグのクレーム

解決へのアプローチ

事象

バグが出てユーザーからクレームがきた

対処療法：謝罪してバグを直すことを約束する

水面下

パターン

特定の機能のバグの発生率が高い

適応的解決：バグの発生率の高い特定の機能のテストをより多く時間をかけて実施する

構造

機能要件が変更されたことがリアルタイムに共有されない

根本的解決：要件管理チームとテストチームを一つのチームにする、要件変更の管理をチーム横断で行う、テストケース作成に要件を管理しているチームもレビュワーとして加わる

メンタルモデル

要件定義を管理するチームとテストケースを作成するチームが対立しがちな心理状態

生成的解決：テストケース作成チームと協力することが要件定期を管理しているチームの評価を上げるため

この氷山モデルの使い方ですが、1人が一気に完成させてしまうのではなく、ぜひ関係者と一緒に氷山モデルを書き出してみてください。みんなで氷山モデルを書き出すその効果は、表面的に現れた事象にだけ対処する対処療法的な対処療法的な対応になりがちな考え方を変えるきっかけになることにあります。

問題を俯瞰して眺めながら、見えていない奥の構造やメンタルモデルにまで視点を深めていく過程で、自分たちが対峙している問題というのは、実はシステム的な問題である、ということに気づけるでしょう。氷山モデルはモデルそのものの見た目の分かりやすさから、比較的受け入れられやすい図であると言えます。システム思考に馴染みのないメンバーに対して、システム思考の必要性・有用性を説く際に、氷山モデルから始めるのは導入をスムーズにしてくれますので、おすすめです。

ループ図

ループ図は、パターンに関係する様々な要素を因果関係を示す矢印でつなげていきます。それによって、そのシステムの挙動がなぜ起こるかを説明する構造を可視化します。フィードバックプロセスによって循環構造になる流れに注目をして表現します。

最初は元から見えていた要素をつなげてつながりを表現していきます。そしてそこに見えていなかった要素を足してつなげていきます。そうやってどんどん足していくことで、時間の経過とともにシステムの挙動がどのように変化するのか、を表現します。

レバレッジポイント（＝システムの挙動に最も大きな影響を与えている箇所）はどこにあるのかを見つけ出すことで、システムに望ましい変化を起こさせるための介入の仕方を考えることができます。ループ図では、自己強化型ループとバランス型ループがあります。自己強化型ループは変化が強化あるいは加速されていくフィードバックプロセスを持つ構造です。好循環、悪循環、成長、衰退、などがあります。バランス型ループは変化がいづれ打ち消されるフィードバックプロセスを持ちます。安定、バランスなどを生み出す構造です。バラン

スをうまくとるまでに時間的な遅れがあるシステムというのは挙動に揺れが発生します。その揺れの規則性が複雑な場合、不規則な揺れに見える場合もあります。

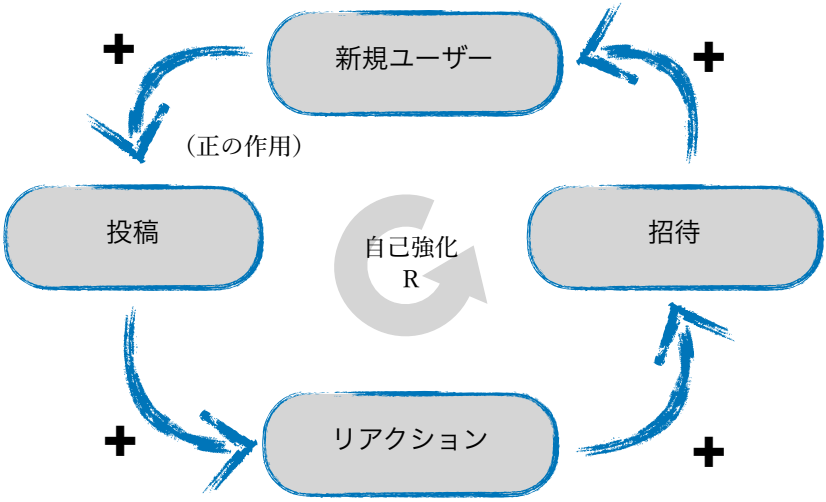
自己強化型ループでシステムが急成長した後にバランス型ループにうまく移行することで、システムは崩壊を免れます。うまく移行できなかった場合はシステムは崩壊してしまいます。無限に成長することはできないからです。稼働中のシステムの行く末を予測する場合に、このままいくとシステムが崩壊してしまうかもしれない、ということを見つけ出すことで対策を打てる可能性が出てきます。あるいは過去のすでに崩壊してしまったシスデムの崩壊の原因を特定するのに有効です。

自己強化型ループがある場合、望ましくない結果を生む方向にループしている場合であれば、構造そのものを変更する必要があります。構造そのものを変更せずに結果の方向を変えようと挙動に干渉しても、望む結果を生み出すシステムにはなりません。

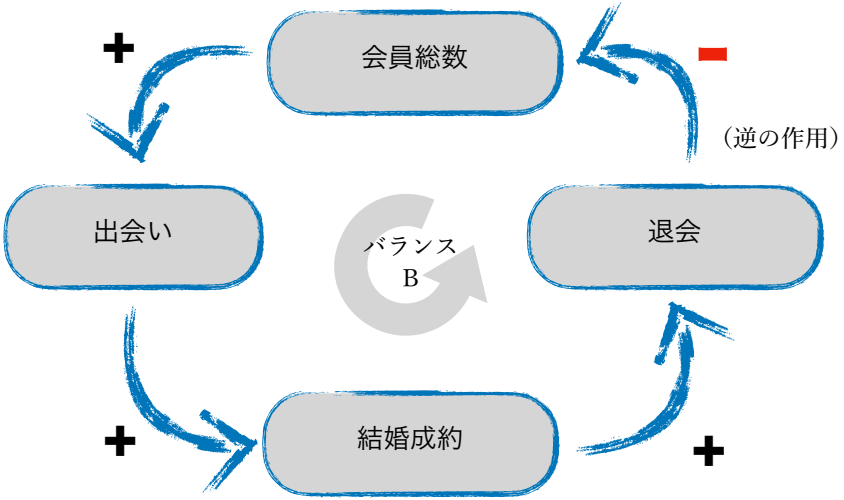
バランス型ループを持つシステムを急成長させたい場合、自己強化型ループを組み入れる必要があります。あるいはバランス型ループになっている構造を自己強化型ループに変更して

いきます。このようにしてシステムの挙動の特徴をフィードバックプロセスに注目して描くことで、システムの挙動を求めている望ましい方向に導くために、自己強化型ループにするのかバランス型ループにするのか、どちらの方が望ましいのかを考える際に、ループ図の作成は役に立ちます。

自己強化型ループ　　SNSサービスの場合



バランス型ループ　　婚活サイトの場合



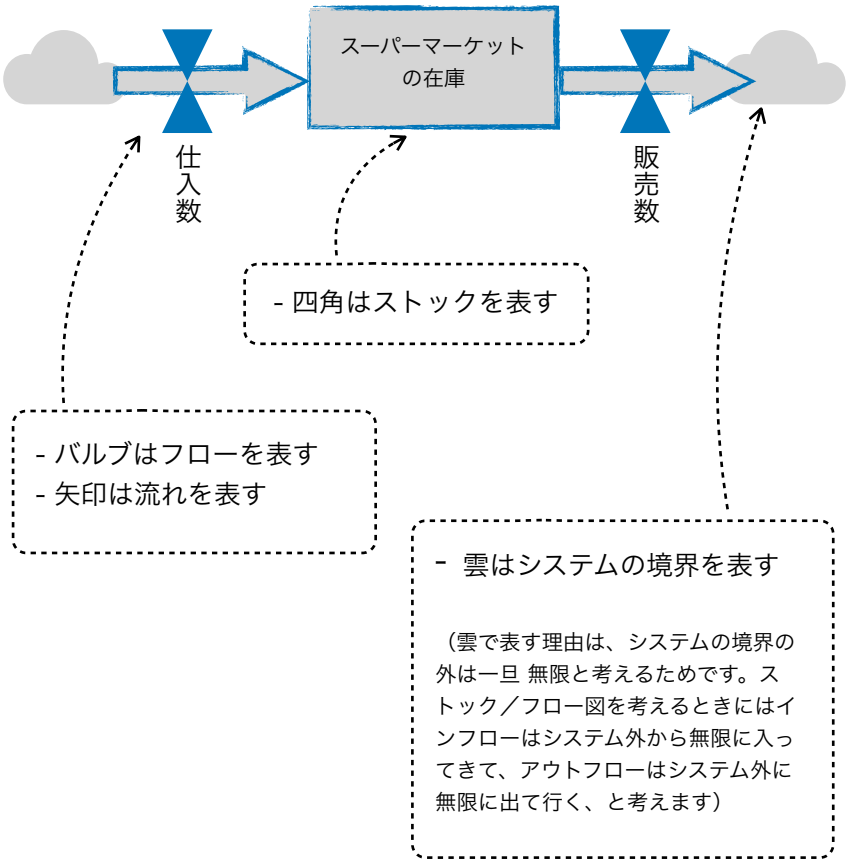
ストック／フロー図

システムの中で蓄積（ストック）する箇所と、システムの流れ（フロー）の関係を記述します。ストックとフローの関係の構造がシステムの挙動の時間的遅延を生み出しています。ストックに入ってくるフローがインフロー、ストックから出て行くフローがアウトフローです。

大抵のシステムでは、このインフローとアウトフローの流れの差（時間的な差、量的な差）がある場合に、ストックがその差を埋めるために緩衝的役割として用意されています。この緩衝作用がシステムの挙動を安定させるために必要なのですが、こうしたシステムの挙動は時間的遅延が働きます。その時間的遅延作用があるために、複雑なシステムの挙動が予測困難なものになってしまうのです。時間的遅延作用が働くことによって、因果関係を直感的に把握することが難しくなり、システムの挙動に驚かされることになります。

それでは例を挙げてシステムのダイナミクスを把握するためにストック／フロー図を用いてシステムの構造を記述していきたいと思います。

スーパーマーケットのストック／フロー図



例えば、スーパーマーケットをストック／フロー図で記述しますと、インフローが仕入れ、ストックがスーパーの店舗の商品、アウトフローが顧客への売り上げです。顧客の来店人数や、個々の商品の販売数は、1日単位では正確に予測することは困難です。多少の幅の中で予測はできても、1個単位、1人単位では予測できません。そのため、スーパーマーケットでは、余裕を持った在庫を抱えることになります（ストック）。これは、在庫切れで次の入荷まで商品がスーパーマーケットの店頭にない場合の機会損失を最小に抑えるためです。そのストックの必要量というのは、仕入れにかかる時間に左右されます。仕入れが頻繁に行えるのであれば、その間持ちこたえられるだけの在庫でＯＫですが、仕入れに時間がかかる場合や、仕入れの頻度が低い場合は、より多くの在庫を抱える必要が出てきます。仕入れの頻度を増やすと配送料がかかってしまうことも影響します。配送料がかかると店頭での販売価格に影響を与えますので、ストックが多い方が販売単価を抑えられます。しかし、商品の賞味期限が短い商品の場合、在庫を多く抱えてしまったことで販売しきれずに廃棄する可能性も高まります。廃棄する商品の数が増えてくると、またそれも販売単価を押し上げることになります。

ストックを少なくし過ぎても、多くし過ぎても、販売単価は上がってしまうのです。ですので、来店客の数の波が大きい店舗は販売単価を下げることは困難になります。できるだけ多くの来店客が来てくれる方がいいのですが、その数も波が少ない方がいい、ということになります。インフローである仕入れと、アウトフローである販売のギャップをスーパーマーケット店舗の在庫というストックで緩衝しているのです。

もし販売予測を見誤って発注してしまいインフローが増え過ぎてしまった場合はストックが増え過ぎることになりますし、急に来店客が増えてしまった場合は、在庫が枯渇して機会損失となります。

スーパーマーケットでは日々の来店客を予測する仕組みと、仕入れのリードタイムを短縮する仕組み、在庫を正しく把握しておく仕組み、が必要となります。こうしたシステムとしての特性を考慮しながら、どのようなＩＴシステムを構築すれば良いか、どのような機能であれば価値が高いか、を考えていきます。

システム原型

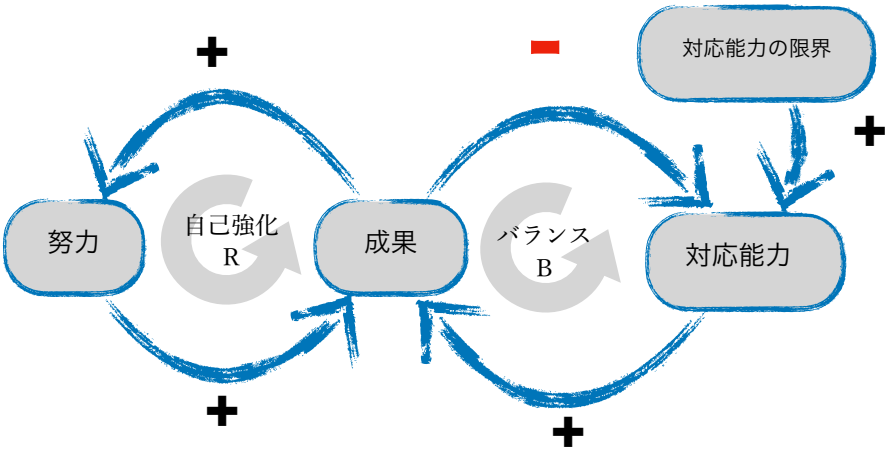
システムの問題の共通的な構造の型をシステム原型といいます。システムには共通の構造上の特徴があることを予め知っておくことで、システムの挙動から構造を発見しやすくなります。

自己強化型ループとバランス型ループの組み合わせでそのシステムの特徴が決まります。現実のシステムではこのシステム原型のように単純ではありませんので、システム原型はあくまでも特徴を把握するための原型として、そこに要素を付け足して対象となるシステムのループを表現していきましょう。実際のシステムでは複数のループが混在していて繋がっていて複雑な挙動を引き起こしているということも多いので、システム原型に当てはめて描いていったものの、実際のシステムのループ構造を暴いてループ図に追加していったことによって、基にしたシステム原型の特徴とは実は違っていた、ということがわかることもあります。

システム原型は、書き始めの際に迷ってしまうことを軽減できる程度に参考にしながら利用するのが良いかと思います。あとは、システム思考に慣れていない人に対して、システムの挙動と構造の関係を説明するために用いるのも効果的です。

ポジティブなループ	ネガティブなループ	システム原型
自己強化型ループ	バランス型ループ	成功の限界 成長と投資不足 共有地の悲劇
自己強化型ループ	自己強化型ループ	成功には成功を
バランス型ループ	自己強化型ループ	問題のすり替わり 応急処置の失敗
バランス型ループ	バランス型ループ	目標のなし崩し エスカレート

成功の限界

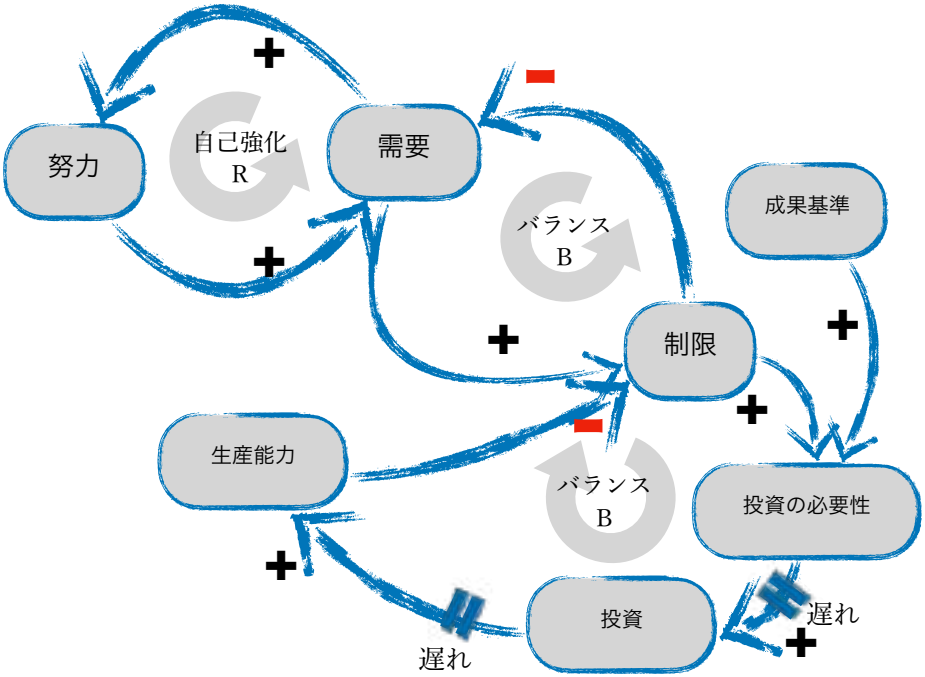


最初は努力をすればするほど成果が出るため、更なる努力をすることにつながります。
しかしいづれそのシステムの成長が限界に達し、良い結果が出なくなっていきます。
最後は対応能力の限界が引き金となり衰退を始めることになります。

具体例

スタートアップ企業が新しい動画アプリを発売した。
このアプリは他社の既存のアプリと違い、画期的な機能があったため口コミでユーザーが増えていった。
同社は動画配信の性能向上やプロモーション活動を重ね、ユーザー数は急拡大していったが、その反面、サーバの応答が追いつかなくなってきた。
そこで、動画処理プログラムの効率化の必要性を感じ、対応を始めたが、それが実現するまでには時間がかかるという事実を甘く見ていたため、その間にユーザーが離れていってしまった。
その結果、投資を削減することに決めたが、この決定が他社との競争に負け衰退していった。

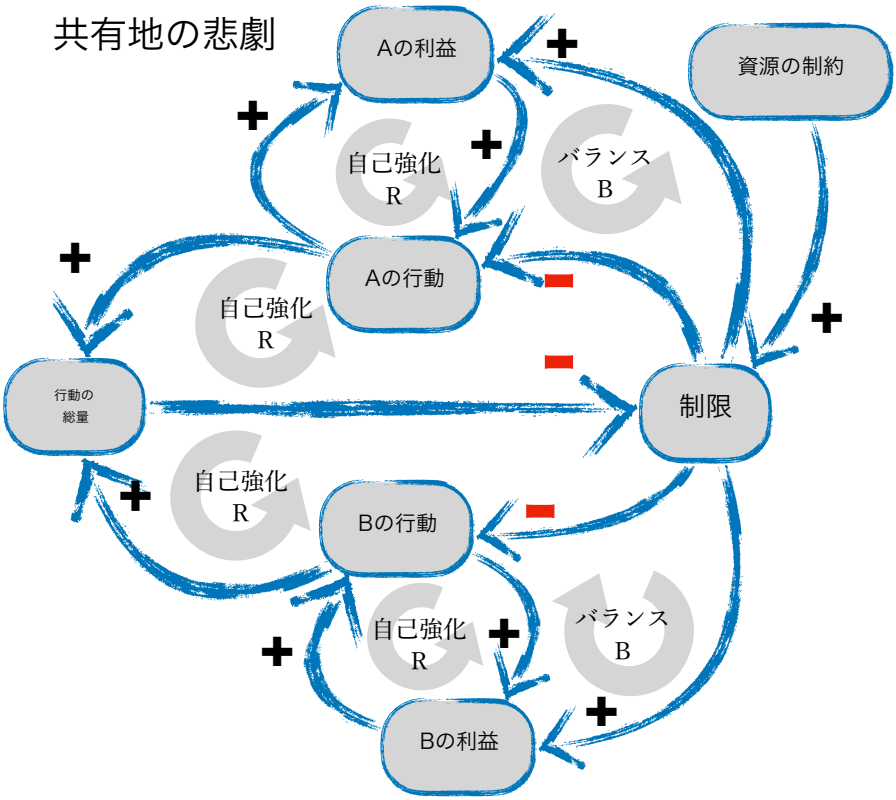
成長と投資不足



さらなる成長を続けるための投資が行われないと、能力が限界に来たところで成長が止まる。この状態になってから投資をしても、成果が現れるまでに「遅れ」があるため、需要に応えるチャンスを失うことになる。その結果、需要が低下し成長がさらに抑制される。

具体例

親切で丁寧なカスタマーサポートが顧客に受け、好業績をあげているものの、現在のコールセンターのオペレーターの数が顧客数に対して少ないという問題を抱えていた。しかし、責任者はコールセンターの人員を増やすことに対して慎重な姿勢をとっていた。その間も顧客は増え続け、それに従ってコールセンターへの問い合わせも徐々に伸びていった。この問い合わせ増は一時的なものであると責任者は考えていたが、日増しに伸びていくのを見過ごすことができず、ついにコールセンターの人員の増強を決定した。それから半年経ち、募集を募った成果が出てコールセンターの人員増強が整ったが、その間に問い合わせの待ち時間の長さに不満を持った顧客は、別の会社のサービスに流れていってしまっていた。

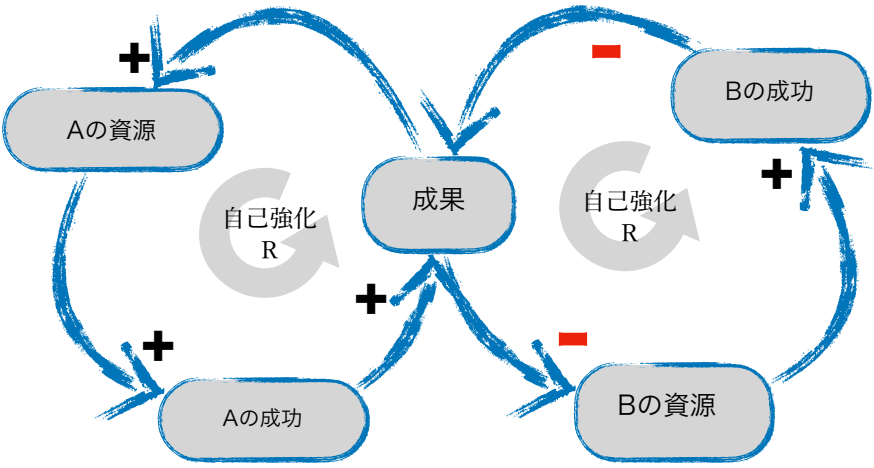


共有資源をそれぞれが自己の利益のみを考えて消費している状態。互いに利己的な行動をとった結果、資源は減少していき、そして最終的には資源が枯渇し、全体にとって望まない結果となる。社会的ジレンマや環境問題などでよく起こるパターン。

具体例

野球のグラウンドを予約できるサイトを構築し、誰もが気軽にグラウンド予約ができるようにした。すると皆が一斉に予約を申し込むようになり、すぐにグラウンドは予約上限に達し、これ以上の予約は不可の状態になった。予約できなかった市民からの苦情が相次いだため、予約者の枠を拡大し、抽選によって割り当てを決めることにした。すると抽選が当たらないことを懸念し、必要以上に予約を入れるようになり、さらにより多くの予約を呼び込むことになった。そしてさらには抽選のための抽選を行う、などの制約が増え、グラウンドを予約できるサイトができる以前よりも予約にかかるプロセスに時間も手間もかかるようになり、予約自体に厳しい事前制限がかけられることとなった。

成功には成功を

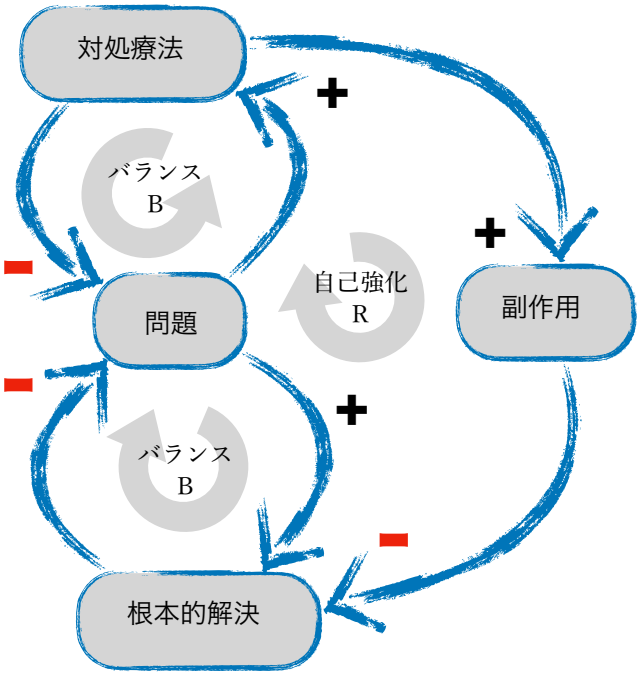


複数の個人や組織が、限られた資源に対して競争関係にあり、山くずしを行なっている状態
そのなかで一方が成功を収めると、そちらにより多くの資源が割り当てられるようになり、その資源を元にさらなる成功を生みだせるようになる。しかし、もう一方は資源を奪われることで衰退し続けてしまう。

具体例

システム開発部では、AとB、それぞれのチームが研究開発を行っていた。ある時、Aチームが開発した技術がメインプロダクトに取り入れられ、大ヒットを収めた。その結果、社内ではAチームに対する評価が上がり、今後の研究に対して、以前よりも多くの予算を与えられるようになった。しかし、システム開発部に対する予算総額は据え置きのため、Bチームは予算を削減され開発者もAチームに異動させられるようになった。そしてBチームは思うように開発を進めることができず、その後も良い結果を収めることができなかった。

問題のすり替わり

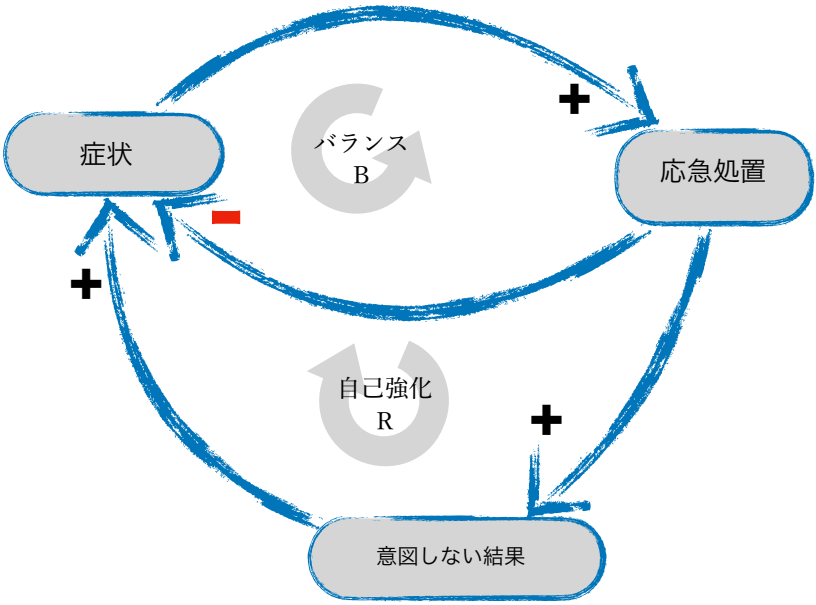


対症療法によって一時的に問題の症状が緩和するため、根本的な解決策を図ろうとしなくなっていく。そして時間が経過すると問題の症状が再発し、また対症療法を実施することになる。対症療法に依存した構造になっていることが、根本的解決策への意欲や関心を低下させるという副作用を強めていく。

具体例

システム開発部では、問題が発生したとき、メンバーはリーダーに助けを求めることが多かった。リーダーは「メンバーが自分たちで問題を解決できるように教育しなければ」とは思うものの、メンバーの考える対応策が思う水準でないことが気に食わず、とりあえずリーダー自身が事細かに指示して対応していた。メンバーの心の中では「問題が起きても、自分たちで考えたことをリーダーに批判されるくらいなら、リーダーに頼んで指示を受けてその通りにしておけばなんとかなる」という依存心が生まれていき、学習への意欲は薄れ、リーダーの助けを求めるのみとなってしまった。

応急処置の失敗

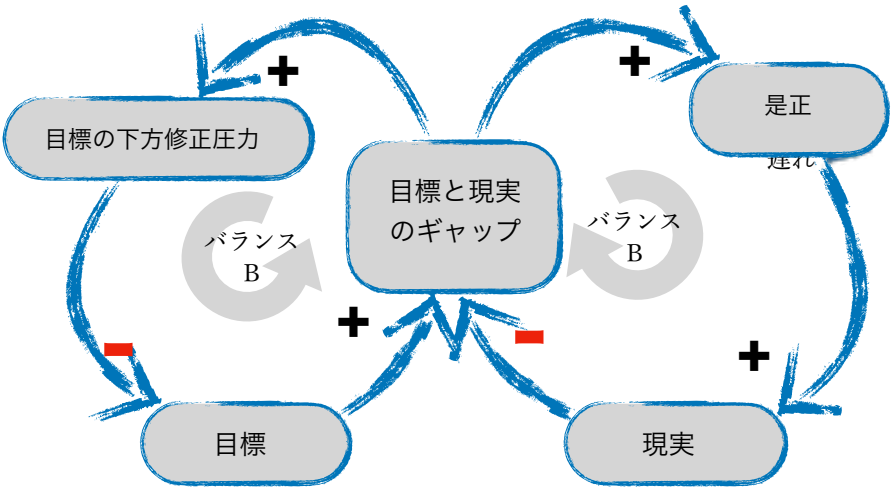


応急処置ばかりで対応していった結果、一時的に症状は緩和されたように見えた後、しばらくして問題が再発したり、以前よりも悪化したりするように、意図しない結果となるパターン。

具体例

ソフトウェア開発チームCは、機能要望の量に対するプレッシャーが高まると、残業をして機能開発を進めた。残業を続けることで、一時的にリリースできる機能数は増えるもの、残業をやめるとリリースできる機能が減り、顧客からの機能要望のプレッシャーが高まるため再度残業を増やして対応。このような対応を何度も繰り返しているうちに、疲弊してきたチームの生産性は落ちていき、退職者も増え、開発できるペースはどんどん落ちていった。そのせいで更に顧客から多くの機能要望を突きつけて開発を求められるようになった。

目標のなし崩し

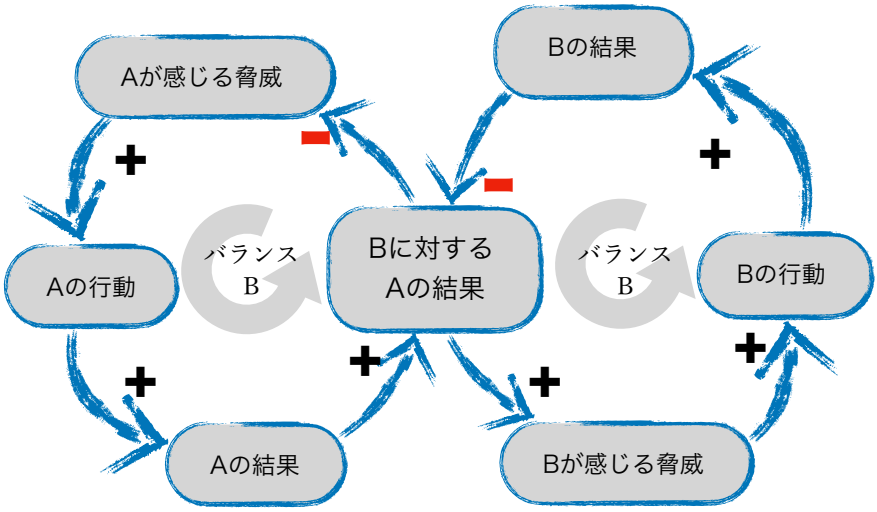


当初の目標レベルが徐々になし崩しに低下していく状態。徐々に進行していくため、その影響に気づかない場合が多い。

具体例

中小企業向けの勤怠管理システムを開発しているD社では、新規顧客を獲得するための販路拡大を目的とした広告宣伝費を増やすために、設備コスト削減策を実施した。それによってソフトウェア開発者のオフィスはなくなり全面リモートになった。コスト削減実施後すぐは、売上は増えたが、その後の売上は下がり続けた。実は、コスト削減策を推し進めたことにより、リモート環境の整備が追いつかず、品質の悪化と生産性の低下によって、既存顧客の離反が相次いでいた。このことが早い段階で分かっていれば、品質を維持するための投資に対する意思決定をできたはずだが、分かった頃には収益は大幅に落ち込み、投資もできない状況に陥った。

エスカレート



Aが現在の立場を脅かされる事態が起きたため行動を起こしてBとのギャップが広がる。すると今度はBがその事態に驚異を感じ行動を起こしてギャップが狭まる。するとまたBの行動の結果に驚異を感じたAが行動を起こす。これらが延々と続き、互いの競争状態が続く。

具体例

ポータルサイトを運営するE社とF社があった。この2社はライバル関係にあった。ある時E社は大胆な広告枠の販売価格引き下げを実施。E社に顧客が流れ始めると、危機を感じたF社がE社よりも低価格で広告枠を販売。するとE社がさらに価格を引き下げた。この行動により、E社F社間の価格競争により両社の利益が減少し、開発するための資金が減ってしまい、両社のサイトは陳腐化。最終的に顧客は別の会社のサービスに取られてしまうこととなった。

モデルを構築するプロセスに価値がある

システム思考において具体的な問題を解決するためにモデルを構築するのであって、システムをモデル化することが目的ではありません。

ですので、モデルを描くときは問題を解決することが目的でなければなりません。そのために目的と関係のない要素を一つ残らず取り除く必要があります。システム思考が役に立たない問題も当然たくさんあります。そしてシステム思考は単独で機能するものでもありません。問題を解決するには、統計処理や市場調査などの伝統的な手法も必要になります。どちらかというと、モデル構築というのは、他のツールを代わりとして用いるのではなく補完するものとして用いることで最大の効果を発揮します。そしてモデル構築というのはプロセスに効果があるのであって、出来上がった成果物に効果があるわけではありません。関係するメンバー全員が集まってモデル構築をするプロセスに関わることが重要です。

対象となる領域を捉え、システム思考によってモデルを構築し続けるというプロセスには終わりはなく、メンバー全員で共に探求を続ける反復プロセスとなった時に、最も効果を発揮

します。問題がどのようにして生じるのか、という気付きを得ながら問題解決への最も効果の高い解決策を探し続けるプロセスなのです。

何かを主張するためにモデル構築を用いることは避けるべきです。何をすべきかについて、はじめから抱いている意見を基にして勝手にモデルを構築したところで大して役には立たないでしょう。自分の内にある意見や価値観をモデルとして表現して自分自身で再確認する、という意味では全く意味がないことはありませんが、メンバーに対して何かの意見を押し付けたいがために、モデルを構築するのはモデルとしての価値を損ないます。モデルというのは現実をありのまま反映したものではなく、適度に抽象化されたものとなります。ですので現実で解決したい問題に対して重要な要素を描くためには、現実の細部を描くことはしません。ですので、モデル自体には解釈の余地が生まれます。見た人によっての捉え方を完全に一致させるのは、出来上がったモデルだけでは不可能です。モデルを構築するプロセスの中でのコミュニケーションによって、モデルは補完されることになります。そしてモデルはそうしたコミュニケーションによる理解を補完します。互いに補完し合うのです。

ですので、あくまでも解決したい問題にフォーカスをして現実に即したモデルを構築しないと、役に立ちません。誰かの思い込みや意見をベースにモデルを構築したとしたら、それが一見現実的に思えても、そのモデルによって現実を捉えたと錯覚させるだけで、そこから導き出された解決策は、現実に対しては機能しないものになるでしょう。モデルには解釈の余地がありますので、そうした錯覚を誘発させる使い方もできてしまうのです。モデルを構築した本人にそうした意図があろうがなかろうが、錯覚を誘発するようなモデル構築プロセスにならないように注意を払いましょう。これはモデルがもつ性質が引き起こす問題として意識しておくといいでしょう。モデルの持つ分かりやすさゆえの問題と言えます。誰かがどこかで描いたモデルを引き継いだり手渡されたりした場合は、そのモデルを疑うようにしましょう。

対象範囲を広く取る

モデルが対象とするその範囲は出来る限り広く取る方が効果的です。局所的な部分を眺めていては問題を取り違えてしまう可能性が高まります。システムのもつダイナミクス性はシステムの各要素間の相互作用から生まれますので、フィードバックが一体どこから繋がってくるのか、を捉えることは、局所的な細部を描くことよりもはるかに重要です。

アジャイル開発とシステム思考の親和性が高いのは、こうしたシステムモデリングにおいて反復的なプロセスが効果を発揮するためでもあります。システムの細部を描くよりも、いかに全体のつながりと重要なポイントを書き出すか、が大事です。要素を増やしすぎたり、細部を描こうとしてしまったりすることによって無駄な時間の浪費を避けましょう。アジャイル開発においてはいかに無駄を排除するか、という点においてもシステム思考との親和性が高いです。アジャイル開発の仕組み自体をシステム思考的観点から見ると、とても効率的なシステムに見えてきます。あなたが取り組んでいるプロジェクトのプロセスそのものをシステム思考でモデリングしてみると、真のアジャイル開発なのか、アジャイル風なだけなのか、まったくそうでないのかが見えてくるはずです。

リスク管理

多くの企業ではリスク管理というものに対しては、どちらかというと消極的で、単にリスクを避ける手法として理解している場合が多いです。

ソフトウェア開発におけるプロジェクト管理は、多くのありえないことを信じる能力を要求される仕事だったりします。後になれば不可能だったとわかる納期、予算、パフォーマンス要素を、実現可能だとチームに信じ込ませるようなことを期待されていたりします。

例えば上司から来月までにたった四人で完成させなければならない仕事をやってみないかと持ちかけられたとします。それに対してそのソフトウェアを開発して完成させてリリースするには時間が足りないと思います、答えたとします。上司から「だからこの仕事を任せる人間として君を選んだんじゃないか」と上司に言われたらどうでしょうか。この仕事を引き受ければ高い評価が得られるかもしれない、しかしどう考えてもこのスケジュールは可能だとは思えない、もしその仕事を引き受けるのであれば、不可能だと思うスケジュールを可能だと信じ込まなければならないでしょう。何度も何度も自分に大丈夫だと言い聞かせるうち

ににそのうち自信が湧いてくるかもしれません。あなたはやれると信じましたが、目の前にある証拠を元に、そのスケジュールが可能だと信じる客観的な根拠があったのでしょうか。信じるに足る証拠を手にすることができ、客観的な根拠があるものだけを信じることが本当の「リスク管理」だと言えます。

リスクをとらなければ他者に支配される

リスクと利益は切っても切れない関係にあります。リスクから逃げ、うまくやれるとわかっていることだけをやろうとする組織は、リスクを積極的に取ってくる相手から出し抜かれることになります。

自社と競争相手たちが一斉に、それぞれ下りのエスカレーターに乗っているような状態だと考えてみるとどうでしょうか。下っていくエスカレーターに逆らって上っていかなければならない状況です。競争相手もそれぞれのエスカレーターに逆らって上っていく必要があります。エスカレーターの流れが速ければ速いほど、同じ位置に留まるには速く足を動かして上

ろうとする必要があります。一瞬でも足を止めてしまえば下へ下がってしまいます。長く立ち止まっていれば下まで落ちてしまい二度と競争に参加できなくなるでしょう。それぞれのエスカレーターには、全員のエスカレーターの速度を調整できるボタンがあります。トップに上り詰めたなら、他の競争相手が追いつけない程度にすべてのエスカレーターの速度を上げるという戦略が取れます。エスカレーターの速度を上げるというのはリスクを取る行為でもありますが、自分が一番の場合は、この戦略は利益をもたらし、他の競争相手には大きな障害となります。負けだすとさらに負ける可能性が高まるわけです。これは何を意味するかというと、もしリスクを取らなければ他人によって支配されるようになる、ということです。リスクを取れば報われる可能性が高まり、リスクから逃げる企業は競争相手に負けてしまうのです。そしてそのリスクもそれぞれの状況によって損害もそのリスクが発現する確率も違うわけです。

では、ここで言うリスクというのは具体的に何のことでしょうか。リスクという言葉の定義をしておきたいと思います。

リスクの定義

リスクとは、「望まない結果を生むかもしれないもの」のことです。この「リスク」という言葉と近い意味で使われている言葉が「問題」です。リスクとまだ起きていない問題のことであり、問題とはすでに実現したリスクのことです。リスクとは発生するまでは抽象的な概念に過ぎません。プロジェクトに影響を与えるかもしれないし与えないかもしれないものです。無視したからといって必ずしっぺ返しを受けるとは限りません。しかしリスクを考えなかったという管理上の過失は責任を問われるべきです。結果的にうまくいったとしても、それは単に「見つからなかった」だけです。リスク管理は問題が発生する前の抽象的な概念の段階で対策を考えるプロセスのことです。リスク管理に近い考え方ですが、問題が発生した後に何をするべきかを考えることを「危機管理」と言います。

リスクに対する対策として選択肢を確保し、後で対応策を取れるようにするために、リスクが実現して問題になってしまう前にやっておく作業、これを「軽減」と言います。

リスクの「軽減」

実際のプロジェクトでこの「軽減」というものを実際に行う例を説明します。あらゆるソフトやプロジェクトに影響を与える最も大きなリスクのうちの一つは開発メンバーの離職です。この場合の軽減策は、最初から人数をやや多めに配置しておき、十分な適性のある予備の開発メンバーをとりあえず見習いやサポートの任務を担わせるというものです。開発メンバーの誰かが辞職したとしても、すぐには新しい開発メンバーを雇う必要はありません。予備の開発メンバーの一人をアシスタントから昇格させ、辞めた開発者の仕事を引き継がせれば、再配置による時間の無駄を最小限に抑えられます。

しかし実際にはこのようなリスクの軽減策が採用されることは多くありません。なぜなら軽減という対策は時間も費用も余分にかかるからです。何事もなくすべてがうまくいってプロジェクトが終わった場合、それだけの時間と費用を掛ける必要はなかったと判断されるでしょう。軽減策は無駄であると判断される傾向にあります。そのため、軽減策を実際に採用するプロジェクトは少なく、そして軽減策を採用できないプロジェクトはリスクに対して脆弱になるため、リスクを管理せず、無視するようになります。これは言い換えると、リス

ク管理の本質は、積極的にリスクを取れるようにすることにあり、リスク管理をしなければ、リスクを無視したり、リスクに対して消極的になり、価値のある活動が出来なくなってしまいます。

現実逃避的ポジティブシンキング

ソフトウェア業界ではなぜか「やればできる」と思い込むことを求められる傾向にあります。そのせいでできないことを示すような分析は叩かれる傾向にあります。明確なリスク管理の基準がないプロジェクトで、リスクを発表するとなると、発表した人が損をするような事になります。やる気がないとか、臆病者だとか、良くない評価をされがちです。しかし、リスク管理を正しくすることによって、少なくともある程度の時間は人々にマイナス思考を許すことができます。プロジェクトが進んでからリスクに不意打ちを喰らわないためには、少なくともマイナス思考は必要です。不確定要素が分かればプロジェクトをしっかり守るためにどれぐらいの備えが必要かも分かります。リスク軽減にかかるコストと、リスクが発生した時の対応のために取っておくコストの合計が、備えとなります。この備えが少ないプ

ロジェクトというのは安全度に欠けると言えます。このような蓄えを『リスク準備』と呼びます。

リスクに対応する

リスクに対してできることは次の四つです。

1 回避する
2 抑制する
3 軽減する
4 かわす

リスクを回避するとはプロジェクトの中でリスクを伴う部分に手をつけないことです。リスクを回避すると当然の結果としてリスクを伴う領域に入ることで得られる利益を見逃すことになります。

リスクの抑制とはリスクが実現した場合にかけなければならない時間と資金をあらかじめ準備しておくことです。この抑制戦略は、実現する可能性が高いリスクを全体として相殺できるだけのリソースは準備しておくというものです。

リスクの軽減とは、リスクが実現する前に、リスクが実現した場合の抑制コストを軽減するための措置をとっておくことです。軽減措置は100%の確率で発生しますので、リスクが実現しなくても軽減コストは節約できません。経験にかかるコストよりその軽減措置による抑制コストの減少分の方が大きくなければ、軽減にコストをかける価値はありません。

リスクをかわすとは、特に何もしなかったが、たまたま運良くリスクの襲来を免れる場合のことを言います。

リスク管理の手順

リスク管理の手順ですが、次のステップをプロジェクトに組み込んで実施します。

1 プロジェクトが直面するリスクの調査をする
2 個々のリスクが実現する確率を推定する
3 個々のリスクに対してリスクが実現した場合にコストとスケジュールに与える影響を推定する
4 リスクが実現しそうになった時、取るべき危機対応措置を決めて危機対応計画を作成できるようにする
5 危機対応措置を実行できるように、リスクが実現する前に取るべき軽減措置を決める
6 全体のプロジェクト計画に軽減措置を追加する
7 実現するとプロジェクトが終わってしまうリスクを特定する
8 リスクが一切実現しないものと仮定して最初のスケジュールを作成する
9 社内の過去のプロジェクト実績や、対象となる業界の過去の情報をもとに不確定要素を盛り込んだ、幅のあるスケジュールを作成する
10 作成したスケジュールを関係者に共有し、それぞれの予想日程や予算に関連する不確定要素を盛り込んでいく
11 リスクを監視し、リスクが実現した時には危機対応計画を実行する

12 プロジェクトの期間中に発現したリスクに対応するためにリスク管理プロセスを継続する

目標スケジュールしかないプロジェクトの危険性

多くのプロジェクトでは作成されたスケジュールがたったひとつだけで、それらは目標スケジュールと同じ意味を持っています。目標スケジュールとは不確定要素による軽減措置を盛り込んでいないので、ほとんどの場合そのスケジュール通りに進行することはありません。ですので、実際のスケジュールというのは、目標スケジュールよりも時間的な余裕が必要で、リスクが発現した場合の危機対応措置を実行した場合のコストと時間も実際のスケジュールには反映されていく必要があります。

ですのでたった一つのスケジュールを作成して終わるのではなく、複数のパターンのスケジュールを用意しておき、さらにプロジェクト期間中に発現していくリスクを盛り込みながら、プロジェクト期間中に常にスケジュールを更新していく必要があります。アジャイル開

発では、そもそもスケジュールを用意しない、スケジュールを計画しない、というものだと多くの人が誤解しています。これは、たったひとつのスケジュールのもとにプロジェクトが進行していくことによる弊害が大きいためであって、スケジュールは必要のないもの、という意味ではありません。スケジュールはたったひとつしかない、目標スケジュールしか作成されていないというものは、顧客の期待値を過剰に高めてしまったり、プロジェクト全体に対する不当なプレッシャーを生み出すことになります。それを避けるためにそもそもスケジュールは用意しないという戦略を取るチームもありますが、複数のスケジュールを用意した上での幅のあるスケジュールというものを運用していくことでそれらの問題を回避することができます。

全体のスケジュールがない場合、リスク管理というのは場当たり的になり、正しく運用することが難しくなります。そういったプロジェクトではリスク管理自体を放棄するようになり、リスクを無視するようになります。リスクが発現して問題となった時に後から対処する危機対応措置ばかりが追加され、チームの生産性は落ちていきます、全体的なスケジュールがひとつもない場合、複数ある危機対応措置を比較したり、あらかじめ用意する軽減措置

によるスケジュールの変化などについて考えることが難しくなります。プロジェクトの全体性が損なわれていくわけです。

アジャイル的なリスク管理というのは決して事後で対処するということを意味しません。リスク管理のプロセス自体をアジャイルに行うということであって、リスクが発現した後の対応をその都度その都度やればいいというわけではありません。アジャイル開発におけるリスク管理とは、このイテレーションの中で発現するリスクに対してのみ管理するわけではなく、そのイテレーションの中で新しく見つけ出したリスクについて管理していきます。リスク管理自体をアジャイルに行なっていくということです。

プロダクトを開発するサイクルとしてイテレーション期間の中で作り切ってリリースするという原則は守りながらも、リスクに関してはプロジェクト期間全体を対象として管理していくことが望ましいと言えます。

モデリング

複雑なシステムの全体性を把握する上でその記述方法・表現方法というのはとても重要です。あらゆる物事を詳細に書き尽くそうとした場合、文章であれば文字数がとても多くなり、すべてを記憶することはできなくなります。読んでいるそばからどんどん忘れていくわけです。ですので、いかにシンプルに表現をしてそして多角的に表現できるか、このためのテクニックというのがモデリングです。というのも、人の思考は表現の方式そのものに影響を受けます。もし日本語で書いていた場合、日本語という言語特性・言語構造が持つ世界観から影響を受けます。英語ネイティブで考える人と、日本語ネイティブで考える人とで、実は同じ意味を表している文章であっても、捉える世界が違ったりします。ですので望ましいのは、特定の言語に強く依存するような表現方法ではなく、直感的に理解ができるもの、いわゆる知覚能力を刺激する方法を用いて表現していくのが望ましいです。

用いる図形はたったの3つでOK

丸、四角、矢印。

この三つの図形を用いるだけで、システムの全体性を表現することは可能です。システム思考のところでも触れましたが、システム思考におけるループ図、ストック／フロー図もモデリングの一種です。有名なものとしてはUMLがありますが、ここで私の言うモデリングというものは、もっと広義のものであり、必ずしもUMLに沿って作らなければいけないというわけでありません。UMLの知識は役に立つと思いますが、大事なのは、統一された表記法としてのルールではなく、システムとしての重要なつながりを把握し、それらを迅速に修正していくことができることです。アジャイル開発では、イテレーションごとにシステムが変化をしていくとことが起こります。今のシステムの現状を正しく捉えるのと同時に、将来にわたって変化するシステムの作用・構造・振る舞いをを把握することが重要です。

システム思考のところでも触れましたが、複雑なシステムというものは、ある一部分を変更した場合、その変更の影響が思わぬところで出てくるということがよく起こります。全体性

をうまく把握しておかなければ、その影響を見逃すことになり、部分最適に陥ってしまいます。開発のサイクルはイテレーション期間内で作れる小さなものを作るというアプローチですが、考える範囲というのはビジネスシーンを含めたシステム全体を対象とする必要があります。なぜならユーザーが影響を受けるのはシステム全体からであり、決して個々の機能が単体で作用するわけではないからです。

常に完成品を提供し続けるというスタンスを維持するためには、システムの全体性を捉え損ねないことがとても重要です。具体的な表現方法や表記法についてはモデリングの章で説明したいと思います。

認知科学（社会心理学）

システムを作るのは人間であり、チーム・組織というものも人間で構成されています。そのため、私たち人間がどのような認知メカニズムを持っているのかについて熟知していることはとても有益です。認知科学に関する知識がない場合、人間というものを誤解して扱うことになります。人の記憶には限界があることは誰しもが同意すると思いますが、その記憶の限界は思ったよりも低いことを理解している人は少なかったりします。実際に記憶できる量と、記憶できると思っている量にギャップがあるわけです。ですのでつい、「これぐらいのことなら覚えていられるだろう」と高を括って、後になって思い出せない、あるいは記憶違いを起こしていることが往々にして起こります。

生来的な特性を理解する

動物はスービタイズという能力を持っています。これは物体の数を把握する時に、認知的負荷を伴わずに本能的に四つまでであれば認識できる能力のことです。動物は四つ以下の物体

を本能的に認識が可能です。五つ以上になるとカウンティング（＝数え上げ）が必要になります。生まれたての赤ん坊でも、四つ以下のものに関しては数を認識できます。カラスや鳩ですら、スービタイズの能力を持っています。カウンティングは後天的に獲得する能力であり、スービタイズは生来的に持っている能力です。ですので何かを表現したいときに複数の選択肢や全体性を表現したい場合、大きく文脈上の意味が変わってしまう場合を除いて、なるべく四つ以下にポイントを絞った方がいいです。五つを超えると認知的負荷がかかるので、全体を把握することに負荷がかかってきます。これはモデリングのところでも触れましたが、モデリングで表現しようとしている要素の数が増えすぎると、かえって全体性がわからなくなります。これは認知的負荷の問題であり、表記方やシステム自体の欠陥というわけでありません。いわば我々人間自体に生来的な限界があり、その限界を認識しておくことが重要です。

抽象概念を無意識に用いてしまうことこそがヒトの特徴

そして私たち人間は、現実と自分たちの頭の中で想像した抽象概念とを区別することが苦手です。むしろ私たち人間は抽象概念の世界の中で生きていると言っても過言ではありません。この取り巻く世界を抽象的な概念で捉え頭の中で処理しているわけです。なぜなら知覚刺激として入ってくる情報のその全てを毎回毎回、精緻に捉え、限られたサイズの脳によって処理し続けるには、世界はあまりにも複雑で、その世界に対して脳はあまりにも小さすぎるのです。ですので、その複雑な世界に対して私たちの脳は「いかに節約して処理をするか」に応えるように進化してきました。抽象概念で世界を捉えているというのは進化の結果であって、それによって、より大きく複雑な思考が行えるようになりました。

抽象概念の一例としては『無限』という概念があります。『無限』というのは現実には存在しないことであり、あくまでも抽象概念の中でしか存在しません。しかしこの『無限』という概念はとても便利な概念です。『無限』という概念は現実に対してはあくまでも近似である概念ですが、一旦『無限』と置いて考えてみると、カウンティングをし続ける作業の中で、開始時の認知リソースを節約できます。カウンティングを開始する時に、「無限にカウ

ンティングすること」をデフォルトにしてしまえば、認知的負荷を減らすことができます。「どこまでカウンティングすればいいか」をカウンティングの開始時に考えないことで、認知リソースを節約しているわけです。そしてこれがバグを生む原因になることがあります。現実の世界では無限にカウンティングをし続けることはできません。結局はどこかで止めなければいけないのですが、抽象概念の中ではデフォルトで「無限にカウンティングする」となっていると、カウンティングをどこで止めるべきなのか、どこで止めなくてはいけないのか、について考えるタイミングを失うわけです。生来的に『無限』という概念を獲得したことで認知リソースを節約できるようになりましたが、その反面、現実との間に不整合を起こし、バグを生み出すことにもつながります。このことを理解しておくと、人は常に連続した繋がりを捉えようとした場合、思考の開始時点では、「デフォルトで上限値に『無限』がセットされている」、と認識することで対策を考えられるようになります。

『注意力』に注意を払う

人の注意力というのは基本的には散漫で、簡単に刺激に対して反応してしまいます。集中できる環境を作るというのはとても重要なのですが、実際にはその集中できる環境を実現できている組織というのは少ないのが実情です。特に日本企業の場合、就業スペースがオープンデスクになっている場合が多く、周囲からの雑音や注意を削ぐような視覚的情報が多くなってしまいます。そのような状況で集中し何かに取り組むというのは非常に難しく、価値の高いものを作り上げるのには適していない環境です。この場合、認知科学的な視点での集中できる環境作りというものを考えることが必要になります。スクラムでは毎日十分程度のデイリースクラムという、スタンドアップミーティング実施し、それぞれが短い言葉で現状の共有をします。これはチームメンバーが協働するための同期を取るためでもあるのですが、そのようにしてメンバーとコミュニケーションを取るタイミングと個人として集中できる時間というのをうまく切り分けることによって、注意力の無駄遣いを減らすことができます。

注意力というのは自分自身でコントロールすることは非常に難しく、外部からの刺激に対して反応しないでおこうとすること自体にも認知リソースを使ってしまいます。ですので、ノイズの多い環境や不必要なまでの過剰なコミュニケーションが自然発生的に起こる環境とい

うのは、認知リソースを多く消費し、生産活動に充てられる認定ソースが残っていない、ということにつながります。だからといって完全にコミュニケーションを断絶させてしまうというのはチームとしてうまく機能することが難しくなりますから、コミュニケーションをとるタイミングとコミュニケーションの取り方に関して、認知科学的アプローチを理解しておくことは重要です。完全な個室を用意することは難しいかもしれませんが、他者からの視線を遮ることができるパーティションや、開発者を雑音から守るためのヘッドホンやイヤーマフなどを提供し、ヘッドホンやイヤホンしている時には極力雑談などを話しかけないようにするという、チーム内でのルールを決めておくというのも効果的です。

心理的安全性と創造性は相関する

大規模なシステムを開発する時は必ずチームで開発を行うことになりますが、チームで何かを作り上げようとする時には、他者と共同作業をすることになります。他社と共同作業をしようとする場合、心理的安全性がとても重要であり、そして柔軟な発想をするためにも心理状況をポジティブな状態にしておくことが効果的です。ポジティブな心理状態の時、人は生

産性が高まり、新しいアイデアを思いつきやすくなり、そして離職率も下がります。個人の心理状態だけでなく他者の心理状況も汲み取れるようになれば、チームの生産性はより高まります。個別の事象ばかりを見るよりも全体を俯瞰して眺めることができると、人は比較的ポジティブになります。全体性を把握することによって心理的に安定するのです。アジャイル開発においては近視眼的な視点が強まる傾向にありますから、全体性を把握するということによって心理的なバランスをとるというのは効果的です。あらかじめビジョンを立てて、長期的に実現したい目標をチームで共有し、トラブルや強い負荷がかかった時に、目の前の事象だけを見るのではなく、ビジョンに照らし合わせて長期的視点で捉え直すことによって、チーム全体の心理的な安定を実現することができます。

例えばバグが出て、ユーザーから不快な思いをしたと報告を受けたとします。その機能を開発した開発者は落ち込むでしょう。そしてまたその報告を最初に受け取った人物も不快な思いをするかもしれません。それによって、もう金輪際バグは出しちゃいけない、と考えたくもなります。そしてバグが出たという報告を受けるたびに、その思いは蓄積し強まっていきます。これを基準に照らし合わせて長期的な視点で捉え直すとどうでしょうか。バグの報

告があることは、少なくともユーザーが積極的に利用している証拠でもあり、もし全くユーザーがいないようなシステムであればバグの報告すらも来ることはないでしょ。ユーザーが積極的に利用している事実が意味することとは、積極的に利用するだけの価値がシステムにはあるわけで、バグの報告を受けるというその事象の裏側には、システムのもたらす価値を享受しているユーザーがたくさんいることを意味します。チームとしてのビジョンが「より多くのユーザーに使ってもらえること」と設定されていれば、そのビジョンに沿った結果が生まれていると解釈すること出来ます。長期的な視点で俯瞰して捉えられれば、一見トラブルに見えたような事象でも、大きい文脈の中では、うまくいっている結果の表れであると捉え直せます。バグを絶対に出しちゃいけないと思い込むことで、過剰な品質の向上に時間を費やしすぎたり、価値の高い新しい機能を作る、ということを後回しにしたりすると、長期的にはそのシステムの価値は下がっていき、徐々にユーザーを失っていくことになります。

人は危機や脅威に対して敏感に反応するという認知的な特性があります。ですのであまりにも多くの刺激が飛び込んでくると視界は狭まり、全体性の把握ができなくなってきます。で

すので認知科学的なアプローチから考えて、全体性を把握し続けられる環境というのを設計するのはとても効果的です。

知識の習得も能力の成長も漸進的に

実際のプロジェクトでも私はこれら四つの領域の知識をチームに対して紹介するようにしています。いきなり全てを紹介するのではなく、必要なタイミング、必要なサイズで適宜教えるようにしています。実際にそれをどうすれば上手く使えるツールとして形にできるのかというのを、手本を見せながら、それぞれのプロジェクトの特性や、プロジェクトの状況に応じて、様々な形で提供することになります。幸いにもソフトウェア開発の現場にいる人たちの多くは、知的好奇心が強いので、興味を持って楽しんで理解しようとしてくれますので、私も楽しんで取り組ませてもらっています。

第一章のまとめ

- ウォーターフォールモデル（計画駆動型）は予測を基に計画を立てることから始める。その予測は『天気予報型』であり、世界を静的なものとして捉えてしまう。それによって現実の動的な世界に対応することができない。
- アジャイルモデルは適応的なスタンスを基にしており、予測よりも現実と実績を重んじる。アジャイルにおける予測は『株過予測型』であり、予測そのものには幅があり、計画も幅があるものを用意する
- アジャイルでは全体性を損ないやすいため、いかに全体性を損なわないか、について様々な工夫が必要となる
- 全体性を損なわないためのテクニックや知識として、『システム思考』『リスク管理』『モデリング』『認知科学』などがある。
- 最初に全てを習得してしまってから実行しようなどと思わないこと。実行しながら漸進的に学習し成長すること。アジャイルで何より大事なのはスタートをきること。プロジェクトを始めよう。

第二章：アジャイルとは何か

アジャイルのメリット

『学習』というプロセスの組み込み

アジャイルという言葉は2001年ソフトウェア開発業界を代表する十七人のメンバーが共同で発表した、後にアジャイル宣言として知られる宣言文に遡ります。これはアジャイルの本質をまとめた文章で従来のウォーターフォールモデルでの問題点との対比のように書き記されています。

「プロセスやツールよりも個人と対話、包括的なドキュメントよりも動くソフトウェア、契約交渉よりも顧客との協調、そして計画に従うことよりも変化への対応」

実際にアジャイルモデルでは従来のウォーターフォール型に比べて少ない人員と短い期間でより質の高いものを低コストで多く提供できます。Google や Amazon、セールスフォースドットコムと言ったＩＴ業界の巨人から、次々に立ち上がってくるスタートアップに至るまで、アジャイルモデルはソフトウェア開発を根本から変えました。

アジャイルでは何かをするのになぜそれだけ長い時間がかかるのか、なぜそれだけ労力が要るのか、なぜ私たちが必要な時間と労力を予測するのが下手なのか、を考えます。アジャイルは不確実性を前提としています。プロダクトを開発するプロセスの中に学習プロセスというものが組み込まれており、どのように開発したのかをチーム自らが評価する仕組みが組み込まれています。ですので常に自分たちは正しい方向に向かっているのか、顧客に価値を提供しているのか、さらに上手く早く進めるやり方はないか、そしてそれを妨げているものはないか、と考えることで、自分たちの仕事を振り返って仕事のやり方そのものを改善していこうというものです。

チームの生産性を飛躍的に上げる

アジャイルがもたらす最大の成果はチームの生産性を飛躍的に上げることです。

基本的にはウォーターフォールモデルを適用している組織は、アジャイルモデルでうまくやっているチームには勝てません。ウォーターフォールモデルでは、組織を統制しようとしたり、絶対に失敗しないようにと見えもしない未来を基にした予測を立て穴がないように

見える完璧な計画を作ることに時間をかけすぎて、ようやく市場に製品を出す頃には、アジャイルモデルでどんどんと製品をリリースしていった競合他社に、市場は開拓され、シェアを奪われていくでしょう。これから先のソフトウェア開発の世界では、アジャイルモデルを取り入れなければ競争に勝つことは難しいでしょう。

アジャイルでは完成する時期はプロジェクト開始時には全くわかりません。それはチームがどのくらいの生産性があるのか、また対象となるビジネス領域がどの程度複雑で、そのビジネス領域において求められている価値を出すためのシステムの複雑さも分からないからです。これは単に調査不足という意味ではなく、プロジェクトを始める前にどんなに綿密に計画を練ったり調査をしたところで、出来上がった膨大なドキュメントは単なる思い込みの蓄積に過ぎず、現実を正しく反映しているとは言えません。実際に始めてみないと分からないことの方が多いのです。

スピードと流れへのこだわり

アジャイルモデルではスピードを追求します。スピードアップを妨げる要素が何なのかをチームのメンバーが把握できるようにしていきます。いかに無駄を減らして流れを生むか、そしてその流れを加速させられるか、その流れを遮るような障害は一体何なのか、アジャイルモデルはこの流れに注目をします。そのため、この流れが生まれる一番最適な期間、これをイテレーション（＝スクラムでは『スプリント』と呼びます）の一単位とし、イテレーションの中で最速のスピードが出せるようにチームを機能させようとします。

ですから単に開発サイクルをイテレーションに区切って一定期間ごとに製品をリリースするということをやっていればアジャイルになるというわけではありません。それは流れを生み出すためにそのような形式を取っているのであって、そのような形式をとったからといって、流れがうまく生み出せていない状態であれば、そのプロジェクトはアジャイルであるとは言い難い状態です。この流れに注目をしたという考え方は、トヨタ生産方式による、「製造工程は初めから終わりまでスムーズに速やかに進むべき」という考え方と一致します。

アジャイルの世界観

アジャイルというのはコンセプトのことであり、具体的な開発手法に関しては、いくつかの流派があります。代表的なものでいくと、エクストリームプログラミング（XP）、スクラム、カンバン、リーン、ユーザー機能駆動開発（FDD）、などがあります。他にも多くの流派があり、それぞれが重心を置いている位置が微妙に違ったり、あとは形や仕組みがほとんど同じなのにキーワードだけが違うといったものがあったりしますので、その全てを網羅することは必要ないかと思います。その中でも最近最も導入事例が多いのはスクラムです。

スクラム

スクラムはフレームワークとして規定されているルールがとてもシンプルで、そのルールの本質を分からないうちからでも、始めることができるので、アジャイルというものがわからない未経験者や初心者でも、実践しながら学ぶ、ということができるので、おすすめです。ただスクラムというのは、そのシンプルさゆえに規定されているルールだけを守っていれば生産性が上がるわけではないという注意点もあります。しかし他のアジャイル開発手法に比べると、比較的安全に始めることができます。導入初期ではスクラムのシンプルなルールを守りながらアジャイルの本質を徐々に理解していき、そして徐々にスクラムの中では規定されていない部分に関してもチームで考えながら自分たちでスクラムという基盤の上に築き上げていくという、チーム自身による開発手法の構築していけばいいと思います。スクラムでは開発期間をスプリントという固定された期間で区切り、例えば1スプリントを2週間と決めたら、第2スプリントも2週間、第3スプリントも2週間、と固定した期間を繰り返すというルールからも、時間に対する制約はかなり厳しく設定されています。2週間でできなかったからといってもう1日延ばしてもらってリリースを1日遅らせる、などといったこ

とは許されません。スプリントが終わった時に完成したものだけをリリースします。これは生産性を高める上で非常に重要なアプローチです。本来時間というのは伸び縮みしません。ですから限られた時間の中で最大限の価値をもたらすような機能を持つプロダクトとサービスを開発して提供するべきであって、その2倍の時間をかけたからといって提供できる価値が2倍になるとは限らないですし、2倍の時間をかけることによる不確実性の増大と、フィードバックループの遅れの方がより大きな問題を生み出すことになります。

スクラムで用いる主なキーワードは次の10個です

3つのロール

- プロダクトオーナー（PO）
- スクラムマスタ（SM）
- 開発者（Dev）

4つのイベント

- スプリントプランニング

- デイリースクラム
- スプリントレビュー
- レトロスペクティブ

3つの成果物

- プロダクトバックログ
- スプリントバックログ
- リリース判断可能なプロダクトインクリメント

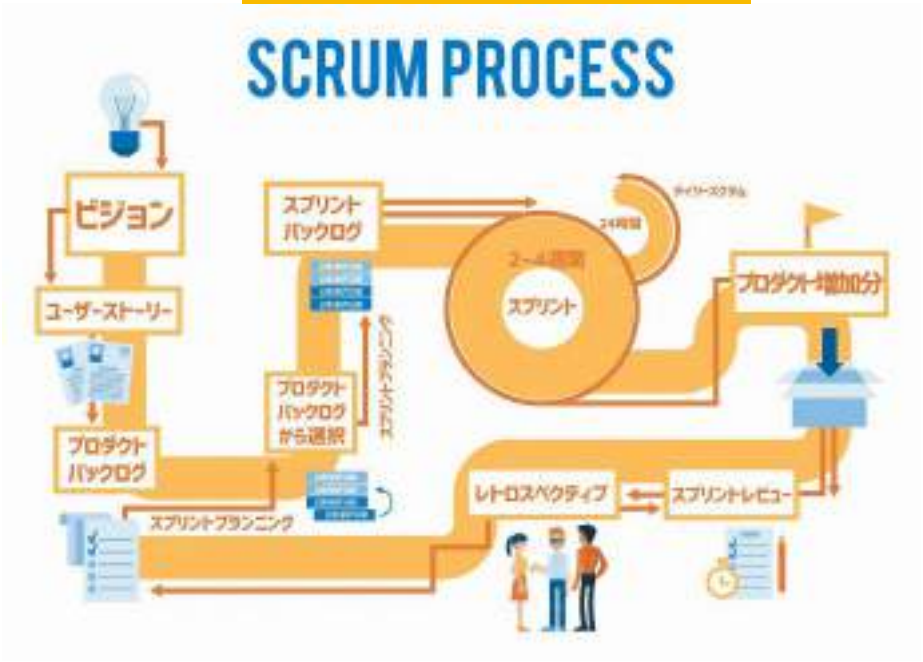
スクラムでは最低限これらだけを準備して開始することができるとしているところが、始めやすい理由でもあり、実践を通じて学習していくことでレベルアップできるので、アジャイル開発を始めたい場合、迷ったらスクラムで良いと思います。4〜9人の少人数で開始して、まずは最小機能を開発してリリースしましょう。1スプリントの期間は1週間にして、レトロスペクティブにてチーム自体を評価しましょう。振り返る回数が多い方が気付きも多く、速く学ぶことができます。

他にもスクラムで用いられる独自のキーワードとしては

- スクラムボード
- プランニングポーカー
- ユーザーストーリー
- ワーキングアグリーメント
- バーンダウンチャート
- ベロシティ
- スパイク
- リリースプランニング
- 妨害リスト

などがあります。

スクラムのプロセス


SCRUM PROCESS
ビジョン
ユーザーストーリー
プロダクト
バックログ
スプリント
バックログ
プロダクト
バックログ
から選択
スプリントプランニング
スプリントプランニング
スプリント
2～4週間
24時間
デイリースクラム
プロダクト増加分
レトロスペクティブ
スプリントレビュー

3つのロール

プロダクトオーナー（PO）

プロダクトオーナーは何を作るのか、作られたプロダクトの価値はどのくらいか、チームそしてプロダクトのビジョンを明確にする立場にあります。価値やリスク、損失などを考慮しながら、優先順位を決めてきます。チームの外側のステークホルダーと積極的にコミュニケーションをとり、ユーザー視点をもつことで、スプリントバックログアイテムに対して設定される『完了の定義』の妥当性や、スプリントレビューにて出荷判断をする役割を担います。何を作るのかについては決めますが、どのように作るかに関しては言及しません。プロダクトバックログにユーザーストーリーをどんどん積み上げていきそのユーザーストーリーをプロダクトバックログの中で優先度順に並び替える仕事を行います。

決してチームのリーダーではなくマネージャーでもありません。他のチームメンバーと同等の立場です。プロダクトが提供する価値の領域に対して精通している必要があります。ユーザーの周辺事情やマーケットを理解している必要があります。プロダクトオーナーは

チームが成果として踏み出すプロダクトに責任を持ちますが、意思決定はチームメンバーが自分たちで行います。そしてプロダクトバックログに積み上げたプロダクトバックログアイテムについてなぜそれが必要なのかをいつでも説明できるようにしておく必要があります。プロダクトバックログの最終的な責任はプロダクトオーナーにありますがチームメンバーとは常に対話しチームメンバーの意見も判断の参考にします。

開発者から見てプロダクトオーナーはいつでも話しかけられる存在でなければいけません。忙しいからということで開発メンバーと顔合わせるのが週に一回というような他のプロジェクトと兼務しているような人をプロダクトオーナーに任命するべきではありません。そして価値を説明する際に各ステークホルダーが納得をする説明ができる必要があります。価値の評価基準を決めることによって、チームが生み出す価値を評価し、その価値を増やしていくことを考えることができます。プロダクトオーナーが複数名で構成されるということもあります。

スクラムマスター（SM）

スクラムの各イベントや開発中のプロセスがうまく進むように目を配り支援する役割です。スクラムの理論、ルール、価値基準などを全員に理解してもらえるように支援します。チーム内に透明性が保たれるように気を配ります。チームが仕事を進める際の障害になる事柄を率先して把握していきます。チームを継続的に改善し成長させていくのがスクラムマスターの仕事です。このスクラムマスターに関しては次は第3章でさらに詳細に説明していきたいと思います。

開発者（Dev）

優れたチームには強い目的意識があります。平凡なレベルで満足しない高い志があることによって、卓越した領域に達することができます。そして優れたチームは自律的で自己組織化ができています。仕事をどのように進めるかは自分たちで決めて管理し実現する力を持っています。優れたチームは機能横断的です。プロダクトの完成に必要なスキルを全て備えています。設計も開発もテストもすべてはフェーズの業務をこなすことができます。チームのメンバーは9人以下に抑えます。人数が多いと他のメンバーが何をしているのかを正確に把

握することが困難になってきます。無理に把握しようとするとスピードが落ちて生産性が下がります。それぞれの抱えている課題仕事の進捗等の全てがそれぞれメンバー間で見えるようにしなければいけません。チームメンバーが多すぎると、やり取りの流れが複雑になりすぎてコミュニケーション不全が起こってしまいます。ですので出来る限り1チーム内での人数は5~9の少人数で構成するべきです。

4つのイベント

スプリントプランニング

毎スプリントの開始時に行います。出来る限り短時間で行うように、あらかじめ時間を決めて決めて、その中で終わるようにスクラムマスターはチームの注意をコントロールします。このスプリントプランニングで決めることは大きく分けて2つになります。1つ目は「今スプリント終了時にどのようなプロダクトインクリメントをリリースできるのか」、2つ目は

「プロダクトインクリメントをリリースするために必要な作業をどのように成し遂げるか」です。

プロダクトバックログから上から順に1スプリントで作れる分を取り出しスプリントバックログに入れます。プロダクトバックログは常に優先度順に並べられているはずなので、これから取っていけば今プロダクトバックログの中にある最も優先度の高いものを選び出すことになります。1スプリントで作れる分を判断する際に参考になるのは過去の実績のベロシティからです。1スプリントでどのくらいを作れるのかは何度かスプリントを回した後にようやく分かってきますが、ベロシティが安定してきたところでチームが1スプリントで作れる規模感がようやく見えてきます。ベロシティとは1スプリントで作ったスプリントバックログアイテムのサイズの合計です。

取り出したプロダクトバックログアイテムに『完了の定義』を設定して、その『完了の定義』を基準に、プロダクトバックログアイテムを分解し、スプリントバックログアイテムにしていきます。分解後のスプリントバックログアイテムに関しても『完了の定義』を設定してい

きます。スプリントバックログアイテムをスプリントバックログに積み上げる際に、個々のスプリントバックログアイテムの『完了の定義』を確認しながら、プランニングポーカーを使って見積もって設定していきます。

プランニングポーカーは、フィボナッチ数列を用いて、相対比較としてサイズを判断して見積もりの数字の書かれたカードを裏向きで出して、全員がテーブルに出したらシャッフルして誰が出したかがわからないようにして、表向きに開いてカードに書かれた数字の平均値を見積もりとします。この数字の単位ですが決して時間ではないのでストーリーポイントと呼んだりもします。チームメンバー全員が経験して理解のしている機能を基準の2として、その機能と比較して、今回の見積もろうとしているスプリントバックログアイテムのサイズがどのくらいなのか、と比較しながら見積もりを進めていきます。この場合のサイズというのは複雑度であって、どのくらいの時間がかかるのかではありません。これは時間という捉えどころのないものを考え出すと、その予測は大してあてにならないため、その結果、誤差が大きくなりすぎてしまうからです。

そして、この見積もりの作業で情報や経験が少なすぎて、全く見積もりの検討もつかないという場合は、それらを調査するタスクとしてスパイクというものをスプリントバックログアイテムとして扱います。このスパイクも完了の定義が必要で、何をどこまで調査すればこのスパイクは完了になるのか、を明確に決めてプロダクトオーナーと合意をしておく必要があります。そして注意点は、このスパイクというものはとても便利なアイテムですので、ついなんでもスパイクで調査しようとしてしまいがちです。1スプリントが全てスパイクで埋まってしまう、みたいなことにならないように気をつけてください。そのスプリントは何も出荷できないスプリントになってしまいます。ですのでスパイクもどこまでやれば完了なのかというのを事前に決めて取り掛かるようにしておいてください。そしてスパイクが完了になったら再度見積もれなかったスプリントバックログアイテムを見積もります。見積もりはスプリントプランニング内だけで終わるばかりではなく、スパイクがいくつか完了したタイミングで見積もることになりますので、スプリントの中で処理をするスプリントバックログアイテムのユーザーストーリーポイントの合計というのは、スプリントプランニングの時点ですべてが完全に決まっている場合ばかりではありません。

スプリントプランニングの中で見積もりが終わりますと、それらの合計値を過去の自チームのベロシティと比較をして、現実的にどのくらい機能をリリースできそうかを判断していきます。優先度の高いものから順に開発をしていくわけですから、優先度の低いものは、もし合計ポイントがチームのベロシティを超えてしまっていたなら、プロダクトバックログに戻します。

そして、それらのスプリントバックログアイテムをどのように開発していくのか、という具体的な作業内容と作業分担について話し合います。基本的には一つのスプリントバックログアイテムを全員で取り掛かるというのが望ましいですから、チームでマルチタスクにならないようにプランニングしてください。そしてスクラムボードのTODOの列にスプリントバックログアイテムを貼り出し、そしてスプリントバックログアイテムに対応する『完了の定義』もDONEの列に貼り出しておきます。これで、いつでも今チームで取り組んでいるものが分かりますし完了の定義を見逃さないようになります。

デイリースクラム

毎日15分程度の常に決まった場所で立ったまま行うミーティングです。基本的に毎日開催します。デイリースクラムでは、メンバーがそれぞれ順番に一方的に報告します。『前回のデイリースクラムから今回のデイリースクラムまでの間に行った実績』、『次のデイリースクラムまでにやる予定』、そして『チームの成果を阻害している障害』について、一方的に報告します。デイリースクラムでは相談や議論は行いません。相談や議論を行いたい場合は、対象者とそのための時間を決めるだけにして実際の相談や議論は別のミーティングで行うようにしてください。デイリースクラムではチームメンバーのそれぞれの状況を手早く把握し、同期を取ることが目的です。デイリースクラムがきっかけとなって様々なことに気がついたり、チームのアクションが取りやすくなったりすることがよくありますので、デイリースクラムは形骸化しないように注意してください。デイリースクラムの時間が15分を超えるようになってきますと、参加者はあれこれ時間の都合がつかないと言い出してデイリースクラムをスキップするようになってきます。時間を15分以下と設定しているのは、1日たった15分の捻出ですらも断る、ということは難しいからです。どんなに親しい人でも1

5分程度であれば時間が取れるはずです。ですのでできる限り短時間で終わらせるように心がけてください。

スプリントレビュー

スプリントレビューとはスプリントの最終日に手掛けたプロダクトインクリメントのチェックと、最終的に出荷されるプロダクト全体の評価行います。スクラムチームがユーザーや他のステークホルダーに対してスプリントの成果を披露し、評価を受ける場になります。出荷できるかの判断をすると同時に、完成しなかったもの、『完了の定義』を満たせなかったもの、合格しなかったものに対しての説明も行います。スプリントレビューの目的は、ユーザーとコミュニケーションをとる場としてフィードバックや協力を引き出すことです。その際に追加の要求はユーザーやプロダクトオーナーから出た場合は、プロダクトバックログに追加していきます。そしてユーザーやプロダクトオーナーからはマーケット状況や、プロダクトの利用状況などの共有も行い、次のスプリントに向けてプロダクトバックログ全体を整理します。

レトロスペクティブ

レトロスペクティブはスプリントレビューの後に行います。レトロスペクティブの目的は、今回のスプリント自体の進め方や、仕事の仕組み上の問題について振り返り、さらに上手く効率よく生産性の高い仕事ができるように、チームの仕事の在り方自体を見直すことにあります。こういった振り返りの場を持つと、大抵はうまくいかなかったところ、まずかった問題点、失敗やミスにばかり目が向いてしまいます。そうなると思考がネガティブになり発想力を奪われていくことになります。ですからスクラムマスターは、レトロスペクティブの最初は、うまくいったこと、学んだことについてメンバーに書き出すように促してみてください。そうすることによって、できなかったことばかりではなく、できたこと、できていることについても注意を向けられるようになり、「１００点ではなかったが８０点は取れた」と受け取れるようになれば、気持ちをポジティブにできますので、発想力を高められます。

レトロスペクティブというのは自分たちの仕事を外側から眺めるというプロセスですので、通常の思考の範囲から外に出なければいけません。その場合発想が柔軟な状態になっていなければなかなか思考の外に出ることは難しく、局所的な問題や、近視眼的な解決策しか思いつかなくなってしまいます。大抵仕組みによる問題の表出というのは、見えない構造部分に原因があります。直接表出している問題の近くにその原因があることはあまりなく、構造的な視点からみた要素の繋がりを辿っていったその先の離れた場所に、原因がある場合が多いのです。ですので仕事の仕組みそのものを振り返る際には、メンバーの心理状態をポジティブにしておいて、視界を広げて発想を柔軟にした状態で振り返るように促してみてください。

とても残念なことに、レトロスペクティブというイベント自体に「とくに価値を感じられない」という感想をもっているチームが多く、最も省略されてしまいやすいイベントです。私が今まで入ったプロジェクトの中でも、レトロスペクティブをもう行なっていない、というチームが半分以上でした。私はこのレトロスペクティブというものがアジャイル開発において最も重要なイベントと考えています。レトロスペクティブが行われていないチームという

のは、生産性を高めたり成長するという機会を逃していると考えてください。そのようなチームが、「今出来なかった事が将来できるようになる」と楽観的になるのは危険です。レトロスペクティブを復活させて自分たちの仕事の振り返り改善するという機会を定期的に設けてください。

3つの成果物

プロダクトバックログ

プロダクトバックログにはプロダクト開発に必要だと把握しているものを全てを入り込んだリストです。プロダクトに対する要求が並べられており、将来どのような変更の加えられる可能性があるのかについての情報源になります。プロダクトバックログはプロダクトオーナーが管理する責任を持ちます。プロダクトバックログにプロダクトバックログアイテムとして入れるのは誰が入れてもいいのですが、それら全てに優先順位を決めて並び替えをしておくというのはプロダクトオーナーが責任を持ちます。プロダクトバックログはその全てを

実現するものではありませんし、プロダクトバックログ自体がここまでで十分、と完成することもありません。プロジェクトの初期段階では理想的な機能やよく理解できる要求が並べられています。しかしひとたびプロダクトをリリースするとマーケットやユーザーの利用環境に合わせて進化していきます。プロダクトバックログは常に更新され、絶えず変化をしていきます。一度プロダクトバックログに入ったアイテムも不要になって取り除かれたり、内容が変更されたりすることもあります。

スプリントバックログ

スプリントバックログは、「スプリントの中で作り切れるであろう、と思われるだけのサイズのプロダクトバックログアイテムが基準になり、より小さく分解され、そしてそれによって追加で必要になる作業などがスプリントバックログアイテムとして積み上げられます。スプリントバックログは開発チームの管理するものですが、正しいスプリントバックログのあるべき姿というのは、今スプリントの中で、何をしようとしているのかについてすべてが可視化されている状態です。スプリントバックログの修正は開発チームが行います。修正の頻

度や粒度ですが日々のデイリースクラムの中で発見できるようなレベルのものです。これによってスプリントの進捗が把握できます。本スプリントで元々計画していたスプリントバックログアイテムのサイズに対して、その時点で残っているものの合計が、進捗を表しています。

出荷判断可能なプロダクトインクリメント

プロダクトインクリメントとは、既にリリース済みのプロダクトに対して今回のスプリントで追加する増分が足されたものを指します。出荷判断可能としていますのは、実際にスプリントの中で行う作業すべてが出荷対象になるわけではありません。例えばプロトタイプやモックアップなどを複数作ったりした場合は、その中で選択されるのはひとつですから、それ以外のものは捨てることになります。そしてその途中で作られたもの選ばれなかったものに関しては、製品として組み込まれませんので、出荷の判断ができません。また、動作しない仕掛かりのものも、動かすことが出来ませんので当然出荷の判断はできません。

スクラムをベースにしてさらに追加するもの

スクラムはシンプルなフレームワークですので、他のアジャイル流派から良いアイディアを拝借して、プロジェクトの特性に合わせて適宜追加してみることも検討してみてください。

私が追加するアイディアは次のものです。

- インセプションデッキ
- リスク管理表
- アジャイルモデリング
- ペアプログラミング・モブプログラミング

これらを追加することで、スクラムの規律性を損なうことなく、更に開発効率を上げることができます。それぞれについて別の章で紹介したいと思います。

機能横断的なメンバーで構成されたチーム

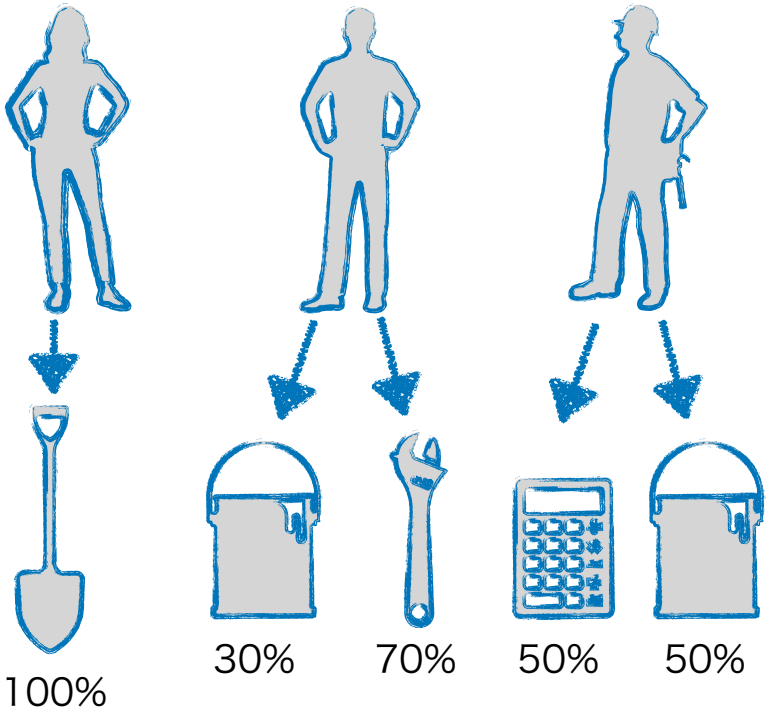
アジャイルでは、透明性が重要になります。リアルタイムで今何を行なっていて、何に頭を悩ませていて、これから先に何を取り込もうとしているのか、をチームメンバーは互いに理解をしておく必要があります。アジャイルでは機能横断的な働きを求めます。この機能横断的な働きを実現するには、各メンバーが何をしているかをリアルタイムで理解していることが重要です。そして各メンバーが何をしているのかを理解するためには、各メンバーが行なっている領域についての経験・知識が必要になってきます。たとえば、あるメンバーがデータベースの設計をしているという時に、データベースの設計はその人が常にあると他のメンバーが思っていたとしたら、他のメンバーはデータベースの設計をしているその人の作業に対して注意を払わなくなるでしょう。これから先もずっとデータベースの設計をしていく人がその人だけなのであれば、他のメンバーが知る必要ないからです。しかし実際にはその人が一時的に業務でいっぱいになって溢れてしまったり、体調不良で作業が滞ったりすることもありえます。その場合データベースを設計するという作業はストップしてしまう恐れがあります。他のメンバーがその業務を担えないという、特定業務に細分化されたチームとなってしまいます。これは機能横断的なチームと真逆のチームの状況です。アジャイル

における理想的なチームというのは、ソフトウェア開発という業務の中では、たった一人で全ての工程を担えるメンバーで構成されたチームです。全ての業務ができるエンジニアをフルスタックエンジニアと呼んだりもしますが、フルスタックエンジニアというものは、アジャイル開発の中において求められてきたエンジニア像でもあります。各開発メンバーが全ての開発工程を実践することができる、ということは、他のメンバーが行っていることに関しても、知識や経験があるということです。それによって、チーム内での情報共有は最低限のコミュニケーションでも同期を取ることができます。これによってある特定の工程の作業ボリュームが増えていると判断した場合、全員でそれに取り掛かって一気に終わらせるということが可能になります。データベース設計は特定の誰かがやるのではなく、ある機能では複雑なデータベースの設計が必要になるとなった場合、開発メンバー全員でデータベース設計に取り掛かれますし、その後のデータベースが絡む作業に関しても全員が理解した状態で進む事ができます。ですのでひとつの作業を一人が担当して最後までやりきるというスタイルというよりは、チーム全員で一つの作業を一気に終わらせて、そしてまた次の作業を全員で一気に終わらせてという流れで進めていきます。特にスクラムやカンバンでは、チームの中でマルチタスクになってしまうことを抑制し、なるべく仕掛かり中になっている作業

を減らし、確実にひとつずつ終わらせていくことを推奨しています。ひとつの機能に対して全員で取り掛かって一気に終わらせる。それによって限られた時間の中で、一つでも多くの機能をリリースできるようになります。五人のメンバーがそれぞれバラバラの機能を手分けして開発していった場合、五つの機能をそれぞれが仕掛かりの状態のまま完成せずにスプリントが終わってしまい、何一つ機能をリリースできない、という最悪の事態の起こる危険性が高まります。チームとしての流れを止めないこと、チームとしての流れをスムーズにしてさらに加速させること、このことにフォーカスをするならば、チーム内でマルチタスクをすることは、スプリントが仕掛品で詰まりを起こしてしまう原因であると認識し、全員でひとつの機能に取り組んで、早く完成まで持っていくことです。そうなった場合、誰かの手が一時的に空いてしまうこともあるかもしれません。だからといって一時的にでも手が空くようなことを避けるために、各メンバーが別の機能に関する作業をし始めると、一気にマルチタスク化が始まります。これはチームメンバーの稼働が可能な時間をリソースとして考え、いかに稼働率を高めるか、を優先した場合に起こる事態です。これを私は『リソース優先スタイル』と呼んでいます。これに対して、いかに早く、流れるように機能を完成させて製品に追加していくか、にフォーカスをして流れていく成果物の流れ方に注意を払うの

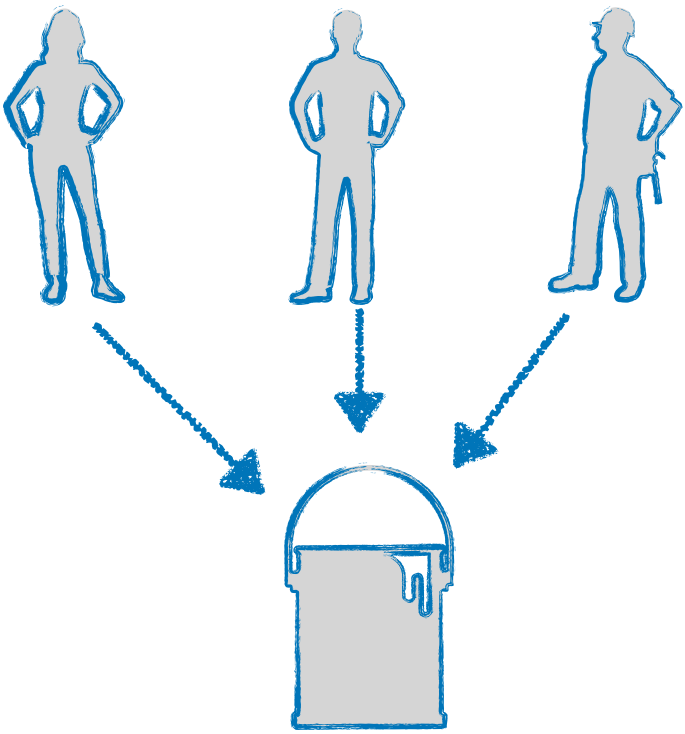
が『フロー優先スタイル』になります。アジャイル開発においては、このフロー優先スタイルで開発を進めていくことがチームの生産性を高める上で重要になってきます。これを具体的にどのように実践するかに関しては、スクラムボードの章で説明したいと思います。

✖リソース優先スタイル



各人の稼働率を100%にすることを優先する

◎フロー優先スタイル



成果を流す（完成に向かわせる）ことを優先

デザインスプリント

ビジネスモデルそのものもまだ固まっていない状態だったり、具体的に何を作っていいかというアイディアから考えなくてはならないという場合であれば、一番最初にデザインスプリントというものを行うこともあります。デザインスプリントでは、モックアップやプロトタイプを手早く作って、ユーザーグループに使ってもらうことで、その反応を観察して様々なヒントを得ます。ユーザーに直接インタビューをすることもあれば、質問票を用意して記入してもらうようなことも行ったりします。アジャイル開発では、ユーザーのことを深く理解することが重要ですので、ユーザーの行動を観察しうまく把握できるようにしておくことはとても有用です。

CI環境

アジャイル開発においては基本的にCI環境は必須です。テストの自動化も必須です。もしテストを自動化していなければ、開発をする作業工数よりもテストにかかる作業工数のほうが遥かに大きくなってしまい、そしてそのテストの品質も低いものになってしまいます。短いスプリント期間の中で高品質な製品をリリースし続けるためには、それらが実現できなければいけません。ですので、CI環境でテストの自動化が行いにくいようなアーキテクチャを採用した場合は、アジャイル開発を行うことは現実的ではないでしょう。ですのでレガシーシステムをそのまま引き継いで、CI環境もテストの自動化もできないような状態でアジャイル開発を行おうとするのは危険です。COBOLで書かれたパッチプログラムがたくさん連なったような レガシーシステムをそのままのアーキテクチャでアジャイル開発を行おうとするのではなく、アジャイル開発が可能なアーキテクチャを用意して少しずつ段階的に機能を置き換えていき、周辺から巻き潰していくように移行していくのがいいでしょう。ここ最近のモダンなプログラミング言語であれば、CI環境でテストの自動化を構築することは可能ですから、アジャイル開発においてはそういったモダンなプログラミング言語が採用されることは必然といえます。逆に言えば、どんなプログラミング言語やどんな

アーキテクチャであってもアジャイル開発が適用できるとまで言い切ってしまうのは少し乱暴な気がします。アジャイル開発というものが存在しなかった時代に生まれたプログラミング言語やアーキテクチャは、アジャイル開発においてはあまり積極的に採用するべきではないと考えます。

アジャイルへの誤解から生まれる新たな問題

アジャイルの悪用

最近ではアジャイルのさまざまな認定資格の仕組みが用意され、たった数日間のトレーニングに参加するだけで認定が得られる資格なども人気です。トレーニングに参加すること自体は問題ではありません。問題なのは、開発プロセスを修正するだけでソフトウェア開発はうまくいく、などと誤解させてしまうようなトレーニングの売り込み方に問題があります。

「ソフトウェア開発における最大の問題点は人であり、開発プロセスは修正可能なので、アジャイルという反復型の開発プロセスにすれば、うまく人を管理さえすればこのプロジェクトは成功する」などと思わせてしまうことで、アジャイルを導入しても、マネージャーはメンバーを管理することを手放せない状態になってしまいます。

そんな状態のマネージャーにとってアジャイルの透明性は気に入ること間違いなしでしょう。開発チームが何を行なっているのかが包み隠さず見えることを悪用できるからです。そしてマネージャーにとって都合のいいアジャイル、というものが出来上がります。しかしそ

のようなものは決して真のアジャイルではありません。アジャイルという名を口にしていても、その実態はアジャイルではなく、マイクロマネジメントによる統制なのです。

そうしたマネージャーが（自分自身であっても、外部のコンサルティング会社に依頼してでも）、開発スケジュールを計画して、開発チームに押し付けます。開発チームは作業を見積もる自由はもはや残っておらず、押し付けられたスケジュールに収めることが求められます。開発メンバーがまだいないプロジェクトで、開始前の時点からすでに1年分のイテレーションに割り当てられたユーザーストーリーが決まっているプロジェクトも多く見てきました。このようなプロジェクトでは、イテレーションに割り当てられたすべてのユーザーストーリーをチームが処理できなければ、その遅れを取り戻すことを強く求められることになり、次のイテレーションで開発メンバーが負担を強いられることになります。こうしたマネージャーがプロダクトオーナーになっているチームは悲惨です。スクラムマスターという肩書きを標榜していたマネージャーもいましたが、このような振る舞いをしている人をスクラムマスターと呼ぶのは、強い違和感を覚えます。

開発メンバーがテストの自動化、ペアプログラミング、リファクタリング、モデリングをしていたら、プロダクトオーナーがそんなものは時間のムダだと言って、それらをやめるように命令した、というプロジェクトもありました。開発チームに、アーキテクチャや設計は求められていない、と言い切ったプロダクトオーナーもいました。このようなチームに、戦略的に技術を考える余裕は与えられていません。プロダクトバックログで最も優先順位の高いアイテムに目を向けて、できるだけ速く完成させることだけが求められていました。このようなことをしていると、技術的負債が蓄積していき、どこかで大幅な生産性の低下を引き起こします。マイクロサービスに挑戦しようとしたかつての希望はその形跡すら残っていない、脆弱で巨大なモノリシックな分散型システムが出来上がります。バグについての議論ばかりで、機能追加について考える余裕はもはやありません。本番環境へのリリースはもはや頻繁とは言えないレベルにまで減っており、手動テストだけのイテレーションが用意され、リリースサイクルの間隔はどんどん長くなっていきます。

アジャイルを導入することで問題を解決できるという淡い期待はもはやどこにもありません。マネージャーは開発メンバーの開発スピードが遅いことを事あるごとに非難します。開

発メンバーは、戦略的に技術を考えることを認めなかったプロダクトオーナーのことを裏で非難します。プロダクトオーナーは、自分のことをチームの一員と考えておらず、開発メンバーと自身が平等であるべきという理念を支持していません。元々の組織に蔓延していた「指示者ｖｓ被指示者」の文化が勝ってしまった末の状況です。私はこのようなプロジェクトに入った際にまず、開発メンバーを助けるためにも「これはアジャイルではない」と言い切るようにしています。

柔軟性のある高品質なソフトウェアを、継続的に開発し続けるためには、高度なエンジニアリングスキルが必要です。単に透明性の高い協調的な環境を作るだけで、こうした能力がチームに身に付くと考えるのは非現実です。アジャイルの開発プロセスを導入するだけではなく、開発メンバーの開発スキルを高めるためのサポートが必要です。コーチング、トレーニング、自己学習、実験的サイドプロジェクトなどを組み合わせることで、そういった環境を提供できます。どれだけすばやくソフトウェアを進化させられるかを考える上で、開発メンバーのスキルを育成することは、切っても切り離せない要素なのです。アジャイルの本質は成長であり、自己組織化なのです。

悪意なくアジャイルという言葉が悪用されているシーンを見る度に、真のアジャイルの実践というものの難しさと、その功罪について考えさせられます。シンプルで簡単そうに見えるがゆえに、導入して終わり、という売り込みが横行していることについても、多くの開発者は辟易しています。ですから、なおさら真のアジャイルの普及と浸透というものについて私の果たす役割は大きいと自負しつつ、まだまだ力及ばずと感じることも多いのです。

アジャイルへの過度な期待

そして、アジャイルというのはソフトウェア開発業に限定したものではなく、あらゆる産業で適用できるものだ、と宣伝されているのを目にすることもありますが、アジャイルというものはソフトウェア開発業だからこそ高い成果を発揮できると言えます。その他の産業ではアジャイルという手法は高い生産性を発揮することは難しい場合が多いと思います。建築業や農業・酪農・水産業、他にも様々な産業などでもアジャイルというものが提供できるかと言われると、適用できなくはないかもしれませんが、ソフトウェア開発業以上に高い生産性を出すのは難しいと思います。ここを理解してもらえると、アジャイルによって最も高い

生産性を出せる産業であるソフトウェア開発業において、アジャイルを採用しないということがどれだけもったいないことか、ということも理解してもらえるのではないかなと思います。

第二章のまとめ

- アジャイルは学習と成長が中心にある。その結果、生産性が高まり、速さを実現できる。学習する速さも高まり、さらに成長する。
- 流れに注目する。稼働率１００％を目指すのではない。リソース優先ではなくフロー優先。
- スクラムはわかりやすいアジャイルフレームワークなので、広く普及しており、事例も多い。
- テスト自動化
- テストツール、オーケストレーションツール、プロビジョニングツール、仮想マシン、などを用いて、ビルド、デプロイ、テストをCI／CD環境によって自動化することで、アジャイル開発がより効率的になり、柔軟で堅甲なシステムを高速で開発できるようになる。CI環境なしでアジャイル開発を実現した場合、生産性が落ちる場合が多い。最低でもテストは自動化しておくべき。手動でテストをするのはなるべく避けよう。手動でテス

トをするにしても極限まで減らすべき。しかしテスト自体をなくしたり削ったりするべきではない。

- アジャイルを誤用しているプロジェクトは実際にはかなり多い。悪用する意図があってのことなのかは分からないが、マイクロマネジメントのためにアジャイルを導入したとしか考えられないようなチーム状況になっている場合もある。アジャイルはマネージャーによって導入されるが、決してマネージャーにとって都合のいいチームを作るためのものではないし、そのように用いるべきでもない。
- アジャイルをどんな産業構造でも適用できるという声には慎重になること。アジャイルはソフトウェア開発において最も高い成果を発揮できる。他の産業構造でも、適用は可能かもしれないが、そもそもアジャイルで取り組むメリットはソフトウェア開発に比較してそれほど大きなものはないはずである。それほどまでにソフトウェア開発におけるアジャイルの効果は大きい。アジャイルに開発しないソフトウェア開発はその価値をドブに捨てているようなものである。まず競争に勝てない。

第三章：アジャイルコーチ・スクラムマスターとは一体何者か

チームファーストの精神

皆さんのチームにはアジャイルコーチやスクラムマスターはいるでしょうか？スクラムを導入しているチームであればスクラムマスターはいると思いますし、もしあなたが質感をマスターをした経験があるとしたらすでに必要な知識について学習していると思います。しかし実際にはこのスクラムマスターと言う役割は非常にとらえどころが難しく、一体何をする人なのかを理解することが難しい役割でもあったりします。

どのように振る舞うのが良いスクラムマスターなのでしょうか？そもそもなぜアジャイルコーチやスクラムマスターという役割が必要なのでしょうか？

アジャイル開発というのは、チームのもたらす価値を最大化するために、人が持つ能力というのを最大限に活かしましょうというものです。ですので、チームとしての価値を高めるためにはそこに集まるメンバーというのはなるべく多様性が高い方がいいです。様々な経験様々な価値観を持った人達が集まった方が、うまく機能させた時のチームとしての生産性は高い傾向にあります。しかし個々人の能力が非常に高かったとしても、チームとしてうまく

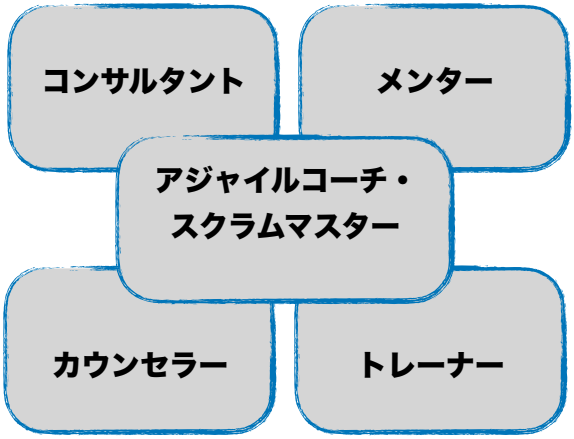
機能させられなければ、思ったほどのパフォーマンスが出ないどころか、最悪の場合、チーム自体が空中分解してしまう恐れもあります。特にソフトウェア開発の世界では、非常に能力の高いエンジニアが、コミュニケーションも卓越しているとは限りません。人とコミュニケーション取るのはそれほど得意ではないがプログラミングコードを書かせると素晴らしい力を発揮するというエンジニアは多いです。逆にコミュニケーション能力が高くてもソフトウェアを作る力が低ければ、チームとして提供できる製品の価値はあまり多くは望めないでしょう。

五人のメンバーでチームを組んだ場合、五人の一人一人の能力を足し合わせた総和以上の価値を、チームが提供できるようにするのがアジャイルコーチやスクラムマスターの役割です。そもそもなぜチームで開発をするのかというと、個々人を足し合わせた成果以上の成果をチームであれば出せるからです。そして個々人では対処できないリスクに対してもチームでリスクを吸収して開発を継続し続けることができます。個人で開発をしていれば、その人がわからない領域だと著しく開発のスピードが落ちますし、もしも体調不良や不測の事態で作業が続けられないようなことになれば、開発がストップしてしまいます。チームで開発を

することによって、そのような事態にもうまく対処することができるようになります。しかしもしチームメンバーが 個人の成果を優先してチームのことを考えずに動き出したりチームのことを考えたコミュニケーションを取らないようになったりすると、チームとしての成果は出なくなります。実際には開発能力の高い天才エンジニア・天才プログラマーとしての圧倒的な開発能力と引き換えに、チームで協働するためのスキルというものに多少難があったりします。ですのでそうしたメンバーをチームとしてうまく機能させるために支援をしていく役割というのが必要になるのです。アジャイルコーチ・スクラムマスターは、そうしたチームファーストの精神をチームメンバーに理解させ、そして自らの役割も明確にして説明するが必要があります。

アジャイルコーチ・スクラムマスターに必要な能力

アジャイルコーチ・スクラムマスターに必要な能力を、類似している四つの職種で求められる能力に分解し、それぞれの職種と何が違うのかという視点で四つに分けて説明していきたいと思います。



コンサルタント

コンサルタントは問題を分析し診断するということが主な仕事になります。実際に問題を解決してしまうコンサルタントというのもいるかとは思いますが、基本的には分析することがメインです。

クライアントとの関係性では基本的には依頼を受けてから仕事を開始することになりますので依頼もされていないのに勝手に分析をするということはありません。コンサルタントはある特定の分野に精通をしておりクライアントにその分野に関する情報を提供し可能な選択肢を示唆します。コンサルタントは洞察力を駆使してクライアントの理解に努めます。コンサルティングで成功するには指導的かつ批判的なアプローチを取ることになります。分析し問題を診断し助言を与え実現までその段階別プロセスのフォローアップに努めます。

それに対してアジャイルコーチ・スクラムマスターはまずメンバーが抱えている問題、チームを抱えている問題をどう分析すればいいか、どう捉えればいいかということを、メンバー

自身が分析できるように支援します。時には環境を整えたりしながら分析が必要であることを促したりします。ですのでアジャイルコーチ・スクラムマスターは問題をどう解決すればいいかという、その問題に対する専門的な知識があったほうがいいですが、それはその方が問題を診断する切り口を見つけやすく促し方がより的確になるためです。しかしアジャイルコーチ・スクラムマスターが直接問題を診断してしまうということはなるべく避けるべきです。

メンター

メンターとは特別なスキルや知識あるいは経験を持つ、いわゆる師匠と呼ばれるものであり、経験が浅いで死にその知識やスキルを伝達する役割となります。

両者は対等の関係ではなく、非常に親密な個人的関係で結ばれていることが一般的です。経験からの助言や 個人的体験談を話したりすることで、経験に基づいた具体的な情報や専門知識を与えていきます。経験を伝えることによって成功する確率を上げたりあるいは無駄な失敗をしなくてもいいように導いてあげるというものです。

それに対してアジャイルコーチ・スクラムマスターはメンバーとは対等関係です。経験や助言というのを使うことはありますがそれによってそのアジャイルコーチ・スクラムマスターが経てきた経験をそのまま経験させようと思ってしまったりあるいは助言をメンバーが常に求めてくるという状態にならないようにしないといけません。メンバーがこれから経験するであろうことを先に経験しているということは有利ではありますが、それらを直接押し付けないように気をつける必要があります。

カウンセラー

カウンセラーは心理的に行き詰まっている人の問題や、心の傷、心理的な妨害要因などについての原因の追求に焦点を当てます。カウンセラーはクライアントが機能不全に陥ってしまう過程やその心の傷を癒す治療のプロセスを理解して初めてその人の個人的な痛みを解決することができます。カウンセラーは「クライアントにはどこか悪いところがあり、セラピーによってそれを治す」という前提に基づいています。クライアントの自我の強さを高め

て現実を直視させ「もう大丈夫だ」と言える状態に導くことに焦点が当てられます。言葉だけでコミュニケーションをするに留まらずのそのクライアントの表情だったり仕草だったり報道を通じて読み解くということも必要になってきます。そしてカウンセラーとクライアントは対等の立場ではなく上下の力関係が築かれることになります。

しかしアジャイルコーチ・スクラムマスターは心理的な行き詰まりを直接解消するのではなく、その心理的な行き詰まりが何から発生しているのか、どのようにすれば解消に向かうことができるのか、という間接的なアプローチを考えます。そうすることによって主従関係が生まれないようにし、対等関係を維持することに努めます。なぜならカウンセラーは直接対峙するのは基本的に一人ですが、アジャイルコーチ・スクラムマスターはチームメンバーに対して対峙するため、チームの中でたった一人であるアジャイルコーチ・スクラムマスターに対して複数人いるメンバーが依存してしまう状態が生まれてしまうと、アジャイルコーチ・スクラムマスターを介してでないと他のメンバーとうまくコミュニケーションが取れないような状態になってしまいます。そのアジャイルコーチ・スクラムマスターの前でないと自分らしくあれない、自分の意見が言えないという状況を生み出しかねません。チーム

の中で自分の意見を述べることができるようになるためには心理的安全性というものが必要になってきますが、もしカウンセラーとして振る舞ってしまった場合、チーム内での心理的安全性を生み出すすことが難しくなります。アジャイルコーチ・スクラムマスターの前では心理的安全性はあるが、他のメンバーの前では心理的安全性がない、といったような状況です。心理的安全性をチームの中でいかにうまく作るか。これがチームとしてのパフォーマンスを引き上げる最も大きな要因になりますので、カウンセラーとしての知識は持っていても損はありませんが、しかしそれに頼ってチームメンバーが抱える心理的な行き詰まりを直接解消しようとするような直接的なアプローチを避けてください。

トレーナー

トレーナーは新しいスキルの開発に焦点を当ててアプローチをする人です。指導する、発表する、質問する、など様々なやり方があり、その中心は、実践、訓練、学習になります。これらのトレーニングを通してクライアントは新たなスキルを身につけ、その知識を職場や他の場面で応用できるようになります。特定の分野の専門家であるトレーナーがクライアント

に様々な訓練を施し、学習を行わせます。そして最終的にクライアントをテストすることでその実力を評価します。トレーナーは基本的に指示を出して、その指示に従ってもらうというスタイルで成果を出そうとします。

それに対してアジャイルコーチ・スクラムマスターは、開発メンバーがどのような成長をしたいのか、そしてその成長を成し遂げるためにどのようなメニューがいいのか、どのようなプログラムが必要なのかは一緒に考えます。予めアジャイルコーチ・スクラムマスターがトレーニングメニューを作ってしまってその通り従わせるというやり方は避けた方がいいです。開発メンバーが自分達でどのように成長すればいいかということを考えやすくしてあげる状況を作ることに焦点を当てます。

これらがそれぞれの近しい職種との類似点と相違点ですが、時と場合によってはよりコンサルタント的に振る舞った方がいいシチュエーションだったり、よりメンター的に振る舞った方がいいシチュエーションだったりすることもあると思いますので、軸足だけを中立に残

しつつ状況によって少しだけその振る舞いを変えるといったこともいいと思います。重要なのはバランスをとることであり、それぞれの領域に踏み込んで振る舞ったとしても、いつまでもその領域に踏み込み続けるということは避けて、なるべくニュートラルであるようにすればいいと思います。常にアジャイルコーチ・スクラムマスターというのはチームとしての成果を最優先で考えますから、そのチームとしての成果を考えた上で、今はこの限定的なシチュエーションにおいては例外的にこのような振る舞いをした方がいい、といったような判断をするのであればそれぞれの領域により偏った行動を一時的に取るというのはありだと思います。

アジャイルコーチ・スクラムマスターというのは複数人を対象としたチームに対してという点で、他の職種とは振る舞いが異なってきます。基本的にはアジャイルコーチ・スクラムマスターは自分が受け持った各チーム内でそのチームの成果を最大化するためにどうすればいいか、と考えますが、複数チームで構成される大規模開発などでは、それぞれのチームにアジャイルコーチ・スクラムマスターがいるということもありますから、さらにもう一つ大きな視点で複数のチームをまとめた大きな括りでのチームとしての成果についても考え

る必要が出てきます。各チームのアジャイルコーチ・スクラムマスターが集まり全体をチームと考えた場合の成果と、それぞれの自分たちがいるチームの成果をどううまく結びつけるか、ということを考える必要があります。

心理的安全性の重要性

アジャイルコーチ・スクラムマスターが最も初めに着手するべきは、心理的安全性を高めることです。心理的安全性というのは、自分が思っていることを自由に言っても過度に批判を受けたり人格的な攻撃を受けない、という安心感のことを指します。この人たちの前であれば自由に何でも言える、自分たちの意見を率直に言える、そういったチームの状況を生み出す必要があります。

ハイパフォーマンスなチームというのはこれらができているチームです。自分たちの考え方や意見というものを率直に言ってその正しいか間違っているかと議論を活発にする上では、そもそもその意見を言ったことによってチームからその意見を言ったことそのものに対して攻撃を受けない、と安心できる状況というのが重要です。意見の内容について何らかの批判を受けたり議論の対象になったとしても、発言すること自体は問題ない、むしろ積極的に発

言をしてもいい、といった状況を作ることがまず一番最初に取り組むべきことです。心理的安全性が高いレベルで確保されたチームを作るのです。

多くのマネージャーはいまだに、不安や恐怖には「やる気を出す力」があると、意識的にも無意識的にも信じ込んでいたりするため、特に注意が必要です。彼らはこう信じています。「人は恐怖を感じることによって、望ましくない事態を回避するために、熱心に仕事をする」という具合に。そして、恐怖よって組織の成果が上向き、事態は良くなる、と。しかし実際には、学習や協力をしなければ成功できない仕事においては、不安や恐怖がやる気を引き出す要因になることはありません。不安も恐怖もは学習を妨げます。神経科学の研究によると、不安や恐怖のせいで認知リソースが消費されてしまい、ワーキングメモリ（作業記憶）の管理や新情報の処理をする脳領域に、資源が届かなくなることが分かっています。そのせいで分析的思考、創造的考察、問題解決ができなります。不安による認知リソースの消費こそが不安を感じている人が最高の仕事をしにくくなる理由なのです。

アジャイルコーチ・スクラムマスターは、心理的安全性を高めるために、チーム内の力関係をフラットにするようにしましょう。ヒエラルキーによる不安があると、心理的安全性は低くなります。そしてアジャイルコーチ・スクラムマスターは往々にしてマネージャークラスの人が任命される場合が多いので、注意が必要です。それはアジャイルコーチ・スクラムマスターというものに対する理解不足からでしょうか、スクラムチームを管理する役割、と誤解されていることが原因の場合が多いように思います。研究によれば、ヒエラルキーが下位のチームメンバーは一般に、上位のメンバーほど安全性を感じないといいます。私たちが絶えず相対的な地位を評価し、他人と比べて自分がどうであるかを、無意識に脳はチェックしていることも明らかになっています。さらには、ヒエラルキーの下位の人は、上位の人がいると考えるだけでストレスを感じてしまうのです。

ですので、アジャイルコーチ・スクラムマスター自身がヒエラルキーを生み出してしまっていないかどうか、について注意を払うべきです。ヒエラルキー上位の人にとってみれば、ヒエラルキー下位の人たちが不安を感じていることについて想像もしませんし、なかなか気付けません。人は自分が感じている感覚を通して他人を見ます。ですので相手がどのように感じているのかについても、自分が感じている感覚を適用して相手もそう感じていると錯覚し

ます。自分が不安を感じていないからといって、相手が不安を感じていないと言い切れるわけではないことを理解することが大事です。アジャイルコーチ・スクラムマスターは自分が居心地のいいチームを作るという誘惑に負けてはいけません。アジャイルコーチ・スクラムマスターにとって居心地の良いチームと、真に良いチームは必ずしも一致しないことを心がけておきましょう。

プロダクトオーナーへの提言

チームの雰囲気づくりにおいてプロダクトオーナーの影響は非常に大きいです。プロダクトオーナーのたった一つの言動で、チームの雰囲気が一変してしまうこともあります。開発メンバーに対するプロダクトオーナーの影響力が大きいチームでは、開発メンバーはプロダクトオーナーの言動を通して考え方や価値観を読み取り、その影響を受け、自らの言動に反映させていきます。心理的安全性を形成する過程でプロダクトオーナーの影響力は重要な要素として考えるべきです。

にも関わらず、多くのプロダクトオーナーが心理的安全性を形成することに成功していないのはなぜでしょうか。その理由は、プロダクトオーナーが他の開発メンバーよりも地位が高い場合が多いからです。リーダー職と所属メンバーの感覚にはズレが生じやすいという「構造的問題」があることを、アジャイルコーチ・スクラムマスターはプロダクトオーナーに理解してもらうように働きかけましょう。

人は、社会的地位が高まれば高まるほど、周りの要求に応えずとも自分の考えが実現しやすくなるため徐々に周囲の反応に疎くなってしまい、空気を読む力が鈍ってきます。これがさらに悪化するとまずいことに、組織の集団圧力に対しても抵抗しやすくなるため、全体から逸脱した行動もできてしまうようになり、長期的には全体の利益を毀損しかねないスタンドプレーに走ることにも繋がります。こういった状況に陥ってしまっている場合、本人はそのような言動に至っているとは思っておらず、それを指摘したとしても、本人が認めることは難しいです。「私はそんなことをしているつもりはない。言いがかりだ」となるわけです。

他方、自身の社会的地位がプロダクトオーナーよりも低いと感じている場合の開発メンバーは、空気を読み、周りと歩調を合わせることで居場所を作っていこうとします。そのため、周りの言動や職場の動向に自然と目が向き、周囲を注意して観察することによって、結果としてプロダクトオーナーと比べてより正確に職場の雰囲気を把握できます。プロダクトオーナーと開発メンバーでは、このズレが自然に生じるために問題が起こります。プロダクトオーナーと開発メンバーでは感じている感覚が違うのです。そのせいによる共通の認識をもつことの難しさが、プロダクトオーナーが積極的に心理的安全性の確保を進めようとしない原因となります。

しかし、チームの心理的安全性の改善はプロダクトオーナーのサポートなしでは不可能です。感覚にズレの生じる構造的な傾向がある一方で、プロダクトオーナーが積極的に動かなければ心理的安全性は高まりません。これが心理的安全性におけるアジャイルチームのパラドクスです。プロダクトオーナーがこの問題に取り組むには、自分の感覚のズレは自然と生じてしまうことを自覚し、自らの努力のみでは解消できないことを認識する必要があります。アジャイルコーチ・スクラムマスターはこの点をうまくプロダクトオーナーに伝えて理

解させてみてください。レトロスペクティブの中で、内省を促すように仕向けるのもひとつの手です。開発メンバーそれぞれに、チームの心理的安全性に対してどのように評価しているかを各々で点数付けさせることによって、開発メンバーとプロダクトオーナーの感じ方の違いを明るみにしましょう。加えて、チーム内だけではなく、それ以外のチームのメンバーなどからもフィードバックをもらうなどして、プロダクトオーナーに周りの声が届くような構造を形成しましょう。つまり、プロダクトオーナー個人の問題として捉えるのではなく、チームの問題として捉えることで、チームに対して、心理的安全性を阻害している構造的な問題に向き合う姿勢をもたらすのです。アジャイルコーチ・スクラムマスターは常にチームの心理的安全性が守られるように努力してください。

チームを信頼する

そして二番目に取り組むべきことは、チームで起こった問題はチームメンバーで解決ができる、とチームを信頼することです。起こった問題そのものを解決しに行こうとしてしまうのは、アジャイルコーチ・スクラムマスターの仕事ではありませんから、メンバーには問題を解決する能力があり、その能力を引き出すためにどのような支援をすればいいのかと考えるのです。

そのためにもなるべく問題からしばらく距離を置いて、どのようにすればメンバー達がうまくその問題を解決できるだろうか、その問題を解決できるようになるためには必要なツールや必要な環境というものは何だろうか、と考えます。チームで問題が解決できない場合はマネジメント層にエスカレーションする、という経路も当然確保しておくべきですが、あまりにも早い段階で、それも頻繁にエスカレーションを行なってしまうと、それが常態化してチームで解決しようとしなくなるでしょう。特にプロジェクトが始まったばかりの頃は、まだチームメンバーがうまく機能しきっていませんので、問題が起こった場合についエスカ

レーションしたくなってしまうものですが、少しだけその誘惑を我慢してチームには解決できる能力がある、と待ちましょう。

速さをもって解決する

そして三番目は、問題の大半はほとんどが速さを持ってすれば解決できると認識し、いかにすれば早く行動ができるか、を考えます。ほとんどのプロジェクトが失敗する理由は「遅いから」です。その場合の遅いというのは開始するスタートが遅いという場合もあれば、開発をするスピードが遅いこともあります。いかに早く作るか、いかに早く行動するか、いかに早く意思決定をするか、速さに対してこだわりを持つようにします。その速さを実現する上で、阻害する要因は何なのか、その阻害する要因をどうすれば取り除けるかを考えます。

見たくないものを見せて聞きたくないことを聞かせる

アジャイルコーチ・スクラムマスターにとってチームメンバーが目標達成したいという意欲・モチベーションが高ければ高いほど喜ばしいと思いますが、その場合においてアジャイルコーチ・スクラムマスターは都合のいい情報だけを与えてモチベーションを上げるようなことをするのではなく、メンバーが見たくないと思うような情報や聞きたくないと思うような情報、いわゆるネガティブな情報もきちんと伝える必要があります。

心理的に安全な環境で仕事をするということは、感じよくあるために、誰もがいつも相手の意見に賛成し迎合することではありません。

ネガティブと受け取ってしまうような情報であったとしても現実にそれが起こっているのであれば、それらの現実にメンバーを向き合わせましょう。ユーザーからの不満たっぷりのフィードバックだったり、トラブルを予兆するような情報だったり、聞くと気持ちが落ち込んでしまうようなものであったとしても、目標を達成するために、まず現実を見せるということを徹底します。見たくないものを見せ、聞きたくないものを聞かせよう、と思うと、

チームの心理状態がネガティブだとそれを受け入れることはなかなか難しいです。ですのでネガティブな情報を受けられるようにチームをポジティブな状態にしておくことが必要になります。チームメンバーの心理状態をポジティブにしておくというのは、チームメンバーが現実を受け止める上での前提だと思ってください。メンバーの心理状態を考慮せず見たくないものを見せる、聞きたくないことを聞かせるということをあまりにも優先しすぎると、結果的にメンバーはそれらを無視するようになっていきます。目標を達成する上で、チームをポジティブにした状態を維持し続けるということはとても大事になってきます。アジャイルコーチ・スクラムマスターはこのメンバーが見たくないものを見るためにはどうすればいいか、聞きたくないことを聞かせるためにはどうすればいいか、と考えてください。

メンバーと話す際には意見と真実は明確に切り分けて説明をする必要があります。アジャイルコーチ・スクラムマスター自身が、率直で誠実で、真実をきちんと語って、そして自分の意見を述べるときは「これはあくまでも自分の意見なので、もしかすると自分の意見は間違っているかもしれないし、ただ今は自分自身の解釈の上からこのように感じているということを伝えている」ということを表明するといいでしょう。心理的安全性は、率直であると

いうことであり、建設的に反対したり気兼ねなく意見を交換し合ったりできるということなのです。どんな職場でも、対立は必ず起きます。それでも心理的安全性があれば、異なる意見を持つ者同士が、どこに納得がいかないかを率直に話せるようになるのです。

そこに存在しているということが重要である

会話を通じて、質問する、傾聴する、賞賛する、肯定の気持ちを表現する、といった行為を行い、チームメンバーに安心感を与えようとすることが大事です。チームメンバーに対する関心を述べ、あなたを支援したい、あなたの幸福感と能力を高めたい、という意欲までをも表現できるまでになればかなりの上級者です。

私はチームのためにここにいます、必要があればいつでも連絡ください、といった言葉を用いることで、チーム内での存在感を示します。自分の意見や主張を前面に押し出すのではなく、ただそこに立って寄り添っているだけ、しかしチームに寄り添う形で常にそばにいると

いう強い存在感が出せるようになれば、一流のアジャイルコーチ・スクラムマスターに近づけるでしょう。

体験を想起する

たとえ話や物語を語ったりすることで、実際にあたかもそのシチュエーションを体験してるかのような感覚をメンバーに提供することができれば、起こらなかった未来についても想起させることができます。これらは将来を予測してリスクを洗い出したりする場面や、開発する機能の優先順位を考える際にも有効です。これから先このプロジェクトはどのようなトラブルに直面するだろうか、プロダクトをリリースした後にユーザーの生活はどのように変化するだろうか、など体験を想起することによってメンバーの想像力が高まります。想像力が高まれば価値の高い選択肢を見つけやすくなり、行動に対しても前向きになれます。

アジャイルコーチ・スクラムマスターを目指す人へ

とらえどころの難しいアジャイルコーチ・スクラムマスターについて説明をしました。スクラムマスターについてはスクラムというフレームワークの中でこのような役割を担いましょうなどと説明されていたりしますが、スクラムマスターを志す人、あるいはその役割を担おうと決めた人であれば、どこかのテキストを参考にしたり、あるいはスクラム研修を受けたりしているかと思いますので、本書ではそうした一般的に手に入る情報に関しては増えませんでしたが、そもそもなぜスクラムマスターが必要なのか、あのようなスタイルなのか、といった視点であまり語られることのない面から、説明をしてみました。

アジャイルコーチ・スクラムマスターはチームメンバーに忠告や助言を行うのではなく、各メンバーのパフォーマンスを引き出すことに集中します。メンバーのパフォーマンスに焦点を当ててより質の高い行動を取れるように促します。そのために次のような質問を用意しておくといいでしょう。

メンバーのパフォーマンスを引き出す質問

- 自分のパフォーマンスを高めるためにあなたが必要としているものは何ですか
- あなたが知らなければならないあるいは学ばなければならないことは何ですか
- あなたが実行しなければならないことは何でしょうか
- 必要としているものをあなたは過去に一度でも体験したことがあるでしょうか
- もしあるとしたらそれはいつどのようなことを通してでしょうか

アジャイルコーチスクラムマスターにとって必要なスキル

1 能動的な傾聴
2 観察力

3　深い信頼関係を築く力
4　存在感
5　質の高い質問
6　フィードバックを与えるテクニック
7　フィードバックを受け取るテクニック
8　体験を想起する力

第三章のまとめ

- アジャイルコーチ・スクラムマスターはチームファーストの精神をチームに吹き込む役割である。
- コンサルタント、メンター、カウンセラー、トレーナーの技能を持ちつつも、しかしその技能の使い方はそれらとは異なる。
- アジャイルコーチ・スクラムマスターの最初の仕事はまずチームの心理的安全性を確保することである。真的安全性を高めて、それを維持せよ。
- チームを信頼するということは、メンバーには問題を解決する能力があり、その能力を引き出す事で、チームは問題を解決できる可能性が高まるように支援することである。そして期待する速度でそれが達成できなくとも、できなかったことよりも、できたことに注意を向けること。
- 各チームメンバーの体験を想起する事でチームの想像力を高めよう。ストーリー形式で語ろう。
- メンバーのパフォーマンスを引き出すことに集中しよう。質の良い質問をしよう。

完全版として「成功するアジャイル開発」に全10章が収録されています。
そちらをお求めください

あとがき

私がアジャイル開発なるものを知ったのはいつだろうか、はっきりとは覚えていませんが、２００４年だったと思います。その頃は私はまだアメリカから帰ってきたばかりで日本で情報システム系の専門学校で教師として働いていた頃でした。プログラミングからシステム設計、ハードウェアを含むコンピューターシステム全般を授業で教えていました。当時の私はまだアジャイルというキーワードの意味を大して理解していませんでしたが、ＸＰという手法について、かなり先鋭的で挑発的な取り組みに思えただけで、特に深入りしませんでした。ただ、ウォーターフォールモデルは破綻しているとは感じていました。卒業生が就職した先で疲弊している話や、システムインテグレーションにまつわる業界の話の中で、すでにウォーターフォールモデルは悲劇の元凶とされていましたから。私が担任として受け持った生徒を卒業させると同時に私もソフトウェア会社に就職しました。２００６年のことです。そこはERPパッケージソフトウェアを自社開発する、かなり攻撃的でカオスな会社でした。配属された部署は一癖も二癖もある猛者たちで、開発エンジニアたちが自ら要件を定義しコードを書きテストを行っていました。そして一定の間隔でリリースし、機能追加をしていました。創業して5年で上場を果たして、国内の大企業向け人事給与システムの市場でトッ

プシェアを取るほどの高い評価を受けていた一見固そうな事業ドメインの印象とは違い、高速に開発することを求められつつも、開発者それぞれに大きな裁量権がありました。私も配属されてすぐに要件を書き、それをもとにコードに書き、テストに流し、戻され、コードを修正し、を何回か繰り返したのち、製品はリリースされていきました。ユーザーは大企業の人事部。３０００社を越える企業に使われるパッケージソフトウェアの開発は、非常に高いプレッシャーが襲ってくるはずなのに、開発エンジニアの誰もがユーザーと直接やり取りし、ユーザーを恐れていませんでした。継続的に製品はインクリメンタルにリリースされていく、まさしくその姿はアジャイルでした。しかし当時はアジャイルという言葉を用いていませんでした。無償でバージョンアップを行なっていくという当時としては画期的なビジネスモデルゆえに、いかに早く価値の高い機能をリリースするかがビジネスの成否を分ける状況だったため、そのなかで試行錯誤した結果、自ずと開発プロセスはアジャイル化されていったのです。リリースサイクルも常に一定というわけではありませんでしたし、統合されたバックログもありませんでした。形式化されたミーティングもなく、しかし開発エンジニアはフロアのあらゆるところで議論を行い、会話も多く、まさしく大きなチームでした。

６００人いる社員の平均年齢が２８歳という急拡大中の組織は、人事制度も細部は貧弱で、ルールの多くは社員の善意に委ねられていました。就業時間もフルフレックスで、何時に来て何時に帰っても会社は基本的には干渉してきませんでした。社長は「開発者の天国を作る」と公言し、今思うとまさに『興奮するフィールド』でした。ＤＢのテーブル数も桁違いで、簡単なクエリですら下手をするとすぐに返ってこない、そんな巨大なシステムでありながらちゃんとした設計書もない。ひらすらソースコードを読み解きながらコードを書いていく。１０万行を越えるプログラムコードを夜中の静かな時間を見計らって必死に読み解く、そんな日々でした。開発エンジニアの誰もが優秀で、法制度にも詳しく、人事給与業務のニーズも完全に把握していました。大企業の給与制度は複雑で、とても多くの特殊なパターンがあり、それをシステムでどのように処理すればいいのか、を考えながら実装していきました。「ブレークスルーを起こせ」。そのようにイノベーションを起こすことを求められていました。そして挑戦による失敗は許容されていました。

私は途中で、社長直下の採用部門に異動になります。海外採用を社長に提案し、予算を与えてもらい、海外のエンジニアを採用する裁量を与えられたことで、何年もの間、インド、中

国の現地の超優秀な学生たちと接する機会を得ることができました。私がいまだにソフトウェア産業が最高の場だと言い張っているのは、このとき見て感じた世界が根底にあります。私は野球というものでアメリカに渡ることができ、そしてソフトウェア開発によってインド、中国、シンガポールに行くことができました。

私は過去に1度だけ社長を激昂させたことがあります。それは、私の不手際でミスしたことを報告しなかったのです。社長に怒られたのは、ミスをしたことではありませんでした。ミスしたことを報告しなかったこと、保身に走ったことを強く怒られたのです。失敗を許容する文化というのは、謝罪をベースとする文化です。許可をベースとする文化ではありません。とりあえずやってみてダメだったら謝ろう、とする姿勢が基本姿勢です。

とにかく忙しかったですし、誰もがハードワークでした。そして利益率は高く、給与も高かったです。やりがいも実利もありました。そして私は次第に自分自身でも事業を興したくなってきました。社長が作ったこの会社のように自分も何か形のあるものを作り上げたいと思うようになりました。そして独立しました。

事業サービスもソフトウェア製品もすべて自分たちで作り上げました。開発も最初に立ち上げたWEBアプリケーションは私1人で作り上げました。ERPパッケージを大きなチームで開発していた際に学んだことを生かして、今度は大きなチームではなく小さなチームで戦うことを選択しました。その後9つのプロダクトをアジャイルプラクティスで開発し、スタートアップならではのエクイティでの資金調達や、メディア露出なども積極的に行いました。スタートアップというものはアジャイルそのものです。規律と変化を同時に生み出さなければならず、沈んでいく船の修理をしながら航海を続ける冒険の旅なのです。生きて帰ってこれるかもわからないその旅の行く先を、目をこらし疑いながらも、根拠もなく自らを信じ込み、進んでいくしかないのです。

その後は、日本を代表する世界的大企業に対して、アジャイルを教える機会を得ることができました。アジャイル化を果たすためのコンサルティングとコーチングでプロジェクトを支援をする仕事をスタートしました。会社が町一帯というほどの巨大企業。超大企業というものの実態に触れることができ、その中で様々なことを学びました。アジャイルを経験したこ

とも見たこともない人に、アジャイルの本質を理解してもらうのがどれほど難しいのか、を痛感しました。そして、どのようにすれば実践しやすくなるだろうか、アジャイルを削ぎ落としていくなら何を削って何を残せばいいだろうか。毎日そんなことを考えながら複数のプロジェクトを渡り歩いていきました。時には力及ばずで、うまくアジャイル化を果たせなかったプロジェクトもありました。私自身も実践の中で多くの気付きと反省を繰り返してながら、成長していきました。

今思うと私の人生がアジャイルそのものかもしれません。

価値ある人生を送りたいと願って日々を過ごしていますが、このアジャイルという哲学を本に書き起こし世に出すことが私の提供できる価値である、と思い立ち、本書を書き上げました。なるべく特定の技術に偏った内容にならないように心がけ、また、プログラミングに詳しくない人でも読み進められるように、できるだけ汎用的な表現を用いることにしました。ソフトウェアの世界の変化は速く、本書を手に取った皆さんの時代背景と常に一致させることは不可能ですが、それでも不朽の本質なるものを伝え、そしていつの時代でも使えるようにとの思いを込めました。

何度失敗してもいい、そこから学んで次に進み、ついに成果を実らせましょう。
限りある人生の時間の中で、リスクを取り、変化に適応し、圧倒的な速さによって価値を届けましょう。

本書が皆様のプロジェクトの助けになることを祈っております。

2021年2月21日

大垣伸悟

巻末付録：スクラムを用いたプロジェクトでの具体的な流れ

スクラムをベースにして実際に私が行ってきた実例を基に、プロジェクトの開始から継続的な製品のリリースサイクルの継続まで、流れを説明していきたいと思います（1スプリント＝1週間とした場合）。

1　プロダクトオーナーを決める
2　開発をする対象となるビジネスドメインやシステムの価値、獲得できる予算について考える
3　メンバーを集めてチームを作る
4　スクラムマスターを決める・招聘する
5　SPRINT#0
　1　インセプションデッキを作る
　　1　ビジョンを作成する
　　2　システムの特徴についてキャッチフレーズを作る

3　トレードオフについて決める
　1　何をやるのか
　2　何をやらないのか
　3　機能／納期／予算／品質のどれを優先するのか
　4　機能／納期／予算／品質のどれを優先しないのか
4　ステークホルダーや関係者を洗い出す
5　インセプションデッキを壁に張り出す
2　ワーキングアグリーメントを作る
3　カスタマージャーニーマップを作る
4　ユーザーストーリーマップを作る
　1　システムに関連してくるユーザーを洗い出す
　2　メインユーザーを決める
　3　メインユーザーのメインシナリオを決める
　4　メインシナリオを分解してエピックストーリーとして横に並べていく

5 エピックストーリーをさらに分解してユーザーストーリーにしてエピックの下段に横に並べていく
6 各ユーザーストーリーに対応する機能を考えられうる全てを書き出し対応するユーザーストーリーの列の下段に並べていく

5 モデリングを行いビジネス環境も含めたシステム全体の繋がりを把握する

6 プロダクトバックログ作成
1 ユーザーストーリーをプロダクトバックログアイテムとして入れる
2 各プロダクトバックログアイテムの価値を算出する
3 各プロダクトバックログアイテムに優先順位をつけて並び替える
4 各ユーザーストーリーに対して開発者が見積りを行う
5 各ユーザーストーリーのROIを算出する
6 ユーザーストーリー以外に必要な作業をプロダクトバックログアイテムにして見積り、プロダクトバックログに入れる

7 ユーザーストーリー以外に必要な作業ばかりが優先されすぎないように気をつけながらプロダクトバックログアイテムに優先順位をつけて並び替える

7 リスク管理表を作成する

1 死亡前死因分析を行う

2 死亡前死因分析からリスクを見つけリスク管理表に入れる

3 各リスクに対する対応策を考えリスク管理表に入れる

4 障害リストを作成する

8 SOX対応について決める

9 環境を構築する

1 アーキテクチャについて議論し決定する

1 プログラミング言語や使用するSaaS・PaaSなどを決める

2 サーバ仕様、ミドルウェアを決める

3 CI、自動テストの運用について決める

4 バージョン管理、チケット管理、ナレッジ共有について決める
5 その他各種ツールについて決める
2 開発環境を用意する
3 本番環境を構築する
4 テスト環境を構築する
5 CI環境を構築する
10 決まったことを随時、ナレッジ共有システムやワーキングアグリーメントに追記していく
6 SPRINT#1〜4
1 DAY1
1 スプリントプランニング
1 プロダクトバックログに入っているプロダクトバックログアイテムの上から順に1スプリントで完成させられそうな分より

も少し多く取り出し、スプリントバックログにスプリントバックログアイテムとして入れる。

2 スプリントバックログアイテムに対して「完了の定義」を決めてプロダクトオーナーと合意する

3 「完了の定義」を基にスプリントバックログアイテムを分割をしたり、重複しているものを一つにまとめる

4 分割したスプリントバックログアイテムを開発者で見積もる

5 プロダクトオーナーがスプリントバックログアイテムのROIを算出する

6 スプリントバックログアイテムを分割する中で、新たに追加で何らかの作業が必要になる場合はそれもスプリントバックログアイテムとして追加して『完了の定義』を決めてプロダクトオーナーと合意する

7 情報が不足していてスプリントバックログアイテムが見積もれない場合、調査するためにスパイクを追加する

2　スクラムボードの作成
　1　スクラムボードにスプリントバックログアイテムを張り出してチームの仕事を可視化する
　2　「完了の定義」をDONEのレーンに張り出しておく
　3　バーンダウンチャートに日付を書く
3　開発を開始する
　1　取り掛かるスプリントバックログアイテムをTODOからDOINGに動かす
　2　動かしたスプリントバックログアイテムの隣に具体的な作業プロセスを『完了の定義』を確認しながら書き出す
　3　各作業プロセスを誰が行うかを決めて名前を書き込む
　4　スプリントバックログアイテムが『完了の定義』を満たすことが開発者間で確認できたら（＝完成）、プロダクトオーナーにレビューを依頼する

5 プロダクトオーナーが、スプリントバックログアイテムが『完了の定義』を満たすと判断したらDONEのレーンに持っていく

4 1日の終わりにバーンダウンチャートに今日分を書き足す

2 DAY2-4

1 デイリースクラムを行う

2 開発を行う

3 プロダクトオーナーは新しいユーザーストーリーや追加機能について考え、プロダクトバックログに追加し価値を算出する

4 バグやシステム障害が発生したら緊急度に応じて対応を決める

1 緊急度が高く対応がすぐにできるものはすぐに行う

2 緊急度が高く対応がすぐにできないものは対応する日を決めてスプリントバックログに入れる

3 緊急度が低いものはプロダクトバックログに入れる

5 リスク管理表、障害リストの更新

1 チームの生産性を高める上での障害になっているものを障害リストに入れる
2 リスクを発見したらリスク管理表に追加する

6 1日の終わりにバーンダウンチャートに今日分を書き足す

3 DAY5

1 スプリントレビューの準備を行う
2 スプリントレビュー
1 完成したスプリントバックログアイテムについて説明・デモを行って『完了の定義』を満たしていることを確認する
2 完成しなかったスプリントバックログアイテムについて説明する
3 プロダクトオーナー・ユーザーは、完成したスプリントバックログアイテムに点数をつけ、開発者へフィードバックを行う
4 新しく要望が出た場合はプロダクトバックログに追加する

3 リリース準備を行う
4 リリースする（デプロイ）
5 レトロスペクティブ
6 軽微なバグの修正、ユーザーマニュアルの校正、リリースノートの作成、Gitブランチの整理などの残作業を行う

7 SPRINT#5

1 DAY1

1 リリースプランニングを行う
2 スプリントプランニングを行う
3 スクラムボードを作成する
    1 スプリントバックログアイテムと『完了の定義』を張り出す
    2 バーンダウンチャートに日付を書く
4 開発を開始
5 1日の終わりにバーンダウンチャートに今日分を書き足す

2 DAY2〜5：以前のスプリントのDAY2〜5と同じ

8　4の倍数のSPRINT毎にリリースプランニングを行う

参考文献

- エイミー・C・エドモンドソン著「チームが機能するとはどういうことか　「学習力」と「実行力」を高める実践アプローチ」英治出版　２０１４年5月３１日発行
- Mike Cohn著「アジャイルな見積もりと計画づくり　価値あるソフトウェアを育てる概念と技法」マイナビ出版　２００９年1月２８日発行
- ジェフ・サザーランド著「スクラム－ 仕事が４倍速くなる〝世界標準〟のチーム戦術」早川書房　２０１５年６月２５日発行
- アネット・シモンズ著「プロフェッショナルは「ストーリー」で伝える」海と月社　２０１２年１２月３日発行
- ドネラ・H・メドウズ著「世界はシステムで動く　いま起きていることの本質をつかむ考え方」英治出版　２０１５年１月３０日発行
- Gregor Hohpe著「The Software Architect Elevator」O'Reilly ２０２０年４月７日発行

- ヘンリー キムジーハウス,キャレン キムジーハウス,フィル サンダール著「コーチング・バイブル（第3版）本質的な変化を呼び起こすコミュニケーション」 東洋経済新報社 2013年4月1日発行
- Steve McConnell著「More Effective Agile ”ソフトウェアリーダー”になるための28の道標」日経BP 2020年6月5日
- クリス・リチャードソン著「マイクロサービスパターン　実践的システムデザインのためのコード解説」株式会社インプレス　2020年3月21日発行
- ジェームズ・カールバック著「マッピングエクスペリエンス」オライリー・ジャパン 2015年1月25日発行
- ジョセフ・オコナー,アンドレア・ラゲス著「コーチングのすべて その成り立ち・流派・理論から実践の指針まで」英治出版　2017年12月8日発行
- アリ・キエフ著「リスクの心理学 できるトレーダーは、なぜ不確実性に勝てるのか」ダイヤモンド社　2003年3月1日発行
- ヴァーン・ヴァーノン著「実践ドメイン駆動設計」翔泳社　2015年3月16日発行

- ビル・トルバート著「行動探求 個人・チーム・組織の変容をもたらすリーダーシップ」英治出版　２０１６年２月２３日
- Kent Beck著「テスト駆動開発」オーム社　２０１７年１０月２０日発行
- 湊宣明著「実践システム・シンキング　論理思考を超える問題解決のスキル」講談社　２０１６年3月２５日発行
- 田中靖浩著「米軍式　人を動かすマネジメント「先の見えない戦い」を勝ち抜くＤ－ＯＤＡ経営」日本経済新聞出版社　２０１６年5月３０日発行
- レイコフ,G. ヌーニェス,R.E著「数学の認知科学」丸善出版　２０１２年１２月1日発行
- クレーグ・ラーマン,バス・ボッデ著「大規模スクラム Large-Scale Scrum(LeSS)」丸善出版　２０１９年1月３０日発行
- Ester Derby,Diana Larsen著「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　２００７年９月1日発行
- Rachel Davies, Liz Sedley著「アジャイルコーチング」オーム社２０１７年1月２１日発行

- Jonathan Rasmusson著「アジャイルサムライ－達人開発者への道－」オーム社 2011年7月16日発行
- Martin Fowler著「新装版 リファクタリング―既存のコードを安全に改善する―」オーム社 2014年7月26日発行
- Andrew Hunt, David Thomas著「達人プログラマー――システム開発の職人から名匠への道」ピアソンエデュケーション 2000年11月1日発行
- エリック・リース著「リーン・スタートアップ」日経BP 2012年4月12日発行
- Nicole Forsgren Ph.D, Jez Humble, Gene Kim著「LeanとDevOpsの科学 テクノロジーの戦略的活用が組織変革を加速する」インプレス 2018年11月22日発行
- Jez Humble, David Farley著「継続的デリバリー 信頼できるソフトウェアリリースのためのビルド・テスト・デプロイメントの自動化」KADOKAWA 2017年7月31日発行
- Jennifer Davis著「Effective DevOps ―4本柱による持続可能な組織文化の育て方」オライリージャパン 2018年3月24日発行

- ジーン・キム, ジェズ・ハンブル, パトリック・ボア, ジョン・ウィリス著「The DevOps ハンドブック 理論・原則・実践のすべて」日経BP ２０１７年６月２２日発行
- レン・バス, インゴ・ウェーバー, リーミン・チュー著「DevOps教科書」日経BP ２０１６年６月１５日発行
- Richard Knaster, Dean Leffingwell著「SAFe 4.0のエッセンス―組織一丸となってリーン - アジャイルにプロダクト開発を行うためのフレームワーク」エスアイビーアクセス ２０１８年７月１日発行
- ディーン・レフィングウェル著「アジャイル開発の本質とスケールアップ 変化に強い大規模開発を成功させる14のベストプラクティス」翔泳社 ２０１０年２月１８日発行
- Kenneth Rubin著「エッセンシャル スクラム: アジャイル開発に関わるすべての人のための完全攻略ガイド」翔泳社 ２０１４年７月８日発行
- Scott W. Ambler, Mark Lines著「ディシプリンド・アジャイル・デリバリー エンタープライズ・アジャイル実践ガイド」翔泳社 ２０１３年６月２２日発行
- Dean Leffingwell著「アジャイルソフトウェア要求」翔泳社 ２０１４年２月１１日発行

- ロバート・C・マーチン著「アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技」SBクリエイティブ　２００８年７月１日発行
- Robert C.Martin著「Clean Architecture 達人に学ぶソフトウェアの構造と設計」KADOKAWA ２０１８年７月２２日発行
- マイケル・C・フェザーズ著「レガシーコード改善ガイド」翔泳社 ２００９年７月１４日発行
- Michael Keeling著「Design It! ―プログラマーのためのアーキテクティング入門」オライリージャパン ２０１９年１１日２５日発行
- Martin Kleppmann著「Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems」O'Reilly Media ２０１７年４月１１日発行
- Stephen Denning著「The Age of Agile: How Smart Companies Are Transforming the Way Work Gets Done」AMACOM ２０１８年２月８日発行
- Jeff Sutherland and Scrum Inc.「Scrum@Scale」Scrum Inc. https://www.scrumatscale.com/scrum-at-scale-guide-online/#event-the-scaled-daily-scrum

著者について

大垣伸悟
株式会社ギガスリート代表取締役

十六歳で野球で渡米。当時はインターネットも携帯電話も普及しておらず、聡明期で情報がなかった中での手探りの毎日。プロ/アマを行ったり来たりの選手生活。
引退後に情報システム系の担任教諭に就任。その後、大企業向けERPパッケージソフトウェア開発会社に開発エンジニアとして入社し基幹業務システムを開発。エンジニアの人員拡大を実現するために採用部門に異動。海外採用を立ち上げ北米、中国、インドの現地の大学の高度ソフトウェア人材の現地採用を行う。
独立し、軟式野球リーグG-LEAGUEを設立。自社サービスのシステム構築から管理ソフトウェアまですべてのプロダクト製品を自社で開発する。

現在は、計画駆動型開発（ウォーターフォールモデル）での経験しかない組織の開発プロセスを、大規模開発をするための複数チーム組成のコンサルティングからアジャイル／スクラム開発のコーチングまで支援。高い成果を出す強いチームを作ることを信条としている。最大手自動車メーカー、大手製造業、総合商社、金融業、大手携帯キャリア、ゲームメーカー、ERPパッケージソフトウェエア会社、大手SIerなどへの支援実績あり。

プログラミングは十五才から始め、最初はCOBOL85で書き始める。様々なプログラミング言語を扱った末、現在に至っては自社開発のプロダクトをHaskellで書いている。
趣味はギターと読書。

アジャイル開発についての解説動画「成功するアジャイル開発」をYouTubeにて公開中。

書籍情報

【エッセンシャル版】成功するアジャイル開発

発行日：２０２１年２月２１日

著者：大垣伸悟

発行者：大垣伸悟

発行所：東京都港区六本木２−４−９アソルティ六本木一丁目ビル１０階

株式会社ギガスリート