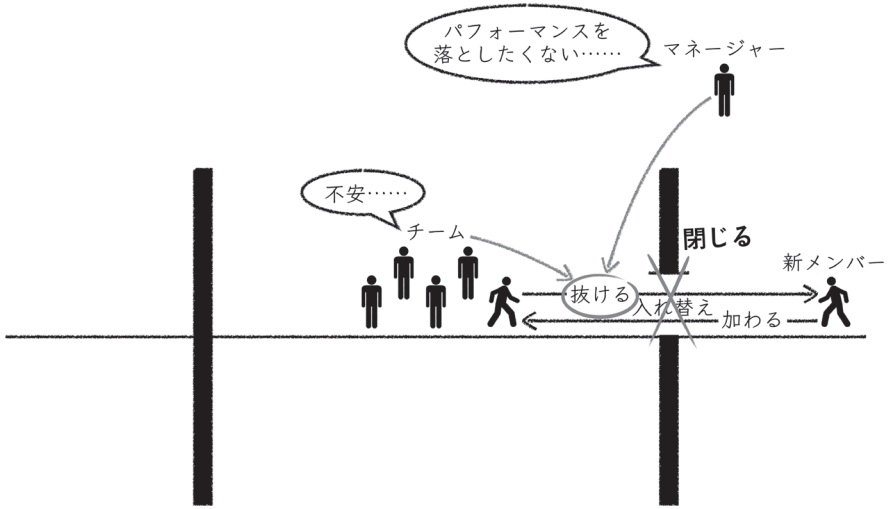
■ チームに変化が加わることを避けたい



しかしこの体制は、時間が経過するに伴って属人化によるリスクが高まります。マネージャーもそれに気づいており、チーム再編の実施も頭をよぎるのですが、なかなか実行には移せません。そうして躊躇しているうちに、チームは何年ものあいだ長く固定されたままになります。

3-3-2 症状 属人化の進行

チームが固定され続けると、属人化が進行します。

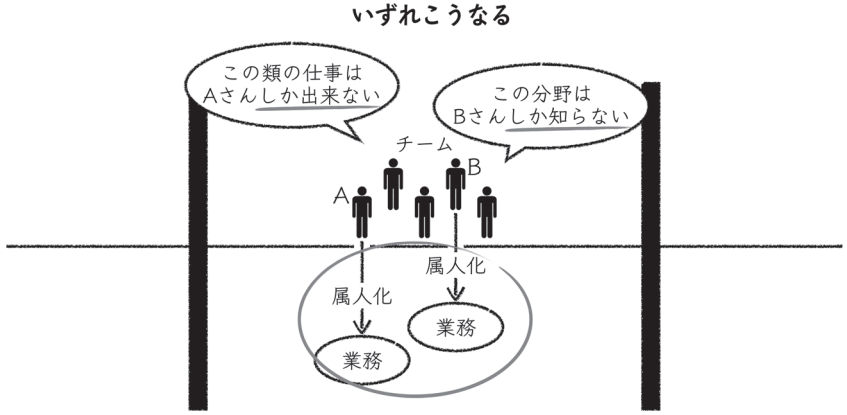
属人化は、メンバーそれぞれに得意分野ができはじめることから生じます。同じチームで長く仕事を続ける中で、みんな、いずれかの分野が得意になっていきます。得意になれば、その分野の仕事は人から頼られるようになるし、自ら引き受けるようになります。得意な人がやった方が短期間で仕事が完了するし、アウトプットの品質も高くなるから当然です。

この症状は、チームを観察することで検知可能です。「この類の仕事はAさんが速い」だとか「この分野ならBさんが詳しい」といった台詞を耳にする機会が増えるからです。そうして、Aさんが素早く対応し、Bさんが知識を駆使して相談事を解決するということが繰り返されます。そこに、他のメンバーが入り込む余地はありません。

■ 固定チームメンバーで仕事を続けると、役割分担まで固定化していく

しかし、この状況が長く続くと、業務の属人化が進み、深刻化します。チーム内の会話に、その変化が現れます。「この類の仕事はAさんしかできない」や、「この分野はBさんしか知らない」と。特定の業務を特定のメンバーに偏って任せ続けた結果、メンバー間での業務に対する知識や能力、経験の差が極端に広がってしまったのです。当人にとっては、1人で業務を担っているのだから、知識や経験を他の人に共有する労力を払う動機がなく、言語化が進みません。自分が理解していれば事足りるからです。こうして属人化が進行するのです。

■ 同じような役割分担が長く続いた結果、業務が属人化する



さがあります。特に、マネージャー自身の上司や利害関係者からの圧力が大きい時に、プロジェクトの実行権限を現場に与えようとしなくなります。現場に任せることを不安に感じ、自らの制御下に置くことで、成功確率を高めようと考えるからです。

マネージャー自身が持つ過去の成功体験による影響もあるでしょう。自ら陣頭指揮をとってプロジェクトを成功に導いた体験が、権限の掌握を好む要因となっているのです。

3-8-2 症状 現場の意見を軽視する意思決定

この組織では、エンジニアリング現場の意見は軽視され、多くのことが、マネージャーの意思に基づいて決定されます。その意思決定の多くは、ビジネス視点によるもので、エンジニアリング課題への考慮はほとんどありません。エンジニアが訴える課題は過小評価されているか、その取り組みはビジネスの足を引っ張る対象とみなされます。

それは、ここでのマネージャーの多くが、ビジネス系の職種でキャリアを築いてきた人たちだからです。ビジネスについてはよく理解しているのですが、ソフトウェアエンジニアリングについてはビジネスほど深く理解していません。そのために、専門領域であるビジネスに偏った意思決定が多くなります。エンジニアリングマネージャーは例外ですが、彼らも上司や利害関係者からの強い圧力によって、意思決定がビジネスに偏りがちになります。

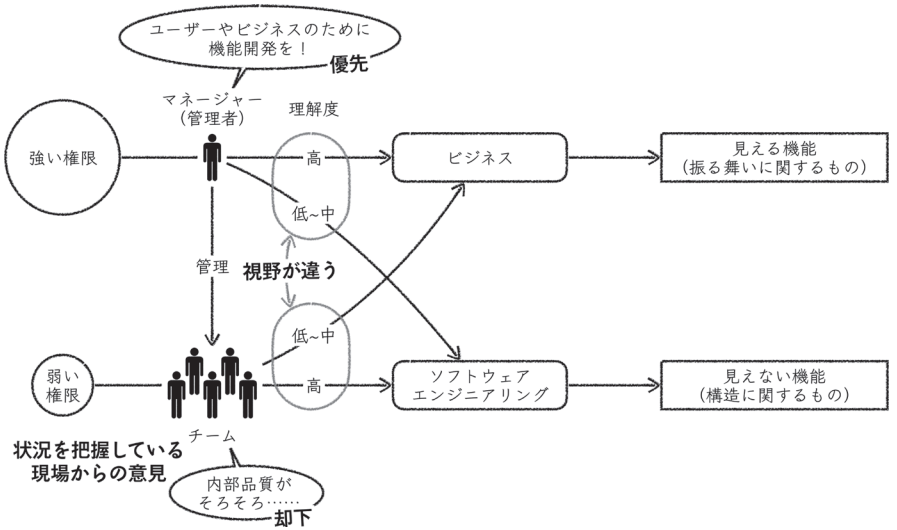
一方で、チームメンバーらは、もちろんソフトウェアエンジニアリングについてよく理解しています。しかし、ビジネスについては興味がないメンバーも多く、彼らの意見はどうしてもビジネス視点に欠けてしまいます[注28]。

同じプロダクトを扱っていても、マネージャーとチームでは、視点が異なっているのです。

両方の意見をうまくバランスさせれば、より優れた意思決定ができそうですが、なかなかそううまくはいきません。マネージャーの権限が強いために、ビジネス視点に偏った意思決定が繰り返されてしまいます。そうしてビジネスのために短期間で大量の機能開発を繰り返します。ソフトウェアの内部品質に大きな問題を抱えることになっても、チームはその改善に取り組むことができません。

注28 「コード品質を高めなければならない」とエンジニアが課題を訴えても、ビジネス視点ではそれがどんな問題を生じさせるのかわかりません。課題や問題の目線合わせができていないのです

■ 内部品質に問題を抱えていても、改善に取り組むことができない



はじめのうちは、チームも様々に意見を出しますが、いずれ口をつぐむようになります。「言ったところで聞いてもらえない」ことを学習するからです。意見に対して繰り返し否定されたり、却下されたり、無視されるうちに、意見すること自体をムダだと感じるようになるのです。いわゆる、学習性無力感です。

そのうえ、チームで決定したことをマネージャーによって後から覆されることもあります。そうするうちにチームは、何でもかんでもマネージャーに許可をもらおうとするようになります。自分たちにその決定権があるかどうかわからないし、あとからマネージャーに覆されるとそこまでの活動がムダになるからです。こうして、自分たちでは何も決められない「無力な他己管理型チーム」ができあがります。チームの口癖は、「上の人たちが」です。「上の人たちが決めてくれないから進められない」というように。

マネージャーは、逆にチームに対して嘆きます。「彼らは自分たちで考えない、動こうとしない」と言うのです。

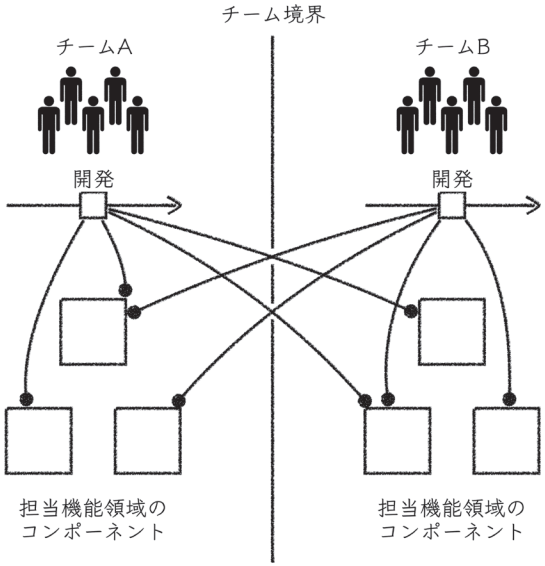
3-8-3 問題点 内部品質の問題が徐々に積み上がることで開発生産性と品質が低下

開発中にどんなに頑張って品質の高いコードを書いたとしても、ソフトウェアの内部は徐々に複雑化し、問題が積み上がっていくものです。その状況が悪化するほど、開発生産性も落ちていきます。だから、時間をとって内部品質の改善に努めるべきだとチームは考えていますが、それが許可されることはありません。

学習性無力感にまで至ったチームであれば、もはや自分たちにできることは何もないと、改善するモチベーションすら失っているかもしれません。そうなれば、内部品質の悪化は

4-5 | ねじれコンウェイ

コミュニケーション構造とシステム構造に乖離がある



システムの構造は、それを設計する組織のコミュニケーションの構造を真似たものになるというのがコンウェイの法則です。しかし、年月とともに両者の構造に乖離が生じてしまうことがあります。そのような状態が「ねじれコンウェイ」です。そこでは所有権をまたいだコードの変更が多発し、コミュニケーションコストの増加やフローの遅延、内部品質の低下に悩まされることになります。

4-5-1 組織設計の概要

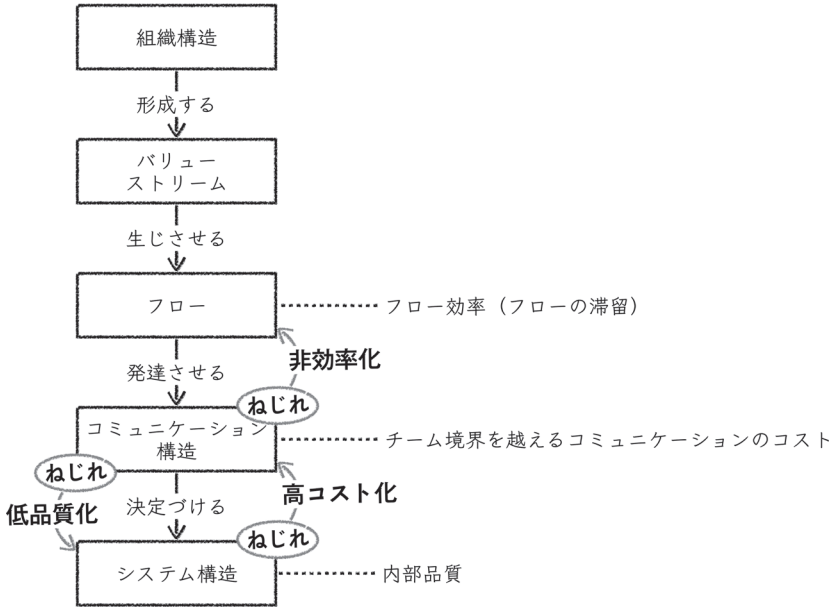
長く開発を続けてきたプロダクトを抱える組織は、自らのコミュニケーションの構造とシステムの構造の間に乖離を生じさせていることがあります。本来、両者はコンウェイの法則によって似た構造になるはずです。それがいつの間にか、そうではなくなってしまったのです。

本書ではこのような状態を「ねじれ」と称します。

組織構造を伝わる順方向の力は、まずバリューストリームを形成し、そこにいくつものフローを生じさせ、プロセス間でのフローユニットのリレーを通してコミュニケーションの構造を築き、最終的にソフトウェアシステムの構造に影響を与えます。

しかし、ねじれが生じると逆方向の力が働き、ソフトウェアシステムの構造が想定外のコミュニケーションパスを作り出すようになります。システムの構造上、どうしても複数のチームで調整しながら開発を進めなければならない箇所ができるからです。こうしてコミュニケーションの構造が複雑になり、それを反映するようにして、フローも複雑になってしまいます。その影響は、コミュニケーションコストの増大やフローの滞留として現れ、時にはソフトウェアの内部品質を悪化させることさえあります。

■ ソフトウェアシステムの構造が組織のコミュニケーション構造と開発フローを複雑にする



この問題は第2章でも扱いましたが、本節ではより詳しく見ていくことになります。

組織が「ねじれコンウェイ」を発症し、自覚症状を持つのは、自社ソフトウェアプロダクトをローンチしてからです。それは数ヵ月後かもしれないし、数年後、あるいは10年以上あとになるかもしれません。いずれにしても、機能追加を繰り返す中で、ソフトウェアシステムの規模もそれなりに大きくなっています。

ビジネスの成長やシステム規模拡大に伴って、開発体制も強化されています。彼らに日々届けられるのは、単一のチームだけでは到底捌ききれないほどの数の開発要求です。その期待に応えるために、組織内には複数のチームが配備され、それぞれが分かれて、日々、開発プロジェクトを進めています。

システムを構成するいくつものコンポーネントは、機能領域ごとにグルーピングされ、オーナーとなる開発チームが決められています。コンポーネントに対する変更はオーナーチームが担いますが、必ずしもそうであるとは限りません。

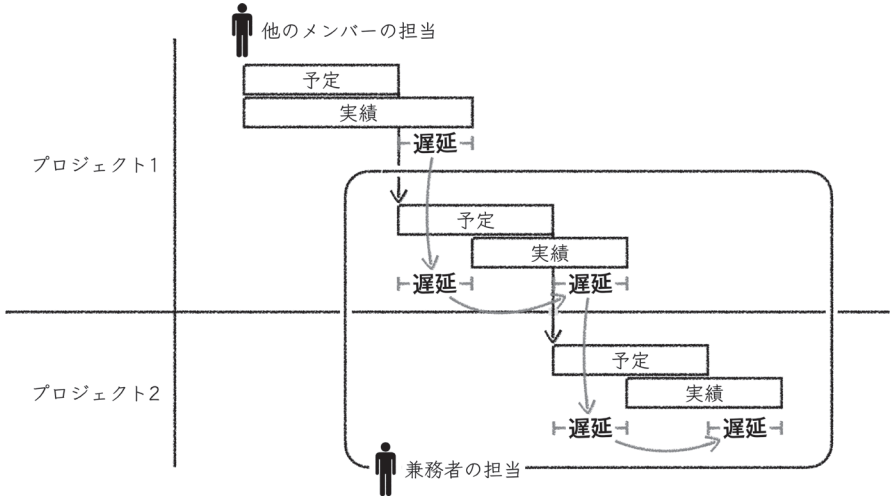
少価値ゆえに調整は難航し、タスク着手時期はプロジェクトが期待するよりはるかに遅くなってしまうからです。

4-6-6 問題点2 プロジェクトをまたいだ遅延の連鎖

兼務者を介して、掛け持ちするプロジェクトが遅延の連鎖を起こすことにも注意が必要です。これには2つのパターンがあります。1つは、兼務者が先行タスクの遅延の影響を受けるパターン。もう1つは、兼務者自身がタスクを遅延させるパターンです。

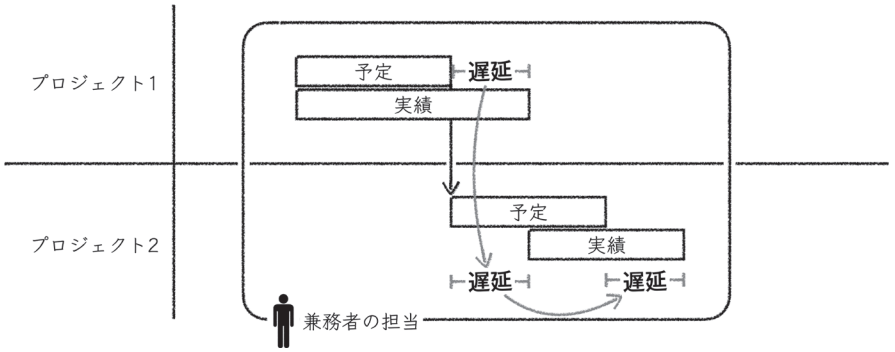
先行タスクが遅延したら、それを待つ兼務者は、予定日に担当タスクを開始できなくなります。その影響で、担当タスクの完了も遅延することになります。そうしている間にも、掛け持ちする別プロジェクトの担当タスクの開始日になってしまい、そちらの開始も遅延してしまいます。

■ 先行タスクの遅延の影響が、兼務者を介して別プロジェクトに飛び火する



兼務者自身が担当タスクを遅らせてしまった時も同様です。遅延を取り戻そうと頑張っている間に、別プロジェクトの担当タスクに取りかかる予定日になってしまうからです。

■ 先行タスクの遅延の影響が、兼務者を介して別プロジェクトに飛び火する



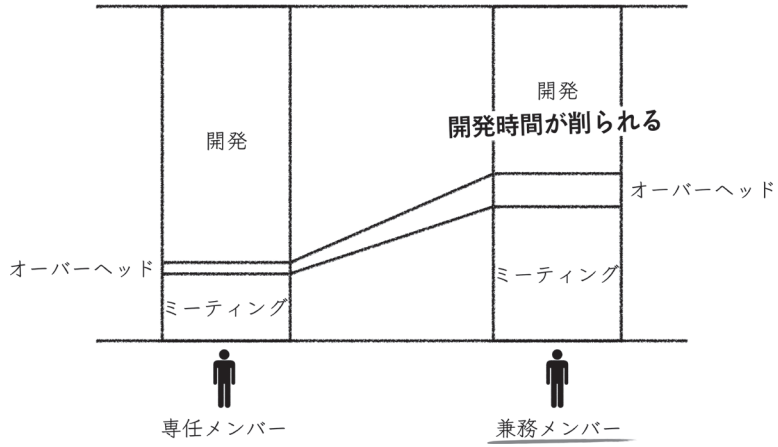
このように、1つのプロジェクトの進捗の遅れが、兼務者を介して複数のプロジェクトに飛び火していくのです。

4-6-7 問題点3 コミュニケーションコストの増大による実務時間減少

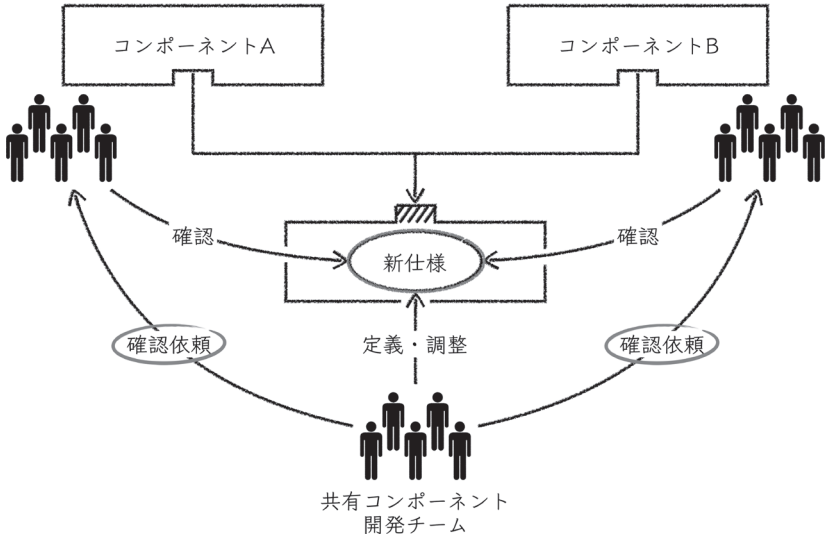
兼務者は、参加するミーティングが増えることで、開発や保守、運用といった実務時間にあてられる時間を奪われます。タスクの進捗が、その分だけ遅くなることは明らかでしょう。

さらに、プログラミングの最中にミーティングが入ることも多くなり、その都度、集中力も奪われてしまいます。本来なら、一度作業に取り掛かれば数時間は続けて取り組むものです。その間、彼らの頭の中では、進行中の作業に関する考えが次々に処理され、構造化されています。しかし、そこに無理な割り込みが入れば、それらの思考が霧散することは避けられません。作業を再開する際は、また時間をかけて一から組み立て直すことになります。こうして、オーバーヘッドによるムダが生じてしまうのです。

■ 兼務によるミーティングの増加によって開発時間が大きく削られる



■ 共有コンポーネント開発チームは常に、仕様の調整にも追われることになる

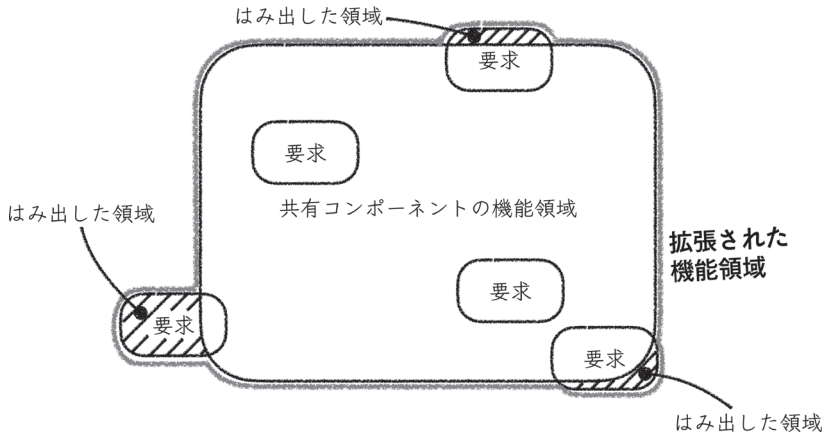


変更の都度、仕様確認するクライアントチームも大変ですが、彼らの意見を聞いて最終的な仕様に落とし込む共有コンポーネント開発チームはもっと大変です。クライアント間で仕様がコンフリクトすることは避けねばなりません。そのような状況下での仕様調整は、センスや技術力、そしてコミュニケーションスキルも要求される、精神のすり減る仕事です。

4-7-4 症状３ 膨らんでいく機能領域

共有コンポーネントは、クライアントからの要求に応えている間に、責務とする機能領域が徐々に広がります。クライアントの要求の中には、そのチームのコンポーネントだけでしか使われないような機能が含まれていることもあります。それを共有コンポーネントに実装してしまうことで、機能領域を隔てる境界線が従来より外側に広がるのです。

■ クライアントチームからの要求に応えている間に、共有コンポーネントが責務とする機能領域が徐々に広がる



だったら、そういった機能には対応しなければそれで済む話だと感じるかもしれませんが、その判断が微妙なのです。捉え方によっては、共有コンポーネントで持つべき機能にも思えるからです。クライアントチーム側も、そう考えたからこそ共有コンポーネントへの要求に含めたのです。

また、ここにチーム目標や評価も絡んできます。共有コンポーネント開発チームの目標には、何らかの形で、クライアントチームへの貢献が含まれていたりもします。それが判断に強く影響し、境界上にある要求機能の取り込みを後押しすることもあるのです。

そうして、共有コンポーネントが責務とする機能領域は、徐々に広がっていきます。

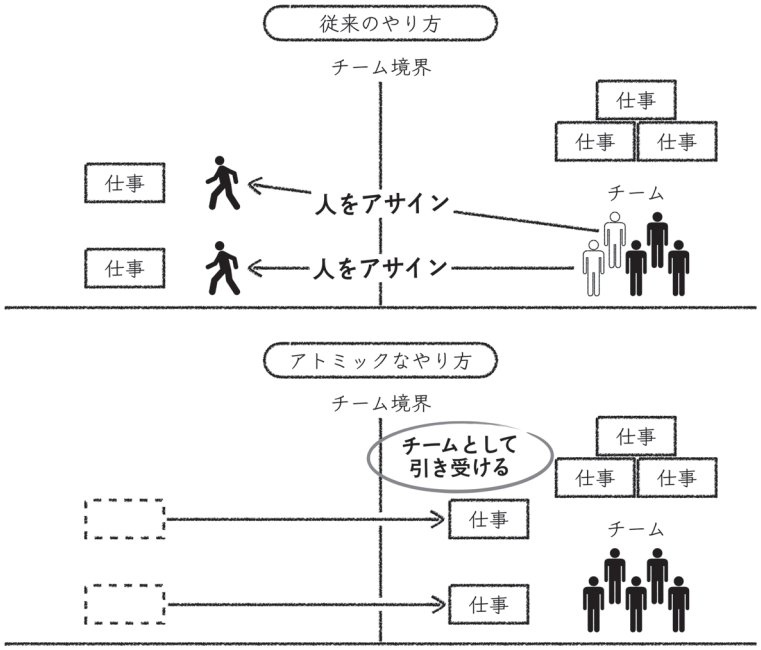
4-7-5 症状４ 仕事に対するモチベーションの低下

共有コンポーネント開発チームのメンバーらは、仕事に対するモチベーションが低くなりやすい点にも注意が必要です。彼らの開発業務のほとんどは、クライアントチームの要求に応えることです。基本的に、受け身の仕事なのです。あとは、保守や運用といった仕事を粛々とこなすだけ。アイデアを出し合ってプロダクトや機能を磨き上げるような、ワクワクする仕事ではありません。

それに加え、彼らは、クライアントチームが配備されているバリューストリーム全体から正式メンバーとして扱われません。あくまでも必要に応じて開発依頼を投げる外部委託先のような存在として扱われます。そのためか、クライアントチームからは、無理な要求を突きつけられることも少なくないのです。そういったことが続けば、ストレスだって溜まります。

チームがそのような状態にあって、よいプロダクトを生み出せるはずもありません。

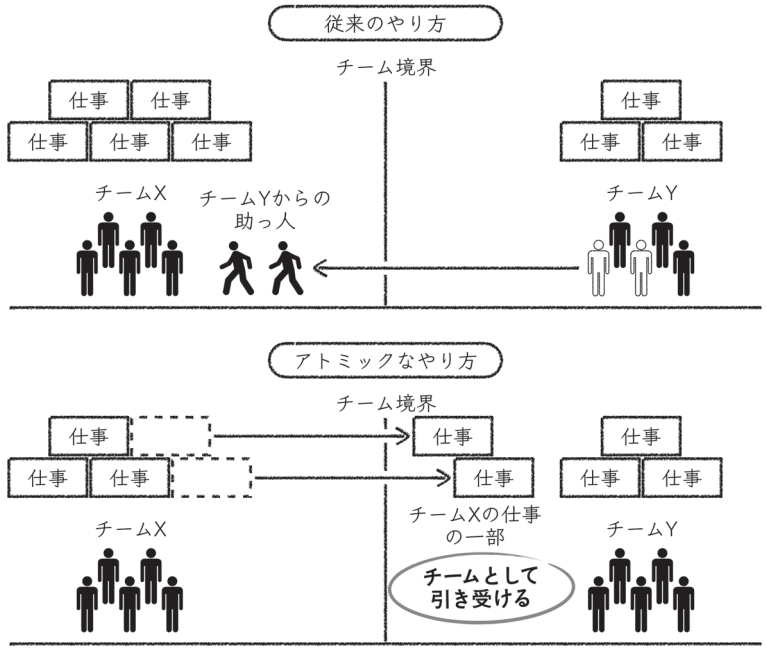
■ アトミック性の高い組織では、業務遂行の単位がチームであることが基本になっている



この特性を重視する組織で活動するチームは、人手が足りない他チームの応援をする時でさえ、アトミックに対応します。従来のやり方ならば、応援を要請しているチーム（X）に対し、別のチーム（Y）がメンバーを貸し出す方法をとることが一般的でしょう。チームYから貸し出されたメンバーが、一時的にチームXのメンバーになるということです。しかし、アトミックなやり方は違います。チームYからメンバーを貸し出すことはしません。人手不足に陥ったチームXの仕事の一部を、チームYが責任を持って引き受けるのです。このやり方は、どちらのチームもメンバー構成が変わらないという点で、チームの安定にもつながります[注7]。

注7　ただし、アトミック性にこだわり過ぎてしまうと、組織の柔軟性が失われることもあります。アトミックにやることが最適であるのかどうか、適宜判断することも重要です

■ アトミック性の高い組織では、チーム間での仕事の応援もアトミックにこなす



チーム単位で業務を遂行するからと言っても、それによって各個人の強みを軽視することはありません。チーム内では、アジャイルソフトウェア開発宣言の中核価値[注8]にもある「個人と対話」を重視します。つまり、それぞれの知識や能力、経験を最大限に活用し、相互作用によってパフォーマンスを高めることを追求するのです。

この原則を適用した「アトミックチーム（Atomic teams）」の実現こそが、本書の組織設計の中核です。

5-2-2 チームの自律

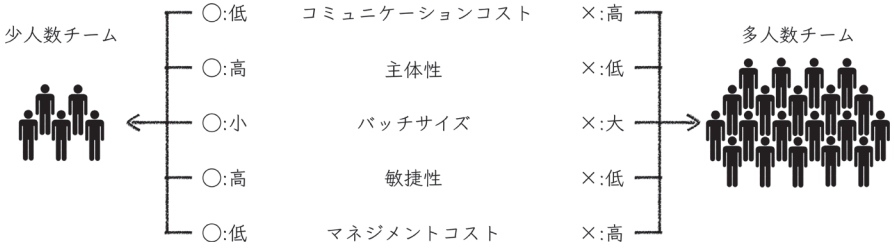
アトミック性を重視する組織では、チームの自律性が育ちやすくなります。

従来のソフトウェアプロジェクトにおける開発業務は、プロジェクトマネージャー（PM）によるオーケストレーション型で進められます。誰が何を、いつ、どのように行うか。それを計画し、実行を指示し、進捗を管理するのはPMです。極端に言えば、「すべてはPMが考え、プロジェクトメンバーはその通りに動く」ということです。プロジェクト成否の大部分は、PMの手腕にかかっています。それが、オーケストレーション型のプロジェクトマネジメントなのです。「無力な他己管理型チーム（3-8）」は、コントローラー気質が非常に強いPMが指揮したプロジェクトのアンチパターンとしてもとらえることができます。

注8　「アジャイルソフトウェア開発宣言 - agilemanifesto.org」
https://agilemanifesto.org/iso/ja/manifesto.html

5-4 | 少人数

チームメンバー数を制限する



チーム編成を少人数とすることで、チーム内でのコミュニケーションコストを下げるとともに、チームメンバーらの主体性を引き出します。

5-4-1 緊密なコミュニケーションの実現

一般的に、チームは少人数がよいとされるその背景は、組織で交わされるコミュニケーションに目を向けることで見えてきます。

自社プロダクトを複数名で開発する人たちは、互いにコミュニケーションを取りながら仕事を進めます。自分1人だけならただ黙々と進めればよいだけですが、2人以上が関わるならコミュニケーションが欠かせません。

その理由の1つは、タスクの多くが互いに従属関係を持つことです。先行タスクが完了しなければ、後続タスクが始められないという関係です。従属関係にあるタスクをそれぞれ別の人が担う時、そこにコミュニケーションが発生します。たとえば、コードレビューを実施するためには、先行タスクであるコード実装によって書かれたコードが必要です。そして、コード実装を終わらせた担当者が、その状況をチームに対して明らかにしなければ、コードレビューは開始されません。そのコミュニケーションには、どのような意図でどんな変更を加えたかについての情報も含まれることになるでしょう。

もう1つ、コミュニケーションは、そこに集まる人々が同じ方向を向いて力を合わせるためにも必要です。これは、「アトミック」原則 (5-2) でも述べました。プロダクトの開発やプロダクトマネジメントには、同じ役割、異なる役割の人たちが様々に集まります。プログラマにデザイナー、テスター、アーキテクト、プロダクトマネージャーあたりはイメージしやすいでしょう。他にも、テクニカルライター、マーケター、営業、カスタマーサポートなど、皆が力を合わせ、共通の目的を果たそうと努めます。

もし、それぞれの目指す先が揃っていなければ、力を合わせることなどできません。そんな状態で優れたプロダクトを生み出すことは困難でしょう。だから、目的を明確にし、それによって協働意思を高めます。コミュニケーションは、その手段としても用いられるのです。

このように、緊密なコミュニケーションは、優れたパフォーマンスや品質をプロダクト開発にもたらすために欠くことのできない存在です。

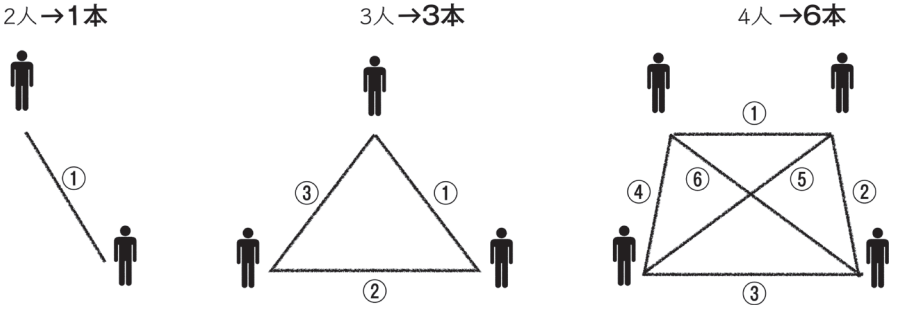
しかし、そこには制約が存在します。1人の人が双方向で定常的にコミュニケーションできるネットワークの大きさの限界です。数名程度なら狙い通りの効果が期待できるでしょう。互いに信頼関係を築き、日々のコミュニケーションを通して、高いパフォーマンスと品質を実現できるはず。では、数十名程度の規模ならどうでしょうか。あるいは数百名なら。人数規模が大きくなるほど、各人が何をやっているかを互いに日々把握し続けることが難しくなっていきます。

たとえば、メンバー全員が毎日同じ時間に集まって開かれるデイリースタンドアップ。メンバーが数名程度のスタートアップ企業や、それに類するチームであれば、短時間で互いの進捗や抱える課題を共有し合うことができます。それが事業の拡大に伴い、人数規模が数十名、数百名に増えたらどうなるでしょうか。全員参加によるスタンドアップでのコミュニケーションは、もはや無謀であることは明白です。順番に情報共有し、全員が発言し終わるのを待つだけで長い時間を要します。そこで聞いた話をすべておぼえることもできません。これではもはや、スタンドアップでのコミュニケーションはコストでしかありません。

このように、双方向コミュニケーションを取るべき対象が増えるほど、そのコストが上がって仕事の効率が落ち、やがて破綻を迎えることになります。

この状況は、しばしばコミュニケーションパスの多さによって説明されます。2人だけならパスはたった1本ですが、3人になると3本、4人なら6本です。さらに、5人なら10本、6人なら15本というように、パスの数は、人数に対して2乗のオーダーで増えます[注16]。このモデルは、コミュニケーションコストを厳密に説明するものではありません。しかし、人数が増えるほどコミュニケーションコストが高くなることを直感的に理解できます。

■ コミュニケーションパスの数は、人数に対して2乗のオーダーで増加する



闇雲に人数を増やし続けると、増員によって得た生産力よりも、コミュニケーションコストによって削られる生産力の方が大きくなるということです。1人あたりの生産力を10だとした場合、2人だと20、3人だと30、4人だと40というように、組織の生産力は線形にの

注16 1人あたりのコミュニケーションパスの数は、メンバー数nから自分1人分を引いた数n-1です。全体のパス数の合計は、n-1をn人分掛けたあとに、重複するパスを考慮して2で割った(n-1)*n/2となります