DevOpsのバイブル ～トヨタ生産方式とLEANスタートアップ

日本アイ・ビー・エム株式会社
GTS、ITSデリバリー
ディスティングイッシュト・エンジニア、技術理事

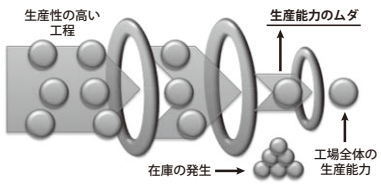
山下 克司 Katsushi Yamashita

【プロフィール】
適用業務パッケージの開発を経てネットワーク分野のテクニカル・リーダーを務める。2007年にディスティングイッシュト・エンジニアの称号を認定され技術理事に就任。2010年から12年まで日本IBMのクラウド・コンピューティング事業の技術統括をするチーフ・テクノロジー・オフィサー（CTO）に就任。現在はデリバリー部門の技術理事。

DevOpsは、クラウド環境を利用してアプリケーションとインフラを同時に継続的に改善し、市場や消費者の要求に高速に対応するプロセスを実行することです。クラウド環境は、これまでの開発だけで行われてきた反復型開発やアジャイル開発手法に、ソフトウェアで制御された柔軟なクラウド・インフラの運用を一体化します。伝統的な職掌に分けられたシステム運用に支えられてきたウォーターフォール型開発は、今日では課題が多く、その開発、運用手法を変革することが求められているのです。

アジャイル開発手法を導入しようとしているプロジェクトは多くありますが、失敗する例も多くあります。原因はさまざまですが、ここでは制約理論（Theory of Constraint）（**図1**）に基づいて説明します。

極端な例ですが、300人で超高速にプログラムを生産することができる環境ができたとしても、たった数人しかいない運用担当者がインフラを設計し物理的なサーバーを準備していたとしたらどうでしょうか。開発サイクルでタイムリーに必要とされるコンピューター資源の準備に数カ月の待ち時間が必要になってしまい、アジャイルの流れは断ち切られてしまいます。そうすると、プロジェクト・マネージャーは資源が必要な時期からさかのぼってスケジュールを作成しなくてはなりません。こうして数カ月前には仕様を確定しておかなくてはプロジェクトが進まず、これではウォーターフォール型開発に逆戻りです。



●工場全体の生産能力は最も低い生産能力に制約される。
http://www.firstedgesolutions.com/systematically-eliminating-manufacturing-bottlenecks/

図1. 制約理論

こうした生産量のギャップを解消するのが制約理論です。開発チームだけが高速にシステム開発をしていても、必要なサーバー調達にいつも膨大な稟議と入札を必要としていては、システムの供給そのものを高速化できないということが、アジャイル開発への取り組みを阻害してきたことを制約理論が示しています。クラウド環境ではサーバーやネットワーク資源を手に入れるのに、数カ月ではなく数分の単位まで高速化が可能になり、開発と運用の生産性のギャップが埋まります。

ドラムロールによる平準化

DevOps環境を構築すると、システムの構築はどのように変化するのでしょうか。ヒントはトヨタ生産方式として有名なLEAN［1］にあります。**図2**に示したのはトヨタ生産方式の14の原則です。これらの原則は4つのP（Philosophy：1 ／ Process：2 ～8 ／ People & Partners: 9 ～11 ／ Problem Solving：12 ～14）にカテゴライズされます。

トヨタ生産方式をITの開発プロセスに適用する場合、重要なキーワードは二つあります。ドラムロールによる生産サイクルの平準化と、「なぜなぜ5回」を繰り返す根本原因の分析です［2］。こうした取り組みを通じて、短期的なコスト削減ではなく長期的な利益を考える、継続的な組織学習へと

〈Philosophy〉
1 短期的なコスト削減ではなく、長期的な利益を考える。

〈Process〉
2 淀みない作業の流れを作り、問題を顕在化させる。
3 次工程に必要なものだけを作る。
4 生産量を平準化して作りすぎない。
5 問題を解決するためにラインを止め、品質を設計に作り込む。
6 標準化活動が自主的で継続的な改善につながる。
7 問題を顕在化させるために目で見る。
8 技術を使うなら実績があり役立つものを使う。

〈People & Partner〉
9 仕事をよく理解し、他人に教えるリーダーを育成する。
10 会社の考え方に従う人とチームを育成する。
11 パートナーやサプライチェーンを尊重し、改善を助ける。

〈Problem Solving〉
12 現地現物を徹底的に理解するように自分の目で確かめる。
13 意思決定は根回しに時間をかけるが、実行は素早く行う。
14 執拗な反省と継続的な改善により学習する組織になる。

●プロセスのムダ取りだけではなく、人材や教育を通じて継続して改善する学習組織を目指す。

図2. トヨタ生産方式の14原則

企業を導くことができるのです。

トヨタ生産方式では、生産に関わるすべての工程が同じペースで進められます。生産ラインにおいて、無駄なく効率良く生産を行うには、最も生産性の低い工程の速度に合わせて全体の生産スピードを調整します。そうすることで工程間在庫や過剰設備を削減することができて、かつ最も高いスループット（生産性）が得られるからです（原則2 ～4を参照）。

例えば、車にタイヤを取り付ける工程が最も生産性が低く、1分に1台分しか作業ができなかったとします。その工場では工場全体の生産能力を最も遅いタイヤ取り付け工程に合わせて毎分1台に調整します。もし、毎分10台分のボンネットを作ることのできるプレス機が導入されていたとしても、それは生産能力の無駄であるとともに9台分の在庫の無駄を発生させていることになります。工場全体には毎分1台のリズ

ムでドラムが打ち鳴らされているように、すべての工程が同じリズムで操業することが最も効率が良く、最大の生産量が得られます。もちろん、工場全体が毎分10台の能力があるにもかかわらず、タイヤ工程だけが毎分1台である場合には、タイヤ取り付け工程の生産性向上に取り組む改善が非常に重要になります。

システム構築の中で、ドラムロールといわれる一つの作業サイクルでは、プロジェクト全員が一つのリリースに向かって作業をします。一つのアイデア（開発テーマ）がコード化されてテスト、計測、学習するサイクルが効率良く繰り返されます。システムの構築、リリースを繰り返す考え方では、新しいプログラムをリリースする際に必要な設計、開発、テストなど多くの工程の要員、機材の容量を、プログラムの生成量（スループット）に合わせて調整することを意味しています。数百人で開発しているのにサーバー構築要員が少ないのでテストが進まないなどの課題を解決できるように、生産能力を調整する必要があるということです。ここでも開発部門と運用部門の融合が必要とされています。

問題を根本から解決する

次に、この生産システムがより堅牢なシステムに成長する仕組みについて説明します。トヨタ生産方式では、もし不具合が発生すると全員が作業を止めて「なぜなぜ5回」という根本原因を追求し改善する活動が行われます。生産工程の一部だけでなく全体を停止することで、全工程を通じて問題の根本原因を解決する「5回のなぜ」を繰り返します。「5回のなぜ」では一つの不具合の原因が問題の発生経緯をさかのぼって解決され、システム全体が障害に強い堅牢な仕組みを手に入れられるようになるまで改善します（**図3**）[3]。

問題が顕在化して不具合が発生していることは「悪」ではありません。問題を顕在化させるシステムをデザインに組み込んでおく、Quality by Designの考え方が非常に重要です。米国GM社の工場がLEANに取り組んだとき、自社の生産ラインが一度も止まらずに問題発生率がゼロであることを誇りにしていた工場長は、問題が発生して生産ラインが停止することを推奨する考え方との間でジレンマを感じたと言います［2]。

システム構築では、問題がプログラムにあったのか、その問題はどうして発生したのか、仕様やインフラの制約など多次元に問題が洗い出されることで、問題が根絶されます。そうすることでこの生産システムは、問題に対して以前よりも少し堅牢になり、次からは少し速くドラムロールを流すことができるようになります。問題の発生と根絶を繰り返すことで生産システムのリリース・スピードをどんどん速くして、最終的には超高速のリリース・システムを構築することがDevOpsの目的です。

リリース・エンジニアリング

DevOps環境において、開発と運用が共通の開発プロセスを共有するために重要なキーワードは、リリース・エンジニアリングです（**図4**）。これまでは独立していた開発ツールと運用ツールを、リリースという視点で統合する管理層を作り出します。IBMが実装する継続的なデリバリーは、開発者に対してリリースに必要な開発環境を継続的に提供することと、自動テストを活用してサービス品質をシミュレーションすることです。同時にサービス品質管理は、品質の管理メトリックスをインフラ運用ツールのインシデント管理と関連付けることで、品質メトリックスとインフラで発生している事象の相関分析を進めます。このようにしてアプリケーションとインフラのメトリックスを同時に改善できるタスク、ワーク・アイテムの管理が実現できます。ここでは、テスト駆動、Failure Based Design、スモークテストなどの品質管理手法が用いられます。

リリース・エンジニアリングを実現するためには、これまでのように“システムそのもの”を設計してきた体制を、“システムを生み出す仕組み”を設計し（工程設計）、サービス品質を継続的に改善する組織に変貌させる必要があります。ここでは継続的に新たなサービスを提供し続けることのできる、変化に強いサービス・システムが求められます。

＊　＊　＊

これまでのようなリスクの高い大規模な開発をウォーターフォール型で実施することは、インフラ面ではアプリケーション・サイロの乱立と複雑化を招きました。また開発面では、大規模開発のリスクを恐れて新しい技術への取り組みが阻害されてきました。一度作ってしまったアプリケーションの仕様を変更しないで何年も利用するのでは、ビジネス上のスピードに追いつけません。リリース・エンジニアリングとは、継続的なデリバリーを通じてこれまでは隔絶していた開発と運用のメトリックスをビジネスの視点で統合したサービス品質の管理を実現することです。

［参考文献］

［1］Eric Ries: STARTUP LESSONS LEARNED, http://www.startuplessonslearned.com/

［2］Jeffry K Liker, ザ・トヨタウェイ,P.289,日経BP社,2004/7/22

［3］Peter Scholtes, The Leader's Handbook: Making Things Happen, Getting Things Done , P415, McGraw-Hill, December 1, 1997

	問題のレベル	対策の対応するレベル	残されるリスク
▼	床が油で汚れている	油を拭いてきれいにする	何度も掃除をしなくてはならない
なぜ？	機械が油を漏らしているから	機械を修理する	何度も修理しなくてはならない
なぜ？	ガスケットが壊れていたから	ガスケットを交換する	何度も交換しなくてはならない
なぜ？	ガスケットの材料が粗悪品だった	ガスケットの材料を変える	他の部位でも粗悪品があるかも
なぜ？	ガスケットの材料費を削減したから	購買の方針を品質重視にする	次の設計で元に戻ってしまうかも
なぜ？	担当者がコスト削減でしか評価されないから	担当者の評価に品質評価を加える	問題が根絶している

●問題の真因を発見するためには、隠れた問題を探る必要がある。

図3. なぜなぜ5回

リリース・エンジニアリング	
●継続的なインテグレーション ●実装用のライブラリー管理	●継続的なデリバリー ●リリースの管理（品質とテスト）
開発ツール	**運用ツール**
●変更管理 ●要求管理 ●ソース・zコード管理 ●品質管理 ●テスト駆動と自動テスト ●サービスのシミュレーション	●構成の自動化 ●トポロジー・プロビジョニング ●インシデント管理 ●品質のモニタリング

●これまで個別に提供されてきた開発ツールと運用ツールを継続的な流れで統合する。

図4. リリース・エンジニアリング