ものづくりとソフトウェア

丸山 宏

20 世紀のソフトウェア開発は，ウォーターフォールだけでなく，アジャイルなども含めて，ものづくりから学んできたことが多い．例えば，アジャイルにおける「リードタイムの短縮」という概念は，実はトヨタ生産方式の精神そのものといえる．一方で，近年のソフトウェアはものづくりに学んだだけでなく，独自の進歩を遂げて，今では DevOps や Software 2.0 を始め，多くのユニークな考え方が提案されている．このような新しい考え方は，ものづくりにも活かせる，価値の高いものである．本稿では，ソフトウェア的なものの考え方が，今後のものづくりにどのように活かせるのか，それに対して本会の会員に代表される，日本のソフトウェアコミュニティがどのように貢献できるのかを議論する．

In early days software development borrowed various ideas, such as the waterfall development, from the traditional manufacturing processes. Even the idea of ”lean”, the heart of the more modern agile process, can be traced back to the Toyota Production System, or TPS. In addition to these ideas borrowed from manufacturing, the software community has invented new ideas on how we develop and maintain software systems, such as DevOps and Software 2.0. In this paper we discuss that now is the time to give these unique ideas back to the manufacturing industry, and the importance of the role of software community, such as JSSST.

1 はじめに

商用の電子計算機が普及し始めた 1960 年代の終わりころ，高性能化する計算機に対してソフトウェア開発が追い付かず，品質や納期の問題が顕在化するという，**ソフトウェア危機**が問題になったことがあった．それがソフトウェア工学が始まるきっかけの１つとなった．

20 世紀におけるソフトウェア工学はものづくりにおける様々な知見を手本にしてきたが，21 世紀に入ってからは，開発と利用を区別しない DevOps の考え方や，探索的にシステム開発を行う Software 2.0 などの，ソフトウェア独自の考え方が現れてきた．ものづくりにおけるソフトウェアの比重が大きくなるにつれ，これらソフトウェア独自の考え方が，ものづくりにも活かせる時代が到来しつつある．

本稿では，まず 2 章において，ウォーターフォール型開発やアジャイル開発など，ソフトウェアがものづくりに学んだやり方について概観する．3 章では，DevOps と Software 2.0 に代表される，ソフトウェア独自の考え方について，ものづくりとの対比において議論する．これらの考え方をものづくりに活かす取り組みについては，4 章で紹介する．その上で，我々ソフトウェアの研究開発に携わる学術コミュニティが，社会に対してどのように責任を果たしていくべきかを 5 章で議論する．

2 ものづくりに学んだソフトウェア

2.1 ウォーターフォール開発

多くの製造業においては，企画・開発・生産のプロセスが明確に分離されている．自動車における典型的な開発プロセスの一例を図 1 に示す．企画・開発・生産の各フェーズにおいて若干のオーバーラップや並行して行われるものもあるが，全体としてウォーター

Manufacturing and Software

Hiroshi Maruyama, Preferred Networks, Inc. / 東京大学 / 花王株式会社, Preferred Networks, Inc. / University of Tokyo / Kao Corp..

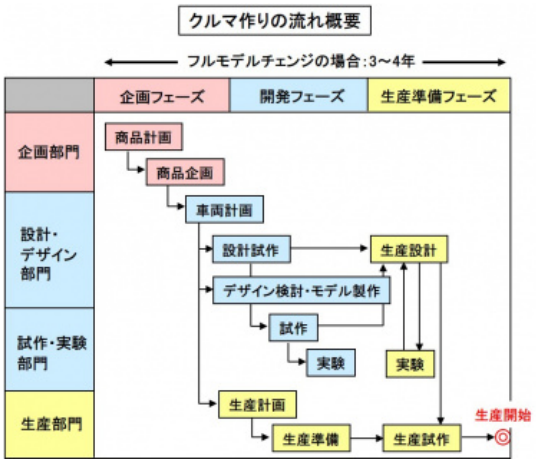

図 1　自動車におけるものづくり流れ（出典：クリッカー 2020 年 11 月号）

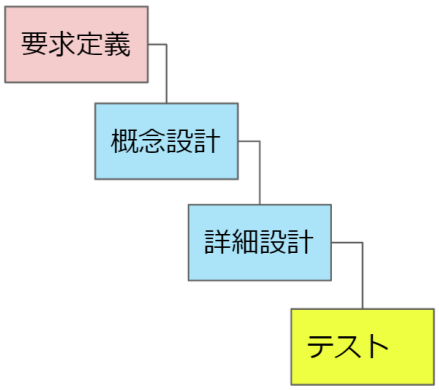

図 2　ウォーターフォール型開発

フォール型のプロセスになっている．これは，いったん開発フェーズに入った製品を企画段階に戻したり，生産フェーズに入った製品の設計を変更したりするという**手戻り**が，非常に難しいからである．

初期のソフトウェア開発も，このようなものづくりの考え方に大きく影響され，前工程への手戻りをできるだけ少なくする，ウォーターフォール型の開発が広く行われてきた（図 2）．

仕様を比較的曖昧なく記述することが可能で，かつその実装方法がよく知られているようなソフトウェア，例えば給与計算や在庫管理のような定形的なシステム（System of Record, SoR と呼ばれる）では，ウォーターフォール型開発はそれなりにうまく行っていた．しかし，仕様が厳密に定義しにくい，あるいは環境変化によって仕様も変化するようなソフトウェア（System of Engagement, SoE と呼ばれる）においては，開発初期における不確実性を事前に見通すことが難しいことから，ウォーターフォール型開発の問題点が指摘されてきた．このようなソフトウェア開発における不確実性には以下のようなものがある．

- 要求定義は顧客の真の期待を表現しているか（各ステークホルダの合意ができているか）
- 開発の各段階の機能は，上位の要求を正しく反映しているか（バグはないか）
- できあがったシステムは非機能要求を満たすか（性能・品質・使いやすさ・セキュリティ・メンテナンスのしやすさなど）
- 開発にかかる工数の見積もりは正しいか（デスマーチに依存していないか）
- システムは事前に想定された通りの環境で運用されるか
- 将来起きる可能性のある新たな要求は何か

2.2　アジャイル開発

ソフトウェア開発における不確実性をうまく扱う方法論として台頭してきたのが，アジャイル開発の考え方である．アジャイル開発においては，上記のような不確実性をできるだけ早く低減するために，要求定義から顧客が実際にソフトウェアを体験できるところまでのリードタイムを短くすることを狙う．このため，開発サイクル（**スプリント**と呼ぶ）を，固定された極めて短時間（通常 1 週間～1 ヶ月程度）で行う．顧客は動いているソフトウェアを見ることで，要求したことと，自分の真の期待とのギャップを知ることができ，また開発チームは，それぞれのスプリントでの開発スピードを計測することにより，その後のスプリントでの工数見積もりをより正確にしていくことができる．

興味深いことに，アジャイル開発の考え方のルーツは，**トヨタ生産システム**（Toyota Production System, TPS）[6] にある．TPS はそもそも多品種少量生産を狙った生産方式であり，そこでの主要な指標は製品の

発注があってから出荷されるまでのリードタイムである．無駄な在庫を持つことなくリードタイムを最小にするために，必要な時に必要な部品を取りに行くための仕組みが必要になる．これが**カンバン方式**である．

アジャイルの本質は開発の方法論というよりも，むしろ組織論であるという捉え方もある[7]．カンバン方式に並び立つ TPS のもう 1 つの主要な概念は「ニンベンの自働化」と言われるものだ． これは，生産工程のどこかで不具合が生じた場合に，その場でラインを止めて問題解決を奨励する考え方である．ラインを止めるということは経済的に大きな損失を招くことではあるが，その判断を現場の作業者に任せているところに大きな特徴がある．現場の判断を尊重し，その結果を振り返ることで知識を共有し，学習する組織になるという考え方は，アジャイル開発にも受け継がれている．

TPS は生産に関する方法論であったが，ソフトウェアのコミュニティはそこから得られた知見をソフトウェア開発に適用することで，アジャイル開発という考え方にたどり着いたのである．

3 独自の進化を遂げたソフトウェア

3.1 DevOps

ものづくりにおいては，製品出荷後に問題が見つかると，製品を回収してそれを修正するには莫大なコストがかかる．このため，出荷の前後の局面を明確に分離し，出荷後には後戻りしないように細心の注意を払うことが求められる．

一方，ソフトウェアの場合には，出荷に相当する概念がデプロイであるが，デプロイ後のソフトウェアを修正することには，それほどコストはかからない．特に，ソフトウェアがインターネット上でサービス（Software as a service, SaaS）として提供されている場合には，ソフトウェアのアップデートはユーザーが気づかないうちに行うことができる．

このことは，ソフトウェアの開発がデプロイ後も継続的に続けられ，開発フェーズと運用フェーズの境目が曖昧になっていることを意味する．開発と運用を境目なく一体化して行うやり方は，DevOps[5] と呼ばれ，ネット上の様々なサービスが DevOps として開発・運用されている．

スムーズな DevOps を行うための様々なテクニックが知られていて，そのいくつかを以下に記す．

- 継続的インテグレーション（CI）　個々の開発者によるソースコードの更新が，自動的にビルドされテストされる仕組み．正しく動く（すべてのテストに通る）コードだけがリポジトリの main ブランチに追加される．
- カナリアリリース　変更されたソフトウェアを，一度にすべてのユーザーに展開するのではなく，一部のユーザーに対してだけ提供し，その結果を観察する方法．ダークローンチとも呼ばれる．
- A/B テスト　ソフトウェアの新しい機能が本当に望ましい結果を得られるのかどうかを，ユーザーをランダムに選んで新機能を使わせその結果を観測することによって決定する方法．
- ソフトウェア定義インフラ　サーバーやストレージ，ネットワークの構成を実際のハードウェア層と切り離してクラウド上で仮想化し，ソフトウェアで制御する方法．
- Game Day　様々な障害に対する耐性を向上させるために，実際にサービスを提供している実環境に対して障害を意図的に注入すること[3]．

3.2 Software 2.0

深層学習が目覚ましい発展を遂げている．2017 年に，テスラ社の人工知能およびオートパイロット部門のディレクタである Andrej Karpathy は，2017 年 11 月に書いたブログの中で，「ニューラルネットワークは新しい識別器というだけではない．われわれがソフトウェアを開発するためのまったく新しい方法なのだ」とし，これを Software 2.0 と呼んだ †1．

深層学習の最も基本的な使い方である教師あり学習は，入出力の例示（訓練データ）を与えて，それをできるだけ精度良く模倣する関数を自動的に得る，というものである．訓練データを仕様と考えれば，Software 2.0 の本質は，アルゴリズムを人手でコーディングするのではなく，仕様を与えてその仕様を満たす実装を

†1　https://medium.com/@karpathy/software-2-0-a64152b37c35

（あるパラメタ空間の中から）機械に自動的に探索させるものだ，とみることができる．

ロボットの制御などで注目されている深層強化学習では，仕様の代わりに効用関数を与えて，その期待値を最大化するようなパラメタを自動的に探索する．この意味で，深層強化学習も探索に基づくプログラミング，すなわち Software 2.0 とみなすことができる．もう少し広く言えば，ベイズ最適化や進化計算などの**ブラックボックス最適化**も Software 2.0 の道具立てと考えることができる．

3.2.1 テスト駆動開発

探索に基づくソフトウェア開発という考え方のルーツは，ソフトウェア工学における**テスト駆動開発**（Test-Driven Development, TDD）に見ることができる．TDD では，コードを書く前にテストケースを書くことが求められる．テストケースは最初は失敗するので，そのテストケースを満たす最小のコードを書く．現在のテストケースが全て満たされると，さらにテストケースを追加していく．このようなコーディングをすると，コードが複雑になるので，時々リファクタリングを行う．

テストケースは，この入力に対してこの出力がほしい，という条件を表現しているという意味で，仕様の一部を表していると考えることができる．テストケースのバリエーションが十分になれば，それは仕様を十分によく近似している，とみなせる．

TDD では，コーディングとリファクタリングは人手で行うが，TDD を一歩進めた**自動デバッグ**（Automatic Program Repair）[4] では，テストケースが通るようなコード変更を機械によって自動的に探す試みである．これは，「作るプログラミング」から「探すプログラミング」への方向性だといえよう．

4 ソフトウェアに学ぶものづくり

初期のソフトウェアはものづくりに学んだが，その後 DevOps や Software 2.0 のようなソフトウェア独自のアイディアが出てきた．それらのアイディアはものづくりに活かせるだろうか．

4.1 ものづくりにおける DevOps

様々な製品が高度化，複雑化する中で，製品におけるソフトウェアの比重が高まっている．たとえば，自動車には各種の ECU やカーナビが装備されていて，それらのソフトウェアのサイズは 1 億行にも上る，という報告がある †2．

これらのソフトウェアは，修理工場等で専用の機器を用いてアップデートするのが一般的であるが，テスラなど一部の車種では，通信ネットワークを通して更新することも可能になっている．また，スマートフォンや，IoT 機器においてもネットワークを介してユーザーの介入なしにアップデートを行うことが一般的になってきている．

このように組み込みソフトウェアをアップデートすることによって，ものづくりの世界においても，製品出荷後にその製品の機能を変えたり，性能を向上させたりすることができる．製品出荷後の開発が一般的になれば，ものづくりにおいても DevOps の考え方が普及してくるだろう．開発チームは製品の出荷後も開発を続けることになり，ソフトウェアの DevOps で試みられている様々なテクニック，たとえばカナリアリリースや A/B テストなどの手法が使われるようになるだろう．

さらに，ソフトウェア定義インフラの考え方を導入するために，製品のハードウェア部分のアーキテクチャ設計が変化することが考えられる．現代の航空機はフライ・バイ・ワイヤによって，補助翼，昇降舵，方向舵などを制御している．ハードウェアが簡潔になることに加えて，ソフトウェアの変更によって異なる性能や味付けの航空機とすることができる．このことを敷衍すると，DevOps 時代にものづくりは，個別の機能を持つハードウェアを，ソフトウェアでデカップリングすることによって，出荷後のソフトウェア変更による機能・性能の変更を可能にするようになると考えられる．

†2 経済産業省 自動車新時代戦略会議（第 1 回）配布資料

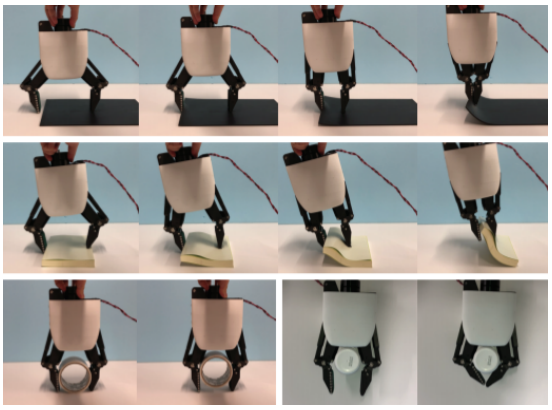

図 3　探索によるロボットハンドの開発 [1]

4.2　探索的ものづくり

従前のものづくりにおいては，企画段階で決められた仕様を満たすため，設計者が自らの経験と知識によって構造や機構を決定していたが，このプロセスを Software 2.0 のように探索に置き換えることは可能だろうか．

探索的なものづくりの萌芽は，自動車産業などで広く使われているソフトウェアによるシミュレーション（Software-in-the-loop Simulation, SILS）に見ることができる．SILS では，実際のハードウェアを作る前に，そこに組み込まれるソフトウェアをテストすることができるので，様々なハードウェア・ソフトウェアの組み合わせを高速に試行錯誤することができる．ソフトウェアにおけるテスト駆動開発の考え方に近い．

SILS があれば，設計パラメタ空間と効用関数を定義することにより，ブラックボックス最適化のテクニックを適用することができる．図 3 は，ロボットハンドの設計の際に物理シミュレータを用い，様々なパラメタをベイズ最適化ツールを用いて最適化したものである．

同様に，素材産業においても探索が使われることが多い．たとえば電気自動車に使われるバッテリの正極，負極および固体電解質の材料として，所望の物理的性質を持つものはなにか，を計算機によって探索する．また，このような材料探索を可能にする商用の汎用原子レベルシミュレータも市場に現れ始めている [3]．

機構の探索においても，材料の探索においても，パラメタ空間が非常に大きくなるので，大規模な計算資源が必要となる．ものづくりにおいては，実験や試作のために様々な設備が必要になるが，探索的ものづくりにおいては，加えて高性能な計算資源を確保することが欠かせないものになるだろう．

4.3　エンジニアリング組織の構築と運営

2.2 節で述べたように，アジャイル開発には組織論の側面がある．アジャイル・マニフェスト [4] に述べられている 4 つの価値基準の 1 つが「プロセスやツールよりも個人と対話」であり，トップダウンの統治によるのではなく，個々のチームメンバーをエンパワーすることによってチーム全体の生産性を高めようとする考え方である．この考え方は，現場の作業者の判断を尊重し，それによって学習する組織になることを目指す TPS にもつながる考え方であり，また近年注目を浴びている新しい組織論である自己管理型組織（TEAL 型組織 [2]）にも共通するものが多い．

ソフトウェア開発の理想形を求めてたどり着いたアジャイルの考え方が，ソフトウェア開発だけでなく，一般の組織全般にも適用可能である，ということは大変に興味深い．筆者はアジャイルリーダーシップの研修を受けたことがあるが，そこでの内容はソフトウェア開発特有のものは皆無で，一般の組織論として成り立つものであった．その意味で，アジャイル型組織の知見と経験を持ったソフトウェア技術者は，ソフトウェア開発以外の世界でも広く活躍できるのではないだろうか．

5　おわりに – ソフトウェア系学会の役割

ソフトウェア的なものの考え方の重要性が増している．それは，ものづくりの様々な段階に活かせるだけではなく，組織や事業のあり方など，より広い分野に広がってきていることを議論してきた．

ソフトウェア系の学術コミュニティは，その知見を

---

†3　https://www.preferred.jp/ja/news/pr20210427/

†4　https://agilemanifesto.org/iso/ja/manifesto.html

ソフトウェアの研究開発に活かすのみならず，ものづくりやひいては社会の課題解決，あるいは社会システムの設計にまで活かしていくための積極的な活動をすべきではないか．

謝辞 本論文での議論は，東京大学工学系研究科 人工物工学研究センターにおけるトヨタ連携講座での議論の中から生まれてきた．トヨタ自動車の皆様，人工物工学研究センターの方々に感謝申し上げる．

参 考 文 献

[1] Ko, T.: A tendon-driven robot gripper with passively switchable underactuated surface and its physics simulation based parameter optimization, *IEEE Robotics and Automation Letters*, Vol. 5, No. 4(2020), pp. 5002–5009.

[2] Laloux, F.: *Reinventing organizations*, Laoux (frederic), 2016.

[3] Robbins, J., Krishnan, K., Allspaw, J., and Limoncelli, T.: Resilience engineering: learning to embrace failure, *Communications of the ACM*, Vol. 55, No. 11(2012), pp. 40–47.

[4] Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T.: Automatic program repair with evolutionary computation, *Communications of the ACM*, Vol. 53, No. 5(2010), pp. 109–116.

[5] パトリック・ボア ジョン・ウィリスジーン・キム: *The DevOps* ハンドブック – 理論・原則・実践のすべて, 日経 BP, 2017.

[6] 大野耐一: トヨタ生産方式――脱規模の経営をめざして, ダイヤモンド社, 1978.

[7] 広木大地: エンジニアリング組織論への招待 不確実性に向き合う思考と組織のリファクタリング, 技術評論社, 2018.