IoT の時代における派生開発の対応

株式会社システムクリエイツ 代表取締役
清水　吉男

世の中は IoT に絡んでビジネスチャンスを掴もうと躍起になっている。だが現実はそんなに甘くはない。これまでの派生開発の中で劣化したソースコードは、今後、IoT の要求に継続的に応えていくには新規開発か大幅な作り変えは避けられないと思われる。だがオフショアなどによる派生開発を長く続けたことの副作用として、組織の開発能力は空洞化しており、とてもすぐに新規開発できる状況ではない。
そこで、派生開発に特化した開発プロセスである「XDDP※1」を活用して、現状を派生開発で凌ぎながら新規開発の準備をする方法について提案する。

1　IoT は時代のうねり

IoT の時代は突然現れたのではない。高速通信の環境が整い、ビッグデータを高速に処理できるクラウドの環境が実用的になった。スマホの裏でクラウドの環境が稼働していることに気づいていない人も多い。

ソフトウェアの開発方法も、2000 年頃から Agile など QCD の同時達成の方法が定着し、それまでの機能の競争から価値の競争に移行してきた。これらの状況の上に、図 1 のように機器とクラウドと IT がつながることで新しい価値を提供しようというアイデアが出るのは自然の流れである。

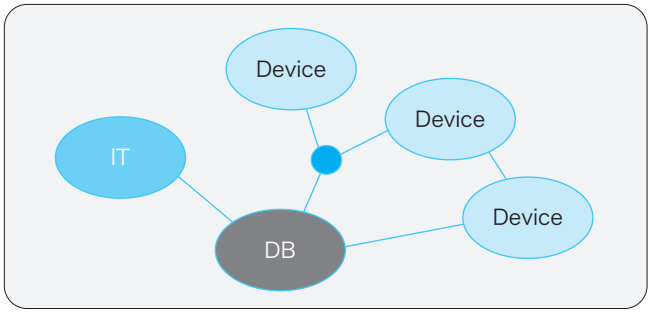

図 1　IoT の構成イメージ

IoT は時代の巨大なうねりである。それを象徴するかのように、エジソンと共に歩んできた GE が、航空機エンジンなどの製造を関連会社に移し、本体を「ソフトウェア会社」に転換させてしまうほどである。「ソフトウェア」によって価値を提供し続けるのである。

果たして、日本の製造業のトップにこれだけの認識ができているだろうか。これまでの日本では、ソフトウェア（この場合プログラムコードを指す）はハードウェアが提供する機能を稼働させるための手段という認識であった。ハードウェアが「主」でソフトウェアが「従」の位置付けである。

1.1　早く買った顧客に後悔させない

だが、IoT の中ではソフトウェアが「主」となり、ソフトウェアの構想を実現するために、ある部分をハードウェアが担当する。もしかすると、既に世の中に出回っているハードウェアから得られる情報を組み合わせて、新しい価値を提供するようなソフトウェアだって出てくるだろう。

ソフトウェアを変更することで提供する価値を変えることもできるし、価値を変化させながら継続的に提供し続けることもできる。それが「Software Defined」である。一部の製品は既にこれを実現しており、自動車だって実現し始めた。これからは「早く買って後悔する」ということはなくなるだろう。

初期の製品には少々使いにくい部分があっても、しばらく待っていれば改良されたソフトェアがダウンロードできる。買い換える必要はない。買い換えてもらえないということは、競争相手にとって厄介である。しかもプラットフォームを無償提供して「デファクトスタンダード化」すれば、世界中の企業や個人がその製品の上で稼働する新しいソフトウェアを開発してくる。

1.2　新しい市場の要求に対応できるか

こうした中では、新規開発時にその後の機能追加や変更を受け入れやすいアーキテクチャを選択することと、多様な要求に応えるためにプログラムコードを劣化させることなく速やかにバージョンアップを提供する仕掛けが求められる。このとき、不適切な変更を加えて混乱させてしまえば他社のソフトェアに乗り換えられてしまう。

【脚注】
※1　XDDP：eXtreme Derivative Development Process の略で、筆者が派生開発に特化した開発したプロセスとして考案したもの。[1]

このように、IoT は新しい時代の到来を実感させる。だが現実問題として、日本の既存の製造業のソフトウェアの開発現場は手放しで歓迎できる状態だろうか。このような要求に対応できるのだろうか。

2 日本の開発現場が犯した選択ミス

この IoT への対応を考えるとき、障害になるのは日本の開発組織に「要求にプロセスで対応する」という発想がないことである。DFD※2 のようなツールを使って、今回のプロジェクトを成功させる合理的なプロセスを設計し、しかも途中で適切に変化させるという発想がないのである。

また、自分たちのプロセスを変化させて、より良い開発手法を取り入れるという発想も見当たらない。その原因を歴史の中で振り返ってみる。

2.1 プロセスで対応することに気づかなかった

1980 年代、市場は「品質」に対する要求を発した。発端は日本の製品である。その中で、モトローラが日本のページャ（ポケベル）の品質に対抗するために編み出したのが「6Σ」である。ISO や CMM の考え方もこの時期に生み出されている。フィリップ・クロスビーの「Quality is Free（邦題：クォリティマネージメント）」には「最初から正しく仕事をすれば高品質は確保できる」とあり、更に、「品質コスト（予防コスト＋評価コスト＋失敗コスト）は、ものごとを正しく行わないことの代償である」と喝破した。

このとき日本では、品質は「テスト工程」で工数を投入して実現しており、結局、「プロセス」で対応するという考え方は受け入れられなかった。この選択ミスが、その後に繰り出される市場の要求に対してことごとく的を外すことになる。

2.2 「自分たちの作業に適合しない」という判断

90 年代に入って納期の短縮要求が出たとき、日本では「人海戦術」で対応した。ちょうど、バブルで新入社員が大量に採用されており、必要な教育もそこそこに一人の担当範囲を狭めて現場に投入した。

このとき、世界では市場の要求を満たすために次々とツールが作られてきた。そこで求められていたのは、自分たちの開発プロセスを変更してツールを効果的に使うことであった。だがこのようなツールも日本ではうまく使えなかった。例えば CASE ツールも、「このツールは我々の開発現場には適合しない」と判断されて導入を見送った組織が多い。

IT 部門でも同じことが起きている。パッケージソフトを導入する際に大幅な「カスタマイズ」を要求した。つまり、自分たちの作業プロセスを変更せずにパッケージソフトを変更させたのである。そこには「プロセスを変化させて納期を短縮する」という発想は見当たらない。

90 年代後半にコストの削減要求が出たときも、単純に「人件費の削減」という発想から、自社のソフトウェア開発部門を子会社化したり、更にオフショア開発へとシフトさせてしまった。「製造」の文化が邪魔をしたと思われる。ソフトウェア開発では、プロセスの工夫と適切なトレーニングによって生産性を何倍も上げることができるが、それを実践した事例がないのだろう。

2.3 オフショア開発の代償

派生開発は新規開発よりもはるかに難しい。その派生開発を不用意にオフショアに出したことでソースコードが激しく劣化し、軽微な変更を依頼したつもりでもテストでバグが多発する。そればかりか、それまで開発要員だった人もオフショア先との中継担当になってしまった。新入社員も実際にソフトウェアを開発することなく年数が過ぎていく。組織によっては 20 年間、まともにソフトウェアを開発していない。当然、そのような組織ではソフトウェアの開発技術力が空洞化しているはずだ。

IT 領域でも、コストの要求から外部のベンダ（SIer と呼ばれている）に丸投げし始めた。その結果、自分たちの業務の要求仕様書も書けないという状態に陥った。更に多段階の下請け構造も助長した。この代償は高くつく。

このように市場の要求への対応のズレは、80 年代の選択ミスに起源を発しており、IoT の時代を前にして大きな障害となる。このことを企業のトップは認識しているだろうか。

ロバート・コール氏は「日本の製造業のトップにソフトウェアがわかる人がいないのはなぜか」と問いかけている。コール氏は同志社大学に席を置いて長く日本の製造業を研究されている。確かに、大手の製造業では「プログラムコード」を開発してきた人は、既に役員クラスにいる。だがその人が「ソフトウェアをわかる」人かどうか。

3 オフショアからの回帰へ

IoT においては、IT を含めたトータルな開発が必要になる。更に、要求に対して機敏な対応が求められることを考えると、今までのようにオフショアで対応することは難しい。相変わらず、IoT だってオフショアでできるだろうと考えている企業のトップがいるとすれば、時代錯誤も甚だしいと言わざるを得ない。

【脚注】

※2 Data Flow Diagram の略。構造化分析で使われるプロセスの表記ツールで、これを作業や開発プロセスの表現に応用する。筆者は、構造化分析を知らない人にも使えるように表記法を改良して「PFD（Process Flow Diagram）」として公表している。筆者の HP（http://homepage3.nifty.com/koha_hp または http://kohablog.cocolog-nifty.com/）の「PFD の書き方」を参照してください。

3.1　開発力の回復に着手すべき

既に述べたように、日本の製造業では派生開発をオフショアで対応してきた組織が少なくない。オフショアに出していない組織でも長く派生開発で回してきた。その結果、IoT向けの機能を組み入れるにも、ソースコードが劣化していて新しい機能の追加や操作を変更する際にバグが吹き出す。また、RTOS すら搭載されていないものも少なくない。

新規開発と派生開発では目的も違うし、実現できることも違う。一言で言えば、新規開発は勝つための仕掛けを組み込む機会であり、そのためのアーキテクチャ設計のような技術が求められるのに対して、派生開発では多様な変更要求に機敏に対応するための技術、あるいはリファクタリングを織り交ぜて劣化を遅らせる技術が求められる。IoT の時代に漕ぎ出すには、新規開発と派生開発の両方の技術を確保することが求められるのである。

4　XDDP でソフトウェアの開発力の回復へ

ソースコードが劣化し、開発技術が空洞化している状況を考えるとすぐには新規開発に取り組めない。現実には、XDDP の持ち味を生かしながら、しばらくは現状のソースコードで派生開発に対応しつつ、並行して新規開発の準備を進めるしかない。

4.1　XDDP の特徴

XDDP は、派生開発専用のプロセスとして開発されたものである。その特徴を幾つかピックアップする。

① 機能追加と変更をそれぞれ異なるプロセスで対応する（図２参照）
一般には、機能追加の中で変更作業をしている。しかしながら、機能追加と変更とでは求められていることが違う。そのため、変更にとっては不適切なプロセスで作業を行っていることになり、そこからバグが発生している。XDDP はこれを根底から改善する。

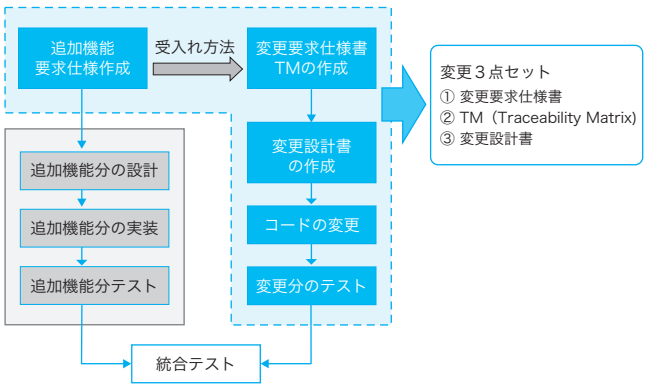

図２　機能追加と変更を別のプロセスで対応する

成果物	カバー範囲	記述の内容（観点が違う）	レビュー機会	
変更要求仕様書	What（Why）	今の振る舞いをどのように変更するか なぜ、変更するのか？ before/after で記述する	○	○
TM	Where	変更する仕様がどこにあるか？	○	
変更計算書	How	具体的な変更方法を記述する 原則として before/after で記述	○	○

・「変更箇所」を「before/after」で表現することで担当者の理解が見え、レビューによって「部分理解」の状態を緩和する
・さらに、「影響箇所」に気づく機会を得てバグも減少する

図３　変更３点セットがレビューを可能にする

② 「変更３点セット」ですべての変更情報を記述する（図３参照）
XDDP では変更を扱う最小限の成果物として、「変更要求仕様書」「TM（Traceability Matrix）」「変更設計書」の３種類を想定している。もちろんこのほかに必要な成果物があれば、プロセスを設計する段階で追加する。追加機能は「追加機能要求仕様書」にまとめるが、この追加機能を受け入れるための変更※3 も変更要求仕様書で扱う。これによって、変更３点セットは今回のすべての変更を扱う文書となる。

③ 変更の成果物は「before / after」を表現する
変更要求仕様書と変更設計書は、変更する個所に対して「before」と「after」の情報を記述することで、担当者が変更個所を理解し認識している様子を表現することになり、レビューの効果を高める。

④ 「変更３点セット」でレビューを可能にする
派生開発では「部分理解※4」の状態で変更作業が強いられるため、思い込みや勘違いが混じる。だからこそ、レビューが必要なのである。XDDP は「変更３点セット」という成果物を、時間差をもって作成する。そのタイミングに合わせてレビューすることで、ソースコードを変更する前に多くのバグを未然に防止する。

XDDP は、部分理解から発する多くの問題を、「変更３点セット」の作成や、それを使ったレビューなど、最小限の秩序を持ち込むことでバグを大幅に減少させる仕組みを提供する。事前にトレーニングなどの適切な準備を伴うことで、工数も従来比で 1/2 ～ 1/5 にもなり、余った時間で新規開発の準備を並行させることができるのである。

【脚注】
※３　一般には、この追加機能を受け入れるための変更がどこにも記述されていないことが多くの問題を生み出している。
※４　新規開発と違って、派生開発の変更の担当者は「全体」を理解できる状況にはないことを表している。

空洞化したソフトウェアの設計技術の多くは、派生開発の中でも機能追加の単位で習得する機会がある。この技術は後の新規開発に役に立つ。また、ソースコードを読んで変更個所や影響個所を調査する際も、調査結果を表現する方法として設計の表現方法を逆に使用する。これも設計技術の習得につながる。こうして、XDDP の取り組みに目的を持ち込むことで、現状の派生開発の品質を確保しながら、並行して新規開発の技術習得を補っていくことができる。

4.2 リファクタリングも XDDP で対応する

一般にリファクタリングは難しいと言われている。それでもソースコードの劣化の状態によっては、今後の変更作業に支障をきたすので「リファクタリング」を行うことになるが、このとき、XDDP が役に立つ。XDDP の枠の中で、今回対応するリファクタリングのパターンを指定し、それを変更要求仕様の形で記述することで、秩序を維持した形でリファクタリングを行うことができる。これは新規開発への時間稼ぎにもなる。

4.3 USDM※5 で要求仕様の作成技術を習得する

派生開発の追加機能に対しては通常の要求仕様書が必要になる。このときの要求仕様書の記述方法を支援するのが「USDM」という表記法である。（変更要求仕様も USDM の表記法を応用している）

以下に USDM の主な特徴を列記する。

① 要求と仕様を階層構造で表現する
② 要求には必ずそれを必要とする理由をつける
③ 機能要求はイベントに始まる一連の振る舞いを一つの要求として扱う
④ 要求に含まれる「動詞」をすべて表現することを目指し、動詞に対して仕様グループを設定する
⑤ 仕様グループに対して一気に仕様化する

派生開発の追加機能に対して短時間でかつ精度の高い要求仕様が書けることは、ベースライン設定後の仕様変更を減らすことになり、更に、リリース後にもこの機能に関して仕様変更が減ることにつながる。もちろん、この技術は新規開発時の備えになる。

4.4 アーキテクチャ設計の準備

派生開発の中では、機能単位の設計技術は習得できるが、アーキテクチャレベルの設計技術は別に習得するしかない。そこで、XDDP で余った工数を充てる。

アーキテクチャの設計は重要である。一般にアーキテクチャスタイルは 5 種類※6 ほど知られているが、自社の製品の特性を考慮しながら、IoT の今後の要求にも対応しやすいアーキテクチャを選択したり、組み合わせたりして試作する必要がある。また、市場の要求も予測しきれないところがあり、新規開発も比較的短い間隔で必要になる可能性がある。そうなると、「部分的」に作り変える要求に対応できるアーキテクチャも有効である。

保守性のような品質は新規開発時の「設計の中で織り込む」必要がある。USDM では保守性のような「作り方の品質要求※7」は作業者への指図として記述する方法を提供しているので、適切に仕様化することで、アーキテクチャと一緒に設計に組み入れることができる。このことは、IoT の時代において有利に作用するはずである。

5 遷宮の発想で両方の開発技術を持つ

派生開発に並行して新規開発の準備を進めていく中で、新規開発のプロジェクトを立ち上げる目処が立ったところで、選ばれたメンバを中心に新規開発チームを作って取りかかることになる。新規開発のメンバは何度かに分けて派生開発のチームから補充していく方法で対応できるだろう。

ソフトウェアシステムも、これからの時代の要求に応え続けるには、派生開発だけで回すことは無理だろう。そのとき、新規開発ができる技術者を確保できなければ、その時点で市場からの撤退を余儀なくされる。だから、しっかりした新規開発の技術と機敏で変更ミスをしない派生開発の技術をバランス良く保有していることが勝ち続ける組織の条件になるだろう。

そこで「遷宮」の考え方が参考になる。日本の神社では、壊れていなくても一定の期間が過ぎれば、その時代の技術を織り交ぜて建て直す。新規に建てる技術を失わないための計らいである。若い建築家は解体する際に外から見えなかった技術を学ぶ。ソフトウェア開発でも、派生開発を知ることで、新規開発の「あるべき姿」が見えてくるし、新規開発を仕掛けることで技術の積み上げを図れる。そのためには派生開発と新規開発の人材を交流させることである。

【脚注】

※5 USDM：Universal Specification Describing Manner の略．要求仕様がモレない表現方法として筆者が開発したものです。［2］
※6 「実践ソフトウェアエンジニアリング」ロジャー・プレスマン、日科技連 第 10 章参照［3］
※7 USDM では、品質要求を「機能を補完する品質要求」と「作り方に関する品質要求」に分けている。

【参考文献】

[1]『派生開発』を成功させるプロセス改善の技術と極意、清水吉男、技術評論社
[2] <改訂第 2 版>要求を仕様化する技術・表現する技術、清水吉男、技術評論社
[3] 実践ソフトウェアエンジニアリング、ロジャー S. プレスマン、日科技連