アジャイル/DevOps 実装とツール(Git,Jenkins,Redmine)

目次

アジャイル/DevOps 実装とツール(Git,Jenkins,Redmine)

はじめに

Agile(アジャイル)開発手法はトヨタ生産方式の Just in Time に触発されて生み出されたと聞いたことがあります(確かに、”Agile Just in Time”でネット検索を行うと沢山の文書が引っ掛かります)。

この経緯から一見アジャイルは日本人に馴染み易い手法と錯覚しそうですが<アジャイルソフトウェア開発宣言>[1]が発表された米国と日本ではソフトウェアに対する感覚が異なります。例えば、アジャイルソフトウェア開発宣言が公表される少し前の時代、顧客に指定された米国製ミドルウェアのマニュアルをみて面食らった経験があります。米国から届いたマニュアルは下記＜例表＞にある実装に関する大量な html で、ロケットの原理を知りたいのに構成部品の説明だけが延々続くようなもどかしさを感じました。日本製品のマニュアルはシステムの対象領域の説明から始まり段階的にかみ砕いて説明していく体裁のものが多いと思いますが、html にはかみ砕いた結果だけが収録されていました。

＜例表＞ 各工程で作る成果物と、◯が成果物を作る工程です

システムの対象領域

対象 / 機能	マシン			利用者	
	認証サーバ	各種サーバ	クライアント	管理者	一般
識別・認証	mod1	mod2	mod3	modA	modB
ファイル共有	mod4	mod5	mod6	modC	modD
アプリ組込み API	mod7	mod8	mod9	modE	modF

分析

設計

実装

マニュアルとしての良し悪しは置いておいて、「製品=実装されたモジュール」ならばマニュアルがモジュールの説明なのは当然です。もう一つ気が付いたことに、厳密に規格化されているという点があります。例えば上の例で modC 固有の処理が 1 命令だけでかつ modC と modD は同時に使われるような場合でも律儀に modC の固有部だけを切り出した実装が行われています。

また、米国発か否かは分かりませんが Java のジェネリクスや TypeScript（JavaScript に型強制を付加した言語）の発明のように型の曖昧さを嫌い、定型的なコードに「オブジェクト・パターン」として名前をつけ、長けた人を「スーパープログラマ」と呼び称賛する文化があります[2]。

このような米国の工業製品的なソフトウェア製造に日本的なチームプレイを加えたのがアジャイルですから、日本では逆に米国の工業品的な製造マインド・技術がアジャイルの実践に必要です。

---

[1] <アジャイルソフトウェア開発宣言>アジャイルソフトウェア開発とその諸原則の定義
https://agilemanifesto.org/iso/ja/principles.html

[2] 一方で、米国では「正常系の機能さえしっかり動作すれば入力エラーや操作ミスで多少おかしな表示が出てもユーザは気にしないよ」と聞きました。また、ソースのコメントが違っていたりすることがソコソコあります...非機能面等、日本とは品質の力の入れどころも異なるようです

アジャイル/DevOps 実装とツール(Git,Jenkins,Redmine)

1. アジャイルと DevOps

そもそもアジャイルって何よ？アジャイル開発手法ってどうやんのよ？
そして、アジャイルという言葉に絡むように出てくる DevOps（でぶおぷす）とは...

1.1. アジャイル

アジャイルソフトウェア開発宣言には以下のように書かれています。
『プロセスやツールよりも個人と対話を、～中略～ 計画に従うことよりも変化への対応を、
～以下略 』
――背後にある原則として、
『要求の変更はたとえ開発の後期であっても歓迎します。
変化を味方につけることによって、お客様の競争力を引き上げます。～中略～
動くソフトウェアを、2-3 週間から 2-3 ヶ月というできるだけ短い時間間隔でリリースします。
～中略～ チームがもっと効率を高めることができるかを定期的に振り返り、それに基づいて自分たちのやり方を最適に調整します。』

上記は 2001 年に有志 17 人により発表されたもので、理念だけが纏められています。具体的な開発手法としてはスクラム[3]等いくつかありますが、共通しているのは短期間（手法により数週間～数日）に区切りチームで小さな開発を繰り返すというものです。

1.2. DevOps

DevOps は「software development (Dev) と IT operations (Ops)の合成語」であるということ以外は普遍的な定義は見当たりません。起源としては、アジャイルと同様にトヨタ生産方式(TPS)[4]にあり[5]、特徴は「継続的なカイゼンとデリバリー」とそれを実現するツールの組み合わせです。

1.3. それ必要なの？

1948 年[6]からトヨタ生産方式の研究が始まり、2001 年にアジャイル、2009 年に DevOps の取り組みが始まっています。それぞれに解決すべき課題があってトヨタ生産方式やアジャイルという方式が生み出されてきました。例えば、以下のような課題や変化への欲求を抱えた現場では継続的なカイゼンとデリバリーが必要になります。

・開発案件と並行して随時 障害対応のパッチを行いたい
・同業他社への優位を保つために、システム改善／機能追加を最低 4 半期に一度行いたい
・半期に 1 度、キャンペーン用のシステム対応がしたい
・数年後に法律改正があり、各サブシステムの対応が必要

---

[3] スクラム https://www.ipa.go.jp/ikc/imanavi.html#section13
[4] トヨタ生産方式 https://global.toyota/jp/company/vision-and-philosophy/production-system/
[5] Analyzing the DNA of DevOps https://opensource.com/article/18/11/analyzing-devops
[6] 年表 https://www.toyota.co.jp/jpn/company/history/75years/common/pdf/production_system.pdf

2. バージョン管理

ソフトウェア製品は機能の追加や UX の向上による製品価値向上のために定期的なバージョンアップが必要になります。そして、出荷日を守るためにいくつかの機能は開発の終盤でリリースを諦める（先延ばしする）場合があります。

平行して、数年後の法律改正の対応や組織変更、即時対応が必要な障害の発生といった事態も発生します。これらのことから複数のバージョンで平行開発ができる仕組みが必要になります[7]。

例えば、新商品のキャンペーンのために以下の 2 つの施策を計画しシステム更改を図る場合

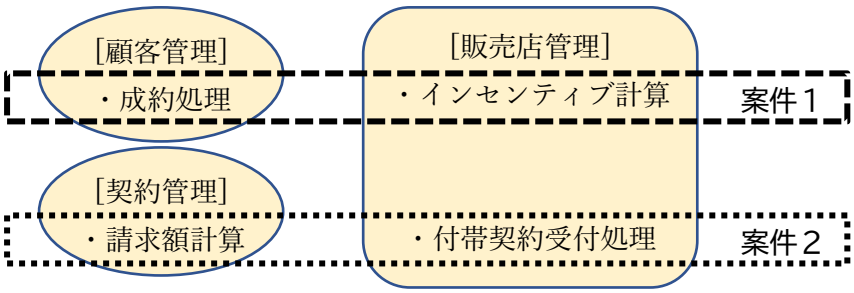
① 期間中契約獲得数に応じて販売店に支払うインセンティブの計算方法を変える【案件 1】
※但し、条件の調整に難航して案件 2 よりも着手が遅れる可能性がある

② 契約毎の付帯契約によって顧客の毎月支払額を割り引く【案件 2】

＜影響箇所とシステム更改＞

[*サブシステム名*]と改修が必要な処理は以下とする。

システム更改の都度サブシステム毎に全 AP を一式揃えたインストール媒体を作成する。



案件 1 と案件 2 はそれぞれ個別に実現できる要件で、状況によりリリース時期が分かれるかもしれない（販売店管理は案件 1、案件 2 の 2 回リリースする）。

2.1. 製造グループ（チーム）

アジャイルの手法はいくつかありますが、いずれにしてもソフトウェアは複数の製造グループが複数のバージョンを整合性を持って平行して保守していける運用体制が必要です。

新商品キャンペーンの例では、開発を担当するチーム編成をサブシステム単位ではなく案件単位にした方が試験迄の行程がスムーズに進みます。

2.2. 影響の局所化

リリース案件毎に更改 AP が異なる場合はリリースする AP の世代に気を付けるだけで済みますが、内容によっては同一の AP が複数の案件の更改対象になります。こうした場合でも、実装を局所化し資材を分離しておくことで影響を切り離すことが可能になります。

---

[7] 複数バージョンの平行開発をアジャイルや DevOps が求めているわけではありません。システムのサイズが大きくて開発案件が集中したときにそのような事態が発生し、アジャイル的な手法はうまく適合できます

2.3. ツール

ある程度規模が大きなシステムでは、リリース案件毎に適切な資材を掻き集めて Just in Time でデリバリするにはツールが必要です。アジャイル／DevOps 手法による開発では以下のようなツールが使われます。

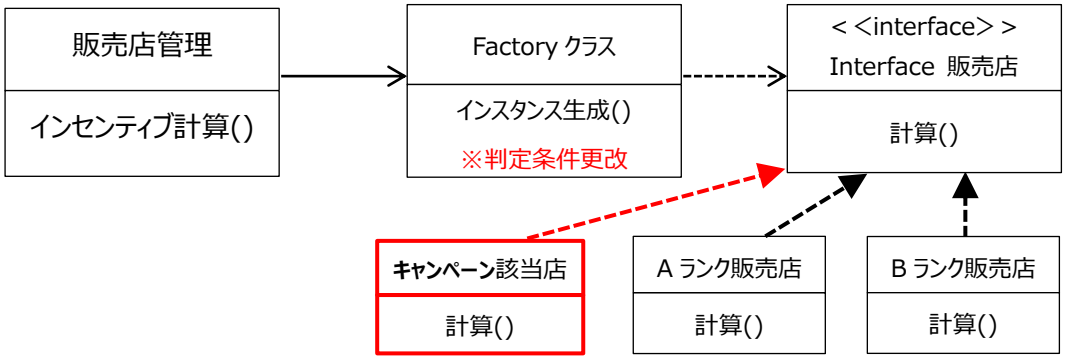
・平行バージョン（ブランチ）管理ができるバージョン管理ツール
・開発資材をビルド・デリバリするツール
・チームの共同作業を補助（仕様管理／スケジュール／作業状況／課題管理...他）するツール

3. 影響を局所化するための実装

開発チームをいくつ用意しても更改対象の資材が被っていては平行開発ができないため、実装段階で資材そのものや構造を細分化しておきます（以下、Java 言語を想定しています)。

(1)更改対象のファイル（クラス）を分ける

更改するクラスが分かれていれば相互に影響する可能性が低くなります[8]。



この例は、Factory パターンを使ってインセンティブの計算方法を振り分けています。こうしておけば、Factory クラス以外の処理に影響することなしに更改できるようになります。

① 販売店管理クラスから Factory クラス:インスタンス生成 に獲得契約数（振分条件）を渡す
② Factory クラスがキャンペーン対象か否かを判定してインスタンス化する
※この判定部分だけは他の案件に影響する可能性があります
③ Factory クラスから販売店管理クラスに②のインスタンスを返す
④ 販売店管理から③で返ってきたインスタンスの計算メソッドを呼び出して計算を行う

```
//インセンティブ計算
long 契約数 = 10000L
private 販売店 店 = RankFactory.create(契約数);
店.計算(契約数);  //額の計算
```

```
public class RankFactory{
    public static 販売店 create(long 契約数){
        if ( 契約数 < 10 ) { return new BRank(); }
        else if ( 契約数 < 100 ) { return new ARank(); }
        else { return new キャンペーン(); }
    }
}
```

[8] クラスが分かれていても強い結合度を持っていると影響が出る可能性があるので、「疎な関係」の適切な設計が前提になります

(2)メソッドを分ける

バージョン毎のソースの更改箇所が異なっていれば、ツールを使って修正内容をマージすることができます。改修要件毎の更改範囲が 1 カ所に集中しないように業務的に意味がある単位でメソッドを切り分ければ衝突の可能性を減らせます。

```
long 手数料率;
long insentive;
if ( 契約数< 10 ) {
   手数料率= 100L;
   insentive = 100L;
   return 契約数* 手数料率+ insentive;
}
else if ( 契約数< 100 ) {
   //RankA店用手数料計算
   手数料率= 100L;
   insentive = 1000L;
   return 契約数* 手数料率+ insentive;
}
else {
   //キャンペーン該当店用手数料計算
   insentive = 10000L;
   return 契約数* insentive; //キャンペーン-乗数計算
}
```

＜平行改修の影響を減らすため…＞
① 分岐制御(ここではif文)中の処理ブロックはメソッド呼び出し式にする
② 同一コードが複数個所に現れたらメソッドに切りだして集約する

```
if ( 契約数< 10 ) { return BRank手数料計算(契約数); }
else if ( 契約数< 100 ) { return ARank手数料計算(契約数); }
else { return キャンペーン手数料計算(契約数); }

/*
 キャンペーン該当店用手数料計算
*/
private long キャンペーン手数料計算(long 契約数){
   long insentive = 10000L;
   return 契約数* insentive; //キャンペーン-乗数計算
}
/*
 RankA店用手数料計算
*/
private long ARank手数料計算(long 契約数){
   long insentive = 1000L;
   return 通常計算(契約数, insentive);
}
/*
 RankB店用手数料計算
*/
private long BRank手数料計算(long 契約数){
   long insentive = 100L;
   return 通常計算(契約数, insentive);
}
/*
 通常店用計算式…契約数×手数料率+インセンティブ
*/
private long 通常計算(long 契約数, long insentive){
   long 手数料率= 100L;
   return 契約数* 手数料率+ insentive;
}
```

(3)分岐処理を減らす・集約する

```
public enum StoreRank{
   S(10000L), //キャンペーン対象店
   A(1000L), //通常Aランク
   B(100L);  //通常Bランク
   long insentive;
   StoreRank(long ins){
      this.insentive = ins;
   }
}

final StoreRank RANK;
if ( 契約数< 10 ) {
   RANK = StoreRank.B;
}
else if ( 契約数< 100 ) {
   RANK = StoreRank.A;
}
else {
   RANK = StoreRank.S;
}
```

(続く)

(続き)

```
long 手数料= 0;
switch(RANK){
   case A:
   case B:
      手数料= 契約数* 手数料率+ RANK.insentive;
      break;
   case S:
      手数料= 契約数* RANK.insentive;
}
```

※同一の条件判定が複数出現する場合や判定結果に関係付く定数（例中ではinsentive）がある場合 enum で条件を表現する

(4)インスタンス変数を作業用に作らない。public にしない

インスタンス変数を作業用に使っていると他のメソッドの作業結果を参照することになるため、他のメソッドの更改内容に影響されます。

① 作業用変数はローカルで宣言し、メソッド間の受け渡しは引数で行う
② 定数は static final を付けてクラス変数として宣言する
③ 外部からインスタンス変数を設定する場合は、コンストラクターを使うかセッターを作る

4. DevOps を実現するツール

アジャイルではコンパクトに素早く開発を進めます。これを継続的に複数のチームで平行して続けるためには整合性を保つための管理・運用ツールが必要です。

4.1. Git（ギット）

Linux の作者が開発したバージョン管理ツールです。GPLv2 ライセンスで公開されているオープンソース[9]で、現状、複数チームでの平行開発はこのツールがないと困難です。

● 以下の url よりダウンロードできます。

https://git-scm.com/

● 日本語によるドキュメントもあります。

https://git-scm.com/book/ja/v2

(1)特徴

プロジェクト毎にリポジトリを作成し、リポジトリの中でバージョンを管理します。

Git の場合はリポジトリ内に平行して存在する分岐（ブランチ）と、各ブランチへの書き戻し（コミット)=リビジョンで管理します。主な特徴／機能は以下があります。

① 全ファイル一括りで版（リビジョン）を管理（スナップショット）

Git 以前のバージョン管理ツール（CVS, Subversion 等）は管理（貸出、更新、保存）をファイル単位で行っていましたが、Git ではリビジョンで全ファイルを圧縮して一括管理しています。

過去のリビジョンの内容もそのコミット時点のスナップショットとして見ることができます。

② リポジトリはリモートとローカルがある

資材の更改は各担当者のローカル環境にリモート（サーバ）にあるリポジトリのクローンを作って行います。クローンはリモートリポジトリのブランチも含めた全資材をローカルにコピーしたもので、ローカルで更新するとローカルだけに新たなリビジョンが作られます。

ローカルで作られたリビジョンはプッシュ（push）しリモートに送ることで同期します。

③ ブランチ（分岐）／マージ（統合）が安全にできる

Git のリビジョンはブランチ毎のスナップショットを世代で管理しています。ブランチは複数作ることができ、相互にマージができます。これにより平行した開発が可能になります。

④ Web 連携情報共有ツール

Git を中核とした Web 上のツールとして、リポジトリの提供やソースレビュー／コメントの追加等の仕組みを提供している GitHub 、GitLab という一部有料のサービスがあります。

(2)注意点等

Git で管理する資材は ローカルの ①作業コピー、②ステージ、③コミットの３つの状態／格納場所と、リモートリポジトリがあります。Git には GUI ツールがいくつかありますが、Git の基本を理解していないと操作できません。いくつかのツールは日本語化も可能ですがメニューの意味が却って解りづらくなっているものもあります（最初はコマンドの方が分かり易い)。

最低限、作業コピーのブランチの切り替え方、ステージへの追加、コミット、リモートへのプッシュ等の基本的な使い方の学習が必要です。また、Windows で開発する場合は資材の文字コードや改行コードの設定、コメントの入れ方等を統一しておく必要があります。

---

[9] https://github.com/git/git#readme

4.2. Jenkins（ジェンキンス）

DevOps と同時に語られる「継続的インテグレーションと継続的デリバリ〔デプロイメント〕(CI/CD)」[10]のためのツールで、MIT License（改変可）で公開されているオープンソース[11]です。

● 以下の url よりダウンロードできます。
https://www.jenkins.io/

● 日本語によるドキュメント（公式の一部を翻訳したもの…若干古い）
https://wiki.jenkins.io/display/JA/Jenkins

(1)特徴

Jenkins はネットワーク上にあるコントローラーからエージェントに指示を出してジョブ（Jenkins に登録したコマンドやシェル）を走らせ、ログの取得や結果の通知が行えます。

コントローラーはブラウザからジョブの登録や結果の確認が可能です。エージェントは n 台のノードと呼ぶマシンに常駐させ、ジョブの実行ノードを選ぶことができます。

その他の主な特徴／機能は以下があります。

① ソースコード管理やビルドツールとの連携

Jenkins にはビルド用ジョブの準備があります。

Git 等のリポジトリと Maven 等のビルドツールをプルダウンメニューから選ぶだけで、ソースコードを取得しビルドするジョブを生成することができます。

② プラグインによる機能拡張

ジョブの実行状況や結果の確認をビジュアルに行ったり、ジョブのスケジューリングを行うための各種のプラグインがあります。

③ 結果の通知や後続ジョブ実行判定

ジョブの実行結果はコントローラーのコンソールから確認する以外にメールで通知したり、他のツールに連絡することができます。簡易なジョブネットを構成することができ、市販製品ほどの機能はありませんがツールとしては市販製品より容易に操作できます。

(2)注意点等

Java11 以降は Java Web Start が使えなくなったので、エージェント側に sshd 等の準備が必要になりました。GUI 向けのプラグイン等多種ありますが、Jenkins のバージョンと合わなかったりバグで動作しない状況もそこそこ発生します。

---

[10] CI はソースから実行形式の作成～試験を継続的（Continuous）に行う、CD は本番環境に実行形式を配備することを継続的に行うというものです。

[11] https://github.com/jenkinsci/jenkins/blob/master/LICENSE.txt

4.3. Selenium

継続的インテグレーション(CI)の概念は試験も含んでいますが、Web アプリケーションの試験を行う場合は、ブラウザを外部から操作するツールが必要になります。ブラウザ操作ツールでは Selenium（セレン[原子番号 34]、セレニウム）が有名で代表的なブラウザは一通り操作できます。ただし、以下の注意が必要です。

① プログラム

ブラウザの機能拡張で操作を記録して再現することも可能（プラグインのインストール要）ですが、試験バリエーションを考えるとプログラミング（Java、C#、Python 他）が必要になります。

② 試験対象のブラウザ

画面表示をしないと動作しないブラウザ（IE 他）があり、他の操作がブラウザに干渉してしまうことがあります（Chromium や Firefox のヘッドレスモードを使えれば画面表示が不要です)。

③ 試験内容

外部の商用サイトではブラウザに表示した内容の保存（スクレイピング）を禁じている場合があります。外部に接続している環境では試験内容に注意が必要です。

④ その他

このツールは、以下の用途にも有効に使えます。

・ネットワーク環境（DB サーバ、AP サーバ、ルータ他）の稼働確認

・改修に伴う想定外の動作（デグレ）が起きてないか、定型的なテストパターンの自動実行

4.4. Redmine（レッドマイン）

Ruby on Rails フレームワークを使用して作成されたプロジェクト管理ツールです。

GPLv2 ライセンスで公開されているオープンソース[12]で、同様のバグ／チケット管理のツール Mantis や Track（いずれもオープンソース）に続いて(ver.1-2010 年)に公開されています。

アジャイルによるプルジェクト管理、例えばスクラムではスタンドアップミーティングでのボトルネックや困りごとの確認、役に立つ情報の交換が重要になりますが、プロジェクト管理ツールを使うと保存したい記録や仕様、メモ等を wiki、フォーラムを使って残すことができます。特に、常に最新の仕様を公開しておくことができるので周知漏れを抑止できます。

(1)特徴

他のツール同様、チケットによるタスク管理やガントチャート、カレンダーによるスケジュール管理、wiki やフォーラム、ファイルアップロード等の情報共有が標準機能でできる他に「付箋」や「かんばん」等の機能拡張を行える各種のプラグインがあります。

(2)注意点等

プラグインには機能や目的が重複しているものも多く、有料のものからフリーの開発が停まってしまっているものまであります。どういった具合にプロジェクトを管理していくかが定まるまでは試行錯誤が必要になります。取り敢えずは基本機能で課題管理ができるようになることが重要になります。

---

[12] https://www.redmine.org/

4.5. アジャイルのドキュメント管理（ツール）

アジャイル開発では“包括的なドキュメントよりも動作するソフトウェア”に価値を認めます[13]。

とはいえ、頭の中にあるイメージでは意思交換ができないので一定のドキュメントは必要になります。

（1） ステークホルダーに開示が必要なドキュメント

① 開発の対象…利用者に向けて何を実現するか（UI、UX）
② 操作方法 …マニュアル

（2） 開発・保守に必要なドキュメント

作った後はソフトウェアの寿命が尽きるまで変更がなければいいですが、そんな保障はありません。実装内容はソースを見れば分りますが、レビューやソフトウェアの改変時に以下の情報が欲しくなります。

㋐ 機能と実装との対比と関連（成果物のリスト、階層図...）
㋑ なぜ、そんな実装になっているのか（経緯、各種調査結果...）
㋒ その実装方法・内容は正しいのか（利用者への説明、他ソフトウェアとの整合、承認...）
㋓ 運用向け（テスト結果、依存ライブラリ、ライセンス／ライセンス期限...）
㋔ その他

（3） ドキュメント管理の方法

ステークホルダー向けのドキュメントは開発の前と後に主に人手で作ることになります。
開発・保守に必要なドキュメントは、実装物やドキュメントやメモの保管庫から生成します。
オープンソースで使えるツールには以下のようなものがあります。

① Redmine の全文検索プラグイン（いくつかあり）
② Fess
Apache ライセンスで提供（フリーソフトなので、無料で利用可能）
URL: https://fess.codelibs.org/ja/

①は Redmine の添付ファイルを対象に全文検索を行えるものがあります。②は独立したサーバで office 製品や PDF、一般ファイル等、各種のデータソースから全文検索ができます。

どちらも問題なく日本語の検索ができます。ただし、②は検索対象の抽出（クロール）が必要で情報の反映がリアルタイムではありません。

上記以外にも全文検索を行えるツールはありますが、上記①、②を含むいずれもセットアップや時折発生するツールのバージョンアップ等、運用に若干手間は掛かります。

[13] アジャイルソフトウェア開発宣言 https://agilemanifesto.org/iso/ja/manifesto.html の一部意訳

5. アジャイル／DevOps でできないこと

アジャイル（“素早い”）手法を適用すると開発期間を短縮することができます。
これには以下の理由があります。

① 仕様決定者（ステークホルダー）の参加場面が増える…仕様決定のスピードアップ

ウォーターフォール開発ではステークホルダーが UI 工程の成果物を承認したらシステムテストまで内容に口をはさむ機会がありません。このためシステムの規模が大きいほど、承認者の組織内の立場が弱いほど仕様が決まらなくなって進捗が滞ります[14]。

アジャイルでは開発粒度を小さくして短期間で回転させます。仕様も詳細に示すことができるうえに、承認の決断もし易くなります。

一番早いのは仕様決定者が仕様を提示することです。

② 開発者間／工程間の引継ぎが不要…ドキュメントが不要

開発チームのメンバーは設計から製造まで一貫して担当します。このため引継ぎ用の設計ドキュメントが不要な上に仕様齟齬がなくなり品質も上がります。実際、設計者が製造を行うようにするとウォーターフォール型の開発でもスピードと品質が上がります。

もし SE/PG 等の役割分担を作るなら、それはたぶんちいさなウォーターフォールです。

以上のように、アジャイルで開発期間を縮めたうえに、あわよくば品質も上げることができますが、前提として以下の存在が必要です。

① 基盤環境（プラットフォーム、ミドルウェア、サービス API）

基盤環境の構築自体も中間成果物の粒度を小さくし、マイルストーンを短くすることでアジャイル的な作業管理が可能です。

② 業務知識と実現技術（を蓄積した要員…または AI？）

アジャイルが生まれた米国では IT 要員の大半がユーザ企業に属している[15]そうです。

日本では PG⇒SE とスキルアップのイメージがありチーム製に適合しづらい仕組みです。

また、システム完工後に開発担当の殆どを解放してしまった企業がシステムトラブルの頻発を受けて開発時の要員を掻き集めに入った[16]という話を聞いたことがあります。ウォーターフォール型の開発でもこのような状態になることを考えると、アジャイル（UX とマニュアルだけを残すような運用）で業務知識の引継ぎをどう実現するか最初によく考えておく必要があります。

以上

---

[14] システム開発の進捗は業務の難易度よりも顧客の参加度合いや意思決定のスピードに左右されます。ウォーターフォールによるプログラム製造（単体試験含む）の全体に占める工数は、業種・業務で異なりますが凡そ全体の 30%～40%割程度です。https://www.ipa.go.jp/files/000004088.pdf

[15] 総務省-日米の ICT 人材の比較
https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/html/nd114140.html

[16] そもそも開発要員が居たらトラブルが起きないということもないような気がしますが…

