Agile and DevOps

アジャイル開発のスピードをビジネスに活かすエンタープライズDevOpsを目指して

平成28年4月7日

戸田　孝一郎
株式会社戦略スタッフ・サービス

Copyrights©2015 SSS Corporation

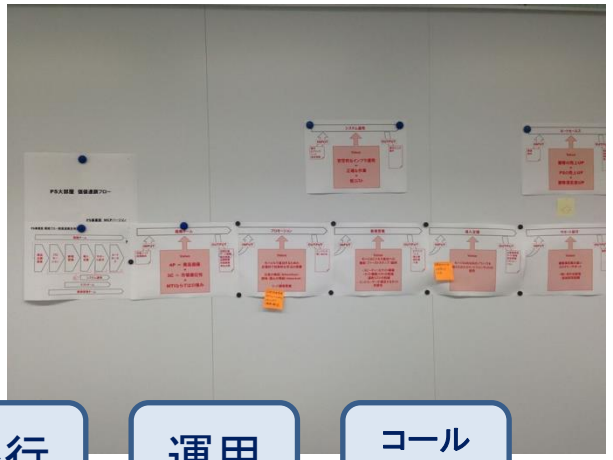
AGENDA

I. プロローグ
II. アジャイル開発の先にDevOpsが見える
III. 運用系でも、アジャイル＆DevOps
IV. DevOps　その目指すモノ
V. DevOpsでシステムを実装する
VI. DevOpsのあるべき姿
VII. パラダイム・シフト

Copyrights©2015 SSS Corporation

Ⅰ．プロローグ

2009年	アジャイル開発を開始
2011年秋	アジャイル開発は成功裏に導入できたが、、、 課題噴出： 開発工程はボトルネックでは無かった。
2012年4月	ビジネスプロセスの全工程の見直しと流水化のプロジェクト開始
2012年10月	DevOpsの導入（全プロセスの見える化、同期化と大部屋化実現）



xx事業部ビジネスプロセス

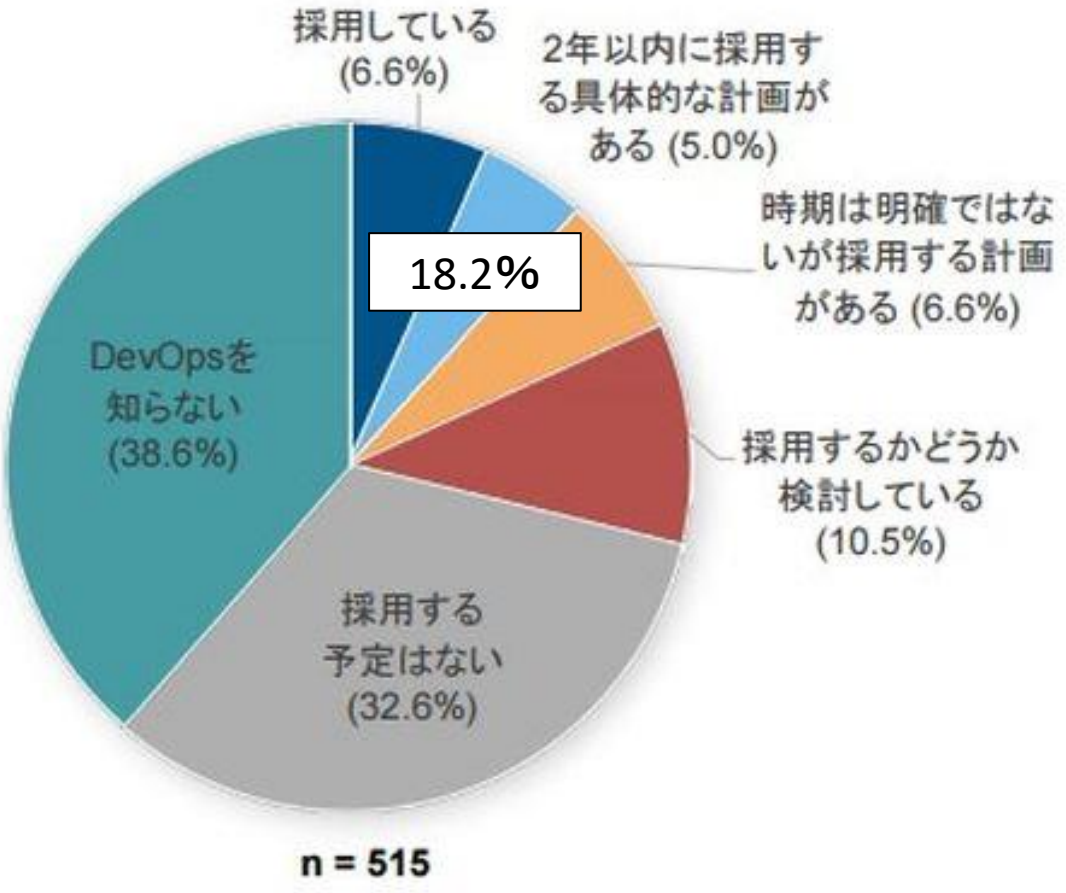
Copyrights ©2015 SSS Corporation

Copyrights©2015 SSS Corporation

94% 事業部門のエグゼクティブの94%が、アプリケーションのリリースをスピードアップしなければならないというプレッシャーに直面していると答えています。

2013年 **66%** 企業の66%がDevOpsの導入を計画

2014年 **88%** 企業の88%がDevOpsの導入を計画　　CA社　グローバル調査



Source: IDC Japan, 1/2016

Copyrights©2015 SSS Corporation

DevOpsとは、

- Development + Operation　の造語
- 2009年に「Velocity 2009」において、flickr のエンジニアにより初めて公の場で用いられた。
  - このプレゼンテーションでは「開発と運用が協力することで、1日に10回以上のペースでリリースが可能になる」という発表とともにDevOpsという単語が用いられた。
- アジャイルとリーンの原則がソフトウエア・サプライチェーン全体に適用される。
- ALM（アプリケーション・ライフサイクル）の視点での管理プロセス
- ビジネス・プロセスとしての側面
- DevOpsとは、人、文化
  - DevOps is not Tool, DevOps is People, People use Tool. (Carnegie Mellon SEI)

JITでのITサービスの提供

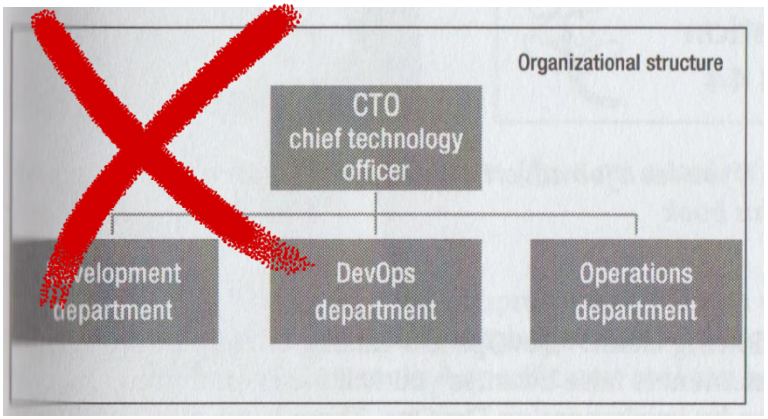
企業におけるITの価値の本質に立ち返る活動

ITの価値の本質　＝　ビジネスをタイムリーに支援する。
ビジネス・スピードを牽引する。
ビジネス領域を拡大させる。
見えなかったモノを見える様にする。

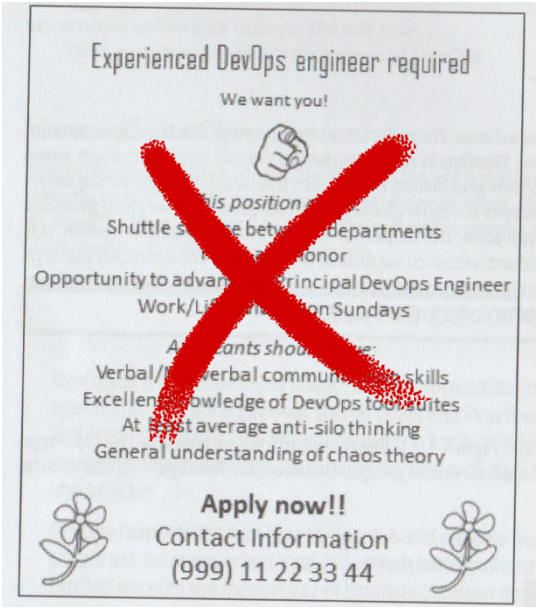
Copyrights©2015 SSS Corporation

Definition of DevOps

The DevOps is not



Organization



Skills

Tools

The DevOps is an Organizational Culture and Behavior of IT Engineers with Agile way.

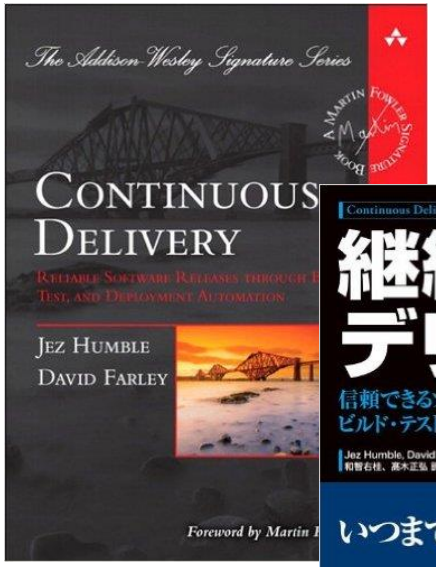
Success of DevOps implementation must evaluate by Business outcomes, NOT only IT Project's scope such as Cost, Speed, and On schedule.

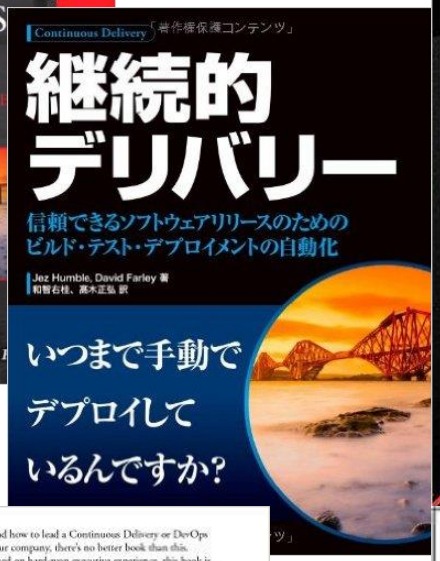
Copyrights©2015 SSS Corporation

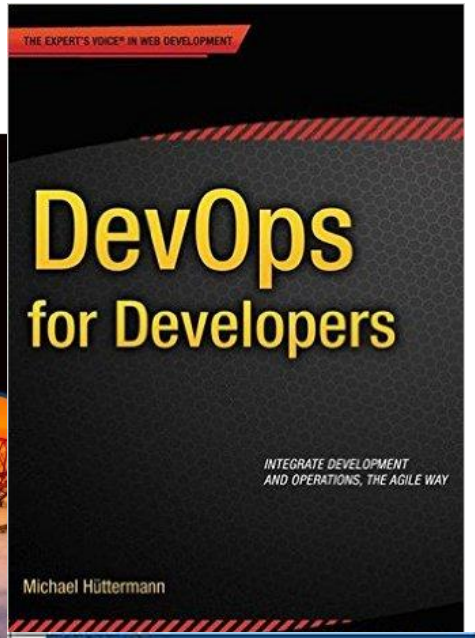
何故、今DevOpsなのか？ （DevOps出現の経緯）

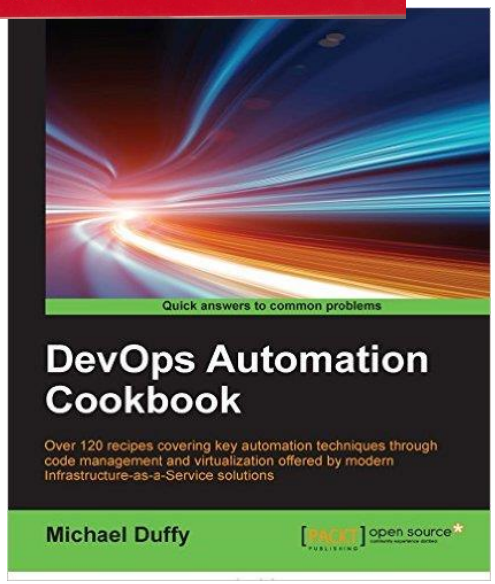
- 開発側からのアプローチ
  - アジャイル開発から進化
  - CI（継続的インテグレーション）からCD（継続的デリバリー）へ、そしてDevOps
- 運用側からのアプローチ
  - ITIL V3で定義（事業継続性；BCP）
  - ISO20000との連携 （開発段階から事前に運用へ引き渡す情報）
- インフラが整ってきた
  - クラウド　 PaaS ＋ Docker（コンテナー技術）
- ビジネス的なニーズ
  - SoE（System of Engagement）の需要の増大に伴い、SoR（System of Record）も頻繁に影響を受ける

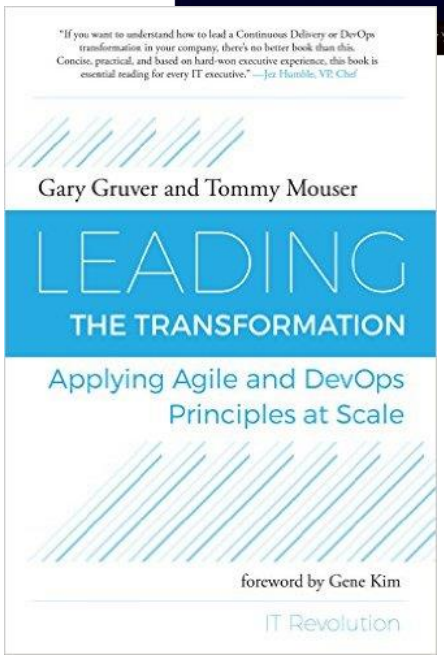
Copyrights©2015 SSS Corporation

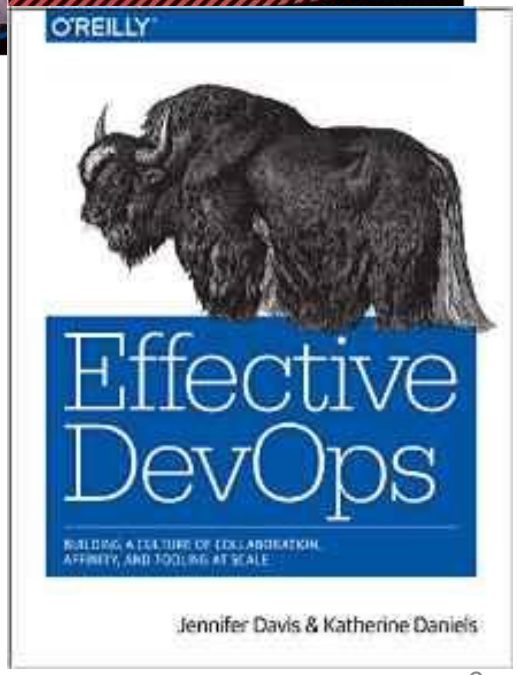












Copyrights©2015 SSS Corporation

Ⅱ．アジャイル開発の先にDevOpsが見える

新しいプラクティス、プロセスの導入

品質の向上、スピードの向上

JITでのリリース（デリバリー＆ディプロイ）

JITでのビジネス支援、全プロセスの見える化、同期化

Copyrights©2015 SSS Corporation

規律あるアジャイル開発

- 稼動するソフトウエアを**より短かい期間を優先**して、数週間から数ヶ月で定期的に提供する。
- ソフトウエアが**正常に機能する**ということが進捗の基本的な評価である。
- アジャイルプロセスは**持続可能な開発**を促進する。スポンサー、開発者、ユーザーは**無期限かつ不断に保守できる**ようにしなければならない。
- 技術的に**優れた良い設計に継続的に配慮する**事は機敏性（アジリティー）を増長させる。
- **簡素が基本**　－やらない仕事をできるだけ多くする
- 最良の構想（アーキテクチャ）、要求仕様、設計は**自己統制された（自律的）チーム**より出現する。
- **定期的にチームは振り返りを行い**、より効果的に出来る方法を思案し、それに基づいてチームの行動に協調と調整が働く。

【アジャイル・マニフェストのアジャイル原則（マニフェストの思想を支える重要な方針）より抜粋】

JKK（自工程完結）での仕事
平準化・流水化
一個流しプロセス
品質への拘り

→ **ジャスト・イン・タイム（JIT）でのデリバリー**

基本的な要件：　タスクの粒度、DoD（完了基準）、継続的インテグレーション
進捗の100%の透明性（見える化）

Copyrights©2015 SSS Corporation

Ⅲ. 運用系でも、アジャイル＆DevOps

ITILの進化

プロセス化

ITIL V1. (1986年～)　：　ITサービスの利用と提供のガイドライン
業務中心

ITIL V2. (2000年～)　：　単体業務からプロセス（プラクティス）で体系化と業務の均一化
①プロセス・アプローチとベスト・プラクティス
②IT技術とビジネス視点で、7つの領域に整理して
サポート（青本）とデリバリー（赤本）を定義

ITIL V3. (2007年～)　：　オープン化への対応とビジネスの継続性（BCP）重視
①ビジネスとITの統合
ビジネス価値を意識
②バリューネットワークの概念を導入
ビジネス視点を原則としALMの実践
DevOps（アジャイル開発）
ⅰ アプリケーション開発（調達）
要件定義、設計、構築
ⅱ サービス・マネジメント
展開、運用、最適化、自動化
ⅲ アプリケーション・ポートフォリオ

バッチ・プロセス
から
一個流しプロセス
へ

③SMO（Service Management Office）← 2011版で追加
プロセスの成熟度を上げる目的
プロセスの構築とプロセスの成熟度を上げる活動を明確に分ける
（CMMIで言うPMOと同様）

Copyrights©2015 SSS Corporation

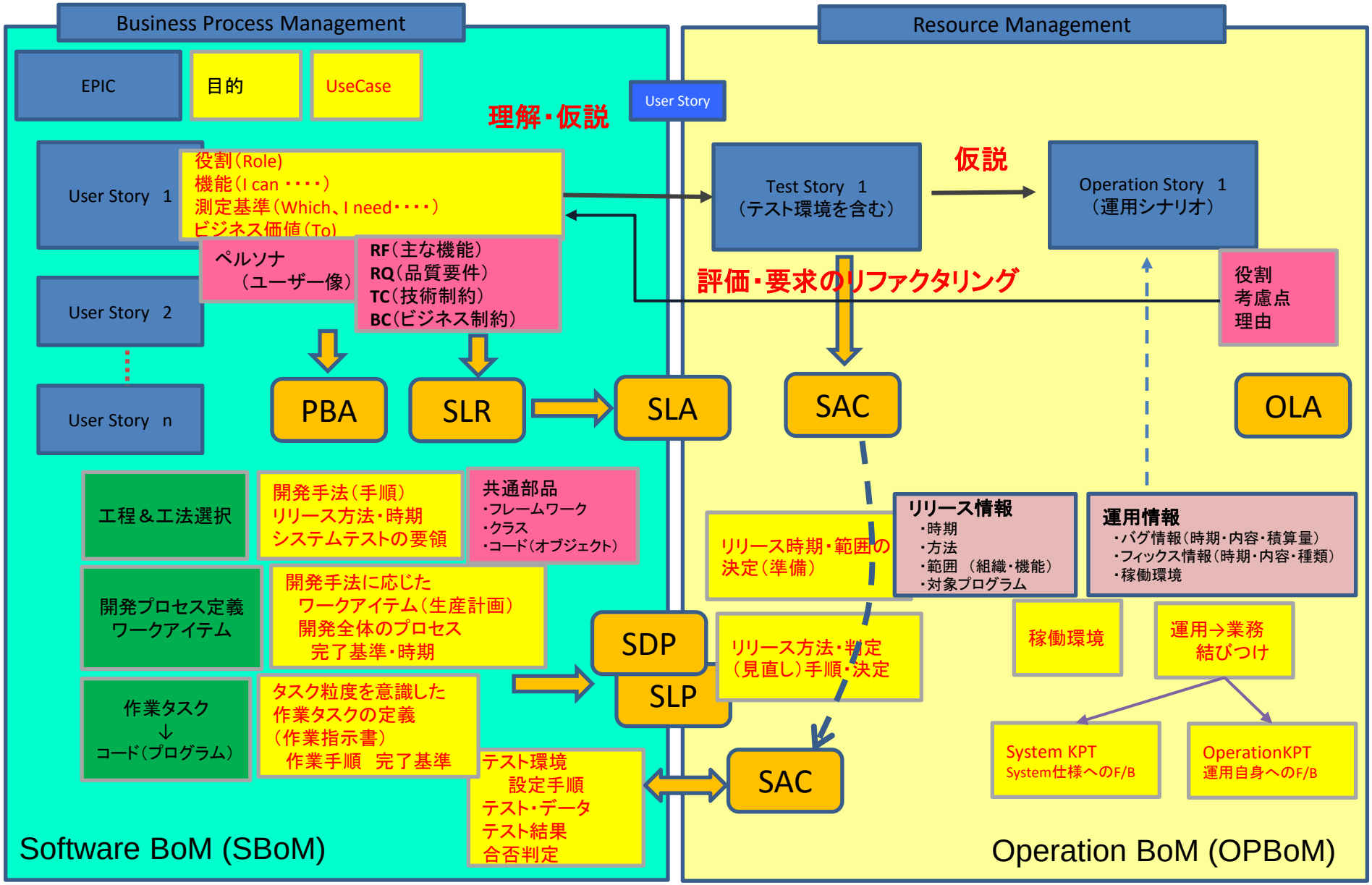
バッチプロセスから一個流しプロセス（アジャイルの流布）への変革に伴い、
運用側として、事前に安定稼働させるための情報が必要になって来た。

計画：	SLR(Service Level Requirements)
要求、設計：	SDP(Service Design Package)，SLA(Service Level Agreement)、UP(User Profile)
開発：	RFC(Request For Change)，SAC(Service Acceptance Criteria)
移行：	RFC(Request For Change)，OLA(Operational Level Agreement)，UP(User Profile)
運用：	PBA(Patterns of Business Activity)

ISO20001で規定されている。
（開発側から運用側へ渡さなければならない情報）

Copyrights©2015 SSS Corporation

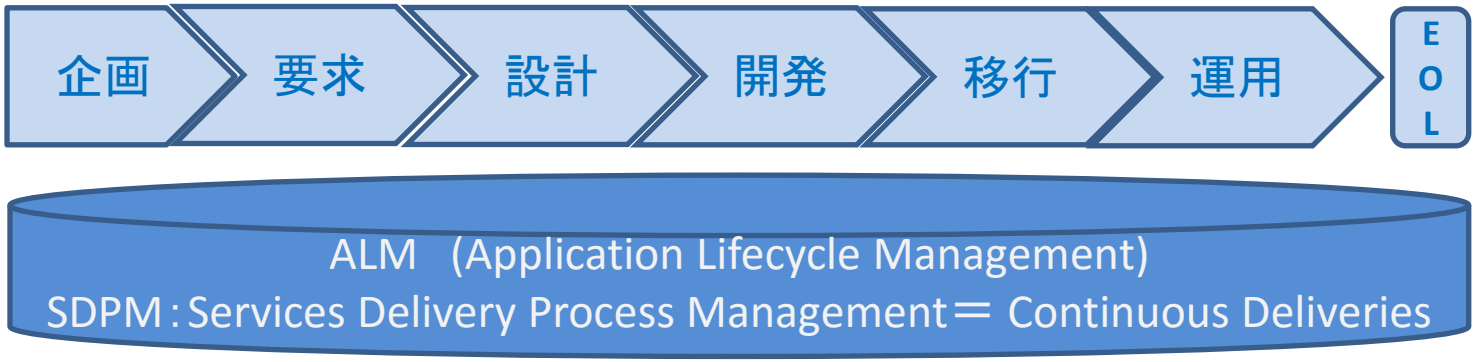
DevOps　Management Map



Copyrights©2015　SSS Corporation

Ⅳ. DevOps　その目指すモノ

ユーザーにITサービスをタイムリーに提供する手法



事業継続性を担保するITサービスの開発・運用・オペレーションの全体最適を指し、事業のタイムリーな変革の足を引っ張らない俊敏性を持つ仕組みを実現できる事です。

ビジネスを止めない

企画からEOL迄、一貫したプロセスの見える化と管理
ITサービスを提供する関連情報の一元化（共有化）も必要となってきます。

Copyrights©2015 SSS Corporation

Ⅴ．DevOpsでシステムを実装する

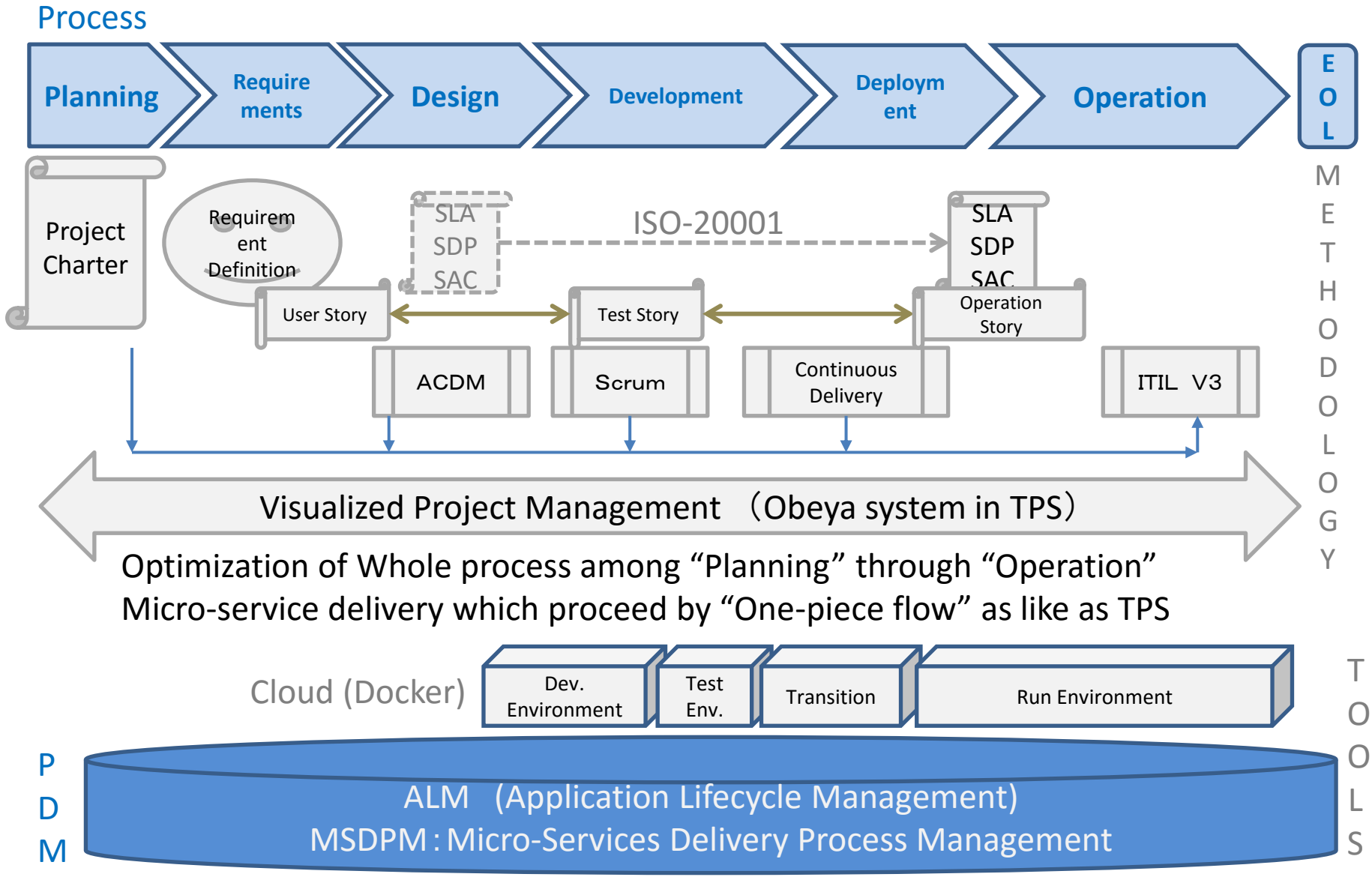
DevOps導入の４大要素

- 規律あるアジャイル開発と継続的デリバリーの仕組み
- 軽量化したITIL V3の仕組み
- クラウド環境　PaaS + Docker
- ALMプロセスでのパラダイムシフト

全ては、ただ一つの基準：『ビジネス価値』

Copyrights©2015 SSS Corporation

Outline of Enterprise DevOps. (Structural Elements)



Copyrights©2015 SSS Corporation

Ⅵ．DevOpsのあるべき姿

Copyrights©2015 SSS Corporation

開発から運用までの全プロセスの全体最適

管理体系が異なる性質の仕事で構成されるプロセスの全体最適を狙った一元化をどう実現するのか？

トータルTPS/TMSでの大部屋方式の導入
- ◆全プロセスでの見える化
- ◆『一個流し』でのプロセスの流水化

タスクの『粒度（時間単位）』と『均一化』

全プロセスでの流水化（澱みなく仕事が流れる）の仕組みの構築とその洗練化の為に常にカイゼン活動が働かなければならない。

Copyrights©2015 SSS Corporation

Ⅶ. パラダイム・シフト

バッチ・プロセス　⇒　一個流しプロセス

まとめて作った方が効率的か？　　個別に作った方が効率的か？

システムの設計＆開発の常識（習慣）を変える

運用の管理サイクルを変える（月単位　⇒　週単位）

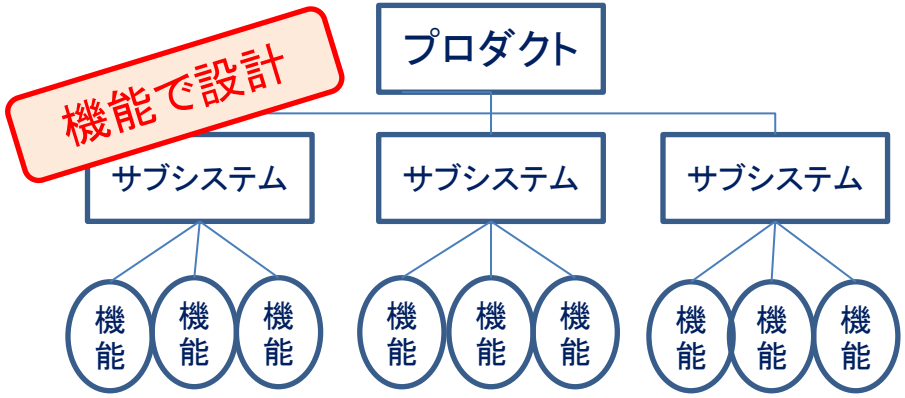
品質はお客様が決める。 品質は開発現場で作り込まれる。（QA部門の役割？）

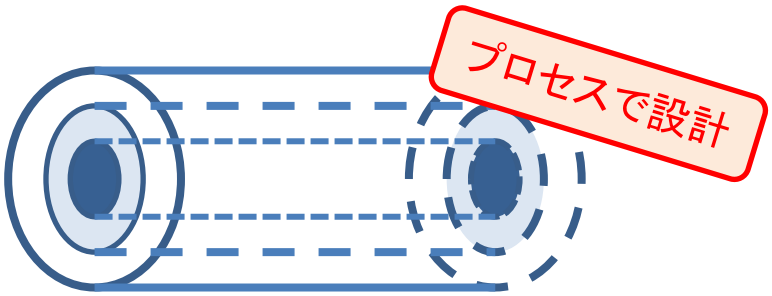
ALM （Immutableと言う概念　＝　EOLの管理）

優れた設計 ≠ 優れたプログラム

タスク ≠ WBS

非機能要件 ≠ 要求仕様





ボディー・プロセス（Body）：　業種を問わず共通している基本プロセス

オルタナティブ・プロセス（Alternative）：　業種特性に影響を受けるプロセス

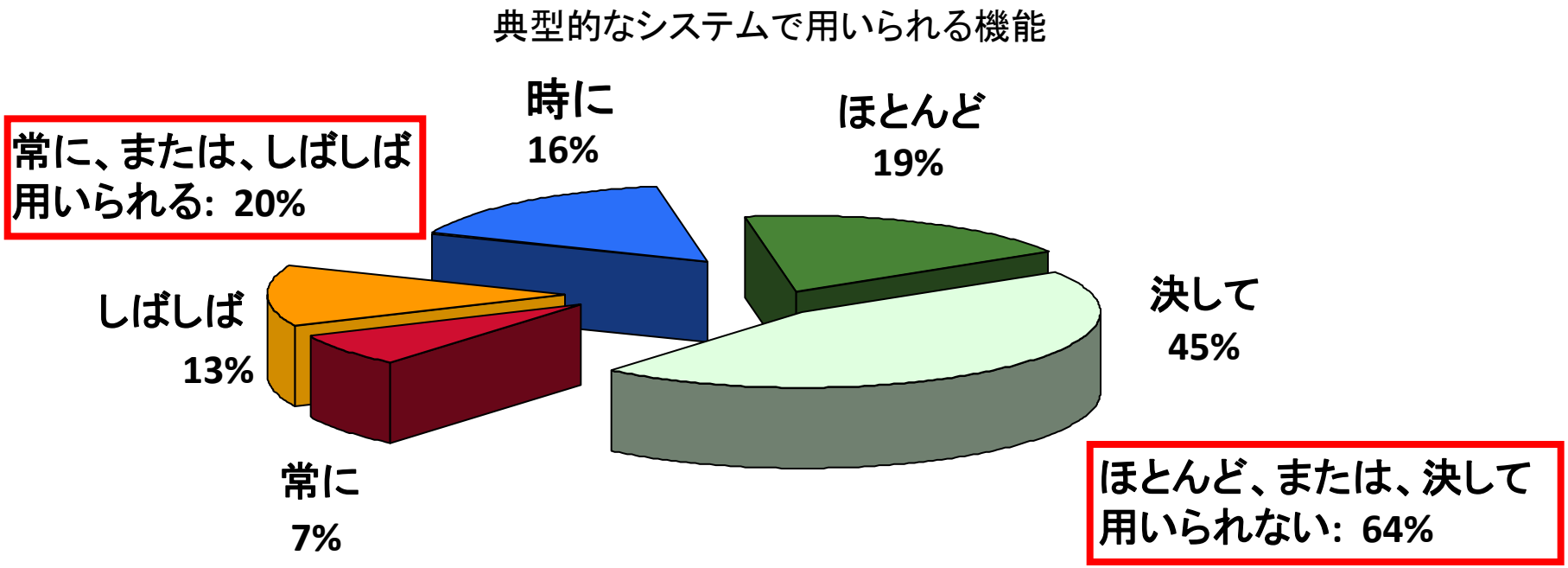
オプション・プロセス（Option）：　企業の商慣習や慣行に影響を受けるプロセス

Copyrights©2015, BSI Corporation

Copyrights©2015 SSS Corporation

システム開発の不都合な真実－1

早期の仕様確定がムダを減らすというのは迷信

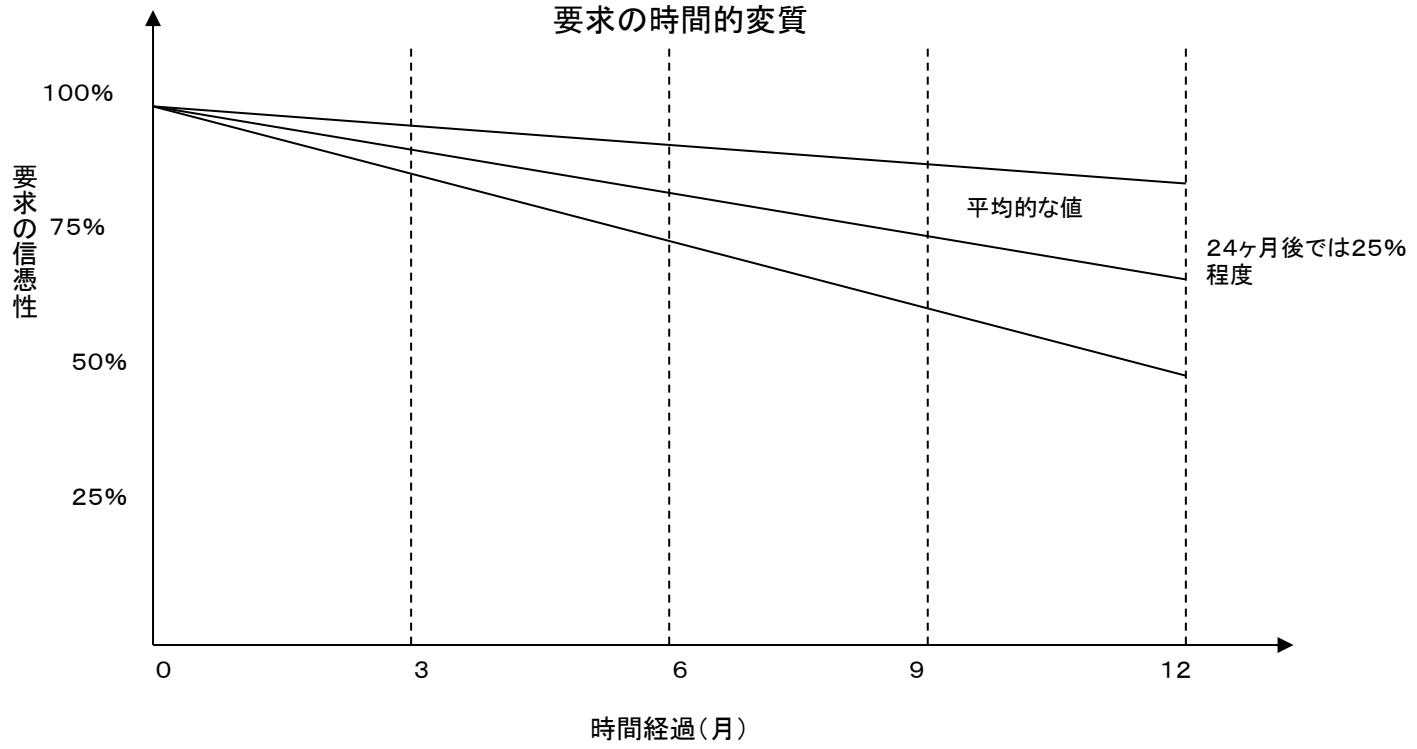


Standish Group Study Reported at XP2002 by Jim Johnson, Chairman

Copyrights©2015 SSS Corporation

システム開発の不都合な真実－２

要求を明確に定義できれば良いシステムができるという迷信



Copyrights©2015 SSS Corporation

アジャイル開発の基本　見えるプロジェクト管理

水蒸気も集まって冷やされて雲になれば見える

開発現場では、色々な情報（データ）が既に存在している
ただ、見ていない。観ようとしない。
現場の説明責任

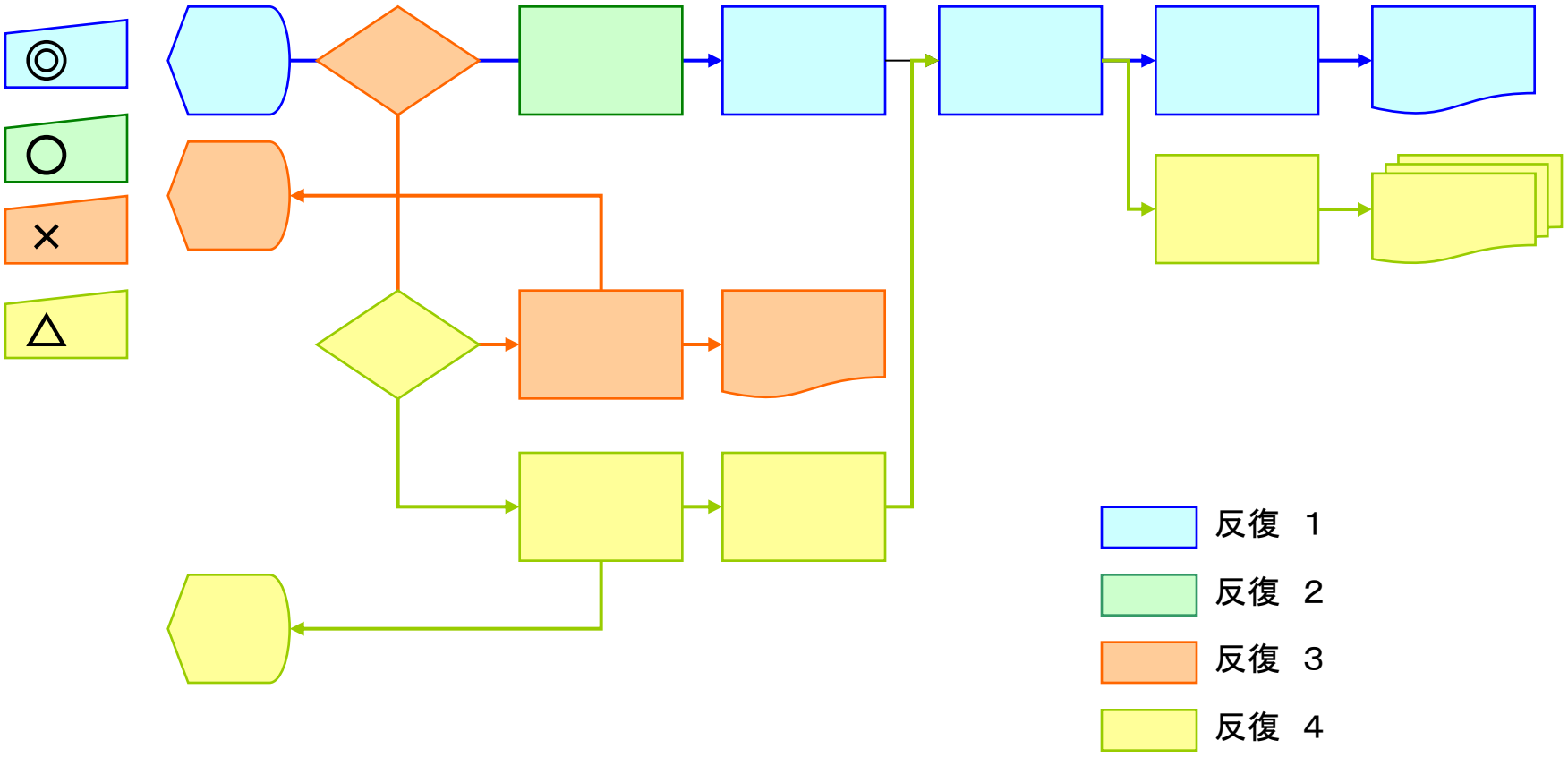
アジャイル開発の肝は、『見える化』 と 『カイゼン』

見えないモノを見える様にする

パラダイム・シフトが起こらなければ、見えない。

Copyrights©2015 SSS Corporation

アジャイル開発におけるソフトウエアの作り方

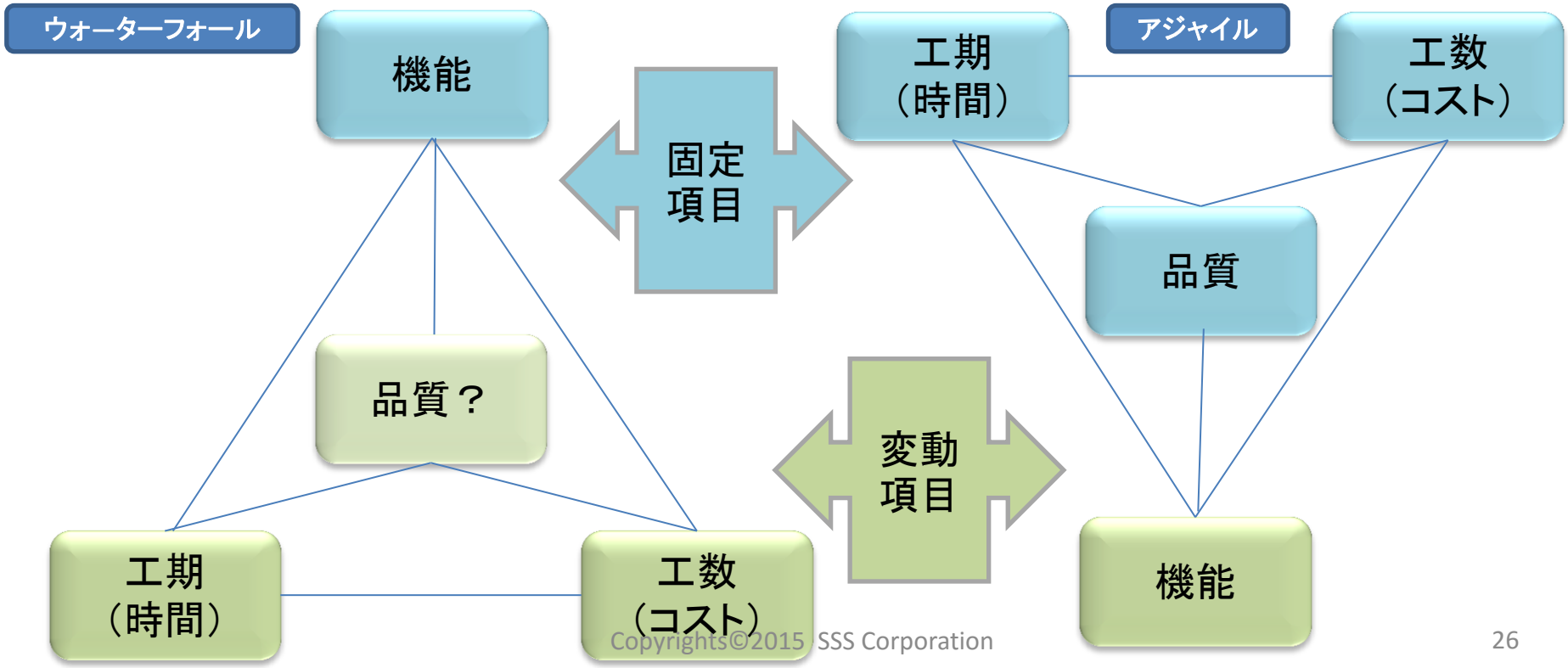


Copyrights©2015 SSS Corporation

アジャイル開発の基本行動

基本的な価値観：

与えられたリソース（資源）の制約下で、最大のパフォーマンスを発揮する。
常にカイゼンを行い、作業効率の向上を目指す。
タイムボックスでの働き方で、時間の有効活用
反復（イタレーション）を利用して、こまめにフィードバックを得て、軌道修正を常に行う。
ゴール（標的）は、固定されていない。常に動く。
常に動作するソフトウエアで確認する。
品質を犠牲にしない。品質を高め、維持する努力。



Copyrights©2015 SSS Corporation

利用者（ユーザー）から見たアジャイル開発のメリット

顧客（ユーザー）から見てこのアジャイル開発手法はどの様なメリットを享受できるのでしょうか？もうシステム屋の言う事は信用しないという経営者の方も多いのではないでしょうか？そうです従来の主流であるウォーターフォール型開発では米国の調査でも計画期間内、計画予算内でプロジェクトが完了した成功率は僅か１６％です。

こうなりますとシステム開発プロジェクトは失敗して当たり前と言う評価を甘んじて受けざるを得ません。アジャイル開発では、この様な失敗を回避可能です。と言いますのも、アジャイル開発とは現有リソース（資源：人、金、時間）を前提に計画を立て実行する手法だからです。

- プロジェクト進捗の見える化

  完了した働くプログラム本数（実現した機能の数）で管理

- 製作している機能の見える化

  ユーザー用語での機能定義（理解できる言葉での設計確認）

  働くプログラムを稼動させて確認

- 絶え間ない擦り合わせ技術での品質向上

  品質とは、要求品質、設計品質、実装品質、検査（テスト）品質

- 優先、重要機能（システム）の早期実現

  システムを構成する重要機能から優先して開発

- 開発プロジェクト期間中での柔軟な変更対応

  プログラム製作期間（イタレーション）に入る前までは、変更自由

- ユーザーが参加すべき作業の見える化

Copyrights©2015 SSS Corporation

スクラム （Scrum）

- Ken Schwaber & Jeff Sutherland （1995年）
- 特徴
  - 軽量
  - 理解しやすい（シンプル）
  - でも、会得するのは難しい
- 三本の柱
  - Transparency （透明性）
  - Inspection （視察、検査）
  - Adaptation （適応、順応、改作）
- 基本的考え方
  - タイムボックス（Time Box）
  - 機能横断的な固定化されたチーム
  - 持続可能なペース

Copyrights©2015 SSS Corporation

スクラム　役割とプロセス

- 役割
  - 開発チーム
  - スクラム・マスター
  - プロダクト・オーナー
- プロセス
  - スプリント計画会議 （パート1、 パート2）
  - デイリー・スクラム （スタンドアップ・ミーティング）
  - スプリント・レビュー
  - スプリント・レトロスペクティブ （振り返り、KPT）

Copyrights©2015 SSS Corporation

スクラム・プロセス

Product Owner
（お客様）

Product Backlog
1.----------
2.----------
3.----------
4.----------
5.----------

確実な優先順位付け
（同位は無し）

追加、修正、変更可能
（Sprintに入っている物以外）

Product Ownerが全て責任を持つ

Sprint Planning

Part 1
2時間

Part 2
2時間

Scrum Master
（ファシリテーター）

Team　デザイン・開発・テスト

Developer (Pair Programming)

毎朝15分のスタンドアップ・ミーティング

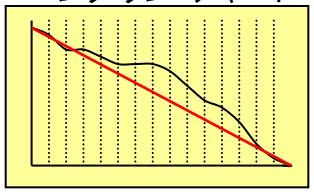
Sprint Task List
(Sprint Backlog)
a.----------
b.----------
c.----------
d.----------
e.----------
f.----------

Daily Scrum

タスクボード

To Do	On Going	Done
c, d	b	a
e, h	g	f

バーンダウン・チャート



Sprint Reveiw
1時間

Retrospective
2時間

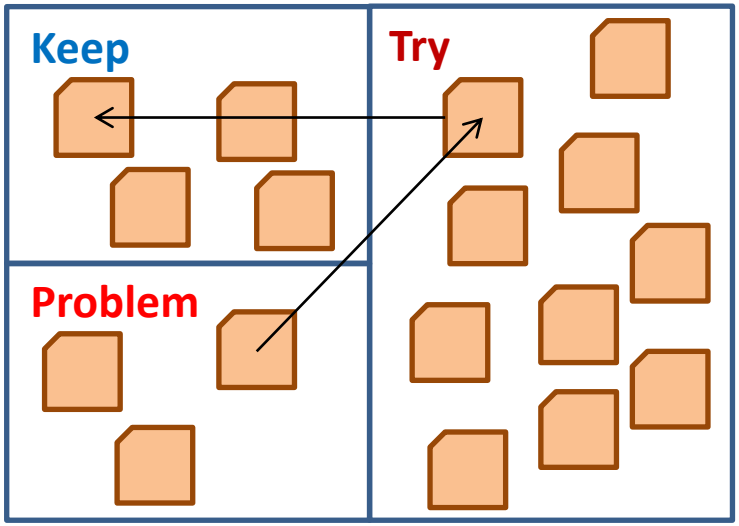
Sprint (Iteration)

1週間～4週間

Copyrights©2015 SSS Corporation

KPTでの振り返り（レトロスペクティブ）

- 毎週行われる進捗報告ミーティング
- 評価とプロセスの改善の重要な機会
- PTKをチームが共有する
  - P（Problem：問題となっていること）
  - T（Try：トライしたいこと）
  - K（keep：キープしたいこと）



小さなPDCAを回すための方法です。

KPTボードは振り返りのツール

①気づきや問題、課題を付箋紙に書き、Problem（問題）のエリアに貼る。

②Problemのエリアに出てきた気づきや問題に対してTry（対策）を立ててTryのエリヤに移動させる。

③実際にやってみて、対策が不十分だった場合には、再度、Problem（問題）のエリアに移動し課題を追記する。再度、対策を立て直す。

④Problemに対する効果が十分出たと判断できた場合には、Keep（継続）のエリアに移動し習慣化、定着化を図る。

ここで大事な事は、**人を責めずにやり方を責める**事です。（誰でも同じようにできる様にする。）

Copyrights©2015 SSS Corporation

スクラム・プロジェクト

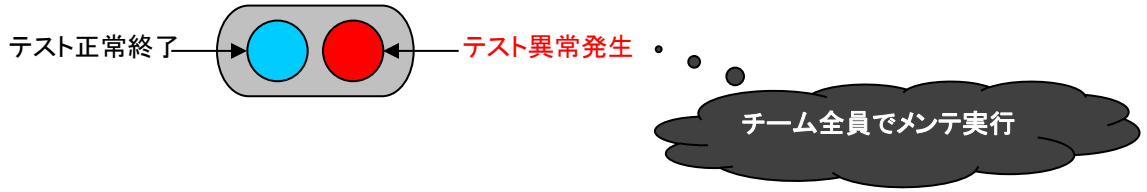
- 計画
  - 計画は常に作り直す。(『計画』よりも『計画作り』が重要)
  - 2レベル計画 (リリース計画、 スプリント計画)
  - 実測駆動の見積り
- タスク
  - WBSとは異なる。(作業指示書:手順を表記)
  - 粒度が重要(できるだけ小さくすると、平準化、多様化に対処できる。)
  - Done(作業完了)の定義
- 要求
  - ユーザーストーリー
- 設計
  - プロセスで設計(ボディー・プロセス。 オルタナティブ・プロセス。 オプション・プロセス)
- コーディング
  - コードの共有(属人性、私物化の排除)
- テスト
  - 単体テスト(TDD)
  - 継続的インテグレーション〔常時結合)=行灯
- 管理
  - 目で見る管理
  - 働くコードで進捗
  - ベロシティーで納期(巡航速度)
- チーム運営
  - 自主研(自主改善活動)=小さなPDCA
  - 報・連・相(毎朝のスタンドアップ・ミーティング)
  - モチベーション(ニコニコ・カレンダー)
  - 残業はチームで相談(全員で残業)

Copyrights©2015 SSS Corporation

継続的インテグレーション（常時結合）

毎日実施

- 行灯（パトライト）
- 自動テスト



- 開発したプログラムを毎日統合して動作確認を行う
- システムの進捗を日々の動作機能で測る事が可能
- 開発チーム内での統合動作不良・改善を日々行う事が可能
- 自動化ツールを使用して開発チームが休息している間も統合試験が可能
- 品質向上に貢献

10分間ビルド

Copyrights©2015 SSS Corporation

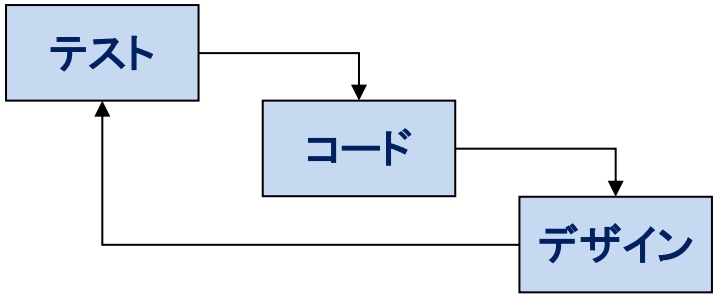
XP（エクストリーム・プログラミング）

- Kent Beck（1999年）
- テスト駆動開発（TDD）＝テストファースト・プログラミング
  ① 実装する機能をテストするプログラムを書く。
  ② コードを書いてテストする。
  ③ デザインを見直す。
  ④ 信号が青になるまで２＆3を繰り返す。
- リファクタリング
  - 完成したコードの見直し（実装された機能を変えずに、コードをシンプルに、見やすくする）
  - 任意の作業（全員が行う。 時間が空いたら行う。）
- ペア・プログラミング
  - ドライバー（コードを書く人）
  - ナビゲーター（コードをチェックする人、ナビゲーションをする人）
  - この役割を1日の中でペアの間で、途中で交代する。
  - ペアの組み合わせを毎日替える。
- １０分間ビルド
  - 自動的にシステム全体をビルドして、全てのテストを10分以内に実行させる。（理想形）

Copyrights©2015 SSS Corporation

テスト駆動開発（TDD）

- **テスト駆動開発**
  - テストの自動化
  - アクセプタンス・テスト（完了・終了の定義）



小さいテストのコード作成
次にソース・コードを作成
最後にデザインをリファクタリングする。
このプロセスを廻す。

Copyrights©2015 SSS Corporation

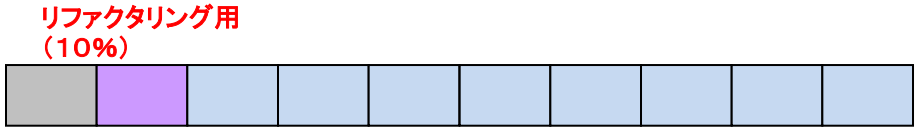
リファクタリング（Refactoring）

正常に動作確認したプログラムに対して下記目的でソースコードの調整を行う作業

- プログラムの可読性を高める
- クラス化・部品化
- 変更容易

仕様レベルでの調整

- 使い易さ
- 誤操作抑止
- 仕様変更



リファクタリングのルール

- リファクタリングと機能改変を同時にはおこなわない。リファクタリングをしてから新しい機能を追加する。
- リファクタリングを始める前と後にはユニットテストを実行しコードの機能が変更ないかを確認する。
- パフォーマンスよりもメンテナンス性を重視する。
- こだわり過ぎてはいけない。
- 小さなリファクタリングとテストの組み合わせを繰り返す。決して一度に大きなリファクタリングをしない。
- *大きなリファクタリングは惨事のもとである。*
  ～Kent Beck

Copyrights©2015 SSS Corporation

リファクタリング基本方針

- コードをできるだけ小さな単位に分割する
- 処理のかたまりに名前をつける
- 処理を重複して記述しない
- リファクタリングの基本的方針のひとつめは、コードをできるだけ小さな単位に分割することです。
  小さな部品のほうが理解しやすく、バグも少なくなるからです。

- また、コードを小さく分割することで、そのまとまりごとにメソッド名やクラス名などの名前を付けることができます。
  実は、この名前がコードがわかりやすくなるかどうかの分かれ目になります。
  コメントを充実させるよりも、リファクタリングによってコードの断片をまとめ、適切な名前を付けるべしというのがリファクタリングの教えです。
  （もちろん、実際にはコメントも重要ではありますが。）
- そして最後に重要なのが、コードの重複をできるだけ少なくすることです。重複コードを見つけたらまず間違いなくリファクタリングの対象になります。
  重複するコードがあるということは、プログラムの変更時に変更しなくてはならない箇所が分散してしまい、変更時のバグが入り込みやすくなってしまうということです。それを防ぐためには、重複するコードをできるだけ少なくするというのはコードの保守性を高めるためにきわめて重要な作業となります。
  ただし、あまりに共通ルーチンを使い回すと、逆にメンテナンスのしやすさを低下させてしまうこともあることを覚えておいてください。
  あまりに多くのモジュールから、ひとつの共通ルーチンが参照されているときは、その共通ルーチンを変更するときの影響が大きすぎることになってしまいます。
  このへんのさじ加減を広い視点で判断し、コードの保守を行っていくことが重要です。
- *どんなプログラマーでもコンピュータに理解できるコードは書ける。しかし優秀なプログラマーだけが、人間に理解できるコードを書くことができる。*
  ～Martin Fowler

Copyrights©2015 SSS Corporation

ペア・プログラミング（Pair programming）

- 二人で一台の端末に向かってプログラム製造を実施する
  - 毎日違ったペア
    - 緊張感の維持
    - この瞬間にこの作業しか出来ない！！
  - 役割を一時間前後で交代
    - 作業者とチェッカー
    - 曖昧な作業が無くなる
    - 手戻りの軽減
  - 品質の向上
    - 動けば良いと言うような安易なプログラム製造ではない
    - 可読性の高いコード
  - 頭脳労働のムダとり
    - 一人で考え込む時間が無くなる。

ペアの交替（定期的：毎日）
作業の交替（一定時間毎）：ドライバーとナビゲーター

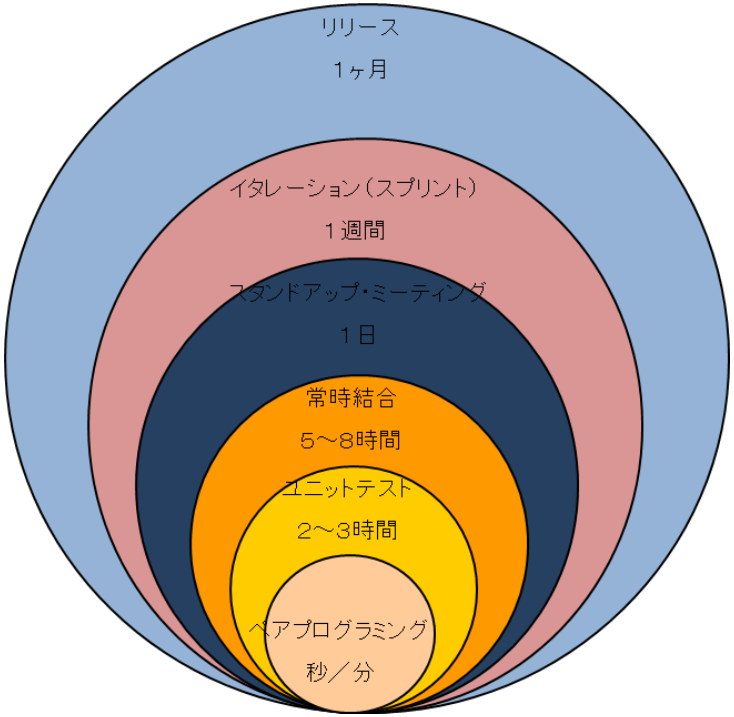
静かなペアプロは
機能してない

Copyrights©2015 SSS Corporation

品質について考える

当り前品質 ⇒ 顕在品質 ⇒ バグフリー
アジャイル・プラクティスの徹底（守）
頻繁なフィードバック・ループの設計

魅力的品質 ⇒ 潜在品質 ⇒ メンテの容易性
ソースコード　＝　ドキュメント
メソッド名と変数名　（名は体を表す）
簡易に書けるセカンド・ランゲージ
（自分のツール作成）



Copyrights©2015 SSS Corporation

アジャイル開発で品質が向上する理由

◆ ムダ取りの原則

作業にムラがあるから、ムリをするようになる。
ムリな作業をした結果、ムダが生じる。
ムラを防止するのは、一定の作業リズム（タクトタイム＝タイムボックス）

◆ タスクの粒度を小さくする。

作業手順（工程設計）を考えなければタスクは小さくできない。
タスクが小さくなれば、ミスを容易に発見できる。手戻りも小さい。
タスクを小さくするとムダが見えてくる。
正味（本来の価値を作り込む）作業、付帯（事前、事後の）作業、ムダな作業
タスクを小さくすることで、整流化が容易になる。（ボトルネックの解消： 一個流し生産）

◆ 全員での作業で透明性が高まる。

一人で抱え込む仕事がなくなる。
事前に他人の目が届く（チェック）

◆ トヨタ生産方式（TPS）の自働化の思想をプロセスに組み込める。

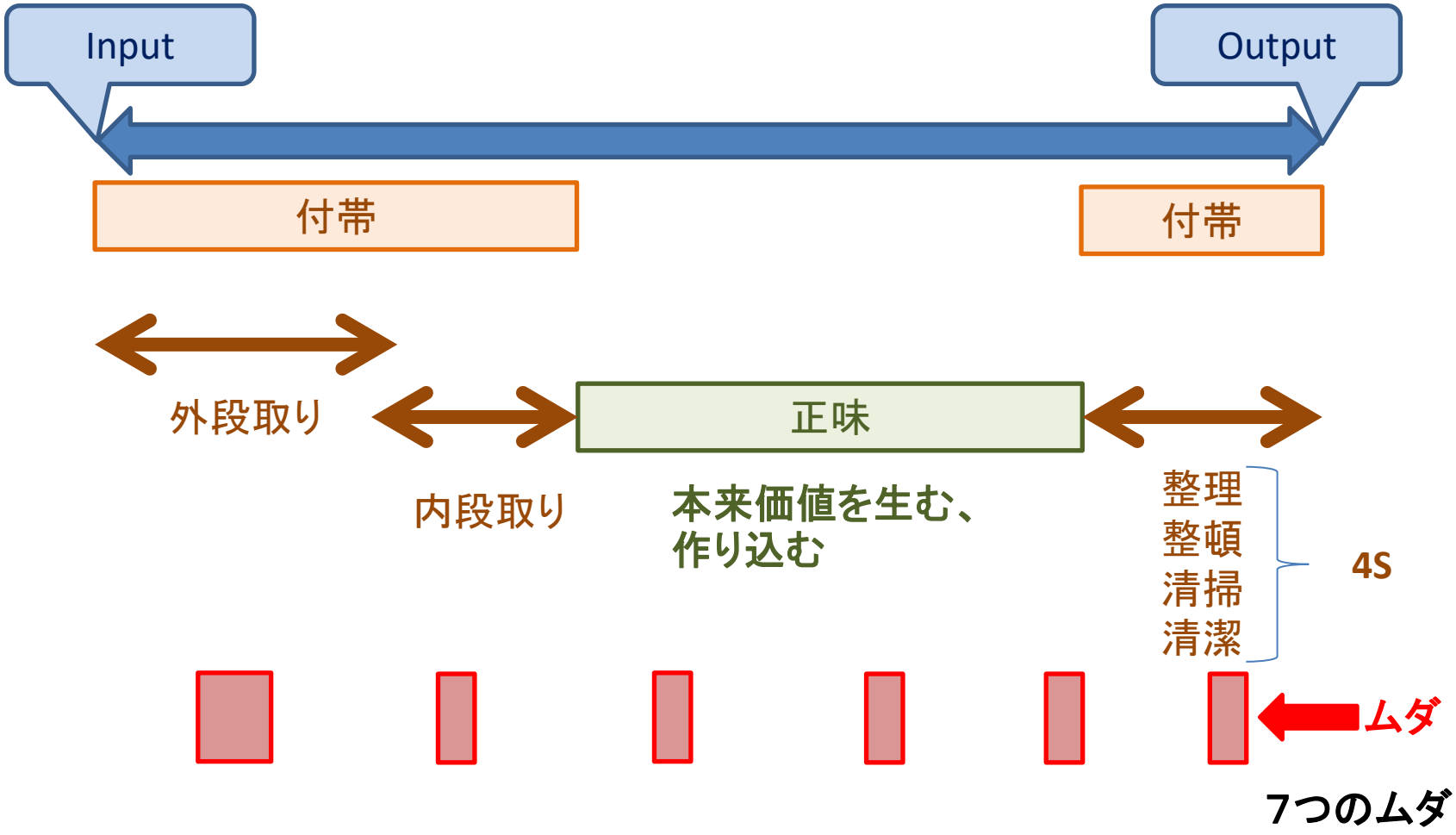
不良作業をしない。不良品を流さない。不良品を受け取らない。（自工程完結）
不良が起こったらラインストップをして、関係者全員が異常に気づく。（見える化）

◆ 作業者（開発者）に直接フィードバックする仕組みが構築できる。

『擦り合わせ』をしながら作業が進む。

Copyrights©2015 SSS Corporation

仕事を分析する



①作りすぎ、②手待ち、③運搬、④加工そのモノ、⑤在庫、⑥動作、⑦不良を作る

Copyrights©2015 SSS Corporation

ソフトウエア製造工程におけるムダの廃除

1. 作りすぎのムダ
   顧客に使用されない機能、実際には不要な機能、真のビジネス価値を生まない機能などの余分な機能を作らない。
   StandishのCHAOSレポートによるとソフトウエアの全機能の64%は全くあるいは殆ど使用されていない。
2. 手待ち（停滞）のムダ
   仕様の提示遅れによる製造開始遅れでの待機、許可待ち、ビルド待ち、障害発生によるテスト待ち
3. 運搬のムダ
   複数プロジェクトでの作業の切り替え、仕様入荷チェック、納品出荷チェックの提供＆受領双方での重複チェック
4. 加工そのもののムダ
   開発現場で機能していない作業項目例えば余分な事務処理、報告書作成や作業分担の誤りによる過剰な作業
5. 在庫のムダ
   最終工程で使用されない文書や計画、コンポーネントなどの中間的な作業成果物と待ち状態で仕掛中のプログラム
6. 動作（作業）のムダ
   関係者の作業場所の移動や複数の開発ツールを使用して開発ツール間の切替・移行
7. 不良をつくるムダ
   バグの作り込み---要求仕様(要件）、設計、コードの欠陥

Copyrights©2015 SSS Corporation

タスクの粒度

- タスクの粒度を小さくすることはTPSにおける小ロット化と同様
  - 「流れ」を作り、負荷を平準化し、柔軟性を高める

Application size, Test Cases, and Test Coverage.
Logical source code statements
By Caper Jones

Statements of Source code	Test Cases	Test Coverage
1	1	100.00%
10	2	100.00%
100	5	95.00%
1,000	15	75.00%
10,000	250	50.00%
100,000	4,000	35.00%
1,000,000	50,000	25.00%
10,000,000	350,000	15.00%

- タスクの粒度は小さいほど良い
  - １日以内、理想は1時間
  - 責任を持って見積ができる
  - バグを作り込まない（簡単にテスト可能）
  - 他のペアと同期がとれる
  - ダイナミックなプロジェクト運営が可能となる（チーム編成の増減、分散開発など）
  - タスクが小さくできないのは、作業対象の内容把握に問題が存在するのではないか？
  - タスクを小さく分割するという事は、作業指示書を作成する事。

Copyrights©2015 SSS Corporation

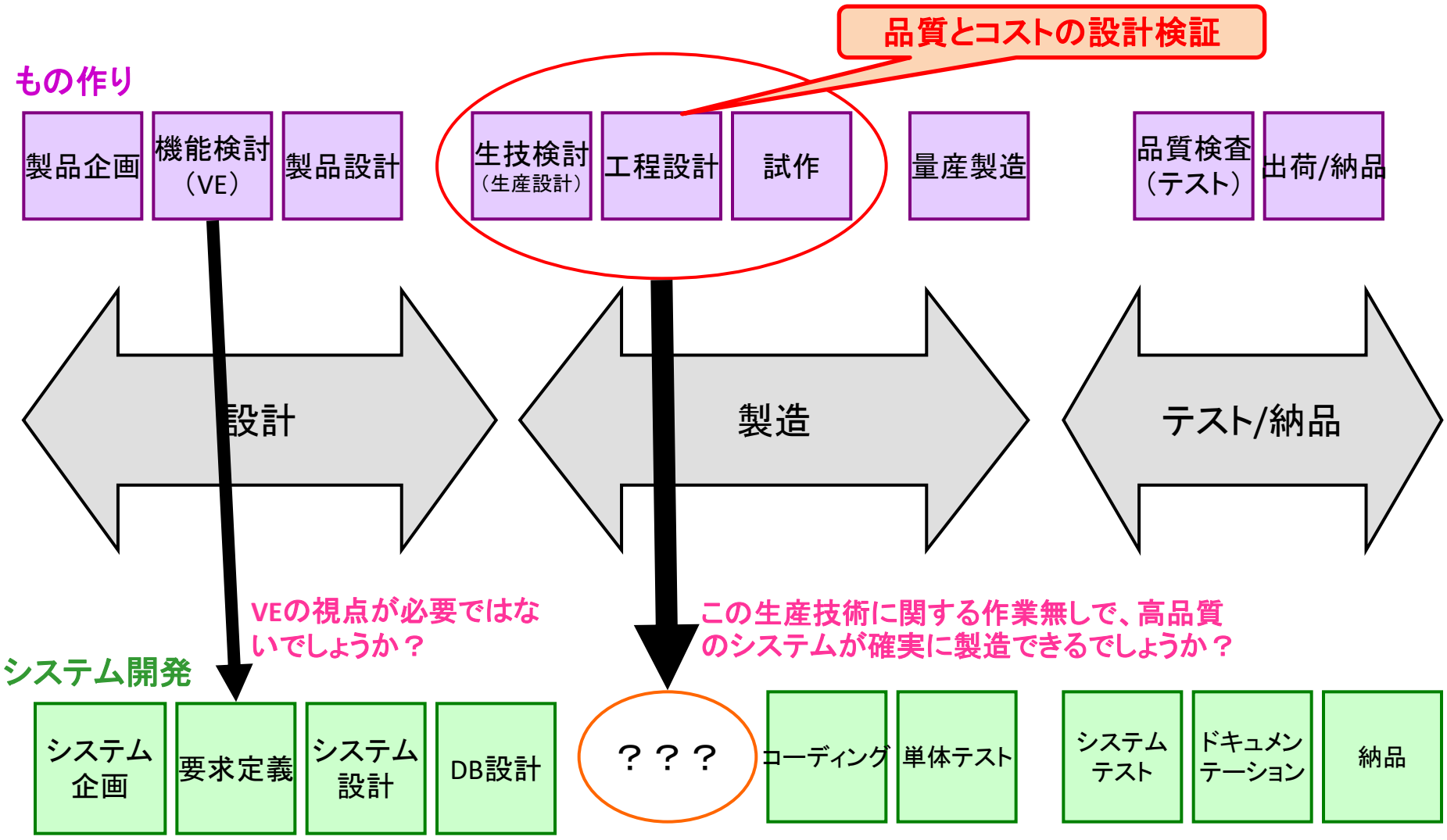
タスクを小さく（粒度）する

例えば、レポートを作る業務（仕事）をタスク分解する。

1. レポートの主旨を確認し、レポートのストーリーを練る。	30分
2. レポートの章立てを決める	10分
3. 各章の基本を決める（文章、図、グラフ、データ、イラスト等）	30分
4. 文章の下書きをする。	30分
5. 図やグラフを作成するためのデータを決め、データを収集する。	30分
6. PCを立ち上げ、EXCEL、PowerPointを起動する。	5分
7. データをEXCELに入力する。（データをインポートする。）	20分
8. グラフを作成する。	10分
9. 文章を構成し、PowerPointに入力する。（コピーする。）	30分
10. PowerPointのレイアウトを決め、文章、図、グラフ、イラストを配置する。	5分
11. ④～⑩を必要ページ分繰り返す。	
12. レポート全体を通して確認する（校正する。）	30分
13. 作成日、作成者名、レポートの題名を記入し、完成させる。	5分
14. レポートを提出する。	5分
	240分
	（4時間）

Copyrights©2015 SSS Corporation

『もの作り』と『システム作り』の相違



設計仕様を図面上のみの検討で、十分でしょうか？

擦り合わせ、微調整が必要ではありませんか？（誰が、何時、どの様に作業できますか？）

Copyrights©2015 SSS Corporation

経験事例の報告（エンジニアの声）

エンジニアAさん： アジャイル初体験、ウォーターフォール開発経験約１５年（主任クラス）３７才
＞情報処理技術者　１種　＞UMTP　L1
＞業務SE（物流／生産管理／公共サービス）　８年　＞ホスト汎用機テクニカルSE　７年
＞今回　．NETは初めての経験　＞JAVA（web）　３年
＞PL／SQL　２年　＞VB．NET　（２週間：本アジャイル開発で初）
＞COBOL、C＋＋　etc（細かい開発は多数）

アジャイル開発の効果： コーディングの生産性は変わらないが、開発作業内で手戻りが無く、結果的に早く完了できる。

体験した感想： とにかく頭が疲れる。集中する。

・品質が高くなる。
・技術的な問題や、方式で悩む時間が少ないので効率は良い。
・気を抜く暇がないので、稼働率は高い
・二人でやっているので、生産性は倍まではいかない。ただし品質が高いので、改修やテスト時の修正工数は少なくなる
・ペアプロ／クロスファンクションにより　ソースコードレベルで情報を共有するため、自然に　可読性／ロジックのシンプルさが感じられる実装となる。
・随時に動かしながら機能拡張をするため、潜在バグ／デグレードのリスクは低い。
・実装が不慣れな要員がいても、ペアの組み合わせにより品質の高い実装が可能となる。
・作業の完了が、視覚的に理解できる。実装の成果がすぐに見れる。
・スタンドアップミーティング／振り返り／タスクの割り振りによりメンバー全員が全体の作業を見渡せる。司会を持ち回りすることにより参加意識が強調される。
人に見られているのに、適当な（動けばいい）コーディングはできない。
・悩んでいる時間が少ない。（随時相談／調査）
・具体的な目標を随時持つことができる。
・タスク担当を明確にすることにより、責任範囲の当事者意識を持つことができる。

Copyrights©2015 SSS Corporation

経験事例の報告（エンジニアの声）

エンジニアBさん： アジャイル初体験、ウォーターフォール開発経験約5年　28才
＞ Oracle Master Bronze 10g　　＞ VB.NET：1年
＞ HTML、JavaScript、CSSなどWeb関係：1年（随時）　　＞ PHP：1年
＞ VB6：3年　　＞ PL/SQL：1年
＞ XML：1ヶ月

アジャイル開発の効果： 生産性が上がった（体感）。

体験した感想： 個人のコミュニケーション能力が高く問われる
・二人で作業を行っているため、単純ミスも少なく精度が高い。
・思った以上に疲れる。気を抜く暇がない。

エンジニアCさん： アジャイル初体験、ウォーターフォール開発経験約4年27才
＞ 初級アドミニストレータ
＞ 詳細設計・PG・/テスト（物流／在庫管理／Web） 4年
＞VB. 2年　　＞ JAVA（web） 3年
＞ PL／SQL 三ヶ月

アジャイル感想：
・一人で開発をしている時は、技術的にわからないことがあるとネットや本等で調べて解決し、作業を進めていくが、ペア・プロの場合横で他の人が見ているので、気軽に調べ物がしにくい。
（まだメンバーに慣れていないため、気軽に質問ができない）
・仕様をよく知っている人とペアを組むと、プログラミングの道筋を立てて開発ができるので、良いと思う。
・反対にある程度わかっている人が作業を行い、ほとんどわかっていない人が横で見ている場合、作業者はどこまで説明しながら進めればいいのかわからない。
・ペア・プロだと常に他の人と一緒に開発を行うので、緊張感が保てる。その反面、気を抜くことができないので、一人で作業をするよりもかなり疲労度が高い。
・タスク毎に見積もり時間を出して作業を行うことと、プログラム1本の見積もり時間しか出ていない状態で作業を行うことと比べると、タスク毎の方が作業を進めやすい。
（プログラム1本の見積もりの場合、ペース配分が上手くいかないことがある）
・ウォーターフォール型の開発に慣れているので、要件定義からいきなりプログラミングに入ることにまだ戸惑いを感じる。外部設計書、内部設計書があった方が開発がしやすい。
・ペア・プロに慣れてきたので、作業進捗が早くなってきた。
・常に隣に他人がいる環境でずっと開発を行うのは疲労度が高く、辛い時もある。

Copyrights©2015 SSS Corporation

ご清聴ありがとうございました。

戸田　孝一郎

米国スクラムアライアンス認定スクラムマスター

IBM認定　アジャイル開発インストラクター

EXINアジャイル・スクラム・ファンデーション認定講師

（社）TPS検定協会　理事　認定TMSカイゼン塾　コーチ

ScrumAlliance™
transforming the world of work

http://www.Ask3S.com

Strategic Staff Services

株式会社　戦略スタッフ・サービス

（本社）〒100-0004 東京都千代田区大手町1-7-2 東京サンケイビル27F 電話：03-3242-6282 FAX：03-3242-6283

お問い合わせはメール（lktoda@ask3s.net）にてお願いいたします。

Copyrights©2015 SSS Corporation