Software Engineering Topics (Development Methodology etc.)

Picolab Co., Ltd.　Yasukazu AOKI

2016. 6. 17

概要

- １．　ゴールの決め方
  - 成功の基準
- ２．　何を作るかの決め方
  - 要件定義・品質・要求開発
  - アーキテクチャ
- ３．　どうやって作るかの決め方
  - 開発方法論/開発手法
  - 開発プロセス
- ４．　作る際の注意：開発計画と進捗管理
- ５．　作る際の注意：プロジェクト管理
- ６．　作る際の注意：開発環境/開発支援ツール
- ７．　プロジェクト成功のために
  - よくある問題
  - チーム作り

Summary

- 1. Define the Goal
  - Criteria for Success
- 2. How to decide what to make
  - Requirement definition, Quality, Requirement Development
  - Architecture
- 3. How to decide how to make
  - Development methodology
  - Development process
- 4. Development plan & Progress management
- 5. Project management
- 6. Development environment/Development support tools
- 7. The key to project success
  - Common problems
  - Team building

PICOLAB

1. ゴールの決め方

- 「成功の基準」を最初に考えてみる
  - 誰が何を決めるか？（ステークホルダー分析）
    - チームのメンバ（の中の誰か一人が案を作る）
      - 最終的にチームメンバみんなの合意は必要
    - 評価する人、利用する（利用してくれそうな）人（と相談）
  - どう決めるか？
    - 段階的な成功基準を考えておく方がリスクヘッジはしやすい
      - 最低：「△△△」は最低限実現する必要がある＆実現できそうだ
      - 現実：「〇〇〇」がチームの力で実現できそうな中で最も良さそう
      - 挑戦：「◎◎◎」は実現困難だが、うまくいけばより多くの人に役立つ
    - 成功基準に含めないものも明示してみると、フォーカスがより明確に
      - 例1：UIの使い勝手はスコープ外（アルゴリズムの開発に集中）
      - 例2：スケーラビリティはスコープ外（新しいユーザ体験可視化に集中）
- 最初に考える成功基準は「仮説」のようなもの
  - やってみないと分からないこともある
    - その成功基準（仮説）で大丈夫そうかどうかの確認（検証）を早めに行うべき
      - 似たような開発の調査、プロトタイピング、チームメンバの実力確認（課題分担）等
    - もし成功基準が適切でなければ、早めに修正すべき
      - とはいえ、実際にはモノを作っている途中でゴールに問題があると分かるケースも
      - その時に重要なのがチーム内のコミュニケーション（日頃の飲み会？）

1. Define the Goal

- Criteria for Success (Core Values)
  - Who decide what? (Stakeholder analysis)
    - A member of the team (makes a plan)
      - Final consensus of all team members is necessary.
    - Evaluators, (Potential) Users, …
  - How to define?
    - Consider the stage of the success criteria to hedge the risks
      - Minimum : Need to achieve
      - Realistic : Looks good and most likely to be achieved
      - Challenging : Difficult to achieve, but more useful
    - What are not part of the success criteria?
      - Exsample1 : Usability of the UI is out of the scope
        (Focus on the development of the algorithm)
      - Exsample2 : Scalability is out of scope
        (focus on the user experience visualization)
- Initial success criteria is a "hypothesis"
  - Do not try it, do not find it
    - Early confirmation (verification) of the success criteria (hypothesis)
      - Similar development studies, Prototyping, Measurement of the ability of team members (by Task sharing), etc.
    - If the success criteria is not appropriate, you should change it ASAP
      - The problem of the goal may be found during the development
      - Communication within the team is important at that time

2. How to decide what to make

- Requirement definition
  - Functional requirements
    - Scenario：Main use case, Variations, Exceptions（準正常系、異常系）
    - System boundary：Screen（User Interface）, External system interface
  - Non-functional requirements
    - 品質が良い ⇒ 「品質」って何？
      - 性能、スケーラビリティ、保守性、拡張性、使いやすさ（ユーザビリティ）等々
- Requirement development
  - Openthology（Open Enterprise Methodology） ⇒ 要求開発アライアンス
    - 戦略的IT化を推進するための組織形成やプロジェクトの構成方法
    - 経営課題やビジネス要求を構造的に分析する方法
    - ビジネスの構造やメカニズムを視覚化する方法
    - ステークホルダーの合意を形成しつつプロジェクトを推進する方法
    - 企画したシステムによるビジネス的価値・効果の検証方法
    - ビジネス要求とシステム要求のトレーサビリティの管理方法
  - “Kotatsu” Model
    - Remove the wall among workers, managers and engineers
- Architecture choices
  - 「アーキテクチャ」って何？
    - 全体と部分の関係
      - 問題分割、依存関係
    - 例えば
      - SOA：Service Oriented Architecture
      - MVC（Model View Control）アーキテクチャ

2a. What is Quality?

- 一般的な品質の定義
  - 定義1
    - 企画品質（quality of planning）
    - 設計品質（quality of design）
    - 製造品質（quality objectives）
    - 使用品質（fitness for use）
  - 定義2
    - 当たり前品質（must-be quality ）
    - 一元的品質（one-dimensional quality）
    - 魅力的品質（attractive quality）
  - 定義3
    - アトリビュートマトリックス

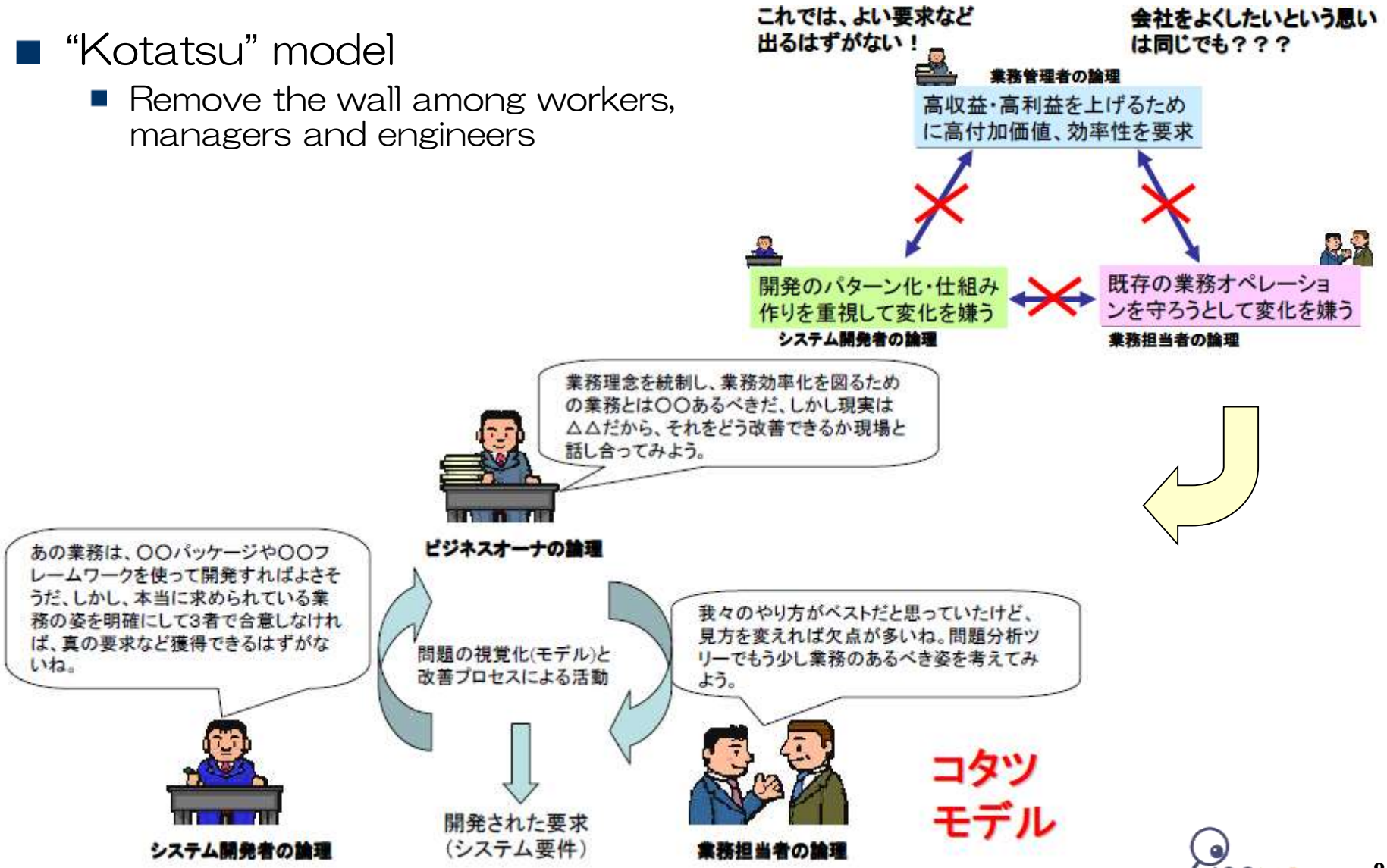
	基本的特徴	差別化的特徴	決定的特徴
肯定的特性	あって当り前	ちょっと違う	興奮する
否定的特性	我慢できる	文句を言いたい	何だこれは
中立的特性	だから何なの	おまけなら欲しい	

出典: http://www001.upp.so-net.ne.jp/meru/shindanshi/seisan/link72.htm

- ISO等の国際標準における品質定義
  - 「ソフトウェアが指定された効用を発揮するために必要な全ての特性」
    (ISO/IEC 9126-1 Software engineering - Product quality - Part1: Quality model)
  - 「品物又はサービスが、使用目的を満たしているがどうかを決定するための評価の対象となる固有の性質・性能の全体」
    (JIS Z 8101)
  - “A quality is a characteristic that a product or service must have. For example, products must be reliable, useable, and repairable. These are some of the characteristics that a good quality product must have. Similarly, service should be courteous, efficient, and effective. These are some of the characteristics that a good quality service must have. In short, a quality is a desirable characteristic.”
    (http://www.praxiom.com/iso-definition.htm ISO 9000の平易版)

2b. Requirement Development

- "Kotatsu" model
  - Remove the wall among workers, managers and engineers



2c. アーキテクチャ ≒ 問題分割構造

- 共有による効率化
  - 同じ効果で投資をN→１
    - 例：ファイルサーバ
      - HDや管理等の投資をN→１に削減
  - 少ない投資で効果を１→N
    - 例：知識の共有
      - 共有による効果が大きい（と期待される）知識とは？
        - » 例えば、過去の業務で実際に役立った知識
      - 共有の手間を小さくするには？
        - » 例えば、知識そのものでなく誰が知っているかを共有
- 分割による効率化
  - サイズを分割した方が開発コストは減る
    - 開発コスト＝サイズ$^{\alpha}$ （$\alpha > 1$）
    - $(a+b)^{\alpha} > a^{\alpha} + b^{\alpha}$ （$\alpha, a, b > 1$）
  - 分割のためのコスト
    - アーキテクチャとインタフェイスの設計
  - 分割によるコスト削減が分割のための投資を上回るように分割する
    - 低い結合度、高い凝集性
- コンポーネント（部品） ＝ 分割 ＋ 共有
  - 物理的な部品は、設計や生産設備は共有できても部品そのものを共有することはできない
    - 例：LSI
  - ソフトウェア的な部品は、電子的なコピーのコストだけで部品そのものを共有できる
    - 例：クラスライブラリ
  - ネットワーク的な部品は、通信コストだけで部品そのものを共有できる
    - 例：Webサーバ、ネットワークプリンタ

DSM（Dependency Structure Matrix）による依存関係分析（Lattix）

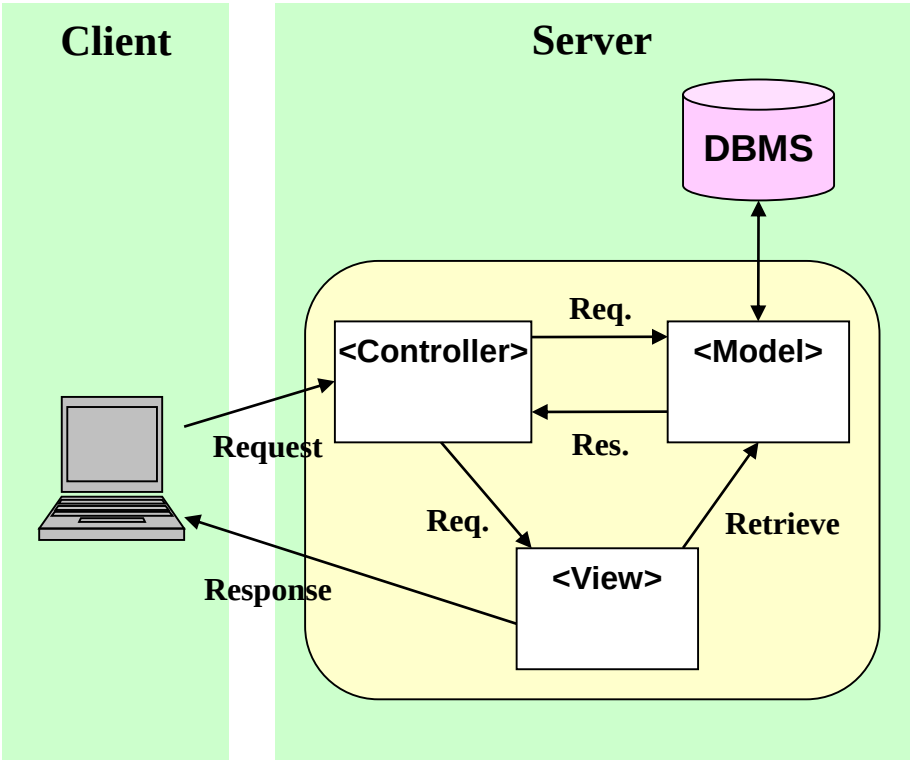
2c. Architecture ≒ Division of the problem

- Sharing
  - Investment: N→1 (with the same return)
    - Example: File Server
      - HDDの調達や管理等の投資をN→1に削減
  - Return: 1→N (with the same investment)
    - Example: Knowledge sharing
      - 共有による効果が大きい（と期待される）知識とは？
        - » 例えば、過去の業務で実際に役立った知識
      - 共有の手間を小さくするには？
        - » 例えば、知識そのものでなく誰が知っているかを共有
- Division
  - サイズを分割した方が開発コストは減る
    - Development Cost ＝ Size$^{\alpha}$ （$\alpha > 1$）
    - （a + b）$^{\alpha}$ ＞ a$^{\alpha}$ + b$^{\alpha}$ （α, a, b ＞1）
  - Cost to divide
    - Architecture & Interface Design
  - Cost Reduction from the Division > Investment to Divide
    - Low Coupling（結合度）, High Cohesion（凝集性）
- Component（部品） ＝ Division ＋ Sharing
  - Physical components: Only production & design can be shared
    - Example: LSI
  - Software components: Component itself can be shared with only digital copy cost
    - Example: Class Library
  - Network components: Component itself can be shared with only communication cost
    - Example: Web Server、Network Printer

Dependency analysis by Lattix
DSM (Dependency Structure Matrix)

2d. MVC-based Web Application Framework

- Web Application based on MVC Architecture



- MVC Architecture
  - ソフトウェアの設計アーキテクチャの一つで、処理の中核を担う「Model」、表示・出力を司る「View」、入力を受け取ってその内容に応じてViewとModelを制御する「Controller」の3要素の組み合わせでシステムを実装する方式。
  - 役割毎に明確に分離することで、開発作業の分業が容易になり、また、仕様変更の影響を受けにくくなる。
- 既存の各種フレームワークについて
  - Ruby on Rails（Ruby）
    - 「設定より規約（CoC：Convention Over Configuration）」により高い生産性を実現する元祖フレームワーク
    - より少ないコードで簡単に開発が可能
  - Symfony（PHP）
    - Ruby on Railsと同等の開発効率を実現
  - Catalyst（Perl）
    - 既存のPerlモジュールを利用可能
    - コンポーネントを自由に組合せて利用可能
  - Grails（Groovy/Java）
    - HibernateやSpring等の既存Javaフレームワークに対するインタフェースを提供

3. どうやって作るかの決め方

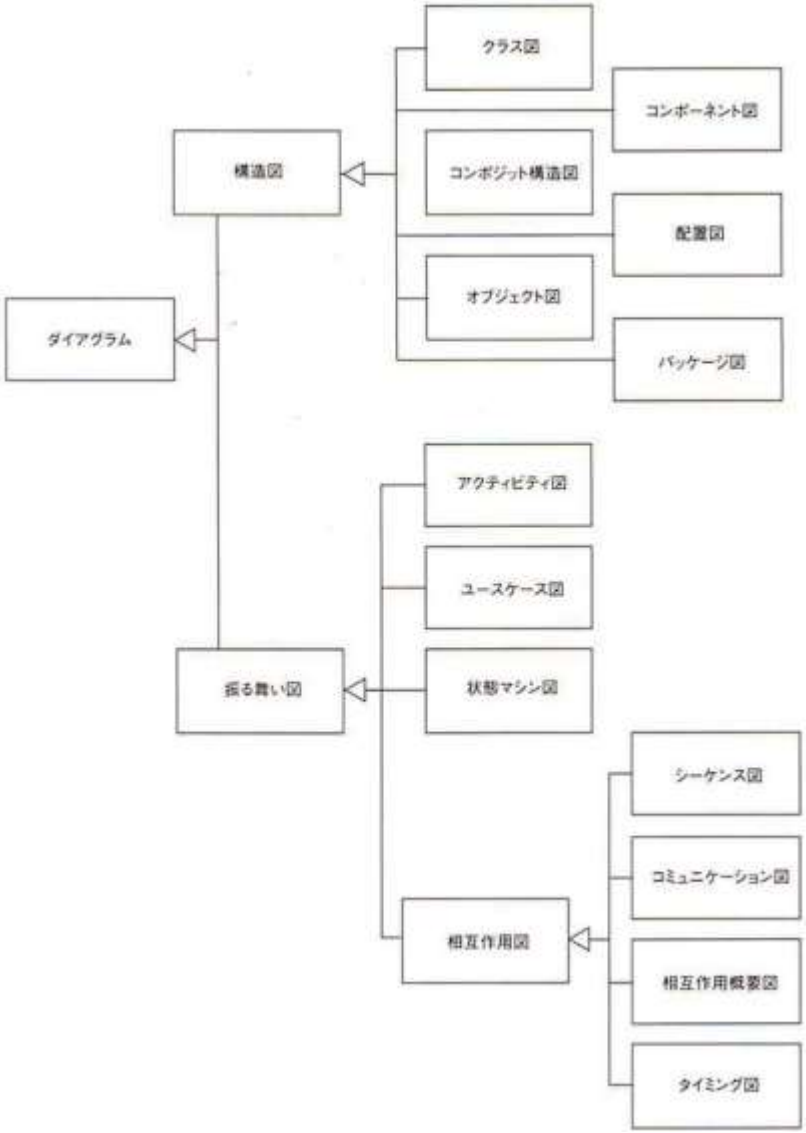
- 開発方法論/開発手法
  - プロセス中心
    - 入力 ⇒ 処理 ⇒ 出力
    - DFD（Data Flow Diagram：データフロー図）
  - データ中心
    - データベース、状態遷移（CRUD）
    - ERD（Entity Relation Diagram：実体関連図）
  - オブジェクト指向
    - オブジェクト ＝ データ（Attribute：属性） ＋ 処理（Behavior：振舞い）
    - UML（Unified Modeling Language）

- 開発プロセス
  - ウォーターフォール
    - 分析(要件定義) ⇒ 設計 ⇒ 実装 ⇒ 試験 ⇒ 導入 ⇒ 運用
    - 明確なマイルストン
  - 反復/段階的開発
    - 考え方
      - Iterative：反復的
      - Incremental：漸増的／差分的
    - 例
      - (R)UP：(Rational) Unified Process　・・・　変化を計画的に管理、大規模向き
      - アジャイル　・・・　変化に開発を適合、小規模向き
        - » XP： eXtreme Programming（Embrace Change：変化を抱擁せよ）
        - » Scrum（朝会とか）

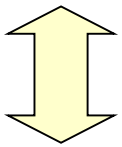
3. How to decide how to make

- Development methodology
  - Process-centric
    - Input ⇒ Process ⇒ Output
    - DFD（Data Flow Diagram：データフロー図）
  - Data-centric
    - Database management systems、State Transision（CRUD）
    - ERD（Entity Relation Diagram：実体関連図）
  - Object-oriented
    - Object ＝ Data（Attribute：属性） ＋ Process（Behavior：振舞い）
    - UML（Unified Modeling Language）
- Development process
  - Waterfall development
    - Analysis (Requirement definition) ⇒ Design ⇒ Implementation ⇒ Test ⇒ Installation ⇒ Maintenance
    - Definite milestones
  - Incremental/Iterative development
    - Plan
      - Iterative：反復的
      - Incremental：漸増的／差分的
    - Example
      - (R)UP：(Rational) Unified Process ・・・ Manage planned changes, Large-size development
      - Agile ・・・ Adapt the development to change, Small-size development
        - » XP： eXtreme Programming（Embrace Change：変化を抱擁せよ）
        - » Scrum（朝会とか）

3a. UML（Unified Modeling Language）



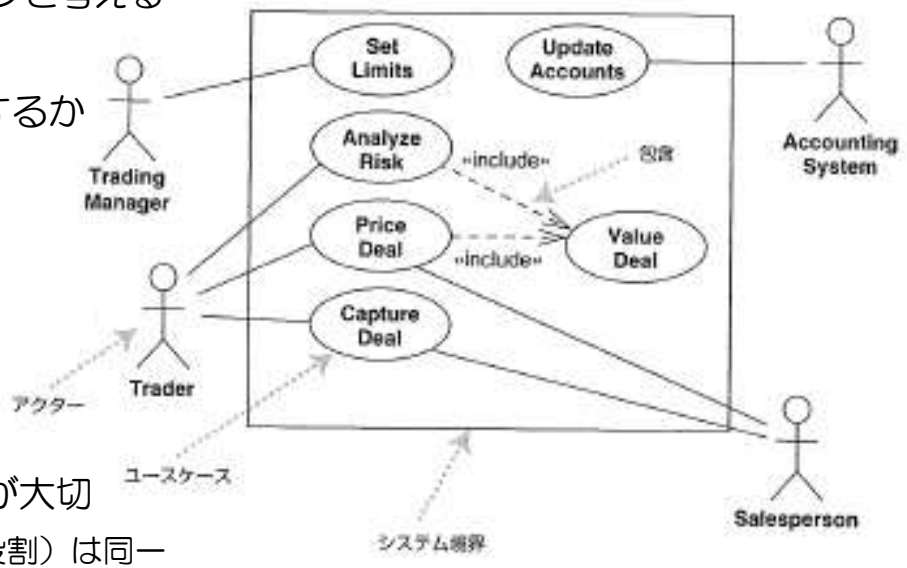
- モデリング言語（Modeling Language）
  - 設計を表現するために方法論で使用される記法
    - 現在はOMG標準
- 設計におけるUMLの活用
  - ユースケース（use case）図
  - クラス図
    - 抽象化
  - 相互作用図
    - クラス間のコラボレーション
  - パッケージ図
    - 大きなシステムのロードマップの構築
- UML以外に活用できるもの
  - CRCカード
  - パターン



- 開発プロセス（Development Process）
  - 設計を行うための手順

3b．Use case diagram

- ユースケース(use case)
  - ユーザの一般的な目的に照らして結び付けられた一群のシナリオ
    - 例）「製品を購入する」という単一のユースケース
      - 購入成功と認証失敗という２つのシナリオを持つ
  - ユースケースを簡単に作成する１つの方法
    - 主な手順を番号付きで書き出して主シナリオとする
    - その他の手順を主シナリオからのバリエーションと考える
  - ユースケースをどのように分割するか
    - ユースケースをどのレベルまで詳細に検討するか
      - リスクが高いほど、より詳しく考察する
      - 実装に必要な場合、反復中に随時詳細を付加



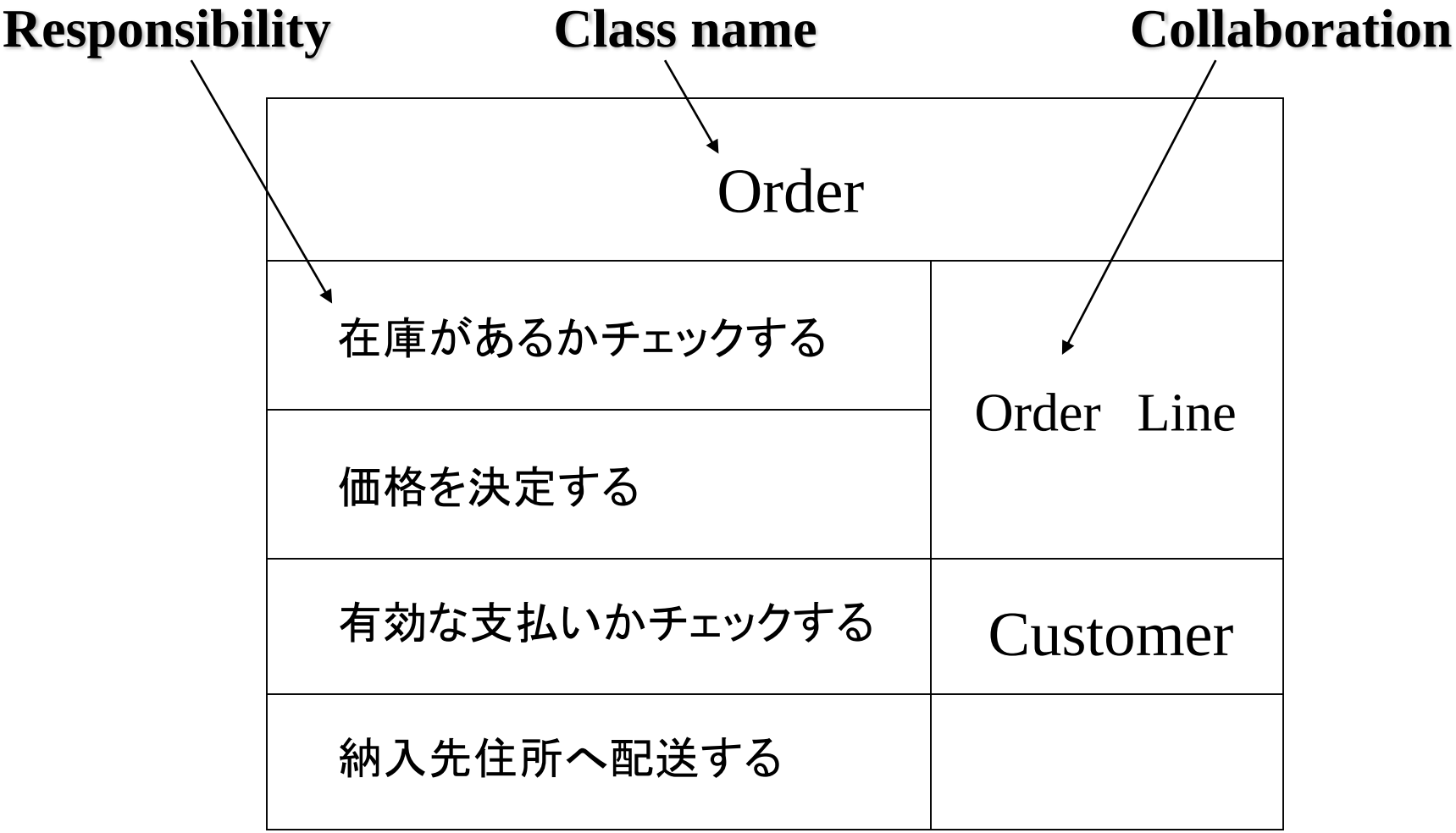
- アクター(actor)
  - システムに対してユーザが果たす役割
    - 個人や役職よりもそのロールを考えることが大切
      - 例）トレーダーが何人いてもそのロール（役割）は同一
    - １人ユーザが複数のロールを持つ場合もある
  - アクターのリストに着目し、それぞれのアクターに必要なユースケースを割り出す

3c. CRC Card

- Class-Responsibility-Collaboration Card
  - Write responsibilities to the cards
    - Responsibility
      - Describes the purpose of the class (in a few lines of sentence) instead of the description of fine data and processes

- Advantage of the CRC card
  - To activate the discussion between developers
    - UML diagrams take a long time to express an alternative idea
  - Useful to examine the interaction between classes

3d. CRC Card Sample



3e．Design by Contract（契約による設計）

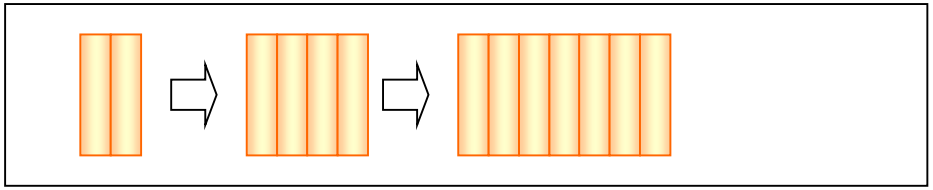
- 3種類の表明(Assertion)
  - 事前条件(pre-condition)
  - 事後条件(post-condition)
  - 不変式(invariant)
- 事前条件と事後条件は操作に適用
  - 操作の事前条件、事後条件は操作の定義内に文書化
  - 例外(exception)とは
    - 事前条件が満たされた状態で操作が呼び出されても、事後条件が満たされた状態でもどることができない場合に発生
- 不変式はクラスに適用
  - 不変式はクラスのインスタンス全てに対して常に真となる
  - 不変式は、あるクラスのすべてのパブリック操作に関連づけられた事前条件と事後条件に追加される

3f．Assertion（表明）

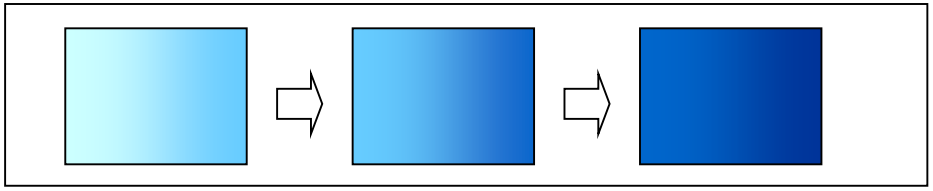
- 表明とは「決して偽にならないはずの命題」
  - 表明がチェックされるのはデバッグの時だけ
    - Eiffelは言語の一部として表明をサポートしている
- 表明ではサブクラスの責務を増やすことだけが可能
  - クラスの不変式と操作の事後条件は全てのサブクラスに適用され、サブクラスではこれらの表明を強化できるが緩めることはできない
  - 逆に、事前条件はサブクラスで強化できないが緩めることはできる
- 表明は安全なサブクラス化に役立つ
  - ポリモルフィズムでは、サブクラスの操作をスーパクラスの操作と矛盾するように再定義できてしまうという危険があるが、表明によりこれを避けられる

3g. Incremental/Iterative development

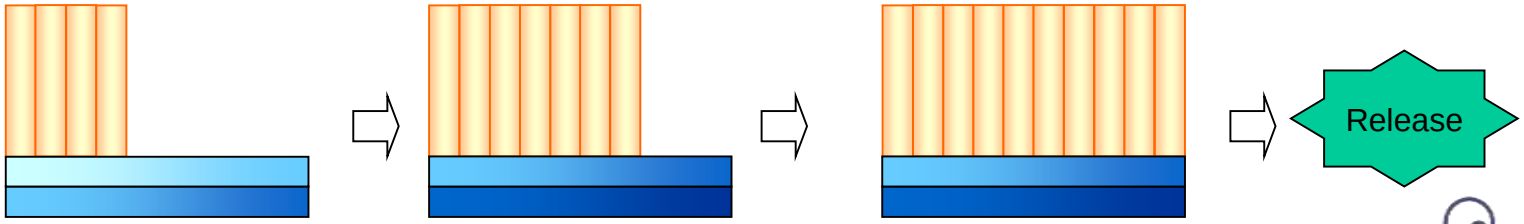
- Incremental development（漸増型）
  - 新規の増分（開発部分）を積み上げていく方法
    - 繰り返しの単位で対象のソフトウェア構造が異なるものや、依存関係のないものに適している
    - 繰り返しの単位の独立性が保てるので分割しやすい



- Iterative development（反復型）
  - ソフトウェアの全体／一部分について、最初は薄く作り少しずつ肉付けしていく方法
    - 重要かつ複雑な部分について、徐々に確認しながら肉付けし中身を濃くしていける

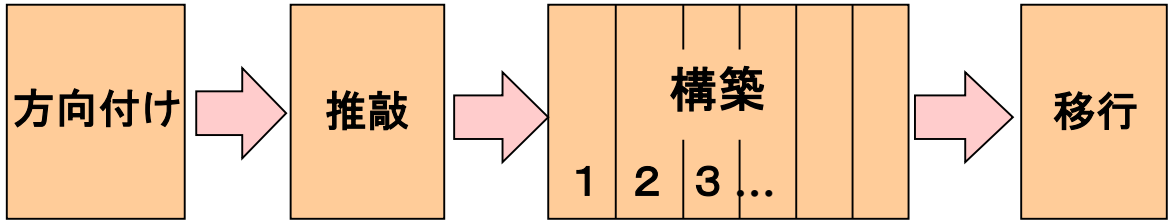


- Combine Incremental & Iterative development

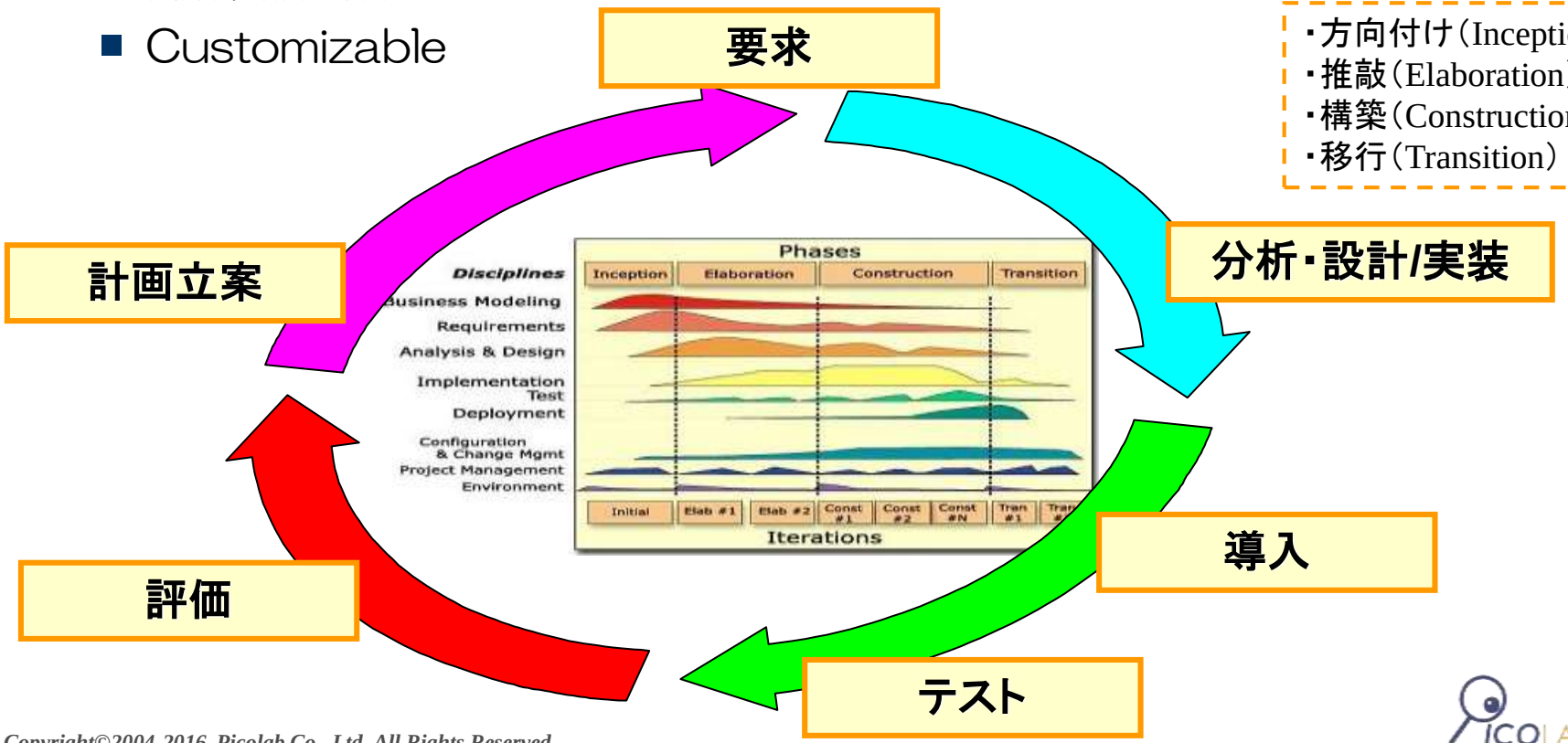


3h. RUP Overview

- RUP（Rational Unified Process）／UP（Unified Process）
  - Iterative
  - Use case driven
  - Architecture centric
  - Risk driven
  - Customizable

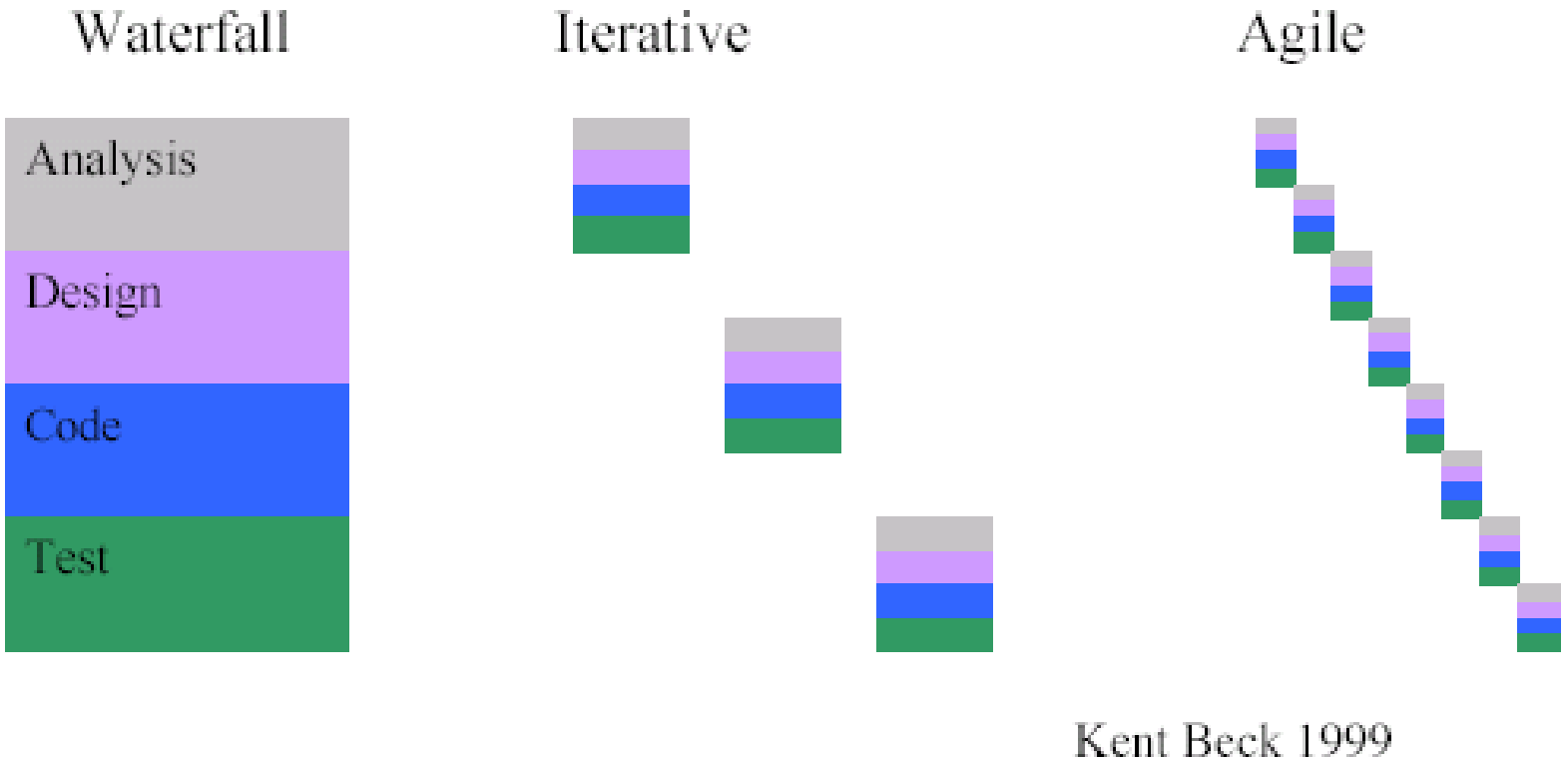


・方向付け（Inception）
・推敲（Elaboration）
・構築（Construction）
・移行（Transition）



3ｉ．Agile Development Overview

- Extreme Programming（XP）
- フィーチャ駆動開発（FDD：Feature Driven Development）
  - http://www.atmarkit.co.jp/farc/rensai/mda_sikumi02/mda_sikumi02.html
- モデルベース開発（MBD：Model Based Development）
  - http://monoist.atmarkit.co.jp/mn/articles/0903/27/news109.html



3ｊ．Agile Development

- アジャイル・マニフェスト
  - 変化を受け入れる（変化を味方につける）
  - 価値あるソフトウェアを最初から顧客に継続的に引渡す
  - 動いているソフトウェアが進捗の尺度
  - 単純さ（作業せずに済む量を最大化する技量）が本質
  - 顧客と開発者は一緒に働く
  - 面と向かって話をする
  - 定期的に振返り、自分たちのやり方を最適に調整する
    - http://www.agilemanifesto.org/principles.html
    - http://www.agilemanifesto.org/iso/ja/
- アジャイル・アライアンス
  - http://www.agilealliance.org/home
- 生産管理の分野では
  - トヨタ生産方式(Lean Manufacturing)
  - TOC(Theory of Constraint)

3k. XP（eXtreme Programming）

- ４つの価値
  - コミュニケーション、シンプルさ、フィードバック、勇気
- １２のプラクティス
  - 計画ゲーム(The Planning Game)
    - ビジネス優先度と技術的見積により次回リリースの範囲を決め、現実の状況変化に応じて計画を更新
  - 小規模リリース(Small Releases)
    - シンプルなシステムを早期にリリースし、以降、新バージョンを短いサイクルでリリース
  - 比喩(Metaphor)
    - システム全体の機能を示すシンプルな喩え話をメンバーが共有することで全開発をガイド
  - シンプルデザイン(Simple Design)
    - システムを出来る限りシンプルに設計し、余分な複雑さは見つけ次第取り除く
  - テスティング(Testing)
    - プログラマは完全に動く単体テストを継続的に書き、顧客は機能の開発終了を示す機能テストを書く
  - リファクタリング(Refactoring)
    - コミュニケーション/柔軟性改善や単純化のために、システム動作を変えずにシステムを再構成する
  - ペアプログラミング(Pair Programming)
    - ２人のプログラマが一台のマシンでコードを書く
  - 共同所有権(Collective Ownership)
    - 誰が書いたどのコードでも、全てのプログラマが修正できる。
  - 継続的インテグレーション(Continuous Integration)
    - テストを 100% パスするように、一日に何回もシステムのビルド＆テストを繰返す
  - 週40時間(40-Hour Week)
    - 週４０時間以上仕事をしない
  - オンサイト顧客(On-Site Customer)
    - 現実のユーザをチームに加えていつも質問に答えられるようにする
  - コーディング標準(Coding Standards)
    - コーディング標準に従って全コードを書く

4. Development plan & Progress management

- WBS（Work Breakdown Structure）による開発計画作り
  - 成果物単位で段階的詳細化（問題分割） ⇒ 成果物一覧/目次
    - 各成果物を得る活動が Work（作業）
    - 最終成果物を最短で得る等の前提で各成果物のマイルストン（完成目標）決定
      - 各成果物の担当者（の最大稼動）、見積作業量、成果物間の依存関係等が制約条件
      - その作業の遅延が全体の遅延に影響するような作業集合が「クリティカルパス」
    - 遅延余裕（バッファ）は各作業に計上せずに、全作業/フェーズの最後に計上

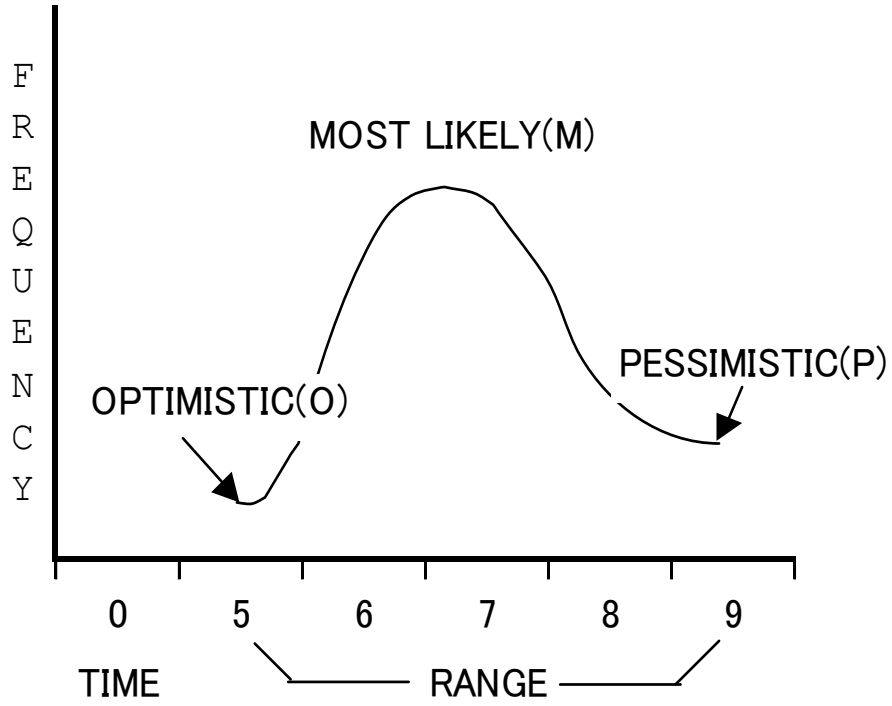
- WBSに基づく進捗管理
  - 進み具合（進捗率）の管理 ⇒ ガントチャート（線票）/イナズマ線
    - 作業段階（準備、着手、作業中、成果物作成中、作成済、確認済等）
      - 各段階に対応した基本進捗率とその段階の**残作業時間**に応じた調整進捗率との組合せ
    - 出来高管理（EVM：Earned Value Management）
      - http://www.atmarkit.co.jp/aig/04biz/evms.html

- WBS管理のツール
  - Excel、MS-Project
  - GanttProject
    - http://sourceforge.jp/projects/sfnet_ganttproject/
  - Redmineでもちょっとしたガントチャートは作れる

4a. Uncertainty of estimates (Estimation risk)

- タスクの作業期間の予測には、例えば、以下の式を使う
  - 楽観的な見積り
  - 悲観的な見積り
  - 適切と思われる見積り

⇒ 予測される期間を算出



$t_e=(O + 4M + P)/6$

Where:

t_e = expected time

O = optimistic estimate

M = most likely estimate

P = pessimistic estimate

4b. Estimation of the duration of the project

- プロジェクト全体の期間を、次のように予測できたとする

CRITICAL ACTIVITY	OPTIMISTIC	MOST LIKELY	PESSIMISTIC	EXPECTED
TASK A	4	5	8	5.33
TASK D	10	12	16	12.33
TASK E	9	12	14	11.83
TASK F	3	4	5	4.00
TASK G	1	1	3	1.33
		34		**34.83**

Scheduled Time = 34 days

Expected Time = 34.83 days

Accuracy of the predicted value?

4c. Risk for the project delay

- How to calculate the accuracy of the predicted value

CRITICAL ACTIVITY	OPTIMISTIC	PESSIMISTIC	RANGE(P-O)	S.D.=σ (RANGE/6)	VARIANCE=σ^2
TASK A	4	8	4	0.667	0.444
TASK D	10	16	6	1.000	1.000
TASK E	9	14	5	0.833	0.694
TASK F	3	5	2	0.333	0.111
TASK G	1	3	2	0.333	0.111
					2.361

↓ 平方根

Project variance(σ^2) = 2.361　　Project S.D.(σ) = 1.537 ← 1.537

※S.D. = STANDARD DEVIATION(標準偏差)
※VARIANCE(分散)

Z = (Ts-Te)/σ だから、Z = (34-34.83)/1.537 = -0.540

こいつを、正規分布表から逆引きして、、、0.7054 と。

1-0.7054 = 0.2946だから、34日以内でプロジェクトが完了する確率は30%！

5. Project management

- Project management
  - プロジェクト管理者（PM：Project Manager）の責任範囲
    - 人（リソース）・物（成果物）・金（費用対効果）
      - ROI（Return On Investment）＝ 利益／投資
    - QCD（Quality：品質、Cost：費用、Delivery：納期）
      - 時は金なり（開発速度≒レイテンシ、開発効率≒スループット）
  - PMBOK（the Project Management Body Of Knowledge）
    - PMI（Project Management Institute）がまとめた管理知識体系
    - PMP（Project Management Professional）資格試験
      - http://www.pmi-japan.org/pmp_license/

- Issue management etc.
  - 課題（遅延リスク等）に対する作業の駆動・管理
    - 新たな課題の識別（粒度は様々？）
    - 課題の担当、期限、対処状況等の設定、更新
  - Progress management ＋ Issue management
    - 定例会議（議事録）
    - 日次/週次報告
  - Issue management tools
    - 課題管理表：Excel／Google Docs
    - Trac, Redmine, Bugzilla, Jira
      - 「チケット」ベース
      - 不具合管理等には向いている
      - 様々な課題をみんなで継続的に管理するにはそれなりのコストがかかる

5a. Project management

- 開発におけるプロジェクト管理で特に注意すべき点
  - 反復型開発
    - 要件は変化する
    - スケジュールは変化する
    - プロダクトは変化する

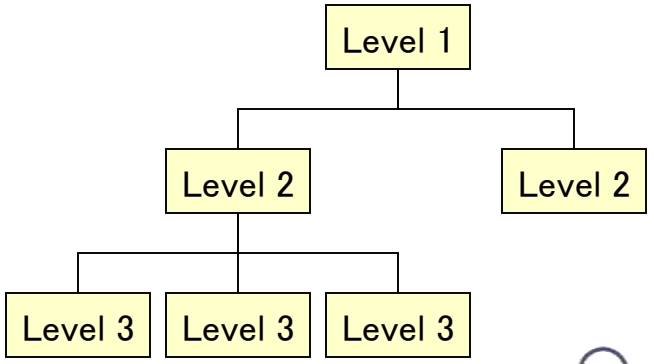
    →変化を管理する
  - 再利用性/拡張性/品質等の管理
    - 非機能要件は明示的に把握しにくい
    - スキル面のリスクも無視できない

    →非機能要件/リスクを管理する
  - フレームワークやコンポーネントの利用
    - アーキテクチャの妥当性は評価しにくい
    - アーキテクチャが不安定だと開発効率は低下

    →アーキテクチャを管理する

5b．Management of the change

- 要件とその変更を管理する
  - 機能的にはユースケース
  - 非機能的には様々（性能、信頼性、拡張性、保守性等）
    →最初から全ての要件は見えていない
    →システムが見えると要件との差分も見えてくる
- WBSとその変更を管理する
  - プロジェクトの成功に必要なアクションステップを明確にする
  - 作業を洗い出すのではなく、成果物（Deliverables）を詳細化
    →スケジュール、責任分担、予算/費用、組織、リスク分析等
    →反復計画およびその実績
- 成果物とその変更を管理する
  - 成果物の電子的な変更管理
    →動くソフトウェア成果物
    →回帰テスト



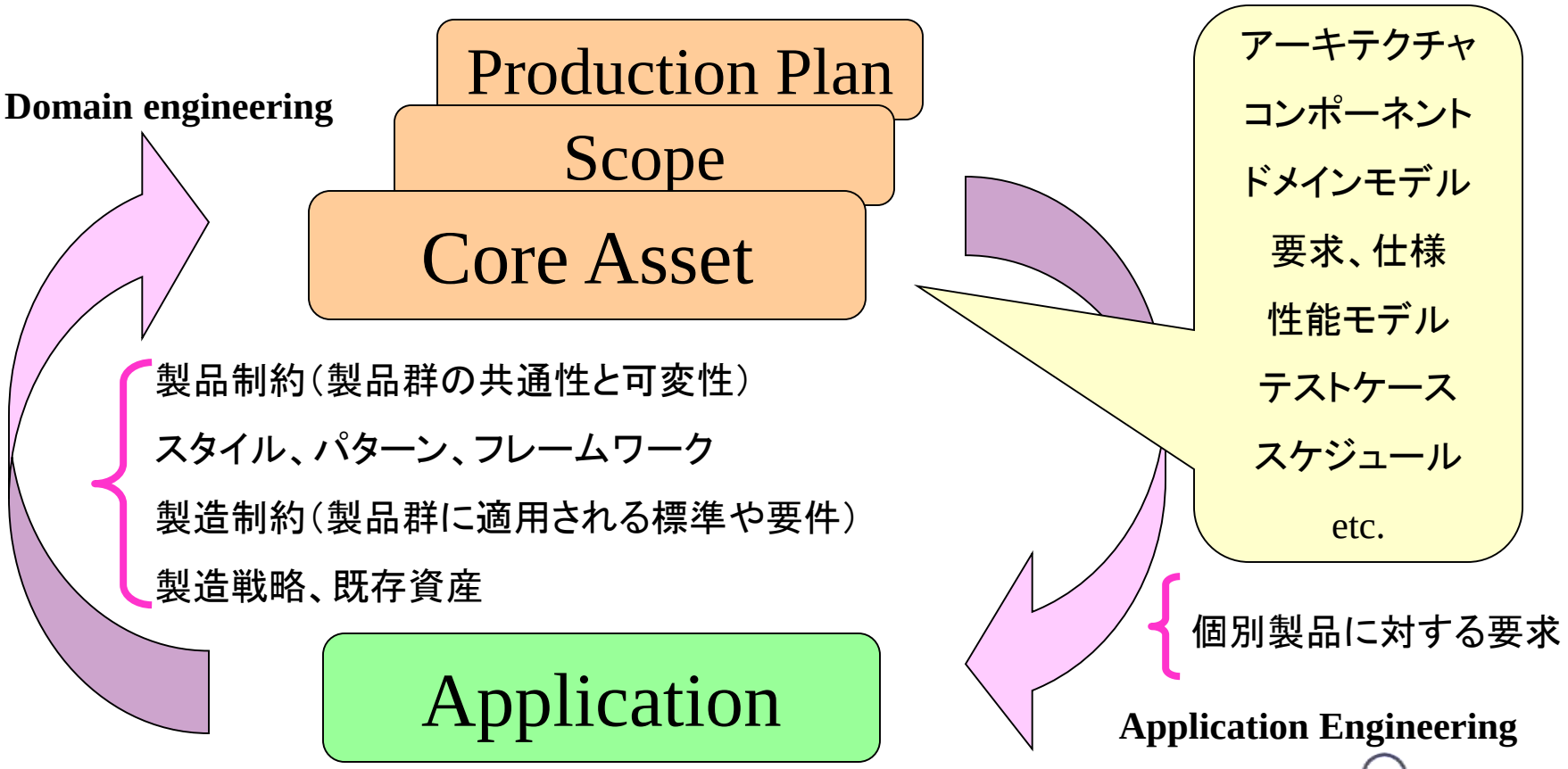
5c. Management of the risk & non-functional requirements

- オブジェクト指向開発だからといって
  - 誰でもQCD向上（高品質、低コスト、短期開発）って訳じゃない

- リスクを管理する
  - 要求リスク
    - ・非機能要件を管理するには、目に見える形（動く形）にすればよい
      - プロジェクト・メトリックス
      - テストケース
  - スキルリスク
    - ・メンタリング(mentoring)
      - 経験を積んだ開発者が一緒にプロジェクトに長期間従事する
  - 技術リスク
    - ・プロトタイピング（コンポーネントの組合せの検証等）
    - ・発生し得る問題にどう対処するか（リスクヘッジ）を考えておく
  - 政治的リスク

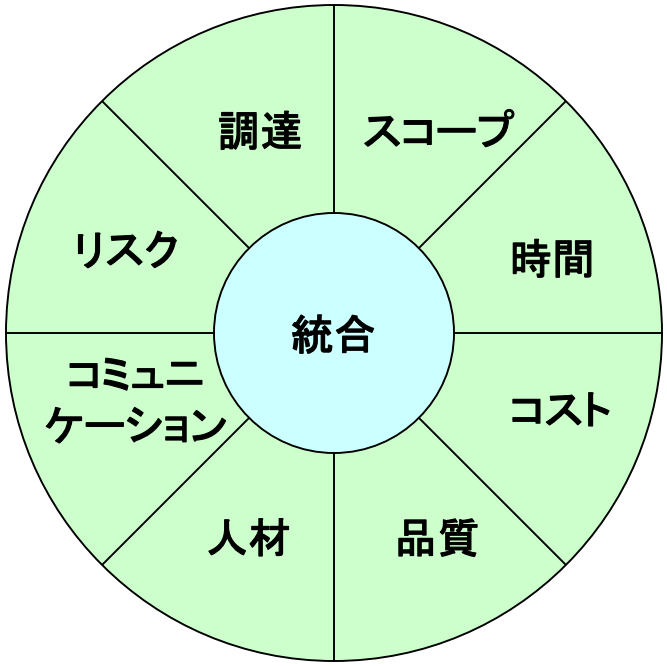
5d. Management of the architecture

- Software Product Line
  - ドメインエンジニアリングによるコア資産開発（資産形成）
  - アプリケーションエンジニアリングによる製品開発（資産運用）
    - http://www.sei.cmu.edu/productlines/



5e. PMBOK Overview

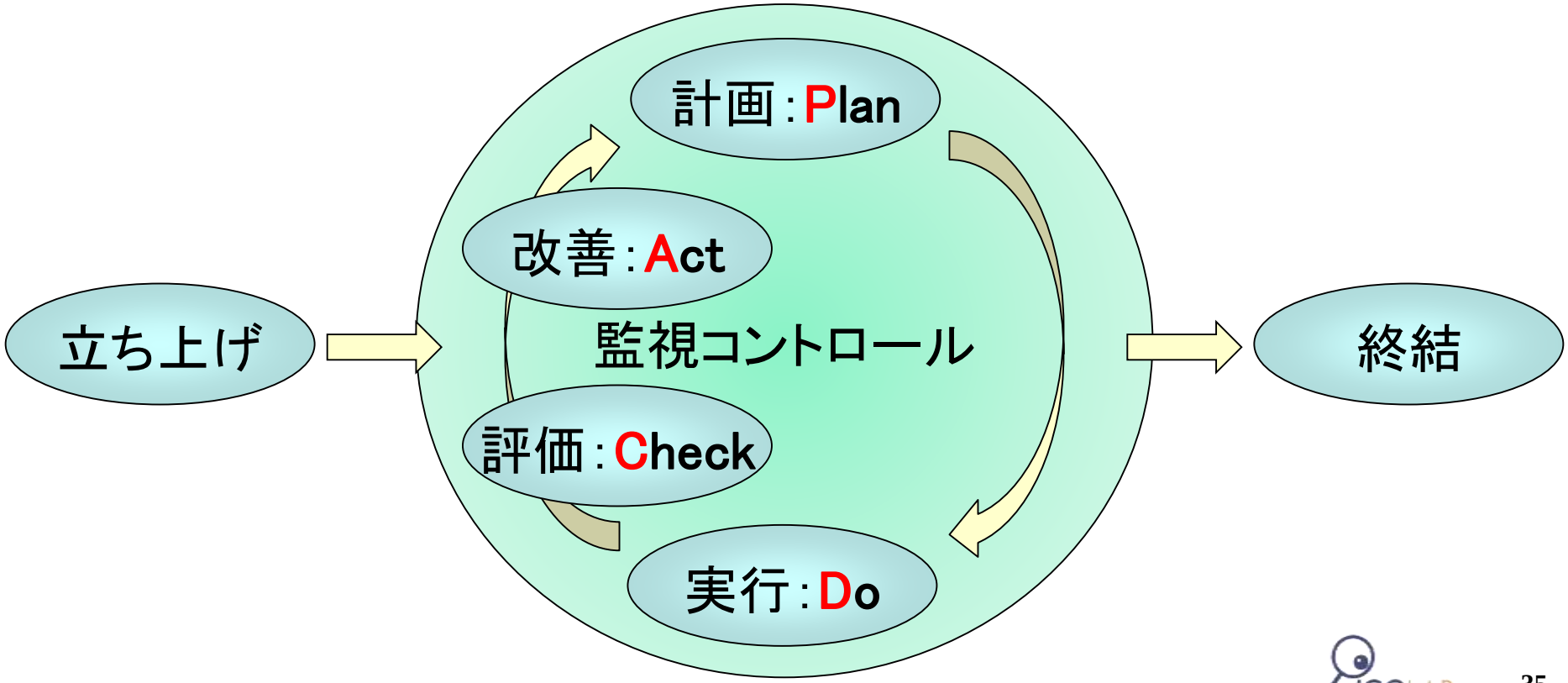
- A Guide to the Project Management Body of Knowledge
  - プロジェクト管理の知識体系
  - PMI(Project Management Institute)による
    - http://www.pmi.org/



- プロジェクト管理を９分野に分類
  - スコープ管理
  - 時間管理
  - コスト管理
  - 品質管理
  - 人材管理
  - コミュニケーション管理
  - リスク管理
  - 調達管理
  - 統合管理
- 各分野ごとに次の４つを体系化
  - 主要なプロセス
  - プロセスの必要情報（入力）
  - プロセスの利用ツール/テクニック
  - プロセスの成果物（出力）

5f. Processes of PMBOK

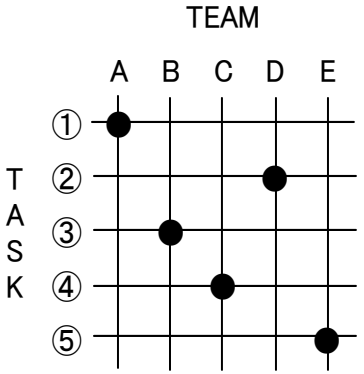
- 「立ち上げ→計画→実行→終結」が管理プロセスの流れ
  - 「計画→実行→評価→改善」のサイクルが変更管理
  - 「計画」と「実際」の差を早期に予測/把握し、計画/遂行を調整
    - 例えば、マイルストンはその差を予測/把握するための検証ポイント

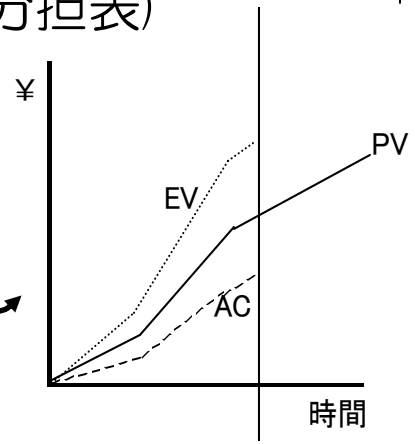


5g. The Seven Icons (Management documents)

プロジェクトの計画・コントロールに際して
キーとなるチャートや図表などのツール

1. WBS(Work Breakdown Structure)
2. CPM(Critical Path Method)
3. BAR CHART
4. “T” CHART(or TRM or RAM：役割分担表)
5. BUDGET THE BARS
6. BASELINE
7. EARNED VALUE





5h．Capability Maturity Model

- ソフトウエアプロセスの成熟度
  - CMM（Capability Maturity Model）
    - レベル１：初期（Initial）
    - レベル２：反復できる（Repeatable）：要件、計画、進捗、外注、品質、構成
    - レベル３：定義された（Defined）
    - レベル４：管理された（Managed）
    - レベル５：最適化する（Optimizing）
  - CMMI（CMM Integrated）：SW/SE/IPPD/SS、連続/段階モデル
- プロジェクトマネジメントの成熟度
  - MicroFrame Technologies & Project Management Technogiesモデル
    - レベル１：場当たり的（Ad-Hoc）
    - レベル２：概略定義された（Abbreviated）
    - レベル３：組織化された（Organized）
    - レベル４：管理された（Managed）
    - レベル５：適応できる（Adaptive）
  - Harold Karzner のPMMM（Project Management Maturity Model）
    - 共通言語，共通プロセス，共通の方法論，ベンチマーキング，継続的改善
  - PMIのOPM3（Organizational Project Management Maturity Model）

6. Development environment / Development support tools

- Integrated development environment（IDE）
  - Eclipse etc.（＋ 各種プラグイン）
    - http://www.eclipse.org/
  - Visual Studio（＋ Team Foundation Server/Visual Studio Online）
    - http://www.visualstudio.com/
  - Rational Team Concert（Jazz）
    - http://www-06.ibm.com/software/jp/rational/products/scm/rtc/
    - http://jazz.net/

- Development support tools
  - 構成管理ツール（ソースコードリポジトリ）
    - GitHub、Subversion、CVS、Visual Source Safe 等
  - テスト支援ツール
    - 単体テストフレームワーク（xUnit）
      - http://ja.wikipedia.org/wiki/XUnit
    - Webアプリケーションの機能テスト、負荷/性能テスト
      - Selenium, Apache Jmeter, Oracle Load Testing, HP LoadRunner等
  - 設計支援ツール
    - UMLモデリングツール
      - astah* community（旧JUDE）
        - » http://astah.change-vision.com/ja/product/astah-community.html
      - Enterprise Architect（EA）
        - » http://www.sparxsystems.jp/ea.htm

6a. Development support tools

- 構成管理（ソースコード管理）ツール
  - gitHub, Subversion, CVS, Visual Source Safe 等
- 課題管理（Issue/Ticket tracking）ツール
  - Trac（http://trac.edgewall.org/）
  - Redmine（http://redmine.jp/）
  - JIRA(＋ GreenHopper), Mantis, Backlog 等
- 継続的インテグレーション（CI）ツール
  - Jenkins（http://jenkins-ci.org/）：旧Hudson
  - CruiseControl, Buildbot, Bamboo 等
- テスト支援ツール
  - 単体テストフレームワーク（xUnit）
    - http://ja.wikipedia.org/wiki/XUnit
  - Webアプリケーションの機能テスト、負荷/性能テスト
    - Selenium, Apache Jmeter, Oracle Load Testing, HP LoadRunner等

6b. Version control（構成管理）

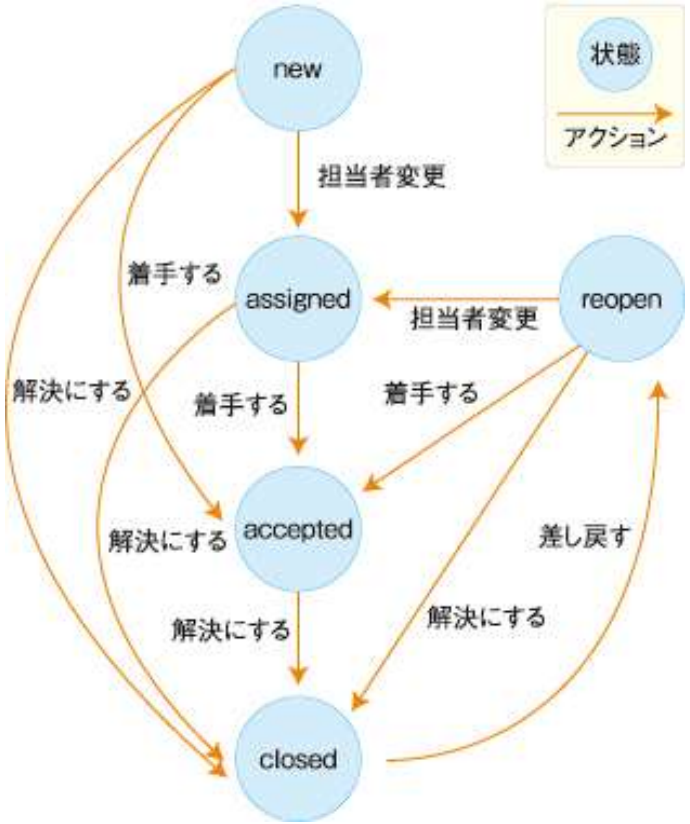
- Version Control System（VCS）
  - 集中型VCS
    - RCS
      - 自分の個々のファイルをローカルで管理
    - CVS
      - グループの複数ファイルをネットワークを介して管理
    - Subversion
      - ファイルでなくプロジェクト・リポジトリ単位で管理
  - 分散型VCS
    - Git
    - Mercurial
    - Bazaar
- ファイルのバージョン・リビジョン（変更履歴）を管理
  - いつ、だれが、どこを、何のために、どう変更したかを管理
  - ロールバック、ブランチ、マージ等が可能

6c．Issue/Ticket tracking（課題管理）

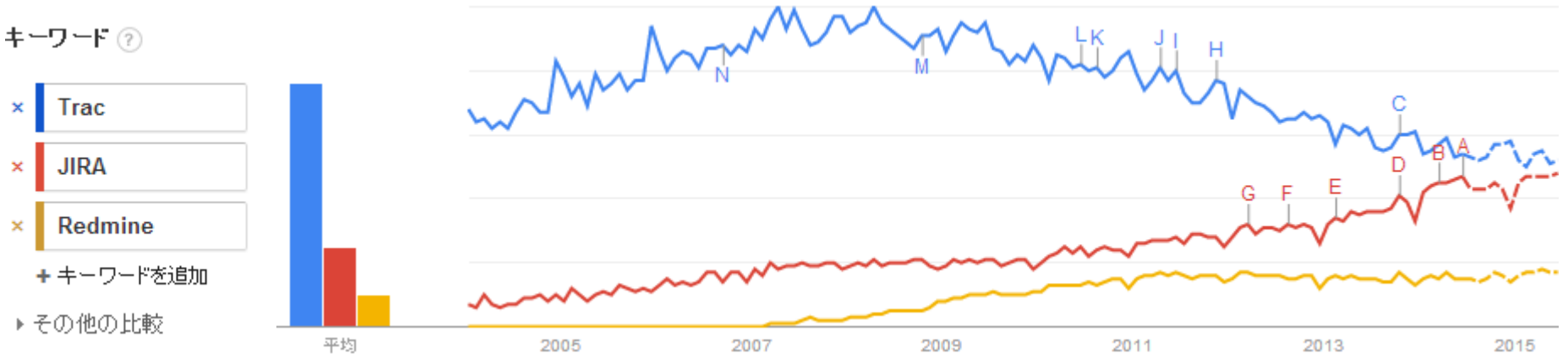
- 課題（Issue）とその解決プロセス（タスク）とは？
  - バグ（不具合）
    - 発見/識別
      - 再現できなければバグじゃない？
      - バグ、それとも、仕様？
    - 原因調査、解決策立案
      - 仕様の詳細化・変更等が必要になることも
    - ソースコード（テストコード、ドキュメント）変更
      - どのバージョン（パッチ等）で変更＆リリースする？
    - ビルド、テスト
      - ビルドが通らないんだけど…
      - どこまでのテストケースを確認？（バグってなかった所が変になった？）
    - 解決完了確認
      - どのリリースで解決してるのを誰が確認して完了？
  - 要望/要求/要件、質問
    - 大雑把な要求もあれば、細かい要求も
      - この要件って、元々はどの要望から来てたっけ？
      - 別の要件が満たされないと、この要件も満たせないよね…

6c1. Issue/Ticket tracking tools

- 課題の情報/対応状態をTicketとして管理
  - Ticket登録
    - 概要、説明、分類、担当者、優先度、コンポーネント、終了予定日
  - コメント追加＆ステータス変更
    - マイルストーンやバージョンの設定
    - 担当者やステータスの設定
- 関連情報との連携
  - Wikiの活用
    - 手順書、周知事項、議事録、仕様
  - 構成管理ツールとの連携
    - チケット→ソース（リビジョン/ファイル/行）
    - コミット（コメント）→チケット（番号）
  - 継続的インテグレーションツールとの連携
    - チケット→関連するジョブやビルド
    - ジョブやビルド→関連するチケット
  - 進捗管理
    - ガントチャート
    - バーンダウンチャート（作業残量の見える化）



6c2. Trac vs Redmine



- Trac
  - Linux系OSに標準インストール
  - Pythonベースで導入も容易
    - Windows向け：TracLightning
    - Linux向け：Kanon Conductor
  - 豊富なプラグインによる機能拡張
  - レポートのカスタマイズ機能
  - 活発なコミュニティ活動＆情報量大
- Redmine
  - 複数プロジェクトに対応
    - チケットの階層関係にも対応
  - チケット属性で集計が容易
    - チャート等で集計結果を可視化
  - Agile開発で使いやすい
    - バージョン/マイルストーンがリリース相当
    - ロードマップがイテレーション計画相当
      （マイルストーン＝リリースバージョン）

	開発チーム	イテレーション	ストーリーカード	タスクカード
Trac	コンポーネント	マイルストーン	親チケット	子チケット
Redmine	複数プロジェクト	バージョン	親チケット	子チケット

6c3. Issue/Ticket tracking notes

- 課題（Issue）の粒度、分類基準
  - 粒度が細かくなりすぎると管理コストも増大する
    - そもそもコードと関係ない課題は細かく管理してもあまり意味がない
  - 統一は難しいが、運用指針は必要
    - いくつかの活用パターンを具体的な事例等で用意しておく
- チケットのワークフロー管理（状態更新、担当者アサイン）
  - 実際の作業状況をチケットに反映させるには手間（コスト）がかかる
    - 最低限のコストでチケットを更新できるように運用を工夫する
    - 登録課題情報をレポート出力に直結させ更新の動機を高める
  - 課題管理担当者（プロジェクト管理者）を決めないと放置されがち
    - 定期的に関係者の打合せ等を設けて確認する
    - 優先度等を明示して「まず対応すべき課題」を明確化する
- 開発文書と課題管理システムのWikiとの使い分け
  - 仕様書や設計書、説明書等の開発文書は成果物の一部
    - 開発文書はコードと同様に管理対象として扱い、Wikiでは書かない
    - 課題としてWikiに書き、開発文書に反映して課題closeもOK

6d. Continuous Integration（CI）

- XP（eXtreme Programming）のプラクティスの一つ
  - いつでも最新のコードでビルドしてテスト（できるように）する
  - 単独でメリットがあり、XPの他のプラクティスと独立に実施可能
- 継続的インテグレーションのメリット
  - エラーやバグは組込まれてすぐ発見され、他人のデバッグ作業に影響しない
    - デバッグ作業で他人のバグに気付かずに無駄な時間を費やさなくて済む
  - コストに見合うだけの十分な数のバグの早期捕捉が重要
    - そのためには、最新(過去)コード取得、ビルド、テストの自動化が必要
- 成功したビルドとは？
  - 自動的なコード取得、ビルド、テストがすべてOKだったビルド
  - ローカルビルドOK→チェックイン（必要に応じ順番に）→マスタービルド
- シングルソースポイント
  - ビルドに必要な最新の全ソースファイルをコマンド一発で呼出せる
    - 全ソースファイルを構成管理システムに入れて管理することが必要
      - 設定ファイル、インストール／デプロイ用スクリプトやテストコードも

6d1. Jenkins

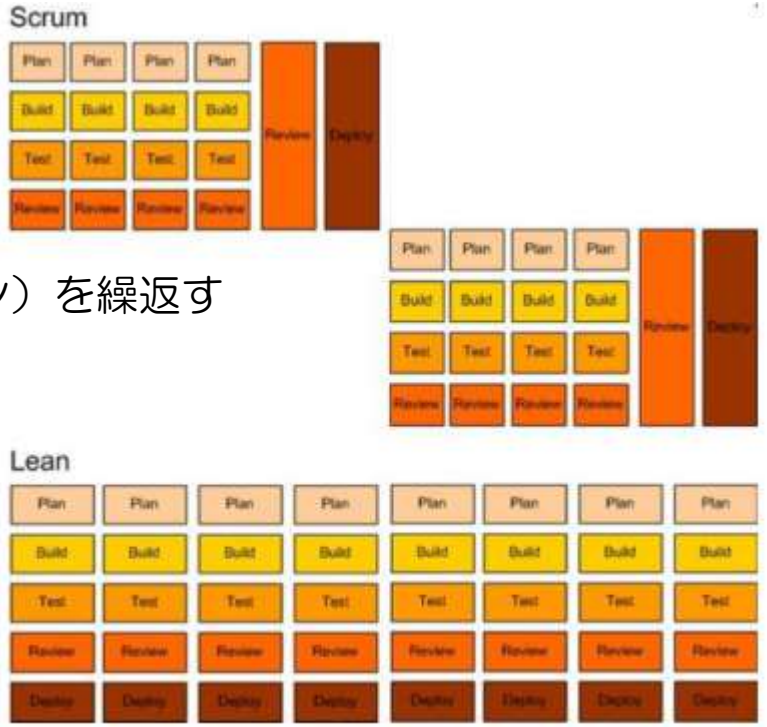
- オープンソースのCIツール
  - リポジトリからチェックアウトしてインテグレーション＆結果通知
    - 静的解析、コンパイル、デプロイ、単体/結合テスト/システムテスト
  - インストール/設定/運用が簡単
    - 必要なツールをJenkinsをビルドするサーバ上に自動でインストール
      - JDK、Ant、Maven、Groovy、Gradle、MongoDB
  - マスター/スレーブ構成
    - スレーブノードを増やしてスケールアウト（余っているPCを有効活用）
  - ブラウザから設定/確認ができる
- プラグインで様々なビルドに対応
  - 構成管理ツール、課題管理ツール、ビルドツール、通知/レポート/UIなどに関する機能がプラグインで提供されている
    - CVS、SVN、Git、Mercurial、Bazaarなどの構成管理ツールと連携

6e. Scrum development with Redmine

- @ITの記事『もし女子高生がRedmineでスクラム開発をしたら』より
  - スクラム開発について
    - 1）「ストーリー」で何を作るかまとめよう
      - http://www.atmarkit.co.jp/ait/articles/1111/14/news132.html
    - 2）手法とツール：「スプリント」と「かんばん」
      - http://www.atmarkit.co.jp/ait/articles/1112/01/news146.html
  - Redmineを使う
    - 3）ALMinium（Redmine＋Backlogs）を使う
      - http://www.atmarkit.co.jp/ait/articles/1112/26/news119.html
    - 4）Redmineでスクラム実践
      - http://www.atmarkit.co.jp/ait/articles/1203/28/news127.html
  - Gitを使う
    - 5）Gitの使い方“超”入門
      - http://www.atmarkit.co.jp/ait/articles/1207/11/news131.html
    - 6）Redmine×Gitでチケット駆動開発
      - http://www.atmarkit.co.jp/ait/articles/1209/19/news140.html
  - Jenkinsを使う
    - 7）Redmine×Jenkins
      - http://www.atmarkit.co.jp/ait/articles/1305/16/news011.html

6e1. What is Scrum?

- 特徴（軽量、理解容易、習得困難）
  - 自己組織化されたチーム
  - 4週間以内の「スプリント」（イテレーション）を繰返す
  - 要求事項は「プロダクトバックログ」で補足
  - 特定の技術的なプラクティスは定めない
- 5つの思想
  - 集中(Focus)
  - コミット(Commitment)
  - オープン(Openness)
  - 敬意(Respect)
  - 勇気(Courage)
- 3つのロール
  - プロダクトオーナー（顧客）、スクラムマスター（上司）、チーム（メンバー）
  - ステークホルダー（利用者等）
- 4つのミーティング
  - スプリント計画（ゴール＋バックログ）、デイリースクラム（15分朝会）、スプリントレビュー、ふりかえり（Keep、Problem、Try）
- 3つの道具
  - プロダクトバックログ（ストーリー：ポイント）、スプリントバックログ（タスク/かんばん：残時間）、バーンダウンチャート
  - Done（受入れ条件）の定義



出典：http://www.ryuzee.com/

プロダクトオーナー
製品に対して責任をもち機能に優先順位を付ける

スクラムマスター
スクラムプロセスがうまくいくようにする。
外部からチームを守る

チーム (7±2人)
プロダクトの開発を行う。
製品の成功に向けて最大限の努力をコミットする

ステークホルダー
製品の利用者、出資者、管理職などの利害関係者。鶏と称す

プロダクトバックログ
製品の機能をストーリー形式で記載
プロダクトオーナーが優先順位を付け、プランニングポーカーで相対見積もり。
項目の追加はいつでも自由だが実施有無や優先順位はPOが決める。

デイリースクラム
毎日チームが以下の３つの質問に答える
・昨日やったこと
・今日やること
・困っていること

バーンダウンチャート
スプリントタスクの「推定残り時間」を更新してグラフにプロットする

Doneの定義
何をもって「完了」とするかを定義したリスト

スプリント計画会議
プロダクトバックログを再度分析・評価し、そのスプリントで開発するプロダクトバックログアイテムを選択する。また選択した項目をタスクにばらす

スプリント
最大4週間までのタイムボックス
各スプリントの長さは同一。この間は外部からの変更を受け入れない

タスク
時間で見積もり

毎日の繰り返し

スプリントバックログ
そのスプリント期間中に行うタスクのリスト

スプリントレビュー
スプリント中の成果である動作するソフトウェアをデモする

ふりかえり
スプリントの中での改善事項を話合い次に繋げる

出荷可能な製品の増分

複数回スプリントを繰り返す

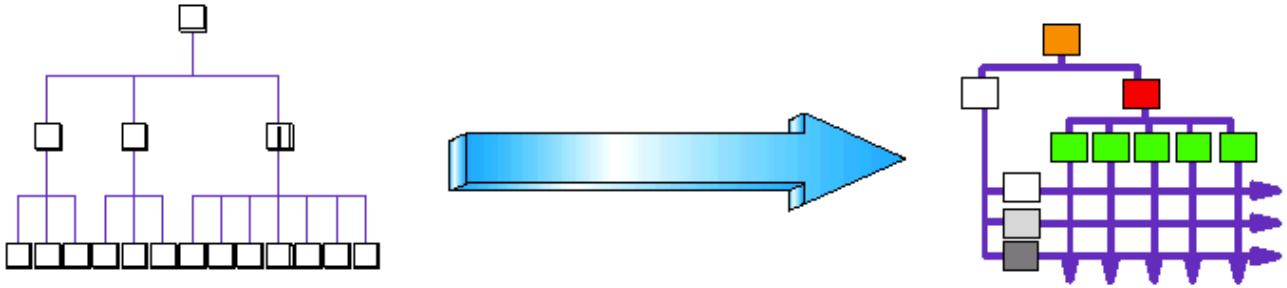
出典：http://www.ryuzee.com/

6f. References

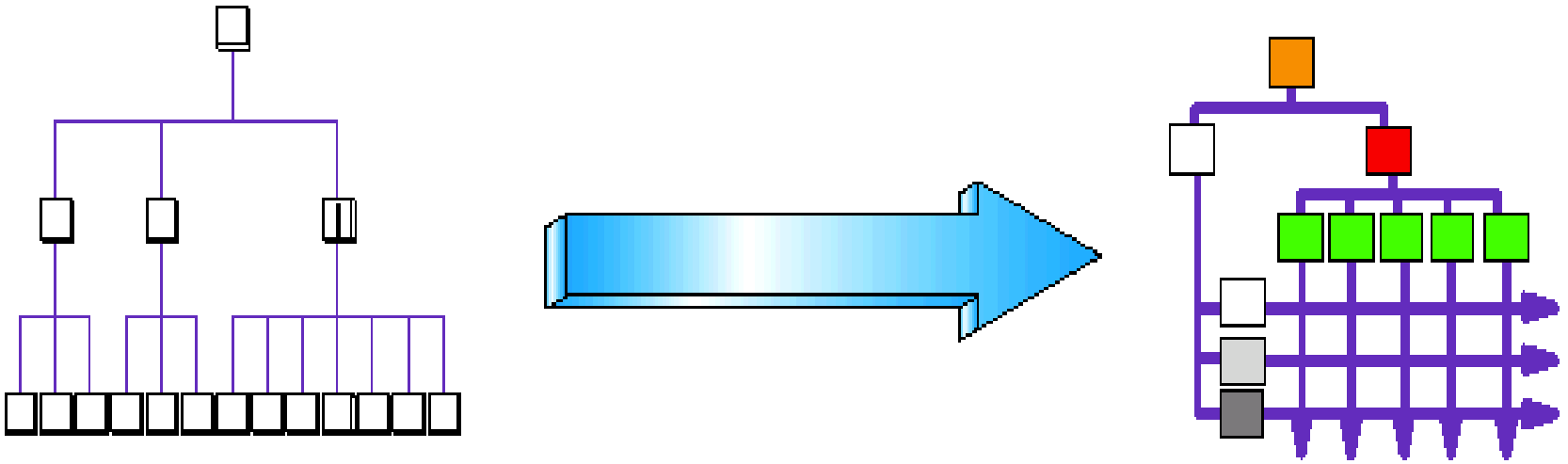
- 課題管理ツール
  - チケット管理システム比較
    - http://confluence.atlassian.jp/display/ATL/Comparison+of+issue+tracking+systems
  - チケット管理システム大決戦！
    - http://www.slideshare.net/SeanOsawa/jira-vs-redmine-vs-trac
  - Redmine 入門
    - http://yohshiy.blog.fc2.com/blog-category-12.html
  - Redmineでアジャイル開発を楽々管理
    - http://www.atmarkit.co.jp/ait/articles/0904/14/news117.html
  - もし女子高生がRedmineでスクラム開発をしたら(ALMinium, Git, Jenkins)
    - http://www.atmarkit.co.jp/ait/articles/1112/26/news119.html
    - http://www.atmarkit.co.jp/ait/articles/1203/28/news127.html
    - http://www.atmarkit.co.jp/ait/articles/1209/19/news140.html
    - http://www.atmarkit.co.jp/ait/articles/1305/16/news011.html
- CIツール
  - 継続的インテグレーションとは？
    - http://objectclub.jp/community/XP-jp/xp_relate/cont-j
    - http://en.wikipedia.org/wiki/Continuous_integration
  - CIツール「Hudson」改め「Jenkins」とは？
    - http://www.atmarkit.co.jp/fjava/rensai4/devtool21/devtool21_1.html
  - Jenkins入門
    - http://www.slideshare.net/kiyOtaka/jenkins-9386883
  - 日本Jenkinsユーザー会
    - http://build-shokunin.org/

7. The key to project success

- Common problems
  - 要件
    - 要件が確定していない、要件が把握しきれていない
  - 変更
    - 変更が頻繁に発生する、成果物に変更を反映しきれない
  - お金（今回は時間）
    - 予算（時間）がない、見積りができない、見積り通りに作業が進まない
  - 組織（今回はチーム）
    - 必要なスキルを持った人材が集まる？
    - 組織（チーム）で必要な情報がうまく共有できる？
      - プロジェクトチームは比較的短いサイクルで出来たり消えたりする



7a. Team building



- チーム開発のための必要条件
  - 明示的な要求仕様
    - 既存（参照）システム
    - プロトタイプシステム
    - 要件定義書
  - 明示的な責務分担
    - 責務分担表
  - 明示的な開発プロセス
- チームへの役割の割当
  - 縦割り
    - 部品（コンポーネント）開発
  - 横割り
    - 実装フェーズ全般
  - 組合せ
    - アジャイル開発プロセス等の活用