リスク駆動開発で、SPIN初実践とその成功
～チームソフトウェアプロセスとふりかえり～

JaSST’13 Kansai

2013/08/02

Takahiro Toku

OMRON Corporation

Mieko Yamauchi

OMRON SOFTWARE Co., Ltd.

自己紹介

徳 隆宏 (@tokutaka)

略歴

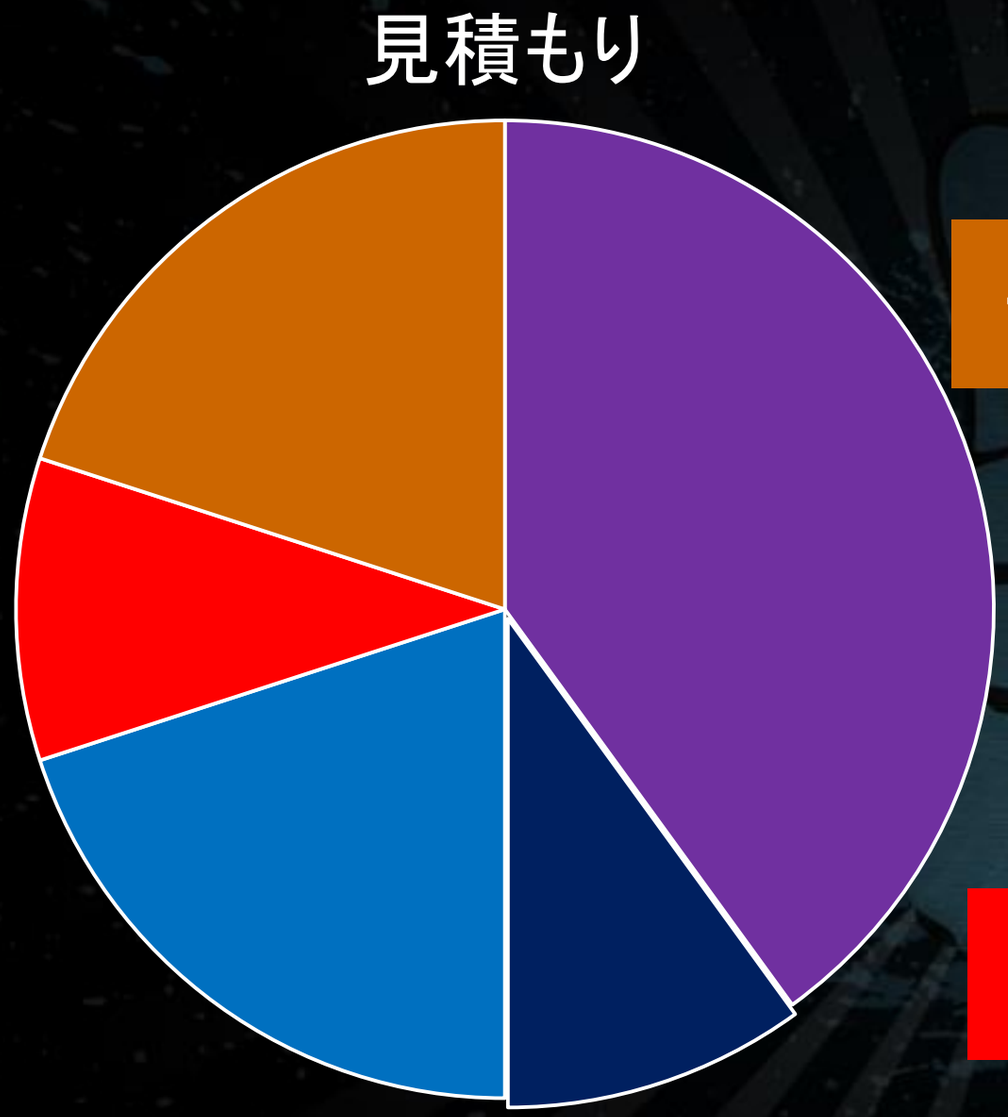
- 2005-2011 @ IBM Japan
  - エンタープライズ・組み込み各種ソフトウェアプロジェクト
  - Japan Quality Inspection Team
  - テストエンジニアリング
- 2011 – Now @ OMRON
  - FA機器ファームウェア開発
    - PLC

山内　美絵子

略歴

- 2008 – Now @ OMRON Software
  - FA機器ファームウェア開発
    - PLC
    - 温度調節器

複雑な開発での工数比率（includes review）



早期にリスクを撲滅できるか？

デバッグ時間を短縮できるか？

本日の説明

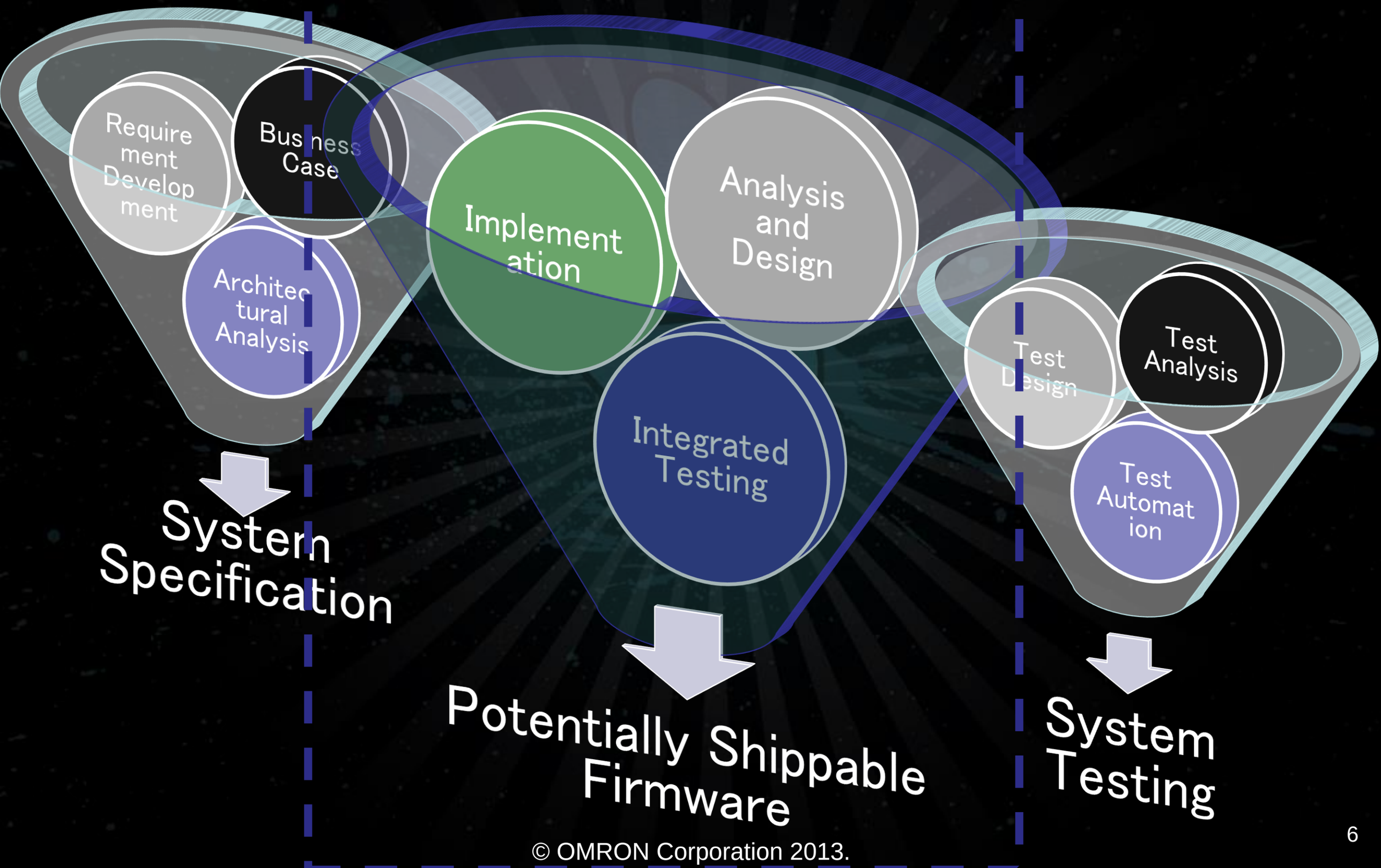
- 背景
  - プロジェクトのコンテキスト
  - リスク分析
- 実践、欠陥傾向
- プロセスふりかえり
- まとめ

プロジェクト

- PLC商品の派生開発プロジェクト
  - PLC(Programmable Logic Controller): 工場の自動機械やボイラーなどで利用する制御装置
  - 信頼性第一

See:
http://www.fa.omron.co.jp/product/special/machine-control/nj5/

Our team's scope



Team Formation

チームリーダー

徳

技術相談、あらゆるレビュー、優先度設定、チーム計画、方法論の改善検討・導入

他メンバー

3～20人など変動

メンバー

山内

派生開発のFeatureをアサイン

スレッド・状態遷移設計の経験が少ない。今回のコンポーネントやSPINは初チャレンジ

設計対象コンポーネント（派生開発）

派生開発

変更箇所

追加制御
5 states

handshake

別チーム

Application
n threads（**可変**）,
each 4 states

既存
5 threads, 20 states

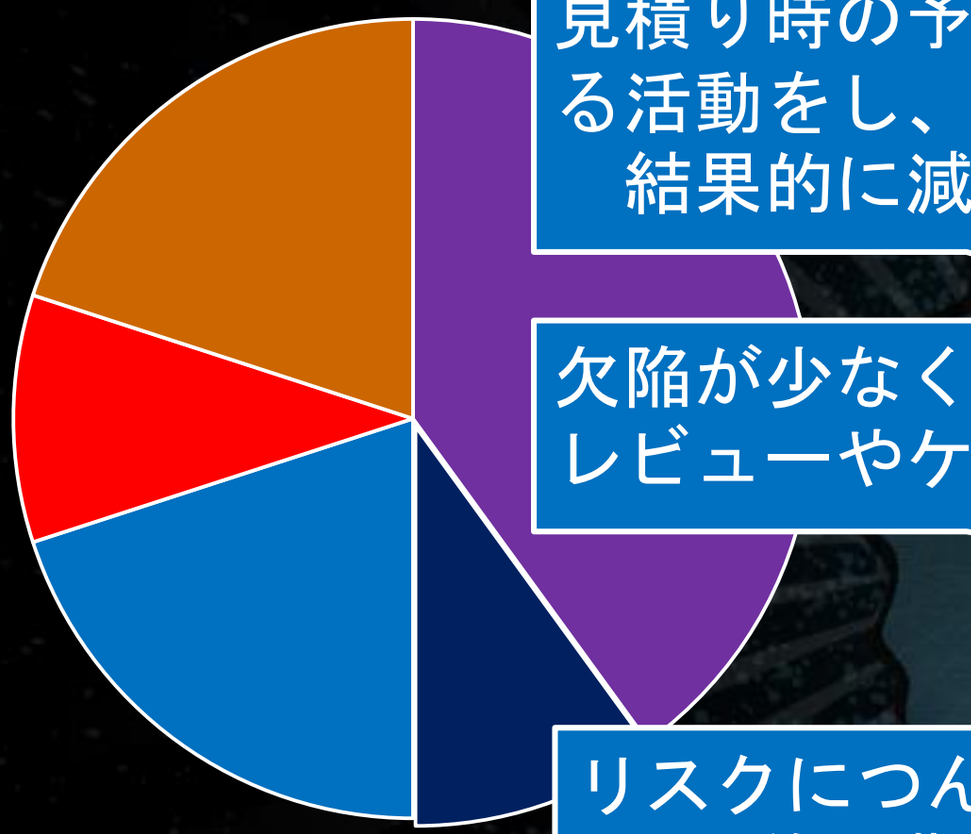
ユーザ操作、外乱

派生開発箇所の難しさ：
・ 別チームとのコラボレーション
・状態遷移・スレッド数
・既存把握した上での修正

変更箇所の難しさ：
・ 十分な設計ドキュメントはあるが
理解して変更することが難しい
・規模：C言語 4万行

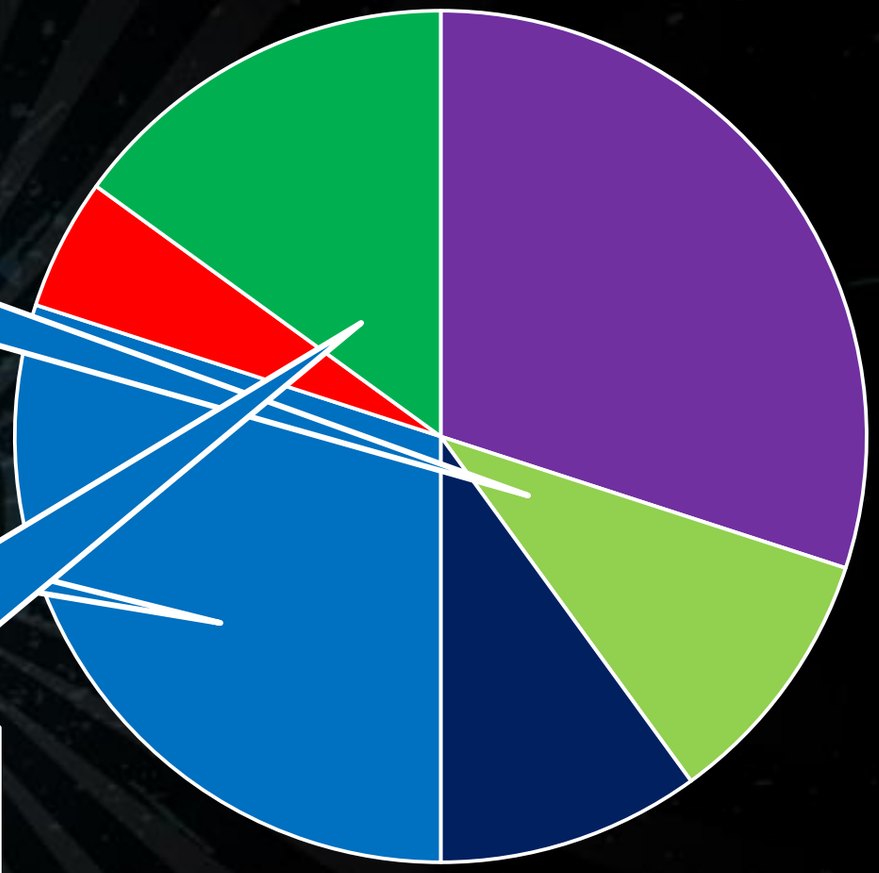
コストや期間超過なく、成功

見積もり



- Analysis and Design
- Implementation
- Testing
- Debug
- Risk(System Test reaction, Rework)

結果



- Analysis and Design
- Risk reduction with SPIN
- Implementation
- Testing
- Debug
- Retrospective, improvement

見積り時の予定と異なる活動をし、リスクを結果的に減らした

欠陥が少なく、テストレビューやケース追加

リスクにつんだ工数を改善に費やした

・過剰見積もりだったのか？

→ No

工数比率は過去の実績から利用

・ではなぜ？

→見積もりの前提が、実行過程で変化した。

（Riskとしてつんだものを別のものに使った）

Initial Risk Matrix

レビュー・テストの観点が不足し、品質確保期間が延びる	チーム間のIF設計に失敗し大幅に手戻り	-
理解・不具合への不安が膨れ、開発期間が長期化するかもしれない	状態遷移の設計、スレッド設計が既存コンポーネントに影響をおよぼす	システムテストで、不具合を検出しその手戻りが長期化する
開発期間に対して、理解する時間が不足する	不具合への不安、闇雲なレビューで士気を下げるかもしれない	不具合を潜在させたまま市場に流出する

受容できないリスク

Probability

Consequence

Risk Mitigation Plan

リスク要因と背景

- 主要なリスク要因
  - 担当者の対象理解が不十分
  - 対象が複雑であり、不具合を潜在させやすい
  - 複雑な構造の検証が困難
- 背景：
  - モデル検査はリーダーがお試し（素振り）済み

対応検討と実行

- Plan A
  - 地道な対象理解、有識者追加、設計期間・レビュー・テスト増
- Plan B
  - 対象理解、複雑な対象の検査を解決する道具の導入

Plan A：やった感が増えるだけ⇒NG
Plan B：早期にリスク減可能⇒OK
⇒ 道具をSPINとし、その利用をリスクとする

Plan B ブリーフィング
（細かいこと、余計に難しくする情報は伝えない）

Process Behavior Model: Promela

Promela Code

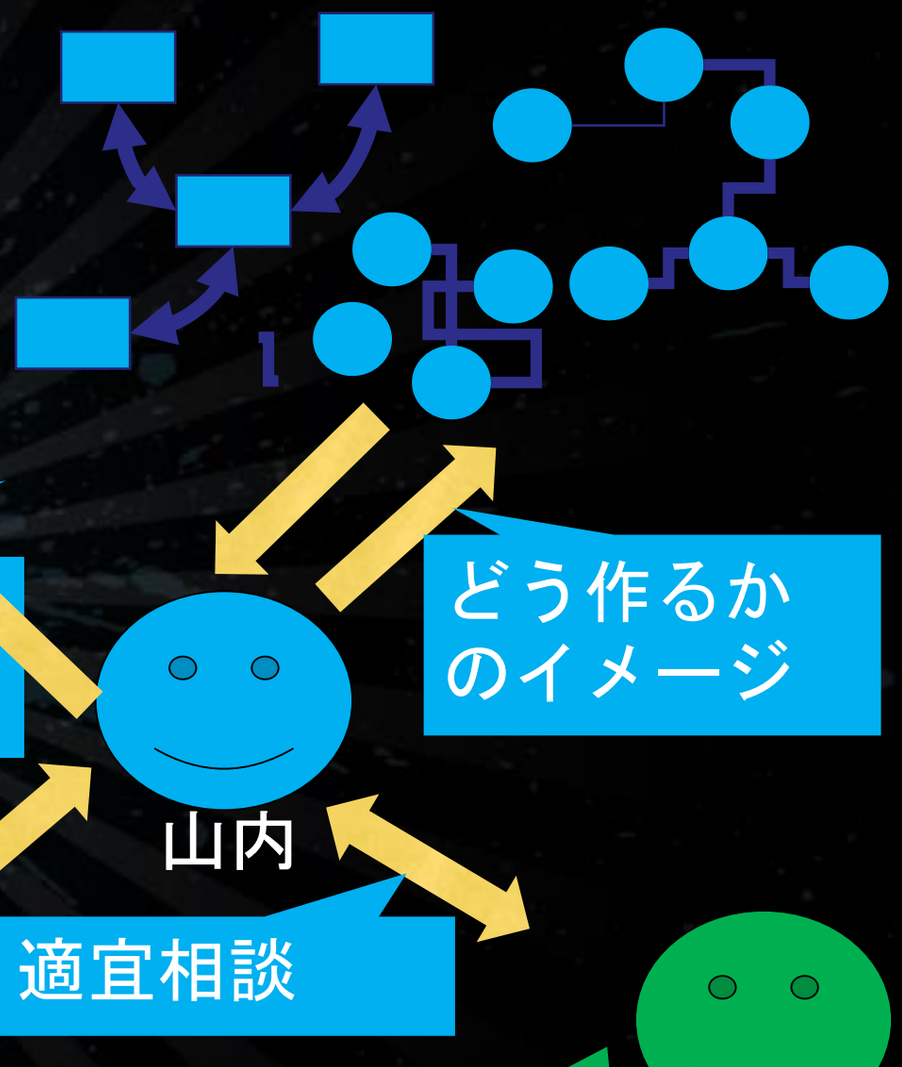
線形時相論理モデル
（という難しいことはいわない）

方式理解
モデル検査

Model Running Environment: SPIN

Simulation

Verification (Assertion, etc)



SPIN導入をどうBacklogに入れたか

- SPIN未導入・導入とで予想されるリスクについて、ブリーフィング
  - 不安を共有し、「どうしたら楽しくなりそうか」ディスカッション
  - 予想不具合の種類、間違えるポイント、SPIN使いすぎにより陥る状況
  - 狙い：
    - Promela/SPINを使うとよいか、最低限の完了基準を理解してもらう
    - 予想される不具合を知ってもらい、道具を効果的に使ってもらう
- リーダーの過去の実践状況に基づき、学習期間をとってBacklogへ
  - 期間は、過去の実績*5倍＋同一期間のバッファ （学習速度の違い）

実践者の視点

アサイン時点

既存構造をあまり知らない

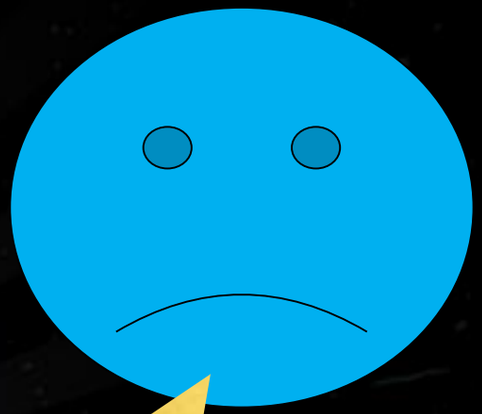
できあがりイメージがあまりない

作業計画をイメージしきれない

自分の能力が十分？不具合出そう

設計対象・仕様がかなり複雑

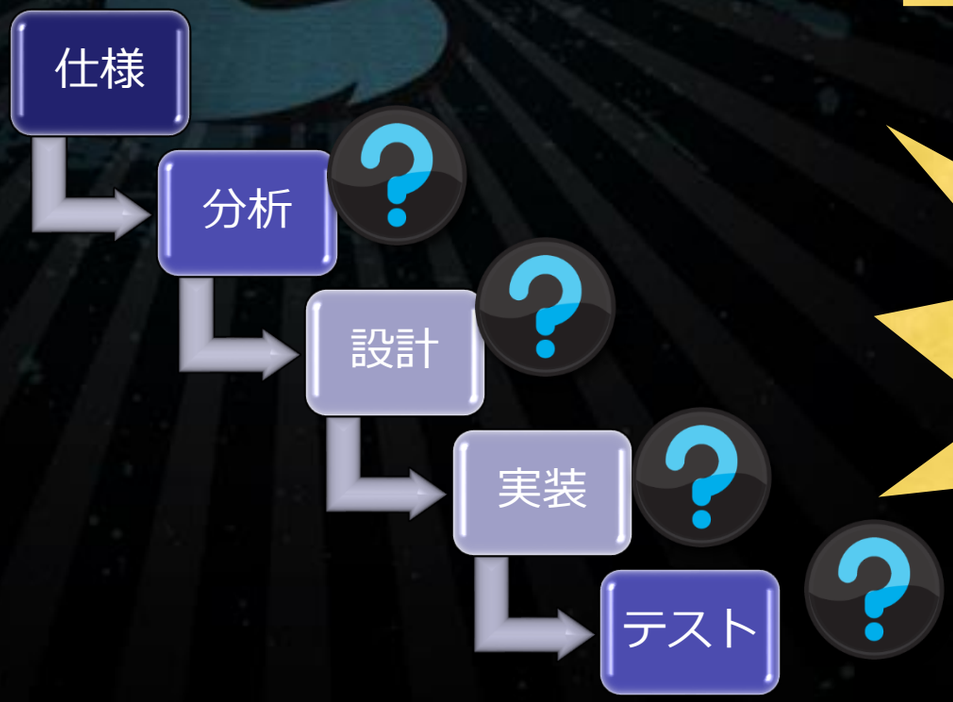
いつ、どうやってどこまで他チームと連携？

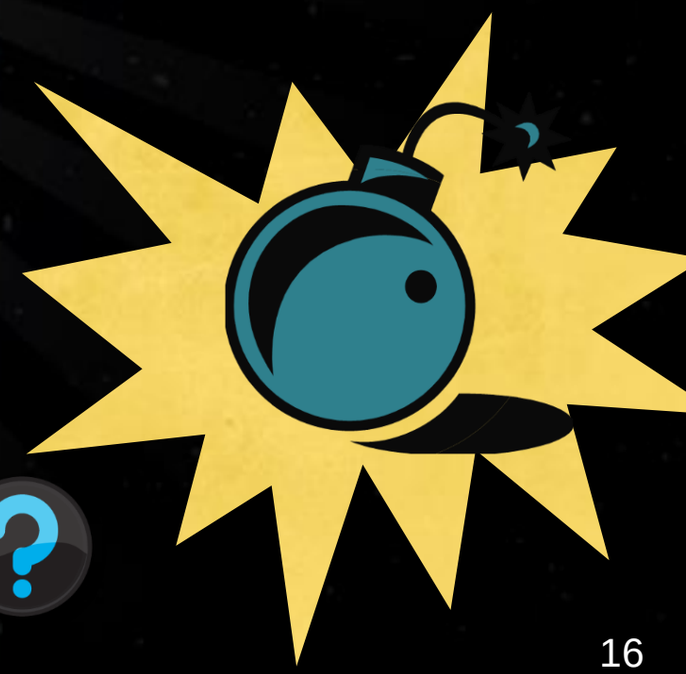


やらかしそうな不安

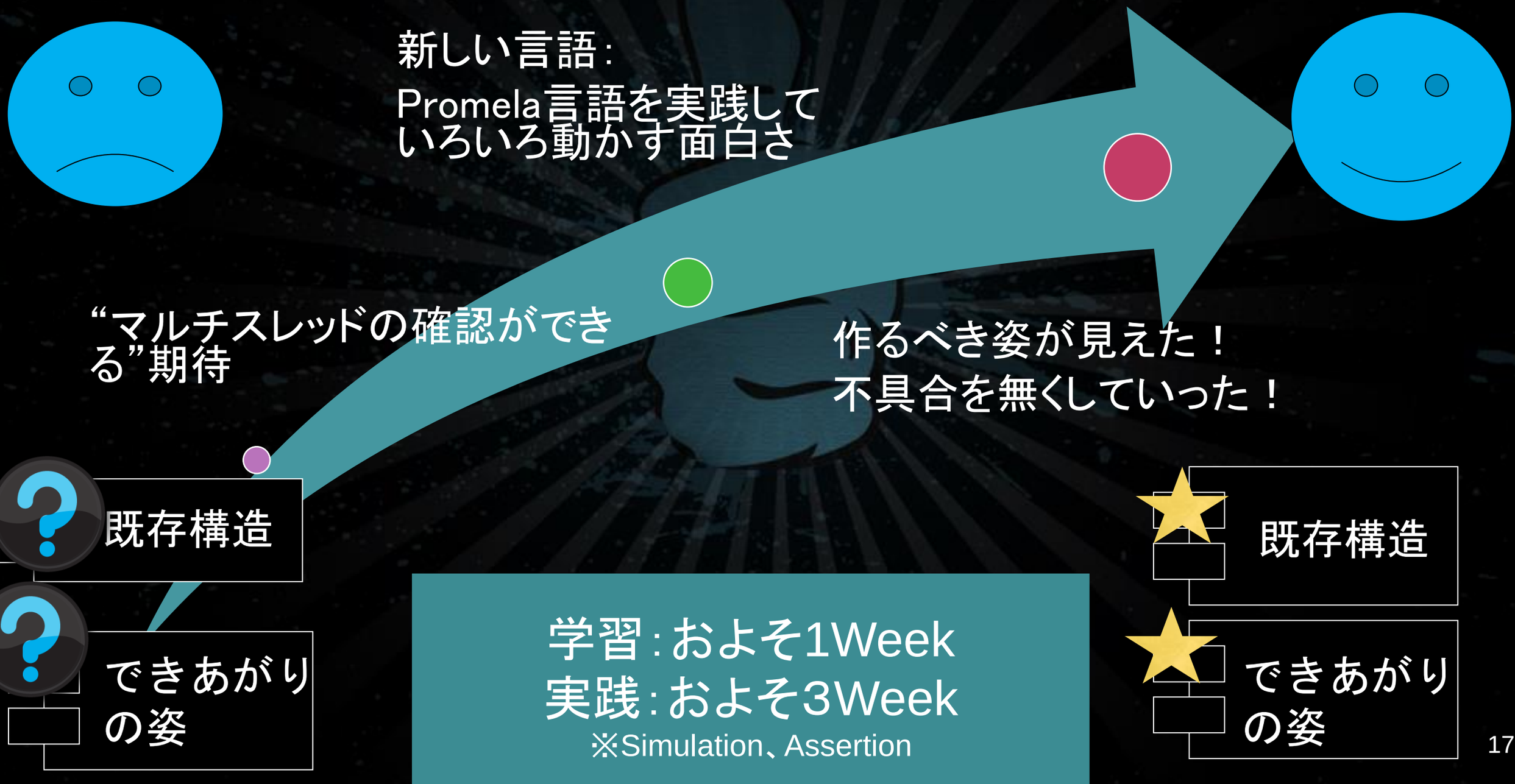
既存構造

できあがりの姿





ブリーフィング
（SPIN概要説明）以降



新しい言語：
Promela言語を実践して
いろいろ動かす面白さ

“マルチスレッドの確認ができる”期待

作るべき姿が見えた！
不具合を無くしていった！

既存構造

できあがりの姿

学習：およそ1Week
実践：およそ3Week
※Simulation、Assertion

既存構造

できあがりの姿

設計モデルの関係

（UML）分析・実現性設計

- コンポーネント図
- スレッドモデル
- 分析レベルの状態遷移モデル

（Promela）LTLモデル

- プロセス間ハンドシェイク、排他制御
- 状態遷移モデル（Promelaでの表現レベル）

（UML）詳細設計

- クラス図、API仕様
- 状態遷移モデル（実装に落ちるレベル）
- シーケンス図、タイミングチャート

このタイミングで実現性検討OKとし、他チームとやり取り。

SPIN適用とリスクとの対応

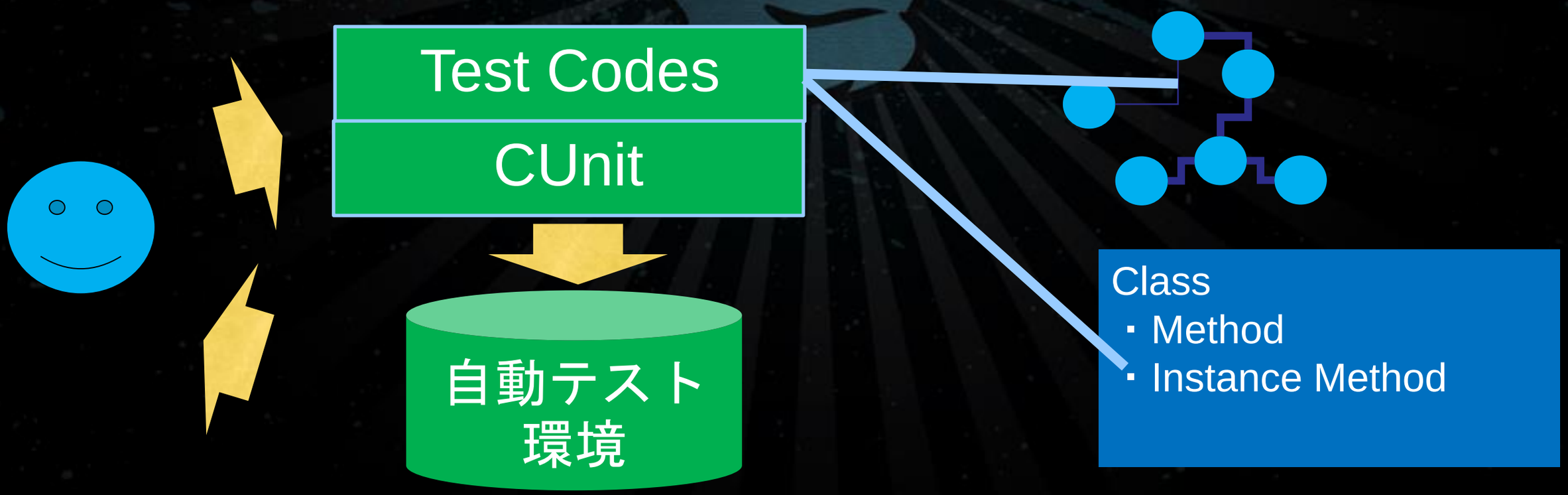
時系列	着手	実現性検討	別チームレビューでIF追加	状態遷移ウォークスルー欠陥検出
リスク	・初めてのSPIN利用 ・構造理解不足	SPINの使いすぎ	設計修正による不具合混入	設計修正による不具合混入
対処	・十分な準備期間 ・既存箇所から順にモデル化	・想定不具合の検証を中心とし、単純化	受容 状態遷移に影響なし	受容 変更は単純。 不安を他手段で対応

Implementation

- 特筆事項なし

Unit Test

- 特別なことは何もしていない、当たり前のコンポーネントテスト
  - テスト観点：コンポーネントの詳細設計どおりに実装されたか
  - 関数レベルの事前・事後・永続条件の確認、境界値などなど
  - 状態遷移設計結果としての、イベント・遷移のテスト



Integration Test

- 特別なことは何もしていない、当たり前のコンポーネント結合テスト
  - 他アプリケーションとのハンドシェイクの確認（データフロー、タイミング）
  - 状態遷移の組み合わせパス動作の確認（完全網羅ではない。外乱・異常に着目）
- 結果： バグが出ない → テストは十分なのか不安に。
- アクション：
  → システムテストケース観点調査および、リスク受容箇所のテストの程度を確認
  割り込み・多重性の点で弱い箇所あり⇒Integrationテスト一部補強

検出した欠陥

- 設計時に検出した不具合
  - SPIN利用時に検出した不具合
    - デッドロック、状態遷移誤り、資源解放漏れetc
  - レビューで検出した不具合
    - APIの観点漏れ（別チームの使い勝手、再利用性）
    - 状態遷移の観点漏れ（外乱の一部取りこぼし）
- テストで検出した不具合
  - 設計と異なる実装箇所

Retrospective

Analysis and Design

- 設計しすぎの心配→相談しながらSPINを利用
- 他チーム用APIを決める前に実現性を見極めた → 手戻りなし
- 設計観点漏れをウォークスルーで検出できた

Testing and Implementation

- テストの目的通りの欠陥が出た（設計実装不一致）
- 設計起因の欠陥がほとんどなく、不安になって外乱系のテストを増やした（でもバグがでなかった）

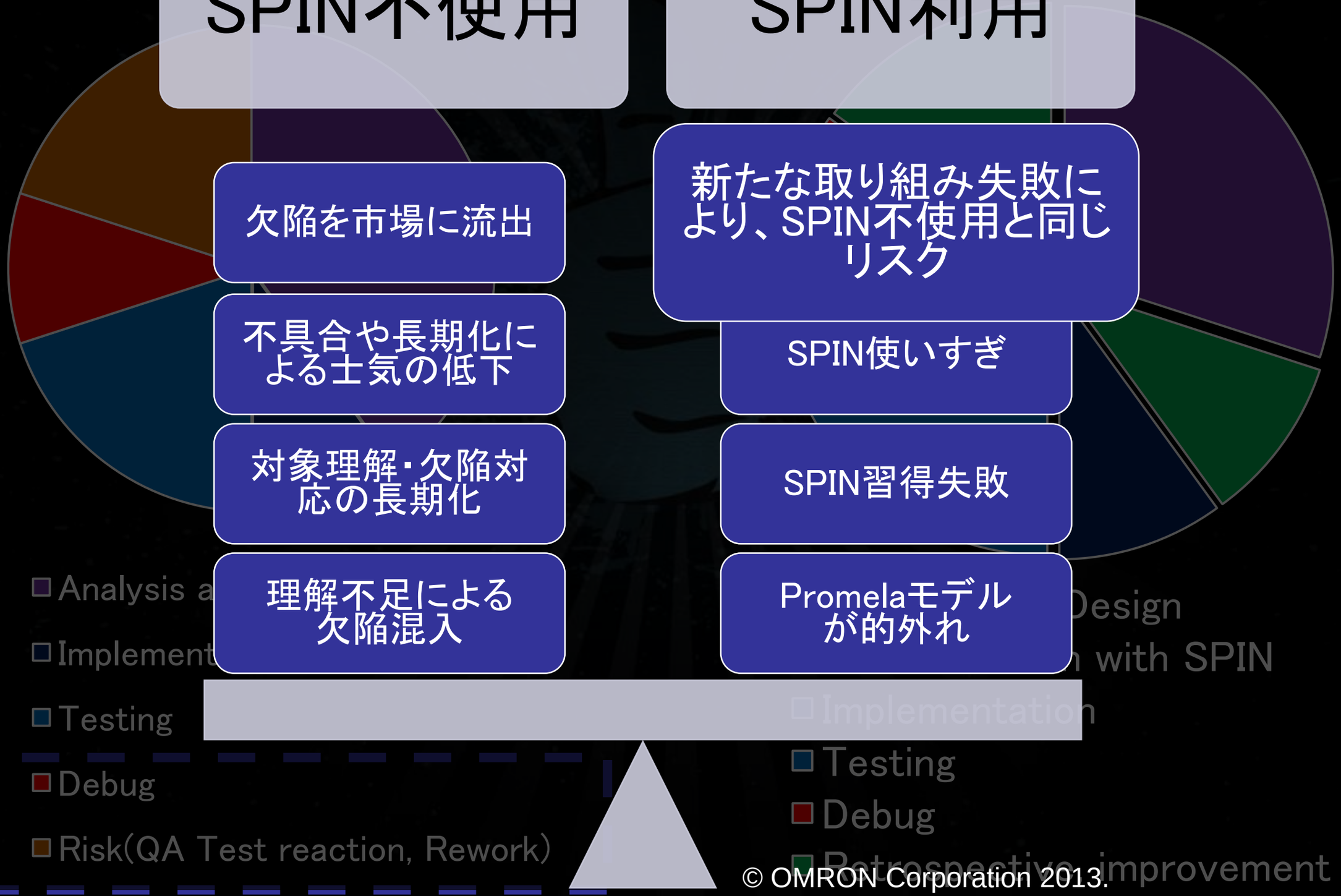
System Testing

- 不安に対する改善時間の使い方として、外乱系やリスク箇所のテスト密度を一部増やした（でもバグが出なかった）

プロセスふりかえり

所詮結果論。リスクが増える措置でもあった

SPIN不使用	SPIN利用
欠陥を市場に流出	新たな取り組み失敗により、SPIN不使用と同じリスク
不具合や長期化による士気の低下	SPIN使いすぎ
対象理解・欠陥対応の長期化	SPIN習得失敗
理解不足による欠陥混入	Promelaモデルが的外れ



再現性は？

必要条件

・継続的リスク管理

Our Team Software Process

Inception (Triggered by Upper Management)

Baseline Planning → Risk Analysis → Backlog Construction

Iteration

Improvement Plan
Praise
Context Sharing
Practice Selection
Debt Analysis
Retrospective
Iteration Planning
Self Estimation
Testing and Implementation
Execution and Review
Continuous Integration
Analysis and Design
Test Automation

Daily Interaction

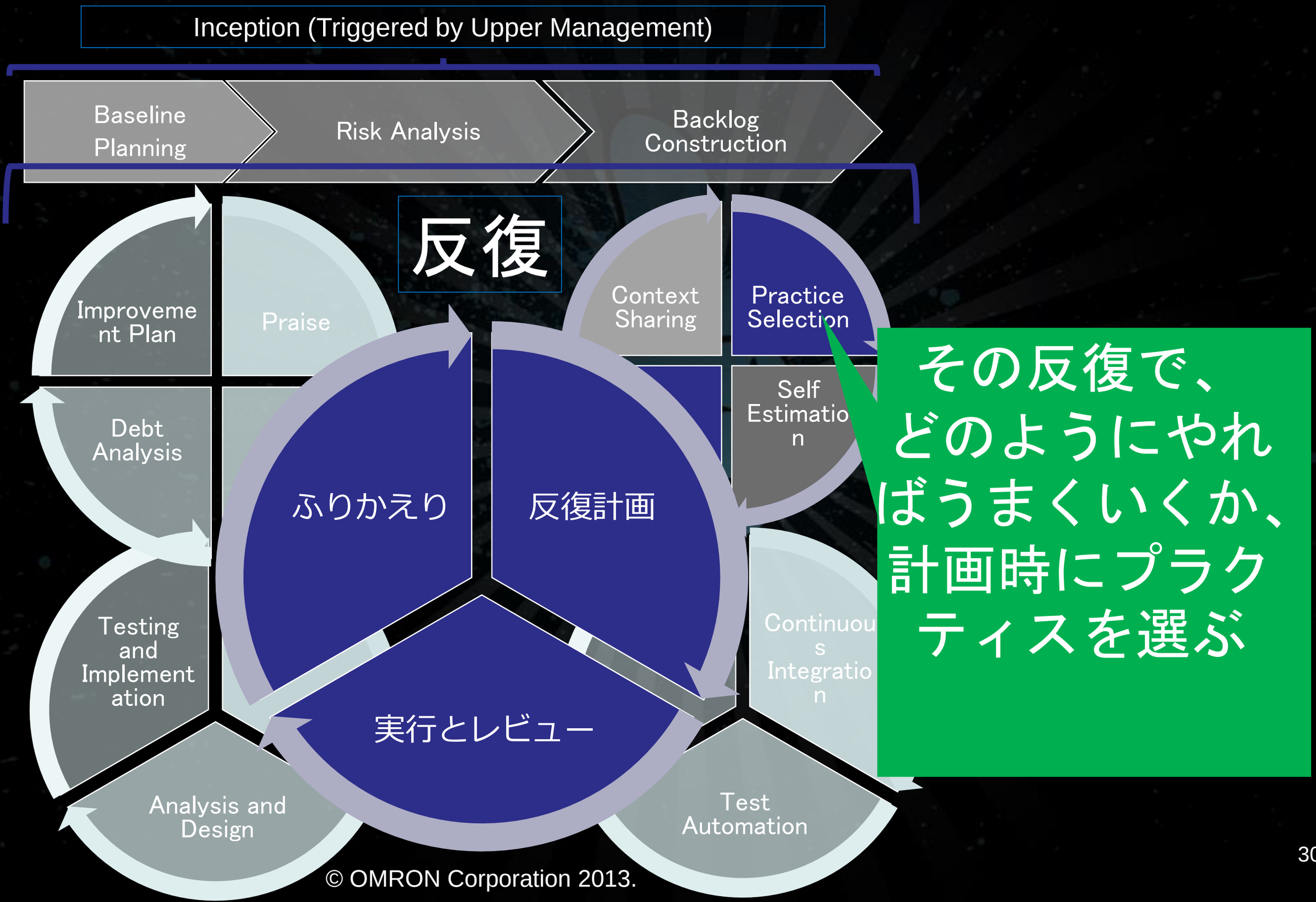
Daily Standup Meeting
Discussion
Daily Logging
Rebuild Backlog
Mitigation Planning
Risk Identification
Risk Evaluation
Risk Analysis

Our Team Software Process（初期）

初期にリスクを識別・分析して、
チームのBacklog(TODO)とする。

チームの開発計画のベースラインを
Backlogとして扱い、
チームは反復的にBacklogに対応する

Our Team Software Process（反復）



その反復で、どのようにやればうまくいくか、計画時にプラクティスを選ぶ

Our Team Software Process（日々）

Inception (Triggered by Upper Management)

Baseline Planning

Risk Analysis

Backlog

Ite

Improvement Plan

Praise

Debt Analysis

ふりかえ

Testing and Implementation

実行

Analysis and Design

Test Automation

Self Estimation

日々、状況を確認し、Backlogを整備する

日々、リスクを識別し、どのように対応するか計画する

Daily Interaction

Daily Standup Meeting

Discussion

Daily Logging

Rebuild Backlog

Mitigation Planning

Risk Identification

Risk Evaluation

Risk Analysis

チームのTODO(backlog)管理

- 以下を放置して仕事を楽しめますか？
  - やると決まってて終わってない（全体計画）
  - やらないと後々まずい・不安になる（リスク）
  - やらないと損すること（改善）

楽しめない要因をBacklogにする

チームのTODO(backlog)管理

- 以下を放置して仕事を楽しめますか？
  - やると決まってて終わってない（全体計画）
  - やらないと後々まずい・不安になる（リスク）
  - やらないと損すること（改善）

楽しむ意思と勇気がBacklogを作る

チームのイテレーション計画（プラクティスを選ぶ）

- 当初想定した方法は、着手時でも一番良いものですか？
  - 案件の完了基準、状況、勉強会などで学んだあとは？
  - 既存開発プロセスでリスクが予想できるなら？
- 昔決めたHowに拘る必要なし。プラクティスを都度選び実行
  - 状況によっては、**モデル検査・形式手法**、プロトタイピング、TDD、テスト設計 etc など、追加・変えればよい

計画変更**リスク**に向き合う必要がある

チームのイテレーション計画（プラクティスを選ぶ）

- 当初想定した方法は、着手時でも一番良いものですか？
  - 案件の完了基準、状況、勉強会などで学んだあとは？
  - 既存開発プロセスでリスクが予想できるなら？
- 昔決めたHowに拘る必要なし。プラクティスを都度選び実行
  - 状況によっては、**モデル検査・形式手法**、プロトタイピング、TDD、テスト設計 etc など、追加・変えればよい

素振りしておけば、計画変更リスクに向き合える

チームのイテレーション計画（プラクティスを選ぶ）

- 当初想定した方法は、着手時でも一番良いものですか？
  - 案件の完了基準、状況、勉強会などで学んだあとは？
  - 既存開発プロセスでリスクが予想できるなら？
- 昔決めたHowに拘る必要なし。プラクティスを都度選び実行
  - 状況によっては、**モデル検査・形式手法**、プロトタイピング、TDD、テスト設計 etc など、追加・変えればよい

楽しむ意思と勇気が、**素振り**の動機になり、計画変更リスクに向き合える

提案:
SPINなんて尻込みせず使いましょう

- 前提
  - 道具の素振りが済んでいる
  - リスクを予想・識別・対応する習慣がある
- 方法
  - 完璧を目指さない。モデル検査しなくてよいことは多い。
- 効果: 探索的テストや改善など時間を得られるかも。使い道はチーム次第

チームのリスク駆動思想

- 常に有効な万能プロセス・方法論・プラクティスはない
  - ゆえに時間軸にそって見直し・変化が必要
- 万能なプレイヤーなどいない
  - ゆえに必要なときに必要なだけ教育/OJTが必要
- その時点でリスクを下げるモデル、設計、テストといった事項を、その時系列にそって目標化する

- あらゆるリスクをなくす＝あらゆる事象を表すモデルなどない
  - UMLで描けるモデルも、形式手法も特定解にすぎない
  - 順序を考え、組み合わせ、洗練させながら利用するのが筋
  - リスクに基づいて、目標・完了基準を決める。
    - 不具合、理解の程度、難しさ、スケジュール、見積もりなどすべてがリスク
    - 継続的に、楽しむ意思と勇気でリスクを打開していくだけ

システム思考風まとめ