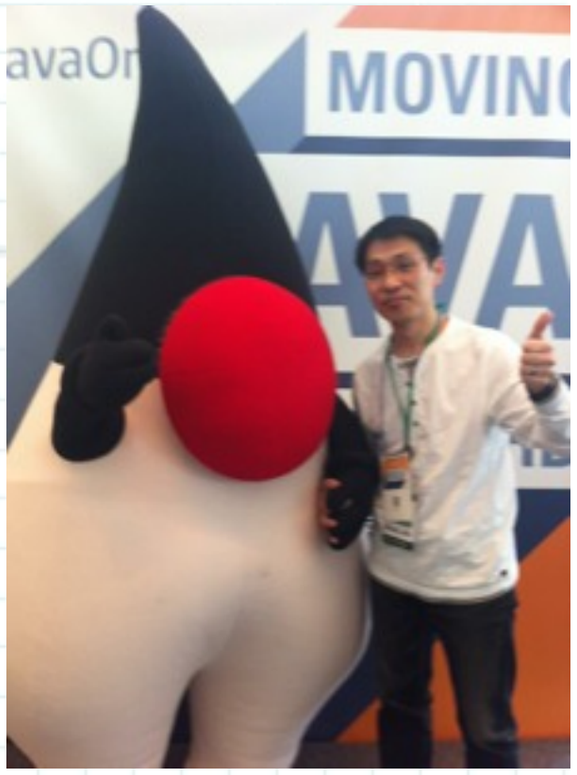
テスト駆動開発の導入

ーペアプログラミングによる学習効果ー

2012.10.26 JaSST 北海道
渡辺修司 (@shuji_w6e)

自己紹介

渡辺 修司



- Javaプログラマ
  - 株式会社エスプラニング所属
  - ウェブアプリやデスクトップアプリの開発
- テスト駆動開発、ユースケース駆動開発
- アジャイル開発
- 最近の趣味はロードバイクとフットサル

@shuji_w6e

- 札幌 Javaコミュニティ
- TDD Boot Camp
- やさしいデスマーチ（Blog）
- 執筆活動
  - WEB DB Press vol.69
  - JUnit実践入門（Soon!）

Comming Soon!

テスト駆動開発

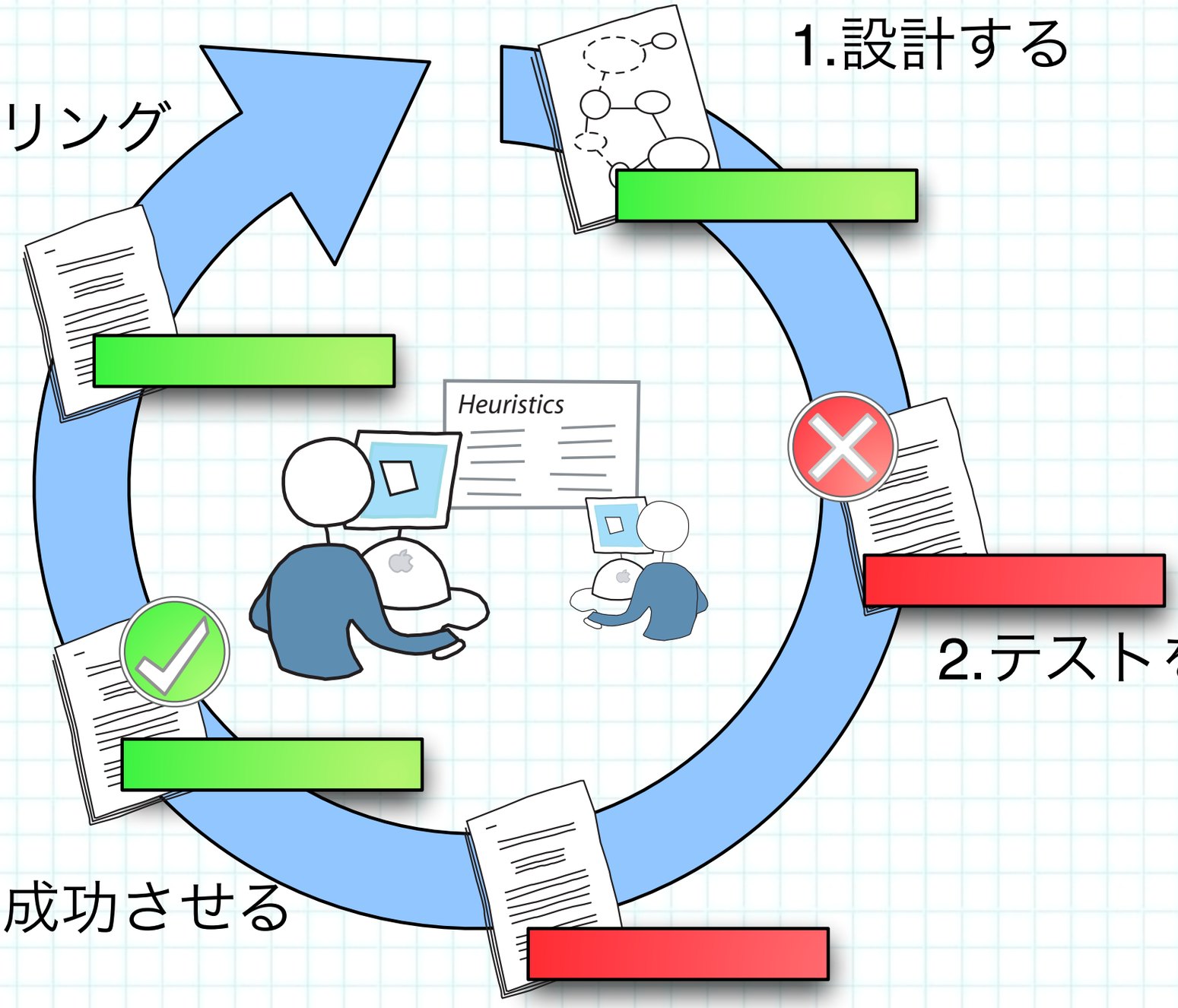
テスト駆動開発とは？

- テストコードをプロダクションコードより先に記述するテストファーストを実践し、インクリメンタルにソフトウェアを開発する開発手法

(TDD: Test Driven Development)

テスト駆動開発のサイクル

1.設計する



2.テストを書く

3.コードを書く

4.テストを成功させる

5.リファクタリング

テスト駆動開発の目的

- 明確で検証可能な仕様
- テスタビリティの高い設計
- 常にリファクタリング可能
- 素早いフィードバック

ペアプログラミング

ペアプログラミングとは？

- ペアで１つの作業を行うプラクティス
- コードを書くドライバーと、ドライバーのサポートをするナビゲーター
- 定期的な役割の交代

ペアプログラミングの効果

- すべてのコードを2人以上が理解している状態
- 曖昧な状態でコードが書かれない
- ツールの使い方やコードの書き方を学べる

問題

典型的なプロジェクトの流れ

(1) 遅れる実装

(2) とりあえずシステムテスト

(3) デバッグ地獄

(4) 後からユニットテスト

(5) 追加開発でリグレッションの発生

ぐだぐだになる原因

- 仕様が複雑でなかなか決まらない
- 納期のプレッシャー
- ユニットテストの工数がない

ぐだぐだになる本当の原因

- 完成の基準が実装者によって異なる
- 理解不足のままで実装している
- 変更に対して脆い
- スキル不足
- 担当機能以外に興味が無い
- 結合すると動かない

事例

プロジェクトの概要

- Java のデスクトップアプリケーション
  - 解析エンジン ＋ GUI（Swing）
- 5–6名で3ヶ月程度の規模
- ミッションクリティカルなシステムのデータを解析するツール

導入前

- カオスなコード
  - GUIとビジネスロジックが相互依存
  - シングルトンパターンの乱用
- 価値のないテストコード
  - privateメソッドのテスト
  - 少しの変更で通らなくなるテスト

開発手法の改善

- Tracによるissue管理
- 外部設計にユースケースを導入
- 自動化（SVN + Maven + JenkinsCI）
- ペアプログラミング
- テストファースト（テスト駆動開発）
- リファクタリング

ペアプログラミングの運用

- 初心者同士ではペアを作らない
- ２週間を目安にペアを変更
- 担当個所が偏らないように作業を配分

テストファーストの導入

- 必須とはしない
- 書ける個所はテストを先に書いてみる
- 「どうテストするか？」を意識する
- イメージが沸かない場合は軽く実装する

開発の流れ

(1) ナビゲータに実装することを説明する

(2) ナビゲータからの質問に答える

(3) 合意したならば、実装を進める

(4) 適宜ツッコミを受ける

(5) 一区切りついたならば役割をチェンジする

効果

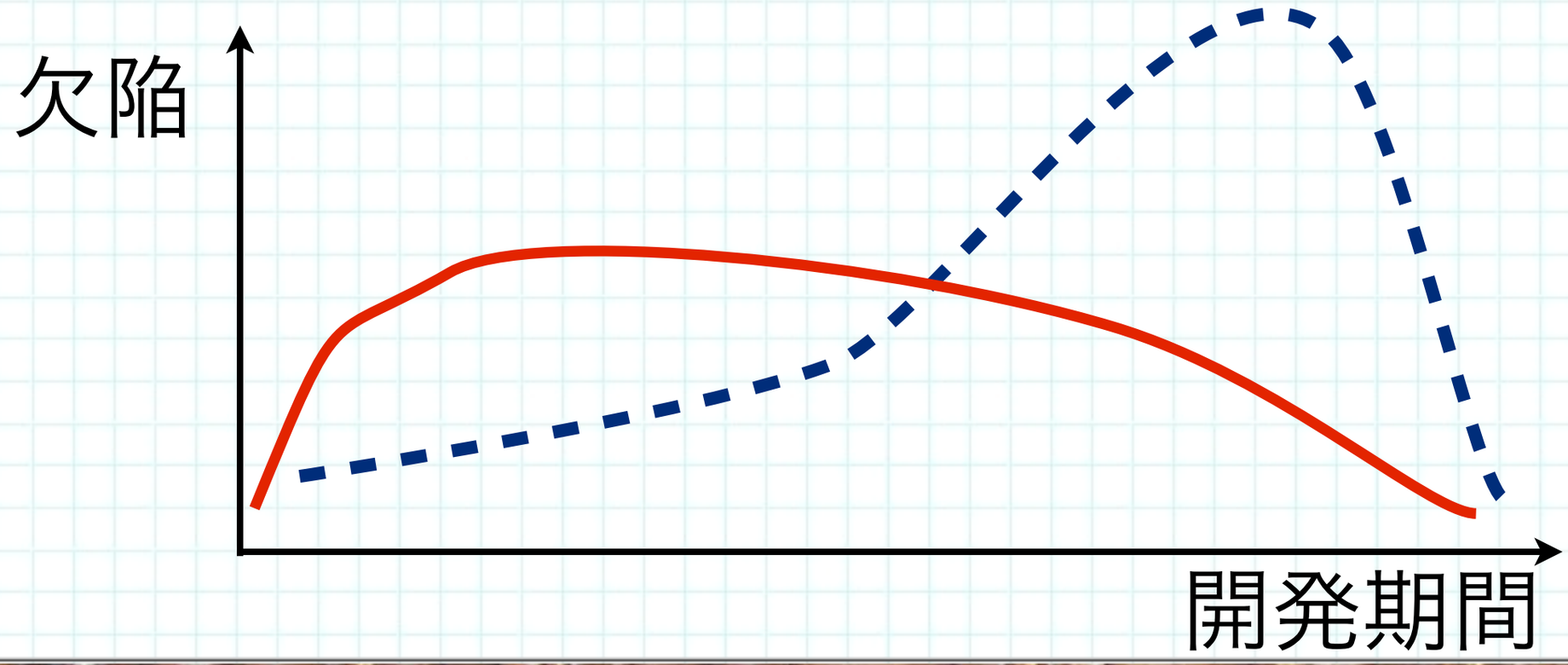
作業時間

- 実装完了時間はほぼ２倍
- デバッグ時間は半減
- システムテスト後の修正時間は半減
- トータルで２−３割増えた程度

  ※慣れてくれば同程度の時間になる可能性

品質

- 不具合の質が変化
- システムテスト時の不注意的な欠陥が激減
- プロジェクト中盤でも安定して動作



教育的効果

- ツールの使い方
- コーディングの技報
- 他人の「考え方」を知ること
- 多くの事を学べるが、プログラミングを作業と考えていると効果は薄い

考察

テスト駆動開発とペアプロの効果

- 効果的なユニットテスト
- ユニットテストの強制
- 考えたことを相手に伝えるプロセス
- プロジェクトナレッジの共有
- プログラマのスキルアップ

効果的なユニットテスト

- CIの導入による素早いフィードバック
- プロジェクト終盤での「辻褄合わせ」の減少
- 「無駄なユニットテスト」の減少
- privateメソッドのテスト
- インターフェイスを意識したテスト

ユニットテストの強制

- 面倒なテストを先に書く
- 後から書くのはもっと面倒
- 「動くことを確認するテスト」から
「動かすためのテスト」へ

考えたことを相手に伝えるプロセス

- バグのあるコードは「不安なコード」
- 自信が無いのでコメントで誤魔化す
- コードに落とす前に言語化して伝える
- 考えが曖昧なままで書かれたコードが減少
- コピペするにも考える
- 会話が増える

プロジェクトナレッジの共有

- 「この機能はXXさんが担当したから解らない」がほとんどなくなる
- 知見をチームに伝搬させる
- 担当ではない機能にも興味を持たせる

プログラマのスキルアップ

- コーディング上のテクニック
  - ツールの使い方
  - デザインパターン
    - リファクタリング
- 問題に対する「考え方」

課題

基礎体力は必要

- プログラミングスキル
  - リファクタリング
  - 設計パターン
- テストスキル

教育的コスト

- 初心者にとって学べることは多い
- プロジェクトのリソースと教育コスト
  - 当然のように教育コストを払ってくれない
  - 余裕のないプロジェクトでは難しい
- 一時的なメンバーでは効果が薄い

実績と経験者

- 経験者がいないと難しい
- ワークショップなどで「素振り」が必要
- 初心者は経験者と組むことで楽
- 経験者は思った以上に疲れる

ワークショップのススメ

- 半日から数日のワークショップ
- テストファーストとペアプロに慣れる
- 小さなアプリを作成できると良い
- 必ずリファクタリングまで回すこと

まとめ

- テスト駆動開発（ テストファースト）
  - 先に振る舞いを考えることの効果
  - テスタビリティの高いシンプルな設計
  - 動作するきれいなコード/リファクタリング
- ペアプログラミング
  - スキルの伝搬
  - プロジェクトナレッジの共有
  - コミュニケーション

質疑応答