DI時代のTDD入門

or: How I learned to stop worrying and love the test.

タワーズクエスト株式会社
プログラマ兼社長
和田 卓人(id:t-wada)

自己紹介

- 名前：和田　卓人（わだ たくと）
- ブログ： http://d.hatena.ne.jp/t-wada
- メール：takuto.wada@gmail.com

- S2AnA, S2Continuations, S2Buri, S2Coffeeコミッタ

- タワーズクエストという会社を経営しています
  - 仲間が増えだしました
  - 一緒に仕事する仲間まだまだ募集中!!

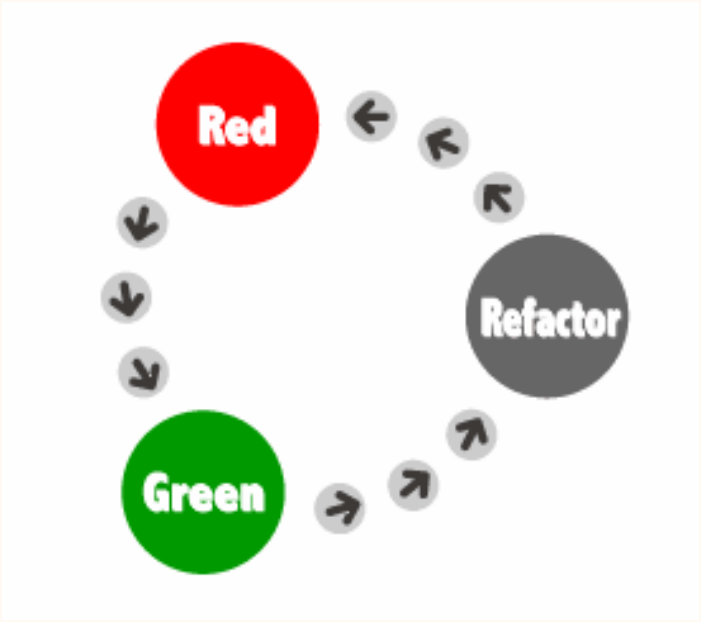
(宣伝) WEB+DB PRESS Vol.35

巻頭特集を執筆
させていただきました。
今日の講演の下敷き
にもなっています

他に最近書いたもの

アジェンダ

- TDD概論
- TDD in Action
  - マクロサイクル
  - ミクロサイクル
- Eclipseを使った実践的TDD
  - ショートカット
  - デモ
- DI時代の実践的TDD
  - 何が変わったのか
  - テストの勘所
- おわりに



今日のまとめ

- TDDのテストは品質保証のためではなく、開発促進のためのツール
- テストには粒度があり、資産価値がある
- DIコンテナによってTDDがやり易くなった
- Eclipseを使いこなすべし!

テスト駆動開発(TDD)とは何か

TDDの流れ

- コードを書いてからテストを行うのではなく
  1. テストコードを書き
  2. そのテストを実行して失敗させ (Red)
  3. 目的のコードを書き
  4. 1で書いたテストを成功させ (Green)
  5. (必要であれば) テストが通る状態のままで、リファクタリングを行う (Refactor)

1～5を一つの単位として、小さいサイクルで繰り返しながら開発を行っていく開発手法です

リファクタリングとは何か

リファクタリング（名詞）：

外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変更させること。

リファクタリング（動詞）：

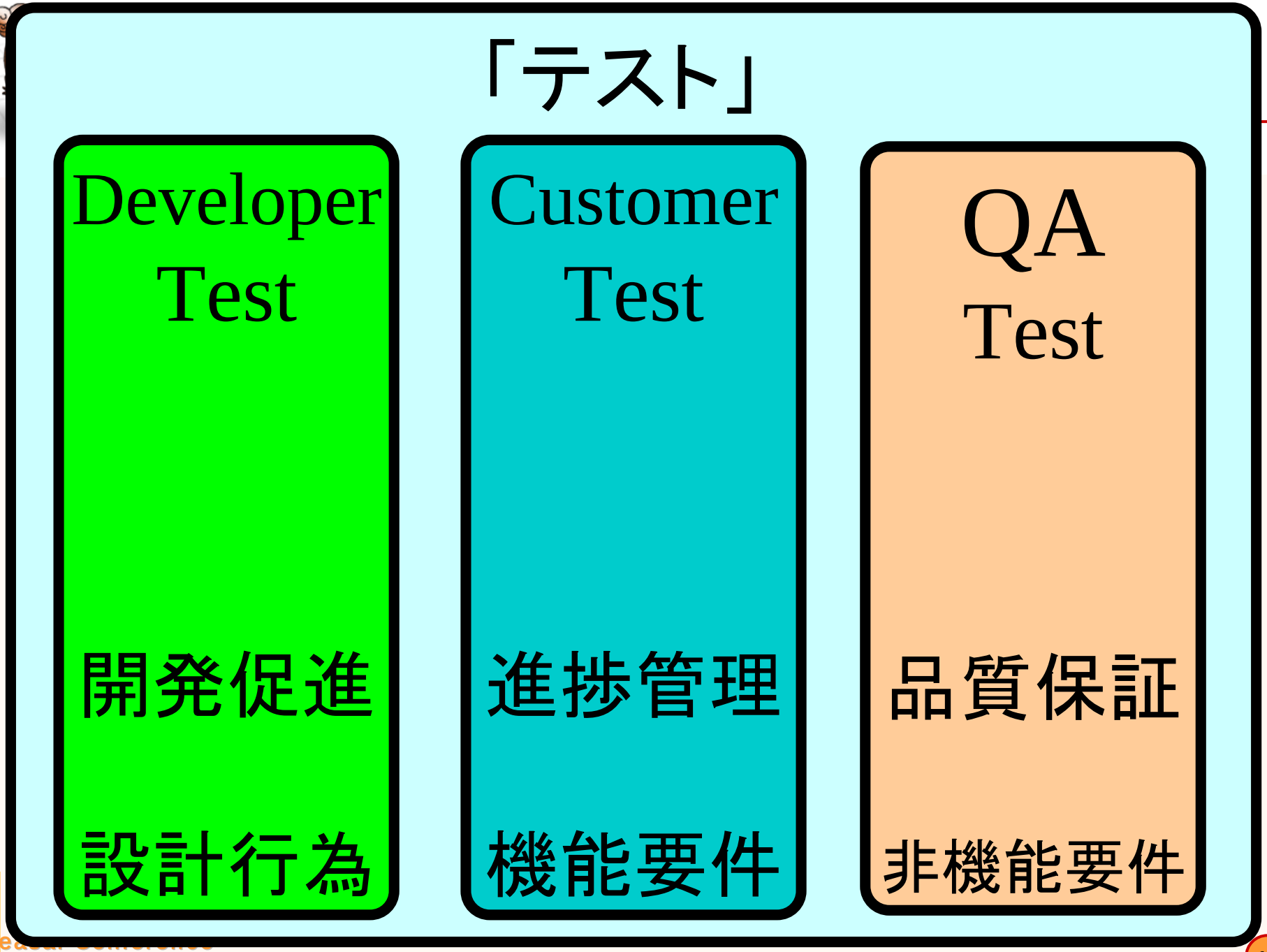
一連のリファクタリングを行って、外部から見た振る舞いの変更なしに、ソフトウェアを再構築すること。

(参考) 「リファクタリングの定義」
http://capsctrl.que.jp/kdmsnr/wiki/bliki/?DefinitionOfRefactoring

TDDにおける「テスト」とは

- TDDの「テスト」とは「Developer Test」のこと
- Developer Testとは…
  - プログラマの
  - プログラマによる
  - プログラマのための
  - プログラムとして書かれたテスト

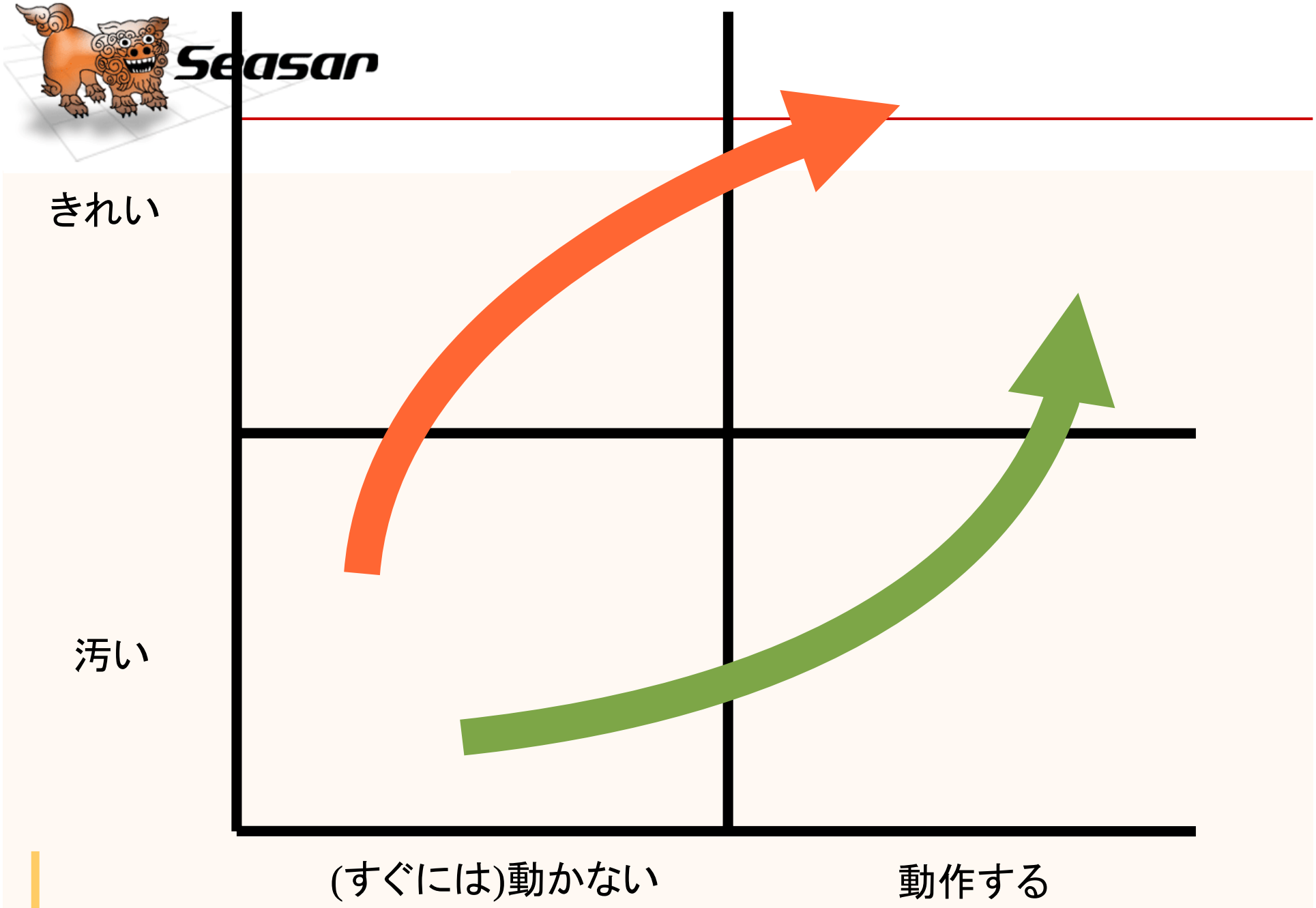
  - 開発促進のツールとしてのテストであり、「品質保証のためのテスト」ではない



目標は「動作する、きれいなコード」

- 私達プログラマの目標とするのは、「動作する、きれいなコード」

- そこに至る道はひとつではない
  - きれいな、から攻めるか
  - 動作する、から攻めるか



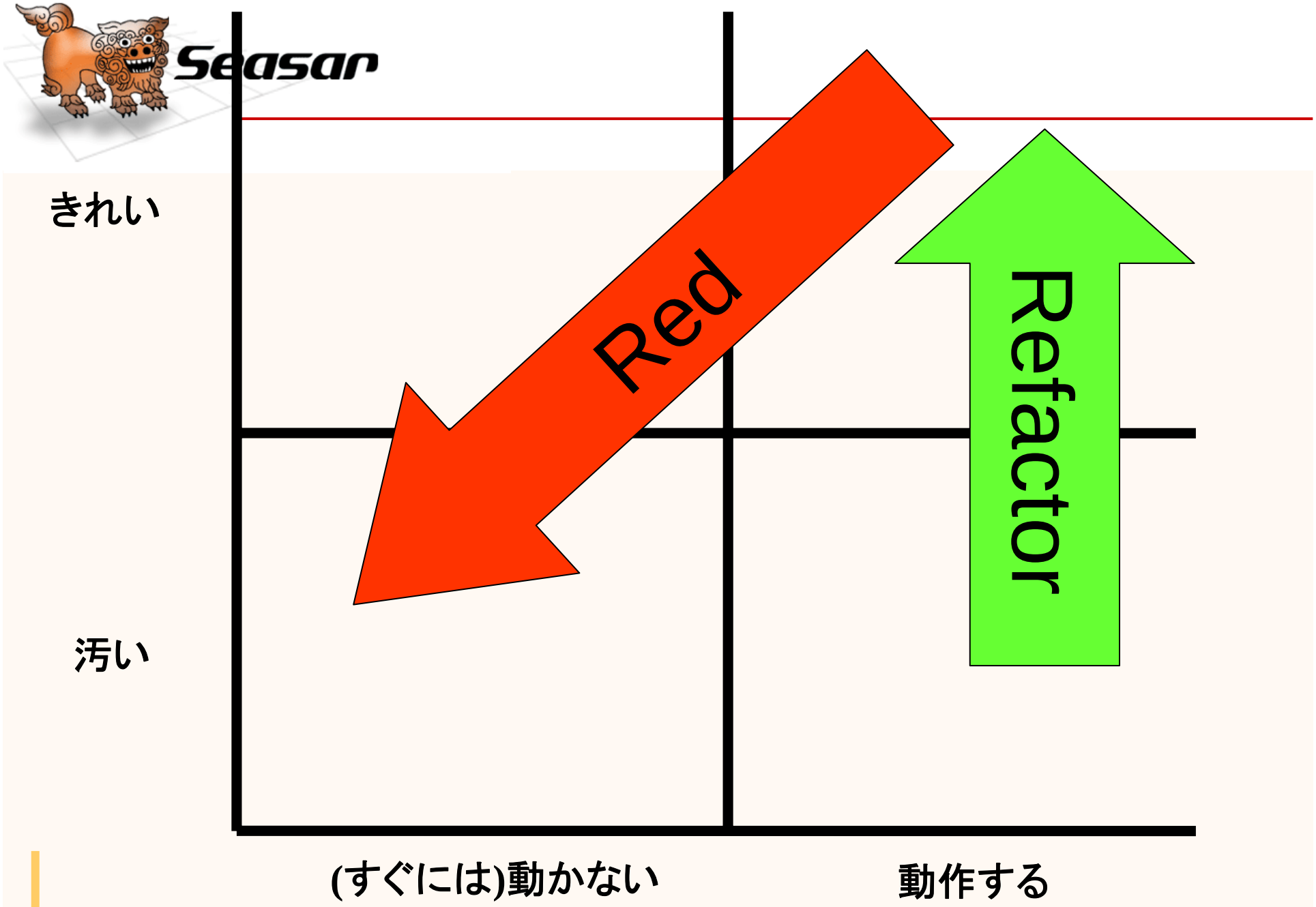
TDDのリズム

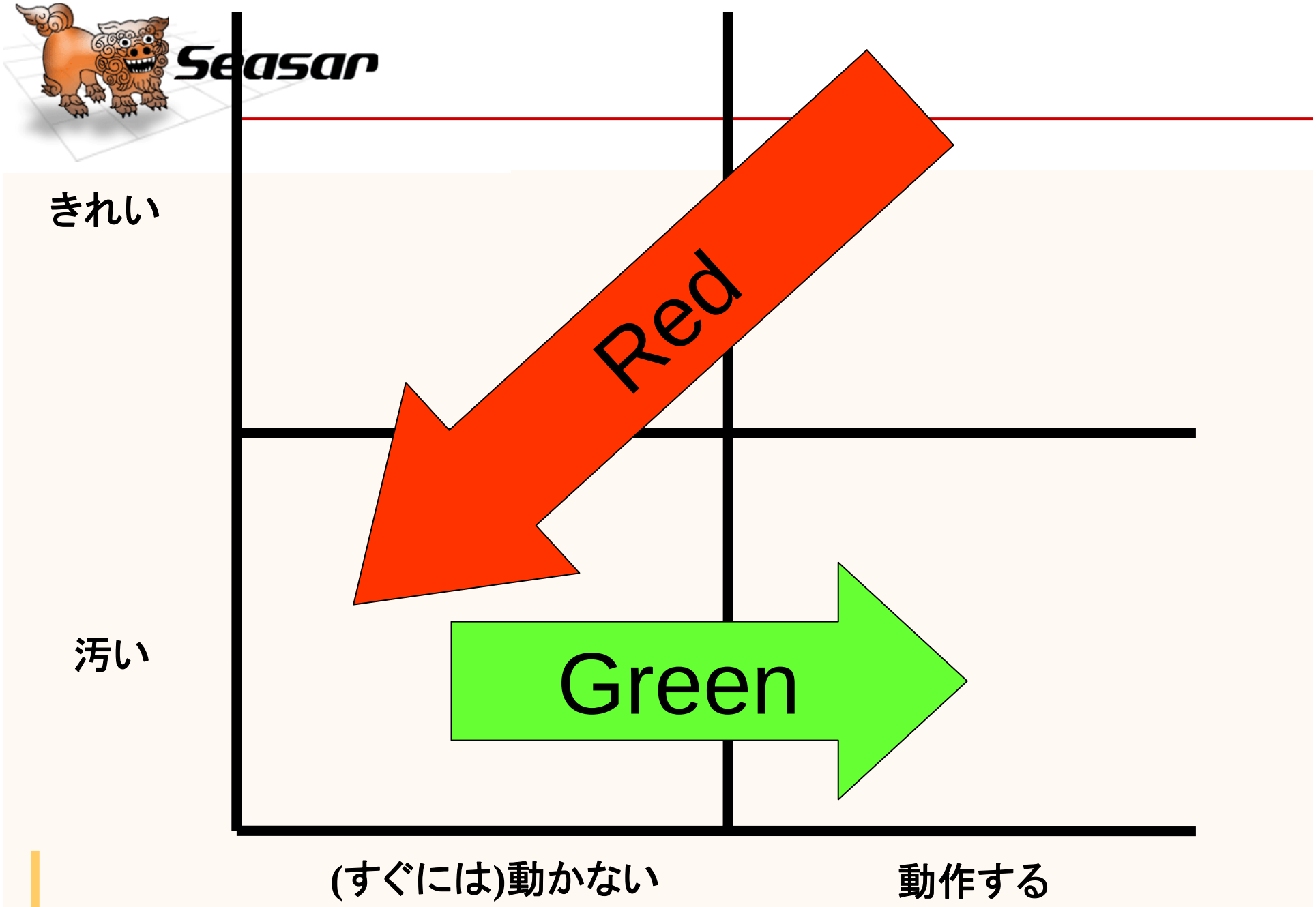
- Red, Green, Refactor
- Red, Green, Commit, Refactor, Green, Commit

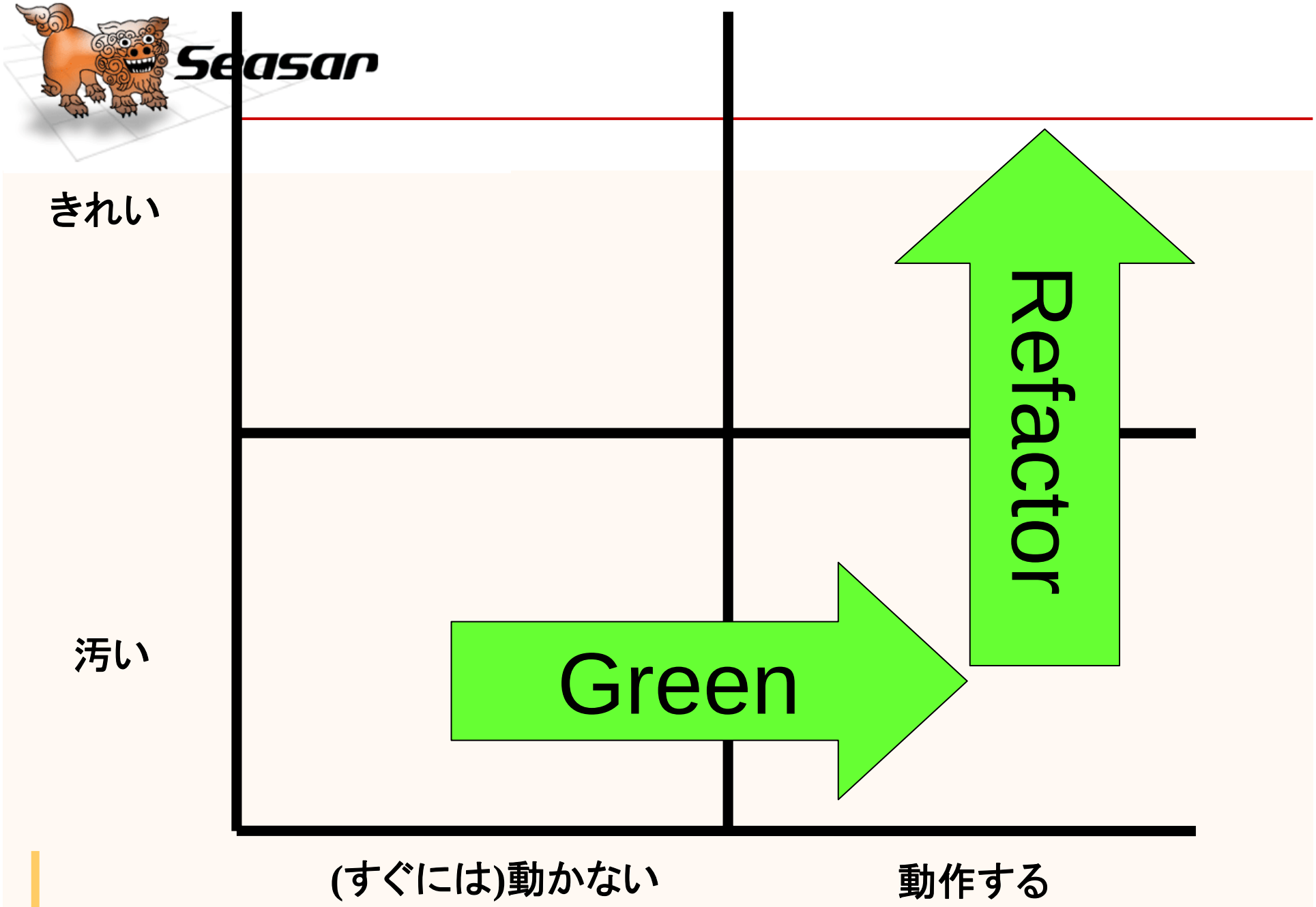
TDDでは、

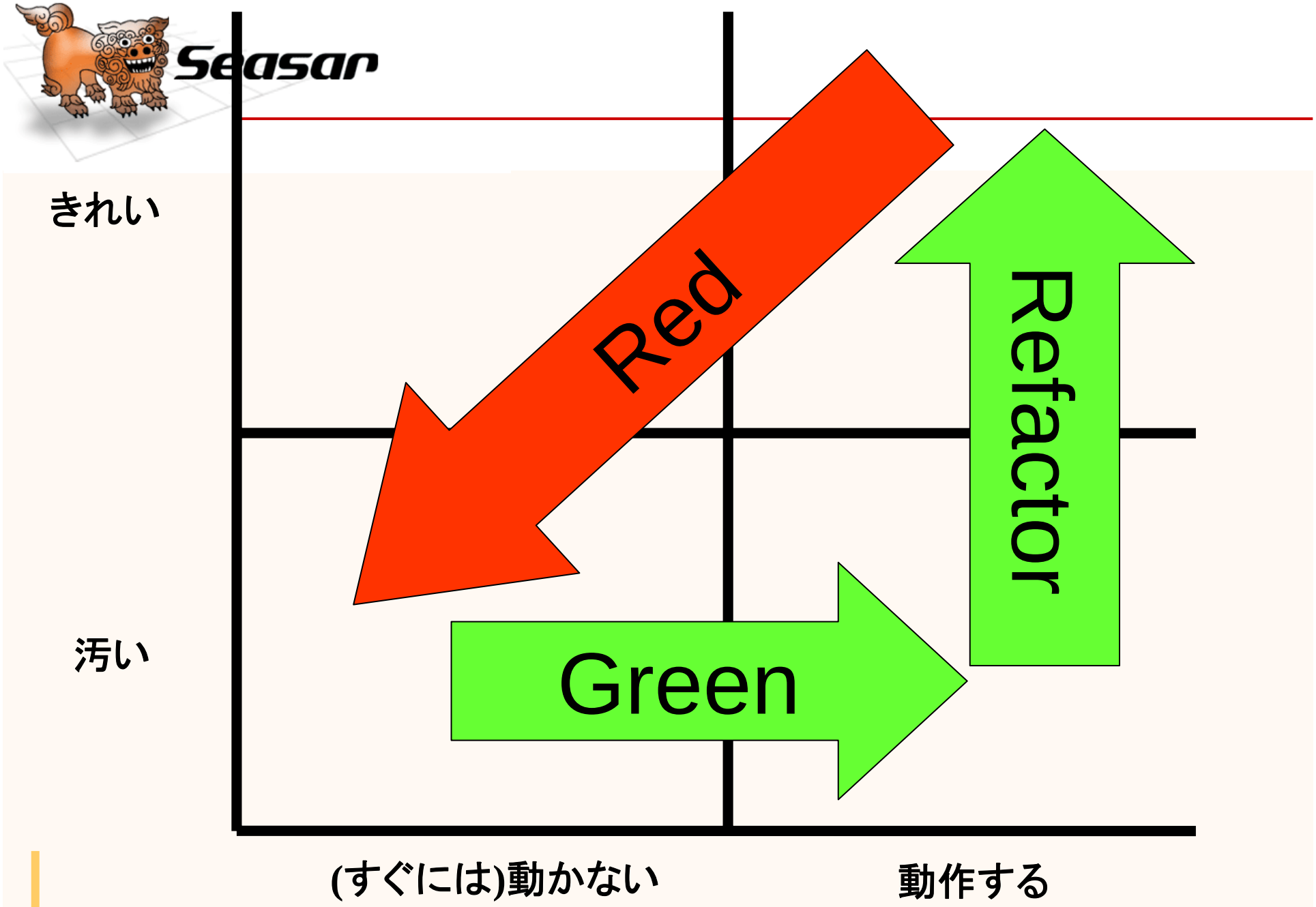
「動作する」を満たしてから

「きれいな」にとりかかる









TDDは設計技法です

- TDDはテスト技法ではなく、設計技法です
  - プログラミングは設計行為である(J.Reeves)
  - Redは仕様の設計
  - Greenは仕様の実装
  - Refactoringは内部設計の改善

参考「ソフトウェア設計とは何か」
http://www.biwa.ne.jp/~mmura/SoftwareDevelopment/WhatIsSoftwareDesignJ.html

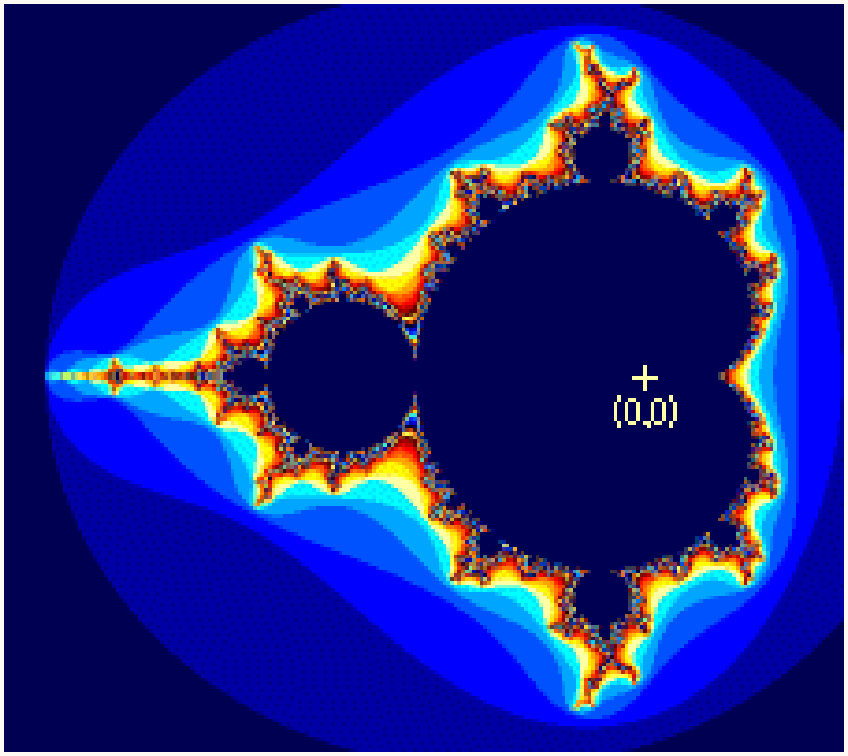
なぜTDDをするのか

- 私達は完璧なプログラマではない
  - 最初から思い通りにコードが書けるほど、私たちは賢くない
  - 最初から思い通りに動作するほど、対象は単純ではない
- すばやく対象に近づき、フィードバックを得て方向修正をしながら開発を行う必要があるから

TDD in Action
(ちょっとプロセスの話も)

実践的なTDDの世界は自己相似形

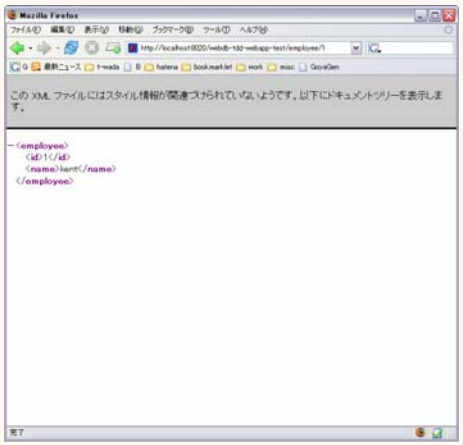
- ミクロサイクル
  - ユニットテスト
  - Red → Green
  - Red → Green → Refactor
  - 仮実装 → 三角測量 → 一般化
  - 学習テスト → ライブラリ活用
  - コミット → 編集 → コミット
- マクロサイクル
  - 機能テスト
  - タスク
  - 受け入れテスト
  - ストーリ
  - (イテレーション)
  - (リリース)
  - (契約)



「で、どうなったら嬉しいの?」

- 最初に書くテストは…
  - Kent Beck曰く、「以降のテストよりもハイレベルで、アプリケーションテストに類似することが多い」(『テスト駆動開発入門』p133)
    - 受け入れテスト
    - End-to-Endの機能テスト
- 「こうなったら嬉しい」という状態をテストとして書く
  ⇒ ToBeから引っ張る
- 受け入れテストから先に作成する意味
  - こうなれば終わり、ということを明確化する
  - 機能が完成したことを明確に知るため

例：「こうなってほしいなぁ」



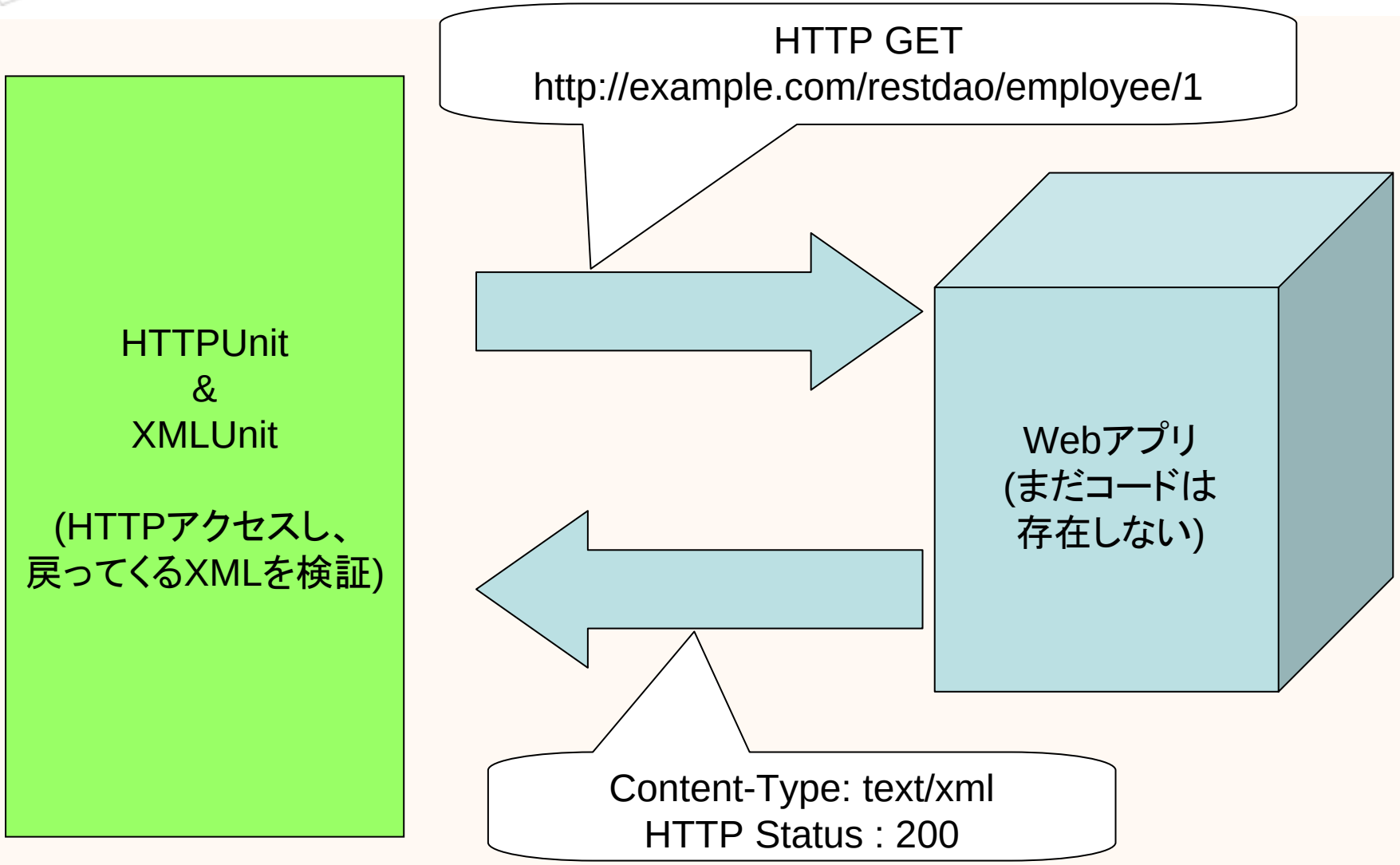
HTTP GET
http://example.com/restdao/employee/1

Webアプリ

```
<employee>
  <id>1</id>
  <name>kent</name>
</employee>
```

Content-Type: text/xml
HTTP Status : 200

「こうなってほしいなぁ」を受け入れテストに



TDDのミクロプロセス

- テストファーストプログラミング
- 仮実装、三角測量、明白な実装
- 学習テスト
- モックオブジェクトテスト技法
- リファクタリング
- コミット、リバート

テストファーストプログラミングの意義

- 実装と設計を分けて考えることができる
  - これから書くコードをどう書くかではなく、これから作るものはどういうものなのかを考える
  - HowではなくWhat
- 利用者の視点を最初から得ることができる
  - “Eat your own dog food”

- 必然的にテスト可能なコードになる
- 実装がないのでブラックボックステストになる

仮実装、三角測量、明白な実装

- 仮実装 & 三角実装

  仮実装：まずはテストが望む結果をそのまま返してGreenにしてしまう

  三角測量：その後に異なるデータでRedにし、一般化する

  – 仮実装 & 三角測量に意味はあるのか? ⇒ ある
    - 設計と実装を切り分ける
    - テストコードの実装とプロダクトコードの実装を切り分ける
    - 対象が複雑な場合に威力を発揮する(WEB+DB PRESS Vol.35, p.38参照)
- 明白な実装
  – 実装イメージがすぐに見えている場合には仮実装を飛ばす
  – 不安がない場合には一気に実装まで

学習テスト

- 問題：未知のライブラリを使って作業するときには、ライブラリの使い方を習得し、活用してコードを書くことになる
  ⇒ 二つのものを相手にすることになってしまう

- ライブラリの使い方の学習目的のみのテストを書く
  - 学習結果がテストとして残る(後の資産となる)
  - 学習のみを目的とするので像がブレにくい

- TDDの原則
  - 「一度に一つのものを相手にする」
  - 「不安をテストにする」

モックオブジェクト

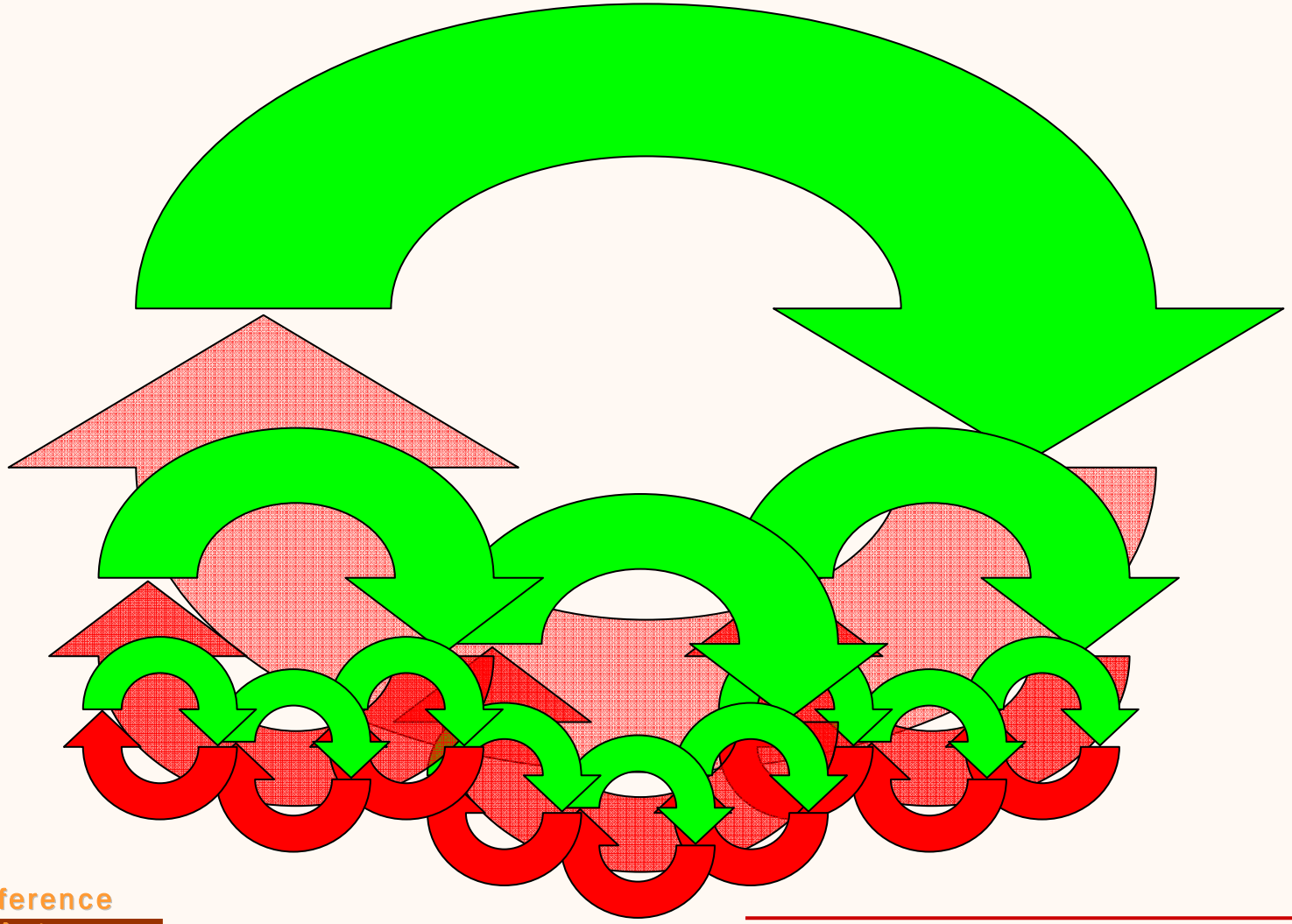
- テストのために作成する、「本物のふりをするニセモノ」のオブジェクト
- セットアップが難しい、またはコストが高いリソースアクセスの代わりになる
  - ロジックのテストを高速に走らせることが出来る
  - 実際の環境では再現の難しい例外状態を作り出すことが出来る
    - ファイルシステムエラー
    - ネットワークエラー
- 便利だが、使いすぎ注意
  - あくまで妄想の産物であるため
- 資産価値も低め

「その不安をテストで表現できないか?」

- 梯子、階段のメタファ
  - 調子のいいときは二段飛ばし
  - 心配なときは一段ずつ
- はしごを上ったらはずしてもいい
  - 用済みのテストを捨てる
  - メンテナンスコストが安心料を上回るとき

- GTD的な時間ルールを決めてもいいかも
  - 「10分以内にグリーンバーにたどり着かない場合には、もう一段粒度の細かいテストを書く」など

Red – Greenのサイクルも自己相似



Eclipseを使った実践的TDD

Eclipse時代のTDD

- ショートカットを使いこなす
  - よく使うショートカットは覚えよう
  - マウスに触らない分だけコーディング効率アップ
  - リファクタリングは「Alt+Shift+ニーモニック」
- コンパイルエラーを逆手に取る
  - Eclipseに「提案」させる
  - コーディング量を減らす
  - アサートファーストの実践
- Quick-JUnit plugin

よく使うショートカット

Ctrl + 1	クイックフィクス
Ctrl + > Ctrl + <	カーソルより前(後)の 警告位置へ移動
Ctrl + J Shift + Ctrl + J	順方向(逆方向) インクリメンタルサーチ
Alt + ↑ Alt + ↓	選択行の上下移動
Ctrl + 0 (※ゼロ)	JUnit Test実行 (Quick-JUnit plugin)

ファイル間の移動も効率的に

Alt + < Alt + >	直前(直後)の 編集位置に移動
Ctrl + E	現在エディタで開いている ファイル一覧から選択
Shift + Ctrl + T	型検索
Shift + Ctrl + R	リソース検索
Ctrl + 9	テスティングペア間の移動 (Quick-JUnit plugin)

リファクタリングもショートカットで

- キーバインドがリファクタリング名に関連づいています

Alt + Shift + R	改名(Rename)
Alt + Shift + M	メソッドの抽出(extract Method)
Alt + Shift + L	ローカル変数の抽出 (extract Local variable)
Alt + Shift + I	インライン化(Inline)
Alt + Shift + C	メソッドシグニチャの変更 (Change method signature)
Alt + Shift + T	その他のリファクタリングのメニュー

と、いうわけで、

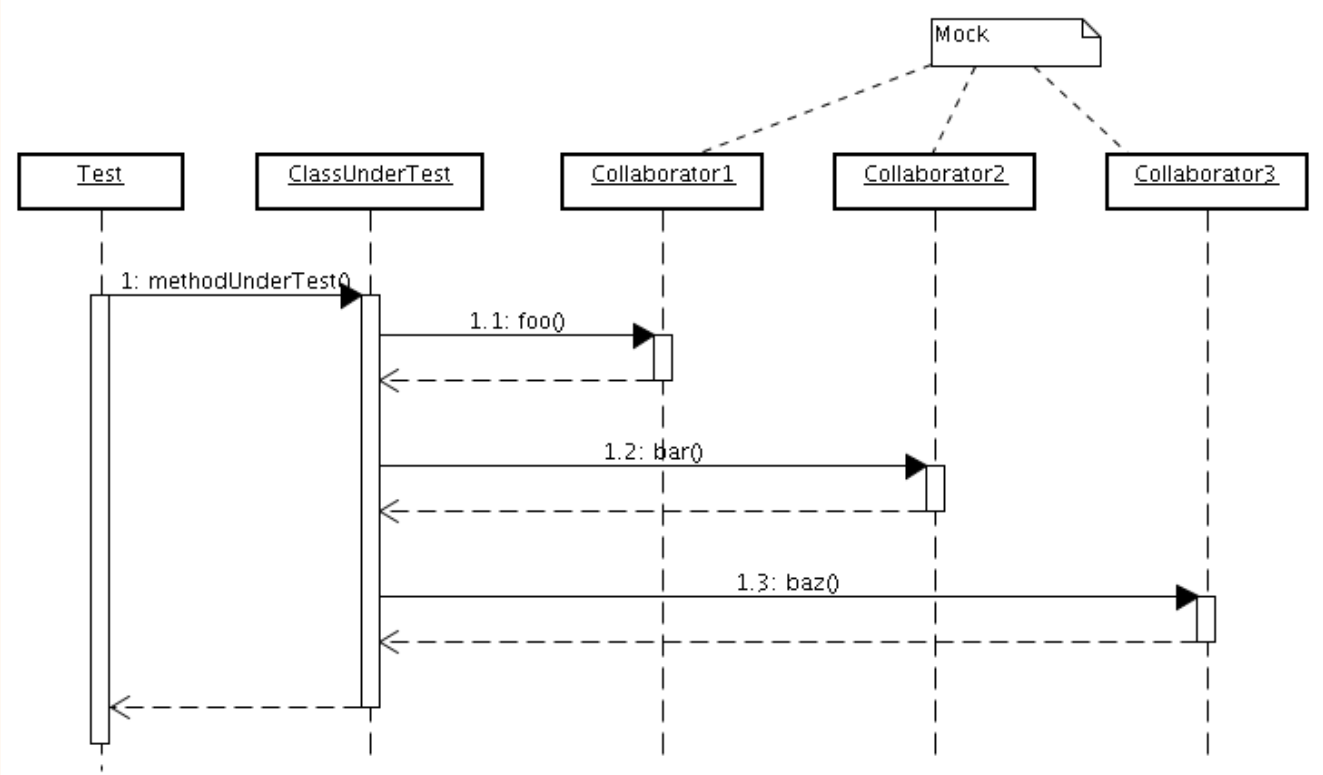
TDD実演デモ

DI時代の実践的TDD

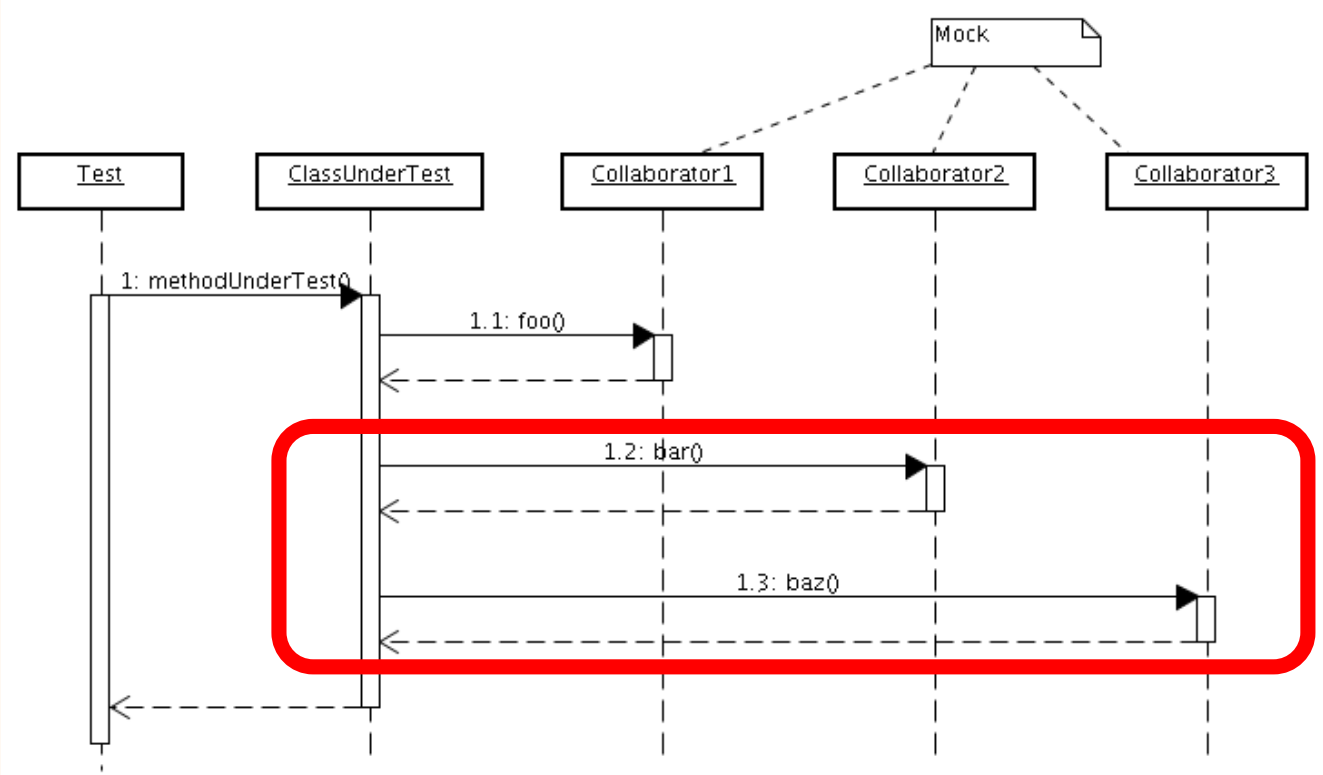
DIがもたらしたインパクト

- DI前の時代
  - オブジェクトの生成を「誰か」が行う必要があった。
    - 単純にnewすると結合度が高く、テストが難しい
    - Factoryに切り出して、テストではFactoryを切り替える戦略を採ることが多かった
    - Factoryが「ものしり」「有名人」になっていた
- DI後の時代
  - DIコンテナが「スーパーファクトリ」であるため、プロダクトコードにオブジェクトの生成知識が不要に
    - テスト時にはモックを簡単に差し込めるようになった
    - 直接のコラボレータをinjectすればいいので、デメテルの法則準拠
    - かつ、DIコンテナの存在はプロダクトコードに現れない

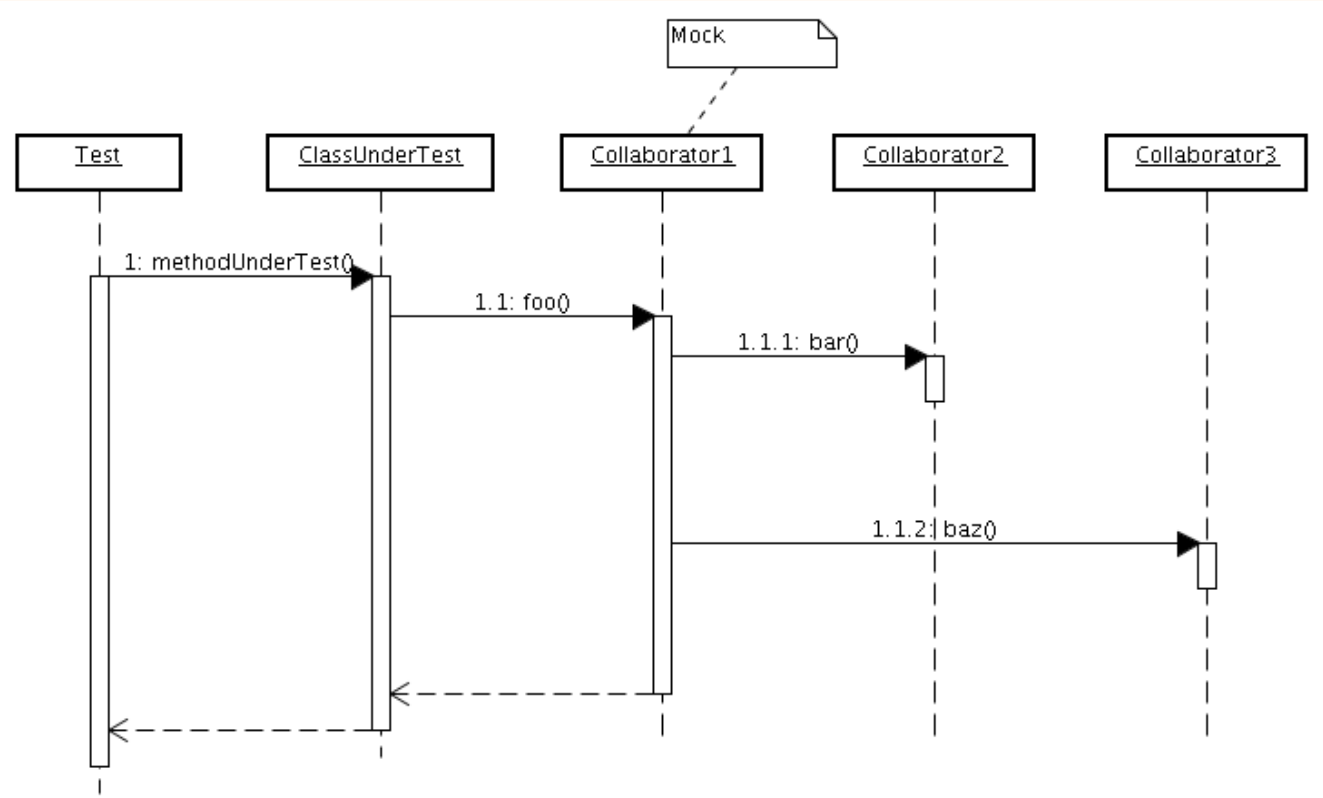
依存が多すぎる例



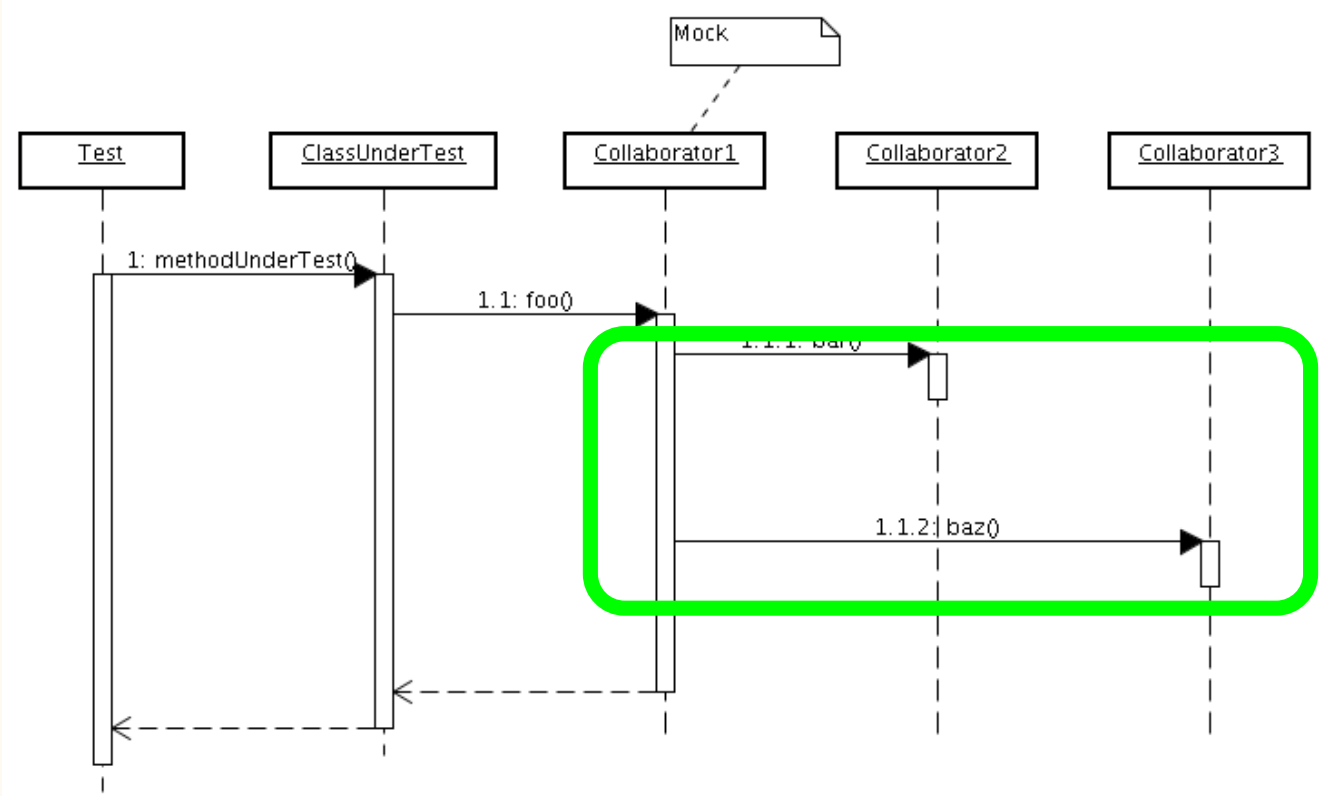
依存が多すぎる例



デメテルの法則に従い、責務を再配置した例



デメテルの法則に従い、責務を再配置した例



DIで可能になった、資産としてのPOJO

- POJO (Plain Old Java Object)
  - 特定のフレームワークやAPIに依存しない「普通の」Javaオブジェクト
  - ピュアなロジック
- 資産とアーキテクチャの分離
  - 背後の仕組みやアーキテクチャに依存しない
- 「POJOであること」が本当の資産化
  - POJOとJUnitが長く残る資産

テストの自己目的化に気を付ける

- 「やらずもがな」なテストはただのコスト
  - getter/setter
  - 右を左への委譲コード
- テストを書くときに考えること
  - そのテストの目的は何か
  - 不安を克服する材料となるか
  - 意図を伝えているか

ではどこをテストするのか

- 条件分岐
- ループ
- 値の加工
  - テストで入れた値がそのまま返ってくるようなテストは何も加工していないと言える
    - 単純な構造ならばテストの必要はないかもしれない
  - 入れてポン、出してポン
- バリデーション
- 状態遷移

低コストの機能テスト方法を探す

- 「実行コスト」が高いからモックを使う
  - コンテナやデータベースのセットアップはコストがかかる
  - テストの実行が遅いとテスト頻度が下がる
  - モックは妄想ベースだが、実行が速い
- ではもし機能テストのコストが低かったら?
  - Jetty
  - H2, Derby
  - ActiveMQ, OpenJMS

おわりに

テストの資産価値

- テストは資産であり、資産価値がある
- どのテストもメンテナンスコストはかかる
- リファクタリングと資産価値
  - リファクタリングの邪魔になるようなテストは価値が低い
  - テストはリファクタリングの支えになるものであって、妨げになるものではない
- 半年後であっても読みやすく、リファクタリングの支えになるテストは資産価値が高い

再チャレンジ可能な世界へ

- 設計の良し悪しは、最初に設計したときに決まるわけではない
- だんだんと「より良い」設計に変えていくことが出来る

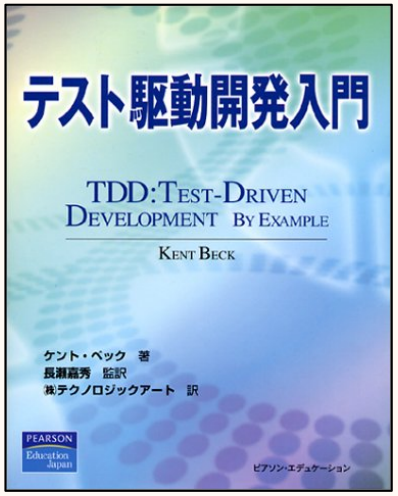
- そのためには…
  - シンプル設計
  - リファクタリング
  - リファクタリングを支える、資産価値の高いテスト

TDDはスキルです

- TDDはスキルです

  と、いうことは…

  - 才能ではない
  - 習得可能
  - 量は質に転化する
  - 写経してみましょう



今日のまとめ

- TDDのテストは品質保証のためではなく、開発促進のためのツール
- テストには粒度があり、資産価値がある
- DIコンテナによってTDDがやり易くなった
- Eclipseを使いこなすべし!

Special Thanks

- かくたにさん、チームかくたにのメンバー
- JavaEE勉強会世話人、今日の司会でもある角田さん
- kdmsnrさん (bliki-jaに今回もお世話になりました)
- yuguiさん(詳細なレポートありがとうございます)
- 溝口八郎右衛門さん
- TDDの父、Kent Beck氏
- 技術評論社の編集者の方々(特にIさん、Sさん)

- 会場にお越しくださった皆様

- そして、masarlさん

Enjoy Testing!

ご清聴
ありがとう
ございました