

#tdd4ec is back !

～テスト駆動開発による
組み込みプログラミングの集い～

大西洋平 @legoboku

2013/5/11 (土)

今日のスケジュール

- 14:00 - 14:30 インTRODクシヨソ
- 14:30 - 15:00 ライトニングトーク
- 15:45 - 17:00 TDDワークシヨツプ
- 17:00 - 17:45 振り返り
- 17:45 - 18:00 クロージング & 撤収
- 18:00 ～ 懇親会

イントロダクション

- 自己紹介をお願いします。
- 以下の質問の解答を1枚ずつ付箋に書いてください。振返りのネタにします。
- なぜTDDに関心を持ったか？
- TDDについて何か疑問は？

TDD ワークショップ

- 解説
 - TDDとは何か？
 - なぜTDDに取り組むか？
- 演習（ライトスケジューラ）
 - TDDのやり方と解説含む

TDDとは何か？

- 高品質なソフトウェアをインクリメンタルに開発する設計技法
- 要求をテストで表現し、テストに成功するようプロダクトコードを書く
- 必ず自動化された回帰テストとプロダクトコードが対となって成長する

なぜTDDに取り組むのか？

(注：大西の所感です)

- TDDを通じてありたい姿は「安定した開発を持続させる」こと
- そのために
 - 成果物の品質を安定させ続けること
 - 変化に追従できる状態を保つこと

安定した開発を持続させる

- 成果物の品質を安定させ続けるには
 - バグの未然防止（=デバッグの防止）
 - 回帰テストのコスト低減
- 変化に追従できる状態を保つには
 - 変更コストの低減

なぜバグの未然防止？ (= デバッグの削減)

- 第一の目的は開発時間を浪費するデバッグを減らすこと
- 「バグをテスト工程で検出する」ではなく「そもそもバグを埋め込まない」

あとでデバッグ VS TDD

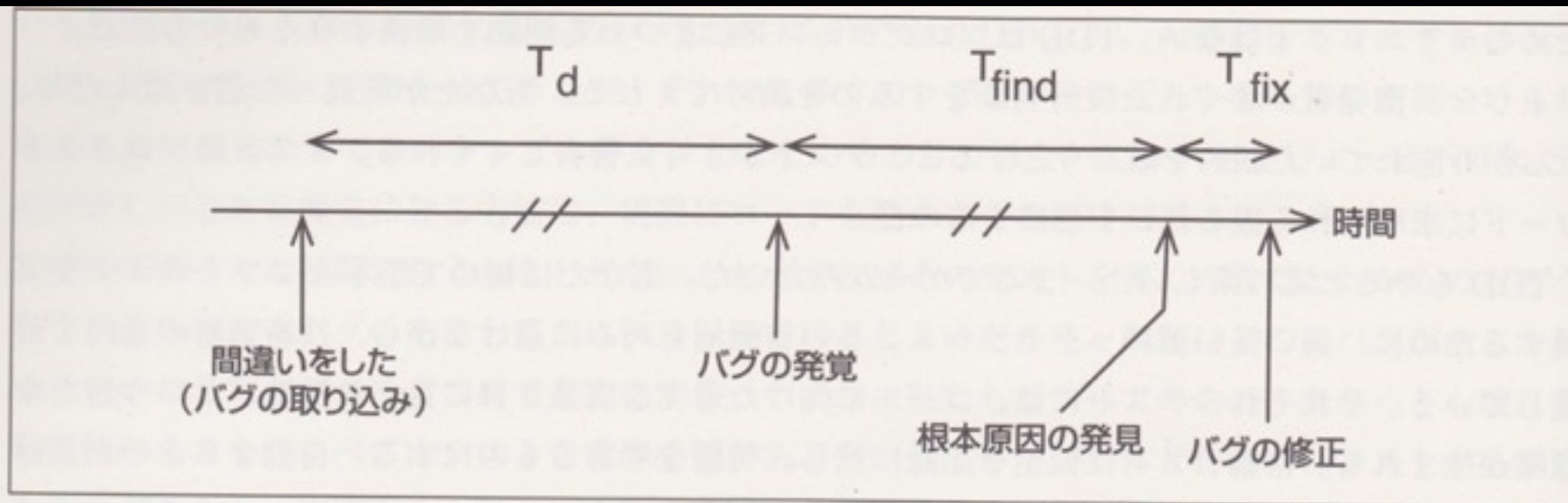


図 1-1 あとでデバッグ型プログラミングの特性

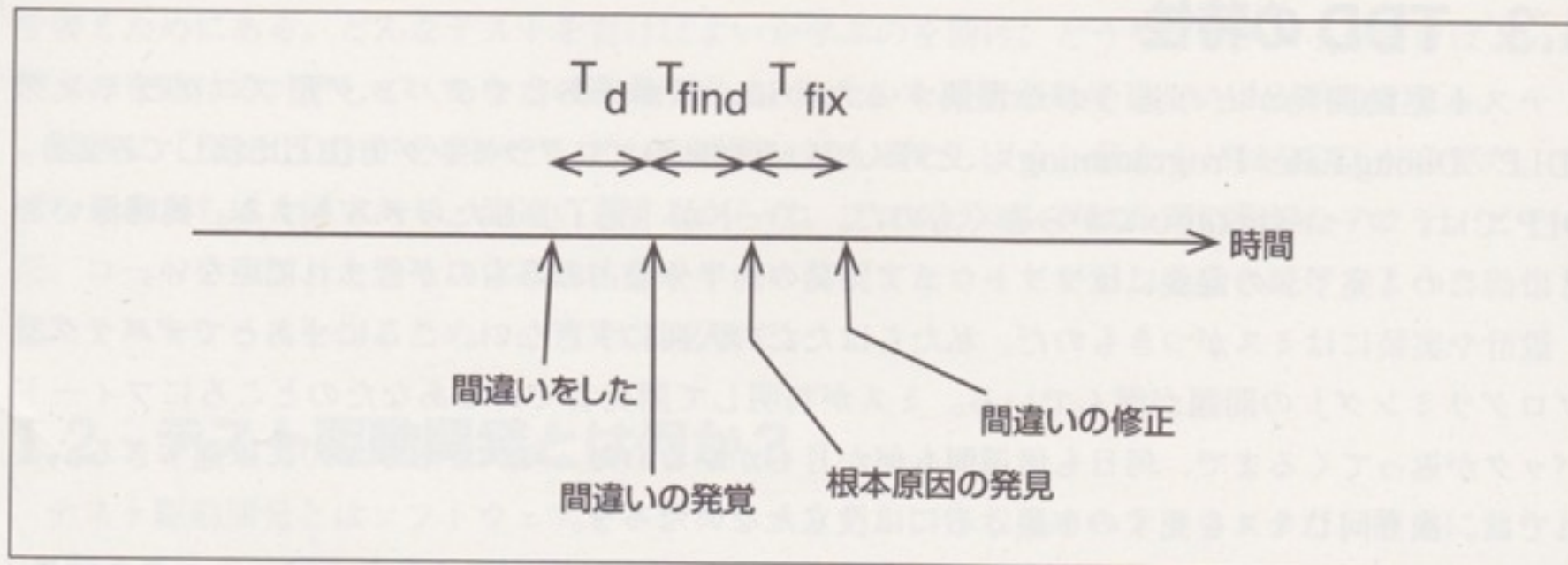
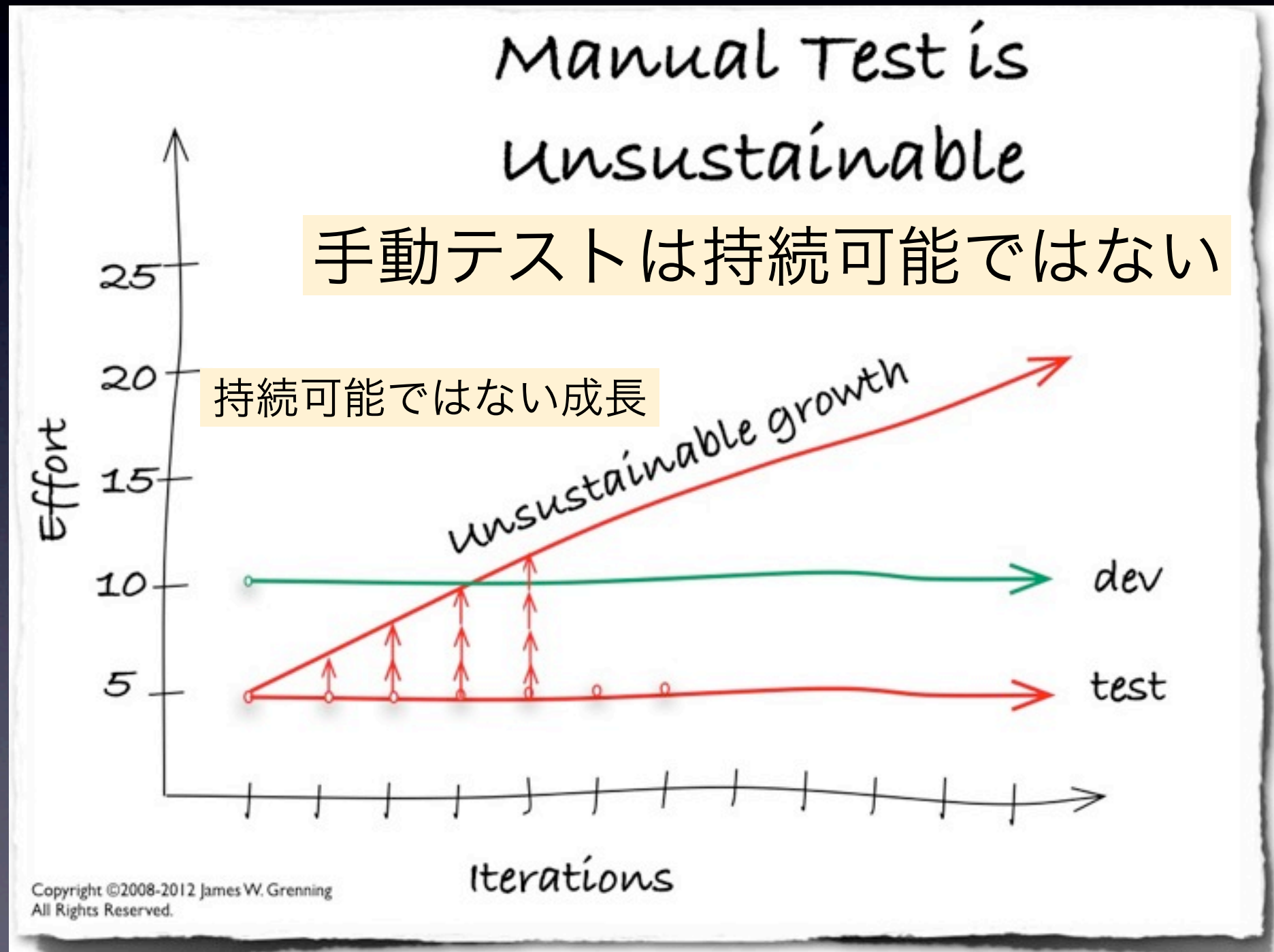


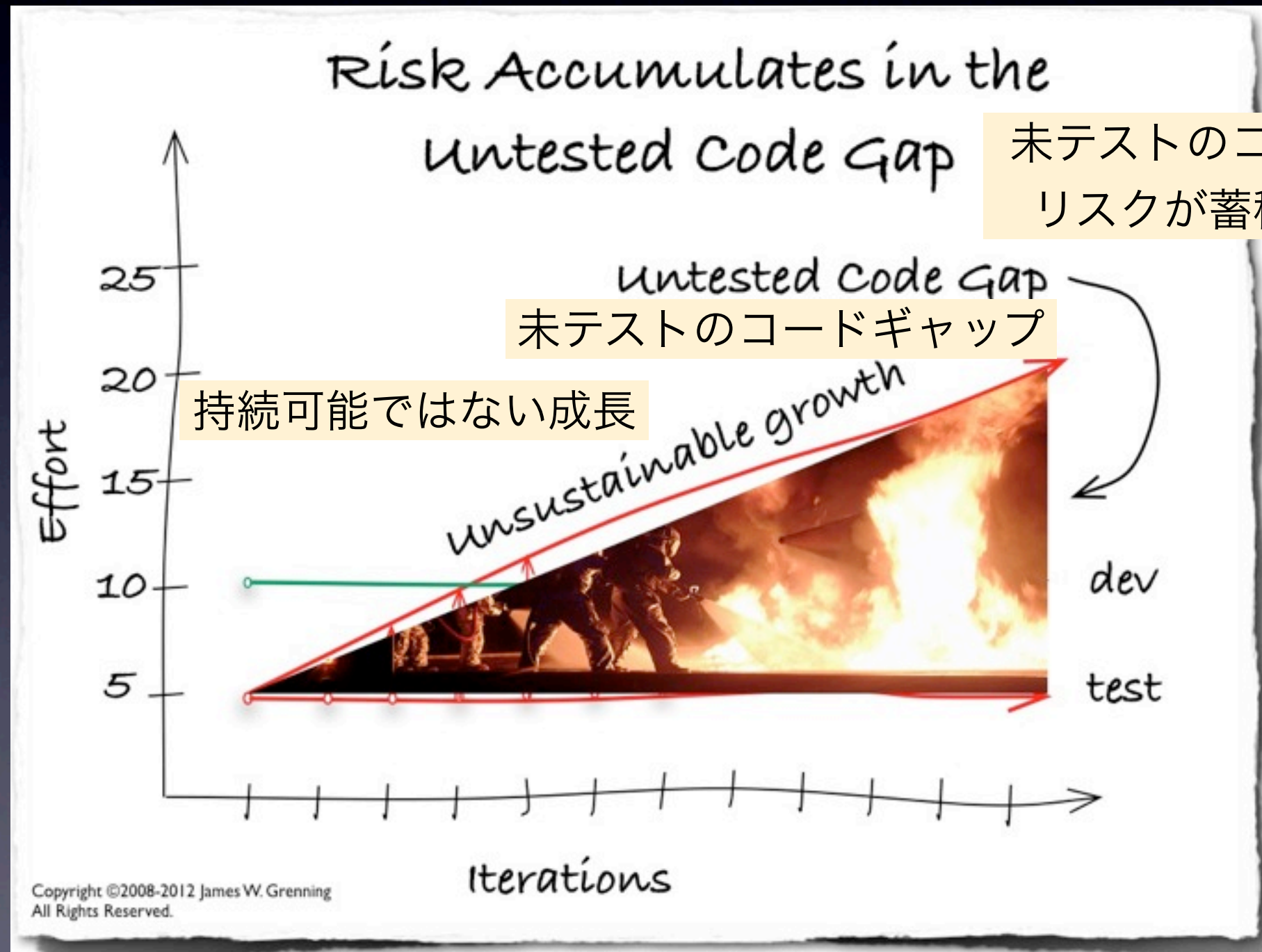
図 1-2 テスト駆動開発の特性

「テスト駆動開発による組み込みプログラミング」 P6より引用

なぜ回帰テストのコスト低減？



なぜ回帰テストのコスト低減？



なぜ変更コストの低減？

- 回帰テストなしの設計変更はリスクが高く「テストした所は触るな」に陥る
- テスト済の箇所の修正を迂回する設計変更が続くとコードの保守性が下がる
- 保守性が下がると、結局、開発コストが増大し俊敏さが失われる

なぜ変更コストの低減？

- 理想的には長期的な変更コストを低減するため、継続的に「テストで保護して」リファクタリングする
- そのための安全装置として自動回帰テストを整備する

演習 LightScheduler

- 8章「プロダクトコードをスパイする」
の例「LightScheduler」をテスト駆動で
実装する
- TDDのやり方は実演しながら説明

LightSchedulerの概要

- ホームオートメーションの一部
- 予めスケジュールに登録された日時で
蛍光灯を点灯・消灯する機能を実現
- （例）2013年5月6日朝8時に点灯

演習：残りを実装する

ライトスケジュール機能のテスト

初期化でライトは変化しない

曜日が違っていて、時刻が違っているとき、ライトは変化しない

曜日が合っていて、時刻が違っているとき、ライトは変化しない

曜日が違っていて、時刻が合っているとき、ライトは変化しない

曜日が合っていて、時刻が合っているとき、適切なライトがオンされる

曜日が合っていて、時刻が合っているとき、適切なライトがオフされる

毎日スケジュールする

特定の日にスケジュールする

平日すべてにスケジュールする

週末にスケジュールする

スケジュールしたイベントを取り除く

存在しないイベントを取り除く

同時に複数のイベントをスケジュールする

同一ライトに複数のイベントをスケジュールする

スケジュールしていないライトスケジュールを取り除く

サポートするイベントの最大数(128)をスケジュールする

多すぎるイベントをスケジュールする

何もスケジュールされていなければ、内イベントがあっても何もオンされない

特に上から6つ
までに取り組ん
でみましょう

「テスト駆動による組み
込みプログラミング」

P139図8-5より引用

設計

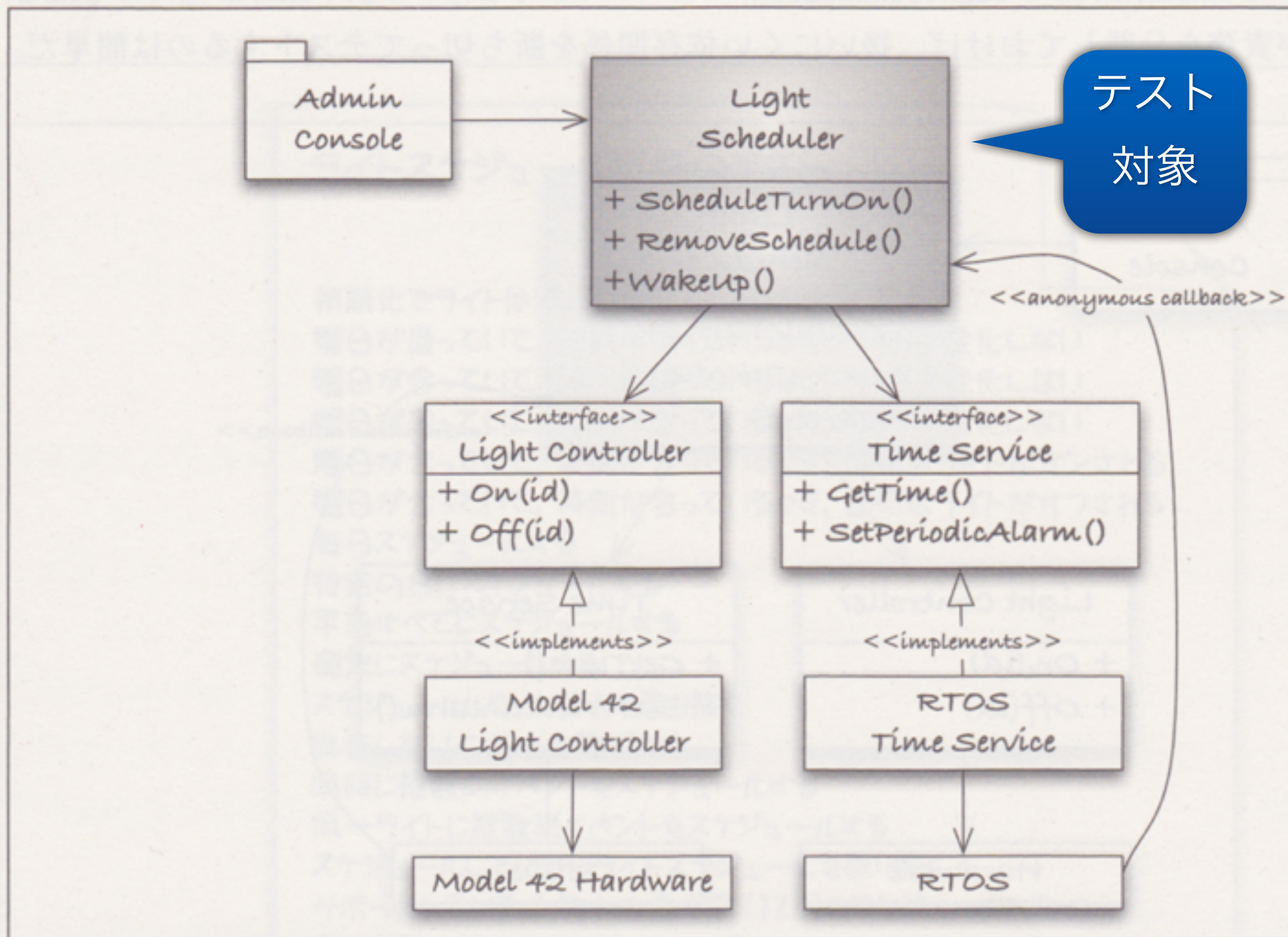


図 8-3 LightScheduler はインターフェイスを通してコラボレータに指示する

テスト設計

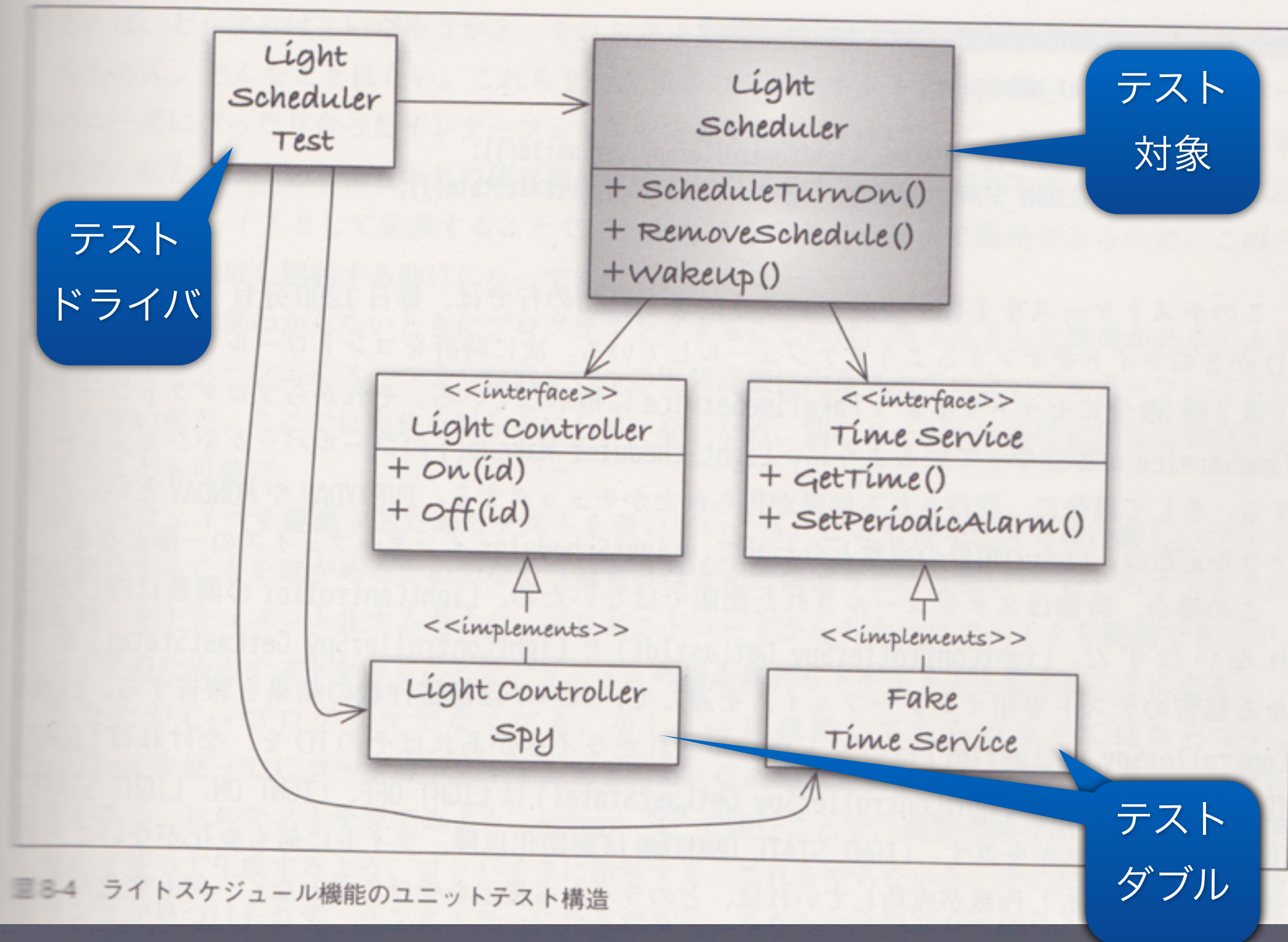


図8-4 ライトスケジュール機能のユニットテスト構造

テスト設計のポイント

- 事前条件はドライバとテストダブル
（テスト用の偽物）から与え、処理結果もテストダブルで受け取る。
- 入口・出口をテストダブルに置換できる設計にしなければならない。

Cでユニットテストを 書く際のポイント

- 基本：リンク時置換
 - インタフェースをhファイルに、実装をcファイルに分離。リンク時にテスト用の実装へ置換する。
- 応用：関数ポインタ、プリプロセッサ

実装を確認

- Makefile : CppUTest含めたビルド指示用
- Makefile_tdd4ec : LightSchedulerビルド用
- test : テストコードおよびテストダブル
 - AllTest.cpp : テストコードのmain関数
 - XXXTest.cpp : 各モジュールのテスト
- src : プロダクトコード

TDDのマイクロサイクル

- 小さなテストを追加する。
- 全てのテストを実行し、新しいテストが失敗すること、あるいはコンパイルすらできないことを確認する。
- テストを成功させるのに必要な小さな変更をする。
- 全てのテストを実行して、新しいテストが成功することを確認する。
- リファクタリングすることで、重複をなくして表現を改善する。

テストの実行方法

- Visual Studioの場合
 - mvsc/tdd4ec/tdd4c.slnを開いて「ソリューションをビルド（F7）」実行
 - デバッグなしで実行（Ctrl+F5）
- make + gccの場合
 - トップディレクトリで「make」