#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 イントロダクション
- 14:30 - 15:00 ライトニングトーク
- 15:00 - 17:00 TDDワークショップ
- 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。
  - 名前、普段の仕事、etc.
- 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）
  - 題材の解説
  - TDDのやり方

TDDとは何か？

- テストを活用して高品質なソフトウェアを段階的に開発する設計技法
- 絶えず動く状態を保ちながら、小さいクリーンなコードから大きなクリーンなコードに育てる方法

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「変化に強く、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

変化に強く、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）

- TDDの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 by James
- 「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

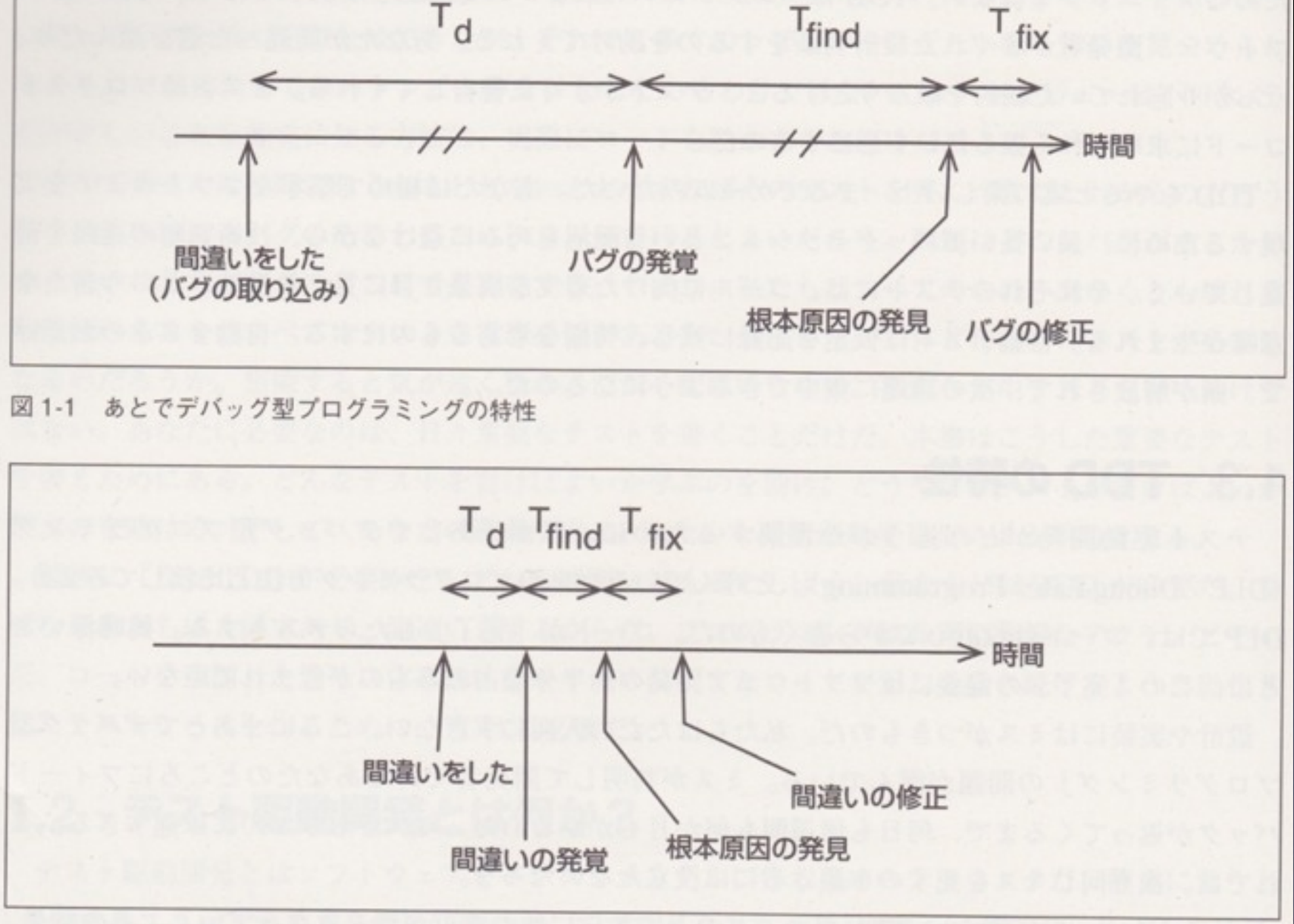
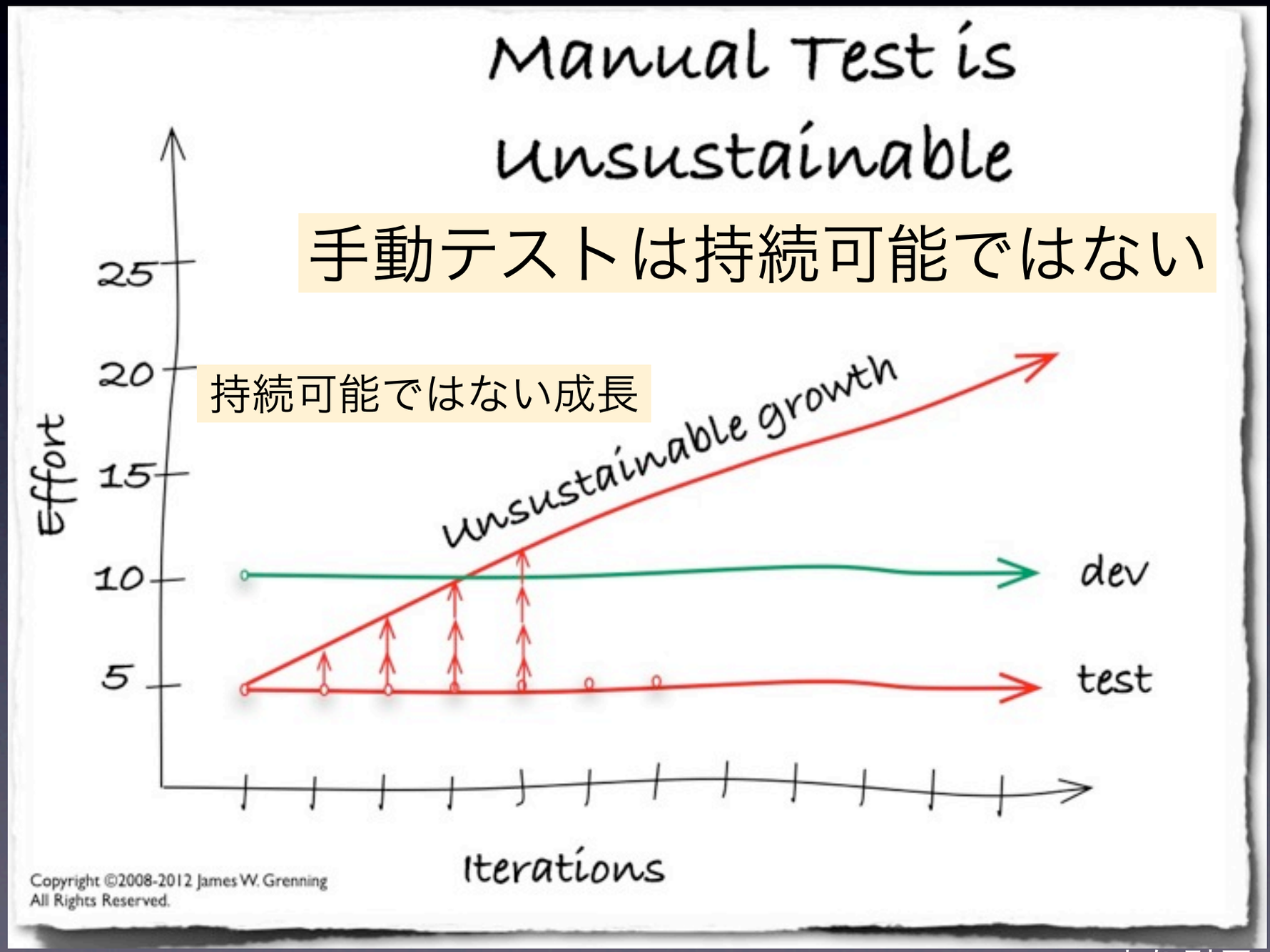


図 1-1　あとでデバッグ型プログラミングの特性

図 1-2　テスト駆動開発の特性

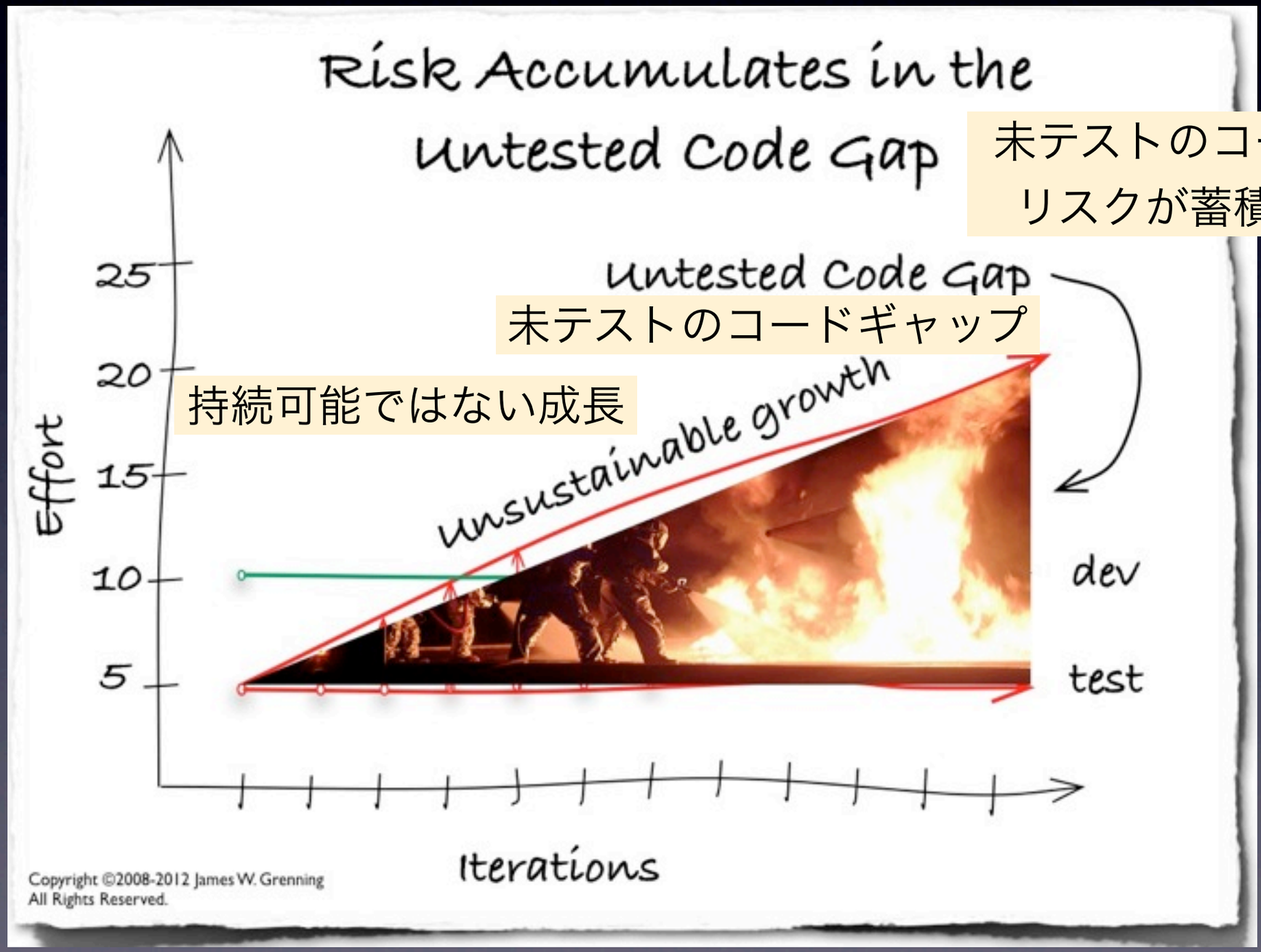
「テスト駆動開発による組み込みプログラミング」 P6より引用

なぜ回帰テストのコスト低減？



http://www.renaissancesoftware.net/blog/archives/206 より引用

なぜ回帰テストのコスト低減？



http://www.renaissancesoftware.net/blog/archives/206 より引用

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

なぜそんなこと気にする？

- 経営の視点
  - 画期的な競合製品が出た時に対応できない
- 人材活用の視点
  - 若手が成長する場の喪失
  - 技術的負債に無頓着なおっさんの増加による技術の空洞化

柴田 芳樹さんのブログより yshibata.blog.so-net.ne.jp/2013-05-10

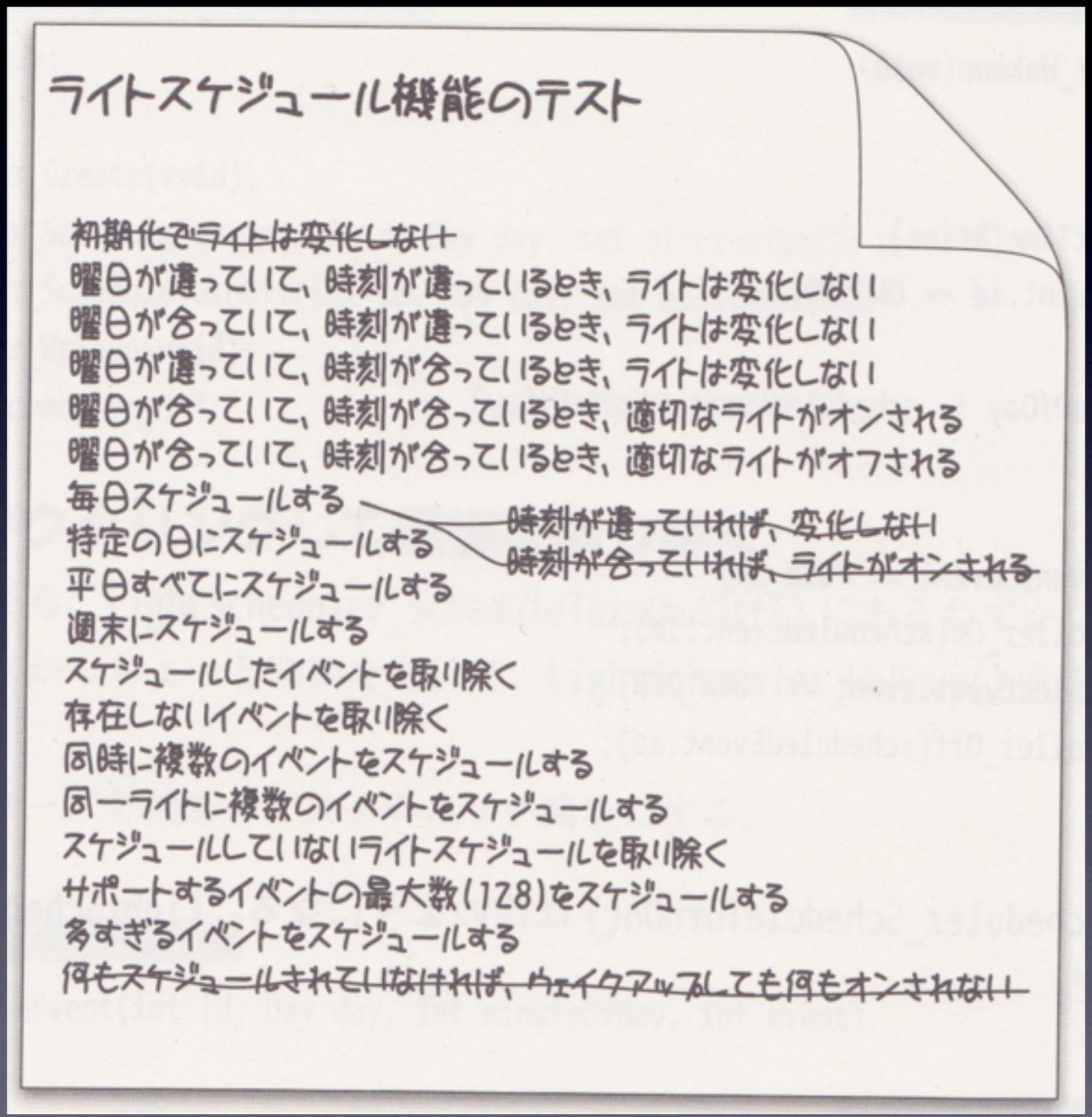
演習 LightScheduler

- 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



・まずは単一イベントにチャレンジ

・テストコードとプロダクトコードのリファクタリングを忘れずに

「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

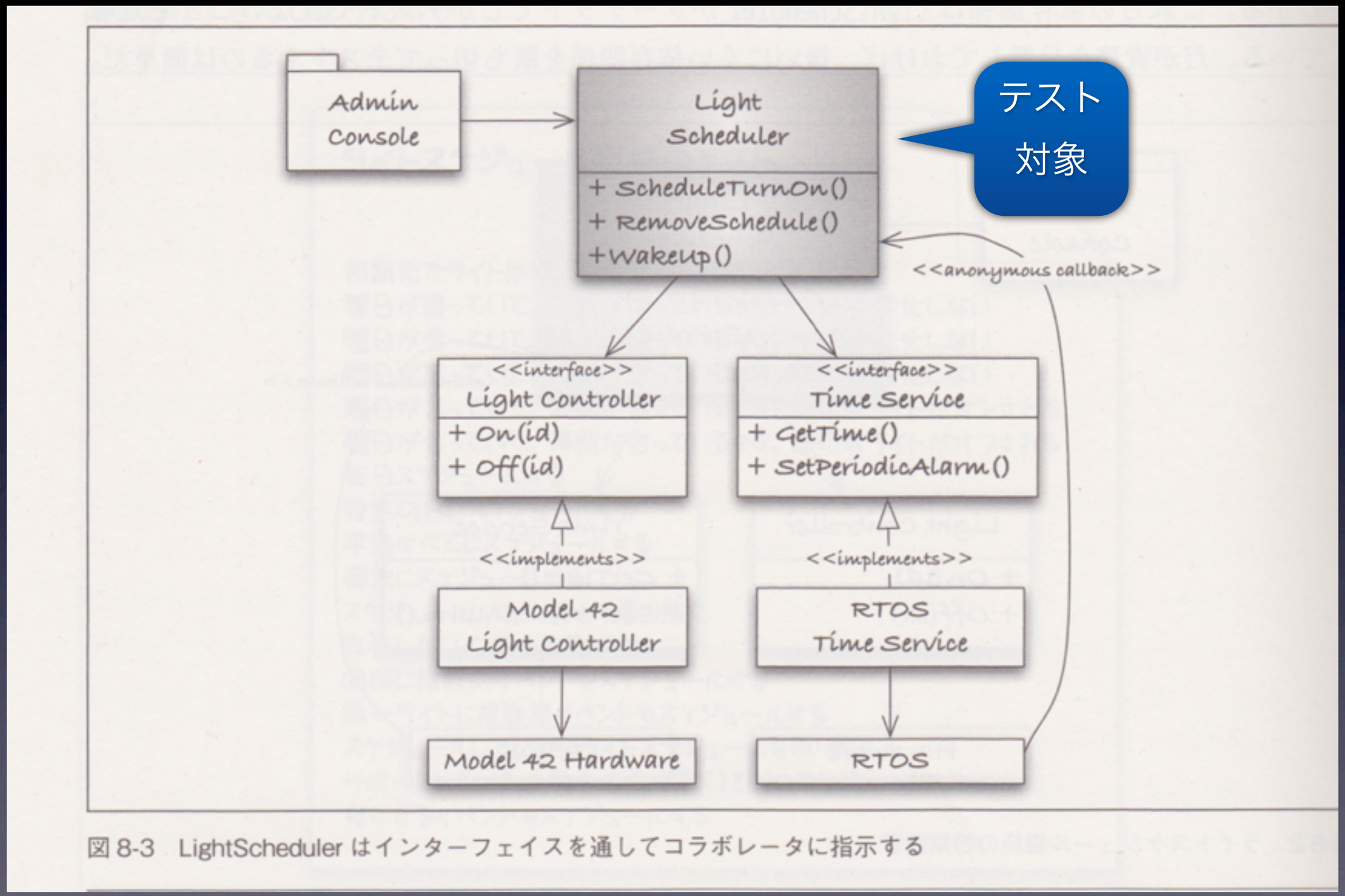


図 8-3　LightScheduler はインターフェイスを通してコラボレータに指示する

「テスト駆動開発による組み込みプログラミング」P128 図8-3より引用

テスト設計

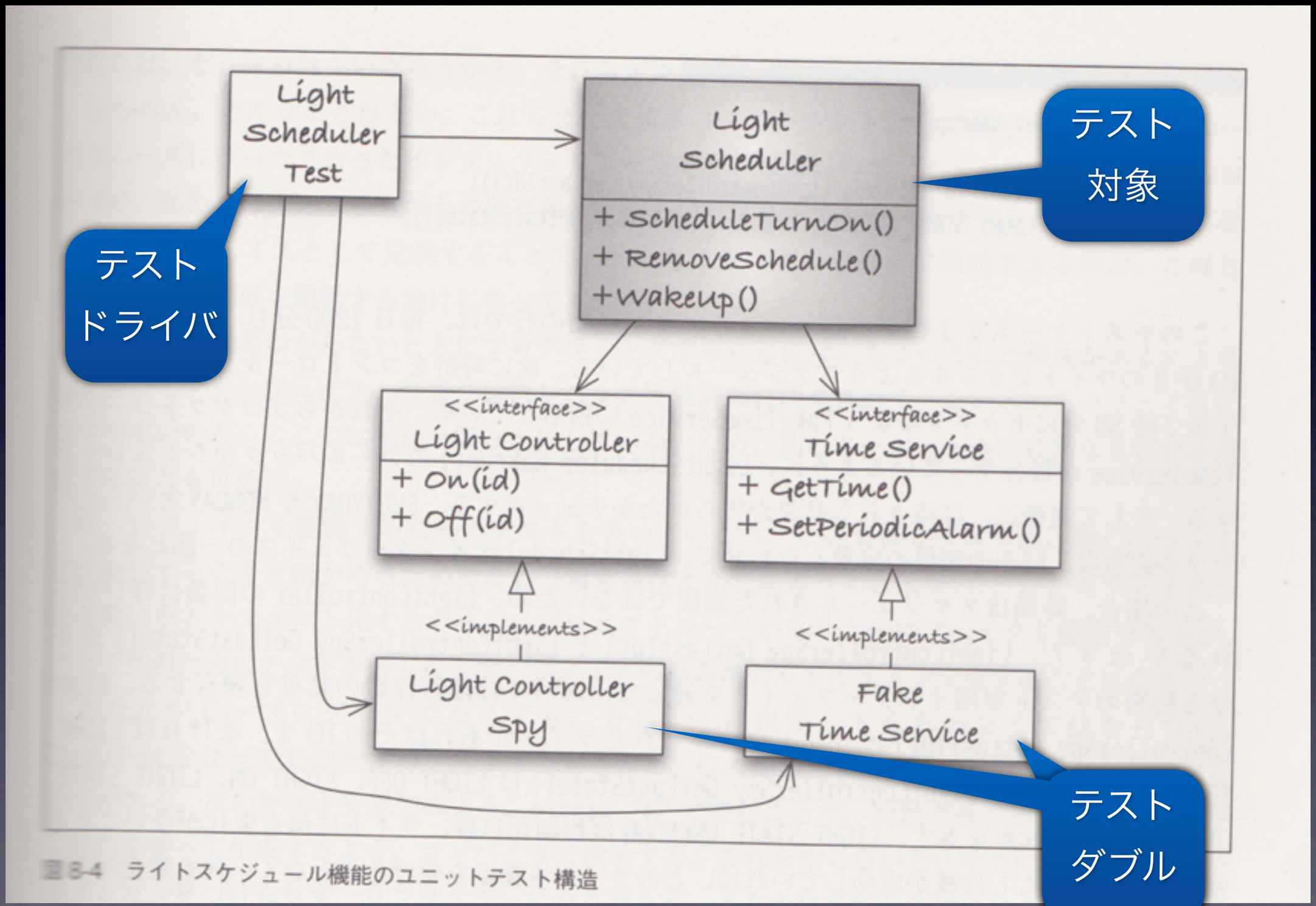


図8-4　ライトスケジュール機能のユニットテスト構造

「テスト駆動開発による組み込みプログラミング」P129 図8-4より引用

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

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」

ペアプロ＋TDDのやり方

- ペアプロ
  - ナビゲータ・ドライバ
    - ピンポン
- TDD
  - マイクロサイクル
  - ボブ・マーティンのTDD三原則

ペアプロのやり方

- ナビゲータ・ドライバ
  - 一人がドライバとなり、プロダクトコードとテストコードの両方を書く
  - もう一人はナビゲータとなり、問題の整理や指示役に徹する
- ピンポン
  - 一人がテストコードを書き、もう一人がプロダクトコードを書く。1回ずつ役を交代。

ボブ・マーティン
TDDの三原則

- 失敗するユニットテストを成功するためのプロダクトコード以外は書いてはいけない。
- 失敗させるのに十分なユニットテスト以外は書いてはいけない。ビルドエラーは失敗に数える。
- 失敗するユニットテスト1つを成功させるのに十分なだけのプロダクトコード以外を書いてはいけない。

「テスト駆動開発による組み込みプログラミング」P39より引用

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。

「テスト駆動開発による組み込みプログラミング」P7より引用

0,1,Nパターン

- コレクションを扱うコードをTDDで実装する場合、いきなりN個の処理を実装しない。
- 最初に0のパターン、次に1のパターンを確認してから、Nのパターンへ。
- 各フェーズでは今その時必要なコードしか書かない。