

テスト駆動開発による 組み込みプログラミングの集い @関西

大西洋平 @legoboku

2013/7/20 (土)

本教材の使用について

- 本教材は使用自由です。社内の勉強会でも自由に使用ください。
- 教材・サンプルは以下のURLからダウンロードできます。
- <https://github.com/yohei1126/tdd4ecisback/archive/master.zip>

今日のスケジュール

- 14:00 - 14:15 概要説明
- 14:15 - 14:30 ライトニングトーク
- 14:30 - 15:00 TDDの概要説明およびデモ
- 15:00 - 17:00 TDDワークショップ
- 17:00 - 17:30 振り返り
- 17:30 - 17:45 結果発表
- 17:45 - 18:00 お片づけ
- 18:30 ~ 懇親会

TDDの概要説明およびデモ

- ・TDDとは何か？
- ・なぜTDDに取り組むか？
- ・演習題材の解説
- ・TDDのデモ

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手法
- ・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

TDDの概要説明およびデモ

- TDDとは何か？
- • なぜTDDに取り組むか？
- 演習題材の解説
- TDDのデモ

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」

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

あとでデバッグ VS TDD

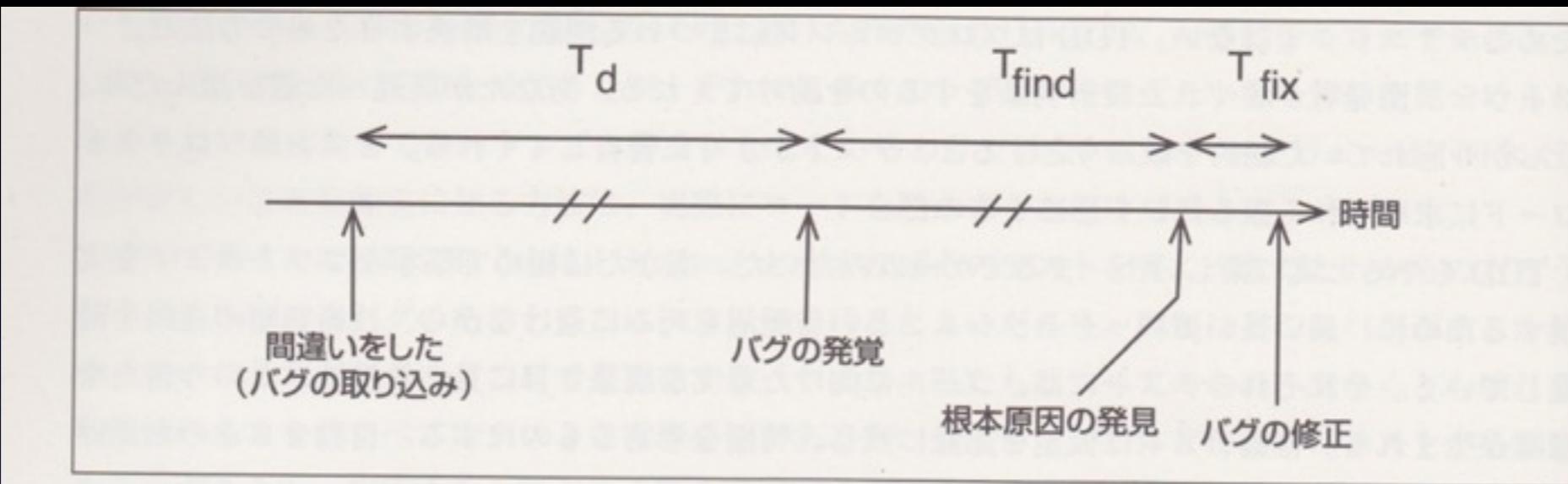


図 1-1 あとでデバッグ型プログラミングの特性

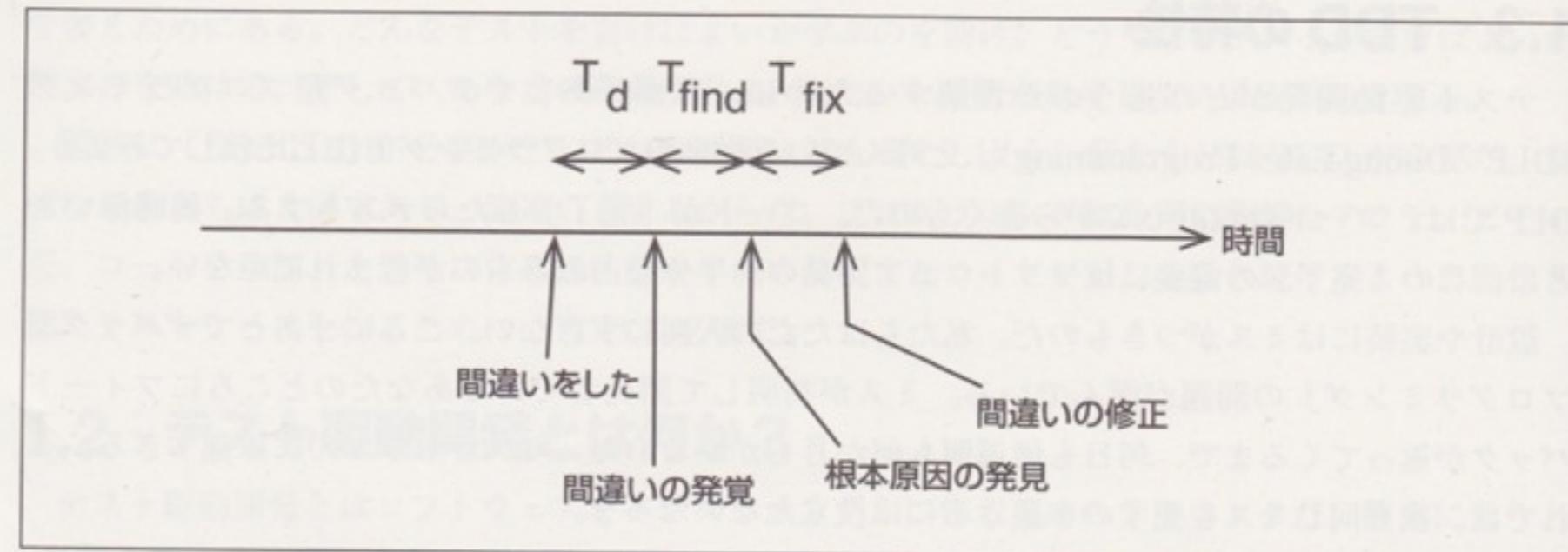
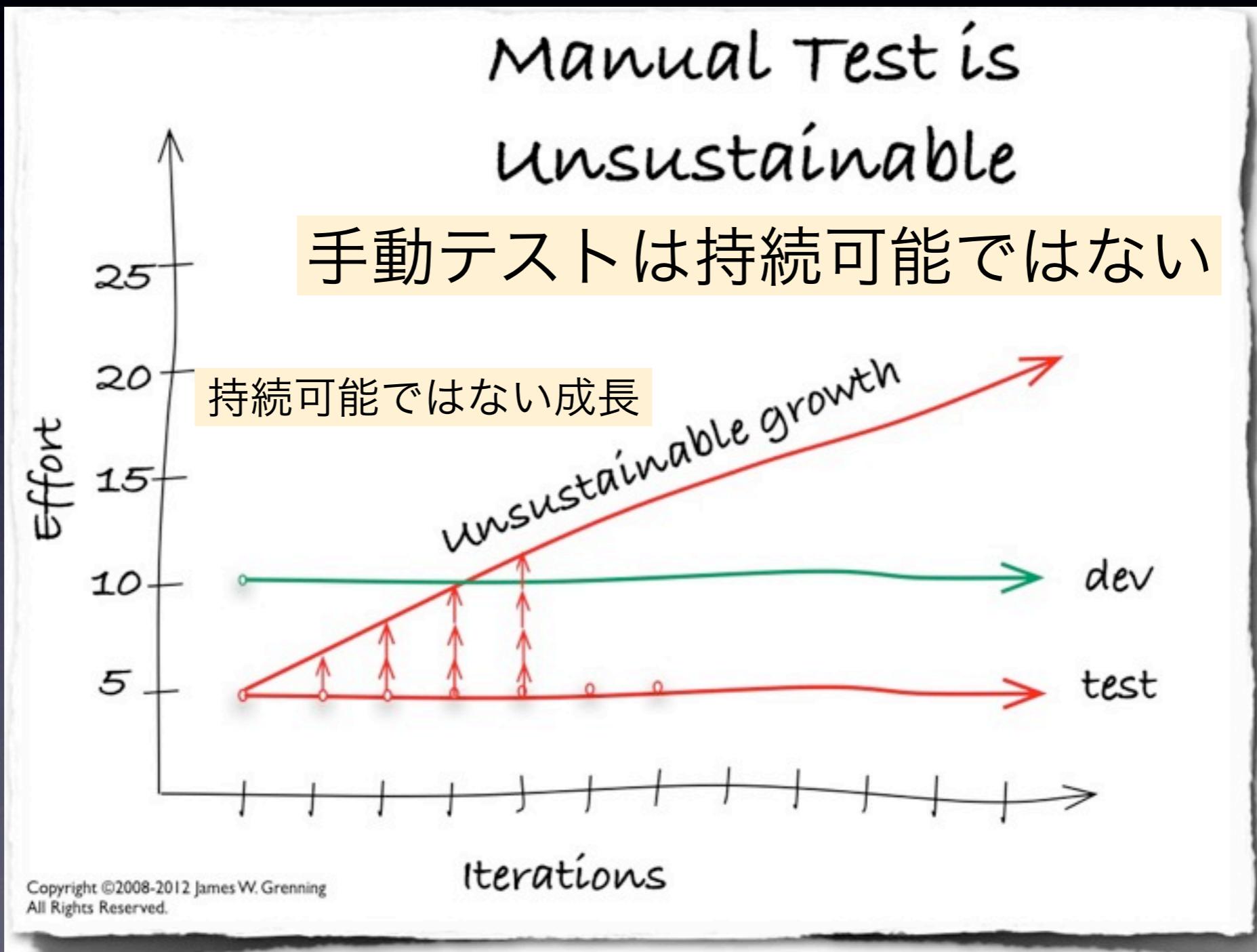


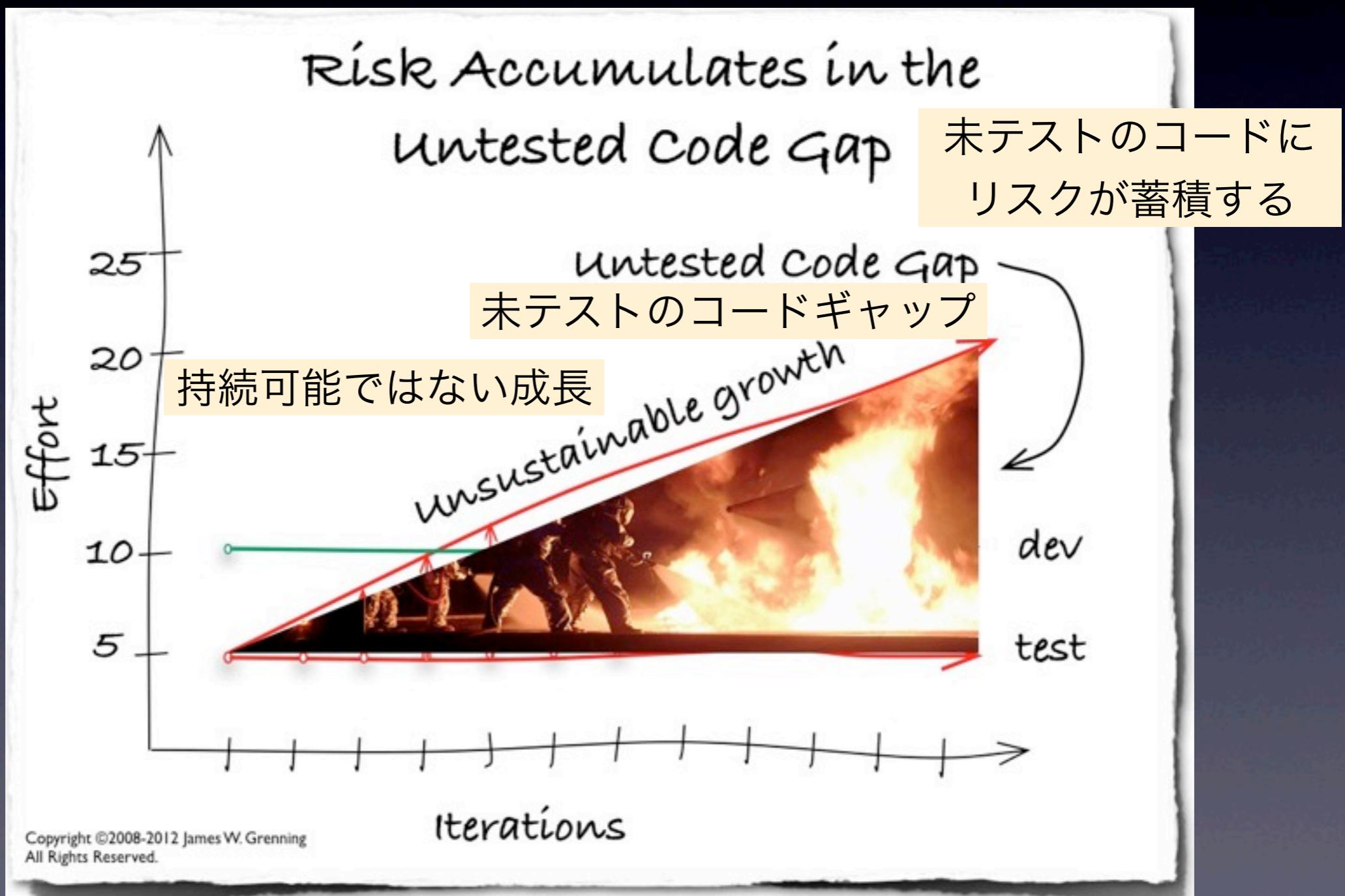
図 1-2 テスト駆動開発の特性

「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？



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？

- 安全装置として自動化された回帰テストを整備する
- 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

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

TDDの概要説明およびデモ

- TDDとは何か？
- なぜTDDに取り組むか？
- • 演習題材の解説
- TDDのデモ

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なければ、ウェイクアップしても何もオンされない~~

- 取消し線はテスト済
- まずは単一イベントにチャレンジ

「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

機能を段階的に追加することに注目

0個
↓
1個
↓
N個

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、ウェイクアップしても何もオンされない

まずはライト
点灯・消灯

複雑なスケ
ジューリング

キャンセル

複数のスケ
ジュール

0,I,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I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。

あとでデバッグ VS TDD

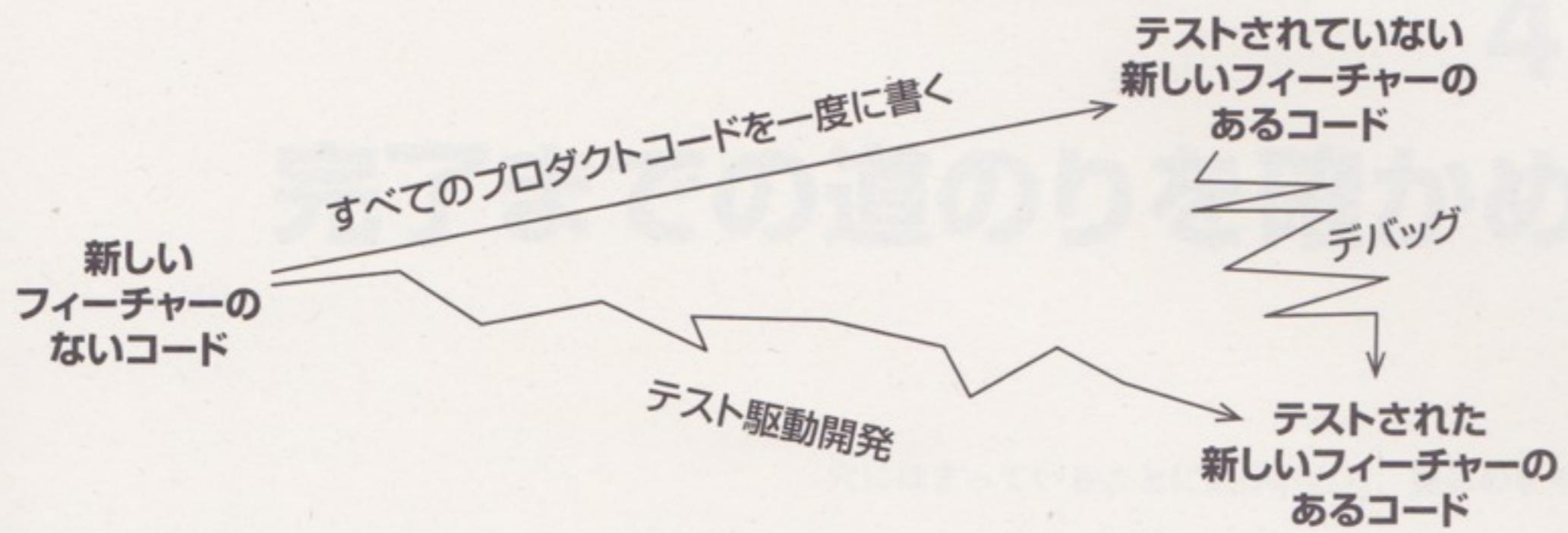
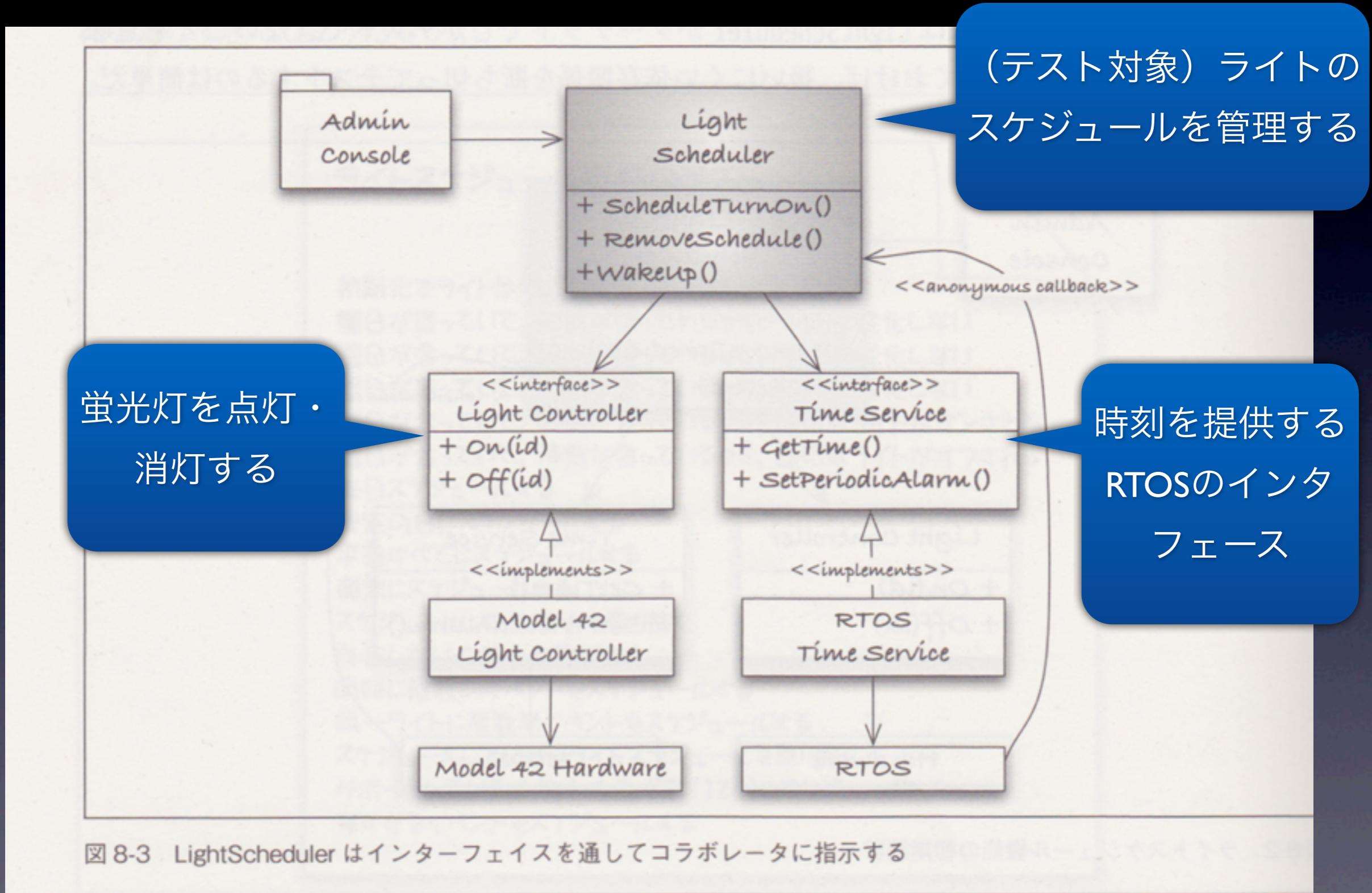


図 3-3 TDD のための踏み石

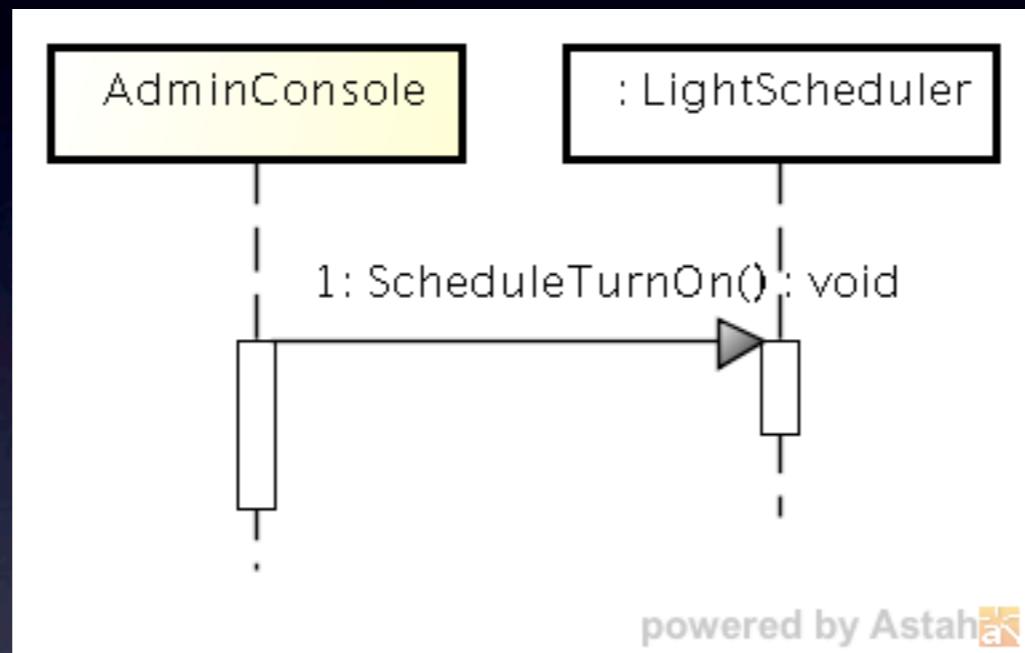
「テスト駆動による組み込みプログラミング」P51より引用

設計

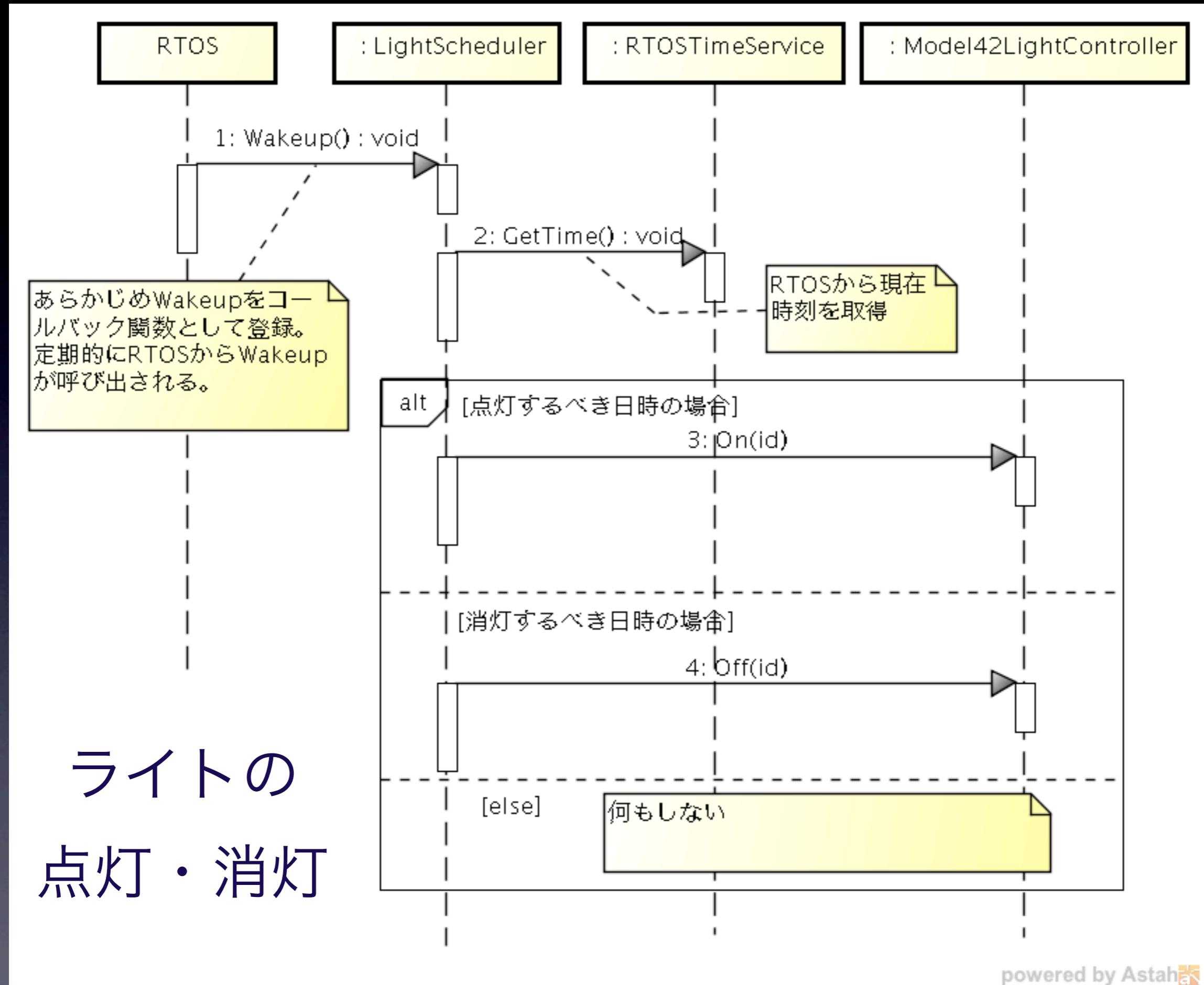


「テスト駆動開発による組み込みプログラミング」 PI28 図8-3より引用

スケジュールの登録

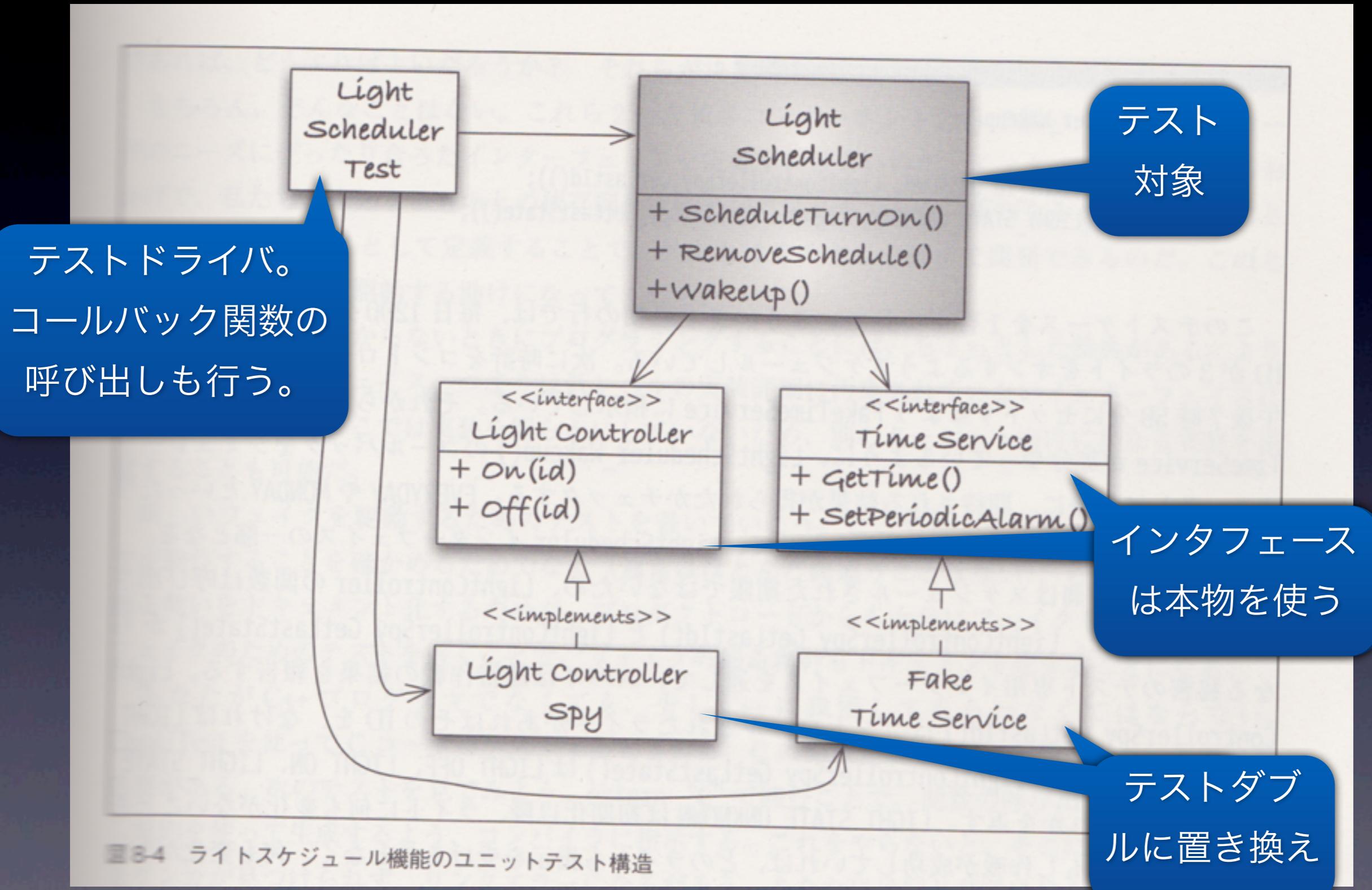


LightSchedulerを利用するサブシステム
からスケジュールが登録される



ライトの
点灯・消灯

テスト設計



「テスト駆動開発による組み込みプログラミング」 PI29 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」

TDDの概要説明およびデモ

- TDDとは何か？
- なぜTDDに取り組むか？
- 演習題材の解説
- • TDDのデモ

TDDのデモ

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、ウェイクアップしても何もオンされない~~

これらへん2~3個
かやってみます！

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。

TDDワークショップ

- ・テストリストに沿って、できる所までTDDでLightSchedulerを開発してください。
- ・ペアプログラミングをやります。ペアを組んでください。
- ・必ずTDDの手順は守って下さい。

ペアプロのやり方

- ナビゲータ・ドライバ
 - ドライバ役がすべてのコードを書く。ナビゲータ役が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く。もう一人がプロダクトコードを書く。1ケースごと交代。

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。

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。