テストは人のためならず

2007/11/11

和田 卓人 (わだ たくと)
タワーズ・クエスト株式会社
プログラマ 兼 取締役社長

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
- コミッタ: いろいろ
- タワーズ・クエストという会社を経営しています

SQL書き方ドリルの「SQUAT」を作った会社です

これまで書いたものとか

- WEB+DB PRESS
  - vol. 35 「実演! テスト駆動開発」
  - vol. 37 「実演! リファクタリング」
  - vol. 42 「REST特集(予定)」
- LifeHacks PRESS
- オープンソースマガジン(リレーコラム)
- コンピュータの名著・古典 100冊 (コラム)

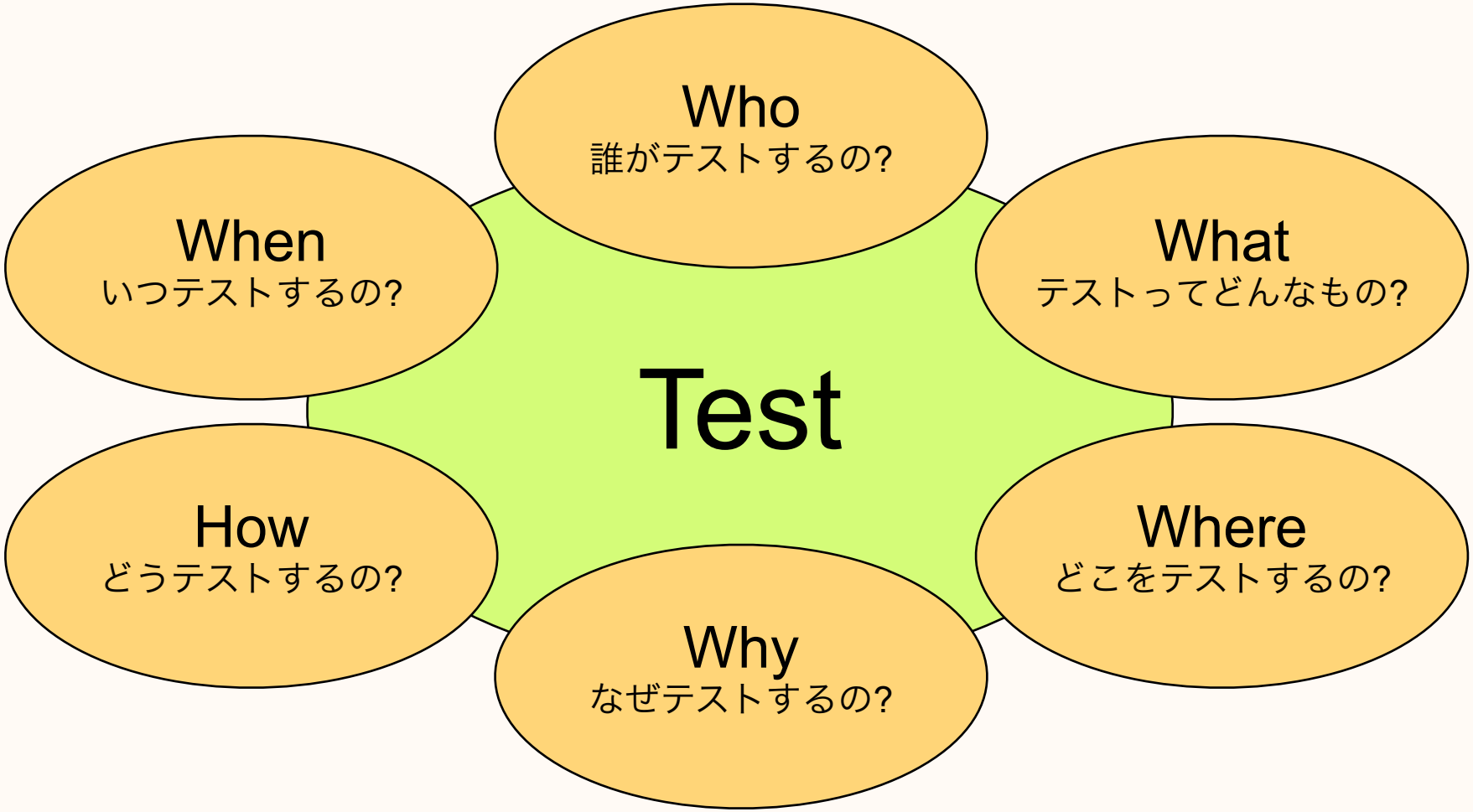
Seasar gihyo.jp

gihyo.jpで連載中

- ［動画で解説］和田卓人の“テスト駆動開発”講座
  - gihyo.jpにてWeb連載を行っています
    http://gihyo.jp/dev/serial/01/tdd/
  - 毎回動画でも解説しています
  - 年内に連載終了まで行く予定です(全20回)
  - お気軽にフィードバックください

- WEB+DB過去記事の特設サイトと動画も健在です

今日の話：テストの5W1H



情けは人のためならず

情(なさ)けは人の為(ため)ならず

人に親切にすれば、その相手のためになるだけでなく、やがてはよい報いとなって自分にもどってくる、ということ

(大辞泉より)

誤答率 48.7%

テストは人のためならず

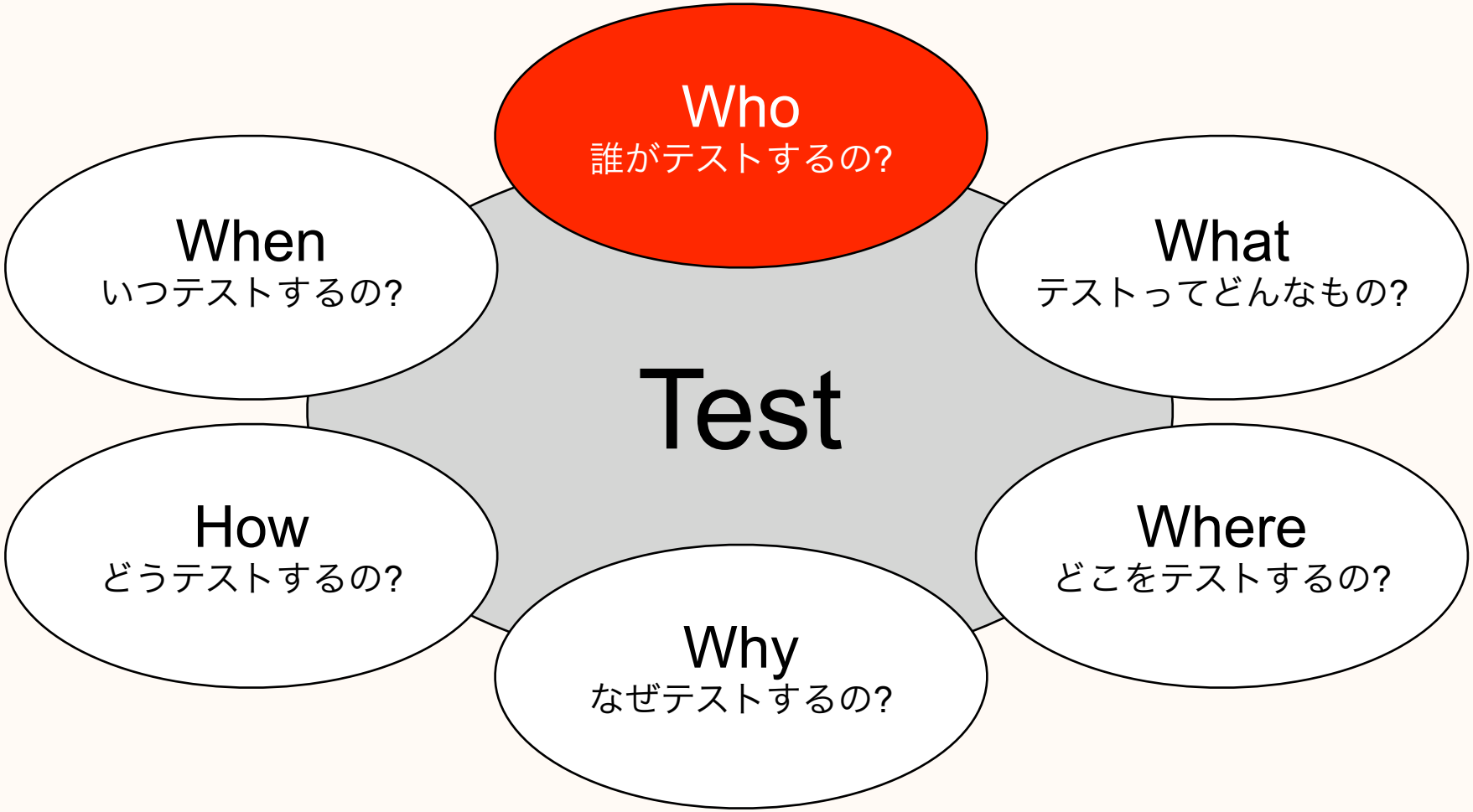
テスト(てすと)は人の為(ため)ならず

テストを理解して書いていくことは、プロジェクトの今後のためになるだけでなく、すぐによい報いとなって自分にもどってくる、ということ

(オレオレ)

誤答率　０％を、目指します

誰がテストするの?



「テスト」という言葉の混乱

単体テスト、ユニットテスト、機能テスト、結合テスト、
受け入れテスト…言葉が多すぎてサッパリわかりませーん

テストは終わらせましたって言ってたクセにバグだらけじゃないか!!
あのさあ、基本的なテストは終わらせてから
リリースしてよねー

「テスト」という言葉の混乱

- 「テスト」という言葉が指すものがバラバラ
  - 単体、ユニット、結合、機能、システム、…
    - たくさんあるけど、何がなにやら…
    - 単体テストとユニットテストって同じもの?
    - テスト範囲による分類には曖昧さ、限界がある
  - 品質保証のためのもの? 動作確認のためのもの?

- 目的に戻って考えてみよう
  - 誰が、何のためにテストを行うのかという視点

「誰が、何のために?」で再分類

- Developer Testing
  - 開発者が行う、開発促進のためのテスト
- Customer Testing
  - お客様と機能の確認のために用いる、進捗管理のためのテスト
- QA Testing
  - 品質保証のためのテスト

「誰が、何のために?」で再分類

「テスト」

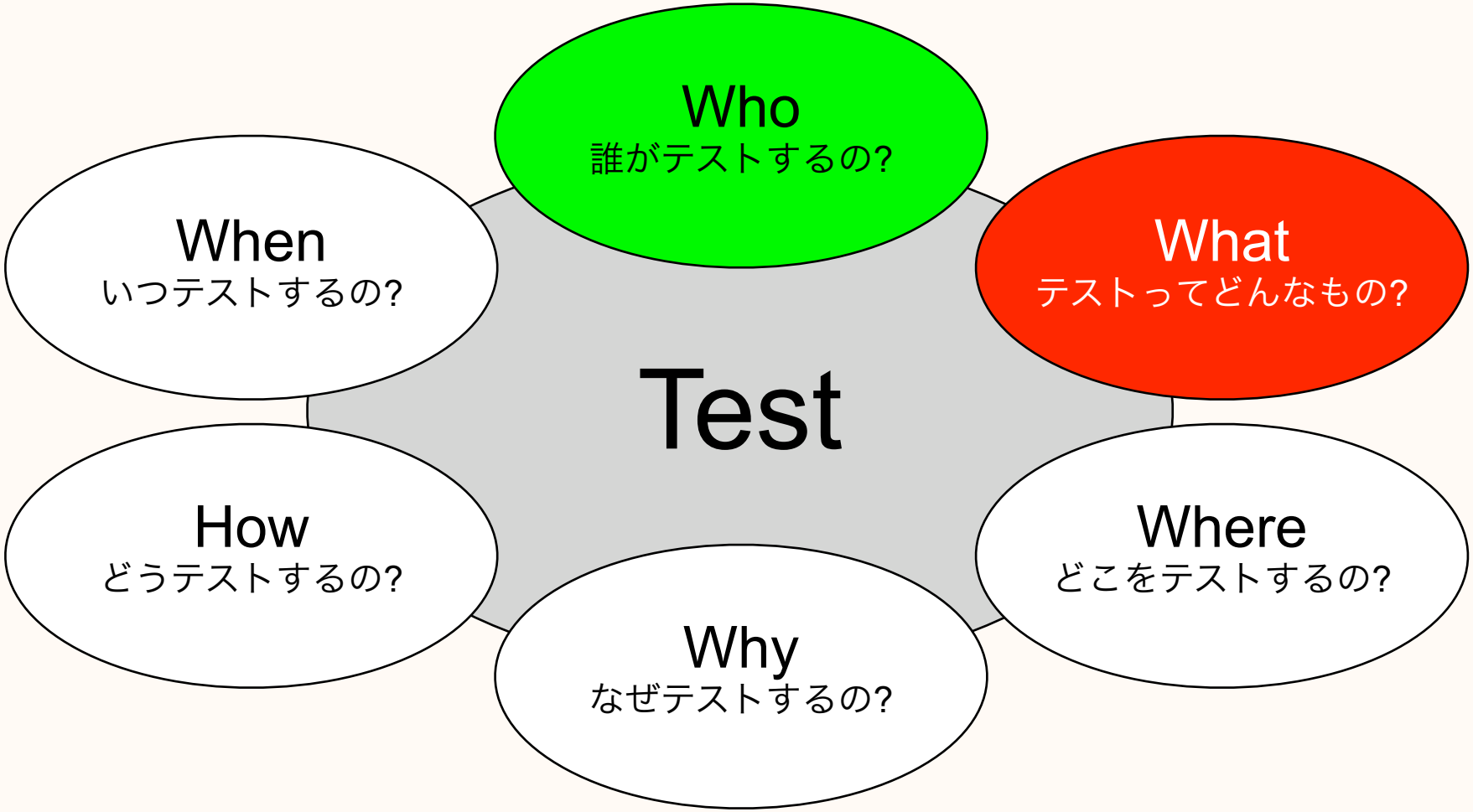
Developer Testing	Customer Testing	QA Testing
開発者	顧客	品質保証担当者
(のロール)	(のロール)	(のロール)
開発促進	進捗管理	品質保証

今日話すのはDeveloper Testing

「Q. 誰がテストするの?」
「A. プログラマ自身です」

- プログラマの
- プログラマによる
- プログラマのための
- プログラムとしてのテストを書きながら
- 開発を行っていくのがDeveloper Testing

テストってどんなもの?



ソフトウェア開発の三本柱

- 現代ソフトウェア開発の三本柱
  - バージョン管理
  - テスティング
  - 自動化
- 三脚椅子のようなもの
  - どれを欠いても不安定

©Artek (http://www.artek.fi/)

Developer Testingのテスト

- Developer Testingにおけるテストとは

  - 素早く動作し
  - 自動化されており
  - テスト対象コードの特定の機能を検証するため
  - プログラマ自身によって書かれたコード

xUnitへの流れ

- それまでも自動テストの仕組みはありました
  - テスト用のmainメソッドとか
  - テスト実行用バッチスクリプトとか

- テスティングフレームワークの登場
  - xUnitファミリー

    SUnit, JUnit, NUnit, PHPUnit,…
  - テスト方法が共有された

    キャズムを越えた?

自動化されたテスト

- テスティングフレームワークがもたらしたもの
  - テスト記述方法の共通化
  - テスト実行方法の共通化
- テストを書いた人でなくとも、同じようにテストを実行できるようになった

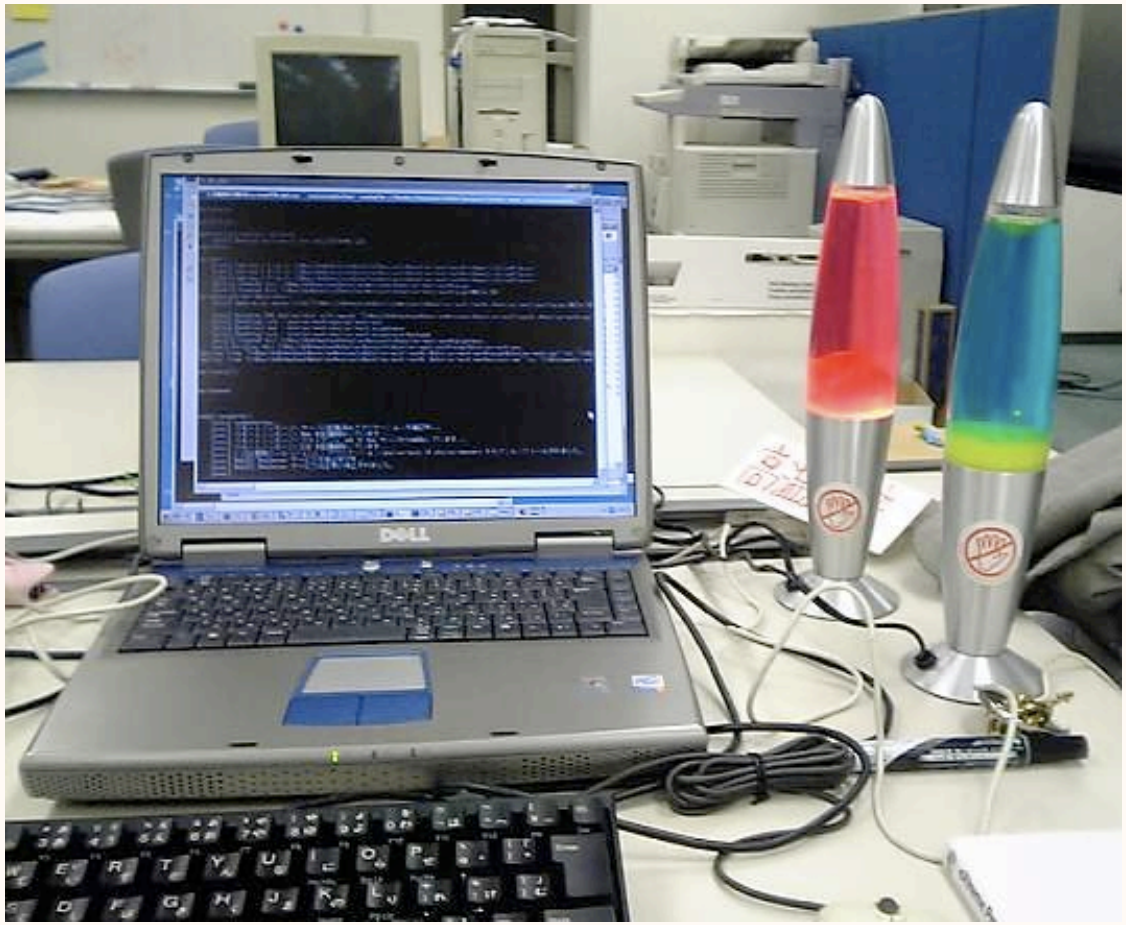
自働化されたテスト

- テスティングフレームワークがもたらしたもの
  - テスト記述方法の共通化
  - テスト実行方法の共通化
- テストを書いた人でなくとも、同じようにテストを実行できるようになったということは…

- コンピュータでも人と同じように実行可能
  - テストの自働化へ

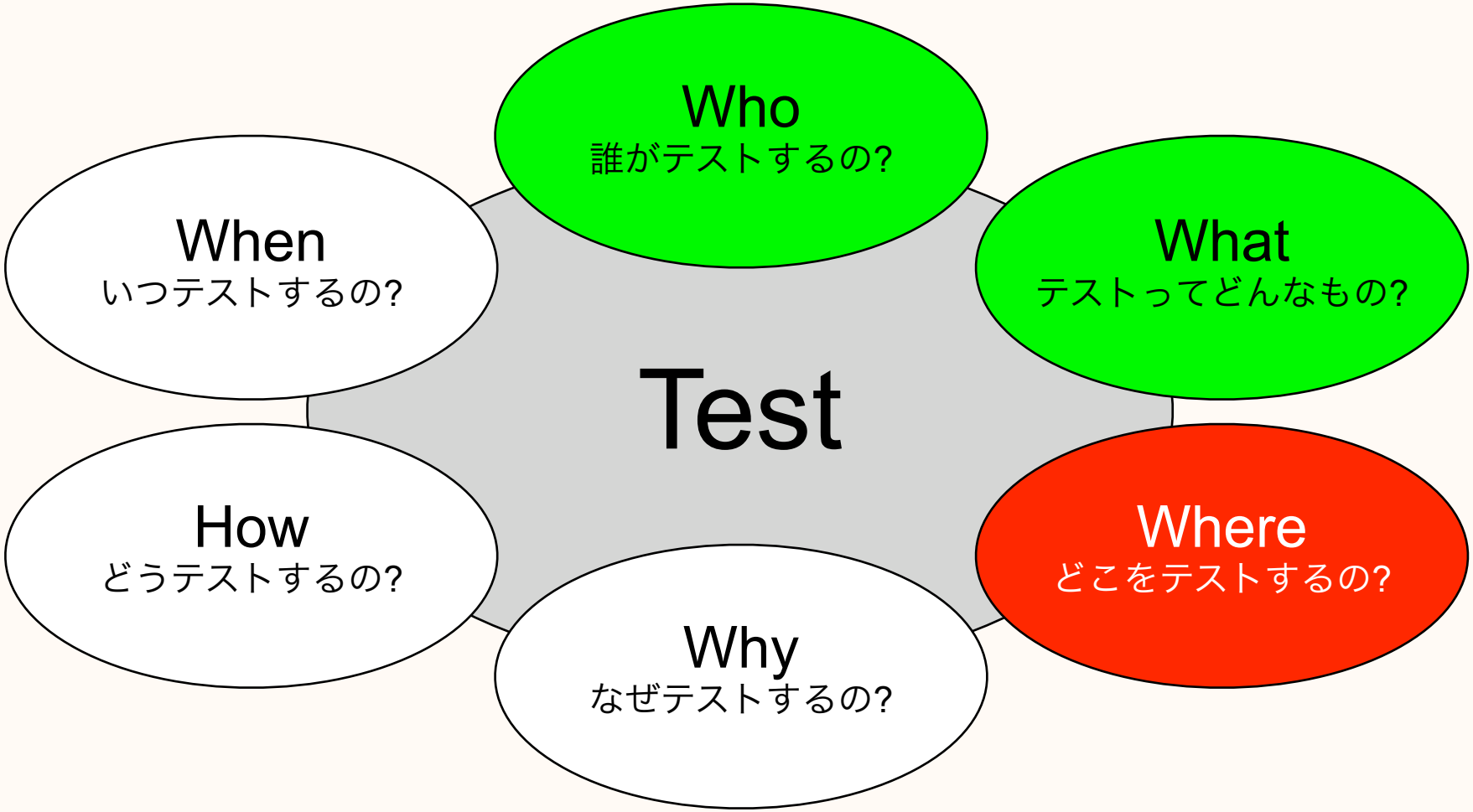
なぜ自働化?

- コンピュータは
  - 正確
  - 一貫している
  - 疲れない



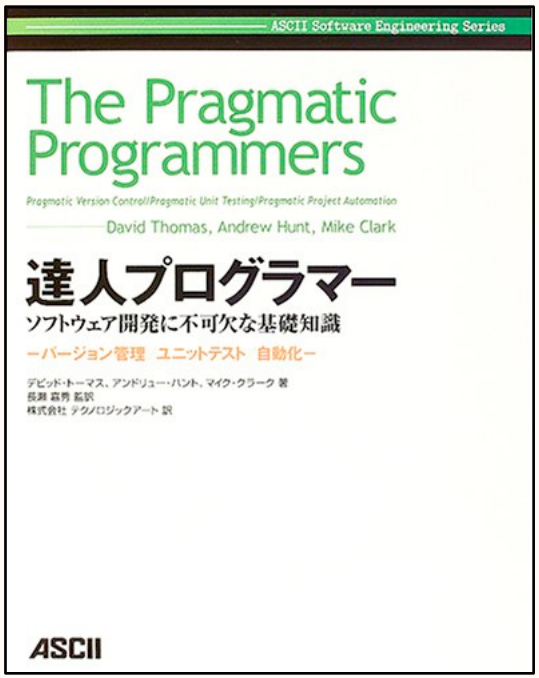
- 現状の「見える化」
- 時間の有効活用
  - 人間がやるべきことはまだ沢山ある

どこをテストするの?



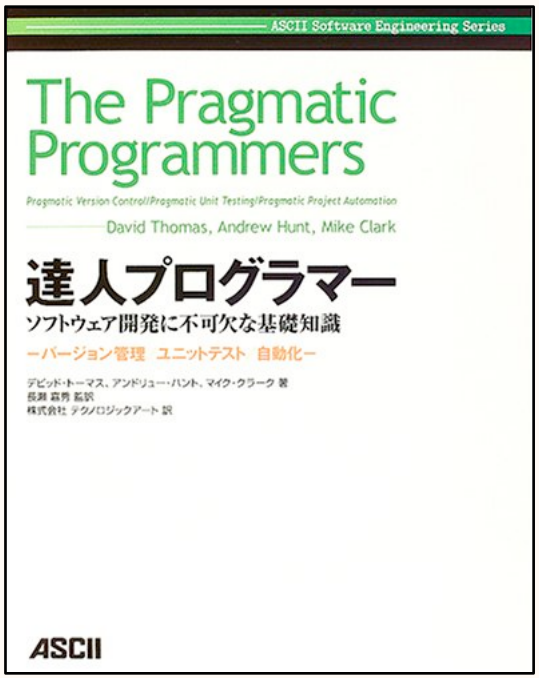
テストする内容

- Right-BICEP
  - Right (結果が適切か)
  - Boundary (境界条件)
  - Invert (逆の関係を確かめる)
  - Cross-Check (クロスチェック)
  - Error conditions (エラー条件)
  - Performance (パフォーマンス)



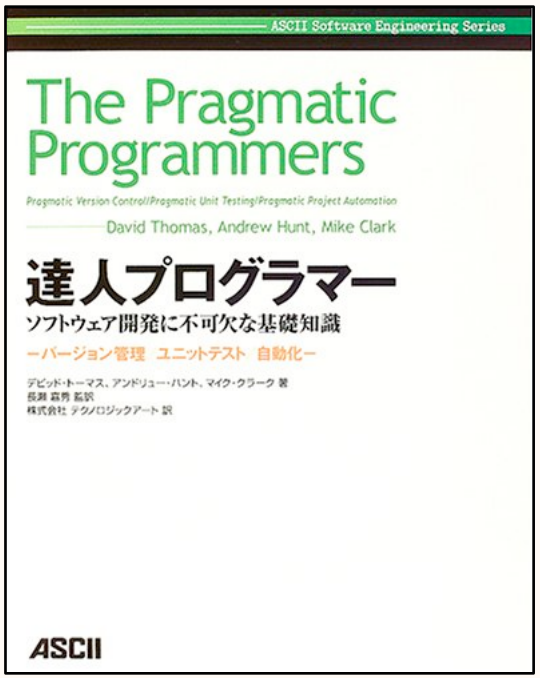
境界条件テストのコツ

- CORRECT
  - Conformance (フォーマットなどの適合性)
  - Ordering (順序)
  - Range (範囲)
  - Reference (参照、事前/事後条件)
  - Existence (存在。null、空、0)
  - Cardinality (数。0-1-nの法則)
  - Time (時間。相対/絶対/並行)

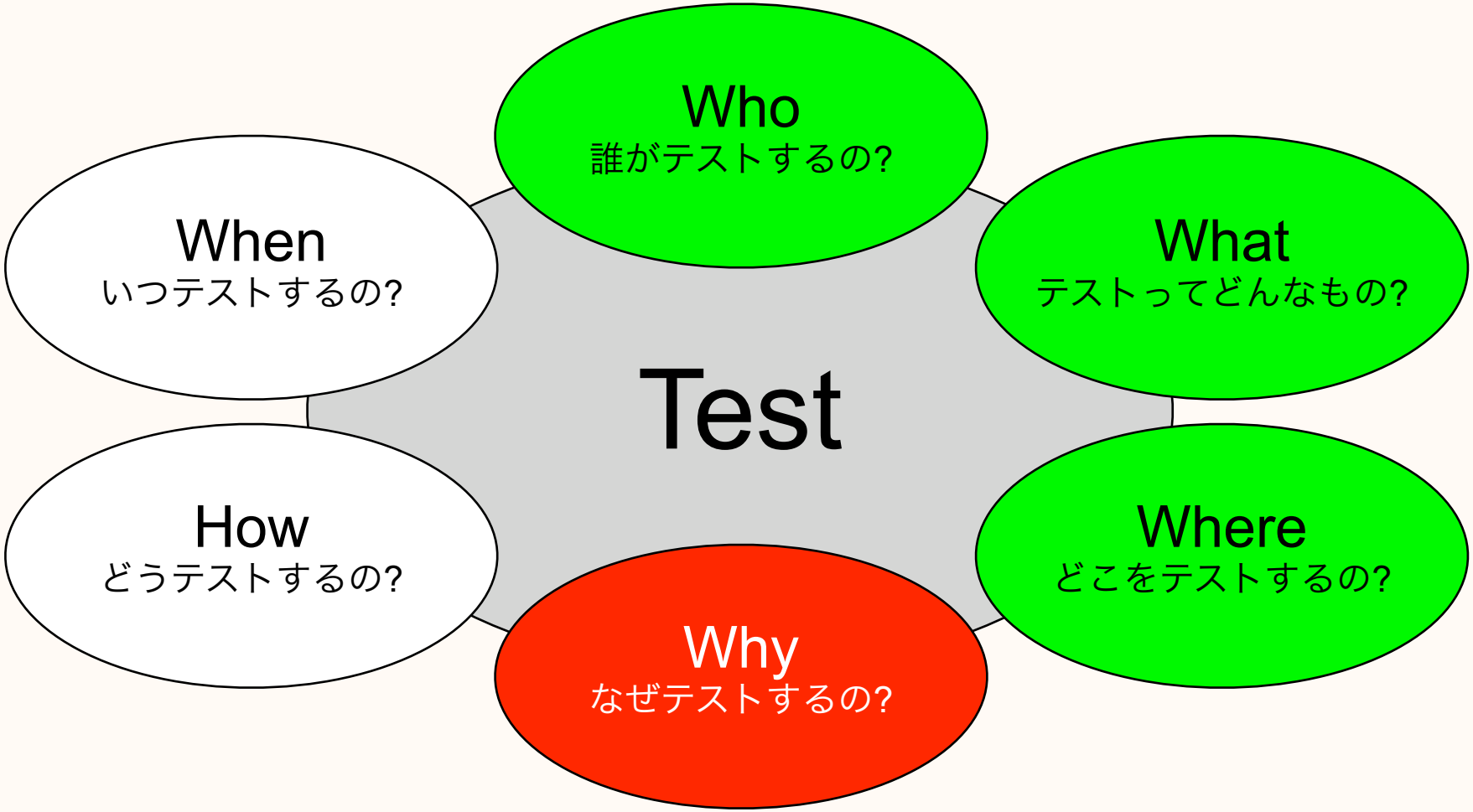


良いテストとは

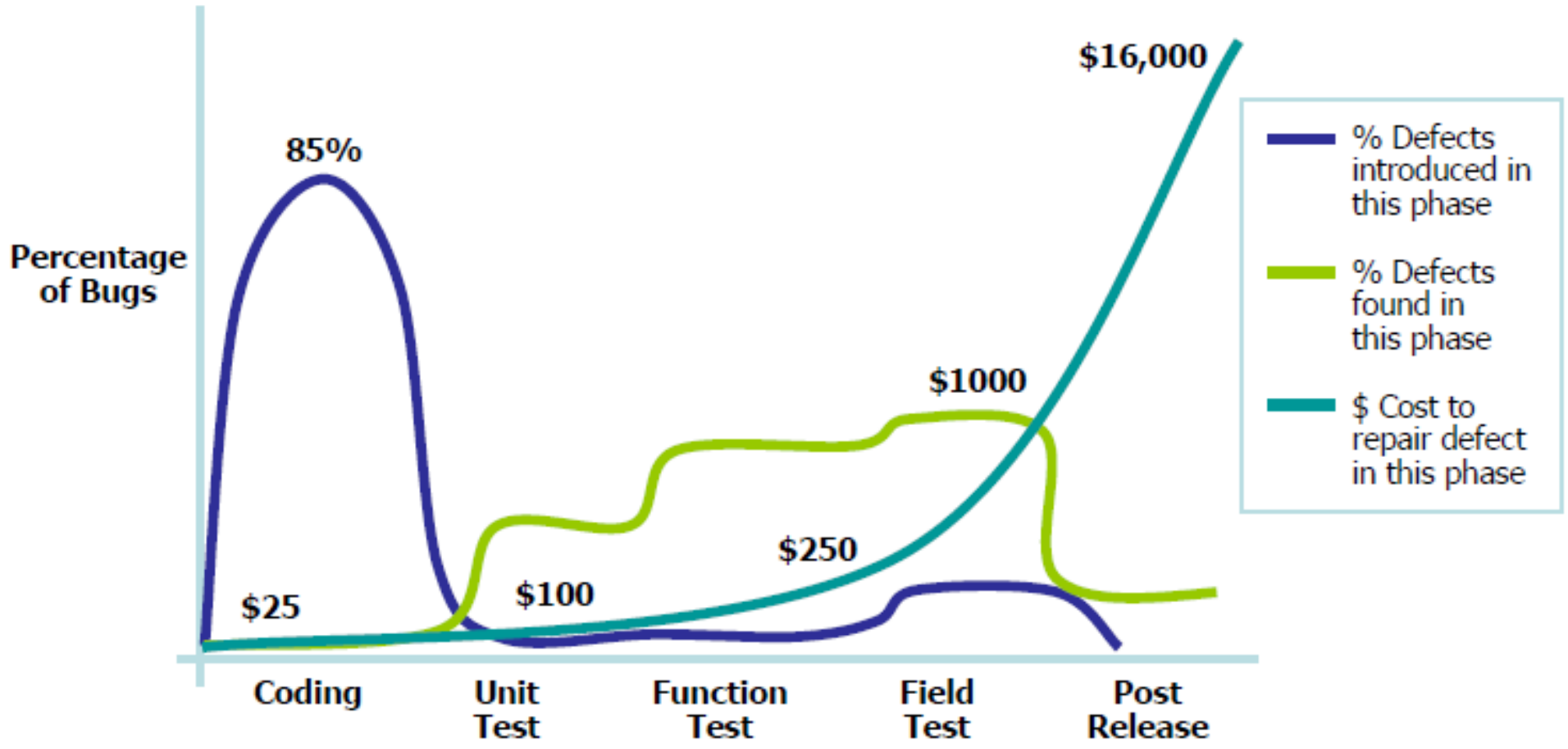
- A-TRIP
  - Automated (自動化されている)
  - Thorough (徹底している)
  - Repeatable (何回でも実行可能)
  - Independent (独立している)
  - Professional (プロのコード)



なぜテストするの?



フェイズとバグと修正コスト



Source: Applied Software Measurement, Capers Jones, 1996

「なぜテストするの?」

- 教科書的な解答：
  - 意図したとおりに動作するか確認する
  - 常に意図したとおりに動作するか確認する
  - コードが信頼できるか確認する
  - 意図を表現する(実行可能なドキュメント)

で、だ。本当のところは?

- ソフトウェア工学的なメリットもいろいろあるけれど、最大の理由は工学的なものじゃない。最大の理由は心理的なもの

- 即座にフィードバックを得るため
- 書いたコードに自信を持つため
- これから書くコードに自信を持つため

Seasar

心理的な側面

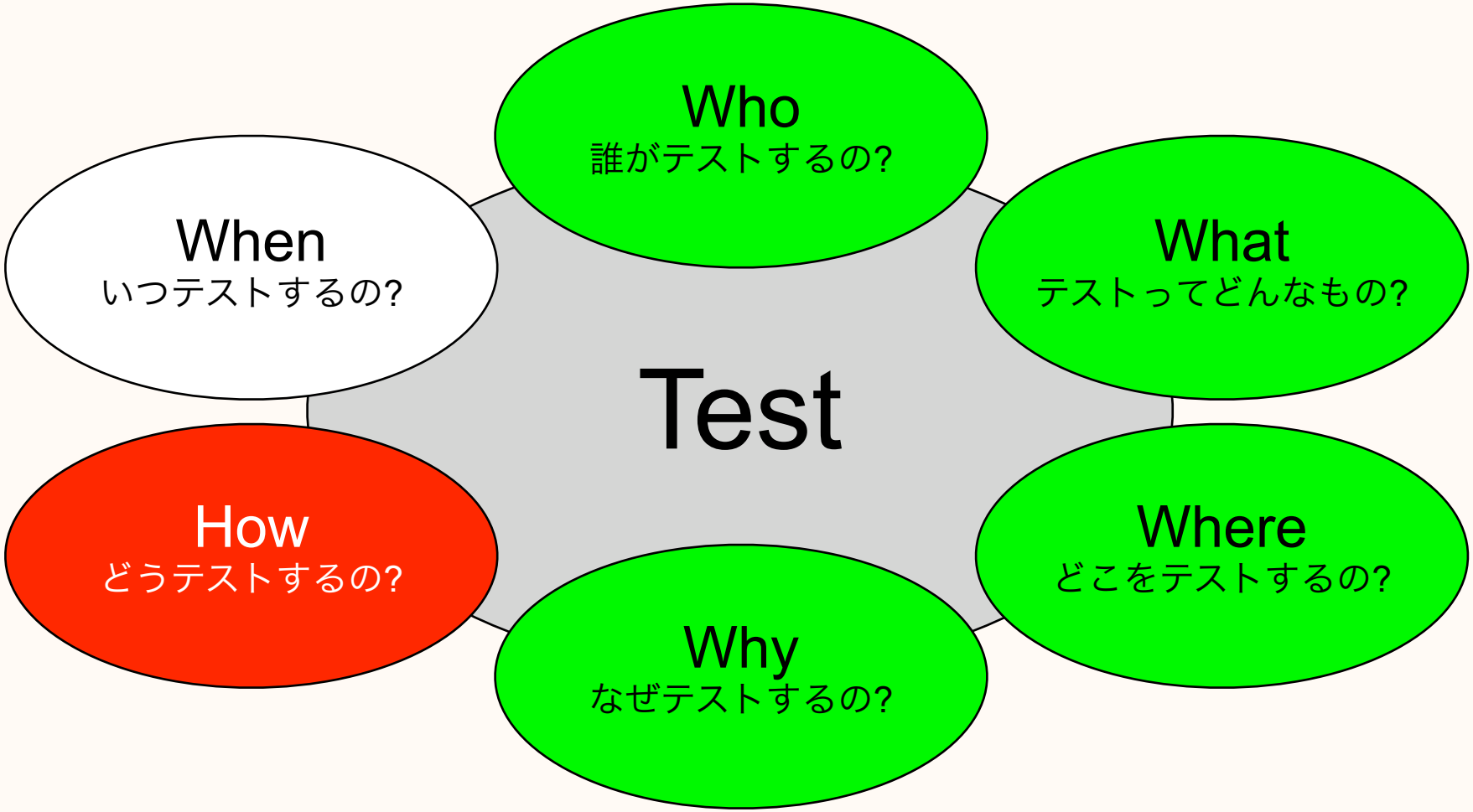
- 自動化されたテスト群があると
  - より自信を持って開発できる
  - よりよい設計が出来る
  - 心に余裕ができ、メンバーに気が回るようになる
- 心理をコードにする
  - いまの不安をテストにする
  - 小さな一歩、大きな一歩

テストの真の目的

- 目的は健康(Health)
  - ソフトウェアの健康
    - 動くコード
    - 無駄のないコード
    - シンプルな設計
    - 変化に適応できる
  - 開発者の健康
    - 体の健康
    - 心の健康

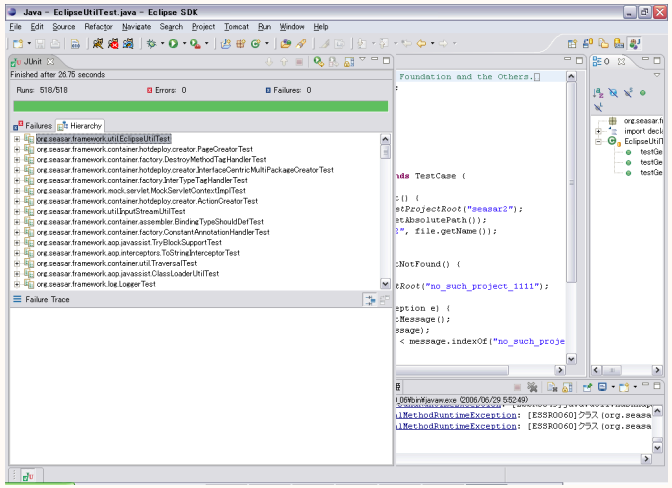
健康

どうテストするの?



JUnit

- Javaのテスティングフレームワーク代表
  - テストが簡単に書ける仕組みを提供
  - IDE(EclipseやNetBeans等)とも相性がいい



```
import junit.framework.TestCase;

public class HogeTest extends TestCase {

    public void testToUpperCase() throws Exception {
        assertEquals("HOGE", "hoge".toUpperCase());
    }
}
```

FowlerはJUnitをこう評した

"Never in the field of software development was so much owed by so many to so few lines of code."

「ソフトウェア工学の記録の中で、これほど多くの恩恵を、これほど多くの人が、これほど少ないコードに対して、受けていることはない」

- Martin Fowler

JUnit3とSeasar

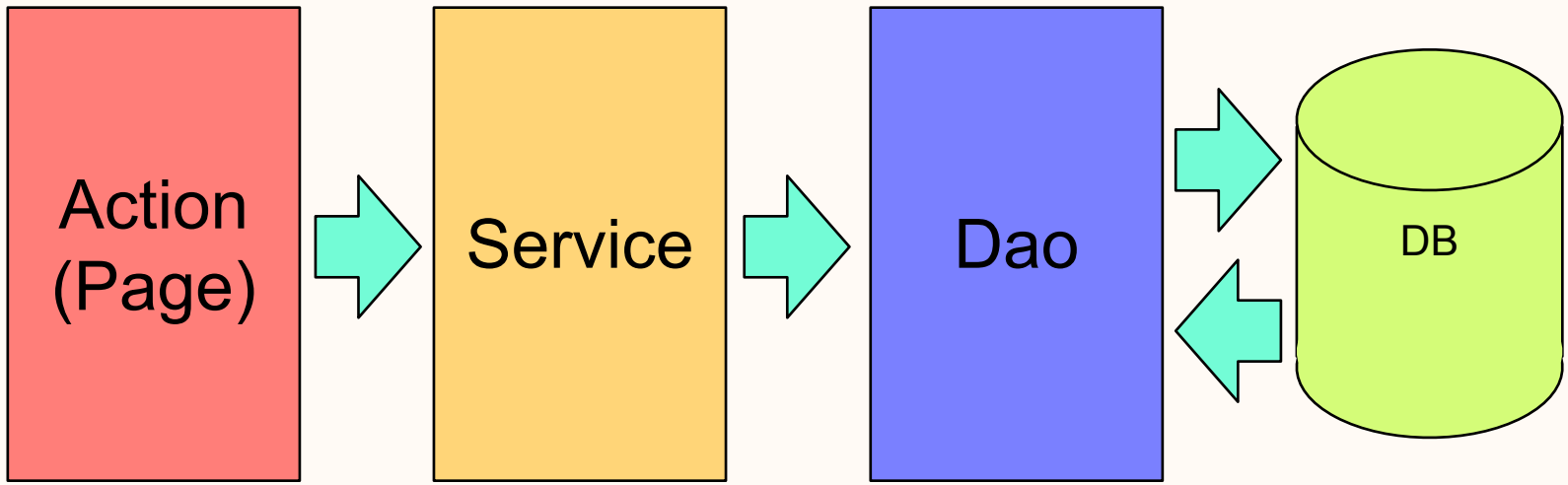
- S2Unit (JUnitのSeasar拡張)
  - テスト名規約によるトランザクション制御
  - MockInterceptorによるモック機能
  - ExcelによるDaoテストのサポート
  - メソッド毎のsetUpが可能
- DIとテストの良い関係
  - デメテルの法則に従い、きれいな設計にしやすい
  - モックオブジェクトも導入しやすい

モックオブジェクト

- テストのために作成する「本物のふりをする偽物」のオブジェクト
- セットアップが難しいオブジェクトや、コストが高いリソースなどの代わりになる
  - 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条件を作り出すことが出来る
- 便利だが、使いすぎ注意

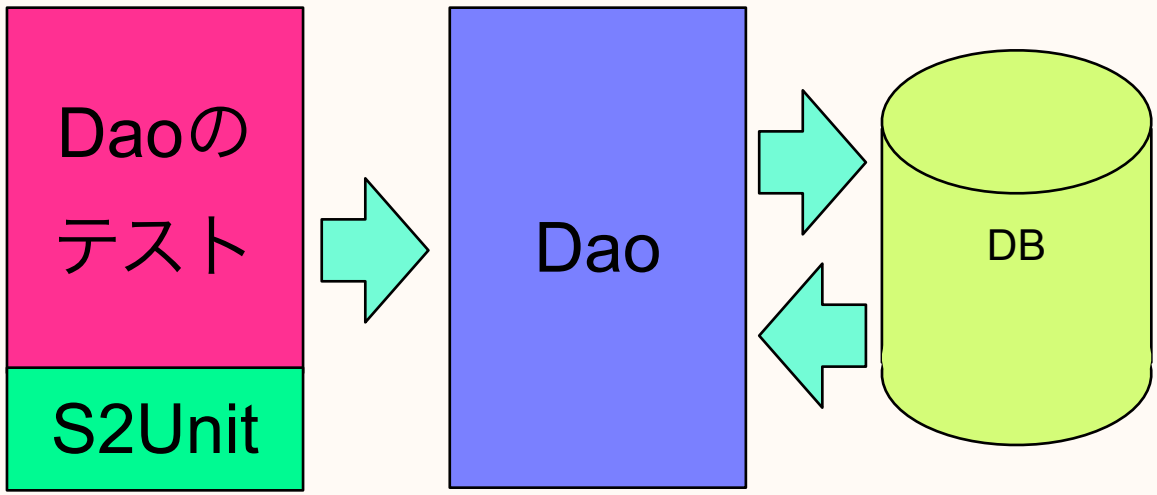
レイヤーとモックとテスト

- 伝統的なレイヤモデルを例にしましょう
  - Action(Page) - Service - Daoの三層構造
  - それぞれが下層レイヤに依存している



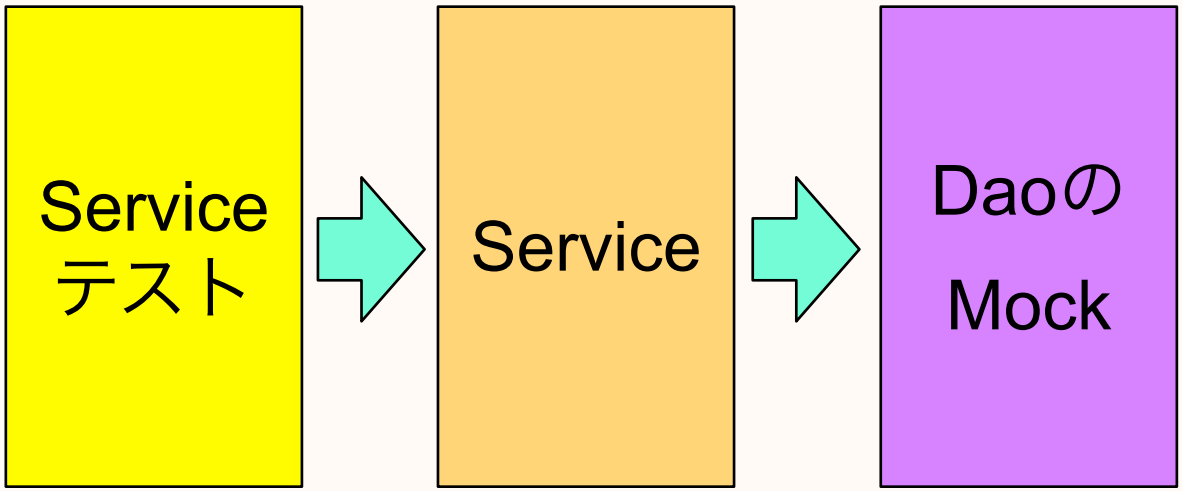
DBに近いレイヤのテスト

- DBに近いレイヤのテストは、実際にDBに接続して、SQLを発行するテストを行いたい
  - S2UnitのDaoテスト支援機能を使う
  - Excelを使ってDBにテスト用データを入れ、テストが終わったらロールバックして掃除



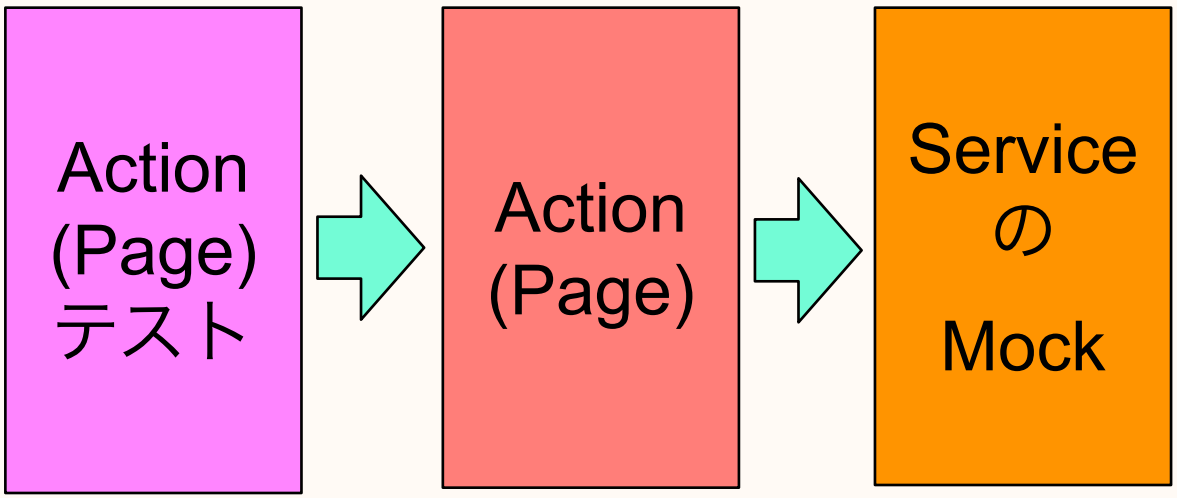
中間レイヤのテスト

- モックを使って軽量なテストを行いたい
  - 実際のDBに繋に行くテストは重いが、Daoのモックを用意することで、本物のDBに依存せずにテストできる
  - 依存が複雑でないときは、普通のS2Unitではなく普通のJUnitでもOK
  - Service内の条件分岐や計算等のロジックに集中する



Action(Page)レイヤのテスト

- Action(Page)レイヤーのテストは出来るだけ軽くしたい
  - Serviceなどの依存レイヤのモックを作り、簡単にテストできるようにする
  - Dxoなどのデータ変換部分を重点的にテストしたい



JUnit 3.8.x → JUnit4へ

- 継承ベースからアノテーションベースへ
  - @Test
  - @Before/@After
- static import多用
  - 独自のassertメソッドを使いやすくなった
- RunWithアノテーションによるテスト拡張
  - @RunWith(いろいろ.class)

JUnit 3.8.x → JUnit4へ

JUnit3

```
import junit.framework.TestCase;


public class SetTest extends TestCase {
    private Set set;


    public void setUp() {
        set = new HashSet();
    }


    public void testInitialSizeIsZero() {
        assertEquals(0, set.size());
    }
}
```

JUnit4

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;

public class SetJUnit4Test {
    private Set set;

    @Before
    public void prepareContext() {
        set = new HashSet();
    }

    @Test
    public void initialSizeIsZero() {
        assertEquals(0, set.size());
    }
}
```

JUnit4.3 → JUnit 4.4

- assertThatメソッドの追加
  - assertEqualsの引数にまつわる混乱を解決
- hamcrestライブラリを同梱
  - is, notなどのメソッド
  - (英語を母語とした人には) 文章のように読める

    assert that expected result is “HOGE” とか

JUnit4.3 → JUnit 4.4

JUnit4.3

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;

public class SetJUnit4Test {
    private Set set;

    @Before
    public void prepareContext() {
        set = new HashSet();
    }

    @Test
    public void initialSizeIsZero() {
        assertEquals(0, set.size());
    }
}
```

JUnit4.4

```
import static org.hamcrest.core.Is.is;
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;

public class SetJUnit44Test {
    private Set set;

    @Before
    public void prepareContext() {
        set = new HashSet();
    }

    @Test
    public void initialSizeIsZero() {
        assertThat(set.size(), is(0));
    }
}
```

JUnit4 + Seasar = S2JUnit4

- RunWith(Seasar2.class)
  - TestCaseクラスではなく、JUnit4流の拡張
- 規約度が増した
  - 書くのを省略できるものはもっと省略可能に

    適切なデフォルト

    トランザクション連携
- EasyMock連携機能追加
  - より柔軟なモックテストが可能になった

テストのテストはどうする?

Q. では、テストコードのテストはどうするの?

A. 手段は二つあります

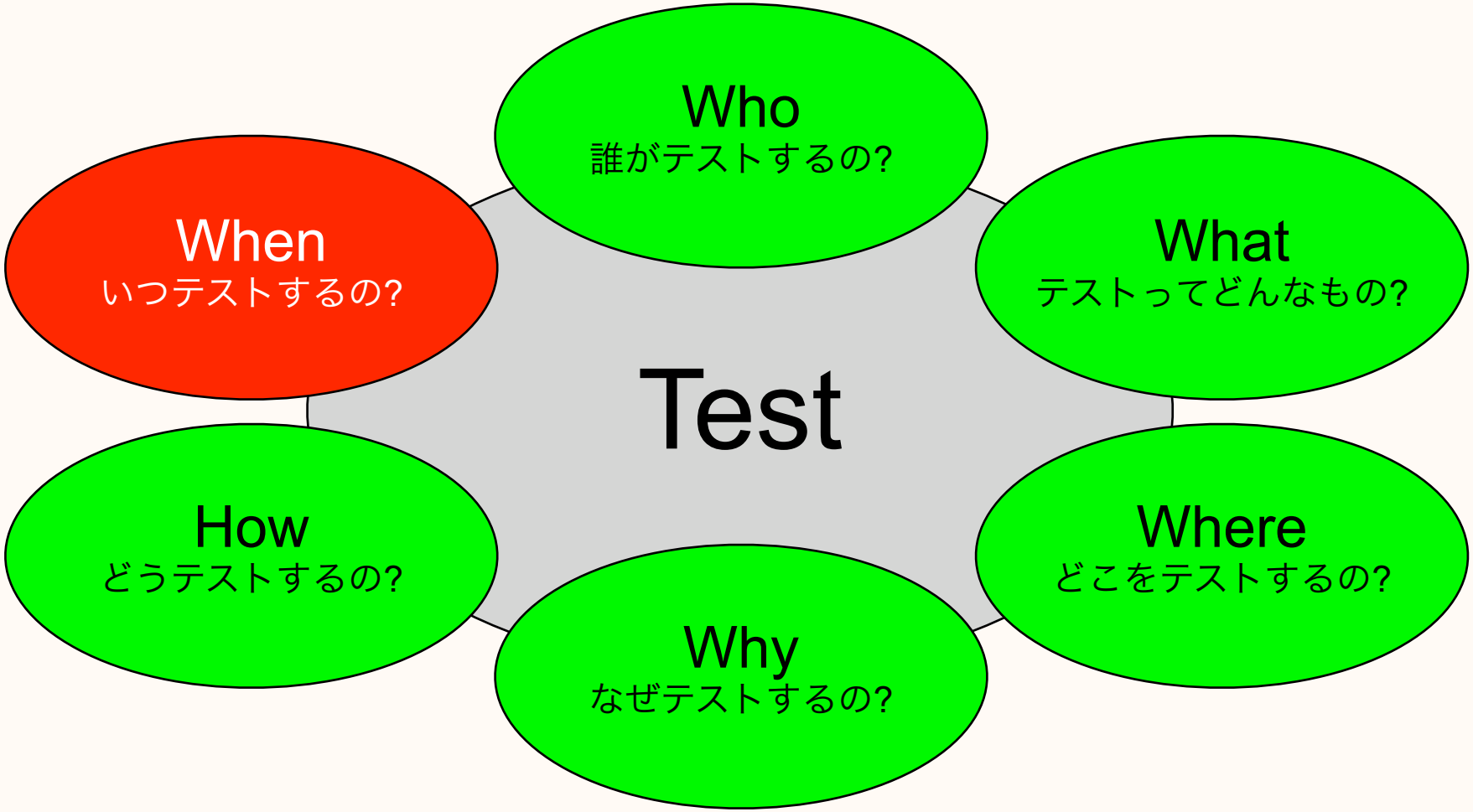
- コードレビュー

  他の人にコードを見てもらうことの効果は、もちろん
  テストコードにも当てはまる

- Mutation Testing

  テスト対象コードを書き換えて、テストがきちんと失敗するかどうか検証する。Mutation Testingを自動化するソフトも出てきている。(JesterやJumbleなど)

いつテストするの?

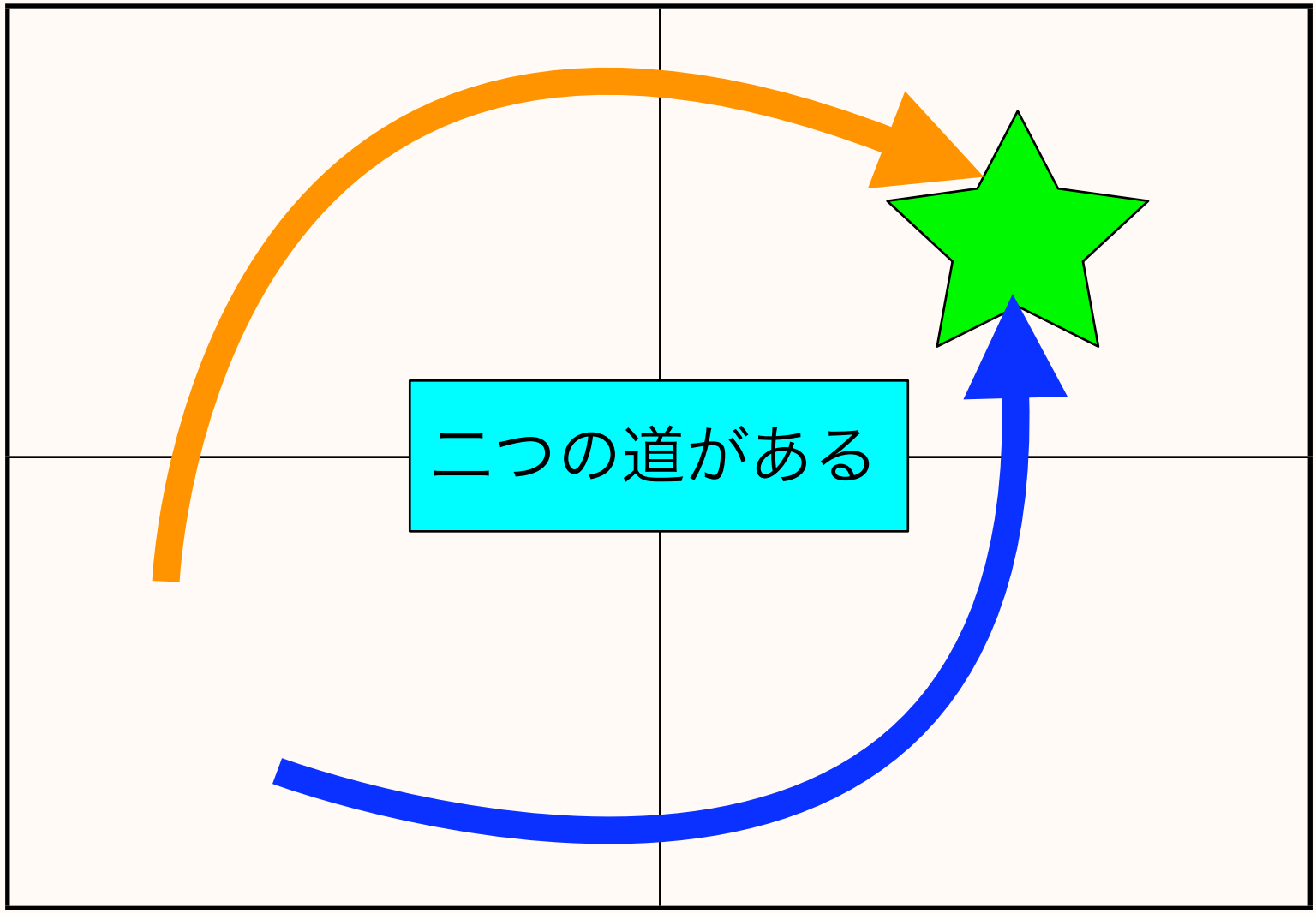


「常に」テストする

- 「Q. いつテストするの?」
  「A. 常に、テストします」

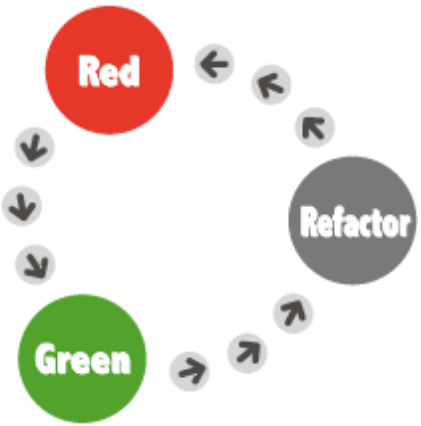
- 「常に」の二つの意味
  - Continuous Integration (継続的結合)
  - Test Driven Development (テスト駆動開発)

「動作する、きれいなコード」へ



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)
  6. 1～5を繰り返す



リファクタリングとは何か

- リファクタリング(名詞)：

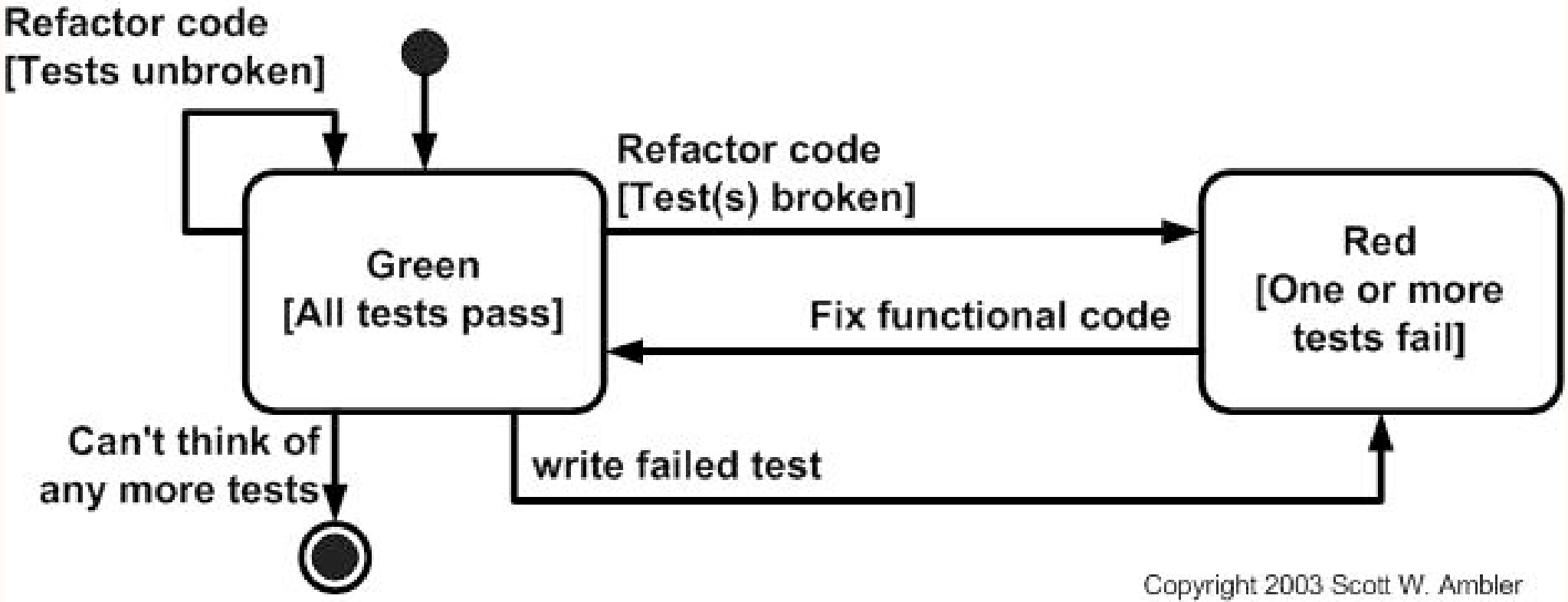
  外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変更させること

- リファクタリング(動詞)：

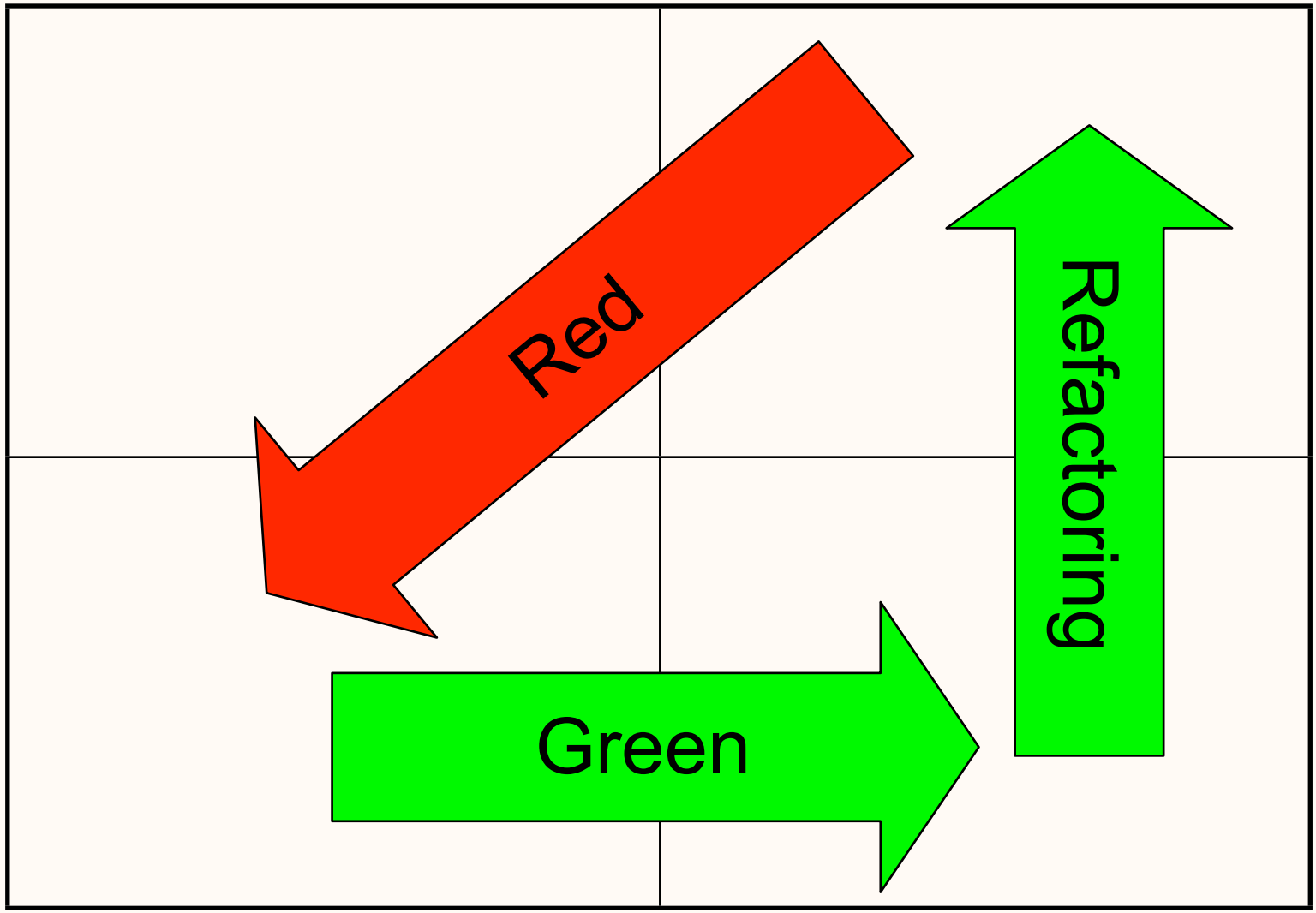
  一連のリファクタリングを行って、外部から見た振る舞いの変更なしに、ソフトウェアを再構築すること

TDDの状態遷移図

- Scott AmblerがTDDのステップをUMLで記述



動作させてから、きれいにする



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

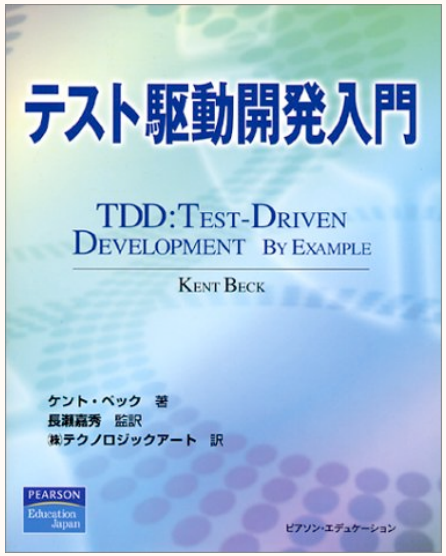
なぜTDDをするのか

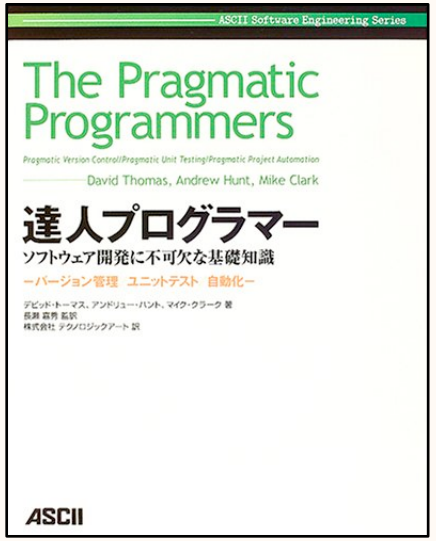
- 私たちは完璧なプログラマではない
  - 最初から思い通りにコードが書けるほど、私たちは賢くない
  - 最初から思い通りに動作するほど、対象は単純ではない

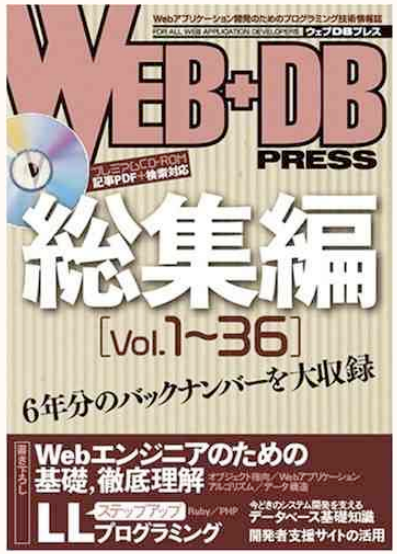
- 素早く対象に近づき、フィードバックを得て
  方向修正をしながら開発を行う必要があるから

テストはスキルです

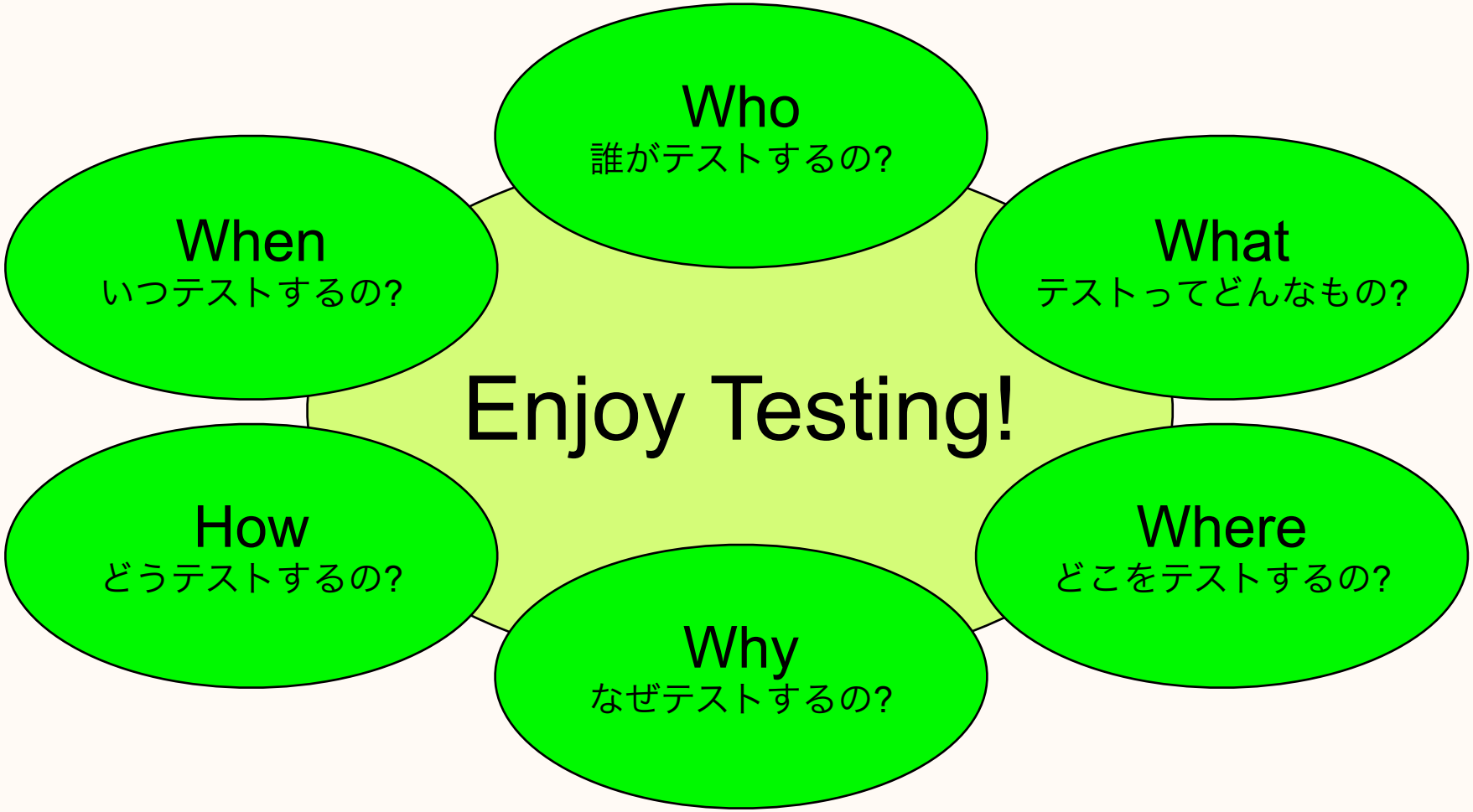
- テストやTDDはスキルです。つまり…
  - 才能ではなく、習得可能です
  - 量は質に転化します
  - 写経してみましょう



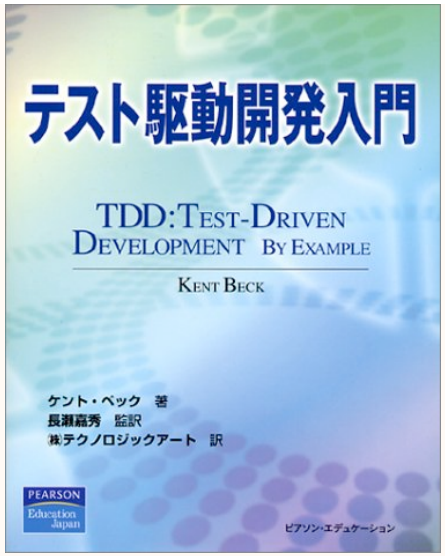


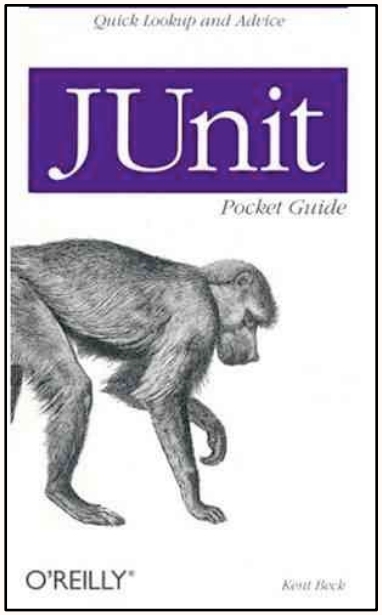


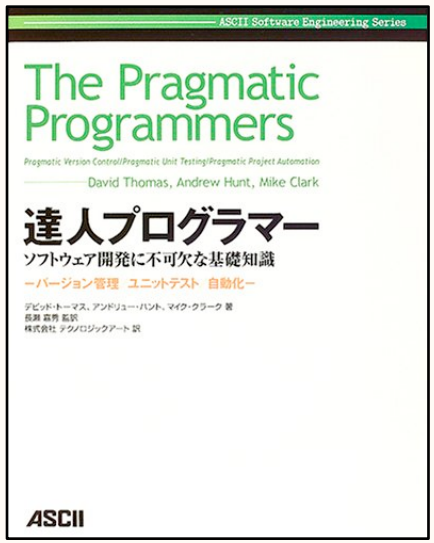
Any Questions?



参考文献



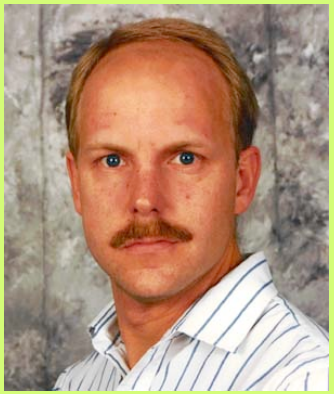




Special Thanks

- 会場の皆様
- 角谷さん
- kdmsnrさん
- ひがさん
- 中村さん
- 技評編集者Iさん
- masarlさん

そして、テストに関する知見をくれた方々



Seasar

ご清聴
ありがとう
ございました