

TDDのパターン

TDD 道場 #20 [拡大版]

とある道場の
乱取稽古
TDD Dojo



BluewaterSoft

2014/8/23 biac



わんくま同盟 名古屋勉強会 #32

スピーカー紹介: biac as 山本 康彦

- 宇宙世紀以前の生まれ
スプートニク1号より3ヶ月ほど前
- 最初は **HONDA**
クルマの設計/研究を10年くらいやってた
- 今は **BluewaterSoft**
を名乗ってアプリ開発とか技術解説記事とか
- 「**NUnitの全貌**」 ⇒
CodeZine 2012/4

Home | ニュース | **記事** | 注目ブックマーク | コミュニティ | デブサミ

C# | Java | VB.NET | C++ | PHP | Ruby | Perl | JavaScript | SQL | Adobe | 言語一覧

C# | 開発/設計/テスト

NUnitの全貌 ~ 基本から、最新バージョンの新機能まで

C#で始めるテスト駆動開発入門(3)

biac [著] 2012/04/13 14:00

いいね! 27 | +1 8 | BI 85 | ツイート 80

ダウンロード サンプルソース (C#) [29.84 KB]

ダウンロード CsTdd03_20130403.zip [29.74 KB]

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

NUnitの最新バージョン2.6の主な機能を解説します。普段からNUnitを使っている開発者でも、「こんな機能があったのか!」と驚くようなことがきっとあるでしょう。

はじめに

C#でTDDしている開発者にとって、おそらく一番なじみのあるテストフレームワークはNUnitでしょう。今年の2月に、そのVersion 2.6が正式リリースされました。2.x系はこれが最後のリリースとなり、次はVersion 3になるとされています ([NUnit Roadmap](#)参照)。この機会に、NUnitの基本機能から、2.6で追加された新機能まで、全体を一通り把握しておきましょう。

■ NUnitの最新版バージョン2.6のテストランナー (nunit.exe)

Copyright © 2002-2011 Charlie Poyar, Michael C. Two, Robert A. Graham, Philip Craig, Ethan Smith, Doug de Torny, Charlie Poyar

Developers: James W. Newell, Michael C. Two, Robert A. Graham, Philip Craig, Ethan Smith, Doug de Torny, Charlie Poyar

Thanks to: Kent Beck and Edd Gray

Model Version: 2.6.0.001

Framework Version: Net 3.5

Test Cases: 100 | Tests Run: 100 | Errors: 0 | Failures: 15 | Time: 1.187



TDD = テスト ファースト + リファクタリング

- Test Driven Development (テスト駆動開発)
- テスト ファースト: RED と GREEN の繰り返し
- リファクタリング: GREEN を維持したまま実装を改善

失敗するはずのユニット テストを1つ書き、
失敗することを確認 (=RED)



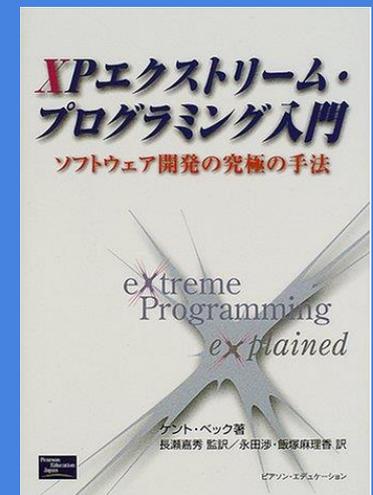
ユニット テストに通るだけの実装を追加し、
成功することを確認 (=GREEN)

TDD のルーツは XP

- 「XP エクストリーム・プログラミング入門」
(Kent Beck 著、2000年)
- XPの価値観 = 「**コーディングこそ開発!**」
をTDDも受け継いでいる
(だから、実装の話だけなのに「開発」と名乗っている)

Erich Gamma が書いた序文より

(XP は) **コーディング**をソフトウェアプロジェクトのキーアクティビティ、つまり「**中心となる活動**」として選んでいる。



Agenda

- TDD そのもののパターン
 - ・ テストファーストのパターン
 - ・ リファクタリングのパターン

TDD のデバッグパターン

TDD の適用パターン

TDD の学習パターン

TDD の導入パターン



ごめんなさい

ちゃんとしたパターン カタログには
なってないです m(_ _)m

* 例: GoF デザイン パターンの
カタログ化方式 ⇒

* 「パターン言語は、一連のちょっとした
知識が、ある一定のスタイルで記述、整列
されており、設計者がその時点で最も適し
た質問を尋ねる（または答える）ことがで
きるようになっている。」（オブジェクト
指向プログラムのためのパターン言語の使
用）

1.3 Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

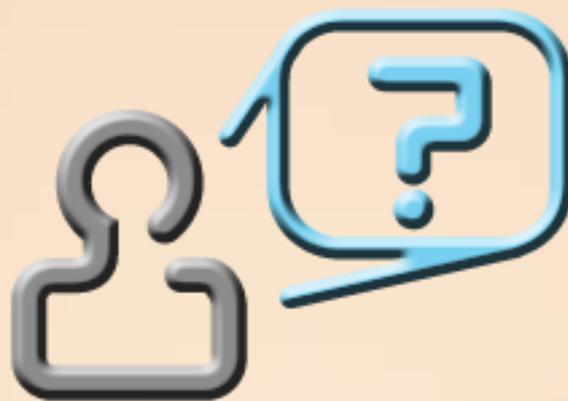
Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures





テストファーストのパターン

テストファーストを上手くやるパターンと、失敗するアンチパターン

テストファースト: 作業のパターン

- TDD 3原則 by Robert C Martin
[ArticleS.UncleBob.TheThreeRulesOfTdd](#) (2005)
より。

1. 失敗するユニットテストを成功させるためにしか、プロダクトコードを書いてはならない。
2. 失敗させるためにしか、ユニットテストを書いてはならない。コンパイルエラーは失敗に数える。
3. ユニットテストを1つだけ成功させる以上に、プロダクトコードを書いてはならない。

テストファースト: 作業のアンチパターン

- × ユニットテストに書いていないプロダクトコードを書く ⇒ そのメソッドは手動テストが必要に!
- × 失敗しないユニットテストを追加する (削除せずに残しておく) ⇒ 仕様変更時のテスト修正作業増
※ コードが安定してから、ドキュメントとして書くのはアリ
- × ユニットテストの失敗を確認しない ⇒ それは失敗しないユニットテストかも (⇒上の項に続く)

テストファースト: 進め方のパターン

- プロダクトコードに1つ分岐(経路)を追加するには、1つか2つのユニットテスト (下記のパターン)
※何もないときに書くプロダクトコードは、1つめの経路(分岐)と考える
- 三角測量 (ユニットテスト 2つ)
 1. 「Fake it!」ピンポイントを通す (例: `return 1;`)
 2. 同値領域全体を通す (例: `return n * n;`)※ 必要ならば 3つ以上も可
- 明白な実装 (ユニットテスト 1つ)

テストファースト: 進め方のアンチパターン

- × 先にユニットテストを全部書く ⇒ 失敗しないユニットテスト (=プロダクトコードの成長に寄与しないムダ) が混ざる

※ XP はインクリメンタル&イテレーティブな開発手法。
そこで生まれた TDD は、究極のインクリメンタル&イテレーティブな手法



テストファースト: テストケース抽出のパターン

- メソッドの外部設計
⇒ 同値領域ごとに具体例 = テストケース
- メソッドの外部設計方法は自由
入出力表や状態遷移表
TODOリスト
頭の中だけ
.....

テストファースト: テストケース抽出のアンチパターン

- × 先にプロダクトコード(=内部設計)を考え、後からそれを満たすテストケース(=外部設計)を考える

⇒ ユニットテストを先に書いていても、
実態は「実装ファースト」。
テストは通ってるのに、正しくないコードに!

※ 実際には、実装を見ながら次のテストケースを決める。
ようは、外部設計と内部設計(実装)とで思考を切り替えること。



テストファースト: ユニットテストの記述パターン [1/2]

- 1メソッド = 1テスト
- 準備 → 実行 → 検証 (→後始末)
 - ※ 検証に複数Assertを必要とすることは、ある

```
public void ユニットテスト例01_複数Assertが必要()  
{  
    // 準備  
    var testData = new Data{ P1=hoge, P2=moge, ...};  
  
    // 実行  
    var result = (new Foo()).Bar(testData);  
  
    // 検証  
    Assert.AreEqual(1, result.D1); // 期待値=1 ⇔ 実際の値=result.D1  
    Assert.AreEqual(2, result.D2);  
}
```



テストファースト: ユニットテストの記述パターン [2/2]

- ユニットテストは独立させる
= テストを実行する順序に依存しないこと
- 期待値と実装は独立させる
※ 期待値が複雑でも、必ずテストコード側だけで生成する

```
public void ユニットテスト例02_期待値が実装に依存する良くない例()  
{  
    var 期待値 = this.Create期待値(プロダクト.Foo(123));  
  
    var result = プロダクト.Buzz(123);  
  
    Assert.AreEqual(期待値, result);  
    // 期待値の具体的な値が分からない (=外部設計が不明瞭)。  
    // Buzz のテストのはずだが、Foo にバグが混入してもレッドに!  
}
```



テストファースト: 外部設計のパターン [1/6]

• 内部状態に関するパターン [1/3]

外部設計として状態を見せることが必要な場合

```
public void Stackクラスのユニットテスト例01_外部設計として状態を見せる()  
{  
    // 準備  
    var stack = new Stack();  
  
    // 実行  
    stack.Push(3);  
  
    // 検証  
    Assert.AreEqual(1, stack.Count); // ←仕様として Count が必要な場合  
    CollectionAssert.AreEqual(new int[] {3, }, stack.Items);  
}
```



テストファースト: 外部設計のパターン [2/6]

• 内部状態に関するパターン [2/3]

外部設計として状態変更と変更結果が観測できるメソッド・ペアが必要な場合

```
public void Stackクラスのユニットテスト例02_状態変更と観測のメソッドペア()
{
    // 準備
    var stack = new Stack();

    // 実行
    stack.Push(3);

    // 検証
    Assert.AreEqual(3, stack.Pop()); // ←Push とペアになるメソッドが必要な場合
}
```



テストファースト: 外部設計のパターン [3/6]

• 内部状態に関するパターン [3/3]

状態を知るためのテスト専用メソッド(「プローブ」)を実装してしまう

```
public void Stackクラスのユニットテスト例03_プローブ()
{
    // 準備
    var stack = new Stack();

    // 実行
    stack.Push(3);

    // 検証
    Assert.AreEqual(1, stack.testGetCount()); // ←テスト専用のメソッド
    // 「#if TEST」などとして、リリースビルドには含まれないように!
}
```



テストファースト: 外部設計のパターン [4/6]

• 内部状態に関するアンチパターン

メンバー変数を(外部設計によらずに)公開してしまう

```
public void Stackクラスのユニットテスト例04_不必要なメンバーを公開()
{
    // 準備
    var stack = new Stack();

    // 実行
    stack.Push(3);

    // 検証
    Assert.AreEqual(1, stack._list.Count);
    // _list (おそらく List 型) を持っているという実装に固定されてしまう
}
```



テストファースト: 外部設計のパターン [5/6]

• 制御できないものは切り離せ!

- * システム クロック
- * 乱数
- * ネットワークアクセス中のエラー
- *

⇒ 制御可能な別物と置き換えてしまう

CodeZine

C#で始めるテスト駆動開発入門 (6)

TDDしにくいモノは切り離せ!

～ 現在日時の扱い方



The screenshot shows a web browser window displaying an article on CodeZine. The article title is "TDDしにくいモノは切り離せ! ~ 現在日時の扱い方" (TDD is difficult to do with things that are hard to separate! ~ Handling the current date and time). The author is "biac" and the date is "2012/08/24 14:00". The article content discusses the challenges of testing code that depends on external factors like system clocks or databases. It suggests that for such code, the strategy is to "separate" the dependencies, meaning to replace them with controllable alternatives. The article is part of a series "C#で始めるテスト駆動開発入門 (6)".



テストファースト: 外部設計のパターン [6/6]

- 時間の掛かるI/Oはモックで騙せ!

- * ネットワーク アクセス

- * データベース アクセス

アクセスそのものをテストする以外は、モックを使う

テストファースト: モック利用のパターン

- ホンモノを使うとテストファーストに支障をきたす場合に、最小限の範囲でモックを使う

*制御できないもの

*時間が掛かる処理

*ホンモノが間に合わないとき

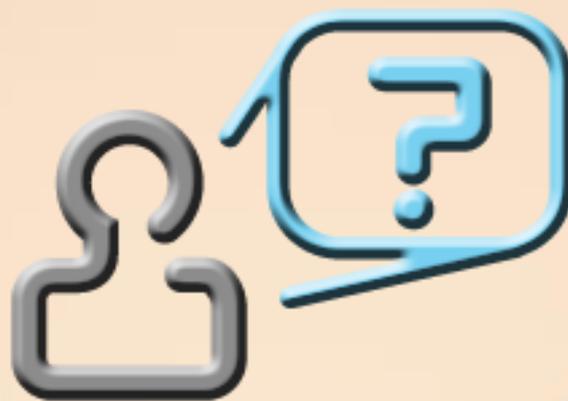
- モックもTDDで作る

- ホンモノを受け取ったら、まず受け入れテスト

⇒ レッドになったら?

ホンモノを直すか、モックをちょっとずつ直す





リファクタリングのパターン

リファクタリングを上手くやるパターンと、失敗するアンチパターン

リファクタリングのアンチパターン

- × 自動化されたユニットテストでカバーされていないコードを書き換える
↑リファクタリングじゃない!! 単なる書き直し
- × ソースコード管理システムを使わない
↑リファクタリングに失敗したら、戻せない!!
リファクタリング開始前に必ずチェックイン!

リファクタリングのアンチパターン [2/2]

- × 一度にまとめて書き換える
⇒ たいてい大量のレッドが出て、しかも、どこで間違っただのか分からない
- × リファクタリング中に外部設計を変更する
プロダクトコードの書き忘れに気付いても、先にテストを追加してから!

リファクタリング: 作業の基本パターン

- 1. グリーン
関連するテストが全部グリーンになることを確認
- 2. 書き換え
ちょっとだけプロダクトコードを書き換える
- 3. グリーン
関連するテストを全部実行してグリーンのままであることを確認する ⇒ 2. へ戻る
※ レッドになったら、2. の変更を元に戻す

※ テストファーストと同様、インクリメンタル&イテレーティブに進める



リファクタリング: 作業のパターン [シグネチャー変更]

- メソッドのシグネチャーを変更する場合
 - ⇒ 新シグネチャーのメソッドを作ってから置き換える
 - 1. 新シグネチャーのメソッドを作る
 - 旧メソッドをコピーして改造 (←テスト保護無し)
 - 2. 旧メソッドの中身をコメントアウトし、旧メソッドの中から新メソッドを呼び出す (GREEN)
 - 3. ユニットテストで、旧メソッドを呼び出している部分を新メソッドの呼び出しに変更 (GREEN)
 - 4. プロダクトコードで、旧メソッドを呼び出している部分を新メソッドの呼び出しに変更 (GREEN)
 - 5. 旧メソッドを削除する (GREEN)
 - 6. [引数を追加した場合]
 - 追加した引数に対する実装を、テストファーストする
- ※ もっと大がかりなリファクタリングも、考え方は同じ。
プロダクトコードが動かなくなる状況を、どうやって避けるか!?



リファクタリング: 作業のパターン [ツールの利用]

- メソッドのシグネチャを変更する場合
⇒ リファクタリング ツールを使うと楽!

1. グリーン

関連するテストが全部グリーンになることを確認

2. 書き換え

ツールを使って、メソッドと呼び出し部分を一気に書き換える
(右はResharperのダイアログの例)

3. グリーン

関連するテストを全部実行してグリーンのみまであることを確認

4. [引数を追加した場合]

追加した引数に対する実装をテストファースト

Change signature

Define signature changes
Specify new name, return type and parameters, then ensure that signature preview shows the correct signature.

Name:
Parse

Return type:
bool

Parameters:

Type	Name	Modifier	Default Value
string[]	args	<none>	
int	count	<none>	

Buttons: Add, Remove, Move Up, Move Down

Calls:
 Modify
 Delegate via overloading method

Signature preview:
public bool Parse(
string[] args,
int count
)

Keep modified files open

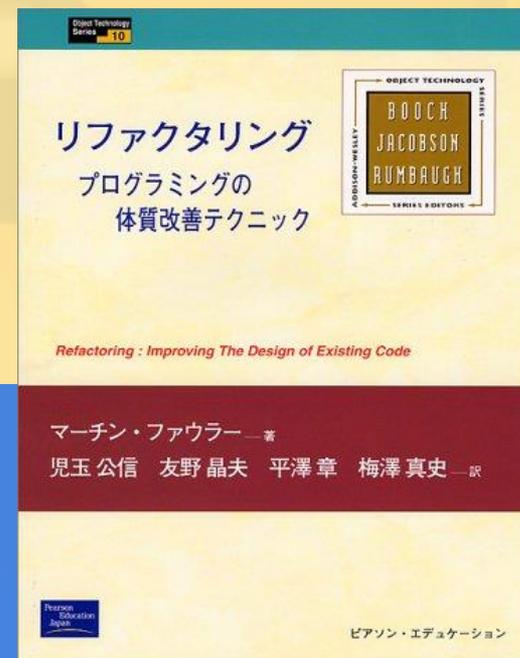
Buttons: Continue, Cancel



リファクタリング: 対象箇所の発見パターン

- コードの不吉な匂いはリファクタリングのサイン
 - * 重複したコード
 - * 長すぎるメソッド
 - * 巨大なクラス
 - * 多すぎる引数
 - * 属性、操作の横恋慕
 - * データの群れ
 - *

リファクタリング
—プログラムの体質改善テクニック
(Martin Fowler 著、2000年) …の第3章 p75～88

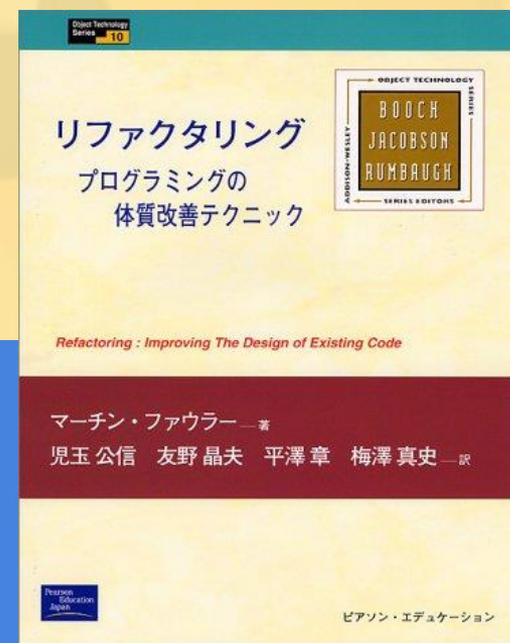


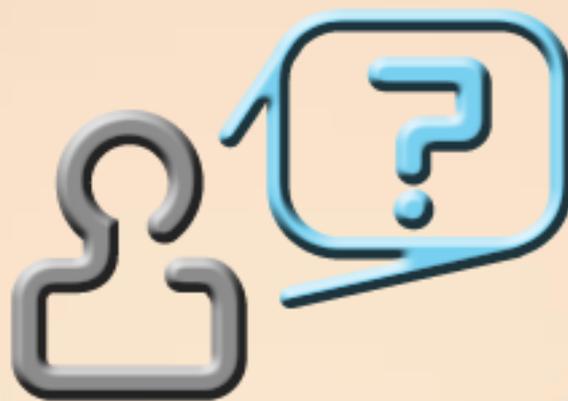
リファクタリング: ゴールのパターン

• リファクタリング カタログ

- * メソッドの抽出
- * メソッドの移動
- * オブジェクトによるデータ値の置き換え
- * 条件記述の分解
- * 問い合わせと更新の分離
- * 委譲による継承の置き換え
- *

リファクタリング
—プログラムの体質改善テクニック
(Martin Fowler 著、2000年) …の第6章～第12章





TDDのデバッグパターン

デバッグと仕様変更の 手順に関するパターン

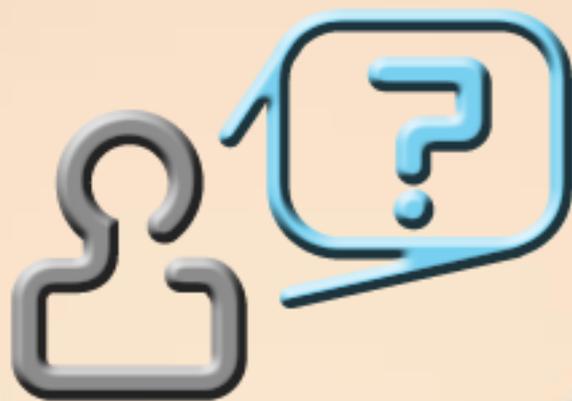
TDDにおけるデバッグ・仕様変更のアンチパターン

- × いきなりプロダクトコードを直す

↑ 自動化されたユニットテストをここで捨てちゃうの!?
もうテストファーストできなくなっちゃうよ!

TDDにおけるデバッグ・仕様変更のパターン

- 1. 修正箇所を特定する
(プロダクトコード または ユニットテスト)
- 2. テストコードを修正する **【テストファースト!】**
 - * テストが間違っていた ⇒ テストを修正
 - * テストが不足していた ⇒ テストを追加
(REDになる。バグを再現したことになる)
 - ※ TDDしたコードのバグとは、外部設計の誤りである
- 3. プロダクトコードを直す
GREENを確認する
必要ならリファクタリングする



TDDの適用パターン

TDDの適用範囲に関するパターン

TDDの適用範囲のアンチパターン

- × 全てのコードをTDDで開発する
UIを持たないコードなら良いけれど…。
あるいは、研究・学習目的ならば、かまわない。
- × TDDしたコードとしてないコードを管理しない
どのような単位でもよいが、きちんと管理する。
アセンブリやパッケージ単位で管理すると、楽。
ちなみに、TDDしたコードはCOカバレッジ100% (のはず)。
- × 使い捨てコードをTDDする
「Kent Beck氏、ごく短期のプロジェクトではテストを省略することを提案」



TDDの適用範囲のパターン

• 仕事としての開発では、コストの低い方法を選択

* テストファーストで余計に必要なコスト

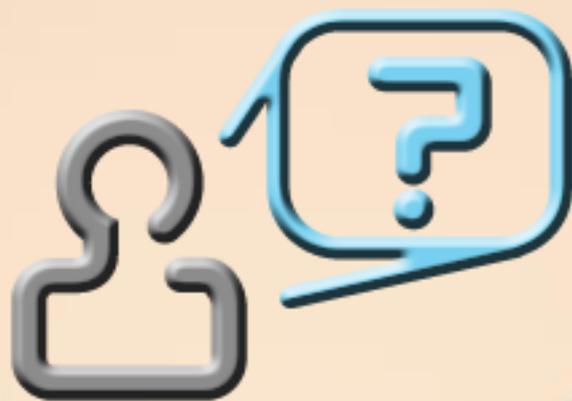
* 手動テスト × 想定される繰り返し回数 のコスト

↑ 安そうな方を選ぶ

一般的に、GUIをテストファーストで作るのは、コストが見合わない。

使い捨てコードや、短期で作り直すことになること分かっているコードも、コストが見合わない。



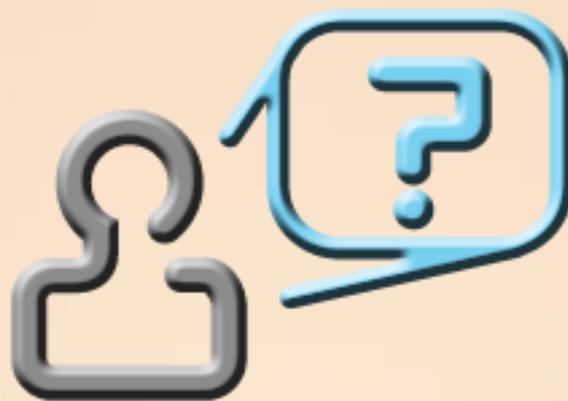


TDDの学習パターン

TDDを学習するときのパターン

TDDの学習パターン

- 1. テストファースト「写経」
テスト駆動開発入門 (Kent Beck 著、2003年)
- 2. リファクタリング「写経」
リファクタリング (Martin Fowler 著、2000年)
- 3. ペアプログラミング
- 4. ソフトウェア開発に関する全て
 - * デザインパターン、アーキテクチャパターン
 - * コーディング技法、テスト技法
 - * 開発プロセス、外部設計(システム)=要件定義、外部設計(メソッド)
 - *



TDDの導入パターン

TDDを組織に導入するときの パターン

TDD導入のアンチパターン

- × 開発プロセス変更の効果を計測できない組織
 - * コード品質を計測していない
 - * 見積り精度が低い = 生産性の変動が計測できない
- × 訓練期間の生産性低下を許容しない
- × TDDを(結合)テストと誤解する
既存のテストを置き換えるものではない
(メソッドレベルの外部設計～テストを置き換えるもの)

最初の2つは、どんなプロセス変更であれ(即効性のあるもの以外は)すべて失敗するパターン。そのような組織は、数字による検討ではなく、ツルの一声で動くことが多いので、TDDの導入には「ツル」を抱き込むことが必須。



まとめ

- さまざまな視点から、TDDのパターン(らしきもの)を見い出してみた。
- **【大事なことなのでもう一度】**
TDDは、
メソッドレベルの開発を
インクリメンタル&イテレーティブに行う



XPエクストリーム・プログラミング入門 (Kent Beck 著、2000年) **【絶版】**



クリック なか見! 検索

XPエクストリーム・プログラミング入門
変化を受け入れる
第2版

XPエクストリーム・プログラミング本]
ゲト ベック (著), Kent Beck (原著), 長瀬 嘉秀
★★★★☆ (4件のカスタマーレビュー)

出品者からお求めいただけます。

中古品の出品: 15¥ 836より

テスト駆動開発入門 (Kent Beck 著、2003年) **【絶版】**



クリック なか見! 検索

テスト駆動開発入門

TDD: TEST-DRIVEN DEVELOPMENT By Example
KENT BECK

テスト駆動開発入門 [単行本]
ゲト ベック (著), Kent Beck (原著), 長瀬 嘉秀
★★★★☆ (12件のカスタマーレビュー)

出品者からお求めいただけます。

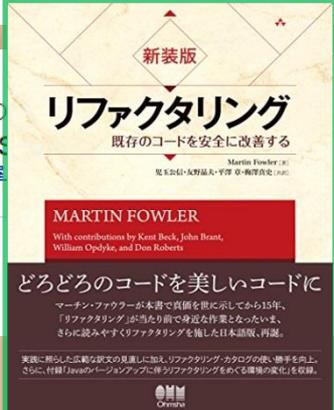
中古品の出品: 7¥ 2,677より

【再刊されました!!】 新装版 リファクタリング (Martin Fowler 著、2014/7/26)

新装版 リファクタリング—既存の (OBJECT TECHNOLOGY SERIES)
Martin Fowler (著), 児玉 公信 (翻訳), 友野 晶夫 (翻訳), 平澤
★★★★★ (1件のカスタマーレビュー)

価格: ¥ 4,536 通常配送無料 詳細

在庫あり。在庫状況について



新装版

リファクタリング
既存のコードを安全に改善する

Martin Fowler (著)
児玉公信・友野晶夫・平澤 幸一 (翻訳)

MARTIN FOWLER
With contributions by Kent Beck, John Beatty,
William Opelyk, and Don Roberts

どろどろのコードを美しいコードに

マーチン・ファウラーが本書で真価を世に示してから15年、「リファクタリング」が当たり前で身近な作業となったいま、さらに読みやすくりファクタリングを施した日本語版、再誌。

実用に限らず広範な読者の見直しに加え、リファクタリング・カタログの使い、手順を向上、さらに、付録「Javaのバージョンアップに伴うリファクタリングをめぐる環境の変化」を収録。

オライリー
O'Reilly

最後に...

日本の開発者は不幸...!? orz





ご清聴ありがとうございました

