

2012/1/14 13:20-16:20

TDD ワークショップ



実際に TDD を体験してみよう

biac

Twitter: @biac
<http://www.tdd-net.jp/>



わんくま同盟 名古屋勉強会 #20

13:20

自己紹介

- 1957年 名古屋生まれ (宇宙時代以前)
- 大学の専攻は航空工学
- 卒業後、HONDA R&D 入社、**機械設計**
- 1994年 プログラマーに転身
- 2001年 Windows XP のリリースと前後して XP と TDD を知る
- 2011年からフリー

宣伝: TDD.NET

- <http://www.tdd-net.jp/>
 - 「VB2010 Express + NUnit 2.5 で、
初めてのTDD Step by Step」 (PDF版, 72ページ)
<http://www.tdd-net.jp/vb2010ee-nunit25-tdd-stepbystep.html>

今日は**C#**でやるけど...

VB.NET で
TDDの実際を
詳しく解説



宣伝: CodeZine にて連載 (予定)

• C#でTDD

第1回は
12/12 掲載

CodeZine
開発者のための実装系Webマガジン

ホーム ニュース 記事 注目ブックマーク カタログナビ ブログ コミュニティ デブサミ2012

C# | Java | VB.NET | C++ | PHP | Ruby | Perl | JavaScript | SQL | Curl | AIR/Flex | 言語一覧

C# 開発/設計/テスト

C#で始めるテスト駆動開発

TDDBC横浜の課題をやってみよう

biac [著] 2011/12/12 14:00

0 ツイート 151

PR 「いいね!」 コアを超える Windows 環境でマルチスレッドプログラミングをしてみる」
PR 「EnterpriseZine」企業システム構築の現場で問題を抱えるITプロが注目する情報はこちらから。

ダウンロード サンプルソース (C#) [3.95 KB]

TDD (テスト駆動開発)

TDDとは、テストファーストとリファクタリングという2つのプラクティスを組み合わせて実施するプログラミング技法です。

TDD = テストファースト + リファクタリング

名称に「開発」と入っていますが、実際にはいわゆる実装工程で利用する技法です。TDDによって「実装時間が2割増えるが、バグは半減する」と言われています^{※1}。

テストファーストは、まずこれから実装するメソッドなどのスペック (外部設計^{※2}) を自動実行できるテストケースとして表現し、次にそのテストケースを満たすように目的のメソッドなど (製品コード) を書く (内部設計) というプラクティスです。

Hot Topics

- Silverlight・WPF トレーニング
XAMLやExpression Blendをハンズオンで学習
- 全7回の集中連載ですばやくマスター!
「速習! Androidアプリケーション開発」
- 従来との違いを具体的なサンプルで解説
「ここが違う! サンプルで見るHTML5」
- Cutがもたらすユーザーエクスペリエンス
最先端の業務アプリ開発環境を体験しよう!

ファイル(F) 検索(N) 無効化(S) 表示(V) イメージ(I) キャッシュ(C) ツール(T) 検証(A) ブラウザー モード: IE9 互換表示(B) ドキュメント モード: IE7 標準(M)

http://web-it-tool.jp/ 100%



わんくま同盟 名古屋勉強会 #20

宣伝: TDD Advent Calendar jp: 2011

- 2011/12/1~12/25
TDDネタで日替わり
で25人がブログ記事
<http://atnd.org/events/22027>

- 電子書籍化
Gihyo Digital Publishing
(技術評論社) から2月上旬
頃に発行予定。無料!
<https://gihyo.jp/dp>
※ 入手には会員登録が必要

偶然、まとめ役をやる事になっ
ちゃったけど、何もしてないw

The screenshot shows the event page for 'TDD Advent Calendar jp: 2011' on the atnd.org website. The page includes details such as the date range (2011/12/01 00:00 to 2011/12/25 23:59), the number of participants (25), and the venue (電網浮遊大陸 (スノーボード, アルク)). It also features a map of the venue, social media links (Twitter, Facebook, etc.), and a list of participants. The page is in Japanese and includes a section for the event rules and a list of participants.



宣伝:「アジャイルマインド」同人誌

- アジャイルマインド勉強会から3月頃発行予定

<http://kokucheese.com/event/index/20715/>

- biacは、製造業から見たアジャイル開発とTDDをネタに、短編小説に挑戦!

The screenshot shows a Mozilla Firefox browser window displaying the event page. The page title is 'アジャイルマインド勉強会～同人誌執筆募集！'. The URL is <http://kokucheese.com/event/index/20715/>. The page content includes a navigation bar with links: 概要, 申込者リスト, お問い合わせ, and キャンセル方法について. The main text describes the event, mentioning that it is a collection of short stories related to agile development and TDD, and that it is open to anyone interested. It also includes a section for '同人誌執筆募集要項' (Event Details) with the following information:

- タイトル: 「アジャイルマインド」(仮)
- 発行時期: 2012年3月予定
- 総ページ数: 150～200ページ程度(調整中)
- 発行部数: 調整中
- 販売価格: 調整中(利益は、イベント費用などに充当する方向で検討中)
- 販売方法: 各種イベントなどで直販予定(調整中)

The page also features a sidebar with a table showing the event status and a section for 'このページは「こくちーず」で作成です' (This page is created with 'kokucheese').



お品書き

- TDDワークショップ

2012/1/14 13:20~16:20

- Part-1: セッション (約1時間)
「TDDって、どんなこと?」
- Part-2: ワークショップ (約1時間)
「TDDをやってみよう!」
- Part-3: レビュー大会 (約40分)
「みんなでコードレビュー♪」

LT登壇者募集!

- 16:20～ ライトニングトーク

お題: テスト・TDDに関すること

スライド資料は無しで構いません。

5分間、

あなたの思いを話してみませんか!

PART - 1

TDD って
どんなこと?

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」と言うけど、**実装**の話。



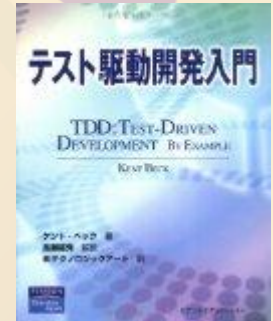
TDD とは?

- **Kent Beck:** テスト駆動開発 (TDD) の提唱者
「**テスト駆動開発入門**」 (原著 2002/11)
の前書きより抜粋。

In Test-Driven Development, we

- * **Write new code only if an automated test has failed**
- * **Eliminate duplication**

These are two simple rules



TDD では、

- ・ 自動テストが失敗した場合のみ、新しいコードを書く。
- ・ 重複を取り除く。

これらは 2つのシンプルな規則だ。

※ 詳細 ⇒ <http://www.tdd-net.jp/2011/12/tdd-kent-beck-5.html>



TDD = テストファースト + リファクタリング

- Kent Beck の定義 (再掲)
 - ・ 自動テストが失敗した場合に限って、新しいコードを書く。
 - ・ 重複を取り除く。
- ↑それぞれ、名前が付いている
ひとつめ = **テストファースト**
ふたつめ = **リファクタリング**

※ Kent Beck の言う「重複を取り除く」には、リファクタリングの結果としてデザインパターンを取り入れることまで含んでいる！
言葉の表面に騙されるな!!

TDD = テストファースト + リファクタリング

- テストファースト = 外部設計

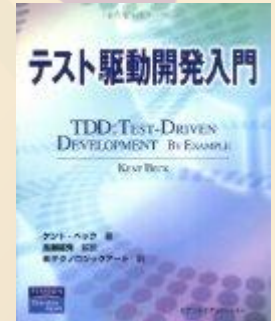
- メソッドレベルの外部設計を、テストケースとして少しずつ決めていきながら、実現できることも立証する。

- リファクタリング = 内部設計

- メソッドレベルの外部設計を固定し、かつ、作ることができることを証明してから、メソッドの内部をどうするか考える。

TDD = RED → GREEN → Refactor

- **Kent Beck**: テスト駆動開発 (TDD) の提唱者
「**テスト駆動開発入門**」 (原著 2002/11)
の前書きより抜粋 (2つの規則の続き)。



2つの規則はプログラミングのタスクにおける順番を意味する。

- **レッド** - 動作しないテストを少しだけ作成する。
おそらく最初はコンパイルできない。
- **グリーン** - テストをすぐに動作させる。
そのためには、どのようなコードでもよい。
- **リファクタリング** - テストを動作させるためだけに作成された**重複**をすべて取り除く。

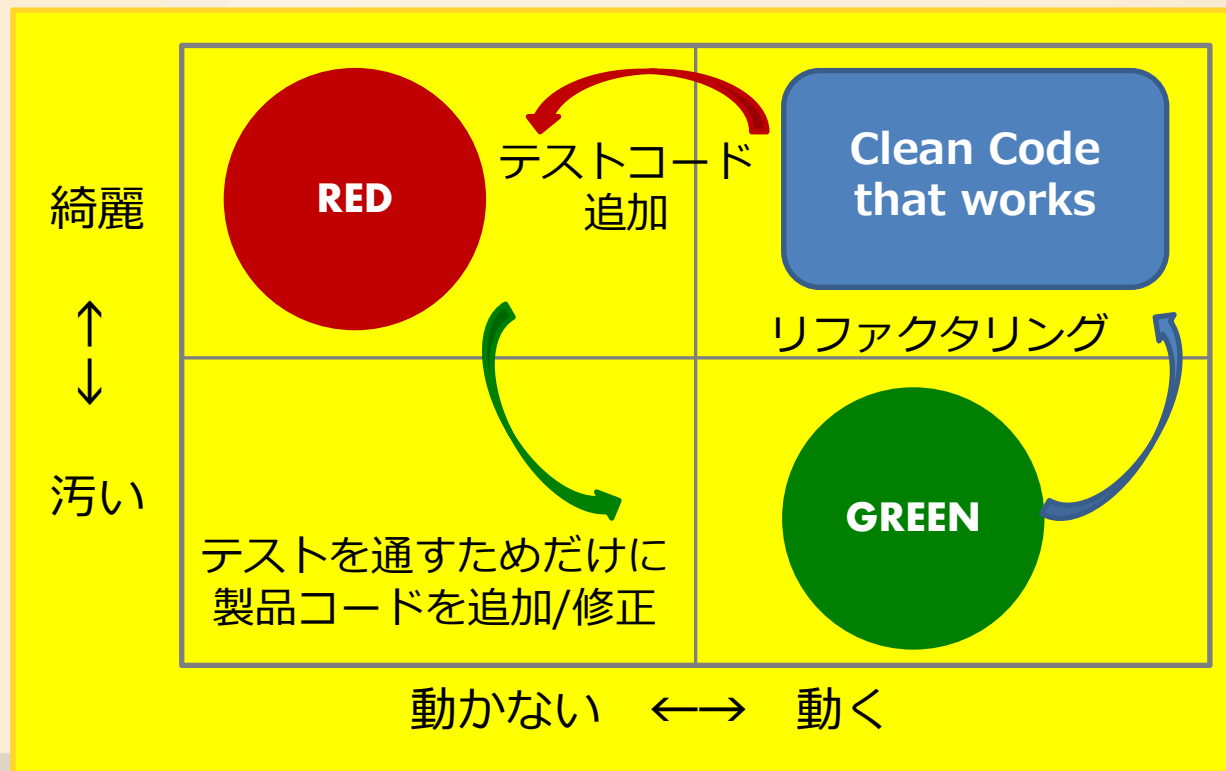
テスト
ファースト



TDD = RED → GREEN → Refactor

● 「黄金の回転」

日本のTDD伝道師 和田卓人氏 (@t_wada)は、RED → GREEN → リファクタリング の繰り返しを、こう呼



TDDのゴールは

**Clean code
that works**

動作するきれいなコード

by Ron Jeffries

※ 和田氏の図を、説明しやすいように改変した。
オリジナルでは、左下から右上に至る「二つの道」の説明から入るため、REDが左下に来る。

TDD と 実際の開発

- 実際の開発では、
TDD したり しなかったりする!

TDDするコスト < 後々手動テストするコスト
⇒ その部分は TDD する (他はしない)

- 例)
TDDする: ロジック, MVVMのVM など
TDDしない: UI, 間違えそうもない単純な
setter/getter, 定数定義 etc.

テストファースト

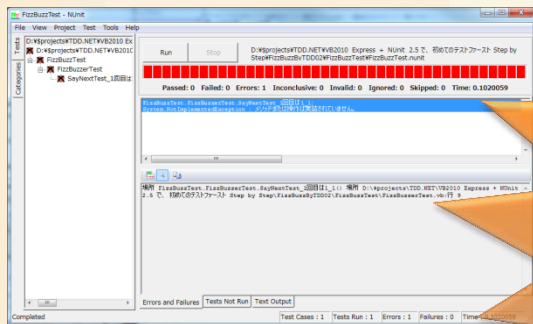
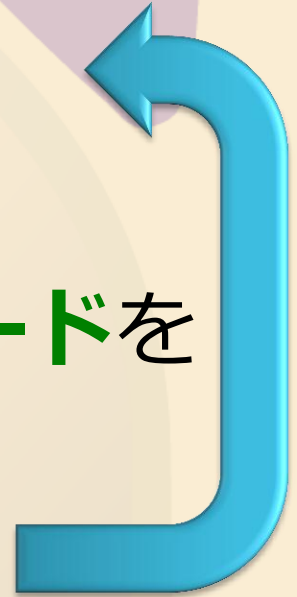
TEST
FIRST

テストファースト: RED → GREEN

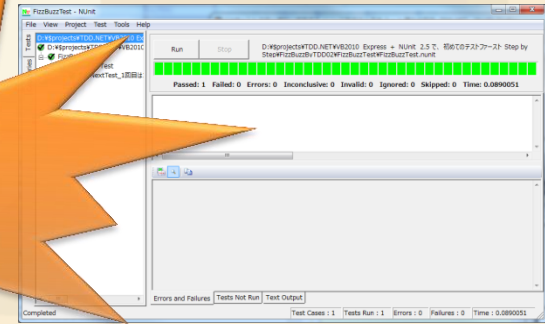
- **テストケース**をひとつ書く
失敗することを確認する (**RED**)



テストケースに通るだけの**製品コード**を
書く
成功することを確認する (**GREEN**)



テストケース
ファーストと
呼ぶべきかも



テストファースト: 先に外部設計を決める

- **メソッドの外部設計** = メソッド利用者に対する**公約**、早く決めねば!

メソッドの中身 - 後からいくらでも変更可能。
みんな、メソッドの外部設計やってる?

- **テストケース** = 外部設計の**厳密な例示**

メソッドの外部設計 = 入出力、オブジェクトの状態遷移

日本語による仕様書 = あいまい

数式やコードによる表現 = 厳密

例示: 全ての入出力をテストコードで記述することは非現実的

テストファースト: 実際にやってみよう

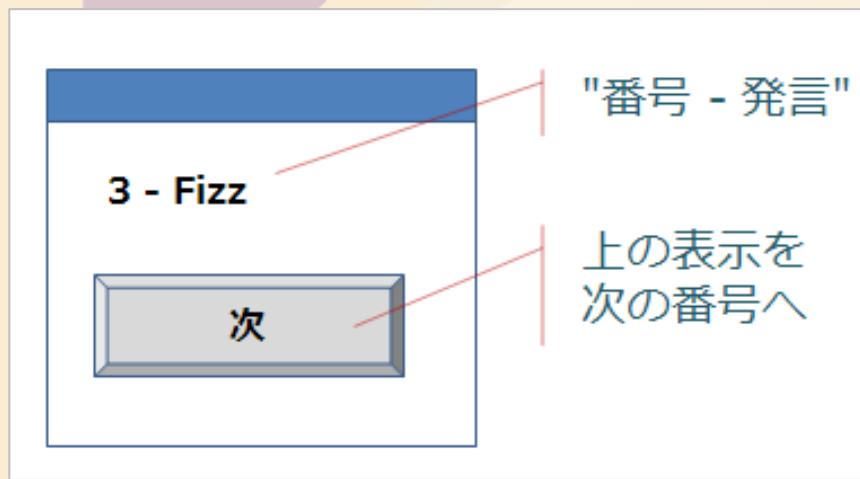
…と言っても。

現状、プログラム全部をテストファーストで作ることは、まず無い。

⇒ **UI 部分は大変すぎ**

※ UIは従来のやり方、ロジックだけTDDでやることが多い

外部設計例: FizzBuzz 画面



プログラムの
外部設計

- 最初は "1 - 1"、次は "2 - 2"、"3 - Fizz"...
- Fizz Buzz ルール
 - 番号が3の倍数のとき、"**Fizz**" と発言する
 - 番号が5の倍数のとき、"**Buzz**"
 - 番号が3と5の倍数のときは "**Fizz Buzz**"

外部設計例: FizzBuzz 刺激と応答

呼び出し (刺激)	返値 (応答)
1回目	"1 - 1"
2回目	"2 - 2"
3回目	"3 - Fizz"
4回目	"4 - 4"
5回目	"5 - Buzz"
6回目	"6 - Fizz"
7回目	"7 - 7"

呼び出し	返値
14回目	"14 - 14"
15回目	"15 - Fizz Buzz"
16回目	"16 - 16"

※ Integer.MaxValue の前後は、
今回は省略します。

メソッドの
外部設計

この表の全部をテストコードに
書くわけじゃないからねっ!!

デモ: テストファーストで *FizzBuzz*

DEMO

- Visual C# 2010 Express
- NUnit 2.6(β)
- ※ UI は作ってあるものとする

DEMO: 最初の方だけちょっと書いておこう

- DEMO中にいくつかTDD用語を喋るので、その説明も兼ねて、最初のほうだけスライドにも載せておきます。

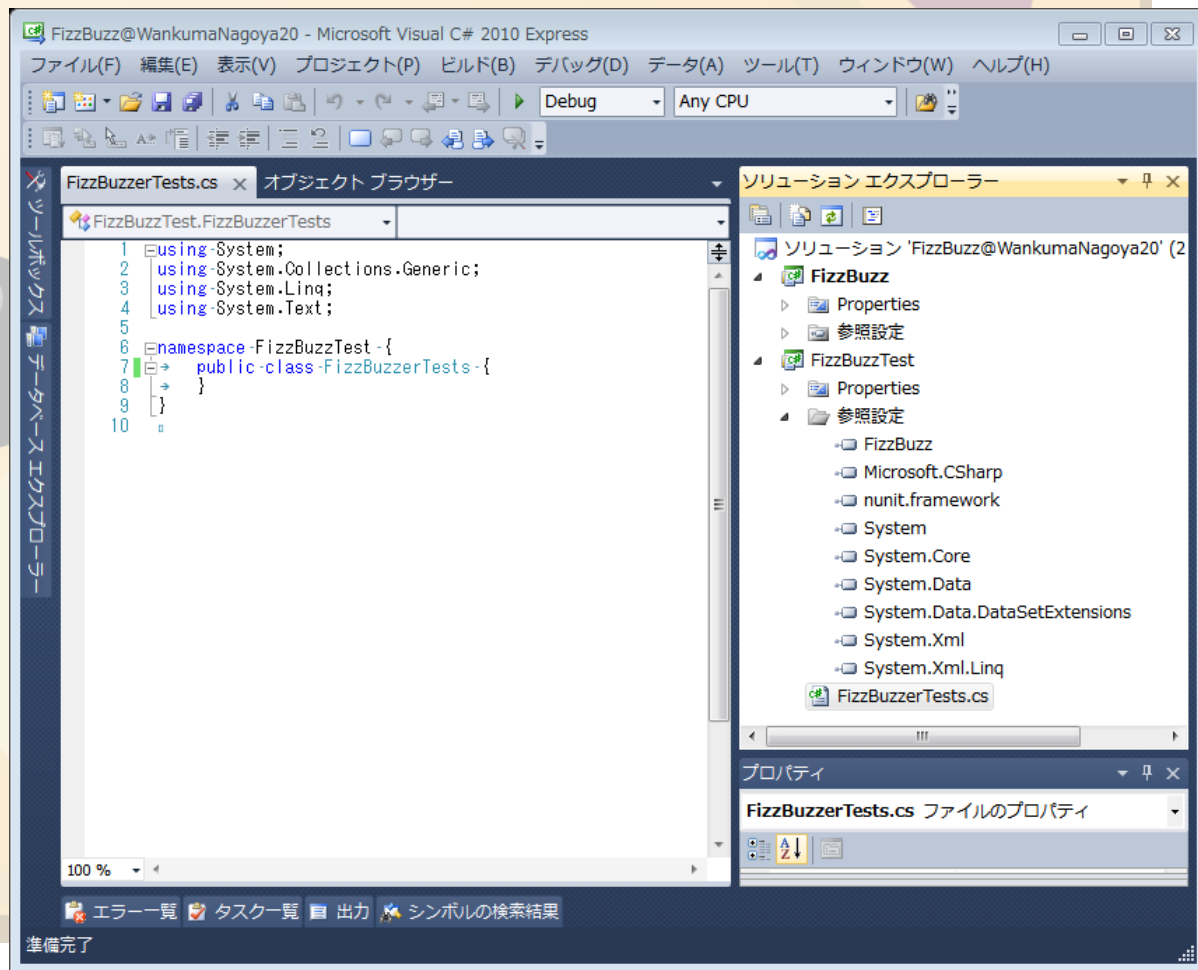


DEMO: (1) 初めてのRED (1/4)

- 最初が一番簡単そうなところからなにしろ、クラス名・メソッドのシグネチャなど、決めなきゃならんことがいっぱいある!!
- その前に、プログラムを書き始める準備 Visual Studio だと…
 - ソリューション(ワークスペース)を作る
 - プロジェクトを作る (テスト用、製品用)
 - ビルド出来るようにする (参照設定)
 - テストコードを書くクラス FizzBuzzerTests

DEMO: (1) 初めてのRED (2/4)

- ようやくプログラムを書く準備ができた



DEMO: (1) 初めてのRED (3/4)

- 最初のテストケース

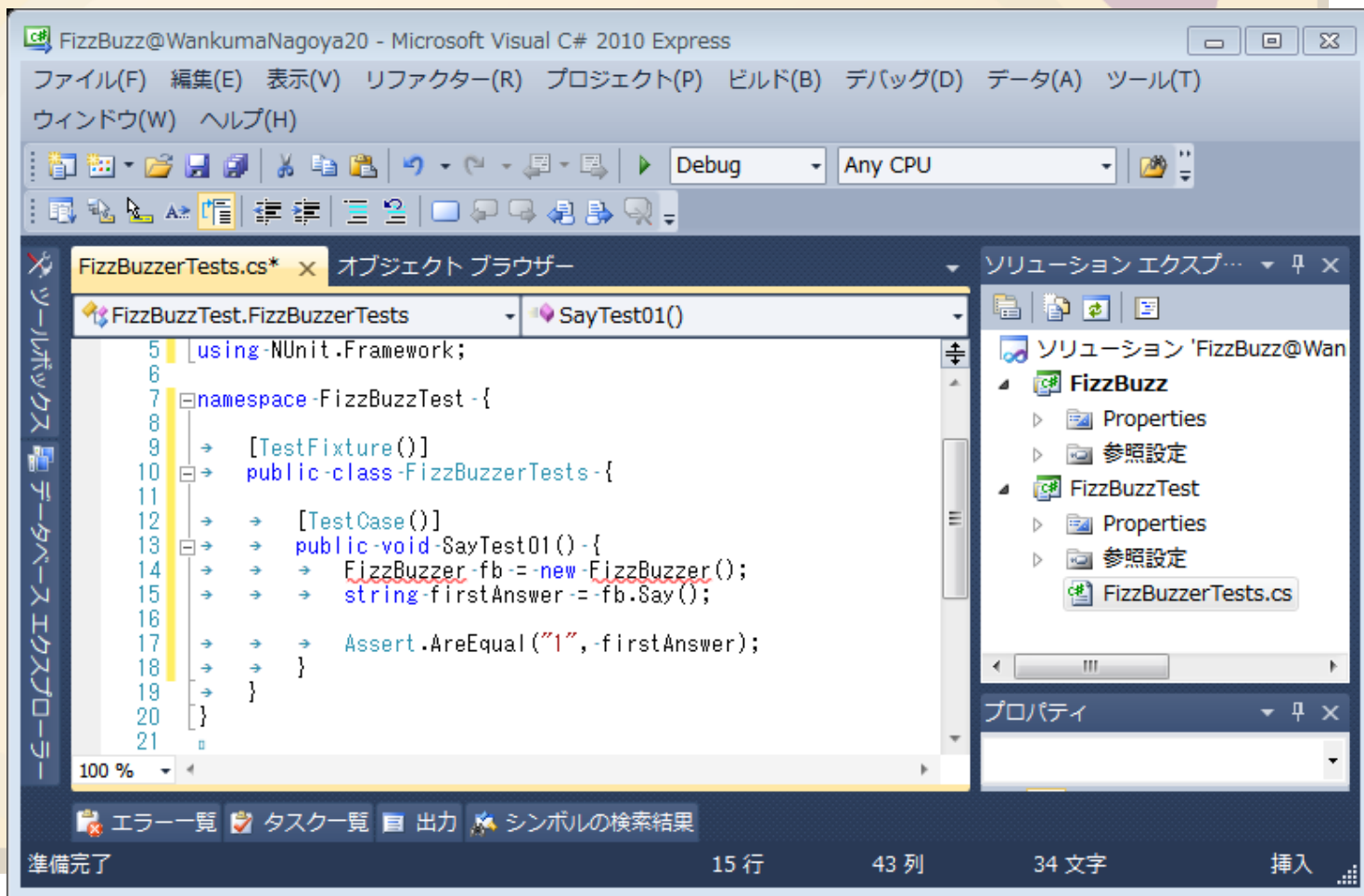
[TestCase()]

```
public void SayTest01() {  
    FizzBuzz fb = new FizzBuzz();  
    string firstAnswer = fb.Say();  
  
    Assert.AreEqual("1", firstAnswer);  
}
```

※ まだ FizzBuzz クラスを作っていないから、コンパイラーに怒られる。(赤線部分)

DEMO: (1) 初めてのRED (4/4)

- 最初はコンパイル出来ない = RED



DEMO: (2) 初めてのGREEN (1/2)

- RED(自動テストが失敗した)ので、
ようやく製品コードを書ける!
 - FizzBuzzerクラスを作る
 - Say() メソッドのスケルトンを作る
 - メソッドの実装は…?

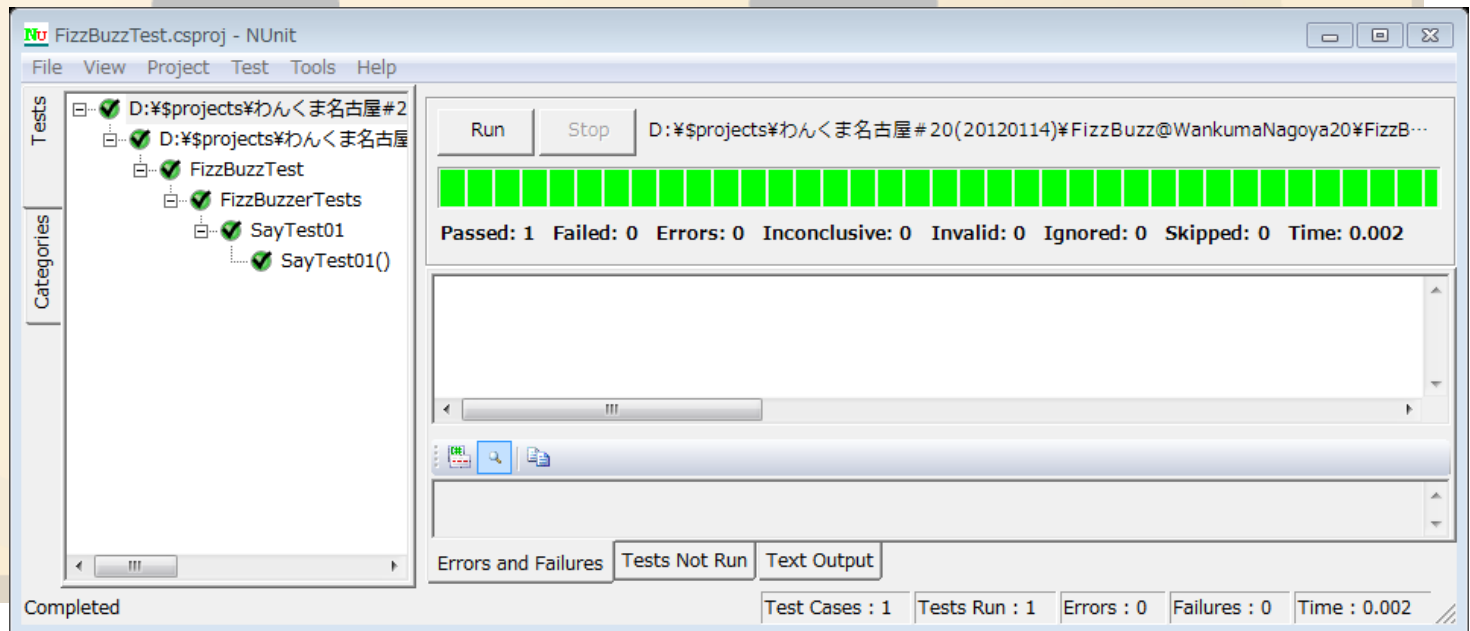


DEMO: (2) 初めてのGREEN (2/2)

- ともかくテストに合格すればいいのよ~!

```
public string Say() {  
    return "1";  
}
```

「仮実装」
(Fake It)



DEMO: (3) 三角測量 (1/4)

- 仮実装のままじゃダメ!
もうひとつテストケースを追加しよう。

2つのテストケースで
製品コードの1ヶ所を
照らし出す!

「三角測量」
(Triangulate)

※ ちなみに、ここで Kent Beck だと、「テストコードにも製品コードにも "1" というリテラルがある。重複している!」、「重複を解消するためにテストを追加する!」、という具合に話が進むはず。「重複の排除」万能!!

DEMO: (3) 三角測量 (2/4)

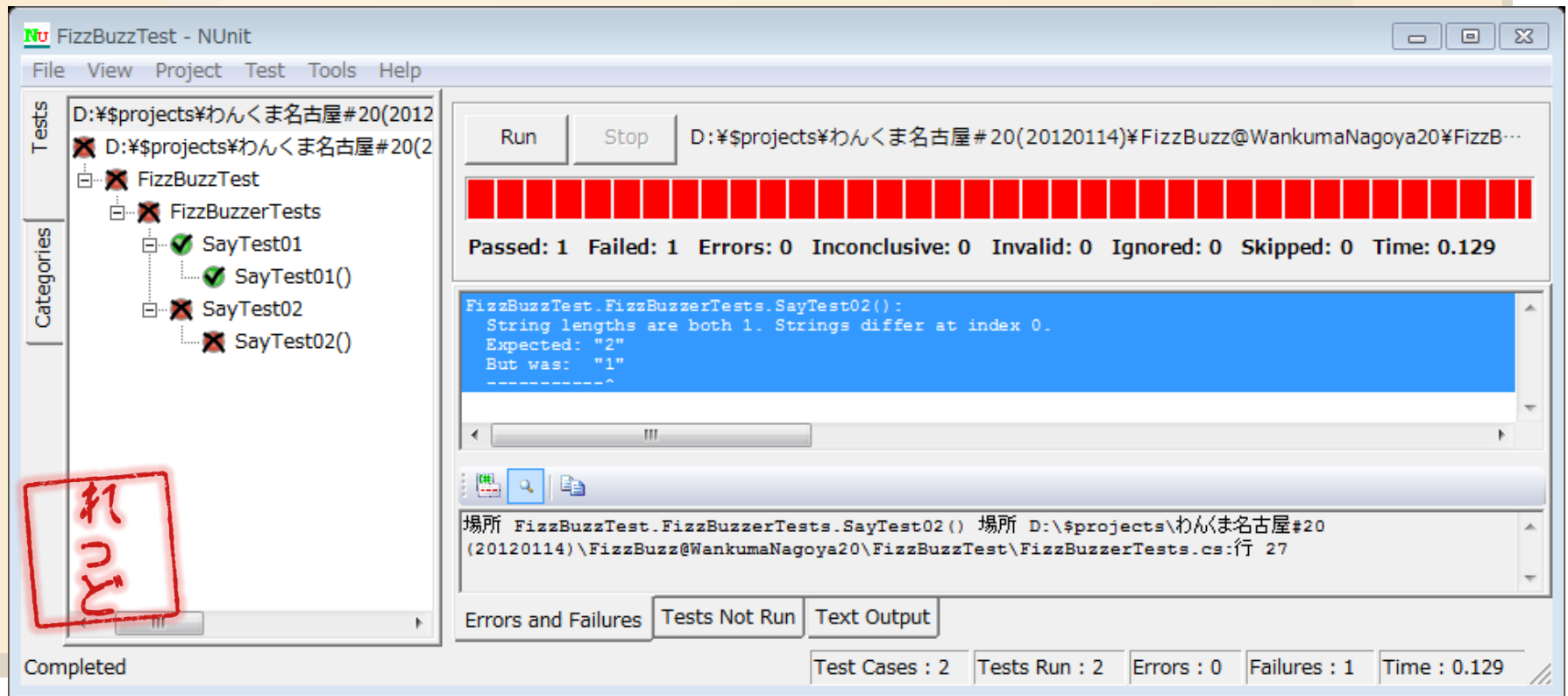
- 2つめのテストケース

[TestCase()]

```
public void SayTest02() {  
    FizzBuzz fb = new FizzBuzz();  
    string firstAnswer = fb.Say();  
    string secondAnswer = fb.Say();  
  
    Assert.AreEqual("2", secondAnswer);  
}
```

DEMO: (3) 三角測量 (3/4)

- 追加したテストケースは、RED
※予想通りのREDになったか? ←重要!



れこど

Completed

Test Cases : 2 Tests Run : 2 Errors : 0 Failures : 1 Time : 0.129



DEMO: (3) 三角測量 (4/4)

- 2つのテストをクリアするコードは?

```
private int callCount;
```

```
public string Say() {  
    callCount++;  
    return callCount.ToString();  
}
```



DEMO: (4) テストをリファクタリング

- 2つテストを書いた。
同じようなテストをこれからも追加する。
ひとつにまとめておこう!

```
[TestCase(1, "1")]  
[TestCase(2, "2")]  
public void SayTest(int count, string expected) {  
    FizzBuzz fb = new FizzBuzz();  
    string answer = null;  
    for (int i = 0; i < count; i++)  
        answer = fb.Say();  
    Assert.AreEqual(expected, answer);  
}
```



- ※ 1. このテストを追加して、ALL GREEN を確認。
- ※ 2. 不安なら、製品コードを一時的に変えて RED を確認。
- ※ 3. 元のテスト(2つ)を削除

DEMO: (5) 明白な実装

- 次は、「3回目のときは "Fizz"」



[TestCase(3, "Fizz")]

- 製品コードは、すぐ書ける

```
public string Say() {  
    callCount++;
```

```
    if (callCount % 3 == 0)  
        return "Fizz";
```



```
    return callCount.ToString();
```

「明白な実装」

Obvious
Implementation

テストファースト: TDD三原則

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。

※ 出典: Robert C Martin (UncleBob) (翻訳 安井力)

<http://www.butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

<http://twitter.com/unclebobmartin>

<http://twitter.com/yattom>

※ 原題は "The Three Laws of TDD."。

いずれにせよ、TDD全体ではなく、**テストファースト**だけにに関するルール。

テストファースト: 小刻みの法則

- テストケース単位でひとつずつ進む
※ TDD三原則の 1. ~ 3.

- ー 一度に取り組む問題を小さくする
- ー すばやいフィードバックを得る
↑ 機械設計者の悲願

1日に何十回でも
試作できるなんて!!

テストファースト: 終了条件

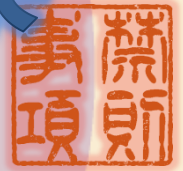
- 失敗するテストケースをもう思いつかない
⇒ テストファースト**終了**
 - ※ TDD三原則の 2.
 - ※ 外部設計を満たすコードが書けることは証明できた。
ここでチェックイン! / コミット!
- 追加のテストケースを書いても良い
 - ※ TDDから外れる, メンテするテストコードが増える
 - 安心を得る
 - APIドキュメントとして

テストファースト: 効果



- 仕様書のレビュー
- ムダなコードは無い
 - ※ すべてのコードが、テストケースを通すために必要 (C0 カバレッジ100%)
- 使いやすいメソッド (外部設計優先)
- Combat Proven (**戦闘証明**済み)

テストファースト: 得られない効果



- **品質保証ではない**

- 開発者が理解したメソッドの外部設計が、プログラム仕様を満たしている保証は無い。
(そのプログラム仕様も、顧客の要求を満たしている保証は無い。) …実装の話なんだから、あたりまえ!
- テストケースを網羅していない (←TDD三原則の2.)

- **アーキテクチャ設計ではない**

- TDDの中からアーキテクチャは出てくるが、実際にはそれでは遅い (とくに多人数開発)

リファクタリング

リファクタ
リング

リファクタリング: 後で内部設計を考える

- 外部設計(公約)を変えない限り、**中身はどれだけ変えてもいい**じゃないか!
 - テストファーストで外部設計は固まった
 - 全テスト合格(=外部設計不変)を維持しながら内部設計を洗練する

テストファースト完了時は、
たとえば
ブレッドボードで組んだ回路

リファクタリング: *GREEN! GREEN! GREEN!*

- 良くなる方向を考える



製品コードを**ちょっと**直す



テストが成功することを確認する
(**GREEN**)



※ 大股に歩くとコケるぞ!
(そのときはいさぎよく戻す)

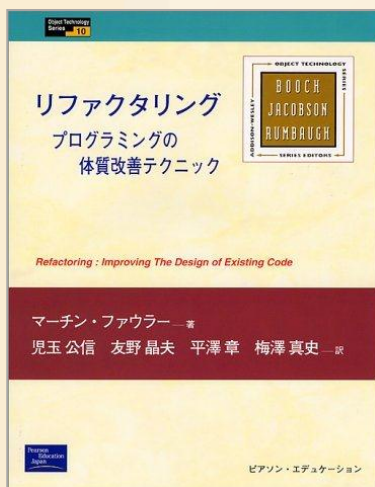
デモ: FizzBuzz のリファクタリング

DEMO

- Visual C# 2010 Express
- NUnit 2.6(β)
- テストファーストは、さっき終わった

リファクタリング: 技法

- ファウラー本を読んでね!



「リファクタリング — プログラムの体質改善テクニック」

マーチン ファウラー (著)

ピアソンエデュケーション (2000/05)

ISBN-13: 978-4894712287

<http://www.amazon.co.jp/exec/obidos/ASIN/4894712288/bluewatersoft-22>

リファクタリング: 目的

- リファクタリングを世に広めた **Martin Fowler** の言葉

リファクタリングとは、**理解や修正が簡単になるための変更**だと私は思っている。同じ変更でも、目的が異なれば、それはリファクタリングとは言わない。

- Martin Fowler's Bliki in Japanese - 「リファクタリングの境界線」
<http://capsctrl.que.jp/kdmsnr/wiki/bliki/?RefactoringBoundary>

リファクタリング: 目安

- 三歩歩いたら読めなくなりそうなコード
⇒ **必須**。ヤレ、ゼツタイ!
- メンテナンスしにくそうだ
⇒ **余裕があれば**
- 将来の拡張に備える
⇒ **実際に拡張するときに**

リファクタリング: 効果

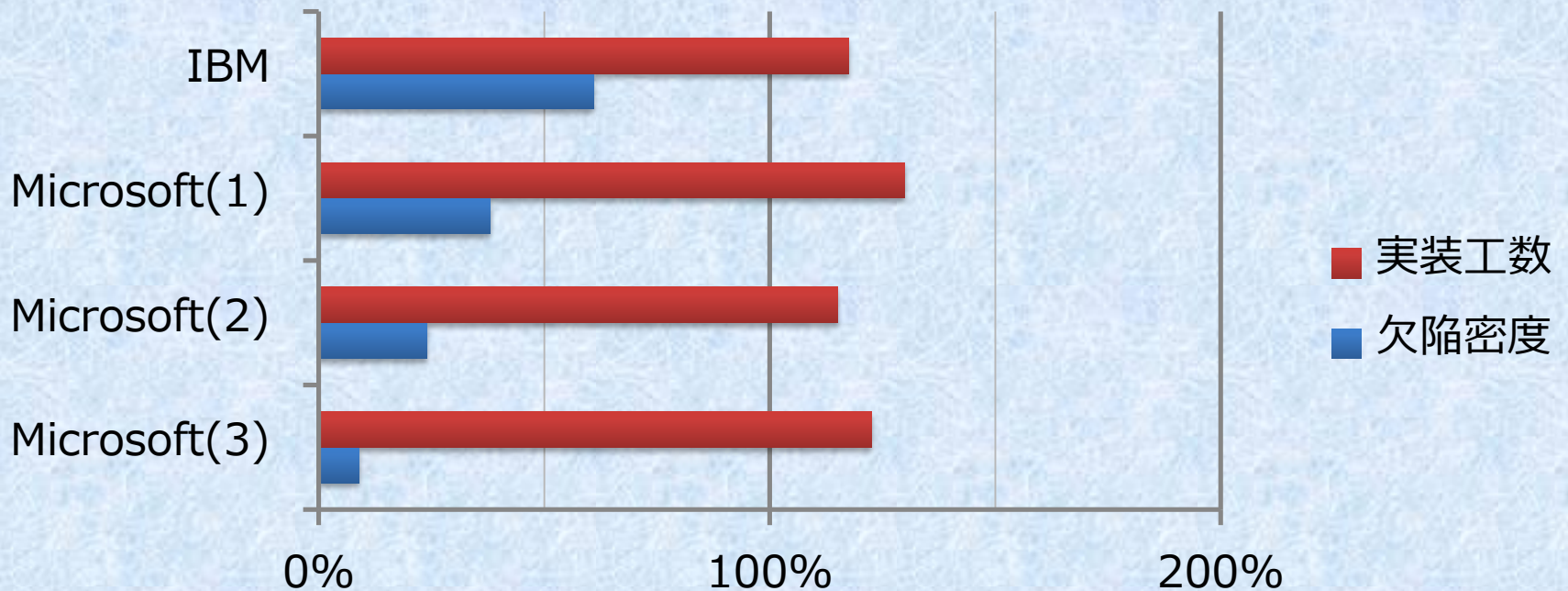
- きれいなコード
 - ・ 読みやすいコード
 - ・ 重複の無いコード



- ・ 追加/修正コストの削減（開発中、カットオーバーまでにも頻繁に発生）
- ・ なんととっても、**気持ちよくコードを書ける！**

※ オマケ効果：発見「こんな書き方も出来るんだ！」

TDD の効果、実例



実装工数は2割増えるが、バグは半分以下になる！

(バグ対応工数 --- バグレポ/トリアーージ/デバッグ/再テスト/トラッキングetc.
の工数 --- が半分以下になる、ということ。)

出典：論文 "Realizing quality improvement through test driven development: results and experiences of four industrial teams" (N. Nagappan 他, 2008) より。

http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf



TDD = テストファースト + リファクタリング



TDD の達人になるには？

- 1に鍛錬
 - "Software Craftsman" Uncle Bob の言葉 「TDD は学術研究の話じゃない、**鍛錬**だよ。」
"TDD is a discipline, not an area of academic study."
<http://twitter.com/unclebobmartin/status/36839149881270272>
 - 写経、ペアプロ、道場
- 2に勉強 …TDDって、ようするに実装の話だもん
 - よいコード、デザインパターン、Mockの使い方、ソースコードリポジトリ… **全部!**
- F# とか関数型言語もやるといいかも

PART - II

TDD を
やってみよう!

チーム作り: 席替えタイム

言語ごとに
チームを作るくま～



わんくま
同盟

わんくま同盟 名古屋勉強会 #20

要求

- キャッシュの最大数に達したときに、最も使われていないデータから順に消される Map のような仕組みが欲しい。
 - Last Recently Used Cache
 - void Put(string キー, string データ)
※Java系では void put(string キー, string データ)
 - string Get(string キー)

要求: 例

- [キャッシュサイズ=2 のとき]

- ひとつも使われていない場合は、最初に追加したものから消える

```
lru.Put("a", "dataA");  
lru.Put("b", "dataB");  
lru.Put("c", "dataC");    // 先に入れた "a" が消える  
lru.Get("a");              // ⇒ null  
lru.Get("b");              // ⇒ "dataB"
```

- Get() されたら使われたとみなす

```
lru.Put("a", "dataA");  
lru.Put("b", "dataB");  
lru.Get("a");              // ⇒ "dataA"  
lru.Put("c", "dataC");    // 使われていない "b" が消える  
lru.Get("a");              // ⇒ "dataA"  
lru.Get("b");              // ⇒ null
```

お題

- (印刷したもの(A4-1枚)を配布します)
- ※クラスやメソッドの外部設計(スペック)の考え方
特定の条件なら簡単 → だんだんと条件を広げていく

「要求」は難しいから、
3つの「お題」に分けるくま～

TDD しよう!

- 約1時間 (15:40まで)
※ 早く完成したチームは手を挙げて!

チーム内でペアプロとかして
助け合うくまよ~

お題 その1: Map (のようなもの)

- クラス名: LruCacheMap

- メソッド:

void Put(string キー, string データ)

string Get(string キー)

※ 必要なら public メソッドも追加して良い。

キャッシュの最大数に達するまでは、
この仕様でも間違っていないくまよ～

- 何個でも入る。
- データに null は入る。
- キーに null は不可。 Put()/Get() とも、「引数に null が渡された」という例外が発生する。
- 同じキーで Put() できる。 データが置き換わる。
- Get() は、該当するデータがないときは null を返す。

お題 その2: キャッシュサイズの導入

- LruCacheMap に入るデータは 2件までとする。
- 2件入っているときに、さらに異なるキーを Put() すると、最初のデータが消える。



消えるデータは次のお題で変わっちゃうけど、
この簡単な仕様で一回作ってみるくまよ～

お題 その3:「データを使う」概念の導入

- Get() でデータを取り出した時、そのときに「使われた」と考える。
- Put() されたときも、「使われた」と考える。
- 2件入っているときに、さらに異なるキーを Put() すると、最も以前に「使われた」データが消える。(要求の例では、2通りに分けて表現されていた)
 - ※ 「使われた」ことをどうやって表現するか?
時刻? 並び順?

参考: お題その1の刺激と応答

- void Put(string キー, string データ)

事前状態 保持データ	引数(刺激)			応答	事後状態 保持データ
	キー	キーの重複	データ		
(any)	null	---	(any)	「引数に null が渡された」例外	変化無し
無し	null 以外	---		無し	1個
1個以上		有り			個数に変化無し、キーが合致したデータは置き換わる
		無し			1個増加

- string Get(string キー)

事前状態 保持データ	引数(刺激)		応答	事後状態 保持データ
	キー	キーの合致		
(any)	null	---	「引数に null が渡された」例外	変化無し
1個以上	null 以外	無し	null	
		有り	キーが合致するデータ	



楽しい(?)仕様変更のお時間

- お題その4: 仕様変更(1)
キャッシュサイズを変更できるように。
 - ※ 減らした時の挙動はどうする?
 - ※ 0個にすることを許す?
- お題その5: 仕様変更(2)
一定時間使われていないデータを削除できるようにする。
 - ※ 「一定時間」は固定でいいか? 可変にするか?
 - ※ どのタイミングで削除する? Put()? Get()?
 - 新メソッド? タイマー?
- お題その6: 仕様変更(3)
Put()/Get()をスレッドセーフにする。
 - ※ どうやってテストを書こう?

PART - III

みんなで
レビ्यूー ♪

レビュー進行

- 1. 各言語チームから、ひとりずつ発表者を選出。(登壇はチーム全員!)
- 2. スクリーンにコードを表示しながら、発表者が解説。
- 3. 参加者全員で質疑応答。

16:20で終了くま～

残り時間÷チーム数＝持ち時間

発表するときの観点

- テストコードを全部説明していると長くなるので、観点を絞ってください。
 - **これはウマく出来た!** --- 上手くできたこと、閃いたテストケースの書き方、美しくできた製品コード、などを自慢してください。
 - **障害と生存戦略** --- テストケースを書くときに問題になったこと、および、その解決策。
 - **もうひとつやりたいリファクタリング** (もっともやっておきたいことを、ひとつだけ)

PART - III

レビキュータイム♪

最後に: 製造業から見るTDD

- TDDは「モノを作るときの当たり前」を細かく繰り返しているだけ!

どういうものを作ろうか?	RED (テストケースで表現)
こうやったらともかく作れるぞ!	GREEN (動けばいいから!)
もっと改良しようじゃなイカ!?	リファクタリング

※しかし、モノ作りの本家である製造業には、TDDの繰り返しスピードは真似できない!

ありがとう

ございました