

NOW LOADING•••



@martin\_lover\_se

# 今日から実践！TDD (テスト駆動開発)入門

～今、お前のレガシーコードの危険が危ない～



**Let's Start TDD !**



## ちょっと前置き

- ・ レガシーコードなにそれおいしいの？
- ・ レガシー( legacy )： 遺産/遺物
  - なんか古臭いもの？？
- ・ じゃあレガシーコードって？
  - 「何年も前に誰かが作り、内容が複雑で何をしているのかよく分からず、まともな仕様書もない」コード？

## ちょっと前置き

### ■こんな経験、ないですか…

- ・ これ作ったやつ絶対殴る。3回だ。
  - 1万行を超えるクラス…
  - 1000行を超えるメソッド…
  - 20階層ネストしたif文…
  - C0001みたいな意味不明な連番のクラス名…
  - 95%同じロジックのコピペメソッド…
  - マジックナンバー祭り…

嗚呼、何とレガシーか！

## ちょっと前置き

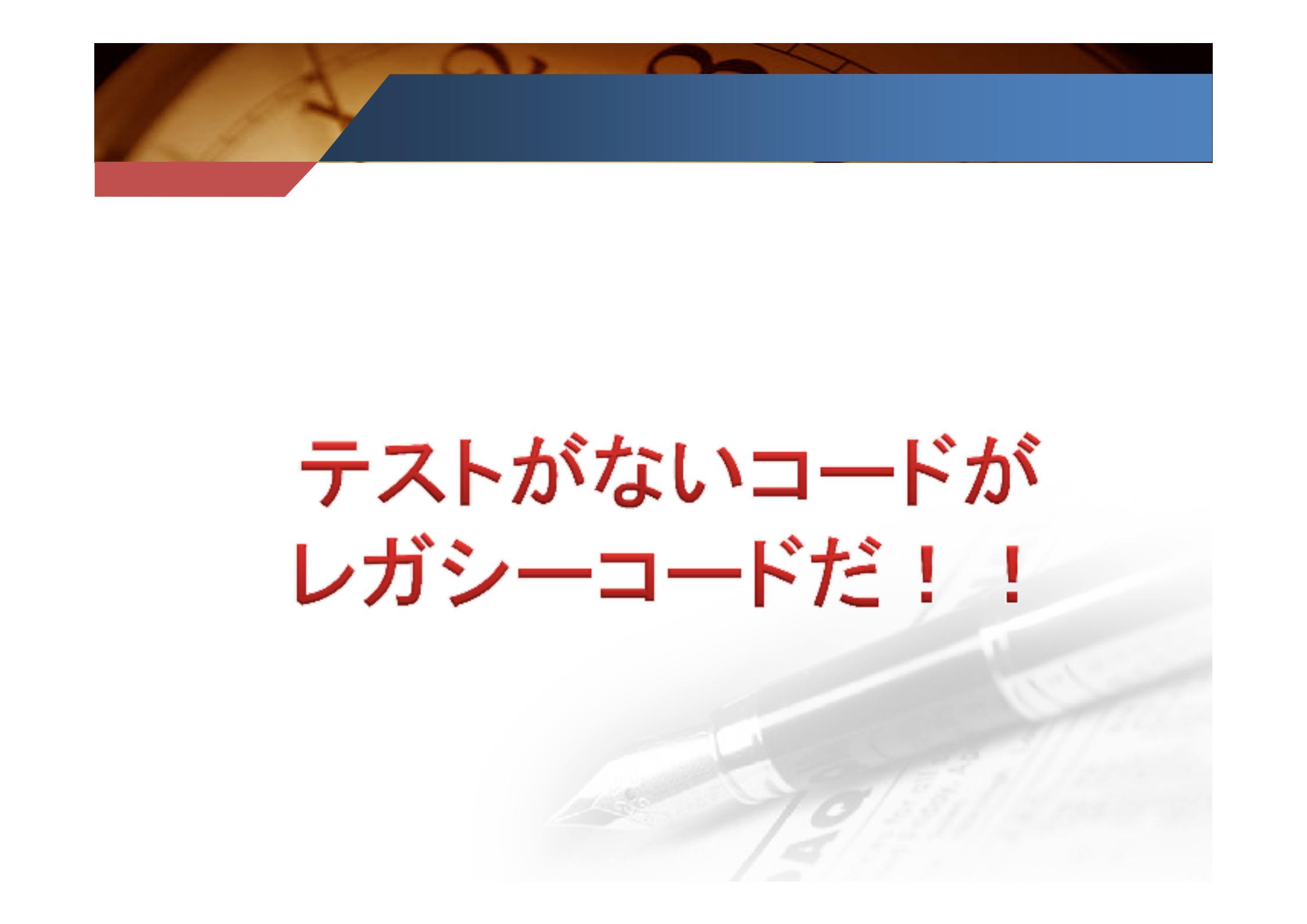
### ■こんな経験、ないですか…

- ・ “動いているコードは触るな”の怪
  - 馬鹿野郎てめえ今なおしたとこの影響範囲全部試験やり直す気か！！
- ・ 「編集して祈る」
  - 他の箇所への影響を細心の注意を払って調査し、奇跡的に目的の機能以外への影響がないことを祈りながらリリースする…

**嗚呼、何とレガシーか！**



違う、そうじゃない



テストがないコードが  
レガシーコードだ！！

# ちょっと前置き

## ■主は言った。「汝テストコードを書け」

- ・ 「**テストがないコードはレガシーコードだ**」  
(マイケル・C・フェザーズ/レガシーコード改善ガイド)
  - ・ どれだけうまく書かれていようと、テストがなければ、コードが良くなっているのか悪くなっているのかが本当には分からない。
- ・ 諸悪の根源は、**テストコードがないこと**
  - ・ 動作が保証された範囲がわからない！
  - ・ ソースコードをきれいにすることもできない！
- ・ 「**保護して変更する**」
  - ・ テストを用意したうえで既存のコードを修正する  
→影響がわかれれば修正は怖くない！



# TDDを始めよう！

## ■ TDD(テスト駆動開発)

- ・ Test Driven Developmentの略
- ・ テストから先に作る開発手法
  - ・ テストの技法ではない
  - ・ アジャイル開発(XP:eXtreme Programming)のプラクティスの1つ
- ・ テスティングフレームワークによるテスト自動化
- ・ TDDのメリット
  - ・ いつでもデプロイ可能な高品質なプログラム
  - ・ 再実行可能なテストコード
    - － いずれも短期間での開発(イテレーション)を繰り返すアジャイル開発では超重要！

# TDDを始めよう！

## ■ TDDが与えてくれるもの

- ・ プログラミングの楽しさの再発見
  - “作ったものが動く”感動
  - “目的に向かって進んでいる”実感
- ・ プログラムを変更(リファクタリング)する勇気
  - 影響がわかれれば直すのは怖くない
  - 仕様変更？オーケーやってやるよ！

# TDDを始めよう！

## ■とにかくやってみよう！

- ・ と、その前に：ペアを決めよう
  - ペアプログラミング：2人1組でプログラムを書く
  - これもXPのプラクティスの1つ
    - ・ パソコンの数が足りないからじゃないよ
- ・ 役割
  - ドライバー(実装する人):
    - ・ プログラムを書く人。目的の機能を完成させることに全神経を集中。
  - オブザーバー(実装しない人):
    - ・ ドライバーのコードを眺め、バグの指摘やコードの簡素化、大局的な問題を考える人。

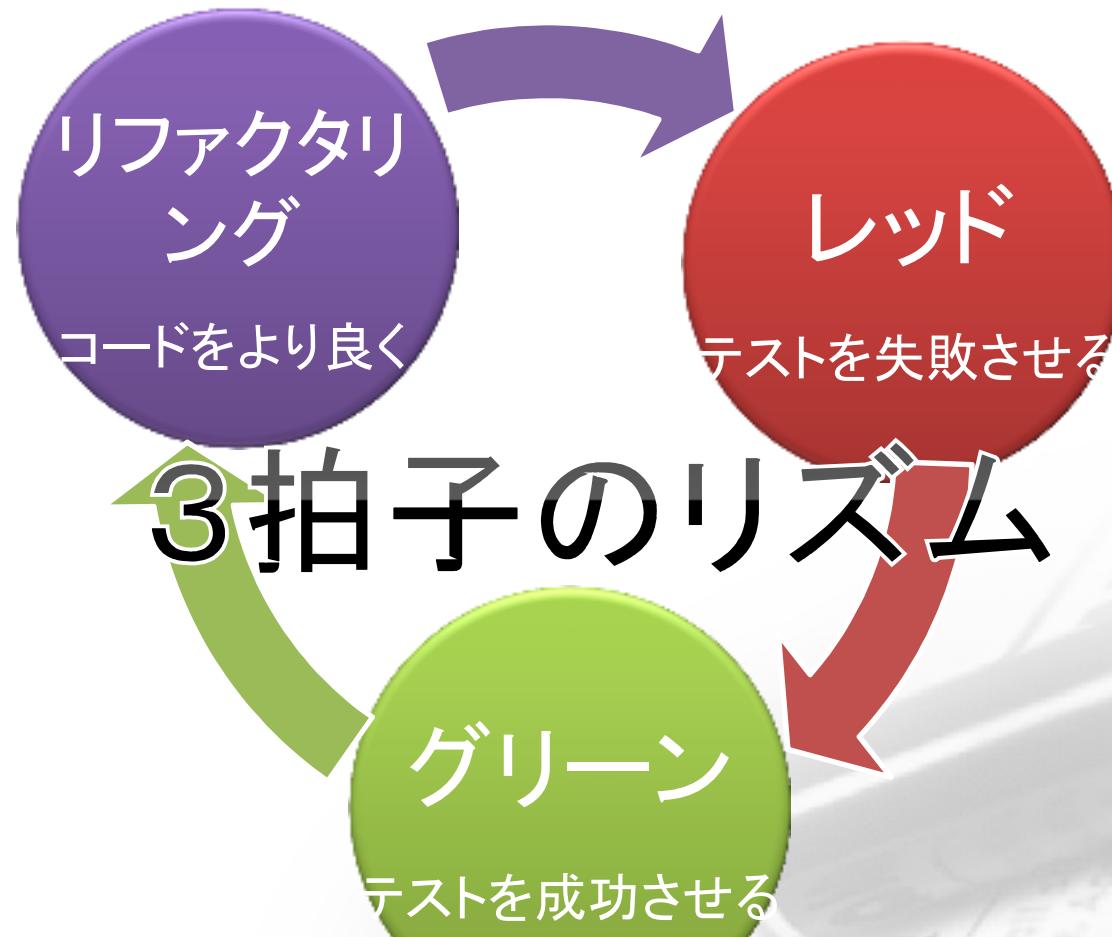
# TDDを始めよう！

## ■ペアプロの約束：3つだけ！

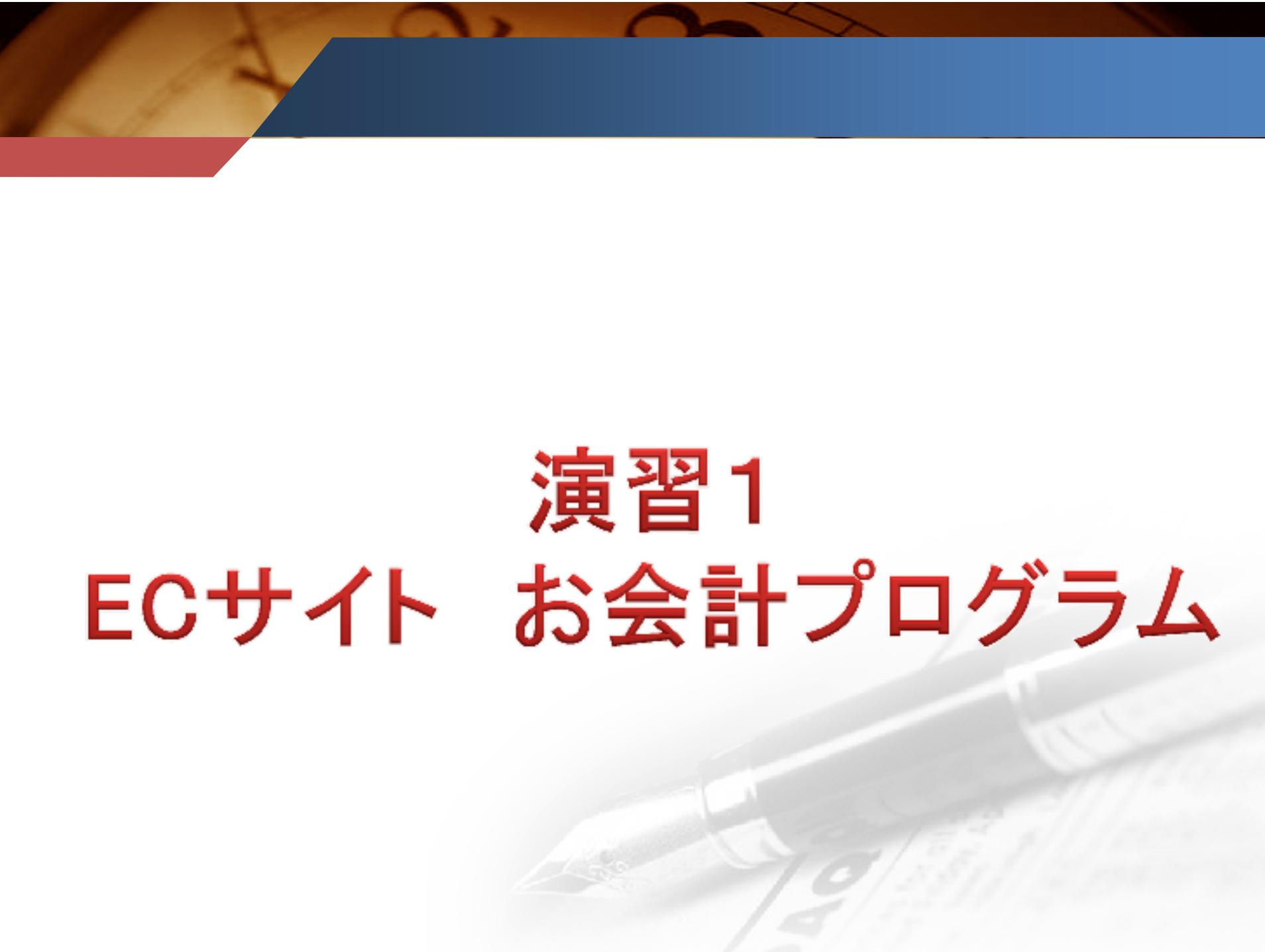
- ①ドライバーとオブザーバーは定期的に交代
  - 交代するタイミングはそれぞれのペアにお任せ
  - ただし最大30分以内
    - » 30分も集中すると疲れるよ
- ②オブザーバーの指摘
  - 読みづらいコードや文法ミスは行を書き終わった後に指摘
    - » 書き終えるまで声かけられるとウザいからね
  - その他大局的な問題などはメモだけしておき、交代時に話す
- ③喜びを共有する
  - テスト通過したらペアと一緒に喜ぶ
    - » ハイタッチとかオススメ、いやマジで

# TDDを始めよう！

## ■TDDの基本的な進め方



覚えるべきことはたった3つ。TDD怖くない！



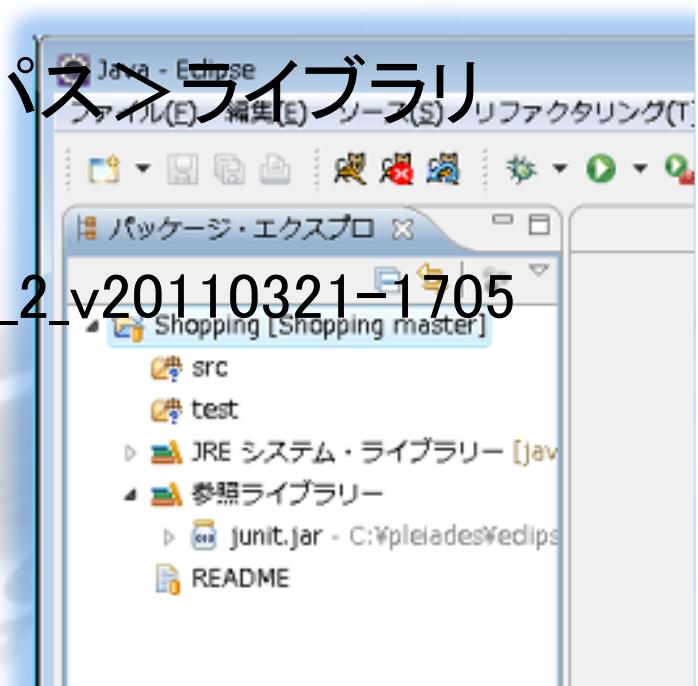
# **ECサイト 演習1 お会計プログラム**

# 演習1：会計プログラム

- ・簡単な例と一緒に進めよう
- ・【お題】ECサイトのお会計プログラム
  - 仕様
    - ・単価から商品の価格を求める
    - ・税込み価格を求める
- ・さあEclipseを起動しよう！

# 演習1：下準備

- ・ Shoppingプロジェクト作成
- ・ テスト用ソースディレクトリ作成
  - 新規>ソース・フォルダー
- ・ jUnitプラグイン設定
  - プロパティー>Javaのビルド・パス>ライブラリ  
外部JARの追加  
..../eclipse/plugins/org.junit\_4.8.2.v4\_8\_2\_v20110321-1705
- ・ ここまででこんな感じ→



# 演習1：下準備

- TODOリスト作成
  - 紙に書いても、メモ帳でもエクセルでも何でもいい
  - おすすめはChromeExt
    - “Google Tasks” →

準備OK!



# 演習1：Getting Start !

- TODOその1：単価から商品の価格を求める

何からやろうか？

# 演習1：Getting Start！

そう テストの作成だ

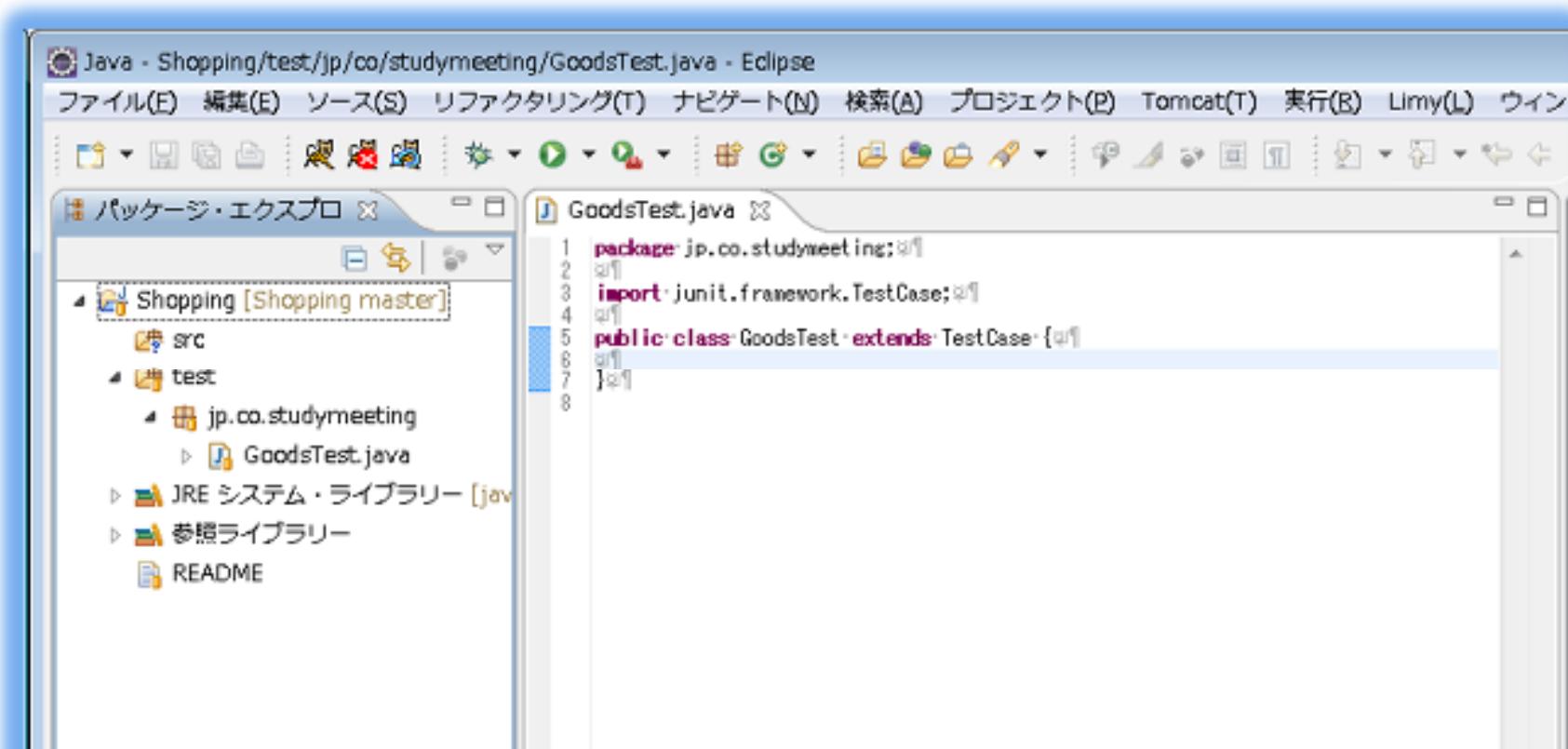
# 手順1 レッド: テストを失敗させる

- ・ 商品を扱うクラスだからGoodsクラスを作ろう
- ・ Junit.framework.TestCaseを継承した  
GoodsTestクラスをtestフォルダに作成
  - ・ テスト対象のクラス名 + Test の命名規約が一般的



# 手順1 レッド: テストを失敗させる

- ・ こんなかんじ



The screenshot shows the Eclipse IDE interface with the title bar "Java - Shopping/test/jp/co/studymeeting/GoodsTest.java - Eclipse". The menu bar includes "ファイル(F)", "編集(E)", "ソース(S)", "リファクタリング(T)", "ナビゲート(N)", "検索(A)", "プロジェクト(P)", "Tomcat(T)", "実行(B)", "Lijmy(L)", and "ウィン". Below the menu is a toolbar with various icons. On the left is the "パッケージ・エクスプローラー" (Package Explorer) showing a project structure under "Shopping [Shopping master]": "src", "test", "jp.co.studymeeting" (containing "GoodsTest.java"), "JRE システム・ライブラリー [jav]", "参照ライブラリー", and "README". The main editor window on the right displays the code for "GoodsTest.java":

```
1 package jp.co.studymeeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6
7 }
```



## 手順1 レッド: テストを失敗させる

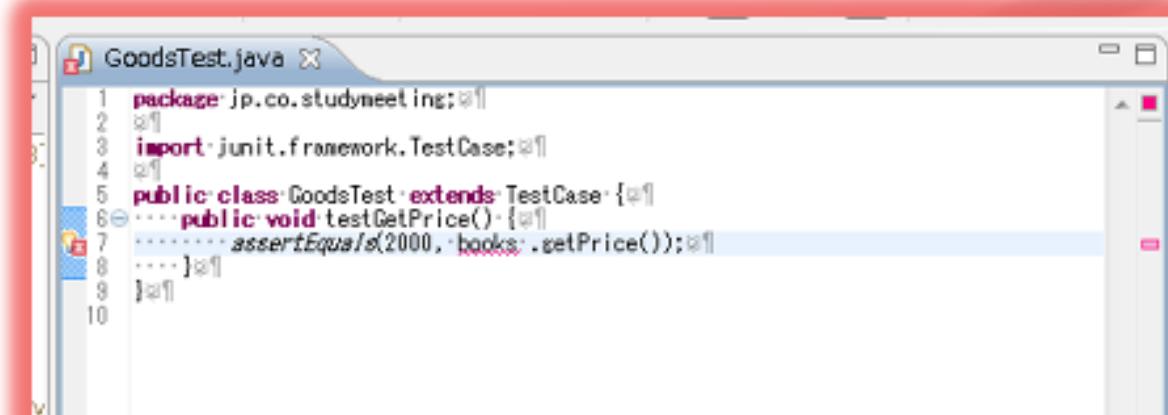
- ・ **テストファースト**: テストメソッドを先に実装
  - テストメソッドの実装
    - ・ **test**から始まるメソッドが自動でテスト対象として実行される
    - ・ 価格を取得するメソッドだから、`getPrice`とすると  
テストメソッドは`testGetPrice`
    - ・ 本の価格を求めてみよう
  - `TestCase`クラスの**assert**メソッドを利用して、値をチェックする。
    - ・ 基本:  
**assertEquals(期待値, 確認する値)**



## 手順1 レッド: テストを失敗させる

- ・ こんなかんじ
- ・ コンパイルエラーがでてるんですけど…

大丈夫だ、問題ない

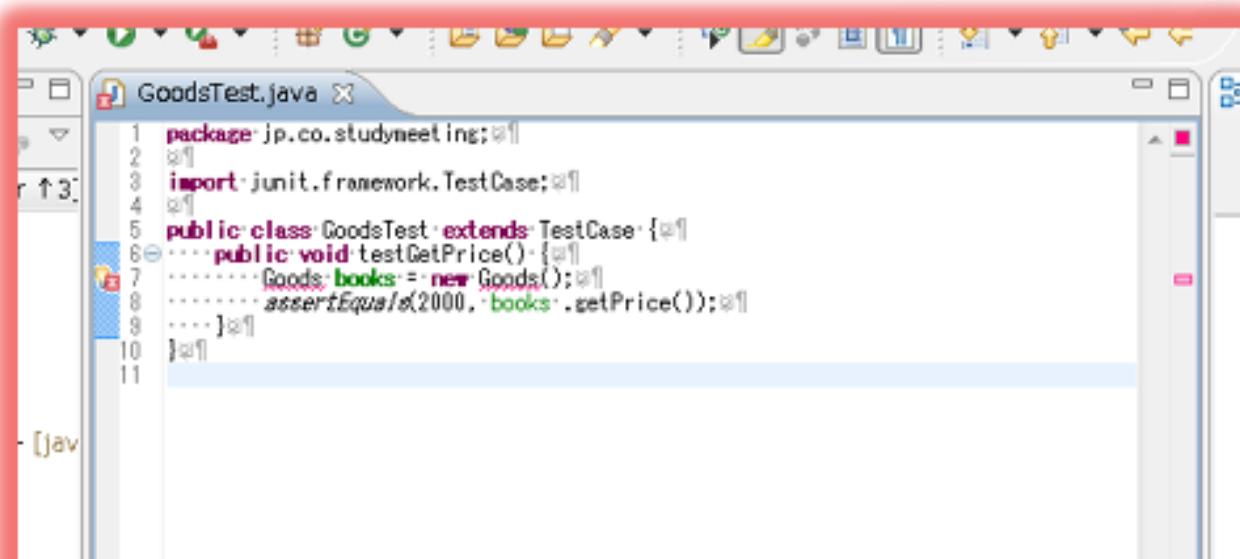


```
GoodsTest.java
1 package jp.co.studyneeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     ...
7     public void testGetPrice() {
8         assertEquals(2000, books.getPrice());
9     }
10}
```

A screenshot of a Java code editor window titled "GoodsTest.java". The code defines a class "GoodsTest" that extends "TestCase". It contains a single test method "testGetPrice" which uses the "assertEquals" assertion to check if the price of a book is 2000. The code editor has a red border around the window.

# 手順1 レッド: テストを失敗させる

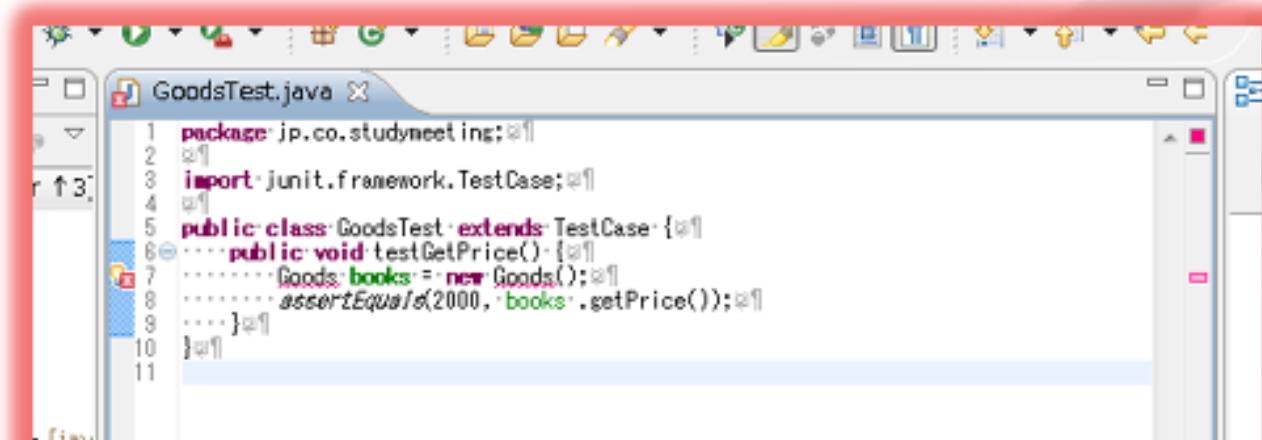
- ・ 変数宣言していないから当然だ
  - 焦らず騒がずインスタンスを生成
  - 商品を扱うクラスだからGoodsクラスと命名
  - Goodsクラスをインスタンス化しよう



```
GoodsTest.java
1 package jp.co.studymeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     public void testGetPrice() {
7         Goods books = new Goods();
8         assertEquals(2000, books.getPrice());
9     }
10 }
11
```

# 手順1 レッド: テストを失敗させる

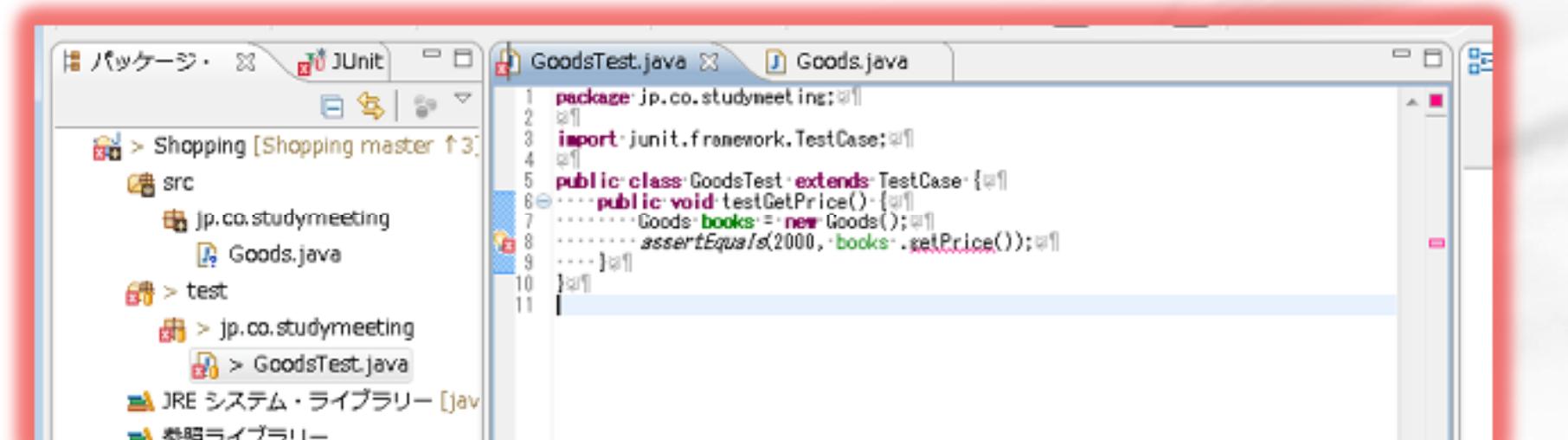
- ・ まだコンパイルエラー…
  - Goodsクラスがない
- ・ Goodsクラスを作ろう
  - クイックフィックスでクラスを作成
  - ソースフォルダをsrcにするのを忘れずに



```
GoodsTest.java
1 package jp.co.studymeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     public void testGetPrice() {
7         Goods books = new Goods();
8         assertEquals(2000, books.getPrice());
9     }
10 }
11
```

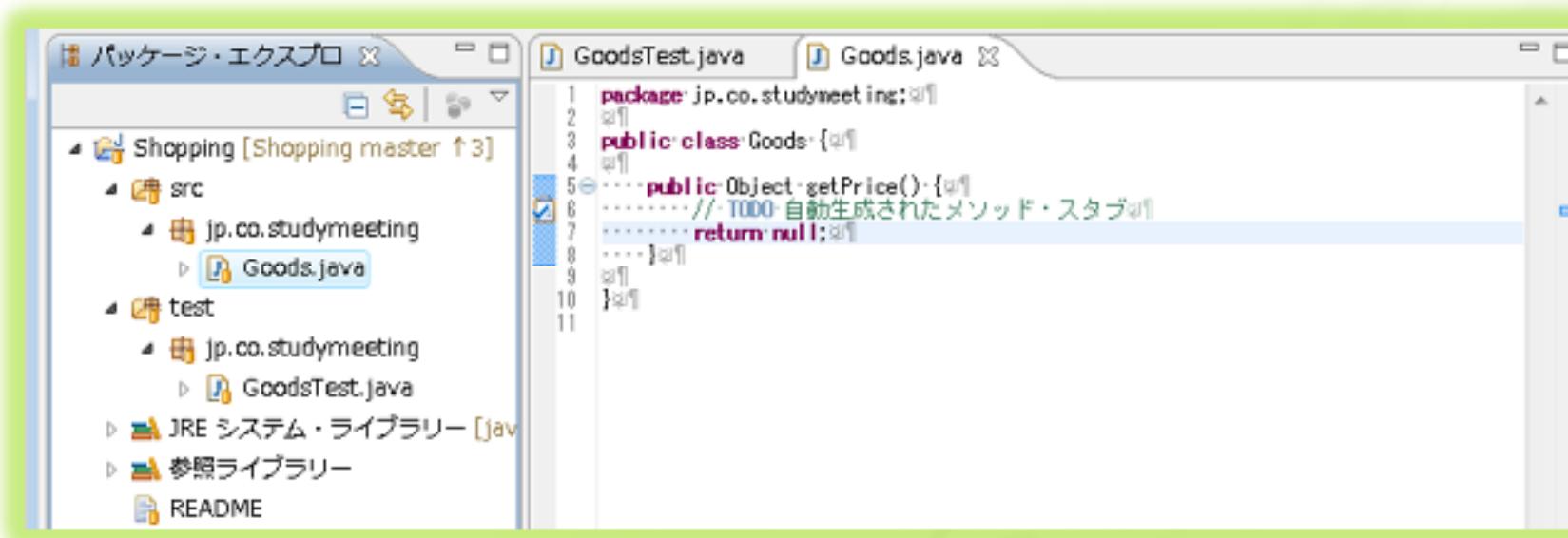
# 手順1 レッド: テストを失敗させる

- まだコンパイルエラー…
  - GoodsクラスにgetPriceメソッドがない
  - またまたクイックフィックスで作ってもらう
    - Eclipseさんあざーっす



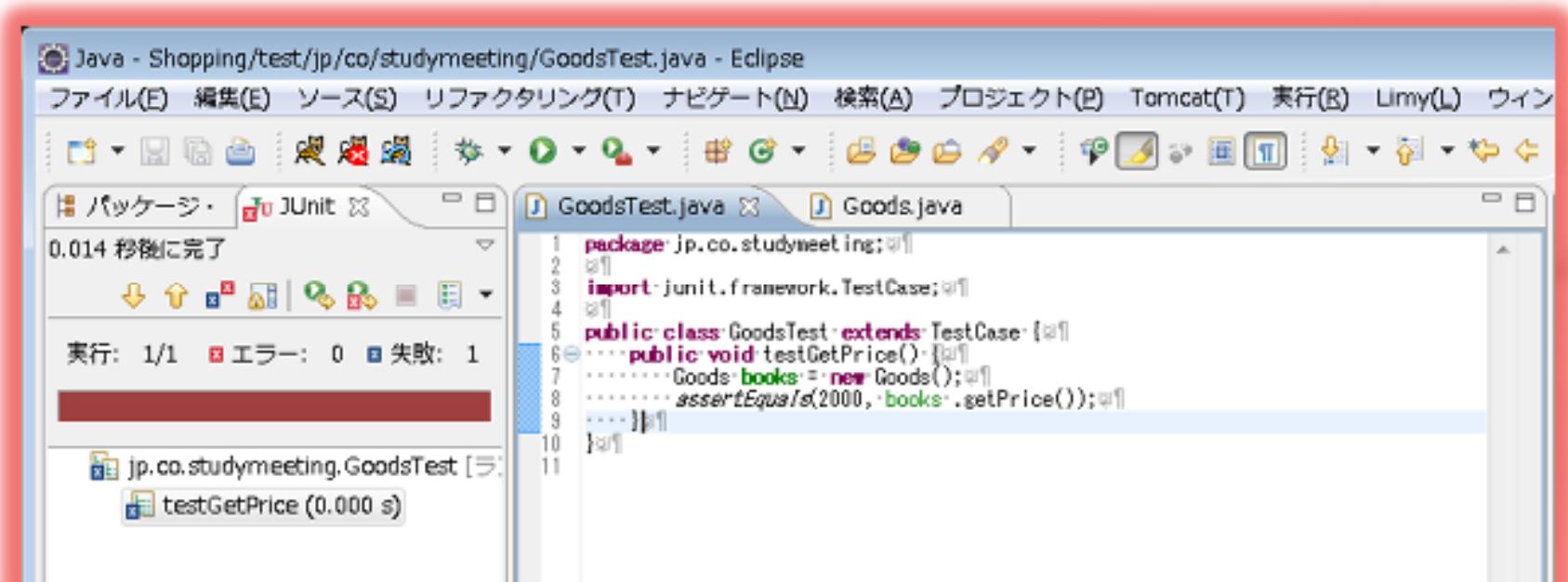
# 手順1 レッド: テストを失敗させる

- ・ コンパイルが通った！
- ・ ペアとハイタッチだ！



# 手順1 レッド: テストを失敗させる

- ・ テストの実行
  - ショートカットを覚えよう。 **Ctr+F11**だ。
- ・ レッドのバーが！ テストが失敗した！
- ・ junit.framework.AssertionFailedError: expected:<2000> but was:<null>  
～2000が期待値だったけどnullがきたぜ～





## 手順1 レッド: テストを失敗させる

おめでとう！  
手順1クリアだ！

ペアとハイタッチだ！

## 【コラム】 1. YAGNIの原則

- ・ なんか面倒くさくね…?

### ■ YAGNI(ヤグニ)の原則

- ・ You Ain't Gonna Need It!
  - “今そんなの必要ないって！”の意
  - 「今」必要なものだけ実装しよう！
  - コンパイルを通過すこと、テストを通過することを一度にやらない！
  - テストから先に作成し、目的に向かって最短の経路で実装する！

## 手順2 グリーン: テストを成功させる

- ・ 最短でテストをパスする実装は?
  - getPrice() で return 2000;
- ・ テストを再実施してみる
  - Ctr+F11



```
1 package jp.co.studyneeting; 2 3 public class Goods { 4 5     public int getPrice() { 6         return 2000; 7     } 8 } 9 10
```

## 手順2 グリーン: テストを成功させる

- グリーンのバーだ！ テストが通った！

The screenshot shows an IDE interface with two windows. The left window is titled 'JUnit' and displays a green progress bar at the bottom with the text '0.031 秒後に完了' (Completed in 0.031 seconds) above it. Below the bar, the status is shown as '実行: 1/1' (Run: 1/1), 'エラー: 0' (Errors: 0), and '失敗: 0' (Failures: 0). The right window is titled 'GoodsTest.java' and contains the following Java code:

```
1 package jp.co.studymeeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     public void testGetPrice() {
7         Goods books = new Goods();
8         assertEquals(2000, books.getPrice());
9     }
10 }
11
```



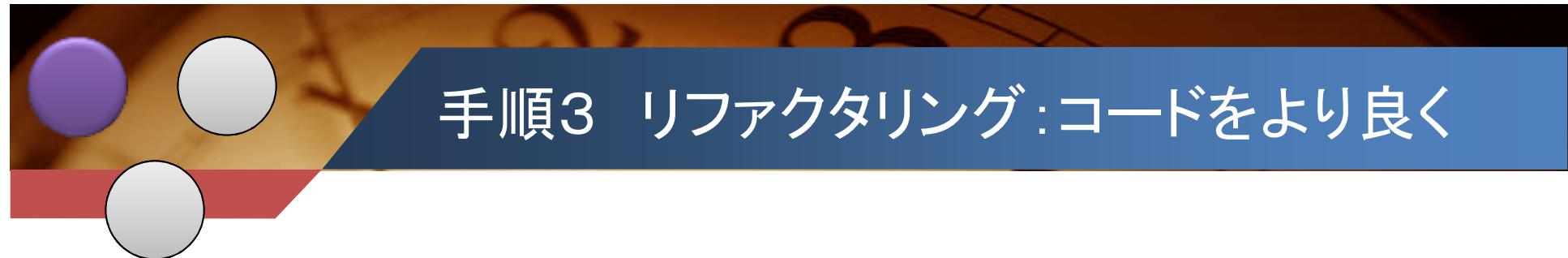
手順2 グリーン: テストを成功させる

おめでとう！  
手順2クリアだ！

ペアとハイタッチだ！

## ■ 仮実装

- ・ このようにテストをクリアするための最小の実装を**仮実装 (Fake It)**という
  - ビジネスロジックが複雑な時や、初めてでよくわからない技術を使う時に有効
  - 自信が持てるコード(テストにより影響が確認できるコード)を少しずつ増やしていくことがミソ



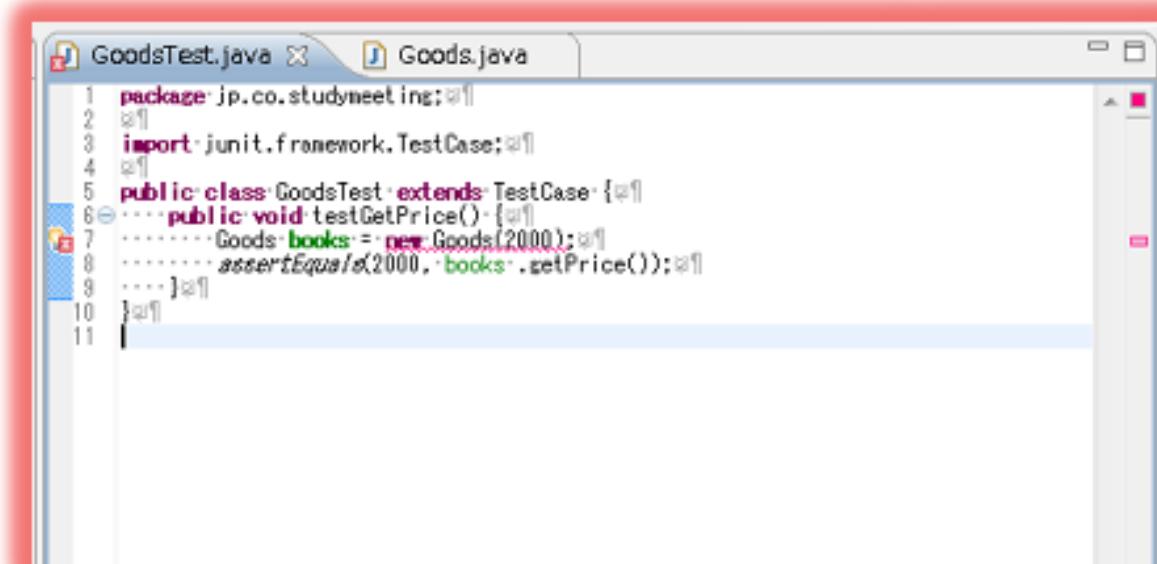
### 手順3 リファクタリング：コードをより良く

- ・ このままでは、単価2000円の商品しか扱えない…

**リファクタリングだ！**

## 手順3 リファクタリング：コードをより良く

- ・ 固定値になっている箇所を変数で扱うには?
  - コンストラクタで単価をもらうよう修正
- ・ コンパイルエラーから、クイックフィックス
  - リファクタリングもテストが先だ！



The screenshot shows an IDE interface with two tabs open:

- GoodsTest.java**:

```
1 package jp.co.studyneeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     public void testGetPrice() {
7         Goods books = new Goods(2000);
8         assertEquals(2000, books.getPrice());
9     }
10 }
```
- Goods.java**:

```
1 package jp.co.studyneeting;
2
3 public class Goods {
4     private int price;
5
6     public Goods(int price) {
7         this.price = price;
8     }
9
10     public int getPrice() {
11         return price;
12     }
13 }
```

The code in `GoodsTest.java` is intended to test the `getPrice()` method of the `Goods` class. The variable `books` is initialized with a new `Goods` object having a price of 2000. The `assertEquals` method is used to verify that the `getPrice()` method returns 2000. The `Goods` class has a private attribute `price` and a constructor that takes an integer parameter. It also has a `getPrice()` method that returns the value of `price`.

## 手順3 リファクタリング：コードをより良く

- ・ Goodsクラスも修正
  - 自動生成したコンストラクタの引数名を修正(いだとなんのことかわからない)
  - 単価をプライベートのフィールド変数に保持
  - getPriceはフィールド変数を返すよう修正(Getter)
- ・ 再テストだ！

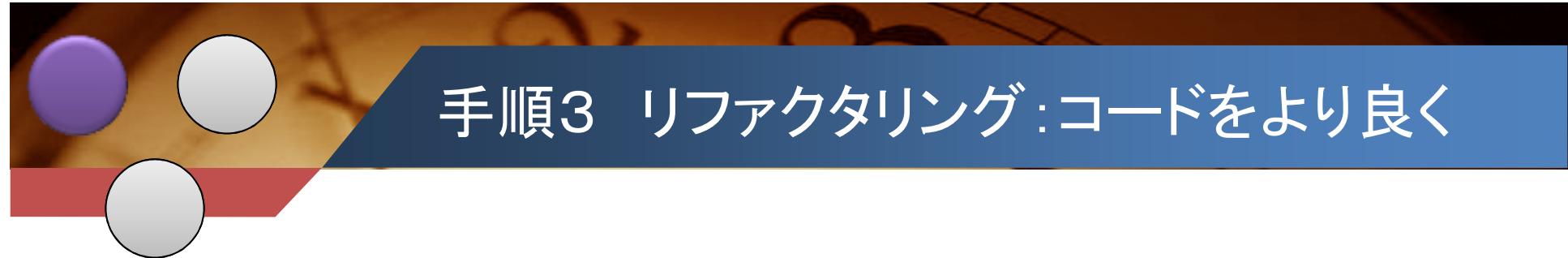
```
GoodsTest.java  Goods.java X
1 package jp.co.studymeeting;
2
3 public class Goods {
4     ...
5     private int price;
6     ...
7     public Goods( int price ) {
8         this.price = price;
9     }
10    ...
11    public int getPrice() {
12        return this.price;
13    }
14    ...
15 }
16
```

## 手順3 リファクタリング：コードをより良く

- ・ テストが通った！

The screenshot shows an IDE interface with two main windows. On the left is the JUnit runner window, which displays the message "0.009 秒後に完了" (Completed in 0.009 seconds) and shows a green progress bar indicating 1/1 test passed. Below the progress bar, it says "実行: 1/1 エラー: 0 失敗: 0". On the right is the code editor window titled "GoodsTest.java", containing the following Java code:

```
1 package jp.co.studymeeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6     public void testGetPrice() {
7         Goods books = new Goods(2000);
8         assertEquals(2000, books.getPrice());
9     }
10 }
11
```



## 手順3 リファクタリング：コードをより良く

おめでとう！  
手順3クリアだ！

ペアとハイタッチだ！

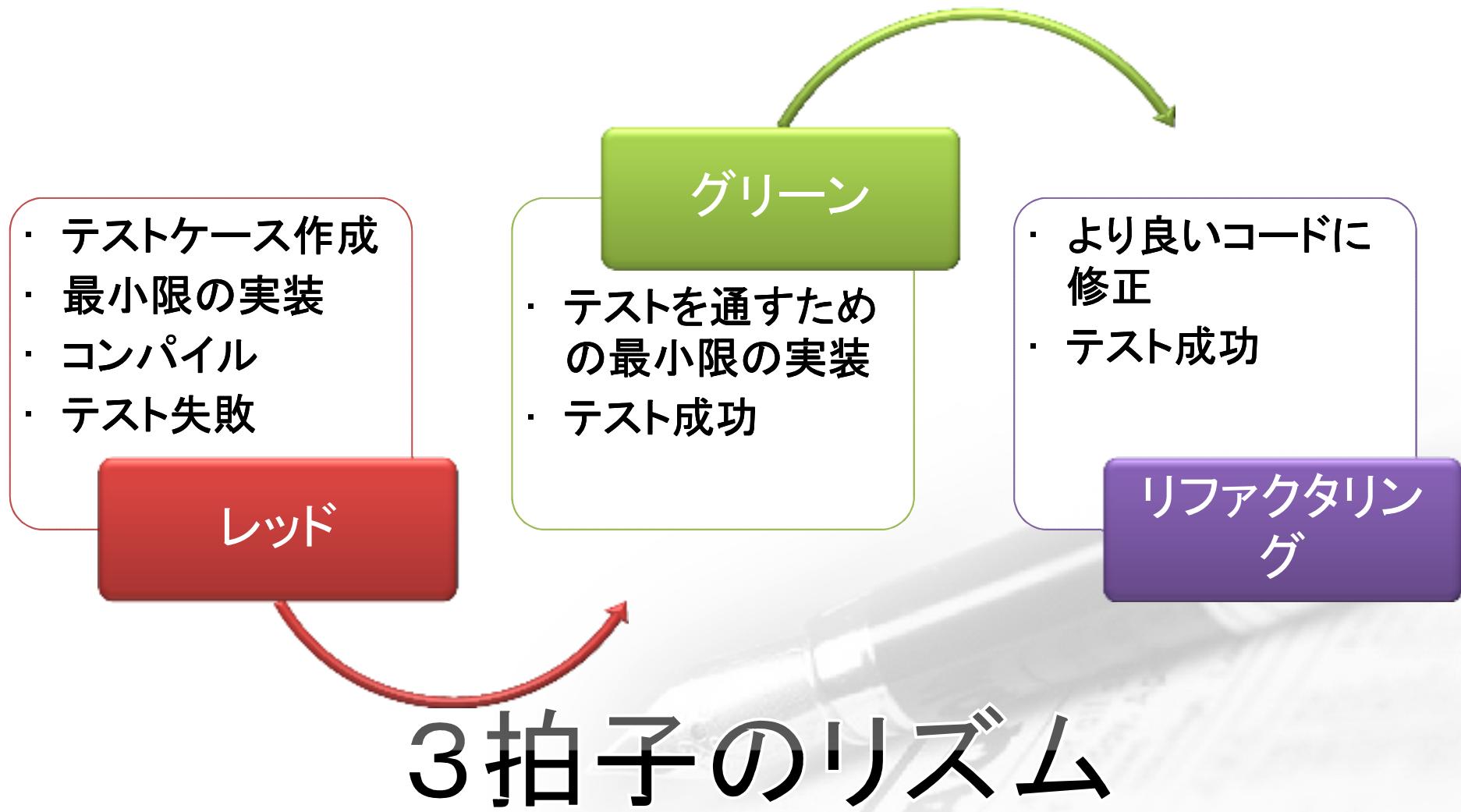


# そして、ようこそ モダンコードの世界へ！

今書いた30行にも満たない簡単なコードは、  
あなたが今まで書いたどんなコードよりも  
モダンなコードだ！  
テストがあるからね！

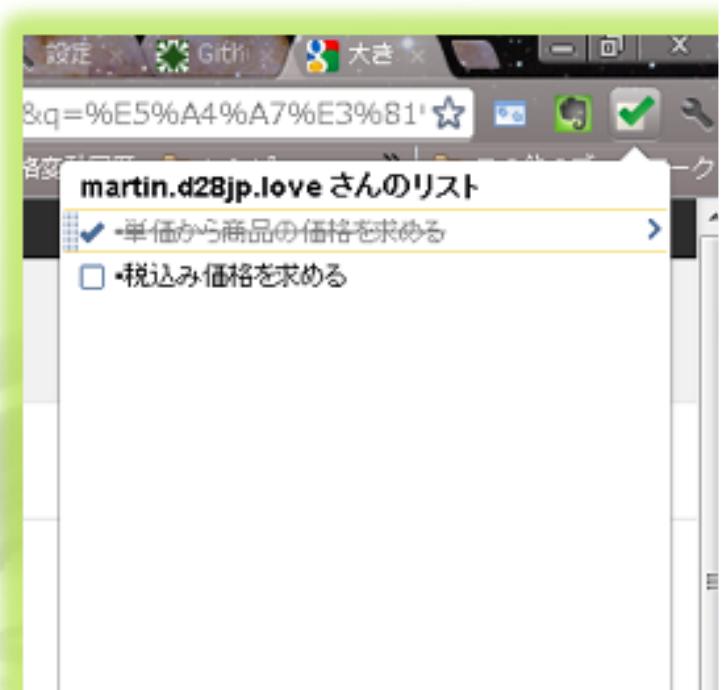
## 【コラム】 3. TDDの基本の流れ

# これがTDDの基本の流れだ！



# 演習1: TODO2 Start !

- ・ 1つめのTODOは消えた
- ・ TODOの2番目、税込み価格を求める
- ・ ちょっとペースあげていくよ！



# 手順1 レッド: テストを失敗させる

- ・ 税込み価格を求めるテストを追加する
- ・ テストの内容がわかるよう、assertメソッドにメッセージ引数を追加する
  - これがないと後で見たときに何のテストかわからなくなる…
  - メソッド自動生成してコンパイルエラーをとり、実行！

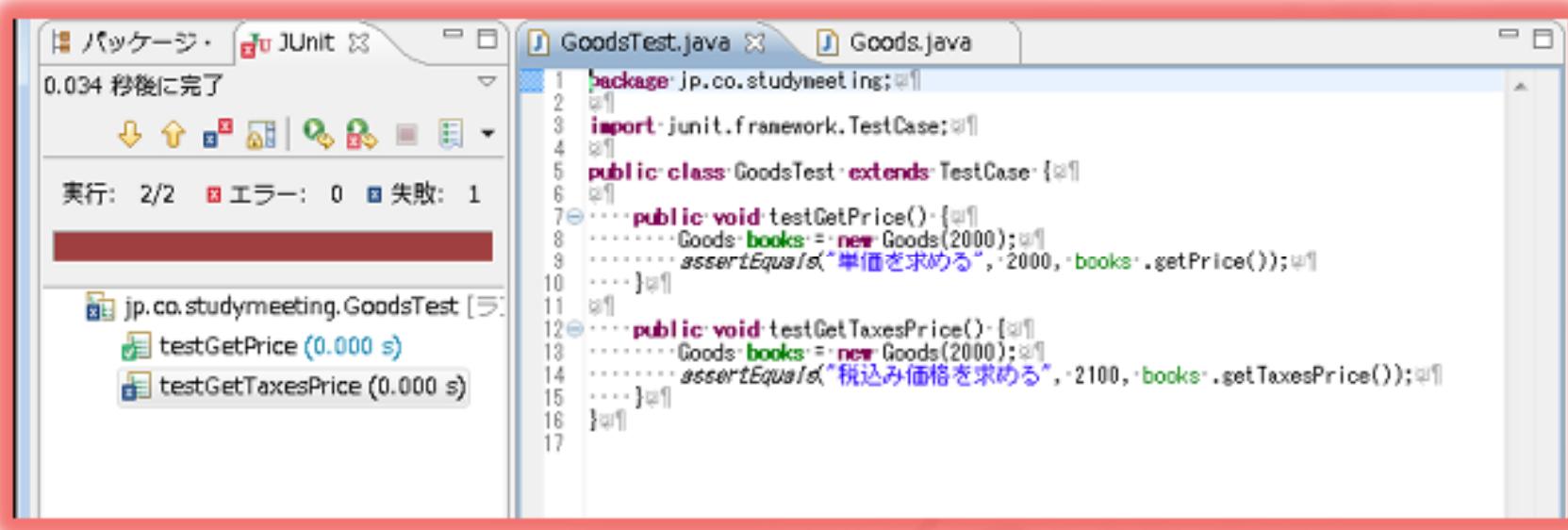


The screenshot shows an IDE window with two tabs: 'GoodsTest.java' and 'Goods.java'. The 'GoodsTest.java' tab is active, displaying the following code:

```
1 package jp.co.studymeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6
7     public void testGetPrice() {
8         Goods books = new Goods(2000);
9         assertEquals("単価を求める", 2000, books.getPrice());
10    }
11
12    public void testGetTaxesPrice() {
13        Goods books = new Goods(2000);
14        assertEquals("税込み価格を求める", 2100, books.getTaxesPrice());
15    }
16 }
17
```

# 手順1 レッド: テストを失敗させる

- ・ テストが失敗した！
- ・ junit.framework.AssertionFailedError: 税込み価格を求める  
expected:<2100> but was:<null>
  - ↑ エラーメッセージが親切になった
- ・ OK仮実装だ！



The screenshot shows an IDE interface with two panes. The left pane is titled 'パッケージ' (Package) and contains a 'JUnit' tab. It displays the test results: '0.034 秒後に完了' (Completed in 0.034 seconds), '実行: 2/2' (Run: 2/2), 'エラー: 0' (Errors: 0), and '失敗: 1' (Failures: 1). Below this, it lists the test classes: 'jp.co.studymeting.GoodsTest [ラ]' with methods 'testGetPrice (0.000 s)' and 'testGetTaxesPrice (0.000 s)'. The right pane is titled 'GoodsTest.java' and shows the Java source code:

```
1 package jp.co.studymeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6
7     public void testGetPrice() {
8         Goods books = new Goods(2000);
9         assertEquals("単価を求める", 2000, books.getPrice());
10    }
11
12    public void testGetTaxesPrice() {
13        Goods books = new Goods(2000);
14        assertEquals("税込み価格を求める", 2100, books.getTaxesPrice());
15    }
16}
```

## 手順2 グリーン: テストを成功させる

- ・ 最短でテストをクリアする実装
  - return price + 100;
- ・ テストが通った！
- ・ OK仮実装をリファクタリングだ！

The screenshot shows an IDE interface with two main windows. On the left is the JUnit Test View window, which displays the test results: "0.009 秒後に完了" (Completed in 0.009 seconds), "実行: 2/2" (Executed: 2/2), "エラー: 0" (Errors: 0), and "失敗: 0" (Failures: 0). On the right is the code editor window titled "GoodsTest.java" and "Goods.java". The code in "GoodsTest.java" is as follows:

```
1 package jp.co.studymeeting;
2
3 import Shopping/test/jp/co/studymeeting/GoodsTest.java
4
5 public class GoodsTest extends TestCase {
6
7     public void testGetPrice() {
8         Goods books = new Goods(2000);
9         assertEquals("単価を求める", 2000, books.getPrice());
10    }
11
12    public void testGetTaxesPrice() {
13        Goods books = new Goods(2000);
14        assertEquals("税込み価格を求める", 2100, books.getTaxesPrice());
15    }
16
17 }
```

The code in "Goods.java" is as follows:

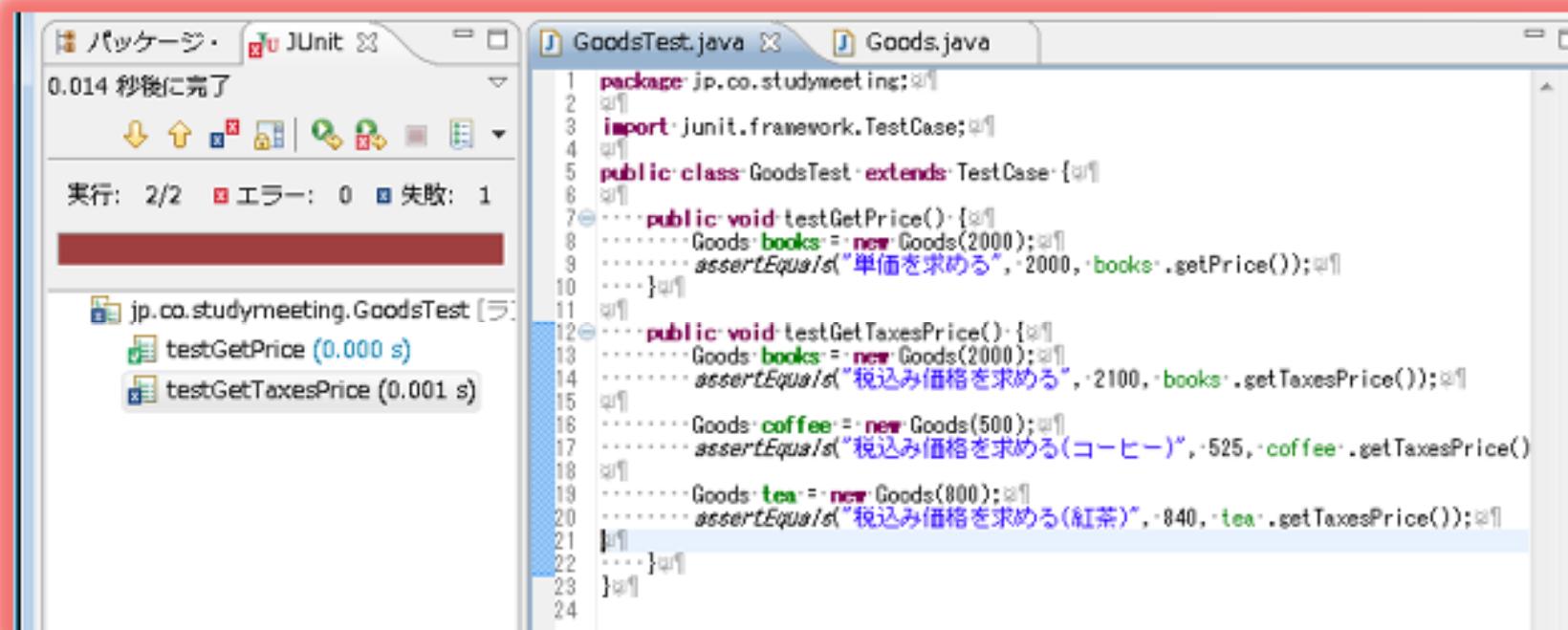
```
1 package jp.co.studymeeting;
2
3 public class Goods {
4     private int price;
5
6     public Goods(int price) {
7         this.price = price;
8     }
9
10    public int getPrice() {
11        return price;
12    }
13
14    public int getTaxesPrice() {
15        return price * 1.08;
16    }
17 }
```

## ■ 三角測量

- ・ 税込み価格を固定で求めてしまっては使い物にならない
  - 正しい解であることを確認するには、複数の値で検証する  
→テストケースを追加する！
- ・ このように複数の値の検証から正しい解を求めるやり方を三角測量(Triangulation)という

## 手順3 リファクタリング：コードをより良く

- ・ テストケースを追加した
- ・ テスト失敗だ！

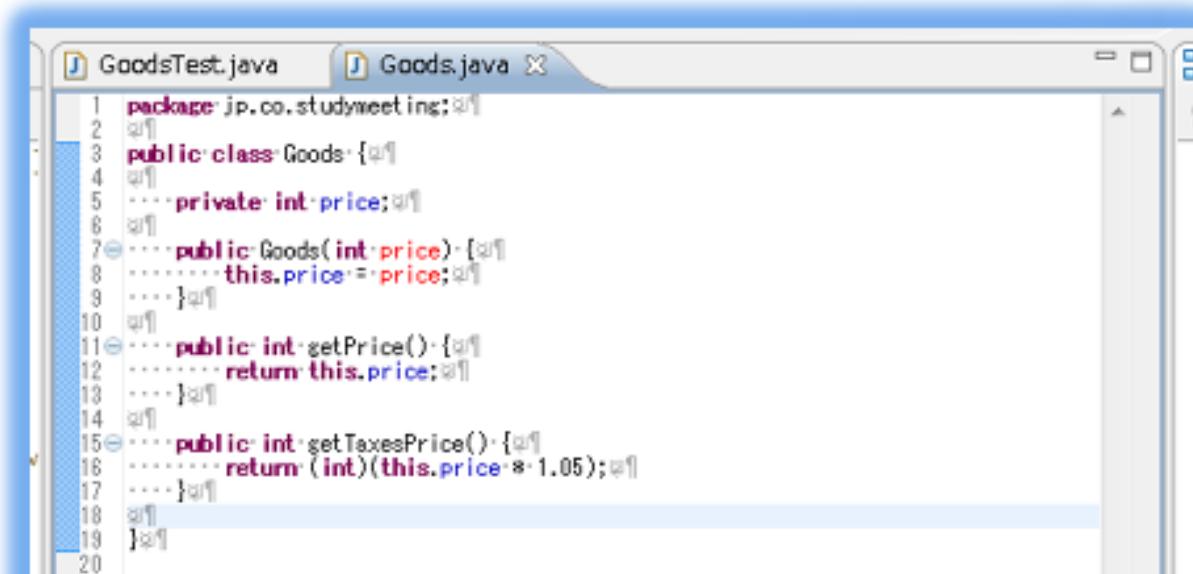


The screenshot shows an IDE interface with two main panes. The left pane is titled 'パッケージ' (Package) and contains a 'JUnit' tab. It displays the message '0.014 秒後に完了' (Completed in 0.014 seconds). Below this, it shows '実行: 2/2 エラー: 0 失敗: 1'. The right pane is titled 'GoodsTest.java' and shows the following Java code:

```
1 package jp.co.studymeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6
7     public void testGetPrice() {
8         Goods books = new Goods(2000);
9         assertEquals("単価を求める", 2000, books.getPrice());
10    }
11
12    public void testGetTaxesPrice() {
13        Goods books = new Goods(2000);
14        assertEquals("税込み価格を求める", 2100, books.getTaxesPrice());
15
16        Goods coffee = new Goods(500);
17        assertEquals("税込み価格を求める(コーヒー)", 525, coffee.getTaxesPrice());
18
19        Goods tea = new Goods(800);
20        assertEquals("税込み価格を求める(紅茶)", 840, tea.getTaxesPrice());
21    }
22}
23
24}
```

## 手順3 リファクタリング：コードをより良く

- ・ テストケースをクリアできる実装
  - `return (int)(this.price * 1.05);`
    - ・ 小数を掛けると型がdoubleになってしまってるので、返り値の型と一致しない。intにキャストする



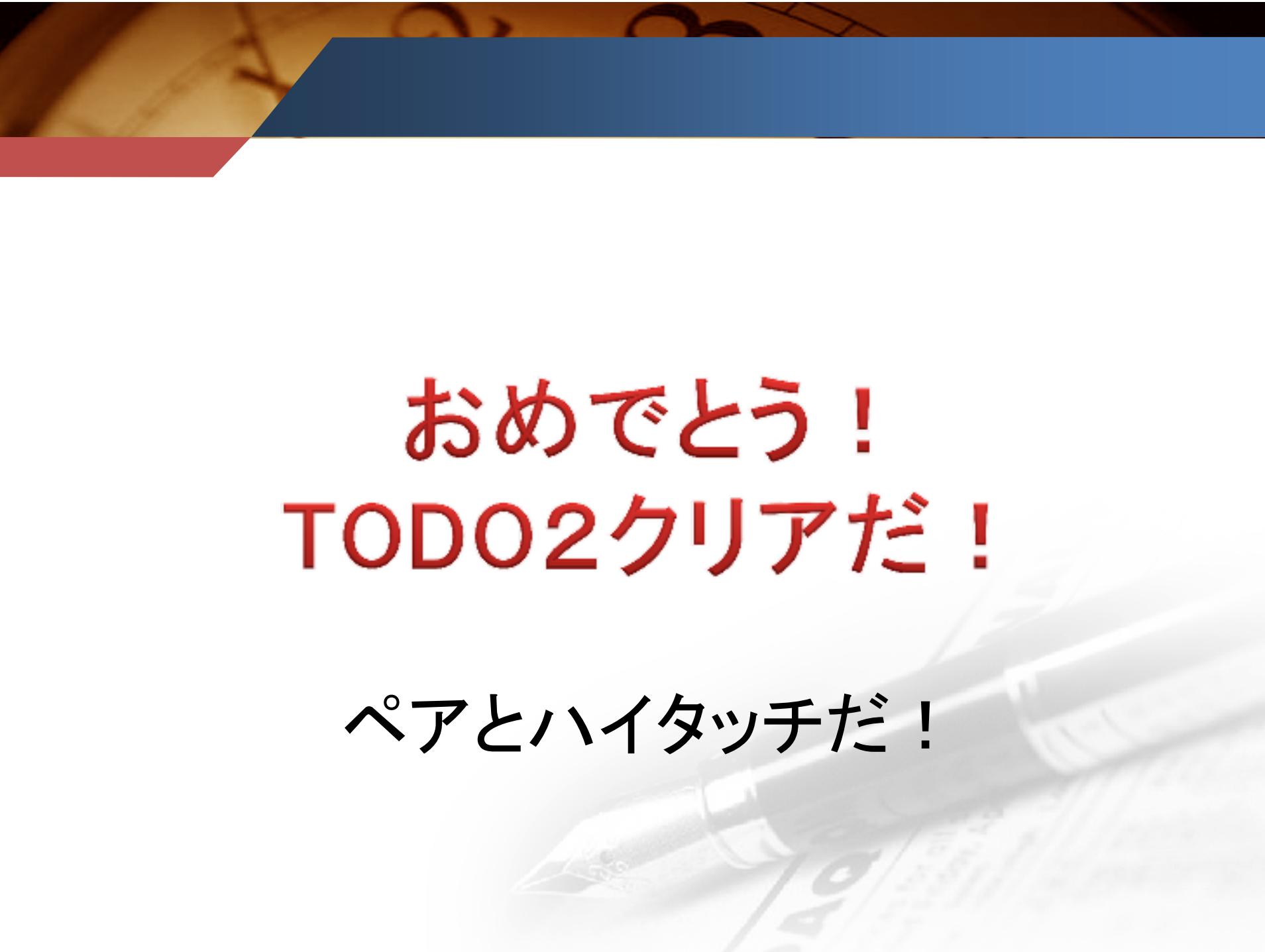
```
GoodsTest.java  Goods.java X
1 package jp.co.studymeting;
2
3 public class Goods {
4     private int price;
5
6     public Goods(int price) {
7         this.price = price;
8     }
9
10    public int getPrice() {
11        return this.price;
12    }
13
14    public int getTaxesPrice() {
15        return (int)(this.price * 1.05);
16    }
17
18 }
19
20 }
```

## 手順3 リファクタリング：コードをより良く

- ・ テスト再実行(**Ctr+F11**)
- ・ テストが通った！

The screenshot shows an IDE interface with two main panes. The left pane is a JUnit test results window titled 'パッケージ' (Package) with a green border. It displays the message '0.144 秒後に完了' (Completed in 0.144 seconds), '実行: 2/2 エラー: 0 失敗: 0' (Executed: 2/2 Errors: 0 Failures: 0), and a green progress bar. The right pane is a code editor titled 'GoodsTest.java' with a blue border. It contains Java code for testing the 'Goods' class. The code includes imports for 'package' and 'junit.framework.TestCase'. It defines a 'GoodsTest' class that extends 'TestCase'. The class has two test methods: 'testGetPrice()' and 'testGetTaxesPrice()'. Both tests create a new 'Goods' object with specific prices (2000 for books, 500 for coffee, 800 for tea) and assert that the calculated prices (2100 for books, 525 for coffee, 840 for tea) are equal to the expected values ('単価を求める', '税込み価格を求める', and '税込み価格を求める(紅茶)').

```
1 package jp.co.studymeeting;
2
3 import junit.framework.TestCase;
4
5 public class GoodsTest extends TestCase {
6
7     public void testGetPrice() {
8         Goods books = new Goods(2000);
9         assertEquals("単価を求める", 2000, books.getPrice());
10    }
11
12    public void testGetTaxesPrice() {
13        Goods books = new Goods(2000);
14        assertEquals("税込み価格を求める", 2100, books.getTaxesPrice());
15
16        Goods coffee = new Goods(500);
17        assertEquals("税込み価格を求める(コーヒー)", 525, coffee.getTaxesPrice());
18
19        Goods tea = new Goods(800);
20        assertEquals("税込み価格を求める(紅茶)", 840, tea.getTaxesPrice());
21    }
22 }
23
24 }
```



おめでとう！  
TODO2クリアだ！

ペアとハイタッチだ！

## 【コラム】 5. 明白な実装

- ・ やっぱりなんか面倒くさくね？

### ■ 明白な実装

- ・ 今回のように実装例が簡単な場合は、仮実装→三角測量の手順を踏まなくてもOK
- ・ テストコード→テスト失敗→テストをパスする実装→テスト成功の手順
- ・ コードに自信がないとき仮実装や三角測量が非常に有効！



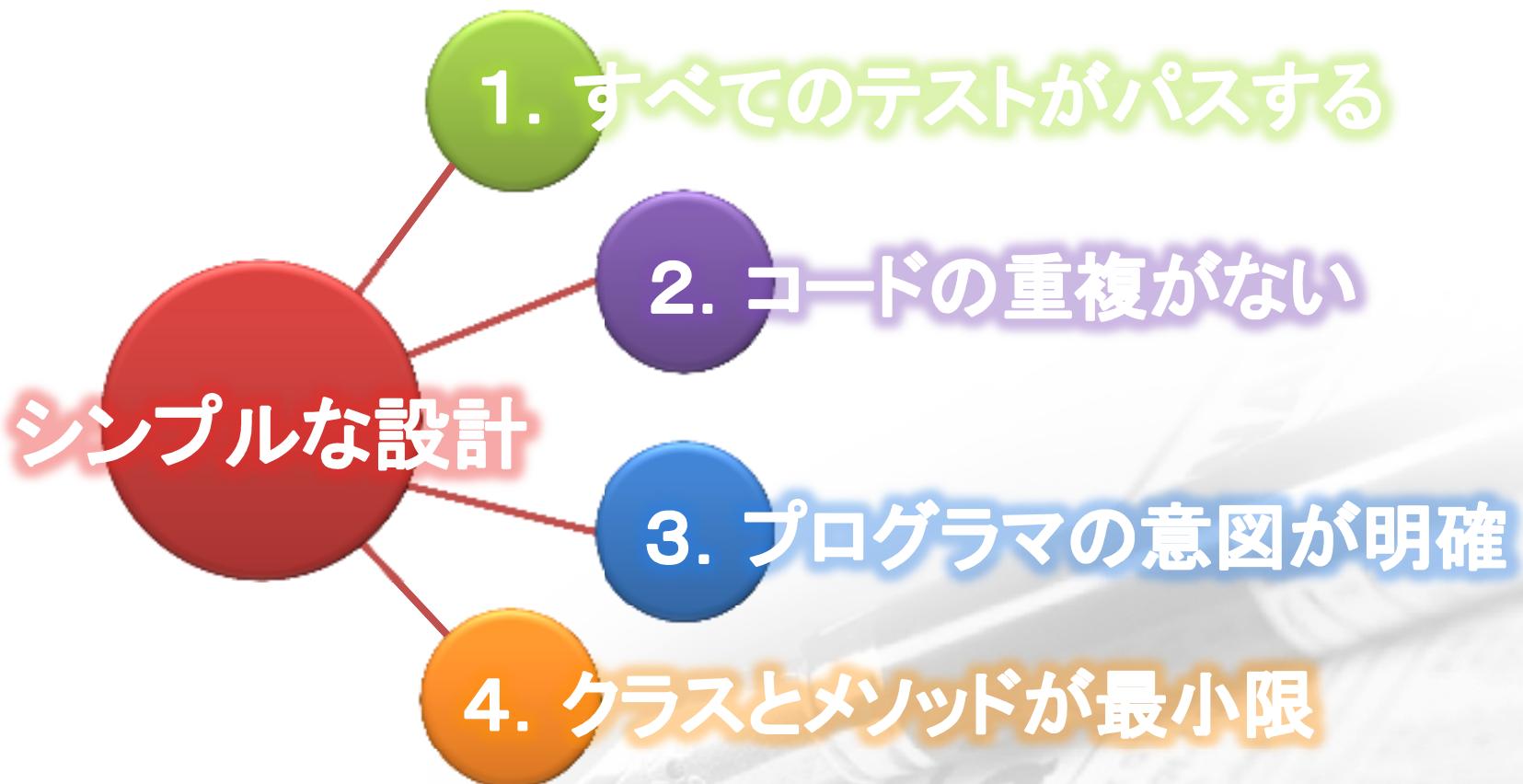
## 【コラム】 6. より良いコードとは？

- ・ リファクタリングで、コードが**より良くなっている**ことが実感できるな！
- ・ ……ちょっと待って。そもそも  
**より良いコード**  
ってなんだよ！  
そんなの知ってたら苦労しないよ！

## 【コラム】 6. より良いコードとは？

TDDで言う“より良いコード”とは  
“シンプルな設計”  
を満たすコードだ！

## ■これが“シンプルな設計”だ！



→1～3を満たした上で、4がある！



# 演習 1

# おしまい



お昼休みに  
しましょうか！



# **演習2**

# **DB de TDD !**

# 演習2: DB de TDD !

- DB…？
  - ドラゴンボールだろJK
- この辺が元ネタ
  - <http://anond.hatelabo.jp/20110316202255>
  - <http://d.hatena.ne.jp/ryoasai/20110317>
- 前提条件
  - 登場人物(抜粋) : 悟空、クリリン、ピッコロ
  - 3人は“Z戦士”と呼ばれる仲間(そういえばZって何?)

## 演習2: DB de TDD !

- ・仕様(TODOリスト)
  - Z戦士は皆“戦闘力”という共通の強さを測る値を持つ
  - ピッコロはクリリンより強い(=戦闘力が高い)
  - 悟空はピッコロよりスゲー強い
  - Z戦士は皆“舞空術”という技で空を飛べる。
  - “舞空術”で移動できる速さは3人で異なる。
    - ・クリリンは“普通に”飛ぶ。
    - ・ピッコロは“ビュンと”飛ぶ。
    - ・悟空は”目にもとまらぬ速さで”飛ぶ。

## 演習2: DB de TDD !

TDDで  
実装してみよう！

## 演習2: DB de TDD !

何をやつたらいいか  
わからないって？

## 演習2: DB de TDD !

プログラムに「正解」はない。  
どう書いたっていいんだ。

あるのは「より良いコードを書きたい」  
というプログラマの気持ちだけだ！

とにかくやってみよう！！

## 演習2: DB de TDD !

- ・仕様(TODOリスト)
  - Z戦士は皆“戦闘力”という共通の強さを測る値を持つ
  - ピッコロはクリリンより強い(=戦闘力が高い)
  - 悟空はピッコロよりスゲー強い
  - Z戦士は皆“舞空術”という技で空を飛べる。
  - “舞空術”で移動できる速さは3人で異なる。
    - ・クリリンは“普通に”飛ぶ。
    - ・ピッコロは“ビュンと”飛ぶ。
    - ・悟空は”目にもとまらぬ速さで”飛ぶ。

NOW PROGRAMMING・・・

# 演習2: 【サンプル実装】

## ・ポイント1 戦闘力の差

- “Z戦士は共通で戦闘力という数値を持つ”
  - Z戦士クラスを作つて、戦士をオブジェクト化するか…
- オブジェクト(戦士)により、戦闘力が異なる
  - さつきの会計PGと同じパターンだー余裕だぜ！

The screenshot shows a Java development environment with two code editors side-by-side.

**ZSenshiTest.java:**

```
1 package jp.co.tdd.dragonball;
2
3 import junit.framework.TestCase;
4
5 public class ZSenshiTest extends TestCase {
6
7     public void testSentoryoku() {
8
9         ZSenshi goku = new ZSenshi(ZSenshi.GOKU);
10        ZSenshi pikkoro = new ZSenshi(ZSenshi.PIKKORO);
11        ZSenshi kuririn = new ZSenshi(ZSenshi.KURIRIN);
12
13        assertEquals("2号は戦闘力を持つ", goku.getSentoryoku());
14        assertEquals("2号は戦闘力を持つ", pikkoro.getSentoryoku());
15        assertEquals("2号は戦闘力を持つ", kuririn.getSentoryoku());
16
17        assertTrue("ピッコロはクリリンより強い", pikkoro.getSentoryoku() > kuririn.getSentoryoku());
18        assertFalse("悟空はピッコロよりスター強い", goku.getSentoryoku() > pikkoro.getSentoryoku());
19    }
20
21 }
22
23
24 }
```

**ZSenshi.java:**

```
1 package jp.co.tdd.dragonball;
2
3 public class ZSenshi {
4
5     public static final String GOKU = "GOKU";
6     public static final String PIKKORO = "PIKKORO";
7     public static final String KURIRIN = "KURIRIN";
8
9     private int sentoryoku;
10
11     public ZSenshi(String name) {
12         if(name.equals(GOKU)) {
13             //悟空：フリーザ戦クリリンのことから悟空ウィキペディアより
14             this.sentoryoku = 150000000;
15         } else if(name.equals(PIKKORO)) {
16             //ピッコロ：フリーザ戦悟空ウィキペディアより
17             this.sentoryoku = 1800000;
18         } else if(name.equals(KURIRIN)) {
19             //クリリン：フリーザ戦悟空ウィキペディアより
20             this.sentoryoku = 12000;
21         }
22     }
23
24     public int getSentoryoku() {
25         return this.sentoryoku;
26     }
27
28 }
29 }
```

## 演習2：ポイント1

- ・ これだと、扱う戦士が増えるごとに定数とif条件増やさなきゃいけないんですけど…
  - ・ Z戦士ってもつといっぱいいるよね。ヤムチャとか。弱いけど
- ・ If文の中とか重複コードだらけだし、意図した動作(戦士によって戦闘力が異なる)をさせるためのコード以外のコードが多すぎる。
  - ・ シンプルな設計じゃない！！
- ・ 重複コードを排除して、キャラ追加しても ZSenshiを直さなくていいようにリファクタしてみよう！

## 演習2: ポイント1

- ・テスト側

- それぞれの戦士のクラスからインスタンス化する  
ようリファクタリング
- みんなZ戦士の仲間だから、戦闘力  
(getSentyoryoku)は同じように使える！

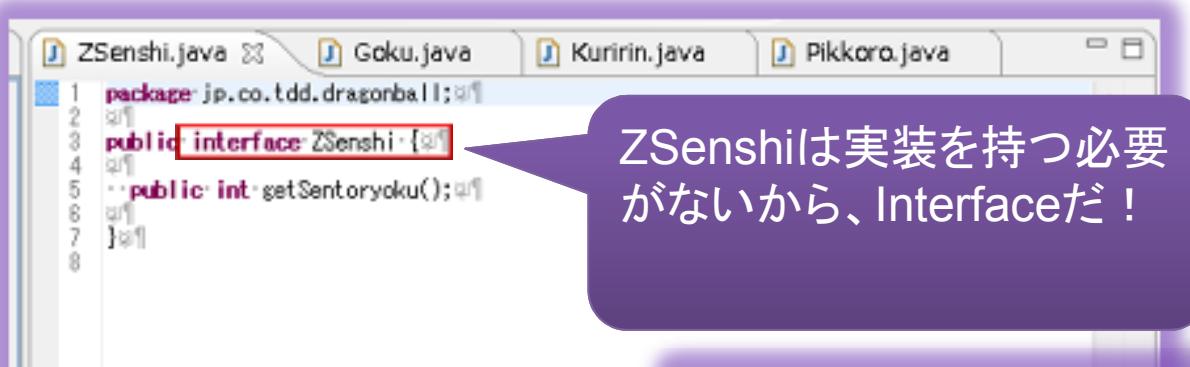
```
1 package jp.co.tdd.dragonball;
2
3 import junit.framework.TestCase;
4
5 public class ZSenshiTest extends TestCase {
6
7     public void testSentyoryoku() {
8
9         ZSenshi senshi = new Goku();
10        assertEquals(true, senshi.getSentyoryoku());
11
12        senshi = new Pikkoro();
13        assertEquals(true, senshi.getSentyoryoku());
14
15        senshi = new Kuririn();
16        assertEquals(true, senshi.getSentyoryoku());
17
18        ZSenshi goku = new Goku();
19        ZSenshi pikkoro = new Pikkoro();
20        ZSenshi kuririn = new Kuririn();
21
22        assertTrue("ピッコロはクリリンより強い", pikkoro.getSentyoryoku() > kuririn.getSentyoryoku());
23    }
24}
```

それぞれの戦士で違うクラスを、“Zsenshi”として同様に扱っている！

## 演習2： ポイント1

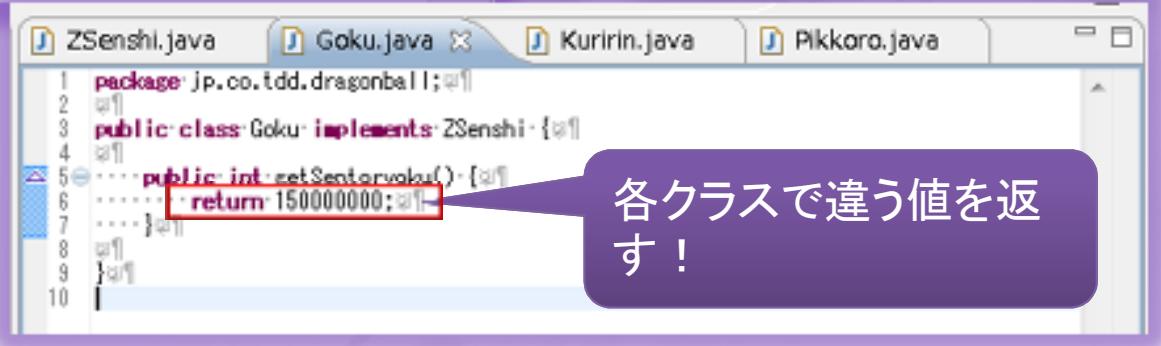
- 実装側

- Zsenshiは**インターフェース化**でコードすっきり！
- 戦闘力は各クラス違う値を返すよう実装した！



```
1 package jp.co.tdd.dragonball;
2
3 public interface ZSenshi {
4     int getSentoryoku();
5 }
6
7 }
```

ZSenshiは実装を持つ必要  
がないから、Interfaceだ！



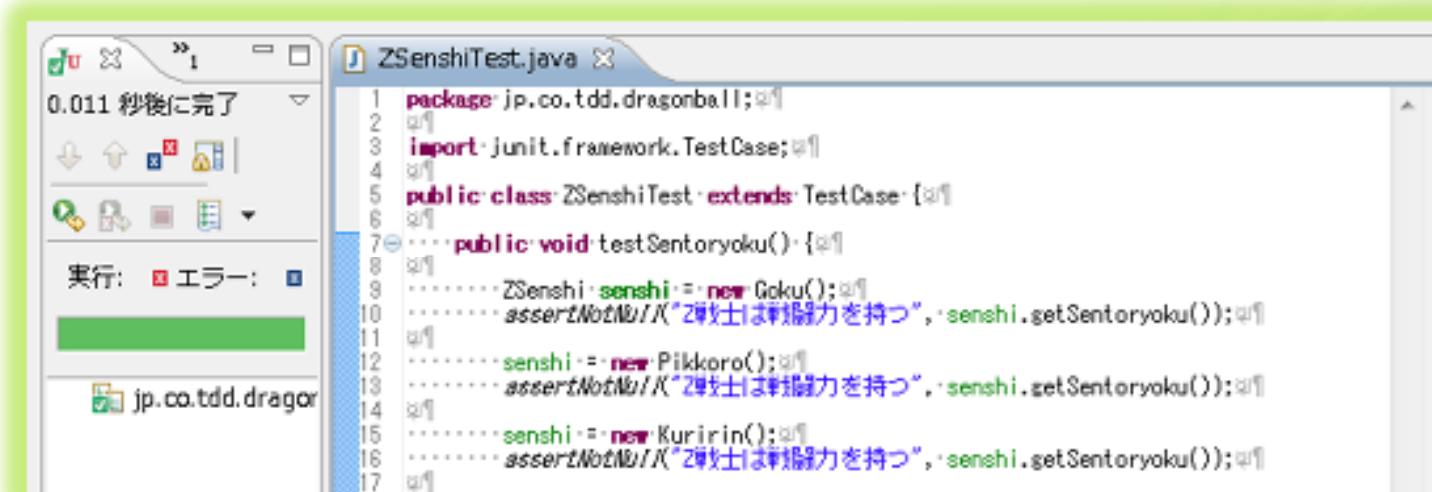
```
1 package jp.co.tdd.dragonball;
2
3 public class Goku implements ZSenshi {
4     int getSentoryoku() {
5         return 150000000;
6     }
7 }
8
9 }
```

各クラスで違う値を返  
す！

## 演習2： ポイント1

- ・テスト実行…

OKオールグリーンだ！  
リファクタリング完了！



The screenshot shows an IDE interface with a green border. On the left is a status bar with "0.011 秒後に完了" (Completed in 0.011 seconds) and a toolbar with various icons. Below that is a "実行" (Run) button with a red error icon and a blue success icon. On the right is a code editor window titled "ZSenshiTest.java". The code is as follows:

```
1 package jp.co.tdd.dragonball;
2
3 import junit.framework.TestCase;
4
5 public class ZSenshiTest extends TestCase {
6
7     public void testSentoryoku() {
8
9         ZSenshi senshi = new Goku();
10        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
11
12        senshi = new Pikkoro();
13        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
14
15        senshi = new Kuririn();
16        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
17    }
}
```

## 【コラム】 7. オブジェクト指向

### ■ シンプルな設計とオブジェクト指向

- ・ テストケースでは“どのキャラか”を意識せず  
に戦闘力を取得している
- ・ これは**オブジェクト指向**でいう**ポリモーフィズム(多様性)**という性質
- ・ テストコードがあるから、自信を持って大胆な  
リファクタリングができる
- ・ リファクタリングによりコードを**シンプルな設計**  
にしていくと、いつの間にか**オブジェクト指向**  
**も身につく！**

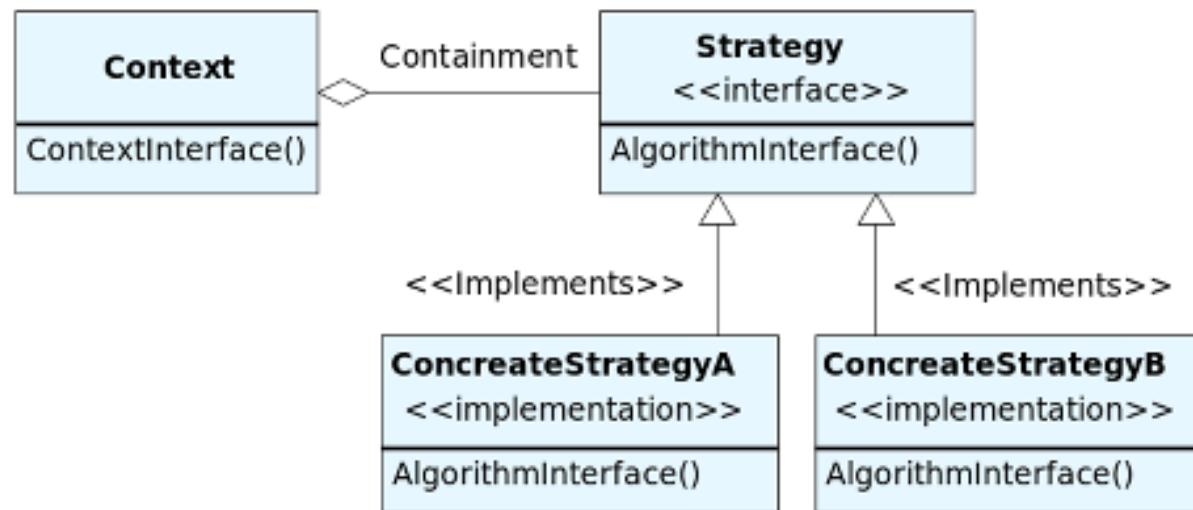
## 【コラム】8. デザインパターン

### ■ デザインパターン

- ・世の中には“デザインパターン”という、オブジェクト指向で楽をするために偉い人が考えた”型”がある
- ・有名なのはGoF(Gang of Four)の23パターン
- ・さっきの実装は実はそのうちの1つ、「ストラテジーパターン」っていうパターンだ！

## 【コラム】9. ストラテジーパターン

### ■ストラテジーパターン



- ・ Contextさん(さっきのケースではテストクラスが該当)が、Stragetyさん(Z戦士が該当)が誰かを意識せず扱う
- ・ かつ各々のStrategy( Z戦士)によりロジックを切り替えられる  
の2つがミソだ！

## 演習2: DB de TDD !

- ・仕様(TODOリスト)

- = Z戦士は皆“戦闘力”という共通の強さを測る値を持つ
- = ピッコロはクリリンより強い(= 戦闘力が高い)
- = 悟空はピッコロよりスゲー強い
- Z戦士は**皆“舞空術”**という技で空を飛べる。
- “舞空術”で**移動できる速さ**は3人で異なる。
  - ・ クリリンは“普通に”飛ぶ。
  - ・ ピッコロは“ビュンと”飛ぶ。
  - ・ 悟空は”目にもとまらぬ速さで”飛ぶ。

NOW PROGRAMMING・・・

# 演習2: 【サンプル実装】

- ・ ポイント2 舞空術
  - クラス(戦士)ごとに、飛び方が異なる
    - ・ さつきと同じストラテジーパターンができる！余裕だぜ！コピペコピペっと

ZSenshiTest.java

```
29  public void testBukujutsu(){  
30    
31  ZSenshi senshi = new Goku();  
32  assertEquals("悟空の舞空術","目にもとまらぬ速さで飛んだ！！",  
33  ,senshi.bukujutsu());  
34    
35  senshi = new Pikkoro();  
36  assertEquals("ピッコロの舞空術","ビュンと飛んだ！！",  
37  ,senshi.bukujutsu());  
38    
39  senshi = new Kuririn();  
40  assertEquals("クリリンの舞空術","普通に飛んだ！！",  
41  ,senshi.bukujutsu());  
42    
43 }  
44 }
```

Kuririn.java

```
9  public String bukujutsu(){  
10  String bukujutsu = "普通に飛んだ！！";  
11  System.out.println(bukujutsu);  
12  return bukujutsu;  
13 }  
14 }
```

Goku.java

```
9  public String bukujutsu(){  
10  String bukujutsu = "目にもとまらぬ速さで飛んだ！！";  
11  System.out.println(bukujutsu);  
12  return bukujutsu;  
13 }  
14 }
```

Pikkoro.java

```
9  public String bukujutsu(){  
10  String bukujutsu = "ビュンと飛んだ！！";  
11  System.out.println(bukujutsu);  
12  return bukujutsu;  
13 }  
14 }
```



## 演習2：ポイント2

- ・ また死ぬほど重複コードだらけだよね。コンソールに出力しない仕様になつたらどーすんの?
  - シンプルな設計じゃない！！
- ・ 舞空術メソッドの仕様が変わつても、各クラスの実装を修正しなくていいようにリファクタしてみよう！

## 演習2： ポイント2

- ・ ZSenshiに、舞空術のふるまいを集約(継承)
  - 実装が必要だから、Abstract Classだ！
- ・ クラス(各Z戦士)間で違うところだけ、違うふるまいをするメソッドを準備してあげよう！

ZSenshiTest.java

```
1 package jp.co.tdd.dragonball;
2
3 public abstract class ZSenshi {
4     ...
5     public abstract int getSentoryoku();
6
7     public abstract String getBukujutsuSokudo();
8
9     public String bukujutsu() {
10         String bukujutsu = getBukujutsuSokudo() + "飛んだ！！";
11         System.out.println(bukujutsu);
12         return bukujutsu;
13     }
14 }
15 }
```

各クラスの違いを吸収する、このメソッドがミソだ！

同じロジックは共有しよう！

Kuririn.java

```
9 public String getBukujutsuSokudo() {
10     ...
11     return "普通に";
12 }
```

違うところだけ実装！

Goku.java

```
9 public String getBukujutsuSokudo() {
10     ...
11     return "目にもとまらぬ速さで";
12 }
```

Pikkoro.java

```
9 public String getBukujutsuSokudo() {
10     ...
11     return "ビュンと";
12 }
```

## 演習2： ポイント2

- ・テスト実行…

OKオールグリーンだ！  
リファクタリング完了！

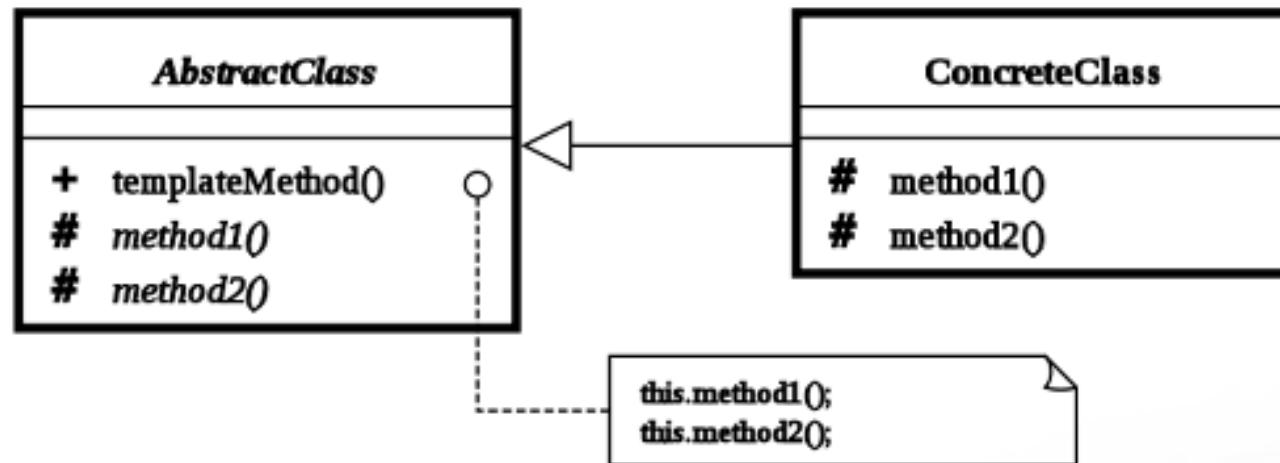


The screenshot shows an IDE interface with a green border. On the left is a status bar with "0.011 秒後に完了" (Completed in 0.011 seconds) and a toolbar with various icons. Below that is a "実行" (Run) button with a red error icon and a blue success icon. On the right is a code editor window titled "ZSenshiTest.java" containing the following Java code:

```
1 package jp.co.tdd.dragonball;
2
3 import junit.framework.TestCase;
4
5 public class ZSenshiTest extends TestCase {
6
7     public void testSentoryoku() {
8
9         ZSenshi senshi = new Goku();
10        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
11
12        senshi = new Piccolo();
13        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
14
15        senshi = new Kuririn();
16        assertEquals("2战士は戦闘力を持つ", senshi.getSentoryoku());
17    }
}
```

## 【コラム】 10. テンプレートメソッドパターン

### ■テンプレートメソッドパターン



- ・ 実はさっきの実装もデザインパターンの1つ、「テンプレートメソッドパターン」だ！
- ・ `templateMethod()=bukujutsu()`、`method1()=getBukujutsuSokudo()`
- ・ 「大体同じ処理なんだけど、ここんとこだけちょっと違うんだよなー」って時に有効だ！
- ・ 共通化したロジックの中で、子クラスで実装する予定のメソッド(ちょっとだけ違うとこ)を準備しておくのがミソだ！



# 演習2 おしまい

# まとめ

## ■ テストがないコードはレガシーコードだ！

- この一言を胸に刻もう

## ■ TDD: テスト駆動開発とは

- テストから先に作る開発手法
- テスティングフレームワークによるテスト自動化
- リファクタリングによるシンプルな設計

## ■ そしてオブジェクト指向へ

- リファクタリングでポリモーフィズムの目覚め
- GoFの23のデザインパターン

## 【おまけ】これからのこと①

### ■この辺の本が面白いから読んでみよう

- バグがないプログラムの作り方 / 川端 光義
- テスト駆動開発入門 / Kent Beck
  - TDDについてもっと深く知れる。
- オブジェクト指向脳の作り方 / 牛尾 剛
  - オブジェクト指向についての理解が深まる。
- アジャイルサムライ / Jonathan Rasmusson
  - アジャイルプロセスの中でなぜTDDが重要か？がわかる。アジャイル開発面白い！
- Clean Coder / Robert C. Martin
  - プロのプログラマとして生きるなら。

## 【おまけ】これからのこと②

### ■もしあなたがWebプログラマなら・・・

MVCモデルのプログラミングにTDDを適応するヒント

- コントローラ(Servletとか、StrutsのActionクラス)に直接ビジネスロジックを書くな！“**DI**”しろ！
  - DI(Dependency Injectoin)で、業務ロジックをPOJO化するんだ！
- モデル、特にDB周りの試験は“**モックオブジェクト**”を使え！
  - 詳しくはWebで
- ビューのjUnitでの試験は難しい。テスト自動化には“**Selenium**”が有効だ！



さいごに…

お楽しみいただけただろうか？

何か感じるところはあつただろうか？

最後に、僕からのメッセージだ！



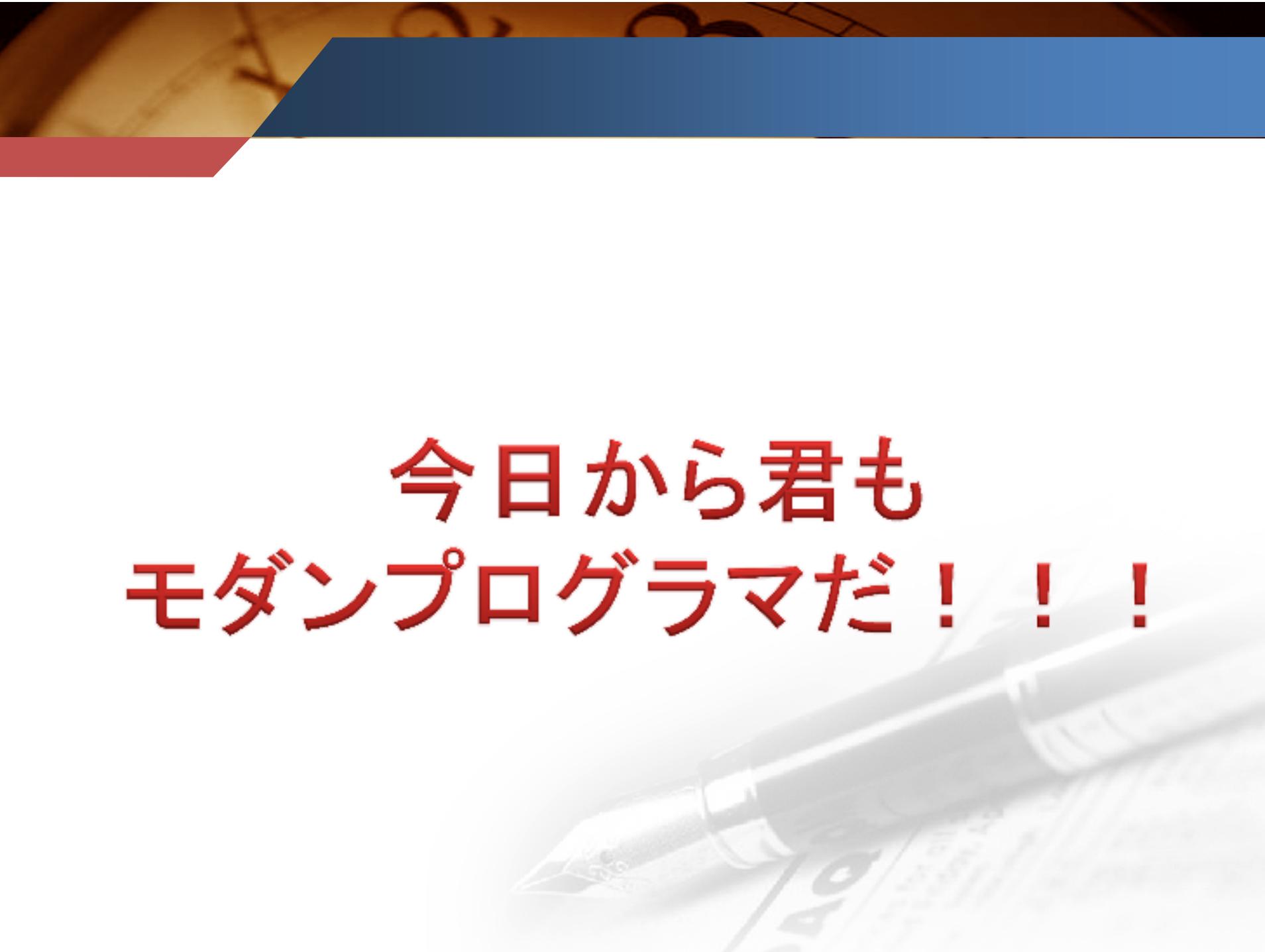
# たくさんプログラムを書こう

- いくらたくさん本を読んでも、練習することでしかプログラムは上達しないぞ！
- まずは**写経**だ！実践あるのみ！！
- つかプログラミングって**超楽しい**!!!!



## 現場でもテストコードを書こう

- ・ ウォーターフォール型の開発でも、TDDは品質向上に有効だ！
- ・ 上司の許可なんていらない。必要なのは君の「**もっと良いシステムを作りたい**」という気持ちと、ほんの少しの勇気だけだ！



今日から君も  
モダンプログラマだ！！！

おわり

ご清聴  
ありがとうございました！

# ありがとうございました

## ■ Special Thanks

- ・ くだらくん
- ・ こうもとくん
- ・ ほんださん
- ・ とおやまくん

## ■ About Me

- ・ 本日のコードGithubにあります
  - <https://github.com/martinloverse>
  - ・ 質問等Twitterへどうぞ！
    - @martin\_lover\_se