車窓からのＴＤＤ

オブジェクト倶楽部　北野弘治

０．はじめに

こんにちは。いまから、私がこれまでに経験したプログラミングの中で，最もユニークな体験を紹介したいと思います.それは,ペアプログラミング[1]とテスト駆動開発[2]（TDD=Test-Driven-Development）を走る電車の中で実践したエピソードです． ペアプログラミングとは，二人で一つのプログラムを開発する手法で,欠陥が少なく可読性の高いシンプルなコードを書くことができます.テスト駆動開発とは、『テストファースト』とも呼ばれ，テストを先に書いてから実装を行う開発の進め方です．この２つの手法は相性がよく，両方とも現在流行している XP のプラクティスとなっています．

体験談に入る前にテスト駆動開発（以下 TDD）について少し説明しょう．今までの開発では,「設計」⇒「実装」⇒「テスト」が基本的な開発の進め方だったはずです．TDD では，これを逆転させて,「テスト」⇒「実装」⇒「設計」，という順序で行います．ただし，最後の「設計」は，実装によって結果的に作られた設計を再設計することから,「リファクタリング」と呼ばれています．

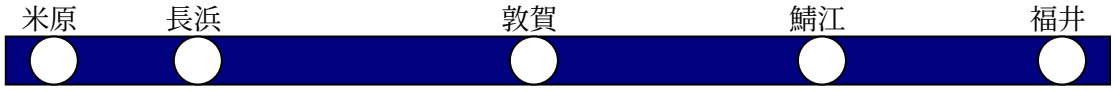
この過程全体を通して，ユニットテストツールである JUnit が，開発のナビゲーションを行います．このツールでは，通らないテストを「赤」，通ったテストを「緑」で表示するため，全体のリズムは３拍子で以下のようになります．

TDD の基本的な手順

ステップ	目的	JUnit の色
1	テストを書く	■（赤）
2	実装する	■（緑）
3	リファクタリングする	■（緑）

さて，この資料では、北野と平鍋の両名でスタック[3]をお題として TDD を「電車の中で」行った実録を示しています。テスト駆動開発の進め方や考え方,そしてそのリズムを感じ取れて頂ければ幸いです。

ちなみに、このペアプロは、米原駅から福井駅までの電車の中で行いました。[4]



１．米原駅　～すべてはここから始まった～

平鍋：「北野くん。北野くんのパソコンって充電どのくらいもつ？！」
北野：「えっ？！何ですか？メールでも見るんですか？？」
平鍋：「いやいや，今日のセミナーでやったスタックの演習だけど、実際にやってみようかと思って。」
北野：「そうですね。ちょっと待ってください。今準備します。」

（パソコンをカバンの中から取りだし、起動する）
北野：「はい、Eclipse 立ち上がりました。」

---

[1] [William02] *Laurie William/Robert Kessler*,『ペアプログラミングーエンジニアとしての指南書』
[2] [Beck02] *Kent Beck*, “Test-Driven Development: by Examle”
[3] 今回のスタックは、C++標準ライブラリのスタック stack<int>の仕様にあわせています。pop() が値を返さないことに注意してください.
[4] 2003 年 4 月に開催した XP アンギャ in 浜松からの帰路で行いました。

平鍋：「じゃぁ、TDD でスタックを作成する前に、スタックの仕様を確認しましょう。」

・isEmpty()でスタックが空の場合、true。それ以外 false を返す。
　　boolean isEmpty()
・size()でスタックのサイズを取得する。
　　int size()
・push()で引数の値をスタックの一番上に積む。
　　void push(int value)
・pop()でスタックの一番上の値を取り除く。
　　void pop()
　　スタックが空の場合、java.util.EmptyStackException が発生する
・top()でスタックの一番上の値を取得する。
　　int top()
　　スタックが空の場合、java.util.EmptyStackException が発生する

北野：「スタックのクラス名は、Stack でいいですよね。」
平鍋：「はい、じゃぁ、最初は私がドライバー[5]になります。」
北野：「はい、わかりました。それじゃぁ、最初にプロジェクトを作成しましょう。」
平鍋：「プロジェクト名は、『stack』でいい？」
北野：「はい。プロジェクトを作成したら、JUnit を組み込んでおきましょう。」
平鍋：「プロジェクト作成と JUnit の組み込みも終わったので、テストケースクラスを作ろうか？」
北野：「そうですね、Stack のテストなので、StackTest というクラス名にしましょうか？」
平鍋：「はい。StackTest クラスのテンプレートを作成しました。」

```
import junit.framework.TestCase;
pulibc class StackTest   extends   TestCase {
}
```

２．長浜駅　～Assert ファースト～

平鍋：「まず、何のテストを書く？」
北野：「そうですね。。。Stack を new したら空．すなわち，isEmpty()が true を返してくるとか？」
平鍋：「OK。じゃぁそのテストコードを書きましょう。testCreate()でいいよね？」
北野：「はい」

```
import junit.framework.TestCase;
pulibc class StackTest   extends   TestCase {
    public void testCreate() {
        assertTrue(stack.isEmpty ());
    }
}
```

平鍋：「これでいい？」
北野：「良いですけど。。。assertTrue しか書いてないですが。。」
平鍋：「いやもちろん、これだけじゃだめだけど、TDD を行っていく時には、まず Assert メソッドを先に書くというのが常套手段なんだよ。それは、なにを確認したいのか？というのを最初に明確にしておきたいから。。。それからその確認を行うための付属のコードを書いていくわけ。」
北野：「なるほど。。納得。。そうだったんですか・・・」
平鍋：「じゃぁ、付属コードも書いて。。これでいいよね。」

```
import junit.framework.TestCase;
pulibc class StackTest   extends   TestCase {
```

5 ペアプログラミングでは, コードを書くほうをドライバー, 横で支援する人をナビゲーターと呼びます.

```
    public void testCreate() {
        Stack stack = new Stack();
        assertTrue(stack.isEmpty());
    }
}
```

平鍋：「ここでコンパイルしてみようか？Eclipse だとどうするんだっけ？」
北野：「Eclipse だと保存すればコンパイルが走ります。僕は、Emacs のキーバインドを使ってるので、Ctrl-x s で保存です。」
平鍋：「じゃぁ保存します。コンパイルエラーが出たね。」
北野：「そうですね、Stack クラスがないと言われてますね。それじゃぁ Stack クラスを作りましょう。」
平鍋：「Eclipse だと簡単に作れるって聞いたけど。。。」
北野：「えっと、コンパイルエラーの所にある X 部分をクリックすると、Stack クラスを作成するというメニューが現れるので、簡単です[6]。」
平鍋：「ほぉ〜。噂通り使い勝手いいねー。」
北野：「Stack クラスのテンプレートが出来ましたね。」

```
public class Stack {
}
```

平鍋：「まだコンパイルエラーが出てますね。えっと、コンパイラは isEmpty メソッドを作れ！と言ってるね。はい。はい。じゃぁ作りましょう。ここでは、コンパイルを通す最小のコードを書くので、boolean のデフォルト値である false を返しておきましょう[7]。」
北野：「コンパイラーもプログラマーとして参加しているみたいですねー。ペアプロならぬトリプロだ！！（笑）。でも冗談抜きで、TDD はコンパイラのチカラも使って開発していくようなかんじですね。」
平鍋：「そうだね。コンパイラのエラーメッセージに耳を傾けると，次にやるべきことがわかる．さらに，Eclipse のような開発環境であれば足らないメソッドやクラスを作成するように催促してくれるからね。頼もしい味方ですね。」

```
public class Stack {
    public boolean isEmpty(){
        return false;
    }
}
```

３．長浜と敦賀の間　〜最初は赤で、次に緑、そしてリファクタリング！！〜

平鍋：「これでコンパイルが通りました。じゃぁテストしてみましょう。」
北野：「赤[8]になりましたね。」
平鍋：「期待通りの赤です。これでステップ１完了．このように、TDD を行う時には、赤にしてそれから緑[9]にして一つ一つ確認していくのが手順．」
北野：「そうですね。なんか、赤から緑にするのって、テスト仕様書にテスト通ったよ！ってチェック印を付けていく感じと似てますね。」
平鍋：「ずーっと緑だと、『ホントにテスト動いてるの？』『もしかして他のテストしてない？』とか不安になるしね。」
北野：「そうなると、TDD では、赤と緑が交互入り変わるようになるわけですね。それもテンポ良く。」
平鍋：「そうそう．基本は，「赤，緑，リファクタリング」の３拍子．リファクタリングは後で見せるね．」
平鍋：「本題に戻って、じゃぁ次にテストを通すための最小のコードを書いてみようか？」
北野：「うーん。。。empty だから。。。size とか変数を作って。。。」

---

[6] Eclipse のこの機能を，「クイックフィックス」（Quick Fix）と呼ぶ．
[7] 実は，最初のテストを失敗させたい，という意図もあって false を返している．
[8] 赤：JUnit を利用してテストを行った場合、テストが失敗する様。
[9] 緑：JUnit を利用してテストを行った場合、テストが成功する様。

平鍋：「えっ？そこまで考えちゃう？？単純に、true を返せばテスト通るでしょ？」
北野：「あっ、そんなんでいいの？？」

```
public class Stack {
    public boolean isEmpty(){
        return true;
    }
}
```

平鍋：「はい、テストを通すのが今の目標なので、今はそんな先まで考えません。」
平鍋：「こういう単純な実装をフェイクする（Fake It!）って TDD 業界では言います。」
北野：「なるほど。。」
平鍋：「テストしてみましょう。」
北野：「緑になりましたね。これで，ステップ２完了という訳ですか．では，ステップ３のリファクタリングしましょう。」
平鍋：「いや、今はまだ重複したコードが見えないからリファクタリングは不要．他の仕様をちょっと実装して，必要になったところで isEmpty()のリファクタリングをしてみようよ。」
北野：「はい。了解です。じゃぁ次は、push()のテストをしてみましょうよ。」
平鍋：「OK。じゃぁ push()の確認を top()を利用して確認してみようか？もう一度ステップ１から始めます．赤の状態を作りましょう．」
北野：「テストメソッドの名前は、testPushAndTop()にしましょう。」
平鍋：「了解。assertEquals を最初に書いてっと」

```
import junit.framework.TestCase;
pulibc class StackTest   extends   TestCase {
    public void testCreate() {
        Stack stack = new Stack();
        assertTrue(stack.isEmpty());
    }
    public void testPushAndTop() {
        Stack stack = new Stack();
        assertEquals(1, stack.top());
    }
}
```

北野：「Stack のインスタンスを作成する部分ですが、ダブってますね。」
平鍋：「setUp メソッドに出そうか？setUp メソッドは、test***メソッドが呼び出される毎に呼び出してくれるから、OK でしょ？」
北野：「では、testPushAndTop メソッドをコメントアウトして、testCreate メソッドの部分を、setUp メソッドを使ってテストしてみましょう。」

```
    Stack stack;
    public void setUp() {
        stack = new Stack();
    }
    public void testCreate() {
        assertTrue(stack.isEmpty());
    }
/*
    public void testPushAndTop() {
        Stack stack = new Stack();
        assertEquals(1, stack.top());
    }
*/
```

平鍋：「これでいい？」

北野：「はい。テストしてみましょう。」
平鍋：「緑のままだ。テスト成功！！じゃぁ testPushAndTop もそうしようか？」
北野：「コメントを外して、あっ！push し忘れてますね。」
平鍋：「あっ。本当だ。期待している値が 1 なので、1 を push すればいいよね。」

```
  public void testPushAndTop() {
      Stack stack = new Stack();
      stack.push(1)
      assertEquals(1, stack.top());
  }
```

北野：「これで，テストは書けました．この状態だと、コンパイル通らないので、まずはコンパイル通るようにしましょう。」
平鍋：「まず、push メソッド。これは、void なので、実装を空にしておきましょう。あと、top メソッド。これは、return 0 とでもしておきましょう。」

```
  public void push(int value) {
  }
  public int top() {
      return 0;
  }
```

北野：「これでコンパイル通りますね。この状態でテストをすれば赤になります。テストしてみましょうか？」
平鍋：「テスト実行！！赤になりました。期待通りテストに失敗しましたね。ステップ１の完了です．」
北野：「じゃぁ、テストを通るようにしましょう。ここでは、top メソッドの所を、『return 1;』に変えればいいんですかね？」
平鍋：「その通り。フェイクです。」

```
  public int top() {
      return 1;
  }
```

平鍋：「テスト実行！！緑になりました。期待通りテストに成功しましたね。ステップ２完了です．」
北野：「じゃぁ、リファクタリングしませんか？」
平鍋：「そうだね。push メソッドで入れた値を top メソッドで返すように変えようか？」

```
  private int value;
  public void push(int value) {
      this.value = value;
  }
  public int top() {
      return value;
  }
```

平鍋：「これでいい？」
北野：「はい。OK です？テストしてみましょうか？緑になるはずですね。」
平鍋：「テスト実行！！緑になりました。リファクタリング成功！ステップ３の完了です．ここではこれ以上はリファクタリングできないので、次のテーマを決めましょう。」
北野：「次は、size メソッドとかはどうでしょうか？」
平鍋：「OK。じゃぁ、Push したら size が増えるというテストを書こうか。」
北野：「testPushAndSize()という名前がいいです。」
平鍋：「はい。assertEquals を最初に書いて。。。これでいい？」

```
  public void testPushAndSize() {
      s.push(1);
      assertEquals(1, s.size());
  }
```

北野：「OK です。じゃぁコンパイル通るようにしましょうか？ここでは size メソッドで return 0 とすればいいですね。」
平鍋：「そうだね。」

```
public int size() {
    return 0;
}
```

北野：「テストしてみましょうか？」
平鍋：「テスト実行！　　赤になりました。size()のテストが通っていないってことですね。」
北野：「それでは、テストを通るようにしましょう。」
平鍋：「size（）の戻り値を１にしましょう。そしてテスト！！緑になりましたね。」

```
public int size() {
    return 1;
}
```

平鍋：ここで，TDD の３つのステップを，もう一度おさらいしておきましょう． このステップを踏み外さないように！」

ステップ	目的	JUnit の色
1	テストを書く	■（赤）
2	実装する	■（緑）
3	リファクタリングする	■（緑）

４．敦賀駅　～とらいあんぎゅれーしょん？？～

平鍋：「そろそろ敦賀につくね。」
平鍋：「今度は、ちょっと趣向を変えて、もう一つテストを追加しましょう。testPushAndSize()に。」
北野：「どういうことですか？」
平鍋：「こういうことです。」

```
public void testPushAndSize() {
    s.push(1);
    assertEquals(1, s.size());
    s.push(2);
    assertEquals(2, s.size());
}
```

平鍋：「2 回 push したら、サイズが２になるはずですよね？そういうテストを追加し、先のテストと今回追加したテストを通るような一番シンプルなプログラムを考えるわけです。」「もう一回赤の状態にもどしてから，前に進む．コードを変更したり追加するときは，必ず赤にするのが TDD の掟.」
北野：「なるほどー。そうすれば、より正確なコードを導くことができますね。僕も意識していませんでしたが、こういうことは日常的にやっていたような気がします。」平鍋：「それじゃ、まず、このテストをしてみて赤になることを確認しましょう！。」
北野：「はい。赤になりましたね。」
平鍋：「じゃぁこの２つのテストを通るコードを考えましょう。北野くんならどうする？」
北野：「サイズの値を持たせるインスタンス変数を Stack クラスに用意して、
push()が呼ばれる度にその値をプラス１する。size()はその値を返すように
すればいいと思います。」
平鍋：「御意。北野くん、ドライバー変わりましょうか？」
北野：「はい。」

```
public class Stack {
    private int size;

    public void push(int value) {
        this.value = value;
        ++size;
    }

    public int size() {
        return size;
```

```
        }

    }
```

北野：「これでどうですか？」
平鍋：「はい。じゃぁコンパイルして、テストしましょう。」
北野：「テスト実行！！緑です。（パチパチ）」
平鍋：「上手くいきましたね。この方法はトライアンギュレーション(Triangulation)と言って、テストケースをいくつか用意し、その全てのテストケースを通過するコードを書くことで，一般解を導きくという方法です。テスト１つだと，フェイクで済んでしまうでしょ，そこにテストを足して，フェイクでは通らないようにする．そこから，正しい実装を導くんです．三角測量のように，２点から測量してその交差点上に解を作ります．TDD 開発のパターンの一つです。」

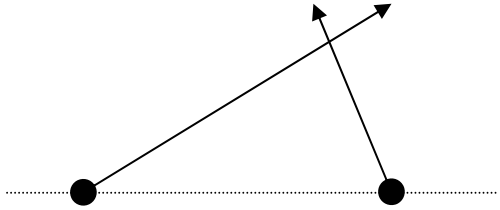


図１：triangulation

平鍋：「じゃぁ、この方法を利用して、今までフェイクしたままになっている部分，のテストを追加して進めていかない？どこか気になるところある？」
北野：「うーん。僕は、isEmpty()が return true になっているのがきにかかりますね。
なんとかしたい。なので、testPushAndTop()で push した後に isEmpty()が
FALSE を返すことをテストとして追加したいですね。」

```
  public void testPushAndTop() {
      stack.push(1);
      assertFalse(stack.isEmpty());
      assertEquals(1, stack.top());
  }
```

北野：「こうです。」
平鍋：「ほー。いいですね。これもトライアンギュレーションですね。」
北野：「テストを実行して、赤になることを確認して、コードを考えましょう。」
平鍋：「もっともシンプルなのは？」

```
  public boolean isEmpty() {
      return size == 0;
  }
```

北野：「こうですかね？size が０だったら true を返す。」
平鍋：「OK。テストしてみましょう。」
北野：「テスト実行！！緑になりました。」

５．北陸トンネルね！いっちょらい！　～Exception テスト～

北野：「次に新しいテストを追加しませんか？」
平鍋：「なにをしますか？」
北野：「pop のテストをしたいですね。まず、なにも push していない時の pop のテストを追加しましょう。」
平鍋：「おぉ、その場合は、EmptyStackException が throw されることを確認しないといけないですね。いままでとちょっと違った感じのテストだよ。」
北野：「そうです。ちょっと面白そうなテストなのでやってみたかったんです。こんなテストでどうでしょうか？」

```
public void testEmptyPop() {
    try {
        stack.pop();
        fail();
    } catch (EmptyStackException exception) {
    }
}
```

平鍋：「うん。pop()の後に fail()をして、ここを通ったらテスト失敗。OK。で、pop()で Exception が throw されるから、期待している例外を catch している。ん？」
北野：「どうしたんですか？」
平鍋：「catch している Exception の変数名だけど、exception ではなくて、expected（期待されている例外）にした方が意味的にいいよね？」
北野：「あぁ。そうですね。分かりました。」
北野：「では、コンパイルしてみましょう。コンパイルエラーになりました。pop()がないと言ってますので、pop()を追加します。」
平鍋：「はい。とりあえず、コンパイルが通るコードを書きましょう。

```
public void pop() {
}
```

北野：「こうですね。これでテストしてみましょう。赤になるはずです。」
平鍋：「はい。赤になりましたね。これは、pop()の後の fail()まで行ってしまっているからですね」
北野：「では、テストを通る最小のコードを書いてみます。」
北野：「pop()に Exception を throw するようにすればいいですよね？」

```
public void pop() {
    throw new EmptyStackException();
}
```

平鍋：「はい。これでいいです。ではテストして緑になるか確認しましょう。」
北野：「緑になりました！！（パチパチ）」
平鍋：「これが、例外が throw される時のテストのやりかたです。」
平鍋：「じゃぁ、ここもトライアンギュレーションをしてコードを正確にしていきましょう。」
北野：「じゃぁ、１回 push していれば、pop しても Exception を throw しないテストをしてみましょう。平鍋さんかわりますか？」
平鍋：「はい。こういうテストですね？」

```
public void testPushAndPop() {
    stack.push(1);
    stack.pop();
    assertEquals(0, stack.size());
}
```

平鍋：「テストしてみましょう。赤になるはずですよね？」
北野：「そうですね。今の pop は無条件で Exception を throw するので、例外が発生しますから、テスト失敗しますね。　テスト実行！！　赤になりました。」
平鍋：「それでは、pop の部分をこう変えましょう。」

```
public void pop() {
    if (isEmpty())
        throw new EmptyStackException();
}
```

平鍋：「isEmpty()が true　つまり size が０の場合は EmptyStackException を throw するようにし

ます。」
北野：「はい。OK です。僕はフツーに size==0 とするのかなぁ～って思ってたんですが、isEmpty()メソッドがありましたね。。(^^ゞ」
平鍋：「じゃぁテストしてみましょう。緑になりましたね。（パチパチ）」
北野：「あれ？top のところでも EmptyStackException が throw されますね。」
平鍋：「そうだっけ？？あ、そうだね。じゃぁ TOP の例外部分のテストもしよう」

```
public void testEmptyTop() {
    try {
        stack.top();
        fail();
    } catch (EmptyStackException exception) {
    }
}
```

平鍋：「これでいい？」
北野：「はい。テストしてみましょう。まだ top()には例外 throw のコードを書いていないので、fail()が実行されて、テストが失敗するはずです。」
平鍋：「OK.テスト実行！！赤だね。」
北野：「では、pop()の所と同じようにテストを通るようなコードを書きましょう。」
平鍋：「そうだね[10]。」

```
public int top() {
    if (isEmpty())
        throw new EmptyStackException();
    return value;
}
```

平鍋：「一度これでテストを通るか確認してみよう。　テスト実行！！　緑になったね。」
北野：「でも、top()と pop()の EmptyStackException を throw する部分が同じロジックになってますね。リファクタリングしましょうよ。」
平鍋：「このリファクタリングには、ExtractMethod を使おう。」

```
public int top() {
    emptyCheck();
    return value;
}

public void pop() {
    emptyCheck();
}

private void emptyCheck() {
    if (isEmpty())
        throw new EmptyStackException();
}
```

平鍋：「これで緑になれば、リファクタリング成功だね。」
北野：「テスト実行！！　緑だー。（ぱちぱち）」

６．トンネルを抜けると、そこは？！　～TDD ラストスパート～

平鍋：「さぁ、TDD がのりにのってきたね！もう北陸トンネルも通りこしてる。さて、次はなに？」
北野：「次は、pop したら一番上の要素を取り除くってことです。」

---

[10] もし気付いていても，ここで top と pop のチェックコードを共通化しよう，とあせってはいけない．まずはダブった形で緑にすること．それから，やおらリファクタリングするのが正しい TDD.

平鍋：「そうか。それはまだやってなかったね。テストを書いてみましょう。」

```
        public void testPushAndPop() {
                s.push(1);
                s.pop();
                assertEquals(0, s.size());
        }
```

平鍋：「push して pop したら、size()が０になる。このテスト。」
北野：「テストしてみましょう。」
平鍋：「赤になるはずだね。やってみよう。赤になったね。」
北野：「今は、pop をしても size の値を変えていないからですね。」
平鍋：「じゃぁ、このテストを通るコードにしましょう。」

```
  public void pop() {
      emptyCheck();
      size--;
  }
```

北野：「pop が呼ばれる毎に size の値をマイナス１する。これで OK です。」
平鍋：「じゃぁ、テストしてみよう。」
北野：「緑になりましたね。」
平鍋：「さぁて、次は？」
北野：「えっと、スッタクの肝心の特性である、先入れ後出しのコンテナが実装されていないとおもうのですが。。。今は一段のスタックなので。。。」
平鍋：「じゃぁ、そのテストを書いてみましょう。」

```
  public void testPushPushPopTop() {
        s.push(1);
        s.push(2);
        assertEquals(2, stack.size());
        s.pop();
        assertEquals(1, stack.top());
  }
```

北野：「Push を２回して、１回 Pop し、その時に Top をすると、１度目に Push した値が取得されるというテストですね。」
平鍋：「じゃぁ、今の状態でどうなるかみてみよう。テスト実行！！」
北野：「赤になりましたね。今はさっきも言いましたが１段だからですよね？」
平鍋：「それじゃぁ、複数段対応にしましょう。どうする？北野君」
北野：「そうですね。int の配列をつかいましょう。」
平鍋：「You Have!」[11]
北野：「I Have!」[12]

```
  private int[] value = new int[10];

  public void push(int value) {
        this.value[size++] = value;
  }

  public int top() {
        emptyCheck();
        return value[size - 1];
```

[11] パイロットが副操縦士に交代するときにいう文句。「あなたにまかせる！」
[12] You have!の返事。「がってんだー！」

```
    }
```

北野：「配列のインデックスにはちょうど size 変数がインデックスになるので、それを利用します。これでいいですか？」
平鍋：「OK。」
北野：「テスト実行！！」
平鍋：「緑になりましたー。。。」
北野：「これで、全ての ToDo リストが終了しましたね。もう鯖江も通過して、そろそろ福井ですね。５０分ぐらいですか？この TDD の演習は。」
平鍋：「そうだね。上手くいったね。もうリファクタリングもする必要もなさそうだし、奇麗なソースがかけてよかった。」
北野：「これを記録して、みんなに公開したいですね。」
平鍋：「そうだね」

Stack.java

```java
import java.util.EmptyStackException;

/**
 * @author koji,hiranabe
 */
public class Stack {
        private int[] value = new int[10];
        private int size;
        public boolean isEmpty() {
                return size == 0;
        }
        public int top() {
                emptyCheck();
                return value[size - 1];
        }
        public void push(int value) {
                this.value[size++] = value;
        }
        public int size() {
                return size;
        }
        public void pop() {
                emptyCheck();
                --size;
        }
        private void emptyCheck() {
                if (isEmpty())
                        throw new EmptyStackException();
        }
}
```

StackTest.java

```java
import java.util.EmptyStackException;
import junit.framework.TestCase;

/**
 * @author koji
 */
public class StackTest extends TestCase {

        private Stack stack;
        protected void setUp() {
                stack = new Stack();
```

```
        }
        public void testCreate() {
                assertTrue(stack.isEmpty());
        }
        public void testPushAndTop() {
                stack.push(1);
                assertFalse(stack.isEmpty());
                assertEquals(1, stack.top());
                stack.push(2);
                assertEquals(2, stack.top());
        }
        public void testPushAndSize() {
                s.push(1);
                assertEquals(1, stack.size());
                stack.push(2);
                assertEquals(2, stack.size());
        }
        public void testEmptyPop() {
                try {
                        stack.pop();
                        fail();
                } catch (EmptyStackException expected) {
                }
        }
        public void testPushAndPop() {
                stack.push(1);
                stack.pop();
                assertEquals(0, stack.size());
        }
        public void testPushPushPopTop() {
                stack.push(1);
                stack.push(2);
                assertEquals(2, stack.size());
                stack.pop();
                assertEquals(1, stack.top());
        }
        public void testEmptyTop() {
                try {
                        stack.top();
                        fail();
                } catch (EmptyStackException expected) {
                }
        }

}
```

6．まとめ　～TDD の利点～

北野：「おもしろかったですね.」
平鍋：「なにが気に入った？」
北野：「まず，コードが変更されたり追加されるときには，必ずテストが追加される，ということ.」
平鍋：「うん．これによって，完成したコードはすべての部分がテストコードを一度は通る，という状態になるんだ.」
北野：「コードに自信がでます．それから，実装を始めるときに，まず実装の中身よりもインターフェイスに意識が集中する，ということ.」
平鍋：「その通り．普通，設計してからコードを書くわけですが，ほとんどの人は，クラスのインターフェイスを実装側の視点で考えてしまう．テストを最初に書くと，そのクラスを使う人の立場で最初のインターフェイスを考えるクセが着くね．他には？」
北野：「？」
平鍋：「大切な点として，クラスの仕様が“テスト可能な仕様”になること． 時々，テストできないクラスというのを作ってしまうことがある．しかし，XP 的には，そういうクラスはダメ．必ずテストで

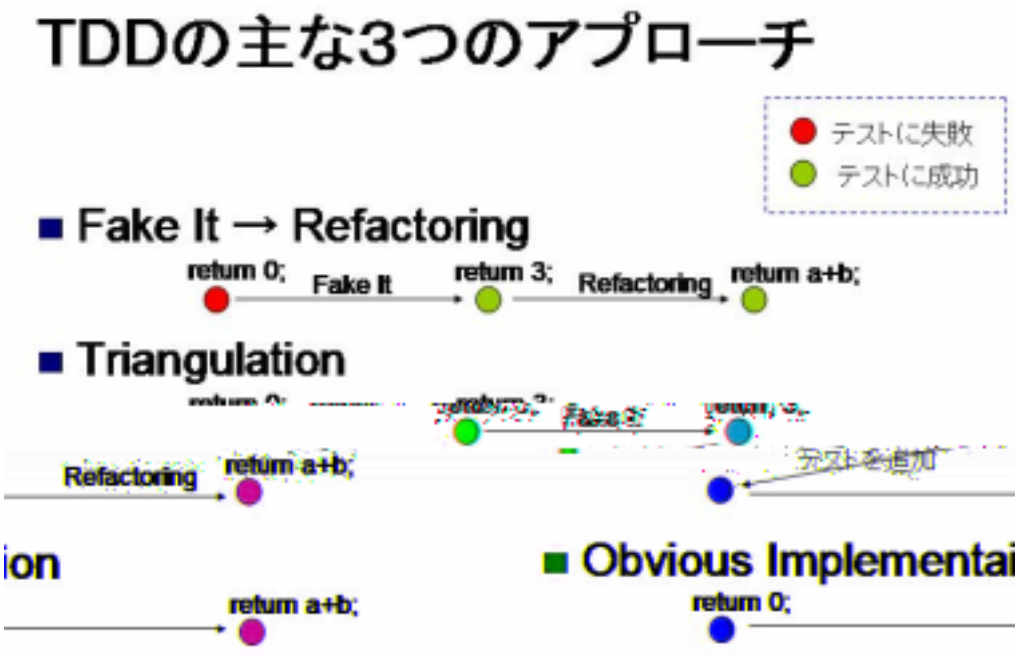
きなくてはならない．TDD を実践すれば，このテスト可能性（Testability）を常に確保できる．」
北野：「そうですね．それから，赤・緑・リファクタリングのリズムがあって，とにかく楽しいですよね！」
平鍋：「Good Point！まとめておこう．」

TDD の効用：
・　すべてのコードがテストを通った状態で完成する．
・　実装よりも使う側のインターフェイスを重視した設計になる．
・　テスト可能性を確保した設計になる．
・　開発にリズムができて，楽しい！

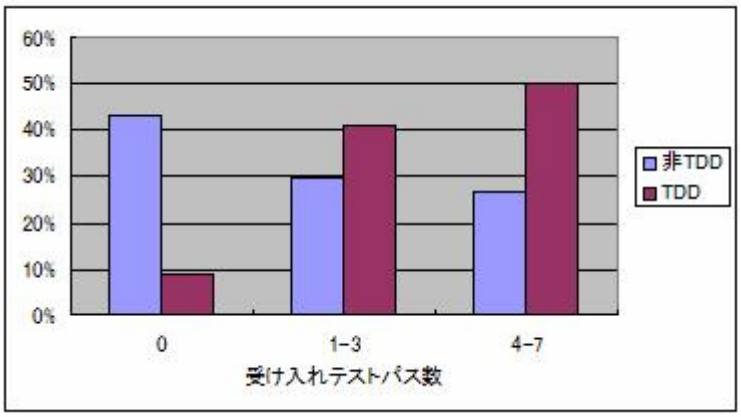
北野：「やってみて思ったのですが，かなり面倒な感じもうけました．いつもこうやって，フェイクしては正しくする，という手順を踏むのですか？」
平鍋：「場合によるね．例えば，一行で実装できてしまうメソッドをフェイクするか，というと，してもいいし，しなくてもいい．直接正解と思われるコードを書いてしまうこともある．これを，“ 明白実装 ”（Obvious Implementation）といいます．自分に自信がある場合，テストを作ってすぐに明白実装をする，という赤緑赤緑の２拍子で進んでもいいです．しかし、自信がない場面では，フェイクして，トライアンギュレーションして，そしてリファクタリングして正解と思われる実装に持っていった方がよい．この切り替えは，ペアの自信と力量，それにコンディションによります．少し寝不足の時などは，小刻みにフェイクした方が，リズムが取れて良いときがありますよ．
平鍋：「もう一度，３つのやり方をまとめてみます．例えば，int add(int a, int b) というメソッドを書きたいとしましょう．そして正解を，return a + b; とします．このとき，３つの方法があります．最初に，assertEquals(3, add(1, 2));というテストを書いて，赤にします．ここからが３つのやり方があります．

１．フェイク⇒リファクタリング
　まず，return 3; と実装．緑にしてから，return a + b;
２．フェイク⇒トライアンギュレーション⇒リファクタリング
　まず，return 3; と実装．緑にしてから，assertTrue(4, add(3, 1)); というテストを書く．そして，赤にし，最後に２つのテストを通る実装として，return a + b;
３．明白実装
　いきなり，return a+b; とします．

図にまとめるとこんな感じだね。」



平鍋：「もう１つ，ここでデータを紹介しよう．TDD をやった場合とやらなかった場合のコードの品質について，こんなデータがある．」
平鍋：「このグラフをみてください。」



平鍋：「これは、TDD と通常の開発方法との比較を行った時のデータなんだ。Laurie Williams と Body George が 2002 年ノースカロライナ州立大学で行った対照実験で、学生さん 150 人にペアを組んでもらって、ボーリングのスコアー計算を７５分間という短い時間で開発してもらったんだ。」
北野：「どうみればいいんですかね？」
平鍋：「このグラフはその実験で７つの受け入れテストを用意していて、その受け入れテストのパス数を横軸、その割合を縦軸に取っているんだ。」
北野：「圧倒的に TDD で開発した方が、テストパス数が多いチームの割合が多いですね。」
平鍋：「そうだね。これで TDD による利益がわかったかな？」

平鍋：「さて、そろそろ帰らないと。。。」
北野：「そうですね。お疲れ様でした。」
平鍋：「お疲れ様」