テスト駆動開発（TDD）とは何か。コードで実践方法を解説します

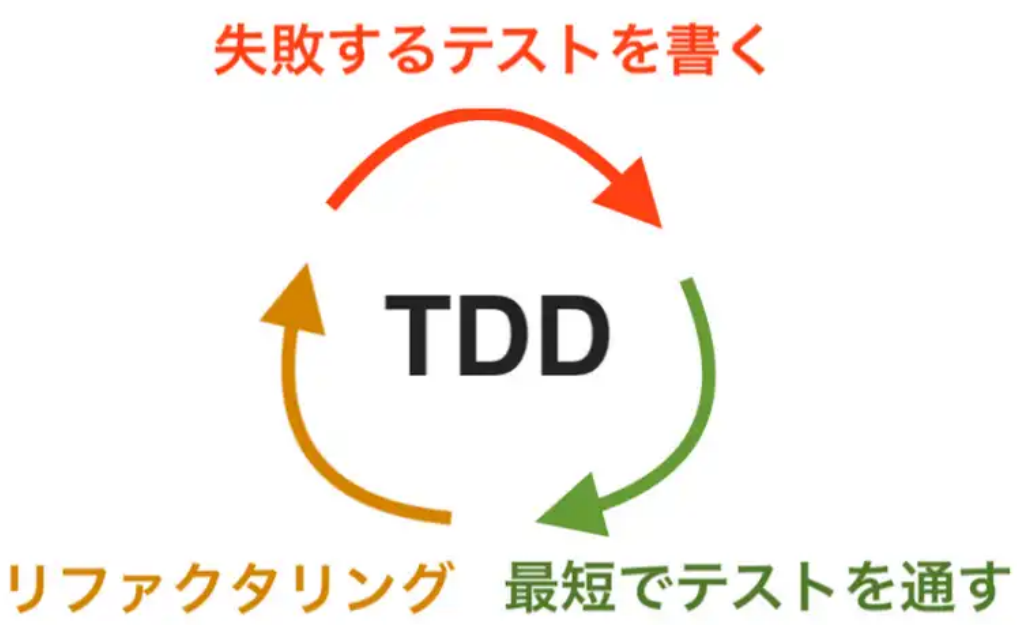
更新日: 2020年07月19日　by @Panda_Program

目次

この記事は、弁護士ドットコム Advent Calendar 2019 - Qiitaの2日目の記事です。

TL;DR

- TDDの実践方法を実際にコードを書いて解説します
- TDDの「レッド・グリーン・リファクタリング」のリズムを学ぼう
- 何度もテストを実行して、プログラムに対する不安を取り除こう



TDDはテスト技法ではなく設計手法

TDD Boot Camp Sendai 9thに参加しました。TDDの伝道師和田さん（@t_wada）を講師に迎え、有志で開かれた勉強会でした。

解説パンダくん

和田さんは「テスト駆動開発」（TDD本。Kent Beck著）の翻訳者だよ。ちなみに、Kent Beckはアジャイルの源流となるエクストリーム・プログラミングの提唱者で、今はFacebookで働いているよ。

午前中は和田さんによるTDDに関する講演とライブコーディング。午後は参加者同士のペアプロで出題されたお題を実装していく活気あるイベントでした。

イベントを通じてTDDはテストファーストのことだと考えていた自分は目を見開かされました。**TDDは単にテストファーストでプログラムを実装することではなく、実装（ソフトウェア）が期待しない動作をすることに対する不安を取り除くための一連の手法だったのです。**

TDD本にもこのように書かれています。

> **皮肉なことに、*TDD*はテスト技法ではない。*TDD*は分析技法であり、設計技法であり、実際には開発のすべてのアクティビティを構造化する技法なのだ。『テスト駆動開発』*p.278*（著 *Kent Beck*, 訳 和田卓人）**

最初は単にテストファーストだと思っていた自分は、TDDの基本動作がとても洗練されていることに感銘を受けました。

そこでプログラミングコンテストAtCoderのABC（AtCoder Beginner Contest）の問題を題材にして、TDDの手法を紹介していきます。

TDD実践の流れ

そもそもTDDは、**「動作する綺麗なコード」**を書くことを目標にして、以下のステップで実装を進めていく手法です。

1. 目標をTODOリストとして書き出す
2. TODOリストから一つピックアップし、テストを書く
3. テストコードを実行して失敗させる（レッド）
4. 実装コードを書く
5. できる限り最短でテストが通るコードを実装する（グリーン）
6. コードの重複を除去する（リファクタリング）
7. 次のTODOを選び、2に進む

この一連のステップを進めていくと、以下のルールを自然と守ることができます。

- テストが落ちているときにのみ実装コードを追加する

- 実装コードは、テストが通っている時にのみ書き換える

これにより、ソフトウェアは開発者がテストという形で意図した動作をするようになり、開発者は「ソフトウェアが期待していない挙動をすることに対する不安」から解放されることになります。

TDDの題材にAtCoderを選んだ理由

TDDを試すに当たり、AtCoderの問題を解くことが適していると考える理由は、以下の3点です。

- 問題文がそのまま仕様であること
- 入力と出力が明確であること
- 入出力のパターンが複数例用意されていること

AtCoderにおいて、競技プログラミング初心者向けのABCというコンテストでは難易度順にA問題からF問題の6問が出題されます。

ここでは、難しいアルゴリズムを使わなくても一般的なエンジニアであれば回答ができるB問題を解いていきます。

以前構築したAtCoderをPHPで解くためのレポジトリを使用します（この環境の準備過程は「競技プログラミングAtCoderを快適に解くための環境構築をする」に記載しています）。

問題（仕様）を読み解く

ABC 第141回 B - Tap Danceの問題文は以下の通りです。

> 高橋君はタップダンスをすることにしました。タップダンスの動きは文字列 *S* で表され、*S* の各文字は *L, R, U, D* のいずれかです。 各文字は足を置く位置を表しており、*1* 文字目から順番に踏んでいきます。 *S* が以下の *2* 条件を満たすとき、またその時に限り、*S* を「踏みやすい」文字列といいます。 ・ 奇数文字目がすべて *R, U, D* のいずれか。 ・ 偶数文字目がすべて *L, U, D* のいずれか。 *S* が「踏みやすい」文字列なら *Yes* を、そうでなければ *No* を出力してください。

この問題の制約は以下の通りです。

> *S* は長さ *1* 以上 *100* 以下の文字列 *S* の各文字は *L, R, U, D* のいずれか

つまり、この問題文は以下の仕様に読み換えることができます。

「1から100まで長さの、L, R, U, Dがランダムに並んだ文字列が入力される。この時、奇数文字目がR・U・Dのいずれかであり、かつ偶数文字目がL、U、Dのいずれかであればyesを出力する。そうでなければnoを出力する」

例えば、入力値がRUDLUDRである場合、偶数文字目、機数文字目が「踏みやすい」という条件を満たすので、出力はyesになります。

TODOリストを作成する

TDDはここから始まります。まずTODOリストを作成していきましょう。

1から100まで長さの、**L,R,U,Dがランダムに並んだ文字列**が入力される時、入力された文字列について、以下のように仕様を分解してTODOリストを作成します。

ただし、今回はyesかnoを出力するのではなく、yesかnoを返却するという実装をしていきます。

- ☐ L,R,U,Dのいずれかの文字を渡した時、yesまたはnoを返す
    - ☐ 文字Rを渡した時、yesを返す
    - ☐ 文字Uを渡した時、yesを返す
    - ☐ 文字Dを渡した時、yesを返す
    - ☐ 文字Lを渡した時、noを返す

また、「奇数文字目がR・U・Dのいずれかであり、かつ偶数文字目がL、U、Dのいずれかであればyesを返す」という条件を以下のように読み替えましょう。

「奇数文字目の少なくとも1つはL、または偶数文字目の少なくとも1つがRであれば、noを返すことができる」

このため、TODOリストに、下記の項目を追加します。

- ☐ 奇数文字目の少なくとも1つがLである場合、noを返す
    - ☐ 文字列LRRを渡した時、noを返す
- ☐ 偶数文字目の少なくとも1つがRである場合、noを返す
    - ☐ 文字列LRRを渡した時、noを返す

早速実装していきましょう。

1周目 - 仮実装

まずは簡単に実装できる「Rを入力すると、Yesを返す」という項目を選びます。

レッド

テストを記述します。文字列Rを入力して文字列Yesを返すため、「Sampleクラスのsolveメソッドの実引数に文字Rを渡すと文字列Yesを返す」テストを記述します。

SampleTest.php

```
class SampleTest extends TestCase
{
    /**
     * @test
     */
    public function 文字Rを渡した時、文字列Yesを返す()
    {
        $sample = new Sample();
        $result = $sample->solve('R');
        $this->assertSame('Yes', $result);
    }
}
```

（これ以降のテストメソッドでは@testアノテーションを省略します）

ここからがTDDの真髄です。この時点でテストを実行して、エラーメッセージを確認します。初めてのレッドです。

```
1) SampleTest::testB
Error: Class 'AtCoder\Sample' not found
```

実は、Sampleクラスとsolveメソッドのどちらも実装していません。

なぜなら、「テストがレッドになって初めて実装コードを追加する」というルールに従っているからです。

テストが落ちることを確認したので、これをグリーンにするための実装コードを書いていきましょう。

グリーン

まずは上記のエラーを解消するためにAtCoder\Sampleクラスを作成します。

Sample.php

```
namespace AtCoder;

class Sample
{
}
```

またテストを実行します。まだテストが落ちることは想像できますね。エラーメッセージは以下です。

```
1) SampleTest::testB
Error: Call to undefined method AtCoder\Sample::solve()
```

次はsolveメソッドを実装してテストを実行します。

```
class Sample
{
    public function solve(string $input): string
    {
    }
}
```

```
1) SampleTest::test
TypeError: Return value of AtCoder\Sample::solve() must be of the type string, none re
```

返り値はStringであるということなので、空の文字列を返すようにしましょう。

```
public function solve(string $input): string
{
    return '';
}
```

これでもまだテストは落ちることが想定できますね。「今はテストが落ちる」という感覚が重要なんです。

テストに慣れると、アプリケーションの動きに対する開発者の予想と結果が一致するようになります。

つまり、**「ああすればレッドになり、こうすればグリーンになる」という感覚を得ることができます。結果、アプリケーションが想定外の動きをすることに対する不安が減少していくのです。**

テストを実行してみましょう。

```
1) SampleTest::文字Rを渡した時、文字列Yesを返す
Failed asserting that two strings are identical.
--- Expected
+++ Actual
@@ @@
-'Yes'
+''
```

案の定テストは落ちています。しかし、テストのエラーメッセージが、テストが通る実装方法を教えてくれています。

solveメソッドは文字列のYesを返すようにしましょう。

```
PHPUnit 8.0.0 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 17 ms, Memory: 4.00MB

OK (1 test, 1 assertion)
```

テストが通りました。初めてのグリーンです。

ただ文字列を返す渡すだけでいいのか、という疑問について

ただ、実装コードでは引数も使ってないし、文字列Yes返すだけです。

本当にこれでいいのかと疑問に思われることでしょう。しかし、アサーションで期待している結果を返すためにベタ書きでもいいのでレッドをグリーンにすることは、TDDにおける立派なテクニックなのです。

このテクニックには**「仮実装」**という名前がついています。

TDDでは**「動作する綺麗なコード」**を書くことが目標です。

仮実装により、最短で「動作するコードを書く」ことができました。

テストがグリーンになったので、次のステップに進みましょう。

リファクタリング

リファクタリングのステップは、「テストがグリーンである間はプログラムの挙動が変わっていない内部実装をどのように書き換えても良い」というスタンスを取ります。

また、リファクタリングの途中でテストがレッドになったら変更箇所を元に戻します。

これにより、開発者はリファクタリングによってプログラムを壊してしまう不安から解放されます。

では、実際にリファクタリングをしていきましょう。

SampleTest.php

```
public function 文字Rを渡した時、文字列Yesを返す()
{
    $sample = new Sample();
    $result = $sample->solve('R');
    $this->assertSame('Yes', $result);
}
```

Sample.php

```
public function solve(string $input): string
{
    return 'Yes';
}
```

ただ、この段階ではあまり書き換えるところが見当たりませんね。これは好みによりますが、強いて言えばテストコードの行数を減らすことくらいでしょうか。

SampleTest.php

```
public function 文字Rを渡した時、文字列Yesを返す()
{
    $sample = new Sample();
    $this->assertSame('Yes', $sample->solve('R'));
}
```

コードを書き換えたのでテストを実行します。

```
PHPUnit 8.0.0 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 17 ms, Memory: 4.00MB

OK (1 test, 1 assertion)
```

テストがグリーンであることを確認して、先に進みます。

TODOリストの見直し

完了した項目にチェックを入れましょう。

- [ ] L,R,U,Dのいずれかの文字を渡した時、YesまたはNoを返す
    - [x] 文字Rを渡した時、Yesを返す
    - [ ] 文字Uを渡した時、Yesを返す
    - [ ] 文字Dを渡した時、Yesを返す
    - [ ] 文字Lを渡した時、Noを返す
- [ ] 奇数文字目の少なくとも1つがLである場合、Noを返す
    - [ ] 文字列UULを渡した時、Noを返す
- [ ] 偶数文字目の少なくとも1つがRである場合、Noを返す
    - [ ] 文字列URUを渡した時、Noを返す

このタイミングでTODOリストを見直して、項目の追加や削除、変更をします。

今回はこのまま次に移りましょう。

2周目 - 明白な実装

次は「文字Lを渡した時、Noを返す」という項目を選びます。

レッド

SampleTest.php

```
public function 文字Lを渡した時、Noを返す()
{
    $sample = new Sample();
    $this->assertSame('No', $sample->solve('L'));
}
```

テストを実行して、レッドであることを確認しましょう。

```
1) SampleTest::文字Lを渡した時、Noを返す
Failed asserting that two strings are identical.
```

```
--- Expected
+++ Actual
@@ @@
-'No'
+'Yes'
```

テストをグリーンにするためのコードを実装していきます。

グリーン

Sample.php

```
public function solve(string $input): string
{
    if ($input === 'L') {
        return 'No';
    }
    return 'Yes';
}
```

テストが通りました。

実装方法が明らかに頭の中にある場合は、仮実装をせずに直接実装します。

これをTDDでは**「明白な実装」**と呼んでいます。

```
OK (2 tests, 2 assertions)
```

リファクタリング

テストコードを見てみましょう。

SampleTest.php

```
public function 文字Rを渡した時、文字列Yesを返す()
{
    $sample = new Sample();
    $this->assertSame('Yes', $sample->solve('R'));
}

public function 文字Lを渡した時、Noを返す()
{
    $sample = new Sample();
    $this->assertSame('No', $sample->solve('L'));
}
```

このリファクタリングでは、重複を排除します。Sampleオブジェクトを生成している箇所を、setUpメソッドにまとめましょう。

SampleTest.php

```
class SampleTest extends TestCase
{
    private $sample;
    protected function setUp()
    {
        $this->sample = new Sample();
    }

    // ...
}
```

リファクタリングの基本は一歩ずつです。まだ$sampleプロパティを消さずに、テストを実行しましょう。

```
OK (2 tests, 2 assertions)
```

テストが壊れていないことを確認してから$sampleプロパティを使用します。

SampleTest.php

```
public function 文字Rを渡した時、文字列Yesを返す()
{
    $sample = new Sample();
    $this->assertSame('Yes', $this->sample->solve('R'));
}

public function 文字Lを渡した時、Noを返す()
{
    $sample = new Sample();
    $this->assertSame('No', $this->sample->solve('L'));
}
```

変数$sampleはもう使われていませんが、ここでまた再度テストを実行します。

```
OK (2 tests, 2 assertions)
```

グリーンであることを確認したら、変数$sampleを削除してテストを実行します。

SampleTest.php

```
    public function 文字Rを渡した時、文字列Yesを返す()
    {
        $this->assertSame('Yes', $this->sample->solve('R'));
    }

    public function 文字Lを渡した時、Noを返す()
    {
        $this->assertSame('No', $this->sample->solve('L'));
    }


    OK (2 tests, 2 assertions)
```

このリファクタリングでは何も壊していないことを確認できました。

他にリファクタリングをする箇所はないので、先に進みます。

TODOリストの見直し

完了した項目にチェックを入れましょう。

また、今の実装では文字U・Dを渡してもYesを返します。問題の条件より「Noを返す場合以外はYesを返す」と読み替えることができるので、TODOの項目を2つ削除してもいいと判断します。

- [x] L,R,U,Dのいずれかの文字を渡した時、YesまたはNoを返す
    - [x] 文字Rを渡した時、Yesを返す
    - [ ] ~~文字Uを渡した時、Yesを返す~~
    - [ ] ~~文字Dを渡した時、Yesを返す~~
    - [x] 文字Lを渡した時、Noを返す
- [ ] 奇数文字目の少なくとも1つがLである場合、Noを返す
    - [ ] 文字列UULを渡した時、Noを返す
- [ ] 偶数文字目の少なくとも1つがRである場合、Noを返す
    - [ ] 文字列URUを渡した時、Noを返す

この記事の構成を練っている段階では、U、Dのテストケースも必要だと考えていました。

しかし、コードを実装しているうちに、今回このケースは不要だと思ったのでリストから削除します。

このように、TODOリストを絶対的なものではなく、状況に応じて柔軟に変更を加えるものとして扱いましょう。

3周目

次は「文字列UULを渡した時、Noを返す」という項目を選びます。

レッド

テストコードを書きます。

```
public function 文字列UULを渡した時、Noを返す()
{
    $this->assertSame('No', $this->sample->solve('UUL'));
}
```

先ほどリファクタリングをしたおかげで、とてもシンプルなコードになりました。

テストを実行して、レッドであることを確認します。

```
1) SampleTest::文字列UULを渡した時、Noを返す
Failed asserting that two strings are identical.
--- Expected
+++ Actual
@@ @@
-'No'
+'Yes'
```

グリーン

現状の実装コードを見てみましょう。

```
public function solve(string $input): string
{
    if ($input === 'L') {
        return 'No';
    }
    return 'Yes';
}
```

さて、困りましたね。ここでもう一度条件と現状を確認しましょう。

「奇数文字目がLならNoを返す」という一般的な条件は、「1文字がLならNoを返す」という条件を包含しています。

現状では後者は実装済みですが、前者は未実装です。

今回のTODOリストの項目、「文字列UULを渡すとき、Noを返す」の文字列UULは3文字目がLであることを利用して、一般化した実装をしていきましょう。

Sample.php

```
public function solve(string $input): string
{
    $len = strlen($input);
    // インデックスを2ずつインクリメントすることで奇数文字目を走査する
    for ($i = 0; $i < $len; $i += 2) {
        if ($input[$i] === 'L') {
            return 'No';
        }
    }

    if ($input === 'L') {
        return 'No';
    }
    return 'Yes';
}
```

テストを実行してみます。

```
PHPUnit 8.0.0 by Sebastian Bergmann and contributors.

...                                                                 3 / 3 (100%)

Time: 34 ms, Memory: 4.00MB

OK (3 tests, 3 assertions)
```

無事に通りました。

三角測量

なお、より一般的な条件を満たすコードを実装するためにテストケースを増やすというテクニックをTDDでは**「三角測量」**と呼びます。

1文字目がLである文字Lを入力するケースと、3文字目がLであるUULを入力するケースを合わせて三角測量をすることで、一般化の方向が定まります。

ただし、三角測量は毎回使うテクニックではありません。実装方法が明らかではない場合に、歩幅を狭めて自分の立ち位置を確認しながら進むために使います。

実際にTDD本では、著者が三角測量を使うときは一般的な実装が本当にわからないときだけと書かれています。実装方法がわかっている場合は、明白な実装で構いません。

リファクタリング

今回は、不要になったコードを削除します。

Sample.php

```
public function solve(string $input): string
{
    $len = strlen($input);

    for ($i = 0; $i < $len; $i += 2) {
        if ($input[$i] === 'L') {
            return 'No';
        }
    }

    return 'Yes';
}
```

テストを実行して、挙動が変わっていないこと、また以前に作成したのテストも通っていることを確認します。

```
OK (3 tests, 3 assertions)
```

TODOリストの見直し

現在のTODOリストは以下の通りです。

- ☑ L,R,U,Dのいずれかの文字を渡した時、YesまたはNoを返す
    - ☑ 文字Rを渡した時、Yesを返す
    - ☐ ~~文字Uを渡した時、Yesを返す~~
    - ☐ ~~文字Dを渡した時、Yesを返す~~
    - ☑ 文字Lを渡した時、Noを返す
- ☑ 奇数文字目の少なくとも1つがLである場合、Noを返す

- [x] 文字列UULを渡した時、Noを返す

- [ ] 偶数文字目の少なくとも1つがRである場合、Noを返す

  - [ ] 文字列URUを渡した時、Noを返す

ここで1文字目と3文字目をLにしたからといって、99文字目がLの時にNoを返すかはわかりません。「奇数文字目の少なくとも1つがLである場合、Noを返す」ことを満たしていると言えないとも考えられます。

そのときは、TODOリストに項目を追加して、99文字目がLになるようなテストを追加しましょう。

テストケースを追加するにつれ、プログラムは堅牢さを獲得し、その動きは開発者が予測できるものになります。

どこまでテストを書けばいいのか、という疑問にTDD本の著者Kent Beckは「不安がなくなるまで」と答えています。

自分がプログラムに対して不安であれば、テストを追加しましょう。

4周目 - 最後のTODO

いよいよ最後の項目「文字列URUを渡した時、Noを返す」です。

レッド

まずはテストコードを追加して、テストを実行します。

```
public function 文字列URUを渡した時、Noを返す()
{
    $this->assertSame('No', $this->sample->solve('URU'));
}
```

```
1) SampleTest::文字列URUを渡した時、Noを返す
Failed asserting that two strings are identical.
--- Expected
+++ Actual
@@ @@
-'No'
+'Yes'
```

グリーン

テストがレッドなので、コードを追加します。奇数文字の時と同様に考えましょう。

Sample.php

```
public function solve(string $input): string
{
    $len = strlen($input);

    for ($i = 0; $i < $len; $i += 2) {
        if ($input[$i] === 'L') {
            return 'No';
        }
    }

    // 偶数文字目を調べるので、インデックスの初期値は1
    for ($i = 1; $i < $len; $i += 2) {
        if ($input[$i] === 'R') {
            return 'No';
        }
    }

    return 'Yes';
}
```

```
PHPUnit 8.0.0 by Sebastian Bergmann and contributors.

....                                                                4 / 4 (100%)

Time: 18 ms, Memory: 4.00MB

OK (4 tests, 4 assertions)
```

テストがグリーンになりました。

リファクタリング

この実装コードでも問題文の条件を満たします。ただ、ループを2回も回していることが気になりますね。

ループの回数を減らすようにリファクタリングをしてみましょう。

```
public function solve(string $input): string
{
    $len = strlen($input);

    for ($i = 0; $i < $len; $i++) {
        if ($i % 2 === 0 && $input[$i] === 'L') {
            return 'No';
```

```
            }
        }

        for ($i = 1; $i < $len; $i += 2) {
            if ($input[$i] === 'R') {
                return 'No';
            }
        }

        return 'Yes';
    }
```

```
OK (4 tests, 4 assertions)
```

テストを実行してもグリーンです。リファクタリングを続けます。

```
    public function solve(string $input): string
    {
        $len = strlen($input);

        for ($i = 0; $i < $len; $i++) {
            if ($i % 2 === 0 && $input[$i] === 'L') {
                return 'No';
            }
            if ($i % 2 === 1 && $input[$i] === 'R') {
                return 'No';
            }
        }

        return 'Yes';
    }
```

```
OK (4 tests, 4 assertions)
```

テストは通っています。ただ、コードを書き換えたことにより、`$input[$i]`に重複が発生しているので、これを修正します。

```
    public function solve(string $input): string
    {
        $len = strlen($input);

        for ($i = 0; $i < $len; $i++) {
            $currentChar = $input[$i];
            if ($i % 2 === 0 && $currentChar === 'L') {
                return 'No';
```

```
            }
            if ($i % 2 === 1 && $currentChar === 'R') {
                return 'No';
            }
        }

        return 'Yes';
    }


    OK (4 tests, 4 assertions)
```

テストが通っていることを確認して、条件文の重複を取り除きましょう。

```
    public function solve(string $input): string
    {
        $len = strlen($input);

        for ($i = 0; $i < $len; $i++) {
            $currentChar = $input[$i];
            if ($i % 2 === 0 && $currentChar === 'L' ||
                $i % 2 === 1 && $currentChar === 'R'
            ) {
                return 'No';
            }
        }

        return 'Yes';
    }
```

今回もテストが通っているので、変更によってプログラムが壊れていないことを確認できました。

```
    OK (4 tests, 4 assertions)
```

今回はこれで完成とします。AtCoderでは解答を作成する際、必ずしもクラスや関数を作る必要はないからです。

さらなるリファクタリング例

プロダクションコードのレベルであれば、下記のように書き換えてもいいでしょう。

```
    class Sample
    {
        public function solve(string $input): string
```

```
        {
            $len = strlen($input);

            for ($i = 0; $i < $len; $i++) {
                $currentChar = $input[$i];

                if ($this->isOddCharL($i, $currentChar) ||
                    $this->isEvenCharR($i, $currentChar)
                ) {
                    return 'No';
                }
            }

            return 'Yes';
        }

        private function isOddCharL($index, $currentChar): bool
        {
            return $index % 2 === 0 && $currentChar === 'L';
        }

        private function isEvenCharR($index, $currentChar): bool
        {
            return $index % 2 === 1 && $currentChar === 'R';
        }
    }
```

もちろんテストは通っています。

```
OK (4 tests, 4 assertions)
```

TODOリストの見直し

今回の項目にチェックを入れましょう。

- [x] L,R,U,Dのいずれかの文字を渡した時、YesまたはNoを返す
    - [x] 文字Rを渡した時、Yesを返す
    - [ ] ~~文字Uを渡した時、Yesを返す~~
    - [ ] ~~文字Dを渡した時、Yesを返す~~
    - [x] 文字Lを渡した時、Noを返す
- [x] 奇数文字目の少なくとも1つがLである場合、Noを返す

- ☑ 文字列UULを渡した時、Noを返す

- ☑ 偶数文字目の少なくとも1つがRである場合、Noを返す
  - ☑ 文字列URUを渡した時、Noを返す

これで全てのTODOリストが完了しました。

提出。 - Accepted

AtCoderでは入力は標準入力から得られ、出力は標準出力で行うため、それに合わせて書き換えます。

```
<?php
$input = trim(fgets(STDIN));
$len = strlen($input);

for ($i = 0; $i < $len; $i++) {
  $currentChar = $input[$i];
  if ($i % 2 === 0 && $currentChar === 'L' ||
      $i % 2 === 1 && $currentChar === 'R'
     ) {
    echo 'No';
    return;
  }
}

echo 'Yes';
```

プログラムが完成したので、問題を提出してみましょう。

AtCoderで提出した問題がACになりました。やりましたね。

まとめ

TDDはテストの技法ではなく、設計・分析技法だとTDD本の著者Kent Beckは語っています。

また、ソフトウェアが開発者の意図しない挙動をするという不安を軽減するための手段でもあります。

もちろんTDDを実践したから、ソフトウェアが絶対にバグを起こさないとは言えません。

しかし、TDDのステップの中には、プログラムの挙動を変えず内部実装を書き換えるリファクタリングが含まれています。

これにより、読みやすく、メンテナンスコストの低いコードを書くことができるんです。

このリファクタリングを安心して行うことができるのは、テストを頻繁に実行することでソフトウェアと開発者の現在地を確認できるからです。

これが、TDDの目的である**「動作する綺麗なコード」**を生み出すことに繋がります。

TDDはその実践方法が十分に体系化されており、誰でも明日から現場で実践できるものなのです。

ぜひTDDを実践して、堅牢なソフトウェアを構築しましょう。

テストのデメリットについて

上でまとめを書いておいて何ですが、TDDのデメリットとして挙げられることがいくつかあります。

ここではいくつかピックアップして、簡単に自分が考えていることを記述していきます。

> 「テストを書いていると実装スピードが遅くなる」

バグを混入している可能性があるコードを本番リリースする方が自分は不安です。

リリースまでのスピードは幾分落ちますが、本番リリース後にバグが出た時の障害対応に割く時間を考えると、長期的に見て時間は節約できると自分は考えています。

また、TDDでは初期実装は確かにスピードダウンするものの、メンテナンスコストが下がるため、保守・運用期間の工数が減少するという研究があります。

> 「テストが増えるとテスト実行時間が遅くなるからTDDはダメだ」

ローカルでの実行なら、@groupアノテーションをつけて、自分が開発している箇所のみテストを実施しましょう。

CI環境なら、テストを並列実行することでテストの実行時間を短縮できます（例：Laravelで並列テストを導入するための道のり）。

> 「テストのメンテナンスが大変」

実はテストにもメンテナンスコストはあるのですよね。

メソッド名が分かりづらかったり、別のテストに依存するテストを書いていたり、ランダムで落ちるテストがあるとはっきり言って大変です。

TDDではリファクタリングのステップでテストに対する見直しをできるので、その段階で異変に気付いておきたいです。

まずはテストにもメンテナンスコストが発生するということを認識することが第一歩です。

なお上記の批判は、TODOリストを作成してレッド・グリーン・リファクタリングのリズムでプログラムを実装するというTDDの手法の批判ではありません。TDDという手法ではなく、テストコード一般に当てはまるものです。

TDD云々以前に、まずはテストコードに対する理解を深めることが先なのでしょう。

プロのプログラマはTDDを実践している

上記の批判を額面通り受け取り、テストコードを書かないことはデメリットが大きいのではないでしょうか。

それは「早く実装できるが汚いコードを書いた上に、バグをユーザーに届けてしまう」ことにつながるからです。

クリーンアーキテクチャで有名なボブおじさんも著書「Clean Coder」の中で、ソフトウェアのプロとして備えるべき最低限のことの一つにTDDを挙げています。

ボブおじさんはテストを書くのが面倒だと思った時には、左腕につけているグリーンバンド（テスト駆動開発者の証）を見て、自分はプロのプログラマだからテストを書くのだと自分を奮い立たせるそうです。

ソフトウェアのバグ混入率を低下させ、ソフトウェアが安定して動作する綺麗なコードを安心して書き続けたいのであれば、TDDは必ずあなたの力になるでしょう。

Happy Coding 🎉

TDDに関する記事

› useContext + useReducer の使いどころ

› Elmを通してFluxを理解する

› 競技プログラミングAtCoderを快適に解くためのPHPの環境を構築する

› 二分ヒープをPHPで手軽に扱う

TDD 設計 AtCoder

Topに戻る

プログラミングをするパンダ

Twitter: @Panda_Program

Next.js（React）、TypeScriptが好き。OOP、TDDとペア・モブプロでテストを書きつつ、クリーンな設計のコードを目指す。PHPも書きます。弁護士ドットコム → BASE