テストコードの注入から始める

レガシーコードのリファクタリング

（サンプル版）

風間 裕也 著

テストコードの注入から始めるレガシーコードのリファクタリング（サンプル版）

風間 裕也　著

2020-12-08 版　　**crabink** 発行

はじめに

本書籍を手にとっていただきありがとうございます。

本書籍では、レガシーコードに対しての最初の一歩を踏み出したいと考えている人に向けて書いた本になります。

想定読者

本書籍では下記のような人を想定読者としています。

- テスト駆動開発（以下、TDD）を知っている
- レガシーコードやリファクタリングとは何か知っている
- 実際の現場でレガシーコードと出会っている
- レガシーコードに対してリファクタリングを行う時に、まず何から手を付ければ良いのか分からない
- 今後、自分がレガシーコードを作らないように心がけたい

TDD についてよく分からないという方は、書籍『テスト駆動開発』 [*1] を読むことをおすすめします。また、和田卓人さんによるテスト駆動開発の動画 [*2] や、TDDBC [*3] への参加もおすすめです。

「レガシーコード」や「リファクタリング」という言葉が分からないという方は、書籍『レガシーコード改善ガイド』 [*4] や書籍『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』 [*5] [*6] を読むことをおすすめします。

[*1] 書籍『テスト駆動開発』 https://www.amazon.co.jp/dp/B077D2L69C/

[*2] TDD Boot Camp 2020 Online #1 基調講演/ライブコーディング https://www.youtube.com/watch?v=Q-FJ3XmFlT8

[*3] TDD Boot Camp(TDDBC) http://devtesting.jp/tddbc/

[*4] 書籍『レガシーコード改善ガイド』 https://www.amazon.co.jp/dp/B01AN97W08

[*5] 書籍『リファクタリング　既存のコードを安全に改善する』 https://www.amazon.co.jp/dp/B01IGW5MG0/

[*6] 書籍『リファクタリング　既存のコードを安全に改善する（第 2 版）』

本書籍によって身につく内容

本書籍を読み、写経することで、下記の 2 点が身につきます。

- 少しずつテストケースを追加して改善していく方法を実感する
- 苦しくないリファクタリングの方法を学ぶ

少しずつテストケースを追加して改善していく方法を実感する

書籍『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』[*7] の第 1 章「リファクタリング-最初の例」の中の節「リファクタリングの第一歩」には下記のように書かれています。

> リファクタリングを開始するとき、最初にすることは常に同じです。対象となるコードについてきちんとしたテスト群を作り上げることです。リファクタリングは非常に秩序だっていて、新たなバグを生み出しにくくなっていますが、人間が作業する以上、間違いを犯す可能性があります。このためテストは大切で、きちっとした一連のテストを用意するべきなのです。

このようにリファクタリングにはテストが大切と書いている一方で、実はこの書籍の第 1 章で扱っている題材については、テストコードが一切記載されていません。そのため、引用した文章内の「きちんとしたテスト群」というのはどんなものなのか想像しづらいと感じています。特に、レガシーコードに対して「きちんとしたテスト群」を書くというのはどういうことなのでしょうか。

自分が最近書いたコードに対してのテストであれば、ある程度考えがまとまっているため、「きちんとしたテスト群」は書けるでしょう。しかし、別の人や数年前の自分が書いたレガシーコードに対してのテストの場合、そもそもどんな振る舞いをしている実装コードなのか把握するところから大変になり、「きちんとしたテスト群」を書く難易度が格段に上がります。

そこで本書籍では、そのようなレガシーコードに対して、「徐々にテストコードを追加

https://www.amazon.co.jp/dp/B0827R4BDW/

[*7] 書籍『リファクタリング　既存のコードを安全に改善する』 https://www.amazon.co.jp/dp/B01IGW5MG0/

する」ことで「少しずつ振る舞いを理解していく」ことによりリファクタリングを行う方法を紹介していきます。

苦しくないリファクタリングの方法を学ぶ

書籍『Clean Code』[*8]の執筆にも関わった Tim Ottinger のブログ記事[*9][*10]では、TDD およびリファクタリングの誤解を色々と書いています。文章のどの部分も示唆に富む素晴らしい内容ですが、その中から抜粋して引用します。

> ・TDD ループを使い始めたからといって自動的にリファクタリングの使い手になるわけではありませんが、TDD プロセスは、これらのスキルを学ぶ機会を提供します。
> ・悪いテストはリファクタリングを妨げます。（中略）悪いテストがあることは、テストがないことよりも悪い場合があります。

TDD を行うにはテストが必要であり、リファクタリングを行うときも前提としてテストが必要です。しかし、引用した文章にも書いている通り、悪いテストは逆にリファクタリングをしづらくしてしまいます。

たまに、「すごい大掛かりな実装コードだったけど、すべてを通るようなテストがあって良かった。」という話を聞いたりします。ですがこの場合は、そもそも「リファクタリングしづらいような悪いテストになっていないか」ということを確認した方が良いでしょう。

本書籍では、論理的かつ構造的にテストを組み上げていくことにより、今後、そのコードを保守するときにも苦しまないようなリファクタリングを行う方法を紹介していきます。

そもそもレガシーコードは作らないようにした方が良いのでは？

本書籍を見つけた人の中には、「そもそもレガシーコードは作らないことが大事」と主張する人がいるかもしれません。この意見ついては私も賛成です。

ただし、レガシーコードは気付かないうちに増えていくものだと考えています。先ほども紹介した Tim Ottinger のブログ記事[*11][*12]から再度抜粋して引用します。

> ・TDD の目標は、迅速なリファクタリングの環境を作り出すこと。

[*8] 書籍『Clean Code』 https://www.amazon.co.jp/dp/B078HYWY5X/
[*9] Tim Ottinger のブログ記事 https://www.industriallogic.com/blog/tdd-purposes-and-practices/
[*10] Tim Ottinger のブログ記事日本語版 https://nihonbuson.hatenadiary.jp/entry/2020/11/20/213000
[*11] Tim Ottinger のブログ記事 https://www.industriallogic.com/blog/tdd-purposes-and-practices/
[*12] Tim Ottinger のブログ記事日本語版 https://nihonbuson.hatenadiary.jp/entry/2020/11/20/213000

・TDD でアサーションが進むことについては納得しますが、優れた設計を強要することは TDD の目的ではありません。
・TDD では設計を行いませんが、設計を改善するための多くの機会を提供します。

この引用からは、TDD によって自然と綺麗なコードや設計になるわけではなく、あくまでもリファクタリングの機会を提供しているだけであることが示唆されています。

そのため意識しないと、大小の差はあれどレガシーコードは増えていきます。本書籍で学ぶことによって、増え始めのレガシーコードにも対処することができます。

本書籍で扱うレガシーコードの変更方法

書籍『レガシーコード改善ガイド』では、レガシーコードの変更方法として下記のように書かれています。

1. 変更点を洗い出す
2. テストを書く場所を見つける
3. 依存関係を排除する
4. テストを書く
5. 変更とリファクタリングを行う

本書籍では、この 5 つの方法を具体的な例を用いて紹介していきます。

本書籍の読み方

本書籍の構成

本書籍では、下記の 3 つの具体的なコードをリファクタリングしていきます。

- 第 1 章…テストしづらい部分を分割する
- 第 2 章…ロジックを分解して整理する
- 第 3 章…要件を元に責務ごとにロジックを分割して整理する

コードの複雑性が単純なものから順番に書かれていますので、第 1 章から順番に読み進めることをオススメします。

本書籍の記述方式

本書籍では Java および JUnit5 を用いて解説していきますが、基本的な考え方はどの言語でも似ていると考えていますので、読者の皆さんの言語に置き換えて読んでいただければ幸いです。

また、コードについては変更部分を中心に記載しています。例えば、下記のように記載します。

リスト 1: 記載の仕方

```
....
    public void sampleMethod(){
        ~~hoge();~~
        fuga();
    }
....
```

「....」は省略を表します。.... の前、もしくは.... の後にはそれまでに出てきたロジックが変更されずに残っていることを示しています。

取り消し線の行は今回の変更によって消去された行を示しています。太字の行は今回の変更によって追加された行を示しています。

つまり今回の例の場合、「hoge();」メソッドから「fuga();」メソッドに変更されたことを示しています。

このように変更箇所を中心に記載しているため、本書籍は写経しながら読みすすめることを強くオススメします。

IDE のショートカット紹介について

本書籍では、書籍『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』[*13] で記載されているようなリファクタリング方法をいくつか実践しています。

しかし昨今では、それらリファクタリング方法を手作業で行わず、IDE による入力補助を使うことで、手早く安全にコードを変更することができます。

そこで、付録では IDE のショートカットによるコードの変更について画像付きで紹介します。今回は IntelliJ の操作でご紹介していますが、他の IDE でも同様の入力補助が

[*13] 書籍『リファクタリング　既存のコードを安全に改善する』https://www.amazon.co.jp/dp/B01IGW5MG0/

できる場合がありますので、適宜読み替えてください。

本文中に出てくるリファクタリング内容の一部には、付録への参照が付いていますので、写経する際は付録の内容を参考にして IDE の入力補助を用いながらリファクタリングを試してみてください。

表紙について

本書籍の表紙は、雲により多くは隠れていて一部だけ建造物が見えている画像[*14]です。これは、最初はテストコードが不足してほとんど動作が確認できていない（建造物が見えていない）ところから、少しずつテストコードを追加していくことを表現しているように感じたため、表紙に採用しました。

[*14] https://www.pexels.com/photo/view-of-cityscape-325185/ より引用

目次

第 1 章

テストしづらい部分を分割する

1.1 題材

レガシーコードになりがちなコードの 1 つとして、現在時刻に依存しているなど、テストしづらい状況であるコードがあります。

例えば、以下のようなコードです。[*1]

リスト 1.1: DeliveryDate.java

```
 1: import java.time.LocalDate;
 2: import java.time.Month;
 3:
 4: public class DeliveryDate {
 5:     public LocalDate getDeliveryDate(){
 6:         LocalDate localDate = LocalDate.now();
 7:         int day = localDate.getDayOfMonth();
 8:         Month month = localDate.getMonth();
 9:         int year = localDate.getYear();
10:
11:         if(day >= 25){
12:             month.plus(1L);
13:         } else if (month.equals(Month.DECEMBER) && day >= 20) {
14:             month.plus(1L);
15:         }
16:
17:         int lastDay;
```

[*1] このコードを見て「なんだよ！ 全然複雑じゃないじゃん！ 現場のコードはもっと複雑だから、こんな内容を見ても意味がないや」と感じた人もいるかもしれません。しかし、このお題で伝えたいことは「複雑なコードに対処すること」ではなく、「テストコードがないものに対処すること」です。テストコードがない状態に対して、一歩でも先に進む方法を今回は紹介します。

```
18:         if(month.equals(Month.APRIL)) {
19:             lastDay = 30;
20:         } else if(month.equals(Month.JUNE)){
21:             lastDay = 30;
22:         } else if(month.equals(Month.SEPTEMBER)){
23:             lastDay = 30;
24:         } else if(month.equals(Month.NOVEMBER)){
25:             lastDay = 30;
26:         } else if(month.equals(Month.FEBRUARY)){
27:             if(year%4 == 0){
28:                 lastDay = 29;
29:             } else {
30:                 lastDay = 28;
31:             }
32:         } else {
33:             lastDay = 31;
34:         }
35:         return LocalDate.of(localDate.getYear(),
36:             localDate.getMonth(), lastDay);
37:     }
38: }
```

このコードは、配送日を指定するコードです。6 行目で現在日時を挿入し、その日時の月末を配送日として設定します。しかし、現在日時が下旬や年末の場合は、配送日が次月に設定されます。

本章ではこのコードに対してリファクタリングを行っていきます。

1.2 最初のテストコード

さて、このようなコードに対して、どのようにテストケースを追加していけば良いでしょうか。

最初に作るべきテストコードでは「とりあえず動くテスト」を目指します。

今回の場合は以下のようなコードをとりあえず作ってみましょう。

リスト 1.2: DeliveryDateTest.java

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class DeliveryDateTest {
```

```
    @Test
    void _配送日のテスト() {
        new DeliveryDate();
        assertEquals(1,1);
    }
}
```

ただ、DeliveryDate クラスを呼び出しただけです。

この状態でテスト実行をしてみましょう。もしもテスト実行して Red になった[*2]場合、以下の 2 つのどちらかが原因でしょう。

- テストフレームワークの設定自体が間違っている
- DeliveryDate クラスの呼び出しの際に必要な設定が足りない

このうち、他のクラスで同じテストフレームワークを用いて動いていた場合は、1 つ目の原因の可能性は限りなく低いでしょう。つまり、今回のテストコードを実行することで、そもそも実装コードを実行できるのか確認することができます。[*3]

テスト実行して Green になった場合、次はメソッドを呼び出します。

リスト 1.3: DeliveryDateTest.java

```
....
    @Test
    void _配送日のテスト() {
        new DeliveryDate();
        new DeliveryDate().getDeliveryDate();
        assertEquals(1,1);
    }
}
```

最初に書いたテストコードと同様、これも getDeliveryDate() メソッドを実行できるのか確認することができます。

このようにテストコードを書くことで、実装コードを手軽に試すことができます。しか

[*2] テスト実行した結果、成功を「Green」、失敗を「Red」と言います。本書籍では以降「Green」「Red」と表記します。詳しくは書籍『テスト駆動開発』(https://www.amazon.co.jp/dp/B077D2L69C/) を参考にしてください。

[*3] Red になったとしても、悲観する必要はありません。「設定が足りないことを教えてくれた」と、テストコードを書くことによって新情報を得られたことを喜ぶべきです。

も一度書くと、自動で何度も実行することができます。

JaSST'18 Tokyo 招待講演 [*4] で、柴田芳樹さんは下記の発言をしていますが、今回の過程を写経すると実感ができると思います。

> Unit テスト作成は自動でデバッグしている感覚
> TDD は常に実装する感覚

1.3 仕様を理解してテストを作る

次は実装コードを見ながら期待値を当てはめていきます。

リスト 1.4: DeliveryDateTest.java

```
....
    @Test
    void _配送日のテスト() {
        ~~new DeliveryDate().getDeliveryDate();~~
        LocalDate actualDate = new DeliveryDate().getDeliveryDate();
        ~~assertEquals(1,1);~~
        assertEquals(LocalDate.of(2020,9,30),actualDate);
    }
}
```

今回の場合、getDeliveryDate メソッドを呼び出すと、今日の日付を元に月末の日付などが返ってきます。今回実行した日付が 2020 年 9 月 13 日だったため、2020 年 9 月 30 日を期待値として設定しました。

1.4 別のテストケースを作る

続いて、実装コードを見て、別のテストケースを作ってみましょう。

[*4] JaSST'18 Tokyo 招待講演「私が経験したソフトウェアテストの変遷」 http://www.jasst.jp/symposium/jasst18tokyo/pdf/A7.pdf

リスト 1.5: DeliveryDateTest.java

```
....
    @Test
    void _配送日のテスト() {
    void _小の月の月末になる場合のテスト() {
        LocalDate actualDate = new DeliveryDate().getDeliveryDate();
        assertEquals(LocalDate.of(2020,9,30),actualDate);
    }

    @Test
    void _大の月の月末になる場合のテスト() {
    LocalDate actualDate = new DeliveryDate().getDeliveryDate();
    assertEquals(LocalDate.of(2020,10,31),actualDate);
    }

}
```

小の月の月末のテストケースを元々作っていたので、大の月の月末の場合のテストケースも作成しようと考えました。

しかし、両方のテストケースにおける DeliveryDate クラスの呼び出し及び getDeliveryDate メソッドの呼び出しに違いがない（パラメータの設定などを行っていない）ため、これら 2 つのテストケースを実行しようとすると、必ずどちらかのテストが Red になります。

つまり、この方法だとうまくいかないことが分かります。このような場合、どうすれば良いのでしょうか。

1.5 依存関係を見つける

今回のテストがうまく行かない理由を考えてみましょう。

リスト 1.6: DeliveryDate.java

```
 1: import java.time.LocalDate;
 2: import java.time.Month;
 3:
 4: public class DeliveryDate {
 5:     public LocalDate getDeliveryDate(){
 6:         LocalDate localDate = LocalDate.now();
 7:         int day = localDate.getDayOfMonth();
 8:         Month month = localDate.getMonth();
```

```
 9:         int year = localDate.getYear();
10:
11:         if(day >= 25){
12:             month.plus(1L);
13:         } else if (month.equals(Month.DECEMBER) && day >= 20) {
14:             month.plus(1L);
15:         }
16:
17:         int lastDay;
18:         if(month.equals(Month.APRIL)) {
19:             lastDay = 30;
20:         } else if(month.equals(Month.JUNE)){
21:             lastDay = 30;
22:         } else if(month.equals(Month.SEPTEMBER)){
23:             lastDay = 30;
24:         } else if(month.equals(Month.NOVEMBER)){
25:             lastDay = 30;
26:         } else if(month.equals(Month.FEBRUARY)){
27:             if(year%4 == 0){
28:                 lastDay = 29;
29:             } else {
30:                 lastDay = 28;
31:             }
32:         } else {
33:             lastDay = 31;
34:         }
35:         return LocalDate.of(localDate.getYear(),
36:             localDate.getMonth(), lastDay);
37:     }
38: }
```

今回、テストケースがうまく行かない最大の理由は 6 行目です。LocalDate.now() で現在の時刻を入れているため、テスト実行日時に依存してしまうのです。

この依存関係を削除する方法を考えましょう。

1.6 依存関係を削除する

（以降をご覧になりたい方はご購入をお願いします。）

第 2 章

ロジックを分解して整理する

2.1 題材

本章では、無造作にロジックが書かれているものに対してリファクタリングを行っていきます。以下のような題材です。

2.1.1 題材元

https://github.com/emilybache/Tennis-Refactoring-Kata
なお、この題材は MIT ライセンスとなっています。
https://github.com/emilybache/Tennis-Refactoring-Kata/blob/master/license.txt

2.1.2 初期コード

最初に存在しているコードは以下のようなものです。

リスト 2.1: 既存の TennisGame1.java

```
 1: public class TennisGame1 implements TennisGame {
 2:
 3:     private int m_score1 = 0;
 4:     private int m_score2 = 0;
 5:     private String player1Name;
 6:     private String player2Name;
 7:
 8:     public TennisGame1(String player1Name, String player2Name) {
 9:         this.player1Name = player1Name;
10:         this.player2Name = player2Name;
11:     }
```

```
12:
13:     public void wonPoint(String playerName) {
14:         if (playerName == "player1")
15:             m_score1 += 1;
16:         else
17:             m_score2 += 1;
18:     }
19:
20:     public String getScore() {
21:         String score = "";
22:         int tempScore=0;
23:         if (m_score1==m_score2)
24:         {
25:             switch (m_score1)
26:             {
27:                 case 0:
28:                         score = "Love-All";
29:                     break;
30:                 case 1:
31:                         score = "Fifteen-All";
32:                     break;
33:                 case 2:
34:                         score = "Thirty-All";
35:                     break;
36:                 default:
37:                         score = "Deuce";
38:                     break;
39:
40:             }
41:         }
42:         else if (m_score1>=4 || m_score2>=4)
43:         {
44:             int minusResult = m_score1-m_score2;
45:             if (minusResult==1) score ="Advantage player1";
46:             else if (minusResult ==-1) score ="Advantage player2";
47:             else if (minusResult>=2) score = "Win for player1";
48:             else score ="Win for player2";
49:         }
50:         else
51:         {
52:             for (int i=1; i<3; i++)
53:             {
54:                 if (i==1) tempScore = m_score1;
55:                 else { score+="-"; tempScore = m_score2;}
56:                 switch(tempScore)
57:                 {
58:                     case 0:
```

```
59:                         score+="Love";
60:                         break;
61:                     case 1:
62:                         score+="Fifteen";
63:                         break;
64:                     case 2:
65:                         score+="Thirty";
66:                         break;
67:                     case 3:
68:                         score+="Forty";
69:                         break;
70:                 }
71:             }
72:         }
73:         return score;
74:     }
75: }
```

2.2 とりあえずテストを実行する

2.2.1 最初のテストケースを作り、実行時エラーにならないことを確認する

まずはテストコードを作ります[*1]。これは実装コードが正しいことを担保するためではなく、とりあえず実装コードが動くかどうかの確認です。

リスト 2.2: TennisGame1Test.java

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TennisGame1Test {
    @Test
    public void _とりあえずテスト実行() {
        new TennisGame1("P1","P2").getScore();
        assertEquals(1,1);
```

[*1] 題材元には既にテストコードが存在していました。ただし「テストコードが無い状態から着手した例を示したい」「用意されているテストコードは構造的に表現されていない」という 2 つの理由により、本書籍ではテストコードが存在しないものとして進めています。

```
    }
}
```

この時点でテスト実行することで、TennisGame("P1", "P2") というクラス生成や getScore() メソッドの呼び出しの部分で実行時エラーが起きないことが確認できます。

2.2.2 期待値を入れてテスト実行する

（以降をご覧になりたい方はご購入をお願いします。）

第 3 章

要件を元に責務ごとにロジックを分割して整理する

3.1 題材

本章では、複数のやりたい処理が 1 つのロジック内に記載されている例に対してリファクタリングを行っていきます。
以下のような題材です。

3.1.1 題材元

https://github.com/emilybache/GildedRose-Refactoring-Kata
なお、この題材は MIT ライセンスとなっています。
https://github.com/emilybache/GildedRose-Refactoring-Kata/blob/master/license.txt

3.1.2 要件

要件として次のように書かれています。(https://github.com/emilybache/GildedRose-Refactoring-Kata/blob/master/GildedRoseRequirements.txt に載っている内容を日本語訳したものになります。)

■コラム: Gilded Rose 要件仕様書
こんにちは、チーム・ギルドローズへようこそ。
我々はアリソンという気さくな宿屋さんが経営する、都会の一等地にある小さな宿

です。また、最高級の商品のみを仕入れて販売もしています。
残念なことに、商品は賞味期限が近づくにつれ、品質が低下していきます。
在庫を更新するシステムがあります。これは、新たな冒険へと旅立ったリーロイという無神経な性格の人物によって開発されました。
あなたの仕事は、システムに新しい機能を追加して、新しいカテゴリーのアイテムを販売できるようにすることです。
最初にシステムの紹介をします。

- すべてのアイテムには、アイテムを販売するための残り日数を示す SellIn 値があります。
- すべてのアイテムには、そのアイテムの価値を示す Quality 値があります。
- 毎日の終わりには、私たちのシステムは、すべての項目の両方の値を変更します。

簡単でしょ？ ここからが面白いところです。

- 販売するための残り日数が無くなると、Quality 値は 2 劣化します。
- Quality 値は決してマイナスにはなりません。
- "Aged Brie"は、日が経つほど Quality 値が上がっていきます。
- Quality 値は 50 以上にはなりません。
- "Sulfuras"は伝説のアイテムなので、販売されたり、Quality 値が低下したりすることはありません。
- "Backstage passes"は、"Aged Brie"と同様、SellIn 値が近づくにつれて Quality 値が上昇し、10 日以内になると毎日 2 上がり、5 日以内になると毎日 3 上がりますが、コンサート終了後には 0 になります。

最近、"Conjured"アイテムのサプライヤーと契約しました。そのため、システムの更新が必要です。

- "Conjured"アイテムは、通常のアイテムの 2 倍の速さで品質が劣化します。

すべてが正常に動作する限り、update-quality メソッドに変更を加えたり、新しいコードを追加したりすることは自由に行ってください。
ただし、Item クラスや Items プロパティは変更しないでください。隅にいるゴブリンのものなので、コードの共有所有権を信じていないので、怒り狂ってあなたを一発で撃ってきます（UpdateQuality メソッドと Items プロパティを静的にしても構

いません。）
　ただし、"Sulfuras "は伝説のアイテムであるため、Quality 値は 80 であり、Quality 値が変わることはありません。

3.1.3　初期コード

　最初に存在しているコードは以下のようなものです。

リスト 3.1: Item.java の初期コード

```
 1: package com.gildedrose;
 2:
 3: public class Item {
 4:
 5:     public String name;
 6:
 7:     public int sellIn;
 8:
 9:     public int quality;
10:
11:     public Item(String name, int sellIn, int quality) {
12:         this.name = name;
13:         this.sellIn = sellIn;
14:         this.quality = quality;
15:     }
16:
17:    @Override
18:    public String toString() {
19:         return this.name + ", " + this.sellIn + ", " + this.quality;
20:     }
21: }
```

リスト 3.2: GildedRose.java の初期コード

```
 1: package com.gildedrose;
 2:
 3: class GildedRose {
 4:     Item[] items;
```

```
 5:
 6:     public GildedRose(Item[] items) {
 7:         this.items = items;
 8:     }
 9:
10:     public void updateQuality() {
11:         for (int i = 0; i < items.length; i++) {
12:             if (!items[i].name.equals("Aged Brie")
13:                     && !items[i].name.equals
14:                         ("Backstage passes to a TAFKAL80ETC concert")) {
15:                 if (items[i].quality > 0) {
16:                     if (!items[i].name.equals(
17:                         "Sulfuras, Hand of Ragnaros")) {
18:                         items[i].quality = items[i].quality - 1;
19:                     }
20:                 }
21:             } else {
22:                 if (items[i].quality < 50) {
23:                     items[i].quality = items[i].quality + 1;
24:
25:                     if (items[i].name.equals
26:                         ("Backstage passes to a TAFKAL80ETC concert")) {
27:                         if (items[i].sellIn < 11) {
28:                             if (items[i].quality < 50) {
29:                                 items[i].quality = items[i].quality + 1;
30:                             }
31:                         }
32:
33:                         if (items[i].sellIn < 6) {
34:                             if (items[i].quality < 50) {
35:                                 items[i].quality = items[i].quality + 1;
36:                             }
37:                         }
38:                     }
39:                 }
40:             }
41:
42:             if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
43:                 items[i].sellIn = items[i].sellIn - 1;
44:             }
45:
46:             if (items[i].sellIn < 0) {
47:                 if (!items[i].name.equals("Aged Brie")) {
48:                     if (!items[i].name.equals
49:                         ("Backstage passes to a TAFKAL80ETC concert")) {
50:                         if (items[i].quality > 0) {
51:                             if (!items[i].name.equals
```

```
52:                                 ("Sulfuras, Hand of Ragnaros")) {
53:                                 items[i].quality = items[i].quality - 1;
54:                             }
55:                         }
56:                     } else {
57:                         items[i].quality =
58:                             items[i].quality - items[i].quality;
59:                     }
60:                 } else {
61:                     if (items[i].quality < 50) {
62:                         items[i].quality = items[i].quality + 1;
63:                     }
64:                 }
65:             }
66:         }
67:     }
68: }
```

なかなかのレガシーコードっぷりです。

また、やりかけのテストコードも存在していました。

リスト 3.3: GildedRoseTest.java の初期コード

```
 1: package com.gildedrose;
 2:
 3: import org.junit.jupiter.api.Test;
 4:
 5: import static org.junit.jupiter.api.Assertions.assertEquals;
 6:
 7: class GildedRoseTest {
 8:
 9:     @Test
10:     void foo() {
11:         Item[] items = new Item[] { new Item("foo", 0, 0) };
12:         GildedRose app = new GildedRose(items);
13:         app.updateQuality();
14:         assertEquals("fixme", app.items[0].name);
15:     }
16:
17: }
```

この状態から手を加えていく方法を考えていきます。

3.2 とりあえずテストを実行する

3.2.1 最初のテストコード

まずは、テスト実行が Green になるようにテストコードを修正します。

リスト 3.4: GildedRoseTest.java にある最初のテストコードを Green にする

```
package com.gildedrose;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class GildedRoseTest {

    @Test
    void foo() {
        Item[] items = new Item[] { new Item("foo", 0, 0) };
        GildedRose app = new GildedRose(items);
        app.updateQuality();
-------assertEquals("fixme", app.items[0].name);
        assertEquals("foo", app.items[0].name);
    }

}
```

期待値を"fixme"から"foo"に書き換えただけです。

今回はたまたまテストクラスがありましたが、テストクラスが無い場合でも同様のテストを作ることになったでしょう。以前書いたように、ポイントは「まずはテストクラスを作成して、とりあえずテスト実行をしてみる」です。

3.3 最初のメソッド切り出し

（以降をご覧になりたい方はご購入をお願いします。）

付録 A

IDE の入力補助の画像解説

ここでは、本書籍内で出てきたリファクタリング内容について IntelliJ を用いた入力補助の方法を画像つきで解説します。

脚注には、Mac の場合のデフォルトのショートカットキーを記載しています。

（以降をご覧になりたい方はご購入をお願いします。）

付録 B

参考文献

（以降をご覧になりたい方はご購入をお願いします。）

テストコードの注入から始めるレガシーコードのリファクタリング（サンプル版）

2020 年 12 月 8 日　v1.0.0

著　者　風間 裕也
編　集　風間 裕也

発行所　crabink