RUP®/XP ガイドライン: テスト先行設計とリファクタリング

Robert C. Martin
Object Mentor, Inc.

Rational Software ホワイト・ペーパー

TP 159, 03/01

Rational®
the software development company

目次

概説

ソフトウェアの分野では、本当に革新的な実践方法が現れるのはまれです。構造化プログラミングやオブジェクト指向は、そのような実践方法の 1 つです。そして、テスト先行設計とリファクタリングも、その 1 つとして数えられます。

リファクタリングの正確な (本来の) 定義は、プログラムの機能を維持したままその構造を変化させる、微少な変更を行うことです。この定義は、ソフトウェアには 2 つの価値があるという概念を示しています。第一に、ソフトウェアが行う内容に価値があります。第二に、ソフトウェアの構造に価値があります。この定義に従うと、リファクタリングはソフトウェアの構造的価値を保守して向上させるための技術と言えます。

より洗練された定義では、リファクタリングは機能の追加と構造的な改良に交互に焦点を置いた無数の微少な変更を通して、ソフトウェアを設計して実装する技術です。この定義は、Fowler 氏が著書「*Refactoring*」(邦訳:「リファクタリング - プログラミングの体質改善テクニック」) (参考資料 [1]) で説明している内容を拡張し、*エクストリーム・プログラミング* (XP) (参考資料 [2]) プロセスにおいてソフトウェアを設計して作成する方法を表しています。

テスト先行設計とリファクタリングは、コードを作成する前にそのテスト・ケースを作成することで、コードを設計して改良する実践方法です。プログラマーはタスクを選択し、プログラムがこのタスクを実行しないと失敗する 1 つか 2 つの簡単な単体テスト・ケースを作成します。その上で、テストを通過するようにプログラムを修正します。ソフトウェアが想定されたすべての動作を行うようになるまで、プログラマーは継続的にテスト・ケースを追加し、テストに通過するようにプログラムを修正します。プログラマーは 1 度に少しずつシステム構造を向上させ、各ステップ間ですべてのテストを実行し、変更によって別の部分に問題が発生していないことを確認します。

リファクタリングの例

テスト先行設計とリファクタリングは、例を使って説明するのが最適です。ここでは、リファクタリングをどのように行うかを説明する簡単なプログラムを設計して実装します。XP では、プログラマー 2 人が 1 台のワークステーションを使用してアクティビティーを行います。[1]

作成するアプリケーションは、簡単な自動車の走行記録です。ガソリンスタンドで給油するたびに、ユーザーは給油量、ガソリン価格、現在の走行距離計の値を入力します。システムはこれらの項目を記録して、レポートを生成します。実装言語には Java を使用します。

リスト 1 のコードを作成することから始めます。

TestAutoMileageLog.java　　リスト 1

```
import junit.framework.*;

public class TestAutoMileageLog extends TestCase
{
    public TestAutoMileageLog(String name)
    {
        super(name);
    }
}
```

最初に作成するのは、単体テストを記述するためのフレームワークです。順番が逆にも見えますが、これがテスト先行概念の基本です。実際のアプリケーション・コードを作成する前に、まずテスト・コードを作成します。どのように機能するかを、手順と共に確認してください。

使用するテスト・フレームワークは JUnit です。これは、Kent Beck 氏と Erich Gamma 氏が作成した、簡単な単体テスト・フレームワークです。リスト 1 に示したコードで、設定に必要な作業はすべて完了しました。

[1] Rational Software ホワイト・ペーパーの「*RUP®/XP ガイドライン: ペア プログラミング*」を参照してください。

続いて、最初のテスト・ケースを考える必要があります。このソフトウェアが何をするかを考えてください。必要なのは、給油したことの記録です。これは、関係のあるデータを保持する FuelingStationVisit オブジェクトが必要であることを意味します。したがって、このオブジェクトを作成し、そのフィールドを問い合わせるテストを作成します。

まず、テスト関数を作成することから始めます。JUnit では、テスト関数は TestCase から派生したクラスのメソッドです。メソッドの名前は、"test" から始まります。リスト 2 を参照してください。

TestAutoMileageLog.java リスト 2

```
import junit.framework.*;

public class TestAutoMileageLog extends TestCase
{
    public TestAutoMileageLog(String name)
    {
        super(name);
    }

    public void testCreateFuelingStationVisit()
    {
        FuelingStationVisit v = new FuelingStationVisit();
    }
}
```

太字の部分が新しいコードです。ここで行ったことは、FuelingStationVisit という名前の新しいオブジェクトを作成することだけです。まだ、作成のための引数は与えていません。ここでは、オブジェクトを確実に作成することにだけ注目します。

明らかに、このコードはコンパイルを通りません (コンパイルしたいですよね?)。コンパイルするには、FuelingStationVisit オブジェクトのコードを作成する必要があります。リスト 3 を参照してください。

TestAutoMileageLog.java リスト 3.1

```
import junit.framework.*;
import FuelingStationVisit;

public class TestAutoMileageLog extends TestCase
{
    public TestAutoMileageLog(String name)
    {
        super(name);
    }

    public void testCreateFuelingStationVisit()
    {
        FuelingStationVisit v = new FuelingStationVisit();
    }
}
```

FuelingStationVisit.java リスト 3.2

```
public class FuelingStationVisit
{
}
```

このコードはコンパイルされ、テストも実行できます。これで、必要な機能を追加する準備ができました。

TestAutoMileageLog.java リスト 4.1

```
import junit.framework.*;
import FuelingStationVisit;
```

```
import java.util.Date;

public class TestAutoMileageLog extends TestCase
{
    public TestAutoMileageLog(String name)
    {
        super(name);
    }

    public void testCreateFuelingStationVisit()
    {
        Date date = new Date();
        double fuel = 2.0;  // 2 ガロン
        double cost = 1.87*2; // 料金 = $1.87/ガロン
        int mileage = 1000; // 走行距離計の値
        double delta = 0.0001; // 浮動小数点数の誤差の許容範囲

        FuelingStationVisit v =
            new FuelingStationVisit(date, fuel, cost, mileage);
        assertEquals(date, v.getDate());
        assertEquals(1.87*2, v.getCost(), delta);
        assertEquals(2, v.getFuel(), delta);
        assertEquals(1000, v.getMileage());
        assertEquals(1.87, v.getPrice(), delta);
    }
}
```

FuelingStationVisit.java　　リスト 4.2

```
import java.util.Date;

public class FuelingStationVisit
{
    private Date itsDate;
    private double itsFuel;
    private double itsCost;
    private int itsMileage;

    public FuelingStationVisit(Date date, double fuel,
                               double cost, int mileage)
    {
        itsDate = date;
        itsFuel = fuel;
        itsCost = cost;
        itsMileage = mileage;
    }

    public Date getDate() {return itsDate;}
    public double getFuel() {return itsFuel;}
    public double getCost() {return itsCost;}
    public double getPrice() {return itsCost/itsFuel;}
    public int getMileage() {return itsMileage;}
}
```

このステップでは、まず TestAutoMileageLog にテストを追加し、FuelingStationVisit にメソッドを追加しています。テストの準備ができるまでに 3 回から 4 回のコンパイルが実行され、ここで最初のテストが実行されました。

なぜこのように極端な漸進主義を採用するのか不思議に思われるかもしれません。簡単に FuelingStationVisit を作成して、その後にテスト・コードを作成できなかったでしょうか。FuelingStationVisit のテストは本当に必要なのでしょうか。ここまでは、テストを最初に作成すること、またはテストを作成すること自体に、ほとんど利点はありません。ただし、ここまでのコードが正常にコンパイルされ実行されることは明らかになりました。したがって、次に変更を行った結果、コンパイル・エラーやテストの失敗が発生した場合、問題はこれまでに作成したコードではなく変更した部分にあることになります。それほど大きな利点には思えませんが、後では非常に重要となります。テストを続けましょう。

続いて、FuelingStationVisit オブジェクトをどこかに配置する必要があります。いくつかのオブジェクトでは、これを保持する必要があります。どのようなオブジェクトで必要になるでしょうか。この情報を保持して管理する「ユーザー」です。したがって、FuelingStationVisit オブジェクトを保持する User オブジェクトを作成できます。しかし、ここで FuelingStationVisit オブジェクトの走行距離フィールドに注目してください。走行距離は自動車の属性です。そして、FuelingStationVisit オブジェクトは、ガソリンスタンドを訪れたときの Vehicle の状態を記録します。したがって、Vehicle オブジェクトを作成して、ここで FuelingStationVisit オブジェクトを保持する方が適切です。

TestAutoMileageLog.java リスト 5.1

```
import junit.framework.*;
import FuelingStationVisit;
import java.util.Date;

public class TestAutoMileageLog extends TestCase
{
    public TestAutoMileageLog(String name)
    {
        super(name);
    }

    . . .

    public void testCreateVehicle()
    {
        Vehicle v = new Vehicle();
        assertEquals(0, v.getNumberOfVisits());
    }
}
```

Vehicle.java リスト 5.2

```
public class Vehicle
{
    public int getNumberOfVisits()
    {
        return 0;
    }
}
```

リスト 5 は最初のステップを示しています。ここで testCreateVehicle という新しいテスト関数を作成しました。この関数は、Vehicle を作成し、そこに含まれる給油回数を 0 にしています。getNumberOfVisits の実装は明らかに間違っていますが、テストは通過します。続いて、より適切なソリューションにリファクタリングします。

Vehicle.java リスト 6

```
import java.util.Vector;

public class Vehicle
{
    private Vector itsVisits = new Vector();

    public int getNumberOfVisits()
    {
        return itsVisits.size();
    }
}
```

今回もテストを通過します。testCreateVehicle 関数だけでなく、すべてのテストを実行していることに注意してください。これにより、今回行った変更によって、これまで動作していた部分が破壊されていないことが保証されます。

次に、Vehicle に給油回数を追加する方法を解決します。最も簡単なテスト・ケースはどのようになるでしょうか。

TestAutoMileageLog.java　リスト 7

```
    public void testAddVisit()
    {
        double fuel = 2.0;  // 2 ガロン
        double cost = 1.87*2; // 料金 = $1.87/ガロン
        int mileage = 1000; // 走行距離計の値
        double delta = 0.0001; // 浮動小数点数の誤差の許容範囲

        Vehicle v = new Vehicle();
        v.addFuelingStationVisit(fuel, cost, mileage);
        assertEquals(1, v.getNumberOfVisits());
    }
```

このテストでは、FuelingStationVisit オブジェクトを作成していない点に注意してください。Vehicle の addFuelingStationVisit メソッドでは、FuelingStationVisit オブジェクトを作成する必要があるようなので、これをリストに追加します。

Vehicle.java　リスト 8

```
    public void addFuelingStationVisit(double fuel, double cost, int mileage)
    {
        FuelingStationVisit v =
            new FuelingStationVisit(new Date(), fuel, cost, mileage);
        itsVisits.add(v);
    }
```

今回も、すべてのテストにパスします。

testAddVisit と testCreateFuelingStationVisit の 2 つの関数に、重複したコードがあることは好ましくありません。どちらの関数も同じローカル変数を作成し、同じ値で初期化しています。この重複を除去します。テスト・プログラムをリファクタリングして、ローカル変数をメンバー変数に変更します。

TestAutoMileageLog.java　リスト 9

```
import junit.framework.*;
import FuelingStationVisit;
import java.util.Date;

public class TestAutoMileageLog extends TestCase
{
    private double fuel = 2.0;        // 2 ガロン
    private double cost = 1.87 * 2;   // 料金 = $1.87/ガロン
    private int mileage = 1000;       // 走行距離計の値
    private double delta = .0001;     // 浮動小数点数の誤差の許容範囲

    public TestAutoMileageLog(String name)
    {
        super(name);
    }

    public void testCreateFuelingStationVisit()
    {
        Date date = new Date();

        FuelingStationVisit v =
            new FuelingStationVisit(date, fuel, cost, mileage);
        assertEquals(date, v.getDate());
```

```
        assertEquals(1.87*2, v.getCost(), delta);
        assertEquals(2, v.getFuel(), delta);
        assertEquals(1000, v.getMileage());
        assertEquals(1.87, v.getPrice(), delta);
    }

    public void testCreateVehicle()
    {
        Vehicle v = new Vehicle();
        assertEquals(0, v.getNumberOfVisits());
    }

    public void testAddVisit()
    {
```

```
        Vehicle v = new Vehicle();
        v.addFuelingStationVisit(fuel, cost, mileage);
        assertEquals(1, v.getNumberOfVisits());
    }
}
```

このリファクタリングには名前が付けられています。これは PROMOTE TEMP TO FIELD と呼ばれています。同様のリファクタリングのリストとこれを適用する方法は、参考資料 [1] と www.refactoring.com にあります。

単体テストの存在により、このリファクタリングによってほかの部分に問題が発生していないことを素早く確認できます。アプリケーションをリファクタリングして再構築するときに、この点を利用していきます。コードに対して不安を感じるような操作を行った場合は、すべてが正常に動作するかを確認するためにテスト段階に戻ります。

続いて、`Vehicle` に `FuelingStationVisit` オブジェクトを追加して、`Vehicle` にレポートを生成するように要求します。最初に最も簡単なテスト・ケースを作成します。

TestAutoMileageLog.java リスト 10

```
    public void testSingleVisitMileageReport()
    {
        Vehicle v = new Vehicle();
        v.addFuelingVisit(fuel, cost, mileage);
        MileageReport r = v.generateMileageReport();
        assertEquals(0,r.getMilesDriven());
        assertEquals(fuel,r.getFuelConsumed());
        assertEquals(0,r.getMilesPerGallon());
        assertEquals(cost,r.getTotalFuelCost());
    }
```

このテスト・ケースを作成するには、レポート生成に関する問題を考える必要があります。まず、`Vehicle` に `generateMileageReport` という名前のメソッドを持たせる必要があると考えました。次に、この関数が `MileageReport` という名前のオブジェクトを返す必要があると考えました。最後に、`MileageReport` にはいくつかの照会メソッドを持たせる必要があると考えました。

これらの照会メソッドが返す値に注目します。1 回の給油だけでは、走行距離や燃費を計算することはできません。これらの値を計算するには、少なくとも 2 回分の給油記録が必要です。一方、ガソリンの消費量とガソリンの価格は 1 回の給油で計算できます。

もちろん、このままではテスト・ケースはコンパイルされません。適切なメソッドとクラスを追加する必要があります。最初に、コンパイルを行うのに必要なだけのコードを追加します。ただし、テストは通過しません。

Vehicle.java リスト 11.1

```
    public MileageReport generateMileageReport()
    {
        return new MileageReport();
    }
```

TestAutoMileageLog.java リスト 11.2

```
    public void testSingleVisitMileageReport()
    {
        Vehicle v = new Vehicle();
        v.addFuelingVisit(fuel, cost, mileage);
        MileageReport r = v.generateMileageReport();
        assertEquals(0,r.getMilesDriven());
        assertEquals(fuel,r.getFuelConsumed());
        assertEquals(0,r.getMilesPerGallon());
        assertEquals(cost,r.getTotalFuelCost());
    }
```

MileageReport.java リスト 11.3

```
public class MileageReport
{
    public int getMilesDriven() {return itsMilesDriven;}
    public double getMilesPerGallon() {return itsMilesPerGallon;}
    public double getTotalFuelCost() {return itsTotalFuelCost;}
    public double getFuelConsumed() {return itsFuelConsumed;}

    private int itsMilesDriven;
    private double itsMilesPerGallon;
    private double itsTotalFuelCost;
    private double itsFuelConsumed;
}
```

リスト 11 に示すコードでは、コンパイルと実行は可能ですが、テストに失敗します。テストを通過するように、コードをリファクタリングする必要があります。まず、最も簡単な方法から行います。

Vehicle.java リスト 12.1

```
    public MileageReport generateMileageReport()
    {
        MileageReport r =  new MileageReport();
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(0);
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(v.getCost());
        r.setFuelConsumed(v.getFuel());
        return r;
    }
```

MileageReport.java リスト 12.2

```
    public void setMilesPerGallon(double mpg) {itsMilesPerGallon = mpg;}
    public void setMilesDriven(int miles)     {itsMilesDriven=miles;}
    public void setTotalFuelCost(double cost) {itsTotalFuelCost=cost;}
    public void setFuelConsumed(double fuel)  {itsFuelConsumed=fuel;}
```

`Vehicle` が 1 度だけ給油していると想定します（後でほかの条件に関するテスト・ケースを追加するので、心配は不要です）。`MileageReport` のフィールドを適切に設定し、これを返します。

`generateMileageReport` をこのように実装するのは明らかに不完全なので、無駄であるようにも思えます。しかし、実装を少しずつ行うことには、各コンパイルとテストの間に変更点が非常に少ないという利点があります。何か障害がある場合は、いつでも最後のバージョンに戻ってやり直すことができます。デバッグをする必要はありません。

リスト 12 のコードはコンパイルされテストを通過しますが、不完全であることは明らかです。コードを完成させるには、ほかのテスト・ケースを考える必要があります。

- 1 度も給油していない場合
- 2 回目以降の給油である場合

1 度も給油していない場合は簡単です。リスト 13.1 のテスト・ケースは失敗しますが、リスト 13.2 により通過します。

TestAutoMileageLog.java リスト 13.1

```
public void testNoVisitsMileageReport()
{
    Vehicle v = new Vehicle();
    MileageReport r = v.generateMileageReport();
    assertEquals(0,r.getMilesDriven());
    assertEquals(0,r.getFuelConsumed(),delta);
    assertEquals(0,r.getMilesPerGallon(),delta);
    assertEquals(0,r.getTotalFuelCost(),delta);
}
```

Vehicle.java リスト 13.2

```
public MileageReport generateMileageReport()
{
    MileageReport r =  new MileageReport();
    if (itsVisits.size() == 0)
    {
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(0);
        r.setFuelConsumed(0);
    }
    else
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(0);
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(v.getCost());
        r.setFuelConsumed(v.getFuel());
    }
    return r;
}
```

次に、2 回目以降の給油のテスト・ケースを考える必要があります。

TestAutoMileageLog.java リスト 14

```
public void testMultipleVisitMileageReport()
{
    Vehicle v = new Vehicle();
    v.addFuelingStationVisit(5, 6.10, 17942);
    v.addFuelingStationVisit(9.8, 12.24, 18234);
    v.addFuelingStationVisit(8.3, 10.11, 18483);
    MileageReport r = v.generateMileageReport();
    assertEquals(541, r.getMilesDriven());
    assertEquals(23.1, r.getFuelConsumed(), delta);
    assertEquals(23.41991, r.getMilesPerGallon(), delta);
    assertEquals(28.45, r.getTotalFuelCost(), delta);
}
```

`Vehicle` に 3 回の給油を設定しました。ここでは、ガソリンの価格を 1 ガロンあたりおよそ 1.20 ドル、燃費をおよそ 30 マイル/ガロン (mpg) と想定しています。したがって、292 マイルの走行距離に対して、給油量を 9.8 ガロン、料金を 12.24 ドルとしました。

ここで、おかしな点があります。走行距離計の各読み取り値は、およそ 30 mpg に基づいています。しかし、走行距離の 541 をガソリン消費量の 23.1 で割り算すると、23.41991 mpg となります。どこに矛盾があるのでしょうか。なぜ 30 mpg に近い値にならないのでしょうか。

よく考えると、ガソリンの消費量は、給油した量すべての合計ではないことがわかります。ガソリンは、各給油の間に消費されます。したがって、初回の給油時に購入したガソリンは mpg の計算には関係ありません。

TestAutoMileageLog.java リスト 15

```
public void testMultipleVisitMileageReport()
{
    Vehicle v = new Vehicle();
    v.addFuelingStationVisit(5, 6.10, 17942);
    v.addFuelingStationVisit(9.8, 12.24, 18234);
    v.addFuelingStationVisit(8.3, 10.11, 18483);
    MileageReport r = v.generateMileageReport();
    assertEquals(541, r.getMilesDriven());
    assertEquals(18.1, r.getFuelConsumed(), delta);
    assertEquals(29.88950, r.getMilesPerGallon(), delta);
    assertEquals(28.45, r.getTotalFuelCost(), delta);
}
```

よくなったようです。テストを作成するときには、何が起こるかわかりません。確かなことがあるとすれば、テストとコードで 2 回の作業を行うときには、コードを 1 度だけ作成する場合よりも誤りが多くなることに気付く必要があるということです。

さて、前のテストを通過させるためのコードを追加する準備ができました。

Vehicle.java リスト 16

```
public MileageReport generateMileageReport()
{
    MileageReport r =  new MileageReport();
    if (itsVisits.size() == 0)
    {
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(0);
        r.setFuelConsumed(0);
    }
    else if (itsVisits.size() == 1)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(0);
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(v.getCost());
        r.setFuelConsumed(v.getFuel());
    }
    else
    {
        int firstOdometerReading = 0;
        int lastOdometerReading = 0;
        double totalCost = 0;
        double fuelConsumption = 0;

        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            if (i==0)
            {
                firstOdometerReading = v.getMileage();
                fuelConsumption -= v.getFuel();
            }
            if (i==itsVisits.size()-1) lastOdometerReading = v.getMileage();
            totalCost += v.getCost();
            fuelConsumption += v.getFuel();
        }

        int distance = lastOdometerReading - firstOdometerReading;
        r.setMilesPerGallon(distance/fuelConsumption);
        r.setMilesDriven(distance);
        r.setTotalFuelCost(totalCost);
        r.setFuelConsumed(fuelConsumption);
```

```
        }
        return r;
    }
```

このコードは、特別なケースがすべて記述されていて不格好です。特別なケースをリファクタリングする必要があります。実際には、3 番目のケースだけでほぼ十分です。ほかの 2 つのケースを削除します。

この場合、`testSingleVisitMileageReport` のテスト・ケースは失敗します。これは、1 回目の給油の場合には、最初に購入したガソリンしか含まれていないためです。前に示したとおり、1 回目の給油時には、ガソリン消費量を 0 にする必要があります。したがって、テスト・ケースとコードを次のように修正します。

Vehicle.java リスト 17

```
    public MileageReport generateMileageReport()
    {
        MileageReport r =  new MileageReport();

        int firstOdometerReading = 0;
        int lastOdometerReading = 0;
        double totalCost = 0;
        double fuelConsumption = 0;

        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            if (i==0)
            {
                firstOdometerReading = v.getMileage();
                fuelConsumption -= v.getFuel();
            }
            if (i==itsVisits.size()-1) lastOdometerReading = v.getMileage();
            totalCost += v.getCost();
            fuelConsumption += v.getFuel();
        }

        int distance = lastOdometerReading - firstOdometerReading;
        r.setMilesPerGallon(distance/fuelConsumption);
        r.setMilesDriven(distance);
        r.setTotalFuelCost(totalCost);
        r.setFuelConsumed(fuelConsumption);

        return r;
    }
```

この関数は長いので、短くして、少し整理する必要があります。まず、コードの一部を別の関数に移動できるように修正します。

Vehicle.java リスト 18

```
    public MileageReport generateMileageReport()
    {
        MileageReport r =  new MileageReport();

        int distance = 0;
        double totalCost = 0;
        double fuelConsumption = 0;
        double firstFuel = 0;
        double mpg = 0;

        if (itsVisits.size() > 0)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
```

```
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
            int firstOdometerReading = firstVisit.getMileage();
            int lastOdometerReading = lastVisit.getMileage();
            distance = lastOdometerReading-firstOdometerReading;
            firstFuel = firstVisit.getFuel();

            for (int i=0; i<itsVisits.size(); i++)
            {
                FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
                totalCost += v.getCost();
                fuelConsumption += v.getFuel();
            }

            fuelConsumption -= firstFuel;
            if (fuelConsumption > 0)
                mpg = distance/fuelConsumption;
        }

        r.setMilesPerGallon(mpg);
        r.setMilesDriven(distance);
        r.setTotalFuelCost(totalCost);
        r.setFuelConsumed(fuelConsumption);

        return r;
    }
```

リスト **18** は中間の段階です。実際には、この状態になるまでに **4** つか **5** つの小さなステップを実行しています。これらの小さなステップごとに、テストを実行して、ほかの部分に問題が発生していないことを確認しています。これらのリファクタリングの目標はコードを分割しやすくすることですが、その方法について明確な意図はありませんでした。したがって、これらの最初のリファクタリングは、ほとんど行き当たりばったりです。十分な時間をかけていませんが、テストでは問題が発生しないことが保証されています。

テストを実行しながらこの時点まで達すると、コードを改善する方法が見つかります。まず、ループを **2** つに分割する[2]ことから始めます。

Vehicle.java リスト 19

```
        if (itsVisits.size() > 0)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
            int firstOdometerReading = firstVisit.getMileage();
            int lastOdometerReading = lastVisit.getMileage();
            distance = lastOdometerReading-firstOdometerReading;
            firstFuel = firstVisit.getFuel();

            for (int i=0; i<itsVisits.size(); i++)
            {
                FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
                fuelConsumption += v.getFuel();
            }
            for (int i=0; i<itsVisits.size(); i++)
            {
                FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
                totalCost += v.getCost();
            }


            fuelConsumption -= firstFuel;
```

[2] www.refactoring.com の「SPLIT LOOP」を参照してください。

```
            if (fuelConsumption > 0)
                mpg = distance/fuelConsumption;
        }
```

テストは通過します。次に、各ループをそれぞれのプライベート・メソッドに抽出します。[3]

Vehicle.java リスト 20

```
    public MileageReport generateMileageReport()
    {
        MileageReport r =  new MileageReport();

        int distance = 0;
        double totalCost = 0;
        double fuelConsumption = 0;
        double firstFuel = 0;
        double mpg = 0;

        if (itsVisits.size() > 0)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
            int firstOdometerReading = firstVisit.getMileage();
            int lastOdometerReading = lastVisit.getMileage();
            distance = lastOdometerReading-firstOdometerReading;
            firstFuel = firstVisit.getFuel();

            fuelConsumption = calculateFuelConsumption();
            totalCost = calculateTotalCost();

            fuelConsumption -= firstFuel;
            if (fuelConsumption > 0)
                mpg = distance/fuelConsumption;
        }

        r.setMilesPerGallon(mpg);
        r.setMilesDriven(distance);
        r.setTotalFuelCost(totalCost);
        r.setFuelConsumed(fuelConsumption);

        return r;
    }

    private double calculateTotalCost()
    {
        double totalCost = 0;
        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            totalCost += v.getCost();
        }
        return totalCost;
    }

    private double calculateFuelConsumption()
    {
        double fuelConsumption = 0;;
        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            fuelConsumption += v.getFuel();
```

[3] www.refactoring.com の「EXTRACT METHOD」を参照してください。

```
        }
        return fuelConsumption;
    }
```

テストは通過します。次に、ガソリン消費の特別なケースを、`calculateFuelConsumption` メソッドに移動します。

Vehicle.java リスト 21

```
    public MileageReport generateMileageReport()
    {
        ,,,
        if (itsVisits.size() > 0)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
            int firstOdometerReading = firstVisit.getMileage();
            int lastOdometerReading = lastVisit.getMileage();
            distance = lastOdometerReading-firstOdometerReading;

            fuelConsumption = calculateFuelConsumption();
            totalCost = calculateTotalCost();

            if (fuelConsumption > 0)
                mpg = distance/fuelConsumption;
        }

        ...

        return r;
    }

    private double calculateTotalCost()
    {
        double totalCost = 0;
        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            totalCost += v.getCost();
        }
        return totalCost;
    }

    private double calculateFuelConsumption()
    {
        double fuelConsumption = 0;
        if (itsVisits.size() > 0)
        {
            for (int i=1; i<itsVisits.size(); i++)
            {
                FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
                fuelConsumption += v.getFuel();
            }
        }
        return fuelConsumption;
    }
```

テストは通過します。これで、2 回目の給油時のガソリン消費量の計算にも、`calculateFuelConsumption` を使用できます。次に、距離を計算する関数を抽出します。

Vehicle.java リスト 22

```
public MileageReport generateMileageReport()
{
    MileageReport r =  new MileageReport();

    int distance = 0;
    double totalCost = 0;
    double fuelConsumption = 0;
    double firstFuel = 0;
    double mpg = 0;

    if (itsVisits.size() > 0)
    {
        distance = calculateDistance();
        fuelConsumption = calculateFuelConsumption();
        totalCost = calculateTotalCost();

        if (fuelConsumption > 0)
            mpg = distance/fuelConsumption;
    }

    r.setMilesPerGallon(mpg);
    r.setMilesDriven(distance);
    r.setTotalFuelCost(totalCost);
    r.setFuelConsumed(fuelConsumption);

    return r;
}

private int calculateDistance()
{
    int distance = 0;
    if (itsVisits.size() > 0)
    {
        FuelingStationVisit firstVisit =
            (FuelingStationVisit)itsVisits.get(0);
        FuelingStationVisit lastVisit =
            (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
        int firstOdometerReading = firstVisit.getMileage();
        int lastOdometerReading = lastVisit.getMileage();
        distance = lastOdometerReading-firstOdometerReading;
    }
    return distance;
}
```

テストは通過します。メイン関数の条件文を取り除いて、不要な部分を整理します。

Vehicle.java リスト 23

```
public MileageReport generateMileageReport()
{
    int distance = calculateDistance();
    double fuelConsumption = calculateFuelConsumption();
    double totalCost = calculateTotalCost();
    double mpg = 0;

    if (fuelConsumption > 0)
        mpg = distance/fuelConsumption;

    MileageReport r =  new MileageReport();
    r.setMilesPerGallon(mpg);
    r.setMilesDriven(distance);
    r.setTotalFuelCost(totalCost);
    r.setFuelConsumed(fuelConsumption);

    return r;
```

```
    }

    private int calculateDistance()
    {
        int distance = 0;
        if (itsVisits.size() > 1)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
            int firstOdometerReading = firstVisit.getMileage();
            int lastOdometerReading = lastVisit.getMileage();
            distance = lastOdometerReading-firstOdometerReading;
        }
        return distance;
    }

    private double calculateTotalCost()
    {
        double totalCost = 0;
        for (int i=0; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            totalCost += v.getCost();
        }
        return totalCost;
    }

    private double calculateFuelConsumption()
    {
        double fuelConsumption = 0;
        if (itsVisits.size() > 1)
        {
            for (int i=1; i<itsVisits.size(); i++)
            {
                FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
                fuelConsumption += v.getFuel();
            }
        }
        return fuelConsumption;
    }
```

テストは通過します。

ずいぶんよくなりました。各関数は必要な機能を備えていて、それぞれが十分に分離されています。メイン関数は小さくなり、理解しやすくなりました。

プログラムがさらに複雑になったと感じるかもしれません。確かに関数の数と行数は増えていますが、プログラムは適切に分割されています。それぞれの関数も理解しやすくなっています。

リスト 16 のケース分析に戻ると、現在は特定の計算関数に関連付けられています。ケース分析の移動が偶然に機能しただけのリスト 17 よりも、はるかに優れています。

このコードは不要に遅いと不満を感じるかもしれません。確かにそのとおりですが、ここでは速度は要求されていません。速度が必要な場合や、現在の実行に要件が適合しない場合は対処できます。ここでは、リスト 23 に示すような明快さと、問題を分離することで十分です。

結論

このホワイト・ペーパーでは、テスト先行設計によるリファクタリングの技術を説明しましたが、本当の目的はプログラミングに対する姿勢を伝えることです。正しく動作するだけでは、プログラムが完了したことにはなりません。実際、プログラムを動作させるのは簡単なことです。プログラムが完了するのは、正しく動作し、そして可能な限りシンプルで簡潔になったときです。

このホワイト・ペーパーでは、この望ましい成果を達成するための方法を示しています。

1. テスト・ケースを作成することで、プログラムを設計します。各テスト・ケースを作成した後に、そのテスト・ケースを通過するコードを作成します。すべてのテストを蓄積し、簡単に繰り返し実行できるようにします。
2. プログラムの一部が動作してからは、その部分が簡潔になるまでリファクタリングを行います。コードを少しずつ変更し、変更を行うごとにテストを実行して、リファクタリングを行います。これにより、変更した部分が問題を起こしていないという信頼が得られ、コードが可能な限り簡潔で明確になるまで、変更を繰り返す勇気が得られます。

参考資料

[1] *Refactoring*, Martin Fowler, Addison Wesley, 1999 (邦訳:「リファクタリング – プログラムの体質改善テクニック」児玉 公信、平澤 章、友野 晶夫、梅沢 真史 訳)

[2] *eXtreme Programming eXplained*, Kent Beck, Addison Wesley, 2000

Rational®
the software development company

Dual Headquarters:
Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail:info@rational.com
Web:www.rational.com
International Locations:www.rational.com/worldwide