RUP®/XP Guidelines: Test-first Design and Refactoring

Robert C. Martin
Object Mentor, Inc.

Rational Software White Paper

Rational®
the e-development company™

Table of Contents

Overview

It is seldom that a truly revolutionary practice surfaces in the software arena. Structured Programming was one such practice. OO was another. Test-first Design and Refactoring is still another.

An accurate, but naïve, definition of refactoring is the act of making tiny changes that preserve a program's function but change its structure. Embedded in this definition is the notion that there are two distinct values to software. First, there is value in what the software does. Secondly, there is value in the structure of the software. According to the definition just given, refactoring is a technique for maintaining and improving the structural value of software.

A more sophisticated definition of refactoring is the technique of designing and implementing software through a myriad of tiny changes that alternately focus on adding function and improving structure. This definition extends the meaning of the word described by Fowler in his book *Refactoring*, (see reference [1]) and describes the way in which software is designed and written in the process of *eXtreme Programming* (XP) (see reference [2]).

Test-first design and refactoring is the practice of designing and then improving code by writing test cases before writing the code that makes them pass. The programmer selects a task, writes one or two very simple unit test cases that fail because the program does not perform this task, and then modifies the program to make the tests pass. The programmer continuously adds more test cases, and makes them pass, until the software does everything it's supposed to do. Then the programmer improves the structure of the system one small step at a time, running all of the tests between each step to make sure nothing has been broken.

A Refactoring Example

The best way to describe test-first design and refactoring is by example. So, here, we will undertake to design and implement a small program, demonstrating how refactoring is accomplished. Note that in XP a pair of programmers using the same workstation would accomplish the activities you are about to see.[1]

The application we will build is a simple auto mileage log. Every time a user visits a filling station, he or she enters the amount of fuel purchased, the price of that fuel, and the current odometer reading of the vehicle. The system keeps track of these items and generates certain useful reports. Our implementation language will be Java.

We start out by writing the code in Listing 1:

TestAutoMileageLog.java Listing 1

```
import junit.framework.*;

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

The first thing we write is the framework for containing our unit tests. This may seem backwards, but it's fundamental to the test-first concept. We write test code first, before we write the actual application code. You will see how it works as we proceed.

The test framework we are using is called JUnit, which is a simple unit-testing framework written by Kent Beck and Erich Gamma. The code above is all that is needed to set it up.

Now we need to consider our first test case. What does this software do? One thing it has to do is record fueling station visits. This implies that there must be a `FuelingStationVisit` object that holds the pertinent data. So we can write a test that creates this object and then queries its fields.

We begin this by writing a test function. In JUnit, a test function is any method of a class derived from `TestCase` whose name begins with the four letters "`test`". See Listing 2.

[1] See the Rational Software white paper titled *RUP®/XP Guidelines: Pair Programming*

TestAutoMileageLog.java Listing 2

```
import junit.framework.*;

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

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

The new code is in boldface. Notice that all we have done is create a new object named `FuelingStationVisit`. We have not given it any construction arguments yet. All we are interested in, at this point, is making sure that we can create the object. Clearly, this will not compile (though it would be interesting if it did). To get it to compile, we have to write the code for the `FuelingStationVisit` object. See Listing 3.

TestAutoMileageLog.java Listing 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 Listing 3.2

```
public class FuelingStationVisit
{
}
```

This code compiles, and the test runs, so we are ready to add the functionality we want.

TestAutoMileageLog.java Listing 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 gallons.
```

```
        double cost = 1.87*2; // Price = $1.87 per gallon
        int mileage = 1000; // odometer reading.
        double delta = 0.0001; //tolerance on floating point equality.

        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 Listing 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;}
}
```

This step was made by adding the tests to `TestAutoMileageLog` first, and then adding the methods to the `FuelingStationVisit`. There were three or four compiles involved before it was ready to be tested. The tests ran the first time.

You might wonder what this extreme incrementalism is buying us. Couldn't we just have written the `FuelingStationVisit` and then written the test code afterwards? Is it necessary to test `FuelingStationVisit` at all? So far, writing the tests first, or even writing them at all, has given us very little benefit—except one. We know, unambiguously, that the above code compiles and executes. We therefore know that if the next change results in compiler errors, or test failures, that the problem was in the change, not in the previous code. This may seem to be a small benefit, but it will become much more important later.

Next, we need to put `FuelingStationVisit` objects somewhere. Some object needs to hold them. What object should that be? It is the *user* who wants to keep and manage this information, so we could create a `User` object to hold the `FuelingStationVisit` objects. However, the mileage field in the `FuelingStationVisit` object makes me wonder. Mileage is an attribute of a vehicle. The `FuelingStationVisit` object is recording part of the state of a `Vehicle` at the moment of the visit. Therefore, we should create a `Vehicle` object and hold the `FuelingStationVisit` objects within it.

TestAutoMileageLog.java Listing 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 Listing 5.2

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

Listing 5 shows the initial step. We have created a new test function named `testCreateVehicle`. This function creates a `Vehicle` and then makes sure that the number of visits contained within it is zero. The implementation of `getNumberOfVisits` is clearly wrong, but it has the benefit of making the test pass. That allows us to refactor it into a better solution.

Vehicle.java Listing 6

```
import java.util.Vector;

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

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

Again, the tests pass. It should be noted that we are running all of the tests, not just the `testCreateVehicle` function. This assures us that our changes haven't broken anything that used to work.

Next, we should figure out how to add a visit to a Vehicle. What would the simplest test case look like?

TestAutoMileageLog.java Listing 7

```
    public void testAddVisit()
    {
        double fuel = 2.0;  // 2 gallons.
        double cost = 1.87*2; // Price = $1.87 per gallon
        int mileage = 1000; // odometer reading.
        double delta = 0.0001; //tolerance on floating point equality.

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

Notice that we did not create a `FuelingStationVisit` object in this test. It looks like the `addFuelingStationVisit` method of `Vehicle` must create the `FuelingStationVisit` object, and then add it to the list.

Vehicle.java Listing 8

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

Again, all of the tests pass.

We should be a little uncomfortable with the duplicated code in the two functions `testAddVisit` and `testCreateFuelingStationVisit`. Both functions create the same local variables and initialize them to the same values. We'd like to get rid of this duplication. Therefore, we'll refactor the test program making the local variables into member variables.

TestAutoMileageLog.java Listing 9

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

public class TestAutoMileageLog extends TestCase
{
    private double fuel = 2.0;        // 2 gallons.
    private double cost = 1.87 * 2;   // Price = $1.87 per gallon
    private int mileage = 1000;       // odometer reading.
    private double delta = .0001;     //tolerance on floating point equality.

    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());
    }
}
```

This particular refactoring has a name. It's called PROMOTE TEMP TO FIELD. You can find a list of similar refactorings and the procedures for applying them in reference [1] and at www.refactoring.com.

Notice that the existence of the unit tests allowed us to quickly verify that this refactoring had not broken anything. We will continue to take advantage of this as we refactor and restructure the application. Whenever we do something to the code that makes us feel uneasy, we can fall back on the tests to make sure everything still works.

Having added `FuelingStationVisit` objects to the `Vehicle`, we can now ask the `Vehicle` to produce reports. We write the test cases first, starting with the simplest case.

TestAutoMileageLog.java Listing 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());
    }
```

To write this test case, we had to think through the issues related to report generation. First, we decided that `Vehicle` should have a method named `generateMileageReport`. Next, we decided that this function should return an object named `MileageReport`. Finally, we decided that `MileageReport` should have several query methods.

The values that these query methods return are quite interesting. A single visit is not enough to calculate miles driven or miles per gallon. To calculate these values, we need at least two visits. On the other hand, a single visit is sufficient to calculate fuel consumption and fuel costs.

Of course, the test case does not compile. Therefore, we have to add the appropriate methods and classes. We first add just enough code to make it compile, but fail its tests.

Vehicle.java Listing 11.1

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

TestAutoMileageLog.java Listing 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 Listing 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;
}
```

The code in Listing 11 compiles and executes, but the test fails. Now we need to refactor the code so that it passes the test. At first, we will take the simplest approach possible.

Vehicle.java Listing 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 Listing 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;}
```

We assume that the `Vehicle` has just one visit. (Don't worry; we'll add other test cases for other conditions later.) We set the fields of the `MileageReport` appropriately, and then return it.

It may seem silly to implement `generateMileageReport` this way since we know for sure the implementation is, at best, incomplete. However, implementing in tiny increments has the benefit that nothing much changes between each compile and test. If anything goes wrong, we can always go back to the last version and start again. We don't have to debug.

The code in Listing 12 compiles and passes the tests, but is clearly incomplete. To complete it, we need to think of some other test cases.

- A Vehicle with no visits
- A Vehicle with more than one visit

The case where there are no visits is simple. The test case in Listing 13.1 fails and the code in Listing 13.2 makes it pass again.

TestAutoMileageLog.java Listing 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 Listing 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;
    }
```

Next, we need to consider the test case that deals with many visits.

TestAutoMileageLog.java Listing 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);
    }
```

We have chosen to put three visits into the `Vehicle`. We based the cost on roughly $1.20 per gallon, and the mileage on roughly 30 miles per gallon (mpg). Therefore, we use 9.8 gallons to travel 292 miles at a cost of $12.24.

There is an odd problem here. We based each odometer reading on approximately 30 mpg. However, when we divide 541, the distance driven, by 23.1, the gallons consumed, we get 23.41991 mpg. Why the discrepancy? Why don't we get something close to 30 mpg?

Upon reflection, it becomes clear that fuel consumption is not the sum of all fuel bought in every visit. Fuel is consumed *between* visits. The fuel purchased at the first visit should not be considered when calculating mpg.

TestAutoMileageLog.java Listing 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);
    }
```

This looks much better. You never know what you'll find when you write tests. One thing if for sure—you are bound to find more errors when you specify things twice, that is, in tests and in code, than if you just write the code.

Now we're ready to try adding the code that makes the previous test pass.

Vehicle.java Listing 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;
    }
```

This code is ugly with all of its special cases. We need to refactor the special cases out. In fact, the third case is general enough as it stands. We should be able to eliminate the other two cases.

When we do this, the `testSingleVisitMileageReport` test case fails. The failure is because the single visit case was including the fuel purchased in the first, and only, visit. As we discovered above, fuel consumption must be zero if there is only one visit. Therefore, we can fix the test case and the code.

Vehicle.java Listing 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;
    }
```

This function is long. We need to shorten it and clean it up a bit. We'll begin by moving bits of the code around so that they can be moved into separate functions.

Vehicle.java Listing 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;
    }
```

Listing 18 is an intermediate step. It actually took four or five much smaller steps to get to this point. At each of those smaller steps, we were able to run the tests to ensure that we hadn't broken anything. The goal of those refactorings was to somehow get the code easier to split apart, but we didn't have a firm notion of how to do this. So, these first refactorings were almost random. They didn't take much time and the tests ensured that nothing was broken.

Having reached this point with the tests still running, we can see a way to improve things. We'll start by splitting the loop[2] in two.

[2] See SPLIT LOOP from www.refactoring.com.

Vehicle.java Listing 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;
            if (fuelConsumption > 0)
                mpg = distance/fuelConsumption;
        }
```

The tests still run. Next, we'll extract each loop into its own private method. [3]

Vehicle.java Listing 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);
```

[3] See EXTRACT METHOD from www.refactoring.com.

```
        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();
        }
        return fuelConsumption;
    }
```

The tests still run. Next, we'll move the special cases for fuel consumption into the `calculateFuelConsumption` method.

Vehicle.java Listing 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;
}
```

The tests still run. Note that `calculateFuelConsumption` can now take the expedient of starting to sum fuel consumption with the *second* visit. Next, we can extract the function to calculate distance.

Vehicle.java Listing 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;
}
```

The tests still run. Now we can remove the conditional in the main function, and clean up a few odds and ends.

Vehicle.java Listing 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;
    }
```

The tests still run.

This is pretty good. Each function is self-contained and well isolated from the others. The main function is small and easy to understand.

You might argue that this has made the program more complicated. While it has certainly increased the function count and the line count, it has also partitioned the program nicely. Each function is easy to understand.

Notice that the case analysis from Listing 16 has returned, but is now associated with the specific calculation functions. This is far better than Listing 17 where the removal of the case analysis just worked by accident.

Some might complain that this code is needlessly slow. This may be true, but we do not appear to require speed. When speed becomes a requirement, and when the current execution fails that requirement, then we can do something about it. Until that time, we'll be happy with the clarity and separation of concerns shown in Listing 23.

Conclusion

Though this paper has demonstrated the techniques of refactoring in the presence of test-first design; its real purpose was to convey an *attitude* of programming. A program is not done when it works. Indeed, making it work is the easy part. A program is done when it works, *and* when it is as simple and clean as possible.

This paper claims that a good way to achieve this desirable outcome is to:

1. Design the program by writing test cases. After each test case is written, write the code that passes that test case. Accumulate all of the tests and make it easy to run them repeatedly.
2. Once a part of the program works, refactor that part until it's clean. Do the refactoring by making a sequence of tiny changes to the code and by running the tests after each change. This will give you the confidence that your changes aren't breaking anything, and the courage to continue making change after change until the code is as clean and clear as you can make it.

References

[1] *Refactoring*, Martin Fowler, Addison Wesley, 1999W

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

Corporate Headquarters
18880 Homestead Road
Cupertino, CA 95014
Toll-free: 800-728-1212
Tel: 408-863-9900
Fax: 408-863-4120
E-mail: info@rational.com
Web: www.rational.com

For International Offices: www.rational.com/worldwide