

RUP/XP 指南： 测试优先设计和 重构

Robert C. Martin
Object Mentor, Inc.

Rational Software 白皮书

TP 159, 03/01

Rational
the software development company

目录

概述1
重构示例...	..1
结论...	...15
参考资料15

概述

在软件领域中，真正革命性的实践很少。结构化编程是这样的一个实践，面向对象是另一个，测试优先设计和重构也是一个。进行细微变更（保留程序的功能，但更改它的结构）是重构的一个精确但略显简单的定义。此定义中所包含的是以下概念：对于软件而言，存在两种完全不同的价值。首先是软件所达到的目的，其次是软件的结构。按照刚才给出的定义，重构是用于保持和提高软件结构价值的技术。通过交替进行那些侧重于添加功能和改进结构的无数细微的变更来设计和实施软件的技术是重构更完善的定义。此定义扩展了 Fowler 在其 *Refactoring* 一书（请参阅参考资料 [1]）中所述的该词汇的含义，并描述了在 *eXtreme Programming* (XP)（请参阅参考资料 [2]）流程中设计和编写软件的方法。

测试优先设计和重构是一种设计代码然后改进代码的方法，其方式是先编写测试用例，然后再编写能使这些测试用例通过测试的代码。程序员选择任务，编写一个或两个由于程序不执行该任务而失败的非常简单的单元测试用例，然后修改此程序并使测试通过。程序员不断地添加更多测试用例，并使这些用例通过测试，直到该软件能够执行设定要做的所有任务。然后程序员可以每次一小步地改进系统结构，并在每两个步骤之间运行所有测试，以确保没有任何破坏。

重构示例

描述测试优先设计和重构的最佳方式是通过示例予以说明。因此，这里我们将设计和实施一个小程序，用于演示如何实现重构。请注意，在 XP 中，使用同一个工作站的两个程序员将完成您即将看到的活动。¹

我们将建立的应用程序是简单的汽车里程表。用户每次进入加油站时，他或她输入购买的燃料量、该燃料价格和车辆的当前里程表读数。系统会跟踪这些项并生成一些有用的报告。实施语言为 Java。

从编写 Listing 1 中的代码开始：

TestAutoMileageLog.java

Listing 1

```
import junit.framework.*;

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

第一件事是编写含单元测试的框架。这似乎是颠倒了次序，但它是测试优先概念的基础。在编写实际应用程序代码之前，先编写测试代码。随着我们编写工作的进行，您将看到它是如何工作的。

使用的测试框架称为 JUnit，是由 Kent Beck 和 Erich Gamma 编写的一个简单的单元测试框架。以上是建立 JUnit 所需的所有代码。

现在需要考虑第一个测试用例。该软件需做什么呢？必须做的一件事是记录加油站来访数。这意味着必须存在保存相关数据的 `FuelingStationVisit` 对象。因此，可以编写测试用例来创建此对象并查询它的字段。

我们先从编写测试函数开始。在 JUnit 中，测试函数是从 `TestCase` 派生出的类的任何方法，它的名称以四个字母“test”开头。请参阅 Listing 2。

¹ 请参阅标题为《RUP/XP 指南：结对编程》的 *Rational Software* 白皮书。

TestAutoMileageLog.java

Listing 2

```

junit.framework.*;

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

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

```

新的代码使用粗体字。请注意，我们所做的全部事情是创建名为 `FuelingStationVisit` 的新对象。我们尚未对其指定任何构造参数。在此刻我们所感兴趣的就是确保能够创建此对象。显然，这还不能编译（尽管如果编译了将很有趣）。要使其能编译，必须编写 `FuelingStationVisitFuelingStationVisit` 对象的代码。请参阅 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
{
}

```

编译此代码并运行测试用例，以便准备添加想要的功能。

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;}
}

```

此步骤首先向 `TestAutoMileageLog` 添加测试用例，然后向 `FuelingStationVisit` 添加方法。在准备测试之前，涉及三到四次编译。然后首次运行这些测试用例。

您可能想知道这种极端的渐进主义在向我们说明什么。难道不能先编写 `FuelingStationVisit`，然后再编写测试代码吗？究竟是否有必要测试 `FuelingStationVisit`？到目前为止，先编写测试，或者甚至仅编写测试，几乎没给我们带来什么好处 - 除了以下一点。我们明确地知道，上面的代码进行了编译和执行。因此我们知道，如果下一步变更导致编译器错误或测试失败，则问题是由变更引起的，而不是由以前的代码引起的。这似乎是很小的优点，但以后它将变得十分重要。

下一步，需要将 `FuelingStationVisit` 对象放置在某些位置。某个对象需要保存它们。应该是哪个对象呢？应是想要保留和管理此信息的用户，因此可以创建 `User` 对象来保存 `FuelingStationVisit` 对象。但是，`FuelingStationVisit` 对象中的 `mileage` 字段造成了困惑。`mileage` 是车辆的属性。`FuelingStationVisit` 对象是车辆来访时的状态记录部分。因此，应该创建 `Vehicle` 对象并在其中保存 `FuelingStationVisit` 对象。

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 显示了初始步骤。我们已创建了名为 `testCreateVehicle` 的新的测试函数。此函数创建一个 `Vehicle` 并确保包含在其中的来访数为零。
`getNumberOfVisits` 的实施显然是错误的，但它的好处就是使测试通过。它可使我们将其重构为更好的解决方案。

Vehicle.java

Listing 6

```
import java.util.Vector;

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

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

测试再次通过。应该注意的是，运行的是所有测试用例，而不仅是 `testCreateVehicle` 函数。这保证了变更未破坏任何内容。

接下来，应该解决如何向 `Vehicle` 添加一次来访。最简单的测试用例应是什么样的呢？

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());
}
```

请注意，在此次测试中未创建 `FuelingStationVisit` 对象。看来 `Vehicle` 的 `addFuelingStationVisit` 方法必须创建 `FuelingStationVisit` 对象，然后将其添加到列表。

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);
}
```

所有测试再次通过。

对于函数 `testAddVisit` 和 `testCreateFuelingStationVisit` 中的重复代码我们应该会感觉到有些别扭。两个函数创建了同样的局部变量并用同样的值进行了初始化。

我们希望除去这种重复。因此，将重构测试程序，使局部变量变为成员变量。

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());
}
}

```

这种特定重构有一个名称，即 PROMOTE TEMP TO FIELD。您可以在参考资料 [1] 和 www.refactoring.com 中找到一系列类似的重构及应用它们的程序。

请注意，单元测试的存在使我们可快速验证此重构没有造成任何破坏。在重构应用程序时，我们将继续利用这一点。无论何时对代码做了令人担心的操作，都可以回退到测试来确保代码仍会继续工作。

将 FuelingStationVisit 对象添加到 Vehicle 之后，现在可以让 Vehicle 生成报告。先编写测试用例，从最简单的用例开始。

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());
}

```

要编写此测试用例，必须考虑与生成报告相关的问题。首先，确定了 Vehicle 应该有一个名为 generateMileageReport 的方法。接着，确定了此函数应返回名为 MileageReport 的对象。最后，确定了 MileageReport 应该具有多个查询方法。这些查询方法返回的值相当有趣。单次来访不足以计算驶过的英里数或每加仑的英里数。要计算这些值，至少需要两次来访。另一方面，单次来访足以计算燃料消耗量和燃料成本。

当然，测试用例无法编译。因此，必须添加合适的方法和类。首先添加刚好使其能编译但测试失败的代码。

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;
}

```

Listing 11 中的代码能编译和执行，但测试失败。现在，需要重构代码，以使其通过测试。首先，使用最简单的方法。

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;} {itsMilesDriven=miles;}
public void setTotalFuelCost(double cost) {itsTotalFuelCost=cost;}
public void setFuelConsumed(double fuel) {itsFuelConsumed=fuel;} {itsFuelConsumed=fuel;}

```

假设 `Vehicle` 仅作了一次来访。（不必担心；稍后会为其他情况添加相应的测试用例。）恰当地设置 `MileageReport` 的字段，然后返回它。

用这种方法实施 `generateMileageReport` 似乎很愚蠢，因为我们的确知道实施是不完整的。不过，以微小的递增方式来实施也有好处，即每次编译和测试之间的变更很小。如果出现错误，始终可以返回上一个版本并重新开始，而不必进行调试。Listing 12 中的代码能够进行编译并通过测试，但显然是不完整的。要完成它，需要考虑某些其他测试用例。

- ☐ 没有来访数的车辆
- ☐ 已作多次来访的车辆

没有来访数的用例很简单。该测试用例在 Listing 13.1 中失败，而 Listing 13.2 中的代码再次使其通过。

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;
}
```

接下来，需要考虑处理多次来访的测试用例。

TestAutoMileageLog.java

Listing 14

```
public void testMultipleVisitsMileageReport()
{
    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);
}
```

我们已选择将三次来访放入 `Vehicle`。其中每加仑大约 1.20 美元，每加仑行驶的英里数 (mpg) 大约为 30 英里。因此我们以 12.24 美元的价格使用 9.8 加仑行驶 292 英里。这里有一个奇怪的问题。每个里程表读数基于约 30 mpg。但是，当用 541（行驶的距离）除以 23.1（消耗的加仑数）时，得到的是 23.41991 mpg。为什么会有此差异呢？为什么未得到接近 30 mpg 的值呢？

经过思考，道理变得清楚了：燃料消耗不是每次来访时购买的所有燃料总和，而是两次来访之间的消耗值。当计算 mpg 时，不应考虑首次来访时购买的燃料。

```

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);
}

```

这看上去好多了。在编写测试用例时，您从不知道将发现什么。有一件事是肯定的 - 若把事情说明两次，您一定会发现更多错误，即在测试和代码中发现的错误比只编写代码发现的错误要多。现在，准备尝试添加代码来通过前次测试。

```

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;
}

```

此代码由于包括所有特殊用例而很难看。需要重构以除去特殊用例。事实上，第三个用例所代表的内容具有足够的普遍性。所以，应该能够除去其他两个用例。

当执行此步骤时，`testSingleVisitMileageReport` 测试用例失败。失败原因是该来访用例包含第一次（且是唯一一次）来访购买的燃料。正如上面所发现的，如果只有一次来访，燃料消耗必须为零。因此，应修正此测试用例和代码。

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;
}

```

此函数很长，需要缩短它并做一些清理。我们将从移动一些代码开始，这样就可以将它们移进不同的函数。

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 = firstVsi t.getFuel ();

for (int i=0; i<i tsVsi ts.size(); i++)
{
    FuelingStationVsi t v = (FuelingStationVsi t) i tsVsi ts.get(i); total Cost
    += v.getCost();
    fuel Consumption += v.getFuel ();
}

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

r.setMilesPerGallon(mpg);
r.setMilesDriven(distance);
r.setTotal Fuel Cost(total Cost);
r.setFuel Consumed(fuel Consumption);

return r;
}

```

Listing 18 是中间步骤。它实际上利用了四或五个小得多的步骤来达到此目的。在每个小步骤中，我们能够运行测试用例来确保没有造成任何破坏。这些重构的目的是用某种方式使代码更易分离，但对于如何做，并没有固定的想法。因此，这些初次重构几乎是随意的。它们并未占用许多时间并且测试已保证了未破坏任何内容。

在测试仍能运行的情况下达到这一目的后，可以看到一种方法可改进代码。我们从将循环²一分为二开始。

Vehicle.java

Listing 19

```

if (i tsVsi ts.size() > 0)
{
    FuelingStationVsi t firstVsi t =
        (FuelingStationVsi t) i tsVsi ts.get(0);
    FuelingStationVsi t lastVsi t =
        (FuelingStationVsi t) i tsVsi ts.get(i tsVsi ts.size()-1); int
    firstOdometerReading = firstVsi t.getMileage();
    int lastOdometerReading = lastVsi t.getMileage();
    distance = lastOdometerReading - firstOdometerReading;
    firstFuel = firstVsi t.getFuel ();

    for (int i=0; i<i tsVsi ts.size(); i++)
    {
        FuelingStationVsi t v = (FuelingStationVsi t) i tsVsi ts.get(i);
        fuel Consumption += v.getFuel ();
    }
    for (int i=0; i<i tsVsi ts.size(); i++)
    {
        FuelingStationVsi t v = (FuelingStationVsi t) i tsVsi ts.get(i); total Cost
        += v.getCost();
    }

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

```

测试仍能运行。接下来，将每个循环抽取到各自的专用方法中。³

² 请参阅 www.refactoring.com 上的 SPLIT LOOP。

³ 请参阅 www.refactoring.com 上的 EXTRACT METHOD。

```

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

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

    if (itsVehicles.size() > 0)
    {
        FuelingStationVist firstVist =
            (FuelingStationVist)itsVehicles.get(0);
        FuelingStationVist lastVist =
            (FuelingStationVist)itsVehicles.get(itsVehicles.size()-1); int
        firstOdometerReading = firstVist.getMileage();
        int lastOdometerReading = lastVist.getMileage();
        distance = lastOdometerReading-firstOdometerReading;
        firstFuel = firstVist.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<itsVehicles.size(); i++)
    {
        FuelingStationVist v = (FuelingStationVist)itsVehicles.get(i); totalCost
        += v.getCost();
    }
    return totalCost;
}

private double calculateFuelConsumption()
{
    double fuelConsumption = 0;
    for (int i=0; i<itsVehicles.size(); i++)
    {
        FuelingStationVist v = (FuelingStationVist)itsVehicles.get(i);
        fuelConsumption += v.getFuel();
    }
    return fuelConsumption;
}

```

测试仍能运行。接下来，将燃料消耗的特殊用例移入 `calculateFuelConsumption` 方法。

```

public MileageReport generateMileageReport()
{
    ...
    if (itsVehicles.size() > 0)
    {
        FuelingStationVisit firstVisit =
            (FuelingStationVisit)itsVehicles.get(0);
        FuelingStationVisit lastVisit =
            (FuelingStationVisit)itsVehicles.get(itsVehicles.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<itsVehicles.size(); i++)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVehicles.get(i);
        totalCost += v.getCost();
    }
    return totalCost;
}

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

```

测试仍能运行。请注意，`calculateFuelConsumption` 现在可以方便地开始累加第二次来访时的燃料消耗。接着可以抽取函数来计算距离。

```

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

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

    if (itsVehicles.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 (itsVists.size() > 0)
    {
        FuelingStationVist firstVist =
            (FuelingStationVist)itsVists.get(0);
        FuelingStationVist lastVist =
            (FuelingStationVist)itsVists.get(itsVists.size()-1);
        int firstOdometerReading = firstVist.getMileage();
        int lastOdometerReading = lastVist.getMileage();
        distance = lastOdometerReading-firstOdometerReading;
    }
    return distance;
}

```

测试仍能运行。现在可以除去主函数中的条件约束并清除一些无关紧要的内容。

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 (itsVists.size() > 1)
    {
        FuelingStationVist firstVist =
            (FuelingStationVist)itsVists.get(0);
        FuelingStationVist lastVist =
            (FuelingStationVist)itsVists.get(itsVists.size()-1);
        int firstOdometerReading = firstVist.getMileage();
        int lastOdometerReading = lastVist.getMileage();
        distance = lastOdometerReading-firstOdometerReading;
    }
    return distance;
}

```



```

    }

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

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

```

测试仍能运行。

这情况相当好。每个函数是独立的并能很好地从其他函数中分离出来。主函数很小并且易于理解。

您可能会争辩说这使程序变得更加复杂了。虽然它的确增加了函数个数和行数，但也对程序进行了详尽的分割。每个函数都易于理解。

请注意，Listing 16 的用例分析已返回，但现在与特定的计算函数相关联。这远比 Listing 17 要好得多，在 Listing 17 中只会偶尔除去用例分析。

有人可能抱怨没必要这么慢。这可能是事实，但我们看来并不需要速度。如果速度是必需的且当前的执行未能满足此需求，我们再对此采取措施。到那时，我们还会对 Listing 23 中所有关注点的明确性与很好的分隔性感到满意。

结论

虽然本白皮书演示了测试优先设计中的重构技术，但它真正的目的是传达编程的思想方法。当程序起作用时，程序并未完成。的确，让程序起作用很简单。当程序起作用并且尽可能简单和明确时，程序才算完成。

本白皮书主张通过以下方法来实现这一令人满意的结果：

1. 通过编写测试用例来设计程序。在编写好每个测试用例后，编写通过该测试用例的代码。积累所有的测试并使这些测试易于反复运行。
2. 一旦程序的某一部分起作用，则重构该部分直到它变得很简洁。通过对代码进行一系列细微的变更并在每次变更后运行测试来进行重构。这将使您确信变更没有造成任何破坏，并有勇气继续不断作出变更，直到尽您所能使代码更简洁和明确。

参考资料

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

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



两家总部：

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

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

免费电话：(800) 728-1212

电子邮件：info@rational.com

Web：www.rational.com

全球网址：www.rational.com/worldwide

Rational、Rational 徽标和 Rational Unified Process 是 Rational Software Corporation 在美国和 / 或其他国家或地区的注册商标。Microsoft、Microsoft Windows、Microsoft Visual Studio、Microsoft Word、Microsoft Project、Visual C++ 和 Visual Basic 是 Microsoft Corporation 的商标或注册商标。其他所有名称均仅用于标识目的，它们是其相应公司的商标或注册商标。ALL RIGHTS RESERVED.

Copyright 2006 Rational Software Corporation.
如有更改，恕不另行通知。