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 **17**

参考资料 **17**

概述

出现在软件领域中的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。

重构的精确但略显简单的定义是作出微小变更（保留程序函数，但更改它的结构）的操作。此定义中所包含的是以下看法：对于软件而言，存在两种完全不同的价值。首先，软件所做的事情有价值。其次，软件的结构有价值。按照刚才给出的定义，重构是用于维护和改进软件的结构价值的技术。

重构更完善的定义是通过交替侧重于添加功能和改进结构的无数微小的更改来设计和实现软件的技术。此定义扩展了 Fowler 在他的书 *Refactoring*（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做法，其方式是先编写测试用例，然后再写这些测试用例要测试的代码。程序员选择任务，编写一个或两个会因为程序未能执行该任务而失败的非常简单的单元测试用例，然后修改程序使测试通过。程序员不断添加更多测试用例，并使这些用例通过测试，直到软件能够执行设定要做的任何事情。然后，程序员一次一小步地改进系统结构，在每一步之间运行所有测试以确保没有造成任何破坏。

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描述。因此，在此我们将负责设计和实现一个小程序，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将要看到的活动。[1]

我们将建立的应用程序是简单的汽车里程表。用户每次访问加油站时，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。系统跟踪这些项并生成某些有用的报告。实现语言将用 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);
    }
}
```

编写的第一件事是包含单元测试的框架。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 编写的一个简单单元测试框架。上面是设置 JUnit 所需的所有代码。

现在需要考虑第一个测试用例。该软件做什么呢？必须做的一件事是记录加油站访问数。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 派生出的类的任一方法，它的名称以四个字母“test”开头。请参阅 Listing 2。

TestAutoMileageLog.java Listing 2

```
junit.framework.*;
```

[1]请参阅标题如下的 Rational Software 白皮书：*RUP®/XP 指南：成对编程。*

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

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

新的代码是粗体字的。请注意，我们所做的所有事情是创建名为 FuelingStationVisit 的新对象。我们尚未对其指定任何构建实参。在此刻我们所感兴趣的就是确保能够创建此对象。

显然，这将不能编译（尽管如果编译了将很有趣）。要使其能编译，必须为 FuelingStationVisit 对象编写代码。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，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，然后向 FuelingStationVisit 添加方法。在准备测试之前，涉及三到四次编译。然后首次运行测试。

您可能想知道这种极端的渐进主义在向我们说明什么。难道不能写出 FuelingStationVisit，然后再写出测试代码吗？究竟有必要测试 FuelingStationVisit 吗？到目前为止，先编写测试，或者甚至完全编写测试，没给我们带来什么利益 — 除了一点。我们明确地知道上面的代码进行了编译并执行。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，则问题是出在变更中，而不出在以前的代码中。这似乎是很小的好处，但以后它将变得更加重要得多。

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？它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 对象中的英里里程字段令人疑惑。英里里程是车辆的属性。而 FuelingStationVisit 对象记录在访问时 Vehicle 的状态部分。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。此函数创建了一个 Vehicle 并确保包含在其中的访问数为零。getNumberOfVisits 的实现显然是错误的，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，而不仅是 testCreateVehicle 函数。这保证了变更未破坏用于运行的任何内容。

接下来，应该解决如何向 Vehicle 添加一次访问。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。

对于两个函数 testAddVisit 和 testCreateFuelingStationVisit 中的重复代码应该会感到有点不妥。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，使局部变量变成成员变量。

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。它被称为“临时队员提升为参赛队员”。您可以在参考资料 [1] 和 www.refactoring.com 中找到一系列类似的重构和可运用它们的过程。

请注意，单元测试的存在可使我们快速验证此重构未破坏任何内容。在重构和重新构建应用程序时，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，都可以求助于测试来确保代码将继续工作。

将 FuelingStationVisit 对象添加到 Vehicle 之后，现在可以让 Vehicle 产生报告。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。单次访问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，至少需要两次访问。另一方面，单次访问足以计算燃料消耗量和燃料成本。

当然，测试用例不会编译。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，以便其通过测试。首先，将使用最简单的可能方法。

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

假设 Vehicle 仅有一次访问。（别担心；以后将为其它条件添加其它测试用例。）恰当地设置 MileageReport 的字段，然后返回它。

用这种方法实现 generateMileageReport 似乎很愚蠢，因为我们的确知道实现甚至是不完全的。不过，以微小的递增方式实现也有好处，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，需要考虑某些其它测试用例。

- 没有访问数的车辆
- 具有多次访问的车辆

没有访问数的用例很简单。Listing 13.1中的测试用例失败，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，需要考虑处理多次访问的测试用例。

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.getFueConsumed(), delta);
        assertEquals(23.41991, r.getMilesPerGallon(), delta);
        assertEquals(28.45, r.getTotalFuelCost(), delta);
}
```

我们已选择将三次访问放入 Vehicle。我们的根据是每加仑大约 1.20 美元的价格，并且每加仑行驶的英里数大约为 30 英里（mpg）。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。我们使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。为什么会有差异呢？为什么未得到接近 30 mpg 的值呢？

经过思考，道理变得明确了：燃料消耗不是每次访问时购买的所有燃料总和。燃料是在各次访问*之间*消耗的。当计算 mpg 时，不应考虑首次访问时购买的燃料。

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);
}
```

这看上去好多了。在编写测试时，您从不知道将发现什么。有一件事是肯定的 — 若将事情指定两次，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。

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;
    }
```

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具有足够的普遍性。应该能够除去其它两个用例。

当执行此步骤时，testSingleVisitMileageReport 测试用例失败。失败原因是单个访问用例包含第一次（且是唯一一次）访问购买的燃料。正如上面发现的，如果只有一次访问，燃料消耗必须为零。因此，可以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 = 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 是中间步骤。实际利用了四个或五个小得多的步骤来达到此目的。在每个更小的步骤中，能够运行测试来确保未破坏任何内容。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机的。它们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。从将循环分割[2] 为两个开始。

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++)
            {
```

[2]请参阅 www.refactoring.com 中的 SPLIT LOOP。

```
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            totalCost += v.getCost();
        }


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

测试仍能运行。接下来，将每个循环抽取到各自的私有方法中。[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);
    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();
```

---

[3]请参阅 www.refactoring.com 中的 EXTRACT METHOD。

```
        }
        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;
    }
```

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

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;
    }
```

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

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;
    }
```

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。

```
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;
}
```

测试仍能运行。

这情况相当好。每个函数是独立的并很好地从其它函数中分离出来。主函数很小并易于理解。

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，但也精细地分割了程序。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。这比 Listing 17 要好得多，在 Listing 17 中用例分析的除去只是偶然运行的。

有人可能抱怨此代码没必要这么慢。这可能是事实，但我们看来并不需要速度。当速度成为需求并且当前执行未能满足此需求时，我们再对此采取些措施。到那时，将对 Listing 23 中显示的关注点的明确性和区分感到满意。

结论

虽然本白皮书演示了存在测试优先设计的情况下的重构技术；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，程序才算是完成的。

本白皮书主张实现此令人满意的结果的好方法是：

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得很容易反复运行这些测试。
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微小的变更并在每次变更后运行测试来进行重构。这将使您确信您的变更未破坏任何内容，并有勇气继续不断作出变更，直到尽您所能使代码简洁和明确。

参考资料

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

[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