Diretrizes RUP®/XP: Teste Anterior ao Design e Refatoramento

Robert C. Martin
Object Mentor, Inc.

Rational Software White Paper

TP 159, 03/01

Rational®
the software development company

Índice Analítico

Visão Geral **1**

Um Exemplo de Refatoramento **1**

Conclusão **15**

Referências **15**

Visão Geral

Raramente uma prática verdadeiramente revolucionária surge na arena de software. A Programação Estruturada foi uma dessas práticas. OO foi outra. Teste Anterior ao Design e Refatoramento outras.

Uma definição precisa, mas ingênua, de refatoramento é o ato de fazer pequenas mudanças que preservam a função do programa, mas alteram sua estrutura. A noção de que há dois valores distintos para o software tem como base essa definição. Primeiramente, há valor no que o software faz. Além disso, há valor na estrutura do software. De acordo com a definição que acaba de ser fornecida, o refatoramento é uma técnica para manter e aprimorar o valor estrutural do software.

Uma definição mais sofisticada de refatoramento é a técnica de projetar e implementar o software por meio de uma variedade de pequenas mudanças que enfatizam alternativamente a inclusão de funções e o primoramento da estrutura. Essa definição estende o significado da palavra descrita por Fowler em seu livro*Refatoramento,* (consulte a referência [1]) e descreve o modo em que o software é projetado e gravado no processo de *Programação eXtrema* (XP) (consulte a referência [2]).

Teste Anterior ao Design e refatoramento são as práticas de projetar e, em seguida, aprimorar o código, gravando casos de teste antes de gravar o código que os transmite. O programador seleciona uma tarefa, grava um ou dois casos de teste de unidade muito simples que falaharão por causa do programa que não executa esta tarefa e, em seguida, modifica o programa para transmitir os testes. O programador inclui contínuamente mais casos de teste e os transmite até que o software faça tudo que deve ser feito. Assim o programador aprimora a estrutura do sistema uma etapa de cada vez, executando todos os testes entre cada etapa para ter certeza de que nada foi quebrado.

Um Exemplo de Refatoramento

A melhor maneira de descrever o teste anterior ao design e refatoramento é com exemplos. Projetaremos e implementaremos um pequeno programa, demonstrando como o refatoramento é realizado. Observe que no XP um par de programadores utilizando a mesma estação de trabalho realizaria as atividades que você está prestes a ver.[1]

O aplicativo que construíremos é um log de milhagem automático simples. Todas as vezes em que um usuário for a um posto de gasolina, ele ou ela digitará a quantidade de combustível comprado, o preço do combustível e a leitura atual do hodômetro do veículo. O sistema controla esses itens e gera determinados relatórios úteis. Nossa linguagem de implementação será Java.

Começamos gravando o código na Listagem 1:

`TestAutoMileageLog.java` Listagem 1

```
importar junit.framework.*;

público classe TestAutoMileageLog estende TestCase
{
    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }
}
```

A primeira coisa que gravamos é a estrutura para conter nossos testes de unidade. Isso pode parecer invertido, mas é fundamental para o conceito test-first. Gravamos o código de teste primeiro, antes de gravarmos o código de aplicativo verdadeiro. Você verá como funciona na medida em que prosseguirmos.

A estrutura de teste que estamos utilizando é chamada de JUnit, que é uma estrutura de teste simples escrita por Kent Beck e Erich Gamma. O código acima é tudo que é necessário para configurá-lo.

Agora precisamos considerar nosso primeiro caso de teste. O que este software faz? Uma coisa que ele tem que fazer é registrar as visitas ao posto de gasolina. Isso significa que deve haver um objeto `FuelingStationVisit` que abrange os dados pertinentes. Então podemos gravar um teste que crie esse objeto e, em seguida, consulte seus campos.

Começamos isso gravando uma função de teste. Na JUnit, uma função de teste é qualquer método de uma classe derivada de `TestCase` cujo nome começa com as quatro letras "`test`". Consulte a Listagem 2.

[1] Consulte o white paper do Rational Software entitulado *RUP®/XP Diretrizes: Programação aos Pares.*

TestAutoMileageLog.java Listagem 2

```
importar junit.framework.*;

público classe TestAutoMileageLog estende TestCase
{
    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }

    público void testCreateFuelingStationVisit()
    {
        FuelingStationVisit v = novo FuelingStationVisit();
    }
}
```

O novo código está em negrito. Observe que tudo que fizemos foi criar um novo objeto chamado `FuelingStationVisit`. Ainda não fornecemos qualquer argumento de construção. Tudo em que estamos interessados, neste ponto, é ter certeza de que podemos criar o objeto.

Certamente, isso não será compilado (embora seria interessante se fosse). Para obter a compilação, temos que gravar o código para o objeto `FuelingStationVisit`. Consulte a Listagem 3.

TestAutoMileageLog.java Listagem 3.1

```
importar junit.framework.*;
importar FuelingStationVisit;

público classe TestAutoMileageLog estende TestCase
{
    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }

    público void testCreateFuelingStationVisit()
    {
        FuelingStationVisit v = novo FuelingStationVisit();
    }
}
```

FuelingStationVisit.java Listagem 3.2

```
público classe FuelingStationVisit
{
}
```

Esse código compila e o teste executa, assim estamos prontos para incluir a funcionalidade que desejarmos.

TestAutoMileageLog.java Listagem 4.1

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

público classe TestAutoMileageLog estende TestCase
{
    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }

    público void testCreateFuelingStationVisit()
    {
        Data data = novo Data();
        duplo combustível = 2.0;  // 2 galões.
```

```
        duplo custo = 1.87*2; // Preço = $1,87 por galão
        int milhagem = 1000; // leitura do hodômetro.
        duplo delta = 0.0001; //tolerância na igualdade do ponto flutuante.

        FuelingStationVisit v =
            novo FuelingStationVisit(data, combustível, custo, milhagem);
        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 Listagem 4.2

```
importar java.util.Date;

público classe FuelingStationVisit
{
    particular Data itsDate;
    particular duplo itsFuel;
    particular duplo itsCost;
    particular int itsMileage;

    público FuelingStationVisit(Data data, duplo combustível,
                                duplo custo, int milhagem)
    {
        itsDate = data;
        itsFuel = combustível;
        itsCost = custo;
        itsMileage = milhagem;
    }

    público Data getDate() {retornar itsDate;}
    público duplo getFuel() {retornar itsFuel;}
    público duplo getCost() {retornar itsCost;}
    público duplo getPrice() {retornar itsCost/itsFuel;}
    público int getMileage() {retornar itsMileage;}
}
```

Essa etapa foi feita incluindo os testes em `TestAutoMileageLog` primeiro e, em seguida, incluindo os métodos em `FuelingStationVisit`. Havia três ou quatro compilações envolvidas antes que estivesse pronto para ser testado. Os testes executaram a primeira vez.

Você deve estar imaginando o que esse incrementalismo extremo está gerando. Não poderíamos ter apenas gravado o `FuelingStationVisit` e, em seguida, gravado o código de teste posteriormente? É mesmo necessário testar o `FuelingStationVisit`? Até agora a gravação dos testes em primeiro lugar ou até a gravação de tudo nos gerou muito pouco benefício—com uma exceção. Sabemos que o código acima copila e executa. Portanto, sabemos que se a próxima alteração resultar em erros compiladores ou defeitos de teste, que o problema estava na alteração, não no código anterior. Isso pode parecer um benefício pequeno, mas se tornará muito mais importante posteriormente.

A seguir, precisamos colocar os objetos `FuelingStationVisit` em algum lugar. Algum objeto precisa segurá-los. Qual objeto deveria ser? É o *usuário* que deseja manter e gerenciar essas informações, então poderíamos criar um objeto `Usuário` para segurar os objetos `FuelingStationVisit`. No entanto, o campo de milhagem no objeto `FuelingStationVisit` me faz pensar. A milhagem é um atributo de um veículo. O objeto `FuelingStationVisit` está registrando parte do estado de um `Veículo` no momento da visita. Portanto, um objeto `Veículo` deveria ser criado e segurar os objetos `FuelingStationVisit` dentro dele.

TestAutoMileageLog.java Listagem 5.1

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

público classe TestAutoMileageLog estende TestCase
{
    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }

    . . .

    público void testCreateVehicle()
    {
        Veículo v = novo Veículo();
        assertEquals(0, v.getNumberOfVisits());
    }
}
```

Vehicle.java Listagem 5.2

```
público classe Veículo
{
    público int getNumberOfVisits()
    {
        retornar 0;
    }
}
```

A Listagem 5 mostra a etapa inicial. Criamos uma nova função de teste chamada `testCreateVehicle`. Essa função cria um `Veículo` e, em seguida, certifica-se de que o número de visitas contidas dentro dele é zero. A implementação de `getNumberOfVisits` é claramente errada, mas tem o benefício de fazer a transmissão do teste. Isso permite refatorá-la para uma solução melhor.

Vehicle.java Listagem 6

```
importar java.util.Vector;

público classeVeículo
{
    particular Vector itsVisits = novo Vector();

    público int getNumberOfVisits()
    {
        retornar itsVisits.size();
    }
}
```

Novamente os testes são transmitidos. Deve ser observado que todos os testes estão sendo executados, não apenas a função `testCreateVehicle`. Isso nos assegura que nossas mudanças não quebraram nada que estivesse funcionando.

A seguir, devemos descobrir como incluir uma visita em um Veículo. Como seria o caso de teste mais simples?

TestAutoMileageLog.java Listagem 7

```
    público void testAddVisit()
    {
        duplo combustível = 2.0;  // 2 galões.
        duplo custo = 1,87*2; // Preço = $1,87 por galão
        int milhagem = 1000; // leitura do hodômetro.
        duplo delta = 0.0001; //tolerância na igualdade do ponto flutuante.

        Veículo v = novo Veículo();
        v.addFuelingStationVisit(combustível, custo, milhagem);
```

```
        assertEquals(1, v.getNumberOfVisits());
    }
```

Observe que não foi criado um objeto `FuelingStationVisit` neste teste. Parece que o método `addFuelingStationVisit` do `Veículo` deve criar o objeto `FuelingStationVisit` e, em seguida, incluí-lo na lista.

`Vehicle.java` Listagem 8

```
    público void addFuelingStationVisit(duplo combustível, duplo custo, int milhagem)
    {
        FuelingStationVisit v =
            novo FuelingStationVisit(novo Data(), combustível, custo, milhagem);
        itsVisits.add(v);
    }
```

Novamente todos os testes são transmitidos.

Devemos ficar um pouco desconfortáveis com o código duplicado nas duas funções `testAddVisit` e `testCreateFuelingStationVisit`. Ambas as funções criam as mesmas variáveis de local e as inicializam com os mesmos valores. Gostaríamos de remover essa duplicação. Portanto, iremos refatorar o programa de teste tornando as variáveis locais em variáveis do membro.

`TestAutoMileageLog.java` Listagem 9

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

público classe TestAutoMileageLog estende TestCase
{
    particular duplo combustível = 2.0;      // 2 galões.
    particular duplo custo = 1.87 * 2;  // Preço = $1,87 por galão
    particular int milhagem = 1000;      // leitura do odômetro.
    particular duplo delta = .0001;      //tolerância na igualdade do ponto flutuante.

    público TestAutoMileageLog(Nome da cadeia)
    {
        super(nome);
    }

    público void testCreateFuelingStationVisit()
    {
        Data data = novo Data();

        FuelingStationVisit v =
            novo FuelingStationVisit(data, combustível, custo, milhagem);
        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);
    }

    público void testCreateVehicle()
    {
        Veículo v = novo Veículo();
        assertEquals(0, v.getNumberOfVisits());
    }

    público void testAddVisit()
    {
```

```
        Veículo v = novo Veículo();
        v.addFuelingStationVisit(combustível, custo, milhagem);
        assertEquals(1, v.getNumberOfVisits());
    }
}
```

Este refatoramento específico tem um nome. É chamado de PROMOVER TEMP PARA O CAMPO. É possível localizar uma lista de refatoramentos semelhantes e os procedimentos para aplicá-los na referência [1] e em www.refactoring.com.

Observe que a existência dos testes de unidade nos permitiu verificar rapidamente se esse refatoramento não tinha quebrado nada. Continuaremos a aproveitar isso enquanto refatoramos e reestruturamos o aplicativo. Sempre que fizermos algo com o código que nos deixe desconfortável, poderemos confiar nos testes para nos certificarmos de que tudo ainda funciona.

Depois de incluir os objetos `FuelingStationVisit` no `Veículo`, podemos pedir que o `Veículo` produza relatórios. Gravamos os casos de teste primeiro, começando com o caso mais simples.

`TestAutoMileageLog.java` Listagem 10

```
    público void testSingleVisitMileageReport()
    {
        Veículo v = novo Veículo();
        v.addFuelingVisit(combustível, custo, milhagem);
        MileageReport r = v.generateMileageReport();
        assertEquals(0,r.getMilesDriven());
        assertEquals(fuel,r.getFuelConsumed());
        assertEquals(0,r.getMilesPerGallon());
        assertEquals(cost,r.getTotalFuelCost());
    }
```

Para gravar esse caso de teste, tivemos que pensar nos problemas relacionados com a geração de relatórios. Primeiramente, decidimos que o `Veículo` deve ter um método chamado `generateMileageReport`. A seguir, decidimos que essa função deve retornar um objeto chamado `MileageReport`. Finalmente, decidimos que `MileageReport` deve ter vários métodos de consulta.

Os valores que esses métodos de consulta retornam são bem interessantes. Uma única visita não é suficiente para calcular as milhas dirigidas ou as milhas por galão. Para calcular esses valores, precisamos de pelo menos duas visitas. Por outro lado, uma única visita é suficiente para calcular o consumo de combustível e os custos.

É claro que o caso de teste não compila. Portanto, temos que incluir os métodos e classes apropriados. primeiramente incluímos apenas o código suficiente para compilar, mas ele falha nos testes.

`Vehicle.java` Listagem 11.1

```
    público MileageReport generateMileageReport()
    {
        retornar novo MileageReport();
    }
```

`TestAutoMileageLog.java` Listagem 11.2

```
    público void testSingleVisitMileageReport()
    {
        Veículo v = novo Veículo();
        v.addFuelingVisit(combustível, custo, milhagem);
        MileageReport r = v.generateMileageReport();
        assertEquals(0,r.getMilesDriven());
        assertEquals(fuel,r.getFuelConsumed());
        assertEquals(0,r.getMilesPerGallon());
        assertEquals(cost,r.getTotalFuelCost());
    }
```

MileageReport.java Listagem 11.3

```
público classe MileageReport
{
    público int getMilesDriven() {retornar itsMilesDriven;}
    público duplo getMilesPerGallon() {retornar itsMilesPerGallon;}
    público duplo getTotalFuelCost() {retornar itsTotalFuelCost;}
    público duplo getFuelConsumed() {retornar itsFuelConsumed;}

    particular int itsMilesDriven;
    particular duplo itsMilesPerGallon;
    particular duplo itsTotalFuelCost;
    particular duplo itsFuelConsumed;
}
```

O código na Listagem 11 compila e executa, mas o teste falha. Agora precisamos refatorar o código para que passe no teste. Inicialmente, utilizaremos a abordagem mais simples possível.

Vehicle.java Listagem 12.1

```
    público MileageReport generateMileageReport()
    {
        MileageReport r =  novo MileageReport();
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(0);
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(v.getCost());
        r.setFuelConsumed(v.getFuel());
        retornar r;
    }
```

MileageReport.java Listagem 12.2

```
    público void setMilesPerGallon(duplo mpg) {itsMilesPerGallon = mpg;}
    público void setMilesDriven(intmilhas)     {itsMilesDriven=miles;}
    público void setTotalFuelCost(duplo custo) {itsTotalFuelCost=cost;}
    público void setFuelConsumed(duplo combustível)  {itsFuelConsumed=fuel;}
```

Supomos que o `Veículo` tenha apenas uma visita. (Não se preocupe; incluiremos outros casos de teste para outras condições posteriormente.) Definimos os campos de `MileageReport` apropriadamente e, em seguida, o retornamos.

Pode parecer simples implementar o `generateMileageReport` desse jeito desde que saibamos com certeza que a implementação está, na verdade, incompleta. No entanto, a implementação em incrementos pequenos tem o benefício de não haver muitas mudanças entre cada compilação e cada teste. Se algo sair errado, podemos sempre voltar para a última versão e começar novamente. Não precisamos depurar.

O código na Listagem 12 compila e transmite os testes, mas está claramente incompleto. Para completá-lo, precisamos pensar em outros casos de teste.

- Um Veículo sem visitas
- Um Veículo com mais de uma visita

O caso em que não há visitas é simples. O caso de teste na Listagem 13.1 falha e o código na Listagem 13.2 o transmite novamente.

TestAutoMileageLog.java Listagem 13.1

```
público void testNoVisitsMileageReport()
{
    Veículo v = novo Veículo();
    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 Listagem 13.2

```
público MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();
    se (itsVisits.size() == 0)
    {
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(0);
        r.setFuelConsumed(0);
    }
    diferente
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(0);
        r.setMilesPerGallon(0);
        r.setMilesDriven(0);
        r.setTotalFuelCost(v.getCost());
        r.setFuelConsumed(v.getFuel());
    }
    retornar r;
}
```

---

a seguir, precisamos considerar o caso de teste que lida com muitas visitas.

TestAutoMileageLog.java Listagem 14

```
público void testMultipleVisitMileageReport()
{
    Veículo v = novo Veículo();
    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);
}
```

Escolhemos colocar três visitas no `Veículo`. O custo tem como base bruta $1,20 por galão e a milhagem de 30 milhas por galão (mpg). Portanto, utilizamos 9,8 galões para viajar 292 milhas a um custo de $12,24.

Há um problema estranho aqui. A leitura do hodômetro tem como base aproximadamente 30 mpg. no entanto, quando dividimos 541, a distância dirigida, por 23,1, os galões consumidos, obtemos 23,41991 mpg. Por que a discrepância? Por que não obtemos algo próximo de 30 mpg?

Depois de uma reflexão, fica claro que o consumo de combustível não é a soma de todo o combustível comprado em cada visita. O combustível é consumido *entre* as visitas. O combustível comprado na primeira visita não deve ser considerado ao calcular mpg.

TestAutoMileageLog.java Listagem 15

```
público void testMultipleVisitMileageReport()
{
    Veículo v = novo Veículo();
    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);
}
```

Isso parece melhor. Você nunca sabe o que encontrará ao gravar os testes. Uma coisa é certa—você tem mais chances de encontrar erros ao especificar coisas duas vezes, ou seja, nos testes e no código, do que se apenas escrever o código.

Agora estamos prontos para tentar incluir o código que faz com que o teste anterior seja transmitido.

Vehicle.java Listagem 16

```
público  MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();
    se (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());
    }
    diferente
    {
        int firstOdometerReading = 0;
        int lastOdometerReading = 0;
        duplo totalCost = 0;
        duplo fuelConsumption = 0;

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

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

```
        }
        retornar r;
    }
```

Esse código é feio com todos os seus casos especiais. Precisamos refatorar os casos especiais. Na verdade, o terceiro caso é geral o suficiente enquanto permanece. Devemos conseguir eliminar os outros dois casos.

Ao fazermos isso o caso de teste `testSingleVisitMileageReport` falha. O defeito ocorre porque o único caso de visita estava incluindo o combustível comprado na primeira e única visita. Como descoberto anteriormente, o consumo de combustível deve ser zero se houver apenas uma visita. Portanto, podemos corrigir o caso de teste e o código.

Vehicle.java Listagem 17

```
público MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();

    int firstOdometerReading = 0;
    int lastOdometerReading = 0;
    duplo custo total = 0;
    duplo fuelConsumption = 0;

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

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

    retornar r;
}
```

Essa função é longa. Precisamos encurtá-la e limpá-la um pouco. começaremos movendo bits do código para que eles possam ser movidos para funções separadas.

Vehicle.java Listagem 18

```
público MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();

    int distância = 0;
    duplo totalCost = 0;
    duplo fuelConsumption = 0;
    duplo firstFuel = 0;
    duplo mpg = 0;

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

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

        retornar r;
    }
```

A Listagem 18 é uma etapa intermediária. Na verdade, demorou quatro ou cinco etapas muito menores para chegar a este ponto. Em cada uma dessas etapas menores, conseguimos executar os testes para assegurar que não tínhamos quebrado nada. A meta desses refatoramentos era obter de alguma forma o código mais fácil para dividir, mas não tínhamos uma noção firme de como fazer isso. Então, esses primeiros refatoramentos foram quase aleatórios. Eles não demoraram muito tempo e os testes asseguraram que nada foi quebrado.

Tendo alcançado este ponto com os testes ainda em execução, podemos ver uma maneira de aprimorar as coisas. Começaremos dividindo o loop[2] em dois.

Vehicle.java Listagem 19

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

Os testes ainda estão em execução. A seguir, extraíremos cada loop para seu próprio método particular.[3]

[2] Consulte DIVIDIR LOOP em www.refactoring.com.

[3] Consulte EXTRAIR MÉTODO em www.refactoring.com.

Vehicle.java Listagem 20

```
público MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();

    int distância = 0;
    duplo custo total = 0;
    duplo fuelConsumption = 0;
    duplo firstFuel = 0;
    duplo mpg = 0;

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

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

    retornar r;
}

particular duplo calculateTotalCost()
{
    duplo totalCost = 0;
    for (int i=0; i<itsVisits.size(); i++)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
        totalCost += v.getCost();
    }
    retornar totalCost;
}

particular duplo calculateFuelConsumption()
{
    duplo fuelConsumption = 0;;
    for (int i=0; i<itsVisits.size(); i++)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
        fuelConsumption += v.getFuel();
    }
    retornar fuelConsumption;
}
```

Os testes ainda estão em execução. A seguir, mkoveremos os casos especiais ara consumo de combustível para o método `calculateFuelConsumption`.

Vehicle.java Listagem 21

```
público MileageReport generateMileageReport()
{
    ,,,

    se (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();

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

    ...

    retornar r;
}

particular duplo calculateTotalCost()
{
    duplo custo total = 0;
    para (int i=0; i<itsVisits.size(); i++)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
        totalCost += v.getCost();
    }
    retornar totalCost;
}

particular duplo calculateFuelConsumption()
{
    duplo fuelConsumption = 0;
    se (itsVisits.size() > 0)
    {
        para (int i=1; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            fuelConsumption += v.getFuel();
        }
    }
    retornar fuelConsumption;
}
```

Os testes ainda estão em execução. Observe que `calculateFuelConsumption` agora pode fazer o expediente de iniciar a soma do consumo de combustível com a *segunda* visita. A seguir, podemos extrair a função para calcular a distância.

Vehicle.java Listagem 22

```
público MileageReport generateMileageReport()
{
    MileageReport r =  novo MileageReport();

    int distância = 0;
    duplo custo total = 0;
    duplo fuelConsumption = 0;
    duplo firstFuel = 0;
    duplo mpg = 0;

    se (itsVisits.size() > 0)
```

```
        {
            distance = calculateDistance();
            fuelConsumption = calculateFuelConsumption();
            totalCost = calculateTotalCost();

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

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

        retornar r;
    }

    particular int calculateDistance()
    {
        int distância = 0;
        se (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;
        }
        retornar distância;
    }
```

Os testes ainda estão em execução. Agora podemos remover a condicional na função principal e limpar algumas miudezas.

Vehicle.java Listagem 23

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

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

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

        retornar r;
    }

    particular int calculateDistance()
    {
        int distância = 0;
        se (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;
        }
        retornar distância;
```

```
}

particular duplo calculateTotalCost()
{
    duplo custo total = 0;
    para (int i=0; i<itsVisits.size(); i++)
    {
        FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
        totalCost += v.getCost();
    }
    retornar totalCost;
}

particular duplo calculateFuelConsumption()
{
    duplo fuelConsumption = 0;
    se (itsVisits.size() > 1)
    {
        para (int i=1; i<itsVisits.size(); i++)
        {
            FuelingStationVisit v = (FuelingStationVisit)itsVisits.get(i);
            fuelConsumption += v.getFuel();
        }
    }
    retornar fuelConsumption;
}
```

Os testes ainda estão em execução.

Isso é muito bom. Cada função é independente e bem isolada dos outros. A função principal é pequena e fácil de entender.

Você pode argumentar que isso tornou o programa mais complicado. Enquanto a contagem da função e a contagem de linhas certamente aumentou, o programa também foi bem particionado. Cada função é fácil de entender.

Observe que a análise de caso da listagem 16 retornou, mas agora está associada às funções de cálculo específico. Isso é bem melhor que a Listagem 17 em que a remoção da análise de caso funcionou apenas por acidente.

Alguns podem reclamar que esse código é desnecessariamente lento. Isso pode ser verdade, mas não parece requerer velocidade. Quando a velocidade se torna um requisito e quando a execução atual falha esse requisito, podemos fazer algo a respeito. Até esse momento, ficaremos felizes com o esclarecimento e a separação das preocupações mostradas na Listagem 23.

Conclusão

Embora esse documento tenha demonstrado as técnicas de refatoramento na presença do teste anterior ao design; a verdadeira finalidade era conduzir uma *atitude* de programação. Um programa não está pronto quando funciona. Sem dúvida, fazer funcionar é a parte fácil. Um programa está pronto quando funciona *e* quando é o mais simples e limpo possível.

Este documento alega que uma boa maneira de alcançar esse resultado desejável é:

1. Projetar o programa gravando casos de teste. Depois de gravar cada caso de teste, gravar o código que transmite esse caso de teste. Acumular todos os testes e facilitar a execução repetidamente.
2. Assim que uma parte do programa funcionar, refatore essa parte até que fique limpo. Faça o refatoramento fazendo uma seqüência de pequenas mudanças no código e executando os testes depois de cada alteração. Isso lhe dará a confiança de que suas mudanças não estão quebrando nada e a coragem para continuar fazendo mudanças após mudanças até que o código esteja o mais limpo e claro possível.

Referências

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

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

Rational®
the software development company

Duas Sedes:

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

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

Sem custo: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
Localização Internacional:www.rational.com/worldwide