Principes et conseils RUP/XP : Conception pilotée par le test et réusinage

Robert C. Martin
Object Mentor, Inc.

Livre blanc de Rational Software

TP 159, 03/01

Rational®
the software development company

Table des matières

Présentation

Il est rare qu'une pratique véritablement révolutionnaire émerge au coeur de l'arène logicielle. La programmation structurée fut l'une d'elle. OO en fut une autre. La conception pilotée par le test et le réusinage en est encore une autre.

Une définition précise, mais un peu naïve du réusinage le qualifie d'acte consistant à effectuer des petits changements qui préservent la fonction d'un programme, mais en modifie la structure. Intégrée à cette définition, est la notion qu'un logiciel possède deux valeurs distinctes. Il y a tout d'abord une valeur dans ce que fait le logiciel. Et il y a ensuite une valeur dans la structure du logiciel. Selon la définition qui vient juste d'être donnée, le réusinage est une technique de gestion et d'amélioration de la valeur structurelle du logiciel.

Une définition plus sophistiquée du réusinage le qualifie de technique de conception et d'implémentation logicielle via une myriade de petits changements alternativement axés sur l'ajout de fonction et l'amélioration de la structure. Cette définition étend la signification du mot, décrite par Fowler dans son livre *Refactoring,* (voir référence [1]) et qui décrit le moyen par lequel un logiciel est conçu et écrit dans le processus de l'*eXtreme Programming* (XP) (voir référence [2]).

Conception pilotée par le test et réusinage est la pratique de conception, puis d'amélioration du code qui consiste à écrire des scénarios de test, avant d'écrire le code qui permet de les passer avec succès. Le programmeur sélectionne une tâche, écrit un ou deux scénarios de test d'unité très simples qui échouent puisque le programme n'effectue pas cette tâche, puis modifie le programme afin qu'il réussisse le test. Le programmeur ajoute sans cesse davantage de scénarios de test et les fait passer avec succès jusqu'à ce que le logiciel effectue toutes les tâches qu'il est supposé remplir. Ensuite, le programmeur améliore la structure du système petit à petit, exécutant tous les tests entre chaque étape pour s'assurer que rien n'est brisé dans le code.

Exemple de réusinage

Le meilleur moyen de décrire la conception pilotée par le test et le réusinage est d'utiliser un exemple. Dans le cas présent, nous allons entreprendre la conception et l'implémentation d'un petit programme, expliquant ainsi comment s'accomplit le réusinage. Notez que dans XP, une paire de programmeurs utilisant le même poste de travail accomplit les tâches que nous allons voir.[1]

L'application que nous allons construire est un simple enregistrement de distances automobiles. Chaque fois qu'un utilisateur se rend dans une station service, il saisit la quantité de carburant acheté, le prix du carburant et le chiffre indiqué au compteur kilométrique du véhicule. Le système conserve la trace de ces éléments et génère des rapports utiles. Java est le langage d'implémentation utilisé.

Nous commençons par écrire le code du listing 1 :

`TestAutoMileageLog.java` Listing 1

```
import junit.framework.*;

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

La première chose que nous écrivons est la structure destinée à contenir les tests de notre unité. Ceci peut sembler se dérouler en marche arrière, mais c'est fondamental pour le concept piloté par le test. Nous écrivons d'abord le code du test, avant d'écrire celui de l'application réelle. Vous allez voir comment cela fonctionne à mesure que nous avançons.

La structure de test que nous utilisons se nomme JUnit. Il s'agit d'une simple structure de test d'unité écrite par Kent Beck et Erich Gamma. Le code ci-dessus est tout ce dont nous avons besoin pour le configurer.

Envisageons à présent notre premier scénario de test. Que doit faire ce logiciel ? Une chose qu'il doit faire est d'enregistrer les visites aux stations service. Ceci implique qu'il doit y avoir un objet `FuelingStationVisit` contenant les données pertinentes. Nous pouvons donc écrire un test qui crée cet objet et interroge ensuite ses champs.

[1] Reportez-vous au livre blanc de Rational Software intitulé *Principes et conseils RUP®/XP : Programmation par paire.*

Pour cela, nous commençons par écrire une fonction test. Dans JUnit, ce type de fonction correspond à n'importe quelle méthode de classe dérivée du `Scénario de test` dont le nom commence par les quatre lettres "`test`". Voir listing 2.

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();
    }
}
```

Le nouveau code est en caractère gras. Notez que tout ce que nous avons fait est de créer un objet nommé `FuelingStationVisit`. Nous ne lui avons pas encore donné d'argument de construction. Tout ce qui nous intéresse à ce point, est de s'assurer que nous pouvons créer l'objet.

Clairement, ceci ne se compilera pas (bien que ce fut intéressant qu'il le fasse). Pou qu'il se compile, nous devons écrire le code de l'objet `FuelingStationVisit` . Voir 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
{
}
```

Ce code se compile et le test s'exécute. Nous pouvons alors ajouter la fonctionnalité souhaitée.

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;}
}
```

Cette étape a été réalisée en ajoutant tout d'abord les tests à `TestAutoMileageLog` , puis en ajoutant les méthodes à `FuelingStationVisit`. Trois ou quatre compilations ont été impliquées avant d'être prêt pour les tests. Les tests se sont déroulés correctement dès la première fois.

Vous vous demandez sûrement pourquoi cette extrême incrémentation nous captive. N'était-il pas possible d'écrire simplement `FuelingStationVisit` et le code du test ensuite ? Est-il véritablement nécessaire de tester `FuelingStationVisit` ? Jusqu'à présent, l'écriture préalable des tests, ou même leur écriture tout simplement, nous a apporté que très peu d'avantage, sauf un. Nous savons, sans équivoque, que le code ci-dessus se compile et s'exécute. Nous savons de ce fait que si la modification suivante aboutit à des erreurs du programme de compilation, ou à l'échec des tests, le problème réside dans cette modification précise, et non dans le code antérieur. Ceci peut paraître un mince avantage, mais il deviendra bien plus important ultérieurement.

Ensuite, nous devons placer des objets `FuelingStationVisit` quelque part. Des objets doivent les contenir. Quel objet cela doit-il être ? C'est l'*utilisateur* qui souhaite conserver et gérer ces informations, aussi devons-nous créer un objet `User` pour contenir les objets `FuelingStationVisit`. Cependant, le champ du kilométrage dans l'objet `FuelingStationVisit` me fait m'interroger. Le kilométrage est l'attribut d'un véhicule. L'objet `FuelingStationVisit` enregistre une partie de l'état d'un `véhicule` au moment de la visite. De ce fait, nous devons créer un objet `Vehicle` et placer les objets `FuelingStationVisit` à l'intérieur.

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;
    }
}
```

Le listing 5 montre l'étape initiale. Nous avons crée une autre fonction de test, nommée `testCreateVehicle`. Cette fonction crée un `Vehicle`, puis s'assure que le nombre de visites indiqué est à zéro. L'implémentation de l'objet `getNumberOfVisits` est clairement fausse, mais elle a le bénéfice de faire passer le test avec succès. Ceci nous permet de reformuler la modification (refactor) en une meilleure solution.

Vehicle.java Listing 6

```
import java.util.Vector;

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

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

Le test s'est de nouveau déroulé avec succès. Il est à noter que nous exécutons tous les tests, et pas seulement la fonction `testCreateVehicle`. Ceci nous assure que les modifications que nous avons apportées n'ont rien cassé de ce qui fonctionnait auparavant.

Ensuite, nous devons envisager comment ajouter une visite à un véhicule. A quoi le scénario de test le plus simple doit-il ressembler ?

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());
    }
```

Notez que nous n'avons pas créé d'objet `FuelingStationVisit` dans ce test. Il semble que la méthode `addFuelingStationVisit` du `Vehicle` doive créer l'objet `FuelingStationVisit`, puis l'ajouter à la liste.

`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);
    }
```

A nouveau, tous les tests se sont déroulés avec succès.

Le code en double dans les deux fonctions `testAddVisit` et `testCreateFuelingStationVisit` devrait nous gêner. Les deux fonctions créent les mêmes variables locales et les initialisent aux même valeurs. Nous devons nous débarrasser de ce doublon. Nous allons donc améliorer le programme de test en transformant les variables locales en variables membres.

`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());
    }
}
```

Cette opération spécifique de réusinage porte un nom. Il s'agit de PROMOTE TEMP TO FIELD. Une liste d'opérations de réusinage et de procédures pour les appliquer figure à la référence [1] et sur le site www.refactoring.com.

Notez que l'existence des tests d'unité permet de vérifier rapidement que cette opération n'a rien cassé. Nous allons continuer à en tirer parti à mesure que nous réusinons et restructurons l'application. Chaque fois que nous faisons quelque chose au code qui nous rend incertain, nous pouvons nous appuyer sur les tests pour nous assurer que tout fonctionne toujours parfaitement.

Après avoir ajouté les objets `FuelingStationVisit` à l'objet `Vehicle`, nous pouvons dès lors demander à l'objet `Vehicle` de produire des rapports. Nous écrivons d'abord les scénarios de test, en commençant par le plus simple.

`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());
    }
```

Pour écrire ce scénario de test, nous avons dû réfléchir aux problèmes liés à la génération de rapports. Nous avons tout d'abord décidé que l'objet `Vehicle` doit disposer d'une méthode nommée `generateMileageReport`. Ensuite, nous avons décidé que cette fonction doit renvoyer un objet nommé `MileageReport`. Enfin, nous avons décidé que l'objet `MileageReport` doit disposer de plusieurs méthodes de requête.

Les valeurs retournées par ces méthodes de requête sont très intéressantes. Une seule visite est insuffisante pour calculer les kilomètres (miles, dans cet exemple) ou les kilomètres par litre (miles par gallon dans notre exemple). Pour calculer ces valeurs, au moins deux visites sont nécessaires. D'autre part, une seule visite est suffisante pour calculer la consommation et le prix du carburant.

Bien sûr, le scénario de test ne se compile pas. Pour cela, nous devons ajouter les méthodes et les classes appropriées. Nous ajoutons en premier juste ce qu'il faut de code pour le faire se compiler, mais échouer aux 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;
}
```

Le code du listing 11 se compile et s'exécute, mais le test échoue. A présent, nous devons modifier le code de sorte qu'il passe le test avec succès. Nous devons prendre en premier l'approche la plus simple 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;}
```

Nous présumons que l'objet `Vehicle` n'a qu'une visite. Ne vous inquiétez pas ! Nous allons ajouter d'autres scénarios de test pour d'autres situations ultérieurement. Nous définissons les champs de `MileageReport` de manière adéquate, puis le renvoyons.

Il peut paraître stupide d'implémenter `generateMileageReport` de cette façon puisque nous sommes certains qu'au mieux, son implémentation est incomplète. Cependant, l'implémentation par petits incréments présente l'avantage que rien ne doit changer entre chaque compilation et test. Si quelque chose se passe mal, il est toujours possible de revenir à la version antérieure et de recommencer. Il n'est pas nécessaire de déboguer.

Le code du listing 12 se compile et passe les tests avec succès, mais il est évident qu'il est incomplet. Pour le terminer, nous devons penser à d'autres scénarios de test.

- Un véhicule sans visite
- Un véhicule avec plusieurs visites

Le cas dans lequel il n'y a pas de visite est simple. Le scénario de test échoue dans le listing 13.1 et le code contenu dans le listing 13.2 le fait passer.

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;
}
```

Ensuite, nous devons envisager le scénario de test correspondant à plusieurs visites.

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);
}
```

Nous avons choisi de mettre trois visites dans l'objet `Vehicle`. Nous avons basé le prix sur environ $1.20 par gallon et le kilométrage à environ 30 miles par gallon (mpg). Aussi, utilisons-nous 9.8 gallons pour effectuer 292 miles au prix de $12.24.

Nous sommes confrontés là à un problème singulier. Chaque lecture de compteur kilométrique est basée sur environ 30 mpg. Cependant, si nous divisons 541, la distance obtenue, par 23.1, les gallons de carburant consommé, nous obtenons 23.41991 mpg. Pourquoi une telle différence ? Pourquoi n'obtenons-nous pas quelque chose de proche de 30 mpg ?

En réfléchissant, il devient clair que la consommation de carburant ne correspond pas à la somme de tout le carburant acheté à chaque visite. Le carburant est consommé *entre* les visites. Le carburant acheté à la première visite ne doit pas être pris en compte lors du calcul.

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);
}
```

Ceci semble beaucoup mieux. Vous ne savez jamais ce que vous allez trouver lorsque vous écrivez des tests. Une chose est sûre cependant, vous êtes susceptible de trouver plus d'erreurs lorsque vous spécifiez les choses deux fois, autrement dit, dans les tests et le code, que si vous spécifiez uniquement le code.

Maintenant, nous sommes prêts à tenter d'ajouter le code qui a fait passer le test précédent.

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;
    }
```

Ce code est horrible avec tous ses cas particuliers. Nous devons donc le modifier pour les supprimer. En fait, le troisième cas est suffisamment général tel qu'il est. Nous devrions donc pouvoir éliminer les deux autres cas.

Si nous procédons ainsi, le scénario de test `testSingleVisitMileageReport` échoue. L'échec est dû au fait que le cas d'une seule visite contenait la quantité de carburant achetée au cours de la première visite. Comme nous l'avons découvert ci-dessus, la consommation de carburant doit être à zéro s'il n'y a qu'une seule visite. Ainsi, nous pouvons corriger le scénario de test et le 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;
}
```

Cette fonction est longue. Nous devons la raccourcir et la nettoyer un peu. Nous allons commencer par bouger des morceaux de code de sorte qu'ils puissent être déplacés en fonctions distinctes.

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;
    }
```

Le listing 18 est une étape intermédiaire. Il a fallu en fait quatre ou cinq plus petites étapes pour arriver à ce point. A chacune de ces plus petites étapes, nous avons pu exécuter les tests pour nous assurer que nous n'avions rien cassé. L'objectif de ces réusinages était d'obtenir que le code soit plus simple à partager, mais nous n'avions aucune certitude sur le moyen d'y parvenir. Ces premiers réusinages ont été faits presque au hasard. Ils n'ont pas pris beaucoup de temps et les tests ont prouvé que rien n'était cassé.

Ayant atteint ce point alors que les tests sont toujours en cours d'exécution, nous entrevoyons un moyen pour améliorer les choses. Nous allons commencer par fractionner la boucle[2] en deux.

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;
        }
```

Les tests s'exécutent toujours. Ensuite, nous allons extraire chaque boucle dans sa propre méthode privée. [3]

[2] Voir SPLIT LOOP sur le site www.refactoring.com.

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();
    }
    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;
}
```

Les tests s'exécutent toujours. Ensuite, nous allons déplacer les cas particuliers pour la consommation de carburant dans la méthode `calculateFuelConsumption`.

[3] Voir EXTRACT METHOD sur le site www.refactoring.com.

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;
}
```

Les tests s'exécutent toujours. Notez qu'à présent `calculateFuelConsumption` peut trouver opportun de commencer à calculer la consommation de carburant à la *seconde* visite. Ensuite, nous pouvons extraire la fonction pour calculer la 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;
    }
```

Les tests s'exécutent toujours. A présent, nous pouvons supprimer la condition dans la fonction principale et nettoyer quelques points qui traînent.

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;
    }
```

Les tests s'exécutent toujours.

C'est plutôt bien. Chaque fonction est autonome et bien isolée des autres. La fonction principale est petite et facile à comprendre.

Vous pouvez cependant déclarer que ceci a rendu le programme plus compliqué. La fonction et la ligne de calcul ont effectivement pris de l'ampleur, mais le programme est désormais judicieusement fractionné. Chaque fonction est aisément compréhensible.

Notez que l'analyse du cas du listing 16 a été renvoyée, mais est désormais associée aux fonctions de calcul spécifiques. Ceci est beaucoup mieux que dans le listing 17 dans lequel la suppression de l'analyse du cas ne fonctionnait que par accident.

Certains trouveront que ce code est inutilement lent. Cela est peut-être vrai, mais il ne nous semble pas qu'il soit nécessaire qu'il soit plus rapide. Dans le cas contraire, et si l'exécution courante ne répond pas à cette exigence, nous pouvons y remédier. Jusque là, nous nous contenterons de la clarté et de la séparation des points mentionnés dans le listing 23.

Conclusion

Bien que ce document ait expliqué les techniques du réusinage en présence de la conception pilotée par le test, son objectif réel était de faire comprendre une *attitude* de programmation. Un programme n'est pas terminé parce qu'il fonctionne. Le faire fonctionner est assurément chose facile. Un programme est terminé lorsqu'il fonctionne *et* lorsqu'il est aussi simple et propre que possible.

Ce document déclare qu'un bon moyen pour y parvenir est de procéder comme suit :

1. Concevez le programme en écrivant des scénarios de test. Une fois chaque scénario de test écrit, rédigez le code qui permet de passer le test avec succès. Cumulez tous les tests et simplifiez leur exécution répétée.
2. Dès qu'une partie du programme fonctionne, modifiez-la jusqu'à ce qu'elle soit propre. Procédez au réusinage en apportant de petites modifications au code et en exécutant les tests après chaque changement. Ceci vous permet de vous assurer que vos modifications ne cassent rien et vous donne le courage de poursuivre vos modifications, l'une après l'autre, jusqu'à ce que le code soit aussi net et clair que possible.

Références

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

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

Rational
the software development company

Sièges :

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

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

Appel gratuit : (800) 728-1212
Adresse électronique : info@rational.com
Site Web : www.rational.com
Sites internationaux : www.rational.com/worldwide

Rational, le logo Rational et Rational Unified Process sont des marques de Rational Software Corporation aux Etats-Unis et/ou dans certains autres pays. Microsoft, Microsoft Windows, Microsoft Visual Studio, Microsoft Word, Microsoft Project, Visual C++ et Visual Basic sont des marques de commerce ou des marques déposées de Microsoft Corporation. Tous les autres noms ne sont utilisés qu'à des fins d'identification et sont des marques de commerce ou des marques déposées de leurs sociétés respectives. TOUS DROITS RESERVES. Rédigé aux Etats-Unis.

© Copyright 2002 Rational Software Corporation.
Document susceptible d'être modifié sans préavis.