# Linee guida per RUP«/XP: Test-first Design e Refactoring

**Robert C. Martin**
**Object Mentor, Inc.**

White Paper del Software Rational

TP 159, 03/01

Rational®
the software development company

# Indice

## Panoramica

Succede raramente che una pratica davvero rivoluzionaria emerga nell'arena del software. La programmazione strutturata ne rappresentava un esempio, OO un altro. Test-first design e refactoring un altro ancora.

Una definizione particolareggiata ma na´ve di refactoring consiste nell'apportare modifiche minori che preservano una funzione di programma anche se ne cambiano la struttura. L'idea che esistono due valori distinti per software è rimasta intrappolata in questa definizione. In primo luogo, esiste un valore in ciò che fa il software. In secondo luogo, la struttura del software possiede valore. Secondo le definizioni date, refactoring è una tecnica per gestire e migliorare il valore strutturale del software.
Una definizione più sofisticata di refactoring è rappresentata dalla tecnica di progettare e implementare il software mediante una miriade di modifiche minori che si focalizzano in modo alternato sull'aggiunta di strutture di funzione e implementazione. Questa definizione amplia il significato delle parole di Fowler nel suo libro Refactoring, (vedere riferimento [1]) e descrive il modo in cui il software viene progettato e scritto nel processo di XP(eXtreme Programming) (vedere riferimento [2]).

Test-first design e refactoring è la pratica di progettare e poi migliorare il codice scrivendo scenari di test prima del codice che ne consente il passaggio. Il programmatore seleziona un compito, scrive uno o due scenari di test delle unità molto semplici, che falliscono perché il programma non esegue tale compito, e poi modifica il programma per consentire di superare i test. Poi aggiunge ininterrottamente scenari di test e ne consente il superamento, fin quando il software non agisce nel modo stabilito. Infine il programmatore migliora la struttura del sistema un passo per volta, eseguendo tutti i test fra ciascun passo per assicurarsi che nulla si è danneggiato.

## Un esempio di refactoring

Il modo migliore per descrivere test-first design e refactoring è facendo degli esempi. Verrà illustrato il modo in cui progettare e implementare un piccolo programma, dimostrando come viene realizzato il refactoring. Notare che in XP, una coppia di programmatori che utilizza la stessa workstation dovrebbe realizzare le attività che verranno affrontate in questo documento.[1]

L'applicazione da costruire è un semplice log di chilometraggio automatico. Ogni volta che un utente si ferma ad un distributore di benzina, inserisce la quantità di carburante da acquistare, il costo e la lettura attuale del contachilometri del veicolo. Il sistema tiene traccia di questi elementi e crea un certo numero di report utili. Il linguaggio di implementazione sarà Java.

Iniziare scrivendo il codice nell'Elenco 1:

TestAutoMileageLog.java Elenco 1

```
import junit.framework.*;

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

In primo luogo, scrivere il framework per contenere i test delle unità. Potrebbe sembrare di ricominciare di nuovo, ma questo passo è fondamentale per
il concetto test-first. Scrivere il codice di test prima dell'attuale codice di applicazione. È possibile vedere come lavora mentre si procede.

Il framework di test in uso si chiama JUnit, è un semplice framework per la verifica delle unità scritto da Kent Beck e Erich Gamma e rappresenta tutto ciò di cui si ha bisogno per impostarlo.

È necessario considerare il primo scenario di test. Cosa è richiesto a questo software? Registrare coloro che si fermano al distributore di benzina. Ciò implica che deve esserci un oggetto `FuelingStationVisit` che raccoglie i dati pertinenti. In questo modo è possibile scrivere un test che crea questo oggetto e poi fa ricerche nei relativi campi.

Iniziare scrivendo una funzione di test. In JUnit, una funzione di test è ogni metodo di una classe derivata dallo `Scenario di test` il cui nome comincia con le quattro lettere “`test`”. Vedere Elenco 2.

[1] Consultare il white paper del Software Rational intitolato Linee giuda per RUP«/XP: Programmazione a coppia.

TestAutoMileageLog.java Elenco 2

```
import junit.framework.*;

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

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

Il nuovo codice è in grassetto. Notare che ciò che è stato fatto è creare un nuovo oggetto denominato `FuelingStationVisit`, senza avergli ancora dato argomenti di costruzione. A questo punto, la cosa importante è assicurarsi di poter creare l'oggetto, che ovviamente non si compila (sebbene dovrebbe essere interessato se lo ha fatto). Per consentirglielo, è necessario scrivere il codice relativo all'oggetto `FuelingStationVisit`. Vedere Elenco 3.

TestAutoMileageLog.java Elenco 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 Elenco 3.2

```
public class FuelingStationVisit
{
}
```

Questo codice si compila mentre il test si esegue; pertanto è possibile aggiungere la funzionalità desiderata.

TestAutoMileageLog.java Elenco 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 Elenco 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;}
 }
```

Questo passo è stato fatto aggiungendo prima i test a `TestAutoMileageLog` e poi i metodi a `FuelingStationVisit`. È stato necessario realizzare tre o quattro compilazioni prima di poter accedere alla fase di test. I test sono stati eseguiti la prima volta.

È necessario chiedersi a cosa porterà questo eccessivo incrementalismo. Non è possibile aver già scritto `FuelingStationVisit` e in seguito il codice di test? Si può non verificare affatto `FuelingStationVisit`? Scrivere prima i test, o non scriverli affatto, ha prodotto un vantaggio molto piccolo, tranne uno. Senza dubbio,
il codice summenzionato si compila e si esegue. Se la modifica successiva determina errori del compilatore o il test fallisce, il problema deve essere ricercato nella modifica e non nel codice precedente. Potrebbe sembrare un piccolo vantaggio, ma in seguito diventerà più importante.

È necessario che gli oggetti `FuelingStationVisit` vengano inseriti da qualche parte e che altri oggetti li contengano. Quali dovrebbero essere questi oggetti? L' utente desidera acquisire e gestire queste informazioni, così da creare un oggetto `utente` per contenere gli oggetti `FuelingStationVisit`. Ad ogni modo, il campo relativo al chilometraggio nell'oggetto `FuelingStationVisit` solleva alcune domande. Il chilometraggio è un attributo di un veicolo. L'oggetto `FuelingStationVisit` registra parte dello stato di `Vehicle` al momento della sosta al distributore. Pertanto, è necessario creare un oggetto `Vehicle` e contenere l'oggetto `FuelingStationVisit` all'interno.

TestAutoMileageLog.java Elenco 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 Elenco 5.2

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

L'Elenco 5 mostra la fase iniziale. È stata creata una funzione di test denominata `testCreateVehicle`, che a sua volta genera un `Vehicle` e si assicura che il numero delle soste contenuto all'interno sia pari a zero. L'implementazione di `getNumberOfVisits` è errata, ma consente di superare il test. Questo permette di eseguire il refactoring del test con una soluzione migliore.

Vehicle.java Elenco 6

```
import java.util.Vector;
public class Vehicle
{
    private Vector itsVisits = new Vector();

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

I test vengono superati nuovamente. È necessario notare che si stanno eseguendo tutti i test, non solo la funzione `testCreateVehicle`, assicurandosi in questo modo che le modifiche apportate non hanno danneggiato niente di ciò che è stato utilizzato per il lavoro.

In seguito, sarà necessario comprendere il modo in cui aggiungere una sosta a Vehicle. A cosa dovrebbe somigliare il più semplice scenario di test?

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

Risulta utile notare che, in questo test, non è stato creato un oggetto `FuelingStationVisit`. Sembra che il metodo `addFuelingStationVisit` di `Vehicle` deve creare l'oggetto `FuelingStationVisit` e poi aggiungerlo all'elenco.

Vehicle.java Elenco 8

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

Tutti i test vengono superati nuovamente.

Il codice duplicato nelle due funzioni `testAddVisit` e `testCreateFuelingStationVisit` potrebbe creare alcuni problemi, dal momento che entrambe le funzioni generano le stesse variabili locali e le inizializzano agli stessi valori.
È necessario liberarsi di questo duplicato. Perciò, verrà eseguito il refactoring del programma di test, trasformando le variabili locali in variabili di membro.

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

Questo particolare refactoring ha un nome. Viene chiamato PROMOTE TEMP TO FIELD. È possibile trovare un elenco di refactoring simili e le procedure per applicarli nei riferimenti [1] e sul sito www.refactoring.com.

È utile notare che i test delle unità consentono di verificare velocemente che questo refactoring non ha danneggiato nulla. Si otterranno maggiori vantaggi man mano che si esegue il refactoring e si ristruttura l'applicazione. Ogni volta che si apporta una modifica poco convincente al codice, si ritorna alla fase di test per assicurarsi che tutto funzioni bene.

Dopo che sono stati aggiunti gli oggetti `FuelingStationVisit` al `Vehicle`, è possibile chiedere al `Vehicle` di produrre report. Scrivere prima gli scenari di test, iniziando con il caso più semplice.

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

Per scrivere questo scenario di test, è necessario pensare tramite le problematiche legate alla generazione del report. In primo luogo, si stabilisce che `Vehicle` deve possedere un metodo denominato `generateMileageReport` e poi che questa funzione deve restituire un oggetto denominato
`MileageReport`. Infine, si stabilisce che `MileageReport` deve avere numerosi metodi di query.
I valori restituiti da questi metodi di query sono molto interessanti. Una sola sosta al distributore non è sufficiente a calcolare le miglia percorse o le miglia per gallone, ma sono necessarie almeno due soste. D'altra parte, una sola sosta è sufficiente a calcolare il consumo di carburante e il relativo costo.

Naturalmente, lo scenario di test non si compila. Pertanto, è necessario aggiungere le classi e i metodi appropriati. Si aggiunge prima codice sufficiente a consentirgli la compilazione, ma i test falliscono.

Vehicle.java Elenco 11.1

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

TestAutoMileageLog.java Elenco 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 Elenco 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;
 }
```

Nell'Elenco 11, il codice si compila e si esegue, ma il test fallisce. Adesso è necessario eseguire il refactoring del codice per consentirgli di superare il test. All' inizio, si sceglie l'approccio più semplice.

Vehicle.java Elenco 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 Elenco 12.2

```
    public void setMilesPerGallon(double mpg) {itsMilesPerGallon = mpg;} public
    void setMilesDriven(int miles) miles) {itsMilesDriven=miles;}
    {itsMilesDriven=miles;} public void setTotalFuelCost(double cost)
    {itsTotalFuelCost=cost;}
    public void setFuelConsumed(double fuel) fuel) {itsFuelConsumed=fuel;} {itsFuelConsumed=fuel;}
```

Si suppone che `Vehicle` fa una sola sosta (altri scenari di test verranno aggiunti in seguito per altre condizioni). Si impostano in modo appropriato i campi di `MileageReport` che poi si restituisce.

Sembrerebbe inutile implementare in questo modo `generateMileageReport` poiché sicuramente l'implementazione risulta, nell'ipotesi migliore, incompleta. Ad ogni modo, è vantaggioso implementare in incrementi minori, perché non si apportano modifiche fra ciascuna fase di compilazione e test. Se qualcosa va storto, è possibile ritornare all'ultima versione e iniziare di nuovo. Non è necessario eseguire il debug.

Nell'Elenco 12, il codice si compila e supera i test, ma è evidente che risulta incompleto. Per completarlo, è necessario pensare ad altri scenari di test.

- Un Vehicle senza soste
- Un Vehicle con più di una sosta

Lo scenario in cui non ci sono soste risulta semplice, mentre lo scenario di test nell'Elenco 13.1 fallisce e il codice nell'Elenco 13.2 gli consente nuovamente di essere superato.

TestAutoMileageLog.java Elenco 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 Elenco 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;
}
```

Successivamente, sarà necessario considerare lo scenario di test che gestisce molte soste.

TestAutoMileageLog.java Elenco 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);
 }
```

Sono state inserite tre soste al distributore in `Vehicle`. Il costo è stato fissato approssimativamente a $1.20 per gallone e il chilometraggio a circa 30 miglia per gallone (mpg). Pertanto, vengono utilizzati 9.8 galloni per percorrere 292 miglia al prezzo di $12.24.

Si è verificato un problema di incongruenza. Ogni lettura del contachilometri è stata fissata a circa 30 miglia per gallone. Ad ogni modo, dividendo 541, ovvero la distanza percorsa, per 23.1, ovvero i galloni consumati, si ottengono 23.41991 miglia per gallone. Perché questa discrepanza? Perché non si ottengono circa 30 miglia per gallone?

Riflettendo, risulta evidente che il consumo di carburante non è la somma di tutto il carburante acquistato ad ogni sosta. Il carburante viene consumato anche fra le soste, perciò quello acquistato alla prima sosta non deve essere considerato nel calcolo delle miglia per gallone.

TestAutoMileageLog.java Elenco 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);
 }
```

Sembra che vada molto meglio. Non si è mai sicuri di ciò che si trova quando si scrivono test. Sicuramente, si riscontrano molti più errori quando le cose vengono specificate due volte, cioè nei test che nel codice, piuttosto che quando viene scritto soltanto il codice. Ora è possibile cercare di aggiungere il codice che consente di superare il test precedente.

Vehicle.java Elenco 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;
    }
```

Questo codice risulta inadatto ai relativi scenari speciali, dei quali è necessario eseguire il refactoring. Infatti, il terzo scenario è adeguato così come si presenta, mentre è possibile eliminare gli altri due.

Quando ciò accade, lo scenario di test `testSingleVisitMileageReport` fallisce. Il fallimento avviene perché il carburante acquistato nella prima, ed unica, sosta stava per essere incluso nello scenario dell'unica sosta. Come illustrato precedentemente, il consumo di carburante deve essere pari a zero se si effettua solo una sosta al distributore. Pertanto, è possibile apportare correzioni allo scenario di test e al codice.

Vehicle.java Elenco 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;
}
```

Questa funzione risulta lunga. È necessario renderla più breve ed eliminare un po' di errori, facendo muovere i bit del codice in modo che possano essere spostati in funzioni separate.

Vehicle.java Elenco 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;
    }
```

L'Elenco 18 rappresenta una fase intermedia. Per arrivare a questo punto, ha eseguito quattro o cinque passi più piccoli, in corrispondenza dei quali è possibile eseguire i test per assicurarsi che nulla è stato danneggiato. L'obiettivo di questi refactoring era semplificare il codice per poterlo suddividere, ma non si aveva una nozione precisa del modo in cui avveniva tutto ciò. Le prime esecuzioni di refactoring erano piuttosto casuali. Non occupavano molto tempo e i test assicuravano che nulla si fosse danneggiato.

Avendo raggiunto questo punto con i test ancora in esecuzione, è possibile cercare un modo per migliorare le cose. Si inizierà con la suddivisione del loop[2] in due.

Vehicle.java Elenco 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;
        }
```

I test sono ancora in esecuzione. In seguito, si ricaverà ogni loop all'interno del relativo metodo privato. [3]

[2] Vedere SPLIT LOOP sul sito www.refactoring.com.

[3] Vedere EXTRACT METHOD sul sito www.refactoring.com.

Vehicle.java Elenco 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;
    }
```

I test sono ancora in esecuzione. Successivamente, gli scenari speciali per il consumo di carburante saranno spostati nel metodo `calculateFuelConsumption`.

Vehicle.java Elenco 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;
}
```

I test sono ancora in esecuzione. Risulta utile notare che `calculateFuelConsumption` potrebbe ora iniziare a fare la somma tra il consumo di carburante e la seconda sosta. In seguito, sarà possibile estrapolare la funzione per calcolare la distanza.

Vehicle.java Elenco 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;
    }
```

I test sono ancora in esecuzione. Ora è possibile togliere le variabili nelle funzioni principali ed eliminare alcune delle cose superflue.

Vehicle.java Elenco 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;
}
```

I test sono ancora in esecuzione.

Adesso va piuttosto bene. Ogni funzione viene contenuta al suo interno e risulta ben isolata dalle altre. La funzione principale è piccola e di facile comprensione.

È possibile affermare che tutto ciò ha complicato maggiormente il programma. Mentre il conteggio della funzione e quello della riga sono stati incrementati, è stato eseguito il partizionamento del programma. Ciascuna funzione è di facile comprensione.

Notare che l'Elenco 16 ha restituito l'analisi dello scenario, che ora è stata associata alle funzioni di calcolo specifiche. Questo è meglio dell'Elenco 17, in cui l'eliminazione dell'analisi dello scenario ha funzionato soltanto in modo accidentale.

Qualcuno potrebbe lamentarsi che il codice risulta lento. Ciò può essere vero, ma non sembra acquisire velocità. Quando la velocità diventa un requisito e l'esecuzione attuale non riesce a superarlo, soltanto allora si può fare qualcosa. Fino a quel momento, la chiarezza e la separazione delle considerazioni illustrate nell'Elenco 23 risulteranno soddisfacenti.

## Conclusioni

Sebbene questo documento abbia mostrato le tecniche di esecuzione del refactoring in presenza di test-first design, lo scopo effettivo era trasmettere un atteggiamento di programmazione. Un programma non può considerarsi ultimato finché non se ne verifica il funzionamento. Al contrario, farlo funzionare è la parte più semplice. Un programma non può considerarsi ultimato finché non se ne verifica il funzionamento e quando è estremamente e privo di errori.

Questo documento afferma che il modo migliore per raggiungere il risultato tanto atteso è:

1. Progettare il programma scrivendo gli scenari di test. Dopo che ogni scenario di test è stato scritto, utilizzare il codice che supera tali scenari. Accumulare tutti i test e semplificare la loro esecuzione ripetutamente.

2. Una volta che si è certi del funzionamento di una parte del programma, eseguire il refactoring di quella parte fin quando non risulta priva di errori, apportando modifiche minori in sequenza al codice ed eseguendo i test dopo ogni modifica effettuata. Risulterà utile per essere sicuri che le modifiche non danneggiano nulla e per avere il coraggio di continuare ad apportare modifiche su modifiche fino a quando il codice non risulterà privo di errori così come lo si desidera.

## Riferimenti

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

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

Rational®
the software development company

Sedi principali:

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

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

Numero verde: (800) 728-1212
E-mail: info@rational.com
Sito Web: www.rational.com
Sito internazionale: www.rational.com/worldwide