RUP/XP 가이드라인: 테스트 우선 설계 및 리팩토링

Robert C. Martin
Object Mentor, Inc.

Rational Software 백서

TP 159, 03/01

Rational®
the software development company

목차

개요

소프트웨어 업계에서 정말로 혁신적인 교범이 표면화되는 일은 거의 없습니다. 이와 같은 교범 중 하나가 구조화된 프로그래밍입니다. OO 가 또 다른 교범입니다. 테스트 우선 설계 및 리팩토링도 이와 같은 교범입니다.

리팩토링의 정확한 정의는 구조를 변경하지만 프로그램의 기능은 유지하는 사소한 변경을 의미합니다. 이 정의에 내포된 것은 소프트웨어에 대한 두 가지 별도의 가치가 있는 개념입니다. 첫 번째, 소프트웨어가 수행하는 것에 대한 가치가 있습니다. 두 번째, 소프트웨어의 구조에 대한 가치가 있습니다. 제공되는 정의에만 따르면, 리팩토링은 소프트웨어의 구조적 가치를 유지보수하고 개선하기 위한 설명입니다.

더욱 상세한 리팩토링의 정의는 번갈아 가면서 기능을 추가하고 구조를 개선하는 데 중점을 두는 수 많은 사소한 변경을 통해 소프트웨어를 설계하고 구현하는 설명입니다. 이 정의는 Fowler 의 책, *Refactoring*(참조서 [1] 참조)에서 설명한 단어의 의미를 확장하며 소프트웨어가 *eXtreme Programming*(XP)(참조서 [2] 참조)의 프로세스에 설계되고 쓰여지는 방법을 설명합니다.

테스트 우선 설계 및 리팩토링은 테스트 케이스를 전달하는 코드를 작성하기 전에 테스트 케이스를 작성하여 코드를 설계한 후 코드를 개선하는 실행입니다. 프로그래머는 타스크를 선택하고, 프로그램이 이 타스크를 수행하지 않아서 실패한 하나 또는 두 개의 단순한 단위 테스트 케이스를 작성합니다. 그런 다음, 테스트가 전달될 수 있도록 프로그램을 수정합니다. 소프트웨어가 수행하기로 되어 있는 모든 것을 수행할 때까지 프로그래머는 계속해서 테스트 케이스를 추가하고 이것을 전달합니다. 그런 다음, 프로그래머는 중단된 것은 없는지 확인하기 위해 각 단계 간에 모든 테스트를 실행하여 한 번에 작은 한 단계씩 시스템의 구조를 개선합니다.

리팩토링 예

테스트 우선 설계 및 리팩토링을 설명하는 가장 좋은 방법은 예를 통해서입니다. 따라서 이 문서에서는 작은 프로그램을 설계 및 구현하여 리팩토링이 어떻게 이루어지는지 시연합니다. XP 에서 동일한 워크스테이션을 사용하고 있는 프로그래머 짝이 이제 보려는 활동을 완수할 것임에 유의하십시오.[1]

빌드될 어플리케이션은 단순한 자동 마일리지 로그입니다. 사용자가 주유소를 방문할 때마다 구매된 연료량, 연료의 비용 및 자동차에서 읽은 현재 주행 거리를 기록합니다. 시스템은 계속하여 이런 항목을 추적하고 유용한 보고서를 생성합니다. 구현 언어는 Java 입니다.

목록 1 에 코드를 작성하여 시작합니다.

TestAutoMileageLog.java 목록 1

```
import junit.framework.*;

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

첫 번째로 작성한 것은 단위 테스트를 포함할 프레임워크입니다. 이것은 진행을 늦추는 것으로 보일 수 있지만 테스트 우선 개념의 기초입니다. 실제 어플리케이션 코드를 작성하기 전에 테스트 코드를 먼저 작성합니다. 진행함에 따라 이 프레임워크가 작동하는 방법을 알게 됩니다.

[1] *RUP /XP 가이드라인: 페어 프로그래밍이라는 Rational Software 백서를 참조하십시오..*

사용 중인 테스트 프레임워크를 JUnit 이라고 하며 이것은 Kent Beck 및 Erich Gamma 가 작성한 단순 단위 테스트 프레임워크입니다. 위의 코드는 이 프레임워크 설정에 필요한 모든 것입니다.

이제 첫 번째 테스트 케이스를 고려해야 합니다. 이 소프트웨어가 무엇을 수행합니까? 수행해야 할 한 가지는 주요소 방문 기록입니다. 이것은 관련된 데이터를 보유하는 FuelingStationVisit 객체가 있어야 함을 의미합니다. 따라서 이 객체를 작성한 후 이의 필드를 조회하는 테스트를 작성할 수 있습니다.

테스트 함수를 작성하여 이를 시작합니다. JUnit 에서 테스트 함수는 TestCase 에서 파생된 클래스의 메소드이며 이름이 "test"라는 네 문자로 시작됩니다. 목록 2 를 참조하십시오.

TestAutoMileageLog.java 목록 2

```
import junit.framework.*;

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

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

새 코드는 굵은체로 표시되어 있습니다. 지금까지 수행한 모든 것은 FuelingStationVisit 라는 새로운 객체를 작성한 것임에 주의하십시오. 아직은 어떤 구축 논의도 제공하지 않았습니다. 이 지점에서 관심 있는 것은 객체를 작성할 수 있는지 확인하는 것입니다.

분명히, 이것은 컴파일되지 않습니다. (컴파일되었다면 흥미로울 것 입니다.) 컴파일되게 하려면 FuelingStationVisit 객체에 대한 코드를 작성해야 합니다. 목록 3 을 참조하십시오.

TestAutoMileageLog.java 목록 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 목록 3.2

```
public class FuelingStationVisit
{
```

```
}
```

이 코드가 컴파일되고 테스트가 실행되어 원하는 기능을 추가시킬 준비가 되었습니다.

TestAutoMileageLog.java 목록 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 목록 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;}
}
```

이 단계는 먼저 TestAutoMileageLog 에 테스트를 추가하고 FuelingStationVisit 에 메소드를 추가하여 작성되었습니다. 테스트될 준비가 되기 이전에 포함되는 3 개 또는 4 개의 컴파일이 있었습니다. 테스트가 처음으로 실행되었습니다.

이 극단적인 점진을 통해 무엇을 얻을 수 있는지 궁금할 수도 있습니다. FuelingStationVisit 를 작성한 다음 나중에 테스트 코드를 작성할 수 없었습니까? FuelingStationVisit 를 테스트할 필요가 조금이라도 있습니까? 지금까지, 처음으로 테스트를 작성하거나 테스트를 조금이라도 작성하는 것은 하나를 제외하고는 거의 장점을 제공하지 않습니다. 위의 코드가 컴파일되고 실행된다는 것을 명확하게 알고 있습니다. 따라서 다음 변경이 컴파일러 오류 또는 테스트 실패를 발생시키는 경우 문제점이 이전 코드에 있는 것이 아니라 변경에 있었던 것임을 알게 됩니다. 이것은 작은 장점인 것으로 보일 수 있지만 나중에 훨씬 중요하게 될 것입니다.

다음으로, FuelingStationVisit 객체를 어딘가에 놓아야 합니다. 일부 객체는 이 객체를 보유하고 있어야 합니다. 그것은 어떤 객체이어야 합니까? 그것은 이 정보를 보존하고 관리하려는 *사용자*입니다. 따라서 사용자 객체를 작성하여 FuelingStationVisit 객체를 보유하도록 할 수 있었습니다. 그러나 FuelingStationVisit 객체의 mileage 필드가 궁금합니다. 마일리지는 자동차의 속성입니다. FuelingStationVisit 객체는 방문한 순간에 기록되는 Vehicle 상태의 일부입니다. 따라서 Vehicle 객체를 작성하고 이 객체 내에 FuelingStationVisit 객체를 포함시켜야 합니다.

TestAutoMileageLog.java 목록 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 목록 5.2

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

```
}
```

목록 5 는 초기 단계를 표시합니다. testCreateVehicle 라는 새 테스트 함수를 작성했습니다. 이 함수는 Vehicle 을 작성한 후 Vehicle 내에 포함되는 방문 수가 0 인지 확인합니다. getNumberOfVisits 의 구현은 정확하게 잘못되었지만 테스트를 전달하는 장점이 있습니다. 이것을 통해 더 나은 솔루션으로 리팩토링할 수 있습니다.

Vehicle.java 목록 6

```
import java.util.Vector;

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

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

다시, 테스트를 전달합니다. testCreateVehicle 함수만이 아니라 모든 테스트를 실행하고 있음에 주의해야 합니다. 이것은 변경으로 인해 작업에 사용된 어느 것도 손상되지 않았음을 보증합니다.

다음으로, Vehicle 에 방문을 추가하는 방법을 해결해야 합니다. 가장 단순한 테스트 케이스는 무엇이라고 생각하십니까?

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

이 테스트에서 FuelingStationVisit 객체를 작성하지 않았음에 주의하십시오. Vehicle 의 addFuelingStationVisit 메소드가 FuelingStationVisit 객체를 작성한 후 이 객체를 목록에 추가해야 하는 것처럼 보입니다.

Vehicle.java 목록 8

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

다시, 모든 테스트를 전달합니다.

두 가지 함수, testAddVisit 및 testCreateFuelingStationVisit 의 중복된 코드로 인해 조금 불안해집니다. 두 가지 함수 모두는 동일한 로컬 변수를 작성하고 이런 변수를 동일한 값으로 초기화합니다. 이 중복을 제거하고 싶어할 것입니다. 따라서 로컬 변수를 구성원 변수로 작성하여 테스트 프로그램을 리팩토링합니다.

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

이 특별한 리팩토링은 이름을 가집니다. 임시를 필드로 승격이라고 합니다. 참조서 [1] 및 www.refactoring.com 에서 유사한 리팩토링 및 프로시저 목록을 찾을 수 있습니다.

단위 테스트가 있으면 이 리팩토링이 아무 것도 손상시키지 않았는지 확인할 수 있음에 유의하십시오. 어플리케이션을 리팩토링하고 재구축할 때 이것을 계속 이용합니다. 불안하게 느껴지는 코드에 무엇인가를 수행할 때마다 모든 것이 잘 작동하는지 확인하기 위해 다시 테스트로 돌아올 수 있습니다.

FuelingStationVisit 객체를 Vehicle 에 추가하였으므로 이제 보고서를 생성하도록 Vehicle 에 요청할 수 있습니다. 가장 단순한 케이스에서 먼저 시작하여 테스트 케이스를 작성할 수 있습니다.

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

이 테스트 케이스를 작성하려면 보고서 생성과 관련된 문제를 깊이 생각해야 합니다. 먼저, Vehicle 이 generateMileageReport 라는 이름의 메소드를 가져야 한다고 결정했습니다. 그런 다음, 이 함수가 MileageReport 라는 이름의 객체를 리턴해야 한다고 결정했습니다. 마지막으로, MileageReport 가 몇몇 조회 메소드를 가져야 한다고 결정했습니다.

이런 조회 방법이 리턴하는 값은 다소 흥미있습니다. 단일 방문만으로는 주행 마일 수 또는 갤런당 마일 수(mpg)를 계산하기에 충분하지 않습니다. 이런 값을 계산하려면 적어도 두 번의 방문이 필요합니다. 그 반면, 연료 소비량 및 연료 비용 계산은 단일 방문으로도 충분합니다.

물론, 테스트 케이스는 컴파일되지 않습니다. 따라서 적절한 메소드 및 클래스를 추가해야 합니다. 먼저 테스트 케이스가 컴파일되기에 충분한 코드를 추가합니다. 그러나 테스트에 실패합니다.

Vehicle.java 목록 11.1

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

TestAutoMileageLog.java 목록 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;
}
```

목록 11 의 코드가 컴파일되고 실행되지만 테스트가 실패합니다. 테스트를 전달할 수 있도록 코드를 리팩토링해야 합니다. 우선, 가능한 단순한 방법부터 사용합니다.

Vehicle.java 목록 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 목록 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;}
```

Vehicle 이 하나의 방문만을 포함하고 있다고 가정합니다. (걱정하지 마십시오. 나중에 다른 조건에 맞는 다른 테스트 케이스를 추가할 것입니다.) MileageReport 의 필드를 적절하게 설정한 후 리턴합니다.

아무리 잘해도 구현이 불완전하다는 것을 확실히 알고 있으므로 이 방법으로 generateMileageReport 를 구현하는 것이 어리석게 보일 수 있습니다. 그러나 작은 증가치로 구현하는 것은 각 컴파일 및 테스트 간에 변경사항이 거의 없다는 장점이 있습니다. 무엇인가가 잘못되면 항상 최신 버전으로 가서 다시 시작할 수 있습니다. 디버그할 필요가 없습니다.

목록 12 의 코드는 컴파일되고 테스트를 전달하지만 분명히 불완전합니다. 이를 완료하려면 다른 일부 테스트 케이스에 대해 생각할 필요가 있습니다.

- 한 번도 방문하지 않은 자동차
- 한 번 이상 방문한 자동차

한 번도 방문하지 않은 케이스는 단순합니다. 목록 13.1 의 테스트 케이스가 실패하고 목록 13.2 의 코드가 다시 테스트를 전달합니다.

TestAutoMileageLog.java 목록 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 목록 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;
}
```

다음으로 많은 방문을 처리하는 테스트 케이스를 고려해야 합니다.

TestAutoMileageLog.java 목록 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);
    }
```

Vehicle 에 세 번의 방문을 끼워 넣기로 선택했습니다. 비용은 갤런당 약 1.20 달러, 마일리지는 갤런당 약 30 마일(30 mpg)을 기초로 했습니다. 따라서 12.24 달러의 비용으로 292 마일을 여행하려면 9.8 갤런을 사용합니다.

여기에 나머지 문제점이 있습니다. 읽고 있는 주행 기록계는 약 30 mpg 를 기반으로 했습니다. 그러나 541 을 주행 거리 23.1 로 나누면 소비된 갤런 23.41991 mpg 를 얻게 됩니다. 왜 mpg 가 서로 다릅니까? 왜 30 mpg 에 가까운 값을 얻지 못합니까?

숙고한 결과, 연료 소비량이 매 방문 시에 구매한 모든 연료의 합이 아니라는 것이 명확해졌습니다. 연료는 방문 *사이에* 소비됩니다. mpg 를 계산할 때 첫 번째 방문에서 구매한 연료가 고려되어서는 안됩니다.

TestAutoMileageLog.java 목록 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);
}
```

이것이 더 나아 보입니다. 테스트를 작성할 때 발견한 것을 결코 알지 못합니다. 한 가지 확실한 것은 두 번 지정할 때, 다시 말해 방금 코드를 작성한 경우보다 테스트 및 코드에서 더 많은 오류가 발견되기 쉽다는 것입니다.

이제, 이전 테스트를 전달하는 코드 추가를 시도할 준비가 되었습니다.

Vehicle.java 목록 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;
}
```

이 코드는 모든 특수 케이스에 대해 좋은 상태가 아닙니다. 특수 케이스를 리팩토링해야 합니다. 사실, 세 번째 케이스는 현재 상태로도 충분히 일반적입니다. 다른 두 케이스를 없앨 수 있어야 합니다.

이것을 수행할 때 testSingleisitMileageReport 테스트 케이스가 실패합니다. 실패의 원인은 단일 방문 케이스가 처음으로 구매한 연료 및 방문만을 포함하고 있기 때문입니다. 위에서 발견한 대로, 한 번만 방문한 경우 연료 소비량이 0 이어야 합니다. 따라서 테스트 케이스 및 코드를 수정할 수 있습니다.

Vehicle.java 목록 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;
}
```

이 함수의 길어서 줄이고 다소 정리해야 합니다. 코드가 개별 함수로 이동될 수 있도록 코드를 조금씩 이동하여 정리를 시작합니다.

Vehicle.java 목록 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;
}
```

목록 18 은 중간 단계입니다. 실제로 이 지점에 도달하는 데 4 개에서 5 개의 보다 더 작은 단계가 걸렸습니다. 각각의 더 작은 단계에서, 손상된 것은 없는지 확인하기 위해 테스트를 실행할 수 있었습니다. 이런 리팩토링의 목표는 어떻게든 분리되기 더 쉬운 코드를 얻는 것이지만 이것을 수행하는 방법에 대한 견고한 개념을 가지지 않았습니다. 따라서 이런 첫 번째 리팩토링은 거의 무작위입니다. 이 리팩토링에는 많은 시간이 소요되지 않았고 테스트가 손상된 것은 없는지 확인했습니다.

여전히 실행 중인 테스트를 통해 이 지점에 도달했다면 개선하는 방법을 볼 수 있습니다. 루프를 2 개로 분리[2] 하여 시작합니다.

Vehicle.java 목록 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;
        }
```

테스트는 계속해서 실행됩니다. 다음으로, 각 루프를 자체 개인용 메소드로 추출합니다.[3]

Vehicle.java 목록 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 =
```

[2]www.refactoring.com 에서 루프 분리를 참조하십시오.

[3]www.refactoring.com 에서 추출 메소드를 참조하십시오.

```
                (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;
    }
```

테스트는 계속해서 실행됩니다. 그런 다음, 연료 소비량에 대한 특수 케이스를 `calculateFuelConsumption` 메소드로 이동합니다.

Vehicle.java 목록 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;
}
```

테스트는 계속해서 실행됩니다. 이제 *두 번째* 방문을 통해 calculateFuelConsumption 이 연료 소비량을 합산하기 위한 시작 수단을 택할 수 있습니다. 그런 다음, 거리를 계산하기 위해 함수를 추출할 수 있습니다.

Vehicle.java 목록 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;
}
```

테스트는 계속해서 실행됩니다. 이제 기본 함수에서 조건을 제거하고 약간의 조각들을 정리할 수 있습니다.

Vehicle.java 목록 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;
    }
```

테스트는 계속해서 실행됩니다.

이것은 아주 좋습니다. 각 함수는 완전 독립되어 다른 함수로부터 잘 분리됩니다. 기본 함수는 작고 이해하기 쉽습니다.

함수가 프로그램을 더 복잡하게 한다고 주장할 수도 있습니다. 함수 총계 및 라인 총계를 확실히 늘리는 동안 이 목록은 프로그램을 훌륭하게 분할하기도 했습니다. 각 함수는 이해하기 쉽습니다.

목록 16 의 케이스 분석이 리턴되었지만 이제 특정 계산 함수와 연관됨에 주의하십시오. 이것은 케이스 분석의 제거가 우연히 작동되었던 목록 17 보다 훨씬 더 좋습니다.

일부는 이 코드가 쓸데없이 느리다고 불평할 수도 있습니다. 이것이 사실일 수 있지만 속도가 빠를 필요는 없습니다. 속도가 요구사항이 되고 현재 실행이 이 요구사항에 적합하지 않으면, 이에 대해 무엇인가를 수행할 수 있습니다. 그 때까지, 목록 23 에 표시된 관심사의 명확성과 분리에 대해 만족할 것입니다.

결론

이 문서가 테스트 우선 설계에서 리팩토링을 수행하는 기술을 시연했지만 실제 목적은 프로그래밍 *자세*를 전달하는 것입니다. 프로그램은 작동시 완료되지 않습니다. 사실, 프로그램이 작동되게 하는 것은 쉬운 부분입니다. 프로그램은 프로그램이 작동할 때 *및* 프로그램이 가능한 단순하고 순수할 때 프로그램이 수행됩니다.

이 문서에서 바람직한 결과를 달성하기 위한 좋은 방법은 다음과 같습니다.

1. 테스트 케이스를 작성하여 프로그램을 설계합니다. 각 테스트 케이스가 작성된 후에 각 테스트 코드를 전달시키는 코드를 작성하십시오. 모든 테스트를 모아서 반복적으로 실행하는 것을 용이하게 하십시오.
2. 프로그램의 한 파트가 작동하면 이 파트에 결점이 없어질 때까지 리팩토링하십시오. 일련의 사소한 변경을 코드에 작성하고 변경한 후에 테스트를 실행하여 리팩토링을 수행하십시오. 이것은 변경사항으로 손상된 것이 없는지에 대한 확신 및 제대로 수행할 수 있을 만큼 코드가 명확해지고 완전해질 때까지 계속해서 변경에 변경을 거듭할 수 있는 용기를 제공합니다.

참조서

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

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

Rational
the software development company

본사:

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

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

무료 전화번호: (800) 728-1212
전자 우편: info@rational.com
웹: www.rational.com
월드와이드: www.rational.com/worldwide