RUP/XP 가이드라인: 테스트 우선 디자인 및 리팩토링

Robert C. Martin
Object Mentor, Inc.

Rational Software 백서

TP 159, 03/01

Rational®
the software development company

목차

개요

소프트웨어 영역에서 완전히 새로운 사례가 발생하는 경우는 드물지만 구조화 프로그래밍과 OO 가 그러한 사례에 속합니다. 테스트 우선 디자인 및 리팩토링 또한 이러한 사례에 속합니다.

리팩토링에 대한 정확한 정의는 프로그램의 기능은 보존하면서 해당 구조를 변경하는 사소한 변경 작업을 의미합니다. 이 정의를 통해 소프트웨어에 뚜렷한 두 가지 가치가 있음을 알 수 있습니다. 첫 번째는 소프트웨어 기능에 대한 가치이며 두 번째 가치는 소프트웨어 구조에 있습니다. 앞의 정의에 따르면 리팩토링은 소프트웨어의 구조적 가치를 유지보수하고 향상시키는 기법입니다.

리팩토링에 대한 보다 세부적인 정의는 기능 추가와 구조 개선에 초점을 두는 사소한 여러 변경사항을 통해 소프트웨어를 디자인하고 구현하는 기법입니다. 이 정의는 Fowler 가 그의 저서 *Refactoring*(참조 [1] 참고)에서 설명한 단어의 의미를 확장하는 것이며 XP(*eXtreme Programming*)(참조 [2] 참고) 프로세스에서 소프트웨어를 디자인하고 작성하는 방법에 대해 설명합니다.

테스트 우선 디자인과 리팩토링은 코드를 작성하기 전에 테스트 케이스를 작성하는 방식으로 코드를 디자인하고 개선하는 사례입니다. 프로그래머는 타스크를 선택하고 하나 또는 두 개의 간단한 유닛 테스트 케이스를 작성한 다음 테스트에 패스할 수 있도록 프로그램을 수정합니다. 테스트 케이스는 프로그램이 이 타스크를 수행하지 않으므로 실패합니다. 프로그래머는 소프트웨어가 필요한 기능을 모두 수행하여 테스트 케이스가 패스될 때까지 계속 추가합니다. 프로그래머는 한 번에 소규모 단계를 하나씩 수행하여 시스템 구조를 개선합니다. 또한 누락되는 부분이 없도록 각 단계 사이에 모든 테스트를 실행합니다.

리팩토링 예제

테스트 우선 디자인 및 리팩토링을 가장 효과적으로 설명할 수 있는 방법은 예제를 사용하는 것입니다. 따라서 이 문서에서는 소규모 프로그램을 디자인 및 구현하여 리팩토링 수행 방법을 설명합니다. XP 에서는 동일한 워크스테이션을 사용하는 두 명의 프로그래머가 이 활동을 수행합니다.[1]을 참조하십시오.

여기서 빌드할 응용프로그램은 간단한 자동 주행거리 기록 응용프로그램입니다. 사용자는 주유소를 방문할 때마다 주입한 연료량, 해당 비용 및 현재 자동차 주행 기록계의 수치를 입력합니다. 시스템은 이 항목을 추적하여 유용한 보고서를 생성합니다. 구현 언어는 Java 입니다.

먼저 목록 1 의 코드를 작성합니다.

TestAutoMileageLog.java 목록 1

```
import junit.framework.*;

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

첫 번째 작성 내용은 유닛 테스트를 포함하는 프레임워크입니다. 순서가 잘못된 것처럼 보일 수도 있지만 테스트 우선 개념에 필요한 기초 작업입니다. 실제 응용프로그램 코드를 작성하기 전에 테스트 코드를 먼저 작성합니다. 작업 진행에 따라 해당 기능을 확인할 수 있습니다.

여기서 사용하는 테스트 프레임워크는 Kent Beck 및 Erich Gamma 가 작성한 간단한 유닛 테스트 프레임워크인 JUnit 입니다. 위의 코드만 작성하면 이 프레임워크가 설정됩니다.

[1] Rational Software 백서, *RUP /XP 가이드라인: 짝 프로그래밍*

이제 첫 번째 테스트 케이스에 대해 알아보겠습니다. 이 소프트웨어에 필요한 기능은 먼저 주유소 방문 내용을 기록하는 것입니다. 즉, 관련 데이터를 보관하는 FuelingStationVisit 오브젝트가 필요합니다. 따라서 이 오브젝트를 작성한 후 해당 필드를 조회하는 테스트를 작성할 수 있습니다.

이 작업을 위해서는 먼저 테스트 함수를 작성합니다. JUnit 에서 테스트 함수는 이름이 "test"로 시작하는 TestCase 에서 파생된 클래스의 메소드입니다. 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 표시됩니다. 위의 작업은 모두 새 오브젝트인 FuelingStationVisit 를 작성하기 위한 것입니다. 구현/구축(Construction) 인수는 아직 제공하지 않았습니다. 이 시점에서는 오브젝트를 작성하는 것이 가장 중요합니다.

또한 이 작업은 컴파일 작업이 아닙니다. 그러나 컴파일 작업도 가능합니다. 컴파일 작업을 수행하려면 FuelingStationVisit 오브젝트에 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 코드를 컴파일하고 테스트가 실행되면 원하는 기능을 추가할 수 있습니다.

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 추가한 다음 FuelingStationVisit 에 메소드를 추가하여 작성됩니다. 테스트 준비가 완료되기 전에 세 가지 또는 네 가지 컴파일이 관련된 상태이며 테스트가 처음 실행되었습니다.

이렇게 점진적으로 내용을 추가하는 이유에 대해 알아볼 필요가 있습니다. 간단하게 FuelingStationVisit 을 먼저 작성한 다음 테스트 코드를 작성하면 된다고 생각할 수 있습니다. 또한 FuelingStationVisit 을 테스트해야 하는지에 대해서도 생각해볼 수 있습니다. 지금까지 테스트를 먼저 작성하거나 조금이라도 작성하는 경우에도 한 가지를 제외하고는 얻는 이점이 거의 없었습니다. 위의 코드는 명확히 컴파일되고 실행됩니다. 따라서 다음 변경으로 인해 컴파일 오류가 발생하거나 테스트에 실패하는 경우, 문제점은 이전 코드가 아닌 해당 변경 작업에서 비롯된 것입니다. 이는 그다지 중요하지 않은 이점으로 보일 수 있지만 후반부에서 그 중요성이 더 커집니다.

다음으로 FuelingStationVisit 오브젝트를 배치해야 합니다. 이 오브젝트는 특정 오브젝트에 포함되어야 하며 해당 오브젝트를 확인해야 합니다. 이 정보를 보관하고 관리하는 사람은 *사용자*이므로 FuelingStationVisit 오브젝트를 보관할 User 오브젝트를 작성할 수 있습니다. 그러나 FuelingStationVisit 오브젝트의 주행거리 필드가 문제입니다. 주행거리는 자동차의 한 속성입니다. FuelingStationVisit 오브젝트는 방문 시 Vehicle 의 일부 상태를 기록합니다. 따라서 Vehicle 오브젝트를 작성하고 그 안에 FuelingStationVisit 오브젝트를 포함해야 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 보여줍니다. 이미 testCreateVehicle 이라는 새 테스트 함수를 작성했습니다. 이 함수는 Vehicle 을 작성한 다음 포함된 방문 횟수가 0 인지 확인합니다. getNumberOfVisits 구현은 잘못된 것이지만 테스트가 패스되는 이점을 갖고 있습니다. 이를 통해 보다 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 테스트가 패스됩니다. 현재 testCreateVehicle 함수뿐만 아니라 모든 테스트를 실행하고 있음에 유의해야 합니다. 이를 통해 변경으로 인해 문제가 발생하지 않음을 알 수 있습니다.

다음으로 Vehicle 에 방문 기록을 추가하는 방법을 결정해야 합니다. 가장 간단한 테스트 케이스 구성을 결정해야 합니다.

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 테스트에서는 FuelingStationVisit 오브젝트를 작성하지 않았습니다. Vehicle 의 addFuelingStationVisit 메소드는 FuelingStationVisit 오브젝트를 작성한 다음 목록에 추가해야 합니다.

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 테스트가 패스됩니다.

testAddVisit 함수와 testCreateFuelingStationVisit 함수의 중복 코드에 유의해야 합니다. 두 함수는 동일한 로컬 변수를 작성하며 동일한 값으로 초기화합니다. 이 중복 코드를 제거해야 합니다. 따라서 구성원 변수에 로컬 변수를 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 특정 리팩토링에는 임시에서 필드로 승격이라는 이름이 지정됩니다. 유사한 리팩토링 및 프로시저를 적용하기 위한 목록은 참조 [1]과 www.refactoring.com 을 참조하십시오.

유닛 테스트를 통해 이 리팩토링에 문제가 없음을 확인했습니다. 응용프로그램을 리팩토링하고 재구성하면서 이 유닛 테스트를 계속 이용합니다. 불필요한 코드를 수정할 때마다 다시 테스트 작업을 수행하여 코드에 문제가 없는지 확인할 수 있습니다.

Vehicle 에 FuelingStationVisit 오브젝트를 추가했으므로 이제 Vehicle 에서 보고서를 생성할 수 있습니다. 먼저 가장 간단한 케이스부터 테스트 케이스를 작성합니다.

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 고려해야 합니다. 먼저 Vehicle 에 generateMileageReport 메소드가 필요한 것으로 결정했습니다. 다음으로 이 함수가 MileageReport 오브젝트를 리턴해야 하는 것으로 결정했습니다. 마지막으로 MileageReport 에 여러 조회 메소드가 필요한 것으로 결정했습니다.

이 조회 메소드가 리턴하는 값에 유의해야 합니다. 1 회 방문만으로는 총 주행 거리 또는 갤런당 주행 거리를 계산할 수 없습니다. 이 값을 계산하려면 최소한 두 번 이상 방문해야 합니다. 반면에 연료 소비량 및 해당 비용을 계산하는 데는 1 회 방문만으로도 충분합니다.

물론 테스트 케이스는 컴파일되지 않습니다. 따라서 해당 메소드 및 클래스를 추가해야 합니다. 먼저 컴파일할 수 있는 코드를 추가하지만 테스트에는 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 목록 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 코드는 컴파일 및 실행되지만 테스트에 실패합니다. 이제 테스트에 패스할 수 있도록 코드를 리팩토링해야 합니다. 처음에는 가능한 간단한 접근 방식을 취합니다.

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 에 1 회 방문만 포함되는 것으로 가정합니다 (다른 조건에 대한 다른 테스트 케이스는 나중에 추가합니다). 필요한 MileageReport 필드를 설정한 후 리턴합니다.

generateMileageReport 구현은 불완전하므로 이러한 방법으로 구현하는 것이 불필요한 작업으로 보일 수도 있습니다. 그러나 약간의 추가 작업을 통한 구현 시 각 컴파일과 테스트 간의 변경을 최소화할 수 있다는 이점이 있습니다. 문제가 발생하는 경우, 항상 최종 버전에서 다시 시작할 수 있으며 디버그 작업이 필요하지 않습니다.

목록 12 의 코드는 컴파일되고 테스트를 패스하지만 불완전한 코드입니다. 이 코드를 완성하려면 몇 가지 다른 테스트 케이스를 고려해야 합니다.

- 방문 기록이 없는 자동차
- 2 회 이상의 방문 기록이 있는 자동차

방문 기록이 없는 케이스는 간단합니다. 목록 13.1 의 테스트 케이스를 실패하고 목록 13.2 의 코드에서는 다시 패스됩니다.

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 방문 기록을 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 3 회 방문 기록을 적용하기로 결정했습니다. 기준 연료 비용은 갤런당 약 $1.20 이며 갤런당 기준 주행 거리는 약 30 마일(mpg)입니다. 따라서 292 마일을 이동하려면 9.8 갤런의 연료가 필요하며 그에 따른 비용은 $12.24 입니다.

여기에는 이상한 문제점이 있습니다. 각 주행 기록계의 기준 눈금을 약 30mpg 로 설정했지만 주행 거리 541 을 소비한 연료(갤런) 23.1 로 나눈 결과 값은 23.41991mpg 입니다. 이러한 차이의 원인과 30mpg 와 근사한 값이 나오지 않는 이유에 대해 의문을 가질 수 있습니다.

생각해보면 실제 연료 소비량과 매번 방문할 때마다 주입한 모든 연료의 합계가 일치하지 않음을 알 수 있습니다. 연료는 방문 *시이*에 소비되므로 첫 번째 방문 시 주입한 연료는 mpg 를 계산할 때 고려하지 않아야 합니다.

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);
}
```

이 구성이 보다 올바른 구성입니다. 테스트 작성 시에는 해당 결과를 알 수 없습니다. 코드만 작성하는 경우보다 테스트 및 코드에 두 번 지정함으로써 보다 많은 오류를 발견할 수 있습니다.

이제 이전 테스트를 패스할 수 있도록 코드를 추가할 수 있습니다.

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 있어 적합하지 않으므로 특수 케이스에 맞게 리팩토링해야 합니다. 실제로는 세 번째 케이스가 가장 일반적이며 나머지 두 케이스를 제거할 수 있어야 합니다.

이 작업을 수행하면 testSingleVisitMileageReport 테스트 케이스가 실패합니다. 실패 이유는 단일 방문 케이스에 첫 번째 방문 시 주입한 연료가 포함되었기 때문입니다. 위에서 확인한 대로 방문 횟수가 1 회뿐인 경우 연료 소비량은 0 이어야 합니다. 따라서 테스트 케이스와 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;
}
```

이렇게 긴 함수는 줄여서 정리해야 합니다. 먼저 약간의 코드를 별도 함수로 이동시키는 작업부터 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 도달하려면 소규모의 네 단계 또는 다섯 단계를 더 수행해야 합니다. 이러한 각 단계에서 테스트를 실행하여 문제가 없는지 확인할 수 있었습니다. 이러한 리팩토링의 목적은 코드를 쉽게 분할하는 것이지만 이 방법에 대해서는 명확히 언급하지 않았습니다. 따라서 이러한 첫 번째 리팩토링은 거의 임의로 수행되었습니다. 시간도 많이 소요되지 않으며 테스트를 통해 문제가 없는지 확인했습니다.

이 시점에서는 테스트가 계속 실행되므로 개선 방법을 확인할 수 있습니다. 먼저 루프[2] 를 두 개로 분할합니다.

Vehicle.java 목록 19

```
        if (itsVisits.size() > 0)
        {
            FuelingStationVisit firstVisit =
                (FuelingStationVisit)itsVisits.get(0);
            FuelingStationVisit lastVisit =
                (FuelingStationVisit)itsVisits.get(itsVisits.size()-1);
```

[2] www.refactoring.com 의 루프 분할을 참조하십시오.

```
            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 계속 실행됩니다. 다음으로 각 루프를 해당 private 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 =
                (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;
```

[3] www.refactoring.com 에서 메소드 추출을 참조하십시오.

```
            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 계속 실행됩니다. 다음으로 연료 소비 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 계속 실행됩니다. 이제 calculateFuelConsumption 은 *두 번째* 방문 기록으로 연료 소비량을 합산할 수 있습니다. 다음으로 거리 계산 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 계속 실행됩니다. 이제 기본 함수에서 조건식을 제거하고 몇 가지 불필요한 요소를 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 계속 실행됩니다.

어느정도 올바른 코드 작성입니다. 각 함수는 필요한 요소를 모두 포함하고 있으며 다른 함수와 분리되어 있습니다. 기본 함수는 길지 않아 쉽게 이해할 수 있습니다.

이 코드가 프로그램을 보다 복잡하게 만든다고 주장할 수도 있습니다. 이 코드로 인해 함수 및 행 수가 늘어나는 것은 사실이지만 프로그램을 적절히 분할했습니다. 각 함수도 쉽게 이해할 수 있습니다.

목록 16 의 케이스 분석이 리턴되었지만 이제 특정 계산 함수와 연관되어 있습니다. 이 목록은 케이스 분석을 우연히 제거한 목록 17 보다 올바른 구성입니다.

이 코드의 속도가 불필요하게 느리다고 불평할 수도 있습니다. 이는 사실일 수도 있지만 속도가 중요하지는 않습니다. 속도를 고려해야 하는 경우 또한 현재 실행으로 속도가 느려지는 경우 필요한 조치를 수행할 수 있습니다. 이러한 경우를 제외하고는 목록 23 에 표시된 관심사항의 명확성과 분리 기능을 활용할 수 있습니다.

결론

이 문서에서는 테스트 우선 디자인에서의 리팩토링 기법에 대해 설명했지만 실제 목적은 프로그래밍 *태도*에 대해 설명하기 위한 것입니다. 프로그램이 실행된다고 완성된 것은 아닙니다. 실제로 프로그램을 실행시키는 것은 쉬운 일입니다. 프로그램이 실행되고 최대한 간단하고 명확한 형태를 갖추어야 완료됩니다.

이 문서에서는 이러한 바람직한 결과를 얻기 위한 방법으로 다음 방법을 추천합니다.

1. 테스트 케이스를 작성하여 프로그램을 디자인합니다. 각 테스트 케이스가 작성된 후 해당 테스트 케이스를 패스하는 코드를 작성합니다. 모든 테스트를 모아 반복적으로 쉽게 실행할 수 있도록 합니다.
2. 프로그램 일부가 실행되면 해당 파트가 명확해질 때까지 리팩토링합니다. 코드에 대한 일련의 사소한 변경 작업을 수행하고 변경할 때마다 테스트를 실행하여 리팩토링을 수행합니다. 이러한 방법을 통해 변경 작업에 따른 문제가 발생하지 않도록 할 수 있으며 코드가 가능한 명확해질 때까지 변경 작업을 계속 수행할 수 있습니다.

참조

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

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

Rational
the software development company

본사 안내:

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

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

수신자 부담 전화번호: (800) 728-1212
전자 우편: info@rational.com
웹: www.rational.com
전세계 지사 안내: www.rational.com/worldwide