

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут

ім. І.І.Сікорського»

Навчально-науковий комплекс

«Інститут прикладного системного аналізу»

**Лабораторна робота №4**

з курсу "Проектування інформаційних систем"

на тему

«Модульне тестування та рефакторинг»

Виконав:

студент ІV курсу

групи ДА-41

Клещ Кирило

Варіант-9

Київ 2017

**Мета роботи:**

Оволодіти навичками створення програмного забезпечення за метолологією

TDD та ознайомитися з процедурами рефакторинга.

**Задача:**

1. Використовувати методологію Test Driven Development для створення класів архітектурної

програмної моделі.

2. Скласти тестові сценарії, які продемонструють функціювання всіх методів проектованої

моделі.

3. Виконати юніт-тестування складових частин (внутрішніх класів), що реалізують об'єкт

моделювання.

4. Виконати "зовнішнє" юніт-тестування для API.

5. Провести рефакторинг коду програми, для поліпшення реалізації.

**Короткі теоретичні відомості:**

Розробка через тестування (англ. тест-Driven Development, TDD) - техніка розробки

програмного забезпечення, яка грунтується на повторенні дуже коротких циклів розробки:

спочатку пишеться тест, що покриває бажану зміну, потім пишеться код, який дозволить пройти тест, і під кінець проводиться рефакторинг нового коду до відповідних стандартів.

Рефакторинг (англ. refactoring) - процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи. В основі рефакторинга лежить послідовність невеликих еквівалентних (тобто таких, що зберігають поведінку) перетворень. Оскільки кожне перетворення маленьке, програмісту легше простежити за його правильністю, і в той же час вся послідовність може привести до істотної перебудови програми і поліпшенню її узгодженості і чіткості.

Модульне тестування, або юніт-тестування (англ. unit testing) - процес у програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми. Ідея полягає в тому, щоб писати тести для кожної нетривіальної функції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергову зміну коду до регресії, тобто до появи помилок у

вже відтестованих місцях програми, а також полегшує виявлення та усунення таких помилок.

**Завдання**

1. Розробити методику випробувань з використанням ISO/IEC/IEEE 29119.

2. Розробити код програми архітектурної моделі. Використовувати Test Driven Development.

3. Провести рефакторинг коду програми, щоб задовольнити вимоги технічного завдання.

**Хід роботи:**

Код моделі:

/\*\*

\* The class serves to calculate the cost of petrol.

\* This value is written in <b>totalCost</b>.

\* **@author** Kirill Kleshch

\* **@version** 1.0

\* Created by Kirill on 13.11.2017.

\*/

**public** **class** Model {

**private** **static** **final** **double** ***fuel\_consumption\_per\_kilometer*** = 0.07;

**private** **static** **final** **double** ***price\_of\_1\_litter*** = 23.78;

/\*\* The amount of gasoline spent \*/

**private** **double** petrol;

/\*\*Distance that the car passed \*/

**private** **double** distance;

/\*\* The price of petrol that was spent \*/

**private** **double** totalCost;

/\*\* Function to get the value of a field {@link Model#totalCost}

\* **@return** return total price

\*/

**public** **double** getTotalCost() {

**return** totalCost;

}

/\*\* Function to get the value of a field {@link Model#petrol}

\* **@return** return amount of petrol spent

\*/

**public** **double** getPetrol() {

**return** petrol;

}

**public** **void** setPetrol(**double** petrol) {

**this**.petrol = petrol;

}

/\*\* Function to get the value of a field {@link Model#distance}

\* **@return** return distance that the car passed

\*/

**public** **double** getDistance() {

**return** distance;

}

**public** **void** setDistance(**double** distance) {

**this**.distance = distance;

}

/\*\* Function of calculating value

\* **@param** distance distance that the car passed

\*/

**public** **void** calculated\_price(**double** distance){

totalCost = distance\****fuel\_consumption\_per\_kilometer***\****price\_of\_1\_litter***;

}

}

Проведемо тестування найбільш складних методів, де можна було зробити помилку:

**import** **static** org.junit.Assert.\*;

/\*\*

\* Created by Kirill on 16.11.2017.

\*/

**public** **class** ModelTest {

@org.junit.Test

**public** **void** getDistanceTest() **throws** Exception {

Model model = **new** Model();

model.setDistance(17.7);

**double** calc\_value = model.getDistance();

**double** exp\_value = 17.7;

*assertEquals*(exp\_value, calc\_value, 0.000001);

}

@org.junit.Test

**public** **void** calculated\_price() **throws** Exception {

Model model = **new** Model();

model.setDistance(35.8);

model.calculated\_price(model.getDistance());

**double** calc\_value = model.getTotalCost();

**double** exp\_value = 59.59268;

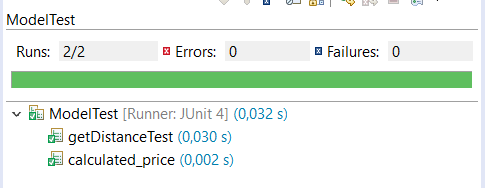
*assertEquals*(exp\_value, calc\_value, 0.00001);

}

}

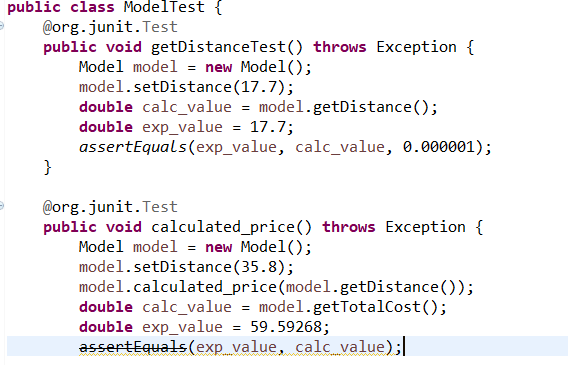
Звернемо увагу,як потрібно перевіряти чи рівні між собою два дабли, за допомогою методу *assertEquals* з заданою точністю.

Перевіримо чи дійсно 2 тести проходять:

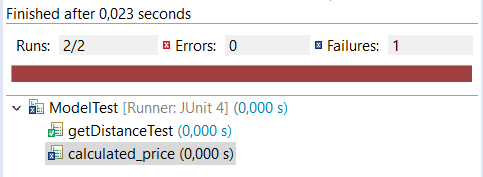


Так, маємо зелену доріжку.

Якщо невірно порівняти 2 дабли, то будемо мати:



То тести, не будуть проходити:



**Рефакторинг:**

/\*\*

\* Created by Kirill on 13.11.2017.

\*/

**import** java.util.Scanner;

**public** **class** Controller {

// Constructor

Model model;

View view;

String stringOfUser = **new** String();

**public** Controller(Model model, View view) {

**this**.model = model;

**this**.view = view;

}

// The Work method

**public** **void** processUser() {

Scanner sc = **new** Scanner(System.***in***);

view.printMessage(View.*ENTER\_DISTANCE*);

**double** bufMessage;

bufMessage = sc.nextDouble();

model.setDistance(bufMessage);

model.calculated\_price(model.getDistance());

view.printMessage(View.*OUTPUT\_WORD*, model.getTotalCost());

}

}

В такому коді моло, що зрозуміло.Особливо все не зручно в методі processUser().

Тому використаємо метод рефакторингу “Извлечение метода”.

Винесемо шматок коду у окремий метод.

/\*\*

\* Created by Kirill on 13.11.2017.

\*/

**import** java.util.Scanner;

**public** **class** Controller {

// Constructor

Model model;

View view;

String stringOfUser = **new** String();

**public** Controller(Model model, View view) {

**this**.model = model;

**this**.view = view;

}

// The Work method

**public** **void** processUser() {

Scanner sc = **new** Scanner(System.***in***);

model.setDistance(inputDIstanceWithScanner(sc));

model.calculated\_price(model.getDistance());

view.printMessage(View.*OUTPUT\_WORD*, model.getTotalCost());

}

**public** **double** inputDIstanceWithScanner(Scanner sc) {

view.printMessage(View.*ENTER\_DISTANCE*);

**double** bufMessage;

bufMessage = sc.nextDouble();

**return** bufMessage;

}

}

**Висновки:**

В ході лабораторної роботи, були розроблені модульні тести для моделі.За допомогою Junit-4, та проведений рефакторнг классу контроллер за методом рефакторингу “Извлечение метода”.