

Міністерство освіти і науки України
Національний Технічний Університет України
«Київський Політехнічний Інститут»
Навчально-науковий комплекс «Інститут прикладного системного аналізу»
Кафедра системного проектування

Лабораторна робота №5
з дисципліни
“Проектування інформаційних систем”
Модульне тестування (Unit-тести) та рефакторинг

Виконала:
студентка групи ДА-72
Мелкозерова Ольга

Київ – 2020

Мета роботи: оволодіти навичками створення програмного забезпечення за методологією 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 функціонування 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.

Хід роботи

Для написання тестів буде використано мову Java, а саме інструмент JUnit. Він є гнучким і досить простим фреймворком для тестування.

Для того, щоб підключити JUnit, нам необхідно додати dependency у pom.xml

```
44     <dependencies>
45         <dependency>
46             <groupId>junit</groupId>
47             <artifactId>junit</artifactId>
48             <version>4.13.1</version>
49             <scope>test</scope>
50         </dependency>
51     </dependencies>
52
```

Далі можемо переходити до створення програми. Досить примітивним, але зрозумілим шляхом є написання тестів для арифметичних функцій. Наприклад, додавання та віднімання. Тут ми завжди знаємо результат і можемо з легкістю визначити коректність роботи класу.

Тож створимо клас CalculatorUTest для наших тестів. Його наповнення буде таким:

```
public class CalculatorUTest {
    int a;
    int b;
    @Before
```

```

public void setUp(){
    a = 4;
    b = 6;
}
@Test
public void testAdd(){
    int res = 10;
    Assert.assertEquals("6 + 4 мають дати 10 в сумі",
        res, Calculator.add(a,b));
}
@Test
public void testSub(){
    int result = -2;
    Assert.assertEquals("4-6 має бути -2», result,
Calculator.sub(a,b));
}}

```

@Before вказує на те, що метод буде виконуватися перед кожним тестованим методом

@Test визначає метод як тестовий

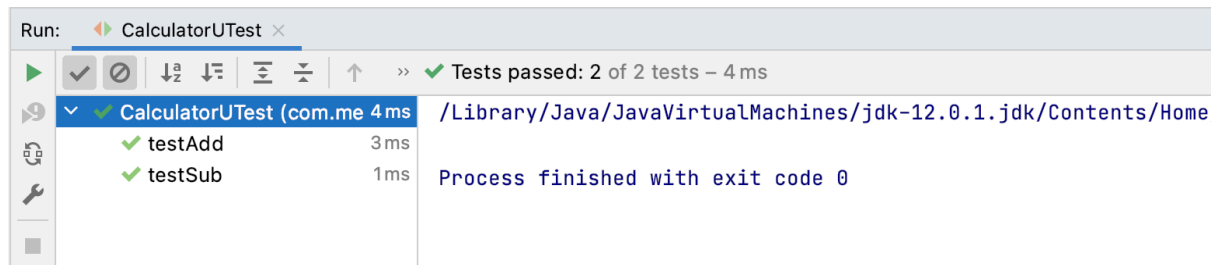
А потім створимо клас Calculator

```

public class Calculator {
    public static int add(int a, int b){
        return a + b; }
    public static int sub(int a, int b){
        return a - b;
    }}

```

Запустимо наше тестування, маємо результат:



Далі перейдемо до вивчення рефакторингу

Наприклад, створимо функцію порівняння двох чисел a та b:

```
public boolean max(int a, int b) {  
    if(a > b) {  
        return true;  
    } else if(a == b) {  
        return false;  
    } else {  
        return false;  
    }  
}
```

Тут ми бачимо, що функція виконує свої дії, проте як для такого простого запиту має загроміздкий код. В такому разі ми можемо просто переписати її як:

```
public boolean max(int a, int b) {  
    return a>b;  
}
```

В такому разі вона буде виконувати ту саму дію, але код буде в рази менший і зрозуміліший. Саме в цьому і є суть рефакторингу.

Висновки

Під час виконання даної лабораторної роботи, а також вивчення теоретичної частини для її розуміння, було досліджено unit тестування та рефакторинг коду на мові програмування Java. Також вивчено різницю між стандартним підходом та Test Driven Development, коли тест пишеться до написання коду методу.

Також було вивчено основні проблеми в коді, які потребують рефакторингу, а також самі методи рефакторингу.