МІНІСТЕРСТВО ОСВІТИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря Сікорського»
НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС
«ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ»
Кафедра Системного проектування

Лабораторна робота №5

з дисципліни: проектування інформаційних систем

на тему «Модульне тестування та рефакторинг»

Виконала:
Студентка 4 курсу
групи ДА-71
Опрісник Роксолана

Київ - 2020

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.

Хід роботи

Клас CustomList реалізує інтерфейс List, він містить реалізації для всіх методів, що ініціалізовані у цьому інтерфейсі.

Спочатку ми просто задаємо порожні тіла для цих методів. Якщо метод має тип повернення, ми повертаємо довільне значення цього типу, наприклад null для Object або false для boolean.

```
public class CustomList<E> implements List<E> {
    private Object[] internal = {};
}
```

Метод isEmpty - найпростіший метод, визначений в інтерфейсі List. Ось наша початкова реалізація:

```
@Override
public boolean isEmpty() {
    return false;
}
```

Цього початкового визначення методу достатньо для компіляції. Основна частина цього методу буде «змушена» вдосконалюватися, коли додаватиметься все більше і більше тестів.

Напишемо перший тестовий приклад, який гарантує, що метод isEmpty повертає true, коли список не містить жодного елемента:

```
@Test
public void givenEmptyList_whenIsEmpty_thenTrueIsReturned() {
    List<Object> list = new CustomList<>();

    assertTrue(list.isEmpty());
}
```

Даний тест не проходить, оскільки метод isEmpty завжди повертає false. Ми можемо змусити його пройти, просто перекинувши повернене значення:

```
@Override
public boolean isEmpty() {
    return true;
}
```

Щоб підтвердити, що метод isEmpty повертає false, коли список не порожній, нам потрібно додати принаймні один елемент:

```
@Test
public void givenNonEmptyList_whenIsEmpty_thenFalseIsReturned() {
    List<Object> list = new CustomList<>();
    list.add(null);

    assertFalse(list.isEmpty());
}
```

Тепер потрібна реалізація методу add. Ось метод додавання, який ми починаємо:

```
@Override
public boolean add(E element) {
    return false;
}
```

Реалізація цього методу не працює, оскільки не вносяться зміни до внутрішньої структури даних списку. Змінимо його, щоб зберегти доданий елемент:

```
@Override
public boolean add(E element) {
    internal = new Object[] { element };
    return false;
}
```

Наш тест все ще не проходить, оскільки метод isEmpty не вдосконалений. Змінимо метод:

```
@Override
public boolean isEmpty() {
    if (internal.length != 0) {
        return false;
    } else {
        return true;
    }
```

```
}
```
На даному етапі тест проходить.

Перейдемо до рефакторингу.
Обидва тести проходять, але код методу isEmpty може бути більш досконалим.
Проведемо рефакторинг.
```
@Override
public boolean isEmpty() {
    return internal.length == 0;
}
```
Ми бачимо, що тести проходять, тому реалізація методу isEmpty може бути зараз завершена.

Початкова реалізація методу size, що дозволяє компілювати клас CustomList:
```
@Override
public int size() {
    return 0;
}
```
Використовуючи метод додавання, ми можемо створити перший тест для методу size, перевіривши, що розмір списку з одним елементом дорівнює 1:
```
@Test
public void givenListWithAnElement_whenSize_thenOneIsReturned() {
    List<Object> list = new CustomList<>();
    list.add(null);

    assertEquals(1, list.size());
}
```
Тест не проходить, бо метод size повертає 0. Змінимо його:
```
@Override
public int size() {
    if (isEmpty()) {
        return 0;
    } else {
        return internal.length;
    }
}
```
Ми можемо зробити рефакторинг методу size, щоб зробити його більш досконалим:
```
@Override
public int size() {
    return internal.length;
}
```

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

Висновки:
Використовуючи TDD, ми можемо реалізовувати вимоги поетапно, зберігаючи при цьому тестове покриття на дуже високому рівні. Крім того, впровадження гарантовано перевіряється, оскільки воно було створене для проходження тестів.