МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
„КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”
НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС „ІНСТИТУТ
ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ”

ЛАБОРАТОРНА РОБОТА №5

з курсу: *«Проектування інформаційних систем*»

на тему: „ **Модульне тестування (Unit-тести) та рефакторинг.***”*

виконав: студент IV курсу
групи ДА-71
Кузнєцов О.А.

КИЇВ
2020

Мета роботи: оволодіти навичками створення програмного забезпечення за метолологією TDD та ознайомитися з процедурами рефакторинга.

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:

Ітерація 1 :

1) Створимо тестовий приклад програми, який будемо тестувати – простий симулятор автомобіля мовою програмування Python.

```
class Car:

    def __init__(self, speed=0):
        self.speed = speed
        self.odometer = 0
        self.time = 0

    def say_state(self):
        print("I'm going {} kph!".format(self.speed))

    def accelerate(self):
        self.speed += 5

    def brake(self):
        self.speed -= 5

    def step(self):
        self.odometer += self.speed
        self.time += 1

    def average_speed(self):
        if self.time != 0:
            return self.odometer / self.time
        else:
            pass
```

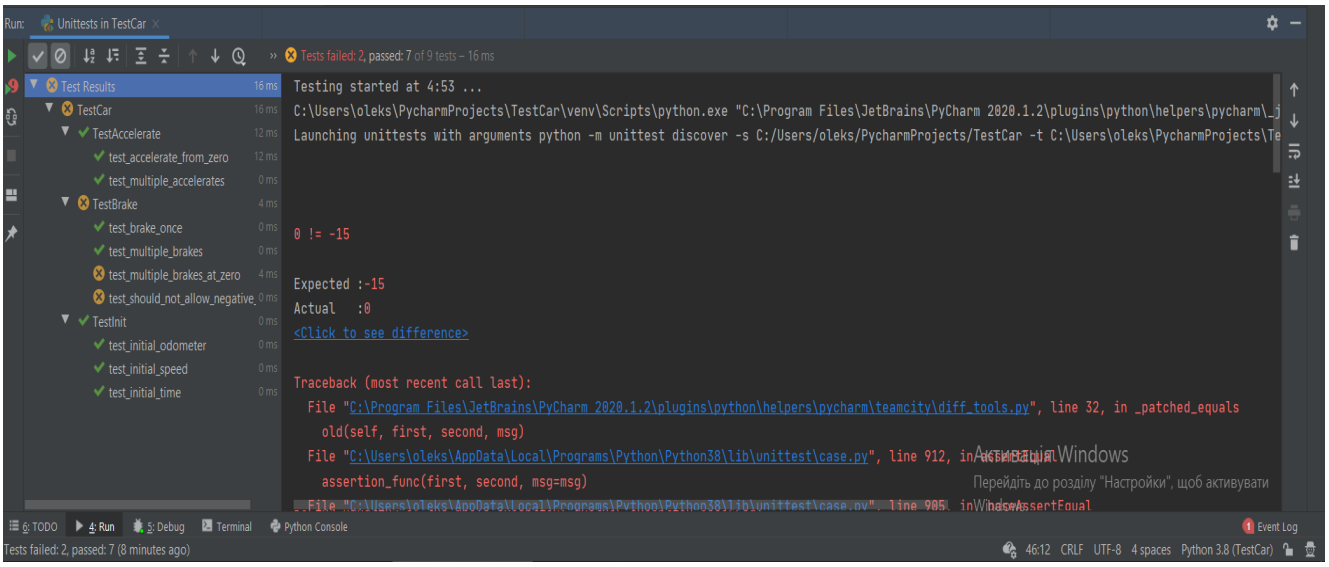
2) Тепер розробимо Unit-тести, що перевіряють функціональність програми, а саме :
1. Перевірка чи створюється клас машина
2. Перевірка ініціалізації данниз – швидкість, одометр і час

3. Тест класу Accelerate
4. Тест клазу Brake

```
import unittest

from Car import Car


class TestCar(unittest.TestCase):
      def setUp(self):
          self.car = Car()


class TestInit(TestCar):
      def test_initial_speed(self):
          self.assertEqual(self.car.speed, 0)

      def test_initial_odometer(self):
          self.assertEqual(self.car.odometer, 0)

      def test_initial_time(self):
          self.assertEqual(self.car.time, 0)


class TestAccelerate(TestCar):
      def test_accelerate_from_zero(self):
          self.car.accelerate()
          self.assertEqual(self.car.speed, 5)

      def test_multiple_accelerates(self):
          for _ in range(3):
            self.car.accelerate()
          self.assertEqual(self.car.speed, 15)


class TestBrake(TestCar):
       def test_brake_once(self):
           self.car.accelerate()
           self.car.brake()
           self.assertEqual(self.car.speed, 0)

       def test_multiple_brakes(self):
            for _ in range(5):
                self.car.accelerate()
            for _ in range(3):
                self.car.brake()
            self.assertEqual(self.car.speed, 10)

       def test_should_not_allow_negative_speed(self):
           self.car.brake()
           self.assertEqual(self.car.speed, 0)

       def test_multiple_brakes_at_zero(self):
           for _ in range(3):
               self.car.brake()
           self.assertEqual(self.car.speed, 0)
```

Тобто виходить що у нас буде проведено 9 тестів

3) За допомогою вбудованого функціоналу в IDE для Python PyCharm проведемо ці тести і подивимося на результати :

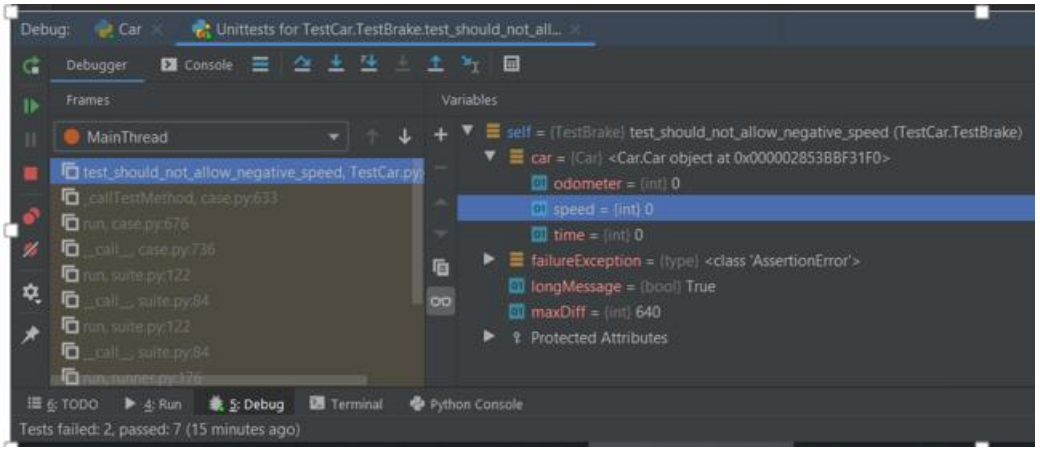


Отримали 2 провалених теста : тест на перевірку того чи може швидкість впасти нижче нуля і тест на багатоповторний виклик функції гальма, інші тести були виконані.

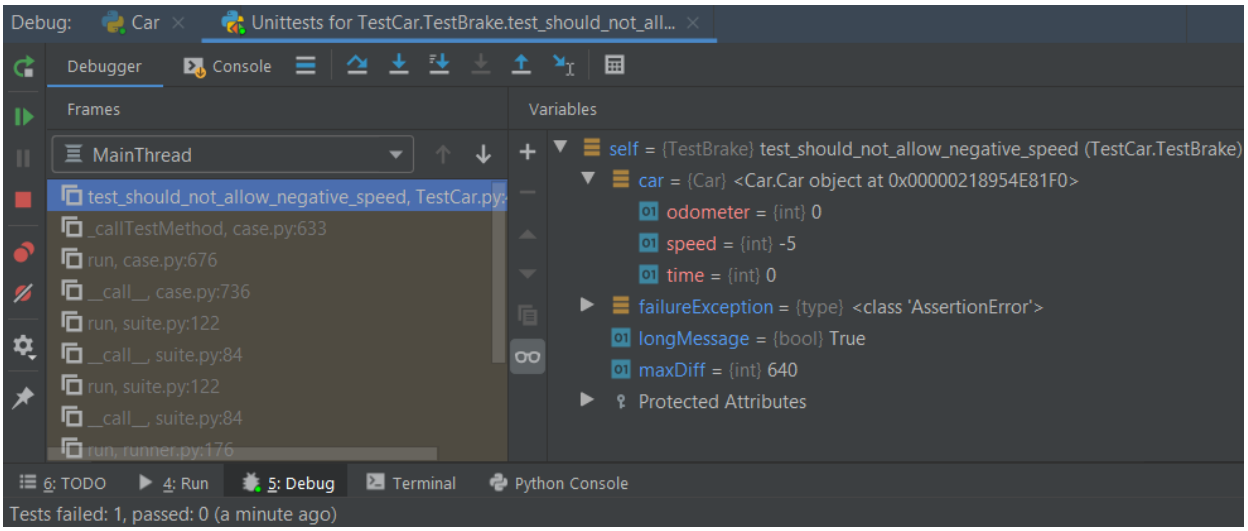
Ітерація 2 :

1) Проведемо дебаг на основі отриманих результатів за допомогою можливостей PyCharm.

```
    def test_should_not_allow_negative_speed(self):  self: test_should_not_allow_negative_speed (TestCar.TestBrake)
        self.car.brake()
        self.assertEqual(self.car.speed, 0)
```



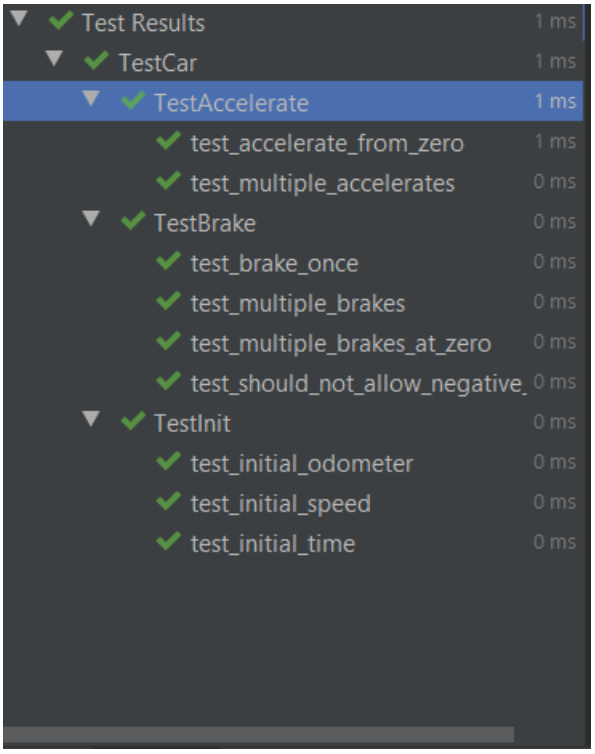
```
def brake(self):  self: <Car.Car object at 0x000002853BBF31F0>
    self.speed -= 5
```



Це показує, що швидкість може стати негативною, що неможливо.
2) Змінимо код в класі Brake як показано нижче :

```
def brake(self):
    if self.speed < 5:
        self.speed = 0
    else:
        self.speed -= 5
```

І ще раз запустимо Unit – тести :



Ітерація 3 :

Рефакторінг :

1) Вигляд коду до рефакторінгу

```
class Car:
def __init__(self, speed=0):
 self.speed = speed
 self.odometer = 0
 self.time = 0
def say_state(self):
 print("I'm going {} kph!".format(self.time))
def accelerate(self):
 self.speed += 5
def brake(self):
 if self.speed < 5:
  self.speed = 0
 else:
  self.speed -= 5
def step(self):
 self.odometer += self.speed
 self.time += 1
def average_speed(self):
 if self.a != 0:
    return self.b / self.a
 else:
  pass
```

2) Вигляд коду після рефакторінгу :

```
class Car:

    def __init__(self, speed=0):
        self.speed = speed
```

```
        self.odometer = 0
        self.time = 0

    def say_state(self):
        print("I'm going {} kph!".format(self.speed))

    def accelerate(self):
        self.speed += 5

    def brake(self):
        if self.speed < 5:
            self.speed = 0
        else:
            self.speed -= 5

    def step(self):
        self.odometer += self.speed
        self.time += 1

    def average_speed(self):
        if self.time != 0:
            return self.odometer / self.time
        else:
            pass
```

Що було зміненно :

1. Була зроблена зрозуміла структура класу, кожен метод був виділений з відступами, щоб можна було би зрозуміти що відносяться до чого
2. В методі say_state було не коректна назва змінної (time при тому що ми виводимо швидкість). З time змінна була перейменована в speed, яка більше відносить до контексту цього методу

3. В методі average_speed були не змістовні назви змінних, які нам не давали ніякого розуміння, що відбувається в данному методі, були змінені на змістовні змінні які розкривають сутність цього методу і що він повертає для нам программі.

Висновки : в даній лабораторній я отримав навички в створенні unit-тестів, а також в подальшому їх аналізі, для того щоб виправити помилки. Також як отримав досвід у рефакторингу коду