МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

Кафедра інформаційних систем і мереж

Лабораторна робота №5

на тему

“TDD”

з дисципліни

“Екстремальне програмування”

Виконала:

студентка групи КН-311

Рибалкіна Марина

Перевірив:

к.т.н.

Щербак С.С..

Львів 2020

**Мета роботи**: ознайомитись з TDD.

**Теоретичний матеріал**:

Керована тестами розробка (КТР), Розробка через тестування (англ. Test-driven development (TDD)) — технологія розробки програмного забезпечення, яка використовує короткі ітерації розробки, що починаються з попереднього написання тестів, які визначають необхідні покращення або нові функції. Кожна ітерація має на меті розробити код, який пройде ці тести. Нарешті, програміст або група вдосконалюють код для погодження змін. Один із ключових моментів TDD полягає у тому, що підготовка тестів перед написанням самого коду пришвидшує процес внесення змін. Варто зауважити, що керована тестами розробка є методологією розробки програмного забезпечення, а не його тестування.

Test-Driven Development відноситься до концепції екстремального програмування, яка стверджує, що спершу потрібно писати тести, а вже потім код, яка веде свій початок з 1999 року, однак, останнім часом спостерігається загальніший інтерес до даної методології.

Програмісти також використовують дану методологію для вдосконалення і зневадження сирцевого коду, раніше написаного з використанням інших методологій розробки.

Вимоги

Керована тестами розробка вимагає від розробника створення автоматизованих модульних тестів, які визначають вимоги до коду безпосередньо перед написанням самого коду. Тест містить перевірки умов, які можуть або виконуватися, або ні. Коли вони виконуються, кажуть, що тест пройдено. Проходження тесту підтверджує поведінка, передбачувана програмістом. Розробники часто використовують програмні каркаси для тестування (англ. testing frameworks) для створення та автоматизації запуску наборів тестів. На практиці модульні тести покривають критичні та нетривіальні ділянки коду. Це може бути код, схильний до частих змін, код, від роботи якого залежить працездатність великої кількості іншого коду, або код з великою кількістю залежностей.

Середовище розробки повинно швидко реагувати на невеликі модифікації коду. Архітектура програми повинна базуватися на використанні безлічі сильно пов’язаних компонентів, які слабо залежать одне від одного , завдяки чому тестування коду спрощується.

TDD не тільки пропонує перевірку коректності, а й впливає на дизайн програми. Спираючись на тести, розробники можуть швидше уявити, яка функціональність необхідна користувачеві. Таким чином, деталі інтерфейсу з'являються задовго до остаточної реалізації рішення.

Зрозуміло, що до тестів застосовуються такі ж вимоги щодо якості коду, як і до основного коду.

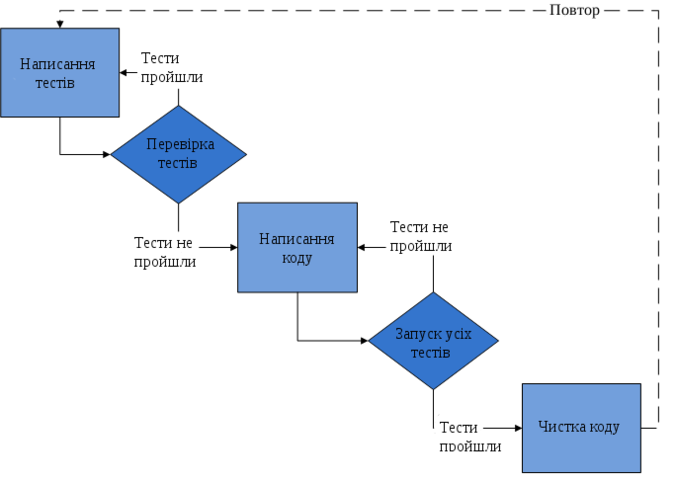
Три закони TDD

Не можна писати жодного вихідного коду, доки спершу не написано падаючого юніт-тесту (англ. unit test).

Не можна писати більше юніт-тесту ніж необхідно для падіння (непроходження тесту). Помилка компіляції - це також падіння (англ. failing).

Не можна писати більше вихідного коду, ніж необхідно для проходження впалого юніт-тесту.

Цикл розробки за методологією TDD

**

*Рис. 1. Графічна репрезентація циклу розробки TDD*

Усе нижчевказане базується на книзі Test-Driven Development by Example,[4] котру багато хто вважає канонічним текстом про дану концепцію у її сучасному вигляді.

1. Додати тест

У розробці через тестування, реалізація кожної нової функції розпочинається з написання тесту. Цей тест звісно ж не буде проходити, адже він написаний до того, як було реалізовано даний функціонал. Для того, щоб написати тест, розробник повинен чітко розуміти майбутні специфікацію та вимоги до нового функціоналу. Це основна відмінність розробки через тестування від більш класичного підходу написання тестів після того, як написано сам код: це змушує розробника фокусуватись на специфікаціях перед написанням коду.

2. Запустити усі тести, і подивитись, чи вони пройшли

Це підтверджує, що усі тести працюють коректно, і, що нові тести помилково не проходять, не вимагаючи при цьому ніякого нового коду.

Нові тести також не повинні проходити з відомої причини. Цей етап є тестуванням самих тестів на те, чи дають вони негативний результат: це виключає можливість того, що новий тест буде завжди проходити, а, отже, буде марним.

3. Написати код

Наступним кроком є написання коду, який змусить тест пройти. Новий код, написаний на даному етапі не буде досконалим, і може, наприклад, змушувати тест проходити у некоректний спосіб. Це є допустимим. тому що на наступних кроках даний код буде поліпшуватись і відточуватись.

Дуже важливим моментом є те, що код написаний виключно для того, щоб проходили тести; ніякої додаткової (а отже і не протестованої) функціональності на даному етапі вносити не дозволяється.

4. Запустити автоматичні тести, і подивитись, чи пройшли вони успішно

Якщо усі тести успішно проходять, програміст може бути впевненим у тому, що його код відповідає усім вимогам, які перевіряються тестами. Це гарна точка, з якої можна перейти на фінальний етап циклу розробки.

5. Удосконалити код

Тепер код при необхідності може бути "вичищений". Шляхом перезапуску усіх тестів, розробник може впевнитись у тому, що рефакторинг коду не порушив його відповідність специфікаціям. Концепція вилучення з коду дублікатів є важливим аспектом будь-якого дизайну програмного забезпечення.

**Хід роботи**:

Спершу було написано наступний код:

import tkinter

import unittest

from HUD import HUD

from \_VideoCapture import \_VideoCapture

class MyTestCase(unittest.TestCase):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(MyTestCase, self).\_\_init\_\_(\*args, \*\*kwargs)

self.my\_hud = HUD(tkinter.Tk(), 'VideoAnalysis', False)

# self.thread1 = threading.Thread(target=self.my\_hud.start\_loop())

def test\_set\_is\_video\_paused(self):

self.my\_hud.set\_is\_video\_paused(True)

self.assertEqual(self.my\_hud.is\_video\_paused, True)

self.my\_hud.set\_is\_video\_paused(False)

self.assertEqual(self.my\_hud.is\_video\_paused, False)

def test\_video\_capture(self):

video = \_VideoCapture()

self.assertEqual(video.video.isOpened(), True)

video = None

try:

video = \_VideoCapture("test")

self.assertEqual(video.video.isOpened(), False)

except:

self.assertEqual(video is None or not video.video.isOpened(), True)

video = \_VideoCapture("AnalyzedVideo1.avi")

self.assertEqual(video.video.isOpened(), True)

def test\_capture\_video\_1(self):

self.my\_hud.play\_video()

try:

self.my\_hud.capture\_video()

except:

self.assertEqual(True, False)

def test\_capture\_video\_2(self):

self.my\_hud.video = None

try:

self.my\_hud.play\_video("kek")

self.assertEqual(True, False)

except:

self.assertEqual(True, True)

try:

self.my\_hud.capture\_video()

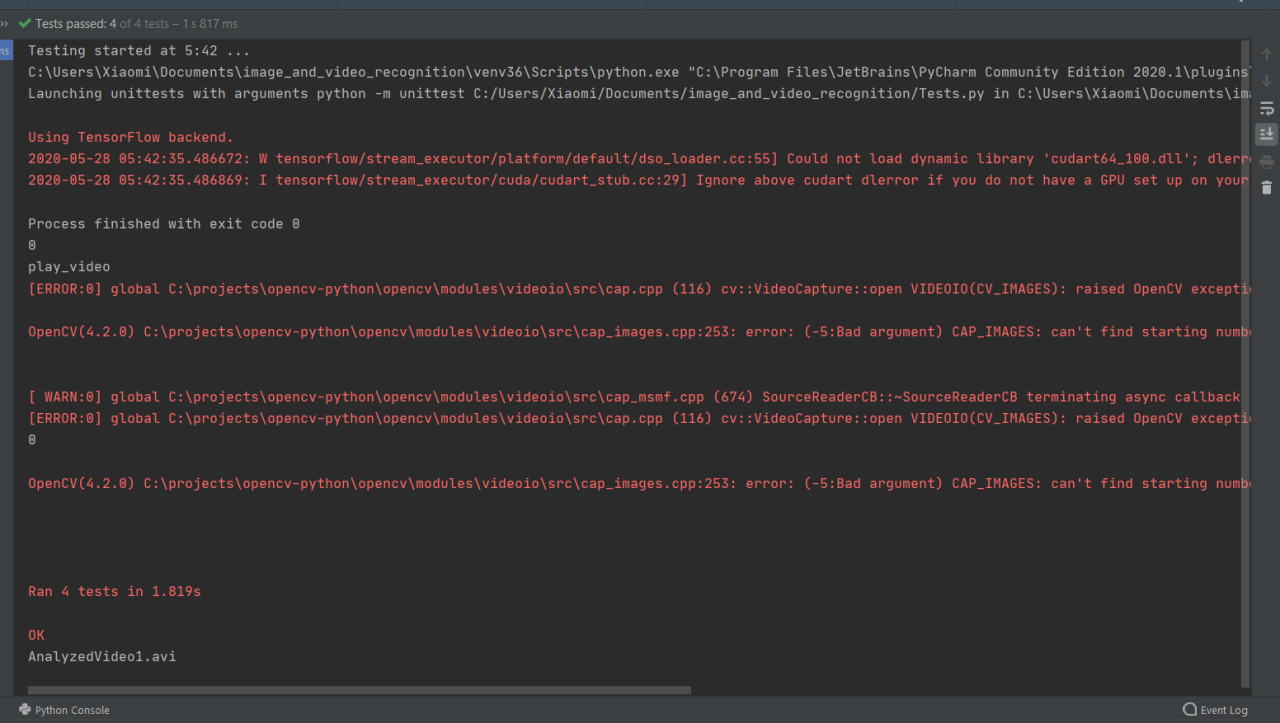
except:

self.assertEqual(True, False)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Після написання і запуску програми було отримано наступний результат:

**

*Рис. 2. Результат роботи тестування*

**Висновок**: у ході даної роботи було ознайомлено з TDD.