**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

# Курс «Базовые компоненты интернет-технологий**»**

Отчет по лабораторной работе №4

«Основные конструкции языка Python»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-32Б |  | преподаватель каф. ИУ5 |
| Панов Г.Д. |  | Гапанюк Ю. Е. |
| Подпись и дата |  | Подпись и дата |

Москва, 2021 г.

### **Задание:**

1. Необходимо для произвольной предметной области реализовать от одного до трех шаблонов проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать [следующий каталог.](https://refactoring.guru/ru/design-patterns/catalog) Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
2. Вместо реализации паттерна Вы можете написать тесты для своей программы решения биквадратного уравнения. В этом случае, возможно, Вам потребуется доработать программу решения биквадратного уравнения, чтобы она была пригодна для модульного тестирования.
3. В модульных тестах необходимо применить следующие технологии:
   * TDD - фреймворк.
   * BDD - фреймворк.
   * Создание Mock-объектов.

**Листинг программы:**

**main.py**

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
  
  
class Developer(ABC):  
  
 @abstractmethod  
 def factory\_method(self):  
 pass  
  
 def some\_operation(self) -> str:  
 # Вызываем фабричный метод, чтобы получить объект-продукт  
 product = self.factory\_method()  
  
 # Работаем с этим продуктом  
 result = f"Creator: Today the company has created a new game: {product.operation()}"  
  
 return result  
  
  
class ComputerDeveloper(Developer):  
 def factory\_method(self) -> Game:  
 return ComputerGame()  
  
  
class MobileDeveloper(Developer):  
 def factory\_method(self) -> Game:  
 return MobileGame()  
  
  
class Game(ABC):  
 @abstractmethod  
 def operation(self) -> str:  
 pass  
  
  
class ComputerGame(Game):  
 def operation(self) -> str:  
 return "Computer game created"  
  
  
class MobileGame(Game):  
 def operation(self) -> str:  
 return "Mobile game created"  
  
  
def client\_code(creator: Developer):  
  
 # print(f"Client: I'm not aware of the creator's class, but it still works.\n"  
 # f"{creator.some\_operation()}", end="")  
 result = (f"Client: I'm not aware of the creator's class, but it still works.\n"  
 f"{creator.some\_operation()}")  
 return result  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("App: Launched with the ComputerDeveloper.")  
 print(client\_code(ComputerDeveloper()))  
 print("\n")  
  
 print("App: Launched with the MobileDeveloper.")  
 print(client\_code(MobileDeveloper()))  
 print("\n")

**tests\_tdd.py**

import unittest  
from main import client\_code  
from main import ComputerDeveloper  
from main import MobileDeveloper  
from unittest.mock import patch  
  
  
  
class TestGameCreators(unittest.TestCase):  
 def test\_computer\_developer(self):  
 expected\_result = "Client: I'm not aware of the creator's class, but it still works." \  
 "\nCreator: Today the company has created a new game: Computer game created"  
 actual\_result = client\_code(ComputerDeveloper())  
 self.assertEqual(expected\_result, actual\_result, "Something is wrong with creating a computer game object")  
  
 def test\_mobile\_developer(self):  
 expected\_result = "Client: I'm not aware of the creator's class, but it still works." \  
 "\nCreator: Today the company has created a new game: Mobile game created"  
 actual\_result = client\_code(MobileDeveloper())  
 self.assertEqual(expected\_result, actual\_result, "Something is wrong with creating a mobile game object")  
  
  
class TestComputerDeveloper(unittest.TestCase):  
 @patch('main.ComputerGame.operation', return\_value="Computer game created")  
 def test\_operation(self, operation):  
 self.assertEqual(operation(self), "Computer game created")  
  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

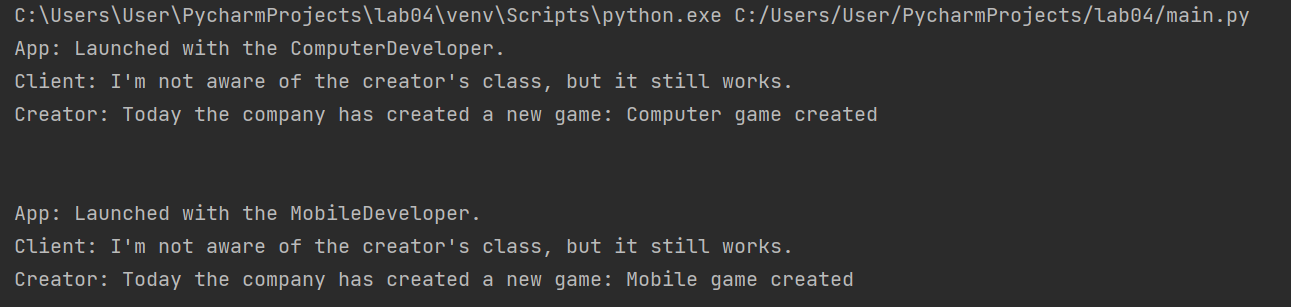
**tests\_bdd.py**

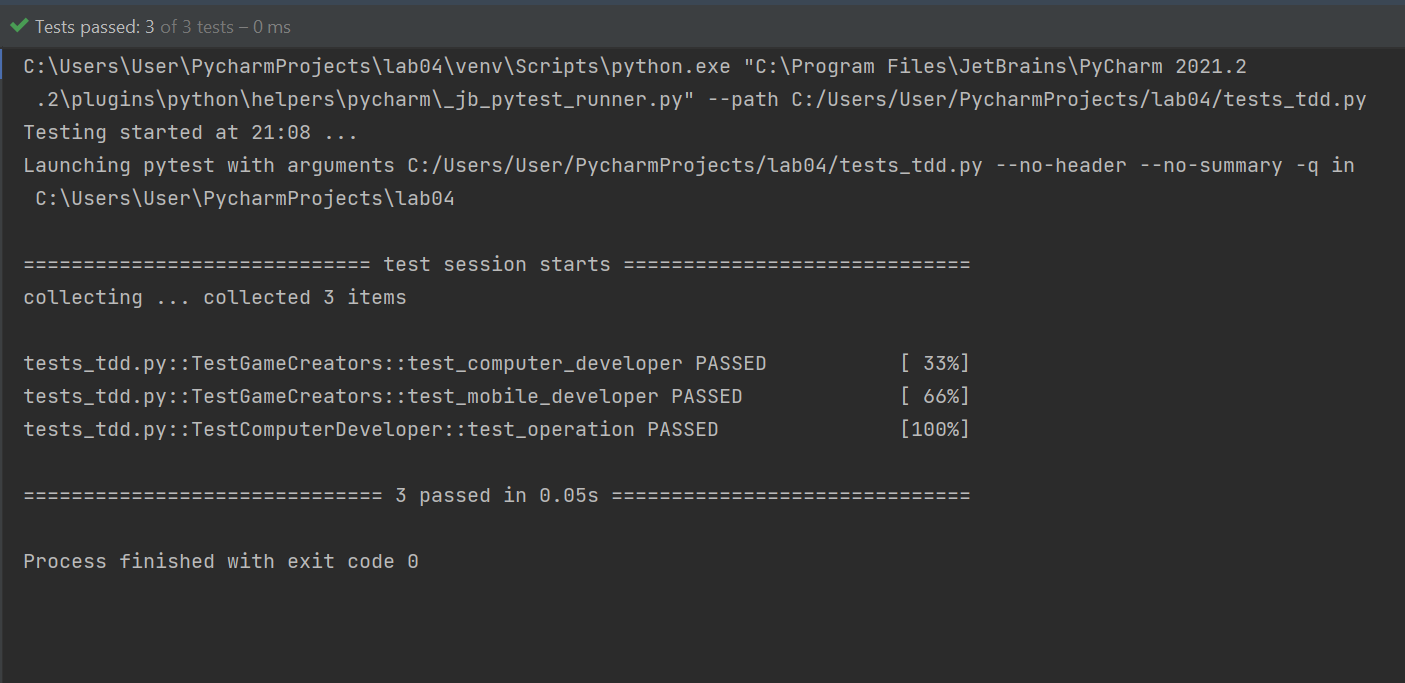
import pytest  
from pytest\_bdd import scenario, given, when, then  
  
@scenario("factory\_method.feature", "Creation of new development departments")  
def test\_start\_working():  
 pass  
  
@given("Need to create a game")  
def test\_creation():  
 pass  
  
@when("The developer creates a game")  
def test\_transfer\_to\_client():  
 pass  
  
@then("We write that the game has been successfully created")  
def test\_result():  
 pass

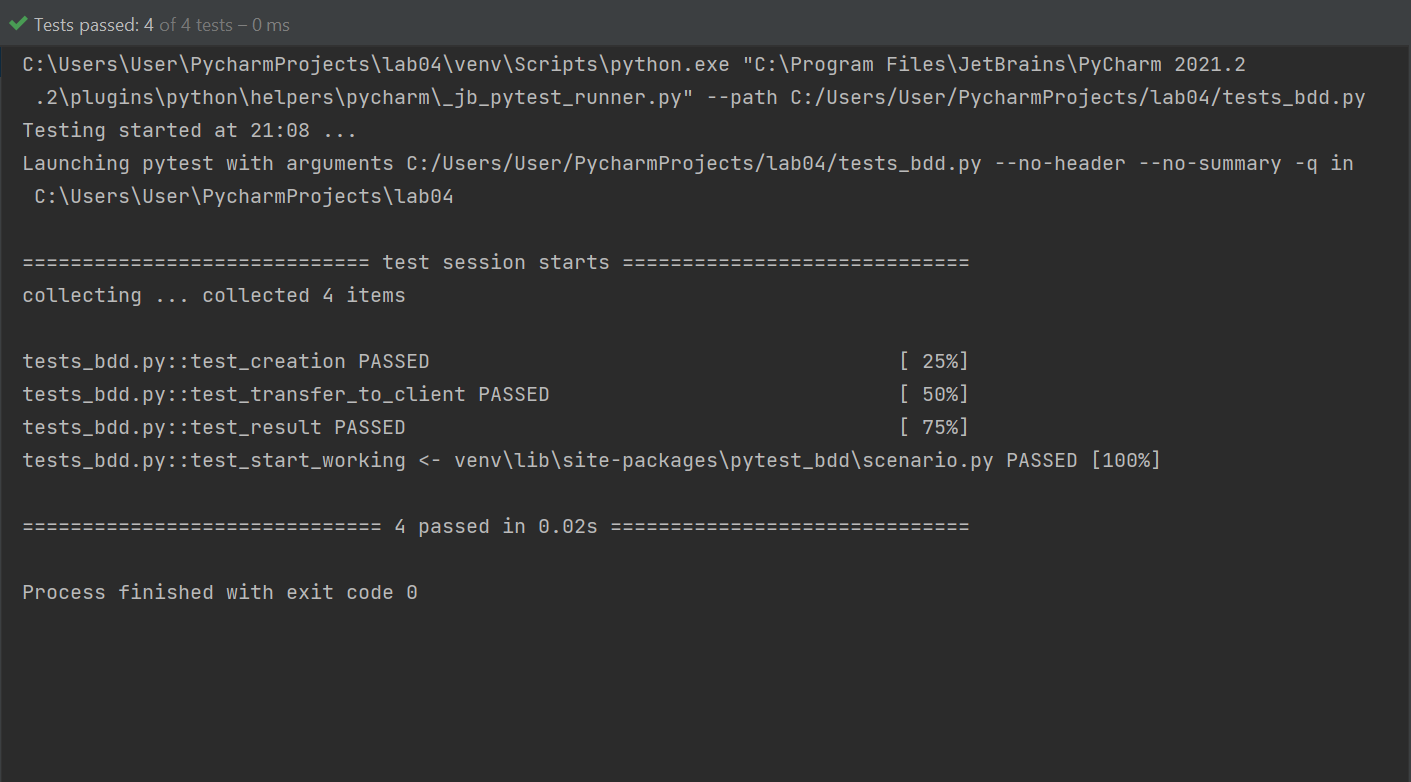
**factory\_method.feature**

Feature: GamesDeveloper  
 A game constructor pattern for developer.  
  
 Scenario: Creation of new development departments  
 Given Need to create a game  
 When The developer creates a game  
 Then We write that the game has been successfully created

**Пример работы:**

****

****

****