**Министерство науки и высшего образования Российской Федерации** **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования** **«Московский государственный технический университет** **имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

**Факультет «Информатика и системы управления»**

**Кафедра ИУ5 «Системы обработки информации и управления»**

Лабораторная работа №4

по дисциплине «Базовые компоненты интернет-технологий»

Выполнила:

студентка группы ИУ5-33Б

Румак Д.П.

Проверил:

Канев А.И.

2021 г.

**Задания**

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.

**Текст программы:**

**Файл decorator.py**

class Service():  
 *"""  
 Базовый интерфейс Компонента определяет поведение, которое изменяется  
 декораторами.  
 """* def operation(self) -> str:  
 pass  
  
  
class Service(Service):  
 *"""  
 Конкретные Компоненты предоставляют реализации поведения по умолчанию. Может  
 быть несколько вариаций этих классов.  
 """* def operation(self) -> str:  
 return "Service"  
  
  
class Decorator(Service):  
 *"""  
 Базовый класс Декоратора следует тому же интерфейсу, что и другие  
 компоненты. Основная цель этого класса - определить интерфейс обёртки для  
 всех конкретных декораторов. Реализация кода обёртки по умолчанию может  
 включать в себя поле для хранения завёрнутого компонента и средства его  
 инициализации.  
 """* \_component: Service = None  
  
 def \_\_init\_\_(self, component: Service) -> None:  
 self.\_component = component  
  
 @property  
 def component(self) -> str:  
 *"""  
 Декоратор делегирует всю работу обёрнутому компоненту.  
 """* return self.\_component  
  
 def operation(self) -> str:  
 return self.\_component.operation()  
  
  
class CoffeeService(Decorator):  
 *"""  
 Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат некоторым образом.  
 """* def operation(self) -> str:  
 *"""  
 Декораторы могут вызывать родительскую реализацию операции вместо того,  
 чтобы вызвать обёрнутый объект напрямую. Такой подход упрощает расширение классов декораторов.  
 """* return f"CoffeeService({self.component.operation()})"  
  
  
class TeaService(Decorator):  
 *"""  
 Декораторы могут выполнять своё поведение до или после вызова обёрнутого  
 объекта.  
 """* def operation(self) -> str:  
 return f"TeaService({self.component.operation()})"  
  
  
def client\_code(component: Service) -> None:  
 *"""  
 Клиентский код работает со всеми объектами, используя интерфейс Компонента.  
 Таким образом, он остаётся независимым от конкретных классов компонентов, с которыми работает.  
 """* print(f"RESULT: {component.operation()}", end="")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Таким образом, клиентский код может поддерживать как простые компоненты...  
 simple = Service()  
 print("Client: I've got a simple component:")  
 client\_code(simple)  
 print("\n")  
  
 # ...так и декорированные.  
 # декораторы могут обёртывать не только простые компоненты, но и другие декораторы.  
 decorator1 = CoffeeService(simple)  
 decorator2 = TeaService(decorator1)  
 print("Client: Now I've got a decorated component:")  
 client\_code(decorator2)

**Файл builder.py**

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
from typing import Any  
  
  
class Builder(ABC):  
  
 @property # позволяет превратить метод класса в атрибут класса  
  
 @abstractmethod # Абстрактным называется объявленный, но не реализованный метод  
 def product(self) -> None:  
 pass  
  
 @abstractmethod # Абстрактным называется объявленный, но не реализованный метод  
 def cup(self) -> None: # чашка  
 pass  
  
 @abstractmethod  
 def saucer(self) -> None: # блюдечко  
 pass  
  
 @abstractmethod  
 def spoon(self) -> None: # ложка  
 pass  
  
  
class Service\_Builder(Builder):  
  
 def \_\_init\_\_(self) -> None:  
 self.reset()  
  
 def reset(self) -> None:  
 self.\_product = ServiceShop()  
  
 @property # позволяет превратить метод класса в атрибут класса  
 def product(self) -> ServiceShop:  
 product = self.\_product  
 self.reset()  
 return product  
  
 def cup(self) -> None:  
 self.\_product.add("чашка")  
  
 def saucer(self) -> None:  
 self.\_product.add("блюдечко")  
  
 def spoon(self) -> None:  
 self.\_product.add("ложка")  
  
  
class ServiceShop():  
  
 def \_\_init\_\_(self) -> None:  
 self.parts = []  
  
 def add(self, part: Any) -> None:  
 self.parts.append(part)  
  
 def list\_parts(self) -> None:  
 print(f"В магазине продаются: {', '.join(self.parts)}", end="")  
  
  
class Director:  
  
 def \_\_init\_\_(self) -> None:  
 self.\_builder = None  
  
 @property # позволяет превратить метод класса в атрибут класса  
 def builder(self) -> Builder:  
 return self.\_builder  
  
 @builder.setter # применяем сеттер к методу builder, то есть делаем метод доступным для записи  
 def builder(self, builder: Builder) -> None:  
 self.\_builder = builder  
  
 def AnnaLafargue(self) -> None:  
 self.builder.cup()  
 self.builder.spoon()  
  
 def Luminarc(self) -> None:  
 self.builder.saucer()  
 self.builder.spoon()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 director = Director()  
 builder = Service\_Builder()  
 director.builder = builder  
  
 print("Анна Лафарг: ")  
 director.AnnaLafargue()  
 builder.product.list\_parts()  
  
 print("\n\nLuminarc: ")  
 director.Luminarc()  
 builder.product.list\_parts()

**Файл command.py**

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
  
  
class Command(ABC):  
 *"""  
 Интерфейс Команды объявляет метод для выполнения команд.  
 """* @abstractmethod  
 def execute(self) -> None:  
 pass  
  
  
class SimpleCommand(Command):  
 *"""  
 Некоторые команды способны выполнять простые операции самостоятельно.  
 """* def \_\_init\_\_(self, payload: str) -> None:  
 self.\_payload = payload  
  
 def execute(self) -> None:  
 print(f"SimpleCommand: See, I can do simple things like unpacking"  
 f"({self.\_payload})")  
  
  
class ComplexCommand(Command):  
 *"""  
 Но есть и команды, которые делегируют более сложные операции другим объектам, называемым «получателями».  
 """* def \_\_init\_\_(self, receiver: Receiver, a: str, b: str) -> None:  
 *"""  
 Сложные команды могут принимать один или несколько объектов-получателей  
 вместе с любыми данными о контексте через конструктор.  
 """* self.\_receiver = receiver  
 self.\_a = a  
 self.\_b = b  
  
 def execute(self) -> None:  
 *"""  
 Команды могут делегировать выполнение любым методам получателя.  
 """* print("ComplexCommand: Complex stuff should be done by a receiver object", end="")  
 self.\_receiver.do\_something(self.\_a)  
 self.\_receiver.do\_something\_else(self.\_b)  
  
  
class Receiver:  
 *"""  
 Классы Получателей содержат некую важную бизнес-логику. Они умеют выполнять  
 все виды операций, связанных с выполнением запроса. Фактически, любой класс  
 может выступать Получателем.  
 """* def do\_something(self, a: str) -> None:  
 print(f"\nReceiver: Working on ({a}.)", end="")  
  
 def do\_something\_else(self, b: str) -> None:  
 print(f"\nReceiver: Also working on ({b}.)", end="")  
  
  
class Invoker:  
 *"""  
 Отправитель связан с одной или несколькими командами. Он отправляет запрос команде.  
 """* \_on\_start = None  
 \_on\_finish = None  
  
 """  
 Инициализация команд.  
 """  
  
 def set\_on\_start(self, command: Command):  
 self.\_on\_start = command  
  
 def set\_on\_finish(self, command: Command):  
 self.\_on\_finish = command  
  
 def do\_something\_important(self) -> None:  
 *"""  
 Отправитель не зависит от классов конкретных команд и получателей.  
 Отправитель передаёт запрос получателю косвенно, выполняя команду.  
 """* print("Invoker: Does anybody want something done before I begin?")  
 if isinstance(self.\_on\_start, Command):  
 self.\_on\_start.execute()  
  
 print("Invoker: ...doing something really important...")  
  
 print("Invoker: Does anybody want something done after I finish?")  
 if isinstance(self.\_on\_finish, Command):  
 self.\_on\_finish.execute()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 """  
 Клиентский код может параметризовать отправителя любыми командами.  
 """  
  
 invoker = Invoker()  
 invoker.set\_on\_start(SimpleCommand("Unpacking details of service..."))  
 receiver = Receiver()  
 invoker.set\_on\_finish(ComplexCommand(  
 receiver, "Assembling of service..", "Installing the service in place.."))  
  
 invoker.do\_something\_important()

**Файл TDD\_test.py**

import unittest  
import sys, os  
from builder import \*  
  
sys.path.append(os.getcwd())  
  
  
class Service\_Builder\_Test(unittest.TestCase):  
 director = Director()  
 builder = Service\_Builder()  
 director.builder = builder  
  
 def test\_Luminarc(self):  
 print("\nLuminarc: ")  
 self.director.Luminarc()  
 self.builder.product.list\_parts()  
  
 def test\_AnnaLafargue(self):  
 print("\nАнна Лафарг: ")  
 self.director.AnnaLafargue()  
 self.builder.product.list\_parts()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

**Файл BDD\_test.py**

from behave import \*  
from TDD\_test import \*  
  
@given("Service\_Builder")  
def first\_step(context):  
 context.a = Service\_Builder\_Test()  
  
@when("test\_AnnaLafargue\_builder return OK")  
def test\_AnnaLafargue\_builder(context):  
 context.a.test\_AnnaLafargue\_builder()  
  
@when("test\_luminarc\_builder return OK")  
def test\_Luminarc\_builder(context):  
 context.a.test\_luminarc\_builder()  
  
@then("Good job")  
def last\_step(context):  
 pass

**Файл testing.feature**

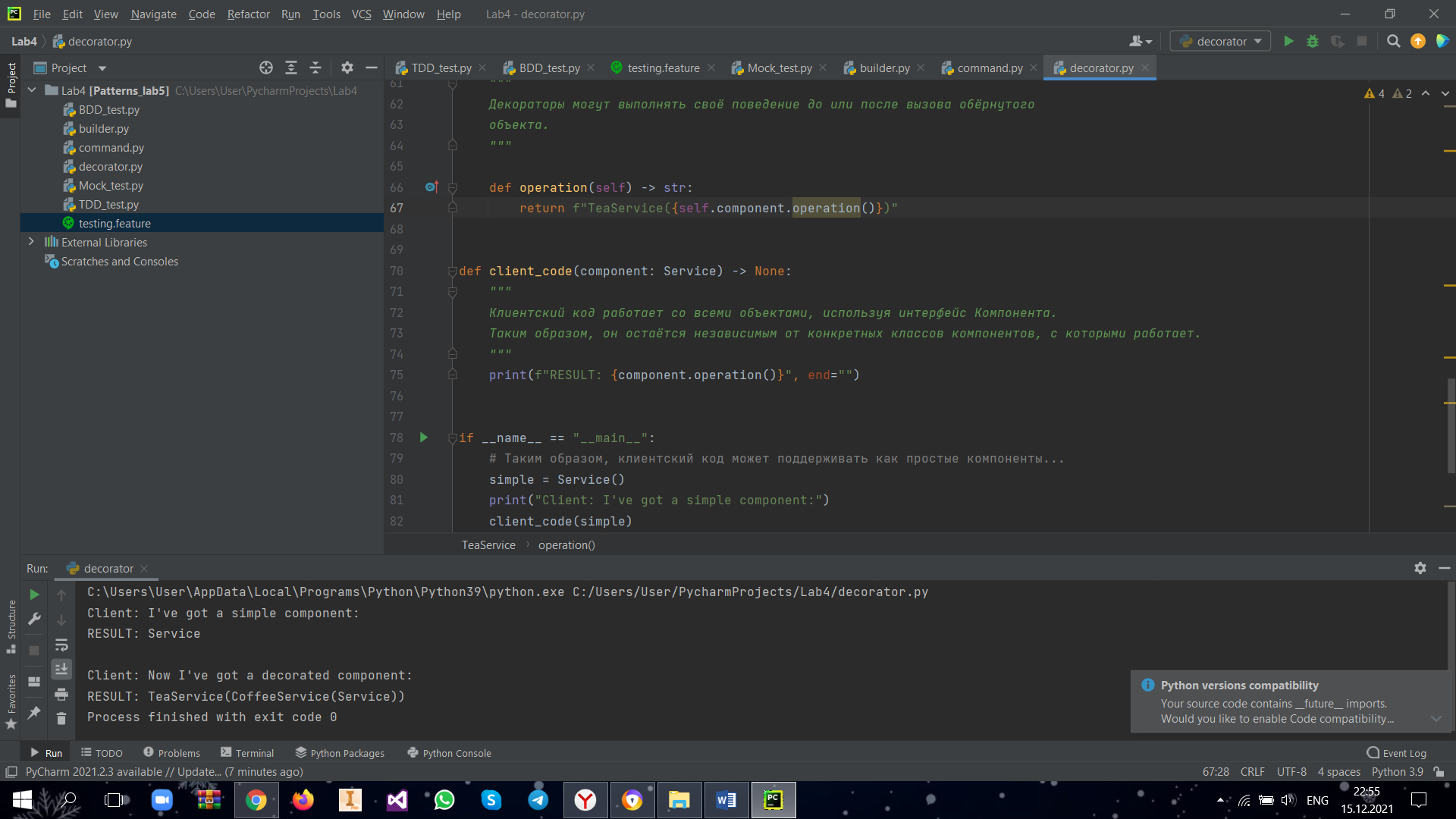
Feature: Test  
 Scenario: Test Builder  
 Given Service\_Builder  
 When test\_AnnaLafargue\_builder return OK  
 And test\_luminarc\_builder return OK  
 Then Good job

**Файл Mock\_test.py**

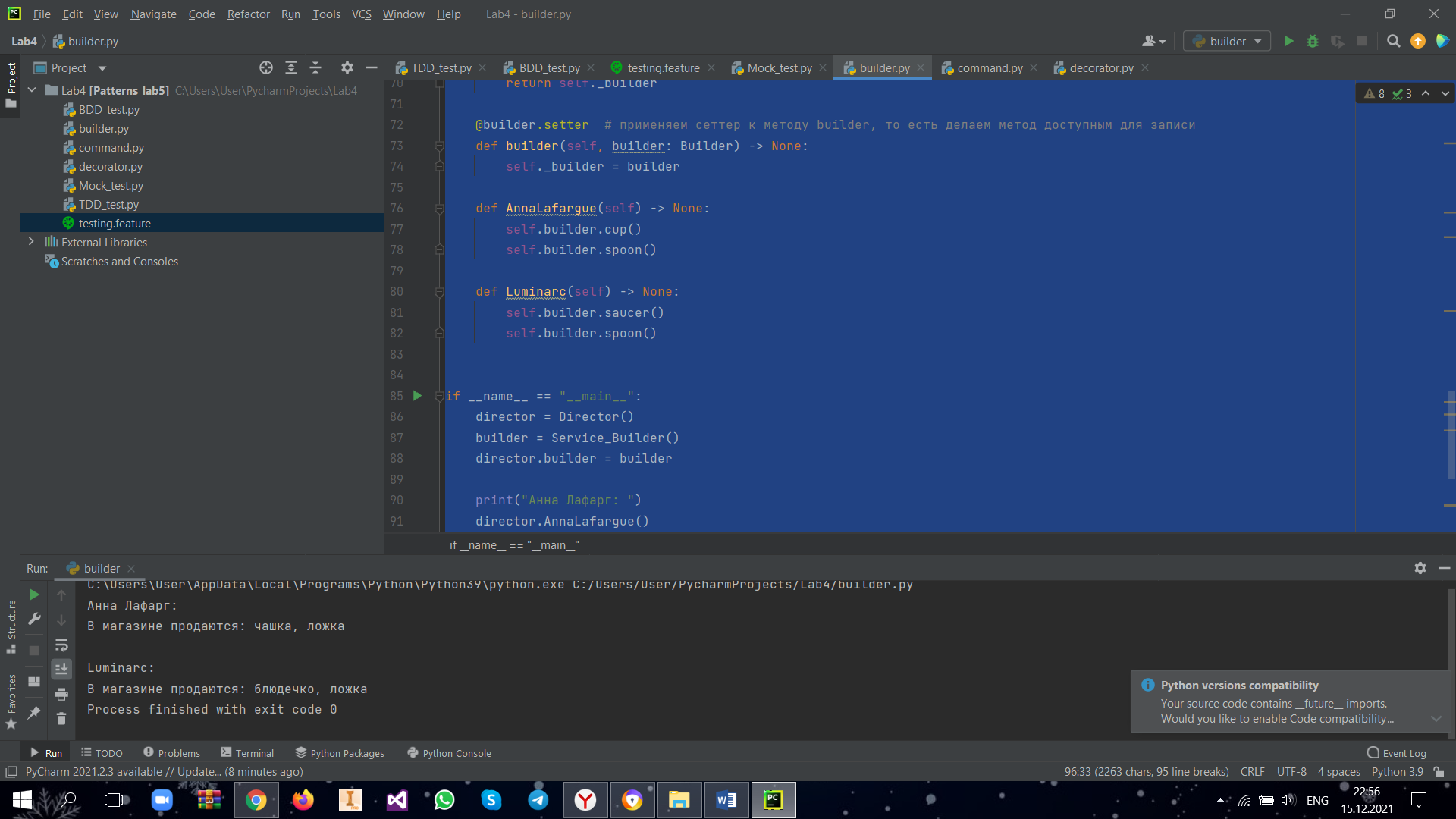
import unittest  
import sys, os  
from unittest.mock import patch, Mock  
  
import builder  
  
sys.path.append(os.getcwd())  
from builder import \*  
  
class Service\_Builder\_Test(unittest.TestCase):  
 @patch.object(builder.Service\_Builder(), 'spoon')  
 def test\_spoon(self, mock\_spoon):  
 mock\_spoon.return\_value = None  
 self.assertEqual(Service\_Builder().spoon(), None)

**Экранные формы с примерами выполнения программы:**

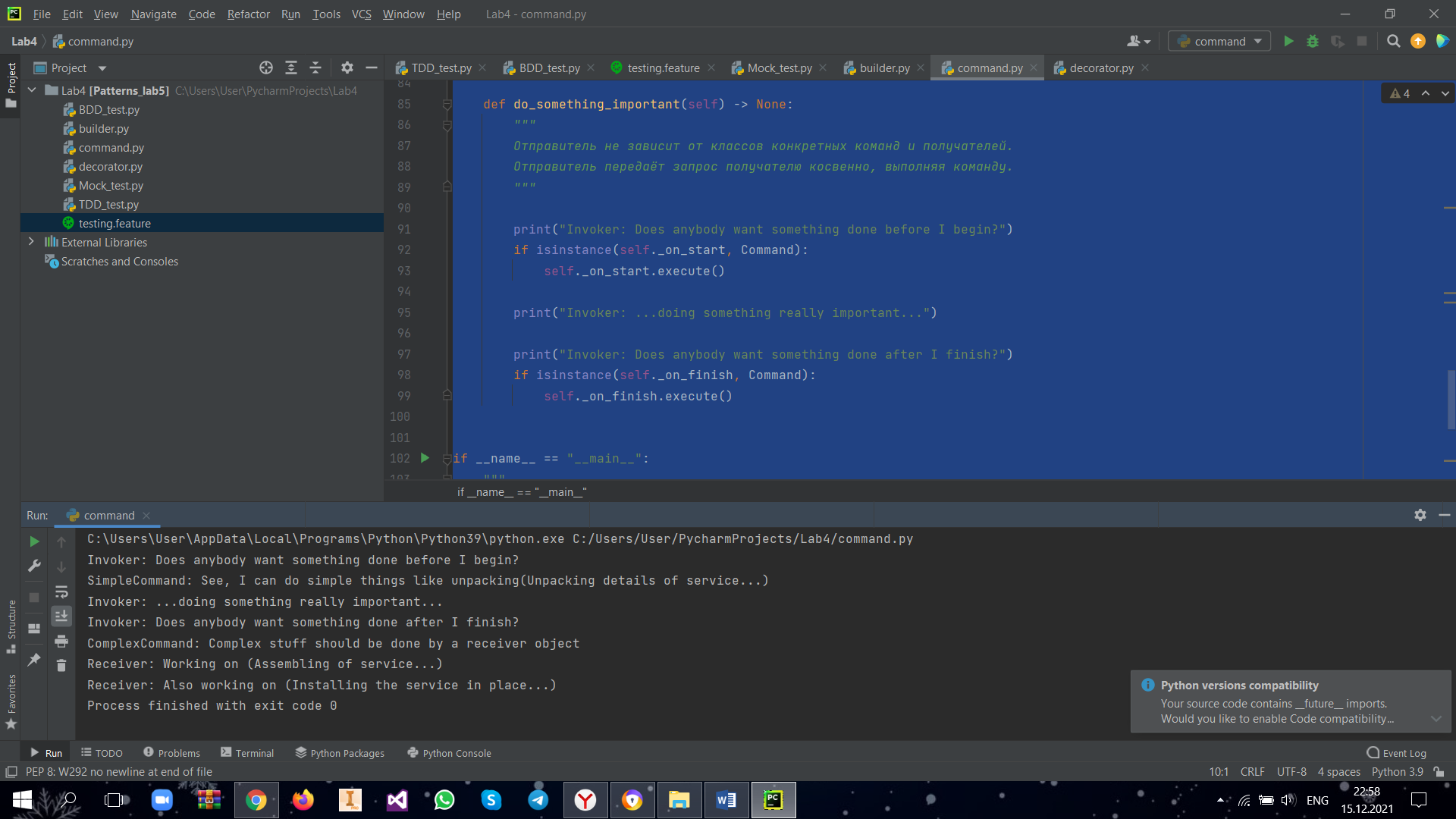
**decorator.py**

****

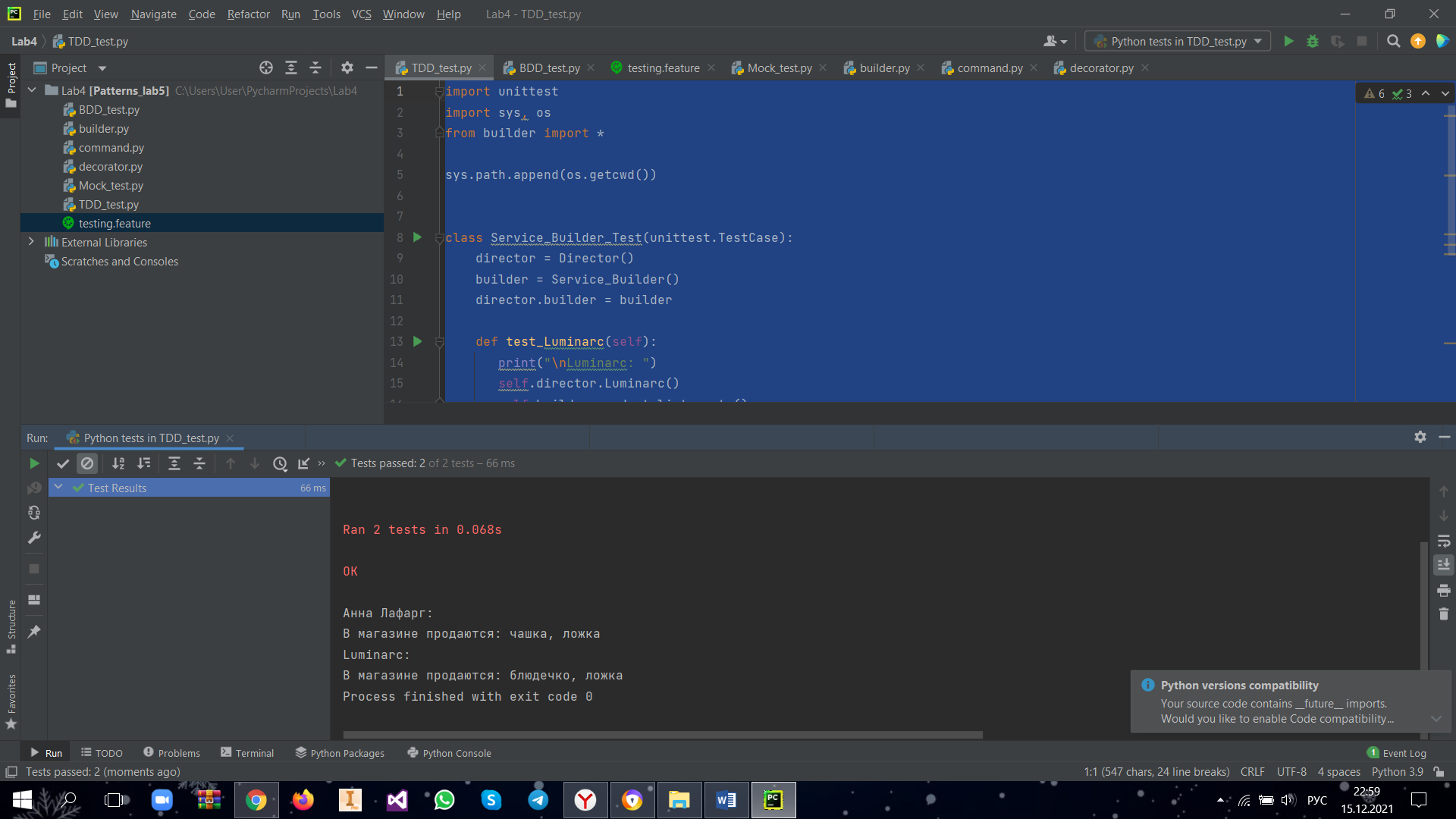
**builder.py**

****

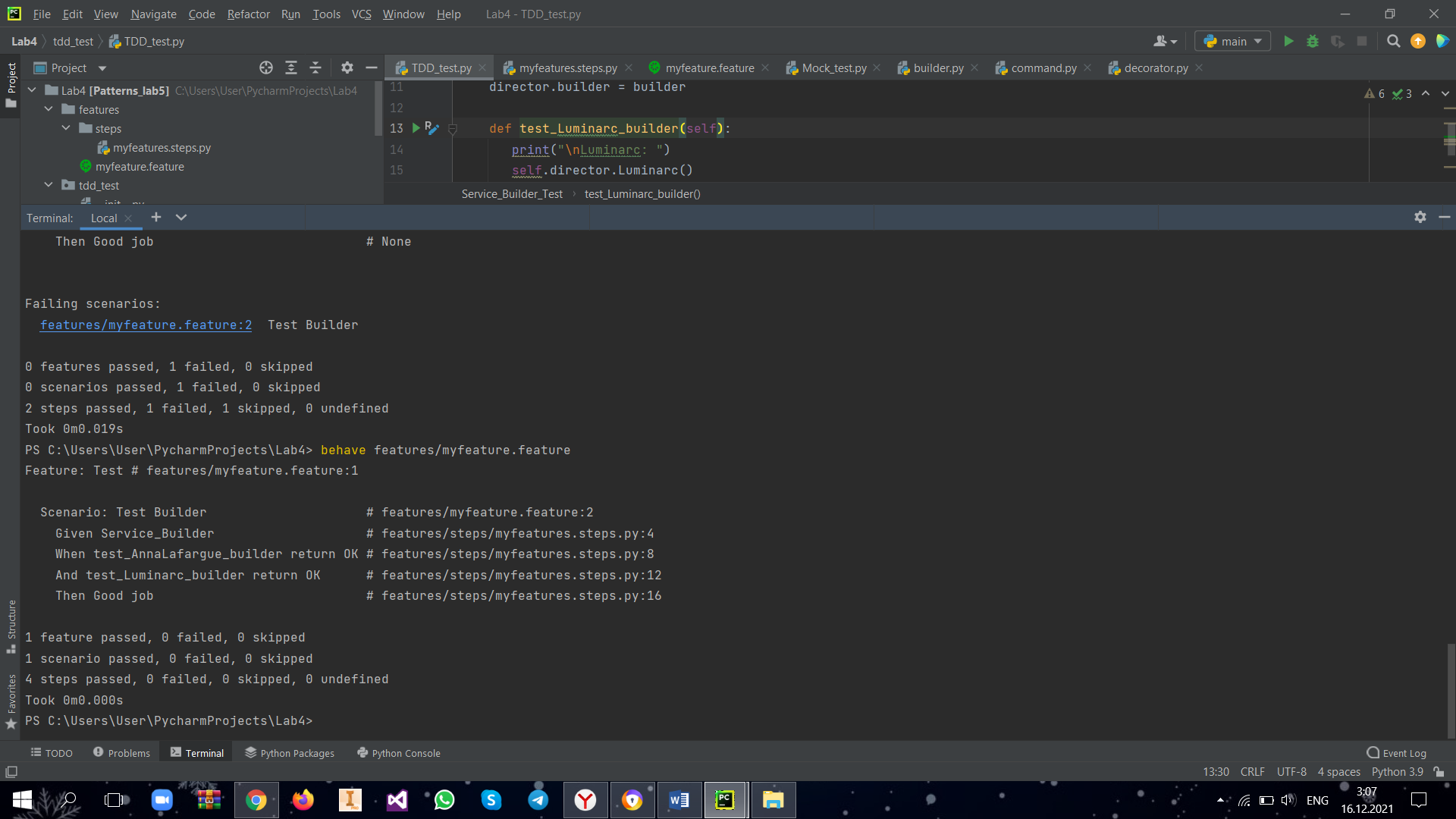
**command.py**

****

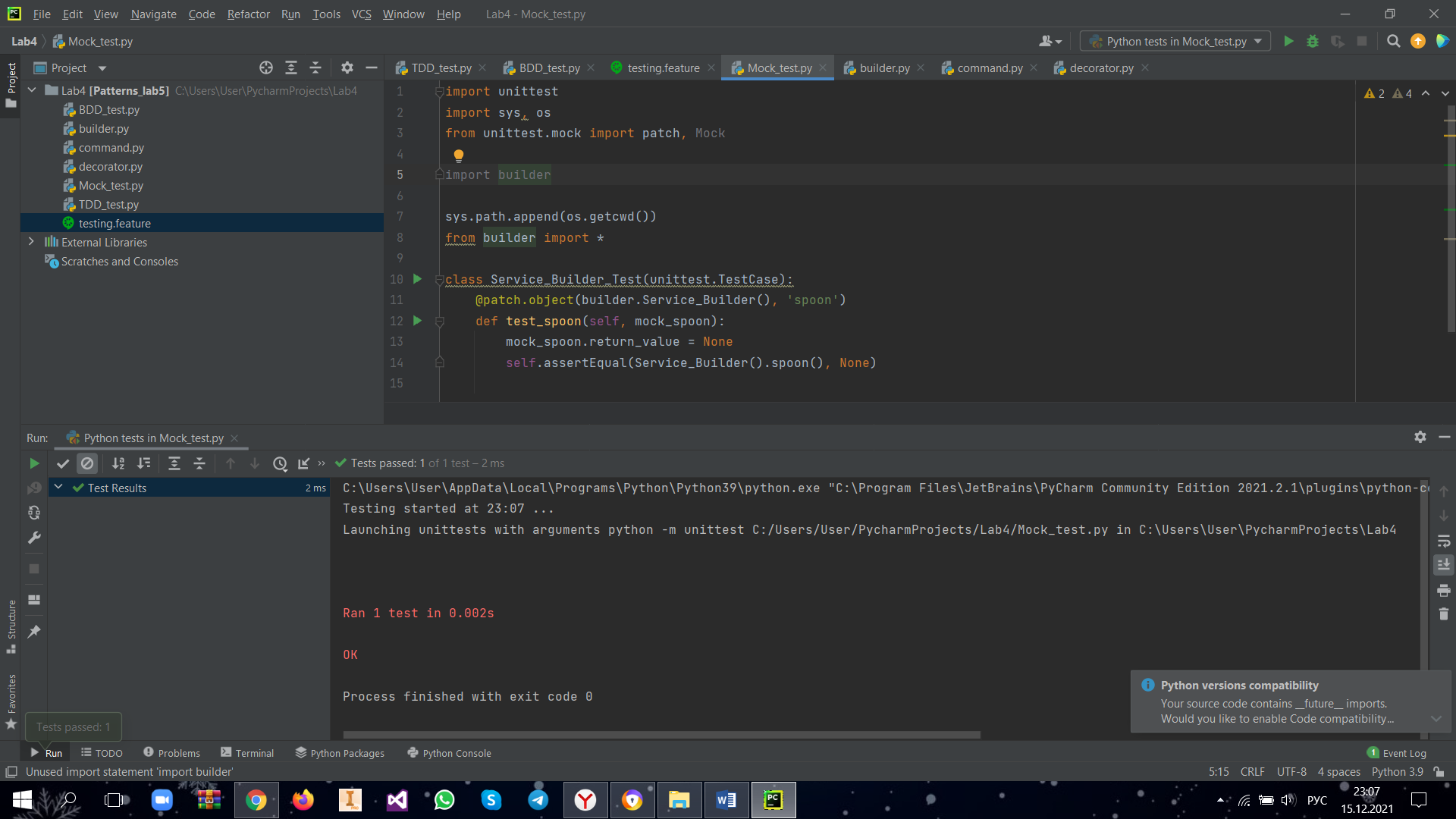
**Тестирование (TDD - фрейворк):**

****

**Тестирование (BDD - фрейворк):**

****

**Тестирование (создание Mock-объектов):**

****