**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 «ГУИМЦ»**

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

Дисциплина «Базовые компоненты ИТ»

ОТЧЕТ

# ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Функциональные возможности языка Python»

Студент: Близнева А.Е., группа ИУ5Ц-51Б

Преподаватель: Гапанюк Ю.Е.

2021г.

**Описание задания**

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. Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.

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.

В модульных тестах необходимо применить следующие технологии:

TDD - фреймворк.

BDD - фреймворк.

Создание Mock-объектов.

**Файл builder.py**

```
from abc import ABC, abstractmethod
from enum import Enum, auto


class BrandType(Enum):
    MSI = auto()
    ASUS = auto()
    ACER = auto()
    HP = auto()

class CpuType(Enum):
    I5 = auto()
    I7 = auto()
    I9 = auto()

class RamType(Enum):
    RAM8 = auto()
    RAM16 = auto()

class CardType(Enum):
    GTX1650 = auto()
    GTX1650ti = auto()
    GTX1660ti = auto()
    RTX2060 = auto()

class Laptop:
    def __init__(self, name):
        self.name = name
        self.brand = None
        self.cpu = None
        self.ram = None
        self.card = None
        self.cost = None

    def __str__(self):
       info: str = f"Laptop name: {self.name} \n" \
                   f"{self.brand} \n" \
```

```
                  f"{self.cpu} \n" \
                  f"{self.ram} \n" \
                  f"{self.card} \n" \
                  f"Cost: {self.cost} rub"

        return info


class Builder(ABC):

    @abstractmethod
    def add_brand(self) -> None: pass

    @abstractmethod
    def add_cpu(self) -> None: pass

    @abstractmethod
    def add_ram(self) -> None: pass

    @abstractmethod
    def add_card(self) -> None: pass


class MSI105LapBuilder(Builder):

    def __init__(self):
        self.laptop = Laptop("MSI 105")
        self.laptop.cost = 50000

    def add_brand(self) -> None:
        self.laptop.brand = BrandType.MSI

    def add_cpu(self) -> None:
        self.laptop.cpu = CpuType.I5

    def add_ram(self) -> None:
        self.laptop.ram = RamType.RAM8

    def add_card(self) -> None:
        self.laptop.card = CardType.GTX1650

    def get_lap(self) -> Laptop:
        return self.laptop


class ASUS101LapBuilder(Builder):

    def __init__(self):
        self.laptop = Laptop("ASUS 101")
        self.laptop.cost = 100000

    def add_brand(self) -> None:
        self.laptop.brand = BrandType.ASUS

    def add_cpu(self) -> None:
        self.laptop.cpu = CpuType.I7

    def add_ram(self) -> None:
        self.laptop.ram = RamType.RAM16

    def add_card(self) -> None:
        self.laptop.card = CardType.RTX2060
```

```
    def get_lap(self) -> Laptop:
        return self.laptop


class Director:
    def __init__(self):
        self.builder = None

    def set_builder(self, builder: Builder):
        self.builder = builder

    def make_lap(self):
        if not self.builder:
            raise ValueError("Builder didn't set")
        self.builder.add_brand()
        self.builder.add_cpu()
        self.builder.add_ram()
        self.builder.add_card()


def check_cost(name1):
    for it1 in (MSI105LapBuilder, ASUS101LapBuilder):
        director1 = Director()
        builder1 = it1()
        director1.set_builder(builder1)
        director1.make_lap()
        laptop1 = builder1.get_lap()
        if laptop1.name == name1:
            return laptop1.cost

def sum_cost(x):
    for it1 in (MSI105LapBuilder, ASUS101LapBuilder):
        director1 = Director()
        builder1 = it1()
        director1.set_builder(builder1)
        director1.make_lap()
        laptop1 = builder1.get_lap()
        x = x + laptop1.cost
    return x


if __name__ == "__main__":
    print("Объекты:")
    director = Director()
    for it in (MSI105LapBuilder, ASUS101LapBuilder):
        builder = it()
        director.set_builder(builder)
        director.make_lap()
        laptop = builder.get_lap()
        print(laptop)
        print('---------------')
    name = "ASUS 101"
    print(name, "Cost:", check_cost(name))
    x = 0
    print(sum_cost(x))
```

```
Объекты:
Laptop name: MSI 105
BrandType.MSI
CpuType.I5
RamType.RAM8
CardType.GTX1650
Cost: 50000 rub
---------------
Laptop name: ASUS 101
BrandType.ASUS
CpuType.I7
RamType.RAM16
CardType.RTX2060
Cost: 100000 rub
---------------
ASUS 101 Cost: 100000
sum =  150000
```

**tdd.py**

```python
import unittest
import sys, os

sys.path.append(os.getcwd())
from builder import *

class TestCost(unittest.TestCase):

    def test_cost(self):
        self.assertEqual(check_cost("ASUS 101"), 100000)

if __name__ == "__main__":
    unittest.main()
```

```
Ran 1 test in 0.002s

OK
Launching unittests with arguments python -m unittest
```

**Папка features**

**build.feature**

```
Feature: Test

  Scenario: Test sum_cost
    Given  I have sum = 0
    When I sum the cost
    Then I expect to get result = 150000
```

**Папка steps**

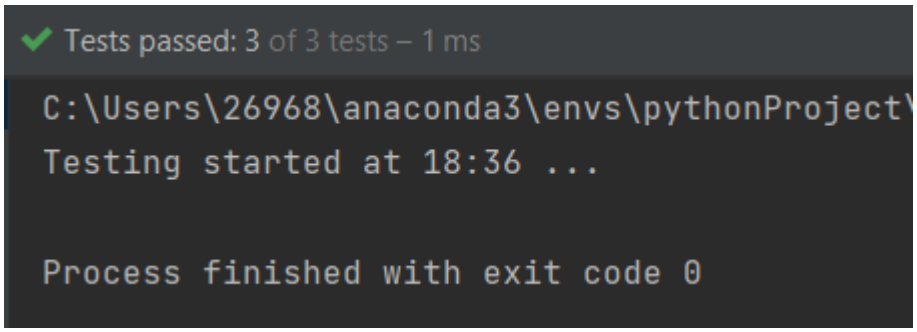
**steps.py**

```
from behave import given, when, then
from builder import *

@given('I have sum = {x:g}')
def step(context, x):
    context.x = x

@when('I sum the cost')
def step(context):
    context.x = sum_cost(context.x)

@then('I expect to get result = {result:g}')
def step(context, result):
    assert context.x == result
```




**mock.py**

```
from builder import *
from unittest import TestCase
from unittest.mock import patch

class TestCost(TestCase):
    @patch('builder.sum_cost', return_value=150000)
    def test_sum_cost(self, x):
        self.assertEqual(sum_cost(0), 150000)
```