**Московский государственный технический**

**университет им. Н.Э. Баумана.**

Факультет «Информатика и управление»

Кафедра ИУ5. Курс «Основы программирования»

Отчет по лабораторной работе №4

Выполнила: Кичикова Александра

Группа: ИУ5- 32Б

Москва, 2021 г.

Задание:

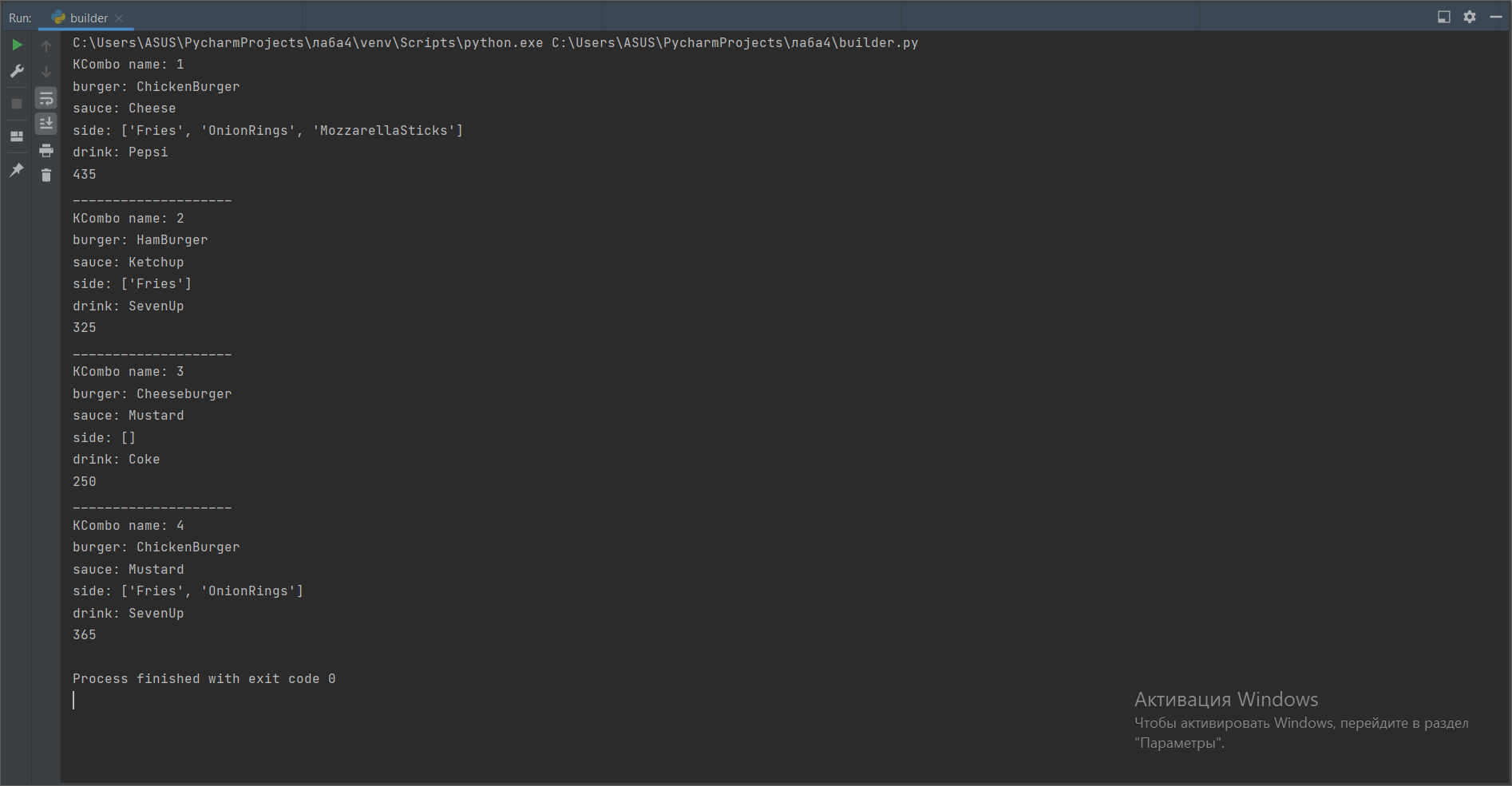
1. Необходимо для произвольной предметной области реализовать от одного до трех шаблонов проектирования: один порождающий, один структурный и один поведенческий. Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
2. Вместо реализации паттерна Вы можете написать 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.

Было реализовано три шаблона проектирования: строитель, композитор, цепь обязанностей, tdd- и bdd-тестирование были реализованы только для строителя, mock-объект был применен для композитора, но, так как было разрешено создать и тестировать только один паттерн, тестирование для композитора реализовано не было.

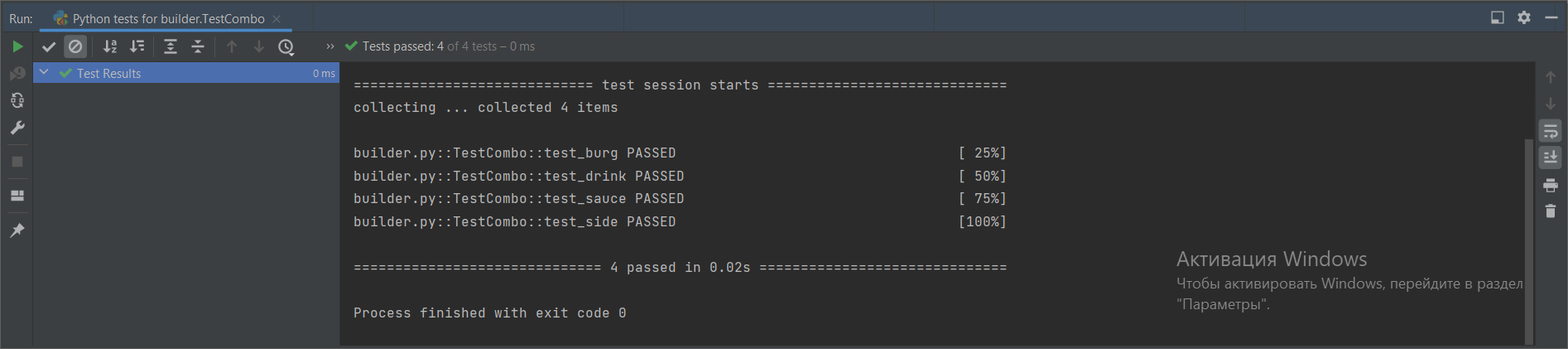
**Builder.py**

from enum import Enum  
import unittest  
  
class Burger(Enum):  
 ChickenBurger = 120  
 HamBurger = 130  
 Cheeseburger = 140  
  
  
class Sauce(Enum):  
 Ketchup = 35  
 Cheese = 30  
 Mustard = 25  
  
  
class Side(Enum):  
 Fries = 70  
 Nuggets = 80  
 OnionRings = 60  
 MozzarellaSticks = 75  
  
  
class Drink(Enum):  
 Pepsi = 80  
 Coke = 85  
 SevenUp = 90  
  
  
class KCombo:  
 def \_\_init\_\_(self, builder):  
 self.name = builder.name  
 self.burger = builder.burger  
 self.burgercost = builder.burgercost  
 self.sauce = builder.sauce  
 self.saucecost = builder.saucecost  
 self.side = builder.side  
 self.sidecost = builder.sidecost  
 self.drink = builder.drink  
 self.drinkcost = builder.drinkcost  
  
 @staticmethod  
 def getBuilder():  
 return Builder()  
  
 def \_\_str\_\_(self):  
 info: str = f"KCombo name: {self.name} \n" \  
 f"burger: {self.burger}\n" \  
 f"sauce: {self.sauce} \n" \  
 f"side: {[it.name for it in self.side]} \n" \  
 f"drink: {self.drink}"  
 return info  
  
 def cost(self):  
 cost = self.burgercost + self.saucecost + self.sidecost + self.drinkcost  
 return cost  
  
  
class Builder:  
  
 def set\_name(self, name: str):  
 self.name = name  
  
 def set\_burger(self, burg: Burger) -> None:  
 self.burger = burg.name  
 self.burgercost = burg.value  
  
 def set\_sauce(self, sauce: Sauce) -> None:  
 self.sauce = sauce.name  
 self.saucecost = sauce.value  
  
 def set\_side(self, side: list) -> None:  
 self.side = side  
 self.sidecost = 0  
 for i in side:  
 self.sidecost += i.value  
  
 def set\_drink(self, drink: Drink) -> None:  
 self.drink = drink.name  
 self.drinkcost = drink.value  
  
 def build(self):  
 return KCombo(self)  
  
  
class TestCombo(unittest.TestCase):  
  
 def test\_burg(self):  
 self.b = KCombo.getBuilder()  
 self.b.set\_burger(Burger.ChickenBurger)  
 self.assertEqual(self.b.burger, 'ChickenBurger')  
 self.assertEqual(self.b.burgercost, 120)  
  
 def test\_sauce(self):  
 self.b = KCombo.getBuilder()  
 self.b.set\_sauce(Sauce.Ketchup)  
 self.assertEqual(self.b.sauce, 'Ketchup')  
 self.assertEqual(self.b.saucecost, 35)  
  
 def test\_side(self):  
 self.b = KCombo.getBuilder()  
 self.b.set\_side([it for it in (Side.Fries,)])  
 for it in self.b.side:  
 self.assertTrue(it.name == 'Fries')  
 self.assertEqual(self.b.sidecost, 70)  
  
 def test\_drink(self):  
 self.b = KCombo.getBuilder()  
 self.b.set\_drink(Drink.Coke)  
 self.assertEqual(self.b.drink, 'Coke')  
 self.assertEqual(self.b.drinkcost, 85)  
  
 def test\_drink(self):  
 self.b = KCombo.getBuilder()  
 self.b.set\_name('Test')  
 self.b.set\_burger(Burger.ChickenBurger)  
 self.b.set\_sauce(Sauce.Ketchup)  
 self.b.set\_side([it for it in (Side.Fries,)])  
 self.b.set\_drink(Drink.Coke)  
 self.kcombo = self.b.build()  
 self.assertEqual(self.kcombo.cost(), 310)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 builder = KCombo.getBuilder()  
 builder.set\_name('1')  
 builder.set\_burger(Burger.ChickenBurger)  
 builder.set\_sauce(Sauce.Cheese)  
 builder.set\_side([it for it in (Side.Fries, Side.OnionRings, Side.MozzarellaSticks,)])  
 builder.set\_drink(Drink.Pepsi)  
 kcombo = builder.build()  
 print(kcombo)  
 print(kcombo.cost())  
 print('\_' \* 20)  
 builder2 = KCombo.getBuilder()  
 builder2.set\_name('2')  
 builder2.set\_burger(Burger.HamBurger)  
 builder2.set\_sauce(Sauce.Ketchup)  
 builder2.set\_side([it for it in (Side.Fries,)])  
 builder2.set\_drink(Drink.SevenUp)  
 kcombo = builder2.build()  
 print(kcombo)  
 print(kcombo.cost())  
 print('\_' \* 20)  
 builder3 = KCombo.getBuilder()  
 builder3.set\_name('3')  
 builder3.set\_burger(Burger.Cheeseburger)  
 builder3.set\_sauce(Sauce.Mustard)  
 builder3.set\_side([])  
 builder3.set\_drink(Drink.Coke)  
 kcombo = builder3.build()  
 print(kcombo)  
 print(kcombo.cost())  
 print('\_' \* 20)  
 builder4 = KCombo.getBuilder()  
 builder4.set\_name('4')  
 builder4.set\_burger(Burger.ChickenBurger)  
 builder4.set\_sauce(Sauce.Mustard)  
 builder4.set\_side([it for it in (Side.Fries, Side.OnionRings,)])  
 builder4.set\_drink(Drink.SevenUp)  
 kcombo = builder4.build()  
 print(kcombo)  
 print(kcombo.cost())

Результаты выполнения программы:



Результаты выполнения тестов:



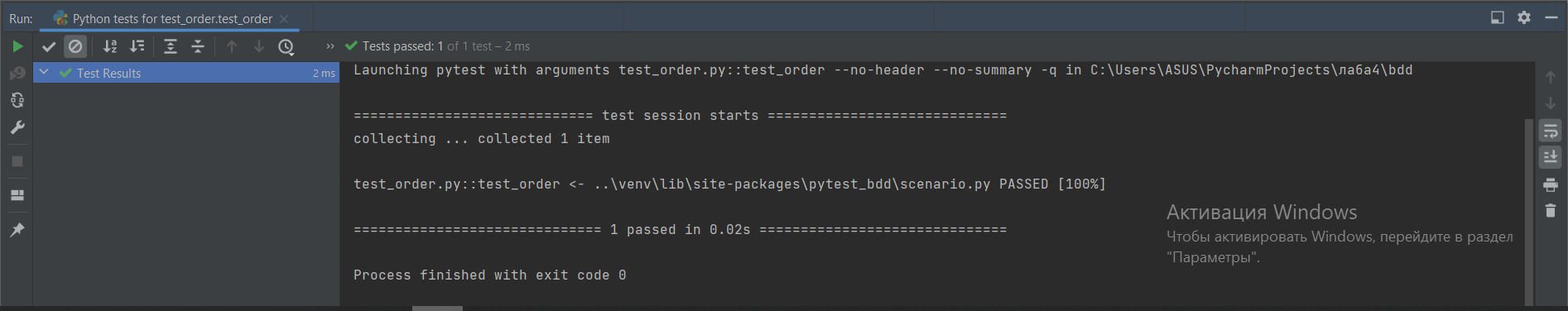
**Order.feature**

Feature: Test  
 In order to provide the best service I need to test builder  
  
 Scenario: KCombo  
 Given A customer wants a Kcombo  
  
 When For burger she orders a chickenburger  
 And For sauce she orders ketchup  
 And For side dish she orders fries and mozzarella sticks  
 And For drink she orders SevenUp  
  
 Then The orders price has to be right

**Test\_order.py**

from pytest\_bdd import scenario, given, when, then  
from builder import Burger, Sauce, Side, Drink, KCombo  
  
  
builder = KCombo.getBuilder()  
  
  
@scenario('order.feature', 'KCombo')  
def test\_order():  
 pass  
  
  
@given('A customer wants a Kcombo')  
def kc():  
 builder.set\_name('1')  
  
  
@when('For burger she orders a chickenburger')  
def burg():  
 builder.set\_burger(Burger.ChickenBurger)  
  
  
@when('For sauce she orders ketchup')  
def sauce():  
 builder.set\_sauce(Sauce.Ketchup)  
  
  
@when('For side dish she orders fries and mozzarella sticks')  
def side():  
 builder.set\_side([it for it in (Side.Fries, Side.MozzarellaSticks,)])  
  
  
@when('For drink she orders SevenUp')  
def drink():  
 builder.set\_drink(Drink.SevenUp)  
  
  
@then('The orders price has to be right')  
def order():  
 kcombo1 = builder.build()  
 builder2 = KCombo.getBuilder()  
 builder2.set\_name('1')  
 builder2.set\_burger(Burger.ChickenBurger)  
 builder2.set\_sauce(Sauce.Ketchup)  
 builder2.set\_side([it for it in (Side.Fries, Side.MozzarellaSticks,)])  
 builder2.set\_drink(Drink.SevenUp)  
 kcombo2 = builder2.build()  
 assert kcombo1.cost() == kcombo2.cost()

Результаты выполнения теста:



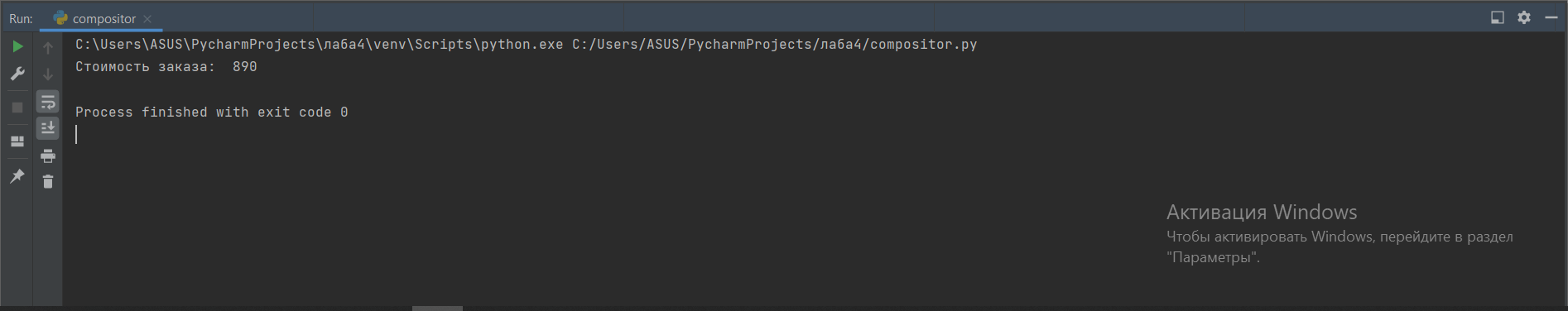
**Mockbuilder.py**

from enum import Enum  
  
class Burger(Enum):  
 ChickenBurger = 120  
 HamBurger = 130  
 Cheeseburger = 140  
  
  
class Sauce(Enum):  
 Ketchup = 35  
 Cheese = 30  
 Mustard = 25  
  
  
class Side(Enum):  
 Fries = 70  
 Nuggets = 80  
 OnionRings = 60  
 MozzarellaSticks = 75  
  
  
class Drink(Enum):  
 Pepsi = 80  
 Coke = 85  
 SevenUp = 90  
  
  
class KCombo:  
 def \_\_init\_\_(self, builder):  
 self.name = '1'  
  
 @staticmethod  
 def getBuilder():  
 return Builder()  
  
 def \_\_str\_\_(self):  
 info: str = f"KCombo name: 1\n" \  
 f"burger: ChickenBurger\n" \  
 f"sauce: Cheese\n" \  
 f"side: [Side.Fries, Side.OnionRings, Side.MozzarellaSticks] \n" \  
 f"drink: Pepsi"  
 return info  
  
 def cost(self):  
 return 435  
  
  
class Builder:  
  
 def set\_name(self, name: str):  
 pass  
  
 def set\_burger(self, burg: Burger) -> None:  
 pass  
  
 def set\_sauce(self, sauce: Sauce) -> None:  
 pass  
  
 def set\_side(self, side: list) -> None:  
 pass  
  
 def set\_drink(self, drink: Drink) -> None:  
 pass  
  
 def build(self):  
 return KCombo(self)

**Compositor.py**

from abc import ABC, abstractmethod  
from mockbuilder import Burger, Sauce, Side, Drink, KCombo  
  
  
class I(ABC):  
 @abstractmethod  
 def cost(self) -> int:  
 pass  
  
 @abstractmethod  
 def name(self) -> str:  
 pass  
  
  
class Product(I):  
 def \_\_init\_\_(self, name: str, cost: int):  
 self.\_\_cost = cost  
 self.\_\_name = name  
  
 def cost(self) -> int:  
 return self.\_\_cost  
  
 def name(self) -> str:  
 return self.\_\_name  
  
  
class Item(I):  
 def \_\_init\_\_(self, name: str):  
 self.\_\_name = name  
 self.products = []  
  
 def cost(self):  
 cost = 0  
 for it in self.products:  
 cost += it.cost()  
 return cost  
  
 def name(self) -> str:  
 return self.\_\_name  
  
 def add\_item(self, item):  
 self.products.append(item)  
  
 def remove\_item(self, item):  
 self.products.remove(item)  
  
 def clear(self):  
 self.products = []  
  
  
class Order(Item):  
 def \_\_init\_\_(self, name: str):  
 super(Order, self).\_\_init\_\_(name)  
  
 def cost(self):  
 cost = 0  
 for it in self.products:  
 ci = it.cost()  
 cost += ci  
 return cost  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 builder= KCombo.getBuilder()  
 builder.set\_name('1')  
 builder.set\_burger(Burger.ChickenBurger)  
 builder.set\_sauce(Sauce.Cheese)  
 builder.set\_side([it for it in (Side.Fries, Side.OnionRings, Side.MozzarellaSticks,)])  
 builder.set\_drink(Drink.Pepsi)  
 kcombo = builder.build()  
 Burg1 = Item('Burger1')  
 Burg1.add\_item(Product('Tomato', 12))  
 Burg1.add\_item(Product('Cheese', 20))  
 Burg1.add\_item(Product('Pickles', 23))  
 order = Order("Order1")  
 order.add\_item(Product('BigPepsi', 150))  
 order.add\_item(kcombo)  
 order.add\_item(Burg1)  
 order.add\_item(Product('KTwister', 250))  
 print(order.cost())

Результаты выполнения программы:



**Chain\_of\_responsibility.py**

from abc import ABC, abstractmethod  
from enum import Enum  
from typing import List, Optional, TypeVar  
from compositor import Product, Item, Order  
  
T = TypeVar("T")  
  
  
class EnumOrder(Enum):  
 OneEasyItem = 1  
 ComplexOrder = 2  
 NotOrder = 3  
  
  
class RequestOrder:  
  
 def \_\_init\_\_(self, order: Order, order\_type: EnumOrder):  
 self.\_\_order = order  
 self.\_\_order\_type = order\_type  
  
 @property  
 def order\_type(self):  
 return self.\_\_order\_type  
  
 @property  
 def order(self):  
 return self.\_\_order  
  
  
class Handler(ABC):  
  
 def \_\_init\_\_(self, successor: Optional[T] = None):  
 self.\_\_successor = successor  
  
 def handle(self, request: RequestOrder) -> None:  
 res = self.\_check\_request(request.order\_type)  
 if res:  
 print('Стоимость заказа ', request.order.cost())  
 if not res and self.\_\_successor:  
 self.\_\_successor.handle(request)  
  
 @property  
 def successor(self):  
 return self.\_\_successor  
  
 @successor.setter  
 def successor(self, successor: Optional[T]):  
 self.\_\_successor = successor  
  
 @abstractmethod  
 def \_check\_request(self, request\_type: EnumOrder) -> bool:  
 ...  
  
  
class Cashier(Handler):  
  
 def \_\_init\_\_(self, successor: Handler = None):  
 super().\_\_init\_\_(successor)  
  
 def \_check\_request(self, request\_type: EnumOrder) -> bool:  
 check = request\_type in (EnumOrder.ComplexOrder, EnumOrder.OneEasyItem)  
 if check and (request\_type is EnumOrder.OneEasyItem):  
 print("Кассир отдает заказ")  
 return check  
 elif check:  
 print("Кассир передает заказ")  
 return not check  
 else:  
 print("Кассир не принимает заказ")  
 return not check  
  
  
class Kitchen(Handler):  
  
 def \_\_init\_\_(self, successor: Handler = None):  
 super().\_\_init\_\_(successor)  
  
 def \_check\_request(self, request\_type: EnumOrder) -> bool:  
 check = request\_type is EnumOrder.ComplexOrder  
 if check:  
 print("Заказ готовится")  
 else:  
 print("Заказ отклонен")  
 return check  
  
  
def request\_handler(request: RequestOrder):  
 print("\_" \* 20)  
 if cashier.handle(request):  
 print(request.order.cost())  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 kitchen = Kitchen()  
 cashier = Cashier()  
 cashier.successor = kitchen  
 Burg1 = Item('Burger1')  
 Burg1.add\_item(Product('Tomato', 12))  
 Burg1.add\_item(Product('Cheese', 20))  
 Burg1.add\_item(Product('Pickles', 23))  
 order1 = Order("Order1")  
 order1.add\_item(Product('BigPepsi', 150))  
 order1.add\_item(Burg1)  
 order1.add\_item(Product('KTwister', 250))  
 request = RequestOrder(order1, EnumOrder.ComplexOrder)  
 request\_handler(request)

Результаты выполнения программы:

