**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

Носкин С.А.

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:**

**decorator.py**

class Product():  
 *"""  
 Базовый интерфейс Компонента определяет поведение, которое изменяется  
 декораторами.  
 """* def operation(self) -> str:  
 pass  
  
  
class Product(Product):  
 *"""  
 Конкретные Компоненты предоставляют реализации поведения по умолчанию. Может  
 быть несколько вариаций этих классов.  
 """* def operation(self) -> str:  
 return "Product"  
  
  
class Decorator(Product):  
 *"""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: Product = None  
  
 def \_\_init\_\_(self, component: Product) -> None:  
 self.\_component = component  
  
 @property #превращает метод класса в атрибут класса.  
 def component(self) -> str:  
 return self.\_component  
  
 def operation(self) -> str:  
 return self.\_component.operation()  
  
  
class Potato(Decorator):  
 def operation(self) -> str:  
 return f"Potato({self.component.operation()})"  
  
  
class Capusta(Decorator):  
 def operation(self) -> str:  
 return f"Capusta({self.component.operation()})"  
  
  
def show(component: Product) -> None:  
 print(f"RESULT: {component.operation()}", end="")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Таким образом, клиентский код может поддерживать как простые компоненты...  
 simple = Product()  
 print("Client: I've got a simple component:")  
 show(simple)  
 print("\n")  
 # ...так и декорированные.  
 #  
 # Декораторы могут обёртывать не только простые  
 # компоненты, но и другие декораторы.  
 decorator1 = Potato(simple)  
 decorator2 = Capusta(decorator1)  
 print("Client: Now I've got a decorated component:")  
 show(decorator2)

**builder.py**

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
from typing import Any  
  
  
class Builder(ABC):  
  
 @property # 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 potato(self) -> None:   
 pass  
  
 @abstractmethod  
 def pomidor(self) -> None:   
 pass  
  
 @abstractmethod  
 def capusta(self) -> None: #Buider- сложные обьекты создаются по частям  
 pass  
#обьявление абстрактнызх классов описываме шо они естмь  
  
class Product\_Builder(Builder):  
  
 def \_\_init\_\_(self) -> None:  
 self.reset()  
  
 def reset(self) -> None:  
 self.\_product = Shop()  
  
 @property # property позволяет превратить метод класса в атрибут класса  
 def product(self) -> Shop:  
 product = self.\_product  
 self.reset()  
 return product  
  
 def potato(self) -> None:  
 self.\_product.add("картофель")  
  
 def pomidor(self) -> None:  
 self.\_product.add("помидор")  
  
 def capusta(self) -> None:  
 self.\_product.add("капуста")  
  
#создание асбтрактных классов  
class Shop():  
  
 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="")  
  
#вывод продуктов  
class Director:  
  
 def \_\_init\_\_(self) -> None:  
 self.\_builder = None  
  
 @property # property позволяет превратить метод класса в атрибут класса  
 def builder(self) -> Builder:  
 return self.\_builder  
  
 @builder.setter # применяется сеттер к методу builder, то есть делаем метод доступным для записи  
 def builder(self, builder: Builder) -> None:  
 self.\_builder = builder  
  
 def Magnit(self) -> None:  
 self.builder.pomidor()  
 self.builder.capusta()  
  
 def ATAK(self) -> None:  
 self.builder.potato()  
 self.builder.capusta()  
#записываемп в магазин   
  
if \_\_name\_\_ == "\_\_main\_\_":  
 director = Director()  
 builder = Product\_Builder()  
 director.builder = builder  
  
 print("Magnit: ")  
 director.Magnit()  
 builder.product.list\_parts()  
  
 print("\n\nATAK: ")  
 director.ATAK()  
 builder.product.list\_parts()

**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 packets of products..."))  
 receiver = Receiver()  
 invoker.set\_on\_finish(ComplexCommand(  
 receiver, "Put the products on the shelves..", "Installing the products in place.."))  
  
 invoker.do\_something\_important()

**BDD\_test.py**

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

**TDD\_test.py**

import unittest  
import sys, os  
from builder import \*  
  
sys.path.append(os.getcwd())  
  
class Product\_Builder\_Test(unittest.TestCase):  
 director = Director()  
 builder = Product\_Builder()  
 director.builder = builder  
 def test\_Magnit\_builder(self):  
 print("Magnit: ")  
 self.director.Magnit()  
 self.builder.product.list\_parts()  
  
 def test\_ATAK\_builder(self):  
 print("\nATAK: ")  
 self.director.ATAK()  
 self.builder.product.list\_parts()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

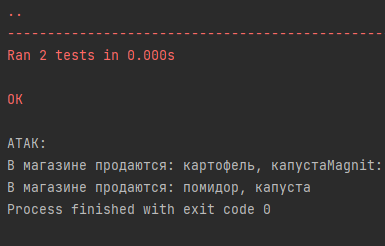
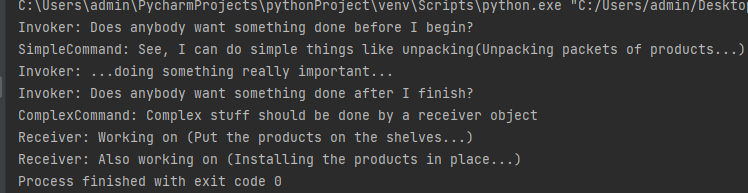
**testing.feature**

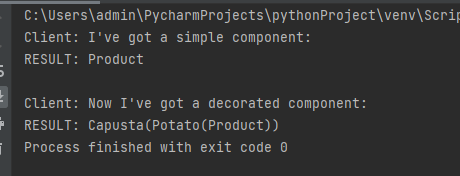
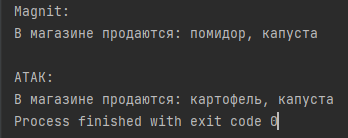
Feature: Test  
 Scenario: Test Builder  
 Given Product\_Builder  
 When test\_ATAK\_builder return OK  
 And test\_Magnit\_builder return OK  
 Then Good job

**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 Product\_Builder\_Test(unittest.TestCase):  
 @patch.object(builder.Product\_Builder(), 'product')  
 def test\_potato(self, mock\_chair):  
 mock\_potato.return\_value = None  
 self.assertEqual(Product\_Builder().potato(), None)

**Результаты программы:**

**** ****

**** ****