|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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»**

Курс «Разработка интернет-приложений»

Отчет по лабораторной работе №4

Выполнила:

студент группы ИУ5-53Б

Латыпова К.Н.

Москва, 2020 г.

1. **Задание**

1)Необходимо для произвольной предметной области реализовать три шаблона проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать [следующий каталог.](https://refactoring.guru/ru/design-patterns/catalog)

2)Для каждой реализации шаблона необходимо написать модульный тест. В модульных тестах необходимо применить следующие технологии:

* 1. TDD - фреймворк.
  2. BDD - фреймворк.
  3. Создание Mock-объектов.

1. **Текст программы**

**main.py:**

from abc import ABC, abstractmethod, abstractproperty  
  
  
class Box(ABC):  
  
 @abstractproperty  
 def box(self):  
  
 pass  
  
 @abstractmethod  
 def add\_pomade(self):  
 pass  
  
 @abstractmethod  
 def add\_ink(self):  
 pass  
  
 @abstractmethod  
 def add\_eyeshadow(self):  
 pass  
  
 @abstractmethod  
 def add\_blush(self):  
 pass  
  
class Box1(Box):  
 *"""Конкретный строитель, строящий бокс первого типа."""* def \_\_init\_\_(self):  
 self.reset()  
  
 def reset(self):  
 self.\_box = Boxx()  
  
 @property  
 def box(self):  
 box = self.\_box  
 return box  
  
 def add\_pomade(self):  
 self.\_box.add("Джинсы", 2200)  
  
 def add\_ink(self):  
 self.\_box.add("Футболка", 900)  
  
 def add\_eyeshadow(self):  
 self.\_box.add("Кроссовки \"Зима\"", 6450)  
  
 def add\_blush(self):  
 self.\_box.add("Куртка", 7800)  
  
 def add\_all(self):  
 self.add\_pomade()  
 self.add\_ink()  
 self.add\_eyeshadow()  
 self.add\_blush()  
  
  
class Box2(Box):  
 *"""Конкретный строитель, строящий бокс второго типа."""* def \_\_init\_\_(self):  
 self.reset()  
  
 def reset(self):  
 self.\_box = Boxx()  
  
 @property  
 def box(self):  
 box = self.\_box  
 return box  
  
 def add\_pomade(self):  
 self.\_box.add("Джинсы", 1200)  
  
 def add\_ink(self):  
 self.\_box.add("Футболка", 1050)  
  
 def add\_eyeshadow(self):  
 self.\_box.add("Кроссовки \"Лето\"", 4999)  
  
 def add\_blush(self):  
 self.\_box.add("Куртка", 6560)  
  
 def add\_all(self):  
 self.add\_pomade()  
 self.add\_ink()  
 self.add\_eyeshadow()  
 self.add\_blush()  
  
  
class Boxx():  
 def \_\_init\_\_(self):  
 self.box = []  
 self.sum = 0  
  
 def add(self, dish, price):  
 self.box.append(dish)  
 self.sum += price  
  
 def list\_box(self):  
 return f"{', '.join(self.box)}"  
  
 def get\_sum(self):  
 return self.sum  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('Заказ №1 ')  
 order = Box1()  
 order.add\_pomade()  
 order.add\_ink()  
 order.add\_blush()  
 print(order.box.list\_box())  
  
 print('\nЗаказ №2 ')  
 order.reset()  
 order.add\_ink()  
 order.add\_eyeshadow()  
 print(order.box.list\_box())  
  
 print('\nЗаказ №3 ')  
 order = Box2()  
 order.add\_all()  
 print(order.box.list\_box())

**main1.py:**

from abc import ABC, abstractmethod  
from main import Box1, Box2  
  
  
class Component(ABC):  
 *"""  
 Базовый класс Компонент объявляет общие операции как для простых, так и для  
 сложных объектов структуры.  
 """* @property  
 def parent(self):  
 return self.\_parent  
  
 @parent.setter  
 def parent(self, parent):  
 self.\_parent = parent  
  
 def add(self, component):  
 pass  
  
 def remove(self, component):  
 pass  
  
 def is\_composite(self):  
 return False  
  
 @abstractmethod  
 def operation(self):  
 pass  
  
 @abstractmethod  
 def get\_price(self):  
 pass  
  
  
class Leaf(Component):  
 *"""Конечный объект, не имеющий вложенных."""* def \_\_init\_\_(self, value, price):  
 self.\_value = value  
 self.\_price = price  
  
 def operation(self):  
 return self.\_value  
  
 def get\_price(self):  
 return self.\_price  
  
  
class Composite(Component):  
 *"""Объект, имеющий вложенные объекты."""* def \_\_init\_\_(self, name):  
 self.\_children = []  
 self.\_name = name  
  
 def add(self, component):  
 self.\_children.append(component)  
 component.parent = self  
  
 def remove(self, component):  
 self.\_children.remove(component)  
 component.parent = None  
  
 def is\_composite(self):  
 return True  
  
 def operation(self):  
 results = []  
 for child in self.\_children:  
 results.append(child.operation())  
 return self.\_name+f"({'+'.join(results)})"  
  
 def get\_price(self):  
 count = 0  
 for child in self.\_children:  
 count += child.get\_price()  
 return count  
  
 def accept(self, visitor1):  
 pass  
  
  
def client\_code(component):  
 print(f"Box: {component.operation()}")  
 print(f'Общая стоимость: {component.get\_price()}', end='\n\n')  
  
  
if \_\_name\_\_=='\_\_main\_\_':  
 catalog = Composite('Каталог')  
 blush = Composite('Кроссовки')  
 blush.add(Leaf('Зима', 6450))  
 blush.add(Leaf('Лето', 4999))  
 boxes = Composite('Боксы')  
 box1 = Box1()  
 box1.add\_all()  
 box2 = Box2()  
 box2.add\_all()  
 boxes.add(Leaf(box1.box.list\_box(), box1.box.get\_sum()))  
 boxes.add(Leaf(box2.box.list\_box(), box2.box.get\_sum()))  
 paper = Leaf('Носки', 150)  
  
 catalog.add(blush)  
 catalog.add(boxes)  
 catalog.add(paper)  
  
 client\_code(catalog)  
 client\_code(boxes)  
 client\_code(paper)  
 client\_code(blush)

**main2.py:**

from main1 import Composite, Component, Leaf, client\_code  
from main import Box1, Box2  
from abc import ABC, abstractmethod  
  
  
class ComponentNew(Component):  
 *"""  
 Интерфейс Компонента объявляет метод accept, который в качестве аргумента  
 может получать любой объект, реализующий интерфейс посетителя.  
 """* @abstractmethod  
 def accept(self, visitor):  
 pass  
  
  
class CompositeNew(Composite, ComponentNew):  
 def accept(self, visitor):  
 visitor.visit\_component(self)  
  
  
class Visitor(ABC):  
 @abstractmethod  
 def visit\_component(self, element):  
 pass  
  
  
class Visitor1(Visitor):  
 def visit\_component(self, element):  
 print('Стоимость: {}'.format(element.get\_price()))  
  
  
class Visitor2(Visitor):  
 def visit\_component(self, element):  
 client\_code(element)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 catalog = CompositeNew('Каталог')  
 blush = Composite('Кроссовки')  
 blush.add(Leaf('Зима', 6450))  
 blush.add(Leaf('Лето', 4999))  
 boxes = Composite('Боксы')  
 box1 = Box1()  
 box1.add\_all()  
 box2 = Box2()  
 box2.add\_all()  
 boxes.add(Leaf(box1.box.list\_box(), box1.box.get\_sum()))  
 boxes.add(Leaf(box2.box.list\_box(), box2.box.get\_sum()))  
 paper = Leaf('Носки', 150)  
  
 catalog.add(blush)  
 catalog.add(boxes)  
 catalog.add(paper)  
  
 visitor1 = Visitor1()  
 visitor2 = Visitor2()  
 print("Первый посетитель:")  
 catalog.accept(visitor1)  
 print("\nВторой посетитель:")  
 catalog.accept(visitor2)

**TDD.py:**

from main2 import \*  
import unittest  
  
  
class SummaTest(unittest.TestCase):  
 def test\_summa\_menu(self):  
 catalog = Composite('Каталог')  
 blush = Composite('Кроссовки')  
 blush.add(Leaf('Зима', 6450))  
 blush.add(Leaf('Лето', 4999))  
 boxes = Composite('Боксы')  
 box1 = Box1()  
 box1.add\_all()  
 box2 = Box2()  
 box2.add\_all()  
 boxes.add(Leaf(box1.box.list\_box(), box1.box.get\_sum()))  
 boxes.add(Leaf(box2.box.list\_box(), box2.box.get\_sum()))  
 paper = Leaf('Носки', 150)  
  
 catalog.add(blush)  
 catalog.add(boxes)  
 catalog.add(paper)  
  
 visitor1 = Visitor1()  
 self.assertEqual(catalog.accept(visitor1), 'visitor\_for\_composite\_new', "Should be 'visitor\_for\_composite\_new'")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

**BDD.py:**

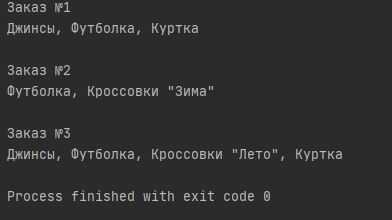
from radish import given, when, then  
  
  
@given("I have the component {component1: g}")  
def have\_component(step, component1):  
 step.context.component1 = component1  
  
  
@when("I get price from them")  
def get\_price\_component(step):  
 step.context.result = step.context.component1.get\_price()  
  
  
@then("I expect the result to be {result: g}")  
def expect\_result(step, result):  
 assert step.context.result == result

**unitest.py:**

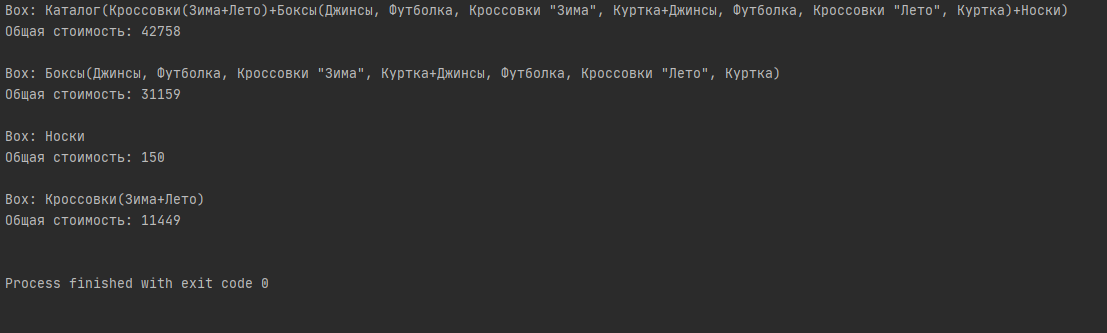
from main import Box1, Box2  
import unittest  
  
  
class SummaTest(unittest.TestCase):  
 def test\_sum\_Box1(self):  
 order = Box1()  
 order.add\_all()  
 self.assertEqual(order.box.get\_sum(), 17350, "Should be 17350")  
  
 def test\_sum\_Box2(self):  
 order = Box2()  
 order.add\_all()  
 self.assertEqual(order.box.get\_sum(), 13809, "Should be 13809")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

1. **Экранные формы с примерами выполнения программы**

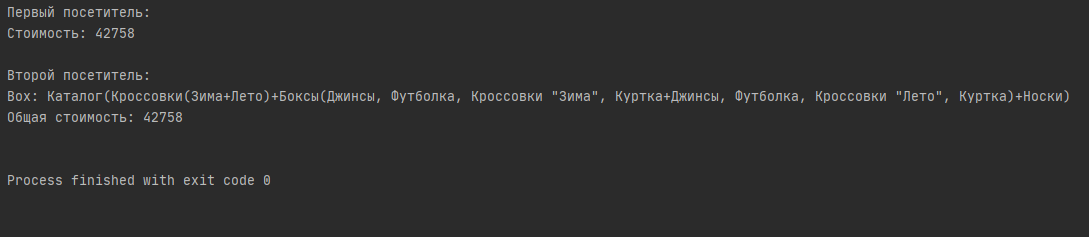
**main.py:**



**main1.py:**



**main2.py:**



**unitest.py:**

