МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Н. Э. Баумана

ЛАБОРАТОРНАЯ РАБОТА №4

Тема:

«Шаблоны проектирования и модульное тестирование в Python»

по учебной дисциплине

«Разработка интернет-приложений»

Группа: *ИУ5-52Б*

Студент: *Кобяк А.В.*

Преподаватель: *Гапанюк Ю. Е.*

Москва, 2020

Задание работы

Цель: изучение реализации шаблонов проектирования и возможностей модульного тестирования в языке Python.

Задание:
Необходимо для произвольной предметной области реализовать три шаблона проектирования: один порождающий, один структурный и один поведенческий.

В качестве справочника шаблонов можно использовать следующий каталог.

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

- TDD - фреймворк.
- BDD - фреймворк.
- Создание Mock-объектов.

Код

Порождающий паттерн – Фабрика + Абстрактная фабрика

Предположим у нас есть мебель – стол и стул. Создаем ее по трем факторам – материал -+ стиль + применение. Материал и стиль задаются условиями пользователем с помощью функций.

```
# порождающий паттерн проектирования
# абстрактная фабрика или просто фабрика
# короче что-то между
# предметная область: мебель
from abc import ABC, abstractmethod


def define_mat(platform):
    if platform == "Деревянный":
        return "дерево"
    elif platform == "Металлический":
        return "металл"
    elif platform == "Плетеный":
        return "бамбук"


def define_stil(stilchik):
    if stilchik == "Современный":
        return "современном"
    elif stilchik == "Классический":
        return "классическом"


# абстрактный класс материала
class Material(ABC):

    @abstractmethod
    def paint(self, mat):
        pass


# абстрактный класс стиля
class Style(ABC):

    @abstractmethod
    def paint(self, stil):
        pass
```

```python
# абстрактный класс того, что делают с мебелью
class Metod(ABC):

    @abstractmethod
    def paint(self):
        pass


# Абстрактная фабрика
class FurnFactory(ABC):

    @abstractmethod
    def create_material(self):
        pass

    @abstractmethod
    def create_style(self):
        pass

    @abstractmethod
    def create_metod(self):
        pass


# класс материала для стола
class TableMaterial(Material):

    def paint(self, mat):
        return f"Создание стола из материала {mat}"


# класс материала для стула
class ChairMaterial(Material):

    def paint(self, mat):
        return f"Создание стула из материала {mat}"


# класс стиля для стола
class TableStyle(Style):

    def paint(self, stil):
        return f"В {stil} стиле"


# класс стиля для стула
class ChairStyle(Style):
```

```
    def paint(self, stil):
        return f"В {stil} стиле"


# класс применения для стола
class TableMetod(Metod):

    def paint(self):
        return "На нём едят"


# класс применения для стула
class ChairMetod(Metod):

    def paint(self):
        return "На нём сидят"


# фабрика для стола
class TableFactory(FurnFactory):

    def create_material(self):
        return TableMaterial()

    def create_style(self):
        return TableStyle()

    def create_metod(self):
        return TableMetod()


# фабрика для стула
class ChairFactory(FurnFactory):

    def create_material(self):
        return ChairMaterial()

    def create_style(self):
        return ChairStyle()

    def create_metod(self):
        return ChairMetod()


# клиентский код
def client_code(factory):
    material = factory.create_material()
    style = factory.create_style()
```

```
    kak = factory.create_metod()

    print(material.paint(define_mat("Деревянный")))
    print(style.paint(define_stil("Современный")))
    print(kak.paint())
    print("**********************************")
    print(material.paint(define_mat("Металлический")))
    print(style.paint(define_stil("Современный")))
    print(kak.paint())
    print("**********************************")
    print(material.paint(define_mat("Плетеный")))
    print(style.paint(define_stil("Классический")))
    print(kak.paint())
    print("**********************************")


if __name__ == "__main__":
    print("Соберем мебель - стол")
    print("================================")
    client_code(TableFactory())

    print('\n')

    print("Соберем мебель - стул")
    print("================================")
    client_code(ChairFactory())

    print('\n')
```

Структурный паттерн – Адаптер

Предположим мы должны мебель упаковать – есть у нас мебель в обычном состоянии – характеризуется размером. Есть мебель упакованная – характеризуется количеством стопок, которые занимают детали + объем коробки. Адаптер нужен для того, чтобы преобразовать мебель из обычного состояния – в сложенное, чтобы выполнить проверку, поместится ли мебель в заданную коробку или нет.

```
# структурный паттерн - АДАПТЕР
# область - упаковка той же самой мебели

# Адаптер - связь между мебелью в упакованном состоянии и той,
# которую только предстоит упаковать
```

```
class Box:

    def __init__(self, count, newsize):
        self.count = count
        self.newsize = newsize

    def get_count(self):
        return self.count

    def get_newsize(self):
        return self.newsize

    def fit(self, packed):
        if (self.count >= packed.get_count()) & (self.newsize >= packed.get_newsi
ze()):
            return f"Поместится " \
                   f"\n"\
                   f"Высота БЛОКА {packed.get_count()}, объём {packed.get_newsize
()}" \
                   f"\n"\
                   f"Высота БОКСА {self.get_count()}, объём {self.get_newsize()}"
        else:
            return f"Не поместится " \
                   f"\n"\
                   f"Высота БЛОКА {packed.get_count()}, объём {packed.get_newsize
()}" \
                   f"\n"\
                   f"Высота БОКСА {self.get_count()}, объём {self.get_newsize()}"


class Furniture:

    def __init__(self, size):
        self.size = size

    def get_size(self):
        return self.size


class Packed:

    def __init__(self, count, newsize):
        self.count = count
        self.newsize = newsize

    def get_count(self):
        return self.count
```

```
    def get_newsize(self):
        return self.newsize


class Adapter(Packed):
    def __init__(self, furn):
        self.furn = furn

    def get_count(self):
        if self.furn.get_size() % 2 == 0:
            return self.furn.get_size() // 2
        else:
            return (self.furn.get_size() // 2) + 1

    def get_newsize(self):
        if self.furn.get_size() % 2 == 0:
            return (self.furn.get_size() // 2) * 3 * 6
        else:
            return ((self.furn.get_size() // 2) + 1) * 3 * 6


def client_code():
    box = Box(3, 60)
    packed1 = Packed(3, 54)
    packed2 = Packed(4, 100)
    furn1 = Furniture(5)
    furn2 = Furniture(7)

    print("Проверим упакованную")
    print("====================================================")
    print(box.fit(packed1))
    print("-----")
    print(box.fit(packed2))
    print("====================================================")

    print("Проверим неупакованную")
    print("====================================================")
    furn1_adapted = Adapter(furn1)
    furn2_adapted = Adapter(furn2)
    print(box.fit(furn1_adapted))
    print("-----")
    print(box.fit(furn2_adapted))


if __name__ == "__main__":
    client_code()
```

Предположим, мы компания, которая доставляет мебель. У нас есть три этапа – упаковка + доставка + сборка на месте. Вот наши состояния. Перехожим именно в таком порядке, не иначе. Нельзя же упаковать уже доставленный товар или собрать еще неупакованный.

```
# поведенческий паттерн - СОСТОЯНИЕ
# область - упаковка - доставка - сборка той же самой мебели

# State - абстрактный класс состояний(в упаковке, в доставке, в сборке)
# Context - Класс нашей мебели
# handle(1, 2, 3) - функции перехода между состояниями
# upakovka, doctavka, sborka - функции, обозначающие процесс в новом сост-ии
from abc import ABC, abstractmethod


class Context:
    """
    Контекст определяет интерфейс, представляющий интерес для клиентов. Он также
    хранит ссылку на экземпляр подкласса Состояния, который отображает текущее
    состояние Контекста.
    """

    _state = None
    """
    Ссылка на текущее состояние Контекста.
    """

    def __init__(self, state) -> None:
        self.transition_to(state)

    def transition_to(self, state):
        """
        Контекст позволяет изменять объект Состояния во время выполнения.
        """

#         print(f"Переход к {type(state).__name__}")
        self._state = state
        self._state.context = self

    """
    Контекст делегирует часть своего поведения текущему объекту Состояния.
    """

    def upakovka(self):
        self._state.handle1()
```

```
    def dostavka(self):
        self._state.handle2()

    def sborka(self):
        self._state.handle3()


class State(ABC):
    """
    Базовый класс Состояния объявляет методы, которые должны реализовать все
    Конкретные Состояния, а также предоставляет обратную ссылку на объект
    Контекст, связанный с Состоянием. Эта обратная ссылка может использоваться
    Состояниями для передачи Контекста другому Состоянию.
    """

    @property
    def context(self) -> Context:
        return self._context

    @context.setter
    def context(self, context: Context) -> None:
        self._context = context

    @abstractmethod
    def handle1(self) -> None:
        pass

    @abstractmethod
    def handle2(self) -> None:
        pass

    @abstractmethod
    def handle3(self) -> None:
        pass

"""
Конкретные Состояния реализуют различные модели поведения, связанные с
состоянием Контекста.
"""


class Upakovka(State):
    def handle1(self) -> None:
        print("Началась упаковка...")
        print("Упаковка кончилась, перехожу к доставке")
        self.context.transition_to(Dostavka())
```

```
    def handle2(self) -> None:
        print("ОТКАЗАНО => Ещё не упаковано - доставка невозможна")

    def handle3(self) -> None:
        print("ОТКАЗАНО => Ещё не упаковано и не доставлено - сборка невозможна")


class Dostavka(State):
    def handle1(self) -> None:
        print("ПРОТИВОРЕЧИЕ => Товар уже упакован и доставляется")

    def handle2(self) -> None:
        print("Товар доставляется...")
        print("Товар доставлен, перехожу к сборке")
        self.context.transition_to(Sborka())

    def handle3(self) -> None:
        print("ОТКАЗАНО => Ещё не доставлено - сборка невозможна")


class Sborka(State):
    def handle3(self) -> None:
        print("Мебель собирают...")
        print("Мебель собрали. Ура!")
        print("Приходите к нам снова")

    def handle2(self) -> None:
        print("ПРОТИВОРЕЧИЕ => Товар уже доставлен")

    def handle1(self) -> None:
        print("ПРОТИВОРЕЧИЕ => Товар уже упакован")


if __name__ == "__main__":
    # Клиентский код.
    print("Пройдем весь процесс сначала")
    print("================================================================")
    context = Context(Upakovka())
    context.upakovka()
    context.dostavka()
    context.sborka()
    print("================================================================")
    print("Попробуем начать с доставки")
    print("================================================================")
    context = Context(Dostavka())
    context.dostavka()
    context.sborka()
    print("================================================================")
```

```
    print("Теперь по противоречиям: попробем доставить неупакованный товар")
    print("================================================================")
    context = Context(Upakovka())
    context.dostavka()
    print("================================================================")
    print("Попробем собрать недоставленный")
    print("================================================================")
    context = Context(Dostavka())
    context.sborka()
    print("================================================================")
    print("Попробуем упаковать собираемый")
    print("================================================================")
    context = Context(Sborka())
    context.upakovka()
    context.dostavka()
```

Тесты

1

```
import unittest
from unittest import TestCase
from unittest.mock import patch
from first import TableFactory
from first import ChairFactory
from third import Context
from third import Upakovka, Dostavka, Sborka


class factoryTestCase(TestCase):
#   def start(self):
#        factory = TableFactory()

    @patch('first.define_mat', return_value="дерево")
    def test_mat(self, define_mat):
        factory = TableFactory()
        material = factory.create_material()
        self.assertEqual("Создание стола из материала дерево", material.paint(def
ine_mat("platform")))

    @patch('first.define_mat', return_value="металл")
    def test_mat(self, define_mat):
        factory = ChairFactory()
        material = factory.create_material()
        self.assertEqual("Создание стула из материала металл", material.paint(def
ine_mat("platform")))

    @patch('first.define_stil', return_value="современном")
```

```
    def test_mat(self, define_stil):
        factory = TableFactory()
        style = factory.create_style()
        self.assertEqual("В современном стиле", style.paint(define_stil("stilchik
")))

    @patch('first.define_stil', return_value="классическом")
    def test_mat(self, define_stil):
        factory = ChairFactory()
        style = factory.create_style()
        self.assertEqual("В классическом стиле", style.paint(define_stil("stilchi
k")))


class ThirdTestOne(TestCase):

#    def setUp(self):
#        self.Context = Context()

    def firstTest(self):
        context = Context(Upakovka())
        self.assertEqual("Началась упаковка...\nУпаковка кончилась, перехожу к до
ставке", context.upakovka())

    def secTest(self):
        context = Context(Dostavka())
        self.assertEqual("ОТКАЗАНО => Ещё не доставлено - сборка невозможна", con
text.sborka())


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

2

```
from behave import *
from second import Furniture
from second import Box
from second import Packed
from second import Adapter

@given('count of box - "{box_count}" and newsize of box - "{box_newsize}" count o
f packed - "{packed_count}" and newsize of packed - "{packed_newsize}"')
def step(context, box_count, box_newsize, packed_count, packed_newsize):
    context.box = Box(box_count, box_newsize)
    context.packed = Packed(packed_count, packed_newsize)

@then('Yes')
```

```
def step(context):
    assert context.box.fit(context.packed) ==   f"Поместится " \
                                                f"\n"\
                                                f"Высота БЛОКА {context.packed.ge
t_count()}, объём {context.packed.get_newsize()}" \
                                                f"\n"\
                                                f"Высота БОКСА {context.box.get_c
ount()}, объём {context.box.get_newsize()}"
```

```
Feature: check

    Scenario: checking packed
    Given count of box - "3" and newsize of box - "60" count of packed - "3" and newsize of packed - "54"
    Then Yes
```

3

```
import unittest
from unittest import TestCase
from third import Context
from third import Upakovka, Dostavka, Sborka


class ThirdTestOne(TestCase):

    def setUp(self):
        self.Context = Context()

    def firstTest(self):
        context = Context(Upakovka())
        self.assertEqual("Началась упаковка...\nУпаковка кончилась, перехожу к до
ставке", context.upakovka())

    def secTest(self):
        context = Context(Dostavka())
        self.assertEqual("ОТКАЗАНО => Ещё не доставлено - сборка невозможна", con
text.sborka())


class ThirdTestTwo(TestCase):
    def firstTest(self):
        context = Context(Dostavka())
        self.assertEqual("ОТКАЗАНО => Ещё не доставлено - сборка невозможна", con
text.sborka())


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

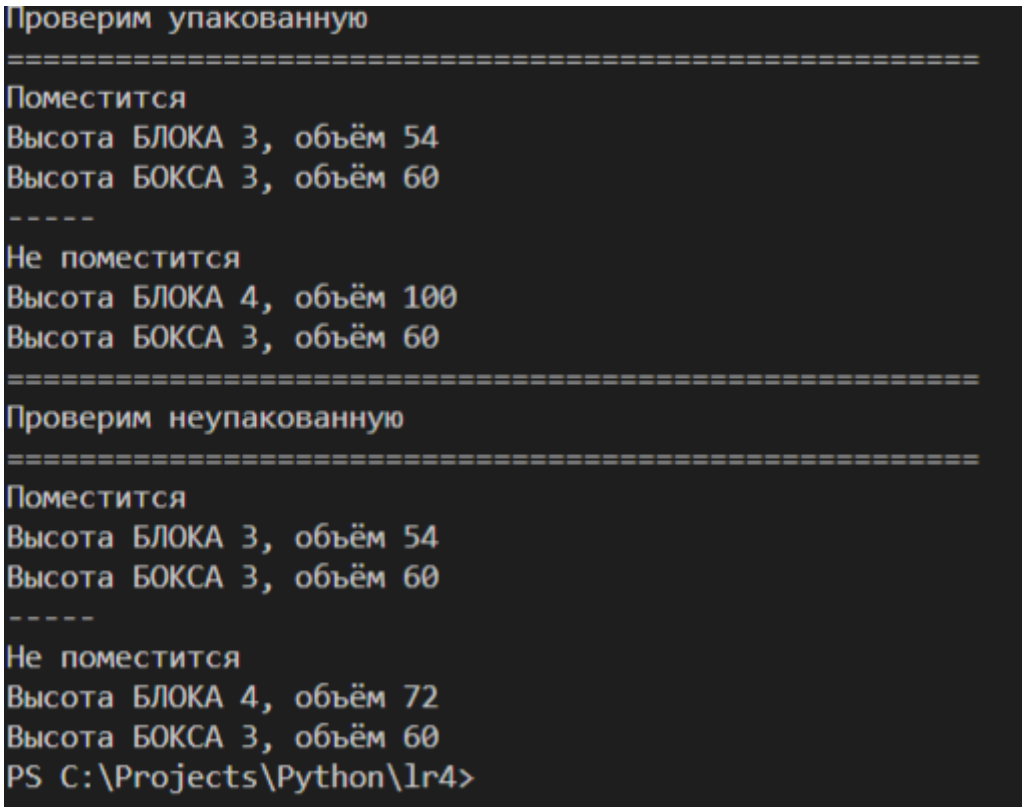
Результат

1 паттерн

```
Соберем мебель - стол
================================
Создание стола из материала дерево
В современном стиле
На нём едят
**********************************
Создание стола из материала металл
В современном стиле
На нём едят
**********************************
Создание стола из материала бамбук
В классическом стиле
На нём едят
**********************************


Соберем мебель - стул
================================
Создание стула из материала дерево
В современном стиле
На нём сидят
**********************************
Создание стула из материала металл
В современном стиле
На нём сидят
**********************************
Создание стула из материала бамбук
В классическом стиле
На нём сидят
**********************************
```

2 паттерн


Проверим упакованную
==========
Поместится
Высота БЛОКА 3, объём 54
Высота БОКСА 3, объём 60

Не поместится
Высота БЛОКА 4, объём 100
Высота БОКСА 3, объём 60
==========
Проверим неупакованную
==========
Поместится
Высота БЛОКА 3, объём 54
Высота БОКСА 3, объём 60

Не поместится
Высота БЛОКА 4, объём 72
Высота БОКСА 3, объём 60
PS C:\Projects\Python\lr4>

3 паттерн

```
================================================================
Началась упаковка...
Упаковка кончилась, перехожу к доставке
Товар доставляется...
Товар доставлен, перехожу к сборке
Мебель собирают...
Мебель собрали. Ура!
Приходите к нам снова
================================================================
Попробуем начать с доставки
================================================================
Товар доставляется...
Товар доставлен, перехожу к сборке
Мебель собирают...
Мебель собрали. Ура!
Приходите к нам снова
================================================================
Теперь по противоречиям: попробем доставить неупакованный товар
================================================================
ОТКАЗАНО => Ещё не упаковано - доставка невозможна
================================================================
Попробем собрать недоставленный
================================================================
ОТКАЗАНО => Ещё не доставлено - сборка невозможна
================================================================
Попробуем упаковать собираемый
================================================================
ПРОТИВОРЕЧИЕ => Товар уже упакован
ПРОТИВОРЕЧИЕ => Товар уже доставлен
PS C:\Projects\Python\lr4>
```