**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»

Выполнил:

студент группы ИУ5-33Б

Зорькин А.В.

Проверил:

Канев А.И.

2021 г.

# Задание:

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.

В модульных тестах необходимо применить следующие технологии:

TDD - фреймворк.

BDD - фреймворк.

Создание Mock-объектов.

# Текст программы:

## Реализация порождающего шаблона (абстрактная фабрика):

**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

shoe\_store.py:

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
  
  
class ShoesFactory(ABC):  
 @abstractmethod  
 def createsneakers(self):  
 pass  
  
 @abstractmethod  
 def createslates(self):  
 pass  
  
  
class Nikefactory(ShoesFactory):  
  
 def createsneakers(self):  
 return Nikesneakers()  
  
 def createslates(self):  
 return Nikeslates()  
  
  
class Adidasfactory(ShoesFactory):  
  
 def createsneakers(self):  
 return Adidassneakers()  
  
 def createslates(self):  
 return Adidasslates()  
  
  
class sneakers(ABC):  
 def color\_sneakers(self):  
 self.color = input()  
 return self.color  
  
 @abstractmethod  
 def receiving\_sneakers(self):  
 pass  
  
  
class Adidassneakers(sneakers):  
 def receiving\_sneakers(self):  
 return ("Create Adidassneakers")  
  
  
class Nikesneakers(sneakers):  
 def receiving\_sneakers(self):  
 return ("Create Nikesneakers")  
  
  
class slates(ABC):  
 @abstractmethod  
 def receiving\_slates(self):  
 pass  
  
 @abstractmethod  
 def style\_slates(self, collaborator: sneakers):  
 pass  
  
  
class Nikeslates(slates):  
 def receiving\_slates(self):  
 return ("Create Nikeslates")  
  
 def style\_slates(self, collaborator=sneakers):  
 return 'Цвет кроссовок был {}'.format(collaborator.color\_sneakers())  
  
  
class Adidasslates(slates):  
 def receiving\_slates(self):  
 return ("Create Adidasslates")  
  
 def style\_slates(self, collaborator=sneakers):  
 return 'Цвет кроссовок был {}'.format(collaborator.color\_sneakers())  
  
  
def client\_code(ShoesFactory):  
 sneaker = ShoesFactory.createsneakers()  
 print(sneaker.receiving\_sneakers())  
 slate = ShoesFactory.createslates()  
 print(slate.style\_slates(sneaker))  
 print(slate.receiving\_slates())  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 """  
 Клиентский код может работать с любым конкретным классом фабрики.  
 """  
 print("Тест первого производства:")  
 print(Nikefactory())  
 client\_code(Nikefactory())  
 print("\n")  
 print("Тест второго производства:")  
 client\_code(Adidasfactory())

## Реализация структурного шаблона (адаптер):

**Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

adapter.py

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
  
  
class ShoesFactory(ABC):  
 @abstractmethod  
 def createsneakers(self):  
 pass  
  
 @abstractmethod  
 def createslates(self):  
 pass  
  
  
class Nikefactory(ShoesFactory):  
  
 def createsneakers(self):  
 return Nikesneakers()  
  
 def createslates(self):  
 return Nikeslates()  
  
  
class Adidasfactory(ShoesFactory):  
  
 def createsneakers(self):  
 return Adidassneakers()  
  
 def createslates(self):  
 return Adidasslates()  
  
  
class sneakers(ABC):  
 def color\_sneakers(self):  
 self.color = input()  
 return self.color  
  
 @abstractmethod  
 def receiving\_sneakers(self):  
 pass  
  
  
class Adidassneakers(sneakers):  
 def receiving\_sneakers(self):  
 return ("Create Adidassneakers")  
  
  
class Nikesneakers(sneakers):  
 def receiving\_sneakers(self):  
 return ("Create Nikesneakers")  
  
  
class slates(ABC):  
 @abstractmethod  
 def receiving\_slates(self):  
 pass  
  
 @abstractmethod  
 def style\_slates(self, collaborator: sneakers):  
 pass  
  
  
class Nikeslates(slates):  
 def receiving\_slates(self):  
 return ("Create Nikeslates")  
  
 def style\_slates(self, collaborator=sneakers):  
 return 'Цвет кроссовок был {}'.format(collaborator.color\_sneakers())  
  
  
class Adidasslates(slates):  
 def receiving\_slates(self):  
 return ("Create Adidasslates")  
  
 def style\_slates(self, collaborator=sneakers):  
 return 'Цвет кроссовок был {}'.format(collaborator.color\_sneakers())  
  
  
class Adaptee:  
 def receiving\_sneakers\_new(self):  
 return ('new factory')  
  
  
class Adapter(Nikesneakers, Adaptee):  
 def receiving\_sneakers(self):  
 return self.receiving\_sneakers\_new()  
  
  
def client\_code():  
 sneaker = Nikefactory().createsneakers()  
 print(sneaker.receiving\_sneakers())  
 sneaker\_new = Adapter().receiving\_sneakers()  
 print(sneaker\_new)  
 sneaker\_new\_color = Adapter().color\_sneakers()  
 print(sneaker\_new\_color)  
if \_\_name\_\_ == "\_\_main\_\_":  
 client\_code()

## Реализация поведенческого шаблона (итератор):

**Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

iterator.py

from abc import ABC, abstractmethod  
  
  
class Iterator(ABC):  
 @abstractmethod  
 def \_\_init\_\_(self, collection, cursor=0):  
 pass  
  
 @abstractmethod  
 def \_\_iter\_\_(self):  
 pass  
  
 @abstractmethod  
 def \_\_next\_\_(self):  
 pass  
  
  
class JustIter:  
 def \_\_init\_\_(self, collection, cursor=0):  
 self.collection = collection  
 self.cursor = cursor  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 try:  
 self.cursor += 1  
 return self.collection[self.cursor - 1]  
 except:  
 raise StopIteration  
  
  
class EvenIter:  
 def \_\_init\_\_(self, collection, cursor=0):  
 self.collection = collection  
 self.cursor = cursor  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 try:  
 self.cursor += 1  
 if (self.collection[self.cursor - 1]) % 2 == 0:  
 return self.collection[self.cursor - 1]  
 else:  
 return 'Нечетное'  
 except:  
 raise StopIteration  
  
  
class Collection(ABC):  
 @abstractmethod  
 def \_\_init\_\_(self):  
 pass  
  
 def iterator(self):  
 pass  
  
  
class JustCollection(Collection):  
 def \_\_init\_\_(self, collection):  
 self.collection = collection  
  
 def iterator(self, cursor=0):  
 return JustIter(self.collection, cursor)  
  
  
class EvenCollection(Collection):  
 def \_\_init\_\_(self, collection):  
 self.collection = collection  
  
 def iterator(self, cursor=0):  
 return EvenIter(self.collection, cursor)  
  
  
def client\_code(collection, cursor=0):  
 for x in collection.iterator(cursor):  
 print(x)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 client\_code(JustCollection([1, 2, 3, 4, 5]), 1)  
 client\_code(EvenCollection([1, 2, 3, 4, 5]))

## Тестирование(TDD - фреймворк):

test.py

import unittest  
import sys, os  
  
sys.path.append(os.getcwd())  
from shoe\_store import \*  
  
  
class Test\_Shoe\_store(unittest.TestCase):  
 def test\_receiving\_sneakers(self):  
 self.assertEqual(Nikefactory().createsneakers().receiving\_sneakers(), 'Create Nikesneakers')  
 self.assertEqual(Adidasfactory().createsneakers().receiving\_sneakers(), 'Create Adidassneakers')  
  
 def test\_style\_slates(self):  
 self.assertEqual(Nikefactory().createslates().receiving\_slates(), 'Create Nikeslates')  
 self.assertEqual(Adidasfactory().createslates().receiving\_slates(), 'Create Adidasslates')

## Тестирование(BDD - фреймворк):

myfeature.feature

*# -- FILE: features/example.feature*Feature: Showing off behave  
  
 Scenario: Function return message about creation  
 Given Factory  
 When test\_receiving\_sneakers return OK  
 And test\_style\_slates return OK  
 Then good job

myfeaturesteps.py

from behave import \*  
  
from tests.test import \*  
  
  
@given('Factory')  
def first\_step(context):  
 context.a = Test\_Shoe\_store()  
  
  
@when('test\_receiving\_sneakers return OK')  
def check\_sneakers(context):  
 context.a.test\_receiving\_sneakers()  
  
  
@when('test\_style\_slates return OK')  
def check\_slates(context):  
 context.a.test\_style\_slates()  
  
  
@then('good job')  
def last\_step(context):  
 pass

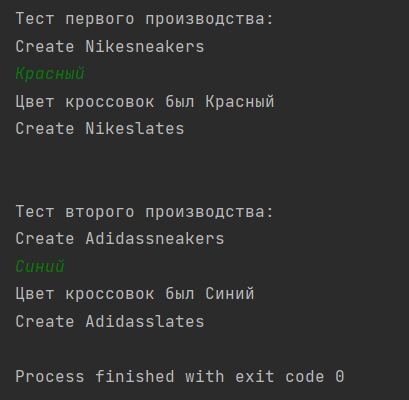
## Тестирование (Создание Mock-объектов):

mock\_test.py

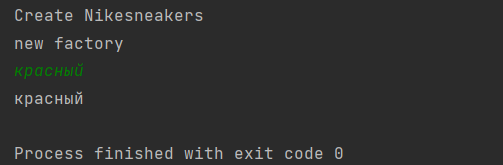
import unittest  
import sys, os  
from unittest.mock import patch, Mock  
  
import shoe\_store  
  
sys.path.append(os.getcwd())  
from shoe\_store import \*  
  
  
class Test\_shoe\_store(unittest.TestCase):  
 @patch.object(shoe\_store.sneakers, 'color\_sneakers')  
 def test\_style\_slates(self, mock\_color\_sneakers):  
 mock\_color\_sneakers.return\_value = "Create Adidasslates(replacement)"  
 self.assertEqual(Adidasfactory().createslates().style\_slates(Adidasfactory().createsneakers()),"Цвет кроссовок был Create Adidasslates(replacement)")

# Экранные формы с примерами выполнения программы:

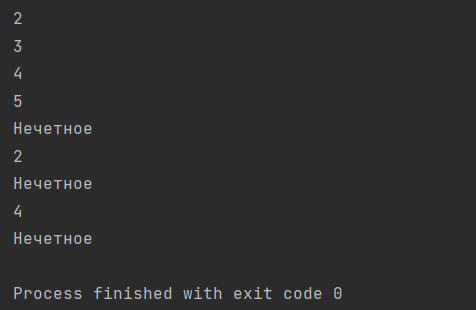
## Реализация порождающего шаблона (абстрактная фабрика):



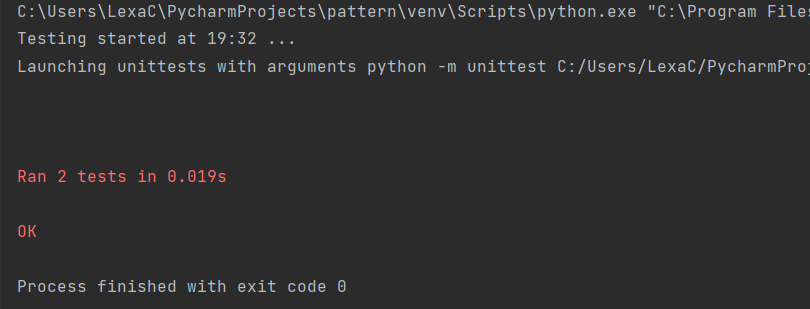
## Реализация структурного шаблона (адаптер):



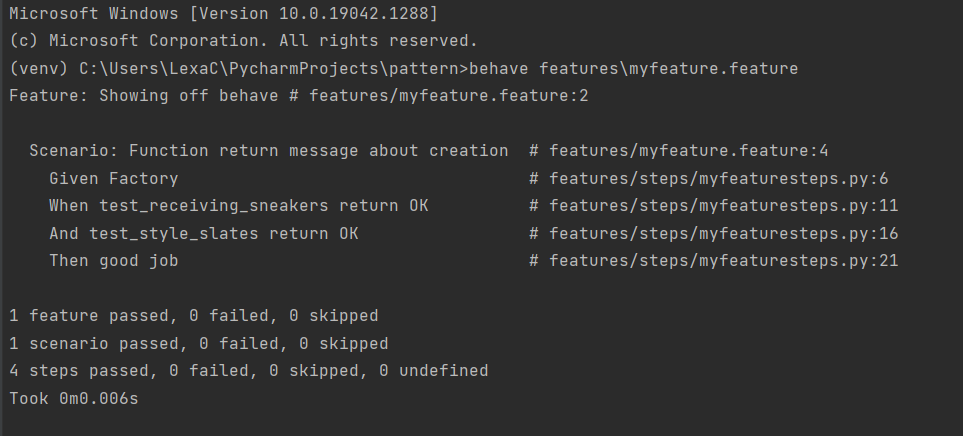
## Реализация поведенческого шаблона (итератор):



## Тестирование(TDD - фреймворк):



## Тестирование(BDD - фреймворк):



## Тестирование (Создание Mock-объектов):

