**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»**

Отчёт по лабораторной работе №4:

«Шаблоны проектирования и модульное тестирование в Python»

Выполнилa:

Студентка группы ИУ5-55Б

Е. И. Мащенко

***2020***

***Постановка задачи***

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

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

***Текст программы***

factory\_method.py (порождающий паттерн)

# порождающий паттерн проектирования

# фабричный метод

# предметная область: создание кросс-платформенных элементов интерфейса без привязывания основного кода программы к конкретным классам элементов

from abc import ABC, abstractmethod

def get\_event():

    return "closeDialog"

class Button(ABC):

    """

    Интерфейс Button объявляет операции, которые должны выполнять все

    конкретные Button.

    """

    @abstractmethod

    def render(self):

        pass

    @abstractmethod

    def onClick(self):

        pass

"""

Конкретные реализации интерфейса Button.

"""

class WindowsButton(Button):

    def render(self):

        return "Отрисовка кнопки в стиле Windows."

    def onClick(self,f):

        return "Вешаем на кнопку обработчик событий Windows:"+f

class HTMLButton(Button):

    def render(self):

        return "Возвращаем HTML-код кнопки."

    def onClick(self,f):

        return "Вешаем на кнопку обработчик события браузера:"+f

class DialogCreator(ABC):

    """

    Класс Создатель объявляет фабричный метод, который должен возвращать объект

    класса Button.

    """

    #фабричный метод

    @abstractmethod

    def createButton(self):

        pass

    def render(self, event):

        """

        Создатель содержит некоторую базовую бизнес-логику, которая основана на объектах,

        возвращаемых фабричным методом.

        """

        # Вызываем фабричный метод, чтобы получить объект-кнопку.

        okButton = self.createButton()

        # Далее, работаем с этим продуктом.

        return  okButton.onClick(event) +"\n" + okButton.render()

"""

Конкретные Создатели переопределяют фабричный метод для того, чтобы изменить тип

результирующего объекта.

"""

class WindowsDialogCreator(DialogCreator):

    """

    Сигнатура метода по-прежнему использует тип

    абстрактного продукта, хотя фактически из метода возвращается конкретный

    продукт. Таким образом, Создатель может оставаться независимым от конкретных

    классов продуктов.

    """

    def createButton(self) -> Button:

        return WindowsButton()

class WebDialogCreator(DialogCreator):

    def createButton(self) -> Button:

        return HTMLButton()

def client\_code(dialog: DialogCreator) -> None:

   print(dialog.render(get\_event()))

if \_\_name\_\_ == "\_\_main\_\_":

    print("App: Launched with the WindowsDialogCreator.")

    client\_code(WindowsDialogCreator())

    print("\n")

    print("App: Launched with the WebDialogCreator.")

    client\_code(WebDialogCreator())

adapter.py (структурный паттерн)

# структурный паттерн проектирования

# адаптер

# предметная область: преобразует один интерфейс в другой, позволяя совместить квадратные колышки и круглые отверстия

#                     Адаптер вычисляет наименьший радиус окружности, в которую можно вписать квадратный колышек,

#                     и представляет его как круглый колышек с этим радиусом.

#Адаптируемый класс

class RoundPeg:

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def get\_radius(self):

        return self.radius

#Адаптируемый класс

class SquarePeg:

    def \_\_init\_\_(self, width):

        self.width = width

    def get\_width(self):

        return self.width

#Целевой класс объявляет интерфейс, с которым может работать клиентский код

class RoundHole:

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def get\_radius(self):

        return self.radius

    def fits(self, round\_peg):

        if self.get\_radius() >= round\_peg.get\_radius():

            return f"Деталь подходит. " \

                   f"Радиус детали: {round\_peg.get\_radius()}, радиус отверстия {self.get\_radius()}"

        else:

            return f"Деталь не подходит. " \

                   f"Радиус детали: {round\_peg.get\_radius()}, радиус отверстия {self.get\_radius()}"

#Адаптер позволяет использовать квадратные колышки и круглые

#отверстия вместе.

class SquarePegAdapter (RoundPeg):

    def \_\_init\_\_(self, square\_peg):

        self.square\_peg = square\_peg

    def get\_radius(self):

        return self.square\_peg.get\_width() / 2

def client\_code():

   hole = RoundHole(5)

   rpeg = RoundPeg(5)

   print(hole.fits(rpeg) )

   small\_sqpeg = SquarePeg(5)

   large\_sqpeg = SquarePeg(12)

   #hole.fits(small\_sqpeg)

   small\_sqpeg\_adapter = SquarePegAdapter(small\_sqpeg)

   large\_sqpeg\_adapter = SquarePegAdapter(large\_sqpeg)

   print(hole.fits(small\_sqpeg\_adapter))

   print(hole.fits(large\_sqpeg\_adapter))

if \_\_name\_\_ == "\_\_main\_\_":

    client\_code()

observer.py (поведенческий паттерн)

# поведенческий паттерн проектирования

# наблюдатель

# предметная область: новостная газета делает рассылку подписчикам, при публикации очередной новости

from abc import ABC, abstractmethod

class Observer(ABC):

    # Общий интерфейс подписчиков

    @abstractmethod

    def update(self, message: str) -> None:

        pass

class EventManager(ABC):

    #базовый класс-издатель

    def \_\_init\_\_(self) -> None:

        self.observers = []     # инициализация списка наблюдателей

    def subscribe(self, observer: Observer) -> None:

       #Регистрация нового наблюдателя на подписку

        print (observer.name + " подписался(-ась) на новостные обновления")

        self.observers.append(observer)

    def unsubscribe(self, observer):

        print (observer.name + " отписался(-ась) от новостных обновлений")

        self.observers.remove(observer)

    def notify(self, message: str) -> None:

        #Передача сообщения всем наблюдателям, подписанным на газету

        for observer in self.observers:

            print(observer.update(message))

class PublisherNewspaper(EventManager):

    #Конкретный класс-издатель, в данном случае новостная газета, в которую добавляются новости

    def add\_news(self, news: str) -> None:

        #Выпуск очередной новости

        self.notify(news)

class Citizen(Observer):

    #Конкретный подписчик -гражданин, который хочет подписаться на новости

    def \_\_init\_\_(self, name: str) -> None:

        self.name = name

    def update(self, message: str):

        #Получение очередной новости

        return f'{self.name} узнал(-а) следующее: {message}'

def client\_code():

    newspaper = PublisherNewspaper()                 # создаем небольшую газету

    person\_one=Citizen('Елена')

    person\_two=Citizen('Максим')

    newspaper.subscribe(person\_one)     # добавляем двух человек, которые

    newspaper.subscribe(person\_two)  # ... ее регулярно выписывают

    # ... и публикуем очередную новость

    newspaper.add\_news('Наблюдатель - поведенческий шаблон проектирования')

    #отписываем от обновлений первого человека

    newspaper.unsubscribe(person\_one)

if \_\_name\_\_ == '\_\_main\_\_':

    client\_code()

***Тестирование***

test\_factory\_method.py (TDD + mock-объекты)

from unittest import TestCase

from unittest.mock import patch

from factory\_method import WindowsDialogCreator

from factory\_method import WebDialogCreator

class FactoryMethodTestCase(TestCase):

    # проверяем работу метода render Создателя для WindowsButton

    @patch('factory\_method.get\_event', return\_value="closeDialog")

    def test\_win\_button(self, get\_event):

        creator=WindowsDialogCreator()

        self.assertEqual("Вешаем на кнопку обработчик событий Windows:closeDialog\nОтрисовка кнопки в стиле Windows.",

                         creator.render(get\_event()))

   # проверяем работу метода render Создателя для HTMLButton

    @patch('factory\_method.get\_event', return\_value="closeDialog")

    def test\_html\_button(self, get\_event):

        creator=WebDialogCreator()

        self.assertEqual("Вешаем на кнопку обработчик события браузера:closeDialog\nВозвращаем HTML-код кнопки.",

                         creator.render(get\_event()))

test\_observer.py (TDD)

#TDD-фреймворк

import unittest

from observer import Citizen

from observer import PublisherNewspaper

class ObserverTestCase(unittest.TestCase):

    # проверка добавления нового подписчика

    def test\_subscribe(self):

        person\_subscriber = Citizen("Петя")

        newspaper = PublisherNewspaper()

        newspaper.subscribe(person\_subscriber)

        self.assertEqual(type(person\_subscriber), type(newspaper.observers[0]))

    # проверка удаления подписчика

    def test\_unsubscribe(self):

        person\_subscriber = Citizen("Петя")

        newspaper = PublisherNewspaper()

        newspaper.subscribe(person\_subscriber)

        newspaper.unsubscribe(person\_subscriber)

        self.assertEqual(0, len(newspaper.observers))

    # проверка реакции на поступление новых кроссовок людей, подписанных на кроссовки

    def test\_react\_sneakers\_subscriber(self):

        newspaper = PublisherNewspaper()

        person\_subscriber = Citizen("Петя")

        message='Наблюдатель - поведенческий шаблон проектирования'

        self.assertEqual( f'Петя узнал(-а) следующее: {message}',

                         person\_subscriber.update(message))

    if \_\_name\_\_ == '\_\_main\_\_':

        unittest.main()

steps.py (BDD)

#BDD-фреймворк

from behave import \*

from adapter import RoundPeg

from adapter import RoundHole

from adapter import SquarePeg

from adapter import SquarePegAdapter

@given('size of round peg - radius "{peg\_size}" and size of round hole - "{hole\_radius}"')

def step(context, peg\_size, hole\_radius):

    context.round\_peg = RoundPeg(int(peg\_size))

    context.hole = RoundHole(int(hole\_radius))

@given('size of square peg - width "{peg\_size}" and size of round hole - "{hole\_radius}"')

def step(context, peg\_size, hole\_radius):

    context.square\_peg = SquarePeg(int(peg\_size))

    context.hole = RoundHole(int(hole\_radius))

@then('peg and hole compatible')

def step(context):

    assert context.hole.fits(context.round\_peg) == f"Деталь подходит. " \

                                                      f"Радиус детали: {context.round\_peg.get\_radius()}, " \

                                                      f"радиус отверстия {context.hole.get\_radius()}", \

        "Тест не пройден"

@then('peg and hole incompatible')

def step(context):

    assert context.hole.fits(context.round\_peg) == f"Деталь не подходит. " \

                                                      f"Радиус детали: {context.round\_peg.get\_radius()}, " \

                                                      f"радиус отверстия {context.hole.get\_radius()}", \

        "Тест не пройден"

@then('the square peg is not comparable to the round hole')

def step(context):

    f = 0

    try:

        context.hole.fits(context.square\_peg)

    except AttributeError:

        f = 1

    finally:

        assert f == 1, "Тест не пройден"

@then('peg and hole compatible after conversion via adapter')

def step(context):

    context.adapter = SquarePegAdapter(context.square\_peg)

    assert context.hole.fits(context.adapter) == f"Деталь подходит. " \

                                                 f"Радиус детали: {context.adapter.get\_radius()}, " \

                                                 f"радиус отверстия {context.hole.get\_radius()}", \

        "Тест не пройден"

@then('peg and hole incompatible after conversion via adapter')

def step(context):

    context.adapter = SquarePegAdapter(context.square\_peg)

    assert context.hole.fits(context.adapter) == f"Деталь не подходит. " \

                                                 f"Радиус детали: {context.adapter.get\_radius()}, " \

                                                 f"радиус отверстия {context.hole.get\_radius()}", \

        "Тест не пройден"

tests\_main\_interface\_via\_adapter.feature

Feature: Compatibility check via wrapper

    Scenario: Checking a square peg of suitable size

        Given size of square peg - width "20" and size of round hole - "10"

        Then peg and hole compatible after conversion via adapter

    Scenario: Checking a square peg if unsuitable size

        Given size of square peg - width "30" and size of round hole - "10"

        Then peg and hole incompatible after conversion via adapter

tests\_main\_interface.feature

Feature: Compatibility check

    Scenario:  Checking a round peg of suitable size

        Given size of round peg - radius "10" and size of round hole - "10"

        Then peg and hole compatible

    Scenario:  Checking a round peg of unsuitable size

        Given size of round peg - radius "20" and size of round hole - "10"

        Then peg and hole incompatible

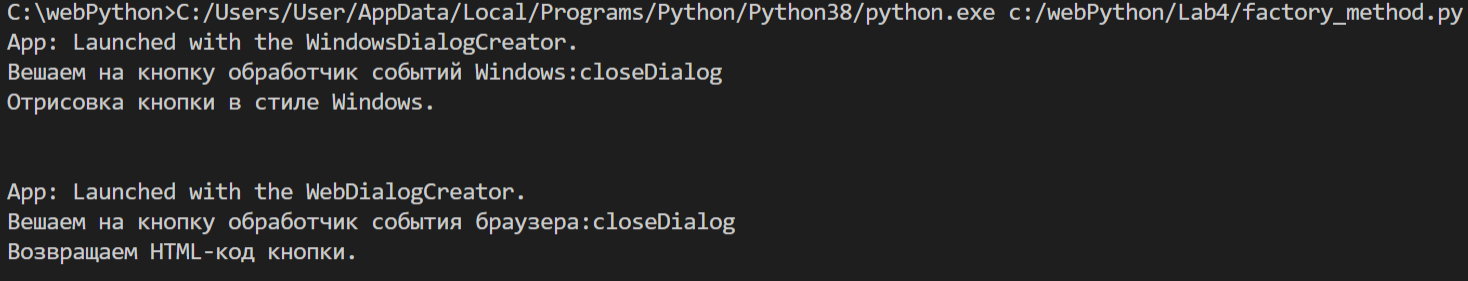
    Scenario:  Checking a square detail

        Given size of square peg - width "10" and size of round hole - "10"

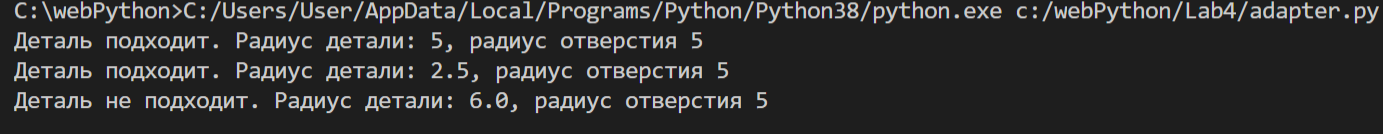
        Then the square peg is not comparable to the round hole

***Экранные формы с примерами выполнения программы***

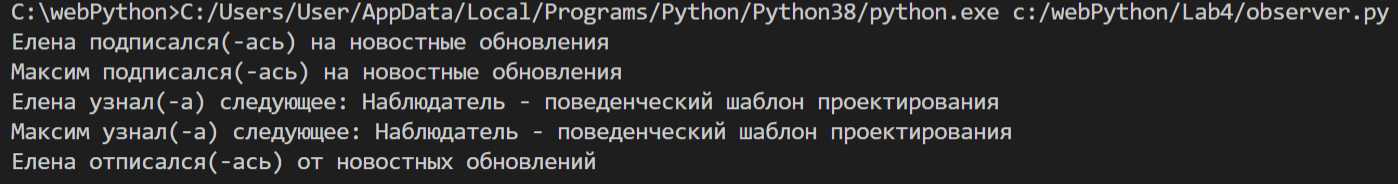
*Задача №1 (порождающий паттерн)*

**

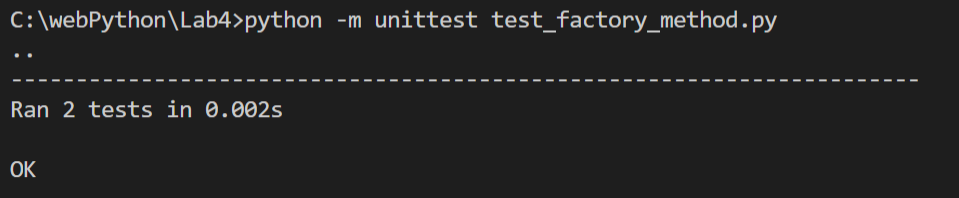
*Задача №2 (структурный паттерн)*

**

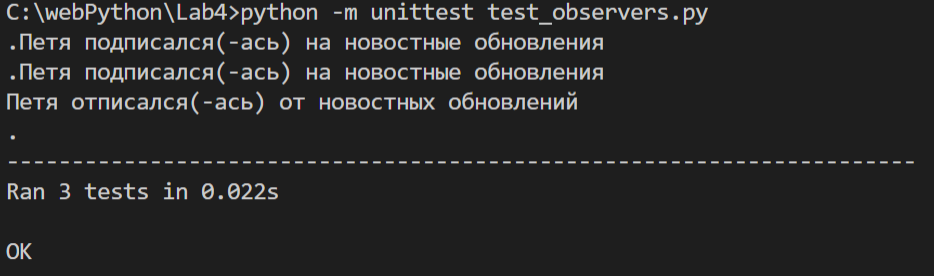
*Задача №3 (поведенческий паттерн)*

**

*Задача №4 (TDD+mock)*

**

*Задача №5 (TDD)*

**

*Задача №6 (BDD)*

**