Московский государственный технический

университет им. Н.Э. Баумана.

Факультет «Информатика и управление»

Кафедра «Системы обработки информации и управления»

Курс «Базовые компоненты интернет-технологий»

Отчет по лабораторной работе №4

«Шаблоны проектирования и модульное тестирование в Python.»

Выполнил:

студент группы ИУ5-33

Ефременко Даниил

Подпись и дата:

Проверил:

преподаватель каф. ИУ5

Канев Антон.

Подпись и дата:

Задание:

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. 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:
o TDD - фреймворк.

o BDD - фреймворк.

o Создание Mock-объектов.

Текст программы:

Файл 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 wardrobe(self) -> None:  # шкаф
        pass

    @abstractmethod
    def chair(self) -> None:  # стул
        pass

    @abstractmethod
    def bed(self) -> None:  # кровать
        pass


class Furniture_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 wardrobe(self) -> None:
        self._product.add("шкаф")

    def chair(self) -> None:
        self._product.add("стул")

    def bed(self) -> None:
        self._product.add("кровать")


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="")


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 Shatura(self) -> None:
        self.builder.chair()
        self.builder.wardrobe()

    def Lasurit(self) -> None:
        self.builder.bed()
        self.builder.chair()


if __name__ == "__main__":
    director = Director()
    builder = Furniture Builder()
    director.builder = builder

    print("Шатура: ")
    director.Shatura()
    builder.product.list_parts()

    print("\n\nЛазурит: ")
```

```
    director.Lasurit()
    builder.product.list_parts()
```

Файл decorator.py:

```
class Furniture():
    """
    Базовый интерфейс Компонента определяет поведение, которое изменяется
    декораторами.
    """
    def operation(self) -> str:
        pass


class Furniture(Furniture):
    """
       Конкретные Компоненты предоставляют реализации поведения по умолчанию.
Может
       быть несколько вариаций этих классов.
       """
    def operation(self) -> str:
        return "Furniture"


class Decorator(Furniture):
    """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: Furniture = None

    def __init__(self, component: Furniture) -> 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 Chair(Decorator):
    def operation(self) -> str:
        return f"Chair({self.component.operation()})"


class Armchair(Decorator):
    def operation(self) -> str:
        return f"Armchair({self.component.operation()})"


def show(component: Furniture) -> None:
    print(f"RESULT: {component.operation()}", end="")


if __name__ == "__main__":
    # Таким образом, клиентский код может поддерживать как простые
компоненты...
    simple = Furniture()
```

```
    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 = Chair(simple)
    decorator2 = Armchair(decorator1)
    print("Client: Now I've got a decorated component:")
    show(decorator2)
```

Файл command.py:

```
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 details of wardrobe..."))
    receiver = Receiver()
    invoker.set_on_finish(ComplexCommand(
        receiver, "Assembling of wardrobe..", "Installing the wardrobe in
place.."))

    invoker.do_something_important()
```

Файл TDD_ test.py:

```
import unittest
import sys, os
from builder import *

sys.path.append(os.getcwd())

class Furniture_Builder_Test(unittest.TestCase):
    director = Director()
    builder = Furniture_Builder()
    director.builder = builder
    def test_Lasurit(self):
       print("Лазурит: ")
       self.director.Lasurit()
       self.builder.product.list_parts()

    def test_Shatura(self):
        print("\nШатура: ")
        self.director.Shatura()
        self.builder.product.list_parts()

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

Файл testing.feature:

```
Feature: Test
    Scenario: Test Builder
      Given Furniture_Builder
      When test_shatura_builder return OK
      And test_lasurit_builder return OK
      Then Good job
```

Файл BDD_ test.py:

```
from behave import *
from TDD_test import *

@given("Furniture_Builder")
def first_step(context):
    context.a = Furniture_Builder_Test()

@when("test_shatura_builder return OK")
```

```
def test_shatura_builder(context):
    context.a.test_shatura_builder()

@when("test_lasurit_builder return OK")
def test_lasurit_builder(context):
    context.a.test_lasurit_builder()

@then("Good job")
def last_step(context):
    pass
```

Файл 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 Furniture_Builder_Test(unittest.TestCase):
    @patch.object(builder.Furniture_Builder(), 'chair')
    def test_chair(self, mock_chair):
        mock_chair.return_value = None
        self.assertEqual(Furniture_Builder().chair(), None)
```

Экранные формы с примерами выполнения программы:

buider.py

```
Шатура:
В магазине продаются: стул, шкаф

Лазурит:
В магазине продаются: кровать, стул
Process finished with exit code 0
```

decorator.py

```
Client: I've got a simple component:
RESULT: Furniture

Client: Now I've got a decorated component:
RESULT: Armchair(Chair(Furniture))
Process finished with exit code 0
```

command.py

```
Invoker: Does anybody want something done before I begin?
SimpleCommand: See, I can do simple things like unpacking(Unpacking details of wardrobe...)
Invoker: ...doing something really important...
Invoker: Does anybody want something done after I finish?
ComplexCommand: Complex stuff should be done by a receiver object
Receiver: Working on (Assembling of wardrobe...)
Receiver: Also working on (Installing the wardrobe in place...)
Process finished with exit code 0
```

Тестирование (TDD – фреймворк):

```
Ran 2 tests in 0.000s

OK
Лазурит:
В магазине продаются: кровать, стул
Шатура:
В магазине продаются: стул, шкаф
Process finished with exit code 0
```

Тестирование (BDD – фреймворк):

```
  Scenario: Test Builder                          # Features/testing.feature:2
    Given  Furniture_Builder                      # Features/steps/test_BDD.py:6
    When  test_shatura_builder return OK          # Features/steps/test_BDD.py:11
    And  test_lasurit_builder return OK           # Features/steps/test_BDD.py:16
    Then  Good job                                # Features/steps/test_BDD.py:21

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
4 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
```

Тестирование (Создание Mock-объектов):

```
Launching unittests with arguments python -m unittest



Ran 1 test in 0.002s

OK

Process finished with exit code 0
```