**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
Семенов В.А.

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. 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:**

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 telephone(self) -> None:  #телефон
        pass

    @abstractmethod
    def tablet(self) -> None:  #планшет
        pass

    @abstractmethod
    def laptop(self) -> None:  #ноутбук
        pass


class Technic_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 telephone(self) -> None:
        self._product.add("телефон")

    def tablet(self) -> None:
        self._product.add("планшет")
```

```
    def laptop(self) -> None:
        self._product.add("ноутбук")


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 Mvideo(self) -> None:
        self.builder.telephone()
        self.builder.tablet()

    def DNS(self) -> None:
        self.builder.laptop()
        self.builder.telephone()


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

    print("М.Видео: ")
    director.Mvideo()
    builder.product.list_parts()

    print("\n\nDNS: ")
    director.DNS()
    builder.product.list_parts()
```

Файл decorator.py:

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


class Technic(Technic):
    """
       Конкретные Компоненты предоставляют реализации поведения по умолчанию.
```

```
Может
        быть несколько вариаций этих классов.
        """
    def operation(self) -> str:
        return "Technic"


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

    def __init__(self, component: Technic) -> 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 Telephone(Decorator):
    def operation(self) -> str:
        return f"telephone({self.component.operation()})"


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


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


if __name__ == "__main__":
    # Таким образом, клиентский код может поддерживать простые компоненты
    simple = Technic()
    print("Client: I've got a simple component:")
    show(simple)
    print("\n")
    # и декорированные.
    #
    # Декораторы могут обёртывать не только простые
    # компоненты, но и другие декораторы.
    decorator1 = Telephone(simple)
    decorator2 = Computer(decorator1)
    print("Client: Now I've got a decorated component:")
    show(decorator2)
```

Файл command.py:

```
from __future__ import annotations
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 telephone..."))
    receiver = Receiver()
    invoker.set_on_finish(ComplexCommand(
        receiver, "Assembling of telephone..", "Installing the telephone in
place.."))

    invoker.do_something_important()
```

Файл TDD_ test.py:

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

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

class Technic_Builder_Test(unittest.TestCase):
    director = Director()
    builder = Technic_Builder()
    director.builder = builder
    def test_Mvideo(self):
       print("\nМ.Видео: ")
       self.director.Mvideo()
       self.builder.product.list_parts()

    def test_DNS(self):
        print("\nDNS: ")
        self.director.DNS()
        self.builder.product.list_parts()

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

Файл testing.feature.py:

```
Feature: Test
    Scenario: Test Builder
      Given Technic_Builder
      When test_Mvideo_builder return OK
      And test_DNS_builder return OK
      Then Good job
```

Файл BDD_ test.py:

```
from behave import *
from TDD_test import *

@given("Technic_Builder")
def first_step(context):
    context.a = Technic_Builder_Test()

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

@when("test_DNS_builder return OK")
def test_DNS_builder(context):
    context.a.test_DNS_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 Technic_Builder_Test(unittest.TestCase):
    @patch.object(builder.Technic_Builder(), 'telephone')
    def test_telephone(self, mock_telephone):
        mock_telephone.return_value = None
        self.assertEqual(Technic_Builder().telephone(), None)
```

## Экранные формы с примерами выполнения программы:

buider.py

```
М.Видео:
В магазине продаются: телефон, планшет

DNS:
В магазине продаются: ноутбук, телефон
Process finished with exit code 0
```

decorator.py

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

Client: Now I've got a decorated component:
RESULT: computer(telephone(Technic))
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 telephone)
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 telephone...)
Receiver: Also working on (Installing the telephone in place.)
Process finished with exit code 0
```

Тестирование (TDD – фреймворк):

```
Ran 2 tests in 0.014s

OK

DNS:
В магазине продаются: ноутбук, телефон
М.Видео:
В магазине продаются: телефон, планшет
Process finished with exit code 0
```

Тестирование (BDD – фреймворк):

```
  Scenario: Test Builder                          # Features/testing.feature:2
    Given Technic_Builder                         # Features/steps/test_BDD.py:6
    When test_Mvideo_builder return OK            # Features/steps/test_BDD.py:11
    And test_DNS_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):

```
Ran 1 test in 0.005s

OK

Process finished with exit code 0
```