**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: |  | Проверил: |
| студент группы ИУ5-33 |  | преподаватель каф. ИУ5 |
| Ефременко Даниил |  | Канев Антон. |
| Подпись и дата: |  | 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:**

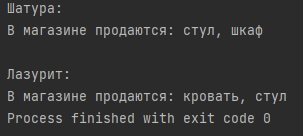
from behave import \*  
from TDD\_test import \*  
  
@given("Furniture\_Builder")  
def first\_step(context):  
 context.a = Furniture\_Builder\_Test()  
  
@when("test\_shatura\_builder return OK")  
def test\_shatura\_builder(context):  
 context.a.test\_shatura\_builder()  
  
@when("test\_lasurit\_builder return OK")  
def test\_lasurit\_builder(context):  
 context.a.test\_lasurit\_builder()  
  
@then("Good job")  
def last\_step(context):  
 pass

**Файл Mock\_test.py:**

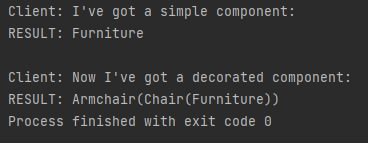
import unittest  
import sys, os  
from unittest.mock import patch, Mock  
  
import builder  
  
sys.path.append(os.getcwd())  
from builder import \*  
  
class Furniture\_Builder\_Test(unittest.TestCase):  
 @patch.object(builder.Furniture\_Builder(), 'chair')  
 def test\_chair(self, mock\_chair):  
 mock\_chair.return\_value = None  
 self.assertEqual(Furniture\_Builder().chair(), None)

**Экранные формы с примерами выполнения программы:**

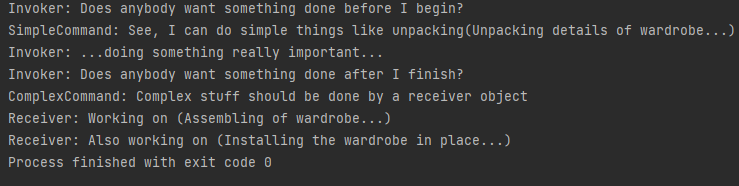
**buider.py**



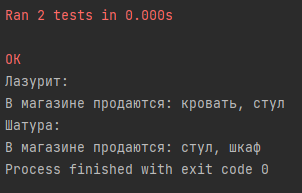
**decorator.py**



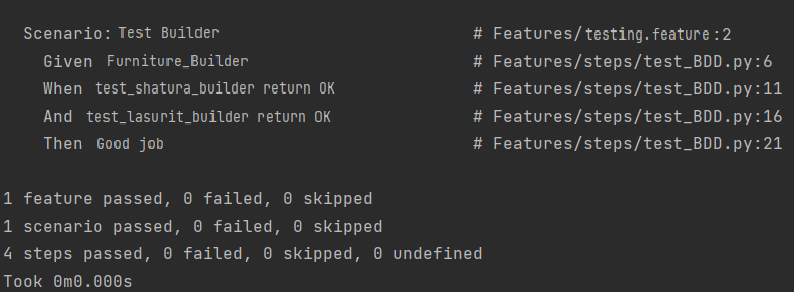
**command.py**

****

**Тестирование (TDD – фреймворк):**

****

**Тестирование (BDD – фреймворк):**

****

**Тестирование (Создание Mock-объектов):**

