**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Радиотехнический»

Кафедра «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»

Отчет по лабораторной работе №4

«**Шаблоны проектирования и модульное тестирование в Python.**»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы РТ5-31Б: |  | преподаватель кафедры ИУ5 |
| Молодцов М.В. |  | Гапанюк Ю. Е. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2024 г.

Постановка задачи

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.](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
2. В модульных тестах необходимо применить следующие технологии:
   * TDD - фреймворк.
   * BDD - фреймворк.
   * Создание Mock-объектов.

Текст программы

Файл «behavioral.py»

class Subject:  
 def init(self):  
 self.\_observers = []  
  
 def add\_observer(self, observer):  
 self.\_observers.append(observer)  
  
 def remove\_observer(self, observer):  
 self.\_observers.remove(observer)  
  
 def notify(self):  
 for observer in self.\_observers:  
 observer.update()  
  
# Наблюдатель  
class Observer:  
 def update(self):  
 pass  
  
# Конкретный наблюдатель  
class ConcreteObserver(Observer):  
 def init(self, name: str):  
 self.\_name = name  
  
 def update(self):  
 print(f"{self.\_name} received update.")

файл «creatiomal.py»

from abc import ABC, abstractmethod  
  
# Абстрактный класс продукта  
class Product(ABC):  
 @abstractmethod  
 def operation(self):  
 pass  
  
# Конкретный продукт 1  
class ConcreteProduct1(Product):  
 def operation(self):  
 return "ConcreteProduct1 operation"  
  
# Конкретный продукт 2  
class ConcreteProduct2(Product):  
 def operation(self):  
 return "ConcreteProduct2 operation"  
  
# Абстрактный создатель  
class Creator(ABC):  
 @abstractmethod  
 def factory\_method(self):  
 pass  
  
 def some\_operation(self):  
 product = self.factory\_method()  
 return product.operation()  
  
# Конкретный создатель 1  
class ConcreteCreator1(Creator):  
 def factory\_method(self):  
 return ConcreteProduct1()  
  
# Конкретный создатель 2  
class ConcreteCreator2(Creator):  
 def factory\_method(self):  
 return ConcreteProduct2()

файл «structural.py»

# Целевая интерфейсная абстракция  
class Target:  
 def request(self):  
 return "Target: The default target's behavior."  
  
# Адаптируемый класс  
class Adaptee:  
 def specific\_request(self):  
 return "Adaptee: A special behavior."  
  
# Адаптер  
class Adapter(Target):  
 def init(self, adaptee: Adaptee):  
 self.\_adaptee = adaptee  
  
 def request(self):  
 return self.\_adaptee.specific\_request()

файл «test\_behavioral.py»

import unittest  
from unittest.mock import Mock  
from design\_patterns.behavioral import Subject, ConcreteObserver  
  
  
class TestObserver(unittest.TestCase):  
  
 def test\_observer\_notification(self):  
 subject = Subject()  
 observer = Mock(spec=ConcreteObserver)  
  
 subject.add\_observer(observer)  
 subject.notify()  
  
 observer.update.assert\_called\_once()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

файл «test\_creational.py»

import unittest  
from design\_patterns.creatiomal import ConcreteCreator1, ConcreteCreator2  
  
  
class TestFactoryMethod(unittest.TestCase):  
  
 def test\_concrete\_creator1(self):  
 creator = ConcreteCreator1()  
 result = creator.some\_operation()  
 self.assertEqual(result, "ConcreteProduct1 operation")  
  
 def test\_concrete\_creator2(self):  
 creator = ConcreteCreator2()  
 result = creator.some\_operation()  
 self.assertEqual(result, "ConcreteProduct2 operation")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

файл «test\_structural.py»

import pytest  
from pytest\_bdd import scenario, given, when, then  
from design\_patterns.structural import Adapter, Target, Adaptee  
  
@pytest.fixture  
def target\_and\_adapter():  
 target = Target()  
 adaptee = Adaptee()  
 adapter = Adapter(adaptee)  
 return target, adapter  
  
@scenario('adapter.feature', 'Use adapter to change behavior')  
def test\_adapter():  
 pass  
  
@given('an adaptee with special behavior')  
def adaptee():  
 return Adaptee()  
  
@when('an adapter is used')  
def adapter(adaptee):  
 return Adapter(adaptee)  
  
@then('the behavior changes to the adaptee’s behavior')  
def behavior(adapter):  
 assert adapter.request() == "Adaptee: A special behavior."

Экранные формы с примерами выполнения программы

