**Московский государственный технический университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования» Отчет по домашнему заданию

|  |  |
| --- | --- |
| Выполнил: | Проверил: |
| студент группы ИУ5-31Б | преподаватель каф. ИУ5 |
| Филенко Александр |  |
| Подпись и дата: | Подпись и дата: |

# Постановка задачи

**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.](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.

# Результат

# Текст программы

# Модуль для реализации фабричного метода

class Product:

    def display(self):

        pass

class ConcreteProductA(Product):

    def display(self):

        return "Product A"

class ConcreteProductB(Product):

    def display(self):

        return "Product B"

class Creator:

    def factory\_method(self):

        pass

    def create\_product(self):

        return self.factory\_method().display()

class ConcreteCreatorA(Creator):

    def factory\_method(self):

        return ConcreteProductA()

class ConcreteCreatorB(Creator):

    def factory\_method(self):

        return ConcreteProductB()

# Модуль для реализации декоратора

class Component:

    def operation(self):

        pass

class ConcreteComponent(Component):

    def operation(self):

        return "Concrete Component"

class Decorator(Component):

    def \_\_init\_\_(self, component):

        self.\_component = component

    def operation(self):

        return f"Decorator({self.\_component.operation()})"

class ConcreteDecoratorA(Decorator):

    def operation(self):

        return f"ConcreteDecoratorA({super().operation()})"

class ConcreteDecoratorB(Decorator):

    def operation(self):

        return f"ConcreteDecoratorB({super().operation()})"

# Модуль для реализации стратегии

class Strategy:

    def algorithm(self):

        pass

class ConcreteStrategyA(Strategy):

    def algorithm(self):

        return "Strategy A"

class ConcreteStrategyB(Strategy):

    def algorithm(self):

        return "Strategy B"

class Context:

    def \_\_init\_\_(self, strategy):

        self.\_strategy = strategy

    def execute\_strategy(self):

        return self.\_strategy.algorithm()

# Модульные тесты с использованием unittest

import unittest

from unittest.mock import Mock

class TestPatterns(unittest.TestCase):

    def test\_factory\_method(self):

        creator\_a = ConcreteCreatorA()

        creator\_b = ConcreteCreatorB()

        self.assertEqual(creator\_a.create\_product(), "Product A")

        self.assertEqual(creator\_b.create\_product(), "Product B")

    def test\_decorator(self):

        component = ConcreteComponent()

        decorator\_a = ConcreteDecoratorA(component)

        decorator\_b = ConcreteDecoratorB(component)

        self.assertEqual(component.operation(), "Concrete Component")

        self.assertEqual(decorator\_a.operation(), "ConcreteDecoratorA(Decorator(Concrete Component))")

        self.assertEqual(decorator\_b.operation(), "ConcreteDecoratorB(Decorator(Concrete Component))")

    def test\_strategy(self):

        strategy\_a = ConcreteStrategyA()

        strategy\_b = ConcreteStrategyB()

        context\_a = Context(strategy\_a)

        context\_b = Context(strategy\_b)

        self.assertEqual(context\_a.execute\_strategy(), "Strategy A")

        self.assertEqual(context\_b.execute\_strategy(), "Strategy B")

# Модульные тесты с использованием pytest (BDD)

import pytest

@pytest.mark.parametrize("input\_value, expected\_output", [

    ("Product A", "Product A"),

    ("ConcreteDecoratorA(Decorator(Concrete Component))", "ConcreteDecoratorA(Decorator(Concrete Component))"),

    ("Strategy A", "Strategy A")

])

def test\_patterns(input\_value, expected\_output):

    mock\_obj = Mock()

    mock\_obj.operation.return\_value = input\_value

    assert mock\_obj.operation() == expected\_output

# Запуск тестов

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

    unittest.main()

    pytest.main()