**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет ИУ

Кафедра ИУ5

Курс «Основы информатики»

Отчет по лабораторной работе №\_5-6\_

«Шаблоны проектирования и модульное тестирование в Python»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-33Б: |  | преподаватель каф. \_\_\_\_ |
| Номоконов В.А |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| 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**

**lab5-6\design\_patterns\Adapter.py**

class ExternalTestService:

    def run\_quiz(self, quiz\_data):

        print(f"Running external quiz with questions: {quiz\_data}")

class LocalTestSystem:

    def start\_test(self, questions):

        print(f"Starting test with questions: {questions}")

class TestAdapter:

    def \_\_init\_\_(self, external\_service):

        self.external\_service = external\_service

    def start\_test(self, questions):

        quiz\_data = {"questions": questions}

        self.external\_service.run\_quiz(quiz\_data)

# Использование

local\_test = LocalTestSystem()

local\_test.start\_test(["Question 1", "Question 2"])

external\_service = ExternalTestService()

adapter = TestAdapter(external\_service)

adapter.start\_test(["Question A", "Question B"])

**lab5-6\design\_patterns\Observer.py**

class CourseNotifier:

    def \_\_init\_\_(self):

        self.\_observers = []

    def subscribe(self, observer):

        self.\_observers.append(observer)

    def unsubscribe(self, observer):

        self.\_observers.remove(observer)

    def notify(self, course):

        for observer in self.\_observers:

            observer.update(course)

class Student:

    def \_\_init\_\_(self, name):

        self.name = name

    def update(self, course):

        print(f"{self.name}, новый курс доступен: {course}")

# Использование

notifier = CourseNotifier()

student1 = Student("Алексей")

student2 = Student("Мария")

notifier.subscribe(student1)

notifier.subscribe(student2)

notifier.notify("Python для начинающих")

# Алексей, новый курс доступен: Python для начинающих

# Мария, новый курс доступен: Python для начинающих

**lab5-6\design\_patterns\Singleton.py**

class SettingsMeta(type):

    \_instances = {}

    def \_\_call\_\_(cls, \*args, \*\*kwargs):

        if cls not in cls.\_instances:

            cls.\_instances[cls] = super().\_\_call\_\_(\*args, \*\*kwargs)

        return cls.\_instances[cls]

class Settings(metaclass=SettingsMeta):

    pass

class PlatformSettings(Settings):

    def \_\_init\_\_(self):

        self.theme = "Light"

        self.language = "English"

    def update\_settings(self, theme: str = None, language: str = None) -> None:

        if theme:

            self.theme = theme

        if language:

            self.language = language

    def display(self) -> dict:

        return {"theme": self.theme, "language": self.language}

# Использование

settings1 = PlatformSettings()

settings2 = PlatformSettings()

settings1.update\_settings(theme="Dark", language="Spanish")

print(settings2.display())  # {'theme': 'Dark', 'language': 'Spanish'}

print(settings1 is settings2)  # True

**lab5-6\tests\features\steps\manage\_courses(BDD).py**

from behave import given, when, then

from course\_manager import CourseManager

@given("I have an empty course list")

def step\_given\_empty\_course\_list(context):

    context.manager = CourseManager()

@when('I add a course titled "{title}" in "{category}" category')

def step\_when\_add\_course(context, title, category):

    context.manager.add\_course(title, category)

@then("I should see the course in the list")

def step\_then\_see\_course\_in\_list(context):

    courses = context.manager.list\_courses()

    assert len(courses) == 1

    assert courses[0]["title"] == "Python Basics"

    assert courses[0]["category"] == "Programming"

@given('I have a course titled "{title}" in "{category}" category')

def step\_given\_existing\_course(context, title, category):

    context.manager = CourseManager()

    context.manager.add\_course(title, category)

@when('I add a course titled "{title}" in "{category}" category')

def step\_when\_add\_duplicate\_course(context, title, category):

    context.manager.add\_course(title, category)

**lab5-6\tests\features\duplicate\_courses.feature**

Feature: Duplicate courses

Scenario: Prevent adding duplicate courses

Given I have a course titled "Python Basics" in "Programming" category

When I add a course titled "Python Basics" in "Programming" category

Then I should not see a duplicate coursedef test\_1():

    return 1

**lab5-6\tests\features\manage\_courses.feature**

Feature: Course management

Scenario: Add a course to the system

Given I have an empty course list

When I add a course titled "Python Basics" in "Programming" category

Then I should see the course in the list

**lab5-6\tests\course\_manager.py**

class Course:

    def \_\_init\_\_(self, title, category):

        self.title = title

        self.category = category

class CourseManager:

    def \_\_init\_\_(self):

        self.courses = []

    def add\_course(self, title, category):

        course = Course(title, category)

        self.courses.append(course)

        return course

    def list\_courses(self):

        return [{"title": c.title, "category": c.category} for c in self.courses]

**lab5-6\tests\test\_course\_manager(TDD).py**

import pytest

from course\_manager import CourseManager

@pytest.fixture

def course\_manager():

    """Фикстура для инициализации CourseManager с тестовыми данными."""

    manager = CourseManager()

    manager.add\_course("Python Basics", "Programming")

    manager.add\_course("Data Science", "Data Analysis")

    return manager

def test\_add\_course(course\_manager):

    """Тест добавления нового курса."""

    new\_course = course\_manager.add\_course("Machine Learning", "AI")

    assert new\_course.title == "Machine Learning"

    assert new\_course.category == "AI"

    assert len(course\_manager.list\_courses()) == 3

def test\_list\_courses(course\_manager):

    """Тест получения списка курсов."""

    courses = course\_manager.list\_courses()

    assert len(courses) == 2

    assert courses[0]["title"] == "Python Basics"

    assert courses[1]["category"] == "Data Analysis"

def test\_no\_duplicate\_courses(course\_manager):

    """Тест, чтобы убедиться, что дублирующиеся курсы не добавляются."""

    course\_manager.add\_course("Python Basics", "Programming")

    courses = course\_manager.list\_courses()

    assert len(courses) == 2  # Список остается неизменным

def test\_empty\_course\_list():

    """Тест, что при пустом менеджере список курсов пуст."""

    manager = CourseManager()

    courses = manager.list\_courses()

    assert courses == []

**lab5-6\tests\test\_notifications(Mock).py**

from unittest.mock import MagicMock

from course\_manager import CourseManager

class Notifier:

    def notify(self, student, course):

        print(f"Notifying {student} about new course: {course}")

def test\_notification():

    notifier = Notifier()

    notifier.notify = MagicMock()

    course\_manager = CourseManager()

    course\_manager.add\_course("Machine Learning", "AI")

    notifier.notify("Alex", "Machine Learning")

    notifier.notify.assert\_called\_once\_with("Alex", "Machine Learning")

**Анализ результатов**









